新编计算机类本科规划教材

数据结构(Java 版)

叶核亚　编著
陈本林　主审

電子工業出版社
Publishing House of Electronics Industry
北京・BEIJING

内 容 简 介

本书全面阐述数据结构方面的基本理论，主要内容包括线性表、栈、队列、串、数组、广义表、树、叉树、图等基本的数据结构以及查找和排序算法。

本书用 Java 语言定义和实现数据结构及算法。全书结构清楚，内容丰富，章节安排合理。叙述深入浅出循序渐进。示例典型实用，算法严谨规范，算法和程序全部调试通过。

本书适合作为高等院校计算机及相关专业本、专科学生教材，也可作为从事计算机软件开发和工程应人员的参考书。

图书在版编目(CIP)数据

数据结构（Java 版）/叶核亚编著. —北京：电子工业出版社，2004.5
新编计算机类本科规划教材
ISBN 7-5053-9857-1

Ⅰ.数… Ⅱ.叶… Ⅲ.①数据结构—高等学校—教材②JAVA 语言—程序设计—高等学校—教材 Ⅳ.
TP311.12 ②TP312
中国版本图书馆 CIP 数据核字（2004）第 034213 号

策划编辑：章海涛
责任编辑：章海涛
印　　刷：北京四季青印刷厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编　100036
经　　销：各地新华书店
开　　本：787×1092　1/16　印张：15.75　字数：400 千字
印　　次：2004 年 5 月第 1 次印刷
印　　数：1～5 000 册　　定价：19.50 元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系
联系电话：（010）68279077。质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件
dbqq@phei.com.cn。

前　言

“数据结构”课程在计算机类专业学生培养中的地位十分重要。在计算机中，现实世界被抽象为数据或数据模型。“数据结构”课程的任务是讨论数据的各种逻辑结构和在计算机中的存储结构，以及各种操作的算法设计。“数据结构”课程的主要目的是培养学生掌握处理数据和编写高效率软件的基本方法，从而为以后进行软件开发和应用及进一步学习后续专业课程打下坚实的基础。“数据结构”课程讨论的知识内容是软件设计的理论基础，数据结构课程介绍的技术方法是软件设计中使用的基本方法。因此，“数据结构”课程是计算机学科本科（专科）的核心课程，是培养程序设计技能的必不可少的重要环节。

本书具有以下特点：

1．内容全面，叙述准确

本书介绍了线性表、栈、队列、串、数组、广义表、树和二叉树、图等基本的数据结构，阐明了数据模型的逻辑结构，讨论了它们在计算机中的存储结构以及能进行的各种操作和这些操作的具体实现。另外，本书讨论了软件设计中应用频繁的查找和排序问题，给出了多种经典查找和排序算法。

本书以基本数据结构和经典算法为主线，全面、准确地阐述了“数据结构”课程的主要内容。本书概念清楚，用语规范，思路清晰。书中示例选择合适，算法分析透彻，程序结构严谨、规范，体现了良好的程序设计素养。

2．采用 Java 语言描述数据结构

随着计算机技术日新月异的变化及网络化的发展趋势，Java 语言以其完全面向对象、简单高效、与平台无关、支持多线程、具有安全性和健壮性等特点，已成为目前最具吸引力且发展势头迅猛的程序设计语言。

Java 语言是完全面向对象的，具有诸多传统程序设计语言无法比拟的优点。Java 语言的语法类似 C++语言，但除去了 C++中那些模糊和容易引起错误的特性，同时引入了很多独特的高级特性。Java 语言比 C++更容易学习，且用 Java 语言编写的程序可读性更好。

本书用 Java 语言定义和实现了全部的数据结构。

3．具有丰富翔实的例程

本书注重理论与实践相结合，注重基本知识的传授与基本技能的培养。本书除例题外，还安排了思考题、课堂练习题、上机实习题以及综合程序设计题等实践环节。这些为巩固读者的理论知识，提高读者的软件设计能力起到了辅助作用，为读者以后进行系统软件和应用软件的开发研究打下了坚实的基础。

书中的所有程序全部调试通过。

本书由叶核亚主编，陈立参编。其中叶核亚编写第 1、2、3、4、6、7、8、9 章，陈立编写第 5 章。本书由陈本林教授主审。陈教授认真细致地审阅了全书，并提出了许多宝贵意见。本书在编写过程中还得到了廖雷、陈瑞、陈建红、阚建飞、李林广、王青云、刁翔等老师的大力帮助，在此一并表示感谢。

尽管作者写作时非常认真和努力，但由于水平有限，时间紧迫，书中难免有疏漏之处，
敬请读者批评指正。

为了使用本书的教师和读者的需要，我们将免费提供本书中所有的图、表和程序，请发
E-mail 至 unicode@phei.com.cn索要。

编著者

2004 年 3 月

目　录

第 1 章　绪　　论

软件设计是计算机学科各个领域的核心。软件设计时要考虑的首要问题是数据的表示、组织和处理方法，因为数据的表示、组织和处理方法直接关系到软件的工程化程度和软件的运行效率。

随着计算机技术的飞速发展，计算机的应用从早期的科学计算扩大到控制、管理和数据处理等各个领域。计算机处理的对象也从简单的数值数据，发展到多媒体数据。各种软件系统处理的数据量越来越大，数据的类型越来越多，数据的结构越来越复杂。因此，针对实际问题，如何合理地组织数据，如何建立合适的数据结构，如何设计优秀的算法，则是软件系统设计的重要问题，而这些就是“数据结构”这门课程讨论的主要内容。

数据结构设计和算法设计是软件系统设计的核心。在计算机领域流传着一句经典名言，就是“数据结构+算法=程序”（瑞士 Niklaus Wirth 教授）。这句话简洁明了地说明了程序（或软件）和数据结构与算法的关系，也简洁明了地说明数据结构课程的重要性。

作为绪论，本章将讨论后续章节中频繁使用的数据、数据类型、数据结构、算法等基本概念，以及算法分析的基本方法。

建议本章授课 4 学时。

1.1　数据结构的基本概念

1.1.1　抽象数据类型与数据结构

1．数据

描述客观事物的数字、字符以及所有能输入到计算机中并能为计算机接受的各种符号集合统称为数据（data）。数据是程序的处理对象。例如，学生成绩管理程序处理的数据是每个学生的情况，包括学号、姓名、年龄、各科成绩等，编译程序处理的数据是用各种高级程序设计语言书写的源程序。近年来，随着技术的进步，数据的形式越来越多，如多媒体技术中涉及的视频和音频信号，经采集转换后都能形成计算机可接受和处理的数据。

表示一个事物的一组数据称为一个数据元素（data element）。数据元素是数据的基本单位，构成数据元素的数据称为该数据元素的数据项（data item）。

2．数据类型

类型是一组值的集合。数据类型（data type）是指一个类型和定义在该类型上的操作集合。数据类型定义了数据的性质、取值范围以及对数据所能进行的各种操作。例如，Java 中

整数类型 int 的值集是{-2^{32}，…，-2，-1，0，1，2，…，2^{32}-1}，还包括对这个整数类型
行的加（+）、减（-）、乘（*）、除（/）和求模（%）操作。

高级程序设计语言都提供了一些基本数据类型，如 Java 中有 int、long、float、double、cha
String 等基本数据类型。利用这些基本数据类型，软件设计人员还可以设计出各种复杂的数
类型。例如，学生姓名可以用字符串类型 String 表示，年龄可以用整数类型 int 表示，成绩
以用浮点类型 float 表示等，而包括学号 、姓名 、成绩等的学生情况就是一种复杂的数据类型

3．抽象数据类型

抽象数据类型（Abstract Data Type，ADT）是指一个逻辑概念上的类型和该类型上的操
集合。

没有定义具体数据类型的数据元素称为抽象数据元素。

从定义看，数据类型和抽象数据类型的定义基本相同。数据类型和抽象数据类型的不同
处仅仅在于：数据类型指的是高级程序设计语言支持的基本数据类型，而抽象数据类型指的
在基本数据类型支持下用户新设计的数据类型。“数据结构”课程主要讨论表、栈、队列、串
数组、树和二叉树、图等典型的数据结构，这些典型的数据结构就是一个个不同的抽象数据
型。

例如，要描述学生信息，可包括学生的学号、姓名
性别、年龄等数据。学生的学号、姓名、性别、年龄
数据就构成学生情况描述的数据项；包括学号、姓名
性别、年龄等数据项的一组数据就构成学生信息的一
数据元素。表 1-1 是一个有 3 个数据元素的学生信息表

表 1-1　学生信息表

学号	姓名	性别	年龄
20020001	王红	男	18
20020002	张明	男	19
20020003	吴宁	女	18

本书采用 Java 语言描述抽象数据类型，抽象数
元素可以声明为 Java 语言中的 Object 类。表 1-1 的学生信息表可以声明为如下的类，其中
据元素类型是 Java 的基本数据类型。

```
class Student
{
 String number;
 String name;
 String sex;
 int age;
}
```

4．数据结构

计算机处理的数据不是杂乱无章的，而是有着内在联系的。只有分析清楚它们的内在联系
对大量的、复杂的数据才能进行合理地组织和有效地处理。例如，由一个班级的学生组成的
批数据，按学号排列具有“顺序”关系，按班长、组长、组员排列具有“层次”关系。这种
系不因数据的改变而改变。

对一个数据元素集合来说，如果在数据元素之间存在一种或多种特定的关系，则称为数
结构（data structure）。因此，“结构”就是指数据元素之间存在的关系。

数据结构与数据类型是两个不同的概念。数据类型研究的是每类数据所共有的特性，关
的是数据集合是怎样的，该数据集合上允许进行的操作有哪些；数据结构研究的是相互关联

数据之间的关系，数据结构关注数据之间的关系是怎样的。例如，一个班级的学生按学号排列就是顺序关系，按班长、组长、组员排列就具有层次关系；而祖父、父亲、我、儿子、孙子间的辈份关系也是层次关系。

