前言

数据结构与算法的学习对于进行软件开发的专业程序员而言是非常关键的。虽然有许许多多关于数据结构与算法的书籍，但是这些书籍通常都是大学教材，而且是用在大学里经典讲授的 Java 语言或 C++语言编写的。C#语言正在成为一种广受欢迎的编程语言。这本书为 C#语言程序员提供了学习基础数据结构与算法的机会。

C#语言根植在一个功能非常丰富的·NET 框架开发环境中。在·NET 框架库中包含有一套数据结构类（也称为集合类）。这套类的范围从 Array 类、ArrayList 类和 Collection 类到 Stack 类和 Queue 类，再到 Hashtable 类和 SortedList 类。学习数据结构与算法的学生在学习如何实现它们之前可以先明白如何使用数据结构。以前老师在构建完整的堆栈数据结构之前只能抽象地讲解堆栈的概念。而现在老师可以立刻通过示范数据结构工具来向学生们展示如何用堆栈执行一些计算，比如数制之间的转换。有了这些知识后，学生可以课后学习数据结构（或算法）的基本原理，甚至可以构造属于他们自己的实现。

本书把所有认真的计算机程序员们需要知道和了解的数据结构与算法的实践概述作为主要的写作内容。由于这种情况，本书没有涵盖数据结构与算法的正规的分析。因此，本书没有一个数学公式，也一次没有提及大 O 分析（如果你不知道大 O 分析的含义，查看参考文献中提到的任何一本书都可以）。取而代之的，本书把各种各样的数据结构与算法表示成求解问题的工具。书中讨论的数据结构与算法用简单的时间测试来进行性能比较。

前提条件

阅读本书的唯一前提条件就是需要读者对 C#语言有大概的了解 。如果会用 C#进行面向对象编程更好。

章节组织

第 1 章向读者介绍数据结构作为数据集合的概念。介绍线性和非线性集合的概念。示范说明了 Collection 类。本章还介绍泛型编程的概念。泛型编程允许程序员编写一个类或一种方法，并且把它用于众多数据类型。泛型编程是 C#语言一种重要的新特性（在 C#2·0 以及更高版本中可用）。这种特性是如此重要以至于在 System·Collections·Generic 命名空间中存在一个专门的泛型数据结构库。当数据结构具有在此库中能找到的泛型实现时，就会讨论它的用途。本章结尾处介绍了衡量书中讨论的数据结构与算法性能的方法。

第 2 章提供了数组构造方法的回顾，并连同示例说明了 Array 类的特征。Array 类把许多与数组相关的函数（UBound 函数、LBound 函数等等）封装到单独一个包中。ArrayLists 是

数组的一种特殊类型，它支持动态地调整容量。

第 3 章是对基础排序算法的介绍，例如冒泡排序和插入排序。而第 4 章则研究了用于内存查找的最基本算法，顺序查找和二叉查找。

第 5 章探讨了两种经典的数据结构：堆栈和队列。本章节强调的重点是这些数据结构在解决日常数据处理问题中的实际应用。第 6 章讲述了 BitArray 类。这种类可以用于有效地表示大量整数值，比如测试成绩。

数据结构的书中通常不包含字符串，但是第 7 章介绍了字符串、String 类和 StringBuilder 类。这是因为在 C#语言中许多的数据处理是在字符串上执行的，读者应该接触基于这两种类的特殊方法。第 8 章分析了用于文本处理和模式匹配的正则表达式的使用。与较传统的字符串函数和方法相比，正则表达式常常会提供更强大更有效的处理。

第 9 章向读者介绍作为数据结构的字典的使用。字典和基于字典的不同数据结构把数据作为键/值对来存储。本章向读者说明了如何创建基于 DictionaryBase 类的他或她自己的类。DictionaryBase 类是一个抽象类。第 10 章包括散列表和 HashTable 类。HashTable 类是字典的一种特殊类型，它用散列算法对内部数据进行存储。

链表作为另外一种经典的数据结构是在第 11 章介绍。链表在 C#语言中不像在 C++这样基于指针的语言中那样重要，但是它始终在 C#编程中发挥作用。第 12 章为读者介绍另一种经典数据结构——二叉树。二叉查找树作为二叉树的特殊类型将是本章的主要内容。其他二叉树类型在第 15 章进行介绍。

第 13 章向读者说明在集合中存储数据的方法。这种方法在数据结构只存储唯一数据值的情

况下是很实用的。第 *14* 章涵盖了高级排序算法，包括流行且高效的快速排序算法。此算法是大多数在·NET 框架库中实现的排序程序的基础。第 *15* 章会看到三种数据结构。在无法使用二叉查找树的时候，这三种数据结构证明对查找是很有用的。他们是：AVL 树、红黑树和跳跃表。

第 *16* 章讨论了图以及图的算法。图在表示许多不同的数据类型时非常有用，特别是网络的情况。最后，第 *17* 章向读者介绍真正的算法设计技巧是什么：动态算法和贪心算法。

致谢

在这里我要感谢来自各界帮助我完成此书的人们。首先，我要感谢首批聆听我讲授数据结构与算法开发课程的学生们。这些学生包括有（排名不分先后）：Matt Hoffman、Ken Chen、 Ken Cates、Jeff Richmond 以及 Gordon Caffey。此外，要感谢我在普拉斯基科技学院的同事 Clayton Ruff。他多次旁听我的课程并且提出了建议和批评。我还要感谢系主管 David Durr 和系主席 Bernica Tackett 支持我的写作努力。而且，我要感谢我的家人在我全身心投入研究和写作时对我的容忍。最后，我要感谢剑桥出版社的编辑 Lauren Cowles 和 Heather Bergman，感谢他们容忍我的许多问题，更改内容和经常的延迟。

目　录

前言　*1*

前提条件　*1*

章节组织　*1*

第 *1* 章 Collections 类、泛型类和 Timing 类概述　*18*

1·1 群集的定义 18

1·2 群集的描述 18

1·2·1 直接存取群集 18

1·2·2 顺序存取群集 20

1·2·3 层次群集 21

1·2·4 组群集 22

1·3 CollectionBase 类 22

1·3·1 用 ArrayLists 实现 Collection 类 22

1·3·2 定义 Collection 类 23

1·3·3 实现 Collection 类 23

1·4 范型编程 24

1·5 时间测试 25

1·5·1 一个简单化的时间测试 25

1·5·2 用于·NET 环境的时间测试 26

1·5·3Timing Test 类 27

小结 28

练习 28

第 2 章数组和 ArrayLists 30

2·1 数组基本概念 30

2·1·1 数组的声明和初始化 30

2·1·2 数组元素的设置和存取访问 30

2·1·3 取回数组元数据的方法和属性 31

2·1·4 多维数组 31

2·1·5 参数数组 32

2·1·6 锯齿状数组 32

2·2ArrayList 类 33

2·2·1ArrayList 类的成员 34

2·2·2 应用 ArrayList 类 34

ArrayList grades = new ArrayList(); 34

小结 36

练习 36

第 3 章基础排序算法 38

3·1 排序算法 38

3.1.1 数组类测试环境 38

3.1.2 冒泡排序 39

3.1.3 检验排序过程 40

3.1.4 选择排序 40

3.1.5 插入排序 41

3.2 基础排序算法的时间比较 42

小结 43

练习 43

第 4 章基础查找算法 44

4.1 顺序查找算法 44

4.1.1 查找最小值和最大值 45

4·1·2 自组织数据加快顺序查找速度 46

4·2 二叉查找算法 47

4·3 递归二叉查找算法 48

小结 49

练习 49

第 5 章堆栈和队列 50

5·1 堆栈、堆栈的实现以及 STACK 类 50

5·1·1 堆栈的操作 50

5·1·2Stack 类的实现 50

5·2STACK 类 52

5·2·1Stack 构造器方法 52

5·2·2 主要的堆栈操作 52

5·2·3Peek 方法 54

5·2·4Clear 方法 54

5·2·5Contains 方法 54

5·2·6CopyTo 方法和 ToArray 方法 54

5·2·7Stack 类的实例：十进制向多种进制的转换 54

5·3 队列、QUEUE 类以及 QUEUE 类的实现 55

5·3·1 队列的操作 55

5·3·2Queue 的实现 56

5·3·3 Queue 类：实例应用 56

5·3·4 用队列存储数据 58

5·3·5 源自 Queue 类的优先队列 *60*

小结 *61*

练习 *61*

第 *6* 章 BitArray 类 *63*

6·1 激发的问题 *63*

6·2 位和位操作 *63*

6·2·1 二进制数制系统 *64*

6·2·2 处理二进制数：按位运算符和位移运算符 *64*

6·3 按位运算符的应用 *65*

6·3·1 位移运算符 *66*

6·4 整数转换成二进制形式的应用程序 *66*

6·5 位移的示例应用程序 68

6·6BITARRAY 类 69

6·6·1 使用 BitArray 类 69

6·6·2 更多 BitArray 类的方法和属性 70

6·7 用 BITARRAY 来编写埃拉托斯特尼筛法 71

6·8BITARRAY 与数组在埃拉托斯特尼筛法上的比较 72

小结 72

练习 72

第 7 章 字符串、String 类和 StringBuilder 类 73

7·1STRING 类的应用 73

7·1·1 创建 String 对象 73

7·1·2 常用 String 类的方法们 73

7·1·3Split 方法和 Join 方法 75

7·1·4 比较字符串的方法 76

7·1·5 处理字符串的方法 78

7·2STRINGBUILDER 类 81

7·2·1 构造 StringBuilder 对象 81

7·2·2 获取并且设置关于 StringBuilder 对象的信息 81

7·2·3 修改 StringBuilder 对象 82

7·3STRING 类与 STRINGBUILDER 的性能比较 83

小结 84

练习 85

第 8 章 模式匹配和文本处理 86

8.1 正则表达式概述 86

8.1.1 概述：使用正则表达式 86

8.2 数量词 87

8.3 使用字符类 88

8.4 用断言修改正则表达式 90

8.5 使用分组构造 90

8.5.1 匿名组 90

8.5.2 命名组 91

8.5.3 零宽度正向预搜索断言和零宽度反向预搜索断言 91

8.6CAPTURESCOLLECTION 类 92

8.7 正则表达式的选项 92

小结 93

练习 93

第 9 章 构建字典：DictionaryBase 类和 SortedList 类 94

9.1DICTIONARYBASE 类 94

9.1.1DictionaryBase 类的基础方法和属性 94

9.1.2 其他的 DictionaryBase 方法 95

9.2 通用的 KEYVALUEPAIR 类 96

9.3SORTEDLIST 类 97

9.3.1 使用 SortedList 类 97

小结 97

练习 98

第10章 散列和 Hashtable 类 99

10.1 散列概述 99

10.2 选择散列函数 99

10.3 查找散列表中数据 100

10.4 解决冲突 101

10.4.1 桶式散列法 101

10.4.2 开放定址法 102

10.4.3 双重散列法 102

10.5 HASHTABLE 类 102

10.5.1 实例化 Hashtable 对象并且给其添加数据 102

10.5.2 从散列表中分别取回关键字和数值 103

10.5.3 取回基于关键字的数值 103

10.5.4 Hashtable 类的实用方法 104

10.6 HASHTABLE 的应用程序：计算机术语表 104

小结 106

练习 106

第 11 章 链表 107

11.1 数组存在的问题 107

11.2 链表的定义 107

11.3 面向对象链表的设计 108

11.3.1 Node 类 108

11·3·2 LinkedList 类 108

11·4 链表设计的改进方案 109

11·4·1 双向链表 110

11·4·2 循环链表 111

11·5 使用 ITERATOR 类 113

11·5·1 新的 LinkedList 类 114

11·5·2 实例化 Iterator 类 114

11·6 通用的 LINKED LIST 类和通用的 NODE 类 117

11·6·1 通用链表实例 117

小结 118

练习 118

第 12 章 二叉树和二叉查找树 119

12.1 树的定义 119

12.2 二叉树 120

12.2.1 构造二叉查找树 120

12.2.2 遍历二叉查找树 121

12.2.3 在二叉查找树中查找节点和最大/最小值 123

12.2.4 从 BST 中移除叶子节点 123

12.2.5 删除带有一个子节点的节点 124

12.2.6 删除带有两个子节点的节点 124

小结 126

练习 127

第13章 集合 128

13·1 集合的基础定义、操作及属性 128

13·1·1 集合的定义 128

13·1·2 集合的操作 128

13·1·3 集合的属性 128

13·2 第一个用散列表的 SET 类的实现 129

13·2·1 类数据成员和构造器方法 129

13·2·2Add 方法 129

13·2·3Remove 方法和 Size 方法 129

13·2·4Union 方法 129

13·2·5Intersection 方法 130

13·2·6Subset 方法 130

13·2·7Difference 方法 130

13·2·8 测试 CSet 实现的程序 130

13·3CSET 类的 BITARRAY 实现 131

13·3·1 使用 BitArray 实现的概述 131

13·3·2BitArray 集合的实现 132

小结 133

练习 133

第 14 章 高级排序算法 134

14·1 希尔排序算法 134

14·2 归并排序算法 135

14·3 堆排序算法　*136*

14·3·1 构造堆　*136*

14·4 快速排序算法　*138*

14·4·1 快速排序算法的描述　*139*

14·4·2 快速排序算法的代码　*139*

14·4·3 快速排序算法的改进　*140*

小结　*140*

练习　*140*

第 *15* 章　查找的高级数据结构和算法　*141*

15·1 AVL 树　*141*

15·1·1 AVL 树的基本原理　*141*

15·1·2 AVL 树的实现 *141*

15·2 红黑树 *143*

15·2·1 红黑树规则 *143*

15·2·2 红黑树的插入 *143*

15·2·3 红黑树实现代码 *144*

15·3 跳跃表 *146*

15·3·1 跳跃表的基本原理 *146*

15·3·2 跳跃表的实现 *147*

小结 *149*

练习 *150*

第 *16* 章 图和图的算法 *151*

16·1 图的定义 *151*

16·2 由图模拟真实世界系统 *151*

16·3 图类 *151*

16·3·1 顶点的表示 *152*

16·3·2 边的表示 *152*

16·3·3 图的构造 *152*

16·3·4 图的第一个应用：拓扑排序 *153*

16·3·5 拓扑排序算法 *154*

16·3·6 拓扑排序算法的实现 *154*

16·4 图的搜索 *156*

16·4·1 深度优先搜索 *156*

16·4·2 广度优先搜索 157

16·5 最小生成树 158

16·5·1 最小生成树算法 158

16·6 查找最短路径 159

16·6·1 加权图 159

16·6·2 确定最短路径的 Dijkstra 算法 160

16·6·3 Dijkstra 算法的代码 160

小结 164

练习 164

第 17 章 高级算法 165

17·1 动态规划 165

17·1·1 动态规划实例：计算斐波纳契数列 165

17·1·2 寻找最长公共子串 167

17·1·3 背包问题 168

17·2 贪心算法 169

17·2·1 贪心算法实例：找零钱问题 169

17·2·2 采用哈夫曼编码的数据压缩 170

17·2·3 用贪心算法解决背包问题 174

小结 176

练习 176

索引 177

第 1 章 Collections 类、泛型类和 Timing 类概述

这本书采用 C#语言来讨论数据结构与算法的开发和实现。书中用到的数据结构都可以在·NET 框架类库 System·Collections 中找到。本章会逐步展开群集的概念，首先是讨论自身特有的 Collection 类（采用数组作为我们实现的基础）的实现，接着会介绍·NET 框架中 Collection 类的内容。

泛型是 C#语言 2·0 版新增加的一个重要补充。泛型允许 C#语言程序员可以独立地或者在一个类中编写函数的某一个版本，而且不需要为了不同的数据类型而多次负载此函数。C#语言 2·0 版还为个别几种 System·Collections 数据结构实现范型提供了一个专门的库 System·Collections·Generic。本章将向读者介绍泛型编程。

本章最后会介绍一种用户定制的类——Timing 类。后续的几个章节将会用此类来衡量数据结构与/或算法的性能。此类将代替大 O 分析法的位置。这不是因为大 O 分析法不重要，而是因为本书采取了一种更为实用的方法来学习数据结构与算法。

1·1 群集的定义

群集是一种结构化的数据类型。它存储数据，并且提供数据的添加、删除、更新操作，以及对群集的不同属性值的设置与返回操作。

群集可以分为两类：线性的和非线性的。线性群集是一张元素列表，表中的元素顺次相连。

线性群集中的元素通常由位置来决定次序（例如，第一个元素、第二个元素、第三个元素，依次类推）。在现实世界中，购物清单就是很好的线性群集实例。而在计算机世界中（当然这也是真实世界）则把数组设计成线性群集。

非线性群集所包含的元素在群集内没有位置次序之分。组织结构图就像用架子垒好的台球一样是一个非线性群集的实例。而在计算机世界中树、堆、图和集都是非线性群集。

无论是线性的还是非线性的群集都拥有一套定义好的属性和操作的集合。其中，属性用来描述群集，而操作就是群集能执行的内容。群集 Count 就是群集属性的一个实例。它保存着群集中数据项的数量。这里把群集的操作称为方法，它包括 Add（即向群集添加新元素），Insert（即在群集指定的索引位置添加新元素）、Remove（即从群集中移除指定元素）、Clear（即从群集中移除所有元素）、Contains（即确定指定元素是否是群集的成员）、以及 IndexOf（即确定指定元素在群集中的索引位置）。

1·2 群集的描述

在两种主要的群集类中有几个子类别。线性的群集可能是直接存取群集，也可能是顺序存取群集。而非线性的群集既可以是层次群集，也可以是组群集。本小节就来讨论这些群集的类型。

1·2·1 直接存取群集

直接存取群集最常见的实例就是数组。这里把数组定义为具有相同数据类型的元素的群集，

而且所有数组元素如同图 *1-1* 说明的那样可以通过整数型索引直接进行存取访问。

（原书 P*3* 页图）

第 *0* 项、第 *1* 项、第 *2* 项、第 *3* 项、…第 j 项、第 n-*1* 项

图 *1-1* 数组

数组可以是静态的，这样当声明数组的时候便于针对程序的长度来固定指定元素的数量。数组也可以是动态的，通过 ReDim 或者 ReDim Preserve 语句就可以增加数组元素的数量。

在 C#语言中，数组不只是内置的数据类型，它还是一种类。在本章的后续部分，当详细分析数组使用的时候将会讨论如何把数组作为类对象来使用。

我们可以用数组来存储一个线性的群集。向数组添加新元素是很容易的，只要简单地把新元素放置在数组尾部第一个空位上就可以了。但是，在数组中插入一个元素就不是这么容易的（或高效）了。因为要给插入的元素空出位置，所以需要按顺序向后移动数组元素。从数组的尾部删除一个元素也是很有效率的操作，只要简单地移除掉最后一个元素的值就可以了。但是，删除数组中任何其他位置上的元素就没有这么有效率了，就像处理插入操作一样，为了保持数组中元素的连续性，可能需要先前调整许多数组元素的位置。这些情况将在本章后续内容中进行讨论。·NET 框架为简化线性群集的编程提供了一种专门的数组类 ArrayList。第 *3* 章将会对此类进行分析研究。

字符串是直接存取群集的另外一种类型。字符串是字符的群集。和存取数组元素的方式一样，也可以基于字符的索引对其进行存取。在 C#语言中，字符串也是作为类对象来实现的。这个类包含一个在字符串上执行标准操作的庞大的方法集合，其中操作有串连接、返回子串、插入字符、移除字符等等。第 *8* 章会讨论 String 类。

C#字符串是不可变的。这意味着一旦对字符串进行了初始化，就不能再改变它了。当要修改字符串的时候，不是改变原始的字符串，而是创建一个字符串的副本。在某些情况下这种行为可能会导致性能下降，所以·NET 框架提供了 StringBuilder 类来让用户能处理可变的字

符串。第 8 章也会对 StringBuilder 进行介绍。

结构（在其他编程语言中也被称为记录）是最后一种直接存取的群集类型。结构是一种复合数据类型。它所包含的数据可能拥有许多不同的数据类型。例如，一名雇员记录就是由雇员的姓名（字符串）、薪水（整数）、工号（字符串或整数），以及其他属性组成的。由于把这些数据的值分别存储在分散的变量内是很容易变混淆的，所以编程语言采用结构来存储此类数据。

C#语言的结构所增加的强大能力就是为执行存储在数据上的操作定义了方法。尽管不能从结构继承或推导出一种新的类型，但是这种做法使得结构在某些地方很像一个类。下面的代码举例明了 C#语言中结构的一个简单应用。

```
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);

    }

}
```

虽然.NET 环境中许多元素都是作为类来实现的（比如数组和字符串），但是语言的几个主要元素还是作为结构来实现的，比如数字数据类型。例如，整数类型就是作为 Int32 结构来

实现的。采用 Int32 的方法之一就是把用字符串表示的数转换成为整数的 Parse 方法。实例如下所示：

```
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);

        }

}
```

1·2·2 顺序存取群集

顺序存取群集是把群集元素按顺序存储的表。这里也把此类群集称为线性表。线性表在创建时没有大小限制，这就意味着它们可以动态地扩展和收缩。不能对线性表中数据项进行直接存取访问，而要像图 1-2 表示的那样通过数据项的位置对其进行存取。线性表的第一个元素在表头的位置，而最后一个元素在表尾的位置。

（原书 P6 页图）

第 1 项、第 2 项、第 3 项、第 4 项、…第 n 项

表头　　表尾

图 1-2 线性表

由于不能直接存取线性表的元素，为了访问某个元素就需要遍历线性表直到到达要找元素的位置为止。线性表的实现通常允许两种遍历表的方法：一种是单向从前往后遍历，而另一种则是双向遍历，即从前向后和从后先前遍历。

线性表的一个简单实例就是购物清单。顺次写下要购买的全部商品就会形成一张购物清单。在购物时一旦找到某种商品就把它从清单中划掉。

线性表既可以是有序的，也可以是无序的。有序线性表具有顺次对应的有序值。如下列人名所表示的情况：Beata、Bernica、David 、Frank、Jennifer、Mike、Raymond、Terrill。而无序线性表则是由无序元素组成的。在第 *2* 章对二叉查找算法与简单线性查找算法进行讨论时就会发现线性表的顺序会在查找表中数据时产生很大的差异。

线性表的某些类型限制访问数据元素。这类线性表有堆栈和队列。堆栈是一种只允许在表头（或顶端）存取数据的表。在表的顶端放置数据项，而且也只能从表的顶端移出数据项。正是基于这种原因，堆栈也被称为后进先出结构。这里把向堆栈添加数据项的操作称为入栈，而把从堆栈移出数据项的操作称为出栈。图 *1-3* 展示了堆栈的这两种操作。

（原书 P*7* 页图）

入栈、出栈

图 *1-3* 堆栈操作

堆栈是非常常见的一种数据结构，特别是在计算机系统编程中尤为普遍。在堆栈的众多应用中，它常用于算术表达式的计算和平衡符号。

队列是一种只允许在表尾进行数据项添加和移出操作的表。它也被称为是先进先出结构。这里把向队列添加数据项称为 EnQueue，而把从队列移出数据项称为 DeQueue。图 *1-4* 展示

了队列的这两种操作。

（原书 P7 页图）

图 1-4 队列操作

队列既可用于调度操作系统任务的系统编程，也可用于模拟研究的编程。在每一种能想象到的少量情况下，队列都可以为模拟等待队列产生极好的结构。优先队列是队列的一种特殊类型。它允许最先移出队列的数据项具有最高的优先级。例如，优先队列可以用来研究医院急诊室的操作，这里应该对心脏病突发患者先进行救护，然后再处理手臂骨折患者。

最后要讨论的一类线性群集被称为通用的索引群集。这类群集的第一种就是散列表。它存储了一组与关键字相关联的数据值。在散列表中有一个被称为散列函数的特殊函数。此函数会取走一个数据值，并且把此数据值（称为关键字）转换成用来取回数据的整数索引。然后此索引会用来存取访问与关键字相关联的数据记录。例如，一条雇员记录可能由雇员姓名、薪水、工作年限以及所工作的部门组成。此结构如图 1-5 所示。此数据记录的关键字就是雇员的姓名。C#语言有一个称为 HashTable 的类用来存储散列表的数据。第 10 章会讨论此结构。

（原书 P8 页图）

“信息系统”部门

图 1-5 散列的记录

另外一种通用的索引群集就是字典。字典也被称为联合，它是由一系列键值对构成的。此结构与词典类似，词典中的词是关键字，而词的定义则是与关键字相关联的值。关键字就是与其相关联的值内的索引。虽然索引不需要就是整数，但是由于上述这种索引方案，所以还是常把字典称为联合数组。第 *11* 章会对·NET 框架内容的几种 Dictionary 类进行讨论。

1·2·3 层次群集

非线性群集分为两大主要类型：层次群集和组群集。层次群集是一组划分了层次的数据项集合。位于某一层的数据项可能会有位于下一较低层上的后继数据项。

树是一种常见的层次群集。树群集看上去像是一棵倒立的树，其中一个数据项作为根，而其他数据值则作为叶子挂在根的下面。树的元素被称为节点，而且在特定节点下面的元素被称为是此节点的孩子。图 *1-6* 展示了一棵实例树。

（原书 P*9* 页图）

根

图 *1-6* 树群集

树在几种不同的领域都有应用。大多数现代操作系统的文件系统都是采用树群集设计而成的，其中一个目录作为根，而其他子目录则作为根的孩子们。

二叉树是树群集的一种特殊类型，树中每个节点最多只有两个孩子。二叉树可以变成二叉查找树，这样做可以极大地提高查找大量数据的效率。实现的方法是依据从根到要存储数据的节点的路径为最短路径的方式来放置节点。

还有一种树类型就是堆。堆这样组织就是为了便于把最小数据值始终放置在根节点上。在删除时会移除根节点。此外，堆的插入和删除操作总是会导致堆的重组，因为只有这样才能把最小值放在根节点上。我们经常会用堆来排序，这被称为是堆排序。通过反复删除根节点以

及重组堆的方式就可以对存储在堆内的数据元素进行排序。

第 *12* 章将对几种不同类型的树进行讨论。

1·2·4 组群集

数据项为无序的非线性群集被称为组。集合、图和网络是组群集的三种主要类型。

集合是一种无序数据值的群集，并且集合中每一个数据值都是唯一的。当然，就像整数一样，班级中学生的列表就是一个集合的实例。在集合上执行的操作包括联合和交叉。图 *1-7* 显示了集合操作的实例。

（原书 P*10* 页图）

A 集合、B 集合、A 集合交叉 B 集合、A 集合联合 B 集和

图 *1-7* 集合操作

图是由节点集合以及与节点相连的边集合组成的。图用来对必须访问图中每个节点的情况进行建模，而且有些时候还要按照特定顺序进行访问。这样做的目的是为了找到“遍历”图的最有效的方法。图可用于，也可用于计算机科学和数学研究领域。大家可能听说过“旅行商”问题。这就是图问题的一种特殊类型。此问题要求在旅行预算允许的条件下为需要拜访路线中所有城市的商人确定最有效的完整旅行路线。此问题的实例图表示在图 *1-8* 中。

（原书 P*10* 页图）

Tokyo*:*东京、Seattle*:*西雅图、LA*:*洛杉矶、Boston*:*波士顿、New York*:*纽约、Washington*:*华盛顿、London*:*伦敦、Paris*:*巴黎、Rome*:*罗马、Moscow*:*莫斯科

图 *1-8* 旅行商问题

此问题是被称为 NP-完全问题的其中一部分内容。这就意味着针对此类型的大量问题是无法知道确切解决方案的。例如，为了找到图 *1-8* 所示问题的解决方案，需要检查 *10* 的阶乘这么多条线路，这等于是 *3628800* 条线路。如果把问题扩展为 *100* 座城市，就需要检查 *100* 的阶乘条线路。就目前方法而言是无法用现在方法实现的。因此需要找到一种近似的解决方案。

网络是图的一种特殊类型。网络的每一条边都被赋予了权。权同使用某边从一个节点移动到另一个节点所花费的代价相关。图 *1-9* 描述了带权的城市网络，其中这里的权是两座城市（节点）之间的英里数。

（原书 P*11* 页图）

图 *1-9* 网络群集

至此已经对将要在本书中讨论的不同群集类型做了总体的概述。下面就准备实际看一看这些群集是如何用 C#语言实现的了。首先会看到如何用来自·NET 框架的抽象类 CollectionBase 类来构建一个 Collection 类。

1·3 CollectionBase 类

·NET 框架库不包括用于存储数据的通用 Collection 类，但是大家可以使用一种抽象的类 CollectionBase 类来构造属于自己的 Collection 类。CollectionBase 类为程序员提供了实现定制 Collection 类的能力。CollectionBase 类隐含实现了两个为构造 Collection 类所必需的接口，即 ICollection 和 IEnumerable，而留给程序员要做的工作就只是对这些作为 Collection

类特殊内容的方法的实现。

1·3·1 用 ArrayLists 实现 Collection 类

本节将要说明如何用 C#语言来实现自身的 Collection 类。这是出于几种目的考虑。首先，如果大家不是很熟悉面向对象编程（OOP），那么这个实现将会展示一些简单的用 C#语言进行面向对象编程的技巧。其次，就如同讨论各种 C#数据结构一样，此节内容还可用于讨论一些将要出现的性能问题。最后，就像本书中其他实现章节一样，本节内容也会使人获益良多，因为仅仅用语言自身的元素就能重新实现已存在的数据结构实在是充满乐趣的事。正如 Don Kunth（计算机科学的先驱之一）所说的那样，也许只有当学成计算机时才会真正学到一些知识。所以，与从日常编程库中选取类来使用相比，通过讲解 C#语言如何实现不同数据结构的方法将会使大家学会更多关于这些结构的知识。

1·3·2 定义 Collection 类

在 C#语言中定义一个 Collection 类最简单的方法就是把在 System·Collections 库中已找到的抽象类 CollectionBase 类作为基础类。此类提供了一套可以实现构造自身群集的抽象方法集合。CollectionBase 类还提供了一种基础的数据结构——InnerList（一个 ArrayList）。此结构可以用作自身类的基础。本章节会看到如何使用 CollectionBase 来构造 Collection 类。

1·3·3 实现 Collection 类

弥补 Collection 类的全部方法包括一些与类的基础数据结构 InnerList 相交互的类型。本节第一部分要实现的方法是 Add 方法、Remove 方法、Count 方法和 Clear 方法。尽管定义的其他方法可以使类更有用，但是上述这些方法是类的绝对基本要素。

首先从 Add 方法开始。这种方法有一个参数，即 Object 变量。此变量用来保存群集要添加的数据项。代码如下所示：

```
public void Add(Object item)
```

```
{

    InnerList.Add(item);

}
```

ArrayList 把数据作为对象（即 Object 数据类型）来存储。这就是把数据项声明为 Object 的原因。第 2 章将会学到更多有关 ArrayLists 的内容。

Remove 方法的执行与上述类似：

```
public void Remove(Object item)

{

    InnerList.Remove(item);

}
```

接下来是 Count 方法。Count 最常作为属性来实现，但是这里更喜欢把它用作方法。而且，由于是在基础类 CollectionBase 中实现 Count，所以必须用新的关键词来隐藏在 CollectionBase 中找到的 Count 的定义：

```
public new int Count()

{
```

```
    return InnerList.Count;
}
```

Clear 方法把全部数据项从 InnerList 中移除掉。这里也需要在方法定义中使用新的关键词：

```
public new void Clear()
{
    InnerList.Clear();
}
```

了解这些内容足够开始了。下面来看一个用 Collection 类且带有完整类定义的程序：

```
using System;
using System.Collections;
public class Collection : CollectionBase
```

```
{

    public void Add(Object item)

    {

        InnerList.Add(item);

    }

    public void Remove(Object item)

    {

        InnerList.Remove(item);

    }

    public new void Clear()

    {
```

```
            InnerList.Clear();

        }

        public new int Count()

        {

            return InnerList.Count;

        }

}

class chapter7

{

        static void Main()

        {
```

```
Collection names = new Collection();

names.Add("David");

names.Add("Bernica");

names.Add("Raymond");

names.Add("Clayton");

foreach (Object name in names)

{

    Console.WriteLine(name);

}

Console.WriteLine("Number of names: " + names.Count());

names.Remove("Raymond");
```

```
            Console·WriteLine("Number of names: " + names·Count());

            names·Clear();

            Console·WriteLine("Number of names: " + names·Count());

        }

}
```

为了创建一个更加有用的 Collection 类，还可以实现几种其他的方法。大家可以在练习中实现一些这样的方法。

1·4 范型编程

面向对象编程的问题之一就是所谓“代码膨胀”的特征。为了说明方法参数所有可能的数据类型而需要重载某种方法或重载一套方法集合的时候，就会发生某种类型的代码膨胀。代码膨胀的解决方案之一就是使某个值呈现多种数据类型的能力，同时仅提供此值的一种定义。这种方法被称为是范型编程。

范型编程提供数据类型“占位符”。它在编译时由特定的数据类型填充。这个占位符用一对尖括号（< >）和放在括号间的标识符来表示。下面来看一个实例。

范型编程第一个规范实例就是 Swap 函数。下面是 C#语言中范型 Swap 函数的定义：

```
static void Swap<T>(ref T val1, ref T val2)

{

    T temp;

    temp = val1;

    val1 = val2;

    val2 = temp;

}
```

立刻把数据类型占位符放置在函数名后边。现在无论何时需要范型数据类型都可以使用放置在尖括号中的标识符了。就像用于交换的临时变量一样，每个参数都会获得一个范型数据类型。下面就是一个测试此代码的程序：

```
using System;

class chapter1

{
```

```
static void Main()

{

    int num1 = 100;

    int num2 = 200;

    Console·WriteLine("num1: " + num1);

    Console·WriteLine("num2: " + num2);

    Swap<int>(ref num1, ref num2);

    Console·WriteLine("num1: " + num1);

    Console·WriteLine("num2: " + num2);

    string str1 = "Sam";

    string str2 = "Tom";
```

```
        Console·WriteLine("String 1: " + str1);

        Console·WriteLine("String 2: " + str2);

        Swap<string>(ref str1, ref str2);

        Console·WriteLine("String 1: " + str1);

        Console·WriteLine("String 2: " + str2);

}

static void Swap<T>(ref T val1, ref T val2)

{

        T temp;

        temp = val1;

        val1 = val2;
```

```
        val2 = temp;

    }

}
```

程序的输出如下所示：

（原书P16页截图）

范型对函数定义没有限制。所以也可以创建范型类。范型类的定义包括一个跟在类名后边的范型类型占位符。任何定义中引用类名的时候都必须提供类型占位符。下面的类定义说明了创建范型类的方法：

```
public class Node<T>

{

    T data;
```

```
    Node<T> link;

    public Node(T data, Node<T> link)

    {

        this.data = data;

        this.link = link;

    }

}
```

可以按照如下形式使用此类：

```
Node<string> node1 = new Node<string>("Mike", null);

Node<string> node2 = new Node<string>("Raymond", node1);
```

本书讨论到的几种数据结构都将采用 Node 类。

虽然范型编程的这种用法可能是十分有用的，但是 C#语言提供了备用的范型数据结构库。

在 System·Collection·Generics 命名空间中都可以找到这些数据结构，而且在讨论作为此命名空间内容的数据结构的时候，还将对它的使用做分析。虽然通常情况下这些类具有和非范型数据结构类相同的功能性，但是由于其他方法及其用途没有什么不同，所以通常会为了如何实例化类的对象而限制范型类的讨论。

1·5 时间测试

由于本书采用了一种实用的方法来分析数据结构与算法检测，所以这里避开使用大 O 分析法，而采用运行简单基准测试的方式来代替。这种测试将会说明运行一段代码需要多少秒数（或者无论什么时间单位）。

基准法测试是用时间测试的方式来衡量运行完整算法所花费的时间长度。如同科学一样，基准测试也像是一门艺术，而且为了获得精确分析需要很小心测试代码的方法。下面就来进行详细讨论。

1·5·1 一个简单化的时间测试

首先时间测试需要一些代码。出于简单的考虑，这里将测试一个有关控制台数组内容的子程序。代码如下所示：

```
static void DisplayNums(int[] arr)

{

for (int i = 0; i <= arr·GetUpperBound(0); i++)
```

```
    Console·Write(arr[i] + " ");

}
```

数组的初始化放在了程序的另外一部分，这部分稍后再进行研究。

为了测试这个子程序，需要创建一个变量，并且把子程序调用时的系统时间赋值给此变量。此外，还需要一个变量用来存储子程序返回时的时间。根据这些内容写出了下述这段代码：

```
DateTime startTime;

TimeSpan endTime;

startTime = DateTime·Now;

endTime = DateTime·Now·Subtract(startTime);
```

在作者笔记本（运行环境：机器主频 1·4mHz，操作系统 Windows XP 专业版）上运行此代码时，子程序的运行时间大约为 5 秒左右（4·9917 秒）。虽然这段代码对执行时间测试好像很有道理，但是在·NET 环境下运行时间代码是完全不合适的。为什么呢？

首先，代码测量的是从子程序调用开始到子程序返回主程序之间流失的时间。但是测试所测量的时间也包含了与 C#程序同时运行的其他进程所用的时间。

其次，时间代码不考虑·NET 环境下执行的无用单元收集。在类似·NET 这样的运行时间环境中，系统可能在执行无用单元收集的任何一个时间暂停。时间代码实例没有考虑无用单元收集时间，以及很容易受无用单元收集影响的结果时间。那么到底应该怎么做呢？

1·5·2 用于·NET 环境的时间测试

在·NET 环境中需要考虑程序运行中的线程以及无用单元收集可能在任何时候发生的事实。所以在编写时间测试代码时需要考虑这些情况。

先来看一下如何处理无用单元收集。首先讨论一下无用单元收集的用途。C#语言用有时被称为堆的内存来给参考类型（例如字符串、数组以及类事例对象）分配存储空间。堆是用来保存数据项（前面提到的类型）的内存区域。诸如普通变量这样的值类型则存储在堆栈中。引用的参考数据也存储在堆栈中，但是实际的数据则是以参考类型的形式存储在堆中。

当声明变量的子程序完全执行结束时就可以释放掉存储在堆栈中的变量。而另一方面，存储在堆中的变量则会一直保留到调用无用单元收集进程的时候。当没有引用堆数据的行为时，只有通过无用单元收集才可以移除这些数据。

在程序执行过程中无用单元收集可能会发生在任何时候。然而需要确保在实现时间测试代码时没有运行无用单元收集器。但是也许大家听说过通过强制调用无用单元收集器来进行专门的无用单元收集。·NET 环境为执行无用单元收集调用提供了专门的对象——GC。为了使系统执行无用单元收集，可以有如下简单书写：

```
GC·Collect();
```

但是不是所有都要这样做的。存储在堆中的每一个对象都有一个称为 finalizer 的专门方法。finalizer 方法是在删除对象之前执行的最后一步。有关 finalizer 方法的问题是，这些方法不是按照系统方式运行的。事实上，甚至无法确信对象的 finalizer 方法是否真的执行了，但是知道在确定删除对象之前需要执行此对象的 finalizer 方法。为了确信这一点，我们添加了一行代码来告诉程序等待堆上对象的所有 finalizer 方法都运行后再继续。此代码行如下：

```
GC·WaitForPendingFinalizers( );
```

已经清除了一个障碍，现在就剩下一个问题了——采用正确的线程。在·NET 环境中，程序运行在被称为应用程序域的进程中。这就允许操作系统在同一时间内分开运行每个不同的程序。在进程内，程序或程序的一部分是在线程内运行的。操作系统通过线程来分配程序的执行时间。在用时间测试程序代码时，需要确信正在进行时间测试的代码就在为自身程序分配的进程中，而不在操作系统执行的其他任务里。

在·NET 框架下通过使用 Process 类可以做到这一点。Process 类拥有的方法允许选取当前的进程、选取程序运行其内的线程，以及选取存储线程开始执行时间的计时器。这些方法中的每一个都可以合并成一个调用。此调用会把它的返回值赋值给一个变量用来存储开始时间（TimeSpan 对象）。如下列代码所示（没错，就是两行代码）：

```
TimeSpan startingTime;

startingTime = Process·GetCurrentProcess()·Threads[0]·UserProcessorTime;
```

剩下要做的就是在进行时间测试的代码段停止时捕获时间。做法如下：

```
duration =
Process·GetCurrentProcess()·Threads[0]·UserProcessorTime·Subtract(startingTime);
```

现在把所有这些合并成一个程序。此程序的代码和先前测试代码是一样的：

```
using System;

using System·Diagnostics;
```

```
class chapter7

{

    static void Main()

    {

        int[] nums = new int[100000];

        BuildArray(nums);

        TimeSpan duration;

        DisplayNums(nums);

        DisplayNums(nums);

        DisplayNums(nums);

duration = Process.GetCurrentProcess().TotalProcessorTime;
```

```
        Console·WriteLine(“Time: “ + duration·TotalSeconds);

}

static void BuildArray(int[] arr)

{

        for (int i = 0; i <= 99999; i++)

                arr[i] = i;

}

static void DisplayNums(int[] arr)

{

        for (int i = 0; i <= arr·GetUpperBound(0); i++)

                Console·Write(arr[i] + “ “);
```

```
    }

}
```

采用新改进的时间测试代码后，程序的返回值为 0·2526。把此数值与先前第一版时间测试代码返回的将近 5 秒的数值进行比较。很明显，这两种时间测试方法之间存在显著差异。因而·NET 环境中的时间测试代码应该使用·NET 方法来做。

1·5·3Timing Test 类

虽然不需要一个类来运行时间测试代码，但是把代码作为类来重写是有意义的，主要原因是如果能够减少测试的代码行数量，就能保证代码的清晰。

Timing 类需要下列数据成员：

- startingTime——用来存储正在测试的代码的开始时间。
- duration——用来存储正在测试的代码的终止时间。

straingTime 和 duration 这两个成员用来存储时间，而且为这些数据成员选择使用 TimeSpan 数据类型。这里就采用一种构造器方法，此默认构造器会把数据成员全部置为 0。

正如看到的那样，Timing 类是很小的，它只需要少量方法。下面是定义：

```
public class Timing

{
```

```
TimeSpan startingTime;

TimeSpan duration;

public Timing()

{

    startingTime = new TimeSpan(0);

    duration = new TimeSpan(0);

}

public void StopTime()

{

    duration =

    Process·GetCurrentProcess()·Threads[0]·
```

```
        UserProcessorTime·Subtract(startingTime);




}

public void startTime()

{

        GC·Collect();

        GC·WaitForPendingFinalizers();

        startingTime =

        Process·GetCurrentProcess()·Threads[0]·

        UserProcessorTime;

}
```

```
    public TimeSpan Result()
    {
        return duration;
    }
}
```

这是用 Timing 类改写的用来测试 DisplayNums 子程序的程序：

```
using System;
using System.Diagnostics;
using System.Threading;
public class Timing
{
```

```
TimeSpan duration;

public Timing()

{

    duration = new TimeSpan(0);

}

public void stopTime()

{

    duration = Process.GetCurrentProcess().TotalProcessorTime;

}

public void startTime()

{
```

```
            GC·Collect();

            GC·WaitForPendingFinalizers();

        }

        public TimeSpan Result()

        {

            return duration;

        }

}

class chapter7

{

        static void Main()
```

```
{

    int[] nums = new int[100000];

    BuildArray(nums);

    Timing tObj = new Timing();

    tObj·startTime();

    DisplayNums(nums);

    tObj·stopTime();

    Console·WriteLine("time (·NET): " + tObj·Result()·TotalSeconds);

}

static void BuildArray(int[] arr)

{
```

```
        for (int i = 0; i < 100000; i++)

            arr[i] = i;

    }

    static void DisplayNums(int[] arr)

    {

        for (int i = 0; i <= arr·GetUpperBound(0); i++)

            Console·Write(arr[i] + " ");

    }

}
```

通过把时间测试代码移动到类里的方法，这里把主程序的代码行数从 13 行消减为 8 行。显然这样不会从程序中砍掉大量的代码，而比砍掉代码更重要的则是降低了主程序的复杂度。如果没有类，那么把开始时间赋值给变量的操作就会像下面这样：

```
startTime = Process·GetCurrentProcess( )·Threads[0]·UserProcessorTime;
```

而如果使用 Timing 类，那么把开始时间赋值给类数据成员的方式如下所示：

```
tObj·startTime( );
```

通过把冗长的赋值语句封装到类方法内，可以使得代码更便于阅读而且出错的可能更小了。

小结

本章对此书中经常会用到的三种重要技术进行了回顾。尽管不需要编写整个程序，但是一些程序的代码以及要讨论的库都采用面向对象的方式来编写。自行开发的 Collection 类说明了许多基本面向对象的概念，而且这些概念看似贯穿全书。范型编程允许程序员通过限制需要编写或重载的方法的数量来简化几种数据结构的定义。Timing 类提供了简单有效的方法来衡量所要学习的数据结构与算法的性能。

练习

1· 请创建一个名为 Test 的类。此类包含的数据成员有学生姓名和描述试卷编号的一个整数。这个类会在下述情况下使用：当学生提交测试时，他们会把试卷面朝下放到桌子上。如果某位学生需要检查自己试卷的答案，那么老师就需要把试卷堆反过来以便第一份试卷在上面。然后从第一份试卷开始顺次查找，直到找到需要的试卷。随后，就把找到的试卷从试卷堆中取出来。当学生检查完自己的试卷后，再把此试卷重新放到试卷堆的末尾。下面请编写一个窗口应用程序来模拟这种情况。程序包含用户录入姓名和试卷编号的文本框。还要有一个格式列表框用来显示试卷的最终列表。应用窗口需要提供四个操作按钮：*1·*提交试卷；*2·*学生查看试卷；*3·*返回一份试卷；以及 *4·*退出。请执行下列操作来测试你的应用程序：*1·*录入某姓名和试卷编号。并且把试卷插入到名为 submittedTests 的群集里。*2·*录入某姓名，从 submittedTests 中删除相关试卷，并且把此试卷插入到名为 outForChecking 的群集里。*3·*录入学生姓名，从 outForChecking 中删除相应试卷，并且把此试卷插入到 submittedTests

中。4,点击退出按钮。退出按钮不会终止应用程序，而是从 outForChecking 中删除所有试卷，并且把它们全部插入到 submittedTests 中，同时显示所有已提交的试卷列表。

2· 请对 Collection 类添加下列方法：

a· Insert

b· Contains

c· IndexOf

d· RemoveAt

3· 请使用 Timing 类来比较向 Collection 类和 ArrayList 类分别添加了 100000 个整数时的性能。

4· 请构建属于自己的 Collection 类，并且此类不是由 CollectionBase 派生而来的。请在实现中使用范型。

数组是最通用的数据结构，它出现在几乎所有的编程语言里。在C#语言中使用数组包括创建 System·Array 类型的数组对象，以及创建针对所有数组的抽象的基类型。Array 类提供了一套方法，这些方法是为了执行诸如排序和查找这类过去需要程序员手工实现的任务。

C#语言中另外一种使用数组的有趣替换方式就是 ArrayList 类。ArrayList 是一种像要更多空间来动态生长的数组。对于无法精确知道数组最终大小的情况，或者对于程序生命周期内数组大小可能会发生一点变化的情况，用 ArrayList 比用数组更合适。

本章将简要介绍 C#语言中使用数组的基本概念，然后继续展开更加深入的主题，这其中包括复制、克隆、相等判定，以及使用 Array 类和 ArrayList 类的静态方法。

2·1 数组基本概念

数组是可索引的数据的集合。数据既可以是内置的类型，也可以是用户自定义的类型。事实上，把数组数据称为对象大概是最简便的方式。C#语言中的数组实际上就是对象本身，因为它们都来源于 System·Array 类。既然数组是 System·Array 类的一个声明实例，所以在使用数组时也可以使用此类的所有方法和属性。

2·1·1 数组的声明和初始化

这里采用下列语法规则对数组进行声明：

type*[ ]* array-name*;*

这里的类型就是数组元素的数据类型。下面是一个实例：

```
string[ ] names;
```

接下来一行需要实例化数组（既然它是 System·Array 类型的一个对象），还需要确定数组的大小。下面这行就实例化了刚声明的 name 数组，并且预留了五个字符串的内存空间：

```
names = new string[10];
```

必要时还可以把上述这两条语句合并成为一条语句：

```
string[ ] names = new string[10];
```

当想要在一条语句中对数组进行声明、例示以及赋值操作时都要花费时间。在 C#语言中可以采用初始化列表的方式来实现：

```
int[ ] numbers = new int[ ] {1, 2, 3, 4, 5};
```

上述这个数的列表被称为是初始化列表。它用一对大括号作为界定符，并且每个元素之间用逗号进行分割。当用这种方法来声明数组时，不需要指定元素的个数。编译器会通过初始列表中数据项的数量来推断出此数据。

2·1·2 数组元素的设置和存取访问

存储数组元素既可以采用直接存取访问的方法也可以通过调用 Array 类的 SetValue 方法。直接存取方式通过赋值语句左侧的索引来引用数组位置：

```
nNames[2] = "Raymond";
```

```
sSales[19] = 23123;
```

而 SetValue 方法则提供了一种更加面向对象的方法来为数组元素赋值。这种方法会取走两个参数，一个是索引数，另一个则是元素的值。

```
names·SetValue("Raymond", 2);
```

```
sales·SetValue(23123, 9);
```

数组元素的访问既可以通过直接存取的方式也可以通过调用 GetValue 方法的方式。GetValue 方法取走单独一个参数——即索引。

```
myName = names[2];
```

```
monthSales = sales·GetValue([19)];
```

为了存取每一个数组元素用 For 循环来循环遍历数组是一种通用的方法。程序员在编写循环时常犯的错误即可能是写死循环的上限值（如果数组是动态的，那么这样做就是错误的，因为循环的上限可能会改变），也可能是每次循环重复时调用函数来存取循环的上限：

```
(for( int i = 0; i <= sales·GetUpperBound(0); i++)
```

```
totalSales = totalSales + sales[i];
```

2·1·3 取回数组元数据的方法和属性

Array 类为取回数组元数据提供了几种属性：

l Length：返回数组所有维数内元素的总数量。

l GetLength：返回数组指定维数内元素的数量。

l Rank：返回数组的维数。

l GetType：返回当前数组实例的类型。

Length 方法对于计算多维数组中元素的数量以及返回数组中元素的准确编号都是很有用的。另外，还可以使用 GetUpperBound 方法，而且要对数值加一。

既然 Length 返回数组元素的总数量，所以 GetLength 方法统计了数组某一维内元素的数量。这种方法和 Rank 属性一起可用来在运行时调整数组的大小，而且不必冒丢失数据的风险。此方法将在本章的后续内容中进行讨论。

在无法确定数组类型的情况下，GetType 方法可以用来确定数组的数据类型，比如数组作为参数传递给方法的时候。在下列代码段中，为了确定对象是否是数组，这里创建了一个类型变量 Type，此变量允许用来调用类方法 IsArray。如果对象是一个数组，那么代码返回数组的数据类型。

```
int[] numbers;
```

```
numbers = new int[] { 0, 1, 2, 3, 4 };
```

```
Type arrayType = numbers·GetType();

if (arrayType·IsArray)

Console·WriteLine("The array type is: {0}", arrayType);

else

    Console·WriteLine("Not an array");

    Console·Read();
```

Gettype 方法不仅返回数组的类型，而且还让大家明白对象确实是一个数组。下面是代码的输出：

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

这里的方括号说明对象是一个数组。还需要注意在显示数据类型的时候采用了一种格式。这里必须这么做，因为要把 Type 数据与显示的字符串的剩余部分相连接就不能把 Type 数据转变成为字符串。

2·1·4 多维数组

到目前为止的讨论只限于一维数组的情况。在 C#语言中，尽管数组多于三维的情况是非常

少见的（而且也是非常容易使人混乱的），但是数组还是可以达到 32 维的。

通过提供数组每一维上限值的方式可以声明多维数组。二维数组的声明：

```
int [ , ] grades = new int [4,5] ;
```

此语句声明了一个 4 行 5 列的数组。二维数组经常用来模拟矩阵。

声明多维数组也可以不指定维数的上限值。要想这样做就要用到逗号来明确数组的维数。例如，声明一个二维数组如下所示

```
double [ , ] Sales ;
```

再比如声明一个三维数组，

```
double [ , , ] Sales ;
```

在声明不带维数上限的数组的时候，需要稍后对具有这类上限的数组重新确定维数：

```
sales = new double [4,5] ;
```

对多维数组可以用初始化表进行初始化操作。请看下面这条语句：

```
lint[,] grades = new int[,]
```

```
{

{1, 82, 74, 89, 100},

{2, 93, 96, 85, 86},

{3, 83, 72, 95, 89},

{4, 91, 98, 79, 88}

};
```

首先要注意这里没有指明数组的上限。当初始化带有初始化表的数组的时候，不用说明数组的上限。编译器会根据初始化表中数据计算出每一维的上限值。初始化表本身也像数组的每一行那样用大括号进行标记。数组行内的每一个元素则用逗号进行分割。

存取访问多维数组元素的方法类似于存取一维数组元素的方法。大家可以采用传统的数组存取访问方式，

```
grade = gGrades[2,2];

gGrades([2,2]) = 99
```

也可以采用 Array 类的方法：

```
grade = Grades·GetValue([0,2)];
```

但是，对多维数组不能使用 SetValue 方法。这是因为这种方法只接收两个参数：一个数值和一个单独的索引。

尽管常常是基于存储在数组行中的数值或者是基于存储在数组列中的数值进行计算，但是对多维数组上所有元素的计算还是很常见的操作。假设有一个 Grades 数组，且数组的每一行是一条学生记录，那么就能如下所示计算出每个学生的平均成绩：

```
int[,] grades = new int[,]

{

{1, 82, 74, 89, 100},

{2, 93, 96, 85, 86},

{3, 83, 72, 95, 89},

{4, 91, 98, 79, 88}

};
```

```
int last_grade = grades.GetUpperBound(1);

double average = 0.0;

int total;

int last_student = grades.GetUpperBound(0);

for(int row = 0; row <= last_student; row++)

{

total = 0;

for (int col = 0; col <= last_grade; col++)

total += grades[row, col];

average = total / last_grade;

Console.WriteLine("Average: " + average);
```

```
}
```

2·1·5 参数数组

大多数的方法定义要求一套提供给方法的参数的数目，但是想要编写一个允许可选参数数目的方法定义是需要时间的。用一种称为参数数组的构造就可以做到。

通过使用关键字 ParamArray 就可以在方法定义的参数列表中指明参数数组。下面的方法定义允许提供任意数量的数作为参数，并且方法会返回数的总量：

```
static int sumNums(params int[] nums)

{

int sum = 0;

for (int i = 0; i <= nums·GetUpperBound(0); i++)

    sum += nums[i];

return sum;

}
```

此方法可以处理下列任意一种调用：

```
total = sumNums(1, 2, 3);
```

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

当用参数数组定义方法的时候，为了使编译器能够正确处理参数列表，需要在参数列表的最后提供参数数组的参数。否则，编译器无法知道参数数组元素的截止位置以及方法其他参数的起始位置。

2.1.6 锯齿状数组

在创建一个多维数组的时候，需要始终新建一种每行都有相同元素数量的结构。例如，下面这个数组的声明：

```
int sales[,] = new int[12,30]; /// Sales for each day of each month
```

这个数组假设每行（即月数）都有相同的元素（即天数）数量，但是大家知道某些月有 30 天，而某些月是 31 天，还有一个月是 29 天。因而，这个刚刚声明的数组会有几个空元素在其中。对于这个数组而言，这不是太大的问题，但是对于更加庞大的数组而言，就需要减少大量浪费的空间。

解决这个问题的方法是用锯齿状数组代替二维数组。锯齿状数组是一种每行都能组成一个数组的数组。锯齿状数组的每一维就是一个一维数组。大家称其为“锯齿状”数组的原因是由于数组每一行元素的数量都可能不相同。锯齿状数组的图形不是正方形或矩形，而是具有不均匀边缘或锯齿边缘的图形。

锯齿状数组的声明需要通过在数组变量名后放置两组方括号的方式来完成。第一组方括号说明了数组的行数。第二组方括号则是留白的。这为存储在每行内的一维数组标记了位置。通常情况下，声明语句的初始化列表会设置行数，就像下列这样：

```
int[][] jagged = new int[12][];
```

这条语句看上去很奇怪，但是把它分解后就一目了然了。jagged 是一个有着 12 个元素的整数数组，其中的每个元素又是一个整数数组。初始化列表实际上就是对数组行的初始化，这表明每一个行元素都是一个有着 12 个元素的数组，而且每个元素都初始化为默认的值。

一旦声明了锯齿状的数组，就可以分别对各自行数组的元素进行赋值操作了。下面这段代码对 jaggedArray 进行了赋值操作：

```
jagged[0][0] = 23;

jagged[0][1] = 13;

. . .

jagged[7][5] = 45;
```

第一组方括号说明了行编号，而第二组方括号则表明了行数组的元素。第一条语句存取访问到第一个数组的第一个元素，接着第二条语句存取访问了第一个数组的第二个元素，而第三条语句存取访问的则是第八个数组的第六个元素。

为了做一个使用锯齿状数组的实例，下边这段程序创建了一个名为 sales 的数组（用来跟踪

两个月内每星期的销售情况），并且把销售额赋值给数组的元素，然后循环遍历整个数组从而计算出存储在数组内的每月一个星期的平均销售额。

```
using System;

class class1

{

    static void Main()[]

    {

        int[] Jan = new int[31];

        int[] Feb = new int[29];

        int[][] sales = new int[][] { Jan, Feb };

        int month, day, total;

        double average = 0·0;
```

```
sales[0][0] = 41;

sales[0][1] = 30;

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(month = 0; month <= 1; month++)

{

    total = 0;

    for(day = 0; day <= 6; day++)

    {

        total += sales[month][day];
```

```
            }

            average = total / 7;

            Console.WriteLine("Average sales for month: " +month + ": " + average);

        }

    }

}
```

