Data Structures and Algorithms with Object-Oriented Design Patterns in C#

Bruno R. Preiss
B.A.Sc., M.A.Sc., Ph.D., P.Eng.
Software Engineer and Architect
SOMA Networks, Inc.
Toronto, Canada

● Colophon
● Dedication
● Preface
● Contents
● Introduction
● Algorithm Analysis
● Asymptotic Notation
● Foundational Data Structures
● Data Types and Abstraction
● Stacks, Queues, and Deques
● Ordered Lists and Sorted Lists
● Hashing, Hash Tables, and Scatter Tables
● Trees
● Search Trees
● Heaps and Priority Queues
● Sets, Multisets, and Partitions
● Garbage Collection and the Other Kind of Heap
● Algorithmic Patterns and Problem Solvers
Data Structures and Algorithms with Object-Oriented Design Patterns in C#

- Sorting Algorithms and Sorters
- Graphs and Graph Algorithms
- C# and Object-Oriented Programming
- Class Hierarchy Diagrams
- Character Codes
- References
- Index

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno
Colophon

Copyright © 19101 by Bruno R. Preiss.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author.

This book was prepared with LaTeX and reproduced from camera-ready copy supplied by the author. The book is typeset using the Computer Modern fonts designed by Donald E. Knuth with various additional glyphs designed by the author and implemented using METAFONT.

METAFONT is a trademark of Addison Wesley Publishing Company.

Java is a registered trademark of Sun Microsystems.

TeX is a trademark of the American Mathematical Society.

UNIX is a registered trademark of AT&T Bell Laboratories.

Microsoft is a registered trademark of Microsoft Corporation.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Dedication

To Patty

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Preface

This book was motivated by my experience in teaching the course *E&CE 250: Algorithms and Data Structures* in the Computer Engineering program at the University of Waterloo. I have observed that the advent of *object-oriented methods* and the emergence of object-oriented *design patterns* has lead to a profound change in the pedagogy of data structures and algorithms. The successful application of these techniques gives rise to a kind of cognitive unification: Ideas that are disparate and apparently unrelated seem to come together when the appropriate design patterns and abstractions are used.

This paradigm shift is both evolutionary and revolutionary. On the one hand, the knowledge base grows incrementally as programmers and researchers invent new algorithms and data structures. On the other hand, the proper use of object-oriented techniques requires a fundamental change in the way the programs are designed and implemented. Programmers who are well schooled in the procedural ways often find the leap to objects to be a difficult one.

---

- **Goals**
- **Approach**
- **Outline**
- **Suggested Course Outline**
- **Online Course Materials**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Goals

The primary goal of this book is to promote object-oriented design using C# and to illustrate the use of the emerging object-oriented design patterns. Experienced object-oriented programmers find that certain ways of doing things work best and that these ways occur over and over again. The book shows how these patterns are used to create good software designs. In particular, the following design patterns are used throughout the text: singleton, container, enumeration, adapter and visitor.

Virtually all of the data structures are presented in the context of a single, unified, polymorphic class hierarchy. This framework clearly shows the relationships between data structures and it illustrates how polymorphism and inheritance can be used effectively. In addition, algorithmic abstraction is used extensively when presenting classes of algorithms. By using algorithmic abstraction, it is possible to describe a generic algorithm without having to worry about the details of a particular concrete realization of that algorithm.

A secondary goal of the book is to present mathematical tools just in time. Analysis techniques and proofs are presented as needed and in the proper context. In the past when the topics in this book were taught at the graduate level, an author could rely on students having the needed background in mathematics. However, because the book is targeted for second- and third-year students, it is necessary to fill in the background as needed. To the extent possible without compromising correctness, the presentation fosters intuitive understanding of the concepts rather than mathematical rigor.
Approach

One cannot learn to program just by reading a book. It is a skill that must be developed by practice. Nevertheless, the best practitioners study the works of others and incorporate their observations into their own practice. I firmly believe that after learning the rudiments of program writing, students should be exposed to examples of complex, yet well-designed program artifacts so that they can learn about the designing good software.

Consequently, this book presents the various data structures and algorithms as complete C# program fragments. All the program fragments presented in this book have been extracted automatically from the source code files of working and tested programs. It has been my experience that by developing the proper abstractions, it is possible to present the concepts as fully functional programs without resorting to pseudo-code or to hand-waving.
Contents

- Colophon
- Dedication
- Preface
  - Goals
  - Approach
  - Outline
  - Suggested Course Outline
  - Online Course Materials
- Introduction
  - What This Book Is About
  - Object-Oriented Design
    - Abstraction
    - Encapsulation
  - Object Hierarchies and Design Patterns
    - Containers
    - Enumerators
    - Visitors
    - Cursors
    - Adapters
    - Singletons
  - The Features of C# You Need to Know
    - Variables
    - Value Types and Reference Types
    - Parameter Passing
    - Classes and Objects
    - Inheritance
    - Interfaces and Polymorphism
    - Other Features
  - How This Book Is Organized
    - Models and Asymptotic Analysis
    - Foundational Data Structures
    - Abstract Data Types and the Class Hierarchy
• **Data Structures**
• **Algorithms**

• **Algorithm Analysis**
  • A Detailed Model of the Computer
    • The Basic Axioms
    • A Simple Example-Arithmetic Series Summation
    • Array Subscripting Operations
    • Another Example-Horner's Rule
    • Analyzing Recursive Methods
      • Solving Recurrence Relations-Repeated Substitution
    • Yet Another Example-Finding the Largest Element of an Array
    • Average Running Times
    • About Harmonic Numbers
    • Best-Case and Worst-Case Running Times
    • The Last Axiom
  • A Simplified Model of the Computer
    • An Example-Geometric Series Summation
    • About Arithmetic Series Summation
    • Example-Geometric Series Summation Again
    • About Geometric Series Summation
    • Example-Computing Powers
    • Example-Geometric Series Summation Yet Again
  • Exercises
  • Projects

• **Asymptotic Notation**
  • An Asymptotic Upper Bound-Big Oh
    • A Simple Example
    • Big Oh Fallacies and Pitfalls
    • Properties of Big Oh
    • About Polynomials
    • About Logarithms
    • Tight Big Oh Bounds
    • More Big Oh Fallacies and Pitfalls
    • Conventions for Writing Big Oh Expressions
  • An Asymptotic Lower Bound-Omega
    • A Simple Example
    • About Polynomials Again
  • More Notation-Theta and Little Oh
Asymptotic Analysis of Algorithms
- Rules For Big Oh Analysis of Running Time
- Example-Prefix Sums
- Example-Fibonacci Numbers
- Example-Bucket Sort
- Reality Check
- Checking Your Analysis

Exercises

Projects

Foundational Data Structures

Arrays
- Extending C# Arrays
- Constructors
- Copy Method
- DynamicArray Indexers
- DynamicArray Properties
- Resizing an Array

Multi-Dimensional Arrays
- Array Subscript Calculations
- An Implementation
- Constructor
- MultiDimensionalArray Indexer
- Matrices
- Dense Matrices
- Canonical Matrix Multiplication

Singly-Linked Lists
- An Implementation
- List Elements
- LinkedList Default Constructor
- Purge Method
- LinkedList Properties
- First and Last Properties
- Prepend Method
- Append Method
- Copy Method
- Extract Method
- InsertAfter and InsertBefore Methods

Exercises
Contents

- Projects
  - Data Types and Abstraction
    - Abstract Data Types
  - Design Patterns
    - Class Hierarchy
    - C# Objects and the IComparable Interface
      - Abstract Comparable Objects
      - Comparison Operators
    - Wrappers for Value Types
    - Containers
      - Abstract Containers
    - Visitors
      - The IsDone Property
      - Abstract Visitors
      - The AbstractContainer Class ToString Method
    - Enumerable Collections and Enumerators
      - Enumerators and foreach
    - Searchable Containers
      - Abstract Searchable Containers
    - Associations
  - Exercises
  - Projects
- Stacks, Queues, and Deques
  - Stacks
    - Array Implementation
      - Fields
      - Constructor and Purge Methods
      - Push and Pop Methods, Top Property
      - Accept Method
      - GetEnumerator Method
    - Linked-List Implementation
      - Fields
      - Constructor and Purge Methods
      - Push and Pop Methods, Top Property
      - Accept Method
      - GetEnumerator Method
    - Applications
      - Evaluating Postfix Expressions
Implementation

Queues

Array Implementation

Fields
Constructor and Purge Methods
Enqueue and Dequeue Methods, Head Property

Linked-List Implementation

Fields
Constructor and Purge Methods
Enqueue and Dequeue Methods, Head Property

Applications

Implementation

Deques

Array Implementation

The "Head" Operations
The "Tail" Operations

Linked List Implementation

The "Head" Operations
The "Tail" Operations

Doubly-Linked and Circular Lists

Exercises

Projects

Ordered Lists and Sorted Lists

Ordered Lists

Array Implementation

Fields
Creating a List and Inserting Items
Finding Items in a List
Removing Items from a List
Positions of Items in a List
Finding the Position of an Item and Accessing by Position
Inserting an Item at an Arbitrary Position
Removing Arbitrary Items by Position

Linked-List Implementation

Fields
Inserting and Accessing Items in a List
Finding Items in a List
Removing Items from a List
Contents

- Positions of Items in a List
- Finding the Position of an Item and Accessing by Position
- Inserting an Item at an Arbitrary Position
- Removing Arbitrary Items by Position

- Performance Comparison: OrderedListAsArray vs. ListAsLinkedList
- Applications

  - Sorted Lists
    - Array Implementation
      - Inserting Items in a Sorted List
      - Locating Items in an Array-Binary Search
      - Finding Items in a Sorted List
      - Removing Items from a List
    - Linked-List Implementation
      - Inserting Items in a Sorted List
      - Other Operations on Sorted Lists

- Performance Comparison: SortedListAsArray vs. SortedListAsList
- Applications
  - Implementation
  - Analysis

- Exercises
- Projects

- Hashing, Hash Tables, and Scatter Tables
  - Hashing-The Basic Idea
    - Example
    - Keys and Hash Functions
      - Avoiding Collisions
      - Spreading Keys Evenly
      - Ease of Computation
  - Hashing Methods
    - Division Method
    - Middle Square Method
    - Multiplication Method
    - Fibonacci Hashing
  - Hash Function Implementations
    - Integral Keys
    - Floating-Point Keys
    - Character String Keys
    - Hashing Containers
Contents

- Using Associations
  - Hash Tables
    - Abstract Hash Tables
    - Separate Chaining
      - Implementation
      - Constructor, Length Property and Purge Methods
      - Inserting and Removing Items
      - Finding an Item
    - Average Case Analysis
  - Scatter Tables
    - Chained Scatter Table
      - Implementation
      - Constructor, Length Property, and Purge Methods
      - Inserting and Finding an Item
      - Removing Items
      - Worst-Case Running Time
    - Average Case Analysis
  - Scatter Table using Open Addressing
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
    - Implementation
      - Constructor, Length Property, and Purge Methods
      - Inserting Items
      - Finding Items
      - Removing Items
    - Average Case Analysis
  - Applications
  - Exercises
  - Projects

- Trees
  - Basics
    - Terminology
    - More Terminology
    - Alternate Representations for Trees
  - N-ary Trees
  - Binary Trees
  - Tree Traversals
Search Trees

- Basics
  - \textit{M-Way Search Trees}
  - \textit{Binary Search Trees}
- Searching a Search Tree
  - Searching an \textit{M}-way Tree
  - Searching a Binary Tree
- Average Case Analysis
  - Successful Search
  - Solving The Recurrence-Telescoping
  - Unsuccessful Search
  - Traversing a Search Tree
- Implementing Search Trees
  - Binary Search Trees
    - Fields
    - \texttt{Find} Method
    - \texttt{Min} Property
  - Inserting Items in a Binary Search Tree
    - \texttt{Insert} and \texttt{AttachKey} Methods
  - Removing Items from a Binary Search Tree
    - \texttt{Withdraw} Method
- AVL Search Trees
  - Implementing AVL Trees
    - Constructor
    - \texttt{AdjustHeight} Method, \texttt{Height} and \texttt{BalanceFactor} Properties
  - Inserting Items into an AVL Tree
    - Balancing AVL Trees
    - Single Rotations
    - Double Rotations
    - Implementation
  - Removing Items from an AVL Tree
- \textit{M-Way Search Trees}
  - Implementing \textit{M-Way Search Trees}
    - Implementation
    - Constructor and \texttt{M} Property
    - Inorder Traversal
  - Finding Items in an \textit{M-Way Search Tree}
    - Linear Search
Contents

- Fields
  - AddTree and RemoveTree
  - MinTree and Min Properties
- Merging Binomial Queues
- Putting Items into a Binomial Queue
- Removing an Item from a Binomial Queue

- Applications
  - Discrete Event Simulation
  - Implementation

- Exercises

- Projects

- Sets, Multisets, and Partitions
  - Basics
    - Implementing Sets
  - Array and Bit-Vector Sets
    - Basic Operations
    - Union, Intersection, and Difference
    - Comparing Sets
    - Bit-Vector Sets
      - Basic Operations
      - Union, Intersection, and Difference
  - Multisets
    - Array Implementation
      - Basic Operations
      - Union, Intersection, and Difference
    - Linked-List Implementation
      - Union
      - Intersection
  - Partitions
    - Representing Partitions
    - Implementing a Partition using a Forest
      - Implementation
      - Constructor
      - Find and Join Methods
    - Collapsing Find
    - Union by Size
    - Union by Height or Rank

- Applications
Contents

- Exercises
- Projects

- Garbage Collection and the Other Kind of Heap
  - What is Garbage?
    - Reduce, Reuse, Recycle
      - Reduce
      - Reuse
      - Recycle
    - Helping the Garbage Collector
  - Reference Counting Garbage Collection
    - When Objects Refer to Other Objects
    - Why Reference Counting Does Not Work
  - Mark-and-Sweep Garbage Collection
    - The Fragmentation Problem
  - Stop-and-Copy Garbage Collection
    - The Copy Algorithm
  - Mark-and-Compact Garbage Collection
    - Handles
  - Exercises
  - Projects

- Algorithmic Patterns and Problem Solvers
  - Brute-Force and Greedy Algorithms
    - Example-Counting Change
      - Brute-Force Algorithm
      - Greedy Algorithm
    - Example-0/1 Knapsack Problem
  - Backtracking Algorithms
    - Example-Balancing Scales
    - Representing the Solution Space
    - Abstract Backtracking Solvers
      - Abstract Solvers
      - Depth-First Solver
      - Breadth-First Solver
    - Branch-and-Bound Solvers
      - Depth-First, Branch-and-Bound Solver
    - Example-0/1 Knapsack Problem Again
  - Top-Down Algorithms: Divide-and-Conquer
    - Example-Binary Search
Contents

- Example-Computing Fibonacci Numbers
- Example-Merge Sorting
- Running Time of Divide-and-Conquer Algorithms
  - Case 1
  - Case 2
  - Case 3
- Summary
- Example-Matrix Multiplication
  - Bottom-Up Algorithms: Dynamic Programming
    - Example-Generalized Fibonacci Numbers
    - Example-Computing Binomial Coefficients
    - Application: Typesetting Problem
      - Example
      - Implementation
  - Randomized Algorithms
    - Generating Random Numbers
      - The Minimal Standard Random Number Generator
      - Implementation
    - Random Variables
      - A Simple Random Variable
      - Uniformly Distributed Random Variables
      - Exponentially Distributed Random Variables
    - Monte Carlo Methods
      - Example-Computing
      - Simulated Annealing
      - Example-Balancing Scales
  - Exercises
  - Projects
- Sorting Algorithms and Sorters
  - Basics
  - Sorting and Sorters
    - Abstract Sorters
    - Sorter Class Hierarchy
  - Insertion Sorting
    - Straight Insertion Sort
    - Implementation
Contents

- Average Running Time
- Binary Insertion Sort

  - Exchange Sorting
    - Bubble Sort
    - Quicksort
      - Implementation
    - Running Time Analysis
      - Worst-Case Running Time
      - Best-Case Running Time
    - Average Running Time
    - Selecting the Pivot

  - Selection Sorting
    - Straight Selection Sorting
      - Implementation
    - Sorting with a Heap
      - Implementation
    - Building the Heap
      - Running Time Analysis
      - The Sorting Phase

  - Merge Sorting
    - Implementation
    - Merging
    - Two-Way Merge Sorting
      - Running Time Analysis

  - A Lower Bound on Sorting

- Distribution Sorting
  - Bucket Sort
    - Implementation
  - Radix Sort
    - Implementation

- Performance Data
- Exercises
- Projects

- Graphs and Graph Algorithms
  - Basics
    - Directed Graphs
    - Terminology
    - More Terminology
Contents

- Directed Acyclic Graphs
- Undirected Graphs
- Terminology
- Labeled Graphs
- Representing Graphs
  - Adjacency Matrices
  - Sparse vs. Dense Graphs
  - Adjacency Lists
- Implementing Graphs
  - Vertices
    - Enumerators
  - Edges
- Graphs and Digraphs
  - Accessors and Mutators
  - Enumerators
  - Graph Traversals
- Directed Graphs
- Abstract Graphs
- Implementing Undirected Graphs
  - Using Adjacency Matrices
  - Using Adjacency Lists
- Comparison of Graph Representations
  - Space Comparison
  - Time Comparison
- Graph Traversals
  - Depth-First Traversal
    - Implementation
    - Running Time Analysis
  - Breadth-First Traversal
    - Implementation
    - Running Time Analysis
  - Topological Sort
    - Implementation
    - Running Time Analysis
  - Graph Traversal Applications:
    - Testing for Cycles and Connectedness
      - Connectedness of an Undirected Graph
      - Connectedness of a Directed Graph
Contents

- **Testing Strong Connectedness**
- **Testing for Cycles in a Directed Graph**
- **Shortest-Path Algorithms**
  - **Single-Source Shortest Path**
    - **Dijkstra's Algorithm**
    - **Data Structures for Dijkstra's Algorithm**
    - **Implementation**
    - **Running Time Analysis**
  - **All-Pairs Source Shortest Path**
    - **Floyd's Algorithm**
    - **Implementation**
    - **Running Time Analysis**
- **Minimum-Cost Spanning Trees**
  - **Constructing Spanning Trees**
  - **Minimum-Cost Spanning Trees**
    - **Prim's Algorithm**
      - **Implementation**
    - **Kruskal's Algorithm**
      - **Implementation**
      - **Running Time Analysis**
  - **Application: Critical Path Analysis**
    - **Implementation**
- **Exercises**
- **Projects**

* C# and Object-Oriented Programming
  - **Variables**
    - **Value Types**
    - **References Types**
      - **Null References**
  - **Parameter Passing**
    - **Pass By Value**
    - **Passing By Reference**
      - **The Trade-off**
    - **In and Out Parameters**
    - **Passing Reference Types**
  - **Objects and Classes**
    - **Class Members: Fields and Methods**
    - **Constructors**
Index

o
  see big oh, see little oh

γ
  see Euler's constant

Ω
  see omega

Θ
  see theta

λ
  see lambda

abstract algorithms
  Tree Traversals

abstract class
  Class Hierarchy, Class Hierarchy, Abstract Classes and Concrete

abstract data type
  Foundational Data Structures, Abstract Data Types

abstract method
  Class Hierarchy

abstract property
  Class Hierarchy

abstract solver
  Abstract Backtracking Solvers

abstract sorter
  Sorting and Sorters

access path
  Inserting Items into an

accessor
  DynamicArray Properties, DynamicArray Properties, get, set

activation record
  The Basic Axioms

activity-node graph
  Application: Critical Path Analysis

actual parameter
  Pass By Value

acyclic
Index

directed graph  
   Directed Acyclic Graphs

directory

adapter  
   PreorderInorder, and Postorder, PreorderInorder, and Postorder

directory

address  
   Abstract Data Types

adjacency lists  
   Adjacency Lists

adjacency matrix  
   Adjacency Matrices

adjacent  
   Terminology

ADT  
   see abstract data type

algorithmic abstraction  
   Algorithmic Abstraction

ancestor  
   More Terminology
   proper  
       More Terminology

and  
   UnionIntersection, and Difference

annealing  
   Simulated Annealing

annealing schedule  
   Simulated Annealing

arc  
   directed  
       Terminology
   undirected  
       Undirected Graphs

ArgumentException  
   Extract Method

arithmetic series  
   About Arithmetic Series Summation

arithmetic series summation  
   An Example-Geometric Series Summation, About Arithmetic Series Summation

arity  
   N-ary Trees

array  
   Foundational Data Structures
Index

as operator
  Run-Time Type Information and ASCII
association
  Searchable Containers
asymptotic behavior
  Asymptotic Notation
attributes
  Abstract Data Types
AVL balance condition
  AVL Search Trees
AVL rotation
  Balancing AVL Trees
AVL tree
  Basics
B-Tree
  B-Trees, B-Trees
Bachmann, P.
  An Asymptotic Upper Bound-Big
backtracking algorithms
  Backtracking Algorithms
bag
  Projects, Multisets
balance condition
  AVL Search Trees, B-Trees
  AVL
    AVL Search Trees
base class
  Class Hierarchy, Derivation and Inheritance
big oh
  An Asymptotic Upper Bound-Big
tightness
  Tight Big Oh Bounds, More Notation-Theta and Little
tightness
  Tight Big Oh Bounds, More Notation-Theta and Little
transitive property
  Properties of Big Oh
binary digit
  Binomial Queues
binary heap
Index

Sorting with a Heap

binary operator

Applications

binary search

Locating Items in an, Example-Binary Search

binary search tree

Binary Search Trees, Binary Search Trees

binary tree

Binary Trees, Binary Trees

complete

Complete Trees

binding

Abstract Data Types, Variables

binomial

Binomial Trees

binomial coefficient

Binomial Trees

bit

Binomial Queues

Boolean

and

UnionIntersection, and Difference

or

UnionIntersection, and Difference

bound

Abstract Data Types

boxing value types

Wrappers for Value Types

branch-and-bound

Branch-and-Bound Solvers

breadth-first spanning tree

Constructing Spanning Trees

breadth-first traversal

Applications, Applications, Breadth-First Traversal, Example-Balancing Scales, Breadth-First Traversal

brute-force algorithms

Brute-Force and Greedy Algorithms

bubble sort

Bubble Sort

bucket sort

Example-Bucket Sort
The No-Arg Constructor

First and Last Properties

Properties and Accessors

Rules For Big Oh

Application: Critical Path Analysis

Conventions for Writing Big

More Terminology

Single-Source Shortest Path

M-Way Search Trees

Associations

A Lower Bound on

Class Hierarchy

The No-Arg Constructor

The Fragmentation Problem

Applications
Index

Binomial Queues, Binomial Queues, Implementing a Partition using
formal parameter
Pass By Value
Fortran
Abstract Data Types
foundational data structure
Foundational Data Structures
fully connected graph
Exercises
garbage
What is Garbage?
garbage collection
What is Garbage?
mark-and-compact
Mark-and-Compact Garbage Collection
mark-and-sweep
Mark-and-Sweep Garbage Collection
reference counting
Reference Counting Garbage Collection
stop-and-copy
Stop-and-Copy Garbage Collection
Gauss, Karl Friedrich
Binomial Trees
generalized Fibonacci numbers
Example-Generalized Fibonacci Numbers
geometric series
About Geometric Series Summation
geometric series summation
An Example-Geometric Series Summation, Example-Geometric Series Summation Again, About
Geometric Series Summation, Example-Geometric Series Summation Yet
get accessor
Properties and Accessors
golden ratio
Fibonacci Hashing
graph
connectedness
Connectedness of an Undirected
dense
Sparse vs. Dense Graphs
directed
Directed Graphs
directed acyclic
  Directed Acyclic Graphs
labeled
  Labeled Graphs
sparse
  Sparse vs. Dense Graphs
traversal
  Graph Traversals
undirected
  Undirected Graphs
graph theory
  Graphs and Graph Algorithms
handle
  Handles
harmonic number
  Average Running Times, About Harmonic Numbers, Average Case Analysis, Solving The Recurrence-Telescoping, Average Running Time
harmonic series
  About Harmonic Numbers
hash function
  Keys and Hash Functions, Keys and Hash Functions
hash table
  Hash Tables
hashing
  division method
    Division Method
  Fibonacci method
    Fibonacci Hashing
  middle-square method
    Middle Square Method
  multiplication method
    Multiplication Method
head
  Singly-Linked Lists
heap
  Basics, Garbage Collection and the
heapify
  Sorting with a Heap
heapsort
  Sorting with a Heap
height
of a node in a tree

More Terminology

of a tree

More Terminology

heuristic

Depth-First Branch-and-Bound Solver

hierarchy

Trees

Horner's rule

Another Example-Horner's Rule, Example-Geometric Series Summation Again, Character String Keys

IComparable interface

C# Objects and the

IEnumerable interface

Enumerable Collections and Enumerators

IEnumerator interface

Enumerable Collections and Enumerators

implement

Abstract Methods and Abstract implementation

implements

Class Hierarchy

in parameter

In and Out Parameters

in-degree

Terminology, Topological Sort

in-place sorting

Insertion Sorting, Selection Sorting

incident

Terminology

increment

Generating Random Numbers

indexer

DynamicArray Indexers, Finding the Position of, Inserting and Accessing Items

infix

Applications

infix notation

Infix Notation

inheritance

Derivation and Inheritance
single

Derivation and Inheritance

inorder traversal

Inorder Traversal, Traversing a Search Tree

M-way tree

Traversing a Search Tree

insertion sorting

Insertion Sorting

straight

Straight Insertion Sort

insertion sorts

Sorter Class Hierarchy

integral type

Integral Keys

interface

Class Hierarchy, Class Hierarchy, Interfaces

internal node

N-ary Trees

internal path length

Unsuccessful Search

complete binary tree

Complete Trees

internal path length of a tree

Successful Search

Internet domain name

Character String Keys

intersection

Sets Multisets, and Partitions, Basics, Union Intersection, and Difference

interval

search

Locating Items in an

inverse modulo W

Multiplication Method

inversion

Average Running Time

is operator

Run-Time Type Information and

isomorphic

Alternate Representations for Trees

isomorphic trees

Exercises
loose asymptotic bound

More Notation-Theta and Little

LR rotation

Double Rotations

Łukasiewicz, Jan

Applications

M-way search tree

M-Way Search Trees

mantissa

Floating-Point Keys

many-to-one mapping

Keys and Hash Functions

mark-and-compact garbage collection

Mark-and-Compact Garbage Collection

mark-and-sweep garbage collection

Mark-and-Sweep Garbage Collection

matrix

Matrices

addition

Matrices

adjacency

Adjacency Matrices

multiplication

Matrices

sparse

Adjacency Matrices

max-heap

Sorting with a Heap

median

Selecting the Pivot

median-of-three pivot selection

Selecting the Pivot

memory leak

What is Garbage?

merge sort

Example-Merge Sorting

merge sorting

Merge Sorting

merge sorts

Sorter Class Hierarchy

mergeable priority queue
Basics
merging nodes in a B-tree
Removing Items from a Mersenne primes
The Minimal Standard Random method
Class Members: Fields and middle-square hashing method
Middle Square Method
min heap
Basics
minimal subgraph
Minimum-Cost Spanning Trees
minimum spanning tree
Minimum-Cost Spanning Trees
mixed linear congruential random number generator
Generating Random Numbers
modulus
Generating Random Numbers
Monte Carlo methods
Monte Carlo Methods
MSIL
HashingHash Tables, and
multi-dimensional array
Multi-Dimensional Arrays
multiplication hashing method
Multiplication Method
multiplicative linear congruential random number generator
Generating Random Numbers
multiset
Multisets
mutator
Properties and Accessors
N-ary tree
N-ary tree
N-ary Trees
N-queens problem
N-queens problem
Exercises
name
Abstract Data Types, Abstract Data Types, Abstract Data Types, Variables
Index

Nary tree

negative cost cycle

Single-Source Shortest Path

nested class

Nested Classes

nested struct

Implementation

Newton, Isaac.

Binomial Trees

no-arg constructor

The No-Arg Constructor
default

The No-Arg Constructor

node

Applications, Basics, N-ary Trees, Binary Trees, Terminology

non-recursive algorithm

Example-Fibonacci Numbers

normalize

Generating Random Numbers

null path length

Leftist Trees, Leftist Trees

null reference

Null References

object-oriented programming

Abstract Data Types

object-oriented programming language

Abstract Data Types

objective function

Brute-Force Algorithm

odd-even transposition sort

Exercises

omega

An Asymptotic Lower Bound-Omega

open addressing

Scatter Table using Open

operator overloading

Operator Overloading

operator precedence

Applications

optimal binary search tree
Exercises

or

Union, Intersection, and Difference

ordered list

Ordered Lists and Sorted

ordered tree

N-ary Trees, Binary Trees

ordinal number

Positions of Items in

oriented tree

N-ary Trees

out parameter

In and Out Parameters

out-degree

Terminology

overloading operators

Operator Overloading

override

Derivation and Inheritance, Derivation and Inheritance

parameter passing

Parameter Passing

parent

Applications, Terminology

parentheses

Applications

partial order

Comparing Sets

partition

Partitions, Kruskal's Algorithm

Pascal

Abstract Data Types

Pascal's triangle

Example-Computing Binomial Coefficients

Pascal, Blaise

Example-Computing Binomial Coefficients

pass-by-reference

Parameter Passing

pass-by-value

Parameter Passing

path

Terminology
access

Inserting Items into an

path length

external
Unsuccessful Search

internal
Unsuccessful Search

weighted
Shortest-Path Algorithms

perfect binary tree
Searching a Binary Tree, AVL Search Trees

period
Generating Random Numbers

pivot
Quicksort

pointer
Projects, References Types

polar coordinates
Properties and Accessors

Polish notation
Applications

polymorphism
Class Hierarchy, Polymorphism

polynomial
About Polynomials, About Polynomials Again

postcondition
Inserting Items in a

postorder traversal
Postorder Traversal

power set
Array and Bit-Vector Sets

precede lexicographically
Radix Sort

precondition
Inserting Items in a

predecessor
Fields, More Terminology

prefix notation
Prefix Notation

preorder traversal
Preorder Traversal
prepend
  Prepend Method
Prim's algorithm
  Prim's Algorithm
primary clustering
  Linear Probing
prime
  relatively
    Multiplication Method
priority queue
  mergeable
    Basics
probability density function
  Exponentially Distributed Random Variables
probe sequence
  Scatter Table using Open
proper subset
  Comparing Sets
proper superset
  Comparing Sets
pruning a solution space
  Branch-and-Bound Solvers
pseudorandom
  Generating Random Numbers
quadratic
  Conventions for Writing Big
quadratic probing
  Quadratic Probing
queue
  Stacks Queues, and Deques
quicksort
  Quicksort
r-value
  Abstract Data Types
radix sorting
  Radix Sort
random number generator
  linear congruential
    Generating Random Numbers
  mixed linear congruential
    Generating Random Numbers
Double Rotations

Basics, Mark-and-Sweep Garbage Collection

AVL
Balancing AVL Trees

double
Double Rotations

LL
Single Rotations, Removing Items from a

LL
Single Rotations, Removing Items from a

LR
Double Rotations

RL
Double Rotations

RR
Single Rotations, Removing Items from a

RR
Single Rotations, Removing Items from a

row-major order
Array Subscript Calculations

RPN
seeReverse-Polish notation

RR rotation
Single Rotations
in a B-tree
Removing Items from a

RTTI
seeRun-time type information
Run-Time Type Information and

sbyte
The Basic Axioms

scales
Example-Balancing Scales

scatter tables
Scatter Tables

scope
Abstract Data Types, Abstract Data Types, Variables
search interval
Locating Items in an
search tree
$M$-way
$M$-Way Search Trees
binary
Binary Search Trees
seed
Generating Random Numbers
selection sorting
Selection Sorting
selection sorts
Sorter Class Hierarchy
sentinel
Singly-Linked Lists, Adjacency Matrices
separate chaining
Separate Chaining
set
SetsMultisets, and Partitions
set accessor
Properties and Accessors
sibling
Terminology
sign
Floating-Point Keys
significant
Floating-Point Keys
simple cycle
More Terminology
simple type
Value Types
simulated annealing
Simulated Annealing
simulation time
Discrete Event Simulation
single inheritance
Derivation and Inheritance
single rotation
Double Rotations
single-ended queue
Queues
singleton
Exercises, Implementation
singly-linked list
Doubly-Linked and Circular Lists
size
Abstract Data Types
slack time
Application: Critical Path Analysis
slide
Handles
solution space
Example-Balancing Scales
solver
Abstract Backtracking Solvers
sort
topological
Topological Sort
sorted list
Ordered Lists and Sorted, Sorted Lists, Basics
sorter
Sorting and Sorters
sorting
in place
Selection Sorting
in-place
Insertion Sorting
sorting algorithm
bucket sort
Example-Bucket Sort
sorting by distribution
Distribution Sorting
sorting by exchanging
Exchange Sorting
sorting by insertion
Insertion Sorting
sorting by merging
Merge Sorting
sorting by selection
Selection Sorting
source
Exercises

spanning tree
Minimum-Cost Spanning Trees

breadth-first
Constructing Spanning Trees

depth-first
Constructing Spanning Trees

minimum
Minimum-Cost Spanning Trees

sparse graph
Sparse vs. Dense Graphs

sparse matrix
Adjacency Matrices

specializes
Class Hierarchy

stable sorts
Basics

stack
Stacks

stack frame
The Basic Axioms

state
Discrete Event Simulation

static binding
Abstract Data Types

Stirling numbers
Partitions, Partitions

stop-and-copy garbage collection
Stop-and-Copy Garbage Collection

straight insertion sorting
Straight Insertion Sort

straight selection sorting
Straight Selection Sorting

string literal
Wrappers for Value Types

strongly connected
Connectedness of a Directed

struct type
Value Types

subgraph
Minimum-Cost Spanning Trees
total order
  Basics
binary trees
  Comparing Trees
transitive
  Sorted Lists, Applications, Basics
transpose
  Matrices
traversal
  Tree Traversals, Example-Balancing Scales, Graph Traversals
  Breadth-First Traversal, Breadth-First Traversal
  Breadth-First Traversal, Breadth-First Traversal
  Depth-First Traversal
inorder
  Inorder Traversal, Traversing a Search Tree
inorder
  Inorder Traversal, Traversing a Search Tree
postorder
  Postorder Traversal
preorder
  Preorder Traversal
tree
  Basics
N-ary
  N-ary Trees
binary
  Binary Trees
equivalence
  Comparing Trees
expression
  Expression Trees
height
  More Terminology
internal path length
  Successful Search
leftist
  Leftist Trees
ordered
Introduction

- What This Book Is About
- Object-Oriented Design
- Object Hierarchies and Design Patterns
- The Features of C# You Need to Know
- How This Book Is Organized
Acknowledgments

Insert acknowledgements here.

Waterloo, Canada

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Outline

This book presents material identified in the *Computing Curricula 1991* report of the ACM/IEEE-CS Joint Curriculum Task Force[45]. The book specifically addresses the following *knowledge units*: AL1: Basic Data structures, AL2: Abstract Data Types, AL3: Recursive Algorithms, AL4: Complexity Analysis, AL6: Sorting and Searching, and AL8: Problem-Solving Strategies. The breadth and depth of coverage is typical of what should appear in the second or third year of an undergraduate program in computer science/computer engineering.

In order to analyze a program, it is necessary to develop a model of the computer. Chapter □ develops several models and illustrates with examples how these models predict performance. Both average-case and worst-case analyses of running time are considered. Recursive algorithms are discussed and it is shown how to solve a recurrence using repeated substitution. This chapter also reviews arithmetic and geometric series summations, Horner's rule and the properties of harmonic numbers.

Chapter □ introduces asymptotic (big-oh) notation and shows by comparing with Chapter □ that the results of asymptotic analysis are consistent with models of higher fidelity. In addition to $O(\cdot)$, this chapter also covers other asymptotic notations ($\Omega(\cdot)$, $\Theta(\cdot)$, and $o(\cdot)$) and develops the asymptotic properties of polynomials and logarithms.

Chapter □ introduces the *foundational data structures*--the array and the linked list. Virtually all the data structures in the rest of the book can be implemented using either one of these foundational structures. This chapter also covers multi-dimensional arrays and matrices.

Chapter □ deals with abstraction and data types. It presents the recurring design patterns used throughout the text as well a unifying framework for the data structures presented in the subsequent chapters. In particular, all of the data structures are viewed as *abstract containers*.

Chapter □ discusses stacks, queues, and deques. This chapter presents implementations based on both foundational data structures (arrays and linked lists). Applications for stacks and queues are presented.

Chapter □ covers ordered lists, both sorted and unsorted. In this chapter, a list is viewed as a *searchable container*. Again several applications of lists are presented.
Chapter 1 introduces hashing and the notion of a hash table. This chapter addresses the design of hashing functions for the various basic data types as well as for the abstract data types described in Chapter 1. Both scatter tables and hash tables are covered in depth and analytical performance results are derived.

Chapter 2 introduces trees and describes their many forms. Both depth-first and breadth-first tree traversals are presented. Completely generic traversal algorithms based on the use of the visitor design pattern are presented, thereby illustrating the power of algorithmic abstraction. This chapter also shows how trees are used to represent mathematical expressions and illustrates the relationships between traversals and the various expression notations (prefix, infix, and postfix).

Chapter 3 addresses trees as searchable containers. Again, the power of algorithmic abstraction is demonstrated by showing the relationships between simple algorithms and balancing algorithms. This chapter also presents average case performance analyses and illustrates the solution of recurrences by telescoping.

Chapter 4 presents several priority queue implementations, including binary heaps, leftist heaps, and binomial queues. In particular this chapter illustrates how a more complicated data structure (leftist heap) extends an existing one (tree). Discrete-event simulation is presented as an application of priority queues.

Chapter 5 covers sets and multisets. Also covered are partitions and disjoint set algorithms. The latter topic illustrates again the use of algorithmic abstraction.

Garbage collection is discussed in Chapter 6. This is a topic that is not found often in texts of this sort. However, because the C# language relies on garbage collection, it is important to understand how it works and how it affects the running times of programs.

Chapter 7 surveys a number of algorithm design techniques. Included are brute-force and greedy algorithms, backtracking algorithms (including branch-and-bound), divide-and-conquer algorithms, and dynamic programming. An object-oriented approach based on the notion of an abstract solution space and an abstract solver unifies much of the discussion. This chapter also covers briefly random number generators, Monte Carlo methods, and simulated annealing.

Chapter 8 covers the major sorting algorithms in an object-oriented style based on the notion of an abstract sorter. Using the abstract sorter illustrates the relationships between the various classes of sorting algorithm and demonstrates the use of algorithmic abstractions.

Finally, Chapter 9 presents an overview of graphs and graph algorithms. Both depth-first and breadth-first graph traversals are presented. Topological sort is viewed as yet another special kind of traversal.
Generic traversal algorithms based on the *visitor* design pattern are presented, once more illustrating *
algorithmic abstraction*. This chapter also covers various shortest path algorithms and minimum-
spanning-tree algorithms.

At the end of each chapter is a set of exercises and a set of programming projects. The exercises are
designed to consolidate the concepts presented in the text. The programming projects generally require
the student to extend the implementation given in the text.
Suggested Course Outline

This text may be used in either a one semester or a two semester course. The course which I teach at Waterloo is a one-semester course that comprises 36 lecture hours on the following topics:

1. Review of the fundamentals of programming in C# and an overview of object-oriented programming with C#. (Appendix □). [4 lecture hours].
2. Models of the computer, algorithm analysis, and asymptotic notation (Chapters □ and □). [4 lecture hours].
3. Foundational data structures, abstraction, and abstract data types (Chapters □ and □). [4 lecture hours].
4. Stacks, queues, ordered lists, and sorted lists (Chapters □ and □). [3 lecture hours].
5. Hashing, hash tables, and scatter tables (Chapter □). [3 lecture hours].
6. Trees and search trees (Chapters □ and □). [6 lecture hours].
7. Heaps and priority queues (Chapter □). [3 lecture hours].
8. Algorithm design techniques (Chapter □). [3 lecture hours].
9. Sorting algorithms and sorters (Chapter □). [3 lecture hours].
10. Graphs and graph algorithms (Chapter □). [3 lecture hours].

Depending on the background of students, a course instructor may find it necessary to review features of the C# language. For example, an understanding of *inner classes* is required for the implementation of *enumerations*. Similarly, students need to understand the workings of *classes*, *interfaces*, and *inheritance* in order to understand the unifying class hierarchy discussed in Chapter □.
Online Course Materials

Additional material supporting this book can be found on the world-wide web at the URL:

http://www.pads.uwaterloo.ca/Bruno.Preiss/books/opus6

In particular, you will find there the source code for all the program fragments in this book as well as an errata list.
What This Book Is About

This book is about the fundamentals of *data structures and algorithms*—the basic elements from which large and complex software artifacts are built. To develop a solid understanding of a data structure requires three things: First, you must learn how the information is arranged in the memory of the computer. Second, you must become familiar with the algorithms for manipulating the information contained in the data structure. And third, you must understand the performance characteristics of the data structure so that when called upon to select a suitable data structure for a particular application, you are able to make an appropriate decision.

This book also illustrates object-oriented design and it promotes the use of common, object-oriented design patterns. The algorithms and data structures in the book are presented in the C# programming language. Virtually all the data structures are presented in the context of a single class hierarchy. This commitment to a single design allows the programs presented in the later chapters to build upon the programs presented in the earlier chapters.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Object-Oriented Design

Traditional approaches to the design of software have been either data oriented or process oriented. Data-oriented methodologies emphasize the representation of information and the relationships between the parts of the whole. The actions which operate on the data are of less significance. On the other hand, process-oriented design methodologies emphasize the actions performed by a software artifact; the data are of lesser importance.

It is now commonly held that object-oriented methodologies are more effective for managing the complexity which arises in the design of large and complex software artifacts than either data-oriented or process-oriented methodologies. This is because data and processes are given equal importance. Objects are used to combine data with the procedures that operate on that data. The main advantage of using objects is that they provide both abstraction and encapsulation.

- Abstraction
- Encapsulation
Abstraction

Abstraction can be thought of as a mechanism for suppressing irrelevant details while at the same time emphasizing relevant ones. An important benefit of abstraction is that it makes it easier for the programmer to think about the problem to be solved.

For example, *procedural abstraction* lets the software designer think about the actions to be performed without worrying about how those actions are implemented. Similarly, *data abstraction* lets the software designer think about the objects in a program and the interactions between those objects without having to worry about how those objects are implemented.

There are also many different *levels of abstraction*. The lower the levels of abstraction expose more of the details of an implementation whereas the higher levels hide more of the details.
Encapsulation

Encapsulation aids the software designer by enforcing *information hiding*. Objects *encapsulate* data and the procedures for manipulating that data. In a sense, the object *hides* the details of the implementation from the user of that object.

There are two very real benefits from encapsulation--*conceptual* and *physical* independence. Conceptual independence results from hiding the implementation of an object from the user of that object. Consequently, the user is prevented from doing anything with an object that depends on the implementation of that object. This is desirable because it allows the implementation to be changed without requiring the modification of the user's code.

Physical independence arises from the fact that the behavior of an object is determined by the object itself. The behavior of an object is not determined by some external entity. As a result, when we perform an operation on an object, there are no unwanted side-effects.
There is more to object-oriented programming than simply encapsulating in an object some data and the procedures for manipulating those data. Object-oriented methods deal also with the classification of objects and they address the relationships between different classes of objects.

The primary facility for expressing relationships between classes of objects is derivation--new classes can be derived from existing classes. What makes derivation so useful is the notion of inheritance. Derived classes inherit the characteristics of the classes from which they are derived. In addition, inherited functionality can be overridden and additional functionality can be defined in a derived class.

A feature of this book is that virtually all the data structures are presented in the context of a single class hierarchy. In effect, the class hierarchy is a taxonomy of data structures. Different implementations of a given abstract data structure are all derived from the same abstract base class. Related base classes are in turn derived from classes that abstract and encapsulate the common features of those classes.

In addition to dealing with hierarchically related classes, experienced object-oriented designers also consider very carefully the interactions between unrelated classes. With experience, a good designer discovers the recurring patterns of interactions between objects. By learning to use these patterns, your object-oriented designs will become more flexible and reusable.

Recently, programmers have to started name the common design patterns. In addition, catalogs of the common patterns are now being compiled and published[15].

The following object-oriented design patterns are used throughout this text:

- Containers
- Enumerators
- Visitors
- Cursors
- Adapters
Singletons

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Containers

A container is an object that holds within it other objects. A container has a capacity, it can be full or empty, and objects can be inserted and withdrawn from a container. In addition, a searchable container is a container that supports efficient search operations.
Enumerators

An enumerator provides a means by which the objects within a container can be accessed one-at-a-time. All enumerators share a common interface, and hide the underlying implementation of the container from the user of that container.
Visitors

A visitor represents an operation to be performed on all the objects within a container. All visitors share a common interface, and thereby hide the operation to be performed from the container. At the same time, visitors are defined separately from containers. Thus, a particular visitor can be used with any container.
A cursor represents the position of an object in an ordered container. It provides the user with a way to specify where an operation is to be performed without having to know how that position is represented.
Adapters

An adapter converts the interface of one class into the interface expected by the user of that class. This allows a given class with an incompatible interface to be used in a situation where a different interface is expected.
Singletons

A singleton is a class of which there is only one instance. The class ensures that there only one instance is created and it provides a way to access that instance.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
The Features of C# You Need to Know

This book does not teach the basics of programming. It is assumed that you have taken an introductory course in programming and that you have learned how to write a program in C#. That is, you have learned the rules of C# syntax and you have learned how to put together C# statements in order to solve rudimentary programming problems. The following paragraphs describe more fully aspects of programming in C# with which you should be familiar.

- Variables
- Value Types and Reference Types
- Parameter Passing
- Classes and Objects
- Inheritance
- Interfaces and Polymorphism
- Other Features
Variables

You must be very comfortable with the notion of a variable as an abstraction for a region of a memory. A variable has attributes such as name, type, value, address size, lifetime, and scope.
Value Types and Reference Types

You must understand the differences between the value types and reference types. In particular, you should understand the subtle differences which arise when assigning and comparing reference types.
Parameter Passing

There are two parameter passing mechanisms in C#, *pass-by-value* and *pass-by-reference*. It is essential that you understand how these mechanisms work both for value types and for reference types.
Classes and Objects

A C# class encapsulates a set of values and a set of operations. The values are represented by the fields of the class and the operations by the methods of the class. In C# a class definition introduces a new type. The instances of a class type are called objects.
Inheritance

In C# one class may be derived from another. The derived class *inherits* all the fields and the methods of the base class or classes. In addition, inherited methods can be overridden in the derived class and new fields and functions can be defined. You should understand how the compiler determines the code to execute when a particular method is called.
Interfaces and Polymorphism

A C# interface comprises a set of method prototypes. Different classes can implement the same interface. In this way, C# facilities polymorphism—the idea that a given abstraction can have many different forms. You should understand how interfaces are used together with abstract classes and inheritance to support polymorphism.
Other Features

This book makes use of other C# features such as exceptions and run-time type information. You can learn about these topics as you work your way through the book.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
How This Book Is Organized

- Models and Asymptotic Analysis
- Foundational Data Structures
- Abstract Data Types and the Class Hierarchy
- Data Structures
- Algorithms

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Models and Asymptotic Analysis

To analyze the performance of an algorithm, we need to have a model of the computer. Chapter presents a series of three models, each one less precise but easier to use than its predecessor. These models are similar, in that they require a careful accounting of the operations performed by an algorithm.

Next, Chapter presents asymptotic analysis. This is an extremely useful mathematical technique because it simplifies greatly the analysis of algorithms. Asymptotic analysis obviates the need for a detailed accounting of the operations performed by an algorithm, yet at the same time gives a very general result.
Foundational Data Structures

When implementing a data structure, we must decide first whether to use an array or a linked list as the underlying organizational technique. For this reason, the array and the linked list are called foundational data structures. Chapter also covers multi-dimensional arrays and matrices.
Chapter introduces the notion of an abstract data type. All of the data structures discussed in this book are presented as instances of various abstract data types. Chapter also introduces the class hierarchy as well as the various related concepts such as enumerators and visitors.
Data Structures

Chapter covers stacks, queues, and deques. Ordered lists and sorted lists are presented in Chapter. The concept of hashing is introduced in Chapter. This chapter also covers the design of hash functions for a number of different object types. Finally, hash tables and scatter tables are presented.

Trees and search trees are presented in Chapters and . Trees are one of the most important non-linear data structures. Chapter also covers the various tree traversals, including depth-first traversal and breadth-first traversal. Chapter presents priority queues and Chapter covers sets, multisets, and partitions.

An essential element of the C# run-time system is the pool of dynamically allocated storage. Chapter presents a number of different approaches for implementing garbage collection, in the process illustrating the actual costs associated with dynamic storage allocation.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Algorithms

The last three chapters of the book focus on algorithms, rather than data structures. Chapter 4 is an overview of various algorithmic patterns. By introducing the notion of an abstract problem solver, we show how many of the patterns are related. Chapter 5 uses a similar approach to present various sorting algorithms. That is, we introduce the notion of an abstract sorter and show how the various sorting algorithms are related.

Finally, Chapter 6 gives a brief overview of the subject of graphs and graph algorithms. This chapter brings together various algorithmic techniques from Chapter 5 with the class hierarchy discussed in the earlier chapters.
Algorithm Analysis

What is an algorithm and why do we want to analyze one? An algorithm is "a...step-by-step procedure for accomplishing some end."[10] An algorithm can be given in many ways. For example, it can be written down in English (or French, or any other "natural" language). However, we are interested in algorithms which have been precisely specified using an appropriate mathematical formalism--such as a programming language.

Given such an expression of an algorithm, what can we do with it? Well, obviously we can run the program and observe its behavior. This is not likely to be very useful or informative in the general case. If we run a particular program on a particular computer with a particular set of inputs, then all know is the behavior of the program in a single instance. Such knowledge is anecdotal and we must be careful when drawing conclusions based upon anecdotal evidence.

In order to learn more about an algorithm, we can "analyze" it. By this we mean to study the specification of the algorithm and to draw conclusions about how the implementation of that algorithm--the program--will perform in general. But what can we analyze? We can

- determine the running time of a program as a function of its inputs;
- determine the total or maximum memory space needed for program data;
- determine the total size of the program code;
- determine whether the program correctly computes the desired result;
- determine the complexity of the program--e.g., how easy is it to read, understand, and modify; and,
- determine the robustness of the program--e.g., how well does it deal with unexpected or erroneous inputs?

In this text, we are concerned primarily with the running time. We also consider the memory space needed to execute the program. There are many factors that affect the running time of a program. Among these are the algorithm itself, the input data, and the computer system used to run the program. The performance of a computer is determined by

- the hardware:
  - processor used (type and speed),
Algorithm Analysis

- memory available (cache and RAM), and
- disk available;
- the programming language in which the algorithm is specified;
- the language compiler/interpreter used; and
- the computer operating system software.

A detailed analysis of the performance of a program which takes all of these factors into account is a very difficult and time-consuming undertaking. Furthermore, such an analysis is not likely to have lasting significance. The rapid pace of change in the underlying technologies means that results of such analyses are not likely to be applicable to the next generation of hardware and software.

In order to overcome this shortcoming, we devise a "model" of the behavior of a computer with the goals of simplifying the analysis while still producing meaningful results. The next section introduces the first in a series of such models.
In this section we develop a detailed model of the running time performance of C# programs. The model developed is independent of the underlying hardware and system software. Rather than analyze the performance of a particular, arbitrarily chosen physical machine, we model the execution of a C# program on the "common language runtime" (see Figure).

A direct consequence of this approach is that we lose some fidelity--the resulting model cannot predict accurately the performance of all possible hardware/software systems. On the other hand, the resulting model is still rather complex and rich in detail.
**Figure:** C# system overview.

- The Basic Axioms
- A Simple Example-Arithmetic Series Summation
A Detailed Model of the Computer

- Array Subscripting Operations
- Another Example-Horner's Rule
- Analyzing Recursive Methods
- Yet Another Example-Finding the Largest Element of an Array
- Average Running Times
- About Harmonic Numbers
- Best-Case and Worst-Case Running Times
- The Last Axiom
The Basic Axioms

The running time performance of the common language runtime is given by a set of axioms which we shall now postulate. The first axiom addresses the running time of simple variable references:

Axiom The time required to fetch an operand from memory is a constant, $T_{\text{fetch}}$, and the time required to store a result in memory is a constant, $T_{\text{store}}$.

According to Axiom, the assignment statement

\[
y = x;
\]

has running time $T_{\text{fetch}} + T_{\text{store}}$. That is, the time taken to fetch the value of variable $x$ is $T_{\text{fetch}}$ and the time taken to store the value in variable $y$ is $T_{\text{store}}$.

We shall apply Axiom to manifest constants too: The assignment

\[
y = 1;
\]

also has running time $T_{\text{fetch}} + T_{\text{store}}$. To see why this should be the case, consider that the constant typically needs to be stored in the memory of the computer, and we can expect the cost of fetching it to be the same as that of fetching any other operand.

The next axiom addresses the running time of simple arithmetic operations:

Axiom The times required to perform elementary arithmetic operations, such as addition, subtraction, multiplication, division, and comparison, are all constants. These times are denoted by $T_+$, $T_-$, $T_\times$, $T_\div$, and $T_\prec$, respectively.

According to Axiom, all the simple operations can be accomplished in a fixed amount of time. In order for this to be feasible, the number of bits used to represent a value must be fixed. In C#, the number of bits needed to represent a number range from 8 (for byte and sbyte) to 64 (for long, ulong and double). It is precisely because the number of bits used is fixed that we can say that the running times are also fixed. If arbitrarily large numbers are allowed, then the basic arithmetic operations can take an arbitrarily long amount of time.
By applying Axioms \( \Box \) and \( \Box \), we can determine that the running time of a statement like

\[
y = y + 1;
\]

is \( 2\tau_{\text{fetch}} + \tau_+ + \tau_{\text{store}} \). This is because we need to fetch two operands, \( y \) and \( 1 \); add them; and, store the result back in \( y \).

C# syntax provides several alternative ways to express the same computation:

\[
\begin{align*}
y &+= 1; \\
++y; \\
y++; \\
\end{align*}
\]

We shall assume that these alternatives require exactly the same running time as the original statement.

The third basic axiom addresses the method call/return overhead:

**Axiom** The time required to call a method is a constant, \( \tau_{\text{call}} \), and the time required to return from a method is a constant, \( \tau_{\text{return}} \).

When a method is called, certain housekeeping operations need to be performed. Typically this includes saving the return address so that program execution can resume at the correct place after the call, saving the state of any partially completed computations so that they may be resumed after the call, and the allocation of a new execution context (stack frame or activation record) in which the called method can be evaluated. Conversely, on the return from a method, all of this work is undone. While the method call/return overhead may be rather large, nevertheless it entails a constant amount of work.

In addition to the method call/return overhead, additional overhead is incurred when parameters are passed to the method:

**Axiom** The time required to pass an argument to a method is the same as the time required to store a value in memory, \( \tau_{\text{store}} \).

The rationale for making the overhead associated with parameter passing the same as the time to store a value in memory is that the passing of an argument is conceptually the same as assignment of the actual parameter value to the formal parameter of the method.

According to Axiom \( \Box \), the running time of the statement

\[
y = F(x);
\]
would be $t_{\text{fetch}} + 2t_{\text{store}} + t_{\text{call}} + T_F(x)$, where $T_F(x)$ is the running time of method $F$ for input $x$. The first of the two stores is due to the passing of the parameter $x$ to the method $F$; the second arises from the assignment to the variable $y$. 
A Simple Example—Arithmetic Series Summation

In this section we apply Axioms, and to the analysis of the running time of a program to compute the following simple arithmetic series summation

$$\sum_{i=1}^{n} i.$$ 

The algorithm to compute this summation is given in Program □.

```csharp
class Example
{
    public static int Sum(int n)
    {
        int result = 0;
        for (int i = 1; i <= n; ++i)
            result += i;
        return result;
    }
}
```

Program: Program to compute $\sum_{i=1}^{n} i$.

The executable statements in Program □ comprise lines 5-8. Table □ gives the running times of each of these statements.
Table: Computing the running time of Program.

Note that the for statement on line 6 of Program has been split across three lines in Table. This is because we analyze the running time of each of the elements of a for statement separately. The first element, the initialization code, is executed once before the first iteration of the loop. The second element, the loop termination test, is executed before each iteration of the loop begins. Altogether, the number of times the termination test is executed is one more than the number of times the loop body is executed. Finally, the third element, the loop counter increment step, is executed once per loop iteration.

Summing the entries in Table we get that the running time, \(T(n)\), of Program is

\[
T(n) = t_1 + t_2 n
\]  

(2.1)

where \(t_1 = 5T_{\text{fetch}} + 2T_{\text{store}} + T_{\text{<}} + T_{\text{return}}\) and \(t_2 = 6T_{\text{fetch}} + 2T_{\text{store}} + T_{\text{<}} + 2T_{\text{+}}\).
Array Subscripting Operations

We now address the question of accessing the elements of an array of data. In general, the elements of a one-dimensional array are stored in consecutive memory locations. Therefore, given the address of the first element of the array, a simple addition suffices to determine the address of an arbitrary element of the array:

**Axiom** The time required for the *address calculation* implied by an array subscripting operation, e.g., \(a[i]\), is a constant, \(T[]\). This time does not include the time to compute the subscript expression, nor does it include the time to access (i.e., fetch or store) the array element.

By applying Axiom, we can determine that the running time for the statement

\[y = a[i];\]

is \(3T_{\text{fetch}} + T[.] + T_{\text{store}}\). Three operand fetches are required: the first to fetch \(a\), the base address of the array; the second to fetch \(i\), the index into the array; and, the third to fetch array element \(a[i]\).
Another Example-Horner's Rule

In this section we apply Axioms □, □, □ and □ to the analysis of the running time of a program which evaluates the value of a polynomial. That is, given the \( n+1 \) coefficients \( a_0, a_1, \ldots, a_n \), and a value \( x \), we wish to compute the following summation

\[
\sum_{i=0}^{n} a_i x^i.
\]

The usual way to evaluate such polynomials is to use Horner's rule, which is an algorithm to compute the summation without requiring the computation of arbitrary powers of \( x \). The algorithm to compute this summation is given in Program □. Table □ gives the running times of each of the executable statements in Program □.

```java
public class Example
{
    public static int Horner(int[] a, int n, int x)
    {
        int result = a[n];
        for (int i = n - 1; i >= 0; --i)
            result = result * x + a[i];
        return result;
    }
}
```

**Program:** Program to compute \( \sum_{i=0}^{n} a_i x^i \) using Horner's rule.
Another Example—Horner’s Rule

<table>
<thead>
<tr>
<th>statement</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>$3T_{\text{fetch}} + T[\cdot] + T_{\text{store}}$</td>
</tr>
<tr>
<td>6a</td>
<td>$2T_{\text{fetch}} + T_{\text{pre}} + T_{\text{store}}$</td>
</tr>
<tr>
<td>6b</td>
<td>$(2T_{\text{fetch}} + T_{\text{pre}}) \times (n + 1)$</td>
</tr>
<tr>
<td>6c</td>
<td>$(2T_{\text{fetch}} + T_{\text{pre}} + T_{\text{store}}) \times n$</td>
</tr>
<tr>
<td>7</td>
<td>$(5T_{\text{fetch}} + T[\cdot] + T_{\text{pre}} + T_{\text{store}} + T_{\text{pre}} + T_{\text{store}}) \times n$</td>
</tr>
<tr>
<td>8</td>
<td>$T_{\text{fetch}} + T_{\text{return}}$</td>
</tr>
<tr>
<td>TOTAL</td>
<td>$(9T_{\text{fetch}} + 2T_{\text{store}} + T_{\text{pre}} + T[\cdot] + T_{\text{pre}} + T_{\text{store}} + T_{\text{pre}} + T_{\text{store}}) \times n$ $+ (8T_{\text{fetch}} + 2T_{\text{store}} + T[\cdot] + T_{\text{pre}} + T_{\text{store}} + T_{\text{return}})$</td>
</tr>
</tbody>
</table>

**Table:** Computing the running time of Program □.

Summing the entries in Table □ we get that the running time, $T(n)$, of Program □ is

$$T(n) = t_1 + t_2 n$$ (2.2)

where $t_1 = 8T_{\text{fetch}} + 2T_{\text{store}} + T[\cdot] + T_{\text{pre}} + T_{\text{return}}$ and $t_2 = 9T_{\text{fetch}} + 2T_{\text{store}} + T_{\text{pre}} + T[\cdot] + T_{\text{pre}} + T_{\text{store}} + T_{\text{pre}} + T_{\text{store}} + T_{\text{return}}$.
Analyzing Recursive Methods

In this section we analyze the performance of a recursive algorithm which computes the factorial of a number. Recall that the factorial of a non-negative integer $n$, written $n!$, is defined as

$$n! = \begin{cases} 
1 & n = 0, \\
\prod_{i=1}^{n} i & n > 0.
\end{cases}$$

(2.3)

However, we can also define factorial recursively as follows

$$n! = \begin{cases} 
1 & n = 0, \\
n \times (n-1)! & n > 0.
\end{cases}$$

It is this latter definition which leads to the algorithm given in Program to compute the factorial of $n$. Table gives the running times of each of the executable statements in Program.

```csharp
public class Example
{
    public static int Factorial(int n)
    {
        if (n == 0)
            return 1;
        else
            return n * Factorial(n - 1);
    }
}
```

Program: Recursive program to compute $n!$. 
Analyzing Recursive Methods

<table>
<thead>
<tr>
<th>statement</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n=0$</td>
</tr>
<tr>
<td>5</td>
<td>$2\tau_{\text{fetch}} + \tau_{\lt}$</td>
</tr>
<tr>
<td>6</td>
<td>$\tau_{\text{fetch}} + \tau_{\text{return}}$</td>
</tr>
<tr>
<td>8</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table: Computing the running time of Program.

Notice that we had to analyze the running time of the two possible outcomes of the conditional test on line 5 separately. Clearly, the running time of the program depends on the result of this test.

Furthermore, the method `Factorial` calls itself recursively on line 8. Therefore, in order to write down the running time of line 8, we need to know the running time, $T(\cdot)$, of `Factorial`. But this is precisely what we are trying to determine in the first place! We escape from this catch-22 by assuming that we already know what is the function $T(\cdot)$, and that we can make use of that function to determine the running time of line 8.

By summing the columns in Table we get that the running time of Program is

$$T(n) = \begin{cases} t_1 & n = 0, \\ T(n-1) + t_2 & n > 0, \end{cases} \quad (2.4)$$

where $t_1 = 3\tau_{\text{fetch}} + \tau_{\lt} + \tau_{\text{return}}$ and

$t_2 = 5\tau_{\text{fetch}} + \tau_{\lt} + \tau_{\text{store}} + \tau_{\times} + \tau_{\text{call}} + \tau_{\text{return}}$. This kind of equation is called a recurrence relation because the function is defined in terms of itself recursively.

- Solving Recurrence Relations-Repeated Substitution
Solving Recurrence Relations-Repeated Substitution

In this section we present a technique for solving a recurrence relation such as Equation called **repeated substitution**. The basic idea is this: Given that \( T(n) = T(n - 1) + t_2 \), then we may also write \( T(n - 1) = T(n - 2) + t_2 \), provided \( n > 1 \). Since \( T(n-1) \) appears in the right-hand side of the former equation, we can substitute for it the entire right-hand side of the latter. By repeating this process we get

\[
T(n) = T(n-1) + t_2 \\
= (T(n-2) + t_2) + t_2 \\
= T(n-2) + 2t_2 \\
= (T(n-3) + t_2) + 2t_2 \\
= T(n-3) + 3t_2 \\
\vdots
\]

The next step takes a little intuition: We must try to discern the pattern which is emerging. In this case it is obvious:

\[
T(n) = T(n-k) + kt_2,
\]

where \( 1 \leq k \leq n \). Of course, if we have doubts about our intuition, we can always check our result by induction:

**Proof (By Induction).** **Base Case** Clearly the formula is correct for \( k = 1 \), since

\[
T(n) = T(n-1) + t_2 = T(n-1) + t_2.
\]

**Inductive Hypothesis** Assume that \( T(n) = T(n-k) + kt_2 \) for \( k = 1, 2, \ldots, l \). By this assumption

\[
T(n) = T(n-l) + lt_2. \tag{2.5}
\]

Note also that using the original recurrence relation we can write
Solving Recurrence Relations—Repeated Substitution

\[ T(n-l) = T(n-l-1) + t_2 \]  \hspace{1cm} (2.6)

for \( l \leq n \). Substituting Equation (2.6) in the right-hand side of Equation (2.5) gives

\[
T(n) = T(n-l-1) + t_2 + lt_2 \\
= T(n-(l+1)) + (l+1)t_2
\]

Therefore, by induction on \( l \), our formula is correct for all \( 0 \leq k \leq n \).

So, we have shown that \( T(n) = T(n-k) + kt_2 \), for \( 1 \leq k \leq n \). Now, if \( n \) was known, we would repeat the process of substitution until we got \( T(0) \) on the right hand side. The fact that \( n \) is unknown should not deter us—we get \( T(0) \) on the right hand side when \( n-k=0 \). That is, \( k=n \). Letting \( k=n \) we get

\[
T(n) = T(n-k) + kt_2 \\
= T(0) + nt_2 \\
= t_1 + nt_2
\]  \hspace{1cm} (2.7)

where \( t_1 = 3t_{\text{fetch}} + \tau_{<} + \tau_{\text{return}} \) and
\[
t_2 = 5t_{\text{fetch}} + \tau_{<} + \tau_{-} + \tau_{\text{store}} + \tau_{x} + \tau_{\text{call}} + \tau_{\text{return}}.
\]

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Yet Another Example-Finding the Largest Element of an Array

In this section we consider the problem of finding the largest element of an array. That is, given an array of \( n \) non-negative integers, \( a_0, a_1, \ldots, a_{n-1} \), we wish to find

\[
\max_{0 \leq i < n} a_i.
\]

The straightforward way of solving this problem is to perform a linear search of the array. The linear search algorithm is given in Program 1 and the running times for the various statements are given in Table 1.

```csharp
public class Example
{
    public static int FindMaximum(int[] a)
    {
        int result = a[0];
        for (int i = 1; i < a.Length; ++i)
            if (a[i] > result)
                result = a[i];
        return result;
    }
}
```

Program: Linear search to find \( \max_{0 \leq i < n} a_i \).

<table>
<thead>
<tr>
<th>statement</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>( 3T_{\text{fetch}} + T[\cdot] + T_{\text{store}} )</td>
</tr>
</tbody>
</table>
Yet Another Example—Finding the Largest Element of an Array

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6a</td>
<td>(T_{\text{fetch}} + T_{\text{store}})</td>
</tr>
<tr>
<td>6b</td>
<td>((2T_{\text{fetch}} + T_\prec) \times n)</td>
</tr>
<tr>
<td>6c</td>
<td>((2T_{\text{fetch}} + T_\prec + T_{\text{store}}) \times (n - 1))</td>
</tr>
<tr>
<td>7</td>
<td>((4T_{\text{fetch}} + T_\prec + T_{\text{store}}) \times (n - 1))</td>
</tr>
<tr>
<td>8</td>
<td>((3T_{\text{fetch}} + T_\prec + T_{\text{store}}) \times ?)</td>
</tr>
<tr>
<td>9</td>
<td>(T_{\text{fetch}} + T_{\text{store}})</td>
</tr>
</tbody>
</table>

**Table:** Computing the running time of Program □.

With the exception of line 8, the running times follow simply from Axioms □, □ and □. In particular, note that the body of the loop is executed \(n-1\) times. This means that the conditional test on line 7 is executed \(n-1\) times. However, the number of times line 8 is executed depends on the data in the array and not just \(n\).

If we consider that in each iteration of the loop body, the variable result contains the largest array element seen so far, then line 8 will be executed in the \(i^{th}\) iteration of the loop only if \(a_i\) satisfies the following

\[ a_i > \left( \max_{0 \leq j < i} a_j \right) . \]

Thus, the running time of Program □, \(T(\cdot)\), is a function not only of the number of elements in the array, \(n\), but also of the actual array values, \(a_0, a_1, \ldots, a_{n-1}\). Summing the entries in Table □ we get

\[ T(n, a_0, a_1, \ldots, a_{n-1}) = t_1 + t_2 n + \sum_{i=1}^{n-1} t_3 \]

where

\[ t_1 = 2T_{\text{store}} - T_{\text{fetch}} - T_+ - T_\prec \]

\[ t_2 = 8T_{\text{fetch}} + 2T_\prec + T_\prec + T_+ + T_{\text{store}} \]

\[ t_3 = 3T_{\text{fetch}} + T_\prec + T_{\text{store}} \].
Yet Another Example—Finding the Largest Element of an Array

While this result may be correct, it is not terribly useful. In order to determine the running time of the program we need to know the number of elements in the array, \( n \), and we need to know the values of the elements in the array, \( a_0, a_1, \ldots, a_{n-1} \). Even if we know these data, it turns out that in order to compute the running time of the algorithm, \( T(n, a_0, a_1, \ldots, a_{n-1}) \), we actually have to solve the original problem!
Average Running Times

In the previous section, we found the function, $T(n, a_0, a_1, \ldots, a_{n-1})$, which gives the running time of Program as a function both of number of inputs, $n$, and of the actual input values. Suppose instead we are interested in a function $T_{\text{average}}(n)$ which gives the running time on average for $n$ inputs, regardless of the values of those inputs. In other words, if we run Program a large number of times on a selection of random inputs of length $n$, what will the average running time be?

We can write the sum of the running times given in Table in the following form

$$T_{\text{average}}(n) = t_1 + t_2 n + \sum_{i=1}^{n-1} p_i t_3 \quad (2.8)$$

where $p_i$ is the probability that line 8 of the program is executed. The probability $p_i$ is given by

$$p_i = P \left[ a_i > \{ \max_{0 \leq j < i} a_j \} \right].$$

That is, $p_i$ is the probability that the $i^{th}$ array entry, $a_i$, is larger than the maximum of all the preceding array entries, $a_0, a_1, \ldots, a_{i-1}$.

In order to determine $p_i$, we need to know (or to assume) something about the distribution of input values. For example, if we know a priori that the array passed to the method FindMaximum is ordered from smallest to largest, then we know that $p_i = 1$. Conversely, if we know that the array is ordered from largest to smallest, then we know that $p_i = 0$.

In the general case, we have no a priori knowledge of the distribution of the values in the input array. In this case, consider the $i^{th}$ iteration of the loop. In this iteration $a_i$ is compared with the maximum of the $i$ values, $a_0, a_1, \ldots, a_{i-1}$ preceding it in the array. Line 6 of Program is only executed if $a_i$ is the largest of the $i+1$ values $a_0, a_1, \ldots, a_i$. All things being equal, we can say that this will happen with
probability $1/(i+1)$. Thus

$$p_i = P\left[a_i > \left(\max_{0\leq j < i} a_j\right)\right]$$

$$= \frac{1}{i + 1}.$$ 

Substituting this expression for $p_i$ in Equation (2.9) and simplifying the result we get

$$T_{\text{average}}(n) = t_1 + t_2n + \sum_{i=1}^{n-1} p_i t_3$$

$$= t_1 + t_2n + t_3 \sum_{i=1}^{n-1} \frac{1}{i + 1}$$

$$= t_1 + t_2n + t_3 \left(\sum_{i=1}^{n} \frac{1}{i} - 1\right)$$

$$= t_1 + t_2n + t_3(H_n - 1)$$

(2.10)

where $H_n = \sum_{i=1}^{n} \frac{1}{i}$, is the $n^{th}$ harmonic number.
About Harmonic Numbers

The series $1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots$ is called the harmonic series, and the summation

$$H_n = \sum_{i=1}^{n} \frac{1}{i}$$

gives rise to the series of harmonic numbers, $H_1, H_2, \ldots$. As it turns out, harmonic numbers often creep into the analysis of algorithms. Therefore, we should understand a little bit about how they behave.

A remarkable characteristic of harmonic numbers is that, even though as $n$ gets large and the difference between consecutive harmonic numbers gets arbitrarily small ($H_n - H_{n-1} = \frac{1}{n}$), the series does not converge! That is, $\lim_{n \to \infty} H_n$ does not exist. In other words, the summation $\sum_{i=1}^{\infty} \frac{1}{i}$ goes off to infinity, but just barely.

Figure helps us to understand the behavior of harmonic numbers. The smooth curve in this figure is the function $y = 1/x$. The descending staircase represents the function $y = 1/|x|$.
About Harmonic Numbers

**Figure:** Computing harmonic numbers.

Notice that the area under the staircase between 1 and \( n \) for any integer \( n > 1 \) is given by

\[
\int_1^n \frac{1}{x} \, dx = \sum_{i=1}^{n-1} \frac{1}{i} = H_{n-1}.
\]

Thus, if we can determine the area under the descending staircase in Figure, we can determine the values of the harmonic numbers.

As an approximation, consider the area under the smooth curve \( y = 1/x \):

\[
\int_1^n \frac{1}{x} \, dx = \ln x \bigg|_1^n = \ln n.
\]

Thus, \( H_{n-1} \) is approximately \( \ln n \) for \( n > 1 \).

If we approximate \( H_{n-1} \) by \( \ln n \), the error in this approximation is equal to the area between the two curves. In fact, the area between these two curves is such an important quantity that it has its own
symbol, \( \gamma \), which is called Euler's constant. The following derivation indicates a way in which to compute Euler's constant:

\[
\gamma = \lim_{n \to \infty} (\sum_{i=1}^{n-1} \left( \frac{1}{i} - \ln \frac{i+1}{i} \right))
\]

\[
= \sum_{i=1}^{\infty} \left( \int_{i}^{i+1} \frac{1}{x} \, dx \right)
\]

\[
= \sum_{i=1}^{\infty} \left( \ln \frac{i+1}{i} \right)
\]

\[
\approx 0.577215
\]

A program to compute Euler's constant on the basis of this derivation is given in Program 1. While this is not necessarily the most accurate or most speedy way to compute Euler's constant, it does give the correct result to six significant digits.

```
public class Example {
    public static double Gamma() {
        double result = 0;
        for (int i = 1; i <= 500000; ++i)
            result += 1.0/i - Math.Log((i + 1.0)/i);
        return result;
    }
}
```

**Program:** Program to compute \( \gamma \).

So, with Euler's constant in hand, we can write down an expression for the \((n-1)th\) harmonic number:

\[
H_{n-1} = \ln n + \gamma - \epsilon_n
\]

where \( \epsilon_n \) is the error introduced by the fact that \( \gamma \) is defined as the difference between the curves on the interval \([1, +\infty)\), but we only need the difference on the interval \([1, n]\). As it turns out, it can be shown
(but not here), that there exists a constant $K$ such that for large enough values of $n$, $|\varepsilon_n| < \frac{K}{n}$.

Since the error term is less than $1/n$, we can add $1/n$ to both sides of Equation and still have an error which goes to zero as $n$ gets large. Thus, the usual approximation for the harmonic number is

$$H_n \approx \ln n + \gamma.$$

We now return to the question of finding the average running time of Program, which finds the largest element of an array. We can now rewrite Equation to give

$$T_{\text{average}}(n) = t_1 + t_2 n + t_3 (H_n - 1)
\approx t_1 + t_2 n + t_3 (\ln n + \gamma - 1)
\approx (t_1 + t_3 (\gamma - 1)) + t_2 n + t_3 \ln n.$$
Best-Case and Worst-Case Running Times

In Section we derived the average running time of Program which finds the largest element of an array. In order to do this we had to determine the probability that a certain program statement is executed. To do this, we made an assumption about the average input to the program.

The analysis can be significantly simplified if we simply wish to determine the worst case running time. For Program, the worst-case scenario occurs when line 8 is executed in every iteration of the loop. We saw that this corresponds to the case in which the input array is ordered from smallest to largest. In terms of Equation , this occurs when \( p_i = 1 \). Thus, the worst-case running time is given by

\[
T_{\text{worst case}}(n) = t_1 + t_2n + \sum_{i=1}^{n-1} p_i t_3 \bigg|_{p_i=1} \\
= t_1 + t_2n + t_3 \sum_{i=1}^{n-1} 1 \\
= t_1 + t_2n + t_3(n - 1) \\
= (t_1 - t_3) + (t_2 + t_3) \times n.
\]

Similarly, the best-case running time occurs when line 8 is never executed. This corresponds to the case in which the input array is ordered from largest to smallest. This occurs when \( p_i = 0 \) and best-case running time is

\[
T_{\text{best case}}(n) = t_1 + t_2n + \sum_{i=1}^{n-1} p_i t_3 \bigg|_{p_i=0} \\
= t_1 + t_2n.
\]

In summary we have the following results for the running time of Program:

\[
T_{\text{worst case}}(n) = (t_1 - t_3) + (t_2 + t_3) \times n, \\
T_{\text{best case}}(n) = t_1 + t_2n.
\]
\[ T(n, a_0, a_1, \ldots, a_{n-1}) = t_1 + t_2 n + \sum_{i=1}^{n-1} \alpha_i \left( \max_{0 \leq j < i} a_j \right) \]

\[ T_{\text{average}}(n) \approx (t_1 + t_3 (\gamma - 1)) + t_2 n + t_3 \ln n \]

\[ T_{\text{worst case}}(n) = (t_1 - t_3) + (t_2 + t_3) \times n \]

\[ T_{\text{best case}}(n) = t_1 + t_2 n \]
The Last Axiom

In this section we state the last axiom needed for the detailed model of the common language runtime. This axiom addresses the time required to create a new object instance:

**Axiom**  The time required to create a new object instance using the `new` operator is a constant, \( \tau_{\text{new}} \). This time does not include any time taken to initialize the object.

By applying Axioms \( A_1 \), \( A_2 \), \( A_3 \) and \( A_4 \), we can determine that the running time of the statement

\[
\text{Int32 ref = new Int32(0);}
\]

is \( \tau_{\text{new}} + \tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{\text{call}} + \mathcal{T}(\text{Int32}) \), where \( \mathcal{T}(\text{Int32}) \) is the running time of the \text{Int32} constructor.

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
A Simplified Model of the Computer

The detailed model of the computer given in the previous section is based on a number of different timing parameters—\( T_{\text{fetch}} \), \( T_{\text{store}} \), \( T_{\text{+}} \), \( T_{\text{-}} \), \( T_{\text{x}} \), \( T_{\text{<=}} \), \( T_{\text{call}} \), \( T_{\text{return}} \), \( T_{\text{new}} \), and \( T_{\text{[.]} \) . While it is true that a model with a large number of parameters is quite flexible and therefore likely to be a good predictor of performance, keeping track of the all of the parameters during the analysis is rather burdensome.

In this section, we present a simplified model which makes the performance analysis easier to do. The cost of using the simplified model is that it is likely to be a less accurate predictor of performance than the detailed model.

Consider the various timing parameters in the detailed model. In a real machine, each of these parameters is a multiple of the basic clock period of the machine. The clock frequency of a modern computer is typically between 500 MHz and 2 GHz. Therefore, the clock period is typically between 0.5 and 2 ns. Let the clock period of the machine be \( T \). Then each of the timing parameters can be expressed as an integer multiple of the clock period. For example, \( T_{\text{fetch}} = k_{\text{fetch}} T \), where \( k_{\text{fetch}} \in \mathbb{Z} \), \( k_{\text{fetch}} > 0 \).

The simplified model eliminates all of the arbitrary timing parameters in the detailed model. This is done by making the following two simplifying assumptions:

- All timing parameters are expressed in units of clock cycles. In effect, \( T=1 \).
- The proportionality constant, \( k \), for all timing parameters is assumed to be the same: \( k=1 \).

The effect of these two assumptions is that we no longer need to keep track of the various operations separately. To determine the running time of a program, we simply count the total number of cycles taken.

- An Example—Geometric Series Summation
- About Arithmetic Series Summation
A Simplified Model of the Computer

- Example-Geometric Series Summation Again
- About Geometric Series Summation
- Example-Computing Powers
- Example-Geometric Series Summation Yet Again

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
An Example-Geometric Series Summation

In this section we consider the running time of a program to compute the following geometric series summation. That is, given a value $x$ and non-negative integer $n$, we wish to compute the summation

$$\sum_{i=0}^{n} x^i.$$

An algorithm to compute this summation is given in Program 1.

```csharp
    public class Example
    {
        public static int GeometricSeriesSum(int x, int n)
        {
            int sum = 0;
            for (int i = 0; i <= n; ++i)
            {
                int prod = 1;
                for (int j = 0; j < i; ++j)
                    prod *= x;
                sum += prod;
            }
            return sum;
        }
    }
```

Program: Program to compute $\sum_{i=0}^{n} x^i$.

Table 1 gives the running time, as predicted by the simplified model, for each of the executable statements in Program 1.
In order to calculate the total cycle counts, we need to evaluate the two series summations $\sum_{i=0}^{n}(i + 1)$ and $\sum_{i=0}^{n}i$. Both of these are arithmetic series summations. In the next section we show that the sum of the series $1, 2, \ldots, n$ is $n(n+1)/2$. Using this result we can sum the cycle counts given in Table to arrive at the total running time of $\frac{11}{2}n^2 + \frac{47}{2}n + 24$ cycles.
About Arithmetic Series Summation

The series, $1, 2, 3, 4, \ldots$, is an arithmetic series and the summation

$$S_n = \sum_{i=1}^{n} i$$

is called the arithmetic series summation.

The summation can be solved as follows: First, we make the simple variable substitution $i = n - j$:

$$\sum_{i=1}^{n} i = \sum_{n-j=1}^{n} (n - j)$$

$$= \sum_{j=0}^{n-1} (n - j)$$

$$= \sum_{j=0}^{n-1} n - \sum_{j=0}^{n-1} j$$

$$= n \sum_{j=0}^{n-1} 1 - \sum_{j=1}^{n} j + n \quad \text{(2.12)}$$

Note that the term in the first summation in Equation 2.12 is independent of $j$. Also, the second summation is identical to the left hand side. Rearranging Equation 2.12, and simplifying gives
There is, of course, a simpler way to arrive this answer. Consider the series, \(1, 2, 3, 4, \ldots, n\), and suppose \(n\) is even. The sum of the first and last element is \(n+1\). So too is the sum of the second and second-last element, and the third and third-last element, etc., and there are \(n/2\) such pairs. Therefore,
\[ S_n = \frac{n}{2}(n + 1). \]

And if \(n\) is odd, then \(S_n = S_{n-1} + n\), where \(n-1\) is even. So we can use the previous result for \(S_{n-1}\) to get
\[ S_n = \frac{n-1}{2}n + n = \frac{n(n + 1)}{2}. \]
Example-Geometric Series Summation Again

In this example we revisit the problem of computing a geometric series summation. We have already seen an algorithm to compute this summation in Section (Program). This algorithm was shown to take \( \frac{11}{2} n^2 + \frac{47}{2} n + 24 \) cycles.

The problem of computing the geometric series summation is identical to that of computing the value of a polynomial in which all of the coefficients are one. This suggests that we could make use of Horner's rule as discussed in Section (Section). An algorithm to compute a geometric series summation using Horner's rule is given in Program (Program).

```
public class Example
{
    public static int GeometricSeriesSum(int x, int n)
    {
        int sum = 0;
        for (int i = 0; i <= n; ++i)
            sum = sum * x + 1;
        return sum;
    }
}
```

Program: Program to compute \( \sum_{i=0}^{n} x^i \) using Horner's rule.

The executable statements in Program comprise lines 5-8. Table gives the running times, as given by the simplified model, for each of these statements.
In Programs □ and □ we have seen two different algorithms to compute the same geometric series summation. We determined the running time of the former to be $\frac{11}{2} n^2 + \frac{47}{2} n + 24$ cycles and of the latter to be $13n+22$ cycles. In particular, note that for all non-negative values of $n$, 

$\left( \frac{11}{2} n^2 + \frac{47}{2} n + 24 \right) > 13n + 22$. Hence, according to our simplified model of the computer, Program □, which uses Horner's rule, always runs faster than Program □!
The series, $1, a, a^2, a^3, \ldots$, is a geometric series and the summation

$$S_n = \sum_{i=0}^{n} a^i$$

is called the geometric series summation.

The summation can be solved as follows: First, we make the simple variable substitution $i=j-1$:

$$\sum_{i=0}^{n} a^i = \sum_{j=1}^{n+1} a^{j-1} = \frac{1}{a} \sum_{j=1}^{n+1} a^j$$

$$= \frac{1}{a} \left( \sum_{j=0}^{n} a^j + a^{n+1} - 1 \right) \tag{2.13}$$

Note that the summation which appears on the right is identical to the left hand side. Rearranging Equation 2.13, and simplifying gives

$$\sum_{i=0}^{n} a_i = \frac{a^{n+1} - 1}{a - 1} \tag{2.14}$$
Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Example-Computing Powers

In this section we consider the running time to raise a number to a given integer power. That is, given a value $x$ and non-negative integer $n$, we wish to compute the $x^n$. A naïve way to calculate $x^n$ would be to use a loop such as

```c
int result = 1;
for (int i = 0; i <= n; ++i)
    result *= x;
```

While this may be fine for small values of $n$, for large values of $n$ the running time may become prohibitive. As an alternative, consider the following recursive definition

$$x^n = \begin{cases} 
1 & n = 0, \\
(x^2)^{n/2} & n > 0, n \text{ is even,} \\
x(x^2)^{n/2} & n > 0, n \text{ is odd.}
\end{cases} \quad (2.15)$$

For example, using Equation (2.15), we would determine $x^{32}$ as follows

$$x^{32} = \left( \left( (x^2)^2 \right)^2 \right)^2,$$

which requires a total of five multiplication operations. Similarly, we would compute $x^{31}$ as follows

$$x^{31} = \left( \left( (x^2)^2 x \right)^2 x \right)^2 x,$$

which requires a total of eight multiplication operations.

A recursive algorithm to compute $x^n$ based on the direct implementation of Equation (2.15) is given in Program [Program]. Table [Table] gives the running time, as predicted by the simplified model, for each of the executable statements in Program [Program].
public class Example
{
    public static int Power(int x, int n)
    {
        if (n == 0)
            return 1;
        else if (n % 2 == 0) // n is even
            return Power(x * x, n / 2);
        else // n is odd
            return x * Power(x * x, n / 2);
    }
}

Program: Program to compute $x^n$.

<table>
<thead>
<tr>
<th>statement</th>
<th>$n=0$ time</th>
<th>$n&gt;0$ time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n&gt;0$ time</td>
<td>$n&gt;0$ time</td>
</tr>
<tr>
<td></td>
<td>$n$ is even</td>
<td>$n$ is odd</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>--</td>
</tr>
<tr>
<td>7</td>
<td>--</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>--</td>
<td>10 + $T([n/2])$</td>
</tr>
<tr>
<td>10</td>
<td>--</td>
<td>12 + $T([n/2])$</td>
</tr>
<tr>
<td>TOTAL</td>
<td>5</td>
<td>18 + $T([n/2])$</td>
</tr>
</tbody>
</table>

TOTAL 5 18 + $T([n/2])$ 20 + $T([n/2])$

Table: Computing the running time of Program □.

By summing the columns in Table □ we get the following recurrence for the running time of Program □.
Example—Computing Powers

As the first attempt at solving this recurrence, let us suppose that \( n = 2^k \) for some \( k > 0 \). Clearly, since \( n \) is a power of two, it is even. Therefore, \( \left\lfloor n/2 \right\rfloor = n/2 = 2^{k-1} \).

For \( n = 2^k \), Equation 2.16 gives

\[
T(2^k) = 18 + T(2^{k-1}), \quad k > 0.
\]

This can be solved by repeated substitution:

\[
\begin{align*}
T(2^k) &= 18 + T(2^{k-1}) \\
&= 18 + 18 + T(2^{k-2}) \\
&= 18 + 18 + 18 + T(2^{k-3}) \\
&\vdots \\
&= 18j + T(2^{k-j}).
\end{align*}
\]

The substitution stops when \( k = j \). Thus,

\[
T(2^k - 1) = 18k + T(1) \\
= 18k + 20 + T(0) \\
= 18k + 20 + 5 \\
= 18k + 25.
\]

Note that if \( n = 2^k \), then \( k = \log_2 n \). In this case, running time of Program \( 2 \) is

\[
T(n) = 18 \log_2 n + 25.
\]

The preceding result is, in fact, the best case—in all but the last two recursive calls of the method, \( n \) was even. Interestingly enough, there is a corresponding worst-case scenario. Suppose \( n = 2^k - 1 \) for some value of \( k > 0 \). Clearly \( n \) is odd, since it is one less than \( 2^k \) which is a power of two and even. Now consider \( \left\lfloor n/2 \right\rfloor \).
\[
\left\lfloor \frac{n}{2} \right\rfloor = \left\lfloor \frac{(2^k - 1)/2}{2} \right\rfloor = \frac{(2^k - 2)}{2} = 2^{k-1} - 1.
\]

Hence, \( \left\lfloor \frac{n}{2} \right\rfloor \) is also odd!

For example, suppose \( n \) is 31 (\( 2^5 - 1 \)). To compute \( x^{31} \), Program □ calls itself recursively to compute \( x^{15} \), \( x^7 \), \( x^3 \), \( x^1 \), and finally, \( x^0 \)--all but the last of which are odd powers of \( x \).

For \( n = 2^k - 1 \), Equation □ gives

\[
T(2^k - 1) = 20 + T(2^{k-1} - 1), \quad k > 1.
\]

Solving this recurrence by repeated substitution we get

\[
\begin{align*}
T(2^k - 1) & = 20 + T(2^{k-1} - 1) \\
& = 20 + 20 + T(2^{k-2} - 1) \\
& = 20 + 20 + 20 + T(2^{k-3} - 1) \\
& \quad \vdots \\
& = 20j + T(2^{k-j} - 1).
\end{align*}
\]

The substitution stops when \( k=j \). Thus,

\[
T(2^k - 1) = 20k + T(2^0 - 1) = 20k + 5.
\]

Note that if \( n = 2^k - 1 \), then \( k = \log_2(n + 1) \). In this case, running time of Program □ is \( T(n) = 20 \log_2(n + 1) + 5 \).

Consider now what happens for an arbitrary value of \( n \). Table □ shows the recursive calls made by Program □ in computing \( x^n \) for various values of \( n \).
Example-Computing Powers

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \lfloor \log_2 n \rfloor + 1 )</th>
<th>powers computed recursively</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1, 0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2, 1, 0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3, 1, 0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4, 2, 1, 0</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>5, 2, 1, 0</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>6, 3, 1, 0</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>7, 3, 1, 0</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>8, 4, 2, 1, 0</td>
</tr>
</tbody>
</table>

Table: Recursive calls made in Program 1.

By inspection we determine that the number of recursive calls made in which the second argument is non-zero is \( \lfloor \log_2 n \rfloor + 1 \). Furthermore, depending on whether the argument is odd or even, each of these calls contributes either 18 or 20 cycles. The pattern emerging in Table 1 suggests that, on average just as many of the recursive calls result in an even number as result in an odd one. The final call (zero argument) adds another 5 cycles. So, on average, we can expect the running time of Program 1 to be

\[
T(n) = 19(\lfloor \log_2 n \rfloor + 1) + 5.
\]  

(2.17)
Example-Geometric Series Summation Yet Again

In this example we consider the problem of computing a geometric series summation for the last time. We have already seen two algorithms to compute this summation in Sections and (Programs and ).

An algorithm to compute a geometric series summation using the closed-form expression (Equation ) is given in Program . This algorithm makes use of Program to compute .

```
1  public class Example
2  {
3      public static int GeometricSeriesSum(int x, int n)
4      {
5          return (Power(x, n + 1) - 1) / (x - 1);
6      }
7  }
```

**Program:** Program to compute \( \sum_{i=0}^{n} x^i \) using the closed-form expression.

To determine the average running time of Program we will make use of Equation, which gives the average running time for the Power method which is called on line 5. In this case, the arguments are \( x \) and \( n+1 \), so the running time of the call to Power is \( 19(\lfloor \log_2(n + 1) \rfloor + 1) + 5 \). Adding to this the additional work done on line 5 gives the average running time for Program:

\[
T(n) = 19(\lfloor \log_2(n + 1) \rfloor + 1) + 18.
\]

The running times of the three programs which compute the geometric series summation presented in this chapter are tabulated below in Table and are plotted for \( 1 \leq n \leq 100 \) in Figure. The plot shows that, according to our simplified model of the computer, Program has the best running time for \( n < 4 \). However as \( n \) increases, Program is clearly the fastest of the three and Program is the slowest for all values of \( n \).
<table>
<thead>
<tr>
<th>Program</th>
<th>T(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>$(\frac{11}{2} n^2 + \frac{47}{2} n + 24)$</td>
</tr>
<tr>
<td>Program</td>
<td>$13n+22$</td>
</tr>
<tr>
<td>Program</td>
<td>$19\left\lfloor \log_2 (n+1) \right\rfloor + 1 + 18$</td>
</tr>
</tbody>
</table>

**Table:** Running times of Programs 1, 2, and 3.

**Figure:** Plot of running time vs. $n$ for Programs 1, 2, and 3.
Exercises

1. Determine the running times predicted by the detailed model of the computer given in Section for each of the following program fragments:

   1. for (int i = 0; i < n; ++i)
      ++k;

   2. for (int i = 1; i < n; i *= 2)
      ++k;

   3. for (int i = n - 1; i != 0; i /= 2)
      ++k;

   4. for (int i = 0; i < n; ++i)
      if (i % 2 == 0)
      ++k;

   5. for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j)
      ++k;

   6. for (int i = 0; i < n; ++i)
      for (int j = i; j < n; ++j)
      ++k;

   7. for (int i = 0; i < n; ++i)
      for (int j = 0; j < i * i; ++j)
      ++k;

2. Repeat Exercise, this time using the simplified model of the computer given in Section.

3. Prove by induction the following summation formulas:

   1. \[ \sum_{i=0}^{n} i = \frac{n(n + 1)}{2} \]
2. \( \sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \)

3. \( \sum_{i=0}^{n} i^3 = \frac{n^2(n+1)^2}{4} \)

4. Evaluate each of the following series summations:
   1. \( \sum_{i=0}^{n} 2^i \)
   2. \( \sum_{i=0}^{n} \left(\frac{1}{2}\right)^i \)
   3. \( \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i \)
   4. \( \sum_{i=-\infty}^{n} 2^i \)

5. Show that \( \sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \), for \( 0 \leq a < 1 \). \textbf{Hint}: Let \( S_n = \sum_{i=0}^{n} a^i \) and show that \( \lim_{n \to \infty} (S_n - aS_n) = 1 \).

6. Show that \( \sum_{i=0}^{\infty} i/2^i = 2 \). \textbf{Hint}: Let \( S_n = \sum_{i=0}^{n} i/2^i \) and show that the difference \( 2S_n - S_n \) is (approximately) a geometric series summation.

7. Solve each of the following recurrences by repeated substitution:
   1. \( T(n) = \begin{cases} 1 & n = 0, \\ T(n-1) + 1 & n > 0 \end{cases} \)
   2. \( T(n) = \begin{cases} 1 & n \leq \alpha, \alpha > 0, \\ T(n-\alpha) + 1 & n > \alpha. \end{cases} \)
   3. \( T(n) = \begin{cases} 1 & n = 0, \\ 2T(n-1) + 1 & n > 0 \end{cases} \)
   4. \( T(n) = \begin{cases} 1 & n = 0, \\ 2T(n-1) + n & n > 0 \end{cases} \)
   5. \( T(n) = \begin{cases} 1 & n = 1, \\ T(n/2) + 1 & n > 1. \end{cases} \)
Exercises

6. \( T(n) = \begin{cases} 
1 & n = 1, \\
2T(n/2) + 1 & n > 1. 
\end{cases} \)

7. \( T(n) = \begin{cases} 
1 & n = 1, \\
2T(n/2) + n & n > 1. 
\end{cases} \)
Projects

1. Write a non-recursive method to compute the factorial of $n$ according to Equation $\square$. Calculate the running time predicted by the detailed model given in Section $\square$ and the simplified model given in Section $\square$.

2. Write a non-recursive method to compute $x^n$ according to Equation $\square$. Calculate the running time predicted by the detailed model given in Section $\square$ and the simplified model given in Section $\square$.

3. Write a program that determines the values of the timing parameters of the detailed model ($T_{\text{fetch}}, T_{\text{store}}, T_+, T_-, T_{\times}, T_{\div}, T_{\text{call}}, T_{\text{return}}, T_{\text{new}}$, and $T[\cdot]$) for the machine on which it is run.

4. Using the program written for Project $\square$, determine the timing parameters of the detailed model for your computer. Then, measure the actual running times of Programs $\square$, $\square$ and $\square$ and compare the measured results with those predicted by Equations $\square$, $\square$ and $\square$ (respectively).

5. Given a sequence of $n$ integers, $\{a_0, a_1, \ldots, a_{n-1}\}$, and a small positive integer $k$, write an algorithm to compute

$$
\sum_{i=0}^{n-1} 2^{ki} a_i,
$$

without multiplication. **Hint**: Use Horner's rule and bitwise shifts.

6. Verify Equation $\square$ experimentally as follows: Generate a large number of random sequences of length $n$, $\{a_0, a_1, a_2, \ldots, a_{n-1}\}$. For each sequence, test the hypothesis that the probability that $a_i$ is larger than all its predecessors in the sequence is $p_i = 1/(i+1)$. (For a good source of random numbers, see Section $\square$).
Asymptotic Notation

Suppose we are considering two algorithms, A and B, for solving a given problem. Furthermore, let us say that we have done a careful analysis of the running times of each of the algorithms and determined them to be $T_A(n)$ and $T_B(n)$, respectively, where $n$ is a measure of the problem size. Then it should be a fairly simple matter to compare the two functions $T_A(n)$ and $T_B(n)$ to determine which algorithm is the best!

But is it really that simple? What exactly does it mean for one function, say $T_A(n)$, to be better than another function, $T_B(n)$? One possibility arises if we know the problem size a priori. For example, suppose the problem size is $n_0$ and $T_A(n_0) < T_B(n_0)$. Then clearly algorithm A is better than algorithm B for problem size $n_0$.

In the general case, we have no a priori knowledge of the problem size. However, if it can be shown, say, that $T_A(n) \leq T_B(n)$ for all $n \geq 0$, then algorithm A is better than algorithm B regardless of the problem size.

Unfortunately, we usually don't know the problem size beforehand, nor is it true that one of the functions is less than or equal the other over the entire range of problem sizes. In this case, we consider the asymptotic behavior of the two functions for very large problem sizes.
An Asymptotic Upper Bound-Big Oh

In 1892, P. Bachmann invented a notation for characterizing the asymptotic behavior of functions. His invention has come to be known as *big oh notation*:

**Definition (Big Oh)** Consider a function $f(n)$ which is non-negative for all integers $n \geq 0$. We say that "$f(n)$ is big oh $g(n)$," which we write $f(n) = \mathcal{O}(g(n))$, if there exists an integer $n_0$ and a constant $c > 0$ such that for all integers $n \geq n_0$, $f(n) \leq cg(n)$.

---

- **A Simple Example**
- **Big Oh Fallacies and Pitfalls**
- **Properties of Big Oh**
- **About Polynomials**
- **About Logarithms**
- **Tight Big Oh Bounds**
- **More Big Oh Fallacies and Pitfalls**
- **Conventions for Writing Big Oh Expressions**

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
A Simple Example

Consider the function $f(n)=8n+128$ shown in Figure □. Clearly, $f(n)$ is non-negative for all integers $n \geq 0$. We wish to show that $f(n) = O(n^2)$. According to Definition □, in order to show this we need to find an integer $n_0$ and a constant $c > 0$ such that for all integers $n \geq n_0$, $f(n) \leq cn^2$.

It does not matter what the particular constants are—as long as they exist! For example, suppose we choose $c=1$. Then

$$f(n) \leq cn^2 \quad \Rightarrow \quad 8n + 128 \leq n^2$$
$$\Rightarrow \quad 0 \leq n^2 - 8n - 128$$
$$\Rightarrow \quad 0 \leq (n - 16)(n + 8).$$

Since $(n+8)>0$ for all values of $n \geq 0$, we conclude that $(n_0 - 16) \geq 0$. That is, $n_0 = 16$.

So, we have that for $c=1$ and $n_0 = 16$, $f(n) \leq cn^2$ for all integers $n \geq n_0$. Hence, $f(n) = O(n^2)$.

Figure □ clearly shows that the function $f(n) = n^2$ is greater than the function $f(n)=8n+128$ to the right of $n=16$.

Of course, there are many other values of $c$ and $n_0$ that will do. For example, $c=2$ and $n_0 = 2 + 4\sqrt{17} \approx 10.2$ will do, as will $c=4$ and $n_0 = 1 + \sqrt{33} \approx 6.7$. (See Figure □).
Figure: Showing that \( f(n) = 8n + 128 = \Theta(n^2) \).
Big Oh Fallacies and Pitfalls

Unfortunately, the way we write big oh notation can be misleading to the naïve reader. This section presents two fallacies which arise because of a misinterpretation of the notation.

**Fallacy** Given that \( f_1(n) = O(g(n)) \) and \( f_2(n) = O(g(n)) \), then \( f_1(n) = f_2(n) \).

Consider the equations:

\[
\begin{align*}
 f_1(n) &= h(n^2) \\
 f_2(n) &= h(n^2).
\end{align*}
\]

Clearly, it is reasonable to conclude that \( f_1(n) = f_2(n) \).

However, consider these equations:

\[
\begin{align*}
 f_1(n) &= O(n^2) \\
 f_2(n) &= O(n^2).
\end{align*}
\]

It *does not* follow that \( f_1(n) = f_2(n) \). For example, \( f_1(n) = n \) and \( f_2(n) = n^2 \) are both \( O(n^2) \), but they are not equal.

**Fallacy** If \( f(n) = O(g(n)) \), then \( g(n) = O^{-1}(f(n)) \).

Consider functions \( f \), \( g \), and \( h \), such that \( f(n) = h(g(n)) \). It is reasonable to conclude that \( g(n) = h^{-1}(f(n)) \) provided that \( h(\cdot) \) is an invertible function. However, while we may write \( f(n) = O(h(n)) \), the equation \( g(n) = O^{-1}(f(n)) \) is nonsensical and meaningless. Big oh is not a mathematical function, so it has no inverse!

The reason for these difficulties is that we should read the notation \( f(n) = O(n^2) \) as "\( f(n) \) is big oh \( n \) squared" not "\( f(n) \) equals big oh of \( n \) squared." The equal sign in the expression does not really denote
mathematical equality! And the use of the functional form, $O(\cdot)$, does not really mean that $O$ is a mathematical function!
Properties of Big Oh

In this section we examine some of the mathematical properties of big oh. In particular, suppose we know that \( f_1(n) = O(g_1(n)) \) and \( f_2(n) = O(g_2(n)) \).

- What can we say about the asymptotic behavior of the sum of \( f_1(n) \) and \( f_2(n) \)? (Theorems \( \square \) and \( \square \)).
- What can we say about the asymptotic behavior of the product of \( f_1(n) \) and \( f_2(n) \)? (Theorems \( \square \) and \( \square \)).
- How are \( f_1(n) \) and \( g_2(n) \) related when \( g_1(n) = f_2(n) \)? (Theorem \( \square \)).

The first theorem addresses the asymptotic behavior of the sum of two functions whose asymptotic behaviors are known:

**Theorem** If \( f_1(n) = O(g_1(n)) \) and \( f_2(n) = O(g_2(n)) \), then

\[
f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n))).
\]

**Proof** By Definition \( \square \), there exist two integers, \( n_1 \) and \( n_2 \) and two constants \( c_1 \) and \( c_2 \) such that \( f_1(n) \leq c_1 g_1(n) \) for \( n \geq n_1 \) and \( f_2(n) \leq c_2 g_2(n) \) for \( n \geq n_2 \).

Let \( n_0 = \max(n_1, n_2) \) and \( c_0 = 2 \max(c_1, c_2) \). Consider the sum \( f_1(n) + f_2(n) \) for \( n \geq n_0 \):

\[
f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n), \quad n \geq n_0
\]
\[
\leq c_0 (g_1(n) + g_2(n))/2
\]
\[
\leq c_0 \max(g_1(n), g_2(n)).
\]

Thus, \( f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n))) \).

According to Theorem \( \square \), if we know that functions \( f_1(n) \) and \( f_2(n) \) are \( O(g_1(n)) \) and \( O(g_2(n)) \),
respectively, the sum \( f_1(n) + f_2(n) \) is \( O(\max(g_1(n), g_2(n))) \). The meaning of \( \max(g_1(n), g_2(n)) \) in this context is the function \( h(n) \) where \( h(n) = \max(g_1(n), g_2(n)) \) for integers all \( n \geq 0 \).

For example, consider the functions \( g_1(n) = 1 \) and \( g_2(n) = 2 \cos^2(n\pi/2) \). Then

\[
    h(n) = \max(g_1(n), g_2(n)) = \max(1, 2 \cos^2(n\pi/2)) = \begin{cases} 1 & \text{if } n \text{ is even}, \\ 2 & \text{if } n \text{ is odd}. \end{cases}
\]

Theorem \( \Box \) helps us simplify the asymptotic analysis of the sum of functions by allowing us to drop the \( \max \) required by Theorem \( \Box \) in certain circumstances:

**Theorem** If \( f(n) = f_1(n) + f_2(n) \) in which \( f_1(n) \) and \( f_2(n) \) are both non-negative for all integers \( n \geq 0 \) such that \( \lim_{n \to \infty} f_2(n)/f_1(n) = L \) for some limit \( L \geq 0 \), then \( f(n) = O(f_1(n)) \).

**Proof** According to the definition of limits, the notation

\[
    \lim_{n \to \infty} \frac{f_2(n)}{f_1(n)} = L
\]

means that, given any arbitrary positive value \( \epsilon \), it is possible to find a value \( n_0 \) such that for all \( n \geq n_0 \)

\[
    \left| \frac{f_2(n)}{f_1(n)} - L \right| \leq \epsilon.
\]

Thus, if we chose a particular value, say \( \epsilon_0 \), then there exists a corresponding \( n_0 \) such that
Consider the sum \( f(n) = f_1(n) + f_2(n) \):

\[
\begin{align*}
f(n) &= f_1(n) + f_2(n) \\
&\leq c_1 f_1(n) + c_2 f_2(n) \\
&\leq c_1 f_1(n) + c_2 (\varepsilon_0 + L) f_1(n), \quad n \geq n_0 \\
&\leq c_0 f_1(n)
\end{align*}
\]

where \( c_0 = c_1 + c_2 (\varepsilon_0 + L) \). Thus, \( f(n) = \Theta(f_1(n)) \).

Consider a pair of functions \( f_1(n) \) and \( f_2(n) \), which are known to be \( O(g_1(n)) \) and \( O(g_2(n)) \), respectively. According to Theorem \( \square \), the sum \( f(n) = f_1(n) + f_2(n) \) is \( O(\max(g_1(n), g_2(n))) \).

However, Theorem \( \square \) says that, if \( \lim_{n \to \infty} f_2(n)/f_1(n) \) exists, then the sum \( f(n) \) is simply \( O(f_1(n)) \) which, by the transitive property (see Theorem \( \square \) below), is \( O(g_1(n)) \).

In other words, if the ratio \( f_1(n)/f_2(n) \) asymptotically approaches a constant as \( n \) gets large, we can say that \( f_1(n) + f_2(n) \) is \( O(g_1(n)) \), which is often a lot simpler than \( O(\max(g_1(n), g_2(n))) \).

Theorem \( \square \) is particularly useful result. Consider \( f_1(n) = n^3 \) and \( f_2(n) = n^2 \).

\[
\lim_{n \to \infty} \frac{f_2(n)}{f_1(n)} = \lim_{n \to \infty} \frac{n^2}{n^3} = \lim_{n \to \infty} \frac{1}{n^1} = 0.
\]

From this we can conclude that \( f_1(n) + f_2(n) = n^3 + n^2 = \Theta(n^3) \). Thus, Theorem \( \square \) suggests that the sum of a series of powers of \( n \) is \( \Theta(n^m) \), where \( m \) is the largest power of \( n \) in the summation.
Properties of Big Oh

We will confirm this result in Section □ below.

The next theorem addresses the asymptotic behavior of the product of two functions whose asymptotic behaviors are known:

**Theorem** If \( f_1(n) = O(g_1(n)) \) and \( f_2(n) = O(g_2(n)) \), then

\[
f_1(n) \times f_2(n) = O(g_1(n) \times g_2(n)).
\]

**extbfProof** By Definition □, there exist two integers, \( n_1 \) and \( n_2 \) and two constants \( c_1 \) and \( c_2 \) such that \( f_1(n) \leq c_1 g_1(n) \) for \( n \geq n_1 \) and \( f_2(n) \leq c_2 g_2(n) \) for \( n \geq n_2 \). Furthermore, by Definition □, \( f_1(n) \) and \( f_2(n) \) are both non-negative for all integers \( n \geq 0 \).

Let \( n_0 = \max(n_1, n_2) \) and \( c_0 = c_1 c_2 \). Consider the product \( f_1(n) \times f_2(n) \) for \( n \geq n_0 \):

\[
f_1(n) \times f_2(n) \leq c_1 g_1(n) \times c_2 g_2(n), \quad n \geq n_0
\]

Thus, \( f_1(n) \times f_2(n) = O(g_1(n) \times g_2(n)) \).

**Theorem □** describes a simple, but extremely useful property of big oh. Consider the functions \( f_1(n) = n^3 + n^2 + n + 1 = O(n^3) \) and \( f_2(n) = n^2 + n + 1 = O(n^2) \). By Theorem □, the asymptotic behavior of the product \( f_1(n) \times f_2(n) \) is \( O(n^3 \times n^2) = O(n^5) \). That is, we are able to determine the asymptotic behavior of the product without having to go through the gory details of calculating that \( f_1(n) \times f_2(n) = n^5 + 2n^4 + 3n^3 + 3n^2 + 2n + 1 \).

The next theorem is closely related to the preceding one in that it also shows how big oh behaves with respect to multiplication.

**Theorem** If \( f_1(n) = O(g_1(n)) \) and \( g_2(n) \) is a function whose value is non-negative for integers \( n \geq 0 \), then

\[
f_1(n) \times g_2(n) = O(g_1(n) \times g_2(n)).
\]

**extbfProof** By Definition □, there exist integers \( n_0 \) and constant \( c_0 \) such that \( f_1(n) \leq c_0 g_1(n) \) for
Properties of Big Oh

Since \( g_2(n) \) is never negative,

\[
f_1(n) \times g_2(n) \leq c_0 g(n) \times g_2(n), \quad n \geq n_0.
\]

Thus, \( f_1(n) \times g_2(n) = \mathcal{O}(g_1(n) \times g_2(n)) \).

Theorem □ applies when we multiply a function, \( f_1(n) \), whose asymptotic behavior is known to be \( \mathcal{O}(g_1(n)) \), by another function \( g_2(n) \). The asymptotic behavior of the result is simply \( \mathcal{O}(g_1(n) \times g_2(n)) \).

One way to interpret Theorem □ is that it allows us to do the following mathematical manipulation:

\[
f_1(n) = \mathcal{O}(g_1(n)) \quad \Rightarrow \quad f_1(n) \times g_2(n) = \mathcal{O}(g_1(n)) \times g_2(n)
\]

\[
\Rightarrow \quad f_1(n) \times g_2(n) = \mathcal{O}(g_1(n) \times g_2(n)).
\]

That is, Fallacy □ notwithstanding, we can multiply both sides of the "equation" by \( g_2(n) \) and the "equality" still holds. Furthermore, when we multiply \( \mathcal{O}(g_1(n)) \) by \( g_2(n) \), we simply bring the \( g_2(n) \) inside the \( \mathcal{O}(\cdot) \).

The last theorem in this section introduces the transitive property of big oh:

**Theorem (Transitive Property)** If \( f(n) = \mathcal{O}(g(n)) \) and \( g(n) = \mathcal{O}(h(n)) \) then \( f(n) = \mathcal{O}(h(n)) \).

**Proof** By Definition □, there exist two integers, \( n_1 \) and \( n_2 \) and two constants \( c_1 \) and \( c_2 \) such that \( f(n) \leq c_1 g(n) \) for \( n \geq n_1 \) and \( g(n) \leq c_2 h(n) \) for \( n \geq n_2 \).

Let \( n_0 = \max(n_1, n_2) \) and \( c_0 = c_1 c_2 \). Then

\[
f(n) \leq c_1 g(n), \quad n \geq n_1
\]

\[
\leq c_1 c_2 h(n), \quad n \geq n_0
\]

\[
\leq c_0 h(n).
\]

Thus, \( f(n) = \mathcal{O}(h(n)) \).
The transitive property of big oh is useful in conjunction with Theorem □. Consider \( f_1(n) = 5n^3 \) which is clearly \( O(n^3) \). If we add to \( f_1(n) \) the function \( f_2(n) = 3n^2 \), then by Theorem □, the sum \( f_1(n) + f_2(n) \) is \( O(f_1(n)) \) because \( \lim_{n \to \infty} f_2(n)/f_1(n) = 0 \). That is, \( f_1(n) + f_2(n) = O(f_1(n)) \). The combination of the fact that \( f_1(n) = O(n^3) \) and the transitive property of big oh, allows us to conclude that the sum is \( O(n^3) \).
About Polynomials

In this section we examine the asymptotic behavior of polynomials in \( n \). In particular, we will see that as \( n \) gets large, the term involving the highest power of \( n \) will dominate all the others. Therefore, the asymptotic behavior is determined by that term.

**Theorem** Consider a polynomial in \( n \) of the form

\[
f(n) = \sum_{i=0}^{m} a_i n^i
\]

where \( a_m > 0 \). Then \( f(n) = \mathcal{O}(n^m) \).

**Proof** Each of the terms in the summation is of the form \( a_i n^i \). Since \( n \) is non-negative, a particular term will be negative only if \( a_i \leq 0 \). Hence, for each term in the summation, \( a_i n^i \leq |a_i| n^i \). Recall too that we have stipulated that the coefficient of the largest power of \( n \) is positive, i.e., \( a_m > 0 \).

\[
f(n) \leq \sum_{i=0}^{m} |a_i| n^i
\]

\[
\leq n^m \sum_{i=0}^{m} |a_i| n^{i-m}, \quad n \geq 1
\]

\[
\leq n^m \sum_{i=0}^{m} |a_i| \frac{1}{n^{m-i}}.
\]

Note that for integers \( n \geq 1, 1/(n^{m-i}) \leq 1 \) for \( 0 \leq i \leq m \). Thus
About Polynomials

\[ f(n) \leq n^m \sum_{i=0}^{m} |a_i|, \quad n \geq 1 \quad (3.1) \]

From Equation (3.1) we see that we have found the constants \( n_0 = 1 \) and \( c = \sum_{i=0}^{m} |a_i| \), such that for all \( n \geq n_0 \), \( f(n) = \sum_{i=0}^{m} a_i n^m \leq cn^m \). Thus, \( f(n) = O(n^m) \).

This property of the asymptotic behavior of polynomials is used extensively. In fact, whenever we have a function, which is a polynomial in \( n \), \( f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_2 n^2 + a_1 n + a_0 \) we will immediately ``drop'' the less significant terms (i.e., terms involving powers of \( n \) which are less than \( m \)), as well as the leading coefficient, \( a_m \), to write \( f(n) = O(n^m) \).
About Logarithms

In this section we determine the asymptotic behavior of logarithms. Interestingly, despite the fact that \( \log n \) diverges as \( n \) gets large, \( \log n < n \) for all integers \( n \geq 0 \). Hence, \( \log n = \mathcal{O}(n) \). Furthermore, as the following theorem will show, \( \log n \) raised to any integer power \( k \geq 1 \) is still \( O(n) \).

**Theorem** For every integer \( k \geq 1 \), \( \log^k n = \mathcal{O}(n) \).

**Proof** This result follows immediately from Theorem and the observation that for all integers \( k \geq 1 \),

\[
\lim_{n \to \infty} \frac{\log^k n}{n} = 0.
\]

This observation can be proved by induction as follows:

**Base Case** Consider the limit

\[
\lim_{n \to \infty} \frac{\log n}{n}
\]

for the case \( k=1 \). Using L'Hôpital's rule we see that

\[
\lim_{n \to \infty} \frac{\log n}{n} = \lim_{n \to \infty} \frac{1}{n} \cdot \frac{1}{\ln 10} = 0
\]

**Inductive Hypothesis** Assume that Equation holds for \( k = 1, 2, \ldots, m \). Consider the case \( k=m+1 \). Using L'Hôpital's rule we see that
Therefore, by induction on \( m \), Equation holds for all integers \( k \geq 1 \).

For example, using this property of logarithms together with the rule for determining the asymptotic behavior of the product of two functions (Theorem), we can determine that since \( \log n = O(n) \), then \( n \log n = O(n^2) \).
Tight Big Oh Bounds

Big oh notation characterizes the asymptotic behavior of a function by providing an upper bound on the rate at which the function grows as \( n \) gets large. Unfortunately, the notation does not tell us how close the actual behavior of the function is to the bound. That is, the bound might be very close (tight) or it might be overly conservative (loose).

The following definition tells us what makes a bound tight, and how we can test to see whether a given asymptotic bound is the best one available.

**Definition (Tightness)** Consider a function \( f(n) = O(g(n)) \). If for every function \( h(n) \) such that \( f(n) = O(h(n)) \) it is also true that \( g(n) = O(h(n)) \), then we say that \( g(n) \) is a tight asymptotic bound on \( f(n) \).

For example, consider the function \( f(n) = 8n + 128 \). In Section [n], it was shown that \( f(n) = O(n^2) \). However, since \( f(n) \) is a polynomial in \( n \), Theorem [n] tells us that \( f(n) = O(n) \). Clearly \( O(n) \) is a tighter bound on the asymptotic behavior of \( f(n) \) than is \( O(n^2) \).

By Definition [n], in order to show that \( g(n) = n \) is a tight bound on \( f(n) \), we need to show that for every function \( h(n) \) such that \( f(n) = O(h(n)) \), it is also true that \( g(n) = O(h(n)) \).

We will show this result using proof by contradiction: Assume that \( g(n) \) is *not* a tight bound for \( f(n) = 8n + 128 \). Then there exists a function \( h(n) \) such that \( f(n) = 8n + 128 = O(h(n)) \), but for which \( g(n) \neq O(h(n)) \). Since \( 8n + 128 = O(h(n)) \), by the definition of big oh there exist positive constants \( c \) and \( n_0 \) such that \( 8n + 128 \leq ch(n) \) for all \( n \geq n_0 \).

Clearly, for all \( n \geq 0, n \leq 8n + 128 \). Therefore, \( g(n) \leq ch(n) \). But then, by the definition of big oh, we have the \( g(n) = O(h(n)) \)---a contradiction! Therefore, the bound \( f(n) = O(n) \) is a tight bound.
More Big Oh Fallacies and Pitfalls

The purpose of this section is to dispel some common misconceptions about big oh. The next fallacy is related to the selection of the constants $c$ and $n_0$ used to show a big oh relation.

**Fallacy**  Consider non-negative functions $f(n)$, $g_1(n)$, and $g_2(n)$, such that $f(n) = g_1(n) \times g_2(n)$. Since $f(n) \leq cg_1(n)$ for all integers $n \geq 0$ if $c = g_2(n)$, then by Definition $f(n) = O(g_1(n))$.

This fallacy often results from the following line of reasoning: Consider the function $f(n) = n \log n$. Let $c = \log n$ and $n_0 = 1$. Then $f(n)$ must be $O(n)$, since $f(n) \leq cn$ for all $n \geq n_0$. However, this line of reasoning is false because according to Definition, $c$ must be a positive constant, not a function of $n$.

The next fallacy involves a misunderstanding of the notion of the **asymptotic upper bound**.

**Fallacy**  Given non-negative functions $f_1(n)$, $f_2(n)$, $g_1(n)$, and $g_2(n)$, such that $f_1(n) = O(g_1(n))$, $f_2(n) = O(g_2(n))$, and for all integers $n \geq 0$, $g_1(n) < g_2(n)$, then $f_1(n) < f_2(n)$.

This fallacy arises from the following line of reasoning. Consider the function $f_1(n) = O(n^2)$ and $f_2(n) = O(n^3)$. Since $n^2 \leq n^3$ for all values of $n \geq 1$, we might be tempted to conclude that $f_1(n) \leq f_2(n)$. In fact, such a conclusion is erroneous. For example, consider $f_1(n) = n$ and $f_2(n) = n^2 + 1$. Clearly, the former is $O(n^2)$ and the latter is $O(n^3)$. Clearly too, $f_2(n) \geq f_1(n)$ for all values of $n \geq 0$!

The previous fallacy essentially demonstrates that while we may know how the asymptotic upper bounds on two functions are related, we don't necessarily know, in general, the relative behavior of the two bounded functions.
This fallacy often arises in the comparison of the performance of algorithms. Suppose we are comparing two algorithms, $A$ and $B$, to solve a given problem and we have determined that the running times of these algorithms are $T_A(n) = \Theta(g_1(n))$ and $T_B(n) = \Theta(g_2(n))$, respectively. Fallacy demonstrates that it is an error to conclude from the fact that $g_1(n) \preceq g_2(n)$ for all $n \geq 0$ that algorithm $A$ will solve the problem faster than algorithm $B$ for all problem sizes.

But what about any one specific problem size? Can we conclude that for a given problem size, say $n_0$, that algorithm $A$ is faster than algorithm $B$? The next fallacy addresses this issue.

**Fallacy** Given non-negative functions $f_1(n), f_2(n), g_1(n),$ and $g_2(n)$, such that $f_1(n) = \Theta(g_1(n))$, $f_2(n) = \Theta(g_2(n))$, and for all integers $n \geq 0, g_1(n) < g_2(n)$, there exists an integer $n_0$ for which then $f_1(n_0) < f_2(n_0)$.

This fallacy arises from a similar line of reasoning as the preceding one. Consider the function $f_1(n) = \Theta(n^2)$ and $f_2(n) = \Theta(n^3)$. Since $n^2 \preceq n^3$ for all values of $n \geq 1$, we might be tempted to conclude that there exists a value $n_0$ for which $f_1(n_0) \preceq f_2(n_0)$. Such a conclusion is erroneous. For example, consider $f_1(n) = n^2$ and $f_2(n) = n + 1$. Clearly, the former is $\Theta(n^2)$ and the latter is $\Theta(n^3)$. Clearly too, since $f_2(n) \succeq f_1(n)$ for all values of $n \geq 0$, there does not exist any value $n_0 \geq 0$ for which $f_1(n_0) \preceq f_2(n_0)$.

The final fallacy shows that not all functions are *commensurate*:

**Fallacy** Given two non-negative functions $f(n)$ and $g(n)$ then either $f(n)=O(g(n))$ or $g(n)=O(f(n))$.

This fallacy arises from thinking that the relation $O(\cdot)$ is like $\preceq$ and can be used to compare any two functions. However, not all functions are commensurate. Consider the following functions:

\[
\begin{align*}
f(n) & = \begin{cases} 
    n & \text{n is even}, \\
    0 & \text{n is odd}.
\end{cases} \\
g(n) & = \begin{cases} 
    0 & \text{n is even}, \\
    n & \text{n is odd}.
\end{cases}
\end{align*}
\]
Clearly, there does not exist a constant $c$ for which $f(n) \leq c g(n)$ for any even integer $n$, since the $g(n)$ is zero and $f(n)$ is not. Conversely, there does not exist a constant $c$ for which $g(n) \leq c f(n)$ for any odd integer $n$, since the $f(n)$ is zero and $g(n)$ is not. Hence, neither $f(n)=O(g(n))$ nor $g(n)=O(f(n))$ is true.
Conventions for Writing Big Oh Expressions

Certain conventions have evolved which concern how big oh expressions are normally written:

- First, it is common practice when writing big oh expressions to drop all but the most significant terms. Thus, instead of $\mathcal{O}(n^2 + n \log n + n)$ we simply write $\mathcal{O}(n^2)$.
- Second, it is common practice to drop constant coefficients. Thus, instead of $\mathcal{O}(3n^2)$, we simply write $\mathcal{O}(n^2)$. As a special case of this rule, if the function is a constant, instead of, say $\mathcal{O}(1024)$, we simply write $O(1)$.

Of course, in order for a particular big oh expression to be the most useful, we prefer to find a tight asymptotic bound (see Definition). For example, while it is not wrong to write $f(n) = n = \mathcal{O}(n^3)$, we prefer to write $f(n)=O(n)$, which is a tight bound.

Certain big oh expressions occur so frequently that they are given names. Table lists some of the commonly occurring big oh expressions and the usual name given to each of them.

<table>
<thead>
<tr>
<th>expression</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{O}(1)$</td>
<td>constant</td>
</tr>
<tr>
<td>$\mathcal{O}(\log n)$</td>
<td>logarithmic</td>
</tr>
<tr>
<td>$\mathcal{O}(\log^2 n)$</td>
<td>log squared</td>
</tr>
<tr>
<td>$\mathcal{O}(n)$</td>
<td>linear</td>
</tr>
<tr>
<td>$\mathcal{O}(n \log n)$</td>
<td>$n \log n$</td>
</tr>
<tr>
<td>$\mathcal{O}(n^2)$</td>
<td>quadratic</td>
</tr>
</tbody>
</table>
Table: The names of common big oh expressions.

<table>
<thead>
<tr>
<th>$O(n^3)$</th>
<th>cubic</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(2^n)$</td>
<td>exponential</td>
</tr>
</tbody>
</table>

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
An Asymptotic Lower Bound-Omega

The big oh notation introduced in the preceding section is an asymptotic upper bound. In this section, we introduce a similar notation for characterizing the asymptotic behavior of a function, but in this case it is a lower bound.

**Definition (Omega)**  Consider a function \( f(n) \) which is non-negative for all integers \( n \geq 0 \). We say that \"\( f(n) \) is omega \( g(n) \),\" which we write \( f(n) = \Omega(g(n)) \), if there exists an integer \( n_0 \) and a constant \( c > 0 \) such that for all integers \( n \geq n_0, f(n) \geq cg(n) \).

The definition of omega is almost identical to that of big oh. The only difference is in the comparison---for big oh it is \( f(n) \leq cg(n) \); for omega, it is \( f(n) \geq cg(n) \). All of the same conventions and caveats apply to omega as they do to big oh.

- A Simple Example
- About Polynomials Again

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
A Simple Example

Consider the function $f(x) = 5n^2 - 64n + 256$ which is shown in Figure □. Clearly, $f(n)$ is non-negative for all integers $n \geq 0$. We wish to show that $f(n) = \Omega(n^2)$. According to Definition □, in order to show this we need to find an integer $n_0$ and a constant $c > 0$ such that for all integers $n \geq n_0$, $f(n) \geq cn^2$.

As with big oh, it does not matter what the particular constants are—as long as they exist! For example, suppose we choose $c=1$. Then

$$f(n) \geq cn^2 \quad \Rightarrow \quad 5n^2 - 64n + 256 \geq n^2$$
$$\Rightarrow \quad 4n^2 - 64n + 256 \geq 0$$
$$\Rightarrow \quad 4(n - 8)^2 \geq 0.$$

Since $(n - 8)^2 \geq 0$ for all values of $n \geq 0$, we conclude that $n_0 = 0$.

So, we have that for $c=1$ and $n_0 = 0$, $f(n) \geq cn^2$ for all integers $n \geq n_0$. Hence, $f(n) = \Omega(n^2)$.

Figure □ clearly shows that the function $f(n) = n^2$ is less than the function $f(n) = 5n - 64n + 256$ for all values of $n \geq 0$. Of course, there are many other values of $c$ and $n_0$ that will do. For example, $c=2$ and $n_0 = 16$. 
Figure: Showing that $f(n) = 4n^2 - 64n + 288 = \Omega(n^2)$.
In this section we reexamine the asymptotic behavior of polynomials in \( n \). In Section \( \square \) we showed that \( f(n) = \mathcal{O}(n^m) \). That is, \( f(n) \) grows asymptotically no more quickly than \( n^m \). This time we are interested in the asymptotic lower bound rather than the asymptotic upper bound. We will see that as \( n \) gets large, the term involving \( n^m \) also dominates the lower bound in the sense that \( f(n) \) grows asymptotically as quickly as \( n^m \). That is, \( f(n) = \Omega(n^m) \).

**Theorem** Consider a polynomial in \( n \) of the form

\[
f(n) = \sum_{i=0}^{m} a_i n^i
\]

where \( a_m > 0 \). Then \( f(n) = \Omega(n^m) \).

**Proof** We begin by taking the term \( a_m n^m \) out of the summation:

\[
f(n) = \sum_{i=0}^{m} a_i n^i
\]

\[
= a_m n^m + \sum_{i=0}^{m-1} a_i n^i.
\]

Since, \( n \) is a non-negative integer and \( a_m > 0 \), the term \( a_m n^m \) is positive. For each of the remaining terms in the summation, \( a_i n^i \geq -|a_i| n^i \). Hence
\[ f(n) \geq a_m n^m - \sum_{i=0}^{m-1} |a_i| n^i \]
\[ \geq a_m n^m - n^{m-1} \sum_{i=0}^{m-1} |a_i| n^{i-(m-1)}, \quad n \geq 1 \]
\[ \geq a_m n^m - n^{m-1} \sum_{i=0}^{m-1} |a_i| \frac{1}{n^{(m-1)-i}}. \]

Note that for integers \( n \geq 1, 1/(n^{(m-1)-i}) \leq 1 \) for \( 0 \leq i \leq (m-1) \). Thus

\[ f(n) \geq a_m n^m - n^{m-1} \sum_{i=0}^{m-1} |a_i|, \quad n \geq 1 \]
\[ \geq n^m \left( a_m - \frac{1}{n} \sum_{i=0}^{m-1} |a_i| \right). \]

Consider the term in parentheses on the right. What we need to do is to find a positive constant \( c \) and an integer \( n_0 \) so that for all integers \( n \geq n_0 \) this term is greater than or equal to \( c \): \[
a_m - \frac{1}{n} \sum_{i=0}^{m-1} |a_i| \geq a_m - \frac{1}{n_0} \sum_{i=0}^{m-1} |a_i| \]

We choose the value \( n_0 \) for which the term is greater than zero:
\[
a_m - \frac{1}{n_0} \sum_{i=0}^{m-1} |a_i| > 0 \]
\[ n_0 > \frac{1}{a_m} \sum_{i=0}^{m-1} |a_i| \]

The value \( n_0 = \left\lceil \frac{1}{a_m} \sum_{i=0}^{m-1} |a_i| \right\rceil + 1 \) will suffice! Thus
From Equation we see that we have found the constants \( n_0 \) and \( c \), such that for all \( n \geq n_0 \),
\[
f(n) = \sum_{i=0}^{m} a_i n^i \geq cn^m.
\]
Thus, \( f(n) = \Omega(n^m) \).

This property of the asymptotic behavior of polynomials is used extensively. In fact, whenever we have a function, which is a polynomial in \( n \), \( f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_2 n^2 + a_1 n + a_0 \) we will immediately "drop" the less significant terms (i.e., terms involving powers of \( n \) which are less than \( m \)), as well as the leading coefficient, \( a_m \), to write \( f(n) = \Omega(n^m) \).
More Notation-Theta and Little Oh

This section presents two less commonly used forms of asymptotic notation. They are:

- A notation, $\Theta(\cdot)$, to describe a function which is both $O(g(n))$ and $\Omega(g(n))$, for the same $g(n)$.
  (Definition □).
- A notation, $o(\cdot)$, to describe a function which is $O(g(n))$ but not $\Theta(g(n))$, for the same $g(n)$.
  (Definition □).

**Definition (Theta)** Consider a function $f(n)$ which is non-negative for all integers $n \geq 0$. We say that ``$f(n)$ is theta $g(n)$,'' which we write $f(n) = \Theta(g(n))$, if and only if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Recall that we showed in Section □ that a polynomial in $n$, say $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_2 n^2 + a_1 n + a_0$ is $O(n^m)$. We also showed in Section □ that a such a polynomial is $\Omega(n^m)$. Therefore, according to Definition □, we will write $f(n) = \Theta(n^m)$.

