MICHAEL McMILLAN

Data Structures and Algorithms Using Visual Basic.NET

CAMBRIDGE

CAMBRIDGE

more information - www.cambridge.org/9780521547659

Data Structures and Algorithms Using Visual Basic.NET

This is the first Visual Basic.NET (VB.NET) book to provide a comprehensive discussion of the major data structures and algorithms. Here, instead of having to translate material on C++ or Java, the professional or student VB.NET programmer will find a tutorial on how to use data structures and algorithms and a reference for implementation using VB.NET for data structures and algorithms from the .NET Framework Class Library as well as those that must be developed by the programmer.

In an object-oriented fashion, the author presents arrays and ArrayLists, linked lists, hash tables, dictionaries, trees, graphs, and sorting and searching as well as more advanced algorithms, such as probabilistic algorithms and dynamic programming. His approach is very practical, for example using timing tests rather than Big O analysis to compare the performance of data structures and algorithms.

This book can be used in both beginning and advanced computer programming courses that use the VB.NET language and, most importantly, by the professional Visual Basic programmer.

Michael McMillan is Instructor of Computer Information Systems at Pulaski Technical College. With more than twenty years of experience in the computer industry, he has written numerous articles for trade journals such as *Software Development* and *Windows NT Systems*. He is the author of *Perl from the Ground Up* and *Object-Oriented Programming with Visual Basic.Net* and coauthor of several books.

Data Structures and Algorithms Using Visual Basic.NET

Michael McMillan
Pulaski Technical College

Cambridge University Press

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 2RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521547659

First published in print format 2005

ISBN-13 978-0-511-11366-6 eBook (NetLibrary)
ISBN-10 0-511-11366-8 eBook (NetLibrary)

ISBN-13 978-0-521-54765-9 paperback
ISBN-10 0-521-54765-2 paperback

Contents

Preface

The Visual Basic.NET (VB.NET) programming language is not usually associated with the study of data structures and algorithms. The primary reason for this must be because most university and college computer science departments don't consider VB.NET to be a "serious" programming language that can be used to study serious topics. This is primarily a historical bias based on Basic's past as a "nonprogrammer's" language often taught to junior high, senior high, and liberal arts college students, but not to computer science or computer engineering majors.

The present state of the language, however, aligns it with other, more serious programming languages, most specifically Java. VB.NET, in its current form, contains everything expected in a modern programming language, from true object-oriented features to the .NET Framework library, which rivals the Java libraries in both depth and breadth.

Included in the .NET Framework library is a set of collection classes, which range from the Array, ArrayList, and Collection classes, to the Stack and Queue classes, to the Hashtable and the SortedList classes. Students of data structures and algorithms can now see how to use a data structure before learning how to implement it. Previously, an instructor had to discuss the concept of, say, a stack, abstractly until the complete data structure was constructed. Instructors can now show students how to use a stack to perform some computations, such as number base conversions, demonstrating the utility of the data structure immediately. With this background, students can then go back and learn the fundamentals of the data structure (or algorithm) and even build their own implementation.

This book is written primarily as a practical overview of the data structures and algorithms all serious computer programmers need to know and

understand. Given this, there is no formal analysis of the data structures and algorithms covered in the book. Hence, there is not a single mathematical formula and not one mention of Big O analysis (for the latter the reader is referred to any of the books listed in the bibliography). Instead, the various data structures and algorithms are presented as problem-solving tools. We use simple timing tests to compare the performance of the data structures and algorithms discussed in the book.

Prerequisites

There are very few prerequisites for this book. The reader should be competent in one or more programming languages, preferably VB.NET, though a course or two in Java will serve as well. C/C++ programmers should not have too much trouble picking up the language. There are no mathematical prerequisites since we don't take a formal approach in the book.

Chapter-by-Chapter Organization

The Introduction provides an overview of object-oriented programming using VB.NET and introduces the benchmark tool used for comparing the performance of the data structures and algorithms studied in the book. This tool is a Timing class developed by the author as a practical means for timing code in the .NET environment.

Chapter 1 introduces the reader to the concept of the data structure as a collection of data. The concepts of linear and nonlinear collections are introduced. The Collection class is demonstrated.

Chapter 2 provides a review of how arrays are constructed in VB.NET, along with demonstrating the features of the Array class. The Array class encapsulates many of the functions associated with arrays (UBound, LBound, and so on) into a single package. Arraylists are special types of arrays that provide dynamic resizing capabilities.

Chapter 3 gives an introduction to the basic sorting algorithms, such as the bubble sort and the insertion sort, and Chapter 4 examines the most fundamental algorithms for searching memory, the sequential and binary searches.

Two classic data structures are examined in Chapter 5—the stack and the queue. This chapter emphasizes the practical use of these data structures in solving everyday problems in data processing. Chapter 6 covers the BitArray

class, which can be used to efficiently represent a large number of integer values, such as test scores.

Strings are not usually covered in a data structures book, but Chapter 7 covers strings, the String class, and the StringBuilder class. We feel that because so much data processing in VB.NET is performed on strings, the reader should be exposed to the special techniques found in the two classes. Chapter 8 examines the use of regular expressions for text processing and pattern matching. Regular expressions often provide more power and efficiency than can be had with more traditional string functions and methods.

Chapter 9 introduces the reader to the use of dictionaries as data structures. Dictionaries, and the different data structures based on them, store data as key/value pairs. This chapter shows the reader how to create his or her own classes based on the DictionaryBase class, which is an abstract class. Chapter 10 covers hash tables and the Hashtable class, which is a special type of dictionary that uses a hashing algorithm for storing data internally.

Another classic data structure, the linked list, is covered in Chapter 11. Linked lists are not as important a data structure in VB.NET as they are in a pointer-based language such as C++, but they still play a role in VB.NET programming. Chapter 12 introduces the reader to yet another classic data structure—the binary tree. A specialized type of binary tree, the binary search tree, comprises the primary topic of the chapter. Other types of binary trees are covered in Chapter 15.

Chapter 13 shows the reader how to store data in sets, which can be useful in situations when only unique data values can be stored in the data structure. Chapter 14 covers more advanced sorting algorithms, including the popular and efficient QuickSort, which forms the basis for most of the sorting procedures implemented in the .NET Framework library. Chapter 15 looks at three data structures that prove useful for searching when a binary search tree is not called for: the AVL tree, the red–black tree, and the skip list.

Chapter 16 discusses graphs and graph algorithms. Graphs are useful for representing many different types of data, especially networks. Finally, Chapter 17 introduces the reader to what are really algorithm design techniques—dynamic algorithms and greedy algorithms.

Acknowledgments

There are several different groups of people who must be thanked for helping me finish this book. First, I owe thanks to a certain group of students who

first sat through my lectures on developing data structures and algorithms in VB.NET. These students include (not in any particular order): Matt Hoffman, Ken Chen, Ken Cates, Jeff Richmond, and Gordon Caffey. Also, one of my fellow instructors at Pulaski Technical College, Clayton Ruff, sat through many of the lectures and provided excellent comments and criticism. I also have to thank my department chair, David Durr, for providing me with an excellent environment for researching and writing. I also need to thank my family for putting up with me while I was preoccupied with research and writing. Finally, I offer many thanks to my editor at Cambridge, Lauren Cowles, for putting up with my many questions and topic changes, and her assistant, Katie Hew, who made the publication of this book as smooth a process as possible.

Introduction

In this preliminary chapter, we introduce a couple of topics we'll be using throughout the book. First, we discuss how to use classes and object-oriented programming (OOP) to aid in the development of data structures and algorithms. Using OOP techniques will make our algorithms and data structures more general and easier to modify, not to mention easier to understand.

The second part of this Introduction familiarizes the reader with techniques for performing timing tests on data structures and, most importantly, the different algorithms examined in this book. Running timing tests (also called benchmarking) is notoriously difficult to get exactly right, and in the .NET environment, it is even more complex than in other environments. We develop a Timing class that makes it easy to test the efficiency of an algorithm (or a data structure when appropriate) without obscuring the code for the algorithm or data structures.

Developing Classes

This section provides the reader with a quick overview of developing classes in VB.NET. The rationale for using classes and for OOP in general is not discussed here. For a more thorough discussion of OOP in VB.NET, see McMillan (2004).

One of the primary uses of OOP is to develop user-defined data types. To aid our discussion, and to illustrate some of the fundamental principles of OOP,

we will develop two classes for describing one or two features of a geometric data processing system: the Point class and the Circle class.

Data Members and Constructors

The data defined in a class, generally, are meant to stay hidden within the class definition. This is part of the principle of encapsulation. The data stored in a class are called data members, or alternatively, fields. To keep the data in a class hidden, data members are usually declared with the Private access modifier. Data declared like this cannot be accessed by user code.

The Point class will store two pieces of data—the *x* coordinate and the *y* coordinate. Here are the declarations for these data members:

```
Public Class Point

  Private x As Integer
  Private y As Integer

  'More stuff goes here'

End Class
```

When a new class object is declared, a constructor method should be called to perform any initialization that is necessary. Constructors in VB.NET are named New by default, unlike in other languages where constructor methods are named the same as the class.

Constructors can be written with or without arguments. A constructor with no arguments is called the *default* constructor. A constructor with arguments is called a parameterized constructor. Here are examples of each for the Point class:

```
Public Sub New()
  x = 0
  y = 0
End Sub

Public Sub New(ByVal xcor As Integer, ByVal ycor As _
               Integer)
  x = xcor
  y = ycor
End Sub
```

Property Methods

After the data member values are initialized, the next set of operations we need to write involves methods for setting and retrieving values from the data members. In VB.NET, these methods are usually written as *Property methods*.

A Property method provides the ability to both set and retrieve the value of a data member within the same method definition. This is accomplished by utilizing a Get clause and a Set clause. Here are the property methods for getting and setting *x*-coordinate and *y*-coordinate values in the Point class:

```
Public Property Xval() As Integer
  Get
    Return x
  End Get
  Set(ByVal Value As Integer)
    x = Value
  End Set
End Property

Public Property Yval() As Integer
  Get
    Return y
  End Get
  Set(ByVal Value As Integer)
    y = Value
  End Set
End Property
```

When you create a Property method using Visual Studio.NET, the editor provides a template for the method definition like this:

```
Public Property Xval() As Integer
  Get

  End Get
  Set(ByVal Value As Integer)

  End Set
End Property
```

Other Methods

Of course, constructor methods and Property methods aren't the only methods we will need in a class definition. Just what methods you'll need depend on the application. One method included in all well-defined classes is a ToString method, which returns the current state of an object by building a string that consists of the data member's values. Here's the ToString method for the Point class:

```
Public Overrides Function ToString() As String
  Return x & "," & y
End Function
```

Notice that the ToString method includes the modifier Overrides. This modifier is necessary because all classes inherit from the Object class and this class already has a ToString method. For the compiler to keep the methods straight, the Overrides modifier indicates that, when the compiler is working with a Point object, it should use the Point class definition of ToString and not the Object class definition.

One additional method many classes include is one to test whether two objects of the same class are equal. Here is the Point class method to test for equality:

```
Public Function Equal(ByVal p As Point) As Boolean
  If (Me.x = p.x) And (Me.y = p.y) Then
    Return True
  Else
    Return False
  End If
End Function
```

Methods don't have to be written as functions; they can also be subroutines, as we saw with the constructor methods.

Inheritance and Composition

The ability to use an existing class as the basis for one or more new classes is one of the most powerful features of OOP. There are two major ways to

use an existing class in the definition of a new class: 1. The new class can be considered a subclass of the existing class (*inheritance*); and 2. the new class can be considered as at least partially made up of parts of an existing class (*composition*).

For example, we can make a Circle class using a Point class object to determine the center of the circle. Since all the methods of the Point class are already defined, we can reuse the code by declaring the Circle class to be a *derived class* of the Point class, which is called the *base class*. A derived class inherits all the code in the base class plus it can create its own definitions.

The Circle class includes both the definition of a point (*x* and *y* coordinates) as well as other data members and methods that define a circle (such as the radius and the area). Here is the definition of the Circle class:

```
Public Class Circle
  Inherits Point

  Private radius As Single

  Private Sub setRadius(ByVal r As Single)
    If (r > 0) Then
      radius = r
    Else
      radius = 0.0
    End If
  End Sub

  Public Sub New(ByVal r As Single, ByVal x As _
                 Integer, ByVal y As Integer)
    MyBase.New(x, y)
    setRadius(r)
  End Sub

  Public Sub New()
    setRadius(0)
  End Sub

  Public ReadOnly Property getRadius() As Single
    Get
      Return radius
    End Get
  End Property
```

```
    Public Function Area() As Single
      Return Math.PI * radius * radius
    End Function

    Public Overrides Function ToString() As String
      Return "Center = " & Me.Xval & "," & Me.Yval & _
             " - radius = " & radius
    End Function

End Class
```

There are a couple of features in this definition you haven't seen before. First, the parameterized constructor call includes the following line:

```
MyBase.New(x,y)
```

This is a call to the constructor for the base class (the Point class) that matches the parameter list. Every derived class constructor must include a call to one of the base classes' constructors.

The Property method getRadius is declared as a ReadOnly property. This means that it only retrieves a value and cannot be used to set a data member's value. When you use the ReadOnly modifer, Visual Studio.NET only provides you with the Get part of the method.

Timing Tests

Because this book takes a practical approach to the analysis of the data structures and algorithms examined, we eschew the use of Big O analysis, preferring instead to run simple benchmark tests that will tell us how long in seconds (or whatever time unit) it takes for a code segment to run.

Our benchmarks will be timing tests that measure the amount of time it takes an algorithm to run to completion. Benchmarking is as much of an art as a science and you have to be careful how you time a code segment to get an accurate analysis. Let's examine this in more detail.

An Oversimplified Timing Test

First, we need some code to time. For simplicity's sake, we will time a subroutine that writes the contents of an array to the console. Here's the

code:

```
Sub DisplayNums(ByVal arr() As Integer)
  Dim index As Integer
  For index = 0 To arr.GetUpperBound(0)
    Console.Write(arr(index))
  Next
End Sub
```

The array is initialized in another part of the program, which we'll examine later.

To time this subroutine, we need to create a variable that is assigned the system time just as the subroutine is called, and we need a variable to store the time when the subroutine returns. Here's how we wrote this code:

```
Dim startTime As DateTime
Dim endTime As TimeSpan
startTime = DateTime.Now
DisplayNums(nums)
endTime = DateTime.Now.Subtract(startTime)
```

Running this code on a laptop (running at 1.4 MHz on Windows XP Professional) takes about 5 seconds (4.9917 seconds to be exact). Whereas this code segment seems reasonable for performing a timing test, it is completely inadequate for timing code running in the .NET environment. Why?

First, this code measures the elapsed time from when the subroutine was called until the subroutine returns to the main program. The time used by other processes running at the same time as the VB.NET program adds to the time being measured by the test.

Second, the timing code used here doesn't take into account garbage collection performed in the .NET environment. In a runtime environment such as .NET, the system can pause at any time to perform garbage collection. The sample timing code does nothing to acknowledge garbage collection and the resulting time can be affected quite easily by garbage collection. So what do we do about this?

Timing Tests for the .NET Environment

In the .NET environment, we need to take into account the thread in which our program is running and the fact that garbage collection can occur

at any time. We need to design our timing code to take these facts into consideration.

Let's start by looking at how to handle garbage collection. First, let's discuss what garbage collection is used for. In VB.NET, reference types (such as strings, arrays, and class instance objects) are allocated memory on something called the *heap*. The heap is an area of memory reserved for data items (the types previously mentioned). Value types, such as normal variables, are stored on the *stack*. References to reference data are also stored on the stack, but the actual data stored in a reference type are stored on the heap.

Variables that are stored on the stack are freed when the subprogram in which the variables are declared completes its execution. Variables stored on the heap, in contrast, are held on the heap until the garbage collection process is called. Heap data are only removed via garbage collection when there is not an active reference to those data.

Garbage collection can, and will, occur at arbitrary times during the execution of a program. However, we want to be as sure as we can that the garbage collector is not run while the code we are timing is executing. We can head off arbitrary garbage collection by calling the garbage collector explicitly. The .NET environment provides a special object for making garbage collection calls, GC. To tell the system to perform garbage collection, we simply write the following:

```
GC.Collect()
```

That's not all we have to do, though. Every object stored on the heap has a special method called a finalizer. The finalizer method is executed as the last step before deleting the object. The problem with finalizer methods is that they are not run in a systematic way. In fact, you can't even be sure an object's finalizer method will run at all, but we know that before we can be certain an object is deleted, its finalizer method must execute. To ensure this, we add a line of code that tells the program to wait until all the finalizer methods of the objects on the heap have run before continuing. The line of code is as follows:

```
GC.WaitForPendingFinalizers()
```

We have cleared one hurdle but one remains: using the proper thread. In the .NET environment, a program is run inside a process, also called an *application domain*. This allows the operating system to separate each different program running on it at the same time. Within a process, a program or a part of a

program is run inside a *thread*. Execution time for a program is allocated by the operating system via threads. When we are timing the code for a program, we want to make sure that we're timing just the code inside the process allocated for our program and not other tasks being performed by the operating system.

We can do this by using the Process class in the .NET Framework. The Process class has methods for allowing us to pick the current process (the process in which our program is running), the thread in which the program is running, and a timer to store the time the thread starts executing. Each of these methods can be combined into one call, which assigns its return value to a variable to store the starting time (a TimeSpan object). Here's the code:

```
Dim startingTime As TimeSpan
  startingTime = Process.GetCurrentProcess.Threads(0). _
                 UserProcessorTime
```

All we have left to do is capture the time when the code segment we're timing stops. Here's how it's done:

```
duration = Process.GetCurrentProcess.Threads(0). _
           UserProcessorTime.Subtract(startingTime)
```

Now let's combine all this into one program that times the same code we tested earlier:

```
Module Module1

  Sub Main()

    Dim nums(99999) As Integer
    BuildArray(nums)
    Dim startTime As TimeSpan
    Dim duration As TimeSpan
    startTime = Process.GetCurrentProcess.Threads(0). _
                UserProcessorTime
    DisplayNums(nums)
    duration = Process.GetCurrentProcess.Threads(0). _
               UserProcessorTime.Subtract (startTime)
    Console.WriteLine("Time: " & duration.TotalSeconds)

  End Sub
```

```
    Sub BuildArray(ByVal arr() As Integer)

      Dim index As Integer
      For index = 0 To 99999
        arr(index) = index
      Next
    End Sub

End Module
```

Using the new-and-improved timing code, the program returns in just 0.2526 seconds. This compares with the approximately 5 seconds return time using the first timing code. Clearly, a major discrepancy between these two timing techniques exists and you should use the .NET techniques when timing code in the .NET environment.

A Timing Test Class

Although we don't need a class to run our timing code, it makes sense to rewrite the code as a class, primarily because we'll keep our code clear if we can reduce the number of lines in the code we test.

A Timing class needs the following data members:

- startingTime—to store the starting time of the code we are testing,
- duration—the ending time of the code we are testing,

The starting time and the duration members store times and we chose to use the TimeSpan data type for these data members. We'll use just one constructor method, a default constructor that sets both the data members to 0.

We'll need methods for telling a Timing object when to start timing code and when to stop timing. We also need a method for returning the data stored in the duration data member.

As you can see, the Timing class is quite small, needing just a few methods. Here's the definition:

```
Public Class Timing

  Private startingTime As TimeSpan
  Private duration As TimeSpan

  Public Sub New()
```

```
    startingTime = New TimeSpan(0)
    duration = New TimeSpan(0)
  End Sub

  Public Sub stopTime()
    duration = Process.GetCurrentProcess.Threads(0). _
               UserProcessorTime.Subtract(startingTime)
  End Sub

  Public Sub startTime()
    GC.Collect()
    GC.WaitForPendingFinalizers()
    startingTime = Process.GetCurrentProcess. _
                   Threads(0).UserProcessorTime
  End Sub

  Public ReadOnly Property Result() As TimeSpan
    Get
      Return duration
  End Get
  End Property

End Class
```

Here's the program to test the DisplayNums subroutine, rewritten with the Timing class:

```
Option Strict On
Imports Timing
Module Module1
  Sub Main()
    Dim nums(99999) As Integer
    BuildArray(nums)
    Dim tObj As New Timing()
    tObj.startTime()
    DisplayNums(nums)
    tObj.stopTime()
    Console.WriteLine("time (.NET): " & _
                      tObj.Result.TotalSeconds)
    Console.Read()
  End Sub
```

```
    Sub BuildArray(ByVal arr() As Integer)
      Dim index As Integer
      For index = 0 To 99999
        arr(index) = index
      Next
    End Sub

End Module
```

By moving the timing code into a class, we've reduced the number of lines in the main program from 13 to 8. Admittedly, that's not a lot of code to cut out of a program, but more important than the number of lines we cut is the reduction in the amount of clutter in the main program. Without the class, assigning the starting time to a variable looks like this:

```
startTime = Process.GetCurrentProcess.Threads(0). _
            UserProcessorTime
```

With the Timing class, assigning the starting time to the class data member looks like this:

```
tObj.startTime()
```

Encapsulating the long assignment statement into a class method makes our code easier to read and less likely to have bugs.

SUMMARY

This chapter introduces two important techniques we'll use throughout the rest of the book—object-oriented programming and the Timing class—that allow us to perform benchmark tests on the code we produce. Using OOP techniques in our coding will make our programs easier to develop, easier to modify, and, finally, easier to explain and understand.

The timing methods we develop in the Timing class make our benchmarks more realistic because they take into the account the environment with which VB.NET programs run. Simply measuring starting and stopping times using the system clock does not account for the time the operating system uses to run other processes or the time the .NET runtime uses to perform garbage collection.

Exercises

1. Using the Point class, develop a Line class that includes a method for determining the length of a line, along with other appropriate methods.
2. Design and implement a Rational number class that allows the user to perform addition, subtraction, multiplication, and division on two rational numbers.
3. The StringBuilder class (found in the System.Text namespace) is supposedly more efficient for working with strings because it is a mutable object, unlike standard strings, which are immutable, meaning that every time you modify a string variable a new variable is actually created internally. Design and run a benchmark that compares the time it takes to create and display a StringBuilder object of several thousand characters to that for a String object of several thousand characters. If the times are close, modify your test so that the two objects contain more characters. Report your results.

CHAPTER 1

Collections

This book discusses the development and implementation of data structures and algorithms using VB.NET. The data structures we use here are found in the .NET Framework class library System.Collections. In this chapter we develop the concept of a collection by first discussing the implementation of our own collection class (using the array as the basis of our implementation) and then by covering the collection classes in the .NET Framework.

COLLECTIONS DEFINED

A collection is a structured data type that stores data and provides operations for adding data to the collection, removing data from the collection, updating data in the collection, and setting and returning the values of different attributes of the collection.

Collections can be broken down into two types—linear and nonlinear. A linear collection is a list of elements where one element follows the previous element. Elements in a linear collection are normally ordered by position (first, second, third, etc.). In the real world, a grocery list exemplifies a linear collection; in the computer world (which is also real), an array is designed as a linear collection.

Nonlinear collections hold elements that do not have positional order within the collection. An organizational chart is an example of a nonlinear

collection, as is a rack of billiard balls. In the computer world, trees, heaps, graphs, and sets are nonlinear collections.

Collections, be they linear or nonlinear, have a defined set of properties that describe them and operations that can be performed on them. An example of a collection property is the collections Count, which holds the number of items in the collection. Collection operations, called methods, include Add (for adding a new element to a collection), Insert (for adding a new element to a collection at a specified index), Remove (for removing a specified element from a collection), Clear (for removing all the elements from a collection), Contains (for determining whether a specified element is a member of a collection), and IndexOf (for determining the index of a specified element in a collection).

Collections Described

Within the two major categories of collections are several subcategories. Linear collections can be either direct access collections or sequential access collections, whereas nonlinear collections can be either hierarchical or grouped. This section describes each of these collection types.

Direct Access Collections

The most common example of a direct access collection is the array. We define an array as a collection of elements with the same data type that are directly accessed via an integer index, as illustrated in Figure 1.1. Arrays can be static, so that the number of elements specified when the array is declared is fixed for the length of the program, or they can be dynamic, where the number of elements can be increased via the Redim or Redim Preserve statements.

In VB.NET, arrays are not only a built-in data type, but they are also a class. Later in this chapter, when we examine the use of arrays in more detail, we will discuss how arrays are used as class objects.

We can use an array to store a linear collection. Adding new elements to an array is easy since we simply place the new element in the first free position at the rear of the array. Inserting an element into an array is not as easy (or

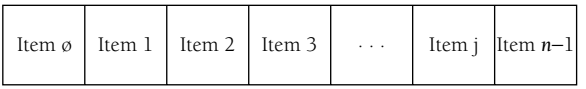


Figure 1.1. Array.

efficient), since we will have to move elements of the array down to make room for the inserted element. Deleting an element from the end of an array is also efficient, since we can simply remove the value from the last element. Deleting an element in any other position is less efficient because, just as with inserting, we will probably have to adjust many array elements up one position to keep the elements in the array contiguous. We will discuss these issues later in the chapter. The .NET Framework provides a specialized array class, ArrayList, for making linear collection programming easier. We will examine this class in Chapter 3.

Another type of direct access collection is the string. A string is a collection of characters that can be accessed based on their index, in the same manner we access the elements of an array. Strings are also implemented as class objects in VB.NET. The class includes a large set of methods for performing standard operations on strings, such as concatenation, returning substrings, inserting characters, removing characters, and so forth. We examine the String class in Chapter 8.

VB.NET strings are immutable, meaning once a string is initialized it cannot be changed. When you modify a string, you create a copy of the string instead of changing the original string. This behavior can lead to performance degradation in some cases, so the .NET Framework provides a StringBuilder class that enables you to work with mutable strings. We'll examine the StringBuilder in Chapter 8 as well.

The final direct access collection type is the structure, known as a user-defined type in Visual Basic 6. A structure is a composite data type that holds data that may consist of many different data types. For example, an employee record consists of the employee's name (a string), salary (an integer), and identification number (a string, or an integer), as well as other attributes. Since storing each of these data values in separate variables could becoming confusing very easily, the language provides the structure for storing data of this type.

A powerful addition to the VB.NET structure is the ability to define methods for performing operations stored on the data in a structure. This makes a structure quite like a class, though you can't inherit from a structure. The following code demonstrates a simple use of a structure in VB.NET:

```
Module Module1
  Public Structure Name
    Dim Fname As String
    Dim Mname As String
    Dim Lname As String
```

```
    Public Function ReturnName() As String
      Return Fname & " " & Mname & " " & Lname
    End Function
    Public Function Initials() As String
      Return Fname.Chars(0) & Mname.Chars(0) & _
        Lname.Chars(0)
    End Function
  End Structure

  Sub Main()
    Dim myname As Name
    Dim fullname As String
    Dim inits As String
    myname.Fname = "Michael"
    myname.Mname = "Mason"
    myname.Lname = "McMillan"
    fullname = myname.ReturnName()
    inits = myname.Initials()
  End Sub

End Module
```

Although many of the elements of VB.NET are implemented as classes (such as arrays and strings), several primary elements of the language are implemented as structures (such as the numeric data types). The Integer data type, for example, is implemented as the Int32 structure. One of the methods you can use with Int32 is the Parse method for converting the string representation of a number into an integer. Here's an example:

```
Dim num As Integer
Dim snum As String
Console.Write("Enter a number: ")
snum = Console.ReadLine()
num = num.Parse(snum)
Console.WriteLine(num + 0)
```

It looks strange to call a method from an Integer variable, but it's perfectly legal since the Parse method is defined in the Int32 structure. The Parse method is an example of a static method, meaning that it is defined in such a way that you don't have to have a variable of the structure type to use the

method. You can call it by using the qualifying structure name before it, like this:

```
num = Int32.Parse(snum)
```

Many programmers prefer to use methods in this way when possible, mainly because the intent of the code becomes much clearer. It also allows you to use the method any time you need to convert a string to an integer, even if you don't have an existing Integer variable.

We will not use many structures in this book for implementation purposes (however, see Chapter 6 on the BitVector structure), but we will use them for creating more complex data to store in the data structures we examine.

Sequential Access Collections

A sequential access collection is a list that stores its elements in sequential order. We call this type of collection a linear list. Linear lists are not limited by size when they are created, meaning they are able to expand and contract dynamically. Items in a linear list are not accessed directly; they are referenced by their position, as shown in Figure 1.2. The first element of a linear list lies at the front of the list and the last element lies at the rear of the list.

Because of the lack of direct access to the elements of a linear list, to access an element you have to traverse through the list until you arrive at the position of the element you are looking for. Linear list implementations usually allow two methods for traversing a list: 1. in one direction from front to rear and 2. from both front to rear and rear to front.

A simple example of a linear list is a grocery list. The list is created by writing down one item after another until the list is complete. The items are removed from the list while shopping as each item is found.

Linear lists can be either ordered or unordered. An ordered list has values in order with respect to each other, as in the following:

Beata Bernica David Frank Jennifer Mike Raymond Terrill

An unordered list consists of elements in any order. The order of a list makes a big difference when performing searches on the data in the list, as you'll see

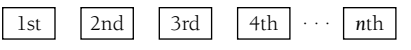


FIGURE 1.2. Linear List.

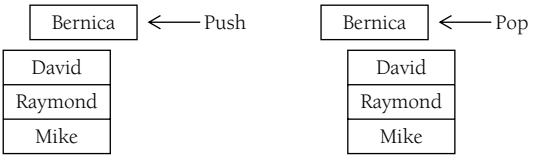


FIGURE 1.3. Stack Operations.

in Chapter 2 when we explore the binary search algorithm versus a simple linear search.

Some types of linear lists restrict access to their data elements. Examples of these types of lists are stacks and queues. A stack is a list where access is restricted to the beginning (or top) of the list. Items are placed on the list at the top and can only be removed from the top. For this reason, stacks are known as Last-In, First-Out structures. When we add an item to a stack, we call the operation a push. When we remove an item from a stack, we call that operation a pop. These two stack operations are shown in Figure 1.3.

The stack is a very common data structure, especially in computer systems programming. Among its many applications, stacks are used for arithmetic expression evaluation and for balancing symbols.

A queue is a list where items are added at the rear of the list and removed from the front of the list. This type of list is known as a First-In, First-Out structure. Adding an item to a queue is called an EnQueue, and removing an item from a queue is called a Dequeue. Queue operations are shown in Figure 1.4.

Queues are used in both systems programming, for scheduling operating system tasks, and in simulation studies. Queues make excellent structures for simulating waiting lines in every conceivable retail situation. A special type of queue, called a priority queue, allows the item in a queue with the highest priority to be removed from the queue first. Priority queues can be used to study the operations of a hospital emergency room, where patients with heart trouble need to be attended to before a patient with a broken arm, for example.

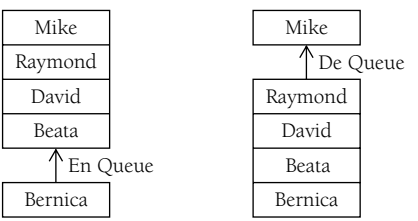


FIGURE 1.4. Queue Operations.

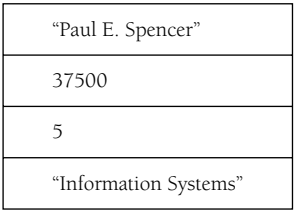


FIGURE 1.5. A record to be hashed.

The last category of linear collections we'll examine is called the generalized indexed collection. The first of these, called a hash table, stores a set of data values associated with a key. In a hash table, a special function, called a hash function, takes one data value and transforms the value (called the key) into an integer index that is used to retrieve the data. The index then is used to access the data record associated with the key. For example, an employee record may consist of a person's name, his or her salary, the number of years the employee has been with the company, and the department in which he or she works. This structure is shown in Figure 1.5. The key to this data record is the employee's name. VB.NET has a class, called Hashtable, for storing data in a hash table. We explore this structure in Chapter 10.

Another generalized indexed collection is the dictionary. A dictionary is made up of a series of key–value pairs, called associations. This structure is analogous to a word dictionary, where a word is the key and the word's definition is the value associated with the key. The key is an index into the value associated with the key. Dictionaries are often called associative arrays because of this indexing scheme, though the index does not have to be an integer. We will examine several Dictionary classes that are part of the .NET Framework in Chapter 11.

Hierarchical Collections

Nonlinear collections are broken down into two major groups: hierarchical collections and group collections. A hierarchical collection is a group of items divided into levels. An item at one level can have successor items located at the next lower level.

One common hierarchical collection is the tree. A tree collection looks like an upside-down tree, with one data element as the root and the other data values hanging below the root as leaves. The elements of a tree are called nodes, and the elements that are below a particular node are called the node's children. A sample tree is shown in Figure 1.6.

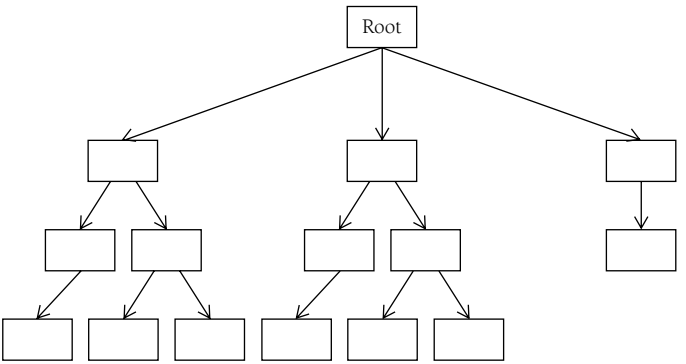


FIGURE 1.6. A tree collection.

Trees have applications in several different areas. The file systems of most modern operating systems are designed as a tree collection, with one directory as the root and other subdirectories as children of the root.

A binary tree is a special type of tree collection where each node has no more than two children. A binary tree can become a binary search tree, making searches for large amounts of data much more efficient. This is accomplished by placing nodes in such a way that the path from the root to a node where the data are stored takes the shortest route possible.

Yet another tree type, the heap, is organized so that the smallest data value is always placed in the root node. The root node is removed during a deletion, and insertions into and deletions from a heap always cause the heap to reorganize so that the smallest value is placed in the root. Heaps are often used for sorts, called a heap sort. Data elements stored in a heap can be kept sorted by repeatedly deleting the root node and reorganizing the heap.

All the varieties of trees are discussed in Chapter 12.

Group Collections

A nonlinear collection of items that are unordered is called a group. The three major categories of group collections are sets, graphs, and networks.

A set is a collection of unordered data values where each value is unique. The list of students in a class is an example of a set, as is, of course, the integers. Operations that can be performed on sets include union and intersection. An example of set operations is shown in Figure 1.7.

A graph is a set of nodes and a set of edges connecting the nodes. Graphs are used to model situations where each of the nodes in a graph must be visited,

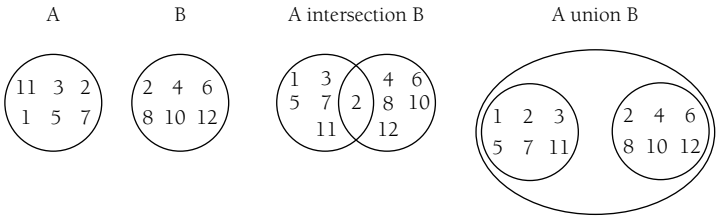


FIGURE 1.7. Set Collection Operations.

sometimes in a particular order, and the goal is to find the most efficient way to "traverse" the graph. Graphs are used in logistics and job scheduling and are well studied by computer scientists and mathematicians. You may have heard of the "Traveling Salesman" problem. This is a particular type of graph problem that involves determining which cities on a salesman's route should be traveled to most efficiently complete the route within the budget allowed for travel. A sample graph of this problem is shown in Figure 1.8.

This problem is part of a family of problems known as NP-complete problems. For large problems of this type, an exact solution is not known. For example, the solution to the problem in Figure 1.8 involves 10 factorial tours, which equals 3,628,800 tours. If we expand the problem to 100 cities, we have to examine 100 factorial tours, which we currently cannot do with current methods. An approximate solution must be found instead.

A network is a special type of graph in which each of the edges is assigned a weight. The weight is associated with a cost for using that edge to move from one node to another. Figure 1.9 depicts a network of cities where the weights are the miles between the cities (nodes).

We've now finished our tour of the different types of collections we are going to discuss in this book. Now we're ready to actually look at how collections are implemented in VB.NET. We're going to start by implementing a collection class using only native data types (i.e., arrays), and then we'll examine the general collection classes that are part of the .NET Framework.

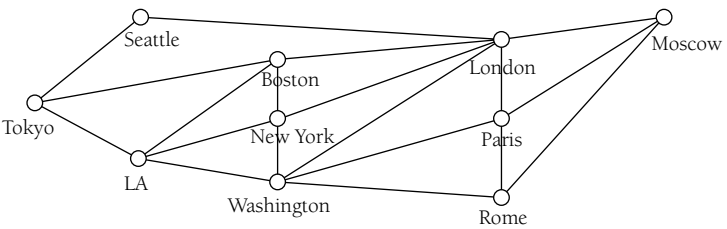


FIGURE 1.8. The Traveling Salesman Problem.

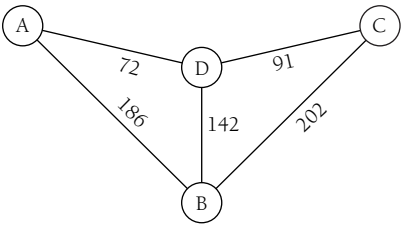


FIGURE 1.9. A Network Collection.

THE VB.NET COLLECTION CLASS

The VB.NET Framework library includes a generic collection class for storing data. The class includes two methods and two properties for adding, removing, retrieving, and determining the number of items in the collection. All data entered into a collection class object get stored as an object. For some applications this is adequate; however, for many applications, data must be stored as its original type. In a later section we'll show you how to build a strongly typed collection class.

Adding Data to a Collection

The Add method is used to store data in a collection. In its simplest form, the method takes just one argument, a data item to store in the collection. Here's a sample:

```
Dim names As New Collection
names.Add("David Durr")
names.Add("Raymond Williams")
names.Add("Bernica Tackett")
names.Add("Beata Lovelace")
```

Each name is added in order to the collection, though we don't normally talk about this type of collection being in order. This is especially true when items are added to a collection in this manner.

Another way to add data to a collection is to also store keys along with the data. The data can then be retrieved either randomly or by the key. If you use a key, it must be a unique string expression. The code looks like this:

```
Dim names As New Collection() 'Ordered by room number
names.Add("David Durr", "300")
```

```
names.Add("Raymond Williams", "301")
names.Add("Bernica Tackett", "302")
names.Add("Beata Lovelace", "303")
```

You can also add items to a collection and specify their order in the collection. An item can be added before or after any other item in the collection by specifying the position of the new item relative to another item. For example, you can insert an item before the third item in the collection or after the second item in the collection.

To insert an item before an existing item, list the position of the existing item as the third argument to the Add method. To insert an item after an existing item, list the position of the existing item as the fourth argument to the method. Here are some examples:

```
Dim names As New Collection()
names.Add("Jennifer Ingram", "300")
names.Add("Frank Opitz", "301")
names.Add("Donnie Gundolf", "302", 1) 'added before
                                        first item
names.Add("Mike Dahly", "303",, 2) 'added after
                                     second item
```

Collection items are retrieved with the Item method. Items can be retrieved either by their index or by a key, if one was specified when the item was added. Using the index and the Count property, we can return each item from a collection using a For loop as follows:

```
Dim index As Integer
For index = 1 To names.Count
  Console.WriteLine(names.Item(index))
Next
```

If you want to retrieve items from a collection by their keys, you must specify a string as the argument to the Item method. The following code fragment iterates through the collection just created using the key of each item to retrieve the name:

```
Dim x As Integer
Dim index As Integer = 300
Dim key As String
```

```
For x = 1 To names.Count
  key = CStr(index)
  Console.WriteLine(names.Item(key))
  index += 1
Next
```

For the sake of completion, we'll end this section discussing how to enumerate a collection. Collections are built primarily as a data structure you use when you don't really care about the position of the elements in the structure. For example, when you build a collection that contains all of the textbox controls on a Windows form, you are primarily interested in being able to perform some task on all the textbox objects in the collection. You're not really interested in which textbox is in which position in the collection.

The standard way to enumerate a collection is using the For Each statement. The Collection class has a built-in enumerator that the For Each statement uses to grab each member of the collection. Here's an example:

```
Dim name As String
For Each name In names
  Console.WriteLine(name)
Next
```

The enumerator has methods for moving from one item to the next and for checking for the end of the collection. If you are building your own collection class, as we do in the next section, you'll need to write your own enumerator. We show you how to do this in the next section.

A Collection Class Implementation Using Arrays

In this section we'll demonstrate how to use VB.NET to implement our own Collection class. This will serve several purposes. First, if you're not quite up to speed on OOP, this implementation will show you some simple OOP techniques in VB.NET. We can also use this section to discuss some performance issues that are going to arise as we discuss the different VB.NET data structures. Finally, we think you'll enjoy this section, as well as the other implementation sections in this book, because it's really quite fun to reimplement the existing data structures using just the native elements of the language. To paraphrase Donald Knuth (one of the pioneers of computer science), you haven't really learned something well until you've taught it to a computer.

So, by teaching VB.NET how to implement the different data structures, we'll learn much more about those structures than if we just choose to use them in our day-to-day programming.

Defining a Custom Collection Class

Before we look at the properties and methods we need for our Collection class, we need to discuss the underlying data structure we're going to use to store our collection: the array. The elements added to our collection will be stored sequentially in the array. We'll need to keep track of the first empty position in the array, which is where a new item is placed when it is added to the collection, using a Protected variable we call pIndex. Each time an item is added, we must increment pIndex by one. Each time an item is removed from the collection, we must decrement pIndex by one.

To make our implementation as general as possible, we'll assign the data type Object to our array. The VB.NET data structures generally use this technique also. However, by overriding the proper methods in these data structure classes, we can create a data structure that allows only a specific data type. You'll see an example of this in Chapter 3, where we create a data structure called an ArrayList that stores only strings.

One implementation decision we need to make is to choose how large to make our array when we instantiate a new collection. Many of the data structures in the .NET Framework are initialized to 16 and we'll use that number for our implementation. This is not specified in the CollectionBase class, however. We're just using the number 16 because it is consistent with other data structures. The code for our collection class using Protected variables is as follows:

```
Public Class CCollection

  Protected pCapacity As Integer = 16
  Protected pArr(16) As Object
  Protected pIndex As Integer
  Protected pCount As Integer
```

We can decide what properties and methods our class should have by looking at what properties and methods are part of the CollectionBase class, the .NET Framework class used to implement collections in VB.NET. Later in the chapter we'll use the CollectionBase class as a base class for another collection class.

CCollection Class Properties

The only property of the class is Count. This property keeps track of the number of elements in a collection. In our implementation, we use a Protected variable pCount, which we increment by one when a new item is added, and we decrement by one when an item is removed, as follows:

```
ReadOnly Property Count()
  Get
    Return pCount
  End Get
End Property
```

CCollection Class Methods

The first method we need to consider is the constructor method. Collection classes normally just have a default constructor method without an initialization list. All we do when the constructor method is called is set the two variables that track items in the collection, pCount and pIndex, to zero:

```
Public Sub New()
  pIndex = 0
  pCount = 0
End Sub
```

The Add method involves our first little "trick." Unlike an array, where we must explicitly create more space when the array is full, we want a collection to expand automatically when it fills up its storage space. We can solve this problem by first checking to see whether the array storing our collection items is full. If it is, we simply redimension the array to store 16 more items. We call a Private function IsFull, to check to see if every array element has data in it. We also have to increment pIndex and pCount by one to reflect the addition of a new item into the collection. The code looks like this:

```
Public Sub Add(ByVal item As Object)
  If (Me.IsFull()) Then
    pCapacity += 16
    ReDim Preserve pArr(pCapacity)
  End If
  pArr(pIndex) = item
```

```
    pIndex += 1
    pCount += 1
  End Sub

  Private Function IsFull() As Boolean
    If (pArr(pCapacity) <> Nothing) Then
      Return True
    Else
      Return False
    End If
  End Function
```

The Clear method erases the contents of the collection, setting the capacity of the collection back to the initial capacity. Our implementation simply redimensions the pArr array to the initial capacity, then we set pIndex and pCount back to zero. Here's the code:

```
  Public Sub Clear()
    ReDim pArr(16)
    pCount = 0
    pIndex = 0
  End Sub
```

The Contains method simply iterates through the underlying array, setting a Boolean flag to True if the item passed to the method is found in the collection, and leaving the flag as False otherwise:

```
  Public Function Contains(ByVal item As Object) _
      As Boolean
    Dim x As Integer
    Dim flag As Boolean = False
    For x = 0 To pArr.GetUpperBound(0)
      If (pArr(x) = item) Then
        flag = True
      End If
    Next
    Return flag
  End Function
```

The CollectionBase class implements a method called CopyTo that allows us to copy the contents of a collection into an array if we should need to

manipulate the elements of the collection in an array instead. This method is created by dimensioning the passed array to the same size as the collection and just copying elements from the Collection class's array to the new array. The following code shows how to do this:

```
Public Sub CopyTo(ByRef arr() As Object)
  Dim x As Integer
  ReDim arr(pCount - 1)
  For x = 0 To pCount - 1
    arr(x) = pArr(x)
  Next
End Sub
```

The IndexOf method returns the index of the position of an item in a collection. If the item requested isn't in the collection, the method returns –1. Here's the code:

```
Public Function IndexOf(ByVal item As Object) As Integer
  Dim x, pos As Integer
  pos = -1
  For x = 0 To pArr.GetUpperBound(0)
    If (pArr(x) = item) Then
      pos = x
    End If
  Next
  Return pos
End Function
```

The IndexOf method uses a simple searching technique, the linear search, to look for the requested item. This type of search, also called a sequential search (for obvious reasons), usually starts at the beginning of the data structure and traverses the items in the structure until the item is found or the end of the list is reached. Each item in the structure is accessed in sequence. When the data set being searched is relatively small, the linear search is the simplest to code and is usually fast enough. However, with large data sets, the linear search proves to be too inefficient and different search techniques are necessary. A more efficient search technique—the binary search—will be discussed in Chapter 2.

The Remove method removes the first occurrence of the specified item in the collection. This method is also implemented with a linear search to find

the item to remove. Once the item is found, the remaining elements in the array are shifted up one space to close the gap left by the removed item. This shifting is handled by a Private method, ShiftUp. Here's the code:

```
Private Sub ShiftUp(ByVal p As Integer)
  Dim x As Integer
  For x = p To pArr.GetUpperBound(0) - 1
    pArr(x) = pArr(x + 1)
  Next
  pIndex = SetIndex()
End Sub
Public Sub Remove(ByVal item As Object)
  Dim x, position As Integer
  For x = 0 To pArr.GetUpperBound(0)
    If (pArr(x) = item) Then
      pArr(x) = ""
      position = x
      Exit For
    End If
  Next
  ShiftUp(position)
  pCount -= 1
End Sub
```

These comprise the primary methods needed by a custom Collection class. The last feature we need to implement is a way to enumerate the collection so that we can iterate over the collection using the For Each statement.

Implementing an Enumerator

To use the For Each statement with our custom Collection class we need to implement the IEnumerable interface. IEnumerable exposes an enumerator, which is necessary for iteration over a collection. The interface has just one method, GetEnumerator, which returns an enumerator. The enumerator we return must implement the methods of the IEnumerator interface, which provides the necessary methods and properties for iterating through a collection.

The best strategy for implementing an enumerator is to create an enumeration class. Since the enumeration class needs considerable information about the class it is enumerating, we will implement the enumeration class as a

Private class inside CCollection. We could also write the enumeration code as part of the CCollection class implementation, but this will make CCollection much more confusing, and doing so also violates the important principles of modularity and encapsulation.

The first thing we have to do is go back and modify the heading for the CCollection class. For our enumeration code to work properly, our class must implement the IEnumerable interface. The heading now looks like this:

```
Public Class CCollection
  Implements IEnumerable
```

We'll start our implementation by showing the code for returning the enumerator. This code implements the GetEnumerator method and is used when you try to actually enumerate a collection object. The method returns a new CEnumerator object, which is the name of our Private enumerator class. The following code is placed in the CCollection class:

```
Public Function GetEnumerator() As IEnumerator _
    Implements System.Collections.IEnumerable. _
    GetEnumerator
  Return New CEnumerator(pArr)
End Function
```

Now let's look at the code for the CEnumerator class.

An enumerator class must implement the Current property and the MoveNext() and Reset() methods. We utilize two Private data members, one that is assigned the collection object and another, an index, that keeps track of the number of elements in the collection. Of course, the class also needs a constructor, which is the first bit of code we'll look at.

The constructor for an enumerator class is passed the object being enumerated as its only argument. The constructor assigns the enumerated object to its own internal variable and sets the index variable to –1, indicating that the enumerator has not returned an element from the collection. Here's the code:

```
Public Sub New(ByVal pArr As Array)
  ccol = pArr
  index = -1
End Sub
```

MoveNext() increments the index and checks to see whether the end of the array has been reached:

```
Public Function MoveNext() As Boolean _
    Implements System.Collections.IEnumerator.MoveNext

  index += 1
  If index >= ccol.Length Then
    Return False
  Else
    Return True
  End If
End Function
```

The method returns False if the array is out of elements and True otherwise.

The Current property is implemented as a read-only property, since its only purpose is to return the current element of the array:

```
Public ReadOnly Property Current() As Object _
    Implements System.Collections.IEnumerator.Current

  Get
    Return ccol(index)
  End Get
End Property
```

The Reset() method is implemented because the IEnumerator class requires it. All it does is set the index back to –1 when appropriate.

Now that we have all the code in place, we can use the CEnumerator class to enumerate our custom collection of elements. We went to all the trouble to develop the class so that we can use a For Each statement to loop through the elements in the collection. When we use the For Each, the enumerator for the collection is called and is used to enumerate the underlying array. One problem we'll see with this implementation is that every element of the array gets displayed, even those elements that don't have a value. This will make your collection list look somewhat messy; consequently, you might want to decrease the initial capacity of the list.

Let's put all this together and look at an example that demonstrates how to use the CEnumerator class by creating a list of names and displaying them

using a For Each statement:

```
Module Module1

  Public Class CCollection
    Implements IEnumerable
    Protected pCapacity As Integer = 8
    Protected pArr(8) As Object
    Protected pIndex As Integer
    Protected pCount As Integer

    Public Sub New()
      pIndex = 0
      pCount = 0
    End Sub

    ReadOnly Property Count()
      Get
        Return pCount
      End Get
    End Property

    Public Sub Add(ByVal item As Object)
      If (Me.IsFull()) Then
        pCapacity += 8
        ReDim Preserve pArr(pCapacity)
      End If
      pArr(pIndex) = item
      pIndex += 1
      pCount += 1
    End Sub

    Private Function IsFull() As Boolean
      If (pArr(pCapacity) <> Nothing) Then
        Return True
      Else
        Return False
      End If
    End Function

    Public Function Contains(ByVal item As Object) As _
        Boolean
      Dim x As Integer
```

```
  Dim flag As Boolean = False
  For x = 0 To pArr.GetUpperBound(0)
    If (pArr(x) = item) Then
      flag = True
    End If
  Next
  Return flag
End Function

Public Sub CopyTo(ByRef arr() As Object)
  Dim x As Integer
  ReDim arr(pCount - 1)
  For x = 0 To pCount - 1
    arr(x) = pArr(x)
  Next
End Sub

Public Sub Display()
  Dim x As Integer
  For x = 0 To pCount - 1
    Console.WriteLine(pArr(x))
  Next
End Sub

Private Sub ShiftUp(ByVal p As Integer)
  Dim x As Integer
  For x = p To pArr.GetUpperBound(0) - 1
    pArr(x) = pArr(x + 1)
  Next
  pIndex = SetIndex()
End Sub

Private Sub RangeShiftUp(ByVal pos As Integer, _
                         ByVal n As Integer)
  Dim y As Integer
  Dim toMove As Object
  y = pos + n
  toMove = pArr(y)
  While (CStr(toMove) <> "" And (y < pCount))
    pArr(pos) = toMove
    pArr(y) = ""
```

```
      pos += 1
      y += 1
      toMove = pArr(y)
    End While
    pIndex = SetIndex()
    For y = pIndex To pCapacity
      pArr(y) = ""
    Next
  End Sub

Public Sub Remove(ByVal item As Object)
  Dim x, position As Integer
  For x = 0 To pArr.GetUpperBound(0)
    If (pArr(x) = item) Then
      pArr(x) = ""
      position = x
      Exit For
    End If
  Next
  ShiftUp(position)
  pCount -= 1
End Sub

Public Sub RemoveAt(ByVal p As Integer)
  pArr(p) = ""
  pCount -= 1
  ShiftUp(p)
End Sub

Public Sub Insert(ByVal item As Object, ByVal pos _
                  As Integer)
  ShiftDown(pos)
  pArr(pos) = item
End Sub

Private Sub ShiftDown(ByVal n As Integer)
  Dim x As Integer
  If (Me.IsFull()) Then
    pCapacity += 16
    ReDim Preserve pArr(pCapacity)
  End If
```

```
  For x = pIndex - 1 To n Step -1
    pArr(x + 1) = pArr(x)
  Next
  pArr(n) = ""
  pIndex += 1
  pCount += 1
End Sub

Private Function SetIndex() As Integer
  Dim x As Integer
  For x = 0 To pArr.GetUpperBound(0)
    If CStr(pArr(x)) = "" Then
      Return x
    End If
  Next
End Function

Public Sub Clear()
  ReDim pArr(7)
  pCount = 0
  pIndex = 0
End Sub

Public Function IndexOf(ByVal item As Object) _
    As Integer
  Dim x, pos As Integer
  pos = -1
  For x = 0 To pArr.GetUpperBound(0)
    If (pArr(x) = item) Then
      pos = x
    End If
  Next
  Return pos
End Function

Public Function GetEnumerator() As IEnumerator _
    Implements System.Collections.IEnumerable. _
    GetEnumerator

  Return New CEnumerator(pArr)
End Function
```

```
  Private Class CEnumerator : Implements IEnumerator
    Private index As Integer
    Private ccol As Array

    Public Sub New(ByVal pArr As Array)
      ccol = pArr
      index = -1
    End Sub

    'Implementation of IEnumerator
    Public ReadOnly Property Current() As Object _
       Implements System.Collections.IEnumerator.Current
      Get
        Return ccol(index)
      End Get
    End Property

    Public Function MoveNext() As Boolean Implements _
        System.Collections.IEnumerator.MoveNext

      index += 1
      If index >= ccol.Length Then
        Return False
      Else
        Return True
      End If
    End Function

    Public Sub Reset() _
        Implements System.Collections.IEnumerator.Reset

      index = -1
    End Sub

  End Class

End Class

Sub Main()
  Dim NameList As New CCollection()
  Dim name As Object
  NameList.Add("David")
  NameList.Add("Mike")
```

```
    NameList.Add("Raymond")
    NameList.Add("Bernica")
    NameList.Add("Jennifer")
    NameList.Add("Frank")
    NameList.Add("Beata")
    NameList.Add("Terrill")
    For Each name In NameList
      Console.WriteLine(name)
    Next
    Console.Write("Press Enter to quit")
    Console.Read()
    End Sub
End Module
```

Building a Strongly Typed Collection Using CollectionBase

One of the problems with using the Collection class to build a collection is that the underlying data structure (ArrayList) used to store data stores the data as objects (the Object type). This means that any type of data can be added to the collection, and if special methods are supposed to be called using the objects stored in the collection, the CType function will have to be used to convert the object to the proper type.

The solution to this problem entails building a *strongly typed* collection using the CollectionBase abstract class. This class, called a MustInherit class in VB.NET, consists of a set of abstract methods (similar to the methods found in the Collection class) that can be implemented in a derived class along with a set of Public methods that can be used as is.

Shortcomings of the Collection Class

Using the Collection class to store class objects presents problems when you want to call class or instance methods using the stored objects. Data objects added to a collection are stored with the Object type, not the actual data type of the object. Because of this, methods defined for a class object cannot be called directly; you must use the CType function to convert the stored object to its original type.

The following short program demonstrates the problem with the Collection class:

```
Option Strict On
Module Module1
  Public Class Student
    Private name As String
    Private id As String
    Private major As String
    Shared count As Integer = 0

    Public Sub New(ByVal n As String, ByVal i As _
                   String, ByVal m As String)
      name = n
      id = i
      major = m
      count += 1
    End Sub

    Public Overrides Function ToString() As String
      Return name & " " & id & " " & major
    End Function

    Public Property CourseGrade() As Double
      Get
        Return grade
      End Get
      Set(ByVal Value As Double)
        grade = Value
      End Set
    End Property

  End Class

  Sub main()
      Dim s1 As New Student("Mike McMillan", "123", _
                            "CS")
      Dim s2 As New Student("Raymond Williams", "234", _
                            "CLASSICS")
      Dim s3 As New Student("David Durr", "345", "ENGL")
      s1.CourseGrade = 89
      s2.CourseGrade = 92
```

```
        s3.CourseGrade = 76
        Dim students As New Collection
        students.Add(s1)
        students.Add(s2)
        students.Add(s3)
        Dim std As Object
        For Each std In students
          'The following line won't work
          std.CourseGrade += std.CourseGrade + 5
          'The following line only works because
          'ToString is inherited from Object
          Console.WriteLine(std.ToString)
        Next
        Console.Read()
    End Sub

End Module
```

The Student class includes a ToString method that displays all the data stored in a class instance. When a Student object is stored in a collection, however, the ToString method is not available because the Student object is actually stored as Object type.

Inside the For Each loop, the line that tries to access the CourseGrade property of the Student class won't work with Option Strict On. The compiler sees the variable std as an Object type and cannot resolve the call to the Property method since CourseGrade is not defined for Object type.

The second line inside the For Each loop does work because ToString is inherited from Object and is therefore automatically resolved. If we had used a less generic method for displaying the value of a Student object, the method would not be resolvable in this context.

To solve this problem we need a Collection class that is strongly typed so that the methods we want to use are available. In the next sections we explore how to use the CollectionBase class to derive our own, strongly typed collections.

CollectionBase Members

Let's look first at lists of the abstract and Public methods of the CollectionBase class before we delve into using the class to build our own collection classes.

The abstract methods are the following:

- **Add:** Adds an object to the end of the collection.
- **Insert:** Inserts an element into the collection at the specified index.
- **Remove:** Removes the first occurrence of a specific object from the collection.
- **Contains:** Determines whether the collection contains a specific element.
- **IndexOf:** Searches for the specified element and returns the zero-based index of the first occurrence within the collection.
- **CopyTo:** Copies the entire collection to a compatible one-dimensional array, starting at the specified index of the target array.

The Public methods of the CollectionBase class are

- **Clear:** Removes all objects from the collection.
- **RemoveAt:** Removes the element at the specified index of the collection.
- **Equals:** References whether two instances are equal.
- **ToString:** Returns a string that represents the current object.
- **GetEnumerator:** Returns an enumerator used to iterate through the collection.
- **GetHashCode:** Serves as a hash function for a particular type.

Deriving a Custom Collection Class from CollectionBase

We can build a strongly typed collection by deriving a subclass from the CollectionBase class and implementing the Add method so that only the proper data type can be added to the collection. Of course, we'll want to implement other abstract methods inherited from CollectionBase as well, but it's the Add method that ensures we have a strongly typed collection.

Besides the public and abstract methods mentioned earlier, the CollectionBase class also contains an ArrayList used to store the objects placed into a collection. This object, which is declared as Protected, is named List. You can use List as is in your derived class. List does not have to be declared in your derived class in order to use it.

To demonstrate how a strongly typed collection is built, we'll derive a class that stores Student objects (from the previous example). The first method is the constructor:

```
Public Class StudentColl
  Inherits CollectionBase

  Public Sub New()
    MyBase.New()
  End Sub
```

Now we can implement the Add method. All we have to do is specify the parameter to the method as the data type of the object we want to add to the collection. Here's the code:

```
Public Sub Add(ByVal stu As Student)
  list.Add(stu)
End Sub
```

That's all there is to it. When the user of the class adds an object to the collection, if the object type isn't Student, then VS.NET will signal an error.

There are several other abstract methods we can implement to build a complete collection class. We'll look at two of them here—Remove and Item—and save the others as exercises.

The Item method can be implemented as either a subprocedure or as a Property method. We'll choose to build the method as a Property here because we can use one of the features of the Property method to keep our collection strongly typed.

Property methods are typically used to allow "assignment" access to private data in a class. "Assignment" access means that rather than a message-passing model, data are passed to a class using assignment statements. When a Property method is declared, VS.NET provides getter and setter submethods in a template model. Here's an example, if you're not familiar with how Property methods work:

```
Public Property Item() As Student
  Get

  End Get
  Set(ByVal Value As Student)

  End Set
End Property
```

For this class, we don't want to provide full access using a Property method because we already have an Add method, so we'll just provide part of the method to allow the value of a collection item to be returned. To do this, we have to declare the method as a ReadOnly method:

```
Public ReadOnly Property Item(ByVal index As Integer) _
    As Student
  Get
    Return list.Item(index)
  End Get
End Property
```

This method can now be used to retrieve an item in a collection at a specified position:

```
Console.Writeline(students.Item(2))
```

but the value at that position cannot be changed.

The other method implementation we show here is the Remove method. There is nothing special we have to do with this method, other than ensure that a legal index is specified as the method parameter. Let's look at the code first:

```
Public Sub Remove(ByVal index As Integer)
  If (index < 0) Or (index > Count - 1) Then
    MsgBox("Invalid index. Can't remove.")
  Else
    list.RemoveAt(index)
  End If
End Sub
```

If the index supplied as the argument to the method is either less than zero or greater than the number of elements in the ArrayList, it is an invalid index. The Remove method tests for this condition and displays a message to the user if the index is indeed invalid. If the index position is valid, the item is removed from the collection.

To conclude this section, we provide a listing of all the code to the StudentColl class:

```
Public Class StudentColl
  Inherits CollectionBase
```

```
    Public Sub New()
      MyBase.New()
    End Sub

    Public Sub Add(ByVal stu As Student)
      list.Add(stu)
    End Sub

    Public Sub Remove(ByVal index As Integer)
      If (index < 0) Or (index > Count - 1) Then
        MsgBox("Invalid index. Can't remove.")
      Else
        list.RemoveAt(index)
      End If
    End Sub

    Public ReadOnly Property Item(ByVal index As _
                                  Integer) As Student
      Get
        Return list.Item(index)
      End Get
    End Property

End Class
```

SUMMARY

The .NET Framework class library revolutionizes Visual Basic programming. In previous versions of Visual Basic, if a programmer wanted to implement an advanced data structure such as a stack or a hash table, he or she had to code the structure from scratch. VB.NET introduces a set of collection classes that make designing and implementing data structures much easier and, most importantly, much more likely to happen. Now, instead of relying on a simple array or user-defined type for structuring data, a programmer can use one of the built-in classes either for the complete data structure or as a base for a custom-designed structure.

This chapter introduced the concept of the collection, describing the two major collection classifications (linear and nonlinear). We spent some time discussing the Collection class that is built into the .NET Framework library, and then we looked in depth at creating a custom collection class. Examining

how to implement a built-in class may seem like a waste of time, but since there are going to be many times in a programmer's career when one must modify an existing class, it is a worthwhile experience to experiment with building these classes from more primitive elements. We ended the chapter exploring how to build strongly typed collections using the CollectionBase abstract class.

Exercises

1. Create a class called Test that has data members for a student's name and a number indicating the test number. This class is used in the following scenario: When a student turns in a test, he or she places it face down on the teacher's desk. If a student wants to check an answer, the teacher has to turn the stack over so that the first test is face up, work through the stack until the student's test is found, and then remove the test from the stack. When the student finishes checking the test, it is reinserted at the end of the stack.