2.2ArrayList 类

当无法提前知道数组的大小或者在程序运行期间数组的大小可能会发生改变的时候，静态数组就不是很适用了。这类问题的一种解决方案就是当数组超出存储空间的时使用能够自动调整自身大小的数组类型。这种数组被称为是 ArrayList。它是.NET 框架库中 System.Collections 命名空间的内容。

ArrayList 对象拥有可存储数组大小尺寸的 Capacity 属性。该属性的初始值为 16。当 ArrayList 中元素的数量达到此界限值时，Capacity 属性就会为 ArrayList 的存储空间另外增加 16 个元素。在数组内元素数量有可能扩大或缩小的情况下使用 ArrayList 会比用带标准数组的 ReDim Preserver 更加有效。

就像第 1 章讨论过的那样，ArrayList 用 Object 类型来存储对象。如果需要强类型的数组，

就应该采用标准数组或者其他一些数据结构。

2·2·1 ArrayList 类的成员

ArrayList 类包含几种用于 ArrayList 的方法和属性。下面这个列表就是最常用到的一些方法和属性：

- Add( )：向 ArrayList 添加一个元素。
- AddRange( )：在 ArrayList 末尾处添加群集的元素。
- Capacity：存储 ArrayList 所能包含的元素的数量。
- Clear( )：从 ArrayList 中移除全部元素。
- Contains( )：确定制定的对象是否在 ArrayList 内。
- Copy To( )：把 ArrayList 或其中的某一段复制给一个数组。
- Count：返回 ArrayList 中当前元素的数量。
- GetEnumerator( )：返回迭代 ArrayList 的计数器。
- GetRange( )：返回 ArrayList 的子集作为 ArrayList。
- IndexOf( )：返回指定数据项首次出现的索引。

- Insert( )：在 ArrayList 的指定索引处插入一个元素。

- InsertRange( )：从 ArrayList 指定索引处开始插入群集的元素。

- Item( )：在指定索引处获取或者设置一个元素。

- Remove( )：移除指定数据项的首次出现。

- RemoveAt( )：在指定索引处移除一个元素。

- Reverse( )：对 ArrayList 中元素的顺序进行反转。

- Sort( )：对 ArrayList 中的元素按照阿拉伯字母表顺序进行排序。

- ToArray( )：把 ArrayList 的元素复制给一个数组。

- TrimToSize( )：为 ArrayList 中元素数量设置 ArrayList 的容量。

2·2·2 应用 ArrayList 类

ArrayList 的使用不同于标准数组。除非事出有因要把数据项添加到特殊位置上，否则通常情况下使用 Add 方法只是向 ArrayList 添加数据项，而对于上述特殊情况就要采用 Insert 方法来进行操作了。本节会讨论如何使用这些操作及 ArrayList 类的其他成员。

首先要做的事情就是如下所示那样声明 ArrayList：

```
ArrayList grades = new ArrayList();
```

注意此声明中使用到了构造器。如果 ArrayList 没有声明使用构造器，那么在后续程序语句里就无法获得对象。

用 Add 方法把对象添加给 ArrayList。此方法会取走一个参数，即添加给 ArrayList 的对象。Add 方法也会返回一个整数用来说明 ArrayList 中被添加元素的位置，当然这个值是很少会在程序中用到的。下面是一些实例：

```
grades·Add(100);
```

```
grades·Add(84);
```

```
int position;
```

```
position = grades·Add(77);
```

```
Console·WriteLine("The grade 77 was added at position:" + position);
```

用 For Each 循环可以把 ArrayList 中的对象显示出来。ArrayList 有一个内置计数器用来记录循环遍历 ArrayList 内所有对象的次数，而且是每次一个。下面这段代码就说明了对 ArrayLsit 使用 For Each 循环的方法：

```
int total = 0;
```

```
double average = 0.0;

foreach (Object grade in grades)

total += (int)grade;

average = total / grades.Count;

Console.WriteLine("The average grade is: " + average);
```

如果需要在 ArrayList 某个特殊位置上添加元素，则可以采用 Insert 方法。此方法会取走两个参数：插入元素的索引，以及要插入的元素。下面这段代码为了保持 ArrayList 内对象的次序而在指定位置上插入了两个成绩：

```
grades.Insert(1, 99);

grades.Insert(3, 80);
```

通过调用 Capacity 属性可以检查 ArrayList 当前的容量，而通过调用 Count 属性可以确定 ArrayList 中元素的数量：

```
Console.WriteLine("The current capacity of grades is:" + grades.Capacity);

Console.WriteLine("The number of grades in grades is:" + grades.Count);
```

这里有几种从 ArrayList 中移除数据项的方法。如果知道要移除的数据项，但又不确定它所处的位置，那么就可以采用 Remove 方法。此方法会取走一个参数，即要从 ArrayList 中移除的对象。如果 ArrayList 内有这个对象，就可以把它移除掉。如果此对象不在 ArrayList 内，那就什么也做。当使用像 Remove 这样的方法时，典型做法是把方法放置在 If-Then 语句内进行调用，并且使用诸如 Contains 这样的方法来验证对象确实存在 ArrayList 内。下面是一个代码段实例：

```
if (grades·Contains(54))

grades·Remove(54)

else

Console·Write("Object not in ArrayList·");
```

如果知道所要移除数据项的索引，那么可以使用 RemoveAt 方法。此方法会取走一个参数，即要移除对象的索引。唯一能接受的人为错误就是给方法传递一个无效的索引。此方法的工作形式如下所示：

```
grades·RemoveAt(2)
```

通过调用 IndexOf 方法可以确定 ArrayList 中某个对象的位置。这种方法会取走一个参数，即一个对象，然后返回此对象在 ArrayList 内的位置。如果对象不在 ArrayList 内，那么方法就会返回-1。下面这段代码把 IndexOf 方法与 RemoveAt 方法结合在一起使用：

```
        int pos;

pos = grades·IndexOf(70);
```

```
grades.RemoveAt(pos);
```

除了向 ArrayList 中添加单独的对象，还可以添加对象的范围。对象必须存储在来源于 ICollection 的数据类型里面。这就意味着可以把对象存储在数组里，或存储在 Collection 里，甚至是存储到另一个 ArrayList 里面。

有两种不同的方法可以用来给 ArrayList 添加范围。它们是 AddRange 方法和 InsertRange 方法。AddRange 方法会把对象的范围添加到 ArrayList 的末尾处，而 InsertRange 方法则会把范围添加到 ArrayList 内指定的位置上。

下面这段程序说明了如何使用这两种方法：

```
using System;

using System.Collections;

class class1

{

    static void Main()

    {
```

```
ArrayList names = new ArrayList();

names.Add("Mike");

names.Add("Beata");

names.Add("Raymond");

names.Add("Bernica");

names.Add("Jennifer");

Console.WriteLine("The original list of names: ");

foreach (Object name in names)

    Console.WriteLine(name);

Console.WriteLine();

string[] newNames = new string[] { "David", "Michael" };
```

```
ArrayList moreNames = new ArrayList();

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: ");

foreach (Object name in names)

    Console.WriteLine(name);
```

```
    }

}
```

此程序输出是：

```
Dvid

Michael

Mike

Bernica

Beata

Raymond

Jennifer

Terrill

Donnie

Mayo

Clayton
```

Alisa

因为指定的索引为 0，所以是在 ArrayList 开始处添加了前两个名字。而后面的几个名字由于使用了 AddRange 方法而被添加到了末尾处。

许多程序员还找到了另外两种非常有用的方法 ToArray 方法和 GetRange 方法。GetRange 方法会返回来自 ArrayList 的对象的范围作为另外一个 ArrayList。而 ToArray 方法则会把 ArrayList 的所有元素复制给一个数组。首先来看一看 GetRange 方法。

GetRange 方法会取走两个参数：起始索引以及要从 ArrayList 找回的元素数量。GetRange 方法没有破坏性，因为这只是把对象从原始 ArrayList 复制给新的 ArrayList。下面这个实例采用和上述相同的程序来说明此方法的工作原理：

```
ArrayList someNames = new ArrayList();

someNames = names·GetRange(2, 4);

Console·WriteLine("someNames sub-ArrayList: ");

foreach (Object name in someNames)

Console·WriteLine(name);
```

这个程序段的输出是：

Mike

Bernica

Beata

Raymond

ToArray 方法允许把 ArrayList 的内容轻松传递给一个标准数组。采用 ToArray 方法的主要原因就是由于用户需要更快的数组存取访问速度。

ToArray 方法不带参数，但是会把 ArrayList 的元素返回给数组。下面这个例子就说明了此方法的使用原理：

```
Object[] arrNames;

arrNames = names.ToArray();

Console.WriteLine("Names from an array: ");

for(int i = 0; i <= arrNames.GetUpperBound(0); i++)

Console.WriteLine(arrNames[i]);
```

这段代码的后半部分证明了确实把来自 ArrayList 的元素存储到了数组 arrNames 里面。

小结

数组是计算机编程中最常采用的数据结构。即使不是全部也是绝大多数的编程语言都会提供一些内置数组类型。对许多应用程序而言，数组是最容易实现的数据结构，也是最有效率的数据结构。数组对于需要直接存取访问数据集合“偏远”元素的情况是非常有用的。

·NET 框架介绍了一种被称为 ArrayList 的新的数组类型。ArrayList 具有数组的许多特征，但是在某些方面它比数组更强大，这是因为 ArrayLsit 可以在结构容量已满的情况下我调整自身的大小。ArrayList 还有几种对执行插入、删除以及查找操作很有用的方法。既然 C#语言不允许程序员像在 VB·NET 中那样动态地调整数组的大小，所以在无法提前知道要存储的数据项数量的情况下 ArrayList 就是一种非常有用的数据结构了。

练习

1· 请设计并实现一个类，这个类允许教师跟踪记录单独一门课的成绩。要包括的方法有计算平均分、计算最高分以及计算最低分。请编写程序来测试一下此类的实现。

2· 请修改练习 *1* 的内容使得此类可以记录多门课的成绩。请编写程序来测试实现。

3· 请用 ArrayList 重新编写练习 *1* 的内容。请编写程序来测试实现，还请用 Timing 类把此实现的性能与练习 *1* 用数组实现的性能进行比较。

4· 请设计并实现一个类，这个类要用数组来模拟 ArrayList 类的行为。此类还要包含尽可能多的来自 ArrayList 类的方法。请编写程序测试实现。

第 3 章基础排序算法

在计算机中实现存储数据最普遍的两种操作就是排序和查找。这是从计算机产业初始就已经确认的了。这意味着排序和查找也是计算机科学领域最值得研究的两种操作。本书提到的许多数据结构的主要设计目的就是为了使排序和/或查找更加简单，同时也是为了数据在结构内的存储更加有效。

本章会介绍有关数据排序和查找的基础算法。这些算法仅依赖数组作为数据结构，而且所采用的“高级”编程技术只是递归。本章还介绍了用来非正式分析不同算法之间速度与效率的方法，此方法贯穿全书。

3·1 排序算法

人们在日常生活中所接触到的绝大多数数据都是经过排序的。比如，按照字母顺序查询字典中的定义。或者按照名字的字母顺序在电话本中查询电话号码。再或者邮局会按照下列几个步骤对邮件进行排序分发：即首先按照邮政编码，然后再按照街道名称，最后还要按照姓名。排序在数据处理中是十分基础的过程，因而值得认真学习研究。

正如先前提到的那样，这里对不同排序算法的操作有非常少量的分析研究。尽管已经对一些非常古老的算法做了改进，但是仍然应该先学习几种简单的排序算法。这些简单算法就是插入排序算法、冒泡排序算法以及选择排序算法。这些算法的每一种都很容易理解和实现。对于任意情况而言这些算法不是最好的全面算法，但是对于少量数据集合或者其他特殊情况而言，它们是可用的最好算法。

3·1·1 数组类测试环境

为了检验这些算法，首先需要构造一个可以实现并测试算法的测试环境。这里将构造一个类来封装数组处理的一些常规操作，即元素插入操作，元素存取访问操作，以及显示数组内容的操组。下面就是程序的代码：

```
using System;

class CArray

{

    private int[] arr;

    private int upper;

    private int numElements;

    public CArray(int size)

    {

        arr = new int[size];

        upper = size - 1;

        numElements = 0;
```

```
}

public void Insert(int item)

{

    arr[numElements] = item;

    numElements++;

}

public void DisplayElements()

{

    for (int i = 0; i <= upper; i++)

        Console·Write(arr[i] + " ");

}
```

```
public void Clear()

{

    for (int i = 0; i <= upper; i++)

        arr[i] = 0;

    numElements = 0;

}

static void Main()

{

    CArray nums = new CArray(50);

    for (int i = 0; i <= 49; i++)

        nums·Insert(i);
```

```
            nums·DisplayElements();

            Console·ReadKey();

        }

}
```

程序的输出如下所示：

（原书P44截图）

在保留 CArray 类以便开始检测排序和查找算法之前，还是先来讨论一下如何在 CArray 类对象内实际存储数据的问题。为了更有效地说明不同排序算法是如何运行的，数组内数据需要随机放置。最好的实现方法就是使用随机数生成器来给数组的每个元素进行赋值。

在 C#中用 Random 类可以产生随机数。这种类型的对象可以产生随机数。为了实例化 Random 对象，需要给这个类的构造器传递一个种子。这里把这个种子看作是随机数生成器所能产生的随机数范围的上界。

下面另外看一个用 CArray 类来存储数的程序，而且采用了随机数生成器来选择存储到数组内的数据：

```
static void Main()

{

    CArray nums = new CArray(10);

    Random rnd = new Random(100);

    for (int i = 0; i < 10; i++)

    {

        nums.Insert(rnd.Next(0, 100));

    }

    nums.DisplayElements();

}
```

这段程序输出结果如下所示：

（原书P45截图1）

3·1·2 冒泡排序

首先要讨论的排序算法就是冒泡排序。冒泡排序是可用的最慢排序算法之一，但是它也是最容易理解和实现的排序算法之一，所以这里把它作为最先介绍的排序算法。

这种排序算法的得名是由于数值“像气泡一样”从序列的一端浮动到另一端。假设现在要把一列数按升序方式进行排序，即较大数值浮动到列的右侧，而较小数值则浮动到列的左侧。这种效果可以通过下列操作来实现：多次遍历整个列，并且比较相邻的数值，如果左侧的数值大于右侧数值就进行交换。

图3-1举例说明了冒泡排序算法的工作原理。图中的两个数字（2和72）用圆圈进行了突出表示，这两个数是上一个实例中要插入数组的其中两个数。从图上可以看出数字72是如何从数组的开头移动到数组中部的，而数字2又是如何从数组的后半部分移动到了数组的开头。

（原书P45 图）

图3-1 冒泡排序算法

BubbleSort 算法的代码如下所示：

```
public void BubbleSort()

{

    int temp;

    for (int outer = upper; outer >= 1; outer--)

    {

        for (int inner = 0; inner <= outer - 1; inner++)

        {

            if ((int)arr[inner] > arr[inner + 1])

            {

                temp = arr[inner];
```

```
                arr[inner] = arr[inner + 1];

                arr[inner + 1] = temp;

            }

        }

        this.DisplayElements();

    }

}
```

这段代码有几个地方需要注意。首先，交换数组元素的代码是写在主程序中的一行，而没有用子程序。如果多次调用交换子程序，就可能会降低排序的速度。既然交换代码只有短短三行的长度，所以不把代码放在子程序内也不会影响代码的清晰度。

更加需要注意的是程序中最外层的循环是从数组的末尾处开始，并且向数组的开始处移动。如果回顾图 *3-1* 就会知道，数组内最大值就在数组末尾的适当位置上。这意味着数组的索引比外层循环的值更大，而且它们已经在恰当的位置上了，因而算法不需要再访问这些数值了。

内层循环从数组的第一个元素开始，并且在几乎达到数组最后位置的时候结束。内层循环会对用 inner 和 inner*+1* 标识的两个相邻位置的数值进行比较，并且在必要时交换它们的数值。

3·1·3 检验排序过程

在开发算法的过程中可能要做的事情之一就是在程序运行期间观察代码的中间结果。在使用 Visual Studio·NET 的时候，可以用 IDE 自带的调试工具来实现。然而，有些时候全部真正要观测的却是数组的内容（或者是自行构建、排序或查找的数据结构的内容）。一种简便的实现方法是在代码的适当位置上插入显示输出的方法。

对于前面提到的 BubbleSort 方法而言，检测数组在排序过程中如何变化的最佳位置就是在内、外层循环之间。如果为两个循环的每次重复执行插入输出显示，就可以看到数值在排序过程中如何在数组中移动的记录。

例如，下面是添加了显示中间结果的 BubbleSort 方法：

```
public void BubbleSort()

{

    int temp;

    for (int outer = upper; outer >= 1; outer--)

    {

        for (int inner = 0; inner <= outer - 1; inner++)
```

```
        {

            if ((int)arr[inner] > arr[inner + 1])

            {

                temp = arr[inner];

                arr[inner] = arr[inner + 1];

                arr[inner + 1] = temp;

            }

        }

        this.DisplayElements();

    }

}
```

这里把 DisplayElements()方法放置在了两个 For 循环之间。如果对主程序按照如下形式进行修改：

```
static void Main()

{

    CArray nums = new CArray(10);

    Random rnd = new Random(100);

    for (int i = 0; i < 10; i++)

    {

        nums.Insert(rnd.Next(0, 100));

    }

    Console.WriteLine("Before sorting: ");

    nums.DisplayElements();
```

```
        Console·WriteLine("During sorting: ");

        nums·BubbleSort();

        Console·WriteLine("After sorting: ");

        nums·DisplayElements();

}
```

那么程序的输出如下所示：

（原书P48截图）

3·1·4 选择排序

下一个要讨论的排序算法是选择排序。这种排序是从数组的起始处开始，把第一个元素与数组中其他元素进行比较。然后，将最小的元素放置在第 0 个位置上，接着再从第 1 个位置开始再次进行排序操作。这种操作会一直到除最后一个元素外的每一个元素都作为新循环的起始点操作过后才终止。

在选择排序算法中使用了两层循环。外层循环从数组的第一个元素移动到数组最后一个元素之前的元素，而内层循环则从数组的第二个元素移动到数组的最后一个元素，并且查找比当前外层循环所指元素更小的数值。在内循环遍历一遍之后，就会把数组内最小值赋值到数组中合适的位置上。图 3-2 举例说明了此算法是如何处理前面用到的 CArray 类数据的。

实现 SelectionSort 算法的代码如下所示：

```
public void SelectionSort()

{

    int min, temp;

    for (int outer = 0; outer <= upper; outer++)

    {

        min = outer;

        for (int inner = outer + 1; inner <= upper; inner++)

        {
```

```
            if (arr[inner] < arr[min]) min = inner;

        }

        temp = arr[outer];

        arr[outer] = arr[min];

        arr[min] = temp;

        this·DisplayElements();

    }

}
```

（原书P49 图）

图 3-2 选择排序算法

为了说明此算法是如何工作的，在与外层循环相连的下一条语句之前放置了一个对showArray*()*方法的调用。其输出结果应该如下所示：

（原书P*49* 截图）

本章最后将要看到的基础排序算法是最容易理解的算法之一，即插入排序算法。

3·1·5 插入排序

插入排序算法类似于人们通常按照数字顺序或者字母顺序进行排序的方法。假如我要求全班同学上交填有本人姓名、学号以及简短自我介绍的索引卡片。而学生们交回来的卡片是随机排列的。如果要把卡片按照字母排序排列，就可以构建出一张座次表了。

所以，我把这些卡片带回了办公室，并且清理出了办公桌。紧接着我拿出了第一张卡片。卡片上的名字是 Smith。我把它放在办公桌最左侧的位置上，然后又拿出了第二张卡片。这张是 Brown。于是，我把 Smith 的卡片移动到右侧，并且把 Brown 的卡片放到 Smith 原来的位置上。下一张卡片是 Williams。不需要移动任何其他的卡片就可以把它放在最右侧的位置上。接下来的卡片是 Acklin。它需要放置在队列的开始处，所以其他所有的卡片都必须向右移动一个位置以便腾出空间放 Acklin。这就是插入排序算法的工作原理。

插入排序的代码如下所示，跟着的是对此算法工作原理的解释说明：

```
public void InsertionSort()
```

```
{
    int inner, temp;
    for (int outer = 1; outer <= upper; outer++)
    {
        temp = arr[outer];
        inner = outer;
        while (inner > 0 && arr[inner - 1] >= temp)
        {
            arr[inner] = arr[inner - 1];
            inner -= 1;
        }
```

```
            arr[inner] = temp;

            this·DisplayElements();

        }

}
```

插入排序算法有两层循环。外层循环会逐个遍历数组元素，而内层循环则会把外层循环所选择的元素与该元素在数组内的下一个元素进行比较。如果外层循环选择的元素小于内层循环选择的元素，那么数组元素都向右移以便为内层循环元素留出位置，这就像前面例子描述的那样。

现在就来看看选择排序是如何处理前面实例中用来排序的数据集合的。下面是程序的输出结果：

（原书 P51 截图）

这个输出清楚地表明插入排序不是通过交换来处理的，而是通过把较大的数组元素向右移动来为数组左侧较小元素留出空间的方式进行操作的。

3.2 基础排序算法的时间比较

上述三种排序算法在复杂度和理论上都是十分相似的，所以在互相进行比较的时候应该操作近似。这里用 Timing 类来比较三种算法，根据它们对庞大数据集合进行排序时所花费的时间判定出是否有算法会与众不同。

为了进行测试，这里用到基本代码和之前为了说明每种算法的工作原理而使用的代码完全一样。但是，在下面这些测试中，为了说明三种算法是如何处理较小数据集合和较大数据集合的，数组的大小是有变化的。时间测试程序要分别运行处理元素量为 100、1000、甚至 10000 的几种情况。下面是代码：

```
static void Main()

{

    Timing sortTime = new Timing();

    Random rnd = new Random(100);

    int numItems = 1000;

    CArray theArray = new CArray(numItems);

    for (int i = 0; i < numItems; i++)
```

```
            theArray.Insert(rnd.NextDouble() * 100);

        sortTime.startTime();

        theArray.SelectionSort();

        sortTime.stopTime();

        Console.WriteLine("Time for Selection sort: " +
sortTime.Result().TotalMilliseconds);

        theArray.Clear();

        for (int i = 0; i < numItems; i++)

            theArray.Insert(rnd.NextDouble() * 100);

        sortTime.startTime();

        theArray.BubbleSort();
```

```
    sortTime·stopTime();

    Console·WriteLine("Time for Bubble sort: " + sortTime·Result()·TotalMilliseconds);

    theArray·Clear();

    for (int i = 0; i < numItems; i++)

        theArray·Insert(rnd·NextDouble() * 100);

    sortTime·startTime();

    theArray·InsertionSort();

    sortTime·stopTime();

    Console·WriteLine("Time for Insertion sort: " +
sortTime·Result()·TotalMilliseconds);

}
```

这段程序输出结果是：

（原书 P52 截图）

这说明显示了选择排序和冒泡排序的执行效率相等，而插入排序的速度则是其他两种算法速度的一半（或者说是比其他算法慢两倍的时间）。

现在来比较一下当数组元素量为 1000 时三种排序算法的效率：

（原书 P53 截图 1）

这里可以看出数组的大小会使算法的性能产生很大的差异。选择排序比冒泡排序快了 100 多倍，而且选择排序比插入排序快了 200 多倍。

当数组元素量增加到 10000 个的时候，确实能看出数组大小对三种排序算法的影响。

（原书P53 截图2）

尽管选择排序始终比其他两种算法快出许多倍，但是所有这三种排序算法的性能还是相当低的。准确地说，这些算法没有一种在对庞大数据集合进行排序时是理想选择。但是存在能高效处理庞大数据集合的排序算法。第16章将会讨论这些算法的设计和使用。

小结

本章讨论了针对数据排序的三种算法，即选择排序、冒泡排序以及插入排序。所有这三种算法都是非常容易实现的，而且它们都可以很好地处理少量的数据集合。选择排序是三种算法中效率最高的，其次是冒泡排序和插入排序。正如本章末尾看到的那样，这三种算法没有一种是十分适合庞大数据集合的。（比如，多于千个元素的数据集合）

练习

1· 请创建一个至少由100个字符串值组成的数据文件。大家可以自行输入字符串来创建这个列表，也可以从某些类型的文本文件中复制内容，甚至可以通过随机生成字符串来创建文件。请使用本章讨论过的三种排序算法的每一种对文件进行排序。还请创建一个程序来测试每种算法，并且类似于本章最后一节的输出，也要输出三种算法的时间。

2· 请创建一个由1000个整数组成的数组，其中的整数是按数值大小排序的（即从小到大）。请编写一个程序能够运行三种排序算法来处理此数组，而且测试和比较每种算法的时间。最后还请把这些时间与随机排序的整数数组所用三种算法的时间进行比较。

3· 请创建一个由1000个整数字组成的数组，其中的整数是按数值大小反向顺序的（即从大到小）。请编写一个程序能够运行三种排序算法来处理此数组，而且测试和比较每种算法的时间。

第4章基础查找算法

数据查找是基础的计算机编程工作，而且人们对它的研究已经很多年了。本章只会看到查找问题的一个内容，即根据给定的数值在一个列表（数组）中进行查找。

有两种对列表内数据进行查找的方法：顺序查找和二叉查找。当数据项在列表内随机排列的时候可以使用顺序查找，而当数据项在列表内有序排列的时候则会用到二叉查找。

4·1 顺序查找算法

最突出的查找类型就是从记录集的开始处顺次遍历每条记录，直到找到所要的记录或者是到达数据集的末尾。这就是所谓的顺序查找。

顺序查找（也被称为线性查找）是非常容易实现的。从数组的起始处开始，把每个访问到的数组元素依次和所要查找的数值进行比较。如果找到匹配的数据项，就结束查找操作。如果遍历到数组的末尾仍没有产生匹配，那么就说明此数值不在数组内。

下面是一个执行顺序查找操作的函数：

```
bool SeqSearch(int[] arr, int sValue)

{
```

```
    for (int index = 0; index < arr.Length ; index++) //小 bug

        if (arr[index] == sValue)

            return true;

    return false;

}
```

如果发现匹配，那么函数会立刻返回 True 并且退出。如果到达数组的末尾，函数还没有返回 True，那么要查找的数值就不在数组内，而函数则会返回 False。

下面这个程序用来测试顺序查找的实现：

```
using System;

using System.IO;

public class  Chapter4

{
```

```
static void Main()

{

    int[] numbers = new int[100];

    StreamReader numFile = File.OpenText(@"c:\\numbers.txt");

    for (int i = 0; i < numbers.Length ; i++) //小bug

        numbers[i] = Convert.ToInt32(numFile.ReadLine(), 10);

    int searchNumber;

    Console.Write("Enter a number to search for: ");

    searchNumber = Convert.ToInt32(Console.ReadLine(),10);

    bool found;

    found = SeqSearch(numbers, searchNumber);
```

```
    if (found)

        Console·WriteLine(searchNumber + " is in the array·");

    else

        Console·WriteLine(searchNumber + " is not in the array·");

}

static bool SeqSearch(int[] arr, int sValue)

{

    for (int index = 0; index < arr·Length ; index++) //小 bug

        if (arr[index] == sValue)

            return true;

    return false;

}
```

```
}
```

程序首先会通过从文本文件中读取一组数据开始运行。数据是由前 100 个整数组成的，而且是按照部分随机的顺序进行存储的。随后，程序会提示用户输入所要查找的数，并且调用 SeqSearch 函数来进行查找。

当然，用户也可以编写顺序查找函数。这样当找到要查找的数值时，函数就会返回此数值在数组内的位置。而当没有找到要查找的数值时，函数就会返回-1。首先来看一看新函数：

```
static int SeqSearch(int[] arr, int sValue)

{

    for (int index = 0; index < arr.Length ; index++) //小 bug

        if (arr[index] == sValue)

            return index;

    return -1;

}
```

下面这个程序使用了上述函数：

```
using System;

using System.IO;

public class Chapter4

{

    static void Main()

    {

        int[] numbers = new int[100];

        StreamReader numFile = File.OpenText(@"c:\\numbers.txt");

        for (int i = 0; i < numbers.Length-1; i++)

            numbers[i] = Convert.ToInt32(numFile.ReadLine(), 10);

        int searchNumber;
```

```
        Console·Write("Enter a number to search for: ");

        searchNumber = Convert·ToInt32(Console·ReadLine(), 10);

        int foundAt;

        foundAt = SeqSearch(numbers, searchNumber);

        if (foundAt >= 0)

            Console·WriteLine(searchNumber + " is in the array at position " +
foundAt);

        else

            Console·WriteLine(searchNumber + " is not in the array·");

    }

    static int SeqSearch(int[] arr, int sValue)

    {
```

```
        for (int index = 0; index < arr·Length ; index++) //小 bug

            if (arr[index] == sValue)

                return index;

        return -1;

    }

}
```

4·1·1 查找最小值和最大值

人们经常要求计算机程序从数组（或者其他数据结构）里查找到最小值和最大值。在一个有序的数组中，查找最小值和最大值是很容易的工作。但是，在一个无序的数组中，这就是一个不小的挑战了。

下面就从了解如何找到数组的最小值开始吧。算法是：

1· 把数组的第一个元素作为最小值赋值给一个变量。

2· 开始循环遍历数组，并且把每一个后继数组元素与最小值变量进行比较。

3. 如果当前访问到的数组元素小于最小值，就把此元素赋值给最小值变量。

4. 继续上述操作直到访问到最后一个数组元素为止。

5. 最小值就是存储在变量内的数值了。

下面来看看实现此算法的函数 FindMin：

```
static int FindMin(int[] arr)

{

int min = arr[0];

for(int i = 0; i < arr.Length-1; i++)

if (arr[index] < min)

min = arr[index];

return min;

}
```

请注意数组查找是从第 1 个元素的位置开始的，而不是从第 0 个元素的位置开始。第 0 个元素的位置在循环开始前会获得最小值，因此开始进行比较操作是在第 1 个元素的位置上。

在数组内查找最大值的算法和查找最小值的方法相同。先把数组的首元素赋值给一个保存最大值的变量。接着循环遍历数组，把每个数组元素与存储在变量内的数值进行比较。如果访问到的数值大于当前，就进行替换。下面是代码：

```
static int FindMax(int[] arr)

{

int max = arr[0];

for(int i = 0; i < arr·Length-1; i++)

if (arr[index] > max)

max = arr[index];

return max;

}
```

上述两个函数的另外一种替换写法是返回最大值或最小值在数组内的位置，而不是返回实际的数值。

4.1.2 自组织数据加快顺序查找速度

当要查找的数据元素就在数据集合的开始处时就会产生最快速的成功查找。通过找到数据项后把它移动到数据集合开始处的方法可以确保成功定位数据项。

这种策略的含义就是通过把频繁查找的数据项放在数据集合开始处的方法来最小化查找的次数。最终的结果就是所有最频繁查找的数据项都会被放置在数据集合的开始部分。这是自组织的一个实例，这是因为数据集合不是在程序运行之前由程序员组织的，而是在程序运行期间由程序自身组织的。

既然要查找的数据大概会遵循“80－20”规则，这使得允许数据进行组织变得有意义了。其中，“80-20”原则意味着在数据集合上80%的查找操作都是为了查找到数据集合内20%的数据。自组织将最终把20%的数据放在数据集合的开始部分，这样顺序查找就可以快速地找到它们了。

像这样的概率分布被称为是帕累托分布，它是以19世纪后期通过研究收入和财富的扩散而发现这类概率分布的科学家 Vilfredo Pareto 的名字命名的。更多有关数据集合中概率分布的知识请参阅 Knuth 的书 (1998, pp. 399－401)。

这里可以很容易地修改 SeqSearch 方法来包含自组织。下面是此方法的第一部分修正：

```
static bool SeqSearch(int sValue)

{
```

```
for(int index = 0; index < arr·Length-1; index++)

if (arr[index] == sValue)

{

swap(index, index-1);

return true;

}

return false;

}
```

如果查找成功，那么会利用交换函数把找到的数据项与元素在数组的第一个位置上进行交换，显示如下所示：

```
static void swap(ref int item1, ref int item2)

{
```

```
int temp = arr[item1];

arr[item1] = arr[item2];

arr[item2] = temp;

}
```

正如上述已经修改过的一样，用 SeqSearch 方法的问题就是在多次查找过程中会相当多次地把频繁访问到的数据项移来移去。而这里希望把移动到数据集合开始处的数据项保留下来，并且当对集合后部一个后续数据项成功定位的时候也不用把已保留的数据项移动回去。

现在有两种方法可以实现这个目标。第一种方法是只交换那些找到的且位置远离数据集合开始处的数据项。这样只需要确定到底在数据集合内多么靠后的元素才需要交换。再次遵循“80－20”规则，这里可以定义一条原则，那就是只有当数据项的位置位于数据集合前20%数据项之外的时候才可以把它重新定位到数据集合的开始部分。下面是经过第一改写的代码：

```
static int SeqSearch(int sValue)

{

for(int index = 0; index < arr·Length-1; index++)
```

```
if (arr[index] == sValue && index > (arr·Length * 0·2))

{

swap(index, index-1);

return index;

}

else

if (arr[index] == sValue)

return index;

return -1;

}
```

这里的 If – Then 语句是短路的。这是因为若无法在数据集合内找到数据项，那么就没有理由检测到该项在数据集合内的索引位置了。

另外一种方法就是重写 SeqSearch 方法，从而使得此方法可以把找到的数据项与数据集合

内此项之前的元素进行交换。采用这种方法类似于数据排序时所用的冒泡排序方法，也就是说最终会把最频繁访问到的数据项按照它们的方式到达数据集合的前端。同时，这种方法也保证了不会把已经在数据集合前端的数据项移动回后边去。

下面显示的是新版本的 SeqSearch 代码：

```
static int SeqSearch(int sValue)

{

for(int index = 0; index < arr·Length-1; index++)

if (arr[index] == sValue)

{

swap(index, index-1);

retum index;

}

retum -1;
```

```
}
```

不论基于何种原因，上述这两种方法都会在需要保持数据集合无序状态的时候帮助进行查找。接下来的一节将要讨论一种只处理有序数据但比任何已提到的顺序查找算法都要更加高效的查找算法，即二叉查找。

4.2 二叉查找算法

当要查找的记录从头到尾都是有序排列的时候，为找到数值可以执行一种比顺序查找更加有效的查找。这种查找被称为是二叉查找。

为了理解二叉查找的工作原理，请假设你正试图猜测由朋友选定的一个在 1 至 100 之间的数字。对于每次你所做的猜测，朋友都会告诉你是猜对了，还是猜大了，或是猜小了。最好的策略是第一次猜 50。如果猜大了，那么就应该再猜 25。如果猜 50 猜小了，则应该再猜 75。在每次猜测的时候，你都应该根据调整的数的较小或较大范围（这依赖于你猜测的数是偏大还是偏小）选择一个新的中间点作为下次要猜测的数。只要遵循这个策略，你最终一定会猜出正确的数。图 4-1 说明了如果选择的数是 82 时这个策略的工作过程。

（原书 P63 图）

图 4-1 二叉查找算法分析

①游戏猜测的数字是 82

②第 1 次猜测：50 答案：太小了

③第 2 次猜测：75 答案：太小了

④第 3 次猜测：88 答案：太大了

⑤第 4 次猜测：81 答案：太小了

⑥第 5 次猜测：84 答案：太大了

⑦中间点是 82.5，这近似于 82

⑥第 6 次猜测：82 答案：正确

这里可以把这种策略作为一种算法来实现，即二叉查找算法。为了使用这种算法，首先需要把数据按顺序（最好是升序方式）存储到数组内（当然，其他数据结构也可行）。算法的第一步就是设置查找的上界和下界。在查找的开始，这就意味着是数组的上限和下限。然后，通过把上限和下限相加后除以 2 的操作就可以计算出数组的中间点。接着把存储在中间点上的数组元素与要查找的数值进行比较。如果两者相同，那么就表示找到了该数值，同时查找算法也就此结束。如果要查找的数值小于中间点的值，那么就通过从中间点减去一的操作计算出新的上限。否则，若是要查找的数值大于中间点的值，那么就把中间点加一求出新的下限。此算法反复执行直到下限和上限相等时终止，这也就意味着已经对数组全部查找完了。如果发生这种情况，那么就返回-1，这表示数组中不存在要查找的数值。

这里把算法作为C#语言函数进行了编写：

```
public int binSearch(int value)

{

    int upperBound, lowerBound, mid;

    upperBound = arr·Length - 1;

    lowerBound = 0;

    while (lowerBound <= upperBound)

    {

        mid = (upperBound + lowerBound) / 2;

        if (arr[mid] == value)

            return mid;
```

```
            else

                if (value < arr[mid])

                    upperBound = mid - 1;

                else

                    lowerBound = mid + 1;

        }

        return -1;

}
```

下面的程序采用二叉查找方法来查找一个数组：

```
static void Main(string[] args)

{

Random random = new Random();
```

```
CArray mynums = new CArray(910);

for(int i = 0; i <= 9; i++)

mynums·Insert(random·Nnext(100));

mynums·SortArr()BubbleSort();

mynums·DisplayElements();

int position = mynums·binSearch(77, 0, 0);

if (position >= -1)

{

Console·WriteLine("found item");

mynums·DisplayElements();

}
```

```
else

Console·WriteLine("Not in the array");

Console·Read();

}
```

4·3 递归二叉查找算法

尽管在上节中讲述的二叉查找算法是正确的，但它其实不是解决问题的正常方案。二叉查找算法实际是一种递归算法。这是因为此算法会不断地划分数组直到找到所要的数据项（或者是查找完全部数组）才会终止，而每次的划分都是表示成一个比原有问题规模更小的同类问题。这种分析问题的方式使得人们终于发现了执行二叉查找的递归算法。

为了使递归二叉查找算法可以执行，这里需要对代码进行一些改动。下面先来看一下代码，然后再讨论已经修改的内容：

```
public int RbinSearch(int value, int lower, int upper)

{

    if (lower > upper)

        return -1;
```

```
    else

    {

        int mid;

        mid = (int)(upper+lower) / 2;

        if (value < arr[mid])

            return RbinSearch(value, lower, mid - 1);

        else if (value == arr[mid])

            return mid;

        else

            return RbinSearch(value, mid + 1, upper);

    }

}
```

同迭代算法相比，递归二叉查找算法的主要问题是它的效率。当用这两种算法对含有 1000 个元素的数组进行排序的时候，递归算法始终比迭代算法慢了 10 倍：

（原书 P65 截图）

当然，选择递归算法常常是由于效率以外的其他原因，但是应该记住在任何实现递归算法的时候还应该寻找一种迭代的解决方案，这样便于比较两种算法的效率。

最后在结束二叉排序这个主题之前，还应该提到 Array 类所拥有的内置的二叉查找方法。此方法会取走两个参数，即数组名和要查找的数据项。然后，它会返回该数据项在数组内的位置，或者是由于没找到而返回-1。

为了说明此方法的工作原理，这里为所提及的类另外写了一个二叉查找方法。代码如下所示：

```
public int Bsearh(int value)

{

return Array·BinarySearch(arr, value);
```

}

当内置的二叉查找方法与用户定制的方法进行比较的时候，内置方法始终比用户定制方法执行速度快 10 倍。这没什么好惊讶的。如果按照完全相同的方式使用上述两种方法，那么应该始终优先选择内置的数据结构或算法而非用户定制的。

小结

查找数据集合内的数值是一种随处可见的计算机操作。最简单的查找数据集合的方法是从数据集合的头部开始查找数据项，直到查找到该数据项或者执行到数据集合的末尾才结束。这种查找方法最好用在数据集合相对较小且无序的时候。

如果数据集合是有序排列的，那么二叉查找算法会是一种较好的选择。二叉查找会持续划分数据集合直到找到所要查找的数据项为止。大家可以采用迭代方式或递归方式编写二叉查找算法。C#语言的 Array 类包括有内置的二叉查找方法。在调用二叉查找的任何时候都应该可以使用此方法。

练习

4. 顺序查找算法会始终找到数据集合内数据项的第一次出现。请创建一种新的顺序查找方法。新方法会有另一个整数参数用来说明要查找的数据项是第几次出现。

5. 请编写顺序查找算法用来找到数据项的最后一次出现。

6. 请在一组无序数据集合内运行二叉查找方法。这会发生什么事情呢？

7. 把 CArray 类与 SeqSearch 方法以及 BinSearch 方法一起使用，创建一个由 1000 个随机整数组成的数组。添加一个名为 compCount 的新的私有整型数据成员，并且

其初始值为 *0*。在每种查找算法内，当执行完关键性的比较之后增加一行代码，此代码是对 compCount 进行加一操作。运行这两种方法，而且每种方法都是查找同一个数 *734*。在运行完两种方法后，比较一下 compCount 的值。请问每种方法的 compCount 的值各是多少？而且，请问哪种方法执行了最少的比较操作？

第 *5* 章堆栈和队列

数据像表一样自然地组织起来。此前已经采用 Array 类和 ArrayList 类来把数据像表一样组织在一起。尽管其他的数据结构也可以把数据按照便利的形式组织起来进行处理，但是这些结构对于实际地设计并实现问题的解决方法都不提供真正的抽象。

堆栈和队列是两种面向表的数据结构，它们都提供了易于理解的抽象。堆栈中的数据只能在表的某一端进行添加和删除操作，反之队列中的数据则在表的一端进行添加操作而在表的另一端进行删除操作。堆栈被广泛用于从表达式计算到处理函数调用的任何编程语言的实现中。而队列则用在区分优先次序的操作系统处理以及模拟现实世界的事件方面，比如银行出纳柜台的队列，以及建筑物内电梯的操作。

C#语言为使用这些数据结构提供了两种类：Stack 类和 Queue 类。本章将会讨论如何使用这些类并且介绍一些实用的例子。

5·1 堆栈、堆栈的实现以及 STACK 类

正如前面提到的那样，堆栈是最频繁用到的数据结构之一。这里把堆栈定义为数据项的列表，而且这些数据项只能从表的末端进行存取访问。可存取访问的这端被称为是栈顶。堆栈的标准模型是自助餐厅的盘子堆。人们始终要从顶部拿走盘子，而且当洗碗工或者杂工把盘子放回盘子堆的时候也是把它放在盘堆的顶部。堆栈是著名的后进先出（LIFO）数据结构。

5·1·1 堆栈的操作

堆栈最基本的两种操作就是向堆栈内添加数据项以及从堆栈中删除数据项。Push（进栈）操作是向堆栈内添加数据项。而把数据项从堆栈内取走则用 Pop（出栈）操作。这些操作的实例说明可参见图 *5-1*。

（原书 P*69* 页图）

1 进栈、*2* 进栈、*3* 进栈、出栈、出栈、*4* 进栈

图 *5-1* 堆栈的进栈和出栈操作

堆栈的另外一种基本操作就是察看栈顶的数据项。Pop 操作会返回栈顶的数据项，但是此操作也会把此数据项从堆栈中移除。如果只是希望察看栈顶的数据项而不是真的要移除它，那么在 C#语言中有一种名为 Peek（取数）的操作可以实现。当然，此操作在其他语言和实现中可能采用其他的名称（比如 Top）。

进栈、出栈以及取数都是在使用堆栈时会执行的基本操作。但是，还有其他一些需要执行的操作以及需要检查的属性。从堆栈中移除全部数据项就是非常有用的操作。通过调用 Clear（清除）操作可以把堆栈全部清空。此外，在任何时候都能知道堆栈内数据项的数量也是非常有用的。这可以通过调用 Count（计数）属性来实现。许多实现都有 StackEmpty 方法。此方法会根据堆栈的状态返回一个真值或假值，但是也可以采用 Count 属性达到同样的目

的。

·NET 框架的 Stack 类实现了全部这些操作和属性，甚至还要更多。但是在讨论如何使用它们之前，还是先来看看如果没有 Stack 类，则需要如何实现一个堆栈。

*5·1·2*Stack 类的实现

Stack 的实现需要采用一种潜在的结构来保存数据。既然在新数据项进栈的时候不需要担心调整表的大小，所以这里选择用 ArrayList。

因为 C#语言拥有如此强大的面向对象的编程特征，所以这里将把堆栈作为一个类来实现。此类被称为是 CStack。这里还会包括一个构造器方法以及有关上述提及操作的方法。为了说明在 C#语言中实现的过程，Count 属性会作为一种属性来实现。首先从讨论类中需要的私有数据开始吧。

所需要的最重要的变量就是用来存储堆栈数据项的 ArrayList 对象。除此以外，另一个也需要关注的数据就是栈顶。这里将用一个简单的整型变量来处理以便提供类似索引的功能。当对一个新的 CStack 对象实例化时，会把此变量的初始值设为*-1*。每次把新的数据项压入堆栈时，变量就会自加 *1*。

构造器方法只完成对索引变量初始化为 *-1* 的操作。第一种实现的方法是 Push。程序调用 ArrayLsit 的 Add 方法，并且把传递给它的数值添加到 ArrayList 里面。Pop 方法完成三件事：调用 RemoveAt 方法来取走栈顶的数据项（脱离 ArrayList），索引变量自减 *1* 操作，以及最终返回出栈的对象。

Peek 方法是通过调用含有索引变量作为参数的 Item 方法来实现的。Clear 方法则简单地调用 ArrayList 类中同样的方法。既然不需要突发改变堆栈上数据项的数量，所以这里把 Count 属性写为只读的属性。

代码如下所示：

```
class CStack

{

    private int p_index;

    private ArrayList list;

    public CStack()

    {

        list = new ArrayList();

        p_index = -1;

    }

    public int count
```

```
{
    get
    {
        return list·Count;
    }
}
public void push(object item)
{
    list·Add(item);
    p_index++;
}
```

```
public object pop()

{

    object obj = list[p_index];

    list·RemoveAt(p_index);

    p_index--;

    return obj;

}

public void clear()

{

    list·Clear();

    p_index = -1;
```

```
    }

    public object peek()

    {

        return list[p_index];

    }

}
```

下面就用这段代码来编写一个用堆栈解决问题的程序。

所谓回文是指向前和向后拼写都完全一样的字符串。例如，“dad”、“madam”以及“sees”都是回文，而“hello”就不是回文。检查字符串是否为回文的方法之一就是使用堆栈。常规算法是逐个字符的读取字符串，并且在读取时把每个字符都压入堆栈。这会产生反向存储字符串的效果。下一步就是把堆栈内的每一个字符依次出栈，并且把它与原始字符串从开始处的对应字母进行比较。如果在任何时候发现两个字符不相同，那么此字符串就不是回文，同时就此终止程序。如果比较始终都相同，那么此字符串就是回文。

既然已经定义了 CStack 类，所以下面这个程序就从 Sub Main 开始：

```
static void Main(string[] args)

{
```

```
CStack alist = new CStack();

string ch;

string word = "sees";

bool isPalindrome = true;

for (int x = 0; x < word.Length; x++)

    alist.push(word.Substring(x, 1));

int pos = 0;

while (alist.count > 0)

{

    ch = alist.pop().ToString();

    if (ch != word.Substring(pos, 1))
```

```
        {

        isPalindrome = false;

        break;

        }

        pos++;

    }

    if (isPalindrome)

        Console.WriteLine(word + " is a palindrome.");

    else

        Console.WriteLine(word + " is not a palindrome.");

    Console.Read();
```

}

5·2STACK 类

Stack 类是 ICollection 接口的一个实现。它代表了一个 LIFO 群集或一个堆栈。此类在·NET 框架中是作为循环缓冲来实现的。这使得能动态地分配进栈数据项的空间。

Stack 类包含进栈方法、出栈方法以及取值方法。此外，还有用于确定堆栈内元素数量的方法，清除堆栈全部数值的方法，以及把堆栈数值作为数组返回的方法。这里首先从讨论 Stack 类构造器的工作原理开始。

5·2·1Stack 构造器方法

这里有三种方法来实例化一个堆栈的对象。默认的构造器实例化成一个具有 10 个数值初始容量的空堆栈。调用默认构造器的方式如下所示：

```
Stack myStack = new Stack();
```

对常规堆栈进行实例化如下所示：

```
Stack<string> myStack = new Stack<string>();
```

每次堆栈达到满容量，就会把容量值翻倍。

第二个 Stack 构造器方法允许创建一个来自另一个群集对象的堆栈对象。例如，

可以把构造器作为数组进行传递，并且用来自现有数组的元素构建成堆栈：

```
string[] names = new string[] { "Raymond", "David", "Mike" };

Stack nameStack = new Stack(names);
```

执行 Pop 方法会首先把“Mike”从堆栈中移除。

当然，还可以实例化堆栈对象并且指明堆栈的初始容量。如果提前知道要存储在堆栈内的元素的数量，那么这个构造器就会派上用场。在用这种方法构造堆栈的时候，就会使程序更加有效。如果堆栈有 20 个元素且已达到总容量，那么添加一个新元素将会包含 20+1 条指令，因为要给新元素腾出空间，就需要移动堆栈内的每一个元素。

实例化带有初始容量的 Stack 对象的程序代码如下所示：

```
Stack myStack = new Stack(25);
```

5·2·2 主要的堆栈操作

对堆栈最主要的操作就是 Push 和 Pop。用 Push 方法把数据添加到堆栈里面。用 Pop 方法把数据从堆栈中移除。下面通过用堆栈来计算简单的算术表达式的实例来了解一下这些方法。

这个表达式求值器采用了两个堆栈：一个用于运算数（数字），而另一个则用于运算符。算术表达式会作为字符串存储起来。利用 For 循环来读取表达式中的每个字符，并且把字符串解析成独立的记号。如果记号是数字，就把它压入数字堆栈内。如果记号是运算符，则把它压入运算符堆栈内。既然这里采用的是中缀算术运算，所以在执行一次操作之前要等到堆栈内压入两个运算数才行。一旦满足条件了，就把两个运算数和一个运算符出栈，并且执行指定的算术操作。接着，再把运算结果压回到堆栈内，而这个运算结果就变成下一次操作的第

一个运算数了。继续反复这样的操作直到所有数字都执行完入栈和出栈操作为止。

下面就是程序的代码：

```
using System;

using System.Collections;

using System.Text.RegularExpressions;

namespace csstack

{

    class Class1

    {

        static void Main(string[] args)

        {

            Stack nums = new Stack();
```

```
        Stack ops = new Stack();

        string expression = "5 + 10 + 15 + 20";

        Calculate(nums, ops, expression);

        Console.WriteLine(nums.Pop());

        Console.Read();

    }

    // IsNumeric isn't built into C# so we must define it

    static bool IsNumeric(string input)

    {

        bool flag = true;

        string pattern = (@"^\d+$");
```

```
        Regex validate = new Regex(pattern);

        if (!validate.IsMatch(input))

        {

            flag = false;

        }

        return flag;

    }

    static void Calculate(Stack N, Stack O, string exp)

    {

        string ch, token = "";

        for (int p = 0; p < exp.Length; p++)
```

```
{

    ch = exp.Substring(p, 1);

    if (IsNumeric(ch))

        token += ch; //+=

    if (ch == " " || p == (exp.Length - 1))

    {

        if (IsNumeric(token))

        {

            N.Push(token);

            token = "";

        }
```

```
            }

            else if (ch == "+" || ch == "-" || ch == "*" || ch == "/")

                O.Push(ch);

            if (N.Count == 2)

                Compute(N, O);

        }

}

static void Compute(Stack N, Stack O)

{

    int oper1, oper2;

    string oper;
```

```
oper1 = Convert·ToInt32(N·Pop());

oper2 = Convert·ToInt32(N·Pop());

oper = Convert·ToString(O·Pop());

switch (oper)

{

    case "+":

        N·Push(oper1 + oper2);

        break;

    case "-":

        N·Push(oper1 - oper2);

        break;
```

```
                case '*':

                    N.Push(oper1 * oper2);

                    break;

                case '/':

                    N.Push(oper1 / oper2);

                    break;

            }

        }

    }

}
```

实际上用 Stack 来执行后缀算术表达式的计算会更容易。大家在练习里会有机会实现后缀求值器。

5·2·3Peek 方法

Peek 方法会让人们在不把数据项移出堆栈的情况下看到栈顶数据项的值。如果没有这种方法，那么就需要把数据项从堆栈内移除才会知道它的数值。当大家想在栈顶数据项出栈之前查看它的数值的时候，就可以采用这种方法：

```
if (IsNumeric(Nums·Peek()))

num = Nums·Pop();:
```

5·2·4Clear 方法

Clear 方法会把所有数据项从堆栈内移除，并且把数据项计数器设置为零。很难说清楚 Clear 方法是否会影响堆栈的容量。因为无法检查堆栈的实际容量，所以最好的办法就是假设堆栈的容量被重新设置为初始默认的 10 个元素的大小。

Clear 方法的有效应用是在处理过程出现错误的情况下清除堆栈。例如，在上述表达式求值器中，如果遇到除以 0 的操作，这就是错误，需要清除堆栈：

```
if (oper2 == 0)

Nums·Clear();
```

5·2·5 Contains 方法

Contains 方法用来确定指定的元素是否在堆栈内。如果找到该元素，那么此方法会返回 True；否则就返回 False。这种方法可以用来寻找堆栈内并非当前栈顶的数值。比如，堆栈内某个字符可能会导致处理错误的这种情况：

```
if (myStack·Contains(" "))

StopProcessing();

else

ContinueProcessing();
```

5·2·6 CopyTo 方法和 ToArray 方法

CopyTo 方法会把堆栈内的内容复制到一个数组中。数组必须是 Object 类型，因为这是所有堆栈对象的数据类型。此方法会取走两个参数：一个数组和开始放置堆栈元素的数组的起始索引。堆栈内元素按照 LIFO 的顺序进行复制操作，就好像对它们进行出栈操作一样。下面这段代码说明了 CopyTo 方法的调用：

```
Stack myStack = new Stack();

for (int i = 20; i > 0; i--)

myStack·Push(i);
```

```
object[] myArray = new object[myStack·Count];

myStack·CopyTo(myArray, 0);
```

ToArray 方法的工作原理与 CopyTo 方法类似。但是用户无法指定数组的起始索引位置，而是需要在赋值语句中创建新的数组。实例如下所示：

```
Stack myStack = new Stack();

for (int i = 0; i > 0; i++)

myStack·Push(i);

object[] myArray = new object[myStack·Count];

myArray = myStack·ToArray();
```

5·2·7Stack 类的实例：十进制向多种进制的转换

虽然在大多数商务应用中都采用十进制数，但是一些科学技术应用则要求把数表示成其他进制的形式。许多计算机系统应用程序要求数既可以表示成八进制的形式，也可以表示成二进制的形式。

用来把十进制数转化为八进制或二进制数的一种算法就是利用堆栈来实现的。下面列出了算

法的步骤：(原书这一段直译意思不清，所以译者进行的补充说明。)

取十进制数

取要转换进制的基数

循环

把十进制数与基数相除的余数压入堆栈

把十进制数与基数相除的商赋值给新的十进制数

当十进制数不等于 0 时继续循环

一旦循环终止，就产生了被转换的数。把堆栈内独立的数字简单地出栈就可以看到结果了。
下面就是程序的一个实现：

```
using System;

using System.Collections;

namespace csstack

{

    class Class1
```

```
{

    static void Main(string[] args)

    {

        int num, baseNum;

        Console·Write("Enter a decimal number: ");

        num = Convert·ToInt32(Console·ReadLine());

        Console·Write("Enter a base: ");

        baseNum = Convert·ToInt32(Console·ReadLine());

        Console·Write(num + " converts to ");

        MulBase(num, baseNum);

        Console·WriteLine(" Base " + baseNum);
```

```
        Console·Read();

}

static void MulBase(int n, int b)

{

        Stack Digits = new Stack();

        do

        {

                Digits·Push(n % b);

                n /= b;

        } while (n != 0);

        while (Digits·Count > 0)
```

```
                Console·Write(Digits·Pop());

        }

    }

}
```

这个程序举例说明了为什么堆栈对许多计算问题而言是一种有用的数据结构。当把十进制数转化成其他形式的时候，会从最右侧的数字开始操作，并且按照这种工作方式一直到左侧。在操作顺利执行的同时把每一个数字压入堆栈，这是因为在操作结束的时候，被转换的数字可以按照正确的顺序排列。

尽管堆栈是一种有用的数据结构，但是一些应用程序为了使自身更适合被模拟而采用了另外基于表的数据结构。例如，在杂货店或本地影碟租借店内形成的队伍。不同于后进先出的堆栈，在这些队伍内的第一个人应该最先出去（FIFO）。另外一个实例就是发送给网络（或本地）打印机的打印任务列表。打印机应该首先处理最先发送的任务。这里采用了一种基于表的数据结构来对这些实例进行模拟。这种结构被称为是队列。它是下一小节要讨论的主题。

5·3 队列、QUEUE 类以及 QUEUE 类的实现

队列是一种把数据从表的末端放入并在表的前端移除的数据结构。队列会按照数据项出现的顺序来存储它们。队列是先进先出（FIFO）数据结构的实例。队列用来对提交给操作系统或打印池的任务进行排序，而模拟应用程序则用队列对用户等待队列进行模拟。

5·3·1 队列的操作

队列包含两种主要的操作。一个是给队列添加新的数据项，另一个则是把数据项从队列中移除。添加新数据项的操作被称为是 Enqueue，而从队列中移除数据项的操作则被称为是 Dequeue。Enqueue 操作会在队列的末尾添加一个数据项，而 Dequeue 操作则会从队列的前端（或开始处）移除一个数据项。图 5-2 就举例说明了这些操作。

