前　言

编者在写这本书时遇到了两个问题。第一个问题是关于数据结构教材。应该说关于数据结构的教材已经很多了。自从美国唐．欧．克努特教授用汇编语言写的《计算机程序设计技巧》第一卷《基本算法》问世以来，已经出现了用 PASCAL、C、C++、JAVA 等语言写的数据结构书。所以，在编者写本书之前，曾经感到很为难。目前，C#语言作为微软在新一代开发平台.NET 推出的、完全面向对象的语言，凭着其简洁、高效、模板、标准化的特性，使得 C#语言像程序设计语言中的一件艺术品，也吸引着越来越多的开发人员。这也使得我院的可视化专业进行专业改革时，决定以 C#语言作为该专业的主要开发语言。所以说，用 C#语言来讲授《数据结构》课程是我院专业改革的结果。而用 C#语言写的数据结构教材目前国内基本上是空白。鉴于此，编者决定写本书。

在接下来的写作过程中，编者遇到了另外一个问题，那就是 C#语言和.NET Framework 的发展。当作者写这本书时，是以 C#语言和.NET Framework 的 2.0 版本来写的。但是，到目前为止，C#语言和.NET Framework 已经出现 3.0 版本了。这使得编者感到了微软技术的发展之快，发出了“学习微软的东西在某种程度上是一种痛苦”之叹！也使编者曾产生了放弃写该书的念头。但作为教师的责任和对新东西的执著使得编者一直坚持，直到该书完稿。也附带说一句：如果读者在阅读过程中，发现有些技术不是最新的技术也不要惊奇，本书是以 C#语言和.NET Framework2.0 版本来写的。

本书的内容

本书分为 8 章，第 1 章介绍了数据结构和算法的基本概念及本书用到的数学和 C#的知识；第 2 章至第 6 章分别讨论了线性表、栈和队列、串和数组、树型结构和图结构等常用的数据结构及其应用，以及在.NET 框架中相应的数据结构；第 7、8 两章分别讨论了排序和查找常用的各种方法及其应用以及在.NET 框架中相应的算法。

本书特点

将数据结构与 C#语言和.NET 框架结合是本书的一大特点。.NET 平台是微软推出的一个新的开发平台，目的是让“不同的语言共享同一平台”。.NET 很可能成为下一代 Windows 操作系统的一部分。而 C#语言作为新一代完全面向对象的语言，是.NET 的母言。本书所有的数据结构和算法都是用 C#语言进行描述，并在相应章节的末尾介绍了在.NET 框架中常用的数据结构和算法。用 C#在.NET 平台开发的技术人员可以从本书中获得许多有益的知识和技术。

使用配套光盘

本书配套光盘中包含以下内容：

1、 code 目录是本书所有的代码及一个《学生信息管理系统》的代码。code 目录包含案例和 chapter1~chapter8 等 9 个子目录。

案例子目录中是《学生信息管理系统》的代码。《学生信息管理系统》是学生上学期学习《C#初级编程》课程所做的一个小系统，是学生在没有学过《数据结构》课程时算法。目的在于让学生比较采用数据结构和算法与不采用数据结构与算法的不同。所以，把这个小的系统作为《数据结构（C#）》课程的学习素材。考虑到有些学校在选用本教材时学生没有做过这个系统，所以，把代码全部给了出来。

chapter1~chapter8 等 8 个目录分别对应本书的相应章节。其中每个目录中的 source 子目录是本书中的有关源代码，涉及各个数据结构的接口、结点类、数据结构类的 C#代码及常用算法都放在相应章节目录下的 source 子目录中。

chapter1~chapter8 等目录中还有一个 project 子目录，里面有一个或多个项目，是使用各种数据结构和常用的排序和查找算法来解决《学生信息管理系统》的项目，是案例内容在数

据结构中的推广和延伸。所有的代码都没有完成，可作为教师教学、学生实验、课程设计等的素材使用。其中，chapter1 中的 project 子目录是各个例题中问题应用的项目。chapter4 由于 string 和 array 是经常使用的数据结构和数据类型，所以，没有 project 子目录而只有 source 子目录。chapter6 由于图的内容高职层次的学生很少涉及，所以也没有 project 子目录而只有 source 子目录。

2、 ppt 目录下是本书的电子课件，可作为教师教学参考、学生自学之用。

3、 pdf 目录下是本书的电子版本，可作为电子图书供读者在电脑上学习使用。

4、 pictures 目录下本书中比较大的图，是用 Microsoft Office Visio 2003 软件画的，目的是为了让教师更好地备课与上课。主要是第 5 章以后章节的部分图。

5、 有一个 stuinfo.txt 文件，是 30 位虚拟学生的信息，可根据实际需要进行增删，但必须修改相应的程序代码。

使用本书及光盘的工具

- Microsoft Visual Studio 2005（如果您想运行本书中的程序，那么您需要在计算机中安装它）；
- Microsoft Office PowerPoint 2003（如果您想使用 ppt 目录中的内容，那么您需要在计算机中安装它）；
- Microsoft Office Visio 2003（如果您想使用 pictures 目录中的内容，那么您需要在计算机中安装它）；
- Adobe Acrobat 7.0 Professional（如果您想使用 pdf 目录中的内容，那么您需要在计算机中安装它）；

致　谢

没有许多人的帮助，编者是不可能完成本书的。尤其要感谢下面这些人。

- 张应辉院长和胡锦德院长一直关注和支持可视化专业的专业改革。特别是胡院长，亲自指导了专业改革，并多次询问该书的进度并对其中的问题给予指示。如果没有二位领导，该书是不可能产生和完成的。
- 出版社的周凌波和郭朝晖老师为本书的修订和出版做了大量的工作。与他们的合作非常愉快，他们尽力使本书的东西通顺流畅。没有他们的工作，本书不可能出版。
- 最后，编者要感谢自己的家人。为了写这本书，编者投入了大量的时间和精力，牺牲了许多的周末和节假日。没有胥璐（编者的妻子）和段楚榆（编者的女儿）的支持，根本不可能有这本书的问世。多少次，编者都想花些时间陪伴家人，但都因为本书而放弃了。现在，本书总算告一段落，编者可以有更多时间幸福地听到女儿的笑声了。

尽管编者在写作过程中非常认真和努力，但由于编者水平有限，书中难免存在错误和不足之处，恳请广大读者批评指正。如果您对本书或光盘有什么意见、问题或想法，欢迎您通过下面的邮件通知编者，编者将不胜感激：

Email:duanez@neusoft.com

请在邮件的主题栏中注明：数据结构（C#）。

编者

2006 年 12 月

第 1 章 绪论.....1
1.1 数据结构.....1
1.1.1 学习数据结构的必要性.....1
1.1.2 基本概念和术语.....1
1.2 算法.....4
1.2.1 算法的特性.....4
1.2.2 算法的评价标准.....5
1.2.3 算法的时间复杂度.....6
1.3 数学预备知识.....7
1.3.1 集合.....7
1.3.2 常用的数学术语.....8
1.3.3 对数.....8
1.3.4 递归.....9
1.4 C#预备知识.....10
1.4.1 接口.....10
1.4.2 泛型编程.....13
本章小结.....20
习题一.....20
第 2 章 线性表.....22
2.1 线性表的逻辑结构.....22
2.1.1 线性表的定义.....22
2.1.2 线性表的基本操作.....22
2.2 顺序表.....24
2.2.1 顺序表的定义.....24
2.2.2 顺序表的基本操作实现.....29
2.2.3 顺序表应用举例.....35
2.3 单链表.....38
2.3.1 单链表的定义.....39
2.3.2 单链表的基本操作实现.....46
2.3.3 单链表应用举例.....56
2.4 其他链表.....61
2.4.1 双向链表.....61
2.4.2 循环链表.....64
2.5 C#中的线性表.....64
本章小结.....67
习题二.....67
第 3 章 栈和队列.....69
3.1 栈.....69
3.1.1 栈的定义及基本运算.....69
3.1.2 栈的存储和运算实现.....70
3.1.3 栈的应用举例.....82
3.1.4 C#中的栈.....87
3.2 队列.....87
3.2.1 队列的定义及基本运算.....87

3.2.2 队列的存储和运算实现.....89
3.2.3 队列的应用举例.....103
3.2.4 C# 中的队列.....104
本章小结.....105
习题三.....105
第 4 章 串和数组.....106
4.1 串.....106
4.1.1 串的基本概念.....106
4.1.2 串的存储及类定义.....106
4.1.3 串的基本操作的实现.....111
4.1.4 C#中的串.....115
4.2 数组.....117
4.2.1 数组的逻辑结构.....117
4.2.2 数组的内存映象.....118
4.2.3 C#中的数组.....119
本章小结.....121
习题四.....121
第 5 章 树和二叉树.....123
5.1 树.....123
5.1.1 树的定义.....123
5.1.2 树的相关术语.....124
5.1.3 树的逻辑表示.....125
5.1.4 树的基本操作.....126
5.2 二叉树.....126
5.2.1 二叉树的定义.....127
5.2.2 二叉树的性质.....128
5.2.3 二叉树的存储结构.....129
5.2.4 二叉链表存储结构的类实现.....132
5.2.5 二叉树的遍历.....137
5.3 树与森林.....141
5.3.2 树、森林与二叉树的转换.....144
5.3.3 树和森林的遍历.....147
5.4 哈夫曼树.....147
5.4.1 哈夫曼树的基本概念.....147
5.4.2 哈夫曼树类的实现.....149
5.4.3 哈夫曼编码.....153
5.5 应用举例.....154
5.6 C#中的树.....157
本章小结.....158
习题五.....159
第 6 章 图.....161
6.1 图的基本概念.....161
6.1.1 图的定义.....161
6.1.2 图的基本术语.....161

6.1.3 图的基本操作.....165
6.2 图的存储结构.....166
6.2.1 邻接矩阵.....167
6.2.2 邻接表.....172
6.3 图的遍历.....185
6.3.1 深度优先遍历.....185
6.3.2 广度优先遍历.....188
6.4 图的应用.....189
6.4.1 最小生成树.....189
6.4.2 最短路径.....199
6.4.3 拓扑排序.....207
本章小结.....210
习题六.....210
第 7 章 排序.....213
7.1 基本概念.....213
7.2 简单排序方法.....214
7.2.1 直接插入排序.....214
7.2.2 冒泡排序.....216
7.2.3 简单选择排序.....217
7.3 快速排序.....219
7.4 堆排序.....222
7.5 归并排序.....230
7.6 基数排序.....232
7.6.1 多关键码排序.....232
7.6.2 链式基数排序.....233
7.7 各种排序方法的比较与讨论.....235
7.8 C#中排序方法.....235
本章小结.....236
习题七.....236
第 8 章 查找.....238
8.1 基本概念和术语.....238
8.2 静态查找表.....238
8.2.1 顺序查找.....238
8.2.2 有序表的折半查找.....239
8.2.3 索引查找.....242
8.3 动态查找表.....243
8.4 哈希表.....252
8.4.1 哈希表的基本概念.....252
8.4.2 常用的哈希函数构造方法.....253
8.4.3 处理冲突的方法.....254
8.5 C#中的查找方法.....256
本章小结.....256
习题八.....256
参考文献.....257

第1章 绪论

数据是外部世界信息的计算机化，是计算机加工处理的对象。运用计算机处理数据时，必须解决四个方面的问题：一是如何在计算机中方便、高效地表示和组织数据；二是如何在计算机存储器（内存和外存）中存储数据；三是如何对存储在计算机中的数据进行操作，可以有哪些操作，如何实现这些操作以及如何对同一问题的不同操作方法进行评价；四是必须理解每种数据结构的性能特征，以便选择一个适合于某个特定问题的数据结构。这些问题就是数据结构这门课程所要研究的主要问题。本章首先说明学习数据结构的必要性和本书的目的，然后解释数据结构及其有关概念，接着讨论算法的相关知识，最后简单介绍本书所要用到的相关数学知识和 C#知识。

1.1数据结构

1.1.1 学习数据结构的必要性

我们知道，虽然每个人都懂得英语的语法与基本类型，但是对于同样的题目，每个人写出的作文，水平却高低不一。程序设计也和写英语作文一样，虽然程序员都懂得语言的语法与语义，但是对于同样的问题，程序员写出来的程序不一样。有的人写出来的程序效率很高，有的人却用复杂的方法来解决一个简单的问题。

当然，程序设计水平的提高仅仅靠看几本程序设计书是不行的。只有多思索、多练习，才能提高自己的程序设计水平；否则，书看得再多，提高也不大。记得刚学程序设计时，常听人说程序设计水平要想提高，最重要的是多看别人写的程序，多去思考问题。从别人写的程序中，我们可以发现效率更高的解决方法；从思考问题的过程中，我们可以了解解决问题的方法常常不只一个。运用先前解决问题的经验，来解决更复杂更深入的问题，是提高程序设计水平的最有效途径。

数据结构正是前人在思索问题的过程中所想出的解决方法。一般而言，在学习程序设计一段时间后，学习“数据结构”便能让你的程序设计水平上一个台阶。如果只学会了程序设计的语法和语义，那么你只能解决程序设计三分之一的问题，而且运用的方法并不是最有效的。但如果学会了数据结构的概念，就能在程序设计上，运用最有效的方法来解决绝大多数的问题。

《数据结构》这门课程的目的有三个。第一个是讲授常用的数据结构，这些数据结构形成了程序员基本数据结构工具箱(toolkit)。对于许多常见的问题，工具箱里的数据结构是理想的选择。就像.NET Framework 中 Windows 应用程序开发中的工具箱，程序员可以直接拿来或经过少许的修改就可以使用，非常方便。第二个是讲授常用的算法，这和数据结构一样，是人们在长期实践过程中的总结，程序员可以直接拿来或经过少许的修改就可以使用。可以通过算法训练来提高程序设计水平。第三个目的是通过程序设计的技能训练促进程序员综合能力的提高。

1.1.2 基本概念和术语

在本小节中，将对一些常用的概念和术语进行介绍，这些概念和术语在以后的章节中会多次出现。

1、数据(Data)

数据是外部世界信息的载体，它能够被计算机识别、存储和加工处理，是计算机程序加工的原料。计算机程序处理各种各样的数据，可以是数值数据，如整数、实数或复数；也可以是非数值数据，如字符、文字、图形、图像、声音等。

2、数据元素(Data Element)和数据项(Data Item)

数据元素是数据的基本单位，在计算机程序中通常被作为一个整体进行考虑和处理。数据元素有时也被称为元素、结点、顶点、记录等。一个数据元素可由若干个数据项(Data Item)组成。数据项是不可分割的、含有独立意义的最小数据单位，数据项有时也称为字段(Field)或域(Domain)。例如，在数据库信息处理系统中，数据表中的一条记录就是一个数据元素。这条记录中的学生学号、姓名、性别、籍贯、出生年月、成绩等字段就是数据项。数据项分为两种，一种叫做初等项，如学生的性别、籍贯等，在处理时不能再进行分割；另一种叫做组合项，如学生的成绩，它可以再分为数学、物理、化学等更小的项。

3、数据对象(Data Object)

数据对象是性质相同的数据元素的集合，是数据的一个子集。例如，整数数据对象是{0,±1,±2,±3,…}，字符数据对象是{a,b,c,…}。

4、数据类型(Data Type)

数据类型是高级程序设计语言中的概念，是数据的取值范围和对数据进行操作的总和。数据类型规定了程序中对象的特性。程序中的每个变量、常量或表达式的结果都应该属于某种确定的数据类型。例如，C#语言中的字符串类型（String，经常写为 string）。一个 String 表示一个恒定不变的字符序列集合，所有的字符序列集合构成 String 的取值范围。我们可以对 String 进行求长度、复制、连接两个字符串等操作。

数据类型可分为两类：一类是非结构的原子类型，如 C#语言中的基本类型（整型、实型、字符型等）；另一类是结构类型，它的成分可以由多个结构类型组成，并可以分解。结构类型的成分可以是非结构的，也可以是结构的。例如，C#语言中数组的成分可以是整型等基本类型，也可以是数组等结构类型。

5、数据结构(Data Structure)

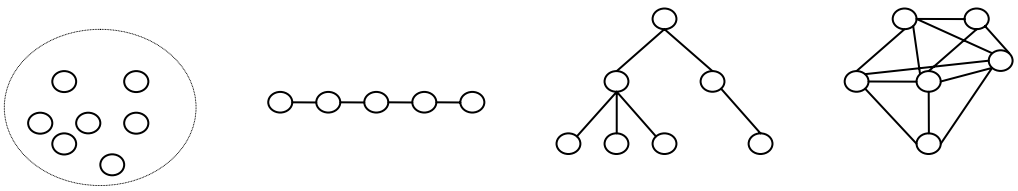
数据结构是相互之间存在一种或多种特定关系的数据元素的集合。在任何问题中，数据元素之间都不是孤立的，而是存在着一定的关系，这种关系称为结构(Structure)。根据数据元素之间关系的不同特性，通常有 4 类基本数据结构：

(1) 集合(Set)：如图 1.1(a)所示，该结构中的数据元素除了存在“同属于一个集合”的关系外，不存在任何其它关系。

(2) 线性结构(Linear Structure)：如图 1.1(b)所示，该结构中的数据元素存在着一对一的关系。

(3) 树形结构(Tree Structure)：如图 1.1(c)所示，该结构中的数据元素存在着一对多的关系。

(4) 图状结构(Graphic Structure)：如图 1.1(d)所示，该结构中的数据元素存在着多对多的关系。



(a) 集合　(b) 线性结构　(c) 树形结构　(d) 图状结构

图 1.1　4 类基本数据结构关系图

由于集合中的元素的关系极为松散，可用其它数据结构来表示，所以本书不做专门介绍。关于集合的概念在 1.3.1 小节中有介绍。

数据结构的形式化定义为：

数据结构(Data Structure)简记为 DS，是一个二元组，

DS = (D,R)

其中：D 是数据元素的有限集合，

R 是数据元素之间关系的有限集合。

下面通过例题来进一步理解后 3 类数据结构。

【例1-1】 学生信息表（如表 1.1 所示.）是一个线性的数据结构，表中的每一行是一个记录（在数据库信息处理系统中，表中的一个数据元素称为一个记录）。一条记录由学号、姓名、行政班级、性别和出生年月等数据项组成。表中数据元素之间的关系是一对一的关系。

表 1.1 学生信息表

学号	姓名	行政班级	性别	出生年月
040303001	雷洪	软件 04103	男	1986.12
040303002	李春	软件 04103	女	1987.3
040303003	周刚	软件 04103	男	1986.9

【例1-2】 家族关系是典型的树形结构，图 1.2 是一个三代的家族关系。在图中，爷爷、儿子、女儿、孙子、孙女或外孙女是一个结点（在树形结构中，数据元素称为结点），他们之间是一对多的关系。其中，爷爷有两个儿子和一个女儿，这是一对三的关系；一个儿子有两个儿子（爷爷的孙子），这是一对二的关系；另一个儿子有一个儿子（爷爷的孙子）和一个女儿（爷爷的孙女），这是一对二的关系；女儿有三个女儿（爷爷的外孙女），这是一对三的关系。树形结构具有严格的层次关系，爷爷在树形结构的最上层，中间层是儿子和女儿，最下层是孙子、孙女和外孙女。不能把这种关系倒过来，因为绝对不会先有儿子或女儿再有爷爷，也不会先有孙子或孙女再有儿子、先有外孙女再有女儿。

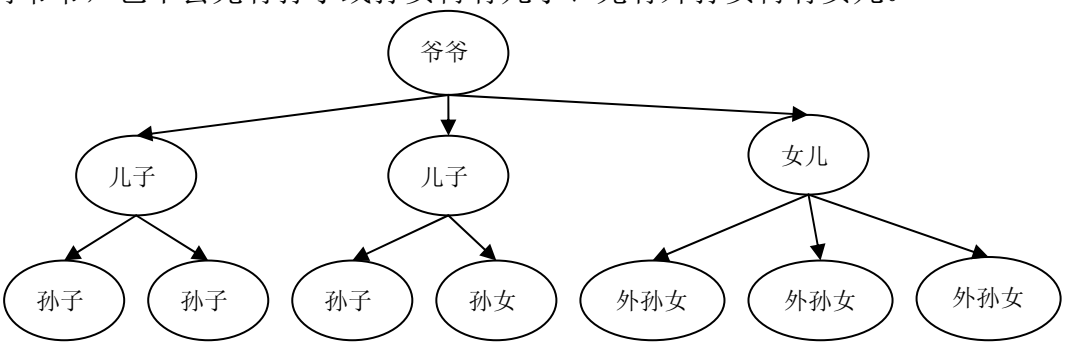


图 1.2 家族关系图

【例1-3】 图 1.3 是四个城市的公路交通图，这是一个典型的图状结构。在图中，每个城市是一个顶点（在图状结构中，数据元素称为顶点），它们之间是多对多的关系。成都与都江堰、雅安直接通公路，都江堰与成都、青城山直接通公路，青城山与都江堰、成都及雅安直接通公路，雅安与成都、青城山直接通公路。这些公路构成了一个公路交通网，所以，又把图状结构称为网状结构(Network Structure)

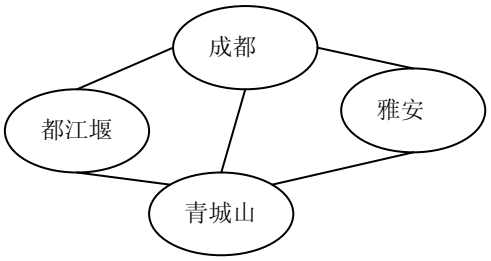


图 1.3 四城市交通图

从数据类型和数据结构的概念可知，二者的关系非常密切。数据类型可以看作是简单的数据结构。数据的取值范围可以看作是数据元素的有限集合，而对数据进行操作的集合可以看作是数据元素之间关系的集合。

数据结构包括数据的逻辑结构和物理结构。上述数据结构的定义就是数据的逻辑结构(Logic Structure)，数据的逻辑结构是从具体问题抽象出来的数学模型，是为了讨论问题的方便，与数据在计算机中的具体存储没有关系。然而，我们讨论数据结构的目的是为了在计算机中实现对它的操作，因此还需要研究在计算机中如何表示和存储数据结构，即数据的物理结构(Physical Structure)。数据的物理结构又称为存储结构(Storage Structure)，是数据在计算机中的表示（又叫映像）和存储，包括数据元素的表示和存储以及数据元素之间关系的表示和存储。

数据的存储结构包括顺序存储结构和链式存储结构两种。顺序存储结构(Sequence Storage Structure)是通过数据元素在计算机存储器中的相对位置来表示出数据元素的逻辑关系，一般把逻辑上相邻的数据元素存储在物理位置相邻的存储单元中。在 C#语言中用数组来实现顺序存储结构。因为数组所分配的存储空间是连续的，所以数组天生就具有实现数据顺序存储结构的能力。链式存储结构(Linked Storage Structure)对逻辑上相邻的数据元素不要求其存储位置必须相邻。链式存储结构中的数据元素称为结点(Node)，在结点中附设地址域(Address Domain)来存储与该结点相邻的结点的地址来实现结点间的逻辑关系。这个地址称为引用(Reference)，这个地址域称为引用域(Reference Domain)。

从 20 世纪 60 年代末到 70 年代初，出现了大型程序，软件也相对独立，人们越来越重视数据结构，认为程序设计的实质是确定数据结构，加上设计一个好的算法，这就是人们常说的“程序=数据结构+算法”。下一节谈谈算法的问题。

1.2 算法

从上节我们知道，算法与数据结构和程序的关系非常密切。进行程序设计时，先确定相应的数据结构，然后再根据数据结构和问题的需要设计相应的算法。由于篇幅所限，下面只从算法的特性、算法的评价标准和算法的时间复杂度等三个方面进行介绍。

1.2.1 算法的特性

算法(Algorithm)是对某一特定类型的问题的求解步骤的一种描述，是指令的有限序列。其中的每条指令表示一个或多个操作。一个算法应该具备以下 5 个特性：

1、有穷性(Finity)：一个算法总是在执行有穷步之后结束，即算法的执行时间是有限的。

2、确定性(Unambiguousness)：算法的每一个步骤都必须有确切的含义，即无二义，并且对于相同的输入只能有相同的输出。

3、输入(Input)：一个算法具有零个或多个输入。它即是在算法开始之前给出的

量。这些输入是某数据结构中的数据对象。

4、 输出(Output)：一个算法具有一个或多个输出，并且这些输出与输入之间存在着某种特定的关系。

5、 能行性(realizability)：算法中的每一步都可以通过已经实现的基本运算的有限次运行来实现。

算法的含义与程序非常相似，但二者有区别。一个程序不一定满足有穷性。例如操作系统，只要整个系统不遭破坏，它将永远不会停止。还有，一个程序只能用计算机语言来描述，也就是说，程序中的指令必须是机器可执行的，而算法不一定用计算机语言来描述，自然语言、框图、伪代码都可以描述算法。

在本书中我们尽可能采用 C#语言来描述和实现算法，使读者能够阅读或上机执行，以便更好地理解算法。

1.2.2 算法的评价标准

对于一个特定的问题，采用的数据结构不同，其设计的算法一般也不同，即使在同一种数据结构下，也可以采用不同的算法。那么，对于解决同一问题的不同算法，选择哪一种算法比较合适，以及如何对现有的算法进行改进，从而设计出更适合于数据结构的算法，这就是算法评价的问题。评价一个算法优劣的主要标准如下：

1、正确性(Correctness)。算法的执行结果应当满足预先规定的功能和性能的要求，这是评价一个算法的最重要也是最基本的标准。算法的正确性还包括对于输入、输出处理的明确而无歧义的描述。

2、可读性(Readability)。算法主要是为了人阅读和交流，其次才是机器的执行。所以，一个算法应当思路清晰、层次分明、简单明了、易读易懂。即使算法已转变成机器可执行的程序，也需要考虑人能较好地阅读理解。同时，一个可读性强的算法也有助于对算法中隐藏错误的排除和算法的移植。

3、健壮性(Robustness)。一个算法应该具有很强的容错能力，当输入不合法的数据时，算法应当能做适当的处理，使得不至于引起严重的后果。健壮性要求表明算法要全面细致地考虑所有可能出现的边界情况和异常情况，并对这些边界情况和异常情况做出妥善的处理，尽可能使算法没有意外的情况发生。

4、运行时间(Running Time)。运行时间是指算法在计算机上运行所花费的时间，它等于算法中每条语句执行时间的总和。对于同一个问题如果有多个算法可供选择，应尽可能选择执行时间短的算法。一般来说，执行时间越短，性能越好。

5、占用空间(Storage Space)。占用空间是指算法在计算机上存储所占用的存储空间，包括存储算法本身所占用的存储空间、算法的输入及输出数据所占用的存储空间和算法在运行过程中临时占用的存储空间。算法占用的存储空间是指算法执行过程中所需要的最大存储空间，对于一个问题如果有多个算法可供选择，应尽可能选择存储量需求低的算法。实际上，算法的时间效率和空间效率经常是一对矛盾，相互抵触。我们要根据问题的实际需要进行灵活的处理，有时需要牺牲空间来换取时间，有时需要牺牲时间来换取空间。

通常把算法在运行过程中临时占用的存储空间的大小叫算法的空间复杂度(Space Complexity)。算法的空间复杂度比较容易计算，它主要包括局部变量所占用的存储空间和系统为实现递归所使用的堆栈占用的存储空间。

如果算法是用计算机语言来描述的，还要看程序代码量的大小。对于同一个问题，在用上面 5 条标准评价的结果相同的情况下，代码量越少越好。实际上，代码量越大，占用的存储空间会越多，程序的运行时间也可能越长，出错的可能

性也越大，阅读起来也越麻烦。

在以上标准中，本书主要考虑程序的运行时间，也考虑执行程序所占用的空间。影响程序运行时间的因素很多，包括算法本身、输入的数据以及运行程序的计算机系统等。计算机的性能由以下因素决定：

1、硬件条件。包括所使用的处理器的类型和速度（比如，使用双核处理器还是单核处理器）、可使用的内存（缓存和 RAM）以及可使用的外存等。

2、实现算法所使用的计算机语言。实现算法的语言级别越高，其执行效率相对越低。

3、所使用的语言的编译器/解释器。一般而言，编译的执行效率高于解释，但解释具有更大的灵活性。

4、所使用的操作系统软件。操作系统的功能主要是管理计算机系统的软件和硬件资源，为计算机用户方便使用计算机提供一个接口。各种语言处理程序如编译程序、解释程序等和应用程序都在操作系统的控制下运行。

1.2.3 算法的时间复杂度

一个算法的时间复杂度(Time Complexity)是指该算法的运行时间与问题规模的对应关系。一个算法是由控制结构和原操作构成的，其执行的时间取决于二者的综合效果。为了便于比较同一问题的不同算法，通常把算法中基本操作重复执行的次数（频度）作为算法的时间复杂度。算法中的基本操作一般是指算法中最深层循环内的语句，因此，算法中基本操作语句的频度是问题规模n的某个函数f(n)，记作：T(n)=0(f(n))。其中“0”表示随问题规模n的增大，算法执行时间的增长率和f(n)的增长率相同，或者说，用“0”符号表示数量级的概念。例如，如$T(n)=\frac{1}{2}n(n-1)$，则 $\frac{1}{2}n(n-1)$的数量级与n^2相同，所以T(n)=0（n^2）。

如果一个算法没有循环语句，则算法中基本操作的执行频度与问题规模n无关，记作0(1)，也称为常数阶。如果算法只有一个一重循环，则算法的基本操作的执行频度与问题规模n呈线性增大关系，记作0（n），也叫线性阶。常用的还有平方阶0（n^2）、立方阶0（n^3）、对数阶0($\log_2 n$)等。

下面举例来说明计算算法时间复杂度的方法。

【例1-4】 分析以下程序的时间复杂度。

```
x=n;                    /*n>1*/
y=0;
while(y < x)
{
    y=y+1;              ①
}
```

解：这是一重循环的程序，while 循环的循环次数为 n，所以，该程序段中语句①的频度是 n，则程序段的时间复杂度是 T(n)=0（n）。

【例1-5】 分析以下程序的时间复杂度。

```
for(i=1; i<n; ++i) {
    for(j=0; j<n; ++j)
    {
        A[i][j]=i*j;        ①
    }
}
```

解：这是二重循环的程序，外层for循环的循环次数是n，内层for循环的循环次数为n，所以，该程序段中语句①的频度为n*n，则程序段的时间复杂度为T(n)=0（n^2）。

【例1-6】 分析以下程序的时间复杂度。

```
x=n;                    /*n>1*/
y=0;
while(x >= (y+1)*(y+1))
{
    y=y+1;                          ①
}
```

解：这是一重循环的程序，while 循环的循环次数为$\sqrt{n}$，所以，该程序段中语句①的频度是$\sqrt{n}$，则程序段的时间复杂度是 T(n)=0（$\sqrt{n}$）。

【例1-7】 分析以下程序的时间复杂度。

```
for(i=0;i<m;i++)
{
     for(j=0;j<t;j++)
     {
         for(k=0;k<n;k++)
         {
             c[i][j]=c[i][j]+a[i][k]*b[k][j];          ①
         }
      }
}
```

解：这是三重循环的程序，最外层 for 循环的循环次数为 m，中间层 for 循环的循环次数为 t，最里层 for 循环的循环次数为 t，所以，该程序段中语句①的频度是 m*n*t，则程序段的时间复杂度是 T(n)=0（m*n*t）。

1.3数学预备知识

本小节和下面的 1.4 小节介绍了本书所要用到的数学和 C#语言的相关知识，如果没有学习过这两节的相关内容，可以先进行学习然后进入后续章节的学习，也可以等到在后面的章节中遇到不熟悉的数学知识和 C#语言的知识，才回过头来学习这两个小节的相关内容。如果已经熟悉这两个小节的内容，可跳过，直接进入第二章的学习。虽然是把数学知识和 C#语言知识分开介绍，但在介绍有关内容时，为了讲授和学习的方便，会把两方面的内容结合起来讲。

1.3.1 集合

1、集合的概念

集合(Set)是由一些确定的、彼此不同的成员(Member)或者元素(Element)构成的一个整体。成员取自一个更大的范围，称为基类型(Base Type)。集合中成员的个数称为集合的基数(Cardinality)。

例如，集合 R 由整数 3、4、5 组成，写成 R={3，4，5}。此时 R 的成员是 3、4、5，R 的基类型是整型，R 的基数是 3。依赖于集合的基类型，它的成员经常有一个线性顺序。

集合的每个成员或者是基类型的一个基本元素(Base Element)，或者它本身也

是一个集合。我们把是集合的成员叫做该集合的子集(Subset)，子集中的每个成员都属于该集合。没有元素的集合称为空集(Empty Set,又称为 Null Set)，记作Φ。如上例中，3 是 R 的成员，记为：3∈R，6 不是 R 的成员，记为：6∉R。{3，4}是 R 的子集。

2、 集合的表示法

1) 穷举法：S＝{2，4，6，8，10}；
2) 描述法：S＝{x|x 是偶数，且 0≤x≤10}。

3、集合的特性

1) 确定性：任何一个对象都能被确切地判断是集合中的元素或不是；
2) 互异性：集合中的元素不能重复；
3) 无序性：集合中元素与顺序无关。

1.3.2 常用的数学术语

计量单位(Unit)：按照IEEE规定的表示法标准，字节缩写为“B”，位缩写为“b”，兆字节(2^{20}字节)缩写为缩写为“MB”，千字节(2^{10}字节)缩写为“KB”。

阶乘函数(Factorial Function)：阶乘函数 n!是指从 1 到 n 之间所有整数的连乘，其中 n 为大于 0 的整数。因此，5!=1·2·3·4·5=120。特别地，0!=1。

取下整和取上整(Floor and Ceiling)：实数 x 的取下整函数(Floor)记为⌊x⌋，返回不超过 x 的最大整数。例如，⌊3.4⌋=3，与⌊3.0⌋的结果相同。实数 x 的取上整函数(Ceiling)记为⌈x⌉，返回不小于 x 的最小整数。例如，⌈3.4⌉=4，与⌈4.0⌉的结果相同。

取模操作符（Modulus）：取模函数返回整除后的余数,有时称为求余。在 C#语言中取模操作符的表示为 n%m。从余数的定义可知，n%m 得到一个整数，满足 n=qm+r，其中 q 为一个整数，且 0≤r＜m。

1.3.3 对数

一般地，如果a(a＞0，a≠1)的b次幂等于N，就是ab=N，那么数b叫做以a为底N的对数（Logarithm），记作$\log_a N=b$，其中a叫做对数的底数，N叫做真数。

从定义可知，负数和零没有对数。事实上，因为 a＞0，所以不论 b 是什么实数，都有 ab＞0，这就是说不论 b 是什么数，N 永远是正数，因此负数和零没有对数。

编程人员经常使用对数，它有两个用途。第一，许多程序需要对一些对象进行编码，那么表示n个编码至少需要多少位呢？答案是⌈$\log_2 n$⌉。例如，如果要存储 1000 个不同的编码，至少需要⌈$\log_2 1000$⌉=10 位（10 位可以产生 1024 个不同的可用编码）。第二，对数普遍用于分析把问题分解为更小子问题算法。在一个线性表中查找指定值所使用的折半查找算法就是这样一种算法。折半查找算法首先与中间元素进行比较，以确定下一步是在上半部分进行查找还是在下半部分进行查找。然后继续将适当的子表分半，直到找到指定的值（折半查找算法在 8.2.3 小节有详细的描述）。一个长度为n的线性表被促逐次分半，直到最后的子表中只有一个元素，一共需要进行多少次呢？答案是$\log_2 n$次。

本书中用到的对数几乎都以 2 为底，这是因为数据结构和算法总是把事情一

分为二，或者用二进制位来存储编码。

1.3.4 递归

一个算法调用自己来完成它的部分工作，在解决某些问题时，一个算法需要调用自身。如果一个算法直接调用自己或间接地调用自己，就称这个算法是递归的(Recursive)。根据调用方式的不同，它分为直接递归(Direct Recursion)和间接递归(Indirect Recursion)。

比如，在收看电视节目时，如果演播室中也有一台电视机播放的是与当前相同的节目，观众就会发现屏幕里的电视套有一层层的电视画面。这种现象类似于直接递归。

如果把两面镜子面对面摆放，便可从任意一面镜子里看到两面镜子无数个影像，这类似于间接递归。

一个递归算法必须有两个部分：初始部分(Base Case)和递归部分(Recursion Case)。初始部分只处理可以直接解决而不需要再次递归调用的简单输入。递归部分包含对算法的一次或多次递归调用，每一次的调用参数都在某种程度上比原始调用参数更接近初始情况。

函数的递归调用可以理解为:通过一系列的自身调用,达到某一终止条件后，再按照调用路线逐步返回。递归是程序设计中强有力的工具，有很多数学函数是以递归来定义的。

如大家熟悉的阶乘函数，我们可以对n!作如下定义：

$$factorial(n)=\begin{cases}1 & n=0\\ n*factorial(n-1) & n>0\end{cases}$$

根据定义，如要计算 n!（factorial(n)），需要先调用 factorial(n-1)计算(n-1)!,而要计算(n-1)!需要先调用 factorial(n-2)计算(n-2)!,以此类推，最终需要调用 factorial(0)计算 0!，然后程序逐步返回，即可计算出 n!。

阶乘函数的C#语言实现如下。

```
public static long fact(int n)
{
    if(n <= 1)
    {
        return 1;
    }
    else
    {
        return n * fact(n-1);
    }
}
```

把递归作为一种主要用于设计和描述简单算法的工具,对于不熟悉它的编程人员而言是很难接受的。递归算法通常不是解决问题最有效的计算机程序，因为递归包含函数调用，函数调用需要时空开销。所以，递归比其他替代选择诸如

while 循环等，所花费的代价更大。但是，递归通常提供了一种能合理有效地解决某些问题的算法，如后面第 5 章中有关树的遍历问题。

1.4 C#预备知识

1.4.1 接口

1、 接口的定义

接口(Interface)定义为一个约定，实现接口的类或结构必须遵守该约定。简单的说，接口是类之间交互时遵守的一个协议。最初接触“类与类之间通过接口交互”这个概念时，误以为接口就是类公开的方法，类之间通过类的方法进行交互。其实接口是独立于类的一个定义，定义了类之间交互的标准。

那么类与类之间直接交互就好了，为什么还要使用接口呢？

在程序设计过程中，如果将一个对象看作多个类型将是非常有用的，因为对象的类型描述了它的能力和行为。比如，设计一个 SortedList 类型来保存一个有序对象集合。只要对象的类型支持将它和其他类型的对象比较的能力，便可将任何继承自 System.Object 类型的对象添加到 SortedList 中。也就是说，希望 SortedList 接受的类型继承自某个假想的 System.Comparable 类型。但是，许多现存的类型并不是继承自 System.Comparable 类型。这样，便不能将这些类型的对象添加到 SortedList 中，SortedList 类型也将变得不是很有用。

在理想情况下，一方面希望 SortedList 中对象的类型能够继承自现存的 System.Object 类型，另一方面，又可以将 SortedList 中对象的类型看作是从 System.Comparable 类型继承而来。这种将一个对象看成多个类型的能力通常称作多继承(Multiple Inheritance)。通用语言运行时(CLR)支持单实现继承和多接口继承。

单实现继承(Single Implementation Inheritance)是指一个类型只能有一个基类型。所以，单实现继承无法实现上述 SortedList 的功能。多接口继承(Multiple Interface Inheritance)是指一个类型可以继承多个接口，而接口是类之间相互交互的一个抽象(Abstract)，把类之间需要交互的内容抽象出来定义成接口，可以更好地控制类之间的逻辑交互。可见，接口内容的抽象好坏关系到整个程序的逻辑质量。另外，可以在任何时候通过开发附加接口和实现来添加新的功能。所以，多接口继承可以实现上述 SortedList 的功能。

关于接口一个很重要的概念是接口只包含成员定义，不包含成员的实现。接口不会继承自任何的 System.Object 派生类型。接口仅仅是一个包含着一组虚方法的抽象类型。成员的实现需要在继承的类或者结构中实现。接口的成员包括静态方法、索引器、常数、事件以及静态构造器等，不包含任何实例字段或实例构造器，所以，不能实例化一个接口。

实现接口的类必须严格按其定义来实现接口的每个成员。接口本身一旦被发布就不能再更改，对已发布的接口进行更改会破坏现有的代码。

根据约定，接口类型的名称要加一个大写的字母 I 前缀。接口定义允许使用修饰符，例如 public、protected、internal 以及 private 等，这和类与结构的定义一样。当然，大部分情况下使用 public。

下面简单介绍在本书用到的.NET 框架中的接口。

(1) IComparable 接口

IComparable 接口定义通用的比较方法。由类型使用的 IComparable 接口提供了一种比较多个对象的标准方式。如果一个类要实现与其它对象的比较，则必

须实现 IComparable 接口。由可以排序的类型，例如值类型实现以创建适合排序等目的类型特定的比较方法。

(2) IEnumerable 接口

IEnumerable 接口公开枚举数，该枚举数支持在集合上进行简单迭代。

IEnumerable 接口可由支持迭代内容对象的类实现。

(3) IEnumerator 接口

IEnumerator 接口支持在集合上进行简单迭代。是所有枚举数的基接口。枚举数只允许读取集合中的数据，枚举数无法用于修改基础集合。

(4) ICollection 接口

ICollection 接口定义所有集合的大小、枚举数和同步方法。ICollection 接口是 System.Collections 命名空间中类的基接口。

(5) IDictionary 接口

IDictionary 接口是基于 ICollection 接口的更专用的接口。IDictionary 实现是键/值对的集合，如 Hashtable 类。

(6) IList 接口

IList接口实现是可被排序且可按照索引访问其成员的值的集合，如ArrayList类。

.NET Framework 2.0 提供了泛型的接口，如 IComparable<T>、IEnumerable<T>、IEnumerator<T>、ICollection<T>、IDictionary<T>和IList<T>等。泛型接口的功能与非泛型接口的功能一样，只不过适用于更多的类。关于泛型的介绍见下一小节。

说到接口，这里要提及 1.1.2 小节讲到的 4 类数据结构中的集合。从集合的定义中，我们知道，集合中的数据元素除了有“同属于一个集合”的关系外，没有任何其它的关系。.NET 框架中的集合概念和数据结构中的集合概念不尽相同。.NET 框架中集合（Collection）定义如下：

从.NET 的角度看，所谓的集合可以定义为一种对象，这种对象提供了一种结构化组织任意对象的方式，并且实现一个或者多个 ICollection、IDictionary 和 System.Collections.IList 接口。这一定义把 System.Collections 名称空间中的“内置”集合划分成了三种类别：

(1) 有序集合：仅仅实现 ICollection 接口的集合，在通常情况下，其数据项的插入顺序控制着从集合中取出对象的顺序。System.Collections.Stack 和 System.Collections.Queue 类都是 ICollection 集合的典型例子。关于 Stack 和 Queue 将在第三章详细介绍。

(2) 索引集合：实现 Ilist 的集合，其内容能经由从零开始的数字检索取出，就象数组一样。System.Collections.ArrayList 类是索引集合的一个例子。关于 ArrayList 将在第二章详细介绍。

(3) 键式集合：实现 IDictionary 接口的集合，其中包含了能被某些类型的键值检索的项。IDictionary 集合的内容通常按键值方式存储，可以用枚举的方式排序检索。System.Collections.Hashtable 类实现了 IDictionary 接口。关于将 Hashtable 在第八章详细介绍。

给定集合的功能在很大程度上受到特定接口或其实现接口的控制。如果对面向对象编程缺乏了解，那么可能对上面说的这些话感到难以理解。不过，至少应该知道，用接口构造的对象不但具有整套类似方法的对象族，而且这些对象在必要的情况下可以被当作同类，这就是多态性(Polymorphism)。

同样，在.NET Framework 2.0 的 System.Collections.Generic 名称空间中提供了泛型的集合类。泛型集合类的功能与非泛型集合类的功能一样，只不过适用于更多的类。关于泛型的介绍见下一小节。

2、接口与抽象类

抽象类(Abstract Class)和接口在定义与功能上有很多相似的地方，在程序中选择使用抽象类还是接口需要比较抽象类和接口之间的具体差别。

抽象类是一种不能实例化而必须从中继承的类，抽象类可以提供实现，也可以不提供实现。子类只能从一个抽象类继承。抽象类应主要用于关系密切的对象。如果要设计大的功能单元或创建组件的多个版本，则使用抽象类。

接口是完全抽象的成员集合，不提供实现。类或者结构可以继承多个接口。接口最适合为不相关的类提供通用功能。如果要设计小而简练的功能块，则使用接口。接口一旦创建就不能更改，如果需要接口的新版本，必须创建一个全新的接口。

3、接口的实现

接口的实现分为隐式实现(Implicit Implementation)和显式实现(Explicit Implementaton)。如果类或者结构要实现的是单个接口，可以使用隐式实现，如果类或者结构继承了多个接口,那么接口中相同名称成员就要显式实现。显式实现是通过使用接口的完全限定名来实现接口成员的。

接口及该接口的 C#实现如下：

```
using System;
using System.Collections;

public interface IBook {
    void ShowBook();
    string GetTitle();
    int GetPages();
    void SetPages(int pages);
}

public class NewBook : IBook
{
    public string title;
    public int pages;
    public string author;

    public NewBook(string title, string author, int pages)
    {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }

    public string GetTitle()
    {
```

```
            return title;
        }

        public int GetPages()
        {
            return pages;
        }

        public void SetPages(int pages)
        {
            this.pages = pages;
        }

        public void ShowBook()
        {
            Console.WriteLine(“Title:{0}”, title);
            Console.WriteLine(“Author:{0}”, author);
            Console.WriteLine(“Pages:{0}”, pages);
        }
    }

    public class App
    {
        static void Main()
        {
            NewBook MyNovel = new NewBook(“China Dream”,
                                          “Robert”,500);
            MyNovel.ShowBook();
        }
    }
```

1.4.2 泛型编程

泛型(Generic Type)是.NET Framework 2.0 最强大的功能。泛型的主要思想就是将算法与数据结构完全分离开来，使得一次定义的算法能够作用于多种数据结构，从而实现高度可重用的开发。通过泛型可以定义类型安全的数据结构，而没有必要使用实际的数据类型。这将显著提高性能并得到更高质量的代码，因为可以重用数据处理算法，而没有必要复制类型特定的代码。

1、泛型问题陈述

在开发通用容器时，需要通用容器能存储各种类型的实例。在.NET Framework 1.1 下，必须使用基于 object 的容器。这意味着，在该容器中使用的数据类型是难以归类的 object，并且容器方法与 object 交互。

基于 object 的容器的 C#实现如下：

```
public class Container
{
    readonly int m_Size;                //容器的容量
```

```
int m_ContainerPointer ;          //容器指针，指示最后一个元素的位置
object[] m_Items;                 //容器数组，存放数据

//无参构造器
public Container () : this(100)
{
    m_ContainerPointer = -1;
    m_Size = 100;
}

//有参构造器
public Container (int size)
{
    m_Size = size;
    m_Items = new object[m_Size];
    m_ContainerPointer = -1;
}

//获取容器元素个数属性
public int Count
{
    get
    {
        return m_ContainerPointer;
    }
}

//容器是否为空
public bool IsEmpty
{
    get
    {
        return (m_ContainerPointer == -1);
    }
}

//容器是否已满
public bool IsFull
{
    get
    {
        return (m_ContainerPointer == m_Size - 1);
    }
}
```

```
        //在容器的尾部插入一个元素
        public void Insert(object item)
       {
            if (IsFull)
            {
                Console.WriteLine("Container is full!");
                return;
            }
            else if (IsEmpty)
            {
                m_Items[++m_ContainerPointer] = item;
            }
            else
           {
                m_Items[++m_ContainerPointer] = item;
           }
       }

       //从容器的尾部删除一个元素
       public object Delete()
      {
           if (m_ContainerPointer >= 0)
           {
              return m_Items[m_ContainerPointer--];
           }
           return null;
       }
   }
```

但是，基于 object 的容器存在以下问题。

(1) 性能问题。在使用值类型时，必须将值类型装箱(Boxing)以便推送和存储，并且在将值类型从容器中取出时将其取消装箱(Unboxing)。装箱和取消装箱都会根据值类型的权限造成重大的性能损失。而且，装箱和取消装箱操作还会增加托管堆上的压力，导致更多的垃圾收集工作，这对于性能而言也不好。即使是在使用引用类型而不是值类型时，仍然存在性能损失，因为必须强制地将 object 转换为需要的实际类型进行类型，造成强制类型转换开销，代码如下：

```
Container c = new Container();
c.Insert("1");
string number = (string)c.Delete();
```

(2) 类型安全。类型转换难以保证每次转换都是成功的，这将导致某些错误在编译时无法被检查出来，而在运行时发生异常。因为编译器允许在任何类型和 object 之间进行强制类型转换，所以将丢失编译时类型安全。例如，以下代码可以正确编译，但是在运行时将引发无效强制类型转换的异常。

```
Container c = new Container();
```

```
c.Insert(1);
```

下面这条语句能够编译，但不是类型安全的，将抛出一个异常。

```
string number = (string)c.Container();
```

解决上述两个问题的办法是提供类型特定的（因而是类型安全的）高性能容器来克服。

对于整型，可以实现并使用 IntContainer：

```
 public class IntContainer
{
        int[] m_Items;
        public void Insert(int item){…}

        public int Delete(){…}
}

IntContainer c = new IntContainer();
c.Insert(1);
int number = c.Delete();
```

对于字符串，可以实现 StringContainer：

```
 public class StringContainer
{
      string[] m_Items;
      public void Insert(string item){...}
      public string Delete(){...}
}

StringContainer c = new StringContainer ();
c.Insert("1");
string number = c.Delete();
```

虽然这解决了性能和类型安全问题，但引起了下一个同样严重的问题。

（3）工作效率。编写类型特定的数据结构是一项乏味、重复且易于出错的工作。并且，无法预知未知类型的使用情况，因此还必须保持基于 object 的数据结构。结果，大多数.NET Framework 1.1 开发人员发现类型特定的数据结构并不实用，还是选择使用基于 object 的数据结构，尽管它们存在缺点。

2、泛型的概念

通过泛型可以定义类型安全的并且对性能或工作效率无损害的类。泛型类的定义如下：

```
public class className<T>
```

它在普通类的定义后面增加了一个类型参数 T，该参数用一对分隔符“<>”括起来。类型参数表示对数据类型的抽象，可以把它理解为一个替换标记或者是一个占位符。它在类的定义代码中的每次出现都表示一个非特定的数据类型。

泛型容器的完整 C#实现如下：

```
public class Container<T>
{
      readonly int m_Size;              //容器的容量
```

```
    int m_ContainerPointer ;          //容器指针，指示最后一个元素的位置
    T[] m_Items;                      //容器数组，存放数据

    //构造器
    public Container():this(100)
    {
          m_ContainerPointer = -1;
          m_Size = 100;
    }

    //构造器
    public Container(int size)
   {
         m_Size = size;
         m_Items = new T[m_Size];
         m_ContainerPointer = -1;
   }

//获取容器元素个数属性
public int Count
{
     get
     {
          return m_ContainerPointer;
     }
}

    //容器是否为空
    public bool IsEmpty
   {
       get
       {
          return (m_ContainerPointer == -1);
       }
    }

    //容器是否已满
    public bool IsFull
   {
       get
       {
           return (m_ContainerPointer == m_Size - 1);
       }
    }
```

```
        //在容器的尾部插入一个元素
        public void Insert(object item)
       {
            if (IsFull)
            {
                Console.WriteLine("Container is full!");
                return;
            }
            else if (IsEmpty)
            {
                m_Items[++m_ContainerPointer] = item;
            }
            else
           {
                m_Items[++m_ContainerPointer] = item;
           }
       }

       //从容器的尾部删除一个元素
       public object Delete()
      {
           if (m_ContainerPointer >= 0)
           {
              return m_Items[m_ContainerPointer--];
           }
           return null;
       }
   }
```

由以上实现可知，在基于 object 的容器实现中使用 object 的地方在一般容器实现中都被替换成了 T.。

和普通类一样，泛型类可以拥有各种成员，包括非静态和静态的字段、方法、构造器、索引函数、委托和事件等。在泛型类的成员中可以自由地使用类型参数来指代数据类型，包括用来定义字段类型和方法成员的返回类型。传递给方法成员的参数类型，以及在方法成员的执行代码中定义局部变量的类型。

类型参数在作为传递给方法的参数使用时，既可以作为一般参数，也可以作为引用参数和输出参数，甚至可以是数组参数。

3、泛型实现

表面上，C# 泛型的语法看起来与 C++模板类似，但是编译器实现和支持它们的方式存在重要差异。与 C++模板相比，C#泛型可以提供增强的安全性，但是在功能方面也受到某种程度的限制。在一些 C++编译器中，在通过特定类型使用模板类之前，编译器甚至不会编译模板代码。当确实指定了类型时，编译器会以内联方式插入代码，并且将每个出现一般类型参数的地方替换为指定的类型。此外，每当使用特定类型时，编译器都会插入特定于该类型的代码，而不管是否

已经在应用程序中的其他某个位置为模板类指定了该类型。C++链接器负责解决该问题，并且并不总是有效。这可能会导致代码膨胀，从而增加加载时间和内存足迹。

在.NET Framework 2.0 中，泛型在 IL（中间语言）和 CLR 本身中具有本机支持。在编译泛型 C#代码时，首先编译器会将其编译为 IL，就像其他任何类型一样。但是，IL 只包含实际特定类型的参数或占位符，并有专用的 IL 指令支持泛型操作。泛型代码的元数据中包含泛型信息。

真正的泛型实例化工作以“on-demand”的方式，发生在 JIT 编译时。当进行 JIT 编译时，JIT 编译器用指定的类型实参来替换泛型 IL 代码元数据中的 T，进行泛型类型的实例化。这会向 JIT 编译器提供类型特定的 IL 元数据定义，就好像从未涉及到泛型一样。这样，JIT 编译器就可以确保方法参数的正确性，实施类型安全检查，甚至执行类型特定的 IntelliSense。

当.NET 将泛型 IL 代码编译为本机代码时，所产生的本机代码取决于指定的类型。如果本机指定的是值类型，则 JIT 编译器将泛型类型参数替换为特定的值类型，并且将其编译为本机代码。JIT 编译器跟踪已经生成的类型特定的 IL 代码。如果 JIT 编译器用已经编译为本机代码的值类型编译泛型 IL 代码，则只是返回对该 IL 代码的引用。因为 JIT 编译器在以后的所有场合中都将使用相同的值类型特定的 IL 代码，所以不存在代码膨胀问题。

如果本机指定的是引用类型，则 JIT 编译器将泛型 IL 代码中的泛型参数替换为 object，并将其编译为本机代码。在以后的任何针对引用类型而不是泛型类型参数的请求中，都将使用该代码。JIT 编译器只会重新使用实际代码。实例仍然按照它们离开托管堆的大小分配空间，并且没有强制类型转换。

4、泛型的好处

泛型使代码可以重用，类型和内部数据可以在不导致代码膨胀的情况下更改，而不管是值类型还是引用类型。可以一次性地开发、测试和部署代码，通过任何类型（包括将来的类型）来重用它，并且全部具有编译器支持和类型安全。因为泛型代码不会强行对值类型进行装箱和取消装箱，或者对引用类型进行向下强制类型转换，所以性能得到显著提高。对于值类型，性能通常会提高 200%；对于引用类型，在访问该类型时，可以预期性能最多提高 100%（当然，整个应用程序的性能可能会提高，也可能不会提高）。

5、泛型类与泛型接口

泛型类封装了不针对任何特定数据类型的操作。泛型类常用于容器类，如链表、哈希表、栈、队列、树等等。这些类中的操作，如对容器添加、删除元素，不论所存储的数据是何种类型，都执行几乎同样的操作。

对大多数情况，推荐使用.NET Framework 2.0 类库 System.Collections.Generic 中所提供的容器类。有关使用这些类的详细信息，请参见基础类库中的泛型。

通常，从一个已有的具体类来创建泛型类，并每次把一个类型改为类型参数，直至达到一般性和可用性的最佳平衡。

泛型接口是比普通接口更为抽象的数据类型。不论是为泛型容器类，还是表示容器中元素的泛型类，定义接口是很有用的。把泛型接口与泛型类结合使用是更好的用法，比如用 IComparable<T>而非 IComparable，以避免值类型上的装箱和拆箱操作。.NET Framework 2.0 类库定义了几个新的泛型接口，以配合 System.Collections.Generic 中新容器类的使用。

本书后面章节有许多有关泛型接口的例子，这里不再举例说明。

本章小结

本章首先介绍了数据结构及其相关的概念，包括数据、数据元素、数据项、数据对象、数据结构、数据的逻辑结构和物理结构等。数据结构是相互之间存在一种或多种特定关系的数据元素的集合。数据结构包括数据的逻辑结构和物理结构。数据的逻辑结构是从具体的问题中抽象出来的数学模型。数据的逻辑结构一般有 4 类：集合、线性结构、树形结构和图状结构。集合中的数据元素除了属于“同一集合这一关系外”，没有其它任何的关系。线性结构中的数据元素之间存在一对一的关系；树形结构中的数据元素之间存在一对多的关系；图状结构的数据元素之间存在多对多的关系。数据的逻辑结构分为两种类型：线性结构和非线性结构。树形结构和图状结构属于非线性结构。数据的物理结构又叫存储结构，是数据元素在计算机中的存储方式。存储结构有两类，顺序存储结构和链式存储结构。顺序存储结构是在计算机中把逻辑上相邻的数据元素存储在物理位置相邻的存储单元中。链式存储结构是逻辑上相邻的数据元素不一定存储在物理位置相邻的存储单元中，数据元素之间的逻辑关系用引用表示。

接着，本章介绍了算法的概念及相关知识。算法是对某一特定类型问题求解步骤的一种描述，是指令的有限序列。算法有五个特性：有穷性、确定性、输入、输出和能行性。算法的评价标准有五点：正确性、可读性、健壮性、运行时间和占用空间。程序实现的算法还要看程序的代码量。算法的时间复杂度是指该算法的运行时间与问题规模的对应关系，通常把算法中基本操作重复执行的次数（频度）作为算法的时间复杂度。

最后，本章简单介绍了本书要用到的数学知识和 C#语言知识。数学知识主要介绍了集合、常用的数学用语、对数和递归。C#语言知识主要介绍了接口和泛型编程。

习题一

1.1 简述下列术语。

数据元素　数据项　数据结构　数据类型　数据逻辑结构　数据存储结构　算法。

1.2 数据结构课程的主要目的是什么？

1.3 分别画出线性结构、树形结构和图状结构的逻辑示意图。

1.4 算法的特性是什么？评价算法的标准是什么？

1.5 什么是算法的时间复杂度？怎样表示算法的时间复杂度？

1.6 分析下面语句段执行的时间复杂度。

(2)
```
for (int i=0; i<n; ++i)
{
        ++p;
}
```

(3)
```
for (int i<0; i<n; ++i)
 {
      for (int j =0; j<m; ++j)
      {
          ++p;
      }
 }
```

(4)
```
i = 1;
```

```
while(i <= n)
{
   i *= 3;
 }
```

(5)
```
int i = 1;
int k = 0;
do
{
        k = k+10*i;
        ++i;
 }
```

第2章 线性表

线性表是最简单、最基本、最常用的数据结构。线性表是线性结构的抽象(Abstract)，线性结构的特点是结构中的数据元素之间存在一对一的线性关系。这种一对一的关系指的是数据元素之间的位置关系，即：（1）除第一个位置的数据元素外，其它数据元素位置的前面都只有一个数据元素；（2）除最后一个位置的数据元素外，其它数据元素位置的后面都只有一个元素。也就是说，数据元素是一个接一个的排列。因此，可以把线性表想象为一种数据元素序列的数据结构。

本书在介绍各种数据结构时，先介绍数据结构的逻辑结构，包括定义、基本操作。然后介绍数据结构的存储结构，先介绍顺序存储结构，再介绍链式存储结构。

2.1 线性表的逻辑结构

2.1.1 线性表的定义

线性表(List)是由 n(n≥0)个相同类型的数据元素构成的有限序列。对于这个定义应该注意两个概念：一是“有限”，指的是线性表中的数据元素的个数是有限的，线性表中的每一个数据元素都有自己的位置(Position)。本书不讨论数据元素个数无限的线性表。二是“相同类型”，指的是线性表中的数据元素都属于同一种类型。虽然有的线性表具有不同类型的数据元素，但本书中所讨论的线性表中的数据元素都属于同一类型。

线性表通常记为：$L=(a_1, a_2, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n)$，L是英文单词list的第 1 个字母。L中包含n个数据元素，下标表示数据元素在线性表中的位置。a_1是线性表中第一个位置的数据元素，我们称作第一个元素。a_n是线性表中最后一个位置的数据元素，我们称作最后一个元素。n为线性表的表长，n=0 时的线性表被称为空表（Empty List）。

线性表中的数据元素之间存在着前后次序的位置关系，将a_{i-1}称为a_i的直接前驱，将a_i称为a_{i+1}的直接后继。除a_1外，其余元素只有一个直接前驱，因为a_1是第一个元素，所以它没有前驱。除a_n外，其余元素只有一个直接后继，因为a_n是最后一个元素，所以它没有后继。

线性表的形式化定义为：线性表(List)简记为 L，是一个二元组，

$$L = (D, R)$$

其中：D 是数据元素的有限集合。

R 是数据元素之间关系的有限集合。

在实际生活中线性表的例子很多。例如，1 到 100 的偶数就是一个线性表：

$$(2, 4, 6, \cdots, 100)$$

表中数据元素的类型是自然数。某个办公室的职员姓名（假设每个职员的姓名都不一样）也可以用一个线性表来表示：

（“zhangsan”,”lisi”,”wangwu”,”zhaoliu”,”sunqi”,”huangba”）

表中数据元素的类型为字符串。

在一个复杂的线性表中，一个数据元素是一个记录，由若干个数据项组成，含有大量记录的线性表又称文件(File)。例如，例子 1.1 中的学生信息表就是一个线性表，表中的每一行是一个记录。一个记录由学号、姓名、行政班级、性别和出生年月等数据项组成。

2.1.2 线性表的基本操作

在选择线性表的表示法之前，程序设计人员首先应该考虑这种表示法要支持的基本操作。从线性表的定义可知，一个线性表在长度上应该能够增长和缩短，也就是说，线性表最重要的操作应该能够在线性表的任何位置插入和删除元素。除此之外，应该可以有办法获得元素的值、读出这个值或者修改这个值。也应该可以生成和清除（或者重新初始化）线性表。

数据结构的运算是定义在逻辑结构层次上的，而运算的具体实现是建立在物理存储结构层次上的。因此，把线性表的操作作为逻辑结构的一部分，每个操作的具体实现只有在确定了线性表的存储结构之后才能完成。数据结构的基本运算不是它的全部运算，而是一些常用的基本运算，而每一个基本运算在实现时也可以根据不同的存储结构派生出一系列相关的运算来。

由于现在只考虑线性表的基本操作，所以以 C#接口的形式表示线性表，接口中的方法成员表示基本操作。并且，为了使线性表对任何数据类型都适用，数据元素的类型使用泛型的类型参数。在实际创建线性表时，元素的实际类型可以用应用程序中任何方便的数据类型来代替，比如用简单的整型或者用户自定义的更复杂的类型来代替。

线性表的接口如下所示。

```
public interface IListDS<T> {
    int GetLength();              //求长度
    void Clear();                 //清空操作
    bool IsEmpty();               //判断线性表是否为空
    void Append(T item);          //附加操作
    void Insert(T item, int i);   //插入操作
    T Delete(int i);              //删除操作
    T GetElem(int i);             //取表元
    int Locate(T value);          //按值查找
}
```

为了和.NET 框架中的接口 IList 相区分，在 IList 后面加了“DS”，“DS”表示数据结构。下面对线性表的基本操作进行说明。

1、求长度：GetLength()

初始条件：线性表存在；

操作结果：返回线性表中所有数据元素的个数。

2、清空操作：Clear()

初始条件：线性表存在且有数据元素；

操作结果：从线性表中清除所有数据元素，线性表为空。

3、判断线性表是否为空：IsEmpty()

初始条件：线性表存在；

操作结果：如果线性表为空返回 true，否则返回 false。

4、附加操作：Append(T item)

初始条件：线性表存在且未满；

操作结果：将值为 item 的新元素添加到表的末尾。

5、插入操作：Insert(T item, int i)

初始条件：线性表存在，插入位置正确()(1≤i≤n+1,n 为插入前的表长)。

操作结果：在线性表的第 i 个位置上插入一个值为 item 的新元素，这样使得原序号为 i,i+1,…,n 的数据元素的序号变为 i+1,i+2,…,n+1，插入后表长=原

表长+1。

6、删除操作：Delete(int i)

初始条件：线性表存在且不为空，删除位置正确(1≤i≤n,n 为删除前的表长)。

操作结果：在线性表中删除序号为 i 的数据元素，返回删除后的数据元素。删除后使原序号为 i+1,i+2,…,n 的数据元素的序号变为 i,i+1,…,n-1，删除后表长=原表长-1。

7、取表元：GetElem(int i)

初始条件：线性表存在，所取数据元素位置正确（1≤i≤n，n 为线性表的表长)；

操作结果：返回线性表中第 i 个数据元素。

8、按值查找：Locate(T value)

初始条件：线性表存在。

操作结果：在线性表中查找值为 value 的数据元素，其结果返回在线性表中首次出现的值为 value 的数据元素的序号，称为查找成功；否则，在线性表中未找到值为 value 的数据元素，返回一个特殊值表示查找失败。

实际上，在.NET 框架中，许多类的求长度、判断满和判断空等操作都用属性来表示，这里在接口中定义为方法是为了说明数据结构的操作运算。实际上，属性也是方法。在后面章节的数据结构如栈、队列等的处理也是如此，就不一一说明了。

2.2 顺序表

2.2.1 顺序表的定义

在计算机内，保存线性表最简单、最自然的方式，就是把表中的元素一个接一个地放进顺序的存储单元，这就是线性表的顺序存储(Sequence Storage)。线性表的顺序存储是指在内存中用一块地址连续的空间依次存放线性表的数据元素，用这种方式存储的线性表叫顺序表(Sequence List)，如图 2.1 所示。顺序表的特点是表中相邻的数据元素在内存中存储位置也相邻。

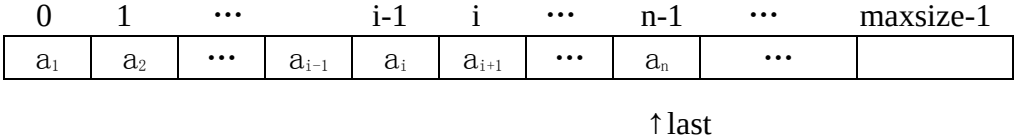


图 2.1 顺序表的存储结构示意图

假设顺序表中的每个数据元素占w个存储单元，设第i个数据元素的存储地址为Loc(a_i)，则有：

$$Loc(a_i)= Loc(a_1)+(i-1)*w \qquad 1\leq i\leq n$$

式中的Loc(a_1)表示第一个数据元素a_1的存储地址，也是顺序表的起始存储地址，称为顺序表的基地址(Base Address)。也就是说，只要知道顺序表的基地址和每个数据元素所占的存储单元的个数就可以求出顺序表中任何一个数据元素的存储地址。并且，由于计算顺序表中每个数据元素存储地址的时间相同，所以顺序表具有随机存取的特点。