**Definition (Little Oh)** Consider a function $f(n)$ which is non-negative for all integers $n \geq 0$. We say that ``$f(n)$ is little oh $g(n)$,'' which we write $f(n) = o(g(n))$, if and only if $f(n)$ is $O(g(n))$ but $f(n)$ is not $\Theta(g(n))$.

Little oh notation represents a kind of loose asymptotic bound in the sense that if we are given that $f(n) = o(g(n))$, then we know that $g(n)$ is an asymptotic upper bound since $f(n) = O(g(n))$, but $g(n)$ is not an asymptotic lower bound since $f(n) = O(g(n))$ and $f(n) \neq \Theta(g(n))$ implies that $f(n) \neq \Omega(g(n))$.

For example, consider the function $f(n) = n + 1$. Clearly, $f(n) = O(n^2)$. Clearly too, $f(n) \neq \Omega(n^2)$, since no matter what $c$ we choose, for large enough $n$, $cn^2 \geq n + 1$. Thus, we may write
\[ f(n) = n + 1 = \Theta(n^2). \]
Asymptotic Analysis of Algorithms

The previous chapter presents a detailed model of the computer which involves a number of different timing parameters—$T_{fetch}$, $T_{store}$, $T_{+}$, $T_{-}$, $T_{\times}$, $T_{\div}$, $T_{<}$, $T_{call}$, $T_{return}$, $T_{new}$, and $T_{[]}$. We show that keeping track of the details is messy and tiresome. So we simplify the model by measuring time in clock cycles, and by assuming that each of the parameters is equal to one cycle. Nevertheless, keeping track of and carefully counting all the cycles is still a tedious task.

In this chapter we introduce the notion of asymptotic bounds, principally big oh, and examine the properties of such bounds. As it turns out, the rules for computing and manipulating big oh expressions greatly simplify the analysis of the running time of a program when all we are interested in is its asymptotic behavior.

For example, consider the analysis of the running time of Program 1, which is just Program 1 again, an algorithm to evaluate a polynomial using Horner's rule.

```
1 public class Example
2 {
3     public static int Horner(int[] a, int n, int x)
4         {
5             int result = a[n];
6             for (int i = n - 1; i >= 0; --i)
7                 result = result * x + a[i];
8             return result;
9         }
10 }
```

Program: Program 1 again.

---

<table>
<thead>
<tr>
<th>statement</th>
<th>detailed model</th>
<th>simple</th>
<th>big oh</th>
</tr>
</thead>
</table>

---

上記の文脈に基づいて、この文書の重要な点は、計算機モデルの詳細なモデルを簡略化することにより、プログラムの実行時間の解析が著しく簡略化されることが示されている。それに伴い、大オーダーの理論を導入し、その性質を検討することで、関心が持たれているのはそれを近似するものである。”

Asymptotic Analysis of Algorithms

The previous chapter presents a detailed model of the computer which involves a number of different timing parameters—$T_{fetch}$, $T_{store}$, $T_{+}$, $T_{-}$, $T_{\times}$, $T_{\div}$, $T_{<}$, $T_{call}$, $T_{return}$, $T_{new}$, and $T_{[]}$. We show that keeping track of the details is messy and tiresome. So we simplify the model by measuring time in clock cycles, and by assuming that each of the parameters is equal to one cycle. Nevertheless, keeping track of and carefully counting all the cycles is still a tedious task.

In this chapter we introduce the notion of asymptotic bounds, principally big oh, and examine the properties of such bounds. As it turns out, the rules for computing and manipulating big oh expressions greatly simplify the analysis of the running time of a program when all we are interested in is its asymptotic behavior.

For example, consider the analysis of the running time of Program 1, which is just Program 1 again, an algorithm to evaluate a polynomial using Horner's rule.

```
1 public class Example
2 {
3     public static int Horner(int[] a, int n, int x)
4         {
5             int result = a[n];
6             for (int i = n - 1; i >= 0; --i)
7                 result = result * x + a[i];
8             return result;
9         }
10 }
```

Program: Program 1 again.

---

<table>
<thead>
<tr>
<th>statement</th>
<th>detailed model</th>
<th>simple</th>
<th>big oh</th>
</tr>
</thead>
</table>

---

上記の文脈に基づいて、この文書の重要な点は、計算機モデルの詳細なモデルを簡略化することにより、プログラムの実行時間の解析が著しく簡略化されることが示されている。それに伴い、大オーダーの理論を導入し、その性質を検討することで、関心が持たれているのはそれを近似するものである。”
Asymptotic Analysis of Algorithms

Table: Computing the running time of Program □.

Table □ shows the running time analysis of Program □ done in three ways—a detailed analysis, a simplified analysis, and an asymptotic analysis. In particular, note that all three methods of analysis are in agreement: Lines 5, 6a, and 8 execute in a constant amount of time; 6b, 6c, and 7 execute in an amount of time which is proportional to \( n \), plus a constant.

The most important observation to make is that, regardless of what the actual constants are, the asymptotic analysis always produces the same answer! Since the result does not depend upon the values of the constants, the asymptotic bound tells us something fundamental about the running time of the algorithm. And this fundamental result does not depend upon the characteristics of the computer and compiler actually used to execute the program!

Of course, you don't get something for nothing. While the asymptotic analysis may be significantly easier to do, all that we get is an upper bound on the running time of the algorithm. In particular, we know nothing about the actual running time of a particular program. (Recall Fallacies □ and □).

- Rules For Big Oh Analysis of Running Time
- Example-Prefix Sums
- Example-Fibonacci Numbers
Asymptotic Analysis of Algorithms

- Example-Bucket Sort
- Reality Check
- Checking Your Analysis

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Rules For Big Oh Analysis of Running Time

In this section we present some simple rules for determining a big-oh upper bound on the running time of the basic compound statements in a C# program.

**Rule 3.1 (Sequential Composition)** The worst-case running time of a sequence of C# statements such as

\[
S_1; \quad S_2; \quad \ldots; \quad S_m;
\]

is \(O(\max(T_1(n), T_2(n), \ldots, T_m(n)))\), where the running time of \(S_i\), the \(i^{th}\) statement in the sequence, is \(O(T_i(n))\).

Rule 3.1 follows directly from Theorem 3.1. The total running time of a sequence of statements is equal to the sum of the running times of the individual statements. By Theorem 3.1, when computing the sum of a series of functions it is the largest one (the \(\max\)) that determines the bound.

**Rule 3.2 (Iteration)** The worst-case running time of a C# for loop such as

\[
\text{for } (S_1; S_2; S_3) \quad S_4;
\]

is \(O\left(\max(T_1(n), T_2(n) \times (I(n) + 1), T_3(n) \times I(n), T_4(n) \times I(n))\right)\), where the running time of statement \(S_i\) is \(O(T_i(n))\), for \(i = 1, 2, 3,\) and 4, and \(I(n)\) is the number of iterations executed in the worst case.

Rule 3.2 appears somewhat complicated due to the semantics of the C# for statement. However, it follows directly from Theorem 3.2. Consider the following simple counted do loop.

\[
\text{for } (\text{int } i = 0; i < n; ++i) \quad S_4;
\]
Here $S_1$ is `int i = 0`, so its running time is constant ($T_1(n) = 1$); $S_2$ is `i < n`, so its running time is constant ($T_2(n) = 1$); and $S_3$ is `++i`, so its running time is constant ($T_3(n) = 1$). Also, the number of iterations is $I(n) = n$. According to Rule 1, the running time of this is $O(\max(1, 1 \times (n + 1), 1 \times n, T_4(n) \times n))$, which simplifies to $O(\max(n, T_4(n) \times n))$.

Furthermore, if the loop body *does anything at all*, its running time must be $T_4(n) = \Omega(1)$. Hence, the loop body will dominate the calculation of the maximum, and the running time of the loop is simply $O(n \times T_4(n))$.

If we don't know the exact number of iterations executed, $I(n)$, we can still use Rule 2 provided we have an upper bound, $I(n) = O(f(n))$, on the number of iterations executed. In this case, the running time is $O(\max(T_1(n), T_2(n) \times (f(n) + 1), T_3(n) \times f(n), T_4(n) \times f(n)))$.

**Rule 3.3 (Conditional Execution)** The worst-case running time of a C# if-then-else such as

```csharp
if (S_1)
    S_2;
else
    S_3;
```

is $O(\max(T_1(n), T_2(n), T_3(n)))$, where the running time of statement $S_i$ is $O(T_i(n))$, for $i = 1, 2, 3$.

Rule 2 follows directly from the observation that the total running time for an if-then-else statement will never exceed the sum of the running time of the conditional test, $S_1$, plus the larger of the running times of the *then part*, $S_2$, and the *else part*, $S_3$. 

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Example-Prefix Sums

In this section, we will determine a tight big-oh bound on the running time of a program to compute the series of sums $S_0, S_1, ..., S_{n-1}$, where

$$S_j = \sum_{i=0}^{j} a_i.$$

An algorithm to compute this series of summations is given in Program □. Table □ summarizes the running time calculation.

```csharp
public class Example
{
    public static void PrefixSums(int[] a, int n)
    {
        for (int j = n - 1; j >= 0; --j)
        {
            int sum = 0;
            for (int i = 0; i <= j; ++i)
            {
                sum += a[i];
                a[j] = sum;
            }
        }
    }
}
```

Program: Program to compute $\sum_{i=0}^{j} a_i$ for $0 \leq j < n$. 

<table>
<thead>
<tr>
<th>statement</th>
<th>time</th>
</tr>
</thead>
</table>
Table: Computing the running time of Program 

Usually the easiest way to analyze a program which contains nested loops is to start with the body of the inner-most loop. In Program, the inner-most loop comprises lines 8 and 9. In all, a constant amount of work is done—this includes the loop body (line 9), the conditional test (line 8b), and the incrementing of the loop index (line 8c).

For a given value of \( j \), the inner-most loop is done a total \( j+1 \) times. And since the outer loop is done for \( j = n - 1, n - 2, \ldots, 0 \), in the worst case, the inner-most loop is done \( n \) times. Therefore, the contribution of the inner loop to the running time of one iteration of the outer loop is \( O(n) \).

The rest of the outer loop (lines 5, 7 and 10) does a constant amount of work in each iteration. This constant work is dominated by the \( O(n) \) of the inner loop. The outer loop is done exactly \( n \) iterations. Therefore, the total running time of the program is \( O(n^2) \).

But is this a tight big oh bound? We might suspect that it is not, because of the worst-case assumption we made in the analysis concerning the number of times the inner loop is executed. The inner-most loop is done exactly \( j+1 \) times for \( j = n - 1, n - 2, \ldots, 0 \). However, we did the calculation assuming the inner loop is done \( O(n) \) times, in each iteration of the outer loop. Unfortunately, in order to determine whether our answer is a tight bound, we must determine more precisely the actual running time of the program.
However, there is one approximate calculation that we can easily make. If we observe that the running time will be dominated by the work done in the inner-most loop, and that the work done in one iteration of the inner-most loop is constant, then all we need to do is to determine exactly the number of times the inner loop is actually executed. This is given by:

\[
\sum_{j=0}^{n-1} j + 1 = \sum_{j=1}^{n} j = \frac{n(n + 1)}{2} = \Theta(n^2)
\]

Therefore, the result \( T(n) = O(n^2) \) is a tight, big-oh bound on the running time of Program.
Example-Fibonacci Numbers

In this section we will compare the asymptotic running times of two different programs that both compute Fibonacci numbers. The Fibonacci numbers are the series of numbers $F_0, F_1, \ldots$, given by

$$
F_n = \begin{cases} 
0 & n = 0, \\
1 & n = 1, \\
F_{n-1} + F_{n-2} & n \geq 2. 
\end{cases}  \tag{3.4}
$$

Fibonacci numbers are interesting because they seem to crop up in the most unexpected situations. However, in this section, we are merely concerned with writing an algorithm to compute $F_n$ given $n$.

Fibonacci numbers are easy enough to compute. Consider the sequence of Fibonacci numbers

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

The next number in the sequence is computed simply by adding together the last two numbers—in this case it is $55=21+34$. Program [ ] is a direct implementation of this idea. The running time of this algorithm is clearly $O(n)$ as shown by the analysis in Table [ ].
public class Example {
    public static int Fibonacci(int n) {
        int previous = -1;
        int result = 1;
        for (int i = 0; i <= n; ++i) {
            int sum = result + previous;
            previous = result;
            result = sum;
        }
        return result;
    }
}

Program: Non-recursive program to compute Fibonacci numbers.

<table>
<thead>
<tr>
<th>statement</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>6</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>7a</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>7b</td>
<td>$\mathcal{O}(1) \times (n + 2)$ iterations</td>
</tr>
<tr>
<td>7c</td>
<td>$\mathcal{O}(1) \times (n + 1)$ iterations</td>
</tr>
<tr>
<td>9</td>
<td>$\mathcal{O}(1) \times (n + 1)$ iterations</td>
</tr>
<tr>
<td>10</td>
<td>$\mathcal{O}(1) \times (n + 1)$ iterations</td>
</tr>
<tr>
<td>11</td>
<td>$\mathcal{O}(1) \times (n + 1)$ iterations</td>
</tr>
<tr>
<td>13</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>TOTAL</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Table: Computing the running time of
Recall that the Fibonacci numbers are defined recursively: \( F_n = F_{n-1} + F_{n-2} \). However, the algorithm used in Program 1 is non-recursive—it is *iterative*. What happens if instead of using the iterative algorithm, we use the definition of Fibonacci numbers to implement directly a recursive algorithm? Such an algorithm is given in Program 2 and its running time is summarized in Table 3.

```java
public class Example {
    public static int Fibonacci(int n) {
        if (n == 0 || n == 1) {
            return n;
        } else {
            return Fibonacci(n - 1) + Fibonacci(n - 2);
        }
    }
}
```

**Program:** Recursive program to compute Fibonacci numbers.

<table>
<thead>
<tr>
<th>statement</th>
<th>( n &lt; 2 )</th>
<th>( n \geq 2 )</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0(1)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0(1)</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>--</td>
<td>( T(n-1) + T(n-2) + O(1) )</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>0(1)</td>
<td>( T(n-1) + T(n-2) + O(1) )</td>
<td></td>
</tr>
</tbody>
</table>

**Table:** Computing the running time of Program 1.

From Table 3 we find that the running time of the recursive Fibonacci algorithm is given by the recurrence
Example-Fibonacci Numbers

But how do you solve a recurrence containing big oh expressions?

It turns out that there is a simple trick we can use to solve a recurrence containing big oh expressions as long as we are only interested in an asymptotic bound on the result. Simply drop the $\mathcal{O}(\cdot)$s from the recurrence, solve the recurrence, and put the $\mathcal{O}(\cdot)$ back! In this case, we need to solve the recurrence

$$T(n) = \begin{cases} 
\mathcal{O}(1) & n < 2, \\
T(n - 1) + T(n - 2) + \mathcal{O}(1) & n \geq 2.
\end{cases}$$

In the previous chapter, we used successfully repeated substitution to solve recurrences. However, in the previous chapter, all of the recurrences only had one instance of $T(\cdot)$ on the right-hand-side--in this case there are two. As a result, repeated substitution won't work.

There is something interesting about this recurrence: It looks very much like the definition of the Fibonacci numbers. In fact, we can show by induction on $n$ that $T(n) \geq F_{n+1}$ for all $n \geq 0$.

Proof (By induction).

**Base Case** There are two base cases:

- $T(0) = 1, \quad F_1 = 1 \quad \Rightarrow \quad T(0) \geq F_1, \quad \text{and}$
- $T(1) = 1, \quad F_2 = 1 \quad \Rightarrow \quad T(1) \geq F_2.$

**Inductive Hypothesis** Suppose that $T(n) \geq F_{n+1}$ for $n = 0, 1, 2, \ldots, k$ for some $k \geq 1$. Then

$$T(k + 1) = T(k) + T(k - 1) + 1 \geq F_{k+1} + F_k + 1 \geq F_{k+2} + 1 \geq F_{k+2}.$$

Hence, by induction on $k$, $T(n) \geq F_{n+1}$ for all $n \geq 0$.

So, we can now say with certainty that the running time of the recursive Fibonacci algorithm, Program
Example-Fibonacci Numbers

is $T(n) = \Omega(F_{n+1})$. But is this good or bad? The following theorem shows us how bad this really is!

**Theorem (Fibonacci numbers)** The Fibonacci numbers are given by the closed form expression

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n) \quad (3.5)$$

where $\phi = (1 + \sqrt{5})/2$ and $\hat{\phi} = (1 - \sqrt{5})/2$.

**Proof (By induction).**

**Base Case** There are two base cases:

$$F_0 = \frac{1}{\sqrt{5}}(\phi^0 - \hat{\phi}^0) = 0$$

$$F_1 = \frac{1}{\sqrt{5}}(\phi^1 - \hat{\phi}^1) = \frac{1}{\sqrt{5}}((1 + \sqrt{5})/2 - (1 - \sqrt{5})/2) = 1$$

**Inductive Hypothesis** Suppose that Equation holds for $n = 0, 1, 2, \ldots, k$ for some $k \geq 1$. First, we make the following observation:

$$\phi^2 = ((1 + \sqrt{5})/2)^2 = 1 + (1 + \sqrt{5})/2 = 1 + \phi.$$  

Similarly,

$$\hat{\phi}^2 = ((1 - \sqrt{5})/2)^2 = 1 + (1 - \sqrt{5})/2 = 1 + \hat{\phi}.$$
Now, we can show the main result:

\[
F_{n+1} = F_n + F_{n-1} \\
= \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n) + \frac{1}{\sqrt{5}}(\phi^{n-1} - \hat{\phi}^{n-1}) \\
= \frac{1}{\sqrt{5}}(\phi^{n-1}(1 + \phi) - \hat{\phi}^{n-1}(1 + \hat{\phi})) \\
= \frac{1}{\sqrt{5}}(\phi^{n-1}\phi^2 - \hat{\phi}^{n-1}\hat{\phi}^2) \\
= \frac{1}{\sqrt{5}}(\phi^{n+1} - \hat{\phi}^{n+1})
\]

Hence, by induction, Equation (1) correctly gives \( F_n \) for all \( n \geq 0 \).

Theorem (2) gives us that \( F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n) \) where \( \phi = (1 + \sqrt{5})/2 \) and \( \hat{\phi} = (1 - \sqrt{5})/2 \).

Consider \( \hat{\phi} \). A couple of seconds with a calculator should suffice to convince you that \( |\hat{\phi}| < 1 \).

Consequently, as \( n \) gets large, \( |\hat{\phi}|^n \) is vanishingly small. Therefore, \( F_n \geq \phi^n - 1 \). In asymptotic terms, we write \( F_n = \Omega(\phi^n) \). Now, since \( \phi \approx 1.62 > (3/2) \), we can write that \( F_n = \Omega((3/2)^n) \).

Returning to Program (3), recall that we have already shown that its running time is \( T(n) = \Omega(F_{n+1}) \).

And since \( F_n = \Omega((3/2)^n) \), we can write that \( T(n) = \Omega((3/2)^{n+1}) = \Omega((3/2)^n) \). That is, the running time of the recursive Fibonacci program grows exponentially with increasing \( n \). And that is really bad in comparison with the linear running time of Program (4)!

Figure (5) shows the actual running times of both the non-recursive and recursive algorithms for computing Fibonacci numbers. Because the largest C# \texttt{int} is 2147483647, it is only possible to compute Fibonacci numbers up to \( F_{46} = 1 836 311 903 \) before overflowing.

The graph shows that up to about \( n=35 \), the running times of the two algorithms are comparable. However, as \( n \) increases past 40, the exponential growth rate of Program (6) is clearly evident. In fact, the actual time taken by Program (7) to compute \( F_{46} \) was in excess of two and a half minutes!
Figure: Actual running times of Programs □ and □.
Example-Bucket Sort

So far all of the asymptotic running time analyses presented in this chapter have resulted in tight big oh bounds. In this section we consider an example which illustrates that a cursory big oh analysis does not always result in a tight bound on the running time of the algorithm.

In this section we consider an algorithm to solve the following problem: Sort an array of \( n \) integers \( a_0, a_1, \ldots, a_{n-1} \), each of which is known to be between 0 and \( m-1 \) for some fixed \( m \). An algorithm for solving this problem, called a bucket sort, is given in Program 1.

```csharp
public class Example
{
    public static void BucketSort(int[] a, int m)
    {
        int[] buckets = new int[m];

        for (int j = 0; j < m; ++j)
            buckets[j] = 0;

        for (int i = 0; i < a.Length; ++i)
            ++buckets[a[i]];

        for (int i = 0, j = 0; j < m; ++j)
            for (int k = buckets[j]; k > 0; --k)
                a[i++] = j;
    }
}
```

Program: Bucket sort.

A bucket sort works as follows: An array of \( m \) counters, or buckets, is used. Each of the counters is set initially to zero. Then, a pass is made through the input array, during which the buckets are used to keep a count of the number of occurrences of each value between 0 and \( m-1 \). Finally, the sorted result is produced by first placing the required number of zeroes in the array, then the required number of ones, followed by the twos, and so on, up to \( m-1 \).
The analysis of the running time of Program is summarized in Table. Clearly, the worst-case running time of the first loop (lines 7-8) is \( O(m) \) and that of the second loop (lines 9-10) is \( O(n) \).

<table>
<thead>
<tr>
<th>statement</th>
<th>cursory analysis</th>
<th>careful analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-8</td>
<td>( O(m) )</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>9-10</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>11-13</td>
<td>( O(mn) )</td>
<td>( O(m+n) )</td>
</tr>
<tr>
<td>TOTAL</td>
<td>( O(mn) )</td>
<td>( O(m+n) )</td>
</tr>
</tbody>
</table>

**Table:** Computing the running time of Program.

Consider nested loops on lines 11-13. Exactly \( m \) iterations of the outer loop are done--the number of iterations of the outer loop is fixed. But the number of iterations of the inner loop depends on \( \text{bucket}[j] \)--the value of the counter. Since there are \( n \) numbers in the input array, in the worst case a counter may have the value \( n \). Therefore, the running time of lines 11-13 is \( O(mn) \) and this running time dominates all the others, so the running time of Program is \( O(mn) \). (This is the *cursory analysis* column of Table.)

Unfortunately, the cursory analysis has not produced a tight bound. To see why this is the case, we must consider the operation of Program more carefully. In particular, since we are sorting \( n \) items, the final answer will only contain \( n \) items. Therefore, line 13 will be executed exactly \( n \) times--not \( mn \) times as the cursory result suggests.

Consider the inner loop at line 12. During the \( j \)th iteration of the outer loop, the inner loop does \( \text{bucket}[j] \) iterations. Therefore, the conditional test at line 12b is done \( \text{bucket}[j] + 1 \) times. Therefore, the total number of times the conditional test is done is

\[
\sum_{j=0}^{m-1} (\text{bucket}[j] + 1) = \sum_{j=0}^{m-1} \text{bucket}[j] + \sum_{j=0}^{m-1} 1
= n + m.
\]
So, the running time of lines 11-13 is $O(m+n)$ and therefore running time of Program is $O(m+n)$. (This is the careful analysis column of Table).
Reality Check

``Asymptotic analysis is nice in theory," you say, "but of what practical value is it when I don't know what $c$ and $n^0$ are?" Fallacies and Fallacies showed us that if we have two programs, A and B, that solve a given problem, whose running times are $T_A = \Theta(n^2)$ and $T_B = \Theta(n^3)$ say, we cannot conclude in general that we should use algorithm A rather than algorithm B to solve a particular instance of the problem. Even if the bounds are both known to be tight, we still don't have enough information. What we do know for sure is that eventually, for large enough $n$, program A is the better choice.

In practice we need not be so conservative. It is almost always the right choice to select program A. To see why this is the case, consider the times shown in Table 1. This table shows the running times computed for a very conservative scenario. We assume that the constant of proportionality, $c$, is one cycle of a 1 GHz clock. This table shows the running times we can expect even if only one instruction is done for each element of the input.

<table>
<thead>
<tr>
<th>$n=1$</th>
<th>$n=8$</th>
<th>$n = 1K$</th>
<th>$n = 1024K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Omega(1)$</td>
<td>1 ns</td>
<td>1 ns</td>
<td>1 ns</td>
</tr>
<tr>
<td>$\Omega(\log n)$</td>
<td>1 ns</td>
<td>3 ns</td>
<td>10 ns</td>
</tr>
<tr>
<td>$\Omega(n)$</td>
<td>1 ns</td>
<td>8 ns</td>
<td>102 ns</td>
</tr>
<tr>
<td>$\Omega(n \log n)$</td>
<td>1 ns</td>
<td>24 ns</td>
<td>1.02 μs</td>
</tr>
<tr>
<td>$\Omega(n^2)$</td>
<td>1 ns</td>
<td>64 ns</td>
<td>10.2 μs</td>
</tr>
<tr>
<td>$\Omega(n^3)$</td>
<td>1 ns</td>
<td>512 ns</td>
<td>1.07 s</td>
</tr>
<tr>
<td>$\Omega(2^n)$</td>
<td>1 ns</td>
<td>256 ns</td>
<td>$10^{25.2}$ years</td>
</tr>
</tbody>
</table>

Table: Actual lower bounds assuming a 1 GHz clock,
\[ c = 1 \text{ cycle and } n_0 = 0. \]
Checking Your Analysis

Having made an asymptotic analysis of the running time of an algorithm, how can you verify that the implementation of the algorithm performs as predicted by the analysis? The only practical way to do this is to conduct an experiment--write out the algorithm in the form of a computer program, compile and execute the program, and measure its actual running time for various values of the parameter, \( n \) say, used to characterize the size of the problem.

However, several difficulties immediately arise:

- How do you compare the results of the analysis which, by definition, only applies asymptotically, i.e., as \( n \) gets arbitrarily large, with the actual running time of a program which, of necessity, must be measured for fixed and finite values of \( n \)?
- How do you explain it when the results of your analysis do not agree with the observed behavior of the program?

Suppose you have conducted an experiment in which you measured the actual running time of a program, \( T(n) \), for a number of different values of \( n \). Furthermore, suppose that on the basis of an analysis of the algorithm you have concluded that the worst-case running time of the program is \( O(f(n)) \). How do you tell from the measurements made that the program behaves as predicted?

One way to do this follows directly from the definition of big oh: there exists \( c > 0 \) such that \( T(n) \leq cf(n) \) for all \( n \geq n_0 \). This suggests that we should compute the ratio \( T(n)/f(n) \) for each of value of \( n \) in the experiment and observe how the ratio behaves as \( n \) increases. If this ratio diverges, then \( f(n) \) is probably too small; if this ratio converges to zero, then \( f(n) \) is probably too big; and if the ratio converges to a constant, then the analysis is probably correct.

What if \( f(n) \) turns out to large? There are several possibilities:

- The function \( f(n) \) is not a tight bound. That is, the analysis is still correct, but the bound is not the tightest bound possible.
- The analysis was for the worst case but the worst case did not arise in the set of experiments conducted.
- A mistake was made and the analysis is wrong.
Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Exercises

1. Consider the function \( f(n) = 3n^2 - n + 4 \). Using Definition show that \( f(n) = O(n^2) \).
2. Consider the function \( f(n) = 3n^2 - n + 4 \). Using Definition show that \( f(n) = \Omega(n^2) \).
3. Consider the functions \( f(n) = 3n^2 - n + 4 \) and \( g(n) = n \log n + 5 \). Using Theorem show that \( f(n) + g(n) = O(n^2) \).
4. Consider the functions \( f(n) = \sqrt{n} \) and \( g(n) = \log n \). Using Theorem show that \( f(n) + g(n) = O(\sqrt{n}) \).
5. For each pair of functions, \( f(n) \) and \( g(n) \), in the following table, indicate if \( f(n) = O(g(n)) \) and if \( g(n) = O(f(n)) \).

<table>
<thead>
<tr>
<th>( f(n) )</th>
<th>( g(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 10n )</td>
<td>( n^2 - 10m )</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>( n^2 \log n )</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>( n + \log n )</td>
</tr>
<tr>
<td>( \log n )</td>
<td>( \sqrt{n} )</td>
</tr>
<tr>
<td>( \ln n )</td>
<td>( \log n )</td>
</tr>
<tr>
<td>( \log(n+1) )</td>
<td>( \log n )</td>
</tr>
<tr>
<td>( \log \log n )</td>
<td>( \log n )</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>( 10^n )</td>
</tr>
<tr>
<td>( n^m )</td>
<td>( m^n )</td>
</tr>
<tr>
<td>( \cos(n \pi/2) )</td>
<td>( \sin(n \pi/2) )</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>( (n \cos n)^2 )</td>
</tr>
</tbody>
</table>
Exercises

6. Show that the Fibonacci numbers (see Equation) satisfy the identities

\[ F_{2n-1} = (F_n)^2 + (F_{n-1})^2 \]
\[ F_{2n} = (F_n)^2 + 2F_nF_{n-1} \]

for \( n \geq 1 \).

7. Prove each of the following formulas:
   1. \( \sum_{i=0}^{n} i = \mathcal{O}(n^2) \)
   2. \( \sum_{i=0}^{n} i^2 = \mathcal{O}(n^3) \)
   3. \( \sum_{i=0}^{n} i^3 = \mathcal{O}(n^4) \)

8. Show that \( \sum_{i=0}^{n} a^i = \mathcal{O}(1) \), where \( 0 \leq a < 1 \) and \( n \geq 0 \).

9. Show that \( \sum_{i=1}^{n} \frac{1}{i} = \mathcal{O}(\log n) \).

10. Solve each of the following recurrences:
   1. \( T(n) = \begin{cases} \mathcal{O}(1) & n = 0, \\ aT(n-1) + \mathcal{O}(1) & n > 0, \ a > 1. \end{cases} \)
   2. \( T(n) = \begin{cases} \mathcal{O}(1) & n = 0, \\ aT(n-1) + \mathcal{O}(n) & n > 0, \ a > 1. \end{cases} \)
   3. \( T(n) = \begin{cases} \mathcal{O}(1) & n = 1, \\ aT([n/a]) + \mathcal{O}(1) & n > 1, \ a \geq 2. \end{cases} \)
   4. \( T(n) = \begin{cases} \mathcal{O}(1) & n = 1, \\ aT([n/a]) + \mathcal{O}(n) & n > 1, \ a \geq 2. \end{cases} \)

11. Derive tight, big oh expressions for the running times of Example-a,Example-b,Example-c,Example-d,Example-f,Example-g,Example-h,Example-i.

12. Consider the C# program fragments given below. Assume that \( n, m, \) and \( k \) are non-negative ints and that the methods \( E, F, G, \) and \( H \) have the following characteristics:
   - The worst case running time for \( E(n, m, k) \) is \( O(1) \) and it returns a value between 1 and \((n+m+k)\).
   - The worst case running time for \( F(n, m, k) \) is \( O(n+m) \).
   - The worst case running time for \( G(n, m, k) \) is \( O(m+k) \).
Exercises

- The worst case running time for \( H(n,m,k) \) is \( O(n+k) \).

Determine a tight, big oh expression for the worst-case running time of each of the following program fragments:

1. \( F(n, 10, 0); \)
   \( G(n, m, k); \)
   \( H(n, m, 1000000); \)

2. \( \text{for (int i = 0; i < n; ++i)} \)
   \( F(n, m, k); \)

3. \( \text{for (int i = 0; i < E(n, 10, 100); ++i)} \)
   \( F(n, 10, 0); \)

4. \( \text{for (int i = 0; i < E(n, m, k); ++i)} \)
   \( F(n, 10, 0); \)

5. \( \text{for (int i = 0; i < n; ++i)} \)
   \( \text{for (int j = i; j < n; ++j)} \)
   \( F(n, m, k); \)

13. Consider the following C# program fragment. What value does \( F \) compute? (Express your answer as a function of \( n \)). Give a tight, big oh expression for the worst-case running time of the method \( F \).

```csharp
public class Example
{
    public static int F(int n)
    {
        int sum = 0;
        for (int i = 1; i <= n; ++i)
            sum = sum + i;
        return sum;
    }
    // ...
}
```

14. Consider the following C# program fragment. (The method \( F \) is given in Exercise 13.) What value does \( G \) compute? (Express your answer as a function of \( n \)). Give a tight, big oh expression for the worst-case running time of the method \( G \).

```csharp
public class Example
{
    // ...
}
```
15. Consider the following C# program fragment. (The method \( F \) is given in Exercise 0 and the method \( G \) is given in Exercise 0). What value does \( H \) compute? (Express your answer as a function of \( n \)). Give a tight, big oh expression for the worst-case running time of the method \( H \).

```csharp
public class Example
{
    // ...
    public int H(int n)
    { return F(n) + G(n); }
}
```
Projects

1. Write a C# method that takes a single integer argument \( n \) and has a worst-case running time of \( O(n) \).
2. Write a C# method that takes a single integer argument \( n \) and has a worst-case running time of \( O(n^2) \).
3. Write a C# method that takes two integer arguments \( n \) and \( k \) and has a worst-case running time of \( O(n^k) \).
4. Write a C# method that takes a single integer argument \( n \) and has a worst-case running time of \( O(\log n) \).
5. Write a C# method that takes a single integer argument \( n \) and has a worst-case running time of \( O(n \log n) \).
6. Write a C# method that takes a single integer argument \( n \) and has a worst-case running time of \( O(2^n) \).
7. The generalized Fibonacci numbers of order \( k \geq 2 \) are given by

\[
F_n^{(k)} = \begin{cases} 
0 & 0 \leq n < k - 1, \\
1 & n = k - 1, \\
\sum_{i=1}^{k} F_{n-i}^{(k)} & n \geq k.
\end{cases}
\]  

(3.6)

Write both recursive and non-recursive methods that compute \( F_n^{(k)} \). Measure the running times of your algorithms for various values of \( k \) and \( n \).
Foundational Data Structures

In this book we consider a variety of abstract data types (ADTs), including stacks, queues, deques, ordered lists, sorted lists, hash and scatter tables, trees, priority queues, sets, and graphs. In just about every case, we have the option of implementing the ADT using an array or using a some kind of linked data structure.

Because they are the base upon which almost all of the ADTs are built, we call the array and the linked list the foundational data structures. It is important to understand that we do not view the array or the linked list as ADTs, but rather as alternatives for the implementation of ADTs.

In this chapter we consider arrays first. We review the support for arrays in C# and then show how to provide arrays with arbitrary subscript ranges, resizeable arrays, multi-dimensional arrays, and matrices. Next, we consider a number of linked list implementation alternatives and we discuss in detail the implementation of a singly-linked list class, LinkedList. It is important to become familiar with this class, as it is used extensively throughout the remainder of the book.
Arrays

Probably the most common way to aggregate data is to use an array. In C# an array is an object that contains a collection of objects, all of the same type. For example,

```csharp
int[] a = new int[5];
```

allocates an array of five integers and assigns it to the variable `a`.

The elements of an array are accessed using integer-valued indices. In C# the first element of an array always has index zero. Thus, the five elements of array `a` are `a[0]`, `a[1]`, ..., `a[4]`. All array objects in C# have an `int` property called `Length`, the value of which is equal to the number of array elements. In this case, `a.Length` has the value `5`.

C# checks at run-time that the index used in every array access is valid. Valid indices fall between zero and `Length - 1`. If an invalid index expression is used, an `IndexOutOfRangeException` exception is thrown.

It is important to understand that in C#, the variable `a` refers to an array object of type `int[]`. In particular, the sequence of statements

```csharp
int[] b;
b = a;
```

causes the variable `b` to refer to the same array object as variable `a`.

Once allocated, the size of a C# array object is fixed. That is, it is not possible to increase or decrease the size of a given array. Of course, it is always possible to allocate a new array of the desired size, but it is up to the programmer to copy the values from the old array to the new one.

How are C# arrays represented in the memory of the computer? The specification of the C# language leaves this up to the system implementers[22]. However, Figure  illustrates a typical implementation scenario.
Arrays

Figure: Memory representation of C# arrays.

The elements of an array typically occupy consecutive memory locations. That way, given $i$ it is possible to find the position of $a[i]$ in constant time. In addition to the array elements, the array object has a Length property, the value of which is represented by an int field called length.

On the basis of Figure, we can now estimate the total storage required to represent an array. Let $S(n)$ be the total storage (memory) needed to represent an array of $n$ ints. $S(n)$ is given by

$$S(n) \geq \text{sizeof(int [n])} \geq (n + 1)\text{sizeof(int)},$$

where the function $\text{sizeof}(X)$ is the number of bytes used for the memory representation of an instance of an object of type $X$.

In C#, the sizes of the simple data types are fixed constants. Hence, $\text{sizeof(int)} = O(1)$. In practice, an array object may contain additional fields. For example, it is reasonable to expect that there is a field which records the position in memory of the first array element. In any event, the overhead associated with a fixed number of fields is $O(1)$. Therefore, $S(n)=O(n)$.

- Extending C# Arrays
- Constructors
- Copy Method
Arrays

- **DynamicArray Indexers**
- **DynamicArray Properties**
- **Resizing an Array**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno
Extending C# Arrays

While the C# programming language does indeed provide built-in support for arrays, that support is not without its shortcomings: Array indices range from zero to n-1, where n is the array length and the size of an array is fixed once allocated.

One way to address these deficiencies is to define a new class with the desired functionality. We do this by defining a DynamicArray class with two fields as shown in Program 1. The first is an array of C# objects and the second is an int which records the lower bound for array indices.

```
public class DynamicArray
{
    protected object [] data;
    protected int baseIndex;

    // ...
}
```

Program: DynamicArray fields.
Constructors

Program gives the code for three DynamicArray class constructors. The main constructor (lines 6-10) takes two arguments, \( n \) and \( m \), which represent the desired array length and the lower bound for array indices, respectively. This constructor allocates an array of objects of length \( n \) and sets the baseIndex field to \( m \). The remaining two constructors (lines 12-16) simply call the main constructor by invoking the this initializer. These constructors simply provide default values for \( m \) and \( n \).

```csharp
public class DynamicArray
{
    protected object[] data;
    protected int baseIndex;

    public DynamicArray(int n, int m)
    {
        data = new object[n];
        baseIndex = m;
    }

    public DynamicArray() : this(0, 0)
    {
    }

    public DynamicArray(int n) : this(n, 0)
    {
    // ...
    }
}
```

Program: DynamicArray constructors.

In C#, when an array is allocated, two things happen. First, memory is allocated for the array object and its elements. Second, each element of the array is initialized with the appropriate default value (in this case null).
For now, we shall assume that the first step takes a constant amount of time. Since there are \( n \) elements to be initialized, the second step takes \( O(n) \) time. Therefore, the running time of the main constructor is \( O(n) \).
Copy Method

Program defines the Copy method of the DynamicArray class. This method provides a way to copy the elements of one array to another. The Copy method is intended to be used like this:

```csharp
DynamicArray a = new DynamicArray(5);
DynamicArray b = new DynamicArray(5);
// ...
b.Copy(a);
```

The effect of doing this is to copy the elements of array a to the elements of array b. Note that after the copy, a and b still refer to distinct DynamicArray instances.

Program shows a simple implementation of the Copy method. To determine its running time, we need to consider carefully the execution of this method.
Copy Method

```java
public class DynamicArray
{
    protected object[] data;
    protected int baseIndex;

    public void Copy(DynamicArray array)
    {
        if (array != this)
        {
            if (data.Length != array.data.Length)
                data = new object [array.data.Length];
            for (int i = 0; i < data.Length; ++i)
                data[i] = array.data[i];
            baseIndex = array.baseIndex;
        }
        // ...
    }
}
```

**Program:** DynamicArray class Copy method.

First, we observe that the Copy method detects and avoids self-copies. That is, the special case

```java
a.Copy(a);
```

is handled properly by doing nothing.

If the array sizes differ, a new array of objects is allocated. As discussed above, this operation takes $O(n)$ in the worst case, where $n$ is the new array length.

Next, there is a loop which copies one-by-one the elements of the input array to the newly allocated array. Clearly this operation takes $O(n)$ time to perform. Finally, the baseIndex field is copied in $O(1)$ time. Altogether, the running time of the Copy method is $T(n)=O(n)$, where $n$ is the size of the array being copied.
DynamicArray Indexers

The elements of a C# array are accessed by enclosing the index expression between brackets [ and ] like this:

```csharp
a[2] = b[3];
```

In order to be able to use the same syntax to access the elements of a DynamicArray object, we define an indexer.

Program defines an indexer that provides both get and set accessor methods. The get accessor takes an index and returns the element found in the array at the given position. The set accessor takes an index and an object value and stores the value in the array at the given position.

```csharp
public class DynamicArray
{
    protected object[] data;
    protected int baseIndex;

    public object this[int position]
    {
        get { return data[position - baseIndex]; }
        set { data[position - baseIndex] = value; }
    }
    // ...
Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
DynamicArray Properties

Program defines three properties of DynamicArrays—Data, BaseIndex, and Length. These properties provide the means to inspect the contents of a DynamicArray object (using the get accessor methods) and the means to modify the contents of a DynamicArray object (using the set accessor methods).

Clearly, the running times of each of the BaseIndex property get and set accessors is a constant. Similarly, the running time of the Length property set accessor is also a constant.

```csharp
public class DynamicArray
{
    protected object[] data;
    protected int baseIndex;

    public object[] Data
    {
        get { return data; }
    }

    public int BaseIndex
    {
        get { return baseIndex; }
        set { this.baseIndex = baseIndex; }
    }

    public int Length
    {
        get { return data.Length; }
        set
        {
            if (data.Length != value)
            {
                object[] newData = new object[value];
                int min = data.Length < value ?
```
DynamicArray Properties

```csharp
object[] newData = new object[value];
int min = data.Length < value ?
    data.Length : value;
for (int i = 0; i < min; ++i)
    newData[i] = data[i];
data = newData;
```

Program: DynamicArray properties.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Resizing an Array

The set accessor of the Length property provides the means to change the size of an array at run time. This method can be used both to increase and to decrease the size of an array.

The running time of this algorithm depends only on the new array length. Let \( n \) be the original size of the array and let \( m \) be the new size of the array. Consider the case where \( m \neq n \). The method first allocates and initializes a new array of size \( m \). Next, it copies at most \( \min(m, n) \) elements from the old array to the new array. Therefore, \( T(m, n) = O(m) + O(\min(m, n)) = O(m) \).
Multi-Dimensional Arrays

A multi-dimensional array of dimension $n$ (i.e., an $n$-dimensional array or simply $n$-D array) is a collection of items which is accessed via $n$ subscript expressions. For example, in C# the $(i,j)^{th}$ element of the two-dimensional array $x$ is accessed by writing $x[i,j]$.

The built-in multi-dimensional arrays suffer the same indignities that simple one-dimensional arrays do: Array indices in each dimension range from zero to $\text{Length} - 1$, where $\text{Length}$ is the array length in the given dimension and the number of dimensions and the size of each dimension is fixed once the array has been allocated.

In order to illustrate how these deficiencies of the C# built-in multi-dimensional arrays can be overcome, we will examine the implementation of a multi-dimensional array class, `MultiDimensionalArray`, that is based on the one-dimensional array class discussed in Section □.

- **Array Subscript Calculations**
- **An Implementation**
- **Constructor**
- **MultiDimensionalArray Indexer**
- **Matrices**
- **Dense Matrices**
- **Canonical Matrix Multiplication**
Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Array Subscript Calculations

The memory of a computer is essentially a one-dimensional array—the memory address is the array subscript. Therefore, a natural way to implement a multi-dimensional array is to store its elements in a one-dimensional array. In order to do this, we need a mapping from the $n$ subscript expressions used to access an element of the multi-dimensional array to the one subscript expression used to access the one-dimensional array. For example, suppose we wish to represent a $2 \times 3$ array of of `int`s, `a`, using a one-dimensional array like this:

```csharp
int[] b = new int[6];
```

Then we need to determine which element of `b`, say `b[k]`, will be accessed given a reference of the form `a[i, j]`. That is, we need the mapping $f$ such that $k = f(i, j)$.

The mapping function determines the way in which the elements of the array are stored in memory. The most common way to represent an array is in row-major order, also known as lexicographic order. For example, consider the $2 \times 3$ two-dimensional array. The row-major layout of this array is shown in Figure 1.

<table>
<thead>
<tr>
<th>position</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b[0]</code></td>
<td><code>a[0,0]</code></td>
</tr>
<tr>
<td><code>b[1]</code></td>
<td><code>a[0,1]</code></td>
</tr>
<tr>
<td><code>b[2]</code></td>
<td><code>a[0,2]</code></td>
</tr>
<tr>
<td><code>b[3]</code></td>
<td><code>a[1,0]</code></td>
</tr>
<tr>
<td><code>b[4]</code></td>
<td><code>a[1,1]</code></td>
</tr>
<tr>
<td><code>b[5]</code></td>
<td><code>a[1,2]</code></td>
</tr>
</tbody>
</table>

**Figure**: Row-major order layout of a 2D array.

In row-major layout, it is the right-most subscript expression (the column index) that increases the
Array Subscript Calculations

fastest. As a result, the elements of the rows of the matrix end up stored in contiguous memory locations.

In Figure □, the first element of the first row is at position $b[0]$. The first element of the second row is at position $b[3]$, since there are 3 elements in each row.

We can now generalize this to an arbitrary $n$-dimensional array. Suppose we have an $n$-D array $a$ with dimensions

$$\delta_1 \times \delta_2 \times \cdots \times \delta_n.$$ 

Then, the position of the element $a[i_1, i_2, \ldots, i_n]$ is given by

$$\sum_{j=1}^{n} f_j i_j$$  \hspace{1cm} (4.1)

where

$$f_j = \begin{cases} 1 & j = n, \\ \prod_{k=j+1}^{n} \delta_k & 1 \leq j < n. \end{cases}$$  \hspace{1cm} (4.2)

The running time required to calculate the position appears to be $O(n^2)$ since the position is the sum of $n$ terms and for each term we need to compute $f_j$, which requires $O(n)$ multiplications in the worst case.

However, the factors $f_j$ are determined solely from the dimensions of the array. Therefore, we need only compute the factors once. Assuming that the factors have been precomputed, the position calculation can be done in $O(n)$ time using the following algorithm:

```c
int offset = 0;
for (int j = 1; j <= n; ++j)
    offset += f_j * i_j;
```

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
An Implementation

In this section we illustrate the implementation of a multi-dimensional array using a one-dimensional array. We do this by defining a class called `MultiDimensionalArray` that is very similar to the `DynamicArray` class defined in Section □.

Program □ defines the fields of the `MultiDimensionalArray` class. Altogether three fields are used. The first, `dimensions`, is an array of length \( n \), where \( n \) is number of dimensions and \( \text{dimension}[i] \) is the size of the \( i^{th} \) dimension (\( \delta_i \)).

```
public class MultiDimensionalArray {
    private int[] dimensions;
    private int[] factors;
    private object[] data;

    // ...
}
```

Program: `MultiDimensionalArray` fields.

The second field, `factors`, is also an array of length \( n \). The \( j^{th} \) element of the `factors` array corresponds to the factor \( f_j \) given by Equation □.

The third field, `data`, is a one-dimensional array used to hold the elements of the multi-dimensional array in row-major order.
Constructor

The constructor for the MultiDimensionalArray class is defined in Program 1. It takes as its lone argument an array of ints which represents the dimensions of the array. For example, to create a $3 \times 5 \times 7$ three-dimensional array, we invoke the constructor like this:

```csharp
MultiDimensionalArray a =
    new MultiDimensionalArray(3, 5, 7);
```

```csharp
public class MultiDimensionalArray
{
    private int[] dimensions;
    private int[] factors;
    private object[] data;

    public MultiDimensionalArray(params int[] args)
    {
        dimensions = new int[args.Length];
        factors = new int[args.Length];
        int product = 1;
        for (int i = args.Length - 1; i >= 0; --i)
        {
            dimensions[i] = args[i];
            factors[i] = product;
            product *= dimensions[i];
        }
        data = new object[product];
    }
    // ...
}
```

**Program:** MultiDimensionalArray constructor.

The constructor copies the dimensions of the array into the dimensions array, and then it computes...
the factors array. These operations take $O(n)$, where $n$ is the number of dimensions. The constructor then allocates a one-dimensional array of length $m$ given by

$$m = \prod_{i=0}^{n-1} \delta_i.$$

The worst-case running time of the constructor is $O(m+n)$. 
MultiDimensionalArray Indexer

The elements of a multi-dimensional array are accessed using the get and set accessor methods of the MultiDimensionalArray indexer. For example, you can access the \((i, j, k)\)th element of a three-dimensional array \(a\) like this:

\[
value = a[i,j,k];
\]

and you can modify the \((i, j, k)\)th element like this:

\[
a[i,j,k] = value;
\]

Program \(\square\) defines an indexer that provides get and set accessors implemented using the GetOffset method. The GetOffset method takes a set of \(n\) indices and computes the position of the corresponding element in the one-dimensional array according to Equation \(\square\). This computation takes \(O(n)\) time in the worst case, where \(n\) is the number of dimensions. Consequently, the running times of the get and set accessors are also \(O(n)\).
public class MultiDimensionalArray
{
    private int[] dimensions;
    private int[] factors;
    private object[] data;

    private int GetOffset(int[] indices)
    {
        if (indices.Length != dimensions.Length)
            throw new IndexOutOfRangeException();
        int offset = 0;
        for (int i = 0; i < dimensions.Length; ++i)
        {
            if (indices[i] < 0 || indices[i] >= dimensions[i])
                throw new IndexOutOfRangeException();
            offset += factors[i] * indices[i];
        }
        return offset;
    }

    public object this[params int[] indices]
    {
        get { return data[GetOffset(indices)]; } 
        set { data[GetOffset(indices)] = value; } 
    }
    // ...
}

Program: MultiDimensionalArray indexer.
Matrices

Multi-dimensional arrays of floating-point numbers arise in many different scientific computations. Such arrays are usually called matrices. Mathematicians have studied the properties of matrices for many years and have developed an extensive repertoire of operations on matrices. In this section we consider two-dimensional matrices of doubles and examine the implementation of simple, matrix multiplication.

The preceding sections show that there are many possible ways to implement matrices. In order to separate interface from implementation, we define the abstract Matrix base class shown in Program 1.

```csharp
public abstract class Matrix
{
    public abstract double this[int i, int j] { get; set; }
    public abstract int Rows { get; }
    public abstract int Columns { get; }
    public abstract Matrix Transpose { get; }
    public abstract Matrix Plus(Matrix matrix);
    public abstract Matrix Times(Matrix matrix);
    public static Matrix operator +(Matrix m1, Matrix m2)
    {
        return m1.Plus(m2);
    }
    public static Matrix operator *(Matrix m1, Matrix m2)
    {
        return m1.Times(m2);
    }
}
```

Program: Matrix abstract class.

This interface defines an indexer (with get and put accessors) and methods for some of the elementary operations on matrices such as computing the transpose of a matrix (Transpose), adding matrices (Plus), and multiplying matrices (Times). In addition, the addition and multiplication operators + and * are overloaded for use with Matrix instances.
Dense Matrices

The simplest way to implement a matrix is to use a two-dimensional array as shown in Program 6. In this case, we use three fields. The first two fields, `numberOfRows` and `numberOfColumns` record the dimensions of the matrix. The third field, `array`, is a C# two-dimensional array of doubles.

```csharp
public class DenseMatrix : Matrix
{
    protected int numberOfRows;
    protected int numberOfColumns;
    protected double[,] array;

    public DenseMatrix(int numberOfRows, int numberOfColumns)
    {
        this.numberOfRows = numberOfRows;
        this.numberOfColumns = numberOfColumns;
        array = new double[numberOfRows, numberOfColumns];
    }

    public override int Rows
    { get { return numberOfRows; } }

    public override int Columns
    { get { return numberOfColumns; } }
    // ...
}
```

Program: `DenseMatrix` fields, constructor, and properties.

The constructor takes two arguments, $m$ and $n$, and constructs the corresponding $m \times n$ matrix. Clearly, the running time of the constructor is $O(mn)$. (Remember, C# initializes all the array elements to zero).
Canonical Matrix Multiplication

Given an \( m \times n \) matrix \( A \) and an \( n \times p \) matrix \( B \), the product \( C = AB \) is an \( m \times p \) matrix. The elements of the result matrix are given by

\[
c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}.
\]  \( (4.3) \)

Accordingly, in order to compute the product matrix, \( C \), we need to compute \( mp \) summations each of which is the sum of \( n \) product terms. An algorithm to compute the matrix product is given in Program \( \square \). The algorithm given is a direct implementation of Equation \( \square \).
Program: DenseMatrix class times method.

The algorithm begins by checking to see that the matrices to be multiplied have compatible dimensions. That is, the number of columns of the first matrix must be equal to the number of rows of the second one. This check takes $O(1)$ time in the worst case.

Next a matrix in which the result will be formed is constructed (line 12-13). The running time for this is $O(mp)$. For each value of $i$ and $j$, the innermost loop (lines 19-20) does $n$ iterations. Each iteration takes a constant amount of time.

The body of the middle loop (lines 16-22) takes time $O(n)$ for each value of $i$ and $j$. The middle loop is
done for $p$ iterations, giving the running time of $O(np)$ for each value of $i$. Since, the outer loop does $m$ iterations, its overall running time is $O(mnp)$. Finally, the result matrix is returned on line 24. This takes a constant amount of time.

In summary, we have shown that lines 9-11 are $O(1)$; lines 12-13 are $O(mp)$; lines 14-23 are $O(mnp)$; and line 24 is $O(1)$. Therefore, the running time of the canonical matrix multiplication algorithm is $O(mnp)$. 
Singly-Linked Lists

The singly-linked list is the most basic of all the linked data structures. A singly-linked list is simply a sequence of dynamically allocated objects, each of which refers to its successor in the list. Despite this obvious simplicity, there are myriad implementation variations. Figure □ shows several of the most common singly-linked list variants.

![Singly-linked list variations.

The basic singly-linked list is shown in Figure □ (a). Each element of the list refers to its successor and the last element contains the null reference. One variable, labeled head in Figure □ (a), is used to keep track of the list.

The basic singly-linked list is inefficient in those cases when we wish to add elements to both ends of the list. While it is easy to add elements at the head of the list, to add elements at the other end (the tail) we need to locate the last element. If the basic basic singly-linked list is used, the entire list needs to be
traversed in order to find its tail.

Figure (b) shows a way in which to make adding elements to the tail of a list more efficient. The solution uses a second variable, `tail`, which refers to the last element of the list. Of course, this time efficiency comes at the cost of the additional space used to store the variable `tail`.

The singly-linked list labeled (c) in Figure illustrates two common programming tricks. There is an extra element at the head of the list called a *sentinel*. This element is never used to hold data and it is always present. The principal advantage of using a sentinel is that it simplifies the programming of certain operations. For example, since there is always a sentinel standing guard, we never need to modify the `head` variable. Of course, the disadvantage of a sentinel such as that shown in (c) is that extra space is required, and the sentinel needs to be created when the list is initialized.

The list (c) is also a *circular list*. Instead of using a `null` reference to demarcate the end of the list, the last element of the list refers to the sentinel. The advantage of this programming trick is that insertion at the head of the list, insertion at the tail of the list, and insertion at an arbitrary position of the list are all identical operations.

Of course, it is also possible to make a circular, singly-linked list that does not use a sentinel. Figure (d) shows a variation in which a single variable is used to keep track of the list, but this time the variable, `tail`, refers to the last element of the list. Since the list is circular in this case, the first element follows the last element of the list. Therefore, it is relatively simple to insert both at the head and at the tail of this list. This variation minimizes the storage required, at the expense of a little extra time for certain operations.

Figure illustrates how the empty list (i.e., the list containing no list elements) is represented for each of the variations given in Figure. Notice that the sentinel is always present in list variant (c). On the other hand, in the list variants which do not use a sentinel, the `null` reference is used to indicate the empty list.
In the following sections, we will present the implementation details of a generic singly-linked list. We have chosen to present variation (b)--the one which uses a head and a tail--since it supports append and prepend operations efficiently.

**Figure:** Empty singly-linked lists.
An Implementation

Figure illustrates the singly-linked list scheme we have chosen to implement. Two related structures are used. The elements of the list are represented using instances of the Element class which comprises three fields, list, datum and next. The main structure is an instance of the LinkedList class which also comprises two fields, head and tail, which refer to the first and last list elements, respectively. The list field of every Element contains a reference to the LinkedList instance with which it is associated. The datum field is used to refer to the objects in the list and the next field refers to the next list element.

Figure: Memory representation of a linked list.

Program defines the LinkedList.Element class. It is used to represent the elements of a linked list. It has three fields, list, datum and next, a constructor and two properties, Datum and Next.
An Implementation

Program also defines the fields of the LinkedList class, head and tail.

```java
public class LinkedList
{
    protected Element head;
    protected Element tail;

    public sealed class Element
    {
        internal LinkedList list;
        internal object datum;
        internal Element next;

        internal Element(LinkedList list, object datum, Element next)
        {
            this.list = list;
            this.datum = datum;
            this.next = next;
        }

        public object Datum
        { get { return datum; } }

        public Element Next
        { get { return next; } }

        // ...
    }
}
```

**Program:** LinkedList fields and LinkedList.Element class.

We can calculate the total storage required, \( S(n) \), to hold a linked list of \( n \) items from the class definitions given in Program as follows:
An Implementation

\[ S(n) = \text{sizeof}(\text{LinkedList}) + n \text{sizeof}(\text{LinkedList.Element}) \]
\[ = 2 \text{sizeof}(\text{LinkedList.Element\ ref}) \]
\[ + n(\text{sizeof}(\text{LinkedList\ ref}) + \text{sizeof}(\text{object\ ref}) \]
\[ + \text{sizeof}(\text{LinkedList.Element\ ref})) \]
\[ = (n + 2) \text{sizeof}(\text{LinkedList.Element\ ref}) \]
\[ + n \text{sizeof}(\text{LinkedList\ ref}) \]
\[ + n \text{sizeof}(\text{object\ ref}) \]

In C# all object references occupy a constant amount of space. Therefore, \( S(n) = O(n) \).
List Elements

The definitions of the methods of the `LinkedList.Element` class are given in Program 1. Altogether, there are three methods--a constructor and two properties.

The constructor simply initializes the field to the provided values. Assigning a value to the `list`, `datum` and `next` fields takes a constant amount of time. Therefore, the running time of the constructor is \(O(1)\).

The `Datum` and `Next` properties provide `get` accessor methods that simply return the values of the corresponding fields. Clearly, the running times of each of these methods is \(O(1)\).

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
LinkedList Default Constructor

The code for the LinkedList default constructor is given in Program 1. Since the fields head and tail are initially null, the list is empty by default. As a result, the constructor does nothing. The running time of the default constructor is clearly constant. That is, $T(n)=O(1)$.

```
public class LinkedList
{
    protected Element head;
    protected Element tail;

    public LinkedList()
    {
        // ...
    }
}
```

Program: LinkedList default constructor.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
**Purge Method**

Program gives the code for the Purge method of the LinkedList class. The purpose of this method is to discard the current list contents and to make the list empty again. Clearly, the running time of Purge is $O(1)$.

```csharp
public class LinkedList
{
    protected Element head;
    protected Element tail;

    public void Purge()
    {
        head = null;
        tail = null;
        // ...
    }
}
```

**Program:** LinkedList class Purge method.

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
LinkedList Properties

Three LinkedList properties are defined in Program. The Head and Tail properties provide get accessors for the corresponding fields of LinkedList. The isEmpty property provides a get accessor that returns a bool result which indicates whether the list is empty. Clearly, the running time of each accessor is $O(1)$.

```csharp
public class LinkedList
{
    protected Element head;
    protected Element tail;

    public Element Head
    { get { return head; } }

    public Element Tail
    { get { return tail; } }

    public bool isEmpty
    { get { return head == null; } }
    // ...
}
```

**Program:** LinkedList class properties.

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
First and Last Properties

Two more LinkedList properties are defined in Program 1. The First property provides a get accessor that returns the first list element. Similarly, the Last property provides a get accessor that returns the last list element. The code for both methods is almost identical. In the event that the list is empty, a ContainerEmptyException exception is thrown.

```csharp
public class LinkedList
{
    protected Element head;
    protected Element tail;

    public object First
    {
        get
        {
            if (head == null)
                throw new ContainerEmptyException();
            return head.datum;
        }
    }

    public object Last
    {
        get
        {
            if (tail == null)
                throw new ContainerEmptyException();
            return tail.datum;
        }
        // ...
    }
}
```
Program: LinkedList class First and Last properties.

We will assume that in a bug-free program, neither the First nor the Last property accessors will be called for an empty list. In that case, the running time of each of these methods is constant. That is, \( T(n) = O(1) \).
Prepend Method

To prepend an element to a linked list is to insert that element in front of the first element of the list. The prepended list element becomes the new head of the list. Program □ gives the algorithm for the Prepend method of the LinkedList class.

```csharp
public class LinkedList
{
    protected Element head;
    protected Element tail;

    public void Prepend(object item)
    {
        Element tmp = new Element(this, item, head);
        if (head == null)
            tail = tmp;
        head = tmp;
    }
    // ...
}
```

Program: LinkedList class Prepend method.

The Prepend method first creates a new LinkedList.Element. Its datum field is initialized with the value to be prepended to the list, item; and the next field refers to the first element of the existing list by initializing it with the current value of head. If the list is initially empty, both head and tail refer to the new element. Otherwise, just head needs to be updated.

Note, the new operator initializes the new LinkedList.Element instance by calling its constructor. In Section □ the running time of the constructor was determined to be O(1). And since the body of the Prepend method adds only a constant amount of work, the running time of the Prepend method is also O(1).
Append Method

The Append method, the definition of which is given in Program 1, adds a new LinkedList.Element at the tail-end of the list. The appended element becomes the new tail of the list.

```
public class LinkedList
{
    protected Element head;
    protected Element tail;

    public void Append(object item)
    {
        Element tmp = new Element(this, item, null);
        if (head == null)
            head = tmp;
        else
        {
            tail.next = tmp;
            tail = tmp;
        }
    }
}
```

Program: LinkedList class Append method.

The Append method first allocates a new LinkedList.Element. Its datum field is initialized with the value to be appended, and the next field is set to null. If the list is initially empty, both head and tail refer to the new element. Otherwise, the new element is appended to the existing list, and the just tail pointer is updated.

The running time analysis of the Append method is essentially the same as for Prepend. I.e, the running time is \( O(1) \).
Copy Method

The code for the Copy method of the LinkedList class is given in Program 1. The Copy method is used to assign the elements of one list to another. It does this by discarding the current list elements and then building a copy of the given linked list.

```csharp
public class LinkedList
{
    protected Element head;
    protected Element tail;

    public void Copy(LinkedList list)
    {
        if (list != this)
        {
            Purge();
            for (Element ptr = list.head;
                 ptr != null; ptr = ptr.next)
            {
                Append(ptr.datum);
            }
        }
        // ...
    }
}
```

Program: LinkedList class Copy method.

The Copy method begins by calling Purge to make sure that the list to which new contents are being assigned is empty. Then, it traverses the list passed to it one-by-one calling the Append method to append the items to the list begin constructed.

In Section 4 the running time for the Append method was determined to be $O(1)$. If the resulting list has $n$ elements, the Append method will be called $n$ times. Therefore, the running time of the Copy
method is $O(n)$.
**Extract Method**

In this section we consider the Extract method of the LinkedList class. The purpose of this method is to delete the specified element from the linked list.

```csharp
public class LinkedList
{
    protected Element head;
    protected Element tail;

    public void Extract(object item)
    {
        Element ptr = head;
        Element prevPtr = null;
        while (ptr != null && ptr.datum != item)
        {
            prevPtr = ptr;
            ptr = ptr.next;
        }
        if (ptr == null)
            throw new ArgumentException("item not found");
        if (ptr == head)
            head = ptr.next;
        else
            prevPtr.next = ptr.next;
        if (ptr == tail)
            tail = prevPtr;
    }
    // ...
}
```

**Program:** LinkedList class Extract method.

The Extract method searches sequentially for the item to be deleted. In the absence of any *a priori*
knowledge, we do not know in which list element the item to be deleted will be found. In fact, the specified item may not even appear in the list!

If we assume that the item to be deleted is in the list, and if we assume that there is an equal probability of finding it in each of the possible positions, then on average we will need to search half way through the list before the item to be deleted is found. In the worst case, the item will be found at the tail—assuming it is in the list.

If the item to be deleted does not appear in the list, the algorithm shown in Program throws an ArgumentException exception. A simpler alternative might be to do nothing—after all, if the item to be deleted is not in the list, then we are already done! However, attempting to delete an item which is not there, is more likely to indicate a logic error in the programming. It is for this reason that an exception is thrown.

In order to determine the running time of the Extract method, we first need to determine the time to find the element to be deleted. If the item to be deleted is not in the list, then the running time of Program is up to the point where it throws the exception (line 16) is $T(n) = O(n)$.

Now consider what happens if the item to be deleted is found in the list. In the worst-case the item to be deleted is at the tail. Thus, the running time to find the element is $O(n)$. Actually deleting the element from the list once it has been found is a short sequence of relatively straight-forward manipulations. These manipulations can be done in constant time. Therefore, the total running time is $T(n) = O(n)$. 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
InsertAfter and InsertBefore Methods

Consider the methods `InsertAfter` and `InsertBefore` of the `LinkedList.Element` class shown in Program 1. Both methods take a single argument that specifies an item to be inserted into the list. The given item is inserted either in front of or immediately following this list element.

```csharp
public class LinkedList
{
    protected Element head;
    protected Element tail;

    public sealed class Element
    {
        internal LinkedList list;
        internal object datum;
        internal Element next;

        public void InsertAfter(object item)
        {
            next = new Element(list, item, next);
            if (list.tail == this)
                list.tail = next;
        }

        public void InsertBefore(object item)
        {
            Element tmp = new Element(list, item, this);
            if (this == list.head)
                list.head = tmp;
            else
            {
                Element prevPtr = list.head;
                while (prevPtr != null && prevPtr.next != this)
                    prevPtr = prevPtr.next;
                prevPtr.next = tmp;
            }
        }
    }

    public void AddLast(object item)
    {
        Element tmp = new Element(this, item, null);
        if (tail == null)
            head = tmp;
        else
        {
            tail.next = tmp;
            tail = tmp;
        }
    }
}
```
Program: LinkedList.Element class InsertAfter and InsertBefore methods.

The InsertAfter method is almost identical to Append. Whereas Append inserts an item after the tail, InsertAfter inserts an item after an arbitrary list element. Nevertheless, the running time of InsertAfter is identical to that of Append, i.e., it is $O(1)$.

To insert a new item before a given list element, it is necessary to traverse the linked list starting from the head to locate the list element that precedes the given list element. In the worst case, the given element is at the tail of the list and the entire list needs to be traversed. Therefore, the running time of the InsertBefore method is $O(n)$. 

```c
    prevPtr = prevPtr.next;
    prevPtr.next = tmp;
}
/* ...
}
// ...
}
```
Exercises

1. How much space does the `DynamicArray` class declared in Program use to store an array of `Int32` of length `N`?
2. How much space does the `LinkedList` class declared in Program use to store a list of `n` `Int32`s?
3. For what value of `N/n` do the two classes use the same amount of space?

2. Consider the `Copy` method of the `DynamicArray` class given in Program. What is the purpose of the test `array != this` on line 8?

3. The `Copy` method of the `DynamicArray` class defined in Program has the effect of making the target of the assignment exactly the same as the source. An alternative version could assign the elements based on their apparent locations in the source an target arrays. That is, assign `a[i]` to `b[i]` for all values of `i` that are valid subscripts in both `a` and `b`. Write an `Copy` method with the modified semantics.

4. The array subscripting methods defined in Program don't test explicitly the index expression to see if it is in the proper range. Explain why the test is not required in this implementation.

5. The `Base` property `set` accessor of the `DynamicArray` class defined in Program simply changes the value of the `baseIndex` field. As a result, after the base is changed, all the array elements appear to have moved. How might the method be modified so that the elements of the array don't change their apparent locations when the base is changed?

6. Equation is only correct if the subscript ranges in each dimension start at zero. How does the formula change when each dimension is allowed to have an arbitrary subscript range?

7. The alternative to `row-major` layout of of multi-dimensional arrays is called `column-major order`. In column-major layout the leftmost subscript expression increases fastest. For example, the elements of the columns of a two-dimensional matrix end up stored in contiguous memory locations. Modify Equation to compute the correct position for column-major layout.

8. Consider the `Times` and `Plus` methods of the `Matrix` interface defined in Program. Implement these methods for the `DenseMatrix` class defined in Program.

9. Which methods are affected if we drop the `tail` member variable from the `LinkedList` class declared in Program? Determine new running times for the affected methods.

10. How does the implementation of the `Prepend` method of the `LinkedList` class defined in Program change when a circular list with a sentinel is used as shown in Figure (c).

11. How does the implementation of the `Append` method of the `LinkedList` class defined in
Exercises

Program\[\textbf{change}\] when a circular list with a sentinel is used as shown in Figure\[\textbf{c}\].

12. Consider the assignment operator for the \texttt{LinkedList} class given in Program\[\textbf{1}\]. What is the purpose of the test \texttt{linkedlist} \(!=\ this\) on line 8?
Projects

1. Complete the implementation of the `DynamicArray` class given in Program to Program. Write a test suite to verify all of the functionality. Try to exercise every line of code in the implementation.

2. Complete the implementation of the `LinkedList` class given in Program to Program. Write a test suite to verify all of the functionality. Try to exercise every line of code in the implementation.

3. Change the implementation of the `LinkedList` class given in Program to Program by removing the `tail` field. That is, implement the singly-linked list variant shown in Figure (a). Write a test suite to verify all of the functionality. Try to exercise every line of code in the implementation.

4. Change the implementation of the `LinkedList` class given in Program to Program so that it uses a circular, singly-linked list with a sentinel as shown in Figure (c). Write a test suite to verify all of the functionality. Try to exercise every line of code in the implementation.

5. The `MultiDimensionalArray` class given in Program to Program only supports subscript ranges starting at zero. Modify the implementation to allow an arbitrary subscript base in each dimension.

6. Design and implement a three-dimensional matrix class `Matrix3D` based on the two-dimensional class `DenseMatrix` given in Program to Program.

7. A row vector is a $1 \times n$ matrix and a column vector is an $n \times 1$ matrix. Define and implement classes `RowVector` and `ColumnVector` as classes derived from the base class `DynamicArray` given in Program to Program. Show how these classes can be combined to implement the `Matrix` interface declared in Program to Program.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
It is said that "computer science is [the] science of abstraction." But what exactly is abstraction? Abstraction is "the idea of a quality thought of apart from any particular object or real thing having that quality." For example, we can think about the size of an object without knowing what that object is. Similarly, we can think about the way a car is driven without knowing its model or make.

Abstraction is used to suppress irrelevant details while at the same time emphasizing relevant ones. The benefit of abstraction is that it makes it easier for the programmer to think about the problem to be solved.
Abstract Data Types

A variable in a procedural programming language such as Fortran, Pascal, C++, and C#, is an abstraction. The abstraction comprises a number of attributes -- name, address, value, lifetime, scope, type, and size. Each attribute has an associated value. For example, if we declare an integer variable in C#, `int x`, we say that the name attribute has value ``x'' and that the type attribute has value ``int''.

Unfortunately, the terminology can be somewhat confusing: The word ``value'' has two different meanings--in one instance it denotes one of the attributes and in the other it denotes the quantity assigned to an attribute. For example, after the assignment statement `x = 5`, the value attribute has the value five.

The name of a variable is the textual label used to refer to that variable in the text of the source program. The address of a variable denotes its location in memory. The value attribute is the quantity which that variable represents. The lifetime of a variable is the interval of time during the execution of the program in which the variable is said to exist. The scope of a variable is the set of statements in the text of the source program in which the variable is said to be visible. The type of a variable denotes the set of values which can be assigned to the value attribute and the set of operations which can be performed on the variable. Finally, the size attribute denotes the amount of storage required to represent the variable.

The process of assigning a value to an attribute is called binding. When a value is assigned to an attribute, that attribute is said to be bound to the value. Depending on the semantics of the programming language, and on the attribute in question, the binding may be done statically by the compiler or dynamically at run-time. For example, in C# the type of a variable is determined at compile time--static binding. On the other hand, the value of a variable is usually not determined until run-time--dynamic binding.

In this chapter we are concerned primarily with the type attribute of a variable. The type of a variable specifies two sets:

- a set of values; and,
- a set of operations.

For example, when we declare a variable, say `x`, of type `int`, we know that `x` can represent an integer in the range $[-2^{31}, 2^{31} - 1]$ and that we can perform operations on `x` such as addition, subtraction,
multiplication, and division.

The type int is an abstract data type in the sense that we can think about the qualities of an int apart from any real thing having that quality. In other words, we don't need to know how ints are represented nor how the operations are implemented to be able to use them or reason about them.

In designing object-oriented programs, one of the primary concerns of the programmer is to develop an appropriate collection of abstractions for the application at hand, and then to define suitable abstract data types to represent those abstractions. In so doing, the programmer must be conscious of the fact that defining an abstract data type requires the specification of both a set of values and a set of operations on those values.

Indeed, it has been only since the advent of the so-called object-oriented programming languages that we see programming languages which provide the necessary constructs to properly declare abstract data types. For example, in C#, the class construct is the means by which both a set of values and an associated set of operations is declared. Compare this with the struct construct of C or Pascal's record, which only allow the specification of a set of values!
Design Patterns

An experienced programmer is in a sense like concert musician--he has mastered a certain repertoire of pieces which he is prepared to play at any time. For the programmer, the repertoire comprises a set of abstract data types with which he is familiar and which he is able to use in her programs as the need arises.

The chapters following this present a basic repertoire of abstract data types. In addition to defining the abstractions, we show how to implement them in C# and we analyze the performance of the algorithms.

The repertoire of basic abstract data types has been designed as a hierarchy of C# classes. This section presents an overview of the class hierarchy and lays the groundwork for the following chapters.

- **Class Hierarchy**
- **C# Objects and the IComparable Interface**
- **Wrappers for Value Types**
- **Containers**
- **Visitors**
- **Enumerable Collections andEnumerators**
- **Searchable Containers**
- **Associations**
Class Hierarchy

The C# class hierarchy which is used to represent the basic repertoire of abstract data types is shown in Figure 1. Two kinds of classes are shown in Figure 1; abstract C# classes, which look like this \texttt{AbstractClass}, and concrete C# classes, which look like this \texttt{ConcreteClass}. In addition, C# interfaces are shown like this \texttt{Interface}. Solid lines in the figure indicate the \textit{specializes} relation between classes and between interfaces; base classes and interfaces always appear to the left of derived classes and interfaces. Dashed lines indicates the the \textit{realizes} relation between a class and the interface(s) it implements. In C# a class may specialize at most one other class and it may realize any number of interfaces. A C# interface may specialize any number of interfaces (but not classes).
A C# interface comprises a set of method and property declarations. An interface does not supply implementations for the methods or properties it declares. In effect, an interface identifies the set of operations provided by every class that implements the interface.

An abstract class in C# is a class which defines only part of an implementation. Consequently, it is not possible to create object instances of abstract classes. In C# an abstract class may contain zero or more abstract methods or abstract properties. A abstract method or property is one for which no implementation is given.

An abstract class is intended to be used as the base class from which other classes are derived. By declaring abstract methods in the base class, it is possible to access the implementations provided by the derived classes through the base-class methods. Consequently, we don't need to know how a particular object instance is implemented, nor do we need to know of which derived class it is an instance.

This design pattern uses the idea of polymorphism. Polymorphism literally means "having many forms." The essential idea is that a C# interface is used to define the set of values and the set of operations—the abstract data type. Then, various different implementations (many forms) of the interface can be made. We do this by defining abstract classes that contain shared implementation features and then by deriving concrete classes from the abstract base classes.
The remainder of this section presents the top levels of the class hierarchy which are shown in Figure 2. The top levels define those attributes of objects which are common to all of the classes in the hierarchy. The lower levels of the hierarchy are presented in subsequent chapters where the abstractions are defined and various implementations of those abstractions are elaborated.

**Figure:** Object class hierarchy.
C# Objects and the IComparable Interface

All C# classes, including arrays, are ultimately derived from the base class called object. The object keyword is an alias for the class System.Object. The following code fragment identifies some of the methods defined in the System.Object class:

```csharp
namespace System
{
    public class Object
    {
        public Object() { ... }
        public virtual bool Equals(object o) { ... }
        public virtual int GetHashCode() { ... }
        public Type GetType() { ... }
        public virtual string ToString() { ... }
    // ...
    }
}
```

Notice that the C# System.Object class contains a method called Equals, the purpose of which is to indicate whether some other object is "equal to" this one. By default, obj1.Equals(obj2) returns true only if obj1 and obj2 refer to the same object.

Of course, any derived class can override the Equals method to do the comparison in a way that is appropriate to that class. For example, the Equals method is overridden in the System.Int32 class as follows: If obj1 and obj2 are Int32s, then obj1.equals(obj2) is true when (int)obj1 is equal to (int)obj2.

So, all C# objects provide a means to test for equality. Unfortunately, they do not provide a means to test whether one object is "less than" or "greater than" another. To overcome this difficulty, C# provides the standard interface called IComparable. The following code fragment defines the IComparable interface.

```csharp
namespace System.Collections
```
C# Objects and the IComparable Interface

```csharp
public interface IComparable
{
    int CompareTo(object o);
}
```

The `IComparable` interface defines a single method. This instance method takes a specified object and compares it with the given object instance. The method returns an integer that is less than, equal to, or greater than zero depending on whether this object instance is less than, equal to, or greater than the specified object instance `o`, respectively.

---

- Abstract Comparable Objects
- Comparison Operators

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Abstract Comparable Objects

The abstract class at the top of the class hierarchy is called ComparableObject. All the other classes in the hierarchy are ultimately derived from this class. As shown in Figure 3, the ComparableObject class implements the IComparable interface discussed in Section 4.

The CompareTo method is defined as an abstract method in Program 6. Program 7 also defines the private method Compare. To understand the operation of the Compare method, consider an expression of the form obj1.CompareTo(obj2). First, the Compare method determines whether obj1 and obj2 are instances of the same type (line 7). If they are, the CompareTo method is called to do the comparison. Thus, the CompareTo method is only ever invoked for instances of the same class.

```
1 public abstract class ComparableObject : IComparable
2 {
3     public abstract int CompareTo(Object obj);
4
5     private int Compare(object obj)
6     {
7         if (GetType() == obj.GetType())
8             return CompareTo(obj);
9         else
10             return GetType().FullName.CompareTo(
11                 obj.GetType().FullName);
12     }
13
14     public override bool Equals(object obj)
15     { return Compare(obj) == 0; }
16
17     // ...  
18 }
```

Program: ComparableObject methods.

If obj1 and obj2 are instances of different types, then the comparison is based on the names of the types (lines 10-11). Suppose obj1 is an instance of the class named Opus6.StackAsArray and
obj2 is an instance of the class named `Opus6.QueueAsLinkedList`. Then obj1 is "less than" obj2 because `StackAsArray` precedes alphabetically `QueueAsLinkedList`.
Comparison Operators

Program □ shows how the comparison operators ==, !=, <, <=, >, and > are implemented. All of these methods invoke the `Compare` method and then interpret the result as needed.

The == and != operators are slightly different to make them more robust and easier to use. Specifically, they do the right thing when either operand of == and != is a null reference.

```csharp
public abstract class ComparableObject : IComparable
{
    public static bool operator ==(ComparableObject c, object o)
    {
        if ((object)c == null || (object)o == null)
            return (object)c == (object)o;
        else
            return c.CompareTo(o) == 0;
    }

    public static bool operator !=(ComparableObject c, object o)
    {
        if ((object)c == null || (object)o == null)
            return (object)c != (object)o;
        else
            return c.CompareTo(o) != 0;
    }

    public static bool operator <(ComparableObject c, object o)
    { return c.CompareTo(o) < 0; }

    public static bool operator >(ComparableObject c, object o)
    { return c.CompareTo(o) > 0; }

    public static bool operator<=(ComparableObject c, object o)
    { return c.CompareTo(o) <= 0; }

    public static bool operator >=(ComparableObject c, object o)
    { return c.CompareTo(o) >= 0; }
}
```
Comparison Operators

Program: ComparableObject comparison operators.

The use of polymorphism in the way shown gives the programmer enormous leverage. The fact all objects are derived from the ComparableObject base class, together with the fact that every concrete class must implement an appropriate CompareTo method, ensures that the comparison operators can be used to compare any pair of ComparableObjects and that the comparisons always work as expected.
Wrappers for Value Types

The value types in C# are bool, char, schar, short, ushort, int, uint, long, ulong, float, and double. There is also the “type” void which is used in the place of the return type to declare a method that returns nothing. Whenever a value type is used in a context where an object is required, C# automatically boxes the value type. Therefore, in C# a value parameter can be used wherever an object is expected.

Each C# value type is an alias for a struct in the System class. E.g., int is an alias for the struct System.Int32 and void is an alias for the struct System.Void. Since all C# structs are ultimately derived from the object class, they all implement the methods defined in that class. Specifically, a value types provide an Equals methods.

Furthermore, most (but not all) of the C# value types implement the IComparable interface. I.e., they provide a CompareTo method for comparing instances of the same type. Unfortunately, since the value types are not derived from the ComparableObject class it is not possible to use the operators shown in Program □ with value type instances.

To circumvent this shortcoming, we might be tempted to try something like this:

class ComparableInt32 :
    ComparableObject,
    System.Int32 // Wrong. Int32 is a struct!
{
    // ...
}

Unfortunately, according to the C# language specification, Int32 is a struct--it cannot be extended[22]. Consequently, we are forced to implement our own wrapper classes if we want them to extend ComparableObject base class.

Program □ defines the ComparableValue abstract class that extends the ComparableObject base class. This class “wraps” an object that implements the IComparable interface.
Wrappers for Value Types

```csharp
public abstract class ComparableValue : ComparableObject
{
    protected IComparable obj;

    public ComparableValue(IComparable obj)
    { this.obj = obj; }

    public virtual object Object
    { get { return obj; } }

    public override int CompareTo(object arg)
    {
        ComparableValue cv = arg as ComparableValue;
        if (obj.GetType() == cv.obj.GetType())
            return obj.CompareTo(cv.obj);
        else
            return obj.GetType().FullName.CompareTo(
                cv.obj.GetType().FullName);
    }

    public override int GetHashCode()
    { return obj.GetHashCode(); }

    public override string ToString()
    { return obj.ToString(); }
}
```

Program: More here

The `ComparableValue` class as a single field `obj` that refers to the wrapped `IComparable` object instance. The constructor takes a `IComparable` object reference and assigns it to the `obj` field. The `Object` property of the `ComparableValue` class provides a `get` accessor that returns the contained object instance. The `GetHashCode` and `ToString` methods simply delegate to the contained `IComparable` instance.

The `CompareTo` method compares a `ComparableValue` with a given `object`. The assumption is that the given object is also a `ComparableValue`. The `CompareTo` method compares the objects contained in the `ComparableValue` wrappers.
Wrappers for Value Types

Programs \( \square, \square \) and \( \square \) define three wrapper classes \( \text{ComparableChar} \), \( \text{ComparableInt32} \), and \( \text{ComparableDouble} \), which are wrappers for C# value types char, int, and double.

```csharp
public class ComparableChar : ComparableValue
{
    public ComparableChar(char c) : base(c)
    {
    }

    public static explicit operator char(ComparableChar c)
    {
        return (char)c.obj;
    }
    // ...
}
```

Program: ComparableChar class.

```csharp
public class ComparableInt32 : ComparableValue
{
    public ComparableInt32(int i) : base(i)
    {
    }

    public static explicit operator int(ComparableInt32 c)
    {
        return (int)c.obj;
    }
    // ...
}
```

Program: ComparableInt32 class.

```csharp
public class ComparableDouble : ComparableValue
{
    public ComparableDouble(double d) : base(d)
    {
    }

    public static explicit operator double(ComparableDouble c)
    {
        return (double)c.obj;
    }
    // ...
}
```

Program: ComparableDouble class.
C# also provides the `string` class for dealing with character sequences. The `string` class is special in that it is closely tied to the definition of the language itself. The C# compiler automatically creates a `string` object for every `string literal`, such as "Hello world.\n", in a C# program. Program defines the class `ComparableString` which wraps a `string` instance using the `ComparableValue` class.

```csharp
public class ComparableString : ComparableValue
{
    public ComparableString(string c) : base(c)
    {
    }

    public static explicit operator string(ComparableString c)
    {
        return (string)c.obj;
    }

    // ...
}
```

**Program:** ComparableString class.

Using these classes it is now possible to write a sequence of statements like:

```csharp
ComparableInt32 i = 1;
ComparableInt32 j = 2;
if (i > j)
    Console.WriteLine((int)i - (int)j);
```

In this sequence, the values 1 and 2 are first boxed by C# and then wrapped in instances of the `ComparableInt32` class. The comparison operator invoked is that given in Program and the `CompareTo` method invoked is that given in Program.

Finally, to make it possible to deal with `ComparableObjects` only, a collection of conversion operators is defined in Program I.e., for each value type a implicit conversion is provided that wraps that value type in the corresponding `ComparableValue` class. Similarly, explicit conversion operators are provided to unwrap the contained values types.
Wrappers for Value Types

```csharp
public abstract class ComparableObject : IComparable
{
    public static implicit operator ComparableObject(char c)
    {
        return new ComparableChar(c);
    }

    public static explicit operator char(ComparableObject c)
    {
        return (char)((ComparableChar)c);
    }

    public static implicit operator ComparableObject(int i)
    {
        return new ComparableInt32(i);
    }

    public static explicit operator int(ComparableObject c)
    {
        return (int)((ComparableInt32)c);
    }

    public static implicit operator ComparableObject(double d)
    {
        return new ComparableDouble(d);
    }

    public static explicit operator double(ComparableObject c)
    {
        return (double)((ComparableDouble)c);
    }

    public static implicit operator ComparableObject(string s)
    {
        return new ComparableString(s);
    }

    public static explicit operator string(ComparableObject c)
    {
        return (string)((ComparableString)c);
    }

    // ...
}
```

Program: More here

By using the methods given in Program □, we can rewrite the code fragment given above as

```csharp
ComparableObject i = 1;
ComparableObject j = 2;
if (i > j)
    Console.WriteLine((int)i - (int)j);
```

The effect of this code fragment is exactly as before. However, this time the objects are refered to by variables whose type is the abstract base class `ComparableObject`. 
A container is an object that contains within it other objects. Many of the data structures presented in this book can be viewed as containers. For this reason, we develop a common interface that is implemented by the various data structure classes.

The `Container` interface is declared in Program. It comprises the three properties, `Count`, `IsEmpty`, and `IsFull`, and two methods, `Purge` and `Accept`. In addition, the `Container` interface extends the `IComparable` interface (and therefore provides a `CompareTo` method) and it extends the `IEnumerable` interface (and therefore provides a `GetEnumerator` method).

```csharp
public interface Container : IComparable, IEnumerable
{
    int Count { get; }
    bool IsEmpty { get; }
    bool IsFull { get; }
    void Purge();
    void Accept(Visitor visitor);
}
```

**Program:** Container interface.

A container may be empty or it may contain one or more other objects. Typically, a container has finite capacity. The `IsEmpty` property provides a `get` accessor that returns `true` when the container is empty and the `IsFull` property provides a `get` accessor that returns `true` when the container is full. The `Count` property provides a `get` accessor that returns the number of objects in the container.

The purpose of the `Purge` method is to discard all of the contents of a container. After a container is purged, the `IsEmpty` property is `true` and the `Count` property is zero.

Conspicuous by their absence from Program are methods for putting objects into a container and for taking them out again. These methods have been omitted from the `Container` interface, because the precise nature of these methods depends on the type of container implemented.
In order to describe the remaining `Accept` methods we need to introduce first the concepts of a `visitor` and an `enumerator`, as well as with the `Visitor`, `IEnumerable` and `IEnumerator` interfaces which represent these concepts. Visitors are discussed below in Section □ and enumerators are discussed in Section □.
Abstract Containers

Program introduces an abstract class called AbstractContainer. It is intended to be used as the base class from which concrete container realizations are derived. As illustrated in Figure, the AbstractContainer class extends the ComparableObject class (defined in Program) and it implements the Container interface (defined in Program).

A single field, count, is used. This field is used to keep track of the number of objects held in the container. The count field is initially zero by default. It is the responsibility of the derived class to update this field as required.

```csharp
public abstract class AbstractContainer : ComparableObject, Container
{
    protected int count;

    public virtual int Count
    { get { return count; } }

    public virtual bool IsEmpty
    { get { return Count == 0; } }

    public virtual bool IsFull
    { get { return false; } }

    // ...
}
```

Program: AbstractContainer fields and properties.

The Count property provides a get accessor that returns the number of items contained in the container. The get accessor simply returns the value of the count field.

IsEmpty and IsFull bool-valued properties which indicate whether a given container is empty or full, respectively. Notice that the IsEmpty get accessor does not directly access the count field.
Instead it uses `Count` property. As long as the `Count` property has the correct semantics, the `IsEmpty` property will too.

In some cases, a container is implemented in a way which makes its capacity finite. When this is the case, it is necessary to be able to determine when the container is full. `IsFull` is a `bool`-valued property that provides a `get` accessor that returns the value `true` if the container is full. The default version always returns `false`. 
Visitors

The `Container` interface described in the preceding section interacts closely with the `Visitor` interface shown in Program 1. In particular, the `Accept` method of the `Container` interface takes as its argument a reference to any class that implements the `Visitor` interface.

```csharp
public interface Visitor
{
    void Visit(object obj);
    bool IsDone { get; }
}
```

**Program:** Visitor interface.

But what is a visitor? A shown in Program 1, a visitor is an object that has a `Visit` method and an `IsDone` property. Of these, the `Visit` method is the most interesting. The `Visit` method takes as its argument a reference to an object instance.

The interaction between a container and a visitor goes like this: The container is passed a reference to a visitor by calling the container's `Accept` method. That is, the container "accepts" the visitor. What does a container do with a visitor? It calls the `Visit` method of that visitor one-by-one for each object contained in the container.

The interaction between a `Container` and its `Visitor` are best understood by considering an example. The following code fragment gives the design framework for the implementation of the `Accept` method in some concrete class, say `SomeContainer`, that implements the `Container` interface:

```csharp
public class SomeContainer : Container
{
    public void Accept(Visitor visitor)
    {
        foreach (object i in this)
        {
            visitor.Visit(i);
        }
    }
}```
The Accept method calls Visit for each object i in the container. Since Visitor is an interface, it does not provide an implementation for the Visit operation. What a visitor actually does with an object depends on the actual class of visitor used.

Suppose that we want to print all of the objects in the container. One way to do this is to create a PrintingVisitor which prints every object it visits, and then to pass the visitor to the container by calling the Accept method. The following code shows how we can declare the PrintingVisitor class which prints an object on the console.

```csharp
public class PrintingVisitor : Visitor
{
    public void Visit(object obj)
    {
        Console.WriteLine(obj);
    }
    // ...
}
```

Finally, given an object c that is an instance of a concrete class SomeContainer that implements the Container interface, we can call the Accept method as follows:

```csharp
Container c = new SomeContainer();
// ...
c.accept(new PrintingVisitor());
```

The effect of this call is to call the Visit method of the visitor for each object in the container.

---

- The IsDone Property
- Abstract Visitors
- The AbstractContainer Class ToString Method

Next | Up | Previous | Contents | Index
The IsDone Property

As shown in Program [ ], the Visitor interface also includes the property IsDone. The IsDone property provides a get accessor that is used to determine whether a visitor has finished its work. That is, the IsDone method returns the bool value true if the visitor "is done."

The idea is this: Sometimes a visitor does not need to visit all the objects in a container. That is, in some cases, the visitor may be finished its task after having visited only some of the objects. The IsDone method can be used by the container to terminate the Accept method early like this:

```csharp
public class SomeContainer : Container
{
    public void Accept(Visitor visitor)
    {
        foreach (object i in this)
        {
            if (visitor.IsDone)
                return;
            visitor.Visit(i);
        }
        // ...
    }
}
```

To illustrate the usefulness of IsDone, consider a visitor which visits the objects in a container with the purpose of finding the first object that matches a given target object. Having found the first matching object in the container, the visitor is done and does not need to visit any more contained objects.

The following code fragment defines a visitor which finds the first object in the container that matches a given object.

```csharp
public class MatchingVisitor : Visitor
{
    private object target;
    private object found;

    public MatchingVisitor(object target)
```
The constructor of the MatchingVisitor visitor takes a reference to an object instance that is the target of the search. That is, we wish to find an object in a container that matches the target. For each object the MatchingVisitor visitor visits, it compares that object with the target and makes found point at that object if it matches. Clearly, the MatchingVisitor visitor is done when the found pointer is non-zero.

Suppose we have a container c that is an instance of a concrete container class, SomeContainer, that implements the Container interface; and an object x that is an instance of a concrete object class, SomeObject. Then, we can call use the MatchingVisitor visitor as follows:

```java
Container c = new SomeContainer();
Object x = new SomeObject();
// ...
c.Accept(new MatchingVisitor(x));
```
Abstract Visitors

Program defines an abstract class called AbstractVisitor that implements the Visitor interface. This class is provided simply as a convenience. It provides a default implementation for the Visit method which does nothing and a default implementation for the IsDone method which always returns false.

```csharp
public abstract class AbstractVisitor : Visitor
{
    public virtual void Visit(object obj)
    {
    }
    public virtual bool IsDone
    {
        get { return false; }
    }
}
```

Program: AbstractVisitor class.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
One of the methods defined in the C# object class is the ToString method. Consequently, every C# object supports the ToString method. The ToString method is required to return a string that represents the object "textually." It is typically invoked in situations where it is necessary to print out the representation of an object.

Program defines the ToString method of the AbstractContainer class. This method is provided to simplify the implementation of classes derived from the AbstractContainer class. The default behavior is to print out the name of the class and then to print each of the elements in the container, by using the Accept method together with a visitor.
The `ToString` method makes use of a `StringBuilder` to accumulate the textual representations of the objects in the container. A C# `StringBuilder` is like a C# string, except it can be modified. In particular, the `StringBuilder` class defines various `Append` methods that can be used to append text to the builder.

In this case, we use a visitor to do the appending. That is, the `Visit` method appends to the string
The AbstractContainer Class ToString Method

builder the textual representation of every object that it visits. (It also makes sure to put in commas as required).

The final result returned by the ToString method consists of the name of the container class, followed by a comma-separated list of the contents of that container enclosed in braces { and }.
Enumerable Collections and Enumerators

In this section we introduce an abstraction called an enumerator. An enumerator provides the means to access one-by-one all the objects in a container. Enumerators are an alternative to using the visitors described in Section □.

The Container interface given in Program □ extends the standard C# IEnumerable interface. The following code fragment defines the IEnumerable interface.

```csharp
namespace System.Collections
{
    public interface IEnumerable
    {
        public IEnumerator GetEnumerator();
    }
}
```

Simply put, a class that is enumerable provides a method that returns an enumerator.

The idea is that for every concrete container class we will also implement a related class that implements the standard C# IEnumerator interface. The following code fragment defines the IEnumerator interface.

```csharp
namespace System.Collections
{
    public interface IEnumerator
    {
        public bool MoveNext();
        public object Current { get; }
        public void Reset();
    }
}
```

The interface comprises two methods, MoveNext and Reset, and a property, Current.
In order to understand the desired semantics, it is best to consider first an example which illustrates the use of an enumerator. Consider a concrete container class, say `SomeContainer`, that implements the `Container` interface. The following code fragment illustrates the use of the enumerator to access one-by-one the objects contained in the container:

```csharp
Container c = new SomeContainer();
// ...
IEnumerator e = c.GetEnumerator();
while (e.MoveNext())
{
    object obj = e.Current;
    Console.WriteLine(obj);
}
e.Reset();
```

In order to have the desired effect, the members of the `IEnumerator` interface must have the following behaviors:

**MoveNext**

The `MoveNext` method is called in the loop-termination test part of the `while` statement. The `MoveNext` conceptually moves the enumerator to the next object to be visited. The `MoveNext` method returns `true` when there are still more objects in the container to be visited and it returns `false` when all of the contained objects have been visited.

**Current**

The `Current` property provides a `get` accessor that returns the next object in the container to be visited. If there is no current object to be visited, this accessor throws a `InvalidOperationException` exception.

**Reset**

The `Reset` method resets the enumerator so that all the objects in the container can be visited again.

Given these semantics for the enumerator methods, the program fragment shown above systematically visits all of the objects in the container and prints each one on its own line of the console.

One of the advantages of using an enumerator object which is separate from the container is that it is possible then to have more than one enumerator associated with a given container. This provides greater flexibility than possible using a visitor, since only one visitor can be accepted by a container at any given time. For example, consider the following code fragment:

```csharp
Container c = new SomeContainer();
// ...
```
Enumerable Collections and Enumerators

IEnumerator e1 = c.GetEnumerator();
while (e1.MoveNext())
{
    object obj1 = e1.Current;
    IEnumerator e2 = c.GetEnumerator();
    while (e2.MoveNext())
    {
        object obj2 = e2.Current;
        if (obj1.Equals(obj2))
            Console.WriteLine(obj1 + "=" + obj2);
    }
}

This code compares all pairs of objects in the container c and prints out those which are equal.

A certain amount of care is required when defining and using enumerators. In order to simplify the implementation of enumerators, we shall assume that while an enumerator is in use, the associated container will not be modified.

---

Enumerators and **foreach**

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Enumerators and foreach

The C# compiler automatically generates code to use an enumerator when the `foreach` statement is used. Thus, given an object `c` that is an instance of a concrete class `SomeContainer` that implements the `Container` interface, we can use the `foreach` statement to enumerate the objects in the container as follows:

```csharp
Container c = new SomeContainer();
// ...
foreach (object obj in c)
{
    Console.WriteLine(obj);
}
```

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
A searchable container is an extension of the container abstraction. It adds to the interface provided for containers methods for putting objects in and taking objects out, for testing whether a given object is in the container, and a method to search the container for a given object.

The definition of the SearchableContainer interface is shown in Program 1. The SearchableContainer interface extends the Container interface given in Program 1. It adds four methods to the inherited interface.

```csharp
public interface SearchableContainer : Container
{
    bool IsMember(ComparableObject obj);
    void Insert(ComparableObject obj);
    void Withdraw(ComparableObject obj);
    ComparableObject Find(ComparableObject obj);
}
```

Program: SearchableContainer interface.

The IsMember method is a bool-valued method which takes as its argument any object derived from the ComparableObject abstract base class. The purpose of this method is to test whether the given object instance is in the container.

The purpose of the Insert method is to put an object into the container. The Insert method takes a ComparableObject and inserts it into the container. Similarly, the Withdraw method is used to remove an object from a container. The argument refers to the object to be removed.

The final method, Find, is used to locate an object in a container and to return a reference to that object. In this case, it is understood that the search is to be done using the comparison methods defined in the ComparableObject class. That is, the Find method is not to be implemented as a search of the container for the given object but rather as a search of the container for an object that compares equal to the given object.
This is an important subtlety in the semantics of \texttt{Find}: The search is not for the given object, but rather for an object that compares equal to the given object. These semantics are particularly useful when using \textit{associations}, which are defined in Section \ref{sec:associations}.

In the event that the \texttt{Find} method fails to find an object equal to the specified object, then it will return \texttt{null}. Therefore, the user of the \texttt{Find} method should test explicitly the returned value to determine whether the search was successful. Also, the \texttt{Find} method does \textit{not} remove the object it finds from the container. An explicit call of the \texttt{Withdraw} method is needed to actually remove the object from the container.
Abstract Searchable Containers

Program introduces an *abstract class* called `AbstractSearchableContainer`. It is intended to be used as the base class from which concrete searchable container realizations are derived. As illustrated in Figure, the `AbstractSearchableContainer` class extends the `AbstractContainer` class (defined in Program) and it implements the `SearchableContainer` interface (defined in Program).

```csharp
public abstract class AbstractSearchableContainer :
    AbstractContainer, SearchableContainer
{
    public abstract bool IsMember(ComparableObject obj);
    public abstract void Insert(ComparableObject obj);
    public abstract void Withdraw(ComparableObject obj);
    public abstract ComparableObject Find(ComparableObject obj);
}
```

Program: `AbstractSearchableContainer` class.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Associations

An association is an ordered pair of objects. The first element of the pair is called the *key*; the second element is the *value* associated with the given key.

Associations are useful for storing information in a database for later retrieval. For example, a database can be viewed as a container that holds key-and-value pairs. The information associated with a given key is retrieved from the database by searching the database for an ordered pair in which the key matches the given key.

Program introduces the Association class. The Association class concrete extension of the ComparableObject class given in Program.

```csharp
public class Association : ComparableObject
{
    protected IComparable key;
    protected object value;

    // ...
}
```

Program: Association fields.

An association has two fields, key and value. The key field is any object that implements the IComparable interface. The value field is any, arbitrary object.

Two constructors and two properties are defined in Program. The first constructor takes two arguments and initializes the key and value fields accordingly. The second constructor takes only one argument which is used to initialize the key field--the value field is set to null.
public class Association : ComparableObject
{
    protected IComparable key;
    protected object value;

    public Association(IComparable key, object value)
    {
        this.key = key;
        this.value = value;
    }

    public Association(IComparable key) : this(key, null)
    {
    }

    public IComparable Key
    {
        get
        {
            return key;
        }
    }

    public object Value
    {
        get
        {
            return value;
        }
    }

    // ...
}

Program: Association constructors.

The Key and the Value property each provides a get accessor. The former returns the value of the key field; the latter, returns the value of the value field.

The remaining methods of the Association class are defined in Program. The CompareTo method is used to compare associations. Its argument is an object that is assumed to be an instance of the Association class. The CompareTo method is one place where an association distinguishes between the key and the value. In this case, the result of the comparison is based solely on the keys of the two associations--the values have no role in the comparison.
public class Association : ComparableObject
{
    protected IComparable key;
    protected object value;

    public override int CompareTo(object obj)
    {
        Association association = obj as Association;
        return key.CompareTo(association.key);
    }

    public override string ToString()
    {
        string result = "Association {" + key;
        if (value != null)
            result += ", " + value;
        return result + "}";
    }

    // ...
}

Program: Association methods.

Program also defines a ToString method. The purpose of the ToString method is to return a textual representation of the association. In this case, the implementation is trivial and needs no further explanation.
Exercises

1. Specify the set of values and the set of operations provided by each of the following C# types:
   1. char,
   2. int,
   3. double, and
   4. string.

2. What are the features of C# that facilitate the creation of user-defined data types.

3. Explain how each of the following C# features supports polymorphism:
   1. interfaces,
   2. abstract classes,
   3. inheritance, and
   4. operator overloading.

4. Suppose we define two concrete classes, A and B, both of which are derived from the ComparableObject class declared in Program. Furthermore, let a and b be instances of classes A and B (respectively) declared as follows:

   ```csharp
   public class A : ComparableObject { ... };  
   public class B : ComparableObject { ... };  
   ComparableObject a = new A();  
   ComparableObject b = new B();
   ```

   Give the sequence of methods called in order to evaluate a comparison such as `a < b`. Is the result of the comparison true or false? Explain.

5. Consider the ComparableInt32 wrapper class defined in Program. Explain the operation of the following program fragment:

   ```csharp
   ComparableObject i = 5;  
   ComparableObject j = 7;  
   bool result = i < j;
   ```

6. Let c be an instance of some concrete class derived from the AbstractContainer class given in Program. Explain how the statement

   ```csharp
   Console.WriteLine(c);
   ```
Exercises

prints the contents of the container on the console.

7. Suppose we have a container \( c \) (i.e., an instance of some concrete class derived from the AbstractContainer class defined in Program [1]) which among other things happens to contain itself. What happens when we invoke the ToString method on \( c \)?

8. Enumerators and visitors provide two ways to do the same thing--to visit one-by-one all the objects in a container. Give an implementation for the Accept method of the AbstractContainer class that uses an enumerator.

9. Is it possible to implement an enumerator using a visitor? Explain.

10. Suppose we have a container which we know contains only instances of the ComparableInt32 class defined in Program [1]. Design a Visitor which computes the sum of all the integers in the container.

11. Consider the following pair of Associations:

    ComparableObject \( a = \) new Association(3, 4);
    ComparableObject \( b = \) new Association(3);

    Give the sequence of methods called in order to evaluate a comparison such as ``\( a == b \)''. Is the result of the comparison true or false? Explain.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Projects

1. Design and implement suitable wrapper classes that extend the `ComparableObject` base class for the C# types `bool`, `byte`, `short`, `long`, `float`, and `void`.
2. Using *visitors*, devise an implementation for the `IsMember` property and the `Find` method of the `AbstractSearchableContainer` class declared in Program [link].
3. Using an *enumerator*, devise an implementation for the `IsMember` property and the `Find` method of the `AbstractSearchableContainer` class declared in Program [link].
4. Devise a scheme using visitors whereby all of the objects contained in one searchable container can be removed from it and transferred to another container.
5. A *bag* is a simple container that can hold a collection of objects. Design and implement a concrete class called `Bag` that extends the `AbstractSearchableContainer` class defined in Program [link]. Use the `DynamicArray` class given in Chapter [link] to keep track of the contents of the bag.
6. Repeat Project [link], this time using the `LinkedList` class given in Chapter [link].
7. In Java it is common to use an *enumeration* as the means to iterate through the objects in a container. In C# we can define an enumeration like this:

   ```csharp
   public interface Enumeration
   {
       bool hasMoreElements { get; }
       object nextElement();
   }
   
   Given an enumeration `e` from some container `c`, the contents of `c` can be printed like this:
   ```
   ```csharp
   while (e.hasMoreElements)
       Console.WriteLine(e.nextElement());
   ```

   Devise a wrapper class to encapsulate a C# enumerator and provide the functionality of a Java enumeration.
Stacks, Queues, and Deques

In this chapter we consider several related abstract data types--stacks, queues, and deques. Each of these can be viewed as a pile of items. What distinguishes each of them is the way in which items are added to or removed from the pile.

In the case of a stack, items are added to and removed from the top of the pile. Consider the pile of papers on your desk. Suppose you add papers only to the top of the pile or remove them only from the top of the pile. At any point in time, the only paper that is visible is the one on top. What you have is a stack.

Now suppose your boss comes along and asks you to complete a form immediately. You stop doing whatever it is you are doing, and place the form on top of your pile of papers. When you have filled-out the form, you remove it from the top of the stack and return to the task you were working on before your boss interrupted you. This example illustrates that a stack can be used to keep track of partially completed tasks.

A queue is a pile in which items are added an one end and removed from the other. In this respect, a queue is like the line of customers waiting to be served by a bank teller. As customers arrive, they join the end of the queue while the teller serves the customer at the head of the queue. As a result, a queue is used when a sequence of activities must be done on a first-come, first-served basis.

Finally, a deque extends the notion of a queue. In a deque, items can be added to or removed from either end of the queue. In a sense, a deque is the more general abstraction of which the stack and the queue are just special cases.

As shown in Figure, we view stacks, queues, and deques as containers. This chapter presents a number of different implementation alternatives for stacks, queues, and deques. All of the concrete classes presented are extensions of the AbstractContainer class defined in Chapter.
Figure: Object class hierarchy.

- **Stacks**
- **Queues**
- **Deques**
- **Exercises**
- **Projects**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Stacks

The simplest of all the containers is a *stack*. A stack is a container which provides exactly one method, `Push`, for putting objects into the container; and one method, `Pop`, for taking objects out of the container. Figure illustrates the basic idea.

**Figure:** Basic stack operations.

Objects which are stored in a stack are kept in a pile. The last item put into the stack is at the top. When an item is pushed into a stack, it is placed at the top of the pile. When an item popped, it is always the top item which is removed. Since it is always the last item to be put into the stack that is the first item to be removed, a stack is a *last-in, first-out* or *LIFO* data structure.

In addition to the `Push` and `Pop` operations, the typical stack implementation also has a property called `Top` that provides a `get` accessor that returns the item at the top of the stack without removing it from the stack.
Program defines the Stack interface. The Stack interface extends the Container interface defined in Program . Hence, it comprises all of the methods inherited from Container plus the three methods Top, Push, and Pop.

```java
public interface Stack : Container
{
    object Top { get; }
    void Push(object obj);
    object Pop();
}
```

Program: Stack interface.

When implementing a data structure, the first issue to be addressed is which foundational data structure(s) to use. Often, the choice is between an array-based implementation and a linked-list implementation. The next two sections present an array-based implementation of stacks followed by a linked-list implementation.

- Array Implementation
- Linked-List Implementation
- Applications

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Array Implementation

This section describes an array-based implementation of stacks. Program StackAsArray introduces the StackAsArray class. The StackAsArray class is a concrete class that extends the AbstractContainer class introduced in Program and implements the Stack interface defined in Program.

```csharp
public class StackAsArray : AbstractContainer, Stack
{
    protected object[] array;

    // ...
}
```

Program: StackAsArray fields.

- **Fields**
- **Constructor and Purge Methods**
- **Push and Pop Methods, Top Property**
- **Accept Method**
- **GetEnumerator Method**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Fields

The StackAsArray class contains one field--an array of objects called array. In addition, the StackAsArray class inherits the count field from the AbstractContainer class. In the array implementation of the stack, the elements contained in the stack occupy positions 0, 1, ..., count - 1 of the array.
Constructor and Purge Methods

The definitions of the StackAsArray constructor and Purge methods are given in Program 1. The constructor takes a single parameter, size, which specifies the maximum number of items that can be stored in the stack. The variable array is initialized to be an array of length size. The constructor requires $O(n)$ time to construct the array, where $n = \text{size}$.

```
public class StackAsArray : AbstractContainer, Stack
{
    protected object[] array;

    public StackAsArray(int size)
    {
        array = new object[size];
    }

    public override void Purge()
    {
        while (count > 0)
        {
            array[--count] = null;
        }
        // ...
    }
}
```

Program: StackAsArray class constructor and Purge methods.

The purpose of the Purge method is to remove all the contents of a container. In this case, the objects in the stack occupy the first count positions of the array. To empty the stack, the Purge method simply assigns the value null to the first count positions of the array. Clearly, the running time for the Purge method is $O(n)$, where $n = \text{count}$. 
Push and Pop Methods, Top Property

Program defines the Push and Pop methods and the Top property of the StackAsArray class. The first of these, Push, adds an element to the stack. It takes as its argument the object to be pushed onto the stack.

```csharp
public class StackAsArray : AbstractContainer, Stack
{
    protected object[] array;

    public void Push(object obj)
    {
        if (count == array.Length)
            throw new ContainerFullException();
        array[count++] = obj;
    }

    public object Pop()
    {
        if (count == 0)
            throw new ContainerEmptyException();
        object result = array[--count];
        array[count] = null;
        return result;
    }

    public object Top
    {
        get
        {
            if (count == 0)
                throw new ContainerEmptyException();
            return array[count - 1];
        }
    }
}```
Push and Pop Methods, Top Property

Program: StackAsArray class Push and Pop Methods, Top property.

The **Push** method first checks to see if there is room left in the stack. If no room is left, it throws a `ContainerFullException` exception. Otherwise, it simply puts the object into the array, and then increases the `count` variable by one. In a correctly functioning program, stack overflow should not occur. If we assume that overflow does not occur, the running time of the **Push** method is clearly $O(1)$.

The **Pop** method removes an item from the stack and returns that item. The **Pop** method first checks if the stack is empty. If the stack is empty, it throws a `ContainerEmptyException` exception. Otherwise, it simply decreases `count` by one and returns the item found at the top of the stack. In a correctly functioning program, stack underflow will not occur normally. The running time of the **Pop** method is $O(1)$.

Finally, the **Top** property provides a `get` accessor that returns the top item in the stack. The `get` accessor does not modify the stack. In particular, it does *not* remove the top item from the stack. The **Top** method first checks if the stack is empty. If the stack is empty, it throws a `ContainerEmptyException` exception. Otherwise, it returns the top item, which is found at position `count - 1` in the array. Assuming stack underflow does not occur normally, the running time of the **Top** method is $O(1)$.

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
**Accept Method**

Program defines the `Accept` method for the `StackAsArray` class. As discussed in Chapter, the purpose of the `Accept` method of a container is to accept a visitor and to cause it to visit one-by-one all of the contained objects.

```csharp
public class StackAsArray : AbstractContainer, Stack
{
    protected object[] array;

    public override void Accept(Visitor visitor)
    {
        for (int i = 0; i < count; ++i)
        {
            visitor.Visit(array[i]);
            if (visitor.IsDone)
                return;
        }
    }
    // ...
}
```

**Program:** `StackAsArray` class `Accept` method.

The body of the `Accept` method is simply a loop which calls the `Visit` method for each object in the stack. The running time of the `Accept` method depends on the running time of the `Visit` method. Let $T_{\text{Visit}}$ be the running time of the `Visit` method. In addition to the time for the method call, each iteration of the loop incurs a constant overhead. Consequently, the total running time for `Accept` is $nT_{\text{Visit}} + O(n)$, where $n$ is the number of objects in the container. And if $T_{\text{Visit}} = O(1)$, the total running time is to $O(n)$.
GetEnumerator Method

As discussed in Section 0, the GetEnumerator method of a Container returns an IEnumerator. An enumerator is meant to be used like this:

```csharp
Stack stack = new StackAsArray(57);
stack.Push(3);
stack.Push(1);
stack.Push(4);
IEnumerator e = stack.GetEnumerator();
while (e.MoveNext())
{
    Object obj = e.Current;
    Console.WriteLine(obj);
}
```

This code creates an instance of the StackAsArray class and assigns it to the variable stack. Next, several ints are pushed onto the stack. Finally, an enumerator is used to systematically print out all of the objects in the stack.

Program 0 defines GetEnumerator method of the StackAsArray class. The GetEnumerator method returns a new instance of the private class StackAsArray.Enumerator that implements the IEnumerator interface (lines 5-32).
public class StackAsArray : AbstractContainer, Stack
{
    protected object[] array;

    private class Enumerator : IEnumerator
    {
        StackAsArray stack;
        protected int position = -1;

        internal Enumerator(StackAsArray stack)
        { this.stack = stack; }

        public bool MoveNext()
        {
            if (++position == stack.count)
                position = -1;
            return position >= 0;
        }

        public object Current
        {
            get
            {
                if (position < 0)
                    throw new InvalidOperationException();
                return stack.array[position];
            }
        }

        public void Reset()
        { position = -1; }
    }

    public override IEnumerator GetEnumerator()
    { return new Enumerator(this); }
    // ...
}
GetEnumerator Method

**Program:** StackAsArray class GetEnumerator method.

The Enumerator class has two fields, stack and position. The stack field refers to the stack whose elements are being enumerated. The position field is used to keep track of the position in the array of the current object.

The `MoveNext` method is called in the loop termination test of the `while` loop given above. The purpose of `MoveNext` method is to advance the enumerator to the next object in the stack. The enumerator resets the position to -1 and returns `false` when there are no more elements. Clearly, the running time of `MoveNext` is $O(1)$.

The `Current` property provides a `get` accessor that returns the current object. It returns the object in the stack specified by the position field provided that the value of the position field is non-negative. Otherwise, it throws a `InvalidOperationException` exception. Clearly, the running time of `Current` is also $O(1)$. 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Linked-List Implementation

In this section we will examine a linked-list implementation of stacks that makes use of the LinkedList data structure developed in Chapter 1. Program 2 introduces the StackAsLinkedList class. The StackAsLinkedList class is a concrete class that extends the AbstractContainer class introduced in Program 3 and implements the Stack interface defined in Program 4.

```
public class StackAsLinkedList : AbstractContainer, Stack
{
    protected LinkedList list;

    // ...
}
```

**Program**: StackAsLinkedList fields.

- **Fields**
- **Constructor and Purge Methods**
- **Push and Pop Methods, Top Property**
- **Accept Method**
- **GetEnumerator Method**
Fields

The implementation of the StackAsLinkedList class makes use of one field--an instance of the LinkedList class called list. In addition, the StackAsLinkedList class inherits the count field from the AbstractContainer class. This list is used to keep track of the objects in the stack. As a result, there are as many elements in the linked list as there are objects in the stack.
Constructor and Purge Methods

The definitions of the constructor and the Purge methods of the StackAsLinkedList class are shown in Program 1. With a linked-list implementation, it is not necessary to preallocate storage space for the objects in the stack. Space is allocated dynamically and incrementally on the basis of demand.

```csharp
public class StackAsLinkedList : AbstractContainer, Stack
{
    protected LinkedList list;

    public StackAsLinkedList()
    {
        list = new LinkedList();
    }

    public override void Purge()
    {
        list.Purge();
        count = 0;
    }

    // ...
}
```

Program: StackAsLinkedList class constructor and Purge methods.

The constructor simply creates an empty LinkedList and assigns it to the list field. Since an empty list can be created in constant time, the running time of the StackAsLinkedList constructor is $O(1)$.

The Purge method of the StackAsLinkedList class simply calls the Purge method of the LinkedList class. The Purge method of the LinkedList class discards all the elements of the list in constant time. Consequently, the running time of the Purge method is also $O(1)$. 
Push and Pop Methods, Top Property

The Push and Pop methods, and the Top property of the StackAsLinkedList class are defined in Program □.

```
public class StackAsLinkedList : AbstractContainer, Stack
{
    protected LinkedList list;

    public void Push(object obj)
    {
        list.Prepend(obj);
        ++count;
    }

    public object Pop()
    {
        if (count == 0)
            throw new ContainerEmptyException();
        object result = list.First;
        list.Extract(result);
        --count;
        return result;
    }

    public object Top
    {
        get
        {
            if (count == 0)
                throw new ContainerEmptyException();
            return list.First;
        }
    }
```

Push and Pop Methods, Top Property

Program: StackAsLinkedList class Push and Pop Methods, Top property.

The implementation of Push is trivial. It takes as its argument the object to be pushed onto the stack and simply prepends that object to the linked list list. Then, one is added to the count variable. The running time of the Push method is constant, since the Prepend method has a constant running time, and updating the count only takes $O(1)$ time.

The Pop method is implemented using the First property and the Extract method of the LinkedList class. The First property is used to get the first item in the linked list. The First get accessor runs in constant time. The Extract method is then called to extract the first item from the linked list. In the worst case, Extract requires $O(n)$ time to extract an item from a linked list of length $n$. But the worst-case time arises only when it is the last element of the list which is to be extracted. In the case of the Pop method, it is always the first element that is extracted. This can be done in constant time. Assuming that the exception which is raised when Pop is called on an empty list does not occur, the running time for Pop is $O(1)$.

The definition of the Top get accessor is quite simple. It returns the first object in the linked list. Provided the linked list is not empty, the running time of Top is $O(1)$. If the linked list is empty, the Top get accessor throws a ContainerEmptyException exception.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Accept Method

The Accept method of the StackAsLinkedList class is defined in Program □. The Accept method takes a visitor and calls its Visit method one-by-one for all of the objects on the stack.

```
public class StackAsLinkedList : AbstractContainer, Stack
{
    protected LinkedList list;

    public override void Accept(Visitor visitor)
    {
        for (LinkedList.Element ptr = list.Head; ptr != null; ptr = ptr.Next)
        {
            visitor.Visit(ptr.Datum);
            if (visitor.IsDone)
                return;
        }
        // ...
    }
```

Program: StackAsLinkedList class Accept method.

The implementation of the Accept method for the StackAsLinkedList class mirrors that of the StackAsArray class shown in Program □. In this case, the linked list is traversed from front to back, i.e., from the top of the stack to the bottom. As each element of the linked list is encountered, the Visit method is called. If $T_{\text{Visit}}$ is the running time of the Visit method, the total running time for Accept is $nT_{\text{Visit}} + O(n)$, where $n = \text{count}$ is the number of objects in the container. If we assume that $T_{\text{Visit}} = O(1)$, the total running time is $O(n)$. 
GetEnumerator Method

Program defines GetEnumerator method of the StackAsLinkedList class. The GetEnumerator method returns an instance of the private class StackAsLinkedList.Enumerator that implements the IEnumerator interface (lines 5-34).

```csharp
public class StackAsLinkedList : AbstractContainer, Stack
{
    protected LinkedList list;

    private class Enumerator : IEnumerator
    {
        StackAsLinkedList stack;
        protected LinkedList.Element position = null;

        internal Enumerator(StackAsLinkedList stack)
        {
            this.stack = stack;
        }

        public bool MoveNext()
        {
            if (position == null)
                position = stack.list.Head;
            else
                position = position.Next;
            return position != null;
        }

        public object Current
        {
            get
            {
                if (position == null)
                    throw new InvalidOperationException();
                return position.Datum;
            }
        }
```
GetEnumerator Method

```csharp
GetEnumerator

```

The Enumerator class has two fields, stack and position. The stack field refers to the stack whose elements are being enumerated. The position field is used to keep track of the position in the linked list of the next object to be enumerated.

The purpose of the MoveNext method is to advance the enumerator to the next position and return false whether there are not more elements to be enumerated. In Program elements remain as long as the position is not null. Clearly, the running time of MoveNext is $O(1)$.

The Current property provides a get accessor that returns the current object. It returns the appropriate object in the linked list, provided that the value of the position variable is not null. Otherwise, it throws a InvalidOperationException exception. Clearly, the running time of this accessor is also $O(1)$.
Applications

Consider the following expression:

\[(5 + 9) \times 2 + 6 \times 5\]  \hspace{1cm} (6.1)

In order to determine the value of this expression, we first compute the sum 5+9 and then multiply that by 2. Then we compute the product 6 \times 5 and add it to the previous result to get the final answer. Notice that the order in which the operations are to be done is crucial. Clearly if the operations are not done in the correct order, the wrong result is computed.

The expression above is written using the usual mathematical notation. This notation is called *infix* notation. What distinguishes this notation is the way that expressions involving binary operators are written. A *binary operator* is an operator which has exactly two operands, such as + and \(\times\). In infix notation, binary operators appear *in between* their operands.

Another characteristic of *infix* notation is that the order of operations is determined by *operator precedence*. For example, the \(\times\) (multiplication) operator has higher precedence than does the + (addition) operator. When an evaluation order is desired that is different from that provided by the precedence, *parentheses*, "(" and ")", are used to override precedence rules. An expression in parentheses is evaluated first.

As an alternative to infix, the Polish logician Jan Łukasiewicz introduced notations which require neither parentheses nor operator precedence rules. The first of these, the so-called *Polish notation*, places the binary operators before their operands. For Equation (6.1) we would write:

\[+ \times + 5 \times 2 \times 5\]

This is also called *prefix* notation, because the operators are written in front of their operands.

While prefix notation is completely unambiguous in the absence of parentheses, it is not very easy to read. A minor syntactic variation on prefix is to write the operands as a comma-separated list enclosed in parentheses as follows:
While this notation seems somewhat foreign, in fact it is precisely the notation that is used for static method calls in C#:

\[ \text{Plus(} \text{Times(} \text{Plus(5,9), 2)\text{), Times(6,5)}\text{)} \]

The second form of Łukasiewicz notation is the so-called *Reverse-Polish notation* (RPN). Equation (6.2) is written as follows in RPN:

\[ 5 \ 9 \ + \ 2 \times 6 \ 5 \times + \]

This notation is also called *postfix notation* for the obvious reason—the operators are written after their operands.

Postfix notation, like prefix notation, does not make use of operator precedence nor does it require the use of parentheses. A postfix expression can always be written without parentheses that expresses the desired evaluation order. For example, the expression \(1 + 2 \times 3\), in which the multiplication is done first, is written \(1 \ 2 \ 3 \times +\); whereas the expression \((1 + 2) \times 3\) is written \(1 \ 2 \ + \ 3 \times\).
Evaluating Postfix Expressions

One of the most useful characteristics of a postfix expression is that the value of such an expression can be computed easily with the aid of a stack of values. The components of a postfix expression are processed from left to right as follows:

1. If the next component of the expression is an operand, the value of the component is pushed onto the stack.
2. If the next component of the expression is an operator, then its operands are in the stack. The required number of operands are popped from the stack; the specified operation is performed; and the result is pushed back onto the stack.

After all the components of the expression have been processed in this fashion, the stack will contain a single result which is the final value of the expression. Figure illustrates the use of a stack to evaluate the RPN expression given in Equation.

Figure: Evaluating the RPN expression in Equation using a stack.
Implementation

Program gives the implementation of a simple RPN calculator. The purpose of this example is to illustrate the use of the Stack class. The program shown accepts very simplified RPN expressions: The expression may contain only single-digit integers, the addition operator, +, and the multiplication operator, *. In addition, the operator = pops the top value off the stack and prints it on the console. Furthermore, the calculator does its computation entirely with integers.

```csharp
public class Algorithms
{
    public static void Calculator(
        TextReader reader, TextWriter writer)
    {
        Stack stack = new StackAsLinkedList();
        int i;
        while ((i = reader.Read()) > 0)
        {
            char c = (char)i;
            if (Char.IsDigit(c))
                stack.Push((int)c - (int)'0');
            else if (c == '+')
            {
                int arg2 = (int)stack.Pop();
                int arg1 = (int)stack.Pop();
                stack.Push (arg1 + arg2);
            }
            else if (c == '*')
            {
                int arg2 = (int)stack.Pop();
                int arg1 = (int)stack.Pop();
                stack.Push (arg1 * arg2);
            }
            else if (c == '=')
            {
                int arg = (int)stack.Pop();
```
int arg = (int)stack.Pop();
writer.WriteLine(arg);

Program: Stack application--a single-digit, RPN calculator.

Notice that the stack variable of the Calculator method may be any object that implements the Stack interface. Consequently, the calculator does not depend on the stack implementation used! For example, if we wish to use a stack implemented using an array, we can simply replace line 6 with the following:

Stack stack = new StackAsArray(10);

The running time of the Calculator method depends upon the number of symbols, operators, and operands, in the expression being evaluated. If there are $n$ symbols, the body of the while loop is executed $n$ times. It should be fairly obvious that the amount of work done per symbol is constant, regardless of the type of symbol encountered. This is the case for both the StackAsArray and the StackAsLinkedList stack implementations. Therefore, the total running time needed to evaluate an expression comprised of $n$ symbols is $O(n)$. 
Queues

In the preceding section we saw that a stack comprises a pile of objects that can be accessed only at one end--the top. In this section we examine a similar data structure called a *single-ended queue*. Whereas in a stack we add and remove elements at the same end of the pile, in a single-ended queue we add elements at one end and remove them from the other. Since it is always the first item to be put into the queue that is the first item to be removed, a queue is a *first-in, first-out* or *FIFO* data structure. Figure illustrates the basic queue operations.

Program defines the `Queue` interface. The `Queue` interface extends the `Container` interface defined in Program. Hence, comprises all the methods inherited from `Container` plus two methods, `Enqueue`, and `Dequeue`, and the `Head` property. As we did with stacks, we examine two queue implementations--an array-based one and a linked-list one.

```
public interface Queue : Container {
    object Head { get; }
    void Enqueue(object obj);
    object Dequeue();
}
```
Queues

Program: Queue interface.

- Array Implementation
- Linked-List Implementation
- Applications

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Array Implementation

Program introduces the QueueAsArray class. The QueueAsArray class is a concrete class that extends the AbstractContainer class introduced in Program and implements the Queue interface defined in Program.

```csharp
public class QueueAsArray : AbstractContainer, Queue
{
    protected object[] array;
    protected int head;
    protected int tail;

    // ...
}
```

Program: QueueAsArray fields.

- Fields
- Constructor and Purge Methods
- Enqueue and Dequeue Methods, Head Property
Fields

QueueAsArray objects comprise three fields--array, head, and tail. The first, array, is an array of objects that is used to hold the contents of the queue. The objects contained in the queue will be held in a contiguous range of array elements as shown in Figure (a). The fields head and tail denote the left and right ends, respectively, of this range.

![Figure: Array implementation of a queue.](http://www.brpreiss.com/books/opus6/html/page152.html)

In general, the region of contiguous elements will not necessarily occupy the leftmost array positions. As elements are deleted at the head, the position of the left end will change. Similarly, as elements are added at the tail, the position of the right end will change. In some circumstances, the contiguous region of elements will wrap around the ends of the array as shown in Figure (b).

As shown in Figure (a), the leftmost element is array[head], and the rightmost element is
When the queue contains only one element, \( \text{head} = \text{tail} \) as shown in Figure (c).

Finally, Figure (b) shows that if the queue is empty, the head position will actually be to the right of the tail position. However, this is also the situation which arises when the queue is completely full! The problem is essentially this: Given an array of length \( n \), then \( 0 \leq \text{head} < n \) and \( 0 \leq \text{tail} < n \). Therefore, the difference between the head and tail satisfies \( 0 \leq |\text{head} - \text{tail}| \leq n - 1 \). Since there are only \( n \) distinct differences, there can be only \( n \) distinct queue lengths, 0, 1, ..., \( n-1 \). It is not possible to distinguish the queue which is empty from the queue which has \( n \) elements solely on the basis of the head and tail fields.

There are two options for dealing with this problem: The first is to limit the number of elements in the queue to be at most \( n-1 \). The second is to use another field, count, to keep track explicitly of the actual number of elements in the queue rather than to infer the number from the head and tail variables. The second approach has been adopted in the implementation given below.
Constructor and Purge Methods

The definitions of the QueueAsArray class constructor and Purge methods are given in Program 2. The constructor takes a single parameter, size, which specifies the maximum number of items that can be stored in the queue. The constructor initializes the fields as follows: The array field is initialized to an array of length size and the head and tail fields, are initialized to represent the empty queue. The total running time for the QueueAsArray constructor is $O(n)$, where $n = \text{size}$.

```csharp
public class QueueAsArray : AbstractContainer, Queue
{
    protected object[] array;
    protected int head;
    protected int tail;

    public QueueAsArray(int size)
    {
        array = new object[size];
        head = 0;
        tail = size - 1;
    }

    public override void Purge()
    {
        while (count > 0)
        {
            array[head] = null;
            if (++head == array.Length)
                head = 0;
            --count;
        }
        // ...
    }
```
Program: QueueAsArray constructor and Purge methods.

The purpose of the Purge method is to remove all the contents of a container. In this case, the objects in the queue occupy contiguous array positions between head and tail. To empty the queue, the Purge method walks through the occupied array positions assigning to each one the value null as it goes. Clearly, the running time for the Purge method is $O(n)$, where $n = \text{count}$.
Enqueue and Dequeue Methods, Head Property

Program defines Enqueue and Dequeue methods and the Head property of the QueueAsArray class.

```csharp
public class QueueAsArray : AbstractContainer, Queue
{
    protected object[] array;
    protected int head;
    protected int tail;

    public object Head
    {
        get
        {
            if (count == 0)
                throw new ContainerEmptyException();
            return array[head];
        }
    }

    public void Enqueue(object obj)
    {
        if (count == array.Length)
            throw new ContainerFullException();
        ++tail == array.Length
        tail = 0;
        array[tail] = obj;
        ++count;
    }

    public object Dequeue()
    {
        if (count == 0)
            throw new ContainerEmptyException();
    }
```
Enqueue and Dequeue Methods, Head Property

Program: QueueAsArray class Enqueue and Dequeue methods, class Head property.

The Head property provides a get accessor that returns the object found at the head of the queue, having first checked to see that the queue is not empty. If the queue is empty, it throws a ContainerEmptyException exception. Under normal circumstances, we expect that the queue will not be empty. Therefore, the normal running time of this accessor is $O(1)$.

The Enqueue method takes a single argument which is a the object to be added to the tail of the queue. The Enqueue method first checks that the queue is not full--a ContainerFullException exception is thrown when the queue is full. Next, the position at which to insert the new element is determined by increasing the tail field by one modulo the length of the array. Finally, the object to be enqueued is put into the array at the correct position and the count is adjusted accordingly. Under normal circumstances (i.e., when the exception is not thrown), the running time of Enqueue is $O(1)$.

The Dequeue method removes an object from the head of the queue and returns that object. First, it checks that the queue is not empty and throws an exception when it is. If the queue is not empty, the method first sets aside the object at the head in the local variable result; it increases the head field by one modulo the length of the array; adjusts the count accordingly; and returns result. All this can be done in a constant amount of time so the running time of Dequeue is a $O(1)$. 

```csharp
if (count == 0)
    throw new ContainerEmptyException();

object result = array[head];
array[head] = null;
if (++head == array.Length)
    head = 0;
--count;
return result;
```
Linked-List Implementation

This section presents a queue implementation which makes use of the singly-linked list data structure, LinkedList, that is defined in Chapter [ ]. Program [ ] introduces the QueueAsLinkedList class. The QueueAsLinkedList extends the AbstractContainer class and implements the Queue interface.