   Write a Windows application to model this situation. Include textboxes for the user to enter a name and a test number. Put a list box on the form for displaying the final list of tests. Provide four buttons for the following actions: 1. Turn in a test; 2. Let student look at test; 3. Return a test; and 4. Exit. Perform the following actions to test your application: 1. Enter a name and a test number. Insert the test into a collection named submittedTests. 2. Enter a name, delete the associated test from submittedTests, and insert the test in a collection named outForChecking. 3. Enter a name, delete the test from outForChecking, and insert it in submittedTests. 4. Press the Exit button. The Exit button doesn't stop the application but instead deletes all tests from outForChecking and inserts them in submittedTests and displays a list of all the submitted tests.
   Use the Collection class from the .NET Framework library.
2. Complete the StudentColl class by implementing the following methods:
   Insert
   Contains
   IndexOf
3. Rewrite Exercise 1 using the custom Collection class implemented in the chapter.

CHAPTER 2

Arrays and ArrayLists

The array is the most common data structure and is found in nearly all computer programming languages. Arrays are implemented somewhat differently in VB.NET than in previous versions of Visual Basic because in VB.NET an array is actually an instantiated Array class object. The Array class provides a set of methods for performing tasks such as sorting and searching that programmers had to build by hand in the past.

A new feature in VB.NET is the ArrayList class. An ArrayList is an array that grows dynamically as more space is needed. For situations where you cannot accurately determine the ultimate size of an array, or where the size of the array will change quite a bit over the lifetime of a program, an ArrayList may be a better choice than an array.

In this chapter we'll quickly touch on the basics of using arrays in VB.NET, then move on to more advanced topics, including copying, cloning, testing for equality, and using the static methods of the Array and ArrayList classes.

Array Basics

An array stores a set of elements ordered by position. The base position, or index, of arrays in VB.NET is zero, which is a big change from previous versions of Visual Basic, since in those versions the array base was actually

user-defined. It is quite common to see Visual Basic 6 (or earlier) array declarations like

```
Dim Sales(1990 To 1999) As Double
```

where 1990 is the lower bound and 1999 is the upper bound.

Strictly speaking, this type of usage is illegal in VB.NET. An array is declared with just the upper bound of the array as the argument and the implied base for the array is always zero. However, one can work around this situation by creating an array using the Array object of the .NET Framework. The following program demonstrates how to build an array with a base index of 1:

```
Module Module1

  Sub Main()
    Dim test As Array
    Dim length(0) As Integer
    Dim lower(0) As Integer
    length(0) = 5
    lower(0) = 1

    test = Array.CreateInstance(GetType(System.Int32),_
                                length, lower)

    Try
      test.SetValue(100, 0)
    Catch e As Exception
      Console.WriteLine(e.Message)
    End Try

    test.SetValue(100, 1)

    Console.WriteLine(test.GetValue(1))
    Console.Write("Press enter to quit")
    Console.Read()
  End Sub

End Module
```

The array is built using an Array object (test). This object is used to create all arrays in VB.NET. The CreateInstance method of the Array class sets

the data type of the array elements, the number of elements in the array, and the lower bound. Next, an attempt is made to place a value at index 0, which should not exist since we set the lower bound to 1. An exception is thrown, letting us know the index lies out of bounds. Then a value is placed in index 1, which is an acceptable location. The output from this program is as follows:

```
Index was outside the bounds of the array.
100
```

Declaring and Initializing Arrays

The previous example is clearly not the way most VB.NET programmers will declare and initialize arrays. Most experienced programmers will use statements they are familiar with, such as

```
Dim names(4) As String
Dim grades(19) As Double
```

One change from Visual Basic 6 is that VB.NET allows the programmer to provide an initialization list when declaring an array. The list comprises a set of elements that are stored in the array based on their position in the list. The following example declares and initializes a list of numbers:

```
Dim numbers() As Integer = {0,1,2,3,4,5}
```

The list is demarked by parentheses and each element is delimited by a comma. An upper bound is not given in the array declaration; the compiler assigns the upper bound based on the number of elements in the initialization list.

You can declare an array without providing the upper bound or providing an initialization list. However, you will eventually have to provide an initialization list or you won't be able to assign values to the array. Hence you code should look like this:

```
Dim numbers() As Integer
numbers = New Integer() {0, 1, 2, 3, 4, 5}
```

The following code fragment leads to a NullReferenceException exception:

```
Dim numbers() As Integer
numbers(0) = 23
```

An alternative method for declaring and initializing an array is to use a more Java-like style,

```
Dim numbers() As Integer = New Integer() {0, 1, 2, 3, 4}
```

but it's likely that only Java programmers moving over to VB.NET will adopt this style.

Setting and Accessing Array Elements

Elements are stored in an array either by direct access or by calling the Array class method SetValue. Direct access involves referencing an array position by index on the left-hand side of an assignment statement:

```
names(2) = "Raymond"
sales(19) = 23123
```

The SetValue method provides a more object-oriented way to set the value of an array element. The method takes two arguments, an index number and the value of the element:

```
names.SetValue(2, "Raymond")
sales.SetValue(19, 23123)
```

Array elements are accessed either by direct access or by calling the GetValue method. The GetValue method takes a single argument, called an index:

```
myName = names(2)
monthSales = sales.GetValue(19)
```

Commonly, one loops through an array to access every array element using a For loop. A frequent mistake novice programmers make when coding the loop is to either hard-code the upper value of the loop (which is a mistake because

the upper bound may change if the array is dynamic) or call a function that accesses the upper bound of the loop for each iteration of the oop:

```
For index = 0 to UBound(sales)
  totalSales = totalSales + sales(index)
Next index
```

Since the number of elements in the sales array isn't going to change during the execution of the loop, it is more efficient to store the upper bound of the array in a variable and use that variable as the upper range of the loop:

```
upper = sales.GetUpperBound(0)
totalSales = 0
For index = 0 to upper
  totalSales += sales(index)
Next
```

Notice the use of the Array class method GetUpperBound to find the upper bound of the array. This method takes a single argument: the dimension of the upper bound you are looking for. There is also a GetLowerBound method, but this method doesn't have much use in VB.NET, unless you're adventurous enough to use the technique we discussed earlier for changing the lower bound of an array.

Methods and Properties for Retrieving Array Metadata

The Array class provides several properties for retrieving metadata about an array:

- **Length:** Returns the total number of elements in all dimensions of an array.
- **GetLength:** Returns the number of elements in specified dimension of an array.
- **Rank:** Returns the number of dimensions of an array.
- **GetType:** Returns the Type of the current array instance.

The Length method proves to be useful for counting the number of elements in a multidimensional array, as well as for returning the exact number of

elements in the array. Otherwise, you can use the GetUpperBound method and add one to the value.

Because Length returns the total number of elements in an array, the GetLength method counts the elements in one dimension of an array. This method, along with the Rank property, can be used to resize an array at runtime without running the risk of losing data. This technique is discussed later in the chapter.

The GetType method is used for determining the data type of an array in a situation where you may not be sure of the array's type, such as when the array gets passed as an argument to a method. In the following code fragment, we create a variable of type Type, which allows us to use a class method, IsArray, to determine whether an object is an array:

```
Dim numbers() As Integer
numbers = New Integer() {0, 1, 2, 3, 4}
Dim arrayType As Type = numbers.GetType()
If (arrayType.IsArray) Then
  Console.WriteLine("The array type is: {0}", arrayType)
Else
  Console.WriteLine("Not an array")
End If
```

If the object is an array, then the code returns the data type of the array.

Not only does the GetType method return the type of the array, but it also lets us know that the object is indeed an array. Here is the output from the code:

```
The array type is: System.Int32[]
```

The brackets indicate the object is an array. Also notice that we use a format when displaying the data type. We have to do this because we can't convert the Type data to string date to concatenate it with the rest of the displayed string.

We can also write this code fragment in a more traditional Visual Basic style, using the isArray function:

```
Dim numbers() As Integer
numbers = New Integer() {0, 1, 2, 3, 4}
If (IsArray(numbers)) Then
  Console.WriteLine("numbers is type: {0}", _
                    numbers.GetType)
```

```
Else
  Console.WriteLine("Not an array")
End If
```

Multidimensional Arrays

So far we have limited our discussion to arrays that have just a single dimension. In VB.NET, an array can have up to 32 dimensions, though arrays with more than three dimensions are very rare (and can be very confusing).

Multidimensional arrays are declared by providing the upper bound of each of the dimensions of the array. The two-dimensional declaration

```
Dim Grades(3,4) As Integer
```

declares an array that consists of 4 rows and 5 columns. Two-dimensional arrays are often used to model matrices.

You can also declare a multidimensional array without specifying the dimension bounds. To do this, you use commas to specify the number of dimensions. For example,

```
Dim Sales(,) As Double
```

declares a two-dimensional array, whereas

```
Dim Sales(,,) As Double
```

declares a three-dimensional array. When you declare arrays without providing the upper bounds of the dimensions, you have to later redimension the array with those bounds:

```
Redim Sales(3,4)
```

Multidimensional arrays can be initialized with an initialization list. Look at the following statement:

```
Dim Grades(,) As Integer = {{1, 82, 74, 89, 100}, _
                            {2, 93, 96, 85, 86}, _
                            {3, 83, 72, 95, 89}, _
                            {4, 91, 98, 79, 88}}
```

First, notice that the upper bounds of the array are not specified. When you initialize an array with an initialization list, you cannot specify the bounds of the array. The compiler computes the upper bounds of each dimension from the data in the initialization list. The initialization list itself is demarked with curly braces, as is each row of the array. Each element in the row is delimited with a comma.

Accessing the elements of a multidimensional array is similar to accessing the elements of a one-dimensional array. You can use the traditional array access technique,

```
grade = Grades(0,2)
Grades(2,2) = 99
```

or you can use the methods of the Array class:

```
grade = Grades.GetValue(0,2)
```

You can't use the SetValue method with a multidimensional array because the method only accepts two arguments—a value and a single index.

It is a common operation to perform calculations on all the elements of a multidimensional array, though often based on either the values stored in the rows of the array or the values stored in the columns of the array. Using the Grades array, if each row of the array comprises a student record, we can calculate the grade average for each student as follows:

```
Dim Grades(,) As Integer = {{1, 82, 74, 89, 100}, _
                            {2, 93, 96, 85, 86}, _
                            {3, 83, 72, 95, 89}, _
                            {4, 91, 98, 79, 88}}
Dim row, col, last_student, total As Integer
Dim last_grade As Integer
Dim average As Double
last_student = Grades.GetUpperBound(0)
last_grade = Grades.GetUpperBound(1)
For row = 0 To last_student
  total = 0
  For col = 1 To last_grade
    total = total + Grades(row, col)
  Next
```

```
    average = total / last_grade
    Console.WriteLine("Average: " & average)
  Next
```

Parameter Arrays

Most method definitions require that a set number of parameters be provided to the method, but there are times when you want to write a method definition that allows an optional number of parameters. You can do this using a construct called a parameter array.

A parameter array is specified in the parameter list of a method definition by using the keyword ParamArray. The following method definition allows any amount of numbers to be supplied as parameters, with the total of the numbers returned from the method:

```
Public Function sumnums(ByVal ParamArray nums() As _
                        Integer) As Integer
  Dim x, sum As Integer
  For x = 0 To nums.GetUpperBound(0)
    sum += nums(x)
  Next
  Return sum
End Function
```

This method will work with either of the following calls:

```
total = sumnums(1, 2, 3)
total = sumnums(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

When you define a method using a parameter array, the parameter array arguments have to be supplied last in the parameter list for the compiler to be able to process the list of parameters correctly. Otherwise, the compiler would know neither the ending point of the parameter array elements nor the beginning of other parameters of the method.

Jagged Arrays

When you create a multidimensional array, you always create a structure that has the same number of elements in each of the rows. For example, look at

the following array declaration:

```
Dim sales(11, 29) As Integer 'Sales for each day of _
                                each month
```

This array assumes each row (month) has the same number of elements (days); however, we know that some months have 30 days, some have 31, and one month has 28 or 29. With the array we've just declared, there will be several empty elements in the array. This isn't much of a problem for this particular array, but with a much larger array we end up with a lot of wasted space.

To solve this problem we use a jagged array instead of a two-dimensional array. A jagged array consists of an array of arrays, with each row of an array made up of an array. Each dimension of a jagged array is a one-dimensional array. We call it a "jagged" array because the number of elements in each row may be different. A picture of a jagged array would not be square or rectangular but would have uneven or jagged edges.

A jagged array is declared by putting two sets of parentheses after the array variable name. The first set of parentheses indicates the number of rows in the array. The second set of parentheses is left blank. This marks the place for the one-dimensional array that is stored in each row. Normally, the number of rows is set in an initialization list in the declaration statement, like this:

```
Dim jaggedArray()() As Integer = {New Integer(9) {}}
```

This statement looks strange, but it makes sense when you break it down. The array jaggedArray is an Integer array of 10 elements, where each of the elements is also an Integer array. The initialization list is actually just the initialization for the rows of the array, indicating that each row element is an array of 10 elements, with each element initialized to the default value.

Once the jagged array is declared, the elements of the individual row arrays can be assigned values. The following code fragment assigns values to jaggedArray:

```
jaggedArray(0)(0) = 23
jaggedArray(0)(1) = 13
...
jaggedArray(7)(5) = 45
```

This notation doesn't look like it belongs in Visual Basic; it looks more like a programmer tried to combine elements of VB.NET and C++. The first

set of parentheses indicates the row number and the second set indicates the element of the row array. The first statement accesses the first element of the first array, the second statement accesses the second element of the first array, and the third statement accesses the sixth element of the eighth array.

Let's look at an example of how to use a jagged array. The following program creates an array named sales (tracking one week of sales for two months), assigns sales figures to its elements, and then loops through the array to calculate the average sales for one week of each of the two months stored in the array:

```
Module module1
  Sub main()
    Dim jan(30) As Integer
    Dim feb(29) As Integer
    Dim sales()() As Integer = {jan, feb}
    Dim mnth, dy, total As Integer
    Dim average As Double
    sales(0)(0) = 41
    sales(0)(1) = 30
    sales(0)(2) = 23
    sales(0)(3) = 34
    sales(0)(4) = 28
    sales(0)(5) = 35
    sales(0)(6) = 45
    sales(1)(0) = 35
    sales(1)(1) = 37
    sales(1)(2) = 32
    sales(1)(3) = 26
    sales(1)(4) = 45
    sales(1)(5) = 38
    sales(1)(6) = 42
    For mnth = 0 To 1
      total = 0
      average = 0.0
      For dy = 0 To 6
        total += sales(mnth)(dy)
        average = total / 7
        Console.WriteLine("Average sales for month " & _
                          mnth & ": " & average)
```

```
        Next
      Next
    End Sub
End Module
```

To demonstrate the flexibility you have in declaring jagged arrays, the sales array is built from two arrays declared earlier:

```
Dim jan(30) As Integer
Dim feb(29) As Integer
Dim sales()() As Integer = {jan, feb}
```

The loop that calculates the averages looks like a standard nested For loop. The only difference is how we access each array element to add it to the total. We have to use the proper syntax for accessing jagged array elements:

```
total += sales(mnth)(dy)
```

You will be tempted to write

```
total += sales(mnth, dy)
```

but this syntax works only for a two-dimensional array, not a jagged array. If you learn to think of a jagged array as an array of arrays, you will be more likely to get the syntax right each time.

Dynamically Resizing Arrays

An array is declared (or initialized) with a size, but this size is not fixed and can vary throughout the lifetime of the program using the array. An array can be dynamically resized using the ReDim and Preserve commands.

The ReDim command used by itself automatically resets all array elements to their default values and resizes the array to the new upper bound. Here's an example:

```
Dim grades() As Integer = {87, 76, 99, 65, 89}
ReDim grades(9)
```

The grades array is resized to 10 elements, with each element set to 0. If you want to retain the values in the array, you have to combine the ReDim command with the Preserve command:

```
ReDim Preserve grades(9)
```

Now the array is resized to 10 elements and the original 5 grades are still in the array.

It's common to resize an array using either a formula or a standard increment of some type. If you're not particularly worried about your array growing too large, you can simply double the number of elements in the array when you need to resize it using the Length method of the Array class:

```
Redim Preserve grades(grades.Length * 2)
```

Multidimensional arrays are resized in a similar manner, but with some important differences. First, let's create a simple two-dimensional array:

```
Dim grades(,) As Integer = {{78, 84, 89, 93}, _
                            {94, 82, 65, 88}, _
                            {100, 78, 82, 85}}
```

This grades array is a two-dimensional array with three rows and four columns. If we want to resize this array by adding both rows and columns, we cannot use the Preserve command; we have to reset all the array elements to zero.

If you want to resize a multidimensional array and retain the values in the array, you can resize only the last dimension. In the case of a two-dimensional array, such as the grades array considered here, this means you can add columns to the array but cannot add any rows. The following line of code resizes the grades array to hold 10 columns rather than 4:

```
ReDim Preserve grades(3, 9)
```

The general rule when resizing multidimensional arrays is that you can only resize the last dimension—all the other dimensions must stay the same size.

The ArrayList Class

Using ReDim and ReDim Preserve consumes precious computer resources, plus you will have to add code to your program to test for when you need to resize. One solution to this problem is to use a type of array that automatically resizes itself when the array runs out of storage space. This array is called an ArrayList and it is part of the System.Collections namespace in the .NET Framework library.

An ArrayList object has a Capacity property that stores its size. The initial value of the property is 16. When the number of elements in an ArrayList reaches this limit, the Capacity property adds another 16 elements to the storage space of the ArrayList. Using an ArrayList in a situation where the number of elements in an array can grow larger, or smaller, can be more efficient than using a ReDim Preserve command with a standard array.

As we discussed in Chapter 1, an ArrayList stores objects using the Object type. If you need a strongly typed array, you should use a standard array or some other data structure.

Members of the ArrayList Class

The ArrayList class includes several methods and properties for working with ArrayLists. Here is a list of some of the most commonly used methods and properties:

- **Add**(): Adds an element to the ArrayList.
- **AddRange**(): Adds the elements of a collection to the end of the ArrayList.
- **Capacity**: Stores the number of elements the ArrayList can hold.
- **Clear**(): Removes all elements from the ArrayList.
- **Contains**(): Determines whether a specified item is in the ArrayList.
- **CopyTo**(): Copies the ArrayList or a segment of it to an array.
- **Count**: Returns the number of elements currently in the ArrayList.
- **GetEnumerator**(): Returns an enumerator to iterate over the ArrayList.
- **GetRange**(): Returns a subset of the ArrayList as an ArrayList.
- **IndexOf**(): Returns the index of the first occurrence of the specified item.

- **Insert**(): Inserts an element into the ArrayList at a specified index.
- **InsertRange**(): Inserts the elements of a collection into the ArrayList starting at the specified index.
- **Item**(): Gets or sets an element at the specified index.
- **Remove**(): Removes the first occurrence of the specified item.
- **RemoveAt**(): Removes an element at the specified index.
- **Reverse**(): Reverses the order of the elements in the ArrayList.
- **Sort**(): Alphabetically sorts the elements in the ArrayList.
- **ToArray**(): Copies the elements of the ArrayList to an array.
- **TrimToSize**(): Sets the capacity of the ArrayList to the number of elements in the ArrayList.

Using the ArrayList Class

ArrayLists are not used like standard arrays. Normally, items are just added to an ArrayList using the Add method, unless there is a reason why an item should be added at a particular position, in which case the Insert method should be used. In this section, we examine how to use these and the other members of the ArrayList class.

The first thing we have to do with an ArrayList is declare it, as we do here:

```
Dim grades As New ArrayList()
```

Notice that a constructor is used in this declaration. If an ArrayList is not declared using a constructor, the object will not be available in later program statements.

Objects are added to an ArrayList using the Add method. This method takes a single argument—an Object to add to the ArrayList. The Add method also returns an integer indicating the position in the ArrayList where the element was added, though this value is rarely used in a program. Here are some examples:

```
grades.Add(100)
grades.Add(84)
```

```
Dim position As Integer
position = grades.Add(77)
Console.WriteLine("The grade 77 was added at " & _
                  "position: " & position)
```

The objects in an ArrayList can be displayed using a For Each loop. The ArrayList has a built-in enumerator that manages iterating through all the objects in the ArrayList, one at a time. The following code fragment demonstrates how to use a For Each loop with an ArrayList:

```
Dim grade As Object
Dim total As Integer = 0
Dim average As Double = 0.0
For Each grade In grades
  total += CInt(grade)
Next
average = total / grades.Count
Console.WriteLine("The average grade is: " & average)
```

If you want to add an element to an ArrayList at a particular position, you can use the Insert method. This method takes two arguments: the index to insert the element and the element to be inserted. The following code fragment inserts two grades in specific positions to preserve the order of the objects in the ArrayList:

```
grades.Insert(1, 99)
grades.Insert(3, 80)
```

You can check the current capacity of an ArrayList by calling the Capacity property and you can determine how many elements are in an ArrayList by calling the Count property:

```
Console.WriteLine("The current capacity of grades is:" _
                  & grades.Capacity)
Console.WriteLine("The number of grades in grades is:" _
                  & grades.Count)
```

There are several ways to remove items from an ArrayList. If you know the item you want to remove, but don't know its position, you can use

the Remove method. This method takes just one argument—an object to remove from the ArrayList. If the object exists in the ArrayList, it is removed. If the object isn't in the ArrayList, nothing happens. When a method like Remove is used, it is typically called inside an If–Then statement using a method that can verify whether the object is actually in the ArrayList, such as the Contains method. Here's a sample code fragment:

```
If (grades.Contains(54)) Then
  grades.Remove(54)
Else
  MsgBox("Object not in ArrayList.")
End If
```

If you know the index of the object you want to remove, you can use the RemoveAt method. This method takes one argument—the index of the object you want to remove. The only exception you can cause is passing an invalid index to the method. The method works like this:

```
grades.RemoveAt(2)
```

You can determine the position of an object in an ArrayList by calling the IndexOf method. This method takes one argument, an object, and returns the object's position in the ArrayList. If the object is not in the ArrayList, the method returns −1. Here's a short code fragment that uses the IndexOf method in conjunction with the RemoveAt method:

```
Dim pos As Integer
pos = grades.IndexOf(70)
grades.RemoveAt(pos)
```

In addition to adding individual objects to an ArrayList, you can also add ranges of objects. The objects must be stored in a data type that is derived from ICollection. This means that the objects can be stored in an array, a Collection, or even in another ArrayList.

There are two different methods you can use to add a range to an ArrayList: AddRange and InsertRange. The AddRange method adds the range of objects to the end of the ArrayList, and the InsertRange method adds the range at a specified position in the ArrayList.

The following program demonstrates how these two methods are used:

```
Module Module1

  Sub Main()
    Dim names As New ArrayList
    names.Add("Mike")
    names.Add("Bernica")
    names.Add("Beata")
    names.Add("Raymond")
    names.Add("Jennifer")
    Dim name As Object
    Console.WriteLine("The original list of names")
    For Each name In names
      Console.WriteLine(name)
    Next
    Console.WriteLine()
    Dim newNames() As String = {"David", "Michael"}
    Dim moreNames As New Collection
    moreNames.Add("Terrill")
    moreNames.Add("Donnie")
    moreNames.Add("Mayo")
    moreNames.Add("Clayton")
    moreNames.Add("Alisa")
    names.InsertRange(0, newNames)
    names.AddRange(moreNames)
    Console.WriteLine("The new list of names")
    For Each name In names
      Console.WriteLine(name)
    Next
    Console.Read()
  End Sub

End Module
```

The output from this program is as follows:

```
David
Michael
Mike
Bernica
```

```
Beata
Raymond
Jennifer
Terrill
Donnie
Mayo
Clayton
Alisa
```

The first two names are added at the beginning of the ArrayList because the specified index is 0. The last names are added at the end because the AddRange method is used.

Two other methods that many programmers find useful are the ToArray method and the GetRange method. The GetRange method returns a range of objects from the ArrayList as another ArrayList. The ToArray method copies all the elements of the ArrayList to an array. Let's look first at the GetRange method.

The GetRange method takes two arguments: the starting index and the number of elements to retrieve from the ArrayList. GetRange is not destructive, in that the objects are just copied from the original ArrayList into the new ArrayList. Here's an example of how the method works, using our same program:

```
Dim someNames As New ArrayList
someNames = names.GetRange(2, 4)
Console.WriteLine("someNames sub-ArrayList")
For Each name In someNames
  Console.WriteLine(name)
Next
```

The output from this program fragment is as follows:

```
Mike
Bernica
Beata
Raymond
```

The ToArray method allows you to easily transfer the contents of an ArrayList to a standard array. The primary reason for using the ToArray method would be to take advantage of the faster access speed of an array.

The ToArray method takes no arguments and returns the elements of the ArrayList to an array. Here's an example of how to use the method:

```
Dim arrNames() As Object
arrNames = names.ToArray()
Console.WriteLine("Names from an array: ")
Dim index As Integer
For index = 0 To arrNames.GetUpperBound(0)
  Console.WriteLine(arrNames(index))
Next
```

The last part of the code fragment proves that the elements from the ArrayList have actually been stored in the array arrNames.

Comparing Arrays to ArrayLists

The first question most programmers ask when they are exposed to the ArrayList is "How does the ArrayList perform compared to a standard array?" This is a legitimate question since arrays are considered built-in data structures and the ArrayList must be imported from the .NET Framework library. There are also some assumed inefficiencies with the ArrayList, primarily the facts that ArrayLists store all data as Objects and they can grow dynamically.

In this section, we examine the performance of these two data structures using the Timing class developed in the Introduction to this book. We'll compare doing calculations using the two data structures and compare insertion and deletion operations. These tests will help us determine whether an efficiency issue will affect our choice of using one of these data structures over the other.

Calculating Large Numbers

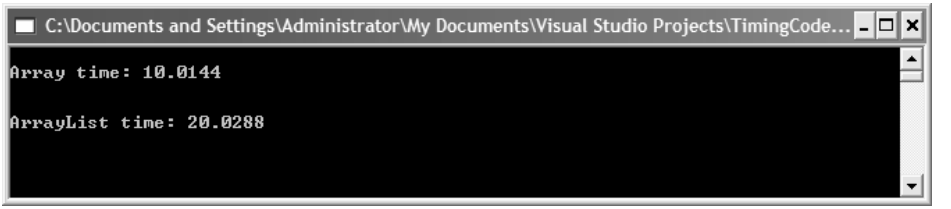
One of the first tasks to look at when comparing arrays and ArrayLists is the time it takes to perform numerical calculations. The test we undertake involves initializing an array and an ArrayList to 100,000 elements. Once the data structures are built, each one is used to compute the sum of its elements.

Here's the program for this test:

```
Option Strict On
Imports Timing
Module Module1

  Sub Main()
    Dim nums(99999) As Integer
    Dim nums1 As New ArrayList
    Dim tObj As New Timing
    Dim tObjArrList As New Timing
    tObj.startTime()
    BuildArray(nums)
    CalcNumber(nums)
    tObj.stopTime()
    Console.WriteLine("Array time: " & _
                      tObj.Result.TotalMilliseconds)
    tObjArrList.startTime()
    BuildArrList(nums1)
    CalcNumber1(nums1)
    tObjArrList.stopTime()
    Console.WriteLine("ArrayList time: " & _
        tObjArrList.Result.TotalMilliseconds)
    Console.Read()
  End Sub

  Sub BuildArray(ByVal arr() As Integer)
    Dim index As Integer
    For index = 0 To 99999
      arr(index) = index
    Next
  End Sub

  Sub CalcNumber(ByVal arr() As Integer)
    Dim bigNum As Long = 0
    Dim index As Integer
    For index = 0 To arr.GetUpperBound(0)
      bigNum += arr(index)
    Next
  End Sub
```

```
    Sub BuildArrList(ByVal nums1 As ArrayList)
      Dim index As Integer
      For index = 0 To 99999
        nums1.Add(index)
      Next
    End Sub

    Sub CalcNumber1(ByVal nums As ArrayList)
      Dim total As Long = 0
      Dim num As Object
      For Each num In nums
        total += CLng(num)
      Next
    End Sub

End Module
```

The output from this program looks like this:



This time is fairly consistent every time we run the program. Occasionally, the time difference would even increase from a 2:1 to a 3:1 ratio.

Element Insertion

Although the ArrayList lost the calculation race, we expect that it will win the element insertion race, especially when we compare inserting a new element into a full array versus inserting a new element into a full ArrayList.

The problem with inserting into a full array is that we have to perform two operations to allow the insertion: First we must allocate more space to the array and then shift a set of array elements over one position to make room for the new element. For a small array, this is really not an issue, but for an array with many elements (such as our test array in the following program, with

one million elements) the time it takes to move up to one million elements to a new position can be prohibitive.

To test how prohibitive inserting into an array can be, we can set up an array with one million elements and insert an item into the 500,000th position. If the array is full, we'll first have to allocate more space using Redim Preserve and then shift every element from 499,999 to 999,999 over one position to make room for the new element. If the array isn't full, we only have to perform the shifting, but in the code that follows we test the worst-case scenario.

Let's start the test by examining the program:

```
Option Strict On
Imports Timing

Module Module1

  Sub Main()
    Dim nums(999999) As Integer
    Dim numsAList As New ArrayList
    Dim i As Integer
    Dim arrTime As New Timing
    Dim arrListTime As New Timing
    For i = 0 To 999999
      nums(i) = i
    Next
    For i = 0 To 999999
      numsAList.Add(i)
    Next
    arrTime.startTime()
    InsertElement(nums, 499999, -1)
    arrTime.stopTime()
    arrListTime.startTime()
    numsAList.Insert(499999, -1)
    arrListTime.stopTime()
    Console.WriteLine()
    Console.WriteLine("Array time: " & _
                      arrTime.Result.TotalMilliseconds)
    Console.WriteLine()
    Console.WriteLine("ArrayList time: " & _
        arrListTime.Result.TotalMilliseconds)
```

```
    Console.Read()
  End Sub

  Sub InsertElement(ByVal arr() As Integer, ByVal pos _
                    As Integer, ByVal item As Integer)
    Dim index As Integer
    If (arr(pos) <> 0) And arr(arr.GetUpperBound(0)) _
        <> 0 Then
      ReDim Preserve arr(CInt(arr.Length * 1.25))
      ShiftDown(arr, pos)
      arr(pos) = item
    ElseIf (arr(pos) <> 0) Then
      ShiftDown(arr, pos)
      arr(pos) = item
    Else
      arr(pos) = item
    End If
  End Sub

  Sub ShiftDown(ByVal arr() As Integer, ByVal pos _
                As Integer)
    Dim index As Integer
    For index = pos To arr.Length - 2
      arr(pos + 1) = arr(pos)
    Next
  End Sub

End Module
```

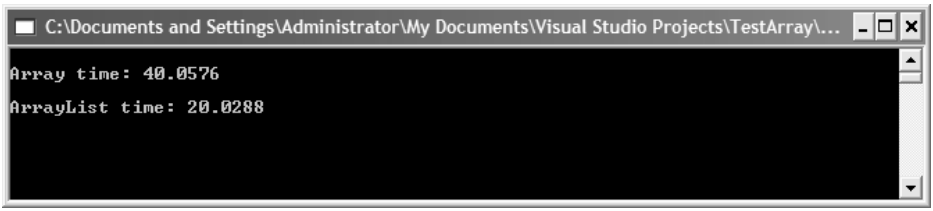
The two subroutines—InsertElement and ShiftDown—perform the hard work. InsertElement first checks to see whether there is an element in the insertion position and whether the array is full and more space is required. If so, then Redim Preserve is called and the size of the array is increased by 25%. The ShiftDown routine is then called to make room for the new element and, finally, the new element is inserted into the array.

If the array isn't full but there is an element in the insertion position, then only the ShiftDown subroutine is called. Otherwise, the element is simply inserted into the requested position.

The ShiftDown subroutine loops through the array, moving array elements from one position to the next position in the index. The only tricky part of

the code involves the For loop. This loop has to stop two positions before the end of the array or an exception will be thrown because the index will be out of bounds.

Here are the results from running this test:



The 20-second difference between the standard array and the ArrrayList is consistent over several runs of the program. These results clearly show the advantage of using an ArrayList over an array when numerous elements and insertions into the middle of the data must be performed.

SUMMARY

The array is the most commonly used data structure in computer programming. Most, if not all, computer languages provide some type of built-in array. For many applications, the array is the easiest data structure to implement and the most efficient. Arrays are useful in situations where you need direct access to "far-away" elements of your data set.

In many programming languages, arrays are not always the best structures to use if the size of the data is likely to change. In VB.NET, however, arrays can be dynamically resized, so this issue does not arise (though some overhead is involved).

The .NET Framework introduces a new type of array called an ArrayList. ArrayLists have many of the features of the array, but they are somewhat more powerful because they can resize themselves when the current capacity of the structure is full. The ArrayList also has several useful methods for performing insertions, deletions, and searches.

The chapter ended with a comparison of arrays and ArrayLists. The comparison revealed that an array is faster for calculating whereas ArrayLists are faster for inserting elements. One conclusion that can be reached from these tests is that arrays are probably the best structure to use when working strictly with numeric values, whereas ArrayLists are more useful when dealing with

nonnumeric data. However, if the data set you are working with is tabular in nature then you will need to use a two-dimensional array.

Exercises

1. Design and implement a class that allows a teacher to track the grades in a single course. Include methods that calculate the average grade, the highest grade, and the lowest grade. Write a program to test your class implementation.
2. Modify Exercise 1 so that the class can keep track of multiple courses. Write a program to test your implementation.
3. Rewrite Exercise 1 using an ArrayList. Write a program to test your implementation and compare its performance to that of the array implementation in Exercise 1 using the Timing class.
4. Design and implement a class that uses an array to mimic the behavior of the ArrayList class. Include as many methods from the ArrayList class as possible. Write a program to test your implementation.

Chapter 3

Basic Sorting Algorithms

The two most common operations performed on data stored in a computer are sorting and searching. This has been true since the beginning of the computing industry, which means that sorting and searching are also two of the most studied operations in computer science. Many of the data structures discussed in this book are designed primarily to make sorting and/or searching easier and more efficient on the data stored in the structure.

This chapter introduces you to the fundamental algorithms for sorting and searching data. These algorithms depend only on the array as a data structure and the only "advanced" programming technique used is recursion. This chapter also introduces you to the techniques we'll use throughout the book to informally analyze different algorithms for speed and efficiency.

Sorting Algorithms

Most of the data we work with in our day-to-day lives is sorted. We look up definitions in a dictionary by searching alphabetically. We look up a phone number by moving through the last names in the book alphabetically. The post office sorts mail in several ways—by zip code, then by street address, and then by name. Sorting is a fundamental process in working with data and deserves close study.

Although researchers have developed some very sophisticated sorting algorithms, there are also several simple sorting algorithms you should study

first. These sorting algorithms are the insertion sort, the bubble sort, and the selection sort. Each of these algorithms is easy to understand and easy to implement. They are not the best overall algorithms for sorting by any means but they are ideal for small data sets and in other special circumstances due to their ease of implementation.

An Array Class Test Bed

To examine these algorithms, we will first need a test bed in which to implement and test them. We'll build a class that encapsulates the normal operations performed with an array: inserting elements, accessing elements, and displaying the contents of the array. Here's the code:

```
Public Class CArray

  Private arr() As Integer
  Private numElements As Integer

Public Sub New(ByVal number As Integer)
  ReDim Preserve arr(number)
  numElements = 0
End Sub

Public Sub Insert(ByVal item As Integer)
  If (numElements > arr.GetUpperBound(0)) Then
    ReDim Preserve arr(numElements * 1.25)
  End If
  arr(numElements) = item
  numElements += 1
End Sub

Public Sub showArray()
  Dim index As Integer
  For index = 0 To numElements - 1
    Console.Write(arr(index) & " ")
  Next
  Console.WriteLine()
End Sub

End Class
```

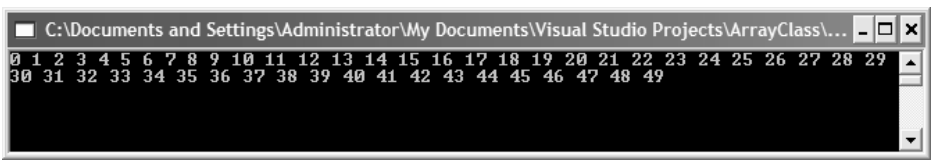
The only tricky part of this class definition resides within the Insert definition. It's entirely possible that user code could attempt to insert an item into the array when the upper bound of the array has been reached. There are two possible ways to handle this situation. One is to alert the user that the end of the array has been reached and not perform the insertion. The other solution is to make the array act like an ArrayList and provide more capacity in the array by using the Redim Preserve statement. That's the choice used here.

You should also note that the showArray() method only accesses those array elements that have data in them. The easiest way to write this method is to loop through the upper bound of the array. This would be a bad decision because there might be array elements where no data are stored, which leaves zeroes or empty strings to be displayed.

Let's see how this class works by writing a simple program to load the array with 50 values (though the original upper bound is only through 9) and display the values:

```
Sub Main()
  Dim theArray As New CArray(9)
  Dim index As Integer
  For index = 0 To 49
    theArray.Insert(index)
  Next
  theArray.showArray()
  Console.Read()
End Sub
```

The output looks like this:



Before leaving the CArray class to begin the examination of sorting and searching algorithms, let's discuss how we're going to actually store data in a CArray class object. To demonstrate most effectively how the different sorting algorithms work, the data in the array need to be in a random order. This is best achieved by using a random number generator to assign each array element to the array.

The easiest way to generate random numbers is to use the Rnd() function. This function returns a random number less than or equal to zero. To generate a random number within a particular range, say from 1 to 100, use the following formula:

```
100 * Rnd() + 1
```

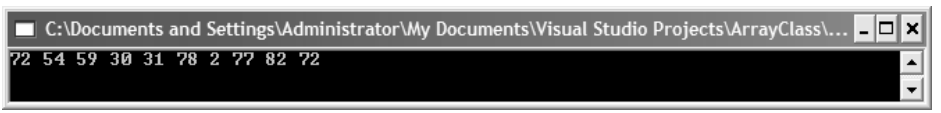
This formula only guarantees that the number will fall in the range of 1 to 100, not that there won't be duplicates in the range. Usually, there won't be that many duplicates so you don't need to worry about it. Finally, to make sure that only an integer is generated, the number generated by this formula is passed to the Int function:

```
Int(100 * Rnd() + 1)
```

Here's another look at a program that uses the CArray class to store numbers, using the random number generator to select the data to store in the array:

```
Sub Main()
  Dim theArray As New CArray(9)
  Dim index As Integer
  For index = 0 To 9
    theArray.Insert(Int(100 * Rnd() + 1))
  Next
  theArray.showArray()
  Console.Read(
End Sub
```

The output from this program is



Bubble Sort

The first sorting algorithm to examine is the bubble sort. The bubble sort is one of the slowest sorting algorithms available, but it is also one of the simplest sorts to understand and implement, which makes it an excellent candidate for our first sorting algorithm.

The sort gets its name because values "float like a bubble" from one end of the list to another. Assuming you are sorting a list of numbers in ascending

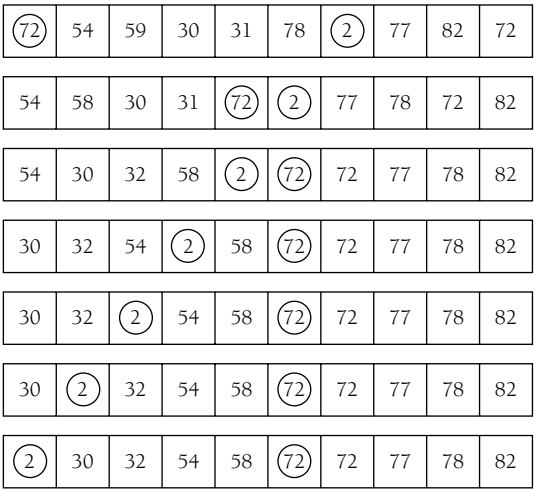


FIGURE 3.1. The Bubble Sort.

order, higher values float to the right whereas lower values float to the left. This behavior is caused by moving through the list many times, comparing adjacent values, and swapping them if the value to the left is greater than the value to the right.

Figure 3.1 illustrates how the bubble sort works. Two numbers from the numbers inserted into the array (2 and 72) from the previous example are highlighted with circles. You can watch how 72 moves from the beginning of the array to the middle of the array, and you can see how 2 moves from just past the middle of the array to the beginning of the array.

Here's the code for the BubbleSort algorithm:

```
Public Sub BubbleSort()
  Dim outer, inner, temp As Integer
  For outer = numElements - 1 To 2 Step -1
    For inner = 0 To outer - 1
      If (arr(inner) > arr(inner + 1)) Then
        temp = arr(inner)
        arr(inner) = arr(inner + 1)
        arr(inner + 1) = temp
      End If
    Next
  Next
End Sub
```

There are several things to notice about this code. First, the code to swap two array elements is written inline rather than as a subroutine. A Swap subroutine might slow down the sorting since it will be called many times. Since the swap code is only three lines long, the clarity of the code is not sacrificed by not putting the code in its own subroutine.

More importantly, notice that the outer loop starts at the end of the array and moves toward the beginning of the array. If you look back at Figure 3.1, you'll see that the highest value in the array is in its proper place at the end of the array. This means that the array indices that are greater than the value "outer" are already in their proper place and the algorithm no longer needs to access these values.

The inner loop starts at the first element of the array and ends when it gets to the next to last position in the array. The inner loop compares the two adjacent positions indicated by inner and inner + 1, swapping them if necessary.

Examining the Sorting Process

One of the things you will probably want to do while developing an algorithm is to view the intermediate results of the code while the program is running. When you're using Visual Studio.NET, it's possible to do this using the Debugging tools available in the Integrated Development Environment (IDE). However, sometimes, all you really want to see is a display of the array (or whatever data structure you are building, sorting, or searching). An easy way to do this is to insert a displaying method in the appropriate place in the code.

For the aforementioned BubbleSort method, the best place to examine how the array changes during the sorting lies between the inner loop and the outer loop. If we do this for each iteration of the two loops, we can view a record of how the values move through the array while they are being sorted.

For example, here is the BubbleSort method modified to display intermediate results:

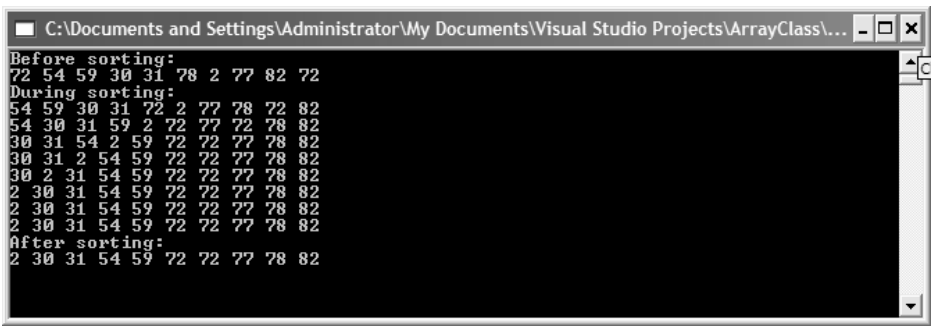
```
Public Sub BubbleSort()
  Dim outer, inner, temp As Integer
  For outer = numElements - 1 To 2 Step -1
    For inner = 0 To outer - 1
      If (arr(inner) > arr(inner + 1)) Then
```

```
          temp = arr(inner)
          arr(inner) = arr(inner + 1)
          arr(inner + 1) = temp
        End If
      Next
      Me.showArray()
    Next
  End Sub
```

The showArray method is placed between the two For loops. If the main program is modified as follows:

```
  Sub Main()
    Dim theArray As New CArray(9)
    Dim index As Integer
    For index = 0 To 9
      theArray.Insert(100 * Rnd() + 1)
    Next
    Console.WriteLine("Before sorting: ")
    theArray.showArray()
    Console.WriteLine("During sorting: ")
    theArray.BubbleSort()
    Console.WriteLine("After sorting: ")
    theArray.showArray()
    Console.Read()
  End Sub
```

the following output is displayed:



Selection Sort

The next sort to examine is the Selection sort. This sort works by starting at the beginning of the array, comparing the first element with the other elements in the array. The smallest element is placed in position 0, and the sort then begins again at position 1. This continues until each position except the last position has been the starting point for a new loop.

Two loops are used in the SelectionSort algorithm. The outer loop moves from the first element in the array to the next to last element; the inner loop moves from the second element of the array to the last element, looking for values that are smaller than the element currently being pointed at by the outer loop. After each iteration of the inner loop, the most minimum value in the array is assigned to its proper place in the array. Figure 3.2 illustrates how this works with the CArray data used before.

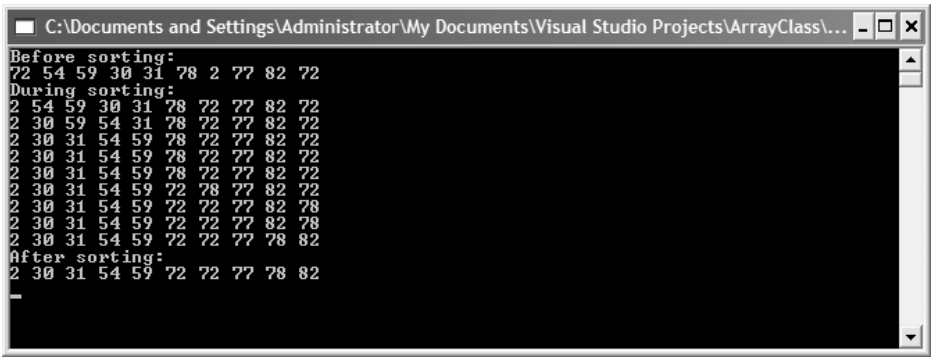
72	54	59	30	31	78	2	77	82	72
2	54	59	30	31	78	72	77	82	72
2	30	59	54	31	78	72	77	82	72
2	30	31	54	59	78	72	77	82	72
2	30	31	54	59	78	72	77	82	72
2	30	31	54	59	78	72	77	82	72
2	30	31	54	59	72	78	77	82	72
2	30	31	54	59	72	72	77	82	78
2	30	31	54	59	72	72	77	82	78
2	30	31	54	59	72	72	77	78	82

FIGURE 3.2. The Selection Sort.

Here's the code to implement the SelectionSort algorithm:

```
Public Sub SelectionSort()
  Dim outer, inner, min, temp As Integer
  For outer = 0 To numElements - 2
    min = outer
    For inner = outer + 1 To numElements - 1
      If (arr(inner) < arr(min)) Then
        min = inner
      End If
    Next
    temp = arr(outer)
    arr(outer) = arr(min)
    arr(min) = temp
  Next
End Sub
```

To demonstrate how the algorithm works, place a call to the showArray() method right before the Next statement that is attached to the outer loop. The output should look something like this:



The final basic sorting algorithm we'll look at in this chapter is one of the simplest to understand: the Insertion sort.

Insertion Sort

The Insertion sort is an analogue to the way we normally sort things numerically or alphabetically. Let's say that I have asked a class of students to each

turn in an index card with his or her name, identification number, and a short biographical sketch. The students return the cards in random order, but I want them to be alphabetized so that I can build a seating chart.

I take the cards back to my office, clear off my desk, and take the first card. The name on the card is Smith. I place it at the top left position of the desk and take the second card. It is Brown. I move Smith over to the right and put Brown in Smith's place. The next card is Williams. It can be inserted at the right without having to shift any other cards. The next card is Acklin. It has to go at the beginning of the list, so each of the other cards must be shifted one position to the right to make room. This is how the Insertion sort works.

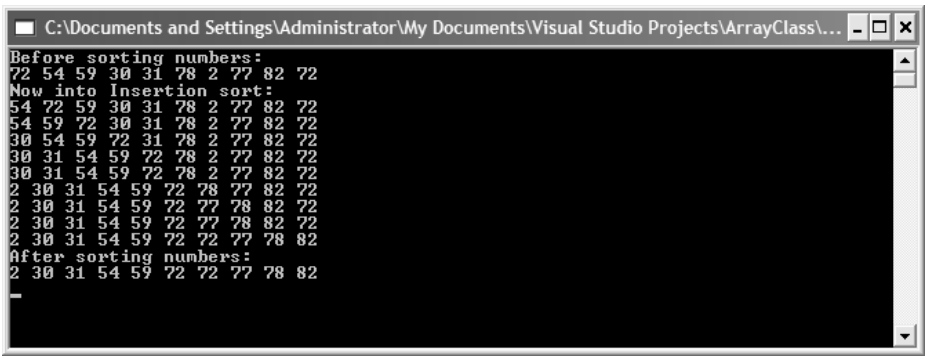
The code for the Insertion sort is as follows:

```
Public Sub InsertionSort()
  Dim inner, outer, temp As Integer
  For outer = 1 To numElements - 1
    temp = arr(outer)
    inner = outer
    While (inner > 0 AndAlso (arr(inner - 1) >= temp))
      arr(inner) = arr(inner - 1)
      inner -= 1
    End While
    arr(inner) = temp
  Next
End Sub
```

The Insertion sort has two loops. The outer loop moves element by element through the array whereas the inner loop compares the element chosen in the outer loop to the element next to it in the array. If the element selected by the outer loop is less than the element selected by the inner loop, array elements are shifted over to the right to make room for the inner loop element, just as described in the preceding example.

The AndAlso operator used in the While loop is used to allow the expression to be short-circuited. Short-circuiting means that the system will determine the value of a complex relational expression from just one part of the expression, without even evaluating the other parts of the expression. The two short-circuiting operators are AndAlso and OrElse. For example, if the first part of an And expression is False and the AndAlso operator is used, the system will evaluate the whole expression as False without testing the other part or parts.

Now let's look at how the Insertion sort works with the set of numbers sorted in the earlier examples. Here's the output:



This display clearly shows that the Insertion sort works not by making exchanges, but by moving larger array elements to the right to make room for smaller elements on the left side of the array.

Timing Comparisons of the Basic Sorting Algorithms

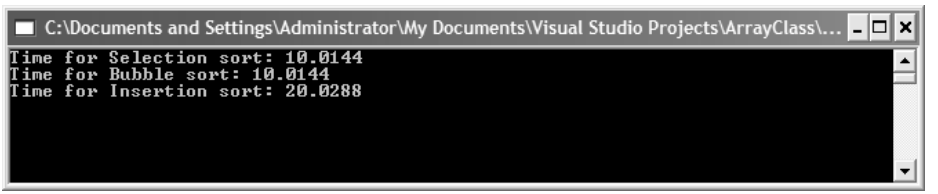
These three sorting algorithms are very similar in complexity and theoretically, at least, should perform similarly. We can use the Timing class to compare the three algorithms to see if any of them stand out from the others in terms of the time it takes to sort a large set of numbers.

To perform the test, we used the same basic code we used earlier to demonstrate how each algorithm works. In the following tests, however, the array sizes are varied to demonstrate how the three algorithms perform with both smaller data sets and larger data sets. The timing tests are run for array sizes of 100 elements, 1,000 elements, and 10,000 elements. Here's the code:

```
Sub Main()
  Dim sortTime As New Timing
  Dim numItems As Integer = 99
  Dim theArray As New CArray(numItems)
  Dim index As Integer
  For index = 0 To numItems
```

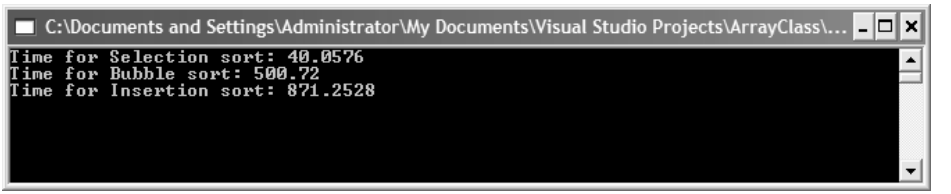
```
      theArray.Insert(CInt((numItems + 1) * Rnd() + 1))
    Next
    sortTime.startTime()
    theArray.SelectionSort()
    sortTime.stopTime()
    Console.WriteLine("Time for Selection sort: " & _
                      sortTime.Result.TotalMilliseconds)
    theArray.clear()
    For index = 0 To numItems
      theArray.Insert(CInt(numItems + 1) * Rnd() + 1)
    Next
    sortTime.startTime()
    theArray.BubbleSort()
    sortTime.stopTime()
    Console.WriteLine("Time for Bubble sort: " & _
                      sortTime.Result.TotalMilliseconds)
    theArray.clear()
    For index = 0 To numItems
      theArray.Insert(CInt((numItems + 1) * Rnd() + 1))
    Next
    sortTime.startTime()
    theArray.InsertionSort()
    sortTime.stopTime()
    Console.WriteLine("Time for Insertion sort: " & _
                      sortTime.Result.TotalMilliseconds)
    Console.Read()
  End Sub
```

The output from this program is as follows:



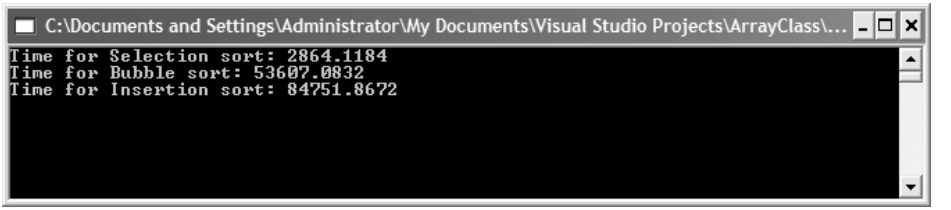
This output indicates that the Selection and Bubble sorts perform at the same speed and the Insertion sort is about half as fast.

Now let's compare the algorithms when the array size is 1,000 elements:



Here we see that the size of the array makes a big difference in the performance of the algorithm. The Selection sort is over 100 times faster than the Bubble sort and over 200 times faster than the Insertion sort.

Increasing the array size to 10,000 elements clearly demonstrates the effect of size on the three algorithms:



The performance of all three algorithms degrades considerably, though the Selection sort is still many times faster than the other two. Clearly, none of these algorithms is ideal for sorting large data sets. There are sorting algorithms, though, that can handle large data sets more efficiently. We'll examine their design and use in Chapter 16.

SUMMARY

In this chapter we discussed three algorithms for sorting data: the Selection sort, the Bubble sort, and the Insertion sort. All of these algorithms are fairly easy to implement and they all work well with small data sets. The Selection sort is the most efficient of the algorithms, followed by the Bubble sort, and then the Insertion sort. As we saw at the end of the chapter, none of these algorithms is well suited for larger data sets (i.e., those with more than a few thousand elements).

Exercises

1. Create a data file consisting of at least 100 string values. You can create the list yourself, or perhaps copy the values from a text file of some type, or you can even create the file by generating random strings. Sort the file using each of the sorting algorithms discussed in the chapter. Create a program that times each algorithm and outputs the times in a similar manner to the output from the last section of this chapter.
2. Create an array of 1000 integers sorted in numerical order. Write a program that runs each sorting algorithm with this array, timing each algorithm, and compare the times. Compare these times to the times for sorting a random array of integers.
3. Create an array of 1000 integers sorted in reverse numerical order. Write a program that runs each sorting algorithm with this array, timing each algorithm, and compare the times.

Chapter 4

Basic Searching Algorithms

Searching for data is a fundamental computer programming task and one that has been studied for many years. This chapter looks at just one aspect of the search problem: searching for a given value in a list (array).

There are two fundamental ways to search for data in a list: the sequential search and the binary search. A sequential search is used when the items in the list are in random order; a binary search is used when the items are sorted in the list.

Sequential Searching

The most obvious type of search is to begin at the beginning of a set of records and move through each record until you find the record you are looking for or you come to the end of the records. This is called a *sequential search*.

A sequential search (also called a linear search) is very easy to implement. Start at the beginning of the array and compare each accessed array element to the value you're searching for. If you find a match, the search is over. If you get to the end of the array without generating a match, then the value is not in the array.

Here's a function that performs a sequential search:

```
Function SeqSearch(ByVal arr() As Integer, _
                   ByVal sValue As Integer) As Integer
```

```
    Dim index As Integer
    For index = 0 To arr.GetUpperBound(0)
      If (arr(index) = sValue) Then
        Return True
      End If
    Next
    Return False
  End Function
```

If a match is found, the function immediately returns True and exits. If the end of the array is reached without the function returning True, then the value being searched for is not in array and the function returns False.

Here's a program to test our implementation of a sequential search:

```
Sub Main()
  Dim numbers(99) As Integer
  Dim numFile As StreamReader
  Dim index As Integer
  Dim searchNumber As Integer
  Dim found As Boolean
  numFile = File.OpenText("c:\numbers.txt")
  For index = 0 To numbers.GetUpperBound(0)
    numbers(index) = CInt(numFile.ReadLine())
  Next
  Console.Write("Enter a number to search for: ")
  searchNumber = CInt(Console.ReadLine())
  found = SeqSearch(numbers, searchNumber)
  If (found) Then
    Console.WriteLine(searchNumber & _
                      " is in the array.")
  Else
    Console.WriteLine(searchNumber & _
                      " is not in the array.")
  End If
  Console.Read()
End Sub
```

The program works by first reading in a set of data from a text file. The data consist of the first 100 integers, stored in the file in a partially random order.

The program then prompts the user to enter a number to search for and calls the SeqSearch function to perform the search.

You can also write the sequential search function so that the function returns the position in the array where the searched-for value is found or a −1 if the value cannot be found. First, let's look at the new function:

```
Function SeqSearch(ByVal arr() As Integer, _
                   ByVal sValue As Integer) As Integer
  Dim index As Integer
  For index = 0 To arr.GetUpperBound(0)
    If (arr(index) = sValue) Then
      Return index
    End If
  Next
  Return -1
End Function
```

The following program uses this function:

```
Sub Main()
  Dim numbers(99) As Integer
  Dim numFile As StreamReader
  Dim index As Integer
  Dim searchNumber As Integer
  Dim foundAt As Integer
  numFile = File.OpenText("c:\numbers.txt")
  For index = 0 To numbers.GetUpperBound(0)
    numbers(index) = CInt(numFile.ReadLine())
  Next
  Console.Write("Enter a number to search for: ")
  searchNumber = CInt(Console.ReadLine())
  found = SeqSearch(numbers, searchNumber)
  If (foundAt >= 0) Then
    Console.WriteLine(searchNumber & " is in the " & _
                      "array at position " & foundAt)
  Else
    Console.WriteLine(searchNumber & " is not in " & _
                      "the array.")
```

```
    End If
    Console.Read()
  End Sub
```

Searching for Minimum and Maximum Values

Computer programs are often asked to search an array (or other data structure) for minimum and maximum values. In an ordered array, searching for these values is a trivial task. Searching an unordered array, however, is a little more challenging.

Let's start by looking at how to find the minimum value in an array. The algorithm is as follows:

1. Assign the first element of the array to a variable as the minimum value.
2. Begin looping through the array, comparing each successive array element with the minimum value variable.
3. If the currently accessed array element is less than the minimum value, assign this element to the minimum value variable.
4. Continue until the last array element is accessed.
5. The minimum value is stored in the variable.

Let's look at a function, FindMin, that implements this algorithm:

```
Function FindMin(ByVal arr() As Integer) As Integer
  Dim min As Integer = arr(0)
  Dim index As Integer
  For index = 1 To arr.GetUpperBound(0)
    If (arr(index) < min) Then
      min = arr(index)
    End If
  Next
  Return min
End Function
```

Notice that the array search starts at position 1 and not position 0. The 0th position is assigned as the minimum value before the loop starts, so we can start making comparisons at position 1.

The algorithm for finding the maximum value in an array works in the same way. We assign the first array element to a variable that holds the maximum amount. Next we loop through the array, comparing each array element with the value stored in the variable, replacing the current value if the accessed value is greater. Here's the code:

```
Function FindMax(ByVal arr() As Integer) As Integer
  Dim max As Integer = arr(0)
  Dim index As Integer
  For index = 1 To arr.GetUpperBound(0)
    If (arr(index) > max) Then
      max = arr(index)
    End If
  Next
  Return max
End Function
```

An alternative version of these two functions could return the position of the maximum or minimum value in the array rather than the actual value.

Making a Sequential Search Faster: Self-Organizing Data

The fastest successful sequential searches occur when the data element being searched for is at the beginning of the data set. You can ensure that a successfully located data item is at the beginning of the data set by moving it there after it has been found.

The concept behind this strategy is that we can minimize search times by putting items that are frequently searched for at the beginning of the data set. Eventually, all the most frequently searched for data items will be located at the beginning of the data set. This is an example of self-organization, in that the data set is organized not by the programmer before the program runs, but by the program while the program is running.

It makes sense to allow your data to organize in this way since the data being searched probably follow the "80–20" rule, meaning that 80% of the searches conducted on your data set are searching for 20% of the data in the data set. Self-organization will eventually put that 20% at the beginning of the data set, where a sequential search will find them quickly.

Probability distributions such as this are called Pareto distributions, named for Vilfredo Pareto, who discovered these distributions by studying the spread of income and wealth in the late 19th century. See Knuth (1998, pp. 399–401) for more on probability distributions in data sets.

We can modify our SeqSearch method quite easily to include self-organization. Here's a first attempt at the method:

```
Public Function SeqSearch(ByVal sValue As Integer) _
    As Boolean
  Dim index, temp As Integer
  For index = 0 To arr.GetUpperBound(0)
    If (arr(index) = sValue) Then
      swap(index, index-1)
      Return True
    End If
  Next
  Return False
End Function
```

If the search is successful, the item found is switched with the element at the first of the array using a swap function such as

```
Private Sub swap(ByRef item1 As Integer, ByRef item2 _
                 As Integer)
  Dim temp As Integer
  temp = arr(item1)
  arr(item1) = arr(item2)
  arr(item2) = temp
End Sub
```

The problem with the SeqSearch method as we've modified it is that frequently accessed items might be moved around quite a bit during the course of many searches. We want to keep items that are moved to the beginning of the data set there and not move them farther back when a subsequent item farther down in the set is successfully located.

There are two ways we can achieve this goal. First, we can only swap found items if they are located away from the beginning of the data set. We only have to determine what is considered to be far enough back in the data set to warrant swapping. Following the "80–20" rule again, we can make a rule that a data item is relocated to the beginning of the data set only if its location lies

outside the first 20% of the items in the data set. Here's the code for this first rewrite:

```
Public Function SeqSearch(ByVal sValue As Integer) _
    As Integer
  Dim index, temp As Integer
  For index = 0 To arr.GetUpperBound(0)
    If (arr(index) = sValue AndAlso _
                     index > (arr.Length * 0.2)) Then
      swap(index, 0)
      Return index
    ElseIf(arr(index) = sValue) Then
      Return index
    End If
  Next
  Return -1
End Function
```

The If–Then statement is short-circuited because if the item isn't found in the data set, there's no reason to test to see where the index is in the data set.

The other way we can rewrite the SeqSearch method is to swap a found item with the element that precedes it in the data set. Using this method, which is similar to how data are sorted using the Bubble sort, the most frequently accessed items will eventually work their way up to the beginning of the data set. This technique also guarantees that if an item is already at the beginning of the data set it won't move back down.

The code for this new version of SeqSearch looks like this:

```
Public Function SeqSearch(ByVal sValue As Integer) _
    As Integer
  Dim index, temp As Integer
  For index = 0 To arr.GetUpperBound(0)
    If (arr(index) = sValue) Then
      swap(index, index - 1)
      Return index
    End If
  Next
  Return -1
End Function
```

Either of these solutions will help your searches when, for whatever reason, you must keep your data set in an unordered sequence. In the next section we will discuss a search algorithm that is more efficient than any of the sequential algorithms already mentioned but that only works on ordered data: the binary search.

Binary Search

When the records you are searching through are sorted into order, you can perform a more efficient search than the sequential search to find a value. This search is called a *binary search*.

To understand how a binary search works, imagine you are trying to guess a number between 1 and 100 chosen by a friend. For every guess you make, the friend tells you if you guessed the correct number, or if your guess is too high, or if your guess is too low. The best strategy then is to choose 50 as the first guess. If that guess is too high, you should then guess 25. If 50 is to low, you should guess 75. Each time you guess, you select a new midpoint by adjusting the lower range or the upper range of the numbers (depending on whether your guess is too high or too low), which becomes your next guess. As long as you follow that strategy, you will eventually guess the correct number. Figure 4.1 demonstrates how this works if the number to be chosen is 82.

We can implement this strategy as an algorithm, the binary search algorithm. To use this algorithm, we first need our data stored in order (ascending, preferably) in an array (though other data structures will work as well). The first step in the algorithm is to set the lower and upper bounds of the search. At the beginning of the search, this means the lower and upper bounds of the array. Then we calculate the midpoint of the array by adding the lower bound and upper bound together and dividing by 2. The array element stored at this position is compared to the searched-for value. If they are the same, the value has been found and the algorithm stops. If the searched-for value is less than the midpoint value, a new upper bound is calculated by subtracting 1 from the midpoint. Otherwise, if the searched-for value is greater than the midpoint value, a new lower bound is calculated by adding 1 to the midpoint. The algorithm iterates until the lower bound equals the upper bound, which indicates the array has been completely searched. If this occurs, a -1 is returned, indicating that no element in the array holds the value being searched for.

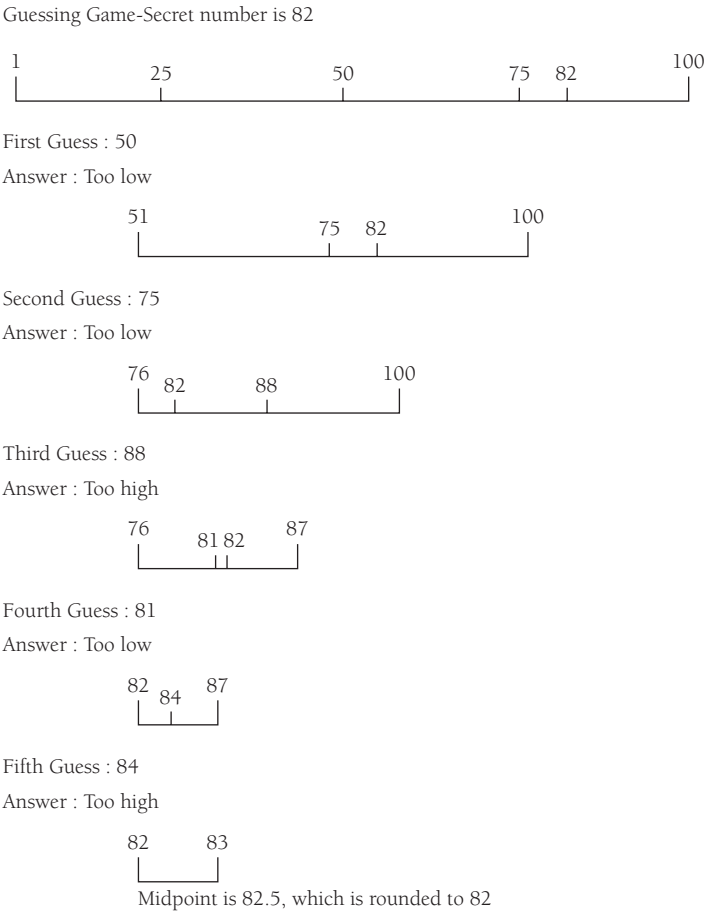


FIGURE 4.1. A Binary Search Analogy.

Here's the algorithm written as a VB.NET function:

```
Public Function binSearch(ByVal value As Integer) _
    As Integer
  Dim upperBound, lowerBound, mid As Integer
  upperBound = arr.GetUpperBound(0)
  lowerBound = 0
  While (lowerBound <= upperBound)
    mid = (upperBound + lowerBound) \ 2
```

```
    If (arr(mid) = value) Then
      Return mid
    ElseIf (value < arr(mid)) Then
      upperBound = mid - 1
    Else
      lowerBound = mid + 1
    End If
  End While
  Return -1
End Function
```

Here's a program that uses the binary search method to search an array:

```
Sub Main()
  Dim mynums As New CArray(9)
  Dim index As Integer
  For index = 0 To 9
    mynums.Insert(CInt(Int(100 * Rnd() + 1)))
  Next
  mynums.SortArr()
  mynums.showArray()
  Dim position As Integer = mynums.binSearch(77, 0, 0)
  If (position > -1) Then
    Console.WriteLine("found item")
    mynums.showArray()
  Else
    Console.WriteLine("Not in the array.")
  End If
  Console.Read()
End Sub
```

A Recursive Binary Search Algorithm

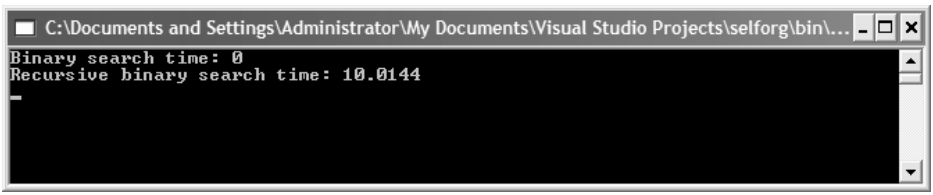
Although our version of the binary search algorithm just developed is correct, it's not really a natural solution to the problem. The binary search algorithm is really a recursive algorithm because, by constantly subdividing the array until we find the item we're looking for (or run out of room in the array), each subdivision is expressing the problem as a smaller version of the original

problem. Viewing the problem this ways leads us to discover a recursive algorithm for performing a binary search.

For a recursive binary search algorithm to work, we have to make some changes to the code. Let's take a look at the code first and then we'll discuss the changes we've made. Here's the code:

```
Public Function RbinSearch(ByVal value As Integer, ByVal _
                           lower As Integer, ByVal _
                           upper As Integer) As Integer
  If (lower > upper) Then
    Return -1
  Else
    Dim mid As Integer
    mid = (upper + lower) \ 2
    If (value < arr(mid)) Then
      RbinSearch(value, lower, mid - 1)
    ElseIf (value = arr(mid)) Then
      Return mid
    Else
      RbinSearch(value, mid + 1, upper)
    End If
  End If
End Function
```

The main problem with the recursive binary search algorithm, compared to the iterative algorithm, is its efficiency. When a 1,000-element array is sorted using both algorithms, the recursive algorithm consistently takes 10 times as much time as the iterative algorithm:



Of course, recursive algorithms are often chosen for reasons other than efficiency, but you should keep in mind that whenever you implement a recursive algorithm you should also look for an iterative solution so that you can compare the efficiency of the two algorithms.

Finally, before we leave the subject of binary search, we should mention that the Array class has a built-in binary search method. It takes two arguments—an array name and an item to search for—and it returns the position of the item in the array, or −1 if the item can't be found.

To demonstrate how the method works, we've written yet another binary search method for our demonstration class. Here's the code:

```
Public Function BSearch(ByVal value As Integer) _
    As Integer
  Return Array.BinarySearch(arr, value)
End Function
```

When the built-in binary search method is compared with our custom-built method, it consistently performs 10 times faster than the custom-built method, which should not be surprising. A built-in data structure or algorithm should always be chosen over one that is custom-built, if the two can be used in exactly the same ways.

Summary

Searching a data set for a value is a ubiquitous computational operation. The simplest method of searching a data set is to start at the beginning and search for the item until either the item is found or the end of the data set is reached. This searching method works best when the data set is relatively small and unordered.

If the data set is ordered, the binary search algorithm makes a better choice. A binary search works by continually subdividing the data set until the item being searched for is found. You can write a binary search algorithm using both iterative and recursive code. The Array class in VB.NET includes a built-in binary search method that should be used whenever a binary search is called for.

Exercises

1. The sequential search algorithm will always find the first occurrence of an item in a data set. Create a new sequential search method that takes a second integer argument indicating which occurrence of an item you want to search for.

2. Write a sequential search method that finds the last occurrence of an item.
3. Run the binary search method on a set of unordered data. What happens?
4. Using the CArray class with the SeqSearch method and the BinSearch method, create an array of 1,000 random integers. Add a new private Integer data member named compCount that is initialized to 0. In each of the search algorithms, add a line of code right after the critical comparison is made that increments compCount by 1. Run both methods, searching for the same number, say 734, with each method. Compare the values of compCount after running both methods. What is the value of compCount for each method? Which method makes the fewest comparisons?

Chapter 5

Stacks and Queues

Data organize naturally as lists. We have already used the Array and ArrayList classes for handling data organized as a list. Although those data structures helped us group the data in a convenient form for processing, neither structure provides a real abstraction for actually designing and implementing problem solutions.

Two list-oriented data structures that provide easy-to-understand abstractions are stacks and queues. Data in a stack are added and removed from only one end of the list; data in a queue are added at one end and removed from the other end of a list. Stacks are used extensively in programming language implementations, from everything from expression evaluation to handling function calls. Queues are used to prioritize operating system processes and to simulate events in the real world, such as teller lines at banks and the operation of elevators in buildings.

VB.NET provides two classes for using these data structures: the Stack class and the Queue class. We'll discuss how to use these classes and look at some practical examples in this chapter.

Stacks, a Stack Implementation, and the Stack Class

The stack is one of the most frequently used data structures, as we just mentioned. We define a stack as a list of items that are accessible only from the

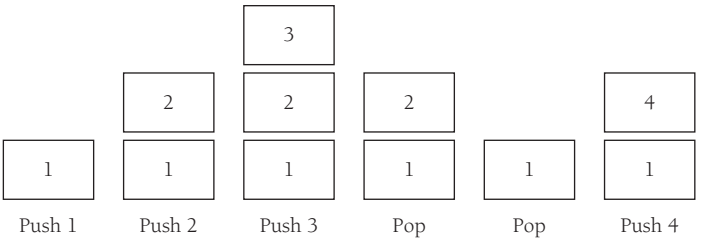


FIGURE 5.1. Pushing and Popping a Stack.

end of the list, which is called the *top* of the stack. The standard model for a stack is a stack of trays at a cafeteria. Trays are always removed from the top, and the when the dishwasher or busperson puts a tray back on the stack, it is placed on the top also. A stack is known as a Last-In, First-Out data structure.

Stack Operations

The two primary operations of a stack are adding items to the stack and taking items off the stack. The *Push* operation adds an item to a stack. We take an item off the stack with a *Pop* operation. These operations are illustrated in Figure 5.1.

The other primary operation to perform on a stack is viewing the top item. The Pop operation returns the top item, but the operation also removes it from the stack. We want to just view the top item without actually removing it. This operation is named *Peek* in VB.NET, though it goes by other names in other languages and implementations (such as *Top*).

Pushing, popping, and peeking are the primary operations we perform when using a stack; however, there are other operations we need to perform and properties we need to examine. It is useful to be able to remove all the items from a stack at one time. A stack is completely emptied by calling the *Clear* operation. It is also useful to know how many items are in a stack at any one time. We do this by calling the *Count* property. Many implementations have a StackEmpty method that returns a True or False value depending on the state of the stack, but we can use the Count property for the same purposes.

The Stack class of the .NET Framework implements all of these operations and properties and many others, but before we examine how to use them, let's look at how you would have to implement a stack if there wasn't a Stack class.

A Stack Class Implementation

A Stack implementation has to use an underlying structure to hold data. We'll choose an ArrayList since we don't have to worry about resizing the list when new items are pushed onto the stack.

Because VB.NET has such useful object-oriented programming features, we'll implement the stack as a class, called CStack. We'll include a constructor method and methods for the aforementioned operations. The Count property is implemented as a property to demonstrate how that's done in VB.NET. Let's start by examining the private data we need in the class.

The most important variable we need is an ArrayList object to store the stack items. The only other data we need to keep track of reside at the top of the stack, which we'll do with a simple Integer variable that functions as an index. The variable is initially set to –1 when a new CStack object is instantiated. Every time a new item is pushed onto the stack, the variable is incremented by 1.

The constructor method does nothing except initialize the index variable to –1. The first method to implement is Push. The code calls the ArrayList Add method and adds the value passed to it to the ArrayList. The Pop method does three things: 1. It calls the RemoveAt method to take the top item off the stack (out of the ArrayList), 2. it decrements the index variable by 1, and 3. it returns the object popped off the stack.

The Peek method is implemented by calling the Item method with the index variable as the argument. The Clear method simply calls an identical method in the ArrayList class. The Count property is written as a read-only property since we don't want to accidentally change the number of items on the stack.

Here's the code:

```
Public Class CStack
  Private p_index As Integer
  Private list As New ArrayList()

  Public Sub New()
    p_index = -1
  End Sub

  ReadOnly Property Count()
    Get
      Return list.Count
    End Get
```

```
    End Property

    Public Sub Push(ByVal value As Object)
      list.Add(value)
      p_index += 1
    End Sub

    Public Function Pop() As Object
      Dim obj As Object = list.Item(p_index)
      list.RemoveAt(p_index)
      p_index -= 1
      Return obj
    End Sub

    Public Sub Clear()
      list.Clear()
      p_index = -1
    End Sub

    Public Function Peek() As Object
      Return list.Item(p_index)
    End Function

  End Class
```

Now let's use this code to write a program that utilizes a stack to solve a problem.

A palindrome is a string that is spelled the same forward and backward. For example, "dad," "madam," and "sees" are palindromes, whereas "hello" is not a palindrome. One way to check strings to see whether they're palindromes is to use a stack. The general algorithm is to read the string character by character, pushing each character onto a stack when it's read. This has the effect of storing the string backward. The next step is to pop each character off the stack, comparing it to the corresponding letter starting at the beginning of the original string. If at any point the two characters are not the same, the string is not a palindrome and we can stop the program. If we get all the way through the comparison, then the string is a palindrome.

Here's the program, starting at Sub Main since we've already defined the CStack class:

```
  Sub Main()
    Dim alist As New CStack()
    Dim ch As String
```

```
    Dim word As String = "sees"
    Dim x As Integer
    Dim isPalindrome As Boolean = True
    For x = 0 To word.Length - 1
      alist.Push(word.Substring(x, 1))
    Next
    x = 0
    While (alist.Count > 0)
      ch = alist.Pop()
      If (ch <> word.Substring(x, 1)) Then
        isPalindrome = False
        Exit While
      End If
      x += 1
    End While
    If (isPalindrome) Then
      Console.WriteLine(word & " is a palindrome.")
    Else
      Console.WriteLine(word & " is not a palindrome.")
    End If
    Console.Write("Press enter to quit")
    Console.Read()
  End Sub
```

The Stack Class

The Stack class is an implementation of the ICollection interface that represents a last-in, first-out collection, or a stack. The class is implemented in the .NET Framework as a circular buffer, which enables space for items pushed on the stack to be allocated dynamically.

The Stack class includes methods for pushing, popping, and peeking values. There are also methods for determining the number of elements in the stack, clearing the stack of all its values, and returning the stack values as an array. Let's start with discussing how the Stack class constructors work.

Stack Constructor Methods

There are three ways to instantiate a stack object. The default constructor instantiates an empty stack that has an initial capacity of 10 values. The

default constructor is called like this:

```
Dim myStack As New Stack()
```

Each time the stack reaches full capacity, the capacity is doubled.

The second Stack constructor method allows you to create a stack object from another collection object. For example, you can pass the constructor an array and a stack is built from the existing array elements:

```
Dim names() As String = {"Raymond", "David", "Mike"}
Dim nameStack As New Stack(names)
```

Executing the Pop method will remove "Mike" from the stack first.

You can also instantiate a stack object and specify the initial capacity of the stack. This constructor comes in handy if you know in advance about how many elements you're going to store in the stack. You can make your program more efficient when you construct your stack in this manner. If your stack has 20 elements in it and it's at total capacity, adding a new element will involve 20 + 1 instructions because each element has to be shifted over to accommodate the new element.

The code for instantiating a Stack object with an initial capacity looks like this:

```
Dim myStack As New Stack(25)
```

Primary Stack Operations

The primary operations you perform with a stack are Push and Pop. Data are added to a stack with the Push method. Data are removed from the stack with the Pop method. Let's look at these methods in the context of using a stack to evaluate simple arithmetic expressions.

This expression evaluator uses two stacks—one for the operands (numbers) and another one for the operators. An arithmetic expression gets stored as a string. We parse the string into individual tokens, using a For loop to read each character in the expression. If the token is a number, it is pushed onto the number stack. If the token is an operator, it is pushed onto the operator stack. Since we are performing infix arithmetic, we wait for two operands to be pushed on the stack before performing an operation. At that point, we

pop the operands and an operand and perform the specified arithmetic. The result is pushed back onto the stack and becomes the first operand of the next operation. This continues until we run out of numbers to push and pop.

Here's the code:

```
Imports System.collections
Module Module1

  Sub Main()
    Dim Nums As New Stack()
    Dim Ops As New Stack()
    Dim expression As String = "5 + 10 + 15 + 20"
    Calculate(Nums, Ops, expression)
    Console.WriteLine(Nums.Pop)
    Console.Write("Press enter to quit")
    Console.Read()
  End Sub

  Public Sub Calculate(ByVal N As Stack, ByVal O As _
                       Stack, ByVal exp As String)
    Dim ch, token, oper As String
    Dim p As Integer = 0
    Dim op1, op2 As Integer
    For p = 0 To exp.Length - 1
      ch = exp.Chars(p)
      If (IsNumeric(ch)) Then
        token &= ch
      End If
      If (ch = " " Or (p = exp.Length - 1)) Then
        If IsNumeric(token) Then
          N.Push(token)
          token = ""
        End If
      ElseIf (ch = "+" Or ch = "-" Or ch = "*" Or _
              ch = "/") Then
        O.Push(ch)
      End If
      If (N.Count = 2) Then
        Compute(N, O)
      End If
```

```
    Next
  End Sub

  Public Sub Compute(ByVal N As Stack, ByVal O As Stack)
    Dim oper1, oper2 As Integer
    Dim oper As String
    oper1 = N.Pop
    oper2 = N.Pop
    oper = O.Pop
    Select Case oper
      Case "+"
        N.Push(oper1 + oper2)
      Case "-"
        N.Push(oper1 - oper2)
      Case "*"
        N.Push(oper1 * oper2)
      Case "/"
        N.Push(oper1 / oper2)
    End Select
  End Sub

End Module
```

It is actually easier to use a Stack to perform arithmetic using postfix expressions. You will get a chance to implement a postfix evaluator in the exercises.

Peek Method

The Peek method lets us look at the value of an item at the top of a stack without having to remove the item from the stack. Without this method, you would have to remove an item from the stack just to get at its value. You will use this method when you want to check the value of the item at the top of the stack before you pop it off:

```
If (IsNumeric(Nums.Peek()) Then
  num = Nums.Pop()
End If
```

Clear Method

The Clear method removes all the items from a stack, setting the item count to zero. It is hard to tell whether the Clear method affects the capacity of a stack, since we can't examine the actual capacity of a stack, so it's best to assume the capacity is set back to the initial default size of 10 elements.

The Clear method comes in handy when you wish to clear a stack because there has been an error in processing. For example, in our expression evaluator, if a division by 0 operation occurs, which is an error, and we want to clear the stack, we invoke the Clear method as follows:

```
If (oper2 = 0) Then
  Nums.Clear()
End If
```

Contains Method

The Contains method determines whether a specified element is located in a stack. The method returns True if the element is found and returns False otherwise. We can use this method to look for a value in the stack that is not currently at the top of the stack, such as a situation in which a certain character in the stack might cause a processing error. Here's the code:

```
If (myStack.Contains(" ") Then
  StopProcessing()
Else
  ContinueProcessing()
End If
```

CopyTo and ToArray Methods

The CopyTo method copies the contents of a stack into an array. The array must be of type Object since that is the data type of all stack objects. The method takes two arguments: an array and the starting array index to begin placing stack elements. The elements are copied in last-in, first-out order, as if

they were popped from the stack. Here's a short code fragment demonstrating a CopyTo method call:

```
Dim myStack As New Stack()
Dim myArray() As Object
Dim x As Integer
For x = 20 To 1 Step -1
  myStack.Push(x)
Next
myStack.CopyTo(myArray, 0)
```

The ToArray method works in a similar manner. You cannot specify a starting array index position, and you must create the new array in an assignment statement. Here's an example:

```
Dim myStack As New Stack()
Dim myArray() As Object
Dim x As Integer
For x = 20 To 1 Step -1
  myStack.Push(x)
Next
myArray = myStack.ToArray()
```

A Stack Class Example: Decimal to Multiple Bases Conversion

Although decimal numbers are used in most business applications, some scientific and technical applications require numbers to be presented in other bases. Many computer system applications require numbers to be in either octal or binary format.

One algorithm that we can use to convert numbers from decimal to octal or binary makes use of a stack. The steps of the algorithm are as follows:

```
Get number
Get base
Loop
  Push the number mod base onto the stack
  Number becomes the number integer-divided by the base
While number not equal to 0
```

Once the loop finishes, you have the converted number, and you can simply pop the individual digits off the stack to see the results. Here's one implementation of the program:

```
Imports System.collections
Module Module1

  Sub Main()
    Dim num As Integer
    Dim base As Integer
    Console.Write("Enter a decimal number: ")
    num = Console.ReadLine
    Console.Write("Enter a base: ")
    base = Console.ReadLine
    Console.Write(num & " converts to ")
    MulBase(num, base)
    Console.WriteLine(" Base " & base)
    Console.Write("Press enter to quit")
    Console.Read()
  End Sub

  Public Sub MulBase(ByVal n As Integer, ByVal b _
                     As Integer)
    Dim Digits As New Stack()
    Do
      Digits.Push(n Mod b)
      n \= b
    Loop While (n <> 0)
    While (Digits.Count > 0)
      Console.Write(Digits.Pop())
    End While
  End Sub

End Module
```

This program illustrates the usefulness of a stack as a data structure for many computational problems. When we convert a decimal number to another form, we start with the right-most digits and work our way to the left. Pushing each digit on the stack as we go works perfectly because, when we finish, the converted digits are in the correct order.

A stack is a very useful data structure, but some applications lend themselves to being modeled using another list-based data structure. Take, for example, the lines that form at the grocery store or your local video rental store. Unlike a stack, where the last one in is the first one out, in these lines the first one in should be the last one out. As another example, consider the list of print jobs sent to a network (or local) printer. The first job sent to the printer should be the first job handled by the printer. These examples are modeled using a list-based data structure called a queue, which is the subject of the next section.

Queues, the Queue Class, and a Queue Class Implementation

A queue is a data structure in which data enter at the rear of a list and are removed from the front of the list. Queues are used to store items in the order in which they occur. Queues are an example of a First-In, First-Out data structure. Queues are used to order processes submitted to an operating system or a print spooler, and simulation applications use queues to model customers waiting in a line.

Queue Operations

The two primary operations involving queues are adding a new item to the queue and removing an item from the queue. The operation for adding a new item is called *Enqueue*, and the operation for removing an item from a queue is called *Dequeue*. The Enqueue operation adds an item at the end of the queue and the Dequeue operation removes an item from the front (or beginning) of the queue. Figure 5.2 illustrates these operations.

The other primary operation to perform on a queue is viewing the beginning item. The Peek method, like its counterpoint in the Stack class, is used to view the beginning item. This method simply returns the item without actually removing it from the queue. There are other properties of the Queue class we can use to aid in our programming. However, before we do discuss them let's look at how we can implement a Queue class.

A Queue Implementation

Implementing the Queue class using an ArrayList is straightforward, as was our implementation of the Stack class. ArrayLists are excellent

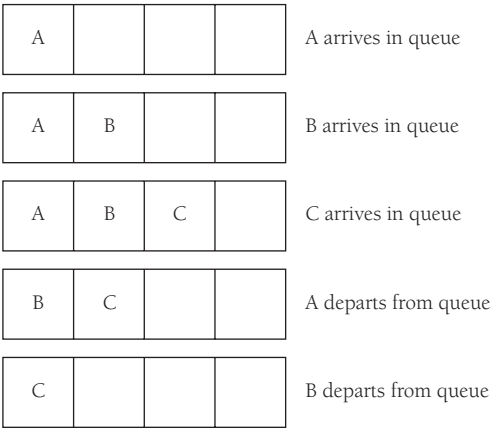


FIGURE 5.2. Queue Operations.

implementation choices for these types of data structures because of their built-in dynamics. When we need to insert an item into our queue, the Arraylist Add method places the item in the next free element of the list. When we need to remove the front item from the queue, the ArrayList moves each remaining item in the list up one element. We don't have to maintain a placeholder, and as a result we can avoid introducing subtle errors in our code.

The following Queue class implementation includes methods for EnQueue, DeQueue, ClearQueue (clearing the queue), Peek, and Count, as well as a default constructor for the class:

```
Public Class CQueue
  Private prear As Integer = 0
  Private pqueue As New ArrayList()
  Public Sub New()
    MyBase.New()
  End Sub
  Public Sub EnQueue(ByVal item As Object)
    pqueue.Add(item)
  End Sub
  Public Sub DeQueue()
    pqueue.RemoveAt(0)
  End Sub
  Public Function Peek() As Object
    Return pqueue.Item(0)
  End Function
```

```
    Public Sub ClearQueue()
      pqueue.Clear()
    End Sub
    Public Function Count() As Integer
      Return pqueue.Count()
    End Function
  End Class
```

The Queue Class: A Sample Application

We have already mentioned the primary methods found in the Queue class and have seen how to use them in our Queue class implementation. We can explore these methods further by looking at a particular programming problem that uses a Queue as its basic data structure. First, though, we need to mention a few of the basic properties of Queue objects.

When a new Queue object is instantiated, the default capacity of the queue is 32 items. By definition, when the queue is full, it is increased by a growth factor of 2.0. This means that, when a queue is initially filled to capacity, its new capacity becomes 64. You are not limited to these numbers however. You can specify a different initial capacity when you instantiate a queue. Here's how:

```
Dim myQueue As New Queue(100)
```

This sets the queue's capacity to 100 items. You can change the growth factor as well. It is the second argument passed to the constructor, as in

```
Dim myQueue As New Queue(32, 3R)
```

This line specifies a growth rate of 3.0 with the default initial capacity. You have to specify the capacity even it's the same as the default capacity since the constructor is looking for a method with a different signature.

As we mentioned earlier, queues are often used to simulate situations where people have to wait in line. One scenario we can simulate with a queue is the annual Single's Night dance at the Elks Lodge. Men and women enter the lodge and stand in line. The dance floor is quite small and there is room for only three dancing couples at a time. As room becomes available on the dance floor, dance partners are chosen by taking the first man and woman in line.

These couples are taken out of the queue and the next pair of dance partners are moved to the front of the queue.

As this action takes place, the program announces the first set of dance partners and who the next people are in line. If there is not a complete couple, the next person in line is announced. If no one is left in line, this fact is displayed.

First, let's look at the data we use for the simulation:

```
F Jennifer Ingram
M Frank Opitz
M Terrill Beckerman
M Mike Dahly
F Beata Lovelace
M Raymond Williams
F Shirley Yaw
M Don Gundolf
F Bernica Tackett
M David Durr
M Mike McMillan
F Nikki Feldman
```

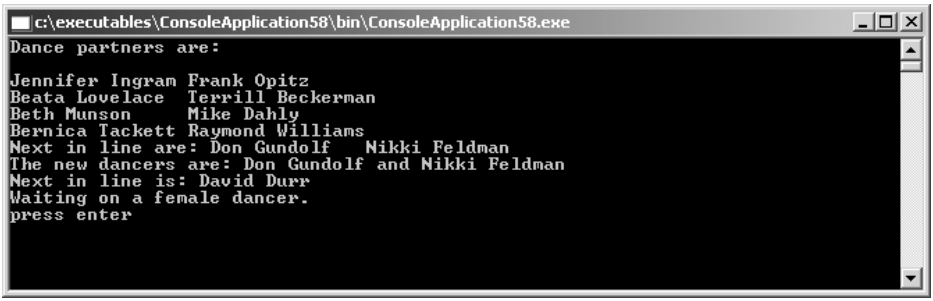
We use a structure to represent each dancer. Two simple String class methods (Chars and Substring) are used to build a dancer. Now here's the program:

```
Imports System.io
Module Module1
  Public Structure Dancer
    Dim Name As String
    Dim Sex As String
  End Structure
  Sub Main()
    Dim males As New Queue()
    Dim females As New Queue()
    formLines(males, females)
    startDancing(males, females)
    If (males.Count > 0 Or females.Count > 0) Then
      headOfLine(males, females)
    End If
    newDancers(males, females)
```

```
  If (males.Count > 0 Or females.Count > 0) Then
    headOfLine(males, females)
  End If
  newDancers(males, females)
  Console.Write("press enter")
  Console.Read()
End Sub

Public Sub newDancers(ByRef male As Queue, ByRef _
                      female As Queue)
  Dim m, w As Dancer
  If (male.Count > 0) And (female.Count > 0) Then
    m = male.Dequeue
    w = female.Dequeue
    Console.WriteLine("The new dancers are: " & _
                      m.Name & " and " & w.Name)
  ElseIf (male.Count > 0) And (female.Count = 0) Then
    Console.WriteLine("Waiting on a female dancer.")
  ElseIf (female.Count > 0) And (male.Count = 0) Then
    Console.WriteLine("Waiting on a male dancer.")
  End If

End Sub
Public Sub headOfLine(ByRef male As Queue, ByRef _
                      female As Queue)
  Dim m, w As Dancer
  If (male.Count > 0) Then
    m = male.Peek()
  End If
  If (female.Count > 0) Then
    w = female.Peek()
  End If
  If (m.Name <> "" And w.Name <> "") Then
    Console.WriteLine("Next in line are: " & m.Name _
                      & Constants.vbTab & w.Name)
  Else
    If (m.Name <> "") Then
      Console.WriteLine("Next in line is: " & m.Name)
    Else
```

```
        Console.WriteLine("Next in line is: " & w.Name)
      End If
    End If
  End Sub

  Public Sub startDancing(ByRef male As Queue, ByRef _
                          female As Queue)
    Dim count As Integer
    Dim m, w As Dancer
    Console.WriteLine("Dance partners are: ")
    Console.WriteLine()
    For count = 0 To 3
      m = male.Dequeue
      w = female.Dequeue
      Console.WriteLine(w.Name & Constants.vbTab _
                        & m.Name)
    Next
  End Sub

  Public Sub formLines(ByRef male As Queue, ByRef _
                       female As Queue)
    Dim d As Dancer
    Dim theFile As File
    Dim inFile As StreamReader
    inFile = theFile.OpenText("c:\dancers.dat")
    Dim line As String
    While (inFile.Peek <> -1)
      line = inFile.ReadLine()
      d.Sex = line.Chars(0)
      d.Name = line.Substring(2, line.Length - 2)
      If (d.Sex = "M") Then
        male.Enqueue(d)
      Else
        female.Enqueue(d)
      End If
    End While
  End Sub

End Module
```

Here's the output from a sample run using our data:



Sorting Data with Queues

Queues are also useful for sorting data. Back in the old days of computing, programs were entered into a mainframe computer via punch cards, where each card held a single program statement. Cards were sorted using a mechanical sorter that utilized binlike structures. We can simulate this process by sorting data using queues. This sorting technique is called a radix sort. It will not be the fastest sort in your programming repertoire, but the radix sort does demonstrate another interesting use of queues.

The radix sort works by making two passes over a set of data, in this case integers in the range 0–99. The first pass sorts the numbers based on the 1's digit and the second pass sorts the numbers based on the 10's digit. Each number is then placed in a bin based on the digit in each of these places. Given the numbers

91 46 85 15 92 35 31 22

the first pass results in this bin configuration:

Bin 0:
Bin 1: 91 31
Bin 2: 92 22
Bin 3:
Bin 4:
Bin 5: 85 15 35
Bin 6: 46
Bin 7:
Bin 8:
Bin 9:

Now put the numbers in order based on which bin they're in:

91 31 92 22 85 15 35 46

Next, take the list and sort by the 10's digit into the appropriate bins:

Bin 0:
Bin 1: 15
Bin 2: 22
Bin 3: 31 35
Bin 4: 46
Bin 5:
Bin 6:
Bin 7:
Bin 8: 85
Bin 9: 91 92

Take the numbers from the bins and put them back into a list, which results in a sorted set of integers:

15 22 31 35 46 85 91 92

We can implement this algorithm by using queues to represent the bins. We need nine queues, one for each digit. We use modulus and integer division for determining the 1's and 10's digits. The rest entails adding numbers to their appropriate queues, taking them out of the queues to resort based on the 1's digit, and then repeating the process for the 10's digit. The result is a sorted list of integers.

Here's the code:

```
Module Module1
  Enum DigitType
    ones = 1
    tens = 10
  End Enum
  Sub Main()
    Dim numQueue(9) As Queue
    Dim done, dten As Integer
    Dim nums() As Integer = {91, 46, 85, 15, 92, 35, _
                             31, 22}
    Dim x, n As Integer
    Dim snum, num As Integer
```

```
    Dim rndom As New Random(100)
    'Build array of queues
    For x = 0 To 9
      numQueue(x) = New Queue()
    Next
    'Display original list
    Console.WriteLine("Original list: ")
    DisplayArray(nums)
    'First pass sort
    RSort(numQueue, nums, 1)
    BuildArray(numQueue, nums)
    Console.WriteLine()
    'Display first pass results
    Console.WriteLine("First pass results: ")
    DisplayArray(nums)
    RSort(numQueue, nums, 10)
    'Second pass sort
    BuildArray(numQueue, nums)
    Console.WriteLine()
    Console.WriteLine("Second pass results: ")
    'Display final results
    DisplayArray(nums)
    Console.WriteLine()
    Console.Write("Press enter")
    Console.Read()
  End Sub

  Public Sub DisplayArray(ByVal n() As Integer)
    Dim x As Integer
    For x = 0 To n.GetUpperBound(0)
      Console.Write(n(x) & " ")
    Next
  End Sub

  Public Sub RSort(ByVal que() As Queue, ByVal n() As _
                   Integer, ByVal digit As DigitType)
    Dim x, snum As Integer
    For x = 0 To n.GetUpperBound(0)
      If digit = DigitType.ones Then
        snum = n(x) Mod 10
```

```
        Else
          snum = n(x) \ 10
        End If
        que(snum).Enqueue(n(x))
      Next
    End Sub

    Public Sub BuildArray(ByVal que() As Queue, ByVal _
                          n() As Integer)
      Dim x, y As Integer
      y = 0
      For x = 0 To 9
        While (que(x).Count > 0)
          n(y) = que(x).Dequeue
          y += 1
        End While
      Next
    End Sub

End Module
```

The RSort subroutine is passed the array of queues, the number array, and a descriptor telling the subroutine whether to sort the 1's digit or the 10's digit. If the sort is on the 1's digit, the program calculates the digit by taking the remainder of the number modulus 10. If the sort is on the 10's digit, the program calculates the digit by taking the number and dividing (in an integer-based manner) by 10.

To rebuild the list of numbers, each queue is emptied by performing successive Dequeue operations while there are items in the queue. This is performed in the BuildArray subroutine. Since we start with the array that is holding the smallest numbers, the number list is built "in order."

Priority Queues: Deriving from the Queue Class

As you know now, a queue is a data structure where the first item placed in the structure is the first item taken out of the structure. This means that the oldest item in the structure is removed first. Many applications, though, require a data structure where an item with the highest priority is removed first, even if it isn't the "oldest" item in the structure. There is a

special case of the Queue made for this type of application—the priority queue.

There are many applications that utilize priority queues in their operations. A good example is process handling in a computer operating system. Certain processes have a higher priority than other processes, such as printing processes, which typically have a low priority. Processes (or tasks) are usually numbered by their priority, with a Priority 0 process having a higher priority than a Priority 20 task.

Items stored in a priority queue are normally constructed as key–value pairs, where the key is the priority level and the value identifies the item. For example, an operating system process might be defined like this:

```
Structure Process
  Dim Priority As Integer
  Dim Name As String
End Structure
```

We cannot use an unmodified Queue object for a priority queue. The DeQueue method simply removes the first item in the queue when it is called. We can, though, derive our own priority queue class from the Queue class, overriding Dequeue to make it do our bidding.

We'll call the class PQueue. We can use all of the Queue methods as is and override the Dequeue method to remove the item that has the highest priority. To remove an item from a queue that is not at the front of the queue, we have to first write the queue items to an array. Then we can iterate through the array to find the highest priority item. Finally, with that item marked, we can rebuild the queue, leaving out the marked item.

Here's the code for the PQueue class:

```
Public Structure pqItem
  Dim Priority As Integer
  Dim Name As String
End Structure

Public Class PQueue
  Inherits Queue

  Public Sub New()
    MyBase.new()
  End Sub
```

```
  Public Overrides Function Dequeue() As Object
    Dim items() As Object
    Dim x, min, minindex As Integer
    items = Me.ToArray 'changes the queue to an array
    min = CType(items(0), pqItem).Priority
    For x = 1 To items.GetUpperBound(0)
      If (CType(items(x), pqItem).Priority < min) Then
        min = CType(items(x), pqItem).Priority
        minindex = x
      End If
    Next
    Me.Clear() 'Clears the queue
    For x = 0 To items.GetUpperBound(0)
      If (x <> minindex And CType(items(x), pqItem). _
          Name <> "") Then
        Me.Enqueue(items(x)) 'rebuild the queue
      End If
    Next
    Return items(minindex)
  End Function

End Class
```

Let's consider a specific example. An emergency waiting room assigns a priority to patients who come in for treatment. A patient presenting symptoms of a heart attack is going to be treated before a patient who has a bad cut. The following program simulates three patients entering an emergency room at approximately the same time. Each patient is seen by the triage nurse, assigned a priority, and added to the queue. The first patient to be treated is the patient removed from the queue by the Dequeue method. The following code demonstrates this simple use of the PQueue class:

```
Sub Main()
  Dim erwait As New PQueue()
  Dim erPatient(4) As pqItem
  Dim nextPatient As pqItem
  Dim x As Integer
  erPatient(0).Name = "Joe Smith"
  erPatient(0).Priority = 1
```

```
    erPatient(1).Name = "Mary Brown"
    erPatient(1).Priority = 0
    erPatient(2).Name = "Sam Jones"
    erPatient(2).Priority = 3
    For x = 0 To erPatient.GetUpperBound(0)
      erwait.Enqueue(erPatient(x))
    Next
    nextPatient = erwait.Dequeue
    Console.WriteLine(nextPatient.Name)
    Console.Write("Press enter")
    Console.Read()
  End Sub
```

The output of this program is "Mary Brown," since she has a higher priority than the other patients.

SUMMARY

Learning to use data structures appropriately and efficiently is one of the skills that separates the expert programmer from the average one. The expert programmer recognizes that organizing a program's data into an appropriate data structure makes it easier to work with the data. In fact, thinking through a computer programming problem using data abstraction makes it easier to come up with a good solution to the problem in the first place.

We looked at using two very common data structures in this chapter—the stack and the queue. Stacks are used for solving many different types of problems in computer programming, especially in systems programming areas such as interpreters and compilers. We also saw how we can use stacks to solve more generic problems, such as determining if a word is a palindrome.

Queues also have many applications. Operating systems use queues for ordering processes (via priority queues) and queues are used quite often for simulating real-world processes. Finally, we used the Queue class to derive a class for implementing a priority queue. The ability to derive new classes from classes in the .NET Framework class library is one of the major strengths of the .NET version of Visual Basic.

Exercises

1. You can use a Stack to check whether a programming statement or a formula has balanced parentheses. Write a Windows application that provides a textbox for the user to enter an expression with parenthesis. Provide a Check Parens button that, when clicked, runs a program that checks the number of parentheses in the expression and highlights a parenthesis that is unbalanced.
2. A postfix expression evaluator works on arithmetic statements that take the following form:

   op1 op2 operator . . .

   Using two stacks, one for the operands and one for the operators, design and implement a Calculator class that converts infix expressions to postfix expressions and then uses the stacks to evaluate the expressions.
3. Design a help-desk priority manager. Store help requests stored in a text file with the following structure:

   priority, id of requesting party, time of request

   The priority is an integer in the range 1–5 with 1 being the least important and 5 being the most important. The id is a four-digit employee identification number and the time is in TimeSpan.Hours, TimeSpan.Minutes, TimeSpan.Seconds format. Write a Windows application that, during the Form_Load event, reads five records from the data file containing help requests, prioritizes the list using a priority queue, and displays the list in a list box. Each time a job is completed, the user can click on the job in the list box to remove it. When all five jobs are completed, the application should automatically read five more data records, prioritize them, and display them in the list box.

Chapter 6

The BitArray Class

The BitArray class is used to represent sets of bits in a compact fashion. Bit sets can be stored in regular arrays, but we can create more efficient programs if we use data structures specifically designed for bit sets. In this chapter we'll look at how to use this data structure and examine some problems that can be solved using sets of bits.

Since many VB.NET programmers have not been properly introduced to working with binary numbers, this chapter also includes a review of binary numbers, bitwise operators, and bitshift operators.

A Motivating Problem

Let's look at a problem we will eventually solve using the BitArray class. The problem involves finding prime numbers. An ancient method, discovered by the third-century B.C. Greek philosopher Eratosthenes, is called the sieve of Eratosthenes. This method involves filtering numbers that are multiples of other numbers, until the only numbers left are primes. For example, let's determine the prime numbers in the set of the first 100 integers. We start with 2, which is the first prime. We move through the set removing all numbers that are multiples of 2. Then we move to 3, which is the next prime. We move through the set again, removing all numbers that are multiples of 3. Then we

move to 5, and so on. When we are finished, all that will be left are prime numbers.

We'll first solve this problem using a regular array. The approach we'll use, which is similar to how we'll solve the problem using a BitArray, is to initialize an array of 100 elements, with each element set to the value 1. Starting with index 2 (since 2 is the first prime), each subsequent array index is checked to see first if its value is 1 or 0. If the value is 1, then it is checked to see if it is a multiple of 2. If it is, the value at that index is set to 0. Then we move to index 3, do the same thing, and so on.

To write the code to solve this problem, we'll use the CArray class developed earlier. The first thing we need to do is create a method that performs the sieve. Here's the code:

```
Public Sub genPrimes()
  Dim inner, outer As Integer
  Dim temp As Integer
  For outer = 2 To arr.GetUpperBound(0)
    For inner = outer + 1 To arr.GetUpperBound(0)
      If (arr(inner) = 1) Then
        If ((inner Mod outer) = 0) Then
          arr(inner) = 0
        End If
      End If
    Next
  Next
End Sub
```

Now all we need is a method to display the primes:

```
Public Sub showPrimes()
  Dim index As Integer
  For index = 2 To arr.GetUpperBound(0)
    If (arr(index) = 1) Then
      Console.Write(index & " ")
    End If
  Next
End Sub
```

And here's a program to test our code:

```
Sub Main()
  Dim size As Integer = 100
  Dim primes As New CArray(size - 1)
  Dim index As Integer
  For index = 0 To size - 1
    primes.Insert(1)
  Next
  primes.genPrimes()
  primes.showPrimes()
  Console.Read()
End Sub
```

This code demonstrates how to use the sieve of Eratosthenes using integers in the array, but it suggests that a solution can be developed using bits, since each element in the array is a 0 or a 1. Later in the chapter we'll examine how to use the BitArray class, both to implement the sieve of Eratosthenes and for other problems that lend themselves to sets of bits.

Bits and Bit Manipulation

Before we look at the BitArray class, we need to discuss how bits are used in VB.NET, since working at the bit level is not something most VB.NET programmers are familiar with. In this section, we'll examine how bits are manipulated in VB.NET, primarily by looking at how to use the bitwise operators to manipulate Byte values.

The Binary Number System

Before we look at how to manipulate Byte values, let's review a little about the binary system. Binary numbers are strings of 0s and 1s that represent base 10 (or decimal) numbers in base 2. For example, the binary number for the integer 0 is

00000000

whereas the binary number for the integer 1 is

00000001

Here are the integers 0–9 displayed in binary:

00000000 = 0d (where d signifies a decimal number)
00000001 = 1d
00000010 = 2d
00000011 = 3d
00000100 = 4d
00000101 = 5d
00000110 = 6d
00000111 = 7d
00001000 = 8d
00001001 = 9d

The best way to convert a binary number to its decimal equivalent is to use the following scheme. Each binary digit, starting with the rightmost digit, represents a successively larger power of 2. If the digit in the first place is a 1, then that represents 2^0. If the second position has a 1, that represents 2^1, and so on.

The binary number

00101010

is equivalent to

$$0 + 2^1 + 0 + 2^3 + 0 + 2^5 + 0 + 0$$
$$= 0 + 2 + 0 + 8 + 0 + 32 + 0 + 0 = 42$$

Bits are usually displayed in sets of eight bits, which makes a byte. The largest number we can express in eight bits is 255, which in binary is

11111111

or

$$1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255$$

A number greater than 255 must be stored in 16 bits. For example, the binary number representing 256 is

00000001 00000000

It is customary, though not required, to separate the lower eight bits from the upper eight bits.

Manipulating Binary Numbers: The Bitwise and Bit-Shift Operators

Binary numbers are not operated on using the standard arithmetic operators. You have to use the bitwise operators (And, Or, Not) or the bit-shift operators (<<, >>, and >>>). In this section, we explain how these operators work; their use via VB.NET applications will be demonstrated in later sections.

First we'll examine the bitwise operators. These are the logical operators with which most programmers are already familiar and are used to combine relational expressions to compute a single Boolean value. With binary numbers, the bitwise operators are used to compare two binary numbers bit by bit, yielding a new binary number.

The bitwise operators work the same way they do with Boolean values. When working with binary numbers, a True bit is equivalent to 1 and a False bit is equivalent to 0. To determine how the bitwise operators work on bits, then, we can use truth tables just as we would with Boolean values. The first two columns in a row are the two operands and the third column is the result of the operation. The truth table (in Boolean) for the And operator is as follows:

True	True	True
True	False	False
False	True	False
False	False	False

The equivalent table for bit values is

1	1	1
1	0	0
0	1	0
0	0	0

The Boolean truth table for the Or operator is

True	True	True
True	False	True
False	True	True
False	False	False

The equivalent table for bit values is

1	1	1
1	0	1
0	1	1
0	0	0

Finally, there is the Xor operator. This is the least known of the bitwise operators because it is not used in logical operations performed by computer programs. When two bits are compared using the Xor operator, the resulting bit is a 1 if exactly one bit of the two operands is 1. Here is the table:

1	1	0
1	0	1
0	1	1
0	0	0

With these tables in mind, we can combine binary numbers with these operators to yield new binary numbers. Here are some examples:

```
00000001 And 00000000 -> 00000000
00000001 And 00000001 -> 00000001
00000010 And 00000001 -> 00000000
```

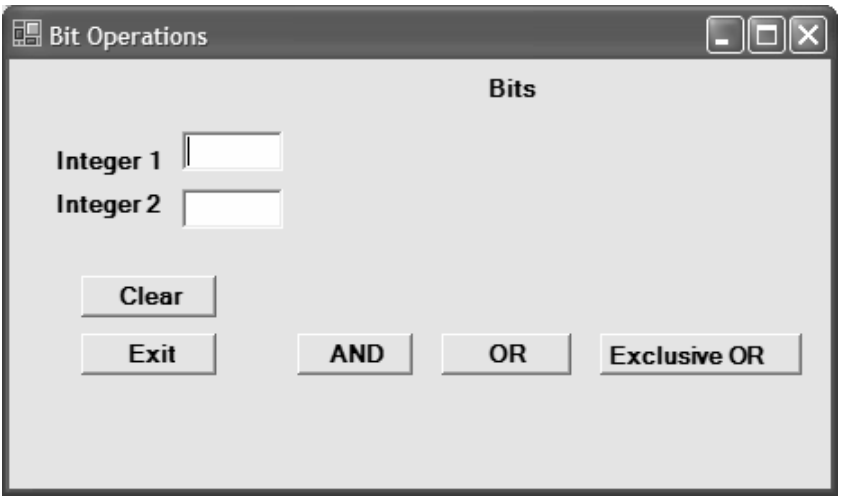
```
00000000 Or 00000001 -> 00000001
00000001 Or 00000000 -> 00000001
00000010 Or 00000001 -> 00000011

00000000 Xor 00000001 -> 00000001
00000001 Xor 00000000 -> 00000001
00000001 Xor 00000001 -> 00000000
```

Now let's look at a VB.NET Windows application that shows in detail how the bitwise operators work.

A BitWise Operator Application

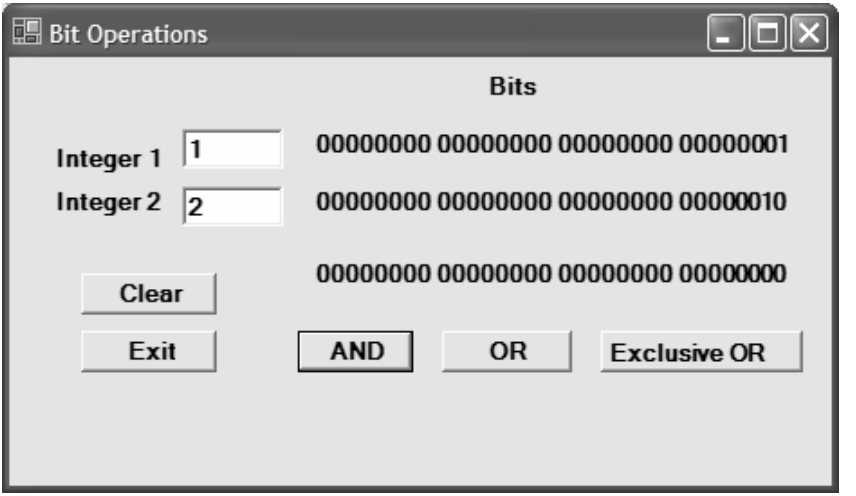
We can demonstrate how the bitwise operators work in VB.NET using a Windows application that applies these operators to a pair of values. We'll use the ConvertBits method developed earlier to help us work with the bitwise operators.

First let's look at the user interface for the application, which goes a long way to explaining how the application works:

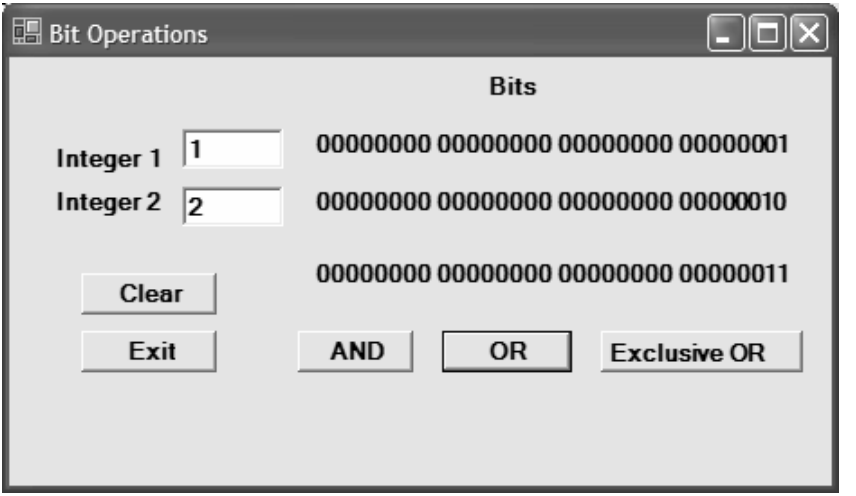


Two integer values are entered and the user selects one of the bitwise operator buttons. The bits that make up each integer value are displayed along with the bit string resulting from the bitwise operation. Here is one

example, combining the values 1 and 2 with an AND (known as ANDing):



Here is the result of ORing the same two values:



Here is the code for the operation:

```
Imports System.text
Public Class Form1
  Inherits System.Windows.Forms.Form
```

```
#Region " Windows Form Designer generated code "
'This code is left out to save space
#End Region

  Private Sub btnAnd_Click(ByVal sender As _
      System.Object, ByVal e As System.EventArgs) _
      Handles btnAnd.Click

    Dim val1, val2 As Integer
    val1 = Int32.Parse(txtInt1.Text)
    val2 = Int32.Parse(txtInt2.Text)
    lblInt1Bits.Text = ConvertBits(val1).ToString
    lblInt2Bits.Text = ConvertBits(val2).ToString
    lblBitResult.Text = ConvertBits(val1 And val2). _
                        ToString
  End Sub

  Private Function ConvertBits(ByVal val As Integer) _
       As StringBuilder
    Dim dispMask As Integer = 1 << 31
    Dim bitBuffer As New StringBuilder(35)
    Dim index As Integer
    For index = 1 To 32
      If (val And dispMask) = 0 Then
        bitBuffer.Append("0")
      Else
        bitBuffer.Append("1")
      End If
      val <<= 1
      If (index Mod 8 = 0) Then
        bitBuffer.Append(" ")
      End If
    Next
    Return bitBuffer
  End Function

  Private Sub btnClear_Click(ByVal sender As _
      System.Object, ByVal e As System.EventArgs) _
      Handles btnClear.Click

    txtInt1.Text = ""
```

```
    txtInt2.Text = ""
    lblInt1Bits.Text = ""
    lblInt2Bits.Text = ""
    lblBitResult.Text = ""
    txtInt1.Focus()
  End Sub

  Private Sub btnOr_Click(ByVal sender As _
      System.Object, ByVal e As System.EventArgs) _
      Handles btnOr.Click

    Dim val1, val2 As Integer
    val1 = Int32.Parse(txtInt1.Text)
    val2 = Int32.Parse(txtInt2.Text)
    lblInt1Bits.Text = ConvertBits(val1).ToString
    lblInt2Bits.Text = ConvertBits(val2).ToString
    lblBitResult.Text = ConvertBits(val1 Or val2). _
                        ToString
  End Sub

  Private Sub btnXor_Click(ByVal sender As _
      System.Object, ByVal e As System.EventArgs) _
      Handles btnXor.Click

    Dim val1, val2 As Integer
    val1 = Int32.Parse(txtInt1.Text)
    val2 = Int32.Parse(txtInt2.Text)
    lblInt1Bits.Text = ConvertBits(val1).ToString
    lblInt2Bits.Text = ConvertBits(val2).ToString
    lblBitResult.Text = ConvertBits(val1 Xor val2). _
                        ToString
  End Sub

End Class
```

BitShift Operators

A binary number consists only of 0s and 1s, with each position in the number representing either the quantity 0 or a power of 2. There are two operators you can use in VB.NET to change the position of bits in a binary number: the left shift operator (<<) and the right shift operator (>>).

Each of these operators take two operators—a value (left) and the number of bits to shift (right). For example, if we write

```
1 << 1
```

the result is 00000010. And we can reverse that result by writing 2 >> 1. Let's look at a more complex example. The binary number representing the quantity 3 is

```
00000011
```

If we write 3 << 1, the result is

```
00000110
```

And if we write 3 << 2, the result is

```
00001100
```

The right shift operator works exactly in reverse of the left shift operator. For example, if we write

```
3 >> 1
```

the result is 00000001.

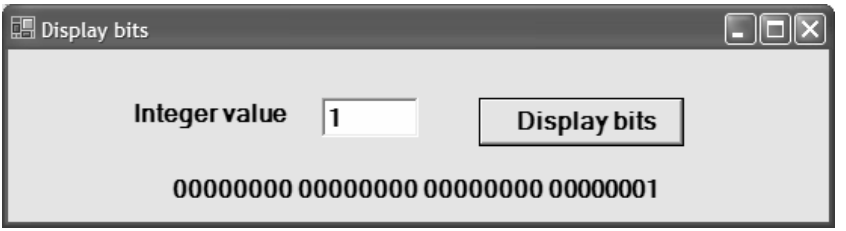
In a later section we'll see how to write a Windows application that demonstrates the use of the bit-shift operators.

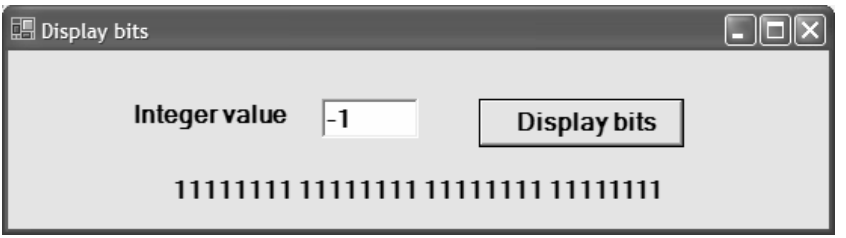
An Integer-to-Binary Converter Application

In this section we demonstrate how to use a few of the bitwise operators to determine the bit pattern of an integer value. The user enters an integer and presses the Display Bits button. The integer value converted to binary is displayed in four groups of eight bits in a label.

The key tool we use to convert an integer into a binary number is a *mask*. The conversion function uses the mask to hide some of the bits in a number while displaying others. When the mask and the integer value (the operands) are combined with the AND operator, the result is a binary string representing the integer value.

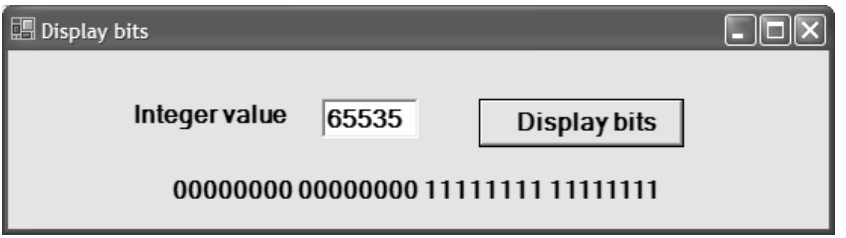
First, let's look at several integer values and their representative binary values:



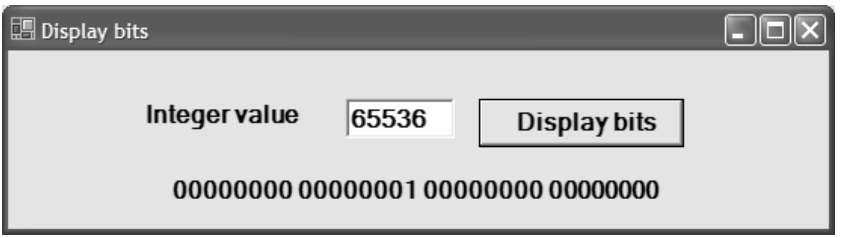


Binary representation of negative integers in computers is not always so straightforward, as shown by this example. For more information, consult a good book on assembly language and computer organization.

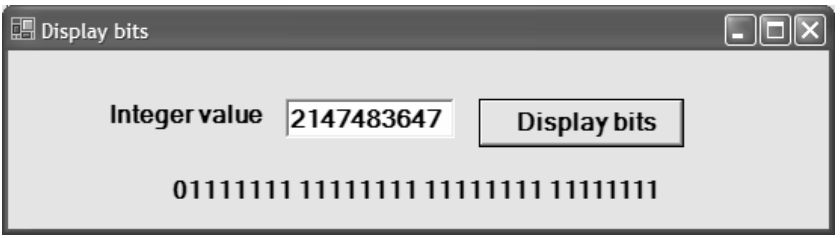
Consider another example:



As you can see, this last value, 65535, is the largest amount that can fit into 16 bits. If we increase the value to 65536, we get the following:



Finally, let's look at what happens when we convert the largest number we can store in an Integer variable in VB.NET:



If we try to enter value 2147483648, we get an error. You may think that the leftmost bit position is available, but it's not because that bit is used to work with negative numbers.

Now let's examine the code that drives this application:

```
Option Strict On
Imports System.text
Public Class Form1
  Inherits System.Windows.Forms.Form

  #Region " Windows Form Designer generated code "
  'Windows generated code is not shown in this listing

  Private Sub DispBits_Click(ByVal sender As System. _
      Object, ByVal e As System.EventArgs) Handles _
      Button1.Click
    Dim bitValues As String
    bitValues = ConvertBits(CInt(byteVal.Text)).ToString
    bitVals.Text = bitValues
  End Sub

  Private Function ConvertBits(ByVal val As Integer) _
      As StringBuilder
    Dim dispMask As Integer = 1 << 31
    Dim bitBuffer As New StringBuilder(35)
    Dim index As Integer
    For index = 1 To 32
      If (val And dispMask) = 0 Then
        bitBuffer.Append("0")
      Else
```

```
        bitBuffer.Append("1")
      End If
      val <<= 1
      If (index Mod 8 = 0) Then
        bitBuffer.Append(" ")
      End If
    Next
    Return bitBuffer
  End Function

End Class
```

The ConvertBits function performs most of the work of the application. The variable dispMask holds the bit mask and the variable bitBuffer holds the string of bits built by the function. We declare bitBuffer as a StringBuilder type to allow us to use the class's Append method to build the string without using concatenation.

The binary string is built in the For loop, which is iterated 32 times since we are building a 32-bit string. To build the bit string, we AND the value with the bit mask. If the result of the operation is 0, a 0 is appended to the string. If the result is 1, a 1 is appended to the string. We then perform a left bit shift on the value in to then compute the next bit in the string. Finally, after every eight bits, we append a space to the string to separate the four 8-bit substrings, making them easier to read.

A Bit-Shift Demonstration Application

This section discusses a Windows application that demonstrates how the bit-shifting operators work. The application provides textboxes for the two operands (a value to shift and the number of bits to shift) and two labels used to show both the original binary representation of the left operand and the resulting bits that result from a bit-shifting operation. The application has two buttons that indicate a left shift or a right shift, as well as Clear and Exit buttons.

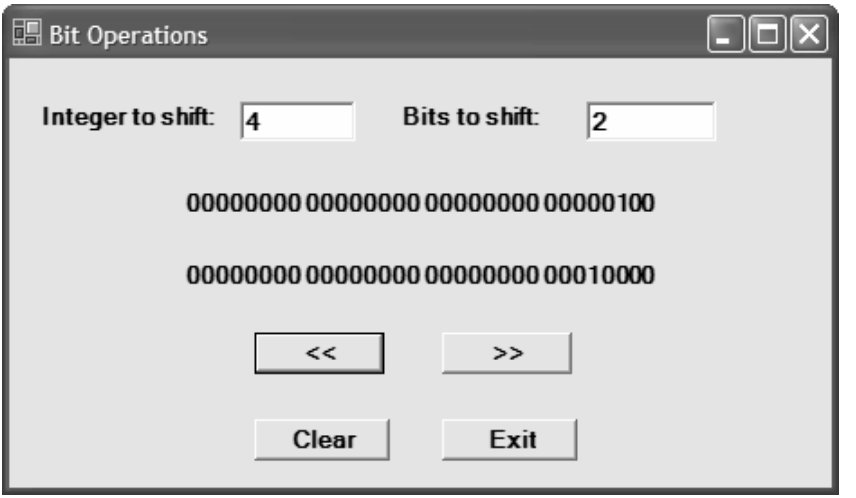
Here's the code for the program:

```
Imports System.text
Public Class Form1
```

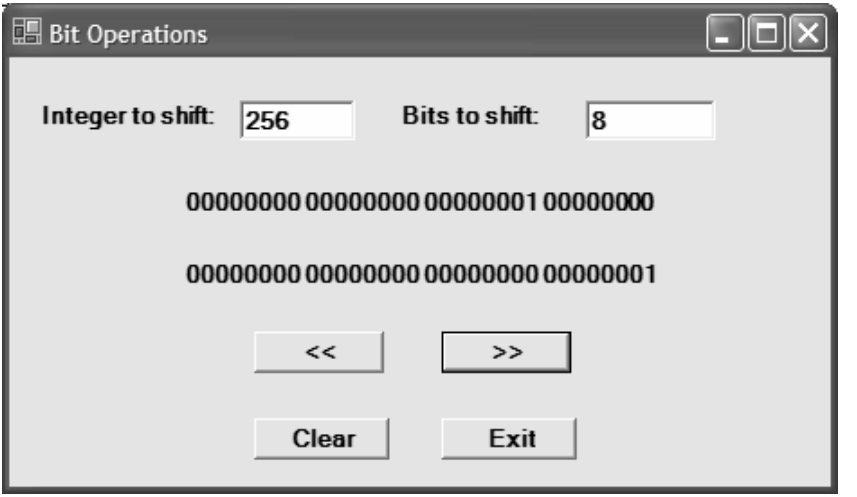
```
  Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "
  'The region containing VS.NET-generated code
   is not shown

  Private Function ConvertBits(ByVal val As Integer) _
      As StringBuilder
    Dim dispMask As Integer = 1 << 31
    Dim bitBuffer As New StringBuilder(35)
    Dim index As Integer
    For index = 1 To 32
      If (val And dispMask) = 0 Then
        bitBuffer.Append("0")
      Else
        bitBuffer.Append("1")
      End If
      val <<= 1
      If (index Mod 8 = 0) Then
        bitBuffer.Append(" ")
      End If
    Next
    Return bitBuffer
  End Function