C#语言中的数组在内存中占用的存储空间就是一组连续的存储区域，因此，数组具有随机存取的特点。所以，数组天生具有表示顺序表的数据存储区域的特性。

把顺序表看作是一个泛型类，类名叫 SeqList<T>。”Seq”是英文单词”Sequence”的前三个字母。SeqList<T>类实现了接口 IListDS<T>。用数组来存储顺序表中的元素，在 SeqList<T>类中用字段 data 来表示。由于经常需要在顺序表中插入或删除数据元素，所以要求顺序表的表长是可变的。因此，数组的容量需要设计得特别大，可以用 System.Array 的 Length 属性来表示。但为了说明顺序表的最大长度（顺序表的容量），在 SeqList<T>类中用字段 maxsize 来表示。maxsize 的值可以根据实际需要修改，这通过 SeqList<T>类中构造器的参数 size 来实现。顺序表中的元素由 data[0]开始依次顺序存放，由于顺序表中的实际元素个数一般达不到 maxsize，因此，在 SeqList<T>类中需要一个字段 last 表示顺序表中最后一个数据元素在数组中的位置。如果顺序表中有数据元素时，last 的变化范围是 0 到 maxsize-1，如果顺序表为空，last=-1。具体表示见图 2.1 所示。由于顺序表空间的限制，当往顺序表中插入数据元素需要判断顺序表是否已满，顺序表已满不能插入元素。所以，SeqList<T>类除了要实现接口 IListDS<T>中的方法外，还需要实现判断顺序表是否已满等成员方法。

顺序表类 SeqList<T>的实现说明如下所示。

```
public class SeqList<T> : IListDS<T> {
    private int maxsize;            //顺序表的容量
    private T[] data;               //数组，用于存储顺序表中的数据元素
    private int last;               //指示顺序表最后一个元素的位置

    //索引器
    public T this[int index]
    {
        get
        {
            return data[index];
        }
        set
        {
            data[index] = value;
        }
    }

    //最后一个数据元素位置属性
    public int Last
    {
        get
        {
            return last;
        }
    }

    //容量属性
    public int Maxsize
```

```
{
    get
    {
        return maxsize;
    }

    set
    {
        maxsize = value;
    }
}

//构造器
public SeqList(int size)
{
    data = new T[size];
    maxsize = size;
    last = -1;
}

//求顺序表的长度
public int GetLength()
{
    return last+1;
}

//清空顺序表
public void Clear()
{
    last = -1;
}

//判断顺序表是否为空
public bool IsEmpty()
{
    if (last == -1)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

```
//判断顺序表是否为满
public bool IsFull()
{
      if (last == maxsize-1)
      {
            return true;
      }
      else
      {
            return false;
      }
}

//在顺序表的末尾添加新元素
public void Append(T item)
{
      if(IsFull())
      {
            Console.WriteLine("List is full");
            return;
     }

     data[++last] = item;
}

//在顺序表的第i个数据元素的位置插入一个数据元素
public void Insert(T item, int i)
{
      if (IsFull())
      {
            Console.WriteLine("List is full");
            return;
      }

      if(i<1 || i>last+2)
      {
            Console.WriteLine("Position is error!");
            return;
      }

      if (i == last + 2)
      {
            data[last+1] = item;
```

```
        }
        else
        {
            for (int j = last; j>= i-1; --j)
            {
                data[j + 1] = data[j];
            }

            data[i-1] = item;
        }
        ++last;
    }

    //删除顺序表的第i个数据元素
    public T Delete(int i)
    {
        T tmp = default(T);
        if (IsEmpty())
        {
            Console.WriteLine("List is empty");
            return tmp;
        }

        if (i < 1 || i > last+1)
        {
            Console.WriteLine("Position is error!");
            return tmp;
        }

        if (i == last+1)
        {
            tmp = data[last--];
        }
        else
        {
            tmp = data[i-1];
            for (int j = i; j <= last; ++j)
            {
                data[j] = data[j + 1];
            }
        }

        --last;
        return tmp;
```

```
        }

        //获得顺序表的第i个数据元素
        public T GetElem(int i)
        {
            if (IsEmpty() || (i<1) || (i>last+1))
            {
                Console.WriteLine("List is empty or Position is error!");
                return default(T);
            }

            return data[i-1];
        }

        //在顺序表中查找值为value的数据元素
        public int Locate(T value)
        {
            if(IsEmpty())
            {
                Console.WriteLine("List is Empty!");
                return -1;
            }

            int i = 0;
            for (i = 0; i <= last; ++i)
            {
                if (value.Equals(data[i]))
                {
                    break;
                }
            }

            if (i > last)
            {
                return -1;
            }
            return i;
        }
    }
```

2.2.2 顺序表的基本操作实现

1、求顺序表的长度

由于数组是 0 基数组，即数组的最小索引为 0，所以，顺序表的长度就是数组中最后一个元素的索引 last 加 1。

求顺序表长度的算法实现如下：

```
public int GetLength()
{
    return last+1;
}
```

2、清空操作

清除顺序表中的数据元素是使顺序表为空，此时，last 等于-1。

清空顺序表的算法实现如下：

```
public void Clear()
{
    last = -1;
}
```

3、判断线性表是否为空

如果顺序表的 last 为-1，则顺序表为空，返回 true，否则返回 false。

判断线性表是否为空的算法实现如下：

```
public bool IsEmpty()
{
    if (last == -1)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

4、判断顺序表是否为满

如果顺序表为满，last 等于 maxsize-1，则返回 true，否则返回 false。

判断顺序表是否为满的算法实现如下：

```
public bool IsFull()
{
    if (last == maxsize - 1)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

5、附加操作

附加操作是在顺序表未满的情况下，在表的末端添加一个新元素，然后使顺序表的last加1。

附加操作的算法实现如下：

```
public void Append(T item)
```

```
{
    if(IsFull())
    {
        Console.WriteLine("List is full");
        return;
    }

    data[++last] = item;
}
```

6、插入操作

顺序表的插入是指在顺序表的第i个位置插入一个值为item的新元素，插入后使原表长为n的表($a_1, a_2, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n$)成为表长为n+1的表($a_1, a_2, \cdots, a_{i-1}$, item, $a_i, a_{i+1}, \cdots, a_n$)。i的取值范围为1≤i≤n+1，i为n+1时，表示在顺序表的末尾插入数据元素。

顺序表上插入一个数据元素的步骤如下：

（1）判断顺序表是否已满和插入的位置是否正确，表满或插入的位置不正确不能插入；

（2）如果表未满和插入的位置正确，则将a_n～a_i依次向后移动，为新的数据元素空出位置。在算法中用循环来实现；

（3）将新的数据元素插入到空出的第 i 个位置上；

（4）修改 last（相当于修改表长），使它仍指向顺序表的最后一个数据元素。

图 2.2 为顺序表的插入操作示意图。

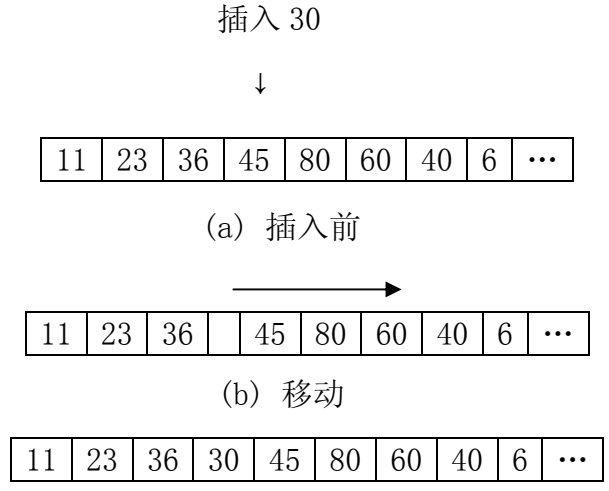


图 2.2 顺序表的插入操作示意图

插入操作的算法实现如下，程序中需要注意的是位置变量 i 的初始值为 1 而不是 0：

```
public void Insert(T item, int i)
{
    //判断顺序表是否已满
    if (IsFull())
    {
```

```
            Console.WriteLine("List is full");
            return;
        }

        //判断插入的位置是否正确，
        // i小于1表示在第1个位置之前插入，
        // i大于last+2表示在最后一个元素后面的第2个位置插入。
        if(i<1 || i>last+2)
        {
            Console.WriteLine(″Position is error!″);
            return;
        }

        //在顺序表的表尾插入数据元素
        if (i == last +2)
        {
            data[i-1] = item;
        }
        else   //在表的其它位置插入数据元素
        {
            //元素移动
            for (int j = last; j>= i-1; --j)
            {
                data[j + 1] = data[j];
            }

            //将新的数据元素插入到第i个位置上
            data[i-1] = item;
        }

        //修改表长
        ++last;
    }
```

算法的时间复杂度分析：顺序表上的插入操作，时间主要消耗在数据的移动上，在第i个位置插入一个元素，从a_i到a_n都要向后移动一个位置，共需要移动n-i+1个元素，而i的取值范围为 1≤i≤n+1，当i等于 1 时，需要移动的元素个数最多，为n个；当i为n+1 时，不需要移动元素。设在第i个位置做插入的概率为p_i，则平均移动数据元素的次数为n/2。这说明：在顺序表上做插入操作平均需要移动表中一半的数据元素，所以，插入操作的时间复杂度为0（n）。

7、删除操作

顺序表的删除操作是指将表中第i个数据元素从顺序表中删除，删除后使原表长为 n 的表（$a_1, a_2, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n$）变为表长为 n-1 的表（$a_1, a_2, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n$）。i的取值范围为 1≤i≤n，i为n时，表示删除顺序表末尾的数据元素。

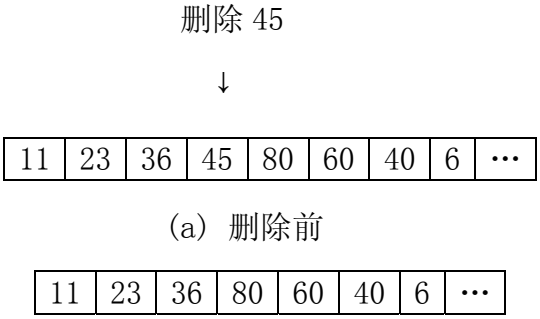
顺序表上删除一个数据元素的步骤如下：

（1）判断顺序表是否为空和删除的位置是否正确，表空或删除的位置不正确不能删除；

（2）如果表未空和删除的位置正确，则将a_{i+1}～a_n依次向前移动。在算法中用循环来实现；

（3）修改 last（相当于修改表长），使它仍指向顺序表的最后一个元素。

图 2.3 为顺序表的删除操作示意图。



(a) 删除前

(b) 删除后

图 2.3 顺序表的删除操作示意图

删除操作的算法实现如下：

```
public T Delete(int i)
{
    T tmp = default(T);

    //判断表是否为空
    if (IsEmpty())
    {
        Console.WriteLine("List is empty");
        return tmp;
    }

    //判断删除的位置是否正确
    // i小于1表示删除第1个位置之前的元素，
    // i大于last+1表示删除最后一个元素后面的第1个位置的元素。
    if (i < 1 || i > last+1)
    {
        Console.WriteLine(″Position is error!″);
        return tmp;
    }

    //删除的是最后一个元素
    if (i == last+1)
    {
        tmp = data[last--];
        return tmp;
    }
```

```
            else   //删除的不是最后一个元素
            {
                //元素移动
                tmp = data[i-1];
                for (int j = i; j <= last; ++j)
                {
                    data[j] = data[j + 1];
                }
            }

            //修改表长
            --last;
            return tmp;
        }
```

算法的时间复杂度分析：顺序表上的删除操作与插入操作一样，时间主要消耗在数据的移动上。在第i个位置删除一个元素，从a_{i+1}到a_n都要向前移动一个位置，共需要移动n-i个元素，而i的取值范围为 1≤i≤n，当i等于 1 时，需要移动的元素个数最多，为n-1 个；当i为n时，不需要移动元素。设在第i个位置做删除的概率为p_i，则平均移动数据元素的次数为(n-1)/2。这说明在顺序表上做删除操作平均需要移动表中一半的数据元素，所以，删除操作的时间复杂度为0（n）。

8、取表元

取表元运算是返回顺序表中第 i 个数据元素，i 的取值范围是 1≤i≤last+1。由于表是随机存取的，所以，如果 i 的取值正确，则取表元运算的时间复杂度为 0（1）。

取表元运算的算法实现如下：

```
public T GetElem(int i)
{
    if (IsEmpty()|| (i<1) || (i>last+1))
    {
        Console.WriteLine("List is empty or Position is error!");
        return default(T);
    }

    return data[i-1];
}
```

9、按值查找

顺序表中的按值查找是指在表中查找满足给定值的数据元素。在顺序表中完成该运算最简单的方法是：从第一个元素起依次与给定值比较，如果找到，则返回在顺序表中首次出现与给定值相等的数据元素的序号，称为查找成功；否则，在顺序表中没有与给定值匹配的数据元素，返回一个特殊值表示查找失败。

按值查找运算的算法实现如下：

```
    public int Locate(T value)
    {
        //顺序表为空
```

```
            if(IsEmpty())
            {
                Console.WriteLine("list is Empty");
                return -1;
            }

            int i = 0;

            //循环处理顺序表
            for (i = 0; i <= last; ++i)
            {
                //顺序表中存在与给定值相等的元素
                if (value.Equals(data[i]))
                {
                    break;
                }
            }

            //顺序表中不存在与给定值相等的元素
            if (i > last)
            {
                return -1;
            }
            return i;
        }
```

算法的时间复杂度分析：顺序表中的按值查找的主要运算是比较，比较的次数与给定值在表中的位置和表长有关。当给定值与第一个数据元素相等时，比较次数为 1；而当给定值与最后一个元素相等时，比较次数为 n。所以，平均比较次数为(n+1)/2，时间复杂度为 0（n）。

由于顺序表是用连续的空间存储数据元素，所以按值查找的方法很多。比如，如果顺序表是有序的，则可以用折半查找法，这样效率可以提高很多。折半查找算法的具体介绍见第 8 章。

2.2.3 顺序表应用举例

【例 2-1】已知顺序表 L，写一算法将其倒置，即实现如图 2.4 所示的操作，其中(a)为倒置前，(b)为倒置后。

11	23	36	45	80	60	40	6

(a)

6	40	60	80	45	36	23	11

(b)

图 2.4 顺序表的倒置

算法思路：把第一个元素与最后一个元素交换，把第二个元素与倒数第二个元素交换。一般地，把第 i 个元素与第 n-i 个元素交换，i 的取值范围是 0 到 n/2

（n 为顺序表的长度）。

存储整数的顺序表的倒置的算法实现如下：

```
public void ReversSeqList(SeqList<int> L)
{
    int tmp = 0;
    int len = L.GetLength();
    for (int i = 0; i<= len/2; ++i)
    {
        tmp = L[i];
        L[i] = L[len - i];
        L[len - i] = tmp;
    }
}
```

该算法只是对顺序表中的数据元素顺序扫描一遍即完成了倒置，所以时间复杂度为 O(n)。

例题说明：这道例题非常简单，把它作为例题的原因是为了和单链表的倒置操作进行比较，来说明顺序存储和链式存储的区别。由于顺序表具有随机存储的性质，所以，对表中任何一个数据元素的定位都是一次性的，并且时间都相同。因此，顺序表的倒置可以通过把相应位置的数据元素交换来完成，这和单链表的差别很大。关于单链表的倒置见例题 2-4。

由于任何线性表都可以进行倒置操作，所以可以把该操作作为 IListDS 接口的一个成员方法进行声明，然后在各线性表类中实现。IListDS 接口中倒置方法 Reverse 的声明如下：

```
public interface IListDS<T> {
        ……
    void Reverse();
}
```

倒置方法在顺序表类 SeqList 中的实现如下：

```
public class SeqList<T> : IListDS<T> {
        ……
    public void Reverse()
    {
        T tmp = Default(T);
        int len = GetLength();
        for (int i = 0; i<= en/2; ++i)
        {
            tmp = data[i];
            data[i] = data[len - i];
            data[len - i] = tmp;
        }
    }
}
```

【例 2-2】有数据类型为整型的顺序表 La 和 Lb，其数据元素均按从小到大的升序排列，编写一个算法将它们合并成一个表 Lc，要求 Lc 中数据元素也按升序排

列。

算法思路：依次扫描 La 和 Lb 的数据元素，比较 La 和 Lb 当前数据元素的值，将较小值的数据元素赋给 Lc，如此直到一个顺序表被扫描完，然后将未完的那个顺序表中余下的数据元素赋给 Lc 即可。Lc 的容量要能够容纳 La 和 Lb 两个表相加的长度。

按升序合并两个表的算法 C#实现如下：

```
public SeqList<int> Merge(Seqlist<int> La, SeqList<int> Lb)
{
    SeqList<int> Lc = new SeqList<int>(La.Maxsize + Lb.Maxsize);
    int i = 0;
    int j = 0;
    int k = 0;

    //两个表中都有数据元素
    while ((i <= (La.GetLength()-1))&& (j <= (Lb.GetLength()-1)))
      {
            if (La[i] <Lb[j])
            {
                Lc.Append(La[i++]);
            }
            else
            {
                Lc.Append(Lb[j++]);
            }
      }

     //a表中还有数据元素
      while (i <= (La.GetLength() - 1))
      {
            Lc.Append(La[i++]);
      }

      //b表中还有数据元素
      while (j <= (Lb.GetLength() - 1))
      {
            Lc.Append(Lb[j++]);
      }

      return Lc;
}
```

算法的时间复杂度是 O(m+n)，m 是 La 的表长，n 是 Lb 的表长。

【例 2-3】已知一个存储整数的顺序表 La，试构造顺序表 Lb，要求顺序表 Lb 中只包含顺序表 La 中所有值不相同的数据元素。

算法思路：先把顺序表 La 的第 1 个元素赋给顺序表 Lb，然后从顺序表 La 的第 2 个元素起，每一个元素与顺序表 Lb 中的每一个元素进行比较，如果不相同，则把该元素附加到顺序表 Lb 的末尾。

从表中删除相同数据元素的算法的 C#实现如下：

```
public SeqList<int> Purge(SeqList<int> La)
{
    SeqList<int> Lb = new SeqList<int>(La.Maxsize);

    //将a表中的第1个数据元素赋给b表
    Lb.Append(La[0]);

    //依次处理a表中的数据元素
    for (int i=1; i<=La.GetLength()-1; ++i)
    {
        int j = 0;

        //查看b表中有无与a表中相同的数据元素
        for (j = 0; j<=Lb.GetLength()-1; ++j)
        {
            //有相同的数据元素
            if (La[i].CompareTo(Lb[j]) == 0)
            {
                break;
            }
         }

         //没有相同的数据元素，将a表中的数据元素附加到b表的末尾。
         if(j>Lb.GetLength()-1)
         {
             Lb.Append(La[i]);
         }
     }

      return Lb;
}
```

算法的时间复杂度是 O(m+n)，m 是 La 的表长，n 是 Lb 的表长。

2.3 单链表

顺序表是用地址连续的存储单元顺序存储线性表中的各个数据元素，逻辑上相邻的数据元素在物理位置上也相邻。因此，在顺序表中查找任何一个位置上的数据元素非常方便，这是顺序存储的优点。但是，在对顺序表进行插入和删除时，需要通过移动数据元素来实现，影响了运行效率。本节介绍线性表的另外一种存储结构——链式存储(Linked Storage)，这样的线性表叫链表(Linked List)。链表不要求逻辑上相邻的数据元素在物理存储位置上也相邻，因此，在对链表进行插入和删除时不需要移动数据元素，但同时也失去了顺序表可随机存储的优点。

2.3.1 单链表的定义

链表是用一组任意的存储单元来存储线性表中的数据元素(这组存储单元可以是连续的，也可以是不连续的)。那么，怎么表示两个数据元素逻辑上的相邻关系呢？即如何表示数据元素之间的线性关系呢？为此，在存储数据元素时，除了存储数据元素本身的信息外，还要存储与它相邻的数据元素的存储地址信息。这两部分信息组成该数据元素的存储映像(Image)，称为结点(Node)。把存储据元素本身信息的域叫结点的数据域(Data Domain)，把存储与它相邻的数据元素的存储地址信息的域叫结点的引用域(Reference Domain)。因此，线性表通过每个结点的引用域形成了一根“链条”，这就是“链表”名称的由来。

如果结点的引用域只存储该结点直接后继结点的存储地址，则该链表叫单链表(Singly Linked List)。把该引用域叫 next。单链表结点的结构如图 2.5 所示，图中 data 表示结点的数据域。

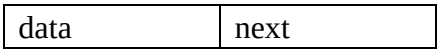


图 2.5 单链表的结点结构

把单链表结点看作是一个类，类名为 Node<T>。单链表结点类的实现如下所示。

```
public class Node<T>
{
  private T data;                  //数据域
  private Node<T> next;        //引用域

  //构造器
  public Node(T val, Node<T> p)
  {
        data = val;
        next = p;
  }

  //构造器
  public Node(Node<T> p)
  {
         next = p;
  }

  //构造器
  public Node(T val)
  {
        data = val;
        next = null;
  }

  //构造器
  public Node()
 {
```

```
            data = default(T);
            next = null;
        }

        //数据域属性
        public T Data
        {
            get
            {
                return data;
            }
            set
            {
                data = value;
            }
        }

        //引用域属性
        public Node<T> Next
        {
            get
            {
                return next;
            }
            set
            {
                next = value;
            }
        }
    }
```

图 2.6 是线性表$(a_1, a_2, a_3, a_4, a_5, a_6)$对应的链式存储结构示意图。

	存储地址	data	next
头引用 H			
867	500	a_4	710
	…	…	…
	560	a_2	930
	…	…	…
	710	a_5	855
	…	…	…
	855	a_6	null
	…	…	…
	867	a_1	560
	…	…	…
	930	a_3	500

图 2.6 链式存储结构示意图

通常，我们把链表画成用箭头相连接的结点的序列，结点间的箭头表示引用域中存储的地址。为了处理上的简洁与方便，在本书中把引用域中存储的地址叫引用。图 2.6 中的链表用图 2. 7 的形式表示。

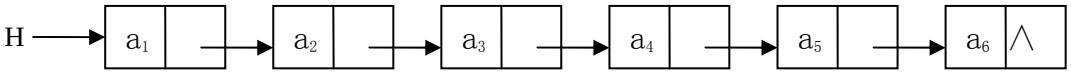


图 2.7 单链表示意图

由图 2.7 可知，单链表由头引用 H 唯一确定。头引用指向单链表的第一个结点，也就是把单链表第一个结点的地址放在 H 中，所以，H 是一个 Node 类型的变量。头引用为 null 表示一个空表。

把单链表看作是一个类，类名叫 LinkList<T>。LinkList<T>类也实现了接口 IListDS<T>。LinkList<T>类有一个字段 head，表示单链表的头引用，所以 head 的类型为 Node<T>。由于链表的存储空间不是连续的，所以没有最大空间的限制，在链表中插入结点时不需要判断链表是否已满。因此，在 LinkList<T>类中不需要实现判断链表是否已满的成员方法。

单链表类 LinkList<T>的实现说明如下所示。

```
public class LinkList<T> : IListDS<T> {
    private Node<T> head;              //单链表的头引用

    //头引用属性
    public Node<T> Head
    {
        get
        {
            return head;
        }

        set
        {
             head = value;
        }
    }

    //构造器
    public LinkList()
    {
        head = null;
    }

    //求单链表的长度
    public int GetLength()
    {
```

```
        Node<T> p = head;

        int len = 0;
        while (p != null)
        {
            ++len;
            p = p.Next;
        }
        return len;
    }

    //清空单链表
    public void Clear()
    {
        head = null;
    }

    //判断单链表是否为空
    public bool IsEmpty()
    {
        if (head == null)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    //在单链表的末尾添加新元素
    public void Append(T item)
    {
        Node<T> q = new Node<T>(item);
        Node<T> p = new Node<T>();

        if (head == null)
        {
              head = q;
              return;
         }

         p = head;
        while (p.Next != null)
```

```
        {
            p = p.Next;
        }

        p.Next = q;
    }

    //在单链表的第i个结点的位置前插入一个值为item的结点
    public void Insert(T item, int i)
    {
        if (IsEmpty() || i < 1)
        {
            Console.WriteLine("List is empty or Position is error!");
            return;
        }

        if (i == 1)
        {
            Node<T> q = new Node<T>(item);
            q.Next = head;
            head = q;
            return;
        }

        Node<T> p = head;
        Node<T> r = new Node<T>();
        int j = 1;

        while (p.Next != null&& j < i)
        {
            r = p;
            p = p.Next;
            ++j;
        }

        if (j == i)
        {
            Node<T> q = new Node<T>(item);
            q.Next = p;
            r.Next = q;
        }
    }

    //在单链表的第i个结点的位置后插入一个值为item的结点
```

```
public void InsertPost(T item, int i)
{
    if (IsEmpty() || i < 1)
    {
        Console.WriteLine("List is empty or Position is error!");
        return;
    }

    if (i == 1)
    {
        Node<T> q = new Node<T>(item);
        q.Next = head.Next;
        head.Next = q;
        return;
    }

    Node<T> p = head;
    int j = 1;

    while (p != null&& j < i)
    {
        p = p.Next;
        ++j;
    }

    if (j == i)
    {
        Node<T> q = new Node<T>(item);
        q.Next = p.Next;
        p.Next = q;
    }
}

//删除单链表的第i个结点
public T Delete(int i)
{
    if (IsEmpty()|| i < 0)
    {
        Console.WriteLine("Link is empty or Position is
                           error!");
        return default(T);
    }

    Node<T> q = new Node<T>();
```

```
        if (i == 1)
        {
            q = head;
            head = head.Next;
            return q.Data;
        }

        Node<T> p = head;
        int j = 1;

        while (p.Next != null&& j < i)
        {
            ++j;
            q = p;
            p = p.Next;
        }

        if (j == i)
        {
            q.Next = p.Next;
            return p.Data;
        }
        else
        {
            Console.WriteLine("The ith node is not exist!");
            return default(T);
        }
    }

    //获得单链表的第i个数据元素
    public T GetElem(int i)
    {
        if (IsEmpty())
        {
            Console.WriteLine("List is empty!");
            return default(T);
        }

        Node<T> p = new Node<T>();
        p = head;
        int j = 1;

        while (p.Next != null&& j < i)
        {
```

```
            ++j;
            p = p.Next;
        }

        if (j == i)
        {
            return p.Data;
        }
        else
        {
            Console.WriteLine("The ith node is not exist!");
            return default(T);
        }
    }

    //在单链表中查找值为value的结点
    public int Locate(T value)
    {
        if(IsEmpty())
        {
            Console.WriteLine("List is Empty!");
            return -1;
        }

        Node<T> p = new Node<T>();
        p = head;
        int i = 1;
        while (!p.Data.Equals(value)&& p.Next != null)
        {
            P = p.Next;
            ++i;
        }

        return i;
    }
}
```

2.3.2 单链表的基本操作实现

1、求单链表的长度

求单链表的长度与顺序表不同。顺序表可以通过指示表中最后一个数据元素的 last 直接求得，因为顺序表所占用的空间是连续的空间，而单链表需要从头引用开始，一个结点一个结点遍历，直到表的末尾。

求单链表长度的算法实现如下：

```
public int GetLength()
{
```

```
        Node<T> p = head;

        int len = 0;
        while (p != null)
        {
            ++len;
            p = p.Next;
        }
        return len;
    }
```

时间复杂度分析：求单链表的长度需要遍历整个链表，所以，时间复杂度为O(n)，n 是单链表的长度。

2、清空操作

清空操作是指清除单链表中的所有结点使单链表为空，此时，头引用 head 为 null。

清空单链表的算法实现如下：

```
public void Clear()
{
    head = null;
}
```

需要注意的是，单链表清空之后，原来结点所占用的空间不会一直保留，而由垃圾回收器进行回收，不用程序员自己处理。这和顺序表的清空操作不一样。顺序表的清空操作需要 last 被置为-1，但为数组分配的空间仍然保留，因为顺序表需要一片连续的空间，而单链表不需要。

3、判断单链表是否为空

如果单链表的头引用为 null，则单链表为空，返回 true，否则返回 false。

判断单链表是否为空的算法实现如下：

```
public bool IsEmpty()
{
    if (head == null)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

4、附加操作

单链表的附加操作也需要从单链表的头引用开始遍历单链表，直到单链表的末尾，然后在单链表的末端添加一个新结点。

附加操作的算法实现如下：

```
public void Append(T item)
{
```

```
        Node<T> q = new Node<T>(item);
        Node<T> p = new Node<T>();

        if (head == null)
        {
            head = q;
            return;
        }

        p = head;
        while (p.Next != null)
        {
            p = p.Next;
        }

        p.Next = q;
    }
```

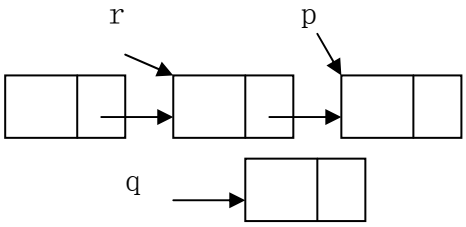
算法的时间复杂度分析：单链表的附加操作需要遍历到最后一个结点，所以，附加操作的时间复杂度为 O(n)，n 是单链表的长度。

5、插入操作

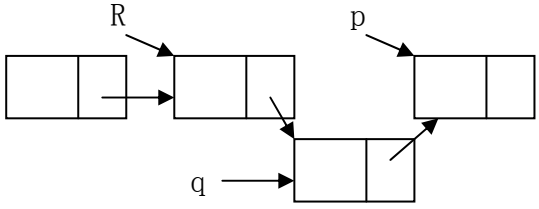
单链表的插入操作是指在表的第 i 个位置结点处插入一个值为 item 的新结点。插入操作需要从单连表的头引用开始遍历，直到找到第 i 个位置的结点。插入操作分为在结点之前插入的前插操作和在结点之后插入的后插操作。

（1）前插操作

前插操作需要查找第 i 个位置的结点的直接前驱。设 p 指向第 i 个位置的结点，q 指向待插入的新结点，r 指向 p 的直接前驱结点，将 q 插在 p 之前的操作如图 2.8 所示。如果要在第一个结点之前插入新结点，则需要把 p 结点的地址保存在 q 的引用域中，然后把 p 的地址保存在头引用中。



(a) 插入前



(b) 插入后

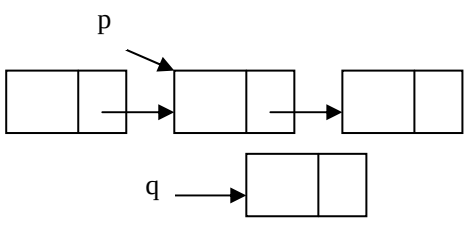
图 2.8 单链表的前插操作示意图

单链表的前插操作的算法实现如下：

```
public void Insert(T item, int i)
{
    if (IsEmpty() || i < 1)
    {
        Console.WriteLine("List is empty or Position is error!");
        return;
    }

    if (i == 1)
    {
        Node<T> q = new Node<T>(item);
        q.Next = head;
        head = q;
        return;
    }

    Node<T> p = head;
    Node<T> r = new Node<T>();
    int j = 1;

    while (p.Next != null&& j < i)
    {
        r = p;
        p = p.Next;
        ++j;
    }

    if (j == i)
    {
        Node<T> q = new Node<T>(item);
        q.Next = p;
        r.Next = q;
    }
    else
    {
        Console.Writeline("Position is error!");
    }
    return;
}
```

（2）后插操作：

设 p 指向第 i 个位置的结点，q 指向待插入的新结点，将 q 插在 p 之后的操作示意图如图 2.9 所示。



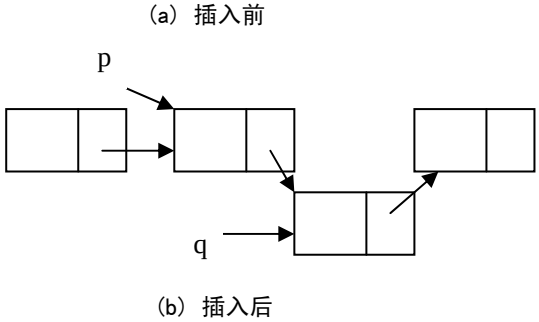


图 2.9 单链表的后插操作示意图

单链表的后插操作的算法实现如下：

```
public void InsertPost(T item, int i)
{
    if (IsEmpty() || i < 1)
    {
        Console.WriteLine("List is empty or Position is error!");
        return;
    }

    if (i == 1)
    {
        Node<T> q = new Node<T>(item);
        q.Next = head.Next;
        head.Next = q;
        return;
    }

    Node<T> p = head;
    int j = 1;

    while (p.Next != null&& j < i)
    {
        p = p.Next;
        ++j;
    }

    if (j == i)
    {
        Node<T> q = new Node<T>(item);
        q.Next = p.Next;
        p.Next = q;
    }
    else
    {
        Console.WriteLine("Position is error!");
```

```
            }
            return;
        }
```

算法的时间复杂度分析：从前插和后插运算的算法可知，在第 i 个结点处插入结点的时间主要消耗在查找操作上。由上面几个操作可知，单链表的查找需要从头引用开始，一个结点一个结点遍历，因为单链表的存储空间不是连续的空间。这是单链表的缺点，而是顺序表的优点。找到目标结点后的插入操作很简单，不需要进行数据元素的移动，因为单链表不需要连续的空间。删除操作也是如此，这是单链表的优点，相反是顺序表的缺点。遍历的结点数最少为 1 个，当 i 等于 1 时，最多为 n，n 为单链表的长度，平均遍历的结点数为 n/2。所以，插入操作的时间复杂度为 O（n）。

因此，线性表的顺序存储和链式存储各有优缺点，线性表如何存储取决于使用的场合。如果不需要经常在线性表中进行插入和删除，只是进行查找，那么，线性表应该顺序存储；如果线性表需要经常插入和删除，而不经常进行查找，则线性表应该链式存储。

6、删除操作

单链表的删除操作是指删除第 i 个结点，返回被删除结点的值。删除操作也需要从头引用开始遍历单链表，直到找到第 i 个位置的结点。如果 i 为 1，则要删除第一个结点，则需要把该结点的直接后继结点的地址赋给头引用。对于其它结点，由于要删除结点，所以在遍历过程中需要保存被遍历到的结点的直接前驱，找到第 i 个结点后，把该结点的直接后继作为该结点的直接前驱的直接后继。删除操作如图 2.10 所示。

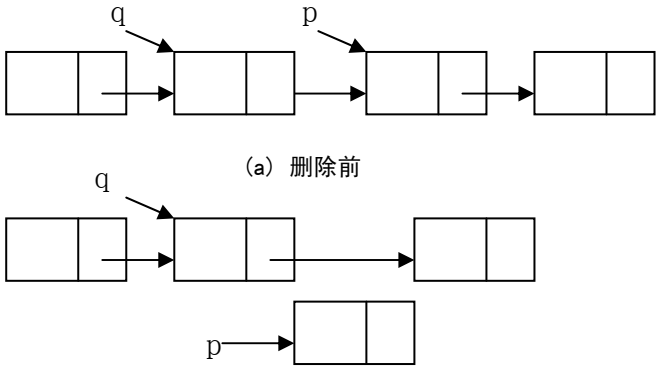


图 2.10 单链表的删除操作示意图

删除操作的算法实现如下：

```
    public T Delete(int i)
    {
        if (IsEmpty()|| i < 0)
        {
            Console.WriteLine("Link is empty or Position is
                                error!");
            return default(T);
        }
```

```
            Node<T> q = new Node<T>();
            if (i == 1)
            {
                q = head;
                head = head.Next;
                return q.Data;
            }

            Node<T> p = head;
            int j = 1;

            while (p.Next != null&& j < i)
            {
                ++j;
                q = p;
                p = p.Next;
            }

            if (j == i)
            {
                q.Next = p.Next;
                return p.Data;
            }
            else
            {
                Console.WriteLine("The ith node is not exist!");
                return default(T);
            }
        }
```

算法的时间复杂度分析：单链表上的删除操作与插入操作一样，时间主要消耗在结点的遍历上。如果表为空则不进行遍历。当表非空时，删除第 i 个位置的结点， i 等于 1 遍历的结点数最少（1 个），i 等于 n 遍历的结点数最多（n 个，n 为单链表的长度），平均遍历的结点数为 n/2。所以，删除操作的时间复杂度为 0（n）。

7、取表元

取表元运算是返回单链表中第 i 个结点的值。与插入操作一样，时间主要消耗在结点的遍历上。如果表为空则不进行遍历。当表非空时，i 等于 1 遍历的结点数最少（1 个），i 等于 n 遍历的结点数最多（n 个，n 为单链表的长度），平均遍历的结点数为 n/2。所以，取表元运算的时间复杂度为 0（n）。

取表元运算的算法实现如下：

```
        public T GetElem(int i)
        {
            if (IsEmpty())
            {
```

```
            Console.WriteLine("List is empty!");
            return default(T);
        }

        Node<T> p = new Node<T>();
        p = head;
        int j = 1;

        while (p.Next != null&& j < i)
        {
            ++j;
            p = p.Next;
        }

        if (j == i)
        {
            return p.Data;
        }
        else
        {
            Console.WriteLine("The ith node is not exist!");
            return default(T);
        }
    }
```

8、按值查找

单链表中的按值查找是指在表中查找其值满足给定值的结点。由于单链表的存储空间是非连续的，所以，单链表的按值查找只能从头引用开始遍历，依次将被遍历到的结点的值与给定值比较，如果相等，则返回在单序表中首次出现与给定值相等的数据元素的序号，称为查找成功；否则，在单链表中没有值与给定值匹配的结点，返回一个特殊值表示查找失败。

按值查找运算的算法如下：

```
    public int Locate(T value)
    {
        if(IsEmpty())
        {
            Console.WriteLine("List is Empty!");
            return -1;
        }

        Node<T> p = new Node<T>();
        p = head;
        int i = 1;
        while (!p.Data.Equals(value)&& p.Next != null)
        {
```

```
            P = p.Next;
            ++i;
        }

        return i;
    }
```

算法的时间复杂度分析：单链表中的按值查找的主要运算是比较，比较的次数与给定值在表中的位置和表长有关。当给定值与第一个结点的值相等时，比较次数为 1；当给定值与最后一个结点的值相等时，比较次数为 n。所以，平均比较次数为(n+1)/2，时间复杂度为 O（n）。

9、单链表的建立

单链表的建立与顺序表的建立不同，它是一种动态管理的存储结构，链表中的每个结点占用的存储空间不是预先分配，而是运行时系统根据需求而生成的。单链表的建立分为在头部插入结点建立单链表和在尾部插入结点建立单链表。由于要读入数据元素，下面以整数为例分别对这两种情况进行讨论。

（1）在单链表的头部插入结点建立单链表。

建立单链表从空表开始，每读入一个数据元素就申请一个结点，然后插在链表的头部。图 2.11 展现了线性表(a_1, a_2, a_3, a_4)的链表建立过程。因为是在链表的头部插入，读入数据的顺序和链表中的逻辑顺序是相反的。

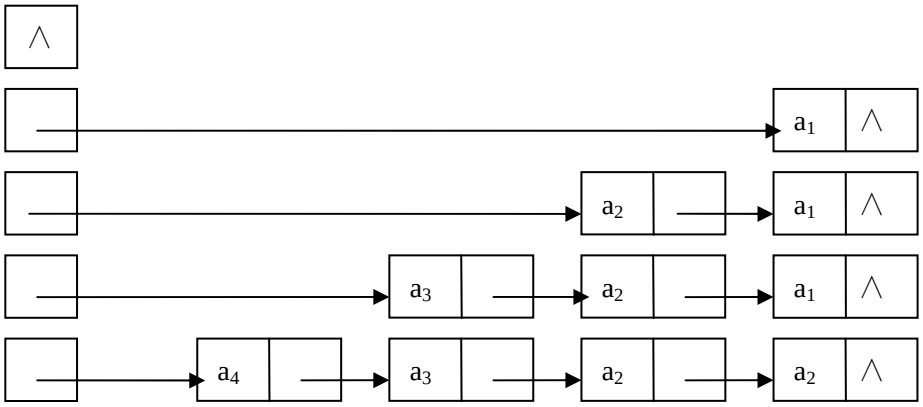


图 2.11 在头部插入结点建立单链表

在头部插入结点建立单链表的算法如下：

```
 LinkList<int> CreateLListHead()
{
        int d;
        LinkList<int> L = new LinkList<int>();

        d = Int32.Parse(Console.ReadLine());

        while (d != -1)
        {
            Node<int> p = new Node<int>(d);
            p.Next = L.Head;
            L.Head = p;
```

```
            d = Int32.Parse(Console.ReadLine());
        }

        return L;
    }
```

-1 是输入数据的结束标志，当输入的数为-1 时，表示输入结束，当然也可以根据需求用其它数作为结束标志。

（2）在单链表的尾部插入结点建立单链表。

头部插入结点建立单链表简单，但读入的数据元素的顺序与生成的链表中元素的顺序是相反的。若希望次序一致，则用尾部插入的方法。因为每次是将新结点插入到链表的尾部，所以需加入一个引用 R 用来始终指向链表中的尾结点，以便能够将新结点插入到链表的尾部。图 2.12 展现了在链表的尾部插入结点建立单链表的过程。

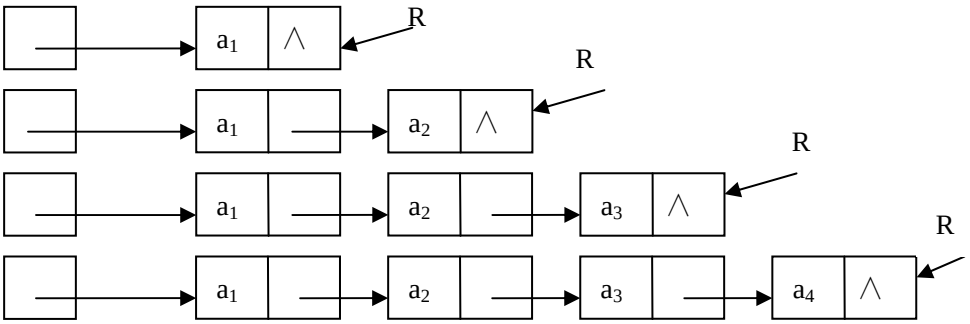


图 2.12 在尾部插入结点建立单链表

算法思路：初始状态时，头引用 head 为 null，尾引用 R 为 null。按线性表中元素的顺序依次读入数据元素，不是结束标志时，申请结点，将新结点插入到 R 所指结点的后面，然后 R 指向新结点。

在尾部插入结点建立单链表的算法如下：

```
 LinkList<int> CreateListTail()
 {
        Node<int> R = new Node<int>();
        int d;
        LinkList<int> L = new LinkList<int>();

        R = L.Head;
        d = Int32.Parse(Console.ReadLine());

        while (d != -1)
        {
            Node<int> p = new Node<int>(d);
            if (L.Head == null)
            {
                L.Head = p;
            }
            else
            {
```

```
                R.Next = p;
            }
            R = p;
            d = Int32.Parse(Console.ReadLine());
        }
        if (R != null)
        {
            R.Next = null;
        }

        return L;
    }
```

-1 是输入数据的结束标志，当输入的数为-1 时，表示输入结束，当然也可以根据需求用其它数作为结束标志。

在上面的算法中，第一个结点的处理和其它结点是不同的，原因是第一个结点加入时链表为空，它没有直接前驱结点，它的地址存放在链表的头引用中；而其它结点有直接前驱结点，其地址放在直接前驱结点的引用域中。在头部插入结点建立单链表的算法中，头引用所指向的结点也是变化的。“第一个结点”的问题在很多操作中都会遇到，如前面讲的在链表中插入结点和删除结点。为了方便处理，其解决办法是让头引用保存的结点地址不变。因此，在链表的头部加入了一个叫头结点(Head Node)的结点，把头结点的地址保存在头引用中。这样，即使是空表，头引用变量也不为空。头结点的加入使得“第一个结点”的问题不再存在，也使得“空表”和“非空表”的处理一致。

头结点的加入完全是为了操作的方便，它的数据域无定义，引用域存放的是第一个结点的地址，空表时该引用域为空。图 2.13 是带头结点的单链表空表和非空表的示意图。

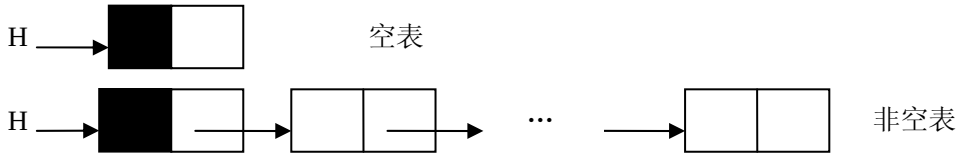


图 2.13 带头结点的单链表

单链表带头结点和不带头结点，操作有所不同，上面讲的操作都是不带头结点的操作。例如：带头结点的单链表的长度是不带头结点的单链表的长度加 1。在需要遍历单链表时，不带头结点的单链表是把头引用 head 赋给一个结点变量，即 p = head，p 为结点变量;而带头结点的单链表是把 head 的引用域赋给一个结点变量，即 p = head.Next，p 为结点变量。

例题中的单链表若没有特别说明，都是指带头结点的单链表。

2.3.3 单链表应用举例

【例 2-4】已知单链表 H，写一算法将其倒置，即实现如图 2.14 所示的操作，其中(a)为倒置前，(b)为倒置后。

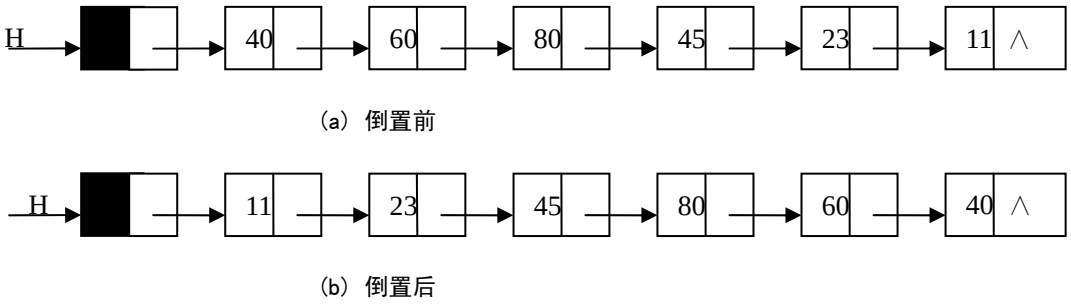


图 2.14 单链表的倒置

算法思路：由于单链表的存储空间不是连续的，所以，它的倒置不能像顺序表那样，把第 i 个结点与第 n-i 个结点交换（i 的取值范围是 1 到 n/2，n 为单链表的长度)。其解决办法是依次取单链表中的每个结点插入到新链表中去。并且，为了节省内存资源，把原链表的头结点作为新链表的头结点。

存储整数的单链表的倒置的算法实现如下：

```
public void ReversLinkList(LinkList<int> H)
{
    Node<int> p = H.Next;
    Node<int> q = new Node<int>();
    H.Next = null;

    while (p != null)
    {
        q = p;
        p = p.Next;
        q.Next = H.Next;
        H.Next = q;
    }
}
```

该算法要对链表中的结点顺序扫描一遍才完成了倒置，所以时间复杂度为 O(n),但比同样长度的顺序表多花一倍的时间，因为顺序表只需要扫描一半的数据元素。

同样，该操作也可以作为 LinkList 类的一个成员方法。倒置方法在单链表类 LinkList 中的实现如下：

```
public class LinkList<T> : IListDS<T> {
        ……
    public void Reverse()
    {
        Node<int> p = head.Next;
        Node<int> q = new Node<int>();
        head.Next = null;
        while (p != null)
        {
            q = p;
```

```
                p = p.Next;
                q.Next = head.Next;
                head.Next = q;
            }
        }
    }
```

【例 2-5】有数据类型为整型的单链表 Ha 和 Hb，其数据元素均按从小到大的升序排列，编写一个算法将它们合并成一个表 Hc，要求 Hc 中结点的值也是升序排列。

算法思路：把 Ha 的头结点作为 Hc 的头结点，依次扫描 Ha 和 Hb 的结点，比较 Ha 和 Hb 当前结点数据域的值，将较小值的结点附加到 Hc 的末尾，如此直到一个单链表被扫描完，然后将未完的那个单链表中余下的结点附加到 Hc 的末尾即可。

将两表合并成一表的算法实现如下：

```
public LinkList<int> Merge(Linklist<int> Ha, LinkList<int> Hb)
{
    LinkList<int> Hc = new LinkList<int>();
    Node<int> p = Ha.Next;
    Node<int> q = Hb.Next;
    Node<int> s = Node<int>();
    Hc = Ha;
    Hc.Next = null;

    while (p != null && q != null)
      {
            if (p.Data < q.Data)
            {
                s = p;
                p = p.Next;
            }
            else
            {
                s = q;
                q = q.Next;
            }

            Hc.Append(s);
      }

      if (p == null)
      {
          p = q;
      }
```

```
        while (p != null)
        {
            s = p;
            p = p.Next;
            Hc.Append(s);
        }

        return Hc;
    }
```

算法的时间复杂度是 O((m+n)*k)，m 是 Ha 的表长，n 是 Hb 的表长，k 是 Hc 的表长。

从上面的算法可知，把结点附加到单链表的末尾是非常花时间的，因为定位最后一个结点需要从头结点开始遍历。而把结点插入到单链表的头部要节省很多时间，因为这不需要遍历链表。但由于是把结点插入到头部，所以得到的单链表是逆序排列而不是升序排列。

把结点插入到链表 Hc 头部合并 Ha 和 Hb 的算法实现如下：

```
public LinkList<int> Merge(Linklist<int> Ha, LinkList<int> Hb)
{
    LinkList<int> Hc = new LinkList<int>();
    Node<int> p = Ha.Next;
    Node<int> q = Hb.Next;
    Node<int> s = Node<int>();
    Hc = Ha;
    Hc.Next = null;

    //两个表非空
    while (p != null && q != null)
    {
        if (p.Data < q.Data)
        {
            s = p;
            p = p.Next;
        }
        else
        {
            s = q;
            q = q.Next;
        }

        s.Next = Hc.Next;
        Hc.Next = s;
    }

    //第2个表非空而第1个表为空
```

```
        if (p == null)
        {
            p = q;
        }

        //将两表中的剩余数据元素附加到新表的末尾
        while (p != null)
        {
            s = p;
            p = p.Next;
            s.Next = Hc.Next;
            Hc.Next = s;
        }

        return Hc;
    }
```

算法的时间复杂度是 O(m+n)，m 是 Ha 的表长，n 是 Hb 的表长。

【例 2-6】已知一个存储整数的单链表 Ha，试构造单链表 Hb，要求单链表 Hb 中只包含单链表 Ha 中所有值不相同的结点。

算法思路：先申请一个结点作为 Hb 的头结点，然后把 Ha 的第 1 个结点插入到 Hb 的头部，然后从 Ha 的第 2 个结点起，每一个结点的数据域的值与 Hb 中的每一个结点的数据域的值进行比较，如果不相同，则把该结点插入到 Hb 的头部。

删除单链表中相同值的结点的算法实现如下：

```
public LinkList<int> Purge(LinkList<int> Ha)
{
    LinkList<int> Hb = new LinkList<int>();
    Node<int> p = Ha.Next;
    Node<int> q = new Node<int>();
    Node<int> s = new Node<int>();

    s = p;
    p = p.Next;
    s.Next = null;
    Hb.Next = s;

    while (p != null)
    {
        s = p;
        p = p.Next;
        q = Hb.Next;

        while (q != null && q.Data != s.Data)
        {
```

```
                q = q.Next;
            }

            if(q == null)
            {
                s.Next = Hb.Next;
                Hb.Next = s;
            }
        }

        return Hb;
    }
```

算法的时间复杂度是 O(m+n)，m 是 Ha 的表长，n 是 Hb 的表长。

2.4 其他链表

2.4.1 双向链表

前面介绍的单链表允许从一个结点直接访问它的后继结点，所以，找直接后继结点的时间复杂度是 O(1)。但是,要找某个结点的直接前驱结点，只能从表的头引用开始遍历各结点。如果某个结点的 Next 等于该结点，那么，这个结点就是该结点的直接前驱结点。也就是说，找直接前驱结点的时间复杂度是 O(n)，n 是单链表的长度。当然，我们也可以在结点的引用域中保存直接前驱结点的地址而不是直接后继结点的地址。这样，找直接前驱结点的时间复杂度只有 O(1)，但找直接后继结点的时间复杂度是 O(n)。如果希望找直接前驱结点和直接后继结点的时间复杂度都是 O(1)，那么，需要在结点中设两个引用域，一个保存直接前驱结点的地址，叫 prev，一个直接后继结点的地址，叫 next，这样的链表就是双向链表(Doubly Linked List)。双向链表的结点结构示意图如图 2.15 所示。

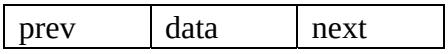


图 2.15 双向链表的结点结构示意图

双向链表结点的定义与单链表的结点的定义很相似,，只是双向链表多了一个字段 prev。双向链表结点类的实现如下所示。

```
public class DbNode<T>
{
    private T data;                    //数据域
    private DbNode<T> prev;            //前驱引用域
    private DbNode<T> next;            //后继引用域

    //构造器
    public DbNode(T val, DbNode<T> p)
    {
        data = val;
        next = p;
    }

    //构造器
```

```
public DbNode(DbNode<T> p)
{
       next = p;
}

//构造器
public DbNode(T val)
{
      data = val;
      next = null;
}

//构造器
public DbNode()
{
      data = default(T);
      next = null;
}

//数据域属性
public T Data
{
      get
      {
            return data;
      }
      set
      {
            data = value;
      }
}

//前驱引用域属性
public DbNode<T> Prev
{
      get
      {
            return prev;
      }
      set
      {
            prev = value;
      }
}
```

```
        //后继引用域属性
        public DbNode<T> Next
        {
            get
            {
                return next;
            }
            set
            {
                next = value;
            }
        }
    }
```

由于双向链表的结点有两个引用，所以，在双向链表中插入和删除结点比单链表要复杂。双向链表中结点的插入分为在结点之前插入和在结点之后插入，插入操作要对四个引用进行操作。下面以在结点之后插入结点为例来说明在双向链表中插入结点的情况。

设 p 是指向双向链表中的某一结点，即 p 存储的是该结点的地址，现要将一个结点 s 插入到结点 p 的后面，插入过程如图 2.16 所示（以 p 的直接后继结点存在为例）。

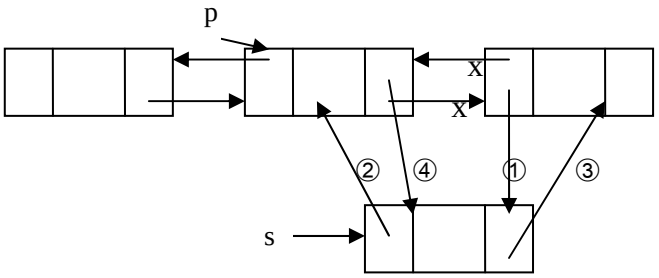


图 2.16 双向链表结点插入示意图

操作如下：

① p.Next.Prev = s;

② s.Prev = p;

③ s.Next = p.Next;

④ p.Next = s;

引用域值的操作的顺序不是唯一的，但也不是任意的，操作③必须放到操作④的前面完成，否则 p 直接后继结点的就找不到了。这一点需要读者把每个操作的含义搞清楚。

双向链表中结点的删除：

以在结点之后删除为例来说明在双向链表中删除结点的情况。设 p 是指向双向链表中的某一结点，即 p 存储的是该结点的地址，现要将一个结点 s 插入到结点 p 的后面，插入过程如图 2.17 所示（以 p 的直接后继结点存在为例）。

操作如下：

① p.Next = P.Next.Next;

② p.Next.Prev = p.Prev;

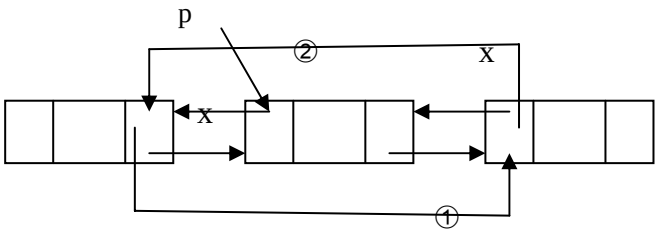
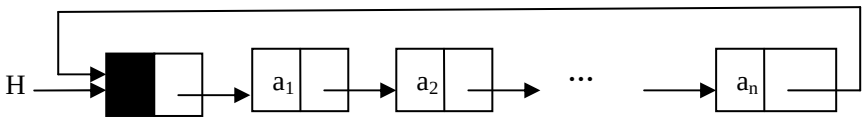


图 2.17 双向链表结点插入示意图

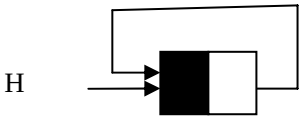
双向链表的其他操作与单链表相似，这里就不一一列举了，读者可以作为习题把双向链表整个类的实现写出来，具体要求见习题二的 2.10 题。

2.4.2 循环链表

有些应用不需要链表中有明显的头尾结点。在这种情况下，可能需要方便地从最后一个结点访问到第一个结点。此时，最后一个结点的引用域不是空引用，而是保存的第一个结点的地址（如果该链表带结点，则保存的是头结点的地址），也就是头引用的值。带头结点的循环链表(Circular Linked List)如图 2.18 所示。



(a) 非空表



(b) 空表

图 2.18 带头结点的单循环链表

循环链表的基本操作与单链表大体相同，只是判断链表结束的条件并不是判断结点的引用域是否为空，而是判断结点的引用域是否为头引用，其它没有较大的变化，所以，这里不再一一详述了，读者可以作为习题把循环链表整个类的实现写出来，具体要求见习题二的 2.11 题。

2.5 C#中的线性表

IList 接口表示一个集合，该集合中的项可被排序且可按索引任意访问。在 C# 1.1 中只提供了非泛型 IList 接口，接口中项的类型是 object。非泛型 IList 接口是从 ICollection 接口继承而来，是所有线性表的基接口。IList 的实现分成三类：只读的，大小不变的和大小可变的。只读的 IList 不能被修改，也就是说，既不能修改表中的项，也不能在表中插入或删除项。大小不变的 Ilist 不能在表中

插入或删除项，但可以修改表中的项。大小可变的 IList 不仅可以修改表中的项，还可以插入或删除项。

非泛型的 IList 接口的声明如下：

```
interface IList : ICollection,IEnumerable
{
    //公有属性
    bool IsFixedSize{get;}          //只读，如果 IList 有固定大小，
                                    //其值为 true,否则为 false。
    bool IsReadOnly{get;}           //只读，如果 IList 是只读的，
                                    //其值为 true,否则为 false。
    object this [T index] {get;set;}  //索引器，得到或设置某个索引的项
                                    //公有方法
    int Add(object value);          //添加某项到表中,返回被插入的新项
                                    //在表中的位置。
    void Clear();                   //清空表。
    int IndexOf(object value);      //如果表中有与 value 相等的项，
                                    //返回其在表中的索引,否则，返回-1。
    bool Contains(object value);    //如果表中有与 value 相等的项，
                                    //返回 true，否则为 false。
    void Insert(int index,object value);    //把值为 value 的项插入到索
                                            //引为 index 的位置。
    void Remove(object value);      //删除表中第一个值为 value 的项。
    void RemoveAt(int index);       //删除表中索引 index 处的项。
}
```

读者可把本书声明的 IListDS 接口与 IList 接口进行比较。

.NET 框架中的一些集合类实现了 IList 接口，如 ArrayList、ListDictionary、StringCollection、StringDictionary。下面以 ArrayList 为例进行说明，其它类的具体情况读者可参看.NET 框架的有关书籍。

ArrayList 类使用数组来实现 IList 接口，所以 ArrayList 可看作顺序表。ArrayList 的容量可动态增长，通常情况下，当 ArrayList 中的元素满时，容量增加一倍，把原来的元素复制到新的空间中。当在 ArrayList 中插入一个元素时，该元素被添加到 ArrayList 的尾部，元素个数自动加 1。另外，需要注意的是，ArrayList 中对元素的操作前提是 ArrayList 是一个有序表，但 ArrayList 本身并不一定是有序的。所以，在对 ArrayList 中的元素进行操作之前，应该对 ArrayList 进行排序。关于排序的算法见第 7 章。

本书不对 ArrayList 类进行详细的介绍，读者可参看.NET 框架的有关书籍，下面以一道例题来说明 ArrayList 的应用。

【例 2-7】ArrayList 的使用。

```
Using System;
using System.Collections;

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

```
    {
        // 创建和初始化一个新的 ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add("Hello");
        myAL.Add("World");
        myAL.Add("!");

        //显示 ArrayList 的属性和值
        Console.WriteLine("myAL");
        Console.WriteLine("Count:{0}", myAL.Count);
        Console.WriteLine("Capacity: {0}", myAL.Capacity);
        Console.Write("Values:");
        PrintValues(myAL);
    }

    //方法，输出 ArrayList 中的每个元素
    public static void PrintValues(Ienumerable myList)
    {
        foreach(object obj in myList)
        {
            Console.Write("{0}", obj);
            Console.WriteLine();
        }
    }
}
```

在 C# 2.0 中不仅提供了非泛型的 Ilist 接口，而且还提供了泛型 Ilist 接口。泛型 Ilist 接口是从 Icollection 泛型接口继承而来，是所有的泛型表的基接口。实现泛型 Ilist 接口的集合提供类似于列表的语法，包括在列表中任意点访问个别项以及插入和删除成员等操作。

泛型的 Ilist 接口的声明如下：

```
public inrterface Ilist<T> : Icollection<T>, Ienumerable<T>,
                             Ienumerable
{
    //公有属性
    T this [int index] {get;set;}          //索引器，得到或设置某个索引的项
                                           //公有方法
    int IndexOf(T value);                  //如果表中有与 value 相等的项，返回
                                           //其在表中的索引,否则，返回-1。
    Void Insert(int index,T value);        //把值为 value 的项插入到索
                                           //引为 index 的位置。
    Void Remove(T value);                  //删除表中第一个值为 value 的项。
}
```

.NET 框架中的一些集合类实现了 Ilist<T>接口，如 List<T>。Ilist<T>相对于 Ilist 的变化是通用的属性和方法被移植入 Icollection<T>了，仅剩下对列表有效的

基于索引访问的属性和方法。

List<T>是 ArrayList 在泛型中的替代品。List<T>的性能比 ArrayList 有很大改变，因为动态数组是.NET 程序使用的最基本的数据结构之一，它的性能影响到应用程序的全局。例如，以前 ArrayList 默认的 Capacity 是 16，而 List<T>的默认 Capacity 是 4，这样可以尽量减小应用程序的工作集。另外，List<T>的方法不是虚拟方法（ArrayList 的方法是虚拟方法），这样可以利用函数内联来提高性能（虚函数不可以被内联）。List<T>也不支持问题很多的 Synchronized 同步访问模式。

本章小结

线性表是最简单、最基本、最常用的数据结构，线性表的特点是数据元素之间存在一对一的线性关系，也就是说，除第一个和最后一个数据元素外，其余数据元素都有且只有一个直接前驱和直接后继。

线性表有两种不同的存储结构，即顺序存储结构和链式存储结构。顺序存储的线性表称为顺序表，顺序表中的存储单元是连续的，在 C#语言中用数组来实现顺序存储。链式存储的线性表称为链表，链表中的存储单元不一定是连续的，所以在一个结点有数据域存放数据元素本身的信息，还有引用域存放其相邻的数据元素的地址信息。单链表的结点只有一个引用域，存放其直接后继结点的地址信息，双向链表的结点有两个引用域，存放其直接前驱结点和直接后继结点的地址信息。循环链表的最后一个结点的引用域存放头引用的值。

对线性表的基本操作有查找、插入、删除等操作。顺序表由于具有随机存储的特点，所以查找比较方便，效率很高，但插入和删除数据元素都需要移动大量的数据元素，所以效率很低。而链表由于其存储空间不要求是连续的，所以插入和删除数据元素的效率很高，但查找需要从头引用开始遍历链表，所以效率很低。因此，线性表采用何种存储结构取决于实际问题，如果只是进行查找等操作而不经常插入和删除线性表中的数据元素，则线性表采用顺序存储结构；反之，采用链式存储结构。

习题二

2.1 说出下面几个概念的含义。

线性表　顺序表　头引用　头结点　单链表　循环链表　双向链表

2.2 在顺序表中进行插入和删除时为什么必须移动数据元素？

2.3 设一顺序表（单链表）中的元素值递增有序。写一算法，将元素 x 插入到表中适当的位置，并保持顺序表（单链表）的有序性。分析算法的时间复杂度。

2.4 已知一整型顺序表 L，编写算法输出表总元素的最大值和最小值。

2.5 编写一算法将整型顺序表 A 中大于 0 的元素放入顺序表 B 中，把小于 0 的元素放入顺序表 C 中。

2.6 对比顺序表和链表，说明二者的主要优点和主要缺点。

2.7 编写算法，逐个输出顺序表中所有的元素。

2.8 在链表设计中，为什么通常采用带头结点的链表结构？

2.9 编写算法，逐个输出单链表中所有结点的值。

2.10 写出双向链表类的实现。

2.11 写出循环链表类的实现。

2.12 写一个复制顺序表的算法。

2.13 写一个复制单链表的算法。

2.14 写一个遍历顺序表的算法。

2.15 写一个遍历单链表的算法。
2.16 将求线性表的长度、判断线性表是否为空及判断线性表是否为满等方法改为属性，重新实现 IListDS<T>、SeqList<T>、LinkList<T>。

第3章 栈和队列

栈和队列是非常重要的两种数据结构，在软件设计中应用很多。栈和队列也是线性结构，线性表、栈和队列这三种数据结构的数据元素以及数据元素间的逻辑关系完全相同，差别是线性表的操作不受限制，而栈和队列的操作受到限制。栈的操作只能在表的一端进行，队列的插入操作在表的一端进行而其它操作在表的另一端进行，所以，把栈和队列称为操作受限的线性表。

3.1 栈

3.1.1 栈的定义及基本运算

栈(Stack)是操作限定在表的尾端进行的线性表。表尾由于要进行插入、删除等操作，所以，它具有特殊的含义，把表尾称为栈顶（Top），另一端是固定的，叫栈底（Bottom）。当栈中没有数据元素时叫空栈(Empty Stack)。

栈通常记为：S= ($a_1, a_2, \cdots, a_n$)，S是英文单词stack的第 1 个字母。a_1为栈底元素，a_n为栈顶元素。这n个数据元素按照$a_1, a_2, \cdots, a_n$的顺序依次入栈，而出栈的次序相反，a_n第一个出栈，a_1最后一个出栈。所以，栈的操作是按照后进先出(Last In First Out，简称LIFO)或先进后出(First In Last Out，简称FILO)的原则进行的，因此，栈又称为LIFO表或FILO表。栈的操作示意图如图 3.1 所示。

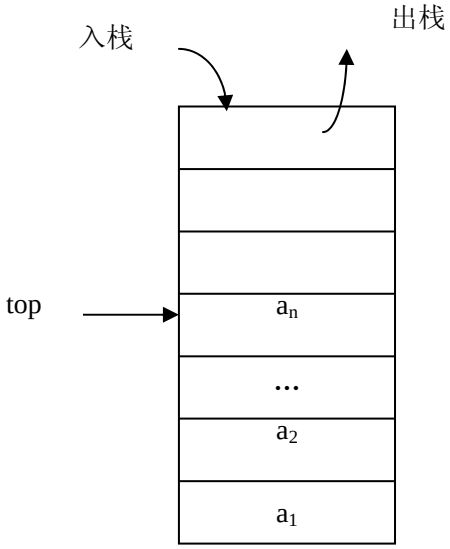


图 3.1 栈的操作示意图

栈的形式化定义为：栈(Stack)简记为 S，是一个二元组，

S = (D, R)

其中：D 是数据元素的有限集合；

R 是数据元素之间关系的有限集合。

在实际生活中有许多类似于栈的例子。比如，刷洗盘子，把洗净的盘子一个接一个地往上放（相当于把元素入栈）；取用盘子的时候，则从最上面一个接一个地往下拿（相当于把元素出栈）。

由于栈只能在栈顶进行操作，所以栈不能在栈的任意一个元素处插入或删除元素。因此，栈的操作是线性表操作的一个子集。栈的操作主要包括在栈顶插入元素和删除元素、取栈顶元素和判断栈是否为空等。

与线性表一样，栈的运算是定义在逻辑结构层次上的，而运算的具体实现是建立在物理存储结构层次上的。因此，把栈的操作作为逻辑结构的一部分，而每个操作的具体实现只有在确定了栈的存储结构之后才能完成。栈的基本运算不是它的全部运算，而是一些常用的基本运算。

同样，我们以 C#语言的泛型接口来表示栈，接口中的方法成员表示基本操作。为表示的方便与简洁，把泛型栈接口取名为 IStack（实际上，在 C#中没有泛型接口 IStack<T>，泛型栈是从 IEnumerable<T>和 ICollection 等接口继承而来，这一点与线性表不一样）。

栈的接口定义如下所示。