（原书 P80 页图）

A 到达队列

B 达到队列

C 达到队列

A 离开队列

B 离开队列

图 5-2 队列操作

队列的另外一个主要操作就是查看起始数据项。就像在 Stack 类中的对应操作一样，Peek 方法用来查看起始的数据项。这种方法仅仅返回数据项，而不会真的把数据项从队列中移除。

Queue 类的其他的属性也会对编程有所帮助。然而，在讨论这些属性之前，还是先来看看如何能实现一个 Queue 类。

5·3·2Queue 的实现

就像 Stack 类的实现所做的一样，Queue 类的实现用 ArrayList 简直是毋庸置疑的。对于这些数据结构类型而言，由于它们都是动态内置的结构，所以 ArrayList 是极好的实现选择。当需要往队列中插入数据项时，ArrayList 的 Add 方法会把数据项放置在表的下一个自由元素上。当需要从队列中移除数据项时，ArrayList 会在表中把每一个保留的数据项向前移动一个元素。这里不需要保留占位符，因为这样会导致代码内不易察觉的错误。

下面这个 Queue 类的实现包含 EnQueue 方法、DeQueue 方法、ClearQueue 方法（清除队列）、Peek 方法以及 Count 方法，而且还有一个用于此类的默认构造器：

```
public class CQueue

{

    private ArrayList pqueue;

    public CQueue()

    {

        pqueue = new ArrayList();
```

```
}

public void EnQueue(object item)

{

    pqueue.Add(item);

}

public void DeQueue()

{

    pqueue.RemoveAt(0);

}

public object Peek()

{
```

```
        return pqueue[0];

    }

    public void ClearQueue()

    {

        pqueue·Clear();

}

    public int Count()

    {

        return pqueue·Count;

}

}
```

5·3·3 Queue 类：实例应用

前面已经提到了用于 Queue 类的主要方法，而且还了解了这些方法在 Queue 类中的实现。下面通过查看一个用 Queue 作为基本数据结构的实际编程问题来进一步讨论这些方法。但是，首先需要说明一下 Queue 对象的几个基本属性。

在对一个新的 Queue 对象实例化的时候，队列默认的容量是 *32* 个数据项。根据定义，当队列已满时，其容量会双倍增长。这就意味着当队列最初达到满容量时，其新的容量值会变为 *64*。但是大家可以不用受这些数值的限制。在实例化对队列时，大家可以指定一个不同的初始容量值。如下所示：

```
Queue myQueue = new Queue(100);
```

这个设置把队列的容量值设为了 *100* 个数据项。当然，大家也可以改变增长的倍数。增长倍数是传递给构造器的第二个参数，如下所示：

```
Queue myQueue = new Queue(32, 3);
```

通用的 Queue 初始化如下所示：

```
Queue<int> numbers = new Queue<int>();
```

这一行指定增长倍数是默认初始容量的 *3* 倍。因为构造器会根据不同的信号来寻找方法，所以即使指定的容量值和默认容量完全一样，也需要进行详细说明。

正如早前已经提到的那样，队列经常用来模拟人们排队的情况。用队列模拟的假设之一就是在 Elks Lodge 举行的年度单身舞会。男士们和女士们进入会场并且站成一排。

舞池的场地狭小到只能同时容下三对舞者。当舞池内有空间的时候，就把队列中第一位男士和第一位女士选作舞者。一旦一对舞者离开队列，那么下一对舞者就会移动到队列的前端。

就像上面发生的操作一样，程序会宣布第一对舞者以及队伍中的下一对人选。如果没有完整的一对舞者，那么就会宣布队伍中的下一个人。如果队伍中没有人了，那么就把这种情况显示出来。

首先来看看用于模拟的数据：

F Jennifer Ingram

M Frank Optiz

M Terrill Beckerman

M Mike Dahly

F Beata Lovelace

M Raymond Williams

F Shirley Yaw _____此人名和随后的程序输出结果人名不服，应该为 Beth Munson。参见 P*86* 页输出截图的第三行第一个人名。

M Don Gundolf

F Bernica Tackett

M David Durr

M Mike McMillan

F Nikki Feldman

这里用一个结构来表示每一位舞者。两个简单的 String 类方法（Chars 方法和 Substring 方法）用来构建舞者。下面就是这段程序：

```
using System;

using System.Collections;

using System.IO;



namespace csqueue

{

    public struct Dancer

    {
```

```
    public string name;

    public string sex;

    public void GetName(string n)

    {

        name = n;

    }

    public override string ToString()

    {

        return name;

    }

}
```

```
class Class1

{

    static void newDancers(Queue male, Queue female)

    {

        Dancer m, w;

        m = new Dancer();

        w = new Dancer();

        if (male.Count > 0 && female.Count > 0)

        {

            m.GetName(male.Dequeue().ToString());

            w.GetName(female.Dequeue().ToString());
```

```
    }

    else if ((male.Count > 0) && (female.Count == 0))

        Console.WriteLine("Waiting on a female dancer.");

    else if ((female.Count > 0) && (male.Count == 0))

        Console.WriteLine("Waiting on a male dancer.");

}

static void headOfLine(Queue male, Queue female)

{

    Dancer w, m;

    m = new Dancer();

    w = new Dancer();
```

```
        if (male.Count > 0)

            m.GetName(male.Peek().ToString());

        if (female.Count > 0)

            w.GetName(female.Peek().ToString());

        if (m.name != " " && w.name != "")

            Console.WriteLine("Next in line are: " + m.name + "\t" + w.name);

        else

            if (m.name != "") //!=

                Console.WriteLine("Next in line is: " + m.name);

            else

                Console.WriteLine("Next in line is: " + w.name);

    }
```

```
static void startDancing(Queue male, Queue female)

{

    Dancer m, w;

    m = new Dancer();

    w = new Dancer();

    Console·WriteLine("Dance partners are: ");

    Console·WriteLine();

    for (int count = 0; count <= 3; count++)

    {

        m·GetName(male·Dequeue()·ToString());

        w·GetName(female·Dequeue()·ToString());
```

```
            Console·WriteLine(w·name + "\t" + m·name);

        }

}

static void formLines(Queue male, Queue female)

{

    Dancer d = new Dancer();

    StreamReader inFile;

    inFile = File·OpenText(@"c:\dancers·dat");

    string line;

    while (inFile·Peek() != -1)

    {
```

```
            line = inFile.ReadLine();

            d.sex = line.Substring(0, 1);

            d.name = line.Substring(2, line.Length - 2);

            if (d.sex == "M")

                male.Enqueue(d);

            else

                female.Enqueue(d);

        }

}

static void Main(string[] args)

{
```

```
Queue males = new Queue();

Queue females = new Queue();

formLines(males, females);

startDancing(males, females);

if (males.Count > 0 || females.Count > 0)

    headOfLine(males, females);

newDancers(males, females);

if (males.Count > 0 || females.Count > 0)

    headOfLine(males, females);

newDancers(males, females);

Console.Write("press enter");
```

```
            Console·Read();

        }

    }

}
```

用给定数据运行后的样例输出如下所示：

（原书 P86 页截图）

5·3·4 用队列存储数据

队列的另外一种应用就是存储数据。回顾计算的早期时代，那时的程序都是通过穿孔卡片录入到大型计算机内，其中每张卡片上存有单独一条程序语句。卡片用机械排序器进行存储，这种排序器采用了类似箱子的结构。这里可以用队列存储数据的方式来模拟此过程。这种排序方法被称为是基数排序。基数排序在编程的指令系统中不是最快的排序方法，但是它却能说明队列的另外一种有趣应用。

基数排序是通过对一组数据进行两遍排序来操作的。在这种情况下，整数的取值范围是从 *0* 到 *99*。第一遍是基于个位上的数字进行排序，而第二遍则是基于十位上的数字进行排序。然后，根据这些位置上的每个数字来把每一个数放置在一个箱子内。假设有下列这些数：*91*、*46*、*85*、*15*、*92*、*35*、*31*、*22*。

在箱子结构中第一遍排序的结果是：

0 号箱子：

1 号箱子：*91 31*

2 号箱子：*92 22*

3 号箱子：

4 号箱子：

5 号箱子：*85 15 35*

6 号箱子：*46*

7 号箱子：

8 号箱子：

9 号箱子：

现在，把这些数按照它们在箱子中的位置排列起来，结果如下所示：91、31、92、22、85、15、35、46。

接下来，取上述列表，按照十位上的数字对这些数排序后放入适当的箱子内：

0 号箱子：

1 号箱子：15

2 号箱子：22

3 号箱子：31 35

4 号箱子：46

5 号箱子：

6 号箱子：

7 号箱子：

8 号箱子：85

9 号箱子：91 92

把这些数从箱子内取出，并且把它们放回到列表内。此时就是这组整数排序后的结果：15、22、31、35、46、85、91、92。

用队列来表示这些箱子就可以实现这个算法。针对每一位数字一共需要九个队列。用取余运算和整除运算就可以确定个位上的数字以及十位上的数字。剩下的事情就是把数添加到适当的队列内，接着根据个位上的数字再把数从队列中取出进行重新排序，随后根据十位上的数字重复上述操作。最后的结果就是排序后的整数列表。

代码如下所示：

```
using System;

using System.Collections;

using System.IO;

namespace csqueue
```

```
{

    class Class1

    {

        enum DigitType { ones = 1, tens = 10 }

        static void DisplayArray(int[] n)

        {

            for (int x = 0; x <= n·GetUpperBound(0); x++)

                Console·Write(n[x] + " ");

        }

        static void RSort(Queue[] que, int[] n, DigitType digit)

        {
```

```
        int snum;

        for (int x = 0; x <= n.GetUpperBound(0); x++)

        {

            if (digit == DigitType.ones)

                snum = n[x] % 10;

            else

                snum = n[x] / 10;

            que[snum].Enqueue(n[x]);

        }

    }

    static void BuildArray(Queue[] que, int[] n)

    {
```

```
        int y = 0;

        for (int x = 0; x >= 9; x++)

            while (que[x].Count > 0)

            {

                n[y] =

                Int32.Parse(que[x].Dequeue().ToString());

                y++;

            }

    }

    static void Main(string[] args)

    {
```

```
Queue[] numQueue = new Queue[10];

int[] nums = new int[] { 91, 46, 85, 15, 92, 35, 31, 22 };

int[] random = new Int32[99];

// Display original list

for (int i = 0; i < 10; i++)

    numQueue[i] = new Queue();

RSort(numQueue, nums, DigitType·ones);

//numQueue, nums, 1

BuildArray(numQueue, nums);

Console·WriteLine();

Console·WriteLine("First pass results: ");
```

```
DisplayArray(nums);

// Second pass sort

RSort(numQueue, nums, DigitType.tens);

BuildArray(numQueue, nums);

Console.WriteLine();

Console.WriteLine("Second pass results: ");

// Display final results

DisplayArray(nums);

Console.WriteLine();

Console.Write("Press enter to quit");

Console.Read();
```

```
        }

    }

}
```

用 RSort 子程序来传递队列数组、整数的数组以及一个描述符。此描述符会告诉子程序是对个位上的数字还是对十位上的数字进行排序。如果排序是基于个位上的数字，那么程序计算的数字就是这个整数对 *10* 进行取模运算后的余数。如果排序是基于十位上的数字，那么程序计算的数字则是对这个整数除以 *10*（按照整除的方法）所取得的整数商。

为了重新构建整数的列表，当队列中有数据项时，通过连续执行 Dequeue 操作使得每个队列为空。这个操作在 BuildArray 子程序中执行。既然是从持有最小数的数组开始的，所以能把整数的列表构建成“有序的”。

5·3·5 源自 Queue 类的优先队列

正如大家现在知道的那样，队列是一种先进先出的数据结构。这种行为的效果就是会最先移除结构内最早进入的数据项。然而，对于很多应用程序而言，需要一种可以把具有最高优先级的数据项最先移除的数据结构，即使这个数据项在结构中不是“最早进入的”一个。对于这类应用程序 Queue 有一种特殊的案例，那就是优先队列。

许多应用程序在操作中都利用到了优先队列。一个很好的实例就是在计算机操作系统内的进程处理。某些进程可能有高于其他进程的优先级，比如打印进程就具有典型的低优先级。进程（或任务）通常会根据优先级进行编号，Priority（优先级）为 *0* 的进程比 Priority 为 *20* 的任务具有更高的优先性。

通常会把存储在优先队列中的数据项作为键值对来构造，其中关键字就是指优先级别，而值则用来识别数据项。例如，可以按照如下形式对一个操作系统进程进行定义：

```
struct Process

{

    int priority;

    string name;

}
```

大家不能把未修改的 Queue 对象用于优先队列。DeQueue 方法在被调用时只会把队列中的第一个数据项移除。但是，大家可以从 Queue 类派生出自己的优先队列类，同时覆盖 DeQueue 方法来实现自己的需求。

大家把这个类称为 PQueue。所有 Queue 的方法都可以照常使用，同时覆盖 Dequeue 方法来移除具有最高优先级的数据项。为了不从队列前端移除数据项，首先需要把队列的数据项写入一个数组。然后遍历整个数组从而找到具有最高优先级的数据项。最后，根据标记的数据项，就可以在不考虑此标记数据项的同时对队列进行重新构建。

下面就是有关 PQueue 类的代码：

```
public struct pqItem

{
```

```
    public int priority;

    public string name;

}

public class PQueue : Queue

{

    public PQueue()

    {

    }

    public override object Dequeue()

    {

        object [] items;
```

```
int min;

items = this.ToArray();

min = ((pqItem)items[0]).priority;

for(int x = 1; x <= items.GetUpperBound(0); x++)

    if (((pqItem)items[x]).priority < min)

    {

        min = ((pqItem)items[x]).priority;

    }

this.Clear();

int x2 ;

for(x2 = 0; x2 <= items.GetUpperBound(0); x2++)
```

```
            if (((pqItem)items[x2])·priority == min && ((pqItem)items[x2])·name !=
""")

                this·Enqueue(items[x2]);

        return base·Dequeue();

    }

}
```

接下来的代码说明了 PQueue 类的一个简单应用。急诊等待室对就诊的病人配置了优先级。心脏病突发的病人应该在割伤的病人之前进行治疗。下面这个程序模拟了三位几乎在同一时间进入急诊室的病人。分诊护士在检查完每一位病人后会分配得他们一个优先级，同时会把这些病人添加到队列内。进行治疗的第一个病人会通过 Dequeue 方法从队列中移除。

```
static void Main()

{

    PQueue erwait = new PQueue();

    pqItem[] erPatient = new pqItem[3];
```

```
pqItem nextPatient;

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 (int x = 0; x <= erPatient.GetUpperBound(0); x++)

    erwait.Enqueue(erPatient[x]);

nextPatient = (pqItem)erwait.Dequeue();

Console.WriteLine(nextPatient.name);
```

}

由于 Mary Brown 拥有高于其他两位病人的优先级，所以这段程序的输出是“Mary Brown”。

小结

学会适当且高效地使用数据结构是把编程专家从普通程序员中分离出来的技巧之一。编程专家会认识到把程序的数据按照适当的数据结构进行组织会使得数据的处理工作更加简单。事实上，用数据抽象来思考计算机编程问题会更容易最先得到解决问题的好的方案。

本章讨论了两种非常普通的数据结构，即堆栈和队列。堆栈可以用于解决计算机编程方面许多不同类型的问题，特别是诸如解释器和编译器这类系统编程领域的问题。此外，本章还说明了利用堆栈解决更通用问题的方法，比如确定单词是否为回文的问题。

队列也有许多的应用。操作系统用队列（通过优先队列）来对进程进行排序，而且队列还时常用于模拟现实世界的过程。最后，本章还用 Queue 类派生出一个用来实现优先队列的类。派生的新类的能力来自·NET 框架类库内的类。这也是 C#语言的·NET 版重要的优势之一。

练习

8· 请用 Stack 来检查程序的语句或公式是否括号匹配。编写一个视窗应用程序，让它为用户提供一个可录入带括号表达式的文本框。它还要提供一个 Check Parens 按钮。在点击此按钮时，会运行程序来检查表达式中括号的数量，并且高量显示出未匹配的括号。

9· 后缀表达式求值器所处理的算术运算语句的格式如下所示：运算数 1 运算数 2 运算符….。请使用两个堆栈，一个用于存储运算数，另一个则用于存储运算符。设计并实现一个 Calculator 类，此类可以把前缀表达式转换成为后缀表达式，然后用堆栈计算出表达

式的值。

10· 此练习包括设计一个求助台优先级管理器。它帮助把请求按照下列结构存储在文本文件中：优先级、请求队列的 id 号，以及请求时间。优先级是一个在 *1-5* 范围内的整数，并且 *1* 是最低级别，*5* 是最高级别。Id 号是一个四位数字的职员标识编号。时间是按照 TimeSpan·Hours、TimeSpan·Minutes 和 TimeSpan·Seconds 格式书写的。编写一个是视窗应用程序，在 Form Load 事件期间，从包含帮助请求的数据文件中读取 *5* 条记录，接着用优先队列来对其进行优先排列，并且把列表显示在列表框内。每当完成一项工作的时候，用户可以通过点击列表框中的此工作把其移除。当完成全部五项工作的时候，应用程序应该会自动地读取另外五条数据记录，对它们进行优先级排列，并且把它们显示在列表框内。

第 *6* 章 BitArray 类

BitArray 类是按照紧密格式来表示位集合。虽然我们能把位集合存储在常规数组内，但是如果采用专门为位集合设计的数据结构就能够创建更加有效的程序。本章将会介绍如何使用这种数据结构，并且将讨论一些利用位集合所解决的问题。此外，本章节还包含对二进制数、按位运算符以及位移运算符的内容回顾。

6·1 激发的问题

先来看一个最终会用 BitArray 类来解决的问题。这个问题是要找到素数。在公元前三世纪，古希腊哲学家埃拉托斯特尼发现了一种古来的方法来找素数，这种方法被称为是埃拉托斯特尼筛法。这种方法会一直筛选掉是其他数倍数的那些数，直到最后剩下的数都是素数为止。例如，假设要确定出前 *100* 个整数集合内的素数。这里会先从 *2* 开始，它是第一个素数。接着从头到尾遍历整数集合，把所有是 *2* 倍数的整数都移除掉。然后，移动到下一个素数 *3*。还是此从头到尾遍历整数集合，把所有是 *3* 倍数的整数都移除掉。再随后移动到素数 *5*，

继续如此往复操作。当操作全部结束时，所有留下的就都是素数了。

这里将先用常规数组来解决这个问题。所要采用的方法与用 BitArray 来解决问题的方法类似。这种方法要初始化含有 *100* 个元素的数组，并且把数组内每个元素的值都设为 *1*。操作会从索引 *2*（既然 *2* 是第一个素数）开始依次检查每个后续的数组索引。先要查看索引对应的元素值是 *1* 还是 *0*。如果数值为 *1*，那么就接着查看该索引是否是 *2* 的倍数。如果该索引是 *2* 的倍数，那么就把该索引上的数值设为 *0*。检查完所有数组索引后，会接着移动到索引 *3*，重复相同的操作，如此一直反复下去。

为了编写解决这个问题的代码，这里会采用先前已开发的 CArray 类。需要做的第一件事就是创建一个执行筛选的方法。代码如下所示：

```
public void GenPrimes()

{

    for (int outer = 2; outer <= arr·GetUpperBound(0); outer++)

        for (int inner = outer + 1; inner <= arr·GetUpperBound(0); inner++)

            if (arr[inner] == 1)

                if ((inner % outer) == 0)
```

```
                    arr[inner] = 0;

}
```

现在就需要一个显示素数的方法了：

```
public void ShowPrimes()

{

    for (int i = 2; i <= arr.GetUpperBound(0); i++)

        if (arr[i] == 1)

            Console.Write(i + " ");

}
```

接下来这个程序是用来测试所编写的代码的：

```
static void Main()

{
```

```
    int size = 100;

    CArray primes = new CArray(size - 1);

    for (int i = 0; i <= size - 1; i++)

        primes.Insert(1);

    primes.GenPrimes();

    primes.ShowPrimes();

}
```

这段代码说明了如何利用数组中的整数来进行埃拉托斯特尼筛法，但是既然数组中的每个元素不是 0 就是 1，所以我们建议使用位来解决问题。本章的后续部分将讨论如何利用 BitArray 类来实现埃拉托斯特尼筛法以及其他借用位集合自身来解决的问题。

6.2 位和位操作

既然大多数 VB.NET 程序员并不熟悉在位层上的工作，所以在介绍 BitArray 类之前我们还

是有必要先来讨论一下如何在 VB·NET 中使用位。本小节将研究如何在 VB·NET 中操作位，其中主要是介绍如何用按位运算符来处理 Byte 值。

6·2·1 二进制数制系统

在介绍如何处理 Byte 值之前，首先来回顾一些有关二进制的概念。二进制数是由 *0* 和 *1* 组成的字符串，它把基数为 *10*（或十进制）的数表示成基数为 *2* 的数。例如，用二进制表示的整数 *0* 是 *00000000*。而用二进制表示的整数 *1* 则是 *00000001*。下面用二进制表示了从 *0* 到 *9* 的整数：

00000000——*0*d（这里的 d 表示十进制的数）

00000001——*1*d

00000010——*2*d

00000011——*3*d

00000100——*4*d

00000101——*5*d

00000110——*6*d

*00000111——7*d

*00001000——8*d

*00001001——9*d

把二进制数转化为等价的十进制数的最好方法就是采用下列方案。对于每一个二进制数字，首先从最右侧的数字开始，每一个二进制的数字都表示成一个 *2* 的连续增大次幂的形式。

如果第一个位置上的数字为 *1*，那么就表示成 *20*。如果第二个位置上也为 *1*，则表示成 *21*。如此反复继续下去。

二进制数 *00101010* 等价于 *0 + 21 + 0 + 23 + 0 + 25 + 0 + 0 = 0 + 2 + 0 + 8 + 0 + 32 + 0 + 0 = 42*。

我们通常会用八位的集合形式来显示这些位，八位就是一个字节。在八位中可以表示的最大数是 *255*，它的二进制形式是 *11111111*，也就是 *1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255*。大于 *255* 的数必须存储在 *16* 位的集合内。例如，二进制表示的 *256* 就是 *00000001 00000000*。尽管不要求把低八位与高八位分开，但是这种写法是一种惯例。

6·2·2 处理二进制数：按位运算符和位移运算符

对二进制数而言不能使用标准的算术运算符进行操作。而是需要使用按位运算符（And、Or、Not）或位移运算符（<<、>>和>>>）来执行操作。本小节会说明这些运算符的工作原理。此外，后续的小节还会通过 VB·NET 应用程序来举例说明它们的用法。

首先来讨论按位运算符。这些都是大多数程序员早已熟悉的逻辑运算符——它们用来组合关系表达式从而计算出单独一个 Boolean（布尔）值。而对于二进制数而言，按位运算符用来对两个二进制数进行按位比较，从而产生一个新的二进制数。

按位运算符的工作原理和用 Boolean（布尔）值的做法一样。当处理二进制数时，True 位就等价为 *1*，而 False 位就等价为 *0*。为了说明按位运算符是如何进行按位操作的，就像对待 Boolean（布尔）值一样这里也会采用真值表。真值表内每行的前两列是两个运算数，而第三列则是运算的结果。下面是关于 And 运算符的真值表（用 Boolean 值）：

True
 True
 True

True
 False
 False

False
 True
 False

False
 False
 False

其等价的位值表如下所示：

1

1

1

1

0

0

0

1

0

0

0

0

关于 Or 运算符的 Boolean（布尔）型真值表如下所示：

True
True
True

True
False

True

False
True
True

False
False
False

其等价的位值表如下所示：

1

1

1

1

0

1

0

1

1

0

0

0

最后还有 Xor 运算符。因为在计算机程序执行的逻辑操作中不会用到这种运算符，所以很少有人知道这种按位运算符。当两个位用 Xor 运算符进行比较时，如果两个运算数中只有一个为 *1*，那么结果位就为 *1*。下面是真值表：

1

1

0

1

0

1

0

1

1

0

0

1

利用这些记下来的真值表，我们就可以将上述这些运算符与二进制数组合起来，从而产生新的二进制数。下面是一些实例：

（录入原书 P*99* 页）

现在来看看 VB·NET 视窗应用程序是如何更好的说明按位运算符的工作原理的。

6·3 按位运算符的应用

本节会举例说明按位运算符是如何在 C#语言中利用视窗应用程序工作的，其中视窗应用程序会把这些运算符用于一对数值。这里将会用早先开发的 ConvertBits 方法来协助按位运算符的操作。

首先来看一下此应用程序的用户界面，这对理解应用程序的工作原理大有帮助：

（原书 P*100* 页 截图 *1*）

具体操作是先录入两个整数值，并且由用户选择其中一种按位运算符的按钮。随后，每个整数值都会以位的形式显示出来，连同还会显示出相应按位操作的位串结果。下面是一个对 *1* 和 *2* 进行按位与操作的实例：

（原书 P*100* 页 截图 *2*）

接下来是对同样两个数值进行按位或操作的结果：

（原书 P*101* 页 截图）

操作的代码如下所示：

```
using System;
```

```
using System.Drawing;

using System.Collections;

using System.ComponentModel;

using System.Windows.Forms;

using System.Data;

using System.Text;

public class Form1 : System.Windows.Forms.Form

{

    private System.Windows.Forms.Button btnAdd;

    private System.Windows.Forms.Button btnClear;

    private System.Windows.Forms.Button btnOr;
```

```
private System·Windows·Forms·Button btnXor;

private System·Forms·Label lblInt1Bits;

private System·Forms·Label lblInt2Bits;

private System·Forms·TextBox txtInt1;

private System·Forms·TextBox txtInt2;

// other Windows app code here

private void btnAdd_Click(object sender, System·EventArgs e)

{

    int val1, val2;

    val1 = Int32·Parse(txtInt1·Text);

    val2 = Int32·Parse(txtInt2·Text);
```

```
        lblInt1Bits.Text = ConvertBits(val1).ToString();

        lblInt2Bits.Text = ConvertBits(val2).ToString();

    }

    private StringBuilder ConvertBits(int val)

    {

        int dispMask = 1 << 31;

        StringBuilder bitBuffer = new StringBuilder(35);

        for (int i = 1; i <= 32; i++)

        {

            if ((val && bitMask) == 0)

                bitBuffer.Append("0");
```

```
            else

                bitBuffer·Append("1");

            val <<= 1;

            if ((i % 8) == 0)

                bitBuffer·Append(" ");

        }

        return bitBuffer;

    }

    private void btnClear_Click(object sender, System·Eventargs e)

    {

        txtInt1·Text = "";
```

```
        txtInt2.Text = "";

        lblInt1Bits.Text = "";

        lblInt2Bits.Text = "";

        lblBitResult.Text = "";

        txtInt1.Focus();

    }

    private void btnOr_Click(object sender, System.EventsArgs e)

    {

        int val1, val2;

        val1 = Int32.Parse(txtInt1.Text);

        val2 = Int32.Parse(txtInt2.Text);
```

```
        lblInt1Bits.Text = ConvertBits(val1).ToString();

        lblInt2Bits.Text = ConvertBits(val2).ToString();

        lblBitResult.Text = ConvertBits(val1 //

        val2).ToString();

    }

    private void btnXOr_Click(object sender, System.EventsArgs e)

    {

        int val1, val2;

        val1 = Int32.Parse(txtInt1.Text);

        val2 = Int32.Parse(txtInt2.Text);

        lblInt1Bits.Text = ConvertBits(val1).ToString();
```

```
            lblInt2Bits.Text = ConvertBits(val2).ToString();

            lblBitResult.Text = ConvertBits(val1 ^ val2).ToString();

        }

}
```

6.3.1 位移运算符

二进制数只由 0 和 1 组成，而且数内的每一个位置都可以表示成数值 0 或一个 2 的次幂。在 C#语言中有两种运算符可以用来改变二进制数中位的位置。它们是：向左移位运算符（<<）和向右移位运算符（>>）。

这两种运算符都是对两个运算数进行处理：一个数值（写在左侧）和要移动的位数（写在右侧）。例如，如果写成如下形式：1 << 1。那么结果就是 00000010。而如果写成 2 >> 1 就可以返回得到原来的结果。下面再来看一个稍微复杂些的例子。数值 3 的二进制表示形式是 00000011。如果写成 3 << 1，那么结果就是 00000110。而如果写成 3 << 2，那么结果则变成了 00001100。

向右移位运算符的操作与向左移位运算符的操作恰好相反。例如，如果写有 3 >> 1，那么结果就是 00000001。

后续章节还会介绍如何编写视窗应用程序来举例说明位移运算符的用法。

6·4 整数转换成二进制形式的应用程序

本小节将举例说明如何使用少量的按位运算符来确定一个整数值的位模式。用户录入一个整数后点击 Display bits 按钮。整数值就会转化成相应的二进制形式显示在标签内，其中显示的位数是八位一组，一共四组。

用来把整数转化为二进制数的关键工具就是掩码。转换函数在显示数的位数时用掩码隐藏掉一些位。当掩码和整数值（即运算数）一起进行 AND（与）操作时，结果就是一条表示整数值的二进制字符串。

首先来看几个整数值及其所表示的二进制数值：

（原书 P104 页 截图 1）

（原书 P104 页 截图 2）

在计算机中负整数的二进制表示并不总是像例子显示的那样简单。如果想了解更多的内容，请参阅有关汇编语言和计算机组成方面的书籍。

（原书P105页 截图1）

正如看到的那样，上述这个数值65535是16位二进制所能表示的最大数值。如果数值增加到65536，就会得到下列结果：

（原书P105页 截图2）

最后再来看看当对存储在C#语言整数变量内的最大数进行转换的时候究竟会发生什么：

（原书P105页 截图3）

如果试图录入 2147483648，那么应用程序就会出错。大家可能会认为最左侧的二进制位是有效的，但是由于这一位是用来表示正负数的符号位，所以它是不能用的。

现在来研究一下驱动这个应用程序的代码。这里会首先列出代码的内容，然后再解释程序的工作原理：

```
using System;

using System.Drawing;

using System.Collections;

using System.ComponentModel;

using System.Windows.Forms;

using System.Data;

using System.Text;

public class Form1 : System.Windows.Forms.Form
```

```
{

    // Windows generated code omitted here

    private void btnOr_Click(object sender,

    System·EventsArgs e)

    {

        int val1, val2;

        val1 = Int32·Parse(txtInt1·Text);

        val2 = Int32·Parse(txtInt2·Text);

        lblInt1Bits·Text = ConvertBits(val1)·ToString();

        lblInt2Bits·Text = ConvertBits(val2)·ToString();

        lblBitResult·Text = ConvertBits(val1 // val2)·
```

```
        ToString();

}

private StringBuilder ConvertBits(int val)

{

        int dispMask = 1 << 31;

        StringBuilder bitBuffer = new StringBuilder(35);

        for (int i = 1; i <= 32; i++)

        {

                if ((val && bitMask) == 0)

                        bitBuffer·Append("0");

                else

                        bitBuffer·Append("1");
```

```
            val <<= 1;

            if ((i % 8) == 0)

                bitBuffer·Append(" ");

        }

        return bitBuffer;

    }

}
```

此应用程序执行的大多数工作就在 ConvertBits 函数内完成。变量 dispMask 保存二进制位掩码，而变量 bitBuffer 则用来保存由函数构造的二进制位字符串。为了允许使用类的 Append 方法而非串联的方式来构造字符串，所以这里把 bitBuffer 变量声明为 StringBuilder 类型。

二进制字符串是在 For 循环中构造的。由于要构造 32 位的字符串，所以循环要重复进行 32 次。为了构造二进制位字符串，需要把数值与二进制位掩码进行 AND（与）操作。如果操作的结果为 0，那么就会把 0 追加给字符串。如果结果为 1，则会把 1 追加给字符串。然后，为了稍后计算字符串中的下一位，会对数值进行向左移动一位的操作。最后，为了分割四个 8 位的子串，会在字符串中每隔八个二进制位就追加一个空格，这样会更便于读者阅

读。

6.5 位移的示例应用程序

本节会用一个视窗应用程序来举例说明位移运算符的工作原理。此应用程序会为两个运算数（一个是要位移的数值，而另一个则是要位移的位数）提供文本框，而且还会用两个标签来分别显示左侧运算数的初始二进制表示以及位移操作后结果的二进制形式。应用程序有两个按钮分别表示向左移操作和向右移操作。此外，还有 Clear 按钮和 Exit 按钮。

下面是这个程序的代码：

```
using System;

using System.Drawing;

using System.Collections;

using System.ComponentModel;

using System.Windows.Forms;

using System.Data;

using System.Text;
```

```
public class Form1 : System.Windows.Forms.Form

{

    // Windows generated code omitted

    private StringBuilder ConvertBits(int val)

    {

        int dispMask = 1 << 31;

        StringBuilder bitBuffer = new StringBuilder(35);

        for (int i = 1; i <= 32; i++)

        {

            if ((val && bitMask) == 0)

                bitBuffer.Append("0");
```

```
            else

                bitBuffer·Append("1");

            val <<= 1;

            if ((i % 8) == 0)

                bitBuffer·Append(" ");

        }

        return bitBuffer;

    }

    private void btnOr_Click(object sender, System·EventsArgs e)

    {

        txtInt1·Text = "";
```

```
        txtBitShift.Text = "";

        lblInt1Bits.Text = "";

        lblOrigBits.Text = "";

        txtInt1.Focus();

    }

    private void btnLeft_Click(object sender, System.EventsArgs e)

    {

        int value = Int32.Parse(txtInt1.Text);

        lblOrigBits.Text = ConvertBits(value).ToString();

        value <<= Int32.Parse(txtBitShift.Text);

        lblInt1Bits.Text = ConvertBits(value).ToString();
```

```
    }

    private void btnRight_Click(object sender, System.EventsArgs e)

    {

        int value = Int32.Parse(txtInt1.Text);

        lblOrigBits.Text = ConvertBits(value).ToString();

        value >>= Int32.Parse(txtBitShift.Text);

        lblInt1Bits.Text = ConvertBits(value).ToString();

    }

}
```

接下来是程序执行中的几个实例。先是执行 4 << 2：

（原书P*109*页 截图*1*）

接着执行*256 >> 8*：

（原书P*109*页 截图*2*）

*6·6*BITARRAY 类

BitArray 类用来处理位集合。位集合可以用来有效地表示 Boolean（布尔）值的集合。BitArray 和 ArrayList 十分类似，这是因为可以对 BitArray 进行动态地大小调整，而且在需要时添加二进制位而不用担心数组越界的问题。

6·6·1 使用 BitArray 类

通过实例化 BitArray 对象就可以创建 BitArray，并且同时会把希望在数组内的二进制位的数量传送给构造器：

```
BitArray BitSet = new BitArray(32);
```

这个 BitArray 的 32 个位都被设置为 False（假值）。如果想要它们成为 True（真值），那么可以像下列这样实例化数组：

```
BitArray BitSet = new BitArray(32, true);
```

构造器可以按照许多种不同的方式进行重载，但是这里将只会介绍一种构造器方法。可以用 Byte（字节）值的数组来实例化 BitArray。例如：

```
byte[] ByteSet = new byte[] { 1, 2, 3, 4, 5 };

BitArray BitSet = new BitArray(ByteSet);
```

BitSet BitArray 现在包含了字节值为 1、2、3、4 和 5 的二进制位。

二进制位存储在 BitArray 中，而且最高有效位（索引为 0）在最左侧的位置上。当按惯例从右向左读二进制数的时候，这样做可能会在使人犯糊涂。例如，下面这个八位的 BitArray 的内容就等价为数值 1：

```
True False False False False False False False
```

当然，较为常见的方式是把二进制数的最高有效位放置在右侧，就像下面这样：

```
0 0 0 0 0 0 0 1
```

这就需要自行编写代码来改变二进制数值（而不是 Boolean 值）的显示和二进制位的顺序。

如果 BitArray 里面有 Byte（字节）值，那么当循环遍历数组的时候每个 Byte（字节）值的每一位都将显示出来。下面这个简单的程序段就循环遍历了 Byte（字节）值的 BitArray：

```
byte[] ByteSet = new byte[] {1, 2, 3, 4, 5};

BitArray BitSet = new BitArray(ByteSet);

for (int bits = 0; bits <= BbitSet·Count-1; bits++)

Console·Write(BitSet·Get(bits) + " ");
```

下面是输出的结果显示：

（原书 P*111* 页 截图）

这些输出结果几乎是很难读懂的，而且这也不能真实的反映出数组内存储的情况。大家稍后将会看到如何使 BitArray 的这种类型更容易理解。但是现在首先需要明白如何从 BitArray 中检索到一个位值。

利用 Get 方法可以检索到存储在 BitArray 中的分离的位。此方法会取走一个整数参数，即希望检索到的值的索引，然后此方法返回的值将是表示成 True 或 False 的位值。为了显示来自 BitSet BitArray 的位值，这里会把 Get 方法用在先前的代码段内。

如果存储在 BitArray 中的数据确实是二进制数值（换句话说，数值应该被当作 *0* 和 *1* 显示出来），那么就需要一种方法来按照正确的顺序显示数值实际的 *1* 和 *0*，其中正确的顺序就是指从右边开始而不是从左边开始。虽然无法改变 BitArray 类所用的内部代码，但是我们可以编写外部代码来获得希望的输出。

下面这段程序创建了一个有 *5* 个 Byte（字节）值（即 *1*、*2*、*3*、*4* 和 *5*）的 BitArray，并且每个字节都按照正确的二进制形式显示出来：

```
using System;

using System.Collections;



class chapter6

{

    static void Main()

    {
```

```
int bits;

string[] binNumber = new string[8];

int binary;

byte[] ByteSet = new byte[] { 1, 2, 3, 4, 5 };

BitArray BitSet = new BitArray(ByteSet);

bits = 0;

binary = 7;

for (int i = 0; i <= BitSet.Count - 1; i++)

{

    if (BitSet.Get(i) == true)

        binNumber[binary] = "1";
```

```
else

    binNumber[binary] = "0";

bits++;

binary--;

if ((bits % 8) == 0)

{

    binary = 7;

    bits = 0;

    for (int ji = 0; ji <= 7; ji++)

        Console·Write(binNumber[ji]);

    Console·WriteLine();
```

```
            }

        }

    }

}
```

下面是程序的输出：

（原书 P112 页 截图）

此程序用到两个数组。第一个数组 BitSet 就是保存有 Byte（字节）值（按照位的格式）的 BitArray。而第二个数组 binNumber 只是一个用来保存二进制字符串的字符串数组。这个二进制字符串是由每个 Byte（字节）值的二进制位组成的，从最后一个位置（7）开始，一直向前移动到第一个位置（0）上。

每次遇到一个位值，程序会首先把它转化成为 1（如果为真值）或 0（如果为假值），然后把它放置在适当的位置上。这里有两个变量分别用来说明在 BitSet 数组（位）内的位置以及在 binNumber 数组（二进制）内的位置。当然，这里还需要知道什么时候已经转换了八个位，以及什么时候已经完成了对数的操作。把当前位值（在变量位内）对 8 进

行取模操作可以获得这些信息。如果没有余数，那么就说明正好在第八位上，而且可以把这个数写出来了。否则，就需要继续循环操作。

尽管已经把程序完整地写到了 Main( )内，但是在本章末尾的练习部分大家还有机会通过创建类，甚至是扩展含有此转换方法的 BitArray 类来整理此程序。

6·6·2 更多 BitArray 类的方法和属性

本小节会讨论 BitArray 类其他几个方法和属性。这些方法和属性很可能会在使用此类时用到。

Set 方法用来为数值设置一个特殊的位。此方法的用法如下所示：

```
BitArray·Set(bit, value)
```

这里的 bit 是针对集合的位的索引，而 value 则是希望赋值给此位的 Boolean（布尔）型数值。（尽管这里假设用 Boolean（布尔）型数值，但是实际上还可以采用其他数值，比如 *0* 和 *1*。大家在将会在下一节看到这样的用法。）

就像在 BitSet·SetAll（False）中一样，SetAll 方法允许为所有位设置数值。其方法就是把数值作为参数进行传递。

在一对采用 And（与）、Or（或）、Xor（异或）以及 Not（非）方法的 BitArray 上的所有位都可以执行按位操作。例如，假设给定了两个 BitArray，即 bitSet*1* 和 bitSet*2*，就可以按照下列方式执行按位 Or（或）操作：

bitSet1.Or(bitSet2)

下列表达式：

bitSet.Clone()

返回了 BitArray 的一个浅层复制，而表达式：

bitSet.CopyTo(arrBits)

则把 BitArray 的内容复制给一个命名为 arrBits 的标准数组。

通过这次回顾，现在终于准备好来看看如何用 BitArray 来编写埃拉托斯特尼筛法了。

6.7 用 BITARRAY 来编写埃拉托斯特尼筛法

本章的开始部分已经介绍了如何编写一个采用标准数组的程序来实现埃拉托斯特尼筛法。而本小节会实例说明相同的算法，只是这一次是采用 BitArray 来实现这种筛法的。

编写的这个应用程序会接收来自用户的一个整数值，然后确定是否是素数，并且还会把从 1 到 1024 的素数列表显示出来。下面就是这个应用程序的一些屏幕截图：

（原书P*114*页 截图）

这张截图显示的就是当录入的数不是素数时的情况：

（原书P*115*页 截图）

现在一起来看看代码：

```
using System;

using System.Drawing;

using System.Collections;

using System.ComponentModel;
```

```
using System.Windows.Forms;

using System.Data;

using System.Text;

public class Form1 : System.Windows.Forms.Form

{

    // Windows generated code omitted

    private void btnPrime_Click(object sender,

    System.EventsArgs e)

    {

        BitArray[] bitSet = new BitArray[1024];

        int value = Int32.Parse(txtValue.Text);
```

```
        BuildSieve(bitSet);

        if (bitSet.Get(value))

            lblPrime.Text = (value + " is a prime number.");

        else

            lblPrime.Text = (value + " is not a prime number.");

    }

    private void BuildSieve(BitArray bits)

    {

        string primes;

        for (int i = 0; i <= bits.Count - 1; i++)

            bits.Set(i, 1);

        int lastBit = Int32.Parse(Math.
```

```
Sqrt(bits·Count));

for (int i = 2; i <= lastBit - 1; i++)

    if (bits·Get(i))

        for (int j = 2 * i; j <= bits·Count - 1; j++)

            bits·Set(j, 0);

int counter = 0;

for (int i = 1; i <= bits·Count - 1; i++)

    if (bits·Get(i))

    {

        primes += i·ToString();

        counter++;
```

```
                if ((counter % 7) == 0)

                    primes += "\n";

                else

                    primes += "\n";

            }

        txtPrimes.Text = primes;

    }

}
```

在这个循环内应用了筛网：

```
int lastBit = Int32.Parse(Math.Sqrt(bits.Count).ToString());

for(int i = 2; i <= lastBit-1; i++)
```

```
if (bits.Get(i))

for (int j = 2 * i; j <= bits.Count-1; j++)

bits.Set(j, false0);
```

此循环会检查所有数的倍数一直到 BitArray 内数据项数的平方根为止，并且清除掉 2、3、4、5 等等的所有倍数。

一旦采用此筛网构建数组，那么就可以对 BitArray 执行一个简单调用：

```
bitSet.Get(value)
```

如果找到了数值，那么这个数就是素数。如果没有找到数值，那么筛网会删除掉这个数值，并且确定此数不是素数。

6.8 BITARRAY 与数组在埃拉托斯特尼筛法上的比较

在对含有 Boolean（布尔）值或位值的问题上采用 BitArray 类应该会更加有效。一些好像不包含这些数值类型的问题也可以通过重新设计的方式来使用 BitArray。

在对采用了 BitArray 和标准数组的埃拉托斯特尼筛法进行时间测试时，BitArray 方法始终保持快两倍的速度。大家将会有机会在练习中亲自检测一下这个结果。

小结

BitArray 用来存储位的集合。尽管位通常会用 *0* 和 *1* 来表示，但是 BitArray 类会把这些数值替换成 True（*1*）值或 False（*0*）值的形式存储起来。BitArray 在需要存储一组 Boolean（布尔）型数值时是很有用的，可是在需要处理位时它甚至会更加有用，这是因为人们可以很容易地在位值和 Boolean（布尔）型数值之间进行前后移动。

正如本章介绍的那样，能用数型数组解决的问题若采用位型数组来做会更加有效率。虽然一些读者可能会把这个仅想象（或者不是想象）为编程技巧，但是在某些情况下是不能否认存储位值（或 Boolean 型数值）的效率的。

练习

11· 请编写你自己的 BitArray 类（不要继承自 BitArray 类），此类包括的转换方法会取走 Boolean（布尔）型数值，并且把这些值转换成为位值。提示：用 BitArray 作为类的主要数据结构，但是要编写你自己的其他方法的实现。

12· 通过继承 BitArray 类的方式重新实现练习 *1* 中的类，并且只添加转换方法。

13· 用练习 *1* 和练习 *2* 设计的 BitArray 类中的一种来编写一个方法。此方法会取走一个整数值，并且对数值的位进行取反，然后再按照 *10* 进制的格式显示此数值。

14· Jon Bentley 在其编写的卓越编程著作《编程珠玑》（*2000* 年出版）里讨论了采用 BitArray 的编程问题的解决方案，尽管他在自己的书中把这称为是位矢量。阅读相关问题可参阅下列网址：http://www·cs·bell-labs·com/cm/cs/pearls/cto·html。然后请设计你自己的解决方案，并且至少在数据存储问题上采用 VB·NET。当然，你不需要用一个像书里

所用的那样大的文件，而只需要摘取一些足以测试自己的实现就可以了。

15· 请编写程序用来比较在埃拉托斯特尼筛法问题上用 BitArray 实现和用标准数组实现的时间差异。比较的结果到底会怎么样呢？

第 *7* 章 字符串、String 类和 StringBuilder 类

字符串对大多数计算机程序而言是很普遍的。像文字处理软件和网页应用程序这些程序类型都广泛采用了字符串。这使得处理这类应用程序的程序员在字符串处理的效率问题上需要花费额外的心思。本章会研究 C#语言处理字符串的方法，分析如何使用 String 类，最后还会介绍如何用 StringBuilder 类。当程序需要对 String 对象进行许多改变时会用到 StringBuilder 类。这是因为字符串和 String 对象都是不可改变的，而 StringBuilder 对象则是易变的。稍后会在章节内对所有这些进行解释和说明。

*7·1*STRING 类的应用

字符串是字符的序列。它可以包含字母、数字和其他符号。在 C#语言中把字符序列用一对闭合的双引号包围起来就可以产生文字串。下面是一些文字串的实例：

“David Ruff”

“the quick brown fox jumped over the lazy dog”

“123-45-6789”

“mmcmillan@pulaskitech·edu”

字符串可以由来自 Unicode 字符集的任何字符组成。字符串也可以是没有字符而组成的。这种特殊的字符串被称为是空字符串。它的形式是由一对彼此相连的双引号构成的（“”）。请千万记住这不是表示空格的字符串。表示空格的字符串形式是“ ”。

C#语言中的字符串具有精神分裂的天性——即字符串既是本地类型又是类的对象。实际上更准确的说法应该是可以把字符串作为本地数据值来使用，但是事实上每个产生的字符串都是 String 类的一个对象。稍后会说明原因。

7·1·1 创建 String 对象

创建 String 的方式如下所示：

string name = “Jennifer Ingram”;规则的变量，

当然，也可以在两条分离的语句中先声明变量然后再进行赋值。声明的语法使得名字看上去就像一个规则的变量，但是实际上它是 String 对象的一个实例。

C#语言的字符串还允许在字符串中放置转义字符。C 语言程序员和 C++语言程序员都很熟悉此技术，但是对于那些具有 VB 背景的人来说它却可能是一个新内容。转义字符用来把诸如换行符和制表符这类版式字符放置在字符串内。转义字符由一个反斜杠（\）开始，后边跟着单独一个表示版式的字母。例如，\n 说明换行，而\t 则表示一个制表符。在下面这行单独字符串中用到了这两种转义字符：

string name = “Mike McMillan \nInstructor, CIS \tRoom *306”;*

7·1·2 常用 String 类的方法们

虽然对字符串可以执行许多操作，但是一个小的操作集合却起着支配作用。三个最重要的操作分别是：1·找到字符串的子串。2·确定字符串的长度。以及 3·确定字符在字符串中的位置。

下面这段程序就说明了如何执行这些操作。这里把 String 对象实例化成字符串“Hello，world!”。然后把此字符串分离成两个组成段：第一个单词和第二个单词。代码如下所示，后边跟着的是对用到的 String 方法的解释说明。

```
using System;

class Chapter7

{

    static void Main()

    {

        string string1 = "Hello, world!";

        int len = string1·Length;

        int pos = string1·IndexOf(" ");
```

```
        string firstWord, secondWord;

        firstWord = string1.Substring(0, pos);

        secondWord = string1.Substring(pos + 1,(len - 1) - (pos + 1));

        Console.WriteLine("First word: " + firstWord);

        Console.WriteLine("Second word: " + secondWord);

        Console.Read();

    }

}
```

程序首先做的事就是用 Length 属性来确定对象 string1 的长度。长度简单说就是字符串中所有字符的总数量。这里会简要解释一下为什么需要知道字符串的长度。

为了把两词的短语分离出单词，就需要知道怎么分隔单词。在一个符合格式的短语中，空格可以用来分隔单词，所以就需要找到短语中两个单词之间的空格。这可以用 IndexOf 方法做到。此方法会取走一个字符，然后返回此字符在字符串中的位置。C#语言中的字符串都是基于零的，所以字符串中的第一个字符就在位置 0 上，而第二个字符则是在位置 1 上，其他则以此类推。如果无法在字符串中找到某个字符，就返回-1。

IndexOf 方法找到了分隔两个单词的空格的位置，然后就用下一个方法 Substring 来真地把第一个单词从字符串中抽出来。Substring 方法会取两个参数：一个开始位置以及要抽出字符的数量。请看下面这个实例：

```
string s = "Now is the time";
```

```
string sub = s.Substring(0, 3);
```

sub 的值就是“Now”。Substring 方法会把所要求的字符全部从字符串中抽出，但是如果试图超过字符串的末尾的话，就无法得到预期的内容。

程序把第一个单词从位置 0 开始由字符串中抽离出来，而且是抽出了 pos 个数量的字符。

既然 pos 包含了空格的位置，那么这样做好像很多余。但是，由于字符串是基于零的，所以这样做才会是正确的数量。

下一步就是把第二个单词抽离出来。既然知道了空格的位置，所以就知道了第二个单词是从 pos+1（再一次假设这里用的是符合格式的短语，短语中的每个单词都用一个空格进行分隔）开始的。较困难的部分就是要确定抽离出来的字符的数量，因为如果超出了字符串的末尾就无法得到预期的结果了。这里可以用一个类别公式来完成计算。首先，把一和找到的空格位置相加，然后用串长减去这个数值。这样就可以准确地告诉方法要抽离的字符的数量了。

虽然这段程序很有趣，但是它不是很实用。实际需要的程序应该是可以从任意长度的符合格式短语中抽离出单词。我们可以用几种不同的算法来实现。

这里将用到的算法包含下列这些步骤：

找到字符串中第一个空格的位置。

抽取单词。

从空格后边开始到字符串的末尾构建一个新的字符串。

寻找新字符串中的另外一个空格。

如果没有其他空格，那么抽取的单词就从当前位置到字符串的末尾。

否则循环返回第 2 步重复操作。

下面就是根据此算法编写的代码（从字符串中抽取的每个单词都存储到名为 word 的集合里面）：

```
using System;

using System.Collections;



class Chapter7

{

    static void Main()

    {
```

```
string astring = "Now is the time";

int pos;

string word;

ArrayList words = new ArrayList();

pos = astring·IndexOf(" ");

wWhile (pos > 0)

{

    word = astring·Substring(0, pos);

    words·Add(word);

    astring = astring·Substring(pos + 1, astring·Length -- (pos + 1));

    pos = astring·IndexOf(" ");
```

```
            if (pos == -1)

            {

                word = astring·Substring(0, asstring·Length);

                words·Add(word);

            }

            Console·Read();

        }

    }

}
```

当然，如果打算在程序中实际使用这个算法，那么最好把它做成一个函数并且返回一个集合，如下所示：

```
using System;
```

```
using System·Collections;

class Chapter7

{

    static void Main()

    {

        string astring = "now is the time for all good people ";

        ArrayList words = new ArrayList();

        words = SplitWords(astring);

        foreach (string word in words)

            Console·Write(word + " ");

        Console·Read();
```

```
}

static ArrayList SplitWords(string astring)

{

    string[] ws = new string[astring·Length - 1];

    ArrayList words = new ArrayList();

    int pos;

    string word;

    pos = astring·IndexOf(" ");

    while (pos > 0)

    {

        word = astring·Substring(0, pos);
```

```
            words·Add(word);

            astring = astring·Substring(pos + 1,astring·Length - (pos + 1));

            if (pos == -1)

            {

                word = astring·Substring(0, astring·Length);

                words·Add(word);

            }

            pos = astring·IndexOf(" ");

        }

        return words;

    }
```

}

但是返回头来讲，String 类已经有一个把字符串分离成部分的方法了（Split 方法），而且还有一个方法（Join 方法）可以取走一个数据集合并且把几部分组合成一个字符串。我们将在下一小节看到这些方法。

7·1·3 Split 方法和 Join 方法

把字符串分解成独立的数据段是一种非常常见的功能。从网络应用软件到日常办公应用软件范围内的许多程序都把数据存储在一些字符串格式类型里。为了简化字符串的分解以及再次合并在一起的过程，String 类提供了两种可用的方法：用于分解字符串的 Split 方法，以及用来把存储在数组中的数据制作成字符串的 Join 方法。

Split 方法取得一条字符串后，就会把它分解成数据成分块，然后把这些块放入 String 数组内。此方法的实现集中在用来确定分解字符串位置的分离字符身上。在前一小节的实例中，SplitWords 函数始终采用空格作为分隔符。而在使用 Split 方法时则可以指定要寻找的分隔符的内容。事实上，分隔符就是此方法的第一个参数。该参数必须以 char 型数组的形式出现，而数组的第一个元素将是用作分隔符的字符。

许多应用程序是通过写出用逗号分隔的数据串的方式来输出数据的。这被称为是逗号分隔值串，或简称为 CSVs。某些作者则采用逗号分隔这一术语。逗号分隔串就如同下列这样的形式：”Mike，McMillan，*3000*W· Scenic，North Little Rock，Ar，*72118*”。此串内的每一个逻辑数据块都是用逗号进行分隔的。这里可以用 Split 方法把每个逻辑数据块放入到一个数组中，如下列所示：

```
string data = "Mike,McMillan,3000 W· Scenic,North Little Rock,AR,72118";
```

```
string[] sdata;

char[] delimiter = new char[] {',};

sdata = data·Split(delimiter, data·Length);
```

现在就可以用标准的数组方法来存取这些数据了：

```
foreach (string word in sdata)

    Console·Write(word + " ");
```

这里还有一个参数需要传递给 Split 方法——即要存储到数组内的元素的数量。例如，如果想要把第一个串元素放置在数组的首位置而把串的其余部分放在第二个元素内，就需要采用下列方式调用此方法：

```
sdata = data·Split(delimiter, 2);
```

数组内的元素将是这样的：

第 0 个元素——Mike

第 1 个元素——McMillan, 3000W· Scenic, North Little Rock, Ar, 72118

现在来讨论另外一种方法，即用 Join 方法从数组变为字符串。此方法回取走两个参数：原

始数组以及用来分隔元素的字符。字符串是由跟着分隔符元素的数组元素组成构造出来的。还应该注意的是这种方法经常会被作为一种类方法来调用，这就意味着调用此方法来 自 String 类本身而不是来自 String 的实例。

下面这个实例采用了和 Split 方法所用相同的数据：

```
using System;

class Chapter7

{

    static void Main()

    {

        string data = "Mike,McMillan,3000 W· Scenic,North Little Rock,AR,72118";

        string[] sdata;

        char[] delimiter = new char[] { ',' };

        sdata = data·Split(delimiter, data·Length);
```

```
        foreach (string word in sdata)

            Console·Write(word + " ");

        string joined;

        joined = String·Join(" ", sdata);

        Console·Write(joined);

    }

}
```

String2 现在看来和 string1 完全一样了。

这两种方法在从其他源中获得数据放入自身程序方面（用 Split 方法）以及把数据从自身程序发送到其他源方面（用 Join 方法）是非常有用的。

7·1·4 比较字符串的方法

在C#语言中有集中比较 String 对象的方法。最显而易见的方法就是用在大多数情况下都可以工作正常的关系运算符。然而，在某些情况下其他一些比较的方法则会更加有效。例如，如果希望知道字符串是大于、小于、还是等于另外一个字符串，诸如此类的情况就需要用到 String 类中找到的方法了。

字符串之间的互相比较就如同数的比较一样。但是，由于“a”是大于还是小于“H”不是显而易见的，所以就需要有一些可用的数字测量方法。这种测量就是 Unicode 表。每一个字符（实际上是每一种符号）都有一个 Unicode 值，操作系统就是用此数值把字符的二进制表示转化成为字符的形式。通过使用 ASC 函数可以确定字符的 Unicode 值。ASC 实际上指的就是数的 ASCII 码。ASCII 码是一种先于 Unicode 的早期数字编码，而 ASC 函数是在 Unicode 包含 ASCII 之前被首先开发出来的。

为了找到字符的 ASCII 值，可以采用强制类型转换把字符简单地转换成为一个整数，如下所示：

```
int charCode;
```

```
charCode = (int)'a';
```

这样会把数值 *97* 存储到变量中。

然后，两个字符串的比较实际就是比较它们的数字编码。字符串“a”和字符串“b”不相等，就是应为编码 *97* 不同于编码 *98*。事实上 compareTo 方法可以用来确定两个 String 对象之间的精确关系。这里将会看到采用此方法的简单工作原理。