```csharp
public class QueueAsLinkedList : AbstractContainer, Queue
{
    protected LinkedList list;

    // ...
}
```

**Program:** QueueAsLinkedList fields.

- **Fields**
- **Constructor and Purge Methods**
- **Enqueue and Dequeue Methods, Head Property**

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Fields

Just like the `StackAsLinkedList` class, the implementation of the `QueueAsLinkedList` class requires only one field--`list`. The `list` field is an instance of the `LinkedList` class. It is used to keep track of the elements in the queue.
Constructor and **Purge** Methods

Program defines the `QueueAsLinkedList` constructor and `Purge` methods. In the case of the linked list implementation, it is not necessary to preallocate storage. The constructor simply initializes the list object as an empty list. The running time of the constructor is $O(1)$.

```csharp
public class QueueAsLinkedList : AbstractContainer, Queue
{
  protected LinkedList list;

  public QueueAsLinkedList()
  {
    list = new LinkedList();
  }

  public override void Purge()
  {
    list.Purge();
    count = 0;
  }

  // ...
}
```

**Program: QueueAsLinkedList** class constructor and Purge methods.

The Purge method empties the queue by invoking the Purge method provided by the LinkedList class and then sets the count field to zero. Since a linked-list can be purged in constant time, the total running time for the Purge method is $O(1)$.
Enqueue and Dequeue Methods, Head Property

The Enqueue and Dequeue methods and the Head property of the QueueAsLinkedList class are given in Program □.

```csharp
public class QueueAsLinkedList : AbstractContainer, Queue
{
    protected LinkedList list;

    public object Head
    {
        get
        {
            if (count == 0)
            {
                throw new ContainerEmptyException();
            }
            return list.First;
        }
    }

    public void Enqueue(object obj)
    {
        list.Append(obj);
        ++count;
    }

    public object Dequeue()
    {
        if (count == 0)
        {
            throw new ContainerEmptyException();
        }
        object result = list.First;
        list.Extract(result);
        --count;
        return result;
    }
}
```
Enqueue and Dequeue Methods, Head Property

Program: QueueAsLinkedList class Enqueue and Dequeue methods, Head property.

The Head property provides a get accessor that returns the object at the head of the queue. The head of the queue is in the first element of the linked list. In Chapter we saw that the running time of LinkedList.First is a constant, Therefore, the normal running time for the Head method is $O(1)$.

The Enqueue method takes a single argument--the object to be added to the tail of the queue. The method simply calls the LinkedList.Append method. Since the running time for Append is $O(1)$, the running time of Enqueue is also $O(1)$.

The Dequeue method removes an object from the head of the queue and returns that object. First, it verifies that the queue is not empty and throws an exception when it is. If the queue is not empty, Dequeue saves the first item in the linked list in the local variable result. Then that item is extracted from the list. Using the LinkedList class from Chapter, the time required to extract the first item from a list is $O(1)$ regardless of the number of items in the list. As a result, the running time of Dequeue is also $O(1)$. 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Applications

The FIFO nature of queues makes them useful in certain algorithms. For example, we will see in Chapter 7 that a queue is an essential data structure for many different graph algorithms. In this section we illustrate the use of a queue in the breadth-first traversal of a tree.

Figure 7 shows an example of a tree. A tree is comprised of nodes (indicated by the circles) and edges (shown as arrows between nodes). We say that the edges point from the parent node to a child node. The degree of a node is equal to the number of children of that node. For example, node A in Figure 7 has degree three and its children are nodes B, C, and D. A child and all of its descendents is called a subtree.

Figure: A tree.

One way to represent such a tree is to use a collection of linked structures. Consider the following interface definition which is an abridged version of the Tree interface described in Chapter 7.

```java
public interface Tree
{
    object Key { get; }
    int Degree { get; }
    Tree GetSubtree(int i);
}
```
Applications

};

Each node in a tree is represented by an object that implements the Tree interface. The \texttt{Key} property provides a get accessor that returns an object which represents the contents of the node. E.g. in Figure \ref{fig:tree}, each node carries a one-character label so the \texttt{Key} property would return a \texttt{char} value that represents that label. The \texttt{Degree} property provides a get accessor that returns the degree of the node and the \texttt{GetSubtree} method takes an \texttt{int} argument \texttt{i} and returns the corresponding child of that node.

One of the essential operations on a tree is a \textit{tree traversal}. A traversal \textit{visits} one-by-one all the nodes in a given tree. To \textit{visit a node} means to perform some computation using the information contained in that node--e.g., print the key. The standard tree traversals are discussed in Chapter \ref{chap:traversals}. In this section we consider a traversal which is based on the levels of the nodes in the tree.

Each node in a tree has an associated level which arises from the position of that node in the tree. For example, node A in Figure \ref{fig:tree} is at level 0, nodes B, C, and D are at level 1, etc. A \textit{breadth-first traversal} visits the nodes of a tree in the order of their levels. At each level, the nodes are visited from left to right. For this reason, it is sometimes also called a \textit{level-order traversal}. The breadth-first traversal of the tree in Figure \ref{fig:tree} visits the nodes from A to L in alphabetical order.

One way to implement a breadth-first traversal of a tree is to make use of a queue as follows: To begin the traversal, the root node of the tree is enqueued. Then, we repeat the following steps until the queue is empty:

1. Dequeue and visit the first node in the queue.
2. Enqueue its children in order from left to right.

Figure \ref{fig:queue} illustrates the breadth-first traversal algorithm by showing the contents of the queue immediately prior to each iteration.
Applications

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>G</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>H</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>I</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>J</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>K</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>L</td>
</tr>
</tbody>
</table>

**Figure:** Queue contents during the breadth-first traversal of the tree in Figure 1.

- **Implementation**
Implementation

Program defines the method BreadthFirstTraversal. This method as its argument any object that implements the Tree interface. The idea is that the method is passed the root of the tree to be traversed. The algorithm makes use of the QueueAsLinkedList data structure, which was defined in the preceding section, to hold the appropriate tree nodes.

The running time of the BreadthFirstTraversal algorithm depends on the number of nodes in the tree which is being traversed. Each node of the tree is enqueued exactly once--this requires a constant amount of work. Furthermore, in each iteration of the loop, each node is dequeued exactly once--again a constant amount of work. As a result, the running time of the BreadthFirstTraversal algorithm is $O(n)$ where $n$ is the number of nodes in the traversed tree.

```
public class Algorithms {
    public static void BreadthFirstTraversal(Tree tree) {
        Queue queue = new QueueAsLinkedList();
        queue.Enqueue(tree);
        while (!queue.IsEmpty)
            {
            Tree t = (Tree)queue.Dequeue();
            Console.WriteLine(t.Key);
            for (int i = 0; i < t.Degree; ++i)
                {
                Tree subTree = t.GetSubtree(i);
                queue.Enqueue(subTree);
                }
            }
    }
}
```

Program: Queue application--breadth-first tree traversal.
Deques

In the preceding section we saw that a queue comprises a pile of objects into which we insert items at one end and from which we remove items at the other end. In this section we examine an extension of the queue which provides a means to insert and remove items at both ends of the pile. This data structure is a deque. The word deque is an acronym derived from double-ended queue.

Figure illustrates the basic deque operations. A deque provides three operations which access the head of the queue, Head, EnqueueHead and DequeueHead, and three operations to access the tail of the queue, Tail, EnqueueTail and DequeueTail.

Program defines the Deque interface. The Deque interface extends the Container interface defined in Program. Hence, it comprises all of the methods inherited from Container plus four methods, EnqueueHead, DequeueHead, EnqueueTail, and DequeueTail, and two properties, Head and Tail.
public interface Deque : Container
{
    object Head { get; }
    object Tail { get; }
    void EnqueueHead(object obj);
    void EnqueueTail(object obj);
    object DequeueHead();
    object DequeueTail();
}

**Program:** Deque interface.
Array Implementation

Program introduces an array implementation of a deque. The DequeAsArray class extends the QueueAsArray class introduced in Program and implements the Deque interface defined in Program. The QueueAsArray class provides almost all the required functionality. Only five of the six operations introduced in the Deque interface need to be implemented.

- The ``Head" Operations
- The ``Tail" Operations

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
The `Head` Operations

Program \[
\text{DequeAsArray class `Head` operations.}
\]

The `EnqueueHead` method takes a single argument which is the object to be added to the head of the deque. The `EnqueueHead` method first checks that the deque is not full--a `ContainerFullException` exception is thrown when the deque is full. Next, the position at which to insert the new element is determined by decreasing the `head` field by one modulo the length of the array. Finally, the object to be enqueued is put into the array at the correct position and the `count` is adjusted accordingly. Under normal circumstances (i.e., when the exception is not thrown), the running time of `EnqueueHead` is \(O(1)\).
The ``Tail'' Operations

Program defines the Tail property and the EnqueueTail and DequeueTail methods of the DequeAsArray class.

```csharp
public class DequeAsArray : QueueAsArray, Deque
{
    public object Tail
    {
        get
        {
            if (count == 0)
                throw new ContainerEmptyException();
            return array[tail];
        }
    }

    public void EnqueueTail(object obj)
    { Enqueue(obj); }

    public object DequeueTail()
    {
        if (count == 0)
            throw new ContainerEmptyException();
        object result = array[tail];
        array[tail] = null;
        if (tail-- == 0)
            tail = array.Length - 1;
        --count;
        return result;
    }
    // ...
}
```
The "Tail" Operations

**Program:** DequeAsArray class "Tail" operations.

The Tail property provides a get accessor that returns the object found at the tail of the deque, having first checked to see that the deque is not empty. If the deque is empty, it throws a ContainerEmptyException exception. Under normal circumstances, we expect that the deque will not be empty. Therefore, the normal running time of this method is $O(1)$.

The EnqueueTail method simply calls the Enqueue method inherited from the QueueAsArray class. Its running time was shown to be $O(1)$.

The DequeueTail method removes an object from the tail of the deque and returns that object. First, it checks that the deque is not empty and throws an exception when it is. If the deque is not empty, the method sets aside the object at the tail in the local variable result; it decreases the tail field by one modulo the length of the array; adjusts the count accordingly; and returns result. All this can be done in a constant amount of time so the running time of DequeueTail is a constant.
Linked List Implementation

Program defines a linked-list implementation of a deque. The DequeAsLinkedList class extends the QueueAsLinkedList class introduced in Program and implements the Deque interface defined in Program. The QueueAsLinkedList implementation provides almost all of the required functionality. Only five of the six operations defined in the Deque interface need to be implemented.

```csharp
public class DequeAsLinkedList : QueueAsLinkedList, Deque
{
    public void EnqueueHead(object obj)
    {
        list.Prepend(obj);
        ++count;
    }

    public object DequeueHead()
    {
        return Dequeue();
    }
    // ...
}
```

Program: DequeAsLinkedList class "Head" operations.

- The "Head" Operations
- The "Tail" Operations
The ``Head" Operations

Program defines the methods EnqueueHead and DequeueHead. The latter is trivial to implement—it simply calls the Dequeue method inherited from the QueueAsLinkedList class.

The EnqueueHead method takes a single argument—the object to be added to the head of the deque. The method simply calls the LinkedList.prepend method. Since the running time for Prepend is $O(1)$, the running time of EnqueueHead is also $O(1)$. 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
The ``Tail'' Operations

Program defines the Tail property and the EnqueueTail and DequeueTail methods of the DequeAsArray class.

```csharp
public class DequeAsLinkedList : QueueAsLinkedList, Deque
{
    public object Tail
    {
        get
        {
            if (count == 0)
                throw new ContainerEmptyException();
            return list.Last;
        }
    }

    public void EnqueueTail(object obj)
    {
        Enqueue(obj);
    }

    public object DequeueTail()
    {
        if (count == 0)
            throw new ContainerEmptyException();
        object result = list.Last;
        list.Extract(result);
        --count;
        return result;
    }

    // ...
}
```

**Program:** DequeAsLinkedList class ``Tail'' operations.
The Tail property provides a get accessor that returns the object at the tail of the deque. The tail of the deque is in the last element of the linked list. In Chapter 0, we saw that the running time of LinkedList.getLast is a constant, Therefore, the normal running time for this accessor is $O(1)$.

The EnqueueTail method simply calls the Enqueue method inherited from the QueueAsLinkedList class. Its running time was shown to be $O(1)$.

The DequeueTail method removes an object from the tail of the deque and returns that object. First, it verifies that the deque is not empty and throws an exception when it is. If the deque is not empty, DequeueTail saves the last item in the linked list in the local variable result. Then that item is extracted from the linked list. When using the LinkedList class from Chapter 0, the time required to extract the last item from a list is $O(n)$, where $n = \text{count}$ is the number of items in the list. As a result, the running time of DequeueTail is $O(n)$. 
Doubly-Linked and Circular Lists

In the preceding section we saw that the running time of DequeueHead is $O(1)$, but that the running time of DequeueTail is $O(n)$, for the linked-list implementation of a deque. This is because the linked list data structure used, LinkedList, is a singly-linked list. Each element in a singly-linked list contains a single reference--a reference to the successor (next) element of the list. As a result, deleting the head of the linked list is easy: The new head is the successor of the old head.

However, deleting the tail of a linked list is not so easy: The new tail is the predecessor of the original tail. Since there is no reference from the original tail to its predecessor, the predecessor must be found by traversing the linked list from the head. This traversal gives rise to the $O(n)$ running time.

In a doubly-linked list, each list element contains two references--one to its successor and one to its predecessor. There are many different variations of doubly-linked lists: Figure illustrates three of them.
Doubly-Linked and Circular Lists

Figure: Doubly-linked and circular list variations.

Figure (a) shows the simplest case: Two variables, say head and tail, are used to keep track of the list elements. One of them refers to the first element of the list, the other refers to the last. The first element of the list has no predecessor, therefore that reference is null. Similarly, the last element has no successor and the corresponding reference is also null. In effect, we have two overlapping singly-linked lists which go in opposite directions. Figure also shows the representation of an empty list. In this case the head and tail variables are both null.

A circular, doubly-linked list is shown in Figure (b). A circular list is formed by making use of variables which would otherwise be null: The last element of the list is made the predecessor of the first element; the first element, the successor of the last. The upshot is that we no longer need both a head and tail variable to keep track of the list. Even if only a single variable is used, both the first and the last list elements can be found in constant time.

Finally, Figure (c) shows a circular, doubly-linked list which has a single sentinel. This variation is similar to the preceding one in that both the first and the last list elements can be found in constant time.
This variation has the advantage that no special cases are required when dealing with an empty list. Figure □ shows that the empty list is represented by a list with exactly one element—the sentinel. In the case of the empty list, the sentinel is both its own successor and predecessor. Since the sentinel is always present, and since it always has both a successor and a predecessor, the code for adding elements to the empty list is identical to that for adding elements to a non-empty list.
Exercises

1. The array-based stack implementation introduced in Program uses a fixed length array. As a result, it is possible for the stack to become full.
   1. Rewrite the Push method so that it doubles the length of the array when the array is full.
   2. Rewrite the Pop method so that it halves the length of the array when the array is less than half full.
   3. Show that the average time for both push and pop operations is \( O(1) \). **Hint:** Consider the running time required to push \( n = 2^k \) items onto an empty stack, where \( k \geq 0 \).

2. Consider a sequence \( S \) of push and pop operations performed on a stack that is initially empty. The sequence \( S \) is a valid sequence of operations if at no point is a pop operation attempted on an empty stack and if the stack is empty at the end of the sequence. Design a set of rules for generating a valid sequence.

3. Devise an implementation of the queue abstract data type **using two stacks**. Give algorithms for the Enqueue and Dequeue operations, and derive tight big-oh expressions for the running times of your implementation.

4. Write each of the following infix expressions in postfix notation:
   1. \( a + b \times c \div d \),
   2. \( a + b \times (c \div d) \),
   3. \( (a + b) \times c \div d \),
   4. \( (a + b) \times (c \div d) \),
   5. \( (a + b \times c) \div d \),
   6. \( (c \div d) \times (a + b) \).

5. Write each of the following postfix expressions in infix notation:
   1. \( w \times y \div z \times - \),
   2. \( w \times y \times z \div - \),
   3. \( w \times - y \div z \times \),
   4. \( w \times - y \times z \div \),
   5. \( w \times y \div - z \times \),
   6. \( y \times z \times w \times - \div \).

6. Devise an algorithm which translates a postfix expression to a prefix expression. **Hint:** Use a stack of strings.
7. The array-based queue implementation introduced in Program □ uses a fixed length array. As a result, it is possible for the queue to become full.
   1. Rewrite the Enqueue method so that it doubles the length of the array when the array is full.
   2. Rewrite the Dequeue method so that it halves the length of the array when the array is less than half full.
   3. Show that the average time for both enqueue and dequeue operations is $O(1)$.

8. Stacks and queues can be viewed as special cases of deques. Show how all the operations on stacks and queues can be mapped to operations on a deque. Discuss the merits of using a deque to implement a stack or a queue.

9. Suppose we add a new operation to the stack ADT called FindMinimum that returns a reference to the smallest element in the stack. Show that it is possible to provide an implementation for FindMinimum that has a worst case running time of $O(1)$.

10. The breadth-first traversal method shown in Program □ visits the nodes of a tree in the order of their levels in the tree. Modify the algorithm so that the nodes are visited in reverse. **Hint:** Use a stack.
Projects

1. Enhance the functionality of the RPN calculator given in Program in the following ways:
   1. Use double-precision, floating-point arithmetic.
   2. Provide the complete repertoire of basic operators: +, -, \times, and \div.
   3. Add an exponentiation operator and a unary negation operator.
   4. Add a clear method that empties the operand stack and a print method that prints out the contents of the operand stack.

2. Modify Program so that it accepts expressions written in prefix (Polish) notation. Hint: See Exercise.

3. Write a program to convert a postfix expression into an infix expression using a stack. One way to do this is to modify the RPN calculator program given in Program to use a stack of infix expressions. A binary operator should pop two strings from the stack and then push a string which is formed by concatenating the operator and its operands in the correct order. For example, suppose the operator is \"\ast\" and the two strings popped from the stack are \"(b+c)\" and \"a\". Then the result that gets pushed onto the stack is the string \"a*(b+c)\".

4. Devise a scheme using a stack to convert an infix expression to a postfix expression. Hint: In a postfix expression operators appear after their operands whereas in an infix expression they appear between their operands. Process the symbols in the prefix expression one-by-one. Output operands immediately, but save the operators in a stack until they are needed. Pay special attention to the precedence of the operators.

5. Modify your solution to Project so that it immediately evaluates the infix expression. That is, create an InfixCalculator method in the style of Program.

6. Consider a string of characters, S, comprised only of the characters (, ), [, ], and . We say that S is balanced if it has one of the following forms:
   - S = \"\"; i.e., S is the string of length zero,
   - S = \"(T)\",
   - S = \"[T]\",
   - S = \"{T}\",
   - S = \"TU\"
   where both T and U are balanced strings. In other words, for every left parenthesis, bracket or brace, there is a corresponding right parenthesis, bracket or brace. For example, \"{ () [ () ] }\" is balanced, but \"{ [ ] }\" is not. Write a program that uses a stack of characters to test whether a given string is balanced.

7. Design and implement a MultipleStack class which provides \(m \geq 1\) stacks in a single container.
The declaration of the class should look something like this:

```csharp
public class MultipleStack : Container
{
    public MultipleStack(int numberOfStacks);
    public void Push(object object, int whichStack);
    public object Pop(int whichStack);
    // ...
}
```

- The constructor takes a single integer argument that specifies the number of stacks in the container.
- The `Push` method takes two arguments. The first gives the object to be pushed and the second specifies the stack on which to push it.
- The `Pop` method takes a single integer argument which specifies the stack to pop.

Choose one of the following implementation approaches:

1. Keep all the stack elements in a single array.
2. Use an array of `Stack` objects.
3. Use a linked list of `Stack` objects.

8. Design and implement a class called `DequeAsDoublyLinkedList` that implements the `Deque` interface using a doubly-linked list. Select one of the approaches shown in Figure □.

9. In Section □, the `DequeAsArray` class extends the `QueueAsArray` class. Redesign the `DequeAsArray` and `QueueAsArray` components of the class hierarchy making `DequeAsArray` the base class and deriving `QueueAsArray` from it.

```
                X
               /\  \\
+              +  -
 \             /\  /\  \\
 a  b          c  d
```

**Figure:** Expression tree for \((a + b) \times (c - d)\).

10. Devise an approach for evaluating an arithmetic expression using a queue (rather than a stack). **Hint:** Transform the expression into a tree as shown in Figure □ and then do a breadth-first traversal of the tree in reverse (see Exercise □). For example, the expression \((a + b) \times (c - d)\) becomes \([d \times b] - [a +] \times\). Evaluate the resulting sequence from left to right using a queue in the same way that a postfix expression is evaluated using a stack.
Ordered Lists and Sorted Lists

The most simple, yet one of the most versatile containers is the list. In this chapter we consider lists as abstract data types. A list is a series of items. In general, we can insert and remove items from a list and we can visit all the items in a list in the order in which they appear.

In this chapter we consider two kinds of lists--ordered lists and sorted lists. In an ordered list the order of the items is significant. Consider a list of the titles of the chapters in this book. The order of the items in the list corresponds to the order in which they appear in the book. However, since the chapter titles are not sorted alphabetically, we cannot consider the list to be sorted. Since it is possible to change the order of the chapters in book, we must be able to do the same with the items of the list. As a result, we may insert an item into an ordered list at any position.

On the other hand, a sorted list is one in which the order of the items is defined by some collating sequence. For example, the index of this book is a sorted list. The items in the index are sorted alphabetically. When an item is inserted into a sorted list, it must be inserted at the correct position.

As shown in Figure, two interfaces are used to represent the different list abstractions--OrderedList and SortedList. The various list abstractions can be implemented in many ways. In this chapter we examine implementations based on the array and the linked list foundational data structures presented in Chapter.

Figure: Object class hierarchy.
Ordered Lists and Sorted Lists

- Ordered Lists
- Sorted Lists
- Exercises
- Projects

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Ordered Lists

An ordered list is a list in which the order of the items is significant. However, the items in an ordered lists are not necessarily sorted. Consequently, it is possible to change the order of items and still have a valid ordered list.

Program defines the OrderedList interface. The OrderedList interface extends the SearchableContainer interface defined in Program. Recall that a searchable container is a container that supports the following additional operations:

Insert
used to put objects into a the container;
Withdraw
used to remove objects from the container;
Find
used to locate objects in the container;
IsMember
used to test whether a given object instance is in the container.

```csharp
    public interface OrderedList : SearchableContainer
    {
        ComparableObject this[int i] { get; }
        Cursor FindPosition(ComparableObject obj);
    }
```

Program: OrderedList interface.

The OrderedList interface adds the following operations:

this[int]
used to access the object at a given position in the ordered list, and
FindPosition
used to find the position of an object in the ordered list.
The `FindPosition` method of the `List` interface takes a `ComparableObject` and searches the list for an object that matches the given one. The return value is a `Cursor`. Program 1 defines the `Cursor` interface.

```java
public interface Cursor {
    ComparableObject Datum { get; }
    void InsertAfter(ComparableObject obj);
    void InsertBefore(ComparableObject obj);
    void Withdraw();
}
```

**Program:** `Cursor` interface.

A cursor “remembers” the position of an item in a list. The Program 2 interface given in Program 3 defines the following operations:

- **Datum**
  used to access the object in the ordered list at the current cursor position;
- **InsertAfter**
  used to insert an object into the ordered list after the current cursor position;
- **InsertBefore**
  used to insert an object into the ordered list before the current cursor position; and
- **Withdraw**
  used to remove from the ordered list the object at the current cursor position.

As we did in the previous chapter with stacks, queues, and deques, we will examine two ordered list implementations--an array-based one and a linked-list one. Section 4 presents an array version of ordered lists; Section 5, an implementation using on the `LinkedList` class.

- **Array Implementation**
- **Linked-List Implementation**
- **Performance Comparison:** `OrderedListAsArray vs. ListAsLinkedList`
- **Applications**
Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Array Implementation

This section presents an array-based implementation of ordered lists. Program 1 introduces the `OrderedListAsArray` class. The `OrderedListAsArray` class extends the `AbstractSearchableContainer` class introduced in Program 1 and it implements the `OrderedList` interface defined in Program 1.

```csharp
public class OrderedListAsArray :
    AbstractSearchableContainer, OrderedList
{
    protected ComparableObject[] array;

    // ...
}
```

**Program:** `OrderedListAsArray` fields.

- Fields
- Creating a List and Inserting Items
- Finding Items in a List
- Removing Items from a List
- Positions of Items in a List
- Finding the Position of an Item and Accessing by Position
- Inserting an Item at an Arbitrary Position
- Removing Arbitrary Items by Position
Fields

The `OrderedListAsArray` class comprises one field, `array`, which is an array of `ComparableObject`s. In addition, the `OrderedListAsArray` class inherits the field `count` from `AbstractContainer`. The `array` variable is used to hold the items in the ordered list. Specifically, the items in the list are stored in array positions 0, 1, ..., *count* − 1. In an ordered list the position of an item is significant. The item at position 0 is the first item in the list; the item at position *count* − 1, the last.

An item at position *i* + 1 is the *successor* of the one at position *i*. That is, the one at *i* + 1 follows or comes after the one at *i*. Similarly, an item at position *i* is the *predecessor* of the one at position *i* + 1; the one at position *i* is said to *precede* or to come before the one at *i* + 1.
Creating a List and Inserting Items

Program 1 gives the definitions of the constructor and the Insert methods of the OrderedListAsArray class. The constructor takes a single argument which specifies the length of array to use in the representation of the ordered list. Thus if we use an array-based implementation, we need to know when a list is declared what will be the maximum number of items in that list. The constructor initializes the array variable as an array with the specified length. The running time of the constructor is clearly $O(n)$, where $n = \text{size}$.

```csharp
1 public class OrderedListAsArray :
2     AbstractSearchableContainer, OrderedList
3 {
4     protected ComparableObject[] array;
5
6     public OrderedListAsArray(int size)
7         { array = new ComparableObject[size]; }
8
9     public override void Insert(ComparableObject obj)
10     {
11         if (count == array.Length)
12             throw new ContainerFullException();
13         array[count] = obj;
14         ++count;
15     }
16     // ...
17 }
```

Program: OrderedListAsArray class constructor and Insert methods.

The Insert method is part of the interface of all searchable containers. Its purpose is to put an object into the container. The obvious question which arises is, where should the inserted item be placed in the ordered list? The simple answer is, at the end.

In Program 2 we see that the Insert method simply adds the new item to the end of the list, provided there is still room in the array. Normally, the array will not be full, so the running time of this method is
Finding Items in a List

Program defines two `OrderedListAsArray` class methods which search for an object in the ordered list. The `IsMember` method tests whether a particular object instance is in the ordered list. The `Find` method locates in the list an object which matches its argument.

```csharp
public class OrderedListAsArray :
    AbstractSearchableContainer, OrderedList
{
    protected ComparableObject[] array;

    public override bool IsMember(ComparableObject obj)
    {
        for (int i = 0; i < count; ++i)
            if ((object)array[i] == (object)obj)
                return true;
        return false;
    }

    public override ComparableObject Find(ComparableObject arg)
    {
        for (int i = 0; i < count; ++i)
            if (array[i] == arg)
                return array[i];
        return null;
    }
}
```

**Program:** OrderedListAsArray class `IsMember` and `Find` methods.

The `IsMember` method is a `bool`-valued method which takes as its argument a `ComparableObject`. This method compares the argument one-by-one with the contents of the array. Note that this method tests whether a particular object instance is contained in the ordered list.
In the worst case, the object sought is not in the list. In this case, the running time of the method is \( O(n) \), where \( n = \text{count} \) is the number of items in the ordered list.

The \texttt{Find} method also does a search of the ordered list. However, it uses the overloaded \( == \) operator to compare the items. Thus, the \texttt{Find} method searches the list for an object which matches its argument. The \texttt{Find} method returns the object found. If no match is found, it returns \texttt{null}. The running time of this method depends on the time required for the comparison operator, \( T(==) \). In the worst case, the object sought is not in the list. In this case the running time is \( n \times T(==) + O(n) \). For simplicity, we will assume that the comparison takes a constant amount of time. Hence, the running time of the method is also \( O(n) \), where \( n = \text{count} \) is the number of items in the list.

It is important to understand the subtle distinction between the search done by the \texttt{IsMember} method and that done by \texttt{Find}. The \texttt{IsMember} method searches for a specific object instance while \texttt{Find} simply looks for a matching object. Consider the following:

```c
ComparableInt32 object1 = 57;
ComparableInt32 object2 = 57;
List list = new OrderedListAsArray(1);
list.Insert(object1);
```

This code fragment creates two \texttt{ComparableInt32} class instances, both of which have the value 57. Only the first object, \texttt{object1}, is inserted into the ordered list \texttt{list}. Consequently, the method call \texttt{list.IsMember(object1)}

returns \texttt{true}; whereas the method call

\texttt{list.IsMember(object2)}

returns \texttt{false}.

On the other hand, if a search is done using the \texttt{Find} method like this:

```c
ComparableObject object3 = list.Find(object2);
```

the search will be successful! After the call, \texttt{object3} and \texttt{object1} refer to the same object.
Removing Items from a List

Objects are removed from a searchable container using the \texttt{Withdraw} method. Program defines the \texttt{Withdraw} method for the \texttt{OrderedListAsArray} class. This method takes a single argument which is the object to be removed from the container. It is the specific object instance which is removed from the container, not simply one which matches (i.e., compares equal to) the argument.

```csharp
public class OrderedListAsArray : AbstractSearchableContainer,有序List
{
    protected ComparableObject[] array;

    public override void Withdraw(ComparableObject obj)
    {
        if (count == 0)
            throw new ContainerEmptyException();
        int i = 0;
        while (i < count && (object)array[i] != (object)obj)
            ++i;
        if (i == count)
            throw new ArgumentException("object not found");
        for (; i < count - 1; ++i)
            array[i] = array[i + 1];
        array[i] = null;
        --count;
    }
}
```

Program: \texttt{OrderedListAsArray} class \texttt{Withdraw} method.

The withdraw method first needs to find the position of the item to be removed from the list. An exception is thrown if the list is empty, or if the object to be removed is not in the list. The number of iterations needed to find an object depends on its position. If the object to be removed is found at position \texttt{i}, then the search phase takes \texttt{O(i)} time.
Removing an object from position $i$ of an ordered list which is stored in an array requires that all of the objects at positions $i+1, i+2, \ldots, \text{count} - 1$, be moved one position to the left. Altogether, $\text{count} - 1 - i$ objects need to be moved. Hence, this phase takes $O(\text{count} - i)$ time.

The running time of the `Withdraw` method is the sum of the running times of the two phases, $O(i) + O(\text{count} - i)$. Hence, the total running time is $O(n)$, where $n = \text{count}$ is the number of items in the ordered list.

Care must be taken when using the `Withdraw` method. Consider the following:

```csharp
ComparableInt32 object1 = 57;
ComparableInt32 object2 = 57;
List list = new OrderedListAsArray(1);
list.Insert(object1);

To remove `object1` from the ordered list, we may write

```csharp
list.Withdraw(object1);
```

However, the call

```csharp
list.Withdraw(object2);
```

will fail because `object2` is not actually in the list. If for some reason we have lost track of `object1`, we can always write:

```csharp
list.Withdraw(list.Find(object2));
```

which first locates the object in the ordered list (`object1`) which matches `object2` and then deletes that object.
Positions of Items in a List

As shown in Program □, objects that implement the Cursor interface can be used to access, insert, and delete objects in an ordered list. Program □ defines private nested class called `OrderedListAsArray.MyCursor` that implements the Cursor interface. The idea is that instances of this class are used by the `OrderedListAsArray` class to represent the abstraction of a position in an ordered list.

```csharp
public class OrderedListAsArray :
    AbstractSearchableContainer, OrderedList
{
    protected ComparableObject[] array;

    protected class MyCursor : Cursor
    {
        private OrderedListAsArray list;
        private int offset;

        internal MyCursor(OrderedListAsArray list, int offset)
        {
            this.list = list;
            this.offset = offset;
        }

        public ComparableObject Datum
        {
            get
            {
                if (offset < 0 || offset >= list.count)
                    throw new IndexOutOfRangeException();
                return list.array[offset];
            }
        }
    }
}
```
The MyCursor class has two fields, list and offset. The list field refers to an OrderedListAsArray instance and the offset field records an offset in that list's array of objects. A single constructor is provided which simply assigns a given values to the list and offset fields. Program also defines the Datum property of the MyCursor class. This property provides a get accessor that returns the item in the array at the position record in the offset field, provided that position is valid. The running time of the accessor is simply $O(1)$. 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Finding the Position of an Item and Accessing by Position

Program defines two more operations of the `OrderedListAsArray` class, the `FindPosition` method and an `indexer` `this[int]`. The `FindPosition` method takes as its argument a `ComparableObject`. The purpose of this method is to search the ordered list for an item which matches the object, and to return its position in the form of an object that implements the `Cursor` interface. In this case, the result is an instance of the private `MyCursor` class.

```csharp
public class OrderedListAsArray :
    AbstractSearchableContainer, OrderedList
{
    protected ComparableObject[] array;

    public virtual Cursor FindPosition(ComparableObject obj)
    {
        int i = 0;
        while (i < count && array[i] != obj)
            ++i;
        return new MyCursor(this, i);
    }

    public ComparableObject this[int offset]
    {
        get
        {
            if (offset < 0 || offset >= count)
                throw new IndexOutOfRangeException();
            return array[offset];
        }
        // ... 
    }
}
```

**Program:** `OrderedListAsArray` class `FindPosition` method and `this[int]` indexer.
The search algorithm used in `FindPosition` is identical to that used in the `Find` method (Program 4). The `FindPosition` uses the overloaded `==` operator to locate a contained object which is equal to the search target. Note that if no match is found, the offset is set to the value `count`, which is one position to the right of the last item in the ordered list. The running time of `FindPosition` is identical to that of `Find`: \( n \times \mathcal{T}(\equiv) + O(n) \), where \( n = count \).

The indexer defined in Program 4 provides a `get` accessor that takes an `int` argument and returns the object in the ordered list at the specified position. In this case, the position is specified using an integer-valued subscript expression. The implementation of this method is trivial--it simply indexes into the array. Assuming the specified offset is valid, the running time of this method is \( O(1) \).
Inserting an Item at an Arbitrary Position

Two methods for inserting an item at an arbitrary position in an ordered list are declared in Program `InsertBefore` and `InsertAfter`. Both of these take one argument—a `ComparableObject`. The effects of these two methods are illustrated in Figure.

**Figure**: Inserting an item in an ordered list implemented as an array.

Figure shows that in both cases a number of items to the right of the insertion point need to be moved over to make room for the item that is being inserted into the ordered list. In the case of `InsertBefore`, items to the right *including the item at the point of insertion* are moved; for `InsertAfter`, only items to the right of the point of insertion are moved, and the new item is inserted in the array location following the insertion point.

Program gives the implementation of the `InsertAfter` method for the `OrderedListAsArray.MyCursor` class. The code for the `InsertBefore` method is identical except for one line as explained below.
Inserting an Item at an Arbitrary Position

Program: OrderedListAsArray.MyCursor class InsertAfter method.

The InsertAfter method takes one argument--a ComparableObject. The method begins by performing some simple tests to ensure that the position is valid and that there is room left in the array to do the insertion.

On line 18 the array index where the new item will ultimately be stored is computed. For InsertAfter the index is $\text{offset} + 1$ as shown in Program. In the case of InsertBefore, the value required is simply $\text{offset}$. The loop on lines 20-21 moves items over and then object being inserted is put in the array on line 22.

```java
public class OrderedListAsArray : 
    AbstractSearchableContainer, OrderedList 
{
    protected ComparableObject[] array;

    protected class MyCursor : Cursor 
    {
        private OrderedListAsArray list;
        private int offset;

        public virtual void InsertAfter(ComparableObject obj)
        {
            if (offset < 0 || offset >= list.count)
                throw new IndexOutOfRangeException();
            if (list.count == list.array.Length)
                throw new ContainerFullException();

            int insertPosition = offset + 1;

            for (int i = list.count; i > insertPosition; --i)
                list.array[i] = list.array[i - 1];
            list.array[insertPosition] = obj;
            ++list.count;
        } // ...
    } // ...

} // ...
```
If we assume that no exceptions are thrown, the running time of `InsertAfter` is dominated by the loop which moves list items. In the worst case, all the items in the array need to be moved. Thus, the running time of both the `InsertAfter` and `InsertBefore` method is $O(n)$, where $n = \text{count}$. 
Removing Arbitrary Items by Position

The final method of the `OrderedListAsArray.MyCursor` class that we will consider is the `Withdraw` method. The desired effect of this method is to remove from the ordered list the item at the position specified by the cursor.

Figure shows the way in which to delete an item from an ordered list which implemented with an array. All of the items remaining in the list to the right of the deleted item need to be shifted to the left in the array by one position.

**Figure:** Withdrawing an item from an ordered list implemented as an array.

Program gives the implementation of the `Withdraw` method. After checking the validity of the position, all of the items following the item to be withdraw are moved one position to the left in the array.
Removing Arbitrary Items by Position

Program: `OrderedListAsArray.MyCursor` class `Withdraw` method.

The running time of the `Withdraw` method depends on the position in the array of the item being deleted and on the number of items in the ordered lists. In the worst case, the item to be deleted is in the first position. In this case, the work required to move the remaining items left is $O(n)$, where $n = \text{count}$. 

```java
public class OrderedDictAsArray :
    AbstractSearchableContainer, OrderedList
{
  protected ComparableObject[] array;

  protected class MyCursor : Cursor
  {
    private OrderedDictAsArray list;
    private int offset;

    public void Withdraw()
    {
      if (offset < 0 || offset >= list.count)
        throw new IndexOutOfRangeException();
      if (list.count == 0)
        throw new ContainerEmptyException();

      int i = offset;
      while (i < list.count - 1)
      {
        list.array[i] = list.array[i + 1];
        ++i;
      }
      list.array[i] = null;
      --list.count;
    }
    // ...
  } // ...
}
```
Removing Arbitrary Items by Position

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Linked-List Implementation

This section presents a linked-list implementation of ordered lists. Program introduces the `OrderedListAsLinkedList` class. The `OrderedListAsLinkedList` class extends the `AbstractSearchableContainer` class introduced in Program and it implements the `OrderedList` interface defined in Program.

```csharp
public class OrderedListAsLinkedList :
    AbstractSearchableContainer, OrderedList
{
    protected LinkedList linkedList;

    // ...
}
```

Program: `OrderedListAsLinkedList` fields.

- **Fields**
- **Inserting and Accessing Items in a List**
- **Finding Items in a List**
- **Removing Items from a List**
- **Positions of Items in a List**
- **Finding the Position of an Item and Accessing by Position**
- **Inserting an Item at an Arbitrary Position**
- **Removing Arbitrary Items by Position**
Fields

Objects of the `OrderedListAsLinkedList` class contain one field, `linkedList`, which is a linked list of `ComparableObject`s. The `linkedList` is used to hold the items in the ordered list. Since a linked list is used, there is no notion of an inherent limit on the number of items which can be placed in the ordered list. Items can be inserted until the available memory is exhausted.
Inserting and Accessing Items in a List

Program 1 gives the code for the constructor, Insert method, and this[int] indexer of the 
OrderedListAsLinkedList class. The constructor simply creates an empty linked list. Clearly, the 
running time of the constructor is \( O(1) \).

```csharp
public class OrderedListAsLinkedList :
    AbstractSearchableContainer, OrderedList
{
    protected LinkedList linkedList;

    public OrderedListAsLinkedList()
    {
        linkedList = new LinkedList();
    }

    public override void Insert(ComparableObject obj)
    {
        linkedList.Append(obj);
        ++count;
    }

    public ComparableObject this[int offset]
    {
        get
        {
            if (offset < 0 || offset >= count)
                throw new IndexOutOfRangeException();

            LinkedList.Element ptr = linkedList.Head;
            for (int i = 0; i < offset && ptr != null; ++i)
                ptr = ptr.Next;
            return (ComparableObject)ptr.Datum;
        }
    }
    // ...
```
Inserting and Accessing Items in a List

Program: OrderedListAsLinkedList class constructor, Insert method, and this[int] indexer.

The Insert method takes a ComparableObject and adds it to the ordered list. As in the case of the ArrayAsLinkedList class, the object is added at the end of the ordered list. This is done simply by calling the Append method from the LinkedList class.

The running time of the Insert method is determined by that of Append. In Chapter this was shown to be $O(1)$. The only other work done by the Insert method is to add one to the count variable. Consequently, the total running time for Insert is $O(1)$.

Program also defines an indexer that provides a get accessor which takes an argument of type int. This method is used to access elements of the ordered list by their position in the list. In this case, the position is specified by a non-negative, integer-valued index. Since there is no way to access directly the $i^{th}$ element of linked list, the implementation of this method comprises a loop which traverses the list to find the $i^{th}$ item. The method returns a reference to the $i^{th}$ item, provided $0 \leq i < \text{count}$. Otherwise, $i$ is not a valid subscript value and the method throws an exception.

The running time of the accessor method depends on the number of items in the list and on the value of the subscript expression. In the worst case, the item sought is at the end of the ordered list. Therefore, the worst-case running time of this algorithm, assuming the subscript expression is valid, is $O(n)$, where $n = \text{count}$. 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Finding Items in a List

Program defines the `IsMember` and `Find` methods of the `ListAsLinkedList` class. The implementations of these methods are almost identical. However, they differ in two key aspects—the comparison used and the return value.

```csharp
public class OrderedListAsLinkedList :
    AbstractSearchableContainer, OrderedList
{
    protected LinkedList linkedList;

    public override bool IsMember(ComparableObject obj)
    {
        for (LinkedList.Element ptr = linkedList.Head;
            ptr != null; ptr = ptr.Next)
        {
            if ((object)ptr.Datum == (object)obj)
                return true;
        }
        return false;
    }

    public override ComparableObject Find(ComparableObject arg)
    {
        for (LinkedList.Element ptr = linkedList.Head;
            ptr != null; ptr = ptr.Next)
        {
            ComparableObject obj = (ComparableObject)ptr.Datum;
            if (obj == arg)
                return obj;
        }
        return null;
    }

    // ...
}
```
Finding Items in a List

Program: OrderedListAsLinkedList class IsMember and Find methods.

The IsMember method tests whether a particular object instance is contained in the ordered list. It returns a bool value indicating whether the object is present. The running time of this method is clearly \( O(n) \), where \( n = \text{count} \), the number of items in the ordered list.

The Find method locates an object which matches a given object. The match is determined by using the overloaded == operator. Find returns a reference to the matching object if one is found. Otherwise, it returns the null value. The running time for this method, is \( n \times T(\==) + O(n) \), where \( T(\==) \) is the time required to do the comparison, and \( n = \text{count} \) is the number of items in the ordered list. This simplifies to \( O(n) \) when the comparison can be done in constant time.
Removing Items from a List

The `Withdraw` method is used to remove a specific object instance from an ordered list. The implementation of the `Withdraw` method for the `OrderedListAsLinkedList` class is given in Program 1.

```csharp
public class OrderedListAsLinkedList : AbstractSearchableContainer, OrderedList
{
    protected LinkedList linkedList;

    public override void Withdraw(ComparableObject obj)
    {
        if (count == 0)
        {
            throw new ContainerEmptyException();
        }
        linkedList.Extract(obj);
        --count;

        // ...
    }
}
```

**Program:** `OrderedListAsLinkedList` class `Withdraw` method.

The implementation of `Withdraw` is straight-forward: It simply calls the `Extract` method provided by the `LinkedList` class to remove the specified object from `linkedList`. The running time of the `Withdraw` method is dominated by that of `Extract` which was shown in Chapter 1 to be $O(n)$, where $n$ is the number of items in the linked list.
Positions of Items in a List

Program gives the definition of a the `OrderedListAsLinkedList.MyCursor` private nested class. The `MyCursor` class implements the `Cursor` interface defined in Program. The purpose of this class is to record the position of an item in an ordered list implemented as a linked list.

```csharp
public class OrderedListAsLinkedList :
    AbstractSearchableContainer, OrderedList
{
    protected LinkedList linkedList;

    protected class MyCursor : Cursor
    {
        private OrderedListAsLinkedList list;
        LinkedList.Element element;

        internal MyCursor(OrderedListAsLinkedList list,
            LinkedList.Element element)
        {
            this.list = list;
            this.element = element;
        }

        public ComparableObject Datum
        {
            get { return (ComparableObject)element.Datum; } }
        // ...
    }
    // ...
}
```

Program: `OrderedListAsLinkedList.MyCursor` class.

The `MyCursor` class has two fields, `list` and `element`. The `list` field refers to an `OrderedListAsLinkedList` instance and the `element` refers to the linked-list element in which a
Positions of Items in a List

given item appears. Notice that this version of MyCursor is fundamentally different from the array version. In the array version, the position was specified by an offset, i.e., by an *ordinal number* that shows the position of the item in the ordered sequence. In the linked-list version, the position is specified by a reference to the element of the linked list in which the item is stored. Regardless of the implementation, both kinds of position provide exactly the same functionality because they both implement the Cursor interface.

The Datum property of the OrderedListAsLinkedList.MyCursor class is also defined in Program □. This property provides a get accessor that dereferences the element field to obtain the required item in the ordered list. The running time is clearly $O(1)$. 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Finding the Position of an Item and Accessing by Position

The `FindPosition` method of the `OrderedListAsLinkedList` class is used to determine the position of an item in an ordered list implemented as a linked list. Its result is an instance of the inner class `MyCursor`. The `FindPosition` method is defined in Program □.

```csharp
public class OrderedListAsLinkedList : AbstractSearchableContainer, OrderedList
{
    protected LinkedList linkedList;

    public virtual Cursor FindPosition(ComparableObject arg)
    {
        LinkedList.Element ptr;
        for (ptr = linkedList.Head;
             ptr != null; ptr = ptr.Next)
        {
            ComparableObject obj = (ComparableObject)ptr.Datum;
            if (obj == arg)
                break;
        }
        return new MyCursor(this, ptr);
    }

    // ...
}
```

Program: OrderedListAsLinkedList class `FindPosition` method

The `FindPosition` method takes as its argument a `ComparableObject` that is the target of the search. The search algorithm used by `FindPosition` is identical to that of `Find`, which is given in Program □. Consequently, the running time is the same: \( n \times T(=) + O(n) \), where \( T(=) \) is the time required to match two `ComparableObjects`, and \( n = \text{count} \) is the number of items in the ordered list.
Inserting an Item at an Arbitrary Position

Once having determined the position of an item in an ordered list, we can make use of that position to insert items into the middle of the list. Two methods are specifically provided for this purpose—\texttt{InsertAfter} and \texttt{InsertBefore}. Both of these take a single argument—the \texttt{ComparableObject} to be inserted into the list.

```csharp
public class OrderedListAsLinkedList : AbstractSearchableContainer, OrderedList
{
    protected LinkedList linkedList;

    protected class MyCursor : Cursor
    {
        private OrderedListAsLinkedList list;
        LinkedList.Element element;

        public virtual void InsertAfter(ComparableObject obj)
        {
            element.InsertAfter(obj);
            ++list.count;
        }
        // ...
    }
    // ...
}
```

\textbf{Program:} \texttt{OrderedListAsLinkedList.MyCursor} class \texttt{InsertAfter} method.

Program gives the implementation for the \texttt{InsertAfter} method of the \texttt{OrderedListAsLinkedList.MyCursor} class. This method simply calls the \texttt{InsertAfter} method provided by the \texttt{LinkedList} class. Assuming no exceptions are thrown, the running time for this method is $O(1)$.

The implementation of \texttt{InsertBefore} is not shown—its similarity with \texttt{InsertAfter} should be
obvious. Since it must call the InsertBefore method provided by the LinkedList class, we expect the worst case running time to be $O(n)$, where $n = \text{count}$. 
Removing Arbitrary Items by Position

The final method to be considered is the Withdraw method of the
OrderedListAsLinkedList.MyCursor class. This method removes an arbitrary item from an
ordered list, where the position of that item is specified by a cursor instance. The code for the
Withdraw method is given in Program [Program: OrderedListAsLinkedList.MyCursor class Withdraw method.]

```csharp
1 public class OrderedListAsLinkedList :
2 AbstractSearchableContainer, OrderedList
3 {
4    protected LinkedList linkedList;
5
6    protected class MyCursor : Cursor
7    {
8        private OrderedListAsLinkedList list;
9        LinkedList.Element element;
10
11        public void Withdraw()
12        {
13            list.linkedList.Extract(element.Datum);
14            --list.count;
15        }
16        // ...
17    }
18    // ...
19 }
20
Program: OrderedListAsLinkedList.MyCursor class Withdraw method.
```

The item in the linked list at the position specified by the cursor is removed by calling the Extract
method provided by the LinkedList class. The running time of the Withdraw method depends on
the running time of the Extract of the LinkedList class. The latter was shown to be $O(n)$ where $n$
is the number of items in the linked list. Consequently, the total running time is $O(n)$.
Performance Comparison: OrderedListAsArray vs. ListAsLinkedList

The running times calculated for the various operations of the two ordered list implementations, OrderedListAsArray and OrderedListAsLinkedList, are summarized below in Table. With the exception of two operations, the running times of the two implementations are asymptotically identical.

<table>
<thead>
<tr>
<th>ordered list implementation</th>
<th>OrderedList-</th>
<th>OrderedList-</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>AsArray</td>
<td>AsLinkedList</td>
</tr>
<tr>
<td>Insert</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>IsMember</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Find</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Withdraw</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>this[int]{get}</td>
<td>$O(1)$</td>
<td>$\neq O(n)$</td>
</tr>
<tr>
<td>FindPosition</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Cursor.Datum{get}</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Cursor.InsertAfter</td>
<td>$O(n)$</td>
<td>$\neq O(1)$</td>
</tr>
<tr>
<td>Cursor.InsertBefore</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Cursor.Withdraw</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

**Table:** Running times of operations on ordered lists.
The two differences are the indexer and the InsertAfter method. The indexing operation can be done constant time when using an array, but it requires $O(n)$ in a linked list. Conversely, InsertAfter requires $O(n)$ time when using an array, but can be done in constant time in the singly-linked list.

Table □ does not tell the whole story. The other important difference between the two implementations is the amount of space required. Consider first the array implementation, OrderedListAsArray. The storage required for an array was discussed in Chapter □. Using that result, the storage needed for an OrderedListAsArray which can hold at most $M$ ComparableObject is given by:

$$\text{sizeof(count)} + \text{sizeof(array)} + \text{sizeof(ComparableObject}[M]) =$$

$$\quad 2 \text{sizeof(int)} + \text{sizeof(array ref)} + M \text{sizeof(ComparableObject ref)}$$

Notice that we do not include in this calculation that space required for the objects themselves. Since we cannot know the types of the contained objects, we cannot calculate the space required by those objects.

A similar calculation can also be done for the OrderedListAsLinkedList class. In this case, we assume that the actual number of contained objects is $n$. The total storage required is given by:

$$\text{sizeof(count)} + \text{sizeof(linkedList)} + \text{sizeof(LinkedList of n items)} =$$

$$\quad \text{sizeof(int)} + n \text{sizeof(ComparableObject ref)}$$

$$\quad + (n+1) \text{sizeof(LinkedList ref)} + (n+2) \text{sizeof(LinkedList.Element ref)}$$

If we assume that integers and object references require four bytes each, the storage requirement for the OrderedListAsArray class becomes $12+4M$ bytes; and for the ListAsList class, $16+12n$ bytes. That is, the storage needed for the array implementation is $O(M)$, where $M$ is the maximum length of the ordered list; whereas, the storage needed for the linked list implementation is $O(n)$, where $n$ is the actual number of items in the ordered list. Equating the two expressions, we get that the break-even point occurs at $n=(M-1)/3$. That is, if $n<(M-1)/3$, the array version uses more memory space; and for $n>(M-1)/3$, the linked list version uses more memory space.

It is not just the amount of memory space used that should be considered when choosing an ordered list implementation. We must also consider the implications of the existence of the limit $M$. The array implementation requires a priori knowledge about the maximum number of items to be put in the ordered list. The total amount of storage then remains constant during the course of execution. On the other hand, the linked list version has no pre-determined maximum length. It is only constrained by the total amount of memory available to the program. Furthermore, the amount of memory used by the linked list version varies during the course of execution. We do not have to commit a large chunk of memory for the duration of the program.
Applications

The applications of lists and ordered lists are myriad. In this section we will consider only one--the use of an ordered list to represent a polynomial. In general, an \( n \)-order polynomial in \( x \), for non-negative integer \( n \), has the form

\[
\sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n
\]

where \( a_n \neq 0 \). The term \( a_i \) is the coefficient of the \( i^{th} \) power of \( x \). We shall assume that the coefficients are real numbers.

An alternative representation for such a polynomial consists of a sequence of ordered pairs:

\[
\{(a_0, 0), (a_1, 1), (a_2, 2), \ldots, (a_n, n)\}.
\]

Each ordered pair, \((a_i, i)\), corresponds to the term \( a_i x^i \) of the polynomial. That is, the ordered pair is comprised of the coefficient of the \( i^{th} \) term together with the subscript of that term, \( i \). For example, the polynomial \( 31 + 41x + 59x^2 \) can be represented by the sequence \( \{(31, 0), (41, 1), (59, 2)\} \).

Consider now the \( 100^{th} \)-order polynomial \( x^{100} + 1 \). Clearly, there are only two nonzero coefficients: \( a_{100} = 1 \) and \( a_0 = 1 \). The advantage of using the sequence of ordered pairs to represent such a polynomial is that we can omit from the sequence those pairs that have a zero coefficient. We represent the polynomial \( x^{100} + 1 \) by the sequence \( \{(1, 100), (1, 0)\} \).

Now that we have a way to represent polynomials, we can consider various operations on them. For example, consider the polynomial
We can compute its derivative with respect to $x$ by differentiating each of the terms to get

$$p'(x) = \sum_{i=0}^{n-1} a_i' x^i,$$

where $a_i' = (i + 1)a_{i+1}$. In terms of the corresponding sequences, if $p(x)$ is represented by the sequence

$$\{(a_0, 0), (a_1, 1), (a_2, 2), \ldots, (a_i, i), \ldots, (a_n, n)\},$$

then its derivative is the sequence

$$\{(a_1, 0), (2a_2, 1), (3a_3, 2), \ldots, (a_i, i - 1), \ldots, (na_n, n - 1)\}.$$

This result suggests a very simple algorithm to differentiate a polynomial which is represented by a sequence of ordered pairs:

1. Drop the ordered pair that has a zero exponent.
2. For every other ordered pair, multiply the coefficient by the exponent, and then subtract one from the exponent.

Since the representation of an $n^{th}$-order polynomial has at most $n+1$ ordered pairs, and since a constant amount of work is necessary for each ordered pair, this is inherently an $\Theta(n)$ algorithm.

Of course, the worst-case running time of the polynomial differentiation will depend on the way that the sequence of ordered pairs is implemented. We will now consider an implementation that makes use of the `OrderedListAsLinkedList` class. To begin with, we need a class to represent the terms of the polynomial. Program 10 gives the definition of the `Term` class and several of its methods.

```java
public class Term : ComparableObject
{
    protected double coefficient;
    protected int exponent;

    public Term(double coefficient, int exponent)
    {
        this.coefficient = coefficient;
    }
```
Each Term instance has two fields, coefficient and exponent, which correspond to the elements of the ordered pair as discussed above. The former is a double and the latter, an int.
The Term class extends the ComparableObject class introduced in Program 4. Therefore, instances of the Term class may be put into a container. Program 5 defines three methods: a constructor, CompareTo, and Differentiate. The constructor simply takes a pair of arguments and initializes the corresponding fields accordingly.

The CompareTo method is used to compare two Term instances. Consider two terms, \( a x^i \) and \( b x^j \). We define the relation \( \prec \) on terms of a polynomial as follows:

\[
  a x^i \prec b x^j \iff (i < j) \vee (i = j \land a < b)
\]

Note that the relation \( \prec \) does not depend on the value of the variable \( x \). The CompareTo method implements the \( \prec \) relation.

Finally, the Differentiate method does what its name says: It differentiates a term with respect to \( x \). Given a term such as \( (a_0, 0) \), it computes the result \((0,0)\); and given a term such as \( (a_i, i) \) where \( i > 0 \), it computes the result \( (i a_i, i - 1) \).

We now consider the representation of a polynomial. Program 6 defines the Polynomial abstract class. The class comprises three abstract methods—Add, Differentiate, and Plus. The Add method is used to add terms to a polynomial. The Differentiate method differentiates the polynomial. The Plus method is used to compute the sum of two polynomials. In addition to these methods, the addition operator \( + \) is overloaded for Polynomial operands.

```java
public abstract class Polynomial
{
  public abstract void Add(Term term);
  public abstract void Differentiate();
  public abstract Polynomial Plus(Polynomial polynomial);
  public static Polynomial operator +(Polynomial p1, Polynomial p2)
  { return p1.Plus(p2); }
}
```

**Program:** Polynomial abstract class.

Program 7 introduces the PolynomialAsOrderedList class. This concrete class implements the Polynomial interface. It has a single field of type OrderedList. In this case, an instance of the OrderedListAsLinkedList class is used to contain the terms of the polynomial.
Applications

Program: PolynomialAsOrderedList class.

Program defines the method Differentiate which has the effect of changing the polynomial to its derivative with respect to x. To compute this derivative, it is necessary to call the Differentiate method of the Term class for each term in the polynomial. Since the polynomial is implemented as a container, there is an Accept method which can be used to perform a given operation on all of the objects in that container. In this case, we define a visitor, DifferentiatingVisitor, which
assumes its argument is an instance of the `Term` class and differentiates it.

After the terms in the polynomial have been differentiated, it is necessary to check for the term \((0,0)\) which arises from differentiating \((a_0, 0)\). The `Find` method is used to locate the term, and if one is found the `Withdraw` method is used to remove it.

The analysis of the running time of the polynomial `Differentiate` method is straightforward. The running time required to differentiate a term is clearly \(O(1)\). So too is the running time of the `Visit` method of the `DifferentiatingVisitor`. The latter method is called once for each contained object. In the worst case, given an \(n^{th}\)-order polynomial, there are \(n+1\) terms. Therefore, the time required to differentiate the terms is \(O(n)\). Locating the zero term is \(O(n)\) in the worst case, and so too is deleting it. Therefore, the total running time required to differentiate a \(n^{th}\)-order polynomial is \(O(n)\).
Sorted Lists

The next type of searchable container that we consider is a sorted list. A sorted list is like an ordered list: It is a searchable container that holds a sequence of objects. However, the position of an item in a sorted list is not arbitrary. The items in the sequence appear in order, say, from the smallest to the largest. Of course, for such an ordering to exist, the relation used to sort the items must be a total order.

Program defines the SortedList interface. Like its unsorted counterpart, the SortedList interface extends the SearchableContainer interface defined in Program.

```csharp
public interface SortedList : SearchableContainer
{
    ComparableObject this[int i] { get; }
    Cursor FindPosition(ComparableObject obj);
}
```

Program: SortedList interface.

In addition to the basic repertoire of operations supported by all searchable containers, sorted lists provide the following operations:

- `this[int]` used to access the object at a given position in the sorted list; and
- `FindPosition` used to find the position of an object in the sorted list.

Sorted lists are very similar to ordered lists. As a result, we can make use of the code for ordered lists when implementing sorted lists. Specifically, we will consider an array-based implementation of sorted lists that is derived from the `OrderedListAsArray` class defined in Section, and a linked-list implementation of sorted lists that is derived from the `OrderedListAsLinkedList` class given in Section.
• Array Implementation
• Linked-List Implementation
• Performance Comparison: SortedListAsArray vs. SortedListAsList
• Applications
Array Implementation

The SortedListAsArray class is introduced in Program 1. The SortedListAsArray class extends the OrderedListAsArray class introduced in Program 1 and it implements the SortedList interface defined in Program 1.

```csharp
public class SortedListAsArray : OrderedListAsArray, SortedList
{
    // ...
}
```

Program: SortedListAsArray class.

There are no addition fields required to implement the SortedListAsArray class. That is, the fields provided by the base class OrderedListAsArray are sufficient.

- Inserting Items in a Sorted List
- Locating Items in an Array-Binary Search
- Finding Items in a Sorted List
- Removing Items from a List
Inserting Items in a Sorted List

When inserting an item into a sorted list we have as a *precondition* that the list is already sorted. Furthermore, once the item is inserted, we have the *postcondition* that the list must still be sorted. Therefore, all the items initially in the list that are larger than the item to be inserted need to be moved to the right by one position as shown in Figure.

**Figure:** Inserting an item into a sorted list implemented as an array.

Program defines the `Insert` method for the `SortedListAsArray` class. This method takes as its argument the object to be inserted in the list. Recall that the `Insert` method provided by the `ListAsLinkedList` class simply adds items at the end of the array. While this is both efficient and easy to implement, it is not suitable for the `SortedListAsArray` class since the items in the array must be end up in order.
Inserting Items in a Sorted List

Program: SortedListAsArray class Insert method.

The Insert method given in Program first checks that there is still room in the array for one more item. Then, to insert the item into the list, all the items in the list that are larger than the one to be inserted are moved to the right. This is accomplished by the loop on lines 7-12. Finally, the item to be inserted is recorded in the appropriate array position on line 13.

In the worst case, the item to be inserted is smaller than all the items already in the sorted list. In this case, all \( n = \text{count} \) items must be moved one position to the right. Therefore, the running time of the Insert method is \( O(n) \).
Locating Items in an Array-Binary Search

Given a sorted array of items, an efficient way to locate a given item is to use a binary search. The `FindOffset` method of the `SortedListAsArray` class defined in Program uses a binary search to locate an item in the array which matches a given item.

```csharp
public class SortedListAsArray : OrderedListAsArray, SortedList
{
    protected int FindOffset(ComparableObject obj)
    {
        int left = 0;
        int right = count - 1;

        while (left <= right)
        {
            int middle = (left + right) / 2;

            if (obj > array[middle])
                left = middle + 1;
            else if (obj < array[middle])
                right = middle - 1;
            else
                return middle;
        }
        return -1;
    }
}
```

Program: `SortedListAsArray` class `FindOffset` method.

The binary search algorithm makes use of a search interval to determine the position of an item in the sorted list. The search interval is a range of array indices in which the item being sought is expected to be found. The initial search interval is $0 \ldots \text{count} - 1$. The interval is iteratively narrowed by comparing
Locating Items in an Array-Binary Search

the item sought with the item found in the array at the middle of the search interval. If the middle item matches the item sought, then we are done. Otherwise, if the item sought is less than the middle item, then we can discard the middle item and the right half of the interval; if the item sought is greater than the middle item, we can discard the middle item and the left half of the interval. At each step, the size of the search interval is approximately halved. The algorithm terminates either when the item sought is found, or if the size of the search interval becomes zero.

In the worst case, the item sought is not in the sorted list. Specifically, the worst case occurs when the item sought is smaller than any item in the list because this case requires two comparisons in each iteration of the binary search loop. In the worst case, \( \lceil \log n \rceil + 2 \) iterations are required. Therefore, the running time of the \texttt{FindOffset} method is \( \lceil \log n \rceil + 2 \times (T_{<r} + T_{<e}) + O(\log n) \), where \( T_{<r} \) and \( TGT \) represents the running times required to compare two \texttt{ComparableObject} instances. If we assume that \( T_{<r} = O(1) \) and \( T_{<e} = O(1) \), then the total running time is simply \( O(\log n) \), where \( n = \text{count} \).

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Finding Items in a Sorted List

Program defines the two methods used to locate items in a sorted list. Both of these methods make use of the FindOffset method described above.

```
public class SortedListAsArray : OrderedListAsArray, SortedList
{
    public override ComparableObject Find(ComparableObject obj)
    {
        int offset = FindOffset(obj);

        if (offset >= 0)
            return array[offset];
        else
            return null;
    }

    private new class MyCursor : OrderedListAsArray.MyCursor
    {
        internal MyCursor(SortedListAsArray list, int index) :
            base(list, index)
        {
        }
        public override void InsertAfter(ComparableObject obj)
        {
            throw new InvalidOperationException();
        }
        public override void InsertBefore(ComparableObject obj)
        {
            throw new InvalidOperationException();
        }
    }

    public override Cursor FindPosition(ComparableObject obj)
    {
        return new MyCursor(this, FindOffset(obj));
    }
    // ...
}
```

Program: SortedListAsArray class Find and FindPosition methods.
Finding Items in a Sorted List

The **Find** method takes a given object and finds the object contained in the sorted list which matches (i.e., compares equal to) the given one. It calls `FindOffset` to determine by doing a binary search the array index at which the matching object is found. `Find` returns a reference to the matching object, if one is found; otherwise, it returns `null`. The total running time of `Find` is dominated by `FindOffset`. Therefore, the running time is $O(\log n)$.

The **FindPosition** method also takes an object, but it returns a `Cursor` instead. `FindPosition` determines the position in the array of an object which matches its second argument.

The implementation of `FindPosition` is trivial: It calls `FindOffset` to determine the position at which the matching object is found and returns an instance of the private class `MyCursor`. (The `MyCursor` class is derived from the class of the same name shown in Program [ ].) The `MyCursor` overrides the inherited `InsertAfter` and `InsertBefore` methods with methods that throw an `InvalidOperationException`. These insert operations are not provided for sorted lists because they allow arbitrary insertion, but arbitrary insertions do not necessarily result in sorted lists.

The total running time of the `FindPosition` method is dominated by `FindOffset`. Therefore like `Find`, the running time of `FindPosition` is $O(\log n)$.
Removing Items from a List

The purpose of the `Withdraw` method is to remove an item from the sorted list. Program defines the `Withdraw` method which takes an object and removes it from the sorted list.

```csharp
public class SortedListAsArray : OrderedListAsArray, SortedList
{
    public override void Withdraw(ComparableObject obj)
    {
        if (count == 0)
            throw new ContainerEmptyException();

        int offset = FindOffset(obj);

        if (offset < 0)
            throw new ArgumentException("object not found");

        int i;
        for (i = offset; i < count - 1; ++i)
            array[i] = array[i + 1];
        array[i] = null;
        --count;

        // ...
    }
}
```

**Program:** `SortedListAsArray` class `Withdraw` method.

The `Withdraw` method makes use of `FindOffset` to determine the array index of the item to be removed. Removing an object from position \(i\) of an ordered list which is stored in an array requires that all of the objects at positions \(i+1, i+2, \ldots, \text{count} - 1\), be moved one position to the left. The worst case is when \(i=0\). In this case, \(\text{count} - 1\) items need to be moved to the left.

Although the `Withdraw` method is able to make use of `FindOffset` to locate the position of the item
Removing Items from a List

to be removed in $O(\log n)$ time, the total running time is dominated by the left shift, which is $O(n)$ in the worst case. Therefore, the running time of withdraw is $O(n)$.  

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno
Linked-List Implementation

This section presents a linked-list implementation of sorted lists that is derived from the `OrderedListAsLinkedList` class given in Section □. The `SortedListAsLinkedList` class is introduced in Program □. The `SortedListAsLinkedList` extends the `OrderedListAsLinkedList` class introduced in Program □ and it implements the `SortedList` interface defined in Program □.

```csharp
public class SortedListAsLinkedList :
    OrderedListAsLinkedList, SortedList
{
    // ...
}
```

**Program:** SortedListAsLinkedList class.

There are no additional fields defined in the `SortedListAsLinkedList` class. The inherited fields are sufficient to implement a sorted list. In fact, the functionality inherited from the `ListAsLinkedList` class is almost sufficient--the only method of which the functionality must change is the Insert operation.

- Inserting Items in a Sorted List
- Other Operations on Sorted Lists

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Inserting Items in a Sorted List

Program gives the implementation of the Insert method of the SortedListAsLinkedList class. This method takes a single argument: the object to be inserted into the sorted list. The algorithm used for the insertion is as follows: First, the existing sorted, linked list is traversed in order to find the linked list element which is greater than or equal to the object to be inserted into the list. The traversal is done using two variables--prevPtr and ptr. During the traversal, the latter keeps track of the current element and the former keeps track of the previous element.

By keeping track of the previous element, it is possible to efficiently insert the new item into the sorted list by calling the InsertAfter method of the LinkedList class. In Chapter, the InsertAfter method was shown to be $O(1)$.

In the event that the item to be inserted is smaller than the first item in the sorted list, then rather than using the InsertAfter method, the Prepend method is used. The Prepend method was also shown to be $O(1)$.

In the worst case, the object to be inserted into the linked list is larger than all of the objects already present in the list. In this case, the entire list needs to be traversed before doing the insertion. Consequently, the total running time for the Insert operation of the SortedListAsLinkedList class is $O(n)$, where $n = \text{count}$. 

public class SortedListAsLinkedList :
    OrderedListAsLinkedList, SortedList
{
    public override void Insert(ComparableObject arg)
    {
        LinkedList.Element ptr;
        LinkedList.Element prevPtr = null;

        for (ptr = linkedList.Head;
            ptr != null; ptr = ptr.Next)
        {
            ComparableObject obj = (ComparableObject)ptr.Datum;
            if (obj >= arg)
                break;
            prevPtr = ptr;
        }
        if (prevPtr == null)
            linkedList.Prepend(arg);
        else
            prevPtr.InsertAfter(arg);
        ++count;
    }
    // ...
}

Program: SortedListAsLinkedList class Insert method.
Unfortunately, it is not possible to do a binary search in a linked list. As a result, it is not possible to exploit the sortedness of the list in the implementation of any of the other required operations on sorted lists. The methods inherited from the `OrderedListAsLinkedList` provide all of the needed functionality.
Performance Comparison: SortedListAsArray vs. SortedListAsList

The running times calculated for the various methods of the two sorted list implementations, SortedListAsArray and SortedListAsLinkedList, are summarized below in Table 1. With the exception of two methods, the running times of the two implementations are asymptotically identical.

<table>
<thead>
<tr>
<th>method</th>
<th>SortedList-AsArray</th>
<th>SortedList-AsLinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>IsMember</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Find</td>
<td>$O(\log n)$</td>
<td>$\neq$ $O(n)$</td>
</tr>
<tr>
<td>Withdraw</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>this[int]{get}</td>
<td>$O(1)$</td>
<td>$\neq$ $O(n)$</td>
</tr>
<tr>
<td>FindPosition</td>
<td>$O(\log n)$</td>
<td>$\neq$ $O(n)$</td>
</tr>
<tr>
<td>Cursor.Datum{get}</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Cursor.Withdraw</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Table: Running times of operations on sorted lists.

Neither the SortedListAsArray nor SortedListAsLinkedList implementations required any additional fields beyond those inherited from their respective base classes, OrderedListAsArray and OrderedListAsLinkedList. Consequently, the space requirements analysis of the sorted list implementations is identical to that of the ordered list implementations given in Section 1.
Applications

In Section we saw that an \(n^{th}\)-order polynomial,

\[
\sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n
\]

where \(a_n \neq 0\), can be represented by a sequence of ordered pairs thus:

\[
\{(a_0, 0), (a_1, 1), (a_2, 2), \ldots, (a_n, n)\}.
\]

We also saw that it is possible to make use of an ordered list to represent such a sequence and that given such a representation, we can write an algorithm to perform differentiation.

As it turns out, the order of the terms in the sequence does not affect the differentiation algorithm. The correct result is always obtained and the running time is unaffected regardless of the order of the terms in the sequence.

Unfortunately, there are operations on polynomials whose running time depends on the order of the terms. For example, consider the addition of two polynomials:

\[
(a_0 + a_1 x + a_2 x^2) + (b_3 x^3 + b_2 x^2 + b_1 x) =
\]

\[
(a_0) + (a_1 + b_1)x + (a_2 + b_2)x^2 + (b_3)x^3
\]

To perform the addition all the terms involving \(x\) raised to the same power need to be grouped together.

If the terms of the polynomials are in an arbitrary order, then the grouping together of the corresponding terms is time consuming. On the other hand, if the terms are ordered, say, from smallest exponent to largest, then the summation can be done rather more efficiently. A single pass through the polynomials will suffice. It makes sense to represent each of the polynomials as a sorted list of terms using, say, the SortedListAsLinkedList class.
Applications

- Implementation
- Analysis

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Implementation

To begin with, we need to represent the terms of the polynomial. Program extends the definition of the Term class introduced in Program --some additions are needed to support the the implementation of polynomial addition.

```csharp
public class Term : ComparableObject
{
    protected double coefficient;
    protected int exponent;

    public Term(Term term) :
        this(term.coefficient, term.exponent)
    {
    }

    public double Coefficient
    {
        get { return coefficient; }
    }

    public int Exponent
    {
        get { return exponent; }
    }

    public static Term operator +(Term t1, Term t2)
    {
        if (t1.exponent != t2.exponent)
            throw new ArgumentException("unequal exponents");
        return new Term(
            t1.coefficient + t2.coefficient, t1.exponent);
    }

    // ...
}
```

Program: Term methods.
Four additional operations are declared in Program 1. The first is a constructor which creates a copy of a given term. The next two properties, Coefficient and Exponent, provide get accessors that return the corresponding fields of a Term instance. Clearly, the running time of each of these operations is $O(1)$.

The final method, operator+, provides the means to add two Terms together. The result of the addition is another Term. The working assumption is that the terms to be added have identical exponents. If the exponents are allowed to differ, the result of the addition is a polynomial which cannot be represented using a single term! To add terms with like exponents, we simply need to add their respective coefficients. Therefore, the running time of the Term addition operator is $O(1)$.

We now turn to the polynomial itself. Program 1 introduces the PolynomialAsSortedList class. This class implements the Polynomial interface defined in Program 1. It has a single field of type SortedList. We have chosen in this implementation to use the linked-list sorted list implementation to represent the sequence of terms.

```java
public class PolynomialAsSortedList : Polynomial
{
    SortedList list;

    public override Polynomial Plus(Polynomial poly)
    {
        PolynomialAsSortedList arg =
            (PolynomialAsSortedList)poly;
        Polynomial result = new PolynomialAsSortedList();
        IEnumerator p1 = list.GetEnumerator();
        IEnumerator p2 = arg.list.GetEnumerator();
        Term term1 = NextTerm(p1);
        Term term2 = NextTerm(p2);
        while (term1 != null && term2 != null)
        {
            if (term1.Exponent < term2.Exponent) {
                result.Add(new Term(term1));
                term1 = NextTerm(p1);
            }
            else if (term1.Exponent > term2.Exponent) {
                result.Add(new Term(term2));
                term2 = NextTerm(p2);
            }
            else {
```

http://www.brpreiss.com/books/opus6/html/page204.html (2 of 4) [2002-11-17 ソタネト 11:04:00]
Program: PolynomialAsSortedList class Plus method.

Program defines the Plus method. This method adds two Polynomials to obtain a third. It is intended to be used like this:

Polynomial p1 = new PolynomialAsSortedList();
Polynomial p2 = new PolynomialAsSortedList();
// ...
Polynomial p3 = p1 + p2;
Data Structures and Algorithms with Object-Oriented Design Patterns in C#

**Analysis**

The proof of the correctness of Program □ is left as an exercise for the reader (Exercise □). We discuss here the running time analysis of the algorithm, as there are some subtle points to remember which lead to a result that may be surprising.

Consider the addition of a polynomial \( p(x) \) with its arithmetic complement \(-p(x)\). Suppose \( p(x) \) has \( n \) terms. Clearly \(-p(x)\) also has \( n \) terms. The sum of the polynomials is the zero polynomial. An important characteristic of the zero polynomial is that it *has no terms*! In this case, exactly \( n \) iterations of the main loop are done (lines 14-31). Furthermore, zero iterations of the second and the third loops are required (lines 32-35 and 36-39). Since the result has no terms, there will be no calls to the `Add` method. Therefore, the amount of work done in each iteration is a constant. Consequently, the best case running time is \( O(n) \).

Consider now the addition of two polynomials, \( p(x) \) and \( q(x) \), having \( l \) and \( m \) terms, respectively. Furthermore, suppose that largest exponent in \( p(x) \) is less than the smallest exponent in \( q(x) \). Consequently, there is no power of \( x \) which the two polynomials have in common. In this case, since \( p(x) \) has the lower-order terms, exactly \( l \) iterations of the main loop (lines 14-31) are done. In each of these iterations, exactly one new term is inserted into the result by calling the `Add` method. Since all of the terms of \( p(x) \) will be exhausted when the main loop is finished, there will be no iterations of the second loop (lines 32-35). However, there will be exactly \( m \) iterations of the third loop (lines 36-39) in each of which one new term is inserted into the result by calling the `Add` method.

Altogether, \( l+m \) calls to the `Add` will be made. It was shown earlier that the running time for the insert method is \( O(k) \), where \( k \) is the number of items in the sorted list. Consequently, the total running time for the \( l+m \) insertions is

\[
\sum_{k=0}^{l+m-1} O(k) = O((l + m)^2).
\]

Consequently, the worst case running time for the polynomial addition given in Program □ is \( O(n^2) \), where \( n=l+m \). This is somewhat disappointing. The implementation is not optimal because it fails to take account of the order in which the terms of the result are computed. That is, the `Add` method repeatedly searches the sorted list for the correct position at which to insert the next term. But we know that correct position is at the end! By replacing in Program □ all of the calls to the `Add` method by
((result as PolynomialAsSortedList).list as SortedListAsLinkedList).linkedList.Append (...);

the total running time can be reduced to $O(n)$ from $O(n^2)$!
Exercises

1. Devise an algorithm to reverse the contents of an ordered list. Determine the running time of your algorithm.
2. Devise an algorithm to append the contents of one ordered list to the end of another. Assume that both lists are represented using arrays. What is the running time of your algorithm?
3. Repeat Exercise 1, but this time assume that both lists are represented using linked lists. What is the running time of your algorithm?
4. Devise an algorithm to merge the contents of two sorted lists. Assume that both lists are represented using arrays. What is the running time of your algorithm?
5. Repeat Exercise 4, but this time assume that both lists are represented using linked lists. What is the running time of your algorithm?
6. The Withdraw method can be used to remove items from a list one at a time. Suppose we want to provide an additional a method, WithdrawAll, that takes one argument and withdraws all the items in a list that match the given argument. We can provide an implementation of the WithdrawAll method in the AbstractSearchableContainer class like this:

   ```csharp
   public class AbstractSearchableContainer :
       AbstractContainer, SearchableContainer
   {
       void WithdrawAll(ComparableObject arg)
       {
           ComparableObject obj;
           while ((obj = Find(arg)) != null)
               Withdraw(obj);
       }
   // ...
   }
   
   ```

   Determine the worst-case running time of this method for each of the following cases:
   1. an array-based implementation of an ordered list,
   2. a linked-list implementation of an ordered list,
   3. an array-based implementation of a sorted list, and
   4. a linked-list implementation of a sorted list.
7. Devise an $O(n)$ algorithm, to remove from an ordered list all the items that match a given item. Assume the list is represented using an array.
8. Repeat Exercise 8, but this time assume the ordered list is represented using a linked list.
9. Consider an implementation of the `OrderedList` interface that uses a doubly-linked list such as the one shown in Figure 8(a). Compare the running times of the operations for this implementation with those given in Table 8.
10. Derive an expression for the amount of space used to represent an ordered list of \( n \) elements using a doubly-linked list such as the one shown in Figure 8(a). Compare this with the space used by the array-based implementation. Assume that integers and pointers each occupy four bytes.
11. Consider an implementation of the `SortedList` interface that uses a doubly-linked list such as the one shown in Figure 8(a). Compare the running times of the operations for this implementation with those given in Table 8.
12. Verify that Program 8 correctly computes the sum of two polynomials.
13. Write an algorithm to multiply a polynomial by a scalar. **Hint:** Use a visitor.
Projects

1. Write a visitor to solve each of the following problems:
   1. Find the smallest element of a list.
   2. Find the largest element of a list.
   3. Compute the sum of all the elements of a list.
   4. Compute the product of all the elements of a list.

2. Design and implement a class called `OrderedListAsDoublyLinkedList` which represents an ordered list using a doubly-linked list. Select one of the approaches shown in Figure.

3. Consider the `Polynomial` class given in Program. Implement a method that computes the value of a polynomial, say \( p(x) \), for a given value of \( x \). **Hint:** Use a visitor that visits all the terms in the polynomial and accumulates the result.

4. Devise and implement an algorithm to multiply two polynomials. **Hint:** Consider the identity

\[
\left( \sum_{i=0}^{n} a_i x^i \right) \times \left( \sum_{j=0}^{m} b_j x^j \right) = \sum_{i=0}^{n} \sum_{j=0}^{m} a_i b_j x^{i+j}.
\]

Write a method to multiply a `Polynomial` by a `Term` and use the polynomial addition operator defined in Program.

5. Devise and implement an algorithm to compute the \( k^{th} \) power of a polynomial, where \( k \) is a positive integer. What is the running time of your algorithm?

6. For some calculations it is necessary to have very large integers, i.e., integers with an arbitrarily large number of digits. We can represent such integers using lists. Design and implement a class for representing arbitrarily large integers. Your implementation should include operations to add, subtract, and multiply such integers, and to compute the \( k^{th} \) power of such an integer, where \( k \) is a small positive integer. **Hint:** Base your design on the `Polynomial` class given in Program.
Hashing, Hash Tables, and Scatter Tables

A very common paradigm in data processing involves storing information in a table and then later retrieving the information stored there. For example, consider a database of driver's license records. The database contains one record for each driver's license issued. Given a driver's license number, we can look up the information associated with that number.

Similar operations are done by the C# compiler. The compiler uses a symbol table to keep track of the user-defined symbols in a C# program. As it compiles a program, the compiler inserts an entry in the symbol table every time a new symbol is declared. In addition, every time a symbol is used, the compiler looks up the attributes associated with that symbol to see that it is being used correctly and to guide the generation of the MSIL code.

Typically the database comprises a collection of key-and-value pairs. Information is retrieved from the database by searching for a given key. In the case of the driver's license database, the key is the driver's license number and in the case of the symbol table, the key is the name of the symbol.

In general, an application may perform a large number of insertion and/or look-up operations. Occasionally it is also necessary to remove items from the database. Because a large number of operations will be done we want to do them as quickly as possible.

- Hashing-The Basic Idea
- Hashing Methods
- Hash Function Implementations
- Hash Tables
- Scatter Tables
- Scatter Table using Open Addressing
Hashing-The Basic Idea

In this chapter we examine data structures which are designed specifically with the objective of providing efficient insertion and find operations. In order to meet the design objective, certain concessions are made. Specifically, we do not require that there be any specific ordering of the items in the container. In addition, while we still require the ability to remove items from the container, it is not our primary objective to make removal as efficient as the insertion and find operations.

Ideally we would build a data structure for which both the insertion and find operations are \( O(1) \) in the worst case. However, this kind of performance can only be achieved with complete \textit{a priori} knowledge. We need to know beforehand specifically which items are to be inserted into the container. Unfortunately, we do not have this information in the general case. So, if we cannot guarantee \( O(1) \) performance in the \textit{worst case}, then we make it our design objective to achieve \( O(1) \) performance \textit{in the average case}.

The constant time performance objective immediately leads us to the following conclusion: Our implementation must be based in some way on an array rather than a linked list. This is because we can access the \( k^{th} \) element of an array in constant time, whereas the same operation in a linked list takes \( O(k) \) time.

In the previous chapter, we consider two searchable containers--the \textit{ordered list} and the \textit{sorted list}. In the case of an ordered list, the cost of an insertion is \( O(1) \) and the cost of the find operation is \( O(n) \). For a sorted list the cost of insertion is \( O(n) \) and the cost of the find operation is \( O(\log n) \) for the array implementation.

Clearly, neither the ordered list nor the sorted list meets our performance objectives. The essential problem is that a search, either linear or binary, is always necessary. In the ordered list, the find operation uses a linear search to locate the item. In the sorted list, a binary search can be used to locate the item because the data is sorted. However, in order to keep the data sorted, insertion becomes \( O(n) \).

In order to meet the performance objective of constant time insert and find operations, we need a way to do them \textit{without performing a search}. That is, given an item \( x \), we need to be able to determine directly from \( x \) the array position where it is to be stored.
Hashing-The Basic Idea

- Example
- Keys and Hash Functions

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Example

We wish to implement a searchable container which will be used to contain character strings from the set of strings \( K \),

\[
K = \{ "ett", "två", "tre", "fyra", "fem", "sex", "sju", "åtta", "nio", "tio", "elva", "tolv" \}.
\]

Suppose we define a function \( h : K \rightarrow \mathbb{Z} \) as given by the following table:

<table>
<thead>
<tr>
<th>( x )</th>
<th>( h(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ett&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;två&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;tre&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;fyra&quot;</td>
<td>4</td>
</tr>
<tr>
<td>&quot;fem&quot;</td>
<td>5</td>
</tr>
<tr>
<td>&quot;sex&quot;</td>
<td>6</td>
</tr>
<tr>
<td>&quot;sju&quot;</td>
<td>7</td>
</tr>
<tr>
<td>&quot;åtta&quot;</td>
<td>8</td>
</tr>
<tr>
<td>&quot;nio&quot;</td>
<td>9</td>
</tr>
<tr>
<td>&quot;tio&quot;</td>
<td>10</td>
</tr>
<tr>
<td>&quot;elva&quot;</td>
<td>11</td>
</tr>
<tr>
<td>&quot;tolv&quot;</td>
<td>12</td>
</tr>
</tbody>
</table>

Then, we can implement a searchable container using an array of length \( n=12 \). To insert item \( x \), we simply store it at position \( h(x)-1 \) of the array. Similarly, to locate item \( x \), we simply check to see if it is found at position \( h(x)-1 \). If the function \( h(\cdot) \) can be evaluated in constant time, then the both the insert and the find operations are \( O(1) \).

We expect that any reasonable implementation of the function \( h(\cdot) \) will run in constant time, since the size of the set of strings, \( K \), is a constant! This example illustrates how we can achieve \( O(1) \) performance...
in the worst case when we have complete, *a priori* knowledge.
Keys and Hash Functions

We are designing a container which will be used to hold some number of items of a given set $K$. In this context, we call the elements of the set $K$ keys. The general approach is to store the keys in an array. The position of a key in the array is given by a function $h(\cdot)$, called a hash function, which determines the position of a given key directly from that key.

In the general case, we expect the size of the set of keys, $|K|$, to be relatively large or even unbounded. For example, if the keys are 32-bit integers, then $|K| = 2^{32}$. Similarly, if the keys are arbitrary character strings of arbitrary length, then $|K|$ is unbounded.

On the other hand, we also expect that the actual number of items stored in the container to be significantly less than $|K|$. That is, if $n$ is the number of items actually stored in the container, then $n \ll |K|$. Therefore, it seems prudent to use an array of size $M$, where $M$ is as least as great as the maximum number of items to be stored in the container.

Consequently, what we need is a function $h : K \mapsto \{0, 1, \ldots, M-1\}$. This function maps the set of values to be stored in the container to subscripts in an array of length $M$. This function is called a hash function.

In general, since $|K| \geq M$, the mapping defined by hash function will be a many-to-one mapping. That is, there will exist many pairs of distinct keys $x$ and $y$, such that $x \neq y$, for which $h(x)=h(y)$. This situation is called a collision. Several approaches for dealing with collisions are explored in the following sections.

What are the characteristics of a good hash function?

- A good hash function avoids collisions.
- A good hash function tends to spread keys evenly in the array.
- A good hash function is easy to compute.
Keys and Hash Functions

- **Avoiding Collisions**
- **Spreading Keys Evenly**
- **Ease of Computation**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Avoiding Collisions

Ideally, given a set of \( n \leq M \) distinct keys, \( \{ k_1, k_2, \ldots, k_n \} \), the set of hash values \( \{ h(k_1), h(k_2), \ldots, h(k_n) \} \) contains no duplicates. In practice, unless we know something about the keys chosen, we cannot guarantee that there will not be collisions. However, in certain applications we have some specific knowledge about the keys that we can exploit to reduce the likelihood of a collision. For example, if the keys in our application are telephone numbers, and we know that the telephone numbers are all likely to be from the same geographic area, then it makes little sense to consider the area codes in the hash function, the area codes are all likely to be the same.
Spreading Keys Evenly

Let $p_i$ be the probability that the hash function $h(\cdot) = i$. A hash function which spreads keys evenly has the property that for $0 \leq i < M$, $p_i = 1/M$. In other words, the hash values computed by the function $h(\cdot)$ are uniformly distributed. Unfortunately, in order to say something about the distribution of the hash values, we need to know something about the distribution of the keys.

In the absence of any information to the contrary, we assume that the keys are equiprobable. Let $K_i$ be the set of keys that map to the value $i$. That is, $K_i = \{k \in K : h(k) = i\}$. If this is the case, the requirement to spread the keys uniformly implies that $|K_i| = |K|/M$. An equal number of keys should map into each array position.
Ease of Computation

This does not mean necessarily that it is easy for someone to compute the hash function, nor does it mean that it is easy to write the algorithm to compute the function; it means that the running time of the hash function should be $O(1)$. 
Hashing Methods

In this section we discuss several hashing methods. In the following discussion, we assume that we are dealing with integer-valued keys, i.e., $\mathbb{K} = \mathbb{Z}$. Furthermore, we assume that the value of the hash function falls between 0 and $M-1$.

- Division Method
- Middle Square Method
- Multiplication Method
- Fibonacci Hashing
Division Method

Perhaps the simplest of all the methods of hashing an integer $x$ is to divide $x$ by $M$ and then to use the remainder modulo $M$. This is called the division method of hashing. In this case, the hash function is

$$h(x) = |x| \mod M.$$ 

Generally, this approach is quite good for just about any value of $M$. However, in certain situations some extra care is needed in the selection of a suitable value for $M$. For example, it is often convenient to make $M$ an even number. But this means that $h(x)$ is even if $x$ is even; and $h(x)$ is odd if $x$ is odd. If all possible keys are equiprobable, then this is not a problem. However if, say, even keys are more likely than odd keys, the function $h(x) = x \mod M$ will not spread the hashed values of those keys evenly.

Similarly, it is often tempting to let $M$ be a power of two. For example, $M = 2^k$ for some integer $k > 1$. In this case, the hash function $h(x) = x \mod 2^k$ simply extracts the bottom $k$ bits of the binary representation of $x$. While this hash function is quite easy to compute, it is not a desirable function because it does not depend on all the bits in the binary representation of $x$.

For these reasons $M$ is often chosen to be a prime number. For example, suppose there is a bias in the way the keys are created that makes it more likely for a key to be a multiple of some small constant, say two or three. Then making $M$ a prime increases the likelihood that those keys are spread out evenly. Also, if $M$ is a prime number, the division of $x$ by that prime number depends on all the bits of $x$, not just the bottom $k$ bits, for some small constant $k$.

The division method is extremely simple to implement. The following C# code illustrates how to do it:

```csharp
public class DivisionMethod
{
    private const int M = 1031; // a prime
    
    public static int H(int x)
    {
        return Math.Abs(x) % M;
    }
}
```
Division Method

In this case, M is a constant. However, an advantage of the division method is that M need not be a compile-time constant--its value can be determined at run time. In any event, the running time of this implementation is clearly a constant.

A potential disadvantage of the division method is due to the property that consecutive keys map to consecutive hash values:

\[
    h(i) \quad = \quad i \\
    h(i + 1) \quad = \quad i + 1 \pmod{M} \\
    h(i + 2) \quad = \quad i + 2 \pmod{M} \\
                   \vdots
\]

While this ensures that consecutive keys do not collide, it does mean that consecutive array locations will be occupied. We will see that in certain implementations this can lead to degradation in performance. In the following sections we consider hashing methods that tend to scatter consecutive keys.
Middle Square Method

In this section we consider a hashing method which avoids the use of division. Since integer division is usually slower than integer multiplication, by avoiding division we can potentially improve the running time of the hashing algorithm. We can avoid division by making use of the fact that a computer does finite-precision integer arithmetic. For example, all arithmetic is done modulo \( W \) where \( W = 2^w \) is a power of two such that \( w \) is the word size of the computer.

The middle-square hashing method works as follows. First, we assume that \( M \) is a power of two, say \( M = 2^k \) for some \( k \geq 1 \). Then, to hash an integer \( x \), we use the following hash function:

\[
\hat{h}(x) = \left\lfloor \frac{M}{W} (x^2 \mod W) \right\rfloor.
\]

Notice that since \( M \) and \( W \) are both powers of two, the ratio \( W/M = 2^{w-k} \) is also a power two. Therefore, in order to multiply the term \((x^2 \mod W)\) by \( M/W \) we simply shift it to the right by \( w-k \) bits!

In effect, we are extracting \( k \) bits from the middle of the square of the key--hence the name of the method.

The following code fragment illustrates the middle-square method of hashing:

```csharp
public class MiddleSquareMethod
{
    private const int k = 10; // M==1024
    private const int w = 32;

    public static int H(int x)
    {
        return (int)((uint)(x * x)) >> (w - k);
    }
}
```

Since \( x \) is an int, the product \( x \times x \) is also an an int. In C#, an int represents a 32-bit quantity and the product of two ints is also a 32-bit quantity. The final result is obtained by shifting the product \( w-k \) bits to the right, where \( w \) is the number of bits in an integer. Note, we cast the product to a uint before the shift so as to cause the right-shift operator to insert zeroes on the left. Therefore, the result always falls between 0 and \( M-1 \).
The middle-square method does a pretty good job when the integer-valued keys are equiprobable. The middle-square method also has the characteristic that it scatters consecutive keys nicely. However, since the middle-square method only considers a subset of the bits in the middle of \( x^2 \), keys which have a large number of leading zeroes will collide. For example, consider the following set of keys:

\[
\{ x \in \mathbb{Z} : |x| < \sqrt{W/M} \}.
\]

This set contains all keys \( x \) such that \(|x| < 2^{(w-k)/2}\). For all of these keys \( h(x) = 0 \).

A similar line of reasoning applies for keys which have a large number of trailing zeroes. Let \( W \) be an even power of two. Consider the set of keys

\[
\{ x \in \mathbb{Z} : x = \pm n\sqrt{W}, \ n \in \mathbb{Z} \}.
\]

The least significant \( w/2 \) bits of the keys in this set are all zero. Therefore, the least significant \( w \) bits of \( x^2 \) are also zero and as a result \( h(x) = 0 \) for all such keys!
Multiplication Method

A very simple variation on the middle-square method that alleviates its deficiencies is the so-called, multiplication hashing method. Instead of multiplying the key $x$ by itself, we multiply the key by a carefully chosen constant $a$, and then extract the middle $k$ bits from the result. In this case, the hashing function is

$$h(x) = \left\lfloor \frac{M}{W} (ax \mod W) \right\rfloor.$$

What is a suitable choice for the constant $a$? If we want to avoid the problems that the middle-square method encounters with keys having a large number of leading or trailing zeroes, then we should choose an $a$ that has neither leading nor trailing zeroes.

Furthermore, if we choose an $a$ that is relatively prime to $W$, then there exists another number $a'$ such that $aa' = 1 \pmod{W}$. In other words, $a'$ is the inverse of $a$ modulo $W$, since the product of $a$ and its inverse is one. Such a number has the nice property that if we take a key $x$, and multiply it by $a$ to get $ax$, we can recover the original key by multiplying the product again by $a'$, since $axa' = aa'x = 1x$.

There are many possible constants which the desired properties. One possibility which is suited for 32-bit arithmetic (i.e., $W = 2^{32}$) is $a = 2654435769$. The binary representation of $a$ is

$$100111100011011011110010110111001.$$

This number has neither many leading nor trailing zeroes. Also, this value of $a$ and $W = 2^{32}$ are relatively prime and the inverse of $a$ modulo $W$ is $a' = 340573321$.

The following code fragment illustrates the multiplication method of hashing:

```csharp
public class MultiplicationMethod
{
    private const int k = 10; // M==1024
    private const int w = 32;
    private const uint a = 2654435769U;
```
The code is a simple modification of the middle-square version. Nevertheless, the running time remains $O(1)$. 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Fibonacci Hashing

The final variation of hashing to be considered here is called the Fibonacci hashing method. In fact, Fibonacci hashing is exactly the multiplication hashing method discussed in the preceding section using a very special value for \( a \). The value we choose is closely related to the number called the golden ratio.

The golden ratio is defined as follows: Given two positive numbers \( x \) and \( y \), the ratio \( \phi = \frac{x}{y} \) is the golden ratio if the ratio of \( x \) to \( y \) is the same as that of \( x+y \) to \( x \). The value of the golden ratio can be determined as follows:

\[
\frac{x}{y} = \frac{x+y}{x} \Rightarrow 0 = x^2 - xy - y^2 \\
\Rightarrow 0 = \phi^2 - \phi - 1 \\
\Rightarrow \phi = \frac{1 + \sqrt{5}}{2}
\]

There is an intimate relationship between the golden ratio and the Fibonacci numbers. In Section it was shown that the \( n \)th Fibonacci number is given by

\[
F_n = \frac{1}{\sqrt{5}} \left( \phi^n - \hat{\phi}^n \right),
\]

where \( \phi = (1 + \sqrt{5})/2 \) and \( \hat{\phi} = (1 - \sqrt{5})/2 \).

The Fibonacci hashing method is essentially the multiplication hashing method in which the constant \( a \) is chosen as the integer that is relatively prime to \( W \) which is closest to \( W/\phi \). The following table gives suitable values of \( a \) for various word sizes.

<table>
<thead>
<tr>
<th>( W )</th>
<th>( a \approx \frac{W}{\phi} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^{16} )</td>
<td>40503</td>
</tr>
<tr>
<td>( 2^{32} )</td>
<td>2654435769</td>
</tr>
</tbody>
</table>
Why is $W/\phi$ special? It has to do with what happens to consecutive keys when they are hashed using the multiplicative method. As shown in Figure, consecutive keys are spread out quite nicely. In fact, when we use $a \approx W/\phi$ to hash consecutive keys, the hash value for each subsequent key falls in between the two widest spaced hash values already computed. Furthermore, it is a property of the golden ratio, $\phi$, that each subsequent hash value divides the interval into which it falls according to the golden ratio!

Figure: Fibonacci hashing.
Hash Function Implementations

The preceding section presents methods of hashing integer-valued keys. In reality, we cannot expect that the keys will always be integers. Depending on the application, the keys might be letters, character strings or even more complex data structures such as Associations or Containers.

In general given a set of keys, $K$, and a positive constant, $M$, a hash function is a function of the form

$$h : K \mapsto \{0, 1, \ldots, M - 1\}.$$

In practice is it convenient to implement the hash function $h$ as the composition of two functions $f$ and $g$. The function $f$ maps keys into integers:

$$f : K \mapsto \mathbb{Z},$$

where $\mathbb{Z}$ is the set of integers. The function $g$ maps non-negative integers into $\{0, 1, \ldots, M - 1\}$:

$$g : \mathbb{Z} \mapsto \{0, 1, \ldots, M - 1\}.$$

Given appropriate functions $f$ and $g$, the hash function $h$ is simply defined as the composition of those functions:

$$h = g \circ f$$

That is, the hash value of a key $x$ is given by $g(f(x))$.

By decomposing the function $h$ in this way, we can separate the problem into two parts: The first involves finding a suitable mapping from the set of keys $K$ to the non-negative integers. The second involves mapping non-negative integers into the interval $[0, M-1]$. Ideally, the two problems would be unrelated. That is, the choice of the function $f$ would not depend on the choice of $g$ and vice versa. Unfortunately, this is not always the case. However, if we are careful, we can design the functions in such a way that $h = g \circ f$ is a good hash function.
The hashing methods discussed in the preceding section deal with integer-valued keys. But this is precisely the domain of the function $g$. Consequently, we have already examined several different alternatives for the function $g$. On the other hand, the choice of a suitable function for $f$ depends on the characteristics of its domain.

In the following sections, we consider various different domains (sets of keys) and develop suitable hash functions for each of them. Each domain considered corresponds to a C# class. Recall that every C# class is ultimately derived from the `System.Object` class and that the `System.Object` class declares a method called `GetHashCode`:

```csharp
namespace System
{
    public class object
    {
        public virtual int GetHashCode () { /* ... */ }
        // ...
    }
}
```

The `GetHashCode` method corresponds to the function $f$ which maps keys into integers.
Integral Keys

Of all the C# types, the so-called integral types are the simplest to hash into integers. The integral data types are byte, sbyte, short, ushort, int, uint, long, ulong, and char. Since the underlying representation of such data types can be viewed as an integer, the hash function is trivial. A suitable function $f$ for an integral data type is the identity function:

$$f(x) = x.$$ 

Program □ completes the definition of the ComparableInt32 wrapper class introduced in Program □. In this case, the GetHashCode method simply returns the value of the boxed Int32 object. Clearly, the running time of the GetHashCode method is $O(1)$.

```csharp
public class ComparableInt32 : ComparableValue
{
    public override int GetHashCode()
    {
        return (int)obj;
    }

    // ...
}
```

Program: ComparableInt32 class GetHashCode method.
Floating-Point Keys

Dealing with floating-point number involves only a little more work. In C# the floating-point data types are `float` and `double`. The size of a `float` is 32 bits and the size of a `double` is 64 bits.

We seek a function $f$ which maps a floating-point value into a non-negative integer. One possibility is to simply reinterpret the bit pattern used to represent the floating point number as an integer. However, this is only possible when the size of the floating-point type does not exceed the size of `int`. This condition is satisfied only by the `float` type.

Another characteristic of floating-point numbers that must be dealt with is the extremely wide range of values which can be represented. For example, when using IEEE floating-point, the smallest double precision quantity that can be represented is $2^{-1074}$ and the largest is $\approx 1.8 \times 10^{308}$. Somehow we need to map values in this large domain into the range of an `int`.

Every non-zero floating-point quantity $x$ can be written uniquely as

$$x = (-1)^s \times m \times 2^e,$$

where $s \in \{0, 1\}$, $0.5 \leq m < 1$ and $-1023 \leq e \leq 1024$. The quantity $s$ is called the sign, $m$ is called the mantissa or significant and $e$ is called the exponent. This suggests the following definition for the function $f$:

$$f(x) = \begin{cases} 0 & x = 0, \\ \lfloor 2(m - \frac{1}{2})W \rfloor & x \neq 0, \end{cases} \quad (8.1)$$

where $W = 2^w$ such that $w$ is the word size of the machine.

This hashing method is best understood by considering the conditions under which a collision occurs between two distinct floating-point numbers $x$ and $y$. Let $m_x$ and $m_y$ be the mantissas of $x$ and $y$, respectively. The collision occurs when $f(x)=f(y)$. 
Floating-Point Keys

Thus, \( x \) and \( y \) collide if their mantissas differ by less than \( \frac{1}{2}W \). Notice that the sign of the number is not considered. Thus, \( x \) and \(-x\) collide. Also, the exponent is not considered. Therefore, if \( x \) and \( y \) collide, then so too do \( x \) and \( y \times 2^k \) for all permissible values of \( k \).

Program completes the definition of the ComparableDouble wrapper class introduced in Program . The GetHashCode function shown computes the hash function defined in Equation .

```java
public class ComparableDouble : ComparableValue
{
    public override int GetHashCode()
    {
        long bits = BitConverter.DoubleToInt64Bits((double)obj);
        return (int)((ulong)bits >> 20);
    }
    // ...
}
```

Program: ComparableDouble class GetHashCode method.

This implementation makes use of the fact that in the IEEE standard floating-point format the least-significant 52 bits of a 64-bit floating-point number represent the quantity \( m' = (m - \frac{1}{2}) \times 2^{63} \). Since an int is a 32-bit quantity, \( W = 2^{32} \), and we can rewrite Equation as follows:

\[
    f(n) = 2(m - \frac{1}{2})W = m'2^{33}/2^{63} = m'/2^{20}.
\]

Thus, we can compute the hash function simply by shifting the binary representation of the floating-point number 20 bits to the right as shown in Program . Clearly the running time of the GetHashCode method is \( O(1) \).
Strings of characters are represented in C# as instances of the `String` class. A character string is simply a sequence of characters. Since such a sequence may be arbitrarily long, to devise a suitable hash function we must find a mapping from an unbounded domain into the finite range of `int`.

We can view a character string, \( s \), as a sequence of \( n \) characters,

\[
\{ s_0, s_1, \ldots, s_{n-1} \},
\]

where \( n \) is the length of the string. (The length of a string can be determined using the `String` property `Length`). One very simple way to hash such a string would be to simply sum the numeric values associated with each character:

\[
f(s) = \sum_{i=0}^{n-1} s_i
\]  

(8.2)

As it turns out, this is not a particularly good way to hash character strings. Given that a C# `char` is a 16-bit quantity, \( 0 \leq s_i \leq 2^{16} - 1 \), for all \( 0 \leq i < n \). As a result, \( 0 \leq f(s) < n(2^{16} - 1) \). For example, given a string of length \( n=5 \), the value of \( f(s) \) falls between zero and \( 327675 \). In fact, the situation is even worse, in North America we typically use only the ASCII subset of the Unicode character set. The ASCII character set uses only the least-significant seven bits of a `char`. If the string is comprised of only ASCII characters, the result falls in the range between zero and 640.

Essentially the problem with a function \( f \) which produces a result in a relatively small interval is the situation which arises when that function is composed with the function \( g(x) = x \mod M \). If the size of the range of the function \( f \) is less than \( M \), then \( h = g \circ f \) does not spread its values uniformly on the interval \([0, M-1] \). For example, if \( M=1031 \) only the first 640 values (62% of the range) are used!

Alternatively, suppose we have `a priori` knowledge that character strings are limited to length \( n=4 \). Then, we can construct an integer by concatenating the binary representations of each of the characters. For example, given \( s = \{ s_0, s_1, s_2, s_3 \} \), we can construct an integer with the function
where $B = 2^7$. Since $B$ is a power of two, this function is easy to write in C#:

```csharp
static int F(String s)
{
    return (int)s[0] << 21 | (int)s[1] << 14
        | (int)s[2] << 7 | (int)s[3];
}
```

While this function certainly has a larger range, it still has a problems--it cannot deal strings of arbitrary length.

Equation (8.3) can be generalized to deal with strings of arbitrary length as follows:

$$f(s) = \sum_{i=0}^{n-1} \bar{s}_i B^{n-i-1}$$

This function produces a unique integer for every possible string. Unfortunately, the range of $f(s)$ is unbounded. A simple modification of this algorithm suffices to bound the range:

$$f(s) = \left( \sum_{i=0}^{n-1} \bar{s}_i B^{n-i-1} \right) \mod W,$$

where $W = 2^w$ such that $w$ is word size of the machine. Unfortunately, since $W$ and $B$ are both powers of two, the value computed by this hash function depends only on the last $W/B$ characters in the character string. For example, for $W = 2^{32}$ and $B = 2^7$, this result depends only on the last five characters in the string--all character strings having exactly the same last five characters collide!

Writing the code to compute Equation 8.4 is actually quite straightforward if we realize that $f(s)$ can be viewed as a polynomial in $B$, the coefficients of which are $\bar{s}_0, \bar{s}_1, ..., \bar{s}_n$. Therefore, we can use Horner's rule (see Section 8.1) to compute $f(s)$ as follows:

```csharp
static int F(string s)
{
    int result = 0;
    for (int i = 0; i < s.Length; ++i)
    {
        result = result * B + (int)s[i];
    }
    return result;
}
```
This implementation can be simplified even further if we make use of the fact that $B = 2^b$, where $b=7$. Since $B$ is a power of two, in order to multiply the variable `result` by $B$ all we need to do is to shift it left by $b$ bits. Furthermore, having just shifted `result` left by $b$ bits, we know that the least significant $b$ bits of the result are zero. And since each character has no more than $b=7$ bits, we can replace the addition operation with an exclusive or operation.

```c
static int F(String s)
{
    int result = 0;
    for (int i = 0; i < s.Length; ++i)
        result = result << b ^ (int)s[i];
    return result;
}
```

Of the 128 characters in the 7-bit ASCII character set, only 97 characters are printing characters including the space, tab, and newline characters (see Appendix □). The remaining characters are control characters which, depending on the application, rarely occur in strings. Furthermore, if we assume that letters and digits are the most common characters in strings, then only 62 of the 128 ASCII codes are used frequently. Notice, the letters (both upper and lower case) all fall between $0101_8$ and $0172_8$. All the information is in the least significant six bits. Similarly, the digits fall between $060_8$ and $071_8$—these differ in the least significant four bits. These observations suggest that using $B = 2^6$ should work well. That is, for $W = 2^{32}$, the hash value depends on the last five characters plus two bits of the sixth-last character.

We have developed a hashing scheme which works quite well given strings which differ in the trailing letters. For example, the strings "temp1", "temp2", and "temp3", all produce different hash values. However, in certain applications the strings differ in the leading letters. For example, the two Internet domain names "ece.uwaterloo.ca" and "cs.uwaterloo.ca" collide when using Equation □. Essentially, the effect of the characters that differ is lost because the corresponding bits have been shifted out of the hash value.
public class ComparableString : ComparableValue
{
    private const int shift = 6;
    private const int mask = 0 << (32 - shift);

    public override int GetHashCode()
    {
        int result = 0;
        string s = (string)obj;
        for (int i = 0; i < s.Length; ++i)
            result = (result & mask) ^ (result << shift) ^ s[i];
        return result;
    }
    // ...
}
<table>
<thead>
<tr>
<th>$x$</th>
<th>Hash($x$) (octal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ett&quot;</td>
<td>01446564</td>
</tr>
<tr>
<td>&quot;två&quot;</td>
<td>01656545</td>
</tr>
<tr>
<td>&quot;tre&quot;</td>
<td>01656345</td>
</tr>
<tr>
<td>&quot;fyra&quot;</td>
<td>0147706341</td>
</tr>
<tr>
<td>&quot;fem&quot;</td>
<td>01474455</td>
</tr>
<tr>
<td>&quot;sex&quot;</td>
<td>01624470</td>
</tr>
<tr>
<td>&quot;sju&quot;</td>
<td>01625365</td>
</tr>
<tr>
<td>&quot;åtta&quot;</td>
<td>0344656541</td>
</tr>
<tr>
<td>&quot;nio&quot;</td>
<td>01575057</td>
</tr>
<tr>
<td>&quot;tio&quot;</td>
<td>01655057</td>
</tr>
<tr>
<td>&quot;elva&quot;</td>
<td>044556741</td>
</tr>
<tr>
<td>&quot;tolv&quot;</td>
<td>065565566</td>
</tr>
</tbody>
</table>

**Table:** Sample character string keys and the hash values obtained using Program □.
Hashing Containers

As explained in Section □, a container is an object which contains other objects. The AbstractContainer class introduced in Program □ implements the Container interface defined in Program □. In this section we show how to define a GetHashCode method in the AbstractContainer class that computes a suitable hash function on any container.

Given a container \( c \) which contains \( n \) objects, \( o_1, o_2, \ldots, o_n \), we can define the hash function \( f(c) \) as follows:

\[
f(c) = \left( \sum_{i=1}^{n} h(o_i) \right) \mod W \tag{8.5}
\]

That is, to hash a container, simply compute the sum of the hash values of the contained objects.

Program □ gives the code for the GetHashCode method of the AbstractContainer class. This method makes use of the Accept method to cause a visitor to visit all of the objects contained in the container. When the visitor visits an object, it calls that object's GetHashCode method and accumulates the result.
public abstract class AbstractContainer :
    ComparableObject, Container
{
    protected int count;

    private class HashCodeVisitor : AbstractVisitor
    {
        private int result = 0;

        public override void Visit(object obj)
        {
            result += obj.GetHashCode();
        }

        public override int GetHashCode()
        {
            return result;
        }
    }

    public override int GetHashCode()
    {
        Visitor visitor = new HashCodeVisitor();
        Accept(visitor);
        return GetType().GetHashCode() + visitor.GetHashCode();
    }

    // ...
}

Program: AbstractContainer class GetHashCode method.

Since the Accept method is an abstract method, every concrete class derived from the AbstractContainer class must provide an appropriate implementation. However, it is not necessary for any derived class to redefine the behavior of the GetHashCode method—the behavior inherited from the AbstractContainer class is completely generic and should suffice for all concrete container classes.

There is a slight problem with Equation □. Different container types that happen to contain identical objects produce exactly the same hash value. For example, an empty stack and an empty list both produce the same hash value. We have avoided this situation in Program □ by adding to the sum the value obtained from hashing the class of the container itself.
Using Associations

Hashing provides a way to determine the position of a given object directly from that object itself. Given an object \( x \) we determine its position by evaluating the appropriate hash function, \( h(x) \). We find the location of object \( x \) in exactly the same way. But of what use is this ability to find an object if, in order to compute the hash function \( h(x) \), we must be able to access the object \( x \) in the first place?

In practice, when using hashing we are dealing with *keyed data*. Mathematically, keyed data consists of ordered pairs

\[
A = \{(k, v) : k \in K, v \in V \},
\]

where \( K \) is a set of keys, and \( V \) is a set of values. The idea is that we will access elements of the set \( A \) using the key. That is, the hash function for elements of the set \( A \) is given by

\[
f_A((k, v)) = f_K(k),
\]

where \( f_K \) is the hash function associated with the set \( K \).

For example, suppose we wish to use hashing to implement a database which contains driver's license records. Each record contains information about a driver, such as her name, address, and perhaps a summary of traffic violations. Furthermore, each record has a unique driver's license number. The driver's license number is the key and the other information is the value associated with that key.

In Section  the Association class was declared which comprises two fields, a key and a value. Given this declaration, the definition of the hash method for Associations is trivial. As shown in Program  it simply calls the GetHashCode method on the key field.
```java
public class Association : ComparableObject
{
    protected IComparable key;
    protected object value;

    public override int GetHashCode()
    {
        return key.GetHashCode();
    }

    // ...

}
```

Program: Association class GetHashCode method.
Hash Tables

A hash table is a searchable container. As such, its interface provides methods for putting an object into the container, finding an object in the container, and removing an object from the container. Program defines the HashTable interface. The HashTable interface extends the SearchableContainerInterface defined in Program. One additional property, called LoadFactor, is declared. The purpose of this property is explained in Section.

```csharp
public interface HashTable : SearchableContainer
{
    double LoadFactor { get; }
}
```

Program: HashTable interface.

- Abstract Hash Tables
- Separate Chaining
- Average Case Analysis

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Abstract Hash Tables

As shown in Figure □, we define an `AbstractHashTable` class from which several concrete realizations are derived.

**Figure**: Object class hierarchy.

Program □ introduces the `AbstractHashTable` class. The `AbstractHashTable` class extends the `AbstractSearchableContainer` class introduced in Program □ and it implements the `HashTable` interface defined in Program □.
Abstract Hash Tables

public abstract class AbstractHashTable :
AbstractSearchableContainer, HashTable
{
    public abstract int Length { get; }

    protected int F(object obj)
    { return obj.GetHashCode(); }

    protected int G(int x)
    { return Math.Abs(x) % Length; }

    protected int H(object obj)
    { return G(F(obj)); }

    // ...
}

Program: AbstractHashTable methods.

Program introduces the Length property and three methods, F, G, and H. The Length property is an abstract property that provides a get accessor that returns the length of a hash table.

The methods F, G, and H correspond to the composition $h = g \circ f$ discussed in Section. The F method takes as an object and calls the GetHashCode method on that object to compute an integer. The G method uses the division method of hashing defined in Section to map an integer into the interval $[0,M-1]$, where $M$ is the length of the hash table. Finally, the H method computes the composition of F and G.

In the following we will consider various ways of implementing hash tables. In all cases, the underlying implementation makes use of an array. The position of an object in the array is determined by hashing the object. The main problem to be resolved is how to deal with collisions--two different objects cannot occupy the same array position at the same time. In the following section, we consider an approach which solves the problem of collisions by keeping objects that collide in a linked list.
Abstract Hash Tables

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Separate Chaining

Figure shows a hash table that uses separate chaining to resolve collisions. The hash table is implemented as an array of linked lists. To insert an item into the table, it is appended to one of the linked lists. The linked list to it is appended is determined by hashing that item.

Table illustrates an example in which there are $M=16$ linked lists. The twelve character strings "ett" and "tolv" have been inserted into the table using the hashed values and in the order given in Table. Notice that in this example since $M=16$, the linked list is selected by the least significant four bits of the hashed value given in Table. In effect, it is only the last letter of a string which determines the linked list in which that string appears.
Separate Chaining

- Implementation
- Constructor, Length Property and Purge Methods
- Inserting and Removing Items
- Finding an Item

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Implementation

Program  introduces the ChainedHashTable class. The ChainedHashTable class extends the AbstractHashTable class introduced in Program . The ChainedHashTable class contains a single field called array. It is declared as an array of LinkedLists. (The LinkedList class is described in Chapter ).

```
public class ChainedHashTable : AbstractHashTable
{
    protected LinkedList[] array;

    // ...
}
```

Program: ChainedHashTable fields.
Constructor, Length Property and Purge Methods

The constructor, Length property, and Purge methods of the ChainedHashTable class are defined in Program 1. The constructor takes a single argument which specifies the size of hash table desired. It creates an array of the specified length and then initializes the elements of the array. Each element of the array is assigned an empty linked list. The running time for the ChainedHashTable constructor is $O(M)$ where $M$ is the size of the hash table.

```csharp
public class ChainedHashTable : AbstractHashTable
{
    protected LinkedList[] array;

    public ChainedHashTable(int length)
    {
        array = new LinkedList[length];
        for (int i = 0; i < length; ++i)
        {
            array[i] = new LinkedList();
        }
    }

    public override int Length
    { get { return array.Length; } }

    public override void Purge()
    {
        for (int i = 0; i < array.Length; ++i)
        {
            array[i].Purge();
            count = 0;
        }
        // ...
    }
}
```

Program: ChainedHashTable class constructor, Length property, and Purge methods.

The Length property provides a get accessor that returns the length of the array field. Clearly its
The purpose of the `Purge` method is to make the container empty. It does this by invoking the `Purge` method one-by-one on each of the linked lists in the array. The running time of the `Purge` method is $O(M)$, where $M$ is the size of the hash table.
Inserting and Removing Items

Program gives the code for inserting and removing items from a ChainedHashTable.

```csharp
public class ChainedHashTable : AbstractHashTable
{
    protected LinkedList[] array;

    public override void Insert(ComparableObject obj)
    {
        array[H(obj)].Append(obj);
        ++count;
    }

    public override void Withdraw(ComparableObject obj)
    {
        array[H(obj)].Extract(obj);
        --count;
    }

    // ...
}
```

Program: ChainedHashTable class Insert and Withdraw methods.

The implementations of the Insert and Withdraw methods are remarkably simple. For example, the Insert method first calls the hash method \( H \) to compute an array index which is used to select one of the linked lists. The Append method provided by the LinkedList class is used to add the object to the selected linked list. The total running time for the Insert operation is \( T\{\text{GetHashCode}\} + O(1) \), where \( T\{\text{GetHashCode}\} \) is the running time of the GetHashCode method. Notice that if the hash method runs in constant time, then so too does hash table insertion operation!

The Withdraw method is almost identical to the Insert method. Instead of calling the Append, it calls the linked list Extract method to remove the specified object from the appropriate linked list. The running time of Withdraw is determined by the time of the Extract operation. In Chapter this
was shown to be $O(n)$ where $n$ is the number of items in the linked list. In the worst case, all of the items in the ChainedHashTable have collided with each other and ended up in the same list. That is, in the worst case if there are $n$ items in the container, all $n$ of them are in a single linked list. In this case, the running time of the Withdraw operation is $T(\text{HashCode}) + O(n)$. 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Finding an Item

The definition of the `Find` method of the `ChainedHashTable` class is given in Program 1. The `Find` method takes as its argument any `ComparableObject`. The purpose of the `Find` operation is to return the object in the container that is equal to the given object.

```csharp
public class ChainedHashTable : AbstractHashTable
{
    protected LinkedList[] array;

    public override ComparableObject Find( ComparableObject obj )
    {
        for ( LinkedList.Element ptr = array[H(obj)].Head; ptr != null; ptr = ptr.Next )
        {
            ComparableObject datum = (ComparableObject)ptr.Datum;
            if ( obj == datum )
                return datum;
        }
        return null;
    }

    // ...
}
```

**Program:** ChainedHashTable class `Find` method.

The `Find` method simply hashes its argument to select the linked list in which it should be found. Then, it traverses the linked list to locate the target object. As for the `Withdraw` operation, the worst case running time of the `Find` method occurs when all the objects in the container have collided, and the item that is being sought does not appear in the linked list. In this case, the running time of the find operation is $nT(=) + T(GetHashCode) + O(n)$. 
Average Case Analysis

The previous section has shown that in the worst case, the running time to insert an object into a separately chained hash table is $O(1)$, and the time to find or delete an object is $O(n)$. But these bounds are no better than the same operations on plain lists! Why have we gone to all the trouble inventing hash tables?

The answer lies not in the worst-case performance, but in the average expected performance. Suppose we have a hash table of size $M$. Let there be exactly $n$ items in the hash table. We call the quantity $\lambda = n/M$ the load factor. The load factor is simply the ratio of the number of items in the hash table to the array length.

Program gives the implementation for the get accessor of the LoadFactor property of the AbstractHashTable class. This method computes $\lambda = n/M$ by calling the Count method to determine $n$ and the Length method to determine $M$.

```csharp
    public abstract class AbstractHashTable : 
    AbstractSearchableContainer, HashTable
    {
        public abstract int Length { get; }

        public virtual double LoadFactor
        {
            get { return (double)Count / Length; }
            // ...
        }
    }
```

Program: AbstractHashTable class LoadFactor method.

Consider a chained hash table. Let $n_i$ be the number of items in the $i^{th}$ linked list, for $i = 0, 1, \ldots, M - 1$. The average length of a linked list is
Average Case Analysis

If we are given the load factor \( \lambda \), we can determine the average running times for the various operations. The average running time of Insert is the same as its worst case time, \( O(1) \)--this result does not depend on \( \lambda \). On the other hand, the average running time for Withdraw does depend on \( \lambda \). It is \( T\langle\text{GetHashCode}\rangle + O(1) + \mathcal{O}(\lambda) \) since the time required to delete an item from a linked list of length \( \lambda \) is \( \mathcal{O}(\lambda) \).

To determine the average running time for the Find operation, we need to make an assumption about whether the item that is being sought is in the table. If the item is not found in the table, the search is said to be unsuccessful. The average running time for an unsuccessful search is

\[
\frac{1}{M} \sum_{i=0}^{M-1} n_i = \frac{n}{M} = \lambda.
\]

Thus, the average running time for a successful search is

\[
T\langle\text{GetHashCode}\rangle + \lambda T\langle=\rangle + O(1) + \mathcal{O}(\lambda).
\]

So, while any one search operation can be as bad as \( O(n) \), if we do a large number of random searches, we expect that the average running time will be \( \mathcal{O}(\lambda) \). In fact, if we have a sufficiently good hash function and a reasonable set of objects in the container, we can expect that those objects are distributed throughout the table. Therefore, any one search operation will not be very much worse than the worst case.

Finally, if we know how many objects will be inserted into the hash table \textit{a priori}, then we can choose a table size \( M \) which is larger than the maximum number of items expected. By doing this, we can ensure that \( \lambda = n/M \leq 1 \). That is, a linked list contains no more than one item on average. In this case, the average time for Withdraw is \( T\langle\text{GetHashCode}\rangle + O(1) \) and for Find it is...
\[ T\{\text{HashCode}\} + T\{=\} + O(1). \]
Scatter Tables

The separately chained hash table described in the preceding section is essentially a linked-list implementation. We have seen both linked-list and array-based implementations for all of the data structures considered so far and hash tables are no exception. Array-based hash tables are called scatter tables.

The essential idea behind a scatter table is that all of the information is stored within a fixed size array. Hashing is used to identify the position where an item should be stored. When a collision occurs, the colliding item is stored somewhere else in the array.

One of the motivations for using scatter tables can be seen by considering again the linked-list hash table shown in Figure. Since most of the linked lists are empty, much of the array is unused. At the same time, for each item that is added to the table, dynamic memory is consumed. Why not simply store the data in the unused array positions?

- Chained Scatter Table
- Average Case Analysis
Chained Scatter Table

Figure illustrates a *chained scatter table*. The elements of a chained scatter table are ordered pairs. Each array element contains a key and a "pointer." All keys are stored in the table itself. Consequently, there is a fixed limit on the number of items that can be stored in a scatter table.

Since the pointers point to other elements in the array, they are implemented as integer-valued array subscripts. Since valid array subscripts start from the value zero, the *null* pointer must be represented not as zero, but by an integer value that is outside the array bounds (say -1).

To find an item in a chained scatter table, we begin by hashing that item to determine the location from which to begin the search. For example, to find the string "elva", which hashes to the value...
Chained Scatter Table

we begin the search in array location \( \lfloor 044556741 \rfloor \mod 16 = 1 \). The item at that location is "fyra", which does not match. So we follow the pointer in location \( 1 \) to location \( 2 \). The item there, "fyra", does not match either. We follow the pointer again, this time to location \( 3 \) where we ultimately find the string we are looking for.

Comparing Figures \( \square \) and \( \square \), we see that the chained scatter table has embedded within it the linked lists which appear to be the same as those in the separately chained hash table. However, the lists are not exactly identical. When using the chained scatter table, it is possible for lists to coalesce.

For example, when using separate chaining, the keys "tre" and "sju" appear in a separate list from the key "tolv". This is because both "tre" and "sju" hash to position \( 5 \), whereas "tolv" hashes to position \( 6 \). The same keys appear together in a single list starting at position \( 5 \) in the chained scatter table. The two lists have coalesced.

---

- Implementation
- Constructor, Length Property, and Purge Methods
- Inserting and Finding an Item
- Removing Items
- Worst-Case Running Time

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Implementation

The ChainedScatterTable class is introduced in Program ChainedScatterTable. This class extends the AbstractHashTableClass introduced in Program AbstractHashTableClass. The scatter table is implemented as an array of Entry structs. The Entry struct is a nested struct defined within the ChainedScatterTable class.

```csharp
public class ChainedScatterTable : AbstractHashTable
{
    protected Entry[] array;

    private const int NULL = -1;

    protected struct Entry
    {
        internal ComparableObject obj;
        internal int next;

        internal Entry(ComparableObject obj, int next)
        {
            this.obj = obj;
            this.next = next;
        }

        // ...
    }
}
```

**Program:** ChainedScatterTable fields and ChainedScatterTable.Entry struct.

Each Entry instance has two fields--obj and next. The former refers to a ComparableObject. The latter indicates the position in the array of the next element of a chain. The value of the constant NULL will be used instead of zero to mark the end of a chain. The value zero is not used to mark the end of a chain because zero is a valid array subscript.
Constructor, Length Property, and Purge Methods

Program defines the constructor, Length property, and Purge methods of the ChainedScatterTable class. The constructor takes a single argument which specifies the size of scatter table desired. It creates an array of the desired length and initializes each element of the array by assigning to it a new Entry instance. Consequently, the running time for the ChainedScatterTable constructor is $O(M)$ where $M$ is the size of the scatter table.

```csharp
public class ChainedScatterTable : AbstractHashTable
{
    protected Entry[] array;

    public ChainedScatterTable(int length)
    {
        array = new Entry[length];
        for (int i = 0; i < length; ++i)
            array[i] = new Entry(null, NULL);
    }

    public override int Length
    { get { return array.Length; } }

    public override void Purge()
    {
        for (int i = 0; i < Length; ++i)
            array[i] = new Entry(null, NULL);
        count = 0;
    }
}
```

**Program:** ChainedScatterTable class constructor, Length, and Purge methods.

The Length property provides a get accessor that returns the length of the array field. Clearly, its
running time is $O(1)$.

The `Purge` method empties the scatter table by assigning null values to each entry in the table. The time required to purge the scatter table is $O(M)$, where $M$ is the length of the table.
Inserting and Finding an Item

Program gives the code for the Insert and Find methods of the ChainedScatterTable class. To insert an item into a chained scatter table we need to find an unused array location in which to put the item. We first hash the item to determine the "natural" location for that item. If the natural location is unused, we store the item there and we are done.

```csharp
public class ChainedScatterTable : AbstractHashTable
{
    protected Entry[] array;

    public override void Insert(ComparableObject obj)
    {
        if (count == Length)
            throw new ContainerFullException();
        int probe = H(obj);
        if (array[probe].obj != null)
        {
            while (array[probe].next != NULL)
                probe = array[probe].next;
            int tail = probe;
            probe = (probe + 1) % Length;
            while (array[probe].obj != null)
                probe = (probe + 1) % Length;
            array[tail].next = probe;
        }
        array[probe] = new Entry(obj, NULL);
        ++count;
    }

    public override ComparableObject Find(ComparableObject obj)
    {
        for (int probe = H(obj);
            probe != NULL; probe = array[probe].next)
    }
```
Inserting and Finding an Item

Program: ChainedScatterTable class Insert and Find methods.

However, if the natural position for an item is occupied, then a collision has occurred and an alternate location in which to store that item must be found. When a collision occurs it must be the case that there is a chain emanating from the natural position for the item. The insertion algorithm given always adds items at the end of the chain. Therefore, after a collision has been detected, the end of the chain is found (lines 12-13).

After the end of the chain is found, an unused array position in which to store the item must be found. This is done by a simple, linear search starting from the array position immediately following the end of the chain (lines 14-17). Once an unused position is found, it is linked to the end of the chain (line 18), and the item is stored in the unused position (line 20).

The worst case running time for insertion occurs when the scatter table has only one unused entry. That is, when the number of items in the table is \( n=M-1 \), where \( M \) is the table size. In the worst case, all of the used array elements are linked into a single chain of length \( M-1 \) and the item to be inserted hashes to the head of the chain. In this case, it takes \( O(M) \) to find the end of the chain. In the worst case, the end of the chain immediately follows the unused array position. Consequently, the linear search for the unused position is also \( O(M) \). Once an unused position has been found, the actual insertion can be done in constant time. Therefore, the running time of the insertion operation is \( T(\text{GetHashCode}) + O(M) \) in the worst case.

Program also gives the code for the Find method which is used to locate a given object in the scatter table. The algorithm is straightforward. The item is hashed to find its natural location in the table. If the item is not found in the natural location but a chain emanates from that location, the chain is followed to determine if that item appears anywhere in the chain.

The worst-case running time occurs when the item for which we are looking is not in the table, the table is completely full, and all of the entries are linked together into a single linked list. In this case, the running time of the Find algorithm is \( T(\text{GetHashCode}) + MT(\text{==}) + O(M) \).
Removing Items

Removing items from a chained scatter table is more complicated than putting them into the table. The goal when removing an item is to have the scatter table end up exactly as it would have appeared had that item never been inserted in the first place. Therefore, when an item is removed from the middle of a chain, items which follow it in the chain have to be moved up to fill in the hole. However, the moving-up operation is complicated by the fact that several chains may have coalesced.

Program gives an implementation of the Withdraw method of the ChainedScatterTable class. The algorithm begins by checking that the table is not empty (lines 7-8). To remove an item, we first have to find it. This is what the loop on lines 9-11 does. If the item to be deleted is not in the table, when this loop terminates the variable i has the value NULL and an exception is thrown (lines 12-13). Otherwise, the item a position i in the table is to be removed.

```csharp
public class ChainedScatterTable : AbstractHashTable {
    protected Entry[] array;

    public override void Withdraw(ComparableObject obj) {
        if (count == 0)
            throw new ContainerEmptyException();
        int i = H(obj);
        while (i != NULL && (object)obj != (object)array[i].obj)
            i = array[i].next;
        if (i == NULL)
            throw new ArgumentException("obj not found");
        for (; ; )
        {  int j = array[i].next;
            while (j != NULL)
            {  int h = H(array[j].obj);
                bool contained = false;
                for (int k = array[i].next;
                    k != array[j].next && !contained;
                    k = array[k].next)
            {
```
Removing Items

The purpose of the loop on lines 14-31 is to fill in the hole in the chain which results when the item at position \( i \) is removed by moving up items which follow it in the chain. What we need to do is to find the next item which follows the item at \( i \) that is safe to move into position \( i \). The loop on lines 15-27 searches the rest of the chain following the item at \( i \) to find an item which can be safely moved.

Figure shows the basic idea. The figures shows a chained scatter table of length ten that contains integer-valued keys. There is a single chain as shown in the figure. However, notice that the values in the chain are not all equal modulo 10. In fact, this chain must have resulted from the coalescing of three chains--one which begins in position 1, one which begins in position 2, and one which begins in position 5.