```
public interface IStack<T> {
     int GetLength();      //求栈的长度
     bool IsEmpty();       //判断栈是否为空
     void Clear();         //清空操作
     void Push(T item);    //入栈操作
     T Pop();              //出栈操作
     T GetTop();           //取栈顶元素
}
```

下面对栈的基本操作进行说明。

1、求栈的长度：GetLength()

初始条件：栈存在；

操作结果：返回栈中数据元素的个数。

2、判断栈是否为空：IsEmpty()

初始条件：栈存在；

操作结果：如果栈为空返回 true，否则返回 false。

3、清空操作：Clear()

初始条件：栈存在；

操作结果：使栈为空。

4、入栈操作：Push(T item)

初始条件：栈存在；

操作结果：将值为 item 的新的数据元素添加到栈顶，栈发生变化。

5、出栈操作：Pop()

初始条件：栈存在且不为空；

操作结果：将栈顶元素从栈中取出，栈发生变化。

6、取栈顶元素：GetTop()

初始条件：栈表存在且不为空；

操作结果：返回栈顶元素的值，栈不发生变化。

3.1.2 栈的存储和运算实现

1、顺序栈

用一片连续的存储空间来存储栈中的数据元素，这样的栈称为顺序栈(Sequence Stack)。类似于顺序表，用一维数组来存放顺序栈中的数据元素。栈顶指示器 top 设在数组下标为 0 的端，top 随着插入和删除而变化，当栈为空时，top=-1。图 3.3 是顺序栈的栈顶指示器 top 与栈中数据元素的关系图。

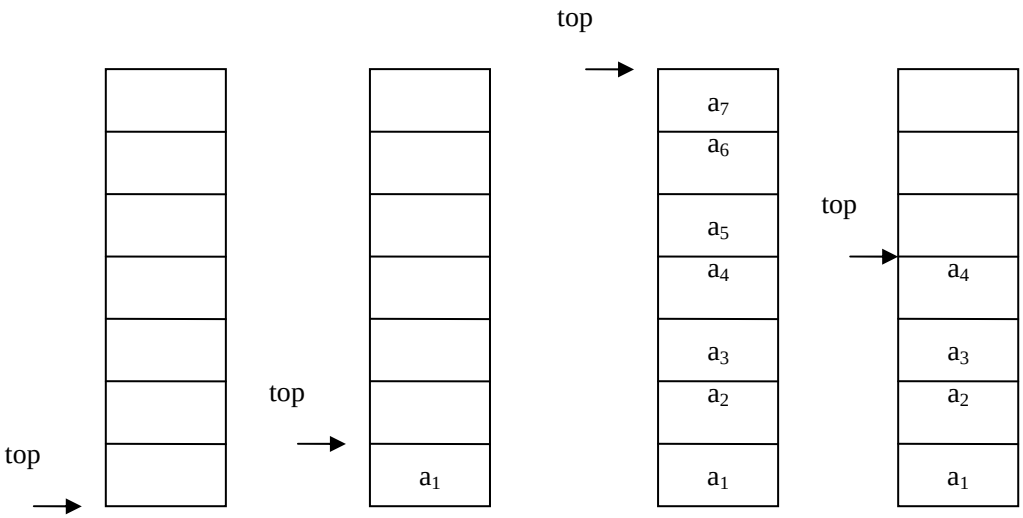


图 3.2 顺序栈的栈顶指示器 top 与栈中数据元素的关系

把顺序栈看作是一个泛型类，类名叫 SeqStack<T>。”Seq”是英文单词”Sequence”的前三个字母。SeqStack<T>类实现了接口 IStack<T>。用数组来存储顺序栈中的元素，在 SeqStack<T>类中用字段 data 来表示。用字段 maxsize 表示栈的容量，与顺序表一样，可以用 System.Array 的 Length 属性来表示，但为了说明顺序栈的容量，在 SeqStackt<T>类中用字段 maxsize 来表示。maxsize 的值可以根据实际需要修改，这通过 SeqStack<T>类的构造器中的参数 size 来实现。顺序栈中的元素由 data[0]开始依次顺序存放。字段 top 表示栈顶，top 的范围是 0 到 maxsize-1，如果顺序栈为空，top=-1。当执行入栈操作时需要判断顺序栈是否已满，顺序栈已满不能插入元素。所以，SeqStack<T>类除了要实现接口 IStack<T>中的方法外，还需要实现判断顺序栈是否已满的成员方法。

顺序栈类 SeqStack<T>的实现说明如下所示。

```
public class SeqStack<T> : IStack<T> {
    private int maxsize;            //顺序栈的容量
    private T[] data;               //数组，用于存储顺序栈中的数据元素
    private int top;                //指示顺序栈的栈顶

    //索引器
    public T this[int index]
    {
        get
        {
            return data[index];
        }
        set
        {
            data[index] = value;
        }
    }
```

```
//容量属性
public int Maxsize
{
    get
    {
        return maxsize;
    }

    set
    {
         maxsize = value;
    }
}

//栈顶属性
public int Top
{
    get
    {
        return top;
    }
}

//构造器
public SeqStack(int size)
{
    data = new T[size];
    maxsize = size;
    top = -1;
}

//求栈的长度
public int GetLength()
{
    return top+1;
}

//清空顺序栈
public void Clear()
{
    top = -1;
}
```

```
//判断顺序栈是否为空
public bool IsEmpty()
{
    if (top == -1)
    {
        return true;
    }
    else
    {
        return false;
    }
}

//判断顺序栈是否为满
public bool IsFull()
{
    if (top == maxsize-1)
    {
        return true;
    }
    else
    {
        return false;
    }
}

//入栈
public void Push(T item)
{
    if(IsFull())
    {
        Console.WriteLine("Stack is full");
        return;
    }

    data[++top] = item;
}

//出栈
public T Pop()
{
    T tmp = default(T);
    if (IsEmpty())
    {
```

```
                Console.WriteLine("Stack is empty");
                return tmp;
            }

            tmp = data[top];
            --top;
            return tmp;
        }

        //获取栈顶数据元素
        public T GetTop()
        {
            if (IsEmpty())
            {
                Console.WriteLine("Stack is empty!");
                return default(T);
            }

            return data[top];
        }
    }
```

顺序栈的基本操作实现有以下 7 种：

（1）求顺序栈的长度

由于数组是 0 基数组，即数组的最小索引为 0，所以，顺序栈的长度就是数组中最后一个元素的索引 top 加 1。

求顺序栈长度的算法实现如下：

```
    public int GetLength()
    {
        return top+1;
    }
```

（2） 清空操作

清除顺序栈中的数据元素是使顺序栈为空，此时，top 等于-1。

清空顺序栈的算法实现如下：

```
    public void Clear()
    {
        top = -1;
    }
```

（3） 判断顺序栈是否为空

如果顺序栈的 top 为-1，则顺序栈为空，返回 true，否则返回 false。

判断顺序栈是否为空的算法实现如下：

```
    public bool IsEmpty()
    {
        if (top == -1)
        {
```

```
            return true;
        }
        else
        {
            return false;
        }
    }
```

（4） 判断顺序栈是否为满

如果顺序栈为满，top 等于 maxsize-1，则返回 true，否则返回 false。

判断顺序栈是否为满的算法实现如下：

```
    public bool IsFull()
    {
        if (top == maxsize - 1)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
```

（5） 入栈操作

入栈操作是在顺序栈未满的情况下，先使栈顶指示器top加1，然后在栈顶添加一个新元素。

入栈操作的算法实现如下：

```
public void Push(T item)
{
    if(IsFull())
    {
        Console.WriteLine("Stack is full");
        return;
    }

    data[++top] = item;
}
```

（6） 出栈操作

顺序栈的出栈操作是指在栈不为空的情况下，使栈顶指示器 top 减 1。

出栈操作的算法实现如下：

```
    public T Pop()
    {
        T tmp = default(T);

        //判断顺序栈是否为空
        if (IsEmpty())
```

```
            {
                Console.WriteLine("Stack is empty");
                return tmp;
            }

            //将栈顶元素赋给一个临时变量
            tmp = data[top];
            --top;
            return tmp;
        }
```

（7）取栈顶元素

如果顺序栈不为空，返回栈顶元素的值，否则返回特殊值表示栈为空。

取栈顶元素运算的算法实现如下：

```
        public T GetTop()
        {
            if (IsEmpty())
            {
                Console.WriteLine("Stack is empty!");
                return default(T);
            }

            return data[top];
        }
```

2、链栈

栈的另外一种存储方式是链式存储，这样的栈称为链栈(Linked Stack)。链栈通常用单链表来表示，它的实现是单链表的简化。所以，链栈结点的结构与单链表结点的结构一样，如图 3.3 所示。由于链栈的操作只是在一端进行，为了操作方便，把栈顶设在链表的头部，并且不需要头结点。

data	next

图 3.3 链栈结点的结构

链栈结点类(Node<T>)的实现如下：

```
        public class Node<T>
        {
            private T data;                 //数据域
            private Node<T> next;       //引用域

            //构造器
            public Node(T val, Node<T> p)
            {
                data = val;
                next = p;
            }

            //构造器
```

```
public Node(Node<T> p)
{
      next = p;
}

//构造器
public Node(T val)
{
    data = val;
    next = null;
}

//构造器
public Node()
{
    data = default(T);
    next = null;
}

//数据域属性
public T Data
{
    get
    {
        return data;
    }
    set
    {
        data = value;
    }
}

//引用域属性
public Node<T> Next
{
    get
    {
        return next;
    }
    set
    {
        next = value;
    }
}
```

}

图 3.4 是链栈示意图。

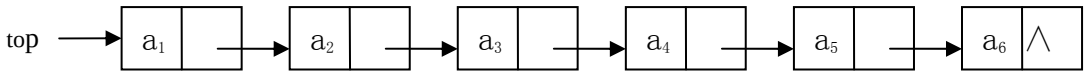


图 3.4 链栈示意图

把链栈看作一个泛型类，类名为 LinkStack<T>。LinkStack<T>类中有一个字段 top 表示栈顶指示器。由于栈只能访问栈顶的数据元素，而链栈的栈顶指示器又不能指示栈的数据元素的个数。所以，求链栈的长度时，必须把栈中的数据元素一个个出栈，每出栈一个数据元素，计数器就增加 1，但这样会破坏栈的结构。为保留栈中的数据元素，需把出栈的数据元素先压入另外一个栈，计算完长度后，再把数据元素压入原来的栈。但这种算法的空间复杂度和时间复杂度都很高，所以，以上两种算法都不是理想的解决方法。理想的解决方法是 LinkStack<T>类增设一个字段 num 表示链栈中结点的个数。

链栈类 LinkStack<T>的实现说明如下所示。

```
public class LinkStack<T> : IStack<T> {
    private Node<T> top;              //栈顶指示器
    private int num;                  //栈中结点的个数

    //栈顶指示器属性
    public Node<T> Top
    {
        get
        {
            return top;
        }
        set
        {
             top = value;
        }
    }

    //元素个数属性
    public int Num
    {
        get
        {
            return num;
        }
        set
        {
             num = value;
        }
    }
```

```
//构造器
public LinkStack()
{
    top = null;
    num = 0;
}

//求链栈的长度
public int GetLength()
{
    return num;
}

//清空链栈
public void Clear()
{
    top = null;
    num = 0;
}

//判断链栈是否为空
public bool IsEmpty()
{
    if ((top == null) && (num == 0))
    {
        return true;
    }
    else
    {
        return false;
    }
}

//入栈
public void Push(T item)
{
    Node<T> q = new Node<T>(item);

    if (top == null)
    {
        top = q;
     }
    else
```

```
            {
                q.Next = top;
                top = q;
            }

            ++num;
        }

        //出栈
        public T Pop()
        {
            if (IsEmpty())
            {
                Console.WriteLine("Stack is empty!");
                return default(T);
            }

            Node<T> p = top;
            top = top.Next;
            --num;

            return p.Data;
        }

        //获取栈顶结点的值
        public T GetTop()
        {
            if (IsEmpty())
            {
                Console.WriteLine("Stack is empty!");
                return default(T);
            }
            return top.Data;
        }
    }
```

链栈的基本操作实现有以下 6 种：

（1）求链栈的长度

num 的大小表示链栈中数据元素的个数，所以通过返回 num 的值来求链栈的长度。

求链栈长度的算法实现如下：

```
        public int GetLength()
        {
            return num;
        }
```

（2） 清空操作

清空操作是清除链栈中的结点，使链栈为空。此时，栈顶指示器 top 等于 null 并且 num 等于 0。

清空链栈的算法实现如下：

```
public void Clear()
{
    top = null;
    num = 0;
}
```

（3） 判断链栈是否为空

如果链栈的栈底指示器为 null 并且 num 等于 0，则链栈为空，返回 true，否则返回 false。

判断链栈是否为空的算法实现如下：

```
public bool IsEmpty()
{
    if ((top == null) && (num == 0))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

（4） 入栈操作

链栈的入栈操作在栈顶添加一个新结点，top指向新的结点，num加1，栈发生变化。

入栈操作的算法实现如下：

```
public void Push()
{
    Node<T> q = new Node<T>(item);

    if (top == null)
    {
        top = q;
    }
    else
    {
        q.Next = top;
        top = q;
    }

    ++num;
}
```

（5） 出栈操作

出栈操作是在栈不为空的情况下，先取出栈顶结点的值，然后将栈顶指示器指向栈顶结点的直接后继结点，使之成为新的栈顶结点，num 减 1，栈发生变化。

出栈操作的算法实现如下：

```
public T Pop()
{
    if (IsEmpty())
    {
        Console.WriteLine("Stack is empty!");
        return default(T);
    }

    Node<T> p = top;
    top = top.Next;
    --num;

    return p.Data;
}
```

（6） 获取链顶结点的值

如果链栈不为空，返回栈顶结点的值，否则返回特殊值表示栈为空，栈不发生变化。

取栈顶元素运算的算法实现如下：

```
public T GetTop()
{
    if (IsEmpty())
    {
        Console.WriteLine("Stack is empty!");
        return default(T);
    }

    return top.Data;
}
```

3.1.3 栈的应用举例

【例 3-1】数制转换问题。数制转换问题是将任意一个非负的十进制数转换为其它进制的数，这是计算机实现计算的基本问题。其一般的解决方法的利用辗转相除法。以将一个十进制数 N 转换为八进制数为例进行说明。假设 N=5142，示例图见图 3.5。

N	N/8(整除)	N%8(求余)	
5142	642	6	低
642	80	2	
80	10	0	
10	1	2	
1	0	1	高

图 3.5 十进制数 N 转换为八进制数示例图

从图 3.5 可知，$(5142)_{10}=(12026)_8$。转换得到的八进制数各个数位是按从低位到高位的顺序产生的，而转换结果的输出通常是按照从高位到低位的顺序依次输出。也就是说，输出的顺序与产生的顺序正好相反，这与栈的操作原则相符。所以，在转换过程中可以使用一个栈，每得到一位八进制数将其入栈，转换完毕之后再依次出栈。

算法思想如下：当 N＞0 时，重复步骤 1 和步骤 2。

步骤 1：若 N≠0，则将 N%8 压入栈中，执行步骤 2；若 N=0。则将栈的内容依次出栈，算法结束。

步骤 2：用 N/8 代替 N，返回步骤 1。

用链栈存储转换得到的数位。

算法实现如下：

```
public void Conversion(int n)
{
    LinkStack<int> s = new LinkStack<int>();
    while(n > 0)
    {
        s.Push(n%8);
        n = n/8;
    }

    while(!s.IsEmpty())
    {
        n = s.Pop();
        Console.WriteLine(“{0}”, n);
     }
}
```

【例 3-2】括号匹配。括号匹配问题也是计算机程序设计中常见的问题。为简化问题，假设表达式中只允许有两种括号：圆括号和方括号。嵌套的顺序是任意的，([]())或[()[()][]]等都为正确的格式，而[(])或([)]等都是不正确的格式。检验括号匹配的方法要用到栈。

算法思想：如果括号序列不为空，重复步骤 1。

步骤 1：从括号序列中取出 1 个括号，分为三种情况：

a) 如果栈为空，则将括号入栈；

b) 如果括号与栈顶的括号匹配，则将栈顶括号出栈。

c) 如果括号与栈顶的括号不匹配，则将括号入栈。

步骤 2：如果括号序列为空并且栈为空则括号匹配，否则不匹配。

算法如下，用顺序栈实现算法：

```
public bool MatchBracket(char[] charlist)
{
    SeqStack<char> s = new SeqStack<char>(50);
    int len = charlist.Length;
    for (int i = 0; i < len; ++i)
    {
        if (s.IsEmpty())
        {
            s.Push(charlist[i]);
        }
        else if(((s.GetTop()=='(') && (charlist[i]==')'))
                ||(s.GetTop()=='[' && charlist[i]==']'))
        {
                s.Pop();
        }
        else
        {
            s.Push(charlist[i]);
        }
    }

    if (s.IsEmpty())
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

【例 3-3】表达式求值。表达式求值是程序设计语言编译中的一个基本问题，它的实现是栈应用的一个典型例子。这里介绍“算符优先算法”，这种算法简单直观且使用广泛。

“算符优先算法”是用运算符的优先级来确定表达式的运算顺序，从而对表达式进行求值。在机器内部，任何一个表达式都是由操作数(Operand)、运算符(Operator)和界限符(Delimiter)组成。操作数和运算符是表达式的主要部分，分界符标志了一个表达式的结束。根据表达式的类型，表达式分为三类，即算术表达式、关系表达式和逻辑表达式。为简化问题，我们仅讨论四则算术运算表达式，并且假设一个算术表达式中只包含加、减、乘、除、左圆括号和右圆括号等符号，并假设‘#’是界限符。

要把一个表达式翻译成正确求值的一个机器指令序列，或者直接对表达式求

值，首先要能够正确解释表达式，这需要了解算术四则运算的规则。算术四则运算的规则如下：

（1） 先乘除后加减；

（2） 先括号内后括号外；

（3） 同级别时先左后右。

我们把运算符和界限符统称为算符。根据上述三条运算规则，在任意相继出现的算符 θ_1和 θ_2之间至多是下面三种关系之一：

（1） $\theta_1 < \theta_2$ θ_1的优选权低于 θ_2；

（2） $\theta_1 = \theta_2$ θ_1的优选权等于 θ_2；

（3） $\theta_1 > \theta_2$ θ_1的优选权高于 θ_2。

表 3-1 定义了算符之间的这种优先关系，为了算法简洁，在表达式的最左边也虚设一个‘#’构成整个表达式的一对括号。

表 3-1 算符之间的优先级关系

θ_1 \ θ_2	+	−	*	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

由表 3-1 可知：

（1）‘#’的优先级最低，当‘#’=‘#’表示整个表达式结束；

（2）同级别的算符遇到时，左边算符的优先级高于右边算符的优先级，如‘+’与‘+’、‘-’与‘-’、‘+’与‘-’等；

（3）‘(’在左边出现时，其优先级低于右边出现的算符，如‘+’、‘-’、‘*’等，‘(’=‘)’表示括号内运算结束；‘(’在右边出现时，其优先级高于左边出现的算符，如‘+’、‘-’、‘*’等；

（4）‘)’在左边出现时，其优先级高于右边出现的算符，如‘+’、‘-’、‘*’等；‘)’在右边出现时，其优先级低于左边出现的算符，如‘+’、‘-’、‘*’等；

（5）‘)’与‘(’、‘#’与‘)’、‘(’与‘#’之间无优先关系，在表达式中不允许相继出现，如果出现认为是语法错误。

为实现算法，使用两个栈，一个存放算符，叫 OPTR，一个存放操作数和运算的结果数，叫 OPND。算法思想如下：

（1） 首先置 OPND 为空，将‘#’入 OPTR；

（2） 依次读入表达式中的每个字符，若是操作数则将该字符入 OPND，若是算符，则和 OPTR 栈顶字符比较优先级，若 OPTR 栈顶字符优先级高，则

将 OPND 栈中的两个操作数和 OPTR 栈顶算符出栈，然后将操作结果入 OPND；若 OPTR 栈顶字符优先级低，则将该字符入 OPTR；若二者优先级相等，则将 OPTR 栈顶字符出栈并读入下一个字符。

表达式求值的算法实现如下。本例的算法是处理整数的，也可以对实数等其它数进行处理，只不过把类型改为实数等相应类型即可。

```
public int EvaluateExpression()
{
    SeqStack<char> optr = new SeqStack <char>(20);
    SeqStack<int> opnd = new SeqStack <int>(20);
    optr.Push('#');
    char c = Console.Read();
    char theta = 0;
    int a = 0;
    int b = 0;

    while (c != '#')
    {
        if((c!='+') && (c!='-')
            && (c!='*') && (c!='/')
            && (c!='(') && (c!=')'))
        {
            optr.Push(c);
        }
        else
        {
            switch(Precede(optr.GetTop(), c))
            {
                Case '<':
                    optr.Push(c);
                    c = Console.Read();
                    break;
                case '=':
                      optr.Pop();
                      c = Console.Read();
                      break;
                 case '>':
                      theta = optr.Pop();
                      a = opnd.Pop();
                      b = opnd.Pop();
                      opnd.Push(Operate(a,theta,b));
                      break;
            }
        }
```

```
        }

        return opnd.GetTop();
    }
```

算法中调用了两个方法。其中，Precede 是判定 optr 栈顶算符与读入的算符之间的优先级关系的方法；Operate 为进行二元运算的方法。这两个方法可作为作业让学生进行练习，见习题三中习题 3.4。

3.1.4 C#中的栈

C#2.0 以下版本只提供了非泛型的 Stack 类，该类继承了 ICollection、IEnumerable 和 ICloneable 接口。C#2.0 提供了泛型的 Stack<T>类，该类继承了 IEnumerable<T>、ICollection 和 IEnumerable 接口。下面的程序说明了泛型 Stack<T>类的主要方法，并对在我们自定义的栈类中没有出现的成员方法进行了注释，关于泛型 Stack<T>类的更具体的信息，读者可参考.NET Framework 的有关书籍。

```
public class Stack<T> : IEnumerable<T>, ICollection, IEnumerable
{
    public Stack();
    public Stack(int capacity);
    public int Count {get;}
    public void Clear();

    //确定某元素是否在Stack<T>中,
    //如果在Stack<T>中找到item，则为true；否则为false。
    public bool Contains(T item);

    //从指定数组索引开始将Stack<T>复制到现有一维Array中。
    public void CopyTo(T[] array, int arrayIndex);

    //返回位于Stack<T>顶部的对象但不将其移除。
    public T Peek();

    public T Pop();
    public void Push(T item);

    //将Stack<T>复制到新数组中。
    public T[] ToArray();

    //如果元素数小于当前容量的90%，
    //将容量设置为Stack<T>中的实际元素数。
    public void TrimExcess();
}
```

3.2 队列

3.2.1 队列的定义及基本运算

队列(Queue)是插入操作限定在表的尾部而其它操作限定在表的头部进行的

线性表。把进行插入操作的表尾称为队尾(Rear)，把进行其它操作的头部称为队头(Front)。当对列中没有数据元素时称为空对列(Empty Queue)。

队列通常记为：Q= ($a_1, a_2, \cdots, a_n$)，Q是英文单词queue的第 1 个字母。a_1为队头元素，a_n为队尾元素。这n个元素是按照$a_1, a_2, \cdots, a_n$的次序依次入队的，出对的次序与入队相同，a_1第一个出队，a_n最后一个出队。所以，对列的操作是按照先进先出(First In First Out)或后进后出(Last In Last Out)的原则进行的，因此，队列又称为FIFO表或LILO表。队列Q的操作示意图如图 3.6 所示。

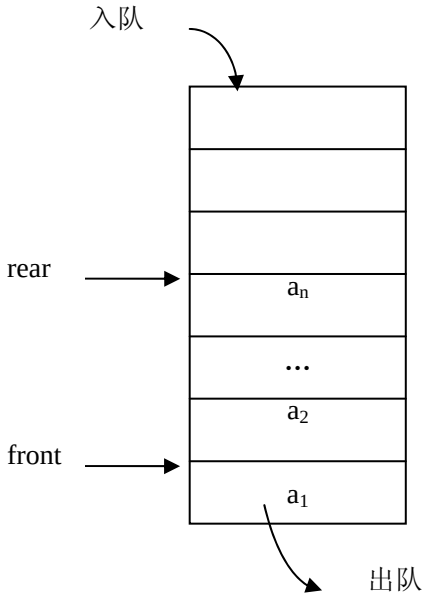


图 3.6　队列的操作示意图

队列的形式化定义为：队列(Queue)简记为 Q，是一个二元组，

Q = (D, R)

其中：D 是数据元素的有限集合；

R 是数据元素之间关系的有限集合。

在实际生活中有许多类似于队列的例子。比如，排队取钱，先来的先取，后来的排在队尾。

队列的操作是线性表操作的一个子集。队列的操作主要包括在队尾插入元素、在队头删除元素、取队头元素和判断队列是否为空等。

与栈一样，队列的运算是定义在逻辑结构层次上的，而运算的具体实现是建立在物理存储结构层次上的。因此，把队列的操作作为逻辑结构的一部分，每个操作的具体实现只有在确定了队列的存储结构之后才能完成。队列的基本运算不是它的全部运算，而是一些常用的基本运算。

同样，我们以 C#语言的泛型接口来表示队列，接口中的方法成员表示基本操作。为了表示的方便与简洁，把泛型队列接口取名为 IQueue<T>（实际上，在C#中泛型队列类是从 IEnumerable<T>、ICollection 和 IEnumerable 接口继承而来，没有 IQueue<T>泛型接口）。

队列接口 IQueue<T>的定义如下所示。

```
public interface IQueue<T> {
    int GetLength();          //求队列的长度
    bool IsEmpty();           //判断对列是否为空
    void Clear();             //清空队列
```

```
    void In(T item);          //入队
    T Out();                  //出队
    T GetFront();             //取对头元素
}
```

下面对队列的基本操作进行说明。

1、求队列的长度：GetLength()

初始条件：队列存在；

操作结果：返回队列中数据元素的个数。

2、判断队列是否为空：IsEmpty()

初始条件：队列存在；

操作结果：如果队列为空返回 true，否则返回 false。

3、清空操作：Clear()

初始条件：队列存在；

操作结果：使队列为空。

4、入队列操作：In(T item)

初始条件：队列存在；

操作结果：将值为 item 的新数据元素添加到队尾，队列发生变化。

5、出队列操作：Out()

初始条件：队列存在且不为空；

操作结果：将队头元素从队列中取出，队列发生变化。

6、取队头元素：GetFront()

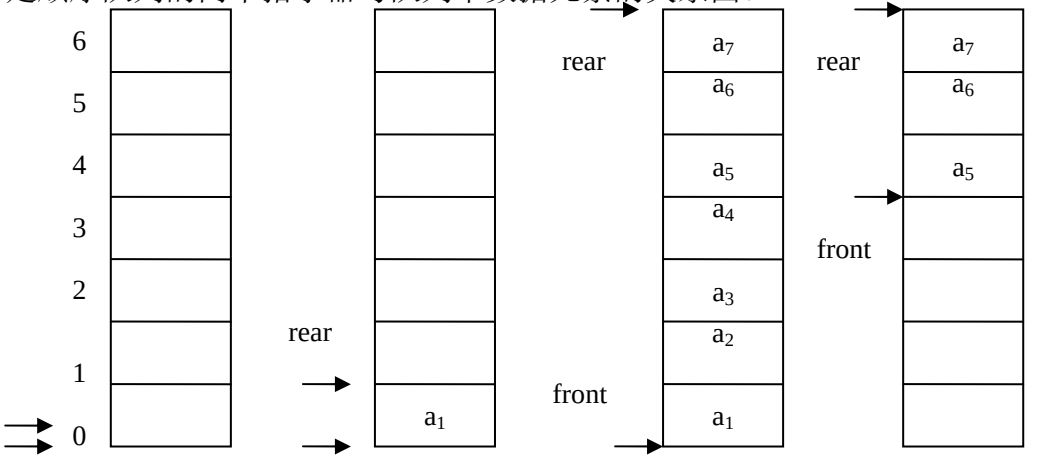
初始条件：队列存在且不为空；

操作结果：返回队头元素的值，队列不发生变化。

3.2.2 队列的存储和运算实现

1、顺序队列

用一片连续的存储空间来存储队列中的数据元素，这样的队列称为顺序队列(Sequence Queue)。类似于顺序栈，用一维数组来存放顺序队列中的数据元素。队头位置设在数组下标为 0 的端，用 front 表示；队尾位置设在数组的另一端，用 rear 表示。front 和 rear 随着插入和删除而变化。当队列为空时，front=rear=-1。图 3.7 是顺序队列的两个指示器与队列中数据元素的关系图。



(a) 空队　(b) 1 个元素　(c) 一般情况　(d) 假溢出

图 3.7 顺序队列的两个指示器与队列中数据元素的关系图

当有数据元素入队时，队尾指示器 rear 加 1，当有数据元素出队时，队头指示器 front 加 1。当 front=rear 时，表示队列为空，队尾指示器 rear 到达数组的上限处而 front 为-1 时，队列为满，如图图 3.7(c)所示。队尾指示器 rear 的值大于队头指示器 front 的值，队列中元素的个数可以由 rear-front 求得。

由图 3.7(d)可知，如果再有一个数据元素入队就会出现溢出。但事实上队列中并未满，还有空闲空间，把这种现象称为“假溢出”。这是由于队列“队尾入队头出”的操作原则造成的。解决假溢出的方法是将顺序队列看成是首尾相接的循环结构，头尾指示器的关系不变，这种队列叫循环顺序队列(Circular sequence Queue)。循环队列如图 3.8 所示。

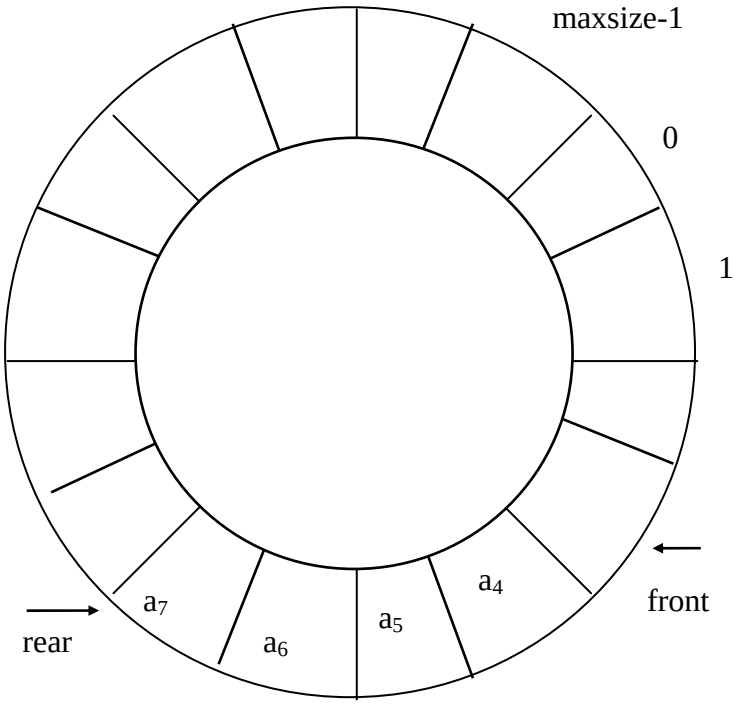


图 3.8 循环顺序队列示意图

当队尾指示器 rear 到达数组的上限时，如果还有数据元素入队并且数组的第 0 个空间空闲时，队尾指示器 rear 指向数组的 0 端。所以，队尾指示器的加 1 操作修改为：

rear = (rear + 1) % maxsize

队头指示器的操作也是如此。当队头指示器 front 到达数组的上限时，如果还有数据元素出队，队头指示器 front 指向数组的 0 端。所以，队头指示器的加 1 操作修改为：

front = (front + 1) % maxsize

循环顺序队列操作示意图如图 3.9 所示。

由图 3.9 可知，队尾指示器 rear 的值不一定大于队头指示器 front 的值，并且队满和队空时都有 rear=front。也就是说，队满和队空的条件都是相同的。解

决这个问题的方法一般是少用一个空间，如图 3.9(d)所示，把这种情况视为队满。所以，判断队空的条件是：rear==front，判断队满的条件是：(rear + 1) % maxsize==front。求循环队列中数据元素的个数可由(rear-front+maxsize)%maxsize 公式求得。

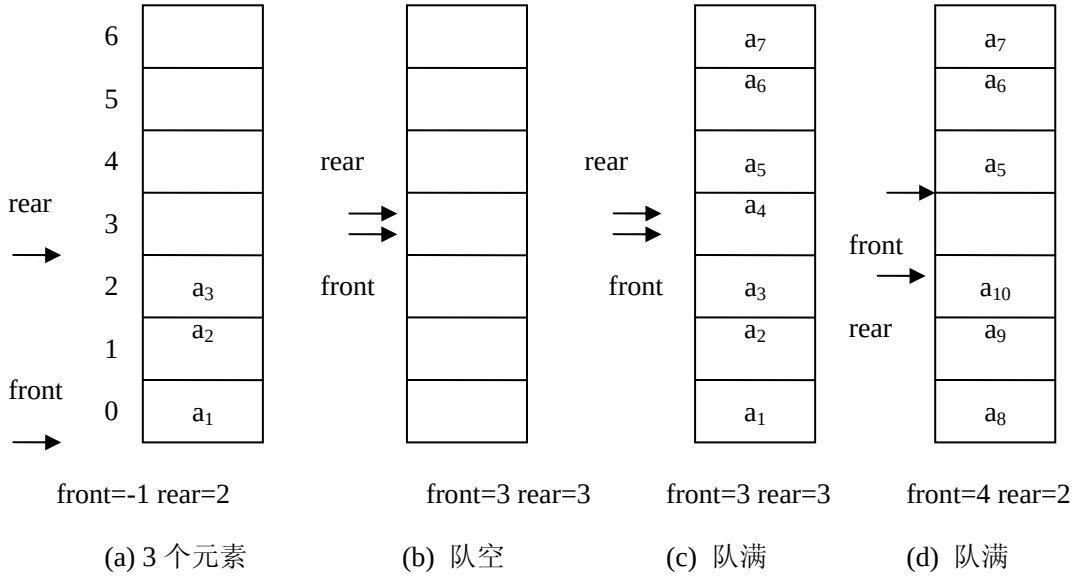


图 3.9 循环顺序队列操作示意图

把循环顺序队列看作是一个泛型类，类名叫 CSeqStack<T>，“C”是英文单词 circular 的第 1 个字母。CSeqStack<T>类实现了接口 IQueue<T>。用数组来存储循环顺序队列中的元素，在 CSeqStack<T>类中用字段 data 来表示。用字段 maxsize 表示循环顺序队列的容量，maxsize 的值可以根据实际需要修改，这通过 CSeqStack<T>类的构造器中的参数 size 来实现，循环顺序队列中的元素由 data[0]开始依次顺序存放。字段 front 表示队头，front 的范围是 0 到 maxsize-1。字段 rear 表示队尾，rear 的范围也是 0 到 maxsize-1。如果循环顺序队列为空，front=rear=-1。当执行入队列操作时需要判断循环顺序队列是否已满，如果循环顺序队列已满，(rear + 1) % maxsize==front，循环顺序队列已满不能插入元素。所以，CSeqStack<T>类除了要实现接口 IQueue<T>中的方法外，还需要实现判断循环顺序队列是否已满的成员方法。

循环顺序队列类 CSeqQueue<T>的实现说明如下所示。

```
public class CSeqQueue<T> : IQueue<T> {
    private int maxsize;          //循环顺序队列的容量
    private T[] data;             //数组，用于存储循环顺序队列中的数据元素
    private int front;            //指示循环顺序队列的队头
    private int rear;             //指示循环顺序队列的队尾

    //索引器
    public T this[int index]
    {
        get
        {
            return data[index];
```

```
        }
        set
        {
            data[index] = value;
        }
    }

    //容量属性
    public int Maxsize
    {
        get
        {
            return maxsize;
        }
        set
        {
            maxsize = value;
        }
    }

    //队头属性
    public int Front
    {
        get
        {
            return front;
        }
        set
        {
            front = value;
        }
    }

    //队尾属性
    public int Rear
    {
        get
        {
            return rear;
        }
        set
        {
            rear = value;
        }
```

```
}

//构造器
public CSeqQueue(int size)
{
    data = new T[size];
    maxsize = size;
    front = rear = -1;
}

//求循环顺序队列的长度
public int GetLength()
{
    return (rear-front+maxsize) % maxsize;
}

//清空循环顺序队列
public void Clear()
{
    front = rear = -1;
}

//判断循环顺序队列是否为空
public bool IsEmpty()
{
    if (front == rear)
    {
        return true;
    }
    else
    {
        return false;
    }
}

//判断循环顺序队列是否为满
public bool IsFull()
{
    if ((rear + 1) % maxsize==front)
    {
        return true;
    }
    else
    {
```

```
                return false;
            }
        }

        //入队
        public void In(T item)
        {
            if(IsFull())
            {
                Console.WriteLine("Queue is full");
                return;
            }

            data[++rear] = item;
        }

        //出队
        public T Out()
        {
            T tmp = default(T);
            if (IsEmpty())
            {
                Console.WriteLine("Queue is empty");
                return tmp;
            }

            tmp = data[++front];
            return tmp;
        }

        //获取队头数据元素
        public T GetFront()
        {
            if (IsEmpty())
            {
                Console.WriteLine("Queue is empty!");
                return default(T);
            }

            return data[front+1];
        }
    }
```

循环顺序队列的基本操作实现有以下 7 种：

（1）求循环顺序队列的长度

循环顺序队列的长度取决于队尾指示器 rear 和队头指示器 front。一般情况下，rear 大于 front，因为入队的元素肯定比出队的元素多。特殊的情况是 rear 到达数组的上限之后又从数组的低端开始，此时，rear 是小于 front 的。所以，rear 的大小要加上 maxsize。因此，循环顺序队列的长度应该是：(rear-front+maxsize)%maxsize。

求循环顺序队列长度的算法实现如下：

```
public int GetLength()
{
    return (rear-front+maxsize) % maxsize;
}
```

（2）清空操作

清除循环顺序队列中的数据元素是使循环顺序队列为空，此时，rear 和 front 均等于-1。

清空循环顺序队列的算法实现如下：

```
public void Clear()
{
    front = rear = -1;
}
```

（3）判断循环顺序队列是否为空

如果循环顺序队列的 rear 等于 front，则循环顺序队列为空，返回 true，否则返回 false。

判断循环顺序队列是否为空的算法实现如下：

```
public bool IsEmpty()
{
    if (front == rear)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

（4）判断循环顺序队列是否为满

如果循环顺序队列为满，(rear + 1) % maxsize=front，则返回 true，否则返回 false。

判断循环顺序队列是否为满的算法实现如下：

```
public bool IsFull()
{
    if ((rear + 1) % maxsize==front)
    {
        return true;
    }
    else
```

```
            {
                return false;
            }
        }
```

（5） 入队操作

入队操作是在循环顺序队列未满的情况下，先使循环顺序队列的rear加1，然后在rear指示的位置添加一个新元素。

入队操作的算法实现如下：

```
    public void Push(T item)
    {
        if(IsFull())
        {
            Console.WriteLine("Queue is full");
            return;
        }

        data[++rear] = item;
    }
```

（6）出队操作

循环顺序队列的出队操作是指在队列不为空的情况下，使队头指示器front加1。

出队操作的算法实现如下：

```
        public T Out()
        {
            T tmp = default(T);

            //判断循环顺序队列是否为空
            if (IsEmpty())
            {
                Console.WriteLine("Queue is empty");
                return tmp;
            }

            tmp = data[++front];
            return tmp;
        }
```

（7）获取队头元素

如果循环顺序队列不为空，返回队头元素的值，否则返回特殊值表示队列为空。

获取队头元素运算的算法实现如下：

```
        public T GetFront()
        {
            if (IsEmpty())
            {
```

```
            Console.WriteLine("Queue is empty!");
            return default(T);
        }

        return data[front+1];
    }
```

2、链队列

队列的另外一种存储方式是链式存储，这样的队列称为链队列(Linked Queue)。同链栈一样，链队列通常用单链表来表示，它的实现是单链表的简化。所以，链队列的结点的结构与单链表一样，如图 3.10 所示。由于链队列的操作只是在一端进行，为了操作方便，把队头设在链表的头部，并且不需要头结点。

data	next

图 3.10 链队列结点的结构

链队列结点类(Node<T>)的实现如下所示：

```
public class Node<T>
{
    private T data;                 //数据域
    private Node<T> next;       //引用域

    //构造器
    public Node(T val, Node<T> p)
    {
        data = val;
        next = p;
    }

    //构造器
    public Node(Node<T> p)
    {
        next = p;
    }

    //构造器
    public Node(T val)
    {
        data = val;
        next = null;
    }

    //构造器
    public Node()
    {
        data = default(T);
        next = null;
```

```
        }

        //数据域属性
        public T Data
        {
            get
            {
                return data;
            }
            set
            {
                data = value;
            }
        }

        //引用域属性
        public Node<T> Next
        {
            get
            {
                return next;
            }
            set
            {
                next = value;
            }
        }
    }
```

图 3.11 是链队列示意图。

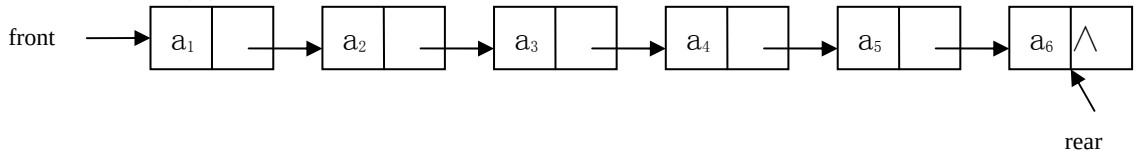


图 3.11 链队列示意图

把链队列看作一个泛型类，类名为 LinkQueue<T>。LinkQueue<T>类中有两个字段 front 和 rear，表示队头指示器和队尾指示器。由于队列只能访问队头的数据元素，而链队列的队头指示器和队尾指示器又不能指示队列的元素个数，所以，与链栈一样，在 LinkQueue<T>类增设一个字段 num 表示链队列中结点的个数。

链队列类 LinkQueue<T>的实现说明如下所示。

```
public class LinkQueue<T> : IQueue<T> {
    private Node<T> front;          //队列头指示器
    private Node<T> rear;           //队列尾指示器
    private int num;                //队列结点个数
```

```
//队头属性
public Node<T> Front
{
    get
    {
        return front;
    }
    set
    {
         front = value;
    }
}

//队尾属性
public Node<T> Rear
{
    get
    {
        return rear;
    }
    set
    {
         rear = value;
    }
}

//队列结点个数属性
public int Num
{
    get
    {
        return num;
    }
    set
    {
         num = value;
    }
}

//构造器
public LinkQueue()
{
    front = rear = null;
```

```
        num = 0;
    }

    //求链队列的长度
    public int GetLength()
    {
        return num;
    }

    //清空链队列
    public void Clear()
    {
        front =rear = null;
        num = 0;
    }

    //判断链队列是否为空
    public bool IsEmpty()
    {
        if ((front == rear)&& (num == 0))
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    //入队
    public void In(T item)
    {
        Node<T> q = new Node<T>(item);

        if (rear == null)
        {
              rear = q;
         }
        else
        {
            rear.Next = q;
            rear = q;
        }
```

```
            ++num;
        }

        //出队
        public T Out()
        {
            if (IsEmpty())
            {
                Console.WriteLine("Queue is empty!");
                return default(T);
            }

            Node<T> p = front;
            front = front.Next;

            if(front == null)
            {
                rear = null;
            }
            --num;

            return p.Data;
        }

        //获取链队列头结点的值
        public T GetFront()
        {
            if (IsEmpty())
            {
                Console.WriteLine("Queue is empty!");
                return default(T);
            }

            return front.Data;
        }
    }
```

链队列的基本操作实现有以下 6 种：

（1） 求链队列的长度

num 的大小表示链队列中数据元素的个数，所以通过返回 num 的值来求链队列的长度。

求链队列长度的算法实现如下：

```
        public int GetLength()
        {
            return num;
```

```
        }
```

（2） 清空操作

清除链队列中的结点是使链队列为空，此时，链队列队头指示器 front 和队尾指示器 rear 等于 null 并且 num 为 0。

清空链队列的算法实现如下：

```
        public void Clear()
        {
            front = rear = null;
            num == 0;
        }
```

（3）判断链队列是否为空

如果链队列的队列队头指示器等于队尾指示器并且 num 为 0，则链队列为空，返回 true，否则返回 false。

判断链队列是否为空的算法实现如下：

```
        public bool IsEmpty()
        {
            if ((front== rear)&&(num==0))
            {
                return true;
            }
            else
            {
                return false;
            }
        }
```

（4）入队操作

链队列的入队操作在队尾添加一个新结点，num加1，队尾指示器rear指向新的结点。

入队操作的算法实现如下：

```
public void In()
{
    Node<T> q = new Node<T>(item);

    if (rear == null)
    {
        rear = q;
    }
    else
    {
        rear.Next = q;
        rear = q;
    }
}
```

（5） 出队操作

出队操作是在链队列不为空的情况下，先取出链队列头结点的值，然后将链队列队头指示器指向链队列头结点的直接后继结点，使之成为新的队列头结点，num 减 1。

出队操作的算法实现如下：

```
public T Out()
{
    if (IsEmpty())
    {
        Console.WriteLine("Queue is empty!");
        return default(T);
    }

    Node<T> p = front;
    front = front.Next;
    --num;

    return p.Data;
}
```

（6）获取链队列头结点的值

如果链队列不为空，返回链队列头结点的值，否则返回特殊值表示队列为空。

获取链队列头结点的值的算法实现如下：

```
public T GetFront()
{
    if (IsEmpty())
    {
        Console.WriteLine("Queue is empty!");
        return default(T);
    }

    return front.Data;
}
```

3.2.3 队列的应用举例

【例 3-4】编程判断一个字符串是否是回文。回文是指一个字符序列以中间字符为基准两边字符完全相同，如字符序列“ACBDEDBCA”是回文。

算法思想：判断一个字符序列是否是回文，就是把第一个字符与最后一个字符相比较，第二个字符与倒数第二个字符比较，依次类推，第 i 个字符与第 n-i 个字符比较。如果每次比较都相等，则为回文，如果某次比较不相等，就不是回文。因此，可以把字符序列分别入队列和栈，然后逐个出队列和出栈并比较出队列的字符和出栈的字符是否相等，若全部相等则该字符序列就是回文，否则就不是回文。

算法中的队列和栈采用什么存储结构都行，本例采用循环顺序队列和顺序栈来实现，其它的情况读者可作为习题，见习题三 3.8。程序中假设输入的都是英文字符而没有其它字符，对于输入其它字符情况的处理读者可以自己去完成。

使用循环顺序队列和顺序栈的程序如下：

```
public static void Main()
{
    SeqStack<char> s = new SeqStack<char>(50);
    CSeqQueue<char> q = new CSeqQueue<char>(50);
    string str = Console.ReadLine();

    for(int i = 0; i < str.Length; ++i)
    {
        s.Push(str[i]);
        q.In(str[i]);
    }

    while(!s.IsEmpty() && !q.IsEmpty())
    {
        if(s.Pop() != q.Out())
        {
            break;
        }
    }

    if(!s.IsEmpty() || !q.IsEmpty())
    {
        Console.WriteLine("这不是回文！");
    }
    else
    {
        Console.WriteLine("这是回文！");
    }
}
```

3.2.4 C# 中的队列

C#2.0 以下版本只提供了非泛型的 Queue 类，该类继承了 ICollection、IEnumerable 和 ICloneable 接口。C#2.0 提供了泛型的 Queue<T>类，该类继承了 IEnumerable<T>、ICollection 和 IEnumerable 接口。以下程序说明了泛型 Queue<T>类的主要方法，并对在我们自定义的队列类中没有出现的成员方法进行了注释，关于泛型 Queue<T>类的更具体的信息，读者可参考.NET Framework 的有关书籍。

```
public class Queue<T> : IEnumerable<T>, ICollection, IEnumerable
{
    public void Clear();

    //确定某元素是否在Queue<T>中。
    //如果在Queue<T> 中找到 item，则为true；否则为false。
    public bool Contains(T item);
```

```
        //从指定数组索引开始将Queue<T>元素复制到现有一维Array 中。
        public void CopyTo(T[] array, int arrayIndex);

        //移除并返回位于Queue<T>开始处的对象。
        //从Queue<T>的开头移除的对象。
        public T Dequeue();

        //返回位于Queue<T>开始处的对象但不将其移除。
        public T Peek();

        //将Queue<T>元素复制到新数组。
        public T[] ToArray();

        //如果元素数小于当前容量的90%，
        //将容量设置为Queue<T> 中的实际元素数。
        public void TrimExcess();
    }
```

本章小结

栈和队列是计算机中常用的两种数据结构，是操作受限的线性表。栈的插入和删除等操作都在栈顶进行，它是先进后出的线性表。队列的删除操作在队头进行，而插入、查找等操作在队尾进行，它是先进先出的线性表。与线性表一样，栈和队列有两种存储结构，顺序存储的栈称为顺序栈，链式存储的栈称为链栈。顺序存储的队列称为顺序对列，链式存储的队列称为链队列。

为解决顺序队列中的假溢出问题，采用循环顺序队列，但出现队空和队满的判断条件相同的问题，判断条件都是：front==rear。采用少用一个存储单元来解决该问题。此时，队满的判断条件是：(rear+1)%maxsize==front，判断队空的条件是：rear==front。

栈适合于具有先进后出特性的问题，如括号匹配、表达式求值等问题；队列适合于具有先进先出特性的问题，如排队等问题。

习题三

3.1 比较线性表、栈和队列这三种数据结构的相同点和不同点。

3.2 如果进栈的元素序列为 1,2,3,4，则可能得到的出栈序列有多少种？写出全部的可能序列。

3.3 如果进栈的元素序列为 A,B,C,D,E,F，能否得到 D,C,E,F,A,B 和 A,C,E,D,B,F 的出栈序列？并说明为什么不能得到或如何得到。

3.4 写出例题 3-3 中的 precede 函数和 Operate 函数。

3.5 写一算法将一顺序栈的元素依次取出，并打印元素值。

3.6 在顺序队列中，什么叫真溢出？什么叫假溢出？为什么顺序队列通常都采用循环顺序队列结构？

3.7 写一算法将一链队列的元素依次取出，并打印元素值。

3.8 将例题 3-4 用链队列和顺序队列实现，并能处理输入的字符不是大小写英文字符的情况。

第4章 串和数组

4.1 串

在应用程序中使用最频繁的类型是字符串。字符串简称串，是一种特殊的线性表，其特殊性在于串中的数据元素是一个个的字符。字符串在计算机的许多方面应用很广。如在汇编和高级语言的编译程序中，源程序和目标程序都是字符串数据。在事务处理程序中，顾客的信息如姓名、地址等及货物的名称、产地和规格等，都被作为字符串来处理。另外，字符串还具有自身的一些特性。因此，把字符串作为一种数据结构来研究。

4.1.1 串的基本概念

串(String)由 n(n≥0)字符组成的有限序列。一般记为：

S=”$c_1c_2\cdots c_n$” (n≥0)

其中，S是串名，双引号作为串的定界符，用双引号引起来的字符序列是串值。c_i（1≤i≤n）可以是字母、数字或其它字符，n为串的长度，当n=0 时，称为空串(Empty String)。

串中任意个连续的字符组成的子序列称为该串的子串(Substring)。包含子串的串相应地称为主串。子串的第一个字符在主串中的位置叫子串的位置。如串s_1”abcdefg”，它的长度是 7，串s_2”cdef”的长度是 4，s_2是s_1的子串，s_2的位置是 3。

如果两个串的长度相等并且对应位置的字符都相等，则称这两个串相等。而在 C#中，比较两个串是否相等还要看串的语言文化等信息。

4.1.2 串的存储及类定义

由于串中的字符都是连续存储的，而在 C#中串具有恒定不变的特性，即字符串一经创建，就不能将其变长、变短或者改变其中任何的字符。所以，这里不讨论串的链式存储，也不用接口来表示串的操作。同样，把串看作是一个类，类名为 StringDS。取名为 StringDS 是为了和 C#自身的字符串类 String 相区别。类 StringDS 只有一个字段，即存放串中字符序列的数组 data。由于串的运算有很多，在类 StringDS 中只包含部分基本的运算。串类 StringDS 的 C#实现如下所示：

```
public class StringDS
{
    private char[] data;   //字符数组

    //索引器
    public char this[int index]
    {
        get
        {
            return data[index];
        }
    }

    //构造器
    public StringDS(char[] arr)
    {
```

```
        data = new char[arr.Length];
        for(int i = 0; i < arr.Length; ++i)
        {
            data[i] = arr[i];
        }
    }

    //构造器
    public StringDS(StringDS s)
    {
        for(int i = 0; i < arr.Length; ++i)
        {
            data[i] = s[i];
        }
    }

    //构造器
    public StringDS(int len)
    {
        char[] arr = new char[len];
        data = arr;
    }

    //求串长
    public int GetLength()
    {
        return data.Length;
    }

    //串比较
    public int Compare(StringDS s)
    {
       int len=((this.GetLength()<=s.GetLength())?
                 this.GetLength():s.GetLength());
       int i = 0;
       for (i = 0; i < len; ++i)
       {
           if (this[i] != s[i])
           {
               break;
           }
       }

       if (i <= len)
```

```
        {
            if (this[i] < s[i])
            {
                return -1;
            }
            else if (this[i] > s[i])
            {
                return 1;
            }
        }
        else if(this.GetLength() == s.GetLength())
        {
            return 0;
        }
        else if (this.GetLength() < s.GetLength())
        {
            return -1;
        }

        return 1;
    }

    //求子串
    public StringDS SubString(int index, int len)
    {
        if ((index<0) || (index>this.GetLength() - 1)
            || (len<0) || (len>this.GetLength() - index))
        {
            Console.WriteLine("Position or Length is error!");
            return null;
        }

        StringDS s = new StringDS(len);

        for (int i = 0; i < len; ++i)
        {
            s[i] = this[i + index-1];
        }

        return s;
    }

     //串连接
      public StringDS Concat(StringDS s)
```

```
{
    StringDS s1 = new StringDS(this.GetLength() +
                  s.GetLength());

    for(int i = 0; i < this.GetLength(); ++i)
    {
        s1.data[i] = this[i];
    }

    for(int j = 0; j < s.GetLength(); ++j)
    {
        s1.data[this.GetLength() + j] = s[j];
    }

    return s1;
}

 //串插入
 public StringDS Insert(int index, StringDS s)
 {
    int len = s.GetLength();
    int len2 = len + this.GetLength();
    StringDS s1 = new StringDS(len2);

    if (index < 0 || index > this.GetLength() - 1)
    {
        Console.WriteLine("Position is error!");
        return null;
    }

    for (int i = 0; i < index; ++i)
    {
        s1[i] = this[i];
    }

    for(int i = index; i < index + len ; ++i)
    {
        s1[i] = s[i - index];
    }

    for (int i = index + len; i < len2; ++i)
    {
        s1[i] = this[i - len];
    }
```

```
        return s1;
    }

     //串删除
    public StringDS Delete(int index, int len)
    {
        if ((index<0) || (index>this.GetLength()-1)
            || (len<0) || (len>this.GetLength()-index))
        {
            Console.WriteLine("Position or Length is error!");
            return null;
        }

        StringDS s = new StringDS(this.GetLength() - len);

        for (int i = 0; i < index; ++i)
        {
            s[i] = this[i];
        }

        for (int i = index + len; i < this.GetLength(); ++i)
        {
            s[i] = this[i];
        }

        return s;
    }

     //串定位
     public int Index(StringDS s)
     {
        if (this.GetLength() < s.GetLength())
        {
            Console.WriteLine("There is not string s!");
            return -1;
        }

        int i = 0;
        int len = this.GetLength() - s.GetLength();
        while (i < len)
        {
            if (this.Compare(s) == 0)
            {
```

```
                break;
            }
        }

        if (i <= len)
        {
            return i;
        }

        return -1;
    }
}
```

4.1.3 串的基本操作的实现

串操作的基本实现有以下 7 种：

1、求串长

求串的长度就是求串中字符的个数，可以通过求数组 data 的长度来求串的长度。求串的长度的实现如下：

```
public int GetLength()
{
    return data.Length;
}
```

2、串比较

如果两个串的长度相等并且对应位置的字符相同，则串相等，返回 0；如果串 s 对应位置的字符大于该串的字符或者如果串 s 的长度大于该串，而在该串的长度返回内二者对应位置的字符相同，则返回-1，该串小于串 s；其余情况返回 1，该串大于串 s。

串比较的算法实现如下：

```
public int Compare(StringDS s)
{
    int len=((this.GetLength()<=s.GetLength())?
              this.GetLength():s.GetLength());
    int i = 0;
    for (i = 0; i < len; ++i)
    {
        if (this[i] != s[i])
        {
            break;
        }
    }

    if (i <= len)
    {
        if (this[i] < s[i])
        {
```

```
                return -1;
            }
            else if (this[i] > s[i])
            {
                return 1;
            }
        }
        else if(this.GetLength() == s.GetLength())
        {
            return 0;
        }
        else if (this.GetLength() < s.GetLength())
        {
            return -1;
        }

        return 1;
    }
```

3、求子串

从主串的index位置起找长度为len的子串，若找到，返回该子串，否则，返回一个空串。

算法实现如下：

```
    public StringDS SubString(int index, int len)
    {
        if ((index<0) || (index>this.GetLength()-1)
            || (len<0) || (len>this.GetLength()-index))
        {
            Console.WriteLine("Position or Length is error!");
            return null;
        }

        StringDS s = new StringDS(len);

        for (int i = 0; i < len; ++i)
        {
            s[i] = this[i + index-1];
        }

        return s;
    }
```

4、串连接

将一个串和另外一个串连接成一个串，其结果返回一个新串，新串的长度是两个串的长度之和，新串的前部分是原串，长度为该串的长度，新串的后部分是串s，长度为串s的长度。

串连接的算法实现如下：

```
public StringDS Concat(StringDS s)
{
    StringDS s1 = new StringDS(this.GetLength()
                + s.GetLength());

    for(int i = 0; i < this.GetLength(); ++i)
    {
        s1.data[i] = this[i];
    }

    for(int j = 0; j < s.GetLength(); ++j)
    {
        s1.data[this.GetLength() + j] = s[j];
    }

    return s1;
}
```

5、串插入

串插入是在一个串的位置index处插入一个串s。如果位置符合条件，则该操作返回一个新串，新串的长度是该串的长度与串s的长度之和，新串的第1部分是该串的开始字符到第index之间的字符，第2部分是串s,第3部分是该串从index位置字符到该串的结束位置处的字符。如果位置不符合条件，则返回一个空串。

串插入的算法如下：

```
public StringDS Insert(int index, StringDS s)
{
    int len = s.GetLength();
    int len2 = len + this.GetLength();
    StringDS s1 = new StringDS(len2);

    if (index < 0 || index > this.GetLength() - 1)
    {
        Console.WriteLine("Position is error!");
        return null;
    }

    for (int i = 0; i < index; ++i)
    {
        s1[i] = this[i];
    }

    for(int i = index; i < index + len ; ++i)
    {
        s1[i] = s[i - index];
```

```
            }

            for (int i = index + len; i < len2; ++i)
            {
                s1[i] = this[i - len];
            }

            return s1;
        }
```

6、串删除

串删除是从把串的第index位置起连续的len个字符的子串从主串中删除掉。如果位置和长度符合条件，则该操作返回一个新串，新串的长度是原串的长度减去len，新串的前部分是原串的开始到第index个位置之间的字符，后部分是原串从第index+len位置到原串结束的字符。如果位置和长度不符合条件，则返回一个空串。

串删除的算法实现如下：

```
        public StringDS Delete(int index, int len)
        {
            if ((index<0) || (index>this.GetLength()-1)
                || (len<0) || (len>this.GetLength()-index))
            {
                Console.WriteLine("Position or Length is error!");
                return null;
            }

            StringDS s = new StringDS(this.GetLength() - len);

            for (int i = 0; i < index; ++i)
            {
                s[i] = this[i];
            }

            for (int i = index + len; i < this.GetLength(); ++i)
            {
                s[i] = this[i];
            }

            return s;
        }
```

7、串定位

查找子串s在主串中首次出现的位置。如果找到，返回子串s在主串中首次出现的位置，否则，返回-1。

串定位的算法实现如下：

```
        public int Index(StringDS s)
```

```
    {
        if (this.GetLength() < s.GetLength())
        {
            Console.WriteLine("There is not string s!");
            return -1;
        }

        int i = 0;
        int len = this.GetLength() - s.GetLength();
        while (i < len)
        {
            if (Compare(s) == 0)
            {
                break;
            }
        }

        if (i <= len)
        {
            return i;
        }

        return -1;
    }
```

4.1.4 C#中的串

在 C#中，一个 String 表示一个恒定不变的字符序列集合。String 类型是封闭类型，所以，它不能被其它类继承，而它直接继承自 object。因此，String 是引用类型，不是值类型，在托管堆上而不是在线程的堆栈上分配空间。String 类型还继承了 IComparable、ICloneable、IConvertible、IComparable<string>、IEnumerable<char>、IEnumerable 和 IEquatable<string>等接口。String 的恒定性指的是一个串一旦被创建，就不能将其变长、变短或者改变其中任何的字符。所以，当我们对一个串进行操作时，不能改变字符串，如在本书定义的 StringDS 类中，串连接、串插入和串删除等操作的结果都是生成了新串而没有改变原串。C#也提供了 StringBuilder 类型来支持高效地动态创建字符串。

在 C#中，创建串不能用 new 操作符，而是使用一种称为字符串驻留的机制。这是因为 C#语言将 String 看作是基元类型。基元类型是被编译器直接支持的类型，可以在源代码中用文本常量(Literal)来直接表达字符串。当 C#编译器对源代码进行编译时，将文本常量字符串存放在托管模块的元数据中。而当 CLR 初始化时，CLR 创建一个空的散列表，其中的键是字符串，值为指向托管堆中字符串对象的引用。散列表就是哈希表，关于散列表的详细介绍见 8.4 小节。当 JIT 编译器编译方法时，它会在散列表中查找每一个文本常量字符串。如果找不到，就会在托管堆中构造一个新的 String 对象（指向字符串），然后将该字符串和指向该字符串对象的引用添加到散列表中；如果找到了，不会执行任何操作。

C#提供的 String 类型中的方法很多，比如构造器有 8 个，比较两个字符串的

方法有 12 个，连接字符串的方法有 9 个。以下列出了该类型中常用的方法，并对每个方法给出了注释。关于 String 更为详细的介绍参考.NET 的有关书籍。

```
public sealed class String:IComparable,ICloneable,IConvertible,
  IComparable<string>,IEnumerable<char>,IEnumerable,IEquatable<string>
{
    //将串初始化为由字符数组指示的值。
    public String(char[] value);

    //确定两个指定的 String 对象是否具有不同的值。
    public static bool operator !=(string a, string b);

    //确定两个指定的 String 对象是否具有同一值。
    public static bool operator ==(string a, string b);

    //返回对此串的引用。
    public object Clone();

    //返回两个串的排序情况。
    public static int Compare(string strA, string strB);

    //返回与指定的串的排序情况。
    public int CompareTo(string strB);

    //返回两个串的字符集的情况。
    public static int CompareOrdinal(string strA, string strB);

    //连接两个串。
    public static string Concat(string str0, string str1);

    //创建一个与指定的串具有相同值的串。
    public static string Copy(string str);

    //将指定数目的字符从指定位置复制到字符数组中的指定位置。
    public void CopyTo(int sourceIndex,char[] destination,
                       int destinationIndex,int count);

    //确定串的开头是否与指定的串匹配。
    public bool StartsWith(string value, StringComparison comparisonType);

    //确定串的末尾是否与指定的串匹配。
    public bool EndsWith(string value, StringComparison comparisonType);

    //将指定的格式项替换为指定的 Object 实例的值的文本等效项。
```

```
        public static string Format(string format, object arg0);

        //返回指定的串第一个匹配项的索引。
        public int IndexOf(string value);

        //在串的指定索引位置插入一个串。
        public string Insert(int startIndex, string value);

        //从串的指定位置开始删除指定数目的字符。
        //一个新的 String，它等于此实例减去 count 数目的字符。
        public string Remove(int startIndex, int count);

      //将串的所有匹配项替换为其他指定的串。
      public string Replace(string oldValue, string newValue);

      //从串指定的字符位置开始检索子字符串
      public string Substring(int startIndex, int length);

      //将串中的字符转换为小写形式。
      public string ToLower();

      //将串中的字符转换为大写形式。
      public string ToUpper();

      //从开始位置和末尾移除空白字符的所有匹配项，
      //如果 trimChars 为 null，则改为移除空白字符。
      public string Trim(params char[] trimChars);
    }
```