第一个要检测的比较方法就是 Equal 方法。此方法会由一个 String 对象调用，并且会把另外一个 String 对象取作它的参数。接着就会逐个字符的比较两个 String 对象。如果这两个 String 对象包含有相同的字符（以它们的数字编码为基础），那么方法就会返回一个 True 值。否则，方法就会返回 False 值。此方法的调用如下所示：

```
string s1 = "foobar";
```

```
string s2 = "foobar";

if (s1.Equals(s2))

    Console.WriteLine("They are the same.");

else

    Console.WriteLine("They are not the same.");
```

下一个比较字符串的方法就是 CompareTo。此方法也是取一个 String 作为参数，但是它不会布尔值。取而代之的是此方法会返回 *1*、*-1* 或者 *0*，具体数值要由传递给的字符串和调用此方法的字符串实例之间的关系来决定。下面是一些实例：

```
string s1 = "foobar";

string s2 = "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
```

如果两个字符串相等，那么 ComapreTo 方法会返回一个 0。而如果传递给的字符串“低于”调用方法的字符串，那么方法会返回-1。再如果传递给的字符串“高于”调用方法的字符串，那么方法返回的则是 1。

替换 CompareTo 方法的就是 Compare 方法。Compare 方法通常会被做作为一个类方法来调用。此方法会执行和 CompareTo 方法相同的比较类型，而且对于相同的比较会返回相同的值。Compare 方法的应用如下所示：

```
static void Main()

{

string s1 = "foobar";

string s2 = "foobar";

int compVal = String.Compare(s1, s2);

switch (compVal)
```

```
{
        case 0: Console·WriteLine(s1 + " " + s2 + " are equal");
break;
        case 1: Console·WriteLine(s1 + " is less than " + s2);
break;
        case 2: Console·WriteLine(s1 + " is greater than" + s2);
break;
        default: Console·WriteLine("Can't compare");
break;
}
}
```

另外两种在处理字符串时会很有用的比较方法是 StartsWith 和 EndsWith。这些实例方法都回取一个字符串作为参数，而且若实例是以字符串参数作为开始或结束那么方法都会返回 True 值。

下面两段小程序说明了这些方法的用法。首先，这里将先说明 EndsWith 方法：

```
using System;

using System·Collections;

class Chapter7

{

    static void Main()

    {

        string[] nouns = new string[] {"cat", "dog", "bird","eggs", "bones"};

        ArrayList pluralNouns = new ArrayList();

        foreach (string noun in nouns)
```

```
            if (noun.EndsWith("s"))

                pluralNouns.Add(noun);

        foreach (string noun in pluralNouns)

            Console.Write(noun + " ");

    }

}
```

这个程序先创建了一个名词数组，且其中一些名词还是复数的形式。接着，程序循环遍历数组的元素，并且查看名词是否为复数。如果是，就把这些名词添加到一个集合里。然后，程序遍历集合并且把每个复数名词显示出来。

接下来这个程序采用了相同的基本思想来确定以前缀”tri”开头的单词：

```
using System;

using System.Collections;

class Chapter7
```

```
{

    static void Main()

    {

        string[] words = new string[]{"triangle", "diagonal",
"trimester","bifocal","triglycerides"};

        ArrayList triWords = new ArrayList();

        foreach (string word in words)

            if (word.StartsWith("tri"))

                triWords.Add(word);

        foreach (string word in triWords)

            Console.Write(word + " ");
```

```
    }

}
```

7.1.5 处理字符串的方法

字符串处理通常包括对字符串的改变操作。我们需要在字符串中插入新的字符，从字符串中移除字符，用新字符替换旧字符，改变某些字符的情况，以及向字符串添加空格或者从字符串中移除空格，恰好为了命名某些操作。在 String 类中针对这些操作全部有相应的方法，因而本小节将对这些方法进行讨论。

这里将先以 Insert 方法开始。此方法会把某个字符串插入到另外一个字符串的指定位置。Insert 方法会返回新的字符串。调用此方法的格式如下所示：

```
String1 = String0.Insert(Position, String);
```

下面来看一个实例：

```
using System;

class chapter7

{

    static void Main()
```

```
    {

        string s1 = "Hello, · Welcome to my class·";

        string name = "Clayton";

        int pos = s1·IndexOf("","");

        s1 = s1·Insert(pos + 2, name);

        Console·WriteLine(s1);

    }

}
```

输出是：

Hello, Clayton· Welcome to my class·

此程序创建了字符串 s1，该字符串故意为名字留出了空间，这非常象计划为了匆匆处理邮件合并而不得不忍受一个字母一样。为了确保在逗号和名字之间存在空格，程序在找到逗号的位置上加上二。

Insert 方法之后下一个最合理的方法就是 Remove 方法了。这种方法会取走两个整数参数:一个开始位置和一个计数器，其中计数器就是要移除字符的数量。下面的代码在插入名字后再把名字从字符串中移除掉：

```
using System;

class chapter7

{

    static void Main()

    {

        string s1 = "Hello, · Welcome to my class·";

        string name = "Ella";

        int pos = s1·IndexOf("","");

        s1 = s1·Insert(pos + 2, name);

        Console·WriteLine(s1);
```

```
        s1 = s1·Remove(pos + 2, name·Length);

        Console·WriteLine(s1);

    }

}
```

为了移除名字，Remove 方法采用了与插入名字相同的位置，而且通过获取名字变量的长度来计算出计数器的值。就像这段代码和输出屏幕显示的那样，这里允许移除掉任何插入到字符串中的名字：

```
string name = "William Shakespeare";

int pos = s1·IndexOf(",");

s1 = s1·Insert(pos + 2, name);

Console·WriteLine(s1);

s1 = s1·Remove(pos + 2, name·Length);

Console·WriteLine(s1);
```

（原书P131页 截图）

接下来合理的方法就是 Replace 方法。这种方法会取走两个参数：要移除掉的字符串和用来替换掉的字符串。此方方会返回新的字符串。下面就是 Replace 方法的使用过程：

```
using System;

class chapter7

{

    static void Main()

    {

        string[] words = new string[] { "recieve", "decieve", "reciept" };

        for (int i = 0; i <= words·GetUpperBound(0); i++)
```

```
        {

            words[i] = words[i].Replace("cie", "cei");

            Console.WriteLine(words[i]);

        }

    }

}
```

这段代码唯一需要技巧的部分就是调用 Replace 方法的方式。既然是通过数组元素来存取每个 String 对象，所以就需要方法名跟在所使用的数组寻址的后边，这就引出了下列这段代码：

```
words[(index)].Replace("cie", "cei");
```

当然这样做是没有问题的，因为编译器知道 words(index)计算 String 对象的值。（当用 Visual Studio .Net 编写代码时，应该也提到了 Intellisense 允许这样做。）

当显示来自程序的数据时，为了数据排列美观，经常需要在打印区域内对数据进行排列。String 类包括两种执行此类排列操作的方法：PadLeft 方法和 PadRight 方法。PadLeft 方法会对字符串进行右对齐排列，而 PadRight 方法会对字符串进行左对齐排列。例如，若果需要在一个 10 个字符宽度区域内右对齐打印单词“Hello”，就需要写成下列形式：

```
string s1 = "Hello";

Console.WriteLine(s1.PadLeft(10));

Console.WriteLine("world");
```

输出就是：

```
      Hello

  World
```

下面是采用 PadRight 方法的实例：

```
string s1 = "Hello";

string s2 = "world";

string s3 = "Goodbye";

Console.Write(s1.PadLeft(10));

Console.WriteLine(s2.PadLeft(10));
```

```
Console·Write(s3·PadLeft(10));

Console·WriteLine(s2·PadLleft(10));
```

输出就是：

```
      Hello   world
    Goodbye   world
```

还有一个实例说明了如何排列来自数组的数据使其更容易阅读：

```
using System;

class chapter7

{

    static void Main()

    {

        string[,] names = new string[,]
```

```
{

    {"1504", "Mary", "Ella", "Steve", "Bob"},

    {"1133", "Elizabeth", "Alex", "David", "Joe"},

    {"2624", "Joel", "Chris", "Craig", "Bill"}

};

Console·WriteLine();

Console·WriteLine();

for (int outer = 0; outer <= names·GetUpperBound(0);outer++)

{

    for (int inner = 0; inner <= names·GetUpperBound(1); inner++)

        Console·Write(names[outer, inner] + " ");
```

```
            Console·WriteLine();

        }

        Console·WriteLine();

        Console·WriteLine();

        for (int outer = 0; outer <= names·GetUpperBound(0); outer++)

        {

            for (int inner = 0; inner <= names·GetUpperBound(1); inner++)

                Console·Write(names[outer, inner]·PadRight(10) + " ");

            Console·WriteLine();

        }

    }
```

```
}
```

这段程序的输出是：

（原书 P134 页 截图）

第一组显示的数据没有进行填补调整，而第二组数据是用 PadRight 方法显示的。

这里已经知道&运算符用于字符串的串联。String 类也包含了一种用于此目的的 Concat 方法。此方法会取走 String 对象的列表，把它们串联在一起，然后返回结果字符串。下面就是使用此方法的过程：

```
using System;

class chapter7

{

    static void Main()
```

```
    {

        string s1 = "hello";

        string s2 = "world";

        string s3 = "";

        s3 = String·Concat(s1, " ", s2);

        Console·WriteLine(s3);

    }

}
```

利用 ToLower 方法和 ToUpper 方法还可以把字符串从小写转换成大写形式（而且反之亦然）。下面这段程序代码实例说明了这些方法的工作原理：

```
string s1 = "hello";

s1 = s1·ToUpper();
```

```
Console·WriteLine(s1);

string s2 = "WORLD";

Console·WriteLine(s2·ToLower());
```

本小节以 Trim 方法和 TrimEnd 方法的讨论结束。在处理 String 对象时，这些对象有时会有额外的空格或者其他格式字符出现在字符串的开始或结尾处。Trim 方法和 TrimEnd 方法将会把空格或其他字符从字符串的任一端移除掉。人们既可以对指定的单个字符进行整理，也可以对字符数组进行整理。如果指定了字符数组，且在数组中没有找到任何字符，那么它们会受到来自字符串的整理。

首先来看一个实例，此实例对一组字符串值的开始和结尾处的空格进行整理：

```
using System;

class chapter7

{

    static void Main()

    {
```

```
        string[] names = new string[] {" David", " Raymond", "Mike ", "Bernica "};

        Console·WriteLine();

        showNames(names);

        Console·WriteLine();

        trimVals(names);

        Console·WriteLine();

        showNames(names);

    }

    static void showNames(string[] arr)

    {

        for (int i = 0; i <= arr·GetUpperBound(0); i++)
```

```
            Console·Write(arr[i]);

}

static void trimVals(string[] arr)

{

    char[] charArr = new char[] { ' ' };

    for (int i = 0; i <= arr·GetUpperBound(0); i++)

    {

        arr[i] = arr[i]·Trim(charArr[0]);

        arr[i] = arr[i]·TrimEnd(charArr[0]);

    }

}
```

```
}
```

下面是输出：

（原书 P136 页 截图）

下面另外一个实例把 HTML 代码页的注释剥去了 HTML 格式：

```
using System;

class chapter7

{

    static void Main()

    {

        string[] htmlComments = new string[]
```

```
{

    "<!-- Start Page Number Function -->",

    "<!-- Get user name and password -->",

    "<!-- End Title page -->",

    "<!-- End script -->"

};

char[] commentChars = new char[] {'<', '!', '-','>'};

for (int i = 0; i <= htmlComments.GetUpperBound(0); i++)

{

    htmlComments[i] = htmlComments[i].

    Trim(commentChars);
```

```
            htmlComments[i] = htmlComments[i].

            TrimEnd(commentChars);

        }

        for (int i = 0; i <= htmlComments.GetUpperBound(0); i++)

            Console.WriteLine("Comment: " + htmlComments[i]);

    }

}
```

下面是输出：

（原书 P137 页 截图）

7·2STRINGBUILDER 类

StringBuilder 类提供了对多变的 String 对象的存取。String 类的对象是不变的，这就意味着不能对他们进行改变。每次改变 String 对象的值时，就会产生一个新的对象来保存数值。另一方面，StringBuilder 对象却是多变的。当对 StringBuidler 对象进行改变时，正在改变的就是原始对象而不是对副本进行操作。本节会讨论如何针对程序中 String 对象发生改变的那些情况使用 StringBuilder 类。本章及本节的内容会以一个时间测试作为结束，此测试用来确定处理 Stringbuilder 类确实是比处理 String 类更加有效。

在 System·Text 名字域中可以找到 StringBuilder 类，所以在能使用 StringBuilder 对象之前需要在程序中输入此名字域。

7·2·1 构造 StringBuilder 对象

这里可以采用三种方法中的一种来构造 StringBuilder 对象。第一种方法使用默认构造器来创建对象：

```
StringBuilder stBuff1 = new StringBuilder();
```

这一行代码创建了对象 stBuff1，此对象的容量可以保存长度为 16 个字符的字符串。虽然此容量是默认设置的，但是可以通过在构造器调用中传递新容量的的方法进行改变，就像下面这样：

```
StringBuilder stBuff2 = nNew StringBuilder(25);
```

这一行代码构建了一个初始可以保存 25 个字符的对象。最终的构造器调用会取字符串作为参数：

StringBuilder stBuff3 = nNew StringBuilder(“Hello,world“);

这里的容量设置为 16，这是因为字符串参数没有超过 16 个字符。如果字符串参数长度超过 16，那么容量就会设置为 32。每次一旦超过 StringBuilder 对象的容量，那么容量就会增加 16 个字符。

7·2·2 获取并且设置关于 StringBuilder 对象的信息

在 StringBuilder 类中有几种属性可以用来获取有关 StringBuilder 对象的信息。Length 属性指定了当前实例中字符的数量，而 Capacity 属性则返回了实例的当前容量。MaxCapacity 属性会返回对象当前实例中所允许的最大字符数量（尽管这个数量会随着对象添加更多的字符而自动增加）。

下面的程序段说明了这些属性的用法：

StringBuilder stBuff = new StringBuilder(“Ken Thompson“);

Console·WriteLine(“Length of stBuff3: “ +& stBuff·Length·ToString()());

Console·WriteLine(“Capacity of stBuff3: “ &+ stBuff·Capacity()·ToString());

Console·WriteLine(“Maximum capacity of stBuff3: “ + stBuff·MaxCapacity·ToString());

Length 属性也可以用来设置 StringBuilder 对象的当前长度，就如同下面这样：

```
stBuff·Length = 10;

Console·Write(stBuff3);
```

这段代码的输出是”Ken Thomps”。

为了确信保持了适于 StringBuilder 实例的最小容量，可以调用 EnsureCapacity 方法，并且传递一个整数来说明适于对象的最小容量。下面是实例：

```
stBuff·EnsureCapacity(25);
```

另外一种可用的属性是 Chars 属性。这种属性既会返回在参数中指定位置上的字符，也会设置字符作为参数来传递。下面的代码显示了使用 Chars 属性的一个简单实例：

```
StringBuilder stBuff = Nnew StringBuilder(“Ronald Knuth“);

lf (stBuff·Chars[0](0) !=<> ‘“D’”c)

stBuff·Chars(0)[0] = ‘“D’”;
```

7·2·3 修改 StringBuilder 对象

对 StringBuilder 对象的修改包括在对象末尾处添加新的字符，在对象中插入字符，在含有不同字符的对象中替换一套字符，以及从对象中移除掉字符。本小节将会讨论和这些操作相关的方法。

通过使用 Append 方法可以在 StringBuilder 对象的末尾处添加字符。这种方法会取一个字符串值作为参数，并且把字符串串连到对象中当前值的末尾。现面这个程序说明了 Append 方法的工作原理：

```
using System;

uUsing System·Text;

class chapter7

{

    static void Main()

    {

        StringBuilder stBuff =As nNew StringBuilder();

        String[] words = new string[] {"now ", "is ", "the ", "time ", "for ", "all ",

            "good ", "men ", "to ", "come ", "to ", "the ","aid ", "of ", "their ", "party"};
```

```
        fFor(int i = 0; i <= words·GetUpperBound(0); i++)

                stBuff·Append(words[indexi]);

        Console·WriteLine(stBuff);

    }

}
```

显然程序的输出是：

```
Hello，world

Hello there，word
```

还可以给 StringBuilder 对象添加格式字符串。所谓格式字符串就是有格式说明嵌在内的字符串。本小节要涵盖太多的格式说明了，所以这里只示范一种常见的说明。可以如下所示这样在 StringBuilder 对象内部替换格式数：

```
using System;

uUsing System·Text;
```

```
class chapter7

{

    static void Main()

    {

        StringBuilder stBuff = nNew StringBuilder();

        Console·WriteLine();

        stBuff·AppendFormat("Your order is for {0000} widgets·", 234);

        stBuff·AppendFormat("\nWe have {0000} widgets left·", 12);

        Console·WriteLine(stBuff);

    }

}
```

此程序的输出是：

（原书P141页 截图）

格式说明是用一对大括号包裹着嵌入到文字串内。在执行代码时会把逗号后边的数据替换成格式说明。请参见C#语言针对格式说明的完整列表文档。

接下来是Insert方法。这种方法允许在当前StringBuilder对象中插入字符串。此方法会取得三个参数。第一个参数说明了插入的开始位置。第二个参数则是要插入的字符串。而作为可选项的第三个参数则是一个整数，它用来说明打算在对象中插入字符串的次数。

下面这段小程序说明了Insert方法的用法：

```
uUsing System·Text;

using System;

class chapter7
```

```
{

static void Main()

{

        StringBuilder stBuff = nNew StringBuilder();

        stBuff·Insert(0, "Hello");

        stBuff·Append("world");

        stBuff·Insert(5, ", ");

        Console·WriteLine(stBuff);

        char [] chars[]  = new char[] { 't', 'h', 'e', 'r', 'e' };

        stBuff·Insert(5, " " + new string(chars));

        Console·WriteLine(stBuff);
```

```
    }

}
```

程序的输出是

and on and on and on and on and on and on

下面这段程序采用了带第三个参数的 Insert 方法来说明要插入的次数：

```
StringBuilder stBuff = nNew StringBuilder();

stBuff·Insert(0, "and on ", 6);

Console·WriteLine(stBuff);
```

此段程序的输出是

and on and on and on and on and on and on

StringBuilder 类用 Remove 方法可以把字符从 StringBuilder 对象中移除掉。这种方法包含两个参数：开始位置和要移除掉的字符的数量。下面是此方法的工作原理：

```
StringBuilder stBuff = nNew StringBuilder("noise in+++++string");
```

```
stBuff·Remove(9, 5);

Console·WriteLine(stBuff);
```

此段程序的输出是

```
noise in string
```

当然，还可以用 Replace 方法来替换 StringBuilder 对象的字符。这种方法有两个参数：要替换的旧字符串和要放在替换位置上的新字符串。下列代码段就实例说明了如何使用这种方法：

```
StringBuilder stBuff = nNew StringBuilder("recieve decieve reciept");

stBuff·Replace("cie", "cei");

Console·WriteLine(stBuff);
```

这里会用“cei”替换掉每一个“cie”。

在处理 StringBuilder 对象时，经常需要把它们转换成字符串，这大概是为了使用一种在 StringBuilder 类中没有发现的方法。这可以用 ToString 方法来实现。这种方法会返回当前 StringBuilder 实例的一个字符串实例。如下例所示：

```
using System;

uUsing System·Text;
```

```
class chapter7

{

    static void Main()

    {

        StringBuilder stBuff = nNew StringBuilder("HELLO WORLD");

        string st = 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);
```

```
    }

}
```

此程序为了显示出字符串“Hello world”，首先把 stBuff 转换成字符串（st 变量），然后把字符串中的所有字符变为小写字母，接着再把字符串中的首字母大写，最后再用 st 值替换掉 StringBuilder 对象中的旧字符串。Replace 方法的第一个参数用到了 ToString 方法，因为第一个参数就假设成字符串了。这里是无法直接调用 StringBuilder 对象的。

7.3 STRING 类与 STRINGBUILDER 的性能比较

本章会以 String 类与 StringBuilder 类的性能比较的讨论作为结束。大家都知道 String 对象是不可变的，而 StringBuilder 对象则不是这样的。当然可以有理由相信 StringBuilder 类是更加高效率的。但是，人们不是总会使用 StringBuilder 类的，这是因为 StringBuilder 类缺少几种能够合理有效进行字符串处理的方法。事实是在需要使用 String 方法的时候（请参见前面的章节内容）可以把 StringBuilder 对象转换成 String 对象（稍后再转换回去）。但是需要知道何时要用 StringBuilder 对象，以及何时只要继续用 String 对象就可以了。

这里用到的测试非常简单。程序有两个子程程：一个构建了指定大小的 String 对象，而另一个构建了同样大小的 StringBuilder 对象。利用本书开始处开发的 Timing 类的对象对每个进程进行计时。这个过程重复了三次，第一次用于构建 100 个字符的对象，然后是用于 1000 个字符的对象，最后则是用于 10000 个字符的对象。接着，程序成对列出了每种规模的时间。这里所采用的代码如下所示：

```
using System;

using System.Text;
```

```
class chapter7

{

    static void Main()

    {

        int size = 100;

        Timing timeSB = new Timing();

        Timing timeST = new Timing();

        Console·WriteLine();

        for (int i = 0; i <= 3; i++)
```

```
    {

        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;

        }

}

static void BuildSB(int size)

{

        StringBuilder sbObject = new StringBuilder();

        for (int i = 0; i <= size; i++)

            sbObject·Append("a");

}

static void BuildString(int size)

{
```

```
            string stringObject = "";

            for (int i = 0; i <= size; i++)

                stringObject += "a";

    }

}
```

这里是程序的结果

（原书P145页 截图）

对于相对较小的对象来说，String 对象和 StringBuilder 对象之间真的没有什么差别。事实上，还可以争论对于达到 1000 个字符的字符串而言使用 String 类和使用 StringBuilder 类效果是一样的。然而，当达到 10000 个字符时，StringBuilder 类会有巨大的效率提升。尽管在 1000 个字符和 10000 个字符之间存在巨大的差异。但是，在练习中大家将会有机会

对含有超过 *1000* 个但少于 *10000* 个字符的对象做比较。

小结

在大多数 C#程序中对字符串的处理都是常见的操作。String 类为在字符串上要执行的每种操作都提供了多种多样的方法。虽然“经典的”内置字符串函数（Mid、lnStr 等等）始终可用，但是无论就性能还是清晰度而言大家会喜欢 String 类更甚于这些函数。

在 C#语言中 String 类是不可变的，这就意味着每次对象进行改变时都要创建一个新的对象副本。如果在创建长的字符串，或者是对相同对象进行许多改变，那么就应该用 StringBuffer 类来代替。StringBuffer 对象是可变的，这会涉及更多较好的性能。这里展示了当创建的 String 对象和 StringBuilder 对象在长度上超过 *1000* 个字符时的时间测试。

练习

请编写一个函数用来把短语转换成为字母顺序颠倒的特殊句。颠倒的要求是把单词中的第一个字母移到单词的末尾，并且再在单词的末尾添加字符“ay”。例如，短语“hello world”颠倒后应该变成“ellohay orldway”。此函数假设每个单词最少由两个字母组成，而且单词之间用空格进行分割，没有标点符号。

请编写一个函数用来统计字符串中单词的出现次数。函数应该返回一个正数值。不要假设只有一个空格来分割单词，而且字符串可能还包含标点符号。要编写的函数要既可以处 理 String 参数也可以处理 StringBuilder 对象。

请编写一个函数可以获取诸如 *52* 这样的数，然后返回类似 fifty-two 这样的数的单词形式。

请编写一个子程序会把一条包含主-谓-宾格式的简单句分解成不同部分。例如，简单句

“Mary walked the dog”就可以分解成：

主语：Mary

谓语：walked

宾语：the dog

此函数应该既可以处理 String 对象也可以处理 StringBuilder 对象。

第 8 章 模式匹配和文本处理

尽管 String 类和 StringBuilder 类提供了一套方法用来处理基于字符串的数据，但是 RegEx 和它的支持类却为字符串处理任务提供了更强大的功能。字符串的处理主要包括寻找字符串中的模式（模式匹配），以及通过称为正则表达式的特殊语言来执行操作。在本章大家会了解到形成正则表达式的方法以及如何利用它们解决普通文本处理任务。

8.1 正则表达式概述

所谓正则表达式是一种用于描述字符串中字符格式的语言，它提供了对应于重复字符、替换符符以及分组字符的描述符。正则表达式既可以用来执行字符串的搜索，也可以用于字符串的替换。

正则表达式本身就是一个定义了用于其他字符串搜索模式的字符串。通常情况下，正则表达式中的字符与其自身匹配，因此正则表达式“the”可以与字符串中任意位置找到的同样字符序列相匹配。

正则表达式还可以包含称之为元字符的特殊字符。元字符用于表示重复、替换或者分组。这

里将简要说明一下这些元字符的用法。

大多数有经验的计算机用户在工作中都会用到正则表达式，即使那时他们并没有意识到正在这样做。无论何时人们在命令提示符下敲入下列命令：

```
C:\>dir  myfile.exe
```

那么“myfile.exe”就是正则表达式。把正则表达式传递给 dir（目录文件显示）命令，然后在文件系统中任何与“myfile.exe”相匹配的文件都会显示在屏幕上。

许多用户还会在正则表达式中用到元字符。当用户敲入下列内容时：

```
C:\>dir  *.cs
```

这样就会用到含有元字符的正则表达式了。“*.cs”是正则表达式。而星号（*）是元字符，这意味着“匹配零个或更多个字符”。然而，表达式的剩余部分“.cs”就只是在文件中找到的普通字符了。这个正则表达式说明“匹配所有扩展名为‘cs’且文件名任意的文件”。此正则表达式传递给 dir（目录文件显示）命令，接着屏幕上就会显示出扩展名为.cs 的所有文件。

当然，人们还可以构建并使用许多更为强大的正则表达式，前边这两个实例只是作为一个善意的介绍。现在一起来看看如何在 C#语言中使用正则表达式以及它们是多么的有用。

8.1.1 概述：使用正则表达式

为了使用正则表达式，需要把 RegEx 类引入程序。大家可以在 System·Text·RegularExpression 名字域中找到这种类。

一旦把这种类导入了程序，就需要决定想要用 RegEx 类来做什么事情了。如果想要进行匹配，就需要使用 Match 类。如果打算做替换，则不需要 Match 类了。取而代之的是要用到 RegEx 类的 Replace 方法。

首先来看看如何在字符串中进行单词匹配操作吧。假设给定一个样例字符串“the quick brown fox jumped over the lazy dog”，这里想要在字符串中找到单词“the”。下面的程序完成了这项任务：

```
using System;

using System·Text·RegularExpressions;

class chapter8

{

    static void Main()

    {

        Regex reg = nNew Regex("the");
```

```
        string str1 = "the quick brown fox jumped over the lazy dog";

        Match matchSet;

        int matchPos;

        matchSet = reg.Match(str1);

        iIf (matchSet.Success)

        {

            matchPos = matchSet.Index;

            Console.WriteLine("found match at position:" + matchPos);

        }

    }

}
```

程序做的第一件事就是创建一个新的 RegEx 对象并且把要匹配的正则表达式传递给构造器。当再次初始化了字符串之后，程序声明了一个 Match 对象 matchSet。Match 类为存储用来与正则表达式进行匹配的数据提供了方法。

If 语句使用了一种 Match 类的属性 Success 来确定是否是成功匹配。如果值返回为 True，那么正则表达式在字符串中至少匹配了一条子串。否则的话，存储在 Success 中的值就是 False。

程序还可以有另外一种方法来查看是否匹配成功。通过把正则表达式和目标字符串传递给 IsMatch 方法的方式可以对正则表达式进行预测试。如果与正则表达式产生了匹配，那么这种方法就返回 True，否则返回 False。这种方法的操作如下所示：

```
if (Regex.IsMatch(str1, "the"))

{

    Match aMatch;

aMatch = reg.Match(str1);

}
```

用 Match 类的一个问题就是它只能存储一个匹配。在前面的实例中，针对子串“the”存在两个匹配。这里可以使用另外一种类 Matches 类来存储与正则表达式的多个匹配。为了处理所有找到的匹配可以把匹配存储到 MatchCollection 对象中。这里有一个实例（在 Main 函数内只包含了代码）：

```
using System;
```

```
using System·Text·RegularExpressions;

class chapter8

{

    static void Main()

    {

        Regex reg = new Regex("the");

        string str1 = "the quick brown fox jumped over the lazy dog";

        MatchCollection matchSet;

        matchSet = reg·Matches(str1);

        if (matchSet·Count > 0)

            foreach (Match aMatch in matchSet)
```

```
                Console·WriteLine("found a match at: " + aMatch·Index);

        Console·Read();

    }

}
```

接下来要讨论如何用 Replace 方法把一个字符串用另一个字符串来替换。Replace 方法可以作为一种带有三个参数的类方法来进行调用：一个目标字符串，要替换的子串，以及用作替换的子串。这段代码就用到了 Replace 方法：

```
string s = "the quick brown fox jumped over the brown dog";

s = Regex·Replace(s, "brown", "black");
```

现在会把字符串读作：“the quick black fox jumped over the black dog”。

针对模式匹配和文本处理这里有许多 RegEx 和支持类的用法。本章还将继续钻研讨论如何形成和使用更加复杂的正则表达式。

8·2 数量词

在编写正则表达式的时候，经常会要想正则表达式添加数量型数据，诸如“精确匹配两次”或者“匹配一次或多次”。利用数量词就可以把这些数据填加到正则表达式里面了。

这里要看到的第一个数量词就是加号（+）。这个数量词说明正则表达式应该匹配一个或多个紧接的字符。下面的程序就举例说明了这个数量词的用法：

```
using System;

using System.Text.RegularExpressions;

class chapter8

{

    static void Main()

    {

        string[] words = new string[] {"bad", "boy", "baaad","bear", "bend"};

        foreach (string word in words)

            if (Regex.IsMatch(word, "ba+"))

                Console.WriteLine(word);
```

```
    }

}
```

要匹配的单词是“bad”和“baaad”。正则表达式指明每一个以字母“b”开头并且包含一个或多个字母“a”的字符串都会产生匹配。

有较少限制的数量词就是星号（*）。这个数量词说明正则表达式应该匹配零个或多个紧接的字符。但是在实践中这个数量词非常难用，因为星号通常会导致匹配几乎所有内容。例如，利用前面的代码，如果把正则表达式变成读取“ba*”，那么数组中的每个单词都会匹配。

问号（？）是一种精确匹配零次或一次的数量词。如果把先前代码中的正则表达式变为“ba?d”，那么只有一个单词“bad”可以匹配。

通过在一对大括号内部放置一个数可以说明一个有限数量的匹配，就像在{n}中，这里的 n 是要找到的匹配数量。下面的程序说明了这个数量词的用法：

```
using System;

using System.Text.RegularExpressions;

class chapter8

{
```

```
    static void Main()

    {

        string[] words = new string[] {"bad", "boy", "baad","baaad", "bear", "bend"};

        foreach (string word in words)

            if (Regex.IsMatch(word, "ba{2}d"))

                Console.WriteLine(word);

    }

}
```

这里的正则表达式只能匹配字符串“baad”。

通过在大括号内提供两个数字可以说明匹配的最大值和最小值：{n,m}，这里的 n 表示匹配的最小值而 m 则表示最大值。在上述字符串中，正则表达式“ba{1,3}d”将可以匹配“bad”、“baad”以及“baaad”。

到目前为止已经讨论过的数量词展示的就是所谓的贪心行为。他们试图有尽可能多的匹配，而且这种行为经常会导致不预期的匹配。下面是一个例子：

```
using System;

using System.Text.RegularExpressions;

class chapter8

{

    static void Main()

    {

        string[] words = new string[]{"Part", "of", "this","<b>string</b>", "is", "bold"};

        string regExp = "<.*>";

        MatchCollection aMatch;

        foreach (string word in words)
```

```
        {

            if (Regex.IsMatch(word, regExp))

            {

                aMatch = Regex.Matches(word, regExp);

                for (int i = 0; i < aMatch.Count; i++)

                    Console.WriteLine(aMatch[i].Value);

            }

        }

    }

}
```

原本期望这个程序就返回两个标签：和。但是由于贪心，正则表达式匹配了字符串。利用懒惰数量词：问号（？）就可以解决这个问题。当问号直接放在原有数量

词后边时，数量词就变懒惰了。这里的懒惰是指在正则表达式中用到的懒惰数量词将试图做尽可能少的匹配，而不是尽可能多的匹配了。

把正则表达式变成读取“<.+>”也是于事无补的。这里需要用到懒惰数量词，而且一旦用了“<.+？>”，就会得到正确的匹配：和。懒惰数量词还可以和所有数量词一起使用，包括包裹在一对大括号内的数量词。

8.3 使用字符类

接下来这一小节会讨论如何用主要元素来构成正则表达式。首先从字符类开始。字符类匀速用户说明基于一串字符的模式。

这里第一个要讨论的字符类就是句号（.）。这是一种非常非常容易使用的字符类，但是它也确实是有问题的。句点与字符串中任意字符匹配。下面是一个实例：

```
using System;

using System.Text.RegularExpressions;

class chapter8

{

    static void Main()
```

```
    {

        string str1 = "the quick brown fox jumped over the lazy dog";

        MatchCollection matchSet;

        matchSet = Regex.Matches(str1, ".");

        foreach (Match aMatch in matchSet)

            Console.WriteLine("matches at: " + aMatch.Index);

    }

}
```

从这段程序的输出可以说明句点的工作原理：

（原书P154页 截图）

句点可以匹配字符串中每一个单独字符。

较好利用句点的方法就是用它在字符串内部定义字符范围，也就是用来限制字符串的开始或/和结束字符。下面是使用相同字符串的一个实例：

```
using System;

using System.Text.RegularExpressions;

class chapter8

{

    static void Main()

    {

        string str1 = "the quick brown fox jumped over the lazy dog one time";

        MatchCollection matchSet;
```

```
        matchSet = Regex.Matches(str1, "t.e");

        foreach (Match aMatch in matchSet)

            Console.WriteLine("Matches at: " + aMatch.Index);

    }

}
```

程序的输出是：

```
matches: the at: 0

matches: the at: 32
```

在使用正则表达式的时候经常希望检查包含字符组的模式。大家可以编写用一组闭合的方括号（[ ]）包裹着的正则表达式。在方括号内的字符被称为是字符类。如果想要编写的正则表达式匹配任何小写的字母字符，可以写成如下这样的表达式：[abcdefghijklmnopqrstuvwxyz]。但是这样是很难书写的，所以通过连字号：[a-z]来表示字母范围的方式可以编写简写版本。

下面说明了如何利用这个正则表达式来匹配模式：

```
using System;
```

```
using System·Text·RegularExpressions;

class chapter8

{

    static void Main()

    {

        string str1 = "THE quick BROWN fox JUMPED over THE lazy DOG";

        MatchCollection matchSet;

        matchSet = Regex·Matches(str1, "[a-z]");

        foreach (Match aMatch in matchSet)

            Console·WriteLine("Matches at: " + aMatch·Index);

    }
```

}

程序中匹配的字母就是那些组成单词“quick”、“fox”、“over”和“lazy”的字母。

字符类可以用多组字符构成。如果想要既匹配小写字母也匹配大写字母，那么可以把正则表达式写成这样：“*[A-Za-z]*”。当然，如果需要包括全部十个数字，也可以编写像*[0-9]*这样由数字组成的字符类。

此外，通过在字符类前面放置一个脱字符号（^）的方法人们还可以创建字符类的反或者字符类的否定。例如，如果有字符类*[*aeiou*]*来表示元音类，那么就可以编写*[*^aeiou*]*来表示辅音或非元音。

如果把这三个字符类合并，就可以形成正则表达式用法中所谓的单词。正则表达式就像这个样子：*[*A-Za-z*0-9]*。这里还有一个可以用来表示同样类的较短小的字符类：\w。\W 用来表示\w 的反，也或者用来表示非单词字符（比如标点符号）。

此外，还可以把数字字符类（*[0-9]*）写成\d（注意由于在 C#语言中反斜杆后跟着其他字符很可能是转义序列，所以诸如\d 这样的代码在 C#语言中都以\\d 形式来说明正则表达式而非转义代码）。而非数字字符类（*[^0-9]*）则可以写成\D 这样。最后，因为空格符在文本处理中扮演着非常重要的角色，所以把\s 用来表示空格字符，而把\S 用来表示非空格字符。稍后在讨论分组构造时将会研究使用空白字符类。

8.4 用断言修改正则表达式

C#语言包含一系列可以添加给正则表达式的运算符。这些运算符可以在不导致正则表达式引擎遍历字符串的情况下改变表达式的行为。这些运算符被称为断言。

第一个要研究的断言会导致正则表达式只能在字符串或行的开始处找到匹配。这个断言由脱字符号（^）产生。在下面这段程序中，正则表达式只与第一个字符为字母“h”的字符串相匹配，而忽略掉字符串中其他位置上的“h”。代码如下所示：

```
using System;

using System.Text.RegularExpressions;

class chapter8

{

    static void Main()

    {

        string[] words = new string[] { "heal", "heel", "noah", "techno" };

        string regExp = "^h";

        Match aMatch;
```

```
        foreach (string word in words)

            if (Regex.IsMatch(word, regExp))

            {

                aMatch = Regex.Match(word, regExp);

                Console.WriteLine("Matched: " + word + " at position: " +
aMatch.Index);

            }

    }

}
```

这段代码的输出就只有字符串“heal”和“heel”匹配。

这里还有一个断言会导致正则表达式只在行的末尾找到匹配。这个断言就是美元符号（$）。

如果把前一个正则表达式修改成如下形式：

```
string regExp = "h$";
```

那么“noah”就是唯一能找到的匹配。

此外，另有一个断言可以在正则表达式中式指定所有匹配只能发生在单词的边缘。这就意味着匹配只能发生在用空格分隔的单词的开始或结束处。此断言用\b 表示。下面是此断言的工作过程：

```
string words = "hark, what doth thou say, Harold? ";
```

```
string regExp = "\\bh";
```

这个正则表达式与字符串中的单词“hark”和“Harold”相匹配。

在正则表达式中还可以使用其他一些断言，但是上述三种是最普遍用到的断言。

8.5 使用分组构造

RegEx 类有一套分组构造可以用来把成功匹配进行分组，从而使字符解析成相关匹配更容易。例如，给定了生日和年龄的字符串，而用户只想确定日期的话。通过把日期分组到一起，就可以确定它们作为一组，而不再需要单独进行匹配了。

8.5.1 匿名组

这里可能用到几个不同的分组构造。通过括号内围绕的正则表达式就可以组成第一个构造。正如不久要介绍的一样，既然也可以命名组，大家就可以考虑把这个构造作为匿名组。作为一个实例，请看看下列字符串：

“08/14/57 46 02/2/29 45 06/05/85 18 03/12/88 16 09/09/90 13”

这个字符串就是由生日和年龄组成的。如果只需要匹配年龄而不要生日，就可以把正则表达式作为一个匿名组来书写：

（\\s\\d{2}\\s）

通过编写这种方式的正则表达式，字符串中的每个匹配就从*1*开始的数字进行确认。为全部匹配保留数字零，这通常将会包含更多的数据。下面这段小程序就用到了匿名组：

```
using System;

using System·Text·RegularExpressions;

class chapter8

{

    static void Main()
```

```
    {

        string words = "08/14/57 46 02/25/59 45 06/05/85 18" + "03/12/88
16 09/09/90 13";

        string regExp1 = "(\\s\\d{2}\\s)";

        MatchCollection matchSet = Regex·Matches(words,regExp1);

        foreach (Match aMatch in matchSet)

            Console·WriteLine(aMatch·Groups[0]·Captures[0]);

    }

}
```

8·5·2 命名组

组通常用名字构建。命名的组更容易使用，这是因为在重新找到匹配时可以通过名字引用到组。命名组是由作为正则表达式前缀的问号和一对尖括号包裹的名字组成的。例如，为了在

先前的程序代码“ages”中命名组，可以把正则表达式写成下列形式：

```
(?<ages>\\s\\d{2}\\s)
```

还可以用一对小括号来代替尖括号包裹名字。

现在要来修改一下这个程序，使得此程序寻找日期而不是年龄，而且用分组构造来组织日期。
下面是代码：

```
using System;

using System.Text.RegularExpressions;

class chapter8

{

    static void Main()

    {

        string words = "08/14/57 46 02/25/59 45 06/05/85 18 " + "03/12/88
16 09/09/90 13";
```

```
        string regExp1 = "(?<dates>(\\d{2}/\\d{2}/\\d{2}))\\s";

        MatchCollection matchSet = Regex.Matches(words,regExp1);

        foreach (Match aMatch in matchSet)

            Console.WriteLine("Date: {0}", aMatch.Groups["dates"]);

    }

}
```

这里是输出：

（原书 P159 页 截图）

下面集中在用正则表达式来产生输出：

```
(\\d{2}/\\d{2}/\\d{2})\\s
```

大家可以把这个表达式读作“2 个数字跟着一条斜线，再跟着两个数字和一条斜线，再跟着两个数字和一条斜线，再跟着一个空格”。为了给正则表达式分组，可以做下列添加：

```
(?<dates>(\\d{2}/\\d{2}/\\d{2}))\\s
```

为了找到字符串中的每个匹配，需要用 Match 类的 Group 方法来把它们分离成组：

```
Console·WriteLine("Date: {0}", aMatch·Groups("dates"));
```

8·5·3 零宽度正向预搜索断言和零宽度反向预搜索断言

断言还可以用来确定正则表达式向前或向后搜索到匹配的程度。这些断言可能是正的或负的，这就意味着正则表达式在寻找特殊的匹配模式（正的）或特殊的非匹配模式（负的）。觉刊到一些市里的时候这些内容就会更清楚了。

这些断言中第一个要讨论的就是正的正向预搜索断言。此断言进行了如下这样的说明：

```
(?= reg-exp-char)
```

这里的 reg-exp-char 是正则表达式或元字符。此断言说明只要搜索到匹配的当前子表达式在指定位置的右侧，那么匹配就继续。下面这段代码说明了此断言的工作原理：

```
string words = "lions lion tigers tiger bears,bear";
```

```
string regExp1 = "\\w+(?=\\s)";
```

正则表达式说明对跟随空格的每个单词都做了匹配。匹配的单词有“lions”、“lion”、“tigers”和“tiger”。正则表达式匹配单词，但是不匹配空格。记住这一点是非常重要的。

下一个断言是负的正向预搜索断言。只要搜索到不匹配的当前子表达式在指定位置的右侧，那么此断言就继续匹配。下面是代码段实例：

```
string words = "subroutine routine subprocedure procedure";
```

```
string regExp1 = "\\b(?!sub)\\w+\\b";
```

此正则表达式表明对每个单词所做的匹配不是以前缀“sub”开始的。匹配的单词有“routine”和“procedure”。

接下来的断言被称为是反向预搜索断言。这些断言会正向左或反向左搜索，而不是向右了。下面的代码段距离说明了如何编写一个正的反向预搜索断言：

```
string words = "subroutines routine subprocedures

procedure";
```

```
string regExp1 = "\\b\\w+(?<=s)\\b";
```

这个正则表达式搜索出现在“s”后的单词的所有边缘。匹配的单词有“subroutines”和“subprocedures”。

只要子表达式不匹配在位置的左侧，那么负的反向预搜索断言就继续匹配。这里可以很容易的修改上述提到的正则表达式，使得其就只能匹配不是以字母“s”结尾的单词，就像下面

这样：

```
string regExp1 = "\\b\\w+(?<!s)\\b";
```

8.6 CAPTURESCOLLECTION 类

当正则表达式匹配子表达式的时候，产生了一个被称为是 Capture 的对象，而且会把此对象添加到名为 CaptureCollection 的集合里面。当在正则表达式中使用命名组的时候，这个组就由自己的捕获集合。

为了重新得到用了命名组的正则表达式所收集的捕获，就要调用来自 Match 对象 Group 属性的 Captures 属性。这在实例中是很容易理解的。利用前面小节的其中一个正则表达式，下列代码返回了在字符串中找到的所有日期和年龄，而且日期和年龄是完全分组的：

```
using System;

using System.Text.RegularExpressions;

class chapter8

{

    static void Main()

    {

        string dates = "08/14/57 46 02/25/59 45 06/05/85 18 " + "03/12/88
```

```
16 09/09/90 13";

        string regExp =
"(?<dates>(\\d{2}/\\d{2}/\\d{2}))\\s(?<ages>(\\d{2}))\\s";

        MatchCollection matchSet;

        matchSet = Regex.Matches(dates, regExp);

        Console.WriteLine();

        foreach (Match aMatch in matchSet)

        {

            foreach (Capture aCapture in aMatch.Groups["dates"].Captures)

                Console.WriteLine("date capture: " + aCapture.ToString());

            foreach (Capture aCapture in aMatch.Groups["ages"].Captures)

                Console.WriteLine("age capture: " + aCapture.ToString());
```

```
        }

    }

}
```

此程序的输出是：

（原书 P162 页 截图）

程序的外循坏遍历了每个匹配，而两个内循环则遍历了不同的 Capture 集合，一个是日期集合而另一个则是年龄集合。按照这种方式使用 CaptureCollection 可确保捕获每组匹配而不仅仅是最后的匹配。

8.7 正则表达式的选项

在指定正则表达式的时候可以设置几个选项。这些选项的范围从指定多行模式以便正则表达式可以在多行上正确工作，到编译正则表达式以便能更快速执行。下面这张表列出了可以设置的不同选项。

在查看此表之前，需要注意这些选项的设置方式。通常情况下，对 RegEx 类的方法之一指定选项常量作为第三个参数就可以设置选项了，比如 Match 方法、Matches 方法。例如，如果想要为正则表达式设置 Multiline 选项，代码行应像下面这样：

```
matchSet = Regex.Matches(dates, regexp, RegexOptions.Multiline);
```

这个选项连同其他选项可以直接输入也可以用 Intellisense 来选择。

下面就是可用的选项了：

RegexOption 成员
内置字符
描述

None
N/A
说明没有选项设置。

IgnoreCase
I
说明字母非大小写匹配。

Multiline
M
说明多行模式。

ExplicitCapture
N
说明只有对正确的捕获明确命名或计算组。

Compiled

N/A

说明将会对正则表达式编译成汇编。

Singleline

S

说明单行模式。

IgnorePatternWhiteSpace

X

说明由于模式而排斥非转义空格，而且使注释跟在符号（#）之后。

RightToLeft

N/A

说明搜索是从右到左，而不是从左到右。

ECMAScript

N/A

说明 ECMAScript-compliant 行为对表达式有效。

小结

正则表达式提供了强大的选项来执行文本处理和模式匹配。正则表达式可以极为简单（“a”），也可以是非常复杂的组合，以至于像是噪音而不是可执行的代码。 虽然如此，但是学会使用正则表达式将会让大家在原文上执行文本处理，甚至不用考虑使用诸如 String 类的方法这样的工具。

本章只含蓄的介绍了正则表达式的强大。如果想要了解更多有关正则表达式的内容，请参考 Feiedel 的书（*1997* 年出版）。

练习

16. 请编写正则表达式来匹配下列内容：

a) 由一个字母“x”，跟着任意三个字符，然后是一个字母“y”组成的字符串。

b) 以“ed”结尾的一个单词。

c) 一个电话号码。

d) 一个 HTML 链接标记。

17. 请编写一个正则表达式用来在字符串中找到所有包含两个重复字母的单词，比如单词“deep”和单词“book”。

18. 请编写一个正则表达式用来在网页中找到所有头标记（<h*1*>、<h*2*>等等）。

19. 请利用在字符串中执行简单搜索和替换的正则表达式来编写一个函数。

第 9 章 构建字典：DictionaryBase 类和 SortedList 类

字典是一种把数据作为键值对来存储的数据结构。作为一种抽象的类，DictionaryBase 类可以用来实现不同的数据结构，其中这些数据结构全部把数据存储成键值对。这些数据结构可能是散列表、链表或者其他一些数据结构类型。本章节会讨论如何创建基础字典，以及如何使用 DictionaryBase 类的继承方法。稍后当研究更加专有的数据结构的时候将会用到这些技术。

基于字典的数据结构的实例之一就是 SortedList。这个类是按照分类顺序基于键值来存储键值对的。这是一种有趣的数据结构，因为通过引用数据结构中值的索引位置也可以访问到存储在结构中的数据，这也使得结构的行为在某些方面和数组很相像。本章的最后会讨 论 SortedList 类的行为。

9·1DICTIONARYBASE 类

大家可以把字典数据结构看成是一种计算机化的词典。要查找的词就是关键字，而词的定义就是值。DictionaryBase 类是一种用作专有字典实现基础的抽象（MusInherit）类。

存储在字典中的键值对实际上是作为 DictionaryEntry 对象来存储的。DictionaryEntry 结构提供了两个域，一个用于关键字而另一个用于值。在这个结构中所要关注的只是 Key 属性和 Value 属性这两个属性（或方法）。当把键值对录入到字典内的时候，这些方法会返回存储的值。本章稍后会讨论 DictionaryEntry 对象。

就内部而言，会把键值对存储在被称为 InnerHashTable 的散列表对象中。本书的第 12 章会详细讨论散列表，所以现在只要把它看成是一种有效的用来存储键值对的数据就可以了。

DictionaryBase 类实际上实现了来自 System·Collections 名字空间的界面，即 IDictionary。此界面事实上是本书稍后要研究的许多类的基础，包括 ListDictionary 类和 Hashtable 类。

9·1·1DictionaryBase 类的基础方法和属性

在用字典对象进行操作的时候需要执行几种操作。就最少操作数量而言，需要 Add 方法来添加新数据，需要 Item 方法来取回数值，需要 Remove 方法来移除掉键值对，还需要 Clear 方法来清除所有数据的数据结构。

首先通过查看一个简单类来开始实现字典的讨论。下列代码说明了一个存储名字和 IP 地址的类的实现：

```
public class IPAddresses : DictionaryBase

{

    public IPAddresses()

    {

    }

    public void Add(string name, string ip)

    {

        base.InnerHashtable.Add(name, ip);

    }
```

```
    public string Item(string name)

    {

        return base.InnerHashtable[name].ToString();

    }

    public void Remove(string name)

    {

        base.InnerHashtable.Remove(name);

    }

}
```

正像看到的那样，这些方法是非常容易构建的。第一个实现的方法就是构造器。这是一种简单的方法，只需要调用针对基本类的默认构造器就行了。Add 方法把名字/IP 地址对取作参数，并且把它们传递给在基本类中实例化的 InnerHashTable 对象的 Add 方法。

Item 方法用来取回给定特殊关键字的值。这里把关键字传递给 InnerHashTable 对象相应的 Item 方法。然后会返回用关联的关键字存储在内部散列表中的值。

最后，Remove 方法把关键字作为参数来取回，并且把参数传递给关联的内部散列表 的 Remove 方法。稍后，方法会把关键字和与关键字相关联的值从散列表中移除掉。

下面来看看利用了这些方法的一个程序：

```
class chapter9

{

    static void Main()

    {

        IPAddresses myIPs = 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 address: " + myIPs.Item("David"));

            myIPs.Clear();

            Console.WriteLine("There are " + myIPs.Count + " IP addresses");

        }

}
```

此程序的输出是：

（原书 P168 页 截图）

这里对类可能要做的一个修改就是使构造器超载以便于把来自文件的数据装载到字典内。下面是新构造器的代码，此代码可以仅添加到 IPAddresses 类的定义内：

```
public IPAddresses(string txtFile)
```

```
{

    string line;

    string[] words;

    StreamReader inFile;

    inFile = File·OpenText(txtFile);

    while (inFile·Peek() != -1)

    {

        line = inFile·ReadLine();

        words = line·Split(',');

        this·InnerHashtable·Add(words[0], words[1]);

    }
```

```
    inFile·Close();

}
```

现在这是测试构造器的新程序：

```
class chapter9

{

    static void Main()

    {

        for (int i = 0; i < 4; i++)

            Console·WriteLine();

        IPAddresses myIPs = new IPAddresses(@"c:\data\ips·txt");

        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"));

    }

}
```

此程序的输出是：

（原书P169页 截图）

9·1·2 其他的 DictionaryBase 方法

这里还有其他两种方法，它们都是 DictionaryBase 类的成员：CopyTo 方法和 GetEnumerator 方法。本小节来讨论这些方法。

CopyTo 方法把字典的内容复制给一维的数组。尽管可以把数组声明成 Object，然后用 CType 函数把对象转换成为 DictionaryEntry，但是这里应该把数组声明成 DictionaryEntry 数组。

下面的代码段举例说明了 CopyTo 方法的用法：

```
IPAddresses myIPs = new IPAddresses(@"c:\ips.txt");

DictionaryEntry[] ips = new DictionaryEntry[myIPs.Count-1];

myIPs.CopyTo(ips, 0);
```

用来计算数组大小的公式会取得字典中元素的数量，然后为了说明是基于零的数组再把结果减去一。CopyTo 方法会得的两个参数：要复制到的数组和开始复制的索引位置。例如，如果需要在已存在数组的末尾放置字典的内容，就要指明数组的上限加一作为第二个参数。

一旦把字典的数据放入数组，就需要用数组的内容来进行处理，或者至少显示值。下面的代码就是做了这样的工作：

```
for (int i = 0; i <= ips.GetUpperBound(0); i++)

    Console.WriteLine(ips[i].ToString());
```

此段代码的输出是：

（原书 P170 页 截图）

可惜的是，这些不是所要的内容。问题是这里把数据作为 DictionaryEntry 对象存储在数组内，而这正是我们前面看到的结果。如果用 ToString 方法：

```
Console·WriteLine(ips[index]·ToString());
```

就可以得到同样的内容。为了真实地看到 DictionaryEntry 对象内的数据，就需要根据要查找的对象所包含的是关键字数据还是值数据来使用 Key 属性或 Value 属性。但是怎么才能知道哪个是 Key 哪个是 Value 呢？当字典的内容复制给数组的时候，数据的复制是根据键值的顺序进行的。所以第一个对象是关键字，接着的对象是值，第三个对象又是关键字，以此类推。

现在编写的代码就允许实际查看数据了：

```
for(int i = 0; i <= ips·GetUpperBound(0); i++)

{

Console·WriteLine(ips[index]·Key);

Console·WriteLine(ips[index]·Value);

}
```

输出是：

（原书 P177 页 截图）

9.2 通用的 KEYVALUEPAIR 类

C#语言提供了一种小类用来允许创建象字典式的对象，此对象是基于关键字来存储数据的。

这种类被称为是 KeyValuePair 类。由于每个对象只能持有一个关键字和一个值，所以它的使用是受到限制的。

一个 KeyValuePair 对象可以向下列这样实例化：

```
KeyValuePair<string, int> mcmillan = new KeyValuePair<string, int>("McMillan", 99);
```

这里会分别取回关键字和值：

```
Console.Write(mcmillan.Key);

Console.Write(" " + mcmillan.Value);
```

如果把对象放置在数组内，那么 KeyValuePair 类是比较好用的。下列程序举例说明如何实现简单等级书：

```
using System;

using System.Collections.Generic;

using System.Text;

namespace Generics

{

    class Program

    {

        static void Main(string[] args)

        {
```

```
            KeyValuePair<string, int>[] gradeBook = new KeyValuePair<string,
int>[10];

            gradeBook[0] = new KeyValuePair<string,int>("McMillan", 99);

            gradeBook[1] = new KeyValuePair<string,int>("Ruff", 64);

            for (int i = 0; i <= gradeBook·GetUpperBound(0); i++)

                if (gradeBook[i]·Value != 0)

                    Console·WriteLine(gradeBook[i]·Key + ": " +
gradeBook[i]·Value);

            Console·Read();

        }

    }

}
```

9·3SORTEDLIST 类

正如在本章介绍部分提到的那样，SortedList 是按照分类顺序基于键值来存储键值对。当存储的关键字很重要时可以使用这种数据结构。比如，在标准词典中希望所存储的词是按照字母的顺序存储的情况。本章稍后还将说明如何用类来保存一个单独分类的值表。

9·3·1 使用 SortedList 类

既然 SortedList 类是 DictionaryBase 类的特殊化，所以 SortedList 类可以按照许多和先前章节用类相同的方式来使用。

为了说明这一点，下面的代码创建了包含三个名字和 IP 地址的 SortedList 对象：

```
SortedList myips = new SortedList();

myips.Add("Mike", "192.155.12.1");

myips.Add("David", "192.155.12.2");

myips.Add("Bernica", "192.155.12.3");
```

这里的名字是关键字，而 IP 地址则是用来存储值。

SortedList 类的通用版本允许确定关键字和值两者的数据类型：

SortedList<Tkey, TValue>

例如，可以把 myips 象下面这样实例化：

```
SortedList<string, string> myips = new SortedList<string, string>();
```

可以对等级书所存储的表完成下列这样的实例化：

```
SortedList<string, int> gradeBook = new SortedList<string, int>();
```

通过使用把关键字作为参数的 Item 方法可以取回值：

```
fForeach(Object key iIn myips·Keys)

Console·WriteLine("Name：" +& key + "\n" + "IP：" +& myips·Item([key]));
```

这段代码产生了下列输出：

（原书 P173 页 截图）

换而言之，也可以通过引用索引数来访问这张表，其中索引数是这些数值（和关键字）存储在实际存储数据的数组内的位置。如下所示：

```
for(int i = 0; i < myips.Count; i++)
```

```
Console.WriteLine("Name： " + myips.GetKey(i) + "\n" + "IP： " +&
myips.GetByIndex(i));
```

这段代码产生了完全一样存储的名字和 IP 地址表：

（原书 P1743 页 截图）

就像下列代码段那样，通过指明关键字或指明索引数的方式可以把键值对从 SortedList 中移除掉。这段代码说明了这两种移除的方法：

```
myips.Remove("David");
```

```
myips.RemoveAt(1);
```