Program: ChainedScatterTable class Withdraw method.

```java
k = array[k].next
{
    if (k == h) contained = true;
}
if (!contained) break;
j = array[j].next;
}
if (j == NULL) break;
array[i].obj = array[j].obj;
i = j;
}
array[i] = new Entry(null, NULL);
for (int j = (i + Length - 1) % Length;
j != i; j = (j + Length - 1) % Length)
{
    if (array[j].next == i)
    {
        array[j].next = NULL;
        break;
    }
}
--count;
}
// ...
```
Suppose we wish to remove item 11 which is in position 2, which is indicated by the box in Figure (a). To delete it, we must follow the chain to find the next item that can be moved safely up to position 2. Item 02 which follows 11 and can be moved safely up to position 2 because that is the location to which it hashes. Moving item 02 up moves the hole down the list to position 3 (Figure (b)). Again we follow the chain to find that item 21 can be moved safely up giving rise to the situation in Figure (c).

Now we have a case where an item cannot be moved. Item 05 is the next candidate to be moved. However, it is in position 5 which is the position to which it hashes. If we were to move it up, then it would no longer be in the chain which emanates from position 5. In effect, the item would no longer be accessible! Therefore, it cannot be moved safely. Instead, we must move item 31 ahead of item 5 as shown in Figure (d). Eventually, the hole propagates to the end of the chain, where it can be delete.
Removing Items easily (Figure \( \square \) (e)).

The loop on lines 15-27 of Program \( \square \) finds the position \( j \) of an item which can be safely moved to position \( i \). The algorithm makes use of the following fact: An item can be safely moved up *only if it does not hash to a position which appears in the linked list between \( i \) and \( j \).* This is what the code on lines 17-24 tests.

When execution reaches line 28, either we have found an item which can be safely moved, or there does not exist such an item. If an item is found, it is moved up (line 29) and we repeat the whole process again. On the other hand, if there are no more items to be moved up, then the process is finished and the main loop (lines 14-31) terminates.

The statement on line 32 does the actual deed of removing the data from the position which \( i \) which by now is at the end of the chain. The final task to be done is to remove the pointer to position \( i \), since there is no longer any data at that position. That is the job of the loop on lines 33-40.
Worst-Case Running Time

Computing a tight bound on the worst-case running time of Program is tricky. Assuming the item to be removed is actually in the table, then the time required to find the item (lines 9-11) is

\[ T\langle \text{GetHashCode}\rangle + M T\langle =\rangle + O(M) \]

in the worst case.

The worst-case running time of the main loop occurs when the table is full, there is only one chain, and no items can be safely moved up in the chain. In this case, the running time of the main loop (lines 14-31) is

\[ \left( \frac{(M - 1)M}{2} \right) T\langle \text{GetHashCode}\rangle + O(M^2). \]

Finally, the worst case running time of the last loop (lines 33-40) is \( O(M) \).

Therefore, the worst-case running time of the Withdraw method for chained scatter tables is

\[ \left( 1 + \frac{(M - 1)M}{2} \right) T\langle \text{GetHashCode}\rangle + M T\langle =\rangle + O(M^2). \]

Clearly we don't want to be removing items from a chained scatter table very often!
Average Case Analysis

The previous section has shown that the worst-case running time to insert or to find an object into a chained scatter table is $O(M)$. The average case analysis of chained scatter tables is complicated by the fact that lists coalesce. However, if we assume that chains never coalesce, then the chains which appear in a chained scatter table for a given set of items are identical to those which appear in a separately chained hash table for the same set of items.

Unfortunately we cannot assume that lists do not coalesce--they do! We therefore expect that the average list will be longer than $\lambda$ and that the running times are correspondingly slower. Knuth has shown that the average number of probes in an unsuccessful search is

$$U(\lambda) \approx 1 + \frac{1}{4}(e^{2\lambda} - 1 - 2\lambda),$$

and the average number of probes in a successful search is approximately

$$S(\lambda) \approx 1 + \frac{1}{8\lambda}(e^{2\lambda} - 1 - 2\lambda) + \frac{\lambda}{4},$$

where $\lambda$ is the load factor[29]. The precise functional form of $U(\lambda)$ and $S(\lambda)$ is not so important here. What is important is that when $\lambda = 1$, i.e., when the table is full, $U(1) \approx 2.1$ and $S(1) \approx 1.8$. Regardless of the size of the table, an unsuccessful search requires just over two probes on average, and a successful search requires just under two probes on average!

Consequently, the average running time for insertion is

$$T\langle\text{GetHashCode}\rangle + O(U(\lambda)) = T\langle\text{GetHashCode}\rangle + O(1),$$

since the insertion is always done in first empty position found. Similarly, the running time for an unsuccessful search is

$$T\langle\text{GetHashCode}\rangle + U(\lambda)T\langle=\rangle + O(U(\lambda)),$$

and for a successful search its
\[ T(\text{GetHashCode}) + S(\lambda)T(\rightarrow) + O(S(\lambda)). \]
Scatter Table using Open Addressing

An alternative method of dealing with collisions which entirely does away with the need for links and chaining is called \textit{open addressing}. The basic idea is to define a \textit{probe sequence} for every key which, when followed, always leads to the key in question.

The probe sequence is essentially a sequence of functions

\[
\{h_0, h_1, \ldots, h_{M-1}\},
\]

where \(h_i\) is a hash function, \(h_i : K \rightarrow \{0, 1, \ldots, M - 1\}\). To insert item \(x\) into the scatter table, we examine array locations \(h_0(x), h_1(x), \ldots\), until we find an empty cell. Similarly, to find item \(x\) in the scatter table we examine the same sequence of locations in the same order.

The most common probe sequences are of the form

\[
h_i(x) = (h(x) + c(i)) \mod M,
\]

where \(i = 0, 1, \ldots, M - 1\). The function \(h(x)\) is the same hash function that we have seen before. That is, the function \(h\) maps keys into integers in the range from zero to \(M-1\).

The function \(c(i)\) represents the collision resolution strategy. It is required to have the following two properties:

\textbf{Property 1}

\(c(0)=0\). This ensures that the first probe in the sequence is

\[
h_0(x) = (h(x) + 0) \mod M = h(x).
\]

\textbf{Property 2}

The set of values

\[
\{c(0) \mod M, c(1) \mod M, c(2) \mod M, \ldots, c(M - 1) \mod M\}
\]
must contain every integer between 0 and $M-1$. This second property ensures that the probe sequence eventually probes *every possible array position*.

- **Linear Probing**
- **Quadratic Probing**
- **Double Hashing**
- **Implementation**
- **Average Case Analysis**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Linear Probing

The simplest collision resolution strategy in open addressing is called linear probing. In linear probing, the function $c(i)$ is a linear function in $i$. That is, it is of the form

$$c(i) = \alpha i + \beta.$$

Property 1 requires that $c(0)=0$. Therefore, $\beta$ must be zero.

In order for $c(i) = \alpha i$ to satisfy Property 2, $\alpha$ and $M$ must be relatively prime. If we know the $M$ will always be a prime number, then any $\alpha$ will do. On the other hand, if we cannot be certain that $M$ is prime, then $\alpha$ must be one. Therefore, linear probing sequence that is usually used is

$$h_i = (h(x) + i) \mod M,$$

for $i = 0, 1, 2, \ldots, M - 1$.

Figure illustrates an example of a scatter table using open addressing together with linear probing. For example, consider the string "åtta". This string hashes to array position 18. The corresponding linear probing sequence begins at position 18 and goes on to positions 28, 38, etc. In this case, the search for the string "åtta" succeeds after three probes.
Linear Probing

To insert an item $x$ into the scatter table, an empty cell is found by following the same probe sequence that would be used in a search for item $x$. Thus, linear probing finds an empty cell by doing a linear search beginning from array position $h(x)$.

An unfortunate characteristic of linear probing arises from the fact that as the table fills, clusters of consecutive cells form and the time required for a search increases with the size of the cluster. Furthermore, when we attempt to insert an item in the table at a position which is already occupied, that item is ultimately inserted at the end of the cluster—thereby increasing its length. This by itself is not inherently a bad thing. After all, when using the chained approach, every insertion increase the length of some chain by one. However, whenever an insertion is made between two clusters that are separated by one unoccupied position, the two clusters become one, thereby potentially increasing the cluster length by an amount much greater than one—a bad thing! This phenomenon is called primary clustering.

---

**Figure:** Scatter table using open addressing and linear probing.

<table>
<thead>
<tr>
<th>0</th>
<th>tio</th>
<th>occupied</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fyr</td>
<td>occupied</td>
</tr>
<tr>
<td>2</td>
<td>âta</td>
<td>occupied</td>
</tr>
<tr>
<td>3</td>
<td>elv</td>
<td>occupied</td>
</tr>
<tr>
<td>4</td>
<td>ett</td>
<td>occupied</td>
</tr>
<tr>
<td>5</td>
<td>två</td>
<td>occupied</td>
</tr>
<tr>
<td>6</td>
<td>tre</td>
<td>occupied</td>
</tr>
<tr>
<td>7</td>
<td>sju</td>
<td>occupied</td>
</tr>
<tr>
<td>010</td>
<td>sex</td>
<td>occupied</td>
</tr>
<tr>
<td>011</td>
<td>tol</td>
<td>occupied</td>
</tr>
<tr>
<td>012</td>
<td>fem</td>
<td>occupied</td>
</tr>
<tr>
<td>013</td>
<td>empty</td>
<td></td>
</tr>
<tr>
<td>014</td>
<td>empty</td>
<td></td>
</tr>
<tr>
<td>015</td>
<td>fem</td>
<td>occupied</td>
</tr>
<tr>
<td>016</td>
<td>empty</td>
<td></td>
</tr>
<tr>
<td>017</td>
<td>nio</td>
<td>occupied</td>
</tr>
</tbody>
</table>

---

**Next** **Up** **Previous** **Contents** **Index**
Quadratic Probing

An alternative to linear probing that addresses the primary clustering problem is called \textit{quadratic probing}. In quadratic probing, the function \( c(i) \) is a quadratic function in \( i \). The general quadratic has the form

\[
c(i) = \alpha i^2 + \beta i + \gamma.
\]

However, quadratic probing is usually done using \( c(i) = i^2 \).

Clearly, \( c(i) = i^2 \) satisfies property 1. What is not so clear is whether it satisfies property 2. In fact, in general it does not. The following theorem gives the conditions under which quadratic probing works:

**Theorem** When quadratic probing is used in a table of size \( M \), where \( M \) is a prime number, the first \( \left\lfloor M/2 \right\rfloor \) probes are distinct.

\textbf{Proof (By contradiction).} Let us assume that the theorem is false. Then there exist two distinct values \( i \) and \( j \) such that \( 0 \leq i < j < \left\lfloor M/2 \right\rfloor \), that probe exactly the same position. Thus,

\[
\begin{align*}
  h_i(x) & = h_j(x) \quad \Rightarrow \quad h(x) + c(i) = h(x) + c(j) \quad (\text{mod } M) \\
  h(x) + i^2 & = h(x) + j^2 \quad (\text{mod } M) \\
  i^2 & = j^2 \quad (\text{mod } M) \\
  i^2 - j^2 & = 0 \quad (\text{mod } M) \\
  (i - j)(i + j) & = 0 \quad (\text{mod } M)
\end{align*}
\]

Since \( M \) is a prime number, the only way that the product \((i-j)(i+j)\) can be zero modulo \( M \) is for either \( i-j \) to be zero or \( i+j \) to be zero modulo \( M \). Since \( i \) and \( j \) are distinct, \( i - j \neq 0 \). Furthermore, since both \( i \) and \( j \) are less than \( \left\lfloor M/2 \right\rfloor \), the sum \( i+j \) is less than \( M \). Consequently, the sum cannot be zero. We have successfully argued an absurdity—if the theorem is false one of two quantities must be zero, neither of which can possibly be zero. Therefore, the original assumption is not correct and the theorem is true.
Applying Theorem we get that quadratic probing works as long as the table size is prime and there are fewer than $n = \frac{M}{2}$ items in the table. In terms of the load factor $\lambda = \frac{n}{M}$, this occurs when $\lambda < \frac{1}{2}$.

Quadratic probing eliminates the primary clustering phenomenon of linear probing because instead of doing a linear search, it does a quadratic search:

$$
\begin{align*}
    h_0(x) &= (h(x) + 0 \mod M) \\
    h_1(x) &= (h(x) + 1 \mod M) \\
    h_2(x) &= (h(x) + 4 \mod M) \\
    h_3(x) &= (h(x) + 9 \mod M) \\
    &\vdots
\end{align*}
$$
Double Hashing

While quadratic probing does indeed eliminate the primary clustering problem, it places a restriction on the number of items that can be put in the table--the table must be less than half full. Double Hashing is yet another method of generating a probing sequence. It requires two distinct hash functions,

\[ h : K \mapsto \{0, 1, \ldots, M - 1\}, \]
\[ h' : K \mapsto \{1, 2, \ldots, M - 1\}. \]

The probing sequence is then computed as follows

\[ h_i(x) = (h(x) + ih'(x)) \mod M. \]

That is, the scatter tables is searched as follows:

\[ h_0 = (h(x) + 0 \times h'(x)) \mod M \]
\[ h_1 = (h(x) + 1 \times h'(x)) \mod M \]
\[ h_2 = (h(x) + 2 \times h'(x)) \mod M \]
\[ h_3 = (h(x) + 3 \times h'(x)) \mod M \]
\[ \vdots \]

Since the collision resolution function is \( c(i)=ih'(x) \), the probe sequence depends on the key as follows: If \( h'(x)=1 \), then the probing sequence for the key \( x \) is the same as linear probing. If \( h'(x)=2 \), the probing sequence examines every other array position. This works as long as \( M \) is not even.

Clearly since \( c(0)=0 \), the double hashing method satisfies property 1. Furthermore, property 2 is satisfied as long as \( h'(x) \) and \( M \) are relatively prime. Since \( h'(x) \) can take on any value between 1 and \( M-1 \), \( M \) must be a prime number.

But what is a suitable choice for the function \( h' \)? Recall that \( h \) is defined as the composition of two functions, \( h = g \circ f \) where \( g(x) = x \mod M \). We can define \( h' \) as the composition \( g' \circ f \), where

\[ g'(x) = 1 + (x \mod (M - 1)). \]  \hspace{1cm} (8.6)
Double hashing reduces the occurrence of primary clustering since it only does a linear search if $h'(x)$ hashes to the value 1. For a good hash function, this should only happen with probability $1/(M-1)$. However, for double hashing to work at all, the size of the scatter table, $M$, must be a prime number. Table 3 summarizes the characteristics of the various open addressing probing sequences.

<table>
<thead>
<tr>
<th>probing sequence</th>
<th>primary clustering</th>
<th>capacity limit</th>
<th>size restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear probing</td>
<td>yes</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>quadratic probing</td>
<td>no</td>
<td>$\lambda &lt; \frac{1}{2}$</td>
<td>$M$ must be prime</td>
</tr>
<tr>
<td>double hashing</td>
<td>no</td>
<td>none</td>
<td>$M$ must be prime</td>
</tr>
</tbody>
</table>

Table: Characteristics of the open addressing probing sequences.
Implementation

This section describes an implementation of a scatter table using open addressing with linear probing. Program ❑ introduces the OpenScatterTable class. The OpenScatterTable class extends the AbstractHashTable class introduced in Program ❑. The scatter table is implemented as an array of elements of the nested struct Entry. Each Entry instance has two fields—obj and state. The former is refers to a ComparableObject. The latter is an EntryState enum the value of which is either EMPTY, OCCUPIED or DELETED.

```csharp
public class OpenScatterTable : AbstractHashTable
{
    protected Entry[] array;

    protected enum EntryState
    {
        EMPTY = 0,
        OCCUPIED,
        DELETED
    }

    protected struct Entry
    {
        internal EntryState state;
        internal ComparableObject obj;

        internal Entry(EntryState state, ComparableObject obj)
        {
            this.state = state;
            this.obj = obj;
        }

        // ...
    }
}
```
Program: OpenScatterTable fields, OpenScatterTable.EntryState enum, and OpenScatterTable.Entry struct.

Initially, all entries are empty. When an object recorded in an entry, the state of that entry is changed to OCCUPIED. The purpose of the third state, DELETED, will be discussed in conjunction with the Withdraw method below.

- Constructor, Length Property, and Purge Methods
- Inserting Items
- Finding Items
- Removing Items
Constructor, Length Property, and Purge Methods

Program defines the constructor, Length property, and Purge methods of the OpenScatterTable class. The OpenScatterTable constructor takes a single argument which specifies the size of scatter table desired. It creates an array of the desired length and initializes each element of the array by assigning to it an new Entry instance. Consequently, the running time for the OpenScatterTable constructor is $O(M)$ where $M$ is the size of the scatter table.

```csharp
public class OpenScatterTable : AbstractHashTable
{
    protected Entry[] array;

    public OpenScatterTable(int length)
    {
        array = new Entry[length];
        for (int i = 0; i < length; ++i)
            array[i] = new Entry(EntryState.EMPTY, null);
    }

    public override int Length
    { get { return array.Length; } }

    public override void Purge()
    {
        for (int i = 0; i < Length; ++i)
            array[i] = new Entry(EntryState.EMPTY, null);
        count = 0;
    }
    // ...
}
```

**Program:** OpenScatterTable class constructor, Length property, and Purge methods.

The Length property provides a get accessor that returns the length of the array field. Clearly, its
running time is $O(1)$.

The `Purge` method empties the scatter table by nulling out all the `Entries` in the array. The time required to purge the scatter table is $O(M)$, where $M$ is the length of the table.
Inserting Items

The method for inserting an item into a scatter table using open addressing is actually quite simple—find an unoccupied array location and then put the item in that location. To find an unoccupied array element, the array is probed according to a probing sequence. In this case, the probing sequence is linear probing. Program defines the methods needed to insert an item into the scatter table.

```csharp
public class OpenScatterTable : AbstractHashTable
{
    protected Entry[] array;

    protected static int C(int i)
    { return i; }

    protected int FindUnoccupied(object obj)
    {
        int hash = H(obj);
        for (int i = 0; i < count + 1; ++i)
        {
            int probe = (hash + C(i)) % Length;
            if (array[probe].state != EntryState.OCCUPIED)
                return probe;
        }
        throw new ContainerFullException();
    }

    public override void Insert(ComparableObject obj)
    {
        if (count == Length)
            throw new ContainerFullException();
        int offset = FindUnoccupied(obj);
        array[offset] = new Entry(EntryState.OCCUPIED, obj);
        ++count;
    }
    // ...
```
Inserting Items

```
24  
25  }  
```

**Program:** OpenScatterTable class C, FindUnoccupied, and Insert methods.

The method C defines the probing sequence. As it turns out, the implementation required for a linear probing sequence is trivial. The method C is the identity function.

The purpose of the private method FindUnoccupied is to locate an unoccupied array position. The FindUnoccupied method probes the array according the probing sequence determined by the C method. At most \( n+1 \) probes are made, where \( n = \text{count} \) is the number of items in the scatter table. When using linear probing it is always possible to find an unoccupied cell in this many probes as long as the table is not full. Notice also that we do not search for an EMPTY cell. Instead, the search terminates when a cell is found, the state of which is not OCCUPIED, i.e., EMPTY or DELETED. The reason for this subtlety has to do with the way items may be removed from the table. The FindUnoccupied method returns a value between 0 and \( M-1 \), where \( M \) is the length of the scatter table, if an unoccupied location is found. Otherwise, it throws an exception that indicates that the table is full.

The Insert method takes a ComparableObject and puts that object into the scatter table. It does so by calling FindUnoccupied to determine the location of an unoccupied entry in which to put the object. The state of the unoccupied entry is set to OCCUPIED and the object is saved in the entry.

The running time of the Insert method is determined by that of FindUnoccupied. The worst case running time of FindUnoccupied is \( O(n) \), where \( n \) is the number of items in the scatter table. Therefore, the running time of Insert is \( T\{\text{GetHashCode}\} + O(n) \).
Finding Items

The Find and FindMatch methods of the OpenScatterTable class are defined in Program □. The FindMatch method takes a ComparableObject and searches the scatter table for an object which matches the given one.

```csharp
public class OpenScatterTable : AbstractHashTable
{
    protected Entry[] array;

    protected int FindMatch(ComparableObject obj)
    {
        int hash = H(obj);
        for (int i = 0; i < Length; ++i)
        {
            int probe = (hash + C(i)) % Length;
            if (array[probe].state == EntryState.EMPTY)
                break;
            if (array[probe].state == EntryState.OCCUPIED
                && obj == array[probe].obj)
            {
                return probe;
            }
        }
        return -1;
    }

    public override ComparableObject Find(ComparableObject obj)
    {
        int offset = FindMatch(obj);
        if (offset >= 0)
            return array[offset].obj;
        else
            return null;
    }
```
Finding Items

    return null;
    }
    // ...
}

Program: OpenScatterTable Class FindMatch and Find methods.

FindMatch follows the same probing sequence used by the Insert method. Therefore, if there is a matching object in the scatter table, FindMatch will make exactly the same number of probes to locate the object as were made to put the object into the table in the first place. The FindMatch method makes at most $M$ probes, where $M$ is the size of the scatter table. However, note that the loop immediately terminates should it encounter an EMPTY location. This is because if the target has not been found by the time an empty cell is encountered, then the target is not in the table. Notice also that the comparison is only attempted for entries which are marked OCCUPIED. Any locations marked DELETED are not examined during the search but they do not terminate the search either.

The running time of the Find method is determined by that of FindMatch. In the worst case FindMatch makes $n$ comparisons, where $n$ is the number of items in the table. Therefore, the running time of Find is $T(\text{GetHashCode}) + nT(=) + O(M)$. 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Removing Items

Removing items from a scatter table using open addressing has to be done with some care. The naïve approach would be to locate the item to be removed and then change the state of its location to EMPTY. However, that approach does not work! Recall that the FindMatch method which is used to locate an item stops its search when it encounters an EMPTY cell. Therefore, if we change the state of a cell in the middle of a cluster to EMPTY, all subsequent searches in that cluster will stop at the empty cell. As a result, subsequent searches for an object may fail even when the target is still in the table!

One way to deal with this is to make use of the third state, DELETED. Instead of marking a location EMPTY, we mark it DELETED when an item is deleted. Recall that that the FindMatch method was written in such a way that it continues past deleted cells in its search. Also, the FindUnoccupied method was written to stop its search when it encounters either an EMPTY or a DELETED location. Consequently, the positions marked DELETED are available for reuse when insertion is done.

Program gives the implementation of the Withdraw. The Withdraw method takes a ComparableObject and removes that object from the scatter table. It does so by first locating the specific object instance using FindInstance and then marking the location DELETED. The implementation of FindInstance has been elided. It is simply a trivial variation of the FindMatch method.
Removing Items

```java
public class OpenScatterTable : AbstractHashTable
{
    protected Entry[] array;

    public override void Withdraw(ComparableObject obj)
    {
        if (count == 0)
            throw new ContainerEmptyException();
        int offset = FindInstance(obj);
        if (offset < 0)
            throw new ArgumentException("object not found");
        array[offset] = new Entry(EntryState.DELETED, null);
        --count;
    }
    // ...
}
```

Program: OpenScatterTable Class Withdraw method.

The running time of the Withdraw method is determined by that of FindInstance. In the worst case FindInstance has to examine every array position. Therefore, the running time of Withdraw is $T\{\text{HashCode}\} + \mathcal{O}(M)$.

There is a very serious problem with the technique of marking locations as DELETED. After a large number of insertions and deletions have been done, it is very likely that there are no cells left that are marked EMPTY. This is because, nowhere in any of the methods (except Purge) is a cell ever marked EMPTY! This has the very unfortunate consequence that an unsuccessful search, i.e., a search for an object which is not in the scatter table, is $\Omega(M)$. Recall that FindMatch examines at most $M$ array locations and only stops its search early when an EMPTY location is encountered. Since there are no more empty locations, the search must examine all $M$ locations.

If we are using the scatter table in an application in which we know a priori that no items will be removed, or perhaps only a very small number of items will be removed, then the Withdraw method given in Program will suffice. However, if the application is such that a significant number of withdrawals will be made, a better implementation is required.

Ideally, when removing an item the scatter table ends up exactly as it would have appeared had that item never been inserted in the first place. Note that exactly the same constraint is met by the Withdraw method for the ChainedScatterTable class given in Program. It turns out that a variation of
Removing Items

that algorithm can be used to implement the Withdraw method for the OpenScatterTable class as shown in Program 1.

```java
public class OpenScatterTableV2 : OpenScatterTable
{
    public override void Withdraw(Comparable obj)
    {
        if (count == 0)
            throw new ContainerEmptyException();
        int i = FindInstance(obj);
        if (i < 0)
            throw new ArgumentException("object not found");
        for (; ; )
        {
            int j = (i + 1) % Length;
            while (array[j].state == EntryState.OCCUPIED)
            {
                int h = H(array[j].obj);
                if (((h <= i && i < j) || (i < j && j < h) ||
                    (j < h && h <= i))
                    break;
                j = (j + 1) % Length;
            }
            if (array[j].state == EntryState.EMPTY)
                break;
            array[i] = array[j];
            i = j;
        }
        array[i] = new Entry(EntryState.EMPTY, null);
        --count;
        // ... 
    }
}
```

Program: OpenScatterTableV2 Withdraw method.

The algorithm begins by checking that the scatter table is not empty. Then it calls FindInstance to determine the position \( i \) of the item to be removed. If the item to be removed is not in the scatter table FindInstance returns -1 and an exception is thrown. Otherwise, FindInstance falls between 0
Removing Items

and $M-1$, which indicates that the item was found.

In the general case, the item to be deleted falls in the middle of a cluster. Deleting it would create a hole in the middle of the cluster. What we need to do is to find another item further down in the cluster which can be moved up to fill in the hole that would be created when the item at position $i$ is deleted. The purpose of the loop on lines 12-20 is to find the position $j$ of an item which can be moved safely into position $i$. Note the implementation here implicitly assumes that a linear probing sequence is used—the $C$ method is not called explicitly. An item at position $j$ can be moved safely to position $i$ only if the hash value of the item at position $j$ is not cyclically contained in the interval between $i$ and $j$.

If an item is found at some position $j$ that can be moved safely, then that item is moved to position $i$ on line 23. The effect of moving the item at position $j$ to position $i$, is to move the hole from position $i$ to position $j$ (line 24). Therefore, another iteration of the main loop (lines 10-25) is needed to fill in the relocated hole in the cluster.

If no item can be found to fill in the hole, then it is safe to split the cluster in two. Eventually, either because no item can be found to fill in the hole or because the hole has moved to the end of the cluster, there is nothing more to do other than to delete the hole. Thus, on line 26 the entry at position $i$ is set to $\text{EMPTY}$ and the associated $\text{obj}$ is set to $\text{null}$. Notice that the third state $\text{DELETED}$ is not required in this implementation of $\text{Withdraw}$.

If we use the $\text{Withdraw}$ implementation of Program $\square$, the scatter table entries will only ever be in one of two two states—$\text{OCCUPIED}$ or $\text{EMPTY}$. Consequently, we can improve the bound on the worst-case for the search from $T\langle \text{GetHashCode} \rangle + nT\langle = \rangle + O(M)$ to $T\langle \text{GetHashCode} \rangle + nT\langle = \rangle + O(n)$, where $n$ is the number of items in the scatter table.

Determining the running time of Program $\square$ is a little tricky. Assuming the item to be deleted is actually in the table, the running time to find the position of that item (line 7) is $T\langle \text{GetHashCode} \rangle + O(n)$, where $n = \text{count}$ is the number of items actually in the scatter table. In the worst case, the scatter table is comprised of a single cluster of $n$ items, and we are deleting the first item of the cluster. In this case, the main loop on lines 10-25 makes a pass through the entire cluster, in the worst case moving the hole to the end of the cluster one position at a time. Thus, the running time of the main loop is $(n - 1)T\langle \text{GetHashCode} \rangle + O(n)$. The remaining lines require a constant amount of additional time.

Altogether, the running time for the $\text{Withdraw}$ method is $nT\langle \text{GetHashCode} \rangle + O(n)$ in the worst case.
Average Case Analysis

The average case analysis of open addressing is easy if we ignore the primary clustering phenomenon. Given a scatter table of size $M$ that contains $n$ items, we assume that each of the $\binom{M}{n}$ combinations of $n$ occupied and $(m-n)$ empty scatter table entries is equally likely. This is the so-called uniform hashing model.

In this model we assume that the entries will either be occupied or empty, i.e., the DELETED state is not used. Suppose a search for an empty cell requires exactly $i$ probes. Then the first $i-1$ positions probed must have been occupied and the $i^{th}$ position probed was empty. Consider the $i$ cells which were probed. The number of combinations in which $i-1$ of the probed cells are occupied and one is empty is $\binom{M-i}{n-i+1}$. Therefore, the probability that exactly $i$ probes are required is

$$P_i = \frac{\binom{M-i}{n-i+1}}{\binom{M}{n}}. \quad (8.7)$$

The average number of probes required to find an empty cell in a table which has $n$ occupied cells is $U(n)$ where

$$U(n) = \sum_{i=1}^{M} i P_i. \quad (8.8)$$

Using Equation $\square$ into Equation $\square$ and simplifying the result gives

$$U(n) = \frac{M+1}{M - n + 1} \approx \frac{1 + \frac{1}{M}}{1 - \lambda + \frac{1}{M}}, \quad \text{where } \lambda = n/M \quad (8.9)$$

$$\approx \frac{1}{1 - \lambda} \quad (8.10)$$
Average Case Analysis

This result is actually quite intuitive. The load factor, $\lambda$, is the fraction of occupied entries. Therefore, $1 - \lambda$ entries are empty so we would expect to have to probe $1/(1 - \lambda)$ entries before finding an empty one! For example, if the load factor is 0.75, a quarter of the entries are empty. Therefore, we expect to have to probe four entries before finding an empty one.

To calculate the average number of probes for a successful search we make the observation that when an item is initially inserted, we need to find an empty cell in which to place it. For example, the number of probes to find the empty position into which the $i^{th}$ item is to be placed is $U(i)$. And this is exactly the number of probes it takes to find the $i^{th}$ item again! Therefore, the average number of probes required for a successful search in a table which has $n$ occupied cells is $S(n)$ where

$$S(n) = \frac{1}{n} \sum_{i=0}^{n-1} U(i). \quad (8.11)$$

Substituting Equation (8.11) in Equation (8.12) and simplifying gives

$$S(n) = \frac{1}{n} \sum_{i=0}^{n} \frac{M + 1}{M - i + 1}$$

$$= \frac{M + 1}{N} (H_{M+1} - H_{M-n+1})$$

$$\approx \frac{1}{\lambda} \ln \frac{1}{1 - \lambda} \quad (8.12)$$

where $H_k$ is the $k^{th}$ harmonic number (see Section 8). Again, there is an easy intuitive derivation for this result. We can use a simple integral to calculate the mean number of probes for a successful search using the approximation $U(n) = 1/(1 - \lambda)$ as follows

$$S(n) = \frac{1}{n} \sum_{i=0}^{n} U(i)$$

$$\approx \frac{1}{\lambda} \int_{0}^{\lambda} \frac{1}{1 - x} \, dx$$

$$\approx \frac{1}{\lambda} \ln \frac{1}{1 - \lambda}.$$

Empirical evidence has shown that the formulas derived for the uniform hashing model characterize the performance of scatter tables using open addressing with quadratic probing and double hashing quite
well. However, they do not capture the effect of primary clustering which occurs when linear probing is used. Knuth has shown that when primary clustering is taking into account, the number of probes required to locate an empty cell is

$$U(n) = \frac{1}{2} \left( 1 + \left( \frac{1}{1 - \lambda} \right)^2 \right),$$  \hspace{1cm} (8.13)

and the number of probes required for a successful search is

$$S(n) = \frac{1}{2} \left( 1 + \frac{1}{1 - \lambda} \right).$$  \hspace{1cm} (8.14)

The graph in Figure compares the predictions of the uniform hashing model (Equations and ) with the formulas derived by Knuth (Equations and ). Clearly, while the results are qualitatively similar, the formulas are in agreement for small load factors and they diverge as the load factor increases.

**Figure:** Number of probes vs. load factor for uniform hashing and linear probing.
Applications

Hash and Scatter tables have many applications. The principal characteristic of such applications is that keyed information needs to be frequently accessed and the access pattern is either unknown or known to be random. For example, hash tables are often used to implement the symbol table of a programming language compiler. A symbol table is used to keep track of information associated with the symbols (variable and method names) used by a programmer. In this case, the keys are character strings and each key hash associated with it some information about the symbol (e.g., type, address, value, lifetime, scope).

This section presents a simple application of hash and scatter tables. Suppose we are required to count the number of occurrences of each distinct word contained in a text file. We can do this easily using a hash or scatter table. Program gives the an implementation.

```csharp
public class Algorithms {
    {
        private class Counter : ComparableInt32 {
            {
                internal Counter(int value) : base(value) {
                {}
                public static Counter operator ++(Counter counter) {
                    counter.obj = (int)counter.obj + 1;
                    return counter;
                }
            }
        }
    public static void WordCounter(
        TextReader reader, TextWriter writer) {
            HashTable table = new ChainedHashTable(1031);
            string line;
            while ((line = reader.ReadLine()) != null) {
                foreach (string word in line.Split(null))
```
Program: Hash/scatter table application--counting words.

The private nested class `Counter` extends the class `ComparableInt32` defined in Section 0. In addition to the functionality inherited from the base class, the `Counter` class overloads the `++` operator which increments the value by one.

The `WordCounter` method does the actual work of counting the words in the input file. The local variable `table` refers to a `ChainedHashTable` that is used to keep track of the words and counts. The objects which are put into the hash table are all instances of the class `Association`. Each association has as its key a `String` class instance, and as its value a `Counter` class instance.

The main loop of the `WordCounter` method reads a line of text from the input stream. Each line of text is split into an array of words and the inner loop processes each word one at a time. For each word, a `Find` operation is done on the hash table to determine if there is already an association for the given key. If none is found, a new association is created and inserted into the hash table. The given word is used as the key of the new association and the value is a counter which is initialized to one. On the other hand, if there is already an association for the given word in the hash table, the corresponding counter is incremented. When the `WordCounter` method reaches the end of the input stream, it simply prints the hash table on the given output stream.

```csharp
foreach (string word in line.Split(null))
{
    Association assoc = (Association) table.Find(
        new Association(word));
    if (assoc == null)
    {
        table.Insert(
            new Association (word, new Counter(1)));
    }
    else
    {
        Counter counter = (Counter)assoc.Value;
        counter++;
    }
}
writer.WriteLine(table);
```
The running time of the `WordCounter` method depends on a number of factors, including the number of different keys, the frequency of occurrence of each key, and the distribution of the keys in the overall space of keys. Of course, the hash/scatter table implementation chosen has an effect as does the size of the table used. For a reasonable set of keys we expect the hash function to do a good job of spreading the keys uniformly in the table. Provided a sufficiently large table is used, the average search and insertion time is bounded by a constant. Under these ideal conditions the running time should be $O(n)$, where $n$ is the number of words in the input file.
Exercises

1. Suppose we know *a priori* that a given key is equally likely to be any integer between \( a \) and \( b \).
   1. When is the *division method of hashing* a good choice?
   2. When is the *middle square method of hashing* a good choice?

2. Compute (by hand) the hash value obtained by Program \( \square \) for the strings "ece.uw.ca" and "cs.uw.ca". **Hint:** Refer to Appendix \( \square \).

3. Canadian postal codes have the format LDL DLD where \( L \) is always a letter (A-Z), \( D \) is always a digit (0-9), and is always a single space. For example, the postal code for the University of Waterloo is N2L 3G1. Devise a suitable hash function for Canadian postal codes.

4. For each type of hash table listed below, show the hash table obtained when we insert the keys

\[
\{"un", "deux", "trois", "quatre", "cinq", "six", "sept", "huit", "neuf", "dix", "onze", "douze"\}
\]

in the order given into a table of size \( M=16 \) that is initially empty. Use the following table of hash values:

<table>
<thead>
<tr>
<th>( x )</th>
<th>Hash(( x )) (octal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;un&quot;</td>
<td>016456</td>
</tr>
<tr>
<td>&quot;deux&quot;</td>
<td>0145446470</td>
</tr>
<tr>
<td>&quot;trois&quot;</td>
<td>016563565063</td>
</tr>
<tr>
<td>&quot;quatre&quot;</td>
<td>010440656345</td>
</tr>
<tr>
<td>&quot;cinq&quot;</td>
<td>0142505761</td>
</tr>
<tr>
<td>&quot;six&quot;</td>
<td>01625070</td>
</tr>
<tr>
<td>&quot;sept&quot;</td>
<td>0162446164</td>
</tr>
<tr>
<td>&quot;huit&quot;</td>
<td>0151645064</td>
</tr>
<tr>
<td>&quot;neuf&quot;</td>
<td>0157446446</td>
</tr>
<tr>
<td>&quot;dix&quot;</td>
<td>01455070</td>
</tr>
<tr>
<td>&quot;onze&quot;</td>
<td>0156577345</td>
</tr>
<tr>
<td>&quot;douze&quot;</td>
<td>014556647345</td>
</tr>
</tbody>
</table>
Exercises

1. chained hash table,
2. chained scatter table,
3. open scatter table using linear probing,
4. open scatter table using quadratic probing, and
5. open scatter table using double hashing. (Use Equation $\square$ as the secondary hash function).

5. For each table obtained in Exercise $\square$, show the result when the key "deux" is withdrawn.
6. For each table considered in Exercise $\square$ derive an expression for the total memory space used to represent a table of size $M$ that contains $n$ items.
7. Consider a chained hash table of size $M$ that contains $n$ items. The performance of the table decreases as the load factor $\lambda = n/M$ increases. In order to keep the load factor below one, we propose to double the size of the array when $n=M$. However, in order to do so we must rehash all of the elements in the table. Explain why rehashing is necessary.
8. Give the sequence of $M$ keys that fills a chained scatter table of size $M$ in the shortest possible time. Find a tight, asymptotic bound on the minimum running time taken to fill the table.
9. Give the sequence of $M$ keys that fills a chained scatter table of size $M$ in the longest possible time. Find a tight, asymptotic bound on the minimum running time taken to fill the table.
10. Consider the chained hash table introduced shown in Program $\square$.

1. Rewrite the Insert method so that it doubles the length of the array when $\lambda = 1$. 
2. Rewrite the Withdraw method so that it halves the length of the array when $\lambda = \frac{1}{2}$. 
3. Show that the average time for both insert and withdraw operations is still $O(1)$.
11. Consider two sets of integers, $S = \{s_1, s_2, \ldots, s_m\}$ and $T = \{t_1, t_2, \ldots, t_n\}$.

1. Devise an algorithm that uses a hash table to test whether $S$ is a subset of $T$. What is the average running time of your algorithm?
2. Two sets are equivalent if and only if both $S \subseteq T$ and $T \subseteq S$. Show that we can test if two sets of integers are equivalent in $O(m+n)$ time (on average).
12. (This question should be attempted after reading Chapter $\square$). Rather than use an array of linked lists, suppose we implement a hash table with an array of binary search trees.

1. What are the worst-case running times for Insert, Find, and Withdraw. 
2. What are the average running times for Insert, Find, and Withdraw.
13. (This question should be attempted after reading Section $\square$). Consider a scatter table with open addressing. Devise a probe sequence of the form 

$$h_i(x) = (h(x) + c(i)) \mod M,$$

where $c(i)$ is a full-period pseudo random number generator. Why is such a sequence likely to be better than either linear probing or quadratic probing?
Projects

1. Complete the implementation of the `ChainedHashTable` class declared in Program by providing suitable definitions for the following operations: `IsMember`, `CompareTo`, `Accept`, and `GetEnumerator`. Write a test program and test your implementation.

2. Complete the implementation of the `ChainedScatterTable` class declared in Program by providing suitable definitions for the following operations: `IsFull`, `IsMember`, `CompareTo`, `Accept`, and `GetEnumerator`. Write a test program and test your implementation.

3. Complete the implementation of the `OpenScatterTable` class declared in Program by providing suitable definitions for the following methods: `IsFull`, `IsMember`, `FindInstance`, `CompareTo`, `Accept`, and `GetEnumerator`. Write a test program and test your implementation.

4. The `Withdraw` method defined in Program has been written under the assumption that linear probing is used. Therefore, it does not call explicitly the collision resolution method `C`. Rewrite the `Withdraw` method so that it works correctly regardless of the collision resolution strategy used.

5. Consider an application that has the following profile: First, \( n \) symbols (character strings) are read in. As each symbol is read, it is assigned an ordinal number from 1 to \( n \). Then, a large number of operations are performed. In each operation we are given either a symbol or a number and we need to determine its mate. Design, implement and test a data structure that provides both mappings in \( O(1) \) time.

6. Spelling checkers are often implemented using hashing. However, the space required to store all the words in a complete dictionary is usually prohibitive. An alternative solution is to use a very large array of bits. The array is initialized as follows: First, all the bits are set to zero. Then for each word \( w \) in the dictionary, we set bit \( h(w) \) to one, where \( h(\cdot) \) is a suitable hash function.

To check the spelling in a given document, we hash the words in the document one-by-one and examine the corresponding bit of the array. If the bit is a zero, the word does not appear in the dictionary and we conclude that it is misspelled. Note if the bit is a one, the word may still be misspelled, but we cannot tell.

Design and implement a spelling checker. **Hint:** Use the `SetAsBitVector` class given in Chapter.
In this chapter we consider one of the most important non-linear information structures—trees. A tree is often used to represent a hierarchy. This is because the relationships between the items in the hierarchy suggest the branches of a botanical tree.

For example, a tree-like organization chart is often used to represent the lines of responsibility in a business as shown in Figure 1. The president of the company is shown at the top of the tree and the vice-presidents are indicated below him. Under the vice-presidents we find the managers and below the managers the rest of the clerks. Each clerk reports to a manager, each manager reports to a vice-president, and each vice-president reports to the president.

A tree is extremely useful for certain kinds of computations. For example, suppose we wish to determine the total salaries paid to employees by division or by department. The total of the salaries in division A can be found by computing the sum of the salaries paid in departments A1 and A2 plus the salary of the vice-president of division A. Similarly, the total of the salaries paid in department A1 is the sum of the

---

**Figure:** Representing a hierarchy using a tree.

It just takes a little imagination to see the tree in Figure 1. Of course, the tree is upside-down. However, this is the usual way the data structure is drawn. The president is called the root of the tree and the clerks are the leaves.
salaries of the manager of department A1 and of the two clerks below him.

Clearly, in order to compute all the totals, it is necessary to consider the salary of every employee. Therefore, an implementation of this computation must visit all the employees in the tree. An algorithm that systematically visits all the items in a tree is called a tree traversal.

In this chapter we consider several different kinds of trees as well as several different tree traversal algorithms. In addition, we show how trees can be used to represent arithmetic expressions and how we can evaluate an arithmetic expression by doing a tree traversal.
The following is a mathematical definition of a tree:

**Definition (Tree)** A tree $T$ is a finite, non-empty set of nodes,

$$T = \{r\} \cup T_1 \cup T_2 \cup \cdots \cup T_n,$$

with the following properties:

1. A designated node of the set, $r$, is called the root of the tree; and
2. The remaining nodes are partitioned into $n \geq 0$ subsets, $T_1, T_2, \ldots, T_n$, each of which is a tree.

For convenience, we shall use the notation $T = \{r, T_1, T_2, \ldots, T_n\}$ to denote the tree $T$.

Notice that Definition is recursive--a tree is defined in terms of itself! Fortunately, we do not have a problem with infinite recursion because every tree has a finite number of nodes and because in the base case a tree has $n=0$ subtrees.

It follows from Definition that the minimal tree is a tree comprised of a single root node. For example $T_a = \{A\}$ is such a tree. When there is more than one node, the remaining nodes are partitioned into subtrees. For example, the $T_b = \{B, \{C\}\}$ is a tree which is comprised of the root node $B$ and the subtree $\{C\}$. Finally, the following is also a tree

$$T_c = \{D, \{E, \{F\}\}, \{G, \{H, \{I\}\}, \{J, \{K\}, \{L\}\}, \{M\}\}\}\}. \tag{9.1}$$

How do $T_a, T_b,$ and $T_c$ resemble their arboreal namesake? The similarity becomes apparent when we consider the graphical representation of these trees shown in Figure. To draw such a pictorial representation of a tree, $T = \{r, T_1, T_2, \ldots, T_n\}$, the following recursive procedure is used: First, we first draw the root node $r$. Then, we draw each of the subtrees, $T_1, T_2, \ldots, T_n$, beside each other below
the root. Finally, lines are drawn from \( r \) to the roots of each of the subtrees.

\[ T_a \quad T_b \quad T_c \]

**Figure:** Examples of trees.

Of course, trees drawn in this fashion are upside down. Nevertheless, this is the conventional way in which tree data structures are drawn. In fact, it is understood that when we speak of ``up'' and ``down,'' we do so with respect to this pictorial representation. For example, when we move from a root to a subtree, we will say that we are moving down the tree.

The inverted pictorial representation of trees is probably due to the way that genealogical *lineal charts* are drawn. A *lineal chart* is a family tree that shows the descendants of some person. And it is from genealogy that much of the terminology associated with tree data structures is taken.
Terminology

Consider a tree $T = \{r, T_1, T_2, \ldots, T_n\}, n \geq 0$, as given by Definition □.

- The degree of a node is the number of subtrees associated with that node. For example, the degree of tree $T$ is $n$.
- A node of degree zero has no subtrees. Such a node is called a leaf.
- Each root $r_i$ of subtree $T_i$ of tree $T$ is called a child of $r$. The term grandchild is defined in a similar manner.
- The root node $r$ of tree $T$ is the parent of all the roots $r_i$ of the subtrees $T_i$, $1 \leq i \leq n$. The term grandparent is defined in a similar manner.
- Two roots $r_i$ and $r_j$ of distinct subtrees $T_i$ and $T_j$ of tree $T$ are called siblings.

Clearly the terminology used for describing tree data structures is a curious mixture of mathematics, genealogy, and botany. There is still more terminology to be introduced, but in order to do that, we need the following definition:

**Definition (Path and Path Length)** Given a tree $T$ containing the set of nodes $R$, a path in $T$ is defined as a non-empty sequence of nodes $P = \{r_1, r_2, \ldots, r_k\}$, where $r_i \in R$, for $1 \leq i \leq k$, such that the $i^{th}$ node in the sequence, $r_i$, is the parent of the $(i + 1)^{th}$ node in the sequence $r_{i+1}$. The length of path $P$ is $k-1$.

For example, consider again the tree $T_c$ shown in Figure □. This tree contains many different paths. In fact, if you count carefully, you should find that there are exactly 29 distinct paths in tree $T_c$. This includes the path of length zero, $\{D\}$; the path of length one, $\{E, F\}$; and the path of length three, $\{D, G, J, K\}$.
More Terminology

Consider a tree $T$ containing the set of nodes $R$ as given by Definition.

- The **level** or **depth** of a node $r_i \in R$ in a tree $T$ is the length of the unique path in $T$ from its root $r$ to the node $r_i$. For example, the root of $T$ is at level zero and the roots of the subtrees are of $T$ are at level one.
- The **height of a node** $r_i \in R$ in a tree $T$ is the length of the longest path from node $r_i$ to a leaf. Therefore, the leaves are all at height zero.
- The **height of a tree** $T$ is the height of its root node $r$.
- Consider two nodes $r_i$ and $r_j$ in a tree $T$. The node $r_i$ is an **ancestor** of the node $r_j$ if there exists a path in $T$ from $r_i$ to $r_j$. Notice that $r_i$ and $r_j$ may be the same node. That is, a node is its own ancestor. However, the node $r_i$ is a **proper ancestor** if there exists a path $p$ in $T$ from $r_i$ to $r_j$ such that the length of the path $p$ is non-zero.
- Similarly, node $r_j$ is a **descendant** of the node $r_i$ if there exists a path in $T$ from $r_i$ to $r_j$. And since $r_i$ and $r_j$ may be the same node, a node is its own descendant. The node $r_j$ is a **proper descendant** if there exists a path $p$ in $T$ from $r_i$ to $r_j$ such that the length of the path $p$ is non-zero.
Alternate Representations for Trees

Figure shows an alternate representation of the tree defined in Equation. In this case, the tree is represented as a set of nested regions in the plane. In fact, what we have is a Venn diagram which corresponds to the view that a tree is a set of sets.

Figure: An alternate graphical representation for trees.

This hierarchical, set-within-a-set view of trees is also evoked by considering the nested structure of computer programs. For example, consider the following fragment of C# code:

class D {
    class E {
        class F {
        }
    }
    class G {
        class H {
            class I {}
        }
        class J {
            class K {}
            class L {}
        }
    }
}
class M {}
}

The nesting structure of this program and the tree given in Equation are isomorphic. Therefore, it is not surprising that trees have an important role in the analysis and translation of computer programs.
In the preceding section we considered trees in which the nodes can have arbitrary degrees. In particular, the general case allows each of the nodes of a tree to have a different degree. In this section we consider a variation in which all of the nodes of the tree are required to have exactly the same degree.

Unfortunately, simply adding to Definition the additional requirement that all of the nodes of the tree have the same degree does not work. It is not possible to construct a tree which has a finite number of nodes all of which have the same degree $N$ in any case except the trivial case of $N=0$. In order to make it work, we need to introduce the notion of an empty tree as follows:

**Definition (N-ary Tree)** An $N$-ary tree $T$ is a finite set of nodes with the following properties:

1. Either the set is empty, $T = \emptyset$, or
2. The set consists of a root, $R$, and exactly $N$ distinct $N$-ary trees. That is, the remaining nodes are partitioned into $N \geq 0$ subsets, $T_0, T_1, ..., T_{N-1}$, each of which is an $N$-ary tree such that $T = \{ R, T_0, T_1, \ldots, T_{N-1} \}$.

According to Definition, an $N$-ary tree is either the empty tree, $\emptyset$, or it is a non-empty set of nodes which consists of a root and exactly $N$ subtrees. Clearly, the empty set contains neither a root, nor any subtrees. Therefore, the degree of each node of an $N$-ary tree is either zero or $N$.

There is subtle, yet extremely important consequence of Definition that often goes unrecognized. The empty tree, $T = \emptyset$, is a tree. That is, it is an object of the same type as a non-empty tree. Therefore, from the perspective of object-oriented program design, an empty tree must be an instance of some object class. It is inappropriate to use the null reference to represent an empty tree, since the null reference refers to nothing at all!

The empty trees are called *external nodes* because they have no subtrees and therefore appear at the extremities of the tree. Conversely, the non-empty trees are called *internal nodes*.
N-ary Trees

Figure shows the following tertiary \((N=3)\) trees:

\[
\begin{align*}
T_a &= \{A, \emptyset, \emptyset, \emptyset\}, \\
T_b &= \{B, \{C, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset\}, \\
T_c &= \{D, \{E, \{F, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset\}, \\
&\quad \{G, \{H, \{I, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset\}, \{J, \{K, \emptyset, \emptyset, \emptyset\}, \{L, \emptyset, \emptyset, \emptyset\}, \emptyset\}, \\
&\quad \{M, \emptyset, \emptyset, \emptyset\}\}, \emptyset\}.
\end{align*}
\]

In the figure, square boxes denote the empty trees and circles denote non-empty nodes. Except for the empty trees, the tertiary trees shown in the figure contain the same sets of nodes as the corresponding trees shown in Figure.

---

**Figure**: Examples of \(N\)-ary trees.

Definitions and both define trees in terms of sets. In mathematics, elements of a set are normally
unordered. Therefore, we might conclude that that relative ordering of the subtrees is not important. However, most practical implementations of trees define an implicit ordering of the subtrees. Consequently, it is usual to assume that the subtrees are ordered. As a result, the two tertiary trees, $T_1 = \{x, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\}$ and $T_2 = \{x, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\}$, are considered to be distinct unequal trees. Trees in which the subtrees are ordered are called *ordered trees*. On the other hand, trees in which the order does not matter are called *oriented trees*. In this book, we shall assume that all trees are ordered unless otherwise specified.

Figure [image] suggests that every $N$-ary tree contains a significant number of external nodes. The following theorem tells us precisely how many external nodes we can expect:

**Theorem** An $N$-ary tree with $n \geq 0$ internal nodes contains $(N-1)n+1$ external nodes.

**Proof** Let the number of external nodes be $l$. Since every node except the root (empty or not) has a parent, there must be $(n+l-1)/N$ parents in the tree since every parent has $N$ children. Therefore, $n=(n+l-1)/N$. Rearranging this gives $l=(N-1)n+1$.

Since the external nodes have no subtrees, it is tempting to consider them to be the leaves of the tree. However, in the context of $N$-ary trees, it is customary to define a *leaf node* as an internal node which has only external subtrees. According to this definition, the trees shown in Figure [image] have exactly the same sets of leaves as the corresponding general trees shown in Figure [image].

Furthermore, since height is defined with respect to the leaves, by having the leaves the same for both kinds of trees, the heights are also the same. The following theorem tells us something about the maximum size of a tree of a given height $h$:

**Theorem** Consider an $N$-ary tree $T$ of height $h \geq 0$. The maximum number of internal nodes in $T$ is given by

$$\frac{N^{h+1} - 1}{N - 1}.$$ 

**Proof** (By induction).

**Base Case** Consider an $N$-ary tree of height zero. It consists of exactly one internal node and $N$ empty subtrees. Clearly the theorem holds for $h=0$ since

$$\left| \frac{N^{h+1} - 1}{N - 1} \right|_{h=0} = 1.$$
**Inductive Hypothesis** Suppose the theorem holds for \( h = 0, 1, 2, \ldots, k \), for some \( k \geq 0 \). Consider a tree of height \( k+1 \). Such a tree consists of a root and \( N \) subtrees each of which contains at most \((N^{k+1} - 1)/(N - 1)\) nodes. Therefore, altogether the number of nodes is at most

\[
N \left( \frac{N^{k+1} - 1}{N - 1} \right) + 1 = \frac{N^{k+2} - 1}{N - 1}.
\]  

That is, the theorem holds for \( k+1 \). Therefore, by induction on \( k \), the theorem is true for all values of \( h \).

An interesting consequence of Theorems □ and □ is that the maximum number of external nodes in an \( N \)-ary tree of height \( h \) is given by

\[
(N - 1) \left( \frac{N^{h+1} - 1}{N - 1} \right) + 1 = N^{h+1}.
\]

The final theorem of this section addresses the maximum number of leaves in an \( N \)-ary tree of height \( h \):

**Theorem** Consider an \( N \)-ary tree \( T \) of height \( h \geq 0 \). The maximum number of leaf nodes in \( T \) is \( N^h \).

**Proof (By induction).**

**Base Case** Consider an \( N \)-ary tree of height zero. It consists of exactly one internal node which has \( N \) empty subtrees. Therefore, the one node is a leaf. Clearly the theorem holds for \( h=0 \) since \( N^0 = 1 \).

**Inductive Hypothesis** Suppose the theorem holds for \( h = 0, 1, 2, \ldots, k \), for some \( k \geq 0 \). Consider a tree of height \( k+1 \). Such a tree consists of a root and \( N \) subtrees each of which contains at most \( N^k \) leaf nodes. Therefore, altogether the number of leaves is at most \( N \times N^k = N^{k+1} \). That is, the theorem holds for \( k+1 \). Therefore, by induction on \( k \), the theorem is true for all values of \( h \).
Binary Trees

In this section we consider an extremely important and useful category of tree structure—binary trees. A binary tree is an $N$-ary tree for which $N$ is two. Since a binary tree is an $N$-ary tree, all of the results derived in the preceding section apply to binary trees. However, binary trees have some interesting characteristics that arise from the restriction that $N$ is two. For example, there is an interesting relationship between binary trees and the binary number system. Binary trees are also very useful for the representation of mathematical expressions involving the binary operations such as addition and multiplication.

Binary trees are defined as follows:

**Definition (Binary Tree)** A binary tree $T$ is a finite set of nodes with the following properties:

1. Either the set is empty, $T = \emptyset$, or
2. The set consists of a root, $r$, and exactly two distinct binary trees $T_L$ and $T_R$, $T = \{r, T_L, T_R\}$.

The tree $T_L$ is called the left subtree of $T$, and the tree $T_R$ is called the right subtree of $T$.

Binary trees are almost always considered to be ordered trees. Therefore, the two subtrees $T_L$ and $T_R$ are called the left and right subtrees, respectively. Consider the two binary trees shown in Figure 1. Both trees have a root with a single non-empty subtree. However, in one case it is the left subtree which is non-empty; in the other case it is the right subtree that is non-empty. Since the order of the subtrees matters, the two binary trees shown in Figure 1 are different.
We can determine some of the characteristics of binary trees from the theorems given in the preceding section by letting $N=2$. For example, Theorem tells us that an binary tree with $n \geq 0$ internal nodes contains $n+1$ external nodes. This result is true regardless of the shape of the tree. Consequently, we expect that the storage overhead of associated with the empty trees will be $O(n)$.

From Theorem we learn that a binary tree of height $h \geq 0$ has at most $2^{h+1} - 1$ internal nodes. Conversely, the height of a binary tree with $n$ internal nodes is at least $\lceil \log_2 n + 1 \rceil - 1$. That is, the height of a binary tree with $n$ nodes is $\Omega(\log n)$.

Finally, according to Theorem, a binary tree of height $h \geq 0$ has at most $2^h$ leaves. Conversely, the height of a binary tree with $l$ leaves is at least $\lceil \log_2 l \rceil$. Thus, the height of a binary tree with $l$ leaves is $\Omega(\log l)$.
Tree Traversals

There are many different applications of trees. As a result, there are many different algorithms for manipulating them. However, many of the different tree algorithms have in common the characteristic that they systematically visit all the nodes in the tree. That is, the algorithm walks through the tree data structure and performs some computation at each node in the tree. This process of walking through the tree is called a tree traversal.

There are essentially two different methods in which to visit systematically all the nodes of a tree—depth-first traversal and breadth-first traversal. Certain depth-first traversal methods occur frequently enough that they are given names of their own: preorder traversal, inorder traversal and postorder traversal.

The discussion that follows uses the tree in Figure 9.1 as an example. The tree shown in the figure is a general tree in the sense of Definition 9.1:

$$T = \{A, \{B, \{C\}\}, \{D, \{E, \{F\}, \{G\}\}, \{H, \{I\}\}\}\}$$

(9.3)

However, we can also consider the tree in Figure 9.1 to be an N-ary tree (specifically, a binary tree if we assume the existence of empty trees at the appropriate positions):

$$T = \{A, \{B, \emptyset, \{C, \emptyset, \emptyset\}\}, \{D, \{E, \{F, \emptyset, \emptyset\}, \{G, \emptyset, \emptyset\}\}, \{H, \{I, \emptyset, \emptyset\}, \emptyset\}\}\}$$
Tree Traversals

**Figure:** Sample tree.

- **Preorder Traversal**
- **Postorder Traversal**
- **Inorder Traversal**
- **Breadth-First Traversal**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Preorder Traversal

The first depth-first traversal method we consider is called *preorder traversal*. Preorder traversal is defined recursively as follows. To do a preorder traversal of a general tree:

1. Visit the root first; and then
2. do a preorder traversal each of the subtrees of the root one-by-one in the order given.

Preorder traversal gets its name from the fact that it visits the root first. In the case of a binary tree, the algorithm becomes:

1. Visit the root first; and then
2. traverse the left subtree; and then
3. traverse the right subtree.

For example, a preorder traversal of the tree shown in Figure visits the nodes in the following order:

\[
A, B, C, D, E, F, G, H, I.
\]

Notice that the preorder traversal visits the nodes of the tree in precisely the same order in which they are written in Equation. A preorder traversal is often done when it is necessary to print a textual representation of a tree.
Postorder Traversal

The second depth-first traversal method we consider is postorder traversal. In contrast with preorder traversal, which visits the root first, postorder traversal visits the root last. To do a postorder traversal of a general tree:

1. Do a postorder traversal each of the subtrees of the root one-by-one in the order given; and then
2. visit the root.

To do a postorder traversal of a binary tree

1. Traverse the left subtree; and then
2. traverse the right subtree; and then
3. visit the root.

A postorder traversal of the tree shown in Figure visits the nodes in the following order:

\[ C, B, F, G, E, I, H, D, A. \]
Inorder Traversal

The third depth-first traversal method is *inorder traversal*. Inorder traversal only makes sense for binary trees. Whereas preorder traversal visits the root first and postorder traversal visits the root last, inorder traversal visits the root *in between* visiting the left and right subtrees:

1. Traverse the left subtree; and then
2. visit the root; and then
3. traverse the right subtree.

An inorder traversal of the tree shown in Figure visits the nodes in the following order:

\[ B, C, A, F, E, G, D, I, H. \]
Breadth-First Traversal

Whereas the depth-first traversals are defined recursively, *breadth-first traversal* is best understood as a non-recursive traversal. The breadth-first traversal of a tree visits the nodes in the order of their depth in the tree. Breadth-first traversal first visits all the nodes at depth zero (i.e., the root), then all the nodes at depth one, and so on. At each depth the nodes are visited from left to right.

A breadth-first traversal of the tree shown in Figure visits the nodes in the following order:

\[ A, B, D, C, E, H, F, G, I. \]
Expression Trees

Algebraic expressions such as

$$a/b + (c - d)e$$

have an inherent tree-like structure. For example, Figure is a representation of the expression in Equation. This kind of tree is called an expression tree.

The terminal nodes (leaves) of an expression tree are the variables or constants in the expression ($a$, $b$, $c$, $d$, and $e$). The non-terminal nodes of an expression tree are the operators ($+$, $-$, $\times$, and $\div$). Notice that the parentheses which appear in Equation do not appear in the tree. Nevertheless, the tree representation has captured the intent of the parentheses since the subtraction is lower in the tree than the multiplication.

![Figure: Tree representing the expression $a/b + (c - d)e$.]

The common algebraic operators are either unary or binary. For example, addition, subtraction, multiplication, and division are all binary operations and negation is a unary operation. Therefore, the non-terminal nodes of the corresponding expression trees have either one or two non-empty subtrees. That is, expression trees are usually binary trees.

What can we do with an expression tree? Perhaps the simplest thing to do is to print the expression...
Expression Trees

represented by the tree. Notice that an inorder traversal of the tree in Figure 1 visits the nodes in the order

$$a, \div, b, +, c, -, d, e.$$  

Except for the missing parentheses, this is precisely the order in which the symbols appear in Equation 1.

This suggests that an inorder traversal should be used to print the expression. Consider an inorder traversal which, when it encounters a terminal node simply prints it out; and when it encounters a non-terminal node, does the following:

1. Print a left parenthesis; and then
2. traverse the left subtree; and then
3. print the root; and then
4. traverse the right subtree; and then
5. print a right parenthesis.

Applying this procedure to the tree given in Figure 1 we get

$$((a \div b) + ((c - d) \times e)),$$  \hspace{1cm} (9.5)

which, despite the redundant parentheses, represents exactly the same expression as Equation 1.

---

- Infix Notation
- Prefix Notation
- Postfix Notation

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Infix Notation

The algebraic expression in Equation is written in the usual way such mathematical expressions are written. The notation used is called *infix notation* because each operator appears *in between* its operands. As we have seen, there is a natural relationship between infix notation and inorder traversal.

Infix notation is only possible for binary operations such as addition, subtraction, multiplication, and division. Writing an operator in between its operands is possible only when it has exactly two operands. In Chapter we saw two alternative notations for algebraic expressions--*prefix* and *postfix*.
Prefix Notation

In prefix notation the operator is written before its operands. Therefore, in order to print the prefix expression from an expression tree, preorder traversal is done. That is, at every non-terminal node we do the following:

1. Print the root; and then
2. print a left parenthesis; and then
3. traverse the left subtree; and then
4. print a comma; and then
5. traverse the right subtree; and then
6. print a right parenthesis.

If we use this procedure to print the tree given in Figure we get the prefix expression

\[ +\left(\div(a,b), \times(-c,d), e\right). \]

(9.6)

While this notation may appear unfamiliar at first, consider the result obtained when we spell out the names of the operators:

\[
\text{Plus}(\text{Div}(a,b), \text{Times}(\text{Minus}(c,d), e))
\]

This is precisely the notation used in a typical programming language to invoke user defined methods \text{Plus}, \text{Minus}, \text{Times}, and \text{Div}. 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Postfix Notation

Since inorder traversal produces an infix expression and preorder traversal produces a prefix expression, it should not come as a surprise that postorder traversal produces a postfix expression. In a postfix expression, an operator always follows its operands. The beauty of postfix (and prefix) expressions is that parentheses are not necessary.

A simple postorder traversal of the tree in Figure gives the postfix expression

\[ a \ b \div \ c \ d \ - \ e \ \times \ + . \]  \hspace{1cm} (9.7)

In Section we saw that a postfix expression is easily evaluated using a stack. So, given an expression tree, we can evaluate the expression by doing a postorder traversal to create the postfix expression and then using the algorithm given in Section to evaluate the expression.

In fact, it is not really necessary to first create the postfix expression before computing its value. The expression can be evaluated by making use of an evaluation stack during the course of the traversal as follows: When a terminal node is visited, its value is pushed onto the stack. When a non-terminal node is visited, two values are popped from the stack, the operation specified by the node is performed on those value, and the result is pushed back onto the evaluation stack. When the traversal terminates, there will be one result in the evaluation stack and that result is the value of the expression.

Finally, we can take this one step further. Instead of actually evaluating the expression, the code to compute the value of the expression is emitted. Again, a postorder traversal is done. However, now instead of performing the computation as each node is visited, the code needed to perform the evaluation is emitted. This is precisely what a compiler does when it compiles an expression such as Equation for execution.
Implementing Trees

In this section we consider the implementation of trees including general trees, \(N\)-ary trees, and binary trees. The implementations presented have been developed in the context of the abstract data type framework presented in Chapter \(\square\). That is, the various types of trees are viewed as classes of containers as shown in Figure \(\square\).

**Figure:** Object class hierarchy

Program \(\square\) defines the \textit{Tree} interface. The \textit{Tree} interface extends the \textit{Container} interface defined in Program \(\square\).
public interface Tree : Container
{
    object Key { get; }
    Tree GetSubtree(int i);
    bool IsLeaf { get; }
    int Degree { get; }
    int Height { get; }
    void DepthFirstTraversal(PrePostVisitor visitor);
    void BreadthFirstTraversal (Visitor visitor);
}

Program: Tree interface.

The Tree interface adds the following operations to those inherited from the Container interface:

Key
This property accesses the object contained in the root node of a tree.

GetSubtree
This method returns the \(i\)th subtree of the given tree.

IsEmpty
This bool-valued property is true if the root of the tree is an empty tree, i.e., an external node.

IsLeaf
This bool-valued property is true if the root of the tree is a leaf node.

Degree
This property accesses the degree of the root node of the tree. By definition, the degree of an external node is zero.

Height
This property accesses the height of the tree. By definition, the height of an empty tree is -1.

DepthFirstTraversal and BreadthFirstTraversal
These methods are like the Accept method of the container class (see Section□). Both of these methods perform a traversal. That is, all the nodes of the tree are visited systematically. The former takes a PrePostVisitor and the latter takes a Visitor. When a node is visited, the appropriate methods of the visitor are applied to that node.
Tree Traversals

Program introduces the AbstractTree class. The AbstractTree class extends the AbstractContainer class introduced in Program and it implements the Tree interface defined in Program. The AbstractTree class provides default implementations for both the DepthFirstTraversal and BreadthFirstTraversal methods. Both of these implementations access abstract properties, such as Key, and call abstract methods, such as and GetSubtree. In effect, they are abstract algorithms. An abstract algorithm describes behavior in the absence of implementation!

- Depth-First Traversal
- Preorder, Inorder, and Postorder Traversals
- Breadth-First Traversal
- Accept Method

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Depth-First Traversal

Program defines the DepthFirstTraversal method of the AbstractTree class. The traversal method takes one argument--any object that implements the PrePostVisitor interface defined in Program.

```csharp
public abstract class AbstractTree : AbstractContainer, Tree
{
    public virtual void DepthFirstTraversal(PrePostVisitor visitor)
    {
        if (visitor.IsDone)
            return;
        if (!IsEmpty)
        {
            visitor.PreVisit(Key);
            for (int i = 0; i < Degree; ++i)
                GetSubtree(i).DepthFirstTraversal(visitor);
            visitor.PostVisit(Key);
        }
        // ...
    }
}
```

**Program:** AbstractTree class DepthFirstTraversal method.

A PrePostVisitor is a visitor with three methods, PreVisit, InVisit, PostVisit, and the property IsDone. During a depth-first traversal the PreVisit and PostVisit methods are each called once for every node in the tree. (The InVisit method is provided for binary trees and is discussed in Section).
Depth-First Traversal

Program: PrePostVisitor interface.

The depth-first traversal method first calls the PreVisit method with the object in the root node. Then, it calls recursively the DepthFirstTraversal method for each subtree of the given node. After all the subtrees have been visited, the PostVisit method is called. Assuming that the IsEmpty, Key, and GetSubtree operations all run in constant time, the total running time of the DepthFirstTraversal method is

\[ n(T_{\text{PreVisit}} + T_{\text{PostVisit}}) + O(n), \]

where \( n \) is the number of nodes in the tree, \( T_{\text{PreVisit}} \) is the running time of PreVisit, and \( T_{\text{PostVisit}} \) is the running time of PostVisit.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Preorder, Inorder, and Postorder Traversals

Preorder, inorder, and postorder traversals are special cases of the more general depth-first traversal described in the preceding section. Rather than implement each of these traversals directly, we make use a design pattern pattern, called adapter, which allows the single method to provide all the needed functionality.

Suppose we have an instance of the PrintingVisitor class (see Section []). The PrintingVisitor class implements the Visitor interface. However, we cannot pass a PrintingVisitor instance to the DepthFirstTraversal method shown in Program because it expects an object that implements the PrePostVisitor interface.

The problem is that the interface implemented by the PrintingVisitor does not match the interface expected by the DepthFirstTraversal method. The solution to this problem is to use an adapter. An adapter converts the interface provided by one class to the interface required by another. For example, if we want a preorder traversal, then the call to the PreVisit (made by DepthFirstTraversal) should be mapped to the Visit method (provided by the PrintingVisitor). Similarly, a postorder traversal is obtained by mapping PostVisit to Visit.

Program defines the AbstractPrePostVisitor class. This class implements the PrePostVisitor interface defined in Program. It provides trivial default implementations for all the required methods.

```csharp
public abstract class AbstractPrePostVisitor : PrePostVisitor
{
    public virtual void PreVisit(object obj)
    {
    }
    public virtual void InVisit(object obj)
    {
    }
    public virtual void PostVisit(object obj)
    {
    }
    public virtual bool IsDone
    {
        get { return false; }
    }
}
```
Preorder, Inorder, and Postorder Traversals

**Program: AbstractPrePostVisitor class.**

Programs [ ] , [ ] and [ ] define three adapter classes--PreOrder, InOrder, and PostOrder. All three classes are similar: They all extend the AbstractPrePostVisitor class defined in Program [ ]; all have a single field that refers to a Visitor; and all have a constructor that takes a Visitor.

```java
public class PreOrder : AbstractPrePostVisitor
{
    protected Visitor visitor;

    public PreOrder(Visitor visitor)
    {
        this.visitor = visitor;
    }

    public override void PreVisit(object obj)
    {
        visitor.Visit(obj);
    }

    public override bool IsDone
    {
        get { return visitor.IsDone; }
    }
}
```

**Program: PreOrder class.**

```java
public class InOrder : AbstractPrePostVisitor
{
    protected Visitor visitor;

    public InOrder(Visitor visitor)
    {
        this.visitor = visitor;
    }

    public override void InVisit(object obj)
    {
        visitor.Visit(obj);
    }

    public override bool IsDone
    {
        get { return visitor.IsDone; }
    }
}
```

**Program: InOrder class.**
public class PostOrder : AbstractPrePostVisitor
{
    protected Visitor visitor;

    public PostOrder(Visitor visitor)
    {
        this.visitor = visitor;
    }

    public override void PostVisit(object obj)
    {
        visitor.Visit(obj);
    }

    public override bool IsDone
    {
        get { return visitor.IsDone; }
    }
}

Program: PostOrder class.

Each class provides a different interface mapping. For example, the PreVisit method of the PreOrder class simply calls the Visit method on the visitor field. Notice that the adapter provides no functionality of its own--it simply forwards method calls to the visitor instance as required.

The following code fragment illustrates how these adapters are used:

Visitor v = new PrintingVisitor();
Tree t = new SomeTree();
// ...
t.DepthFirstTraversal(new PreOrder(v));
t.DepthFirstTraversal(new InOrder(v));
t.DepthFirstTraversal(new PostOrder(v));

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Breadth-First Traversal

Program defines the BreadthFirstTraversal method of the AbstractTree class. As defined in Section, a breadth-first traversal of a tree visits the nodes in the order of their depth in the tree and at each level the nodes are visited from left to right.

```csharp
public abstract class AbstractTree : AbstractContainer, Tree
{
    public virtual void BreadthFirstTraversal(Visitor visitor)
    {
        Queue queue = new QueueAsLinkedList();
        if (!IsEmpty)
            queue.Enqueue(this);
        while (!queue.IsEmpty || !visitor.IsDone)
        {
            Tree head = (Tree)queue.Dequeue();
            visitor.Visit(head.Key);
            for (int i = 0; i < head.Degree; ++i)
            {
                Tree child = head.GetSubtree(i);
                if (!child.IsEmpty)
                    queue.Enqueue(child);
            }
        }
    // ...
}
```

Program: AbstractTree class BreadthFirstTraversal method.

We have already seen in Section a non-recursive breadth-first traversal algorithm for N-ary trees. This algorithm makes use of a queue as follows. Initially, the root node of the given tree is enqueued, provided it is not the empty tree. Then, the following steps are repeated until the queue is empty:
Breadth-First Traversal

1. Remove the node at the head of the queue and call it head.
2. Visit the object contained in head.
3. Enqueue in order each non-empty subtree of head.

Notice that empty trees are never put into the queue. Furthermore, it should be obvious that each node of the tree is enqueued exactly once. Therefore, it is also dequeue exactly once. Consequently, the running time for the breadth-first traversal is \( nT\{\text{visit}\} + O(n) \).
Accept Method

The AbstractTree class replaces the functionality provided by the single method Accept with two different kinds of traversal. Whereas the Accept method is allowed to visit the nodes of a tree in any order, the tree traversals visit the nodes in two different, but well-defined orders. Consequently, we have chosen to provide a default implementation of the Accept method which does a preorder traversal.

Program shows the implementation of the Accept method of the AbstractTree class. This method uses the PreOrder adapter to pass on a given visitor to the DepthFirstTraversal method.

```csharp
public abstract class AbstractTree : AbstractContainer, Tree
{
    public override void Accept(Visitor visitor)
    {
        DepthFirstTraversal(new PreOrder(visitor));
        // ...
    }
}
```

Program: AbstractTree class Accept method.
This section describes the implementation of an enumerator which can be used to step through the contents of any tree instance. For example, suppose we have declared a variable `tree` which refers to a `BinaryTree`. Then we can view the `tree` instance as a container and print its contents as follows:

```csharp
Tree tree = new BinaryTree();
// ...
IEnumerator e = tree.GetEnumerator();
while (e.MoveNext())
{
    Object obj = e.Current;
    Console.WriteLine(obj);
}
```

Every concrete class that implements the `Container` interface must provide a `GetEnumerator` method. This method returns an object that implements the `IEnumerator` interface defined in Section 10. The enumerator can then be used to systematically visit the contents of the associated container.

We have already seen that when we systematically visit the nodes of a tree, we are doing a tree traversal. Therefore, the implementation of the enumerator must also do a tree traversal. However, there is a catch. A recursive tree traversal method such as `DepthFirstTraversal` keeps track of where it is *implicitly* using the C# virtual machine stack. However, when we implement an enumerator we must keep track of the state of the traversal *explicitly*. This section presents an enumerator implementation which does a preorder traversal of the tree and keeps track of the current state of the traversal using a stack from Chapter 10.

Program 12 introduces the private nested class `Enumerator` declared within the `AbstractTree` class. The `Enumerator` class implements the `IEnumerator` interface defined in Section 10. The `Enumerator` contains two fields—`tree` and `stack`. As shown in Program 12, the `GetEnumerator` method of the `AbstractTree` class returns a new instance of the `Enumerator` class each time it is called.
Program: AbstractTree class GetEnumerator method and the Enumerator class.

- Constructor
- MoveNext Method and Current Property
Constructor

The code for the Enumerator constructor method is given in Program 1. Notice that it uses an instance of the StackAsLinkedList class. (The linked-list implementation of stacks is described in Section 1). An empty stack can be created in in constant time. Therefore, the running time of the constructor is $O(1)$.

```csharp
public abstract class AbstractTree : AbstractContainer, Tree
{
    protected class Enumerator : IEnumerable
    {
        public Enumerator(Tree tree)
        {
            this.tree = tree;
            stack = new StackAsLinkedList();
        }
        // ...
    }
    // ...
}
```

Program: AbstractTree Enumerator constructor.
MoveNext Method and Current Property

Program defines the standard operations provided by enumerators, the MoveNext and Reset methods, and the Current property. The Enumerator uses the stack to keep track nodes in the tree to be enumerated. As long as the stack is not empty, the Current property provides a get accessor that returns the key of the tree node at the top of the stack. Clearly, the running time for this accessor is $O(1)$.

```csharp
public abstract class AbstractTree : AbstractContainer, Tree
{
    protected class Enumerator : IEnumerator
    {
        public bool MoveNext()
        {
            if (stack.IsEmpty)
            {
                if (!tree.IsEmpty)
                    stack.Push(tree);
            }
            else
            {
                Tree top = (Tree)stack.Pop();
                for (int i = top.Degree - 1; i >= 0; --i)
                {
                    Tree subtree = (Tree)top.GetSubtree(i);
                    if (!subtree.IsEmpty)
                        stack.Push(subtree);
                }
                return !stack.IsEmpty;
            }

        public object Current
        {
            get
            {
            }
        }
    }
```
The `MoveNext` method advances the enumerator to the next item and returns `true` as long as there are still more items in the container. If the stack is empty, the enumeration has not yet begun. In this case, the `MoveNext` method pushes the root node of the tree onto the stack (provided the tree is not empty). If the stack is not empty, `MoveNext` method pops the top tree from the stack and then pushes its subtrees onto the stack (provided that they are not empty). Notice the order is important here. In a preorder traversal, the first subtree of a node is traversed before the second subtree. Therefore, the second subtree should appear in the stack below the first subtree. That is why the subtrees are pushed in reverse order. The running time for `MoveNext` is $O(d)$ where $d$ is the degree of the tree node at found at the top of the stack.
General Trees

This section outlines an implementation of general trees in the sense of Definition □. The salient features of the definition are first, that the nodes of a general tree have arbitrary degrees; and second, that there is no such thing as an empty tree.

The recursive nature of Definition □ has important implications when considering the implementation of such trees as containers. In effect, since a tree contains zero or more subtrees, when implemented as a container, we get a container which contains other containers!

Figure □ shows the approach we have chosen for implementing general trees. This figure shows how the general tree $T_c$ in Figure □ can be stored in memory. The basic idea is that each node has associated with it a linked list of the subtrees of that node. A linked list is used because there is no a priori restriction on its length. This allows each node to have an arbitrary degree. Furthermore, since there are no empty trees, we need not worry about representing them. An important consequence of this is that the implementation never makes use of the null reference!
Figure: Representing general trees using linked lists.

Program introduces the GeneralTree class which is used to represent general trees as specified by Definition. The GeneralTree class extends the AbstractTree class introduced in Program.

```java
public class GeneralTree : AbstractTree
{
    protected object key;
    protected int degree;
    protected LinkedList list;

    // ...
}
```

Program: GeneralTree fields.

- Fields
- Constructor and Purge Methods
- Key Property and GetSubtree Method
- AttachSubtree and DetachSubtree Methods

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Fields

The `GeneralTree` class definition comprises three fields--key, degree, and list. The first, key, represents the root node of the tree. The second, an integer degree, records the degree of the root node of the tree. The third, list, is an instance of the `LinkedList` class defined in Chapter 4. It is used to contain the subtrees of the given tree.
**Constructor and Purge Methods**

Program defines the GeneralTree constructor and Purge methods. According to Definition, a general tree must contain at least one node—an empty tree is not allowed. Therefore, the constructor takes one argument, any object instance. The constructor initializes the fields as follows: The key field is assigned the argument; the degree field is set to zero; and, the list field is assigned an empty linked list. The running time of the constructor is clearly $O(1)$.

```csharp
public class GeneralTree : AbstractTree
{
    protected object key;
    protected int degree;
    protected LinkedList list;

    public GeneralTree(object key)
    {
        this.key = key;
        degree = 0;
        list = new LinkedList();
    }

    public override void Purge()
    {
        list.Purge();
        degree = 0;
    }
    // ...
}
```

**Program**: GeneralTree class constructor and Purge methods.

The Purge method of a container normally empties the container. In this case, the container is a general tree which is not allowed to be empty. Thus, the Purge method shown in Program discards the subtrees of the tree, but it does not discard the root. The running time of the Purge method is clearly
$O(1)$. 
Key Property and GetSubtree Method

Program 1 defines the various GeneralTree class methods for manipulating general trees. The Key property provides a get accessor that returns the object contained by the root node of the tree. Clearly, its running time is $O(1)$.

```csharp
public class GeneralTree : AbstractTree
{
    protected object key;
    protected int degree;
    protected LinkedList list;

    public override object Key
    {
        get { return key; }
    }

    public override Tree GetSubtree(int i)
    {
        if (i < 0 || i >= degree)
            throw new IndexOutOfRangeException();
        LinkedList.Element ptr = list.Head;
        for (int j = 0; j < i; ++j)
            ptr = ptr.Next;
        return (GeneralTree)ptr.Datum;
    }

    public void AttachSubtree(GeneralTree t)
    {
        list.Append(t);
        ++degree;
    }

    public GeneralTree DetachSubtree(GeneralTree t)
    {
        list.Extract(t);
        --degree;
    }
```
Key Property and GetSubtree Method

```c
    118 t = Extract(t);
    119    --degree;
    120    return t;
    121 }
    122    // ...
    123 }
```

**Program:** GeneralTree class Key property, GetSubtree, AttachSubtree and DetachSubtree methods.

The GetSubtree method takes as its argument an int, \( i \), which must be between 0 and \( \text{degree} - 1 \), where \( \text{degree} \) is the degree of the root node of the tree. It returns the \( i^{th} \) subtree of the given tree. The GetSubtree method simply takes \( i \) steps down the linked list and returns the appropriate subtree. Assuming that \( i \) is valid, the worst case running time for GetSubtree is \( O(d) \), where \( d = \text{degree} \) is the degree of the root node of the tree.
AttachSubtree and DetachSubtree Methods

Program also defines two methods for manipulating the subtrees of a general tree. The purpose of the AttachSubtree method is to add the specified subtree to the root of a given tree. This method takes as its argument a GeneralTree instance which is to be attached. The AttachSubtree method simply appends to the linked list a pointer to the tree to be attached and then adds one the degree variable. The running time for AttachSubtree is $O(1)$.

Similarly, the DetachSubtree method removes the specified subtree from the given tree. This method takes as its argument the GeneralTree instance which is to be removed. It removes the appropriate item from the linked list and then subtracts one from the degree variable. The running time for DetachSubtree is $O(d)$ in the worst case, where $d = \text{degree}$. 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
**N-ary Trees**

We now turn to the implementation of $N$-ary trees as given by Definition. According to this definition, an $N$-ary tree is either an empty tree or it is a tree comprised of a root and exactly $N$ subtrees. The implementation follows the design pattern established in the preceding section. Specifically, we view an $N$-ary tree as a container.

Figure illustrates the way in which $N$-ary trees can be represented. The figure gives the representation of the tertiary ($N=3$) tree

$$\{A, \{B, \emptyset, \emptyset\}, \emptyset, \emptyset\}.$$ 

The basic idea is that each node has associated with it an array of length $N$ of pointers to the subtrees of that node. An array is used because we assume that the *arity* of the tree, $N$, is known *a priori*.

---

**Figure:** Representing $N$-ary trees using pointer arrays.

Notice that we explicitly represent the empty trees. That is, a separate object instance is allocated for each empty tree. Of course, an empty tree contains neither root nor subtrees.

Program introduces the the `NaryTree` class which represents $N$-ary trees as specified by Definition
The class NaryTree extends the AbstractTree class introduced in Program ❯.

```java
public class NaryTree : AbstractTree
{
    protected object key;
    protected int degree;
    protected NaryTree[] subtree;

    // ...
}
```

Program: NaryTree fields.

- **Fields**
- **Constructors**
- **IsEmpty Property**
- **Key Property, AttachKey and DetachKey Methods**
- **GetSubtree, AttachSubtree and DetachSubtree Methods**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Fields

The implementation the NaryTree class is very similar to that of the GeneralTree class. The NaryTree class definition also comprises three fields--key, degree, and subtree. The first, key, represents the root node of the tree. The second, an integer degree, records the degree of the root node of the tree. The third, subtree, is an array of NaryTrees. This array contains the subtrees of the given tree.
The NaryTree class declares two constructors. Implementations for the two constructors are given in Program 2. The first constructor takes a single argument of type int which specifies the degree of the tree. This constructor creates an empty tree. It does so by setting the key field to null, and by setting the subtree array to null. The running time of this constructor is $O(1)$.

```csharp
public class NaryTree : AbstractTree
{
    protected object key;
    protected int degree;
    protected NaryTree[] subtree;

    public NaryTree(int degree)
    {
        key = null;
        this.degree = degree;
        subtree = null;
    }

    public NaryTree(int degree, object key)
    {
        this.key = key;
        this.degree = degree;
        subtree = new NaryTree[degree];
        for (int i = 0; i < degree; ++i)
            subtree[i] = new NaryTree(degree);
    }
    // ...
}
```

Program: NaryTree constructors.

The second constructor takes two arguments. The first specifies the degree of the tree, and the second is
Constructors

any object instance. This constructor creates a non-empty tree in which the specified object occupies
the root node. According to Definition $\square$, every internal node in an $N$-ary tree must have exactly $N$
subtrees. Therefore, this constructor creates and attaches $N$ empty subtrees to the root node. The running
time of this constructor is $O(N)$, since $N$ empty subtrees are created and constructed and the constructor
for an empty $N$-ary tree takes $O(1)$ time.
**IsEmpty Property**

The `IsEmpty` property provides a `get` accessor that indicates whether a given \(N\)-ary tree is the empty tree. The implementation of this method is given in Program [Program](#). In this implementation, the `key` field is `null` if the tree is the empty tree. Therefore, `IsEmpty` method simply tests the `key` field. Clearly, this is a constant time operation.

```csharp
public class NaryTree : AbstractTree
{
    protected object key;
    protected int degree;
    protected NaryTree[] subtree;

    public override bool IsEmpty
    { get { return key == null; } }

    public override object Key
    {
        get
        {
            if (IsEmpty)
                throw new InvalidOperationException();
            return key;
        }
    }

    public void AttachKey(object obj)
    {
        if (!IsEmpty)
            throw new InvalidOperationException();
        key = obj;
        subtree = new NaryTree[degree];
        for (int i = 0; i < degree; ++i)
            subtree[i] = new NaryTree(degree);
    }
}
```
```java
IsEmpty Property

```

```java
27  }
28  
29  
30  public object DetachKey()
31  {
32      if (!IsLeaf)
33          throw new InvalidOperationException();
34      object result = key;
35      key = null;
36      subtree = null;
37      return result;
38  }
39  // ... 
40 
```

Program: NaryTree methods.

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Key Property, AttachKey and DetachKey Methods

Program also defines three operations for manipulating the root of an $N$-ary tree. The Key property provides a get accessor that returns the object contained in the root node of the tree. Clearly, this operation is not defined for the empty tree. If the tree is not empty, the running time of this method is $O(1)$.

The purpose of AttachKey is to insert the specified object into a given $N$-ary tree at the root node. This operation is only defined for an empty tree. The AttachKey method takes as its argument an object to be inserted in the root node and assigns that object to the key field. Since the node is no longer empty, it must have exactly $N$ subtrees. Therefore, $N$ new empty subtrees are created and attached to the node. The running time is $O(N)$ since $N$ subtrees are created, and the running time of the constructor for an empty $N$-ary tree takes $O(1)$.

Finally, DetachKey is used to remove the object from the root of a tree. In order that the tree which remains still conforms to Definition, it is only permissible to remove the root from a leaf node. And upon removal, the leaf node becomes an empty tree. The implementation given in Program throws an exception if an attempt is made to remove the root from a non-leaf node. Otherwise, the node is a leaf which means that its $N$ subtrees are all empty. When the root is detached, the array of subtrees is also discarded. The running time of this method is clearly $O(1)$.
GetSubtree, AttachSubtree and DetachSubtree Methods

Program defines the three methods for manipulating the subtrees of an N-ary tree. The GetSubtree method takes as its argument an int, i, which must be between 0 and N-1. It returns the i th subtree of the given tree. Note that this operation is only defined for a non-empty N-ary tree. Given that the tree is not empty, the running time is O(1).

```csharp
public class NaryTree : AbstractTree
{
    protected object key;
    protected int degree;
    protected NaryTree[] subtree;

    public override Tree GetSubtree(int i)
    {
        if (IsEmpty)
            throw new InvalidOperationException();
        return subtree[i];
    }

    public void AttachSubtree(int i, NaryTree t)
    {
        if (IsEmpty || !subtree[i].IsEmpty)
            throw new InvalidOperationException();
        subtree[i] = t;
    }

    NaryTree DetachSubtree(int i)
    {
        if (IsEmpty)
            throw new InvalidOperationException();
        NaryTree result = subtree[i];
        subtree[i] = new NaryTree(degree);
        return result;
    }
}
```
GetSubtree, AttachSubtree and DetachSubtree Methods

```cpp
27    return result;
28 }
29    /
30   . . .
31 }
```

**Program:** NaryTree methods.

The AttachSubtree method takes two arguments. The first is an integer \( i \) between 0 and \( N-1 \). The second is an NaryTree instance. The purpose of this method is to make the \( N \)-ary tree specified by the second argument become the \( i \)th subtree of the given tree. It is only possible to attach a subtree to a non-empty node and it is only possible to attach a subtree in a place occupied by an empty subtree. If none of the exceptions are thrown, the running time of this method is simply \( O(1) \).

The DetachSubtree method takes a single argument \( i \) which is an integer between 0 and \( N-1 \). This method removes the \( i \)th subtree from a given \( N \)-ary tree and returns that subtree. Of course, it is only possible to remove a subtree from a non-empty tree. Since every non-empty node must have \( N \) subtrees, when a subtree is removed it is replaced by an empty tree. Clearly, the running time is \( O(1) \) if we assume that no exceptions are thrown.
This section presents an implementation of binary trees in the sense of Definition [1]. A binary tree is essentially an $N$-ary tree where $N=2$. Therefore, it is possible to implement binary trees using the `NaryTree` class presented in the preceding section. However, because the `NaryTree` class implementation is a general implementation which can accommodate any value of $N$, it is somewhat less efficient in both time and space than an implementation which is designed specifically for the case $N=2$. Since binary trees occur quite frequently in practice, it is important to have a good implementation.

Another consequence of restricting $N$ to two is that we can talk of the left and right subtrees of a tree. Consequently the interface provided by a binary tree class is quite different from the general interface provided by an $N$-ary tree class.

Figure [2] shows how the binary tree given in Figure [3] is be represented. The basic idea is that each node of the tree contains two fields that refer to the subtrees of that node. Just as we did for $N$-ary trees, we represent explicitly the empty trees. Since an empty tree node contains neither root nor subtrees it is represented by a structure in which all the fields are `null`. 
The `BinaryTree` class is declared in Program 2. The `BinaryTree` class extends the `AbstractTree` class introduced in Program 1.

```java
public class BinaryTree : AbstractTree
{
    protected object key;
    protected BinaryTree left;
    protected BinaryTree right;

    // ...
}
```

**Program:** BinaryTree fields.
Binary Trees

- **Constructors**
- **Purge Method**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Fields

The `BinaryTree` class has three fields—`key`, `left`, and `right`. The first, `key`, represents the root node of the tree. The latter two, represent the left and right subtrees of the given tree. All three fields are `null` if the node represents the empty tree. Otherwise, the tree must have a root and two subtrees. Consequently, all three fields are non-`null` in a non-empty node.
Constructors

Program defines constructors for the BinaryTree class. The first one takes three arguments and assigns each of them to the corresponding field. Clearly the running time of this constructor is $O(1)$.

```csharp
public class BinaryTree : AbstractTree
{
    protected object key;
    protected BinaryTree left;
    protected BinaryTree right;

    public BinaryTree(
        object key, BinaryTree left, BinaryTree right)
    {
        this.key = key;
        this.left = left;
        this.right = right;
    }

    public BinaryTree() : this(null, null, null)
    {
    }

    public BinaryTree(object key) :
        this(key, new BinaryTree(), new BinaryTree())
    {
    }

    public BinaryTree Left
    {
        get
        {
            if (IsEmpty)
                throw new InvalidOperationException();
            return left;
        }
    }
```

Constructors

```java
public BinaryTree Right
{
    get
    {
        if (IsEmpty)
            throw new InvalidOperationException();
        return right;
    }
}
```

---

**Program:** BinaryTree constructors, Left and Right properties

The second constructor, the no-arg constructor, creates an empty binary tree. It simply sets all three fields to null.

The third constructor takes as its argument any object. The purpose of this constructor is to create a binary tree with the specified object as its root. Since every binary tree has exactly two subtrees, this constructor creates two empty subtrees and assigns them to the left and right fields.

Program also defines the Left and Right properties. These properties provide get accessors that return the left and right subtrees of the given tree, respectively.
Purge Method

The Purge method for the BinaryTree class is defined in Program □. The purpose of the Purge method is to make the tree empty. It does this by assigning null to all the fields. Clearly, the running time of the Purge method is \( O(1) \).

```csharp
public class BinaryTree : AbstractTree
{
    protected object key;
    protected BinaryTree left;
    protected BinaryTree right;

    public override void Purge()
    {
        key = null;
        left = null;
        right = null;
    }
    // ...
}
```

Program: BinaryTree class Purge method.
Binary Tree Traversals

Program defines the DepthFirstTraversal method of the BinaryTree class. This method supports all three tree traversal methods—preorder, inorder, and postorder. The implementation follows directly from the definitions given in Section. The traversal is implemented using recursion. That is, the method calls itself recursively to visit the subtrees of the given node. Note that the recursion terminates properly when an empty tree is encountered since the method does nothing in that case.

```csharp
public class BinaryTree : AbstractTree
{
    protected object key;
    protected BinaryTree left;
    protected BinaryTree right;

    public override void DepthFirstTraversal(PrePostVisitor visitor)
    {
        if (!IsEmpty)
        {
            visitor.PreVisit(key);
            Left.DepthFirstTraversal(visitor);
            visitor.InVisit(key);
            Right.DepthFirstTraversal(visitor);
            visitor.PostVisit(key);
        }
        // ...  
    }

Program: BinaryTree class DepthFirstTraversal method.
```

The traversal method takes as its argument any object that implements the PrePostVisitor interface.
defined in Program □. As each node is "visited" during the course of the traversal, the PreVisit, InVisit, and PostVisit methods of the visitor are invoked on the object contained in that node.
Comparing Trees

A problem which is relatively easy to solve is determining if two trees are equivalent. Two trees are *equivalent* if they both have the same topology and if the objects contained in corresponding nodes are equal. Clearly, two empty trees are equivalent. Consider two non-empty binary trees $T_A = \{R_A, T_{AL}, T_{AR}\}$ and $T_B = \{R_B, T_{BL}, T_{BR}\}$. Equivalence of trees is given by

$$T_A = T_B \iff R_A = R_B \land T_{AL} = T_{BL} \land T_{AR} = T_{BR}.$$ 

A simple, recursive algorithm suffices to test the equivalence of trees.

Since the `BinaryTree` class is ultimately derived from the `ComparableObject` class introduced in Program 4, it must provide a `CompareTo` method to compare binary trees. Recall that the `CompareTo` method is used to compare two objects, say `obj1` and `obj2` like this:

```csharp
int result = obj1.CompareTo(obj2);
```

The `CompareTo` method returns a negative number if `obj1 < obj2`, a positive number if `obj1 > obj2`, and zero if `obj1 == obj2`.

So what we need is to define a *total order* relation on binary trees. Fortunately, it is possible to define such a relation for binary trees provided that the objects contained in the nodes of the trees are drawn from a totally ordered set.

**Theorem** Consider two binary trees $T_A$ and $T_B$ and the relation $<$ given by

$$T_A < T_B \iff T_B \neq \emptyset \land (T_A = \emptyset \lor \begin{array}{l}
R_A \neq \emptyset \land (T_{AL} < T_{BL} \lor
R_A = R_B \land (T_{AL} = T_{BL} \land (T_{AR} < T_{BR}))))
\end{array}$$

where $T_A$ is either $\emptyset$ or $T_A = \{R, T_{AL}, T_{AR}\}$ and $T_B$ is $T_B = \{R, T_{BL}, T_{BR}\}$. The relation $<$ is a total order.
The proof of Theorem is straightforward albeit tedious. Essentially we need to show the following:

- For any two distinct trees $T_A$ and $T_B$, such that $T_A \neq T_B$, either $T_A < T_B$ or $T_B < T_A$.
- For any three distinct trees $T_A$, $T_B$, and $T_C$, if $T_A < T_B$ and $T_B < T_C$ then $T_A < T_C$.

The details of the proof are left as an exercise for the reader (Exercise).

Program gives an implementation of the `compareTo` method for the `BinaryTree` class. This implementation is based on the total order relation $<$ defined in Theorem. The argument of the `compareTo` method can be any `object` instance. However, normally that object will be another `BinaryTree` instance. Therefore, the cast on line 10 is normally successful.

```java
public class BinaryTree : AbstractTree
{
    protected object key;
    protected BinaryTree left;
    protected BinaryTree right;

    public override int CompareTo(object obj)
    {
        BinaryTree arg = (BinaryTree)obj;
        if (isEmpty)
            return arg.isEmpty ? 0 : -1;
        else if (arg.isEmpty)
            return 1;
        else
        {
            int result = ((IComparable)Key).CompareTo(arg.Key);
            if (result == 0)
                result = Left.CompareTo(arg.Left);
            if (result == 0)
                result = Right.CompareTo(arg.Right);
            return result;
        }
        // ... 
    }
```
Comparing Trees

Program: BinaryTree class CompareTo method.

The CompareTo method compares the two binary trees this and arg. If they are both empty trees, CompareTo returns zero. If this is empty and arg is not, CompareTo returns -1; and if arg is empty and this is not, it returns 1.

Otherwise, both trees are non-empty. In this case, CompareTo first compares their respective roots. We assume that the roots implement the IComparable interface and, therefore, we use the CompareTo method to compare them. If the roots are equal, then the left subtrees are compared. Then, if the roots and the left subtrees are equal, the right subtrees are compared.

Clearly the worst-case running occurs when comparing identical trees. Suppose there are exactly n nodes in each tree. Then, the running time of the CompareTo method is \( nT(\text{CompareTo}) + O(n) \), where \( T(\text{CompareTo}) \) is the time needed to compare the objects contained in the nodes of the trees.
Applications

Section shows how a stack can be used to compute the value of a postfix expression such as

\[
a b \div c d - e \times f.
\]  

(9.8)

Suppose instead of evaluating the expression we are interested in constructing the corresponding expression tree. Once we have an expression tree, we can use the methods described in Section to print out the expression in prefix or infix notation. Thus, we have a means for translating expressions from one notation to another.

It turns out that an expression tree can be constructed from the postfix expression relatively easily. The algorithm to do this is a modified version of the algorithm for evaluating the expression. The symbols in the postfix expression are processed from left to right as follows:

1. If the next symbol in the expression is an operand, a tree comprised of a single node labeled with that operand is pushed onto the stack.
2. If the next symbol in the expression is a binary operator, the top two trees in the stack correspond to its operands. Two trees are popped from the stack and a new tree is created which has the operator as its root and the two trees corresponding to the operands as its subtrees. Then the new tree is pushed onto the stack.

After all the symbols of the expression have been processed in this fashion, the stack will contain a single tree which is the desired expression tree. Figure illustrates the use of a stack to construct the expression tree from the postfix expression given in Equation.
Figure: Postfix to infix conversion using a stack of trees.
Implementation

Program introduces the ExpressionTree class. This class provides a static method, called ParsePostfix, which translates a postfix expression to an infix expression using the method described above. This method reads an expression from the input stream one character at a time. The expression is assumed to be a syntactically valid postfix expression comprised of single-digit numbers, single-letter variables, and the binary operators +, -, *, and /.

```csharp
public class ExpressionTree : BinaryTree
{
    public ExpressionTree(char c) : base(c)
    {
    }

    public static ExpressionTree ParsePostfix(TextReader reader)
    {
        Stack stack = new StackAsLinkedList();

        int i;
        while ((i = reader.Read()) >= 0)
        {
            char c = (char)i;
            if (Char.IsLetterOrDigit(c))
                stack.Push(new ExpressionTree(c));
            else if (c == '+' || c == '-' || c == '*' || c == '/')
            {
                ExpressionTree result = new ExpressionTree(c);
                result.AttachRight((ExpressionTree)stack.Pop());
                result.AttachLeft((ExpressionTree)stack.Pop());
                stack.Push(result);
            }
        }
        return (ExpressionTree)stack.Pop();
    }

    // ...
}
```
Program: Binary tree application--postfix to infix conversion.

Since only binary operators are allowed, the resulting expression tree is a binary tree. Consequently, the ExpressionTree class extends the BinaryTree class introduced in Program 0.

The main program loop, lines 11-23, reads characters from the input stream one at a time. If a letter or a digit is found, a new tree with the character as its root is created and pushed onto the stack (line 15). If an operator is found, a new tree is created with the operator as its root (line 18). Next, two trees are popped from the stack and attached to the new tree which is then pushed onto the stack (lines 19-21).

When the ParsePostfix method encounters the end-of-file, its main loop terminates. The resulting expression tree is popped from the stack and returned from the ParsePostfix method.

Program 0 defines the ToString method for the ExpressionTree class. This method can be used to print out the expression represented by the tree.
Implementation

Program: Binary tree application--printing infix expressions.

The ToString method constructs a string that represents the expression using a PrePostVisitor which does a depth-first traversal and accumulates its result in a string builder like this: At each non-terminal node of the expression tree, the depth-first traversal first calls PreVisit, which appends a left parenthesis to the string builder. In between the traversals of the left and right subtrees, the InVisit method is called, which appends a textual representation of the object contained within the node to the string builder. Finally, after traversing the right subtree, PostVisit appends a right parenthesis to the string builder. Given the input \( \frac{ab}{cd-e^+} \), the program constructs the expression tree as shown in Figure 2, and then forms the infix expression

\[
(((a)/(b)) + (((c)-(d))*(e))).
\]

The running time of the ParsePostfix method depends upon the number of symbols in the input. The running time for one iteration the main loop is \( O(1) \). Therefore, the time required to construct the
expression tree given \( n \) input symbols is \( O(n) \). The \texttt{DepthFirstTraversal} method visits each node of the expression tree exactly once and a constant amount of work is required to print a node. As a result, printing the infix expression is also \( O(n) \) where \( n \) is the number of input symbols.

The output expression contains all of the input symbols plus the parentheses added by the \texttt{ToString} method. It can be shown that a valid postfix expression that contains \( n \) symbols, always has \((n-1)/2\) binary operators and \((n+1)/2\) operands (Exercise \[\square\]). Hence, the expression tree contains \((n-1)/2\) non-terminal nodes and since a pair of parentheses is added for each non-terminal node in the expression tree, the output string contains \(2n-1=O(n)\) symbols altogether. Therefore, the overall running time needed to translate a postfix expression comprised of \( n \) symbols to an infix expression is \( O(n) \).
Exercises

1. For each tree shown in Figure show the order in which the nodes are visited during the following tree traversals:
   1. preorder traversal,
   2. inorder traversal (if defined),
   3. postorder traversal, and
   4. breadth-first traversal.

   ![Sample trees for Exercise](http://www.brpreiss.com/books/opus6/html/page299.html)

   Figure: Sample trees for Exercise.

2. Write a visitor that prints the nodes of a general tree in the format of Equation.

3. Derive an expression for the total space needed to represent a tree of \( n \) internal nodes using each of the following classes:
   1. GeneralTree introduced in Program,
   2. NaryTree introduced in Program, and
   3. BinaryTree introduced in Program.

4. A full node in a binary tree is a node with two non-empty subtrees. Let \( l \) be the number of leaf nodes in a binary tree. Show that the number of full nodes is \( l-1 \).

5. The generic DepthFirstTraversal method defined in Program is a recursive method. Write a non-recursive depth-first traversal method that has exactly the same effect as the recursive version.

6. Program defines a visitor that prints using infix notation the expression represented by an expression tree. Write a visitor that prints the same expression in prefix notation with the following
Exercises

format:

\[ +/\,(a, b), \star(-/\,(c, d), e) \].

7. Repeat Exercise 7, but this time write a visitor that the expression in postfix notation with the following format:

\[ ab/cd-e\star\star. \]

8. The visitor defined in Program 8 prints many redundant parentheses because it does not take into consideration the precedence of the operators. Rewrite the visitor so that it prints

\[ a/b+(c-d)\star e \]

rather than

\[ (((a)/(b))+(((c)-(d))\star(e))). \]

9. Consider postfix expressions involving only binary operators. Show that if such an expression contains \( n \) symbols, it always has \( (n-1)/2 \) operators and \( (n+1)/2 \) operands.


11. Generalize Theorem 11 so that it applies to \( N \)-ary trees.

12. Consider two binary trees, \( T_A = \{R_A, T_{AL}, T_{AR}\} \) and \( T_B = \{R_B, T_{BL}, T_{BR}\} \) and the relation \( \simeq \) given by

\[ T_A \simeq T_B \iff (T_A = \emptyset \land T_B = \emptyset) \lor \\
   ((T_A \neq \emptyset \land T_B \neq \emptyset) \land \\
   ((T_{AL} \simeq T_{BL} \land T_{AR} \simeq T_{BR}) \lor \\
   (T_{AL} \simeq T_{BR} \land T_{AR} \simeq T_{BL}))). \]

If \( T_A \simeq T_B \), the trees are said to be isomorphic. Devise an algorithm to test whether two binary trees are isomorphic. What is the running time of your algorithm?
Projects

1. Devise an algorithm to compute the height of a tree. Write an implementation of your algorithm as the `Height` method of the `AbstractTree` class introduced in Program.
2. Devise an algorithm to count the number of internal nodes in a tree. Write an implementation of your algorithm as the `get` accessor for the `Count` property of the `AbstractTree` class introduced in Program.
3. Devise an algorithm to count the number of leaves in a tree. Write an implementation of your algorithm as a method of the `AbstractTree` class introduced in Program.
4. Devise an abstract (generic) algorithm to compare trees. (See Exercise). Write an implementation of your algorithm as the `CompareTo` method of the `AbstractTree` class introduced in Program.
5. The `Enumerator` class introduced in Program does a preorder traversal of a tree.
   1. Write an enumerator class that does a postorder traversal.
   2. Write an enumerator class that does a breadth-first traversal.
   3. Write an enumerator class that does an inorder traversal. (In this case, assume that the tree is a `BinaryTree`).
6. Complete the `GeneralTree` class introduced in Program by providing suitable definitions for the following operations: `IsEmpty`, `IsLeaf`, `Degree`, and `CompareTo`. Write a test program and test your implementation.
7. Complete the `NaryTree` class introduced in Program by providing suitable definitions for the following operations: `Purge`, `IsLeaf`, `Degree`, and `CompareTo`. Write a test program and test your implementation.
8. Complete the `BinaryTree` class introduced in Program by providing suitable definitions for the following operations: `IsEmpty`, `IsLeaf`, `Degree`, `Key`, `AttachKey`, `DetachKey`, `AttachLeft`, `DetachLeft`, `AttachRight`, `DetachRight`, and `GetSubtree`. Write a test program and test your implementation.
9. Write a visitor that draws a picture of a tree on the screen.
10. Design and implement an algorithm that constructs an expression tree from an infix expression such as

\[
a/b+(c-d)*e.
\]

**Hint:** See Project.
Search Trees

In the preceding chapter we consider trees in which the relative positions of the nodes in the tree are unconstrained. In other words, a given item may appear anywhere in the tree. Clearly, this allows us complete flexibility in the kind of tree that we may construct. And depending on the application, this may be precisely what we need. However, if we lose track of an item, in order to find it again it may be necessary to do a complete traversal of the tree (in the worst case).

In this chapter we consider trees that are designed to support efficient search operations. In order to make it easier to search, we constrain the relative positions of the items in the tree. In addition, we show that by constraining the shape of the tree as well as the relative positions of the items in the tree, search operations can be made even more efficient.
Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
A tree which supports efficient search, insertion, and withdrawal operations is called a search tree. In this context the tree is used to store a finite set of keys drawn from a totally ordered set of keys $K$. Each node of the tree contains one or more keys and all the keys in the tree are unique, i.e., no duplicate keys are permitted.

What makes a tree into a search tree is that the keys do not appear in arbitrary nodes of the tree. Instead, there is a data ordering criterion which determines where a given key may appear in the tree in relation to the other keys in that tree. The following sections present two related types of search trees, $M$-way search trees and binary search trees.

- $M$-Way Search Trees
- Binary Search Trees
**M-Way Search Trees**

**Definition (M-way Search Tree)** An *M-way search tree* \( T \) is a finite set of keys. Either the set is empty, \( T = \emptyset \); or the set consists of \( n \) \( M \)-way subtrees \( T_0, T_1, ..., T_{n-1} \), and \( n-1 \) keys, \( k_1, k_2, ..., k_{n-1} \).

\[ T = \{ T_0, k_1, T_1, k_2, T_2, ..., k_{n-1}, T_{n-1} \}, \]

where \( 2 \leq n \leq M \), such that the keys and nodes satisfy the following *data ordering properties*:

1. The keys in each node are distinct and ordered, i.e., \( k_i < k_{i+1} \) for \( 1 \leq i \leq n-1 \).
2. All the keys contained in subtree \( T_{i-1} \) are less than \( k_i \). The tree \( T_{i-1} \) is called the *left subtree* with respect to the key \( k_i \).
3. All the keys contained in subtree \( T_i \) are greater than \( k_i \). The tree \( T_{i+1} \) is called the *right subtree* with respect to the key \( k_i \).

Figure □ gives an example of an *M-way search tree* for \( M=4 \). In this case, each of the non-empty nodes of the tree has between one and three keys and at most four subtrees. All the keys in the tree satisfy the data ordering properties. Specifically, the keys in each node are ordered and for each key in the tree, all the keys in the left subtree with respect to the given key are less than the given key, and all the keys in the right subtree with respect to the given key are larger than the given key. Finally, it is important to note that the topology of the tree is not determined by the particular set of keys it contains.
Figure: An $M$-way search tree.
Binary Search Trees

Just as the binary tree is an important category of $N$-ary trees, the *binary search tree* is an important category of $M$-way search trees.

**Definition (Binary Search Tree)** A *binary search tree* $T$ is a finite set of keys. Either the set is empty, $T = \emptyset$, or the set consists of a root $r$ and exactly two binary search trees $T_L$ and $T_R$, $T = \{r, T_L, T_R\}$, such that the following properties are satisfied:

1. All the keys contained in left subtree, $T_L$, are less than $r$.
2. All the keys contained in the right subtree, $T_R$, are greater than $r$.

Figure shows an example of a binary search tree. In this case, since the nodes of the tree carry alphabetic rather than numeric keys, the ordering of the keys is alphabetic. That is, all the keys in the left subtree of a given node precede alphabetically the root of that node, and all the keys in the right subtree of a given node follow alphabetically the root of that node. The empty trees are shown explicitly as boxes in Figure. However, in order to simplify the graphical representation, the empty trees are often omitted from the diagrams.

![Binary Search Tree Diagram](http://www.brpreiss.com/books/opus6/html/page304.html)
Searching a Search Tree

The main advantage of a search tree is that the data ordering criterion ensures that it is not necessary to do a complete tree traversal in order to locate a given item. Since search trees are defined recursively, it is easy to define a recursive search method.

- Searching an $M$-way Tree
- Searching a Binary Tree
Searching an $M$-way Tree

Consider the search for a particular item, say $x$, in an $M$-way search tree. The search always begins at the root. If the tree is empty, the search fails. Otherwise, the keys contained in the root node are examined to determine if the object of the search is present. If it is, the search terminates successfully. If it is not, there are three possibilities: Either the object of the search, $x$, is less than $k_1$, in which case subtree $T_0$ is searched; or $x$ is greater than $k_{n-1}$, in which case subtree $T_{n-1}$ is searched; or there exists an $i$ such that $1 \leq i < n - 1$ for which $k_i < x < k_{i+1}$, in which case subtree $T_i$ is searched.

Notice that when $x$ is not found in a given node, only one of the $n$ subtrees of that node is searched. Therefore, a complete tree traversal is not required. A successful search begins at the root and traces a downward path in the tree, which terminates at the node containing the object of the search. Clearly, the running time of a successful search is determined by the depth in the tree of object of the search.

When the object of the search is not in the search tree, the search method described above traces a downward path from the root which terminates when an empty subtree is encountered. In the worst case, the search path passes through the deepest leaf node. Therefore, the worst-case running time for an unsuccessful search is determined by the height of the search tree.
Searching a Binary Tree

The search method described above applies directly to binary search trees. As above, the search begins at the root node of the tree. If the object of the search, $x$, matches the root $r$, the search terminates successfully. If it does not, then if $x$ is less than $r$, the left subtree is searched; otherwise $x$ must be greater than $r$, in which case the right subtree is searched.

Figure shows two binary search trees. The tree $T_{\alpha}$ is an example of a particularly bad search tree because it is not really very tree-like at all. In fact, it is topologically isomorphic with a linear, linked list. In the worst case, a tree which contains $n$ items has height $O(n)$. Therefore, in the worst case an unsuccessful search must visit $O(n)$ internal nodes.

Figure: Examples of search trees.

On the other hand, tree $T_{\beta}$ in Figure is an example of a particularly good binary search tree. This tree is an instance of a perfect binary tree.
Definition (Perfect Binary Tree) A *perfect binary tree* of height \( h \geq 0 \) is a binary tree \( T = \{ r, T_L, T_R \} \) with the following properties:

1. If \( h=0 \), then \( T_L = \emptyset \) and \( T_R = \emptyset \).
2. Otherwise, \( h>0 \), in which case both \( T_L \) and \( T_R \) are both perfect binary trees of height \( h-1 \).

It is fairly easy to show that a perfect binary tree of height \( h \) has exactly \( 2^{h+1} - 1 \) internal nodes. Conversely, the height of a perfect binary tree with \( n \) internal nodes is \( \log_2(n + 1) \). If we have a search tree that has the shape of a perfect binary tree, then every unsuccessful search visits exactly \( h+1 \) internal nodes, where \( h = \log_2(n + 1) \). Thus, the worst case for unsuccessful search in a perfect tree is \( O(\log n) \). 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Average Case Analysis

- Successful Search
- Solving The Recurrence-Telescoping
- Unsuccessful Search
- Traversing a Search Tree

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Successful Search

When a search is successful, exactly $d+1$ internal nodes are visited, where $d$ is the depth in the tree of object of the search. For example, if the object of the search is at the root which has depth zero, the search visits just one node—the root itself. Similarly, if the object of the search is at depth one, two nodes are visited, and so on. We shall assume that it is equally likely for the object of the search to appear in any node of the search tree. In that case, the average number of nodes visited during a successful search is $\bar{d} + 1$, where $\bar{d}$ is the average of the depths of the nodes in a given tree. That is, given a binary search tree with $n > 0$ nodes,

$$\bar{d} = \frac{1}{n} \sum_{i=1}^{n} d_i,$$

where $d_i$ is the depth of the $i^{th}$ node of the tree.

The quantity $\sum_{i=1}^{n} d_i$ is called the internal path length. The internal path length of a tree is simply the sum of the depths (levels) of all the internal nodes in the tree. Clearly, the average depth of an internal node is equal to the internal path length divided by $n$, the number of nodes in the tree.

Unfortunately, for any given number of nodes $n$, there are many different possible search trees. Furthermore, the internal path lengths of the various possibilities are not equal. Therefore, to compute the average depth of a node in a tree with $n$ nodes, we must consider all possible trees with $n$ nodes. In the absence of any contrary information, we shall assume that all trees having $n$ nodes are equiprobable and then compute the average depth of a node in the average tree containing $n$ nodes.

Let $I(n)$ be the average internal path length of a tree containing $n$ nodes. Consider first the case of $n=1$. Clearly, there is only one binary tree that contains one node—the tree of height zero. Therefore, $I(1)=0$.

Now consider an arbitrary tree, $T_n(l)$, having $n \geq 1$ internal nodes altogether, $l$ of which are found in its left subtree, where $0 \leq l < n$. Such a tree consists of a root, the left subtree with $l$ internal nodes and and a right subtree with $n-l-1$ internal nodes. The average internal path length for such a tree is the sum of the average internal path length of the left subtree, $I(l)$, plus that of the right subtree, $I(n-l-1)$, plus $n-1$ because the nodes in the two subtrees are one level lower in $T_n(l)$.
In order to determine the average internal path length for a tree with \( n \) nodes, we must compute the average of the internal path lengths of the trees \( T_n(l) \) averaged over all possible sizes, \( l \), of the (left) subtree, \( 0 \leq l < n \).

To do this we consider an ordered set of \( n \) distinct keys, \( k_0 < k_1 < \cdots < k_{n-1} \). If we select the \( l \)th key, \( k_l \), to be the root of a binary search tree, then there are \( l \) keys, \( k_0, k_1, \ldots, k_{l-1} \), in its left subtree and \( n-l-1 \) keys, \( k_{l+1}, k_{l+2}, \ldots, k_{n-1} \), in its right subtree.

If we assume that it is equally likely for any of the \( n \) keys to be selected as the root, then all the subtree sizes in the range \( 0 \leq l < n \) are equally likely. Therefore, the average internal path length for a tree with \( n \geq 1 \) nodes is

\[
I(n) = \frac{1}{n} \sum_{i=0}^{n-1} \left( I(i) + I(n - i - 1) + n - 1 \right), \quad n > 1
\]

Thus, in order to determine \( I(n) \) we need to solve the recurrence

\[
I(n) = \begin{cases} 
0 & n = 1, \\
\frac{2}{n} \sum_{i=0}^{n-1} I(i) + n - 1 & n > 1.
\end{cases} \tag{10.1}
\]

To solve this recurrence we consider the case \( n > 1 \) and then multiply Equation 10.1 by \( n \) to get

\[
nI(n) = 2 \sum_{i=0}^{n-1} I(i) + n^2 - n. \tag{10.2}
\]

Since this equation is valid for any \( n > 1 \), by substituting \( n-1 \) for \( n \) we can also write

\[
(n - 1)I(n - 1) = 2 \sum_{i=0}^{n-2} I(i) + n^2 - 3n + 2, \tag{10.3}
\]

which is valid for \( n > 2 \). Subtracting Equation 10.2 from Equation 10.3 gives
\[ nI(n) - (n - 1)I(n - 1) = 2I(n - 1) + 2n - 2, \]

which can be rewritten as

\[ I(n) = \frac{(n + 1)I(n - 1) + 2n - 2}{n}. \]  \hspace{1cm} (10.4)

Thus, we have shown the solution to the recurrence in Equation 10.4 is the same as the solution of the recurrence

\[
I(n) = \begin{cases} 
0 & n = 1, \\
1 & n = 2, \\
\frac{(n + 1)I(n - 1) + 2n - 2}{n} & n > 2.
\end{cases}
\]  \hspace{1cm} (10.5)
Solving The Recurrence-Telescoping

This section presents a technique for solving recurrence relations such as Equation 10.5 called *telescoping*. The basic idea is this: We rewrite the recurrence formula so that a similar functional form appears on both sides of the equal sign. For example, in this case, we consider \( n > 2 \) and divide both sides of Equation 10.5 by \( n+1 \) to get

\[
\frac{I(n)}{n+1} = \frac{I(n-1)}{n} + \frac{2}{n} - \frac{4}{n(n+1)}.
\]

Since this equation is valid for any \( n > 2 \), we can write the following series of equations:

\[
\begin{align*}
\frac{I(n)}{n+1} &= \frac{I(n-1)}{n} + \frac{2}{n} - \frac{4}{n(n+1)}, \quad n > 2 \\
\frac{I(n-1)}{n} &= \frac{I(n-2)}{n-1} + \frac{2}{n-1} - \frac{4}{(n-1)n}, \quad n-1 > 2 \\
\frac{I(n-2)}{n-1} &= \frac{I(n-3)}{n-2} + \frac{2}{n-2} - \frac{4}{(n-2)(n-1)}, \quad n-2 > 2 \\
&\vdots \\
\frac{I(n-k)}{n-k+1} &= \frac{I(n-k-1)}{n-k} + \frac{2}{n-k} - \frac{4}{(n-k)(n-k+1)}, \quad n-k > 2 \\
&\vdots \\
\frac{I(3)}{4} &= \frac{I(2)}{3} + \frac{2}{3} - \frac{4}{3\cdot 4} \tag{10.7}
\end{align*}
\]

Each subsequent equation in this series is obtained by substituting \( n-1 \) for \( n \) in the preceding equation. In principle, we repeat this substitution until we get an expression on the right-hand-side involving the base case. In this example, we stop at \( n-k-1=2 \).

Because Equation 10.6 has a similar functional form on both sides of the equal sign, when we add Equation 10.6 through Equation 10.7 together, most of the terms cancel leaving
where $H_n$ is the $n$th harmonic number. In Section it is shown that $H_n \approx \ln n + \gamma$, where $\gamma \approx 0.577 215$ is called Euler's constant. Thus, we get that the average internal path length of the average binary search tree with $n$ internal nodes is

$$I(n) = 2(n + 1)H_n - 4n \approx 2(n + 1)(\ln n + \gamma) - 4n.$$

Finally, we get to the point: The average depth of a node in the average binary search tree with $n$ nodes is

$$\bar{d} = \frac{I(n)}{n} = 2 \left( \frac{n + 1}{n} \right) H_n - 4 \approx 2 \left( \frac{n + 1}{n} \right) (\ln n + \gamma) - 4 = O(\log n).$$
Unsuccessful Search

All successful searches terminate when the object of the search is found. Therefore, all successful searches terminate at an internal node. In contrast, all unsuccessful searches terminate at an external node. In terms of the binary tree shown in Figure □, a successful search terminates in one of the nodes which are drawn as a circles and an unsuccessful search terminates in one of the boxes.

The preceding analysis shows that the average number of nodes visited during a successful search depends on the internal path length, which is simply the sum of the depths of all the internal nodes. Similarly, the average number of nodes visited during an unsuccessful search depends on the external path length, which is the sum of the depths of all the external nodes. Fortunately, there is a simple relationship between the internal path length and the external path length of a binary tree.

**Theorem** Consider a binary tree $T$ with $n$ internal nodes and an internal path length of $I$. The external path length of $T$ is given by

$$E = I + 2n.$$ 

In other words, Theorem □ says that the difference between the internal path length and the external path length of a binary tree with $n$ internal nodes is $E-I=2n$.

**Proof (By induction).**

**Base Case** Consider a binary tree with one internal node and internal path length of zero. Such a tree has exactly two empty subtrees immediately below the root and its external path length is two. Therefore, the theorem holds for $n=1$.

**Inductive Hypothesis** Assume that the theorem holds for $n = 1, 2, 3, \ldots, k$ for some $k \geq 1$. Consider an arbitrary tree, $T_k$, that has $k$ internal nodes. According to Theorem □, $T_k$ has $k+1$ external nodes. Let $I_k$ and $E_k$ be the internal and external path length of $T_k$, respectively, According to the inductive hypothesis, $E_k - I_k = 2k$.

Consider what happens when we create a new tree $T_{k+1}$ by removing an external node from $T_k$ and replacing it with an internal node that has two empty subtrees. Clearly, the resulting tree has $k+1$ internal
Unsuccessful Search

Furthermore, suppose the external node we remove is at depth $d$. Then the internal path length of $T_{k+1}$ is $I_{k+1} = I_k + d$ and the external path length of $T_{k+1}$ is

$$E_{k+1} = E_k - d + 2(d+1) = E_k + d + 2.$$  

The difference between the internal path length and the external path length of $T_{k+1}$ is

$$E_{k+1} - I_{k+1} = (E_k + d + 2) - (I_k + d)$$

$$= E_k - I_k + 2$$

$$= 2(k + 1).$$

Therefore, by induction on $k$, the difference between the internal path length and the external path length of a binary tree with $n$ internal nodes is $2n$ for all $n \geq 1$.

Since the difference between the internal and external path lengths of any tree with $n$ internal nodes is $2n$, then we can say the same thing about the average internal and external path lengths averaged over all search trees. Therefore, $E(n)$, the average external path length of a binary search tree is given by

$$E(n) = I(n) + 2n$$

$$= 2(n + 1)H_n - 2n$$

$$\approx 2(n + 1)(\ln n + \gamma) - 2n.$$

A binary search tree with internal $n$ nodes has $n+1$ external nodes. Thus, the average depth of an external node of a binary search tree with $n$ internal nodes, $\bar{e}$, is given by

$$\bar{e} = E(n)/(n + 1)$$

$$= 2H_n - 2n/(n + 1)$$

$$\approx 2(\ln n + \gamma) - 2n/(n + 1)$$

$$= O(\log n).$$

These very nice results are the raison d'être for binary search trees. What they say is that the average number of nodes visited during either a successful or an unsuccessful search in the average binary search tree having $n$ nodes is $O(\log n)$. We must remember, however, that these results are premised on the assumption that all possible search trees of $n$ nodes are equiprobable. It is important to be aware that in practice this may not always be the case.
Traversing a Search Tree

In Section □, the inorder traversal of a binary tree is defined as follows:

1. Traverse the left subtree; and then
2. visit the root; and then
3. traverse the right subtree.

It should not come as a surprise that when an inorder traversal of a binary search tree is done, the nodes of the tree are visited in order!

In an inorder traversal the root of the tree is visited after the entire left subtree has been traversed and in a binary search tree everything in the left subtree is less than the root. Therefore, the root is visited only after all the keys less than the root have been visited.

Similarly, in an inorder traversal the root is visited before the right subtree is traversed and everything in the right subtree is greater than the root. Hence, the root is visited before all the keys greater than the root are visited. Therefore, by induction, the keys in the search tree are visited in order.

Inorder traversal is not defined for arbitrary N-ary trees--it is only defined for the case of N=2. Essentially this is because the nodes of N-ary trees contain only a single key. On the other hand, if a node of an M-way search tree has n subtrees, then it must contain n-1 keys, such that \(2 < n \leq M\). Therefore, we can define inorder traversal of an M-way tree as follows:

To traverse a node of an M-way tree having n subtrees,

\[
\text{Traverse } T_0; \text{ and then} \\
\text{visit } k_1; \text{ and then} \\
\text{traverse } T_1; \text{ and then} \\
\text{visit } k_2; \text{ and then} \\
\text{traverse } T_2; \text{ and then} \\
\vdots
\]
Traversing a Search Tree

2\(n-2\).

visit \(k_{n-1}\); and then

2\(n-1\).

traverse \(T_{n-1}\).
Implementing Search Trees

Since search trees are designed to support efficient searching, it is only appropriate that they implement the `SearchableContainer` interface. Recall from Section [that the searchable container interface includes the operations `Find`, `IsMember`, `Insert`, and `Withdraw`.

**Figure:** Object class hierarchy

Program defines the `SearchTree` interface. The `SearchTree` interface extends the `Tree` interface defined in Program and the `SearchableContainer` interface defined in Program.

```csharp
public interface SearchTree : Tree, SearchableContainer
{
    ComparableObject Min { get; }
    ComparableObject Max { get; }
}
```

**Program:** `SearchTree` interface.

In addition, two properties are defined—Min and Max. The Min property provides a `get` accessor that returns the object contained in the search tree having the smallest key. Similarly, the Max provides a `get` accessor that returns the contained object having the largest key.
Implementing Search Trees

- Binary Search Trees
- Inserting Items in a Binary Search Tree
- Removing Items from a Binary Search Tree

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Binary Search Trees

The class `BinarySearchTree` introduced in Program represents binary search trees. Since binary trees and binary search trees are topologically similar, the `BinarySearchTree` class extends the `BinaryTree` introduced in Program. In addition, because it represents search trees, the `BinarySearchTree` class implements the `SearchTree` interface defined in Program.

```csharp
public class BinarySearchTree : BinaryTree, SearchTree
{
    // ...
}
```

**Program:** BinarySearchTree class.

- **Fields**
- **Find Method**
- **Min Property**
Fields

The BinarySearchTree class inherits the three fields key, left, and right from the BinaryTree class. The first refers to any object instance, and the latter two are BinaryTree instances which are the subtrees of the given tree. All three fields are null if the node represents the empty tree. Otherwise, the tree must have a root and two subtrees. Therefore, all three fields are non-null in an internal node.

Program defines the three properties Key, Left, and Right which access key, and the left and right subtrees, respectively, of a given binary search tree. In the BinaryTree class the left and right fields are BinaryTree instances. However, in a binary search tree, the subtrees will be instances of the BinarySearchTree class. The Left and Right accessors cast the left and right fields to the appropriate type. Similarly, the Key accessor casts the key field to a ComparableObject.

```csharp
public class BinarySearchTree : BinaryTree, SearchTree
{
    new public virtual ComparableObject Key
    { get { return (ComparableObject)base.Key; } }

    new public virtual BinarySearchTree Left
    { get { return (BinarySearchTree)base.Left; } }

    new public BinarySearchTree Right
    { get { return (BinarySearchTree)base.Right; } }
    // ...
}
```

Program: BinarySearchTree class Key, Left and Right properties.
Find Method

Program □ gives the code for the `Find` method of the `BinarySearchTree` class. The `Find` method takes as its argument any `ComparableObject`. The purpose of the method is to search the tree for an object which matches the argument. If a match is found, `Find` returns the matching object. Otherwise, `Find` returns `null`.

```csharp
public class BinarySearchTree : BinaryTree, SearchTree
{
    public virtual ComparableObject Find(ComparableObject obj)
    {
        if (IsEmpty)
            return null;
        int diff = obj.CompareTo(Key);
        if (diff == 0)
            return Key;
        else if (diff < 0)
            return Left.Find(obj);
        else
            return Right.Find(obj);
    }

    public virtual ComparableObject Min
    {
        get
        {
            if (IsEmpty)
                return null;
            else if (Left.IsEmpty)
                return Key;
            else
                return Left.Min;
        }
    }
}
```
Find Method

```
27         
28         // ... 
29     }
```

**Program:** BinarySearchTree class Find and Min methods.

The recursive **Find** method starts its search at the root and descends one level in the tree for each recursive call. At each level at most one object comparison is made (line 7). The worst case running time for a search is

\[ nT\langle \text{compareTo}\rangle + O(n), \]

where \( T\langle \text{compareTo}\rangle \) is the time to compare two objects and \( n \) is the number of internal nodes in the tree. The same asymptotic running time applies for both successful and unsuccessful searches.

The average running time for a successful search is \((\bar{d} + 1)T\langle \text{compareTo}\rangle + O(\bar{d})\), where

\[ \bar{d} = 2(n + 1)H_n/n - 4 \]

is the average depth of an internal node in a binary search tree. If \( T\langle \text{compareTo}\rangle = O(1) \), the average time of a successful search is \( O(\log n) \).

The average running time for an unsuccessful search is \( \bar{e}T\langle \text{compareTo}\rangle + O(\bar{e}) \), where

\[ \bar{e} = 2H_n - 4n/(n + 1) \]

is the average depth of an external node in a binary search tree. If \( T\langle \text{compareTo}\rangle = O(1) \), the average time of an unsuccessful search is \( O(\log n) \).
**Min Property**

Program also shows a recursive implementation of the `Min` property `get` accessor of the `BinarySearchTree` class. It follows directly from the data ordering property of search trees that to find the node containing the smallest key in the tree, we start at the root and follow the chain of left subtrees until we get to the node that has an empty left subtree. The key in that node is the smallest in the tree. Notice that no object comparisons are necessary to identify the smallest key in the tree.