4.2 数组

数组是一种常用的数据结构，可以看作是线性表的推广。数组作为一种数据结构，其特点是结构中的数据元素可以是具有某种结构的数据，甚至可以是数组，但属于同一数据类型。数组在许多高级语言里面都被作为固定类型来使用。

4.2.1 数组的逻辑结构

数组是 n(n≥1)个相同数据类型的数据元素的有限序列。一维数组可以看作是一个线性表，二维数组可以看作是“数据元素是一维数组”的一维数组，三维数组可以看作是“数据元素是二维数组”的一维数组，依次类推。

图 4.1 是一个 m 行 n 列的二维数组。

$$A=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

图 4.1 m 行 n 列的二维数组

数组是一个具有固定格式和数量的数据有序集，每一个数据元素通过唯一的下标来标识和访问。通常，一个数组一经定义，每一维的大小及上下界都不能改

变。所以，在数组上不能进行插入、删除数据元素等操作。数组上的操作一般有：
1、取值操作：给定一组下标，读其对应的数据元素；
2、赋值操作：给定一组下标，存储或修改与其对应的数据元素；
3、清空操作：将数组中的所有数据元素清除；
4、复制操作：将一个数组的数据元素赋给另外一个数组；
5、排序操作：对数组中的数据元素进行排序，这要求数组中的数据元素是可排序的；
6、反转操作：反转数组中数据元素的顺序。

4.2.2 数组的内存映象

通常，采用顺序存储结构来存储数组中的数据元素，因为数组中的元素要求连续存放。本质上，计算机的内存是一个一维数组，内存地址就是数组的下标。所以，对于一维数组，可根据数组元素的下标得到它的存储地址，也可根据下标来访问一维数组中的元素。而对于多维数组，需要把多维的下标表达式转换成一维的下标表达式。当行列固定后，要用一组连续的存储单元存放数组中的元素，有一个次序约定问题，这产生了两种存储方式：一种是以行序为主序（先行后列）的顺序存放，另一种是以列序为主序（先列后行）的顺序存放。图 4.2 给出了图 4.1 中的二维数组的两种存放方式示意图。

下面按元素的下标求地址：

当以行序为主序进行存储时，设数组的基址是$Loc(a_{11})$，每个数据元素占w个存储单元，则a_{11}的物理地址可由下式计算：

$$Loc(a_{ij})= Loc(a_{11})+((i-1)*n+j-1)*w \qquad (4\text{-}1)$$

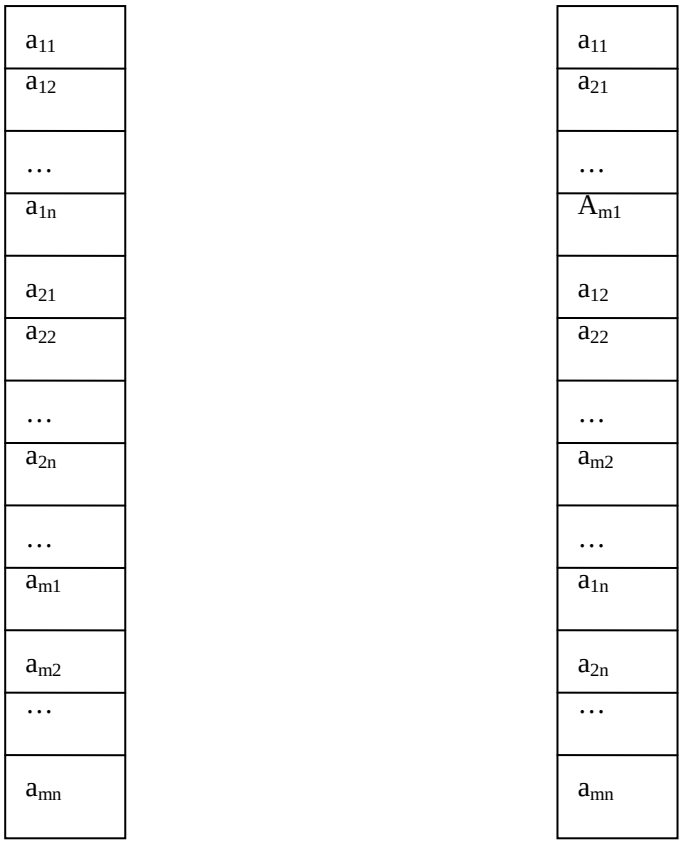
这是因为数组元素a_{ij}的前面有i-1 行，每一行有n个数据元素，在第i行中a_{ij}的前面还有j-1 个元素。

当以列序为主序进行存储时，则a_{11}的物理地址可由下式计算：

$$Loc(a_{ij})= Loc(a_{11})+((j-1)*m+i-1)*w \qquad (4\text{-}2)$$

这是因为数组元素a_{ij}的前面有j-1 列，每一列有m个数据元素，在第j列中a_{ij}的前面还有i-1 个元素。

由以上的公式可知，数组元素的存储位置是其下标的线性函数，一旦确定了数组各维的长度，就可以计算任意一个元素的存储地址，并且时间相等。所以，存取数组中任意一个元素的时间也相等，因此，数组是一种随机存储结构。



(a) 以行为主序　　(b) 以列为主序

图 4.2 m 行 n 列二维数组的存放方式

4.2.3 C#中的数组

C#支持一维数组、多维数组及交错数组（数组的数组）。所有的数组类型都隐含继承自 System.Array。Array 是一个抽象类，本身又继承自 System.Object。所以，数组总是在托管堆上分配空间，是引用类型。任何数组变量包含的是一个指向数组的引用，而非数组本身。当数组中的元素的值类型时，该类型所需的内存空间也作为数组的一部分而分配；当数组的元素是引用类型时，数组包含是只是引用。Array 还继承了 ICloneable、IList、ICollection、IEnumerable 等接口。

C#中的数组一般是 0 基数组（最小索引为 0），这是为了和其它语言共享代码。C#也支持非 0 基数组。C#除了能创建静态数组外，还可以创建动态数组，这通过使用 Array 的静态方法 CreateInstance 方法来实现。

与 String 一样，Array 中的方法有许多。以下列出了 Array 类型中常用的方法，并对每个方法给出了注释。关于 Array 更为详细的介绍参考.NET 的有关书籍。

```
public abstract class Array : ICloneable, IList, ICollection, IEnumerable
{
    //判断 Array 是否具有固定大小。
    public bool IsFixedSize {get;}

   //获取 Array 元素的个数。
```

```
    public int Length{get;}

    //获取 Array 的秩（维数）。
    public int Rank { get; }

    //实现的 IComparable 接口，在.Array 中搜索特定元素。
    public static int BinarySearch(Array array, object value);

  //实现的 IComparable<T>泛型接口，在 Array 中搜索特定元素。
  public static int BinarySearch<T>(T[] array, T value);

  //实现 IComparable 接口，在 Array 的某个范围中搜索值。
  public static int BinarySearch(Array array, int index,
                                 int length,object value);

//实现的 IComparable<T>泛型接口，在 Array 中搜索值。
public static int BinarySearch<T>(T[] array,
                                  int index, int length, T value);

//Array 设置为零、false 或 null，具体取决于元素类型。
public static void Clear(Array array, int index, int length);

//System.Array 的浅表副本。
public object Clone();

//从第一个元素开始复制 Array 中的一系列元素
//到另一 Array 中（从第一个元素开始）。
public static void Copy(Array sourceArray,
                        Array destinationArray, int length);
//将一维 Array 的所有元素复制到指定的一维 Array 中。
public void CopyTo(Array array, int index);

//创建使用从零开始的索引、具有指定 Type 和维长的多维 Array。
public static Array CreateInstance(Type elementType,
                                   params int[] lengths);
//返回 ArrayIEnumerator。
public IEnumerator GetEnumerator();

//获取 Array 指定维中的元素数。
public int GetLength(int dimension);

//获取一维 Array 中指定位置的值。
public object GetValue(int index);
```

```
        //返回整个一维 Array 中第一个匹配项的索引。
        public static int IndexOf(Array array, object value);

        //返回整个.Array 中第一个匹配项的索引。
        public static int IndexOf<T>(T[] array, T value);

      //返回整个一维 Array 中最后一个匹配项的索引。
      public static int LastIndexOf(Array array, object value);

      //反转整个一维 Array 中元素的顺序。
      public static void Reverse(Array array);

    //设置给一维 Array 中指定位置的元素。
    public void SetValue(object value, int index);

    //对整个一维 Array 中的元素进行排序。
    public static void Sort(Array array);
}
```

本章小结

字符串简称串，是在应用程序中使用最频繁的数据类型。串由n(n≥0)字符组成的有限序列，一般记为S=”$c_1c_2\cdots c_n$”。串中的字符都是连续存储的，而在C#中串具有恒定不变的特性，即字符串一经创建，就不能将其变长、变短或者改变其中任何的字符。所以，串一般采用顺序存储。串的基本操作有求串长、串比较、求子串、串连接、串插入、串删除和串定位等操作。

数组可以看作是线性表的推广，其特点是结构中的数据元素属于同一数据类型。数组是n(n≥1)个相同数据类型的数据元素的有限序列。通常，数组采用顺序存储结构来存储数组中的数据元素，存放方式有两种：以行序为主序（先行后列）的顺序存放和以列序为主序（先列后行）的顺序存放。

习题四

1.5 设s=”I am a teacher”,i=”excellent”,r=”student”。用StringDs类中的方法求：

（1） 串s、i、r的长度；

（2） S.SubString(8, 7)、i.SubString(2, 1)；

（3） S.IndexOf(“tea”)、i.IndexOf(“cell”)、r.IndexOf(“den”)。

1.6 串的替换操作是指：已知串s、t、r，用r替换s中出现的所有与t相等且不重叠的子串。写出算法，方法名为Replace。

1.7 已知下列字符串：

a=”THIS”,f=”A SMPLE” c=”GOOD”,d=”NE”,b=”␣”,g=”IS”,

s=a.Concat(b.Concat(a.SubString(3,2)).(f.SubString(2,7))),

t=f.Replace(f.SubString(3,6),c),

u=c.SubString(3,1).Concat(d),

v=s.Concat(b.Concat(t.ConCat(b.Concat(u))))。

问s,t,v,GetLength(s),v.IndexOf(g),u.IndexOf(g)各是什么。

1.8 设已知两个串为：

S1=”bc cad cabcadf”,S2=”abc”。试求两个串的长度，并判断 S2 串是否是 S1 串的子串，如果 S2 是 S1 的子串，指出 S2 在 S1 中的起始位置。

1.9 已知：s=”(XYZ)+*”，t=”(X+Z)*Y”，试利用连接、求子串和替换等基本运算，将 s 转化为 t。

第5章　树和二叉树

前面几章介绍了线性结构，线性结构中的数据元素是一对一的关系。本章和下一章介绍两种非常重要的非线性结构：树形结构和图状结构。树形结构是一对多的非线性结构，非常类似于自然界中的树，数据元素之间既有分支关系，又有层次关系。树形结构在现实世界中广泛存在，如家族的家谱、一个单位的行政机构组织等都可以用树形结构来形象地表示。树形结构在计算机领域中也有着非常广泛的应用，如 Windows 操作系统中对磁盘文件的管理、编译程序中对源程序的语法结构的表示等都采用树形结构。在数据库系统中，树形结构也是数据的重要组织形式之一。树形结构有树和二叉树两种，树的操作实现比较复杂，但树可以转换为二叉树进行处理，所以，本章主要讨论二叉树。

5.1 树

5.1.1 树的定义

树(Tree)是 n(n≥0)个相同类型的数据元素的有限集合。树中的数据元素叫结点(Node)。n=0 的树称为空树(Empty Tree)；对于 n＞0 的任意非空树 T 有：

（1）有且仅有一个特殊的结点称为树的根(Root)结点，根没有前驱结点；

（2）若n＞1，则除根结点外，其余结点被分成了m(m＞0)个互不相交的集合T_1，T_2，…，T_m，其中每一个集合T_i(1≤i≤m)本身又是一棵树。树T_1，T_2，…，T_m称为这棵树的子树(Subtree)。

由树的定义可知，树的定义是递归的，用树来定义树。因此，树（以及二叉树）的许多算法都使用了递归。

树的形式定义为：树(Tree)简记为 T，是一个二元组，

T = (D, R)

其中：D 是结点的有限集合；

R 是结点之间关系的有限集合。

在实际生活中树的例子很多。就说我们的《数据结构》书吧，它总的知识点分成三个部分：线性结构部分、非线性结构部分和应用部分。第一部分由线性表、栈和队列及串和数组等章节组成，第二部分由树和二叉树及图等章节组成，第三部分由排序和查找等章节组成。当然每一章节又由几个小节的内容组成，这是一个典型的树的实例。

图 5.1 是一棵具有 10 个结点的树，即T={A,B,C,D,E,F,G,H,I,J}。结点A是树T的根结点，根结点A没有前驱结点。除A之外的其余结点分成了三个互不相交的集合：T_1={B,E,F,G}，T_2={C,H}，T_3={ D,I,J}，分别形成了三棵子树，B、C和D分别成为这三棵子树的根结点，因为这三个结点分别在这三棵子树中没有前驱结点。

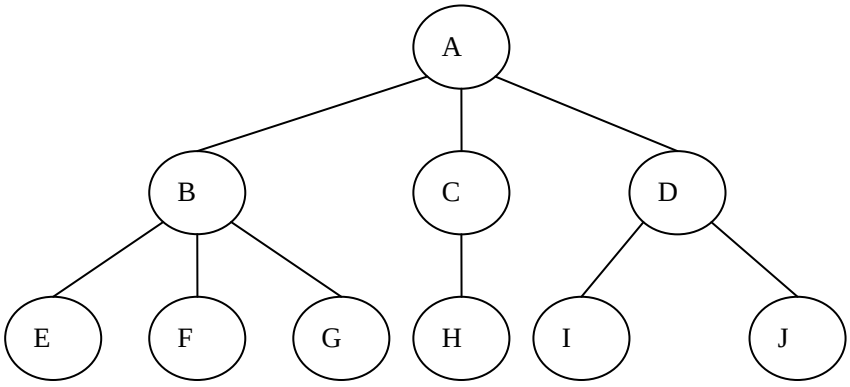


图 5.1 树的示意图

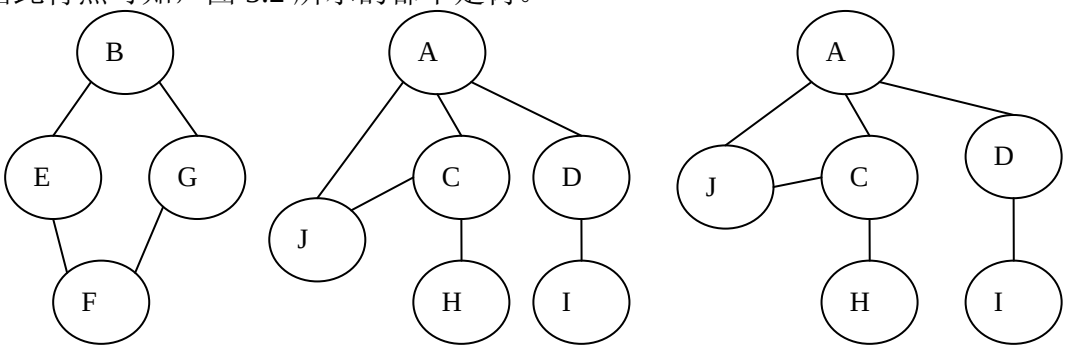
从树的定义和图 5.1 的示例可以看出，树具有下面两个特点：

（1）树的根结点没有前驱结点，除根结点之外的所有结点有且只有一个前驱结点。

（2）树中的所有结点都可以有零个或多个后继结点。

实际上，第（1）个特点表示的就是树形结构的“一对多关系”中的“一”，第（2）特点表示的是“多”。

由此特点可知，图 5.2 所示的都不是树。



（a） （b） （c）

图 5.2 非树结构示意图

5.1.2 树的相关术语

树的相关术语有以下一些：

1、结点(Node)：表示树中的数据元素，由数据项和数据元素之间的关系组成。在图 5.1 中，共有 10 个结点。

2、结点的度(Degree of Node)：结点所拥有的子树的个数，在图 5.1 中，结点 A 的度为 3。

3、树的度(Degree of Tree)：树中各结点度的最大值。在图 5.1 中，树的度为 3。

4、叶子结点(Leaf Node)：度为 0 的结点，也叫终端结点。在图 5.1 中，结点 E、F、G、H、I、J 都是叶子结点。

5、分支结点(Branch Node)：度不为 0 的结点，也叫非终端结点或内部结点。在图 5.1 中，结点 A、B、C、D 是分支结点。

6、孩子(Child)：结点子树的根。在图 5.1 中，结点 B、C、D 是结点 A 的孩子。

7、双亲(Parent)：结点的上层结点叫该结点的双亲。在图 5.1 中，结点 B、C、D 的双亲是结点 A。

8、祖先(Ancestor)：从根到该结点所经分支上的所有结点。在图 5.1 中，结点 E 的祖先是 A 和 B。

9、子孙(Descendant)：以某结点为根的子树中的任一结点。在图 5.1 中，除 A 之外的所有结点都是 A 的子孙。

10、兄弟(Brother)：同一双亲的孩子。在图 5.1 中，结点 B、C、D 互为兄弟。

11、结点的层次(Level of Node)：从根结点到树中某结点所经路径上的分支数称为该结点的层次。根结点的层次规定为 1，其余结点的层次等于其双亲结点的层次加 1。

12、堂兄弟(Sibling)：同一层的双亲不同的结点。在图 5.1 中，G 和 H 互为堂兄弟。

13、树的深度(Depth of Tree)：树中结点的最大层次数。在图 5.1 中，树的深度为 3。

14、无序树(Unordered Tree)：树中任意一个结点的各孩子结点之间的次序构成无关紧要的树。通常树指无序树。

15、有序树(Ordered Tree)：树中任意一个结点的各孩子结点有严格排列次序的树。二叉树是有序树，因为二叉树中每个孩子结点都确切定义为是该结点的左孩子结点还是右孩子结点。

16、森林(Forest)：m(m≥0)棵树的集合。自然界中的树和森林的概念差别很大，但在数据结构中树和森林的概念差别很小。从定义可知，一棵树有根结点和 m 个子树构成，若把树的根结点删除，则树变成了包含 m 棵树的森林。当然，根据定义，一棵树也可以称为森林。

5.1.3 树的逻辑表示

树的逻辑表示方法很多，这里只讲几种常见的表示方法。

1、直观表示法

它象日常生活中的树木一样。整个图就象一棵倒立的树，从根结点出发不断扩展，根结点在最上层，叶子结点在最下面，如图 5.1 所示。

2、凹入表示法

每个结点对应一个矩形，所有结点的矩形都右对齐，根结点用最长的矩形表示，同一层的结点的矩形长度相同，层次越高，矩形长度越短，图 5.1 中的树的凹入表示法如图 5.3 所示。

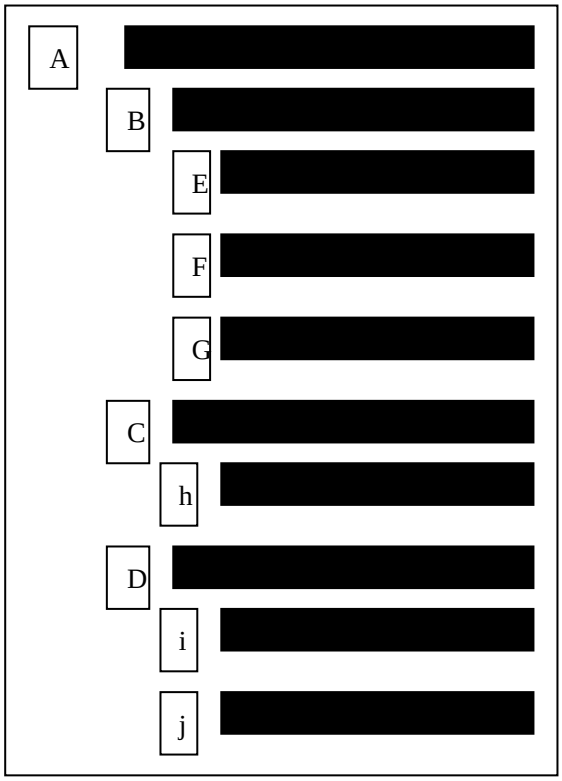


图 5.3 树的凹入表示法

3、广义表表示法

用广义表的形式表示根结点排在最前面，用一对圆括号把它的子树结点括起

来，子树结点用逗号隔开。图 5.1 的树的广义表表示如下：

（A（B（E，F，G），C（H），D（I，J）））

4、嵌套表示法

类似数学中所说的文氏图表示法，如图 5.4 所示。

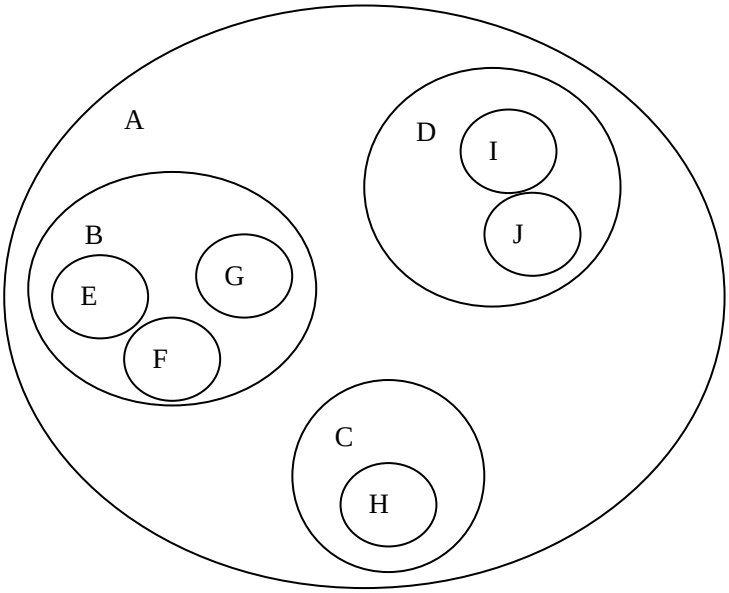


图 5.4 树的嵌套表示法

5.1.4 树的基本操作

树的操作很多，比如访问根结点，得到结点的值、求结点的双亲结点、某个子结点和某个兄弟结点。又比如，插入一个结点作为某个结点的最左子结点、最右子结点等。删除结点也是一样。也可按照某种顺序遍历一棵树。在这些操作中，有些操作是针对结点的（访问父亲结点、兄弟结点或子结点），有些操作是针对整棵树的（访问根结点、遍历树）。如果象前面几种数据结构用接口表示树的操作的话，就必须把结点类的定义写出来。但本章的重点不是树而是二叉树。所以，树的操作不用接口来表示，只给出操作的名称和功能。

树的基本操作通常有以下 10 种：

1、Root()：求树的根结点，如果树非空，返回根结点，否则返回空；

2、Parent(t)：求结点 t 的双亲结点。如果 t 的双亲结点存在，返回双亲结点，否则返回空；

3、Child(t,i)：求结点 t 的第 i 个子结点。如果存在，返回第 i 个子结点，否则返回空；

4、RightSibling(t)：求结点 t 第一个右边兄弟结点。如果存在，返回第一个右边兄弟结点，否则返回空；

5、Insert(s,t,i)：把树 s 插入到树中作为结点 t 的第 i 棵子树。成功返回 true，否则返回 false；

6、Delete(t,i)：删除结点 t 的第 i 棵子树。成功返回第 i 棵子树的根结点，否则返回空；

7、Traverse(TraverseType)：按某种方式遍历树；

8、Clear()：清空树；

9、IsEmpty()：判断树是否为空树。如果是空树，返回 true，否则返回 false；

10、GetDepth()：求树的深度。如果树不为空，返回树的层次，否则返回 0。

5.2 二叉树

5.2.1 二叉树的定义

二叉树(Binary Tree)是 n(n≥0)个相同类型的结点的有限集合。n=0 的二叉树称为空二叉树(Empty Binary Tree)；对于 n＞0 的任意非空二叉树有：

（1）有且仅有一个特殊的结点称为二叉树的根(Root)结点，根没有前驱结点；

（2）若n＞1，则除根结点外，其余结点被分成了 2 个互不相交的集合T_L，T_R，而T_L、T_R本身又是一棵二叉树，分别称为这棵二叉树的左子树(Left Subtree)和右子树(Right Subtree)。

二叉树的形式定义为：二叉树(Binary Tree)简记为 BT，是一个二元组，

BT = (D, R)

其中：D 是结点的有限集合；

R 是结点之间关系的有限集合。

由树的定义可知，二叉树是另外一种树形结构，并且是有序树，它的左子树和右子树有严格的次序,若将其左、右子树颠倒，就成为另外一棵不同的二叉树。因此，图 5.5(a)和图 5.5 (b)所示是不同的二叉树。

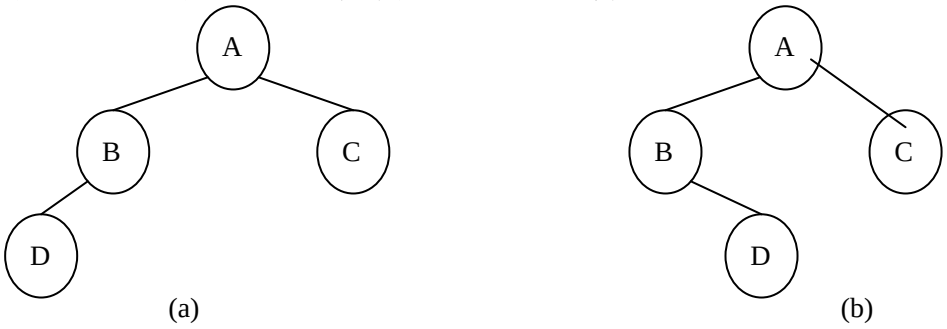


图 5.5 两棵不同的二叉树

二叉树的形态共有 5 种：空二叉树、只有根结点的二叉树、右子树为空的二叉树、左子树为空的二叉树和左、右子树非空的二叉树。二叉树的 5 种形态如图 5.6 所示。

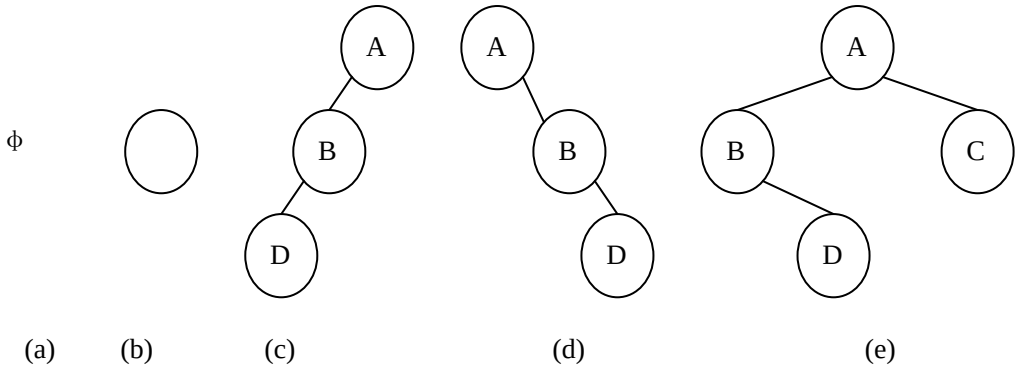


图 5.6 二叉树的 5 种形态

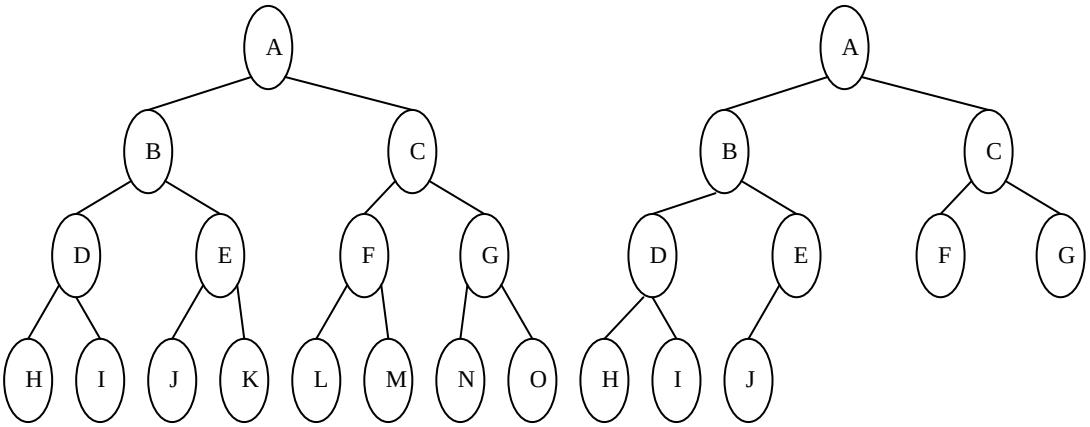
下面介绍两种特殊的二叉树。

（1）满二叉树(Full Binary Tree)：如果一棵二叉树只有度为 0 的结点和度为 2 的结点，并且度为 0 的结点在同一层上，则这棵二叉树为满二叉树，如图 5.7(a)所示。

由定义可知，对于深度为k的满二叉树的结点个数为 2^k-1。

（2）完全二叉树(Complete Binary Tree)：深度为 k，有 n 个结点的二叉树当且仅当其每一个结点都与深度为 k，有 n 个结点的满二叉树中编号从 1 到 n 的结点一一对应时，称为完全二叉树，如图 5.7(b)所示。

完全二叉树的特点是叶子结点只可能出现在层次最大的两层上，并且某个结点的左分支下子孙的最大层次与右分支下子孙的最大层次相等或大 1。



(a)满二叉树　　(b)完全二叉树

图 5.7 满二叉树与完全二叉树

5.2.2 二叉树的性质

性质 1　一棵非空二叉树的第 i 层上最多有 2^{i-1} 个结点 $(i \geqslant 1)$。

证明：采用数学归纳法进行证明。当n=1 时，二叉树只有 1 层，这一层只有根结点一个结点，所以第 1 层的结点数为 $2^{1-1}=1$，结论成立。假设当n=N时结论成立，即第N层最多有 2^{N-1} 个结点；当n=N+1 时，根据二叉树的定义，第N层的每个结点最多有 2 个子结点，所以第N+1 层上最多有 $2^{N-1}*2=2^{N}=2^{(N+1)-1}$ 个结点，结论成立。综上所述，性质 1 成立。

性质 2　若规定空树的深度为 0，则深度为k的二叉树最多有 $2^{k}-1$ 个结点 $(k \geqslant 0)$。

证明：当k=0 时，空树的结点数为 $2^{0}-1=0$，结论成立。当深度为k(k＞0)时，由性质 1 可知，第i(1≤i≤k)层最多有 2^{i-1} 个结点，所以二叉树的最多结点数是：

$$\sum_{i=1}^{k} 2^{i-1} = 2^{k} - 1$$

性质 3　具有n个结点的完全二叉树的深度k为 $\log_2 n+1$。

证明：根据性质 2 和完全二叉树的定义可知，当一棵完全二叉树的结点数为 n、深度为 k 时，有

$$2^{k-1}-1<n\leqslant 2^{k}-1$$

即　　$2^{k-1}\leqslant n<2^{k}$

对不等式取对数，有　　$k-1\leqslant \log_2 n<k$

由于k是整数，所以有 $k=\log_2 n+1$。

性质 4　对于一棵非空二叉树，如果度为 0 的结点数目为 n_0，度为 2 的结点数目为 n_2，则有 $n_0= n_2+1$。

证明：设n为二叉树的结点总数，n_1二叉树中度为 1 的结点数目，则有

$$n= n_0+ n_1+ n_2 \qquad (5\text{-}1)$$

在二叉树中，除根结点外，其余结点都有唯一的一个进入分支。设 B 为二叉树中的分支总数，则有

$$B=n-1 \qquad (5\text{-}2)$$

这些分支由度为 1 和度为 2 的结点发出的，一个度为 1 的结点发出一个分支，一个度为 2 的结点发出 2 个分支，所以有

$$B= n_1+2\ n_2 \qquad (5\text{-}3)$$

综合上面 3 个式子，可以得到

$$n_0= n_2+1$$

性质 5 对于具有 n 个结点的完全二叉树，如果按照从上到下和从左到右的顺序对所有结点从 1 开始编号，则对于序号为 i 的结点，有：

（1）如果 i>1，则序号为 i 的结点的双亲结点的序号为 i/2（“/”表示整除）；如果 i=1，则该结点是根结点，无双亲结点。

（2）如果 2i≤n，则该结点的左孩子结点的序号为 2i；若 2i>n，则该结点无左孩子。

（3）如果 2i+1≤n，则该结点的右孩子结点的序号为 2i+1；若 2i+1>n，则该结点无右孩子。

性质 5 的证明比较复杂，故省略。我们用实际例子检验性质 5 的正确性。对于图 5.7(b)所示的完全二叉树，如果按照从上到下和从左到右的顺序对所有结点从 1 开始编号，则结点和结点序号的对应关系如下：

1	2	3	4	5	6	7	8	9	10
A	B	C	D	E	F	G	H	I	J

该完全二叉树的结点总数 n=10。对于结点 D，相应的序号为 4，则结点 D 的双亲结点的序号为 4/2=2，即结点 B；左孩子结点的序号为 2i=2*4=8，即结点 H；右孩子结点的序号为 2i+1=2*4+1=9，即结点 I。对于结点 E，相应的序号为 5，则结点 E 的双亲结点的序号为 5/2=2，即结点 B；左孩子结点的序号为 2i=2*5=10，即结点 J；右孩子结点的序号为 2i+1=2*5+1=11>n=10，即结点 E 没有右孩子。

5.2.3 二叉树的存储结构

二叉树的存储结构主要有三种：顺序存储结构、二叉链表存储结构和三叉链表存储结构。

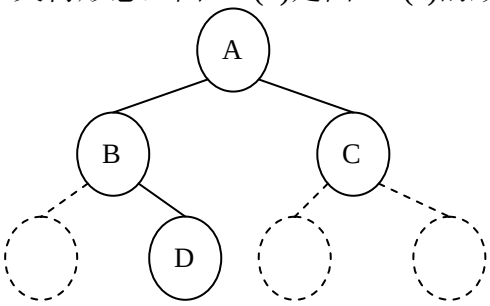
1、二叉树的顺序存储结构

对于一棵完全二叉树，由性质 5 可计算得到任意结点 i 的双亲结点序号、左孩子结点序号和右孩子结点序号。所以，完全二叉树的结点可按从上到下和从左到右的顺序存储在一维数组中，其结点间的关系可由性质 5 计算得到，这就是二叉树的顺序存储结构。图 5.7(a)所示的二叉树的顺序存储结构为：

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

但是，对于一棵非完全二叉树，不能简单地按照从上到下和从左到右的顺序

存放在一维数组中，因为数组下标之间的关系不能反映二叉树中结点之间的逻辑关系。所以，应该对一棵非完全二叉树进行改造，增加空结点（并不存在的结点）使之成为一棵完全二叉树，然后顺序存储在一维数组中。图 5.8(a)是图 5.6(e)的完全二叉树形态，图 5.8(b)是图 5.8(a)的顺序存储示意图。

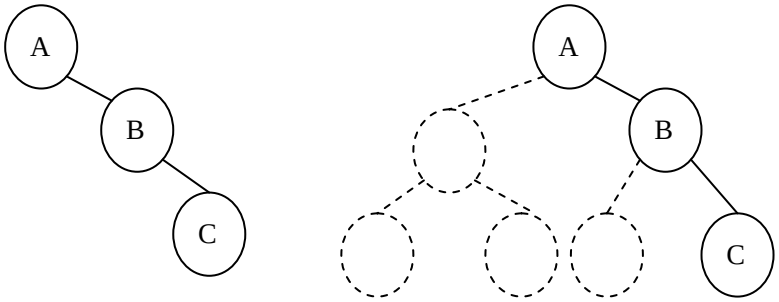


A	B	C	∧	D	∧	∧

(a) 改造后的完全二叉树　　(b) 改造后的完全二叉树顺序存储示意图

图 5.8 一般二叉树的改造及其顺序存储示意图

显然，顺序存储对于需增加很多空结点才能改造为一棵完全二叉树的二叉树不适合，因为会造成空间的大量浪费。实际上，采用顺序存储结构，是对非线性的数据结构线性化，用线性结构来表示二叉树的结点之间的逻辑关系，所以，需要增加空间。一般来说，有大约一半的空间被浪费。最差的情况是右单支树，如图 5.9 所示，一棵深度为k的右单支树，只有k个结点，却需要分配 2^k-1 个存储单元。



(a) 一棵右单支树　　(b) 改造后的右单支树对应的完全二叉树

A	∧	B	∧	∧	∧	C

(c) 改造后的右单支树顺序存储示意图

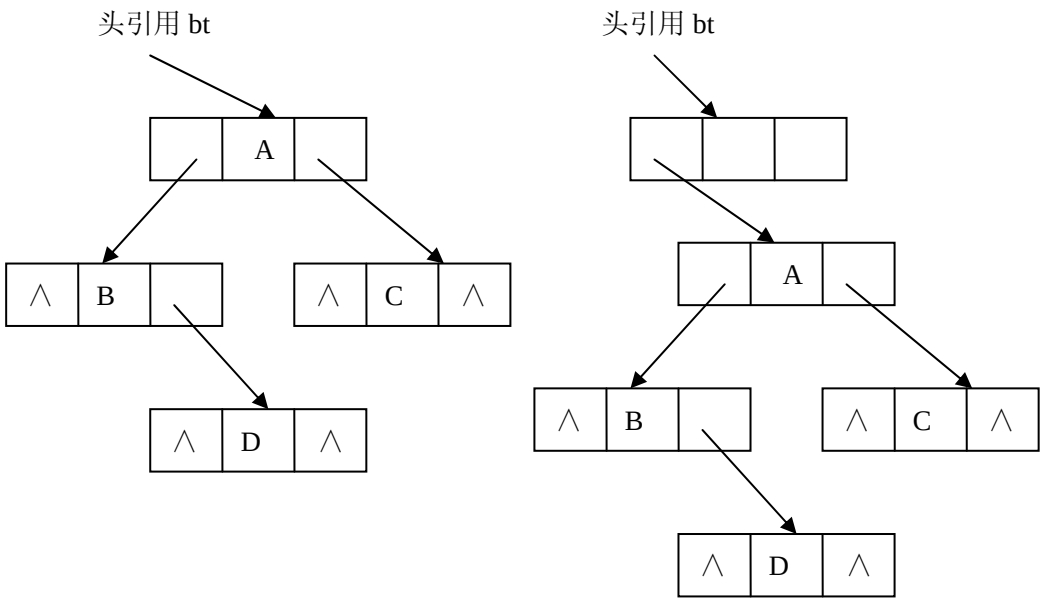
图 5.9 右单支树及其顺序存储示意图

2、二叉树的二叉链表存储结构

二叉树的二叉链表存储结构是指二叉树的结点有三个域：一个数据域和两个引用域，数据域存储数据，两个引用域分别存放其左、右孩子结点的地址。当左孩子或右孩子不存在时，相应域为空，用符号 NULL 或∧表示。结点的存储结构如下所示：

lChild	data	rChild

图 5.10 是图 5.8(a)所示的二叉树的二叉链表示意图。图 5.10(a)是不带头结点的二叉链表，图 5.10(b)是带头结点的二叉链表。



(a) 不带头结点的二叉链表　　(b) 带头结点的二叉链表

图 5.10 二叉树的二叉链表示意图

由图 5.10 所示的二叉树有 4 个结点，每个结点中有两个引用，共有 8 个引用，其中 3 个引用被使用，5 个引用是空的。由性质 4 可知：由 n 个结点构成的二叉链表中，只有 n-1 个引用域被使用，还有 n+1 个引用域是空的。

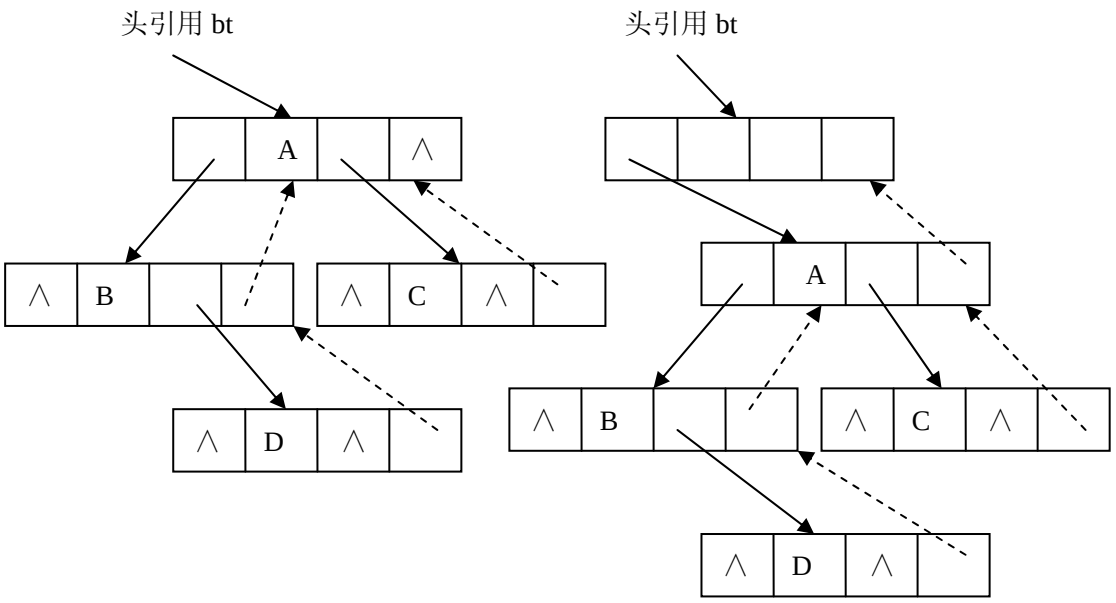
3、二叉树的三叉链表存储结构

使用二叉链表，可以非常方便地访问一个结点的子孙结点，但要访问祖先结点非常困难。可以考虑在每个结点中再增加一个引用域存放其双亲结点的地址信息，这样就可以通过该引用域非常方便地访问其祖先结点。这就是下面要介绍的三叉链表。

二叉树的三叉链表存储结构是指二叉树的结点有四个域：一个数据域和三个引用域，数据域存储数据，三个引用域分别存放其左、右孩子结点和双亲结点的地址。当左、右孩子或双亲结点不存在时，相应域为空，用符号 NULL 或∧表示。结点的存储结构如下所示：

lChild	data	rChild	parent

图 5.11 是图 5.8(a)所示的二叉树的三叉链表示意图。图 5.11(a)是不带头结点的三叉链表，图 5.11(b)是带头结点的三叉链表。



(a) 不带头结点的三叉链表　　(b) 带头结点的三叉链表

图 5.11 二叉树的三叉链表示意图

5.2.4 二叉链表存储结构的类实现

二叉树的二叉链表的结点类有 3 个成员字段：数据域字段 data、左孩子引用域字段 lChild 和右孩子引用域字段 rChild。二叉树的二叉链表的结点类的实现如下所示。

```
public class Node<T>
{
  private T data;                    //数据域
  private Node<T> lChild;            //左孩子
  private Node<T> rChild;            //右孩子

  //构造器
  public Node(T val, Node<T> lp, Node<T> rp)
  {
      data = val;
      lChild = lp;
      lChild = rp;
  }

  //构造器
  public Node(Node<T> lp, Node<T> rp)
  {
      data = default(T);
      lChild = lp;
      rChild = rp;
  }
```

```
//构造器
public Node(T val)
{
    data = val;
    lChild = null;
    rChild = null;
}

//构造器
public Node()
{
    data = default(T);
    lChild = null;
    rChild = null;
}

//数据属性
public T Data
{
    get
    {
        return data;
    }
    set
    {
        value = data;
    }
}

//左孩子属性
public Node<T> LChild
{
    get
    {
        return lChild;
    }
    set
    {
        lChild = value;
    }
}

//右孩子属性
```

```
        public Node<T> RChild
        {
            get
            {
                return rChild;
            }
            set
            {
                rChild = value;
            }
        }
    }
```

不带头结点的二叉树的二叉链表比带头结点的二叉树的二叉链表的区别与不带头结点的单链表与带头结点的单链表的区别一样。下面只介绍不带头结点的二叉树的二叉链表的类 BiTree<T>。BiTree<T>类只有一个成员字段 head 表示头引用。以下是 BiTree<T>类的实现。

```
public class BiTree<T>
{
    private Node<T> head;    //头引用

    //头引用属性
    public Node<T> Head
    {
        get
        {
            return head;
        }
        set
        {
            head = value;
        }
    }

    //构造器
    public BiTree()
    {
        head = null;
    }

    //构造器
    public BiTree(T val)
    {
        Node<T> p = new Node<T>(val);
        head = p;
```

```
}

//构造器
public BiTree(T val, Node<T> lp, Node<T> rp)
{
    Node<T> p = new Node<T>(val,lp,rp);
    head = p;
}

//判断是否是空二叉树
public bool IsEmpty()
{
    if (head == null)
    {
        return true;
    }
    else
    {
        return false;
    }
}

//获取根结点
public Node<T> Root()
{
    return head;
}

//获取结点的左孩子结点
public Node<T> GetLChild(Node<T> p)
{
    return p.LChild;
}

//获取结点的右孩子结点
public Node<T> GetRChild(Node<T> p)
{
    return p.RChild;
}

//将结点p的左子树插入值为val的新结点,
//原来的左子树成为新结点的左子树
public void InsertL(T val, Node<T> p)
{
```

```
    Node<T> tmp = new Node<T>(val);
    tmp.LChild = p.LChild;
    p.LChild = tmp;
}

//将结点p的右子树插入值为val的新结点，
//原来的右子树成为新结点的右子树
public void InsertR(T val, Node<T> p)
{
    Node<T> tmp = new Node<T>(val);
    tmp.RChild = p.RChild;
    p.RChild = tmp;
}

//若p非空，删除p的左子树
public Node<T> DeleteL(Node<T> p)
{
    if ((p == null) || (p.LChild == null))
    {
        return null;
    }

    Node<T> tmp = p.LChild;
    p.LChild = null;

    return tmp;
}

//若p非空，删除p的右子树
public Node<T> DeleteR(Node<T> p)
{
    if ((p == null) || (p.RChild == null))
    {
        return null;
    }

    Node<T> tmp = p.RChild;
    p.RChild = null;

    return tmp;
}

//判断是否是叶子结点
public bool IsLeaf(Node<T> p)
```

```
        {
            if ((p != null) && (p.LChild == null) && (p.RChild == null))
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
```

由于类中基本操作都比较简单，这里不一一详细说明。

5.2.5 二叉树的遍历

二叉树的遍历是指按照某种顺序访问二叉树中的每个结点，使每个结点被访问一次且仅一次。遍历是二叉树中经常要进行的一种操作，因为在实际应用中，常常要求对二叉树中某个或某些特定的结点进行处理，这需要先查找到这个或这些结点。

实际上，遍历是将二叉树中的结点信息由非线性排列变为某种意义上的线性排列。也就是说，遍历操作使非线性结构线性化。

由二叉树的定义可知，一棵二叉树由根结点、左子树和右子树三部分组成，若规定 D、L、R 分别代表遍历根结点、遍历左子树、遍历右子树，则二叉树的遍历方式有 6 种：DLR、DRL、LDR、LRD、RDL、RLD。由于先遍历左子树和先遍历右子树在算法设计上没有本质区别，所以，只讨论三种方式：DLR（先序遍历）、LDR（中序遍历）和 LRD（后序遍历）。

除了这三种遍历方式外，还有一种方式：层序遍历(Level Order)。层序遍历是从根结点开始，按照从上到下、从左到右的顺序依次访问每个结点一次仅一次。

由于树的定义是递归的，所以遍历算法也采用递归实现。下面分别介绍这四种算法，并把它们作为 BiTree<T>类成员方法。

1、先序遍历（DLR）

先序遍历的基本思想是：首先访问根结点，然后先序遍历其左子树，最后先序遍历其右子树。先序遍历的递归算法实现如下，注意：这里的访问根结点是把根结点的值输出到控制台上。当然，也可以对根结点作其它处理。

```
        public void PreOrder(Node<T> root)
        {
            //根结点为空
            if (root == null)
            {
                return;
            }

            //处理根结点
            Console.WriteLine("{0}", root.Data);

            //先序遍历左子树
```

```
        PreOrder(root.LChild);

        //先序遍历右子树
        PreOrder(root.RChild);
    }
```

对于图5.7(b)所示的二叉树，按先序遍历所得到的结点序列为：

A B D H I E J C F G

2、中序遍历（LDR）

中序遍历的基本思想是：首先中序遍历根结点的左子树，然后访问根结点，最后中序遍历其右子树。中序遍历的递归算法实现如下：

```
    public void InOrder(Node<T> root)
    {
        //根结点为空
        if (root == null)
        {
            return;
        }

        //中序遍历左子树
        InOrder(root.LChild);

        //处理根结点
        Console.WriteLine("{0}", root.Data);

        //中序遍历右子树
        InOrder(root.RChild);
    }
```

对于图5.7(b)所示的二叉树，按中序遍历所得到的结点序列为：

H D I B J E A F C G

3、后序遍历（LRD）

后序遍历的基本思想是：首先后序遍历根结点的左子树，然后后序遍历根结点的右子树，最后访问根结点。后序遍历的递归算法实现如下，

```
    public void PostOrder(Node<T> root)
    {
        //根结点为空
        if (root == null)
        {
            return;
        }

        //后序遍历左子树
        PostOrder(root.LChild);

        //后序遍历右子树
```

```
        PostOrder(root.RChild);

        //处理根结点
        Console.WriteLine("{0}", root.Data);
    }
```

对于图5.7(b)所示的二叉树，按后序遍历所得到的结点序列为：

H I D J E B F G C A

4、层序遍历（Level Order）

层序遍历的基本思想是：由于层序遍历结点的顺序是先遇到的结点先访问，与队列操作的顺序相同。所以，在进行层序遍历时，设置一个队列,将根结点引用入队，当队列非空时，循环执行以下三步：

（1） 从队列中取出一个结点引用，并访问该结点；

（2） 若该结点的左子树非空，将该结点的左子树引用入队；

（3） 若该结点的右子树非空，将该结点的右子树引用入队；

层序遍历的算法实现如下：

```
    public void LevelOrder(Node<T> root)
    {
        //根结点为空
        if (root == null)
        {
            return;
        }

        //设置一个队列保存层序遍历的结点
        CSeqQueue<Node<T>> sq = new CSeqQueue<Node<T>>(50);

        //根结点入队
        sq.In(root);

        //队列非空，结点没有处理完
        while (!sq.IsEmpty())
        {
            //结点出队
            Node<T> tmp = sq.Out();

            //处理当前结点
            Console.WriteLine("{o}", tmp);

            //将当前结点的左孩子结点入队
            if (tmp.LChild != null)
            {
                sq.In(tmp.LChild);
            }
```

```
//将当前结点的右孩子结点入队
```

```
            if (tmp.RChild != null)
            {
                sq.In(tmp.RChild);
            }
        }
    }
```

对于图5.7(b)所示的二叉树，按层次遍历所得到的结点序列为：

A B C D E F G H I J

5.3 树与森林

5.3.1 树的存储

在实际应用中，人们采用多种形式的存储结构来表示树，既有顺序存储结构，又有链式存储结构，但无论采用哪种存储结构，都要求存储结构不但能存储结点本身的信息，还能存储树中各结点之间的逻辑关系。下面介绍几种常用的树的存储方式。

1、双亲表示法

从树的定义可知，除根结点外，树中的每个结点都有唯一的一个双亲结点。根据这一特性，可用一组连续的存储空间（一维数组）存储树中的各结点。树中的结点除保存结点本身的信息之外，还要保存其双亲结点在数组中的位置（数组的序号），树的这种表示法称为双亲表示法。

由于树的结点只保存两个信息，所以树的结点用结构体 PNode<T>来表示。结构中有两个字段：数据字段 data 和双亲位置字段 pPos。而树类 PTree<T>只有一个成员数组字段 nodes，用于保存结点。PNode<T>和 PTree<T>都只给出了成员字段，没有给出其它成员，比如属性、构造器、成员方法等等，感兴趣的读者可以自己添加。

树的双亲表示法的结点的结构如下所示：

data	pPos

树的双亲表示法的结点的结构体 PNode<T>和树类 PTree<T>的定义如下：

```
public struct PNode<T>
{
    public T data;
    public int pPos;
    …
}

public class PTree<T>
{
    private PNode<T>[] nodes;
        …
}
```

图 5.1 所示的树的双亲表示法如图 5.12 所示。图中 pPos 域的值为-1 表示该结点是根结点，无双亲结点。

序号	data	pPos
0	A	-1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	1
6	G	1
7	H	2
8	I	3
9	J	3

图 5.12 树的双亲表示法

树的双亲表示法对于实现 Parent(t)操作和 Root()操作非常方便。Parent(t)操作可以在常量时间内实现，反复调用 Parent(t)操作，直到遇到无双亲的结点（其 pPos 值为-1）时，便找到了树的根，这就是 Root()操作的执行过程。但要实现查找孩子结点和兄弟结点等操作非常困难，因为这需要查询整个数组。要实现这些操作，需要在结点结构中增设存放第 1 个孩子在数组中的序号的域和存放第 1 个兄弟在数组中的序号的域。

2、孩子链表表示法

孩子链表表示法也是用一维数组来存储树中各结点的信息。但结点的结构与双亲表示法中结点的结构不同，孩子链表表示法中的结点除保存本身的信息外，不是保存其双亲结点在数组中的序号，而是保存一个链表的第一个结点的地址信息。这个链表是由该结点的所有孩子结点组成。每个孩子结点保存有两个信息，一个是每个孩子结点在一维数组中的序号，另一个是下一个孩子结点的地址信息。

孩子结点的结构如下所示：

index	nextChild

孩子结点类 ChildNode 的定义如下（与前面一样，只列出了成员字段）：

```
public class ChildNode
{
    private int index;
    private ChildNode nextChild;
    …
}
```

树的孩子链表表示法的结点的结构如下所示：

data	firstChild

树的孩子链表表示法[ ]定义如下：

```
public class CLNode<T>
{
    private T data;
```

```
    private ChildNode firstChild;
    …
}
```

树类 CLTree<T>的定义如下：

```
public class CLTree<T>
{
    private Node<T>[] nodes;
        …
}
```

图 5.1 所示的树的孩子链表表示法如图 5.13 所示。

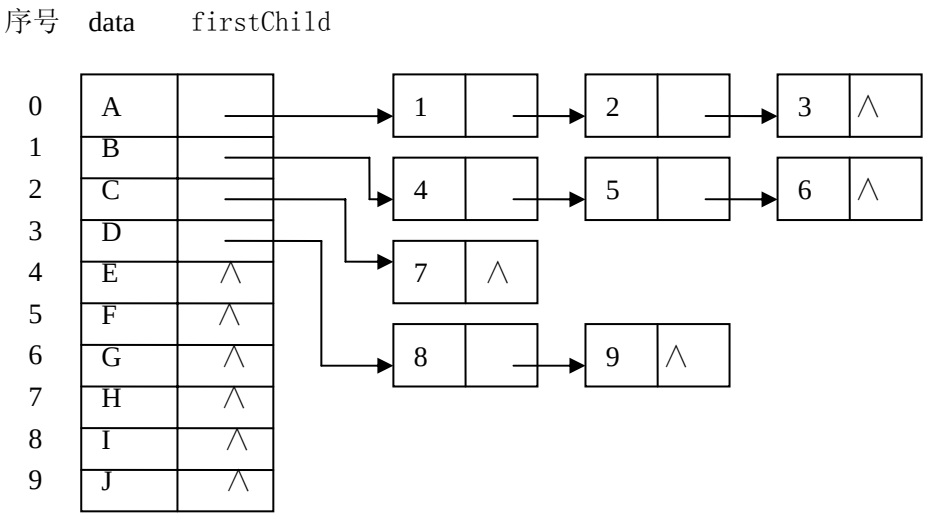


图 5.13 树的的孩子链表表示法

树的孩子链表表示法对于实现查找孩子结点等操作非常方便，但对于实现查找双亲结点、兄弟结点等操作则比较困难。

3、孩子兄弟表示法

这是一种常用的数据结构，又称二叉树表示法，或二叉链表表示法，即以二叉链表作为树的存储结构。每个结点除存储本身的信息外，还有两个引用域分别存储该结点第一个孩子的地址信息和下一个兄弟的地址信息。树类 CSTree<T>只有一个成员字段 head，表示头引用。

树的孩子兄弟表示法的结点的结构如下所示：

树的孩子兄弟表示法的结点类 CSNode<T>的定义如下（与前面一样，只列出了成员字段）：

```
public class CSNode<T>
{
    private T data;
    private CSNode<T> firstChild;
    private CSNode<T> nextSibling;
    …
}
```

树类 CSTree<T>的定义如下（与前面一样，只列出了成员字段）：

```
public class CSTree<T>
{
    private CSNode<T> head;;
        …
}
```

图 5.1 所示的树的孩子兄弟表示法如图 5.14 所示。

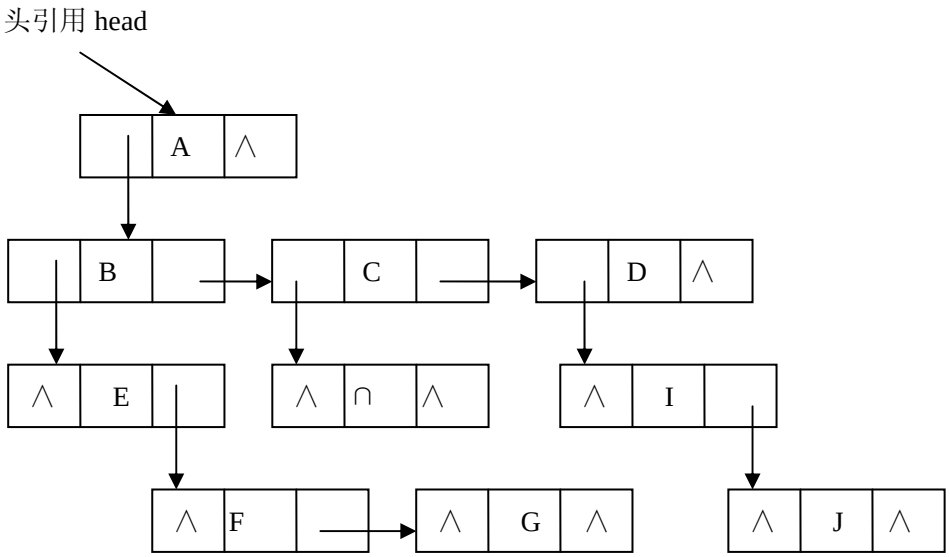


图 5.14 树的孩子兄弟表示法

树的孩子兄弟表示法对于实现查找孩子、兄弟等操作非常方便，但对于实现查找双亲结点等操作则非常困难。如果在树的结点中再增加一个域来存储孩子的双亲结点的地址信息，则就可以较方便地实现上述操作了。

5.3.2 树、森林与二叉树的转换

从树的孩子兄弟表示法可知，树可以用二叉链表进行存储，所以，二叉链表可以作为树和二叉树之间的媒介。也就是说，借助二叉链表，树和二叉树可以相互进行转换。从物理结构来看，它们的二叉链表是相同的，只是解释不同而已。并且，如果设定一定的规则，就可用二叉树来表示森林，森林和二叉树也可以相互进行转换。

1、树转换为二叉树

由于二叉树是有序的，为了避免混淆，对于无序树，我们约定树中的每个结点的孩子结点按从左到右的顺序进行编号。如图 5.1 所示的树，根结点 A 有三个孩子 B、C、D，规定结点 B 是结点 A 的第一个孩子，结点 C 是结点 A 的第 2 个孩子，结点 D 是结点 A 的第 3 个孩子。

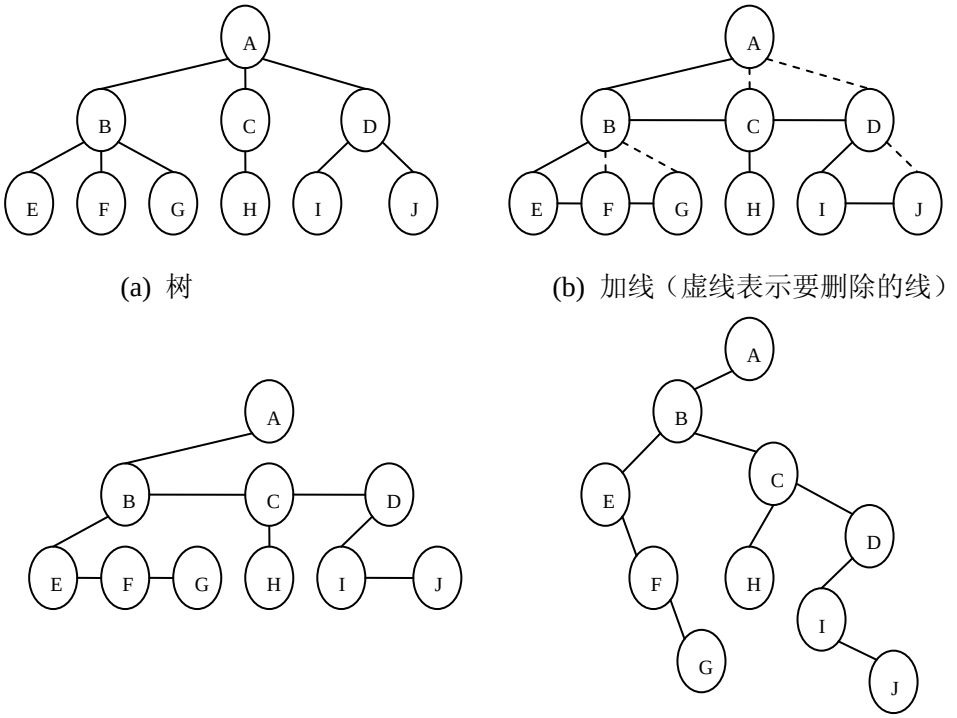
将树转换成二叉树的步骤是：

（1）加线。就是在所有兄弟结点之间加一条连线；

（2）抹线。就是对树中的每个结点，只保留他与第一个孩子结点之间的连线，删除它与其它孩子结点之间的连线；

（3）旋转。就是以树的根结点为轴心，将整棵树顺时针旋转一定角度，使之结构层次分明。

图 5.15 是图 5.1 所示的树转换为二叉树的转换过程示意图。



(a) 树　　(b) 加线（虚线表示要删除的线）

(c) 抹线　　(d) 旋转

图 5.15 树转换为二叉树的过程示意图

2、森林转换为二叉树

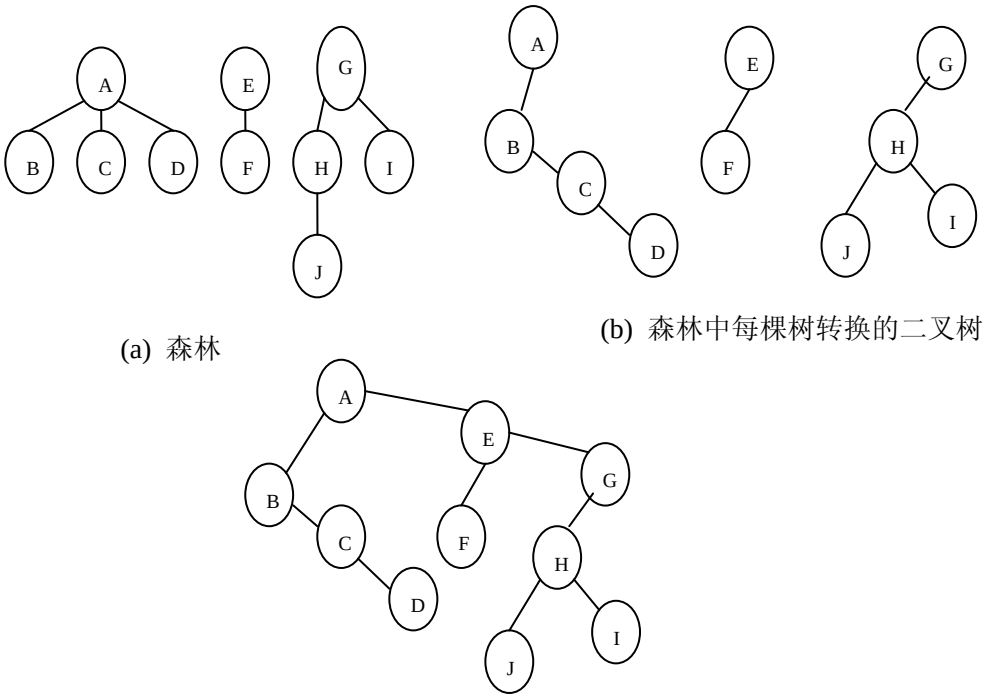
森林是由若干棵树组成，可以将森林中的每棵树的根结点看作是兄弟，由于每棵树都可以转换为二叉树，所以森林也可以转换为二叉树。

将森林转换为二叉树的步骤是：

（1）先把每棵树转换为二叉树；

（2）第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树的根结点的右孩子结点，用线连接起来。当所有的二叉树连接起来后得到的二叉树就是由森林转换得到的二叉树。