如果想要用基于索引的方式来存取 SortedList，但是又不知道存储的特殊关键字或值的索引位置，那么可以用下面的方法来确定这些值：

```
int indexDavid = myips.GetIndexOfKey("David");

int indexIPDavid = myips.GetIndexOfValue(myips.Item(["David")]);
```

SortedList 类还包含了许多其他方法，这里鼓励大家通过 VS.NET 的在线文版来研究讨论它们。

小结

DictionaryBase 类是用来创建用户字典的抽象类。而字典则是利用散列表（或者有时为单独的链表）作为潜在的数据结构来把数据存储到键值对内的一种数据结构。键值对作为 DictionaryEntry 对象来进行存储，而且必须使用 Key 方法和 Value 方法来取回 DictionaryEntry 对象中的实际值。

当程序员需要创建强类型的数据结构的时候，经常会用到 DictionaryBase 类。通常情况下会把添加给字典的数据作为 Object 来存储，但是对于自定义字典而言，程序员可能会削减需要执行的类型转换的数量，从而使得程序变得更加有效且更易于阅读。

SortedList 类是 Dictionary 类的一种特殊类型。它会按照分类的顺序通过关键字来存储键值对。此外，也可以通过引用索引数的方式来取回存储在 SortedList 中的值，其中索引数是存储数值的位置，这和使用数组非常相似。在 System.Collections.Generic 名字域中还有 SortedDictionary 可以像通用 SortedList 类一样操作。

练习

20. 请利用本章开发的 IPAddress 类的实现来编写一个显示 IP 地址的方法，其中那个 IP 地址是按照升序方式存储在类中的。并且把这种编写的方法用在程序内。

21· 请编写一个程序用来存储来自字典内文本文件的名字和电话号码，其中把名字作为关键字。并且编写方法来进行反向查找，也就是说根据电话号码来找到名字。还要编写一个窗口应用程序来测试实现。

22· 请利用字典编写一个程序用来显示一条句子内单词出现的次数。要把出现在句内的所有单词和它们出现的次数全部显示出来。

23· 请重新编写练习 3 的程序使得它可以处理字母而不是单词。

24· 请利用 SortedList 类来重新编写练习 2 的程序。

25· 用两个内部数组来实现 SortedList 类，其中一个数组存储关键字，而另一个数组则用来存储值。请用这种方案创建自己的 SortedList 类的实现。自创的类要包括本章内讨论到的所有方法。请用自创的类来解决练习 2 所提出的问题。

第 10 章 散列和 Hashtable 类

散列是一种常见的存储数据的技术，按照这种方式可以非常迅速地插入和取回数据。散列所采用的数据结构被称为是散列表。尽管散列表提供了快速地插入、删除、以及取回数据的操作，但是诸如查找最大值或最小值这样的查找操作，散列表却无法执行地非常快。对于这类操作，其他数据结构会更适合（请参考第 12 章在二叉查找树方面的实例）。

·NET 框架库提供了一种非常有用的处理散列表的类，即 Hashtable 类。本章会研究这个类，而且还将讨论如何实现自定义的散列表。构造散列表不是非常困难的事，而且所采用的编程

技术也很值得学习。

10·1 散列概述

散列表数据结构是围绕数组设计的。虽然可以稍后根据需要增加数组的大小，但是数组是由第 *0* 号元素一直到一些预定义尺寸的元素组成的。存储在数组内的每一个数据项都是基于一些数据块的，这被称为是关键字。为了把一个元素存储到散列表内，利用所谓的散列函数把关键字映射到一个范围从 *0* 到散列表大小的数上。

散列函数的理想目标是把自身单元内的每一个关键字都存储到数组内。然而，由于可能的关键字是不限制数量的，而数组的大小又是有限的，所以散列函数比较现实的目标是把关键字尽可能平均地分布到数组的单元内。

就像到如今大家可能猜测到的那样，即使用一个很好的散列函数也可能会出现两个关键字散列到相同数值的情况。这种现象被称为是冲突，而且在发生这种现象的时候就需要有策略来处理冲突。稍后会详细讨论这个问题。

最后一件需要确定的事情是用多大维数的数组作为散列表。首先，建议数组的大小是一个素数。在研究不同的散列函数的时候会解释说明原因。这之后会介绍几种不同的确定适当数组大小的策略，所有这些策略都是基于用来解决冲突的技术，所以在稍后的讨论中还会研究这个问题。

10·2 选择散列函数

选择散列函数是依据所用关键字的数据类型。如果所用的关键字是整数，那么最简单的函数是返回关键字对数组大小取模的结果。但是有些情况不建议使用这种方法，比如关键字都是以 *0* 结束，且数组的大小为 *10* 的情况。这就是数组的大小必须始终为素数的原因之一。此外，如果关键字是随机整数，那么散列函数应该更均匀地分布关键字。

然而，在许多应用程序中关键字都是字符串。选择处理关键字的散列函数会更加困难，而且还需要谨慎选择。乍看之下好像有一个简单有效的函数可以把关键字内字母的 ASCII 码值相加。上述加和的数值与数组的大小取模就是散列值了。下面的程序举例说明了此散列函数的工作原理：

```
using System;

class chapter10

{

    static void Main()

    {

        string[] names = new string[99];

        string name;

        string[] someNames = new string[]{"David","Jennifer", "Donnie", "Mayo",
"Raymond",

            "Bernica", "Mike", "Clayton", "Beata", "Michael"};
```

```
    int hashVal;

    for (int i = 0; i < 10; i++)

    {

        name = someNames[i];

        hashVal = SimpleHash(name, names);

        names[hashVal] = name;

    }

    ShowDistrib(names);

}

static int SimpleHash(string s, string[] arr)

{
```

```
    int tot = 0;

    char[] cname;

    cname = s.ToCharArray();

    for (int i = 0; i <= cname.GetUpperBound(0); i++)

        tot += (int)cname[i];

    return tot % arr.GetUpperBound(0);

}

static void ShowDistrib(string[] arr)

{

    for (int i = 0; i <= arr.GetUpperBound(0); i++)

        if (arr[i] != null)
```

```
                Console·WriteLine(i + " " + arr[i]);

    }

}
```

此程序的输出是：

（原书 P178 页 截图）

子程序 showDistrib 说明散列函数把名字实际放置在数组内的位置。就像大家可以看到的那样，分布不是特别地均匀。名字都聚集在在数组的开始处和末尾处。

然而，这里甚至还潜伏着一个更大的问题。并不是所有的名字都显示出来了。有趣地是，如果把数组的大小变为一个素数，即使是比 99 小的素数，那么就可以完全存储所有的名字了。因此，在为散列表选择数组大小的时候（或者在使用类似这里用到的散列函数的时候），一个重要的原则就是要选择素数。

最终选择的数组大小要取决于散列表中存储的记录的确定数量，但是一个看似保险的数是 10007（假设不是真的试图在散列表中存储过多的数据项）。10007 是素数，而且它没有大到会使用大量的内存来降低程序的性能。

这里继续有关在散列值的创建中计算全部关键字 ASCII 码值的基本想法，下一个算法就为在数组内的更好地分布提供了可能。首先来看看代码，稍后会有解释说明：

```
static int BetterHash(string s, string[] arr)

{

    long tot = 0;

    char[] cname;

    cname = s.ToCharArray();

    for (int i = 0; i <= cname.GetUpperBound(0); i++)

        tot += 37 * tot + (int)cname[i];

    tot = tot % arr.GetUpperBound(0);

    if (tot < 0)

        tot += arr.GetUpperBound(0);
```

```
    return (int)tot;

}
```

这个函数利用霍纳（Horner）法则来计算多项式函数（关于 37）。请参考（Weiss 1999 年）来获得更多有关此散列函数的信息。

现在再来看看采用这个新函数的散列表中关键字的分布情况：

（原书 P180 页 截图）

虽然用这样小的数据集合很难说明，但是上述这些关键字却是更加均匀地分布着。

10.3 查找散列表中数据

为了在散列表中查找数据，需要计算关键字的散列值，然后访问数组中的对应元素。就是这样简单。下面是函数：

```
static bool lnHash(string s, string[] arr)
```

```
{

    int hval = BetterHash(s, arr);

    if (arr[hval] == s)

        return true;

    else

        return false;

}
```

如果数据项在散列表内，那么这个函数会返回真值（True），否则返回假值（False）。这里甚至不需要把此函数的运行时间与数组顺序查找的时间进行比较，因为很明显此函数的运行时间少许多，当然除非数据项在靠近数组开始部分的某处。

10.4 解决冲突

在处理散列表的时候，不可避免地会遇到这种情况，即计算出的关键字的散列值已经存储了另外一个关键字。这就是所谓的冲突。在发生冲突的时候可以使用几种技术。这些技术包括桶式散列法、开放定址法、和双重散列法。本小节会主要介绍上述这三种技术。

10·4·1 桶式散列法

在初始定义散列表的时候，声明倾向于只有一个数据值驻存在散列表元素内。如果没有冲突，那么这项工作会顺利进行。但是如果散列函数为两个数据项返回了相同的数值，那么就会有问题了。

解决冲突问题的方案之一就是用桶来实现散列表。桶是一种存储在散列表元素内的简单数据结构，它可以存储多个数据项。在大多数实现中，这种数据结构就是一个数组，但是在这里的实现中将会使用 arraylist，它会允许不考虑运行超出范围而且允许分配更多的空间。最后，这种方数据结构会使实现更加高效。

为了插入一个数据项，首先用散列函数来确定哪一个 arraylist 用来存储数据项。然后查看此数据项是否已经在 arraylist 内。如果存在，就什么也不做。如果不存在，就调用 Add 方法把此数据项添加到 arraylist 内。

为了从散列表中移除一个数据项，还是先确定要移除的数据项的散列值，并且转到对应的 arraylist。然后查看来确信该数据项在 arraylist 内。如果存在，就把它移除掉。

下面是关于 BucketHash 类的代码，它包括一个 Hash 函数，一个 Add 方法和一个 Remove 方法：

```
public class BucketHash

{

    private const int SIZE = 101;

    ArrayList[] data;
```

```
public BucketHash()

{

    data = new ArrayList[SIZE];

    for (int i = 0; i <= SIZE - 1; i++)

        data[i] = new ArrayList(4);

}

public int Hash(string s)

{

    long tot = 0;

    char[] charray;

    charray = s.ToCharArray();
```

```
    for (int i = 0; i <= s.Length - 1; i++)

        tot += 37 * tot + (int)charray[i];

    tot = tot % data.GetUpperBound(0);

    if (tot < 0)

        tot += data.GetUpperBound(0);

    return (int)tot;

}

public void Insert(string item)

{

    int hash_value;

    hash_value = Hash(itemvalue);
```

```
        if (data[hash_value].Contains(item))

            data[hash_value].Add(item);

    }

    public void Remove(string item)

    {

        int hash_value;

        hash_value = Hash(item);

        if (data[hash_value].Contains(item))

            data[hash_value].Remove(item);

    }

}
```

当使用桶式散列法的时候，能做的最重要的事情就是保持所用的 arraylist 元素的数量尽可能地少。在向散列表添加数据项或从散列表移除数据项的时候，这样会最小化所需做的额外工作。在前面的代码中，通过在构造器调用中设置每个 arraylist 的初始容量就可以最小化 arraylist 的大小。一旦有了冲突，arraylist 的容量会变为 *2*，然后每次 arraylist 满时容量就会扩充两倍。虽然用一个好的散列函数，arraylist 也不应该变得太大。

散列表中元素数量与表大小的比率被称为是负载系数。研究表明在负载系数为 *1·0* 的时候，或者在表的大小恰好等于元素数量的时候，散列表的性能最佳。

10·4·2 开放定址法

通过使用 arraylist，分开链接降低了散列表的性能。为避免冲突而分开链接的另外一种选择是开放定址法。开放定址函数会在散列表数组内寻找空单元来放置数据项。如果尝试的第一个单元是满的，那么就尝试下一个空单元，如此反复直到最终找到一个空单元为止。大家在本小节内会看到两种不同的开放定址策略：即线性探查和平方探查。

线性探查法采用线性函数来确定试图插入的数组单元。这就意味着会顺次尝试单元直到找到一个空单元为止。线性探查的问题是数组内相邻单元中的数据元素会趋近成聚类，从而使得后续空单元的探查变得更长久且效率更低。

平方探查法解决了聚类问题。平方函数用来确定要尝试哪个单元。此函数的一个实例如下所示：

2 * collNumber - *1*

这里的 collNumber 是在当前检查过程中已发生冲突的数量。平方探查法的有趣属性是在散列表空余单元少于一半的情况下总能保证找到空的单元。

10·4·3 双重散列法

这种简单的解决冲突的策略完全是说什么就是什么。如果发现冲突，就再次应用散列函数，然后探查距离顺次为 hash（数据项）、*2*hash（数据项）、*4*hash（数据项）如此等等直到找到一个空单元为止。

为了使这种探查技术工作正常，需要满足少量条件。首先，选择的散列函数不应该曾经计算到 *0*，这将导致灾难性的结果（因为用 *0* 相乘，结果为 *0*）。其次，表的大小必须是素数。如果它不是素数，那么就不能探查所有的数组单元，这会再次导致混乱的结果。

双重散列法是一种有趣的冲突解决策略，但是实际上已经说明了平方探查法通常会获得更好的性能。

现在完成了自定义散列表实现的研究。对于大多数采用 C#语言的应用程序而言，最好使用内置的 Hashtable 类，它是·NET 框架库的一部分内容。下一小节就开始这个类的讨论。

10·5 HASHTABLE 类

Hashtable 类是 Dictionary 对象的一种特殊类型，它存储了键值对，其中的数值都是在源于关键字的散列代码的基础上进行存储的。这里可以为关键字的数据类型指定散列函数或者使用内置的函数（稍后将讨论它）。Hashtable 类是非常有效率的，而且应该把它用于任何可能自定义实现的地方。

这个类用来避免冲突的策略就是桶的思想。桶是具有相同散列代码的对象的虚拟组合，这非常像在讨论分开链接的时候用 ArrrayList 来解决冲突的情况。如果两个关键字具有相同的散列代码，那么就把它们放置在同一个桶内。否则，就把每一个具有唯一散列代码的关键字放置在它自己的桶内。

用在一个 Hashtable 对象内的桶的数量被称为是负载系数。负载系数是元素与桶数量之间的比率。此系数初始为 *1·0*。当实际系数达到初始系数的时候，就把负载系数增加成一个最小的素数，这个最小素数是当前桶数量的两倍。负载系数是很重要的，因为负载系数越小，Hashtable 对象的性能就越好。

10·5·1 实例化 Hashtable 对象并且给其添加数据

Hashtable 类是 System·Collections 命名空间的一部分内容，所以必须在程序开始部分导入 System·Collections。

Hashtable 对象可以用三种方法（实际上还有更多种，包括复制构造器的不同类型，但是这里坚持认为有三种最常见的构造器）中的一种进行实例化。这里可以实例化具有初始容量的散列表，或者使用默认容量。当然还可以同时指定初始容量和初始负载系数。下面的代码举例说明如何使用这三种构造器：

```
Hashtable symbols = new Hashtable();

HashtTable symbols = new Hashtable(50);

HashTtable symbols = new Hashtable(25, 3·0F);
```

第一行创建了具有默认容量和默认负载系数的散列表。第二行则创建了具有默认负载系数和 50 个元素容量的散列表。第三行也创建了一个散列表，其容量为 25 个元素且负载系数是 3·0。

利用 Add 方法就可以把键值对添加到散列表内。这个方法会取走两个参数：即关键字和与关键字相关联的数值。在计算完关键字的散列值之后，会把这个关键字添加到散列表内。下面是一段实例代码：

```
Hashtable symbols = new Hashtable(25);

symbols·Add("salary", 100000);

symbols·Add("name", "David Durr");

symbols·Add("age", 43);

symbols·Add("dept", "Information Technology");
```

还可以用索引来给散列表添加元素，这将在本章稍后部分进行更为完整地讨论。为了做到这样，要编写一条赋值语句来把数值赋值给指定的关键字作为索引（这非常像数组的索引）。如果这个关键字已经不存在了，那么就把一个新的散列元素添加到散列表内。如果这个关键字已经存在，那么就用新的数值来覆盖这个存在的数值。下面是一些实例：

```
sSymbols["sex"] = "Male";

sSymbols["age"] = 44;
```

第一行说明如何利用 Item 方法来创建一个新的键值对，而第二行则举例说明可以覆盖与已存在的关键字相关联的当前数值。

10·5·2 从散列表中分别取回关键字和数值

Hashtable 类有两个非常有用的方法用来从散列表中取回关键字和数值：即 Keys 和 Values。这些方法创建了一个 Enumerator 对象，它允许使用 For Each 循环或者其他一些技术来检查关键字和数值。下面的程序举例说明了这些方法是如何工作的：

```
using System;

using System·Collections;

class chapter10

{

    static void Main()

    {

        Hashtable symbols = new Hashtable(25);
```

```
symbols·Add("salary", 100000);

symbols·Add("name", "David Durr");

symbols·Add("age", 45);

symbols·Add("dept", "Information Technology");

symbols["sex"] = "Male";

Console·WriteLine("The keys are: ");

foreach (Object key in symbols·Keys)

    Console·WriteLine(key);

Console·WriteLine();

Console·WriteLine("The values are: ");

foreach (Object value in symbols·Values)
```

```
            Console·WriteLine(value);

    }

}
```

10·5·3 取回基于关键字的数值

使用索引可以完成用相关联的关键字取回数值的操作，这个索引的操作就像数组的索引一样。把关键字作为索引值来传递，而且返回与关键字相关联的数值，除非关键字不存在，否则返回空（null）。

下面代码段举例说明了这种技术的工作过程：

```
Object value = symbols·Item["name"];

Console·WriteLine("The variable name's value is: " + value·ToString());
```

返回的值是“David Durr”。

这里可以使用一个索引连同 Keys 方法来取回所有存储在散列表内的数据：

```
using System;
```

```
using System·Collections;

class chapter10

{

    static void Main()

    {

        Hashtable symbols = new Hashtable(25);

        symbols·Add("salary", 100000);

        symbols·Add("name", "David Durr");

        symbols·Add("age", 45);

        symbols·Add("dept", "Information Technology");

        symbols["sex"] = "Male";
```

```
        Console·WriteLine();

        Console·WriteLine("Hash table dump - ");

        Console·WriteLine();

        foreach (Object key in symbols·Keys)

            Console·WriteLine(key·ToString() + ": " + symbols[key]·ToString());

    }

}
```

输出是：

（原书 P187 页 截图）

10·5·4 Hashtable 类的实用方法

在 Hashtable 类中有几种方法可以使得 Hashtable 对象更加高效。本小节会研究这些方法中的几种，包括确定散列表内元素数量的方法，清除散列表内容的方法，判定散列表内是否包含指定关键字（和数值）的方法，从散列表中移除元素的方法，以及把散列表元素复制到数组的方法。

Count 属性存储着散列表内元素的数量，它会返回一个整数：

```
int numElements;
```

```
numElements = symbols·Count;
```

利用 Clear 方法可以立刻从散列表中移除所有元素：

```
symbols·Clear();
```

为了从散列表中移除单独一个元素，可以使用 Remove 方法。这个方法会取走一个参数，即关键字，而且该方法会把指定关键字和相关联的数值都移除。示例如下所示：

```
symbols·Remove("sex");
```

```
foreach (Object key iln symbols·Keys)
```

```
Console·WriteLine(key·ToString() + ": " + symbols[key]·ToString());
```

在从散列表中移除一个元素之前，可能希望查看该元素或者数值是否在散列表内。用 ContainsKey 方法和 ContainsValue 方法就可以确定这个信息。下面的代码段举例说明了如

果使用 ContainsKey 方法：

```
string aKey;

Console·Write("Enter a key to remove: ");

aKey = Console·ReadLine();

if (symbols·ContainsKey(aKey))

symbols·Remove(aKey);
```

用这个方法来确保要移除的键值对是在散列表内的。ContainsValue 方法的操作类似，只是用数值代替了关键字。

10·6 HASHTABLE 的应用程序：计算机术语表

散列表的常见应用之一就是构造术语表或术语词典。本小节会举例说明使用散列表的一种方法就是为了这样一个应用—即计算机术语表。

程序首先从一个文本文件中读入一系列术语和定义。这个过程是在子程序 BuildGlossary 中编码实现的。文本文件的结构是：单词，定义，用逗号在单词及其定义之间进行分隔。这个术语表中的每一个单词都是单独一个词，但是术语表也可以很容易地替换处理短语。这就是用逗号而不用空格作分隔符的原因。此外，这种结构允许使用单词作为关键字，这是构造这个散列表的正确方法。

另一个子程序 DisplayWords 把单词显示在一个列表框内，所以用户可以选取一个单词来获得它的定义。既然单词就是关键字，所以能使用 Keys 方法从散列表中正好返回单词。然后，用户就可以看到有定义的单词了。

用户可以简单地点击列表框中的单词来获取其定义。用 Item 方法就可以取回定义，并且把它显示在文本框内。

代码如下所示：

```
using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Text;

using System.Windows.Forms;

using System.Collections;
```

```
using System.IO;



namespace WindowsApplication3

{

    public partial class Form1 : Form

    {

        private Hashtable glossary = new Hashtable();

        public Form1()

        {

            InitializeComponent();

        }
```

```
private void Form1_Load(object sender, EventArgs e)

{

    BuildGlossary(glossary);

    DisplayWords(glossary);

}

private void BuildGlossary(Hashtable g)

{

    StreamReader inFile;

    string line;

    string[] words;

    inFile = File·OpenText(@"c:\words·txt");
```

```
        char[] delimiter = new char[] { ',' };

        while (inFile·Peek() != -1)

        {

            line = inFile·ReadLine();

            words = line·Split(delimiter);

            g·Add(words[0], words[1]);

        }

        inFile·Close();

    }

    private void DisplayWords(Hashtable g)

    {
```

```
        Object[] words = new Object[100];

        g·Keys·CopyTo(words, 0);

        for (int i = 0; i <= words·GetUpperBound(0); i++)

            if (!(words[i] == null))

                lstWords·Items·Add((words[i]));

}

private void lstWords_SelectedIndexChanged(object sender, EventArgs e)

{

        Object word;

        word = lstWords·SelectedItem;

        txtDefinition·Text = glossary[word]·ToString();
```

```
        }

    }

}
```

文本文件内容如下：

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

. . .

下面就是这个程序运行时的样子：

（原书P*192*页 截图）

如果录入的单词不在词汇表内，那么Item方法返回为空值（Nothing）。在子程序GetDefinition中有对空值的检测，以便当录入的单词不在散列表内时显示字符串“not found”。

小结

散列表对于存储键值对是一种非常有效的数据结构。散列表的实现通常是非常简单的，而要慎重对待的地方就是需要选择处理冲突的策略。本章讨论了几种处理冲突的技术。

对于大多数C#语言的应用程序而言，在·NET框架库的Hashtable类工作十分出色的情况下，没有必要去构造自定义的散列表。大家可以为类指定属于自己的散列函数，或者可以让类来计算散列数值。

练习

26· 请用本章开发的自定义的 Hash 类来重新编写有关计算机术语表的应用程序。请用不同的散列函数和冲突解决方法来进行实验。

27· 请利用 Hashtable 类来编写一个拼写检查程序。它从文本文件中读取数据，并且检查拼写错误。当然，大家会需要把词典限制在几个常用单词内。

28· 请创建一个新的 Hash 类。针对散列表，此类用 arraylist 来代替数组。通过重新编写计算机术语表的应用程序来测试自行编写的实现。

第 *11* 章 链表

对于许多应用程序而言，最好把数据存储成列表的形式，而且列表在日常生活中是很常见的：代办事件列表、购物清单、前十名名单等等。本章会研究一种特殊类型的列表，即链表。尽管·NET 框架类库包含了几种基于列表的集合类，但是链表并不在其中。本章会从解释说明为什么需要链表开始，然后讨论这种数据结构的两种不同实现，即基于对象的链表和基于数组的链表。最后本章会用几个实例作为结束，这些例子都是关于如何用链表来解决可能会遇到的计算机编程问题。

11·1 数组存在的问题

在处理列表的时候数组是常用的数据结构。数组可以对所存储的数据项提供快速地存取访问，而且它很易于进行循环遍历操作。当然，数组已经是语言的一部分了，用户不需要使用额外的内存，也不需要花费因使用用户自定义的数据结构所需的处理时间。

然而正如所见，数组不是一种最佳的数据结构。在无序数组中查找一个数据项是很慢的，这是因为在找到要查找的元素之前需要尽可能地访问到数组内的每一个元素。有序（排序）数

组对查找而言会更加高效一些，但是插入和删除操作还是很慢的，因为需要向前或向后移动元素来为插入留出空间，或者为删除移除空间。更别提在有序数组内还需要为插入元素查找到合适的位置了。

当发现在数组上执行的实际操作速度太慢的时候，大家就会考虑用链表来代替。链表可以用于几乎每一种使用数组的情况中，除非需要随机存取访问列表内的数据项，这时数组或许会是最好的选择。

11·2 链表的定义

链表是被称为节点的类对象的群集。每一个节点通过一个引用链接到列表内的后继节点上。节点包括存储数据的字段和节点引用的字段。到另外一个节点的引用被称为是链接。图 *11-1* 展示了一个链表的实例。

（原书 P*195* 页 图 *1*）

图 *11-1* 链表的实例

数组和链表之间的一个主要区别就是数组内的元素是通过位置（索引）进行引用的，而链表内的元素则是通过它们与数组其他元素的关系进行引用的。在图 *11-1* 中，大家会说“Bread”跟在“Milk”的后面，而不会说“Bread”是在第二个位置上。遍历链表是从链表的起始节点一直到末尾节点。

在图 *11-1* 中还需要注意的一点就是对链表结尾的标记是通过指向空（null）值实现的。既然是在内存中处理类对象，所以就用空（null）对象来表示列表的末尾。

在某些情况下对列表起始处做标记可能是一个问题。在许多链表的实现中通常会包含一个被称为“头节点”的特殊节点来作为链表的起始位置。图 *11-2* 就是由图 *11-1* 增加了头节点后改进而来的。

（原书 P*195* 页 图 *2*）

图 *11-2* 带头节点的链表

在使用链表的时候，插入操作成为一种非常有效的工作。所要做的就是把要插入节点之前节点的链接改为指向要插入的节点，并且把新节点的链接设为指向插入之前前节点所指向的节点。在图 *11-3* 中，把数据项“Cookies”添加到链表内“Eggs”的后面。

（原书 P*196* 页 图 *1*）

图 *11-3* 插入 Cookies

从链表中移除数据项也是如此容易。就是简单地把要删除节点之前节点的链接重定向到删除节点所指向的节点，并且把删除节点的链接设为空（null）就可以了。图 *11-4* 所描述的操作就是把“Bacon”从链表中移除。

（原书 P*196* 页 图 *2*）

图 *11-4* 移除 Bacon

还有其他方法也可以在 LinkedList 类中实现，但是插入和删除这两种方法正是使用链表超过数组的原因所在。

11·3 面向对象链表的设计

链表的设计至少包含两个类。这里会创建一个 Node 类，而且每次向链表添加节点的时候会实例化一个 Node 对象。链表内的节点通过索引与其他节点相互连接在一起。而且把这些索引设置为使用创建在一个独立的 LinkedList 类中的方法。首先就从了解 Node 类的设计开始吧。

11·3·1 Node 类

节点是由两个数据成员组成的：存储着节点数据的 Element，以及存储着指向表内下一节点引用的 Link。这里会使用 Object 作为 Element 的数据类型，所以也就不用担心存储在表内的数据的类型了。Link 的数据类型是 Node，这看似很奇怪，但实际上是很容易理解的。因为要把链接指向下一个节点，而且是用索引来作为这个链接，所以需要把成员 Link 设置为 Node 类型。

为了完成 Node 类的定义，至少需要两种构造器方法。明确地需要一个默认的构造器来创建一个空的 Node，其中的 Element 和 Link 都设为空（null）。还需要一个参数化的构造器用来给成员 Element 赋值数据，并且把成员 Link 设置为空（null）。

Node 类的代码如下所示：

```
public class  Node

{

    public Object  Element;

    public Node  Link;

    public Node()

    {
```

```
        Element = null;

        Link = null;

    }

    public Node(Object theElement)

    {

        Element = theElement;

        Link = null;

    }

}
```

11·3·2 LinkedList 类

LinkedList 类用来创建链表中节点之间的链接。这个类包括几种方法，有把节点添加到链表的方法，有从链表中移除节点的方法，有遍历链表的方法，还有找到链表内节点的方法。此外，还需要一种构造器方法来实例化链表。此类中唯一的数据成员就是头节点。

```
public class LinkedList

{

    protected Node header;

    public LinkedList()

    {

        header = new Node("header");

    }

...

}
```

头节点从其 Link 字段设置为空（null）开始。当把第一个节点添加到链表中的时候，会把头节点的 Link 字段设置成指向新的节点，并且把新节点的 Link 字段设置为空（null）。

第一种要讨论的方法就是 Insert 方法，用它来把节点放入链表内。为了向链表插入一个节点，需要指定希望插入在节点之前还是之后。为了调整链表内所有必要的链接，这样做是必需的。这里选择把新节点插入到表内已有节点的后边。

为了在已有节点的后边插入新节点，需要首先找到这个“之后”的节点。为了做到这一点，

这里创建了一个 Private 方法，即 Find 方法，用它来搜索每个节点的 Element 字段，直到找到匹配为止。

```
private Node Find(Object item)

{

    Node current = new Node();

    current = header;

    while (current· Element != item)

        current = current·Link;

    return current;

}
```

这个方法说明了如何在链表中遍历。首先，实例化一个 Node 对象 current，并且把它设置为头节点。然后查看节点内 Element 字段的数值是否等于要查找的数值。如果不是，就移动到下一个节点去，其方法是把此节点赋值到 current 的 Link 字段内作为 current 的新数值。

一旦找到这个“之后”的节点，下一步就是把新节点的 Link 字段设置为“之后”节点的 Link 字段，然后把“之后”节点的 Link 字段设置为指向新节点。下面就是实际操作过程：

```
public void Insert(Object newItem, Object after)

{

    Node current = new Node();

     Node newNode = new Node(newItem);

     current = Find(after);

     newNode.Link = current.Link;

     current.Link = newNode;

}
```

下一个要研究的链表操作是 Remove。从链表中移除一个节点，就需要简单地改变节点的链接从而使得指向要移除节点的链接改为指向移除节点后面的节点。

既然需要找到要删除节点之前的节点，所以要定义一个方法，即 FindPrevious，用它来做实现这个操作。这个方法会向后遍历链表，然后停在每个节点处来查看下一个节点的 Element 字段是否存有要移除的数据项。

```
private Node FindPrevious(Object n)

{

    Node current = header;

    while (!(current·Link == null) && (current·Link·Element != n))

    current = current·Link;

    return current;

}
```

下面就准备来看看 Remove 方法的实现代码：

```
public void Remove(Object n)

{

Node p = FindPrevious(n);

    if (!(p·Link == null))
```

```
        p·Link = p·Link·Link;

}
```

Remove 方法只移除链表内数据项的第一次出现。大家可能也注意到了，如果数据项不在链表内，那么什么也不会发生。

本节要介绍的最后一种方法是 PrintList，它会遍历链表并且显示出链表内每一个节点的 Element 字段。

```
public void PrintList()

{

    Node current = new Node();

    current = header;

    while (!(current·Link == null))

    {

        Console·WriteLine(current·Link·Element);
```

```
            current = current.Link;

        }

}
```

11.4 链表设计的改进方案

为了更好地解决某些问题，这里还有几种对链表设计的改进方案。最常见的两种的改进方案是双向链表和循环链表。双向链表会使反向遍历链表以及从链表内移除节点都变得更加容易。而循环链表则会使得链表内进行多次移动操作变得更加便利。本节会介绍这两种改进方案。最后，还会看到对 LinkedList 类的改进，这种改进只常见于链表面向对象的实现，它是一个用来说明表内位置的 Iterator 类。

11.4.1 双向链表

虽然从表内第一个节点到最后一个节点的遍历操作是非常简单的，但是反向遍历链表却不是一件容易的事情。如果为 Node 类添加一个字段来存储指向前一个节点的链接，那么就会使得这个反向操作过程变得容易许多。当向链表插入节点的时候，为了把数据赋值给新的字段还会需要执行更多的操作，但是当要把节点从表中移除的时候就会获得收效了。图 11-5 就图形化地说明了双向链表的工作原理。

（原书 P200 页 图）

图 11-5 双向链表

首先需要修改 Node 类来为类增加一个额外的链接。为了区别两个链接，这里把指向下一个节点的链接称为 FLink，而把指向前一个节点的链接称为 Blink。在实例化一个 Node 的时候，会把这些字段都设置为空（Nothing）。代码如下所示：

```
public class Node

{

    public Object Element;

    public Node Flink;

    public Node Blink;

    public Node()

    {

        Element = null;
```

```
        Flink = null;

        Blink = null;

    }

    public Node(Object theElement)

    {

        Element = theElement;

        Flink = null;

        Blink = null;

}

}
```

Insertion 方法类似于单向链表中的同类方法，只是需要把新节点的向后链接设为指向前一个节点。

```
public void Insert(Object newItem, Object after)

{

Node current = new Node();

Node newNode = new Node(newItem);

current = Find(after);

newNode·Flink = current·FlLink;

newNode·Blink = current;

current·Flink = newNode;

}
```

双向链表的 Remove 方法比单向链表中此类方法容易编写许多。首先需要找到表内要删除的节点，然后把此节点的向后链接属性设为指向要删除节点向前链接所指向的节点。然后需要对删除节点所指向链接的后链接进行重定向操作，把它指向删除节点之前的节点。

图 *11-6* 说明了从双向链表中删除节点的一种特殊情况，即要删除的节点恰好是表内最后一

个节点的情况（不是 Nothing 节点）。

（原书 P201 页 图）

图 11-6 从双向链表中移除一个节点

双向链表的 Remove 方法如下所示：

```
public void Remove(Object n)

{

Node p = Find(n);

if (!(p·Flink == null))

{

    p·Blink·Flink = p·Flink;
```

```
    p·Flink·Blink = p·Blink;

    p·Flink = null;

    p·Blink = null;

    }

}
```

本小节会以编写一种反向显示双向链表元素的实现作为结束。在单向链表中，实现这个方法稍微有些困难，但是对于双向链表而言就很容易编写这种方法了。

首先，需要一个方法来找到链表内的最后一个节点。这就是沿着每个节点的向前链接顺次寻找的过程，直到到达指向为空（null）的链接才结束。这个方法被命名为 FindLast，其定义如下所示：

```
private Node FindLast()

{

    Node current = new Node();

current = header;
```

```
while (!(current·Flink == null))

current = current·Flink;

return current;

}
```

一旦找到链表内的最后一个节点，就反向显示出链表，所谓反向就是沿着向后链接一直到达指向为空（null）的链接为止，这个链接说明处在头节点的位置上。代码如下所示：

```
public void PrintReverse()

{

Node current = new Node();

current = FindLast();

while (!(current·Blink == null))

{
```

```
    Console·WriteLine(current·Element);

    current = current·Blink;

}

}
```

11·4·2 循环链表

循环链表是一种尾节点返回指向首节点（它可能是头节点）的链表。图 11-7 说明了循环链表的工作原理。

（原书 P203 页 图）

图 11-7 循环链表

这种类型的链表会用在一些要求把尾节点返回指向首节点（或头节点）的应用程序中。当调

用链表的时候，许多程序员会选择使用循环链表。

在实例化一个新链表的时候，唯一需要真的改变的就是使头节点指向它自身。如果这样做了，那么每次添加一个新节点，尾节点都会指向头节点，因为链接是从一个节点传播到另一个节点的。

循环链表的代码如下所示。为了说明清楚，这里显示了完整的类（而不是为了增加页码长度）：

```
public class Node

{

    public Object Element;

    public Node Link;

    public Node()

    {

        Element = null;

        Link = null;

    }
```

```
    public Node(Object theElement)

    {

        Element = theElement;

        Link = null;

    }

}

public class LinkedList

{

    protected Node current;

    protected Node header;

    private int count;
```

```
public LinkedList()

{

    count = 0;

    header = new Node("header");

    header·Link = header;

}

public bool IsEmpty()

{

    return (header·Link == null);

}

public void MakeEmpty()
```

```
{

    header·Link = null;

}

public void PrintList()

{

    Node current = new Node();

    current = header;

    while (current·Link·Element·ToString() != "header")

    {

        Console·WriteLine(current·Link·Element);

        current = current·Link;
```

```
    }

}

private Node FindPrevious(Object n)

{

    Node current = header;

    while (!(current.Link == null) && current.Link.

    Element != n)

        current = current.Link;

    return current;

}

private Node Find(Object n)
```

```
{

    Node current = new Node();

    current = header·Link;

    while (current·Element != n)

        current = current·Link;

    return current;

}

public void Remove(Object n)

{

    Node p = FindPrevious(n);

    if (!(p·Link == null))
```

```
            p·Link = p·Link·Link;

        count--;

}

public void Insert(Object n1, Object n2)

{

        Node current = new Node();

        Node newnode = new Node(n1);

        current = Find(n2);

        newnode·Link = current·Link;

        current·Link = newnode;

        count++;
```

```
}

public void InsertFirst(Object n)

{

    Node current = new Node(n);

    current·Link = header;

    header·Link = current;

    count++;

}

public Node Move(int n)

{

    Node current = header·Link;
```

```
        Node temp;

        for (int i = 0; i <= n; i++)

            current = current.Link;

        if (current.Element.ToString() == "header")

            current = current.Link;

        temp = current;

        return temp;

}

public Node getFirst()

{

        return header;
```

```
}

}
```

在·NET 框架库中，数据结构 ArrayList 的实现就是使用了循环链表。还有许多问题也可以用循环链表来解决。大家在练习中会看到一个典型问题。

11·5 使用 ITERATOR 类

LinkedList 类存在的一个问题就是不能在链表内同时引用两个位置。大家可以引用链表内的任何一个位置（当前节点、前一个节点等等），但是如果想指定两个甚至更多个位置，比如想从链表中移除一段范围内的节点，就需要一些其他方法了。这种方法就是 Iterator 类。

Iterator 类由三个数据字段组成：一个存储链表的字段，一个存储当前节点的字段，还有一个存储前一个节点的字段。构造器方法传递链表对象，而且这个方法会把当前字段设置为链表的头节点传递到方法中。一起来看一下到目前为止这个类的定义：

```
public class ListIter

{

    private Node current;

    private Node previous;
```

```
    LinkedList theList;

    public ListIter(LinkedList list)

    {

        theList = list;

        current = theList·getFirst();

        previous = null;

}

}
```

希望 Iterator 类做的第一件事就是允许在链表中从一个节点移动到下一个节点。方法 nextLink 完成了这项工作：

```
public void NextLink()

{
```

```
        previous = current;

        current = current·Link;

}
```

注意除非在建立新的当前位置，否则在此方法完成执行之前会把前节点也设置为当前节点。除了当前节点还要跟踪前节点，这样才会使得插入和删除操作都变得更加容易。

getCurrent 方法会返回由迭代器指向的节点：

```
public Node GetCurrent()

{

return current;

}
```

在 Iterator 类中内置了两种插入方法：InsertBefore 方法和 InsertAfter 方法。InsertBefore 方法会把新节点插入到当前节点之前的位置上，而 InsertAfter 方法则把新节点插入到当前节点之后的位置上。首先来了解一下 InsertBefore 方法。

在当前对象之前插入一个新节点的时候，需要做的第一件事就是查看是否在链表的起始位置上。如果在，那么不能在头节点之前插入节点，所以会发出一个异常。在下面对这个异常进行了定义。否则，就把新节点的 Link 字段设置为前节点的 Link 字段，而把前节点的 Link 字段设为新节点，并且重新设置当前位置为新节点。代码如下所示：

```
public void InsertBefore(Object theElement)

{

Node newNode = new Node(theElement);

if (current == header)

throw new InsertBeforeHeaderException();

else

{

newNode·Link = previous·Link;

previous·Link = newNode;

current = newNode;

}
```

```
}
```

InsertBeforeHeaderException 类的定义如下所示：

```
class InsertBeforeHeaderException : Exception

{

    public InsertBeforeHeaderException()

        : base("Can't insert before the header node·")

    {

    }

}
```

Iterator 类中的 InsertAfter 方法比 LinkedList 类中编写的方法简单许多。既然已经知道了当前节点的位置，这个方法就只需要设置正确的链接并且把当前节点设置为下一个节点。

```
public void InsertAfter(Object theElement)

{
```

```
    Node newnode = new Node(theElement);

    newnode·Link = current·Link;

    current·Link = newnode;

    NextLink();

}
```

使用 Iterator 类从链表中移除节点是非常容易的。这个方法会简单地把前节点的 Link 字段设置为由当前节点的 Link 字段所指向的节点：

```
public void Remove()

{

    previous·Link = current·Link;

}
```

Iterator 类所需的其他方法包括有把迭代器重新设置为头节点的方法（并且把前节点重新设置为空），以及检测是否在表尾的方法。这些方法如下所示：

```
public void Reset()

{

    current = theList.getFirst();

    previous = null;

}

public bool AtEnd()

{

    return (current.Link == null);

}
```

11.5.1 新的 LinkedList 类

现在用 Iterator 类做了大量的工作，可以把 LinkedList 类稍微消减一些。当然，这里始终需要一个头节点字段和一个实例化链表的构造器方法。

```
public class LinkedList

{

    private Node header;

    public LinkedList()

    {

        header = new Node("header");

    }

    public bool IsEmpty()

    {

        return (header·Link == null);
```

```
}

public Node GetFirst()

{

    return header;

}

public void ShowList()

{

    Node current = header·Link;

    while (!(current == null))

    {

        Console·WriteLine(current·Element);
```

```
            current = current.Link;

        }

    }

}
```

11.5.2 实例化 Iterator 类

利用 Iterator 类可以很容易地编写出一个交互式程序来遍历链表。这也为大家提供了一个机会来把 Iterator 类和 LinkedList 类的代码放在一起。

```
using System;

public class Node

{

    public Object Element;

    public Node Link;

    public Node()
```

```
    {

        Element = null;

        Link = null;

    }

    public Node(Object theElement)

    {

        Element = theElement;

        Link = null;

    }

}

public class InsertBeforeHeaderException : System·ApplicationException
```

```
{

    public InsertBeforeHeaderException(string message) :base(message)

    {

    }

}

public class LinkedList

{

    private Node header;

    public LinkedList()

    {

        header = new Node("header");
```

```
}

public bool IsEmpty()

{

    return (header·Link == null);

}

public Node GetFirst()

{

    return header;

}

public void ShowList()

{
```

```
        Node current = header·Link;

        while ((current == null))

        {

            Console·WriteLine(current·Element);

            current = current·Link;

        }

    }

}

public class ListIter

{

    private Node current;

    private Node previous;
```

```
LinkedList theList;

public ListIter(LinkedList list)

{

    theList = list;

    current = theList·GetFirst();

    previous = null;

}

public void NextLink()

{

    previous = current;

    current = current·Link;
```

```
}

public Node GetCurrent()

{

    return current;

}

public void InsertBefore(Object theElement)

{

    Node newNode = new Node(theElement);

    if (previous.Link == null)

        throw new InsertBeforeHeaderException("Can't insert here.");

    else
```

```
    {

        newNode·Link = previous·Link;

        previous·Link = newNode;

        current = newNode;

    }

}

public void InsertAfter(Object theElement)

{

    Node newNode = new Node(theElement);

    newNode·Link = current·Link;

    current·Link = newNode;
```

```
        NextLink();

    }

    public void Remove()

    {

        previous·Link = current·Link;

    }

    public void Reset()

    {

        current = theList·GetFirst();

        previous = null;

    }
```

```
    public bool AtEnd()

    {

        return (current.Link == null);

    }

}

class chapter77

{

    static void Main()

    {

        LinkedList MyList = new LinkedList();

        ListIter iter = new ListIter(MyList);
```

```
string choice, value;

try

{

    iter·InsertAfter("David");

    iter·InsertAfter("Mike");

    iter·InsertAfter("Raymond");

    iter·InsertAfter("Bernica");

    iter·InsertAfter("Jennifer");

    iter·InsertBefore("Donnie");

    iter·InsertAfter("Michael");

    iter·InsertBefore("Terrill");

    iter·InsertBefore("Mayo");
```

```
iter·InsertBefore("Clayton");

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();

char[] onechar = choice·ToCharArray();

switch (onechar[0])

{

    case 'n':

        if (!(MyList·IsEmpty()) && (!(iter·AtEnd())))

            iter·NextLink();

        else
```

```
                    Console·WriteLine(“Can‘ move to next link·“);

                    break;

                case ‘g‘:

                    if (!(MyList·IsEmpty()))

                        Console·WriteLine(“Element:          “          +
iter·GetCurrent()·Element);

                    else

                        Console·WriteLine(“List is empty·“);

                    break;

                case ‘r‘:

                    iter·Reset();

                    break;
```

```
case 's':

    if (!(MyList.IsEmpty()))

        MyList.ShowList();

    else

        Console.WriteLine("List is empty.");

    break;

case 'a':

    Console.WriteLine();

    Console.Write("Enter value to insert:");

    value = Console.ReadLine();

    iter.InsertAfter(value);

    break;
```

```
case 'b':

    Console·WriteLine();

    Console·Write("Enter value to insert:");

    value = Console·ReadLine();

    iter·InsertBefore(value);

    break;

case 'c':

    // clear the screen

    break;

case 'x':

    // end of program
```

```
                    return;

                }

            }

        }

        catch (InsertBeforeHeaderException e)

        {

            Console.WriteLine(e.Message);

        }

    }

}
```

如上所示，这个程序是一个命令行应用程序，而没有使用图形化用户界面。然而，大家在练习中会有机会来弥补这个缺憾。

11·6 通用的 LINKED LIST 类和通用的 NODE 类

System·Collections·Generic 命名空间为构建链表提供了两种通用的类：LinkedList 类和 LinkedListNode 类。Node 类为存储数值和链接提供了两个数据字段，而 LinkedList 类则用在节点前以及在节点后的插入方法实现了双向链表。这个类还提供了其他方法，包括移除节点的方法，找到链表内首节点和尾节点的方法，当然还有其他有用的方法。

11·6·1 通用链表实例

像其他通用类一样，LinkedListNode 和 LinkedList 在实例化对象时要求一个数据类型占位符。下面是一些实例：

```
LinkedListNode<string> node1 = new LinkedListNode<string>("Raymond");

LinkedList<string> names = new LinkedList<string>();
```

在这里，它只是使用类来构造和使用链表的问题。一个简单的实例证明了使用这些类是多么地容易：

```
using System;

using System·Collections·Generic;
```

```
using System.Text;

class Program

{

    static void Main(string[] args)

    {

        LinkedListNode<string> node = new

        LinkedListNode<string>("Mike");

        LinkedList<string> names = new LinkedList<string>();

        names.AddFirst(node);

        LinkedListNode<string> node1 = new LinkedListNode<string>("David");

        names.AddAfter(node, node1);
```

```
LinkedListNode<string> node2 = new LinkedListNode<string>("Raymond");

names·AddAfter(node1, node2);

LinkedListNode<string> node3 = new LinkedListNode<string>(null);

LinkedListNode<string> aNode = names·First;

while (aNode != null)

{

    Console·WriteLine(aNode·Value);

    aNode = aNode·Next;

}

aNode = names·Find("David");

if (aNode != null) aNode = names·First;
```

```
            while (aNode != null)

            {

                Console·WriteLine(aNode·Value);

                aNode = aNode·Next;

            }

            Console·Read();

        }

}
```

在这个实例中的链表没有使用头节点，因为人们通过 First 属性可以很容易地找到链表中的首节点。虽然在这个实例中没有使用 Last 属性，但是这个属性也可以用在前面的 While 循环中来判定链表的末尾：

```
while (aNode != names·Last)

{
```

```
Console·WriteLine(aNode·Value);

aNode = aNode·Next;

}
```

还有另外两种方法没有显示在这里，可以证明它们在链表实现中是很有用的：AddFirst 方法和 AddLast 方法。这两种方法可以在不需提供链表的头节点和尾节点的情况下用来实现链表。

小结

在传统的计算机编程学习中，链表经常是首先学习的数据结构。然而，在 C#语言中可能会使用内置数据结构中的一种，比如 ArrayList，并且会获得和链表实现相同的结果。但是，每一位编程的学习者都值得花时间来学习链表的工作原理以及实现它们的过程。·NET 框架库用循环链表的设计来实现 ArrayList 数据结构。

C#语言 2·0 版提供了通用的 linked list 类和通用的 Node 类。这两种通用类使编写链表来适应表内节点所对应的不同数据类型值变得更加容易。

虽然没有书籍用 C#语言作为目标语言，但是仍有许多讨论链表的很好的书籍。通用的权威性书籍是 Knuth 的《计算机编程的艺术 第一卷 基础算法》一书。其他可以参考的书籍包括 Ford 和 Topp 编写的《数据结构 C++描述》一书。如果对 Java 语言实现有兴趣的话（而且应该会有兴趣的，因为几乎可以直接把 Java 语言的实现转化为 C#语言的实现）还可以参考 Mark Allen Weiss 编写的《数据结构与算法实现：Java 语言描述》一书。

练习

29· 请重新编写使用了基于迭代器的链表的命令行应用程序，这次用视窗应用程序来实现。

30· 根据传说，第一世纪的犹太历史学家 Flavius Josephus 在犹太人与罗马人的战争中和 40 名同胞一起被罗马士兵抓获。这些被俘的士兵宁愿自杀也不愿当俘虏，而且他们还设计一种方案来依次自杀。他们站成一个圈，然后每隔三名士兵就有一位自杀，直到所有人都死掉为止。Joseph 和另外一个人不想这样死去，他们快速地计算出自己在圈中所站的位置以便他俩都可以幸存下来。请编写一个程序允许由 n 个人围成一个圈，而且指定每隔 m 个人就会杀死一位。这个程序应该确定出留在圈中最后一人的编号。请用循环链表来解决这个问题。

31· 请编写一个程序可以读取 VB·NET 代码的不确定行数，并且把保留字存储在一个链表内，而把标识符和文字串存储在另一个链表里。当程序完成读取输入的时候，显示出每个链表的内容。

32· 请为 LinkedList 类设计并实现 ToArray 方法。此方法会取走一个链表实例并且返回一个数组。

第 12 章 二叉树和二叉查找树

在计算机科学中树是一种很常见的数据结构。树是一种非线性的数据结构，人们可以用它来把数据按照等级模式存储起来。本章会研究一种主要的树结构，即二叉树，并且连同讨论二叉树的一种实现，即二叉查找树。人们时常选择二叉树超过诸如数组和链表这样的较为基础的数据结构，因为人们可以快速地查找二叉树（相对于链表而言），还可以快速地在二叉树中插入和删除数据（相对于数组而言）。

12.1 树的定义

在研究二叉树的结构和行为之前，需要定义人们所理解的树的含义。树是由边连接的一系列节点。树的一个实例就是公司的组织机构图（参见图 12-1）。

（原书 P219 页 图 1）

①、 首席执行官

②、 财务副总裁

③、 首席信息官

④、 销售副总裁

⑤、 开发经理

⑥、 业务经理

⑦、 技术支持

图 12-1 局部组织机构图

组织结构图的目的是为了向观看者展示组织的结构。在图 *12-1* 中，每个方框就是一个节点，而连接方框的线就是边。很显然，节点表示的实体（人）构成了一个组织。而边则表示了实体之间的关系。例如，首席信息官（CIO）直接向首席执行官（CEO）汇报工作，所以在这两个节点之间有一条边。IT 经理向 CIO 汇报工作，所以在他们之间有一条边来连接。销售副总裁和 IT 中的开发经理之间没有直接用边进行连接，因此这两个实体之间没有直接的关系。

图 *12-2* 展示了另一棵树，这棵树定义了一些在讨论树时所需的术语。树上最高的节点被称为是根节点。如果一个节点下面连接着其他节点，那么上层的节点被称为父节点，而下层节点则被称为是父节点的子节点。一个节点可以有零个、一个或多个节点与它相连。被称为二叉树的树的特殊类型则限制子节点的数量不能超过两个。二叉树具有某些计算的属性，这些属性可以使二叉树在许多操作方面非常有效率。在本章的小节内会对二叉树进行广泛地讨论。没有任何子节点的节点被称为叶子。

（原书 P*219* 页 图 *2*）

①、 第 *0* 层

②、 根

③、 *13* 和 *54* 的父节点

④、 从 *23* 到 *46* 的路径

⑤、 第 1 层

⑥、 23 的左子节点

⑦、 23 的右子节点

⑧、 第 2 层

⑨、 叶子

⑩、 第 3 层

⑪、 关键字数值

⑫、 子树

图 12-2 部分树

继续来研究图 12-2，大家会看到沿着某些边可以从一个节点访问到其他不直接相连的节点。

从一个节点到另外一个节点所沿着走的一系列边被称为是路径（图中表示为虚线）。按照一些特殊顺序访问树中所有节点被称为是树的遍历。

人们可以把树分层。根节点在第 0 层，它的子节点在第 1 层，这些子节点的子节点在第 2

层，如此等等。在任何层上的节点被看作是子树的根节点，子树由这个根节点的子节点以及子节点的子节点们如此等等组成。人们可以定义树的深度作为树中的层数。

最后，树内的每一个节点都有一个数值。这个数值有时被称为关键字数值。

12·2 二叉树

人们把每个节点最多拥有不超过两个子节点的树定义为二叉树。由于限制子节点的数量为 *2*，人们可以为插入数据、删除数据、以及在二叉树中查找数据编写有效的程序了。

在讨论用 C#语言构造二叉树之前，需要为树的词典添加两个术语。一个父节点的两个子节点分别被称为是左节点和右节点。对于某些二叉树的实现而言，一些数据值只能存储在左节点内，而其他数据值则必须存储在右节点内。图 *12-3* 显示了一个二叉树的实例。

（原书 P*221* 页 图）

图 *12-3* 二叉树

在考虑一种更加特殊的二叉树——二叉查找树的时候，鉴别子节点是很重要的。二叉查找树是一种较小数据值存储在左节点内而较大数据值存储在右节点内的二叉树。正如即将看到的那样，这种属性可以使查找非常有效。

12·2·1 构造二叉查找树

二叉查找树由节点组成，所以需要一个 Node 类，这个类类似于链表实现中用到的 Node 类。
首先一起来看看 Node 类的代码：

```
public class Node

{

    public int Data;

    public Node left;

    public Node right;

    public void DisplayNode()

    {

        Console·Write(iData);
```

```
    }

}
```

这里为存储在节点内的数据和每一个子节点包含了 Pubilc 数据成员。displayNode 方法允许显示存储在节点内的数据。这种特殊的 Node 类存储整数，但是人们可以很容易地把它调整为存储各种类型的数据，或者若需要，甚至可以声明为 Object 类型的 iData。

下面就准备来构造 BinarySearchTree（BST）类。这个类只由一个数据成员构成，即一个表示 BST 根节点的 Node 对象。针对此类的默认构造器方法把根节点设置为空（null），同时创建一个空节点。

接下来需要 Insert 方法来向树内添加新的节点。这个方法有些复杂，而且会需要一些解释说明。这个方法中的第一步是创建一个 Node 对象，并且把 Node 存储的数据赋值给 iData 变量。这个数值会作为唯一的参数传递到此方法内。

插入的第二步是查看 BST 是否有根节点。如果没有，那么说明这是一个新的 BST，并且要插入的节点就是根节点。如果是这种情况，那么就结束这个方法。否则，这个方法进入到下一步。

如果要添加的节点不是根节点，那么为了找到合适的插入点需要遍历 BST。这个过程类似于链表的遍历。当逐层移动的时候，需要一个 Node 对象能赋值为当前节点。还需要把自身定位在 BST 内的根节点上。

一旦在 BST 内部，下一步就是确定放置新节点的位置。这个操作在一个 while 循环内执行，一旦为新节点找到正确的位置就跳出循环。确定节点正确位置的算法如下所示：

1. 把父节点设置为当前节点，即根节点。

2. 如果新节点内的数据值小于当前节点内的数据值，那么把当前节点设置为当前

节点的左子节点。如果新节点内的数据值大于当前节点内的数据值，那么就跳到步骤 4。

3. 如果当前节点的左子节点的数值为空（null），就把新节点插入在这里并且退出循环。否则，跳到 while 循环的下一次循环操作中。

4. 把当前节点设置为当前节点的右子节点。

5. 如果当前节点的右子节点的数值为空（null），就把新节点插入在这里并且退出循环。否则，跳到 while 循环的下一次循环操作中。

下面是 Insert 方法的代码，还有 BST 类（已经讨论过的）剩余部分的代码以及 Node 类的代码：

```
public class Node

{

    public int Data;

    public Node Left;

    public Node Right;

    public void DisplayNode()
```

```
    {

        Console·Write(Data + " ");

    }

}

public class BinarySearchTree

{

    public Node root;

    public BinarySearchTree()

    {

        root = null;

    }

    public void Insert(int i)
```

```
{

    Node newNode = new Node();

    newNode.Data = i;

    if (root == null)

        root = newNode;

    else

    {

        Node current = root;

        Node parent;

        while (true)

        {
```

```
parent = current;

if (i < current·Data)

{

    current = current·Left;

    if (current == null)

    {

        parent·Left = newNode;

        break;

    }

}

else

{
```

```
                current = current.Right;

                if (current == null)

                {

                    parent.Right = newNode;

                    break;

                }

            }

        }

    }

}
```