  Private Sub btnClear_Click(ByVal sender As System. _
      Object, ByVal e As System.EventArgs) _
      Handles btnClear.Click
    txtInt1.Text = ""
    txtBitShift.Text = ""
    lblInt1Bits.Text = ""
    lblOrigBits.Text = ""
    txtInt1.Focus()
  End Sub

  Private Sub btnExit_Click(ByVal sender As System. _
      Object, ByVal e As System.EventArgs) _
      Handles btnExit.Click
    End
  End Sub
```

```
  Private Sub btnLeft_Click(ByVal sender As System. _
      Object, ByVal e As System.EventArgs) _
      Handles btnLeft.Click
    Dim value As Integer
    value = Int32.Parse(txtInt1.Text)
    lblOrigBits.Text = ConvertBits(value).ToString
    value <<= Int32.Parse(txtBitShift.Text)
    lblInt1Bits.Text = ConvertBits(value).ToString
  End Sub

  Private Sub btnRight_Click(ByVal sender As System. _
      Object, ByVal e As System.EventArgs) _
      Handles btnRight.Click
    Dim value As Integer
    value = Int32.Parse(txtInt1.Text)
    lblOrigBits.Text = ConvertBits(value).ToString
    value >>= Int32.Parse(txtBitShift.Text)
    lblInt1Bits.Text = ConvertBits(value).ToString
  End Sub

End Class
```

Let's look at some examples of the application in action. Here is 4 << 2:



Here is 256 >> 8:



The BitArray Class

The BitArray class is used to work with sets of bits. A bit set is used to efficiently represent a set of Boolean values. A BitArray is very similar to an ArrayList, in that BitArrays can be resized dynamically, adding bits when needed without worrying about going beyond the upper bound of the array.

Using the BitArray Class

A BitArray is created by instantiating a BitArray object, passing the number of bits you want in the array into the constructor:

```
Dim BitSet As New BitArray(32)
```

The 32 bits of this BitArray are set to False. If we wanted them to be True, we could instantiate the array like this:

```
Dim BitSet As New BitArray(32, True)
```

The constructor can be overloaded many different ways, but we'll look at just one more constructor method here. You can instantiate a BitArray using

an array of Byte values. For example, consider the following code:

```
Dim ByteSet As Byte() = {1, 2, 3, 4, 5}
Dim BitSet As New BitArray(ByteSet)
```

The BitSet BitArray now contains the bits for the Byte values 1, 2, 3, 4, and 5.

Bits are stored in a BitArray with the most significant bit in the leftmost (index 0) position. This can be confusing to read when you are accustomed to reading binary numbers from right to left. For example, here are the contents of an eight-bit BitArray that is equal to the number 1:

```
True False False False False False False False
```

Of course, we are more accustomed to viewing a binary number with the most significant bit to the right, as in

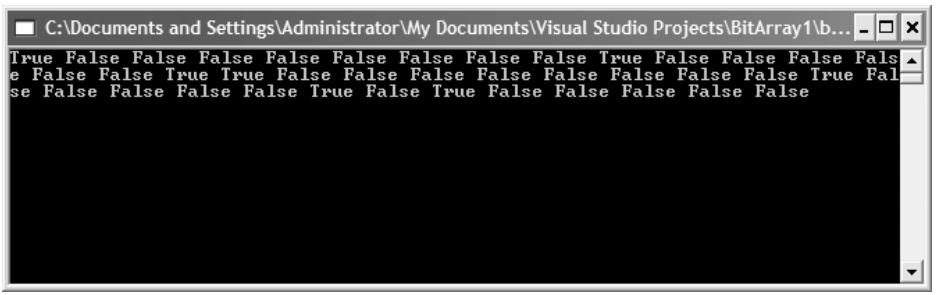
0 0 0 0 0 0 0 1

We will have to write our own code to change both the display of bit values (rather than Boolean values) and the order of the bits.

If you have Byte values in the BitArray, each bit of each Byte value will display when you loop through the array. Here is a simple program fragment to loop through a BitArray of Byte values:

```
Dim ByteSet As Byte() = {1, 2, 3, 4, 5}
Dim BitSet As New BitArray(ByteSet)
For bits = 0 To BitSet.Count - 1
  Console.Write(BitSet.Get(bits) & " ")
Next
```

The output looks like this:



This output is next to impossible to read and it doesn't really reflect what is stored in the array. We'll see later how to make this type of BitArray easier to understand. First, though, we need to learn how to retrieve a bit value from a BitArray.

The individual bits stored in a BitArray are retrieved using the Get method. This method takes an Integer argument—the index of the value we wish to retrieve, and the return value is a bit value represented by True or False. The Get method is used in the previous code segment to display the bit values from the BitSet BitArray.

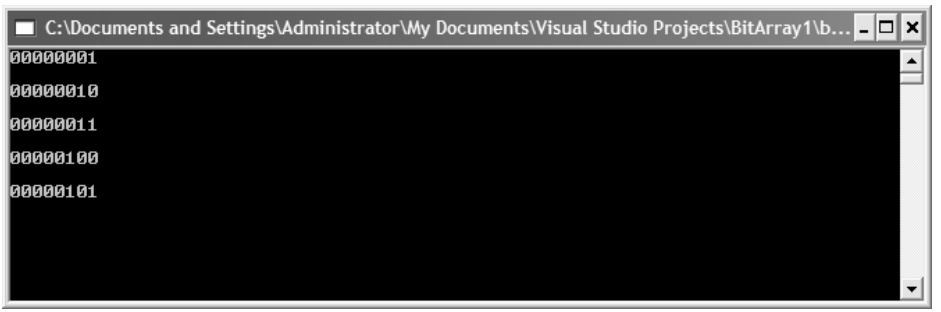
If the data we are storing in a BitArray are actually binary values (i.e., values that should be shown as 0s and 1s), we need a way to display the actual 1s and 0s of the values in the proper order—starting at the right rather than the left. Although we cannot change the internal code used by the BitArray class, we can write external code that gives us the output we want.

The following program creates a BitArray of five Byte values (1,2,3,4,5) and displays each byte in its proper binary form:

```
Module Module1

  Sub Main()
    Dim bits As Integer
    Dim binNumber(7) As String
    Dim binary As Integer
    Dim ByteSet As Byte() = {1, 2, 3, 4, 5}
    Dim BitSet As New BitArray(ByteSet)
    Dim i As Integer
    bits = 0
    binary = 7
    For i = 0 To BitSet.Count - 1
      If (BitSet.Get(i) = True) Then
        binNumber(binary) = "1"
      Else
        binNumber(binary) = "0"
      End If
      bits += 1
      binary -= 1
      If (bits Mod 8 = 0) Then
        binary = 7
```

```
            bits = 0
            Dim index As Integer
            For index = 0 To 7
              Console.Write(binNumber(index))
            Next
            Console.WriteLine()
            Console.WriteLine()
          End If
        Next
        Console.Read()
      End Sub

  End Module
```

Here is the output:



There are two arrays used in this program. The first array, BitSet, is a BitArray that holds the Byte values (in bit form). The second array, binNumber, is just a string array used to store a binary string. This binary string will be built from the bits of each Byte value, starting at the last position (7) and moving forward to the first position (0).

Each time a bit value is encountered, it is first converted to 1 (if True) or 0 (if False) and then placed in the proper position. Two variables are used to tell us where we are in the BitSet array (bits) and in the binNumber array (binary). We also need to know when we've converted eight bits and are finished with a number. We do this by taking the current bit value (in the variable bits) modulo 8. If there is no remainder then we're at the eighth bit and we can write out a number. Otherwise, we continue in the loop.

We've written this program completely in Sub Main(), but in the exercises at the end of the chapter you'll get an opportunity to clean the program up by creating a class or even extending the BitArray class to include this conversion technique.

More BitArray Class Methods and Properties

In this section, we discuss a few more of the BitArray class methods and properties you're most likely to use when working with the class.

The Set method is used to set a particular bit to a value. The method is used like this:

```
BitArray.Set(bit, value)
```

where *bit* is the index of the bit to set, and *value* is the Boolean value you wish to assign to the bit. (Although Boolean values are supposed to be used here, you can actually use other values, such as 0s and 1s. You'll see how to do this in the next section.)

The SetAll method allows you to set all the bits to a value by passing the value in as the argument, as in BitSet.SetAll(False).

You can perform bitwise operations on all the bits in a pair of BitArrays using the And, Or, Xor, and Not methods. For example, given that we have two BitArrays, bitSet1 and bitSet2, we can perform a bitwise Or like this:

```
bitSet1.Or(bitSet2)
```

The expression

```
bitSet.Clone()
```

returns a shallow copy of a BitArray, whereas the expression

```
bitSet.CopyTo(arrBits)
```

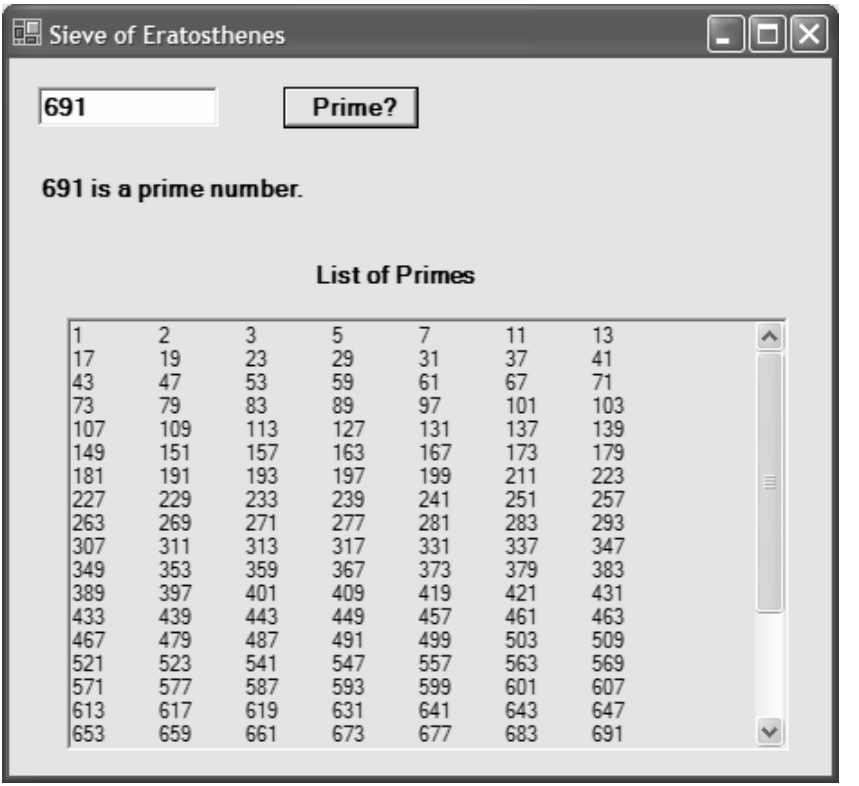
copies the contents of the BitArray to a standard array named arrBits.

With this overview, we are now ready to see how we can use a BitArray to write the sieve of Eratosthenes.

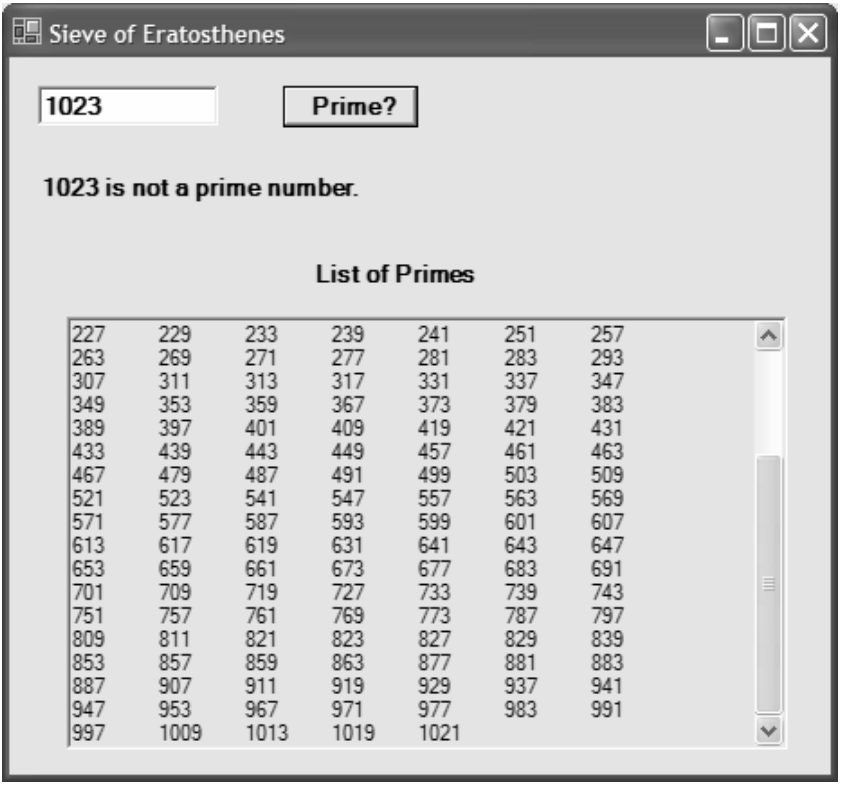
Using a BitArray to Write the Sieve of Eratosthenes

At the beginning of the chapter, we showed you how to write a program to implement the sieve of Eratosthenes using a standard array. In this section we demonstrate the same algorithm, this time using a BitArray to implement the sieve.

The application we've written accepts an integer value from the user, determines the primacy of the number, and also shows a list of the primes from 1 through 1,024. Let's look at a few screen shots of the application:



Here is what happens when the number is not prime:



Now let's look at the code:

```
Public Class Form1
  Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "
  'The VS.NET generated code for the form
   is not displayed

#End Region

  Private Sub Button1_Click(ByVal sender As System. _
      Object, ByVal e As System.EventArgs) Handles _
      btnPrime.Click
    Dim bitSet As New BitArray(1024)
```

```
    Dim value As Integer = Int32.Parse(txtValue.Text)
    BuildSieve(bitSet)
    If (bitSet.Get(value)) Then
      lblPrime.Text =(value &" is a prime number.")
    Else
      lblPrime.Text =(value &" is not a prime number.")
    End If
  End Sub

  Private Sub BuildSieve(ByRef bits As BitArray)
    Dim i, j As Integer
    Dim primes As String
    For i = 0 To bits.Count - 1
      bits.Set(i, 1)
    Next
    Dim lastBit As Integer = CInt(Math.Sqrt(bits.Count))
    For i = 2 To lastBit - 1
      If (bits.Get(i)) Then
        For j = 2 * i To bits.Count - 1 Step i
          bits.Set(j, 0)
        Next
      End If
    Next
    Dim counter As Integer = 0
    For i = 1 To bits.Count - 1
        If (bits.Get(i)) Then
        primes &= CStr(i)
        counter += 1
        If (counter Mod 7 = 0) Then
          primes &= Constants.vbCrLf
        Else
          primes &= Constants.vbTab
        End If
      End If
    Next
    txtPrimes.Text = primes
  End Sub

End Class
```

The sieve is applied in the this loop:

```
Dim lastBit As Integer = CInt(Math.Sqrt(bits.Count))
For i = 2 To lastBit - 1
  If (bits.Get(i)) Then
    For j = 2 * i To bits.Count - 1 Step i
      bits.Set(j, 0)
    Next
  End If
Next
```

The loop works through the multiples of all the numbers up through the square root of the number of items in the BitArray, eliminating all multiples of the numbers 2, 3, 4, 5, and so on.

Once the array is built using the sieve, we can then make a simple call to the BitArray:

```
bitSet.Get(value)
```

If the value is found, then the number is prime. If the value is not found, then it was eliminated by the sieve and the number is not prime.

Comparison of BitArray to Array for Sieve of Eratosthenes

Using a BitArray class is supposed to be more efficient for problems that involve Boolean or bit values. Some problems that don't seem to directly involve these types of values can be redesigned so that a BitArray can be used.

When the Sieve of Eratosthenes method is timed using both a BitArray and a standard array, the BitArray method is consistently faster by a factor of 2. You will get an opportunity to check these results for yourself in the exercises.

Summary

The BitArray class is used to store sets of bits. Although bits are normally represented by 0s and 1s, the BitArray class stores its values as True (1) or False (0) values instead. BitArrays are useful when you need to store a set of

Boolean values, but they are even more useful when you need to work with bits, since we can easily move back and forth between bit values and Boolean values.

Problems that can be solved using arrays of numbers can be more efficiently solved using arrays of bits. Although some readers may see this as a just fancy (or not so fancy) programming trick, the efficiency of storing bit values (or Boolean values) cannot be denied for certain situations.

Exercises

1. Write your own BitArray class (without inheriting from the BitArray class) that includes a conversion method that takes Boolean values and converts them to bit values. (Hint: Use a BitArray as the main data structure of the class but write your own implementation of the other methods.)
2. Reimplement the class in Exercise 1 by inheriting from the BitArray class and adding just a conversion method.
3. Using one of the BitArray classes designed in Exercises 1 and 2, write a method that takes an integer value, reverses its bits, and displays the value in base-10 format.
4. In his excellent book on programming, *Programming Pearls* (Bentley 2000), Jon Bentley discusses the solution to a programming problem that involves using a BitArray, though he calls it a bit vector in his book. Read about the problem at the following Web site: http://www.cs.bell-labs.com/cm/cs/pearls/cto.html and design your own solution to at least the data storage problem using VB.NET. Of course, you don't have to use a file as large as the one used in the book, just pick something that adequately tests your implementation.
5. Write a program that compares the times for both the BitArray implementation of the sieve of Eratosthenes and the standard array implementation. What are your results?

CHAPTER 7

Strings, the String Class, and the StringBuilder Class

Strings are common to most computer programs. Certain types of programs, such as word processors and Web applications, make heavy use of strings, which forces the programmer of such applications to pay special attention to the efficiency of string processing. In this chapter we examine how VB.NET works with strings, how to use the String class, and how to work with the StringBuilder class. The StringBuilder class is used when a program must make many changes to a String object because strings and String objects are immutable, whereas StringBuilder objects are mutable. We'll explain all this later in the chapter.

Working with the String Class

A string is a series of characters that can include letters, numbers, and other symbols. String literals are created in VB.NET by enclosing a series of characters within a set of double quotation marks. Here are some examples of string literals:

```
"David Ruff"
"the quick brown fox jumped over the lazy dog"
```

```
"123-45-6789"
"mmcmillan@pulaskitech.edu"
```

A string can consist of any character that is part of the Unicode character set. A string can also consist of no characters. This special string, called the *empty string*, is shown by placing two double quotation marks next to each other (""). Keep in mind that this is not the string that represents a space. That string looks like this: " ".

Strings in VB.NET have a schizophrenic nature: They are both native types and objects of a class. Actually, to be more precise, we should say that we can work with strings as if they are native data values, but in reality every string created is an object of the String class. We'll explain later why this is so.

Instantiating String Objects

String objects can be instantiated in two ways. First, we can create a String object as if it were a native data type:

```
Dim name As String = "Jennifer Ingram"
```

We can also create a string using more conventional object-oriented syntax:

```
Dim name As New String("David Durr")
```

You cannot call the String class constructor without initializing the object in the constructor call because there is no default constructor method for the String class.

We are not limited to calling the String class constructor with just a string literal. We can also instantiate String objects using other types of data. For example, we can instantiate a string from an array of characters:

```
Dim chars() As Char = {"H"c, "e"c, "l"c, "l"c, "o"c, _
 " "c, ","c, " "c, "w"c, "o"c, _
 "r"c, "l"c, "d"c, "!"c}
Dim greeting As New String(chars)
Console.WriteLine(greeting)
```

This program fragment outputs the following:

```
Hello, world!
```

You can also instantiate a string with just part of a character array. The constructor will take a character array name, a starting position, and a length, and the constructor will build a string that begins at the starting position of the array and continues until the number specified by the length is reached. Here is an example:

```
Dim partGreet As New String(chars, 0, 5)
```

Another option is to instantiate a string using the same character repeated *n* times. For example, if we want to create a string of 20 spaces, we call the constructor like this:

```
Dim spaces As New String(" "c, 20)
```

Finally, you can use the String class copy constructor to instantiate a string using a reference to another String object. Here's an example:

```
Dim st As String = "Goodbye"
Dim parting As New String(st)
```

Frequently Used String Class Methods

Although there are many operations you can perform on strings, a small set of operations dominate. Three of the top operations are 1. finding a substring in a string, 2. determining the length of a string, and 3. determining the position of a character in a string.

The following short program demonstrates how to perform these operations. A String object is instantiated to the string "Hello world". We then break the string into its two constituent pieces: the first word and the second word. Here's the code:

```
Module Module1

  Sub Main()
    Dim string1 As New String("Hello world")
    Dim len As Integer = string1.Length()
    Dim pos As Integer
    pos = string1.IndexOf(" ")
```

```
    Dim firstWord, secondWord As String
    firstWord = string1.Substring(0, pos)
    secondWord = string1.Substring(pos + 1, _
                                   len - (pos + 1))
    Console.WriteLine("First word: " & firstWord)
    Console.WriteLine("Second word: " & secondWord)
    Console.Read()
    End Sub

End Module
```

The first thing we do is use the Length method to determine the length of the object string1. The length is simply the total number of all the characters in the string. We'll explain shortly why we need to know the length of the string.

To break up a two-word phrase into separate words we need to know what separates the words. In a well-formed phrase, a space separates words, so we want to find the space between the two words in this phrase. We can do this with the IndexOf method. This method takes a character and returns the character's position in the string. Strings in VB.NET are zero-based, so the first character in the string is at position 0, the second character is at position 1, and so on. If the character can't be found in the string, a −1 is returned.

The IndexOf method finds the position of the space separating the two words and is used in the next method, Substring, to actually pull the first word out of the string. The Substring method takes two arguments: a starting position and the number of characters to pull. Look at the following example:

```
Dim s As New String("Now is the time")
Dim sub As String = s.Substring(0,3)
```

The value of sub is "Now". The Substring method will pull as many characters out of a string as you ask it to, but if you try to go beyond the end of the string, an exception is thrown.

The first word is pulled out of the string by starting at position 0 and pulling out *pos* number of characters. This may seem odd, since pos holds the position of the space, but since strings are zero-based, this is the correct number.

The next step is to pull out the second word. Since we know where the space is, we know that the second word starts at pos + 1 (again, we're assuming

we're working with a well-formed phrase where each word is separated by exactly one space). The harder part is deciding exactly how many characters to pull out, knowing that an exception will be thrown if we try to go beyond the end of the string. There is a formula of sorts we can use for this calculation. First, we add 1 to the position where the space was found and then subtract that value from the length of the string. That will tell the method exactly how many characters to extract.

Although this short program is interesting, it's not very useful. What we really need is a program that will pull the words out of a well-formed phrase of any length. There are several different algorithms we can use to do this.

The algorithm we'll use here contains the following steps:

1. Find the position of the first space in the string.
2. Extract the word.
3. Build a new string starting at the position past the space and continuing until the end of the string.
4. Look for another space in the new string.
5. If there isn't another space, extract the word from that position to the end of the string.
6. Otherwise, loop back to step 2.

Here is the code we built from this algorithm (with each word extracted from the string being stored in a collection named words):

```
Module Module1

  Sub Main()
    Dim string1 As New String("now is the time")
    Dim pos As Integer
    Dim word As String
    Dim words As New Collection
    pos = string1.IndexOf(" ")
    While (pos > 0)
      word = string1.Substring(0, pos)
      words.Add(word)
      string1 = string1.Substring(pos + 1, string1. _
                                  Length() - (pos + 1))
      pos = string1.IndexOf(" ")
```

```
        If (pos = -1) Then
            word = string1.Substring(0, string1.Length)
            words.Add(word)
        End If
      End While
      Console.Read()
    End Sub

End Module
```

Of course, if we were going to actually use this algorithm in a program we'd make it a function and have it return a collection, like this:

```
Module Module1

    Sub Main()
      Dim string1 As New String("now is the time for " & _
                                "all good people")
      Dim words As New Collection
      words = SplitWords(string1)
      Dim word As Object
      For Each word In words
        Console.WriteLine(word)
      Next
      Console.Read()
    End Sub

    Function SplitWords(ByVal st As String) As Collection
      Dim word As String
      Dim ws As New Collection
      Dim pos As Integer
      pos = st.IndexOf(" ")
      While (pos > 0)
        word = st.Substring(0, pos)
        ws.Add(word)
        st = st.Substring(pos + 1, st.Length - (pos + 1))
        pos = st.IndexOf(" ")
        If (pos = -1) Then
            word = st.Substring(0, st.Length)
            ws.Add(word)
        End If
```

```
    End While
    Return ws
  End Function

End Module
```

However, the String class already has a method for splitting a string into parts (the Split method) as well as a method that can take a data collection and combine its parts into a string (the Join method). We look at those methods in the next section.

The Split and Join Methods

Breaking up strings into individual pieces of data is a very common function. Many programs, from Web applications to everyday office applications, store data in some type of string format. To simplify the process of breaking up strings and putting them back together, the String class provides two methods to use: the Split method for breaking up strings and the Join method for making a string out of the data stored in an array.

The Split method takes a string, breaks it into constituent pieces, and puts those pieces into a String array. The method works by focusing on a separating character to determine where to break up the string. In the example in the previous section, the SplitWords function always used the space as the separator. We can specify what separator to look for when using the Split method.

Many application programs export data by writing out strings of data separated by commas. These are called *comma-separated value* strings. Some authors use the term *comma-delimited*. A comma-delimited string looks like this:

"Mike, McMillan,3000 W. Scenic,North Little Rock,AR,72118"

Each logical piece of data in this string is separated by a comma. We can put each of these logical pieces into an array using the Split method like this:

```
data = "Mike, McMillan,3000 W. Scenic,North Little " & _
       "Rock, AR,72118"
Dim sdata() As String
sdata = data.Split(",")
```

Now we can access these data using standard array techniques:

```
Dim index As Integer
For index = 0 To sdata.GetUpperBound(0)
  Console.WriteLine(sdata(index))
Next
```

There is one more parameter we can pass to the Split method—the number of elements we want to store in the array. For example, if we want to put the first string element in the first position of the array, and the rest of the string in the second element, we would call the method like this:

```
sdata = data.Split(",",2)
```

The elements in the array are as follows:

0th element = Mike

1st element = McMillan,3000 W. Scenic,North Little Rock,AR,72118

We can go the other way, from an array to a string, using the Join method. This method takes two arguments: the original array and a character to separate the elements. A string is built consisting of each array element followed by the separator element. We should also mention that this method is often called as a class method, meaning we call the method from the String class itself and not from a String instance.

Here's an example using the same data we used for the Split method:

```
Module Module1

  Sub Main()
    Dim string1 As New String _
        ("Mike,McMillan,3000 W. Scenic," & _
         "North Little Rock,AR,72118")
    Dim words() As String
    words = string1.Split(",", 2)
    Dim string2 As String
    string2 = String.Join(",", words)
    Console.WriteLine("New string: " & string2)
    Console.Read()
  End Sub

End Module
```

We see that string2 now looks exactly like string1.

These methods are useful for getting data into your program from another source (using the Split method) and sending data out of your program to another source (using the Join method).

Methods for Comparing Strings

There are several ways to compare String objects in VB.NET. The most obvious way is to use the relational operators, which for most situations will work just fine. However, there are situations where other comparison techniques prove to be more useful, such as if we want to know whether a string is greater than, less than, or equal to another string. For these type of situations we have to use methods found in the String class.

Strings are compared with each other much as we compare numbers. However, since it's not obvious whether "a" is greater than or less than "H", we have to have some sort of numeric scale to use. That scale is the Unicode table. Each character (actually every symbol) has a Unicode value, which the operating system uses to convert a character's binary representation to that character. You can determine a character's Unicode value by using the ASC function. ASC actually refers to the ASCII code of a number. ASCII is an older numeric code that precedes Unicode, and the ASC function was first developed before Unicode subsumed ASCII.

The ASC function takes a character as an argument and returns the numeric code for that character. For example, consider the following code:

```
Dim charCode As Integer
charCode = ASC("a")
```

Here the value 97 is stored in the variable.

Two strings are compared, then, by actually comparing their numeric codes. The strings "a" and "b" are not equal because code 97 is not code 98. The CompareTo method actually lets us determine the exact relationship between two String objects. We'll see how to use that method shortly.

The first comparison method we'll examine is the Equals method. This method is called from a String object and takes another String object as its argument. It then compares the two String objects character by character. If they contain the same characters (based on their numeric codes), the method returns True. Otherwise, the method returns False. The method is called like this:

```
Dim s1 As New String("foobar")
Dim s2 As String = "foobar"
If (s1.Equals(s2)) Then
  Console.WriteLine("They are the same.")
Else
  Console.WriteLine("They are not the same.")
End If
```

The next method for comparing strings is CompareTo. This method also takes a string as an argument but it doesn't return a Boolean value. Instead, the method returns 1, −1, or 0, depending on the relationship between the passed-in string and the string instance calling the method. Here are some examples:

```
Dim s1 As New String("foobar")
Dim s2 As String = "foobar"
Console.WriteLine(s1.CompareTo(s2))' Returns 0
s2 = "foofoo"
Console.WriteLine(s1.CompareTo(s2))' Returns -1
s2 = "fooaar"
Console.Writeline(s1.CompareTo(s2))' Returns 1
```

If two strings are equal, the CompareTo method returns a 0; if the passed-in string is "below" the method-calling string, the method returns a −1; if the passed-in string is "above" the method-calling string, the method returns a 1.

An alternative to the CompareTo method is the Compare method, which is usually called as a class method. This method performs the same type of comparison as the CompareTo method and returns the same values for the same comparisons. The Compare method is used like this:

```
Dim s1 As New String("foobar")
Dim s2 As String = "foobar"
Dim compVal As Integer = String.Compare(s1, s2)
Select Case compVal
  Case 0
    Console.WriteLine(s1 & " " & s2 & " are equal")
  Case 1
    Console.WriteLine(s1 & " is less than " & s2)
```

```
    Case 2
      Console.WriteLine(s1 & " is greater than " & s2)
    Case Else
      Console.WriteLine("Can't compare.")
  End Select
```

Two other comparison methods that can be useful when working with strings are StartsWith and EndsWith. These instance methods take a string as an argument and return True if the instance either starts with or ends with the string argument.

Next we present two short programs that demonstrate the use of these methods. First, we'll demonstrate the EndsWith method:

```
Module Module1

  Sub Main()
    Dim nouns() As String = {"cat", "dogs", "bird", _
                             "eggs", "bones"}
    Dim pluralNouns As New Collection
    Dim noun As String
    For Each noun In nouns
      If (noun.EndsWith("s")) Then
          pluralNouns.Add(noun)
      End If
    Next
    For Each noun In pluralNouns
      Console.WriteLine(noun)
    Next
      Console.Read()
  End Sub

End Module
```

First, we create an array of nouns, some of which are in plural form. Then we loop through the elements of the array, checking to see whether any of the nouns are plurals. If a plural is found, it is added to a collection. Then we loop through the collection, displaying each plural.

We use the same basic idea in the next program to determine which words start with the prefix "tri":

```
Module Module1

  Sub Main()
    Dim words() As String = {"triangle", "diagonal", _
                             "trimester", "bifocal", _
                             "triglycerides"}
    Dim triWords As New Collection
    Dim word As String
    For Each word In words
      If (word.StartsWith("tri")) Then
            triWords.Add(word)
      End If
    Next
    For Each word In triWords
      Console.WriteLine(word)
    Next
    Console.Read()
  End Sub

End Module
```

There is one additional comparison technique we need to examine that is not part of the String class: the Like operator. This operator works in much the same way as the regular expression engine (discussed in the next chapter), but without all the flexibility of regular expressions. However, the Like operator is able to discern many simple patterns that come up frequently in string-processing situations.

With this operator, a string is compared to a pattern. If there is a match, the expression returns True; otherwise the expression returns False. The pattern can consist of either a complete string or a string made up of characters and special symbols that are used as wildcards. These wildcards can be used to match any single character, number, or ranges of characters and/or numbers.

The wildcards and range operators used in Like comparisons are the following:

?: matches any single character,

*: matches zero or more characters,

#: matches any single digit,

[char-list]: matches any character in the list, and

[!char-list]: matches any character not in the list.

Here are some examples using the Like operator:

```
Module Module1

  Sub Main()
    Dim s1 As String = "foobar"
    Dim aMatch As String
    aMatch = IIf(s1 Like "foo?ar", "match", "no match")
    Console.WriteLine(aMatch)
    aMatch = IIf(s1 Like "f*", "match", "no match")
    Console.WriteLine(aMatch)
    Dim s2 As New String("H2")
    aMatch = IIf(s2 Like "[hH][0-9]", "match", _
                 "no match")
    Console.WriteLine(aMatch)
    Console.Read()
  End Sub

End Module
```

The output from this program is as follows:

```
match
match
match
```

The first match works because the "?" in the pattern "foo?ar" matches the "b" in "foobar". The second match works because the first letter in the string matches the first letter in the pattern and the "*" in the pattern matches any of the other characters in the string. Be careful when using the asterisk in a pattern because it tends to match even when you don't want it to, leading to it being called a "greedy" operator. (Such "greedy" behavior will be discussed in the next chapter.)

The most interesting match in the program is the last one. This example might be useful when processing a bunch of HTML text in search of heading tags. A heading tag in HTML starts with the letter "H" (or "h"),

followed by a digit indicating the level of the heading. The pattern we used "[hH] [0-9]" leads the Like operator to look for either an "h" or an "H" in the first character and any digit 0 through 9 in the second character. Again, we'll see more examples of this behavior in the next chapter when we discuss regular expressions.

Methods for Manipulating Strings

String processing usually involves making changes to strings. We need to insert new characters into a string, remove characters that don't belong anymore, replace old characters with new characters, change the case of certain characters, and add or remove space from strings, just to name a few operations. There are methods in the String class for all of these operations, and in this section we'll examine them.

We'll start with the Insert method. This method inserts a string into another string at a specified position. Insert returns a new string. The method is called like this:

```
String1 = String0.Insert(Position, String)
```

Let's look at an example:

```
Module Module1

  Sub Main()
    Dim s1 As New String("Hello, . Welcome to my class.")
    Dim name As String = "Clayton"
    Dim pos As Integer = s1.IndexOf(",")
    s1 = s1.Insert(pos + 2, name)
    Console.WriteLine(s1)
    Console.Read()
  End Sub

End Module
```

Here's the output:

```
Hello, Clayton. Welcome to my class.
```

The program creates a string, s1, which deliberately leaves space for a name, much like you'd do with a letter you plan to run through a mail merge. We

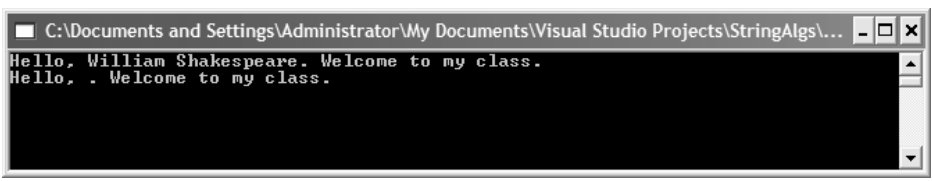
add two to the position where we find the comma to make sure there is a space between the comma and the name.

The next most logical method after Insert is Remove. This method takes two Integer arguments, a starting position and a count, which is the number of characters you want to remove. Here's code that removes a name from a string after the name has been inserted:

```
Module Module1

  Sub Main()
    Dim s1 As New String("Hello, . Welcome to my class.")
    Dim name As String = "Ella"
    Dim pos As Integer = s1.IndexOf(",")
    s1 = s1.Insert(pos + 2, name)
    Console.WriteLine(s1)
    s1 = s1.Remove(pos + 2, name.Length())
    Console.WriteLine(s1)
    Console.Read()
  End Sub

End Module
```

The Remove method uses the same position for inserting a name to remove the name, and the count is calculated by taking the length of the name variable. This allows us to remove any name inserted into the string, as shown by the following code fragment and output screen:

```
Dim name As String = "William Shakespeare"
Dim pos As Integer = s1.IndexOf(",")
s1 = s1.Insert(pos + 2, name)
Console.WriteLine(s1)
s1 = s1.Remove(pos + 2, name.Length())
Console.WriteLine(s1)
```



The next logical method is the Replace method. This method takes two arguments: a string of characters to remove and a string of characters to replace them with. The method returns the new string. Here's how to use Replace:

```
Module Module1

  Sub Main()
    Dim words() As String = {"recieve", "decieve", _
                             "reciept"}
    Dim index As Integer
    For index = 0 To words.GetUpperBound(0)
      words(index) = words(index).Replace("cie", "cei")
      Console.WriteLine(words(index))
    Next
    Console.Read()
  End Sub

End Module
```

The only tricky part of this code involves the way the Replace method is called. Since we're accessing each String object via an array element, we have to use array addressing followed by the method name, causing us to write the following fragment:

```
words(index).Replace("cie", "cei")
```

There is no problem with doing this, of course, because the compiler knows that `words(index)` evaluates to a String object. (We should also mention that Intellisense allows this when writing the code using Visual Studio.NET.)

When displaying data from our programs, we often want to arrange the data within a printing field to line the data up nicely. The String class includes two methods for performing this alignment—PadLeft and PadRight. The PadLeft method right-aligns a string and the PadRight method left-aligns a string. For example, if you want to print the word "Hello" in a 10-character field right-aligned, you would code it like this:

```
Dim s1 As String = "Hello"
Console.WriteLine(s1.PadLeft(10))
Console.WriteLine("world")
```

This would give the output

```
     Hello
world
```

Here's an example using PadRight:

```
Dim s1 As New String("Hello")
Dim s2 As New String("world")
Dim s3 As New String("Goodbye")
Console.Write(s1.PadLeft(10))
Console.WriteLine(s2.PadLeft(10))
Console.Write(s3.PadLeft(10))
Console.WriteLine(s2.PadLeft(10))
```

The output of this is

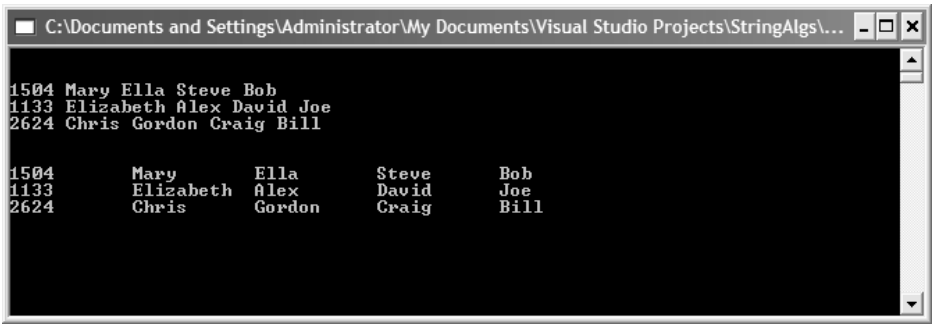
```
  Hello     world
Goodbye     world
```

Here's one more example that demonstrates how we can align data from an array to make the data easier to read:

```
Module Module1

  Sub Main()
    Dim names(,) As String = _
        {{"1504", "Mary", "Ella", "Steve", "Bob"}, _
        {"1133", "Elizabeth", "Alex", "David", "Joe"}, _
        {"2624", "Chris", "Gordon", "Craig", "Bill"}}
    Dim inner, outer As Integer
    Console.WriteLine()
    Console.WriteLine()
    For outer = 0 To names.GetUpperBound(0)
      For inner = 0 To names.GetUpperBound(1)
        Console.Write(names(outer, inner) & " ")
      Next
      Console.WriteLine()
```

```
      Next
      Console.WriteLine()
      Console.WriteLine()
      For outer = 0 To names.GetUpperBound(0)
        For inner = 0 To names.GetUpperBound(1)
          Console.Write(names(outer, inner).PadRight(10) _
                        & " ")
        Next
        Console.WriteLine()
      Next
      Console.Read()
    End Sub

End Module
```

The output from this program looks like this:



The first set of data is displayed without padding and the second set is displayed using the PadRight method.

We already know that the & (ampersand) operator is used for string concatentation. The String class also includes a method Concat for this purpose. This method takes a list of String objects, concatenates them, and returns the resulting string. Here's how to use the method:

```
Module Module1

  Sub Main()
    Dim s1 As New String("hello")
    Dim s2 As New String("world")
```

```
    Dim s3 As New String("")
    s3 = String.Concat(s1, " ", s2)
    Console.WriteLine(s3)
    Console.Read()
  End Sub

End Module
```

We can convert strings from lowercase to uppercase (and vice-versa) using the ToLower and ToUpper methods. The following program fragment demonstrates how these methods work:

```
Dim s1 As New String("hello")
s1 = s1.ToUpper()
Console.WriteLine(s1)
Dim s2 As New String("WORLD")
Console.WriteLine(s2.ToLower())
```

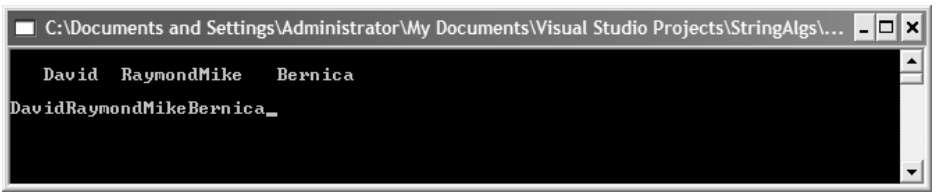
We end this section with a discussion of the Trim and TrimEnd methods. String objects can sometimes have extra spaces or other formatting characters at the beginning or at the end of the string. The Trim and TrimEnd methods will remove spaces or other characters from either end of a string. You can specify either a single character to trim or an array of characters. When you specify an array of characters, if any of the characters in the array are found, they will be trimmed from the string.

Let's first look at an example that trims spaces from the beginning and end of a set of string values:

```
Module Module1

  Sub Main()
    Dim names() As String = _
      {" David", " Raymond", "Mike ", "Bernica "}
    Console.WriteLine()
    showNames(names)
    Console.WriteLine()
    trimVals(names)
    Console.WriteLine()
    showNames(names)
    Console.Read()
  End Sub
```

```
    Sub showNames(ByVal arr() As String)
      Dim index As Integer
      For index = 0 To arr.GetUpperBound(0)
        Console.Write(arr(index))
      Next
    End Sub

    Sub trimVals(ByVal arr() As String)
      Dim index As Integer
      For index = 0 To arr.GetUpperBound(0)
        arr(index) = arr(index).Trim(" "c)
        arr(index) = arr(index).TrimEnd(" "c)
      Next
    End Sub

  End Module
```

Here is the output:

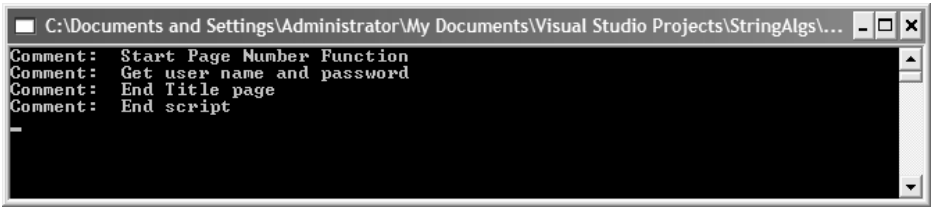


Here's another example, showing how comments from a page of HTML code are stripped of HTML formatting:

```
  Module Module1

    Sub Main()
      Dim htmlComments() As String = _
                {"<!-- Start Page Number Function -->", _
                 "<!-- Get user name and password -->", _
                 "<!-- End Title page -->", _
                 "<!-- End script -->"}
      Dim commentChars() As Char = {"<", "!", "-", ">"}
      Dim index As Integer
      For index = 0 To htmlComments.GetUpperBound(0)
```

```
        htmlComments(index) = htmlComments(index). _
                              Trim(commentChars)
        htmlComments(index) = htmlComments(index). _
                              TrimEnd(commentChars)
      Next
      For index = 0 To htmlComments.GetUpperBound(0)
        Console.WriteLine("Comment: " & _
                          htmlComments (index))
      Next
      Console.Read()
    End Sub

End Module
```

Here's the output:



The StringBuilder Class

The StringBuilder class provides access to mutable String objects. Objects of the String class are immutable, meaning that they cannot be changed. Every time you change the value of a String object, a new object is created to hold the value. StringBuilder objects, in contrast, are mutable. When you make a change to a StringBuilder object, you are changing the original object, not working with a copy. In this section, we discuss how to use the StringBuilder class for those situations where many changes are to be made to the String objects in your programs. We end the section, and the chapter, with a timing test to determine whether working with the StringBuilder class is indeed more efficient than working with the String class.

The StringBuilder class is found in the System.Text namespace, so you must import this namespace into your program before you can use StringBuilder objects.

Constructing StringBuilder Objects

You can construct a StringBuilder object in one of three ways. The first way is to create the object using the default constructor:

```
Dim stBuff1 As New StringBuilder()
```

This line creates the object stBuff1 with the capacity to hold a string 16 characters in length. This capacity is assigned by default, but it can be changed by passing in a new capacity in a constructor call, like this:

```
Dim stBuff2 As New StringBuilder(25)
```

This line builds an object that can initially hold 25 characters.

The final constructor call takes a string as the argument:

```
Dim stBuff3 As New StringBuilder("Hello, world")
```

The capacity is set to 16 since the string argument didn't exceed 16 characters. Had the string argument been longer than 16, the capacity would have been set to 32. Every time the capacity of a StringBuilder object is exceeded, the capacity is increased by 16 characters.

Obtaining and Setting Information about StringBuilder Objects

There are several properties in the StringBuilder class you can use to obtain information about a StringBuilder object. The Length property specifies the number of characters in the current instance and the Capacity property returns the current capacity of the instance. The MaxCapacity property returns the maximum number of characters allowed in the current instance of the object (though this is automatically increased if more characters are added to the object).

The following program fragment demonstrates how to use these properties:

```
Dim stBuff As New StringBuilder("Stephan Jay Gould")
Console.WriteLine("Length of stBuff3: " & _
                  stBuff.Length())
```

```
Console.WriteLine("Capacity of stBuff3: " & _
                  stBuff.Capacity())
Console.WriteLine("Maximum capacity of stBuff3:" & _
                  stBuff.MaxCapacity)
```

The Length property can also be used to set the current length of a StringBuilder object, as in the following code fragment:

```
stBuff.Length = 10
Console.Write(stBuff3)
```

This code outputs

```
Ken Thomps
```

To ensure that a minimum capacity is maintained for a StringBuilder instance, you can call the EnsureCapacity method, passing in an integer that states the minimum capacity for the object. Here's an example:

```
stBuff.EnsureCapacity(25)
```

Another property you can use is the Chars property. This property either returns the character in the position specified in its argument or sets the character passed as an argument. The following code shows a simple example using the Chars property:

```
Dim stBuff As New StringBuilder("Ronald Knuth")
If (stBuff.Chars(0) <> "D"c) Then
    stBuff.Chars(0) = "D"
End If
```

Modifying StringBuffer Objects

We can modify a StringBuilder object by appending new characters to the end of the object, inserting characters into an object, replacing a set of characters in an object with different characters, and removing characters from an object. We discuss the methods responsible for these operations in this section.

You can add characters to the end of a StringBuilder object by using the Append method. This method takes a string value as an argument and

concatenates the string to the end of the current value in the object. The following program demonstrates how the Append method works:

```
Imports System.text
Module Module1

  Sub Main()
    Dim stBuff As New StringBuilder()
    Dim words() As String = _
        {"now ", "is ", "the ", "time ", "for ", "all ", _
         "good ", "men ", "to ", "come ", "to ", "the ", _
         "aid ", "of ", "their ", "party"}
    Dim index As Integer
    For index = 0 To words.GetUpperBound(0)
      stBuff.Append(words(index))
    Next
    Console.WriteLine(stBuff)
    Console.Read()
  End Sub

End Module
```

The output is, of course,

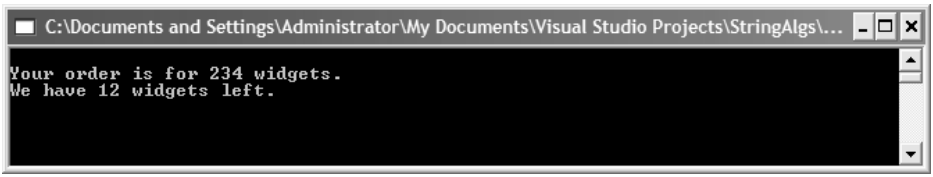
```
Now is the time for all good men to come to the aid of
their party
```

A formatted string can be appended to a StringBuilder object. A formatted string is a string that includes a format specification embedded in the string. There are too many format specifications to cover in this section, so we'll just demonstrate a common specification. We can place a formatted number within a StringBuilder object like this:

```
Imports System.text
Module Module1

  Sub Main()
    Dim stBuff As New StringBuilder
    Console.WriteLine()
    stBuff.AppendFormat _
        ("Your order is for {0000} widgets.", 234)
```

```
    stBuff.AppendFormat(Constants.vbCrLf & _
                        "We have {0000} widgets left.", 12)
    Console.WriteLine(stBuff)
    Console.Read()
  End Sub

End Module
```

The output from this program looks like this:



The format specification is enclosed within curly braces that are embedded in a string literal. The data after the comma are placed into the specification when the code is executed. See the VB.NET documentation for a complete list of format specifications.

Next we consider the Insert method. This method allows us to insert a string into the current StringBuilder object. The method can take up to three arguments. The first argument specifies the position to begin the insertion. The second argument is the string you want to insert. The third argument, which is optional, is an integer that specifies the number of times you want to insert the string into the object.

Here's a small program that demonstrates how to use the Insert method:

```
Imports System.text
Module Module1

  Sub Main()
    Dim stBuff As New StringBuilder
    stBuff.Insert(0, "Hello")
    stBuff.Append("world")
    stBuff.Insert(5, ", ")
    Console.WriteLine(stBuff)
    Dim chars() As Char = {"t"c, "h"c, "e"c, "r"c, "e"}
    stBuff.Insert(5, " " & chars)
    Console.WriteLine(stBuff)
```

```
    Console.Read()
  End Sub

End Module
```

The output is

```
Hello, world
Hello there, world
```

The following program utilizes the Insert method using the third argument for specifying the number of insertions to make:

```
Dim stBuff As New StringBuilder
stBuff.Insert(0, "and on ", 6)
Console.WriteLine(stBuff)
```

Here's the output:

```
and on and on and on and on and on and on
```

The StringBuilder class has a Remove method for removing characters from a StringBuilder object. This method takes two arguments: a starting position and the number of characters to remove. Here's how it works:

```
Dim stBuff As New StringBuilder("noise in +++++string")
stBuff.Remove(9, 5)
Console.WriteLine(stBuff)
```

The program outputs

```
noise in string
```

You can replace characters in a StringBuilder object with the Replace method. This method takes two arguments: the old string to replace and the new string to put in its place. The following code fragment demonstrates how the method works:

```
Dim stBuff As New StringBuilder("recieve decieve reciept")
stBuff.Replace("cie", "cei")
Console.WriteLine(stBuff)
```

Each "cie" is replaced with "cei".

When working with StringBuilder objects, you will often want to convert them to strings, perhaps to use a method that isn't found in the StringBuilder class. You can do this with the ToString method. This methods returns a String instance of the current StringBuilder instance. Here's an example:

```
Imports System.text
Module Module1

  Sub Main()
    Dim stBuff As New StringBuilder("HELLO WORLD")
    Dim st As String = stBuff.ToString()
    st = st.ToLower()
    st = st.Replace(st.Substring(0, 1), _
                    st.Substring(0, 1). ToUpper())
    stBuff.Replace(stBuff.ToString, st)
    Console.WriteLine(stBuff)
    Console.Read()
  End Sub

End Module
```

This program displays the string, "Hello world" by first converting stBuff to a string (the st variable), making all the characters in the string lowercase, capitalizing the first letter in the string, and then replacing the old string in the StringBuilder object with the value of st. The ToString method is used in the first argument of Replace because the first parameter is supposed to be a string. You can't call the StringBuilder object directly here.

Comparing the Efficiency of the String and StringBuilder Classes

We end this chapter with a discussion of how the String class and the StringBuilder class compare in efficiency. We know that String objects are immutable and StringBuilder objects are not. It is reasonable to believe, then, that the StringBuilder class is more efficient. However, we don't want to always use the StringBuilder class because the StringBuilder class lacks several methods we need to perform reasonably powerful string processing. It is true that we can transform StringBuilder objects into String objects (and then back again)

when we need to use String methods (see the previous section), but we must know when we need to use StringBuilder objects and when we can get by with just String objects.

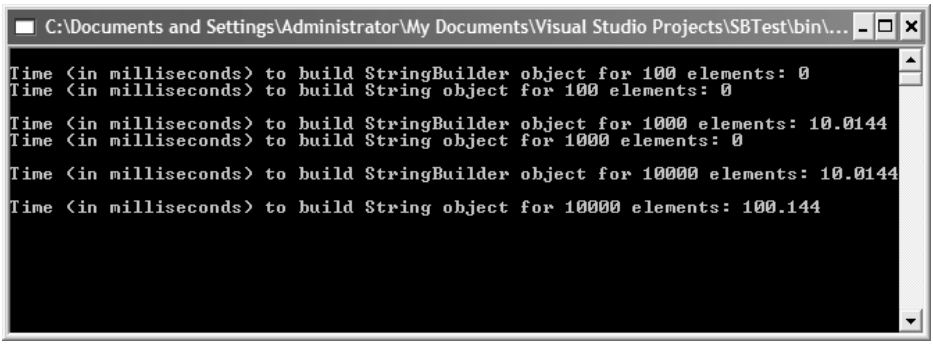
The test we use is very simple. Our program has two subroutines—one that builds a String object of a specified size and another that builds a StringBuilder object of the same size. Each of the subroutines is timed, using objects from the Timing class we developed at the beginning of the book. This procedure is repeated three times, first for building objects of 100 characters, then for 1,000 characters, and finally for 10,000 characters. The times are then listed in pairs for each size.

Here's the code we used:

```
Option Strict On
Imports Timing
Imports System.text
Module Module1

  Sub Main()
    Dim size As Integer = 100
    Dim timeSB As New Timing
    Dim timeST As New Timing
    Dim index As Integer
    Console.WriteLine()
    For index = 1 To 3
      timeSB.startTime()
      BuildSB(size)
      timeSB.stopTime()
      timeST.startTime()
      BuildString(size)
      timeST.stopTime()
      Console.WriteLine _
        ("Time (in milliseconds) to build StringBuilder " & _
         "object for " & size & " elements: " & _
         timeSB. Result.TotalMilliseconds)
      Console.WriteLine _
        ("Time (in milliseconds) to build String object " & _
         "for " & size & " elements: " & timeST.Result. _
         TotalMilliseconds)
```

```
        Console.WriteLine()
        size *= 10
      Next
      Console.Read()
    End Sub

    Sub BuildSB(ByVal size As Integer)
      Dim sbObject As New StringBuilder
      Dim index As Integer
      For index = 1 To size
        sbObject.Append("a")
      Next
    End Sub

    Sub BuildString(ByVal size As Integer)
      Dim stringObject As String = ""
      Dim index As Integer
      For index = 1 To size
        stringObject &= "a"
      Next
    End Sub

End Module
```

Here are the results:



For relatively small objects, there is really no difference between String objects and StringBuilder objects. In fact, you can argue that for strings of up to 1,000 characters, using the String class is just as efficient as using the StringBuilder class. However, when we get to 10,000 characters, the tremendous

increase in efficiency for the StringBuilder class becomes evident. There is, though, a vast difference between 1,000 characters and 10,000 characters. In the exercises, you'll get the opportunity to compare objects that hold more than 1,000 but less than 10,000 characters.

Summary

String processing is a common operation in most VB.NET programs. The String class provides a multitude of methods for performing every kind of operation on strings you will need. Although the "classic" built-in string functions (Mid, InStr, etc.) are still available for use, you should prefer the String class methods to these functions, both for performance and for clarity.

String class objects in VB.NET are immutable, meaning that every time you make a change to an object, a new copy of the object is created. If you are creating long strings, or are making many changes to the same object, you should use the StringBuffer class instead. StringBuffer objects are mutable, allowing for much better performance, as demonstrated in timing tests when String objects and StringBuilder objects of over 1,000 characters in length are created.

Exercises

1. Write a function that converts a phrase into pig Latin. A word is converted to pig Latin by removing the first character of the word, placing it at the back of the word, and adding the characters "ay" to the word. For example, "hello world" in pig Latin is "ellohay orldway." Your function can assume that each word consists of at least two letters and that each word is separated by one space, with no punctuation marks.
2. Write a function that counts the occurrences of a word in a string. The function should return an integer. Do not assume that just one space separates words and assume that a string can contain punctuation. Write the function so that it works with either a String argument or a StringBuilder object.
3. Write a function that takes a number, such as 52, and returns the number as a word, as in fifty-two.

4. Write a subroutine that takes a simple sentence in noun-verb-object form and parses the sentence into its different parts. For example, the sentence "Mary walked the dog" is parsed into this:

   Noun: Mary

   Verb: walked

   Object: the dog

   This function should work with both String objects and StringBuilder objects.

CHAPTER 8

Pattern Matching and Text Processing

Whereas the String and StringBuilder classes provide a set of methods that can be used to process string-based data, the RegEx and its supporting classes provide much more power for string-processing tasks. String processing mostly involves looking for patterns in strings (pattern matching) and it is performed via a special language called a regular expression. In this chapter we look at how to form regular expressions and how to use them to solve common text-processing tasks.

Regular Expressions

A regular expression is a language that describes patterns of characters in strings, along with descriptors for repeating characters, alternatives, and groupings of characters. Regular expressions can be used to both perform searches in strings and perform substitutions in strings.

A regular expression itself consists of just a string of characters that define a pattern you want to search for in another string. Generally, the characters in a regular expression match themselves, so that the regular expression "the" matches that sequence of characters wherever they are found in a string.

A regular expression can also include special characters called *metacharacters*. Metacharacters are used to signify repetition, alternation, or grouping. We will examine how these metacharacters are used later in the chapter.

Most experienced computer users have used regular expressions in their work, even if they weren't aware they were doing so at the time. Whenever you type the following command at a command prompt:

```
C:\>dir myfile.exe
```

you are using the regular expression "myfile.exe". The regular expression is passed to the dir command and any files in the file system matching "myfile.exe" are displayed on the screen.

Most users have also used metacharacters in regular expressions. When you type

```
C:\>dir *.vb
```

you are using a regular expression that includes a metacharacter. The regular expression is "*.vb". The asterisk (*) is a metacharacter that means "match zero or more characters"; the rest of the expression (".vb") gives just normal characters found in a file. This regular expression states "match all files that have any file name and the extension 'vb'." This regular expression is passed to the dir command and all files with a.vb extension are displayed on the screen.

Of course, there are much more powerful regular expressions we can build and use, but these first two examples serve as a good introduction. Now let's look at how we use regular expressions in VB.Net and how to construct useful regular expressions.

Working with Regular Expressions

To use regular expressions, we have to import the RegEx class into our programs. This class is found in the System.Text.RegularExpressions namespace.

Once we have the class imported into our program, we have to decide what we want to do with the RegEx class. If we want to perform matching, we need to use the Match class. If we're going to make substitutions, we don't need the Match class. Instead, we can use the Replace method of the RegEx class.

Let's start by looking at how to match words in a string. Given a sample string, "the quick brown fox jumped over the lazy dog," we want to find out where the word "the" is found in the string. The following program performs this task:

```
Module Module1

  Sub main()
    Dim reg As New Regex("the")
    Dim str1 As String = _
      "the quick brown fox jumped over the lazy dog"
    Dim matchSet As Match
    Dim matchPos As Integer
    matchSet = reg.Match(str1)
    If (matchSet.Success) Then
      matchPos = matchSet.Index()
      Console.WriteLine("found match at position: " & _
                        matchPos)
    End If
    Console.Read()
  End Sub

End Module
```

The first thing we do is create a new RegEx object and pass the constructor the regular expression we're trying to match. After we initialize a string to match against, we declare a Match object, matchSet. The Match class provides methods for storing data concerning a match made with the regular expression.

The If statement uses one of the Match class properties, Success, to determine whether there was a successful match. If the value returns True, then the regular expression matched at least one substring in the string. Otherwise, the value stored in Success is False.

There's another way a program can check to see whether a match has been successful. You can pretest the regular expression by passing it and the target string to the IsMatch method. This method returns True if a match is generated by the regular expression and returns False otherwise. The method works like this:

```
If (Regex.IsMatch(str1, "the")) Then
  Dim aMatch As Match
```

```
    Dim pos As Integer
    aMatch = reg.Match(str1)
  End If
```

Unfortunately, the Match class can only store one match. In our example, there are two matches for the substring "the". We can use another class, the Matches class, to store multiple matches with a regular expression. We can store the matches in a MatchCollection object to facilitate working with all the matches found. Here's an example:

```
Module Module1

  Sub main()
    Dim reg As New Regex("the")
    Dim str1 As String = _
      "the quick brown fox jumped over the lazy dog"
    Dim matchSet As MatchCollection
    Dim matchPos As Integer
    matchSet = reg.Matches(str1)
    If (matchSet.Count > 0) Then
      Dim aMatch As Match
      For Each aMatch In matchSet
        Console.WriteLine("found a match at: " & _
                          aMatch.Index)
      Next
    End If
    Console.Read()
  End Sub

End Module
```

Next, we examine how to use the Replace method to replace one string with another string. The Replace method can be called as a class method with three arguments: a target string, a substring to replace, and the substring to use as the replacement. Here's a code fragment that uses the Replace method:

```
Dim s As String = _
    "the quick brown fox jumped over the brown dog"
s = Regex.Replace(s, "brown", "black")
```

The string now reads, "the quick black fox jumped over the black dog".

There are many more uses of the RegEx and supporting classes for pattern matching and text processing. We will examine them as we delve deeper into how to form and use more complex regular expressions.

QUANTIFIERS

When writing regular expressions, we often want to add quantity data to a regular expression, such as "match exactly twice" or "match one or more times." We can add these data to our regular expressions using quantifiers.

The first quantifier we'll look at is the plus sign (+). This quantifier indicates that the regular expression should match one or more of the immediately preceding characters. The following program demonstrates how to use this quantifier:

```
Sub main()
  Dim words() As String = _
        {"bad", "boy", "baaad", "bear", "bend"}
  Dim word As String
  Dim aMatch As Match
  For Each word In words
    If (Regex.IsMatch(word, "ba+")) Then
        Console.WriteLine(word)
    End If
  Next
  Console.Read()
End Sub
```

The words matched are "bad" and "baaad". The regular expression specifies that a match is generated for each string that starts with the letter "b" and includes one or more of the letter "a" in the string.

A less restrictive quantifier is the asterisk (*). This quantifier indicates that the regular expression should match zero or more of the immediately preceding characters. This quantifier is very hard to use in practice because the asterisk usually ends up matching almost everything. For example, using the preceding code, if we change the regular expression to read "ba*", every word in the array is matched.

The question mark (?) is a quantifier that matches exactly once or zero times. If we change the regular expression in the preceding code to "ba?d", the only word that matches is "bad".

A more definite number of matches can be specified by placing a number inside a set of curly braces, as in {*n*}, where *n* is the number of matches to find. The following program demonstrates how this quantifier works:

```
Sub main()
  Dim words() As String = _
    {"bad", "boy", "baad", "baaad", "bear", "bend"}
  Dim word As String
  Dim aMatch As Match
  For Each word In words
    If (Regex.IsMatch(word, "ba{2}d")) Then
        Console.WriteLine(word)
    End If
  Next
  Console.Read()
End Sub
```

This regular expression matches only the string "baad".

You can specify a minimum and a maximum number of matches by providing two digits inside the curly braces: {*n*,*m*}, where *n* is the minimum number of matches and *m* is the maximum. The following regular expression will match "bad", "baad", and "baaad" in our sample string:

```
"ba{1,3}d"
```

We could have also matched the same number of strings here by writing `"ba{1,}d"`, which specifies at least one match, but without specifying a maximum number.

The quantifiers we've discussed so far exhibit what is called *greedy* behavior. They try to make as many matches as possible, and this behavior often leads to matches that you didn't really mean to make. Here's an example:

```
Sub main()
  Dim words() As String = {"Part", "of", "this", _
                           "<b>string</b>", "is", "bold"}
  Dim word As String
  Dim regexp As String = "<.*>"
```

```
    Dim aMatch As MatchCollection
    For Each word In words
      If (Regex.IsMatch(word, regexp)) Then
          aMatch = Regex.Matches(word, regexp)
          Dim i As Integer
          For i = 0 To aMatch.Count - 1
            Console.WriteLine(aMatch(i).Value)
          Next
          Console.WriteLine()
      End If
    Next
    Console.Read()
End Sub
```

We expect this program to return just the two tags: and . Instead, because of greediness, the regular expression matches string. We can solve this problem using the lazy quantifier: the question mark (?). When the question mark is placed directly after a quantifier, it makes the quantifier lazy. Being lazy means the regular expression the lazy quantifier is used in will try to make as few matches as possible, instead of as many as possible.

Changing the regular expression to read "< .+ >" doesn't help either. We need to use the lazy quantifier, and once we do ("< .+?>"), we get the right matches: and .

The lazy quantifier can be used with all the quantifiers, including the quantifiers enclosed in curly braces.

Using Character Classes

In this and the following sections, we examine how to use the major elements that make up regular expressions. We start with character classes, which allow us to specify a pattern based on a series of characters.

The first character class we discuss is the period (.). This is a very easy character class to use but it is also very problematic. The period matches any character in a string. Here's an example:

```
Sub main()
  Dim str1 As String = _
    "the quick brown fox jumped over the lazy dog"
```

```
    Dim matchSet As MatchCollection
    matchSet = Regex.Matches(str1, ".")
    Dim aMatch As Match
    For Each aMatch In matchSet
      Console.WriteLine("matches at: " & aMatch.Index)
    Next
    Console.Read()
  End Sub
```

The output from this program illustrates how the period works:

The period matches every single character in the string!

A better way to use the period is to use it to define a range of characters within a string that are bound by a beginning and/or an ending character. Here's one example, using the same string:

```
Sub main()
  Dim str1 As String = _
    "the quick brown fox jumped over the lazy dog " & _
    "one time"
  Dim matchSet As MatchCollection
  matchSet = Regex.Matches(str1, "t.e")
  Dim aMatch As Match
  For Each aMatch In matchSet
```

```
      Console.WriteLine("matches at: " & aMatch.Index)
    Next
    Console.Read()
  End Sub
```

The output from this program is

```
matches: the at: 0
matches: the at: 32
```

When using regular expressions, we often want to check for patterns that include groups of characters. We can write a regular expression that consists of such a group by enclosing the group in square brackets ([]). The characters inside the brackets are called a *character class*. If we wanted to write a regular expression that matched any lowercase alphabetic character, we would write the expression like the following: [abcdefghijklmnopqrstuvwxyz].

But that's fairly hard to write, so we can write a shorter version by indicating a range of letters using a hyphen: [a–z].

Here's how we can use this regular expression to match a pattern:

```
Sub main()
  Dim str1 As String = _
    "THE quick BROWN fox JUMPED over THE lazy DOG"
  Dim matchSet As MatchCollection
  matchSet = Regex.Matches(str1, "[a-z]")
  Dim aMatch As Match
  For Each aMatch In matchSet
    Console.WriteLine("matches: " & aMatch.Value)
  Next
  Console.Read()
End Sub
```

The letters matched are those that make up the words "quick," "fox," "over," and "lazy."

Character classes can be formed using more than one groups. If we want to match both lowercase letters and uppercase letters, we can write the following regular expression: "[A-Za-z]". You can also write a character class consisting of digits. For example, if you want to include all 10 digits you would use [0-9].

We can create the reverse, or negation, of a character class by placing a caret (^) before the character class. For example, if we have the character class [aeiou] representing the class of vowels, we can write [^aeiou] to represent the consonants, or nonvowels.

If we combine these three character classes, we form what is called a *word* in regular expression parlance. The regular expression would be [A-Za-z0-9]. There is also a shorter character class we can use to express this same class: \w. The negation of \w, or the regular expression to express a nonword character (such as a punctuation mark) is expressed by \W.

The character class for digits ([0-9]) can also be written as \d, and the character class for nondigits ([^0-9]) can be written as \D. Finally, because white space plays such an important role in text processing, \s is used to represent white-space characters whereas \S represents non-white-space characters. We will examine using the white-space character classes later when we examine the grouping constructs.

Modifying Regular Expressions Using Assertions

VB.NET includes a set of assertions you can add to a regular expression that change the behavior of the expression without causing the regular expression engine to advance through the string. These are called *assertions*.

The first assertion we'll examine causes a regular expression to find matches only at the beginning of a string or a line. This assertion is made using the caret symbol (^). In the following program, the regular expression matches strings that have the letter "h" only as the first character in the string. An "h" in other places is ignored. Here's the code:

```
Sub main()
  Dim words() As String = {"heal", "heel", "noah", _
                           "techno"}
  Dim regexp As String = "^h"
  Dim aMatch As Match
  Dim word As String
  For Each word In words
    If (Regex.IsMatch(word, regexp)) Then
        aMatch = Regex.Match(word, regexp)
        Console.WriteLine("Matched: " & word & _
                          " at position: " & _
                          aMatch.Index)
    End If
```

```
    Next
    Console.Read()
End Sub
```

The output of this code shows that just the strings "heal" and "heel" match.

There is also an assertion that causes a regular expression to find matches only at the end of the line. This assertion is the dollar sign ($). Modifying the previous regular expression to

```
Dim regexp As String = "h$"
```

yields "noah" as the only match found.

Another assertion you can make in a regular expression is to specify that all matches can occur only at word boundaries. This means that a match can only occur at the beginning or end of a word that is separated by spaces. This assertion is made with \b. Here's how the assertion works:

```
Dim words As String = "hark, what doth thou say, " & _
                      "Harold? "
Dim regexp As String = "\bh"
```

This regular expression matches the words "hark" and "Harold" in the string.

There are other assertions you can use in regular expressions, but these are three of the most commonly used.

Using Grouping Constructs

The RegEx class has a set of grouping constructs you can use to put successful matches into groups, which makes it easier to parse a string into related matches. For example, if you are given a string of birthday dates and ages and you want to identify just the dates, by grouping the dates together, you can identify them as a group and not just as individual matches.

Anonymous Groups

There are several different grouping constructs you can use. The first construct is formed by surrounding the regular expression in parentheses. You can think of this as an anonymous group, since groups can also be named, as we'll see

in a moment. As an example, look at the following string:

```
"08/14/57 46 02/25/29 45 06/05/85 18 03/12/88 16
 09/09/90 13"
```

This string is a combination of birthdates and ages. If we want to match just the ages, not the birthdates, we can write the regular expression as an anonymous group:

```
(\s\d{2}\s)
```

By writing the regular expression this way, each match in the string is identified by a number, starting at one. Number zero is reserved for the entire match, which will usually include much more data. Here is a little program that uses an anonymous group:

```
Sub main()
  Dim words As String = "08/14/57 46 02/25/59 45 " & _
                        "06/05/85 18 03/12/88 16 " & _
                        "09/09/90 13"
  Dim regexp1 As String = "(\s\d{2}\s)"
  Dim matchSet As MatchCollection
  Dim aMatch As Match
  matchSet = Regex.Matches(words, regexp1)
  For Each aMatch In matchSet
    Console.WriteLine(aMatch.Groups(0).Captures(0))
  Next
  Console.Read()
End Sub
```

Named Groups

Groups are more commonly built using names. A named group is easier to work with because we can refer to the group by name when retrieving matches. A named group is formed by prefixing the regular expression with a question mark and a name enclosed in angle brackets. For example, to name the group in our example "ages," we write the regular expression like this:

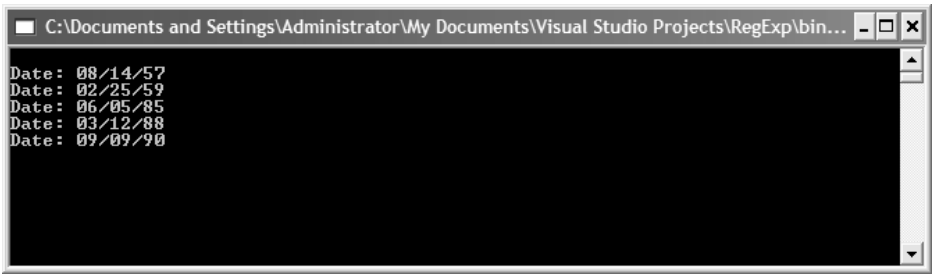
```
(?<ages>\s\d{2}\s)
```

The name can also be surrounded by single quotes instead of angle brackets.

Now let's modify the previous program to search for dates instead of ages, and use a grouping construct to organize the dates. Here's the code:

```
Sub main()
  Dim dates As String = _
    "08/14/57 46 02/25/59 45 06/05/85 18 " & _
    "03/12/88 16 09/09/90 13"
  Dim regexp As String =
"(?<dates>(\d{2}/\d{2}/\d{2}))\s"
  Dim matchSet As MatchCollection
  Dim aMatch As Match
  matchSet = Regex.Matches(dates, regexp)
  Console.WriteLine()
  For Each aMatch In matchSet
    Console.WriteLine("Date: {0}", aMatch.Groups("dates"))
  Next
  Console.Read()
End Sub
```

Here's the output:



Let's focus on the regular expression used to generate the output:

```
(\d{2}/\d{2}/\d{2}))\s
```

You can read this expression as "two digits followed by a slash, followed by two more digits and a slash, followed by two more digits and a slash, followed

by a space." To make the regular expression a group, we make the following additions:

```
(?<dates>(\d{2}/\d{2}/\d{2}))\s
```

For each match found in the string, we pull out the group by using the Groups method of the Match class:

```
Console.WriteLine("Date: {0}", aMatch.Groups("dates"))
```

Zero-Width Lookahead and Lookbehind Assertions

Assertions can also be made that determine how far into a match a regular expression will look for matches, going either forward or backward. These assertions can be either positive or negative, meaning that the regular expression is looking for either a particular pattern to match (positive) or for a particular pattern not to match (negative). This explanation will be clearer when we see some examples.

The first of these assertions we examine is the positive lookahead assertion. This assertion is stated like this:

```
(?= reg-exp-char)
```

where `reg-exp-char` is a regular expression character or metacharacter. This assertion states that a match continued only if the current subexpression being checked matches at the specified position on the right. Here's a code fragment that demonstrates how this assertion works:

```
Dim words As String = _
    "lions lion tigers tiger bears,bear"
Dim regexp2 As String = "\w+(?=\s)"
```

The regular expression indicates that a match is made on each word that is followed by a space. The words that match are "lions", "lion", "tigers", and "tiger". The regular expression matches the words but does not match the space. This aspect is very important to remember.

The next assertion is the negative lookahead assertion. This assertion continues a match only if the current subexpression being checked does not match at the specified position on the right. Here's an example code fragment:

```
Dim words As String = _
    "subroutine routine subprocedure procedure"
Dim regexp2 As String = "\b(?!sub)\w+\b"
```

This regular expression indicates that a match is made on each word that does not begin with the prefix "sub". The words that match are "routine" and "procedure".

The next assertions are called lookbehind assertions. These assertions look for positive or negative matches to the left instead of to the right. The following code fragment demonstrates how to write a positive lookbehind assertion:

```
Dim words As String = _
    "subroutines routine subprocedures procedure"
Dim regexp1 As String = "\b\w+(?<=s)\b"
```

This regular expression looks for word boundaries that occur after an "s". The words that match are "subroutines" and "subprocedures".

A negative lookbehind assertion continues a match only if the subexpression does not match at the position on the left. We can easily modify the previous regular expression to match only words that don't end with the letter "s" like this:

```
Dim regexp1 As String = "\b\w+(?<!s)\b"
```

The CapturesCollection Class

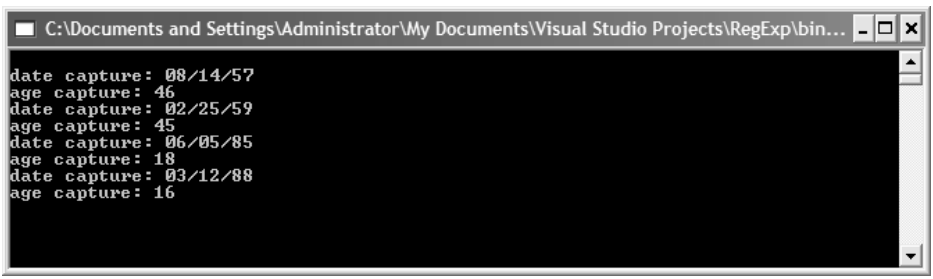
When a regular expression matches a subexpression, an object called a Capture is created and is added to a collection called a CapturesCollection. When you use a named group in a regular expression, that group has its own collection of captures.

To retrieve the captures collected from a regular expression that uses a named group, you call the Captures property from a Match objects Groups property. This is easier to see in an example. Using one of the regular

expressions from the previous section, we construct the following code to return all the dates and ages found in a string, properly grouped:

```
Sub main()
 Dim dates As String = _
   "08/14/57 46 02/25/59 45 06/05/85 18 " & _
   "03/12/88 16 09/09/90 13"
 Dim regexp As String = _
   "(?<dates>(\d{2}/\d{2}/\d{2}))\s" & _
   "(?<ages>(\d{2}))\s"
 Dim matchSet As MatchCollection
 Dim aMatch As Match
 Dim aCapture As Capture
 matchSet = Regex.Matches(dates, regexp)
 Console.WriteLine()
 For Each aMatch In matchSet
   For Each aCapture In aMatch.Groups("dates").Captures
     Console.WriteLine("date capture: " & _
                       aCapture.ToString)
   Next
   For Each aCapture In aMatch.Groups("ages").Captures
     Console.WriteLine("age capture: " & _
                       aCapture.ToString)
   Next
 Next
 Console.Read()
End Sub
```

The output from this program looks like this:



The outer loop moves through each match; the two inner loops move through the different Capture collections, one for the dates and one for the ages. Using the CapturesCollection in this way ensures that each group match gets captured, not just the last match.

Regular Expression Options

There are several options you can set when specifying a regular expression. These options range from specifying the multiline mode so that a regular expression will work properly on more than one line of text to compiling a regular expression so that it will execute faster. The table that follows lists the different options you can set.

Before we view the table, we need to mention how these options are set. Generally, you can set an option by specifying the options constant value as the third argument to one of the RegEx class's methods, such as Match as Matches. For example, if we want to set the Multiline option for a regular expression, the line of code looks like this:

```
matchSet = Regex.Matches(dates, regexp, _
                         RegexOptions.Multiline)
```

This option, along with the other options, can either be typed in directly or selected with Intellisense.

Here are the options available:

RegexOption member	Inline character	Description
None	N/A	Specifies that no options are set.
IgnoreCase	I	Specifies case-insensitive matching.
Multiline	M	Specifies multiline mode.
ExplicitCapture	N	Specifies that the only valid captures are explicitly named or numbered groups.

RegexOption member	Inline character	Description
Compiled	N/A	Specifies that the regular expression will be compiled to assembly.
Singleline	S	Specifies single-line mode.
IgnorePatternWhiteSpace	X	Specifies that unescaped white space is excluded from the pattern and enables comments following a pound sign (#).
RightToLeft	N/A	Specifies that the search is from right to left instead of from left to right.
ECMAScript	N/A	Specifies that ECMAScript-compliant behavior is enabled for the expression.

Summary

Regular expressions present powerful options for performing text processing and pattern matching. Regular expressions can run the gamut from ridiculously simple ("a") to complex combinations that look more like line noise than executable code. Nonetheless, learning to use regular expressions will allow you to perform text processing on texts for which you would never consider using tools such as the methods of the String class or the traditional built-in Visual Basic string functions.

This chapter can only hint at the power of regular expressions. To learn more about regular expressions, consult Friedl's book *Mastering Regular Expressions* (Friedl 1997).

Exercises

1. Write regular expressions to match the following:
   - a string consisting of an "x", followed by any three characters, and then a "y"
   - a word ending in "ed"

- a phone number
- an HTML anchor tag

2. Write a regular expression that finds all the words in a string that contain double letters, such as “deep” and “book”.
3. Write a regular expression that finds all the header tags (<h1>, <h2>, etc.) in a Web page.
4. Write a function, using a regular expression, that performs a simple search and replace in a string.

CHAPTER 9

Building Dictionaries: The DictionaryBase Class and the SortedList Class

A *dictionary* is a data structure that stores data as a *key–value* pair. The DictionaryBase class is used as an abstract class to implement different data structures that all store data as key–value pairs. These data structures can be hash tables, linked lists, or some other data structure type. In this chapter, we examine how to create basic dictionaries and how to use the inherited methods of the DictionaryBase class. We will use these techniques later when we explore more specialized data structures.

One example of a dictionary-based data structure is the SortedList. This class stores key–value pairs in sorted order based on the key. It is an interesting data structure because you can also access the values stored in the structure by referring to the value's index position in the data structure, which makes the structure behave somewhat like an array. We examine the behavior of the SortedList class at the end of the chapter.

The DictionaryBase Class

You can think of a dictionary data structure as a computerized word dictionary. The word you are looking up is the key, and the definition of the word is the value. The DictionaryBase class is an abstract (MustInherit) class that is used as a basis for specialized dictionary implementations.

The key–value pairs stored in a dictionary are actually stored as DictionaryEntry objects. The DictionaryEntry structure provides two fields, one for the key and one for the value. The only two properties (or methods) we're interested in with this structure are the Key and Value properties. These methods return the values stored when a key–value pair is entered into a dictionary. We explore DictionaryEntry objects later in the chapter.

Internally, key–value pairs are stored in a hash table object called InnerHashTable. We discuss hash tables in more detail in Chapter 12, so for now just view it as an efficient data structure for storing key–value pairs.

The DictionaryBase class actually implements an interface from the System.Collections namespace, IDictionary. This interface actually forms the basis for many of the classes we'll study later in this book, including the ListDictionary class and the Hashtable class.

Fundamental DictionaryBase Class Methods and Properties

When working with a dictionary object, there are several operations you want to perform. At a minimum, you need an Add method to add new data, an Item method to retrieve a value, a Remove method to remove a key–value pair, and a Clear method to clear the data structure of all data.

Let's begin the discussion of implementing a dictionary by looking at a simple example class. The following code shows the implementation of a class that stores names and IP addresses:

```
Public Class IPAddresses
  Inherits DictionaryBase

  Public Sub New()
    MyBase.new()
  End Sub
  Public Sub Add(ByVal name As String, ByVal ip _
                 As String)
```

```
    MyBase.InnerHashtable.Add(name, ip)
  End Sub

  Public Function Item(ByVal name As String) As String
    Return CStr(MyBase.InnerHashtable.Item(name))
  End Function

  Public Sub Remove(ByVal name As String)
    MyBase.InnerHashtable.Remove(name)
  End Sub

End Class
```

As you can see, these methods were very easy to build. The first method implemented is the constructor. This is a simple method that does nothing but call the default constructor for the base class. The Add method takes a name–IP address pair as arguments and passes them to the Add method of the InnerHashTable object, which is instantiated in the base class.

The Item method is used to retrieve a value given a specific key. The key is passed to the corresponding Item method of the InnerHashTable object. The value stored with the associated key in the inner hash table is returned.

Finally, the Remove method receives a key as an argument and passes the argument to the associated Remove method of the inner hash table. The method then removes both the key and its associated value from the hash table.

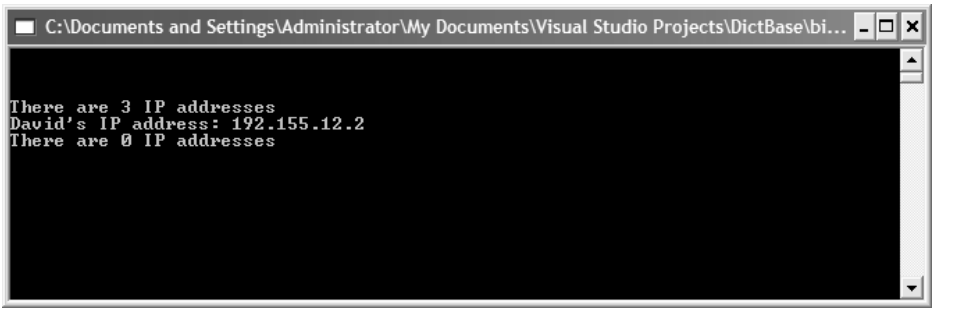
There are two methods we can use without implementing them—Count and Clear. The Count method returns the number of DictionaryEntry objects stored in the inner hash table; Clear removes all the DictionaryEntry objects from the inner hash table.

Let's look at a program that utilizes these methods:

```
Sub Main()
  Dim myIPs As New IPAddresses
  myIPs.Add("Mike", "192.155.12.1")
  myIPs.Add("David", "192.155.12.2")
  myIPs.Add("Bernica", "192.155.12.3")
  Console.WriteLine("There are " & myIPs.Count() & _
                    " IP addresses.")
```

```
    Console.WriteLine("David's ip: " & _
                      myIPs.Item("David"))
    myIPs.Clear()
    Console.WriteLine("There are " & myIPs.Count() & _
                      " IP addresses.")
    Console.Read()
  End Sub
```

The output from this program looks like this:



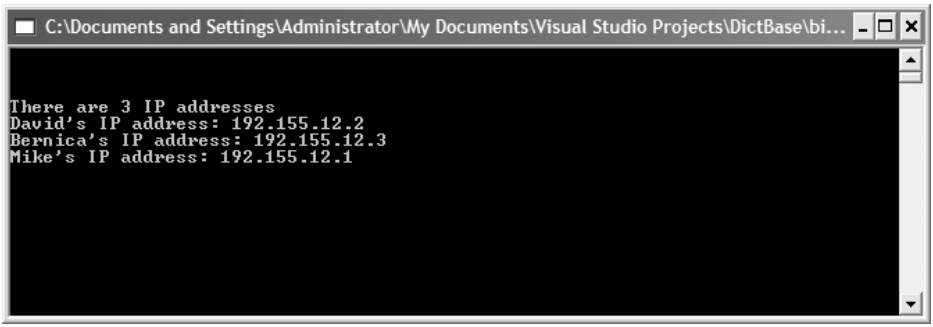
One modification we might want to make to the class is to overload the constructor so that we can load data into a dictionary from a file. Here's the code for the new constructor, which you can just add into the IPAddresses class definition:

```
  Public Sub New(ByVal txtFile As String)
    MyBase.New()
    Dim line As String
    Dim words() As String
    Dim inFile As StreamReader
    inFile = File.OpenText(txtFile)
    While (inFile.Peek <> -1)
      line = inFile.ReadLine()
      words = line.Split(",")c)
      Me.InnerHashtable.Add(words(0), words(1))
    End While
    inFile.Close()
  End Sub
```

Now here's a new program to test the constructor:

```
Sub Main()
  Dim myIPs As New IPAddresses(c:\data\ips.txt")
  Dim index As Integer
  For index = 1 To 3
    Console.WriteLine()
  Next
  Console.WriteLine("There are {0} IP addresses", _
                    myIPs.Count)
  Console.WriteLine("David's IP address: " & _
                    myIPs.Item("David"))
  Console.WriteLine("Bernica's IP address: " & _
                    myIPs.Item("Bernica"))
  Console.WriteLine("Mike's IP address: " & _
                    myIPs.Item("Mike"))
  Console.Read()
End Sub
```

The output this program is the following:



Other DictionaryBase Methods

There are two other methods that are members of the DictionaryBase class: CopyTo and GetEnumerator. We discuss these methods in this section.

The CopyTo method copies the contents of a dictionary to a one-dimensional array. The array should be declared as a DictionaryEntry array,

though you can declare it as Object and then use the CType function to convert the objects to DictionaryEntry.

The following code fragment demonstrates how to use the CopyTo method:

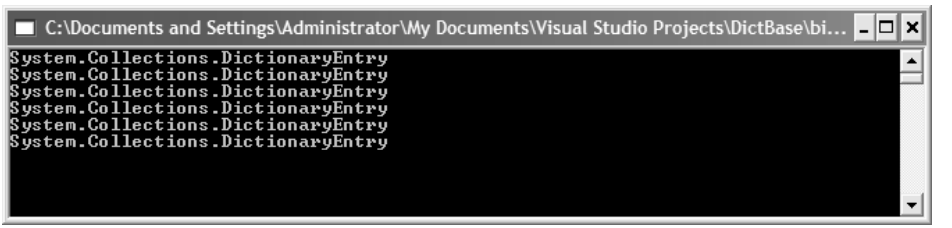
```
Dim myIPs As New IPAddresses("c:\ips.txt")
Dim ips((myIPs.Count-1) As DictionaryEntry
myIPs.CopyTo(ips, 0)
```

The formula used to size the array takes the number of elements in the dictionary and then subtracts one to account for a zero-based array. The CopyTo method takes two arguments: the array to copy to and the index position to start copying from. If you want to place the contents of a dictionary at the end of an existing array, for example, you would specify the upper bound of the array plus one as the second argument.

Once we get the data from the dictionary into an array, we want to work with the contents of the array, or at least display the values. Here's some code to do that:

```
Dim index As Integer
For index = 0 To ips.GetUpperBound(0)
  Console.WriteLine(ips(index))
Next
```

The output from this code looks like this:



Unfortunately, this is not what we want. The problem is that we're storing the data in the array as DictionaryEntry objects, and that's exactly what we see. If we use the ToString method

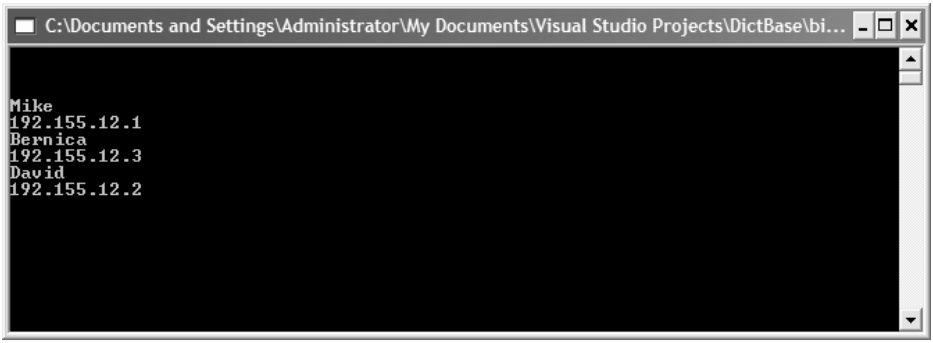
```
Console.WriteLine(ips(index).ToString())
```

we get the same thing. To actually view the data in a DictionaryEntry object, we have to use either the Key property or the Value property, depending on whether the object we're querying holds key data or value data. So how do we know which is which? When the contents of the dictionary are copied to the array, the data get copied in key–value order. So the first object is a key, the second object is a value, the third object is a key, and so on.

Now we can write a code fragment that allows us to actually see the data:

```
Dim index As Integer
For index = 0 To ips.GetUpperBound(0)
  Console.WriteLine(ips(index).Key)
Console.WriteLine(ips(index).Value)
Next
```

The output looks like this:



The SortedList Class

As we mentioned in the chapter's introduction, a SortedList is a data structure that stores key–value pairs in sorted order based on the key. We can use this data structure when it is important for the keys to be sorted, such as in a standard word dictionary, where we expect the words in the dictionary to be sorted alphabetically. Later in the chapter we'll also see how the class can be used to store a list of single, sorted values.

Using the SortedList Class

We can use the SortedList class in much the same way we used the classes in the previous sections, since the SortedList class is but a specialization of the DictionaryBase class.

To demonstrate this, the following code creates a SortedList object that contains three names and IP addresses:

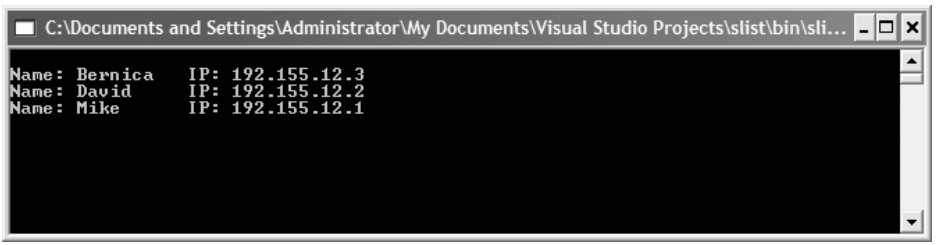
```
Dim myips As New SortedList
myips.Add("Mike", "192.155.12.1")
myips.Add("David", "192.155.12.2")
myips.Add("Bernica", "192.155.12.3")
```

The name is the key and the IP address is the stored value.

We can retrieve the values by using the Item method with a key as the argument:

```
Dim key As Object
For Each key In myips.Keys
  Console.WriteLine("Name: " & key & Constants.vbTab & _
                    "IP: " & myips.Item(key))
Next
```

This fragment produces the following output:

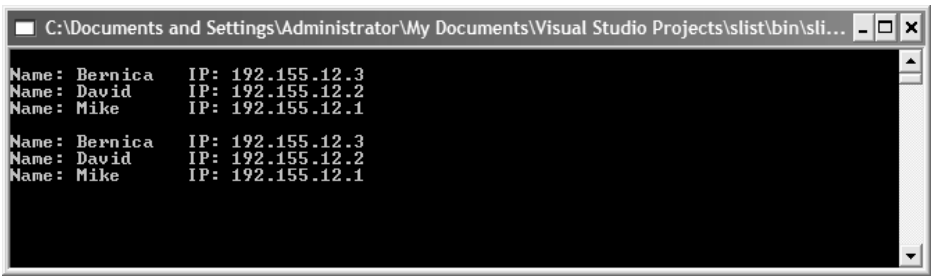


Alternatively, we can also access this list by referencing the index numbers where these values (and keys) are stored internally in the arrays that actually store the data. Here's how:

```
Dim i As Integer
For i = 0 To myips.Count - 1
```

```
    Console.WriteLine("Name: " & myips.GetKey(i) & _
                      Constants.vbTab & "IP: " & _
                      myips.GetByIndex(i))
Next
```

This code fragment produces the exact same sorted list of names and IP addresses:



A key–value pair can be removed from a SortedList by specifying either a key or an index number, as in the following code fragment, which demonstrates both removal methods:

```
myips.Remove("David")
myips.RemoveAt(1)
```

If you want to use index-based access into a SortedList but don't know the indexes where a particular key or value is stored, you can use the following methods to determine those values:

```
Dim indexDavid As Integer = myips.GetIndexOfKey("David")
Dim indexIPDavid As Integer = _
    myips.GetIndexOfValue(myips.Item("David"))
```

The SortedList class contains many other methods, and you are encouraged to explore them via VS.NET's online documentation.

Summary

The DictionaryBase class is an abstract class used to create custom dictionaries. A dictionary is a data structure that stores data in key–value pairs, using a

hash table (or sometimes a singly linked list) as the underlying data structure. The key–value pairs are stored as DictionaryEntry objects and you must use the Key and Value methods to retrieve the actual values in a DictionaryEntry object.

The DictionaryBase class is often used when the programmer wants to create a strongly typed data structure. Normally, data added to a dictionary is stored as an Object type, but with a custom dictionary, the programmer can reduce the number of type conversions that must be performed, making the program more efficient and easier to read.

The SortedList class is a particular type of Dictionary class, one that stores the key–value pairs in order sorted by the key. You can also retrieve the values stored in a SortedList by referencing the index number where the value is stored, much like you do with an array.

Exercises

1. Using the implementation of the IPAddresses class developed in this chapter, devise a method that displays the IP addresses stored in the class in ascending order. Use the method in a program.
2. Write a program that stores names and phone numbers from a text file in a dictionary, with the name being the key. Write a method that does a reverse lookup, that is, finds a name given a phone number. Write a Windows application to test your implementation.
3. Using a dictionary, write a program that displays the number of occurrences of a word in a sentence. Display a list of all the words and the number of times they occur in the sentence.
4. Rewrite Exercise 3 to work with letters rather than words.
5. Rewrite Exercise 2 using the SortedList class.
6. The SortedList class is implemented using two internal arrays, one that stores the keys and one that stores the values. Create your own SortedList class implementation using this scheme. Your class should include all the methods discussed in this chapter. Use your class to solve the problem posed in Exercise 2.

CHAPTER 10

Hashing and the Hashtable Class

Hashing is a very common technique for storing data in such a way that the data can be inserted and retrieved very quickly. Hashing uses a data structure called a *hash table*. Although hash tables provide fast insertion, deletion, and retrieval, they perform poorly for operations that involve searching, such as finding the minimum or maximum value. For these types of operations, other data structures are preferred (see, for example, Chapter 14 on binary search trees).

The .NET Framework library provides a very useful class for working with hash tables, the Hashtable class. We will examine this class in this chapter, but we will also discuss how to implement a custom hash table. Building hash tables is not very difficult and the programming techniques used are well worth knowing.

An Overview of Hashing

A hash table data structure is designed around an array. The array consists of elements 0 through some predetermined size, though we can increase the size later if necessary. Each data item is stored in the array based on some piece of the data, called the *key*. To store an element in the hash table, the key is

mapped into a number in the range of 0 to the hash table size using a function called a *hash function*.

Ideally, the hash function stores each key in its own cell in the array. However, because there are an unlimited number of possible keys and a finite number of array cells, a more realistic goal of the hash function is to attempt to distribute the keys as evenly as possible among the cells of the array.

Even with a good hash function, as you have probably guessed by now, it is possible for two keys to hash to the same value. This is called a *collision* and we have to have a strategy for dealing with collisions when they occur. We'll discuss this in detail in the following.

The last thing we have to determine is how large to dimension the array used as the hash table. First, it is recommended that the array size be a prime number. We will explain why when we examine the different hash functions. After that, there are several different strategies for determining the proper array size, all of them based on the technique used to deal with collisions, so we'll examine this issue in the following discussion also.

Choosing a Hash Function

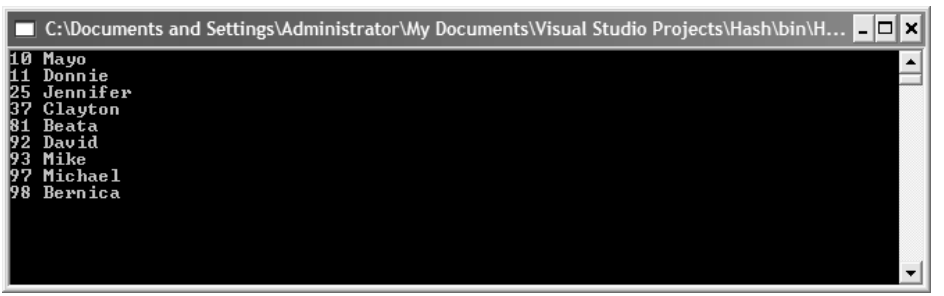
Choosing a hash function depends on the data type of the key you are using. If your key is an integer, the simplest function is to return the key modulo the size of the array. There are circumstances when this method is not recommended, such as when the keys all end in zero and the array size is 10. This is one reason why the array size should always be prime. Also, if the keys are random integers then the hash function should more evenly distribute the keys.

In many applications, however, the keys are strings. Choosing a hash function to work with keys proves to be more difficult and the hash function should be chosen carefully. A simple function that at first glance seems to work well is to add the ASCII values of the letters in the key. The hash value is that value modulo the array size. The following program demonstrates how this function works:

```
Option Strict On

Module Module1

  Sub Main()
    Dim names(99), name As String
    Dim someNames() As String = {"David", "Jennifer", _
      "Donnie", "Mayo", "Raymond", "Bernica", "Mike", _
```

```
        "Clayton", "Beata", "Michael"}
      Dim hashVal, index As Integer
      For index = 0 To 9
        name = someNames(index)
        hashVal = SimpleHash(name, names)
        names(hashVal) = name
      Next
      showDistrib(names)
      Console.Read()
    End Sub

    Function SimpleHash(ByVal s As String, _
                        ByVal arr() As String) As Integer
      Dim tot, index As Integer
      For index = 0 To s.Length - 1
        tot += Asc(s.Chars(index))
      Next
      Return tot Mod arr.GetUpperBound(0)
    End Function

    Sub showDistrib(ByVal arr() As String)
      Dim index As Integer
      For index = 0 To arr.GetUpperBound(0)
        If (arr(index) <> "") Then
          Console.WriteLine(index & " " & arr(index))
        End If
      Next
    End Sub

  End Module
```

The output from this program looks like this:



The showDistrib subroutine shows us where the names are actually placed into the array by the hash function. As you can see, the distribution is not particularly even. The names are bunched at the beginning of the array and at the end.

There is an even bigger problem lurking here, though. Not all of the names are displayed. Interestingly, if we change the size of the array to a prime number, even a prime lower than 99, all the names are stored properly. Hence, one important rule when choosing the size of your array for a hash table (and when using a hash function such as the one we're using here) is to choose a number that is prime.

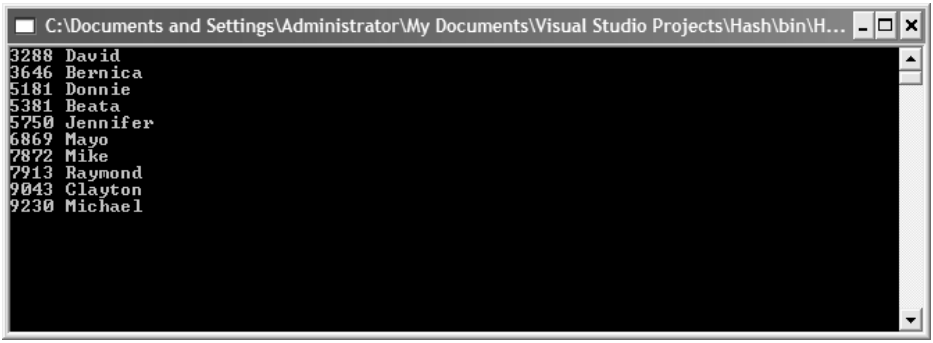
The size you ultimately choose will depend on your determination of the number of records stored in the hash table, but a safe number seems to be 10,007 (given that you're not actually trying to store that many items in your table). The number 10,007 is prime and its memory requirements are not large enough to degrade the performance of your program.

Maintaining the basic idea of using the computed total ASCII value of the key in the creation of the hash value, this next algorithm provides for a better distribution in the array. First, let's look at the code:

```
Function BetterHash(ByVal s As String, ByVal arr() _
                    As String) As Integer
  Dim index As Integer
  Dim tot As Long
  For index = 0 To s.Length - 1
    tot += 37 * tot + Asc(s.Chars(index))
  Next
  tot = tot Mod arr.GetUpperBound(0)
  If (tot < 0) Then
    tot += arr.GetUpperBound(0)
  End If
  Return CInt(tot)
End Function
```

This function uses Horner's rule to compute the polynomial function (of 37). See Weiss (1999) for more information on this hash function.

Now let's look at the distribution of the keys in the hash table using this new function:



These keys are more evenly distributed though it's hard to tell with such a small data set.

Searching for Data in a Hash Table

To search for data in a hash table, we need to compute the hash value of the key and then access that element in the array. It is that simple. Here's the function:

```
Function inHash(ByVal s As String, ByVal arr() As _
                String) As Boolean
  Dim hval As Integer
  hval = BetterHash(s, arr)
  If (arr(hval) = s) Then
    Return True
  Else
    Return False
  End If
End Function
```

This function returns True if the item is in the hash table and returns False otherwise. We don't even need to compare the time this function runs versus a sequential search of the array since this function clearly runs in less time, unless of course the data item is somewhere close to the beginning of the array.

Handling Collisions

When working with hash tables, it is inevitable that you will encounter situations where the hash value of a key works out to a value that is already storing another key. This is called a collision and there are several techniques you can use when a collision occurs. These techniques include bucket hashing, open addressing, and double hashing. In this section we will briefly cover each of these techniques.

Bucket Hashing

When we originally defined a hash table, we stated that it is preferred that only one data value resides in a hash table element. This works great if there are no collisions, but if a hash function returns the same value for two data items, we have a problem.

One solution to the collision problem is to implement the hash table using *buckets*. A bucket is a simple data structure stored in a hash table element that can store multiple items. In most implementations, this data structure is an array, but in our implementation we'll make use of an arraylist, thereby precluding us from having to worry about running out of space and allocating more space. In the end, this will make our implementation more efficient.

To insert an item, we first use the hash function to determine in which arraylist to store the item. Then we check to see whether the item is already in the arraylist. If it is we do nothing; if it's not, then we call the Add method to insert the item into the arraylist.

To remove an item from a hash table, we again first determine the hash value of the item to be removed and go to that arraylist. We then check to make sure the item is in the arraylist, and if it is, we remove it.

Here's the code for a BucketHash class that includes a Hash function, an Add method, and a Remove method:

```
Public Class BucketHash

  Private Const SIZE As Integer = 101
  Private data() As ArrayList

  Public Sub New()
    Dim index As Integer
    ReDim data(SIZE)
    For index = 0 To SIZE - 1
```

```
      data(index) = New ArrayList(4)
    Next
  End Sub

  Private Function Hash(ByVal s As String) As Integer
    Dim index As Integer
    Dim tot As Long
    For index = 0 To s.Length - 1
      tot += 37 * tot + Asc(s.Chars(index))
    Next
    tot = tot Mod data.GetUpperBound(0)
    If (tot < 0) Then
      tot += data.GetUpperBound(0)
    End If
    Return CInt(tot)
  End Function

  Public Sub Insert(ByVal item As String)
    Dim hash_value As Integer
    hash_value = Hash(item)
    If Not (data(hash_value).Contains(item)) Then
      data(hash_value).Add(item)
    End If
  End Sub

  Public Sub Remove(ByVal item As String)
    Dim hash_value As Integer
    hash_value = Hash(item)
    If (data(hash_value).Contains(item)) Then
      data(hash_value).Remove(item)
    End If
  End Sub

End Class
```

When using bucket hashing, you should keep the number of arraylist elements used as low as possible. This minimizes the extra work that has to be done when adding items to or removing items from the hash table. In the preceding code, we minimize the size of the arraylist by setting the initial capacity of each arraylist to 1 in the constructor call. Once we have a collision, the arraylist capacity becomes 2, and then the capacity continues to

double every time the arraylist fills up. With a good hash function, though, the arraylist shouldn't get too large.

The ratio of the number of elements in the hash table to the table size is called the *load factor*. Studies have shown that peak hash table performance occurs when the load factor is 1.0, or when the table size exactly equals the number of elements.

Open Addressing

Separate chaining decreases the performance of your hash table by using arraylists. An alternative to separate chaining for avoiding collisions is *open addressing*. An open addressing function looks for an empty cell in the hash table array in which to place an item. If the first cell tried is full, the next empty cell is tried, and so on until an empty cell is eventually found. We will look at two different strategies for open addressing in this section: linear probing and quadratic probing.

Linear probing uses a linear function to determine the array cell to try for an insertion. This means that cells will be tried sequentially until an empty cell is found. The problem with linear probing is that data elements will tend to cluster in adjacent cells in the array, making successive probes for empty cells longer and less efficient.

Quadratic probing eliminates the clustering problem. A quadratic function is used to determine which cell to attempt. An example of such a function is

```
2 * collNumber – 1
```

where collNumber is the number of collisions that have occurred during the current probe. An interesting property of quadratic probing is that it guarantees an empty cell being found if the hash table is less than half empty.

Double Hashing

This simple collision-resolution strategy does exactly what its name proclaims: If a collision is found, the hash function is applied a second time and then it probes at the distance sequence hash(item), 2hash(item), 4hash(item), etc. until an empty cell is found.

To make this probing technique work correctly, a few conditions must be met. First, the hash function chosen must never evaluate to zero, which would lead to disastrous results (since multiplying by zero produces zero). Second,

the table size must be prime. If the size isn't prime, then all the array cells will not be probed, again leading to chaotic results.

Double hashing is an interesting collision-resolution strategy, but it has been shown in practice that quadratic probing usually leads to better performance.

We are now finished examining custom hash table implementations. For most applications using VB.NET, you are better off using the built-in Hashtable class, which is part of the .NET Framework library. We begin our discussion of this class next.

The Hashtable Class

The Hashtable class is a special type of Dictionary object that stores key–value pairs, with the values being stored based on the hash code derived from the key. You can specify a hash function or use the one built in (which will be discussed later) for the data type of the key. Because of the Hashtable class's efficiency, it should be used in place of custom implementations whenever possible.

The strategy the class uses to avoid collisions involves the concept of a bucket. A bucket is a virtual grouping of objects that have the same hash code, much like we used an ArrayList to handle collisions when we discussed separate chaining. If two keys have the same hash code, they are placed in the same bucket. Every key with a unique hash code is placed in its own bucket.

The number of buckets used in a Hashtable object is called the *load factor*. The load factor is the ratio of the elements to the number of buckets. Initially, the factor is set to 1.0. When the actual factor reaches the initial factor, the load factor is increased to the smallest prime number that is twice the current number of buckets. The load factor is important because the smaller the load factor, the better the performance of the Hashtable object.

Instantiating and Adding Data to a Hashtable Object

The Hashtable class is part of the System.Collections namespace, so you must import System.Collections at the beginning of your program.

A Hashtable object can be instantiated in various ways. We will focus on the three most common constructors here. You can instantiate the hash table with an initial capacity or by using the default capacity. You can also specify both the initial capacity and the initial load factor. The following code demonstrates

how to use these three constructors:

```
Dim symbols As New Hashtable()
Dim symbols As New Hashtable(50)
Dim symbols As New Hashtable(25, 3.0)
```

The first line creates a hash table with the default capacity and the default load factor. The second line creates a hash table with a capacity of 50 elements and the default load factor. The third line creates a hash table with an initial capacity of 25 elements and a load factor of 3.0.

Key–value pairs are entered into a hash table using the Add method. This method takes two arguments: the key and the value associated with the key. The key is added to the hash table after computing its hash value. Here is some example code:

```
Dim symbols As New Hashtable(25)
symbols.Add("salary", 100000)
symbols.Add("name", "David Durr")
symbols.Add("age", 43)
symbols.Add("dept", "Information Technology")
```

You can also add elements to a hash table using the Item method, which we discuss more completely later. To do this, you write an assignment statement that assigns a value to the key specified in the Item method. If the key doesn't already exist, a new hash element is entered into the table; if the key already exists, the existing value is overwritten by the new value. Here are some examples:

```
symbols.Item("sex") = "Male"
symbols.Item("age") = 44
```

The first line shows how to create a new key–value pair using the Item method; the second line demonstrates that you can overwrite the current value associated with an existing key.

Retrieving the Keys and the Values Separately from a Hash Table

The Hashtable class has two very useful methods for retrieving the keys and values separately from a hash table: Keys and Values. These methods create

an Enumerator object that allows you to use a For Each loop, or some other technique, to examine the keys and the values.

The following program demonstrates how these methods work:

```
Option Strict On
Imports System.Collections

Module Module1

  Sub main()
    Dim symbols As New Hashtable(25)
    symbols.Add("salary", 100000)
    symbols.Add("name", "David Durr")
    symbols.Add("age", 43)
    symbols.Add("dept", "Information Technology")
    symbols.Item("sex") = "Male"
    Dim key, value As Object
    Console.WriteLine("The keys are: ")
    For Each key In symbols.Keys
      Console.WriteLine(key)
    Next
    Console.WriteLine()
    Console.WriteLine("The values are: ")
    For Each value In symbols.Values
      Console.WriteLine(value)
    Next
    Console.Read()
  End Sub

End Module
```

Retrieving a Value Based on the Key

The primary method for retrieving a value using its associated key is the Item method. This method takes a key as an argument and returns the value associated with the key, or nothing if the key doesn't exist.

The following short code segment demonstrates how the Item method works:

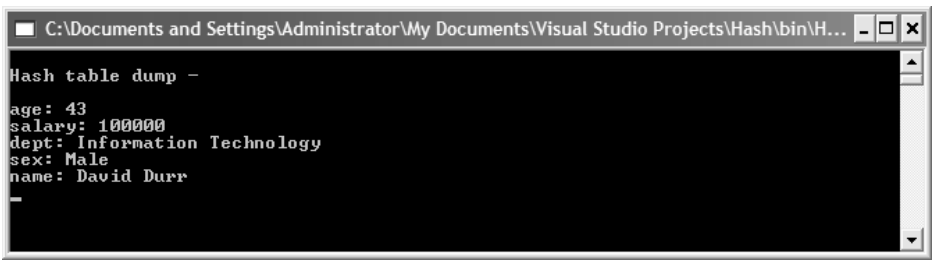
```
value = symbols.Item("name")
Console.WriteLine("The variable name's value is: " & _
                  CStr(value))
```

The value returned is “David Durr”.

We can use the Item method along with the Keys method to retrieve all the data stored in a hash table:

```
Sub main()
  Dim symbols As New Hashtable(25)
  symbols.Add("salary", 100000)
  symbols.Add("name", "David Durr")
  symbols.Add("age", 43)
  symbols.Add("dept", "Information Technology")
  symbols.Item("sex") = "Male"
  Dim key, value As Object
  Console.WriteLine()
  Console.WriteLine("Hash table dump - ")
  Console.WriteLine()
  For Each key In symbols.Keys _
    Console.WriteLine(CStr(key) & ": " & _
    CStr(symbols.Item(key)))
  Next
  Console.Read()
End Sub
```

The output looks like this:



Utility Methods of the Hashtable Class

There are a number of methods in the Hashtable class that can help you become more productive with Hashtable objects. In this section we examine several of them, including methods for determining the number of elements in a hash table, clearing the contents of a hash table, determining whether a specified key (and value) is contained in a hash table, removing elements from a hash table, and copying the elements of a hash table to an array.

The number of elements in a hash table is stored in the Count property, which returns an integer:

```
Dim numElements As Integer
numElements = symbols.Count
```

We can immediately remove all the elements of a hash table using the Clear method:

```
symbols.Clear()
```

To remove a single element from a hash table, you can use the Remove method. This method takes a single argument, a key, and removes both the specified key and its associated value. Here's an example:

```
symbols.Remove("sex")
For Each key In symbols.Keys
  Console.WriteLine(CStr(key) & ": " & _
                    CStr(symbols.Item(key)))
Next
```

Before you remove an element from a hash table, you may want to check to see whether either the key or the value is in the table. We can determine this information with the ContainsKey method and the ContainsValue method. The following code fragment demonstrates how to use the ContainsKey method:

```
Dim aKey As String
Console.Write("Enter a key to remove: ")
aKey = Console.ReadLine()
If (symbols.ContainsKey(aKey)) Then symbols.Remove(aKey)
```

Using this method ensures that the key–value pair you wish to remove exists in the hash table. The ContainsValue method works similarly with values instead of keys.

There are times when you may need to transfer the contents of a Hashtable object to an array. The Hashtable class has a method called CopyTo that handles this automatically. This method takes two arguments, an array and a starting index, and copies the elements of a hash table to an array. The array to which you copy must be an Object array.

Here's some sample code using the CopyTo method:

```
Dim numElements As Integer = symbols.Count
Dim syms(numElements - 1) As Object
```

```
symbols.CopyTo(syms, 0)
Console.WriteLine("Displaying symbol table from array: ")
Dim index As Integer
For index = 0 To syms.GetUpperBound(0)
  Console.WriteLine("{0}: {1}", _
      CType(syms(index), DictionaryEntry).Key, _
      CType(syms(index), DictionaryEntry).Value)
Next
```

Notice carefully how we write out the contents of the array. The elements of the hash table are written to the array as DictionaryEntry objects. These objects cannot be displayed directly, but we can use the CType function to utilize the Key and Value methods of the DictionaryEntry class to display the values.

A Hashtable Application: Constructing a Glossary

One common use of a hash table is to build a glossary, or dictionary, of terms. In this section we demonstrate one way to use a hash table for just such a use: We will construct a glossary of computer terms.

The program works by first reading in a set of terms and definitions from a text file. This process is coded in the BuildGlossary subroutine. The text file is structured as *word,definition*, with the comma being the delimiter between a word and the definition. Each word in this glossary is a single word, but the glossary could easily work with phrases instead. That's why a comma is used as the delimiter, rather than a space. Also, this structure allows us to use the word as the key, which is the proper way to build this hash table.

Another subroutine, DisplayWords, displays the words in a list box so that the user can pick one to get a definition. Since the words are the keys, we can use the Keys method to return just the words from the hash table. The user can then see which words have definitions.

To retrieve a definition, the user simply clicks on a word in the list box. The definition is retrieved using the Item method and is displayed in the textbox.

Here's the code:

```
Option Strict On
Imports System.io
Imports System.collections

Public Class Form1
```

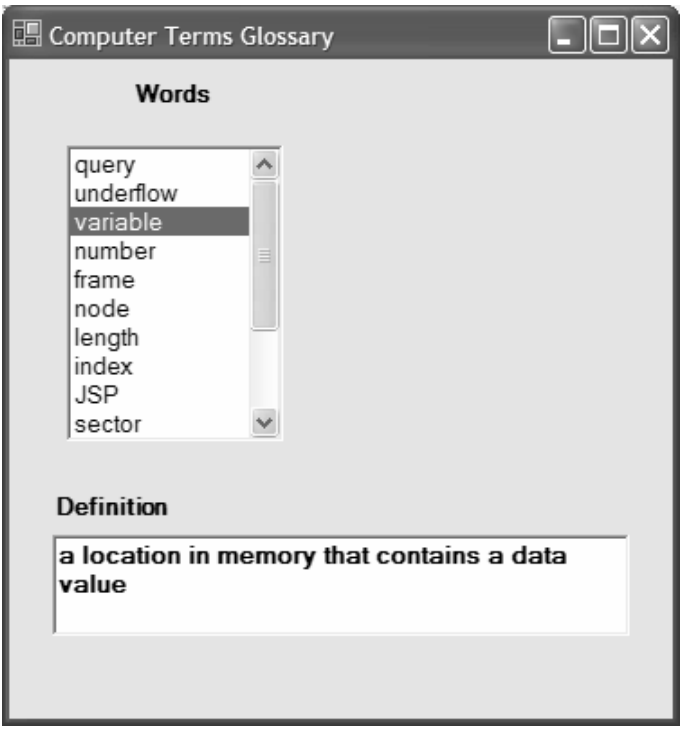
```
  Inherits System.Windows.Forms.Form

  Dim glossary As New Hashtable

'The region holding Windows-generated code is not shown here

  Sub BuildGlossary(ByRef g As Hashtable)
    Dim inFile As StreamReader
    Dim line As String
    Dim words() As String
    inFile = File.OpenText("c:\words.txt")
    While (inFile.Peek <> -1)
      line = inFile.ReadLine
      words = line.Split(","c)
      g.Add(words(0), words(1))
    End While
    inFile.Close()
  End Sub
  Sub DisplayWords(ByVal g As Hashtable)
    Dim words(99) As Object
    g.Keys.CopyTo(words, 0)
    Dim index As Integer
    For index = 0 To words.GetUpperBound(0) Step 3
      If Not (words(index) Is Nothing) Then
        lstWords.Items.Add(words(index))
      End If
    Next
  End Sub

  Private Sub Form1_Load(ByVal sender As _
      System.Object, ByVal e As System.EventArgs) _
      Handles MyBase.Load
    BuildGlossary(glossary)
    DisplayWords(glossary)
  End Sub

  Private Sub lstWords_SelectedIndexChanged(ByVal _
      sender As System.Object, ByVal e As _
      System.EventArgs) Handles _
      lstWords.SelectedIndexChanged
    Dim word As Object
```

```
        word = lstWords.SelectedItem
        txtDef.Text = CStr(glossary.Item(word))
    End Sub

End Class
```

The text file looks like this:

adder,an electronic circuit that performs an addition operation on binary values addressability,the number of bits stored in each addressable location in memory bit,short for binary digit
block,a logical group of zero or more program statements
call,the point at which the computer begins following the instructions in a subprogram
compiler,a program that translates a high-level program into machine code
data,information in a form a computer can use
database,a structured set of data
. . .

Here's how the program looks when it runs:



If a word is entered that is not in the glossary, the Item method returns Nothing. There is a test for Nothing in the GetDefinition subroutine so that the string "not found" is displayed if the word entered is not in the hash table.

Summary

A hash table is a very efficient data structure for storing key–value pairs. The implementation of a hash table is mostly straightforward, with the tricky part having to do with choosing a strategy for collisions. This chapter discussed several techniques for handling collisions.

For most VB.NET applications, there is no reason to build a custom hash table, because the Hashtable class of the .NET Framework library works quite well. You can specify your own hash function for the class or you can let the class calculate hash values

Exercises

1. Rewrite the computer terms glossary application using the custom-designed Hash class developed in the chapter. Experiment with different hash functions and collision-resolution strategies.
2. Using the Hashtable class, write a spelling-checker program that reads through a text file and checks for spelling errors. You will, of course, have to limit your dictionary to several common words.
3. Create a new Hash class that uses an ArrayList instead of an array for the hash table. Test your implementation by rewriting (yet again) the computer terms glossary application.

Chapter 11

Linked Lists

For many applications, data are best stored as lists, and lists occur naturally in day-to-day life—to-do lists, grocery lists, and top-ten lists. In this chapter we explore one particular type of list, the linked list. Although the .NET Framework class library contains several list-based collection classes, the linked list is not among them. The chapter starts with an explanation of why we need linked lists, then we explore two different implementations of the data structure—object-based linked lists and array-based linked lists. The chapter finishes up with several examples of how linked lists can be used for solving computer programming problems you may run across.

Shortcomings of Arrays

The array is the natural data structure to use when working with lists. Arrays provide fast access to stored items and are easy to loop through. And, of course, the array is already part of the language and you don't have to use extra memory and processing time using a user-defined data structure.

However, as we've seen, the array is not the perfect data structure. Searching for an item in an unordered array can be slow because you might have to visit every element in the array before finding the element you're searching for. Ordered (sorted) arrays are much more efficient for searching, but insertions and deletions take extra time because you have to shift the elements up or

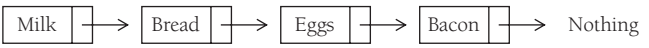


FIGURE 11.1. An Example Linked List.

down to either make space for an insertion or remove space with a deletion. Moreover, in an ordered array, you have to search for the proper space in the array into which to insert an element.

When you determine that the operations performed on an array are too slow for practical use, you can consider using the linked list as an alternative. The linked list can be used in almost every situation where an array is used, except if you need random access to the items in the list, when an array is probably the better choice.

DEFINITION OF A LINKED LIST

A linked list is a collection of class objects called nodes. Each node is linked to its successor node in the list using a reference to the successor node. A node consists of a field for storing data and the field for the node reference. The reference to another node is called a link. An example linked list is shown in Figure 11.1.

A major difference between an array and a linked list is that whereas the elements in an array are referenced by position (the index), the elements of a linked list are referenced by their relationship to the other elements of the array. In Figure 11.1, we say that "Bread" follows "Milk," not that "Bread" is in the second position. Moving through a linked list involves following the links from the beginning node to the ending node.

Another thing to notice in Figure 11.1 is that we mark the end of a linked list by pointing to the special value Nothing. Since we are working with class objects in memory, we use the equivalent of a null object, Nothing, to denote the end of the list.

Marking the beginning of a list can be problematic in some cases. Many linked-list implementations commonly include a special node, called the "header," to denote the beginning of a linked list. The linked list of Figure 11.1 is redesigned with a header node in Figure 11.2.

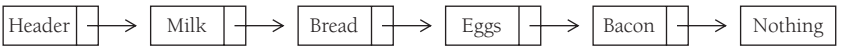


FIGURE 11.2. A Linked List with a Header Node.

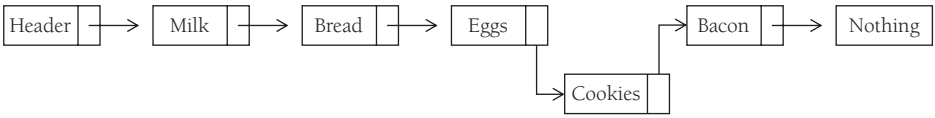


FIGURE 11.3. Inserting Cookies

Insertion becomes a very efficient task when using a linked list. All that is involved is changing the link of the node previous to the inserted node to point to the inserted node, and setting the link of the new node to point to the node the previous node pointed to before the insertion. Figure 11.3 show how the item "Cookies" gets added to the linked list after "Eggs."

Removing an item from a linked list is just as easy. We simply redirect the link of the node before the deleted node to point to the node the deleted node points to and set the deleted node's link to Nothing. The diagram of this operation is shown in Figure 11.4, where we remove "Bacon" from the linked list.

There are other methods we can, and will, implement in the LinkedList class, but insertion and deletion are the two methods that define why we use linked lists over arrays.

AN OBJECT-ORIENTED LINKED-LIST DESIGN

Our design of a linked list will involve at least two classes. We'll create a Node class and instantiate a Node object each time we add a node to the list. The nodes in the list are connected via references to other nodes. These references are set using methods created in a separate LinkedList class. Let's start by looking at the design of the Node class.

The Node Class

A node is made up of two data members: Element, which stores the node's data, and Link, which stores a reference to the next node in the list. We'll use Object for the data type of Element, just so we don't have to worry about what

FIGURE 11.4. Removing Bacon

kind of data we store in the list. The data type for Link is Node, which seems strange but actually makes perfect sense. Since we want the link to point to the next node, and we use a reference to make the link, we have to assign a Node type to the link member.

To finish up the definition of the Node class, we need at least two constructor methods. We definitely want a default constructor that creates an empty Node, with both the Element and Link members set to Nothing. We also need a parameterized constructor that assigns data to the Element member and sets the Link member to Nothing.

Here's the code for the Node class:

```
Public Class Node
  Public Element As Object
  Public Link As Node

  Public Sub New()
    Element = Nothing
    Link = Nothing
  End Sub

  Public Sub New(theElement As Object)
    Element = theElement
    Link = nothing
  End Sub
End Class
```

The LinkedList Class

The LinkedList class is used to create the linkage for the nodes of our linked list. The class includes several methods for adding nodes to the list, removing nodes from the list, traversing the list, and finding a node in the list. We also need a constructor method that instantiates a list. The only data member in the class is the header node.

Here's the code for the LinkedList class:

```
Public Class LinkedList
  Protected header As Node

  Public Sub New()
```

```
    header = New Node("header")
  End Sub
  ...
End Class
```

The header node starts out with its Link field set to Nothing. When we add the first node to the list, the header node's Link field is assigned a reference to the new node, and the new node's Link field is assigned Nothing.

The first method we'll examine is the Insert method, which we use to put a node into our linked list. To insert a node into the list, you have to specify which node you want to insert before or after. This enables you to adjust all the necessary links in the list. We'll choose to insert a new node after an existing node in the list.

To insert a new node after an existing node, we have to first find the "after" node. To do this, we create a Private method, Find, that searches through the Element field of each node until a match is found:

```
Private Function Find(ByVal item As Object) As Node
  Dim current As New Node()
  current = header
  While (current.Element <> item)
    current = current.Link
  End While
  Return current
End Function
```

This method demonstrates how we move through a linked list. First, we instantiate a Node object, current, and assign it as the header node. Then we check to see whether the value in the node's Element field equals the value we're searching for. If it does not, we move to the next node by assigning the node in the Link field of current as the new value of current.

Once we've found the "after" node, the next step is to set the new node's Link field to the Link field of the "after" node, and then set the "after" node's Link field to a reference to the new node. Here's how it's done:

```
Public Sub Insert(ByVal newItem As Object, ByVal after _
                  As Object)
  Dim current As New Node()
  Dim newnode As New Node(newItem)
```

```
    current = Find(after)
    newnode.Link = current.Link
    current.Link = newnode
  End Sub
```

The next linked-list operation we explore is Remove. To remove a node from a linked list, we simply have to change the link of the node that points to the removed node to point to the node after the removed node.

Since we need to find the node before the node we want to remove, we'll define a method, FindPrevious, that does this. This method steps through the list, stopping at each node and looking ahead to the next node to see whether that node's Element field holds the item we want to remove. Here is the code:

```
  Private Function FindPrevious(ByVal x As Object) As Node
    Dim current As Node = header
    While (Not (current.Link Is Nothing) And _
                current.Link.element <> x)
      current = current.Link
    End While
    Return current
  End Function
```

Now we're ready to see how the code for the Remove method looks:

```
  Public Sub Remove(ByVal x As Object)
    Dim p As Node = FindPrevious(x)
    If (Not (p.Link Is Nothing)) Then
      p.Link = p.Link.Link
    End If
  End Sub
```

The Remove method removes the first occurrence of an item in a linked list only. You will also notice that if the item is not in the list, nothing happens.

The last method we'll define in this section is PrintList, which traverses the linked list and displays the Element fields of each node in the list. Here's the code:

```
  Public Sub PrintList()
    Dim current As New Node()
```

```
    current = header
    While (Not (current.Link Is Nothing))
      Console.WriteLine(current.Link.Element)
      current = current.Link
    End While
  End Sub
```

Linked-List Design Modifications

There are several modifications we can make to our linked-list design to better solve certain problems. Two of the most common modifications are the doubly linked list and the circularly linked list. A doubly linked list makes it easier to move backward through a linked list and to remove a node from the list. A circularly linked list is convenient for applications that move more than once through a list. We'll look at both of these modifications in this section. Finally, we'll look at a modification to the LinkedLast class that is common only to object-oriented implementations of a linked list—an Iterator class for denoting position in the list.

Doubly Linked Lists

Although traversing a linked list from the first node in the list to the last node is very straightforward, it is not as easy to traverse a linked list backward. We can simplify this procedure if we add a field to our Node class that stores the link to the previous node. When we insert a node into the list, we'll have to perform more operations to assign data to the new field, but we gain efficiency when we have to remove a node from the list, since we don't have to look for the previous node. Figure 11.5 illustrates graphically how a doubly linked list works.

We first need to modify the Node class to add an extra link to the class. To distinguish between the two links, we'll call the link to the next node the

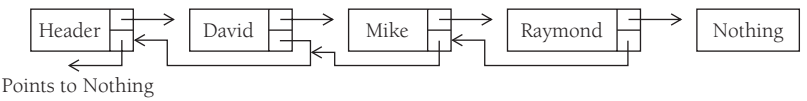


Figure 11.5. A Doubly-Linked List.

FLink, and the link to the previous node the BLink. These fields are set to Nothing when a Node is instantiated. Here's the code:

```
Public Class Node
  Public Element As Object
  Public Flink As Node
  Public Blink As Node

  Public Sub New()
    Element = ""
    Flink = Nothing
    Blink = Nothing
  End Sub

  Public Sub New(ByVal obj As Object)
    Element = obj
    Flink = Nothing
    Blink = Nothing
  End Sub
End Class
```

The Insertion method resembles the same method in a singularly linked list, except we have to set the new node's back link to point to the previous node. Here's the code:

```
Public Sub Insert(ByVal x As Object, ByVal y As Object)
  Dim current As New Node()
  Dim newnode As New Node(x)
  current = Find(y)
  newnode.Flink = current.Flink
  newnode.Blink = current
  current.Flink = newnode
End Sub
```

The Remove method for a doubly linked list is much simpler to write than for a singularly linked list. We first need to find the node in the list; then we set the node's back link property to point to the node pointed to by the deleted node's forward link. Then we need to redirect the back link of the link the deleted node points to and point it to the node before the deleted node.

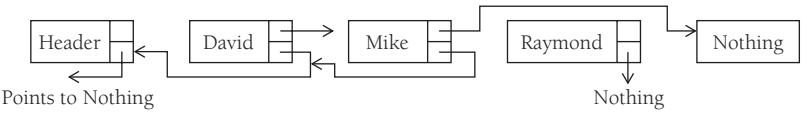


FIGURE 11.6. Removing a Node From a Doubly-Linked.

Figure 11.6 illustrates a special case of deleting a node from a doubly linked list when the node to be deleted is the last node in the list (other than the Nothing node).

The code for the Remove method of a doubly linked list is the following:

```
Public Sub Remove(ByVal x As Object)
  Dim p As Node = Find(x)
  If (Not (p.Flink Is Nothing)) Then
    p.Blink.Flink = p.Flink
    p.Flink.Blink = p.Blink
    p.Flink = Nothing
    p.Blink = Nothing
  End If
End Sub
```

We'll end this section on implementing doubly linked lists by writing a method that prints the elements of a linked list in reverse order. In a singularly linked list, this could be somewhat difficult, but with a doubly linked list, the method is easy to write.

First, we need a method that finds the last node in the list. This is just a matter of following each node's forward link until we reach a link that points to Nothing. This method, called FindLast, is defined as follows:

```
Private Function FindLast() As Node
  Dim current As New Node()
  current = header
  While (Not (current.Flink Is Nothing))
    current = current.Flink
  End While
  Return current
End Function
```

FIGURE 11.7. A Circularly-Linked List.

Once we find the last node in the list, to print the list in reverse order we just follow the backward link until we get to a link that points to Nothing, which indicates we're at the header node. Here's the code:

```
Public Sub PrintReverse()
  Dim current As New Node()
  current = FindLast()
  While (Not (current.Blink Is Nothing))
    Console.WriteLine(current.Element)
    current = current.Blink
  End While
End Sub
```

Circularly Linked Lists

A circularly linked list is a list in which the last node points back to the first node (which may be a header node). Figure 11.7 illustrates how a circularly linked list works.

This type of linked list is used in certain applications that require the last node pointing back to the first node (or the header). Many programmers choose to use circularly linked lists when a linked list is called for.

The only real change we have to make to our code is to point the Header node to itself when we instantiate a new linked list. If we do this, every time we add a new node the last node will point to the Header, since that link is propagated from node to node.

The code for a circularly linked list (showing the complete class for clarity) is the following:

```
Public Class LinkedList
  Protected current As ListNode
  Protected header As ListNode
  Private count As Integer = 0

  Public Sub New()
    header = New ListNode("header")
```

```
  header.link = header
End Sub

Public Function IsEmpty() As Boolean
  Return (header.link Is Nothing)
End Function

Public Sub MakeEmpty()
  header.link = Nothing
End Sub

Public Sub PrintList()
  Dim current As New ListNode
  current = header
  While (Not (current.link.element = "header"))
    Console.WriteLine(current.link.element)
    current = current.link
  End While
End Sub

Private Function FindPrevious(ByVal x As Object) _
   As ListNode
  Dim current As ListNode = header
  While (Not (current.link Is Nothing) And _
         current.link.element <> x)
    current = current.link
  End While
  Return current
End Function

Private Function Find(ByVal x As Object)
  Dim current As New ListNode
  current = header.link
  While (current.element <> x)
    current = current.link
  End While
  Return current
End Function

Public Sub Remove(ByVal x As Object)
  Dim p As ListNode = FindPrevious(x)
  If (Not (p.link Is Nothing)) Then
    p.link = p.link.link
```

```
    End If
    count -= 1
  End Sub

  Public Sub Insert(ByVal x As Object, ByVal y _
                    As Object)
    Dim current As New ListNode
    Dim newnode As New ListNode(x)
    current = Find(y)
    newnode.link = current.link
    current.link = newnode
    count += 1
  End Sub

  Public Sub InsertFirst(ByVal x As Object)
    Dim current As New ListNode(x)
    current.link = header
    header.link = current
    count += 1
  End Sub

  Public Function Move(ByVal n As Integer) As ListNode
    Static current As ListNode = header.link
    Dim temp As ListNode
    Dim x As Integer
    For x = 1 To n
      current = current.link
    Next
    If (current.element = "header") Then
      current = current.link
    End If
    temp = current
    Return temp
  End Function

  Public ReadOnly Property NumNodes() As Integer
    Get
      Return count
    End Get
  End Property
End Class
```

In VB.NET, the ArrayList data structure is implemented using a circularly linked list. There are also many problems that can be solved using a circularly linked list. We look at one typical problem in the exercises

Using an Iterator Class

One problem with the LinkedList class is that you can't refer to two positions in the linked list at the same time. We can refer to any one position in the list (the current node, the previous node, etc.), but if we want to specify two or more positions, such as if we want to remove a range of nodes from the list, we'll need some other mechanism. This mechanism is an iterator class.

The iterator class consists of three data fields, one each for storing the linked list, the current node, and the previous node. The constructor method is passed a linked-list object, and the method sets the current field to the header node of the list passed into the method. Let's look at our definition of the class so far:

```
Public Class ListIter
  Private current As Node
  Private previous As Node
  Private theList As LinkedList

    Public Sub New(list As LinkedList)
      theList = list
      current = theList.getFirst()
      previous = Nothing
    End Sub
```

The first thing we want an Iterator class to do is allow us to move from node to node through the list. The method nextLink does this:

```
Public Sub nextLink()
  previous = current
  current = current.Link
End Sub
```

Notice that, in addition to establishing a new current position, the previous node is also set to the node that is current before the method finishes

executing. Keeping track of the previous node in addition to the current node makes insertion and removal easier to perform.

The getCurrent method returns the node pointed to by the iterator:

```
Public Function getCurrent() As Node
  Return current
End Function
```

Two insertion methods are built into the Iterator class: InsertBefore and InsertAfter. InsertBefore inserts a new node before the current node; InsertAfter inserts a new node after the current node. Let's look at the InsertBefore method first.

Before inserting a new node before the current object we need to check to see whether we are at the beginning of the list. If we are, then we can't insert a node before the header node, so we throw an exception. (This exception is defined momentarily.) Otherwise, we set the new node's Link field to the Link field of the previous node, set the previous node's Link field to the new node, and reset the current position to the new node. Here's the code:

```
Public Sub InsertBefore(theElement As Object)
  Dim newNode As New Node(theElement)
  If (current = header) Then
    Throw New InsertBeforeHeaderException()
  Else
    newNode.Link = previous.Link
    previous.Link = newNode
    current = newNode
  End If
End Sub
```

The InsertBeforeHeader Exception class definition is

```
Public Class InsertBeforeHeaderException
      Inherits Exception
  Public Sub New()
    MyBase.New("Can't insert before the header node.")
  End Sub
End Class
```

The InsertAfter method in the Iterator class is much simpler than the method we wrote in the LinkedList class. Since we already know the position of the current node, the method just needs to set the proper links and set the current node to the next node. Here's the code:

```
Public Sub InsertAfter(theElement As Object)
  Dim newnode As New Node(theElement)
  newnode.Link = current.Link
  current.Link = newnode
  nextLink()
End Sub
```

Removing a node from a linked list is extremely easy using an Iterator class. The method simply sets the Link field of the previous node to the node pointed to by the current node's Link field:

```
Public Sub Remove()
  previous.Link = current.Link
End Sub
```

Other methods we need in an Iterator class include methods to reset the iterator to the header node (and the previous node to Nothing) and a method to test whether we're at the end of the list. These methods are as follows:

```
Public Sub Reset()
  current = theList.getFirst()
  previous = Nothing
End Sub

Public Function atEnd() As Boolean
  Return (current.Link Is Nothing)
End Function
```

The New LinkedList Class

With the Iterator class doing a lot of the work now, we can slim down the LinkedList class quite a bit. Of course, we still need a header field and a

constructor method to instantiate the list. Here's the code:

```
Public Class LinkedList
  Protected header As Node

  Public Sub New()
    header = New Node("Header")
  End Sub

  Public Function IsEmpty() As Boolean
    Return (header.Link Is Nothing)
  End Function

  Public Function getFirst() As Node
    return header
  End Function

  Public Sub showList()
    Dim current As Node = header.Link
    While (Not (current Is Nothing))
      Console.WriteLine(current.Element)
      current = current.Link
    End While
  End Sub

End Class
```

Demonstrating the Iterator Class

Using the Iterator class, it's easy to write an interactive program to move through a linked list. This also gives us a chance to put all the code for both the Iterator class and the LinkedList class in one place. The result is as follows:

```
Imports System
Imports Microsoft.VisualBasic

Module Module1
  Public Class Node
    Public Element As Object
```

```
  Public Link As Node

  Public Sub New()
    Element = Nothing
    Link = Nothing
  End Sub

  Public Sub New(ByVal obj As Object)
    Element = obj
    Link = Nothing
  End Sub

  Public Sub ShowNode()
    Console.Write("(" & Element & ")")
  End Sub

End Class

Public Class InsertBeforeHeaderException
      Inherits Exception
  Public Sub New()
    MyBase.New("Can't insert before the header node.")
  End Sub
End Class

Public Class LinkedList
  Protected header As Node

  Public Sub New()
    header = New Node("Header")
  End Sub

  Public Function IsEmpty() As Boolean
    Return (header.Link Is Nothing)
  End Function

  Public Function getFirst() As Node
    return header
  End Function

  Public Sub showList()
    Dim current As Node = header.Link
    While (Not (current Is Nothing))
      Console.WriteLine(current.Element)
```

```
      current = current.Link
    End While
  End Sub

End Class

Public Class ListIter
  Private current As Node
  Private previous As Node
  Private theList As LinkedList

  Public Sub New(list As LinkedList)
    theList = list
    current = theList.getFirst()
    previous = Nothing
  End Sub

  Public Sub Reset()
    current = theList.getFirst()
    previous = Nothing
  End Sub

  Public Function atEnd() As Boolean
    Return (current.Link Is Nothing)
  End Function

  Public Sub nextLink()
    previous = current
    current = current.Link
  End Sub

  Public Function getCurrent() As Node
    Return current
  End Function

  Public Sub InsertAfter(theElement As Object)
    Dim newnode As New Node(theElement)
    newnode.Link = current.Link
    current.Link = newnode
    nextLink()
  End Sub

  Public Sub InsertBefore(theElement As Object)
    Dim newNode As New Node(theElement)
```

```
    If (previous Is Nothing) Then
      Throw new InsertBeforeHeaderException
    Else
      newNode.Link = previous.Link
      previous.Link = newNode
      current = newNode
    End If
  End Sub

  Public Sub Remove()
    previous.Link = current.Link
  End Sub

End Class

Sub Main()
  Dim MyList As New LinkedList()
  Dim iter As New ListIter(MyList)
  Dim choice, value As String
  Try
    iter.InsertAfter("David")
    iter.InsertAfter("Mike")
    iter.InsertAfter("Raymond")
    iter.InsertAfter("Bernica")
    iter.InsertAfter("Jennifer")
    iter.InsertBefore("Donnie")
    iter.InsertAfter("Mike")
    iter.InsertBefore("Terrill")
    iter.InsertBefore("Mayo")
    While (true)
      Console.WriteLine("(n) Move to next node")
      Console.WriteLine("(g) Get value in current" & _
                        "node")
      Console.WriteLine("(r) Reset iterator")
      Console.WriteLine("(s) Show complete list")
      Console.WriteLine("(a) Insert after")
      Console.WriteLine("(b) Insert before")
      Console.WriteLine("(c) Clear the screen")
      Console.WriteLine("(x) Exit")
      Console.WriteLine
      Console.Write("Enter your choice: ")
```

```
choice = Console.ReadLine()
choice = choice.ToLower()
Select Case choice
  Case "n"
    If Not (MyList.IsEmpty()) And Not (iter. _
            atEnd()) Then
      iter.nextLink()
    Else
      Console.WriteLine("Can't move to next" & _
                        "link.")
    End If
  Case "g"
    If Not (MyList.IsEmpty()) Then
      Console.WriteLine("Element: " & iter. _
                         GetCurrent.Element)
    Else
      Console.WriteLine("List is empty.")
    End If
  Case "r"
    iter.Reset()
  Case "s"
    If Not(MyList.IsEmpty()) Then
      MyList.showList()
    Else
      Console.WriteLine("List is empty.")
    End If
  Case "a"
    Console.WriteLine()
    Console.Write("Enter value to insert: ")
    value = Console.ReadLine()
    iter.InsertAfter(value)
  Case "b"
    Console.WriteLine()
    Console.Write("Enter value to insert: ")
    value = Console.ReadLine()
    iter.InsertBefore(value)
  Case "c"
    Shell("c:\cls.bat",,True,)
  Case "x"
```

```
            End
        End Select
      End While
    Catch e As InsertBeforeHeaderException
        Console.WriteLine(e.Message)
    End Try
  End Sub
End Module
```

Indeed, this program is a character-based program and doesn't use a graphical user interface. You will get a chance to remedy this in the exercises, however.

Summary

In the traditional study of computer programming, linked lists are often the first data structure considered. In VB.NET, however, it is possible to use one of the built-in data structures, such as the ArrayList, and achieve the same result as implementing a linked list. However, it is well worth every programming student's time to learn how linked lists work and how to implement them. VB.NET uses a circularly linked list design to implement the ArrayList data structure.

There are several good books that discuss linked lists, though none of them use VB.NET as the target book. The definitive source, as usual, is Knuth's (1998) *The Art of Computer Programming, Volume I, Fundamental Algorithms*. Other books you might consult for more information include *Data Structures with C++*, by Ford and Topp (1996), and, if you're interested in Java implementations (and you should be because you can almost directly convert a Java implementation to one in VB.NET), consult *Data Structures and Algorithm Analysis in Java* (Weiss 1999).

Exercises

1. Rewrite the Console application that uses an iterator-based linked list as a Windows application.
2. According to legend, the 1st-century Jewish historian, Flavius Josephus, was captured along with a band of 40 compatriots by Roman soldiers during

the Jewish–Roman war. The Jewish soldiers decided that they preferred suicide to being captured and devised a plan for their demise. They were to form a circle and kill every third soldier until they were all dead. Joseph and one other decided they wanted no part of this and quickly calculated where they needed to place themselves in the circle so that they would both survive. Write a program that allows you to place *n* people in a circle and specify that every *m*th person will be killed. The program should determine the number of the last person left in the circle. Use a circularly linked list to solve the problem.

3. Write a program that can read an indefinite number of lines of VB.NET code and store reserved words in one linked list and identifiers and literals in another linked list. When the program is finished reading input, display the contents of each linked list.
4. Design and implement a ToArray method for the LinkedList class that takes a linked-list instance and returns an array.

Chapter 12

Binary Trees and Binary Search Trees

Trees constitute a very common data structure in computer science. A tree is a nonlinear data structure that is used to store data in a hierarchical manner. We examine one primary tree structure in this chapter, the binary tree, along with one implementation of the binary tree, the binary search tree. Binary trees are often chosen over more fundamental structures, such as arrays and linked lists, because you can search a binary tree quickly (as opposed to a linked list) and you can quickly insert data and delete data from a binary tree (as opposed to an array).

Definition of a Tree

Before we examine the structure and behavior of the binary tree, we need to define what we mean by a tree. A *tree* is a set of *nodes* connected by *edges*. An example of a tree is a company's organization chart (see Figure 12.1).

The purpose of an organization chart is to communicate to the viewer the structure of the organization. In Figure 12.1, each box is a node and the lines connecting the boxes are the edges. The nodes, obviously, represent the entities (people) that make up an organization. The edges represent relationships among the entities. For example, the CIO (Chief Information Officer) reports directly to the CEO (Chief Executive Officer), so there is an edge between

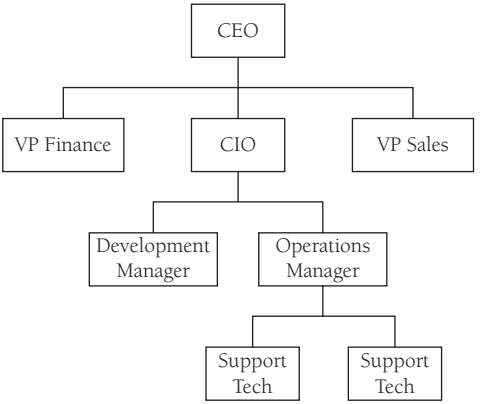


FIGURE 12.1. A Partial Organizational Chart.

these two nodes. The Development Manager reports to the CIO, so there is an edge connecting them. The VP (Vice President) of Sales and the Development Manager do not have a direct edge connecting them, so there is not a direct relationship between these two entities.

Figure 12.2 displays another tree that defines a few terms we need when discussing trees. The top node of a tree is called the *root* node. If a node is connected to other nodes below it, the top node is called the parent, and the nodes below it are called the parent's children. A node can have zero, one, or more nodes connected to it. Special types of trees, called *binary* trees,

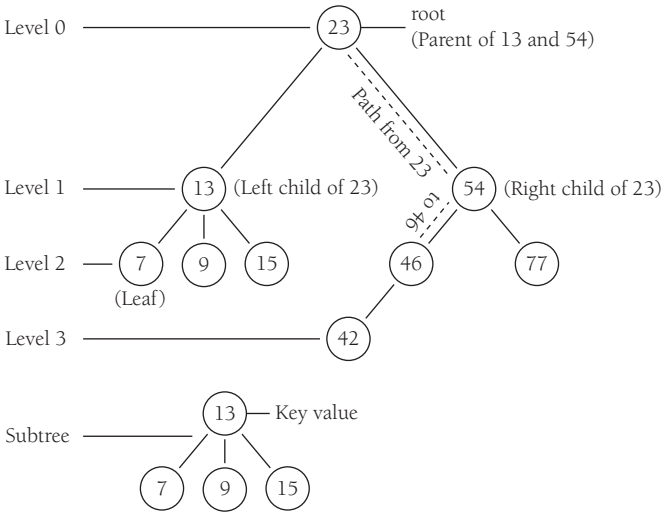


FIGURE 12.2. Parts of a tree.

restrict the number of children to no more than two. Binary trees have certain computational properties that make them very efficient for many operations. Binary trees are discussed extensively in the sections to follow. A node without any child nodes is called a *leaf*.

Continuing to examine Figure 12.2, you can see that by following certain edges, you can travel from one node to other nodes that are not directly connected. The series of edges you follow to get from one node to another is called a *path* (depicted in the figure with dashed lines). Visiting all the nodes in a tree in some particular order is known as a tree *transversal*.

A tree can be broken down into *levels*. The root node is at Level 0, its children are at Level 1, those node's children are at Level 2, and so on. A node at any level is considered the root of a *subtree*, which consists of that root node's children, its children's children, and so on. We can define the *depth* of a tree as the number of layers in the tree.

Finally, each node in a tree has a value. This value is sometimes referred to as the *key* value.

Binary Trees

A binary tree is defined as a tree where each node can have no more than two children. By limiting the number of children to two, we can write efficient programs for inserting data, deleting data, and searching for data in a binary tree.

Before we discuss building a binary tree in VB.NET, we need to add two terms to our tree lexicon. The child nodes of a parent node are referred to as the *left* node and the *right* node. For certain binary tree implementations, certain data values can only be stored in left nodes and other data values must be stored in right nodes. An example binary tree is shown in Figure 12.3.

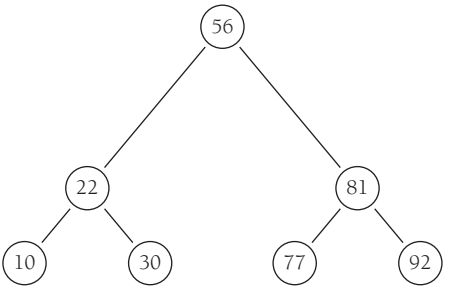


Figure 12.3. A Binary Tree.

Identifying the child nodes is important when we consider a more specific type of binary tree—the *binary search tree*. A binary search tree is a binary tree in which data with lesser values are stored in left nodes and data with greater values are stored in right nodes. This property provides for very efficient searches, as we shall soon see.

Building a Binary Search Tree

A binary search tree is made up of nodes, so we need a Node class that is similar to the Node class we used in the linked-list implementation. Let's look at the code for the Node class first:

```
Public Class Node

  Public iData As Integer
  Public Left As Node
  Public Right As Node

  Public Sub displayNode()
    Console.Write(iData)
  End Sub

End Class
```

We include Public data members for the data stored in the node and for each child node. The displayNode method allows us to display the data stored in a node. This particular Node class holds integers, but we could adapt the class easily to hold any type of data, or even declare iData of Object type if we need to.

Next we're ready to build a BinarySearchTree (BST) class. The class consists of just one data member—a Node object that represents the root node of the BST. The default constructor method for the class sets the root node to Nothing, creating an empty node.

We next need an Insert method to add new nodes to our tree. This method is somewhat complex and will require some explanation. The first step in the method is to create a Node object and assign the data held by the Node to the iData variable. This value is passed in as the only argument to the method.

The second insertion step is to check whether our BST has a root node. If it does not, then this is a new BST and the node we are inserting is the root

node and the method is finished. Otherwise, the method moves on to the next step.

If the node being added is not the root node, then we have to prepare to traverse the BST to find the proper insertion point. This process is similar to traversing a linked list. We need a Node object that we can assign to the current node as we move from level to level. We also need to position ourselves inside the BST at the root node.

Once we're inside the BST, the next step is to determine where to put the new node. This is performed inside a While loop that we break once we've found the correct position for the new node. The algorithm for determining the proper position for a node is as follows:

1. Set the parent node to be the current node, which is the root node.
2. If the data value in the new node is less than the data value in the current node, set the current node to be the left child of the current node. If the data value in the new node is greater than the data value in the current node, skip to Step 4.
3. If the value of the left child of the current node is Nothing (null), insert the new node here and exit the loop. Otherwise, skip to the next iteration of the While loop.
4. Set the current node to the right-child node of the current node.
5. If the value of the right child of the current node is Nothing (null), insert the new node here and exit the loop. Otherwise, skip to the next iteration of the While loop.

The code for the Insert method, along with the rest of the code for the BinarySearchTree class and the Node class is as follows:

```
Public Class Node
  Public iData As Integer
  Public Left As Node
  Public Right As Node

  Public Sub displayNode()
    Console.Write(iData & " ")
  End Sub

End Class

Public Class BinarySearchTree

  Public root As Node
```

```
    Public Sub New()
      root = Nothing
    End Sub

    Public Sub Insert(ByVal i As Integer)
      Dim newNode As New Node()
      newNode.iData = i
      If (root Is Nothing) Then
        root = newNode
      Else
        Dim current As Node = root
        Dim parent As Node
        While (True)
          parent = current
          If (i < current.iData) Then
            current = current.Left
            If (current Is Nothing) Then
              parent.Left = newNode
              Exit While
            End If
          Else
            current = current.Right
            If (current Is Nothing) Then
              parent.Right = newNode
              Exit While
            End If
          End If
        End While
      End If
    End Sub

  End Class
```

Traversing a Binary Search Tree

We now have the basics to implement the BinarySearchTree class, but all we can do so far is insert nodes into the BST. We need to be able to traverse the BST so that we can visit the different nodes in several different orders.

There are three traversal methods used with BSTs: *inorder*, *preorder*, and *postorder*. An inorder traversal visits all the nodes in a BST in ascending order

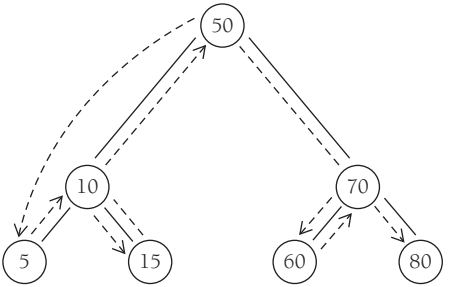


FIGURE 12.4. Inorder Traversal Order.

of the node key values. A preorder traversal visits the root node first, followed by the nodes in the subtrees under the left child of the root, followed by the nodes in the subtrees under the right child of the root. Although it's easy to understand why we would want to perform an inorder traversal, it is less obvious why we need preorder and postorder traversals. We'll show the code for all three traversals now and explain their uses in a later section.

An inorder traversal can best be written as a recursive procedure. Since the method visits each node in ascending order, the method must visit both the left node and the right node of each subtree, following the subtrees under the left child of the root before following the subtrees under the right side of the root. Figure 12.4 diagrams the path of an inorder traversal.

Here's the code for an inorder traversal method:

```
Public Sub inOrder(ByVal theRoot As Node)

  If (Not (theRoot Is Nothing))Then
    inOrder(theRoot.Left)
    theRoot.displayNode()
    inOrder(theRoot.Right)
  End If

End Sub
```

To demonstrate how this method works, let's examine a program that inserts a series of numbers into a BST. Then we'll call the inOrder method to display the numbers we've placed in the BST. Here's the code:

```
Sub Main()

  Dim nums As New BinarySearchTree()
  nums.Insert(23)
```

```
    nums.Insert(45)
    nums.Insert(16)
    nums.Insert(37)
    nums.Insert(3)
    nums.Insert(99)
    nums.Insert(22)
    Console.WriteLine("Inorder traversal: ")
    nums.inOrder(nums.root)

  End Sub
```

Here's the output:

```
  Inorder traversal:
  3 16 22 23 37 45 99
```

This list represents the contents of the BST in ascending numerical order, which is exactly what an inorder traversal is supposed to do.

Figure 12.5 illustrates the BST and the path the inorder traversal follows.

Now let's examine the code for a preorder traversal:

```
  Public Sub preOrder(ByVal theRoot As Node)

    If (Not (theRoot Is Nothing)) Then
      theRoot.displayNode()
      preOrder(theRoot.Left)
      preOrder(theRoot.Right)
    End If

  End Sub
```

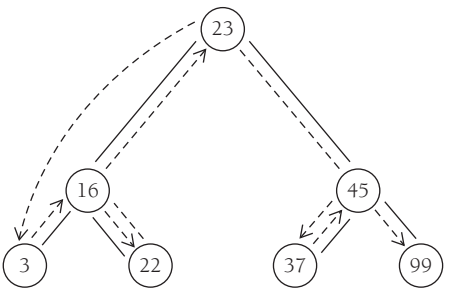


FIGURE 12.5. Inorder Traversal Path.

Notice that the only difference between the preOrder method and the inOrder method is where the three lines of code are placed. The call to the displayNode method was sandwiched between the two recursive calls in the inOrder method and it is the first line of the preOrder method.

If we replace the call to inOrder with a call to preOrder in the previous sample program, we get the following output:

```
Preorder traversal:
23 16 3 22 45 37 99
```

Finally, we can write a method for performing postorder traversals:

```
Public Sub postOrder(ByVal theRoot As Node)

  If (Not (theRoot Is Nothing)) Then
    postOrder(theRoot.Left)
    postOrder(theRoot.Right)
    theRoot.displayNode()
  End If

End Sub
```

Again, the difference between this method and the other two traversal methods is where the recursive calls and the call to displayNode are placed. In a postorder traversal, the method first recurses over the left subtrees and then over the right subtrees. Here's the output from the postOrder method:

```
Postorder traversal:
3 22 16 37 99 45 23
```

We'll look at some practical programming examples using BSTs that use these traversal methods later in this chapter.

Finding a Node and Minimum and Maximum Values in a Binary Search Tree

Three of the easiest things to do with BSTs are find a particular value, find the minimum value, and find the maximum value. We examine these operations in this section.

The code for finding the minimum and maximum values is almost trivial in both cases, owing to the properties of a BST. The smallest value in a BST will always be found at the last left-child node of a subtree beginning with the left child of the root node. In contrast, the largest value in a BST is found at the last right-child node of a subtree beginning with the right child of the root node.

We provide the code for finding the minimum value first:

```
Public Function FindMin() As Integer

  Dim current As Node = root
  While (Not (current.Left Is Nothing))
    current = current.Left
  End While
  Return current.iData

End Function
```

The method starts by creating a Node object and setting it to the root node of the BST. The method then tests to see whether the value in the Left child is Nothing. If a non-Nothing node exists in the Left child, the program sets the current node to that node. This continues until a node is found whose Left child is equal to Nothing. This means there is no smaller value below, and the minimum value has been found.

Now here's the code for finding the maximum value in a BST:

```
Public Function FindMax() As Integer

  Dim current As Node = root
  While (Not (current.Right Is Nothing))
    current = current.Right
  End While
  Return current.iData

End Function
```

This method looks almost identical to the FindMin() method, except the method moves through the right children of the BST instead of the left children.

The last method we'll look at here is the Find method, which is used to determine whether a specified value is stored in the BST. The method first

creates a Node object and sets it to the root node of the BST. Next it tests to see whether the key (the data we're searching for) is in that node. If it is, the method simply returns the current node and exits. If the data are not found in the root node, the data we're searching for get compared to the data stored in the current node. If the key is less than the current data value, the current node is set to the left child. If the key is greater than the current data value, the current node is set to the right child. The last segment of the method will return Nothing as the return value of the method if the current node is null (Nothing), indicating the end of the BST has been reached without finding the key. When the While loop ends, the value stored in current is the value being searched for.

Here's the code for the Find method:

```
Public Function Find(ByVal key As Integer) As Node

  Dim current As Node = root
  While (current.iData <> key)
    If (key < current.iData) Then
      current = current.Left
    Else
      current = current.Right
    End If
    If (current Is Nothing) Then
      Return Nothing
    End If
  End While
  Return current

End Function
```

Removing a Leaf Node from a BST

The operations we've performed on a BST thus far have not been that complicated, at least in comparison with the operation we explore in this section—removal. For some cases, removing a node from a BST is almost trivial; for other cases, it is quite involved and demands that we pay special care to the code we write, or otherwise we run the risk of destroying the correct hierarchical order of the BST.

Let's start our examination of removing a node from a BST by discussing the simplest case: removing a leaf. Removing a leaf is the simplest case since

there are no child nodes to take into consideration. All we have to do is set each child node of the target node's parent to Nothing. Of course, the node will still be there, but there will not be any references to the node.

The following code fragment shows how to delete a leaf node (and also includes the beginning of the Delete method, which declares some data members and moves to the node to be deleted):

```
Public Function Delete(ByVal key As Integer) As Boolean

  Dim current As Node = root
  Dim parent As Node = root
  Dim isLeftChild As Boolean = True

  While (current.iData <> key)
      parent = current
      If (key < current.iData) Then
        isLeftChild = True
        current = current.Right
      Else
        isLeftChild = False
        current = current.Right
      End If
      If (current Is Nothing) Then
        Return False
      End If
  End While
  If (current.Left Is Nothing And current.Right Is _
        Nothing) Then
      If (current Is root) Then
        root = Nothing
      ElseIf (isLeftChild) Then
        parent.Left = Nothing
      Else
        parent.Right = Nothing
  End If

  'More code to follow

End Function
```

The While loop takes us to the node we're deleting. The first test is to check whether the left child and the right child of that node are set to Nothing. Then we test to see whether this node is the root node. If it is, we set it to Nothing; otherwise, we either set the left node of the parent to Nothing (if isLeftChild is True) or we set the right node of the parent to Nothing.

Deleting a Node with One Child

When the node to be deleted has one child, there are four conditions we have to check for: 1. The node's child can be a left child; 2. the node's child can be a right child; 3. the node to be deleted can be a left child; or 4. the node to be deleted can be a right child.

Here's the code fragment:

```
ElseIf (current.Right Is Nothing) Then
  If (current Is root) Then
      root = current.Left
  ElseIf (isLeftChild) Then
      parent.Left = current.Left
  Else
      parent.Right = current.Right
  End If
ElseIf (current.Left Is Nothing) Then
  If (current Is root) Then
      root = current.Right
  ElseIf (isLeftChild) Then
      parent.Left = parent.Right
  Else
      parent.Right = current.Right
  End If
```

First we test to see whether the right node is set to Nothing. If it is, then we test to see whether we're at the root. If we are, we move the left child to the root node. Otherwise, if the node is a left child, we set the new parent left node to the current left node, or if we're at a right child, we set the parent right node to the current right node.

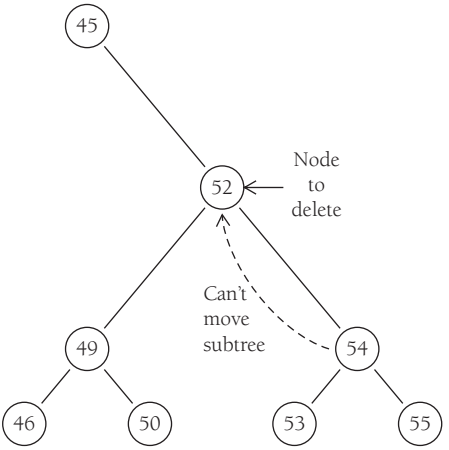


FIGURE 12.6. Deleting a node with two children.

Deleting a Node with Two Children

Deletion now gets tricky when we have to delete a node with two children. Why? Look at Figure 12.6. If we need to delete the node marked 52, what do we do to rebuild the tree? We can't replace it with the subtree starting at the node marked 54 because 54 already has a left child.

The solution to this problem is to move the inorder successor into the place of the deleted node. This works fine unless the successor itself has children, but there is a way around that scenario also. Figure 12.7 diagrams how using the inorder successor works.

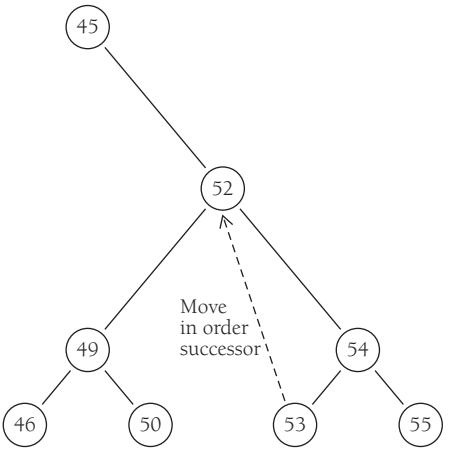


FIGURE 12.7. Moving the Inorder successor.

To find the successor, go to the original node's right child. This node has to be larger than the original node by definition. Then it begins following the left-child paths until it runs out of nodes. Since the smallest value in a subtree (like a tree) must be at the end of the path of left-child nodes, following this path to the end will leave us with the smallest node that is larger than the original node.

Here's the code for finding the successor to a deleted node:

```
Public Function getSuccessor(ByVal delNode As Node) _
    As Node
  Dim successorParent As Node = delNode
  Dim successor As Node = delNode
  Dim current As Node = delNode.Right
  While (Not (current Is Nothing))
      successorParent = successor
      successor = current
      current = current.Left
  End While
  If (Not (successor Is delNode.Right)) Then
      successorParent.Left = successor.Right
      successor.Right = delNode.Right
  End If
  Return successor
End Function
```

Now we need to look at two special cases, one in which the successor is the right child of the node to be deleted and another in which the successor is the left child of the node to be deleted. Let's start with the former.

First, the node to be deleted is marked as the current node. Remove this node from the right child of its parent node and assign it to point to the successor node. Then, remove the current node's left child and assign to it the left-child node of the successor node. Here's the code fragment for this operation:

```
Else
  Dim successor As Node = getSuccessor(current)
  If (current Is root) Then
      root = successor
  ElseIf (isLeftChild) Then
      parent.Left = successor
```

```
    Else
        parent.Right = successor
    End If
    successor.Left = current.Left
End If
```

Now let's look at the situation when the successor is the left child of the node to be deleted. The algorithm for performing this operation is as follows:

1. Assign the right child of the successor to the successor's parent left-child node.
2. Assign the right child of the node to be deleted to the right child of the successor node.
3. Remove the current node from the right child of its parent node and assign it to point to the successor node.
4. Remove the current node's left child from the current node and assign it to the left-child node of the successor node.

Part of this algorithm is carried out in the GetSuccessor method and part of it is carried out in the Delete method. The code fragment from the GetSuccessor method is as follows:

```
If (Not (successor Is delNode.Right)) Then
  successorParent.Left = successor.Right
  successor.Right = delNode.Right
End If
```

The code from the Delete method is

```
If (current Is root) Then
  root = successor
ElseIf (isLeftChild) Then
  parent.Left = successor
Else
  parent.Right = successor
End If
successor.Left = current.Left
```

This completes the code for the Delete method. Because this code is somewhat complicated, some binary search tree implementations simply mark nodes for deletion and include code to check for the marks when performing searches and traversals.

Here's the complete code for Delete:

```
Public Function Delete(ByVal key As Integer) As Boolean
  Dim current As Node = root
  Dim parent As Node = root
  Dim isLeftChild As Boolean = True

  While (current.iData <> key)
    parent = current
    If (key < current.iData) Then
      isLeftChild = True
      current = current.Right
    Else
      isLeftChild = False
      current = current.Right
    End If
    If (current Is Nothing) Then
      Return False
    End If
  End While
  If (current.Left Is Nothing And current.Right _
      Is Nothing) Then
    If (current Is root) Then
        root = Nothing
    ElseIf (isLeftChild) Then
        parent.Left = Nothing
    Else
        parent.Right = Nothing
    End If
  ElseIf (current.Right Is Nothing) Then
    If (current Is root) Then
      root = current.Left
    ElseIf (isLeftChild) Then
      parent.Left = current.Left
    Else
```

```
        parent.Right = current.Right
      End If
    ElseIf (current.Left Is Nothing) Then
      If (current Is root) Then
          root = current.Right
      ElseIf (isLeftChild) Then
          parent.Left = parent.Right
      Else
          parent.Right = current.Right
      End If
    Else
      Dim successor As Node = getSuccessor(current)
      If (current Is root) Then
          root = successor
      ElseIf (isLeftChild) Then
          parent.Left = successor
      Else
          parent.Right = successor
      End If
          successor.Left = current.Left
    End If
    Return True
  End Function
```

SUMMARY

Binary search trees are a special type of data structure called a tree. A tree is a collection of nodes (objects that consist of fields for data and links to other nodes) that are connected to other nodes. A binary tree is a specialized tree structure in which each node can have only two child nodes. A binary search tree is a specialization of the binary tree that follows the condition that lesser values are stored in left-child nodes and greater values are stored in right-child nodes.

Algorithms for finding the minimum and maximum values in a binary search tree are very easy to write. We can also simply define algorithms for traversing binary search trees in different orders (inorder, preorder, or postorder). These definitions make use of recursion, keeping the number of lines of code to a minimum while complicating their analysis.

Binary search trees are most useful when the data stored in the structure are obtained in a random order. If the data in the tree are obtained in sorted or close-to-sorted order the tree will be unbalanced and the search algorithms will not work as well.

Exercises

1. Write a program that generates 10,000 random integers in the range of 0–9 and stores them in a binary search tree. Using one of the algorithms discussed in this chapter, display a list of each of the integers and the number of times each appears in the tree.
2. Add a function to the BinarySearchTree class that counts the number of edges in a tree.
3. Rewrite Exercise 1 so that it stores the words from a text file. Display all the words in the file and the number of times they occur in the file.
4. An arithmetic expression can be stored in a binary search tree. Modify the BinarySearchTree class so that an expression such as $2 + 3 * 4/5$ can be properly evaluated using the correct operator precedence rules.

CHAPTER 13

Sets

A set is a collection of unique elements. The elements of a set are called *members*. The two most important properties of sets are that the members of a set are unordered and that no member can occur in a set more than once. Sets play a very important role in computer science but are not included as a data structure in VB.NET.

This chapter discusses the development of a Set class. Rather than provide just one implementation, however, we provide two. For nonnumeric items, we provide a fairly simple implementation using a hash table as the underlying data structureThe problem with this implementation is its efficiency. A more efficient Set class for numeric values utilizes a bit array as its data store. This forms the basis of our second implementation.

FUNDAMENTAL SET DEFINITIONS, OPERATIONS, AND PROPERTIES

A set is defined as an unordered collection of related members in which no member occurs more than once. A set is written as a list of members surrounded by curly braces, such as {0,1,2,3,4,5,6,7,8,9}. We can write a set in any order, so the previous set can be written as {9,8,7,6,5,4,3,2,1,0} or any other combination of the members such that all members are written just once.

Set Definitions

Here are some definitions you need to know to work with sets:

1. A set that contains no members is called the *empty set*. The *universe* is the set of all possible members.
2. Two sets are considered *equal* if they contain exactly the same members.
3. A set is considered a *subset* of another set if all the members of the first set are contained in the second set.

Set Operations

The fundamental operations performed on sets are the following:

1. *Union*: A new set is obtained by combining the members of one set with the members of a second set.
2. *Intersection*: A new set is obtained by adding all the members of one set that also exist in a second set.
3. *Difference*: A new set is obtained by adding all the members of one set except those that also exist in a second set.

Set Properties

The following properties are defined for sets.

1. The intersection of a set with the empty set is the empty set. The union of a set with the empty set is the original set.
2. The intersection of a set with itself is the original set. The union of a set with itself is the original set.
3. Intersection and union are *commutative*. In other words, set1 intersection set2 is equal to set2 intersect set1, and the same is true for the union of the two sets.
4. Intersection and union are *associative*. set1 intersection (set2 intersection set3) is equal to (set1 intersection set2) intersection s3. The same is true for the union of multiple sets.
5. The intersection of a set with the union of two other sets is *distributive*. In other words, set1 intersection (set2 union set3) is equal to (set1 intersection

set2) union (set1 intersection set3). This also works for the union of a set with the intersection of two other sets.

6. The intersection of a set with the union of itself and another set yields the original set. This is also true for the union of a set with the intersection of itself and another set. This is called the *absorption law*.
7. The following equalities exist when the difference of the union or intersection of two sets is taken from another set:
   - set1 difference (set2 union set3) equals (set1 difference set2) intersection (set1 difference set3)
   - set1 difference (set2 intersection set3) equals (set1 difference set2) union (set1 difference set3)

   These equalities are known as *DeMorgan's Laws*.

A Set Class Implementation Using a Hash Table

Our first Set class implementation will use a hash table to store the members of the set. The Hashtable class is one of the more efficient data structures in the .NET Framework library and it should be your choice for most class implementations when speed is important. We will call our class CSet since Set is a reserved word in VB.NET.

Class Data Members and Constructor Method

We only need one data member and one constructor method for our CSet class. The data member is a hash table and the constructor method instantiates the hash table. Here's the code:

```
Public Class CSet

  Private data As Hashtable

  Public Sub New()
    data = New Hashtable
  End Sub

  'More code to follow

End Class
```

Add Method

To add members to a set, the Add method needs to first check to make sure the member isn't already in the set. If it is, then nothing happens. If the member isn't in the set, it is added to the hash table. Here's the code:

```
Public Sub Add(ByVal item As Object)
  If Not (data.ContainsValue(item)) Then
    data.Add(Hash(item), item)
  End If
End Sub
```

Since items must be added to a hash table as a key–value pair, we calculate a hash value by adding the ASCII value of the characters of the item being added to the set. Here's the Hash function:

```
Private Function Hash(ByVal item As Object) As String
  Dim index, hashValue As Integer
  Dim s As String = CStr(item)
  For index = 0 To s.Length - 1
    hashValue += Asc(s.Chars(index))
  Next
  Return CStr(hashValue)
End Function
```

Remove and Size Methods

We also need to be able to remove members from a set and determine the number of members (size) in a set. These are straightforward methods:

```
Public Sub Remove(ByVal item As Object)
  data.Remove(Hash(item))
End Sub

Public Function Size() As Integer
  Return data.Count
End Function
```

Union Method

The Union method combines two sets using the Union operation to form a new set. The method first builds a new set by adding all the members of the first set. Then the method checks each member of the second set to see whether it is already a member of the first set. If it is, the member is skipped over, and if not, the member is added to the new set. Here's the code:

```
Public Function Union(ByVal aSet As CSet) As CSet
  Dim tempSet As New CSet
  Dim hashObject As Object
  For Each hashObject In data.Keys
    tempSet.Add(Me.data.item(hashObject))
  Next
  For Each hashObject In aSet.data.Keys
      If (Not (Me.data.ContainsKey(hashObject))) Then
        tempSet.Add(aSet.data.Item(hashObject))
      End If
  Next
  Return tempSet
End Function
```

Intersection Method

The Intersection method loops through the keys of one set, checking to see whether that key is found in the passed-in set. If it is, the member is added to the new set; otherwise, it is skipped. Here's the code:

```
Public Function Intersection(ByVal aSet As CSet) As CSet
  Dim tempSet As New CSet
  Dim hashObject As Object
  For Each hashObject In data.Keys
    If (aSet.data.Contains(hashObject)) Then
      tempSet.Add(aSet.getValue(hashObject))
    End If
  Next
  Return tempSet
End Function
```

Subset Method

The first requirement for a set to be a subset of another set is that the first set must be smaller in size in the second set. The Subset method checks the size of the sets first, and if the first set qualifies, it then checks to see that every member of the first set is a member of the second set. The code is as follows:

```
Public Function Subset(ByVal aSet As CSet) As Boolean
  Dim hashObject As Object
  Dim key As Object
  If (Me.Size > aSet.Size) Then
    Return False
  Else
    For Each key In Me.data.Keys
      If (Not (aSet.data.Contains(key))) Then
        Return False
      End If
    Next
  End If
  Return True
End Function
```

Difference Method

We've already examined how to obtain the difference of two sets. To perform this computationally, the method loops over the keys of the first set, looking for any matches in the second set. A member is added to the new set if it exists in the first set and is not found in the second set. Here's the code:

```
Public Function Difference(ByVal aSet As CSet) As CSet
  Dim tempSet As New CSet
  Dim hashObject As Object
  For Each hashObject In data.Keys
    If (Not (aSet.data.Contains(hashObject))) Then
      tempSet.Add(data.Item(hashObject))
    End If
  Next
  Return tempSet
End Function
```

A Program to Test the CSet Implementation

Let's construct a program that tests our implementation of the CSet class by creating two sets, performing a union of the two sets, performing an intersection of the two sets, finding the subset of the two sets, and finding the difference of the two sets.

Here is the program, along with the complete implementation of the CSet class:

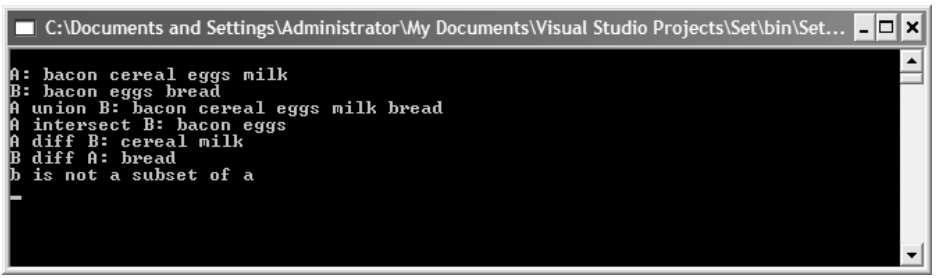
```
Module Module1
  Public Class CSet

    Private data As Hashtable
    Public Sub New()
      data = New Hashtable
    End Sub

    Public Sub Add(ByVal item As Object)
      If Not (data.ContainsValue(item)) Then
        data.Add(Hash(item), item)
      End If
    End Sub

    Public Sub Remove(ByVal item As Object)
      data.Remove(Hash(item))
    End Sub

    Public Function Size() As Integer
      Return data.Count
    End Function

    Public Function isSubset(ByVal aSet As CSet) As _
        Boolean
      Dim hashObject As Object
      Dim key As Object
      If (Me.Size > aSet.Size) Then
        Return False
      Else
        For Each key In Me.data.Keys
          If (Not (aSet.data.Contains(key))) Then
            Return False
```

```
      End If
    Next
  End If
  Return True
End Function

Private Function Hash(ByVal item As Object) As _
    String
  Dim index, hashValue As Integer
  Dim s As String = CStr(item)
  For index = 0 To s.Length - 1
    hashValue += Asc(s.Chars(index))
  Next
  Return CStr(hashValue)
End Function

Public Function Intersection(ByVal aSet As CSet) _
    As CSet
  Dim tempSet As New CSet
  Dim hashObject As Object
  For Each hashObject In data.Keys
    If (aSet.data.Contains(hashObject)) Then
      tempSet.Add(aSet.data.Item(hashObject))
    End If
  Next
  Return tempSet
End Function

Public Function Union(ByVal aSet As CSet) As CSet
  Dim tempSet As New CSet
  Dim hashObject As Object
  For Each hashObject In data.Keys
    tempSet.Add(Me.data.Item(hashObject))
  Next
  For Each hashObject In aSet.data.Keys
    If (Not (Me.data.ContainsKey(hashObject))) Then
      tempSet.Add(aSet.data.Item(hashObject))
    End If
  Next
  Return tempSet
```

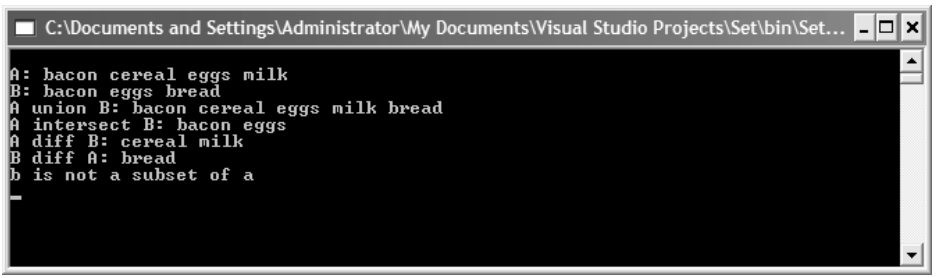
```
    End Function

    Public Function Difference(ByVal aSet As CSet) _
        As CSet
      Dim tempSet As New CSet
      Dim hashObject As Object
      For Each hashObject In data.Keys
        If (Not (aSet.data.Contains(hashObject))) Then
          tempSet.Add(data.Item(hashObject))
        End If
      Next
      Return tempSet
    End Function

    Public Overrides Function ToString() As String
      Dim str As String = ""
      Dim key As Object
      For Each key In data.Keys
        str &= data.Item(key) & " "
      Next
      Return str
    End Function

  End Class

  Sub Main()
    Dim setA As New CSet
    Dim setB As New CSet
    setA.add("milk")
    setA.add("eggs")
    setA.add("bacon")
    setA.add("cereal")
    setB.add("bacon")
    setB.add("eggs")
    setB.add("bread")
    Dim setC As New CSet
    setC = setA.Union(setB)
    Console.WriteLine()
    Console.WriteLine("A: " & setA.ToString)
    Console.WriteLine("B: " & setB.ToString)
    Console.WriteLine("A union B: " & setC.ToString)
    setC = setA.Intersection(setB)
```

```
      Console.WriteLine("A intersect B: " & setC.ToString)
      setC = setA.Difference(setB)
      Console.WriteLine("A diff B: " & setC.ToString)
      setC = setB.Difference(setA)
      Console.WriteLine("B diff A: " & setC.ToString)
      If (setB.isSubset(setA)) Then
        Console.WriteLine("b is a subset of a")
      Else
        Console.WriteLine("b is not a subset of a")
      End If
      Console.Read()
    End Sub

End Module
```

The output from this program looks like this:



If we comment out the line where "bread" is added to setB, we get the following output:



In the first example, setB could not be a subset of subA because it contained bread. Removing bread as a member makes setB a subset of subA, as shown in the second screen.

A BitArray Implementation of the CSet Class

The previous implementation of the CSet class works for objects that are not numbers, but it is still somewhat inefficient, especially for large sets. When we have to work with sets of numbers, a more efficient implementation uses the BitArray class as the data structure to store set members. The BitArray class was discussed in depth in Chapter 7.

Overview of Using a BitArray Implementation

There are several advantages to using a BitArray to store integer set members. First, because we are really only storing Boolean values, the storage space requirement is small. The second advantage is that the four main operations we want to perform on sets (union, intersection, difference, and subset) can be performed using simple Boolean operators (And, Or, and Not). The implementations of these methods are much faster than the implementations using a hash table.

The storage strategy for creating a set of integers using a BitArray is as follows: Consider adding the member 1 to the set. We simply set the array element in index position 1 to True. If we add 4 to the set, the element at position 4 is set to True, and so on.

We can determine which members are in the set by simply checking to see whether the value at that array position is set to True. We can easily remove a member from the set by setting that array position to False.

Computing the union of two sets using Boolean values is simple and efficient. Since the union of two sets is a combination of the members of both sets, we can build a new union set by Oring the corresponding elements of the two BitArrays. In other words, a member is added to the new set if the value in the corresponding position of either BitArray is True.

Computing the intersection of two sets is similar to computing the union, only for this operation we use the And operator instead of the Or operator. Similarly, the difference of two sets is found by executing the And operator with a member from the first set and the negation of the corresponding member of the second set. We can determine whether one set is a subset of another set by using the same formula we used for finding the difference. For example, if

```
setA(index) And Not (setB(index))
```

evaluates to False then setA is not a subset of setB.

The BitArray Set Implementation

We can construct the following code for a CSet class based on a BitArray:

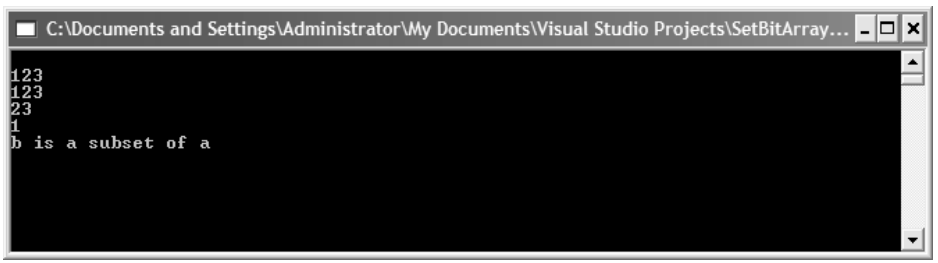
```
Public Class CSet

  Private data As BitArray

  Public Sub New()
    data = New BitArray(5)
  End Sub

  Public Sub Add(ByVal item As Integer)
    data(item) = True
  End Sub

  Public Function isMember(ByVal item As Integer) As _
      Boolean
    Return data(item)
  End Function

  Public Sub Remove(ByVal item As Integer)
    data(item) = False
  End Sub

  Public Function Union(ByVal aSet As CSet) As CSet
    Dim tempSet As New CSet
    Dim index As Integer
    For index = 0 To data.Count - 1
      tempSet.data(index) = Me.data(index) Or _
                            aSet.data(index)

    Next
    Return tempSet
  End Function

  Public Function Intersection(ByVal aSet As CSet) _
      As CSet
    Dim tempSet As New CSet
    Dim index As Integer
```

```
    For index = 0 To data.Count - 1
      tempSet.data(index) = Me.data(index) _
                            And aSet.data(index)
    Next
    Return tempSet
  End Function

  Public Function Difference(ByVal aSet As CSet) As CSet
    Dim tempSet As New CSet
    Dim index As Integer
    For index = 0 To data.Count - 1
      tempSet.data(index) = Me.data(index) And Not _
                            aSet.data(index))
    Next
    Return tempSet
  End Function

  Public Function isSubset(ByVal aSet As CSet) _
      As Boolean
    Dim tempSet As New CSet
    Dim index As Integer
    For index = 0 To data.Count - 1
      If (Me.data(index)And Not (aSet.data(index)))Then
        Return False
      End If
    Next
    Return True
  End Function

  Public Overrides Function ToString() As String
    Dim index As Integer
    Dim str As String = ""
    For index = 0 To data.Count - 1
      If (data(index)) Then
        str &= index
      End If
    Next
    Return str
  End Function

End Class
```

Here's a program to test our implementation:

```
Sub Main()
  Dim setA As New CSet
  Dim setB As New CSet
  setA.Add(1)
  setA.Add(2)
  setA.Add(3)
  setB.Add(2)
  setB.Add(3)
  Dim setC As New CSet
  setC = setA.Union(setB)
  Console.WriteLine()
  Console.WriteLine(setA.ToString)
  Console.WriteLine(setC.ToString)
  setC = setA.Intersection(setB)
  Console.WriteLine(setC.ToString)
  setC = setA.Difference(setB)
  Console.WriteLine(setC.ToString)
  Dim flag As Boolean = setB.isSubset(setA)
  If (flag) Then
    Console.WriteLine("b is a subset of a")
  Else
    Console.WriteLine("b is not a subset of a")
  End If
  Console.Read()
End Sub
```

The output from this program looks like this:



Summary

Sets and set theory provide much of the foundation of computer science theory. Whereas some languages provide a built-in set data type (Pascal), and other languages provide a set data structure via a library (Java), VB.NET provides neither a set data type nor data structure.

The chapter discussed two different implementations of a set class, one using a hash table as the underlying data store and the other implementation using a bit array as the data store. The bit array implementation is only applicable for storing integer set members, whereas the hash table implementation will store members of any data type. The bit array implementation is inherently more efficient than the hash table implementation and should be used any time you are storing integer values in a set.

Exercises

1. Create two pairs of sets using both the hash table implementation and the bit array implementation. Both implementations should use the same sets. Using the Timing class, compare the major operations (union, intersection, difference, and subset) of each implementation and report the actual difference in times.
2. Modify the hash table implementation so that it uses an ArrayList to store the set members rather than a hash table. Compare the running times of the major operations of this implementation with those of the hash table implementation. What is the difference in times?

Chapter 14

Advanced Sorting Algorithms

In this chapter we examine algorithms for sorting data that are more complex than those examined in Chapter 4. These algorithms are also more efficient, and one of them, the QuickSort algorithm, is generally considered to be the most efficient sort to use in most situations. The other sorting algorithms we'll examine are the ShellSort, the MergeSort, and the HeapSort.

To compare these advanced sorting algorithms, we'll first discuss how each of them is implemented, and in the exercises you will use the Timing class to determine the efficiency of these algorithms

ShellSort Algorithm

The ShellSort algorithm is named after its inventor, Donald Shell. This algorithm fundamentally improves on the insertion sort. The algorithm's key concept is that it compares distant items, rather than adjacent items, as is done in the insertion sort. As the algorithm loops through the data set, the distance between each item decreases until at the end the algorithm is comparing items that are adjacent.

ShellSort sorts distant elements by using an increment sequence. The sequence must start with 1, but it can then be incremented by any amount. A

good increment to use is based on the code fragment

```
While (h <= numElements / 3)
  h = h * 3 + 1
End While
```

where numElements is the number of elements in the data set being sorted, such as an array.

For example, if the sequence number generated by this code is 4, every fourth element of the data set is sorted. Then a new sequence number is chosen, using this code:

```
h = (h - 1) / 3
```

Then the next *h* elements are sorted, and so on.

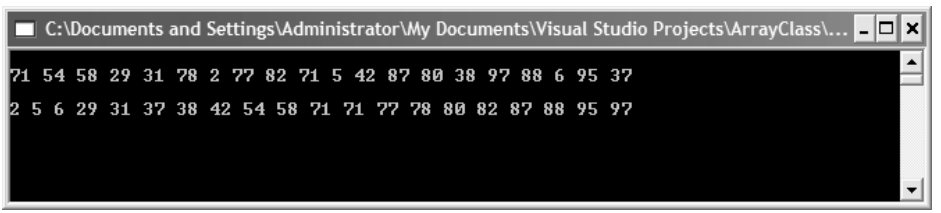
Let's look at the code for the ShellSort algorithm (using the ArrayClass code from Chapter 4):

```
Public Sub ShellSort()
  Dim inner, outer, temp As Integer
  Dim h As Integer = 1
  While (h <= numElements / 3)
    h = h * 3 + 1
  End While
  While (h > 0)
    For outer = h To numElements - 1
      temp = arr(outer)
      inner = outer
      While (inner > h - 1 AndAlso arr(inner - h) >= temp)
        arr(inner) = arr(inner - h)
        inner -= h
      End While
      arr(inner) = temp
    Next
    h = (h - 1) / 3
  End While
End Sub
```

Here's some code to test the algorithm:

```
Sub Main()
  Const SIZE As Integer = 19
  Dim theArray As New CArray(SIZE)
  Dim index As Integer
  For index = 0 To SIZE
    theArray.Insert(Int(100 * Rnd() + 1))
  Next
  Console.WriteLine()
  theArray.showArray()
  Console.WriteLine()
  theArray.ShellSort()
  theArray.showArray()
  Console.Read()
End Sub
```

The output from this code looks like this:



The ShellSort is often considered a good advanced sorting algorithm to use because of its fairly easy implementation, but its performance is acceptable even for data sets in the tens of thousands of elements.

MergeSort Algorithm

The MergeSort algorithm exemplifies a recursive algorithm. This algorithm works by breaking up the data set into two halves and recursively sorting each half. When the two halves are sorted, they are brought together using a merge routine.

The easy work comes when sorting the data set. Let's say we have the following data in the set: 71 54 58 29 31 78 2 77. First, the data set is broken

up into two separate sets: 71 54 58 29 and 31 78 2 77. Then each half is sorted to give 29 54 58 71 and 2 31 77 78. Then the two sets are merged, resulting in 2 29 31 54 58 71 77 78. The merge process compares the first two elements of each data set (stored in temporary arrays), copying the smaller value to yet another array. The element not added to the third array is then compared to the next element in the other array. The smaller element is added to the third array, and this process continues until both arrays run out of data.

But what if one array runs out of data before the other? Because this is likely to happen, the algorithm makes provisions for this situation. The algorithm uses two extra loops that are called only if one or the other of the two arrays still has data in it after the main loop finishes.

Now let's look at the code for performing a MergeSort. The first two methods are the MergeSort and the recMergeSort methods. The first method simply launches the recursive subroutine recMergeSort, which performs the sorting of the array. Here is our code:

```
Public Sub mergeSort()
  Dim tempArray(numElements) As Integer
  recMergeSort(tempArray, 0, numElements - 1)
End Sub

Public Sub recMergeSort(ByVal tempArray() As Integer, _
                        ByVal lbound As Integer, ByVal _
                        ubound As Integer)
  If (lbound = ubound) Then
    Return
  Else
    Dim mid As Integer = (lbound + ubound) \ 2
    recMergeSort(tempArray, lbound, mid)
    recMergeSort(tempArray, mid + 1, ubound)
    merge(tempArray, lbound, mid + 1, ubound)
  End If
End Sub
```

In recMergeSort, the first If statement is the base case of the recursion, returning to the calling program when the condition becomes True. Otherwise, the middle point of the array is found and the routine is called recursively on the bottom half of the array (the first call to recMergeSort) and then on

the top half of the array (the second call to recMergeSort). Finally, the entire array is merged by calling the merge method.

Here is the code for the merge method:

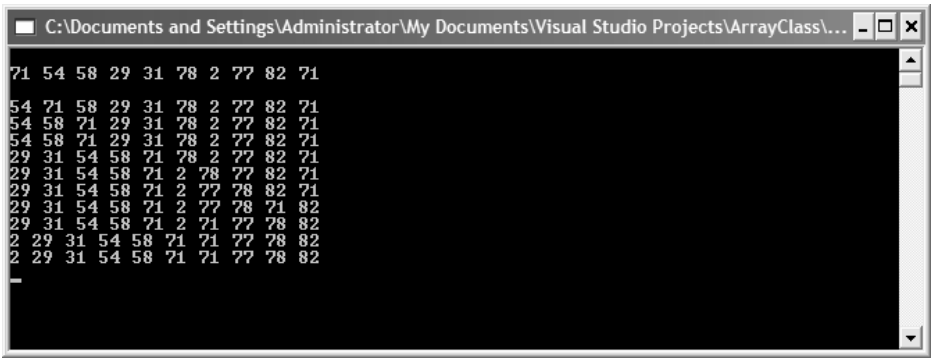
```
Public Sub merge(ByVal tempArray() As Integer, ByVal _
                 lowp As Integer, ByVal highp As _
                 Integer, ByVal ubound As Integer)
  Dim j As Integer = 0
  Dim lbound As Integer = lowp
  Dim mid As Integer = highp - 1
  Dim n As Integer = ubound - lbound + 1
  While (lowp <= mid And highp <= ubound)
    If (arr(lowp) < arr(highp)) Then
      tempArray(j) = arr(lowp)
      j += 1
      lowp += 1
    Else
      tempArray(j) = arr(highp)
      j += 1
      highp += 1
    End If
  End While
  While (lowp <= mid)
    tempArray(j) = arr(lowp)
    j += 1
    lowp += 1
  End While
  While (highp <= ubound)
    tempArray(j) = arr(highp)
    j += 1
    highp += 1
  End While
  For j = 0 To n - 1
    arr(lbound + j) = tempArray(j)
  Next
End Sub
```

This method is called each time the recMergeSort subroutines perform a preliminary sort. To demonstrate better how this method works along with

recMergeSort, let's add one line of code to the end of the merge method:

```
Me.showArray()
```

With this one line, we can view the array in its different temporary states before it is completely sorted. Here's the output:



The first line shows the array in the original state. The second line shows the beginning of the lower half being sorted. By the fifth line, the lower half is completely sorted. The sixth line shows that the upper half of the array is beginning to be sorted and the ninth line shows that both halves are completed sorted. The tenth line is the output from the final merge and the eleventh line is just another call to the showArray method.

HeapSort Algorithm

The HeapSort algorithm makes use of a data structure called a *heap*. A heap is similar to a binary tree, but with some important differences. The HeapSort algorithm, although not the fastest algorithm in this chapter, has some attractive features that encourage its use in certain situations.

Building a Heap

Unlike binary trees, heaps are usually built using arrays rather than using node references. There are two very important conditions for a heap: 1. A heap must be complete, meaning that each row must be filled in, and 2. each

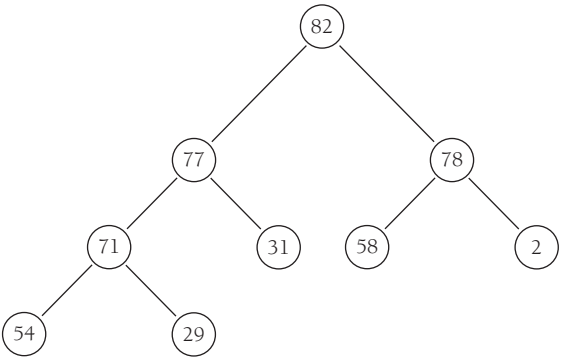


FIGURE 14.1. A Heap.

node contains data that are greater than or equal to the data in the child nodes below it. An example of a heap is shown in Figure 14.1. The array that stores the heap is shown in Figure 14.2.

The data we store in a heap are built from a Node class, similar to the nodes we've used in other chapters. This particular Node class, however, will hold just one piece of data, its primary, or key, value. We don't need any references to other nodes but we prefer using a class for the data so that we can easily change the data type of the data being stored in the heap if we need to. Here's the code for the Node class:

```
Class Node

  Public data As Integer

  Public Sub New(ByVal key As Integer)
    data = key
  End Sub

End Class
```

Heaps are built by inserting nodes into the heap array. A new node is always placed at the end of the array in an empty array element. However, doing this

82	77	78	71	31	58	2	54	29
0	1	2	3	4	5	6	7	8

FIGURE 14.2. An Array For Storing the Heap in Figure 14-1.

will probably break the heap condition because the new node's data value may be greater than some of the nodes above it. To restore the array to the proper heap condition, we must shift the new node up until it reaches its proper place in the array. We do this with a method called ShiftUp. Here's the code:

```
Public Sub ShiftUp(ByVal index As Integer)
  Dim parent As Integer = (index - 1) / 2
  Dim bottom As Node = heapArray(index)
  While (index > 0 And heapArray(parent).data < _
         bottom.data)
    heapArray(index) = heapArray(parent)
    index = parent
    parent = (parent - 1) / 2
  End While
  heapArray(index) = bottom
End Sub
```

And here's the code for the Insert method:

```
Public Function Insert(ByVal key As Integer) As Boolean
  If (currSize = maxSize) Then
    Return False
  End If
  Dim newNode As New Node(key)
  heapArray(currSize) = newNode
  ShiftUp(currSize)
  currSize += 1
  Return True
End Function
```

The new node is added at the end of the array. This immediately breaks the heap condition, so the new node's correct position in the array is found by the ShiftUp method. The argument to this method is the index of the new node. The parent of this node is computed in the first line of the method. The new node is then saved in a Node variable, bottom. The While loop then finds the correct spot for the new node. The last line then copies the new node from its temporary location in bottom to its correct position in the array.

Removing a node from a heap always means removing the node with highest value. This is easy to do because the maximum value is always in the root node. The problem is that once the root node is removed, the heap is incomplete and must be reorganized. To make the heap complete again we use the following algorithm:

1. Remove the node at the root.
2. Move the node in the last position to the root.
3. Trickle the last node down until it is below a larger node and above a smaller node.

Applying this algorithm continually removes the data from the heap in sorted order. Here is the code for the Remove and TrickleDown methods:

```
Public Function Remove() As Node
  Dim root As Node = heapArray(0)
  currSize -= 1
  heapArray(0) = heapArray(currSize)
  ShiftDown(0)
  Return root
End Function

Public Sub ShiftDown(ByVal index As Integer)
  Dim largerChild As Integer
  Dim top As Node = heapArray(index)
  While (index < (currSize \ 2))
    Dim leftChild As Integer = 2 * index + 1
    Dim rightChild As Integer = leftChild + 1
    If (rightChild <currSize And heapArray(leftChild)._
        data < heapArray(rightChild).data) Then
    largerChild = rightChild
    Else
      largerChild = leftChild
    End If
    If (top.data >= heapArray(largerChild).data) Then
      Exit While
    End If
    heapArray(index) = heapArray(largerChild)
```

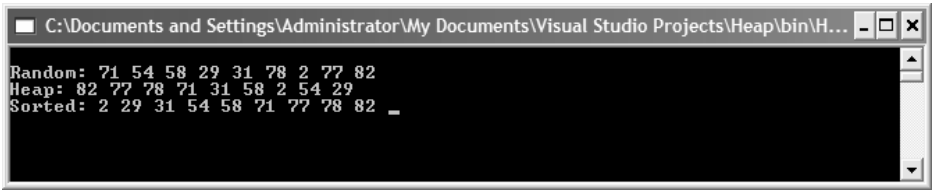
```
      index = largerChild
    End While
    heapArray(index) = top
  End Sub
```

We now have all we need to perform a heap sort, so let's look at a program that builds a heap and then sorts it:

```
  Sub Main()
    Const SIZE As Integer = 9
    Dim aHeap As New Heap(SIZE)
    Dim sortedHeap(SIZE) As Node
    Dim index As Integer
    For index = 0 To SIZE - 1
      Dim aNode As New Node(Int(100 * Rnd() + 1))
      aHeap.InsertAt(index, aNode)
      aHeap.incSize()
    Next
    Console.Write("Random: ")
    aHeap.showArray()
    Console.WriteLine()
    Console.Write("Heap: ")
    For index = SIZE \ 2 - 1 To 0 Step -1
      aHeap.ShiftDown(index)
    Next
    aHeap.showArray()
    For index = SIZE - 1 To 0 Step -1
      Dim bigNode As Node = aHeap.Remove()
      aHeap.InsertAt(index, bigNode)
    Next
    Console.WriteLine()
    Console.Write("Sorted: ")
    aHeap.showArray()
    Console.Read()
  End Sub
```

The first For loop begins the process of building the heap by inserting random numbers into the heap. The second loop heapifies the heap,

and the third For loop then uses the Remove method and the TrickleDown method to rebuild the heap in sorted order. Here's the output from the program:



HeapSort is the second fastest of the advanced sorting algorithms we examine in this chapter. Only the QuickSort algorithm, which we discuss in the next section, is faster.

QuickSort Algorithm

QuickSort has a reputation, deservedly earned, as the fastest algorithm of the advanced algorithms we're discussing in this chapter. This is true only for large, mostly unsorted data sets. If the data set is small (100 elements or less), or if the data are relatively sorted, you should use one of the fundamental algorithms discussed in Chapter 4.

Description of the QuickSort Algorithm

To understand how the QuickSort algorithm works, imagine you are a teacher and you have to alphabetize a stack of student papers. You will pick a letter from the middle of the alphabet, such as M, putting student papers whose name starts with A through M in one stack and those whose names start with N through Z in another stack. Then you split the A–M stack into two stacks and the N–Z stack into two stacks using the same technique. Then you do the same thing again until you have a set of small stacks (A–C, D–F, . . . , X–Z) of two or three elements that sort easily. Once the small stacks are sorted, you simply put all the stacks together and you have a set of sorted papers.

As you should have noticed, this process is recursive, since each stack is divided into smaller and smaller stacks. Once a stack is broken down into one element, that stack cannot be further divided and the recursion stops.

How do we decide where to split the array into two halves? There are many choices, but we'll start by just picking the first array element:

```
mv = arr(first)
```

Once that choice is made, we next have to get the array elements into the proper "half" of the array (keeping in mind that it is entirely possible that the two halves will not be equal, depending on the splitting point). We accomplish this by creating two variables, first and last, storing the second element in first and the last element in last. We also create another variable, theFirst, to store the first element in the array. The array name is arr for the sake of this example.

Figure 14.3 describes how the QuickSort algorithm works.

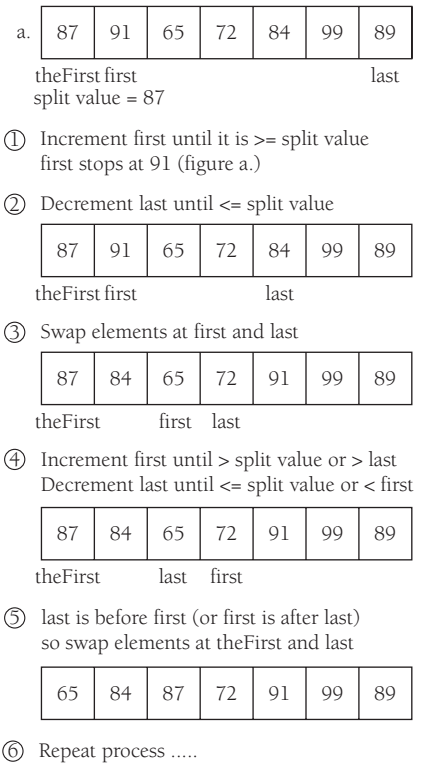


FIGURE 14.3. The Splitting an Array.

Code for the QuickSort Algorithm

Now that we've reviewed how the algorithm works, let's see how it's coded in VB.NET:

```
Public Sub QSort()
  recQSort(0, numElements - 1)
End Sub

Public Sub recQSort(ByVal first As Integer, ByVal last _
                    As Integer)
  If ((last - first) <= 0) Then
    Return
  Else
    Dim pivot As Integer = arr(last)
    Dim part As Integer = Me.Partition(first, last)
    recQSort(first, part - 1)
    recQSort(part + 1, last)
  End If
End Sub

Public Function Partition(ByVal first As Integer, _
      ByVal last As Integer) As Integer
  Dim pivotVal As Integer = arr(first)
  Dim theFirst As Integer = first
  Dim okSide As Boolean
  first += 1
  Do
    okSide = True
    While (okSide)
      If (arr(first) > pivotVal) Then
        okSide = False
      Else
        first += 1
        okSide = (first <= last)
      End If
    End While
    okSide = (first <= last)
    While (okSide)
```

```
        If (arr(last) <= pivotVal) Then
          okSide = False
        Else
          last -= 1
          okSide = (first <= last)
        End If
      End While
      If (first < last) Then
        Swap(first, last)
        Me.ShowArray()
        first += 1
        last -= 1
      End If
    Loop While (first <= last)
    Swap(theFirst, last)
    Me.ShowArray()
    Return last
  End Function

  Public Sub Swap(ByVal item1 As Integer, ByVal item2 _
                  As Integer)
    Dim temp As Integer = arr(item1)
    arr(item1) = arr(item2)
    arr(item2) = temp
  End Sub
```

An Improvement to the QuickSort Algorithm

If the data in the array are random, then picking the first value as the "pivot" or "partition" value is perfectly acceptable. Otherwise, however, making this choice will inhibit the performance of the algorithm.

A popular method for picking this value is to determine the median value in the array. You can do this by taking the upper bound of the array and dividing it by 2, for example using the following code:

```
theFirst = arr(arr.GetUpperBound(0) / 2)
```

Studies have shown that using this strategy can reduce the running time of the algorithm by about 5% (see Weiss 1999, p. 243).

Summary

The algorithms discussed in this chapter are all quite a bit faster than the fundamental sorting algorithms discussed in Chapter 4, but it is universally accepted that the QuickSort algorithm is the fastest sorting algorithm and should be used for most sorting scenarios. The Sort method that is built into several of the .NET Framework library classes is implemented using QuickSort, which explains the dominance of QuickSort over other sorting algorithms.

Exercises

1. Write a program that compares all four advanced sorting algorithms discussed in this chapter. To perform the tests, create a randomly generated array of 1,000 elements. What is the ranking of the algorithms? What happens when you increase the array size to 10,000 elements and then to 100,000 elements?
2. Using a small array (less than 20 elements), compare the sorting times between the insertion sort and QuickSort. What is the difference in time? Can you explain why this difference occurs?

CHAPTER 15

Advanced Data Structures and Algorithms for Searching

In this chapter, we present a set of advanced data structures and algorithms for performing searching. The data structures we cover include the red–black tree, the splay tree, and the skip list. AVL trees and red–black trees are two solutions to the problem of handling unbalanced binary search trees. The skip list is an alternative to using a treelike data structure that foregoes the complexity of the red–black and splay trees.

AVL Trees

Named for the two computer scientists who discovered this data structure—G. M. Adelson-Velskii and E. M. Landis—in 1962, AVL trees provide another solution to maintaining balanced binary trees. The defining characteristic of an AVL tree is that the difference between the height of the right and left subtrees can never be more than one.

AVL Tree Fundamentals

To guarantee that the tree always stays "in balance," the AVL tree continually compares the heights of the left and right subtrees. AVL trees utilize a technique, called a rotation, to keep them in balance.

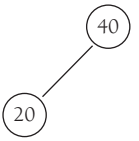


FIGURE 15.1.

To understand how a rotation works, let's look at a simple example that builds a binary tree of integers. Starting with the tree shown in Figure 15.1, if we insert the value 10 into the tree, the tree becomes unbalanced, as shown in Figure 15.2. The left subtree now has a height of 2, but the right subtree has a height of 0, violating the rule for AVL trees. The tree is balanced by performing a *single right rotation*, moving the value 40 down to the right, as shown in Figure 15.3.

Now look at the tree in Figure 15.4. If we insert the value 30 we get the tree in Figure 15.5. This tree is unbalanced. We fix it by performing what is called a *double rotation*, moving 40 down to the right and 30 up to the right, resulting in the tree shown in Figure 15.6.

AVL Tree Implementation

Our AVL tree implementation consists of two classes: 1. a Node class used to hold data for each node in the tree and 2. the AVLTree class, which contains the methods for inserting nodes and rotating nodes.

The Node class for an AVL tree implementation is built similarly to nodes for a binary tree implementation, but with some important differences. Each node in an AVL tree must contain data about its height, so a data member for height is included in the class. We also have the class implement the IComparable interface to compare the values stored in the nodes. Also, because the height of a node is so important, we include a ReadOnly property method to return a node's height.

Here is the code for the Node class:

```
Public Class Node
  Implements IComparable

  Public element As Object
  Public left As Node
  Public right As Node
```

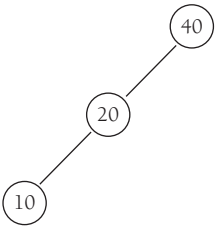


FIGURE 15.2.

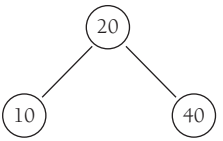


FIGURE 15.3.

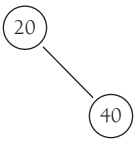


FIGURE 15.4.

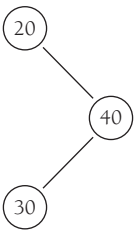


FIGURE 15.5.

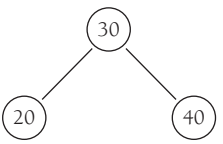


FIGURE 15.6.

```
    Public height As Integer

    Public Sub New(ByVal data As Object, ByVal lt As _
                   Node, ByVal rt As Node)
      element = data
      left = lt
      right = rt
      height = 0
    End Sub

    Public Sub New(ByVal data As Object)
      element = data
      left = Nothing
      right = Nothing
    End Sub

    Public Function CompareTo(ByVal obj As Object) As _
          Integer Implements System.IComparable.CompareTo
      Return (Me.element.compareto(obj.element))
    End Function

    Public ReadOnly Property getHeight() As Integer
      Get
        If (Me Is Nothing) Then
          Return -1
        Else
          Return Me.height
        End If
      End Get
    End Property

End Class
```

The first method in the AVLTree class we examine is the Insert method. This method determines where to place a node in the tree. The method is recursive, either moving left when the current node is greater than the node to be inserted or moving right when the current node is less than the node to be inserted.

Once the node is in its place, the difference in height of the two subtrees is calculated. If the tree is determined to be unbalanced, a left or right rotation or a double left or double right rotation is performed. Here's the

code (with the code for the different rotation methods shown after the Insert method):

```
Private Function Insert(ByVal item As Object, _
                        ByVal n As Node) As Node
  If (n Is Nothing) Then
    n = New Node(item, Nothing, Nothing)
  ElseIf (item.compareto(n.element) < 0) Then
    n.left = Insert(item, n.left)
    If height(n.left) - height(n.right) = 2 Then
      If (item.CompareTo(n.left.element) < 0) Then
        n = rotateWithLeftChild(n)
      Else
        n = doubleWithLeftChild(n)
      End If
    End If
  ElseIf (item.compareto(n.element) > 0) Then
    n.right = Insert(item, n.right)
    If (height(n.right) - height(n.left) = 2) Then
      If (item.compareto(n.right.element) > 0) Then
        n = rotateWithRightChild(n)
      Else
        n = doubleWithRightChild(n)
      End If
    End If
  Else
    'do nothing, duplicate value
  End If
  n.height = Math.Max(height(n.left), height(n.right)) + 1
  Return n
End Function
```

The different rotation methods are as follows:

```
Private Function rotateWithLeftChild (ByVal n2 As _
                                      Node) As Node
  Dim n1 As Node = n2.left
  n2.left = n1.right
  n1.right = n2
  n2.height = Math.Max(height(n2.left), _
                       height(n2.right)) + 1
```

```
    n1.height = Math.Max(height(n1.left), n2.height) + 1
    Return n1
  End Function

  Private Function rotateWithRightChild (ByVal n1 As _
                                         Node) As Node
    Dim n2 As Node = n1.right
    n1.right = n2.left
    n2.left = n1
    n1.height = Math.Max(height(n1.left), _
                         height(n1.right)) + 1
    n2.height = Math.Max(height(n2.right), n1.height) + 1
    Return n2
  End Function

  Private Function doubleWithLeftChild (ByVal n3 As _
                                        Node) As Node
    n3.left = rotateWithRightChild(n3.left)
    Return rotateWithLeftChild(n3)
  End Function

  Private Function doubleWithRightChild (ByVal n1 As _
                                         Node) As Node
    n1.right = rotateWithLeftChild(n1.right)
    Return rotateWithRightChild(n1)
  End Function
```

There are many other methods we can implement for this class (e.g., the methods from the BinarySearchTree class). We leave the implementation of these other methods to the exercises. Also, we have purposely not implemented a deletion method for the AVLTree class. Many AVL tree implementations use *lazy deletion*. This system of deletion marks a node for deletion but doesn't actually delete the node from the tree. The performance cost of deleting nodes and then rebalancing the tree is often prohibitive. You will get a chance to experiment with lazy deletion in the exercises.

Red–Black Trees

AVL trees are not the only solution to dealing with an unbalanced binary search tree. Another data structure you can use is the *red–black tree*. A red–black tree is one in which the nodes of the tree are designated as either red

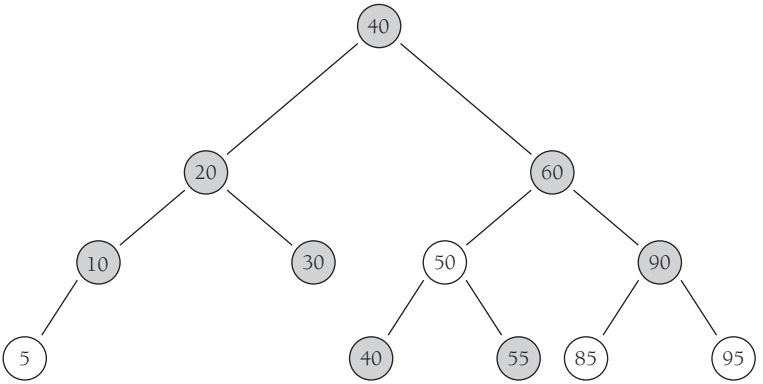


FIGURE 15.7. A Red–Black Tree.

or black, depending on a set of rules. By properly coloring the nodes in the tree, the tree stays nearly perfectly balanced. Figure 15.7 shows an example of a red–black tree (with black nodes shaded):

Red–Black Tree Rules

The following rules are used when working with red–black trees:

1. Every node in the tree is colored either red or black.
2. The root node is colored black.
3. If a node is red, the children of that node must be black.
4. Each path from a node to a Nothing reference must contain the same number of black nodes.

As a consequence of these rules, a red–black tree stays in very good balance, which means searching a red–black tree is quite efficient. As with AVL trees, though, these rules also make insertion and deletion more difficult.

Red–Black Tree Insertion

Inserting a new item into a red–black tree is complicated because it can lead to a violation of one of the aforementioned rules. For example, look at the red–black tree in Figure 15.8. We can insert a new item into the tree as a black node. If we do so, we are violating rule 4. So the node must be colored red. If the parent node is black, everything is fine. If the parent node is red, however, then rule 3 is violated. We have to adjust the tree either by having nodes change color or by rotating nodes as we did with AVL trees.

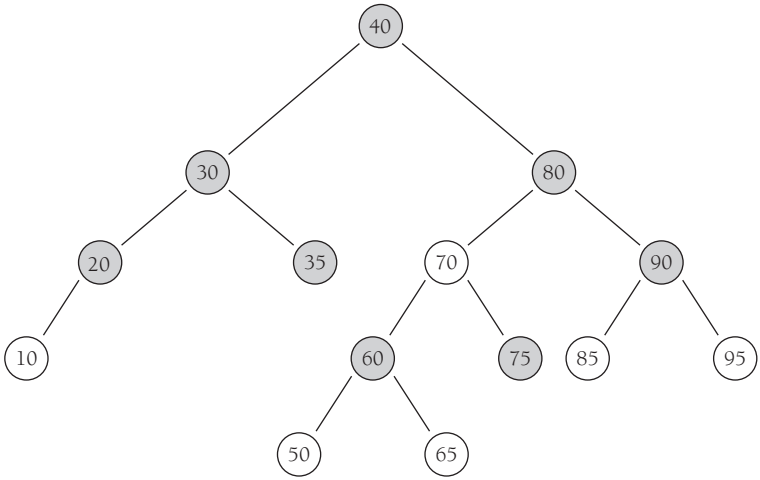


FIGURE 15.8.

We can make this process more concrete by looking at a specific example. Let's say we want to insert the value 55 into the tree shown in Figure 15.8. As we work our way down the tree, we notice that the value 60 is black and has two red children. We can change the color of each of these nodes (60 to red, 50 and 65 to black), then rotate 60 to 80's position, and then perform other rotations to put the subtree back in order. We are left with the red–black tree shown in Figure 15.9. This tree now follows all the red–black tree rules and is well balanced.

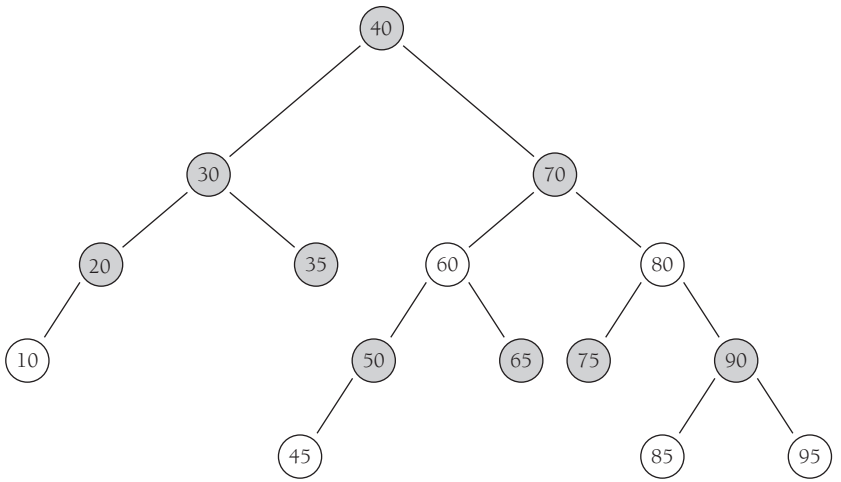


FIGURE 15.9.

Red–Black Tree Implementation Code

Rather than break up the code with explanations, we show the complete code for a red–black tree implementation in one piece, with a description of the code to follow. We start with the Node class and continue with the RedBlack class. Here is the code:

```
Module Module1

  Public Class Node
    Public element As String
    Public left As Node
    Public right As Node
    Public color As Integer
    Const RED As Integer = 0
    Const BLACK As Byte = 1

    Public Sub New(ByVal element As String, ByVal left _
                   As Node, ByVal right As Node)
      Me.element = element
      Me.left = left
      Me.right = right
      Me.color = BLACK
    End Sub

    Public Sub New(ByVal element As String)
      Me.element = element
      Me.left = Nothing
      Me.right = Nothing
    End Sub

  End Class

  Public Class RBTree
    Const Red As Integer = 0
    Const Black As Integer = 1
    Private current As Node
    Private parent As Node
    Private grandParent As Node
    Private greatParent As Node
    Private header As Node
```

```
Private nullNode As Node

Public Sub New(ByVal element As String)
  current = New Node("")
  parent = New Node("")
  grandParent = New Node("")
  greatParent = New Node("")
  nullNode = New Node("")
  nullNode.left = nullNode
  nullNode.right = nullNode
  header = New Node(element)
  header.left = nullNode
  header.right = nullNode
End Sub

Public Sub Insert(ByVal item As String)
  grandParent = header
  parent = grandParent
  current = parent
  nullNode.element = item
  While (current.element.CompareTo(item) <> 0)
    Dim greatParent As Node = grandParent
    grandParent = parent
    parent = current
    If (item.CompareTo(current.element) < 0) Then
      current = current.left
    Else
      current = current.right
    End If
    If (current.left.color = Red And _
        current.right.color = Red) Then
      HandleReorient(item)
    End If
  End While
  If (Not (current Is nullNode)) Then
    Return
  End If
  current = New Node(item, nullNode, nullNode)
  If (item.CompareTo(parent.element) < 0) Then
    parent.left = current
```

```
    Else
      parent.right = current
    End If
    HandleReorient(item)
  End Sub
  Public Function FindMin() As String
    If (Me.isEmpty()) Then
      Return Nothing
    End If
    Dim itrNode As Node = header.right
    While (Not (itrNode.left Is nullNode))
      itrNode = itrNode.left
    End While
    Return itrNode.element
  End Function

  Public Function FindMax() As String
    If (Me.isEmpty()) Then
      Return Nothing
    End If
    Dim itrNode As Node = header.right
    While (Not (itrNode.right Is nullNode))
      itrNode = itrNode.right
    End While
    Return itrNode.element
  End Function

  Public Function Find(ByVal e As String) As String
    nullNode.element = e
    Dim current As Node = header.right
    While (True)
      If (e.CompareTo(current.element) < 0) Then
        current = current.left
      ElseIf (e.CompareTo(current.element) > 0) Then
        current = current.right
      ElseIf (Not (current Is nullNode)) Then
        Return current.element
      Else
        Return Nothing
      End If
    End While
```

```
End Function

Public Sub MakeEmpty()
  header.right = nullNode
End Sub

Public Function isEmpty() As Boolean
  Return (header.right Is nullNode)
End Function

Public Sub PrintRBTree()
  If (Me.isEmpty()) Then
    Console.WriteLine("Empty")
  Else
    PrintRB(header.right)
  End If
End Sub
Private Sub PrintRB(ByVal n As Node)
  If (Not n Is nullNode) Then
    PrintRB(n.left)
    Console.WriteLine(n.element)
    PrintRB(n.right)
  End If
End Sub

Public Sub HandleReorient(ByVal item As String)
  current.color = Red
  current.left.color = Black
  current.right.color = Black
  If (parent.color = Red) Then
    grandParent.color = Red
    If ((item.CompareTo(grandParent.element) < 0) __
         <> (item.CompareTo(parent.element))) Then
      current = rotate(item, grandParent)
      current.color = Black
    End If
    header.right.color = Black
  End If
End Sub

Private Function rotate(ByVal item As String, _
                        ByVal parent As Node) As Node
```

```
      If (item.CompareTo(parent.element) < 0) Then
        If (item.CompareTo(parent.left.element) _
            < 0) Then
          parent.left = rotateWithLeftChild _
                        (parent.left)
        Else
          parent.left = rotateWithRightChild _
                        (parent.left)
        End If
        Return parent.left
      Else
        If (item.CompareTo(parent.right.element) _
            < 0) Then
          parent.right = rotateWithLeftChild _
                         (parent.right)
        Else
          parent.right = rotateWithRightChild _
                         (parent.right)
        End If
        Return parent.right
      End If
    End Function

    Public Function rotateWithLeftChild (ByVal k2 As _
                                         Node) As Node
      Dim k1 As Node = k2.left
      k2.left = k1.right
      k1.right = k2
      Return k1
    End Function

    Public Function rotateWithRightChild (ByVal k1 As _
                                          Node) As Node
      Dim k2 As Node = k1.right
      k1.right = k2.left
      k2.left = k1
      Return k2
    End Function

  End Class

End Module
```

The handleReorient method is called whenever a node has two red children. The rotate methods are similar to those used with AVL trees. Also, because dealing with the root node constitutes a special case, the RedBlack class includes a root sentinel node as well as the nullNode node, which indicates the node is a reference to Nothing.

Skip Lists

Although AVL trees and red–black trees are efficient data structures for searching and sorting data, the rebalancing operations necessary with both data structures to keep the tree balanced introduce considerable overhead and complexity. There is another data structure we can use, especially for searching, that provides the efficiency of trees without the worries of rebalancing. This data structure is called a skip list.

Skip List Fundamentals

Skip lists are built from one of the fundamental data structures for searching—the linked list. As we know, linked lists perform well for insertion and deletion but are not so good at searching, since we have to travel to each node sequentially. However, there is no reason why we have to travel each link successively. When we want to go from the bottom of a set of stairs to the top and we want to get there quickly, what do we do? We take the stairs two or three at a time (or more if we're blessed with long legs).

We can implement the same strategy in a linked list by creating different levels of links. We start with level 0 links; these point to the next node in the list. Then we have a level 1 link, which points to the second node in the list, skipping one node; a level 2 link, which points to the third node in the list, skipping two nodes; and so on. When we search for an item, we can start at a high link level and traverse the list until we get to a value that is greater than the value we're looking for. We can then back up to the previous visited node, and move down to the lowest level, searching node by node until we encounter the searched-for value. To illustrate the difference between a skip list and a linked list, study the diagrams in Figures 15.10 and 15.11.

Let's look at how a search is performed on the level 1 skip list shown in Figure 15.11. The first value we'll search for is 1133. Looking at the basic linked list first, we see that we have to travel to four nodes to find 1133. Using

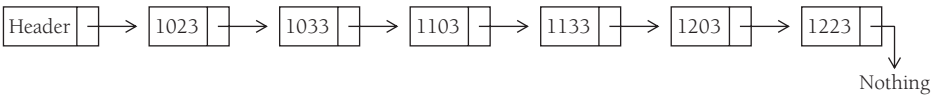


FIGURE 15.10. Basic Linked List.

a skip list, though, we only have to travel to two nodes. Clearly, using the skip list is more efficient for such a search.

Now let's look at how a search for 1203 is performed with the skip list. The level 1 links are traversed until the value 1223 is found. This is greater than 1203 so we back up to the node storing the value 1133 and drop down one level and start using level 0 links. The next node is 1203 so the search ends. This example makes the skip list search strategy clear. Start at the highest link level and traverse the list using those links until you reach a value greater than the value you're searching for. At that point, back up to the last node visited and move down to the next link level and repeat the same steps. Eventually, you will reach the link level that leads you to the searched-for value.

We can make the skip list even more efficient by adding more links. For example, every fourth node can have a link that points four nodes ahead, every sixth node can have a link that points six nodes ahead, and so on. The problem with this scheme is that when we insert or delete a node, we have to rearrange a tremendous number of node pointers, making our skip list much less efficient.

The solution to this problem is to allocate nodes to the link levels randomly. The first node (after the header) might be a level 2 node, whereas the second node might be a level 4 node, the third node a level 1 node again, and so on. Distributing link levels randomly makes the other operations (other than search) more efficient, and it doesn't really affect search times. The probability distribution used to determine how to distribute nodes randomly is based on the fact that about 50% of the nodes in a skip list will be level 0 nodes, 25% of the nodes will be level 1 nodes, 12.5% will be level 2 nodes, 5.75% will be level 3 nodes, and so on.

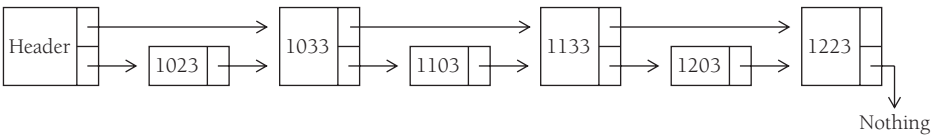


FIGURE 15.11. Skip list with links 2 nodes ahead (level 1).

All that's left to explain is how we determine the number of levels to use in the skip list. The inventor of the skip list, William Pugh, a professor of Computer Science currently at the University of Maryland, worked out a formula in his paper that first described skip lists (available at ftp://ftp.cs.umd.edu/pub/skipLists/). Here it is, expressed in VB.NET code:

```
CInt(Math.Ceiling(Math.Log(maxNodes) / Math. _
                  Log(1 / PROB)) - 1)
```

In this code maxNodes is an approximation of the number of nodes that will be required and PROB is a probability constant, usually 0.25.

Skip List Implementation

We need two classes for a skip list implementation: a class for nodes and a class for the skip list itself. Let's start with the class for nodes.

The nodes we'll use for this implementation will store a key and a value, as well as an array for storing pointers to other nodes. Here's the code:

```
Public Class SkipNode

  Public key As Integer
  Public value As Object
  Public link() As SkipNode

  Public Sub New(ByVal level As Integer, ByVal key _
                 As Integer, ByVal value As Object)
    Me.key = key
    Me.value = value
    ReDim link(level)
  End Sub

End Class
```

Now we're ready to build the skip list class. The first thing we need to do is determine which data members we need for the class. Here's a list of what we'll need:

- **maxLevel:** stores the maximum number of levels allowed in the skip list,
- **level:** stores the current level,

- **header**: identifies the beginning node that provides entry into the skip list,
- **probability**: stores the current probability distribution for the link levels,
- **NIL**: stores a special value that indicates the end of the skip list, and
- **PROB**: stores the probability distribution for the link levels.

Our code then takes the following form:

```
Public Class SkipList

  Private maxLevel As Integer
  Private level As Integer
  Private header As SkipNode
  Private probability As Single
  Private Const NIL As Integer = Int32.MaxValue
  Private Const PROB As Single = 0.5
```

The constructor for the SkipList class is written in two parts. The first is a Public constructor with a single argument passing in the total number of nodes in the skip list. The second part comprises a Private constructor that does most of the work. Let's view the methods first before explaining how they work:

```
Private Sub New(ByVal probable As Single, ByVal _
                maxLevel As Integer)
  Me.probability = probable
  Me.maxLevel = maxLevel
  level = 0
  header = New SkipNode(maxLevel, 0, Nothing)
  Dim nilElement As SkipNode = New SkipNode _
                               (maxLevel, NIL, Nothing)
  Dim i As Integer
  For i = 0 To maxLevel - 1
    header.link(i) = nilElement
  Next
End Sub

Public Sub New(ByVal maxNodes As Long)
```

```
    Me.New(PROB, CInt(Math.Ceiling(Math.Log(maxNodes) / _
                      Math.Log(1 / PROB)) - 1))
End Sub
```

The Public constructor performs two tasks. First, the node total is passed into the constructor method as the only parameter in the method. Second, a call is made to the Private constructor, where the real work of initializing a skip list object is performed. This call has two arguments: the probability constant and a formula for determining the maximum number of link levels for the skip list, both of which we've already discussed.

The body of the Private constructor sets the values of the data members, creates a header node for the skip list, creates a "dummy" node for each of the header's links, and then initializes the links to that element.

The first thing we do with a skip list is insert nodes into the list. Here's the code for the Insert method of the SkipList class:

```
Public Sub Insert(ByVal key As Integer, ByVal value _
                  As Object)
  Dim update(maxLevel) As SkipNode
  Dim cursor As SkipNode = header
  Dim i As Integer
  For i = level To 0 Step -1
    While (cursor.link(i).key < key)
      cursor = cursor.link(i)
    End While
    update(i) = cursor
  Next
  cursor = cursor.link(0)
  If (cursor.key = key) Then
    cursor.value = value
  Else
    Dim newLevel As Integer = genRandomLevel()
    If (newLevel > level) Then
      For i = level + 1 To newLevel - 1
        update(i) = header
      Next
      level = newLevel
    End If
```

```
      cursor = New SkipNode(newLevel, key, value)
      For i = 0 To newLevel - 1
        cursor.link(i) = update(i).link(i)
        update(i).link(i) = cursor
      Next
    End If
  End Sub
```

The method first determines where in the list to insert the new SkipNode (the first For loop). Next, the list is checked to see whether the value to insert is already there. If it is not, then the new SkipNode is assigned a random link level using the Private method genRandomLevel (which we'll discuss next) and the item is inserted into the list (the line before the last For loop). Link levels are determined using the probabilistic method genRandomLevel. Here's the code:

```
  Private Function genRandomLevel() As Integer
    Dim ran As New Random
    Dim newLevel As Integer = 0
    While (newLevel < maxLevel And _
           CInt(Int((1 * Rnd()) + 0)) < probability)
      newLevel += 1
    End While
    Return newLevel
  End Function
```

Before we cover the Search method, which is the focus of this section, let's look at how to perform deletion in a skip list. First, let's view the code for the Delete method:

```
  Public Sub Delete(ByVal key As Integer)
    Dim update(maxLevel + 1) As SkipNode
    Dim cursor As SkipNode = header
    Dim i As Integer
    For i = level To 0 Step -1
      While (cursor.link(i).key < key)
        cursor = cursor.link(i)
      End While
      update(i) = cursor
```

```
    Next
    cursor = cursor.link(0)
    If (cursor.key = key) Then
      For i = 0 To level - 1
        If (update(i).link(i) Is cursor) Then
          update(i).link(i) = cursor.link(i)
        End If
      Next
      While (level > 0 And header.link(level).key = NIL)
        level -= 1
      End While
    End If
    End Sub
  End Class
```

This method, like the Insert method, is split into two parts. The first part, highlighted by the first For loop, finds the item to be deleted in the list. The second part, highlighted by the If statement, adjusts the links around the deleted SkipNode and readjusts the levels.

Now we're ready to discuss the Search method. The method starts at the highest level, following those links until a key with a higher value than the key being searched for is found. The method then drops down to the next lowest level and continues the search until a higher key is found. It drops down again and continues searching. The method will eventually stop at level 0, exactly one node away from the item in question. Here's the code:

```
  Public Function Search(ByVal key As Integer) As Object
    Dim cursor As SkipNode = header
    Dim i As Integer
    For i = level - 1 To 0 Step -1
      Dim nextElement As SkipNode = cursor.link(i)
      While (nextElement.key < key)
        cursor = nextElement
        nextElement = cursor.link(i)
      End While
    Next
    cursor = cursor.link(0)
    If (cursor.key = key) Then
      Return cursor.value
```

```
    Else
      Return "object not found"
    End If
  End Function
```

We've now provided enough functionality to implement a SkipList class. In the exercises at the end of the chapter, you will get a chance to write code that uses the class.

Skip lists offer an alternative to tree-based structures. Most programmers find them easier to implement and their efficiency is comparable to treelike structures. If you are working with a completely or nearly sorted data set, skip lists are probably a better choice than trees.

Summary

The advanced data structures discussed in this chapter are based on the discussions in Chapter 12 of Weiss (1999). AVL trees and red–black trees offer good solutions to the balancing problems experienced when using fairly sorted data with binary search trees. The major drawback to AVL and red–black trees is that the rebalancing operations come with quite a bit of overhead and can slow down performance on large data sets.

For extremely large data sets, skip lists offer an alternative even to AVL and red–black trees. Because skip lists use a linked-list structure rather than a tree structure, rebalancing is unnecessary, making them more efficient in many situations.

Exercises

1. Write FindMin and FindMax methods for the AVLTree class.
2. Using the Timing class, compare the times for the methods implemented in Exercise 1 to the same methods in the BinarySearchTree class. Your test program should insert a sorted list of approximately 100 randomly generated integers into the two trees.
3. Write a deletion method for the AVLTree class that utilizes lazy deletion. There are several techniques you can use, but a simple one is to simply add a Boolean field to the Node class that signifies whether or not the node

is marked for deletion. Your other methods must then take this field into account.

4. Write a deletion method for the RedBlack class that adheres to the red–black rules.
5. Design and implement a program that compares AVL trees and red–black trees to skip lists. Which data structure performs the best?

Chapter 16

Graphs and Graph Algorithms

The study of networks has become one of the great scientific hotbeds of this new century, though mathematicians and others have been studying networks for many hundreds of years. Recent developments in computer technology (i.e., the Internet), and in social theory (the social network, popularly conceived in the concept of "six degrees of separation"), have put a spotlight on the study of networks.

In this chapter, we look at how networks are modeled with graphs. We're not talking about graphs such as pie graphs or bar graphs. We define what a graph is, how they're represented in VB.NET, and how to implement important graph algorithms. We also discuss the importance of picking the correct data representation when working with graphs, since the efficiency of graph algorithms depends on the data structure used.

Graph Definitions

A *graph* consists of a set of *vertices* and a set of *edges*. Think of a map of a U.S. state. Each town is connected with other towns via some type of road. A map is a type of graph. Each town is a vertex and a road that connects two towns is an edge. Edges are specified as a *pair*, (v1, v2), where v1 and v2 are two vertices in the graph. A vertex can also have a *weight*, sometimes also called a cost.

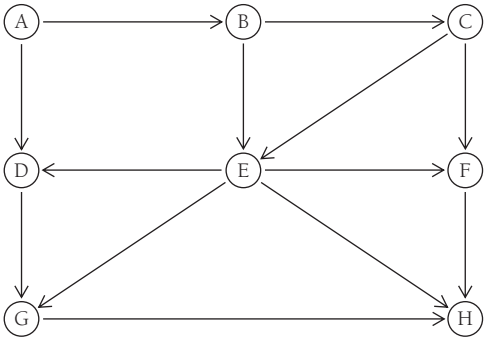


FIGURE 16.1. A Digraph (directed Graph).

A graph whose pairs are ordered is called a *directed graph*, or just a *digraph*. An ordered graph is shown in Figure 16.1. If a graph is not ordered, it is called an *unordered graph*, or just a graph. An example of an unordered graph is shown in Figure 16.2.

A *path* is a sequence of vertices in a graph such that all vertices are connected by edges. The length of a path is the number of edges from the first vertex in the path to the last vertex. A path can also consist of a vertex to itself, which is called a *loop*. Loops have a length of 0.

A *cycle* is a path of at least 1 in a directed graph so that the beginning vertex is also the ending vertex. In a directed graph, the edges can be the same edge, but in an undirected graph, the edges must be distinct.

An undirected graph is considered *connected* if there is a path from every vertex to every other vertex. In a directed graph, this condition is called *strongly connected*. A directed graph that is not strongly connected, but is considered connected, is called *weakly connected*. If a graph has a edge between every set of vertices it is said to be a *complete graph*.

REAL-WORLD SYSTEMS MODELED BY GRAPHS

Graphs are used to model many different types of real-world systems. One example is traffic flow. The vertices represent street intersections and the edges represent the streets themselves. Weighted edges can be used to represent

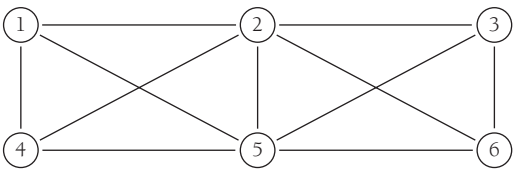


FIGURE 16.2. An Unordered Graph.

speed limits or the number of lanes. Modelers can use the system to determine the best routes and the streets most likely to suffer from traffic jams.

Any type of transportation system can be modeled using a graph. For example, an airline can model its flight system using a graph. Each airport is a vertex and each flight from one vertex to another is an edge. A weighted edge can represent the cost of a flight from one airport to another, or perhaps the distance from one airport to another, depending on what is being modeled.

The Graph Class

At first glance, a graph looks much like a tree and you might be tempted to try to build a graph class like a tree. There are problems with using a reference-based implementation, however, so we will look at a different scheme for representing both vertices and edges.

Representing Vertices

The first step we have to take to build a Graph class is to build a Vertex class to store the vertices of a graph. This class has the same duties the Node class had in the LinkedList and BinarySearchTree classes.

The Vertex class needs two data members—one for the data that identify the vertex and the other for a Boolean member we use to keep track of "visits" to the vertex. We call these data members label and wasVisited, respectively.

The only method we need for the class is a constructor method that allows us to set the label and wasVisited data members. We won't use a default constructor in this implementation because every time we make a first reference to a vertex object, we will be performing instantiation.

Here's the code for the Vertex class:

```
Public Class Vertex

  Public wasVisited As Boolean
  Public label As String

  Public Sub New(ByVal label As String)
    Me.label = label
    wasVisited = False
  End Sub

End Class
```

	V_0	V_1	V_2	V_3	V_4
V_0	0	0	1	0	0
V_1	0	0	1	0	0
V_2	1	1	0	1	1
V_3	0	0	1	0	0
V_4	0	0	1	0	0

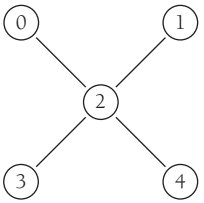


FIGURE 16.3. An Adjacency Matrix.

We will store the list of vertices in an array and will reference them in the Graph class by their position in the array.

Representing Edges

The real information about a graph is stored in the edges, since the edges detail the structure of the graph. As we mentioned earlier, it is tempting to represent a graph like a binary tree, but doing so would be a mistake. A binary tree has a fairly fixed representation, since a parent node can only have two child nodes, whereas the structure of a graph provides much more flexibility. There can be many edges linked to a single vertex or just one edge, for example.

The method we'll choose for representing the edges of a graph is called an adjacency matrix. This is a two-dimensional array in which the elements indicate whether an edge exists between two vertices. Figure 16.3 illustrates how an adjacency matrix works for the graph in the figure.

The vertices are listed as the headings for the rows and columns. If an edge exists between two vertices, a 1 is placed in that position. If an edge doesn't exist, a 0 is used. Obviously, you can also use Boolean values here.

Building a Graph

Now that we have a way to represent vertices and edges, we're ready to build a graph. First, we need to build a list of the vertices in the graph. Here is some code for a small graph that consists of four vertices:

```
Dim nVertices As Integer = 0
vertices(nVertices) = new Vertex("A")
nVertices += 1
```

```
vertices(nVertices) = new Vertex("B")
nVertices += 1
vertices(nVertices) = new Vertex("C")
nVertices += 1
vertices(nVertices) = new Vertex("D")
```

Then we need to add the edges that connect the vertices. Here is the code for adding two edges:

```
adjMatrix(0,1) = 1
adjMatrix(1,0) = 1
adjMatrix(1,3) = 1
adjMatrix(3,1) = 1
```

This code states that an edge exists between vertices A and B and that an edge exists between vertices B and D.

With these pieces in place, we're ready to look at a preliminary definition of the Graph class (along with the definition of the Vertex class):

```
Public Class Vertex

  Public wasVisited As Boolean
  Public label As String

  Public Sub New(ByVal label As String)
    Me.label = label
    wasVisited = False
  End Sub

End Class

Public Class Graph
  Private Const NUM_VERTICES As Integer = 20
  Private vertices() As Vertex
  Private adjMatrix(,) As Integer
  Private numVerts As Integer

  Public Sub New()
    ReDim vertices(NUM_VERTICES)
    ReDim adjMatrix(NUM_VERTICES, NUM_VERTICES)
    numVerts = 0
    Dim j, k As Integer
```

```
    For j = 0 To NUM_VERTICES - 1
      For k = 0 To NUM_VERTICES - 1
        adjMatrix(j, k) = 0
      Next
    Next
  End Sub

  Public Sub addVertex(ByVal label As String)
    vertices(numVerts) = New Vertex(label)
    numVerts += 1
  End Sub

  Public Sub addEdge(ByVal start As Integer, ByVal _
                     eend As Integer)
    adjMatrix(start, eend) = 1
    adjMatrix(eend, start) = 1
  End Sub

  Public Sub showVertex(ByVal v As Integer)
    Console.Write(vertices(v).label)
  End Sub

End Class
```

The constructor method redimensions the vertices array and the adjacency matrix to the number specified in the constant NUM_VERTICES. The data member numVerts stores the current number in the vertex list so that it is initially set to zero, since arrays are zero-based. Finally, the adjacency matrix is initialized by setting all elements to zero.

The addVertex method takes a string argument for a vertex label, instantiates a new Vertex object, and adds it to the vertices array. The addEdge method takes two integer values as arguments. These integers represent two vertices and indicate that an edge exists between them. Finally, the showVertex method displays the label of a specified vertex.

A First Graph Application: Topological Sorting

Topological sorting involves displaying the specific order in which a sequence of vertices must be followed in a directed graph. The sequence of courses a college student must take to obtain a degree can be modeled with a directed

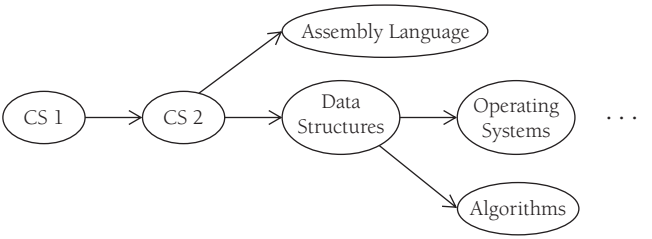


FIGURE 16.4. A Directed Graph Model of Computer Science Curriculum Sequence.

graph. For example, a student can't take the Data Structures course until he or she has taken the first two introductory Computer Science courses. Figure 16.4 depicts a directed graph modeling part of the typical Computer Science curriculum.

A topological sort of this graph would result in the following sequence:

1. CS1
2. CS2
3. Assembly Language
4. Data Structures
5. Operating Systems
6. Algorithms

Courses 3 and 4 can be taken at the same time, as can 5 and 6.

An Algorithm for Topological Sorting

The basic algorithm for topological sorting is very simple:

1. Find a vertex that has no successors.
2. Add the vertex to a list of vertices.
3. Remove the vertex from the graph.
4. Repeat Step 1 until all vertices are removed.

Of course, the challenge lies in the details of the implementation but this is the crux of topological sorting.

The algorithm will actually work from the end of the directed graph to the beginning. Look again at Figure 16.4. Assuming that Operating Systems and Algorithms are the last vertices in the graph (ignoring the ellipsis), then neither of them have successors and so they are added to the list and removed from the graph. Next come Assembly Language and Data Structures. These

vertices now have no successors, so they are removed from the list. Next will be CS2. Because its successors have been removed, it is added to the list. Finally, we're left with CS1.

Implementing the Algorithm

We need two methods for topological sorting: one to determine whether a vertex has no successors and another for removing a vertex from a graph. Let's look at the method for determining no successors first.

A vertex with no successors will be found in the adjacency matrix on a row in which all the columns are zeroes. Our method will use nested For loops to check each set of columns row by row. If a 1 is found in a column, then the inner loop is exited and the next row is tried. If a row is found with all zeroes in the columns, then that row number is returned. If both loops complete and no row number is returned, then a –1 is returned, indicating there is no vertex with no successors. Here's the code:

```
Public Function noSuccessors() As Integer
  Dim isEdge As Boolean
  Dim row, col As Integer
  For row = 0 To numVertices - 1
    isEdge = False
    For col = 0 To numVertices - 1
      If adjMatrix(row, col) > 0 Then
        isEdge = True
        Exit For
      End If
    Next
    If (Not (isEdge)) Then
      Return row
    End If
  Next
  Return -1
End Function
```

Next we need to see how to remove a vertex from the graph. The first thing we have to do is remove the vertex from the vertex list. This is easy. Then we need to remove the row and column from the adjacency matrix, followed by moving the rows and columns above and to the right of the vertex downward and to the left to fill the void left by the removed vertex.

To perform this operation, we write a method named delVertex, which includes two helper methods, moveRow and moveCol. Here is the code:

```
Public Sub delVertex(ByVal vert As Integer)
  Dim j, row, col As Integer
  If (vert <> numVertices - 1) Then
    For j = vert To numVertices - 1
      vertices(j) = vertices(j + 1)
    Next
    For row = vert To numVertices - 1
      moveRow(row, numVertices)
    Next
    For col = vert To numVertices - 1
      moveCol(row, numVertices - 1)
    Next
  End If
End Sub

Private Sub moveRow(ByVal row As Integer, ByVal _
                    length As Integer)
  Dim col As Integer
  For col = 0 To length - 1
    adjMatrix(row, col) = adjMatrix(row + 1, col)
  Next
End Sub

Private Sub moveCol(ByVal col As Integer, ByVal _
                    length As Integer)
  Dim row As Integer
  For row = 0 To length - 1
    adjMatrix(row, col) = adjMatrix(row, col + 1)
  Next
End Sub
```

Now we need a method to control the sorting process. Here's the code:

```
Public Sub TopSort()
  Dim origVerts As Integer = numVertices
  While (numVertices > 0)
    Dim currVertex As Integer = noSuccessors()
    If (currVertex = -1) Then
```

```
        Console.WriteLine("Error: graph has cycles")
        Return sub
      End If
      gStack.Push(vertices(currVertex).label)
      delVertex(currVertex)
    End While
    Console.Write("Topological sorting order: ")
    While (gStack.Count > 0)
      Console.Write(gStack.Pop & " ")
    End While
  End Sub
```

The TopSort method loops through the vertices of the graph, finding a vertex with no successors, deleting it, and then moving on to the next vertex. Each time a vertex is deleted, its label gets pushed onto a stack. A stack is a convenient data structure to use because the first vertex found is actually the last (or one of the last) vertices in the graph. When the TopSort method is complete, the contents of the stack will have the last vertex pushed down to the bottom of the stack and the first vertex of the graph at the top of the stack. We merely have to loop through the stack, popping each element to display the correct topological order of the graph.

These are all the methods we need to perform topological sorting on a directed graph. Here's a program that tests our implementation:

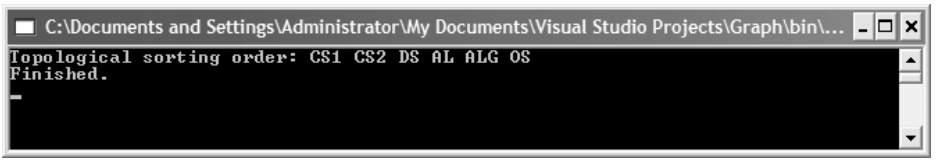
```
Sub Main()
  Dim theGraph As New Graph
  theGraph.addVertex("A")
  theGraph.addVertex("B")
  theGraph.addVertex("C")
  theGraph.addVertex("D")
  theGraph.addEdge(0, 1)
  theGraph.addEdge(1, 2)
  theGraph.addEdge(2, 3)
  theGraph.addEdge(3, 4)
  theGraph.TopSort()
  Console.WriteLine()
  Console.WriteLine("Finished.")
  Console.Read()
End Sub
```

The output from this program shows that the order of the graph is A B C D.

Now let's look at how we would write the program to sort the graph shown in Figure 16.4:

```
Sub Main()
  Dim theGraph As New Graph
  theGraph.addVertex("CS1")
  theGraph.addVertex("CS2")
  theGraph.addVertex("DS")
  theGraph.addVertex("OS")
  theGraph.addVertex("ALG")
  theGraph.addVertex("AL")
  theGraph.addEdge(0, 1)
  theGraph.addEdge(1, 2)
  theGraph.addEdge(1, 5)
  theGraph.addEdge(2, 3)
  theGraph.addEdge(2, 4)
  theGraph.TopSort()
  Console.WriteLine()
  Console.WriteLine("Finished.")
  Console.Read()
End Sub
```

The output from this program looks like this:



Searching a Graph

Determining which vertices can be reached from a specified vertex is a common activity performed on graphs. We might want to know which roads lead from one town to other towns on the map, or which flights can take us from one airport to other airports.

These operations are performed on a graph using a search algorithm. There are two fundamental searches we can perform on a graph: a *depth-first* search

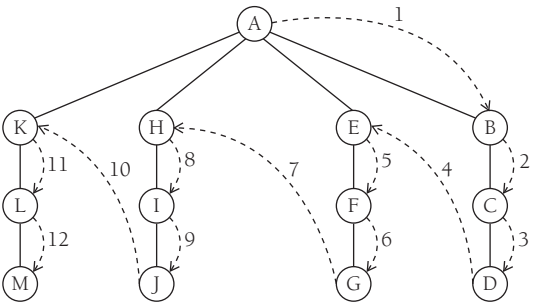


FIGURE 16.5. Depth-First Search.

and a *breadth-first* search. In this section we examine each of these search algorithms.

Depth-First Search

Depth-first search involves following a path from the beginning vertex until it reaches the last vertex, then backtracking and following the next path until it reaches the last vertex, and so on until there are no more paths left. A diagram of a depth-first search is shown in Figure 16.5.

At a high level, the depth-first search algorithm works like this: First, pick a starting point, which can be any vertex. Visit the vertex, push it onto a stack, and mark it as visited. Then you go to the next vertex that is unvisited, push it on the stack, and mark it. This continues until you reach the last vertex. Then you check to see whether the top vertex has any unvisited adjacent vertices. If it doesn't, then you pop it off the stack and check the next vertex. If you find one, you start visiting adjacent vertices until there are no more, check for more unvisited adjacent vertices, and continue the process. When you finally reach the last vertex on the stack and there are no more adjacent, unvisited vertices, you've performed a depth-first search.

The first piece of code we have to develop must give us a method for getting an unvisited, adjacent matrix. Our code must first go to the row for the specified vertex and determine whether the value 1 is stored in one of the columns. If it is, then an adjacent vertex exists. We can then easily check to see whether the vertex has been visited or not. Here's the code for this method:

```
Private Function getAdjUnvisitedVertex(ByVal v As _
    Integer) As Integer
```

```
    Dim j As Integer
    For j = 0 To numVertices - 1
      If adjMatrix(v, j) = 1 And _
         vertices(j).wasVisited = False Then
        Return j
      End If
    Next
    Return -1
  End Function
```

Now we're ready to look at the method that performs the depth-first search:

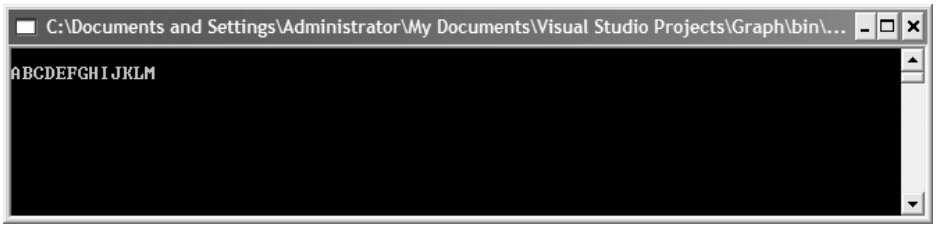
```
  Public Sub DepthFirstSearch()
    vertices(0).wasVisited = True
    showVertex(0)
    gStack.Push(0)
    Dim v As Integer
    While (gStack.Count > 0)
      v = getAdjUnvisitedVertex(gStack.Peek)
      If v = -1 Then
        gStack.Pop()
      Else
        vertices(v).wasVisited = True
        showVertex(v)
        gStack.Push(v)
      End If
    End While
    Dim j As Integer
    For j = 0 To numVertices - 1
      vertices(j).wasVisited = False
    Next
  End Sub
```

Here is a program that performs a depth-first search on the graph shown in Figure 16.5:

```
  Sub Main()
    Dim aGraph As New Graph
```

```
    aGraph.addVertex("A")
    aGraph.addVertex("B")
    aGraph.addVertex("C")
    aGraph.addVertex("D")
    aGraph.addVertex("E")
    aGraph.addVertex("F")
    aGraph.addVertex("G")
    aGraph.addVertex("H")
    aGraph.addVertex("I")
    aGraph.addVertex("J")
    aGraph.addVertex("K")
    aGraph.addVertex("L")
    aGraph.addVertex("M")
    aGraph.addEdge(0, 1)
    aGraph.addEdge(1, 2)
    aGraph.addEdge(2, 3)
    aGraph.addEdge(0, 4)
    aGraph.addEdge(4, 5)
    aGraph.addEdge(5, 6)
    aGraph.addEdge(0, 7)
    aGraph.addEdge(7, 8)
    aGraph.addEdge(8, 9)
    aGraph.addEdge(0, 10)
    aGraph.addEdge(10, 11)
    aGraph.addEdge(11, 12)
    aGraph.DepthFirstSearch()
    Console.WriteLine()
    Console.Read()
  End Sub
```

The output from this program looks like this:



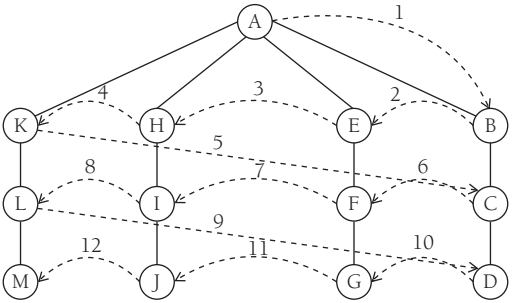


FIGURE 16.6. Breath-First Search.

Breadth-First Search

A breadth-first search starts at a first vertex and tries to visit vertices as close to the first vertex as possible. In essence, this search moves through a graph layer by layer, first examining the layers closer to the first vertex and then moving down to the layers farthest away from the starting vertex. Figure 16.6 demonstrates how breadth-first search works.

The algorithm for breadth-first search uses a queue instead of a stack, though a stack could be used. The algorithm is as follows:

1. Find an unvisited vertex that is adjacent to the current vertex, mark it as visited, and add it to a queue.
2. If an unvisited, adjacent vertex can't be found, remove a vertex from the queue (as long as there is a vertex to remove), make it the current vertex, and start over.
3. If the second step can't be performed because the queue is empty, the algorithm is finished.

Now let's look at the code for the algorithm:

```
Public Sub BreadthFirstSearch()
  Dim gQueue As New Queue
  vertices(0).wasVisited = True
  showVertex(0)
  gQueue.Enqueue(0)
  Dim vert1, vert2 As Integer
  While (gQueue.Count > 0)
    vert1 = gQueue.Dequeue()
    vert2 = getAdjUnvisitedVertex(vert1)
```

```
      While (vert2 <> -1)
        vertices(vert2).wasVisited = True
        showVertex(vert2)
        gQueue.Enqueue(vert2)
        vert2 = getAdjUnvisitedVertex(vert1)
      End While
    End While
    Dim index As Integer
    For index = 0 To numVertices - 1
      vertices(index).wasVisited = False
    Next
  End Sub
```

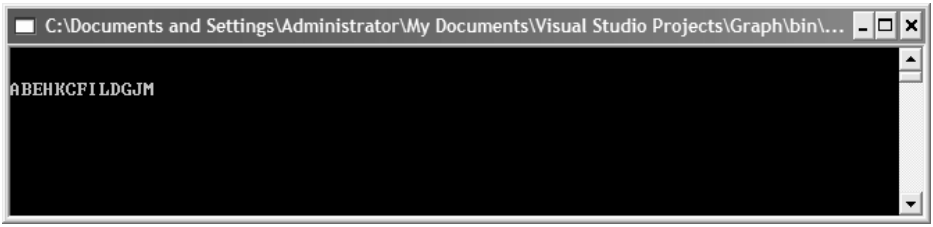
Notice that there are two loops in this method. The outer loop runs while the queue has data in it, and the inner loop checks adjacent vertices to see whether they've been visited. The For loop simply cleans up the vertices array for other methods.

A program that tests this code, using the graph from Figure 16.6, is the following:

```
Sub Main()
  Dim aGraph As New Graph
  aGraph.addVertex("A")
  aGraph.addVertex("B")
  aGraph.addVertex("C")
  aGraph.addVertex("D")
  aGraph.addVertex("E")
  aGraph.addVertex("F")
  aGraph.addVertex("G")
  aGraph.addVertex("H")
  aGraph.addVertex("I")
  aGraph.addVertex("J")
  aGraph.addVertex("K")
  aGraph.addVertex("L")
  aGraph.addVertex("M")
  aGraph.addEdge(0, 1)
  aGraph.addEdge(1, 2)
  aGraph.addEdge(2, 3)
  aGraph.addEdge(0, 4)
```

```
    aGraph.addEdge(4, 5)
    aGraph.addEdge(5, 6)
    aGraph.addEdge(0, 7)
    aGraph.addEdge(7, 8)
    aGraph.addEdge(8, 9)
    aGraph.addEdge(0, 10)
    aGraph.addEdge(10, 11)
    aGraph.addEdge(11, 12)
    Console.WriteLine()
    aGraph.BreadthFirstSearch()
    Console.Read()
  End Sub
```

The output from this program looks like this:



Minimum Spanning Trees

When a network is first designed, there can be more than the minimum number of connections between the nodes of the network. The extra connections are a wasted resource and should be eliminated, if possible. The extra connections also make the network unnecessarily complex and difficult for others to study and understand. What we want is a network that contains just the minimum number of connections necessary to connect the nodes. Such a network, when applied to a graph, is called a *minimum spanning tree*.

A minimum spanning tree is so named because it is constructed from the minimum of number of edges necessary to cover every vertex (spanning), and it is in tree form because the resulting graph is acyclic. There is one important point you need to keep in mind: One graph can contain multiple minimum spanning trees; the minimum spanning tree you create depends entirely on the starting vertex.

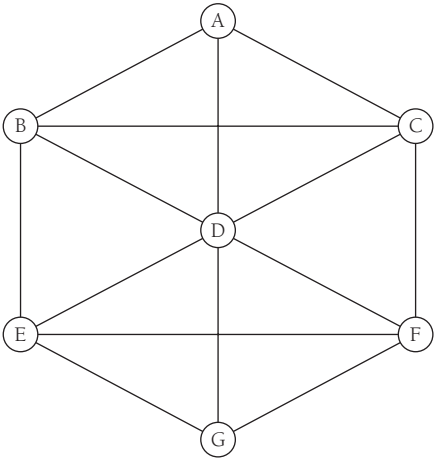


FIGURE 16.7. Graph For Minimum Spanning Tree.

A Minimum Spanning Tree Algorithm

Figure 16.7 depicts a graph for which we want to construct a minimum spanning tree.

The algorithm for a minimum spanning tree is really just a graph search algorithm (either depth-first or breadth-first) with the additional component of recording each edge that is traveled. The code also looks similar. Here's the method:

```
Public Sub mst()
  vertices(0).wasVisited = True
  gStack.Push(0)
  Dim currVertex, ver As Integer
  While (gStack.Count > 0)
    currVertex = gStack.Peek()
    ver = getAdjUnvisitedVertex(currVertex)
    If (ver = -1) Then
      gStack.Pop()
    Else
      vertices(ver).wasVisited = True
      gStack.Push(ver)
      showVertex(currVertex)
      showVertex(ver)
```

```
        Console.Write(" ")
      End If
    End While
    Dim j As Integer
    For j = 0 To numVertices - 1
      vertices(j).wasVisited = False
    Next
  End Sub
```

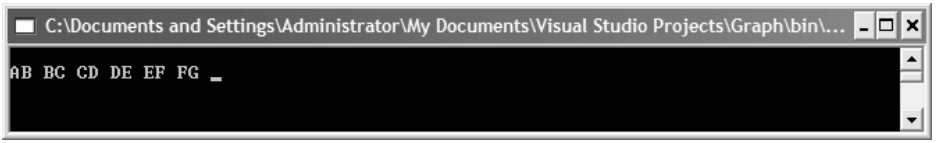
If you compare this method to the method for depth-first search, you'll see that the current vertex is recorded by calling the showVertex method with the current vertex as the argument. Calling this method twice, as shown in the code, creates the display of edges that define the minimum spanning tree.

Here is a program that creates the minimum spanning tree for the graph in Figure 16.7:

```
Sub Main()
  Dim aGraph As New Graph
  aGraph.addVertex("A")
  aGraph.addVertex("B")
  aGraph.addVertex("C")
  aGraph.addVertex("D")
  aGraph.addVertex("E")
  aGraph.addVertex("F")
  aGraph.addVertex("G")
  aGraph.addEdge(0, 1)
  aGraph.addEdge(0, 2)
  aGraph.addEdge(0, 3)
  aGraph.addEdge(1, 2)
  aGraph.addEdge(1, 3)
  aGraph.addEdge(1, 4)
  aGraph.addEdge(2, 3)
  aGraph.addEdge(2, 5)
  aGraph.addEdge(3, 5)
  aGraph.addEdge(3, 4)
  aGraph.addEdge(3, 6)
  aGraph.addEdge(4, 5)
  aGraph.addEdge(4, 6)
  aGraph.addEdge(5, 6)
```

```
    Console.WriteLine()
    aGraph.mst()
    Console.Read()
  End Sub
```

The output from this program looks like this:



A diagram of the minimum spanning tree is shown in Figure 16.8.

Finding the Shortest Path

One of the most common operations performed on graphs is finding the shortest path from one vertex to another. Consider the following example: For vacation, you are going to travel to 10 major league baseball cities to watch games over a two-week period. You want to minimize the number of miles you have to drive to visit all 10 cities using a shortest-path algorithm. Another shortest-path problem involves creating a network of computers, where the cost could be the time to transmit between two computers or the cost of establishing and maintaining the connection. A shortest-path algorithm can determine the most effective way you can build the network.

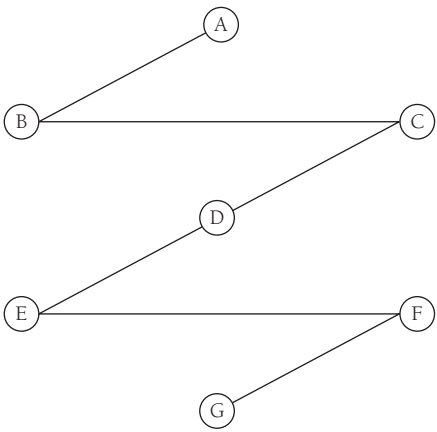


Figure 16.8. The Minimum Spanning Tree for Figure 16–7.

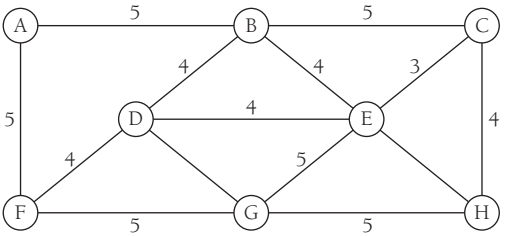


FIGURE 16.9. A Weighted Graph.

Weighted Graphs

In a weighted graph, each edge in the graph has an associated weight, or cost. A weighted graph is shown in Figure 16.9. Weighted graphs can have negative weights, but we will limit our discussion here to positive weights. We also focus here only on directed graphs.

Dijkstra's Algorithm for Determining the Shortest Path

One of the most famous algorithms in computer science is *Dijkstra's algorithm* for determining the shortest path of a weighted graph, named for the late computer scientist Edsger Dijkstra, who invented the algorithm in the late 1950s.

Dijkstra's algorithm finds the shortest path from any specified vertex to any other vertex and, it turns out, to all the other vertices in the graph. It does this by using what is commonly termed a *greedy* strategy or algorithm. A greedy algorithm (about which we'll have more to say in Chapter 17 breaks a problem into pieces, or stages, determining the best solution at each stage, with each subsolution contributing to the final solution. A classic example of a greedy algorithm is making change with coins. For example, if you buy something at the store for 74 cents using a dollar bill, the cashier, if he or she is using a greedy algorithm and wants to minimize the number of coins returned, will return to you a quarter and a penny. Of course, there are other solutions to making change for 26 cents, but a quarter and a penny is the optimal solution.

We use Dijkstra's algorithm by creating a table to store known distances from the starting vertex to the other vertices in the graph. Each vertex adjacent to the original vertex is visited, and the table is updated with information about the weight of the adjacent edge. If a distance between two vertices is known,

but a shorter distance is discovered by visiting a new vertex, that information is changed in the table. The table is also updated by indicating which vertex leads to the shortest path.

The tables that follow show us the progress the algorithm makes as it works through the graph. In these tables, the value Infinity indicates we don't know the distance; in code we use a large value that cannot represent a weight. The first table shows us the table values before vertex A is visited:

Vertex	Visited	Weight	Via Path
A	False	0	0
B	False	Infinity	n/a
C	False	Infinity	n/a
D	False	Infinity	n/a
E	False	Infinity	n/a
F	False	Infinity	n/a
G	False	Infinity	n/a

After A is visited, the table looks like this:

Vertex	Visited	Weight	Via Path
A	True	0	0
B	False	2	A
C	False	Infinity	n/a
D	False	1	A
E	False	Infinity	n/a
F	False	Infinity	n/a
G	False	Infinity	n/a

Next we visit vertex D:

Vertex	Visited	Weight	Via Path
A	True	0	0
B	False	2	A
C	False	3	D
D	True	1	A
E	False	3	D
F	False	9	D
G	False	5	D

Then vertex B is visited:

Vertex	Visited	Weight	Via Path
A	True	0	0
B	True	2	A
C	False	3	D
D	True	1	A
E	False	3	D
F	False	9	D
G	False	5	D

And we continue like this until we visit the last vertex, G:

Vertex	Visited	Weight	Via Path
A	True	0	0
B	True	2	A
C	True	3	D
D	True	1	A
E	True	3	D
F	True	6	D
G	False	5	D

Code for Dijkstra's Algorithm

The first piece of code for the algorithm is the Vertex class, which we've seen before:

```
Class Vertex
  Public label As String
  Public isInTree As Boolean

  Public Sub New(ByVal lab As String)
    label = lab
    isInTree = False
  End Sub
End Class
```

We also need a class that helps keep track of the relationship between a distant vertex and the original vertex used to compute shortest paths. This is called the DistOriginal class:

```
Class DistOriginal
  Public distance As Integer
  Public parentVert As Integer

  Public Sub New(ByVal pv As Integer, ByVal d As _
                 Integer)
```

```
    distance = d
    parentVert = pv
  End Sub
End Class
```

The Graph class that we've used before now has a new set of methods for computing shortest paths. The first of these is the Path() method, which drives the shortest path computations:

```
Public Sub Path()
  Dim startTree As Integer = 0
  vertexList(startTree).isInTree = True
  nTree = 1
  Dim j As Integer
  For j = 0 To nVerts - 1
    Dim tempDist As Integer = adjMat(startTree, j)
    sPath(j) = New DistOriginal(startTree, tempDist)
  Next
  While (nTree < nVerts)
    Dim indexMin As Integer = getMin()
    Dim minDist As Integer = sPath(indexMin).distance
    currentVert = indexMin
    startToCurrent = sPath(indexMin).distance
    vertexList(currentVert).isInTree = True
    nTree += 1
    adjustShortPath()
  End While
  displayPaths()
  nTree = 0
  For j = 0 To nVerts - 1
    vertexList(j).isInTree = False
  Next
End Sub
```

This method uses two other helper methods, getMin and adjustShortPath. Those methods are explained shortly. The For loop at the beginning of the method looks at the vertices reachable from the beginning vertex and places them in the sPath array. This array holds the minimum distances from the different vertices and will eventually hold the final shortest paths.

The main loop (the While loop) performs three operations:

1. Find the entry in sPath with the shortest distance.
2. Make this vertext the current vertex.
3. Update the sPath array to show distances from the current vertex.

Much of this work is performed by the getMin and adjustShortPath methods:

```
Public Function getMin() As Integer
  Dim minDist As Integer = infinity
  Dim indexMin As Integer = 0
  Dim j As Integer
  For j = 1 To nVerts - 1
    If (Not vertexList(j).isInTree And sPath(j). _
        distance < minDist) Then

      minDist = sPath(j).distance
      indexMin = j
    End If
  Next
  Return indexMin
End Function

Public Sub adjustShortPath()
  Dim column As Integer = 1
  While (column < nVerts)
    If (vertexList(column).isInTree) Then
      column += 1
    Else
      Dim currentToFringe As Integer = _
            adjMat (currentVert, column)
      Dim startToFringe As Integer = _
            startToCurrent + currentToFringe
      Dim sPathDist As Integer = sPath(column).distance
      If (startToFringe < sPathDist) Then
        sPath(column).parentVert = currentVert
        sPath(column).distance = startToFringe
      End If
      column += 1
```

```
      End If
    End While
  End Sub
```

The getMin method steps through the sPath array until the minimum distance is determined, which is then returned by the method. The adjustShortPath method takes a new vertex, finds the next set of vertices connected to this vertex, calculates shortest paths, and updates the sPath array until a shorter distance is found.

Finally, the displayPaths method shows the final contents of the sPath array. To make the graph available for other algorithms, the nTree variable is set to 0 and the isInTree flags are all set to False.

To put all this into context, here is a complete application that includes all the code for computing the shortest paths using Dijkstra's algorithm, along with a program to test the implementation:

```
Module Module1
  Class DistOriginal
    Public distance As Integer
    Public parentVert As Integer

    Public Sub New(ByVal pv As Integer, ByVal d _
                   As Integer)
      distance = d
      parentVert = pv
    End Sub
  End Class

  Class Vertex
    Public label As String
    Public isInTree As Boolean

    Public Sub New(ByVal lab As String)
      label = lab
      isInTree = False
    End Sub
  End Class

  Class Graph
    Private Const max_verts As Integer = 20
    Private Const infinity As Integer = 1000000
```

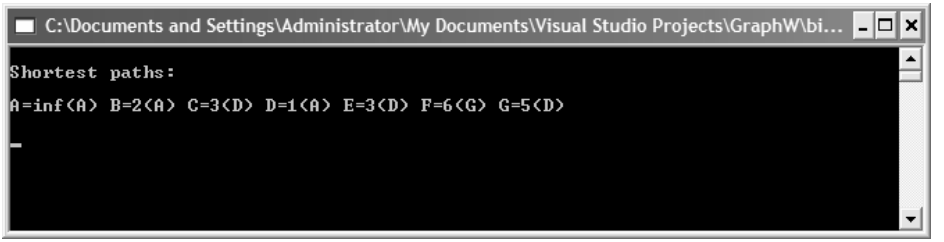
```
Private vertexList() As Vertex
Private adjMat(,) As Integer
Private nVerts As Integer
Private nTree As Integer
Private sPath() As DistOriginal
Private currentVert As Integer
Private startToCurrent As Integer

Public Sub New()
  ReDim vertexList(max_verts)
  ReDim adjMat(max_verts, max_verts)
  nVerts = 0
  nTree = 0
  Dim j, k As Integer
  For j = 0 To max_verts - 1
    For k = 0 To max_verts - 1
      adjMat(j, k) = infinity
    Next k
  Next j
  ReDim sPath(max_verts)
End Sub

Public Sub addVertex(ByVal lab As String)
  vertexList(nVerts) = New Vertex(lab)
  nVerts += 1
End Sub

Public Sub addEdge(ByVal start As Integer, _
                   ByVal theEnd As Integer, _
                   ByVal weight As Integer)
  adjMat(start, theEnd) = weight
End Sub

Public Sub Path()
  Dim startTree As Integer = 0
  vertexList(startTree).isInTree = True
  nTree = 1
  Dim j As Integer
  For j = 0 To nVerts - 1
    Dim tempDist As Integer = adjMat(startTree, j)
    sPath(j) = New DistOriginal(startTree, tempDist)
```

```
  Next
  While (nTree < nVerts)
    Dim indexMin As Integer = getMin()
    Dim minDist As Integer = _
        sPath(indexMin).distance
    currentVert = indexMin
    startToCurrent = sPath(indexMin).distance
    vertexList(currentVert).isInTree = True
    nTree += 1
    adjustShortPath()
  End While
  displayPaths()
  nTree = 0
  For j = 0 To nVerts - 1
    vertexList(j).isInTree = False
  Next
End Sub

Public Function getMin() As Integer
  Dim minDist As Integer = infinity
  Dim indexMin As Integer = 0
  Dim j As Integer
  For j = 1 To nVerts - 1
    If (Not vertexList(j).isInTree And _
      sPath(j).distance < minDist) Then
      minDist = sPath(j).distance
      indexMin = j
    End If
  Next
  Return indexMin
End Function

Public Sub adjustShortPath()
  Dim column As Integer = 1
  While (column < nVerts)
    If (vertexList(column).isInTree) Then
      column += 1
    Else
      Dim currentToFringe As Integer = _
        adjMat (currentVert, column)
```

```
        Dim startToFringe As Integer = _
          startToCurrent + currentToFringe
        Dim sPathDist As Integer = _
              sPath(column).distance
        If (startToFringe < sPathDist) Then
            sPath(column).parentVert = currentVert
            sPath(column).distance = startToFringe
        End If
        column += 1
      End If
    End While
  End Sub

  Public Sub displayPaths()
    Dim j As Integer
    For j = 0 To nVerts - 1
      Console.Write(vertexList(j).label + "=")
      If (sPath(j).distance = infinity) Then
        Console.Write("inf")
      Else
        Console.Write(sPath(j).distance)
      End If
      Dim parent As String = _
          vertexList (sPath(j).parentVert).label
      Console.Write("(" + parent + ") ")
    Next
    Console.WriteLine("")
  End Sub
End Class

Sub Main()
  Dim theGraph As Graph
  theGraph = New Graph
  theGraph.addVertex("A")
  theGraph.addVertex("B")
  theGraph.addVertex("C")
  theGraph.addVertex("D")
  theGraph.addVertex("E")
  theGraph.addVertex("F")
  theGraph.addVertex("G")
```

```
        theGraph.addEdge(0, 1, 2)
        theGraph.addEdge(0, 3, 1)
        theGraph.addEdge(1, 3, 3)
        theGraph.addEdge(1, 4, 10)
        theGraph.addEdge(2, 5, 5)
        theGraph.addEdge(2, 0, 4)
        theGraph.addEdge(3, 2, 2)
        theGraph.addEdge(3, 5, 8)
        theGraph.addEdge(3, 4, 2)
        theGraph.addEdge(3, 6, 4)
        theGraph.addEdge(4, 6, 6)
        theGraph.addEdge(6, 5, 1)
        Console.WriteLine()
        Console.WriteLine("Shortest paths:")
        Console.WriteLine()
        theGraph.path()
        Console.WriteLine()
        Console.Read()
    End Sub

End Module
```

The output from this program looks like this:



SUMMARY

Graphs are one of the most important data structures used in computer science. Graphs are used regularly to model everything from electrical circuits to university course schedules to truck and airline routes.

Graphs are made up of vertices that are connected by edges. Graphs can be searched in several ways; the two most common are depth-first search and breadth-first search. Another important algorithm performed on a graph

is one that determines the minimum spanning tree, which is the minimum number of edges necessary to connect all the vertices in a graph.

The edges of a graph can have weights, or costs. When working with weighted graphs, an important operation is determining the shortest path from a starting vertex to the other vertices in the graph. This chapter looked at one algorithm for computing shortest paths, Dijkstra's algorithm.

Weiss (1999) contains a more technical discussion of the graph algorithms covered in this chapter, and LaFore (1998) contains very good practical explanations of all the algorithms we covered here.

Exercises

1. Build a weighted graph that models a map of the area where you live. Use Dijkstra's algorithm to determine the shortest path from a starting vertex to the last vertex.
2. Take the weights off the graph in Exercise 1 and build a minimum spanning tree.
3. Still using the graph from Exercise 1, write a Windows application that allows the user to search for a vertex in the graph using either a depth-first search or a breadth-first search.
4. Using the Timing class, determine which of the searches implemented in Exercise 3 is more efficient.

CHAPTER 17

Advanced Algorithms

In this chapter we look at two advanced topics: dynamic programming and greedy algorithms. *Dynamic programming* is a technique that is often considered to be the reverse of recursion. Whereas a recursive solution starts at the top and breaks the problem down solving all small problems until the complete problem is solved; a dynamic programming solution starts at the bottom, solving small problems and combining them to form an overall solution to the big problem.

A *greedy algorithm* is an algorithm that looks for "good solutions" as it works toward the complete solution. These good solutions, called *local optima*, will hopefully lead to the correct final solution, called the *global optimum*. The term "greedy" comes from the fact that these algorithms take whatever solution looks best at the time. Often, greedy algorithms are used when it is almost impossible to find a complete solution, owing to time and/or space considerations, yet a suboptimal solution is acceptable.

A good source for more information on advanced algorithms and data structures is (Cormen, 2001).

DYNAMIC PROGRAMMING

Recursive solutions to problems are often elegant but inefficient. The VB.NET compiler, along with other language compilers, will not efficiently translate

the recursive code to machine code, resulting in an inefficient, though elegant computer program.

Many programming problems that have recursive solutions can be rewritten using the techniques of dynamic programming. A dynamic programming solution builds a table, usually using an array, that holds the results of the different subsolutions. Finally, when the algorithm is complete, the solution is found in a distinct spot in the table.

A Dynamic Programming Example: Computing Fibonacci Numbers

The Fibonacci numbers can be described by the following sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .

There is a simple recursive program you can use to generate any specific number in this sequence. Here is the code for the function:

```
Function recurFib(ByVal n As Integer) As Long
  If (n < 2) Then
    Return n
  Else
    Return recurFib(n - 1) + recurFib(n - 2)
  End If
End Function
```

A simple program that uses this function is

```
Sub Main()
  Dim num As Integer = 5
  Dim fibNumber As Long = recurFib(num)
  Console.Write(fibNumber)
  Console.Read()
End Sub
```

The problem with this function is that it is extremely inefficient. We can see exactly how inefficient this recursion is by examining the tree in Figure 17.1.

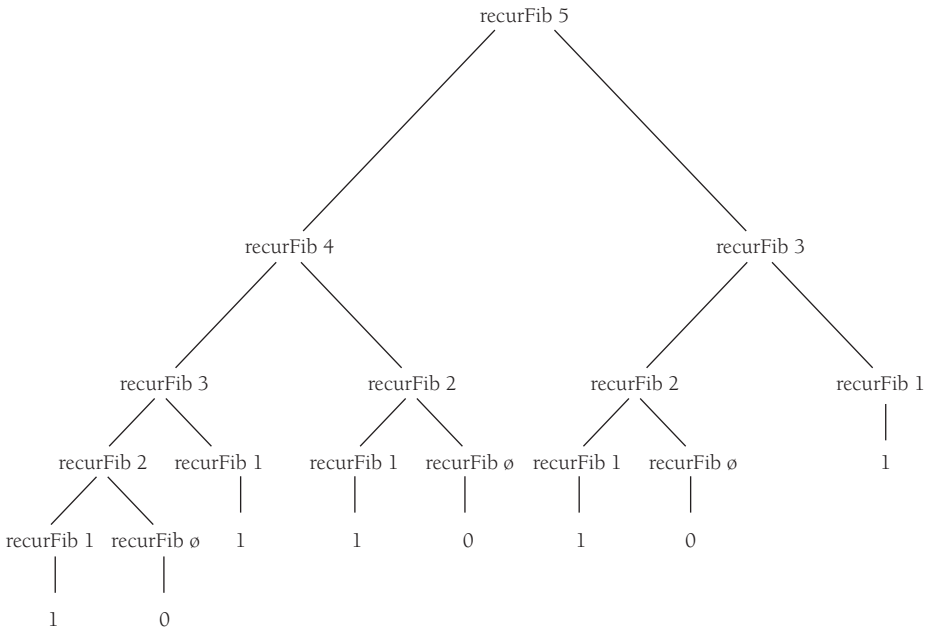


FIGURE 17.1. Tree generated from Recursive Fibonacci Computation.

The problem with the recursive solution is that too many values are recomputed during a recursive call. If the compiler could keep track of the values that are already computed, the function would not be nearly so inefficient. We can design an algorithm using dynamic programming techniques that is much more efficient than the recursive algorithm.

An algorithm designed using dynamic programming techniques starts by solving the simplest subproblem it can solve, using that solution to solve more complex subproblems until the entire problem is solved. The solutions to each subproblem are typically stored in an array for easy access.

We can easily comprehend the essence of dynamic programming by examining the dynamic programming algorithm for computing a Fibonacci number. Here's the code followed by an explanation of how it works:

```
Function iterFib(ByVal n As Integer) As Long
  Dim index As Integer
  Dim val(n) As Integer
  If (n = 1 Or n = 2) Then
    Return 1
```

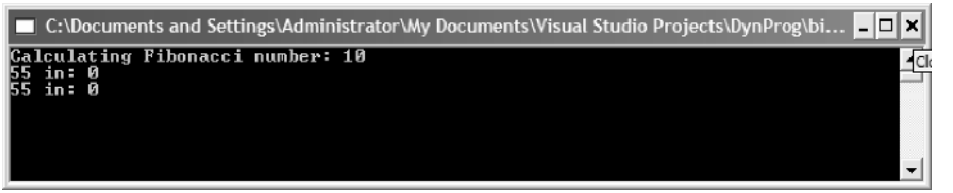
```
    Else
      val(1) = 1
      val(2) = 2
      For index = 3 To n - 1
        val(index) = val(index - 1) + val(index - 2)
      Next
      Return val(n - 1)
    End If
  End Function
```

The array val is where we store our intermediate results. The first part of the If statement returns the value 1 if the argument is 1 or 2. Otherwise, the values 1 and 2 are stored in the indices 1 and 2 of the array. The For loop runs from 3 to the input argument, assigning each array element the sum of the previous two array elements, and when the loop is complete, the last value in the array is returned.

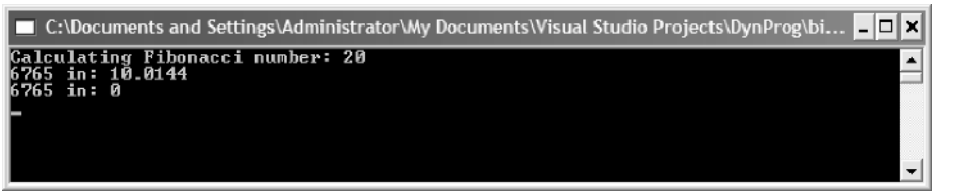
Let's compare the times it takes to compute a Fibonacci number using both the recursive version and the iterative version. First, here's the program we use for the comparison:

```
  Sub Main()
    Dim tObj As New Timing
    Dim tObj1 As New Timing
    Dim num As Integer = 10
    Dim fib_num As Integer
    tObj.startTime()
    fib_num = recurFib(num)
    tObj.stopTime()
    Console.WriteLine("Calculating Fibonacci number: " & _
                      num)
    Console.WriteLine(fib_num & " in: " & _
                      tObj.Result.TotalMilliseconds)
    tObj1.startTime()
    fib_num = iterFib(num)
    tObj1.stopTime()
    Console.WriteLine(fib_num & " in: " & _
                      tObj1.Result.TotalMilliseconds)
    Console.Read()
  End Sub
```

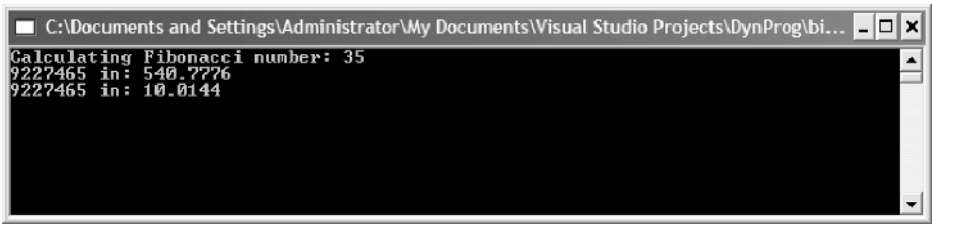
If we run this program to test the two functions for small Fibonacci numbers, we'll see little difference, or even see that the recursive function performs a little faster:



If we try a larger number, say 20, we get the following results:



For an even larger number, such as 35, the disparity is even greater:



This is a typical example of how dynamic programming can help improve the performance of an algorithm. As we mentioned earlier, a program using dynamic programming techniques usually utilizes an array to store intermediate computations, but we should point out that in some situations, such as the Fibonacci function, an array is not necessary. Here is the iterFib function written without the use of an array:

```
Function iterFib1(ByVal n As Integer) As Long
  Dim index As Integer
  Dim last, nextLast, result As Long
  last = 1
```

```
    nextLast = 1
    result = 1
    For index = 2 To n - 1
      result = last + nextLast
      nextLast = last
      last = result
    Next
    Return result
  End Function
```

Both iterFib and iterFib1 calculate Fibonacci numbers in about the same time.

Finding the Longest Common Substring

Another problem that lends itself to a dynamic programming solution is finding the longest common substring in two strings. For example, in the words "raven" and "havoc," the longest common substring is "av".

Let's look first at the brute-force solution to this problem. Given two strings, A and B, we can find the longest common substring by starting at the first character of A and comparing each character to the characters in B. When a nonmatch is found, move to the next character of A and start over with the first character of B, and so on.

There is a better solution using a dynamic programming algorithm. The algorithm uses a two-dimensional array to store the results of comparisons of the characters in the same position in the two strings. Initially, each element of the array is set to 0. Each time a match is found in the same position of the two arrays, the element at the corresponding row and column of the array is incremented by 1; otherwise the element is set to 0.

To reproduce the longest common substring, the second through the next to last rows of the array are examined and a column entry with a value greater than 0 corresponds to one character in the substring. If no common substring was found, all the elements of the array are 0.

Here is a complete program for finding a longest common substring:

```
Module Module1

  Sub LCSubstring(ByVal word1 As String, ByVal word2 _
      As String, ByRef warr1() As String, ByRef _
```

```
    warr2() As String, ByRef arr(,) As Integer)
  Dim i, j, k, len1, len2 As Integer
  len1 = word1.Length
  len2 = word2.Length
  For k = 0 To word1.Length - 1
    warr1(k) = word1.Chars(k)
    warr2(k) = word2.Chars(k)
  Next
  For i = len1 - 1 To 0 Step -1
    For j = len2 - 1 To 0 Step -1
      If (warr1(i) = warr2(j)) Then
        arr(i, j) = 1 + arr(i + 1, j + 1)
      Else
        arr(i, j) = 0
      End If
    Next
  Next
End Sub

Function ShowString(ByVal arr(,) As Integer, ByVal _
                    wordarr() As String) As String
  Dim i, j As Integer
  Dim substr As String = ""
  For i = 0 To arr.GetUpperBound(0)
    For j = 0 To arr.GetUpperBound(1)
      If (arr(i, j) > 0) Then
        substr &= wordarr(j)
      End If
    Next
  Next
  Return substr
End Function

Sub DispArray(ByVal arr(,) As Integer)
  Dim row, col As Integer
  For row = 0 To arr.GetUpperBound(0)
    For col = 0 To arr.GetUpperBound(0)
      Console.Write(arr(row, col))
    Next
```

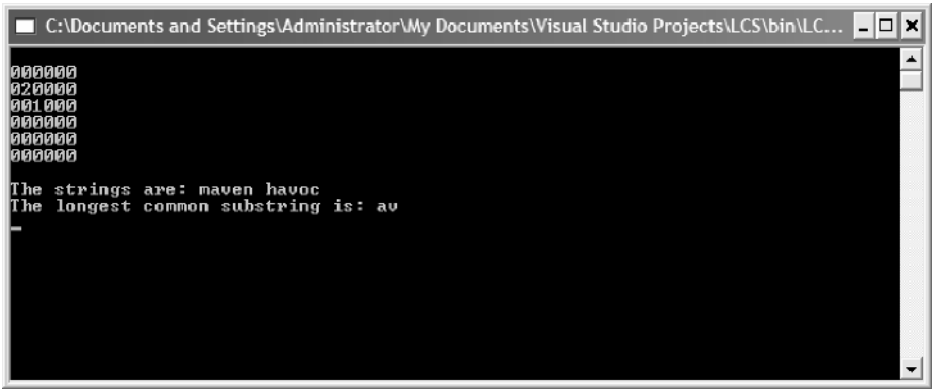
```
        Console.WriteLine()
      Next
    End Sub

    Sub Main()
      Dim word1 As String = "maven"
      Dim word2 As String = "havoc"
      Dim warray1(word1.Length) As String
      Dim warray2(word2.Length) As String
      Dim substr As String
      Dim larray(word1.Length, word2.Length) As Integer
      LCSubstring(word1, word2, warray1, warray2, larray)
      Console.WriteLine()
      DispArray(larray)
      substr = ShowString(larray, warray1)
      Console.WriteLine()
      Console.WriteLine("The strings are: " & word1 & _
                        " " & word2)
      If (substr > "") Then
        Console.WriteLine ("The longest common " & _
                           "substring is: " & substr)
      Else
        Console.WriteLine("There is no common substring")
      End If
      Console.Read()
    End Sub

End Module
```

The function LCSubstring does the work of building the two-dimensional array that stores the values that determine the longest common substring. The first For loop simply turns the two strings into arrays. The second For loop performs the comparisons and builds the array.

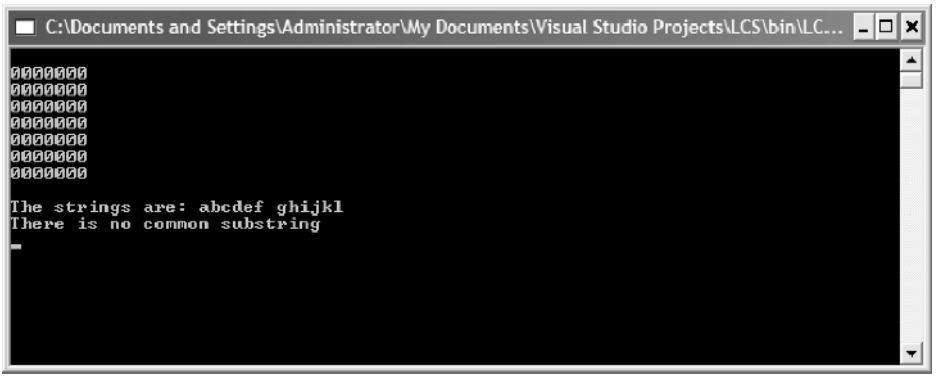
The function ShowString examines the array built in LCSubstring, checking to see whether any elements have a value greater than 0, and returning the corresponding letter from one of the strings if such a value is found.

The subroutine DispArray displays the contents of an array, which we use to examine the array built by LCSubstring when we run the preceding

program:



The encoding stored in larray shows us that the second and third characters of the two strings make up the longest common substring of "maven" and "havoc." Here's another example:



Clearly, these two strings have no common substring, so the array is filled with zeroes.

The Knapsack Problem

A classic problem in the study of algorithms is the knapsack problem. Imagine you are a safecracker and you break open a safe filled with all sorts of treasures but all you have to carry the loot is a small backpack. The items in the safe

differ both in size and value. You want to maximize your take by filling your backpack with those items that are worth the most.

There is, of course, a brute-force solution to this problem but the dynamic programming solution is more efficient. The key idea to solving the knapsack problem with a dynamic programming solution is to calculate the maximum value for every value up to the total capacity of the knapsack. See Sedgewick (1998, pp. 596–598) for a very clear and succinct explanation of the knapsack problem. The example problem in this section is based on the material from that book.

If the safe in our example has five items, the items have a size of 3, 4, 7, 8, and 9, respectively, and values of 4, 5, 10, 11, and 13, respectively, and the knapsack has a capacity of 16, then the proper solution is to pick items 3 and 5 with a total size of 16 and a total value of 23.

The code for solving this problem is quite short, but it won't make much sense without the context of the whole program, so let's look at a program to solve the knapsack problem:

```
Module Module1

  Sub Main()
    Dim capacity As Integer = 16
    Dim size() As Integer = {3, 4, 7, 8, 9}
    Dim values() As Integer = {4, 5, 10, 11, 13}
    Dim totval(capacity) As Integer
    Dim best(capacity) As Integer
    Dim n As Integer = values.Length
    Dim i, j As Integer
    For j = 0 To n - 1
      For i = 0 To capacity
        If (i >= size(j)) Then
          If (totval(i) < totval(i - size(j)) + _
                          values(j)) Then
            totval(i) = totval(i - size(j)) + values(j)
            best(i) = j
          End If
        End If
      Next
    Next
```

```
    Console.WriteLine("The maximum value is: " & _
                      totval(capacity))
    Console.Read()
  End Sub

End Module
```

The items in the safe are modeled using both the size array and the values array. The totval array is used to store the highest total value as the algorithm works through the different items. The best array stores the item that has the highest value. When the algorithm is finished, the highest total value will be in the last position of the totval array, with the next highest value in the next-to-last position, and so on. The same situation holds for the best array. The item with the highest value will be stored in the last element of the best array, the item with the second highest value in the next-to-last position, and so on.

The heart of the algorithm lies the second If statement in the nested For loop. The current best total value is compared to the total value of adding the next item to the knapsack. If the current best total value is greater, nothing happens. Otherwise, this new total value is added to the totval array as the best current total value and the index of that item is added to the best array. Here is the code again:

```
If (totval(i) < totval(i - size(j)) + values(j)) Then
  totval(i) = totval(i - size(j)) + values(j)
  best(i) = j
End If
```

If we want to see the items that generated the total value, we can examine them in the best array:

```
Console.WriteLine _
            ("The items that generate this value are: ")
Dim totcap As Integer = 0
i = capacity
While (totcap <= capacity)
  Console.WriteLine("Item with best value: " & best(i))
  totcap += values(best(i))
  i -= 1
End While
```

Remember, all the items that generate a previous best value are stored in the array, so we move down through the best array, returning items until their sizes equal the total capacity of the knapsack.

Greedy Algorithms

In the previous section, we examined dynamic programming algorithms that can be used to optimize solutions that are found using some less efficient algorithm, often based on recursion. For many problems, though, resorting to dynamic programming is overkill and a simpler algorithm will suffice.

One type of simpler algorithm is the *greedy* algorithm. A greedy algorithm is one that always chooses the best solution at the time, with no regard for how that choice will affect future choices. Using a greedy algorithm generally indicates that the implementer hopes that the series of "best" local choices made will lead to a final "best" choice. If so, then the algorithm has produced an optimal solution; if not, a suboptimal solution has been found. However, for many problems, it is not worth the trouble to find an optimal solution, so using a greedy algorithm works just fine.

A First Greedy Algorithm Example: The Coin-Changing Problem

The classic example of following a greedy algorithm is making change. Let's say you buy some items at the store and the change from your purchase is 63 cents. How does the clerk determine the change to give you? If the clerk follows a greedy algorithm, he or she gives you two quarters, a dime, and three pennies. That is the smallest number of coins that will equal 63 cents (given that we don't allow fifty-cent pieces).

It has been proven that an optimal solution for coin changing can always be found using the current American denominations of coins. However, if we introduce some other denomination to the mix, the greedy algorithm doesn't produce an optimal solution.

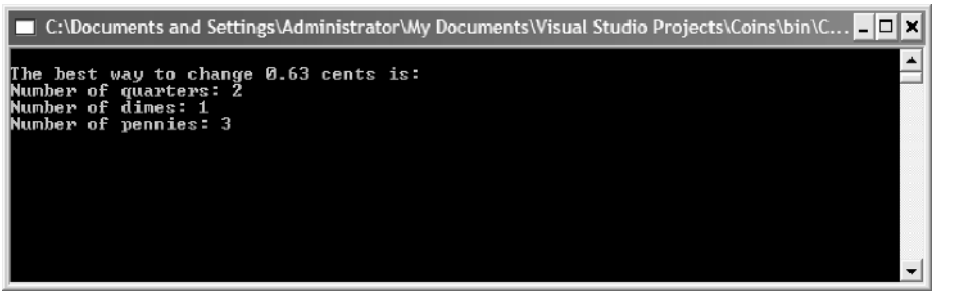
Here's a program that uses a greedy algorithm to make change (under the assumption that the change will be less than one dollar):

```
Module Module1

  Sub MakeChange(ByVal origAmount As Decimal, ByVal _
    remainAmount As Decimal, ByVal coins() As Integer)
```

```
  If (origAmount Mod 0.25 < origAmount) Then
    coins(3) = Int(origAmount / 0.25)
    remainAmount = origAmount Mod 0.25
    origAmount = remainAmount
  End If
  If (origAmount Mod 0.1 < origAmount) Then
    coins(2) = Int(origAmount / 0.1)
    remainAmount = origAmount Mod 0.1
    origAmount = remainAmount
  End If
  If (origAmount Mod 0.05 < origAmount) Then
    coins(1) = Int(origAmount / 0.05)
    remainAmount = origAmount Mod 0.05
    origAmount = remainAmount
  End If
  If (origAmount Mod 0.01 < origAmount) Then
    coins(0) = Int(origAmount / 0.01)
    remainAmount = origAmount Mod 0.01
  End If
End Sub

Sub ShowChange(ByVal arr() As Integer)
  If (arr(3) > 0) Then
    Console.WriteLine("Number of quarters: " & arr(3))
  End If
  If (arr(2) > 0) Then
    Console.WriteLine("Number of dimes: " & arr(2))
  End If
  If (arr(1) > 0) Then
    Console.WriteLine("Number of nickels: " & arr(1))
  End If
  If (arr(0) > 0) Then
    Console.WriteLine("Number of pennies: " & arr(0))
  End If
End Sub

Sub Main()
  Dim origAmount As Decimal = 0.63
  Dim toChange As Decimal = origAmount
  Dim remainAmount As Decimal
  Dim coins(3) As Integer
```

```
        MakeChange(origAmount, remainAmount, coins)
        Console.WriteLine("The best way to change " & _
                          toChange & " cents is: ")
        ShowChange(coins)
        Console.Read()
    End Sub

End Module
```

The MakeChange subroutine starts with the highest denomination, quarters, and tries to make as much change with them as possible. The total number of quarters is stored in the coins array. Once the original amount becomes less than a quarter, the algorithm moves to dimes, again trying to make as much change with dimes as possible. The algorithm proceeds to nickels and then to pennies, storing the total number of each coin type in the coins array. Here's some output from the program:



As we mentioned earlier, this greedy algorithm always find the optimal solution using the standard American coin denominations. What would happen, though, if a new coin, say a 22-cent piece, is put into circulation? In the exercises, you'll get a chance to explore this question.

Data Compression Using Huffman Coding

Compressing data is an important technique for the practice of computing. Data sent over the Internet need to be sent as compactly as possible. There are many different schemes for compressing data, but one particular scheme makes use of a greedy algorithm—*Huffman coding*. This algorithm is named for the late David Huffman, an information theorist and computer scientist who invented the technique in the 1950s. Data compressed using a Huffman code can achieve savings of 20% to 90%.

When data are compressed, the characters that make up the data are usually translated into some other representation to save space. A typical compression scheme is to translate each character to a binary character code, or bit string. For example, we can encode the character "a" as 000, the character "b" as 001, the character "c" as 010, and so on. This is called a *fixed-length code*.

A better idea, though, is to use a *variable-length code*, where the characters with the highest frequency of occurrence in the string have shorter codes and the lower frequency characters have longer codes, since these characters aren't used as much. The encoding process then is just a matter of assigning a bit string to a character based on the character's frequency. The Huffman code algorithm takes a string of characters, translates them to a variable-length binary string, and creates a binary tree for the purpose of decoding the binary strings. The path to each left child is assigned the binary character 0 and each right child is assigned the binary character 1.

The algorithm works as follows: Start with a string of characters you want to compress. For each character in the string, calculate its frequency of occurrence in the string. Then sort the characters into order from lowest frequency to highest frequency. Take the two characters with the smallest frequencies and create a node with each character (and its frequency) as children of the node. The parent node's data element consists of the sum of the frequencies of the two child nodes. Insert the node back into the list. Continue this process until every character is placed into the tree.

When this process is complete, you have a complete binary tree that can be used to decode the Huffman code. Decoding involves following a path of 0s and 1s until you get to a leaf node, which will contain a character.

To see how all this works, examine Figure 17.2.

Now we're ready to examine the VB.NET code for constructing a Huffman code. Let's start with the code for creating a Node class. This class is quite a bit different than the Node class for binary search trees, since all we want to do here is store some data and a link. Here is the code for the Node class:

```
Public Class Node

  Public data As HuffmanTree
  Public link As Node
  Public Sub New(ByVal newData As HuffmanTree)
    data = newData
  End Sub

End Class
```

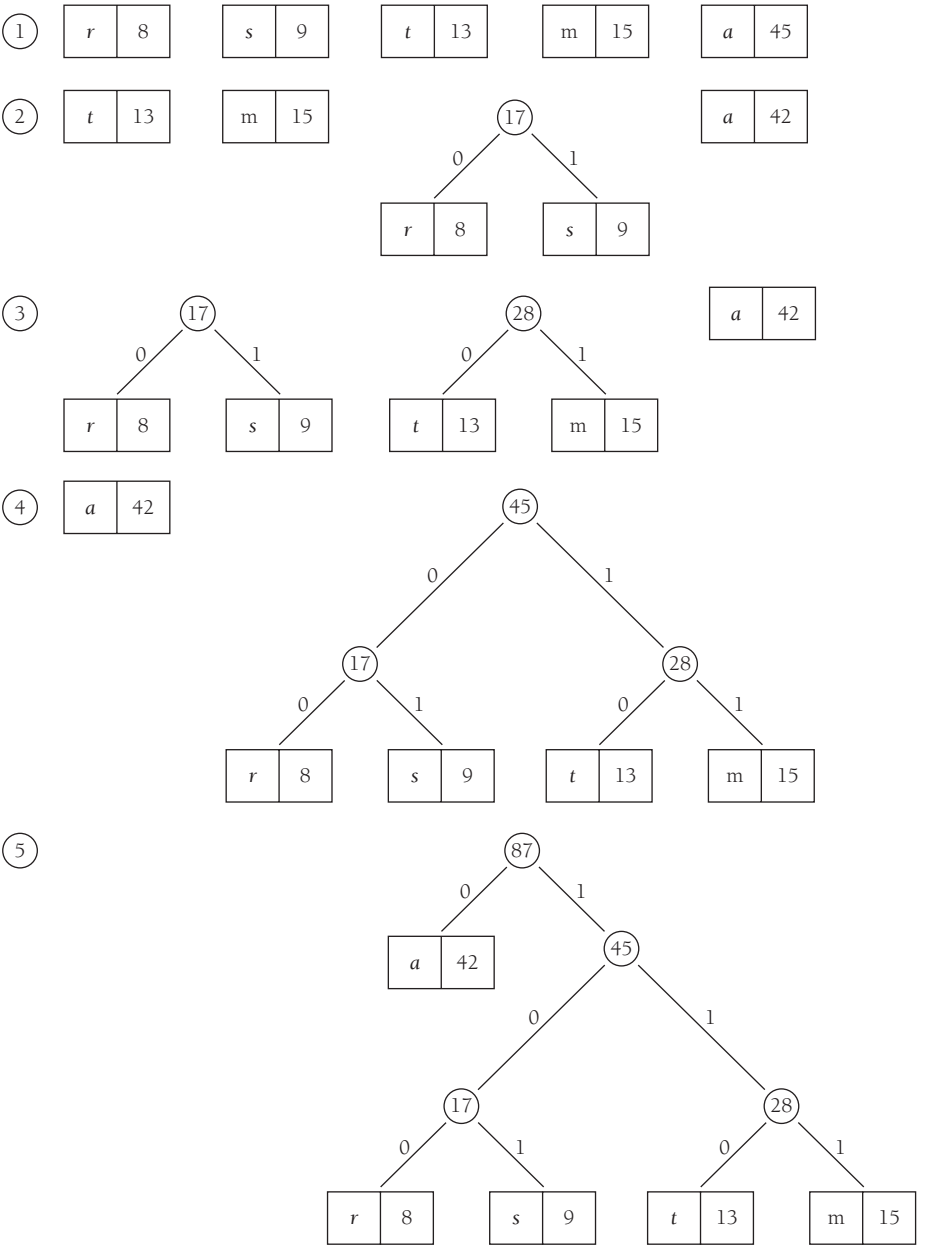


FIGURE 17.2. Constructing a Huffman Code.

The next class to examine is the TreeList class. This class is used to store the list of nodes that are placed into the binary tree, using a linked list as the storage technique. Here's the code:

```
Public Class TreeList

  Private count As Integer = 0
  Private first As Node

  Public Sub addLetter(ByVal Letter As String)
    Dim hTemp As New HuffmanTree(Letter)
    Dim eTemp As New Node(hTemp)
    If (first Is Nothing) Then
      first = eTemp
    Else
      eTemp.link = first
      first = eTemp
    End If
    count += 1
  End Sub

  Public Sub sortTree()
    Dim otherList As New TreeList
    Dim aTemp As HuffmanTree
    While Not (Me.first Is Nothing)
      aTemp = Me.removeTree()
      otherList.insertTree(aTemp)
    End While
    Me.first = otherList.first
  End Sub

  Public Sub mergeTree()
    If Not (first Is Nothing) Then
      If Not (first.link Is Nothing) Then
        Dim aTemp As HuffmanTree = removeTree()
        Dim bTemp As HuffmanTree = removeTree()
        Dim sumTemp As New HuffmanTree
        sumTemp.setLeftChild(aTemp)
        sumTemp.setRightChild(bTemp)
        sumTemp.setFreq(aTemp.getFreq + bTemp.getFreq)
```

```
        insertTree(sumTemp)
      End If
    End If
  End Sub

  Public Function removeTree() As HuffmanTree
    If Not (first Is Nothing) Then
      Dim hTemp As HuffmanTree
      hTemp = first.data
      first = first.link
      count -= 1
      Return hTemp
    End If
    Return Nothing
  End Function

  Public Sub insertTree(ByVal hTemp As HuffmanTree)
    Dim eTemp As New Node(hTemp)
    If (first Is Nothing) Then
      first = eTemp
    Else
      Dim p As Node = first
      While Not (p.link Is Nothing)
        If (p.data.getFreq <= hTemp.getFreq And _
            p.link.data.getFreq >= hTemp.getFreq) Then
            Exit While
        End If
        p = p.link
      End While
      eTemp.link = p.link
      p.link = eTemp
    End If
    count += 1
  End Sub

  Public Function length() As Integer
    Return count
  End Function

End Class
```

This class makes use of the HuffmanTree class, so let's view that code now:

```
Public Class HuffmanTree

  Private leftChild As HuffmanTree
  Private rightChild As HuffmanTree
  Private Letter As String
  Private freq As Integer

  Public Sub New(ByVal Letter As String)
    Me.Letter = Letter
  End Sub

  Public Sub setLeftChild(ByVal newChild As HuffmanTree)
    leftChild = newChild
  End Sub

  Public Sub setRightChild(ByVal newChild As _
                           HuffmanTree)
    rightChild = newChild
  End Sub

  Public Sub setLetter(ByVal newLetter As String)
    Letter = newLetter
  End Sub

  Public Sub incFreq()
    freq += 1
  End Sub

  Public Sub setFreq(ByVal newFreq As Integer)
    freq = newFreq
  End Sub

  Public Function getLeftChild() As HuffmanTree
    Return leftChild
  End Function
  Public Function getRightChild() As HuffmanTree
    Return rightChild
  End Function

  Public Function getLetter() As String
    Return Letter
```

```
  End Function
  Public Function getFreq() As Integer
    Return freq
  End Function
End Class
```

Finally, we need a program to test the implementation:

```
Sub Main()
  Dim input As String
  Console.Write("Enter a string to encode: ")
  input = Console.ReadLine
  Dim treeList As New TreeList
  Dim i As Integer
  For i = 0 To input.Length - 1
    treeList.addSign(input.Chars(i))
  Next
  treeList.sortTree()
  ReDim signTable(input.Length)
  ReDim keyTable(input.Length)
  While (treeList.length > 1)
    treeList.mergeTree()
  End While
  makeKey(treeList.removeTree, "")
  Dim newStr As String = translate(input)
  For i = 0 To signTable.Length - 1
    Console.WriteLine(signTable(i) & " : " & _
                      keyTable(i))
  Next
  Console.WriteLine("The original string is " & _
                    input.Length * 16 & " bits long")
  Console.WriteLine("The new string is " & _
                    newStr.Length & " bits long.")
  Console.WriteLine _
         ("The coded string looks like this: " & newStr)
  Console.Read()
End Sub
```

```
Function translate(ByVal original As String) As String
  Dim newStr As String = ""
  Dim i, j As Integer
  For i = 0 To original.Length - 1
    For j = 0 To signTable.Length - 1
      If (original.Chars(i) = signTable(j)) Then
        newStr &= keyTable(j)
      End If
    Next
  Next
  Return newStr
End Function

Sub makeKey(ByVal tree As HuffmanTree, ByVal code _
            As String)
  If (tree.getLeftChild Is Nothing) Then
    signTable(pos) = tree.getSign()
    keyTable(pos) = code
    pos += 1
  Else
    makeKey(tree.getLeftChild, code & "0")
    makeKey(tree.getRightChild, code & "1")
  End If
End Sub
```

A Greedy Solution to the Knapsack Problem

Earlier in this chapter we examined the knapsack problem and wrote a program to solve the problem using dynamic programming techniques. In this section we look at the problem again, this time looking for a greedy algorithm to solve the problem.

To use a greedy algorithm to solve the knapsack problem, the items we are placing in the knapsack need to be "continuous" in nature. In other words, they must be items like cloth or gold dust that cannot be counted discretely. If we are using these types of items, then we can simply divide the unit price by the unit volume to determine the value of the item. An optimal solution is to place as much of the item with the highest value in the knapsack as possible until the item is depleted or the knapsack is full, followed by as much of the second highest value item as possible, and so on. The reason we can't find

an optimal greedy solution using discrete items is that we can't put "half a television" into a knapsack.

Let's look at an example. You are a carpet thief and you have a knapsack that will store only 25 "units" of carpeting. Therefore, you want to get as much of the "good stuff" as you can to maximize your take. You know that the carpet store you're going to rob has the following carpet styles and quantities on hand (with unit prices):

- Saxony, 12 units, $1.82
- Loop, 10 units, $1.77
- Frieze, 12 units, $1.75
- Shag, 13 units, $1.50

The greedy strategy dictates that you take as many units of Saxony as possible, followed by as many units of Loop, then Frieze, and finally Shag. Being the computational type, you first write a program to model your heist. Here is the code you come up with:

```
Option Strict On
Module Module1
  Public Class Carpet
    Implements IComparable
    Private item As String
    Private val As Single
    Private unit As Integer

    Public Sub New(ByVal i As String, ByVal v As _
                   Single, ByVal u As Integer)
      item = i
      val = v
      unit = u
    End Sub
    Public Function CompareTo(ByVal obj As Object) As _
        Integer Implements System.IComparable.CompareTo
      Return Me.val.CompareTo(CType(obj, Carpet).val)
    End Function

    Public ReadOnly Property getUnit() As Integer
      Get
        Return unit
```

```
    End Get
  End Property

  Public ReadOnly Property getItem() As String
    Get
      Return item
    End Get
  End Property

  Public ReadOnly Property getVal() As Single
    Get
      Return val * unit
    End Get
  End Property

  Public ReadOnly Property itemVal() As Single
    Get
      Return val
    End Get
  End Property
End Class

Public Class Knapsack
  Private quantity As Single
  Dim items As New SortedList
  Dim itemList As String

  Public Sub New(ByVal max As Single)
    quantity = max
  End Sub

  Public Sub FillSack(ByVal objects As ArrayList)
    Dim pos As Integer = objects.Count - 1
    Dim totalUnits As Integer = 0
    Dim totalVal As Single = 0
    Dim tempTot As Integer = 0
    While (totalUnits < quantity)
      tempTot += CType(objects(pos), Carpet).getUnit
      If (tempTot <= quantity) Then
        totalUnits += CType(objects(pos), _
                            Carpet).getUnit
        totalVal += CType(objects(pos), Carpet).getVal
```

```
            items.Add(CType(objects(pos), Carpet). _
                  getItem, CType(objects(pos), Carpet). _
                  getUnit)
          Else
            Dim tempUnit As Single
            Dim tempVal As Single
            tempUnit = quantity - totalUnits
            tempVal = CType(objects(pos), Carpet). _
                           itemVal * tempUnit
            totalVal += tempVal
            totalUnits += CInt(tempUnit)
            items.Add(CType(objects(pos), Carpet). _
                      getItem, tempUnit)
          End If
          pos -= 1
        End While
      End Sub

      Public Function getItems() As String
        Dim k, v As Object
        For Each k In items.GetKeyList
          itemList &= CStr(k) & ": " & _
                      CStr(items.Item(k)) & " "
        Next
        Return itemList
      End Function
    End Class

    Sub Main()
      Dim c1 As New Carpet("Frieze", 1.75, 12)
      Dim c2 As New Carpet("Saxony", 1.82, 9)
      Dim c3 As New Carpet("Shag", 1.5, 13)
      Dim c4 As New Carpet("Loop", 1.77, 10)
      Dim rugs As New ArrayList
      rugs.Add(c1)
      rugs.Add(c2)
      rugs.Add(c3)
      rugs.Add(c4)
      rugs.Sort()
      Dim k As New Knapsack(25)
```

```
        k.FillSack(rugs)
        Console.WriteLine(k.getItems)
        Console.Read()
    End Sub
End Module
```

The Carpet class is used for two reasons—to encapsulate the data about each type of carpeting and to implement the IComparable interface, so we can sort the carpet types by their unit cost.

The Knapsack class does most of the work in this implementation. It provides a list to store the carpet types and it provides a method, FillSack, to determine how the knapsack gets filled. Also, the constructor method allows the user to pass in a quantity that sets the maximum number of units the knapsack can hold.

The FillSack method loops through the carpet types, adding as much of the most valuable carpeting as possible into the knapsack, then moving on to the next type. At the point where the knapsack becomes full, the code in the Else clause of the If statement puts the proper amount of carpeting into the knapsack.

This code works because we can cut the carpeting wherever we want. If we were trying to fill the knapsack with some other item that does not come in continuous quantities, we would have to move to a dynamic programming solution.

SUMMARY

This chapter examined two advanced techniques for algorithm design: dynamic programs and greedy algorithms. Dynamic programming is a technique where a bottom-up approach is taken to solving a problem. Rather than working its way down to the bottom of a calculation, such as done with recursive algorithm, a dynamic programming algorithm starts at the bottom and builds on those results until the final solution is reached.

Greedy algorithms look for solutions as quickly as possible and then stop before looking for all possible solutions. A problem solved with a greedy algorithm will not necessarily be the optimal solution because the greedy algorithm will have stopped with a "sufficient" solution before finding the optimal solution.

Exercises

1. Rewrite the longest common substring code as a class.
2. Write a program that uses a brute-force technique to find the longest common substring. Use the Timing class to compare the brute-force method with the dynamic programming method. Use the program from Exercise 1 for your dynamic programming solution.
3. Write a Windows application that allows the user to explore the knapsack problem. The user should be able to change the capacity of the knapsack, the sizes of the items, and the values of the items. The user should also create a list of item names that is associated with the items used in the program.
4. Find at least two new coin denominations that make the greedy algorithm for coin changing shown in the chapter produce suboptimal results.
5. Using a "commercial" compression program, such as WinZip, compress a small text file. Then compress the same text file using a Huffman code program. Compare the results of the two compression techniques.
6. Using the code from the "carpet thief" example, change the items being stolen to televisions. Can you fill up the knapsack completely? Make changes to the example program to answer the question.

References

Bentley, Jon. *Programming Pearls. Second Edition*. Reading, Massachusetts: Addison-Wesley, 2000.

Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Clifford Stein. *Introduction to Algorithms*. Cambridge, Massachusetts: The MIT Press, 2001.

Ford, William and William Topp. *Data Structures with C++*. Upper Saddle River, New Jersey: Prentice Hall, 1996.

Friedel, Jeffrey E. F. *Mastering Regular Expressions*. Sebastopol, California: O'Reilly and Associates, 1997.

Knuth, Donald E., *The Art of Computer Programming, Volume 1, Fundamental Algorithms*. Reading, Massachusetts: Addison Wesley, 1998.

LaFore, Robert. *Data Structures and Algorithms in Java*. Corte Madera, California: Waite Group Press, 1998.

McMillan, Michael. *Object-Oriented Programming with Visual Basic.NET*. New York: Cambridge University Press, 2004.

Sedgewick, Robert. *Algorithms in C*. Reading, Massachusetts: Addison Wesley, 1998.

Weiss, Mark Allen. *Data Structures and Algorithm Analysis in Java*. Reading, Massachusetts: Addison Wesley, 1999.

Index

C

E

M

Q

S

T