图 5.16 是森林转换为二叉树的转换过程示意图。



(a) 森林　　(b) 森林中每棵树转换的二叉树

(c) 森林转化得到的二叉树

图 5.16 森林转换为二叉树的转换过程示意图

3、二叉树转换为树

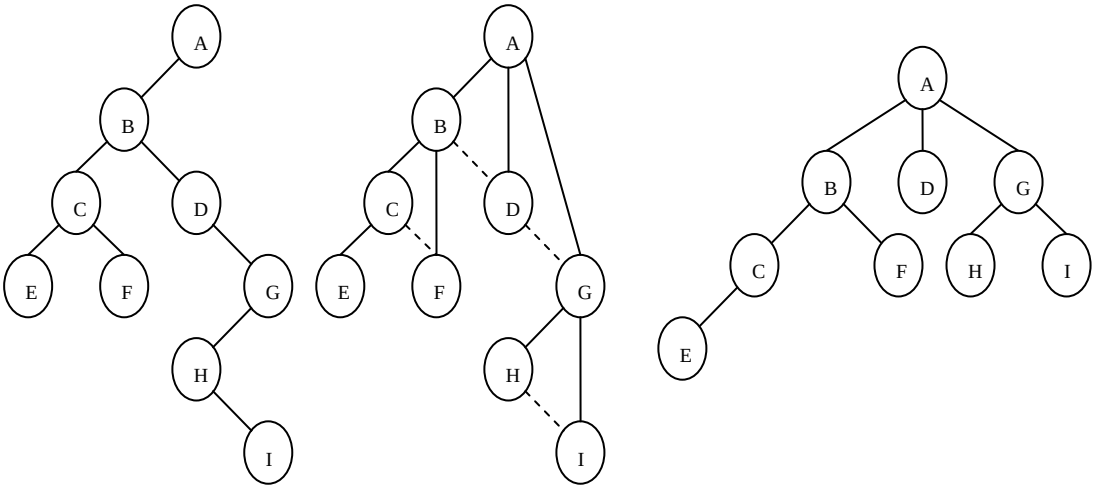
二叉树转换为树是树转换为二叉树的逆过程，其步骤是：

（1）若某结点的左孩子结点存在，将左孩子结点的右孩子结点、右孩子结点的右孩子结点……都作为该结点的孩子结点，将该结点与这些右孩子结点用线连接起来；

（2）删除原二叉树中所有结点与其右孩子结点的连线；

（3）整理（1）和（2）两步得到的树，使之结构层次分明。

图 5.17 是二叉树转换为树的过程示意图。



(a) 二叉树　　(b) 连线（虚线表示要删除的线）　　(c) 二叉树转换后得到的树

图 5.17 二叉树转换为树的过程示意图

4、二叉树转换为森林

二叉树转换为森林比较简单，其步骤如下：

（1）先把每个结点与右孩子结点的连线删除，得到分离的二叉树；

（2）把分离后的每棵二叉树转换为树；

（3）整理第（2）步得到的树，使之规范，这样得到森林。

5.3.3 树和森林的遍历

1、树的遍历

树的遍历通常有两种方式。

（1）先序遍历，即先访问树的根结点，然后依次先序遍历树中的每棵子树。

（2）后序遍历，即先依次后序遍历树中的每棵子树，然后访问根结点。

对图 5.1 所示的树进行先序遍历所得到的结点序列为：

A B E F G C H D I J

对此树进行后序遍历得到的结点序列为：

E F G B H C I J D A

根据树与二叉树的转换关系以及二叉树的遍历定义可以推知，树的先序遍历与其转换的相应的二叉树的先序遍历的结果序列相同；树的后序遍历与其转换的二叉树的中序遍历的结果序列相同；树的层序遍历与其转换的二叉树的后序遍历的结果序列相同。因此，树的遍历算法可以采用相应的二叉树的遍历算法来实现。

另外，在后面讲图的广度优先遍历算法中要提到树的层序遍历算法。树的层序遍历算法与二叉树的层序遍历算法类似，也是从树的根结点开始，按从从上到下从左到右的顺序进行遍历。

2、森林的遍历

森林的遍历有两种方式。

（1）先序遍历，即先访问森林中第一棵树的根结点，然后先序遍历第一棵树中的每棵子树，最后先序遍历除第一棵树之后剩余的子树森林。

（2）中序遍历，即先中序遍历森林中第一棵树的根结点的所有子树，然后访问第一棵树的根结点，最后中序遍历除第一棵树之后剩余的子树森林。

图 5.16（a）所示的森林的先序遍历的结点序列为：

A B C D E F G H J I

此森林的中序遍历的结点序列为：

B C D A F E J H I G

由森林与二叉树的转换关系以及森林与二叉树的遍历定义可知，森林的前序遍历和中序遍历与所转换得到的二叉树的先序遍历和中序遍历的结果序列相同。

5.4 哈夫曼树

5.4.1 哈夫曼树的基本概念

首先给出定义哈夫曼树所要用到的几个基本概念。

（1）路径(Path)：从树中的一个结点到另一个结点之间的分支构成这两个结点间的路径。

（2）路径长度(Path Length)：路径上的分支数。

（3）树的路径长度(Path Length of Tree)：从树的根结点到每个结点的路径长度之和。在结点数目相同的二叉树中，完全二叉树的路径长度最短。

（4）结点的权(Weight of Node)：在一些应用中，赋予树中结点的一个有实际意义的数。

（5）结点的带权路径长度(Weight Path Length of Node)：从该结点到树的根结点的路径长度与该结点的权的乘积。

（6）树的带权路径长度（WPL）：树中所有叶子结点的带权路径长度之和，记为

$$WPL = \sum_{k=1}^{n} W_k \cdot L_k$$

其中，W_k为第k个叶子结点的权值，L_k为第k个叶子结点的路径长度。在图5.17(a)所示的二叉树中，结点B的路径长度为 1，结点C和D的路径长度为 2，结点E、F和G的路径长度为 3，结点H的路径长度为 4，结点I的路径长度为 5。该树的路径长度为：1+2*2+3*3+4+5=23。如果结点B、C、D、E、F、G、H、I的权分别是 1、2、3、4、5、6、7、8，则这些结点的带权路径长度分别是 1*1、2*2、2*3、3*4、3*5、3*6、4*7、5*8，该树的带权路径长度为 3*5+3*6+5*8=73。

那么，什么是哈夫曼树呢？

哈夫曼树(Huffman Tree)，又叫最优二叉树，指的是对于一组具有确定权值的叶子结点的具有最小带权路径长度的二叉树。

在图 5.18 所示的的四棵二叉树，都有 4 个叶子结点，其权值分别为 1、2、3、4，它们的带权路径长度分别为：

(a)WPL=1×2+2×2+3×2+4×2=20

(b)WPL=1×1+2×2+3×3+4×3=28

(c)WPL=1×3+2×3+3×2+4×1=19

(d)WPL=2×1+1×2+3×3+4×3=29

其中，图 5.18(c)所示的二叉树的带权路径长度最小，这棵树就是哈夫曼树。可以验证，哈夫曼树的带权路径长度最小。

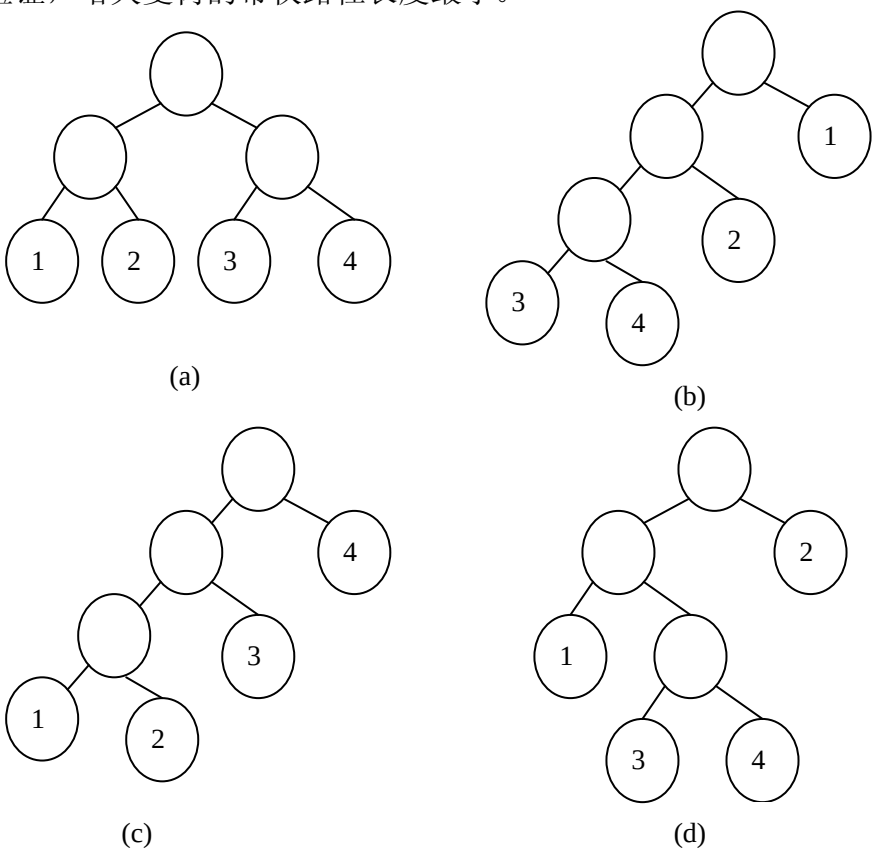


图 5.18 具有不同带权路径长度的二叉树

那么，如何构造一棵哈夫曼树呢？哈夫曼最早给出了一个带有一般规律的算法，俗称哈夫曼算法。现叙述如下：

（1）根据给定的n个权值{$w_1,w_2,\cdots,w_n$}，构造n棵只有根结点的二叉树集合F={$T_1,T_2,\cdots,T_n$}；

（2）从集合 F 中选取两棵根结点的权最小的二叉树作为左右子树，构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树根结点权值之和。

（3）在集合 F 中删除这两棵树，并把新得到的二叉树加入到集合 F 中；

（4）重复上述步骤，直到集合中只有一棵二叉树为止，这棵二叉树就是哈夫曼树。

图 5.19 是给出了前面提到的叶子结点权值集合 W={1,2,3,4}的哈夫曼树的构造过程。

由二叉树的性质 4 和哈夫曼树的特点可知，一棵有 n 个叶子结点构造的哈夫曼树共有 2n-1 个结点。

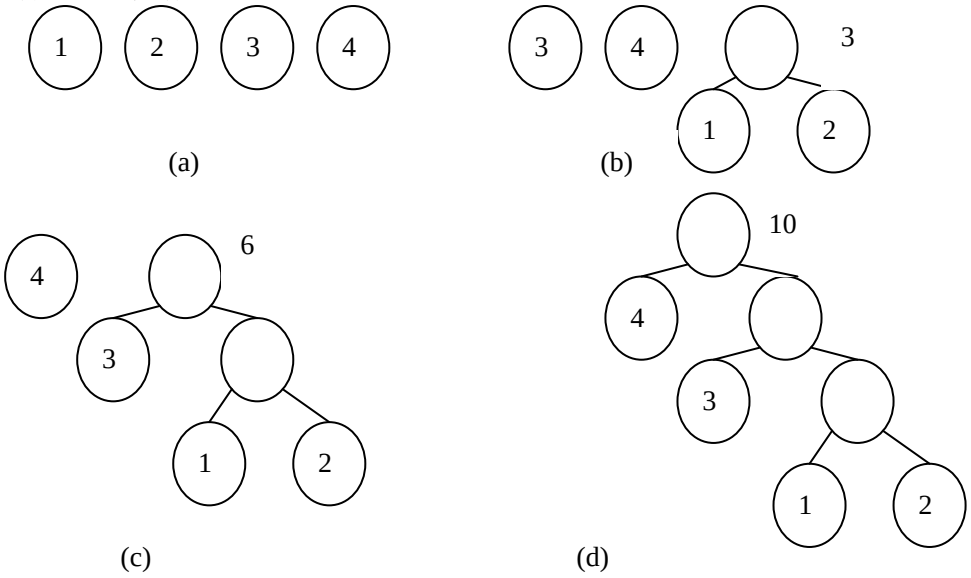


图 5.19 哈夫曼树的构造过程

5.4.2 哈夫曼树类的实现

由哈夫曼树的构造算法可知，用一个数组存放原来的 n 个叶子结点和构造过程中临时生成的结点，数组的大小为 2n-1。所以，哈夫曼树类 HuffmanTree 中有两个成员字段：data 数组用于存放结点，leafNum 表示哈夫曼树叶子结点的数目。结点有四个域，一个域 weight，用于存放该结点的权值；一个域 lChild，用于存放该结点的左孩子结点在数组中的序号；一个域 rChild，用于存放该结点的右孩子结点在数组中的序号；一个域 parent，用于判定该结点是否已加入哈夫曼树中。哈夫曼树结点的结构为。

weight	Child	rChild	parent

所以，结点类 Node 有 4 个成员字段，weight 表示该结点的权值，lChild 和 rChild 分别表示左、右孩子结点在数组中的序号，parent 表示该结点是否已加入哈夫曼树中，如果 parent 的值为-1，表示该结点未加入到哈夫曼树中。当该结点已加入到哈夫曼树中时，parent 的值为其双亲结点在数组中的序号。

结点类 Node 的定义如下：

```
public class Node
{
    private int weight;      //结点权值
    private int lChild;      //左孩子结点
    private int rChild;      //右孩子结点
    private int parent;      //父结点

    //结点权值属性
    public int Weight
    {
        get
        {
            return weight;
        }
        set
        {
            weight = value;
        }
    }

    //左孩子结点属性
    public int LChild
    {
        get
        {
            return lChild;
        }
        set
        {
            lChild = value;
        }
    }

    //右孩子结点属性
    public int RChild
    {
        get
        {
            return rChild;
        }
        set
        {
            rChild = value;
```

```
            }
        }

        //父结点属性
        public int Parent
        {
            get
            {
                return parent;
            }
            set
            {
                parent = value;
            }
        }

        //构造器
        public Node()
        {
            weight = 0;
            lChild = -1;
            rChild = -1;
            parent = -1;
        }

        //构造器
        public Node(int w, int lc, int rc, int p)
        {
            weight = w;
             lChild = lc;
             rChild = rc;
             parent = p;
        }
    }
```

哈夫曼树类 HuffmanTree 中只有一个成员方法 Create，它的功能是输入 n 个叶子结点的权值，创建一棵哈夫曼树。哈夫曼树类 HuffmanTree 的实现如下。

```
    public class HuffmanTree
    {
        private Node[] data;       //结点数组
        private int leafNum;       //叶子结点数目

        //索引器
        public Node this[int index]
        {
```

```
        get
        {
            return data[index];
        }
        set
        {
            data[index] = value;
        }
    }

    //叶子结点数目属性
    public int LeafNum
    {
        get
        {
            return leafNum;
        }
        set
        {
            leafNum = value;
        }
    }

    //构造器
    public HuffmanTree (int n)
    {
        data = new Node[2*n-1];
        leafNum = n;
    }

    //创建哈夫曼树
    public void Create()
    {
        int m1;
        int m2;
        int x1;
        int x2;

        //输入 n 个叶子结点的权值
        for (int i = 0; i < this.leafNum; ++i)
        {
            data[i].Weight = Console.Read();
        }
```

```
            //处理 n 个叶子结点，建立哈夫曼树
            for (int i = 0; i < this.leafNum - 1; ++i)
            {
                max1 = max2 = Int32.MaxValue;
                tmp1 = tmp2 = 0;

                //在全部结点中找权值最小的两个结点
                for (int j = 0; j < this.leafNum + i; ++j)
                {
                    if ((data[i].Weight < max1)
                         && (data[i].Parent == -1))
                    {
                        max2 = max1;
                        tmp2 = tmp1;
                        tmp1 = j;
                        max1 = data[j].Weight;
                    }
                    else if ((data[i].Weight < max2)
                              && (data[i].Parent == -1))
                    {
                        max2 = data[j].Weight;
                        tmp2 = j;
                    }
                }

                data[tmp1].Parent = this.leafNum + i;
                data[this.leafNum + i].Weight = data[tmp1].Weight
                                              + data[tmp2].Weight;
                data[this.leafNum + i].LChild = tmp1;
                data[this.leafNum + i].RChild = tmp2;
            }
        }
    }
```

5.4.3 哈夫曼编码

在数据通信中，通常需要把要传送的文字转换为由二进制字符 0 和 1 组成的二进制串，这个过程被称之为编码(Encoding)。例如，假设要传送的电文为 DCBBADD，电文中只有 A、B、C、D 四种字符，若这四个字符采用表 5-1(a) 所示的编码方案，则电文的代码为 11100101001111，代码总长度为 14。若采用表 5-1(b) 所示的编码方案，则电文的代码为 0110101011100，代码总长度为 13。

表 5-1 字符集的不同编码方案

字符	码
A	00
B	01
C	10
D	11

(a)

字符	码
A	111
B	10
C	110
D	0

(b)

哈夫曼树可用于构造总长度最短的编码方案。具体构造方法如下：

设需要编码的字符集为{$d_1,d_2,\cdots,d_n$}，各个字符在电文中出现的次数或频率集合为{$w_1,w_2,\cdots,w_n$}。以$d_1,d_2,\cdots,d_n$作为叶子结点，以$w_1,w_2,\cdots,w_n$作为相应叶子结点的权值来构造一棵哈夫曼树。规定哈夫曼树中的左分支代表 0，右分支代表 1，则从根结点到叶子结点所经过的路径分支组成的 0 和 1 的序列便为该结点对应字符的编码就是哈夫曼编码(Huffman Encoding)。

图 5-20 就是电文 DCBBADD 的哈夫曼树，其编码就是表 5-1(b)。

在建立不等长编码中，必须使任何一个字符的编码都不是另一个编码的前缀，这样才能保证译码的唯一性。例如，若字符 A 的编码是 00，字符 B 的编码是 001，那么字符 A 的编码就成了字符 B 的编码的后缀。这样，对于代码串 001001，在译码时就无法判定是将前两位码 00 译成字符 A 还是将前三位码 001 译成 B。这样的编码被称之为具有二义性的编码，二义性编码是不唯一的。而在哈夫曼树中，每个字符结点都是叶子结点，它们不可能在根结点到其它字符结点的路径上，所以一个字符的哈夫曼编码不可能是另一个字符的哈夫曼编码的前缀，从而保证了译码的非二义性。

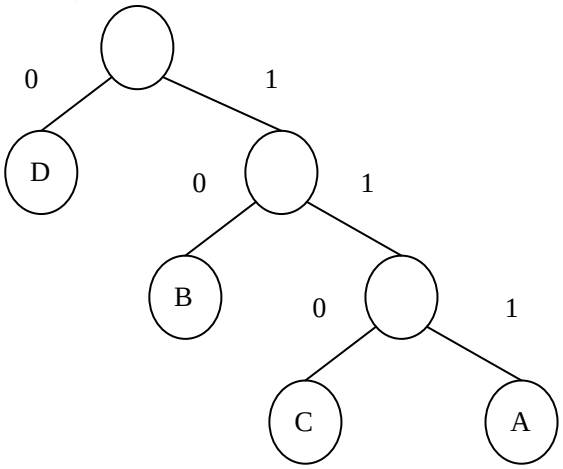


图 5-20 哈夫曼编码

5.5 应用举例

【例 5-1】编写算法，在二叉树中查找值为 value 的结点。

算法思路：在二叉树中查找具有某个特定值的结点就是遍历二叉树，对于遍历到的结点，判断其值是否等于 value，如果是则返回该结点，否则返回空。本节例题的算法都作为 BiTree<T>的成员方法。

算法实现如下：

```
Node<T> Search(Node<T> root, T value)
{
    Node<T> p = root;

    if (p == null)
    {
        return null;
    }

    if (!p.Data.Equals(value))
    {
        return p;
    }

    if (p.LChild != null)
    {
        return Search(p.LChild, value);
    }

    if (p.RChild != null)
    {
        return Search(p.RChild, value);
    }

    return null;
}
```

【例 5-2】统计出二叉树中叶子结点的数目。

算法思路：用递归实现该算法。如果二叉树为空，则返回 0；如果二叉树只有一个结点，则根结点就是叶子结点，返回 1，否则返回根结点的左分支的叶子结点数目和右分支的叶子结点数目的和。

算法实现如下：

```
int CountLeafNode(Node<T> root)
{
    if (root == null)
    {
         return 0;
    }
    else if (root.LChild == null && root.RChild == null)
    {
         return 1;
    }
    else
    {
```

```
            return (CountLeafNode(root.LChild) +
                    CountLeafNode(root.RChild));
        }
    }
```

【例 5-3】编写算法，求二叉树的深度。

算法思路：用递归实现该算法。如果二叉树为空，则返回 0；如果二叉树只有一个结点（根结点），返回 1，否则返回根结点的左分支的深度与右分支的深度中较大者加 1。

算法实现如下：

```
int GetHeight(Node<T> root)
{
    int lh;
    int rh;

    if (root == null)
    {
        return 0;
    }
    else if (root.LChild == null && root.RChild == null)
    {
        return 1;
    }
    else
    {
        lh = GetHeight(root.LChild);
        rh = GetHeight(root.RChild);
        return (lh>rh?lh:rh) + 1;
    }
}
```

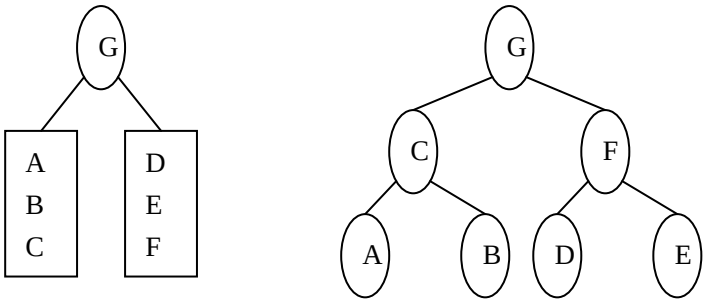
【例 5-4】已知结点的后序序列和中序序列如下：

后序序列：A B C D E F G

中序序列：A C B G D F E

请构造该二叉树。

构造过程如图 5.21 所示。首先由结点的后序序列知该二叉树的根结点为 G，再由结点的中序序列可知左子树的中序序列为（A C B），右子树的中序序列为（D F E）。反过来再由结点的后序序列可知左子树的后序序列为（A B C），右子树的后序序列为（D E F）。类似地，可由左子树的后序序列和中序序列构造左子树，由右子树的后序序列和中序序列构造右子树。



(a) (b)

图 5.21 由后序序列和中序序列构造一棵二叉树的过程

5.6 C#中的树

C#中的树很多。比如，Windows Form 程序设计和 Web 程序设计中都有一种被称为 TreeView 的控件。TreeView 控件是一个显示树形结构的控件，此树形结构与 Windows 资源管理器中的树形结构非常类似。不同的是，TreeView 可以由任意多个节点对象组成。每个节点对象都可以关联文本和图像。另外，Web 程序设计中的 TreeView 的节点还可以显示为超链接并与某个 URL 相关联。每个节点还可以包括任意多个子节点对象。包含节点及其子节点的层次结构构成了 TreeView 控件所呈现的树形结构。

DOM(Document Object Model)是 C#中树形结构的另一个例子。文档对象模型 DOM 不是 C#中独有的，它是 W3C 提供的能够让程序和脚本动态访问和更新文档内容、结构和样式的语言平台。DOM 被分为不同的部分（Core DOM，XML DOM 和 HTML DOM）和不同的版本（DOM 1/2/3），Core DOM 定义了任意结构文档的标准对象集合，XML DOM 定义了针对 XML 文档的标准对象集合，而 HTML DOM 定义了针对 HTML 文档的标准对象集合。C#提供了一个标准的接口来访问并操作 HTML 和 XML 对象集。后面将以 XML 对象集为例进行说明，对 HTML 对象集的操作类似。

DOM 允许将 XML 文档的结构加载到内存中，由此可以获得在 XML 文档中执行更新、插入和删除操作的能力。DOM 是一个树形结构,文件中的每一项都是树中的一个结点。每个结点下面还有子结点。还可以用结点表示数据，并且数据和元素是不同的。在 C#中使用很多类来访问 DOM，主要的类见表 5-2 所示。

表 5-2 C#中访问 DOM 的类

类	详细说明
XmlNode	用于创建对象，这个对象可以保持 XML 文档的一个结点
XmlDocument	用于保持一个完整的 XML 文档对象。允许文档导航和编辑
XmlDocumentFragment	用于保持一个 XML 片段。可以将这个 XML 片段插入到文档中，或用作其它方面
XmlElement	用于在 XML 文档中操作元素类型的结点集合
XmlNodeList	代表 XML 文档中一个有序的结点集合
XmlNamedNodeMap	用于通过索引或索引值访问一个结点集合
XmlAttribute	用于在 XML 文档中操作属性类型结点
XmlCDataSection	用于操作 CDATA 部分
XmlText	用于保持一个元素或属性的文本内容
XmlComment	用于操作注释
XmlDocumentType	用于保持与文档类型声明相关的信息
XmlEntity	用于保持一个实体
XmlEntityReference	用于保持一个实体引用
XmlNotation	用于保持在一个文档类型声明（DTD）中声明的注释
XmlProcessingInstruction	用于保持一个处理指令

C#中的树还有很多，限于篇幅，这里就不一一列举了，感兴趣的读者可以参考有关的书籍和文献资料。

本章小结

树形结构是一种非常重要的非线性结构，树形结构中的数据元素称为结点，它们之间是一对多的关系，既有层次关系，又有分支关系。树形结构有树和二叉树两种。

树是递归定义的，树由一个根结点和若干棵互不相交的子树构成，每棵子树的结构与树相同，通常树指无序树。树的逻辑表示通常有四种方法，即直观表示法、凹入表示法、广义表表示法和嵌套表示法。树的存储方式有 3 种，即双亲表示法、孩子链表表示法和孩子兄弟表示法。

二叉树的定义也是递归的，二叉树由一个根结点和两棵互不相交的子树构成，每棵子树的结构与二叉树相同，通常二叉树指有序树。重要的二叉树有满二叉树和完全二叉树。二叉树的性质主要有 5 条。二叉树的的存储结构主要有三种：顺序存储结构、二叉链表存储结构和三叉链表存储结构，本书给出了二叉链表存储结构的 C#实现。二叉树的遍历方式通常有四种：先序遍历（DLR）、中序遍历（LDR）、后序遍历（LRD）和层序遍历(Level Order)。

森林是 m(m≥0)棵树的集合。树、森林与二叉树的之间可以进行相互转换。树的遍历方式有先序遍历和后序遍历两种，森林的遍历方式有先序遍历和中序遍历两种。

哈夫曼树是一组具有确定权值的叶子结点的具有最小带权路径长度的二叉树。哈夫曼树可用于解决最优化问题，在数据通信等领域应用很广。

习题五

5.1 如图 5.22，试回答下列问题：

（1）树的根结点是哪个结点？哪些是终端结点？哪些是非终端结点？

（2）各结点的度分别是多少？树的度是多少？

（3）各结点的层次分别是多少？树的深度是多少？以 B 为根的子树深度是多少？

（4）结点 F 的双亲是哪个结点？祖先是哪个结点？孩子是哪些结点？兄弟又是哪些结点？

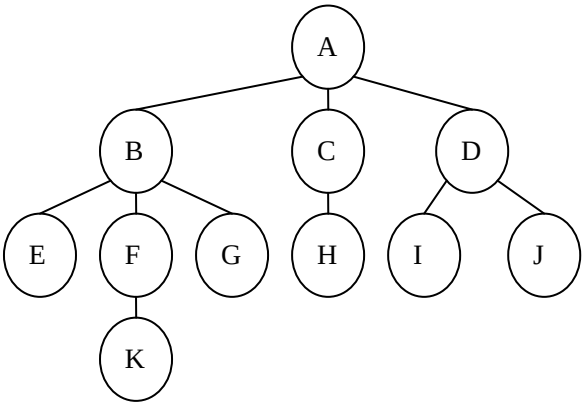


图 5.22

5.2 树和二叉树的区别是什么？

5.3 分别画出图 5.23 所示二叉树的二叉链表、三叉链表和顺序存储结构示意图。

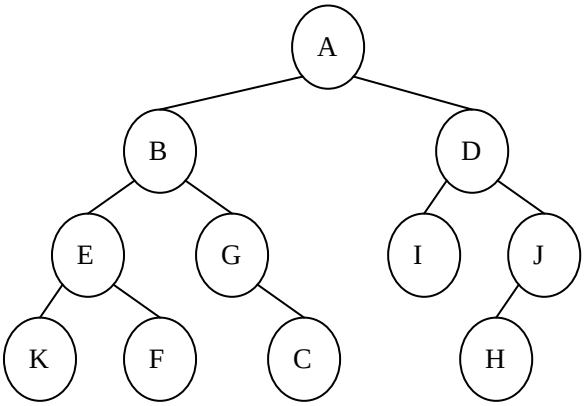


图 5.23

5.4 分别画出图 5.26 所示二叉树的先序遍历、中序遍历和后序遍历的结点访问序列。

5.5 试找出满足下列条件的所有二叉树。

（1）先序序列和中序序列相同；

（2）后序序列和中序序列相同；

（3）先序序列和后序序列相同。

5.6 已知一棵二叉树的先序序列和中序序列分别为 ABCDEFG 和 CBEDAFG，试画出这棵二叉树。

5.7 高度为 h 的完全二叉树中，最多有多少个结点？最少有多少个结点？

5.8 设二叉树中所有分支结点均有非空左右子二叉树，并且叶子结点数目为 n，

二叉树共有多少个结点？

5.9 编写算法，求二叉树中分支结点的数目。

5.10 编写算法，将二叉树中所有结点的左、右子树相互交换。

5.11 编写算法，判断给定的二叉树是否为完全二叉树。

5.12 将图 5.24 中所示的森林转换为二叉树。

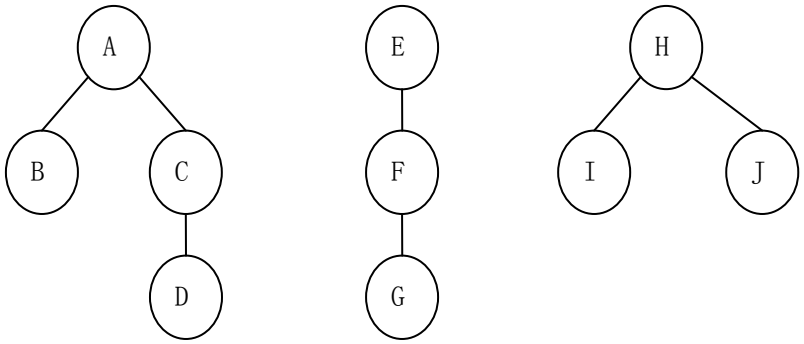


图 5.24

5.13 将图 5.25 中所示的二叉树转换为森林。

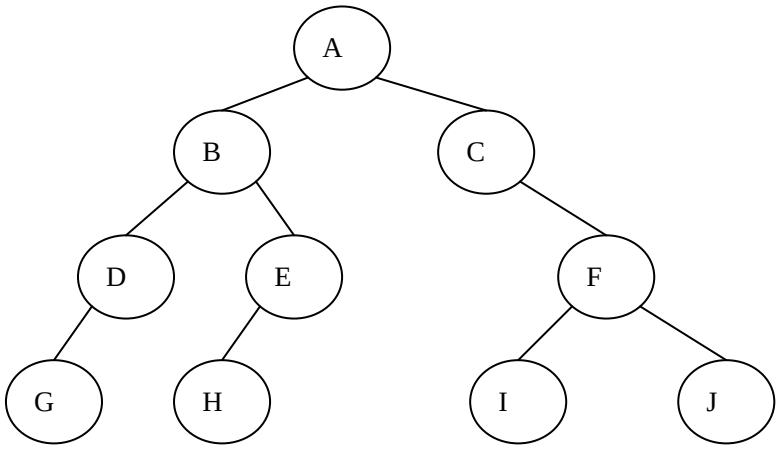


图 5.25

5.14 假设用于通讯的电文由 8 个字母{A，B，C，D，E，F，G，I}组成，字母在电文中出现的次数分别为 32，12，7，18，3，5，26，8，构造相应的哈夫曼编码。

第6章 图

图状结构简称图，是另一种非线性结构，它比树形结构更复杂。树形结构中的结点是一对多的关系，结点间具有明显的层次和分支关系。每一层的结点可以和下一层的多个结点相关，但只能和上一层的一个结点相关。而图中的顶点（把图中的数据元素称为顶点）是多对多的关系，即顶点间的关系是任意的，图中任意两个顶点之间都可能相关。也就是说，图的顶点之间无明显的层次关系，这种关系在现实世界中大量存在。因此，图的应用相当广泛，在自然科学、社会科学和人文科学等许多领域都有着非常广泛的应用。

6.1 图的基本概念

6.1.1 图的定义

图(Graph)是由非空的顶点（Vertex）集合和描述顶点之间的关系——边（Edge）或弧（Arc）的集合组成。其形式化定义为：

G=（V，E）

V={v_i|v_i∈某个数据元素集合}

E={(v_i,v_j)|v_i,v_j∈V∧P(v_i,v_j)}或E={<v_i,v_j>|v_i,v_j∈V∧P(v_i,v_j)}

其中，G 表示图，V 是顶点的集合，E 是边或弧的集合。在集合 E 中，P(vi,vj)表示顶点 vi 和顶点 vj 之间有边或弧相连。图 6.1 给出了图的示例。

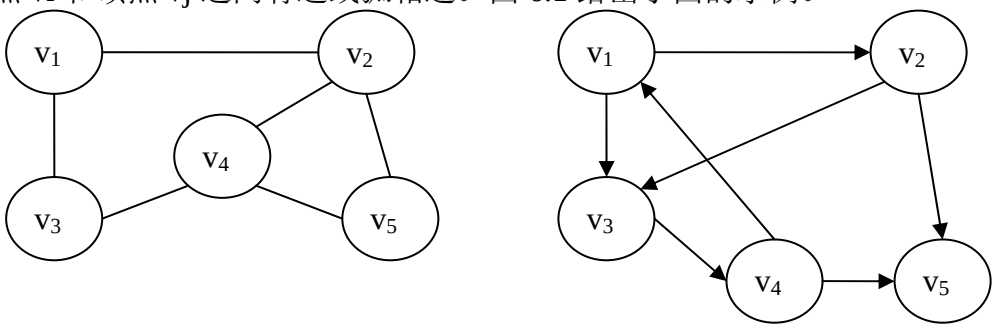


图 6.1 图的示例

在图 6.1(a)中，V={v_1,v_2,v_3,v_4,v_5}

E={(v_1,v_2),(v_1,v_3),(v_2,v_4),(v_2,v_5),(v_3,v_4),(v_4,v_5)}

在图 6.1(b)中，V={v_1,v_2,v_3,v_4,v_5}

E={<v_1,v_2>,<v_1,v_3>,<v_2,v_3>,<v_2,v_5)><v_3,v_4>,<v_4,v_1>,<v_4,v_5>}。

6.1.2 图的基本术语

图的基本术语有以下 14 种：

1、无向图：在一个图中，如果任意两个顶点v_i和v_j构成的偶对(v_i,v_j)∈E是无序的，即顶点之间的连线没有方向，则该图称为无向图(Undirected Graph)。图 6.1(a)

是一个无向图。

2、有向图：在一个图中，如果任意两个顶点v_i和v_j构成的偶对$<v_i,v_j>\in E$是有序的，即顶点之间的连线有方向，则该图称为有向图(Directed Graph)。图 6.1(b)是一个有向图。

3、边、弧、弧头、弧尾：无向图中两个顶点之间的连线称为边(Edge)，边用顶点的无序偶对(v_i,v_j)表示，称顶点v_i和顶点v_j互为邻接点(Adjacency Point)，(v_i,v_j)边依附与顶点v_i和顶点v_j。有向图中两个顶点之间的连线称为弧(Arc)，弧用顶点的有序偶对$<v_i,v_j>$表示，有序偶对的第一个结点v_i称为始点（或弧尾），在图中不带箭头的一端；有序偶对的第二个结点v_j称为终点（或弧头），在图中带箭头的一端。

4、无向完全图：在一个无向图中，如果任意两个顶点之间都有边相连，则称该图为无向完全图(Undirected Complete Graph)。可以证明，在一个含有 n 个顶点的无向完全图中，有 n(n-1)/2 条边。

5、有向完全图：在一个有向图中，如果任意两个顶点之间都有弧相连，则称该图为有向完全图(Directed Complete Graph)。可以证明，在一个含有 n 个顶点的有向完全图中，有 n(n-1)条弧。

6、顶点的度、入度、出度：在无向图中，顶点 v 的度（Degree）是指依附于顶点 v 的边数，通常记为 TD(v)。在有向图中，顶点的度等于顶点的入度(In Degree)与顶点的出度之和。顶点 v 的入度是指以该顶点 v 为弧头的弧的数目，记为 ID(v)；顶点 v 的出度(Out Degree)是指以该顶点 v 为弧尾的弧的数目，记为 OD(v)。所以，顶点 v 的度 TD(v)= ID(v)+ OD(v)。

例如，在无向图 6.1(a)中有：

$TD(v_1)=2$ $TD(v_2)=3$ $TD(v_3)=2$ $TD(v_4)=3$ $TD(v_5)=2$

在有向图 6.1(b)中有：

$ID(v_1)=2$ $OD(v_1)=1$ $TD(v_1)=3$

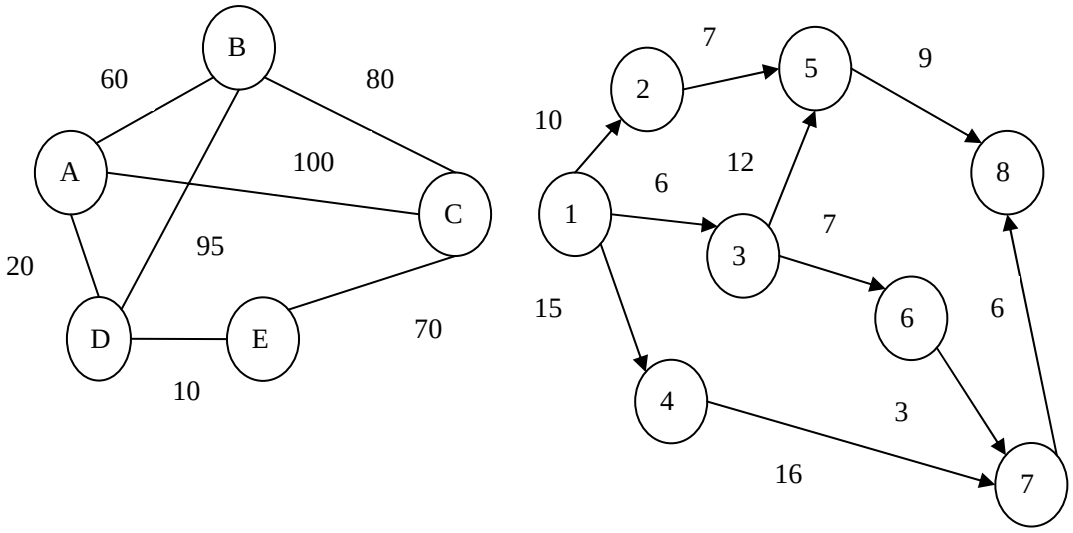
$ID(v_2)=1$ $OD(v_2)=2$ $TD(v_2)=3$

$ID(v_3)=2$ $OD(v_3)=1$ $TD(v_3)=3$

$ID(v_4)=1$ $OD(v_4)=2$ $TD(v_4)=3$

$ID(v_5)=2$ $OD(v_5)=0$ $TD(v_5)=2$

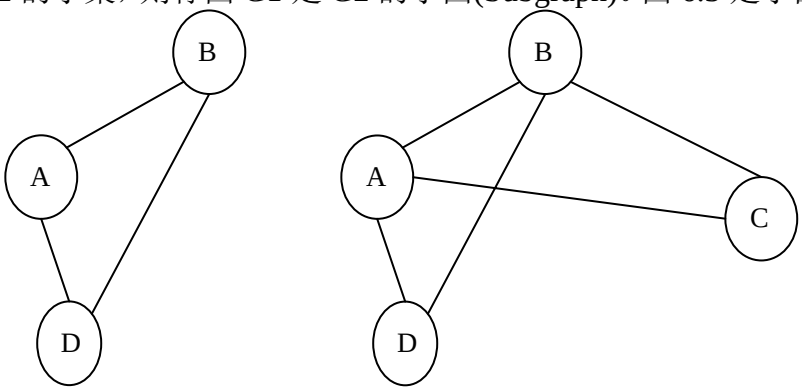
7、权、网：有些图的边（或弧）附带有一些数据信息，这些数据信息称为边（或弧）的权（Weight）。在实际问题中，权可以表示某种含义。比如，在一个地方的交通图中，边上的权值表示该条线路的长度或等级。在一个工程进度图中，弧上的权值可以表示从前一个工程到后一个工程所需要的时间或其它代价等。边（或弧）上带权的图称为网或网络（Network）。图 6.2 是带权图的示例图。



(a) 交通网络图 (b) 施工进度图

图 6.2 带权图

8、子图：设有两个图 G1=（V1，E1），G2=（V2，E2），如果 V1 是 V2 子集，E1 也是 E2 的子集，则称图 G1 是 G2 的子图(Subgraph)。图 6.3 是子图的示例图。



(a) G1 (b) G2

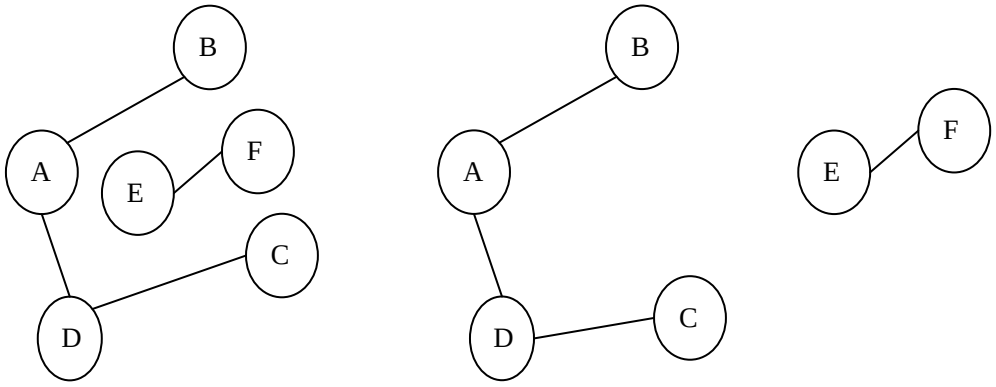
图 6.3 子图示例图

9、路径、路径长度：在无向图G中，若存在一个顶点序列V_p，V_{i1}，V_{i2}，…，V_{im}，V_q，使得（V_p，V_{i1}），（V_{i1}，V_{i2}），…，（V_{im}，V_q）均属于E（G）。则称顶点V_p到V_q存在一条路径(Path)。若G为有向图，则路径也是有向的。它由E（G）中的弧<V_p，V_{i1}>，<V_{i1}，V_{i2}>，…，<V_{im}，V_q>组成。路径长度(Path Length)定义为路径上边或弧的数目。在图 6.2(a)中，从顶点A到顶点B存在 4 条路径，长度分别为 1、2、2、4。在图 6.2(b)中，从顶点 1 到顶点 7 存在 2 条路径，长度分别为 2 和 3。

10、简单路径、回路、简单回路：若一条路径上顶点不重复出现，则称此路径为简单路径(Simple Path)。第一个顶点和最后一个顶点相同的路径称为回路(Cycle)或环。除第一个顶点和最后一个顶点相同其余顶点都不重复的回路称为简单回路(Simple Cycle)，或者简单环。

11、连通、连通图、连通分量：在无向图中，若两个顶点之间有路径，则称这两个顶点是连通的(Connect)。如果无向图 G 中任意两个顶点之间都是连通的，则称图 G 是连通图(Connected Graph)。连通分量(Connected Compenent)是无向图 G

的极大连通子图。极大连通子图是一个图的连通子图，该子图不是该图的其它连通子图的子图。图 6.3 是连通图，图的连通分量的示例见图 6.4 所示。图 6.4(a)中的图 G 有两个连通分量。



(a) 无向图 G　　(b) 图 G 的两个连通分量

图 6.4 连通图和图的连通分量

12、强连通图、强连通分量：在有向图中，若图中任意两个顶点之间都存在从一个顶点到另一个顶点的路径，则称该有向图是强连通图(Strongly Connected Graph)。有向图的极大强连通子图称为强连通分量(Strongly Connected Component)。极大强连通子图是一个有向图的强连通子图，该子图不是该图的其它强连通子图的子图。图 6.5 是强连通图，图 6.6 是强连通分量的示例图。图 6.6(a)中的有向图 G 有两个强连通分量。

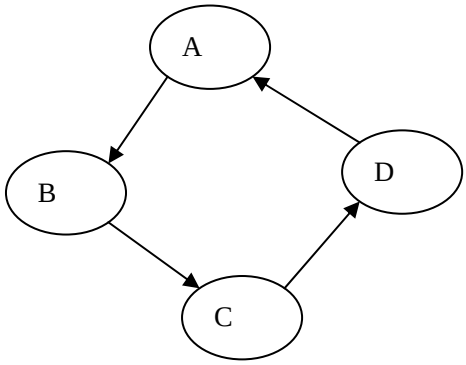
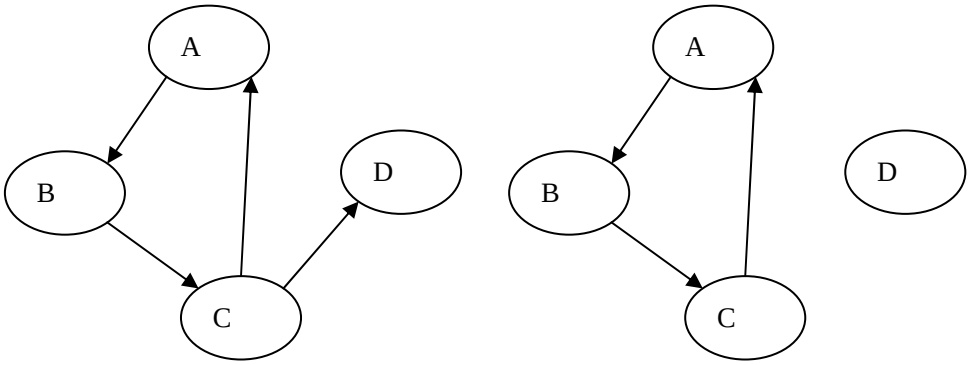


图 6.5 强连通图



(a) 有向图 G　　(b) 图 G 的两个强连通分量

图 6.6 强连通分量示例图

13、生成树：所谓连通图 G 的生成树(Spanning Tree)是指 G 的包含其全部顶点的一个极小连通子图。所谓极小连通子图是指在包含所有顶点并且保证连通的前提下包含原图中最少的边。一棵具有 n 个顶点的连通图 G 的生成树有且仅有 n-1 条边，如果少一条边就不是连通图，如果多一条边就一定有环。但是，有 n-1 条边的图不一定是生成树。图 6.7 就是图 6.3(a)的一棵生成树。

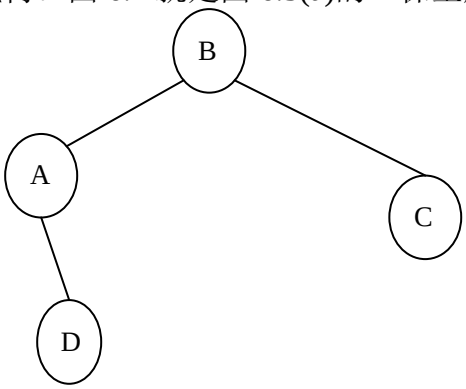


图 6.7 图 6.3(a)的一棵生成树

14、生成森林：在非连通图中，由每个连通分量都可得到一个极小连通子图，即一棵生成树。这些连通分量的生成树就组成了一个非连通图的生成森林(Spanning Forest)。

6.1.3 图的基本操作

图的基本操作用一个接口来表示，为表示图的基本操作，同时给出了顶点类的实现。由于顶点只保存自身信息，所以顶点类 Node<T>很简单，里面只有一个字段 data。

顶点的类 Node<T>的实现如下所示。

```
public Class Node<T>
{
    private T data;    //数据域

    //构造器
    public Node(T v)
    {
        data = v;
    }

    //数据域属性
    public T Data
    {
        get
        {
            return data;
        }
        set
        {
            data = value;
```

```
            }
        }
    }
```

图的接口IGraph<T>的定义如下所示。

```
public interface IGraph<T>
{
    //获取顶点数
    int GetNumOfVertex();

    //获取边或弧的数目
    int GetNumOfEdge();

    //在两个顶点之间添加权为v的边或弧
    void SetEdge(Node<T> v1, Node<T> v2, int v);

    //删除两个顶点之间的边或弧
    void DelEdge(Node<T> v1, Node<T> v2);

    //判断两个顶点之间是否有边或弧
    bool IsEdge(Node<T> v1, Node<T> v2);
}
```

下面对图的基本操作进行说明：

1、GetNumOfVertex()
初始条件：图存在；
操作结果：返回图中的顶点数。

2、GetNumOfEdge()
初始条件：图存在；
操作结果：返回图中的边或弧的数目。

3、SetEdge(Node<T> v1, Node<T> v2, int v)
初始条件：图存在，顶点 v1 和 v2 是图的两个顶点；
操作结果：在顶点 v1 和 v2 之间添加一条边或弧并设边或弧的值为 v。

4、DelEdge(Node<T> v1, Node<T> v2)
初始条件：图存在，顶点 v1 和 v2 是图的两个顶点并且 v1 和 v2 之间有一条边或弧；
操作结果：删除顶点 v1 和 v2 之间的边或弧。

5、IsEdge(Node<T> v1, Node<T> v2)
初始条件：图存在，顶点 v1 和 v2 是图的两个顶点；
操作结果：如果 v1 和 v2 之间有一条边或弧，返回 true,否则返回 false。

6.2 图的存储结构

图是一种复杂的数据结构，顶点之间是多对多的关系，即任意两个顶点之间都可能存在联系。所以，无法以顶点在存储区的位置关系来表示顶点之间的联系，即顺序存储结构不能完全存储图的信息，但可以用数组来存储图的顶点信息。要存储顶点之间的联系必须用链式存储结构或者二维数组。图的存储结构有多种，这里只介绍两种基本的存储结构：邻接矩阵和邻接表。

6.2.1 邻接矩阵

邻接矩阵（Adjacency Matrix）是用两个数组来表示图，一个数组是一维数组，存储图中顶点的信息，一个数组是二维数组，即矩阵，存储顶点之间相邻的信息，也就是边（或弧）的信息，这是邻接矩阵名称的由来。

假设图G=（V，E）中有n个顶点，即V={v_0,v_1,…,v_{n-1}}，用矩阵A[i][j]表示边（或弧）的信息。矩阵A[i][j]是一个n×n的矩阵，矩阵的元素为：

$$A[i][j]=\begin{cases}1 \text{ 若（vi,vj)或<vi,vj>是E(G)中的边或弧}\\0 \text{ 若（vi,vj)或<vi,vj>不是E(G)中的边或弧}\end{cases}$$

若 G 是网，则邻接矩阵可定义为：

$$A[i][j]=\begin{cases}wij \text{ 若（vi, vj)或 < vi, vj > 是E(G)中的边或弧}\\0\text{或}\infty \text{ 若（vi, vj)或 < vi, vj > 不是E(G)中的边或弧}\end{cases}$$

其中，wij 表示边(vi,vj)或弧<vi,vj>上的权值；∞表示一个计算机允许的大于所有边上权值的数。

图6.1(a)、图6.2(a)、图6.2(b)的图的邻接矩阵如图6.8(a)、6.8(b)、6.8(c)所示。

$$A=\begin{pmatrix}0&1&1&0&0\\1&0&0&1&1\\1&0&0&1&0\\0&1&1&0&1\\0&1&0&1&0\end{pmatrix} \quad (a)$$

$$A=\begin{pmatrix}\infty&60&100&20&\infty\\60&\infty&80&95&\infty\\100&80&\infty&\infty&70\\20&95&\infty&\infty&10\\\infty&\infty&70&10&\infty\end{pmatrix} \quad (b)$$

$$A=\begin{pmatrix} \infty & 10 & 6 & 15 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 7 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 12 & 7 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 16 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 9 \\ \infty & \infty & \infty & \infty & \infty & \infty & 3 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 6 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix} \quad (c)$$

图 6.8 图 6.1(a)、图 6.2(a)、图 6.2(b)的图的邻接矩阵

从图的邻接矩阵表示法可以看出这种表示法的特点是：

（1）无向图或无向网的邻接矩阵一定是一个对称矩阵。因此，在具体存放邻接矩阵时只需存放上（或下）三角矩阵的元素即可。

（2）可以很方便地查找图中任一顶点的度。对于无向图或无向网而言，顶点 vi 的度就是邻接矩阵中第 i 行或第 i 列中非 0 或非∞的元素的个数。对于有向图或有向网而言，顶点 vi 的入度是邻接矩阵中第 i 列中非 0 或非∞的元素的个数，顶点 vi 的出度是邻接矩阵中第 i 行中非 0 或非∞的元素的个数。

（3）可以很方便地查找图中任一条边或弧的权值，只要 A[i][j]为 0 或∞，就说明顶点 vi 和 vj 之间不存在边或弧。但是，要确定图中有多少条边或弧，则必须按行、按列对每个元素进行检测，所花费的时间代价是很大的。这是用邻接矩阵存储图的局限性。

下面以无向图的邻接矩阵类的实现来说明图的邻接矩阵表示的类的实现。

无向图邻接矩阵类 GraphAdjMatrix<T>中有三个成员字段，一个是 Node<T>类型的一维数组 nodes，存放图中的顶点信息；一个是整型的二维数组 matirx，表示图的邻接矩阵，存放边的信息；一个是整数 numEdges，表示图中边的数目。因为图的邻接矩阵存储结构对于确定图中边或弧的数目要花费很大的时间代价，所以设了这个字段。

无向图邻接矩阵类 GraphAdjMatrix<T>的实现如下所示。

```
public class GraphAdjMatrix<T> : IGraph<T>
{
    private Node<T>[] nodes;        //顶点数组
    private int numEdges;           //边的数目
    private int[ ,] matrix;         //邻接矩阵数组

   //构造器
    public GraphAdjMatrix (int n)
    {
```

```
        nodes = new Node<T>[n];
        matrix = new int[n,n];
        numEdges = 0;
    }

    //获取索引为index的顶点的信息
    public Node<T> GetNode(int index)
    {
        return nodes[index];
    }

    //设置索引为index的顶点的信息
    public void SetNode(int index, Node<T> v)
    {
        nodes[index] = v;
    }

    //边的数目属性
    public int NumEdges
    {
        get
        {
            return numEdges;
        }
        set
        {
            numEdges = value;
        }
    }

    //获取matrix[index1, index2]的值
    public int GetMatrix(int index1, int index2)
    {
        return matrix[index1, index2];
    }

    //设置matrix[index1, index2]的值
    public void SetMatrix(int index1, int index2)
    {
        matrix[index1, index2] = 1;
    }

    //获取顶点的数目
    public int GetNumOfVertex()
```

```
{
    return nodes.Length;
}

//获取边的数目
public int GetNumOfEdge()
{
    return numEdges;
}

//判断v是否是图的顶点
public bool IsNode(Node<T> v)
{
    //遍历顶点数组
    foreach (Node<T> nd in nodes)
    {
        //如果顶点nd与v相等，则v是图的顶点，返回true
        if (v.Equals(nd))
        {
            return true;
        }
    }

    return false;
}

//获取顶点v在顶点数组中的索引
public int GetIndex(Node<T> v)
{
    int i = -1;

    //遍历顶点数组
    for (i = 0; i < nodes.Length; ++i)
    {
        //如果顶点v与nodes[i]相等，则v是图的顶点，返回索引值i。
        if (nodes[i].Equals(v))
        {
            return i;
        }
    }
    return i;
}

//在顶点v1和v2之间添加权值为v的边
```

```
public void SetEdge(Node<T> v1, Node<T> v2, int v)
{
    //v1或v2不是图的顶点
    if (!IsNode(v1) || !IsNode(v2))
    {
        Console.WriteLine("Node is not belong to Graph!");
        return;
    }

    //不是无向图
    if(v != 1)
    {
         Console.WriteLine("Weight is not right!");
         return;
     }

    //矩阵是对称矩阵
    matrix[GetIndex(v1), GetIndex(v2)] = v;
    matrix[GetIndex(v2), GetIndex(v1)] = v;
    ++numEdges;
}

//删除顶点v1和v2之间的边
public void DelEdge(Node<T> v1, Node<T> v2)
{
    //v1或v2不是图的顶点
    if (!IsNode(v1) || !IsNode(v2))
    {
        Console.WriteLine("Node is not belong to Graph!");
        return;
    }

    //顶点v1与v2之间存在边
    if (matrix[GetIndex(v1), GetIndex(v2)] == 1)
    {
        //矩阵是对称矩阵
        matrix[GetIndex(v1), GetIndex(v2)] = 0;
        matrix[GetIndex(v2), GetIndex(v1)] = 0;
        --numEdges;
    }
}

//判断顶点v1与v2之间是否存在边
public bool IsEdge(Node<T> v1, Node<T> v2)
```

```
        {
            //v1或v2不是图的顶点
            if (!IsNode(v1) || !IsNode(v2))
            {
                Console.WriteLine("Node is not belong to Graph!");
                return false;
            }

            //顶点v1与v2之间存在边
            if (matrix[GetIndex(v1), GetIndex(v2)] == 1)
            {
                return true;
            }
            else  //不存在边
            {
                return false;
            }
        }
    }
```

无向图邻接矩阵类 GraphAdjMatrix<T>除了实现了接口 IGraph<T>中的方法外，本身还有两个成员方法，一个是 IsNode，功能是判断一个顶点是否是无向图的顶点，因为我们对不是图中的顶点进行处理是毫无意义的；一个是 GetIndex，功能是得到图的某个顶点在 nodes 数组中的序号，因为 matrix 数组的下标是整数而不是顶点类型。

由于无向图邻接矩阵类 GraphAdjMatrix<T>中的成员方法的实现比较简单，这里就不一一进行说明。

6.2.2 邻接表

邻接表(Adjacency List)是图的一种顺序存储与链式存储相结合的存储结构，类似于树的孩子链表表示法。顺序存储指的是图中的顶点信息用一个顶点数组来存储，一个顶点数组元素是一个顶点结点，顶点结点有两个域，一个是数据域 data，存放与顶点相关的信息，一个是引用域 firstAdj，存放该顶点的邻接表的第一个结点的地址。顶点的邻接表是把所有邻接于某顶点的顶点构成的一个表，它是采用链式存储结构。所以，我们说邻接表是图的一种顺序存储与链式存储相结合的存储结构。其中，邻接表中的每个结点实际上保存的是与该顶点相关的边或弧的信息，它有两个域，一个是邻接顶点域 adjvex，存放邻接顶点的信息，实际上就是邻接顶点在顶点数组中的序号；一个是引用域 next，存放下一个邻接顶点的结点的地址。顶点结点和邻接表结点的结构如图 6.9 所示。

(a) 顶点结点　　(b) 邻接表结点

图 6.9 顶点结点和邻接表结点的结构

而对于网的邻接表结点还需要存储边上的信息（如权值），所以结点应增设

一个域 info。网的邻接表结点的结构如图 6.10 所示。

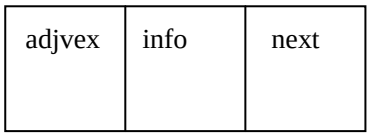


图 6.10 网的邻接表结点

图 6.1(a)的邻接表如图 6.11 所示。

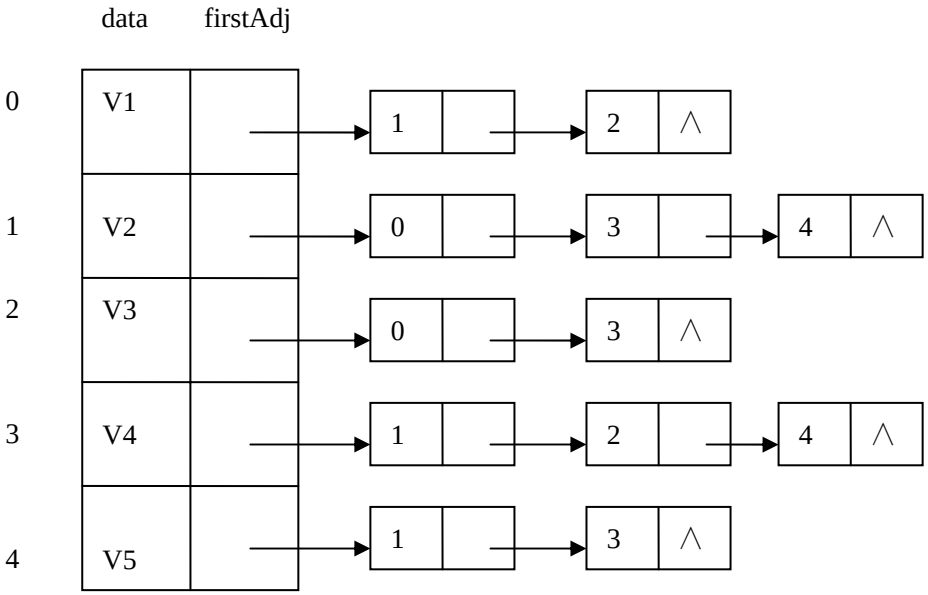


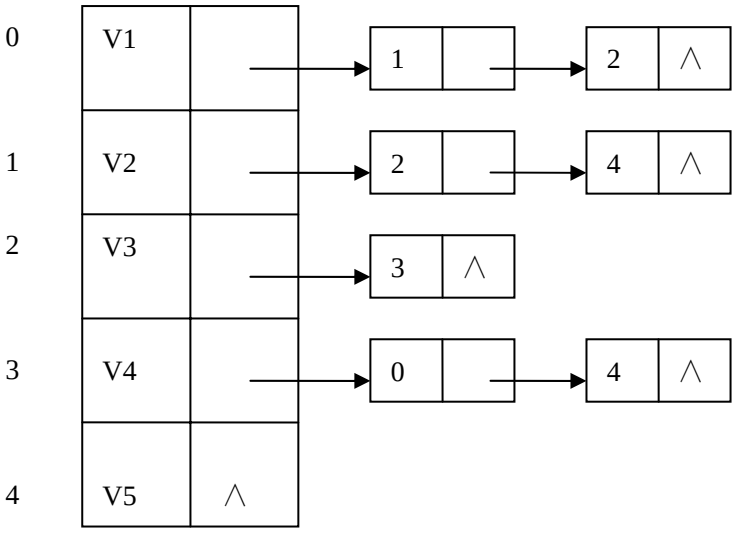
图 6.11 图 6.1(a)的邻接表

若无向图中有 n 个顶点和 e 条边，则它的邻接表需 n 个顶点结点和 2e 个邻接表结点，在边稀疏（$e<<\frac{n(n-1)}{2}$）的情况下，用邻接表存储图比用邻接矩阵节省存储空间，当与边相关的信息较多时更是如此。

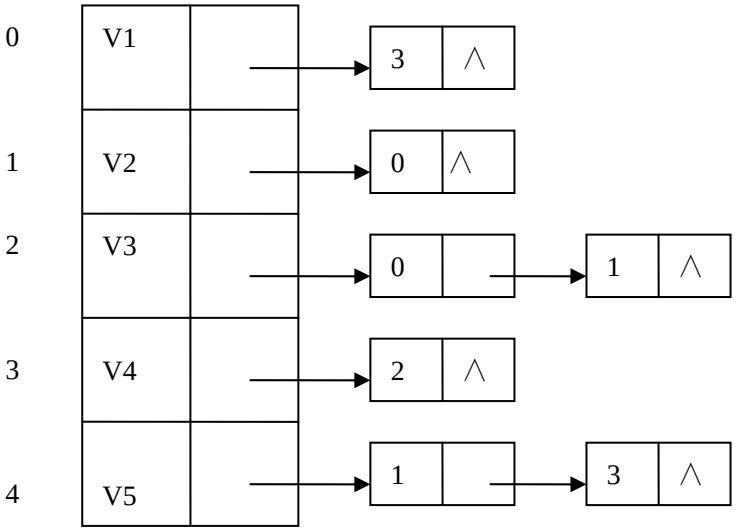
在无向图的邻接表中，顶点 vi 的度恰为第 i 个邻接表中的结点数；而在有向图中，第 i 的邻接表中的结点数只是顶点 vi 的出度，为求入度，必须遍历整个邻接表。在所有邻接表中其邻接顶点域的值为 i 的结点的个数是顶点 vi 的入度。有时，为了便于确定顶点的入度或者以顶点 vi 为头的弧，可以建立一个有向图的逆邻接表，即对每个顶点 vi 建立一个以 vi 为头的弧的邻接表。图 6.12 是图 6.1(b)的邻接表和逆邻接表。

在建立邻接表或逆邻接表时，若输入的顶点信息即为顶点的编号，则建立邻接表的时间复杂度为 O（n+e），否则，需要查找才能得到顶点在图中的位置，则时间复杂度为 O（n*e）。

在邻接表上很容易找到任一顶点的第一个邻接点和下一个邻接点。但要判定任意两个顶点（vi 和 v j）之间是否有边或弧相连，则需查找第 i 个或 j 个邻接表，因此，不如邻接矩阵方便。



(a) 邻接表



(b) 逆邻接表

图 6.12 图 6.1(b)的邻接表和逆邻接表

下面以无向图邻接表类的实现来说明图的邻接表类的实现。

无向图邻接表的邻接表结点类 adjListNode<T>有两个成员字段，一个是 adjvex，存储邻接顶点的信息，类型是整型；一个是 next，存储下一个邻接表结点的地址，类型是 adjListNode<T>。adjListNode<T>的实现如下所示。

```
public class adjListNode<T>
{
    private int adjvex;                    //邻接顶点
    private adjListNode<T> next;           //下一个邻接表结点

    //邻接顶点属性
    public int Adjvex
    {
        get
        {
```

```
            return adjvex;
        }
        set
        {
            adjvex = value;
        }
    }

    //下一个邻接表结点属性
    public adjListNode<T> Next
    {
        get
        {
            return next;
        }
        set
        {
            next = value;
        }
    }

    //构造器
    public adjListNode(int vex)
    {
        adjvex = vex;
        next = null;
    }
}
```

无向图邻接表的顶点结点类 VexNode<T>有两个成员字段，一个 data，它存储图的顶点本身的信息，类型是 Node<T>；一个是 firstAdj，存储顶点的邻接表的第 1 个结点的地址，类型是 adjListNode<T>。VexNode<T>的实现如下所示。

```
public class VexNode<T>
{
    private Node<T> data;              //图的顶点
    private adjListNode<T> firstAdj;  //邻接表的第1个结点

    //图的顶点属性
    public Node<T> Data
    {
        get
        {
            return data;
        }
        set
```

```
            {
                data = value;
            }
        }

        //邻接表的第1个结点属性
        public adjListNode<T> FirstAdj
        {
            get
            {
                return firstAdj;
            }
            set
            {
                firstAdj = value;
            }
        }

        //构造器
        public VexNode()
        {
            data = null;
            firstAdj = null;
        }

        //构造器
        public VexNode(Node<T> nd)
        {
            data = nd;
            firstAdj = null;
        }

        //构造器
        public VexNode(Node<T> nd, adjListNode<T> alNode)
        {
            data = nd;
            firstAdj = alNode;
        }
    }
```