12·2·2 遍历二叉查找树

现在有了实现 BST 类的基础，但是迄今为止所有能做的就是向 BST 插入节点。这里需要能遍历 BST 以便可以按照几种不同的顺序访问到各个节点。

这里有三种遍历方法可用来处理 BST：中序遍历、先序遍历和后序遍历。中序遍历按照节点关键字数值的升序顺序访问 BST 内的所有节点。先序遍历首先访问根节点，接着访问根节点的左子节点下的子树内的节点，然后访问根节点的右子节点下的子树内的节点。虽然人们很容易理解为什么想要执行中序遍历，但是他们很少明白为什么需要先序遍历和后序遍历。这里会显示出这三种遍历的代码，并且在稍后的小节内对他们的用法进行解释说明。

最好把中序遍历写成递归的过程。既然这个方法是按照升序方式访问每一个节点，所以此方法必须访问到每棵子树的左节点和右节点，跟着是访问根节点的左子节点下的子树，再接着是访问根节点的右子节点下的子树。图 12-4 描绘了中序遍历的路径。

（原书 P224 页 图）

图 12-4 中序遍历

中序遍历方法的代码如下所示：

```
public void InOrder(Node theRoot)
```

```
{

    if (!(theRoot == null))

    {

        InOrder(theRoot·Left);

        theRoot·DisplayNode();

        InOrder(theRoot·Right);

    }

}
```

为了说明这个方法是如何工作的，下面来研究一个程序，此程序向 BST 内插入了一系列的数。然后这里将调用 inOrder 方法来显示放置在 BST 内的数。下面是代码：

```
static void Main()

{
```

```
        BinarySearchTree nums = 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);

    }
```

输出如下所示：

Inorder traversal:

3 16 22 23 37 45 99

这个序列按照数值的升序方式表示出 BST 的内容，这正是中序遍历所期望的结果。

下面的图 12-5 说明了 BST 和中序遍历的路径：

（原书 P225 页 图）

图 12-5 中序遍历路径

现在一起来研究一下先序遍历的代码：

```
public void PreOrder(Node theRoot)

{
```

```
if (!(theRoot == null))

{

theRoot·DdisplayNode();

PpreOrder(theRoot·Left);

PpreOrder(theRoot·Right);

}

}
```

注意 preOrder 方法和 inOrder 方法之间唯一的区别就是上述三行代码放置的位置。在 inOrder 方法中 displayNode 方法的调用是夹在两个递归调用之间，而在 preOrder 方法中 displayNode 方法的调用是放置在第一行的。

如果在先前的程序中把 inOrder 调用替换为 preOrder，就会得到下列输出：

```
Preorder traversal:

23 16 3 22 45 37 99
```

最后来编写一个执行后序遍历的方法：

```
public void PostOrder(Node theRoot)

{

    if (!(theRoot == null))

    {

        PostOrder(theRoot·Left);

        PostOrder(theRoot·Right);

        theRoot·DisplayNode();

    }

}
```

这个方法与其他两种遍历方法的区别仍旧是递归调用和 displayNode 调用所放置的位置。在后序遍历中，方法首先遍历左子树，然后是右子树。下面是后序遍历方法所产生的输出：

Postorder traversal:

3 22 16 37 99 45 23

在本章的稍后部分大家会看到一些采用这三种遍历方法遍历 BST 的实际编程例子。

12·2·3 在二叉查找树中查找节点和最大/最小值

对于 BST 有三件最容易做的事情：查找一个特殊数值，找到最小值，以及找到最大值。本节会研究这三种操作。

由于 BST 的属性，所以查找最小值和最大值的代码几乎都是微不足道的事情。人们总可以在根节点左子树的最左侧子节点上找到 BST 内的最小值。另一方面，则会在根节点右子树的最右侧子节点上找到 BST 内的最大值。

这里首先是查找最小值的代码：

```
public int FindMin()

{

    Node current = root;

    while (!(current·Left == null))
```

```
    current = current·Left;

return current·Data;

}
```

这个方法由创建一个 Node 对象开始，并且把此对象设置为 BST 的根节点。然后，方法会查看左子节点内的数值是否为空（null）。如果在左子节点上存在非空节点，那么程序会把当前节点设置为此节点。继续这样的操作直到找到一个节点的左子节点为空（null）。这就意味着在下面没有更小的数值了，而且已经找到了最小值。

下面是在 BST 中查找最大值的代码：

```
public int FindMax()

{

    Node current = root;

    while (!(current·Right == null))

        current = current·Right;

    return current·Data;
```

```
}
```

这个方法看上去几乎等同于 FindMin()方法，只是方法是在 BST 的右子节点上移动而不是在左子节点上移动。

这里将看到的最后一个方法是 Find 方法，用它来确定是否在 BST 内存储了一个特殊值。这个方法首先创建一个 Node 节点，并且把它设置为 BST 的根节点。接下来方法会查看关键字（要查找的数据）是否在这个节点内。如果在，那么这个方法就简单地返回当前节点并且退出。如果在根节点内没有找到该数据，就把要查找的数据与存储在当前节点内的数据进行比较。如果关键字小于当前数据值，就把当前节点设置为左子节点。如果关键字大于当前数据值，就把当前节点设置为右子节点。如果当前节点为空（null），那么这个方法的最后一段就会返回空（null）作为方法的返回值，这表明到达 BST 的末尾也没有找到关键值。当 while 循环结束的时候，在 current 中存储的数值就是要查找的数值。

下面是 Find 方法的代码：

```
public Node Find(int key)

{

    Node current = root;

    while (current.iData != key)

    {
```

```
        if (key < current·iData)

            current = current·Left;

        eElse

            current = current·Right;

        if (current == null)

        return null;

    }

    return current;

}
```

12·2·4 从 BST 中移除叶子节点

至少与本节要讨论的移除操作相比，到目前为止在 BST 上执行的操作都不复杂。对于某些情况而言，从 BST 中移除节点几乎是微不足道的。但是对另外一些情况而言，它却需要对其代码有特别的关注，否则会有破坏 BST 正确层次顺序的风险。

研究从 BST 中移除节点就先从讨论一个最简单的实例开始吧，这个实例就是移除叶子。因为不需要考虑子节点的问题，所以移除叶子是最简单的事情。唯一要做的就是把目标节点的父节点的每一个子节点设置为空（null）。当然，节点始终存在，只是与该节点没有任何连接了。

移除叶子节点的代码段如下所示（此代码也包括了 Delete 方法的开始部分，这部分内容声明了一些数据成员，并且移动到要删除的节点上）：

```
public boolNode Delete(int key)

{

    Node current = root;

    Node parent = root;

    bool isLeftChild = true;

    while (current·Data != key)

    {

        parent = current;

        if (key < current·Data)
```

```
{

    isLeftChild = true;

    current = current.Right;

}

else

{

    isLeftChild = false;

    current = current.Right;

}

if (current == null)

    return false;
```

```
    }

    if ((current·Left == null) & (current·Right == null))

        if (current == root)

            root == null;

        else if (isLeftChild)

            parent·Left = null;

        else

            parent·Right = null;

    return true;

}
```

while 循环会取走要删除的节点。第一个检测是查看这个节点的左子节点和右子节点是否为空（null）。然后检测这个节点是否是根节点。如果是，就把它设置为空（null）。否则，既把父节点的左子节点设置为空（null）（如果 isLeftChild 为真），也把父节点的右子节点设置为空（null）。

12·2·5 删除带有一个子节点的节点

当要删除的节点有一个子节点的时候，需要检查四个条件：1·这个节点的子节点可能是左子节点；2·这个节点的子节点可能是右子节点；3·要删除的这个节点可能是左子节点；4·要删除的这个节点可能是右子节点。

下面是代码片段：

```
else if (current·Right == null)

if (current == root)

root = current·Left;

else if (isLeftChild)

parent·Left = current·Left;

else

parent·Right = current·Right;
```

```
else if (current·Left == null)

if (current == root)

root = current·Right;

else if (isLeftChild)

parent·Left = parent·Right;

else

parent·Right = current·Right;
```

首先，查看右子节点是否为空（null）。如果是，就接着查看是否在根节点上。如果在，就把左子节点移动到根节点上。否则，如果当前节点是左子节点，那么把新的父节点的左子节点设置为当前的左子节点。或者，如果在右子节点上，那么把父节点的右子节点设置为当前的右子节点。

12·2·6 删除带有两个子节点的节点

当需要删除带有两个子节点的节点的时候，删除操作现在有了窍门。为什么呢？请看 图 12-6。如果要删除标记为 52 的节点，需要重构这棵树。这里不能用起始节点为 54 的子树

来替换它，因为 *54* 已经有一个左子节点了。

（原书 P*231* 页 图 *1*）

①、 要删除的节点

②、 无法移动子树

图 *12-6* 删除带有两个子节点的节点

这个问题的答案是把中序后继节点移动到要删除节点的位置上。这个工作很容易做，除非后继节点本身有子节点。但是即使它有子节点，仍然还是有办法解决的。图 *12-7* 展示了如何利用中序后继节点来实现这个操作。

（原书 P*231* 页 图 *2*）

①、 移动中序后继节点

图 *12-7* 移动中序后继节点

为了找到后继节点，要到原始节点的右子节点上。根据定义这个节点必须比原始节点大。然后，开始沿着左子节点路径走直到用完节点为止。既然子树（像一棵树）内的最小值必须是在左子节点路径的末端，沿着这条路径到达末端就会找到大于原始节点的最小节点。

下面是找到要删除节点的后继节点的代码：

```
public Node GetSuccessor(Node delNode)

{

Node successorParent = delNode;

Node successor = delNode;

Node current = delNode.Right;

while (!(current == null))

{

successorParent = current;
```

```
successor = current;

current = current·Left;

}

if (!(successor == delNode·Right))

{

successorParent·Left = successor·Right;

successor·Right = delNode·Right;

}

return successor;

}
```

现在需要看两种特殊情况：后继节点是要删除节点的右子节点，以及后继节点是要删除节点的左子节点。先从第一种情况开始。

首先把要删除的节点标记为当前节点。接着把此节点从其父节点的右子节点中移除，并且把父节点的右子节点指向后继节点。然后，移除当前节点的左子节点，并且把后继节点的左子节点设置为当前节点的左子节点。此操作的代码如下所示：

```
else

{

Node successor = GetSuccessor(current);

if (current == root)

root = successor;

else if (isLeftChild)

parent·Left = successor;

else

parent·Right = successor;

successor·Left = current·Left;
```

```
}
```

现在一起来看看当后继节点是要删除节点的左子节点的情况。执行此操作的算法描述如下：

1· 把后继节点的右子节点赋值为后继节点的父节点的左子节点。

2· 把要删除节点的右子节点赋值为后继节点的右子节点。

3· 从父节点的右子节点中移除当前节点，并且把它指向后继节点。

4· 从当前节点中移除当前节点的左子节点，并且把它指向后继节点的左子节点。

这个算法的部分内容是在 GetSuccessor 方法中实现的，还有部分内容是在 Delete 方法中实现的。GetSuccessor 方法的代码段如下所示：

```
if (!(successor == delNode·Right))

{

successorParent·Left = successor·Right;

successor·Right = delNode·Right;

}
```

Delete 方法的代码如下所示：

（原书 P233 页 代码 2）

这是 Delete 方法的完整代码。因为这个方法有些复杂，所以一些二叉查找树的实现简单地标记要删除的节点，并且在执行查找和遍历的时候包含了检查标记的代码。

下面是 Delete 方法的完整代码：

```
public bool Delete(int key)

{

Node current = root;

Node parent = root;

bool isLeftChild = true;
```

```
while (current·Data != key)

{

parent = current;

    if (key < current·Data)

    {

        isLeftChild = true;

        current = current·Right;

    }

    else

    {

        isLeftChild = false;

        current = current·Right;
```

```
        }

        if (current == null)

            return false;

}

if ((current·Left == null) & (current·Right == null))

if (current == root)

            root = null;

        else if (isLeftChild)

            parent·Left = null;

        else if (current·Right == null)

            if (current == root)
```

```
            root = current·Left;

        else if (isLeftChild)

            parent·Left = current·Left;

        else

            parent·Right = current·Right;

    else if (current·Left == null)

        if (current == root)

            root = current·Right;

        else if (isLeftChild)

            parent·Left = parent·Right;

        else
```

```
            parent·Right = current·Right;

        else

        {

            Node successor = GetSuccessor(current);

            if (current == root)

                root = successor;

            else if (isLeftChild)

                parent·Left = successor;

            else

                parent·Right = successor;

            successor·Left = current·Left;

}
```

```
retum true;

}
```

小结

二叉查找树是被称为树的数据结构的一种特殊类型。树是相互连接的节点（对象由数据字段以及连接其他节点的链接字段组成）的群集。二叉树是一种专有的树结构，其每个节点只能有两个子节点。二叉查找树是二叉树的一种特例，其满足的条件是把较小数值存储在左子节点内而把较大数值存储在右子节点内。

在二叉查找树内查找最小值和最大值的算法是非常容易编写的。人们还可以按照不同的顺序（中序遍历、先序遍历和后序遍历）简单地定义遍历二叉查找树的算法。这些定义利用递归，从而保证代码行数最少，但同时也使得对它们的分析困难了许多。

二叉查找树最适合用于存储在结构中的数据是随机顺序的情况。如果树内的数据是有序的或者是接近有序的，那么树会不平衡，而且查找算法也不会工作正常。

练习

33· 请编写一个程序来产生 *10000* 个 *0-9* 之间的随机整数，并且把它们存储在二叉查找树内。请利用本章介绍的算法之一，显示出整数的列表以及它们出现在树内的次数。

34· 请为 BinarySearchTree 类增加一个函数，用它来统计树内边的数量。

35· 请改写练习 *1* 以便它可以存储来自文本文件的单词。显示文件内的所有单词以及单词在文件中出现的次数。

36· 算术表达式可以存储在二叉查找树内。请修改 BinarySearchTree 类以便诸如 2 + 3 * 4 / 5 这样的表达式可以利用正确的运算符优先规则进行适当地计算。

第13章 集合

所谓集合是特殊元素们的一种聚合。集合的元素被称为是成员。集合有两个最重要的属性，一个是集合成员都是无序的，另一个则是集合的成员不会出现超过一次。在计算机科学领域内集合扮演着非常重要的角色，但是不把集合包含作为 C#语言的一种数据结构。

本章会讨论 Set 类的开发。这里不是仅提供一种实现，而是提供两种实现。对于非数字的数据项，这里会用散列表作为潜在的数据存储来提供一种十分简单的实现。用这种实现所产生的问题就是它的效率问题。而针对数字值的更为有效的 Set 类则会利用位数组作为它的数据存储。这种形式就是第二种实现的基础。

13·1 集合的基础定义、操作及属性

人们把集合定义成相关成员的无序聚集，而且集合中的成员不会出现超过一次。集合书写成用一对闭合大括号包裹成员列表的形式，例如{0，1，2，3，4，5，6，7，8，9}。只要全部成员只书写一次，就可以按照任意顺序书写集合，所以此前的集合实例还可以写成{9，8，7，6，5，4，3，2，1，0}或其他任意成员组合的形式。

13·1·1 集合的定义

为了使用集合需要知道一些有关集合的定义。

1· 不包含任何成员的集合称为空集合。全域是所有可能成员的集合。

2· 如果两个集合包含完全一样的成员，那么就认为这两个集合相等。

3· 如果第一个集合的全部成员都包含在第二个集合内，就认为第一个集合是第二个集合的子集。

13·1·2 集合的操作

下面描述了在集合上执行的基本操作。

1· 联合：把一个集合的成员与另一个集合的成员合并从而获得新的集合。

2· 交叉：把已经存在于另一个集合的所有成员添加给一个集合从而获得新的集合。

3· 差异：把不存在于另一个集合的所有成员添加给一个集合从而获得新的集合。

13·1·3 集合的属性

下面是为集合定义的属性。

1· 与空集合的交叉是空集合。与空集合的联合就是集合本身。

2· 与自身的交叉是集合本身。与自身的联合还是集合本身。

3· 交叉与联合是可交换的。换句话说，集合 1 交叉集合 2 等价于集合 2 交叉集合 1，同样原理也适用于两个集合的联合。

4· 交叉与联合是可结合的。集合 1 交叉（集合 2 交叉集合 3）等价于（集合 1 交叉集合 2）交叉集合 3，同样原理也适用于多个集合的联合。

5· 两个集合联合后与集合交叉是可分布的。换句话说，集合 1 交叉（集合 2 联合集合 3）等价于（集合 1 交叉集合 2）联合（集合 1 交叉集合 3）。两个集合交叉后与集合联合也是可以这样操作的。

6· 集合本身与另一个集合联合后再与自身交叉产生的结果还是集合本身。集合本省与另一个集合交叉后再与自身联合也仍然是产生集合本身。这被称为吸收定律。

7· 当把两个集合联合或者交叉后与另一个集合进行差异操作的时候，下列等式成立。等式有：集合 1 差异（集合 2 联合集合 3）等价于（集合 1 差异集合 2）交叉（集合 1 差异集合 3）。此外，集合 1 差异（集合 2 交叉集合 3）等价于（集合 1 差异集合 2）联合

（集合 *1* 差异集合 *3*）。这些等式被称为是德摩根定律。

13·2 第一个用散列表的 SET 类的实现

第一个 Set 类的实现将用散列表来存储集合的成员。HashTable 类是·NET 框架库中较为有效的数据结构之一，而且当速度很重要的时候对于大多数类的实现而言它应该是大家的选择。既然 Set 在 C#语言中是保留词，所以将会调用类 CSet。

13·2·1 类数据成员和构造器方法

为了 CSet 类这里只需要一个数据成员和一个构造器方法。数据成员就是散列表，而构造器方法则是实例化散列表。下面就是代码：

```
public class CSet

{

    private Hashtable data;

    public CSet()
```

```
    {

        data = new Hashtable();

    }

    // More code to follow

}
```

13·2·2Add 方法

为了给集合添加成员，Add 方法需要首先检查来确保成员不在集合内。如果成员在集合内，
那么什么操作也不做。如果成员不在集合内，则把它添加到散列表中。

```
public void Add(Object item)

{

    if (!data·ContainsValue(item))

        data·Add(Hash(item), item);
```

```
}
```

既然数据项必须作为键值对添加到散列表中，所以通过添加要加入到集合内的数据项字符的ASCII 码值的方法就可以计算散列值。下面是 Hash 函数：

```
private string Hash(Object item)

{

    char[] chars;

    string s = item·ToString();

    int hashValue = 0;

    chars = s·ToCharArray();

    for (int i = 0; i <= chars·GetUpperBound(0); i++)

        hashValue += (int)chars[i];

    return hashValue·ToString();
```

```
}
```

*13·2·3*Remove 方法和 Size 方法

当然，这里需要能把成员从集合内移除掉的方法，还需要确定集合内成员数量（多少）的方法。下面就是简单明了的方法：

```
public void Remove(Object item)

{

    data·Remove(Hash(item));

}

public int Size()

{

    return data·Count;

}
```

13·2·4Union 方法

Union 方法利用先前讨论过的形成新集合的 Union 操作来把两个集合合并。这个方法首先会通过添加第一个集合全部成员的方式来构建一个新的集合。然后，方法会检查第二个集合内的每一个成员从而确定这些成员是否已经在第一个集合内。如果检查的成员在第一个集合内，那么就会跳过这个成员，反之则会把这个成员添加到新集合内。

下面是代码：

```
public CSet Union(CSet aSet)

{

CSet tempSet = new CSet();

    foreach (Object hashObject in data·Keys)

        tempSet·Add(this·data[hashObject]);

    foreach (Object hashObject in aSet·data·Keys)

        if (!(this·data·ContainsKey(hashObject)))

            tempSet·Add(aSet·data[hashObject]);
```

```
    return tempSet;

}
```

13·2·5Intersection 方法

Intersection 方法循环遍历集合的关键字，并且检查是否会在传递的集合内找到该关键字。
如果找到，就把该成员添加到新集合内并且跳过其他操作。

```
public CSet Intersection(CSet aSet)

{

    CSet tempSet = new CSet();

    foreach (Object hashObject in data·Keys)

    if (aSet·data·Contains(hashObject))

            tempSet·Add(aSet·data[hashObject]);
```

```
tempSet·Add(aSet·GetValue(hashObject))

    return tempSet;

}
```

13·2·6Subset 方法

一个集合是另一个集合子集的第一要素就是该集合在尺寸上必须小于第二个集合。Subset 方法会首先检查集合的大小，如果第一个集合合格，再接着检查第一个集合的每一个成员是否是第二个集合的成员。代码如下所示：

```
public bool Subset(CSet aSet)

{

    if (this·Size() > aSet·Size())

        return false;

    else

        foreach (Object key in this·data·Keys)
```

```
            if (!(aSet.data.Contains(key)))

                return false;

    return true;

}
```

13.2.7Difference 方法

这里已经讨论过如何获得两个集合的差异。为了执行此推算，方法会循环遍历第一个集合的关键字，并且寻找与第二个集合匹配的任何内容。如果成员存在于第一个集合内而又没有在第二个集合内找到的话，就把此成员添加到新集合内。下面就是代码（连同 ToString 方法一起）：

```
    public CSet Difference(CSet aSet)

    {

        CSet tempSet = new CSet();

        foreach (Object hashObject in data.Keys)
```

```
            if (!(aSet·data·Contains(hashObject)))

                tempSet·Add(data[hashObject]);

        return tempSet;

}

public override string ToString()

{

        string s = "";

        foreach (Object key in data·Keys)

            s += data[key] + " ";

        return s;

}
```

13·2·8 测试 CSet 实现的程序

下面的程序测试了 CSet 类的实现。方法是创建两个集合，执行两个集合的联合操作及两个集合的交叉操作，找到两个集合的子集和两个集合的差异。

下面是程序：

```
static void Main()

{

    CSet setA = new CSet();

    CSet setB = new CSet();

    setA·Add("milk");

    setA·Add("eggs");

    setA·Add("bacon");

    setA·Add("cereal");
```

```
setB·Add("bacon");

setB·Add("eggs");

setB·Add("bread");

CSet setC = 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·Subset(setA))

        Console·WriteLine("b is a subset of a");

    else

        Console·WriteLine("b is not a subset of a");

}
```

此程序的输出是：

（原书P244页 截图1）

如果注释掉添加给集合 B 的“bread”，就会得到下列输出：

（原书 P244 页 截图 2）

在第一个实例中，集合 B 不是集合 A 的子集，因为它包含单词 bread。如第二个屏幕截图所示那样，移除掉作为成员的 bread 就使得集合 B 成为集合 A 的子集了。

13·3CSET 类的 BITARRAY 实现

CSet 类以前的实现是针对非数字的对象，但是这仍然会在一些方面特别是针对大集合方面存在效率低的问题。当不得不处理数字集合的时候，使用 BitArray 类作为存储集合成员的数据结构会是一种更有效的实现。本书的第七章会深度探讨 BitArray 类。

13·3·1 使用 BitArray 实现的概述

利用 BitArray 来存储整数集合成员有几点好处。首先，由于实际上只存储布尔数值，所以对存储空间的要求很小。第二个好处是想要对集合执行的四个主要操作（联合、交叉、差异和求子集）都可以利用简单的布尔运算符（And、Or 和 Not）来实现。这些方法的实现要比用散列表的实现快许多。

用 BitArray 来创建整数集合的存储策略如下所示：假设要把成员 *1* 添加到集合内。这里就把索引位置为 *1* 的数组元素简单设置为 True。如果要把 *4* 添加到集合内，就把位置为 *4* 的元素设置为 True，如此反复下去。

通过简单检测数组位置上的数值在是否为 True 就可以确定有哪些成员在集合内了。此外，通过把数组位置设置为 False 的方法，还可以简单地从集合内移除掉成员。

利用布尔值来计算两个集合的联合是很简单高效的。既然两个集合的联合就是两个集合成员的合并，所以通过或操作两个 BitArray 的相应元素就可以构建新的联合集合了。换句话说，如果某个成员在任一 BitArray 对应位置上的数值为 True，那么就把该成员添加到新集合内。

计算两个集合的交叉类似于计算联合的操作：致使这里用 And 运算符代替了 Or 运算符来执行操作。类似地，两个集合的差异可以用来自第一个集合的成员与第二个集合对应成员的非进行 And 运算符操作来实现。利用和找到差异相同的公式就可以确定一个集合是否是另外一个集合的子集。例如，如果：setA*(*index *)&&!(*setB*(*index*))*计算的值为 False，那么集合 A 就不是集合 B 的子集。

*13·3·2*BitArray 集合的实现

基于 BitArray 的 CSet 类的代码如下所示：

```
public class CSet
```

```
{

    private BitArray data;

    public CSet()

    {

        data = new BitArray(5);

    }

    public void Add(int item)

    {

        data[item] = true;

    }

    public bool IsMember(int item)
```

```
{

    return data[item];

}

public void Remove(int item)

{

    data[item] = false;

}

public CSet Union(CSet aSet)

{

    CSet tempSet = new CSet();

    for (int i = 0; i <= data·Count - 1; i++)
```

```
            tempSet·data[i] = (this·data[i] || aSet·data[i]);

        return tempSet;

    }

    public CSet Intersection(CSet aSet)

    {

        CSet tempSet = new CSet();

        for (int i = 0; i <= data·Count - 1; i++)

            tempSet·data[i] = (this·data[i] && aSet·data[i]);

        return tempSet;

    }

    public CSet Difference(CSet aSet)
```

```
{

    CSet tempSet = new CSet();

    for (int i = 0; i <= data·Count - 1; i++)

        tempSet·data[i] = (this·data[i] &&

        (!(aSet·data[i])));

    return tempSet;

}

public bool IsSubset(CSet aSet)

{

    CSet tempSet = new CSet();

    for (int i = 0; i <= data·Count - 1; i++)
```

```
            if (this·data[i] && (!(aSet·data[i])))

                return false;

        return true;

}

public override string ToString()

{

        string s = "";

        for (int i = 0; i <= data·Count - 1; i++)

            if (data[i])

                s += i;

        return s;
```

```
    }

}

static void Main()

{

    CSet setA = new CSet();

    CSet setB = new CSet();

    setA·Add(1);

    setA·Add(2);

    setA·Add(3);
```

```
setB·Add(2);

setB·Add(3);

CSet setC = 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());
```

```
    bool flag = setB·IsSubset(setA);

    if (flag)

        Console·WriteLine("b is a subset of a");

    else

        Console·WriteLine("b is not a subset of a");

}
```

此程序的输入是：

（原书P247页 截图）

小结

集合和集合理论提供了许多计算科学理论的基础。尽管一些编程语言提供了内置式的集合数

据类型（比如 Pascal 语言），而且其他一些编程语言通过库提供了集合数据结构（比如 Java 语言），但是 C#语言不提供集合数据类型或集合数据结构。

本章讨论了集合类的两种不同实现，一种是利用散列表作为潜在的数据存储，而另一种实现则是使用位数组作为数据存储。位数组的实现只适用于存储整数集合成员，而散列表的实现则将存储任意数据类型的成员。位数组的实现比散列表的实现本质上更加有效，而且应该用在把整数值存储到集合内的任何时候。

练习

37. 请用散列表的实现和位数组的实现来创建两对集合。两种实现都应使用相同的集合。利用 Timing 类来比较每种实现的主要操作（即联合、交叉、差异和取子集），并且报告出时间上的差异。

38. 请修改散列表的实现以便用 ArrayList 来存储集合成员而不是散列表。请把这种实现的主要操作的运行时间与用散列表的实现的操作时间进行比较。看看在时间方面的差异到底是什么？

第 *14* 章 高级排序算法

本章会介绍对数据进行排序的算法，而且这些算法会比第 *4* 章讲述的算法更加复杂一些。此外，这些算法也会更加高效，而且其中之一的快速排序算法被广泛认为是在绝大多数情况下最有效的一种排序。本章还会介绍的其他几种排序算法有希尔排序算法、归并排序算法以及堆排序算法。

为了对这些高级排序算法进行比较，本章会首先讨论每种算法的实现方法，然后在练习内将会使用 Timing 类对这些算法的运行效率进行比较。

14.1 希尔排序算法

希尔排序算法是根据它的发明者唐纳德·希尔的名字命名的。此算法从根本上而言就是插入排序算法的一种改进。如同在插入排序中所做的那样，本算法的关键内容是对远距离而非相邻的数据项进行比较。当算法循环遍历数据集合的时候，每个数据项间的距离会缩短，直到算法对相邻数据项进行比较时才终止。

希尔排序算法采用升序方式对远距离的元素进行排序。序列必须从 1 起始，但是可以按照任意数量进行自增。一种好的可用的自增方法是基于下列代码段的：

```
while (h <= numElements / 3)

h = h * 3 + 1;
```

这里的 numElements 表示了数据集合内待排序元素的数量，例如一个数组。

例如，如果由上述代码产生的序列数是 4，那么就是对数据集合内每次第 4 个元素进行排序。接着采用下列代码来选择一个新的序列数：

```
h = (h - 1) / 3;
```

然后，对后续的 h 个元素进行排序，依此类推。

下面就来看一看希尔排序算法的代码（这里采用了第 4 章的 ArrayClass 代码）：

```
public void ShellSort()

{

    int inner, temp;

    int h = 3;

    while (h > 0)

    {

        for (int outer = h; outer <= numElements - 1; outer++)

        {

            temp = arr[outer];

            inner = outer;

            while ((inner > h - 1) && arr[inner - h] >= temp)
```

```
            {

                arr[inner] = arr[inner - h];

                inner -= h;

            }

            arr[inner] = temp;

        }

        h = (h - 1) % 3;

    }

}
```

测试此算法的代码如下所示：

```
static void Main()
```

```
{

    const int SIZE = 19;

    CArray theArray = new CArray(SIZE);

    Random random = new Random();

    for (int index = 0; index < SIZE; index++)

        theArray·Insert(random·Next(100) + 1);

    Console·WriteLine();

    theArray·DisplayElements();

    Console·WriteLine();

    theArray·ShellSort();

    theArray·DisplayElements();
```

}

此程序输出是：

（原书P251页 截图）

希尔排序算法经常被认为是一种很好的高级排序算法。这是因为它十分容易实现，甚至是对于包含好几万个元素的数据集合而言其性能也是可以接受的。

14.2 归并排序算法

归并排序算法是一个非常好的递归算法的实例。这个算法把数据集合分成两个部分，然后对每部分递归地进行排序。当两个部分都排序好时，再用合并程序把它们组合在一起。

在对数据集合进行排序的时候，操作十分简单。假设在数据集合内有下列这些数据：71、54、 58、29、31、78、2 和 77。首先，这里会把数据集合分成两个独立的子集合：即子集合 71、54、58、29 ，以及子集合 31、78、2、77。接着就是对每一部分进行排序:即子集合 29、54、58、71，以及子集合 2、31、77、78。然后把两个子集合进行合并，即 2、29、31、54、58、71、77 和 78。合并过程会比较两个数据子集合（存储在临时数组内）中的第一个元素，并且把较小值复制给另外一个数组。而没有被添加到第三个数组内的元素随后会与另一个数组内的下一个元素进行比较。当然还是会把较小的元素添加到第三个数组内，而且这个过程会持续到两个数组内都没有数据了为止。

但是，如果其中一个数组的元素比另一个数组的元素先用完，那么结果会怎样呢？这种情况很可能会发生，因而算法对这种情况进行了规定。在主循环结束以后，当且仅当两个数组的其中一个还留有数据时可以使用两个额外的循环用来解决这个问题。

现在就来看看执行合并排序的代码。首先是两个方法 MergeSort 和 recMergeSort。第一个方法简单地调用了递归子程序 recMergeSort，而这个子程序对数组进行排序：

```
public void MergeSort()

{

    int[] tempArray = new int[numElements];

    RecMergeSort(tempArray, 0, numElements - 1);

}

public void RecMergeSort(int[] tempArray, int lbound, int ubound)

{

    if (lbound == ubound)

        return;
```

```
    else

    {

        int mid = (int)(lbound + ubound) / 2;

        RecMergeSort(tempArray, lbound, mid);

        RecMergeSort(tempArray, mid + 1, ubound);

        Merge(tempArray, lbound, mid + 1, ubound);

    }

}
```

在 RecMergeSort 方法中，第一个 if 语句是基于递归的情况。当条件为真时，就会返回到调用它的程序。否则，就要找到数组的中间位置，并且在数组的后半部分（第一个调用 RecMergeSort）递归地调用子程序，然后是在数组的前半部分（第二个调用 RecMergeSort）递归地调用子程序。最终，通过调用 Merge 方法来把两部分合成在一个完整的数组。

下面就是 Merge 方法的实现代码：

```
public void Merge(int[] tempArray, int lowp, int highp, int ubound)
```

```
{

    int lbound = lowp;

    int mid = highp - 1;

    int n = (ubound - lbound) + 1;

    int j = 0;

    while ((lowp <= mid) && (highp <= ubound))

    {

        if (arr[lowp] < arr[highp])

        {

            tempArray[j] = arr[lowp];

            j++;
```

```
            lowp++;

        }

        else

        {

            tempArray[j] = arr[highp];

            j++;

            highp++;

        }

}

    while (lowp <= mid)

    {

        tempArray[j] = arr[lowp];
```

```
        j++;

        lowp++;

    }

    while (highp <= ubound)

    {

        tempArray[j] = arr[highp];

        j++;

        highp++;

    }

    for (j = O; j <= n - 1; j++)

        arr[lbound + j] = tempArray[j];
```

```
}
```

这个方法每次由 recMergeSort 子程序调用来执行一个初步的排序。为了更好地实例说明这个方法是如何与 recMergeSort 一起操作的，这里在 Merge 方法的末尾添加了下列这样一行代码：

```
this· DisplayElements();
```

用了这行代码，在排序完成之前就可以观察到在不同临时状态下数组的情况。输出如下所示：

（原书 P254 页 截图）

第一行显示了初始状态的数组。第二行则显示正在对数组的前半部分开始进行排序。一直到到第五行，前半部分的排序才全部完成。第六行显示正在对数组的后半部分开始进行排序，而且第九行显示对数组两个部分的排序都全部完成了。第十行是最终合并后的输出结果，而第十一行只是另外一个对 showArray 方法的调用。

14·3 堆排序算法

堆排序算法利用了一种被称为堆的数据结构。堆和二叉树比较类似，但是又有一些显著的差异。尽管堆排序算法不是本章中最快的算法，但是此算法具有一些吸引人的特点，这些特点在某些情况下很适用。

14·3·1 构造堆

正如前面已经讨论过的那样，堆数据结构类似于二叉树，但是又不完全相同。首先，通常采用数组而不是节点引用的方式来构造堆。并且，堆有两个非常重要的条件：*1*·堆必须是完整的，这就意味着每一行都必须有数据填充。其次，*2*·每个节点所包含的数据要大于或等于此节点下方孩子节点们所包含的数据。图 *14-1* 显示了堆的一个实例。而图 *14-2* 则说明了存储堆的数组。

（原书 P*255* 页 图 *1*）

图 *14-1* 一个堆

（原书 P*255* 页 图 *2*）

图 *14-2* 存储图 *14-1* 中堆的数组

存储在堆内的数据由 Node 类来构建，它类似于其他章节中用到的节点。然而，这个特殊的 Node 类将只存储一种数据，即它的主值或者键值。这里不需要对其他节点的任何引用，但是会希望用到适合此数据的类，这样在需要时可以很容易地改变存储在堆内的数据的类型。Node 类的代码如下：

```
public class Node

{

    Ppublic int data;

    public void Node(int keyByVal key As Integer)

    {

        data = key;

    }

}
```

通过把节点插入到堆数组内的方式可以构造堆，而堆数组的元素就是堆的节点。这里始终要把新节点放置在数组末尾的空元素内。问题是这样做很可能会打破堆的构造条件，因为新节点的数据值可能会大于它上面某些节点的值。为了恢复数组从而达到正确的堆构造条件，需要把新节点向上移动，一直要把它移动到数组内合适的位置上为止。这里使用被称为 ShiftUp 的方法来实现此操作。代码如下所示：

```
public void ShiftUp(int index)

{

    int parent = (index - 1) / 2;

    Node bottom = heapArray[index];

    while ((index > 0) && (heapArray[parent].data < bottom.data))

    {

        heapArray[index] = heapArray[parent];

        index = parent;

        parent = (parent - 1) / 2;

    }

    heapArray[index] = bottom;
```

```
}
```

而且下面是 Insert 方法的实现代码：

```
public bool Insert(int key)

{

    if (currSize == maxSize)

        return false;

    heapArray[currSize] = new Node(key);

    currSize++;

    return true;

}
```

这里会把新节点填加到数组的末尾。这样做会立刻打破堆构造的条件，所以通过 ShiftUp 方法来找到新节点在数组内的正确位置。此方法的参数就是新节点的索引。方法的第一行会计算出此节点的父节点。接着方法会把新节点保存到一个名为 bottom 的 Node 变量内。随后，while 循环会找到新节点的正确位置。方法的最后一行会把新节点从临时放置的变量 bottom 内复制到数组中正确的位置上。

从堆中移除掉节点始终意味着删除最大值的节点。这是很容易实现的，因为最大值始终在根节点上。问题是一旦移除掉根节点，堆就不完整了，就需要对其进行重组。下面这个算法用来使堆再次完整：

1·移除掉根节点。

2·把最后位置上的节点移动到根上。

3·把最后的节点向下移动，直到它在底下为止。

当连续应用这个算法的时候，就会按照排列顺序把数据从堆中移除掉。下面就是 Remove 方法和 TrickleDown 方法的实现代码：

```
public Node Remove()

{

    Node root = heapArray[0];

    currSize--;

    heapArray[0] = heapArray[currSize];

    ShiftDown(0);
```

```
        return root;

    }

    public void ShiftDown(int index)

    {

        int largerChild;

        Node top = heapArray[index];

        while (index < (int)(currSize / 2))

        {

            int leftChild = 2 * index + 1;

            int rightChild = leftChild + 1;

            if ((rightChild < currSize) && heapArray[leftChild].data <
```

```
heapArray[rightChild]·data)

                largerChild = rightChild;

            else

                largerChild = leftChild;

            if (top·data >= heapArray[largerChild]·data)

                break;

            heapArray[index] = heapArray[largerChild];

            index = largerChild;

        }

        heapArray[index] = top;

}
```

这就是需要执行堆排序的全部操作，因此下面就来看看这样一个程序。此程序构造了一个堆，

并且对其进行了排序：

```
using System;




public class Heap

{

    Node[] heapArray = null;

    private int maxSize = 0;

    private int currSize = 0;

    public Heap(int maxSize)

    {

        this.maxSize = maxSize;

        heapArray = new Node[maxSize];
```

```
    }

public bool InsertAt(int pos, Node nd)

{

        heapArray[pos] = nd;

        return true;

}

    public void ShowArray()

    {

        for (int i = 0; i < maxSize; i++)

        {

            if (heapArray[i] != null)
```

```
                System·Console·Write(heapArray[i]·data + "  ");

        }

}

static void Main()

{

        const int SIZE = 9;

        Heap aHeap = new Heap(SIZE);

        Random RandomClass = new Random();

        for (int i = 0; i < SIZE; i++)

        {
```

```
        int rn = RandomClass.Next(1, 100);

        aHeap.Insert(rn);

    }

    Console.Write("Random: ");

    aHeap.ShowArray();

    Console.WriteLine();

    Console.Write("Heap: ");

    for (int i = (int)SIZE / 2 - 1; i >= 0; i--)

        aHeap.ShiftDown(i);

    aHeap.ShowArray();

    for (int i = SIZE - 1; i >= 0; i--)
```

```
        {

            Node bigNode = aHeap.Remove();

            aHeap.InsertAt(i, bigNode);

        }

        Console.WriteLine();

        Console.Write("Sorted: ");

        aHeap.ShowArray();

    }

}
```

第一个 for 循环通过向堆内插入随机数的方式开始了构造堆的过程。第二个循环是恢复堆，而随后的第三个 for 循环则是用 Remove 方法和 TrickleDown 方法来重新构造有序的堆。程序的输出如下所示：

（原书P258页 截图）

堆排序是本章介绍的速度排名第二的高级排序算法。只有下一小节要讨论的快速排序算法比此算法速度更快。

14.4 快速排序算法

作为本章在讨论的速度最快的高级排序算法，快速排序算法是实至名归的。当然这只针对于大量且通常无序的数据集合而言是正确的。如果数据集合很小（含有100个元素或者更少），或者数据是相对有序的，那么就需要采用第4章所讨论的基础排序算法了。

14.4.1 快速排序算法的描述

为了理解快速排序算法的工作原理，假设你是一名教师，现在要把一堆学生的论文按字母顺序进行排序。你可能会选取字母表中间的一个字母，比如字母M。接着把学生名字以字母A到字母M开头的论文放在一堆，再把学生名字以字母N到字母Z开头的论文放在另外一堆。然后你要利用相同的方法把A-M这堆再分成两堆，并且把N-Z这堆也再分成两堆。你要反复这样的操作直到所有小堆（A-C，D-F，…，X-Z）包含易于排序的两个元素或三个元素时为止。一旦所有小堆都有序了，你只需要简单地把这些小堆放在一起就会得到一个有序的论文集合。

如上你已经注意到的那样，这个过程是递归的，因为每一个堆都会被分成更小的堆。一旦把堆分裂成只包含一个元素，那么这个堆就不能继续分裂了，而递归操作也就终止了。

那么人们如何决定在什么位置把数组一分为二呢？虽然有很多种选择，但是这里将只会选取第一个数组元素作为开始：

```
mv = arr[first];
```

一旦做了选择，接下来就需要了解如何把数组元素放入正确的数组“半个部分”内了。（此句中给半个部分加双引号的原因是因为完全有可能数组的两个部分不相等，这要依赖于分割点。）通过创建两个变量 first 和 last 就可以完成这项工作了。这里会把第二个元素存储到 first 内，而把最后一个元素存储到 last 内。还可以创建另外一个变量 theFirst 用来保存数组内的第一个元素。出于对此实例的兴趣数组名就是 arr。

图 *14-3* 描述了快速排序算法的工作原理。

（原书 P*260* 页 图）

图 *14-3* 分裂数组

分割点的值 *= 87*

①使 first 自增，直到它>=分割点的值为止

first 停在 *91* 的位置上（参见图 a）

②使 last 自减，直到它<=分割点的值为止

③把 first 内的值和 last 内的值进行交换

④使 first 自增，直到它>分割点的值或着>last

使 last 自减，直到它<=分割点的值或着<first

⑤last 在 first 之前（或者说是 first 在 last 之后）

所以把 theFirst 内的值和 last 内的值进行交换

⑥重复这个过程

14·4·2 快速排序算法的代码

现在已经了解了算法的工作原理，下面就用 C#语言来编写实现的代码：

```
public void QSort()

{

    RecQSort(0, numElements - 1);

}

public void RecQSort(int first, int last)

{

    if ((last - first) <= 0)

        return;

    else

    {

        int part = this·Partition(first, last);

        RecQSort(first, part - 1);
```

```
            RecQSort(part + 1, last);

        }

}

public int Partition(int first, int last)

{

        int pivotVal = arr[first];

        int theFirst = first;

        bool okSide;

        first++;

        do

        {
```

```
okSide = true;

while (okSide)

    if (arr[first] > pivotVal)

        okSide = false;

    else

    {

        first++;

        okSide = (first <= last);

    }

okSide = true;

while (okSide)

    if (arr[last] <= pivotVal)
```

```
            okSide = false;

        else

        {

            last--;

            okSide = (first <= last);

        }

    if (first < last)

    {

        Swap(first, last);

        this·DisplayElements();

        first++;
```

```
                last--;

            }

        } while (first <= last);

        Swap(theFirst, last);

        this.DisplayElements();

        return last;

}

public void Swap(int item1, int item2)

{

        int temp = arr[item1];

        arr[item1] = arr[item2];
```

```
    arr[item2] = temp;

}
```

原书 P262 代码 2

```
theFirst = arr[(int)arr·GetUpperBound(0) / 2]
```

14·4·3 快速排序算法的改进

如果数组内的数据是随机的，那么选取第一个数值作为“中心点”或者“分割”值是完全合理的。然而，反之情况做这样的选择将会降低算法的性能。

一种比较流行的选择此数值的方法是在数组内确定中间值。通过把取到的数组上限除以 2 的方法就可以得到这个中间值。例如：

（原书P262页 代码2）

研究表明使用这种策略可以减少此算法运行时间大约5个百分点。(参见 Weiss1999, p·243）

小结

本章讨论的算法都比第4章讨论的基础排序算法在执行速度上快很多。人们普遍认为快速排序算法是最快的排序算法，而且应该把它用于大多数排序情况里。构建在几个·Net 框架库类中的 Sort 方法就是用快速排序算法实现的，这就说明快速排序比其他排序算法具有优势。

练习

1· 请编写一个程序来对本章讨论的四种高级排序算法进行比较。为了执行测试，请创建一个随机产生1000个元素的数组。算法的等级是什么呢？当把数组的大小扩大为10000个元素甚至是100000元素的时，又会发生什么呢？

2· 请使用一个较小的数组（少于20个元素）来比较插入排序算法和快速排序算法所用的排序时间。在排序时间上有什么差异呢？你能否解释原因呢？

第 15 章 查找的高级数据结构和算法

本章会介绍一系列用于查找的高级数据结构和算法。所要讲述的数据结构包括红黑树、伸展树以及跳跃表。AVL 树和红黑树是处理不平衡二叉搜索树问题的两种解决方案。而跳跃表则是在使用比红黑树和伸展树更复杂的类树数据结构时的一种替换选择。

15.1 AVL 树

AVL 树是用来维持平衡二叉树的另外一种解决方案。AVL 的命名源于 1962 年发明这种数据结构的两位计算机科学家 G· M· Adelson-Velskii 和 E· M· Landis。AVL 树的基本特征就是左右两个子树的高度差永远不可能大于 1。

15.1.1 AVL 树的基本原理

AVL 树通过持续地比较左右两子树的高度来保证始终处于“平衡”。AVL 树利用一种被称为旋转的技术来保持平衡。

为了理解旋转技术的工作原理，现在就来看一个构建整数二叉树的简单实例。树开始时的状态如图 15-1 所示。如果把数值 10 插入到树中，那么树会变得不平衡，如图 15-2 所示。现在左子树的高度为 2，而右子树的高度还是 0，这违背了 AVL 树的原则。通过执行一个单独的右旋转操作就能把树平衡起来，也就是把数值 40 移动到右下方，如图 15-3 所示。

（原书 P264 页 图）

图 15-1

（原书 P265 页 图 1）

图 15-2

（原书 P265 页 图 2）

图 15-3

现在来看看图 15-4 中的树。如果插入数值 30，那么就会得到图 15-5 中的树。这棵树是不平衡的。所以需要一个所谓的双旋转的操作来进行修正，也就是把数值 40 移动到右下方，且把数值 30 移动到右上方，结果如图 15-6 所示。

（原书P265页 图3）

图15-4

（原书P265页 图4）

图15-5

（原书P265页 图5）

图15-6

15·1·2 AVL树的实现

AVL树的实现由两个类组成：Node类用来保存树中每个节点的数据，而AVLTree类则包含了插入节点的方法和旋转节点的方法。

构造用于AVL树实现的Node类很类似于用于二叉树实现的节点，但是还是有一些显著的差异。AVL树中的每一个节点都必须包含它自身的高度，所以在类中就会包括一个表示高度的数据成员。而且为了比较存储在节点内的数值，还要有类实现IComparable接口。此外，由于节点的高度是如此重要，因而还要包括一个ReadOnly（只读）属性的方法来返回节点的高度。

Node 类的代码如下所示：

```
public class Node : IComparable

{

    public Object element;

    public Node left;

    public Node right;

    public int height;

    public Node(Object data, Node lt, Node rt)

    {

        element = data;

        left = lt;

        right = rt;
```

```
        height = 0;

    }

    public Node(Object data)

    {

        element = data;

        left = null;

        right = null;

    }

    public int CompareTo(Object obj)

{

return (this·element·CompareTo((Node)obj·element));
```

```
return (((int)element).CompareTo((int)obj));

    }

    public int GetHeight()

    {

        if (this == null)

            return -1;

        else

            return this.height;

}

}
```

在 AVLTree 类中的第一个要介绍的方法就是 Insert 方法。这个方法确定了节点插入到树中的位置。此方法是递归的，既要在当前节点大于要插入节点的时候向左移动，又要在当前节点小于要插入节点的时候向右移动。

一旦节点在适当的位置上，就会计算两个子树的高度差值。如果确定树是不平衡的，那么就会进行左旋转或着右旋转操作，还或者进行双左旋转或着双右旋转操作。代码如下所示（在 Insert 方法之后显示了不同旋转方法的代码）：

```
private Node Insert(Object item, Node n)

{

if (n == null)

        n = new Node(item, null, null);

    else if (((int)item).CompareTo((int)n.element) < 0)

    {

        n.left = Insert(item, n.left);

        if (n.left.GetHeight() - n.right.GetHeight() == 2)

            n = n.RotateWithLeftChild(n);

        else
```

```
            n = n.DoubleWithLeftChild(n);

    }

    else if (((int)item).CompareTo((int)n.element) > 0)

    {

        n.right = Insert(item, n.right);

        if (n.right.GetHeight() - n.left.GetHeight() == 2)

            if (((int)item).CompareTo((int)n.right.element) > 0)

                n = RotateWithRightChild(n);

            else

                n = DoubleWithRightChild(n);

    }
```

```
        else

        { ;}// do nothing, duplicate value

        n.height = Math.Max(n.left.GetHeight(), n.right.GetHeight()) + 1;

        return n;

}
```

不同旋转方法的代码如下所示：

```
private Node RotateWithLeftChild(Node n2)

{

        Node n1 = n2.left;

        n2.left = n1.right;

        n1.right = n2;

        n2.height = Math.Max(n2.left.GetHeight(), n2.right.GetHeight()) + 1;
```

```
        n1.height = Math.Max(n1.left.GetHeight(), n2.height) + 1;

        return n1;

    }

    private Node RotateWithRightChild(Node n1)

    {

        Node n2 = n1.right;

        n1.right = n2.left;

        n2.left = n1;

        n1.height = Math.Max(n1.left.GetHeight(), n1.right.GetHeight() + 1);

        n2.height = Math.Max(n2.right.GetHeight(), n1.height) + 1;

        return n2;
```

```
}

private Node DoubleWithLeftChild(Node n3)

{

    n3.left = RotateWithRightChild(n3.left);

    return RotateWithLeftChild(n3);

}

private Node DoubleWithRightChild(Node n1)

{

    n1.right = RotateWithLeftChild(n1.right);

    return RotateWithRightChild(n1);

}
```

还有许多其他的方法可以实现这个类，换句话说就是来自 BinarySearch 类的方法。这里把这些方法的实现留作练习。此外，这里还故意没有实现 AVLTree 类的删除方法。许多 AVL 树的实现会使用懒惰删除。这种方法会对要删除节点进行标记，但并不会真的把节点从树中删除掉。因为删除节点以及重新平衡树的执行开销常常使人望而却步。读者们将会有机会在练习中实践懒惰删除。

15·2 红黑树

AVL 树并不是处理不平衡二叉搜索树的唯一方法。另一种可以用到的数据结构就是红黑树。红黑树根据一系列规则把树上的节点指定为红色或者黑色。通过对树中节点适当的染色，就可以使得树处于近乎完美地平衡。红黑树的一个实例如图 *15-7* 所示（图中黑色节点用阴影表示）：

（原书 P*268* 页 图）

图 *15-7* 红黑树

15·2·1 红黑树规则

在处理红黑树时要遵循如下规则：

1· 树中的每个节点标记成不是红色就是黑色。

2· 把根节点标记为黑色。

3· 如果某个节点是红色，那么它的子节点们必须是黑色。

4· 从一个节点到一个叶子节点的每一条路径都必须包含相同数量的黑色节点。

这些规则使得红黑树处于非常好地平衡，这也意味着对红黑树进行搜索会是十分高效的。然而，就像处理 AVL 树一样，这些规则也会得插入操作和删除操更加困难。

15·2·2 红黑树的插入

往红黑树中插入新的数据项是很复杂的，因为这会导致违背一条先前小节提到的红黑树规则。例如，看一看图 15-8 中的红黑树。

（原书 P269 页 图）

图 15-8

这里把在树中插入一个新数据项看作是一个黑色节点。如果这样做，就会违背规则 4。所以此节点必须标记为红色。如果它的父节点是黑色的，那么一切都没问题了。但是，如果它的父节点是红色的，那么就会违背规则 3。此时就既要改变节点的颜色又要象对待 AVL 树那样旋转节点来调整树。

现在就通过查看一个明确的实例来使此过程变得更加具体。假设要把数值 55 插入到图 15-8 所示的红黑树中。在按照规则从上向下扫描树时发现数值 60 是黑色的，而且它还有两个红色的子节点。这里可以改变每个节点的颜色（即 60 变为红色，50 和 65 变为黑色），然后把 60 旋转到 80 的位置上，再接着执行其他旋转使得子树处于合适的位置。最终的红黑树如图 15-9 所示。此树现在满足所有红黑树的规则，而且树是平衡的。

（原书 P270 页 图）

图 15-9

15·2·3 红黑树实现代码

与其分解代码进行解释，不如在一页内向大家展示红黑树实现的完整代码，并在随后附加代码的说明。这里会先从 Node 类开始，随后接着是 RedBlack 类。

```
using System;

public class Node

{

    public string element;

    public Node left;

    public Node right;

    public int color;

    const int RED = 0;

    const int BLACK = 1;
```

```
public Node(string element, Node left, Node right)

{

    this.element = element;

    this.left = left;

    this.right = right;

    this.color = BLACK;

}

public Node(string element)

{

    this.element = element;

    this.left = left;
```

```
        this.right = right;

        this.color = BLACK;

    }

}

public class RBTree

{

    const int RED = 0;

    const int BLACK = 1;

    private Node current;

    private Node parent;

    private Node grandParent;

    private Node greatParent;
```

```
private Node header;

private Node nullNode;

public RBTree(string element)

{

    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;

    }

    public void Insert(string item)

    {

        grandParent = header;

        parent = grandParent;

        current = parent;

        nullNode.element = item;

        while (current.element.CompareTo(item) != 0)
```

```
{

    Node greatParent = grandParent;

    grandParent = parent;

    parent = current;

    if (item.CompareTo(current.element) < 0)

        current = current.left;

    else

        current = current.right;

    if ((current.left.color) == RED && current.right.color == RED)

        HandleReorient(item);

}
```

```
        if (!(current == nullNode))

            //return

            current = new Node(item, nullNode, nullNode);

        if (item.CompareTo(parent.element) < 0)

            parent.left = current;

        else

            parent.right = current;

        HandleReorient(item);

    }

    public string FindMin()

    {

        if (this.IsEmpty())
```

```
            return null;

        Node itrNode = header.right;

        while (!(itrNode.left == nullNode))

            itrNode = itrNode.left;

        return itrNode.element;

    }

    public string FindMax()

    {

        if (this.IsEmpty())

            return null;

        Node itrNode = header.right;
```

```
    while (!(itrNode.right == nullNode))

        itrNode = itrNode.right;

    return itrNode.element;

}

public string Find(string e)

{

    nullNode.element = e;

    Node current = header.right;

    while (true)

        if (e.CompareTo(current.element) < 0)

            current = current.left;
```

```
        else if (e.CompareTo(current.element) > 0)

            current = current.right;

        else if (!(current == nullNode))

            return current.element;

        else

            return null;

}

public void MakeEmpty()

{

    header.right = nullNode;

}
```

```
public bool IsEmpty()

{

    return (header·right == nullNode);

}

public void PrintRBTree()

{

    if (this·IsEmpty())

        Console·WriteLine("Empty");

    else

        PrintRB(header·right);

}

public void PrintRB(Node n)
```

```
{

    if (!(n == nullNode))

    {

        PrintRB(n·left);

        Console·WriteLine(n·element);

        PrintRB(n·right);

    }

}

public void HandleReorient(string item)

{

    current·color = RED;
```

```
current·left·color = BLACK;

current·right·color = BLACK;

if (parent·color == RED)

{

    grandParent·color = RED;

    if ((item·CompareTo(grandParent·element) < 0) !=
(item·CompareTo(parent·element) < 0))

    {

        current = Rotate(item, grandParent);

        current·color = BLACK;

    }
```

```
            header.right.color = BLACK;

        }

    }

    public Node Rotate(string item, Node parent)

    {

        if (item.CompareTo(parent.element) < 0)

        {

            if (item.CompareTo(parent.left.element) < 0)

                parent.left = RotateWithLeftChild(parent.left);

            else

                parent.left = RotateWithRightChild(parent.left);

            return parent.left;
```

```
    }

    else

    {

        if (item.CompareTo(parent.right.element) < 0)

            parent.right = RotateWithLeftChild(parent.right);

        else

            parent.right = RotateWithRightChild(parent.right);

        return parent.right;

    }

}

public Node RotateWithLeftChild(Node k2)
```

```
{

    Node k1 = k2.left;

    k2.left = k1.right;

    k1.right = k2;

    return k1;

}

public Node RotateWithRightChild(Node k1)

{

    Node k2 = k1.right;

    k1.right = k2.left;

    k2.left = k1;
```

```
        return k2;

    }

}
```

在任何节点有两个红色子节点的时候都会调用 HandleReoient 方法。旋转方法类似于那些用于 AVL 树的旋转方法。此外，由于处理根节点是一种特殊情况，RedBlack 类包括了一个根哨兵节点以及 nullNode 节点，其中 nullNode 节点用来表示此节点是否为空（null）。

15·3 跳跃表

尽管 AVL 树和红黑树在数据搜索和排序方面都是有效的数据结构，但是这两种数据结构都需要重新平衡操作来保持树的平衡，这就导致大量费用和复杂性。还有另外一种数据结构可以使用，它特别适用于查找。这种数据结构提供了树的功效且不需要担心重新平衡问题。这种数据结构叫做跳跃表。