The running time analysis of the accessor follows directly from that of the `Find` method. The worst case running time of `Min` is $O(n)$ and the average running time is $O(\log n)$, where $n$ is the number of internal nodes in the tree.
The simplest way to insert an item into a binary search tree is to pretend that the item is already in the tree and then follow the path taken by the `Find` method to determine where the item would be. Assuming that the item is not already in the tree, the search will be unsuccessful and will terminate at an external, empty node. That is precisely where the item to be inserted is placed!

- **Insert and AttachKey Methods**
Insert and AttachKey Methods

The Insert method of the BinarySearchTree class is defined in Program 319. This method takes as its argument the object which is to be inserted into the binary search tree. It is assumed in this implementation that duplicate keys are not permitted. That is, all of the keys contained in the tree are unique.

```csharp
public class BinarySearchTree : BinaryTree, SearchTree
{
    public virtual void Insert(ComparableObject obj)
    {
        if (IsEmpty)
            AttachKey(obj);
        else
        {
            int diff = obj.CompareTo(Key);
            if (diff == 0)
                throw new ArgumentException("duplicate key");
            if (diff < 0)
                Left.Insert(obj);
            else
                Right.Insert(obj);
        }
        Balance();
    }

    public override void AttachKey(object obj)
    {
        if (!IsEmpty)
            throw new InvalidOperationException();
        key = obj;
        left = new BinarySearchTree();
        right = new BinarySearchTree();
    }
```
The `Insert` method behaves like the `Find` method until it arrives at an external, empty node. Once the empty node has been found, it is transformed into an internal node by calling the `AttachKey` method. `AttachKey` works as follows: The object being inserted is assigned to the `key` field and two new empty binary trees are attached to the node.

Notice that after the insertion is done, the `Balance` method is called. However, as shown in Program 27, the `BinarySearchTree.Balance` method does nothing. (Section 3 describes the class `AVLTree` which is derived from the `BinarySearchTree` class and which inherits the `Insert` method but overrides the `Balance` operation).

The asymptotic running time of the `Insert` method is the same as that of `Find` for an unsuccessful search. That is, in the worst case the running time is $n T(\text{compareTo}) + O(n)$ and the average case running time is

$$
\bar{e} T(\text{compareTo}) + O(\bar{e}),
$$

where $\bar{e} = 2H_n - 2n/(n + 1)$ is the average depth of an external node in a binary search tree with $n$ internal nodes. When $T(\text{compareTo}) = O(1)$, the worst case running time is $O(n)$ and the average case is $O(\log n)$. 
Removing Items from a Binary Search Tree

When removing an item from a search tree, it is imperative that the tree which remains satisfies the data ordering criterion. If the item to be removed is in a leaf node, then it is fairly easy to remove that item from the tree since doing so does not disturb the relative order of any of the other items in the tree.

For example, consider the binary search tree shown in Figure (a). Suppose we wish to remove the node labeled 4. Since node 4 is a leaf, its subtrees are empty. When we remove it from the tree, the tree remains a valid search tree as shown in Figure (b).

![Diagram](http://www.brpreiss.com/books/opus6/html/page320.html)

**Figure:** Removing a leaf node from a binary search tree.

To remove a non-leaf node, we move it down in the tree until it becomes a leaf node since a leaf node is easily deleted. To move a node down we swap it with another node which is further down in the tree. For example, consider the search tree shown in Figure (a). Node 1 is not a leaf since it has an empty left subtree but a non-empty right subtree. To remove node 1, we swap it with the smallest key in its right subtree, which in this case is node 2, Figure (b). Since node 1 is now a leaf, it is easily deleted. Notice that the resulting tree remains a valid search tree, as shown in Figure (c).
To move a non-leaf node down in the tree, we either swap it with the smallest key in the right subtree or with the largest one in the left subtree. At least one such swap is always possible, since the node is a non-leaf and therefore at least one of its subtrees is non-empty. If after the swap, the node to be deleted is not a leaf, the we push it further down the tree with yet another swap. Eventually, the node must reach the bottom of the tree where it can be deleted.
Withdraw Method

Program gives the code for the Withdraw method of the BinarySearchTree class. The Withdraw method takes as its argument the object instance to be removed from the tree. The algorithm first determines the location of the object to be removed and then removes it according to the procedure described above.

```csharp
public class BinarySearchTree : BinaryTree, SearchTree
{
    public virtual void Withdraw(ComparableObject obj)
    {
        if (IsEmpty)
            throw new ArgumentException("object not found");
        int diff = obj.CompareTo(Key);
        if (diff == 0)
            { if (!Left.IsEmpty) { ComparableObject max = Left.Max;
                        key = max;
                        Left.Withdraw(max); }
            else if (!Right.IsEmpty) { ComparableObject min = Right.Min;
                                        key = min;
                                        Right.Withdraw(min); }
            else
                DetachKey(); }
        else if (diff < 0)
            Left.Withdraw(obj);
        else
            Right.Withdraw(obj);
    }
```
Program: BinarySearchTree class Withdraw method.

```c++
    else
        Right.Withdraw(obj);
    Balance();
    // ...
    }
```
AVL Search Trees

The problem with binary search trees is that while the average running times for search, insertion, and withdrawal operations are all $O(\log n)$, any one operation is still $O(n)$ in the worst case. This is so because we cannot say anything in general about the shape of the tree.

For example, consider the two binary search trees shown Figure 1. Both trees contain the same set of keys. The tree $T_a$ is obtained by starting with an empty tree and inserting the keys in the following order

$$1, 2, 3, 4, 5, 6, 7.$$

The tree $T_b$ is obtained by starting with an empty tree and inserting the keys in this order

$$4, 2, 6, 1, 3, 5, 7.$$

Clearly, $T_b$ is a better tree search tree than $T_a$. In fact, since $T_b$ is a perfect binary tree, its height is $\log_2(n + 1) - 1$. Therefore, all three operations, search, insertion, and withdrawal, have the same worst case asymptotic running time, $O(\log n)$.

The reason that $T_b$ is better than $T_a$ is that it is the more balanced tree. If we could ensure that the search trees we construct are balanced then the worst-case running time of search, insertion, and withdrawal, could be made logarithmic rather than linear. But under what conditions is a tree balanced?

If we say that a binary tree is balanced if the left and right subtrees of every node have the same height, then the only trees which are balanced are the perfect binary trees. A perfect binary tree of height $h$ has exactly $2^{h+1} - 1$ internal nodes. Therefore, it is only possible to create perfect trees with $n$ nodes for $n = 1, 3, 7, 15, 31, 63, \ldots$. Clearly, this is an unsuitable balance condition because it is not possible to create a balanced tree for every $n$.

What are the characteristics of a good balance condition?
1. A good balance condition ensures that the height of a tree with \( n \) nodes is \( O(\log n) \).

2. A good balance condition can be maintained efficiently. That is, the additional work necessary to balance the tree when an item is inserted or deleted is \( O(1) \).

Adelson-Velskii and Landis were the first to propose the following balance condition and show that it has the desired characteristics.

**Definition (AVL Balance Condition)** An empty binary tree is **AVL balanced**. A non-empty binary tree, \( T = \{r, T_L, T_R\} \), is AVL balanced if both \( T_L \) and \( T_R \) are AVL balanced and

\[
|\hat{h}_L - \hat{h}_R| \leq 1,
\]

where \( \hat{h}_L \) is the height of \( T_L \) and \( \hat{h}_R \) is the height of \( T_R \).

Clearly, all perfect binary trees are AVL balanced. What is not so clear is that heights of all trees that satisfy the AVL balance condition are logarithmic in the number of internal nodes.

**Theorem** The height, \( h \), of an AVL balanced tree with \( n \) internal nodes satisfies

\[
\log_2(n + 1) + 1 \leq h \leq 1.440 \log(n + 2) - 0.328.
\]

**Proof** The lower bound follows directly from Theorem. It is in fact true for all binary trees regardless of whether they are AVL balanced.

To determine the upper bound, we turn the problem around and ask the question, what is the minimum number of internal nodes in an AVL balanced tree of height \( h \)?

Let \( \mathcal{T}_h \) represent an AVL balanced tree of height \( h \) which has the smallest possible number of internal nodes, say \( N_h \). Clearly, \( \mathcal{T}_h \) must have at least one subtree of height \( h-1 \) and that subtree must be \( \mathcal{T}_{h-1} \). To remain AVL balanced, the other subtree can have height \( h-1 \) or \( h-2 \). Since we want the smallest number of internal nodes, it must be \( \mathcal{T}_{h-2} \). Therefore, the number of internal nodes in \( \mathcal{T}_h \) is

\[
N_h = N_{h-1} + N_{h-2} + 1, \quad \text{where} \quad h \geq 2.
\]

Clearly, \( \mathcal{T}_0 \) contains a single internal node, so \( N_0 = 1 \). Similarly, \( \mathcal{T}_1 \) contains exactly two nodes, so \( N_1 = 2 \). Thus, \( N_h \) is given by the recurrence
The remarkable thing about Equation (10.8) is its similarity with the definition of Fibonacci numbers (Equation (10.5)). In fact, it can easily be shown by induction that

\[ N_h \geq F_{h+2} - 1 \]

for all \( h \geq 0 \), where \( F_k \) is the \( k \)th Fibonacci number.

**Base Cases**

\[ N_0 = 1, \quad F_2 = 1 \quad \implies \quad N_0 \geq F_2 - 1, \]
\[ N_1 = 2, \quad F_3 = 2 \quad \implies \quad N_1 \geq F_3 - 1. \]

**Inductive Hypothesis** Assume that \( N_h \geq F_{h+2} - 1 \) for \( h = 0, 1, 2, \ldots, k \). Then

\[
N_{h+1} = N_h + N_{h-1} + 1 \\
\geq F_{h+2} - 1 + F_{h+1} - 1 + 1 \\
\geq F_{h+3} - 1 \\
\geq F_{(h+1)+2} - 1.
\]

Therefore, by induction on \( k \), \( N_h \geq F_{h+2} - 1 \), for all \( h \geq 0 \).

According to Theorem (10.7), the Fibonacci numbers are given by

\[ F_n = \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n), \]

where \( \phi = (1 + \sqrt{5})/2 \) and \( \hat{\phi} = (1 - \sqrt{5})/2 \). Furthermore, since \( \hat{\phi} \approx -0.618, |\hat{\phi}^n / \sqrt{5}| < 1 \).

Therefore,
\[ N_h \geq F_{h+2} - 1 \quad \Rightarrow \quad N_h \geq \phi^{h+2} / \sqrt{5} - 2 \]
\[ \Rightarrow \quad \sqrt{5}(N_h + 2) \geq \phi^{h+2} \]
\[ \Rightarrow \quad \log_{\phi}(\sqrt{5}(N_h + 2)) \geq h + 2 \]
\[ \Rightarrow \quad h \leq \log_{\phi}(N_h + 2) + \log_{\phi} \sqrt{5} - 2 \]
\[ \Rightarrow \quad h \leq 1.440 \log_2(N_h + 2) - 0.328 \]

This completes the proof of the upper bound.

So, we have shown that the AVL balance condition satisfies the first criterion of a good balance condition--the height of an AVL balanced tree with \( n \) internal nodes is \( \Theta(\log n) \). What remains to be shown is that the balance condition can be efficiently maintained. To see that it can, we need to look at an implementation.

- Implementing AVL Trees
- Inserting Items into an AVL Tree
- Removing Items from an AVL Tree
Implementing AVL Trees

Having already implemented a binary search tree class, `BinarySearchTree`, we can make use of much of the existing code to implement an AVL tree class. Program introduces the `AVLTree` class which extends the `BinarySearchTree` class introduced in Program. The `AVLTree` class inherits most of its functionality from the binary tree class. In particular, it uses the inherited `Insert` and `Withdraw` methods. However, the inherited `Balance`, `AttachKey` and `DetachKey` methods are overridden and a number of new methods are declared.

```csharp
public class AVLTree : BinarySearchTree {
    protected int height;
    // ...
}
```

Program: AVLTree fields.

Program indicates that an additional field is added in the `AVLTree` class. This turns out to be necessary because we need to be able to determine quickly, i.e., in \(O(1)\) time, that the AVL balance condition is satisfied at a given node in the tree. In general, the running time required to compute the height of a tree containing \(n\) nodes is \(O(n)\). Therefore, to determine whether the AVL balance condition is satisfied at a given node, it is necessary to traverse completely the subtrees of the given node. But this cannot be done in constant time.

To make it possible to verify the AVL balance condition in constant time, the field `height` has been added. Thus, every node in an `AVLTree` keeps track of its own height. In this way it is possible for the `Height` property `get` accessor to run in constant time—-all it needs to do is to return the value of the `height` field. And this makes it possible to test whether the AVL balanced condition satisfied at a given node in constant time.
Implementing AVL Trees

- **Constructor**
- **AdjustHeight Method, Height and BalanceFactor Properties**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Constructor

A no-arg constructor is shown in Program 1. This constructor creates an empty AVL tree. The height field is set to the value -1, which is consistent with the empty tree. Notice that according to Definition 1, the empty tree is AVL balanced. Therefore, the result is a valid AVL tree. Clearly, the running time of the constructor is $O(1)$.

```csharp
public class AVLTree : BinarySearchTree
{
    protected int height;

    public AVLTree()
    { height = -1; }

    public override int Height
    { get { return height; } }

    protected void AdjustHeight()
    {
        if (IsEmpty)
            height = -1;
        else
            height = 1 + Math.Max(Left.Height, Right.Height);
    }

    protected int BalanceFactor
    { get
        {
            if (IsEmpty)
                return 0;
            else
                return Left.Height - Right.Height;
        }
    }
```
Program: AVLTree class constructor, AdjustHeight method, Height and BalanceFactor properties.
AdjustHeight Method, Height and BalanceFactor Properties

The Height property provides a get accessor that returns the value of the height field. Clearly the running time of this method is constant.

The purpose of AdjustHeight is to recompute the height of a node and to update the height field. This method must be called whenever the height of one of the subtrees changes in order to ensure the height field is always up to date. The AdjustHeight method determines the height of a node by adding one to the height of the highest subtree. Since the running time of the Height accessor is constant, so too is the running time of AdjustHeight.

The BalanceFactor property provides a get accessor that returns the difference between the heights of the left and right subtrees of a given AVL tree. By Definition, the empty node is AVL balanced. Therefore, the BalanceFactor is zero for an empty tree. Again, since the running time of the Height accessor is constant, the running time of BalanceFactor is also constant.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Inserting Items into an AVL Tree

Inserting an item into an AVL tree is a two-part process. First, the item is inserted into the tree using the usual method for insertion in binary search trees. After the item has been inserted, it is necessary to check that the resulting tree is still AVL balanced and to balance the tree when it is not.

Just as in a regular binary search tree, items are inserted into AVL trees by attaching them to the leaves. To find the correct leaf we pretend that the item is already in the tree and follow the path taken by the `Find` method to determine where the item should go. Assuming that the item is not already in the tree, the search is unsuccessful and terminates an an external, empty node. The item to be inserted is placed in that external node.

Inserting an item in a given external node affects potentially the heights of all the nodes along the access path, i.e., the path from the root to that node. Of course, when an item is inserted in a tree, the height of the tree may increase by one. Therefore, to ensure that the resulting tree is still AVL balanced, the heights of all the nodes along the access path must be recomputed and the AVL balance condition must be checked.

Sometimes increasing the height of a subtree does not violate the AVL balance condition. For example, consider an AVL tree $T = \{r, T_L, T_R\}$. Let $h_L$ and $h_R$ be the heights of $T_L$ and $T_R$, respectively. Since $T$ is an AVL tree, then $|h_L - h_R| \leq 1$. Now, suppose that $h_L = h_R + 1$. Then, if we insert an item into $T_R$, its height may increase by one to $h_R' = h_R + 1$. The resulting tree is still AVL balanced since $h_L - h_R' = 0$. In fact, this particular insertion actually makes the tree more balanced! Similarly if $h_L = h_R$ initially, an insertion in either subtree will not result in a violation of the balance condition at the root of $T$.

On the other hand, if $h_L = h_R + 1$ and an the insertion of an item into the left subtree $T_L$ increases the height of that tree to $h_L' = h_L + 1$, the AVL balance condition is no longer satisfied because $h_L' - h_R = 2$. Therefore it is necessary to change the structure of the tree to bring it back into balance.
Inserting Items into an AVL Tree

- Balancing AVL Trees
- Single Rotations
- Double Rotations
- Implementation

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Balancing AVL Trees

When an AVL tree becomes unbalanced, it is possible to bring it back into balance by performing an operation called a *rotation*. It turns out that there are only four cases to consider and each case has its own rotation.
Single Rotations

Figure (a) shows an AVL balanced tree. For example, the balance factor for node \( A \) is zero, since its left and right subtrees have the same height; and the balance factor of node \( B \) is +1, since its left subtree has height \( h+1 \) and its right subtree has height \( h \).
Single Rotations

Figure: Balancing an AVL tree with a single (LL) rotation.

Suppose we insert an item into $A_L$, the left subtree of $A$. The height of $A_L$ can either increase or remain the same. In this case we assume that it increases. Then, as shown in Figure (b), the resulting tree is no longer AVL balanced. Notice where the imbalance has been manifested--node $A$ is balanced but node $B$ is not.

Balance can be restored by reorganizing the two nodes $A$ and $B$, and the three subtrees, $A_L$, $A_R$, and $B_R$, as shown in Figure (c). This is called an LL rotation, because the first two edges in the insertion path from node $B$ both go to the left.

There are three important properties of the LL rotation:

1. The rotation does not destroy the data ordering property so the result is still a valid search tree. Subtree $A_L$ remains to the left of node $A$, subtree $A_R$ remains between nodes $A$ and $B$, and subtree $B_R$ remains to the right of node $B$.
2. After the rotation both $A$ and $B$ are AVL balanced. Both nodes $A$ and $B$ end up with zero balance factors.
3. After the rotation, the tree has the same height it had originally. Inserting the item did not increase the overall height of the tree!

Notice, the LL rotation was called for because the root became unbalanced with a positive balance factor (i.e., its left subtree was too high) and the left subtree of the root also had a positive balance factor.

Not surprisingly, the left-right mirror image of the LL rotation is called an RR rotation. An RR rotation is called for when the root becomes unbalanced with a negative balance factor (i.e., its right subtree is too high) and the right subtree of the root also has a negative balance factor.
The preceding cases have dealt with access paths LL and RR. Clearly two more cases remain to be implemented. Consider the case where the root becomes unbalanced with a positive balance factor but the left subtree of the root has a negative balance factor. This situation is shown in Figure (b).
Double Rotations

Figure: Balancing an AVL tree with a double (LR) rotation.

The tree can be restored by performing an RR rotation at node $A$, followed by an LL rotation at node $C$. The tree which results is shown in Figure (c). The LL and RR rotations are called single rotations. The combination of the two single rotations is called a double rotation and is given the name LR rotation because the first two edges in the insertion path from node $C$ both go left and then right.

Obviously, the left-right mirror image of the LR rotation is called an RL rotation. An RL rotation is called for when the root becomes unbalanced with a negative balance factor but the right subtree of the root has a positive balance factor. Double rotations have the same properties as the single rotations: The result is a valid, AVL-balanced search tree and the height of the result is the same as that of the initial tree.

Clearly the four rotations, LL, RR, LR, and RL, cover all the possible ways in which any one node can become unbalanced. But how many rotations are required to balance a tree when an insertion is done? The following theorem addresses this question:

**Theorem** When an AVL tree becomes unbalanced after an insertion, exactly one single or one double rotation is required to balance the tree.

**Proof** When an item, $x$, is inserted into an AVL tree, $T$, that item is placed in an external node of the tree. The only nodes in $T$ whose heights may be affected by the insertion of $x$ are those nodes which lie on the access path from the root of $T$ to $x$. Therefore, the only nodes at which an imbalance can appear are those along the access path. Furthermore, when a node is inserted into a tree, either the height of the tree remains the same or the height of the tree increases by one.
Double Rotations

Consider some node $c$ along the access path from the root of $T$ to $x$. When $x$ is inserted, the height of $c$ either increases by one, or remains the same. If the height of $c$ does not change, then no rotation is necessary at $c$ or at any node above $c$ in the access path.

If the height of $c$ increases then there are two possibilities: Either $c$ remains balanced or an imbalance appears at $c$. If $c$ remains balanced, then no rotation is necessary at $c$. However, a rotation may be needed somewhere above $c$ along the access path.

On the other hand, if $c$ becomes unbalanced, then a single or a double rotation must be performed at $c$. After the rotation is done, the height of $c$ is the same as it was before the insertion. Therefore, no further rotation is needed above $c$ in the access path.

Theorem suggests the following method for balancing an AVL tree after an insertion: Begin at the node containing the item which was just inserted and move back along the access path toward the root. For each node determine its height and check the balance condition. If the height of the current node does not increase, then the tree is AVL balanced and no further nodes need be considered. If the node has become unbalanced, a rotation is needed to balance it. After the rotation, the height of the node remains unchanged, the tree is AVL balanced and no further nodes need be considered. Otherwise, the height of the node increases by one, but no rotation is needed and we proceed to the next node on the access path.
Implementation

Program gives the code for the LLRotation method of the AVLTree class. This code implements the LL rotation shown in Figure. The purpose of the LLRotation method is to perform an LL rotation at the root of a given AVL tree instance.

```csharp
public class AVLTree : BinarySearchTree
{
    protected int height;

    protected void LLRotation()
    {
        if (IsEmpty)
            throw new InvalidOperationException();
        AVLTree tmp = Right;
        right = left;
        left = Right.left;
        Right.left = Right.right;
        Right.right = tmp;

        object tmpObj = key;
        key = Right.key;
        Right.key = tmpObj;

        Right.AdjustHeight();
        AdjustHeight();
    }
}
```

Program: AVLTree class LLRotation method.

The rotation is simply a sequence of variable manipulations followed by two height adjustments. Notice the rotation is done in such a way so that the the given AVLTree instance remains the root of the tree.
Implementation

This is done so that if the tree has a parent, it is not necessary to modify the contents of the parent.

The AVLTree class also requires an RRRotation method to implement an RR rotation. The implementation of that method follows directly from Program □. Clearly, the running time for the single rotations is \( O(1) \).

Program □ gives the implementation for the LRRotation method of the AVLTree class. This double rotation is trivially implemented as a sequence of two single rotations. As above, the method for the complementary rotation is easily derived from the given code. The running time for each of the double rotation methods is also \( O(1) \).

```java
public class AVLTree : BinarySearchTree
{
    protected int height;

    protected void LRRotation()
    {
        if (IsEmpty)
            throw new InvalidOperationException();
        Left.RRRotation();
        LLRotation();
    }
    // ...
}
```

Program: AVLTree class LRRotation method.

When an imbalance is detected, it is necessary to correct the imbalance by doing the appropriate rotation. The code given in Program □ takes care of this. The Balance method tests for an imbalance using the BalanceFactor property. The balance test itself takes constant time. If the node is balanced, only a constant-time height adjustment is needed.
Otherwise, the Balance method of the AVLTree class determines which of the four cases has occurred, and invokes the appropriate rotation to correct the imbalance. To determine which case has occurred, the Balance method calls the BalanceFactor property get accessor at most twice. Therefore, the time for selecting the case is constant. In all only one rotation is done to correct the imbalance. Therefore, the running time of this method is $O(1)$.

The Insert method for AVL trees is inherited from the BinarySearchTree class (see Program 4). The Insert method calls AttachKey to do the actual insertion. The AttachKey method is overridden in the AVLTree class as shown in Program 4.

```java
public class AVLTree : BinarySearchTree
{
    protected int height;

    protected override void Balance()
    {
        AdjustHeight();
        if (BalanceFactor > 1)
        {
            if (Left.BalanceFactor > 0)
                LLRotation();
            else
                LRRotation();
        }
        else if (BalanceFactor < -1)
        {
            if (Right.BalanceFactor < 0)
                RRRotation();
            else
                RLLRotation();
        }
        // ...
    }

Program: AVLTree class Balance method.
```
Implementation

```java
public class AVLTree : BinarySearchTree
{
    protected int height;

    public override void AttachKey(object obj)
    {
        if (!IsEmpty)
            throw new InvalidOperationException();
        key = obj;
        left = new AVLTree();
        right = new AVLTree();
        height = 0;
    }
    // ...
}
```

**Program:** AVLTree class AttachKey method.

The very last thing that the `Insert` method does is to call the `Balance` method, which has also been overridden as shown in Program. As a result the `Insert` method adjusts the heights of the nodes along the insertion path and does a rotation when an imbalance is detected. Since the height of an AVL tree is guaranteed to be $\mathcal{O}(\log n)$, the time for insertion is simply $\mathcal{O}(\log n)$. 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Removing Items from an AVL Tree

The method for removing items from an AVL tree is inherited from the `BinarySearchTree` class in the same way as AVL insertion. (See Program [ ]). All the differences are encapsulated in the `DetachKey` and `Balance` methods. The `Balance` method is discussed above. The `DetachKey` method is defined in Program [ ]

```csharp
public class AVLTree : BinarySearchTree
{
    protected int height;

    public override object DetachKey()
    {
        height = -1;
        return base.DetachKey();
    }

    // ...
}
```

**Program**: AVLTree class DetachKey method.

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
M-Way Search Trees

As defined in Section 1, an internal node of an $M$-way search tree contains $n$ subtrees and $n-1$ keys, where $2 \leq n \leq M$, for some fixed value of $M \geq 2$. The preceding sections give implementations for the special case in which the fixed value of $M=2$ is assumed (binary search trees). In this section, we consider the implementation of $M$-way search trees for arbitrary, larger values of $M \gg 2$.

Why are we interested in larger values of $M$? Suppose we have a very large data set--so large that we cannot get it all into the main memory of the computer at the same time. In this situation we implement the search tree in secondary storage, i.e., on disk. The unique characteristics of disk-based storage vis-à-vis memory-based storage make it necessary to use larger values of $M$ in order to implement search trees efficiently.

The typical disk access time is 1-10 ms, whereas the typical main memory access time is 10-100 ns. Thus, main memory accesses are between 10000 and 1000000 times faster than typical disk accesses. Therefore to maximize performance, it is imperative that the total number of disk accesses be minimized.

In addition, disks are block-oriented devices. Data are transferred between main memory and disk in large blocks. The typical block sizes are between 512 bytes and 4096 bytes. Consequently, it makes sense to organize the data structure to take advantage of the ability to transfer entire blocks of data efficiently.

By choosing a suitably large value for $M$, we can arrange that one node of an $M$-way search tree occupies an entire disk block. If every internal node in the $M$-way search tree has exactly $M$ children, we can use Theorem 1 to determine the height of the tree:

$$h \geq \lceil \log_M((M-1)n + 1) \rceil - 1, \quad \text{(10.9)}$$

where $n$ is the number of internal nodes in the search tree. A node in an $M$-way search tree that has $M$ children contains exactly $M$-1 keys. Therefore, altogether there are $K=(M-1)n$ keys and Equation 10.9 becomes $h \geq \lceil \log_M(K + 1) \rceil - 1$. Ideally the search tree is well balanced and the inequality becomes an equality.

For example, consider a search tree which contains $K = 2097151$ keys. Suppose the size of a disk block is such that we can fit a node of size $M=128$ in it. Since each node contains at most 127 keys, at
least 16513 nodes are required. In the best case, the height of the \( M \)-way search tree is only two and at most three disk accesses are required to retrieve any key! This is a significant improvement over a binary tree, the height of which is at least 20.
Implementing $M$-Way Search Trees

In order to illustrate the basic ideas, this section describes an implementation of $M$-way search trees in main memory. According to Definition, each internal node of an $M$-way search tree has $n$ subtrees, where $n$ is at least two and at most $M$. Furthermore, if a node has $n$ subtrees, it must contain $n-1$ keys.

Figure shows how we can implement a single node of an $M$-way search tree. The idea is that we use two arrays in each node--the first holds the keys and the second contains pointers to the subtrees. Since there are at most $M$ subtrees but only $M-1$ keys, the first element of the array of keys is not used.

Figure: Representing a node of an $M$-way search tree.
Implementation

Program introduces the MWayTree class. The MWayTree class extends the AbstractTree class introduced in Program and it implements the SearchableContainer interface defined in Program. The two fields, key and subtree, correspond to the components of a node shown in Figure. (Remember, the count field is inherited from the AbstractContainer base class introduced in Program).

```csharp
public class MWayTree : AbstractTree, SearchTree
{
    protected ComparableObject[] key;
    protected MWayTree[] subtree;

    public MWayTree(int m)
    {
        if (m < 2)
            throw new ArgumentException("invalid degree");
        key = new ComparableObject[m];
        subtree = new MWayTree[m];
    }

    public virtual int M
    {
        get { return subtree.Length; } }
    // ...
}
```

Program: MWayTree fields.

The first field, key, is an array ComparableObject instances. It is used to record the keys contained in the node. The second field, subtree, is an array of MWayTree instances which are the subtrees of the given node.

The inherited count field keeps track of the number of keys contained in the node. Recall, a node which
Implementation

contains \( \text{count} \) keys has \( \text{count} + 1 \) subtrees. We have chosen to keep track of the number of keys of a node rather than the number of subtrees because it simplifies the coding of the algorithms by eliminating some of the special cases.
Constructor and $M$ Property

Program defines the constructor and the $M$ property of the $M$WayTree class. The constructor takes a single, integer-valued argument that specifies the desired value of $M$. Provided $M$ is greater than or equal to two, the constructor creates two arrays of length $M$. Note, a node in an $M$-way tree contains at most $M-1$ keys. In the implementation shown, $key[0]$ is never used. Because the constructor allocates arrays of length $M$, its worst-case running time is $O(M)$.

The $M$ property shown in Program provides a get accessor that returns the value of $M$. That is, it returns the maximum number of subtrees that a node is allows to have. This is simply given by the length of the subtree array. The running time of the accessor is clearly $O(1)$.
Inorder Traversal

Whereas inorder traversal of an $N$-ary tree is *not* defined for $N \gt 2$, inorder traversal *is* defined for an $M$-way search tree: By definition, the inorder traversal of a search tree visits all the keys contained in the search tree *in order*.

Program □ is an implementation of the algorithm for depth-first traversal of an $M$-way search tree given in Section □. The keys contained in a given node are visited (by calling the InVisit method of the visitor) *in between* the appropriate subtrees of that node. That is, key $k_i$ is visited *in between* subtrees $T_{i-1}$ and $T_i$. 
Inorder Traversal

```java
1  public class MWayTree : AbstractTree, SearchTree
2  {
3     protected ComparableObject[] key;
4     protected MWayTree[] subtree;
5
6     public override void DepthFirstTraversal(
7         PrePostVisitor visitor)
8     {
9         if (!IsEmpty)
10            {
11                for (int i = 0; i <= count + 1; ++i)
12                    {
13                        if (i >= 2)
14                            visitor.PostVisit(key[i - 1]);
15                        if (i >= 1 && i <= count)
16                            visitor.InVisit(key[i]);
17                        if (i <= count - 1)
18                            visitor.PreVisit(key[i + 1]);
19                        if (i <= count)
20                            subtree[i].DepthFirstTraversal(visitor);
21                    }
22            }
23    } // ...
24  }
```

Program: MWayTree class DepthFirstTraversal method.

In addition, the PostVisit method is called on \(k_{i-1}\) after subtree \(T_{i-1}\) has been visited, and the PreVisit method is called on \(k_{i+1}\) before subtree \(T_i\) is visited.

It is clear that the amount of work done at each node during the course of a depth-first traversal is proportional to the number of keys contained in that node. Therefore, the total running time for the depth-first traversal is \(K(T\{\text{PreVisit}\} + T\{\text{InVisit}\} + T\{\text{PostVisit}\}) + O(K)\), where \(K\) is the number of keys contained in the search tree.
Finding Items in an \( M \)-Way Search Tree

Two algorithms for finding items in an \( M \)-way search tree are described in this section. The first is a naive implementation using linear search. The second version improves upon the first by using a binary search.

- Linear Search
- Binary Search

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Linear Search

Program □ gives the naïve version of the Find method of the MWayTree class. The Find method takes a ComparableObject and locates the item in the search tree which matches the given object.

```csharp
public class MWayTree : AbstractTree, SearchTree
{
    protected ComparableObject[] key;
    protected MWayTree[] subtree;

    public virtual ComparableObject Find(ComparableObject obj)
    {
        if (IsEmpty)
            return null;
        int i;
        for (i = count; i > 0; --i)
        {
            int diff = obj.CompareTo(key[i]);
            if (diff == 0)
                return key[i];
            if (diff > 0)
                break;
        }
        return subtree[i].Find(obj);
    }
    // ...
}
```

Program: MWayTree class Find method (linear search).

Consider the execution of the Find method for a node $T$ of an $M$-way search tree. Suppose the object of the search is $x$. Clearly, the search fails when $T = \emptyset$ (lines 10-11). In this case, null is returned.
Suppose \( T = \{T_0, k_1, T_1, k_2, T_2, \ldots, k_{n-1}, T_{n-1}\} \). The linear search on lines 13-20 considers the keys \( k_{n-1}, k_{n-2}, k_{n-3}, \ldots, k_1 \), in that order. If a match is found, the matching object is returned immediately (lines 16-17).

Otherwise, when the main loop terminates there are three possibilities: \( i = 0 \) and \( x < k_{i+1}; \) \( 1 \leq i \leq n - 2 \) and \( k_i < x < k_{i+1}; \) or \( i = n-1 \) and \( k_i < x \). In all three cases, the appropriate subtree in which to continue search is \( T_i \) (line 21).

Clearly the running time of Program is determined by the main loop. In the worst case, the loop is executed \( M-1 \) times. Therefore, at each node in the search path at most \( M-1 \) object comparisons are done.

Consider an unsuccessful search in an \( M \)-way search tree. The running time of the \texttt{Find} method is

\[
(M - 1)(h + 1)T_{\text{compareTo}} + O(Mh)
\]

in the worst case, where \( h \) is the height of the tree and \( T_{\text{compareTo}} \) is the time required to compare two objects. Clearly, the time for a successful search has the same asymptotic bound. If the tree is balanced and \( T_{\text{compareTo}} = O(1) \), then the running time of Program is \( O(M \log_M K) \), where \( K \) is the number of keys in the tree.
Binary Search

We can improve the performance of the $M$-way search tree search algorithm by recognizing that since the keys are kept in a sorted array, we can do a binary search rather than a linear search. Program 9 gives an alternate implementation for the Find method of the MWayTree class. This method makes use of the FindIndex method which does the actual binary search.

```csharp
public class MWayTree : AbstractTree, SearchTree
{
    protected ComparableObject[] key;
    protected MWayTree[] subtree;

    protected int FindIndex(ComparableObject obj)
    {
        if (IsEmpty || obj < key[1])
            return 0;
        int left = 1;
        int right = count;
        while (left < right)
        {
            int middle = (left + right + 1) / 2;
            if (obj < key[middle])
                right = middle - 1;
            else
                left = middle;
        }
        return left;
    }

    public ComparableObject Find(ComparableObject obj)
    {
        if (IsEmpty)
            return null;
        int index = FindIndex(obj);
        if (index != 0 && obj == key[index])
            return key[index];
        return null;
    }
}
```
The **FindIndex** method as its argument a ComparableObject, say $x$, and returns an int in the range between 0 and $n-1$, where $n$ is the number of subtrees of the given node. The result is the largest integer $i$, if it exists, such that $x \geq k_i$ where $k_i$ is the $i$th key. Otherwise, it returns the value 0.

**FindIndex** determines its result by doing a binary search. In the worst case, $\lceil \log_2 (M - 1) \rceil + 1$ iterations of the main loop (lines 14-21) are required to determine the correct index. One object comparison is done before the loop (line 10) and one comparison is done in each loop iteration (line 17). Therefore, the running time of the **FindIndex** method is

$$((\lceil \log_2 (M - 1) \rceil + 2)T\{\text{compareTo}\} + O(\log_2 M)).$$

If $T\{\text{compareTo}\} = O(1)$, this simplifies to $O(\log_2 M)$.

The **Find** method of the MWayTree class does the actual search. It calls **FindIndex** to determine largest integer $i$, if it exists, such that $x \geq k_i$ where $k_i$ is the $i$th key (line 29). If it turns out that $x = k_i$, then the search is finished (lines 30-31). Otherwise, **Find** calls itself recursively to search subtree $T_i$ (line 33).

Consider a search in an $M$-way search tree. The running time of the second version of **Find** is

$$(h + 1)(\lceil \log_2 (M - 1) \rceil + 2)T\{\text{compareTo}\} + O(h \log_2 M),$$

where $h$ is the height of the tree and regardless of whether the search is successful. If the tree is balanced and $T\{\text{compareTo}\} = O(1)$, then the running time of Program [] is simply $O((\log_2 M)(\log M K))$, where $K$ is the number of keys in the tree.
Inserting Items into an $M$-Way Search Tree

The method for inserting items in an $M$-way search tree follows directly from the algorithm for insertion in a binary search tree given in Section 6. The added wrinkle in an $M$-way tree is that an internal node may contain between 1 and $M-1$ keys whereas an internal node in a binary tree must contain exactly one key.

Program 7 gives the implementation of the Insert method of the MWayTree class. This method takes as its argument the object to be inserted into the search tree.

```csharp
public class MWayTree : AbstractTree, SearchTree
{
    protected ComparableObject[] key;
    protected MWayTree[] subtree;

    public virtual void Insert(ComparableObject obj)
    {
        if (IsEmpty)
        {
            subtree[0] = new MWayTree(M);
            key[1] = obj;
            subtree[1] = new MWayTree(M);
            count = 1;
        }
        else
        {
            int index = FindIndex(obj);
            if (index != 0 && obj == key[index])
                throw new ArgumentException("duplicate key");
            if (!IsFull)
            {
                for (int i = count; i > index; --i)
                {
                    key[i + 1] = key[i];
                    subtree[i + 1] = subtree[i];
                }
                count = count + 1;
            }
        }
    }
}
```
Inserting Items into an M-Way Search Tree

Program: MWayTree class Insert method.

The general algorithm for insertion is to search for the item to be inserted and then to insert it at the point where the search terminates. If the search terminates at an external node, that node is transformed to an internal node of the form \( \{ \emptyset, x, \emptyset \} \), where \( x \) is the key just inserted (lines 10-16).

If the search terminates at an internal node, we insert the new item into the sorted list of keys at the appropriate offset. Inserting the key \( x \) in the array of keys moves all the keys larger than \( x \) and the associated subtrees to the right one position (lines 23-33). The hole in the list of subtrees is filled with an empty tree (line 31).

The preceding section gives the running time for a search in an \( M \)-way search tree as

\[
(h + 1)(\lceil \log_2 (M - 1) \rceil + 2)T\{\text{compareTo}\} + O(h \log M),
\]

where \( h \) is the height of the tree. The additional time required to insert the item into the node once the correct node has been located is \( O(M) \). Therefore, the total running time for the Insert algorithm given in Program [ ] is

\[
(h + 1)(\lceil \log_2 (M - 1) \rceil + 2)T\{\text{compareTo}\} + O(h \log M) + O(M).
\]
Removing Items from an $M$-Way Search Tree

The algorithm for removing items from an $M$-way search tree follows directly from the algorithm for removing items from a binary search tree given in Section □. The basic idea is that the item to be deleted is pushed down the tree from its initial position to a node from which it can be easily deleted. Clearly, items are easily deleted from leaf nodes. In addition, consider an internal node of an $M$-way search tree of the form

$$T = \{ T_0, k_1, T_1, \ldots, T_{i-1}, k_i, T_i, \ldots, k_{n-1}, T_{n-1} \}.$$ 

If both $T_{i-1}$ and $T_i$ are empty trees, then the key $k_i$ can be deleted from $T$ by removing both $k_i$ and $T_i$, say. If $T_{i-1}$ is non-empty, $k_i$ can be pushed down the tree by swapping it with the largest key in $T_{i-1}$; and if $T_i$ is non-empty, $k_i$ can be pushed down the tree by swapping it with the smallest key in $T_i$.

Program □ gives the code for the Withdraw method of the MWayTree class. The general form of the algorithm follows that of the Withdraw method for the BinarySearchTree class (Program □).

```csharp
public class MWayTree : AbstractTree, SearchTree
{
    protected ComparableObject[] key;
    protected MWayTree[] subtree;

    public virtual void Withdraw(ComparableObject obj)
    {
        if (IsEmpty)
            throw new ArgumentException("object not found");
        int index = FindIndex(obj);
        if (index != 0 && obj == key[index])
        {
            if (!subtree[index - 1].IsEmpty)
            {
                ComparableObject max = subtree[index - 1].Max;
                key[index] = max;
                subtree[index - 1].Withdraw(max);
            }
        }
    }
}
```
Removing Items from an M-Way Search Tree

Program: MWayTree class Withdraw method.

```java
subtree[index - 1].Withdraw(max);
}

else if (!subtree[index].IsEmpty)
{
    ComparableObject min = subtree[index].Min;
    key[index] = min;
    subtree[index].Withdraw(min);
}
else
{
    --count;
    int i;
    for (i = index; i <= count; ++i)
    {
        key[i] = key[i + 1];
        subtree[i] = subtree[i + 1];
    }
    key[i] = null;
    subtree[i] = null;
    if (count == 0)
        subtree[0] = null;

    }
    else
        subtree[index].Withdraw(obj);
}
// ...
```
B-Trees

Just as AVL trees are balanced binary search trees, B-trees are balanced $M$-way search trees. By imposing a balance condition, the shape of an AVL tree is constrained in a way which guarantees that the search, insertion, and withdrawal operations are all $O(\log n)$, where $n$ is the number of items in the tree. The shapes of B-Trees are constrained for the same reasons and with the same effect.

Definition (B-Tree) A B-Tree of order $M$ is either the empty tree or it is an $M$-way search tree $T$ with the following properties:

1. The root of $T$ has at least two subtrees and at most $M$ subtrees.
2. All internal nodes of $T$ (other than its root) have between $\left\lfloor \frac{M}{2} \right\rfloor$ and $M$ subtrees.
3. All external nodes of $T$ are at the same level.

A B-tree of order one is clearly impossible. Hence, B-trees of order $M$ are really only defined for $M \geq 2$. However, in practice we expect that $M$ is large for the same reasons that motivate $M$-way search trees--large databases in secondary storage.

Figure gives an example of a B-tree of order $M=3$. By Definition, the root of a B-tree of order three has either two or three subtrees and the internal nodes also have either two or three subtrees. Furthermore, all the external nodes, which are shown as small boxes in Figure, are at the same level.
B-Trees

Figure: A B-tree of order 3.

It turns out that the balance conditions imposed by Definition are good in the same sense as the AVL balance conditions. That is, the balance condition guarantees that the height of B-trees is logarithmic in the number of keys in the tree and the time required for insertion and deletion operations remains proportional to the height of the tree even when balancing is required.

**Theorem** The minimum number of keys in a B-tree of order \( M \geq 2 \) and height \( h \geq 0 \) is

\[
\begin{align*}
n_h &= 2 \left\lfloor \frac{M}{2} \right\rfloor^h - 1.
\end{align*}
\]

**Proof** Clearly, a B-tree of height zero contains at least one node. Consider a B-tree order \( M \) and height \( h > 0 \). By Definition, each internal node (except the root) has at least \( \left\lfloor \frac{M}{2} \right\rfloor \) subtrees. This implies the minimum number of keys contained in an internal node is \( \left\lfloor \frac{M}{2} \right\rfloor - 1 \). The minimum number of keys a level zero is 1; at level one, \( 2 \left( \left\lfloor \frac{M}{2} \right\rfloor - 1 \right) \); at level two, \( 2 \left( \left\lfloor \frac{M}{2} \right\rfloor \right) \left( \left\lfloor \frac{M}{2} \right\rfloor - 1 \right) \); at level three, \( 2 \left( \left\lfloor \frac{M}{2} \right\rfloor \right)^2 \left( \left\lfloor \frac{M}{2} \right\rfloor - 1 \right) \); and so on.

Therefore the minimum number of keys in a B-tree of height \( h > 0 \) is given by the summation

\[
\begin{align*}
n_h &= 1 + 2 \left( \left\lfloor \frac{M}{2} \right\rfloor - 1 \right) \sum_{i=0}^{h-1} \left\lfloor \frac{M}{2} \right\rfloor^i \\
&= 1 + 2 \left( \left\lfloor \frac{M}{2} \right\rfloor - 1 \right) \left( \left\lfloor \frac{M}{2} \right\rfloor^h - 1 \right) \\
&= 2 \left( \left\lfloor \frac{M}{2} \right\rfloor \right)^h - 1.
\end{align*}
\]
A corollary of Theorem 4 is that the height, $h$, of a B-tree containing $n$ keys is given by

$$h \leq \log_{M/2} \left( \left( \frac{n}{2} + 1 \right) / 2 \right).$$

Thus, we have shown that a B-tree satisfies the first criterion of a good balance condition—the height of B-tree with $n$ internal nodes is $O(\log n)$. What remains to be shown is that the balance condition can be efficiently maintained during insertion and withdrawal operations. To see that it can, we need to look at an implementation.

- Implementing B-Trees
- Inserting Items into a B-Tree
- Removing Items from a B-Tree
Implementing B-Trees

Having already implemented the $M$-way search tree class, $\text{MWayTree}$, we can make use of much the existing code to implement a B-tree class. Program introduces the $\text{BTree}$ class which extends the class $\text{MWayTree}$ class introduced in Program. With the exception of the two methods which modify the tree, $\text{Insert}$ and $\text{Withdraw}$, the $\text{BTree}$ class inherits all its functionality from the $M$-way tree class. Of course, the $\text{Insert}$ and $\text{Withdraw}$ methods need to be redefined in order to ensure that every time tree is modified the tree which results is a B-tree.

```csharp
public class BTree : MWayTree
{
    protected BTree parent;

    public BTree(int m) : base(m)
    {
    }

    public virtual void AttachSubtree(int i, MWayTree arg)
    {
        BTree btree = (BTree)arg;
        subtree[i] = btree;
        btree.parent = this;
    }

    // ...
}
```

**Program:** $\text{BTree}$ fields.

- **Fields**
- **Constructor and AttachSubtree Methods**
Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Fields

To simplify the implementation of the algorithms, a parent field has been added. The parent field refers to the BTree node which is the parent of the given node. Whereas the subtree field of the MWayTree class allows an algorithm to move down the tree; the parent field admits movement up the tree. Since the root of a tree has no parent, the parent field of the root node is assigned the value null.
Constructor and AttachSubtree Methods

Program defines a constructor that takes a single `int` argument `M` and creates an empty B-tree of order `M`. By default, the `parent` field is null.

The AttachSubtree method is used to attach a new subtree to a given node. The AttachSubtree routine takes an integer, `i`, and an `M`-way tree (which must be a B-tree instance), and makes it the `i`th subtree of the given node. Notice that this method also modifies the `parent` field in the attached node.
Inserting Items into a B-Tree

The algorithm for insertion into a B-Tree begins as do all the other search tree insertion algorithms: To insert item \( x \), we begin at the root and conduct a search for it. Assuming the item is not already in the tree, the unsuccessful search will terminate at a leaf node. This is the point in the tree at which the \( x \) is inserted.

If the leaf node has fewer than \( M-1 \) keys in it, we simply insert the item in the leaf node and we are done. For example, consider a leaf node with \( n<M \) subtrees and \( n-1 \) keys of the form

\[
T = \{T_0, k_1, T_1, k_2, T_2, \ldots, k_{n-1}, T_{n-1}\}.
\]

For every new key inserted in the node, a new subtree is required too. In this case because \( T \) is a leaf, all its subtrees are empty trees. Therefore, when we insert item \( x \), we really insert the pair of items \((x, \emptyset)\).

Suppose the key to be inserted falls between \( k_i \) and \( k_{i+1} \), i.e., \( k_i < x < k_{i+1} \). When we insert the pair \((x, \emptyset)\) into \( T \) we get the new leaf \( T' \) given by

\[
T' = \{T_0, k_1, T_1, k_2, T_2, \ldots, k_i, x, \emptyset, k_{i+1}, T_{i+1}, \ldots, k_{n-1}, T_{n-1}\}.
\]

What happens when the leaf is full? That is, suppose we wish to insert the pair, \((x, \emptyset)\) into a node \( T \) which already has \( M-1 \) keys. Inserting the pair in its correct position gives a result of the form

\[
T' = \{T_0, k_1, T_1, k_2, T_2, \ldots, k_M, T_M\}.
\]

However, this is not a valid node in a B-tree of order \( M \) because it has \( M+1 \) subtrees and \( M \) keys. The solution is to split node \( T \) in half as follows

\[
T'_L = \{T_0, k_1, T_1, \ldots, k_{[M/2]-1}, T_{[M/2]-1}\}
\]

\[
T'_R = \{T_{[M/2]}, k_{[M/2]+1}, T_{[M/2]+1}, \ldots, k_M, T_M\}.
\]

Note, \( T'_L \) is a valid B-tree node because it contains \([M/2] \) subtrees and \([M/2] - 1 \) keys. Similarly, \( T'_R \) is a valid B-tree node because it contains \([\lceil(M + 1)/2 \rceil] \) subtrees and \([\lceil(M + 1)/2 \rceil - 1 \) keys. Note that there is still a key left over, namely \( k_{[M/2]} \).
There are now two cases to consider—either \( T \) is the root or it is not. Suppose \( T \) is not the root. Where we once had the single node \( T \), we now have the two nodes, \( T'_L \) and \( T'_R \), and the left-over key, \( k[\lfloor M/2 \rfloor] \). This situation is resolved as follows: First, \( T'_L \) replaces \( T \) in the parent of \( T \). Next, we take the pair \( \langle k[\lfloor M/2 \rfloor], T'_R \rangle \) and recursively insert it in the parent of \( T \).

Figure illustrates this case for a B-tree of order three. Inserting the key 6 in the tree causes the leaf node to overflow. The leaf is split in two. The left half contains key 5; and the right, key 7; and key 6 is left over. The two halves are re-attached to the parent in the appropriate place with the left-over key between them.
**Figure:** Inserting items into a B-tree (insert 6).

If the parent node fills up, then it too is split and the two new nodes are inserted in the grandparent. This process may continue all the way up the tree to the root. What do we do when the root fills up? When the root fills, it is also split. However, since there is no parent into which to insert the two new children, a new root is inserted above the old root. The new root will contain exactly two subtrees and one key, as allowed by Definition □.

Figure □ illustrates this case for a B-tree of order three. Inserting the key 3 in the tree causes the leaf node to overflow. Splitting the leaf and reattaching it causes the parent to overflow. Similarly, splitting the parent and reattaching it causes the grandparent to overflow but the grandparent is the root. The root is split and a new root is added above it.
**Figure:** Inserting items into a B-tree (insert 3).

Notice that the height of the B-tree only increases when the root node splits. Furthermore, when the root node splits, the two halves are both attached under the new root. Therefore, the external nodes all remain at the same depth, as required by Definition □.
Implementation

Insertion in a B-tree is a two-pass process. The first pass moves down the tree from the root in order to locate the leaf in which the insertion is to begin. This part of the algorithm is quite similar to the Find method given in Program □. The second pass moves from the bottom of the tree back up to the root, splitting nodes and inserting them further up the tree as needed. Program □ gives the code for the first (downward) pass (Insert method) and the Program □ gives the code for the second (upward) pass (InsertPair method).

```
public class BTree : MWayTree
{
    protected BTree parent;

    public override void Insert(ComparableObject obj)
    {
        if (IsEmpty)
        {
            if (parent == null)
            {
                AttachSubtree(0, new BTree(M));
                key[1] = obj;
                AttachSubtree(1, new BTree(M));
                count = 1;
            }
            else
                parent.InsertPair(obj, new BTree(M));
        }
        else
        {
            int index = FindIndex(obj);
            if (index != 0 && obj == key[index])
                throw new ArgumentException("duplicate key");
            subtree[index].Insert(obj);
        }
    }
```
Program: BTree class Insert method.

In the implementation shown, the downward pass starts at the root node and descends the tree until it arrives at an external node. If the external node has no parent, it must be the root and, therefore, the tree is empty. In this case, the root becomes an internal node containing a single key and two empty subtrees (lines 11-13). Otherwise, we have arrived at an external node in a non-empty tree and the second pass begins by calling InsertPair to insert the pair \((\pi, \emptyset)\) in the parent.

The upward pass of the insertion algorithm is done by the recursive InsertPair method shown in Program ```. The InsertPair method takes two arguments. The first, object, is a ComparableObject and the second, child, is a BTree. It is assumed that all the keys in child are strictly greater than object.

```java
public class BTree : MWayTree
{
    protected BTree parent;

    protected virtual void InsertPair(
        ComparableObject obj, BTree child)
    {
        int index = FindIndex(obj);
        if (!IsFull)
        {
            InsertKey(index + 1, obj);
            InsertSubtree(index + 1, child);
            ++count;
        }
        else
        {
            ComparableObject extraKey =
                InsertKey(index + 1, obj);
            BTree extraTree = InsertSubtree(index + 1, child);
            if (parent == null)
            {
                BTree left = new BTree(M);
                BTree right = new BTree(M):
```
Implementation

```
BTree left = new BTree(M);
BTree right = new BTree(M);
left.AttachLeftHalfOf(this);
right.AttachRightHalfOf(this);
right.InsertPair(extraKey, extraTree);
AttachSubtree(0, left);
key[1] = key[(M + 1)/2];
AttachSubtree(1, right);
}
else
{
    count = (M + 1)/2 - 1;
    BTree right = new BTree(M);
    right.AttachRightHalfOf(this);
    right.InsertPair(extraKey, extraTree);
    parent.InsertPair(key[(M + 1)/2], right);
    }
    }
    
} // ...
```

Program: BTree class InsertPair method.

The InsertPair method calls FindIndex to determine the position in the array of keys at which pair (object, child) should be inserted (line 8). If this node is full (line 9), the InsertKey is called to insert the given key at the specified position in the key array (line 11) and InsertSubtree is called to insert the given tree at the specified position in the subtree array (line 12).

In the event that the node is full, the InsertKey method returns the key which falls off the right end of the array. This is assigned to extraKey (line 17). Similarly, the InsertSubtree method returns the tree which falls off the right end of the array. This is assigned to extraTree (line 19).

The node has now overflowed and it is necessary to balance the B-tree. If the node overflows and it is the root (line 20), then two new B-trees, left and right are created (lines 22-23). The first \(\lfloor M/2 \rfloor - 1\) keys and \(\lfloor M/2 \rfloor\) subtrees of the given node are moved to the left tree by the AttachLeftHalfOf method (line 24); and the last \(\lfloor (M + 1)/2 \rfloor - 2\) keys and \(\lfloor (M + 1)/2 - 1 \rfloor\) subtrees of the given node are moved to the right tree by the AttachRightHalfOf method (line 25). Then, the pair (extraKey, extraTree) is inserted into the right tree (line 26).
The left-over key is the one in the middle of the array, i.e., $k_{\lceil M/2 \rceil}$. Finally, the root node is modified so that it contains the two new subtrees and the single left-over key (lines 27-30).

If the node overflows and it is not the root, then one new B-tree is created, right (line 35). The last $\lceil (M + 1)/2 \rceil - 2$ keys and $\lceil (M + 1)/2 - 1 \rceil$ subtrees of the given node are moved to the left tree by the AttachRightHalfOf method (line 36) and the pair (extraKey,extraTree) is inserted in the right tree (line 37). The first $\lceil M/2 \rceil - 1$ keys and $\lceil M/2 \rceil$ subtrees of the given node remain attached to it.

Finally, the InsertPair method calls itself recursively to insert the left-over key, $k_{\lceil M/2 \rceil}$, and the new B-tree, right, into the parent of this (line 38). This is the place where the parent field is needed!
Running Time Analysis

The running time of the downward pass of the insertion algorithm is identical to that of an unsuccessful search (assuming the item to be inserted is not already in the tree). That is, for a B-tree of height $h$, the worst-case running time of the downward pass is

$$(h + 1)(\lceil \log_2 (M - 1) \rceil + 2)T\{\text{CompareTo}\} + O(h \log M).$$

The second pass of the insertion algorithm does the insertion and balances the tree if necessary. In the worst case, all of the nodes in the insertion path up to the root need to be balanced. Each time the `InsertPair` method is invoked, it calls `FindIndex` which has running time $$(\lceil \log_2 (M - 1) \rceil + 2)T\{\text{CompareTo}\} + O(\log M)$$ in the worst case. The additional time required to balance a node is $O(M)$. Therefore, the worst-case running time of the upward pass is

$$(h + 1)(\lceil \log_2 (M - 1) \rceil + 2)T\{\text{CompareTo}\} + O(hM).$$

Therefore, the total running time for insertion is

$$2(h + 1)(\lceil \log_2 (M - 1) \rceil + 2)T\{\text{CompareTo}\} + O(hM).$$

According to Theorem, the height of a B-tree is $h \leq \log_{\lceil M/2 \rceil}((n + 1)/2)$, where $n$ is the number of keys in the B-tree. If we assume that two keys can be compared in constant time, i.e., $T\{\text{CompareTo}\} = O(1)$, then the running time for insertion in a B-tree is simply $O(M \log n)$. 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Removing Items from a B-Tree

The algorithm for removing items from a B-tree is similar to the algorithm for removing item from an AVL tree. That is, once the item to be removed has been found, it is pushed down the tree to a leaf node where it can be easily deleted. When an item is deleted from a node it is possible that the number of keys remaining is less than \( \lceil M/2 \rceil - 1 \). In this case, balancing is necessary.

The algorithm of balancing after deletion is like the balancing after insertion in that it progresses from the leaf node up the tree toward the root. Given a node \( T \) which has \( \lceil M/2 \rceil - 2 \) keys, there are four cases to consider.

In the first case, \( T \) is the root. If no keys remain, \( T \) becomes the empty tree. Otherwise, no balancing is needed because the root is permitted to have as few as two subtrees and one key. For the remaining cases \( T \) is not the root.

In the second case \( T \) has \( \lceil M/2 \rceil - 2 \) keys and it also has a sibling immediately on the left with at least \( \lceil M/2 \rceil \) keys. The tree can be balanced by doing an LL rotation as shown in Figure □. Notice that after the rotation, both siblings have at least \( \lceil M/2 \rceil - 1 \) keys. Furthermore, the heights of the siblings remain unchanged. Therefore, the resulting tree is a valid B-tree.
The third case is the left-right mirror of the second case. That is, \( T \) has \( \lceil M/2 \rceil - 2 \) keys and it also has a sibling immediately on the right with a least \( \lceil M/2 \rceil \) keys. In this case, the tree can be balanced by doing an RR rotation.

In the fourth and final case, \( T \) has \( \lceil M/2 \rceil - 2 \) keys, and its immediate sibling(s) have \( \lceil M/2 \rceil - 1 \) keys. In this case, the sibling(s) cannot give-up a key in a rotation because they already have the minimum number of keys. The solution is to merge \( T \) with one of its siblings as shown in Figure.

**Figure:** LL rotation in a B-tree.
Removing Items from a B-Tree

Figure: Merging nodes in a B-tree.

The merged node contains \( \left\lceil \frac{M}{2} \right\rceil - 2 \) keys from \( T \), \( \left\lceil \frac{M}{2} \right\rceil - 1 \) keys from the sibling, and one key from the parent (the key \( x \) in Figure). The resulting node contains \( 2 \left\lceil \frac{M}{2} \right\rceil - 2 \) keys altogether, which is \( M-2 \) if \( M \) is even and \( M-1 \) if \( M \) is odd. Either way, the resulting node contains no more than \( M-1 \) keys and is a valid B-tree node. Notice that in this case a key has been removed from the parent of \( T \). Therefore, it may be necessary to balance the parent. Balancing the parent may necessitate balancing the grandparent, and so on, up the tree to the root.
Applications

There are many applications for search trees. The principal characteristic of such applications is that a database of keyed information needs to be frequently accessed and the access pattern is either unknown or known to be random. For example, *dictionaries* are often implemented using search trees. A dictionary is essentially a container that contains ordered key/value pairs. The keys are words in a source language and, depending on the application, the values may be the definitions of the words or the translation of the word in a target language.

This section presents a simple application of search trees. Suppose we are required to translate the words in an input file one-by-one from some source language to another target language. In this example, the translation is done one word at a time. That is, no natural language syntactic or semantic processing is done.

In order to implement the translator we assume that there exists a text file, which contains pairs of words. The first element of the pair is a word in the source language and the second element is a word in the target language. To translate a text, we first read the words and the associated translations and build a search tree. The translation is created one word at a time by looking up each word in the text.

Program □ gives an implementation of the translator. The `Translate` method uses a search tree to hold the pairs of words. In this case, an AVL tree is used. However, this implementation works with all the search tree types described in this chapter (e.g., `BinarySearchTree`, `AVLTree`, `MWayTree`, and `BTree`).
Applications

```java
public class Algorithms {

    public static void Translate(TextReader dictionary, TextReader inputText, TextWriter outputText) {
        SearchTree searchTree = new AVLTree();
        string line;
        while(((line = dictionary.ReadLine()) != null) {
            string[] words = line.Split(null);
            if (words.Length != 2) {
                throw new InvalidOperationException();
                searchTree.Insert(new Association(words[0], words[1]));
            }
        }

        while(((line = inputText.ReadLine()) != null) {
            foreach (string word in line.Split(null)) {
                ComparableObject obj =
                searchTree.Find(new Association(word));
                if (obj == null) {
                    outputText.WriteLine("{}", word);
                } else {
                    Association assoc = (Association)obj;
                    outputText.WriteLine("{}", assoc.Value);
                }
            } outputText.WriteLine();
        }
    }
}
```

Program: Application of search trees--word translation.

The `Translate` method reads pairs of strings from the input stream (lines 8-12). The `Association`
class defined in Section is used to contain the key/value pairs. A new instance is created for each key/value pair which is then inserted into the search tree (lines 13-14). The process of building the search tree terminates when the end-of-file is encountered.

During the translation phase, the Translate method reads words one at a time from the input stream and writes the translation of each word on the output stream. Each word is looked up as it is read (lines 21-22). If no key matches the given word, the word is printed followed by a question mark (lines 23-24). Otherwise, the value associated with the matching key is printed (lines 27-28).
Exercises

1. For each of the following key sequences determine the binary search tree obtained when the keys are inserted one-by-one in the order given into an initially empty tree:
   1. 1, 2, 3, 4, 5, 6, 7.
   2. 4, 2, 1, 3, 6, 5, 7.
   3. 1, 6, 7, 2, 4, 3, 5.

2. For each of the binary search trees obtained in Exercise determine the tree obtained when the root is withdrawn.

3. Repeat Exercises and for AVL trees.

4. Derive an expression for the total space needed to represent a tree of \( n \) internal nodes using each of the following classes:
   1. `BinarySearchTree` introduced in Program,
   2. `AVLTree` introduced in Program,
   3. `MWayTree` introduced in Program, and
   4. `BTree` introduced in Program.
   
   **Hint:** For the `MWayTree` and `BTree` assume that the tree contains are \( k \) keys, where \( k \geq n \).

5. To delete a non-leaf node from a binary search tree, we swap it either with the smallest key its right subtree or with the largest key in its left subtree and then recursively delete it from the subtree. In a tree of \( n \) nodes, what is the maximum number of swaps needed to delete a key?

6. Devise an algorithm to compute the internal path length of a tree. What is the running time of your algorithm?

7. Devise an algorithm to compute the external path length of a tree. What is the running time of your algorithm?

8. Suppose that you are given a sorted sequence of \( n \) keys, \( k_0 \leq k_1 \leq \cdots \leq k_{n-1} \), to be inserted into a binary search tree.
   1. What is the minimum height of a binary tree that contains \( n \) nodes.
   2. Devise an algorithm to insert the given keys into a binary search tree so that the height of the resulting tree is minimized.
   3. What is the running time of your algorithm?

9. Devise an algorithm to construct an AVL tree of a given height \( h \) that contains the minimum number of nodes. The tree should contain the keys \( 1, 2, 3, \ldots, N_h \), where \( N_h \) is given by Equation.
10. Consider what happens when we insert the keys \(1, 2, 3, \ldots, 2^h + 1\) one-by-one in the order given into an initially empty AVL tree for \(h \geq 0\). Prove that the result is always a perfect tree of height \(h\).

11. The \texttt{Find} method defined in Program \texttt{0} is recursive. Write a non-recursive method to find a given item in a binary search tree.

12. Repeat Exercise 10 for the \texttt{Min} method defined in Program \texttt{0}.

13. Devise an algorithm to select the \(k\)th key in a binary search tree. For example, given a tree with \(n\) nodes, \(k=0\) selects the smallest key, \(k=n-1\) selects the largest key, and \(k = \left\lceil n/2 \right\rceil - 1\) selects the median key.

14. Devise an algorithm to test whether a given binary search tree is AVL balanced. What is the running time of your algorithm?

15. Devise an algorithm that takes two values, \(a\) and \(b\) such that \(a \leq b\), and which visits all the keys \(x\) in a binary search tree such that \(a \leq x \leq b\). The running time of your algorithm should be \(O(N + \log n)\), where \(N\) is the number of keys visited and \(n\) is the number of keys in the tree.

16. Devise an algorithm to merge the contents of two binary search trees into one. What is the running time of your algorithm?

17. (This question should be attempted after reading Chapter \texttt{0}). Prove that a complete binary tree (Definition \texttt{0}) is AVL balanced.

18. Do Exercise 10.

19. For each of the following key sequences determine the 3-way search tree obtained when the keys are inserted one-by-one in the order given into an initially empty tree:
   1. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
   2. 3, 1, 4, 5, 9, 2, 6, 8, 7, 0.
   3. 2, 7, 1, 8, 4, 5, 9, 0, 3, 6.

20. Repeat Exercise 10 for B-trees of order 3.
Projects

1. Complete the implementation of the BinarySearchTree class introduced in Program by providing suitable definitions for the following operations: IsMember and Max. You must also have a complete implementation of the base class BinaryTree. (See Project). Write a test program and test your implementation.

2. Complete the implementation of the AVLTree class introduced in Program by providing suitable definitions for the following operations: RRRotation, and RLRotation. You must also have a complete implementation of the base class BinarySearchTree. (See Project). Write a test program and test your implementation.

3. Complete the implementation of the MWayTree class introduced in Program by providing suitable definitions for the following operations: Purge, Count, isEmpty, IsLeaf, Degree, Key, GetSubtree, IsMember, Min, Max, BreadthFirstTraversal, and GetEnumerator. Write a test program and test your implementation.

4. Complete the implementation of the BTree class introduced in Program by providing suitable definitions for the following methods: InsertKey, InsertSubtree, AttachLeftHalfOf, AttachRightHalfOf, and Withdraw. You must also have a complete implementation of the base class MWayTree. (See Project). Write a test program and test your implementation.

5. The binary search tree Withdraw method shown in Program is biased in the following way: If the key to be deleted is in a non-leaf node with two non-empty subtrees, the key is swapped with the maximum key in the left subtree and then recursively deleted from the left subtree. Following a long series of insertions and deletions, the search tree will tend to have more nodes in the right subtrees and fewer nodes in the left subtrees. Devise and conduct an experiment that demonstrates this phenomenon.

6. Consider the implementation of AVL trees. In order to check the AVL balance condition in constant time, we record in each node the height of that node. An alternative to keeping track of the height information explicitly is to record in each node the difference in the heights of its two subtrees. In an AVL balanced tree, this difference is either -1, 0 or +1. Replace the height field of the AVL class defined in Program with one called diff and rewrite the various methods accordingly.

7. The M-way tree implementation given in Section is an internal data structure--it is assumed that all the nodes reside in the main memory. However, the motivation for using an M-way tree is that it is an efficient way to organize an external data structure--one that is stored on disk. Design,
implement and test an external $M$-way tree implementation.
In this chapter we consider priority queues. A priority queue is essentially a list of items in which each item has associated with it a priority. In general, different items may have different priorities and we speak of one item having a higher priority than another. Given such a list we can determine which is the highest (or the lowest) priority item in the list. Items are inserted into a priority queue in any, arbitrary order. However, items are withdrawn from a priority queue in order of their priorities starting with the highest priority item first.

For example, consider the software which manages a printer. In general, it is possible for users to submit documents for printing much more quickly than it is possible to print them. A simple solution is to place the documents in a FIFO queue (Chapter [ ]). In a sense this is fair, because the documents are printed on a first-come, first-served basis.

However, a user who has submitted a short document for printing will experience a long delay when much longer documents are already in the queue. An alternative solution is to use a priority queue in which the shorter a document, the higher its priority. By printing the shortest documents first, we reduce the level of frustration experienced by the users. In fact, it can be shown that printing documents in order of their length minimizes the average time a user waits for his document.

Priority queues are often used in the implementation of algorithms. Typically the problem to be solved consists of a number of subtasks and the solution strategy involves prioritizing the subtasks and then performing those subtasks in the order of their priorities. For example, in Chapter [ ] we show how a priority queue can improve the performance of backtracking algorithms, in Chapter [ ] we will see how a priority queue can be used in sorting and in Chapter [ ] several graph algorithms that use a priority queue are discussed.
Heaps and Priority Queues

- Leftist Heaps
- Binomial Queues
- Applications
- Exercises
- Projects

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Basics

A priority queue is a container which provides the following three operations:

**Enqueue**
used to put objects into the container;

**Min**
accesses the smallest object in the container; and

**DequeueMin**
removes the smallest object from the container.

A priority queue is used to store a finite set of keys drawn from a totally ordered set of keys $K$. As distinct from search trees, duplicate keys *are* allowed in priority queues.

Program defines the `PriorityQueue` interface. The `PriorityQueue` interface extends the `Container` interface defined in Program. In addition to the inherited methods, the `PriorityQueue` interface comprises the three methods listed above.

```csharp
public interface PriorityQueue : Container
{
    void Enqueue(ComparableObject obj);
    ComparableObject Min { get; }
    ComparableObject DequeueMin();
}
```

**Program:** PriorityQueue interface.

Program defines the `MergeablePriorityQueue` interface. The `MergeablePriorityQueue` interface extends the `PriorityQueue` interface defined in Program. A *mergeable priority queue* is one which provides the ability to merge efficiently two priority queues into one. Of course it is always possible to merge two priority queues by dequeuing the elements of one queue and enqueueing them in the other. However, the mergeable priority queue implementations we will consider allow more efficient merging than this.
It is possible to implement the required functionality using data structures that we have already considered. For example, a priority queue can be implemented simply as a list. If an unsorted list is used, enqueueing can be accomplished in constant time. However, finding the minimum and removing the minimum each require $O(n)$ time where $n$ is the number of items in the queue. On the other hand, if a sorted list is used, finding the minimum and removing it is easy--both operations can be done in constant time. However, enqueueing an item in a sorted list requires $O(n)$ time.

Another possibility is to use a search tree. For example, if an AVL tree is used to implement a priority queue, then all three operations can be done in $O(\log n)$ time. However, search trees provide more functionality than we need. Search trees support finding the largest item with $\text{Max}$, deletion of arbitrary objects with $\text{Withdraw}$, and the ability to visit in order all the contained objects via $\text{DepthFirstTraversal}$. All these operations can be done as efficiently as the priority queue operations. Because search trees support more methods than we really need for priority queues, it is reasonable to suspect that there are more efficient ways to implement priority queues. And indeed there are!

A number of different priority queue implementations are described in this chapter. All the implementations have one thing in common--they are all based on a special kind of tree called a min heap or simply a heap.

**Definition ((Min) Heap)** A (Min) Heap is a tree,

$$T = \{R, T_0, T_1, T_2, \ldots, T_{n-1}\},$$

with the following properties:

1. Every subtree of $T$ is a heap; and,
2. The root of $T$ is less than or equal to the root of every subtree of $T$. That is, $R \leq R_i$ for all $i$, $0 \leq i < n$, where $R_i$ is the root of $T_i$.

According to Definition, the key in each node of a heap is less than or equal to the roots of all the subtrees of that node. Therefore, by induction, the key in each node is less than or equal to all the keys contained in the subtrees of that node. Note, however, that the definition says nothing about the relative
ordering of the keys in the subtrees of a given node. For example, in a binary heap either the left or the right subtree of a given node may have the larger key.
Binary Heaps

A binary heap is a heap-ordered binary tree which has a very special shape called a complete tree. As a result of its special shape, a binary heap can be implemented using an array as the underlying foundational data structure. Array subscript calculations are used to find the parent and the children of a given node in the tree. And since an array is used, the storage overhead associated with the subtree fields contained in the nodes of the trees is eliminated.
Complete Trees

The preceding chapter introduces the idea of a perfect tree (see Definition 1). Complete trees and perfect trees are closely related, yet quite distinct. As pointed out in the preceding chapter, a perfect binary tree of height \( h \) has exactly \( n = 2^{h+1} - 1 \) internal nodes. Since, the only permissible values of \( n \) are

\[
0, 1, 3, 7, 15, 31, \ldots, 2^{h+1} - 1, \ldots,
\]

there is no perfect binary tree which contains, say 2, 4, 5, or 6 nodes.

However, we want a data structure that can hold an arbitrary number of objects so we cannot use a perfect binary tree. Instead, we use a complete binary tree, which is defined as follows:

**Definition (Complete Binary Tree)** A complete binary tree of height \( h \geq 0 \), is a binary tree \( \{R, T_L, T_R\} \) with the following properties.

1. If \( h=0 \), \( T_L = \emptyset \) and \( T_R = \emptyset \).
2. For \( h>0 \) there are two possibilities:
   1. \( T_L \) is a perfect binary tree of height \( h-1 \) and \( T_R \) is a complete binary tree of height \( h-1 \); or
   2. \( T_L \) is a complete binary tree of height \( h-1 \) and \( T_R \) is a perfect binary tree of height \( h-2 \).

Figure 1 shows an example of a complete binary tree of height four. Notice that the left subtree of node 1 is a complete binary tree of height three; and the right subtree is a perfect binary tree of height two. This corresponds to case 2 (b) of Definition 1. Similarly, the left subtree of node 2 is a perfect binary tree of height two; and the right subtree is a complete binary tree of height two. This corresponds to case 2 (a) of Definition 1.
Does there exist an complete binary with exactly \( n \) nodes for every integer \( n > 0 \)? The following theorem addresses this question indirectly by defining the relationship between the height of a complete tree and the number of nodes it contains.

**Theorem** A complete binary tree of height \( h \geq 0 \) contains at least \( 2^h \) and at most \( 2^{h+1} - 1 \) nodes.

**Proof** First, we prove the lower bound by induction. Let \( m_h \) be the minimum number of nodes in a complete binary tree of height \( h \). To prove the lower bound we must show that \( m_h = 2^h \).

**Base Case** There is exactly one node in a tree of height zero. Therefore, \( m_0 = 1 = 2^0 \).

**Inductive Hypothesis** Assume that \( m_h = 2^h \) for \( h = 0, 1, 2, \ldots, k \), for some \( k \geq 0 \). Consider the complete binary tree of height \( k+1 \) which has the smallest number of nodes. Its left subtree is a complete tree of height \( k \) having the smallest number of nodes and its right subtree is a perfect tree of height \( k-1 \).

From the inductive hypothesis, there are \( 2^k \) nodes in the left subtree and there are exactly \( 2^{(k-1)+1} - 1 \) nodes in the perfect right subtree. Thus,

\[
m_{k+1} = 1 + 2^k + 2^{(k-1)+1} - 1 = 2^{k+1}.
\]
Therefore, by induction $m_h = 2^h$ for all $h \geq 0$, which proves the lower bound.

Next, we prove the upper bound by induction. Let $M_h$ be the maximum number of nodes in a complete binary tree of height $h$. To prove the upper bound we must show that $M_h = 2^{h+1} - 1$.

**Base Case** There is exactly one node in a tree of height zero. Therefore, $M_0 = 1 = 2^1 - 1$.

**Inductive Hypothesis** Assume that $M_h = 2^{h+1} - 1$ for $h = 0, 1, 2, \ldots, k$, for some $k \geq 0$.

Consider the complete binary tree of height $k+1$ which has the largest number of nodes. Its left subtree is a perfect tree of height $k$ and its right subtree is a complete tree of height $k$ having the largest number of nodes.

There are exactly $2^{k+1} - 1$ nodes in the perfect left subtree. From the inductive hypothesis, there are $2^{k+1} - 1$ nodes in the right subtree. Thus,

$$M_{k+1} = 1 + 2^{k+1} - 1 + 2^{k+1} - 1 = 2^{(k+1)+1} - 1.$$ 

Therefore, by induction $M_h = 2^{h+1} - 1$ for all $h \geq 0$, which proves the upper bound.

It follows from Theorem that there exists exactly one complete binary tree that contains exactly $n$ internal nodes for every integer $n \geq 0$. It also follows from Theorem that the height of a complete binary tree containing $n$ internal nodes is $h = \lceil \log_2 n \rceil$.

Why are we interested in complete trees? As it turns out, complete trees have some useful characteristics. For example, in the preceding chapter we saw that the internal path length of a tree, i.e., the sum of the depths of all the internal nodes, determines the average time for various operations. A complete binary tree has the nice property that it has the smallest possible internal path length:

**Theorem** The internal path length of a binary tree with $n$ nodes is at least as big as the internal path length of a complete binary tree with $n$ nodes.

**Proof** Consider a binary tree with $n$ nodes that has the smallest possible internal path length. Clearly, there can only be one node at depth zero—the root. Similarly, at most two nodes can be at depth one; at most four nodes can be at depth two; and so on. Therefore, the internal path length of a tree with $n$ nodes is always at least as large as the sum of the first $n$ terms in the series
But this summation is precisely the internal path length of a complete binary tree!

Since the depth of the average node in a tree is obtained by dividing the internal path length of the tree by \( n \), Theorem \( \square \) tells us that complete trees are the best possible in the sense that the average depth of a node in a complete tree is the smallest possible. But how small is small? That is, does the average depth grow logarithmically with \( n \). The following theorem addresses this question:

**Theorem** The internal path length of a complete binary tree with \( n \) nodes is

\[
\sum_{i=1}^{n} \lfloor \log_2 i \rfloor = (n + 1) \lfloor \log_2 (n + 1) \rfloor - 2 \lfloor \log_2 (n+1) \rfloor + 1 + 2.
\]

**Proof** The proof of Theorem \( \square \) is left as an exercise for the reader (Exercise \( \square \)).

From Theorem \( \square \) we may conclude that the internal path length of a complete tree is \( \mathcal{O}(n \log n) \). Consequently, the depth of the average node in a complete tree is \( \mathcal{O}(\log n) \).
Complete N-ary Trees

The definition for complete binary trees can be easily extended to trees with arbitrary fixed degree \( N \geq 2 \) as follows:

**Definition (Complete N-ary Tree)** A complete N-ary tree \( T \) of height \( h \geq 0 \), is an N-ary tree \( \{ R, T_0, T_1, T_2, \ldots, T_{N-1} \} \) with the following properties.

1. If \( h=0 \), \( T_i = \emptyset \) for all \( i \), \( 0 \leq i < N \).
2. For \( h>0 \) there exists a \( j \), \( 0 \leq j < N \) such that
   1. \( T_i \) is a perfect binary tree of height \( h-1 \) for all \( i \), \( 0 \leq i < j \);
   2. \( T_j \) is a complete binary tree of height \( h-1 \); and,
   3. \( T_i \) is a perfect binary tree of height \( h-2 \) for all \( i:j<i<N \).

Note that while it is expressed in somewhat different terms, the definition of a complete N-ary tree is consistent with the definition of a binary tree for \( N=2 \). Figure shows an example of a complete ternary (\( N=3 \)) tree.

---

**Figure**: A complete ternary tree.

Informally, a complete tree is a tree in which all the levels are full except for the bottom level and the bottom level is filled from left to right. For example in Figure, the first three levels are full. The
Complete N-ary Trees

fourth level which comprises nodes 14-21 is partially full and has been filled from left to right.

The main advantage of using complete binary trees is that they can be easily stored in an array. Specifically, consider the nodes of a complete tree numbered consecutively in *level-order* as they are in Figures [1] and [2]. There is a simple formula that relates the number of a node with the number of its parent and the numbers of its children.

Consider the case of a complete binary tree. The root node is node 1 and its children are nodes 2 and 3. In general, the children of node $i$ are $2i$ and $2i+1$. Conversely, the parent of node $i$ is $\lfloor i/2 \rfloor$. Figure [3] illustrates this idea by showing how the complete binary tree shown in Figure [2] is mapped into an array.

![Array representation of a complete binary tree.](http://www.brpreiss.com/books/opus6/html/page357.html)

**Figure:** Array representation of a complete binary tree.

A remarkable characteristic of complete trees is that filling the bottom level from left to right corresponds to adding elements at the end of the array! Thus, a complete tree containing $n$ nodes occupies the first $n$ consecutive array positions.

The array subscript calculations given above can be easily generalized to complete $N$-ary trees. Assuming that the root occupies position 1 of the array, its $N$ children occupy positions 2, 3, ..., $N+1$. In general, the children of node $i$ occupy positions

$$N(i-1) + 2, N(i-1) + 3, N(i-1) + 4, \ldots, Ni + 1,$$

and the parent of node $i$ is found at

$$\left\lfloor (i - 1)/N \right\rfloor.$$
Implementation

A binary heap is a heap-ordered complete binary tree which is implemented using an array. In a heap the smallest key is found at the root and since the root is always found in the first position of the array, finding the smallest key is a trivial operation in a binary heap.

In this section we describe the implementation of a priority queue as a binary heap. As shown in Figure, we define a concrete class called BinaryHeap for this purpose.

```
1 public class BinaryHeap : AbstractContainer, PriorityQueue
2 {
3     protected ComparableObject[] array;
4     // ...
5 }
```

Program: BinaryHeap fields.
• **Fields**
• **Constructor and Purge Methods**
Fields

The BinaryHeap class has a rather simple implementation. In particular, it requires only a single field, array, which is declared as an array of ComparableObjects. This array is used to hold the objects which are contained in the binary tree. When there are \( n \) items in the heap, those items occupy array positions 1, 2, \(...\), \( n \).
Constructor and Purge Methods

Program defines the BinaryHeap constructor. The constructor takes a single argument of type int which specifies the maximum capacity of the binary heap. The constructor allocates an array of the specified size plus one. This is done because array position zero will not be used. The running time of the constructor is \( O(n) \), where \( n \) is the maximum length of the priority queue.

```csharp
public class BinaryHeap : AbstractContainer, PriorityQueue
{
    protected ComparableObject[] array;

    public BinaryHeap(int length)
    {
        array = new ComparableObject[length + 1];
    }

    public override void Purge()
    {
        while (count > 0)
        {
            array[count--] = null;
        }
        // ...
    }
}
```

Program: BinaryHeap class constructor and Purge methods.

The purpose of the Purge method is to make the priority queue empty. The Purge method assigns the value null to the array positions one-by-one. Clearly the worst-case running time for the Purge method is \( O(n) \), where \( n \) is the maximum length of the priority queue.
Putting Items into a Binary Heap

There are two requirements which must be satisfied when an item is inserted in a binary heap. First, the resulting tree must have the correct shape. Second, the tree must remain heap-ordered. Figure illustrates the way in which this is done.

Since the resulting tree must be a complete tree, there is only one place in the tree where a node can be added. That is, since the bottom level must be filled from left to right, the node must be added at the next available position in the bottom level of the tree as shown in Figure (a).

Figure: Inserting an item into a binary heap.

In this example, the new item to be inserted has the key 2. Note that we cannot simply drop the new item into the next position in the complete tree because the resulting tree is no longer heap ordered. Instead, the hole in the heap is moved toward the root by moving items down in the heap as shown in Figure (b) and (c). The process of moving items down terminates either when we reach the root of the tree or when the hole has been moved up to a position in which when the new item is inserted the result is a
Program gives the code for inserting an item in a binary heap. The Enqueue method of the BinaryHeap class takes as its argument the item to be inserted in the heap. If the priority queue is full an exception is thrown. Otherwise, the item is inserted as described above.

```java
public class BinaryHeap : AbstractContainer, PriorityQueue
{
    protected ComparableObject[] array;

    public virtual void Enqueue(ComparableObject obj)
    {
        if (count == array.Length - 1)
            throw new ContainerFullException();
        count++;
        int i = count;
        while (i > 1 && array[i/2] > obj)
        {
            array[i] = array[i / 2];
            i /= 2;
        }
        array[i] = obj;
    }
}
```

Program: BinaryHeap class Enqueue method.

The implementation of the algorithm is actually remarkably simple. Lines 11-15 move the hole in the heap up by moving items down. When the loop terminates, the new item can be inserted at position $i$. Therefore, the loop terminates either at the root, $i=1$, or when the key in the parent of $i$, which is found at position $\lfloor i/2 \rfloor$, is smaller than the item to be inserted.

Notice too that a good optimizing compiler will recognize that the subscript calculations involve only division by two. Therefore, the divisions can be replaced by bitwise right shifts which usually run much more quickly.

Since the depth of a complete binary tree with $n$ nodes is $\lceil \log_2 n \rceil$, the worst case running time for the Enqueue operation is
\[ \lfloor \log_2 n \rfloor T_{\text{act}} + O(\log n), \]

where \( T_{\text{act}} \) is the time required to compare to objects. If \( T_{\text{act}} = \Theta(1) \), the \texttt{Enqueue} operation is simply \( O(\log n) \) in the worst case.
Removing Items from a Binary Heap

The `DequeueMin` method removes from a priority queue the item having the smallest key. In order to remove the smallest item, it needs first to be located. Therefore, the `DequeueMin` operation is closely related to `Min`.

The smallest item is always at the root of a min heap. Therefore, the `Min` operation is trivial. Program □ gives the code for the `Min` property `get` accessor of the `BinaryHeap` class. Assuming that no exception is thrown, the running time of the accessor is clearly $O(1)$.

```csharp
public class BinaryHeap : AbstractContainer, PriorityQueue
{
    protected ComparableObject[] array;

    public virtual ComparableObject Min
    {
        get
        {
            if (count == 0)
                throw new ContainerEmptyException();
            return array[1];
        }
        // ...
    }
}
```

**Program:** `BinaryHeap` class `Min` property.

Since the bottom row of a complete tree is filled from left to right as items are added, it follows that the bottom row must be emptied from right to left as items are removed. So, we have a problem: The datum to be removed from the heap by `DequeueMin` is in the root, but the node to be removed from the heap is in the bottom row.

Figure □ (a) illustrates the problem. The `DequeueMin` operation removes the key 2 from the heap, but
it is the node containing key 6 that must be removed from the tree to make it into a complete tree again. When key 2 is removed from the root, a hole is created in the tree as shown in Figure (b).

The trick is to move the hole down in the tree to a point where the left-over key, in this case the key 6, can be reinserted into the tree. To move a hole down in the tree, we consider the children of the empty node and move up the smallest key. Moving up the smallest key ensures that the result will be a min heap.

The process of moving up continues until either the hole has been pushed down to a leaf node, or until the hole has been pushed to a point where the left over key can be inserted into the heap. In the example shown in Figure (b)-(c), the hole is pushed from the root node to a leaf node where the key 6 is ultimately placed is shown in Figure (d).

Program gives the code for the DequeueMin method of the BinaryHeap class. This method implements the deletion algorithm described above. The main loop (lines 13-23) moves the hole in the tree down by moving up the child with the smallest key until either a leaf node is reached or until the hole has been moved down to a point where the last element of the array can be reinserted.
public class BinaryHeap : AbstractContainer, PriorityQueue
{
    protected ComparableObject[] array;

    public virtual ComparableObject DequeueMin()
    {
        if (count == 0)
            throw new ContainerEmptyException();
        ComparableObject result = array[1];
        ComparableObject last = array[count];
        --count;
        int i = 1;
        while (2 * i < count + 1)
        {
            int child = 2 * i;
            if (child + 1 < count + 1
                && array[child + 1] < array[child])
                child += 1;
            if (last <= array[child])
                break;
            array[i] = array[child];
            i = child;
        }
        array[i] = last;
        return result;
    }
    // ...
}

Program: BinaryHeap class DequeueMin method.

In the worst case, the hole must be pushed from the root to a leaf node. Each iteration of the loop makes at most two object comparisons and moves the hole down one level. Therefore, the running time of the DequeueMin operation is

$$[\log_2 n](T_{ilT} + T_{ile}) + O(\log n),$$

where \( n = \text{count} \) is the number of items in the heap. If \( T_{ilT} = O(1) \) and \( T_{ile} = O(1) \), the
Removing Items from a Binary Heap

DequeMin operation is simply $O(\log n)$ in the worst case.
Leftist Heaps

A leftist heap is a heap-ordered binary tree which has a very special shape called a leftist tree. One of the nice properties of leftist heaps is that it is possible to merge two leftist heaps efficiently. As a result, leftist heaps are suited for the implementation of mergeable priority queues.
Leftist Trees

A leftist tree is a tree which tends to "lean" to the left. The tendency to lean to the left is defined in terms of the shortest path from the root to an external node. In a leftist tree, the shortest path to an external node is always found on the right.

Every node in binary tree has associated with it a quantity called its null path length which is defined as follows:

**Definition (Null Path and Null Path Length)**

Consider an arbitrary node $x$ in some binary tree $T$. The null path of node $x$ is the shortest path in $T$ from $x$ to an external node of $T$.

The null path length of node $x$ is the length of its null path.

Sometimes it is convenient to talk about the null path length of an entire tree rather than of a node:

**Definition (Null Path Length of a Tree)**

The null path length of an empty tree is zero and the null path length of a non-empty binary tree $T = \{R, T_L, T_R\}$ is the null path length its root $R$.

When a new node or subtree is attached to a given tree, it is usually attached in place of an external node. Since the null path length of a tree is the length of the shortest path from the root of the tree to an external node, the null path length gives a lower bound on the cost of insertion. For example, the running time for insertion in a binary search tree, Program \[\text{Program}\], is at least

$$dT{\text{compareTo}} + \Omega(d)$$

where $d$ is the null path length of the tree.

A leftist tree is a tree in which the shortest path to an external node is always on the right. This informal idea is defined more precisely in terms of the null path lengths as follows:
Leftist Trees

**Definition (Leftist Tree)** A *leftist tree* is a binary tree $T$ with the following properties:

1. Either $T = \emptyset$, or
2. $T = \{R, TL, TR\}$, where both $TL$ and $TR$ are leftist trees which have null path lengths $d_L$ and $d_R$, respectively, such that

   $$d_L \geq d_R.$$ 

Figure shows an example of a leftist heap. A leftist heap is simply a heap-ordered leftist tree. The external depth of the node is shown to the right of each node in Figure . The figure clearly shows that it is not necessarily the case in a leftist tree that the number of nodes to the left of a given node is greater than the number to the right. However, it is always the case that the null path length on the left is greater than or equal to the null path length on the right for every node in the tree.

---

**Figure**: A leftist heap.

The reason for our interest in leftist trees is illustrated by the following theorems:

**Theorem** Consider a leftist tree $T$ which contains $n$ internal nodes. The path leading from the root of $T$ downwards to the rightmost external node contains at most $\lceil \log_2(n + 1) \rceil$ nodes.

**Proof** Assume that $T$ has null path length $d$. Then $T$ must contain at least $2^{d-1}$ leaves. Otherwise,
there would be a shorter path than \( d \) from the root of \( T \) to an external node.

A binary tree with exactly \( l \) leaves has exactly \( l-1 \) non-leaf internal nodes. Since \( T \) has at least \( 2^d - 1 \) leaves, it must contain at least \( n \geq 2^d - 1 \) internal nodes altogether. Therefore, \( d \leq \log_2(n + 1) \).

Since \( T \) is a leftist tree, the shortest path to an external node must be the path on the right. Thus, the length of the path to the rightmost external is at most \( \lfloor \log_2(n + 1) \rfloor \).

There is an interesting dichotomy between AVL balanced trees and leftist trees. The shape of an AVL tree satisfies the AVL balance condition which stipulates that the difference in the heights of the left and right subtrees of every node may differ by at most one. The effect of AVL balancing is to ensure that the height of the tree is \( \mathcal{O}(\log n) \).

On the other hand, leftist trees have an "imbalance condition" which requires the null path length of the left subtree to be greater than or equal to that of the right subtree. The effect of the condition is to ensure that the length of the right path in a leftist tree is \( \mathcal{O}(\log n) \). Therefore, by devising algorithms for manipulating leftist heaps which only follow the right path of the heap, we can achieve running times which are logarithmic in the number of nodes.

The dichotomy also extends to the structure of the algorithms. For example, an imbalance sometimes results from an insertion in an AVL tree. The imbalance is rectified by doing rotations. Similarly, an insertion into a leftist tree may result in a violation of the "imbalance condition." That is, the null path length of the right subtree of a node may become greater than that of the left subtree. Fortunately, it is possible to restore the proper condition simply by swapping the left and right subtrees of that node.
Implementation

This section presents an implementation of leftist heaps that is based on the binary tree implementation described in Section [Section]. Program [Program] introduces the LeftistHeap class. The LeftistHeap class extends the BinaryTree class introduced in Program [Program] and it implements the MergeablePriorityQueue interface defined in Program [Program].

```csharp
public class LeftistHeap : BinaryTree, MergeablePriorityQueue
{
    protected int nullPathLength;

    // ...
}
```

Program: LeftistHeap fields.

- Fields
Fields

Since a leftist heap is a heap-ordered binary tree, it inherits from the `BinaryTree` base class the three fields: `key`, `left`, and `right`. The `key` refers to the object contained in the given node and the `left` and `right` fields refer to the left and right subtrees of the given node, respectively. In addition, the field `nullPathLength` records the null path length of the given node. By recording the null path length in the node, it is possible to check the leftist heap balance condition in constant time.
Merging Leftist Heaps

In order to merge two leftist heaps, say $h_1$ and $h_2$, declared as follows

```csharp
MergeablePriorityQueue h1 = new LeftistHeap();
MergeablePriorityQueue h2 = new LeftistHeap();
```

we invoke the `Merge` method like this:

```csharp
h1.Merge(h2);
```

The effect of the `Merge` method is to take all the nodes from $h_2$ and to attach them to $h_1$, thus leaving $h_2$ as the empty heap.

In order to achieve a logarithmic running time, it is important for the `Merge` method to do all its work on the right sides of $h_1$ and $h_2$. It turns out that the algorithm for merging leftist heaps is actually quite simple.

To begin with, if $h_1$ is the empty heap, then we can simply swap the contents of $h_1$ and $h_2$. Otherwise, let us assume that the root of $h_2$ is larger than the root of $h_1$. Then we can merge the two heaps by recursively merging $h_2$ with the right subheap of $h_1$. After doing so, it may turn out that the right subheap of $h_1$ now has a larger null path length than the left subheap. This we rectify by swapping the left and right subheaps so that the result is again leftist. On the other hand, if $h_2$ initially has the smaller root, we simply exchange the roles of $h_1$ and $h_2$ and proceed as above.

Figure illustrates the merge operation. In this example, we wish to merge the two trees $T_1$ and $T_2$ shown in Figure (a). Since $T_2$ has the larger root, it is recursively merged with the right subtree of $T_1$. The result of that merge replaces the right subtree of $T_1$ as shown in Figure (b). Since the null path length of the right subtree is now greater than the left, the subtrees of $T_1$ are swapped giving the leftist heap shown in Figure (c).
Figure: Merging leftist heaps.
Program □ gives the code for the Merge method of the LeftistHeap class. The Merge method makes use of two other methods, SwapContentsWith and SwapSubtrees. The SwapContentsWith method takes as its argument a leftist heap, and exchanges all the contents (key and subtrees) of this heap with the given one. The SwapSubtrees method exchanges the left and right subtrees of this node. The implementation of these routines is trivial and is left as a project for the reader (Project □). Clearly, the worst-case running time for each of these routines is $O(1)$.

The Merge method only visits nodes on the rightmost paths of the trees being merged. Suppose we are merging two trees, say $T_1$ and $T_2$, with null path lengths $d_1$ and $d_2$, respectively. Then the running time of the Merge method is

$$(d_1 - 1 + d_2 - 1)T_{\text{iact}} + O(d_1 + d_2)$$

where $T_{\text{iact}}$ is time required to compare two keys. If we assume that the time to compare two keys is a constant, then we get $O(\log n_1 + \log n_2)$, where $n_1$ and $n_2$ are the number of internal nodes in trees $T_1$ and $T_2$, respectively.
public class LeftistHeap : BinaryTree, MergeablePriorityQueue
{
    protected int nullPathLength;

    public virtual void Merge(MergeablePriorityQueue queue)
    {
        LeftistHeap arg = (LeftistHeap)queue;
        if (IsEmpty)
            SwapContentsWith(arg);
        else if (!arg.IsEmpty)
        {
            if (Key > arg.Key)
                SwapContentsWith(arg);
            Right.Merge(arg);
            if (Left.nullPathLength < Right.nullPathLength)
                SwapSubtrees();
            nullPathLength = 1 + Math.Min(
                Left.nullPathLength, Right.nullPathLength);
        }
        // ...
    }
}

Program: LeftistHeap class Merge method.
Putting Items into a Leftist Heap

The Enqueue method of the LeftistHeap class is used to put items into the heap. Enqueue is easily implemented using the Merge operation. That is, to enqueue an item in a given heap, we simply create a new heap containing the one item to be enqueued and merge it with the given heap. The algorithm to do this is shown in Program □.

```csharp
public class LeftistHeap : BinaryTree, MergeablePriorityQueue
{
    protected int nullPathLength;

    public void Enqueue(ComparableObject obj)
    {
        Merge(new LeftistHeap(obj));
    }

    // ...
}
```

Program: LeftistHeap class Enqueue method.

The expression for the running time for the Insert operation follows directly from that of the Merge operation. That is, the time required for the Insert operation in the worst case is

\[(d - 1)T(\text{insert}) + O(d),\]

where \(d\) is the null path length of the heap into which the item is inserted. If we assume that two keys can be compared in constant time, the running time for Insert becomes simply \(O(\log n)\), where \(n\) is the number of nodes in the tree into which the item is inserted.
Removing Items from a Leftist Heap

The Min property get accessor locates the item with the smallest key in a given priority queue and the DequeueMin method removes it from the queue. Since the smallest item in a heap is found at the root, the Min property get accessor is easy to implement. Program □ shows how it can be done. Clearly, the running time of the accessor is $O(1)$.

```csharp
public class LeftistHeap : BinaryTree, MergeablePriorityQueue
{
    protected int nullPathLength;

    public virtual CompareableObject Min
    {
        get
        {
            if (IsEmpty)
                throw new ContainerEmptyException();
            return Key;
        }
    } // ...
}
```

Program: LeftistHeap class Min property.

Since the smallest item in a heap is at the root, the DequeueMin operation must delete the root node. Since a leftist heap is a binary heap, the root has at most two children. In general when the root is deleted, we are left with two non-empty leftist heaps. Since we already have an efficient way to merge leftist heaps, the solution is to simply merge the two children of the root to obtain a single heap again! Program □ shows how the DequeueMin operation of the LeftistHeap class can be implemented.
Removing Items from a Leftist Heap

Program: LeftistHeap class DequeueMin method.

The running time of Program □ is determined by the time required to merge the two children of the root (line 17) since the rest of the work in DequeueMin can be done in constant time. Consider the running time to delete the root of a leftist heap \( T \) with \( n \) internal nodes. The running time to merge the left and right subtrees of \( T \)

\[
(d_L - 1 + d_R - 1)T_{\text{isct}} + O(d_L + d_R),
\]

where \( d_L \) and \( d_R \) are the null path lengths of the left and right subtrees \( T \), respectively. In the worst case, \( d_R = 0 \) and \( d_L = \lceil \log_2 n \rceil \). If we assume that \( T_{\text{isct}} = O(1) \), the running time for DequeueMin is \( O(\log n) \).
Binomial Queues

A binomial queue is a priority queue that is implemented not as a single tree but as a collection of heap-ordered trees. A collection of trees is called a forest. Each of the trees in a binomial queue has a very special shape called a binomial tree. Binomial trees are general trees. That is, the maximum degree of a node is not fixed.

The remarkable characteristic of binomial queues is that the merge operation is similar in structure to binary addition. That is, the collection of binomial trees that make up the binomial queue is like the set of bits that make up the binary representation of a non-negative integer. Furthermore, the merging of two binomial queues is done by adding the binomial trees that make up that queue in the same way that the bits are combined when adding two binary numbers.
Binomial Trees

A binomial tree is a general tree with a very special shape:

**Definition (Binomial Tree)** The binomial tree of order \( k \geq 0 \) with root \( R \) is the tree \( B_k \) defined as follows

1. If \( k = 0 \), \( B_k = B_0 = \{ R \} \). That is, the binomial tree of order zero consists of a single node, \( R \).
2. If \( k > 0 \), \( B_k = \{ R, B_0, B_1, \ldots, B_{k-1} \} \). That is, the binomial tree of order \( k > 0 \) comprises the root \( R \), and \( k \) binomial subtrees, \( B_0, B_1, \ldots, B_{k-1} \).

Figure shows the first five binomial trees, \( B_0, B_1, B_2, B_3, B_4 \). It follows directly from Definition that the root of \( B_k \), the binomial tree of order \( k \), has degree \( k \). Since \( k \) may arbitrarily large, so too can the degree of the root. Furthermore, the root of a binomial tree has the largest fanout of any of the nodes in that tree.
The number of nodes in a binomial tree of order $k$ is a function of $k$:

**Theorem** The binomial tree of order $k$, $B_k$, contains $2^k$ nodes.

**Proof (By induction).** Let $n_k$ be the number of nodes in $B_k$, a binomial tree of order $k$.

**Base Case** By definition, $B_0$ consists a single node. Therefore $n_0 = 1 = 2^0$.

**Inductive Hypothesis** Assume that $n_k = 2^k$ for $k = 0, 1, 2, \ldots, l$, for some $l \geq 0$. Consider the binomial tree of order $l+1$:

$$B_{l+1} = \{R, B_0, B_1, B_2, \ldots, B_l\}.$$
Therefore the number of nodes in $B_{l+1}$ is given by

$$n_{l+1} = 1 + \sum_{i=0}^{l} n_i$$

$$= 1 + \sum_{i=0}^{l} 2^i$$

$$= 1 + \frac{2^{l+1} - 1}{2 - 1}$$

$$= 2^{l+1}.$$

Therefore, by induction on $l$, $n_k = 2^k$ for all $k \geq 0$.

It follows from Theorem that binomial trees only come in sizes that are a power of two. That is, $n_k \in \{1, 2, 4, 8, 16, \ldots\}$. Furthermore, for a given power of two, there is exactly one shape of binomial tree.

**Theorem** The height of $B_k$, the binomial tree of order $k$, is $k$.

**Proof** (By induction). Let $h_k$ be the height of $B_k$, a binomial tree of order $k$.

**Base Case** By definition, $B_0$ consists of a single node. Therefore $h_0 = 0$.

**Inductive Hypothesis** Assume that $h_k = k$ for $k = 0, 1, 2, \ldots, l$, for some $l \geq 0$. Consider the binomial tree of order $l+1$:

$$B_{l+1} = \{R, B_0, B_1, B_2, \ldots, B_l\}.$$

Therefore the height $B_{l+1}$ is given by

$$h_{l+1} = 1 + \max_{0 < i \leq l} h_i$$

$$= 1 + \max_{0 < i \leq l} i$$

$$= l + 1.$$
Therefore, by induction on $l$, $h_k = k$ for all $k \geq 0$.

Theorem □ tells us that the height of a binomial tree of order $k$ is $k$ and Theorem □ tells us that the number of nodes is $n_k = 2^k$. Therefore, the height of $B_k$ is exactly $O(\log n)$.

Figure □ shows that there are two ways to think about the construction of binomial trees. The first way follows directly from the Definition □. That is, binomial $B_k$ consists of a root node to which the $k$ binomial trees $B_0, B_1, ..., B_{k-1}$ are attached as shown in Figure □ (a).

---

Figure: Two views of binomial tree $B_4$. 
Alternatively, we can think of $B_k$ as being comprised of two binomial trees of order $k-1$. For example, Figure (b) shows that $B_4$ is made up of two instances of $B_3$. In general, suppose we have two trees of order $k-1$, say $B_{k-1}^1$ and $B_{k-1}^2$, where $B_{k-1}^1 = \{R^1, B_0^1, B_1^1, B_2^1, \ldots, B_{k-2}^1\}$. Then we can construct a binomial tree of order $k$ by combining the trees to get

$$B_k = \{R^1, B_0^1, B_1^1, B_2^1, \ldots, B_{k-2}^1, B_{k-1}^2\}.$$ 

Why do we call $B_k$ a binomial tree? It is because the number of nodes at a given depth in the tree is determined by the binomial coefficient. And the binomial coefficient derives its name from the binomial theorem. And the binomial theorem tells us how to compute the $n^{th}$ power of a binomial. And a binomial is an expression which consists of two terms, such as $x+y$. That is why it is called a binomial tree!

**Theorem (Binomial Theorem)** The $n^{th}$ power of the binomial $x+y$ for $n \geq 0$ is given by

$$(x + y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i},$$

where $\binom{n}{i} = \frac{n!}{i!(n-i)!}$ is called the binomial coefficient.

**Proof** The proof of the binomial theorem is left as an exercise for the reader (Exercise ).

The following theorem gives the expression for the number of nodes at a given depth in a binomial tree:

**Theorem** The number of nodes at level $l$ in $B_k$, the binomial tree of order $k$, where $0 \leq l \leq k$, is given by the binomial coefficient $\binom{k}{l}$.

**Proof** (By induction). Let $n_k(l)$ be the number of nodes at level $l$ in $B_k$, a binomial tree of order $k$.

**Base Case** Since $B_0$ contains a single node, there is only one level in the tree, $l=0$, and exactly one node at that level. Therefore, $n_0(0) = 1 = \binom{0}{0}$.

**Inductive Hypothesis** Assume that $n_k(l) = \binom{k}{l}$ for $k = 0, 1, 2, \ldots, \hat{k}$, for some $\hat{k} \geq 0$. The
Binomial Trees

binomial tree of order $h+1$ is composed of two binomial trees of height $h$, one attached under the root of the other. Hence, the number of nodes at level $l$ in $B_{h+1}$ is equal to the number of nodes at level $l$ in $B_h$ plus the number of nodes at level $l-1$ in $B_h$:

\[
n_{h+1}(l) = n_h(l) + n_h(l - 1) \\
= \binom{h}{l} + \binom{h}{l - 1} \\
= \frac{h!}{(h - l)!(l - 1)!} + \frac{h!}{(h - (l - 1))!(l - 1)!} \\
= \frac{h!(h + 1 - l)}{(h + 1 - l)!(l - 1)!} + \frac{h! l}{(h + 1 - l)!(l - 1)!} \\
= \frac{h!(h + 1 - l) + h! l}{(h + 1 - l)!(l - 1)!} \\
= \frac{(h + 1)!}{(h + 1 - l)!!} \\
= \binom{h + 1}{l}
\]

Therefore by induction on $h$, $n_h(l) = \binom{h}{l}$. 

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Binomial Queues

If binomial trees only come in sizes that are powers of two, how do we implement a container which holds an arbitrary number of items \( n \) using binomial trees? The answer is related to the binary representation of the number \( n \). Every non-negative integer \( n \) can be expressed in binary form as

\[
n = \sum_{i=0}^{\lfloor \log_2 n \rfloor} b_i 2^i,
\]

where \( b_i \in \{0, 1\} \) is the \( i^{th} \) binary digit or bit in the representation of \( n \). For example, \( n=27 \) is expressed as the binary number \( 11011_2 \) because \( 27=16+8+2+1 \).

To make a container which holds exactly \( n \) items we use a collection of binomial trees. A collection of trees is called a forest. The forest contains binomial tree \( B_i \) if the \( i^{th} \) bit in the binary representation of \( n \) is a one. That is, the forest \( F_n \) which contains exactly \( n \) items is given by

\[
F_n = \{ B_i : b_i = 1 \},
\]

where \( b_i \) is determined from Equation (11.1). For example, the forest which contains 27 items is \( F_{27} = \{ B_4, B_3, B_1, B_0 \} \).

The analogy between \( F_n \) and the binary representation of \( n \) carries over to the merge operation. Suppose we have two forests, say \( F_n \) and \( F_m \). Since \( F_n \) contains \( n \) items and \( F_m \) contains \( m \) items, the combination of the two contains \( n+m \) items. Therefore, the resulting forest is \( F_{n+m} \).

For example, consider \( n=27 \) and \( m=10 \). In this case, we need to merge \( F_{27} = \{ B_4, B_3, B_1, B_0 \} \) with \( F_{10} = \{ B_3, B_1 \} \). Recall that two binomial trees of order \( k \) can be combined to obtain a binomial tree of order \( k+1 \). For example, \( B_1 + B_1 = B_2 \). But this is just like adding binary digits! In binary notation, the sum 27+10 is calculated like this:
The merging of $F_{27}$ and $F_{28}$ is done in the same way:

\begin{array}{c|c|c|c|c|c|c}
B_5 & B_4 & \emptyset & B_3 & B_2 & B_1 & B_0 \\
\hline
& B_3 & \emptyset & B_2 & \emptyset & B_1 & \emptyset \\
+ & & & B_3 & \emptyset & B_2 & \emptyset \\
& & & & B_5 & \emptyset & B_2 & \emptyset & B_1 & \emptyset & B_0 & F_{27} \\
\hline
& & B_3 & \emptyset & B_1 & \emptyset & F_{10} & F_{37} \\
\end{array}

Therefore, the result is $F_{37} = \{B_5, B_2, B_0\}$.
Implementation

- Heap-Ordered Binomial Trees
- Adding Binomial Trees
- Binomial Queues
- Fields
- AddTree and RemoveTree
- MinTree and Min Properties

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Heap-Ordered Binomial Trees

Since binomial trees are simply general trees with a special shape, we can make use of the 
GeneralTree class presented in Section □ to implement the BinomialTree class. (See Figure □).

Program □ introduces the BinomialQueue class and the inner class BinomialTree. The 
BinomialTree class extends the GeneralTree class introduced in Program □.

No new fields are declared in the BinomialTree class. Remember that the implementation of the 
GeneralTree class uses a linked list to contain the pointers to the subtrees, since the degree of a node 
in a general tree may be arbitrarily large. Also, the GeneralTree class already keeps track of the 
degree of a node in its degree field. Since the degree of the root node of a binomial tree of order \(k\) is \(k\), 
it is not necessary to keep track of the order explicitly. The degree variable serves this purpose nicely.

```csharp
public class BinomialQueue :
    AbstractContainer, MergeablePriorityQueue
{
    protected LinkedList treeList;

    protected class BinomialTree : GeneralTree
    {
        // ...
    }
    // ...
}
```

Program: BinomialTree class.
Adding Binomial Trees

Recall that we can combine two binomial trees of the same order, say $k$, into a single binomial tree of order $k+1$. Each of the two trees to be combined is heap-ordered. Since the smallest key is at the root of a heap-ordered tree, we know that the root of the result must be the smaller root of the two trees which are to be combined. Therefore, to combine the two trees, we simply attach the tree with the larger root under the root of the tree with the smaller root. For example, Figure (b) illustrates how two heap-ordered binomial trees of order two are combined into a single heap-ordered tree of order three.

![Figure: Adding binomial trees.](http://www.brpreiss.com/books/opus6/html/page375.html)

The `Add` method defined in Program (b) provides the means to combine two binomial trees of the same order. The `Add` method takes a `BinomialTree` and attaches the specified tree to this node. This is only permissible when both trees have the same order.
public class BinomialQueue :
        AbstractContainer, MergeablePriorityQueue
{
    protected LinkedList treeList;

    protected class BinomialTree : GeneralTree
    {
        internal virtual BinomialTree Add(BinomialTree tree)
        {
            if (degree != tree.degree)
                throw new ArgumentException(
                    "incompatible degrees");
            if (Key > tree.Key)
                SwapContentsWith(tree);
            AttachSubtree(tree);
            return this;
        }
    }
// ...
}
Binomial Queues

A binomial queue is a mergeable priority queue implemented as a forest of binomial trees. In this section we present a linked-list implementation of the forest. That is, the forest is represented using a linked list of binomial trees.

Program introduces the BinomialQueue class. The BinomialQueue class extends the AbstractContainer class introduced in Program and it implements the MergeablePriorityQueue interface defined in Program.

```csharp
public class BinomialQueue :
    AbstractContainer, MergeablePriorityQueue
{
    protected LinkedList treeList;

    public BinomialQueue()
    {
        treeList = new LinkedList();
    }

    // ...
}
```

Program: BinomialQueue fields.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Fields

The `BinomialQueue` class definition contains the single field `treeList`, which is an instance of the `LinkedList` class introduced in Program 1. The binomial trees contained in the linked list are stored in increasing order. That is, the binomial tree at the head of the list has the smallest order, and the binomial tree at the tail has the largest order.
AddTree and RemoveTree

The AddTree and RemoveTree methods of the BinomialQueue class facilitate the implementation of the various priority queue operations. These methods are defined in Program AddTree and RemoveTree methods. The AddTree method takes a BinomialTree and appends that tree to treeList. AddTree also adjusts the count in order to keep track of the number of items in the priority queue. It is assumed that the order of the tree which is added is larger than all the others in the list and, therefore, that it belongs at the end of the list. The running time of AddTree is clearly $O(1)$.

```csharp
class BinomialQueue : AbstractContainer, MergeablePriorityQueue
{
    protected LinkedList treeList;

    protected virtual void AddTree(BinomialTree tree)
    {
        treeList.Append(tree);
        count += tree.Count;
    }

    protected virtual void removeTree(BinomialTree tree)
    {
        treeList.Extract(tree);
        count -= tree.Count;
    }
    // ...
}
```

Program: BinomialQueue class AddTree and RemoveTree methods.

The RemoveTree method takes a binomial tree and removes it from the list. It is assumed that the specified tree is actually in the list. RemoveTree also adjust the count as required. The running time of RemoveTree depends on the position of the tree in the list. A binomial queue which contains exactly $n$ items altogether has at most $\left\lceil \log_2(n + 1) \right\rceil$ binomial trees. Therefore, the running time of
AddTree and RemoveTree

RemoveTree is $O(\log n)$ in the worst case.
MinTree and Min Properties

A binomial queue that contains \( n \) items consists of at most \( \lceil \log_2(n + 1) \rceil \) binomial trees. Each of these binomial trees is heap ordered. In particular, the smallest key in each binomial tree is at the root of that tree. So, we know that the smallest key in the queue is found at the root of one of the binomial trees, but we do not know which tree it is.

The \texttt{MinTree} property \texttt{get} accessor is used to determine which of the binomial trees in the queue has the smallest root. As shown in Program 4, the \texttt{MinTree} simply traverses the entire linked list to find the tree with the smallest key at its root. Since there are at most \( \lceil \log_2(n + 1) \rceil \) binomial trees, the worst-case running time of \texttt{MinTree} is

\[
(\lceil \log_2(n + 1) \rceil - 1)T_{\text{tree}} + O(\log n).
\]
MinTree and Min Properties

Program:

```java
public class BinomialQueue :
    AbstractContainer, MergeablePriorityQueue
{
    protected LinkedList treeList;

    protected virtual BinomialTree MinTree
    {
        get
        {
            BinomialTree minTree = null;
            for (LinkedList.Element ptr = treeList.Head;
                 ptr != null; ptr = ptr.Next)
            {
                BinomialTree tree = (BinomialTree)ptr.Datum;
                if (minTree == null || tree.Key < minTree.Key)
                    minTree = tree;
            }
            return minTree;
        }
    }

    public virtual ComparableObject Min
    {
        get
        {
            if (count == 0)
                throw new ContainerEmptyException();
            return MinTree.Key;
        }
    }
    // ...
```

Program: BinomialQueue class MinTree and Min methods.

Program also defines the Min property get accessor that returns the smallest key in the priority queue. The Min property uses the MinTree property to locate the tree with the smallest key at its root and returns that key. Clearly, the asymptotic running time of the Min accessor is the same as that of the MinTree accessor.
Merging Binomial Queues

Merging two binomial queues is like doing binary addition. For example, consider the addition of $F_{27}$ and $F_{10}$.

```
+   B_4 B_3 ∅ B_1 B_0 F_{27}
   B_3 ∅ B_1 ∅ F_{10}
   B_5 ∅ ∅ B_2 ∅ B_0 F_{37}
```

The usual algorithm for addition begins with the least-significant "bit." Since $F_{27}$ contains a $B_0$ tree and $F_{10}$ does not, the result is simply the $B_0$ tree from $F_{27}$.

In the next step, we add the $B_1$ from $F_{27}$ and the $B_1$ from $F_{10}$. Combining the two $B_1$s we get a $B_2$ which we carry to the next column. Since there are no $B_1$s left, the result does not contain any. The addition continues in a similar manner until all the columns have been added up.

Program gives an implementation of this addition algorithm. The Merge method of the BinomialQueue class takes a BinomialQueue and adds its subtrees to this binomial queue.

```java
public class BinomialQueue :
    AbstractContainer, MergeablePriorityQueue
{
    protected LinkedList treeList;

    public virtual void Merge(MergeablePriorityQueue queue)
    {
        BinomialQueue arg = (BinomialQueue)queue;
        LinkedList oldList = treeList;
        treeList = new LinkedList();
        count = 0;
        LinkedList.Element p = oldList.Head;
```
Merging Binomial Queues

```java
LinkedList.Element p = oldList.Head;
LinkedList.Element q = arg.treeList.Head;
BinomialTree carry = null;
for (int i = 0; p!=null || q!=null || carry!=null; ++i) {
    BinomialTree a = null;
    if (p != null) {
        BinomialTree tree = (BinomialTree)p.Datum;
        if (tree.Degree == i) {
            a = tree;
            p = p.Next;
        }
    }
    BinomialTree b = null;
    if (q != null) {
        BinomialTree tree = (BinomialTree)q.Datum;
        if (tree.Degree == i) {
            b = tree;
            q = q.Next;
        }
    }
    BinomialTree sum = Sum(a, b, carry);
    if (sum != null)
        AddTree(sum);
    carry = Carry(a, b, carry);
    arg.Purge();
}
// ...
```

**Program:** BinomialQueue class Merge method.

Each iteration of the main loop of the algorithm (lines 15-41) computes the $i^{th}$ "bit" of the result—the $i^{th}$ bit is a binomial tree of order $i$. At most three terms need to be considered: the carry from the preceding
Merging Binomial Queues

iteration and two $B$s, one from each of the queues that are being merged.

Two methods, Sum and Carry, compute the result required in each iteration. Program defines both Sum and Carry. Notice that the Sum method simply selects and returns one of its arguments. Therefore, the running time for Sum is clearly $O(1)$.

```java
public class BinomialQueue :
    AbstractContainer, MergeablePriorityQueue
{
    protected LinkedList treeList;

    protected virtual BinomialTree Sum(
        BinomialTree a, BinomialTree b, BinomialTree c)
    {
        if (a != null && b == null && c == null)
            return a;
        else if (a == null && b != null && c == null)
            return b;
        else if (a == null && b == null && c != null)
            return c;
        else if (a != null && b != null && c != null)
            return c;
        else
            return null;
    }

    protected BinomialTree Carry(
        BinomialTree a, BinomialTree b, BinomialTree c)
    {
        if (a != null && b != null && c == null)
            { return a.Add(b); }
        else if (a != null && b == null && c != null)
            { return a.Add(c); }
        else if (a == null && b != null && c != null)
            { return b.Add(c); }
        else if (a != null && b != null && c != null)
            { return a.Add(b); }
        else
            return null;
    }
```
Merging Binomial Queues

```java
return null;

// ...
```

**Program:** BinomialQueue class Sum and Carry methods.

In the worst case, the Carry method calls the Add method to combine two BinomialTrees into one. Therefore, the worst-case running time for Carry is

\[ T_{\text{Add}} + O(1). \]

Suppose the Merge method of Program is used to combine a binomial queue with \( n \) items with another that contains \( m \) items. Since the resulting priority queue contains \( n+m \) items, there are at most \( \log_2 (n + m + 1) \) binomial trees in the result. Thus, the worst-case running time for the Merge operation is

\[ [\log_2 (n + m + 1)] T_{\text{Add}} + O(\log(n + m)). \]
Putting Items into a Binomial Queue

With the Merge method at our disposal, the Enqueue operation is easy to implement. To enqueue an item in a given binomial queue, we create another binomial queue that contains just the one item to be enqueued and merge that queue with the original one.

Program shows how easily this can be done. Creating the empty queue (line 9) takes a constant amount of time. Creating the binomial tree $B_0$ with the one object at its root (line 10) can also be done in constant time. Finally, the time required to merge the two queues is

$$[\log_2(n + 2)]T_{\text{merge}} + O(\log n),$$

where $n$ is the number of items originally in the queue.

```csharp
public class BinomialQueue :
    AbstractContainer, MergeablePriorityQueue
{
    protected LinkedList treeList;

    public virtual void Enqueue(ComparableObject obj)
    {
        BinomialQueue queue = new BinomialQueue();
        queue.AddTree(new BinomialTree(obj));
        Merge(queue);
    }
    // ...
}
```

**Program:** BinomialQueue class Enqueue method.
Removing an Item from a Binomial Queue

A binomial queue is a forest of heap-ordered binomial trees. Therefore, to dequeue the smallest item from the queue, we must withdraw the root of one of the binomial trees. But what do we do with the rest of the tree once its root has been removed?

The solution lies in realizing that the collection of subtrees of the root of a binomial tree is a forest! For example, consider the binomial tree of order $k$,

$$B_k = \{R, B_0, B_1, B_2, \ldots, B_{k-1}\}$$

Taken all together, its subtrees form the binomial queue $F_{2^k-1}$:

$$F_{2^k-1} = \{B_0, B_1, B_2, \ldots, B_{k-1}\}.$$ 

Therefore, to delete the smallest item from a binomial queue, we first identify the binomial tree with the smallest root and remove that tree from the queue. Then, we consider all the subtrees of the root of that tree as a binomial queue and merge that queue back into the original one. Program □ shows how this can be coded.
Removing an Item from a Binomial Queue

Program: BinomialQueue class DequeueMin method.

The DequeueMin method begins by using the MinTree accessor to find the tree with the smallest root and then removing that tree using RemoveTree (lines 11-12). The time required to find the appropriate tree and to remove it is

$$(\lceil \log_2(n+1) \rceil - 1)T_{i\rightarrow r} + O(\log n),$$

where $n$ is the number of items in the queue.

A new binomial queue is created on line 14. All the children of the root of the minimum tree are
Removing an Item from a Binomial Queue

detached from the tree and added to the new binomial queue (lines 15-21). In the worst case, the minimum tree is the one with the highest order. i.e., $B_{\lfloor \log_2 n \rfloor}$ and the root of that tree has $\lfloor \log_2 n \rfloor$ children. Therefore, the running time of the loop on lines 15-21 is $O(\log n)$.

The new queue is then merged with the original one (line 22). Since the resulting queue contains $n-1$ keys, the running time for the Merge operation in this case is

$$[\log_2 n] T_{\text{merge}} + O(\log n).$$
Applications

- Discrete Event Simulation
- Implementation

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Discrete Event Simulation

One of the most important applications of priority queues is in *discrete event simulation*. Simulation is a tool which is used to study the behavior of complex systems. The first step in simulation is *modeling*. We construct a mathematical model of the system we wish to study. Then we write a computer program to evaluate the model. In a sense the behavior of the computer program mimics the system we are studying.

The systems studied using *discrete event simulation* have the following characteristics: The system has a *state* which evolves or changes with time. Changes in state occur at distinct points in simulation time. A state change moves the system from one state to another instantaneously. State changes are called *events*.

For example, suppose we wish to study the service received by customers in a bank. Suppose a single teller is serving customers. If the teller is not busy when a customer arrives at the bank, the that customer is immediately served. On the other hand, if the teller is busy when another customer arrives, that customer joins a queue and waits to be served.

We can model this system as a discrete event process as shown in Figure. The state of the system is characterized by the state of the server (the teller), which is either busy or idle, and by the number of customers in the queue. The events which cause state changes are the arrival of a customer and the departure of a customer.

Figure: A simple queueing system.

If the server is idle when a customer arrives, the server immediately begins to serve the customer and therefore changes its state to busy. If the server is busy when a customer arrives, that customer joins the queue.

When the server finishes serving the customer, that customer departs. If the queue is not empty, the server immediately commences serving the next customer. Otherwise, the server becomes idle.
How do we keep track of which event to simulate next? Each event (arrival or departure) occurs at a discrete point in *simulation time*. In order to ensure that the simulation program is correct, it must compute the events in order. This is called the *causality constraint*—events cannot change the past.

In our model, when the server begins to serve a customer we can compute the departure time of that customer. So, when a customer arrives at the server we *schedule* an event in the future which corresponds to the departure of that customer. In order to ensure that events are processed in order, we keep them in a priority queue in which the time of the event is its priority. Since we always process the pending event with the smallest time next and since an event can schedule new events only in the future, the causality constraint will not be violated.
Implementation

This section presents the simulation of a system comprised of a single queue and server as shown in Figure □. Program □ defines the class Event which represents events in the simulation. There are two parts to an event, a type (either ARRIVAL or DEPARTURE), and a time.

```csharp
public class Simulation
{
    private enum EventType
    {
        ARRIVAL,
        DEPARTURE
    }

    private class Event : Association
    {
        internal Event(EventType type, double time) :
            base(time, (int)type)
        {
        }

        internal double Time
        {
            get { return (double)Key; }
        }

        internal EventType Type
        {
            get { return (EventType)((int)Value); }
        }

        // ...
    }
}
```

Program: Event class.

Since events will be put into a priority queue, the Event class is derived from the Association class introduced in Section □. An association is an ordered pair comprised of a key and a value. In the case of the Event class, the key is the time of the event and the value is the type of the event. Therefore, the
Implementation

Events in a priority queue are prioritized by their times.

Program [ ] defines the Run method which implements the discrete event simulation. This method takes one argument, timeLimit, which specifies the total amount of time to be simulated.