1.1.2 数据的逻辑结构

数据结构课程主要讨论三方面的问题：数据的逻辑结构、数据的存储结构和数据的操作。

数据的逻辑结构是对数据元素之间逻辑关系的描述，它可以用一个数据元素的集合和定义在此集合上的若干关系来表示，数据的逻辑结构经常被简称为数据结构。

按照数据元素之间存在的逻辑关系的不同数学特性，基本的数据结构有 3 种：线性结构、树结构和图结构。

1．线性结构

线性结构的定义是：除第一个和最后一个数据元素外，每个数据元素只有一个前驱数据元素和一个后继数据元素，第一个数据元素没有前驱数据元素，最后一个数据元素没有后继数据元素。

线性结构如图 1.1（a）所示，其中数据元素 B 有一个前驱数据元素 A，有一个后继数据元素 C。

2．树结构

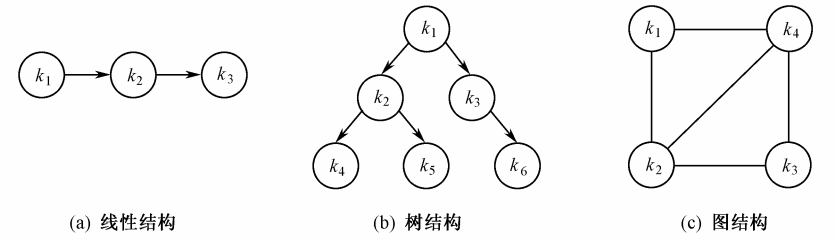
树结构的一般定义是：除根结点外，每个数据元素只有一个前驱数据元素，可有零个或若干个后继数据元素，根结点没有前驱数据元素。

树结构如图 1.1（b）所示，其中数据元素 B 有一个前驱数据元素 A，有两个后继数据元素 D 和 E。

3．图结构

图结构的一般定义是：每个数据元素可有零个或若干个前驱数据元素，可有零个或若干个后继数据元素。

图结构如图 1.1（c）所示，其中数据元素 E 有两个前驱数据元素 B 和 C。



(a) 线性结构　　(b) 树结构　　(c) 图结构

图 1.1　三种基本的数据结构

图 1.1 以图示法表示数据的逻辑结构。图示法中，圆圈表示一个数据元素，圆圈中的字符表示数据元素的标记或值，连线表示数据元素间的关系。

1.1.3 数据的存储结构

任何需要计算机进行管理和处理的数据元素都必须首先按某种方式存储在计算机中。数

据元素在计算机中的存储表示方式称为数据的存储结构，也称为物理结构。数据的存储结构要能正确地表示出数据元素间的逻辑关系。

数据的逻辑结构是从逻辑关系角度观察数据，它与数据的存储无关，是独立于计算机的而数据的存储结构是逻辑结构在计算机中的实现，它是依赖于计算机的。

1．两种存储结构

数据存储结构的基本形式有 2 种：顺序存储结构和链式存储结构。

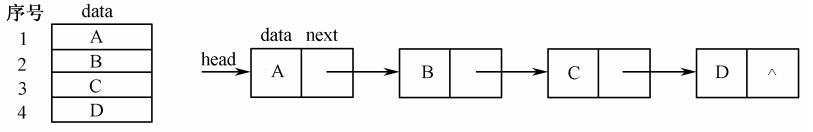
顺序存储结构是把数据元素存储在一块地址连续的空间中，其特点是逻辑上相邻的数据元素在物理上也相邻，数据间的逻辑关系表现在数据元素的存储位置关系上。

例如，采用高级程序设计语言中的数组可以实现顺序存储结构，数组元素之间的顺序体了线性结构数据元素之间的逻辑次序。

指针是指向物理存储单元地址的变量。由数据元素域和指针域组成的一个整体称为一结点（node）。链式存储结构是使用指针把相互直接关联的结点（即直接前驱结点或直接后结点）连接起来，其特点是逻辑上相邻的数据元素在物理上（即内存存储位置上）不一定相邻数据间的逻辑关系表现在结点的链接关系上。

就像自行车的链条是由每一节链串起来的一样，结点在链条中的位置可根据需要重新组形成新的链条。

对于数据元素是{A，B，C，D}的线性结构，其顺序和链式存储结构如图 1.2 所示。



(a) 顺序存储结构　　(b) 链式存储结构

图 1.2　两种存储结构

在顺序存储结构中，所有的存储空间都被数据元素占用了；而链式存储结构中，每个结除了要保存数据元素外，还要保存指向后继结点的链信息。因此，每个结点至少由两部分组成数据域——保存数据元素，链——指向后继结点。

【例 1.1】 学生信息表的两种存储结构。

一个班级的学生信息表按学号排列起来，构成一个线性表，其顺序与链式存储结构如1.3 所示。

顺序存储结构和链式存储结构是两种最基本、最常用的存储结构。除此之外，利用顺序存储结构和链式存储结构进行组合，还可以有一些更复杂的存储结构。

2．存储密度

如果所有的存储空间都用来存储数据元素，则这种存储结构是紧凑结构，否则称为非紧结构。显然，顺序存储结构是紧凑结构，链式存储结构是非紧凑结构。

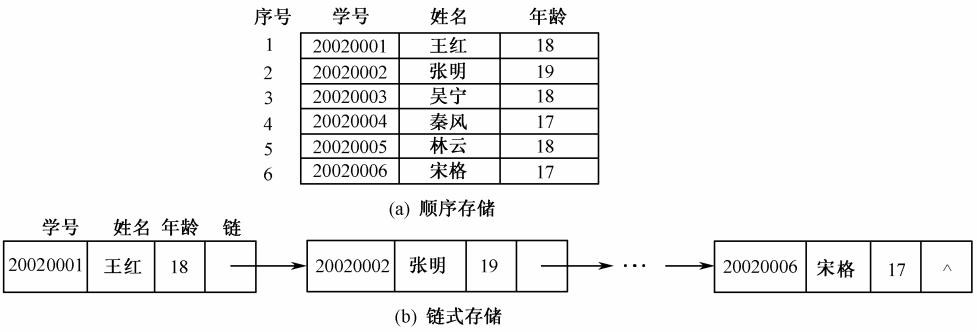


图 1.3　学生信息表的两种存储结构

结构的存储密度定义为数据本身所占的存储量和整个结构所占的存储总量之比，即

$$d=\frac{\text{数据本身所占的存储量}}{\text{整个结构所占的存储总量}}$$

紧凑结构的存储密度为 1，非紧凑结构的存储密度小于 1。存储密度越大，则存储空间的利用率越高。但是存储附加的信息会带来操作上的方便。

3．选择数据的存储结构

在逻辑结构的基础上，选择一种合适的存储结构，使得在以下两方面的综合性能最佳：数据操作所花费的时间少，占用的存储空间少。

计算机运行任何程序都要花费一定的时间和占用一定的内存空间。例如，存储一个线性表，当不需要频繁插入和删除时，采用顺序存储结构是可行的。因为这时存储密度大，占用的空间少。而当插入和删除操作很频繁时，就需要采用链式存储结构。因为这时虽然占用的空间较多，使存储密度变小，但节省了操作的时间。这是以空间为代价换取了时间，在总的时间和空间性能上仍是可行的。

1.1.4　数据的操作

对一种数据类型的数据进行的某种处理称为数据的操作，对一种数据类型的数据进行的所有操作称为数据的操作集合。

数据的操作是定义在数据的逻辑结构上的，每种逻辑结构都有一个操作集合，不同的逻辑结构有不同的操作。操作的具体实现与存储结构有关。下面列举线性结构、树结构和图结构中几种常用的操作：

- 访问数据元素。
- 统计数据元素个数。
- 更新数据元素值。
- 插入数据元素，在数据结构中增加新的结点。
- 删除数据元素，将指定数据元素从数据结构中删除。
- 查找——在数据结构中查找满足一定条件的数据元素。插入、删除、更新操作都包括一个查找操作，以确定所需插入、删除、更新数据元素的确切位置。
- 排序——在线性结构中数据元素数目不变的情况下，将数据元素按某种指定的顺序重新排列。

【例 1.2】 学生信息表的操作。

对图 1.3 所示的学生信息表进行操作，结果如下：

- 访问学号为“ 20020003 ”的数据元素，得到学生姓名为“ 吴宁 ”，年龄为 18。
- 统计数据元素个数，共 6 个学生。
- 更新学号为“ 20020005 ”、年龄值为 20 的数据元素（如图 1.4（a）所示）。
- 插入学号为“ 20020007 ”、姓名为“ 刘强 ”、年龄为 18 的数据元素（如图 1.4（a）所示）
- 删除学号为“ 20020004 ”的数据元素（如图 1.4（b）所示）。
- 查找年龄等于 20 的数据元素，序号为 4（如图 1.4（b）所示）。
- 按学号排序。

序号	学号	姓名	年龄
1	20020001	王红	18
2	20020002	张明	19
3	20020003	吴宁	18
4	20020004	秦风	17
5	20020005	林云	20
6	20020006	宋格	17
7	20020007	刘强	18

(a) 更改、插入结点

序号	学号	姓名	年龄
1	20020001	王红	18
2	20020002	张明	19
3	20020003	吴宁	18
4	20020005	林云	20
5	20020006	宋格	17
6	20020007	刘强	18
7			

(b) 删除结点

图 1.4 学生信息表的操作

1.2 算法与算法设计

1.2.1 算法

通俗地说，算法是对问题求解过程的一种描述，是为解决一类问题给出的一个确定的、有限长的操作序列。

1．算法定义

曾获图灵奖的著名计算科学家 D.Knuth 对算法做过一个为学术界广泛接受的描述性的定义：一个算法（algorithm），就是一个有穷规则的集合，其规则确定了一个解决某一特定类型问题的操作序列。算法的规则必须满足如下 5 个特征：