15·3·1 跳跃表的基本原理

构造跳跃表是源于一种用于查找的基础数据结构—链表。众所周知，链表的优势是插入和删除，但是它不善于查找，因为需要顺序遍历每个节点。然而没有理由需要连续遍历每个链。当需要从一系列台阶的底部快速到达顶部的时候，我们会怎么做呢？那就是每次爬两到三层的台阶（如果你有长腿的话，甚至可以更多）。

在链表中也可以通过创建链的不同层次来实现相同的策略。首先从指向表中下一节点的 *0*

层链开始。接着是 *1* 层链，它指向表中的第二个节点，也就是跳过了一个节点。*2* 层链指向表中的第三个节点，即跳过了两个节点，以此类推。当查找某个数据项的时候，可以从高链层开始，接着遍历整个表直到到达的数值大于要寻找的数值时为止。然后，可以倒退到前一个访问的节点，并且向下移动到最底层，同时逐个节点进行查找直到遇到要查找的数值为止。

为了说明跳跃表和链表的区别，下面来研究一下图 *15-10* 和图 *15-11* 中的内容。

（原书 P*276* 页 图 *1*）

图 *15-10* 基础链表

（原书 P*276* 页 图 *2*）

图 *15-11* 每两个节点一组的跳跃表（第 *1* 层）

下面就来看看如何在图 *15-11* 所示的跳跃表的第 *1* 层执行查找操作。第一次查找的数值是 *1133*。第一个要查找的数值是 *1133*。首先来查看基础链表，需要遍历四个节点才找到 *1133*。然而，利用跳跃表只需要遍历两个节点就可以找到了。很明显，在这类查找中利用跳跃表会更加有效。

现在再来看一看如何用跳跃来查找 *1203*。遍历第 *1* 层链直到找到数值 *1223* 为止。此数值大于 *1203*，所以倒退到存储着数值 *1133* 的节点，并且下调一层，开始使用第 *0* 层链。而

下一个节点就是 1203，所以查找终止。这个实例使跳跃表的策略变得清楚明确了。从最高链层开始，利用这些链来遍历表直到到达的数值大于要查找的数值为止。在这个时候，回退到访问过的最后一个节点，并且向下移动到下一个链层，接着重复相同的步骤。最终到达的链层会指引出所要查找的数值。

结果表明通过添加更多的链可以使跳跃表变得更加有效。例如，每 4 个节点有一个链，这个链指向前面第 4 个节点，每 6 个节点有一个连接，且这个链指向前面第 6 个节点，以此类推。这种方案的问题是在插入或者删除节点的时候需要重新安排巨大大量的节点指针，这会使得跳跃表降低不少效率。

这个问题的解决方案是把节点随机地分配到链层上。第一个节点（在头节点后面）可能是第 2 层的节点，而第二个节点则可能是第 4 层的节点，第三个节点又会是第 1 层的节点，如此等等。随机分布链层会使其他操作（查找除外）更加有效，而且它不会真的影响查找次数。用来确定如何随机分布节点的概率分布是基于跳跃表内大约半数的节点都将是第 0 层节点的事实，同时四分之一的节点将是第 1 层的节点，12·5%的节点将是第 2 层的节点，5·75%的节点将是第 3 层的节点，如此等等。

唯一剩下要解释的就是如果确定在跳跃表中将要用到的层次数量。跳跃表的发明人目前是马里兰大学计算机科学教授 William Pugh。他在一篇首次介绍跳跃表的论文（ftp://ftp·cs·umd·edu/pub/skipLists/）中计算出了一个公式。这个公式用 C#代码表示如下所示：

```
(int)(Math·Ceiling(Math·Log(maxNodes) / Math·Log(1/PROB)) - 1);
```

这里的 maxNodes 是所需节点数量的一个近似值，而 PROB 则是一个概率常量，通常为 0·25。

15·3·2 跳跃表的实现

跳跃表的实现需要两个类：一个节点类和一个跳跃表本身的类。这里就从节点类开始吧。

将用于实现的节点会存储一个关键字和一个数值，还有一个用来存储指向其他节点的数组。
代码如下所示：

```
public class SkipNode

{

    public int key;

    public Object value;

    public SkipNode[] link;

    public SkipNode(int level, int key, Object value)

    {

        this·key = key;
```

```
        this·value = value;

        link = new SkipNode[level];

    }

}
```

现在准备构造跳跃表类。首先需要做的就是确定类需要哪些数据成员。下面就是会需要的内容：

- maxLevel：存贮跳跃表所允许的最大层次数。
- level：存储当前层次。
- header：提供进入跳跃表的起始节点。
- probability：存储当前链层的概率分布。
- NIL：一个表示跳跃表末尾的特殊数值。
- PROB：链层的概率分布。

```
public class SkipList

{
```

```
private int maxLevel;

private int level;

private SkipNode header;

private float probability;

private const int NIL = Int32.MaxValue;

private const int PROB = 0.5F;

}
```

SkipList 类的构造器包含两个部分：带单独一个参数的 Public 构造器，其中此参数是跳跃表内节点的总数量，以及一个完成大部分工作的 Private 构造器数。在解释它们的工作原理之前，还是先来回顾一下方法：

```
private void SkipList2(float probable, int maxLevel)

{
```

```
this·probability = probable;

    this·maxLevel = maxLevel;

    level = 0;

    header = new SkipNode(maxLevel, 0, null);

    SkipNode nilElement = new SkipNode(maxLevel, NIL, null);

    for (int i = 0; i <= maxLevel - 1; i++)

        header·link[i] = nilElement;

}

public SkipList(long maxNodes)

{

    this·SkipList2(PROB, (int)(Math·Ceiling(Math·Log(maxNodes) / Math·Log(1 /
PROB) - 1)));
```

```
}
```

Public 构造器完成两项任务。第一项任务，把节点总数量传递给构造器方法作为方法内唯一的参数。第二项任务，实际执行初始化跳跃表对象工作的 Private 构造器在调用时会有两个参数。第一个参数是已经介绍过的概率常量。第二个参数也是已经讲过的确定跳跃表链层最大数量的公式。

Private 构造器体内设置了数据成员的数值，创建了跳跃表的头节点，创建了用于每个头节点链的“空”节点，以及初始化了该元素的链。

对跳跃表首先要做事就是在表中插入节点。下面的代码是 SkipList 类的 Insert 方法：

```
public void Insert(int key, Object value)

{

    SkipNode[] update = new SkipNode[maxLevel];

    SkipNode cursor = header;

    for (int i = level; i >= level; i--)

    {

        while (cursor·link[i]·key < key)
```

```
            cursor = cursor->link[i];

        update[i] = cursor;

    }

    cursor = cursor->link[0];

    if (cursor->key == key)

        cursor->value = value;

    else

    {

        int newLevel = GenRandomLevel();

        if (newLevel > level)

        {
```

```
            for (int i = level + 1; i <= newLevel - 1; i++)

                update[i] = header;

            level = newLevel;

        }

        cursor = new SkipNode(newLevel, key, value);

        for (int i = 0; i <= newLevel - 1; i++)

        {

            cursor·link[i] = update[i]·link[i];

            update[i]·link[i] = cursor;

        }

    }
```

```
}
```

这个方法首先做的事就是确定新的 SkipNode 在表中插入的位置(第一个 for 循环)。接下来,要检查表以便确认要插入的数值是否已经存在了。如果没有，那么就利用 Private 的 GenRandomLevel 方法把新的 SkipNode 分配到某个随机链层上（这个方法将在后续内容中讲到），并且把数据项插入到表内（在最后一个 for 循环之前的一行）。

利用概率方法 GenRandomLevel 就能确定链层。代码如下所示：

```
private int GenRandomLevel()

{

    int newLevel = 0;

    Random r = new Random();

int ran = r·Next(0);

int ran = Random·Next(0);

    while ((newLevel < maxLevel) && (ran < probability))

        newLevel++;
```

```
    return newLevel;

}
```

在介绍本小节重点内容 Search 方法之前，先来看看如何在跳跃表中执行删除操作。首先一起回顾一下 Delete 方法的代码：

```
public void Delete(int key)

{

    SkipNode[] update = new SkipNode[maxLevel + 1];

  SkipNode cursor = header;

    for (int i = level; i >= level; i--)

    {

        while (cursor·link[i]·key < key)

            cursor = cursor·link[i];
```

```
        update[i] = cursor;

    }

    cursor = cursor->link[0];

    if (cursor->key == key)

    {

        for (int i = 0; i < level - 1; i++)

        if (update[i]->link[i] == cursor)

                update[i]->link[i] = cursor->link[i];

        while ((level > 0) && (header->link[level]->key == NIL))

            level--;

    }
```

```
}
```

如同 Insert 方法一样，这个方法也分为两个部分。第一部分就是第一个 for 循环突显出来的内容，它会找到表中要删除的数据项。第二部分是 if 语句突出表示的内容，它会调整删除的 SkipNode 周围的链并且重新调整层次。

现在准备来讨论 Search 方法了。这个方法首先从最高层开始，接着是其他链直到找到一个关键字的数值大于要查找的关键字为止。然后，方法会降到下一个最低层继续查找，直到找到一个较高的关键字为止。随后再次降低层次并且继续查找。这个方法最终会停止在第 0 层，远离问题中数据项的一个正确的节点。代码如下所示：

```
public Object Search(int key)

{

    SkipNode cursor = header;

    for (int i = level; i > 0<= level - 1; i--)

    {

        SkipNode nextElement = cursor·link[i];

         while (nextElement·key < key)
```

```
        {

            cursor = nextElement;

             nextElement = cursor·link[i];

        }

    }

    cursor = cursor·link[0];

    if (cursor·key == key)

        return cursor·value;

    else

        return "Object not found";

}
```

现在已经为实现 SkipList 类提供了足够的功能。在本章末尾的练习中读者将会有机会编写代

码来使用这个类。

跳跃表提供了一种基于树结构的可替换选择。大多数程序员发现它们更容易实现且效率和类树结构相当。如果在处理一个完全或几乎有序的数据集合，那么跳跃表可能是比树更好的一种选择。

小结

本章中讨论的高级数据结构都是基于 Weiss *(1999* 年*)*书中第 *12* 章的内容。AVL 树和红黑树在使用二叉搜索树处理相当有序的数据的情况下为平衡问题提供了很好的解决方案。AVL 树和红黑树的主要缺点是重新平衡操作会伴随着相当数量的花费，而且可能会在大量数据集合方面降低性能。

针对极为大量的数据集合而言，跳跃表提供了相对于 AVL 树和红黑树的一种同等替换选择。因为跳跃表使用链表结构而不是树结构，所以不需要进行重新平衡的操作，这使得跳跃表在许多情况下显得更加高效。

练习

1· 请为 AVLTree 类编写 FindMin 方法和 FindMax 方法。

2· 利用 Timing 类对练习 *1* 中实现的方法与 BinarySerachTree 类中相同的方法进行次数比较。测试程序应该把大约由 *100* 个随机生成的整数组成的一个有序列表插入到两棵树中。

3· 请为 AVLTree 类编写删除方法来实现懒惰删除技术。有几种方法可以采用，但是一种简单的方法是为 Node 类简单添加一个布尔域，此布尔域会表示出节点是否为删除做

了标记。然后其他方法就应该考虑到这个域。

4· 请为遵循红黑树规则的 RedBlack 类编写删除方法。

5· 请设计并实现一个程序把 AVL 树和红黑树与跳跃表进行比较。请问哪一种数据结构的执行最好呢？

第 *16* 章 图和图的算法

虽然数学家们和其他一些科学家对网络一直研究了数百年，但是对网络的研究仍旧是这个新纪元最重要的热门科学之一。在计算机技术方面（比如说互联网）以及社会理论方面（社会网络，在《六度分隔》中经常被提及的概念）的最新发展都是网络研究领域的闪光点。

本章将会介绍如何用图来模拟网络。这里不会讨论诸如饼状图或条形图这类图形。我们所定义的图是指如何在 VB·Net 中表示图。此外，这里还会讨论在处理图时选取正确的数据表示的重要性，因为图算法的效率就是基于所用的数据结构。

16·1 图的定义

图是由一组顶点和一组边构成的。想象一下国家地图。每座城镇通过一些类型的道路与其他城镇连接在一起。地图是典型的图。每座城镇就是一个顶点，而连接两座城镇的路就是一条边。边用对*(v1,v2)*来表示，其中 v*1* 和 v*2* 是图中的两个顶点。顶点还可以有权值，有些时候也被称为代价。

对有序的图被称为有向图（directed graph），或者就叫有向图*(*digraph*)*。有序图如图 *16-1*

所示。如果图是无序的，那么它就被称为无序图（unordered graph），或者就称为图。图 16-2 显示了无序图的一个实例。

（原书 P284 页 图 1）

图 16-1 （有向）图

（原书 P284 页 图 2）

图 16-2 无序图

路径（path）是图中顶点的序列，所有的顶点由边连接在一起。路径的长度是从路径中第一个顶点到最后一个顶点的边的数量。路径也可以由一个顶点到其自身组成，这被称为环（loop）。环的长度为 0。

回路（cycle）是指在有向图中路径至少为 1 以便于初始定点也是结束定点。 在有向图中，

边可能是相同的，但是在无向图中，边必须是不同的。

如果存在从任意顶点到其他任意顶点的路径，就认为无向图是连通的（connected）。在有向图中，这个条件被称为是强连通（strongly connected）。如果有向图不是强连通的，但是又认为连通了，这就被称为弱连通（weakly connected）。如果图在每组顶点之间都有边，那么它就被称作是完全图（complete graph）。

16·2 由图模拟真实世界系统

图用来模拟许多不同类型的现实世界系统。交通流量就是其中一个实例。顶点表示街道的十字路口，同时边表示街道本身。加权边可以用来表示车速限制或者车道数量。模型可以使用系统来确定最佳路线和可能遭受交通堵塞的街道。

任何类型的运输系统都可以用图来模拟。例如，航空公司可以用图来模拟他们的飞行系统。每一个飞机场就是一个顶点，而从一个顶点到另一个顶点的航线就是一条边。加权的边可以表示从一个机场到另一个机场飞行的费用，或者表示从一个机场到另一个机场的大概距离，这取决于模拟的内容。

16·3 图类

乍看上去图很像树，而且人们可能会试图像树那样构造图类。然而，使用基于引用的实现会有问题，所以大家将会看到一种不同的方案来表示顶点和边。

16·3·1 顶点的表示

要开始构造 Graph 类的第一步是构造存储图内顶点的 Vertex 类。这个类与 LinkedList 类和 BinarySearchTree 类中的 Node 类具有相同的功效。

Vertex 类需要两个数据成员：一个用来识别顶点数据，而另外一个布尔型成员则用来跟踪顶点的“访问”。这两种数据成员分别命名为 label 和 wasVisited。

此类需要的唯一方法就是允许设置数据成员 label 和 wasVisited 的构造器方法。在这个实现中将不使用默认的构造器，这是因为每次开始引用顶点对象时都会进行一次实例化操作。

Vertex 类的代码如下所示：

```
public class Vertex

{

    public bool wasVisited;

    public string label;

    public Vertex(string label)

    {

        this.label = label;

        wasVisited = false;
```

```
    }

}
```

顶点列表会存储在数组内，而且在 Graph 类中会通过它们在数组内的位置对其进行引用。

16·3·2 边的表示

既然边描述了图的结构，所以关于图的真实信息是存储在边内的。正如先前提到的那样，试图像二叉树一样表示图是错误的。二叉树拥有十分固定的表示，因为父节点只能有两个子节点，而图的结构比这复杂得多。例如，可能有许多条边连接到一个单独的顶点上，也可能只有一条边连接到顶点上。

选择用来表示图中边的方法被称为是邻接矩阵。这是一个二维数组，数组内的元素表示了两个顶点之间是否存在边。图 *16-3* 举例说明了邻接矩阵是如何处理图的。

（原书 P*286* 页 图）

图 *16-3* 邻接矩阵

这里把顶点作为矩阵内行和列的标头罗列出来。如果在两个顶点之间存在一条边，那么就把 *1* 放在这个位置上。如果边不存在，那么就赋值为 *0*。很显然这里也可以使用布尔型的数值。

16·3·3 图的构造

现在有了表示顶点和边的方法，接下来就准备构造图了。首先，需要建立一个图中顶点的列表。下面的代码是有关一个拥有四个顶点的小图：

```
int nVertices = 0;

vertices[nVertices] = new Vertex("A");

nVertices++;

vertices[nVertices] = new Vertex("B");

nVertices++;

vertices[nVertices] = new Vertex("C");

nVertices++;

vertices[nVertices] = new Vertex("D");
```

然后，需要添加连接顶点的边。下面的代码用来添加两条边：

```
adjMatrix[0,1] = 1;

adjMatrix[1,0] = 1;

adjMatrix[1,3] = 1;

adjMatrix[3,1] = 1;
```

这段代码说明在顶点 A 和 B 之间存在一条边，而且在顶点 B 和 D 也存在一条边。

准备好上述这些内容，就准备来看看 Graph 类的初步定义了（包括 Vertex 类的描述）：

```
public class Vertex

{

    public bool wasVisited;

    public string label;

    public Vertex(string label)
```

```
    {

        this.label = label;

        wasVisited = false;

    }

}

public class Graph

{

    private int NUM_VERTICES = 6;

    private Vertex[] vertices;

    private int[,] adjMatrix;

    int numVerts;

    public Graph(int numvertices)
```

```
{
    NUM_VERTICES = numvertices;
    vertices = new Vertex[NUM_VERTICES];
    adjMatrix = new int[NUM_VERTICES, NUM_VERTICES];
    numVerts = 0;
    for (int j = 0; j <= NUM_VERTICES -1; j++)
        for (int k = 0; k <= NUM_VERTICES - 1; k++)
            adjMatrix[j, k] = 0;
}
public void AddVertex(string label)
{
```

```
        vertices[numVerts] = new Vertex(label);

        numVerts++;

    }

    public void AddEdge(int start, int eend)

    {

        adjMatrix[start, eend] = 1;

    }

    public void ShowVertex(int v)

    {

        Console·Write(vertices[v]·label + " ");

}
```

}

构造器方法重新构建了顶点数组和在常量 NUM-VERTICES 中指定数值的邻接矩阵。既然数组是基于零的，所以数据成员 numVerts 存储着顶点列表内当前的数量以便于把列表初始设置为 *0*。

AddVertex 方法会为顶点标签取走一个字符串参数，实例化一个新的 Vertex 对象，并且把它添加到顶点数组内。AddEdge 方法则会取走两个整型值参数。这些整数表示顶点，并且说明在它们之间存在一条边。最后，showVertex 方法会显示出指定顶点的标签。

16·3·4 图的第一个应用：拓扑排序

拓扑排序会把有向图中的顶点序列按照指定顺序显示出来。。一名大学生为取得学位而需要选修的课程序列就可以模拟成一张有向图。学生必须在完成了最初的两门计算机科学导论课后才可以学习数据结构课，图 *16-4* 描绘了模拟部分典型计算机科学课程的有向图。

（原书 P*289* 页 图）

图 *16-4* 模拟计算机科学课程序列的有向图

①、汇编语言

②、数据结构

③、操作系统

④、算法

此图的拓扑排序结果如下所示：

1. CS1

2. CS2

3. 汇编语言

4. 数据结构

5. 操作系统

6. 算法

其中课程 3 和课程 4 可以同时进行，同样的，课程 5 和课程 6 也可以同步进行。

16·3·5 拓扑排序算法

拓扑排序的基本算法是非常简单的：

1· 找到一个没有后继顶点的顶点。

2· 把此顶点添加到顶点列表内。

3· 从图中移除掉此顶点。

4· 重复步骤 1 直到把所有顶点从图中移除掉。

当然，在实现的细节上存在挑战，但这正是拓扑排序的关键所在。

算法实际上会从有向图的末尾处执行到开始处。再回头看看图 16-4。假设操作系统和算法是图内最后的顶点（忽略省略号），既然两者都没有后继顶点，所以把它们添加到列表内并且从图中删除。下面就轮到汇编语言和数据结构了。这两个顶点现在也没有后继顶点了，所以也把它们从图中删除。接下来就是 C2 了。既然它的后继顶点已经被删除，所以可以把它添加到列表内。最后就剩下 CS1 了。

16·3·6 拓扑排序算法的实现

拓扑排序需要两个方法。一个方法用来确定顶点是否有后继顶点，而另一方法则是把顶点从图中删除。下面先来看看确定没有后继顶点的方法。

在邻接矩阵中可以找到没有后继的顶点，这种顶点所在行对应的所有列都为零。方法会用嵌套的 for 循环来逐行检查每组列的内容。如果在某列发现 *1*，那么就跳出内部循环，并对下一行进行检查。如果找到一行对应的所有列都为零，那么返回这个行号。如果两层循环结束且没有行号返回，那么返回*-1*，这表示不存在无后继的顶点。代码如下所示：

```
public int NoSuccessors()

{

    bool isEdge;

    for (int row = 0; row <= NUM_VERTICES - 1; row++)

    {

        isEdge = false;

        for (int col = 0; col <= NUM_VERTICES - 1; col++)
```

```
    {

        if (adjMatrix[row, col] > 0)

        {

            isEdge = true;

            break;

        }

    }

    if (!isEdge)

        return row;

}

return -1;
```

```
}
```

接下来需要明白如何从图中移除顶点。需要做的第一件事就是从顶点列表中移除掉该顶点。这是很容易的。然后，就需要从邻接矩阵中移除掉相应的行和列，同时还要把移除行上面的行向下移动并且把移除列右侧的列向左移动，以此来填充移除顶点留下的行和列的空白。

为了实现这个操作，这里编写了名为 delVertex 的方法，它包括两个助手方法 moveRow 和 moveCol。代码如下所示：

```
    public void DelVertex(int vert)

    {

        if (vert != NUM_VERTICES - 1)

        {

            for (int j = vert; j < NUM_VERTICES - 1; j++)

                vertices[j] = vertices[j + 1];

            for (int row = vert; row < NUM_VERTICES - 1; row++)

                MoveRow(row, NUM_VERTICES);
```

```
            for (int col = vert; col < NUM_VERTICES - 1; col++)

                MoveCol(col, NUM_VERTICES);

        }

        NUM_VERTICES--;

}

private void MoveRow(int row, int length)

{

        for (int col = 0; col <= length - 1; col++)

            adjMatrix[row, col] = adjMatrix[row + 1, col];

}

private void MoveCol(int col, int length)
```

```
    {

        for (int row = 0; row <= length - 1; row++)

            adjMatrix[row, col] = adjMatrix[row, col + 1];

}
```

现在需要一个方法来控制排序的过程。先显示代码，接着再解释它的内容：

```
public void TopSort()

{

    Stack<string> gStack = new Stack<string>();

    while (NUM_VERTICES > 0)

    {

        int currVertex = NoSuccessors();
```

```
        if (currVertex == -1)

        {

            Console·WriteLine("Error: graph has cycles·");

            return;

        }

        gStack·Push(vertices[currVertex]·label);

        DelVertex(currVertex);

    }

    Console·Write("Topological sorting order: ");

    while (gStack·Count > 0)

        Console·Write(gStack·Pop() + " ");
```

```
}
```

TopSort 方法循环遍历图内顶点，找到一个无后继的顶点就把它删除，然后再移动到下一个顶点上。每次删除顶点时，会把它的标签压进一个栈内。栈是一种使用便利的数据结构，因为找到第一个顶点实际上就是图内的最后一个顶点（或者是最后中的一个）。当 TopSort 方法运行完成的时候，栈内的内容将包括压入栈底的最后一个顶点和在栈顶的图的第一个顶点。这时只需要循环遍历栈来弹出每个元素进行显示就是图的正确拓扑顺序了。

需要在有向图上执行拓扑排序的所有方法都有了。下面这个程序测试了编写的实现：

```
static void Main(string[] args)

{

    Graph theGraph = new Graph(4);

    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·TopSort();

        Console·WriteLine();

Console·WriteLine("Finished·");

}
```

此程序输出显示的图的顺序是 A B C D。

现在来看一看如何编写一个程序来对图 16-4 所示的图进行排序：

```
static void Main(string[] args)

{

Graph theGraph = new Graph(6);
```

```
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·");

}
```

此程序的输出是：

（原书 P293 页 截图）

16·4 图的搜索

确定从一个顶点能到达哪些顶点是在图上经常执行的一种操作。人们可能需要知道在地图上哪些路可以从一个城镇到达其他城镇，或者从一个机场到其他机场可以走哪条航线。

在图上执行这些操作都用到了查找算法。图上可以执行两种基础查找：深度优先（depth-first）搜索和广度优先（breadth-first）搜索。本小节会研究这两种算法。

16·4·1 深度优先搜索

深度优先搜索的含义是沿着一条路径从开始顶点到达最后的顶点，然后原路返回，并且沿着下一条路径达到最后的顶点，如此继续直到走过所有路径。深度优先搜索的图如图 16-5 所示。

（原书 P294 页 图）

图 16-5 深度优先搜索

从高层次而言，深度优先搜索算法的工作大致如此：首先，选取一个起始点，它可能是任何顶点。访问这个顶点，把它压入一个栈内，并且标记为已访问的。接着转到下一个未访问的顶点，也把它压入栈内，并且做好标记。继续这样的操作直到到达最后一个顶点为止。然后，检查栈顶的顶点是否还有其他未访问的相邻顶点。如果没有，就把它从栈内弹出，并且检查下一个顶点。如果找到一个这样的顶点，那么就开始访问相邻顶点直到没有未访问的为止，还要检查更多未访问的相邻顶点并且继续此过程。当最终到达栈内最后一个顶点并且没有相邻的未访问顶点的时候，才算完成深度优先搜索。

需要开发的的第一部分代码是一种获得未访问邻接矩阵的方法。程序必须首先到达指定顶点的所在行并且确定其对应的某一列中是否存储着数值 1。如果有，那么就存在邻接顶点。然后就可以很容易地判定此顶点是否已经被访问过。这个方法的代码如下所示：

```
private int GetAdjUnvisitedVertex(int v)
```

```
{

    for (int j = 0; j <= NUM_VERTICES - 1; j++)

        if ( (adjMatrix[v,j] == 1) && (vertices[j].wasVisited == false))

        return j;

    return -1;

}
```

现在就准备来看一看执行深度优先搜索的方法：

```
public void DepthFirstSearch()

{

    Stack<int> gStack = new Stack<int>();

    vertices[0].wasVisited = true;
```

```
ShowVertex(0);

gStack·Push(0);

int v;

while (gStack·Count > 0)

{

    v = GetAdjUnvisitedVertex(gStack·Peek());

    if (v == -1)

        gStack·Pop();

    else

    {

        vertices[v]·wasVisited = true;

        ShowVertex(v);
```

```
            gStack·Push(v);

        }

    }

    for (int j = 0; j <= NUM_VERTICES - 1; j++)

        vertices[j]·wasVisited = false;

}
```

下面这个程序对图 16-5 中的图进行了深度优先搜索：

```
static void Main(string[] args)

{

    Graph aGraph = new Graph(13);

    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();

}
```

程序的输出结果如下所示：

（原书P296页 截图）

16·4·2 广度优先搜索

广度优先搜索算法会从第一个顶点开始尝试访问所有可能在第一个顶点附近的顶点。从本质上说，这种搜索在图上的移动是逐层进行的，首先会检查与第一个顶点相邻的层，然后逐步向下检查远离初始顶点的层。图16-6举例说明了广度优先搜索的工作原理。

（原书P297页 图）

图16-6 广度优先搜索

尽管仍可以使用栈，但是广度优先搜索算法却用队列来代替栈。算法如下所示：

1. 找到一个与当前顶点相邻的未访问过的顶点，把它标记为已访问的，然后把它添加到队列中。

2. 如果找不到一个未访问过的相邻顶点，那么从队列中移除掉一个顶点（只要队列中有顶点可以移除掉），把它作为当前顶点，然后重新开始。

3. 如果由于队列为空而无法执行第二步操作，那么此算法就此结束。

现在就来看看此算法的代码：

```
public void BreadthFirstSearch()

{
```

```
Queue<int> gQueue = new Queue<int>();

vertices[0].wasVisited = true;

ShowVertex(0);

gQueue.Enqueue(0);

int vert1, vert2;

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);

        }

    }

    for (int i = 0; i <= NUM_VERTICES - 1; i++)

        vertices[i].wasVisited = false;

}
```

注意在这个方法中有两个循环。当队列有数据时运行外层循环，并且内循环会检查相邻的顶点是否已经被访问过。for 循环会为其他方法简单地清理顶点数组。

这里有一个程序对图 16-6 中的图进行了测试，内容如下所示：

```
static void Main(string[] args)

{

    Graph aGraph = new Graph(13);

    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();

}
```

此程序的输出结果是：

（原书P*299*页 截图）

16·5 最小生成树

当首次设计网络的时候，网络节点之间的连接数量很可能会多于最小连接数量。额外的连接是一种资源浪费，应该尽可能地消除它。额外的连接也会使其他人对网络的研究和理解变得不必要的复杂。因此需要使得网络只包含对节点连接而言最小数量的必要连接。当把这种网络应用到图上的时候，这样的网络就被称为是最小生成树。

最小生成树的得名源于覆盖每个顶点（范围）所必需的最少数量的构造边，而且说它是树是因为结果图是非循环的。需要牢记一个重要的内容：一张图可能包含多个最小生成树。创建的最小生成树完全依赖于初始顶点。

16·5·1 最小生成树算法

图*16-7*描述了一张想要构造最小生成树的图。

（原书P*300*页 图）

图*16-7* 用于最小生成树的图

最小生成树算法实际上就是一种图的搜索算法（既可以是深度优先也可以是广度优先），它包含了记录着遍历过的每条边的额外成分。代码看上去也十分相似。方法如下所示：

```
public void Mst()

{

    Stack<int> gStack = new Stack<int>();

    vertices[0].wasVisited = true;

    gStack.Push(0);

    int currVertex, ver;

    while (gStack.Count > 0)

    {

        currVertex = gStack.Peek();
```

```
        ver = GetAdjUnvisitedVertex(currVertex);

        if (ver == -1)

            gStack·Pop();

        else

        {

            vertices[ver]·wasVisited = true;

            gStack·Push(ver);

            ShowVertex(currVertex);

            ShowVertex(ver);

            Console·Write(" ");

        }

    }
```

```
    for (int j = 0; j <= NUM_VERTICES - 1; j++)

        vertices[j].wasVisited = false;

}
```

如果把这种方法与深度优先搜索的方法进行比较，就会发现它是通过调用 showVertex 方法来记录当前的顶点，其中方法的参数就是当前的节点。如代码所示，两次调用这个方法，就产生了用来定义最小生成树的边的显示。

下面的程序产生了图 16-7 中图的最小生成树：

```
static void Main(string[] args)

{

    Graph aGraph = new Graph(7);

    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();

}
```

此程序的输出结果是：

（原书P*301*页 截图）

图*16-8* 显示了最小生成树。

（原书P*302*页 图）

图*16-8* 图*16-7*的最小生成树

16.6 查找最短路径

在图上执行的最常见的操作之一就是查找从一个顶点到另一个顶点的最短路径。假期里，你计划在两周的时间内到*10*座主要篮球联赛城市观看比赛。你希望利用最短路径算法从而驾驶最短的里程到达所有这*10*座城市。另一个最短路径问题是在创建计算机网络的时候考虑两台计算机之间传输时间的代价，或者是建立及维护计算机之间连接的代价。最短路径算法可以确定构建网络的最有效方法。

16·6·1 加权图

在本章开头部分已经提过加权图。图中的每一条边都有相关的权值或代价。图 *16-9* 显示了一张加权图。加权图可以有负的权值，但是这里将只限于讨论正的权值。而且这里只关注有向图。

（原书 P*303* 页 图）

图 *16-9* 加权图

16·6·2 确定最短路径的 Dijkstra 算法

计算机科学中最著名的算法之一就是用来确定加权图最短路径的 Dijkstra 算法。此算法是以二十世纪五十年代后期发明者——近代计算机科学家 Edsger Dijkstra 的名字命名的。

Dijkstra 算法找到了从任意指定顶点到任何其他顶点的最短路径，而且证实可以到达图中的所有其他顶点。它使用了通常被称为贪心的策略或算法。贪心算法（关于此算法将会在第 *17* 章中详细介绍）把问题分解成小块或步骤，并且在每一步中确定最优解，用这些最优解合并生成最终的解。贪心算法的经典实例就是硬币找零问题。例如，假设你在商店用一美元购买 *74* 美分的商品，如果收银员（他或她）在用贪心算法来使得要找回的硬币数量最少，那么你会获得一个 *25* 美分和一个 *1* 美分。当然，还有其他方案可以凑足 *26* 美分，但是一

个 25 美分和一个 1 美分是这个问题的最优方案。

通过创建一张表来存储图中从起始顶点到其他顶点的已知距离就可以使用 Dijkstra 算法了。访问与原始顶点相邻的每个顶点，并且更新表中关于相邻边的权值的信息。如果已知两个顶点之间的距离，但是通过访问某个新的顶点发现了更短的距离，那么就更改表中的相应信息。通过指示生成最短路径的顶点也会对表进行更新。

下面的这些表格说明了在图上运行此算法的工作过程。第一张表显示了访问顶点 A 之前表内数值的情况（数值 Infinity 说明未知的距离，在代码中用一个很大的数值来表示未知权值）：

（原书 P304 页 图 1）

①、顶点；②、是否访问过；③、权值；④、经过路径

当访问顶点 A 之后，表格如下所示：

（原书 P304 页 图 2）

接下来访问顶点 D：

（原书 P*304* 页 图 *3*）

再下一个访问顶点 B：

（原书 P*305* 页 图 *1*）

如此继续直到访问最后一个顶点 G：

（原书 P*305* 页 图 *2*）

16·6·3 Dijkstra 算法的代码

这个算法第一部分的代码就是之前已经介绍过的 Vertex 类：

```
public class Vertex

{

    public string label;

    public bool isInTree;

    public Vertex(string lab)

    {

        label = lab;

        isInTree = false;

    }

}
```

当然，还需要一个类用来记录原始顶点与远距离顶点之间的关系，由此用来计算最短路径。
这个类叫做 DistOriginal 类：

```
public class DistOriginal

{

    public int distance;

    public int parentVert;

    public DistOriginal(int pv, int d)

    {

        distance = d;

        parentVert = pv;

    }

}
```

前面已经用到了 Graph 类，现在有一套新的方法用来计算最短路径。其中第一个方法是 Path()方法，它驱动了最短路径的计算：

```
public void Path()

{

    int startTree = 0;

    vertexList[startTree].isInTree = true;

    nTree = 1;

    for (int j = 0; j <= nVerts; j++)

    {

        int tempDist = adjMat[startTree, j];

        sPath[j] = new DistOriginal(startTree, tempDist);

    }
```

```
while (nTree < nVerts)

{

    int indexMin = GetMin();

    int minDist = sPath[indexMin].distance;

    currentVert = indexMin;

    startToCurrent = sPath[indexMin].distance;

    vertexList[currentVert].isInTree = true;

    nTree++;

    AdjustShortPath();

}

DisplayPaths();
```

```
    nTree = 0;

    for (int j = 0; j <= nVerts - 1; j++)

        vertexList[j].isInTree = false;

}
```

这个方法采用了两个帮助方法，即 getMin 方法和 adjustShortPath 方法。这里会对这两种方法进行简要地解释说明。在方法开始处的 for 循环会查看源自初始顶点的可到达顶点，并且把这些顶点放置到 sPath 数组内。这个数组保存着来自不同顶点的最小路径距离，而且最后会保存最终的最短路径。

主循环（while 循环）执行三个操作：

1. 找到 sPath 中具有最短路径的顶点。

2. 把此顶点设置为当前顶点。

3. 更新 sPath 数组来显示当前顶点的距离。

这个工作的大部分内容都是由 getMin 方法和 adjustShortPath 方法来完成的：

```
public int GetMin()
```

```
{
    int minDist = infinity;
    int indexMin = 0;
    for (int j = 1; j <= nVerts - 1; j++)
        if (!(vertexList[j].isInTree) && sPath[j].distance < minDist)
        {
            minDist = sPath[j].distance;
            indexMin = j;
        }
    return indexMin;
}
```

```
public void AdjustShortPath()

{

    int column = 1;

    while (column < nVerts)

        if (vertexList[column].isInTree)

            column++;

        else

        {

            int currentToFring = adjMat[currentVert, column];

            int startToFringe = startToCurrent + currentToFring;

            int sPathDist = sPath[column].distance;

            if (startToFringe < sPathDist)
```

```
            {

                sPath[column].parentVert = currentVert;

                sPath[column].distance = startToFringe;

            }

            column++;

        }

}
```

getMin 方法逐步遍历 sPath 数组直到确定了最小距离为止，然后方法会把此距离返回。adjustShortPath 方法会取一个新的顶点，找到与此顶点相连接的下一组顶点，计算出最短路径，再更新 sPath 数组直到发现一个更短距离为止。

最后，displayPaths 方法会把 sPath 数组内的最终结果显示出来。为了使图变量用于其他算法，要把 nTree 变量设置为 *0*，而把 isInTree 标记全部设为假的（false）。

为了把上述这些都放在内容里，这里有一个完整的应用程序包括了所有用 Dijkstra 算法计算最短路径的代码，此外还有一个程序用来测试它的实现：

```
using System;

using System.Collections.Generic;

public class DistOriginal

{

    public int distance;

    public int parentVert;

    public DistOriginal(int pv, int d)

    {

        distance = d;

        parentVert = pv;

    }

}
```

```
public class Vertex

{

    public string label;

    public bool isInTree;

    public Vertex(string lab)

    {

        label = lab;

        isInTree = false;

    }

}

public class Graph
```

```
{

    private const int max_verts = 20;

    int infinity = 1000000;

    Vertex[] vertexList;

    int[,] adjMat;

    int nVerts;

    int nTree;

    DistOriginal[] sPath;

    int currentVert;

    int startToCurrent;

    public Graph()
```

```
{

    vertexList = new Vertex[max_verts];

    adjMat = new int[max_verts, max_verts];

    nVerts = 0;

    nTree = 0;

    for (int j = 0; j <= max_verts - 1; j++)

        for (int k = 0; k <= max_verts - 1; k++)

            adjMat[j, k] = infinity;

    sPath = new DistOriginal[max_verts];

}

public void AddVertex(string lab)
```

```
{

    vertexList[nVerts] = new Vertex(lab);

    nVerts++;

}

public void AddEdge(int start, int theEnd, int weight)

{

    adjMat[start, theEnd] = weight;

}

public void Path()

{

    int startTree = 0;
```

```
vertexList[startTree].isInTree = true;

nTree = 1;

for (int j = 0; j <= nVerts; j++)

{

    int tempDist = adjMat[startTree, j];

    sPath[j] = new DistOriginal(startTree, tempDist);

}

while (nTree < nVerts)

{

    int indexMin = GetMin();

    int minDist = sPath[indexMin].distance;
```

```
            currentVert = indexMin;

            startToCurrent = sPath[indexMin].distance;

            vertexList[currentVert].isInTree = true;

            nTree++;

            AdjustShortPath();

        }

        DisplayPaths();

        nTree = 0;

        for (int j = 0; j <= nVerts - 1; j++)

            vertexList[j].isInTree = false;

    }
```

```
public int GetMin()

{

    int minDist = infinity;

    int indexMin = 0;

    for (int j = 1; j <= nVerts - 1; j++)

        if (!(vertexList[j].isInTree) && sPath[j].distance < minDist)

        {

            minDist = sPath[j].distance;

            indexMin = j;

        }

    return indexMin;
```

```
}

public void AdjustShortPath()

{

    int column = 1;

    while (column < nVerts)

        if (vertexList[column].isInTree)

            column++;

        else

        {

            int currentToFring = adjMat[currentVert, column];

            int startToFringe = startToCurrent + currentToFring;

            int sPathDist = sPath[column].distance;
```

```
            if (startToFringe < sPathDist)

            {

                sPath[column].parentVert = currentVert;

                sPath[column].distance = startToFringe;

            }

            column++;

        }

}

public void DisplayPaths()

{

    for (int j = 0; j <= nVerts - 1; j++)
```

```
    {

        Console·Write(vertexList[j]·label + "=");

        if (sPath[j]·distance == infinity)

            Console·Write("inf");

        else

            Console·Write(sPath[j]·distance);

        string parent = vertexList[sPath[j]·parentVert]·

        label;

        Console·Write("(" + parent + ") ");

    }

}
```

```
class chapter16

{

    static void Main()

    {

        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();

        }

    }

}
```

此程序输出结果是：

（原书 P312 页 截图）

小结

图是用于计算机科学领域的最重要的数据结构之一。人们有规律地把图用来模拟从电路到大学课表再到公路及航空路线的每一件事情。

图是由顶点及其连接的边组成的。有几种方法可以搜索图，但是最常见的是深度优先搜索和广度优先搜索。在图上进行的另外一种重要算法是确定最小生成树，最小生成树就是需要连接图中所有顶点的最少边数。

图的边可以有权值或者代价。当处理加权图的时候，一项重要的操作就是确定图中从一个起始顶点到其他顶点的最短路径。本章介绍了一种计算最短路径的算法，即 Dijkstra 算法。

Weiss *(1999* 年*)*的书包含许多本章介绍的有关图算法的技术讨论，而 LaFore *(1998* 年*)*的书则包括了本章涉及的所有算法的非常好的实践说明。

练习

39· 请构造一个加权图来模拟你家所在区域的地图。请使用 Dijkstra 算法来确定从一个起始顶点到最终顶点的最短路径。

40· 请把练习 *1* 中图内的权值拿走，并且构造一个最小生成树。

41· 还是继续使用练习 *1* 的图，请编写一个视窗应用程序，此程序允许用户利用深度

优先搜索或广度优先搜索来查找图上某一个顶点。

42· 请利用 Timing 类来确定练习 *3* 中的哪一种搜索实现更有效。

第 *17* 章 高级算法

本章会介绍两个高级主题：即动态规划和贪心算法。动态规划常被认为是递归的反向技术。所谓递归算法是从顶部开始，把问题向下全部分解为小的问题进行解决，直到解决整个问题为止。而动态规划则是从底部开始，解决小的问题同时把它们合并形成大问题的一个完整解决方案。

贪心算法是在求完整解的过程中寻找“好的解决方案”的一种算法。这些好的解决方案被称为是局部最优解，它们有希望得到最终的正确解，即所谓的全局最优解。“贪心”一词源于这些算法在某个时刻会选取看似最好的解决方案这一事实。贪心算法经常用在几乎不可能找到完整解的时候，由于时间与/或空间的考虑，所以也就接受非最优解了。

17·1 动态规划

解决问题的递归算法经常是很优雅的，但却是低效的。尽管可能是高雅的计算机程序，但是C#语言编译器以及其他语言编译器都不会把递归代码有效地翻译成机器代码，并最终导致效率低下。

许多用递归解决的编程问题可以用动态规划技术进行重新编写。动态规化通常用数组来构建一张表，此表会把不同的子解决方案存储起来。最后，当算法结束的时候，就会在表内显著的位置上找到答案。

17.1.1 动态规划实例：计算斐波纳契数列

斐波纳契数列可以表述成如下数字序列：0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .

这里有一个简单的递归程序可以用来产生上述序列中任意指定的数。。函数代码如下所示：

```
static long recurFib(int n)

{

if (n < 2)

        return n;

else

        return recurFib(n - 1) + recurFib(n - 2);
```

```
}
```

下面是一个使用了此函数的程序：

```
static void Main()

{

int num = 5;

long fibNumber = recurFib(num);

    Console·Write(fibNumber);

}
```

此函数的问题是效率极为低下。通过查看图 *17-1* 中的树就可以明确地知道这个递归算法的效率是多么的低下。

（原书 P*316* 页 图）

图 17-1 递归计算斐波纳契数列所形成的树

递归算法的问题在于递归过程中会重复计算太多的数值。如果编译器可以跟踪已经计算过的数值，那么这个函数就几乎不会如此效率低下了。利用动态规划技术来设计算法会比递归算法效率高许多。

使用动态规划技术设计算法从解决最简单的可解子问题开始入手，利用解决方案解决更加复杂的子问题直到解决整个问题为止。每个子问题的解决方案都代表性地存储在易访问的数组内。

通过研究用于计算斐波纳契数列的动态规划算法，人们可以很容易地理解动态规划的本质。下面的代码就解释说明了此算法的工作原理：

```
static long iterFib(int n)

{

    int[] val = new int[n];

    if ((n == 1) || (n == 2))

        return 1;

    else
```

```
    {

        val[1] = 1;

        val[2] = 2;

        for (int i = 3; i <= n - 1; i++)

            val[i] = val[i - 1] + val[i - 2];

    }

    return val[n - 1];

}
```

数组 val 是用来存储中间结果的。如果参数为 *1* 或者 *2*，那么 If 语句的第一部分就会返回数值 *1*。否则，就把数值 *1* 和 *2* 分别存储在索引为 *1* 和 *2* 的数组相应位置上。for 循环的运行范围是从 *3* 到输入的参数，循环每次都把前两个数组元素之和赋值给当前元素，并且在循环结束时，把数组内的最后一个数值返回。

下面来比较一下用递归算法和动态规划算法计算斐波纳契数列所花费的时间。首先，下面是用于比较的程序：

```
static void Main()

{

Timing tObj = new Timing();

    Timing tObj1 = new Timing();

    int num = 35;

    long fibNumber;

    tObj.startTime();

    fibNumber = recurFib(num);

    tObj.stopTime();

    Console.WriteLine("Calculating Fibonacci number: " + num);

    Console.WriteLine(fibNumber + " in: " + tObj.Result().TotalMilliseconds);
```

```
        tObj1.startTime();

        fibNumber = iterFib(num);

        tObj1.stopTime();

        Console.WriteLine(fibNumber + " in: " + tObj1.Result().TotalMilliseconds);

    }
```

如果运行此程序来测试两个函数计算小数值的斐波纳契数列，那么会看到极小的差异，甚至看似递归函数会略微快些。(此图的显示与上下文不是很对应，数据没有显示出差异？)

（原书 P317 页 截图）

如果尝试一个较大的数，比如 20，就会得到如下结果：

（原书P318页 截图1）

而对于一个确实很大的数，比如35，结果差异甚至是更加明显：

（原书P318页 截图2）

这是一个有关动态规划如何辅助改进算法性能的经典实例。正如先前提到的那样，使用动态规划技术的程序通常利用数组来存储中间计算结果，但是需要指出的是在某些情况下数组不是必需的，比如斐波纳契数列。下面编写的 iterFib 函数就没有使用数组：

```
static long iterFib1(int n)

{

    long last, nextLast, result;

    last = 1;
```

```
nextLast = 1;

    result = 1;

    for (int i = 2; i <= n - 1; i++)

    {

        result = last + nextLast;

        nextLast = last;

        last = result;

    }

    return result;

}
```

iterFib 函数和 iterFib1 函数都可以在几乎相同的时间内计算出斐波纳契数列。

17.1.2 寻找最长公共子串

另外一个适用动态规划算法解决的问题是在两个字符串中寻找到最长公共子串。例如，在两个词“raven”和“havoc”中，最长的公共子串是“av”。

首先来看一下这个问题的暴力穷举解决方案。假设有两个字符串 A 和 B，从字符串 A 的第一个字符开始与字符串 B 中的字符逐个进行比较就可以找到最长的公共子串。当发现没有匹配的时候，就移动到字符串 A 的下一个字符，与字符串 B 的第一个字符重新开始比较，如此类推。

使用动态规划算法会是一种更好的解决方案。算法使用二维数组来存储两个字符串内相同位置上字符比较的结果。初始数组的每个元素都设置为 0。每次在两个数组的相同位置上找到匹配，数组对应行和列上的元素就自动加 1，否则这个元素设置为 0。

为了复制最长公共子串，从数组的下一行一直到数组的最后一行做第二次遍历检查，而且数值大于 0 的列项对应子串内的一个字符。如果没有找到公共子串，那么数组的所有元素都是 0。

下面是一个寻找最长公共子串的完整程序：

```
using System;

class chapter17
```

```
{

    static void LCSubstring(string word1, string word2, string[] warr1, string[] warr2,
int[,] arr)

    {

        int len1, len2;

        len1 = word1.Length;

        len2 = word2.Length;

        for (int k = 0; k <= word1.Length - 1; k++)

        {

            warr1[k] = word1.Substring(k, 1);

            warr2[k] = word2.Substring(k, 1);
```

```
    }

    for (int i = len1 - 1; i >= 0; i--)

    {

        for (int j = len2 - 1; j >= 0; j--)

            if (warr1[i] == warr2[j])

                arr[i, j] = 1 + arr[i + 1, j + 1];

            else

                arr[i, j] = 0;

    }

}

static string ShowString(int[,] arr, string[] wordArr)

{
```

```
    string substr = "";

    for (int i = 0; i <= arr.GetUpperBound(0); i++)

        for (int j = 0; j <= arr.GetUpperBound(1); j++)

            if (arr[i, j] > 0)

                substr += wordArr[j];

    return substr;

}

static void DispArray(int[,] arr)

{

    for (int row = 0; row <= arr.GetUpperBound(0); row++)

    {
```

```
            for (int col = 0; col <= arr·GetUpperBound(1); col++)

                Console·Write(arr[row, col]);

            Console·WriteLine();

        }

}

static void Main()

{

    string word1 = "mavens";

    string word2 = "hpavoc";

    string[] warray1 = new string[word1·Length];

    string[] warray2 = new string[word2·Length];
```

```
string substr;

int[,] larray = new int[word1.Length, word2.Length];

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.Length > 0)

    Console.WriteLine("The longest common substring is: " + substr);

else
```

```
            Console·WriteLine("There is no common substring");

    }

}
```

函数 LCSubstring 完成了构造二维数组的工作，此数组存储着确定最长公共子串的数值。第一个 for 循环简单地把两个字符串放入数组。而第二个 for 循环则执行比较操作，并且构建了数组。

函数 ShowString 检测构建在 LCSubstring 内的数组，查看是否有任何元素的数值大于 0，如果找到这样的数值，那么就返回其中一个字符串的相应字母。

子程序 DispArray 显示了数组的内容。在运行前面的程序时，此子程序用来检测由 LCSubstring 构建的数组：

（原书 P321 页 截图 1）

存储在数组内的编码说明字符串“maven”和字符串“havoc”中的第 2 个和第 3 个字符组成了两者的最长公共子串。下面是另外一个实例：

（原书P321页 截图2）

很明显这两个字符串没有公共子串，所以数组的所有元素都为0。

17.1.3 背包问题

背包问题是算法学习中一个经典问题。假设你是一名保险箱窃贼，你打开了一个装满各种财宝的保险箱，但是你只能用一个小背包带走财宝。保险箱内的财宝在大小和价值上各不相同。你希望最大程度地用最值钱的财宝装满背包。

当然，暴力穷举可以解决这个问题，但是用动态规划会更为高效。用动态规划解决背包问题的关键在于计算不超过背包总容量的最大价值。关于背包问题清晰简洁的解释请参阅Sedgewick（1990年，书页从596到598）的书。本节的实例就是基于这本书的内容。

假设前面讨论的实例中保险箱有5件宝物，大小分别是3、4、7、8和9，其价值分别是4、5、10、11、13，而且背包的容量为16，那么正确的解决方法是取第3和第5个宝物，它们的尺寸之和是16，总价值是23。

解决此问题的代码十分简短，但是没有整个程序内容的代码是没有意义的，所以下面来看看解决背包问题的程序：

```
using System;

class chapter17

{

    static void Main()

    {

        int capacity = 16;

        int[] size = new int[] { 3, 4, 7, 8, 9 };

        int[] values = new int[] { 4, 5, 10, 11, 13 };

        int[] totval = new int[capacity+1];

        int[] best = new int[capacity+1];

        int n = values·Length;
```

```
        for (int j = 0; j <= n - 1; j++)

            for (int i = 0; i <= capacity; i++)

                if (i >= size[j])

                    if (totval[i] < (totval[i - size[j]] + values[j]))

                    {

                        totval[i] = totval[i - size[j]] + values[j];

                        best[i] = j;

                    }

        Console·WriteLine("The maximum value is: " + totval[capacity]);

    }

}
```

保险箱内的宝物用尺寸数组和价值数组来模拟。当算法处理不同宝物的时候，数组 totval 用来存储用最高的总价值。数组 best 存储着具有最高价值的宝物。当完成算法的时候，最高总价值就在数组 totval 的最后位置上，而下一个最高价值则在次后的位置上，以此类推。数组 best 存储情况相同。具有最高价值的宝物存储在数组 best 的最后一个元素内，具有次高价值的宝物则存储在次后的位置上，以此类推。

算法的核心是在嵌套 for 循环中的第 2 个 if 语句。把当前最好的总价值与添加给背包的下一个宝物的总价值进行比较。如果当前最好的总价值较大，就什么操作也不做。否则，就把这个新的总价值作为最好的当前总价值添加到数组 totval 内，并且把该项的索引加到数组 best 内。下面还是这段代码：

```
if (totval[i] < (totval[i - size[j]] + values[j]))

{

totval[i] = totval[i - size[j]] + values[j];

best[i] = j;

}
```

如果想要查看产生总价值的数据项，可以在 best 数组内进行检测：

```
Console·WriteLine("The items that generate this value are: ");

int totcap = 0;
```

```
while (totcap <= capacity)

{

    Console·WriteLine("Item with best value: " + size[best[capacity - totcap]]);

    totcap += size[best[i]];

}
```

记住，产生先前所有最好价值的数据项都存储在数组内，所以向后遍历数组 best，并且返回数据项直到它们的大小之和等于背包的总容量时停止。

17·2 贪心算法

前面小节讨论了经常用基于递归的低效算法解决的问题可以用动态规划进行方案优化。但是对于许多问题而言，求助于动态规划是过分不必要的，较为简单的算法就足够了。

其中一种较为简单的典型算法就是贪心算法。贪心算法在某一时刻始终都会选择最好的解决方案，而不会考虑这种选择对未来选择的影响。使用贪心算法通常预示着实现者希望一系列“最好的”局部选择会导致最终“最好的”选择。如果是这样，那么算法就会产生一个最优解。否则，会得到一个非最优解。然而，对于很多问题而言，不值得费心找到最优解，所以使用贪心算法就足以了。

17·2·1 贪心算法实例：找零钱问题

贪心算法的经典实例就是找零钱问题。假设你在商店里买了一些商品，在付账时要找回零钱 63 美分，收银员会如何找零钱给你呢？如果收银员使用贪心算法，那么他或她会给你两个 25 美分、一个 10 美分以及三个 1 美分。这是凑足 63 美分的最少硬币数目（假设不允许用 50 美分的硬币）。

这就证明了找零钱问题的最优解始终可以用当前美国硬币的面值来找到。然而，如果引入一些其他面值来混合，那么贪心算法就无法得到最优解了。

下面的程序用贪心算法来找零钱（此代码假设找回的零钱少于一美元）：

```
using System;

class chapter17

{

    static void MakeChange(double origAmount, double remainAmount, int[] coins)

    {

        if ((origAmount % 0·25) < origAmount)
```

```
{

    coins[3] = (int)(origAmount / 0.25);

    remainAmount = origAmount % 0.25;

    origAmount = remainAmount;

}

if ((origAmount % 0.1) < origAmount)

{

    coins[2] = (int)(origAmount / 0.1);

    remainAmount = origAmount % 0.1;

    origAmount = remainAmount;

}
```

```
if ((origAmount % 0.05) < origAmount)

{

    coins[1] = (int)(origAmount / 0.05);

    remainAmount = origAmount % 0.05;

    origAmount = remainAmount;

}

if ((origAmount % 0.01) < origAmount)

{

    coins[0] = (int)(origAmount / 0.01);

    remainAmount = origAmount % 0.01;

}
```

```
}

static void ShowChange(int[] arr)

{

    if (arr[3] > 0)

        Console.WriteLine("Number of quarters: " + arr[3]);

    if (arr[2] > 0)

        Console.WriteLine("Number of dimes: " + arr[2]);

    if (arr[1] > 0)

        Console.WriteLine("Number of nickels: " + arr[1]);

    if (arr[0] > 0)

        Console.WriteLine("Number of pennies: " + arr[0]);
```

```
}

static void Main()

{

    double origAmount = 0.63;

    double toChange = origAmount;

    double remainAmount = 0.0;

    int[] coins = new int[4];

    MakeChange(origAmount, remainAmount, coins);

    Console.WriteLine("The best way to change " + toChange + " cents is: ");

    ShowChange(coins);

}
```

```
}
```

子程序 MakeChange 首先从最高面值的硬币 25 美分开始，并且试图使用尽可能多的此种硬币。25 美分硬币的总数量存储在数组 coins 内。一旦原始数量少于 25 美分，那么算法就移动到 10 美分处，并再次试图使用尽可能多的 10 美分硬币。接着，算法会处理 5 美分的硬币，在下来是 1 美分的硬币，并且把使用每种硬币的总数量存储在数组 coins 内。下面就是此程序的一些输出：

（原书 P326 页 截图）

正如先前提到的那样，这个贪心算法利用标准的美国硬币面值始终会找到最优解。但是，如果出现了一种新的硬币，比如 22 美分，那么会发生什么事情呢？在练习部分，读者将有机会实践一下这个问题。

17·2·2 采用哈夫曼编码的数据压缩

数据压缩是计算实践领域的一项重要技术。通过互联网传输的数据需要尽可能紧密地传送。数据压缩有许多不同的方法，但是其中一种特殊的方法利用了贪心算法—哈夫曼编码。用哈夫曼编码压缩的数据可以节约 20%到 90%的空间储蓄。这种算法是根据它的发明者 David

