CAMBRIDGE 大学计算机教育国外著名教材系列 影印版

Data Structures and Algorithms Using C#

数据结构与算法（C#语言版）

Michael McMillan 著

清华大学出版社

大学计算机教育国外著名教材系列（影印版）

Data Structures and Algorithms Using C#

数据结构与算法

（C#语言版）

Michael McMillan

清华大学出版社
北京

图书在版编目(CIP)数据

数据结构与算法：C#语言版 = Data Structures and Algorithms Using C#: 英文 /（美）麦克米兰（McMillan, M.）著. --影印本. --北京：清华大学出版社，2009.5
（大学计算机教育国外著名教材系列（影印版））

ISBN 978-7-302-19798-0

Ⅰ. 数… Ⅱ. 麦… Ⅲ. ①数据结构－高等学校－教材－英文 ②算法分析－高等学校－教材－英文 ③C 语言－程序设计－高等学校－教材－英文 Ⅳ. TP311.12 TP301.6 TP312

中国版本图书馆 CIP 数据核字（2009）第 045368 号

责任印制：杨　艳

出版发行：清华大学出版社　　**地　　址：**北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编：**100084
社　总　机：010-62770175　　**邮　购：**010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
印 装 者：北京鑫海金澳胶印有限公司
发 行 者：全国新华书店
开　本：185×230　　**印张：**22.25
版　次：2009 年 5 月第 1 版　　**印　次：**2009 年 5 月第 1 次印刷
印　数：1~3000
定　价：35.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：010-62770177 转 3103　　产品编号：028859-01

出 版 说 明

进入21世纪，世界各国的经济、科技以及综合国力的竞争将更加激烈。竞争的中心无疑是对人才的竞争。谁拥有大量高素质的人才，谁就能在竞争中取得优势。高等教育，作为培养高素质人才的事业，必然受到高度重视。目前我国高等教育的教材更新较慢，为了加快教材的更新频率，教育部正在大力促进我国高校采用国外原版教材。

清华大学出版社从1996年开始，与国外著名出版公司合作，影印出版了“大学计算机教育丛书（影印版）”等一系列引进图书，受到国内读者的欢迎和支持。跨入21世纪，我们本着为我国高等教育教材建设服务的初衷，在已有的基础上，进一步扩大选题内容，改变图书开本尺寸，一如既往地请有关专家挑选适用于我国高校本科及研究生计算机教育的国外经典教材或著名教材，组成本套“大学计算机教育国外著名教材系列（影印版）”，以飨读者。深切期盼读者及时将使用本系列教材的效果和意见反馈给我们。更希望国内专家、教授积极向我们推荐国外计算机教育的优秀教材，以利我们把“大学计算机教育国外著名教材系列（影印版）”做得更好，更适合高校师生的需要。

清华大学出版社

DATA STRUCTURES AND ALGORITHMS USING C#

C# programmers: no more translating data structures from C++ or Java to use in your programs! Mike McMillan provides a tutorial on how to use data structures and algorithms plus the first comprehensive reference for C# implementation of data structures and algorithms found in the .NET Framework library, as well as those developed by the programmer.

The approach is very practical, using timing tests rather than Big O notation to analyze the efficiency of an approach. Coverage includes array and ArrayLists, linked lists, hash tables, dictionaries, trees, graphs, and sorting and searching algorithms, as well as more advanced algorithms such as probabilistic algorithms and dynamic programming. This is the perfect resource for C# professionals and students alike.

Michael McMillan is Instructor of Computer Information Systems at Pulaski Technical College, as well as an adjunct instructor at the University of Arkansas at Little Rock and the University of Central Arkansas. Mike's previous books include *Object-Oriented Programming with Visual Basic.NET, Data Structures and Algorithms Using Visual Basic.NET*, and *Perl from the Ground Up*. He is a co-author of *Programming and Problem-Solving with Visual Basic.NET*. Mike has written more than twenty-five trade journal articles on programming and has more than twenty years of experience programming for industry and education.

Preface

The study of data structures and algorithms is critical to the development of the professional programmer. There are many, many books written on data structures and algorithms, but these books are usually written as college textbooks and are written using the programming languages typically taught in college—Java or C++. C# is becoming a very popular language and this book provides the C# programmer with the opportunity to study fundamental data structures and algorithms.

C# exists in a very rich development environment called the .NET Framework. Included in the .NET Framework library is a set of data structure classes (also called collection classes), which range from the Array, ArrayList, and Collection classes to the Stack and Queue classes and to the HashTable and the SortedList classes. The data structures and algorithms student 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 computation, such as number base conversions, demonstrating the utility of the data structure immediately. With this background, the student 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 Oh analysis (if you don't know what this means, look at any of the books mentioned in the bibliography). Instead, the various data structures and algorithms are presented as problem-solving tools.

Simple timing tests are used to compare the performance of the data structures and algorithms discussed in the book.

Prerequisites

The only prerequisite for this book is that the reader have some familiarity with the C# language in general, and object-oriented programming in C# in particular.