无向图邻接表类 GraphAdjList<T>有一个成员字段 adjList，表示邻接表数组，数组元素的类型是 VexNode<T>。GraphAdjList<T>实现了接口 IGraph<T>中的方法。与无向图邻接矩阵类 GraphAdjMatrix<T>一样，GraphAdjList<T>实现了两个成员方法 IsNode 和 GetIndex。功能与 GraphAdjMatrix<T>一样。无向图邻接表类 GraphAdjList<T>的实现如下所示。

```
public class GraphAdjList<T> : IGraph<T>
{
    //邻接表数组
    private VexNode<T>[] adjList;

    //索引器
    public VexNode<T> this[int index]
    {
        get
        {
            return adjList[index];
        }
        set
        {
            adjList[index] = value;
        }
    }

    //构造器
    public GraphAdjList(Node<T>[] nodes)
    {
        adjList = new VexNode<T>[nodes.Length];
        for (int i = 0; i < nodes.Length; ++i )
        {
            adjList[i].Data = nodes[i];
            adjList[i].FirstAdj = null;
        }
    }

    //获取顶点的数目
    public int GetNumOfVertex()
    {
        return adjList.Length;
    }

    //获取边的数目
    public int GetNumOfEdge()
    {
        int i = 0;

        //遍历邻接表数组
        foreach (VexNode<T> nd in adjList)
        {
            adjListNode<T> p = nd.FirstAdj;
```

```
            while (p != null)
            {
                ++i;
                p = p.Next
            }
        }

        return i / 2;
    }

    //判断v是否是图的顶点
    public bool IsNode(Node<T> v)
    {
        //遍历邻接表数组
        foreach (VexNode<T> nd in adjList)
        {
            //如果v等于nd的data，则v是图中的顶点，返回true
            if (v.Equals(nd.Data))
            {
                return true;
            }
        }

        return false;
    }

    //获取顶点v在邻接表数组中的索引
    public int GetIndex(Node<T> v)
    {
        int i = -1;

        //遍历邻接表数组
        for (i = 0; i < adjList.Length; ++i)
        {
            //邻接表数组第i项的data值等于v，则顶点v的索引为i
            if (adjList[i].Data.Equals(v))
            {
                return i;
            }
        }
        return i;
    }
```

```
//在顶点v1和v2之间添加权值为v的边
public void SetEdge(Node<T> v1, Node<T> v2,int v)
{
    //v1或v2不是图的顶点或者v1和v2之间存在边
    if (!IsNode(v1) || !IsNode(v2) || IsEdge(v1,v2))
    {
        Console.WriteLine("Node is not belong to Graph!");
        return;
    }

    //权值不对
    if(v != 1)
    {
         Console.WriteLine("Weight is not right!");
         return;
    }

    //处理顶点v1的邻接表
    adjListNode<T> p = new adjListNode<T>(GetIndex(v2));

    //顶点v1没有邻接顶点
    if (adjList[GetIndex(v1)].FirstAdj == null)
    {
        adjList[GetIndex(v1)].FirstAdj = p;
    }
    //顶点v1有邻接顶点
    else
    {
        p.Next = adjList[GetIndex(v1)].FirstAdj;
        adjList[GetIndex(v1)].FirstAdj = p;
    }

    //处理顶点v2的邻接表
    p = new adjListNode<T>(GetIndex(v1));

    //顶点v2没有邻接顶点
    if (adjList[GetIndex(v2)].FirstAdj == null)
    {
        adjList[GetIndex(v2)].FirstAdj = p;
    }
    //顶点v1有邻接顶点
    else
    {
        p.Next = adjList[GetIndex(v2)].FirstAdj;
```

```
            adjList[GetIndex(v2)].FirstAdj = p;
        }
    }

    //删除顶点v1和v2之间的边
    public void DelEdge(Node<T> v1, Node<T> v2)
    {
        //v1或v2不是图的顶点
        if (!IsNode(v1) || !IsNode(v2))
        {
            Console.WriteLine("Node is not belong to Graph!");
            return;
        }

        //顶点v1与v2之间有边
        if (IsEdge(v1,v2))
        {
            //处理顶点v1的邻接表中的顶点v2的邻接表结点
            adjListNode<T> p = adjList[GetIndex(v1)].FirstAdj;
            adjListNode<T> pre = null;

            while (p != null)
            {
                if (p.Adjvex != GetIndex(v2))
                {
                    pre = p;
                    p = p.Next;
                }
            }

            pre.Next = p.Next;

            //处理顶点v2的邻接表中的顶点v1的邻接表结点
            p = adjList[GetIndex(v2)].FirstAdj;
            pre = null;

            while (p != null)
           {
                if (p.Adjvex != GetIndex(v1))
                {
                    pre = p;
                    p = p.Next;
                }
            }
```

```
                pre.Next = p.Next;
            }
        }

        //判断v1和v2之间是否存在边
        public bool IsEdge(Node<T> v1, Node<T> v2)
        {
            //v1或v2不是图的顶点
            if (!IsNode(v1) || !IsNode(v2))
            {
                Console.WriteLine("Node is not belong to Graph!");
                return false;
            }

            adjListNode<T> p = adjList[GetIndex(v1)].FirstAdj;
            while (p != null)
            {
                if (p.Adjvex == GetIndex(v2))
                {
                    return true;
                }

                p = p.Next;
            }

            return false;
        }
    }
```

下面对成员方法进行说明：

1、GetNumOfVertex()

算法思路：求无向图的顶点数比较简单，直接返回 adjList 数组的长度就可以了。

算法实现如下：

```
        public int GetNumOfVertex()
        {
            return adjList.Length;
        }
```

2、GetNumOfEdge()

算法思路：求无向图的边数比求顶点数要复杂一些，需要求出所有顶点的邻接表的结点的个数，然后除以 2。

算法实现如下：

```
        public int GetNumOfEdge()
        {
```

```
            int i = 0;
            foreach (VexNode<T> nd in adjList)
            {
                adjListNode<T> p = nd.FirstAdj;

                while (p != null)
                {
                    ++i;
                }
            }

            return i / 2;
        }
```

3、SetEdge(Node<T> v1, Node<T> v2, int v)

算法思路：首先判断顶点 v1 和 v2 是否是图的顶点和 v1 和 v2 是否存在边。如果 v1 和 v2 不是图的顶点和 v1 和 v2 存在边，不作处理。然后，判断 v 的值是否为 1，为 1 不作处理。否则，先分配一个邻接表结点，其 adjvex 域是 v2 在 adjList 数组中的索引号，然后把该结点插入到顶点 v1 的邻接表的表头；然后再分配一个邻接表结点，其 adjvex 域是 v1 在 adjList 数组中的索引号，然后把该结点插入到顶点 v2 的邻接表的表头。

本算法是把邻接表结点插入到顶点邻接表的表头，当然，也可以插入到邻接表的表尾，或者按照某种要求插入，只是对插入这个操作而言，在表的头部插入是最简单的，而本书在后面关于图的处理，如图的深度优先遍历和广度优先遍历等，对图的顶点没有特殊要求，所以采用了在邻接表的头部插入结点。如果对图的顶点有特殊要求，则需要按照一定的要求进行插入，需要修改这里的代码。

算法实现如下：

```
        public void SetEdge(Node<T> v1, Node<T> v2, int v)
        {
            if (!IsNode(v1) || !IsNode(v2) || IsEdge(v1,v2))
            {
                Console.WriteLine("Node is not belong to Graph!");
                return;
            }

            if(v != 1)
            {
                 Console.WriteLine("Weight is not right!");
                 return;
            }

            adjListNode<T> p = new adjListNode<T>(GetIndex(v2));
            if (adjList[GetIndex(v1)].FirstAdj == null)
            {
                adjList[GetIndex(v1)].FirstAdj = p;
```

```
            }
            else
            {
                p.Next = adjList[GetIndex(v1)].FirstAdj;
                adjList[GetIndex(v1)].FirstAdj = p;
            }

            p = new adjListNode<T>(GetIndex(v1));
            if (adjList[GetIndex(v2)].FirstAdj == null)
            {
                adjList[GetIndex(v2)].FirstAdj = p;
            }
            else
            {
                p.Next = adjList[GetIndex(v2)].FirstAdj;
                adjList[GetIndex(v2)].FirstAdj = p;
            }
        }
```

4、DelEdge(Node<T> v1, Node<T> v2)

算法思路：首先判断顶点 v1 和 v2 是否是图的顶点以及 v1 和 v2 是否存在边。如果 v1 和 v2 不是图的顶点或 v1 和 v2 不存在边，不作处理。否则，先在顶点 v1 的邻接表中删除 adjVex 的值等于顶点 v2 在 adjList 数组中的序号结点，然后删除顶点 v2 的邻接表中 adjVex 的值等于顶点 v1 在 adjList 数组中的序号结点。

算法实现如下：

```
        public void DelEdge(Node<T> v1, Node<T> v2)
        {
            if (!IsNode(v1) || !IsNode(v2))
            {
                Console.WriteLine("Node is not belong to Graph!");
                return;
            }

            if (IsEdge(v1,v2))
            {
                adjListNode<T> p = adjList[GetIndex(v1)].FirstAdj;
                adjListNode<T> pre = null;

                while (p != null)
                {
                    if (p.Adjvex != GetIndex(v2))
                    {
                        pre = p;
                        p = p.Next;
```

```
                }
            }

            pre.Next = p.Next;

            p = adjList[GetIndex(v2)].FirstAdj;
            pre = null;

            while (p != null)
            {
                if (p.Adjvex != GetIndex(v1))
                {
                    pre = p;
                    p = p.Next;
                }
            }

            pre.Next = p.Next;
        }
    }
```

5、IsEdge(Node<T> v1, Node<T> v2)

算法思路：首先判断顶点 v1 和 v2 是否是图的顶点。如果 v1 和 v2 不是图的顶点，不作处理。否则，在顶点 v1（或 v2）的邻接表中查找是否存在 adjVex 的值等于 v2（或 v1）在 adjList 中的序号的结点，如果存在，则返回 true，否则返回 false。

算法实现如下：

```
    public bool IsEdge(Node<T> v1, Node<T> v2)
    {
        if (!IsNode(v1) || !IsNode(v2))
        {
            Console.WriteLine("Node is not belong to Graph!");
            return false;
        }

        adjListNode<T> p = adjList[GetIndex(v1)].FirstAdj;
        while (p != null)
        {
            if (p.Adjvex == GetIndex(v2))
            {
                return true;
            }

            p = p.Next;
```

```
            }

            return false;
        }
```

6.3 图的遍历

图的遍历是指从图中的某个顶点出发，按照某种顺序访问图中的每个顶点，使每个顶点被访问一次且仅一次。图的遍历与树的遍历操作功能相似。图的遍历是图的一种基本操作，图的许多其他操作都是建立在遍历操作的基础之上的。

然而，图的遍历要比树的遍历复杂得多。这是因为图中的顶点之间是多对多的关系，图中的任何一个顶点都可能和其它的顶点相邻接。所以，在访问了某个顶点之后，从该顶点出发，可能沿着某条路径遍历之后，又回到该顶点上。例如，在图 6.1(b)的图中，由于图中存在回路，因此在访问了 v1、v3、v4 之后，沿着边<v4，v1>又回到了 v1 上。为了避免同一顶点被访问多次，在遍历图的过程中，必须记下每个已访问过的顶点。为此，可以设一个辅助数组 visited[n]，n 为图中顶点的数目。数组中元素的初始值全为 0，表示顶点都没有被访问过，如果顶点 vi 被访问，visited[i-1]为 1。

图的遍历有深度优先遍历和广度优先遍历两种方式，它们对图和网都适用。

6.3.1 深度优先遍历

图的深度优先遍历（Depth_First Search）类似于树的先序遍历，是树的先序遍历的推广。

假设初始状态是图中所有顶点未曾被访问过，则深度优先遍历可从图中某个顶点 v 出发，访问此顶点，然后依次从 v 的未被访问的邻接顶点出发深度优先遍历图，直至图中所有和 v 有路径相通的顶点都被遍历过。若此时图中尚有未被访问的顶点，则另选图中一个未被访问的顶点作为起始点，重复上述过程，直到图中所有顶点都被访问到为止。

【例 6-1】按深度优先遍历算法对图 6.13(a)进行遍历。

图 6.13(a)所示的无向图的深度优先遍历的过程如图 6.13(b)所示。假设从顶点 v1 出发，在访问了顶点 v1 之后，选择邻接顶点 v2，因为 v2 未被访问过，所以从 v2 出发进行深度优先遍历。依次类推，接着从 v4、v8、v5 出发进行深度优先遍历。当访问了 v5 之后，由于 v5 的邻接顶点 v2 和 v8 都已被访问，所以遍历退回到 v8。由于同样的理由，遍历继续退回到 v4、v2 直到 v1。由于 v1 的另一个邻接顶点 v3 未被访问，所以又从 v3 开始进行深度优先遍历，这样得到该图的深度优先遍历的顶点序列为 v1→v2→v4→v8→v5→v3→v6→v7。

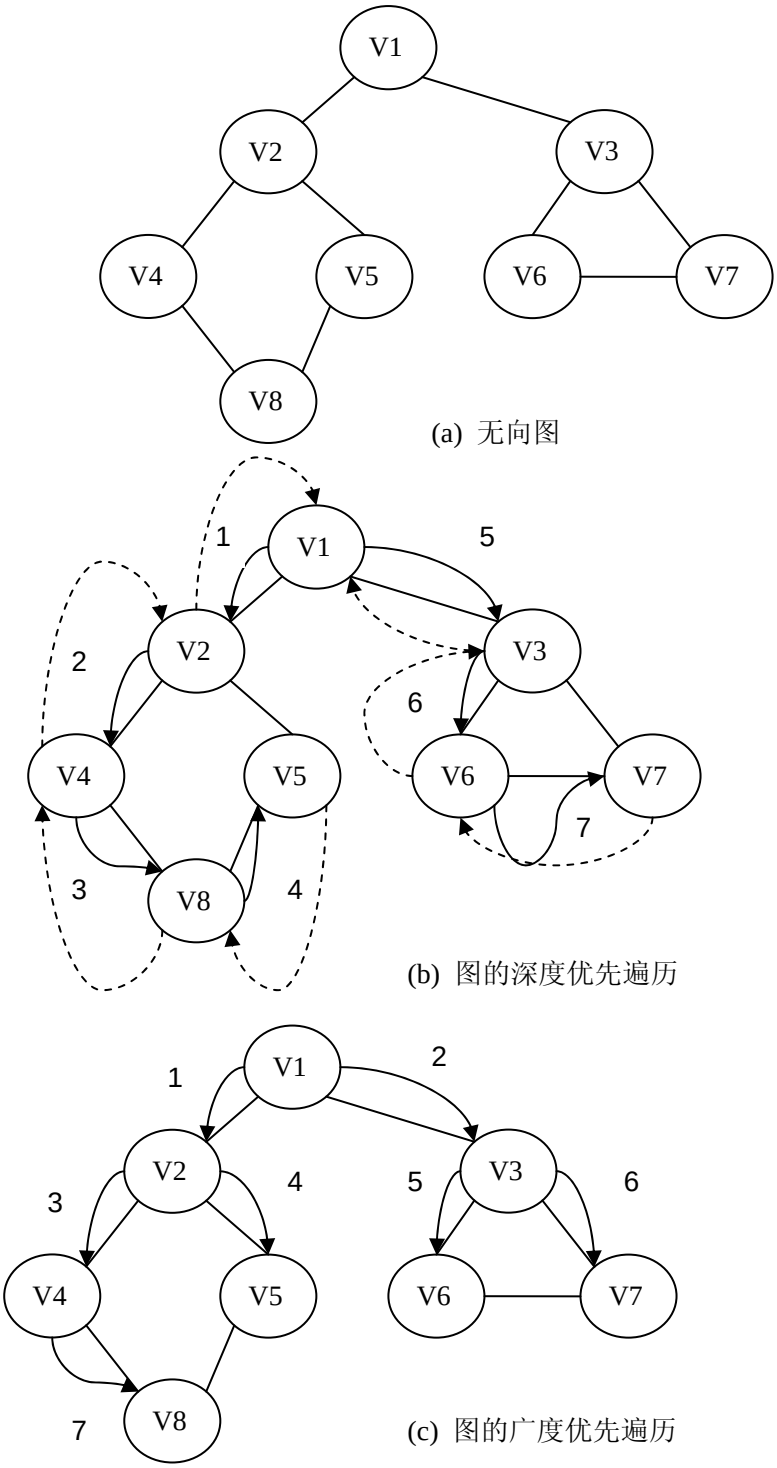


图 6.13 图的遍历

显然，这是一个递归的过程。下面以无向图的邻接表存储结构为例来实现图的深度优先遍历算法。在类中增设了一个整型数组的成员字段 visited，它的初始值全为 0，表示图中所有的顶点都没有被访问过。如果顶点 vi 被访问，visited[i-1]为 1。并且，把该算法作为无向图的邻接表类 GraphAdjList<T>的成员方法。

由于增设了成员字段 visited，所以在类的构造器中添加以下代码。

```
public GraphAdjList(Node<T>[] nodes)
{
```

```
        adjList = new VexNode<T>[nodes.Length];
        for (int i = 0; i < nodes.Length; ++i )
        {
            adjList[i].Data = nodes[i];
            adjList[i].FirstAdj = null;
        }

        //以下为添加的代码
        visited = new int[adjList.Length];
        for (int i = 0; i < visited.Length; ++i)
        {
            visited[i] = 0;
        }
    }
```

无向图的深度优先遍历算法的实现如下：

```
    public void DFS()
    {
        for (int i = 0; i < visited.Length; ++i)
        {
            if (visited[i] == 0)
            {
                DFSAL(i);
            }
        }
    }

    //从某个顶点出发进行深度优先遍历
    public void DFSAL(int i)
    {
        visited[i] = 1;
        adjListNode<T> p = adjList[i].FirstAdj;

        while (p != null)
        {
            if (visited[p.Adjvex] == 0)
            {
                DFSAL(p.Adjvex);
            }

            p = p.Next;
        }
    }
```

分析上面的算法，在遍历图时，对图中每个顶点至多调用一次DFS方法，因为一旦某个顶点被标记成已被访问，就不再从它出发进行遍历。因此，遍历图的

过程实质上是对每个顶点查找其邻接顶点的过程。其时间复杂度取决于所采用的存储结构。当图采用邻接矩阵作为存储结构时，查找每个顶点的邻接顶点的时间复杂度为$O(n^2)$，其中，n为图的顶点数。而以邻接表作为图的存储结构时，查找邻接顶点的时间复杂度为O(e)，其中，e为图中边或弧的数目。因此，当以邻接表作为存储结构时，深度优先遍历图的时间复杂度为O(n+e)。

6.3.2 广度优先遍历

图的广度优先遍历（Breadth_First Search）类似于树的层序遍历。

假设从图中的某个顶点 v 出发，访问了 v 之后，依次访问 v 的各个未曾访问的邻接顶点。然后分别从这些邻接顶点出发依次访问它们的邻接顶点，并使“先被访问的顶点的邻接顶点”先于“后被访问的顶点的邻接顶点”被访问，直至图中所有已被访问的顶点的邻接顶点都被访问。若此时图中尚有顶点未被访问，则另选图中未被访问的顶点作为起点，重复上述过程，直到图中所有的顶点都被访问为止。换句话说，广度优先遍历图的过程是以某个顶点 v 作为起始点，由近至远，依次访问和 v 有路径相通且路径长度为 1，2，…的顶点。

【例 6-2】按广度优先遍历算法对图 6.13(a)进行遍历。

图 6.13(a)所示的无向图的广度优先遍历的过程如图 6.13(c)所示。假设从顶点 v1 开始进行广度优先遍历，首先访问顶点 v1 和它的邻接顶点 v2 和 v3，然后依次访问 v2 的邻接顶点 v4 和 v5，以及 v3 的邻接顶点 v6 和 v7，最后访问 v4 的邻接顶点 v8。由于这些顶点的邻接顶点都已被访问，并且图中所有顶点都已被访问，由此完成了图的遍历，得到的顶点访问序列为：v1→v2→v3→v4→v5→v6→v7→v8，其遍历过程如图 6.13(c)所示。

和深度优先遍历类似，在广度优先遍历中也需要一个访问标记数组，我们采用与深度优先遍历同样的数组。并且，为了顺序访问路径长度为 1，2，…的顶点，需在算法中附设一个队列来存储已被访问的路径长度为 1，2，…的顶点。

以邻接表作为存储结构的无向图的广度优先遍历算法的实现如下，队列是循环顺序队列。

```
public void BFS()
{
    for (int i = 0; i < visited.Length; ++i)
    {
        if (visited[i] == 0)
        {
            BFSAL(i);
        }
    }
}

//从某个顶点出发进行广度优先遍历
public void BFSAL(int i)
{
    visited[i] = 1;
    CSeqQueue<int> cq = new CSeqQueue<int>(visited.Length);
    cq.In(i);
```

```
            while (!cq.IsEmpty())
            {
                int k = cq.Out();
                adjListNode<T> p = adjList[k].FirstAdj;

                while (p != null)
                {
                    if (visited[p.Adjvex] == 0)
                    {
                        visited[p.Adjvex] = 1;
                        cq.In(p.Adjvex);
                    }

                    p = p.Next;
                }
            }
        }
```

分析上面的算法，每个顶点至多入队列一次。遍历图的过程实质上是通过边或弧查找邻接顶点的过程，因此，广度优先遍历算法的时间复杂度与深度优先遍历相同，两者的不同之处在于对顶点的访问顺序不同。

6.4 图的应用

6.4.1 最小生成树

1、最小生成树的基本概念

由生成树的定义可知，无向连通图的生成树不是唯一的，对连通图的不同遍历就得到不同的生成树。图 6.14 所示是图 6.13(a)所示的无向连通图的部分生成树。

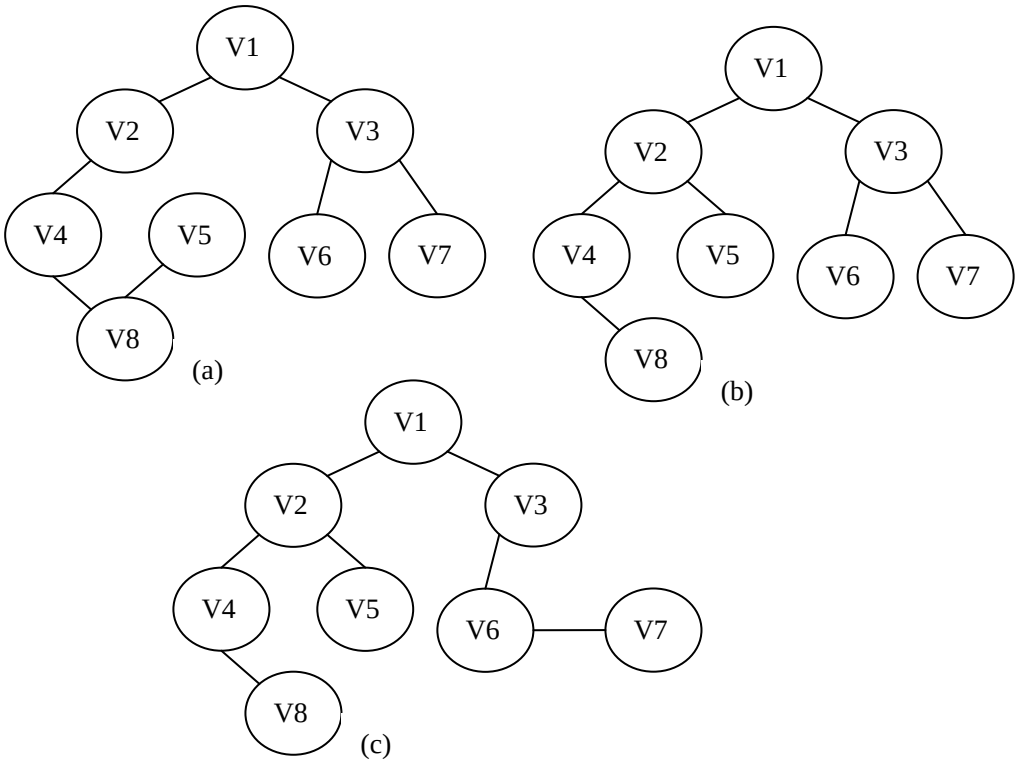


图 6.14 图 6.13(a)所示的无向连通图的生成树

如果是一个无向连通网，那么它的所有生成树中必有一棵边的权值总和最小的生成树，我们称这棵生成树为最小代价生成树(Minimum Cost Spanning Tree)，简称最小生成树。

许多应用问题都是一个求无向连通网的最小生成树问题。例如，要在 n 个城市之间铺设光缆，铺设光缆的费用很高，并且各个城市之间铺设光缆的费用不同。一个目标是要使这 n 个城市的任意两个之间都可以直接或间接通信，另一个目标是要使铺设光缆的总费用最低。如果把 n 个城市看作是图的 n 个顶点，两个城市之间铺设的光缆看作是两个顶点之间的边，这实际上就是求一个无向连通网的最小生成树问题。

由最小生成树的定义可知，构造有 n 个顶点的无向连通网的最小生成树必须满足以下三个条件：

（1）构造的最小生成树必须包括 n 个顶点；

（2）构造的最小生成树有且仅有 n-1 条边；

（3）构造的最小生成树中不存在回路。

构造最小生成树的方法有许多种，典型的方法有两种，一种是普里姆(Prim)算法，一种是克鲁斯卡尔(Kruskal)算法。

2、普里姆(Prim)算法

假设 G=（V，E）为一无向连通网，其中，V 为网中顶点的集合，E 为网中边的集合。设置两个新的集合 U 和 T，其中，U 为 G 的最小生成树的顶点的集合，T 为 G 的最小生成树的边的集合。普里姆算法的思想是：令集合 U 的初值为 U={u1}（假设构造最小生成树时从顶点 u1 开始），集合 T 的初值为 T={}。从所有的顶点 u∈U 和顶点 v∈V-U 的带权边中选出具有最小权值的边（u, v），将顶点 v 加入集合 U 中，将边（u, v）加入集合 T 中。如此不断地重复直到 U=V 时，最小生成树构造完毕。此时，集合 U 中存放着最小生成树的所有顶点，集合 T

中存放着最小生成树的所有边。

【例 6-3】以图 6.2(a)为例，说明用普里姆算法构造图的无向连通网的最小生成树的过程。

为了分析问题的方便，把图 6.2(a)中所示的无向连通网重新画在图 6.15 中，如图 6.15(a)所示。初始时，算法的集合 U={A}，集合 V-U={B,C,D,E}，集合 T={}，如图 6.15(b)所示。在所有 u 为集合 U 中顶点、v 为集合 V-U 中顶点的边(u,v)中寻找具有最小权值的边，寻找到的边是(A,D)，权值为 20，把顶点 B 加入到集合 U 中，把边(A,D)加入到集合 T 中，如图 6.15(c)所示。在所有 u 为集合 U 中顶点、v 为集合 V-U 中顶点的边(u,v)中寻找具有最小权值的边，寻找到的边是(D,E)，权值为 10，把顶点 E 加入到集合 U 中，把边(D,E)加入到集合 T 中，如图 6.15(d)所示。随后依次从集合 V-U 中加入到集合 U 中的顶点为 B、C，依次加入到集合 T 中的边为(A,B)（权值为 60）、(E,C) （权值为 70），分别如图 6.15(e)、(f)所示。最后得到的图 6.15(f)所示就是原无向连通网的最小生成树。

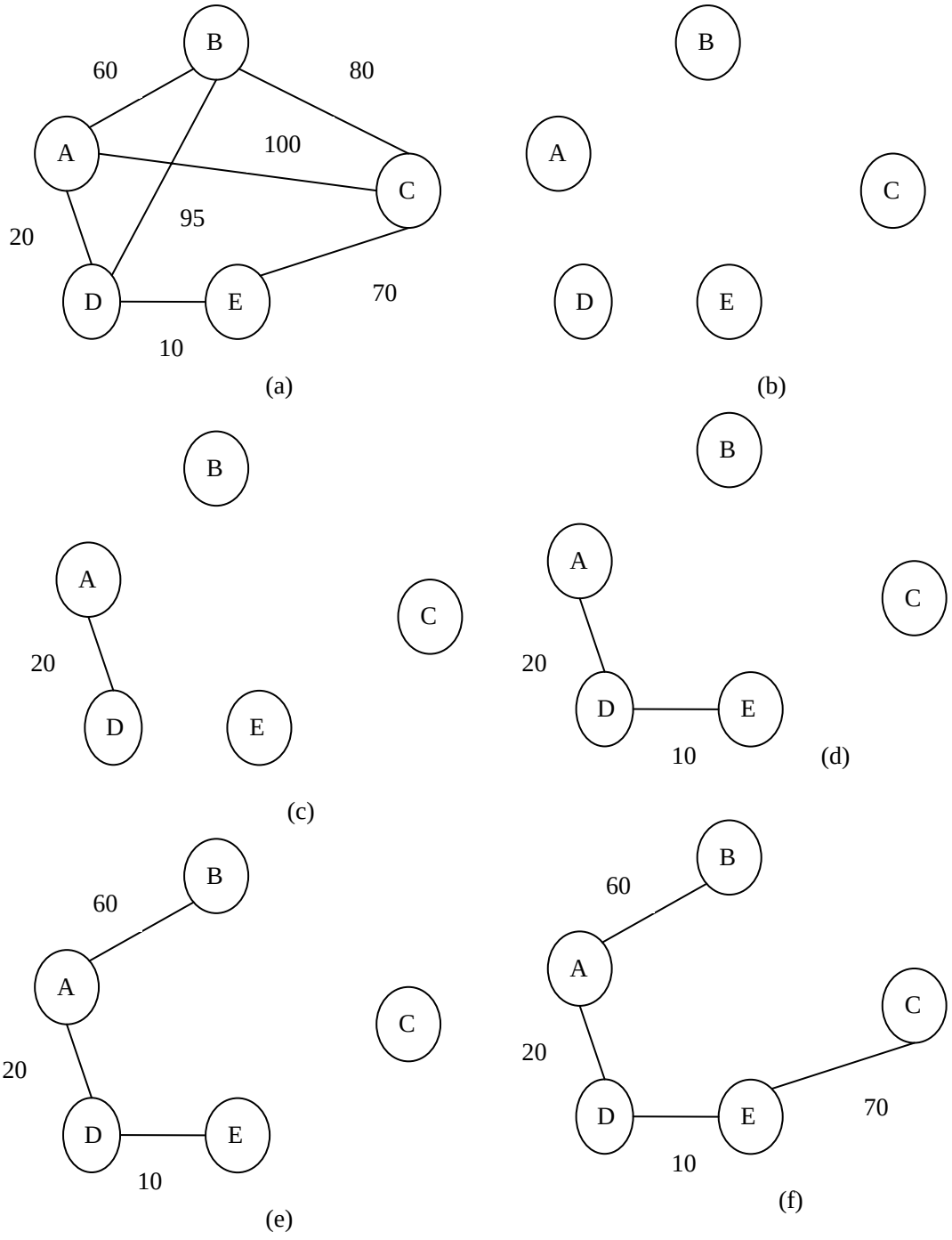


图 6.15 普里姆算法构造最小生成树的过程

本书以无向网的邻接矩阵类 NetAdjMatrix<T>来实现普里姆算法。NetAdjMatrix<T>类的成员字段与无向图邻接矩阵类 GraphAdjMatrix<T>的成员字段一样，不同的是，当两个顶点间有边相连接时，matirx 数组中相应元素的值是边的权值，而不是 1。

无向网邻接矩阵类 NetAdjMatrix<T>的实现如下所示。

```
public class NetAdjMatrix<T> : IGraph<T>
{
    private Node<T>[] nodes;       //顶点数组
    private int numEdges;          //边的数目
```

```
private int[,] matrix;        //邻接矩阵数组

//构造器
public NetAdjMatrix (int n)
{
    nodes = new Node<T>[n];
    matrix = new int[n,n];
    numEdges = 0;
}

//获取索引为index的顶点的信息
public Node<T> GetNode(int index)
{
    return nodes[index];
}

//设置索引为index的顶点的信息
public void SetNode(int index, Node<T> v)
{
    nodes[index] = v;
}

//边的数目属性
public int NumEdges
{
    get
    {
        return numEdges;
    }
    set
    {
        numEdges = value;
    }
}

//获取matrix[index1, index2]的值
public int GetMatrix(int index1, int index2)
{
    return matrix[index1, index2];
}

//设置matrix[index1, index2]的值
public void SetMatrix(int index1, int index2, int v)
{
```

```
        matrix[index1, index2] = v;
    }

    //获取顶点的数目
    public int GetNumOfVertex()
    {
        return nodes.Length;
    }

    //获取边的数目
    public int GetNumOfEdge()
    {
        return numEdges;
    }

    //v是否是无向网的顶点
    public bool IsNode(Node<T> v)
    {
        //遍历顶点数组
        foreach (Node<T> nd in nodes)
        {
            //如果顶点nd与v相等，则v是图的顶点，返回true
            if (v.Equals(nd))
            {
                return true;
            }
        }

        return false;
    }

    //获得顶点v在顶点数组中的索引
    public int GetIndex(Node<T> v)
    {
        int i = -1;

        //遍历顶点数组
        for (i = 0; i < nodes.Length; ++i)
        {
            //如果顶点nd与v相等，则v是图的顶点，返回索引值
            if (nodes[i].Equals(v))
            {
                return i;
            }
```

```
        }
        return i;
    }

    //在顶点v1、v2之间添加权值为v的边
    public void SetEdge(Node<T> v1, Node<T> v2, int v)
    {
        //v1或v2不是无向网的顶点
        if (!IsNode(v1) || !IsNode(v2))
        {
            Console.WriteLine("Node is not belong to Graph!");
            return;
        }

        //矩阵是对称矩阵
        matrix[GetIndex(v1), GetIndex(v2)] = v;
        matrix[GetIndex(v2), GetIndex(v1)] = v;
        ++numEdges;
    }

    //删除v1和v2之间的边
    public void DelEdge(Node<T> v1, Node<T> v2)
    {
        //v1或v2不是无向网的顶点
        if (!IsNode(v1) || !IsNode(v2))
        {
            Console.WriteLine("Node is not belong to Graph!");
            return;
        }

        //v1和v2之间存在边
        if (matrix[GetIndex(v1), GetIndex(v2)] != int.MaxValue)
        {
            //矩阵是对称矩阵
            matrix[GetIndex(v1), GetIndex(v2)] = int.MaxValue;
            matrix[GetIndex(v2), GetIndex(v1)] = int.MaxValue;
            --numEdges;
        }
    }

    //判断v1和v2之间是否存在边
    public bool IsEdge(Node<T> v1, Node<T> v2)
    {
        //v1或v2不是无向网的顶点
```

```
            if (!IsNode(v1) || !IsNode(v2))
            {
                Console.WriteLine("Node is not belong to Graph!");
                return false;
            }

            //v1和v2之间存在边
            if (matrix[GetIndex(v1), GetIndex(v2)] != int.MaxValue)
            {
                return true;
            }
            Else  //v1和v2之间不存在边
            {
                return false;
            }
        }
    }
```

为实现普里姆算法，需要设置两个辅助一维数组 lowcost 和 closevex，lowcost 用来保存集合 V-U 中各顶点与集合 U 中各顶点构成的边中具有最小权值的边的权值；closevex 用来保存依附于该边的在集合 U 中的顶点。假设初始状态时，U={u1}（u1 为出发的顶点），这时有 lowcost[0]=0，它表示顶点 u1 已加入集合 U 中。数组 lowcost 元素的值是顶点 u1 到其他顶点所构成的直接边的权值。然后不断选取权值最小的边(ui,uk)(ui∈U,uk∈V-U)，每选取一条边，就将 lowcost[k]置为 0，表示顶点 uk 已加入集合 U 中。由于顶点 uk 从集合 V-U 进入集合 U 后，这两个集合的内容发生了变化，就需要依据具体情况更新数组 lowcost 和 closevex 中部分元素的值。把普里姆算法 Prim 作为 NetAdjMatrix<T>类的成员方法。

普里姆算法 Prim 的实现如下：

```
public int[] Prim()
{
    int[] lowcost = new int[nodes.Length];   //权值数组
    int[] closevex = new int[nodes.Length];  //顶点数组
    int mincost = int.MaxValue;              //最小权值

    //辅助数组初始化
    for (int i = 1; i < nodes.Length; ++i)
    {
        lowcost[i] = matrix[0, i];
        closevex[i] = 0;
    }

    //某个顶点加入集合U
    lowcost[0] = 0;
    closevex[0] = 0;
```

```
        for(int i=0; i<nodes.Length; ++i)
        {
            int k = 1;
            int j = 1;

            //选取权值最小的边和相应的顶点
            while(j < nodes.Length)
            {
                if (lowcost[j] < mincost && lowcost[j] != 0)
                {
                    k = j;
                }
                ++j;
            }

            //新顶点加入集合U
            lowcost[k] = 0;

            //重新计算该顶点到其余顶点的边的权值
            for (j = 1; j < nodes.Length; ++j)
            {
                if (matrix[k, j] < lowcost[j])
                {
                    lowcost[j] = matrix[k, j];
                    closevex[j] = k;
                }
            }
        }

        return closevex;
    }
```

表 6.1 给出了在用普里姆算法构造图 6.15(a)的最小生成树的过程中数组 closevex 和 lowcost 及集合 U，V-U 的变化情况，读者可进一步加深对普里姆算法的理解。

在普里姆算法中，第一个for循环的执行次数为n-1，第二个for循环中又包括了一个while循环和一个for循环，执行次数为 $2(n-1)^2$，所以普里姆算法的时间复杂度为$O(n^2)$。

表 6.1 在用普里姆算法构造图 6.15(a)的最小生成树的过程中参数变化

顶点	i=0		1		2		3		4	
	low cost	close vex	low cost	close vex	low cost	close vex	low cost	close vex	low cost	close vex
A	0	0	0	3	0	3	0	3	0	3
B	60	0	60	0	60	0	0	0	0	0

C	100 0	100 0	70 4	70 4	0 4
D	20 0	0 0	0 0	0 0	0 0
E	∞ 0	10 3	0 3	0 3	0 3
U	A	A,D	A,D,E	A,D,E,B	A,D,E,B,C
V-U	B,C,D,E	B,C,E	B,C	C	
T	{}	{(A,D)}	{(A,D) (D,E)}	{(A,D) (D,E) (A,B)}	{(A,D) (D,E) (A,B) (E,C)}

3、克鲁斯卡尔(Kruskal)算法

克鲁斯卡尔算法的基本思想是：对一个有 n 个顶点的无向连通网，将图中的边按权值大小依次选取，若选取的边使生成树不形成回路，则把它加入到树中；若形成回路，则将它舍弃。如此进行下去，直到树中包含有 n-1 条边为止。

【例 6-4】以图 6.2(a)为例说明用克鲁斯卡尔算法求无向连通网最小生成树的过程。

第一步：首先比较网中所有边的权值，找到最小的权值的边(D,E)，加入到生成树的边集 TE 中，TE={(D,E)}。

第二步：再比较图中除边(D,E)的边的权值，又找到最小权值的边(A,D)并且不会形成回路，加入到生成树的边集 TE 中，TE={(A,D),(D,E)}。

第三步：再比较图中除 TE 以外的所有边的权值，找到最小的权值的边(A,B)并且不会形成回路，加入到生成树的边集 TE 中，TE={(A,D),(D,E),(A,B)}。

第四步：再比较图中除 TE 以外的所有边的权值，找到最小的权值的边(E,C)并且不会形成回路，加入到生成树的边集 TE 中，TE={(A,D),(D,E),(A,B),(E,C)}。

此时，边集 TE 中已经有 n-1 条边，所以求图 6.15(a)的无向连通网的最小生成树的过程已经完成，如图 6.16 所示。这个结果与用普里姆算法得到的结果相同。

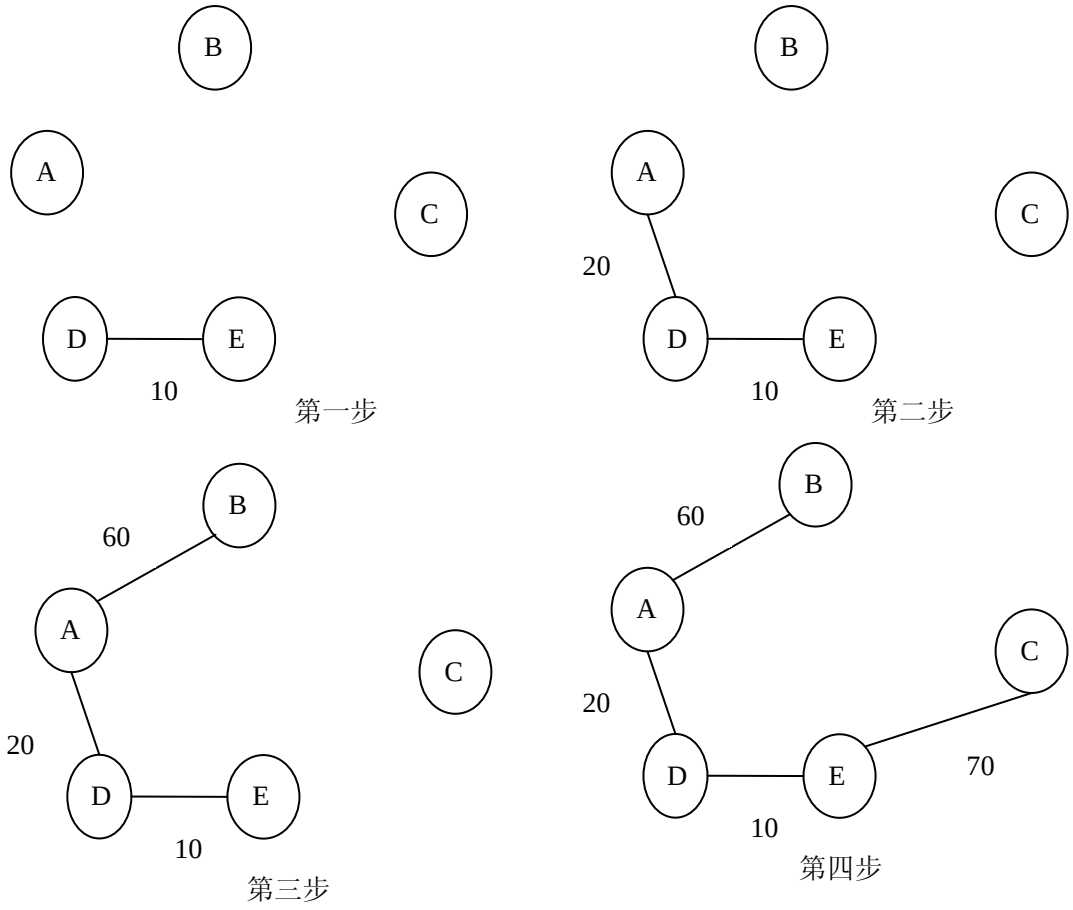


图 6.16 图 6.15(a)的无向连通网用克鲁斯卡尔算法求最小生成树的过程

6.4.2 最短路径

1、最短路径的概念

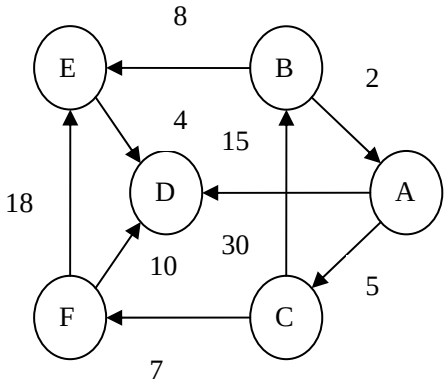
最短路径问题是图的又一个比较典型的应用问题。例如，n 个城市之间的一个公路网，给定这些城市之间的公路的距离，能否找到城市 A 到城市 B 之间一条距离最近的通路呢？如果城市用顶点表示，城市间的公路用边表示，公路的长度作为边的权值。那么，这个问题就可归结为在网中求顶点 A 到顶点 B 的所有路径中边的权值之和最小的那一条路径，这条路径就是两个顶点之间的最短路径(Shortest Path)，并称路径上的第一个顶点为源点（Source），最后一个顶点为终点（Destination）。在不带权的图中，最短路径是指两个顶点之间经历的边数最少的路径。

最短路径可以是求某个源点出发到其它顶点的最短路径，也可以是求网中任意两个顶点之间的最短路径。这里只讨论单源点的最短路径问题，感兴趣的读者可参考有关文献，了解每一对顶点之间的最短路径。

网分为无向网和有向网，当把无向网中的每一条边(vi,vj)都定义为弧<vi,vj>和弧<vj,vi>，则有向网就变成了无向网。因此，不失一般性，我们这里只讨论有向网上的最短路径问题。

图 6.17 是一个有向网及其邻接矩阵。该网从顶点 A 到顶点 D 有 4 条路径，分别是：路径（A,D），其带权路径长度为 30；路径（A,C,F,D），其带权路径长度为 22；路径（A,C,B,E,D），其带权路径长度为 32；路径（A,C,F,E,D），其带权路径长度为 34。路径（A,C,F,D）称为最短路径，其带权路径长度 22 称为最短距

离。



(a) 有向带权网

$$\begin{pmatrix} 0 & \infty & 5 & 30 & \infty & \infty \\ 2 & 0 & \infty & \infty & 8 & \infty \\ \infty & 15 & 0 & \infty & \infty & 7 \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 4 & 0 & \infty \\ \infty & \infty & \infty & 10 & 18 & 0 \end{pmatrix}$$

(b) 邻接矩阵

图 6.17 有向网及其邻接矩阵

2、狄克斯特拉（Dikastra）算法

对于求单源点的最短路径问题，狄克斯特拉（Dikastra）提出了一个按路径长度递增的顺序逐步产生最短路径的构造算法。狄克斯特拉的算法思想是：设置两个顶点的集合 S 和 T，集合 S 中存放已找到最短路径的顶点，集合 T 中存放当前还未找到最短路径的顶点。初始状态时，集合 S 中只包含源点，设为 v0，然后从集合 T 中选择到源点 v0 路径长度最短的顶点 u 加入到集合 S 中，集合 S 中每加入一个新的顶点 u 都要修改源点 v0 到集合 T 中剩余顶点的当前最短路径长度值，集合 T 中各顶点的新的最短路径长度值为原来的当前最短路径长度值与从源点过顶点 u 到达该顶点的新的最短路径长度中的较小者。此过程不断重复，直到集合 T 中的顶点全部加到集合 S 中为止。

【例 6-5】以图 6.17 为例说明用狄克斯特拉算法求有向网的从某个顶点到其余顶点最短路径的过程。

图 6.18(a)~(f)给出了狄克斯特拉算法求从顶点 A 到其余顶点的最短路径的过程。图中虚线表示当前可选择的边，实线表示算法已确定包括到集合 S 中所有顶点所对应的边。

第一步：列出顶点 A 到其余各顶点的路径长度，它们分别为 0、∞、5、30、∞、∞。从中选取路径长度最小的顶点 C（从源点到顶点 C 的最短路径为 5）。

第二步：找到顶点 C 后，再观察从源点经顶点 C 到各个顶点的路径是否比第一步所找到的路径要小（已选取的顶点则不必考虑），可发现，源点到顶点 B 的路径长度更新为 20（A，C，B），源点到顶点 F 的路径长度更新为 12（A，C，F），其余的路径则不变。然后，从已更新的路径中找出路径长度最小的顶点 F（从源点到顶点 F 的最短路径为 12）。

第三步：找到顶点 C、F 以后，再观察从源点经顶点 C、F 到各顶点的路径是否比第二步所找到的路径要小（已被选取的顶点不必考虑），可发现，源点到顶点 D 的路径长度更新为 22（A，C，F，D），源点到顶点 E 的路径长度更新为 30（A，C，F，E），其余的路径不变。然后，从已更新的路径中找出路径长短最小的顶点 D（从源点到顶点 D 的最短路径为 22）。

第四步：找到顶点 C、F、D 后，现在只剩下最后一个顶点 E 没有找到最短路径了，再观察从源点经顶点 C、F、D 到顶点 E 的路径是否比第三步所找到的

路径要小（已选取的顶点则不必考虑），可以发现，源点到顶点 E 的路径长度更新为 28（A，B，E），其余的路径则不变。然后，从已更新的路径中找出路径长度最小的顶点 E（从源点到顶点 E 的最短路径为 28）。

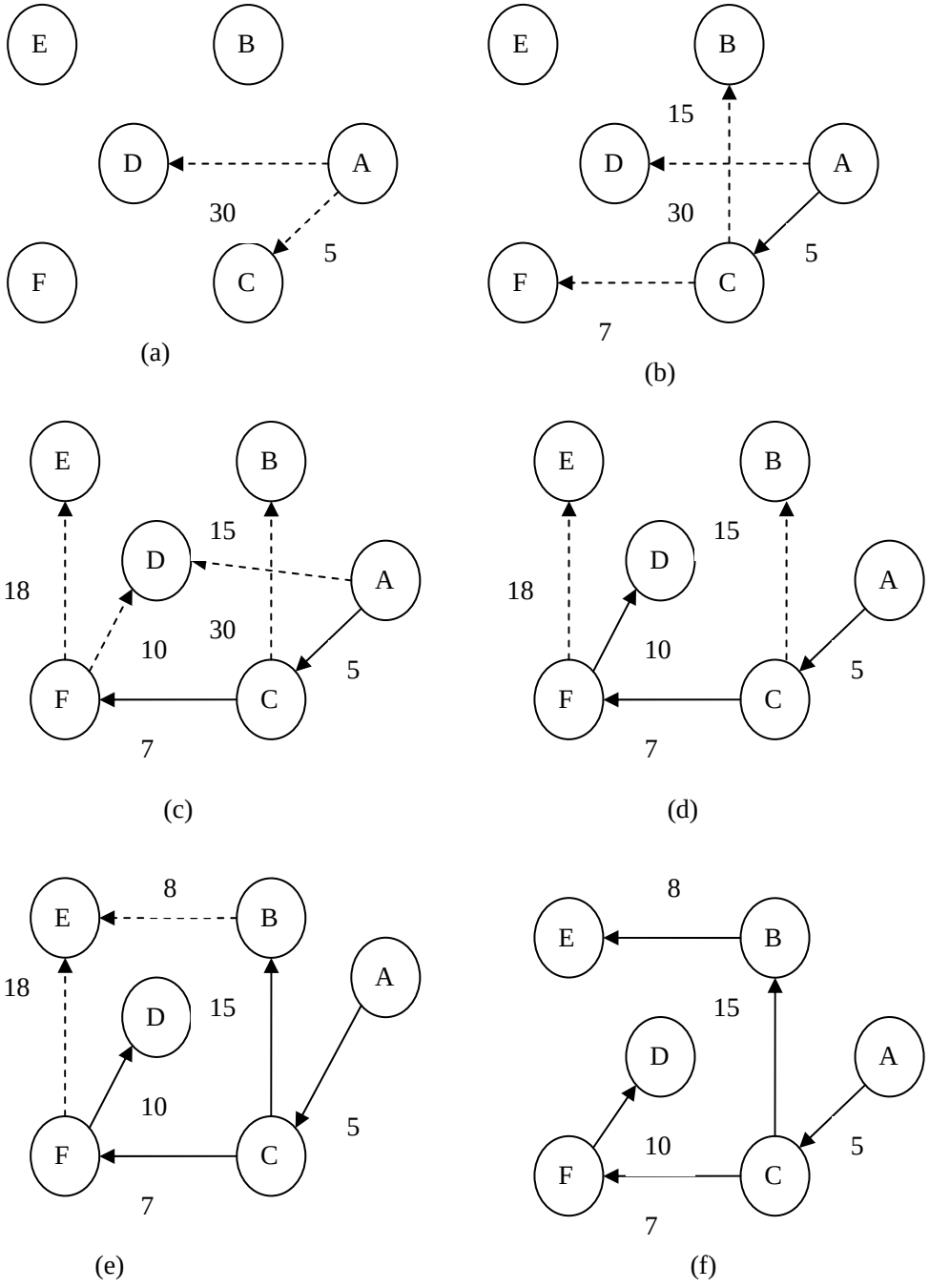


图 6.18 狄克斯特拉算法求从顶点 A 到其余顶点的最短路径的过程

3、有向网的邻接矩阵类的实现

本书以有向网的邻接矩阵类 DirecNetAdjMatrix<T>来实现狄克斯特拉算法。DirecNetAdjMatrix<T>有三个成员字段，一个是 Node<T>类型的一维数组 nodes，存放有向网中的顶点信息，一个是整型的二维数组 matirx，表示有向网的邻接矩阵，存放弧的信息，一个是整数 numArcs，表示有向网中弧的数目，有向网的邻接矩阵类 DirecNetAdjMatrix<T>的实现如下所示。

```
public class DirecNetAdjMatrix<T> : IGraph<T>
```

```
{
    private Node<T>[] nodes;       //有向网的顶点数组
    private int numArcs;           //弧的数目
    private int[,] matrix;        //邻接矩阵数组

    //构造器
    public DirecNetAdjMatrix (int n)
    {
        nodes = new Node<T>[n];
        matrix = new int[n,n];
        numArcs = 0;
    }

    //获取索引为index的顶点的信息
    public Node<T> GetNode(int index)
    {
        return nodes[index];
    }

    //设置索引为index的顶点的信息
    public void SetNode(int index, Node<T> v)
    {
        nodes[index] = v;
    }

    //弧数目属性
    public int NumArcs
    {
        get
        {
            return numArcs;
        }
        set
        {
            numArcs = value;
        }
    }

    //获取matrix[index1, index2]的值
    public int GetMatrix(int index1, int index2)
    {
        return matrix[index1, index2];
    }
```

```
//设置matrix[index1, index2]的值
public void SetMatrix(int index1, int index2, int v)
{
    matrix[index1, index2] = v;
}

//获取顶点数目
public int GetNumOfVertex()
{
    return nodes.Length;
}

//获取弧的数目
public int GetNumOfEdge()
{
    return numArcs;
}

//判断v是否是网的顶点
public bool IsNode(Node<T> v)
{
    //遍历顶点数组
    foreach (Node<T> nd in nodes)
    {
        //如果顶点nd与v相等，则v是图的顶点，返回true
        if (v.Equals(nd))
        {
            return true;
        }
    }

    return false;
}

//获取v在顶点数组中的索引
public int GetIndex(Node<T> v)
{
    int i = -1;

    //遍历顶点数组
    for (i = 0; i < nodes.Length; ++i)
    {
        //如果顶点nd与v相等，则v是图的顶点，返回索引值
        if (nodes[i].Equals(v))
```

```
            {
                return i;
            }
        }
        return i;
    }

    //在v1和v2之间添加权为v的弧
    public void SetEdge(Node<T> v1, Node<T> v2, int v)
    {
        //v1或v2不是网的顶点
        if (!IsNode(v1) || !IsNode(v2))
        {
            Console.WriteLine("Node is not belong to Graph!");
            return;
        }

        matrix[GetIndex(v1), GetIndex(v2)] = v;
        ++numArcs;
    }

    //删除v1和v2之间的弧
    public void DelEdge(Node<T> v1, Node<T> v2)
    {
        //v1或v2不是网的顶点
        if (!IsNode(v1) || !IsNode(v2))
        {
            Console.WriteLine("Node is not belong to Graph!");
            return;
        }

        //v1和v2之间存在弧
        if (matrix[GetIndex(v1), GetIndex(v2)] != int.MaxValue)
        {
            matrix[GetIndex(v1), GetIndex(v2)] = int.MaxValue;
            --numArcs;
        }
    }

    //判断v1和v2之间是否存在弧
    public bool IsEdge(Node<T> v1, Node<T> v2)
    {
        //v1或v2不是网的顶点
        if (!IsNode(v1) || !IsNode(v2))
```

```
            {
                Console.WriteLine("Node is not belong to Graph!");
                return false;
            }

            //v1和v2之间存在弧
            if (matrix[GetIndex(v1), GetIndex(v2)] != int.MaxValue)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
```

4、狄克斯特拉算法的实现

为实现狄克斯特拉算法，引入两个数组，一个一维数组 ShortPathArr，用来保存从源点到各个顶点的最短路径的长度，一个二维数组 PathMatrixArr，用来保存从源点到某个顶点的最短路径上的顶点，如 PathMatrix[v][w]为 true，则 w 为从源点到顶点 v 的最短路径上的顶点。为了该算法的结果被其他算法使用，把这两个数组作为算法的参数使用。另外，为了表示某顶点的最短路径是否已经找到，在算法中设了一个一维数组 final,如果 final[i]为 true,则表示已经找到第 i 个顶点的最短路径。i 是该顶点在邻接矩阵中的序号。同样，把该算法作为类 DirecNetAdjMatrix<T>的成员方法来实现。

```
public void Dijkstra(ref bool[,] pathMatricArr,
                     ref int[] shortPathArr, Node<T> n)
{
     int k = 0;
     bool[] final = new bool[nodes.Length];

     //初始化
     for (int i = 0; i < nodes.Length; ++i)
     {
          final[i] = false;
          shortPathArr[i] = matrix[GetIndex(n),i];

          for (int j = 0; j < nodes.Length; ++j)
          {
               pathMatricArr[i,j] = false;
          }
          if (shortPathArr[i] != 0 && shortPathArr[i] < int.MaxValue)
          {
               pathMatricArr[i,GetIndex(n)] = true;
```

```
                pathMatricArr[i,i] = true;
            }
        }

        // n为源点
        shortPathArr[GetIndex(n)] = 0;
        final[GetIndex(n)] = true;

        //处理从源点到其余顶点的最短路径
        for (int i = 0; i < nodes.Length; ++i)
        {
            int min = int.MaxValue;

            //比较从源点到其余顶点的路径长度
            for (int j = 0; j < nodes.Length; ++j)
            {
                //从源点到j顶点的最短路径还没有找到
                if (!final[j])
                {
                    /从源点到j顶点的路径长度最小
                    if (shortPathArr[j] < min)
                    {
                        k = j;
                        min = shortPathArr[j];
                    }
                }
            }

            //源点到顶点k的路径长度最小
            final[k] = true;

            //更新当前最短路径及距离
            for (int j = 0; j < nodes.Length; ++j)
            {
                if (!final[j] && (min + matrix[k,j] < shortPathArr[j]))
                {
                    shortPathArr[j] = min + matrix[k,j];
                    for (int w = 0; w < nodes.Length; ++w)
                    {
                        pathMatricArr[j,w] = pathMatricArr[k,w];
                    }
                    pathMatricArr[j,j] = true;
                }
            }
```

```
        }
    }
```

6.4.3 拓扑排序

拓扑排序(Topological Sort)是图中重要的运算之一，在实际中应用很广泛。例如，很多工程都可分为若干个具有独立性的子工程，我们把这些子工程称为“活动”。每个活动之间有时存在一定的先决条件关系，即在时间上有着一定的相互制约的关系。也就是说，有些活动必须在其它活动完成之后才能开始，即某项活动的开始必须以另一项活动的完成为前提。在有向图中，若以图中的顶点表示活动，以弧表示活动之间的优先关系，这样的有向图称为 AOV 网（Active On Vertex Network）。

在 AOV 网中，若从顶点 vi 到顶点 vj 之间存在一条有向路径，则称 vi 是 vj 的前驱，vj 是 vi 的后继。若<vi,vj>是 AOV 网中的弧，则称 vi 是 vj 的直接前驱，vj 是 vi 的直接后继。

例如，一个软件专业的学生必须学习一系列的基本课程（如表 6.2 所示）。其中，有些课程是基础课，如“高等数学”、“程序设计基础”，这些课程不需要先修课程，而另一些课程必须在先学完某些课程之后才能开始学习。如通常在学完“程序设计基础”和“离散数学”之后才开始学习“数据结构”等等。因此，可以用 AOV 网来表示各课程及其之间的关系，如图 6.19 所示。

表 6.2 软件专业必修课程

课程编号	课程名称	先决条件
c1	程序设计基础	无
c2	离散数学	c1
c3	数据结构	c1,c2
c4	汇编语言	c1
c5	语言的设计与实现	c3,c4
c6	计算机原理	c11
c7	编译原理	c3,c5
c8	操作系统	c3,c6
c9	高等数学	无
c10	线性代数	c9
c11	普通物理	c9
c12	数值分析	c9,c10,c11

在 AOV 网中，不应该出现有向环路，因为有环意味着某项活动以自己作为先决条件，这样就进入了死循环。如果图 6.19 的有向图出现了有向环路，则教学计划将无法编排。因此，对给定的 AOV 网应首先判定网中是否存在环。检测的办法是对有向图进行拓扑排序（Topological Sort），若网中所有顶点都在它的拓扑有序序列中，则 AOV 网中必定不存在环。

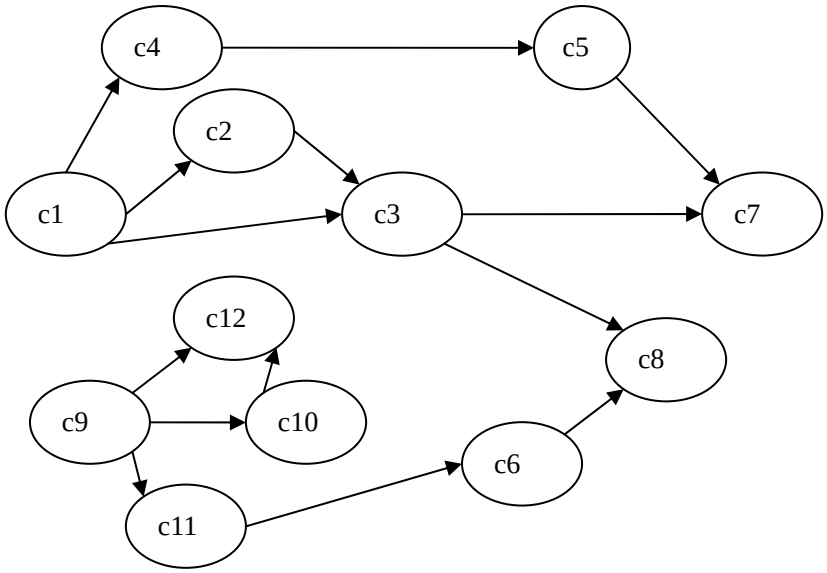


图 6.19 表示课程之间关系的有向图

实现一个有向图的拓扑有序序列的过程称为拓扑排序。可以证明，任何一个有向无环图，其全部顶点都可以排成一个拓扑序列，而其拓扑有序序列不一定是唯一的。例如，图 6.19 的有向图的拓扑有序序列有多个，这里列举两个如下：

(c1,c2,c3,c4,c5,c7,c8,c9,c10,c11,c6,c12,c8)

和 (c9,c10,c11,c6,c1,c12,c4,c2,c3,c5,c7,c8)

由上面两个序列可知，对于图中没有弧相连的两个顶点，它们在拓扑排序的序列中出现的次序没有要求。例如，第一个序列中 c1 先于 c9，第二个则反之。拓扑排序的任何一个序列都是一个可行的活动执行顺序，它可以检测到图中是否存在环，因为如果有环，则环中的顶点无法输出，所以得到的拓扑有序序列没有包含图中所有的顶点。

下面是拓扑排序算法的描述：

（1）在有向图中选择一个入度为 0 的顶点（即没有前驱的顶点），由于该顶点没有任何先决条件，输出该顶点；

（2）从图中删除所有以它为尾的弧；

（3）重复执行（1）和（2），直到找不到入度为 0 的顶点，拓扑排序完成。

如果图中仍有顶点存在，却没有入度为 0 的顶点，说明 AOV 网中有环路，否则没有环路。

【例 6-6】以图 6.20(a)为例求出它的一个拓扑有序序列。

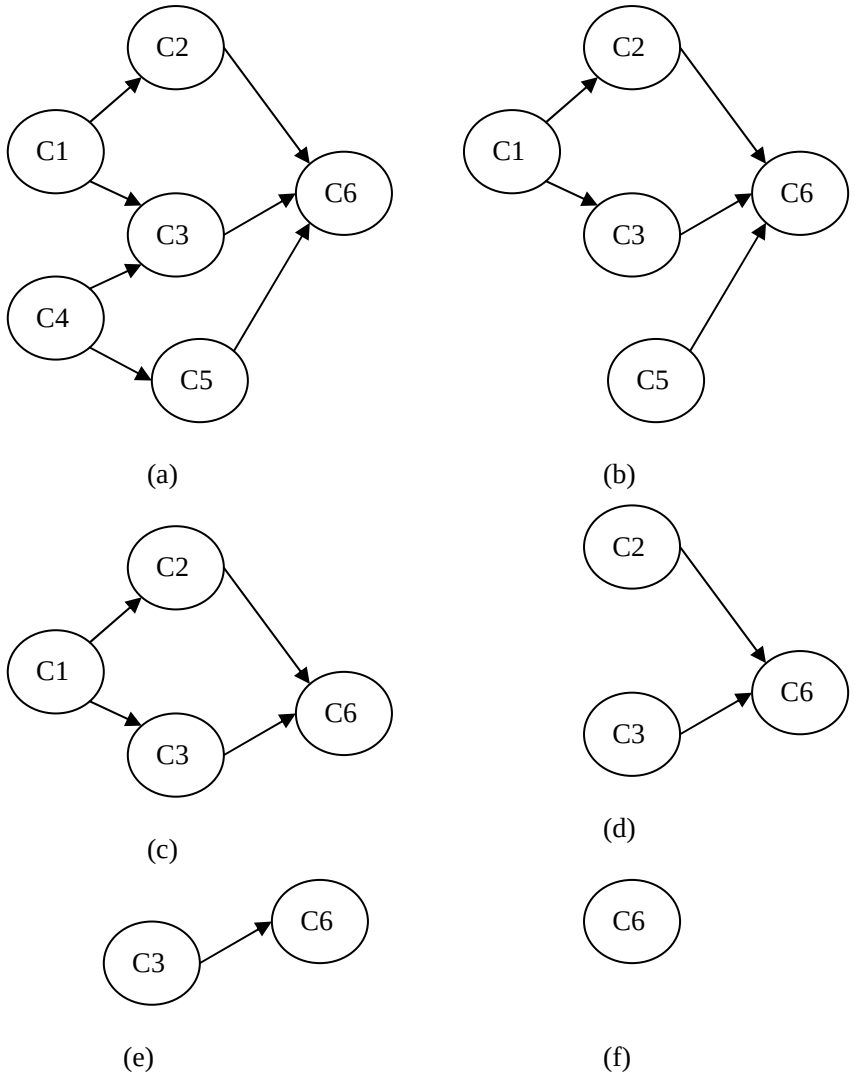


图 6.20 拓扑排序过程

第一步：在图 6.20(a)所示的有向图中选取入度为 0 的顶点 c4，删除顶点 c4 及与它相关联的弧<c4,c3>，<c4,c5>，得到图 6.31(b)所示的结果，并得到第一个拓扑有序序列顶点 c4。

第二步：再在图 6.20(b)中选取入度为 0 的顶点 c5，删除顶点 c5 及与它相关联的弧<c5,c6>，得到图 6.20(c)所示的结果，并得到两个拓扑有序序列顶点 c4，c5。