- 有穷性——算法必须在执行有穷步骤之后结束。
- 确定性——算法的每一个步骤必须是确切定义的。
- 输入——算法有零个或多个输入。
- 输出——算法有一个或多个输出，与输入有某种特定关系。
- 可行性——算法中有待执行的操作必须是相当基本的，即是说，它们原则上都是能够精确地进行的，而且用笔和纸做有穷次就可以完成。

有穷性和可行性是算法最重要的两个特征。

2．算法的描述

算法可用文字、高级程序设计语言或类似于高级程序设计语言的伪码描述。此时，算法是由语义明确的操作步骤组成的有限序列，它精确地指出怎样从给定的输入信息得到要求的输出信息。

【例 1.3】 学生信息表的顺序查找算法。

在学生信息表中，按“姓名”进行顺序查找（sequential search）的算法思路为：对于给定值 k，从线性表的一端开始，依次比较每个元素的姓名栏，如果存在姓名与 k 相同的数据元素则查找成功；否则查找不成功。

这种方法是可行的，但查找范围大，耗时多，查找不成功时，需要将表中的所有元素全部比较后才能确定。

设学生信息表按顺序存储，按“姓名”查找值“name="宋格"”，顺序查找算法描述如下：

- 设 i=1，比较第 i 个元素的姓名是否等于 name，如果相等，则查找成功，查找过程结束；否则，i++，继续比较。
- 学生信息表中，所有元素的姓名都不等于 name，则查找不成功。

学生信息表的顺序查找过程如图 1.5 所示。

	序号	学号	姓名	年龄	
i=	1	20020001	王红	18	第1个元素的姓名等于name?
	2	20020002	张明	19	第2个元素的姓名等于name?
	3	20020003	吴宁	18	第3个元素的姓名等于name?
	4	20020005	林云	20	第4个元素的姓名等于name?
	5	20020006	宋格	17	第5个元素的姓名等于name? 是，则查找成功
	6	20020007	刘强	18	
	7				

图 1.5 学生信息表的顺序查找

3．算法与数据结构

算法是建立在数据的逻辑结构与存储结构上的。数据结构和算法之间存在着本质的联系，失去一方，另一方就没有意义。在研究一种数据结构时，总是离不开对这种数据结构需要施加的各种操作，而且只有通过对这些操作的算法的研究，才能更清楚地理解这种数据结构的意义和作用。反过来，在研究一种算法时，也总是自然地联系到该算法建立在什么样的数据结构之上。

（1）对于同样的逻辑结构和存储结构，根据问题的不同要求，采用不同的算法。

【例 1.4】 字典的分块查找算法。

一部字典是一个顺序存储的线性表，且已按字母顺序排序，与学生成绩表的逻辑结构和存储结构都相同。因学生成绩表的数据量较小，可以采用顺序查找算法；而字典的数据量较大则必须采用分块查找算法。

例如，在字典中查找单词 state。我们通常先查找索引表，找到以字母 s 开头的页码，再根据 state 后几个字母，就可快速准确地定位 state，查阅 state 的含义。那么，如何知道以字母 s 开头的单词的起始页码？为此，每部字典都设计了一个索引表，指出每个字母对应单词的起始页码（如图 1.6 所示）。

索引表

序号	首字母	起始页
1	a	1
2	b	12
3	c	23
…	…	…
19	s	450
…	…	…

字典

页码	单词
1	aardvark
	abstract
	…
12	baa1
	…
461	state

图 1.6 字典与其索引表示意图

一部字典可以看成是由首字母相同、大小不等的若干块（block）组成，每个块的起始页码由索引表给出。

在字典中查找给定单词 word，分块查找（blocking search）算法必须分两步进行：

- 根据 word 的首字母，查找索引表，确定 word 所在块的起始页码。

➢ 从起始页开始，按顺序查找 word，得到查找成功与否的信息。

（2）同样的数据结构，不同的存储结构，则采用的算法必然不同。

存储结构不同的线性表其排序算法必然不同。例如，冒泡排序、折半插入排序等适用于顺序存储结构的线性表，而不能用于链式存储结构的线性表；适用于链式存储结构线性表的排序算法有直接插入排序、简单选择排序等（详见第 3 章和第 7 章）。

4．算法与程序

算法是用抽象的语言描述解决特定问题的每一步的操作。程序是计算机能理解和执行的指令序列。其中，指令既可以是机器指令、汇编指令，也可以是高级语言的语句命令。

通常，一个程序实现一个算法。算法和程序的区别是，算法的执行是有穷的，而程序的执行可以是无限的。例如操作系统程序，只要系统不受破坏，操作系统的运行将不会停止。

1.2.2 算法设计

1．算法设计目标

算法设计应满足以下 5 条目标：

- 正确性——算法应确切地满足具体问题的需求，这是算法设计的基本目标。
- 可读性——算法的可读性有利于人们对算法的理解，这既有利于程序的调试和维护，也有利于算法的交流和移植。算法的可读性主要体现在两方面：一是类名、对象名、方法名等的命名要见名知意；二是要有足够多的注释。
- 健壮性——输入非法数据时，算法要能做出适当的处理，而不应产生不可预料的结果。
- 高时间效率——算法的时间效率指算法的执行时间。执行时间短的算法称为高时间效率的算法。
- 高空间效率——算法在执行时一般要求额外的内存空间。内存要求低的算法称为高空间效率的算法。

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和内存要求低的算法。算法的高时间效率和高空间效率通常是矛盾的，在目前情况下，首先考虑算法的时间效率目标。

2．算法设计实现

【例 1.5】 求最大值算法的设计与调用。

从两个整数类型数据中得到较大数值的算法设计如下：

```
int Max(int a,int b)
{
   return (a>=b)?a:b;
}
```

主函数中，调用 Max(Max(a,b),c)将从三个整数中返回最大值，如：

```
System.out.println("max="+Max(Max(a,b),c));
```

【例 1.6】 求两个整数的最大公因数的算法实现。

设有不全为 0 的整数 a 和 b，记 gcd(a, b)为它们的最大公因数，即同时能整除 a 和 b 的公因数中的最大者。按照欧几里德（Euclid）的辗转相除算法，gcd(a, b)具有如下性质：

$\gcd(a, b)= \gcd(b, a)$

$\gcd(a, b)= \gcd(-a, b)$ $0 \le a \% b < b$

$\gcd(a, 0)=|a|$

$\gcd(a, b)= \gcd(b, a \% b)$

以循环语句实现算法如下：

```
int gcd1(int a,int b)
{
   int k=0;
   do
   {
      k=a%b;
      a=b;
      b=k;
   }while(k!=0);
   return a;
}
```

以递归函数实现算法如下：

```
int gcd2(int a,int b)
{
 if(b==0)
   return a;
 if(a<0)
   return gcd2(-a,b);
 if(b<0)
   return gcd2(a,-b);
   return gcd2(b,a%b);
}
```

1.2.3 算法分析

1．算法的时间复杂度

算法的执行时间等于所有语句执行时间的总和，是算法所处理的数据个数 n **的函数，表**
为 $O(f(n))$**，**$O(f(n))$**称为该算法的时间复杂度（**time complexity**）。**$O(1)$**表示时间是一个常数，不**
依赖于 n**；**$O(n)$**表示时间与** n **成正比，是线性关系；**$O(n^2)$**、**$O(n^3)$**和** $O(2^n)$**分别称为平方阶、立**
方阶和指数阶；$O(\log_2 n)$**为对数阶。**

通常用算法的时间复杂度来表示算法的时间效率。若两个算法的执行时间分别为 $O(1)$**和**
$O(n)$**，当** n **充分大时，显然** $O(1)$**的执行时间要少。同样，**$O(n)$**和** $O(\log_2 n)$**相比较，当** n **充分大**
时，因 $\log_2 n$ **的值比** n **小，则** $O(\log_2 n)$**所对应的算法速度要快得多。**

【例 1.7**】 分析算法的时间复杂度。**

（1**）一个简单语句的时间复杂度为** $O(1)$**。**

```
sum=0;
```

该语句的执行时间是一常量，时间复杂度为 $O(1)$**。**

（2）一个循环的时间复杂度为 $O(n)$。

```
int n=10,sum=0;
for(int i=0;i<n;i++)
    sum+=n;
```

for 语句循环执行 n 次，其中循环体语句的执行时间是一常量，所以 for 语句的时间复杂度为 $O(n)$。

（3）时间复杂度为 $O(n^2)$的二重循环。

```
int n=9;
for(int i=0;i<n;i++)
    for(int j=0;j<n;j++)
        System.out.print(i*j);
```

外层循环执行 n 次，每执行一次外层循环时，内层循环执行 n 次。所以，二重循环中的循环体语句执行 $n\times n$ 次，时间复杂度为 $O(n^2)$。如果

```
int n=9;
for(int i=0;i<n;i++)
    for(int j=0;j<i;j++)
        System.out.print(i*j);
```

外层循环执行 n 次，每执行一次外层循环时，内层循环执行 i 次。此时，二重循环的执行次数为 $\sum_{i=1}^{n} i=\frac{n(n+1)}{2}$，则时间复杂度仍为 $O(n^2)$。

（4）时间复杂度为 $O(n\log_2 n)$的二重循环。

```
int n=8;
for(int i=1;i<=n;i*=2)
  for(int j=1;j<=n;j++)
        System.out.print(i*j);
```

所以，外层循环执行 $\log_2 n$ 次，外层循环每执行一次，i 就乘以 2，直至 $k>n$。内层循环执行次数恒为 n。此时，总的循环次数为 $\sum_{i=1}^{\log_2 n} n=O(n\log_2 n)$，则时间复杂度为 $O(n\log_2 n)$。

（5）时间复杂度为 $O(n)$的二重循环。

```
int n=8;
for(int i=1;i<=n;i*=2)
   for(int j=1;j<=i;j++)
        System.out.print(i*j);
```