Chapter-by-Chapter Organization

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. This chapter also introduces the concept of generic programming, which allows the programmer to write one class, or one method, and have it work for a multitude of data types. Generic programming is an important new addition to C# (available in C# 2.0 and beyond), so much so that there is a special library of generic data structures found in the System.Collections.Generic namespace. When a data structure has a generic implementation found in this library, its use is discussed. The chapter ends with an introduction to methods of measuring the performance of the data structures and algorithms discussed in the book.

Chapter 2 provides a review of how arrays are constructed, 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 is 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. The emphasis in this chapter is on 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. Because so much

data processing in C# 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 C# as they are in a pointer-based language such as C++, but they still have a role in C# 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, is 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 in which 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 is 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 algorithm design techniques really are: dynamic algorithms and greedy algorithms.

ACKNOWLEDGEMENTS

There are several different groups of people who must be thanked for helping me finish this book. First, thanks to a certain group of students who first sat through my lectures on developing data structures and algorithms. 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 dean, David Durr, and my department chair, Bernica Tackett, for supporting my writing endeavors. I also need to thank my family for putting up with me while I was preoccupied with research and writing. Finally, many thanks to my editors at Cambridge, Lauren Cowles and Heather Bergman, for putting up with my many questions, topic changes, and habitual lateness.

Contents

CHAPTER 1

An Introduction to Collections, Generics, and the Timing Class

This book discusses the development and implementation of data structures and algorithms using C#. The data structures we use in this book 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.

An important addition to C# 2.0 is generics. Generics allow the C# programmer to write one version of a function, either independently or within a class, without having to overload the function many times to allow for different data types. C# 2.0 provides a special library, System.Collections.Generic, that implements generics for several of the System.Collections data structures. This chapter will introduce the reader to generic programming.

Finally, this chapter introduces a custom-built class, the Timing class, which we will use in several chapters to measure the performance of a data structure and/or algorithm. This class will take the place of Big O analysis, not because Big O analysis isn't important, but because this book takes a more practical approach to the study of data structures and algorithms.

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, as well as operations for 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 is a good example of 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 if 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.

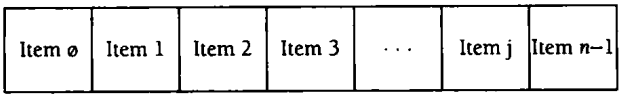

FIGURE 1.1. Array.

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 C#, arrays are not only a built-in data type, 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 efficient), since we will have to move elements of the array down in order 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 C#. 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.

C# strings are immutable, meaning once a string is initialized it cannot be changed. When you modify a string, a copy of the string is created 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 struct (also called structures and records in other languages). A struct is a composite data type that holds data that may consist of many different data types. For example, an employee

record consists of employee' name (a string), salary (an integer), identification number (a string, or an integer), as well as other attributes. Since storing each of these data values in separate variables could become confusing very easily, the language provides the struct for storing data of this type.

A powerful addition to the C# struct is the ability to define methods for performing operations stored on the data in a struct. This makes a struct somewhat like a class, though you can't inherit or derive a new type from a structure. The following code demonstrates a simple use of a structure in C#:

```
using System;

  public struct Name {

  private string fname, mname, lname;

  public Name(string first, string middle, string last) {
    fname = first;
    mname = middle;
    lname = last;
  }

  public string firstName {
    get {
      return fname;
    }

    set {
      fname = firstName;
    }
  }
  public string middleName {
    get {
      return mname;
    }

    set {
      mname = middleName;
    }
  }

  public string lastName {
    get {
```

```
        return lname;
      }

      set {
        lname = lastName;
      }
    }

    public override string ToString() {
      return (String.Format("{0} {1} {2}", fname, mname,
              lname));
    }

    public string Initials() {
     return (String.Format("{0}{1}{2}", fname.Substring(0,1),
            mname.Substring(0,1), lname.Substring(0,1)));
    }
  }

  public class NameTest {
    static void Main() {
      Name myName = new Name("Michael", "Mason", "McMillan");
      string fullName, inits;
      fullName = myName.ToString();
      inits = myName.Initials();
      Console.WriteLine("My name is {0}.", fullName);
      Console.WriteLine("My initials are {0}.", inits);
    }
  }
```

Although many of the elements in the .NET environment 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:

```
using System;

public class IntStruct {
  static void Main() {
```

```
        int num;
        string snum;
        Console.Write("Enter a number: ");
        snum = Console.ReadLine();
        num = Int32.Parse(snum);
        Console.WriteLine(num);
    }
}
```

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 is at the front of the list and the last element is at the rear of the list.

Because there is no 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—in one direction from front to rear, and 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 in respect to each other, as in:

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 on the list, as you'll see in Chapter 2 when we explore the binary search algorithm versus a simple linear search.

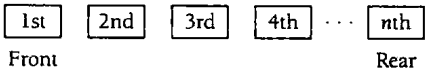

FIGURE 1.2. Linear List.