第三步：再在图 6.20(c)中选取入度为 0 的顶点 c1，删除顶点 c1 及与它相关联的弧<c1,c2>，<c1,c3>，得到图 6.20(d)所示的结果，并得到三个拓扑有序序列顶点 c4，c5，c1。

第四步：再在图 6.20(d)中选取入度为 0 的顶点 c2，删除顶点 c2 及与它相关联的弧<c2,c6>，得到图 6.20(e)所示的结果，并得到四个拓扑有序序列顶点 c4，c5，c1，c2。

第五步：再在图 6.20(e)中选取入度为 0 的顶点 c3，删除顶点 c3 及与它相关联的弧<c3,c6>，得到图 6.20(f)所示的结果，并得到五个拓扑有序序列顶点 c4，c5，c1，c2，c3。

第六步：最后选取仅剩下的最后一个顶点 c6，拓扑排序结束，得到图 6.20(a)

的一个拓扑有序序列（c4，c5，c1，c2，c3，c6）。

本章小结

图是另一种比树形结构更复杂的非线性数据结构，图中的数据元素称为顶点，顶点之间是多对多的关系。图分为有向图和无向图，带权值的图称为网。

图的存储结构很多，一般采用数组存储图中顶点的信息，邻接矩阵采用矩阵也就是二维数组存储顶点之间的关系。无向图的邻接矩阵是对称的，所以在存储时可以只存储上三角矩阵或下三角矩阵的数据；有向图的邻接矩阵不是对称的。邻接表用一个链表来存储顶点之间的关系，所以邻接表是顺序存储与链式存储相结合的存储结构。

图的遍历方法有两种：深度优先遍历和广度优先遍历。图的深度优先遍历类似于树的先序遍历，是树的先序遍历的推广，它访问顶点的顺序是后进先出，与栈一样。图的广度优先遍历类似于树的层序遍历，它访问顶点的顺序是先进先出，与队列一样。

图的应用很广，本章重点介绍了三个方面的应用。最小生成树是一个无向连通网中边的权值总和最小的生成树。构造最小生成树必须包括 n 个顶点、n-1 条边及不存在回路。构造最小生成树的常用算法有普里姆算法和克鲁斯卡尔算法两种。

最短路径问题是图的一个比较典型的应用问题。最短路径是网中求一个顶点到另一个顶点的所有路径中边的权值之和最小的路径。可以求从一个顶点到网中其余顶点的最短路径，这称之为单源点问题，也可以求网中任意两个顶点之间的最短路径。本章只讨论了单源点问题。解决单源点问题的算法是狄克斯特拉算法。

拓扑排序是图中重要的运算之一，在实际中应用很广泛。AOV 网是顶点之间存在优先关系的有向图。拓扑排序是解决 AOV 网中是否存在环路的有效手段，若拓扑有序序列中包含 AOV 网中所有的顶点，则 AOV 网中不存在环路，否则存在环路。

习题六

6.1 画出无向图 6.21 的邻接矩阵和邻接表的示意图，并写出每个顶点的度。

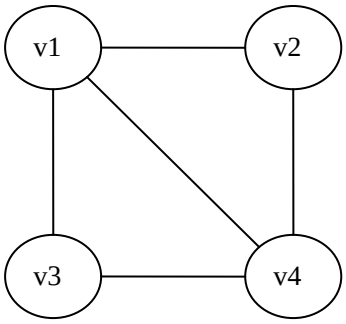


图 6.21

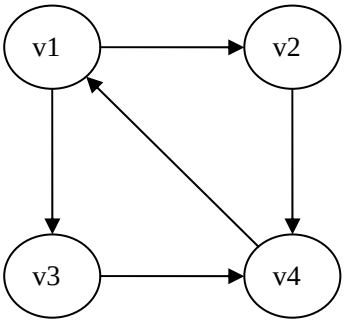


图 6.22

6.2 画出有向图 6.22 的邻接矩阵和邻接表的示意图，并写出每个顶点的入度和出度。

6.3 写出有向图的邻接矩阵类的实现。

6.4 写出有向图的邻接表类的实现。

6.5 写出无向网的邻接表类的实现。

6.6 写出有向网的邻接表类的实现。

6.7 对应图 6.23，写出从顶点 v1 出发进行深度优先遍历和广度优先遍历得到的顶点序列。

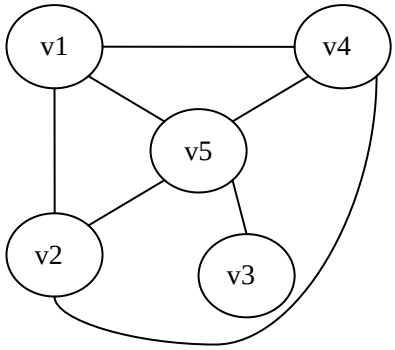


图 6.23

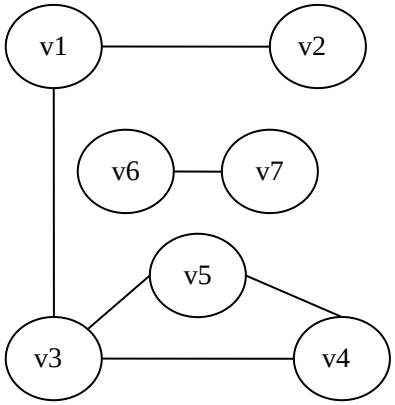


图 6.24

6.8 求图 6.24 的连通分量。

6.9 对应图 6.25，分别用普里姆算法和克鲁斯卡尔算法画出得到最小生成树的过程。

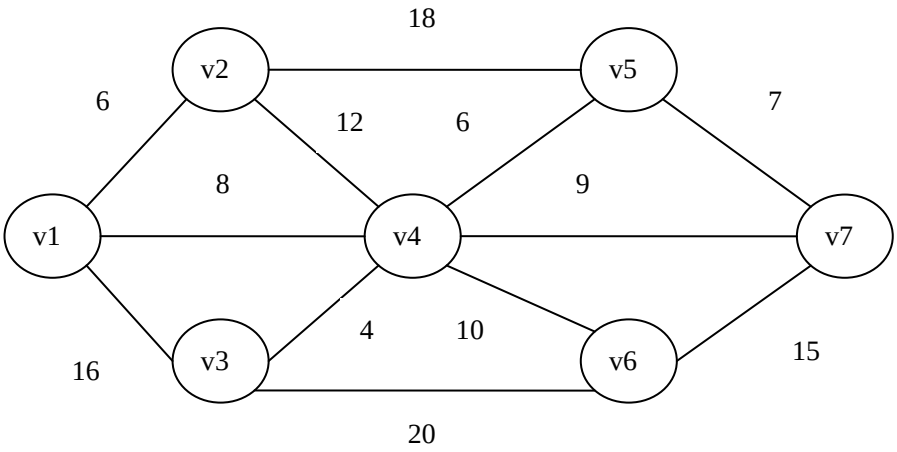


图 6.25

6.10 对于图 6.26，利用狄克斯特拉算法求从顶点 v1 到其余各顶点的最短路径。

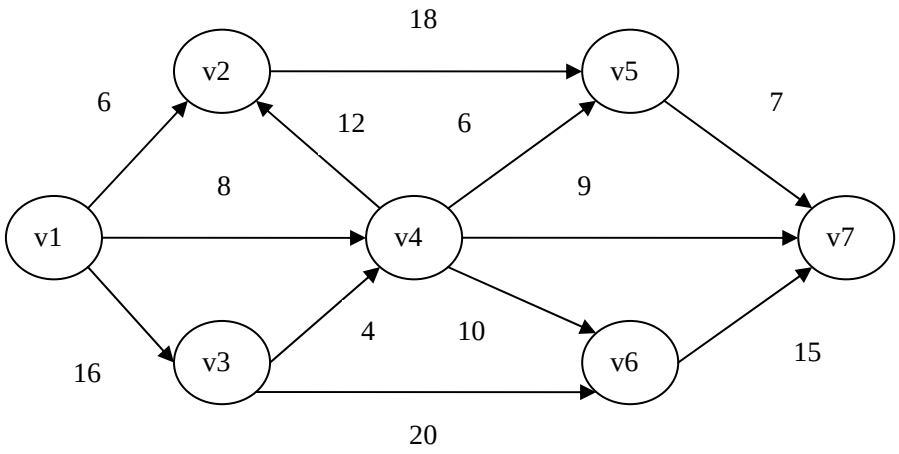


图 6.26

6.11 写出图 6.27 的拓扑排序。

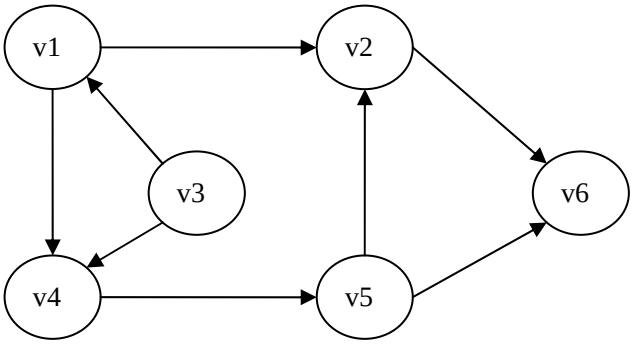


图 6.27

第7章 排序

排序（Sort）是计算机程序设计中的一种重要操作，也是日常生活中经常遇到的问题。例如，字典中的单词是以字母的顺序排列，否则，使用起来非常困难。同样，存储在计算机中的数据的次序，对于处理这些数据的算法的速度和简便性而言，也具有非常深远的意义。

7.1 基本概念

排序是把一个记录（在排序中把数据元素称为记录）集合或序列重新排列成按记录的某个数据项值递增（或递减）的序列。

表 7-1 是一个学生成绩表，其中某个学生记录包括学号、姓名及计算机文化基础、C 语言、数据结构等课程的成绩和总成绩等数据项。在排序时，如果用总成绩来排序，则会得到一个有序序列；如果以数据结构成绩进行排序，则会得到另一个有序序列。

表 7-1 学生成绩表

学号	姓名	计算机文化基础	C 语言	数据结构	总成绩
04103101	张三	85	92	86	263
04103102	李四	90	91	93	274
04103103	王五	66	63	64	193
04103104	何六	75	74	73	222
…	…	…	…	…	…

作为排序依据的数据项称为“排序项”，也称为记录的关键码(Keyword)。关键码分为主关键码(Primary Keyword)和次关键码(Secondary Keyword)。一般地，若关键码是主关键码，则对于任意待排序的序列，经排序后得到的结果是唯一的；若关键码是次关键码，排序的结果不一定唯一，这是因为待排序的序列中可能存在具有相同关键码值的记录。此时，这些记录在排序结果中，它们之间的位置关系与排序前不一定保持一致。如果使用某个排序方法对任意的记录序列按关键码进行排序，相同关键码值的记录之间的位置关系与排序前一致，则称此排序方法是稳定的；如果不一致，则称此排序方法是不稳定的。

例如，一个记录的关键码序列为（31，2，15，7，91，7*），可以看出，关键码为 7 的记录有两个（第二个加“*”号以区别，以下同）。若采用一种排序方法得到的结果序列为（2，7，7*，15，31，91），则该排序方法是稳定的；若采用另外一种排序方法得到的结果序列为（1，7*，7，15，31，91），则这种排序方法是不稳定的。

由于待排序的记录的数量不同，使得排序过程中涉及的存储器不同，可将排序方法分为内部排序（Internal Sorting）和外部排序（External Sorting）两大类。

内部排序指的是在排序的整个过程中，记录全部存放在计算机的内存中，并且在内存中调整记录之间的相对位置，在此期间没有进行内、外存的数据交换。

外部排序指的是在排序过程中，记录的主要部分存放在外存中，借助于内存逐步调整记录之间的相对位置。在这个过程中，需要不断地在内、外存之间交换数据。

显然，内部排序适用于记录不多的文件。而对于一些较大的文件，由于内存容量的限制，不能一次全部装入内存进行排序，此时采用外部排序较为合适。但

外部排序的速度比内部排序要慢的多。内部排序和外部排序各有许多不同的排序方法。本书只讨论内部排序的各种方法。

排序问题的记录采用线性结构，同时，允许存取任意位置的记录，这和第 2 章讨论的线性表完全吻合。所以，排序问题的数据结构是线性表。

任何算法的实现都和算法所处理的数据元素的存储结构有关。线性表的两种典型存储结构是顺序表和链表。由于顺序表具有随机存取的特性，存取任意一个数据元素的时间复杂度为 O(1)，而链表不具有随机存取特性，存取任意一个数据元素的时间复杂度为 O(n)，所以，排序算法基本上是基于顺序表而设计的。

由于排序是以记录的某个数据项为关键码进行排序的，所以，为了讨论问题的方便，假设顺序表中只存放记录的关键码，并且关键码的数据类型是整型，也就是说，本章使用的顺序表是整型的顺序表 SeqList<int>，下面讨论各种排序方法简写为 SeqList。

排序有非递增有序和非递减排序排序两种。不失一般性，本章讨论的所有排序算法都是按关键码非递减有序设计的。

7.2 简单排序方法

7.2.1 直接插入排序

直接插入排序(direct Insert Sort)的基本思想是：顺序地将待排序的记录按其关键码的大小插入到已排序的记录子序列的适当位置。子序列的记录个数从 1 开始逐渐增大，当子序列的记录个数与顺序表中的记录个数相同时排序完毕。

设待排序的顺序表 sqList 中有 n 个记录，初始时子序列中只有一个记录 sqList[0]。第一次排序时，准备把记录 sqList[1]插入到已排好序的子序列中，这时只需要比较 sqList[0]和 sqList[1]的大小，若 sqList[0]≤sqList[1]，说明序列已有序，否则将 sqList[1]插入到 sqList[0]的前面，这样子序列的大小增大为 2。第二次排序时，准备把记录 sqList[2]插入到已排好序的子序列中，这需要先比较 sqList[2] 和 sqList[1]以确定是否需要把 sqList[2]插入到 sqList[1]之前。如果 sqList[2]插入到 sqList[1]之前，再比较 sqList[2]和 sqList[0]以确定是否需要把 sqList[2]插入到 sqList[0]之前。这样的过程一直进行到 sqList[n-1]插入到子序列中为止。这时，顺序表 sqList 就是有序的。

直接插入排序的算法如下所示，算法中记录的比较表示记录关键码的比较，顺序表中只存放了记录的关键码：

```
public void InsertSort(SeqList<int> sqList)
{
      for (int i = 1; i < sqList.Last; ++i)
      {
           if (sqList[i] < sqList[i - 1])
           {
                int tmp = sqList[i];
                int j = 0;
                for (j = i - 1; j >= 0&&tmp<sqList[j]; --j)
                {
                    sqList[j + 1] = sqList[j];
                }
                sqList[j + 1] = tmp;
           }
```

```
            }
        }
```

直接插入排序算法的时间复杂度分为最好、最坏和随机三种情况：

（1） 最好的情况是顺序表中的记录已全部排好序。这时外层循环的次数为n-1，内层循环的次数为0。这样，外层循环中每次记录的比较次数为1，所以直接插入排序算法在最好情况下的时间复杂度为0(n)。

（2） 最坏情况是顺序表中记录是反序的。这时内层循环的循环系数每次均为i。这样，整个外层循环的比较次数为

$$\sum_{i=1}^{n-1}(i+1)=\frac{(n-1)(n+1)}{2}$$

因此，直接插入排序算法在最坏情况下的时间复杂度为$0(n^2)$。

（3） 如果顺序表中的记录的排列是随机的，则记录的期望比较次数为$n^2/4$。因此，直接插入排序算法在一般情况下的时间复杂度为$0(n^2)$。

可以证明，顺序表中的记录越接近于有序，直接插入排序算法的时间效率越高，其时间效率在0(n)到$0(n^2)$之间。

直接插入排序算法的空间复杂度为 0(1)。因此，直接插入排序算法是一种稳定的排序算法。

【例 7-1】关键码序列为（42，20，17，27，13，8，17*，48），用直接插入排序算法进行排序。

排序过程如图 7.1 所示。

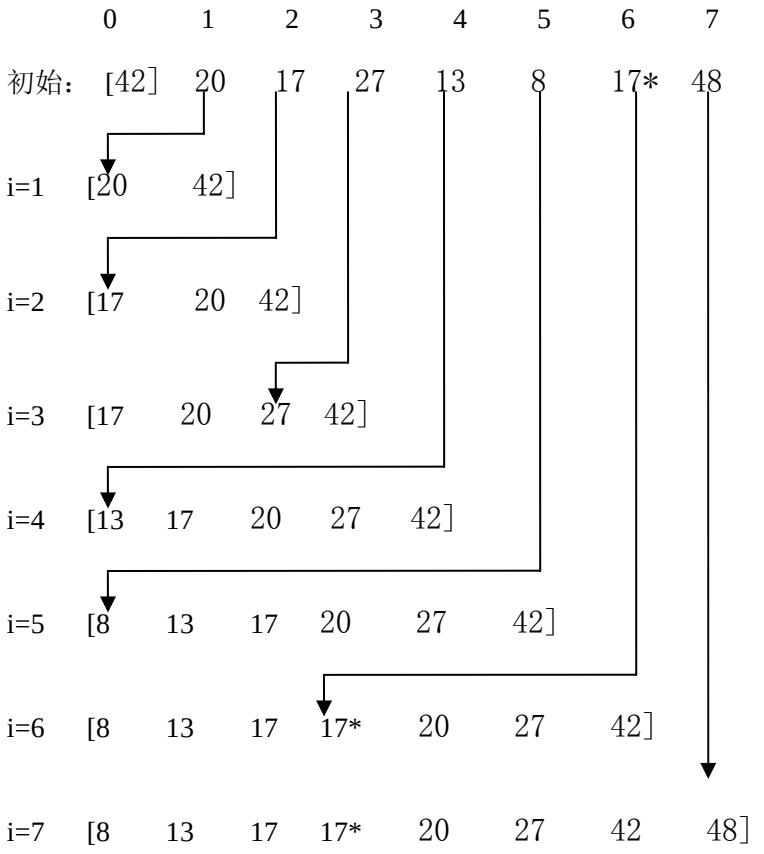


图 7.1 直接插入排序示例图

7.2.2 冒泡排序

冒泡排序(Bubble Sort)的基本思想是：将相邻的记录的关键码进行比较，若前面记录的关键码大于后面记录的关键码，则将它们交换，否则不交换。

设待排序的顺序表 sqList 中有 n 个记录，冒泡排序要进行 n-1 趟，每趟循环均是从最后两个记录开始。第 1 趟循环到第 2 个记录的关键码与第 1 个记录的关键码比较后终止，第 2 趟循环到第 3 个记录的关键码与第 2 个记录的关键码比较结束后终止。一般地，第 i 趟循环到第 i+1 个记录的关键码与第 i 个记录的关键码比较后终止，所以，第 n-1 趟循环到第 n 个记录的关键码与第 n-1 个记录的关键码比较后终止。

冒泡排序算法的实现如下所示，算法中记录的比较表示记录关键码的比较，顺序表中只存放了记录的关键码：

```
public void BubbleSort(SeqList<int> sqList)
{
    int tmp;
    for (int i = 0; i < sqList.Last; ++i)
    {
        for (int j = sqList.Last - 1; j >= i; --j)
        {
            if (sqList[j + 1] < sqList[j])
            {
                tmp = sqList[j + 1];
                sqList[j + 1] = sqList[j];
                sqList[j] = tmp;
            }
        }
    }
}
```

冒泡排序算法的最好情况是记录已全部排好序，这时，循环 n-1 次，每次循环都因没有数据交换而退出。因此，冒泡排序算法在最好情况下的时间复杂度为 O(n)。冒泡排序算法的最坏情况是记录全部逆序存放，这时，循环 n-1 次，总比较次数为：

$$\sum_{i=0}^{n-2}(n-1-i)=\frac{n(n-1)}{2}$$

总的移动次数为比较次数的 3 倍，因为被进行一次比较，需要进行 3 次移动。因此，冒泡排序算法在最坏情况下的时间复杂度为$O(n^2)$。

冒泡排序算法只需要一个辅助空间用于交换记录，所以，冒泡排序算法是一种稳定的排序方法。

【例 7-2】关键码序列为（42，20，17，27，13，8，17*，48），用冒泡排序算法进行排序。

排序过程如图 7.2 所示。

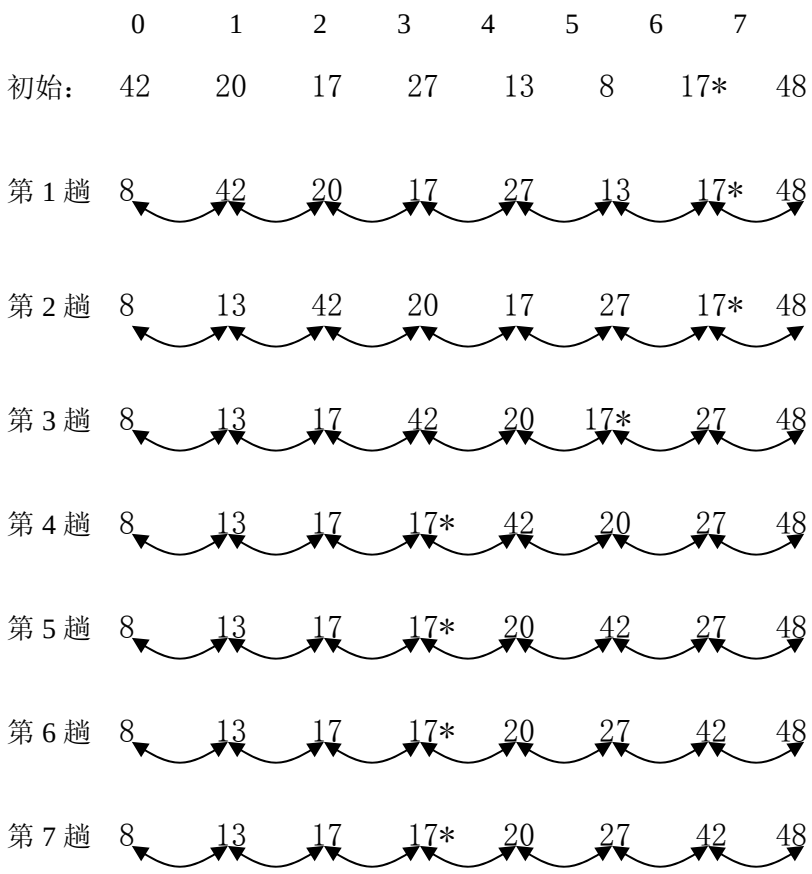


图 7.2 冒泡排序示例图

7.2.3 简单选择排序

简单选择排序(Simple Select Sort)算法的基本思想是：从待排序的记录序列中选择关键码最小（或最大）的记录并将它与序列中的第一个记录交换位置；然后从不包括第一个位置上的记录序列中选择关键码最小（或最大）的记录并将它与序列中的第二个记录交换位置；如此重复，直到序列中只剩下一个记录为止。

设待排序的顺序表 sqList 中有 n 个记录，简单选择排序要进行 n-1 趟，第 1 趟从 n 个记录选择关键码最小（或最大）的记录并与第 1 个记录交换位置；第 2 趟从第 2 个记录开始的 n-1 个记录中选择关键码最小（或最大）的记录并与第 2 个记录交换位置。一般地，第 i 趟从第 i 个记录开始的 n-i+1 个记录中选择关键码最小（或最大）的记录并与第 i 个记录交换位置，所以，第 n-1 趟比较最后两个记录选择关键码最小（或最大）的记录并与第 n-1 个记录交换位置。

简单选择排序算法的实现如下所示，算法中记录的比较表示记录关键码的比较，顺序表中只存放了记录的关键码：

```
public void SimpleSelectSort(SeqList<int> sqList)
{
    int tmp = 0;
    int t = 0;
    for (int i = 0; i < sqList.Last; ++i)
    {
        t = i;
```

```
            for (int j = i + 1; j <= sqList.Last; ++j)
            {
                  if (sqList[t] > sqList[j])
                  {
                      t = j;
                  }
              }
              tmp = sqList[i];
              sqList[i] = sqList[t];
              sqList[t] = tmp;
        }
    }
```

在简单选择排序中，第一次排序要进行 n-1 次比较，第二次排序要进行 n-2 次比较，第 n-1 排序要进行 1 次比较，所以总的比较次数为：

$$\sum_{i=0}^{n-2}(n-1-i)=\frac{n(n-1)}{2}$$

在各次排序时，记录的移动次数最好 0 次，最坏为 3 次，所以，总的移动次数最好为 0 次，最坏为 3 次。因此，简单选择排序算法的时间复杂度为$O(n^2)$。

简单选择排序算法只需要一个辅助空间用于交换记录，所以，简单选择排序算法是一种稳定的排序方法。

【例 7-3】关键码序列为（42，20，17，27，13，8，17*，48），用简单选择排序算法进行排序。

排序过程如图 7.3 所示。

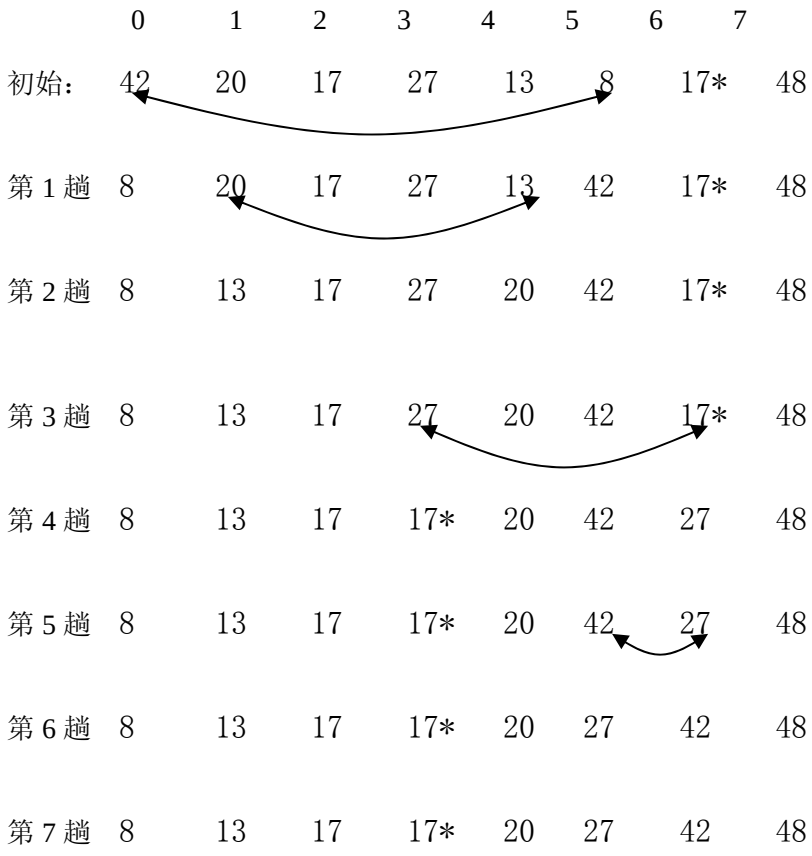


图 7.3 简单选择排序示例图

7.3 快速排序

快速排序(Quick Sort)的基本思想是：通过不断比较关键码，以某个记录为界（该记录称为支点），将待排序列分成两部分。其中，一部分满足所有记录的关键码都大于或等于支点记录的关键码，另一部分记录的关键码都小于支点记录的关键码。把以支点记录为界将待排序列按关键码分成两部分的过程，称为一次划分。对各部分不断划分，直到整个序列按关键码有序为止。

设待排序的顺序表 sqList 中有 n 个记录，一般地，第一次划分把第一个记录作为支点。首先，把支点复制到一个临时存储空间中，并设两个指示器，一个指示器 low,指向顺序表的低端（第一个记录所在位置），一个指示器 high，指向顺序表的高端（最后一个记录所在位置）。然后，从 high 所指向的记录开始，将记录的关键码与支点（在临时存储空间中）的关键码进行比较，如果 high 所指向的记录的关键码大于支点的关键码，high 指示器向顺序表的低端方向移动一个记录的位置，否则，将 high 所指的记录复制到 low 所指的存储空间中。接着，又将 low 移到下一个记录，从 low 所指向的记录开始，将记录的关键码与临时存储空间的记录的关键码进行比较，如果 low 所指向的记录的关键码小于临时存储空间的记录的关键码，low 指示器向顺序表的高端方法移动一个记录的位置，否则，将 low 所指的记录复制到 high 所指的存储空间中，high 指示器向顺序表的低端移动一个记录的位置。如此重复，直到 low 和 high 指示器指向同一个记录，将临时空间的记录赋给 low 所指向的存储空间，第一次划分结束。

快速排序的算法实现如下所示，算法中记录的比较表示记录关键码的比较，

顺序表中只存放了记录的关键码：

```
    public void QuickSort(SeqList<int> sqList, int low, int high)
    {
          int i = low;
          int j = high;
          int tmp = sqList[low];
          while (low < high)
          {
               while ((low < high) && (sqList[high] >= tmp))
               {
                    --high;
               }
               sqList[low] = sqList[high];
               ++low;
               while ((low < high) && (sqList[low] <= tmp))
               {
                    ++low;
               }
               sqList[high] = sqList[low];
               --high;
           }
           sqList[low] = tmp;

           if (i < low-1)
           {
                QuickSort(sqList, i, low-1);
           }
           if (low+1 < j)
           {
                QuickSort(sqList, low+1, j);
           }
       }
   }
```

【例 7-4】关键码序列为（42，20，17，27，13，8，17*，48），写出用快速排序算法进行排序的过程。

快速排序过程如下所示。

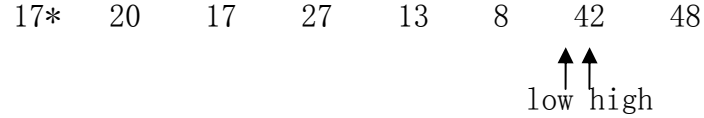
```
          0     1     2     3     4     5     6     7
初始：   42    20    17    27    13    8    17*   48
         ↑                                       ↑
        low                                     high
```

第一次划分：

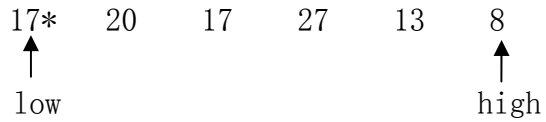
第 1 次查找交换，得到如下结果：

```
        17*   20    17    27    13    8    17*    48
```

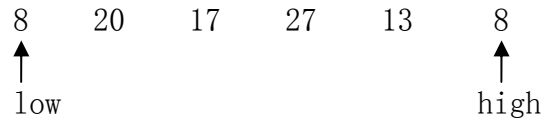
第 2 次查找交换，得到如下结果，第一次划分结束



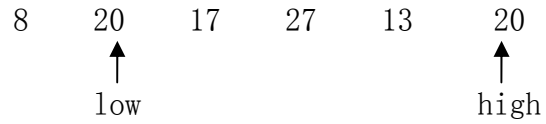
第二次划分：



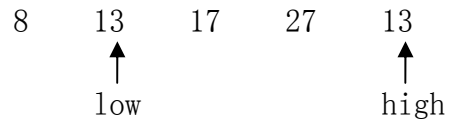
第 1 次查找交换，得到如下结果：



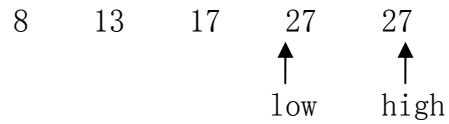
第 2 次查找交换，得到如下结果：



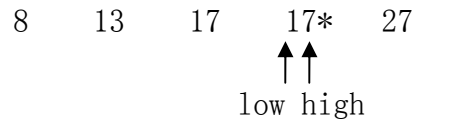
第 3 次查找交换，得到如下结果：



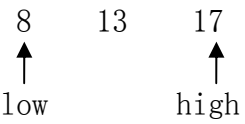
第 4 次查找交换，得到如下结果：



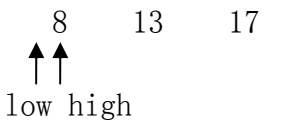
第 5 次查找交换，得到如下结果,第二次划分结束。



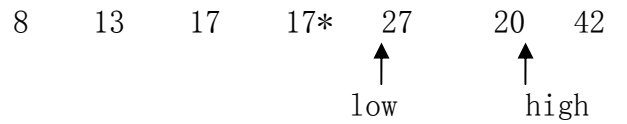
第三次划分：



第 1 次查找交换，得到如下结果，第三次划分结束。

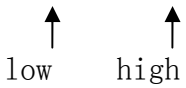


第四次划分：



第 1 次查找交换，得到如下结果：

8 13 17 17* 20 27 42 48



第 2 次查找交换，得到如下结果，第四次划分结束。

8 13 17 17* 20 27 42 48

快速排序算法的时间复杂度和每次划分的记录关系很大。如果每次选取的记录都能均分成两个相等的子序列，这样的快速排序过程是一棵完全二叉树结构（即每个结点都把当前待排序列分成两个大小相当的子序列结点，n个记录待排序列的根结点的分解次数就构成了一棵完全二叉树），这时分解次数等于完全二叉树的深度$\log_2 n$。每次快速排序过程无论把待排序列这样划分，全部的比较次数都接近于n-1 次，所以，最好情况下快速排序的时间复杂度为0（$n\log_2 n$）。快速排序算法的最坏情况是记录已全部有序，此时n个记录待排序列的根结点的分解次数就构成了一棵单右支二叉树。所以在最坏情况下快速排序算法的时间复杂度为$0(n^2)$。一般情况下，记录的分布是随机的，序列的分解次数构成一棵二叉树，这样二叉树的深度接近于$\log_2 n$，所以快速排序算法在一般情况下的时间复杂度为0（$n\log_2 n$）。

另外，快速排序算法是一种不稳定的排序的方法。

7.4 堆排序

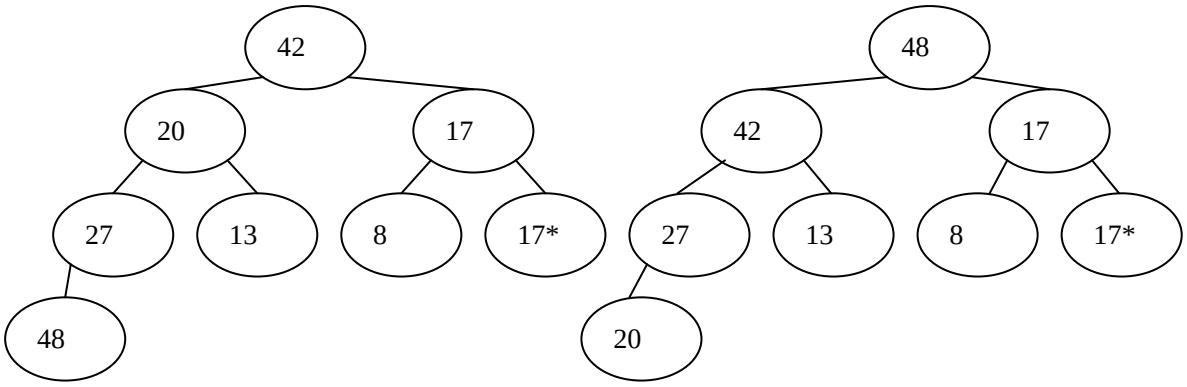
在直接选择排序中，顺序表是一个线性结构，要从有n个记录的顺序表中选择出一个最小的记录需要比较n-1 次。如能把待排序的n个记录构成一个完全二叉树结构，则每次选择出一个最大（或最小）的记录比较的次数就是完全二叉树的高度，即$\log_2 n$次，则排序算法的时间复杂度就是$0(n\log_2 n)$。这就是堆排序(Heap Sort)的基本思想。

堆分为最大堆和最小堆两种。最大堆的定义如下：

设顺序表 sqList 中存放了 n 个记录，对于任意的 i(0≤i≤n-1)，如果 2i+1<n 时有 sqList[i]的关键码不小于 sqList[2i+1]的关键码；如果 2i+2<n 时有 sqList[i] 的关键码不小于 sqList[2i+2] 的关键码，则这样的堆为最大堆。

如果把这 n 个记录看作是一棵完全二叉树的结点，则 sqList[0]对应完全二叉树的根，sqList[1]对应树根的左孩子结点，sqList[2]对应树根的右孩子结点，sqList[3]对应 sqList[1]的左孩子结点，sqList[4]对应 sqList[2]的右孩子结点，如此等等。在此基础上，只需调整所有非叶子结点满足：sqList[i] 的关键码不小于 sqList[2i+1] 的关键码和 sqList[i] 的关键码不小于 sqList[2i+2] 的关键码，则这样的完全二叉树就是一个最大堆。

图 7.4(a)所示是一棵完全二叉树，图 7.4(b)所示是一个最大堆。



(a) 完全二叉树　　(b) 最大堆

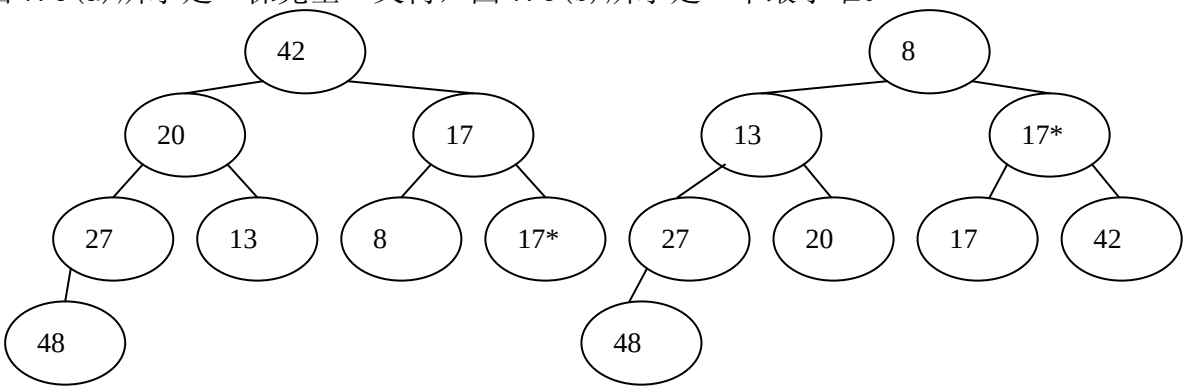
图 7.4 完全二叉树和最大堆

类似地，最小堆的定义如下：

设顺序表 sqList 中存放了 n 个记录,对于任意的 i(0≤i≤n-1),如果 2i+1<n 时有 sqList[i] 的关键码不大于 sqList[2i+1] 的关键码；如果 2i+2<n 时有 sqList[i] 的关键码不大于 sqList[2i+2] 的关键码，则这样的堆为最小堆。

如果把这 n 个记录看作是一棵完全二叉树的结点，则 sqList[0]对应完全二叉树的根,sqList[1]对应树根的左孩子结点,sqList[2]对应树根的右孩子结点，sqList[3]对应 sqList[1]的左孩子结点，sqList[4]对应 sqList[2]的右孩子结点，如此等等。在此基础上，只需调整所有非叶子结点满足：sqList[i] 的关键码不大于 sqList[2i+1] 的关键码和 sqList[i] 的关键码不大于 sqList[2i+2] 的关键码，则这样的完全二叉树就是一个最小堆。

图 7.5(a)所示是一棵完全二叉树，图 7.5(b)所示是一个最小堆。



(a) 完全二叉树　　(b) 最小堆

图 7.5 完全二叉树和最小堆

由堆的定义可知，堆有如下两个性质：

（1）最大堆的根结点是堆中关键码最大的结点，最小堆的根结点是堆中关键码最小的结点，我们称堆的根结点记录为堆顶记录。

（2）对于最大堆，从根结点到每个叶子结点的路径上，结点组成的序列都是递减有序的；对于最小堆，从根结点到每个叶子结点的路径上，结点组成的序列都是递增有序的。

堆排序的过程是：设有 n 个记录，首先将这 n 个记录按关键码建成堆，将堆顶记录输出，得到 n 个记录中关键码最大（或最小）的记录。然后，再把剩下的 n-1 个记录，输出堆顶记录，得到 n 个记录中关键码次大（或次小）的记录。如

此反复，便可得到一个按关键码有序的序列。

因此，实现堆排序需解决两个问题：

（1）如何将 n 个记录的序列按关键码建成堆；

（2）输出堆顶记录后，怎样调整剩下的 n-1 个记录，使其按关键码成为一个新堆。

首先，以最大堆为例讨论第一个问题：如何将 n 个记录的序列按关键码建成堆。

根据前面的定义，顺序表 sqList 中的 n 个记录构成一棵完全二叉树，所有的叶子结点都满足最大堆的定义。对于第 1 个非叶子结点 sqList[i](i=(n-1)/2),由于其左孩子结点 sqList[2i+1]和右孩子结点 sqList[2i+2]都已是最大堆，所以，只需首先找出 sqList[2i+1]结点和 sqList[2i+2]结点中关键码的较大者，然后与 sqList[i]结点的关键码进行比较，如果 sqList[i]结点的关键码大于或等于这个较大的结点的关键码，则以 sqList[i]结点为根结点的完全二叉树已满足最大堆的定义；否则，对换 sqList[i]结点和关键码较大的结点，对换后以 sqList[i]结点为根结点的完全二叉树满足最大堆的定义。按照这样的方法，再调整第 2 个非叶子结点 sqList[i-1] （i=(n-1)/2），第 3 个非叶子结点 sqList[i-2]，……，直到根结点。当根结点调整完后，则这棵完全二叉树就是一个最大堆了。

当要调整结点的左右孩子结点是叶子结点时，上述调整过程非常简单；当要调整结点的左右孩子结点不是叶子结点时，上述调整过程要稍微复杂一些。因为调整过后，可能引起以左孩子结点（或右孩子结点）为根结点的完全二叉树不是一个最大堆，这时，需要调整以左孩子结点（或右孩子结点）为根结点的完全二叉树，使之成为一个最大堆。

图 7.6 说明了如何把图 7.4(a)所示的完全二叉树建成图 7.4(b)所示的最大堆的过程。

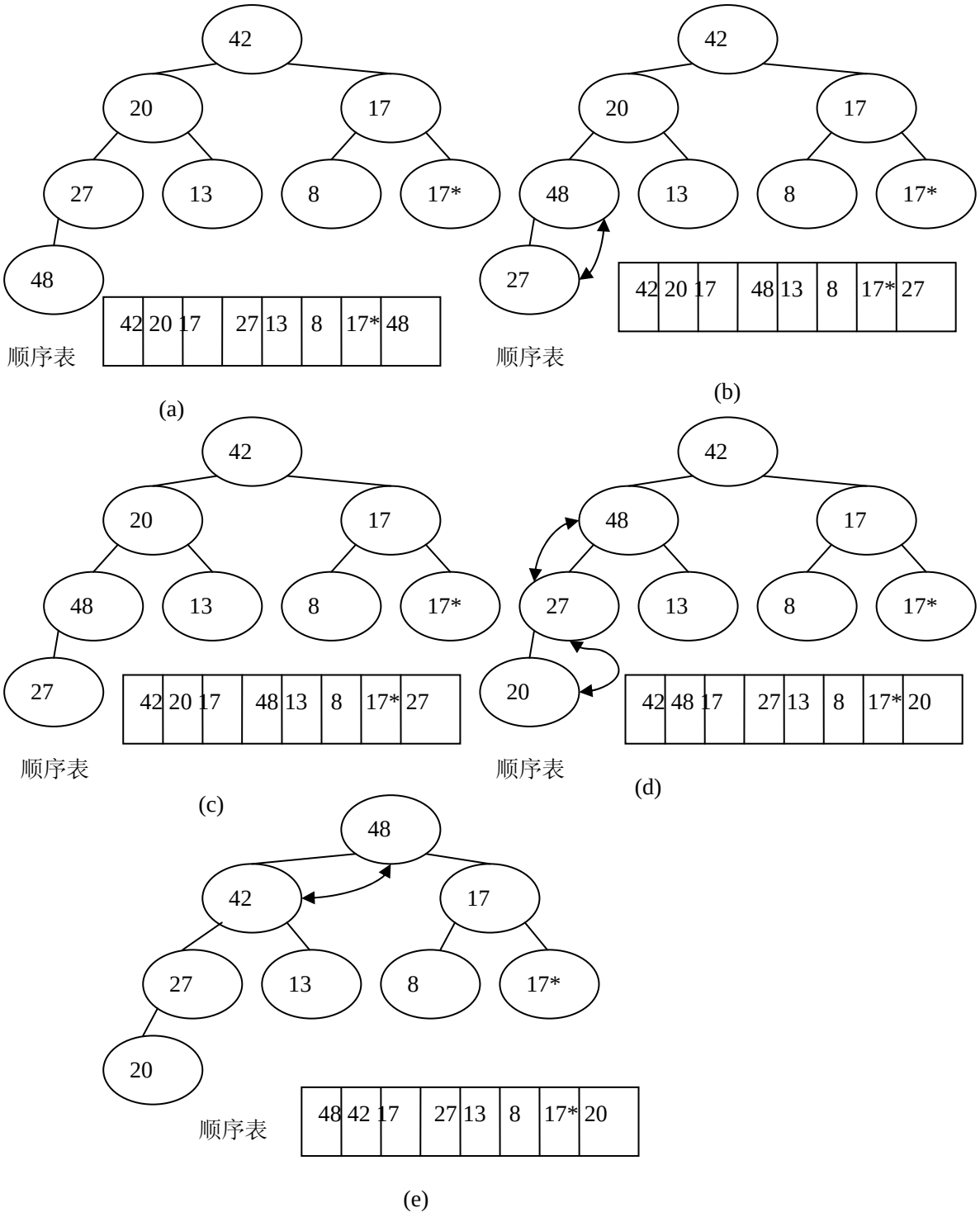


图 7.6 完全二叉树建成最大堆的过程

第一步：从 i=(n-1)/2=(7-1)/2=3 开始，sqList[3]的关键码 27 小于 sqList[7]的关键码 48，所以，sqList[3]与 sqList[7]交换，这样，以 sqList[3]为根结点的完全二叉树是一个最大堆，如图 7.6(b)所示。

第二步：当 i=2 时，由于 sqList[2] 的关键码 17 不小于 sqList[5]的关键码 8 和 sqList[6]的关键码 17*，所以不需要调整，如图 7.6(c)所示。

第三步：当 i=1 时，由于 sqList[1]的关键码 20 小于 sqList[3]的关键码 48，所以将 sqList[1]与 sqList[3]交换，这样导致 sqList[3]的关键码 20 小于

sqList[7]的关键码 27,所以将 sqList[3]与 sqList[7]交换,这样,以 sqList[1]为根结点的完全二叉树是一个最大堆，如图 7.6(d)所示。

第四步：当 i=0 时，对堆顶结点记录进行调整，由于 sqList[0] 的关键码 42 小于 sqList[1] 的关键码 48，所以将 sqList[0]与 sqList[1]交换，这样，以 sqList[0]为根结点的完全二叉树是一个最大堆，如图 7.6(e)所示，整个堆建立的过程完成。

建堆的算法如下所示，算法中记录的比较表示记录关键码的比较，顺序表中只存放了记录的关键码。

```
public void CreateHeap(SeqList<int> sqList, int low, int high)
{
    if ((low < high) && (high <= sqList.Last))
    {
        int j = 0;
        int tmp = 0;
        int k = 0;
        for (int i = high / 2; i >= low; --i)
        {
            k = i;
            j = 2 * k + 1;
            tmp = sqList[i];
            while (j <= high)
            {
                if ((j < high) && (j + 1 <= high)
                     && (sqList[j] < sqList[j + 1]))
                {
                    ++j;
                }

                if (tmp < sqList[j])
                {
                    sqList[k] = sqList[j];
                    k = j;
                    j = 2 * k + 1;
                }
                else
                {
                    j = high + 1;
                }
             }
            sqList[k] = tmp;
        }
    }
 }
```

把顺序表中的记录建好堆后，就可以进行堆排序了。

堆排序的算法如下所示，算法中记录的比较表示记录关键码的比较，顺序表中只存放了记录的关键码：

```
public void HeapSort(SeqList<int> sqList)
{
    int tmp = 0;

    CreateHeap(sqList, 0, sqList.Last);

    for (int i = sqList.Last; i > 0; --i)
    {
        tmp = sqList[0];
        sqList[0] = sqList[i];
        sqList[i] = tmp;

        CreateHeap(sqList, 0, i-1);
    }
}
```

堆排序算法是基于完全二叉树的排序方法。把一棵完全二叉树调整为堆，以及每次堆顶记录交换后进行调整的时间复杂度均为0（$n\log_2 n$），所以，堆排序算法的时间复杂度为0（$n\log_2 n$）。

图 7.7 是图 7.6(e)所示的最大堆进行堆排序的过程。

第一步：将堆顶记录（关键码为 48）与顺序表最后一个记录（关键码为 20）进行交换，使得堆顶记录的关键码 20 比根结点的左孩子结点的关键码 42 小，于是重新调整堆（记录的范围是从顺序表的第一个记录到倒数第二个记录，为了表示出顺序表的最后一个记录不在调整的范围之内，图 7.7(a)已经把最后一个结点从完全二叉树中去掉，以下同）。调整过程见图 7.7(b)所示。

第二步：将堆顶记录（关键码为 42）与顺序表倒数第二个记录（关键码为 17*）进行交换，使得堆顶记录的关键码 17*比根结点的左孩子结点的关键码 22 小，于是重新调整堆（记录的范围是从顺序表的第一个记录到倒数第三个记录）。调整过程见图 7.7(c)所示。

第三步：将堆顶记录（关键码为 27）与顺序表倒数第三个记录（关键码为 8）进行交换，使得堆顶记录的关键码 8 比根结点的左孩子结点的关键码 20 小，于是重新调整堆（记录的范围是从顺序表的第一个记录到倒数第四个记录）。调整过程见图 7.7(d)所示。

第四步：将堆顶记录（关键码为 20）与顺序表倒数第四个记录（关键码为 13）进行交换，使得堆顶记录的关键码 13 比根结点的左孩子结点的关键码 17*小，于是重新调整堆（记录的范围是从顺序表的第一个记录到倒数第五个记录）。调整过程见图 7.7(e)所示。

第五步：将堆顶记录（关键码为 17*）与顺序表倒数第五个记录（关键码为 8）进行交换，使得堆顶记录的关键码比根结点的右孩子结点的关键码 17 值小，于是重新调整堆（记录的范围是从顺序表的第一个记录到倒数第六个记录）。调整过程见图 7.7(f)所示。

第六步：将堆顶记录（关键码为 17）与顺序表倒数第六个记录（关键码为 8）进行交换，使得堆顶记录的关键码 8 比根结点的左孩子结点的关键码 13 小，于

是重新调整堆（记录的范围是从顺序表的第一个记录到倒数第七个记录）。调整过程见图 7.7(g)所示。

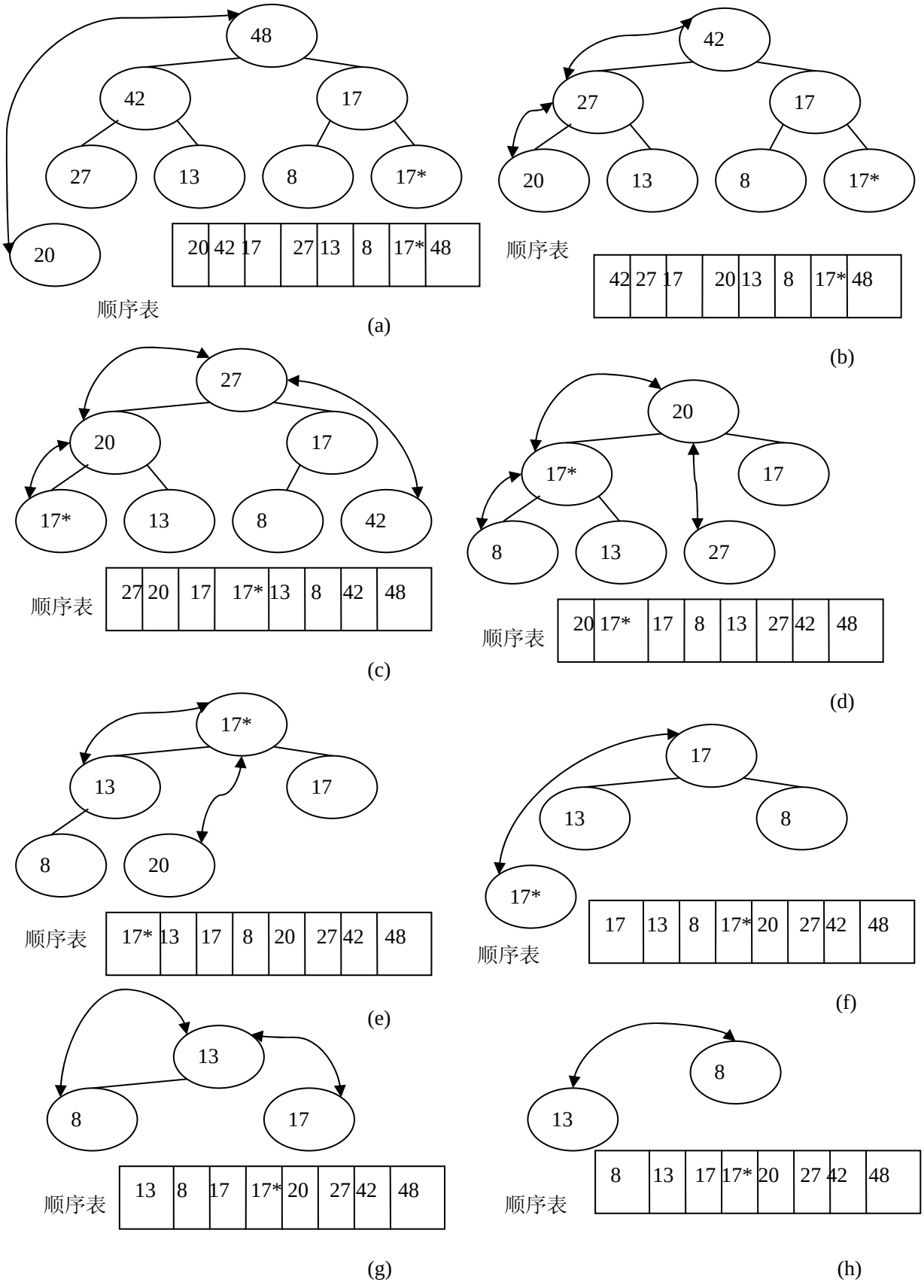


图 7.7 图 7.6(e)所示的最大堆进行堆排序的过程

第七步：将堆顶记录（关键码为 13）与顺序表第二个记录（关键码为 13）

进行交换，调整过程结束。调整过程见图 7.7(h)所示。

7.5 归并排序

归并排序(merge Sort)主要是二路归并排序。二路归并排序的基本思想是：将两个有序表合并为一个有序表。

假设顺序表 sqList 中的 n 个记录为 n 个长度为 1 的有序表，从第 1 个有序表开始，把相邻的两个有序表两两进行合并成一个有序表，得到 n/2 个长度为 2 的有序表。如此重复，最后得到一个长度为 n 的有序表。

一趟二路归并排序算法的实现如下所示，算法中记录的比较代表记录关键码的比较，顺序表中只存放了记录的关键码：

```
public void Merge(SeqList<int> sqList, int len)
{
    int m = 0;                      //临时顺序表的起始位置
    int l1 = 0;                     //第1个有序表的起始位置
    int h1;                         //第1个有序表的结束位置
    int l2;                         //第2个有序表的起始位置
    int h2;                         //第2个有序表的结束位置
    int i = 0;
    int j = 0;

    //临时表，用于临时将两个有序表合并为一个有序表
    SeqList<int> tmp = new SeqList<int>(sqList.GetLength());

   //归并处理
    while (l1 + len < sqList.GetLength())
   {
       l2 = l1 + len;               //第2个有序表的起始位置
       h1 = l2 - 1;                 //第1个有序表的结束位置

      //第2个有序表的结束位置
       h2 = (l2 + len - 1 < sqList.GetLength())
            ? l2 + len - 1 : sqList.Last;

       j = l2;
       i = l1;

       //两个有序表中的记录没有排序完
       while ((i<=h1) && (j<=h2))
       {
            //第1个有序表记录的关键码小于第2个有序表记录的关键码
            if (sqList[i] <= sqList[j])
            {
                tmp[m++] = sqList[i++];
            }
            //第2个有序表记录的关键码小于第1个有序表记录的关键码
```

```
                else
                {
                    tmp[m++] = sqList[j++];
                }
            }

            //第1个有序表中还有记录没有排序完
            while (i <= h1)
            {
                tmp[m++] = sqList[i++];
            }

            //第2个有序表中还有记录没有排序完
            while (j <= h2)
            {
                tmp[m++] = sqList[j++];
            }

            l1 = h2 + 1;
        }

        i = l1;
        //原顺序表中还有记录没有排序完
        while (i < sqList.GetLength())
        {
            tmp[m++] = sqList[i++];
        }

        //临时顺序表中的记录复制到原顺序表，使原顺序表中的记录有序
        for (i = 0; i < sqList.GetLength(); ++i)
        {
            sqList[i] = tmp[i];
        }
    }
```

二路归并排序算法的实现如下：

```
 public void MergeSort(SeqList<int> sqList)
 {
      int k = 1;     //归并增量

      while (k < sqList.GetLength())
      {
          Merge(sqList, k);
          k *= 2;
       }
```

}

对于n个记录的顺序表，将这n个记录看作叶子结点，若将两两归并生成的子表看作它们的父结点，则归并过程对应于由叶子结点向根结点生成一棵二叉树的过程。所以，归并趟数约等于二叉树的高度减1，即$\log_2 n$，每趟归并排序记录关键码比较的次数都约为n/2，记录移动的次数为2n（临时顺序表的记录复制到原顺序表中记录的移动次数为n）。因此，二路归并排序的时间复杂度为O（$n\log_2 n$）。而二路归并排序使用了n个临时内存单元存放记录，所以，二路归并排序算法的空间复杂度为O（n）。

【例 7-5】关键码序列为（42，20，17，27，13，8，17*，48），写出用二路归并排序算法进行排序的过程。

二路归并排序过程如图 7.8 所示。

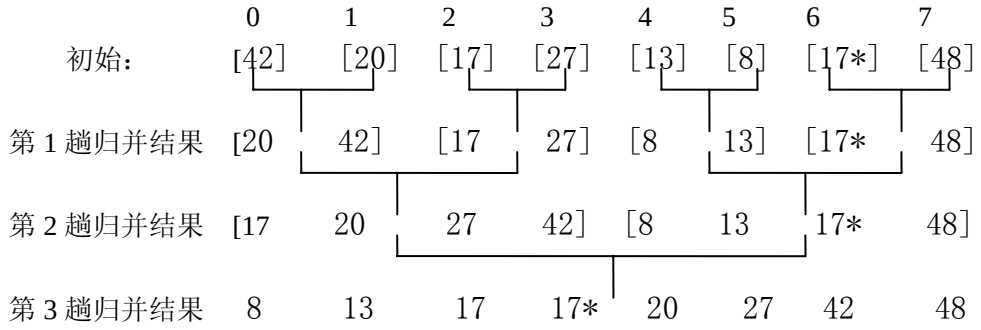


图 7.8 二路归并排序过程

7.6 基数排序

基数排序(Radix Sort)的设计思想与前面介绍的各种排序方法完全不同。前面介绍的排序方法主要是通过关键码的比较和记录的移动这两种操作来实现排序的，而基数排序不需要进行关键码的比较和记录的移动。基数排序是一种借助于多关键码排序的思想，是将单关键码按基数分成多关键码进行排序的方法，是一种分配排序。

7.6.1 多关键码排序

下面用一个具体的例子来说明多关键码排序的思想。

一副扑克牌有 52 张牌，可按花色和面值进行分类，其大小关系如下：

花色：梅花<方块<红心<黑心

面值：2<3<4<5<6<7<8<9<10<J<Q<K<A

在对这 52 张牌进行升序排序时，有两种排序方法：

方法一：可以先按花色进行排序，将牌分为 4 组，分别是梅花组、方块组、红心组和黑心组，然后，再对这 4 个组的牌分别进行排序。最后，把 4 个组连接起来即可。

方法二：可以先按面值进行排序，将牌分为 13 组，分别是 2 号组、3 号组、4 号组、…、A 号组，再将这 13 组的牌按花色分成 4 组，最后，把这四组的牌连接起来即可。

设序列中有n个记录，每个记录包含d个关键码$\{k^1, k^2, \cdots, k^d\}$，序列有序指的是对序列中的任意两个记录$r_i$和$r_j$（$1 \leqslant i \leqslant j \leqslant n$），$(k_i^1, k_i^2, \cdots, k_i^d) < (k_j^1, k_j^2, \cdots, k_j^d)$

其中，k^1称为最主位关键码，k^d称为最次位关键码。

多关键码排序方法按照从最主位关键码到最次位关键码或从最次位关键码到最主位关键码的顺序进行排序，分为两种排序方法：

（1）最高位优先法（MSD法）。先按k^1排序，将序列分成若干子序列，每个子序列中的记录具有相同的k^1值；再按k^2排序，将每个子序列分成更小的子序列；然后，对后面的关键码继续同样的排序分成更小的子序列，直到按k^d排序分组分成最小的子序列后，最后将各个子序列连接起来，便可得到一个有序的序列。前面介绍的扑克牌先按花色再按面值进行排序的方法就是MSD法。

（2）最次位优先法（LSD法）。先按k^d排序，将序列分成若干子序列，每个子序列中的记录具有相同的k^d值；再按k^{d-1}排序，将每个子序列分成更小的子序列；然后，对后面的关键码继续同样的排序分成更小的子序列，直到按k^1排序分组分成最小的子序列后，最后将各个子序列连接起来，便可得到一个有序的序列。前面介绍的扑克牌先按面值再花色按进行排序的方法就是LSD法。

7.6.2 链式基数排序

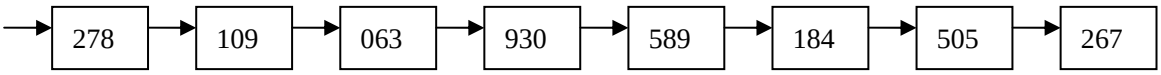
有时，待排序序列中的记录的关键码只有一个，但这个关键码可以拆分成若干项，每项作为一个关键码，则单关键码排序就可以按照多关键码的排序方法进行排序。比如，关键码为 4 位的整数，可以将关键码拆分成 4 项，每位对应一项。又比如，关键码由 5 个字符组成的字符串，可以把每个字符作为一个关键码。拆分后的每个关键码的取值范围相同（数字是 0~9，字符是 a~z），称关键码的取值范围为基，记做 RADIX。

链式基数排序的基本思想是：设立RADIX个队列，队列编号为 0～RADIX-1。首先从最低位关键码起，按关键码的不同值将待排序列中的记录分配到这RADIX个队列中。如k_i^d=8，则把第i个记录分配到第 8 号队列中；然后从小到大将记录再依次收集起来。这时，n个记录已经按最低位关键码有序。如此重复进行，直至最高位关键码，次数为关键码的个数。这样就得到了一个有序的序列。为了减少记录的移动次数，队列采用链式分配，因此，RADIX个队列是RADIX个链表。每个链表设立两个引用，一个指向链表的头，即指向第一个进入该队列的记录，记为f[i]；另一个引用指向表尾，即指向当前队列中刚进入的记录，记为e[i]。

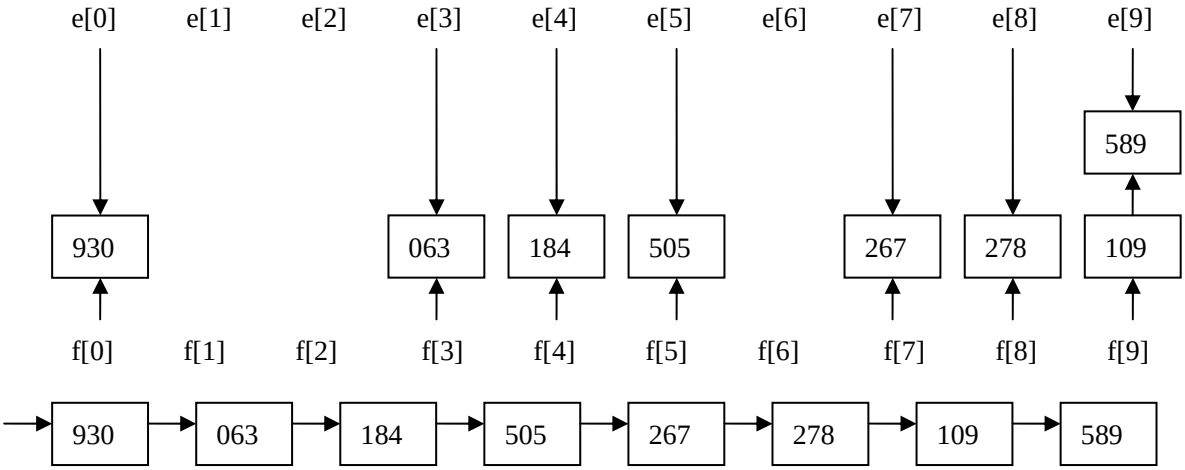
为快速、方便地将每个非空的队列链接到链表的后面，还设立了一个引用 r，r 的初值指向第一个非空队列的队尾。然后，每收集一个非空队列，总是使 r 引用指向新形成的链表的最后一个结点。在收集操作中，实际上没有进行记录的移动，而仅仅是修改有关的引用，从而形成按关键码某位的值的大小排序的链表。

【例 7-6】关键码序列为（278，109，063，930，589，184，505，269），用静态链表存储，头结点指向第一个记录，写出用链式基数排序算法进行排序的过程。