外层循环执行 $\log_2 n$ 次。内层循环执行 i 次，随着外层循环的增长而成倍递增。此时，总的循环次数为 $\sum_{i=1}^{\log_2 n} 2^i=O(n)$，则时间复杂度为 $O(n)$。

时间复杂度随 n 变化情况的比较如表 1-2 所示。

表 1-2　时间复杂度随 n 变化情况的比较

时间复杂度	$n=8(2^3)$	$N=10$	$n=100$	$n=1000$
$O(1)$	1	1	1	1
$O(\log_2 n)$	3	3.322	6.644	9.966
$O(n)$	8	10	100	1000
$O(n\log_2 n)$	24	33.22	664.4	9966
$O(n^2)$	64	100	10 000	10^6

2．算法的空间复杂度

与算法的时间复杂度概念类似，算法在执行时要求的额外内存空间称为算法的空间复杂度，分析从略。

1.3　Java 语言简介

在结构化程序设计中，数据的描述用数据类型表示，对数据的操作用过程或函数表示。例如，在描述栈时，先定义栈的数据类型，再用过程实现对栈的操作，这种方式是可行的，但不符合面向对象的程序设计思想。由于数据的描述和对数据的操作两者是分离的，这将导致程序的重用性差、可移植性差、数据维护困难等缺点。

实际上，在数据结构的理论中，数据的逻辑结构、存储结构以及对数据所进行的操作三者是一体的，是相互依存的，所以用面向对象的封装、继承和多态等类的特性能够更深入地刻画数据结构。例如，将栈声明为一个类 Stack，而入栈、出栈等对栈的操作则声明为类的方法。此时，对设计者而言，需要设计栈这个类（或接口）并实现其中的方法。对使用者而言，只需要知道类中的方法即可使用 Stack 类，因此增强了数据的重用性、可移植性，使数据易于维护。

在面向对象的编程语言中，Java 具有面向对象、简单高效、与平台无关的特点，支持多线程、分布性和并发机制；用 Java 开发的应用软件在可移植性、健壮性、安全性方面大大优于已存在的其他编程语言。Java 提供了丰富的类库，可广泛用于面向对象的事件描述、处理综合等面向对象的开发应用。

Java 提供了对象的自引用方式用以实现数据的链式存储结构，这种方式避免直接使用指针所带来的安全隐患，使 Java 语言可以实现面向对象的数据结构。

1.3.1　Java 的安装、编辑、编译和运行

1．系统需求

基于 Windows 平台的机器要求如下。

- 硬件配置：Pentium II 以上 PC，64MB 以上内存，100MB 以上硬盘。
- 系统运行环境：Windows 98/NT/2000，Internet Explorer 5.0 以上。

2．Java 软件环境的安装

1）安装 Java 的开发环境 JDK 1.3

JDK 1.3.0 有以下一些程序和文档：

- j2sdk1_3_0-win.exe——JDK 开发包，运行它即可安装 Java。

- j2sdk1_3_0-doc.zip——相应的帮助文档，解压缩即可。

以上两个文件可在 http://java.sun.com/products 中下载。安装成功后，需要进行环境配置设 Java 安装在 C:\jdk1.3.0_02 文件夹中，新建 java2.bat 文件，内容如下：

```
set path=C:\jdk1.3.0_02\bin                    //设置系统查找 Java 编译环境的路径
set classpath=.;C:\jdk1.3.0_02\lib             //设置系统查找 Java 包的路径
```

注意：以上述方式设置的 path 将覆盖 path 原有内容，有可能影响其他程序运行。通常法是在 path 原有内容的基础上添加新路径。

Windows 98 下，在 C:\autoexec.bat 文件中添加新路径，将“;C:\jdk1.3.0_02\bin”添加到 path 原有内容之后，或添加以下命令：

```
set path=%path%;C:\jdk1.3.0_02\bin  //%path%表示 path 原有内容
set classpath=.;C:\jdk1.3.0_02\lib
```

类似地，在 Windows NT/2000 中，对环境变量 path 添加“;C:\jdk1.3.0_02\bin”，新建环境变量 classpath 值为“.;C:\jdk1.3.0_02\lib”。

2）安装 UltraEdit 文本编辑器

UltraEdit 是一个功能强大的文本编辑器。

3．编辑 Java 源程序

由于 JDK 没有提供一个集成开发环境，可以使用 UltraEdit 编辑 Java 源程序。保存文件为 D:\myjava\Hello.java，如图 1.7 所示。

4．编译和运行 Java 程序

打开 MS-DOS 窗口，进入 Java 源程序所在的文件夹（设为 D:\myjava），然后输入命令：

```
D:\myjava>java2
```

执行 java2.bat，完成 path 和 classpath 路径的设置。如果已在 autoexec.bat 中添加路径，或设置过系统环境变量 path 和 classpath，则系统启动时已自动完成设置，不需要执行上述命令。

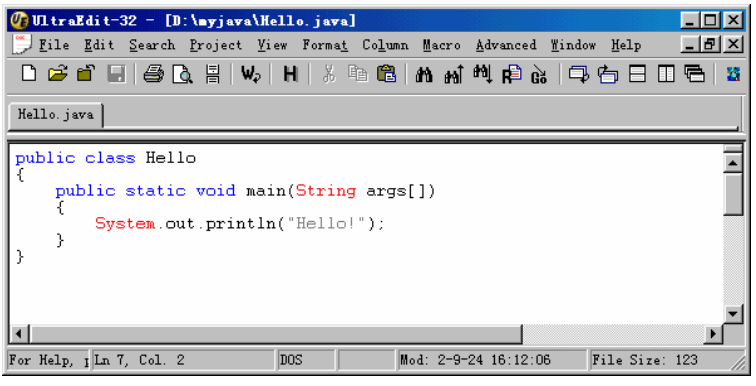


图 1.7　使用 UltraEdit 编辑 Java 源程序

注意：文件名必须与类名相同，通常首字母（H）大写。

然后输入如下编译命令：

```
D:\myjava>javac Hello.java
```

如果编译正确，将会产生 Hello.class 文件，否则产生异常，需要重新检查路径设置是否正确。再输入如下运行命令，运行 Hello.class 文件：

```
D:\myjava>java Hello
```

可以看到屏幕上显示运行结果，如图 1.8 所示。

图 1.8　Java 程序的编译和运行

5．查看 Java 的帮助文档

假设 Java 的帮助文档解压在 C:\jdk1.3.0_02\jdk1.3 文件夹中，打开其中的“ \docs\index.htm 文件，就可运行 Java 的帮助文档。

1.3.2　数据类型与流程控制

1．数据类型

在 Java 语言中，数据类型分为简单类型（primitive type）和引用类型（reference type）。Jav 共定义了 8 种简单类型，它们的分类及关键字如下：

- 整型——byte，short，int，long。
- 浮点型——float，double。
- 逻辑型——boolean。
- 字符型——char。

引用类型包括类（class）、数组（array）和接口（interface）。

2．操作符

Java 的操作符主要分为 4 类：算术操作符，位操作符，关系操作符和布尔操作符。表 1 按优先级从高到低的次序列出 Java 定义的所有操作符，分隔符的优先级最高，表中“ 左⇨右 表示从左向右的操作次序。

表 1-3　操作符的优先级

优先级	运　算　符	结合性
1	.　[]　()　;　,	
2	++　--　+=　~　!　+　－（一元）	右⇨左
3	*　/　%	左⇨右
4	+　－（二元）	左⇨右
5	<<　>>　>>>	左⇨右
6	<　>　<=　>=　instanceof	左⇨右
7	= =　!=	左⇨右
8	&	左⇨右

续表

优先级	运 算 符	结合性
9	∧	左⇨右
10	|	左⇨右
11	&&	左⇨右
12	||	左⇨右
13	？：	右⇨左
14	= *= /= %= += −= <<= >>= >>>= &= ∧= |=	右⇨左

注意：“+”、“-”和“+=”作为一元操作符时的优先级比作为二元操作符时高。

3．流程控制

按程序的执行流程，程序的控制结构可分为 3 种：顺序结构，分支结构和循环结构。这种结构的流程图如图 1.9 所示。

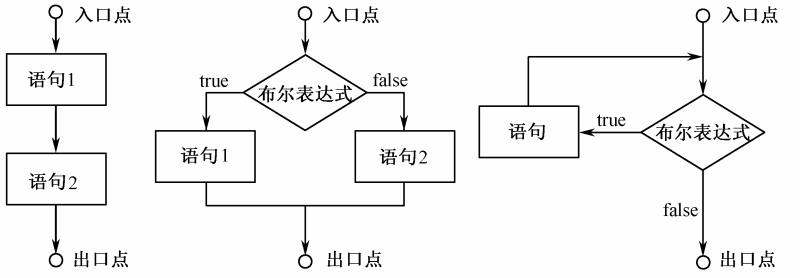


图 1.9　3 种控制结构

1）if 语句

Java 有 2 种分支语句实现分支结构，if 语句实现二路分支，switch 语句实现多路分支。

if 语句的定义格式如下：

```
if(<布尔表达式>)
  <语句 1>;
[else<语句 2>;]
```

2）switch 语句

switch 语句的特点是根据表达式的取值决定多分支选择。

switch 语句的定义格式如下：

```
 switch(<表达式>)
{
    case<常量 1>：<语句 1>；break;
    case<常量 2>：<语句 2>；break;
    ……
    [default: <语句>;]
 }
```