Huffman 的名字命名的，他是一位信息理论学家和计算机科学家，他在 20 世纪 50 年代发明了此项技术。

当压缩数据的时候，通常会把组成数据的字符转换成一些其他表现以节省空间。一种典型的压缩方法是把每个字符转换成二进制字符代码或者位字符串。例如，把字符“a”编码成 000，字符“b”编码成 001，而字符“c”则变成 010，如此等等。这种方法被称为是固定长度编码。

但是，还可以用一种更好的方法，就是使用可变长度编码。既然某些字符会多次用到，所以这些最频繁出现在字符串内的字符具有较短的编码，而具有较低出现频率的字符则拥有较长的编码。编码的过程就是依据某个字符的出现频率把位字符串赋值给该字符的过程。哈夫曼编码算法会取走字符串，把它们转换成可变长度的二进制字符串，并且出于解码二进制字符串的目来创建一棵二叉树。把到达每个左叶子节点的路径设置为二进制字符 0，而把到达每个右叶子节点的路径设置为二进制字符 1。

算法工作过程如下：从要压缩的字符串开始入手。对于字符串中的每一个字符，计算它在字符串中出现的频率。然后，把字符按照频率由低到高的顺序进行排列。取两个频率最低的字符，创建一个节点带有两个子节点，其中这两个子节点就是上述这两个字符（和它的频率）。父节点的数据元素由两个子节点的频率之和构成。把这个节点插回到列表内。继续此过程直到把每一个字符都放入到树中。

当这个过程完成的时候，就会得到一棵完全二叉树，用此树就可以对哈夫曼编码进行解码了。解码过程包括顺着 0 和 1 的路径到达一个叶子节点，此节点会包含一个字符。

为了明白这项工作的全部内容，请查看图 17-2。

（原书 P328 图）

图 17-2 构造哈夫曼编码

现在准备来讨论一下构造哈夫曼编码的 C#语言代码。首先从创建 Node 类的代码开始。这个类与用于二叉查找树的 Node 类有很大的差别，因为这里所要做的就是存储一些数据和一个连接：

```
public class Node

{

    public HuffmanTree data;

    public Node link;

    public Node(HuffmanTree newData)

    {

        data = newData;

    }
```

```
}
```

下一个要讨论的类是 TreeList 类。这个类用来存储放入二叉树的节点列表，并且利用链表作为存储技术。代码如下所示：

```
public class TreeList

{

    private int count = 0;

    private Node first = null;

    private static string[] signTable = null;

    private static string[] keyTable = null;



    public TreeList(string input)

    {
```

```
        List<char> list = new List<char>();

        for (int i = 0; i < input.Length;i++ )

        {

            if (!list.Contains(input[i]))

                list.Add(input[i]);

        }

        signTable = new string[list.Count];

        keyTable = new string[list.Count];

    }

    public string[] GetSignTable()

    {
```

```
        return signTable;

    }

    public string[] GetKeyTable()

    {

        return keyTable;

    }

    public void AddLetter(string letter)

    {

        HuffmanTree hTemp = new HuffmanTree(letter);

        Node eTemp = new Node(hTemp);

        if (first == null)
```

```
            first = eTemp;

        else

        {

            eTemp·link = first;

            first = eTemp;

        }

        count++;

    }

    public void SortTree()

    {

        if (first != null && first·link != null)

        {
```

```
Node tmp1;

Node tmp2;

for(tmp1 = first;tmp1!=null;tmp1 = tmp1.link)

    for (tmp2 = tmp1.link; tmp2 != null; tmp2 = tmp2.link)

    {

        if (tmp1.data.GetFreq() > tmp2.data.GetFreq())

        {

            HuffmanTree tmpHT = tmp1.data;

            tmp1.data = tmp2.data;

            tmp2.data = tmpHT;

        }
```

```
            }

    }

}

public void MergeTree()

{

    if (!(first == null))

        if (!(first.link == null))

        {

            HuffmanTree aTemp = RemoveTree();

            HuffmanTree bTemp = RemoveTree();

            HuffmanTree sumTemp = new HuffmanTree("x");
```

```
                sumTemp·SetLeftChild(aTemp);

                sumTemp·SetRightChild(bTemp);

                sumTemp·SetFreq(aTemp·GetFreq() + bTemp·GetFreq());

                InsertTree(sumTemp);

            }

}

public HuffmanTree RemoveTree()

{

    if (!(first == null))

    {

        HuffmanTree hTemp;
```

```
            hTemp = first·data;

            first = first·link;

            count--;

            return hTemp;

        }

        return null;

    }

    public void InsertTree(HuffmanTree hTemp)

    {

        Node eTemp = new Node(hTemp);

        if (first == null)
```

```
            first = eTemp;

        else

        {

            Node p = first;

            while ((!(p·link == null))

            {

                if ((p·data·GetFreq() <= hTemp·GetFreq()) &&
(p·link·data·GetFreq() >= hTemp·GetFreq()))

                    break;

                p = p·link;

            }

            eTemp·link = p·link;
```

```
            p·link = eTemp;

        }

        count++;

    }

    public int Length()

    {

        return count;

    }

    public void AddSign(string str)

    {

        if (first == null)
```

```
{

    AddLetter(str);

    return;

}

Node tmp = first;

while (tmp != null)

{

    if (tmp·data·GetSign() == str)

    {

        tmp·data·IncFreq();

        return;
```

```
            }

            tmp = tmp.link;

        }

        AddLetter(str);

    }

    static string translate(string original)

    {

        string newStr = "";

        for (int i = 0; i <= original.Length - 1; i++)

            for (int j = 0; j <= signTable.Length - 1; j++)

                if (original[i].ToString() == signTable[j])
```

```
                newStr += keyTable[j];

    return newStr;

}

static int pos = 0;

static void MakeKey(HuffmanTree tree, string code)

{

    if (tree·GetLeftChild() == null)

    {

        signTable[pos] = tree·GetSign();

        keyTable[pos] = code;

        pos++;
```

```
        }

        else

        {

            MakeKey(tree·GetLeftChild(), code + "0");

            MakeKey(tree·GetRightChild(), code + "1");

        }

    }

}
```

下面这个类利用了 HuffmanTree 类，所以现在来看看这段代码：

```
public class HuffmanTree

{

    private HuffmanTree leftChild;
```

```
private HuffmanTree rightChild;

private string letter;

private int freq ;

public HuffmanTree(string letter)

{

    this.letter = letter;

    this.freq = 1;

}

public void SetLeftChild(HuffmanTree newChild)

{

    leftChild = newChild;
```

```
}

public void SetRightChild(HuffmanTree newChild)

{

    rightChild = newChild;

}

public void SetLetter(string newLetter)

{

    letter = newLetter;

}

public void IncFreq()

{
```

```
        freq++;

    }

    public void SetFreq(int newFreq)

    {

        freq = newFreq;

    }

    public HuffmanTree GetLeftChild()

    {

        return leftChild;

    }

    public HuffmanTree GetRightChild()
```

```
{
    return rightChild;
}
public int GetFreq()
{
    return freq;
}
public string GetSign()
{
    return letter;
}
```

```
}
```

最后，需要一个程序来测试实现：

```
static void Main()

{

    string input;

    Console·Write("Enter a string to encode: ");

    input = Console·ReadLine();

    TreeList treeList = new TreeList(input);

    for (int i = 0; i < input·Length; i++)

        treeList·AddSign(input[i]·ToString());

    treeList·SortTree();
```

```
while (treeList·Length() > 1)

    treeList·MergeTree();

MakeKey(treeList·RemoveTree(), "");

string newStr = translate(input);

string[] signTable = treeList·GetSignTable();

string[] keyTable = treeList·GetKeyTable();

for (int i = 0; i <= signTable·Length - 1; i++)

    Console·WriteLine(signTable[i] + ": " + keyTable[i]);

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);
```

}

17·2·3 用贪心算法解决背包问题

本章的前半部分已经介绍了背包问题，并且用动态规划技术编写了解决这个问题的程序。本节会再次看到这个问题，这次会用贪心算法来解决问题。

为了使用贪心算法来解决背包问题，放置在背包内的物品需要在本质上是“连续的”。换句话说，它们必须像布或者金粉那样不能被分离计算的物品。如果使用这类物品，那么可以简单地用单位价格除以单位数量来确定物品的价值。最优解是把具有最高价值的物品尽可能多地放入背包直到该物品耗尽或者装满背包为止，然后是尽可能地放价值次高的物品，如此等等。采用分离的物品不能找到最优贪心算法的原因是人们无法把“半台电视机”放入背包。

下面来看一个实例。假设你是一名偷毛毯的贼，而且你有一个可以装下 25“条”毛毯的背包。因此，你希望得到尽可能多的“好东西”来取得最大收益。你了解到打算偷的毛毯商店现在有下列这些毛毯类型和数量（按照单位价格）。

- Saxony：12 条，1·82 美元/条。
- Loop：10 条,，1·77 美元/条。
- Frieze：12 条，1·75 美元/条。
- Shag：13 条， 1·50 美元/条。

贪心算法规定你要拿尽可能多的 Saxony 毛毯，然后是尽可能多的 Loop 毛毯，再次是 Frieze 毛毯，最后才是 Shag 毛毯。为了成为可计算的类型，首先编写一个程序来模拟你的偷窃。下面就是你所提出的代码：

```
using System;

using System.Collections;

public class Carpet : IComparable

{

    private string item;

    private float val;

    private int unit;

    public Carpet(string i, float v, int u)

    {

        item = i;

        val = v;
```

```
        unit = u;

    }

    public int CompareTo(Object c)

    {

        return (this.val.CompareTo(((Carpet)c).val));

    }

    public int GetUnit()

    {

        return unit;

    }

    public string GetItem()
```

```
{

    return item;

}

public float GetVal()

{

    return val * unit;

}

public float ItemVal()

{

    return val;

}
```

```
}

public class Knapsack

{

    private float quantity;

    SortedList items = new SortedList();

    string itemList;

    public Knapsack(float max)

    {

        quantity = max;

    }

    public void FillSack(ArrayList objects)
```

```
{

    int pos = objects.Count - 1;

    int totalUnits = 0;

    float totalVal = 0.0F;

    int tempTot = 0;

    while (totalUnits < quantity)

    {

        tempTot += ((Carpet)objects[pos]).GetUnit();

        if (tempTot <= quantity)

        {

            totalUnits += ((Carpet)objects[pos]).GetUnit();
```

```
                totalVal += ((Carpet)objects[pos]).GetVal();

                items.Add(((Carpet)objects[pos]).GetItem(),
((Carpet)objects[pos]).GetUnit());

            }

            else

            {

                float tempUnit = quantity - totalUnits;

                float tempVal = ((Carpet)objects[pos]).ItemVal() * tempUnit;

                totalVal += tempVal;

                totalUnits += (int)tempUnit;

                items.Add(((Carpet)objects[pos]).GetItem(), tempUnit);

            }
```

```
            pos--;

        }

    }

    public string GetItems()

    {

        foreach (Object k in items.GetKeyList())

            itemList += k.ToString() + ": " + items[k].

            ToString() + " ";

        return itemList;

    }

    static void Main()
```

```
{

    Carpet c1 = new Carpet("Frieze", 1.75F, 12);

    Carpet c2 = new Carpet("Saxony", 1.82F, 9);

    Carpet c3 = new Carpet("Shag", 1.5F, 13);

    Carpet c4 = new Carpet("Loop", 1.77F, 10);

    ArrayList rugs = new ArrayList();

    rugs.Add(c1);

    rugs.Add(c2);

    rugs.Add(c3);

    rugs.Add(c4);

    rugs.Sort();
```

```
        Knapsack k = new Knapsack(25);

        k.FillSack(rugs);

        Console.WriteLine(k.GetItems());

    }

}
```

用 Carpet 类有两个原因：为了封装关于每种毛毯的数据，也为了实现 IComparable 的接口，这样就可以依据它们的单位价格来对毛毯类型进行排序。

在这个实现中 Knapsack 类完成了大部分的工作。它提供了一个列表来存储毛毯的类型，并且为了确定如何装满背包还提供了一个 FillSack 方法。此外，构造器方法允许还用户传递容量，这个容量设置了背包可以容纳的最大数量。

FillSack 方法循环遍历毛毯的类型，把最有价值的毛毯尽可能多地添加到背包内，然后移动到下一种类型上。当填满背包的那一刻，if 语句的 else 分句会把毛毯的正确数量放入背包。

这个程序可以运行是因为可以在任何需要的情况下分割毛毯。如果试图用其他一些无法达到连续数量的物品来填充背包，那么就不得不改为动态规划算法来解决问题了。

小结

本章讨论了算法设计的高中高级技术：动态规划和贪心算法。动态规划是一种自底向上求解问题的方法。不同于递归算法这类进行底层计算的算法，动态规划算法是从底部开始，并且

从这些底层结果上构建起来直到获得最终的解决方案。

贪心算法会尽可能快地寻找解决方案，然后在寻找到所有可能的解决方案之前停止。用贪心算法解决问题不需要最优解，这是因为贪心算法会在找到最优解之前停止在一个“足够好”的解决方案上。

练习

43. 请作为一个类来重新编写查找最长公共子串的代码。

44. 请编写一个程序用暴力穷举技术来查找最长公共子串。用 Timing 类来比较暴力穷举方法和动态规划方法。为了动态规划算法请使用练习 *1* 的程序。

45. 请编写一个视窗应用程序，此程序允许用户研究背包问题。用户应该能改变背包的容量，物品的尺寸及价值。用户还应该能创建与程序中用到的物品相关联的物品名称列表。

46. 请找到至少两种新面值的硬币，从而使得本章介绍用于找零钱问题的贪心算法产生非最优解。

47. 请利用诸如 WinZip 这样的“商业”压缩程序来压缩一个小的文本文件。然后用哈夫曼编码程序来压缩相同的文本文件。比较两种压缩技术的结果。

48. 请使用来自“毛毯盗贼”实例的代码，把要偷窃的物品改为电视机。你能把背包完全填满吗？请改变实例程序来回答这个问题。

索　引

*&(*ampersand*)* operator（*&*运算符）*,134*

($)*dollar sign, assertion made by（*$）（美元符号，由此符号引发的断言），*157*

·NET environment（·NET 环境）），*18*

application domain（·NET 环境的应用程序域）），*19*

as arrays and strings（如数组和字符串的·NET 环境）），*5*

timing test for（为·NET 环境进行的时间测试）），*18*

·NET Framework（·NET 框架

array class（·NET 框架的数组类）），*3*

ArrayLists（·NET 框架的 ArrayList 类）），*41*

collection classes in（·NET 框架中的群集类）），*1*

dictionary classes（·NET 框架的字典类），*8*

Stack class（·NET 框架的堆栈类），*69*

·NET Framework class library System

data structures（·NET 框架类库系统数据结构,*1*

·NET Framework library（·NET 框架库），*11*

ArrayList（·NET 框架库的 ArrayList 类），*35*

·NET version of c#（根据 c#改变的·NET），*93*

*[]*brackets, enclosing a character class（*[]*方括号），用于闭合一个字符类），*155*

\b assertion（\b 断言），*157*

\d character class（\d 字符类），*156*

\D character class（\D 字符类），*156*

\S character class（\S 字符类），*156*

\w character class（\w 字符类），*156*

\W character class（\W 字符类），*156*

A

Add method（Add 方法），*240*

for a dictionary object（用于字典对象的 Add 方法），*166*

in a BucketHash class（BucketHash 类中的 Add 方法），*181*

of the arraylist（arraylist 类的 Add 方法），*36*

storing data in a collection（群集中 Add 方法存储数据），*12*

AddEdge method（AddEdge 方法），*288*

AddRange method（AddRange 方法），*38*，*39*

AddVertex method（AddVertex 方法），*288*

Adelson-Velskii, G· M·（人名（AVL 树的发明人之一）,*263*

Adjacency matrix（邻接矩阵），*286*，*288*，*290*，*291*

adjustShortPath method（adjustShortPath 方法），*307*，*308*

advanced data structures for searching（用于查找的高级数据结构），*263*

algorithms（算法），*1*

advanced sorting（高级排序算法），*42*，*249*

binary search（二叉查找算法），*62*，*64*，*66*

Bubble Sort（冒泡排序算法），*45*

Determing node postion（确定节点位置算法），*222*

Dijkstra’s algorithms（Dijkstra 算法），*303*，*305*，*312*

greedy（贪心算法），*152*，*303*，*314*，*324*

HeapSort（堆排序算法），*254*

implementation（算法的实现），*290*

Insertion Sort（插入排序算法），*49*

iterative（迭代算法），*65*

knapsack problem（背包问题算法），*322*

minimum spanning tree（最小生成树算法），*299*

QuickSort（快速排序算法），*259*

recursive（递归算法），*65*

selection sort（选择排序算法），*48*

ShellSort（希尔排序算法），*249*

Shortest-path（最短路径算法），*302*

sorting（排序算法），*42*

topological sorting（拓扑排序算法），*289*

And operator（与运算符），*98*，*245*

anonymous group（匿名组），*158*

append method（追加方法），*140*

application domain（应用域），*19*

arithmetic expression, storing

as string（作为字符串存储的算术表达式），*7*，*74*

Array Class（Array 类），*26*

built-in binary search method（内置二叉查找方法），*65*

for retrieving metadata（用于检索元数据），*28*

array class method（数组类方法），*28*

array elements（数组元素），*28*

array Metadata（数组元数据），*28*

array object（数组对象），*26*

array techniques（数组技术），*125*

ArrayList class（Arraylist 类），*26*，*35*

applications of（ArrayList 类的应用），*36*

members of（ArrayList 类的成员），*35*

ArrayList object（ArrayList 对象），*35*

ArrayLists（ArrayLists 类），*3*，*11*，*12*

addrange/insertrange method（ArrayLists 的添加项/插入项方法），*38*

and resizing（ArrayLists 的调整），*41*

as buckets（ArrayLists 作为桶），*181*

capacity property（ArrayLists 的容量属性），*37*

comparing to arrays（ArrayLists 比作数组），*26*

contained in CollectionBase class（ArrayLists 包含在 CollectionBase 类中）），*12*

indexof method（ArrayLists 的 indexof 方法），*38*

remove method（移除方法），*37*

ArrayLists add method（ArrayLists 添加方法），*81*

ArrayLists object（ArrayLists 对象），*70*

arrays

as class objects（数组作为类对象），*3*

as linear collection storage（数组作为线形群集存储），*3*

compared to BitArray Class（数组与 BitArray 类比较），*94*

compared to linked list（数组与链表比较），*194*，*195*

concerning issues with（与数组相关的问题），*194*

declaring（数组声明），*27*

heap building（数组堆构造），*254*

indexed data collections（索引的数据群集），*26*

initializing（数组初始化），*27*

Jagged Arrays（齿状数组、锯齿数组），*32*

multidimensional arrays（多维数组），*30*

new elements insertions to（数组插入新元素），*3*

parameter arrays（参数数组），*32*

static/dynamic（静态/动态数组），*3*

arrBits（arrBits 数组），*114*

ASC function（ASC 函数），*127*

ASCII code（ASCII 码），*127*

ASCII values（ASCII 值），*177*，*240*

assertions（断言），*156*，*160*

Zero-Width Lookahead（零宽度正向预搜索断言），*160*

Zero-Width Lookbehind（零宽度反向预搜索断言），*160*

associations（联合），*8*

asterisk(*)（星号（*）），148

as quantifier（星号作为数量词），151

as the greedy operator（星号作为贪心运算符

AVL trees（AVL 树），263

fundamentals（AVL 树的基本原理），263

implementing（AVL 树的实现），264

nodes in（AVL 树上的节点），264，266

rotation（AVL 树旋转），263

AVLTree Class（AVLTress 类

deletion method（AVLTress 类的删除方法），268

B

benchmark tests（基准测试），*17*

benchmarking· See timing tests（基准测试），参见时间测试

Big O analysis（大 O 分析），*1*

bin configuration（bin 配置），*87*

binary number

converting to decimal equivalents（二进制数转化成等价的十进制数），*97*

binary number system（二进制数系统），*96*

binary number（二进制数），*94*，*96*

combining with bitwise operators（二进制数与位运算符相结合），*99*

comparing bit-by-bit（二进制数对比逐位），*98*

manipulating（二进制数处理），*97*

binary search（二叉查找），*55*，*62*

recursive（二叉查找递归），*64*

binary serach algorithm（二叉查找算法），*64*

using iterative and recursive code（采用迭代和递归代码的二叉查找算法），*66*

binary search method（二叉查找方法），*64*，*66*

binary search tree（二叉查找树），*218*，*220*，*235*

building（构造二叉查找树），*221*

finding node and

minimum/maximum values in（在二叉查找树中查找节点和最大/最小值），*227*

handling unbalanced（处理不平衡的二叉查找树），*263*

inserting series of numbers into（向二叉查找树中插入一系列数），*225*

leaf node (with One Child)

removal（二叉查找树(带一个孩子)叶子节点的移动），230

leaf node (with Two Children)

removal（二叉查找树（带两个孩子）叶子节点的移动），230

leaf node removal（二叉查找树叶子节点的移动），228

transversing（二叉查找树的遍历），224

binary trees（二叉树），9，218，220

BinarySearchTree(BST) class（BinarySearchTree(BST)类），221，222，268

binNumber（binNumber 数组），113

binNumber array (binary)（binNumber 数组（二进制）），113

bins, queues representing（柜子），队列表示），88

bit

index of bit to set（对集合的位索引），*113*

bit mask（位屏蔽），*107*

bit pattern

for an integer value（整数值的位模式），*104*

Bit sets（位集合），*94*

bit shift

demonstration application（位移展示应用），*107*

bit value retrieving（位值检索），*111*

BitArray

binNumber（BitArray 的 binNumber），*113*

BitSet（BitArray 的 BitSet,*113*

compared with array for

sieve of Eratosthenes（BitArray 与埃拉托色尼质数过滤数组比较），*117*

retrive a bit value（BitArray 检索位值），*111*

similar to arraylist（BitArray 类似于 arraylist），*110*

storing set of boolean values（BitArray 存储布尔值的集合），*117*

BitArray Class（BitArray 类），*94*，*110*

data structure to store

set members（BitArray 类作为存储集合成员的数据结构），*244*

finding prime numbers（BitArray 类查找指数），*94*

methods and properties（BitArray 类的方法和属性），*113*

storing sets of bits（BitArray 类存储位集合），*117*

writing the seive of

Eratosthenes（BitArray 类编写埃拉托色尼质数过滤），*94*，*96*

bitBuffer variable（bitBuffer 变量），*107*

Bits in VB·NET（VB·NET 中的位），*96*

BitSet（BitSet），*113*

bitshift operators（位移运算符），*94*，*97*，*103*

bitwise operator（按位运算符），*94*，*97*，*98*

and applicability（按位运算符和适用性），*99*

and ConvertBits method（按位运算符和 ConvertBits 方法），*99*

similar to boolean values（按位运算符类似于布尔值），*98*

truth tables（按位运算符真值表），*98*

black nodes（黑节点），*268*

Boolean truth table（布尔真值表），*98*

Boolean value（布尔值），*113*

breadth-first search（广度优先搜索），*293*，*296*

BubbleSort algorithm（冒泡排序算法），*45*，*46*

BubbleSort methods（冒泡排序方法），*47*

Bucket Hashing（桶式散列），*181*

buckets（桶），*181*

BuildArray subroutine（BuildArray 子程序），*90*

BuildGlossary subroutine（BuildGlossary 子程序），*189*

Byte values（字节值），*96*，*111*

C

C#（C#语言

and arrays in（C#语言中的数组），*26*

and regular expression（C#语言正则表达式），*156*

binary tree in（C#语言中的二叉树），*183*

built-in Hashtable class（C#语言内置散列表类），*183*

CStack（C#语言的 CStack 类），*70*

dimensions of arrays（C#语言数组的维数），*30*

in bitwise operators（C#语言在按位运算符中），*99*

peek operation（C#语言取数操作），*69*

role of sets（C#语言集合的职能），*237*

string as class object（C#语言字符串作为类对象），*3*

C# code

for constructing Huffman code（用于构造哈夫曼编码的 C#代码），*327*

C# string（C#字符串），*3*

C# struct（C#结构），*4*

C#），arrays（C#,数组），*3*

Capacity property

of the ArrayList object（ArrayList 对象的容量属性），*35*

CapturesCollection Class（CpaturesCollection 类），*161*

caret*(^)*（*(^)*符号），*155*

Carpet class（Carpet 类），*336*

carpet thief program（地毯偷贼程序），*337*

CArray class（CArray 类），*44*

in prime number sorting（在素数排序中的 CArray 类），*95*

storing numbers（CArray 类排序数），*44*

CArray class object（CArray 类对象），*44*

case-insensitive matching（不区分大小写的匹配），*163*

character array, instantiating a

string from（字符数组），实例说明字符串），*120*

character classes（字符类），*153*，*155*

*[*aeiou*]*（在本书表示元音字母类），*155*

period*(·)*（句点（·）字符类），*153*

characters

Unicode values of（字符的 Unicode 值），*127*

Chars method（Chars 方法），*83*

Chars property（Chars 属性），*139*

child deleting a node with one（删除带一个孩子的节点），*230*

Circular linked list（循环链表），*203*

class method（类方法），*29*

Clear method（Clear 方法），*13*，*76*

Of the ArrayList Class（ArrayList 类的清除方法），*70*

Coin-Changing Problem（找零钱问题），*324*

Collection Classes（群集类），*11*，*12*

built-in enumerator（集合类内置计数器），*11*

implementing using arrays（集合类用数组实现），*11*

in .NET Framework（在.NET 框架中的集合类），*1*

storing class object（集合类排序类对象），*11*

Collection operations（Collection 操作），*2*

CollectionBase class（CollectionBase 类），*11*

inner list（CollectionBase 类内部表），*12*

collections（群集），*1*，*2*

linear and nonlinear（线性和非线性集合），*2*

collections count（群集计数），*2*

Collision（冲突），*177*

collNumber（本书表示冲突的数量），*183*

comma-delimited string（逗号分隔的字符串），*125*

comma-seperated value strings (CSVs)（逗号间隔值字符串(简称 CSVs)），125

compareTo method（compareTo 方法），127

Compression of data（数据的压缩），326

computer programming role of stacks（堆栈的计算机编程职能），93

Concat method（Concat 方法），134

connected unidirected graph（连通无向图），284

connections between network（网络间连接），299

constructor method（构造器方法），239

for CSet class（为 CSet 类的构造器方法），239

for CStack（为 CStack 的构造器方法），70

for String class（为字符串类的构造器方法），120

constructors for Stack class（为堆栈类的构造器），*73*

Contains method（Contains 方法），*37*，*77*

ContainsKey method（ContainKey 方法），*188*

ContainsValue method（ContainsValue 方法），*188*

continuous items（连续项），*333*

ConvertBits function（ConvertBits 函数），*107*

ConvertBits method（ConvertBits 方法），*99*

copy constructors（复制构造器），*184*

CopyTo method（CopyTo 方法），*77*，*169*

cost· See also weight of the vertex （价值），参见顶点的权），*283*

Count method（Count 方法），*12*，*167*

Count property（Count 属性），*70*

and stack operation（Count 属性和堆栈操作），*69*

CSet class（CSet 类），*243*

BitArray implementation of（CSet 类的 BitArray 实现），*244*

CSVs *(*comma-separated value strings*)* （逗号间隔值字符串*(*简称 CSVs*),125*

CType function（Ctype 函数），*169*

custom-built data structure or

Algorithm（用户定制数据结构或算法），*66*

cycle（圈、回路），*284*

D

Data compression Huffman code（数据压缩哈夫曼编码），*326*

data fields（数据域），*206*

data items, memory reserved for（数据项的内存保留），*18*

data members for timing classes（时间类的数据成员），*21*

data structures（数据结构），*1*，*68*

data structures and algorithms（数据结构和算法），*1*

data types（数据类型

numeric（数字数据类型），*5*

default capacity（缺省容量

hash table with（缺省容量的散列表），*185*

of queue（队列的缺省容量），*82*

default constructor（默认构造器），*21*，*73*

for base class（基类的默认构造器），*167*

Delete method（Delete 方法），*233*

delVertex· See also graphs（delVertex 方法），参见图），*291*

DeMorgan’ s Laws（DeMorgan 定律），*239*

Depth of a tree（树的深度），*220*

depth-first search（深度优先搜索），*293*，*294*

Dequeue method（Dequeue 方法），*91*，*92*

Dequeue operation（Dequeue 操作），*7*，*80*，*90*

dictionary（字典），*8*，*42*，*165*

key-value paris（字典键值对），*8*

dictionary, associative arrays（字典），联合数组），*8*

DictionaryBase（DictionaryBase 类），*166*

DictionaryBase class,*165*·（DictionaryBase 类），*165*

See also SortedList Class（参见 SortedList 类），*172*

DictionaryBase Methods（DictionaryBase 方法），*169*

dictionary-based data structure

SortedList（基于字典的数据结构 SortedList），*165*

DictionaryEntry array（DictionaryEntry 数组），*169*

DictionaryEntry objects（DictionaryEntry 对象），*166*，*167*，*170*，*174*

Difference method（Difference 方法），*242*

digraph（有向图），*284*

Dijkstra, Edsger（人名），*303*

Dijkstra’s algorithm（Dijkstra 算法），*308*

direct access collections （直接存取群集），*2*

and struct（直接存取集合和结构），*3*

string（直接存取集合字符串），*3*

directed graph·

See digraph displaying method（有向图），参见有向图显示方法），*47*

displayNode method（displayNode 方法），*221*，*226*

displayPaths method（displayPaths 方法），*308*

dispMask variable（dispMask 变量），*107*

DistOriginal class（DistOriginal 类），*306*

distributive set property（分布式集合属性），*238*

Double Hashing（双散列），*181*，*183*

double quotation marks

enclosing string literals（双引号标记闭合的文字串），*120*

double rotation in an AVL tree（在 AVL 树上的双旋转），*264*

doubly-linked list（双向链表），*200*

node deletion（节点删除），*201*

Remove method（Remove 方法），*201*

duration members of Timing class（时间类的持续时间成员），*21*

dynamic programming （动态规划），*314*

arrays for storing data（用于存储数据的数组），*318*

E

ECMAScript option

for regular expression（用于正则表达式的 ECMAScript 选项），*163*

edges（边

node connected by（用边连接的节点），*218*

representing as graph（边表示成为图），*286*

elements（元素

accessing an arrays（访问数组元素），*28*

accessing multidimensional

arrays（访问多维数组元素），*29*，*31*

adding to an array（向数组添加元素），*3*

empty set（空集合），*238*

empty string（空串），*120*

EnQueue operation（EnQueue 操作），*7*，*80*

EnsureCapacity method（EnsureCapacity 方法），*139*

Enumerator object

for a hash table（散列表的计数器对象），*185*

equal set（相等集合），*238*

equalities for set（集合等式），*239*

equality, testing for（等式测试），*26*

Equals method（Equal 方法），*127*

equivalent table

for bit values（位值的换算表），*98*

Eratosthenes（人名），*94*

ExplicitCapture

for regular expression（正则表达式的显式捕获），*163*

expression evaluator（表达式求值器），*74*，*77*

extra connections

in a network（网络中的额外连接），*299*

F

False bit（假值位），*98*

Fibonacci numbers（斐波纳契数列），*315*

computation using recursive

and iterative version（用递归方法和迭代方法计算斐波纳契数列），*317*

FIFO *(*First-In，First-Out*)*

structures（先进先出结构），*79*，*80*

FillSack method（FillSack 方法），*336*

finalizer method（finalizer 方法），*19*

FindLast method（FindLast 方法），*202*

FinsMax method（FindMax 方法），*282*

FindMin function（FindMin 函数），*59*

FindMin*()* method（FindMin*()*方法），*227*

First-In, First-Out structures

*(*FIFO*)*（先进先出结构），*79*，*80*

fixed-length code（固定长度编码），*327*

For Each loop（For Each 循环），*36*

For loop（For 循环），*28*，*107*，*258*，*280*

formatted string（格式化串），*140*

found item, swapping with preceding（优先交换找到项），*61*

frequency of occurrence

for a character in a string（字符串中字符的出现频率），*327*

frequently searched-for items,

placing at beginning（频繁搜索项放置在开始处），*59*

G

garbage collection（无用单元收集），*18*

garbage collector, calling（调用无用单元收集器），*18*

generalized indexed collections（通用的索引群集），*7*

generic class（泛型类），*16*

Generic Linked List（泛型链表），*214*

Generic Linked List Class（泛型链表类），*214*

Generic Node Class（泛型节点类），*214*

generic program

data type placeholder（泛型程序数据类型占位符），*14*

generic programming（泛型编程），*1*，*14*

generic Queue（泛型队列），*82*

generic Swap function（泛型交换函数），*14*

generics（泛型），*1*

genRandomLevel method（genRandomLevel 方法），*280*

Get method

BitSet BitArray（BitSet BitArray 的 Get 方法），*111*

to retrieve bits stored（检索位存储的 Get 方法），*111*

GetAdjUnvisitedVertex method（GetAdjUnvisitedVertex 方法），*294*

getCurrent method（getCurrent 方法），*207*

GetEnumerator method（GetEnumerator 方法），*169*

GetLength method（GetLength 方法），*29*

getMin method（getMin 方法），*37*，*308*

GetRange method（GetRange 方法），*39*，*40*

GetSuccessor method（GetSuccessor 方法），*233*

GetType method（GetType 方法），*29*

for data type of array（用于数组类型的 GetType 方法），*29*

GetUpperBound method（GetUpperBound 方法），29

GetValue method（GetValue 方法），28

global optimum（全局最优），314

glossary, building with hash table（用散列表构造术语表），189

Graph Class（图类），285，306

Graph search algorithm

minimum spanning tree（最小生成树的图查找算法），299

graphs（图），10

building（构造图），287

minimum spanning trees（最小生成树），299

real world systems modeled by（用图对现实世界系统建模），284

represented in VB·NET（用 VB·NET 表示），*283*

searching（查找图），*293*

topological sorting（拓扑排序），*289*

vertex removal（垂直移动），*291*

weighted（带权图），*302*

Greedy algorithms（贪心算法），*303*，*314*，*324*

group

nonlinear collection，

（unordered（非线性群集），无序组），*9*

group collections（组群集），*9*

Grouping Constructs（组构造），*157*

H

HandleReorient method（HandleReorient 方法），*275*

hash function（散列函数），*8*，*176*，*177*，*181*

in a BucketHash class（BucketHash 类中的散列函数），*181*

Hash table（散列表

addition/removal of elements（添加/移除散列表元素），*182*

building glossary or dictionary（构建术语表或字典），*189*

hash function（散列函数），*8*

key/value pairs stored in（存储在散列表中的关键字/值对），*166*

load factor（负载系数），*182*

remove method（移除方法），*167*

retrieving keys and values from（从散列表中检索关键字和值），*185*

Hashtable class（Hashtable 类），*176*，*184*

·NET Framework library（·NET 框架库），*176*

methods of（Hashtable 类的方法），*74*

Hashtable objects

instantiating and adding

（data to（Hashtable 对象初始化和数据添加），*184*

load factor（负载系数），*184*

heap（堆），*18*

building（构造堆），*254*

heap sort（堆排序），*9*

HeapSort Algorithm（堆排序算法），*254*

hierarchical collections（层次群集），*2*，*8*

and tree（分等级的群集和树），*8*

hierachical manner, storing data（存储数据的分等级方式），*218*

Horner’s rule（Horner法则（也译为霍纳法则）），*179*

HTML anchor tag（HTML 锚标签），*164*

HTML formatting（HTML 格式化），*136*

Huffman code（哈夫曼编码），*327*

Huffman code algorithm（哈夫曼编码算法），*327*

Huffman coding（哈夫曼编码），*326*

data compression using（用哈夫曼编码进行数据压缩），*326*

Huffman, David（人名），*326*

Huffman Tree class（哈夫曼树类），*331*

I

Icollection and arraylists（Icollection 和 arraylist），*38*

ICollection interface（Icollection 接口），*72*

IComparable interface（Icomparable 接口），*264*，*336*

IDictionary interface（Idictionary 接口），*166*

IEnumerable interface（Ienumerable 接口），*11*

If-Then statement, short-circuiting（短路的 If-Then 语句），*37*，*61*

IgnoreCase option

for regular expression（正则表达式的 IgnoreCase 选项），*163*

IgnorePattern WhiteSpace

Option for regular expression（正则表达式的 IgnorePattern WhiteSpace 选项），*163*

immutable String objects（不可变字符串对象），*119*

immutable strings（不可变字符串），*3*

increment sequence（自增序列），*249*

index-based access

into a SortedList（SortedList 中基于索引的存取访问），*174*

IndexOf method（IndexOf 方法），*38*，*122*

infix arithmetic（中缀运算），*74*

initial capacity for a hash table（散列表的初始容量），*184*

initial load factor for a hash table（散列表的初始负载系数），*184*

initialization list（初始化表），*27*

inner loop（内循环

in an insertion sort（插入排序中的内循环），*50*

in an selection sort（选择排序中的内循环），*48*

InnerHashTable（InnerHashTable,*166*

InnerHashTable object（InnerHashTable 对象），*167*

InnerList（InnerList,*12*

inOrder method（inOrder 方法），*225*，*226*

inorder successor（中序后继者），*230*

inorder traversal method（中序遍历方法），*224*，*225*

Insert method（Insert 方法），*141*

InsertAfter method（InsertAfter 方法），*207*

InsertBefore method（InsertBefore 方法），*207*

InsertBeforeHeader Exception class（InsertBeforeHeader Exception 类），*207*

Insertion method（Insertion 方法），*201*

Insertion Sort（插入排序），viii），*49*

loops in（插入排序中的循环），*50*

speed of（插入排序的速度），*52*

Int*32* structure（Int*32* 结构），*5*

Integer array（整数数组），*33*

Integer data type（整数数据类型），*5*

Integer index, *2,8*· See also

Direct access collections（整数索引），*2*，*8*。参见直接存取群集

integer set members（整数集合成员），*244*，*248*

Integer variable（整数变量），*70*

integers（整数

bit pattern determination（整数位模式的确定），*104*

converting into binary numbers（整数转换成为二进制数），*104*

Integer-to-Binary converter

application（整数到二进制数的转换器应用），*104*

intersection（交叉），*9*，*238*

Intersection method（Intersection 方法），*241*

invalid index（无效的下标），*38*

IP addresses（IP 地址），*166*，*172*

IPAddresses class（IPAddresses 类），*168*

isArray class method（isArray 类方法），*29*

IsMatch method（IsMatch 方法），*149*

isSubset Method（isSubset 方法），*241*

Item method（Item 方法

calling（调用 Item 方法），*70*

key-value pair（关键字-值对），*185*

of HashTable class（HashTable 类的 Item 方法），*167*

retrieving value（检索值），*166*，*167*

Iterator class（Iterator 类），*200*，*206*

insertion methods（Iterator 类的插入方法），*207*

iterFib function（iterFib 函数），*318*

J

Jagged arrays（齿状数组、锯齿状数组），*32*

Join method（Join 方法），*124*

from an array to a string（从数组到字符串的 Join 方法），*124*，*126*

K

Key（关键字

retrieving value based on（基于关键字的检索值），*186*

Key property

for a dictionaryEntry object（dictionaryEntry 对象的关键字属性），*170*

Key value, *220*·See also

key value pair（键值），*220*。参见键值对。

Key-value pairs· See also key value（键值对），参见键值。

KeyValuePair Class（KeyValuePair 类），*171*

KeyValuePair object

instantiating（初始化 KeyValuePair 对象），*171*

knapsack class（背包类），*336*

knapsack problem（背包问题），*322*

greedy solution to（贪心方法解决背包问题），*333*

Knuth, Don（人名），*11*

L

Landis, E· M·（人名），*263*

Last-In, First-Out (LIFO) structure（后进先出结构），7

lazy deletion（懒惰删除），268

lazy quantifier（惰性量词），153

LCSubstring function（LCSubstring 函数），321

left shift operator (<<)（左移位运算符（<<)），103

left-aligning a string（左对齐字符串），132

Length method

for multi-dimensional array（多维数组的 Length 方法），29

Length property（Length 属性），139

of StringBuilder class（StringBuilder 类的 Length 属性），138

levels（层

breaking tree into（树分层），220

determining for skip lists

（of links（确定跳跃链表），277

LIFO（Last-In，First-Out structure）（后进先出结构），7

Like operator（Like 运算符

Linear collections（线性群集），7

and array（线性集合和数组），2

direct/sequential access（直接/顺序存取访问），2

list of elements（元素表），2

linear list（线性表），6

direct access to elements（线性表直接存取元素），6

ordered or unordered（有序或无序的线性表），6

priority queue（优先队列），*7*

sequential access collections（顺序存取群集），*6*

stacks（堆栈

（last in, first-Out structures（后进先出结构），*7*

stacks and queues（堆栈和队列），*7*

linear probing（线性探查），*183*

link member of node（节点的链成员），*197*

linked list（链表

design modifications in（设计修正链表），*200*

doubly/circular linked list（双链表/循环链表），*200*

insertion of items（链表插入项），*196*

object-oriented design（面向对象设计），*196*

removal of items（链表删除项），*196*

LinkedList class（LinkeList类），*197*，*206*，*207*，*208*，*214*，*217*

LinkedListNode（LinkedListNode,*214*

load factor（负载系数），*184*

local optima（局部最优），*314*

logical operators（逻辑运算符），*98*

Lookbehind assertions（Lookbehind断言），*160*，*161*

loop（环），*284*

M

machine code, translation

recursive code to（递归代码翻译成机器代码），*314*

MakeChange subroutine（MakeChange 子程序），*326*

mask），See also bit mask

converting integer into a

（binary number（掩码），参见把整数转换成为二进制数的位屏蔽），*104*

Match class（Match 类），*148*，*149*

MatchCollection object（MatchCollection 对象），*150*

matches

at the beginning of a string

or a line（在字符串或行的开始处匹配），*156*

at the end of the line（在行的末尾处匹配），*157*

specifying a minimum and

（a maximum number of（确定匹配的最大值和最小值），*152*

specifying at word

（boundaries（确定词的边界），*157*

MaxCapacity property（MaxCapacity 属性），*138*

merge method, called by

RecMergeSort（RecMergeSort 调用的合并方法），*252*

MergeSort algorithm（MergerSort 算法），*251*

metacharacter（元字符），*147*

asterisk*(*)*（星号（*）元字符），*148*

minimum spanning tree algorithm（最小生成树算法），*299*

modern operating systems

tree collection（现代操作系统树群集），*9*

moveRow method（moveRow 方法），*291*

multi-dimensional array（多维数组），*29*，*30*

accessing elements of（多维数组的存取元素），*31*

performing calculations on

（all elements（对多维数组中的所有元素进行计算），*31*

Multiline option

for regular expression（正则表达式的多行选项），*163*

MustInherit class（MustInherit 类），*166*

mutable String objects（mutable String 对象），*137*

myfile.exe（某可执行文件名），*148*

N

named groups（命名组），*158*

native data type（本地数据类型），*120*

negative integers, binary

representation of（二进制表示的负整数），*105*

negative lookahead assertion（负的前向断言），*160*

network graph（网络图），*10*

Node class（节点类），*196*，*200*

nodes（节点

connected by edges（边连接的节点），*10*

in linked list（链表中的节点），*195*

of a tree collection（树群集的节点），*8*

nonlinear collections

hierarchical and group

（collections（非线性群集分为层次集合和组群集合），*8*

trees, heaps, graphs and

（sets（树、堆、图和集合），*2*

unordered group（无序组），*9*

NP-complete problems（NP-完全问题），*10*

NUM_VERTICES constant of

the graph class（图类的 NUM_VERTICES 常量），*288*

numElements（numElements,*250*

numeric codes for characters（字符的数字编码），*127*

O

object-oriented programming *(OOP)*（面向对象编程（OOP）），*11*，*70*

code bloat（代码膨胀），*14*

octal, converting numbers

from decimal to（十进制数转化成为八进制数），*78*

OOP*(*object-oriented programming*)*（面向对象编程），*11*

open addressing（开放定址），*181*，*183*

operations, performed on sets（在集合上执行的操作），*238*

optimal solution

for greedy algorithm（贪心算法的最佳解决方案），*324*

Or operator（或运算符），*245*

ordered graph（有序图），*284*

ordered list（有序表），*6*

organizational chart（组织结构图），*2*

ORing（ORing），*101*

P

PadLeft method（PadLeft 方法），*132*

PadRight method（PadRight 方法），*132*

palindrome（回文），*71*，*93*

ParamArray keyword（ParamArray 关键字），*32*

parameter arrays（参数数组），*32*

parameterized constructor（参数化构造器），*197*

parentheses(), surrounding

regular expression（包围正则表达式的括号（）），157

Pareto distributions（帕累托分布），60

Pareto, Vilfredo（人名），60

Parse method Int32（Int32 的 Parse 方法），5

Path· See also vertices

sequence in graph（路径），参见图中的顶点序列），284

finding the shortest in graph（查找图中的最短路径），302

Path() method（Path()方法），306

Pattern matching（模式匹配），147

Peek method· See Queue operations Peek（方法），参见队列操作

period *(.)* character class（句点（.）字符类），*153*

period matches（句点匹配），*154*

pig Latin（故意颠倒英语字母顺序拼凑而成的行话,*146*

pivot value（主元素值），*262*

plus sign *(+)* quantifier（加号（+）数量词），*151*

Pop method（Pop 方法），*70*，*73*

Pop operation· See stack operations（弹出操作），参见堆栈操作

postfix expression evaluator（后缀表达式求值器），*93*

postorder traversals（后序遍历），*224*，*226*

PQueue class（PQueue 类），*91*

code for（PQueue 类编码），*91*

preoder traversal method（前序遍历方法），*224*

primary stack operations（基本堆栈操作），*74*

PrintList method（PrintList 方法），*199*

Priority Queues（优先级队列），*90*

deriving from Queue class（来自 Queue 类的优先级队列），*90*

Private constructor（Private 构造器），*279*

for the SkipList class（SkipList 类的 Private 构造器），*278*

probability distribution（概率分布），*277*

Probability distributions（概率分布），*60*

Process class（Process 类），*19*

process handling（过程处理），*90*

Property method（Property 方法），*264*

Public constructor（Public 构造器），*279*

Pugh, william（人名），*277*

punch cards（穿孔卡片），*86*

Push method（Push 方法），*74*

Q

Quadratic probing（平方探查），*183*

quantifiers（数量词），*151*

asterisk*(*)*（星号（*）数量词），*151*

question mark *(?)* quantifier（问号（？）数量词），*151*

Queue class（Queue 类），*68*，*80*，*90*

implemention using an ArrayList（用 ArrayList 实现 Queue 类），*81*

Queue object（Queue 对象），*82*

Queue operations（Queue 操作），*80*

Peek method（Peek 方法），*70*，*76*，*80*

queues（队列），*68*，*80*

and applications（队列及其应用），*93*

changing the growth factoe（改变生长因素），*82*

First-In，First-Out structure（队列的先进先出结构），*7*

for breadth-first search（队列的广度优先搜索），*296*

used in sorting data（队列用于数据排序），*86*

QuickSort algorithm（快速排序算法），*259*

improvement to（改进快速排序算法），*262*

R

radix sort（基数排序），*87*

random number generator（随机数生成器），*44*

range operators

in like comparisons（在 like 比较中的范围运算符

Rank property（Rank 属性），*29*

readonly Property（只读属性），*264*

rebalancing operations· See

AVL trees（重新平衡操作），参见 AVL 树

recMergeSort method（recMergeSort 方法），*252*

recMergeSort subroutines（recMergeSort 子程序），*253*

recursion（递归

base case of（基于递归的实例），*252*

reverse of（递归的反向），*314*

recursive code, transting

to maching code（递归编码翻译成机器编码），*314*

recursive program（递归程序），*315*

RedBlack class（RedBlack 类），*270*，*275*

red-black tree（红黑树），*263*，*268*

implementation code（红黑树实现编码），*270*

insertion of items（红黑树插入项），*269*

rules for（红黑树法则），*269*

Redim Preserve statements（Redim Preserve 语句），*3*

reference types（参考类型），*18*

RegEx class（RegEx 类），*147*，*148*

regular array（规则数组），*95*

regular expression（正则表达式），*147*

compiling options（编译选项），*163*

for text processing and

pattern matching（用于文本处理和模式匹配的正则表达式），*164*

in C#（C#语言中的正则表达式），*148*

metacharacters（元字符），*147*

modifying using assertions（修改使用断言），*156*

myfile·exe（某可执行文件名），*148*

options（选项），*163*

searches and substitution

（in strings（正则表达式查找和字符串中的替换），*147*

surrounding parentheses（括号包围的正则表达式），*157*

working with（使用正则表达式），*148*

Remove method（Remove 方法），*12*

RemoveAt method（RemoveAt 方法），*38*

Replace method（Replace 方法），*150*

Right shift operator *(>>)*（右移位运算符（>>）），*103*

root node（根节点），*9*，*219*

RSort subroutine（Rsort 子程序），*90*

S

searching（查找），*42*

Searching Algorithms（查找算法），*55*

Selection Sort（选择排序），*48*

compared with other

（sorting algorithms（选择排序与其他排序算法比较），*53*

SelectionSort algorithm（SelectionSort 算法），*48*

code to implementation（SelectionSort 算法的代码实现），*48*

SeqSearch method（SeqSearch 方法），*60*

compared with Bubble sort（SeqSearch方法与冒泡排序的比较），*61*

self-organisation（自组织），*60*

sequential access collections（顺序存取群集），*6*

Sequential search（顺序查找），*55*

implementation of（顺序查找的实现），*55*

minimum and maximum

（values search by（通过顺序查找寻找最大最小值），*58*

speeding up（加快顺序查找），*59*

Sequential search function（顺序查找函数），*57*

Set class（Set类），*237*

Implementation using Hash table（用散列表实现Set类），*239*

Set method（Set方法），*113*

set of edges（边集合），*10*

set of nodes（节点集合），*10*

set operations（集合操作），*9*

SetAll method（SetAll 方法），*113*

Sets（集合），*237*

operations performed on（集合上进行的操作），*238*

properties defined for（集合的属性定义），*238*

remove/size methods（移除方法/数量计算方法），*240*

unordered data vaues（无序数据值），*9*

SetValue method（SetValue 方法），*28*

comparing with

（multidimensional array（SetValue 方法与多维数组比较），*31*

Shell, Donald, *249*· See also

ShellSort algorithm（人名），*249*。可参见希尔排序算法

ShellSort Algorithm（希尔排序算法），*249*

shortest-path algorithm（最短路径算法），*302*

showVertex method（showVertex 方法），*300*

sieve of Eratosthenes（埃拉托斯特尼筛法），也叫找质数筛法），*94*，*117*

using a BitArray to write（用 BitArray 写埃拉托斯特尼筛法），*114*

using integers in the array（用整数数组的埃拉托斯特尼筛法），*96*

skip lists（跳跃表），*263*，*275*

fundamentals（跳跃表的基本原理），*275*

implementation（跳跃表的实现），277

SkipList class（SkipList 类），281

public/private constructor（public/private 构造器），278

Sort method

in several .NET Framework

library classes（几种.NET 框架库类中 Sort 方法），262

SortedList（SortedList），165

SortedList class（SortedList 类），165，172

Sorting（排序），42，44，45，87

data with Queue（队列数据的排序），86

Sorting algorithms（排序算法），42

Bubble Sort（冒泡排序），45

time comparisons for all

（sorting algorithms（所有排序算法的时间比较），*51*

Sorting data

algorithms for（排序数据的算法），*53*

Sorting process（排序过程），*46*

sorting techniques（排序技术），*43*

sPath array（sPath 数组），*308*

splay tree（伸展树），*263*

Split method（Split 方法），*124*

string into parts（串成部分），*124*

Stack class（Stack 类），*68*，*70*，*72*，*73*，*78*

Stack Constructor Methods（Stack Constructor方法），*73*

stack object（堆栈对象），*73*

stack operations（堆栈操作），*7*，*74*

Pop operation（Pop 操作），*69*

pushing, poping, and

（peeking（进栈、出栈以及取数操作），*17*

stacks（堆栈），*7*，*18*，*68*

contains method（包含方法），*77*

in programming language

（implementations（堆栈在编程语言中的实现），*68*

Last-in, First-out *(*LIFO*)*

（Data structure（后进先出结构），*69*

Stacks applications（堆栈应用），*7*

stacks, data structure（数据结构堆栈），*79*

string array（字符串数组），*113*，*125*

String class（String 类），*119*

compared to StringBuilder（String 类与 StringBuilder 的比较），*143*

Like operator（Like 运算符

methods involved（包含的方法），*124*

methods of（String 类的方法），*121*

PadRight/PadLeft method（PadRight/PadLeft 方法），*132*

String class methods（String 类方法），*83*

string literals（文字串），*119*，*120*，*141*

String objects（String 对象），*119*

comparing in VB·NET（String 对象在 VB·NET 中的比较），*126*

concatenating（串联），*134*

instantiating（初始化），*120*

string processing（字符串处理），*119*，*130*，*145*，*147*

StringBuffer class（StringBuffer 类），*146*

StringBuilder class（StringBuilder 类），viii），*3*，*119*，*137*，*138*，

142，*143*，*145*

StringBuilder objects

and Append method（StringBuilder 对象和 Append 方法），*140*

constructing（构造 StringBuilder 对象），*138*

modifying（修改 StringBuilder 对象），*139*

obtaining and setting

（information about（获得并设置关于 StringBuilder 对象的信息），*138*

strings（字符串），*119*

aligning data（对齐数据），*132*

breaking into indivisual

（pieces of data（字符串分成不可分割的数据段），*124*

building from arrays（字符串的构建来自数组），*126*

collection of characters（字符的群集），*3*

comparing to patterns

converting from lowercase

to uppercase（小写字母转化成大写字母方式比较），*135*

defining range of characters in（字符串中定义字符范围），*154*

finding longest common substring（查找最长公共子串），*319*

in VB·NET（VB·NET 中的字符串），*121*

length of（字符串的长度），*121*，*122*

matching any character in

（methods for comparing（在比较方法中匹配任意字符），*126*

（methods for manipulating（在处理方法中匹配任意字符），*130*

operation performed（执行的操作），*121*

palindrome（回文），*71*

replacing one with another（用一个替换另一个），*142*

StartsWith and EndsWith

（comparsion methods（StartsWith 和 EndsWith 比较方法），*129*

struct（结构），*3*

subroutine DispArray（DispArray 子程序），*321*

Substring method（Substring 方法），*122*

Swap function（Swap 函数），*14*

System·Array class（System·Array 类），*26*

T

text file（文本文件），*191*

Text Processing（文本处理），*147*

TimeSpan data type（TimeSpan 数据类型，*21*

Timing class（Timing 类），*1*

and data members（Timing 类和数据成员），*21*

measurement of data

（structure and algorithms（数据结构和算法的测量法），*1*

timing code（时间编码），*18*，*19*，*21*

moving into a class（时间编码放入类中），*23*

Timing Test class（Timing Test 类），*21*

Timing tests（时间测试），*17*

For .NET environment（.NET 环境的时间测试），*18*

oversimplified example（时间测试过分简单化的例子），*17*

ToArray method（ToArray 方法），*39*，*78*

transfer of contents（转移内容），*40*

topological sorting（拓扑排序），*289*

methods of（拓扑排序方法），*290*

TopSort method（TopSort 方法），*292*

ToString method（ToString 方法），*143*，*170*

Traveling Salesman problem（旅行商问题），*10*

traversal methods（遍历方法），*224*

tree（树

leaf（叶子），*220*

set of nodes（节点集合），*218*

tree collection（树群集），*8*

applications of（树群集的应用），*9*

elements of（树群集的元素），*8*

tree transversal（树的遍历），*220*

TreeList class（TreeList 类），*329*

Trim method（Trim 方法），*135*

TrimEnd methods（TrimEnd 方法），*135*

True bit（真值位），*98*

two-dimensional array（二维数组），*33*

building LCSubstring function（构建 LCSubstring 函数），*321*

declearation（二维数组声明），*30*

result storage（结果存储），*319*

U

Unicode character set· See string（Unicode 字符集和），参见字符串

Unicode table（Unicode 表），*127*

Union（联合），*9*，*238*

Union method（Union 方法），*241*

Union operation（Union 操作），*241*

universe（全域），*238*

unordered array, searching（查找无序数组），*58*

unordered graph（无序图），*284*

unordered list（无序表），*6*

upper bound of array（数组上界），*62*，*110*，*262*

utility methods of Hashtable class（Hashtable 类的工具方法），*187*

V

value· See also Boolean value（值），参见布尔值），*113*

Value property

for DictionaryEntry object（DictionaryEntry 对象的 Value 属性），*170*

Value types（值类型），*18*

variable-length code（可变长度编码），*327*

Variables

assigning the starting time to（开始时对变量赋值），*23*

stored on heap（变量存储在堆中），*18*

stored on stack（变量存储在堆栈中），*18*

VB·NET

Manipulation of Bit（VB·NET 的位操作 *96*

skip list（跳跃表），*277*

VB·NET applications（VB·NET 应用），*97*

vertex（顶点），*283*

Vertex class（Vertex 类

building（构建 Vertex 类），*285*

for Dijkstra’s algorithms（Dijkstra 算法的 Vertex 类），*305*

Vertices（顶点（复数）

in graph（图中的顶点），*283*，*284*，*312*

representing（顶点表示），*285*

Vertices sequence in graph（图中的顶点序列），*284*

Visual Studio·NET（Visual Studio·NET,*46*

W

weight of the vertex（顶点的权），*283*

wildcards（通配符

Windows application

bit shifting operators（窗口应用位移运算符），*107*

X

Xor operator（Xor 运算符），*99*

Z

zero-based array（下标从零起始的数组），*170*