The Simulation class contains a single field, called eventList, which is a priority queue. This priority queue is used to hold the events during the course of the simulation.

```java
public class Simulation {
    private PriorityQueue eventList = new LeftistHeap();

    public void Run(double timeLimit) {
        boolean serverBusy = false;
        int numberInQueue = 0;
        RandomVariable serviceTime = new ExponentialRV(100.0);
        RandomVariable interArrivalTime =
            new ExponentialRV(100.0);
        eventList.Enqueue(new Event(EventType.ARRIVAL, 0));
        while (!eventList.IsEmpty)
            { Event evt = (Event)eventList.DequeueMin();
              double t = evt.Time;
              if (t > timeLimit)
                { eventList.Purge(); break; }
              switch (evt.Type)
                {
                case EventType.ARRIVAL:
                  if (!serverBusy)
                    { serverBusy = true;
                      eventList.Enqueue(
                        new Event(EventType.DEPARTURE,
                                  t + serviceTime.Next));
                    }
                  else
                    ++numberInQueue;
                  eventList.Enqueue(
                      new Event(EventType.ARRIVAL,
                                  t + interArrivalTime.Next));
                  break;
                ....
            ```
Implementation

```java
    t + interArrivalTime.Next));
    break;

    case EventType.DEPARTURE:
        if (numberInQueue == 0)
            serverBusy = false;
        else
            --numberInQueue;
            eventList.Enqueue(
                new Event(EventType.DEPARTURE,
                t + serviceTime.Next));
        break;
    }
```

Program: Application of priority queues--discrete event simulation.

The state of the system being simulated is represented by the two variables `serverBusy` and `numberInQueue`. The first is a `bool` value which indicates whether the server is busy. The second keeps track of the number of customers in the queue.

In addition to the state variables, there are two instances of the class `ExponentialRV`. The class `ExponentialRV` is a random number generator defined in Section □. It implements the `RandomVariable` interface defined in Program □. This interface defines a property called `Next` which is used to sample the random number generator. Every time `Next` property `get` accessor is called, a different (random) result is returned. The random values are exponentially distributed around a mean value which is specified in the constructor. For example, in this case both `serviceTime` and `interArrivalTime` produce random distributions with the mean value of 100 (lines 9-11).

It is assumed that the `eventList` priority queue is initially empty. The simulation begins by enqueuing a customer arrival at time zero (line 12). The `while` loop (lines 13-44) constitutes the main simulation loop. This loop continues as long as the `eventList` is not empty, i.e., as long as there is an event to be simulated.

Each iteration of the simulation loop begins by dequeuing the next event in the event list (line 14). If the time of that event exceeds `timeLimit`, the event is discarded, the `eventList` is purged, and the simulation is terminated. Otherwise, the simulation proceeds.
The simulation of an event depends on the type of that event. The `switch` statement (line 18) invokes the appropriate code for the given event. If the event is a customer arrival and the server is not busy, `serverBusy` is set to `true` and the `serviceTime` random number generator is sampled to determine the amount of time required to service the customer. A customer departure is scheduled at the appropriate time in the future (lines 23-25). On the other hand, if the server is already busy when the customer arrives, we add one to the `numberInQueue` variable (line 28).

Another customer arrival is scheduled after every customer arrival. The `interArrivalTime` random number generator is sampled, and the arrival is scheduled at the appropriate time in the future (lines 29-31).

If the event is a customer departure and the queue is empty, the server becomes idle (lines 34-35). When a customer departs and there are still customers in the queue, the next customer in the queue is served. Therefore, `numberInQueue` is decreased by one and the `serviceTime` random number generator is sampled to determine the amount of time required to service the next customer. A customer departure is scheduled at the appropriate time in the future (lines 37-40).

Clearly the execution of the `Simulation` method given in Program 1 mimics the modeled system. Of course, the program given produces no output. For it to be of any practical value, the simulation program should be instrumented to allow the user to study its behavior. For example, the user may be interested in knowing statistics such as the average queue length and the average waiting time that a customer waits for service. And such instrumentation can be easily incorporated into the given framework.
Exercises

1. For each of the following key sequences determine the binary heap obtained when the keys are inserted one-by-one in the order given into an initially empty heap:
   1. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
   2. 3, 1, 4, 1, 5, 9, 2, 6, 5, 4.
   3. 2, 7, 1, 8, 2, 8, 1, 8, 2, 8.

2. For each of the binary heaps obtained in Exercise 1 determine the heap obtained after three consecutive DequeueMin operations.

3. Repeat Exercises 1 and 2 for a leftist heap.

4. Show the result obtained by inserting the keys \(1, 2, 3, \ldots, 2^h\) one-by-one in the order given into an initially empty binomial queue.

5. A full binary tree is a tree in which each node is either a leaf or its is a full node (see Exercise 6). Consider a complete binary tree with \(n\) nodes.
   1. For what values of \(n\) is a complete binary tree a full binary tree.
   2. For what values of \(n\) is a complete binary a perfect binary tree.

6. Prove by induction Theorem 1.

7. Devise an algorithm to determine whether a given binary tree is a heap. What is the running time of your algorithm?

8. Devise an algorithm to find the largest item in a binary min heap. **Hint:** First, show that the largest item must be in one of the leaves. What is the running time of your algorithm?

9. Suppose we are given an arbitrary array of \(n\) keys to be inserted into a binary heap all at once.
   Devise an \(O(n)\) algorithm to do this. **Hint:** See Section 1.

10. Devise an algorithm to determine whether a given binary tree is a leftist tree. What is the running time of your algorithm?

11. Prove that a complete binary tree is a leftist tree.

12. Suppose we are given an arbitrary array of \(n\) keys to be inserted into a leftist heap all at once.
    Devise an \(O(n)\) algorithm to do this. **Hint:** See Exercises 1 and 2.

13. Consider a complete binary tree with its nodes numbered as shown in Figure 1. Let \(K\) be the number of a node in the tree. The the binary representation of \(K\) is

\[
K = \sum_{i=0}^{k} b_i 2^i
\]
where $k = \lceil \log_2 K \rceil$.

1. Show that path from the root to a given node $K$ passes through the following nodes:

   $b_k$
   $b_k b_{k-1}$
   $b_k b_{k-1} b_{k-2}$
   $\vdots$
   $b_k b_{k-2} b_{k-3} \ldots b_2 b_1$
   $b_k b_{k-1} b_{k-2} \ldots b_2 b_1 b_0$.

2. Consider a complete binary tree with $n$ nodes. The nodes on the path from the root to the $n_{th}$ are special. Show that every non-special node is the root of a perfect tree.

14. The Enqueue algorithm for the BinaryHeap class does $O(\log n)$ object comparisons in the worst case. In effect, this algorithm does a linear search from a leaf to the root to find the point at which to insert a new key. Devise an algorithm that a binary search instead. Show that the number of comparisons required becomes $O(\log \log n)$. **Hint:** See Exercise □.

15. Prove Theorem □.

Projects

1. Design and implement a sorting algorithm using one of the priority queue implementations described in this chapter.

2. Complete the BinaryHeap class introduced in Program by providing suitable definitions for the following operations: CompareTo, IsFull, Accept, and GetEnumerator. Write a test program and test your implementation.

3. Complete the LeftistHeap class introduced in Program by providing suitable definitions for the following operations: LeftistHeap (constructor), Left, Right, SwapContentsWith, and SwapSubtrees. You will require a complete implementation of the base class BinaryTree. (See Project). Write a test program and test your implementation.

4. Complete the implementation of the BinomialTree class introduced in Program by providing suitable definitions for the following operations: BinomialTree (constructor), Count, and SwapContentsWith. You must also have a complete implementation of the base class GeneralTree. (See Project). Write a test program and test your implementation.

5. Complete the implementation of the BinomialQueue class introduced in Program by providing suitable definitions for the following methods: BinomialQueue (constructor), Purge, CompareTo, Accept, and GetEnumerator. You must also have a complete implementation of the BinomialTree class. (See Project). Write a test program and test your implementation.

6. The binary heap described in this chapter uses an array as the underlying foundational data structure. Alternatively we may base an implementation on the BinaryTree class described in Chapter. Implement a priority queue class that extends the BinaryTree class (Program) and implements the PriorityQueue interface (Program).

7. Implement a priority queue class using the binary search tree class from Chapter. Specifically, extend the BinarySearchTree class (Program) and implement the PriorityQueue interface (Program). You will require a complete implementation of the base class BinarySearchTree. (See Project). Write a test program and test your implementation.

8. Devise and implement an algorithm to multiply two polynomials:
\[
\left( \sum_{i=0}^{n} a_i x^i \right) \times \left( \sum_{j=0}^{m} b_j x^j \right).
\]

Generate the terms of the result in order by putting intermediate product terms into a priority queue. That is, use the priority queue to group terms with the same exponent. **Hint:** See also Project □.
Sets, Multisets, and Partitions

In mathematics a set is a collection of elements, especially a collection having some feature or features in common. The set may have a finite number of elements, e.g., the set of prime numbers less than 100; or it may have an infinite number of elements, e.g., the set of right triangles. The elements of a set may be anything at all--from simple integers to arbitrarily complex objects. However, all the elements of a set are distinct--a set may contain only one instance of a given element.

For example, \( \{ \} \), \( \{ a \} \), \( \{ a, b, c, d \} \), and \( \{ d, e \} \) are all sets the elements of which are drawn from \( U = \{ a, b, c, d, e \} \). The set of all possible elements, \( U \), is called the universal set. Note also that the elements comprising a given set are not ordered. Thus, \( \{ a, b, c \} \) and \( \{ b, c, a \} \) are the same set.

There are many possible operations on sets. In this chapter we consider the most common operations for combining sets--union, intersection, difference:

**union**

The union (or conjunction) of sets \( S \) and \( T \), written \( S \cup T \), is the set comprised of all the elements of \( S \) together with all the elements of \( T \). Since a set cannot contain duplicates, if the same item is an element of both \( S \) and \( T \), only one instance of that item appears in \( S \cup T \). If \( S = \{ a, b, c, d \} \) and \( T = \{ d, e \} \), then \( S \cup T = \{ a, b, c, d, e \} \).

**intersection**

The intersection (or disjunction) of sets \( S \) and \( T \) is written \( S \cap T \). The elements of \( S \cap T \) are those items which are elements of both \( S \) and \( T \). If \( S = \{ a, b, c, d \} \) and \( T = \{ d, e \} \), then \( S \cap T = \{ d \} \).

**difference**

The difference (or subtraction) of sets \( S \) and \( T \), written \( S - T \), contains those elements of \( S \) which are not also elements of \( T \). That is, the result \( S - T \) is obtained by taking the set \( S \) and removing from it those elements which are also found in \( T \). If \( S = \{ a, b, c, d \} \) and \( T = \{ d, e \} \), then \( S - T = \{ a, b, c \} \).
Figure 1 illustrates the basic set operations using a *Venn diagram*. A Venn diagram represents the membership of sets by regions of the plane. In Figure 1, the two sets $S$ and $T$ divide the plane into the four regions labeled $I$-$IV$. The following table illustrates the basic set operations by enumerating the regions that comprise each set.

<table>
<thead>
<tr>
<th>set</th>
<th>region(s) of Figure 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U$</td>
<td>I, II, III, IV</td>
</tr>
<tr>
<td>$S$</td>
<td>I, II</td>
</tr>
<tr>
<td>$S'$</td>
<td>III, IV</td>
</tr>
<tr>
<td>$T$</td>
<td>II, III</td>
</tr>
<tr>
<td>$S \cup T$</td>
<td>I, II, III</td>
</tr>
<tr>
<td>$S \cap T$</td>
<td>II</td>
</tr>
<tr>
<td>$S-T$</td>
<td>I</td>
</tr>
<tr>
<td>$T-S$</td>
<td>III</td>
</tr>
</tbody>
</table>

**Figure:** Venn diagram illustrating the basic set operations.
Sets, Multisets, and Partitions

- Exercises
- Projects

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Basics

In this chapter we consider sets the elements of which are integers. By using integers as the universe rather than arbitrary objects, certain optimizations are possible. For example, we can use a bit-vector of length $N$ to represent a set whose universe is $\{0, 1, \ldots, N - 1\}$. Of course, using integers as the universe does not preclude the use of more complex objects, provided there is a one-to-one mapping between those objects and the elements of the universal set.

A crucial requirement of any set representation scheme is that it supports the common set operations including $\text{union}$, $\text{intersection}$, and $\text{set difference}$. We also need to compare sets and, specifically, to determine whether a given set is a subset of another.

Implementing Sets
Implementing Sets

As discussed above, this chapter addresses the implementation of sets of integers. A set is a collection of elements. Naturally, we want to insert and withdraw objects from the collection and to test whether a given object is a member of the collection. Therefore, we consider sets as being derived from the SearchableContainer class defined in Chapter [1]. (See Figure [2]). In general, a searchable container can hold arbitrary objects. However, in this chapter we will assume that the elements of a set are integers.

Figure: Object class hierarchy

Program [3] defines the Set interface. The Set interface extends the the SearchableContainer interface defined in Program [4]. Five new methods are declared--Union, Intersection, Difference, Equals, and IsSubset. These methods correspond to the various set operations discussed above.
public interface Set : SearchableContainer
{
    Set Union(Set set);
    Set Intersection(Set set);
    Set Difference(Set set);
    bool Equals(Set set);
    bool IsSubset(Set set);
}

**Program:** Set interface.

Program defines the AbstractSet class. The AbstractSet class extends the AbstractSearchableContainer class introduced in Program and it implements the Set interface defined in Program. As shown in Figure, all of the concrete set classes discussed in this chapter are derived from the AbstractSet class.
Implementing Sets

Program: AbstractSet class.

The AbstractSet class defines a field called universeSize. This field is used to record the size of the universal set. The constructor for the AbstractSet class is given in Program. It takes a single argument, \( N = \text{universeSize} \), which specifies that the universal set shall be \( \{0, 1, \ldots, N - 1\} \).

The items contained in a set are integers. Therefore, the AbstractSet class defines abstract methods called as Insert, IsMember, and Withdraw, that take int arguments.

However, the methods of the SearchableContainer interface such as Insert, IsMember, and Withdraw, expect their arguments to be ComparableObjects. For this reason, the AbstractSet class provides a default implementation for each such method which converts its argument to an int and then invokes the like-named abstract method that takes an int argument.
Array and Bit-Vector Sets

In this section we consider finite sets over a finite universe. Specifically, the universe we consider is \( \{0, 1, \ldots, N - 1\} \), the set of integers in the range from zero to \( N-1 \), for some fixed and relatively small value of \( N \).

Let \( U = \{0, 1, \ldots, N - 1\} \) be the universe. Every set which we wish to represent is a subset of \( U \).

The set of all subsets of \( U \) is called the power set of \( U \) and is written \( 2^U \). Thus, the sets which we wish to represent are the elements of \( 2^U \). The number of elements in the set \( U \), written \( |U| \), is \( N \). Similarly, \( |2^U| = 2^{|U|} = 2^N \). This observation should be obvious: For each element of the universal set \( U \) there are only two possibilities: Either it is, or it is not, a member of the given set.

This suggests a relatively straightforward representation of the elements of \( 2^U \)--an array of \( \text{bool} \) values, one for each element of the universal set. By using array subscripts in \( U \), we can represent the set implicitly. That is, \( i \) is a member of the set if the \( i^{th} \) array element is true.

Program introduces the class \texttt{SetAsArray}. The \texttt{SetAsArray} class extends the \texttt{AbstractSet} class defined in Program. This class uses an array of length \( N = \text{numberOfItems} \) to represent the elements of \( 2^U \) where \( U = \{0, 1, \ldots, N - 1\} \).

```csharp
public class SetAsArray : AbstractSet, Set
{
    protected bool[] array;

    // ...
}
```

Program: \texttt{SetAsArray} fields.
Array and Bit-Vector Sets

- **Basic Operations**
- **Union, Intersection, and Difference**
- **Comparing Sets**
- **Bit-Vector Sets**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Basic Operations

Program defines the constructor for the SetAsArray class as well as the three basic operations--Insert, IsMember, and Withdraw. The constructor takes a single argument \( N = \text{numberOfItems} \), which defines the universe and, consequently, the size of the array of \( \text{bool} \) values. The constructor creates the empty set by initializing all the elements of the \( \text{bool} \) array to false. Clearly, the running time of the constructor is \( O(N) \).

```csharp
public class SetAsArray : AbstractSet, Set
{
    protected bool[] array;

    public SetAsArray(int n) : base(n)
    {
        array = new bool[universeSize];
        for (int item = 0; item < universeSize; ++item)
            array[item] = false;
    }

    public override void Insert(int item)
    { array[item] = true; }

    public override bool IsMember(int item)
    { return array[item]; }

    public override void Withdraw(int item)
    { array[item] = false; }

    // ...
}
```

Program: SetAsArray class constructor, Insert, Withdraw, and IsMember methods.

The Insert method is used to put an item into the set. The method takes an int argument that specifies the item to be inserted. Then the corresponding element of array is set to true to indicate that the item has been added to the set. The running time of the Insert operation is \( O(1) \).
The **IsMember** method is used to test whether a given item is an element of the set. The semantics are somewhat subtle. Since a set does not actually keep track of the specific object instances that are inserted, the membership test is based on the *value* of the argument. The method simply returns the value of the appropriate element of the *array*. The running time of the **IsMember** operation is $O(1)$.

The **Withdraw** method is used to take an item out of a set. The withdrawal operation is the opposite of insertion. Instead of setting the appropriate array element to *true*, it is set to *false*. The running time of the **Withdraw** is identical to that of **Insert**, viz., is $O(1)$. 

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Program defines the three methods, Union, Intersection, and Difference. These methods correspond to $\cup$, $\cap$, and $\setminus$, respectively.

```csharp
public class SetAsArray : AbstractSet, Set
{
    protected bool[] array;

    public virtual Set Union(Set set)
    {
        SetAsArray arg = (SetAsArray)set;
        if (universeSize != arg.universeSize)
            throw new ArgumentException("mismatched sets");
        SetAsArray result = new SetAsArray(universeSize);
        for (int i = 0; i < universeSize; ++i)
            result.array[i] = array[i] | arg.array[i];
        return result;
    }

    public virtual Set Intersection(Set set)
    {
        SetAsArray arg = (SetAsArray)set;
        if (universeSize != arg.universeSize)
            throw new ArgumentException("mismatched sets");
        SetAsArray result = new SetAsArray(universeSize);
        for (int i = 0; i < universeSize; ++i)
            result.array[i] = array[i] & arg.array[i];
        return result;
    }

    public virtual Set Difference(Set set)
    {
        SetAsArray arg = (SetAsArray)set;
        if (universeSize != arg.universeSize)
            throw new ArgumentException("mismatched sets");
        SetAsArray result = new SetAsArray(universeSize);
        for (int i = 0; i < universeSize; ++i)
            result.array[i] = array[i] - arg.array[i];
        return result;
    }
}
```
Program: SetAsArray class Union, Intersection and Difference methods.

The set union operator takes one argument which is assumed to be a SetAsArray instance. It computes the SetAsArray obtained from the union of this and set. The implementation given requires that the sets be compatible. Two sets are deemed to be compatible if they have the same universe. The result also has the same universe. Consequently, the bool array in all three sets has the same length, N. The set union method creates a result array of the required size and then computes the elements of the array as required. The i\textsuperscript{th} element of the result is true if either the i\textsuperscript{th} element of s or the i\textsuperscript{th} element of t is true. Thus, set union is implemented using the bool or operator, ||.

The set intersection method is almost identical to set union, except that the elements of the result are computed using the bool and operator. The set difference method is also very similar. In this case, an item is an element of the result only if it is a member of s and not a member of t.

Because all three methods are almost identical, their running times are essentially the same. That is, the running time of the set union, intersection, and difference operations is O(N), where \(N = \text{NumberOfItems}\).
Comparing Sets

There is a special family of operators for comparing sets. Consider two sets, say $S$ and $T$. We say that $S$ is a subset of $T$, written $S \subseteq T$, if every element of $S$ is also an element of $T$. If there is at least one element of $T$ that is not also an element of $S$, we say that $S$ is a proper subset of $T$, written $S \subset T$. We can also reverse the order in which the expressions are written to get $T \supset S$ or $T \supseteq S$, which indicates that $T$ is a (proper) superset of $S$.

The set comparison operators follow the rule that if $S \subseteq T$ and $T \subseteq S$ then $S = T$, which is analogous to a similar property of numbers: $x \leq y \land y \leq x \iff x = y$. However, set comparison is unlike numeric comparison in that there exist sets $S$ and $T$ for which neither $S \subseteq T$ nor $T \subseteq S$! For example, clearly this is the case for $S = \{1, 2\}$ and $T = \{2, 3\}$. Mathematically, the relation $\subseteq$ is called a partial order because there exist some pairs of sets for which neither $S \subseteq T$ nor $T \subseteq S$ holds; whereas the relation $\leq$ (among integers, say) is a total order.

Program defines the methods Equals and IsSubset each of which take argument that is assumed to be a SetAsArray instance. The former tests for equality and the latter determines whether the relation $\subseteq$ holds between this and set. Both operators return a bool result. The worst-case running time of each of these operations is clearly $O(N)$. 
public class SetAsArray : AbstractSet, Set
{
    protected bool[] array;

    public virtual bool Equals(Set set)
    {
        SetAsArray arg = (SetAsArray)set;
        if (universeSize != arg.universeSize)
            throw new ArgumentException("mismatched sets");
        for (int item = 0; item < universeSize; ++item)
            if (array[item] != arg.array[item])
                return false;
        return true;
    }

    public virtual bool IsSubset(Set set)
    {
        SetAsArray arg = (SetAsArray)set;
        if (universeSize != arg.universeSize)
            throw new ArgumentException("mismatched sets");
        for (int item = 0; item < universeSize; ++item)
            if (array[item] && !arg.array[item])
                return false;
        return true;
    }
    // ...
}

Program: SetAsArray class Equals and IsSubset methods.

A complete repertoire of comparison methods would also include methods to compute ⊆, ⊇, =, and ≠. These operations follow directly from the implementation shown in Program (Exercise ).
Bit-Vector Sets

The common language runtime uses one byte of memory to store a single C# bool value. Furthermore, each bool in an array of bools occupies two bytes of memory. However, since there are only the two values true and false, a single bit is sufficient to hold a bool value. Therefore, we can realize a significant reduction in the memory space required to represent a set if we use an array of bits. Furthermore, by using bitwise operations to implement the basic set operations such as union and intersection, we can achieve a commensurate reduction in execution time. Unfortunately, these improvements are not free—the operations Insert, IsMember, and Withdraw, all slow down by a constant factor.

Since C# does not directly support arrays of bits, we will simulate an array of bits using an array of ints. Program 1 illustrates how this can be done. The constant BITS is defined as the number of bits in a single int.

```
public class SetAsBitVector : AbstractSet, Set
{
    protected int[] vector;

    protected const int BITS = 8*sizeof(int);
    // ...

Program: SetAsBitVector fields.
```

- **Basic Operations**
- **Union, Intersection, and Difference**
Basic Operations

Program 1 defines the constructor for the \texttt{SetAsBitVector} class as well as the three basic operations--\texttt{Insert}, \texttt{IsMember}, and \texttt{Withdraw}. The constructor takes a single argument

\[ N = \text{numbe}r\text{ofItems}, \]

which specifies the universe and, consequently, the number of bits needed in the bit array. The constructor creates an array of \texttt{ints} of length \( \lceil N / w \rceil \), where \( w = \text{BITS} \) is the number of bits in an \texttt{int}, and sets the elements of the array to zero. The running time of the constructor is \( O(\lceil N / w \rceil) = O(N) \).

```csharp
public class SetAsBitVector : AbstractSet, Set
{
    protected int[] vector;

    public SetAsBitVector(int n) : base(n)
    {
        vector = new int[(n + BITS - 1) / BITS];
        for (int i = 0; i < vector.Length; ++i)
            vector[i] = 0;
    }

    public override void Insert(int item)
    {
        vector[item / BITS] |= 1 << item % BITS;
    }

    public override void Withdraw(int item)
    {
        vector[item / BITS] &= ~(1 << item % BITS);
    }

    public override bool IsMember(int item)
    {
        return (vector[item / BITS] &
                (1 << item % BITS)) != 0;
    }
}```
Basic Operations

```
(1 << item % BITS)) != 0;

}  // ...  

```

**Program**: SetAsBitVector class constructor, Insert, Withdraw, and IsMember methods.

To insert an item into the set, we need to change the appropriate bit in the array of bits to one. The $i^{th}$ bit of the bit array is bit $i \mod \, w$ of word $i/w$. Thus, the Insert method is implemented using a bitwise or operation to change the $i^{th}$ bit to one as shown in Program 1. Even though it is slightly more complicated than the corresponding operation for the SetAsArray class, the running time for this operation is still $O(1)$. Since $w = \text{BITS}$ is a power of two, it is possible to replace the division and modulo operations, / and %, with shifts and masks like this:

```
vector[item >> shift] |= 1 << (item & mask);
```

for a suitable definition of the constants shift and mask. Depending on the compiler and machine architecture, doing so may improve the performance of the Insert operation by a constant factor. Of course, its asymptotic performance is still $O(1)$.

To withdraw an item from the set, we need to clear the appropriate bit in the array of bits and to test if an item is a member of the set, we test the corresponding bit. The IsMember and Withdraw methods in Program 1 show how this can be done. Like Insert, both these methods have constant worst-case running times.
Union, Intersection, and Difference

The implementations of the union, intersection, and difference methods for operands of type `SetAsBitVector` are shown in Program. The code is quite similar to that for the `SetAsArray` class given in Program.:

```csharp
public class SetAsBitVector : AbstractSet, ISet
{
    protected int[] vector;

    public virtual Set Union(Set set)
    {
        SetAsBitVector arg = (SetAsBitVector)set;
        if (universeSize != arg.universeSize)
            throw new ArgumentException("mismatched sets");
        SetAsBitVector result = new SetAsBitVector(universeSize);
        for (int i = 0; i < vector.Length; ++i)
            result.vector[i] = vector[i] | arg.vector[i];
        return result;
    }

    public virtual Set Intersection(Set set)
    {
        SetAsBitVector arg = (SetAsBitVector)set;
        if (universeSize != arg.universeSize)
            throw new ArgumentException("mismatched sets");
        SetAsBitVector result = new SetAsBitVector(universeSize);
        for (int i = 0; i < vector.Length; ++i)
            result.vector[i] = vector[i] & arg.vector[i];
        return result;
    }

    public virtual Set Difference(Set set)
    {
        SetAsBitVector arg = (SetAsBitVector)set;
        if (universeSize != arg.universeSize)
            throw new ArgumentException("mismatched sets");
        SetAsBitVector result = new SetAsBitVector(universeSize);
        for (int i = 0; i < vector.Length; ++i)
            result.vector[i] = vector[i] ^ arg.vector[i];
        return result;
    }
}
```
Program: SetAsBitVector class Union, Intersection and Difference methods.

Instead of using the bool operators &&, ||, and !, we have used the bitwise operators &, |, and . By using the bitwise operators, \( w = \text{BITS} \) bits of the result are computed in each iteration of the loop. Therefore, the number of iterations required is \( \lceil N/w \rceil \) instead of \( N \). The worst-case running time of each of these operations is \( O(\lceil N/w \rceil) = O(N) \).

Notice that the asymptotic performance of these SetAsBitVector class operations is the same as the asymptotic performance of the SetAsArray class operations. That is, both of them are \( O(N) \). Nevertheless, the SetAsBitVector class operations are faster. In fact, the bit-vector approach is asymptotically faster than the the array approach by the factor \( w \).
Multisets

A multiset is a set in which an item may appear more than once. That is, whereas duplicates are not permitted in a regular set, they are permitted in a multiset. Multisets are also known simply as bags.

Sets and multisets are in other respects quite similar: Both support operations to insert and withdraw items; both provide a means to test the membership of a given item; and both support the basic set operations of union, intersection, and difference. As a result, the Multiset interface is essentially the same as the Set interface as shown in Program.

```csharp
public interface Multiset : SearchableContainer
{
    Multiset Union(Multiset set);
    Multiset Intersection(Multiset set);
    Multiset Difference(Multiset set);
    bool Equals(Multiset set);
    bool IsSubset(Multiset set);
}
```

Program: Multiset interface.

- Array Implementation
- Linked-List Implementation
Array Implementation

A regular set may contain either zero or one instance of a particular item. As shown in the preceding section if the number of possible items is not excessive, we may use an array of `bool` variables to keep track of the number of instances of a particular item in a regular set. The natural extension of this idea for a multiset is to keep a separate count of the number of instances of each item in the multiset.

Program introduces the `MultisetAsArray` class. The `MultisetAsArray` class extends the `AbstractSet` class defined in Program and it implements the `Multiset` interface defined in Program. The multiset is implemented using an array of `N = numberOfItems` counters. Each counter is an `int` in this case.

```csharp
public class MultisetAsArray : AbstractSet, Multiset
{
    protected int[] array;

    // ...
}
```

**Program:** MultisetAsArray class.

- Basic Operations
- Union, Intersection, and Difference
Basic Operations

Program defines the constructor for the MultisetAsArray class as well as the three basic operations--Insert, IsMember, and Withdraw. The constructor takes a single argument, \( N = \text{numberOfItems} \), and initializes an array of length \( N \) counters all to zero. The running time of the constructor is \( O(N) \).

```csharp
public class MultisetAsArray : AbstractSet, Multiset
{
    protected int[] array;

    public MultisetAsArray(int n) : base(n)
    {
        array = new int[universeSize];
        for (int item = 0; item < universeSize; ++item)
            array[item] = 0;
    }

    public override void Insert(int item)
    { ++array[item]; }

    public override void Withdraw(int item)
    { if (array[item] > 0) --array[item]; }

    public override bool IsMember(int item)
    { return array[item] > 0; }
    // ...
}
```

Program: MultisetAsArray class constructor, Insert, Withdraw, and IsMember methods.
Basic Operations

To insert an item, we simply increase the appropriate counter; to delete an item, we decrease the counter; and to test whether an item is in the set, we test whether the corresponding counter is greater than zero. In all cases the operation can be done in constant time.
Union, Intersection, and Difference

Because multisets permit duplicates but sets do not, the definitions of union, intersection, and difference are slightly modified for multisets. The union of multisets $S$ and $T$, written $S \cup T$, is the multiset comprised of all the elements of $S$ together with all the element of $T$. Since a multiset may contain duplicates, it does not matter if the same element appears in $S$ and $T$.

The subtle difference between union of sets and union of multisets gives rise to an interesting and useful property. If $S$ and $T$ are regular sets,

$$\min(|S|, |T|) \leq |S \cup T| \leq |S| + |T|.$$ 

On the other hand, if $S$ and $T$ are multisets,

$$|S \cup T| = |S| + |T|.$$ 

The intersection of sets $S$ and $T$ is written $S \cap T$. The elements of $S \cap T$ are those items which are elements of both $S$ and $T$. If a given element appears more than once in $S$ or $T$ (or both), the intersection contains $m$ copies of that element, where $m$ is the smaller of the number of times the element appears in $S$ or $T$. For example, if $S = \{0, 1, 1, 2, 2\}$ and $T = \{1, 2, 2, 3\}$, the intersection is $S \cap T = \{1, 2, 2\}$.

The difference of sets $S$ and $T$, written $S - T$, contains those elements of $S$ which are not also elements of $T$. That is, the result $S - T$ is obtained by taking the set $S$ and removing from it those elements which are also found in $T$.

Program □ gives the implementations of the union, intersection, and difference methods of MultisetAsArray class. This code is quite similar to that of the SetAsArray class (Program □) and the SetAsBitVector class (Program □). The worst-case running time of each of these operations is $O(N)$.

```java
public class MultisetAsArray : AbstractSet, Multiset
{
    protected int[] array;

    public virtual Multiset Union(Multiset set)
```
public virtual Multiset Union(Multiset set) {
    MultisetAsArray arg = (MultisetAsArray)set;
    if (universeSize != arg.universeSize)
        throw new ArgumentException("mismatched sets");
    MultisetAsArray result =
        new MultisetAsArray(universeSize);
    for (int i = 0; i < universeSize; ++i)
        result.array[i] = array[i] + arg.array[i];
    return result;
}

public virtual Multiset Intersection(Multiset set) {
    MultisetAsArray arg = (MultisetAsArray)set;
    if (universeSize != arg.universeSize)
        throw new ArgumentException("mismatched sets");
    MultisetAsArray result =
        new MultisetAsArray(universeSize);
    for (int i = 0; i < universeSize; ++i)
        result.array[i] = Math.Min(
            array[i], arg.array[i]);
    return result;
}

public virtual Multiset Difference(Multiset set) {
    MultisetAsArray arg = (MultisetAsArray)set;
    if (universeSize != arg.universeSize)
        throw new ArgumentException("mismatched sets");
    MultisetAsArray result =
        new MultisetAsArray(universeSize);
    for (int i = 0; i < universeSize; ++i)
        if (arg.array[i] <= array[i])
            result.array[i] = array[i] - arg.array[i];
    return result;
}

// ...
Program: MultisetAsArray class Union, Intersection and Difference methods.

Instead of using the bool operators &&, ||, and !, we have used + (integer addition), Math.Min and – (integer subtraction). The following table summarizes the operators used in the various set and multiset implementations.

<table>
<thead>
<tr>
<th>operation</th>
<th>SetAsArray</th>
<th>SetAsBitVector</th>
<th>MultisetAsArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>intersection</td>
<td>&amp;&amp;</td>
<td>&amp;</td>
<td>Math.Min</td>
</tr>
<tr>
<td>difference</td>
<td>&amp;&amp; and !</td>
<td>&amp; and</td>
<td>&lt;= and –</td>
</tr>
</tbody>
</table>

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Linked-List Implementation

The array implementation of multisets is really only practical if the number of items in the universe, \( N = |U| \), is not too large. If \( N \) is large, then it is impractical, or at least extremely inefficient, to use an array of \( N \) counters to represent the multiset. This is especially so if the number of elements in the multisets is significantly less than \( N \).

If we use a linked list of elements to represent a multiset \( S \), the space required is proportional to the size of the multiset, \(|S|\). When the size of the multiset is significantly less than the size of the universe, \(|S| \ll |U|\), it is more efficient in terms of both time and space to use a linked list.

Program \( \square \) introduces the MultisetAsLinkedList class. The MultisetAsLinkedList extends the AbstractSet class defined in Program \( \square \) and it implements the Multiset interface defined in Program \( \square \). In this case a linked list of ints is used to record the contents of the multiset.

How should the elements of the multiset be stored in the list? Perhaps the simplest way is to store the elements in the list in no particular order. Doing so makes the Insert operation efficient—it can be done in constant time. Furthermore, the IsMember and Withdraw operations both take \( O(n) \) time, where \( n \) is the number of items in the multiset, regardless of the order of the items in the linked list.

Consider now the union, intersection, and difference of two multisets, say \( S \) and \( T \). If the linked list is unordered, the worst case running time for the union operation is \( O(m+n) \), where \( m=|S| \) and \( n=|T| \). Unfortunately, intersection, and difference are both \( O(mn) \).

If, on the other hand, we use an ordered linked list, union, intersection, and difference can all be done in \( O(m+n) \) time. The trade-off is that the insertion becomes an \( O(n) \) operation rather than a \( O(1) \). The MultisetAsLinkedList implementation presented in this section records the elements of the multiset in an ordered linked list.
public class MultisetAsLinkedList : AbstractSet, Multiset
{
    protected LinkedList list;

    // ...
}

Program: MultisetAsLinkedList fields.

- Union
- Intersection
Union

The union operation for MultisetAsLinkedList class requires the merging of two ordered, linked lists as shown in Program 1. We have assumed that the smallest element contained in a multiset is found at the head of the linked list and the largest is at the tail.

```csharp
public class MultisetAsLinkedList : AbstractSet, Multiset
{
    protected LinkedList list;

    public virtual Multiset Union(Multiset set)
    {
        MultisetAsLinkedList arg = (MultisetAsLinkedList)set;
        if (universeSize != arg.universeSize)
            throw new ArgumentException("mismatched sets");
        MultisetAsLinkedList result =
            new MultisetAsLinkedList(universeSize);
        LinkedList.Element p = list.Head;
        LinkedList.Element q = arg.list.Head;
        while (p != null && q != null)
        {
            if (((int)p.Datum <= (int)q.Datum)
            {
                result.list.Append(p.Datum);
                p = p.Next;
            }
            else
            {
                result.list.Append(q.Datum);
                q = q.Next;
            }
        }
        for ( ; p != null; p = p.Next)
            result.list.Append(p.Datum);
        for ( ; q != null; q = q.Next)
```
Program: MultisetAsLinkedList class Union method.

The Union method computes its result as follows: The main loop of the program (lines 14-26) traverses the linked lists of the two operands, in each iteration appending the smallest remaining element to the result. Once one of the lists has been exhausted, the remaining elements in the other list are simply appended to the result (lines 27-30). The total running time for the Union method is $O(m+n)$, where $m = |\text{this}|$ and $n = |\text{set}|$. 
Intersection

The implementation of the intersection operator for the MultisetAsLinkedList class is similar to that of union. However, instead of merging of two ordered, linked lists to construct a third, we compare the elements of two lists and append an item to the third only when it appears in both of the input lists.

The Intersection method is shown in Program □.

```csharp
public class MultisetAsLinkedList : AbstractSet, Multiset
{
    protected LinkedList list;

    public virtual Multiset Intersection(Multiset set)
    {
        MultisetAsLinkedList arg = (MultisetAsLinkedList)set;
        if (universeSize != arg.universeSize)
            throw new ArgumentException("mismatched sets");
        MultisetAsLinkedList result = new MultisetAsLinkedList(universeSize);
        LinkedList.Element p = list.Head;
        LinkedList.Element q = arg.list.Head;
        while (p != null && q != null)
        {
            int diff = (int)p.Datum - (int)q.Datum;
            if (diff == 0)
                result.list.Append(p.Datum);
            if (diff <= 0)
                p = p.Next;
            if (diff >= 0)
                q = q.Next;
        }
        return result;
    }
    // ...
}
```
**Program:** MultisetAsLinkedList class Intersection method.

The main loop of the program traverses the linked lists of both input operands at once using two variables (lines 14-23). If the next element in each list is the same, that element is appended to the result and both variables are advanced. Otherwise, only one of the variables is advanced--the one pointing to the smaller element.

The number of iterations of the main loop actually done depends on the contents of the respective linked lists. The best case occurs when both lists are identical. In this case, the number of iterations is \( m \), where \( m = |\text{this}| = |\text{set}| \). In the worst case case, the number of iterations done is \( m+n \). Therefore, the running time of the Intersection method is \( O(m+n) \).
Partitions

Consider the finite universal set \( U = \{ 0, 1, \ldots, N - 1 \} \). A partition of \( U \) is a finite set of sets \( P = \{ S_1, S_2, \ldots, S_p \} \) with the following properties:

1. The sets \( S_1, S_2, \ldots, S_p \) are pairwise disjoint. That is, \( S_i \cap S_j = \emptyset \) for all values of \( i \) and \( j \) such that \( 1 \leq i < j \leq p \).
2. The sets \( S_1, S_2, \ldots, S_p \) span the universe \( U \). That is,

\[
\bigcup_{i=1}^{p} S_i = S_1 \cup S_2 \cup \cdots \cup S_p = U.
\]

For example, consider the universe \( U = \{1, 2, 3\} \). There are exactly five partitions of \( U \):

\[
\begin{align*}
P_0 &= \{ \{1\}, \{2\}, \{3\} \}, \\
P_1 &= \{ \{1\}, \{2, 3\} \}, \\
P_2 &= \{ \{2\}, \{1, 3\} \}, \\
P_3 &= \{ \{3\}, \{1, 2\} \}, \text{ and} \\
P_4 &= \{ \{1, 2, 3\} \}.
\end{align*}
\]

In general, given a universe \( U \) of size \( n > 0 \), i.e., \( |U| = n \), there are \( \sum_{m=0}^{n} \binom{n}{m} \) partitions of \( U \), where \( \binom{n}{m} \) is the so-called Stirling number of the second kind which denotes the number of ways to partition a set of \( n \) elements into \( m \) nonempty disjoint subsets.

Applications which use partitions typically start with an initial partition and refine that partition either by joining or by splitting elements of the partition according to some application-specific criterion. The
Partitions

result of such a computation is the partition obtained when no more elements can be split or joined.

In this chapter we shall consider only applications that begin with the initial partition of \(U\) in which each item in \(U\) is in a separate element of the partition. Thus, the initial partition consists of \(|U|\) sets, each of size one (like \(P_0\) above). Furthermore, we restrict the applications in that we only allow elements of a partition to be joined—we do not allow elements to split.

The two operations to be performed on partitions are:

**Find**

Given an item in the universe, say \(i \in U\), find the element of the partition that contains \(i\). That is, find \(S_j \in P\) such that \(i \in S_j\).

**Join**

Given two distinct elements of a partition \(P\), say \(S_i \in P\) and \(S_j \in P\) such that \(i \neq j\), create a new partition \(P'\) by removing the two elements \(S_i\) and \(S_j\) from \(P\) and replacing them with a single element \(S_i \cup S_j\).

For example, consider the partition \(P = \{S_1, S_2, S_3\} = \{\{1\}, \{2, 3\}, \{4\}\}\). The result of the operation \(\text{find}(3)\) is the set \(S_2 = \{2, 3\}\) because 3 is a member of \(S_2\). Furthermore, when we \(\text{join}\) sets \(S_1\) and \(S_3\), we get the partition \(P' = \{\{1, 4\}, \{2, 3\}\}\).
Representing Partitions

Program defines the Partition interface. The Partition interface extends the Set interface defined in Program. Since a partition is a set of sets, it makes sense to derive Partition from Set. The two methods, Find and Join, correspond to the partition operations described above.

```
public interface Partition : Set
{
    Set Find(int item);
    void Join(Set set1, Set set2);
}
```

Program: Partition interface.

The elements of a partition are also sets. Consequently, the objects contained in a Partition are also implement the Set interface. The Find method of the Partition class expects as its argument an int and returns the Set which contains the specified item.

The Join method takes two arguments, both of them references to Sets. The two arguments are expected to be distinct elements of the partition. The effect of the Join operation is to remove the specified sets from the partition and replace them with a Set which represents the union of the two.
Implementing a Partition using a Forest

A partition is a set of sets. Consequently, there are two related issues to consider when developing an approach for representing partitions:

1. How are the individual elements or parts of the partition represented?
2. How are the elements of a partition combined into the whole?

This section presents an approach in which each element of a partition is a tree. Therefore, the whole partition is a forest.

For example, Figure shows how the partition

\[ P = \{S_1, S_2, S_3, S_4\} \]
\[ = \{\{0, 4\}, \{2, 6, 8\}, \{10\}, \{1, 3, 5, 7, 9, 11\}\} \]

can be represented using a forest. Notice that each element of the universal set \( U = \{0, 1, \ldots, 11\} \) appears in exactly one node of exactly one tree.

Figure: Representing a partition as a forest.
Implementing a Partition using a Forest

The trees in Figure □ have some very interesting characteristics. The first characteristic concerns the shapes of the trees: The nodes of the trees have arbitrary degrees. The second characteristic concerns the positions of the keys: there are no constraints on the positions of the keys in a tree. The final characteristic has to do with the way the tree is represented: Instead of pointing to its children, each node of the tree points to its parent!

Since there is no particular order to the nodes in the trees, it is necessary to keep track of the position of each node explicitly. Figure □ shows how this can be done using an array. (This figure shows the same partition as in Figure □). The array contains a node for each element of the universal set \( U \). Specifically, the \( i \)th array element holds the node that contains item \( i \). Having found the desired node, we can follow the chain of parent pointers to find the root of the corresponding tree.

![Diagram of a partition using a forest](http://www.brpreiss.com/books/opus6/html/page407.html)

**Figure**: Finding the elements of a partition.

- Implementation
- Constructor
- Find and Join Methods

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Implementation

Program declares two classes--PartitionAsForest and the inner class PartitionTree. The latter is used to represent the individual elements or parts of a partition and the former encapsulates all of the parts that make up a given partition.

```csharp
public class PartitionAsForest : AbstractSet, Partition
{
    protected PartitionTree[] array;

    protected class PartitionTree : AbstractSet, Set, Tree
    {
        protected PartitionAsForest partition;
        internal int item;
        internal PartitionTree parent;
        internal int rank;

        public PartitionTree(
            PartitionAsForest partition, int item) :
            base(partition.UniverseSize)
        {
            this.partition = partition;
            this.item = item;
            parent = null;
            rank = 0;
            count = 1;
        }
        // ...
    }
    // ...
}
```

Program: PartitionAsForest and PartitionTree fields.
The `PartitionTree` class extends the `AbstractSet` class defined in Program and it implements the `Tree` interface defined in Program. Since we are representing the parts of a partition using trees, it makes sense that they implement the `Tree` interface. On the other hand, since a partition is a set of sets, we must derive the parts of a partition from the `AbstractSet` class.

The `PartitionTree` class has four fields—`partition`, `item`, `parent`, and `rank`. Each instance of this class represents one node of a tree. The `partition` field refers to the `PartitionAsForest` instance that contains this tree. The `parent` field refers to the parent of a given node and the `item` field records the element of the universal set that the given node represents. The remaining variable, `rank`, is optional. While it is not required in order to provide the basic functionality, as shown below, the `rank` variable can be used in the implementation of the `Join` operation to improve the performance of subsequent `Find` operations.

The `PartitionAsForest` class represents a complete partition. The `PartitionAsForest` class extends the `AbstractSet` class defined in Program and it implements the `Partition` interface defined in Program. The `PartitionAsForest` class contains a single field, `array`, which is an array `PartitionTrees`. The $i^{th}$ element of the array always refers to the tree node that contains element $i$ of the universe.
Constructor

Program 1 gives the code for the `PartitionTree` constructor. The constructor creates a tree comprised of a single node. It takes an argument which specifies the element of the universal set that the node is to represent. The `parent` field is set to `null` to indicate that the node has no parent. Consequently, the node is a root node. Finally, the `rank` field is initialized to zero. The running time of the constructor is $O(1)$.

Program 2 shows the constructor for the `PartitionAsForest` class. The constructor takes a single argument $N$ which specifies that the universe shall be $U = \{0, 1, \ldots, N - 1\}$. It creates an initial partition of the universe consisting of $N$ parts. Each part contains one element of the universal set and, therefore, comprises a one-node tree.

```csharp
public class PartitionAsForest : AbstractSet, Partition
{
    protected PartitionTree[] array;

    public PartitionAsForest(int n) : base(n)
    {
        array = new PartitionTree[universeSize];
        for (int item = 0; item < universeSize; ++item)
            array[item] = new PartitionTree(this, item);
        count = universeSize;
    }
    // ...
}
```

Program: `PartitionAsForest` constructors.
Find and Join Methods

Two elements of the universe are in the same part of the partition if and only if they share the same root node. Since every tree has a unique root, it makes sense to use the root node as the "handle" for that tree. Therefore, the find operation takes an element of the universal set and returns the root node of the tree that contains that element. And because of way in which the trees are represented, we can follow the chain of parent pointers to find the root node.

Program gives the code for the Find method of the PartitionAsForest class. The Find method takes as its argument an int and returns a Set. The argument specifies the item of the universe that is the object of the search.

```csharp
public class PartitionAsForest : AbstractSet, Partition
{
    protected PartitionTree[] array;

    public virtual Set Find(int item)
    {
        PartitionTree ptr = array[item];
        while (ptr.parent != null)
        {
            ptr = ptr.parent;
        }
        return ptr;
    }

    // ...
}
```

Program: PartitionAsForest class Find method.

The Find operation begins at the node array[item] and follows the chain of parent fields to find the root node of the tree that contains the specified item. The result of the method is the root node.

The running time of the Find operation is $O(d)$ where $d$ is the depth in the tree of the node from which the search begins. If we don't do anything special to prevent it, the worst case running time is $O(N)$, where $N$ is the size of the universe. The best performance is achieved when every non-root node points to the root node. In this case, the running time is $O(1)$. 
Another advantage of having the parent field in each node is that the join operation can be implemented easily and efficiently. For example, suppose we wish to join the two sets $S_1$ and $S_2$ shown in Figure 1. While there are many possible representations for $S_1 \cup S_2$, it turns out that there are two simple alternatives which can be obtained in constant time. These are shown in Figure 2. In the first alternative, the root of $S_2$ is made a child of the root of $S_1$. This can be done in constant time simply by making the parent field of the root of $S_2$ refer to the root of $S_1$. The second alternative is essentially the same as the first except that the roles of $S_1$ and $S_2$ are exchanged.

**Figure:** Alternatives for joining elements of a partition.

Program 3 gives the simplest possible implementation for the join operation. The join method of the PartitionAsForest class takes two arguments—both of the references to Sets. Both arguments are required to be references to distinct PartitionTree instances which are contained in the given partition. Furthermore, both of them are required to be root nodes. Therefore, the sets that the arguments represent are disjoint. The method CheckArguments makes sure that the arguments satisfy these
Find and Join Methods

The Join operation is trivial and executes in constant time: It simply makes one node the parent of the other. In this case, we have arbitrarily chosen that the node specified by the first argument shall always become the parent.

```java
public class PartitionAsForest : AbstractSet, Partition
{
    protected PartitionTree[] array;

    protected virtual void CheckArguments(
        PartitionTree s, PartitionTree t)
    {
        if (!IsMember(s) || s.parent != null ||
            !IsMember(t) || t.parent != null || s == t)
            throw new ArgumentException("incompatible sets");
    }

    public virtual void Join(Set s, Set t)
    {
        PartitionTree p = (PartitionTree)s;
        PartitionTree q = (PartitionTree)t;
        CheckArguments(p, q);
        q.parent = p;
        --count;
    }
    // ...
}
```

Program: PartitionAsForest class simple Join method.
Collapsing Find

Unfortunately, using the Join algorithm given in Program can result in particularly bad trees. For example, Figure shows the worst possible tree that can be obtained. Such a tree is bad because its height is \( O(N) \). In such a tree both the worst case and the average case running time for the \texttt{Find} operation is \( O(N) \).

![Figure: A degenerate tree.]

There is an interesting trick we can play that can improve matters significantly. Recall that the find operation starts from a given node and locates the root of the tree containing that node. If, having found the root, we replace the parent of the given node with the root, the next time we do a \texttt{Find} it will be more efficient.

In fact, we can go one step further and replace the parent of every node along the search path to the root. This is called a \textit{collapsing find} operation. Doing so does not change the asymptotic complexity of the \texttt{Find} operation. However, a subsequent \texttt{Find} operation which begins at any point along the search path to the root will run in constant time!
Collapsing Find

Program gives the code for a collapsing version of the Find operation. The Find method first determines the root node as before. Then, a second pass is made up the chain from the initial node to the root, during which the parent of each node is assigned the root. Clearly, this version of Find is slower than the one given in Program because it makes two passes up the chain rather than one. However, the running of this version of Find is still $O(d)$, where $d$ is the depth of the node from which the search begins.

```java
public class PartitionAsForestV2 : PartitionAsForest
{
    public override Set Find(int item)
    {
        PartitionTree root = array[item];
        while (root.parent != null)
        {
            root = root.parent;
            PartitionTree ptr = array[item];
            while (ptr.parent != null)
            {
                PartitionTree tmp = ptr.parent;
                ptr.parent = root;
                ptr = tmp;
            }
        }
        return root;
    }
    // ...
}
```

**Program:** PartitionAsForest class collapsing Find method.

Figure illustrates the effect of a collapsing find operation. After the find, all the nodes along the search path are attached directly to the root. That is, they have had their depths decreased to one. As a side-effect, any node which is in the subtree of a node along the search path may have its depth decreased by the collapsing find operation. The depth of a node is never increased by the find operation. Eventually, if we do enough collapsing find operations, it is possible to obtain a tree of height one in which all the non-root nodes point directly at the root.
**Figure:** Example of collapsing find.
Union by Size

While using collapsing find does mitigate the negative effects of poor trees, a better approach is to avoid creating bad trees in the first place. As shown in Figure 1, when we join to trees we have a choice—which node should we choose to be the root of the new tree. A simple, but effective choice is to attach the smaller tree under the root of the larger one. In this case, the smaller tree is the one which has fewer nodes. This is the so-called union-by-size join algorithm. Program 2 shows how this can be done.

```csharp
public class PartitionAsForestV2 : PartitionAsForest
{
    public override void Join(Set s, Set t)
    {
        PartitionTree p = (PartitionTree)s;
        PartitionTree q = (PartitionTree)t;
        CheckArguments(p, q);
        if (p.Count > q.Count)
        {
            q.parent = p;
            p.Count += q.Count;
        }
        else
        {
            p.parent = q;
            q.Count += p.Count;
        }
        --count;
    }
    // ...
}
```

**Program:** PartitionAsForest class union-by-size Join method.

The implementation uses the count field of the Container class, from which PartitionTree is derived, to keep track of the number of items contained in the tree. (Since each node contains one item
from the universal set, the number of items contained in a tree is equal to the number of nodes in that tree). The algorithm simply selects the tree with the largest number of nodes to become the root of the result and attaches the root of the smaller tree under that of the larger one. Clearly, the running time of the union-by-size version of \texttt{Join} is \(O(1)\).

The following theorem shows that when using the union-by-size join operation, the heights of the resulting trees grow logarithmically.

**Theorem** Consider an initial partition \(P\) of the universe \(U = \{0, 1, \ldots, N - 1\}\) comprised of \(N\) sets of size 1. Let \(S\) be an element of the partition obtained from \(P\) after some sequence of union-by-size join operations, such that \(|S|=n\) for some \(n \geq 1\). Let \(T\) be the tree representing the set \(S\). The height of tree \(T\) satisfies the inequality

\[
 h \leq \left\lfloor \log_2 n \right\rfloor.
\]

**Proof (By induction).**

**Base Case** Since a tree comprised of a single node has height zero, the theorem clearly holds for \(n=1\).

**Inductive Hypothesis** Suppose the theorem holds for trees containing \(n\) nodes for \(n = 1, 2, \ldots, k\) for some \(k \geq 1\). Consider a union-by-size join operation that produces a tree containing \(k+1\) nodes. Such a tree is obtained by joining a tree \(T_l\) having \(l \leq k\) nodes with another tree \(T_m\) that has \(m \leq k\) nodes, such that \(l+m=k+1\).

Without loss of generality, suppose \(1 \leq l \leq (k + 1)/2\). As a result, \(l\) is less than or equal to \(m\). Therefore, the union-by-size algorithm will attach \(T_l\) under the root of \(T_m\). Let \(h_l\) and \(h_m\) be the heights of \(T_l\) and \(T_m\) respectively. The height of the resulting tree is \(\max(h_l + 1, h_m)\).

According to the inductive hypothesis, the height of \(T_m\) is given by

\[
 h_m \leq \left\lfloor \log_2 m \right\rfloor \\
\leq \left\lfloor \log_2(k + 1 - l) \right\rfloor \\
\leq \left\lfloor \log_2(k + 1) \right\rfloor.
\]

Similarly, the quantity \(h_l + 1\) is bounded by
\[
\begin{align*}
    h_{n+1} + 1 & \leq \lfloor \log_2 n \rfloor + 1 \\
    & \leq \lfloor \log_2 ((n + 1)/2) \rfloor + 1 \\
    & \leq \lfloor \log_2 (n + 1) \rfloor.
\end{align*}
\]

Therefore, the height of the tree containing \(k+1\) nodes is no greater than \(\max(h_1 + 1, h_m) = \lfloor \log_2 (k + 1) \rfloor\). By induction on \(k\), the theorem holds for all values of \(n \geq 1\).

Note that Theorem \(\Box\) and its proof does not require that we use the collapsing find algorithm of Section \(\Box\). That is, the height of a tree containing \(n\) nodes is guaranteed to be \(O(\log n)\) when the simple find is used. Of course, there is nothing precluding the use of the collapsing find in conjunction with the union-by-size join method. And doing so only makes things better.
Union by Height or Rank

The union-by-size join algorithm described above controls the heights of the trees indirectly by basing the join algorithm on the sizes of the trees. If we explicitly keep track of the height of a node in the node itself, we can accomplish the same thing.

Program gives an implementation of the Join method that always attaches the shorter tree under the root of the taller one. This method assumes that the rank field is used to keep track of the height of a node. (The reason for calling it rank rather than height will become evident shortly).

```csharp
public class PartitionAsForestV3 : PartitionAsForestV2
{
    public override void Join(Set s, Set t)
    {
        PartitionTree p = (PartitionTree)s;
        PartitionTree q = (PartitionTree)t;
        CheckArguments(p, q);
        if (p.rank > q.rank)
            q.parent = p;
        else
        {
            p.parent = q;
            if (p.rank == q.rank)
                q.rank += 1;
        }
        --count;
    }
    // ...
}
```

Program: PartitionAsForest class union-by-rank Join method.

The only time that the height of node increases is when joining two trees that have the same height. In this case, the height of the root increases by exactly one. If the two trees being joined have different
Union by Height or Rank

heights, attaching the shorter tree under the root of the taller one has no effect on the height of the root.

Unfortunately, there is a slight complication if we combine union-by-height with the collapsing find. Since the collapsing find works by moving nodes closer to the root, it affects potentially the height of any node moved. It is not at all clear how to recompute efficiently the heights that have changed. The solution is not to do it at all!

If we don't recompute the heights during the collapsing find operations, then the height fields will no longer be exact. Nevertheless, the quantities remain useful estimates of the heights of nodes. We call the estimated height of a node its rank and the join algorithm which uses rank instead of height is called union by rank.

Fortunately, Theorem □ applies equally well when union-by-rank is used. That is, the height of tree which contains \( n \) nodes is \( \mathcal{O}(\log n) \). Thus, the worst-case running time for the Find operation grows logarithmically with \( n \). And as before, collapsing find only makes things better.
Applications

One of the most important applications of partitions involves the processing of equivalence relations. Equivalence relations arise in many interesting contexts. For example, two nodes in an electric circuit are electrically equivalent if there is a conducting path (a wire) connecting the two nodes. In effect, the wires establish an electrical equivalence relation over the nodes of a circuit.

A similar relation arises among the classes in a C# program. Consider the following C# code fragment:

```csharp
interface I {}
class A : I {}
class B : I {}
class C : A {}
class D : B {}

The three classes A, B, C and D are equivalent in the sense that they all implement the same interface I. In effect, the class declarations establish an equivalence relation over the classes in a C# program.

Definition (Equivalence Relation) An equivalence relation over a universal set \( U \) is a relation \( \equiv \) with the following properties:

1. The relation \( \equiv \) is reflexive. That is, for every \( x \in U \), \( x \equiv x \).
2. The relation \( \equiv \) is symmetric. That is, for every pair \( x \in U \) and \( y \in U \), if \( x \equiv y \) then \( y \equiv x \).
3. The relation \( \equiv \) is transitive. That is, for every triple \( x \in U \), \( y \in U \) and \( z \in U \), if \( x \equiv y \) and \( y \equiv z \) then \( x \equiv z \).

An important characteristic of an equivalence relation is that it partitions the elements of the universal set \( U \) into a set of equivalence classes. That is, \( U \) is partitioned into \( P = \{ S_1, S_2, \ldots, S_p \} \), such that for every pair \( x \in U \) and \( y \in U \), \( x \equiv y \) if and only if \( x \) and \( y \) are in the same element of the partition. That is, \( x \equiv y \) if there exists a value of \( i \) such that \( x \in S_i \land y \in S_i \).

For example, consider the universe \( U = \{0, 1, \ldots, 9\} \) and the equivalence relation \( \equiv \) defined over \( U \)
defines as follows:

\[ 0 \equiv 0, 1 \equiv 1, 2 \equiv 2, 3 \equiv 3, 4 \equiv 4, 5 \equiv 5, 6 \equiv 6, 7 \equiv 7, 8 \equiv 8, 9 \equiv 9. \quad (12.1) \]

This relation results in the following partition of \( U \):

\[ \{ \{0\}, \{1, 2\}, \{3, 4, 5\}, \{6, 7, 8, 9\} \}. \]

The list of equivalences in Equation 12.1 contains many redundancies. Since we know that the relation \( \equiv \) is reflexive, symmetric and transitive, it is possible to infer the complete relation from the following list

\[ 1 \equiv 2, 3 \equiv 4, 3 \equiv 5, 6 \equiv 7, 6 \equiv 8, 6 \equiv 9. \]

The problem of finding the set of equivalence classes from a list of equivalence pairs is easily solved using a partition. Program 12.1 shows how it can be done using the \texttt{PartitionAsForest} class defined in Section 12.1.
public class Algorithms {
    public static void EquivalenceClasses(
        TextReader reader, TextWriter writer)
    {
        string line = reader.ReadLine();
        int n = Convert.ToInt32(line);
        Partition p = new PartitionAsForest(n);

        while ((line = reader.ReadLine()) != null)
        {
            string[] words = line.Split(null);
            int i = Convert.ToInt32(words[0]);
            int j = Convert.ToInt32(words[1]);
            Set s = p.Find(i);
            Set t = p.Find(j);
            if (s != t)
                p.Join(s, t);
            else
                writer.WriteLine("redundant pair:{0},{1}", i, j);
        }
        writer.WriteLine(p);
    }
}

Program: Application of disjoint sets--finding equivalence classes.

The algorithm first gets a positive integer \( n \) from the input and creates a partition, \( p \), of the universe \( U = \{0, 1 \ldots, n-1\} \) (lines 7-12). As explained in Section, the initial partition comprises \( n \) disjoint sets of size one. That is, each element of the universal set is in a separate element of the partition.

Each iteration of the main loop processes one equivalence pair (lines 10-21). An equivalence pair consists of two numbers, \( i \) and \( j \), such that \( i \in U \) and \( j \in U \). The find operation is used to determine the sets \( s \) and \( t \) in partition \( p \) that contain elements \( i \) and \( j \), respectively (lines 15-16).

If \( s \) and \( t \) are not the same set, then the disjoint sets are united using the join operation (lines 17-18). Otherwise, \( i \) and \( j \) are already in the same set and the equivalence pair is redundant (line 20). After all the pairs have been processed, the final partition is printed (line 22).
1. For each of the following implementations derive an expression for the total memory space required to represent a set which contains of \( n \) elements drawn from the universe \( U = \{0, 1, \ldots, N - 1\} \).
   1. `SetAsArray` (Program),
   2. `SetAsBitVector` (Program),
   3. `MultisetAsArray` (Program), and
   4. `MultisetAsLinkedList` (Program).

2. In addition to \( = \) and \( \subseteq \), a complete repertoire of set operators includes \( \cap \), \( \cup \), \( \Delta \) and \( \not\subseteq \). For each of the set implementations listed in Exercise show how to implement the remaining operators.

3. The *symmetric difference* of two sets \( S \) and \( T \), written \( S \Delta T \) is given by
   \[
   S \Delta T = (S \cup T) - (S \cap T).
   \]

   For each of the set implementations listed in Exercise devise an algorithm to compute symmetric difference. What is the running time of your algorithm?

4. The *complement* of a set \( S \) over universe \( U \), written \( S' \) is given by
   \[
   S' = U - S.
   \]

   Devise an algorithm to compute the complement of a set represented as a bit vector. What is the running time of your algorithm?

5. Devise an algorithm to sort a list of integers using a multiset. What is the running time of your algorithm? **Hint:** See Section.

6. Consider a multiset implemented using linked lists. When the multiset contains duplicate items, each of those items occupies a separate list element. An alternative is to use a linked list of ordered pairs of the form \((i, n_i)\) where \(i\) is an element of the universal set \( U \) and \( n_i \) is a non-negative integer that counts the number of instances of the element \( i \) in the multiset.

   Derive an expression for the total memory space required to represent a multiset which contains of \( n \) instances of \( m \) distinct element drawn from the universe \( U = \{0, 1, \ldots, N - 1\} \).
7. Consider a multiset implemented as described in Exercise [ ]. Devise algorithms for set union, intersection, and difference. What are the running times of your algorithms?

8. Consider the initial partition \( P = \{\{0\}, \{1\}, \{2\}, \ldots, \{9\}\} \). For each of the methods of computing the union listed below show the result of the following sequence join operations:

\[
\text{join}(0, 1), \text{join}(2, 3), \text{join}(2, 4), \text{join}(2, 5), \text{join}(6, 7), \text{join}(8, 9), \text{join}(6, 8), \text{join}(0, 6), \text{join}(0, 2).
\]

   1. simple union,
   2. union by size,
   3. union by height, and
   4. union by rank.

9. For each final partition obtained in Exercise [ ], show the result of performing a collapsing find operation for item 9.

10. Consider the initial partition \( P \) of the universe \( U = \{0, 1, \ldots, N-1\} \) comprised of \( N \) sets [24].

    1. Show that \( N \)-1 join operations can be performed before the number of elements in the partition is reduced to one.
    2. Show that if \( n \) join operations are done ( \( 0 \leq n < N \)), the size of the largest element of the partition is at most \( n+1 \).
    3. A singleton is an element of a partition that contains only one element of the universal set. Show that when \( n \) join operations are done ( \( 0 \leq n < N \)), at least \( \max\{N - 2n, 0\} \) singletons are left.
    4. Show that if less than \( \lceil N/2 \rceil \) join operations are done, at least one singleton is left.
Projects

1. Complete the `SetAsArray` class introduced in Program by providing suitable definitions for the following operations: Purge, IsEmpty, IsFull, Count, Accept, and GetEnumerator. Write a test program and test your implementation.

2. Complete the `SetAsBitVector` class introduced in Program by providing suitable definitions for the following methods: Purge, IsEmpty, IsFull, Count, Accept, and GetEnumerator. Write a test program and test your implementation.

3. Rewrite the Insert, Withdraw, and IsMember methods of the `SetAsBitVector` implementation so that they use bitwise shift and mask operations rather than division and modulo operations. Compare the running times of the modified methods with the original ones and explain your observations.

4. Complete the `MultisetAsArray` class introduced in Program by providing suitable definitions for the following methods: Purge, Count, Accept, and GetEnumerator. Write a test program and test your implementation.

5. Complete the `MultisetAsLinkedList` class introduced in Program by providing suitable definitions for the following methods: Purge, IsEmpty, IsFull, Count, CompareTo, Accept, and GetEnumerator. Write a test program and test your implementation.

6. Design and implement a multiset class in which the contents of the set are represented by a linked list of ordered pairs of the form \((i, n_i)\), where \(i\) an the element of the universal set \(U\) and \(n_i\) is a non-negative integer that counts the number of instances of the element \(i\) in the multiset. (See Exercises and ).

7. Write a program to compute the number of ways in which a set of \(n\) elements can be partitioned. That is, compute \(\sum_{m=0}^{n} \binom{n}{m}\) where

\[
\binom{n}{m} = \begin{cases} 1 & n = 1, \\ 1 & n = m, \\ m^{\binom{n-1}{m}} + \binom{n-1}{m-1} & \text{otherwise}. \end{cases}
\]

**Hint:** See Section .
Garbage Collection and the Other Kind of Heap

A C# object is an instance of a class, an array instance, or a delegate. Every object instance in a C# program occupies some memory. The manner in which a C# object is represented in memory is left up to the implementor of the common language runtime and, in principle, can vary from one implementation to another. However, object data typically occupy contiguous memory locations.

The region of memory in which objects are allocated dynamically is often called a heap. In Chapter we consider heaps and heap-ordered trees in the context of priority queue implementations. Unfortunately, the only thing that the heaps of Chapter and the heap considered here have in common is the name. While it may be possible to use a heap (in the sense of Definition) to manage a region of memory, typical implementations do not. In this context the technical meaning of the term heap is closer to its dictionary definition--``a pile of many things.''

The amount of memory required to represent a C# object is determined by the number and the types of its fields. For example, fields of the C# simple types, occupy between one and sixteen bytes. E.g., bool occupies one byte; char, short, and ushort occupy two bytes; int, uint, and float occupy four bytes; long, ulong, and double occupy eight bytes; and decimal occupies sixteen bytes. A field which refers to an object or to an interface typically requires only four bytes.

In addition to the memory required for the fields of an object, there is a fixed, constant amount of extra storage set aside in every object (eight bytes). This extra storage carries information used by the common language runtime to make sure that object is used correctly and to aid the process of garbage collection.

Every object in a C# program is created explicitly by invoking the new operator. Invoking the new operator causes the common language runtime to perform the following steps:

1. An unused region of memory large enough to hold an instance of the desired class is found.
2. All of the fields of the object are assigned their default initial values.
3. The appropriate constructor is run to initialize the object instance.
4. A reference to the newly created object is returned.
• What is Garbage?
• Reference Counting Garbage Collection
• Mark-and-Sweep Garbage Collection
• Stop-and-Copy Garbage Collection
• Mark-and-Compact Garbage Collection
• Exercises
• Projects
What is Garbage?

While C# provides the means to create an object, the language does not provide the means to destroy an object explicitly. As long as a program contains a reference to some object instance, the common language runtime is required to ensure that the object exists. If the C# language provided the means to destroy objects, it would be possible for a program to destroy an object even when a reference to that object still existed. This situation is unsafe because the program could attempt later to invoke a method on the destroyed object, leading to unpredictable results.

The situation which arises when a program contains a reference (or pointer) to a destroyed object is called a dangling reference (or dangling pointer). By disallowing the explicit destruction of objects, C# eliminates the problem of dangling references.

Languages that support the explicit destruction of objects typically require the program to keep track of all the objects it creates and to destroy them explicitly when they are not longer needed. If a program somehow loses track of an object it has created then that object cannot be destroyed. And if the object is never destroyed, the memory occupied by that object cannot be used again by the program.

A program that loses track of objects before it destroys them suffers from a memory leak. If we run a program that has a memory leak for a very long time, it is quite possible that it will exhaust all the available memory and eventually fail because no new objects can be created.

It would seem that by disallowing the explicit destruction of objects, a C# program is doomed to eventual failure due to memory exhaustion. Indeed this would be the case, were it not for the fact that the C# language specification requires the common language runtime to be able to find unreferenced objects and to reclaim the memory locations allocated to those objects. An unreferenced object is called garbage and the process of finding all the unreferenced objects and reclaiming the storage is called garbage collection.

Just as the C# language does not specify precisely how objects are to be represented in the memory of a common language runtime, the language specification also does not stipulate how the garbage collection is to be implemented or when it should be done. Garbage collection is usually invoked when the total amount of memory allocated to a C# program exceeds some threshold. Typically, the program is suspended while the garbage collection is done.

In the analyses presented in the preceding chapters we assume that the running time of the new operator...
is a fixed constant, $\tau_{\text{new}}$, and we completely ignore the garbage collection overhead. In reality, neither assumption is valid. Even if sufficient memory is available, the time required by the common language runtime to locate an unused region of memory depends very much on the data structures used to keep track of the memory regions allocated to a program as well as on the way in which a program uses the objects it creates. Furthermore, invoking the `new` operator may trigger the garbage collection process. The running time for garbage collection can be a significant fraction of the total running time of a program.

---

- **Reduce, Reuse, Recycle**
- **Helping the Garbage Collector**

[Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.]
Reduce, Reuse, Recycle

Modern societies produce an excessive amount of waste. The costs of doing so include the direct costs of waste disposal as well as the damage to the environment caused by the manufacturing, distribution, and ultimate disposal of products. The slogan "reduce, reuse, recycle," prescribes three strategies for reducing the environmental costs associated with waste materials.

These strategies apply equally well to C# programs! A C# program that creates excessive garbage may require more frequent garbage collection than a program that creates less garbage. Since garbage collection can take a significant amount of time to do, it makes sense to use strategies that decrease the cost of garbage collection.

- **Reduce**
- **Reuse**
- **Recycle**
Reduce

A C# program that does not create any object instances or arrays does not create garbage. Similarly, a program that creates all the objects it needs at the beginning of its execution and uses the same objects until it terminates also does not create garbage. By reducing the number of objects a program creates dynamically during its execution, we can reduce or even eliminate the need for garbage collection.
Reuse

Sometimes, a C# program will create many objects which are used only once. For example, a program may create an object in the body of a loop that is used to hold "temporary" information that is only required for the particular iteration of the loop in which it is created. Consider the following:

```csharp
for (int i = 0; i < 1000000; ++i)
{
    SomeClass obj = new SomeClass(i);
    Console.WriteLine(obj);
}
```

This creates a million instances of the `SomeClass` class and prints them out. If the `SomeClass` class has a property, say `Value`, that provides a `set` accessor, we can reuse an a single object instance like this:

```csharp
SomeClass obj = new SomeClass();
for (int i = 0; i < 1000000; ++i)
{
    obj.Value = i;
    Console.WriteLine(obj);
}
```

Clearly, by reusing a single object instance, we have dramatically reduced the amount of garbage produced.
Recycle

Recycling of objects is a somewhat more complex strategy for reducing the overhead associated with garbage collection. Instead of leaving an unused object around for the garbage collector to find, it is put into a container of unused objects. When a new object is needed, the container is searched first to see if an unused one already exists. Because a container always refers to the objects it contains, those objects are never garbage collected.

The recycling strategy can indeed reduce garbage collection overhead. However, it puts the burden back on the programmer to explicitly put unused objects into the container (avoid memory leaks) and to make sure objects put into the container are really unused (avoid dangling references). Because the recycling strategy undermines some of the benefits of garbage collection, it should be used with great care.
Helping the Garbage Collector

The preceding section presents strategies for avoiding garbage collection. However, there are times when garbage collection is actually desirable. Imagine a program that requires a significant amount of memory. Suppose the amount of memory required is very close to the amount of memory available for use by the common language runtime. The performance of such a program is going to depend on the ability of the garbage collector to find and reclaim as much unused storage as possible. Otherwise, the garbage collector will run too often. In this case, it pays to help out the garbage collector.

How can we help out the garbage collector? Since the garbage collector collects only unreferenced objects it is necessary to eliminate all references to objects which are no longer needed. This is done by assigning the value `null` to every variable that refers to an object that is no longer needed. Consequently, helping the garbage collector requires a program to do a bit more work.
Reference Counting Garbage Collection

The difficulty in garbage collection is not the actual process of collecting the garbage—it is the problem of finding the garbage in the first place. An object is considered to be garbage when no references to that object exist. But how can we tell when no references to an object exist?

A simple expedient is to keep track in each object of the total number of references to that object. That is, we add a special field to each object called a reference count. The idea is that the reference count field is not accessible to the C# program. Instead, the reference count field is updated by the common language runtime itself.

Consider the statement

```csharp
object p = new ComparableInt32(57);
```

which creates a new instance of the `ComparableInt32` class. Only a single variable, `p`, refers to the object. Thus, its reference count should be one.

Now consider the following sequence of statements:

```csharp
object p = new ComparableInt32(57); object q = p;
```

This sequence creates a single `ComparableInt32` instance. Both `p` and `q` refer to the same object. Therefore, its reference count should be two.

In general, every time one reference variable is assigned to another, it may be necessary to update several
reference counts. Suppose \( p \) and \( q \) are both reference variables. The assignment

\[
p = q;
\]

would be implemented by the common language runtime as follows:

\[
\begin{aligned}
&\text{if (} p \neq q \text{)} \\
&\quad \begin{aligned}
&\quad \text{if (} p \neq \text{null}) \\
&\quad \quad --p\text{.refCount;}
&\quad p = q;
&\quad \text{if (} p \neq \text{null}) \\
&\quad \quad ++p\text{.refCount;}
&\end{aligned}
\end{aligned}
\]

For example suppose \( p \) and \( q \) are initialized as follows:

\[
\text{object } p = \text{new ComparableInt32}(57);
\]
\[
\text{object } q = \text{new ComparableInt32}(99);
\]

As shown in Figure (a), two ComparableInt32 objects are created, each with a reference count of one. Now, suppose we assign \( q \) to \( p \) using the code sequence given above. Figure (b) shows that after the assignment, both \( p \) and \( q \) refer to the same object--its reference count is two. And the reference count on ComparableInt32(57) has gone to zero which indicates that it is garbage.

---

**Figure:** Reference counts before and after the assignment \( p = q \).

The costs of using reference counts are twofold: First, every object requires the special reference count field. Typically, this means an extra word of storage must be allocated in each object. Second, every time one reference is assigned to another, the reference counts must be adjusted as above. This increases significantly the time taken by assignment statements.
The advantage of using reference counts is that garbage is easily identified. When it becomes necessary to reclaim the storage from unused objects, the garbage collector needs only to examine the reference count fields of all the objects that have been created by the program. If the reference count is zero, the object is garbage.

It is not necessary to wait until there is insufficient memory before initiating the garbage collection process. We can reclaim memory used by an object immediately when its reference goes to zero. Consider what happens if we implement the C# assignment $p = q$ in the common language runtime as follows:

```csharp
if (p != q)
{
    if (p != null)
        if (--p.refCount == 0)
            heap.Release(p);
    p = q;
    if (p != null)
        ++p.refCount;
}
```

Notice that the `Release` method is invoked immediately when the reference count of an object goes to zero, i.e., when it becomes garbage. In this way, garbage may be collected incrementally as it is created.
When Objects Refer to Other Objects

The ComparableInt32 objects considered in the preceding examples are very simple objects—they contain no references to other objects. Reference counting is an ideal strategy for garbage collecting such objects. But what about objects that refer to other objects? For example, consider the Association class described in Chapter □ which represents a \((\text{key}, \text{value})\) pair. We can still use reference counting, provided we count all to an object including references from other objects.

Figure □ (a) illustrates the contents memory following the execution of this statement:

```csharp
object p = new Association(
    new ComparableComparableInt32(57),
    new ComparableInt32(99));
```

The reference count of the Association is one, because the variable \(p\) refers to it. Similarly, the reference counts of the two ComparableInt32 instances are one because the Association refers to both of them.
Figure: Reference counting when objects refer to other objects.

Suppose we assign the value `null` to the variable `p`. As shown in Figure (b), the reference count of the association becomes zero—it is now garbage. However, until the `Association` instance continues to exist until it is garbage collected. And because it still exists, it still refers to the `ComparableInt32` objects.

Figure (d) shows that the garbage collection process adjusts the reference counts on the objects to which the association refers only when the association is garbage collected. The two `ComparableInt32` objects are now unreferenced and can be garbage collected as well.
Why Reference Counting Does Not Work

So far, reference counting looks like a good idea. However, the reference counting does not always work. Consider a circular, singly-linked list such as the one shown in Figure (a). In the figure, the variable head refers to the head of the linked list and the last element of the linked list also refers to the head. Therefore, the reference count on the first list element is two; whereas, the remaining list elements each has a reference count of one.

Consider what happens when we assign the value null to the head variable. This results in the situation shown in Figure (b). The reference count on the first list element has been decreased by one because the head variable no longer refers to it. However, its reference count is not zero, because the tail of the list still refers to the head.

We now have a problem. The reference counts on all the lists elements are non-zero. Therefore, they are not considered to be garbage by a reference counting garbage collector. On the other hand, no external references to the linked-list elements remain. Therefore, the list elements are indeed garbage.

This example illustrates the Achilles' heel of reference counting--circular data structures. In general, reference counting will fail to work whenever the data structure contains a cycle of references. C# does not prevent the creation of cyclic structures. Therefore, reference counting by itself is not a suitable garbage collection scheme for arbitrary objects. Nevertheless, it is an extremely useful technique for
dealing with simple objects that don't refer to other objects, such as ComparableInt32s and strings.
This section presents the *mark-and-sweep* garbage collection algorithm. The mark-and-sweep algorithm was the first garbage collection algorithm to be developed that is able to reclaim cyclic data structures. Variations of the mark-and-sweep algorithm continue to be among the most commonly used garbage collection techniques.

When using mark-and-sweep, unreferenced objects are not reclaimed immediately. Instead, garbage is allowed to accumulate until all available memory has been exhausted. When that happens, the execution of the program is suspended temporarily while the mark-and-sweep algorithm collects all the garbage. Once all unreferenced objects have been reclaimed, the normal execution of the program can resume.

The mark-and-sweep algorithm is called a *tracing* garbage collector because it traces out the entire collection of objects that are directly or indirectly accessible by the program. The objects that a program can access directly are those objects which are referenced by local variables on the processor stack as well as by any static variables that refer to objects. In the context of garbage collection, these variables are called the *roots*. An object is indirectly accessible if it is referenced by a field in some other (directly or indirectly) accessible object. An accessible object is said to be *live*. Conversely, an object which is not *live* is garbage.

The mark-and-sweep algorithm consists of two phases: In the first phase, it finds and marks all accessible objects. The first phase is called the *mark* phase. In the second phase, the garbage collection algorithm scans through the heap and reclaims all the unmarked objects. The second phase is called the *sweep* phase. The algorithm can be expressed as follows:

```plaintext
for each root variable r
    Mark(r);

Sweep();
```

In order to distinguish the live objects from garbage, we record the state of an object in each object. That
Mark-and-Sweep Garbage Collection

is, we add a special bool field to each object called, say, marked. By default, all objects are unmarked when they are created. Thus, the marked field is initially false.

An object p and all the objects indirectly accessible from p can be marked by using the following recursive Mark method:

```csharp
void Mark(object p)

    if(!p.marked)

        p.marked = true;
        for each object q referenced by p
            Mark(q);
```

Notice that this recursive Mark algorithm does nothing when it encounters an object that has already been marked. Consequently, the algorithm is guaranteed to terminate. And it terminates only when all accessible objects have been marked.

In its second phase, the mark-and-sweep algorithm scans through all the objects in the heap, in order to locate all the unmarked objects. The storage allocated to the unmarked objects is reclaimed during the scan. At the same time, the marked field on every live object is set back to false in preparation for the next invocation of the mark-and-sweep garbage collection algorithm:

```csharp
void Sweep()

    for each object p in the heap

        if (p.marked)
            p.marked = false
        else
            heap.Release(p);
```

Figure illustrates the operation of the mark-and-sweep garbage collection algorithm. Figure (a) shows the conditions before garbage collection begins. In this example, there is a single root variable.
Figure (b) shows the effect of the *mark* phase of the algorithm. At this point, all live objects have been marked. Finally, Figure (c) shows the objects left after the *sweep* phase has been completed. Only live objects remain in memory and the marked fields have all been set to *false* again.

Because the mark-and-sweep garbage collection algorithm traces out the set of objects accessible from the roots, it is able to correctly identify and collect garbage even in the presence of reference cycles. This is the main advantage of mark-and-sweep over the reference counting technique presented in the preceding section. A secondary benefit of the mark-and-sweep approach is that the normal manipulations
of reference variables incurs no overhead.

The main disadvantage of the mark-and-sweep approach is the fact that normal program execution is suspended while the garbage collection algorithm runs. In particular, this can be a problem in a program that interacts with a human user or that must satisfy real-time execution constraints. For example, an interactive application that uses mark-and-sweep garbage collection becomes unresponsive periodically.
The Fragmentation Problem

Fragmentation is a phenomenon that occurs in a long-running program that has undergone garbage collection several times. The problem is that objects tend to become spread out in the heap. Live objects end up being separated by many, small unused memory regions. The problem in this situation is that it may become impossible to allocate memory for an object. While there may indeed be sufficient unused memory, the unused memory is not contiguous. Since objects typically occupy consecutive memory locations it is impossible to allocate storage.

The mark-and-sweep algorithm does not address fragmentation. Even after reclaiming the storage from all garbage objects, the heap may still be too fragmented to allocate the required amount of space. The next section presents an alternative to the mark-and-sweep algorithm that also defragments (or compacts) the heap.
Stop-and-Copy Garbage Collection

The section describes a garbage collection approach that collects garbage and defragments the heap called *stop-and-copy*. When using the stop-and-copy garbage collection algorithm, the heap is divided into two separate regions. At any point in time, all dynamically allocated object instances reside in only one of the two regions—the *active* region. The other, *inactive* region is unoccupied.

When the memory in the active region is exhausted, the program is suspended and the garbage-collection algorithm is invoked. The stop-and-copy algorithm copies all of the live objects from the active region to the inactive region. As each object is copied, all references contained in that object are updated to reflect the new locations of the referenced objects.

After the copying is completed, the active and inactive regions exchange their roles. Since the stop-and-copy algorithm copies only the live objects, the garbage objects are left behind. In effect, the storage occupied by the garbage is reclaimed all at once when the active region becomes inactive.

As the stop-and-copy algorithm copies the live objects from the active region to the inactive region, it stores the objects in contiguous memory locations. Thus, the stop-and-copy algorithm automatically defragments the heap. This is the main advantage of the stop-and-copy approach over the mark-and-sweep algorithm described in the preceding section.

The costs of the stop-and-copy algorithm are twofold: First, the algorithm requires that all live objects be copied every time garbage collection is invoked. If an application program has a large memory footprint, the time required to copy all objects can be quite significant. A second cost associated with stop-and-copy is the fact that it requires twice as much memory as the program actually uses. When garbage collection is finished, at least half of the memory space is unused.

- The Copy Algorithm
The Copy Algorithm

The stop-and-copy algorithm divides the heap into two regions—an active region and an inactive region. For convenience, we can view each region as a separate heap and we shall refer to them as activeHeap and inactiveHeap. When the stop-and-copy algorithm is invoked, it copies all live objects from the activeHeap to the inactiveHeap. It does so by invoking the Copy method given below starting at each root:

\[
\text{for each root variable } r
\]

\[
r = \text{Copy}(r, \text{inactiveHeap});
\]

\[
\text{Swap(activeHeap, inactiveHeap)};
\]

The Copy method is complicated by the fact that it needs to update all object references contained in the objects as it copies those objects. In order to facilitate this, we record in every object a reference to its copy. That is, we add a special field to each object called forward which is a reference to the copy of this object.

The recursive Copy method given below copies a given object and all the objects indirectly accessible from the given object to the destination heap. When the forward field of an object is null, it indicates that the given object has not yet been copied. In this case, the method creates a new instance of the object class in the destination heap. Then, the fields of the object are copied one-by-one. If the field is a value type, the value of that field is copied. However, if the field refers to another object, the Copy method calls itself recursively to copy that object.

\[
\text{object Copy(object } p, \text{ Heap destination)}
\]

\[
\text{if } (p == \text{null})
\]

\[
\text{return null;}
\]

\[
\text{if } (p.\text{forward} == \text{null})
\]

\[
q = \text{destination}.\text{NewInstance}(p.\text{getType()});
\]
The Copy Algorithm

```java
    p.forward = q;
    for each field f in p

        if (f is a value type)
            q.f = p.f;
        else
            q.f = Copy(p.f, destination);

    q.forward = null;

    return p.forward;
```

If the `Copy` method is invoked for an object whose `forward` field is non-null, that object has already been copied and the `forward` field refers to the copy of that object in the destination heap. In that case, the `Copy` method simply returns a reference to the previously copied object.

Figure 1 traces the execution of the stop-and-copy garbage collection algorithm. When the algorithm is invoked and before any objects have been copied, the `forward` field of every object in the active region is null as shown in Figure 1(a). In Figure 1(b), a copy of object `A`, called `A'`, has been created in the inactive region, and the `forward` field of `A` refers to `A'`.

![Figure 1](http://www.brpreiss.com/books/opus6/html/page430.html)
The Copy Algorithm

Figure: Stop-and-copy garbage collection.

Since $A$ refers to $B$, the next object copied is object $B$. As shown in Figure (c), fragmentation is eliminated by allocating storage for $B'$ immediately next to $A'$. Next, object $C$ is copied. Notice that $C$ refers to $A$, but $A$ has already been copied. Object $C'$ obtains its reference to $A'$ from the forward field of $A$ as shown in Figure (d).

After all the live objects have been copied from the active region to the inactive region, the regions exchange their roles. As shown in Figure (e), all the garbage has been collected and the heap is no
The Copy Algorithm

longer fragmented.
Mark-and-Compact Garbage Collection

The mark-and-sweep algorithm described in Section has the unfortunate tendency to fragment the heap. The stop-and-copy algorithm described in Section avoids fragmentation at the expense of doubling the size of the heap. This section describes the mark-and-compact approach to garbage collection which eliminates fragmentation without the space penalty of stop-and-copy.

The mark-and-compact algorithm consists of two phases: In the first phase, it finds and marks all live objects. The first phase is called the mark phase. In the second phase, the garbage collection algorithm compacts the heap by moving all the live objects into contiguous memory locations. The second phase is called the compaction phase. The algorithm can be expressed as follows:

\[
\text{for each root variable } r \\
\quad \text{Mark}(r); \\
\quad \text{Compact}();
\]

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Handles

The common language runtime specification does not prescribe how reference variables are implemented. One approach is for a reference variable to be implemented as an index into an array of object handles. Every object instance has its own handle. The handle for an object typically contains a reference to a System.Type instance that describes the type of the object and a pointer to the region in the heap where the object data resides.

The advantage of using handles is that when the position in the heap of an object is changed, only the handle for that object needs to be modified. All other references to that object are unaffected because such references actually refer to the handle. The cost of using handles is that the handle must be dereferenced every time an object is accessed.

The mark-and-compact algorithm uses the handles in two ways: First, the marked flags which are set during the mark operation are stored in the handles rather than in the objects themselves. Second, compaction is greatly simplified because when an object is moved only its handle needs to be updated—all other objects are unaffected.

Figure (a) illustrates how object references are implemented using handles. Figure (a) shows a circular, singly-linked list as it is usually drawn and Figure (b) shows how the list is represented when using handles. Each reference variable actually contains an index into the array of handles. For example, the head variable selects the handle at offset 2 and that handle points to linked list element A. Similarly, the next field of list element A selects the handle at offset 5 which refers to list element B. Notice that when an object is moved, only its handle needs to be modified.
The handle is a convenient place in which to record information used by the garbage collection algorithm. For example, we add a \texttt{bool} field to each handle, called \texttt{marked}. The \texttt{marked} field is used to mark live objects as follows:

```c
void Mark(object p)
{
    if (!handle[p].marked)
    {
        handle[p].marked = true;
        for each object q referenced by p
            Mark(q);
    }
}
```

Notice that this version of the \texttt{Mark} method marks the object handles rather than the objects themselves.

Once all of the live objects in the heap have been identified, the heap needs to be defragmented. Perhaps the simplest way to defragment the heap is to \textit{slide} the objects in the heap all to one end, removing the unused memory locations separating them. The following version of the \texttt{Compact} method does just this:
void Compact()

    long offset = 0;
    for each object p in the heap

        if (handle[p].marked)

            handle[p].object = heap.Move(p, offset);
            handle[p].marked = false;
            offset += sizeof(p);

This algorithm makes a single pass through the objects in the heap, moving the live objects towards the lower heap addresses as it goes. The Compact method only modifies the object handles--object data remain unchanged. This algorithm also illustrates an important characteristic of the sliding compaction algorithm--the relative positions of the objects in the heap remains unchanged after the compaction operation. Also, when the compaction method has finished, the marked fields have all been set back to false in preparation for the next garbage collection operation.
Exercises

1. Let \( M \) be the size of the heap and let \( f \) be the fraction of the heap occupied by live data. Estimate the running time of the \texttt{Mark} method of the \textit{mark-and-sweep} garbage collection scheme as a function of \( f \) and \( M \).

2. Repeat Exercise \( \square \) for the \texttt{Copy} method Estimate the running time of the \texttt{Copy} method of the \textit{stop-and-copy} garbage collection scheme.

3. Repeat Exercise \( \square \) for the \texttt{Copy} method Estimate the running time of the \texttt{Compact} method of the \textit{stop-and-compact} garbage collection scheme.

4. Using your answers to Exercises \( \square, \square \) and \( \square \), show that running time of garbage collection is inversely proportional to the amount of storage recovered.

5. The efficiency of a garbage collection scheme is the rate at which memory is reclaimed. Using your answers to Exercises \( \square \) and \( \square \) compare the efficiency of \textit{mark-and-sweep} with that of \textit{stop-and-copy}.

6. Devise a non-recursive algorithm for the \texttt{Mark} method of the \textit{mark-and-sweep} garbage collection scheme.

7. Repeat Exercise \( \square \) for the \texttt{Copy} method of the \textit{stop-and-copy} garbage collection scheme.

8. Repeat Exercise \( \square \) for the \texttt{Mark} method of the \textit{mark-and-compact} garbage collection scheme.

9. Consider the use of \textit{handles} for representing object references. Is it correct to assume that the order which objects appear in the heap is the same as the order in which the corresponding handles appear in the array of handles? How does this affect compaction of the heap?

10. Consider the \texttt{Compact} method of the \textit{mark-and-compact} garbage collection scheme. The algorithm visits the objects in the heap in the order in which they appear in the heap, rather than in the order in which the corresponding handles appear in the array of handles. Why is this necessary?

11. The \texttt{Compact} method of the \textit{mark-and-compact} garbage collection scheme slides the objects in the heap all to one end, but leaves the handles where they are. As a result, the handle array becomes fragmented. What modifications are necessary in order to compact the handle array as well as the heap?
Projects

1. Devise and conduct a set of experiments to measure garbage collection overhead. For example, write a program that creates a specified number of garbage objects as quickly as possible. Determine the number of objects needed to trigger garbage collection. Measure the running time of your program when no garbage collection is performed and compare it to the running time observed when garbage collection is invoked.

2. C# provides the means for accessing memory directly by using pointers and unsafe blocks. Therefore, we can simulate a heap using a C# array of ints. Write a C# class that manages an array of ints. Your class should implement the following interface:

```csharp
public interface Heap
{
    int Acquire(int size);
    int Release(int offset);
    int this[int offset] { get; set; }
}
```

The Acquire method allocates a region of size consecutive ints in the array and returns the offset of the first int in the region. The Release method release a region of ints at the specified offset which was obtained previously using Acquire. The indexer this[] provides get and set accessors to access a value in the array at a given offset.

3. Using an array of ints simulate the mark-and-sweep garbage collection as follows:
   1. Write a class that implements the Handle interface given below:

```csharp
public interface Handle
{
    int Size { get; }
    int GetInt(int offset);
    Handle GetReference(int offset);
    SetInt(int offset, int value);
    SetReference(int offset, Handle h);
}
```

A handle refers to an object that contains either ints or other handles. The size of an object is total the number of ints and handles it contains. The various store and fetch
methods are used to insert and remove items from the object to which this handle refers.

2. Write a class that implements the `Heap` interface given below

```java
public interface Heap {
    Handle Acquire(int size);
    void Release(Handle h);
    void CollectGarbage();
}
```

The `Acquire` method allocates a handle and space in the heap for an object of the given size. The `Release` method releases the given handle but does not reclaim the associated heap space. The `CollectGarbage` method performs the actual garbage collection operation.

4. Using the approach described in Project 0, implement a simulation of *mark-and-compact* garbage collection.

5. Using the approach described in Project 0 implement a simulation of *reference-counting* garbage collection.
This chapter presents a number of different algorithmic patterns. Each pattern addresses a category of problems and describes a core solution strategy for that category. Given a problem to be solved, we may find that there are several possible solution strategies. We may also find that only one strategy applies or even that none of them do. A good programmer is one who is proficient at examining the problem to be solved and identifying the appropriate algorithmic technique to use. The following algorithmic patterns are discussed in this chapter:

- **direct solution strategies**
  - Brute force algorithms and greedy algorithms.
- **backtracking strategies**
  - Simple backtracking and branch-and-bound algorithms.
- **top-down solution strategies**
  - Divide-and-conquer algorithms.
- **bottom-up solution strategies**
  - Dynamic programming.
- **randomized strategies**
  - Monte Carlo algorithms and simulated annealing.
In this section we consider two closely related algorithm types—brute-force and greedy. Brute-force algorithms are distinguished not by their structure or form, but by the way in which the problem to be solved is approached. A brute-force algorithm solves a problem in the most simple, direct or obvious way. As a result, such an algorithm can end up doing far more work to solve a given problem than a more clever or sophisticated algorithm might do. On the other hand, a brute-force algorithm is often easier to implement than a more sophisticated one and, because of this simplicity, sometimes it can be more efficient.

Often a problem can be viewed as a sequence of decisions to be made. For example, consider the problem of finding the best way to place electronic components on a circuit board. To solve this problem we must decide where on the board to place each component. Typically, a brute-force algorithm solves such a problem by exhaustively enumerating all the possibilities. That is, for every decision we consider each possible outcome.

A greedy algorithm is one that makes the sequence of decisions (in some order) such that once a given decision has been made, that decision is never reconsidered. For example, if we use a greedy algorithm to place the components on the circuit board, once a component has been assigned a position it is never again moved. Greedy algorithms can run significantly faster than brute force ones. Unfortunately, it is not always the case that a greedy strategy leads to the correct solution.

- Example-Counting Change
- Example-0/1 Knapsack Problem
Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Example-Counting Change

Consider the problem a cashier solves every time he counts out some amount of currency. The cashier has at his disposal a collection of notes and coins of various denominations and is required to count out a specified sum using the smallest possible number of pieces.

The problem can be expressed mathematically as follows: Let there be $n$ pieces of money (notes or coins), $P = \{p_1, p_2, \ldots, p_n\}$, and let $d_i$ be the denomination of $p_i$. For example, if $p_i$ is a dime, then $d_i = 10$. To count out a given sum of money $A$ we find the smallest subset of $P$, say $S \subseteq P$, such that $\sum_{p_i \in S} d_i = A$.

One way to represent the subset $S$ is to use $n$ variables $X = \{x_1, x_2, \ldots, x_n\}$, such that

$$x_i = \begin{cases} 1 & p_i \in S, \\ 0 & p_i \notin S. \end{cases}$$

Given $\{d_1, d_2, \ldots, d_n\}$ our objective is to minimize

$$\sum_{i=1}^{n} x_i$$

subject to the constraint

$$\sum_{i=1}^{n} d_i x_i = A.$$
Example-Counting Change

- **Brute-Force Algorithm**
- **Greedy Algorithm**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Brute-Force Algorithm

Since each of the elements of $X = \{x_1, x_2, \ldots, x_n\}$ is either a zero or a one, there are $2^n$ possible values for $X$. A brute-force algorithm to solve this problem finds the best solution by enumerating all the possible values of $X$.

For each possible value of $X$ we check first if the constraint $\sum_{i=1}^{n} d_i x_i = A$ is satisfied. A value which satisfies the constraint is called a feasible solution. The solution to the problem is the feasible solution which minimizes $\sum_{i=1}^{n} x_i$ which is called the objective function.

Since there are $2^n$ possible values of $X$ the running time of a brute-force solution is $\Omega(2^n)$. The running time needed to determine whether a possible value is a feasible solution is $O(n)$ and the time required to evaluate the objective function is also $O(n)$. Therefore, the running time of the brute-force algorithm is $O(n2^n)$.
Greedy Algorithm

A cashier does not really consider all the possible ways in which to count out a given sum of money. Instead, he counts out the required amount beginning with the largest denomination and proceeding to the smallest denomination.

For example, suppose we have ten coins: five pennies, two nickels, two dimes, and one quarter. That is, \(d_1, d_2, \ldots, d_{10}\) = \(\{1, 1, 1, 1, 5, 5, 10, 10, 25\}\). To count out 32 cents, we start with a quarter, then add a nickel followed by two pennies. This is a greedy strategy because once a coin has been counted out, it is never taken back. Furthermore, the solution obtained is the correct solution because it uses the fewest number of coins.

If we assume that the pieces of money (notes and coins) are sorted by their denomination, the running time for the greedy algorithm is \(O(n)\). This is significantly better than that of the brute-force algorithm given above.

Does this greedy algorithm always produce the correct answer? Unfortunately it does not. Consider what happens if we introduce a 15-cent coin. Suppose we are asked to count out 20 cents from the following set of coins: \(\{1, 1, 1, 1, 10, 10, 15\}\). The greedy algorithm selects 15 followed by five ones--six coins in total. Of course, the correct solution requires only two coins. The solution found by the greedy strategy is a feasible solution, but it does not minimize the objective function.
Example-0/1 Knapsack Problem

The 0/1 knapsack problem is closely related to the change counting problem discussed in the preceding section: We are given a set of \( n \) items from which we are to select some number of items to be carried in a knapsack. Each item has both a weight and a profit. The objective is to choose the set of items that fits in the knapsack and maximizes the profit.

Let \( w_i \) be the weight of the \( i^{\text{th}} \) item, \( p_i \) be the profit accrued when the \( i^{\text{th}} \) item is carried in the knapsack, and \( C \) be the capacity of the knapsack. Let \( x_i \) be a variable the value of which is either zero or one. The variable \( x_i \) has the value one when the \( i^{\text{th}} \) item is carried in the knapsack.

Given \( \{w_1, w_2, \ldots, w_n\} \) and \( \{p_1, p_2, \ldots, p_n\} \), our objective is to maximize

\[
\sum_{i=1}^{n} p_i x_i
\]

subject to the constraint

\[
\sum_{i=1}^{n} w_i x_i \leq C.
\]

Clearly, we can solve this problem by exhaustively enumerating the feasible solutions and selecting the one with the highest profit. However, since there are \( 2^n \) possible solutions, the running time required for the brute-force solution becomes prohibitive as \( n \) gets large.

An alternative is to use a greedy solution strategy which solves the problem by putting items into the knapsack one-by-one. This approach is greedy because once an item has been put into the knapsack, it is never removed.

How do we select the next item to be put into the knapsack? There are several possibilities:

**Greedy by Profit**

At each step select from the remaining items the one with the highest profit (provided the capacity of the knapsack is not exceeded). This approach tries to maximize the profit by choosing the most...
profitable items first.

**Greedy by Weight**

At each step select from the remaining items the one with the least weight (provided the capacity of the knapsack is not exceeded). This approach tries to maximize the profit by putting as many items into the knapsack as possible.

**Greedy by Profit Density**

At each step select from the remaining items the one with the largest profit density, $\frac{p_i}{w_i}$ (provided the capacity of the knapsack is not exceeded). This approach tries to maximize the profit by choosing items with the largest profit per unit of weight.

While all three approaches generate feasible solutions, we cannot guarantee that any of them will always generate the optimal solution. In fact, it is even possible that none of them does! Table gives an example where this is the case.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$w_i$</th>
<th>$p_i$</th>
<th>$\frac{p_i}{w_i}$</th>
<th>profit</th>
<th>weight</th>
<th>density</th>
<th>optimal solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>40</td>
<td>0.4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>35</td>
<td>0.7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>45</td>
<td>18</td>
<td>0.4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>4</td>
<td>0.2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>10</td>
<td>1.0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>2</td>
<td>0.4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>total weight</td>
<td>100</td>
<td>80</td>
<td>85</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>total profit</td>
<td>40</td>
<td>34</td>
<td>51</td>
<td>55</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table: 0/1 knapsack problem example (C=100).

The bottom line about greedy algorithms is this: Before using a greedy algorithm you must make sure that it always gives the correct answer. Fortunately, in many cases this is true.
Backtracking Algorithms

In this section we consider backtracking algorithms. As in the preceding section, we view the problem to be solved as a sequence of decisions. A backtracking algorithm systematically considers all possible outcomes for each decision. In this sense, backtracking algorithms are like the brute-force algorithms discussed in the preceding section. However, backtracking algorithms are distinguished by the way in which the space of possible solutions is explored. Sometimes a backtracking algorithm can detect that an exhaustive search is unnecessary and, therefore, it can perform much better.
Consider the set of scales shown in Figure 1. Suppose we are given a collection of \( n \) weights, \( \{w_1, w_2, \ldots, w_n\} \), and we are required to place all of the weights onto the scales so that they are balanced.

The problem can be expressed mathematically as follows: Let \( x_i \) represent the pan in which weight \( w_i \) is placed such that

\[
0 \quad \text{if } w_i \text{ is placed in the left pan}, \\
1 \quad \text{if } w_i \text{ is placed in the right pan}.
\]

The scales are balanced when the sum of the weights in the left pan equals the sum of the weights in the right pan,

\[
\sum_{i=1}^{n} w_i x_i = \sum_{i=1}^{n} w_i (1 - x_i).
\]

Given an arbitrary set of \( n \) weights, there is no guarantee that a solution to the problem exists. A solution always exists if, instead of balancing the scales, the goal is to minimize the difference between between
the total weights in the left and right pans. Thus, given \( \{w_1, w_2, \ldots, w_n\} \), our objective is to minimize \( \delta \) where

\[
\delta = \left| \sum_{i=1}^{n} w_i x_i - \sum_{i=1}^{n} w_i (1 - x_i) \right|
\]

subject to the constraint that all the weights are placed on the scales.

Given a set of scales and collection of weights, we might solve the problem by trial-and-error: Place all the weights onto the pans one-by-one. If the scales balance, a solution has been found. If not, remove some number of the weights and place them back on the scales in some other combination. In effect, we search for a solution to the problem by first trying one solution and then backing-up to try another.

Figure shows the solution space for the scales balancing problem. In this case the solution space takes the form of a tree: Each node of the tree represents a partial solution to the problem. At the root (node A) no weights have been placed yet and the scales are balanced. Let \( \delta \) be the difference between the sum of the weights currently placed in the left and right pans. Therefore, \( \delta = 0 \) at node A.

Node B represents the situation in which weight \( w_1 \) has been placed in the left pan. The difference between the pans is \( \delta = -w_1 \). Conversely, node C represents the situation in which the weight \( w_1 \) has been placed in the right pan. In this case \( \delta = +w_1 \). The complete solution tree has depth \( n \) and \( 2^n \) leaves. Clearly, the solution is the leaf node having the smallest \( |\delta| \) value.

Figure: Solution space for the scales balancing problem.
In this case (as in many others) the solution space is a tree. In order to find the best solution a backtracking algorithm visits all the nodes in the solution space. That is, it does a tree traversal. Section presents the two most important tree traversals—depth-first and breadth-first. Both kinds can be used to implement a backtracking algorithm.
Representing the Solution Space

This section presents an interface for the nodes of a solution space. By using an interface, we hide the details of the specific problem to be solved from the backtracking algorithm. In so doing, it is possible to implement completely generic backtracking problem solvers.

Although a backtracking algorithm behaves as if it is traversing a solution tree, it is important to realize that it is not necessary to have the entire solution tree constructed at once. Instead, the backtracking algorithm creates and destroys the nodes dynamically as it explores the solution space.

Program defines the Solution interface. Each instance of a class that implements the Solution interface represents a single node in the solution space.

```csharp
public interface Solution
{
    bool IsFeasible { get; }
    bool IsComplete { get; }
    int Objective { get; }
    int Bound { get; }
    IEnumerable_Successors { get; }
}
```

Program: Solution interface.

The Solution interface comprises the following properties:

**IsFeasible**
- This get accessor returns true if the solution instance is a feasible solution to the given problem. A solution is feasible if it satisfies the problem constraints.

**IsComplete**
- This get accessor returns true if the solution instance represents a complete solution. A solution is complete when all possible decisions have been made.

**Objective**
- This get accessor returns the value of the objective function for the given solution instance.

**Bound**
Representing the Solution Space

This `get` accessor returns a value that is a lower bound (if it exists) on the objective function for the given solution instance as well as all the solutions that can possibly be derived from that instance. This is a hook provided to facilitate the implementation of *branch-and-bound* backtracking which is described in Section □.

**Successors**

This `get` accessor returns an `IEnumerable` object that represents all of the successors (i.e., the children) of the given solution instance. It is assumed that the children of the given node are created *dynamically*.
Abstract Backtracking Solvers

The usual way to implement a backtracking algorithm is to write a method which traverses the solution space. This section presents an alternate, object-oriented approach that is based on the notion of an abstract solver.

Think of a solver as an abstract machine, the sole purpose of which is to search a given solution space for the best possible solution. A machine is an object. Therefore, it makes sense that we represent it as an instance of some class.

Program defines the Solver interface. The Solver interface consists of the single method Solve. This method takes as its argument a Solution that is the node in the solution space from which to begin the search. The Solve method returns the to the best solution found.

```csharp
public interface Solver
{
    Solution Solve(Solution initial);
}
```

Program: Solver interface.

- Abstract Solvers
- Depth-First Solver
- Breadth-First Solver
Abstract Solvers

Program defines the AbstractSolver class. The AbstractSolver class implements the Solver interface defined in Program. The AbstractSolver class contains two fields, bestSolution and bestObjective, two concrete methods, UpdateBest and Solve and the abstract method Search. Since Search is an abstract method, its implementation must be given in a derived class.

```csharp
public abstract class AbstractSolver : Solver
{
    protected Solution bestSolution;
    protected int bestObjective;

    protected abstract void Search(Solution initial);

    public virtual Solution Solve(Solution initial)
    {
        bestSolution = null;
        bestObjective = int.MaxValue;
        Search(initial);
        return bestSolution;
    }

    public virtual void UpdateBest(Solution solution)
    {
        if (solution.IsComplete && solution.IsFeasible
            && solution.Objective < bestObjective)
        {
            bestSolution = solution;
            bestObjective = solution.Objective;
        }
    }
}
```
**Program:** AbstractSolver class.

The *Solve* method does not search the solution space itself—it merely sets things up for the *Search* method. It is the *Search* method, which is provided by a derived class, that does the actual searching. When *Search* returns it is expected that the *bestSolution* field will refer to the best solution and that *bestObjective* will be the value of the objective function for the best solution.

The *UpdateBest* method is meant to be called by the *Search* method as it explores the solution space. As each complete solution is encountered, the *UpdateBest* method is called to keep track of the solution which *minimizes* the objective function.
Depth-First Solver

This section presents a backtracking solver that finds the best solution to a given problem by performing depth-first traversal of the solution space. Program defines the DepthFirstSolver class. The DepthFirstSolver class extends the AbstractSolver class defined in Program. It provides an implementation for the Search method.

```csharp
public class DepthFirstSolver : AbstractSolver
{
    protected override void Search(Solution current)
    {
        if (current.IsComplete)
            UpdateBest(current);
        else
        {
            foreach (Solution successor in current.Successors)
            {
                Search(successor);
            }
        }
    }
}
```

Program: DepthFirstSolver class.

The Search method does a complete, depth-first traversal of the solution space. Note that the implementation does not depend upon the characteristics of the problem being solved. In this sense the solver is a generic, abstract solver and can be used to solve any problem that has a tree-structured solution space!

Since the Search method in Program visits all the nodes in the solution space, it is essentially a brute-force algorithm. And because the recursive method backs up and then tries different alternatives, it is called a backtracking algorithm.
Breadth-First Solver

If we can find the optimal solution by doing a depth-first traversal of the solution space, then we can find the solution with a breadth-first traversal too. As defined in Section, a breadth-first traversal of a tree visits the nodes in the order of their depth in the tree. That is, first the root is visited, then the children of the root are visited, then the grandchildren are visited, and so on.

The **BreadthFirstSolver** class is defined in Program. The **BreadthFirstSolver** class extends the **AbstractSolver** class defined in Program. It simply provides an implementation for the **Search** method.

```csharp
public class BreadthFirstSolver : AbstractSolver
{
    protected override void Search(Solution initial)
    {
        Queue queue = new QueueAsLinkedList();
        queue.Enqueue(initial);
        while (!queue.IsEmpty)
        {
            Solution current = (Solution)queue.Dequeue();
            if (current.IsComplete)
                UpdateBest(current);
            else
                {
                    foreach (Solution soln in current.Successors)
                    {
                        queue.Enqueue(soln);
                    }
                }
        }
    }
}
```

**Program:** BreadthFirstSolver class.
The Search method implements a non-recursive, breadth-first traversal algorithm that uses a queue to keep track of nodes to be visited. The initial solution is enqueued first. Then the following steps are repeated until the queue is empty:

1. Dequeue the first solution in the queue.
2. If the solution is complete, call the UpdateBest method to keep track of the solution which minimizes the objective function.
3. Otherwise the solution is not complete. Enqueue all its successors.

Clearly, this algorithm does a complete traversal of the solution space.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Branch-and-Bound Solvers

The depth-first and breadth-first backtracking algorithms described in the preceding sections both naively traverse the entire solution space. However, sometimes we can determine that a given node in the solution space does not lead to the optimal solution--either because the given solution and all its successors are infeasible or because we have already found a solution that is guaranteed to be better than any successor of the given solution. In such cases, the given node and its successors need not be considered. In effect, we can prune the solution tree, thereby reducing the number of solutions to be considered.

For example, consider the scales balancing problem described in Section □. Consider a partial solution $P_k$ in which we have placed $k$ weights onto the pans ($0 \leq k < n$) and, therefore, $n-k$ weights remain to be placed. The difference between the weights of the left and right pans is given by

$$
\delta = \sum_{i=1}^{k} w_i x_i - \sum_{i=1}^{k} w_i (1 - x_i),
$$

and the sum of the weights still to be placed is

$$
r = \sum_{i=k+1}^{n} w_i.
$$

Suppose that $|\delta| > r$. That is, the total weight remaining is less than the difference between the weights in the two pans. Then, the best possible solution that we can obtain without changing the positions of the weights that have already been placed is $\hat{\delta} = |\delta| - r$. The quantity $\hat{\delta}$ is a lower bound on the value of the objective function for all the solutions in the solution tree below the given partial solution $P_k$.

In general, during the traversal of the solution space we may have already found a complete, feasible solution for which the objective function is less than $\hat{\delta}$. In that case, there is no point in considering any
of the solutions below $P_k$. That is, we can prune the subtree rooted at node $P_k$ from the solution tree. A backtracking algorithm that prunes the search space in this manner is called a \textit{branch-and-bound} algorithm.
Depth-First, Branch-and-Bound Solver

Only a relatively minor modification of the simple, depth-first solver shown in Program □ is needed to transform it into a branch-and-bound solver. Program □ defines the DepthFirstBranchAndBoundSolver class.

```csharp
public class DepthFirstBranchAndBoundSolver : AbstractSolver
{
    protected override void Search(Solution current)
    {
        if (current.IsComplete)
            UpdateBest(current);
        else
        {
            foreach (Solution successor in current.Successors)
            {
                if (successor.IsFeasible &&
                    successor.Bound < bestObjective)
                    Search(successor);
            }
        }
    }
}
```

**Program:** DepthFirstBranchAndBoundSolver class.

The only difference between the simple, depth-first solver and the branch-and-bound version is the if statement on lines 11-12. As each node in the solution space is visited two tests are done: First, the IsFeasible accessor is called to check whether the given node represents a feasible solution. Next, the Bound accessor is called to determine the lower bound on the best possible solution in the given subtree. The second test determines whether this bound is less than the value of the objective function of the best solution already found. The recursive call to explore the subtree is only made if both tests succeed. Otherwise, the subtree of the solution space is pruned.

The degree to which the solution space may be pruned depends strongly on the nature of the problem
Depth-First, Branch-and-Bound Solver

being solved. In the worst case, no subtrees are pruned and the branch-and-bound method visits all the nodes in the solution space. The branch-and-bound technique is really just a heuristic --sometimes it works and sometimes it does not.

It is important to understand the trade-off being made: The solution space is being pruned at the added expense of performing the tests as each node is visited. The technique is successful only if the savings which accrue from pruning exceed the additional execution time arising from the tests.
Consider again the 0/1 knapsack problem described in Section 0. We are given a set of $n$ items from which we are to select some number of items to be carried in a knapsack. The solution to the problem has the form \( \{x_1, x_2, \ldots, x_n\} \), where \( x_i \) is one if the \( i^{th} \) item is placed in the knapsack and zero otherwise. Each item has both a weight, \( w_i \), and a profit, \( p_i \). The goal is to maximize the total profit,

\[
\sum_{i=1}^{n} p_i x_i,
\]

subject to the knapsack capacity constraint

\[
\sum_{i=1}^{n} w_i x_i \leq c.
\]

A partial solution to the problem is one in which only the first \( k \) items have been considered. That is, the solution has the form \( S_k = \{x_1, x_2, \ldots, x_k\} \), where \( 1 \leq k < n \). The partial solution \( S_k \) is feasible if and only if

\[
\sum_{i=1}^{k} w_i x_i \leq c. \quad (14.1)
\]

Clearly if \( S_k \) is infeasible, then every possible complete solution containing \( S_k \) is also infeasible.

If \( S_k \) is feasible, the total profit of any solution containing \( S_k \) is bounded by

\[
\sum_{i=1}^{k} p_i x_i + \sum_{i=k+1}^{n} p_i. \quad (14.2)
\]

That is, the bound is equal the actual profit accrued from the \( k \) items already considered plus the sum of the profits of the remaining items.
Clearly, the 0/1 knapsack problem can be solved using a backtracking algorithm. Furthermore, by using Equations □ and □ a branch-and-bound solver can potentially prune the solution space, thereby arriving at the solution more quickly.

For example, consider the 0/1 knapsack problem with \(n=6\) items given in Table □. There are \(2^n = 64\) possible solutions and the solution space contains \(2^{n+1} - 1 = 127\) nodes. The simple DepthFirstSolver given in Program □ visits all 127 nodes and generates all 64 solutions because it does a complete traversal of the solution tree. The BreadthFirstSolver of Program □ behaves similarly. On the other hand, the DepthFirstBranchAndBoundSolver shown in Program □ visits only 67 nodes and generates only 27 complete solutions. In this case, the branch-and-bound technique prunes almost half the nodes from the solution space!
Top-Down Algorithms: Divide-and-Conquer

In this section we discuss a top-down algorithmic paradigm called *divide and conquer*. To solve a given problem, it is subdivided into one or more subproblems each of which is similar to the given problem. Each of the subproblems is solved independently. Finally, the solutions to the subproblems are combined in order to obtain the solution to the original problem.

Divide-and-conquer algorithms are often implemented using recursion. However, not all recursive methods are divide-and-conquer algorithms. Generally, the subproblems solved by a divide-and-conquer algorithm are *non-overlapping*.

- Example-Binary Search
- Example-Computing Fibonacci Numbers
- Example-Merge Sorting
- Running Time of Divide-and-Conquer Algorithms
- Example-Matrix Multiplication

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Example-Binary Search

Consider the problem of finding the position of an item in a sorted list. That is, given the sorted sequence $S = \{a_1, a_2, \ldots, a_n\}$ and an item $x$, find $i$ (if it exists) such that $a_i = x$. The usual solution to this problem is binary search.

Binary search is a divide-and-conquer strategy. The sequence $S$ is split into two subsequences, $S_L = \{a_1, a_2, \ldots, a_{n/2}\}$ and $S_R = \{a_{n/2}+1, a_{n/2}+1 \ldots, a_n\}$. The original problem is split into two subproblems: Find $x$ in $S_L$ or $S_R$. Of course, since the original list is sorted, we can quickly determine the list in which $x$ must appear. Therefore, we only need to solve one subproblem.

Program defines the method BinarySearch which takes four arguments, array, $x$, $i$ and $n$. This method looks for the position in array at which item $x$ is found. Specifically, it considers the following elements of the array:

$$array[i], array[i + 1], array[i + 2], \ldots, array[i + n - 1].$$
public class Example {

    public static int BinarySearch(ComparableObject[] array, ComparableObject target, int i, int n) {
        if (n == 0)
            throw new ArgumentException("empty array");
        if (n == 1)
            {
            if (array[i] == target)
                return i;
            throw new ArgumentException("target not found");
            }
        else
            {
            int j = i + n / 2;
            if (array[j] <= target)
                return BinarySearch(array, target, j, n - n/2);
            else
                return BinarySearch(array, target, i, n/2);
            }
    }

Program: Divide-and-conquer example--binary search.

The running time of the algorithm is clearly a function of \( n \), the number of elements to be searched. Although Program  works correctly for arbitrary values of \( n \), it is much easier to determine the running time if we assume that \( n \) is a power of two. In this case, the running time is given by the recurrence

\[
T(n) = \begin{cases} 
O(1) & n \leq 1, \\
T(n/2) + O(1) & n > 1. 
\end{cases}
\]  

Equation  is easily solved using repeated substitution:
\[
T(n) = T(n/2) + 1 \\
= T(n/4) + 2 \\
= T(n/8) + 3 \\
\vdots \\
= T(n/2^k) + k
\]

Setting \( n/2^k = 1 \) gives \( T(n) = \log n + 1 = O(\log n) \).
Example-Computing Fibonacci Numbers

The Fibonacci numbers are given by following recurrence

\[
F_n = \begin{cases} 
0 & n = 0, \\
1 & n = 1, \\
F_{n-1} + F_{n-2} & n \geq 2.
\end{cases} 
\] (14.4)

Section presents a recursive method to compute the Fibonacci numbers by implementing directly Equation. (See Program). The running time of that program is shown to be \( T(n) = \Omega((3/2)^n) \).

In this section we present a divide-and-conquer style of algorithm for computing Fibonacci numbers. We make use of the following identities

\[
F_{2k-1} = (F_k)^2 + (F_{k-1})^2 \\
F_{2k} = (F_k)^2 + 2F_kF_{k-1}
\]

for \( k \geq 1 \). (See Exercise). Thus, we can rewrite Equation as

\[
F_n = \begin{cases} 
0 & n = 0, \\
1 & n = 1, \\
(F_{[n/2]})^2 + (F_{[n/2]-1})^2 & n \geq 2 and n is odd, \\
(F_{[n/2]})^2 + 2F_{[n/2]}F_{[n/2]-1} & n \geq 2 and n is even.
\end{cases} 
\] (14.5)

Program defines the method Fibonacci which implements directly Equation. Given \( n \geq 1 \) it computes \( F_n \) by calling itself recursively to compute \( F_{[n/2]} \) and \( F_{[n/2]-1} \) and then combines the two results as required.
Example-Computing Fibonacci Numbers

```java
public class Example {
  public static int Fibonacci(int n) {
    if (n == 0 || n == 1)
      return n;
    else {
      int a = Fibonacci((n + 1) / 2);
      int b = Fibonacci((n + 1) / 2 - 1);
      if (n % 2 == 0)
        return a * (a + 2 * b);
      else
        return a * a + b * b;
    }
  }
}
```

**Program**: Divide-and-conquer Example--computing Fibonacci numbers.

To determine a bound on the running time of the `Fibonacci` method in Program □ we assume that $T(n)$ is a non-decreasing function. That is, $T(n) \geq T(n - 1)$ for all $n \geq 1$. Therefore $T(\lfloor n/2 \rfloor) \geq T(\lfloor n/2 \rfloor - 1)$. Although the program works correctly for all values of $n$, it is convenient to assume that $n$ is a power of 2. In this case, the running time of the method is upper-bounded by $T(n)$ where

$$T(n) = \begin{cases} O(1) & n \leq 1, \\ 2T(n/2) + O(1) & n > 1. \end{cases} \quad (14.6)$$

Equation □ is easily solved using repeated substitution:
\[ T(n) = 2T(n/2) + 1 \\
= 4T(n/4) + 1 + 2 \\
= 8T(n/8) + 1 + 2 + 4 \\
\vdots \\
= 2^k T(n/2^k) + \sum_{i=0}^{k-1} 2^i \\
\vdots \\
= nT(1) + n - 1 \quad (n = 2^k). \]

Thus, \( T(n) = 2n - 1 = O(n) \).
Example-Merge Sorting

Sorting algorithms and sorters are covered in detail in Chapter □. In this section we consider a divide-and-conquer sorting algorithm—merge sort. Given an array of \( n \) items in arbitrary order, the objective is to rearrange the elements of the array so that they are ordered from the smallest element to the largest one.

The merge sort algorithm sorts a sequence of length \( n > 1 \) by splitting it into two subsequences—one of length \( \lfloor n/2 \rfloor \), the other of length \( \lceil n/2 \rceil \). Each subsequence is sorted and then the two sorted sequences are merged into one.

Program □ defines the method MergeSort which takes three arguments, array, \( i \), and \( n \). The method sorts the following \( n \) elements:

\[
array[i], \, array[i + 1], \, array[i + 2], \ldots, \, array[i + n - 1].
\]

The MergeSort method calls itself as well as the Merge method. The purpose of the Merge method is to merge two sorted sequences, one of length \( \lfloor n/2 \rfloor \), the other of length \( \lceil n/2 \rceil \), into a single sorted sequence of length \( n \). This can easily be done in \( O(n) \) time. (See Program □).
Example-Merge Sorting

```java
public class Example {
    public static void MergeSort(ComparableObject[] array, int i, int n) {
        if (n > 1) {
            MergeSort(array, i, n / 2);
            MergeSort(array, i + n / 2, n - n / 2);
            Merge(array, i, n / 2, n - n / 2);
        }
    }
}
```

**Program:** Divide-and-conquer example--merge sorting.

The running time of the `MergeSort` method depends on the number of items to be sorted, \( n \). Although `Program` works correctly for arbitrary values of \( n \), it is much easier to determine the running time if we assume that \( n \) is a power of two. In this case, the running time is given by the recurrence

\[
T(n) = \begin{cases} 
O(1) & \text{if } n \leq 1, \\
2T(n/2) + O(n) & \text{if } n > 1.
\end{cases}
\]  
(14.7)

Equation is easily solved using repeated substitution:

\[
T(n) = 2T(n/2) + n \\
= 4T(n/4) + 2n \\
= 8T(n/8) + 3n \\
\vdots \\
= 2^kT(n/2^k) + kn
\]

Setting \( n/2^k = 1 \) gives \( T(n) = n + n \log n = O(n \log n) \).
Running Time of Divide-and-Conquer Algorithms

A number of divide-and-conquer algorithms are presented in the preceding sections. Because these algorithms have a similar form, the recurrences which give the running times of the algorithms are also similar in form. Table summarizes the running times of Programs , , and .

<table>
<thead>
<tr>
<th>program</th>
<th>recurrence</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>( T(n) = T(n/2) + O(1) )</td>
<td>( \Theta(\log n) )</td>
</tr>
<tr>
<td>Program</td>
<td>( T(n) = 2T(n/2) + O(1) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Program</td>
<td>( T(n) = 2T(n/2) + O(n) )</td>
<td>( \Theta(n \log n) )</td>
</tr>
</tbody>
</table>

Table: Running times of divide-and-conquer algorithms.

In this section we develop a general recurrence that characterizes the running times of many divide-and-conquer algorithms. Consider the form of a divide-and-conquer algorithm to solve a given problem. Let \( n \) be a measure of the size of the problem. Since the divide-and-conquer paradigm is essentially recursive, there must be a base case. That is, there must be some value of \( n \), say \( n_0 \), for which the solution to the problem is computed directly. We assume that the worst-case running time for the base case is bounded by a constant.

To solve an arbitrarily large problem using divide-and-conquer, the problem is divided into a number of smaller problems, each of which is solved independently. Let \( a \) be the number of smaller problems to be solved (\( a \in \mathbb{Z}, a \geq 1 \)). The size of each of these problems is some fraction of the original problem, typically either \( \lceil n/b \rceil \) or \( \lfloor n/b \rfloor \) (\( b \in \mathbb{Z}, b \geq 1 \)).

The solution to the original problem is constructed from the solutions to the smaller problems. The running time required to do this depends on the problem to be solved. In this section we consider
polynomial running times. That is, $O(n^k)$ for some integer $k \geq 0$.

For the assumptions stated above, the running time of a divide-and-conquer algorithm is given by

$$T(n) = \begin{cases} 
O(1) & n \leq n_0, \\
\alpha T([n/b]) + O(n^k) & n > n_0.
\end{cases} \quad (14.8)$$

In order to make it easier to find the solution to Equation (14.8), we drop the $O(\cdot)$'s as well as the $\lceil \cdot \rceil$ from the recurrence. We can also assume (without loss of generality) that $n_0 = 1$. As a result, the recurrence becomes

$$T(n) = \begin{cases} 
1 & n = 1, \\
\alpha T(n/b) + n^k & n > 1.
\end{cases}$$

Finally, we assume that $n$ is a power of $b$. That is, $n = b^m$ for some integer $m \geq 0$. Consequently, the recurrence formula becomes

$$T(b^m) = \begin{cases} 
1 & m = 0, \\
T(b^{m-1}) + b^{mk} & m > 0.
\end{cases} \quad (14.9)$$

We solve Equation (14.9) as follows. Divide both sizes of the recurrence by $a^m$ and then telescope:

$$\frac{T(b^m)}{a^m} = \frac{T(b^{m-1})}{a^{m-1}} + \left(\frac{b^k}{a}\right)^m \quad (14.10)$$

$$\frac{T(b^{m-1})}{a^{m-1}} = \frac{T(b^{m-2})}{a^{m-2}} + \left(\frac{b^k}{a}\right)^{m-1}$$

$$\frac{T(b^{m-2})}{a^{m-2}} = \frac{T(b^{m-3})}{a^{m-3}} + \left(\frac{b^k}{a}\right)^{m-2}$$

$$\vdots$$

$$\frac{T(b)}{a} = T(1) + \left(\frac{b^k}{a}\right) \quad (14.11)$$
Adding Equation \( T(1) = 1 \) through Equation \( T(n) \), substituting \( T(1) = 1 \) and multiplying both sides by \( a^m \) gives

\[
T(n) = a^m \sum_{i=0}^{m} \left( \frac{b^k}{a} \right)^i.
\]  \hspace{1cm} (14.12)

In order to evaluate the summation in Equation \( T(n) \) we must consider three cases:

- **Case 1** (\( a > b^k \))
- **Case 2** (\( a = b^k \))
- **Case 3** (\( a < b^k \))
- **Summary**
Case 1 \((a > b^k)\)

In this case, the term \(b^k/a\) falls between zero and one. Consider the *infinite* geometric series summation:

\[
\sum_{i=0}^{\infty} \left( \frac{b^k}{a} \right)^i = \frac{a}{a-b^k} = C
\]

Since the infinite series summation approaches a finite constant \(C\) and since each term in the series is positive, the *finite* series summation in Equation \(\square\) is bounded from above by \(C\):

\[
\sum_{i=0}^{m} \left( \frac{b^k}{a} \right)^i \leq C
\]

Substituting this result into Equation \(\square\) and making use of the fact that \(n = b^m\), and therefore \(m = \log_b n\), gives

\[
T(n) \leq C a^m = O(a^m) = O(a^{\log_b n}) = O(a^{\log_b n \log_b a}) = O(n^{\log_b a}).
\]
Case 2 \((a = b^k)\)

In this case the term \(b^k / a\) is exactly one. Therefore, the series summation in Equation ■ is simply

\[
\sum_{i=0}^{m} \left( \frac{b^k}{a} \right)^i = m + 1.
\]

Substituting this result into Equation ■ and making use of the fact that \(n = b^m\) and \(a = b^k\) gives

\[
T(n) = (m + 1)a^m
\]

\[
= O(ma^m)
\]

\[
= O(m(b^k)^m)
\]

\[
= O((b^m)^k m)
\]

\[
= O(n^k \log_b n).
\]
Case 3 \( (a < b^k) \)

In this case the term \( \frac{b^k}{a} \) is greater than one and we make use of the general formula for a finite geometric series summation (see Section □) to evaluate the summation:

\[
\sum_{i=0}^{m} \left( \frac{b^k}{a} \right)^i = \frac{(\frac{b^k}{a})^{m+1} - 1}{b^k/a - 1}.
\]

Substituting this result in Equation □ and simplifying gives:

\[
T(n) = a^m \left( \frac{(\frac{b^k}{a})^{m+1} - 1}{b^k/a - 1} \right) \\
= a^m \left( \frac{(\frac{b^k}{a})^m - a/b^k}{1 - a/b^k} \right) \\
= O(a^m (\frac{b^k}{a})^m) \\
= O(b^{km}) \\
= O(n^k)
\]
Summary

For many divide-and-conquer algorithms the running time is given by the general recurrence shown in Equation 14.13. Solutions to the recurrence depend on the relative values of the constants $a$, $b$, and $k$. Specifically, the solutions satisfy the following bounds:

$$T(n) = \begin{cases} O(n \log_b n) & a > b^k, \\ O(n^k \log_b n) & a = b^k, \\ O(n^k) & a < b^k. \end{cases}$$

Table shows how to apply Equation 14.13 to find the running times of the divide-and-conquer algorithms described in the preceding sections. Comparing the solutions in Table with those given in shows the results obtained using the general formula agree with the analyses done in the preceding sections.

<table>
<thead>
<tr>
<th>program</th>
<th>recurrence</th>
<th>$a$</th>
<th>$b$</th>
<th>$k$</th>
<th>case</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>$T(n)=T(n/2)+O(1)$</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>$a = b^k$</td>
<td>$O(n^0 \log_2 n)$</td>
</tr>
<tr>
<td>Program</td>
<td>$T(n)=2T(n/2)+O(1)$</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>$a &gt; b^k$</td>
<td>$O(n^{\log_2 2})$</td>
</tr>
<tr>
<td>Program</td>
<td>$T(n)=2T(n/2)+O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$a = b^k$</td>
<td>$O(n^{1 \log_2 n})$</td>
</tr>
</tbody>
</table>

Table: Computing running times using Equation 14.13.
Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Example-Matrix Multiplication

Consider the problem of computing the product of two matrices. That is, given two $n \times n$ matrices, $A$ and $B$, compute the $n \times n$ matrix $C = A \times B$, the elements of which are given by

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}. \quad (14.14)$$

Section shows that the direct implementation of Equation results in an $O(n^3)$ running time. In this section we show that the use of a divide-and-conquer strategy results in a slightly better asymptotic running time.

To implement a divide-and-conquer algorithm we must break the given problem into several subproblems that are similar to the original one. In this instance we view each of the $n \times n$ matrices as a $2 \times 2$ matrix, the elements of which are $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$ submatrices. Thus, the original matrix multiplication, $C = A \times B$ can be written as

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix},$$

where each $A_{i,j}$, $B_{i,j}$, and $C_{i,j}$ is an $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$ matrix.

From Equation we get that the result submatrices can be computed as follows:

$$\begin{align*}
C_{1,1} &= A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1} \\
C_{1,2} &= A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2} \\
C_{2,1} &= A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1} \\
C_{2,2} &= A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2}.
\end{align*}$$
Here the symbols + and \( \times \) are taken to mean addition and multiplication (respectively) of \( \left( \frac{n}{2} \right) \times \left( \frac{n}{2} \right) \) matrices.

In order to compute the original \( n \times n \) matrix multiplication we must compute eight \( \left( \frac{n}{2} \right) \times \left( \frac{n}{2} \right) \) matrix products (divide) followed by four \( \left( \frac{n}{2} \right) \times \left( \frac{n}{2} \right) \) matrix sums (conquer). Since matrix addition is an \( O(n^2) \) operation, the total running time for the multiplication operation is given by the recurrence:

\[
T(n) = \begin{cases} 
O(1) & n = 1, \\
8T(n/2) + O(n^2) & n > 1.
\end{cases}
\]  

(14.15)

Note that Equation (14.15) is an instance of the general recurrence given in Equation (14.14). In this case, \( a = 8, b = 2, \) and \( k = 2. \) We can obtain the solution directly from Equation (14.14). Since \( a > b^k, \) the total running time is \( O(n^\log_2 a) = O(n^\log_2 8) = O(n^3). \) But this no better than the original, direct algorithm!

Fortunately, it turns out that one of the eight matrix multiplications is redundant. Consider the following series of seven \( \left( \frac{n}{2} \right) \times \left( \frac{n}{2} \right) \) matrices:

\[
\begin{align*}
M_0 &= (A_{1,1} + A_{2,2}) \times (B_{1,1} + B_{2,2}) \\
M_1 &= (A_{1,2} - A_{2,2}) \times (B_{2,1} + B_{2,2}) \\
M_2 &= (A_{1,1} - A_{2,1}) \times (B_{1,1} + B_{1,2}) \\
M_3 &= (A_{1,1} + A_{1,2}) \times B_{2,2} \\
M_4 &= A_{1,1} \times (B_{1,2} - B_{2,2}) \\
M_5 &= A_{2,2} \times (B_{2,1} - B_{1,1}) \\
M_6 &= (A_{2,1} + A_{2,2}) \times B_{1,1}
\end{align*}
\]

Each equation above has only one multiplication. Ten additions and seven multiplications are required to compute \( M_0 \) through \( M_6. \) Given \( M_0 \) through \( M_6, \) we can compute the elements of the product matrix \( C \) as follows:

\[
\begin{align*}
C_{1,1} &= M_0 + M_1 - M_3 + M_5 \\
C_{1,2} &= M_3 + M_4 \\
C_{2,1} &= M_5 + M_6 \\
C_{2,2} &= M_0 - M_2 + M_4 - M_6
\end{align*}
\]
Altogether this approach requires seven $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$ matrix multiplications and 18 $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$ additions. Therefore, the worst-case running time is given by the following recurrence:

$$T(n) = \begin{cases} O(1) & n = 1, \\ 7T(n/2) + O(n^2) & n > 1. \end{cases} \tag{14.16}$$

As above, Equation 14.16 is an instance of the general recurrence given in Equation 14.15, and we obtain the solution directly from Equation 14.15. In this case, $a=7$, $b=2$, and $k=2$. Therefore, $a > b^k$ and the total running time is

$$O(n^{\log_b a}) = O(n^{\log_2 7}).$$

Note $\log_2 7 \approx 2.807355$. Consequently, the running time of the divide-and-conquer matrix multiplication strategy is $O(n^{2.8})$ which is better (asymptotically) than the straightforward $O(n^3)$ approach.
Bottom-Up Algorithms: Dynamic Programming

In this section we consider a bottom-up algorithmic paradigm called *dynamic programming*. In order to solve a given problem, a series of subproblems is solved. The series of subproblems is devised carefully in such a way that each subsequent solution is obtained by combining the solutions to one or more of the subproblems that have already been solved. All intermediate solutions are kept in a table in order to prevent unnecessary duplication of effort.

- Example-Generalized Fibonacci Numbers
- Example-Computing Binomial Coefficients
- Application: Typesetting Problem
Example-Generalized Fibonacci Numbers

Consider the problem of computing the generalized Fibonacci numbers. The generalized Fibonacci numbers of order \( k \geq 2 \) are given by

\[
F_n^{(k)} = \begin{cases} 
0 & \text{if } 0 \leq n < k - 1, \\
1 & \text{if } n = k - 1, \\
\sum_{i=1}^{k} F_{n-i}^{(k)} & \text{if } n \geq k.
\end{cases}
\]  

(14.17)

Notice that the "normal" Fibonacci numbers considered in Section are the same as the generalized Fibonacci numbers of order 2.

If we write a recursive method that implements directly Equation, we get an algorithm with exponential running time. For example, in Section it is shown that the time to compute the second-order Fibonacci numbers is \( T(n) = \Theta((3/2)^n) \).

The problem with the direct recursive implementation is that it does far more work than is needed because it solves the same subproblem many times. For example, to compute \( F_{10}^{(2)} \) it is necessary to compute both \( F_{9}^{(2)} \) and \( F_{8}^{(2)} \). However, in computing \( F_{9}^{(2)} \), it is also necessary to compute \( F_{8}^{(2)} \), and so on.

An alternative to the top-down recursive implementation is to do the calculation from the bottom up. In order to do this we compute the series of sequences

\[
S_0 = \{ F_0^{(k)} \} \\
S_1 = \{ F_0^{(k)}, F_1^{(k)} \} \\
\vdots \\
S_n = \{ F_0^{(k)}, F_1^{(k)}, \ldots, F_n^{(k)} \}.
\]
Notice that we can compute $S_{i+1}$ from the information contained in $S_i$ simply by using Equation □.

Program □ defines the method `Fibonacci` which takes two integer arguments $n$ and $k$ and computes the $n^{th}$ Fibonacci number of order $k$ using the approach described above. This algorithm uses an array to represent the series of sequences $S_0, S_1, \ldots, S_n$. As each subsequent Fibonacci number is computed it is added to the end of the array.