3）for 语句

Java 有 3 种循环语句 for、while、do-while，它们的共同点是根据循环条件来判断是否执行循环体。除此之外，每个语句都有自己的特点，应根据不同的问题选择合适的循环语句。而对于有些问题，用 3 种循环语句都可以实现。

for 语句的定义格式如下：

```
for(<表达式 1>;<表达式 2>;<表达式 3>)
    <语句>;
```

4）while 语句

while 语句的定义格式如下：

```
while(<布尔表达式>)
    <语句>;
```

5）do-while 语句

do-while 循环语句的定义格式如下：

```
 do
{
   <语句>;
  }while(<布尔表达式>);
```

6）转向语句

Java 语言提供了 3 种无条件转移语句：return，break 和 continue。

【例 1.8】 打印数字塔。

打印如下形式的数字塔：

```
                    1
                  1 2 1
                1 2 3 2 1
              1 2 3 4 3 2 1
            1 2 3 4 5 4 3 2 1
          1 2 3 4 5 6 5 4 3 2 1
        1 2 3 4 5 6 7 6 5 4 3 2 1
      1 2 3 4 5 6 7 8 7 6 5 4 3 2 1
    1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1
```

程序如下：

```
public class Dig9
{
   public static void main(String args[])
   {
      int i,j,n=9;
      for(i=1;i<=n;i++)                    //外层循环
      {
         for(j=0;j<n-i;j++)                //前导空格
```

```
                System.out.print("  ");
            for(j=1;j<i;j++)
                System.out.print(j+" ");
            for(j=i;j>=1;j--)
                System.out.print(j+" ");
            System.out.println();          //换行
        }
    }
}
```

4．数组

1）一维数组

声明一维数组变量的格式如下：

```
<类型> <数组名> []
```

只有用 new 操作符为数组分配空间后，数组才真正占用一片连续的存储单元。

使用 new 创建一维数组的格式如下：

```
<数组名> =new <类型>[<长度>]
```

可以在声明数组的同时，为数组赋初值。例如，

```
int a[]={1，2，3，4，5};
```

通过下标可以访问数组中的任何元素。数组元素的访问格式如下：

```
<数组名>[<下标表达式>]
```

Java 提供 length 方法返回数组的长度，即数组的元素个数，其格式如下：

```
<数组名>.length
```

2）二维数组

如果数组的元素类型也是数组，这种结构称为多维数组，最常用的是二维数组。通常以证明多个下标的形式来定义多维数组。例如，

```
int mat[][]=new int[10][10]；
```

声明了一个二维数组 mat，系统为它分配了 10×10 个存储单元。同样也可以初始化二维数组即将数组元素的值用多层花括号括起来。例如，

```
int mat[2][3]={{1，2，3},{4，5，6}};
```

【例 1.9】 输出循环移位方阵。

如果一维数组 table 中有如下值：

1	3	0	4	2

则输出如下形式的循环移位方阵：

1	3	0	4	2
3	0	4	2	1
0	4	2	1	3
4	2	1	3	0
2	1	3	0	4

程序如下：

```
public class Shift
{
   public static void main(String args[])
   {
      int i=0,j=0,n=5;
      int table[]=new int [n];
      for(i=0;i<table.length;i++)                    //产生 n 个随机数
         table[i]=(int)(Math.random()*10);
      for(i=0;i<table.length;i++)                    //循环 n 次
      {
         for(j=0;j<table.length;j++)                 //输出一维数组
            System.out.print(table[(i+j)%n]+"  ");
         System.out.println();
      }
   }
}
```

本例将产生的 *n* 个随机数存放在数组 table **中，每次产生不同的运行结果。**

1.3.3 类与对象

1．类的声明

Java **类的定义格式分为两部分：类声明和类主体。**

类声明中包括关键字 class**、类名及类的属性。类声明的格式如下：**

```
[<修饰符>] class <类名> [extends <超类名>] [implements<接口名>]
```

包含类主体的类结构如下：

```
<类声明>
    {
      <成员变量的声明>
      <成员方法的声明及实现>
}
```

声明成员变量必须给出变量名及其所属的类型，同时还可以指定其他特性。其格式如下

[<**修饰符**>][static][final]<**变量类型**> <**变量名**>

声明成员方法的格式如下：

```
[<修饰符>] <返回值类型> <方法名> ([<参数列表>]) [throws <异常类>]
{
   <方法体>
}
```

2．对象的创建和使用

对象是类的实例，属于某个已知的类。因此，定义对象之前，一定要先定义好该对象的类
对象声明的格式如下：

```
<类名> <对象名>
```

new 操作符用来创建新的对象，并为之分配内存。new 只需要一个参数，就是对一个构造
方法的调用。构造方法是用于初始化新对象的特别方法。

在声明对象的同时可以使用 new 操作符创建对象。其格式如下：

```
<类名> <对象名> = new <类名>([<参数列表>])
```

例如，

```
Date1 a=new Date1();
```

当用 new 创建了一个对象时，系统为对象中的变量进行了初始化。

通过对象引用类的成员变量的格式如下：

```
<对象名>.<变量名>
```

通过对象调用成员方法的格式如下：

```
<对象名>.<方法名>
```

在 Java 中，程序员只需创建所需的对象，而不需显式地销毁它们。Java 的垃圾回收机制
自动判断对象是否在使用，并能够自动销毁不再使用的对象，收回对象所占的资源。

Java 程序不直接释放动态分配的内存，而是由系统自动进行内存垃圾收集。

3．类的封装性

封装（encapsulation）的含义是：类的设计者把类设计成一个黑匣子，使用者只能看见类
中定义的公共方法，而看不见方法的实现细节，也不能直接对类中的数据进行操作。这样可以
防止外部的干扰和误用。即使改变了类中数据的定义，只要方法名不改变，就不会对使用该类
的程序产生任何影响。反过来说，封装减少了程序对类中数据表达的依赖性。这就是类的抽象
性、隐藏性和封装性。

1）访问权限

Java 定义了 4 种类的访问权限。表 1-4 说明了每一种权限修饰符允许的访问级别。

表 1-4　权限修饰符允许的访问级别

权限修饰符	同一类	同一包	不同包的子类	其他类
公有的（public）	✓	✓	✓	✓
保护的（protected）	✓	✓	✓	
默认的	✓	✓		
私有的（private）	✓			

2）设置类的访问权限

在声明一个类时，使用权限修饰符来设置类的访问权限，如：

```
public class Date1                    //公有的类
private class Date2                   //私有的类
```

3）设置类成员的访问权限

成员变量和成员方法都是 Java 类的成员。当声明一个类的成员时，可以使用权限修饰符

允许或不允许其他类的对象访问其成员，如：

```
private int year, month, day;      //私有的类成员
```

4）实例成员与类成员

Java 类包括两种不同类型的成员：实例成员和类成员。类成员也称为静态成员。

实例成员：当创建类的一个对象时，系统会为每一个对象创建一组类的实例变量。没用关键字 static 修饰的方法就是实例方法。实例方法只能通过对象来调用。实例方法体既可以访问类变量，也可以访问实例变量。

类成员：用关键字 static 修饰的变量称为类变量或静态变量。系统运行时，只为该类的第一个对象分配内存单元，其后所有新创建的对象都共享该变量。

用关键字 static 修饰的方法称为类方法或静态方法。类方法体只能访问类变量。类方法既可以通过对象来调用，也可以通过类名来调用。

1.3.4 类的继承性与多态性

继承（inheritance）是面向对象编程语言的基本要素之一。从现有类出发定义一个新类，称为新类继承了现有的类。其中，被继承（inherited）的现有类叫做超类（superclass），继承的新类叫做它的子类（subclass）。或者说，子类是由超类派生（也叫衍生）出的类。

Java 中的类都是 Object 的子类。Object 类定义了所有对象都必须具有的基本状态和行为，如等待条件变量、转化为字符串、通知其他对象条件变量已改变等。Java 中的每个类都从 Object 类继承了变量和方法。

1．创建子类

在 Java 中，除 Object 之外的每个类都有超类，如果没有显式地标明超类，则表示超类为 Object 类。

在类的声明时可说明类的超类，声明格式如下：

```
public class <子类名> extends <超类名>
```

子类继承超类中所有可被子类访问的成员变量，当子类成员变量与超类成员变量同名时，称子类成员变量将超类同名成员变量隐藏。子类继承超类中所有可被子类访问的成员方法，子类成员方法与超类成员方法同名时，称子类成员方法覆盖了超类的同名成员方法。

2．this、super 引用和 instanceof 对象操作符

Java 中，每个对象都具有对自身的访问权，称为 this 引用。每个对象对超类的访问，称为 super 引用。对象操作符 instanceof 用来测试一个指定对象是否为指定类（或它的子类）的实例，若是则返回 true，否则返回 false。

在类声明时除了可以说明类的超类，还可以声明最终类或抽象类等性质。

3．最终类与抽象类

1）最终类与最终方法

最终类是指不能被继承的类，即最终类不能有子类。用关键字 final 来说明最终类，final 通常放在 class 的前面。例如，

```
final class C1                    //合法，C1 为最终类
```

如果创建最终类似乎不必要，而又想保护类中的一些方法不被覆盖，可以在声明中用关键字 final 来指明哪些方法不能被子类覆盖，称为最终方法。例如，

```
final void m1()                    //合法，m1 为最终方法
```

2）抽象类与抽象方法

当声明一个方法为抽象方法时，意味着这个方法必须被子类的方法覆盖；或者为了确保一个方法能被子类的方法覆盖，则声明该方法为抽象方法。用关键字 abstract 来说明抽象方法，例如，

```
abstract void m3()                 //合法，m3 为抽象方法
```

构造方法是不能被声明为抽象的。方法声明中，static 和 abstract 也不能同时存在。

任何包含抽象方法的类必须被声明为抽象类。抽象类是不能直接被实例化的类。用关键字 abstract 来说明抽象类，例如，

```
abstract class D1                  //合法，D1 为抽象类
class D2 extends D1                //合法，D2 可以为抽象类的子类
```

抽象类的子类必须实现超类中的所有抽象方法，或者将自己也声明为抽象的。

注意：一个类不能既是最终类，又是抽象类，即关键字 abstract 和 final 不能合用。习惯上在类声明中，关键字 public 或 private 放在 final 或 abstract 前面。