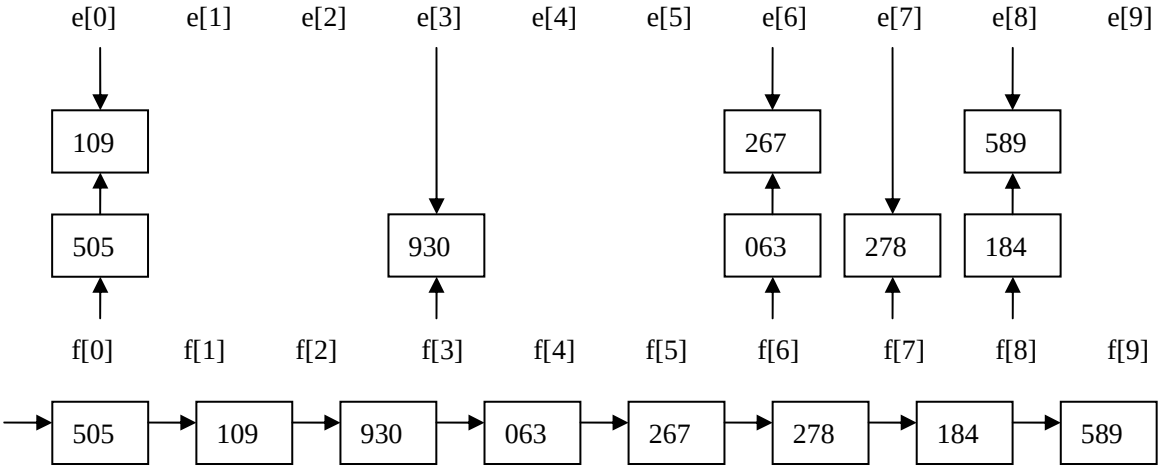
链式基数排序过程如图 7.9 所示。



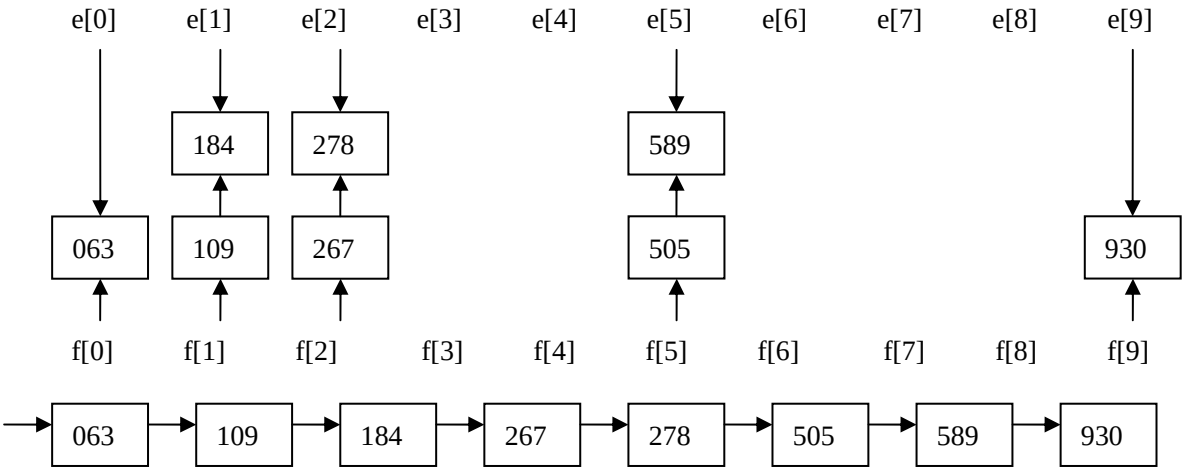
(a) 排序前



(b) 第一趟排序后



(c) 第二趟排序后



(d) 第三趟排序后

图 7.9 链式基数排序过程

设待排序序列有 n 个记录、d 个关键码，基为 radix，则链式基数排序的时间复杂度为 O(d(n+radix))。其中一趟分配的时间复杂度为 O(n)，收集的时间复杂度为 O(radix)，共进行 d 趟分配和收集。

链式基数排序需要 2* radix 个指向队列的引用，以及用于静态链表的 n 个引用。

链式基数排序是稳定的排序方法。

7.7 各种排序方法的比较与讨论

排序在计算机程序设计中非常重要，上面介绍的各种排序方法各有优缺点，适用的场合也各不相同。在选择排序方法时应考虑的因素有：

（1）待排序记录的数目 n 的大小；

（2）记录本身除关键码外的其它信息量的大小；

（3）关键码的情况；

（4）对排序稳定性的要求；

（5）语言工具的条件，辅助空间的大小等。

综合考虑以上因素，可以得出如下结论：

（1）若排序记录的数目 n 较小（如 n≤50）时，可采用直接插入排序或简单选择排序。由于直接插入排序所需的记录移动操作较简单选择排序多，因而当记录本身信息量较大时，用简单选择排序比较好。

（2）若记录的初始状态已经按关键码基本有序，可采用直接插入排序或冒泡排序。

（3）若排序记录的数目n较大，则可采用时间复杂度为O（$n\log_2 n$）的排序方法（如快速排序、堆排序或归并排序等）。快速排序的平均性能最好，在待排序序列已经按关键码随机分布时，快速排序最适合。快速排序在最坏情况下的时间复杂度是O（n^2），而堆排序在最坏情况下的时间复杂度不会发生变化，并且所需的辅助空间少于快速排序。但这两种排序方法都是不稳定的排序，若需要稳定的排序方法，则可采用归并排序。

（4）基数排序可在 O(d×n)（d 为关键码的个数，当 n 远远大于 d 时）时间内完成对 n 个记录的排序，但基数排序只适合于字符串和整数这种有明显结构特征的关键码。当 n 很大、d 较小时，可采用基数排序。

（5）前面讨论的排序算法，除基数排序外，其它排序算法都是采用顺序存储结构。在待排序的记录非常多时，为避免花大量的时间进行记录的移动，可以采用链式存储结构。直接插入排序和归并排序都可以非常容易地在链表上实现，但快速排序和堆排序却很难在链表上实现。此时，可以提取关键码建立索引表，然后对索引表进行排序。也可以引入一个整形数组 t[n]作为辅助表，排序前令 t[i]=i，1≤i≤n。若排序算法中要求交换记录 R[i]和 R[j]，则只须交换 t[i]和 t[j]即可。排序结束后，数组 t[n]就存放了记录之间的顺序关系。

7.8 C#中排序方法

C#语言中的许多类都提供了排序的成员方法。比如，在第四章介绍的数组类 Array 就提供了排序方法 Sort。Array 类中的排序方法采用的是快速排序算法，并且要求数组中的数据元素必须实现 IComparable 接口，这样数据元素之间才能进行比较。实际上，任何类型的数据都是使用比较器进行排序，所以，该类型要实现排序方法，都必须实现 IComparable 接口。泛型类实现了泛型 IComparable 接口，非泛型类实现非泛型 IComparable 接口。C#中提供了 Comparer 类来实现各种比较器。

又比如，泛型 List 类也实现了 Sort 方法。Sort 方法使用 Comparer 类的比较器来决定 List<T>类中数据元素的顺序。比较器首先检查 List<T>类中数据元素是否实现了泛型 IComparable 接口，如果实现了比较器就使用该实现，否则，比较器再检查数据元素是否实现了非泛型 IComparable 接口，如果实现了就使用该实现。如果这两种接口都没有实现，比较器将抛出一个 InvalidOperationException 异常。实际上，泛型 List<T>类的 Sort 方法使用了数组类 Array 的 Sort 方法。

同样，ASP.NET 2.0 中的GridView控件也实现了Sort方法。GridView控件的功能类似于DataGrid控件，但比DataGrid控件更吸引人，使用更方便，程序员写的代码也更少。Sort方法使用排序表达式和排序方向对GridView控件进行排序。排序表达式用于决定要排序的列，它可以对多列进行排序。排序方向决定是升序排序还是降序排序。

C#中实现了排序方法的类还有很多，由于篇幅限制，这里就不一一列举了，感兴趣的读者可以查阅.NET的有关书籍。

本章小结

排序是计算机程序设计中的一种重要操作，是把一个没有序的记录序列重新排列成按记录的某个关键码有序的序列的过程。排序方法按涉及的存储器不同分为内部排序和外部排序两类。内部排序指记录存放在内存中并且在内存中调整记录之间的相对位置，没有内、外存的数据交换。外部排序指记录的主要部分存放在外存中，借助于内存调整记录之间的相对位置，需要在内、外存之间交换数据。

排序方法按记录在排序前后的位置关系是否一致，分为稳定排序和不稳定排序。稳定排序方法在排序前后相同关键码值的记录之间的位置关系不变，不稳定排序方法在排序前后相同关键码值的记录之间的位置关系改变。

本章主要介绍了常用的内部排序方法，包括三种简单排序方法，即直接插入排序、冒泡排序和简单选择排序，这三种排序方法在最好情况下的时间复杂度为O(n)，在平均情况下和最坏情况下的时间复杂度都为$O(n^2)$,并且都是稳定的排序方法。

快速排序方法的平均性能最好，时间复杂度为$O(n\log_2 n)$，所以，当待排序序列已经按关键码随机分布时，快速排序是最适合的。但快速排序在最坏情况下的时间复杂度是$O(n^2)$。快速排序方法是不稳定的排序方法。

堆排序方法在最好情况下、平均情况下和最坏情况下的时间复杂度不会发生变化，为$O(n\log_2 n)$，并且所需的辅助空间少于快速排序方法。堆排序方法也是不稳定的排序方法。

归并排序方法在最好情况下、平均情况下和最坏情况下的时间复杂度不会发生变化，为$O(n\log_2 n)$，但需要的辅助空间大于堆排序方法，但归并排序方法是稳定的排序方法。

以上排序方法都是通过记录关键码的比较和记录的移动来进行排序，而基数排序方法是一种借助于多关键码排序的思想，是将单关键码按基数分成多关键码进行排序的方法。

一般情况下，排序都采用顺序存储结构（基数排序方法除外），而当记录非常多时可以采用链式存储结构，但快速排序和堆排序却很难在链表上实现。

习题七

7.1 试解释以下概念：

关键码　主关键码　次主关键码　内部排序　外部排序

7.2 什么是稳定的排序方法？什么是不稳定的排序方法？对不稳定的排序方法

各举一例进行说明。

7.3 设待排序记录的关键码序列为（11，4，18，33，29，9，18*，21，5，19），分别写出使用下列排序方法进行排序的过程。

（1）直接插入排序；

（2）冒泡排序；

（3）简单选择排序；

（4）快速排序；

（5）堆排序；

（6）归并排序。

7.4 试比较直接插入排序、冒泡排序、简单选择排序、快速排序、堆排序和归并排序的特点和各自的适用范围，分析它们算法的时间复杂度。

7.5 判断下列序列是否为堆，若是则进一步推出是最大堆还是最小堆。

（1）（50，36，41，19，23，4，20，18，12，22）

（2）（43，5，47，1，19，11，59，15，48，41）

（3）（50，36，41，19，23，20，18，12，22）

（4）（9，13，17，21，22，31，33，24，27，23）

7.6 一个线性表中的数据元素为正整数或负整数。试设计一算法，将正整数和负整数分开，使线性表的前一部分的数据元素为负整数，后一部分的数据元素为正整数。不要求对这些数据元素有序，但要求尽量减少交换的次数。

7.7 对长度为 n 的待排序序列进行快速排序时，所需进行的比较次数依赖于这 n 个数据元素的初始序列。

（1）当 n=7 时，在最好情况下进行多少次比较？并说明理由。

（2）给出 n=7 时的最好情况的初始序列。

7.8 试编写一个双向冒泡算法，即在排序过程中以交替的正、反两个方向进行扫描。若第 1 趟把关键码最大的记录放到最末尾，则第 2 趟把关键码最小的记录放到最前端，如此反复进行之。

7.9 假设有 10000 个 1～10000 的互不相同的数构成一无序集合。试设计一个算法实现排序，要求以尽可能少的比较次数和移动次数实现。

第8章 查找

在日常生活中，我们经常需要进行查找。比如，在英汉词典中查找某个英文单词的中文解释，在图书馆中查找一本书。查找是为了得到某个信息而进行的工作。

在程序设计中，查找是对数据结构中的记录（和排序一样，在查找中把数据元素称为记录）进行处理时经常采用的一种操作。查找又称检索，它是计算机科学中的重要研究课题之一，查找的目的就是从确定的数据结构中找出某个特定的记录。查找在许多程序中耗时最多，因此，一个好的查找方法会大大提高程序的运行速度。

8.1 基本概念和术语

查找(Search)是在数据结构中确定是否存在关键码等于给定关键码的记录的过程。关键码的概念在第 7 章已经讨论过，关键码有主关键码和次关键码。主关键码能够唯一区分各个不同的记录，次关键码通常不能唯一区分各个不同的记录。以主关键码进行的查找是最经常、也是最主要的查找。

查找有静态查找(Static Search)和动态查找(dynamic Search)两种。静态查找是指只在数据结构里查找是否存在关键码等于给定关键码的记录而不改变数据结构，动态查找是指在查找过程中插入数据结构中不存在的记录，或者从数据结构中删除已存在的记录。

进行查找使用的数据结构称为查找表，查找表分为静态查找表和动态查找表。静态查找表用于静态查找，动态查找表用于动态查找。

在查找表里进行查找的结果有两种：查找成功和查找不成功。查找成功是指在查找表中找到了要查找的记录，查找不成功是指在查找表中没有找到要查找的记录。

和第 7 章讨论的排序问题一样，在各种具体的查找问题中，虽然不同记录的数据域差别很大，但查找算法只与记录的关键码有关，与其它域无关。不失一般性，假设查找表中只存储记录的关键码。并且，为了讨论问题的方便，假设关键码是整型。

衡量查找算法的最主要的标准是平均查找长度（Average Search Length，简称 ASL）。平均查找长度是指在查找过程中进行的关键码比较次数的平均值，其数学定义为：

$$ASL = \sum_{i=1}^{n} p_i \cdot c_i$$

其中，p_i是要查找记录的出现概率，c_i是查找相应记录需进行的关键码比较次数。

8.2 静态查找表

由于静态查找不需要在静态查找表中插入或删除记录，所以，静态查找表的数据结构是线性结构，可以是顺序存储的静态查找表或链式存储的静态查找表。本书采用第 2 章介绍的顺序表作为顺序存储的静态查找表，单链表作为链式存储的静态查找表。

8.2.1 顺序查找

顺序查找(Sequnce Search)又称线性查找(Linear Search)，其基本思想是：从静态查找表的一端开始，将给定记录的关键码与表中各记录的关键码逐一比较，若表中存在要查找的记录，则查找成功，并给出该记录在表中的位置；否则，查

找失败，给出失败信息。

下面以顺序表为例，记录从下标为 1 的单元开始存放，0 号单元用来存放要查找的记录，被称为监视哨。并且，监视哨设在顺序表的最低端，称为低端监视哨。也可以把监视哨设在顺序表的高端，称为高端监视哨。

顺序表的顺序查找的算法实现如下所示，算法中记录的比较表示记录关键码的比较，顺序表中只存放了记录的关键码：

```
public int SeqSearch(SeqList<int> sqList, int data)
{
    sqList[0] = data;
    int i = 0;

    for (i = sqList.GetLength(); sqList[i] > data; --i);
    return i;
}
```

性能分析：假设顺序表中每个记录的查找概率相同，即$p_i=1/n$（i=1，2，…，n），查找表中第i个记录，需进行n-i+1 次比较，即$c_i= n-i+1$。当查找成功时，顺序查找的平均查找长度为：

$$ASL = \sum_{i=1}^{n} p_i.c_i = \sum_{i=1}^{n} \frac{1}{n}.(n-i+1) = \frac{n+1}{2}$$

当查找不成功时，关键码的比较次数总是 n+1 次。

顺序查找的基本操作是关键码的比较，因此，查找表的长度就是查找算法的时间复杂度，即为 O(n)。

在许多情况下，顺序查找表中的记录的查找概率是不相等的。为了提高查找效率，查找表应根据记录 “查找概率越高关键码的比较次数越少，查找概率越低，关键码的比较次数越多”的原则来存储记录。

顺序查找虽然简单，但效率很低，特别是当查找表中的记录很多时更是如此。

8.2.2 有序表的折半查找

折半查找(Binary Search)又叫二分查找，其基本思想是：在有序表中，取中间的记录作为比较对象，如果要查找记录的关键码等于中间记录的关键码，则查找成功；若要查找记录的关键码小于中间记录的关键码，则在中间记录的左半区继续查找；若要查找记录的关键码大于中间记录的关键码，则在中间记录的右半区继续查找。不断重复上述查找过程，直到查找成功，或有序表中没有所要查找的记录，查找失败。

假设顺序表 sqList 是有序的，与快速排序一样，设两个指示器，一个 low，指示要查找的第 1 个记录的位置，开始时指向 sqList[1]的位置，sqList[0]留做存放要查找的记录的关键码；一个 high，指示要查找的最后一个记录的位置，开始时指示顺序表最后一个记录的位置。设要查找的记录的关键码为 key，当 low 不大于 high 时，反复执行以下步骤（记录的比较表示记录关键码的比较，有序表中只存放了记录的关键码）：

（1）计算中间记录的位置 mid，mid=(low+high)/2；

（2）若 key=sqList[mid]，查找成功，退出循环；

（3）若 key＜sqList[mid]，high=mid-1，转（1）；

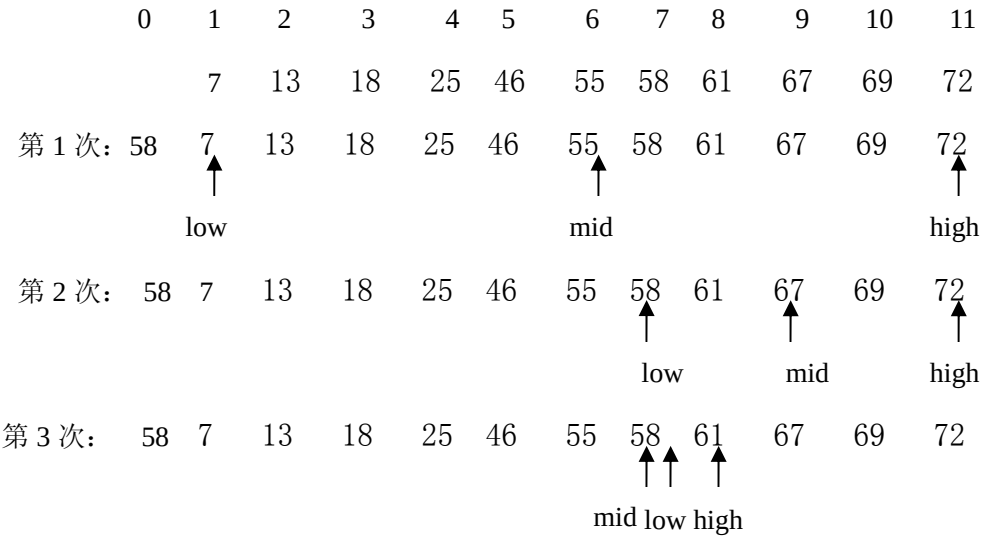
（4）若 key＞sqList[mid]，low=mid+1，转（1）；

有序表的折半查找的算法实现如下所示，算法中记录的比较表示记录关键码的比较，顺序表中只存放了记录的关键码：

```
public int BinarySearch(SeqList<int> sqList, int key)
{
    sqList[0] = key;                    //0单元存放要查找的记录
    int mid = 0;
    int flag = -1;
    int low = 1;                        //设置初始区间的下限值
    int high = sqList.GetLength();      //设置初始区间的上限值

   //记录没有查找完
    while (low <= high)
    {
        //取中点
        mid = (low + high)/2;

        //查找成功，记录位置存放到flag中
        if (sqList[0] == sqList[mid])
        {
            flag = mid;
            break;
        }
        //调整到左半区
        else if(sqList[0] < sqList[mid])
        {
            high = mid -1;
        }
        //调整到右半区
        else
        {
            low = mid  + 1;
        }
    }

    if (flag > 0)
    {
        Console.WriteLine("Search is successful!");
        return flag;
    }
    else
    {

        Console.WriteLine("Search is failing!");
        return -1;
```

```
        }
    }
```

【例 8-1】已知有序表中的记录按关键码排列如下：
（7，13，18，25，46，55，58，61，67，69，72）
试在表中查找关键码为 58 和 30 的记录。
查找关键码为 58 的记录的过程如下所示：

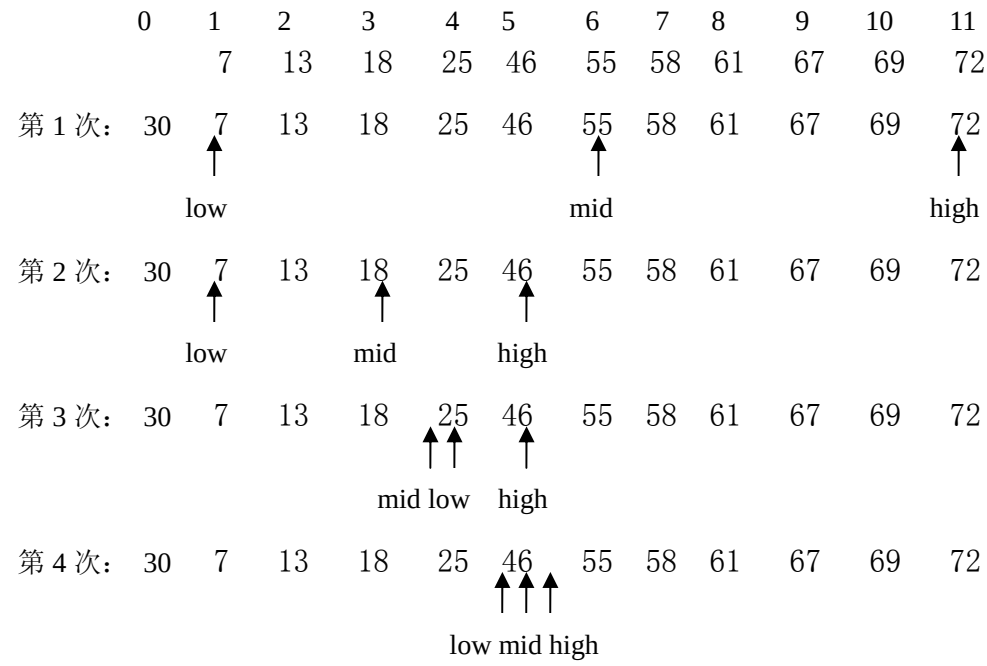


第 1 次比较，low=1，high=11，mid=(1+11)/2=6，58>sqList[6]=55，则 low=mid+1=7，high 不变。

第 2 次比较，low=7，high=11，mid=(7+11)/2=9，58<sqList[9]=67，则 low 不变，high=mid-1=8。

第 3 次比较，low=7，high=8，mid=(7+8)/2=7，58=sqList[7]=58，查找成功。

查找关键码为 30 的记录的过程如下所示：



第 1 次比较，low=1，high=11，mid=(1+11)/2=6，30<sqList[6]=55，则

high=mid-1=5，low 不变。

第 2 次比较，low=1，high=5，mid=(1+5)/2=3，30>sqList[3]=18，则 high 不变， low=mid+1=4。

第 3 次比较，low=4，high=5，mid=(4+5)/2=4，30>sqList[4]=25，则 high 不变， low=mid+1=5。

第 4 次比较，low=5，high=5，mid=(5+5)/2=5，30<sqList[5]=46，则 low 不变， high=mid-1=4，由于 high<low，表示有序表中没有关键码为 30 的记录，查找结束。

性能分析：从折半查找的过程来看，以有序表的中点作为比较对象，并以中点将有序表分为两个子表，对定位到的子表进行递归操作。所以，对有序表中的每个记录的查找过程，可用二叉树来描述，这棵二叉树称为判定树，如图 8.1 所示。

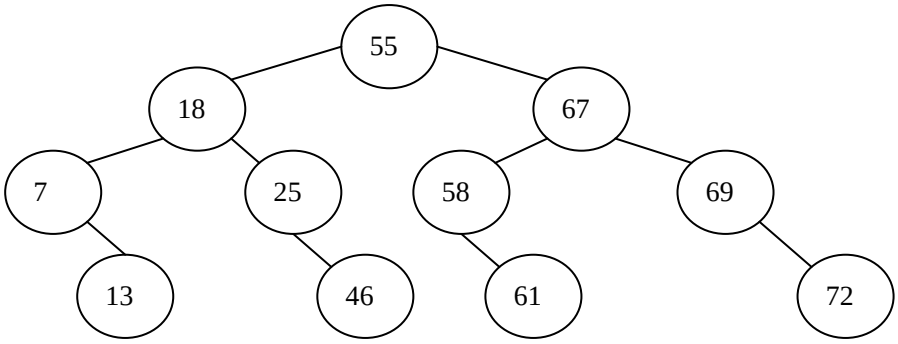


图 8.1 折半查找过程的判定树

从图 8.1 所示的判定树可知，查找有序表中任何一个记录的过程，即是从判定树的根结点到该记录结点路径上的各结点关键码与给定记录关键码的比较过程。所以，比较次数为该记录结点在判定树中的层次数。因此，由二叉树的性质可知，折半查找在查找成功时所进行的关键码比较次数为$\log_2(n+1)$次。

下面分析折半查找的平均查找长度。为了便于讨论，以层次为k的满二叉树（$n=2^k-1$）为例。假设顺序表中每个记录的查找概率是相等的，即$p_i=1/n$，则折半查找的平均查找长度为：

$$ASL=\sum_{i=1}^{n}p_i.c_i=\sum_{i=1}^{n}\frac{1}{n}.2^{i-1}=\frac{n+1}{2}\log_2(n+1)-1\approx\log_2(n+1)-1$$

所以，折半查找的时间复杂度为$O(\log_2 n)$。

折半查找的平均效率较高，但要求是有序表。

8.2.3 索引查找

当顺序表中的记录个数非常大时，使用前面介绍的查找算法，其效率非常低。此时提高查找效率的一个常用方法是在顺序表上建立一个索引表。索引表和教科书前面使用的索引非常相似。为了进行区分，把要建立索引表的顺序表称为主表。

索引查找(Index Search)又称分块查找，是对顺序查找的一种改进。为了提高查找的效率，索引表采用顺序存储并且必须有序，而主表中的记录不一定按关键码有序。因为对于记录个数非常大的主表而言，要按关键码有序，实现起来需要花费较多时间。所以，索引查找只要求主表中的记录按关键码分段有序，将主表分成若干个子表，并对子表建立索引表。因此，主表的每个子表由索引表中的记录确定。索引表中的记录由两个域组成：一个域 data，存放对应子表中的最大关键码的值，一个域 link，存放对应子表的第一个记录在主表中的位置。

图 8.2 是一个主表和一个索引表的结构图。

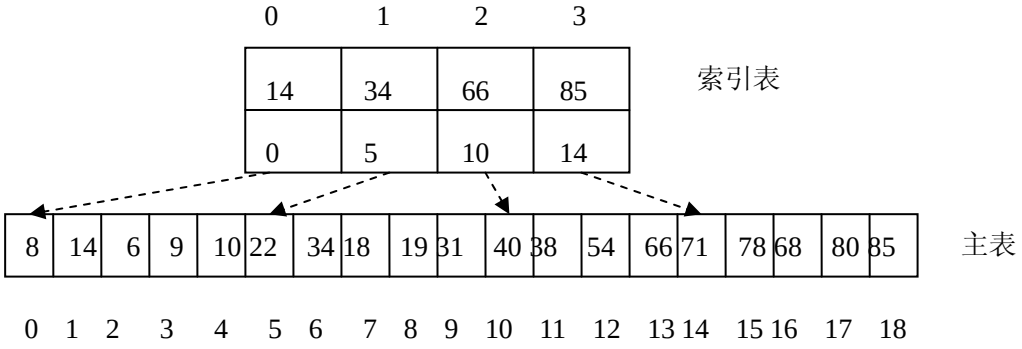


图 8.2 一个主表和一个索引表的结构图

索引查找分两步进行。先确定待查找记录所在的子表，然后在子表中进行顺序查找。比如，在图 8.2 中，现在要查找关键码为 54 的记录。先将 54 依次与索引表中每个记录的 data 域的值进行比较，由于 34<54<66，则若在主表中存在关键码为 54 的记录，则该记录必定在主表的第 3 个子表中。由于相应的 link 域的值为 10，所以，从主表的第 11 个记录（数组的下标为 10）开始进行顺序查找。当比较到主表的第 13 个记录（数组的下标为 12）时，关键码相等，说明主表中有要查找的记录，则查找成功。当然，如果比较到第 15 个记录（因为索引表的下一个记录的 link 域的值为 14）仍然不等，说明主表中不存在要查找的记录，查找失败。

性能分析：索引查找由索引表查找和子表查找两步完成。设 n 个记录的顺序表分为 m 个子表，且每个子表均有 t 个记录，则 t=n/m。这样，索引查找的平均查找长度为：

$$ASL=ASL_{索引表}+ASL_{子表}=(m+1)/2+(n/m+1)/2=(m+n/m)/2+1$$

当主表中的记录个数非常多时，索引表本身可能也很大，此时可按照建立索引表的方法对索引表再建立索引表，这样的索引表称为二级索引表。同样的方法还可建立三级索引表。二级以上的索引结构称为多级索引结构。

8.3 动态查找表

静态查找表一旦生成后，所含记录在查找过程中一般固定不变，而动态查找表则不然。动态查找表在查找过程中动态生成，即若表中存在关键码为 key 的记录，则查找成功返回，否则，则插入关键码为 key 的记录。在动态查找表中，经常需要对表中的记录进行插入和删除操作，所以动态查找表采用灵活的存储方法来组织查找表中的记录，以便高效率地实现查找、插入和删除等操作。

动态查找表有很多，本节重点介绍二叉排序树。

1、二叉排序树的基本概念

二叉排序树(Binary Sorting Tree)或者是一棵空树，或者是一棵具有以下性质的二叉树：

（1）若左子树非空，则左子树上所有结点关键码的值均小于根结点关键码的值；

（2）若右子树非空，则右子树上所有结点关键码的值均大于根结点关键码的值；

（3）左右子树也都是二叉排序树。

二叉排序树的示例如图 8.3 所示。

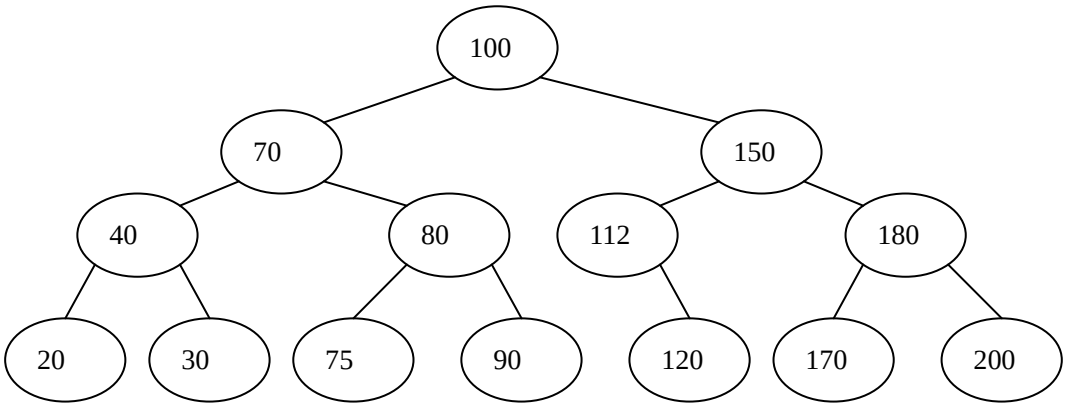


图 8.3 二叉排序树

本节以不带头结点的二叉树的二叉链表作为二叉排序树的存储结构，并且，不失一般性，假设动态查找表只存储记录的关键码，而关键码的数据类型为整型。

2、二叉排序树上的查找操作

在二叉排序树上查找某个记录是否存在的过程和在顺序表中查找某个记录是否存在的过程非常相似。从二叉排序树的基本概念可知，其查找过程为：

（1）若二叉排序树为空，则查找失败（下面的算法实现以返回 1 表示查找失败）；

（2）若二叉排序树非空，则将给定记录的关键码与二叉排序树根结点的关键码进行比较，如果相等，则查找成功（下面的算法实现以返回 0 代表查找成功），查找过程结束。否则，执行以下两步中的一步；

（3）若给定记录的关键码小于二叉排序树根结点的关键码，查找将在根结点的左子树上继续进行，转（1）；

（4）若给定记录的关键码大于二叉排序树根结点的关键码，查找将在根结点的右子树上继续进行，转（1）。

二叉排序树上的查找算法的实现如下：

```
public int Search(BiTree<int> bt, int key)
{
      Node<int> p;

      //二叉排序树为空
      if (bt.IsEmpty() == true)
      {
            Console.WriteLine("The Binary Sorting Tree is empty!");
            return 1;
      }

      p = bt.Head;
      //二叉排序树非空
      while (p != null)
      {
            //存在要查找的记录
            if (p.Data == key)
```

```
            {
                Console.WriteLine("Search is Successful!");
                return 0;
            }
            //待查找记录的关键码大于结点的关键码
            else if (p.Data < key)
            {
                p = p.LChild;
            }
            //待查找记录的关键码小于结点的关键码
            else
            {
                p = p.RChild;
            }
        }

        return 1;
    }
```

3、二叉排序树上的插入操作

在二叉排序树上插入某个记录，首先要在二叉排序树上进行查找，如果要插入的记录在二叉排序树上存在，则不插入（在下面的算法实现中以返回 1 表示不插入）。如果要插入的记录在二叉排序树上不存在，则把该记录插入到查找失败时的结点的左孩子结点或右孩子结点上。因此，二叉排序树上的插入过程首先是一个查找过程。这个查找过程和前面讨论的查找过程不同之处在于：这里的查找过程要同时记住当前结点的位置，以便当查找不成功时把由要查找的记录生成的结点的地址赋给当前结点的左孩子引用域或右孩子引用域。并且，新插入的结点一定是作为叶子结点进行插入的。

二叉排序树上的插入算法的实现如下：

```
public int Insert(BiTree<int> bt, int key)
{
    Node<int> p;
    Node<int> parent = new Node<int>();

    p = bt.Head;
    //二叉排序树非空
    while (p != null)
    {
        //存在关键码等于key的结点
        if (p.Data == key)
        {
            Console.WriteLine("Record  is exist!");
            return 1;
        }
```

```
                parent = p;
                //记录的关键码大于结点的关键码
                if (p.Data < key)
                {
                    p = p.RChild;
                }
                //记录的关键码小于结点的关键码
                else
                {
                     p = p.LChild;
                }
            }

            p = new Node<int>(key);
            //二叉排序树为空
            if(parent == null)
            {
                bt.Head = p;
            }
            //待插入记录的关键码小于结点的关键码
            else if (p.Data < parent.Data)
            {
                 parent.LChild = p;
            }
            //待插入记录的关键码大于结点的关键码
            else
            {
                 parent.RChild = p;
            }
            return 0;
        }
```

由二叉排序树上的插入算法可以得出一个结论：构造一棵二叉排序树的过程就是从一棵空的二叉排序树上开始逐个插入结点的过程。下面的例子说明了这个过程。

【例 8-2】已知记录的关键码序列为（63，90，70，55，67，42，88），试构造一棵二叉排序树。

构造二叉排序树的过程如图 8.4 所示，Φ代表空的二叉排序树：

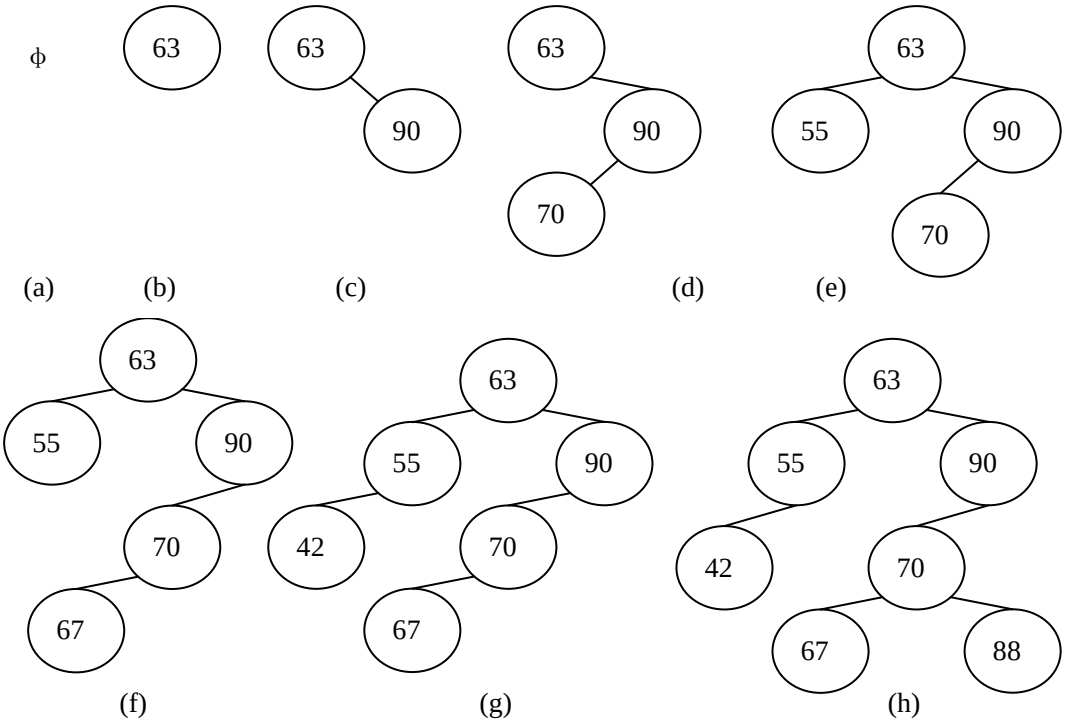


图 8.4 二叉排序树的构造过程

4、二叉排序树上的删除操作

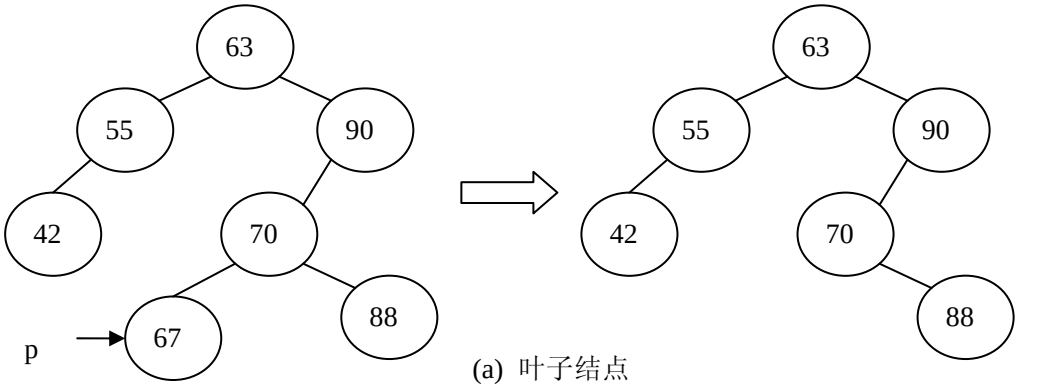
在二叉排序树上删除一个记录首先要求判断二叉排序树是否为空，如果二叉排序树为空，则不进行删除，返回（下面的算法实现以返回 1 表示二叉排序树为空）；然后判断查找该记录是否在二叉排序树中存在，若不存在则返回（下面的算法实现以返回-1 表示记录不存在）；若存在，则分以下几种情况进行处理。

（1）若被删除的结点是叶子结点，则直接删除该结点，并置其父亲结点的相应引用域为空；

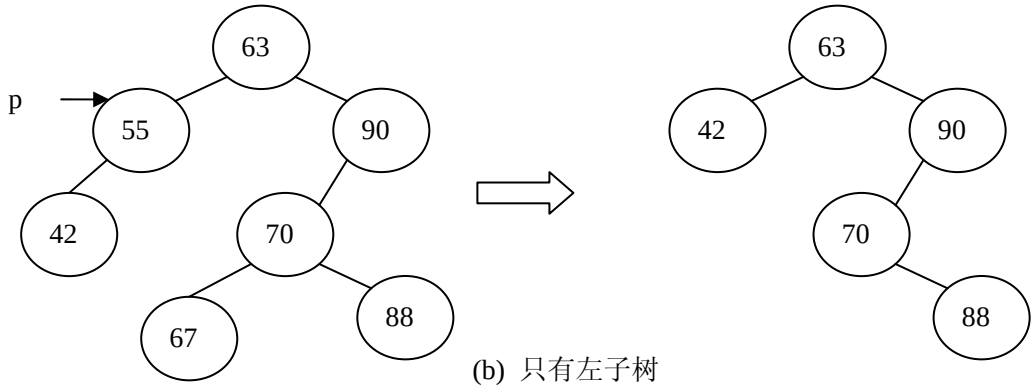
（2）若被删除结点有左孩子或右孩子，则删除该结点并把该结点的左孩子或右孩子赋给其父亲结点的相应引用域即可；

（3）若被删除结点有左孩子和右孩子，则首先寻找被删除结点在该二叉排序树中的中序遍历中直接前驱（或直接后继）结点，也就是说，该结点是所有关键码的值大于被删除结点关键码的值的结点中关键码的值最小的结点，即被删除结点的右子树的最左结点；然后用直接前驱（或直接后继）结点取代被删除结点；最后从二叉排序中删除直接前驱（或直接后继）结点，本节的算法以直接后继结点为例进行说明。

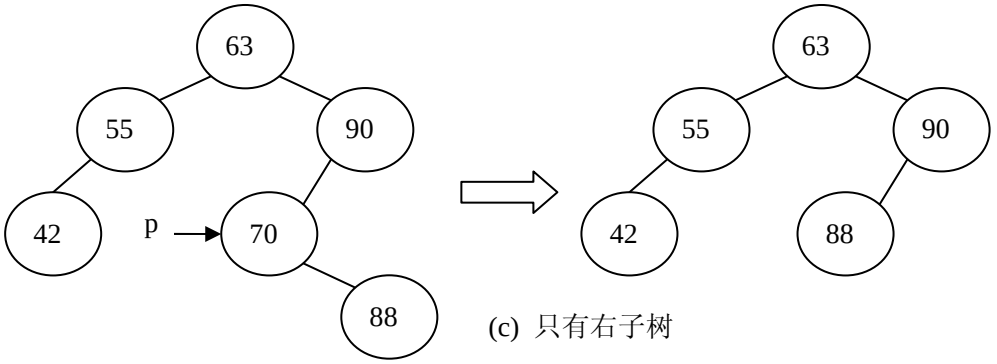
二叉排序树的删除情况如图 8.5 所示。



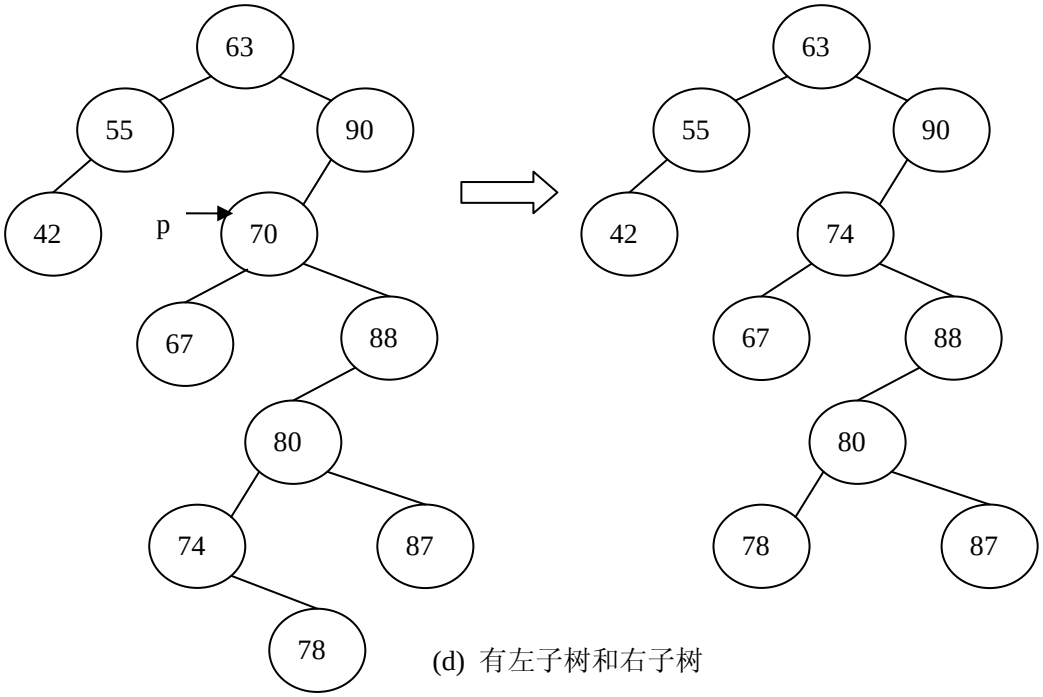
(a) 叶子结点



(b) 只有左子树



(c) 只有右子树



(d) 有左子树和右子树

图 8.5 二叉排序树的删除

二叉排序树的删除算法实现如下：

```
public int Delete(BiTree<int> bt, int key)
{
     Node<int> p;
     Node<int> parent = new Node<int>();
     Node<int> s = new Node<int>();
     Node<int> q = new Node<int>();

     //二叉排序树为空
     if (bt.IsEmpty() == true)
     {
          Console.WriteLine("The Binary Sorting is empty!");
          return 1;
     }

     p = bt.Head;
     parent = p;
     //二叉排序树非空
     while (p != null)
     {
          //存在关键码等于key的结点
          if (p.Data == key)
          {
              //结点为叶子结点
              if (bt.IsLeaf(p))
```

```
{
    if (p == bt.Head)
    {
        bt.Head = null;
    }
    else if (p == parent.LChild)
    {
        parent.LChild = null;
    }
    else
    {
        parent.RChild = null;
    }
}
//结点的右子结点为空而左子结点非空
else if ((p.RChild==null) && (p.LChild!=null))
{
    if (p == parent.LChild)
    {
        parent.LChild = p.LChild;
    }
    else
    {
        parent.RChild = p.LChild;
    }
}
//结点的左子结点为空而右子结点非空
else if ((p.LChild==null) && (p.RChild!=null))
{
    if (p == parent.LChild)
    {
        parent.LChild = p.RChild;
    }
    else
    {
        parent.RChild = p.RChild;
    }
}
//结点的左右子结点均非空
else
{
    q = p;
    s = p.RChild;
    while (s.LChild != null)
```

```
                    {
                        q = s;
                        s = s.LChild;
                    }
                    p.Data = s.Data;
                    if (q != p)
                    {
                        q.LChild = s.RChild;
                    }
                    else
                    {
                        q.RChild = s.RChild;
                    }
                }
                return 0;
            }
            //待删除记录的关键码大于结点的关键码
            else if (p.Data < key)
            {
                parent = p;
                p = p.RChild;
            }
            //待删除记录的关键码小于结点的关键码
            else
            {
                parent = p;
                p = p.LChild;
            }
        }
        return -1;
    }
```

对给定序列建立二叉排序树，若左右子树分布均匀，则其查找过程类似于有序表的折半查找。若给定的序列原来是有序的，则建立的二叉排序树就蜕化为单链表，则其查找的效率与单链表相同。因此，应该对二叉排序树进行调整，特别是当进行了插入或删除之后，使得二叉排序树的左右子树分布均匀。平衡二叉树就是一棵左右子树分布均匀的二叉排序树。

5、平衡二叉树

平衡二叉树又称 AVL 树，其或者是一棵空树，或者是一棵具有下列性质的二叉排序树：

（1）树中结点的平衡因子的绝对值不超过 1；

（2）左右子树均是平衡二叉树。

平衡因子是二叉排序树中结点的左子树的深度减去它的右子树的深度。

平衡二叉树的示例如图 8.6 所示。

从平衡二叉树的定义可知，平衡二叉树中所有结点的平衡因子只能是-1，0

和 1，所以，其深度和$\log_2 n$是同一数量级（n是平衡二叉树中结点的个数）。因此，它的平均查找长度也和$\log_2 n$是同一数量级。

平衡二叉树为了保持其平衡的特性，当进行了插入或删除操作之后，常常需要进行调整。由于篇幅限制，这里就不做介绍。

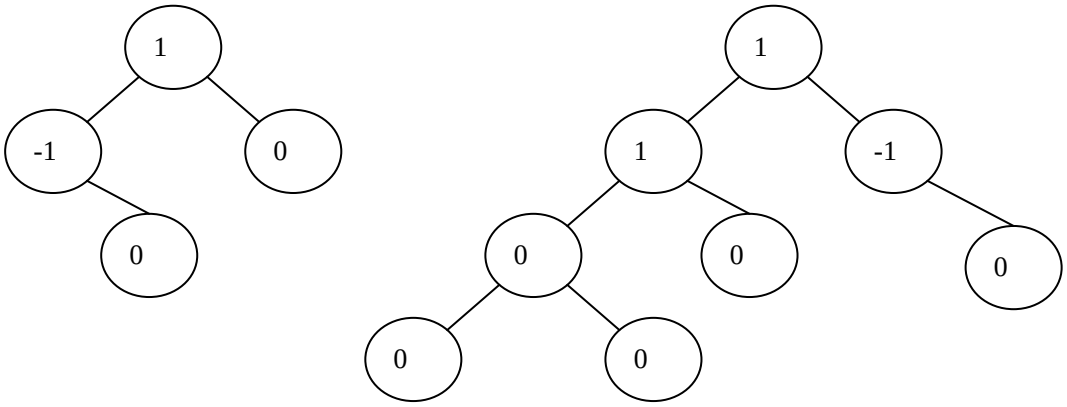


图 8.6 平衡二叉树

8.4 哈希表

在前面介绍的静态查找表和动态查找表中，记录在查找表中的存放位置是随机的，与记录的关键码之间没有关系，所以，查找时需要进行一系列的关键码比较才能确定被查记录在查找表中的位置，即查找算法是建立在关键码比较的基础之上的，查找效率由比较的范围决定。如果能构造一个查找表，使记录的存放位置与记录的关键码之间存在某种对应关系，则可以直接由记录的关键码得到记录的存放位置，则查找可以很快完成，这样查找的效率将得到极大的提高。哈希表(Hash Table)就是这样一种查找表，记录的关键码与记录存放位置之间的映射函数就是哈希函数。因此，哈希表就是通过哈希函数来确定记录存放位置的一种数据结构。

8.4.1 哈希表的基本概念

哈希表也叫散列表，其构造方法是：对于 n 个记录，设置一个长度为 m（m≥n）地址连续的查找表，通过一个函数 H，将每个记录的关键码映射为查找表中的一个单元的地址，并把该记录存放在该单元中。这样的查找表就是哈希表，函数 H 就是哈希函数，它实际是记录的关键码到内存单元的映射，因此，哈希函数的值称为哈希地址。从哈希表的构造方法可知，构造哈希表时一定要使用记录的主关键码，不能使用次关键码。

但是存在这样的问题，对于两个不同的关键码k_i和k_j（$i \neq j$），有$H(d_i)=H(d_j)$，这种情况称为哈希冲突。通常把具有不同关键码而具有相同哈希地址的记录称作“同义词”，由同义词引起的冲突称作同义词冲突。在哈希表中，同义词冲突是很难避免的，除非记录的关键码的变化范围小于哈希地址的变化范围。例如，一个标识符至多由 4 个英文字母政策，则不同的标识符可能有

$$26^4+26^3+26^2+26=475254 \text{ (个)}$$

如果一个标识符对应一个存储地址，那就不会发生冲突了，但这是不可能也没有必要，因为存储空间难以满足，并且，这样会导致当关键码取值不连续时非常浪费存储空间。一般情况下，记录的关键码的变化范围远远大于哈希地址的变化范围。

解决哈希冲突的方法有很多，其基本思想是：当发生哈希冲突时，通过哈希冲突函数来产生一个新的哈希地址，使得原为同义词的记录的哈希地址不同。哈

希冲突函数产生的哈希地址仍有可能发生冲突，此时再使用新的哈希冲突函数得到新的哈希地址，直到不存在哈希冲突为止。这样就把要存放的 n 个记录通过哈希函数映射到了 m 个连续的内存单元中，从而完成了哈希表的建立。

显然，一旦哈希表建立，在哈希表中进行查找的方法就是以要查找的记录的关键码为自变量，使用哈希函数得到一个哈希地址，比较要查找记录的关键码与哈希地址中的记录的关键码，如果二者相等，则查找成功；否则，使用哈希冲突函数得到一个新的哈希地址，再比较二者的关键码，如果二者相等，则查找成功，否则，再以新的哈希冲突函数得到新的哈希地址，继续比较，直到查找成功或者使用完所有的哈希冲突函数都查找失败为止。

【例 8-3】已知 12 个记录的关键码序列为（12，22，25，38，15，47，29，16，21，67，78，56），试构造哈希表存放这 12 个记录。

设计哈希函数为 H(key)=key mod 12，key 为记录的关键码，哈希表的内存空间为 12 个存储单元，建立的哈希表如下所示。

0	1	2	3	4	5	6	7	8	9	10	11
12	25	38	15	16	29	78	67	56	21	22	47

8.4.2 常用的哈希函数构造方法

对于哈希法，主要考虑两个问题，其一是如何构造哈希函数，其二是如何解决哈希冲突。对于如何构造哈希函数，应解决两个主要问题：

（1）哈希函数应是一个压缩映像函数，它应具有较大的压缩性，以节省存储空间；

（2）哈希函数应具有较好的散列性，冲突是不可避免的，但应尽量减少。也就是使哈希函数尽可能均匀地把记录映射到各个存储地址上，这样可以提高查找效率。

构造哈希函数时，一般都要对关键码进行计算，为了尽量避免产生相同的哈希函数值，应使关键码的所有组成成分都能起作用。

常用的哈希函数构造方法如下：

1、直接定址法

H(key)=key 或 H(key)=a*key+b（其中 a、b 为常数）

即取记录关键码的某个线性函数值为哈希地址，这类函数是一一对应函数，不会产生冲突，但要求地址集合与记录关键码集合的大小相同。因此，这种方法适用于记录不多的情况。

【例 8-4】记录序列的关键码为（5，10，15，20，25，30，35，40，45，50，55，60），选取哈希函数为 H(key)=key/5-1，哈希表的内存空间为 12 个存储单元，建立的哈希表如下所示：

0	1	2	3	4	5	6	7	8	9	10	11
5	10	15	20	25	30	35	40	45	50	55	60

2、除留余数法

H(key)=key mod r （r 是一个常数）

即取记录的关键码除以 r 的余数作为哈希地址。使用除留余数法时，r 的选取很重要。若哈希表的表长为 len，则要求 r 接近或等于 len，但不大于 len。r 一般选取质数或不包含小于 20 的质因数的合数。例 8-3 的哈希函数就是采用除留余数法构造的。

3、数字分析法

假设记录的关键码是以 r 为基的数（如：以 10 为基的十进制数），并且哈希

表中可能出现的关键码都是事先可知的，则可取关键码的若干数位作为哈希地址。例如，有 80 个记录，要构造的哈希表长度为 100。不失一般性，取其中 8 个记录的关键码进行分析，8 个关键码如下所示：

1 2 3 1 0 3 4 6
1 2 5 6 0 7 8 3
1 1 4 2 0 1 2 8
2 1 1 3 1 5 1 7
1 2 8 5 0 4 3 5
2 2 2 4 1 6 5 1
2 1 7 8 1 8 7 4
2 1 6 0 0 0 0 2

分析上述 8 个关键码可知，关键码从左到右的第 1、2、5 位的取值比较集中，不宜作为哈希地址，剩余的第 3、4、6、7、8 位取值比较均匀，可选取其中的两位作为 哈希地址。设选取第 6、7 位作为哈希地址，则 8 个关键码的哈希地址为 34、78、12、51、43、65、87、02。

4、平方取中法

顾名思义，取关键码的平方后的中间几位作为哈希地址。这是一种较常用的构造哈希函数的方法。但是，在选定哈希函数时不一定能知道关键码的全部情况，取其中的哪几位也不一定合适，而一个数平方后的中间几位数与数的每一位都相关，所以，随机分布的关键码得到的哈希地址也是随机的。取的位数由表长决定。

8.4.3 处理冲突的方法

哈希法不可避免地会出现冲突，所以应用哈希法时，关键的问题是如何解决冲突。解决冲突的方法基本上有两大类。一类是开放地址法。当发生冲突时，用某种方法形成一个探测的序列，沿着这个序列一个个单元地查询，直到找到这个关键码或找到一个开放的地址（即没有进行存储的空单元）为止。另一类是链表法。当发生冲突时， 拉出一条链，建立一个链接方式的子表，使具有相同哈希函数值的关键码被链接在一个子表中。

1、开放地址法

开放地址法是指当由关键码得到的哈希地址一旦产生冲突，即该地址中已经存放了记录，就去寻找下一个哈希地址，直到找到空的哈希地址为止。只要哈希表足够大，空的哈希地址总能找到，并将记录存入。

开放地址法很多，这里介绍几种。

（1）线性探测法。线性探测法是从发生冲突的地址开始，依次探测下一个空闲的地址（当到达哈希表的表尾时，又从哈希表的头开始探测），直到找到一个空闲单元为止（当哈希表的长度不小于记录的个数，一定可以找到空闲的地址单元）。

线性探测法容易产生堆积问题，这是由于当连续出现 i 个同义词后（设第一个同义词占用的单元为 d,则后面的同义词占用的单元是 d+1，d+2，…，d+i-1），此时随后任何到 d+1，d+2，…，d+i-1 单元上的哈希映射都会由于前面的同义词的堆积而产生冲突。

【例 8-5】已知 9 个记录的关键码为（12，22，25，38，24，47，29，16，36），试构造哈希表存放这 9 个记录。

设计哈希函数为 H(key)=key mod 12，哈希表的内存空间为 12 存储单元，建立的哈希表如下所示。

0	1	2	3	4	5	6	7	8	9	10	11
12	25	38	24	16	29	36				22	47

12、22、25、38 均是由哈希函数得到的没有冲突的哈希地址而直接存入的，但 H(24)=0 与 H(12)相冲突，所以，探测下一个哈希地址，而地址单元 1 和 2 都已经存放记录，直到探测到地址单元 3 为空闲单元，把 24 存入地址单元 3 中。同样 H(36)=0 也出现冲突，依次探测，找到地址单元 6 为空闲，把 36 存入地址单元 6 中。

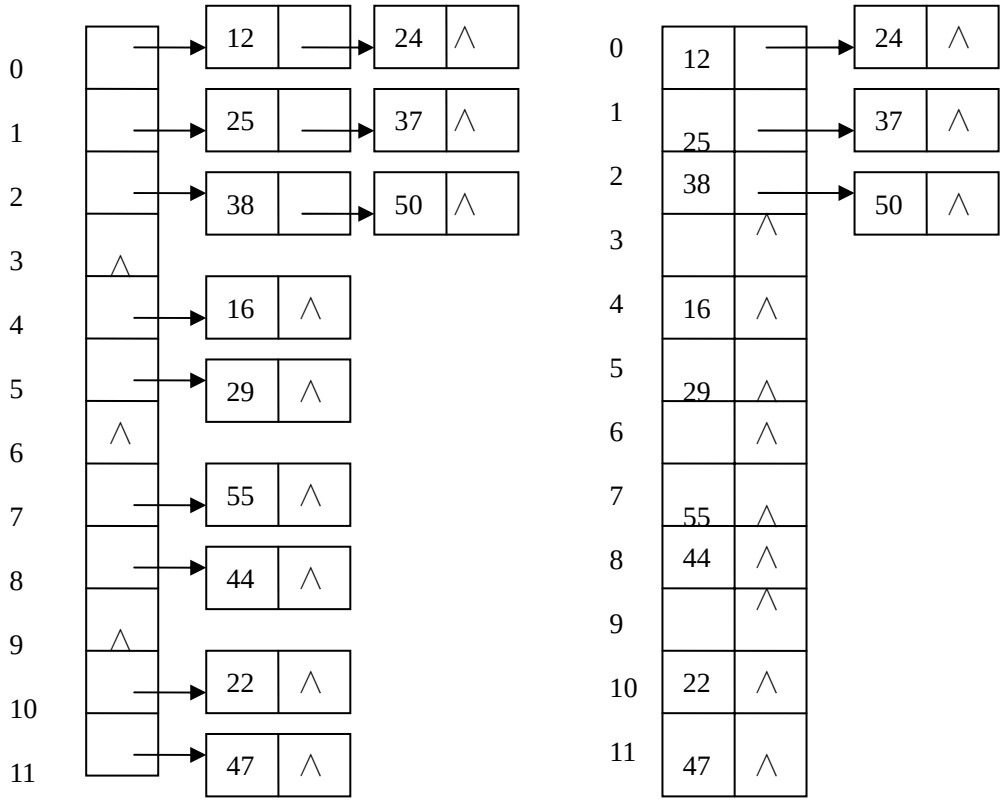
（2）二次探测法。线性探测法由于是一个地址单元一个地址单元探测，探测的增量是线性的，所以容易产生堆积问题。如果探测的增量以平方增加，比如说是 2^2、3^2等，则产生堆积的概率会大大减少，这就是二次探测法。在例 8-5 中，当H(24)与H(12)相冲突时，把增量变为 2^2，则H(24)=0+4=4，则只需探测一次就可以定位，16、36 的处理也是如此。

3、链表法

用链表法解决哈希冲突有两种方法：第一种方法是把所有的同义词用单链表链接起来；第二种方法是当哈希表中相应位置为空时直接存放，当哈希表中相应位置非空时用单链表链接起来。

【例 8-5】已知 12 个记录的关键码为（12，22，25，38，24，47，29，16，37，44，55，50），试构造哈希表存放这 12 个记录，哈希函数用除留余数法，解决哈希冲突用链表法。

由于记录的个数是 12，所以哈希表的长度为 12，哈希函数为 H(key)=key mod 12，构造的哈希表如图 8.7 所示。



(a) 采用第一种方法的链表法　　(b) 采用第二种方法的链表法

图 8.7 用链表法解决哈希冲突

8.5 C#中的查找方法

C#语言中的许多类都提供了查找方法。比如，在.NET Framework 2.0 中新增的泛型数组类Array就提供了查找方法Find。Array类的Find方法是一个静态方法，由Array类直接调用，功能是在数组内查找满足条件的元素，并返回找到的第一个元素，否则，返回空。Find方法的时间复杂度是O(n)，n是数组的长度。又比如，前面介绍的泛型List类和泛型LinkedList类也提供了查找方法Find。泛型List类的Find方法是在List中搜索与指定谓词所定义的条件相匹配的元素，并返回整个 List 中的第一个匹配元素；否则返回元素类型的默认值。泛型LinkedList类的Find方法在LinkedList中查找包含指定值的第一个节点，如果找到，则为包含指定值的第一个 LinkedListNode；否则为空引用。

Windows Form 中的RichTextBox控件也提供了查找的成员方法。该方法搜索指定的文本，并返回搜索字符串的第一个字符在控件中的位置。如果返回负值，则未在控件的内容中找到所要搜索的文本字符串。可以使用此方法创建可提供给控件用户的搜索功能，还可以使用此方法搜索要替换为特定格式的文本。例如，如果用户在控件中输入了日期，便可以在使用控件的 SaveFile 方法之前用 Find 方法在文档中搜索所有日期并将它们替换为适当的格式。

C#中实现了查找方法的类还有很多，比如许多视图类和集合类等都提供了查找方法。由于篇幅限制，这里就不一一列举了，感兴趣的读者可以查阅.NET 的有关书籍。

本章小结

查找又称检索，是在程序设计中对数据结构中的记录进行处理时经常采用的一种操作。同排序一样，查找是对关键码进行处理，关键码分为主关键码和次关键码，以主关键码进行的查找是最经常、也是最主要的查找。

查找有静态查找和动态查找两种，静态查找只在数据结构里查找是否存在某个记录而不改变数据结构。实现静态查找的数据结构称为静态查找表；动态查找要在查找过程中插入数据结构中不存在的记录，或者从数据结构中删除已存在的记录。实现动态查找的数据结构称为动态查找表。衡量查找算法的标准是平均查找长度，它是指在查找过程中进行的关键码比较次数的平均值。实现动态查找的数据结构称为动态查找表。

静态查找表的查找方法主要有顺序查找、折半查找和索引查找等。顺序查找不要求查找表中的记录有序，效率不是很高，适合于记录不是很多的情况。折半查找要求查找表中的记录有序，查找效率很高，适合于记录比较多的情况。索引查找要求查找表分段有序，适合于记录非常多的情况。动态查找表主要介绍了二叉排序树。二叉排序树是一棵二叉树，其左子树结点关键码的值小于根结点关键码的值，右子树结点关键码的值大于根结点关键码的值。二叉排序树上的操作主要有查找、插入和删除等操作。

在哈希表中查找记录不需要进行关键码的比较，而是通过哈希函数确定记录的存放位置。哈希函数的构造方法很多，主要有直接定址法、除留余数法、数字分析法和平方取中法等。由于同义词会产生哈希冲突，解决哈希冲突的方法主要有开放地址法和链表法等，其中开放地址法主要有线性探测法和二次探测法等。

习题八

8.1 试解释以下概念：

静态查找　静态查找表　动态查找　动态查找表　平均查找长度

8.2 以顺序表作为静态查找表实现顺序查找算法，并将监视哨设在顺序表的高

端。

8.3 记录按关键码排列的有序表（6，13，20，25，34，56，64，78，92），采用折半查找，画出判定树，并给出查找关键码为 13 和 55 的记录的过程。

8.4 编写一个算法，利用折半查找算法在一个有序表中插入一个记录（关键码为 x），并保持表的有序性。

8.5 已知一个长度为 12 的记录的关键码序列为（37，7，32，29，20，28，22，15，17，23，1，9），要求：

（1）按各记录的顺序构造一棵二叉排序树。

（2）在（1）的基础上插入关键码为 41 的记录，画出对应的二叉排序树。

（3）在（2）的基础上删除关键码为 29 的记录，画出对应的二叉排序树。

8.6 有 n 个结点的二叉排序树共有多少种不同的形态？

8.7 对于题 8.5 的记录序列构造两个哈希表。

（1）采用哈希函数 H(Key)=key%12，用线性探测法处理冲突；

（2）用链表法处理冲突。

8.8 在哈希表的存储结构中，发生冲突的可能性与哪些因素有关？为什么？

参考文献

[1]余绍军．数据结构．长沙：中南大学出版社，2004

[2]黄国瑜，叶乃菁编著．数据结构(java 语言版) ．北京：清华大学出版社，2002

[3]严蔚敏，吴伟民．数据结构(C 语言版) ．北京：清华大学出版社，1997

[4]Sara Baase，Allen Van Gelder．计算机算法——设计与分析导论（第三版）．北京：高等教育出版社，2001

[5][美]Clifford A.Shaffer 著．张铭，刘晓丹译．数据结构与算法分析(Java 版) ．北京：电子工业出版社，2001

[6]朱站立编著．数据结构．西安：西安电子科技大学出版社，2002

[7]李益明，邓文华主编．数据结构(C 语言版) ．北京：电子工业出版社，2004

[8]郑宇军编著．C#2.0 程序设计教程．北京：清华大学出版社，2004

[9][美]Kris Jamsa 著．张春晖，李越等译．C/C++/C#程序员适用大全——C/C++/C#最佳编程指南．北京：中国水利水电出版社，2002

[10](美)Jeffrey Richter 著．李建忠译．.NET 框架程序设计（修订版）．北京：清华大学出版社，2003

[11][美]Mickey Williams 著．冉晓旻，罗邓，郭炎译．Visual C#.NET 技术内幕．北京：清华大学出版社，2003

[12]朱儒荣，朱辉编著．数据结构常见题型分析及模拟题．西安：西北工业大学出版社，2000

[13] Bruno R，Preiss，P.Eng．Data Structures and Algorithms with Object-Oriented Design Patterns in C#．http://www.brpreiss.com