```java
public class Example {
    public static int Fibonacci(int n, int k) {
        if (n < k - 1)
            return 0;
        else if (n == k - 1)
            return 1;
        else {
            int[] f = new int[n + 1];
            for (int i = 0; i < k - 1; ++i)
                f[i] = 0;
            f[k - 1] = 1;
            for (int i = k; i <= n; ++i)
                {
                    int sum = 0;
                    for (int j = 1; j <= k; ++j)
                        sum += f[i - j];
                    f[i] = sum;
                }
            return f[n];
        }
    }
}

Program: Dynamic programming example--computing generalized Fibonacci numbers.

The worst-case running time of the `Fibonacci` method given in Program □ is a function of both $n$ and $k$:
\[ T(n, k) = \begin{cases} 
O(1) & 0 \leq n < k, \\
O(kn) & n \geq k. 
\end{cases} \]
Consider the problem of computing the binomial coefficient

\[
\binom{n}{m} = \frac{n!}{(n-m)!m!}
\]  \hspace{1cm} (14.18)

given non-negative integers \( n \) and \( m \) (see Theorem □).

The problem with implementing directly Equation □ is that the factorials grow quickly with increasing \( n \) and \( m \). For example, \( 13! = 6227020800 > 2^{31} \). Therefore, it is not possible to represent \( n! \) for \( n \geq 13 \) using 32-bit integers. Nevertheless it is possible to represent the binomial coefficients \( \binom{n}{m} \) up to \( n=33 \) without overflowing. For example, \( \binom{33}{16} = 1166803110 < 2^{31} \).

Consider the following recursive definition of the binomial coefficients:

\[
\binom{n}{m} = \begin{cases} 
1 & \text{if } m = 0, \\
1 & \text{if } n = m, \\
\binom{n-1}{m} + \binom{n-1}{m-1} & \text{otherwise.}
\end{cases}
\]  \hspace{1cm} (14.19)

This formulation does not require the computation of factorials. In fact, the only computation needed is addition.

If we implement Equation □ directly as a recursive method, we get a method whose running time is given by
Example: Computing Binomial Coefficients

\[
T(n, m) = \begin{cases} 
O(1) & m = 0, \\
O(1) & n = m, \\
T(n - 1, m) + T(n - 1, m - 1) + O(1) & \text{otherwise.}
\end{cases}
\]

which is very similar to Equation \[\text{□}\]. In fact, we can show that \(T(n, m) = \Omega\left(\binom{n}{m}\right)\) which (by Equation \[\text{□}\]) is not a very good running time at all! Again the problem with the direct recursive implementation is that it does far more work than is needed because it solves the same subproblem many times.

An alternative to the top-down recursive implementation is to do the calculation from the bottom up. In order to do this we compute the series of sequences

\[
\begin{align*}
S_0 &= \{ \binom{0}{0} \} \\
S_1 &= \{ \binom{1}{0}, \binom{1}{1} \} \\
S_2 &= \{ \binom{2}{0}, \binom{2}{1}, \binom{2}{2} \} \\
&\quad \vdots \\
S_n &= \{ \binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \ldots, \binom{n}{n} \}.
\end{align*}
\]

Notice that we can compute \(S_{i+1}\) from the information contained in \(S_i\) simply by using Equation \[\text{□}\]. Table \[\text{□}\] shows the sequence in tabular form--the \(i^{th}\) row of the table corresponds the sequence \(S_i\). This tabular representation of the binomial coefficients is known as Pascal's triangle \[\checkmark\].
Program defines the method Binom which takes two integer arguments \( n \) and \( m \) and computes the binomial coefficient \( \binom{n}{m} \) by computing Pascal's triangle. According to Equation, each subsequent row depends only on the preceding row—it is only necessary to keep track of one row of data. The implementation shown uses an array of length \( n \) to represent a row of Pascal's triangle. Consequently, instead of a table of size \( O(n^2) \), the algorithm gets by with \( O(n) \) space. The implementation has been coded carefully so that the computation can be done in place. That is, the elements of \( S_{i+1} \) are computed in reverse so that they can be written over the elements of \( S_i \) that are no longer needed.

```
public class Example {
    public static int Binom(int n, int m) {
        int[] b = new int[n + 1];
        b[0] = 1;
        for (int i = 1; i <= n; ++i) {
            b[i] = 1;
            for (int j = i - 1; j > 0; --j)
                b[j] += b[j - 1];
        }
        return b[m];
    }
}
```

Program: Dynamic programming example—computing Binomial coefficients.

The worst-case running time of the Binom method given in Program is clearly \( O(n^2) \).
Application: Typesetting Problem

Consider the problem of typesetting a paragraph of justified text. A paragraph can be viewed as a sequence of \( n > 0 \) words, \( \{w_1, w_2, \ldots, w_n\} \). The objective is to determine how to break the sequence into individual lines of text of the appropriate size. Each word is separated from the next by some amount of space. By stretching or compressing the space between the words, the left and right ends of consecutive lines of text are made to line up. A paragraph looks best when the amount of stretching or compressing is minimized.

We can formulate the problem as follows: Assume that we are given the lengths of the words, \( \{l_1, l_2, \ldots, l_n\} \), and that the desired length of a line is \( D \). Let \( W_{i,j} \) represent the sequence of words from \( w_i \) to \( w_j \) (inclusive). That is,

\[
W_{i,j} = \{w_i, w_{i+1}, \ldots, w_j\},
\]

for \( 1 \leq i \leq j \leq n \).

Let \( L_{i,j} \) be the sum of the lengths of the words in the sequence \( W_{i,j} \). That is,

\[
L_{i,j} = \sum_{k=i}^{j} l_k.
\]

The natural length, for the sequence \( W_{i,j} \) is the sum of the lengths of the words, \( L_{i,j} \) plus the normal amount of space between those words. Let \( s \) be the normal size of the space between two words. Then the natural length of \( W_{i,j} \) is \( L_{i,j} + (j - i)s \). Note, we can also define \( L_{i,j} \) recursively as follows:

\[
L_{i,j} = \begin{cases} 
  l_i & i = j, \\
  L_{i,j-1} + l_j & i < j.
\end{cases}
\]  

(14.20)
In general, when we typeset the sequence $W_{i,j}$ all on a single line, we need to stretch or compress the spaces between the words so that the length of the line is the desired length $D$. Therefore, the amount of stretching or compressing is given by the difference $D - (L_{i,j} + (j - i)s)$. However, if the sum of the lengths of the words, $L_{i,j}$, is longer than the desired line length $D$, it is not possible to typeset the sequence on a single line.

Let $P_{i,j}$ be the penalty associated with typesetting the sequence $L_{i,j}$ on a single line. Then,

$$P_{i,j} = \begin{cases} 
|D - L_{i,j} - (j - i)s| & D \geq L_{i,j}, \\
\infty & D < L_{i,j}.
\end{cases} \tag{14.21}$$

This definition is of penalty is consistent with the stated objectives: The penalty increases as the difference between the natural length of the sequence and the desired length increases and the infinite penalty disallows lines that are too long.

Finally, we define the quantity $C_{i,j}$ for $1 \leq i \leq j \leq n$ as the minimum total penalty required to typeset the sequence $W_{i,j}$. In this case, the text may be all on one line or it may be split over more than one line. The quantity $C_{i,j}$ is given by

$$C_{i,j} = \begin{cases} 
P_{i,j} & i = j, \\
\min \{P_{i,j}, \min_{i \leq k < j} (P_{i,k} + C_{k+1,j})\} & \text{otherwise.} \tag{14.22}
\end{cases}$$

We obtain Equation \Box as follows: When $i=j$ there is only one word in the paragraph. The minimum total penalty associated with typesetting the paragraph in this case is just the penalty which results from putting the one word on a single line.

In the general case, there is more than one word in the sequence $W_{i,j}$. In order to determine the optimal way in which to typeset the paragraph we consider the cost of putting the first $k$ words of the sequence on the first line of the paragraph, $P_{i,k}$, plus the minimum total cost associated with typesetting the rest of the paragraph $C_{k+1,j}$. The value of $k$ which minimizes the total cost also specifies where the line break should occur.
• Example
• Implementation

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Example

Suppose we are given a sequence of \( n=5 \) words, \( W = \{w_1, w_2, w_3, w_4, w_5\} \) having lengths \( \{10, 10, 10, 12, 50\} \), respectively, which are to be typeset in a paragraph of width \( D=60 \). Assume that the normal width of an inter-word space is \( s=10 \).

We begin by computing the lengths of all the subsequences of \( W \) using Equation 14.21. The lengths of all \( n(n-1)/2 \) subsequences of \( W \) are tabulated in Table 14.21.

<table>
<thead>
<tr>
<th>( L_{i,j} )</th>
<th>( j=1 )</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>20</td>
<td>30</td>
<td>42</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10</td>
<td>20</td>
<td>32</td>
<td>82</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>10</td>
<td>22</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>12</td>
<td>62</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table: Typesetting problem.

Given \( L_{i,j}, D, \) and \( s \), it is a simple matter to apply Equation 14.21 to obtain the one-line penalties, \( P_{i,j} \), which measure the amount of stretching or compressing needed to set all the words in a given subsequence on a single line. These are tabulated in Table 14.21.
Given the one-line penalties $P_{i,j}$, we can use Equation 1 to find for each subsequence of $W$ the minimum total penalty, $C_{i,j}$, associated with forming a paragraph from the words in that subsequence. These are tabulated in Table 1.

The $C_{1,5}$ entry in Table 1 gives the minimum total cost of typesetting the entire paragraph. The value 22 was obtained as follows:

$$C_{1,5} = \min \{ C_{1,1} + C_{2,5}, P_{1,2} + C_{3,5}, P_{1,3} + C_{4,5}, P_{1,4} + C_{5,5}, P_{1,5} \}$$

$$= P_{1,4} + C_{5,5}$$

$$= 12 + 10.$$  

This indicates that the optimal solution is to set words $w_1, w_2, w_3$, and $w_4$ on the first line of the paragraph and leave $w_5$ by itself on the last line of the paragraph. Figure 1 illustrates this result.
This formulation of the typesetting problem seems like overkill. Why not just typeset the lines of text one-by-one, minimizing the penalty for each line as we go? In other words why don't we just use a greedy strategy? Unfortunately, the obvious greedy solution strategy does not work!

For example, the greedy strategy begins by setting the first line of text. To do so it must decide how many words to put on that line. The obvious thing to do is to select the value of $k$ for which $P_{1,k}$ is the smallest. From Table we see that $P_{1,3} = 10$ has the smallest penalty. Therefore, the greedy approach puts three words on the first line as shown in Figure.

Since the remaining two words do not both fit on a single line, they are set on separate lines. The total of the penalties for the paragraph typeset using the greedy algorithm is $P_{1,3} + P_{4,4} + P_{5,5} = 68$. Clearly, the solution is not optimal (nor is it very pleasing esthetically).
Implementation

Program defines the method `Typeset` which takes three arguments. The first, `l`, is an array of `n` integers that gives the lengths of the words in the sequence to be typeset. The second, `D`, specifies the desired paragraph width and the third, `s`, specifies the normal inter-word space.

```csharp
public class Example
{
    public static void Typeset(int[] l, int D, int s)
    {
        int n = l.Length;
        int[,] L = new int[n, n];
        for (int i = 0; i < n; ++i)
        {
            L[i, i] = l[i];
            for (int j = i + 1; j < n; ++j)
            {
                L[i, j] = L[i, j - 1] + l[j];
            }
        }
        int[,] P = new int[n, n];
        for (int i = 0; i < n; ++i)
        {
            for (int j = i; j < n; ++j)
            {
                if (L[i, j] < D)
                {
                    P[i, j] = Math.Abs(D - L[i, j] - (j-i)*s);
                }
                else
                {
                    P[i, j] = int.MaxValue;
                }
            }
        }
        int[,] c = new int[n, n];
        for (int j = 0; j < n; ++j)
        {
            c[j, j] = P[j, j];
            for (int i = j - 1; i >= 0; --i)
            {
                int min = P[i, j];
                for (int k = i; k < i; ++k)
                {
                    if (P[i, k] + P[k, j] < min)
                    {
                        min = P[i, k] + P[k, j];
                    }
                }
                c[i, j] = min;
            }
        }
    }
}
```
Program: Dynamic programming example--typesetting a paragraph.

The method first computes the lengths, $L_{i,j}$, of all possible subsequences (lines 6-12). This is done by using the dynamic programming paradigm to evaluate the recursive definition of $L_{i,j}$ given in Equation 1. The running time for this computation is clearly $O(n^2)$.

The next step computes the one-line penalties $P_{i,j}$ as given by Equation 2 (lines 13-21). This calculation is a straightforward one and its running time is also $O(n^2)$.

Finally, the minimum total costs, $C_{i,j}$, of typesetting each subsequence are determined for all possible subsequences (lines 22-37). Again we make use of the dynamic programming paradigm to evaluate the recursive definition of $C_{i,j}$ given in Equation 3. The running time for this computation is $O(n^3)$. As a result, the overall running time required to determine the best way to typeset a paragraph of $n$ words is $O(n^3)$.
Randomized Algorithms

In this section we discuss algorithms that behave randomly. By this we mean that there is an element of randomness in the way that the algorithm solves a given problem. Of course, if an algorithm is to be of any use, it must find a solution to the problem at hand, so it cannot really be completely random.

Randomized algorithms are said to be methods of last resort. This is because they are used often when no other feasible solution technique is known. For example, randomized methods are used to solve problems for which no closed-form, analytic solution is known. They are also used to solve problems for which the solution space is so large that an exhaustive search is infeasible.

To implement a randomized algorithm we require a source of randomness. The usual source of randomness is a random number generator. Therefore, before presenting randomized algorithms, we first consider the problem of computing random numbers.

- Generating Random Numbers
- Random Variables
- Monte Carlo Methods
- Simulated Annealing
Generating Random Numbers

In this section we consider the problem of generating a sequence of random numbers on a computer. Specifically, we desire an infinite sequence of statistically independent random numbers uniformly distributed between zero and one. In practice, because the sequence is generated algorithmically using finite-precision arithmetic, it is neither infinite nor truly random. Instead, we say that an algorithm is "good enough" if the sequence it generates satisfies almost any statistical test of randomness. Such a sequence is said to be pseudorandom.

The most common algorithms for generating pseudorandom numbers are based on the linear congruential random number generator invented by Lehmer. Given a positive integer \( m \) called the modulus and an initial seed value \( X_0 (0 \leq X_0 < m) \), Lehmer's algorithm computes a sequence of integers between 0 and \( m-1 \). The elements of the sequence are given by

\[
X_{i+1} = (aX_i + c) \mod m,
\]

where \( a \) and \( c \) are carefully chosen integers such that \( 2 \leq a < m \) and \( 0 \leq c < m \).

For example, the parameters \( a=13, c=1, m=16 \), and \( X_0 = 0 \) produce the sequence

\[0, 1, 14, 7, 12, 13, 10, 3, 8, 9, 6, 15, 4, 5, 2, 11, 0, \ldots\]

The first \( m \) elements of this sequence are distinct and appear to have been drawn at random from the set \( \{0, 1, 2 \ldots, 15\} \). However since \( X_m = X_0 \) the sequence is cyclic with period \( m \).

Notice that the elements of the sequence alternate between odd and even integers. This follows directly from Equation and the fact that \( m=16 \) is a multiple of 2. Similar patterns arise when we consider the elements as binary numbers:

\[0000, 0001, 1110, 0111, 1100, 1101, 1010, 0011, 1000, \ldots\]

The least significant two bits are cyclic with period four and the least significant three bits are cycle with
Generating Random Numbers

period eight! (These patterns arise because \( m=16 \) is also a multiple of 4 and 8). The existence of such patterns make the sequence less random. This suggests that the best choice for the modulus \( m \) is a prime number.

Not all parameter values result in a period of \( m \). For example, changing the multiplier \( a \) to 11 produces the sequence

\[
0, 1, 12, 5, 8, 9, 4, 13, 0, \ldots
\]

the period of which is only \( m/2 \). In general because each subsequent element of the sequence is determined solely from its predecessor and because there are \( m \) possible values, the longest possible period is \( m \). Such a generator is called a full period generator.

In practice the increment \( c \) is often set to zero. In this case, Equation becomes

\[
X_{i+1} = aX_i \mod m. \tag{14.24}
\]

This is called a multiplicative linear congruential random number generator. (For \( c \neq 0 \) it is called a mixed linear congruential generator).

In order to prevent the sequence generated by Equation from collapsing to zero, the modulus \( m \) must be prime and \( X_0 \) cannot be zero. For example, the parameters \( a=6, m=13, \) and \( X_0 = 1 \) produce the sequence

\[
1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1, \ldots
\]

Notice that the first 12 elements of the sequence are distinct. Since a multiplicative congruential generator can never produce a zero, the maximum possible period is \( m-1 \). Therefore, this is a full period generator.

As the final step of the process, the elements of the sequence are normalized by division by the modulus:

\[
U_i = X_i / m.
\]

In so doing, we obtain a sequence of random numbers that fall between zero and one. Specifically, a mixed congruential generator (\( c \neq 0 \)) produces numbers in the interval \([0,1)\), whereas a multiplicative congruential generator (\( c=0 \)) produces numbers in the interval \((0,1)\).
- The Minimal Standard Random Number Generator
- Implementation
The Minimal Standard Random Number Generator

A great deal of research has gone into the question of finding an appropriate set of parameters to use in Lehmer's algorithm. A good generator has the following characteristics:

- It is a full period generator.
- The generated sequence passes statistical tests of randomness.
- The generator can be implemented efficiently using 32-bit integer arithmetic.

The choice of modulus depends on the arithmetic precision used to implement the algorithm. A signed 32-bit integer can represent values between \(-2^{31}\) and \(2^{31} - 1\). Fortunately, the quantity \(2^{31} - 1 = 2147483647\) is a prime number! Therefore, it is an excellent choice for the modulus \(m\).

Because Equation \(c\) is slightly simpler than Equation \(d\), we choose to implement a multiplicative congruential generator \((c=0)\). The choice of a suitable multiplier is more difficult. However, a popular choice is \(a = 16807\) because it satisfies all three criteria given above: It results in a full period random number generator; the generated sequence passes a wide variety of statistical tests for randomness; and it is possible to compute Equation \(d\) using 32-bit arithmetic without overflow.

The algorithm is derived as follows: First, let \(q = m \div a\) and \(r = m \mod a\) In this case, \(q = 127773\), \(r = 2836\), and \(r<q\).

Next, we rewrite Equation \(d\) as follows:

\[
X_{i+1} = aX_i \mod m
\]

\[
= aX_i - m(aX_i \div m)
\]

\[
= aX_i - m(X_i \div q) + m(X_i \div q - aX_i \div m).
\]

This somewhat complicated formula can be simplified if we let \(\delta(X_i) = X_i \div q - aX_i \div m\).
\[ X_{i+1} = a X_i - m \left( \frac{X_i}{q} \right) + m \delta(X_i) \]

\[ = a \left( q \left( \frac{X_i}{q} \right) + X_i \mod q \right) - m \left( \frac{X_i}{q} \right) + m \delta(X_i) \]

\[ = a \left( X_i \mod q \right) + (aq - m) \left( \frac{X_i}{q} \right) + m \delta(X_i) \]

Finally, we make use of the fact that \( m = aq - r \) to get

\[ X_{i+1} = a \left( X_i \mod q \right) - \left( \frac{X_i}{q} \right) + m \delta(X_i). \]  \hspace{1cm} (14.25)

Equation \( \square \) has several nice properties: Both \( a \left( X_i \mod q \right) \) and \( \left( \frac{X_i}{q} \right) \) are positive integers between 0 and \( m-1 \). Therefore the difference \( X_i \mod q - \left( \frac{X_i}{q} \right) \) can be represented using a signed 32-bit integer without overflow. Finally, \( \delta(X_i) \) is either a zero or a one. Specifically, it is zero when the sum of the first two terms in Equation \( \square \) is positive and it is one when the sum is negative. As a result, it is not necessary to compute \( \delta(X_i) \)--a simple test suffices to determine whether the third term is 0 or \( m \).
Implementation

We now describe the implementation of a random number generator based on Equation \( \square \). Program \( \square \) defines the `RandomNumberGenerator` class. This class has only static properties. In addition, the constructor is declared private to prevent instantiation. Because there can only be one instance of a static field, the implementation of the `RandomNumberGenerator` class is an example of the singleton design pattern.

```csharp
public sealed class RandomNumberGenerator
{
    private static int seed = 1;
    private const int a = 16807;
    private const int m = 2147483647;
    private const int q = 127773;
    private const int r = 2336;

    private RandomNumberGenerator()
    {
    }

    public static int Seed
    {
        get { return seed; }
        set
        {
            if (value < 1 || value >= m)
                throw new ArgumentException("invalid seed");
            seed = value;
        }
    }

    public static double Next
    {
        get
        {
```
Program: RandomNumberGenerator class.

The Seed property provides a set accessor to specify the initial seed, $X_0$. The seed must fall between 0 and $m-1$. If it does not, an exception is thrown. The Seed property also provides a get accessor that returns the current seed value.

The Next property generates the elements of the random sequence. Each subsequent call to its get accessor returns the next element of the sequence. The implementation follows directly from Equation [27]. Notice that the return value is normalized. Therefore, the values computed by the Next accessor are uniformly distributed on the interval (0,1).
Random Variables

In this section we introduce the notion of an abstract *random variable*. In this context, a random variable is an object that behaves like a random number generator in that it produces a pseudorandom number sequence. The distribution of the values produced depends on the class of random variable used.

Program defines the RandomVariable interface. The RandomVariable interface provides the single property Next. Given an instance, say \( rv \), of a class that implements the RandomVariable interface, repeated calls of the form

\[ rv.Next; \]

are expected to return successive elements of a pseudorandom sequence.

```csharp
public interface RandomVariable
{
    double Next { get; }
}
```

**Program:** RandomVariable interface.

- A Simple Random Variable
- Uniformly Distributed Random Variables
- Exponentially Distributed Random Variables
A Simple Random Variable

Program defines the SimpleRV class. The SimpleRV class implements the RandomVariable interface defined in Program. This class generates random numbers uniformly distributed in the interval (0,1).

```csharp
public class SimpleRV : RandomVariable
{
    public double Next
    {
        get { return RandomNumberGenerator.Next; }
    }
}
```

Program: SimpleRV class.

The implementation of the SimpleRV class is trivial because the RandomNumberGenerator class generates the desired distribution of random numbers. Consequently, the Next accessor simply calls RandomNumberGenerator.Next.
Uniformly Distributed Random Variables

Program defines the UniformRV class. This class generates random numbers which are uniformly distributed in an arbitrary interval \((u,v)\), where \(u<v\). The parameters \(u\) and \(v\) are specified in the constructor.

```csharp
public class UniformRV : RandomVariable
{

    protected double u = 0.0;
    protected double v = 1.0;

    public UniformRV(double u, double v)
    {
        this.u = u;
        this.v = v;
    }

    public double Next
    {
        get
        {
            return u + (v - u) * RandomNumberGenerator.Next;
        }
    }
}
```

Program: UniformRV class.

The UniformRV class is also quite simple. Given that the RandomNumberGenerator class generates a sequence random numbers \(U_i\) uniformly distributed on the interval \((0,1)\), the linear transformation

\[
V_i = u + (v - u)U_i
\]
Uniformly Distributed Random Variables

It suffices to produce a sequence of random numbers $V_i$ uniformly distributed on the interval $(u,v)$. 

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Exponentially Distributed Random Variables

Program defines the ExponentialRV class. This class generates exponentially distributed random numbers with a mean value of \( \mu \). The mean value \( \mu \) is specified in the constructor.

```csharp
public class ExponentialRV : RandomVariable
{
    protected double mu = 1.0;

    public ExponentialRV(double mu)
    {
        this.mu = mu;
    }

    public double Next
    {
        get
        {
            return -mu * Math.Log(RandomNumberGenerator.Next);
        }
    }
}
```

**Program:** ExponentialRV class.

The ExponentialRV class generates a sequence of random numbers, \( X_i \), exponentially distributed on the interval \((0, \infty)\) and having a mean value \( \mu \). The numbers are said to be exponentially distributed because the probability that \( X_i \) falls between 0 and \( z \) is given by

\[
P[0 < X_i < z] = \int_0^z p(x)dx,
\]

where \( p(x) = \frac{1}{\mu} e^{-x/\mu} \). The function \( p(x) \) is called the probability density function. Thus,
Exponentially Distributed Random Variables

\[ P[0 < X_i < z] = \int_0^z \frac{1}{\mu} e^{-x/\mu} \, dx \]
\[ = 1 - e^{-z/\mu}. \]

Notice that \( P[0 < X_i < z] \) is a value between zero and one. Therefore, given a random variable, \( U_i \), uniformly distributed between zero and one, we can obtain an exponentially distributed variable \( X_i \) as follows:

\[
U_i = 1 - e^{X_i/\mu} \quad \Rightarrow \quad X_i = -\mu \ln(U_i - 1)
\]
\[
= X_i = -\mu \ln(U'_i), \quad U'_i = U_i - 1 \quad (14.26)
\]

Note, if \( U_i \) is uniformly distributed on \((0,1)\), then so too is \( U'_i \). The implementation of the \texttt{Next} method follows directly from Equation (14.26).
Monte Carlo Methods

In this section we consider a method for solving problems using random numbers. The method exploits the statistical properties of random numbers in order to ensure that the correct result is computed in the same way that a gambling casino sets the betting odds in order to ensure that the "house" will always make a profit. For this reason, the problem solving technique is called a Monte Carlo method.

To solve a given problem using a Monte Carlo method we devise an experiment in such a way that the solution to the original problem can be obtained from the experimental results. The experiment typically consists of a series of random trials. A random number generator such as the one given in the preceding section is used to create the series of trials.

The accuracy of the final result usually depends on the number of trials conducted. That is, the accuracy usually increases with the number of trials. This trade-off between the accuracy of the result and the time taken to compute it is an extremely useful characteristic of Monte Carlo methods. If only an approximate solution is required, then a Monte Carlo method can be very fast.

Example-Computing
Example-Computing $\pi$

This section presents a simple, Monte Carlo algorithm to compute the value of $\pi$ from a sequence of random numbers. Consider a square positioned in the $x$-$y$ plane with its bottom left corner at the origin as shown in Figure [1]. The area of the square is $r^2$, where $r$ is the length of its sides. A quarter circle is inscribed within the square. Its radius is $r$ and its center is at the origin of $x$-$y$ plane. The area of the quarter circle is $\pi r^2 / 4$.

![Figure: Illustration of a Monte Carlo method for computing $\pi$.](image)

Suppose we select a large number of points at random inside the square. Some fraction of these points will also lie inside the quarter circle. If the selected points are uniformly distributed, we expect the fraction of points in the quarter circle to be

$$f = \frac{\pi r^2 / 4}{r^2} = \frac{\pi}{4}.$$

Therefore by measuring $f$, we can compute $\pi$. Program □ shows how this can be done.
Example-Computing #tex2html_wrap_inline68478#

Program:

```java
public class Example {
    public static double Pi(int trials) {
        int hits = 0;
        for (int i = 0; i < trials; ++i) {
            double x = RandomNumberGenerator.Next;
            double y = RandomNumberGenerator.Next;
            if (x * x + y * y < 1.0)
                ++hits;
        }
        return 4.0 * hits / trials;
    }
}
```

The Pi method uses the `RandomNumberGenerator` defined to generate (x,y) pairs uniformly distributed on the unit square (r=1). Each point is tested to see if it falls inside the quarter circle. A given point is inside the circle when its distance from the origin, \( \sqrt{x^2 + y^2} \) is less than r. In this case since r=1, we simply test whether \( x^2 + y^2 < 1 \).

How well does Program work? When 1000 trials are conducted, 792 points are found to lie inside the circle. This gives the value of 3.168 for \( \pi \), which is only 0.8\% too large. When 10^8 trials are conducted, 78535956 points are found to lie inside the circle. In this case, we get \( \pi \approx 3.14143824 \) which is within 0.005\% of the correct value!
Simulated Annealing

Despite its name, simulated annealing has nothing to do either with simulation or annealing. Simulated annealing is a problem solving technique based loosely on the way in which a metal is annealed in order to increase its strength. When a heated metal is cooled very slowly, it freezes into a regular (minimum-energy) crystalline structure.

A simulated annealing algorithm searches for the optimum solution to a given problem in an analogous way. Specifically, it moves about randomly in the solution space looking for a solution that minimizes the value of some objective function. Because it is generated randomly, a given move may cause the objective function to increase, to decrease or to remain unchanged.

A simulated annealing algorithm always accepts moves that decrease the value of the objective function. Moves that increase the value of the objective function are accepted with probability

\[ p = e^{\Delta / T}, \]

where \( \Delta \) is the change in the value of the objective function and \( T \) is a control parameter called the temperature. That is, a random number generator that generates numbers distributed uniformly on the interval (0,1) is sampled, and if the sample is less than \( p \), the move is accepted.

By analogy with the physical process, the temperature \( T \) is initially high. Therefore, the probability of accepting a move that increases the objective function is initially high. The temperature is gradually decreased as the search progresses. That is, the system is cooled slowly. In the end, the probability of accepting a move that increases the objective function becomes vanishingly small. In general, the temperature is lowered in accordance with an annealing schedule.

The most commonly used annealing schedule is called exponential cooling. Exponential cooling begins at some initial temperature, \( T_0 \), and decreases the temperature in steps according to \( T_{k+1} = \alpha T_k \), where \( 0 < \alpha < 1 \). Typically, a fixed number of moves must be accepted at each temperature before proceeding to the next. The algorithm terminates either when the temperature reaches some final value, \( T_f \), or when some other stopping criterion has been met.
The choice of suitable values for $\alpha$, $T_0$, and $T_f$ is highly problem-dependent. However, empirical evidence suggests that a good value for $\alpha$ is 0.95 and that $T_0$ should be chosen so that the initial acceptance probability is 0.8. The search is terminated typically after some fixed, total number of solutions have been considered.

Finally, there is the question of selecting the initial solution from which to begin the search. A key requirement is that it be generated quickly. Therefore, the initial solution is generated typically at random. However, sometimes the initial solution can be generated by some other means such as with a greedy algorithm.
Example-Balancing Scales

Consider again the *scales balancing problem* described in Section 4. That is, we are given a set of \( n \) weights, \( \{w_1, w_2, \ldots, w_n\} \), which are to be placed on a pair of scales in the way that minimizes the difference between the total weight in each pan. Feasible solution to the problem all have the form \( X = \{x_1, x_2, \ldots, x_n\} \), where

\[
x_i = \begin{cases} 
0 & \text{if } w_i \text{ is placed in the left pan,} \\
1 & \text{if } w_i \text{ is placed in the right pan.}
\end{cases}
\]

To solve this problem using simulated annealing, we need a strategy for generating random moves. The move generator should make small, random changes to the current solution and it must ensure that all possible solutions can be reached. A simple approach is to use the formula

\[
X_{i+1} = X_i \odot U
\]

where \( X_i \) is the initial solution, \( X_{i+1} \) is a new solution, \( U = \{u_1, u_2, \ldots, u_n\} \) is a sequence of zeroes and ones generated randomly, and \( \odot \) denotes elementwise addition modulo two.
Exercises

1. Consider the greedy strategy for counting out change given in Section □. Let \( \{d_1, d_2, \ldots, d_n\} \) be the set of available denominations. For example, the set \( \{1, 5, 10, 25, 100, 200\} \) represents the denominations of the commonly circulated Canadian coins. What condition(s) must the set of denominations satisfy to ensure the greedy algorithm always finds an optimal solution?

2. Devise a greedy algorithm to solve optimally the scales balancing problem described in Section □.
   1. Does your algorithm always find the optimal solution?
   2. What is the running time of your algorithm?

3. Consider the following 0/1-knapsack problem:

<table>
<thead>
<tr>
<th>( i )</th>
<th>( w_i )</th>
<th>( p_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>( C )</td>
<td></td>
<td>18</td>
</tr>
</tbody>
</table>

1. Solve the problem using the greedy by profit, greedy by weight and greedy by profit density strategies.
2. What is the optimal solution?

4. Consider the breadth-first solver shown in Program □. Suppose we replace the queue (line 3) with a priority queue.
   1. How should the solutions in the priority queue be prioritized?
   2. What possible benefit might there be from using a priority queue rather than a FIFO queue?

5. Repeat Exercise □, but this time consider what happens if we replace the queue with a LIFO stack.

6. Repeat Exercises □ and □, but this time consider a branch-and-bound breadth-first solver.
Exercises

7. (This question should be attempted after reading Chapter [ ]). For some problems the solution space is more naturally a graph rather than a tree.

   1. What problem arises if we use the DepthFirstSolver given in Program [ ] to explore a search space that is not a tree.
   2. Modify the DepthFirstSolver so that it explores a solution space that is not a tree.
      **Hint**: See Program [ ].
   3. What problem arises if we use the BreadthFirstSolver given in Program [ ] to explore a search space that is not a tree.
   4. Modify the BreadthFirstSolver so that it explores a solution space that is not a tree.
      **Hint**: See Program [ ].

8. Devise a backtracking algorithm to solve the \(N\)-queens problem: Given an \(N \times N\) chess board, find a way to place \(N\) queens on the board in such a way that no queen can take another.

9. Consider a binary search tree that contains \(n\) keys, \(k_1, k_2, \ldots, k_n\), at depths \(d_1, d_2, \ldots, d_n\), respectively. Suppose the tree will be subjected to a large number of \text{Find} operations. Let \(P_i\) be the probability that we access key \(k_i\). Suppose we know \textit{a priori} all the access probabilities. Then we can say that the \textit{optimal binary search tree} is the tree which minimizes the quantity

\[
\sum_{i=1}^{n} p_i (d_i + 1).
\]

   1. Devise a dynamic programming algorithm that, given the access probabilities, determines the optimal binary search tree.
   2. What is the running time of your algorithm?
      **Hint**: Let \(C_{i,j}\) be the \textit{cost} of the optimal binary search tree that contains the set of keys \(\{k_i, k_{i+1}, k_{i+2}, \ldots, k_j\}\) where \(i \leq j\). Show that

\[
C_{i,j} = \begin{cases} 
    p_i & \text{if } i = j, \\
    \min_{i \leq k \leq j} \{C_{i,k-1} + C_{k+1,j} + \sum_{l=i}^{j} p_l\} & \text{if } i < j.
\end{cases}
\]

10. Consider the typesetting problem discussed in Section [ ]. The objective is to determine how to break a given sequence of words into lines of text of the appropriate size. This was done either by stretching or compressing the space between the words. Explain why the greedy strategy always finds the optimal solution if we stretch but do not compress the space between words.

11. Consider two complex numbers, \(a+bi\) and \(c+di\). Show that we can compute the product \((ac-bd)+(ad+bc)i\) with only three multiplications.

12. Devise a divide-and-conquer strategy to find the root of a polynomial. For example, given a
polynomial such as \( p(x) = 2x^2 + 3x - 4 \), and an interval \([u,v]\) such that \( p(u) \) and \( p(v) \) have opposite signs, find the value \( r, u \leq r \leq v \), such that \( p(r) = 0 \).

13. Devise an algorithm to compute a *normally distributed random variable*. A normal distribution is complete defined by its mean and standard deviation. The probability density function for a normal distribution is

\[
p(x) = \frac{1}{\sigma \sqrt{2\pi}} \exp \left( -\frac{1}{2\sigma^2}(x - \mu)^2 \right),
\]

where \( \mu \) is the mean and \( \sigma \) is the standard deviation of the distribution. **Hint:** Consider the *central limit theorem*.

14. Devise an algorithm to compute a *geometrically distributed random variable*. A geometrically distributed random variable is an integer in the interval \([1, \infty)\) given by the probability density function

\[
P[X = i] = \theta(1 - \theta)^{i-1},
\]

where \( \theta^{-1} \) is the mean of the distribution.

**Hint:** Use the fact \( P[X = i] = P[i - 1 < Z \leq i] \), where \( Z \) is an exponentially distributed random variable with mean \( \mu = -1/\ln(1 - \theta) \).

15. Do Exercise □.
Projects

1. Design a class that implements the Solution interface defined in Program to represent the nodes of the solution space of a 0/1-knapsack problem described in Section.

Devise a suitable representation for the state of a node and then implement the following properties IsFeasible, IsComplete, Objective, Bound, and Successors. Note, the Successors method returns an IEnumerable object that that represents all the successors of a given node.

1. Use your class with the DepthFirstSolver defined in Program to solve the problem given in Table.
2. Use your class with the BreadthFirstSolver defined in Program to solve the problem given in Table.
3. Use your class with the DepthFirstBranchAndBoundSolver defined in Program to solve the problem given in Table.

2. Do Project for the change counting problem described in Section.
3. Do Project for the scales balancing problem described in Section.
4. Do Project for the N-queens problem described in Exercise.
5. Design and implement a GreedySolver class, along the lines of the DepthFirstSolver and BreadthFirstSolver classes, that conducts a greedy search of the solution space. To do this you will have to add a method to the Solution interface:

```csharp
public interface GreedySolution : Solution
{
    Solution GreedySuccessor();
}
```

6. Design and implement a SimulatedAnnealingSolver class, along the lines of the DepthFirstSolver and BreadthFirstSolver classes, that implements the simulated annealing strategy described in Section. To do this you will have to add a method to the Solution interface:
public interface SimulatedAnnealingSolution : Solution
{
    Solution RandomSuccessor();
}

7. Design and implement a dynamic programming algorithm to solve the change counting problem. Your algorithm should always find the optimal solution—even when the greedy algorithm fails.

8. Consider the divide-and-conquer strategy for matrix multiplication described in Section □.
   1. Rewrite the implementation of the Times method of the Matrix class introduced in Program □.
   2. Compare the running time of your implementation with the $O(n^3)$ algorithm given in Program □.

9. Consider random number generator that generates random numbers uniformly distributed between zero and one. Such a generator produces a sequence of random numbers $x_1, x_2, x_3, \ldots$. A common test of randomness evaluates the correlation between consecutive pairs of numbers in the sequence. One way to do this is to plot on a graph the points

   \[(x_1, x_2), (x_2, x_3), (x_3, x_4), \ldots\]

   1. Write a program to compute the first 1000 pairs of numbers generated using the UniformRV defined in Program □.
   2. What conclusions can you draw from your results?
Sorting Algorithms and Sorters

- Basics
- Sorting and Sorters
- Insertion Sorting
- Exchange Sorting
- Selection Sorting
- Merge Sorting
- A Lower Bound on Sorting
- Distribution Sorting
- Performance Data
- Exercises
- Projects

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Consider an arbitrary sequence \( S = \{ s_1, s_2, s_3, \ldots, s_n \} \) comprised of \( n \geq 0 \) elements drawn from a some universal set \( U \). The goal of sorting is to rearrange the elements of \( S \) to produce a new sequence, say \( S' \), in which the elements of \( S \) appear in order.

But what does it mean for the elements of \( S' \) to be in order? We shall assume that there is a relation, \(<\), defined over the universe \( U \). The relation \(<\) must be a total order, which is defined as follows:

**Definition** A total order is a relation, say \(<\), defined on the elements of some universal set \( U \) with the following properties:

1. For all pairs of elements \((i, j) \in U \times U\), exactly one of the following is true: \( i < j \), \( i = j \), or \( j < i \).

   (All elements are commensurate.)

2. For all triples \((i, j, k) \in U \times U \times U\), \( i < j \land j < k \iff i < k \).

   (The relation \(<\) is transitive.)

In order to sort the elements of the sequence \( S \), we determine the permutation \( P = \{ p_1, p_2, p_3, \ldots, p_n \} \) of the elements of \( S \) such that

\[
\xi_{p_1} \leq \xi_{p_2} \leq \xi_{p_3} \leq \cdots \leq \xi_{p_n}.
\]

In practice, we are not interested in the permutation \( P \), per se. Instead, our objective is to compute the sorted sequence \( S' = \{ s'_1, s'_2, s'_3, \ldots, s'_n \} \) in which \( s'_{i} = \xi_{p_i} \) for \( 1 \leq i \leq n \).

Sometimes the sequence to be sorted, \( S \), contains duplicates. That is, there exist values \( i \) and \( j \), \( 1 \leq i < j \leq n \), such that \( \xi_i = \xi_j \). In general when a sequence that contains duplicates is sorted, there is no guarantee that the duplicated elements retain their relative positions. That is, \( \xi_i \) could appear either before or after \( \xi_j \) in the sorted sequence \( S' \). If duplicates retain their relative positions in the sorted sequence the sort is said to be stable. In order for \( \xi_i \) and \( \xi_j \) to retain their relative order in the sorted
sequence, we require that \( \sigma'_{p_i} \) precedes \( \sigma'_{p_j} \) in \( S' \). Therefore, the sort is stable if \( p_i < p_j \).
Sorting and Sorters

The traditional way to implement a sorting algorithm is to write a method that sorts an array of data. This chapter presents an alternate, object-oriented approach that is based on the notion of an *abstract sorter*.

Think of a sorter as an abstract machine, the sole purpose of which is to sort arrays of data. A machine is an object. Therefore, it makes sense that we represent it as an instance of some class. The machine sorts data. Therefore, the class will have a method, say `Sort`, which sorts an array of data.

Program defines the `Sorter` interface. The interface consists of the single method `Sort`. This method takes as its argument an array of `ComparableObjects` and it sorts the objects therein.

```csharp
public interface Sorter
{
    void Sort(ComparableObject[] array);
}
```

Program: `Sorter` interface.

- Abstract Sorters
- Sorter Class Hierarchy
Abstract Sorters

Program defines the AbstractSorter class. The AbstractSorter class implements the Sorter interface defined in Program. The AbstractSorter comprises the two fields, array and n, the concrete methods Swap and Sort (ComparableObject[]), and the no-arg abstract method Sort(). Since the no-arg Sort method is an abstract method, an implementation must be given in a derived class.

```csharp
public abstract class AbstractSorter : Sorter
{
    protected ComparableObject[] array;
    protected int n;

    protected abstract void Sort();

    public virtual void Sort(ComparableObject[] array)
    {
        n = array.Length;
        this.array = array;
        if (n > 0)
            Sort();
        this.array = null;
    }

    protected virtual void Swap(int i, int j)
    {
        ComparableObject tmp = array[i];
        array[i] = array[j];
        array[j] = tmp;
    }
}
```

Program: AbstractSorter class.
Abstract Sorters

The `Sort(ComparableObject[])` method does not sort the data itself. It is the no-arg `Sort` method, which is provided by a derived class, that does the actual sorting. The `Sort(ComparableObject[])` method merely sets-up things by initializing the fields of `AbstractSorter` as follows: The array field refers to the array of objects to be sorted and the length of that array is assigned to the `n` field.

The `Swap` method is used to implement most of the sorting algorithms presented in this chapter. The `swap` method takes two integers arguments. It exchanges the contents of the array at the positions specified by the arguments. The exchange is done as a sequence of three assignments. Therefore, the `Swap` method runs in constant time.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Sorter Class Hierarchy

This chapter describes nine different sorting algorithms. These are organized into the following five categories:

- insertion sorts
- exchange sorts
- selection sorts
- merge sorts
- distribution sorts.

As shown in Figure 1, the sorter classes have been arranged in a class hierarchy that reflects this classification scheme.

Figure: Sorter class hierarchy
Insertion Sorting

The first class of sorting algorithm that we consider comprises algorithms that sort by insertion. An algorithm that sorts by insertion takes the initial, unsorted sequence, \( S = \{ s_1, s_2, s_3, \ldots, s_n \} \), and computes a series of sorted sequences \( S'_0, S'_1, S'_2, \ldots, S'_n \), as follows:

1. The first sequence in the series, \( S'_0 \), is the empty sequence. That is, \( S'_0 = \{ \} \).
2. Given a sequence \( S'_i \) in the series, for \( 0 \leq i < n \), the next sequence in the series, \( S'_{i+1} \), is obtained by inserting the \( (i+1)^{th} \) element of the unsorted sequence \( s_{i+1} \) into the correct position in \( S'_i \).

Each sequence \( S'_i \), \( 0 \leq i < n \), contains the first \( i \) elements of the unsorted sequence \( S \). Therefore, the final sequence in the series, \( S'_n \), is the sorted sequence we seek. That is, \( S' = S'_n \).

Figure illustrates the insertion sorting algorithm. The figure shows the progression of the insertion sorting algorithm as it sorts an array of ten integers. The array is sorted in place. That is, the initial unsorted sequence, \( S \), and the series of sorted sequences, \( S'_0, S'_1, \ldots \), occupy the same array.
In the $i^{th}$ step, the element at position $i$ in the array is inserted into the sorted sequence $S'_i$ which occupies array positions 0 to ($i-1$). After this is done, array positions 0 to $i$ contain the $i+1$ elements of $S'_{i+1}$. Array positions ($i+1$) to ($n-1$) contain the remaining $n-i-1$ elements of the unsorted sequence $S$. 

**Figure:** Insertion sorting.
As shown in Figure 1, the first step ($i=0$) is trivial--inserting an element into the empty list involves no work. Altogether, $n-1$ non-trivial insertions are required to sort a list of $n$ elements.

- **Straight Insertion Sort**
- **Average Running Time**
- **Binary Insertion Sort**
Straight Insertion Sort

The key step of any insertion sorting algorithm involves the insertion of an item into a sorted sequence. There are two aspects to an insertion--finding the correct position in the sequence at which to insert the new element and moving all the elements over to make room for the new one.

This section presents the *straight insertion sorting* algorithm. Straight insertion sorting uses a *linear search* to locate the position at which the next element is to be inserted.

- Implementation
Implementation

Program defines the StraightInsertionSorter class. The StraightInsertionSorter extends the AbstractSorter class defined in Program. It simply provides an implementation for the no-arg Sort method.

```
public class StraightInsertionSorter : AbstractSorter
{
    protected override void Sort()
    {
        for (int i = 1; i < n; ++i)
        {
            for (int j = i;
                j > 0 && array[j - 1] > array[j]; --j)
            {
                Swap(j, j - 1);
            }
        }
    }
}
```

Program: StraightInsertionSorter class Sort method.

In order to determine the running time of the Sort method, we need to determine the number of iterations of the inner loop (lines 6-10). The number of iterations of the inner loop in the $i^{th}$ iteration of the outer loop depends on the positions of the values in the array. In the best case, the value in position $i$ of the array is larger than that in position $i-1$ and zero iterations of the inner loop are done. In this case, the running time for insertion sort is $O(n)$. Notice that the best case performance occurs when we sort an array that is already sorted!

In the worst case, $i$ iterations of the inner loop are required in the $i^{th}$ iteration of the outer loop. This occurs when the value in position $i$ of the array is smaller than the values at positions 0 through $i-1$. Therefore, the worst case arises when we sort an array in which the elements are initially sorted in reverse. In this case the running time for insertion sort is $O(n^2)$. 


Average Running Time

The best case running time of insertion sorting is $O(n)$ but the worst-case running time is $O(n^2)$. Therefore, we might suspect that the average running time falls somewhere in between. In order to determine it, we must define more precisely what we mean by the *average* running time. A simple definition of average running time is to say that it is the running time needed to sort the average sequence. But what is the average sequence?

The usual way to determine the average running time of a sorting algorithm is to consider only sequences that contain no duplicates. Since every sorted sequence of length $n$ is simply a permutation of an unsorted one, we can represent every such sequence by a permutation of the sequence $S = \{1, 2, 3, \ldots, n\}$. When computing the average running time, we assume that every permutation is equally likely. Therefore, the average running time of a sorting algorithm is the running time averaged over all permutations of the sequence $S$.

Consider a permutation $P = \{p_1, p_2, p_3, \ldots, p_n\}$ of the sequence $S$. An *inversion* in $P$ consists of two elements, say $p_i$ and $p_j$, such that $p_i > p_j$ but $i < j$. That is, an inversion in $P$ is a pair of elements that are in the wrong order. For example, the permutation $\{1, 4, 3, 2\}$ contains three inversions--(4,3), (4,2), and (3,2). The following theorem tells us how many inversions we can expect in the average sequence:

**Theorem** The average number of inversions in a permutation of $n$ distinct elements is $n(n-1)/4$.

**Proof** Let $S$ be an arbitrary sequence of $n$ distinct elements and let $S^R$ be the same sequence, but in reverse.

For example, if $S = \{s_1, s_2, s_3, \ldots, s_n\}$, then $S^R = \{s_n, s_{n-1}, s_{n-2}, \ldots, s_1\}$.

Consider any pair of distinct elements in $S$, say $s_i$ and $s_j$ where $1 \leq i < j \leq n$. There are two distinct
possibilities: Either $i < j$, in which case $(i, j)$ is an inversion in $S^R$; or $j < i$, in which case $(j, i)$ is an inversion in $S$. Therefore, every pair contributes exactly one inversion either to $S$ or to $S^R$.

The total number of pairs in $S$ is $\binom{n}{2} = \frac{n(n-1)}{2}$. Since every such pair contributes an inversion either to $S$ or to $S^R$, we expect on average that half of the inversions will appear in $S$. Therefore, the average number of inversions in a sequence of $n$ distinct elements is $n(n-1)/4$.

What do inversions have to do with sorting? As a list is sorted, inversions are removed. In fact, since the inner loop of the insertion sort method swaps adjacent array elements, inversions are removed one at a time! Since a swap takes constant time, and since the average number of inversions is $n(n-1)/4$, the average running time for the insertion sort method is $O(n^2)$.
Binary Insertion Sort

The straight insertion algorithm presented in the preceding section does a linear search to find the position in which to do the insertion. However, since the element is inserted into a sequence that is already sorted, we can use a binary search instead of a linear search. Whereas a linear search requires \( O(n) \) comparisons in the worst case, a binary search only requires \( O(\log n) \) comparisons. Therefore, if the cost of a comparison is significant, the binary search may be preferred.

Program [ ] defines the BinaryInsertionSorter class. The BinaryInsertionSorter class extends the AbstractSorter class defined in Program [ ]. The framework of the Sort method is essentially the same as that of the StraightInsertionSorter class.
public class BinaryInsertionSorter : AbstractSorter
{
    protected override void Sort()
    {
        for (int i = 1; i < n; ++i)
        {
            ComparableObject tmp = array[i];
            int left = 0;
            int right = i;
            while (left < right)
            {
                int middle = (left + right) / 2;
                if (tmp >= array[middle])
                    left = middle + 1;
                else
                    right = middle;
            }
            for (int j = i; j > left; --j)
                Swap(j - 1, j);
        }
    }
}

Program: BinaryInsertionSorter class Sort method.

Exactly, n-1 iterations of the outer loop are done (lines 5-20). In each iteration, a binary search search is done to determine the position at which to do the insertion (lines 7-17). In the $i^{th}$ iteration of the outer loop, the binary search considers array positions 0 to $i$ (for $1 \leq i < n$). The running time for the binary search in the $i^{th}$ iteration is $O([\log_2(i + 1)]) = O(\log i)$. Once the correct position is found, at most $i$ swaps are needed to insert the element in its place.

The worst-case running time of the binary insertion sort is dominated by the $i$ swaps needed to do the insertion. Therefore, the worst-case running time is $O(n^2)$. Furthermore, since the algorithm only swaps adjacent array elements, the average running time is also $O(n^2)$ (see Section □). Asymptotically, the binary insertion sort is no better than straight insertion.

However, the binary insertion sort does fewer array element comparisons than insertion sort. In the $i^{th}$
iteration of the outer loop, the binary search requires \( \left\lfloor \log_2 (i + 1) \right\rfloor \) comparisons, for \( 1 \leq i < n \). Therefore, the total number of comparisons is

\[
\sum_{i=1}^{n-1} \left\lfloor \log_2 (i + 1) \right\rfloor = \sum_{i=1}^{n} \left\lfloor \log_2 i \right\rfloor = (n + 1) \left\lfloor \log_2 (n + 1) \right\rfloor + 2 \left\lfloor \log_2 (n+1) \right\rfloor + 2 = \mathcal{O}(n \log n)
\]

(This result follows directly from Theorem □).

The number of comparisons required by the straight insertion sort is \( \mathcal{O}(n^2) \) in the worst case as well as on average. Therefore on average, the binary insertion sort uses fewer comparisons than straight insertion sort. On the other hand, the previous section shows that in the best case the running time for straight insertion is \( O(n) \). Since the binary insertion sort method always does the binary search, its best case running time is \( \mathcal{O}(n \log n) \). Table □ summarizes the asymptotic running times for the two insertion sorts.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>best case</td>
</tr>
<tr>
<td>straight insertion sort</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>binary insertion sort</td>
<td>( \mathcal{O}(n \log n) )</td>
</tr>
</tbody>
</table>

Table: Running times for insertion sorting.
Exchange Sorting

The second class of sorting algorithm that we consider comprises algorithms that sort by exchanging pairs of items until the sequence is sorted. In general, an algorithm may exchange adjacent elements as well as widely separated ones.

In fact, since the insertion sorts considered in the preceding section accomplish the insertion by swapping adjacent elements, insertion sorting can be considered as a kind of exchange sort. The reason for creating a separate category for insertion sorts is that the essence of those algorithms is insertion into a sorted list. On the other hand, an exchange sort does not necessarily make use of such a sorted list.

- Bubble Sort
- Quicksort
- Running Time Analysis
- Average Running Time
- Selecting the Pivot
Bubble Sort

The simplest and, perhaps, the best known of the exchange sorts is the bubble sort. Figure shows the operation of bubble sort.
To sort the sequence $S = \{S_0, S_1, S_2, \ldots, S_{n-1}\}$, bubble sort makes $n-1$ passes through the data. In each pass, adjacent elements are compared and swapped if necessary. First, $S_0$ and $S_1$ are compared; next, $S_1$ and $S_2$; and so on.

Notice that after the first pass through the data, the largest element in the sequence has bubbled up into the last array position. In general, after $k$ passes through the data, the last $k$ elements of the array are correct and need not be considered any longer. In this regard the bubble sort differs from the insertion sort algorithms—the sorted subsequence of $k$ elements is never modified (by an insertion).

Figure also shows that while $n-1$ passes through the data are required to guarantee that the list is sorted in the end, it is possible for the list to become sorted much earlier! When no exchanges at all are made in a given pass, then the array is sorted and no additional passes are required. A minor algorithmic modification would be to count the exchanges made in a pass, and to terminate the sort when none are made.

Program defines the BubbleSorter class. The BubbleSorter class extends the AbstractSorter class defined in Program. It simply provides an implementation for the no-arg Sort method.

```java
public class BubbleSorter : AbstractSorter
{
    protected override void Sort()
    {
        for (int i = n; i > 1; --i)
            for (int j = 0; j < i - 1; ++j)
                if (array[j] > array[j + 1])
                    Swap(j, j + 1);
    }
}
```

Program: BubbleSorter class Sort method.

The outer loop (lines 5-8) is done for $i = n - 1, n - 2, n - 3, \ldots, 2$. That makes $n-1$ iterations in total. During the $i^{th}$ iteration of the outer loop, exactly $i-1$ iterations of the inner loop are done (lines 6-8). Therefore, the number of iterations of the inner loop, summed over all the passes of the outer loop is
\[
\sum_{i=2}^{n} (i - 1) = \sum_{i=1}^{n-1} i = \frac{n(n - 1)}{2}.
\]

Consequently, the running time of bubble sort is \(\Theta(n^2)\).

The body of the inner loop compares adjacent array elements and swaps them if necessary (lines 7-8). This takes at most a constant amount of time. Of course, the algorithm will run slightly faster when no swapping is needed. For example, this occurs if the array is already sorted to begin with. In the worst case, it is necessary to swap in every iteration of the inner loop. This occurs when the array is sorted initially in reverse order. Since only adjacent elements are swapped, bubble sort removes inversions one at a time. Therefore, the average number of swaps required is \(O(n^2)\). Nevertheless, the running time of bubble sort is always \(\Theta(n^2)\).
Quicksort

The second exchange sort we consider is the quicksort algorithm. Quicksort is a divide-and-conquer style algorithm. A divide-and-conquer algorithm solves a given problem by splitting it into two or more smaller subproblems, recursively solving each of the subproblems, and then combining the solutions to the smaller problems to obtain a solution to the original one.

To sort the sequence \( S = \{s_1, s_2, s_3, \ldots, s_n\} \), quicksort performs the following steps:

1. Select one of the elements of \( S \). The selected element, \( p \), is called the pivot.
2. Remove \( p \) from \( S \) and then partition the remaining elements of \( S \) into two distinct sequences, \( L \) and \( G \), such that every element in \( L \) is less than or equal to the pivot and every element in \( G \) is greater than or equal to the pivot. In general, both \( L \) and \( G \) are unsorted.
3. Rearrange the elements of the sequence as follows:

\[
S' = \left\{ l_1, l_2, \ldots, l_{|L|}, p, g_1, g_2, \ldots, g_{|G|} \right\}
\]

Notice that the pivot is now in the position in which it belongs in the sorted sequence, since all the elements to the left of the pivot are less than or equal to the pivot and all the elements to the right are greater than or equal to it.
4. Recursively quicksort the unsorted sequences \( L \) and \( G \).

The first step of the algorithm is a crucial one. We have not specified how to select the pivot. Fortunately, the sorting algorithm works no matter which element is chosen to be the pivot. However, the pivot selection affects directly the running time of the algorithm. If we choose poorly the running time will be poor.

Figure illustrates the detailed operation of quicksort as it sorts the sequence \( \{3, 1, 4, 1, 5, 9, 2, 6, 5, 4\} \). To begin the sort, we select a pivot. In this example, the value 4 in the last array position is chosen. Next, the remaining elements are partitioned into two sequences, one which contains values less than or equal to 4 ( \( L = \{3, 1, 2, 1\} \)) and one which contains values greater than or equal to 4 ( \( G = \{5, 9, 4, 6, 5\} \)). Notice that the partitioning is accomplished by exchanging elements. This is why quicksort is considered to be an exchange sort.
Figure: "Quick" sorting.

After the partitioning, the pivot is inserted between the two sequences. This is called restoring the pivot. To restore the pivot, we simply exchange it with the first element of $G$. Notice that the 4 is in its correct position in the sorted sequence and it is not considered any further.
Now the quicksort algorithm calls itself recursively, first to sort the sequence \( L = \{3, 1, 2, 1\} \); second to sort the sequence \( G = \{9, 4, 6, 5, 5\} \). The quicksort of \( L \) selects 1 as the pivot, and creates the two subsequences \( L' = \{1\} \) and \( G' = \{2, 3\} \). Similarly, the quicksort of \( G \) uses 5 as the pivot and creates the two subsequences \( L'' = \{5, 4\} \) and \( G'' = \{9, 6\} \).

At this point in the example the recursion has been stopped. It turns out that to keep the code simple, quicksort algorithms usually stop the recursion when the length of a subsequence falls below a critical value called the cut-off. In this example, the cut-off is two (i.e., a subsequence of two or fewer elements is not sorted). This means that when the algorithm terminates, the sequence is not yet sorted. However as Figure shows, the sequence is almost sorted. In fact, every element is guaranteed to be less than two positions away from its final resting place.

We can complete the sorting of the sequence by using a straight insertion sort. In Section it is shown that straight insertion is quite good at sorting sequences that are almost sorted. In fact, if we know that every element of the sequence is at most \( d \) positions from its final resting place, the running time of straight insertion is \( O(dn) \) and since \( d=2 \) is a constant, the running time is \( O(n) \).

---

- **Implementation**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Implementation

Program \( \square \) introduces the \texttt{AbstractQuickSorter} class. The \texttt{AbstractQuickSorter} class extends the \texttt{AbstractSorter} class defined in Program \( \square \). It declares the abstract method \texttt{SelectPivot} the implementation of which is provided by a derived class.

```
1 public abstract class AbstractQuickSorter : AbstractSorter
2 {
3     protected const int CUTOFF = 2; // minimum cut-off
4     
5     protected abstract int SelectPivot(int left, int right);
6     // ...
7 }
```

**Program:** AbstractQuickSorter fields.

Program \( \square \) defines a \texttt{Sort} method of the \texttt{AbstractQuickSorter} class that takes two integer arguments, \texttt{left} and \texttt{right}, which denote left and right ends, respectively, of the section of the array to be sorted. That is, this \texttt{Sort} method sorts

\[
\text{array[left], array[left + 1], \ldots, array[right]}
\]
Implementation

```java
public abstract class AbstractQuickSorter : AbstractSorter
{
    protected void Sort(int left, int right)
    {
        if (right - left + 1 > CUTOFF)
        {
            int p = SelectPivot(left, right);
            Swap(p, right);
            ComparableObject pivot = array[right];
            int i = left;
            int j = right - 1;
            for (; ;)
            {
                while (i < j && array[i] < pivot) ++i;
                while (i < j && array[j] > pivot) --j;
                if (i >= j) break;
                Swap(i++, j--);
            }
            if (array[i] > pivot)
                Swap(i, right);
            if (left < i)
                Sort(left, i - 1);
            if (right > i)
                Sort(i + 1, right);
        }
    }
    // ...
}
```

Program: AbstractQuickSorter class recursive Sort method.

As discussed above, the AbstractQuickSorter only sorts sequences whose length exceeds the cut-off value. Since the implementation shown only works correctly when the number of elements in the sequence to be sorted is three or more, the cut-off value of two is used (line 5).

The algorithm begins by calling the method SelectPivot which chooses one of the elements to be the pivot (line 7). The implementation of SelectPivot is discussed below. All that we require here is that the value \( p \) returned by SelectPivot satisfies \( \text{left} \leq p \leq \text{right} \). Having selected an element to be the pivot, we hide the pivot by swapping it with the right-most element of the sequence (line 8). The
Implementation

pivot is hidden in order to get it out of the way of the next step.

The next step partitions the remaining elements into two sequences—one comprised of values less than or equal to the pivot, the other comprised of values greater than or equal to the pivot. The partitioning is done using two array indices, i and j. The first, i, starts at the left end and moves to the right; the second, j, starts at the right end and moves to the left.

The variable i is increased as long as array[i] is less than the pivot (line 14). Then the variable j is decreased as long as array[j] is greater than the pivot (line 15). When i and j meet, the partitioning is done (line 16). Otherwise, \( i < j \) but \( array[i] \geq pivot \geq array[j] \). This situation is remedied by swapping array[i] and array[j] (line 17).

When the partitioning loop terminates, the pivot is still in array[right]; the value in array[i] is greater than or equal to the pivot; everything to the left is less than or equal to the pivot; and everything to the right is greater than or equal to the pivot. We can now put the pivot in its proper place by swapping it with array[i] (lines 19-20). This is called restoring the pivot. With the pivot in its final resting place, all we need to do is sort the subsequences on either side of the pivot (lines 21-24).

Program [□] defines the no-arg Sort method of the AbstractQuickSorter class. The no-arg Sort acts as the front end to the recursive Sort given in Program [□]. It calls the recursive Sort method with left set to zero and right set to \( n-1 \), where \( n \) is the length of the array to be sorted. Finally, it uses a StraightInsertionSorter to finish sorting the list.

```java
public abstract class AbstractQuickSorter : AbstractSorter
{
    protected override void Sort()
    {
        Sort(0, n - 1);
        Sorter sorter = new StraightInsertionSorter();
        sorter.Sort(array);
    }
    // ...
}
```

Program: AbstractQuickSorter class Sort method.
Running Time Analysis

The running time of the recursive `Sort` method (Program □) is given by

\[
T(n) = \begin{cases} 
O(1) & n \leq 2, \\
T(\{\text{SelectPivot}\}) + T(i) + T(n - i - 1) + O(n) & n > 2, 
\end{cases} 
\]

(15.1)

where \(n\) is the number of elements in sequence to be sorted, \(T(\{\text{SelectPivot}\})\) is the running time of the `SelectPivot` method, and \(i\) is the number of elements which end up to the left of the pivot, \(0 \leq i \leq n - 1\).

The running time of `Sort` is affected by the `SelectPivot` method in two ways: First, the value of the pivot chosen affects the sizes of the subsequences. That is, the pivot determines the value \(i\) in Equation □. Second, the running time of the `SelectPivot` method itself, \(T(\{\text{SelectPivot}\})\), must be taken into account. Fortunately, if \(T(\{\text{SelectPivot}\}) = O(n)\), we can ignore its running time because there is already an \(O(n)\) term in the expression.

In order to solve Equation □, we assume that \(T(\{\text{SelectPivot}\}) = O(n)\) and then drop the \(O(\cdot)\)s from the recurrence to get

\[
T(n) = \begin{cases} 
1 & n \leq 2, \\
T(i) + T(n - i - 1) + n & n > 2, 
\end{cases} 
\]

(15.2)

Clearly the solution depends on the value of \(i\).

- **Worst-Case Running Time**
- **Best-Case Running Time**
Worst-Case Running Time

In the worst case the \( i \) in Equation \( 1 \) is always zero. In this case, we solve the recurrence using repeated substitution like this:

\[
T(n) = T(n - 1) + n \\
= T(n - 2) + (n - 1) + n \\
= T(n - 3) + (n - 2) + (n - 1) + n \\
\vdots \\
= T(n - k) + \sum_{j=n-k}^{n} j \\
\vdots \\
= T(2) + \sum_{j=2}^{n} j \\
= n(n + 1)/2 \\
= \mathcal{O}(n^2).
\]

The worst case occurs when the two subsequences are as unbalanced as they can be--one sequence has all the remaining elements and the other has none.
Best-Case Running Time

In the best case, the partitioning step divides the remaining elements into two sequences with exactly the same number of elements. For example, suppose that $n = 2^m - 1$ for some integer $m > 0$. After removing the pivot, $2^m - 2$ elements remain. If these are divided evenly, each sequence will have $2^{m-1} - 1$ elements. In this case Equation gives

$$
T(2^m - 1) = 2T(2^{m-1} - 1) + 2^m - 1
= 2^2 T(2^{m-2} - 1) + 2 \cdot 2^m - 2 - 1
= 2^3 T(2^{m-3} - 1) + 3 \cdot 2^m - 3 - 2 - 1
= \ldots
= 2^k T(2^{m-k} - 1) + k 2^m - \sum_{j=1}^{k} j
= 2^{m-1} T(1) + (m - 1) 2^m - \sum_{j=1}^{m-1} j, \quad m - k = 1
= \frac{(2^m (2^m - 1) - m(m - 1))}{2}
= \frac{[(n + 1)(2 \log_2 (n + 1) - 1) - (\log_2 (n + 1) - 1) \log_2 (n + 1)]}{2}
= O(n \log n).
$$
Average Running Time

To determine the average running time for the quicksort algorithm, we shall assume that each element of the sequence has an equal chance of being selected for the pivot. Therefore, if \( i \) is the number of elements in a sequence of length \( n \) less than the pivot, then \( i \) is uniformly distributed in the interval \([0, n-1]\). Consequently, the average value of \( T(i) = \frac{1}{n} \sum_{j=0}^{n-1} T(j) \). Similarly, the average the value of \( T(n - i - 1) = \frac{1}{n} \sum_{j=0}^{n-1} T(n - j - 1) \). To determine the average running time, we rewrite Equation (15.2) thus:

\[
T(n) = \begin{cases} 
1 & n \leq 2, \\
\frac{1}{n} \sum_{j=0}^{n-1} T(j) + \frac{1}{n} \sum_{j=0}^{n-1} T(n - j - 1) + n & n > 2
\end{cases}
\]

Equation (15.3)

To solve this recurrence we consider the case \( n > 2 \) and then multiply Equation (15.3) by \( n \) to get

\[
nT(n) = 2 \sum_{j=0}^{n-1} T(j) + n^2.
\]

Equation (15.4)

Since this equation is valid for any \( n > 2 \), by substituting \( n-1 \) for \( n \) we can also write

\[
(n - 1)T(n - 1) = 2 \sum_{j=0}^{n-2} T(j) + n^2 - 2n + 1.
\]

Equation (15.5)

which is valid for \( n > 3 \). Subtracting Equation (15.5) from Equation (15.4) gives

\[
nT(n) - (n - 1)T(n - 1) = 2T(n - 1) + 2n - 1
\]

which can be rewritten as
\[
\frac{T(n)}{n + 1} = \frac{T(n - 1)}{n} + \frac{2}{n + 1} - \frac{1}{n(n + 1)}. \quad (15.6)
\]

Equation (15.6) can be solved by telescoping like this:

\[
\frac{T(n)}{n + 1} = \frac{T(n - 1)}{n} + \frac{2}{n + 1} - \frac{1}{(n)(n + 1)}
\]

\[
\frac{T(n - 1)}{n - 1} = \frac{T(n - 2)}{n - 1} + \frac{2}{n} \quad \frac{1}{(n - 1)(n)}
\]

\[
\frac{T(n - 1)}{n - 1} = \frac{T(n - 3)}{n - 2} + \frac{2}{n - 1} - \frac{1}{(n - 2)(n - 1)}
\]

\[
\vdots \quad \frac{T(n - k)}{n - k + 1} = \frac{T(n - k - 1)}{n - k} + \frac{2}{n - k + 1} - \frac{1}{(n - k)(n - k + 1)}
\]

\[
\vdots \quad \frac{T(3)}{4} = \frac{T(2)}{2} + \frac{2}{4} - \frac{1}{(3)(4)}. \quad (15.8)
\]

Adding together Equation (15.7) through Equation (15.8) gives

\[
\frac{T(n)}{n + 1} = \frac{T(2)}{3} + 2 \sum_{i=4}^{n+1} \frac{1}{i} - \sum_{i=3}^{n} \frac{1}{i(i + 1)}
\]

\[
= 2 \sum_{i=1}^{n+1} \frac{1}{i} - \sum_{i=1}^{n} \frac{1}{i(i + 1)} - 2
\]

\[
= 2H_{n+1} + \frac{1}{n + 1} - 3, 
\]

where \(H_{n+1}\) is the \((n + 1)^{th}\) harmonic number. Finally, multiplying through by \(n+1\) gives

\[
T(n) = 2(n + 1)H_{n+1} - 3n - 2.
\]

In Section it is shown that \(H_n \approx \ln n + \gamma\), where \(\gamma \approx 0.577215\) is called Euler's constant. Thus,
we get that the average running time of quicksort is

\[ T(n) \approx 2(n + 1)(\ln(n + 1) + \gamma) - 3n - 3 = \mathcal{O}(n \log n). \]

Table 4 summarizes the asymptotic running times for the quicksort method and compares it to those of bubble sort. Notice that the best-case and average case running times for the quicksort algorithm have the same asymptotic bound!

<table>
<thead>
<tr>
<th>algorithm</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble sort</td>
<td>( \mathcal{O}(n^2) )</td>
</tr>
<tr>
<td>quicksort (random pivot selection)</td>
<td>( \mathcal{O}(n \log n) )</td>
</tr>
</tbody>
</table>

Table: Running times for exchange sorting.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Selecting the Pivot

The analysis in the preceding section shows that selecting a good pivot is important. If we do a bad job of choosing the pivot, the running time of quicksort is $O(n^2)$. On the other hand, the average-case analysis shows that if every element of a sequence is equally likely to be chosen for the pivot, the running time is $O(n \log n)$. This suggests that we can expect to get good performance simply by selecting a random pivot!

If we expect to be sorting random input sequences, then we can achieve random pivot selection simply by always choosing, say, the first element of the sequence to be the pivot. Clearly this can be done in constant time. (Remember, the analysis requires that $T(\text{SelectPivot}) = O(n)$.) As long as each element in the sequence is equally likely to appear in the first position, the average running time will be $O(n \log n)$.

In practice it is often the case that the sequence to be sorted is almost sorted. In particular, consider what happens if the sequence to be sorted using quicksort is already sorted. If we always choose the first element as the pivot, then we are guaranteed to have the worst-case running time! This is also true if we always pick the last element of the sequence. And it is also true if the sequence is initially sorted in reverse.

Therefore, we need to be more careful when choosing the pivot. Ideally, the pivot divides the input sequence exactly in two. That is, the ideal pivot is the median element of the sequence. This suggests that the SelectPivot method should find the median. To ensure that the running time analysis is valid, we need to find the median in $O(n)$ time.

How do you find the median? One way is to sort the sequence and then select the $\lceil n/2 \rceil$th element. But this is not possible, because we need to find the median to sort the sequence in the first place!

While it is possible to find the median of a sequence of $n$ elements in $O(n)$ time, it is usually not necessary to do so. All that we really need to do is select a random element of the sequence while avoiding the problems described above.

A common way to do this is the median-of-three pivot selection technique. In this approach, we choose as the pivot the median of the element at the left end of the sequence, the element at the right end of the
Selecting the Pivot

sequence, and the element in the middle of the sequence. Clearly, this does the right thing if the input sequence is initially sorted (either in forward or reverse order).

Program □ defines the MedianOfThreeQuickSorter class. The MedianOfThreeQuickSorter class extends the abstract AbstractQuickSorter class introduced in Program □. It provides an implementation for the SelectPivot method based on median-of-three pivot selection. Notice that this algorithm does exactly three comparisons to select the pivot. As a result, its running time is $O(1)$. In practice this scheme performs sufficiently well that more complicated pivot selection approaches are unnecessary.

```
public class MedianOfThreeQuickSorter : AbstractQuickSorter
{
    protected override int SelectPivot(int left, int right)
    {
        int middle = (left + right) / 2;
        if (array[left] > array[middle])
            Swap(left, middle);
        if (array[left] > array[right])
            Swap(left, right);
        if (array[middle] > array[right])
            Swap(middle, right);
        return middle;
    }
}
```

Program: MedianOfThreeQuickSorter class SelectPivot method.
Selection Sorting

The third class of sorting algorithm that we consider comprises algorithms that sort by selection. Such algorithms construct the sorted sequence one element at a time by adding elements to the sorted sequence in order. At each step, the next element to be added to the sorted sequence is selected from the remaining elements.

Because the elements are added to the sorted sequence in order, they are always added at one end. This is what makes selection sorting different from insertion sorting. In insertion sorting elements are added to the sorted sequence in an arbitrary order. Therefore, the position in the sorted sequence at which each subsequent element is inserted is arbitrary.

Both selection sorts described in this section sort the arrays in place. Consequently, the sorts are implemented by exchanging array elements. Nevertheless, selection differs from exchange sorting because at each step we select the next element of the sorted sequence from the remaining elements and then we move it into its final position in the array by exchanging it with whatever happens to be occupying that position.
Straight Selection Sorting

The simplest of the selection sorts is called straight selection. Figure illustrates how straight selection works. In the version shown, the sorted list is constructed from the right (i.e., from the largest to the smallest element values).
Straight Selection Sorting

**Figure:** Straight selection sorting.

At each step of the algorithm, a linear search of the unsorted elements is made in order to determine the position of the largest remaining element. That element is then moved into the correct position of the array by swapping it with the element which currently occupies that position.

For example, in the first step shown in Figure 1, a linear search of the entire array reveals that 9 is the largest element. Since 9 is the largest element, it belongs in the last array position. To move it there, we swap it with the 4 that initially occupies that position. The second step of the algorithm identifies 6 as the largest remaining element and moves it next to the 9. Each subsequent step of the algorithm moves one element into its final position. Therefore, the algorithm is done after $n-1$ such steps.

---

- **Implementation**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Implementation

Program defines the StraightSelectionSorter class. This class is derived from the AbstractSorter base defined in Program and it provides an implementation for the no-arg Sort method. The Sort method follows directly from the algorithm discussed above. In each iteration of the main loop (lines 5-12), exactly one element is selected from the unsorted elements and moved into the correct position. A linear search of the unsorted elements is done in order to determine the position of the largest remaining element (lines 8-10). That element is then moved into the correct position (line 11).

```
public class StraightSelectionSorter : AbstractSorter
{
    protected override void Sort()
    {
        for (int i = n; i > 1; --i)
        {
            int max = 0;
            for (int j = 1; j < i; ++j)
                if (array[j] > array[max])
                    max = j;
            Swap(i - 1, max);
        }
    }
}
```

**Program:** StraightSelectionSorter class Sort method.

In all $n-1$ iterations of the outer loop are needed to sort the array. Notice that exactly one swap is done in each iteration of the outer loop. Therefore, $n-1$ data exchanges are needed to sort the list.

Furthermore, in the $i^{th}$ iteration of the outer loop, $i-1$ iterations of the inner loop are required and each iteration of the inner loop does one data comparison. Therefore, $O(n^2)$ data comparisons are needed to sort the list.
The total running time of the straight selection sort method is $O(n^2)$. Because the same number of comparisons and swaps are always done, this running time bound applies in all cases. That is, the best-case, average-case and worst-case running times are all $O(n^2)$. 
Sorting with a Heap

Selection sorting involves the repeated selection of the next element in the sorted sequence from the set of remaining elements. For example, the straight insertion sorting algorithm given in the preceding section builds the sorted sequence by repeatedly selecting the largest remaining element and prepending it to the sorted sequence developing at the right end of the array.

At each step the largest remaining element is withdrawn from the set of remaining elements. A linear search is done because the order of the remaining elements is arbitrary. However, if we consider the value of each element as its priority, we can view the set of remaining elements as a priority queue. In effect, a selection sort repeatedly dequeues the highest priority element from a priority queue.

Chapter presents a number of priority queue implementations, including binary heaps, leftist heaps and binomial queues. In this section we present a version of selection sorting that uses a binary heap to hold the elements that remain to be sorted. Therefore, it is called a heapsort. The principal advantage of using a binary heap is that it is easily implemented using an array and the entire sort can be be done in place.

As explained in Section, a binary heap is a complete binary tree which is easily represented in an array. The nodes of the heap occupy positions 1 through \( n \) of the array. The root is at position 1. In general, the children of the node at position \( i \) of the array are found at positions \( 2i \) and \( 2i+1 \), and the parent is found at position \( \lceil i/2 \rceil \).

The heapsort algorithm consists of two phases. In the first phase, the unsorted array is transformed into a heap. (This is called heapifying the array). In this case, a max-heap rather than a min-heap is used. The data in a max heap satisfies the following condition: For every node in the heap that has a parent, the item contained in the parent is greater than or equal to the item contained in the given node.

The second phase of heapsort builds the sorted list. The sorted list is built by repeatedly selecting the largest element, withdrawing it from the heap, and adding it to the sorted sequence. As each element is withdrawn from the heap, the remaining elements are heapified.
Implementation
Implementation

In the first phase of heapsort, the unsorted array is transformed into a max heap. Throughout the process we view the array as a complete binary tree. Since the data in the array is initially unsorted, the tree is not initially heap-ordered. We make the tree into a max heap from the bottom up. That is, we start with the leaves and work towards the root. Figure illustrates this process.

![Diagram of heap operations](http://www.brpreiss.com/books/opus6/html/page504.html)

**Figure**: Combining heaps by percolating values.

Figure (a) shows a complete tree that is not yet heap ordered--the root is smaller than both its children. However, the two subtrees of the root are heap ordered. Given that both of the subtrees of the root are already heap ordered, we can heapify the tree by percolating the value in the root down the tree.

To percolate a value down the tree, we swap it with its largest child. For example, in Figure (b) we...
Implementation

swap 3 and 7. Swapping with the largest child ensures that after the swap, the new root is greater than or equal to both its children.

Notice that after the swap the heap-order is satisfied at the root, but not in the left subtree of the root. We continue percolating the 3 down by swapping it with 6 as shown in Figure (c). In general, we percolate a value down either until it arrives in a position in which the heap order is satisfied or until it arrives in a leaf. As shown in Figure (d), the tree obtained when the percolation is finished is a max heap.

Program introduces the HeapSorter class. The HeapSorter class extends the AbstractSorter class defined in Program. The PercolateDown method shown in Program implements the algorithm described above. The PercolateDown method takes two arguments: the number of elements in the array to be considered, n, and the position, i, of the node to be percolated.

```java
public class HeapSorter : AbstractSorter
{
    protected const int baseIndex = 1;
    protected void PercolateDown(int i, int length)
    {
        while (2 * i <= length)
        {
            int j = 2 * i;
            if (j < length &&
                array[j + 1 - baseIndex] > array[j - baseIndex])
                j = j + 1;
            if (array[i - baseIndex] >= array[j - baseIndex])
                break;
            Swap(i - baseIndex, j - baseIndex);
            i = j;
        }
    }
}
```

Program: HeapSorter class PercolateDown method.

The purpose of the PercolateDown method is to transform the subtree rooted at position i into a max heap. It is assumed that the left and right subtrees of the node at position i are already max heaps. Recall
that the children of node \( i \) are found at positions \( 2i \) and \( 2i+1 \). **PercolateDown** percolates the value in position \( i \) down the tree by swapping elements until the value arrives in a leaf node or until both children of \( i \) contain smaller value.

A constant amount of work is done in each iteration. Therefore, the running time of the **PercolateDown** method is determined by the number of iterations of its main loop (lines 7-17). In fact, the number of iterations required in the worst case is equal to the height in the tree of node \( i \).

Since the root of the tree has the greatest height, the worst-case occurs for \( i = 1 \). In Chapter 4 it is shown that the height of a complete binary tree is \( \lfloor \log_2 n \rfloor \). Therefore the worst-case running time of the **PercolateDown** method is \( \mathcal{O}(\log n) \).

Recall that **BuildHeap** calls **PercolateDown** for \( i = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \lfloor n/2 \rfloor - 1, \ldots, 1 \). If we assume that the worst-case occurs every time, the running time of **BuildHeap** is \( \mathcal{O}(n \log n) \).
Building the Heap

The BuildHeap method shown in Program 1 transforms an unsorted array into a max heap. It does so by calling the PercolateDown method for \( i = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \lfloor n/2 \rfloor - 2, \ldots, 1. \)

```csharp
public class HeapSorter : AbstractSorter {
    protected const int baseIndex = 1;
    protected void BuildHeap()
    {
        for (int i = n / 2; i > 0; --i)
            PercolateDown(i, n);
    }
    // ...
}
```

Program: HeapSorter class BuildHeap method.

Why does BuildHeap start percolating at \( \lfloor n/2 \rfloor \)? A complete binary tree with \( n \) nodes has exactly \( \lfloor n/2 \rfloor \) leaves. Therefore, the last node in the array which has a child is in position \( \lfloor n/2 \rfloor \). Consequently, the BuildHeap method starts doing percolate down operations from that point.

The BuildHeap visits the array elements in reverse order. In effect the algorithm starts at the deepest node that has a child and works toward the root of the tree. Each array position visited is the root of a subtree. As each such subtree is visited, it is transformed into a max heap. Figure 2 illustrates how the BuildHeap method heapifies an array that is initially unsorted.
Figure: Building a heap.
Building the Heap

- Running Time Analysis
- The Sorting Phase

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Running Time Analysis

The BuildHeap method does exactly \( \left\lfloor \frac{n}{2} \right\rfloor \) PercolateDown operations. As discussed above, the running time for PercolateDown is \( \mathcal{O}(h_i) \), where \( h_i \) is the height in the tree of the node at array position \( i \). The highest node in the tree is the root and its height is \( \mathcal{O}(\log n) \). If we make the simplifying assumption that the running time for PercolateDown is \( \mathcal{O}(\log n) \) for every value of \( i \), we get that the total running time for BuildHeap is \( \mathcal{O}(n \log n) \).

However, \( n \log n \) is not a tight bound. The maximum number of iterations of the PercolateDown loop done during the entire process of building the heap is equal to the sum of the heights of all of the nodes in the tree! The following theorem shows that this is \( O(n) \).

**Theorem** Consider a perfect binary tree \( T \) of height \( h \) having \( n = 2^{h+1} - 1 \) nodes. The sum of the heights of the nodes in \( T \) is \( 2^{h+1} - 1 - (h + 1) = n - \log_2(n + 1) \).

**Proof** A perfect binary tree has 1 node at height \( h \), 2 nodes at height \( h-1 \), 4 nodes at height \( h-2 \) and so on. In general, there are \( 2^i \) nodes at height \( h-i \). Therefore, the sum of the heights of the nodes is \( \sum_{i=0}^{h} (h - i)2^i \).

The summation can be solved as follows: First, we make the simple variable substitution \( i=j-1 \):
\[
\sum_{i=0}^{h} (h - i)2^i = \sum_{j=0}^{h} (h - (j - 1))2^{j-1}
\]

\[
= \frac{1}{2} \sum_{j=1}^{h+1} (h - j + 1)2^j
\]

\[
= \frac{1}{2} \sum_{j=0}^{h} (h - j + 1)2^j - (h + 1)/2
\]

\[
= \frac{1}{2} \sum_{j=0}^{h} (h - j)2^j + \sum_{j=0}^{h} 2^j - (h + 1)/2
\]

\[
= \frac{1}{2} \sum_{j=0}^{h} (h - j)2^j + (2^{h+1} - 1 - h + 1)/2 \quad (15.9)
\]

Note that the summation which appears on the right hand side is identical to that on the left. Rearranging Equation (15.9) and simplifying gives:

\[
\sum_{i=0}^{h} (h - i)2^i = 2^{h+1} - 1 - h + 1
\]

\[
= n - \log_2(n + 1).
\]

It follows directly from Theorem □ that the sum of the heights of a perfect binary tree is \(O(n)\). But a heap is not a perfect tree---it is a complete tree. Nevertheless, it is easy to show that the same bound applies to a complete tree. The proof is left as an exercise for the reader (Exercise □). Therefore, the running time for the BuildHeap method is \(O(n)\), were \(n\) is the length of the array to be heapified.

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
The Sorting Phase

Once the max heap has been built, heapsort proceeds to the selection sorting phase. In this phase the sorted sequence is obtained by repeatedly withdrawing the largest element from the max heap. Figure illustrates how this is done.
**Figure:** Heap sorting.

The largest element of the heap is always found at the root and the root of a complete tree is always in array position one. Suppose the heap occupies array positions 1 through $k$. When an element is withdrawn from the heap, its length decreases by one. That is, after the withdrawal the heap occupies array positions 1 through $k-1$. Thus, array position $k$ is no longer required by the max heap. However, the next element of the sorted sequence belongs in position $k$!

So, the sorting phase of heapsort works like this: We repeatedly swap the largest element in the heap (always in position 1) into the next position of the sorted sequence. After each such swap, there is a new value at the root of the heap and this new value is pushed down into the correct position in the heap using the `PercolateDown` method.

Program gives the `Sort` method of the `HeapSorter` class. The `Sort` method embodies both phases of the heapsort algorithm. In the first phase of heapsort the `BuildHeap` method is called to transform the array into a max heap. As discussed above, this is done in $O(n)$ time.

```java
public class HeapSorter : AbstractSorter
{
    protected const int baseIndex = 1;

    protected override void Sort()
    {
        BuildHeap();
        for (int i = n; i >= 2; --i)
        {
            Swap(i - baseIndex, 1 - baseIndex);
            PercolateDown(1, i - 1);
        }
        // ...
    }
}
```

**Program:** `HeapSorter` class `Sort` method.

The second phase of the heapsort algorithm builds the sorted list. In all $n-1$ iterations of the loop on lines 8-12 are required. Each iteration involves one swap followed by a `PercolateDown` operation.
Since the worst-case running time for PercolateDown is $O(\log n)$, the total running time of the loop is $O(n \log n)$. The running time of the second phase asymptotically dominates that of the first phase. As a result, the worst-case running time of heapsort is $O(n \log n)$. 
Merge Sorting

The fourth class of sorting algorithm we consider comprises algorithms that sort by merging. Merging is the combination of two or more sorted sequences into a single sorted sequence.

Figure illustrates the basic, two-way merge operation. In a two-way merge, two sorted sequences are merged into one. Clearly, two sorted sequences each of length \( n \) can be merged into a sorted sequence of length \( 2n \) in \( O(2n)=O(n) \) steps. However in order to do this, we need space in which to store the result. That is, it is not possible to merge the two sequences in place in \( O(n) \) steps.

Figure: Two-way merging.

Sorting by merging is a recursive, divide-and-conquer strategy. In the base case, we have a sequence with exactly one element in it. Since such a sequence is already sorted, there is nothing to be done. To sort a sequence of \( n>1 \) elements:

1. Divide the sequence into two sequences of length \( \lfloor n/2 \rfloor \) and \( \lceil n/2 \rceil \);
2. recursively sort each of the two subsequences; and then,
3. merge the sorted subsequences to obtain the final result.

Figure illustrates the operation of the two-way merge sort algorithm.
Figure: Two-way merge sorting.

- Implementation
- Merging
- Two-Way Merge Sorting
- Running Time Analysis
Implementation

Program declares the TwoWayMergeSorter class. The TwoWayMergeSorter class extends the AbstractSorter class defined in Program. A single field, tempArray, is declared. This field is an array of ComparableObjects. Since merge operations cannot be done in place, a second, temporary array is needed. The tempArray field keeps track of that array.

```csharp
public class TwoWayMergeSorter : AbstractSorter {

    ComparableObject[] tempArray;

    // ...
}
```

Program: TwoWayMergeSorter fields.
Merging

The Merge method of the TwoWayMergeSorter class is defined in Program □. Altogether, this method takes three integer parameters, left, middle, and right. It is assumed that

\[ left \leq middle \leq right. \]

Furthermore, it is assumed that the two subsequences of the array,

\[ array[left], array[left + 1], \ldots, array[middle], \]

and

\[ array[middle + 1], array[middle + 2], \ldots, array[right], \]

are both sorted. The Merge method merges the two sorted subsequences using the temporary array, tempArray. It then copies the merged (and sorted) sequence into the array at

\[ array[left], array[left + 1], \ldots, array[right]. \]
public class TwoWayMergeSorter : AbstractSorter
{
    ComparableObject[] tempArray;

    protected void Merge(int left, int middle, int right)
    {
        int i = left;
        int j = left;
        int k = middle + 1;
        while (j <= middle & & k <= right)
        {
            if (array[j] < array[k])
                tempArray[i++] = array[j++];
            else
                tempArray[i++] = array[k++];
        }
        while (j <= middle)
            tempArray[i++] = array[j++];
        for (i = left; i < k; ++i)
            array[i] = tempArray[i];
    }
    // ...
Two-Way Merge Sorting

Program gives the code for two `Sort` methods of the `TwoWayMergeSorter` class. The no-arg `Sort` method sets things up for the second, recursive `Sort` method. First, it allocates a temporary array, the length of which is equal to the length of the array to be sorted (line 7). Then it calls the recursive `Sort` method which sorts the array (line 8). After the array has been sorted, the no-arg `Sort` discards the temporary array (line 9).

```csharp
public class TwoWayMergeSorter : AbstractSorter
{
    ComparableObject[] tempArray;

    protected override void Sort()
    {
        tempArray = new ComparableObject[n];
        Sort(0, n - 1);
        tempArray = null;
    }

    protected void Sort(int left, int right)
    {
        if (left < right)
        {
            int middle = (left + right) / 2;
            Sort(left, middle);
            Sort(middle + 1, right);
            Merge(left, middle, right);
        }
        // ...
    }
}
```

Program: `TwoWayMergeSorter` class `Sort` methods.
Two-Way Merge Sorting

The second `Sort` method implements the recursive, divide-and-conquer merge sort algorithm described above. The method takes two parameters, `left` and `right`, that specify the subsequence of the array to be sorted. If the sequence to be sorted contains more than one element, the sequence is split in two (line 16), each half is recursively sorted (lines 17-18), and then two sorted halves are merged (line 19).
Running Time Analysis

The running time of merge sort is determined by the running time of the recursive `Sort` method. (The no-arg `Sort` method adds only a constant amount of overhead). The running time of the recursive `Sort` method is given by the following recurrence:

\[
T(n) = \begin{cases} 
O(1) & n = 1, \\
T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n) & n > 1,
\end{cases} \quad (15.10)
\]

where \( n = \text{right} - \text{left} + 1 \).

In order to simplify the solution of Equation \(15.10\) we shall assume that \( n = 2^k \) for some integer \( k \geq 0 \). Dropping the \( O(\cdot) \)s from the equation we get

\[
T(n) = \begin{cases} 
1 & n = 1, \\
2T(n/2) + n & n > 1,
\end{cases}
\]

which is easily solved by repeated substitution:

\[
T(n) = 2T(n/2) + n = 4T(n/4) + 2n = 8T(n/8) + 3n \ldots = 2^k T(n/2^k) + kn \ldots = nT(1) + n \log_2 n = n + n \log_2 n
\]

Therefore, the running time of merge sort is \( O(n \log n) \).
A Lower Bound on Sorting

The preceding sections present three \( O(n \log n) \) sorting algorithms--quicksort, heapsort, and the two-way merge sort. But is \( O(n \log n) \) the best we can do? In this section we answer the question by showing that any sorting algorithm that sorts using only binary comparisons must make \( \Omega(n \log n) \) such comparisons. If each binary comparison takes a constant amount of time, then running time for any such sorting algorithm is also \( \Omega(n \log n) \).

Consider the problem of sorting the sequence \( S = \{a, b, c\} \) comprised of three distinct items. That is, \( a \neq b \land a \neq c \land b \neq c \). Figure 513 illustrates a possible sorting algorithm in the form of a decision tree. Each node of the decision tree represents one binary comparison. That is, in each node of the tree, exactly two elements of the sequence are compared. Since there are exactly two possible outcomes for each comparison, each non-leaf node of the binary tree has degree two.
For example, suppose that \(a < b < c\). Consider how the algorithm shown in Figure \(\square\) discovers this. The first comparison compares \(a\) and \(b\) which reveals that \(a < b\). The second comparison compares \(a\) and \(c\) to find that \(a < c\). At this point it has been determined that \(a < b\) and \(a < c\) -- the relative order of \(b\) and \(c\) is not yet known. Therefore, one more comparison is required to determine that \(b < c\). Notice that the algorithm shown in Figure \(\square\) works correctly in all cases because every possible permutation of the sequence \(S\) appears as a leaf node in the decision tree. Furthermore, the number of comparisons required in the worst case is equal to the height of the decision tree!

Any sorting algorithm that uses only binary comparisons can be represented by a binary decision tree. Furthermore, it is the height of the binary decision tree that determines the worst-case running time of the algorithm. In general, the size and shape of the decision tree depends on the sorting algorithm and the number of items to be sorted.

Given an input sequence of \(n\) items to be sorted, every binary decision tree that correctly sorts the input sequence must have at least \(n!\) leaves -- one for each permutation of the input. Therefore, it follows directly from Theorem \(\square\) that the height of the binary decision tree is at least \(\lceil \log_2 n! \rceil\):

\[
\lceil \log_2 n! \rceil \geq \log_2 n! \geq \sum_{i=1}^{\lfloor n/2 \rfloor} \log_2 i \geq \sum_{i=1}^{\lfloor n/2 \rfloor} \log_2 \frac{n}{2} = \Omega(n \log n).
\]

Since the height of the decision tree is \(\Omega(n \log n)\), the number of comparisons done by any sorting algorithm that sorts using only binary comparisons is \(\Omega(n \log n)\). Assuming each comparison can be done in constant time, the running time of any such sorting algorithm is \(\Omega(n \log n)\).
The final class of sorting algorithm considered in this chapter consists of algorithms that sort by distribution. The unique characteristic of a distribution sorting algorithm is that it does not make use of comparisons to do the sorting.

Instead, distribution sorting algorithms rely on a priori knowledge about the universal set from which the elements to be sorted are drawn. For example, if we know a priori that the size of the universe is a small, fixed constant, say $m$, then we can use the bucket sorting algorithm described in Section B. Similarly, if we have a universe the elements of which can be represented with a small, finite number of bits (or even digits, letters, or symbols), then we can use the radix sorting algorithm given in Section C.

- Bucket Sort
- Radix Sort
Bucket Sort

Bucket sort is possibly the simplest distribution sorting algorithm. The essential requirement is that the size of the universe from which the elements to be sorted are drawn is a small, fixed constant, say \( m \).

For example, suppose that we are sorting elements drawn from \( \{0, 1, \ldots, m - 1\} \), i.e., the set of integers in the interval \([0,m-1]\). Bucket sort uses \( m \) counters. The \( i \)\textsuperscript{th} counter keeps track of the number of occurrences of the \( i \)\textsuperscript{th} element of the universe. Figure \[
\begin{array}{c}
\begin{array}{cccc}
3 & 1 & 4 & 1 \\
1 & 5 & 9 & 2 \\
5 & 6 & 5 & 4
\end{array}
data \\
\begin{array}{cccc}
0 & 2 & 1 & 1 \\
2 & 2 & 1 & 0 \\
0 & 1 & 0 & 1
\end{array}counts \\
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 9 & 9
\end{array}
\] illustrates how this is done.

In Figure \[
\begin{array}{c}
\begin{array}{cccc}
3 & 1 & 4 & 1 \\
1 & 5 & 9 & 2 \\
5 & 6 & 5 & 4
\end{array}data \\
\begin{array}{cccc}
0 & 2 & 1 & 1 \\
2 & 2 & 1 & 0 \\
0 & 1 & 0 & 1
\end{array}counts \\
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 9 & 9
\end{array}
\] the universal set is assumed to be \( \{0, 1, \ldots, 9\} \). Therefore, ten counters are required--one to keep track of the number of zeroes, one to keep track of the number of ones, and so on. A single pass through the data suffices to count all of the elements. Once the counts have been determined, the sorted sequence is easily obtained. For example, the sorted sequence contains no zeroes, two ones, one two, and so on.

- Implementation
Implementation

Program introduces the BucketSorter class. The BucketSorter class extends the AbstractSorter class defined in Program. This bucket sorter is designed to sort specifically an array of ComparableInt32s. The BucketSorter class contains two fields, m and count. The integer m simply keeps track of the size of the universe. The count variable is an array of integers used to count the number of occurrences of each element of the universal set.

Program: BucketSorter class fields and constructor.

The constructor for the BucketSorter class takes a single argument which specifies the size of the universal set. The variable m is set to the specified value, and the count array is initialized to have the required size.

Program defines the no-arg Sort method. It begins by setting all of the counters to zero (lines 8-9). This can clearly be done in $O(m)$ time.
public class BucketSorter : AbstractSorter
{
    protected int m;
    protected int[] count;

    protected override void Sort()
    {
        for (int i = 0; i < m; ++i)
            count[i] = 0;
        for (int j = 0; j < n; ++j)
            ++count[(int)array[j]];
        for (int i = 0, j = 0; i < m; ++i)
        {
            for ( ; count[i] > 0; --count[i])
                array[j++] = i;
        }
        // ...
    }
}

Program: BucketSorter class Sort method.

Next, a single pass is made through the data to count the number of occurrences of each element of the universe (lines 10-11). Since each element of the array is examined exactly once, the running time is \(O(n)\).

In the final step, the sorted output sequence is created (lines 12-14). Since the output sequence contains exactly \(n\) items, the body of the inner loop (line 14) is executed exactly \(n\) times. During the \(i^{th}\) iteration of the outer loop (line 12), the loop termination test of the inner loop (line 13) is evaluated \(\text{count}[i] + 1\) times. As a result, the total running time of the final step is \(O(m+n)\).

Thus, the running time of the bucket sort method is \(O(m+n)\). Note that if \(m=O(n)\), the running time for bucket sort is \(O(n)\). That is, the bucket sort algorithm is a linear-time sorting algorithm! Bucket sort breaks the \(\Omega(n \log n)\) bound associated with sorting algorithms that use binary comparisons because bucket sort does not do any binary comparisons. The cost associated with breaking the \(\Omega(n \log n)\) running time bound is the \(O(m)\) space required for the array of counters. Consequently, bucket sort is practical only for small \(m\). For example, to sort 16-bit integers using bucket sort requires the use of an array of \(2^{16} = 65536\) counters.
Radix Sort

This section presents a sorting algorithm known as least-significant-digit-first radix sorting. Radix sorting is based on the bucket sorting algorithm discussed in the preceding section. However, radix sorting is practical for much larger universal sets than it is practical to handle with a bucket sort.

Radix sorting can be used when each element of the universal set can be viewed as a sequences of digits (or letters or any other symbols). For example, we can represent each integer between 0 and 99 as a sequence of two, decimal digits. (For example, the number five is represented as `05`).

To sort an array of two-digit numbers, the algorithm makes two sorting passes through the array. In the first pass, the elements of the array are sorted by the least significant decimal digit. In the second pass, the elements of the array are sorted by the most significant decimal digit. The key characteristic of the radix sort is that the second pass is done in such a way that it does not destroy the effect of the first pass. Consequently, after two passes through the array, the data is contained therein is sorted.

Each pass of the radix sort is implemented as a bucket sort. In the example we base the sort on decimal digits. Therefore, this is called a radix-10 sort and ten buckets are required to do each sorting pass.

Figure illustrates the operation of the radix-10 sort. The first radix sorting pass considers the least significant digits. As in the bucket sort, a single pass is made through the unsorted data, counting the number of times each decimal digit appears as the least-significant digit. For example, there are no elements that have a 0 as the least-significant digit; there are two elements that have a 1 as the least-significant digit; and so on.
Radix Sort

Figure: Radix sorting.

After the counts have been determined, it is necessary to permute the input sequence so that it is sorted by the least-significant digits. To do this permutation efficiently, we compute the sequence of offsets given by

$$\text{offset}[i] = \begin{cases} 
0, & i = 0, \\
\sum_{j=0}^{i-1} \text{count}[j], & 0 < i < R,
\end{cases}$$  \hspace{1cm} (15.11)

where $R$ is the sorting radix. Note that $\text{offset}[i]$ is the position in the permuted sequence of the first occurrence of an element whose least significant digit is $i$. By making use of the offsets, it is possible to permute the input sequence by making a single pass through the sequence.

The second radix sorting pass considers the most significant digits. As above a single pass is made through the permuted data sequence counting the number of times each decimal digit appears as the most-
Radix Sort

significant digit. Then the sequence of offsets is computed as above. The sequence is permuted again using the offsets producing the final, sorted sequence.

In general, radix sorting can be used when the elements of the universe can be viewed as \( p \)-digit numbers with respect to some radix, \( R \). That is, each element of the universe has the form

\[
\sum_{i=0}^{p-1} d_i R^i,
\]

where \( d_i \in \{0, 1, \ldots, R-1\} \) for \( 0 \leq i < p \). In this case, the radix sort algorithm must make \( p \) sorting passes from the least significant digit, \( d_0 \), to the most significant digit, \( d_{p-1} \), and each sorting pass uses exactly \( R \) counters.

Radix sorting can also be used when the universe can be viewed as the cross-product of a finite number of finite sets. That is, when the universe has the form

\[
U = U_1 \times U_2 \times U_3 \times \cdots \times U_p,
\]

where \( p > 0 \) is a fixed integer constant and \( U_i \) is a finite set for \( 1 \leq i \leq p \). For example, each card in a 52-card deck of playing cards can be represented as an element of \( U = U_1 \times U_2 \), where \( U_1 = \{\spadesuit, \heartsuit, \diamondsuit, \clubsuit\} \) and \( U_2 = \{\text{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K}\} \).

Before we can sort over the universe \( U \), we need to define what it means for one element to precede another in \( U \). The usual way to do this is called lexicographic ordering. For example in the case of the playing cards we may say that one card precedes another if its suit precedes the other suit or if the suits are equal but the face value precedes that of the other.

In general, given the universe \( U = U_1 \times U_2 \times U_2 \times \cdots \times U_p \), and two elements of \( U \), say \( x \) and \( y \), represented by the \( p \)-tuples \( x = (x_1, x_2, \ldots, x_p) \) and \( y = (y_1, y_2, \ldots, y_p) \), respectively, we say that \( x \) lexicographically precedes \( y \) if there exists \( 1 \leq k \leq p \) such that \( x_k < y_k \) and \( x_i = y_i \) for all \( 1 \leq i < k \).

With this definition of precedence, we can radix sort a sequence of elements drawn from \( U \) by sorting with respect to the components of the \( p \)-tuples. Specifically, we sort first with respect to \( U_p \), then \( U_{p-1} \), and so on down to \( U_1 \). Notice that the algorithm does \( p \) sorting passes and in the \( i^{th} \) pass it requires \( |U_i| \) counters. For example to sort a deck of cards, two passes are required. In first pass the cards are sorted into 13 piles according to their face values. In the second pass the cards are sorted into four piles.
according to their suits.
Implementation

Program introduces the RadixSorter class. The RadixSorter class extends the AbstractSorter class defined in Program. This radix sort is designed to sort specifically an array of ComparableInt32s.

```csharp
public class RadixSorter : AbstractSorter
{
    protected const int r = 8;
    protected const int R = 1 << r;
    protected const int p = (32 + r - 1) / r;
    protected int[] count = new int[R];
    // ...
}
```

Program: RadixSorter fields.

Three constants are declared in the RadixSorter class--R, r, and p. The constant R represents the radix and \( r = \log_2 R \). The constant p is the number sorting passes needed to sort the data. In this case \( r=8 \) and \( R = 2^r = 256 \). Therefore, a radix-256 sort is being done. We have chosen R as a power of two because that way the computations required to implement the radix sort can be implemented efficiently using simple bit shift and mask operations. In order to sort \( b \)-bit integers, it is necessary to make \( p = \lceil \log_R 2^b \rceil = \lceil b/r \rceil \) sorting passes.

One more field is defined in the RadixSorter class--count. The count field is an array of integers used to implement the sorting passes. An array of integers of length R is created and assigned to the count array.

The no-arg Sort method shown in Program begins by creating a temporary array of ComparableObjects of length \( n \). Each iteration of the main loop corresponds to one pass of the radix sort (lines 8-29). In all \( p \) iterations are required.
public class RadixSorter : AbstractSorter
{
    protected override void Sort()
    {
        ComparableObject[] tempArray =
            new ComparableObject[n];

        for (int i = 0; i < p; ++i)
        {
            for (int j = 0; j < R; ++j)
                count[j] = 0;
            for (int k = 0; k < n; ++k)
            {
                ++count[((int)array[k] >> (r*i)) & (R-1)];
                tempArray[k] = array[k];
            }
            int pos = 0;
            for (int j = 0; j < R; ++j)
            {
                int tmp = pos;
                pos += count[j];
                count[j] = tmp;
            }
            for (int k = 0; k < n; ++k)
            {
                int j = ((int)tempArray[k] >> (r*i)) & (R-1);
                array[count[j]++] = tempArray[k];
            }
        }
        // ...
    }

Program: RadixSorter class Sort method.

During the $i^{th}$ pass of the main loop the following steps are done: First, the $R$ counters are all set to zero (lines 10-11). This takes $O(R)$ time. Then a pass is made through the input array during which the number of occurrences of each radix-$R$ digit in the $i^{th}$ digit position are counted (lines 12-16). This pass takes $O(n)$ time. Notice that during this pass all the input data is copied into the temporary array.
Next, the array of counts is transformed into an array of offsets according to Equation \( \square \). This requires a single pass through the counter array (lines 17-23). Therefore, it takes \( O(R) \) time. Finally, the data sequence is permuted by copying the values from the temporary array back into the input array (lines 24-28). Since this requires a single pass through the data arrays, the running time is \( O(n) \).

After the \( p \) sorting passes have been done, the array of data is sorted. The running time for the \texttt{Sort} method of the \texttt{RadixSorter} class is \( O(p(R + n)) \). If we assume that the size of an integer is 32 bits and given that \( R=256 \), the number of sorting passes required is \( p=4 \). Therefore, the running time for the radix sort is simply \( O(n) \). That is, radix sort is a linear-time sorting algorithm.
**Performance Data**

In order to better understand the actual performance of the various sorting algorithms presented in this chapter, it is necessary to conduct some experiments. Only by conducting experiments is it possible to determine the relative performance of algorithms with the same asymptotic running time.

To measure the performance of a sorting algorithm, we need to provide it with some data to sort. To obtain the results presented here, random sequences of integers were sorted. That is, for each value of \( n \), the `RandomNumberGenerator` class defined in Section was used to create a sequence of \( n \) integers. In all cases (except for bucket sort) the random numbers are uniformly distributed in the interval \([1, 2^{31} - 1]\). For the bucket sort the numbers are uniformly distributed in \([0, 2^{10} - 1]\).

Figures , , and show the actual running times of the sorting algorithms presented in this chapter. These running times were measured on an Intel Pentium III, which has a 1 GHz clock and 256MB RAM under the WindowsME operating system. The programs were compiled using the C# compiler provided with the Microsoft .NET beta SDK (\texttt{csc}) and run under the Microsoft common language runtime. The times shown are elapsed CPU times, measured in seconds.

Figure shows the running times of the \( \mathcal{O}(n^2) \) sorts for sequences of length \( n, 100 \leq n \leq 20\,000 \). Notice that the bubble sort has the worst performance and that the binary insertion sort has the best performance. Figure clearly shows that, as predicted, binary insertion is better than straight insertion. Notice too that all of the \( \mathcal{O}(n^2) \) sorts require more than 25 seconds of execution time to sort an array of 20000 integers.
The performance of the $O(n^2)$ sorts is shown in Figure 1. In this case, the length of the sequence varies between $n=100$ and $n = 100\,000$. The graph clearly shows that the $O(n \log n)$ algorithms are significantly faster than the $O(n^2)$ ones. All three algorithms sort 100000 integers in under 2 seconds. Merge sort and quicksort display almost identical performance.
Figure: Actual running times of the $O(n \log n)$ sorts.

Figure shows the actual running times for the bucket sort and radix sort algorithms. Both these algorithms were shown to be $O(n)$ sorts. The graph shows results for $n$ between 100 and 1,000,000. The universe used to test bucket sort was $\{0, 1, \ldots, 1023\}$. That is, a total of $m=1024$ counters (buckets) were used. For the radix sort, 32-bit integers were sorted by using the radix $R=256$ and doing $p=4$ sorting passes.
Figure: Actual running times of the $O(n)$ sorts.

Clearly, the bucket sort has the better running time. For example, it sorts 500000 10-bit integers in under 2 seconds. Radix sort performs extremely well too. It sorts 500000 32-bit integers in just over 2 seconds, only slightly slower than the bucket sort.
Exercises

1. Consider the sequence of integers

   \[ S = \{8, 9, 7, 9, 3, 2, 3, 8, 4, 6\} \]

   For each of the following sorting algorithms, draw a sequence of diagrams that traces the execution of the algorithm as it sorts the sequence \( S \): straight insertion sort, binary insertion sort, bubble sort, quick sort, straight selection sort, heapsort, merge sort, and bucket sort.

2. Draw a sequence of diagrams that traces the execution of a radix-10 sort of the sequence

   \[ S = \{89, 79, 32, 38, 46, 26, 43, 38, 32, 79\} \]

3. For each of the sorting algorithms listed in Exercises and indicate whether the sorting algorithm is stable.

4. Consider a sequence of three distinct keys \( \{a, b, c\} \). Draw the binary decision tree that represents each of the following sorting algorithms: straight insertion sort, straight selection sort, and bubble sort.

5. Devise an algorithm to sort a sequence of exactly five elements. Make your algorithm as efficient as possible.

6. Prove that the swapping of a pair of adjacent elements removes at most one inversion from a sequence.

7. Consider the sequence of elements \( \{\bar{r}_1, \bar{r}_2, \ldots, \bar{r}_n\} \). What is the maximum number of inversions that can be removed by the swapping of a pair of distinct elements \( \bar{r}_i \) and \( \bar{r}_j \)? Express the result in terms of the distance between \( \bar{r}_i \) and \( \bar{r}_j \): \( d(\bar{r}_i, \bar{r}_j) = j - i + 1 \).

8. Devise a sequence of keys such that exactly eleven inversions are removed by the swapping of one pair of elements.

9. Prove that binary insertion sort requires \( O(n \log n) \) comparisons.

10. Consider an arbitrary sequence \( \{\bar{r}_1, \bar{r}_2, \ldots, \bar{r}_n\} \). To sort the sequence, we determine the permutation \( \{p_1, p_2, \ldots, p_n\} \) such that

    \[ \bar{r}_{p_1} \leq \bar{r}_{p_2} \leq \cdots \leq \bar{r}_{p_n} \]
Exercises

Prove that bubble sort requires at least \( p \) passes where

\[
p = \max_{1 \leq i \leq n} (i - p_i).
\]

11. Modify the bubble sort algorithm (Program 4) so that it terminates the outer loop when it detects that the array is sorted. What is the running time of the modified algorithm? **Hint:** See Exercise 4.

12. A variant of the bubble sorting algorithm is the so-called odd-even transposition sort. Like bubble sort, this algorithm a total of \( n-1 \) passes through the array. Each pass consists of two phases: The first phase compares \( array[i] \) with \( array[i + 1] \) and swaps them if necessary for all the odd values of \( i \). The second phase does the same for the even values of \( i \).

   1. Show that the array is guaranteed to be sorted after \( n-1 \) passes.
   2. What is the running time of this algorithm?

13. Another variant of the bubble sorting algorithm is the so-called cocktail shaker sort. Like bubble sort, this algorithm a total of \( n-1 \) passes through the array. However, alternating passes go in opposite directions. For example, during the first pass the largest item bubbles to the end of the array and during the second pass the smallest item bubbles to the beginning of the array.

   1. Show that the array is guaranteed to be sorted after \( n-1 \) passes.
   2. What is the running time of this algorithm?

14. Consider the following algorithm for selecting the \( k^{th} \) largest element from an unsorted sequence of of \( n \) elements, \( S = \{s_1, s_2, \ldots, s_n\} \).

   1. If \( n \leq 5 \), sort \( S \) and select directly the \( k^{th} \) largest element.
   2. Otherwise \( n > 5 \): Partition the sequence \( S \) into subsequences of length five. In general, there will be \( \lceil n/5 \rceil \) subsequences of length five and one of length \( n \mod 5 \).
   3. Sort by any means each of the subsequences of length five. (See Exercise 4).
   4. Form the sequence \( M = \{m_1, m_2, \ldots, m_{\lceil n/5 \rceil}\} \) containing the \( \lfloor n/5 \rfloor \) median values of each of the subsequences of length five.
   5. Apply the selection method recursively to find the median element of \( M \). Let \( m \) be the median of the medians.
   6. Partition \( S \) into three subsequences, \( S = \{L, E, G\} \) such that all the elements in \( L \) are less than \( m \), all the elements in \( E \) are equal to \( m \), and all the elements of \( G \) are greater than \( m \).
   7. If \( k \leq |L| \) then apply the method recursively to select the \( k^{th} \) largest element of \( L \); if \( |L| < k \leq |L| + |E| \), the result is \( m \); otherwise apply the method recursively to select the \( (k - (|L| + |E|) + 1)^{th} \) largest element of \( G \).

   1. What is the running time of this algorithm?
   2. Show that if we use this algorithm to select the pivot the worst-case running time of quick
Exercises

sort is \( O(n \log n) \).

15. Show that the sum of the heights of the nodes in a complete binary tree with \( n \) nodes altogether is \( n - b(n) \), where \( b(n) \) is the number of ones in the binary representation of \( n \).
Projects

1. Design and implement an algorithm that finds all the duplicates in a random sequence of keys.
2. Suppose we wish to sort a sequence of data represented using the linked-list class LinkedList introduced in Program 4. Which of the sorting algorithms described in this chapter is the most appropriate for sorting a linked list? Design and implement a linked list sorter class that implements this algorithm.
3. Replace the Sort method of the MergeSorter class with a non-recursive version. What is the running time of the non-recursive merge sort?
4. Replace the Sort method of the AbstractQuickSorter class with a non-recursive version. What is the running time of the non-recursive quick sort? **Hint:** Use a stack.
5. Design and implement a radix-sorter class that sorts an array of strings.
6. Design and implement a RandomPivotQuickSorter class that uses a random number generator (see Section 9) to select a pseudorandom pivot. Run a sequence of experiments to compare the running times of random pivot selection with median-of-three pivot selection.
7. Design and implement a MeanPivotQuickSorter class that partitions the sequence to be sorted into elements that are less than the mean and elements that are greater than the mean. Run a sequence of experiments to compare the running times of the mean pivot quick sorter with median-of-three pivot selection.
8. Design and implement a MedianPivotQuickSorter class that uses the algorithm given in Exercise 9 to select the median element for the pivot. Run a sequence of experiments to compare the running times of median pivot selection with median-of-three pivot selection.
9. Design and implement a sorter class that sorts using a PriorityQueue instance. (See Chapter 9).
Graphs and Graph Algorithms

A graph is simply a set of points together with a set of lines connecting various points. Myriad real-world application problems can be reduced to problems on graphs.

Suppose you are planning a trip by airplane. From a map you have determined the distances between the airports in the various cities that you wish to visit. The information you have gathered can be represented using a graph as shown in Figure (a). The points in the graph represent the cities and the lines represent the distances between them. Given such a graph, you can answer questions such as "What is the shortest distance between LAX and JFK?" or "What is the shortest route that visits all of the cities?"
Figure: Real-world examples of graphs.

An electric circuit can also be viewed as a graph as shown in Figure (b). In this case the points in the graph indicate where the components are connected (i.e., the wires) and the lines represent the components themselves (e.g., resistors and capacitors). Given such a graph, we can answer questions such as \"What are the mesh equations that describe the circuit's behavior?\"

Similarly, a logic circuit can be reduced to a graph as shown in Figure (c). In this case the logic gates are represented by the points and arrows represent the signal flows from gate outputs to gate inputs. Given such a graph, we can answer questions such as \"How long does it take for the signals to propagate from the inputs to the outputs?\" or \"Which gates are on the critical path?\"

Finally, Figure (d) illustrates that a graph can be used to represent a finite state machine. The points of the graph represent the states and labeled arrows indicate the allowable state transitions. Given such a graph, we can answer questions such as \"Are all the states reachable?\" or \"Can the finite state machine
This chapter is a brief introduction to the body of knowledge known as graph theory. It covers the most common data structures for the representation of graphs and introduces some fundamental graph algorithms.
Basics

- Directed Graphs
- Terminology
- More Terminology
- Directed Acyclic Graphs
- Undirected Graphs
- Terminology
- Labeled Graphs
- Representing Graphs

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Directed Graphs

We begin with the definition of a directed graph:

**Definition (Directed Graph)** A directed graph, or digraph, is an ordered pair $G = (V, E)$ with the following properties:

1. The first component, $V$, is a finite, non-empty set. The elements of $V$ are called the vertices of $G$.
2. The second component, $E$, is a finite set of ordered pairs of vertices. That is, $E \subseteq V \times V$. The elements of $E$ are called the edges of $G$.

For example, consider the directed graph $G_1 = (V_1, E_1)$ comprised of four vertices and six edges:

$$
V_1 = \{a, b, c, d\}, \\
E_1 = \{(a, b), (a, c), (b, c), (c, a), (c, d), (d, d)\}.
$$

The graph $G$ can be represented graphically as shown in Figure 1. The vertices are represented by appropriately labeled circles, and the edges are represented by arrows that connect associated vertices.

**Figure:** A directed graph.

Notice that because the pairs that represent edges are ordered, the two edges $(a,c)$ and $(c,a)$ are distinct. Furthermore, since $E_1$ is a mathematical set, it cannot contain more than one instance of a given edge. And finally, an edge such as $(d,d)$ may connect a node to itself.
Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Terminology

Consider a directed graph $G = (V, E)$ as given by Definition □.

- Each element of $V$ is called a vertex or a node of $G$. Hence, $V$ is the set of vertices (or nodes) of $G$.
- Each element of $E$ is called an edge or an arc of $G$. Hence, $E$ is the set of edges (or arcs) of $G$.
- An edge $(v, w) \in E$ can be represented as $v \rightarrow w$. An arrow that points from $v$ to $w$ is known as a directed arc. Vertex $w$ is called the head of the arc because it is found at the arrow head. Conversely, $v$ is called the tail of the arc. Finally, vertex $w$ is said to be adjacent to vertex $v$.
- An edge $e=(v,w)$ is said to emanate from vertex $v$. We use notation $A(v)$ to denote the set of edges emanating from vertex $v$. That is, $A(v) = \{(v_0, v_1) \in E : v_0 = v\}$.
- The out-degree of a node is the number of edges emanating from that node. Therefore, the out-degree of $v$ is $|A(v)|$.
- An edge $e=(v,w)$ is said to be incident on vertex $w$. We use notation $I(w)$ to denote the set of edges incident on vertex $w$. That is, $I(w) = \{(v_0, v_1) \in E : v_1 = w\}$.
- The in-degree of a node is the number of edges incident on that node. Therefore, the in-degree of $w$ is $|I(w)|$.

For example, Table □ enumerates the sets of emanating and incident edges and the in- and out-degrees for each of the vertices in graph $G_1$ shown in Figure □.

<table>
<thead>
<tr>
<th>vertex $v$</th>
<th>$A(v)$</th>
<th>out-degree</th>
<th>$I(v)$</th>
<th>in-degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>${(a, b), (a, c)}$</td>
<td>2</td>
<td>${(c, a)}$</td>
<td>1</td>
</tr>
<tr>
<td>$b$</td>
<td>${(b, c)}$</td>
<td>1</td>
<td>${(a, b)}$</td>
<td>1</td>
</tr>
</tbody>
</table>
There is still more terminology to be introduced, but in order to do that, we need the following definition:

**Definition (Path and Path Length)**

A path in a directed graph \( G = (V, E) \) is a non-empty sequence of vertices

\[
P = \{v_1, v_2, \ldots, v_k\},
\]

where \( v_i \in V \) for \( 1 \leq i \leq k \) such that \( (v_i, v_{i+1}) \in E \) for \( 1 \leq i < k \). The length of path \( P \) is \( k - 1 \).

For example, consider again the graph \( G_1 \) shown in Figure \( \square \). Among the paths contained in \( G_1 \) there is the path of length zero, \( \{a\} \); the path of length one, \( \{b, c\} \); the path of length two, \( \{a, b, c\} \); and so on. In fact, this graph generates an infinite number of paths! (To see how this is possible, consider that \( \{a, c, a, c, a, c, a, c, a, c, a\} \) is a path in \( G_1 \)). Notice too the subtle distinction between a path of length zero, say \( \{d\} \), and the path of length one \( \{d, d\} \).
More Terminology

Consider the path \( P = \{v_1, v_2, \ldots, v_k\} \) in a directed graph \( G = (V, E) \).

- Vertex \( v_{i+1} \) is the \textit{successor} of vertex \( v_i \) for \( 1 \leq i < k \). Each element \( v_i \) of path \( P \) (except the last) has a \textit{successor}.
- Vertex \( v_i-1 \) is the \textit{predecessor} of vertex \( v_i \) for \( 1 < i \leq k \). Each element \( v_i \) of path \( P \) (except the first) has a \textit{predecessor}.
- A path \( P \) is called a \textit{simple} path if and only if \( v_i \neq v_j \) for all \( i \) and \( j \) such that \( 1 \leq i < j \leq k \). However, it \textit{is} permissible for \( v_1 \) to be the same as \( v_k \) in a simple path.
- A \textit{cycle} is a path \( P \) of non-zero length in which \( v_1 = v_k \). The \textit{length of a cycle} is just the length of the path \( P \).
- A \textit{loop} is a cycle of length one. That is, it is a path of the form \( \{v, v\} \).
- A \textit{simple cycle} is a path that is both a cycle and simple.

Referring again to graph \( G' \) in Figure \( \) we find that the path \( \{a, b, c, d\} \) is a simple path of length three. Conversely, the path \( \{c, a, c, d\} \) also has length three but is not simple because vertex \( c \) occurs twice in the sequence (but not at the ends). The graph contains the path \( \{a, b, c, a\} \) which is a cycle of length three, as well as \( \{a, c, a, c, a\} \), a cycle of length four. The former is a simple cycle but the latter is not.
Directed Acyclic Graphs

For certain applications it is convenient to deal with graphs that contain no cycles. For example, a tree (see Chapter__) is a special kind of graph that contains no cycles.

**Definition (Directed Acyclic Graph (DAG))**

A directed, acyclic graph is a directed graph that contains no cycles.

Obviously, all trees are DAGs. However, not all DAGs are trees. For example consider the two directed, acyclic graphs, $G_2$ and $G_3$, shown in Figure__. Clearly $G_2$ is a tree but $G_3$ is not.

---

**Figure:** Two directed, acyclic graphs.

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Undirected Graphs

An undirected graph is a graph in which the nodes are connected by *undirected arcs*. An undirected arc is an edge that has no arrow. Both ends of an undirected arc are equivalent--there is no head or tail. Therefore, we represent an edge in an undirected graph as a set rather than an ordered pair:

**Definition (Undirected Graph)** An *undirected graph* is an ordered pair \( G = (V, E) \) with the following properties:

1. The first component, \( V \), is a finite, non-empty set. The elements of \( V \) are called the *vertices* of \( G \).
2. The second component, \( E \), is a finite set of sets. Each element of \( E \) is a set that is comprised of exactly two (distinct) vertices. The elements of \( E \) are called the *edges* of \( G \).

For example, consider the undirected graph \( G_4 = (V_4, E_4) \) comprised of four vertices and four edges:

\[
V_4 = \{a, b, c, d\} \\
E_4 = \{\{a, b\}, \{a, c\}, \{b, c\}, \{c, d\}\}
\]

The graph \( G_4 \) can be represented *graphically* as shown in Figure \[\square\]. The vertices are represented by appropriately labeled circles, and the edges are represented by lines that connect associated vertices.

![Graph G_4](image)

**Figure**: An undirected graph.

Notice that because an edge in an undirected graph is a set, \( \{a, b\} = \{b, a\} \), and since \( E_4 \) is also a set, it
Undirected Graphs

cannot contain more than one instance of a given edge. Another consequence of Definition is that there cannot be an edge from a node to itself in an undirected graph because an edge is a set of size two and a set cannot contain duplicates.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Terminology

Consider an undirected graph \( G = (V, E) \) as given by Definition □.

- An edge \( \{v, w\} \in E \) emanates from and is incident on both vertices \( v \) and \( w \).
- The set of edges emanating from a vertex \( v \) is the set \( \mathcal{A}(v) = \\{(v_0, v_1) \in E : v_0 = v \lor v_1 = v\} \). The set of edges incident on a vertex \( w \) is \( \mathcal{I}(w) = \mathcal{A}(w) \).
Labeled Graphs

Practical applications of graphs usually require that they be annotated with additional information. Such information may be attached to the edges of the graph and to the nodes of the graph. A graph which has been annotated in some way is called a *labeled graph*. Figure shows two examples of this.

For example, we can use a directed graph with labeled vertices such as $G_5$ in Figure to represent a finite state machine. Each vertex corresponds to a state of the machine and each edge corresponds to an allowable state transition. In such a graph we can attach a label to each vertex that records some property of the corresponding state such as the latency time for that state.

We can use an undirected graph with labeled edges such as $G_6$ in Figure to represent geographic information. In such a graph, the vertices represent geographic locations and the edges represent possible routes between locations. In such a graph we might use a label on each edge to represent the distance between the end points.
Representing Graphs

Consider a directed graph \( G = (V, \mathcal{E}) \). Since \( \mathcal{E} \subseteq V \times V \), graph \( G \) contains at most \(|V|^2\) edges. There are \(2|V|^2\) possible sets of edges for a given set of vertices \( V \). Therefore, the main concern when designing a graph representation scheme is to find a suitable way to represent the set of edges.

- **Adjacency Matrices**
- **Sparse vs. Dense Graphs**
- **Adjacency Lists**
### Adjacency Matrices

Consider a directed graph $G = (V, E)$ with $n$ vertices, $V = \{v_1, v_2, \ldots, v_n\}$. The simplest graph representation scheme uses an $n \times n$ matrix $A$ of zeroes and ones given by

$$A_{i,j} = \begin{cases} 1 & (v_i, v_j) \in E, \\ 0 & \text{otherwise}. \end{cases}$$

That is, the $(i,j)^{th}$ element of the matrix, is a one only if $v_i \rightarrow v_j$ is an edge in $G$. The matrix $A$ is called an adjacency matrix.

For example, the adjacency matrix for graph $G_1$ in Figure is

$$A_1 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Clearly, the number of ones in the adjacency matrix is equal to the number of edges in the graph.

One advantage of using an adjacency matrix is that it is easy to determine the sets of edges emanating from a given vertex. For example, consider vertex $v_i$. Each one in the $i^{th}$ row corresponds to an edge that emanates from vertex $v_i$. Conversely, each one in the $i^{th}$ column corresponds to an edge incident on vertex $v_i$.

We can also use adjacency matrices to represent undirected graphs. That is, we represent an undirected graph $G = (V, E)$ with $n$ vertices, using an $n \times n$ matrix $A$ of zeroes and ones given by

$$A_{i,j} = \begin{cases} 1 & \{v_i, v_j\} \in E, \\ 0 & \text{otherwise}. \end{cases}$$
Since the two sets $\{v_i, v_j\}$ and $\{v_j, v_i\}$ are equivalent, matrix $A$ is symmetric about the diagonal. That is, $A_{i,j} = A_{j,i}$. Furthermore, all of the entries on the diagonal are zero. That is, $A_{i,i} = 0$ for $1 \leq i \leq n$.

For example, the adjacency matrix for graph $G_4$ in Figure 1 is

$$
A_4 = \begin{bmatrix}
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 \\
0 & 0 & 1 & 0
\end{bmatrix}
$$

In this case, there are twice as many ones in the adjacency matrix as there are edges in the undirected graph.

A simple variation allows us to use an adjacency matrix to represent an edge-labeled graph. For example, given numeric edge labels, we can represent a graph (directed or undirected) using an $n \times n$ matrix $A$ in which the $A_{i,j}$ is the numeric label associated with edge $\{v_i, v_j\}$ in the case of a directed graph, and edge $\{v_i, v_j\}$, in an undirected graph.

For example, the adjacency matrix for the graph $G_6$ in Figure 2 is

$$
A_6 = \begin{bmatrix}
\infty & 31 & 41 & \infty \\
31 & \infty & 59 & \infty \\
41 & 59 & \infty & 26 \\
\infty & \infty & 26 & \infty
\end{bmatrix}
$$

In this case, the array entries corresponding to non-existent edges have all been set to $\infty$. Here $\infty$ serves as a kind of sentinel. The value to use for the sentinel depends on the application. For example, if the edges represent routes between geographic locations, then a route of length $\infty$ is much like one that does not exist.

Since the adjacency matrix has $|V|^2$ entries, the amount of space needed to represent the edges of a graph is $O(|V|^2)$, regardless of the actual number of edges in the graph. If the graph contains relatively few edges, e.g., if $|E| \ll |V|^2$, then most of the elements of the adjacency matrix will be zero (or $\infty$). A matrix in which most of the elements are zero (or $\infty$) is a sparse matrix.
Sparse vs. Dense Graphs

Informally, a graph with relatively few edges is *sparse*, and a graph with many edges is *dense*. The following definition defines precisely what we mean when we say that a graph "has relatively few edges":

**Definition (Sparse Graph)** A *sparse graph* is a graph $G = (V, E)$ in which $|E| = O(|V|)$.

For example, consider a graph $G = (V, E)$ with $n$ nodes. Suppose that the out-degree of each vertex in $G$ is some fixed constant $k$. Graph $G$ is a *sparse graph* because $|E| = k|V| = O(|V|)$.

A graph that is not sparse is said to be *dense*:

**Definition (Dense Graph)** A *dense graph* is a graph $G = (V, E)$ in which $|E| = \Theta(|V|^2)$.

For example, consider a graph $G = (V, E)$ with $n$ nodes. Suppose that the out-degree of each vertex in $G$ is some fraction $f$ of $n$, $0 < f \leq 1$. For example, if $n=16$ and $f=0.25$, the out-degree of each node is 4. Graph $G$ is a *dense graph* because $|E| = f|V|^2 = \Theta(|V|^2)$.
Adjacency Lists

One technique that is often used for a sparse graph, say $G = (V, E)$, uses $|V|$ linked lists—one for each vertex. The linked list for vertex $v_i \in V$ contains the elements of $\{w : (v_i, w) \in A(v_i)\}$, the set of nodes adjacent to $v_i$. As a result, the lists are called adjacency lists.

Figure shows the adjacency lists for the directed graph $G_1$ of Figure and the directed graph $G_4$ of Figure. Notice that the total number of list elements used to represent a directed graph is $|E|$ but the number of lists elements used to represent an undirected graph is $2 \times |E|$. Therefore, the space required for the adjacency lists is $O(|E|)$. 