当我们定义一个抽象概念时，可以声明一个抽象类只描述其中的结构，而不实现每个方法，这个抽象类可以作为一个超类被它的所有子类共享，而其中的方法由每个子类去实现。

4．类的多态性

所谓多态（polymorphism），意为一个名字可具有多种语义。在面向对象语言中，多态是指一个方法可能有多种版本，一次单独的方法调用（invoke）可能是这些版本中的任何一种，即实现“一个接口，多个方法”。

1）方法的重载

一个类中如果有许多同名的方法带有不同的参数，称为方法的重载（method overloading）。例如，在 Java 中，函数 abs()返回一个数的绝对值，参数类型有 4 种：

```
 static int abs(int a)
static long abs(long a)
static float abs(float a)
 static double abs(double a)
```

方法重载时：

- 参数必须不同，可以是参数个数不同，也可以是参数类型不同，甚至是参数顺序不同。
- 返回值可以相同，也可以不同。

重载的价值在于，它允许通过使用一个普通的方法来访问一系列相关的方法。当调用一个方法时，具体执行哪一个方法根据调用方法的参数决定，Java 运行系统仅执行与调用的参数相匹配的重载方法。

2）方法的覆盖

在前述继承规则中有：子类继承超类中所有可被子类访问的成员方法，如果子类方法与超类方法同名，则不能继承，此时子类的方法称为覆盖（override）了超类中的那个方法。

进行方法覆盖时，应注意以下 3 点：

- 子类不能覆盖超类中声明为 final 或 static 的方法。
- 子类必须覆盖超类中声明为 abstract 的方法，或者子类方法也声明为 abstract。
- 子类覆盖超类中同名方法时，子类方法声明必须与超类被覆盖方法的声明一样。

1.3.5 Java 的接口、内部类与包

1．接口

接口是没有实现的方法和变量的集合。接口定义格式如下：

```
[<修饰符>] interface <接口名>
{
    方法 1;
    方法 2;
}
```

一旦定义了一个接口，一个或更多的类就能实现这个接口。用关键字 implements 声明一个类将实现一个接口，并且一个类可以实现多个接口，多个接口用逗号隔开。类声明的格式如下：

```
[<修饰符>]class<类名>[extends<超类名>] [implements <接口名 1>，<接口名 2>…]
```

Java 只支持单重继承机制，使用继承与接口可以实现多重继承功能。

2．内部类

一个类被嵌套定义于另一个类中，称为内部类（inner class），也称为嵌套类。包含内部类的类称为外部类。与一般的类相同，内部类可以具有成员变量和成员方法。通过建立内部类的对象，可以存取其成员变量和调用其成员方法。

3．引用 Java 定义的包

包（package）是 Java 提供的一种区别类名字空间的机制。Java 提供了丰富的类库，分别放在不同的包中。运行 C:\jdk1.3.0_02\jdk1.3\docs\api\index.html，打开 Java 的帮助文档，可以看到所有的包及其中的类和方法。

图 1.10 所示的是 java.lang 包中的 Math 类，其中定义了许多方法实现数学函数的功能。语言包 java.lang 是自动导入的，程序中可以使用其中的类。如果要使用其他包中的类，必须用 import 语句导入。import 语句的格式如下：

```
import <包名 1> [.<包名 1> [.<包名 1>…]] .<类名 1>｜*;
```

例如，

```
import java.applet.Applet;          //导入 java.applet 包中的 Applet 类
import java.awt.*;                  //导入 java.awt 包中的所有类
```

4．Java 的常用包

Java 的常用包有：

- java.lang——语言包。

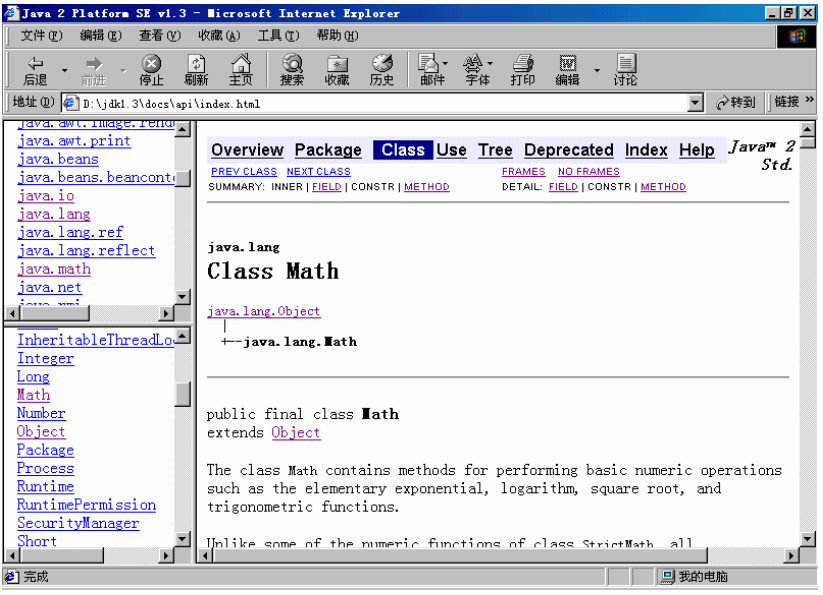


图 1.10　Java **的包**

- java.awt——**抽象窗口工具包。**
- java.applet——**实现** Applet **应用程序。**
- java.util——**实用包。**
- java.text——**文本包。**
- java.io——**输入输出流的文件包。**
- java.net——**网络功能。**

1）**语言包**

Java **语言包** java.lang **所提供的类构成了** Java **语言的核心。语言包中的类是** Java **类库中最基本的类，**Java **系统自动隐含地将这个包引入用户程序，用户无需用** import **语句导入它，就可使用其中的类。语言包中最主要的类有：**

- Object **类——**Java **类层次的根，所有其他的类都是由** Object **类派生出来的。**
- **数据类型包装类——简单数据类型的类包装，如** Integer，Float，Boolean **等。**
- Math **类——数学类，提供一组数学函数和常数。**
- String **类——字符串类。**
- System **和** Runtime **类——系统和运行时类，提供访问系统和运行时环境资源。**
- Thread **类——线程类，提供多线程环境的线程管理和操作。**
- Class **类——类操作类，为类提供运行时信息。**
- Throwable **类，**Exception **类和** Error **类——错误和异常处理类。**
- Process **类——过程类。**

2）**实用包**

Java **实用包** java.util **提供了实现各种不同实用功能的类。实用包包括：**

- **日期类——包括** Data **类，**Calendar **类和** GregorianCalendar **类。**
- **数据结构类——包括** LinkedList **类，**Vector **类，**Stack **类和** Hashtable **类等。**
- Random **类——随机数类。**

5．自定义包

包的定义格式如下：

```
package <包名>
```

用圆点“.”将每个包名分隔就能形成包等级，其格式如下：

```
package <包名 1>[.<包名 2>[.<包名 3>]];
```

本书将声明的一些公用的类放在包 ds_java 中。操作步骤如下：

- 创建文件夹 D:\ds_java。
- 环境变量 classpath 设置为：

```
set  classspath=.;C:\jdk1.3.0_02\lib;D:\
```

- 声明包 ds_java：

```
package ds_java;                //声明包

public class OnelinkNode;       //声明类
```

声明包的语句放在类的声明之前。

- 编译类。设源程序为 LinearList1.java，将编译后生成的 LinearList1.class 文件复制到 D:\ds_java 文件夹中。
- 引用包中的类，之后在其他类中可用 import 语句引用 ds_java 包中声明的类：

```
import ds_java.LinearList1;
```

引用包的语句放在类的声明之前。

1.3.6 异常处理

根据程序运行时出现的错误性质将运行错误分为两类：错误和异常。

1）致命性的错误

如果程序进入了死循环，或递归无法结束，或内存溢出，这些现象称为错误。错误只能在编程阶段解决，运行时程序本身无法解决，只能依靠其他程序干预，否则会一直处于非正常状态。

2）非致命性的异常

如果操作时除数为 0，或操作数超出数据范围，或打开一个文件时发现文件不存在，或待装入的类文件丢失，或网络连接中断等，这些现象称为异常。在源程序中加入异常处理代码，当程序运行中出现异常时，这些代码修正错误，解决问题，使程序仍可以继续运行直至正常结束。

由于异常是可以检测和处理的，所以产生了相应的异常处理机制。目前，大多数面向对象的语言都提供了异常处理机制，而错误处理一般由系统承担，语言本身不提供错误处理机制。

1．异常处理机制

Java 提供了异常处理机制，它是通过面向对象的方法来处理异常的。

当程序发生异常时，发现异常的代码可以抛出（throw）一个异常，生成一个异常对象，并把它提交给运行系统，运行系统捕获（catch）该异常，交由程序员编写的相应代码进行异常处理。

Java 通过错误类（Error）和异常类（Exception）来处理错误和异常。

Error 类对象是由 Java 虚拟机生成并抛出给系统，如内存溢出错、栈溢出错、动态链接等。通常，Java 程序不对错误进行处理。

Exception 类对象是 Java 程序抛出和处理的对象。它有各种不同的子类分别对应于不同类型的异常，如除数为 0 的算术异常、数组下标越界异常、空指针异常等。

2．异常的产生、捕获与处理

1）异常的产生

例如，下列语句因除数为 0 将会产生一个算术异常（ArithmeticException）：

```
System.out.print("3/0="+(3/0));
```

而下列语句将产生一个数组越界异常（ArrayIndexOutOfBoundsException）：

```
int a[]={5,6,7,8};
System.out.println(a[5]) ;
```

2）使用 try-catch-finally 语句捕获和处理异常

Java 的异常处理机制提供了 try-catch-finally 语句来捕获和处理一个或多个异常，其语法格式如下：

```
try
{
    <语句 1>
}
catch(ExceptionType1 e)
{
    <语句 2>
}
finally
{
    <语句 3>
 }
```