---

By definition, a sparse graph has $|E| = \mathcal{O}(|V|)$. Hence the space required to represent a sparse graph using adjacency lists is $\mathcal{O}(|V|)$. Clearly this is asymptotically better than using adjacency matrices which require $\mathcal{O}(|V|^2)$ space.
Implementing Graphs

In keeping with the design framework used throughout this text, we view graphs as specialized containers. Formally, the graph $G = (V, E)$ is an ordered pair comprised of two sets—a set of vertices and a set of edges. Informally, we can view a graph as a container with two compartments, one which holds vertices and one which holds edges. There are four kinds of objects—vertices, edges, undirected graphs, and directed graphs. Accordingly, we define four interfaces: Vertex, Edge, Graph, and Digraph. (See Figure).

**Figure:** Object class hierarchy

- Vertices
- Edges
- Graphs and Digraphs
- Directed Graphs
- Abstract Graphs
- Implementing Undirected Graphs
Vertices

What exactly is a vertex? The answer to this question depends on the application. At the very minimum, every vertex in a graph must be distinguishable from every other vertex in that graph. We can do this by numbering consecutively the vertices of a graph. In addition, some applications require vertex-weighted graphs. A weighted vertex can be viewed as one which carries a "payload". The payload is an object that represents the weight on the vertex.

Program defines the Vertex interface. The Vertex interface extends the IComparable interface.

```csharp
public interface Vertex : IComparable
{
    int Number { get; }
    object Weight { get; }
    IEnumerable IncidentEdges { get; }
    IEnumerable EmittingEdges { get; }
    IEnumerable Predecessors { get; }
    IEnumerable Successors { get; }
}
```

Program: Vertex interface.

Every vertex in a graph is assigned a unique number. The Number property provides a get accessor that returns the number of a vertex. The Weight property provides a get accessor that returns an object that represents the weight associated with a weighted vertex. If the vertex is is unweighted, the Weight property is returns null.

- Enumerators
Enumerators

Program also declares four `IEnumerable` properties. The `IncidentEdges` property can be used to enumerate the elements of the $\mathcal{I}(v)$; the `EmanatingEdges` property can be used to enumerate the elements of $\mathcal{A}(v)$. Similarly, the `Predecessors` property can be used to enumerate the elements of $\mathcal{P}(v) = \{u : (u, v) \in \mathcal{I}(v)\}$ and the `Successors` property can be used to enumerate the elements of $\mathcal{S}(v) = \{w : (v, w) \in \mathcal{A}(v)\}$. The elements of $\mathcal{P}(v)$ and $\mathcal{S}(v)$ are vertices whereas the elements of $\mathcal{I}(v)$ and $\mathcal{A}(v)$ are edges.
Edges

An edge in a directed graph is an ordered pair of vertices; an edge in an undirected graph is a set of two vertices. Because of the similarity of these concepts, we use the same class for both—the context in which an edge is used determines whether it is directed or undirected.

Program defines the Edge interface. The Edge interface extends the IComparable interface.

```csharp
public interface Edge : IComparable
{
    Vertex V0 { get; }
    Vertex V1 { get; }
    object Weight { get; }
    bool IsDirected { get; }
    Vertex MateOf(Vertex vertex);
}
```

Program: Edge interface.

An edge connects two vertices, \( v_0 \) and \( v_1 \). The properties \( V0 \) and \( V1 \) provide get accessors that return these vertices. The IsDirected property is a bool-valued accessor that returns true if the edge is directed. When an Edge is directed, it represents \( v_0 \rightarrow v_1 \). That is, \( v_1 \) is the head and \( v_0 \) is the tail. Alternatively, when an Edge is undirected, it represents \( \{v_0, v_1\} \).

For every instance \( e \) of a class that implements the Edge interface, the Mate property satisfies the following identities:

\[
\begin{align*}
    e.\text{Mate}(e.V0) & \equiv e.V1 \\
    e.\text{Mate}(e.V1) & \equiv e.V0.
\end{align*}
\]

Therefore, if we know that a vertex \( v \) is one of the vertices of \( e \), then we can find the other vertex by calling \( e.\text{Mate}(v) \).
Graphs and Digraphs

Directed graphs and undirected graphs have many common characteristics. In addition, we can view a directed graph as an undirected graph with arrowheads added. As shown in Figure, we have chosen to define the Graph interface to represent undirected graphs and to derive Digraph interface from it. We have chosen this approach because many algorithms for undirected graphs can also be used with directed graphs. On the other hand, it is often the case that algorithms for directed graphs cannot be used with undirected graphs.

Program defines the Graph interface. The Graph interface extends the Container interface defined in Program.

```csharp
1 public interface Graph : Container
2 {
3     int NumberOfEdges { get; }
4     int NumberOfVertices { get; }
5     bool IsDirected { get; }
6     void AddVertex(int v);
7     void AddVertex(int v, object weight);
8     Vertex GetVertex(int v);
9     void AddEdge(int v, int w);
10    void AddEdge(int v, int w, object weight);
11   Edge GetEdge(int v, int w);
12   bool IsEdge(int v, int w);
13   bool IsConnected { get; }
14   bool IsCyclic { get; }
15   IEnumerable<Vertices> { get; }
16   IEnumerable<Edges> { get; }
17   void DepthFirstTraversal(PrePostVisitor visitor, int start);
18   void BreadthFirstTraversal(Visitor visitor, int start);
19 }
```

Program: Graph interface.
There are essentially three groups of methods declared in Program 4: accessors and mutators, enumerators, and traversals. The operations performed by the methods are explained in the following sections.

- **Accessors and Mutators**
- **Enumerators**
- **Graph Traversals**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Accessors and Mutators

The Graph interface declares the following accessor properties and mutator methods:

**NumberOfEdges**
This property provides a get accessor that returns the number of edges contained by the graph.

**NumberOfVertices**
This property provides a get accessor that returns the number of vertices contained by the graph.

**IsDirected**
This bool-valued property is true if the graph is a directed graph.

**AddVertex**
This method inserts a vertex into a graph. All the vertices contained in a given graph must have a unique vertex number. Furthermore, if a graph contains \( n \) vertices, those vertices shall be numbered 0, 1, ..., \( n-1 \). Therefore, the next vertex inserted into the graph shall have the number \( n \).

**GetVertex**
This method takes an integer, say \( i \) where \( 0 \leq i < n \), and returns the \( i^{th} \) vertex contained in the graph.

**AddEdge**
This method inserts an edge into a graph. If the graph contains \( n \) vertices, both arguments must fall in the interval \([0, n-1]\).

**IsEdge**
This bool-valued method takes two integer arguments. It returns true if the graph contains an edge that connects the corresponding vertices.

**GetEdge**
This method takes two integer arguments. It returns the edge instance (if it exists) that connects the corresponding vertices. The behavior of this method is undefined when the edge does not exist. (An implementation will typically throw an exception).

**IsCyclic**
This bool-valued property is true if the graph is cyclic.

**IsConnected**
This bool-valued property is true if the graph is connected. Connectedness of graphs is discussed in Section □.
Enumerators

All the other container classes considered in this text have only one associated enumerator. When dealing with graphs, it is convenient to have two enumerators. The following methods each returns an IEnumerable object as follows:

**Vertices**
This property can be used to enumerate the elements of $\mathcal{V}$.

**Edges**
This property can be used to enumerate the elements of $\mathcal{E}$.
Graph Traversals

The following traversal methods are analogous to the `Accept` method of the container class (see Section □). Each of these methods takes a `visitor` and performs a traversal. That is, all the `vertices` of the graph are visited systematically. When a vertex is visited, the `Visit` method of the visitor is applied to that vertex.

**DepthFirstTraversal**

This methods accepts two arguments--a `PrePostVisitor` and an integer. The integer specifies the starting vertex for a depth-first traversal of the graph.

**BreadthFirstTraversal**

This methods accepts two arguments--a `Visitor` and an integer. The integer specifies the starting vertex for a breadth-first traversal of the graph.

Graph traversal algorithms are discussed in Section □.
Directed Graphs

Program [square_root] defines the Digraph interface. The Digraph interface extends the Graph interface defined in Program [square_root].

```csharp
public interface Digraph : Graph
{
    bool IsStronglyConnected { get; }
    void TopologicalOrderTraversal(Visitor visitor);
}
```

Program: Digraph interface.

The Digraph interface adds the following operations which apply only to directed graphs to the inherited interface:

**IsStronglyConnected**
This bool-valued property is true if the directed graph is strongly connected. Strong connectedness is discussed in Section [square_root].

**TopologicalOrderTraversal**
A topological sort is an ordering of the nodes of a directed graph. This traversal visits the nodes of a directed graph in the order specified by a topological sort.
Abstract Graphs

Program 1 introduces the AbstractGraph class. The AbstractGraph class extends the AbstractContainer class introduced in Program 9 and it implements the Graph interface defined in Program 9.

```csharp
public abstract class AbstractGraph : AbstractContainer, Graph
{
    protected int numberOfVertices;
    protected int numberOfEdges;
    protected Vertex[] vertex;

    public AbstractGraph(int size)
    {
        vertex = new Vertex[size];
    }

    protected class GraphVertex : ComparableObject, Vertex
    {
        protected AbstractGraph graph;
        protected int number;
        protected object weight;
        // ...
    }

    protected class GraphEdge : ComparableObject, Edge
    {
        protected AbstractGraph graph;
        protected int v0;
        protected int v1;
        protected object weight;
        // ...
    }

    protected abstract IEnumerable GetIncidentEdges(int v);
    protected abstract IEnumerable GetEmanatingEdges(int v);
```
The AbstractGraph class serves as the base class from which the various concrete graph implementations discussed in Section 4 are derived. The AbstractGraph class also provides implementations for the graph traversals described in Section 4 and for the algorithms that test for cycles and connectedness described in Section 5.

As shown in Program 4, the AbstractGraph class defines two nested classes, GraphVertex and GraphEdge. Both classes extend the ComparableObject class introduced in Program 0.

The GraphVertex class implements the Vertex interface. It comprises three fields—graph, number and weight. The graph field refers to the graph instance that contains this vertex. Each vertex in a graph with n vertices is assigned a unique number in the interval [0, n-1]. The number field records this number. The weight field is used to record the weight on a weighted vertex.

The GraphEdge class implements the Edge interface. It comprises four fields—graph, v0, v1, and weight. The graph field refers to the graph instance that contains this edge. The v0 and v1 record the vertices that are the end-points of the edge. The weight field is used to record the weight on a weighted edge.
Implementing Undirected Graphs

This section describes two concrete classes--GraphAsMatrix and GraphAsLists. These classes both represent undirected graphs. The GraphAsMatrix class represents the edges of a graph using an adjacency matrix. The GraphAsLists class represents the edges of a graph using adjacency lists.

- Using Adjacency Matrices
- Using Adjacency Lists

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Using Adjacency Matrices

The GraphAsMatrix is introduced in Program 4. The GraphAsMatrix class extends the AbstractGraph class introduced in Program 1.

```csharp
public class GraphAsMatrix : AbstractGraph
{
    protected Edge[,] matrix;

    public GraphAsMatrix(int size) : base(size)
    {
        matrix = new Edge[size, size];
    }
    // ...
}
```

Program: GraphAsMatrix fields and constructor.

Each instance of the GraphAsMatrix class represents an undirected graph, say $G = (\mathcal{V}; \mathcal{E})$. The set of vertices, $\mathcal{V}$, is represented using the vertex array inherited from the AbstractGraph base class. Each vertex is represented by a separate GraphVertex instance.

Similarly, The set of edges, $\mathcal{E}$, is represented using the matrix field which is a two-dimensional array of Edges. Each edge is represented by a separate GraphEdge instance.

The GraphAsMatrix constructor takes a single argument of type int that specifies the maximum number of vertices that the graph may contain. This quantity specifies the length of the array of vertices and the dimensions of the adjacency matrix. The implementation of the GraphAsMatrix class is left as programming project for the reader (Project 4).
Using Adjacency Lists

Program introduces the GraphAsLists class. The GraphAsLists extends the AbstractGraph class introduced in Program. The GraphAsLists class represents the edges of a graph using adjacency lists.

```csharp
public class GraphAsLists : AbstractGraph
{
    protected LinkedList[] adjacencyList;
    public GraphAsLists(int size) : base(size)
    {
        adjacencyList = new LinkedList[size];
        for (int i = 0; i < size; ++i)
            adjacencyList[i] = new LinkedList();
        // ...
    }
}
```

Program: GraphAsLists fields and constructor.

Each instance of the GraphAsLists class represents an undirected graph, say $G = (V, E)$. The set of vertices, $V$, is represented using the vertex array inherited from the AbstractGraph base class. The set of edges, $E$, is represented using the adjacencyList field, which is an array of linked lists. The $i^{th}$ linked list, adjacencyList[i], represents the set $A(v_i)$ which is the set of edges emanating from vertex $v_i$. The implementation uses the LinkedList class given in Section.

The GraphAsLists constructor takes a single argument of type int that specifies the maximum number of vertices that the graph may contain. This quantity specifies the lengths of the array of vertices and the array of adjacency lists. The implementation of the GraphAsLists class is left as programming project for the reader (Project).
Comparison of Graph Representations

In order to make the appropriate choice when selecting a graph representation scheme, it is necessary to understand the time/space trade-offs. Although the details of the implementations have been omitted, we can still make meaningful conclusions about the performance that we can expect from those implementations. In this section we consider the space required as well as the running times for basic graph operations.

- **Space Comparison**
- **Time Comparison**
Space Comparison

Consider the representation of a directed graph \( G = (V, E) \). In addition to the \(|V|\) GraphVertex class instances and the \(|E|\) GraphEdge class instances contained by the graph, there is the storage required by the adjacency matrix. In this case, the matrix is a \(|V| \times |V|\) matrix of Edges. Therefore, the amount of storage required by an adjacency matrix implementation is

\[
|V| \times \text{sizeof(GraphVertex)} + |E| \times \text{sizeof(GraphEdge)}
\]

\[
+ |V| \times \text{sizeof(Vertex ref)} + |V|^2 \times \text{sizeof(Edge ref)} + O(1).
\]

On the other hand, consider the amount of storage required when we represent the same graph using adjacency lists. In addition to the vertices and the edges themselves, there are \(|V|\) linked lists. If we use the LinkedList class defined in Section \( \Box \), each such list has a head and tail field. Altogether there are \(|E|\) linked lists elements each of which refers to the linked list itself, to the next element of the list and contains an Edge. Therefore, the total space required is

\[
|V| \times \text{sizeof(GraphVertex)} + |E| \times \text{sizeof(GraphEdge)}
\]

\[
+ |V| \times \text{sizeof(Vertex ref)} + |V| \times \text{sizeof(LinkedList ref)}
\]

\[
+ 2|V| \times \text{sizeof(LinkedList.Element ref)} +
\]

\[
+ |E| \times (\text{sizeof(LinkedList ref)} + \text{sizeof(LinkedList.Element ref)} + \text{sizeof(Edge ref)}) + O(1).
\]

Notice that the space for the vertices and edges themselves cancels out when we compare Equation \( \Box \) with Equation \( \Box \). If we assume that all object references require the same amount of space, we can conclude that adjacency lists use less space than adjacency matrices when

\[
|E| < \frac{|V|^2 - |V|}{3}.
\]

For example, given a 10 node graph, the adjacency lists version uses less space when there are fewer than
30 edges. As a rough rule of thumb, we can say that adjacency lists use less space when the average degree of a node, \( \bar{d} = |\mathcal{E}|/|\mathcal{V}| \), satisfies \( \bar{d} \lesssim |\mathcal{V}|/3 \).
Time Comparison

The following four operations are used extensively in the implementations of many different graph algorithms:

**find edge \((v,w)\)**

Given vertices \(v\) and \(w\), this operation locates the corresponding \(Edge\) instance. When using an adjacency matrix, we can find an edge in constant time.

When adjacency lists are used, the worst-case running time is \(O(|\mathcal{A}(v)|)\), since \(|\mathcal{A}(v)|\) is the length of the adjacency list associated with vertex \(v\).

This is the operation performed by the \(GetEdge\) method of the \(Graph\) interface.

**enumerate all edges**

In order to locate all the edges in when using adjacency matrices, it is necessary to examine all \(|V|^2\) matrix entries. Therefore, the worst-case running time needed to enumerate all the edges is \(O(|V|^2)\).

On the other hand, to enumerate all the edges when using adjacency lists requires the traversal of \(|V|\) lists. In all there are \(|E|\) edges. Therefore the worst case running time is \(O(|V| + |E|)\).

This operation is performed using the enumerator obtained using the \(Edges\) property of the \(Graph\) interface.

**enumerate edges emanating from \(v\)**

To enumerate all the edges that emanate from vertex \(v\) requires a complete scan of the \(v^{th}\) row of an adjacency matrix. Therefore, the worst-case running time when using adjacency matrices is \(O(|V|)\).

Enumerating the edges emanating from vertex \(v\) is a trivial operation when using adjacency lists. All we need do is traverse the \(v^{th}\) list. This takes \(O(|\mathcal{A}(v)|)\) time in the worst case.
This operation is performed using the enumerator obtained using the `EmanatingEdges` property of the `Vertex` interface.

**enumerate edges incident on w**

To enumerate all the edges are incident on vertex \( w \) requires a complete scan of the \( u^{th} \) column of an adjacency matrix. Therefore, the worst-case running time when using adjacency matrices is \( \mathcal{O}(|\mathcal{V}|) \).

Enumerating the edges incident on vertex \( w \) is a non-trivial operation when using adjacency lists. It is necessary to search every adjacency list in order to find all the edges incident on a given vertex. Therefore, the worst-case running time is \( \mathcal{O}(|\mathcal{V}| + |\mathcal{E}|) \).

This operation is performed using the enumerator obtained using the `IncidentEdges` property of the `Vertex` interface.

Table 1 summarizes these running times.

<table>
<thead>
<tr>
<th>operation</th>
<th>representation scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>adjacency matrix</td>
</tr>
<tr>
<td>find edge ((v, w))</td>
<td>( \mathcal{O}(1) )</td>
</tr>
<tr>
<td>enumerate all edges</td>
<td>( \mathcal{O}(</td>
</tr>
<tr>
<td>enumerate edges emanating from ( v )</td>
<td>( \mathcal{O}(</td>
</tr>
<tr>
<td>enumerate edges incident on ( w )</td>
<td>( \mathcal{O}(</td>
</tr>
</tbody>
</table>

**Table:** Comparison of graph representations.
Graph Traversals

There are many different applications of graphs. As a result, there are many different algorithms for manipulating them. However, many of the different graph algorithms have in common the characteristic that they systematically visit all the vertices in the graph. That is, the algorithm walks through the graph data structure and performs some computation at each vertex in the graph. This process of walking through the graph is called a graph traversal.

While there are many different possible ways in which to systematically visit all the vertices of a graph, certain traversal methods occur frequently enough that they are given names of their own. This section presents three of them--depth-first traversal, breadth-first traversal and topological sort.

- Depth-First Traversal
- Breadth-First Traversal
- Topological Sort
- Graph Traversal Applications:
  Testing for Cycles and Connectedness

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Depth-First Traversal

The *depth-first traversal* of a graph is like the depth-first traversal of a tree discussed in Section □. A depth-first traversal of a tree always starts at the root of the tree. Since a graph has no root, when we do a depth-first traversal, we must specify the vertex at which to begin.

A depth-first traversal of a tree visits a node and then recursively visits the subtrees of that node. Similarly, depth-first traversal of a graph visits a vertex and then recursively visits all the vertices adjacent to that node. The catch is that the graph may contain cycles, but the traversal must visit every vertex at most once. The solution to the problem is to keep track of the nodes that have been visited, so that the traversal does not suffer the fate of infinite recursion.

**Figure:** Depth-first traversal.

For example, Figure □ illustrates the depth-first traversal of the directed graph $G_1$ starting from vertex $c$. The depth-first traversal visits the nodes in the order

$$c, a, b, d.$$  

A depth-first traversal only follows edges that lead to unvisited vertices. As shown in Figure □, if we omit the edges that are not followed, the remaining edges form a tree. Clearly, the depth-first traversal of this tree is equivalent to the depth-first traversal of the graph.
Depth-First Traversal

- Implementation
- Running Time Analysis

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Implementation

Program □ gives the code for the DepthFirstTraversal method of the AbstractGraph class. In fact, two DepthFirstTraversal methods are defined. One of them accepts two arguments; the other, three. As indicated in Program □, the two-argument method is declared public whereas the three-argument one is protected.

The user of the Graph interface only sees the two-argument DepthFirstTraversal method. This method takes any PrePostVisitor and an integer. The idea is that the Visit method of the visitor is called once for each vertex in the graph and the vertices are visited in depth-first traversal order starting from the vertex specified by the integer.
public abstract class AbstractGraph : AbstractContainer, Graph
{
    protected int numberOfVertices;
    protected int numberOfEdges;
    protected Vertex[] vertex;

    public virtual void DepthFirstTraversal(
        PrePostVisitor visitor, int start)
    {
        bool[] visited = new bool[numberOfVertices];
        for (int v = 0; v < numberOfVertices; ++v)
            visited[v] = false;
        DepthFirstTraversal(visitor, vertex[start], visited);
    }

    protected virtual void DepthFirstTraversal(
        PrePostVisitor visitor, Vertex v, bool[] visited)
    {
        if (visitor.IsDone)
            return;
        visitor.PreVisit(v);
        visited[v.Number] = true;
        foreach (Vertex to in v.Successors)
        {
            if (!visited[to.Number])
                DepthFirstTraversal(visitor, to, visited);
        }
        visitor.PostVisit(v);
    }
}

Program: AbstractGraph class DepthFirstTraversal method.

In order to ensure that each vertex is visited at most once, an array of length $|\mathcal{V}|$ of bool values called visited is used (line 10). That is, $\text{visited}[i] = \text{true}$ only if vertex $i$ has been visited. All the array elements are initially false (lines 11-12). After initializing the array, the two-argument method calls the three-argument one, passing it the array as the third argument.
The three-argument method returns immediately if the visitor is done. Otherwise, it visits the specified node, and then it follows all the edges emanating from that node and recursively visits the adjacent vertices if those vertices have not already been visited.
Running Time Analysis

The running time of the depth-first traversal method depends on the graph representation scheme used. The traversal visits each node in the graph at most once. When a node is visited, all the edges emanating from that node are considered. During a complete traversal enumerates every edge in the graph.

Therefore, the worst-case running time for the depth-first traversal of a graph is represented using an adjacency matrix is

$$|V| \times (T_{\text{PreVisit}} + T_{\text{PostVisit}}) + O(|V|^2).$$

When adjacency lists are used, the worst case running time for the depth-first traversal method is

$$|V| \times (T_{\text{PreVisit}} + T_{\text{PostVisit}}) + O(|V| + |E|).$$

Recall that for a sparse graph $|E| = O(|V|)$. If the sparse graph is represented using adjacency lists and if $T_{\text{PreVisit}} = O(1)$ and $T_{\text{PostVisit}} = O(1)$ the worst-case running time of the depth-first traversal is simply $O(|V|)$.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Breadth-First Traversal

The *breadth-first traversal* of a graph is like the breadth-first traversal of a tree discussed in Section □. The breadth-first traversal of a tree visits the nodes in the order of their depth in the tree. Breadth-first tree traversal first visits all the nodes at depth zero (i.e., the root), then all the nodes at depth one, and so on.

Since a graph has no root, when we do a breadth-first traversal, we must specify the vertex at which to start the traversal. Furthermore, we can define the depth of a given vertex to be the length of the shortest path from the starting vertex to the given vertex. Thus, breadth-first traversal first visits the starting vertex, then all the vertices adjacent to the starting vertex, and then all the vertices adjacent to those, and so on.

Section □ presents a non-recursive breadth-first traversal algorithm for $N$-ary trees that uses a queue to keep track vertices that need to be visited. The breadth-first graph traversal algorithm is very similar.

First, the starting vertex is enqueued. Then, the following steps are repeated until the queue is empty:

1. Remove the vertex at the head of the queue and call it $v$.
2. Visit $v$.
3. Follow each edge emanating from $v$ to find the adjacent vertex and call it $t$. If $t$ has not already been put into the queue, enqueue it.

Notice that a vertex can be put into the queue at most once. Therefore, the algorithm must somehow keep track of the vertices that have been enqueued.
Breadth-First Traversal

Figure: Breadth-first traversal.

Figure illustrates the breadth-first traversal of the directed graph $G_1$ starting from vertex $a$. The algorithm begins by inserting the starting vertex, $a$, into the empty queue. Next, the head of the queue (vertex $a$) is dequeued and visited, and the vertices adjacent to it (vertices $b$ and $c$) are enqueued. When, $b$ is dequeued and visited we find that there is only adjacent vertex, $c$, and that vertex is already in the queue. Next vertex $c$ is dequeued and visited. Vertex $c$ is adjacent to $a$ and $d$. Since $a$ has already been enqueued (and subsequently dequeued) only vertex $d$ is put into the queue. Finally, vertex $d$ is dequeued and visited. Therefore, the breadth-first traversal of $G_1$ starting from $a$ visits the vertices in the sequence $a$, $b$, $c$, $d$.

- Implementation
- Running Time Analysis
Implementation

Program □ gives the code for the BreadthFirstTraversal method of the AbstractGraph class. This method takes any Visitor and an integer. The Visit method of the visitor is called once for each vertex in the graph and the vertices are visited in breadth-first traversal order starting from the vertex specified by the integer.

```csharp
public abstract class AbstractGraph : AbstractContainer, Graph
{
    protected int numberOfVertices;
    protected int numberOfEdges;
    protected Vertex[] vertex;

    public void BreadthFirstTraversal(
        Visitor visitor, int start)
    {
        bool[] enqueued = new bool[numberOfVertices];
        for (int v = 0; v < numberOfVertices; ++v)
            enqueued[v] = false;

        Queue queue = new QueueAsLinkedList();

        queue.Enqueue(vertex[start]);
        enqueued[start] = true;
        while (!queue.IsEmpty && !visitor.IsDone)
        {
            Vertex v = (Vertex)queue.Dequeue();
            visitor.Visit(v);
            foreach (Vertex to in v.Successors)
            {
                if (!enqueued[to.Number])
                {
                    queue.Enqueue(to);
                    enqueued[to.Number] = true;
                }
            }
        }
    }
```
Implementation

```c
    } // ... 

Program: AbstractGraph class BreadthFirstTraversal method.

A bool-valued array, enqueued, is used to keep track of the vertices that have been put into the queue. The elements of the array are all initialized to false (lines 10-12). Next, a new queue is created and the starting vertex is enqueued (lines 14-17).

The main loop of the BreadthFirstTraversal method comprises lines 18-30. This loop continues as long as there is a vertex in the queue and the visitor is willing to do more work (line 18). In each iteration exactly one vertex is dequeued and visited (lines 20-21). After a vertex is visited, all the successors of that node are examined (lines 22-29). Every successor of the node that has not yet been enqueued is put into the queue and the fact that it has been enqueued is recorded in the array enqueued (lines 24-28).
```
Running Time Analysis

The breadth-first traversal enqueues each node in the graph at most once. When a node is dequeued, all the edges emanating from that node are considered. Therefore, a complete traversal enumerates every edge in the graph.

The actual running time of the breadth-first traversal method depends on the graph representation scheme used. The worst-case running time for the traversal of a graph represented using an adjacency matrix is

\[ |\mathcal{V}| \times T(\text{visit}) + O(|\mathcal{V}|^2). \]

When adjacency lists are used, the worst case running time for the breadth-first traversal method is

\[ |\mathcal{V}| \times T(\text{visit}) + O(|\mathcal{V}| + |\mathcal{E}|). \]

If the graph is sparse, then \( |\mathcal{E}| = O(|\mathcal{V}|) \). Therefore, if a sparse graph is represented using adjacency lists and if \( T(\text{visit}) = O(1) \), the worst-case running time of the breadth-first traversal is just \( O(|\mathcal{V}|) \).
Topological Sort

A topological sort is an ordering of the vertices of a directed acyclic graph given by the following definition:

**Definition (Topological Sort)** Consider a directed acyclic graph \( G = (V, E) \). A topological sort of the vertices of \( G \) is a sequence \( S = \{v_1, v_2, \ldots, v_{|V|}\} \) in which each element of \( V \) appears exactly once. For every pair of distinct vertices \( v_i \) and \( v_j \) in the sequence \( S \), if \( v_i \rightarrow v_j \) is an edge in \( G \), i.e., \( (v_i, v_j) \in E \), then \( i < j \).

Informally, a topological sort is a list of the vertices of a DAG in which all the successors of any given vertex appear in the sequence after that vertex. Consider the directed acyclic graph \( G_1 \) shown in Figure 1. The sequence \( S = \{a, b, c, d, e, f, g, h, i\} \) is a topological sort of the vertices of \( G_1 \). To see that this is so, consider the set of vertices:

\[
E = \{(a, b), (a, c), (a, e), (b, d), (b, e), (c, f), (c, h), (d, g), (e, g), (e, h), (e, i), (f, h), (g, i), (h, i)\}.
\]

The vertices in each edge are in alphabetical order, and so is the sequence \( S \).
It should also be evident from Figure that a topological sort is not unique. For example, the following are also valid topological sorts of the graph $G_T$:

- $S' = \{a, c, b, f, e, d, h, g, i\}$
- $S'' = \{a, b, d, e, g, c, f, h, i\}$
- $S''' = \{a, c, f, h, b, e, d, g, i\}$
- ...

One way to find a topological sort is to consider the in-degrees of the vertices. (The number above a vertex in Figure is the in-degree of that vertex). Clearly the first vertex in a topological sort must have in-degree zero and every DAG must contain at least one vertex with in-degree zero. A simple algorithm to create the sort goes like this:

Repeat the following steps until the graph is empty:

1. Select a vertex that has in-degree zero.
2. Add the vertex to the sort.
3. Delete the vertex and all the edges emanating from it from the graph.
Implementation

Instead of implementing an algorithm that computes a topological sort, we have chosen to implement a traversal that visits the vertices of a DAG in the order given by the topological sort. The topological order traversal can be used to implement many other graph algorithms. Furthermore, given such a traversal, it is easy to define a visitor that computes a topological sort.

In order to implement the algorithm described in the preceding section, an array of integers of length $|V|$ is used to record the in-degrees of the vertices. As a result, it is not really necessary to remove vertices or edges from the graph during the traversal. Instead, the effect of removing a vertex and all the edges emanating from that vertex is simulated by decreasing the apparent in-degrees of all the successors of the removed vertex.

In addition, we use a queue to keep track of the vertices that have not yet been visited, but whose in-degree is zero. Doing so eliminates the need to search the array for zero entries.

Program defines the TopologicalOrderTraversal method of the AbstractGraph class. This method takes as its argument a Visitor. The Visit method of the visitor is called once for each vertex in the graph. The order in which the vertices are visited is given by a topological sort of those vertices.
Implementation

```java
public abstract class AbstractGraph : AbstractContainer, Graph
{
    protected int numberOfVertices;
    protected int numberOfEdges;
    protected Vertex[] vertex;

    public void TopologicalOrderTraversal(Visitor visitor)
    {
        int[] indegree = new int[numberOfVertices];
        for (int v = 0; v < numberOfVertices; ++v)
            indegree[v] = 0;
        foreach (Edge e in Edges)
        {
            Vertex to = e.V1;
            ++indegree[to.Number];
        }

        Queue queue = new QueueAsLinkedList();
        for (int v = 0; v < numberOfVertices; ++v)
            if (indegree[v] == 0)
                queue.Enqueue(vertex[v]);
        while (!queue.IsEmpty && !visitor.IsDone)
        {
            Vertex v = (Vertex)queue.Dequeue();
            visitor.Visit(v);
            foreach (Vertex to in v.Successors)
            {
                if (indegree[to.Number] == 0)
                    queue.Enqueue(to);
            }
        }
    }
}
```

Program: AbstractGraph class TopologicalOrderTraversal method.

The algorithm begins by computing the in-degrees of all the vertices. An array of integers of length $V$ called `inDegree` is used for this purpose. First, all the array elements are set to zero. Then, for each
edge \((v_0, v_1) \in V\), array element \(\text{inDegree}(v_1)\) is increased by one (lines 12-16). Next, a queue to hold vertices is created. All vertices with in-degree zero are put into this queue (lines 18-21).

The main loop of the \texttt{TopologicalOrderTraversal} method comprises lines 22-31. This loop continues as long as the queue is not empty and the visitor is not finished. In each iteration of the main loop exactly one vertex is dequeued and visited (lines 24-25).

Once a vertex has been visited, the effect of removing that vertex from the graph is simulated by decreasing by one the in-degrees of all the successors of that vertex. When the in-degree of a vertex becomes zero, that vertex is enqueued (lines 26-30).
Running Time Analysis

The topological-order traversal enqueues each node in the graph at most once. When a node is dequeued, all the edges emanating from that node are considered. Therefore, a complete traversal enumerates every edge in the graph.

The worst-case running time for the traversal of a graph represented using an adjacency matrix is

\[ |V| \times T_{\text{visit}} + O(|V|^2). \]

When adjacency lists are used, the worst case running time for the topological-order traversal method is

\[ |V| \times T_{\text{visit}} + O(|V| + |E|). \]
Graph Traversal Applications: Testing for Cycles and Connectedness

This section presents several graph algorithms that are based on graph traversals. The first two algorithms test undirected and directed graphs for connectedness. Both algorithms are implemented using the depth-first traversal. The third algorithm tests a directed graph for cycles. It is implemented using a topological-order traversal.

- Connectedness of an Undirected Graph
- Connectedness of a Directed Graph
- Testing Strong Connectedness
- Testing for Cycles in a Directed Graph

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Connectedness of an Undirected Graph

Definition (Connectedness of an Undirected Graph) An undirected graph $G = (V, E)$ is connected if there is a path in $G$ between every pair of vertices in $V$.

Consider the undirected graph shown in Figure 1. It is tempting to interpret this figure as a picture of two graphs. However, the figure actually represents the undirected graph $G_A = (V, E)$, given by

$$V = \{a, b, c, d, e, f\}$$
$$E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{d, e\}, \{e, f\}\}.$$

Clearly, the graph $G_A$ is not connected. For example, there is no path between vertices $a$ and $d$. In fact, the graph $G_A$ consists of two, unconnected parts, each of which is a connected sub-graph. The connected sub-graphs of a graph are called connected components.

Figure: An unconnected, undirected graph with two (connected) components.

A traversal of an undirected graph (either depth-first or breadth-first) starting from any vertex will only visit all the other vertices of the graph if that graph is connected. Therefore, there is a very simply way to test whether an undirected graph is connected: Count the number of vertices visited during a traversal of the graph. Only if all the vertices are visited is the graph connected.

Program 2 shows how this can be implemented. The IsConnected property of the AbstractGraph class provides a bool-valued get accessor that returns true if the graph is connected.
Connectedness of an Undirected Graph

Program: AbstractGraph class IsConnected property.

The property is implemented using a the DepthFirstTraversal method and a visitor that simply counts the number of vertices it visits. The Visit method adds one the count field of the counter each time it is called.

The worst-case running time of the IsConnected property is determined by the time taken by the DepthFirstTraversal. Clearly in this case $T(\text{Visit}) = O(1)$. Therefore, the running time of IsConnected is $O(|V|^2)$ when adjacency matrices are used to represent the graph and $O(|V| + |E|)$.
when adjacency lists are used.
Connectedness of a Directed Graph

When dealing with directed graphs, we define two kinds of connectedness, *strong* and *weak*. Strong connectedness of a directed graph is defined as follows:

**Definition (Strong Connectedness of a Directed Graph)** A directed graph $G = (\mathcal{V}, \mathcal{E})$ is *strongly connected* if there is a path in $G$ between every pair of vertices in $\mathcal{V}$.

For example, Figure shows the directed graph $G_0 = (\mathcal{V}, \mathcal{E})$ given by

\[
\begin{align*}
\mathcal{V} &= \{a, b, c, d, e, f\} \\
\mathcal{E} &= \{(a, b), (b, c), (b, e), (c, a), (d, e), (e, f), (f, d)\}.
\end{align*}
\]

Notice that the graph $G_0$ is *not* connected! For example, there is no path from any of the vertices in $\{d, e, f\}$ to any of the vertices in $\{a, b, c\}$. Nevertheless, the graph “looks” connected in the sense that it is not made of up of separate parts in the way that the graph $G_8$ in Figure is.

This idea of “looking” connected is what *weak connectedness* represents. To define weak connectedness we need to introduce first the notion of the undirected graph that underlies a directed graph: Consider a directed graph $G = (\mathcal{V}, \mathcal{E})$. The underlying undirected graph is the graph $\hat{G} = (\mathcal{V}, \hat{\mathcal{E}})$ where $\hat{\mathcal{E}}$ represents the set of undirected edges that is obtained by removing the arrowheads from the directed edges in $G$:

\[
\hat{\mathcal{E}} = \{\{v, w\} : (v, w) \in \mathcal{E} \lor (w, v) \in \mathcal{E}\}.
\]
Figure: An weakly connected directed graph and the underlying undirected graph.

Weak connectedness of a directed graph is defined with respect to its underlying, undirected graph:

**Definition (Weak Connectedness of a Directed Graph)** A directed graph \( G = (V, E) \) is *weakly connected* if the underlying undirected graph \( \overline{G} \) is connected.

For example, since the undirected graph \( \overline{G} \) in Figure is connected, the directed graph \( G \) is weakly connected. Consider what happens when we remove the edge \((b, e)\) from the directed graph \( G \). The underlying undirected graph that we get is \( \overline{G_b} \) in Figure. Therefore, when we remove edge \((b, e)\) from \( G \), the graph that remains is neither strongly connected nor weakly connected.
Testing Strong Connectedness

A traversal of a directed graph (either depth-first or breadth-first) starting from a given vertex will only visit all the vertices of an undirected graph if there is a path from the start vertex to every other vertex. Therefore, a simple way to test whether a directed graph is strongly connected uses $|V|$ traversals--one starting from each vertex in $V$. Each time the number of vertices visited is counted. The graph is strongly connected if all the vertices are visited in each traversal.

Program □ shows how this can be implemented. It shows the get accessor for the `IsStronglyConnected` property of the `AbstractGraph` class which returns the `bool` value `true` if the graph is `strongly` connected.
Testing Strong Connectedness

Program: AbstractGraph class IsStronglyConnected property.

The accessor consists of a loop over all the vertices of the graph. Each iteration does a DepthFirstTraversal using a visitor that counts the number of vertices it visits. The running time for one iteration is essentially that of the DepthFirstTraversal since $T(\text{visit}) = O(1)$ for the counting visitor. Therefore, the worst-case running time for the IsConnected method is $O(|V|^3)$ when adjacency matrices are used and $O(|V|^2 + |V| \cdot |E|)$ when adjacency lists are used to represent the graph.
Testing for Cycles in a Directed Graph

The final application of graph traversal that we consider in this section is to test a directed graph for cycles. An easy way to do this is to attempt a topological-order traversal using the algorithm given in Section [Section]. This algorithm only visits all the vertices of a directed graph if that graph contains no cycles.

To see why this is so, consider the directed cyclic graph $G_{10}$ shown in Figure [Figure]. The topological traversal algorithm begins by computing the in-degrees of the vertices. (The number shown below each vertex in Figure [Figure] is the in-degree of that vertex).

At each step of the traversal, a vertex with in-degree of zero is visited. After a vertex is visited, the vertex and all the edges emanating from that vertex are removed from the graph. Notice that if we remove vertex $a$ and edge $(a,b)$ from $G_{10}$, all the remaining vertices have in-degrees of one. The presence of the cycle prevents the topological-order traversal from completing.

Therefore, the a simple way to test whether a directed graph is cyclic is to attempt a topological traversal of its vertices. If all the vertices are not visited, the graph must be cyclic.

Program [Program] gives the implementation of the IsCyclic property of the AbstractGraph class. This property provides a bool-valued get accessor that returns true if the graph is cyclic. The implementation uses a visitor that counts the number of vertices visited during a TopologicalOrderTraversal of the graph.
public abstract class AbstractGraph : AbstractContainer, Graph
{
    protected int numberOfVertices;
    protected int numberOfEdges;
    protected Vertex[] vertex;

    public bool IsCyclic
    {
        get
        {
            CountingVisitor visitor = new CountingVisitor();
            TopologicalOrderTraversal(visitor);
            return visitor.Count != numberOfVertices;
        }

        // ...
    }
}

Program: AbstractGraph class IsCyclic property.

The worst-case running time of the IsCyclic property is determined by the time taken by the TopologicalOrderTraversal. Since $T(visит) = O(1)$, the running time of IsCyclic is $O(|V|^2)$ when adjacency matrices are used to represent the graph and $O(|V| + |E|)$ when adjacency lists are used.
Shortest-Path Algorithms

In this section we consider edge-weighted graphs, both directed and undirected, in which the weight measures the cost of traversing that edge. The units of cost depend on the application.

For example, we can use a directed graph to represent a network of airports. In such a graph the vertices represent the airports and the edges correspond to the available flights between airports. In this scenario there are several possible cost metrics: If we are interested in computing travel time, then we use an edge-weighted graph in which the weights represent the flying time between airports. If we are concerned with the financial cost of a trip, then the weights on the edges represent the monetary cost of a ticket. Finally, if we are interested the actual distance traveled, then the weights represent the physical distances between airports.

If we are interested in traveling from point A to B, we can use a suitably labeled graph to answer the following questions: What is the fastest way to get from A to B? Which route from A to B has the least expensive airfare? What is the shortest possible distance traveled to get from A to B?

Each of these questions is an instance of the same problem: Given an edge-weighted graph, $G = (V, E)$, and two vertices, $v_a \in V$ and $v_d \in V$, find the path that starts at $v_a$ and ends at $v_d$ that has the smallest weighted path length. The weighted length of a path is defined as follows:

**Definition (Weighted Path Length)** Consider an edge-weighted graph $G = (V, E)$. Let $C'(v_i, v_j)$ be the weight on the edge connecting $v_i$ to $v_j$. A path in $G$ is a non-empty sequence of vertices $P = \{v_1, v_2, \ldots, v_k\}$. The weighted path length of path $P$ is given by

$$\sum_{i=1}^{k-1} C'(v_i, v_{i+1}).$$

The weighted length of a path is the sum of the weights on the edges in that path. Conversely, the unweighted length of a path is simply the number of edges in that path. Therefore, the unweighted length of a path is equivalent to the weighted path length obtained when all edge weights are one.
Shortest-Path Algorithms

- Single-Source Shortest Path
- All-Pairs Source Shortest Path

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno
Single-Source Shortest Path

In this section we consider the single-source shortest path problem: Given an edge-weighted graph \( G = (V, E) \) and a vertex \( v_a \in V \), find the shortest weighted path from \( v_a \) to every other vertex in \( V \).

Why do we find the shortest path to every other vertex if we are interested only in the shortest path from, say, \( v_a \) to \( v_d \)? It turns out that in order to find the shortest path from \( v_a \) to \( v_d \), it is necessary to find the shortest path from \( v_a \) to every other vertex in \( G \)! If a vertex is ignored, say \( v_i \), then we will not consider any of the paths from \( v_a \) to \( v_d \) that pass through \( v_i \). But if we fail to consider all the paths from \( v_a \) to \( v_d \), we cannot be assured of finding the shortest one.

Furthermore, suppose the shortest path from \( v_a \) to \( v_d \) passes through some intermediate node \( v_i \). That is, the shortest path is of the form \( P = \{v_a, \ldots, v_i, \ldots, v_d\} \). It must be the case that the portion of \( P \) between \( v_a \) to \( v_i \) is also the shortest path from \( v_a \) to \( v_i \). Suppose it is not. Then there exists another shorter path from \( v_a \) to \( v_i \). But then, \( P \) would not be the shortest path from \( v_a \) to \( v_d \), because we could obtain a shorter one by replacing the portion of \( P \) between \( v_a \) and \( v_i \) by the shorter path.

Consider the directed graph \( G_{11} \) shown in Figure. The shortest weighted path between vertices \( b \) and \( f \) is the path \( \{b, a, c, e, f\} \), which has the weighted path length nine. On the other hand, the shortest unweighted path is from \( b \) to \( f \) is the path of length three, \( \{b, c, e, f\} \).

![Figure: Two edge-weighted directed graphs.](http://www.brpreiss.com/books/opus6/html/page567.html)
As long as all the edge weights are non-negative (as is the case for $G_{11}$), the shortest-path problem is well defined. Unfortunately, things get a little tricky in the presence of negative edge weights.

For example, consider the graph $G_{12}$ shown in Figure 1. Suppose we are looking for the shortest path from $d$ to $f$. Exactly two edges emanate from vertex $d$, both with the same edge weight of five. If the graph contained only positive edge weights, there could be no shorter path than the direct path $\{d, f\}$.

However, in a graph that contains negative weights, a long path gets "shorter" when we add edges with negative weights to it. For example, the path $\{d, a, c, e, f\}$ has a total weighted path length of four, even though the first edge, $(d, a)$, has the weight five.

But negative weights are even more insidious than this: For example, the path $\{d, a, c, d, a, c, f\}$, which also joins vertex $d$ to $f$, has a weighted path length of two but the path $\{d, a, c, d, a, c, d, a, e, f\}$ has length zero. That is, as the number of edges in the path increases, the weighted path length decreases! The problem in this case is the existence of the cycle $\{d, a, c, d\}$ the weighted path length of which is less than zero. Such a cycle is called a negative cost cycle.

Clearly, the shortest-path problem is not defined for graphs that contain negative cost cycles. However, negative edges are not intrinsically bad. Solutions to the problem do exist for graphs that contain both positive and negative edge weights, as long as there are no negative cost cycles. Nevertheless, the problem is greatly simplified when all edges carry non-negative weights.
Dijkstra's Algorithm

Dijkstra's algorithm is a greedy algorithm for solving the single-source, shortest-path problem on an edge-weighted graph in which all the weights are non-negative. It finds the shortest paths from some initial vertex, say $v_0$, to all the other vertices one-by-one. The essential feature of Dijkstra's algorithm is the order in which the paths are determined: The paths are discovered in the order of their weighted lengths, starting with the shortest, proceeding to the longest.

For each vertex $v$, Dijkstra's algorithm keeps track of three pieces of information, $k_v$, $d_v$, and $p_v$:

$k_v$

The bool-valued flag $k_v$ indicates that the shortest path to vertex $v$ is known. Initially, $k_v = \text{false}$ for all $v \in V$.

$d_v$

The quantity $d_v$ is the length of the shortest known path from $v_0$ to $v$. When the algorithm begins, no shortest paths are known. The distance $d_v$ is a tentative distance. During the course of the algorithm candidate paths are examined and the tentative distances are modified.

Initially, $d_v = \infty$ for all $v \in V$ such that $v \neq v_0$, while $d_{v_0} = 0$.

$p_v$

The predecessor of vertex $v$ on the shortest path from $v_0$ to $v$. That is, the shortest path from $v_0$ to $v$ has the form $\{v_0, \ldots, p_v, v\}$.

Initially, $p_v$ is unknown for all $v \in V$.

Dijkstra's algorithm proceeds in phases. The following steps are performed in each pass:

1. From the set of vertices for with $k_v = \text{false}$, select the vertex $v$ having the smallest tentative distance $d_v$.
2. Set $k_v \leftarrow \text{true}$.
Dijkstra’s Algorithm

3. For each vertex \( w \) adjacent to \( v \) for which \( k_w \neq \text{true} \), test whether the tentative distance \( d_w \) is greater than \( d_v + C(v, w) \). If it is, set \( d_w \leftarrow d_v + C(v, w) \) and set \( p_w \leftarrow v \).

In each pass exactly one vertex has its \( k_v \) set to \( \text{true} \). The algorithm terminates after \(|V|\) passes are completed at which time all the shortest paths are known.

Table illustrates the operation of Dijkstra's algorithm as it finds the shortest paths starting from vertex \( b \) in graph \( G_11 \) shown in Figure.

<table>
<thead>
<tr>
<th>vertex</th>
<th>initially</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( \infty )</td>
<td>3</td>
<td>b</td>
<td>( \checkmark )</td>
<td>3</td>
<td>b</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>( b )</td>
<td>0</td>
<td>--</td>
<td>( \checkmark )</td>
<td>0</td>
<td>--</td>
<td>( \checkmark )</td>
<td>0</td>
</tr>
<tr>
<td>( c )</td>
<td>( \infty )</td>
<td>5</td>
<td>b</td>
<td>4</td>
<td>a</td>
<td>( \checkmark )</td>
<td>4</td>
</tr>
<tr>
<td>( d )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>6</td>
<td>c</td>
<td>( \checkmark )</td>
<td>6</td>
</tr>
<tr>
<td>( e )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>8</td>
<td>c</td>
<td>8</td>
<td>c</td>
</tr>
<tr>
<td>( f )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>11</td>
<td>d</td>
<td>9</td>
</tr>
</tbody>
</table>

Table: Operation of Dijkstra’s algorithm.

Initially all the tentative distances are \( \infty \), except for vertex \( b \) which has tentative distance zero. Therefore, vertex \( b \) is selected in the first pass. The mark \( \checkmark \) beside an entry in Table indicates that the shortest path is known (\( k_v = \text{true} \)).

Next we follow the edges emanating from vertex \( b \), \( b \rightarrow a \) and \( b \rightarrow c \), and update the distances accordingly. The new tentative distances for \( a \) becomes 3 and the new tentative distance for \( c \) is 5. In both cases, the next-to-last vertex on the shortest path is vertex \( b \).

In the second pass, vertex \( a \) is selected and its entry is marked with \( \checkmark \) indicating the shortest path is
known. There is one edge emanating from \( a, a \rightarrow c \). The distance to \( c \) via \( a \) is 4. Since this is less than the tentative distance to \( c \), vertex \( c \) is given the new tentative distance 4 and its predecessor on the shortest-path is set to \( a \). The algorithm continues in this fashion for a total of \( V \) passes until all the shortest paths have been found.

The shortest-path information contained in the right-most column of Table can be represented in the form of a vertex-weighted graph as shown in Figure.

---

**Figure:** The shortest-path graph for \( G_{11} \).

This graph contains the same set of vertices as the problem graph \( G_{11} \). Each vertex \( v \) is labeled with the length \( d_v \) of the shortest path from \( b \) to \( v \). Each vertex (except \( b \)) has a single emanating edge that connects the vertex to the next-to-last vertex on the shortest-path. By following the edges in this graph from any vertex \( v \) to vertex \( b \), we can construct the shortest path from \( b \) to \( v \) in reverse.
Data Structures for Dijkstra's Algorithm

The implementation of Dijkstra's algorithm described below uses the `Entry` struct declared in Program. Each `Entry` value has three fields, `known`, `distance`, and `predecessor`, which correspond to the variables $k_v$, $d_v$, and $p_v$, respectively.

```csharp
public class Algorithms {
    public struct Entry {
        public bool known;
        public int distance;
        public int predecessor;

        public Entry(bool known, int distance, int predecessor) {
            this.known = known;
            this.distance = distance;
            this.predecessor = predecessor;
        }
    }
}
```

**Program:** `GraphAlgorithms Entry` struct.

In each pass of its operation, Dijkstra's algorithm selects from the set of vertices for which the shortest-path is not yet known the one with the smallest tentative distance. Therefore, we use a priority queue to represent this set of vertices.

The priority assigned to a vertex is its tentative distance. The class `Association` class introduced in Program is used to associate a priority with a given vertex instance.
Implementation

An implementation of Dijkstra's algorithm is shown in Program 3. The DijkstraAlgorithm method takes two arguments. The first is a directed graph. It is assumed that the directed graph is an edge-weighted graph in which the weights are int's. The second argument is the number of the start vertex, v0.

The DijkstraAlgorithm method returns its result in the form of a shortest-path graph. Therefore, the return value is a Digraph instance.

```csharp
public class Algorithms {

    public static Digraph DijkstraAlgorithm(Digraph g, int s) {
        int n = g.NumberOfVertices;
        Entry[] table = new Entry[n];
        for (int v = 0; v < n; ++v)
            table[v] = new Entry(false, int.MaxValue, int.MaxValue);
        table[s].distance = 0;
        PriorityQueue queue = new BinaryHeap(g.NumberOfEdges);
        queue.Enqueue(new Association(0, g.GetVertex(s)));
        while (!queue.IsEmpty)
        {
            Association assoc = (Association)queue.DequeueMin();
            Vertex v0 = (Vertex)assoc.Value;
            if (!table[v0.Number].known)
            {
                table[v0.Number].known = true;
                foreach (Edge e in v0.EmanatingEdges)
                {
                    Vertex v1 = e.MateOf(v0);
                    int d = table[v0.Number].distance + (int)e.Weight;
                    if (table[v1.Number].distance > d)
                    {
```

Program: Dijkstra's algorithm.

The main data structures used are called `table` and `queue` (lines 6 and 12). The former is an array of $n = |V|$ `Entry` elements. The latter is a priority queue. In this case, a `BinaryHeap` of length $|E|$ is used. (See Section □).

The algorithm begins by setting the tentative distance for the start vertex to zero and inserting the start vertex into the priority queue with priority zero (lines 10-12).

The main loop of the method comprises lines 13-33. In each iteration of this loop the vertex with the smallest distance is dequeued (line 15). The vertex is processed only if its table entry indicates that the shortest path is not already known (line 17).

When a vertex $v_0$ is processed, its shortest path is deemed to be known (line 19). Then each vertex $v_1$ adjacent to vertex is considered (lines 20-31). The distance to $v_1$ along the path that passes through $v_0$ is computed (lines 23-24). If this distance is less than the tentative distance associated with $v_1$, entries in the table for $v_1$ are updated, and the $v_1$ is given a new priority and inserted into the priority queue (lines 25-30).

The main loop terminates when all the shortest paths have been found. The shortest-path graph is then constructed using the information in the table (lines 34-39).

```java
26  {
27      table[v1.Number].distance = d;
28      table[v1.Number].predecessor = v0.Number;
29      queue.Enqueue(new Association(d, v1));
30  }
31 }
32 }
33 }
34           Digraph result = new DigraphAsLists(n);
35           for (int v = 0; v < n; ++v)
36               result.AddVertex(v, table[v].distance);
37           for (int v = 0; v < n; ++v)
38               if (v != s)
39                 result.AddEdge(v, table[v].predecessor);
40             return result;
41       }
42   }
```
Running Time Analysis

The running time of the Dijkstra's Algorithm method is dominated by the running time of the main loop (lines 13-33). (It is easy to see that lines 5-9 and 34-39 run in $O(|V|)$ time).

To determine the running time of the main loop, we proceed as follows: First, we ignore temporarily the time required for the Enqueue and Dequeue operations in the priority queue. Clearly, each vertex in the graph is processed exactly once. When a vertex is processed all the edges emanating from it are considered. Therefore, the time (ignoring the priority queue operations) taken is $O(|V|+|E|)$ when adjacency lists are used and $O(|V|^2)$ when adjacency matrices are used.

Now, we add back the worst-case time required for the priority queue operations. In the worst case, a vertex is enqueued and subsequently dequeued once for every edge in the graph. Therefore, the length of the priority queue is at most $|E|$. As a result, the worst-case time for each operation is $O(\log |E|)$.

Thus, the worst-case running time for Dijkstra's algorithm is

$$O(|V| + |E| \log |E|),$$

when adjacency lists are used, and

$$O(|V|^2 + |E| \log |E|),$$

when adjacency matrices are used to represent the input graph.
All-Pairs Source Shortest Path

In this section we consider the *all-pairs, shortest path* problem: Given an edge-weighted graph \( G = (\mathcal{V}, \mathcal{E}) \), for each pair of vertices in \( \mathcal{V} \) find the *length* of the shortest weighted path between the two vertices.

One way to solve this problem is to run Dijkstra's algorithm \(|\mathcal{V}|\) times in turn using each vertex in \( \mathcal{V} \) as the initial vertex. Therefore, we can solve the all-pairs problem in \( \mathcal{O}(|\mathcal{V}|^2 + |\mathcal{V}| |\mathcal{E}| \log |\mathcal{E}|) \) time when adjacency lists are used, and \( \mathcal{O}(|\mathcal{V}|^3 + |\mathcal{V}| |\mathcal{E}| \log |\mathcal{E}|) \), when adjacency matrices are used. However, for a dense graph \( |\mathcal{E}| = \Theta(|\mathcal{V}|^2) \) the running time of Dijkstra's algorithm is \( \mathcal{O}(|\mathcal{V}|^3 \log |\mathcal{V}|) \), regardless of the representation scheme used.

---

- **Floyd's Algorithm**
- **Implementation**
- **Running Time Analysis**
Floyd's Algorithm

Floyd's algorithm uses the dynamic programming method to solve the all-pairs shortest-path problem on a dense graph. The method makes efficient use of an adjacency matrix to solve the problem. Consider an edge-weighted graph $G = (V, E)$, where $C(v, w)$ represents the weight on edge $(v, w)$. Suppose the vertices are numbered from 1 to $|V|$. That is, let $V = \{v_1, v_2, \ldots, v_n\}$. Furthermore, let $V_k$ be the set comprised of the first $k$ vertices in $V$. That is, $V_k = \{v_1, v_2, \ldots, v_k\}$, for $0 \leq k \leq |V|$.

Let $P_k(v, w)$ be the shortest path from vertex $v$ to $w$ that passes only through vertices in $V_k$, if such a path exists. That is, the path $P_k(v, w)$ has the form

$$P_k(v, w) = \{v, \ldots, w\} \in V_k$$

Let $D_k(v, w)$ be the length of path $P_k(v, w)$:

$$D_k(v, w) = \begin{cases} |P_k(v, w)| & P_k(v, w) \text{ exists,} \\ \infty & \text{otherwise.} \end{cases}$$

Since $V_0 = \emptyset$, the $P_0$ paths are correspond to the edges of $G$:

$$P_0(v, w) = \begin{cases} \{v, w\} & (v, w) \in E, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Therefore, the $D_0$ path lengths correspond to the weights on the edges of $G$:

$$D_0(v, w) = \begin{cases} C(v, w) & (v, w) \in E, \\ \infty & \text{otherwise.} \end{cases}$$

Floyd's algorithm computes the sequence of matrices $D_0, D_1, \ldots, D_{|V|}$ The distances in $D_k$ represent
paths with intermediate vertices in $\mathcal{V}_i$. Since $\mathcal{V}_{i+1} = \mathcal{V}_i \cup \{v_{i+1}\}$, we can obtain the distances in $D_{i+1}$ from those in $D_i$ by considering only the paths that pass through vertex $v_{i+1}$. Figure shows how this is done.

**Figure:** Calculating $D_{i+1}$ in Floyd's algorithm.

For every pair of vertices $(v,w)$, we compare the distance $D_i(v,w)$ (which represents the shortest path from $v$ to $w$ that does not pass through $v_{i+1}$) with the sum $D_i(v,v_{i+1}) + D_i(v_{i+1},w)$ (which represents the shortest path from $v$ to $w$ that does pass through $v_{i+1}$). Thus, $D_{i+1}$ is computed as follows:

$$D_{i+1}(v,w) = \min\{D_i(v,v_{i+1}) + D_i(v_{i+1},w), D_i(v,w)\}.$$
Implementation

An implementation of Floyd's algorithm is shown in Program □. The FloydsAlgorithm method takes as its argument a directed graph. The directed graph is assumed to be an edge-weighted graph in which the weights are ints.

```csharp
public class Algorithms
{
    public static Digraph FloydsAlgorithm(Digraph g)
    {
        int n = g.NumberOfVertices;
        int[,] distance = new int[n, n];
        for (int v = 0; v < n; ++v)
            for (int w = 0; w < n; ++w)
                distance[v, w] = int.MaxValue;

        foreach (Edge e in g.Edges)
        {
            distance[e.V0.Number, e.V1.Number] = (int)e.Weight;
        }

        for (int i = 0; i < n; ++i)
            for (int v = 0; v < n; ++v)
                for (int w = 0; w < n; ++w)
                    if (distance[v, i] != int.MaxValue &&
                        distance[i, w] != int.MaxValue)
                        {
                            int d = distance[v, i] + distance[i, w];
                            if (distance[v, w] > d)
                                distance[v, w] = d;
                        }

        Digraph result = new DigraphAsMatrix(n);
        for (int v = 0; v < n; ++v)
            result.AddVertex(v);
```

Implementation

for (int v = 0; v < n; ++v)
    result.AddVertex(v);
for (int v = 0; v < n; ++v)
    for (int w = 0; w < n; ++w)
        if (distance[v, w] != int.MaxValue)
            result.AddEdge(v, w, distance[v, w]);
    return result;

Program: Floyd's algorithm.

The Floyd's Algorithm method returns its result in the form of an edge-weighted directed graph. Therefore, the return value is a Digraph.

The principal data structure use by the algorithm is a $|V| \times |V|$ matrix of integers called distance. All the elements of the matrix are initially set to $\infty$ (lines 6-9). Next, an edge enumerator is used to visit all the edges in the input graph in order to transfer the weights from the graph to the distance matrix (lines 11-14).

The main work of the algorithm is done in three, nested loops (lines 16-25). The outer loop computes the sequence of distance matrices $D_1, D_2, \ldots, D_{|V|}$. The inner two loops consider all possible pairs of vertices. Notice that as $D_{i+1}$ is computed, its entries overwrite those of $D_i$.

Finally, the values in the distance matrix are transferred to the result graph (lines 27-33). The result graph contains the same set of vertices as the input graph. For each finite entry in the distance matrix, a weighted edge is added to the result graph.
Running Time Analysis

The worst-case running time for Floyd's algorithm is easily determined. Creating and initializing the distance matrix is $O(|V|^2)$ (lines 6-9). Transferring the weights from the input graph to the distance matrix requires $O(|V| + |E|)$ time if adjacency lists are used, and $O(|V|^2)$ time when an adjacency matrix is used to represent the input graph (lines 11-14).

The running time for the three nested loops is $O(|V|^3)$ in the worst case. Finally, constructing the result graph and transferring the entries from the distance matrix to the result requires $O(|V|^2)$ time. As a result, the worst-case running time of Floyd's algorithm is $O(|V|^3)$.
Minimum-Cost Spanning Trees

In this section we consider undirected graphs and their subgraphs. A subgraph of a graph $G = (\mathcal{V}, \mathcal{E})$ is any graph $G' = (\mathcal{V}', \mathcal{E}')$ such that $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{E}' \subseteq \mathcal{E}$. In particular, we consider connected undirected graphs and their minimal subgraphs. The minimal subgraph of a connected graph is called a spanning tree:

**Definition (Spanning Tree)** Consider a connected, undirected graph $G = (\mathcal{V}, \mathcal{E})$. A spanning tree of $G$ is a subgraph of $G$, say $T = (\mathcal{V}', \mathcal{E}')$, with the following properties:

1. $\mathcal{V}' = \mathcal{V}$.
2. $T$ is connected.
3. $T$ is acyclic.

Figure shows an undirected graph, $G_{13}$, together with three of its spanning trees. A spanning tree is called a tree because every acyclic undirected graph can be viewed as a general, unordered tree. Because the edges are undirected, any vertex may be chosen to serve as the root of the tree. For example, the spanning tree of $G_{13}$ given in Figure (c) can be viewed as the general, unordered tree

$$\{b, \{a\}, \{c\}, \{e, \{d\}, \{f\}\}\}.$$
According to Definition, a spanning tree is connected. Therefore, as long as the tree contains more than one vertex, there can be no vertex with degree zero. Furthermore, the following theorem guarantees that there is always at least one vertex with degree one:

**Theorem**  Consider a connected, undirected graph \( G = (\mathcal{V}, \mathcal{E}) \), where \(|\mathcal{V}| > 1\). Let \( T = (\mathcal{V}, \mathcal{E}') \) be a spanning tree of \( G \). The spanning tree \( T \) contains at least one vertex of degree one.

**Proof (By contradiction).** Assume that there is no vertex in \( T \) of degree one. That is, all the vertices in \( T \) have degree two or greater. Then by following edges into and out of vertices we can construct a path that is cyclic. But a spanning tree is acyclic—a contradiction. Therefore, a spanning tree always contains at least one vertex of degree one.

According to Definition, the edge set of a spanning tree is a subset of the edges in the spanned graph. How many edges must a spanning tree have? The following theorem answers the question:
**Theorem** Consider a connected, undirected graph \( G = (\mathcal{V}, \mathcal{E}) \). Let \( T = (\mathcal{V}, \mathcal{E}') \) be a spanning tree of \( G \). The number of edges in the spanning tree is given by

\[
|\mathcal{E}'| = |\mathcal{V}| - 1.
\]

**Proof (By induction).** We can prove Theorem by induction on \( |\mathcal{V}| \), the number of vertices in the graph.

**Base Case** Consider a graph that contains only one node, i.e., \( |\mathcal{V}| = 1 \). Clearly, the spanning tree for such a graph contains no edges. Since \( |\mathcal{V}| - 1 = 0 \), the theorem is valid.

**Inductive Hypothesis** Assume that the number of edges in a spanning tree for a graph with \( |\mathcal{V}| \) has been shown to be \( |\mathcal{V}| - 1 \) for \( |\mathcal{V}| = 1, 2, \ldots, k \).

Consider a graph \( G_{k+1} = (\mathcal{V}, \mathcal{E}) \) with \( k+1 \) vertices and its spanning tree \( T_{k+1} = (\mathcal{V}, \mathcal{E}') \). According to Theorem, \( G_{k+1} \) contains at least one vertex of degree one. Let \( v \in \mathcal{V} \) be one such vertex and \( \{v, w\} \in \mathcal{E}' \) be the one edge emanating from \( v \) in \( T_{k+1} \).

Let \( T_k \) be the graph of \( k \) nodes obtained by removing \( v \) and its emanating edge from the graph \( T_{k+1} \). That is, \( T_k = (\mathcal{V} - \{v\}, \mathcal{E}' - \{v, w\}) \).

Since \( T_{k+1} \) is connected, so too is \( T_k \). Similarly, since \( T_{k+1} \) is acyclic, so too is \( T_k \). Therefore \( T_k \) is a spanning tree with \( k \) vertices. By the inductive hypothesis \( T_k \) has \( k-1 \) edges. Thus, \( T_{k+1} \) as \( k \) edges.

Therefore, by induction on \( k \), the spanning tree for a graph with \( |\mathcal{V}| \) vertices contains \( |\mathcal{V}| - 1 \) edges.
Constructing Spanning Trees

Any traversal of a connected, undirected tree visits all the vertices in that tree, regardless of the node from which the traversal is started. During the traversal certain edges are traversed while the remaining edges are not. Specifically, an edge is traversed if it leads from a vertex that has been visited to a vertex that has not been visited. The set of edges which are traversed during a traversal forms a spanning tree.

The spanning tree obtained from a breadth-first traversal starting at vertex \( v \) of graph \( G \) is called the *breadth-first spanning tree* of \( G \) rooted at \( v \). For example, the spanning tree shown in Figure (c) is the breadth-first spanning tree of \( G_{13} \) rooted at vertex \( b \).

Similarly, the spanning tree obtained from a depth-first traversal is the *depth-first spanning tree* of \( G \) rooted at \( v \). The spanning tree shown in Figure (d) is the depth-first spanning tree of \( G_{13} \) rooted at vertex \( c \).
Minimum-Cost Spanning Trees

The total cost of an edge-weighted undirected graph is simply the sum of the weights on all the edges in that graph. A minimum-cost spanning tree of a graph is a spanning tree of that graph that has the least total cost:

**Definition (Minimal Spanning Tree)** Consider an edge-weighted, undirected, connected graph \( G = (V, E) \), where \( C(v, w) \) represents the weight on edge \( \{v, w\} \in E \). The minimum spanning tree of \( G \) is the spanning tree \( T = (V, E') \) that has the smallest total cost,

\[
\sum_{\{v, w\} \in E'} C(v, w).
\]

Figure shows edge-weighted graph \( G_{14} \) together with its minimum-cost spanning tree \( T_{14} \). In general, it is possible for a graph to have several different minimum-cost spanning trees. However, in this case there is only one.

**Figure:** An edge-weighted, undirected graph and a minimum-cost spanning tree.

The two sections that follow present two different algorithms for finding the minimum-cost spanning tree. Both algorithms are similar in that they build the tree one edge at a time.
Prim's Algorithm

Prim's algorithm finds a minimum-cost spanning tree of an edge-weighted, connected, undirected graph \( G = (V, E) \). The algorithm constructs the minimum-cost spanning tree of a graph by selecting edges from the graph one-by-one and adding those edges to the spanning tree.

Prim's algorithm is essentially a minor variation of Dijkstra's algorithm (see Section □). To construct the spanning tree, the algorithm constructs a sequence of spanning trees \( T_0, T_1, \ldots, T_{|V|-1} \), each of which is a subgraph of \( G \). The algorithm begins with a tree that contains one selected vertex, say \( v_a \in V \). That is, \( T_0 = \{\{v_a\}, \emptyset\} \).

Given \( T_i = \{V_i, E_i\} \), we obtain the next tree in the sequence as follows. Consider the set of edges given by

\[
\mathcal{H}_i = \bigcup_{u \in V_i} A(u) - \bigcup_{u \in V_i} I(u).
\]

The set \( \mathcal{H}_i \) contains all the edges \( \{v, w\} \) such that exactly one of \( v \) or \( w \) is in \( V_i \) (but not both). Select the edge \( \{v, w\} \in \mathcal{H}_i \) with the smallest edge weight,

\[
C'(v, w) = \min_{\{v', w'\} \in \mathcal{H}_i} C'(v', w').
\]

Then \( T_{i+1} = \{V_{i+1}, E_{i+1}\} \), where \( V_{i+1} = V_i \cup \{v\} \) and \( E_{i+1} = E \cup \{\{v, w\}\} \). After \( |V|-1 \) such steps we get \( T_{|V|-1} \) which is the minimum-cost spanning tree of \( G \).

Figure □ illustrates how Prim's algorithm determines the minimum-cost spanning tree of the graph \( G_{14} \) shown in Figure □. The circled vertices are the elements of \( V_i \), the solid edges represent the elements of \( E_i \) and the dashed edges represent the elements of \( \mathcal{H}_i \).
Figure: Operation of Prim's algorithm.
Implementation

An implementation of Prim's algorithm is shown in Program 4. This implementation is almost identical to the version of Dijkstra's algorithm given in Program 4. In fact, there are only four differences between the two algorithms. These are found on lines 3, 23-25, 34, and 36.

```csharp
public class Algorithms
{
    public static Graph PrimsAlgorithm(Graph g, int s)
    {
        int n = g.NumberOfVertices;
        Entry[] table = new Entry[n];
        for (int v = 0; v < n; ++v)
            table[v] = new Entry(false,
                                 int.MaxValue, int.MaxValue);
        table[s].distance = 0;
        PriorityQueue queue = new BinaryHeap(g.NumberOfEdges);
        queue.Enqueue(new Association(0, g.GetVertex(s)));
        while (!queue.IsEmpty)
        {
            Association assoc = (Association)queue.DequeueMin();
            Vertex v0 = (Vertex)assoc.Value;
            if (!table[v0.Number].known)
            {
                table[v0.Number].known = true;
                foreach (Edge e in v0.EmanatingEdges)
                {
                    Vertex v1 = e.MateOf(v0);
                    int d = (int)e.Weight;
                    if (!table[v1.Number].known &&
                        table[v1.Number].distance > d)
                    {
                        table[v1.Number].distance = d;
                        table[v1.Number].predecessor = v0.Number;
                        queue.Enqueue(new Association(d, v1));
                    }
                }
            }
        }
    }
}
```
Program: Prim's algorithm.

The `PrimsAlgorithm` method takes two arguments. The first is an undirected graph instance. We assume that the graph is edge-weighted and that the weights are `ints`. The second argument is the number of the start vertex, `v0`.

The `PrimsAlgorithm` method returns a minimum-cost spanning tree represented as an undirected graph. Therefore, the return value is a `Graph`.

The running time of Prim's algorithm is asymptotically the same as Dijkstra's algorithm. That is, the worst-case running time is

$$O(|V| + |E| \log |E|),$$

when adjacency lists are used, and

$$O(|V|^2 + |E| \log |E|),$$

when adjacency matrices are used to represent the input graph.
Kruskal's Algorithm

Like Prim's algorithm, *Kruskal's algorithm* also constructs the minimum spanning tree of a graph by adding edges to the spanning tree one-by-one. At all points during its execution the set of edges selected by Prim's algorithm forms exactly one tree. On the other hand, the set of edges selected by Kruskal's algorithm forms a forest of trees.

Kruskal's algorithm is conceptually quite simple. The edges are selected and added to the spanning tree in increasing order of their weights. An edge is added to the tree only if it does not create a cycle.

The beauty of Kruskal's algorithm is the way that potential cycles are detected. Consider an undirected graph $G = (V, E)$. We can view the set of vertices, $V$, as a *universal set* and the set of edges, $E$, as the definition of an *equivalence relation* over the universe $V$. (See Definition □). In general, an equivalence relation partitions a universal set into a set of equivalence classes. If the graph is connected, there is only one equivalence class--all the elements of the universal set are equivalent. Therefore, a *spanning tree* is a minimal set of equivalences that result in a single equivalence class.

Kruskal's algorithm computes, $P_0, P_1, \ldots, P_{|V|-1}$, a sequence of partitions of the set of vertices $V$. (Partitions are discussed in Section □). The initial partition consists of $|V|$ sets of size one:

$$P_0 = \{\{v_1\}, \{v_2\}, \ldots, \{v_{|V|}\}\}.$$

Each subsequent element of the sequence is obtained from its predecessor by *joining* two of the elements of the partition. Therefore, $P_i$ has the form

$$P_i = \{S_0^i, S_1^i, \ldots, S_{|V|-1-i}^i\},$$

for $0 \leq i \leq |V| - 1$.

To construct the sequence the edges in $E$ are considered one-by-one in increasing order of their weights. Suppose we have computed the sequence up to $P_i$ and the next edge to be considered is $\{v, w\}$. If $v$ and $w$ are both members of the same element of partition $P_i$ then the edge forms a cycle, and is not part of the minimum-cost spanning tree.
On the other hand, suppose $v$ and $w$ are members of two different elements of partition $P_i$, say $S_k^i$ and $S_l^i$ (respectively). Then $\{v, w\}$ must be an edge in the minimum-cost spanning tree. In this case, we compute $P_{i+1}$ by joining $S_k^i$ and $S_l^i$. That is, we replace $S_k^i$ and $S_l^i$ in $P_i$ by the union $S_k^i \cup S_l^i$.

Figure □ illustrates how Kruskal’s algorithm determines the minimum-cost spanning tree of the graph $G_{14}$ shown in Figure □. The algorithm computes the following sequence of partitions:

$$
\begin{align*}
P_0 &= \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}\} \\
P_1 &= \{\{a, d\}, \{b\}, \{c\}, \{e\}, \{f\}\} \\
P_2 &= \{\{a, d\}, \{b\}, \{c\}, \{e, f\}\} \\
P_3 &= \{\{a, d\}, \{b\}, \{c, e, f\}\} \\
P_4 &= \{\{a, c, d, e, f\}, \{b\}\} \\
P_5 &= \{\{a, b, c, d, e, f\}\}
\end{align*}
$$
Figure: Operation of Kruskal's algorithm.

- Implementation
- Running Time Analysis
Implementation

An implementation of Kruskal's algorithm is shown in Program 5. The `KruskalsAlgorithm` method takes as its argument an edge-weighted, undirected graph. This implementation assumes that the edge weights are `int`s. The method computes the minimum-cost spanning tree and returns it in the form of an edge-weighted undirected graph.

```csharp
public class Algorithms {
    public static Graph KruskalsAlgorithm(Graph g) {
        int n = g.NumberOfVertices;

        Graph result = new GraphAsLists(n);
        for (int v = 0; v < n; ++v)
            result.AddVertex(v);

        PriorityQueue queue =
            new BinaryHeap(g.NumberOfEdges);
        foreach (Edge e in g.Edges)
        {
            int weight = (int)e.Weight;
            queue.Enqueue(new Association(weight, e));
        }

        Partition partition = new PartitionAsForest(n);
        while (!queue.IsEmpty &amp;amp; partition.Count > 1)
        {
            Association assoc = (Association)queue.DequeueMin();
            Edge e = (Edge)assoc.Value;
            int n0 = e.V0.Number;
            int n1 = e.V1.Number;
            Set s = partition.Find(n0);
            Set t = partition.Find(n1);
            if (s != t)
```
Implementation

    set s = partition.Find(n1);
    if (s != t)
    {
        partition.Join(s, t);
        result.AddEdge(n0, n1);
    }
    return result;
}

Program: Kruskal's algorithm.

The main data structures used by the method are a priority queue to hold the edges, a partition to detect cycles and a graph for the result. This implementation uses a BinaryHeap (Section 0) for the priority queue (lines 11-12), a PartitionAsForest (Section 0) for the partition (line 19) and a GraphAsLists for the spanning tree (line 7).
Running Time Analysis

The Kruskal's Algorithm method begins by creating an graph to hold the result spanning tree (lines 7-9). Since a spanning tree is a sparse graph the GraphAsLists class is used to represent it. Initially the graph contains $|V|$ vertices but no edges. The running time for lines 7-9 is $O(|V|)$.

Next all of the edges in the input graph are inserted one-by-one into the priority queue (lines 11-17). Since there are $|E|$ edges, the worst-case running time for a single insertion is $O(\log |E|)$. Therefore, the worst-case running time to initialize the priority queue is

$$O(|V| + |E| \log |E|),$$

when adjacency lists are used, and

$$O(|V|^2 + |E| \log |E|),$$

when adjacency matrices are used to represent the input graph.

The main loop of the method comprises lines 20-33. This loop is done at most $|E|$ times. In each iteration of the loop, one edge is removed from the priority queue (lines 22-23). In the worst-case this takes $O(\log |E|)$ time.

Then, two partition *find* operations are done to determine the elements of the partition that contain the two end-points of the given edge (lines 24-27). Since the partition contains at most $|V|$ elements, the running time for the find operations is $O(\log |V|)$. If the two elements of the partition are distinct, then an edge is added to the spanning tree and a *join* operation is done to unite the two elements of the partition (lines 28-32). The join operation also requires $O(\log |V|)$ time in the worst-case. Therefore, the total running time for the main loop is $O(|E| \log |V| + |E| \log |V|)$.

Thus, the worst-case running time for Kruskal's algorithm is

$$O(|V| + |E| \log |E| + |E| \log |V|),$$
when adjacency lists are used, and

\[ O(|V|^2 + |E| \log |E| + |E| \log |V|), \]

when adjacency matrices are used to represent the input graph.
Application: Critical Path Analysis

In the introduction to this chapter it is stated that there are myriad applications of graphs. In this section we consider one such application—critical path analysis. Critical path analysis crops up in a number of different contexts, from the planning of construction projects to the analysis of combinational logic circuits.

For example, consider the scheduling of activities required to construct a building. Before the foundation can be poured, it is necessary to dig a hole in the ground. After the building has been framed, the electricians and the plumbers can rough-in the electrical and water services and this rough-in must be completed before the insulation is put up and the walls are closed in.

We can represent the set of activities and the scheduling constraints using a vertex-weighted, directed acyclic graph (DAG). Each vertex represents an activity and the weight on the vertex represents the time required to complete the activity. The directed edges represent the sequencing constraints. That is, an edge from vertex $v$ to vertex $w$ indicates that activity $v$ must complete before $w$ may begin. Clearly, such a graph must be acyclic.

A graph in which the vertices represent activities is called an activity-node graph. Figure shows an example of an activity-node graph. In such a graph it is understood that independent activities may proceed in parallel. For example, after activity $A$ is completed, activities $B$ and $C$ may proceed in parallel. However, activity $D$ cannot begin until both $B$ and $C$ are done.

**Figure:** An activity-node graph.

Critical path analysis answers the following questions:
1. What is the minimum amount of time needed to complete all activities?
2. For a given activity \( v \), is it possible to delay the completion of that activity without affecting the overall completion time? If yes, by how much can the completion of activity \( v \) be delayed?

The activity-node graph is a vertex-weighted graph. However, the algorithms presented in the preceding sections all require edge-weighted graphs. Therefore, we must convert the vertex-weighted graph into its edge-weighted dual. In the dual graph the edges represent the activities, and the vertices represent the commencement and termination of activities. For this reason, the dual graph is called an event-node graph.

Figure 1 shows the event-node graph corresponding to the activity node graph given in Figure 0. Where an activity depends on more than one predecessor it is necessary to insert dummy edges.

**Figure:** The event-node graph corresponding to Figure 0.

For example, activity \( D \) cannot commence until both \( B \) and \( C \) are finished. In the event-node graph vertex 2 represents the termination of activity \( B \) and vertex 3 represents the termination of activity \( C \). It is necessary to introduce vertex 4 to represent the event that both \( B \) and \( C \) have completed. Edges \( 2 \to 4 \) and \( 3 \to 4 \) represent this synchronization constraint. Since these edges do not represent activities, the edge weights are zero.

For each vertex \( v \) in the event node graph we define two times. The first \( E_v \) is the earliest event time for event \( v \). It is the earliest time at which event \( v \) can occur assuming the first event begins at time zero. The earliest event time is given by

\[
E_w = \begin{cases} 
0 & \text{if } w = v_i, \\
\min_{(v, w) \in I(w)} E_v + C(v, w) & \text{otherwise}, 
\end{cases}
\]

(16.3)

where \( v_i \) is the initial event, \( I(w) \) is the set of incident edges on vertex \( w \) and \( C(v,w) \) is the weight on vertex \( (v,w) \).
Similarly, $L_v$ is the latest event time for event $v$. It is the latest time at which event $v$ can occur. The latest event time is given by

$$L_v = \begin{cases} E_v, & w = v_f, \\ \max_{(v,w) \in A(w)} E_w - C(v,w) & \text{otherwise}, \end{cases}$$

(16.4)

where $v_f$ is the final event.

Given the earliest and latest event times for all events, we can compute time available for each activity. For example, consider an activity represented by edge $(v,w)$. The amount of time available for the activity is $L_w - E_v$ and the time required for that activity is $C(v,w)$. We define the slack time for an activity as the amount of time by which an activity can be delayed without affecting the overall completion time of the project. The slack time for the activity represented by edge $(v,w)$ is given by

$$S(v,w) = L_w - E_v - C(v,w).$$

(16.5)

Activities with zero slack are critical. That is, critical activities must be completed on time--any delay affects the overall completion time. A critical path is a path in the event-node graph from the initial vertex to the final vertex comprised solely of critical activities.

Table gives the results from obtained from the critical path analysis of the activity-node graph shown in Figure. The tabulated results indicate the critical path is

$$\{A, C, D, E, G\}.$$
Table: Critical path analysis results for the activity-node graph in Figure □.
Implementation

Given an activity-node graph, the objective of critical path analysis is to determine the slack time for each activity and thereby to identify the critical activities and the critical path. We shall assume that the activity node graph has already been transformed to an edge-node graph. The implementation of this transformation is left as a project for the reader (Project]). Therefore, the first step is to compute the earliest and latest event times.

According to Equation, the earliest event time of vertex $w$ is obtained from the earliest event times of all its predecessors. Therefore, must compute the earliest event times in topological order. To do this, we define the EarliestTimeVisitor shown in Program.
public class Algorithms {
    private class EarliestTimeVisitor : AbstractVisitor {
        private int[] earliestTime;

        internal EarliestTimeVisitor(int[] earliestTime) {
            this.earliestTime = earliestTime;
        }

        public override void Visit(object obj) {
            Vertex w = (Vertex)obj;
            int max = earliestTime[0];
            foreach (Edge e in w.IncidentEdges) {
                max = Math.Max(max, earliestTime[e.V0.Number] + (int)e.Weight);
            }
            earliestTime[w.Number] = max;
        }
    }
}

Program: Critical path analysis--computing earliest event times.

The EarliestTimeVisitor has one field, earliestTime, which is an array used to record the $E_v$ values. The Visit method of the EarliestTimeVisitor class implements directly Equation \[\min_{(v,w) \in \mathcal{I}(w)} E_v + C(v,w)\]. It uses an IncidentEdges enumerator to determine all the predecessors of a given node and computes $\min_{(v,w) \in \mathcal{I}(w)} E_v + C(v,w)$.

In order to compute the latest event times, it is necessary to define also a LatestTimeVisitor. This visitor must visit the vertices of the event-node graph in reverse topological order. Its implementation follows directly from Equation \[\] and Program \[\].

Program \[\] defines the method called CriticalPathAnalysis that does what its name implies. This method takes as its argument a Digraph that represents an event-node graph. This implementation assumes that the edge weights are ints.
public class Algorithms {
    public static Digraph CriticalPathAnalysis(Digraph g) {
        int n = g.NumberOfVertices;
        int[] earliestTime = new int[n];
        earliestTime[0] = 0;
        g.TopologicalOrderTraversal(
            new EarliestTimeVisitor(earliestTime));

        int[] latestTime = new int[n];
        latestTime[n - 1] = earliestTime[n - 1];
        g.DepthFirstTraversal(new PostOrder(
            new LatestTimeVisitor(latestTime)), 0);

        Digraph slackGraph = new DigraphAsLists(n);
        for (int v = 0; v < n; ++v)
            slackGraph.AddVertex(v);
        foreach (Edge e in g.Edges) {
            int slack = latestTime[e.V1.Number] -
                earliestTime[e.V0.Number] - (int)e.Weight;
            slackGraph.AddEdge(e.V0.Number, e.V1.Number,
                (int)e.Weight);
        }
        return DijkstrasAlgorithm(slackGraph, 0);
    }
}

Program: Critical path analysis--finding the critical paths.

The method first uses the EarliestTimeVisitor in a topological order traversal to compute the earliest event times which are recorded in the earliestTime array (lines 7-10). Next, the latest event times are computed and recorded in the latestTime array. Notice that this is done using a LatestTimeVisitor in a postorder depth-first traversal (lines 12-15). This is because a postorder depth-first traversal is equivalent to a topological order traversal in reverse!

Once the earliest and latest event times have been found, we can compute the slack time for each edge. In
the implementation shown, an edge-weighted graph is constructed that is isomorphic with the original event-node graph, but in which the edge weights are the slack times as given by Equation (lines 17-26). By constructing such a graph we can make use of Dijkstra's algorithm find the shortest path from start to finish since the shortest path must be the critical path (line 27).

The Dijkstra's Algorithm method given in Section returns its result in the form of a shortest-path graph. The shortest-path graph for the activity-node graph of Figure is shown in Figure. By following the path in this graph from vertex 9 back to vertex 0, we find that the critical path is \{A, C, D, E, G\}.

---

**Figure:** The critical path graph corresponding to Figure.

---

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Exercises

1. Consider the undirected graph $G_A$ shown in Figure 1. List the elements of $V$ and $E$. Then, for each vertex $v \in V$ do the following:
   1. Compute the in-degree of $v$.
   2. Compute the out-degree of $v$.
   3. List the elements of $A(v)$.
   4. List the elements of $I(v)$.

![Diagram of graphs](image)

**Figure:** Sample graphs.

2. Consider the directed graph $G_A$ shown in Figure 2.
   1. Show how the graph is represented using an adjacency matrix.
   2. Show how the graph is represented using adjacency lists.
3. Repeat Exercises 1 and 2 for the directed graph $G_B$ shown in Figure 3.
4. Consider a depth-first traversal of the undirected graph $G_A$ shown in Figure 4 starting from vertex $a$.
   1. List the order in which the nodes are visited in a preorder traversal.
   2. List the order in which the nodes are visited in a postorder traversal.
   Repeat this exercise for a depth-first traversal starting from vertex $d$. 
5. List the order in which the nodes of the undirected graph $G_A$ shown in Figure are visited by a \textit{breadth-first traversal} that starts from vertex $a$. Repeat this exercise for a breadth-first traversal starting from vertex $d$.

6. Repeat Exercises and for the directed graph $G_B$ shown in Figure.

7. List the order in which the nodes of the directed graph $G_B$ shown in Figure are visited by a \textit{topological order traversal} that starts from vertex $a$.

8. Consider an undirected graph $G = (V, E)$. If we use a $|V| \times |V|$ adjacency matrix $A$ to represent the graph, we end up using twice as much space as we need because $A$ contains redundant information. That is, $A$ is symmetric about the diagonal and all the diagonal entries are zero. Show how a one-dimensional array of length $|V|(|V| - 1)/2$ can be used to represent $G$. Hint: consider just the part of $A$ above the diagonal.

9. What is the relationship between the sum of the degrees of the vertices of a graph and the number of edges in the graph?

10. A graph with the maximum number of edges is called a \textit{fully connected graph}. Draw fully connected, undirected graphs that contain 2, 3, 4, and 5 vertices.

11. Prove that an undirected graph with $n$ vertices contains at most $n(n-1)/2$ edges.

12. Every tree is a directed, acyclic graph (DAG), but there exist DAGs that are not trees.
   1. How can we tell whether a given DAG is a tree?
   2. Devise an algorithm to test whether a given DAG is a tree.

13. Consider an acyclic, connected, undirected graph $G$ that has $n$ vertices. How many edges does $G$ have?

14. In general, an undirected graph contains one or more \textit{connected components}. A connected component of a graph $G$ is a subgraph of $G$ that is \textit{connected} and contains the largest possible number of vertices. Each vertex of $G$ is a member of exactly one connected component of $G$.
   1. Devise an algorithm to count the number of connected components in a graph.
   2. Devise an algorithm that labels the vertices of a graph in such a way that all the vertices in a given connected component get the same label and vertices in different connected components get different labels.

15. A \textit{source} in an directed graph is a vertex with zero in-degree. Prove that every DAG has at least one source.

16. What kind of DAG has a unique topological sort?

17. Under what conditions does a \textit{postorder} depth-first traversal of a DAG visit the vertices in \textit{reverse} topological order.

18. Consider a pair of vertices, $v$ and $w$, in a directed graph. Vertex $w$ is said to be \textit{reachable} from vertex $v$ if there exists a path in $G$ from $v$ to $w$. Devise an algorithm that takes as input a graph, $G = (V, E)$, and a pair of vertices, $v, w \in V$, and determines whether $w$ is reachable from $v$.

19. An \textit{Eulerian walk} is a path in an undirected graph that starts and ends at the same vertex \textit{and traverses every edge} in the graph. Prove that in order for such a path to exist, all the nodes must have even degree.

20. Consider the binary relation $\prec$ defined for the elements of the set $\{a, b, c, d\}$ as follows:
Exercises

\{a \prec b, a \prec c, b \prec c, b \prec d, c \prec d, d \prec a\}.

How can we determine whether \(\prec\) is a total order?

21. Show how the single-source shortest path problem can be solved on a DAG using a topological-order traversal. What is the running time of your algorithm?

22. Consider the directed graph \(G_C\) shown in Figure \(\square\). Trace the execution of Dijkstra's algorithm as it solves the single-source shortest path problem starting from vertex \(a\). Give your answer in a form similar to Table \(\square\).

\[\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\downarrow & \downarrow & \downarrow \\
1 & 10 & 8 \\
\text{d} & \text{e} & \text{f} \\
\downarrow & \downarrow & \downarrow \\
15 & 5 & 4 \\
\text{g} & \text{h} \\
\end{array}\]

Figure: Sample weighted graphs.

23. Dijkstra's algorithm works as long as there are no negative edge weights. Given a graph that contains negative edge weights, we might be tempted to eliminate the negative weights by adding a constant weight to all of the edge weights to make them all positive. Explain why this does not work.

24. Dijkstra's algorithm can be modified to deal with negative edge weights (but not negative cost cycles) by eliminating the known flag \(k\) and by inserting a vertex back into the queue every time its tentative distance \(d\) decreases. Explain why the modified algorithm works correctly. What is the running time of the modified algorithm?

25. Consider the directed graph \(G_D\) shown in Figure \(\square\). Trace the execution of Floyd's algorithm as it solves the all-pairs shortest path problem.

26. Prove that if the edge weights on an undirected graph are distinct, there is only one minimum-cost spanning tree.

27. Consider the undirected graph \(G_D\) shown in Figure \(\square\). Trace the execution of Prim's algorithm as it finds the minimum-cost spanning tree starting from vertex \(a\).

28. Repeat Exercise \(\square\) using Kruskal's algorithm.

29. Do Exercise \(\square\).
Projects

1. Devise a graph description language. Implement a method that reads the description of a graph and constructs a graph object instance. Your method should be completely generic—it should not depend on the graph implementation used.

2. Extend Project by writing a method that prints the description of a given graph object instance.

3. Complete the implementation of the GraphAsMatrix class introduced in Program by providing suitable definitions for the following operations: GraphAsMatrix (constructor), Purge, AddVertex, GetVertex, AddEdge, GetEdge, IsEdge, Vertices, Edges, IncidentEdges, and EmanatingEdges. Write a test program and test your implementation.

4. Repeat Project for the GraphAsLists class.

5. The DigraphAsMatrix class can be implemented by extending the GraphAsMatrix class introduced in Program to implement the Digraph interface defined in Program:

```csharp
public class DigraphAsMatrix : GraphAsMatrix, Digraph
{
    // ...
}
```

Implement the DigraphAsMatrix class by providing suitable definitions for the following methods: DigraphAsMatrix (constructor), Purge, AddEdge, GetEdge, IsEdge, and Edges. You must also have a complete implementation of the base class GraphAsMatrix (see Project). Write a test program and test your implementation.

6. Repeat Project for the DigraphAsLists class.

7. Add a method to the Digraph interface that returns the undirected graph which underlies the given digraph. Write an implementation of this method for the AbstractGraph class introduced in Program.

8. Devise an approach using an enumerator and a stack to perform a topological-order traversal by doing a postorder depth-first traversal in reverse.

9. The single-source shortest path problem on a DAG can be solved by visiting the vertices in topological order. Write an visitor for use with the TopologicalOrderTraversal method that solves the single-source shortest path problem on a DAG.

10. Devise and implement an method that transforms a vertex-weighted activity-node graph into an edge-weighted event-node graph.
Projects

11. Complete the implementation of the critical path analysis methods. In particular, you must implement the `LatestTimeVisitor` along the lines of the `EarliestTimeVisitor` defined in Program □.
C# and Object-Oriented Programming

This appendix is a brief overview of programming in C#. It identifies and describes the features of C# that are used throughout this text. This appendix is not a C# tutorial—if you are not familiar with C#, you should read one of the many C# programming books.

- Variables
- Parameter Passing
- Objects and Classes
- Nested Classes
- Inheritance and Polymorphism
- Exceptions

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Variables

A variable is a programming language abstraction that represents a storage location. A C# variable has the following attributes:

name
The name of a variable is the label used to identify a variable in the text of a program.

type
The type of a variable determines the set of values that the variable can have and the set of operations that can be performed on that variable.

value
The value of a variable is the content of the memory location(s) occupied by that variable. How the contents of the memory locations are interpreted is determined by the type of the variable.

lifetime
The lifetime of a variable is the interval of time in the execution of a C# program during which a variable is said to exist. Local variables exist as long as the method in which they are declared is active. Non-static fields of a class exist as long as the object of which they are members exist. Static fields of a class exist as long as the class in which they are defined remains loaded in the C# common language runtime.

scope
The scope of a variable is the range of statements in the text of a program in which that variable can be referenced.

Consider the C# variable declaration statement:

```csharp
int i = 57;
```

This statement defines a variable and binds various attributes with that variable. The name of the variable is `i`, the type of the variable is `int`, and its initial value is 57.

Some attributes of a variable, such as its name and type, are bound at compile time. This is called static binding. Other attributes of a variable, such as its value, may be bound at run time. This is called dynamic binding.

There are two kinds of C# variables—local variables and fields. A local variable is a variable declared inside a method. A field is a variable declared in some struct or class. (Classes are discussed in Section
Variables

1. The *type* of a C# variable is either one of the *value* types or it is a reference type.

- **Value Types**
- **References Types**

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Value Types

The C# value types are the simple types, the enumerated types, and structs. The simple types are `bool`, `char`, `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, and `decimal`. The C# language specification[22] defines the range of values for each simple type and the set of operations supported by each type. The enumerated types are declared using the C# `enum` construct. C# structs are declared using the `struct` construct.

Every variable of a value type is a distinct instance of that type. Thus, an assignment statement such as

```csharp
y = x;
```

takes the value of the variable `x` and copies that value into the variable `y`. After the assignment, `x` and `y` remain distinct instances that happen to have equal values.

A comparison of the the form

```csharp
if (x == y)
    { /* ... */ }
```

tests whether the values contained in the variables `x` and `y` are equal.
References Types

In C#, every variable that is not one of the value types is a reference type. Such a variable can be thought of as a reference to (or a pointer to) an object of the appropriate type.

Every object to which a reference variable refers is an instance of some C# class. In C#, class instances must be explicitly created. An instance of a class is created using the new operator like this:

```csharp
Foo f = new Foo();
```

If we follow this with an assignment statement such as

```csharp
Foo g = f;
```

then both f and g refer to the same object! Note that this is very different from what happens when you assign one value type to another.

A comparison of the the form

```csharp
if (f == g)
    { /* ... */ }
```

usually tests whether the f and g refer to the same object instances (provided the equality operator has not be overriden). If f and g refer to distinct object instances that happen to be equal, the test still fails. To test whether two distinct object instances are equal, it is necessary to invoke the Equals method like this:

```csharp
if (f.Equals(g))
    { /* ... */ }
```
• Null References
Null References

In C#, it is possible for a reference type variable to refer to nothing at all. A reference that refers to nothing at all is called a *null reference*. By default, an uninitialized reference is null.

We can explicitly assign the null reference to a variable like this:

```csharp
f = null;
```

Also, we can test explicitly for the null reference like this:

```csharp
if (f == null)
    { /* ... */ }
```
Parameter Passing

Parameter passing methods are the ways in which parameters are transferred between methods when one method calls another. C# provides two parameter passing methods--pass-by-value and pass-by-reference.

- Pass By Value
- Passing By Reference
- In and Out Parameters
- Passing Reference Types

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Consider the pair of C# methods defined in Program. On line 7, the method One calls the method Two. In general, every method call includes a (possibly empty) list of arguments. The arguments specified in a method call are called actual parameters. In this case, there is only one actual parameter—\( y \).

```csharp
1 public class Example
2 {
3     public static void One()
4     {
5         int x = 1;
6         Console.WriteLine(x);
7         Two(x);
8         Console.WriteLine(x);
9     }
10
11     public static void Two(int y)
12     {
13         y = 2;
14         Console.WriteLine(y);
15     }
16 }
```

**Program:** Example of pass-by-value parameter passing.

On line 11 the method Two is defined as accepting a single argument of type int called \( y \). The arguments which appear in a method definition are called formal parameters. In this case, the formal parameter is a value type.

The semantics of pass-by-value work like this: The effect of the formal parameter definition is to create a local variable of the specified type in the given method. For example, the method Two has a local variable of type int called \( y \). When the method is called, the values of the actual parameters are used
assigned to the *formal parameters* before the body of the method is executed.

Since the formal parameters give rise to local variables, if a new value is assigned to a formal parameter, that value has no effect on the actual parameter. Therefore, the output obtained produced by the method One defined in Program is:

```
1
2
1
```
Passing By Reference

Consider the methods `One` and `Two` defined in Program □. The only difference between this code and the code given in Program □ is the use of the keyword `ref` in the definition and use of the method `Two`. In this case, the formal parameter `y` is declared to be a `reference` to an `int`. Therefore, when the method is called, the actual parameter must also be a reference to an `int`. The expression `ref x` on line 7 provides a reference to the variable `x`. Thus, in this case the parameter passing is pass-by-reference.

```
public class Example
{
    public static void One()
    {
        int x = 1;
        Console.WriteLine(x);
        Two(ref x);
        Console.WriteLine(x);
    }

    public static void Two(ref int y)
    {
        y = 2;
        Console.WriteLine(y);
    }
}
```

Program: Example of pass-by-reference parameter passing.

A reference formal parameter is not a variable. When a method is called that has a reference formal parameter, the effect of the call is to associated the reference with the corresponding actual parameter. That is, the reference becomes an alternative name for the corresponding actual parameter.

A reference formal parameter can be used in the called method everywhere that a variable can be used. In particular, if the reference formal parameter is used where its value is required, it is the value of the
actual parameter that is obtained. Similarly, if the reference parameter is used where a reference is required, it is a reference to the actual parameter that is obtained. Therefore, the output obtained produced by the method One defined in Program is:

```
1
2
2
```

- The Trade-off
The Trade-off

Clearly, the parameter passing approach used constrains the functionality of the called method: When pass-by-value is used, the called method cannot modify the actual parameters; when pass-by-reference is used, the called method is able to modify the actual parameters. In addition, the two approaches have different time and space requirements that need to be understood in order to make the proper selection.

Pass-by-value creates a local variable and initializes that local variable by copying the value of the actual parameter. This means that space is used (on the stack) for the local variable and that time is taken to initialize that local variable. For small variables these penalties are insignificant. However, if the variable is a large struct, the time and space penalties may become prohibitive.

On the other hand, pass-by-reference does not create a local variable nor does it require the copying of the actual parameter. However, because of the way that it must be implemented, every time a reference formal parameter is used to access the corresponding actual parameter, a small amount of extra time is taken to dereference the reference. As a result, it is typically more efficient to pass small variables by value and large variables by reference.
In and Out Parameters

C# actually supports three different flavours of pass-by-reference: `ref`, `in`, and `out`. The latter two are special cases of the first.

An `in` parameter is a parameter that is passed by reference `in` to a method. Specifically, the actual parameter must already have a value and that value is assigned to the formal parameter of the method. Furthermore, it is not possible to assign a new value to an `in` parameter in the body of the method.

An `out` parameter is a parameter that is passed by reference `out` from a method. Specifically, the formal parameter must be assigned a value by the method before the method is called, and that value is then returned to the actual parameter of the calling method. It is not possible to use the value of an `out` parameter in the method body until a value has been assigned to that parameter.
Passing Reference Types

Program illustrates parameter passing of reference types. In this case, the variables x, y and z are all reference types. The type of x, y and z is Obj. Thus, x, y and z refer to instances of the Obj class defined on lines 3-6.

```csharp
public class Example
{
    public class Obj
    {
        public int field = 1;
    }

    public static void One()
    {
        Obj x = new Obj();
        Console.WriteLine(x.field);
        Two(x);
        Console.WriteLine(x.field);
        Three(ref x);
        Console.WriteLine(x.field);
    }

    public static void Two(Obj y)
    {
        y.field = 2;
        Console.WriteLine(y.field);
    }

    public static void Three(ref Obj z)
    {
        z = new Obj();
        Console.WriteLine(z.field);
    }
}
```
The semantics of parameter passing for reference types work exactly as they do for value types: In pass-by-value, the effect of the formal parameter definition is to create a local variable of the specified type in the given method. For example, the method Two has a local variable of type Obj called y. When the method is called, the *actual parameters* are assigned to the *formal parameters* before the body of the method is executed. Since x and y are reference types, when we assign y to x, we make them both refer to the same instance of the Obj class. Therefore, the Two method modifies the original Obj instance.

In pass-by-reference, the formal parameter ends up being a reference to a reference type. For example, in the method Three the formal parameter z refers to the reference parameter x. Thus, when we create a new Obj instance and assign it to z, the referenced variable x is modified.

The output obtained produced by the method One defined in Program is:

1
2
2
1
1
A C# class defines a data structure that contains fields, methods, and nested types. Every type in C# has a class that is directly or indirectly derived from the object class. The class of an object determines what it is and how it can be manipulated. A class encapsulates data, operations, and semantics. This encapsulation is like a contract between the implementer of the class and the user of that class.

The class construct is what makes C# an object-oriented language. A C# class definition groups a set of values with a set of operations. Classes facilitate modularity and information hiding. The user of a class manipulates object instances of that class only through the methods provided by that class.

It is often the case that different classes possess common features. Different classes may share common values; they may perform the same operations; they may support common interfaces. In C# such relationships are expressed using derivation and inheritance.
Class Members: Fields and Methods

A class groups a set of values and a set of operations. The values and the operations of a class are called its members. Fields implement the values and methods implement the operations.

Suppose we wish to define a class to represent complex numbers. The Complex class definition shown in Program illustrates how this can be done. Two fields, real and imag, are declared. These represent the real and imaginary parts of a complex number (respectively). Program also defines two properties, Real and Imag, each of which provide get and set accessors that can be used to access the real and imaginary parts of a complex number (respectively).

```csharp
public class Complex
{
    private double real;
    private double imag;

    public double Real
    {
        get { return real; }
        set { real = value; }
    }

    public double Imag
    {
        get { return imag; }
        set { imag = value; }
    }

    // ...
}
```

Program: Complex class fields, Real and Imag properties.

Every object instance of the Complex class contains its own fields. Consider the following variable declarations:
Complex c = new Complex();
Complex d = new Complex();

Both c and d refer to distinct instances of the Complex class. Therefore, each of them has its own real and imag field. The fields of an object are accessed using the dot operator. For example, c.real refers to the real field of c and d.imag refers to the imag field of d.

Program also defines the properties Real and Imag. In general, a property is an attribute of an instance of the class. Again, the dot operator is used to specify the object on which the operation is performed. For example, c.Real = 1.0 invokes the set accessor of the Real property on c and Console.WriteLine(d.Imag) invokes get accessor of the Imag property on d.
Constructors

A constructor is a method that has the same name as its class (and which has no return value). Three constructors are defined in Program . The purpose of a constructor is to initialize an object. A constructor is invoked whenever a new instance of a class is created using the new operator.

Consider the following sequence of variable declarations:

```csharp
Complex c = new Complex();     // calls Complex ()
Complex d = new Complex(2.0);  // calls Complex (double)
Complex i = new Complex(0, 1); // calls Complex (double, double)
```

Consider the constructor that takes two double arguments, x and y (lines 6-10). This constructor initializes the complex number by assigning x and y to the real and imag fields, respectively.

- The No-Arg Constructor

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
The No-Arg Constructor

The constructor which takes no arguments is called the no-arg constructor. For example, the no-arg constructor is invoked when a variable is declared like this:

```csharp
Complex c = new Complex();
```

If there are no constructors defined in a C# class, the C# compiler provides a default no-arg constructor. The default no-arg constructor does nothing. The fields simply retain their initial, default values.

```csharp
public class Complex
{
    private double real;
    private double imag;

    public Complex(double x, double y)
    {
        real = x;
        imag = y;
    }

    public Complex() : this(0, 0)
    {
    }

    public Complex(double x) : this(x, 0)
    {
        // ...
    }
}
```

Program: Complex constructors.

Program gives an implementation for the no-arg constructor of the Complex class (lines 12-13). This constructor uses the an initializer called this. In C# one constructor can invoke another constructor by calling using the this initializer. In this case, the no-arg constructor invokes the two-arg
The No-Arg Constructor

constructor to set both real and imag fields to zero.
Properties and Accessors

A C# property defines one or two methods for accessing an object. A property that provides a `get` accessor can be used to access the contents of an object. A property that provides a `set` accessor can be used to modify an object.

An `get` accessor is a method that accesses the contents of an object but does not modify that object. In the simplest case, a `get` accessor just returns the value of one of the fields. In general, a `get` accessor performs some computation using the fields as long as that computation does not modify any of the fields.

A `set` accessor is a method that modifies an object. A method that modifies an object is also known as a `mutator`. In the simplest case, a `set` accessor modifies a single field of an object. In general, a `set` accessor may modify any number of the fields of an object.

Program defines two more properties of the `Complex` class--`getR` and `Theta`. The `R` and `Theta` properties provide `get` and `set` accessors that access a complex number using polar coordinates.
Properties and Accessors

Program: Complex class R and Theta properties.

By defining suitable accessors, it is possible to hide the implementation of the class from the user of that class. Consider the following statements:

```csharp
Console.WriteLine(c.real);
Console.WriteLine(c.Real);
```

The first statement depends on the implementation of the Complex class. If we change the implementation of the class from the one given (which uses rectangular coordinates) to one that uses
polar coordinates, then the first statement above must also be changed. On the other hand, the second statement does not need to be modified, provided we reimplement the Real property when we switch to polar coordinates.
Member Access Control

Every member of a class, be it a field or a method, has an access control attribute which affects the manner in which that member can be accessed. The members of a class can be private, public, protected, internal, or protected internal. For example, the fields real and imag declared in Program are both private. Private members can be used only by methods of the class in which the member is declared.

On the other hand, public members of a class can be used by any method in any class. All of the operations defined in Programs and all declared to be public.

In effect, the public part of a class defines the interface to that class and the private part of the class encapsulates the implementation of that class. By making the implementation of a class private, we ensure that the code which uses the class depends only on the interface and not on the implementation of the class. Furthermore, we can modify the implementation of the class without affecting the code of the user of that class.

Protected members are similar to private members. That is, they can be used by methods of the class in which the member is declared. In addition, protected members can also be used by methods of all the classes derived from the class in which the member is declared. The protected category is discussed again in Section.

An internal member can be accessed by any method in the same program in which the member is declared. Finally, a protected internal member can be accessed by methods in the class in which the member is declared, by methods in classes derived from the class in which the member is declared, and any method in the same program in which the member is declared.
Operator Overloading

Program illustrates operator overloading in C#. Operator overloading allows the programmer to use the built-in operators for user-defined types.

```csharp
public class Complex
{
    private double real;
    private double imag;

    public static Complex operator +(Complex c1, Complex c2)
    {
        return new Complex(c1.Real+c2.Real, c1.Imag+c2.Imag);
    }

    public static Complex operator -(Complex c1, Complex c2)
    {
        return new Complex(c1.Real-c2.Real, c1.Imag-c2.Imag);
    }
    // ...
}
```

**Program:** Complex class operators.

To overload the built-in + and * operators so that they may be used with Complex operands, we define static methods called operator+ and operator*. Given Complex variables c, d and e the expression c+d*e calls the method operator* to compute the product of d and e, and then calls the method operator+ to compute the final sum.
Nested Classes

In C# it is possible to define one class inside another. A class defined inside another one is called a nested class.

Consider the following C# code fragment:

```csharp
public class A
{
    int y;

    public static class B
    {
        int x;

        void F() {}
    }
}
```

This fragment defines the class A which contains the nested class B.

A nested class behaves like any "outer" class. It may contain methods and fields, and it may be instantiated like this:

```csharp
A.B obj = new A.B();
```

This statement creates a new instance of the nested class B. Given such an instance, we can invoke the F method in the usual way:

```csharp
obj.F();
```

Note, it is not necessarily the case that an instance of the outer class A exists even when we have created an instance of the inner class. Similarly, instantiating the outer class A does not create any instances of the inner class B.

The methods of a nested class may access all the members (fields or methods) of the nested class but
they can access only static members (fields or methods) of the outer class. Thus, \( F \) can access the field \( x \), but it cannot access the field \( y \).
Inheritance and Polymorphism

- Derivation and Inheritance
- Polymorphism
- Multiple Inheritance
- Run-Time Type Information and Casts

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.
Derivation and Inheritance

This section reviews the concept of a derived class. Derived classes are an extremely useful feature of C# because they allow the programmer to define new classes by extending existing classes. By using derived classes, the programmer can exploit the commonalities that exist among the classes in a program. Different classes can share values, operations, and interfaces.

Derivation is the definition of a new class by extending an existing class. The new class is called the derived class and the existing class from which it is derived is called the base class. In C# there can be only one base class (single inheritance).

Consider the Person class defined in Program and the Parent class defined in Program. Because parents are people too, the Parent class is derived from the Person class. Derivation in C# is indicated by a colon followed by the name of the base class in the declaration of the derived class.
public class Person
{
    public enum Sex
    {
        MALE,
        FEMALE
    };

    protected string name;
    protected Sex sex;

    public Person(string name, Sex sex)
    {
        this.name = name;
        this.sex = sex;
    }

    public override string ToString()
    {
        return name;
    }
}

Program: Person class.

public class Parent : Person
{
    protected Person[] children;

    public Parent(string name, Sex sex,
                  Person[] children) : base(name, sex)
    {
        this.children = children;
    }

    public Person GetChild(int i)
    {
        return children[i];
    }

    public override string ToString()
    {
        /* ... */
    }
}

Program: Parent class.
A derived class *inherits* all the members of its base class. That is, the derived class contains all the fields contained in the base class and the derived class supports all the same operations provided by the base class. For example, consider the following variable declarations:

```csharp
Person p = new Person();
Parent q = new Parent();
```

Since `p` is a `Person`, it has the fields `name` and `sex` and method `ToString`. Furthermore, since `Parent` is derived from `Person`, then the object `q` also has the fields `name` and `sex` and method `toString`.

A derived class can *extend* the base class in several ways: New fields can be defined, new methods can be defined, and existing methods can be *overridden*. For example, the `Parent` class adds the field `children` and the method `GetChild`.

If a method is defined in a derived class that has exactly the same *signature* (name and types of arguments) as a method in a base class, the method in the derived class *overrides* the one in the base class. For example, the `ToString` method in the `Parent` class overrides the `ToString` method in the `Person` class. Therefore, `p.ToString()` invokes `Person.ToString`, whereas `q.ToString(...)` invokes `Parent.ToString`. Note that C# requires the use of the keyword `overrides` in the declaration of a method that overrides an inherited method.

An instance of a derived class can be used anywhere in a program where an instance of the base class may be used. For example, this means that a `Parent` may be passed as an actual parameter to a method in which the formal parameter is a `Person`.

It is also possible to assign a derived class object to a base class variable like this:

```csharp
Person p = new Parent();
```

However, having done so, it is not possible to call `p.GetChild(...)`, because `p` is a `Person` and a `Person` is not necessarily a `Parent`.

---

- **Derivation and Access Control**
Members of a class can be private, public or protected. As explained in Section □, private members are accessible only by methods of the class in which the member is declared. In particular, this means that the methods of a derived class cannot access the private members of the base classes even though the derived class has inherited those members! On the other hand, if we make the members of the base class public, then all classes can access those members directly, not just derived classes.

This is where protected access control comes in. Protected members can be used by methods of the class in which the member is declared as well as by methods of all the classes derived from the class in which the member is declared.
Polymorphism

Polymorphism literally means "having many forms." Polymorphism arises when a set of distinct classes share a common interface. Because the derived classes are distinct, their implementations may differ. However, because the derived classes share a common interface, instances of those classes are used in exactly the same way.

- Interfaces
- Abstract Methods and Abstract Classes
- Method Resolution
- Abstract Classes and Concrete Classes
- Algorithmic Abstraction
Interfaces

Consider a program for creating simple drawings. Suppose the program provides a set of primitive graphical objects, such as circles, rectangles, and squares. The user of the program selects the desired objects, and then invokes commands to draw, to erase, or to move them about. Ideally, all graphical objects support the same set of operations. Nevertheless, the way that the operations are implemented varies from one object to the next.

We implement this as follows: First, we define a C# interface which represents the common operations provided by all graphical objects. A C# interface declares a set of methods. An object that supports an interface must provide

Program defines the GraphicsPrimitives interface comprised of three methods, Draw, Erase, and MoveTo, the methods declared in the interface.

```
1 public interface GraphicsPrimitives
2 {
3     void Draw();
4     void Erase();
5     void MoveTo(Point p);
6 }
```

Program: GraphicsPrimitives interface.

The Draw method is invoked in order to draw a graphical object. The Erase method is invoked in order to erase a graphical object. The MoveTo method is used to move an object to a specified position in the drawing. The argument of the MoveTo method is a Point struct. Program defines the Point struct which represents a position in a drawing.
public struct Point
{
    int x;
    int y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    // ...
}

Program: Point struct.
Abstract Methods and Abstract Classes

Consider the `GraphicalObject` class defined in Program 1. The `GraphicalObject` class implements the `GraphicsPrimitives` interface. This is indicated by a colon followed by the name of the interface in the declaration of the abstract class.

```
public abstract class GraphicalObject : GraphicsPrimitives
{
    protected Point center;

    public GraphicalObject(Point p)
    { this.center = p; }

    public abstract void Draw();

    public virtual void Erase()
    {
        SetPenColor(BACKGROUND_COLOR);
        Draw();
        SetPenColor(FOREGROUND_COLOR);
    }

    public virtual void MoveTo(Point p)
    {
        Erase();
        center = p;
        Draw();
    }
}
```

Program: `GraphicalObject` class.

The `GraphicalObject` class has a single field, `center`, which is a `Point` that represents the position in a drawing of the center-point of the graphical object. The constructor for the
Abstract Methods and Abstract Classes

GraphicalObject class takes as its argument a Point and initializes the center field accordingly.

Program shows a possible implementation for the Erase method: In this case we assume that the image is drawn using an imaginary pen. Assuming that we know how to draw a graphical object, we can erase the object by changing the color of the pen so that it matches the background color and then redrawing the object.

Once we can erase an object as well as draw it, then moving it is easy. Just erase the object, change its center point, and then draw it again. This is how the MoveTo method shown in Program is implemented.

We have seen that the GraphicalObject class provides implementations for the Erase and MoveTo methods. However, the GraphicalObject class does not provide an implementation for the Draw method. Instead, the method is declared to be abstract. We do this because until we know what kind of object it is, we cannot possibly know how to draw it!

Consider the Circle class defined in Program. The Circle class extends the GraphicalObject class. Therefore, it inherits the field center and the methods Erase and MoveTo. The Circle class adds an additional field, radius, and it overrides the Draw method. The body of the Draw method is not shown in Program. However, we shall assume that it draws a circle with the given radius and center point.

```java
public class Circle : GraphicalObject
{
    protected int radius;

    public Circle(Point p, int r) : base(p)
    {
        radius = r;
    }

    public override void Draw()
    {
        /* ... */
    }
}
```

Program: Circle class.

Using the Circle class defined in Program we can write code like this:

```java
Circle c = new Circle(new Point(0, 0), 5);
c.Draw();
c.MoveTo(new Point(10, 10));
```
c.Drase();

This code sequence declares a circle object with its center initially at position (0,0) and radius 5. The circle is then drawn, moved to (10,10), and then erased.

Program defines the Rectangle class and Program defines the Square class. The Rectangle class also extends the GraphicalObject class. Therefore, it inherits the field center and the methods Erase and MoveTo. The Rectangle class adds two additional fields, height and width, and it overrides the Draw method. The body of the Draw method is not shown in Program. However, we shall assume that it draws a rectangle with the given dimensions and center point.

```
public class Rectangle : GraphicalObject
{
    protected int height;
    protected int width;

    public Rectangle(Point p, int ht, int wid) : base(p)
    {
        height = ht;
        width = wid;
    }

    public override void Draw()
    { /* ... */ }
}
```

Program: Rectangle class.

The Square class extends the Rectangle class. No new fields or methods are declared--those inherited from GraphicalObject or from Rectangle are sufficient. The constructor simply arranges to make sure that the height and width of a square are equal!

```
public class Square : Rectangle
{
    public Square(Point p, int wid) : base(p, wid, wid)
    {
    }
}
```

Program: Square class.
Method Resolution

Consider the following sequence of instructions:

```csharp
GraphicalObject g1 = new Circle(new Point (0,0), 5);
GraphicalObject g2 = new Square(new Point (0,0), 5);
g1.Draw();
g2.Draw();
```

The statement `g1.Draw()` calls `Circle.Draw` whereas the statement `g2.Draw()` calls `Rectangle.Draw`.

It is as if every object of a class "`knows" the actual method to be invoked when a method is called on that object. E.g, a `Circle` "`knows" to call `Circle.Draw`, `GraphicalObject.Erase` and `GraphicalObject.MoveTo`, whereas a `Square` "`knows" to call `Rectangle.Draw`, `GraphicalObject.Erase` and `GraphicalObject.MoveTo`.

In this way, C# ensures that the "correct" method is actually called, regardless of how the object is accessed. Consider the following sequence:

```csharp
Square s = new Square(new Point(0,0), 5);
Rectangle r = s;
GraphicalObject g = r;
```

Here `s`, `r` and `g` all refer to the same object, even though they are all of different types. However, because the object is a `Square`, `s.Draw()`, `r.Draw()` and `g.Draw()` all invoke `Rectangle.Draw`.

Abstract Classes and Concrete Classes

In C# an abstract class is one that does not provide implementations for all its methods. A class must be declared abstract if any of the methods in that class are abstract. For example, the GraphicalObject class defined in Program is declared abstract because its Draw method is abstract.

An abstract class is meant to be used as the base class from which other classes are derived. The derived class is expected to provide implementations for the methods that are not implemented in the base class. A derived class that implements all the missing functionality is called a concrete class.

In C# it is not possible to instantiate an abstract class. For example, the following declaration is illegal:

```csharp
GraphicalObject g = new GraphicalObject(new Point(0, 0)); // Wrong.
```

If we were allowed to declare `g` in this way, then we could attempt to invoke the non-existent method `g.Draw()`.
Algorithmic Abstraction

Abstract classes can be used in many interesting ways. One of the most useful paradigms is the use of an abstract class for algorithmic abstraction. The Erase and MoveTo methods defined in Program are examples of this.

The Erase and MoveTo methods are implemented in the abstract class GraphicalObject. The algorithms implemented are designed to work in any concrete class derived from GraphicalObject, be it Circle, Rectangle or Square. In effect, we have written algorithms that work regardless of the actual class of the object. Therefore, such algorithms are called abstract algorithms.

Abstract algorithms typically invoke abstract methods. For example, both MoveTo and Erase ultimately invoke Draw to do most of the actual work. In this case, the derived classes are expected to inherit the abstract algorithms MoveTo and Erase and to override the abstract method Draw. Thus, the derived class customizes the behavior of the abstract algorithm by overriding the appropriate methods. The C# method resolution mechanism ensures that the ``correct" method is always called.
Multiple Inheritance

In C# a class can be derived from only one base class. That is, the following declaration is not allowed:

```csharp
class A {}
class B {}
class C : A, B // Wrong;
{
}
```

Nevertheless, it is possible for a class to extend a base class and to implement one or more interfaces:

```csharp
class A {}
interface D {}
interface E {}
class C : A, D, E
{
}
```

The derived class `C` inherits the members of `A` and it implements all the methods defined in the interfaces `D` and `E`.

It is possible to use derivation in the definition of interfaces. And in C# it is possible for an interface to extend more than one base interface:

```csharp
interface E {}
interface F {}
interface D : E, F
{
}
```

In this case, the derived interface `D` comprises all the methods inherited from `E` and `F` as well as any new methods declared in the body of `D`.
Run-Time Type Information and Casts

Consider the following declarations which make use of the Rectangle and Square classes defined in Programs

```csharp
Rectangle r = new Rectangle(new Point(0,0), 5, 10);
Square s = new Square(new Point(0,0), 15);
```

Clearly, the assignment

```csharp
r = s;
```

is valid because Square is derived from Rectangle. That is, since a Square is a Rectangle, we may assign s to r.

On the other hand, the assignment

```csharp
s = r; // Wrong.
```

is not valid because a Rectangle instance is not necessarily a Square.

Consider now the following declarations:

```csharp
Rectangle r = new Square(new Point(0,0), 20);
Square s;
```

The assignment s=r is still invalid because r is a Rectangle, and a Rectangle is not necessarily a Square, despite the fact that in this case it actually is!

In order to do the assignment, it is necessary to convert the type of r from a Rectangle to a Square. This is done in C# using a cast operator:

```csharp
s = (Square)r;
```

The C# common language runtime checks at run-time that r actually does refer to a Square and if it
does not, the operation throws a ClassCastException. (Exceptions are discussed in Section ☐).

To determine the type of the object to which \( r \) refers, we must make use of run-time type information. In C# the is operator can be used to test whether a particular object is an instance of some class. Thus, we can determine the class of an object like this:

\[
\text{if (} r \text{ is Square)} \\
\quad s = \text{(Square)} r;
\]

This code does not throw an exception because the cast operation is only attempted when \( r \) actually is a Square.

Alternatively, we may use the as operator to do the conversion like this:

\[
s = r \text{ as Square;}
\]

The as operator returns null (and does not throw an exception) if the object to which \( r \) refers is not a Square.
Exceptions

Sometimes unexpected situations arise during the execution of a program. Careful programmers write code that detects errors and deals with them appropriately. However, a simple algorithm can become unintelligible when error-checking is added because the error-checking code can obscure the normal operation of the algorithm.

Exceptions provide a clean way to detect and handle unexpected situations. When a program detects an error, it throws an exception. When an exception is thrown, control is transferred to the appropriate exception handler. By defining a method that catches the exception, the programmer can write the code to handle the error.

In C#, an exception is an object. All exceptions in C# are ultimately derived from the base class called System.Exception. For example, consider the class A defined in Program 1. Since the A class extends the System.Exception class, A is an exception that can be thrown.
A method throws an exception by using the `throw` statement: The `throw` statement is similar to a `return` statement. A `return` statement represents the normal termination of a method and the object returned matches the return value of the method. A `throw` statement represents the abnormal termination of a method and the object thrown represents the type of error encountered. The `F` method in Program  throws an `A` exception.

Exception handlers are defined using a `try` block: The body of the `try` block is executed either until an exception is thrown or until it terminates normally. One or more exception handlers follow a `try` block. Each exception handler consists of a `catch` clause which specifies the exceptions to be caught, and a block of code, which is executed when the exception occurs. When the body of the `try` block throws an exception for which an exception is defined, control is transferred to the body of the exception handler.

In this example, the exception thrown by the `F` method is caught by the `G` method. In general when an exception is thrown, the chain of methods called is searched in reverse (from caller to callee) to find the closest matching `catch` statement. When a program throws an exception that is not caught, the program terminates.

```csharp
public class Example
{
    public class A : System.Exception
    {
    }

    static void F()
    {
        throw new A();
    }

    static void G()
    {
        try
        {
            F();
        }
        catch (A exception)
        {
            // ...
        }
    }
}
```

Program: Using exceptions in C#.
Class Hierarchy Diagrams

**AbstractClass** specializes **ConcreteClass**

**Interface** specializes **Interface**

**Interface** realizes **ConcreteClass**

---

**Figure:** Key for the class hierarchy diagrams.
Figure: Complete class hierarchy diagram.
### Character Codes

<table>
<thead>
<tr>
<th>bits 6-3</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>SOH</td>
<td>STX</td>
<td>ETX</td>
<td>EOT</td>
<td>ENQ</td>
<td>ACK</td>
<td>BEL</td>
</tr>
<tr>
<td>1</td>
<td>BS</td>
<td>HT</td>
<td>NL</td>
<td>VT</td>
<td>NP</td>
<td>CR</td>
<td>SO</td>
<td>SI</td>
</tr>
<tr>
<td>2</td>
<td>DLE</td>
<td>DC1</td>
<td>DC2</td>
<td>DC3</td>
<td>DC4</td>
<td>NAK</td>
<td>SYN</td>
<td>ETB</td>
</tr>
<tr>
<td>3</td>
<td>CAN</td>
<td>EM</td>
<td>SUB</td>
<td>ESC</td>
<td>FS</td>
<td>GS</td>
<td>RS</td>
<td>US</td>
</tr>
<tr>
<td>4</td>
<td>SP</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
<td>'</td>
</tr>
<tr>
<td>5</td>
<td>(</td>
<td>)</td>
<td>*</td>
<td>+</td>
<td>,</td>
<td>-</td>
<td>.</td>
<td>/</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>:</td>
<td>;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>?</td>
</tr>
<tr>
<td>010</td>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
</tr>
<tr>
<td>011</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>012</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
</tr>
<tr>
<td>013</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>[</td>
<td>\</td>
<td>]</td>
<td>^</td>
<td>_</td>
</tr>
<tr>
<td>014</td>
<td>\</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
</tr>
<tr>
<td>015</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td>o</td>
</tr>
<tr>
<td>016</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
</tr>
<tr>
<td>017</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>{</td>
<td></td>
<td></td>
<td>}</td>
<td>~</td>
</tr>
</tbody>
</table>

**Table:** 7-bit ASCII character set.
References


References


15 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.


Stephen R. Schach. *Classical and Object-Oriented Software Engineering*. Irwin, Chicago, IL,
References

42

43

44

45

46

47

48

49

50
The notation \( \lfloor x \rfloor \) denotes the *floor function*, which is defined as follows: For any real number \( x \), \( \lfloor x \rfloor \) is the greatest integer less than or equal to \( x \). While we are on the subject, there is a related function, the *ceiling function*, written \( \lceil x \rceil \). For any real number \( x \), \( \lceil x \rceil \) is the smallest integer greater than or equal to \( x \).

In fact, we would normally write \( \lfloor x \rfloor \), but we have not yet seen the \( \lfloor \rfloor \) notation which is introduced in Chapter [ ].
Guillaume François Antoine de L'Hôpital, marquis de Sainte-Mesme, is known for his rule for computing limits which states that if \( \lim_{n \to a} f(n) = \lim_{n \to a} g(n) = \infty \) and \( \lim_{n \to a} f(n) = \lim_{n \to a} g(n) = -\infty \), then

\[
\lim_{n \to a} \frac{f(n)}{g(n)} = \lim_{n \to a} \frac{f'(n)}{g'(n)},
\]

where \( f(n) \) and \( g(n) \) are the first derivatives with respect to \( n \) of \( f(n) \) and \( g(n) \), respectively. The rule is also effective if \( \lim_{n \to a} f(n) = \lim_{n \to a} g(n) = 0 \).
Functions which are commensurate are functions which can be compared one with the other.
This notion of the looseness (tightness) of an asymptotic bound is related to but not exactly the same as that given in Definition □.
Fibonacci numbers are named in honor of Leonardo Pisano (Leonardo of Pisa), the son of Bonaccio (in Latin, Filius Bonaccii), who discovered the series in 1202.
numbers.

These running times were measured on an Intel Pentium III, which has a 1 GHz clock and 256MB RAM under the WindowsME operating system. The programs were compiled using the C# compiler provided with the Microsoft .NET beta SDK (csc) and run under the Microsoft common language runtime.

...represents.

The *address* attribute is sometimes called its *l-value* and the *value* attribute is sometimes called is *r-value*. This terminology arises from considering the semantics of an assignment statement such as $y = x$. The meaning of such as statement is ```take the value of variable $x$ and store it in...```
memory at the *address* of variable y." So, when a variable appears on the right-hand-side of an assignment, we use its r-value; and when it appears on the left-hand-size, we use its l-value.

---

For a complete list of the methods defined in the `System.Object` class, you should consult *The C# Language Specifications*[22].
The word *deque* is usually pronounced like ``deck'' and sometimes like ``deek.''

...NAME=7497> .
A total order is a relation, say $\prec$, defined on a set of elements, say $\mathbb{S}$, with the following properties:

1. For all pairs of elements $x, y \in \mathbb{S}$, such that $x \neq y$, exactly one of either $x \prec y$ or $y \prec x$ holds. (All elements are commensurate).
2. For all triples $x, y, z \in \mathbb{S}$, (The relation is transitive).

(See also Definition 4).
825

This is the Swedish word for the number two. The symbol å in the Unicode character set can be represented in a C# program using the Unicode escape \u00E5.
I have been advised that a book without sex will never be a best seller. ``Sex'' is the Swedish word for the number six.
Two numbers $x$ and $y$ are *relatively prime* if there is no number other than one that divides both $x$ and $y$ evenly.

...prime

...i.

What else would it be?
Isomorphic is a fancy word that means being of identical or similar form or shape or structure.
...Landis

Russian mathematicians G. M. Adel'son-Vel'skiĭ and E. M. Landis published this result in 1962.
...trees.

Obviously since B-Trees are $M$-way trees, the "B" in $B$-Tree does not stand for binary. B-Trees were invented by R. Bayer and E. McCreight in 1972, so the "B" either stands for balanced or Bayer—take your pick.
Isaac Newton discovered the binomial theorem in 1676 but did not publish a proof. Leonhard Euler attempted a proof in 1774. Karl Friedrich Gauss produced the first correct proof in 1812.

Stirling numbers of the second kind are given by the formula...
where \( n > 0 \) and 1620#1620.

...structures.

Mark-and-sweep garbage collection is described by John McCarthy in a paper on the LISP language published in 1960.
The reader may find it instructive to compare Program □ with Program □ and Program □.
...space.

The reader may find it instructive to compare Program □ with Program □ and Program □.
The table is named in honor of *Blaise Pascal* who published a treatise on the subject in 1653.
Prime numbers of the form 1917#1917 are known as *Mersenne primes*.

For convenience, we use the notation 1921#1921 to denote 1922#1922.
Unfortunately, the fame of bubble sort exceeds by far its practical value.
...zero.

There is also the symmetrical case in which $i$ is always $n-1$. 
Footnotes

Copyright © 2001 by Bruno R. Preiss, P.Eng. All rights reserved.