其中，<语句 1>是可能产生异常的代码；<语句 2>是捕获某种异常对象时进行处理的代码，ExceptionType1 代表某种异常类，e 为相应的对象；<语句 3>是其后必须执行的代码，无论是否捕获到异常。

catch 语句可以有一个或多个，但至少要有一个 catch 语句，finally 语句可以省略。

try-catch-finally 语句的作用是：当 try 语句中的代码产生异常时，根据异常的不同，由不同 catch 语句中的代码对异常进行捕获并处理；如果没有异常，则 catch 语句不执行；而无论是否捕获到异常都执行 finally 中的代码。

3．抛出异常

在捕获一个异常前，必须有一段代码生成一个异常对象并把它抛出。抛出异常的代码既可以是 Java 运行时系统，也可以是程序员自己编写的代码，即 try 语句中的代码本身不会由系统产生异常，而是由程序员故意抛出异常。

使用 throw 语句抛出异常的格式如下：

```
throw <异常对象>
```

其中，throw 是关键字，<异常对象>是创建的异常类对象。

4．抛出异常的方法与调用方法处理异常

在实际编程中,有时并不需要由产生异常的方法自己处理,而需要在该方法之外处理异常这时与异常有关的方法就有两个：抛出异常的方法和处理异常的方法。

1）抛出异常的方法

在方法声明中，添加 throws 子句表示该方法将抛出异常。带有 throws 子句的方法的声明格式如下：

```
[<修饰符>]<返回值类型><方法名>([<参数列表>])[throws<异常类>]
```

其中，throws 是关键字，<异常类>是方法要抛出的异常类，可以声明多个异常类，用逗号隔开。

2）由调用方法处理异常

由一个方法抛出异常后，系统将异常向上传递，由调用方法来处理这些异常。

1.3.7 Java 的标准数据流

标准输入输出是指在字符方式下（如 DOS），程序与系统进行交互的方式，分为 3 种：

- stdin（标准输入）——对象是键盘。
- stdout（标准输出）——对象是屏幕。
- stderr（标准错误输出）——对象也是屏幕。

Java 通过系统类 System 实现标准输入输出的功能。System 类在 java.lang 包中，声明为一个 final 类：

```
public final class System extends Object
```

System 类不能创建对象，直接使用，其中有 3 个成员：in，out 和 err。

1．标准输入 System.in

System.in 作为字节输入流类 InputStream 的对象 in 实现标准输入，其中有 read()方法从键盘接收数据。

```
public int read() throws IOException            //返回读入的一个字节
public int read(byte[] b) throws IOException    //读入的多个字节返回缓冲区
```

如果输入流结束，返回-1；发生 I/O 错时，抛出 IOException 异常。

2．标准输出 System.out

System.out 作为 PrintStream 类的对象 out 实现标准输出。其中有 print()和 println()两个方法，这两个方法支持 Java 的任意基本类型作为参数。

```
public void print(long l)
public void println()
```

两者的区别在于，println()在输出时加一个回车符，而 print()则不输出回车符。其实我

在前面的例子中已经多次使用该方法，如

```
System.out.println("Hello!");
```

3．标准错误输出 System.err

System.err 与 System.out 相同，以 PrintStream 类的对象 err 实现标准错误输出。

习　题　1

1.1　简述下列术语：数据、数据元素、数据项、数据类型、抽象数据类型、数据结构、逻辑结构、存储结构、结点、算法。

1.2　请选择下列各式的时间复杂度。

（1）$100n^3$

（2）$6n^2-12n+1$

（3）1024

（4）$n+2\log_2 n$

（5）$n(n+1)(n+2)/6$

（6）$2n+1+100n$

（A）$O(1)$；（B）$O(2^n)$；（C）$O(n)$；（D）$O(n^2)$；（E）$O(\log_2 n)$；（F）$O(n^3)$。

实　习　1

1．实验目的：掌握 Java 程序的设计、编辑、编译与运行。

正确安装并熟练使用 UltraEdit 环境编辑程序，正确安装并熟练使用 Java 的 JDK 环境编译、调试、运行程序。

理解 Java 程序的类的格式，掌握基本数据类型，熟练运用分支、循环等语句控制程序流程。

2．题意

输出下列方阵：

n=4

1	2	6	7
3	5	8	13
4	9	12	14
10	11	15	16

3．实验要求

（1）改变 n 的值，观察运行结果。

（2）改变方向得到如下不同的方阵。

1	3	4	10
2	5	9	11
6	8	12	15
7	13	14	16

（3）采用多种方式数据输入。

分别从命令行、键盘输入数据（标准输入流 System.in）。

第2章　线　性　表

线性表是最简单、最基本的一种数据结构。线性表可以用顺序存储结构和链式存储结构存储。可以在线性表的任意位置，对线性表进行插入和删除数据元素的操作。

本章讨论线性表的逻辑结构，以顺序存储结构实现的线性表（顺序表）和以链式存储结构实现的线性表（链表）的结点结构和操作实现，分析、比较顺序表和链表的优缺点，并介绍以数据元素为字符的线性表——串。

建议本章授课 10 学时，实验 4 学时。

2.1　线性表的概念

线性表（linear list）是相同类型的数据元素的有限序列，数据元素之间具有顺序关系。除第一个和最后一个数据元素外，每个数据元素只有一个前驱数据元素和一个后继数据元素，第一个数据元素没有前驱数据元素，最后一个数据元素没有后继数据元素。数据元素可以是一个数、符号或字符串，也可以是其他更复杂的数据形式。

例如，英文字母表{A，B，C，…，Z}可以看成是一个线性表，数据元素是单个字母，数据元素间是按顺序排列的。又如，将一个班级学生的成绩按学号排列构成的学生成绩表是一个线性表，数据元素是每个学生的情况，包括学号、姓名、成绩等。

2.1.1　线性表的抽象数据类型

1．线性表的数据元素

线性表是由 n（$n\geq 0$）个相同类型的数据元素 a_1，a_2，…，a_n 组成的有限序列，记为：

$$\text{LinearList}=\{a_1，a_2，\cdots，a_n\}$$

其中，n 表示线性表的元素个数，称为线性表的长度。若 $n=0$，则称为空表。若 $n>0$，对于线性表中第 i 个数据元素 a_i，有且仅有一个直接前驱数据元素 a_{i-1} 和一个直接后继数据元素 a_{i+1}，而 a_1 没有前驱数据元素，a_n 没有后继数据元素。

线性表中的数据元素具有相同的属性，属于同一种数据类型。当我们讨论线性表的抽象数据类型时，我们使用抽象数据元素 a_i 表示线性表的数据元素。具体设计应用程序时，抽象的数据类型将被具体的数据类型所取代。例如，在 Java 中，使用 Object 类描述不确定类型的数据元素，因为 Object 类是 Java 类层次的根，所有其他的类都是由 Object 类派生出来的。使用整数类型 int、字符类型 char 或字符串类型 String 等基本或引用类型描述确定类型的数据元素。

2．线性表的基本操作

线性表的特点是可以在任意位置进行插入和删除数据元素的操作。对线性表进行的操作主要有：

- 求长度——求线性表的数据元素个数。
- 访问——对线性表中指定位置的数据元素进行存取、替换等操作。
- 插入——在线性表指定位置上，插入一个新的数据元素，插入后仍为一个线性表。
- 删除——删除线性表指定位置的数据元素，同时保证更改后的线性表仍然具有线性表的连续性。
- 复制——重新复制一个线性表。
- 合并——将两个或两个以上的线性表合并起来，形成一个新的线性表。
- 查找——在线性表中查找满足某种条件的数据元素。
- 排序——对线性表中的数据元素按关键字的值，以递增或递减的次序进行排列。
- 遍历——按次序访问线性表中的所有数据元素，并且每个数据元素恰好访问一次。

在 Java 中，将线性表的抽象数据类型声明为一个抽象类 LinearList，其中，数据元素的类型是 Object 类，用一些抽象方法描述对线性表的多个基本操作。声明如下：

```
abstract public class LinearList                    //线性表的抽象类
   {
      abstract public int length();                     //返回线性表的长度
      abstract public boolean isEmpty();                //判断线性表是否为空
      abstract public boolean isFull();                 //判断线性表是否已满
      abstract public Object get(int i);                //返回指定位置数据元素
      abstract public Object set(int i,Object obj);     //设置指定位置数据元素
      abstract public void insert(int i,Object obj);    //将数据元素插入指定位
      abstract public int indexof(Object obj);    //返回给定数据元素首次出现位
      abstract public boolean contains(Object obj);//查找是否包含给定数据元素
      abstract public void remove(int index);       //删除指定位置的数据元素
      abstract public void remove(Object obj);      //删除首次出现的给定数据元素
      abstract public void output();                //输出线性表全部数据元素
}
```

2.1.2 线性表的顺序存储结构

线性表有两种存储结构：顺序存储结构和链式存储结构。用顺序存储结构实现的线性表称为顺序表，用链式存储结构实现的线性表称为链表。

顺序表用一组连续的存储单元顺序存放线性表的数据元素，数据元素在内存的物理存储次序与它们在线性表中的逻辑次序是一致的，即数据元素 a_i 与其前驱数据元素 a_{i-1} 及后继数据元素 a_{i+1} 的位置相邻。

线性表中的数据元素属于同一种类型，因此每个数据元素在内存占用的空间大小相同。假设每个数据元素占据 c 个存储单元，第一个数据元素的地址为 $\mathrm{Loc}(a_1)$，则第 i 个数据元素的地址为：

$$\mathrm{Loc}(a_i)=\mathrm{Loc}(a_1)+(i-1)\times c$$

由此可知，每个数据元素的地址是它在线性表中位置的线性函数，是可以直接计算出来的，而每次计算所花费的时间都是相同的。因此，在顺序存储结构的线性表中，每个数据元素是可以随机访问的。顺序表的存储结构如图 2.1 所示。

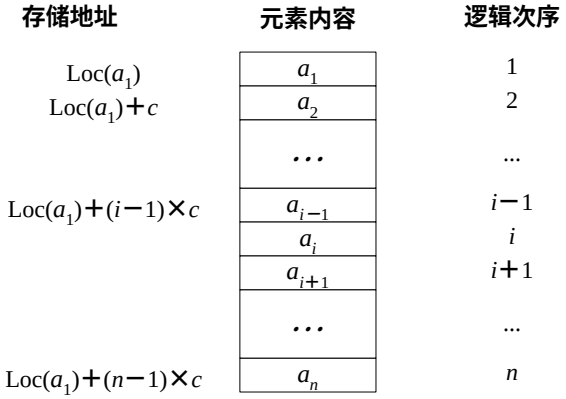


图 2.1　顺序表的存储结构

1．顺序表的类定义

用高级程序设计语言中的数组可以实现顺序表。下面声明的 LinearList1 类表示顺序表，其中一维数组 table 是数据元素所占用的存储空间，元素为整型，n 表示顺序表的长度。声明如下：

```
public class LinearList1                    //线性表的顺序存储结构
{
   private int table[];                     //私有成员
   private int n;                           //记载顺序表的长度
}
```

LinearList1 类的一个对象表示一个线性表。当我们需要使用一个线性表时，创建 LinearList1 类的一个对象，通过这个对象调用公有的（public）方法进行相应的操作或访问类中成员。

从程序员角度看，能够使用类的功能即可，不必关心类中的具体设计；另一方面，从类设计的角度看，不需要也不能够向类之外的对象提供直接访问成员的功能，所以类中成员都设置为私有的（private），对外是不可见的。这就是类的封装。

为使算法简单起见，LinearList1 类中 table 的元素类型声明为基本数据类型 int。

2．顺序表的操作

1）顺序表的初始化

使用构造方法创建顺序表对象，为顺序表分配存储空间，设置顺序表为空状态。算法如下：

```
public LinearList1(int n)                   //为顺序表分配 n 个存储单元
{                                           //此时顺序表长度 this.n 为 0
   table=new int [n];          //所占用的存储单元个数 this.table.length 等于 n
   this.n=0;
}
```

2）判断顺序表的空与满状态

当 n=0 时，顺序表为空状态；当 $n \geq$table.length 时，顺序表为满状态。算法如下：

```
public boolean isEmpty()              //判断顺序表是否为空
{
   return n==0;
}
public boolean isFull()               //判断顺序表是否已满
{
   return n>=table.length;            //table.length 返回数组长度
}
```

3）返回顺序表长度

```
public int length()                   //返回顺序表的长度
{
   return n;
}
```

注意：LinearList1 类中使用 table.length 返回数组 table 的存储单元个数，此时 length 是系统定义的数组对象的长度，没有括号；而 LinearList1 类中又将 length()定义为返回线性表长度的方法，调用时必须加上括号，如 list1.length()。

4）获得顺序表的第 i 个数据元素值

下述算法中的参数 i 指顺序表的第 i 个元素。由于 Java 的数组下标从 0 开始，所以各算法中必须将 i 转换成相应元素位置的下标。算法如下：

```
public int get(int i)                 //返回第 i 个元素值
{
   if(i>0 && i<=n)
      return table[i-1];
   else
      return -1;
}
```

5）设置顺序表的第 i 个数据元素值

```
public void set(int i,int k)          //设置第 i 个元素值为 k
{
   if(i>0 && i<=n+1)
   {
      table[i-1]=k;
      if(i==n+1)
         n++;
   }
}
```

6）查找

在线性表中查找给定值 k 的过程为：从线性表的第一个数据元素开始，依次比较线性表中的数据元素是否为 k，若当前数据元素与 k 相等，则查找成功；否则继续与下一个数据元素进

行比较，当比较完线性表全部数据元素后仍未找到，则返回查找不成功信息。算法如下：

```
public boolean contains(int k)        //查找线性表是否包含 k 值
{                                     //查找成功时返回 true，否则返回 false
   int j=indexof(k);
   if(j!=-1)
      return true;
   else
     return false;
}
public int indexof(int k)             //查找 k 值
{                                     //查找成功时返回 k 值首次出现位置，否则返回-1
   int j=0;
   while(j<n && table[j]!=k)
      j++;
   if(j>=0 && j<n)
      return j;
   else return -1;
}
```

线性表的其他查找算法将在第 7 章中详细讨论。

7）在顺序表的第 i 个位置上插入数据元素

要想在顺序表的第 i 个位置上插入给定值 k，使得插入后线性表仍然保持连续性，首先必须将第 i 到第 n 个位置上的数据元素依次向后移动一个位置，即依次向后移动从 a_n 到 a_i 的数据元素，空出第 i 个内存单元位置，然后在第 i 个位置上放入给定值 k，其过程如图 2.2 所示。图中 size 表示数组的存储单元个数。

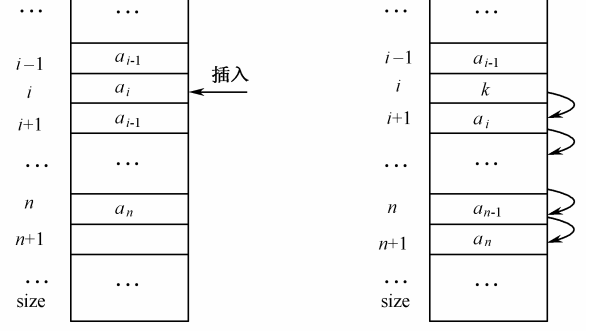


图 2.2　顺序表中插入数据元素

在顺序表的第 i 个位置插入数据元素，算法如下：

```
public void insert(int i,int k)          //插入 k 值作为第 i 个值。
{
   int j;
```

```
        if(!isFull())
        {
            if(i<=0) i=1;
            if(i>n)
            i=n+1;
            for(j=n-1;j>=i-1;j--)
                table[j+1]=table[j];
            table[i-1]=k;
            n++;
        }
        else
            System.out.println("数组已满，无法插入"+k+"值！");
    }
    public void insert(int k)            //添加 k 值到顺序表最后，重载
    {
        insert(n+1,k);
    }
```

8）**删除顺序表的第 i 个数据元素**

若想删除顺序表的第 i 个数据元素，使得删除后线性表仍然保持连续性，则必须将顺序表中原来的第 i+1 到第 n 位置上的数据元素依次向前移动。数据元素移动次序是从 a_{i+1} 到 a_n，将 a_{i+1} 移动到第 i 位置上，实际上就是删除了 a_i，如图 2.3 所示，图中 size 表示数组的存储单元个数。

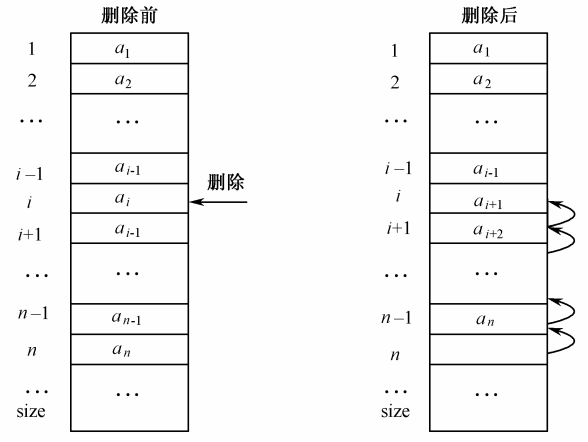


图 2.3　删除顺序表中的数据元素

删除顺序表的第 i 个数据元素的算法如下：

```
    public void remove(int k)            //删除 k 值首次出现的数据元素
    {
        int i=indexof(k);                //查找 k 值的位置
        if(i!=-1)
        {
```

```
            for(int j=i;j<n-1;j++)        //删除第 i 个值
                table[j]=table[j+1];
            table[n-1]=0;
            n--;
        }
        else
            System.out.println(k+"值未找到，无法删除!");
    }
```

3．顺序表操作的算法分析

从上述顺序表操作的算法可以看出，判断顺序表的空与满状态，返回顺序表长度，获得
设置数据元素值等操作都与数据元素的个数 n 无关，所以时间复杂度为 $O(1)$。而在线性表中
查找给定值，插入和删除数据元素等操作都与数据元素的个数 n 有关，时间复杂度为 $O(n)$。

顺序表查找算法所花费的时间主要用于数据元素进行比较，时间复杂度将在第 7 章中详细
讨论。

在顺序表中进行插入或删除数据元素操作时，算法所花费的时间主要用于移动数据元素
显然，若在顺序表的第一个位置插入数据元素，则移动次数为 n；若在顺序表的最后一个位置
插入数据元素，则移动次数为 0。设在第 i 个位置插入数据元素的概率为 p_i，则在顺序表中插
入一个数据元素所做的平均移动次数为

$$\sum_{i=1}^{n+1}(n+1-i)\times p_i$$

如果在各位置插入数据元素的概率相同，即

$$p_1 = p_2 = \Lambda = p_n = \frac{1}{n+1}$$

则有

$$\sum_{i=0}^{n}(n-i)\times p_i = \frac{1}{n+1}\sum_{i=0}^{n}(n-i) = \frac{1}{n+1}\times\frac{n(n+1)}{2} = \frac{n}{2}$$

也就是说，在等概率情况下，插入一个数据元素平均需要移动线性表全部数据元素的一半
时间复杂度为 $O(n)$。

与插入操作同理，在等概率情况下，删除一个数据元素平均需要移动顺序表全部数据元素
的一半，时间复杂度为 $O(n)$。

综上所述，线性表的顺序存储结构的特点是，其存储次序直接反映了其逻辑次序，所有的
存储空间都可以用来存放数据元素，因此存储密度较高，具有可以直接访问第 i 个数据元素的
优点；但插入和删除操作很不方便，每插入或删除一个数据元素，都需要移动大量的数据元素，
其平均移动次数是线性表长度的一半；另外，申请数组空间时，需要给出数组存储单元的个数
（即数组的最大长度）这个数值不易精确计算出，只能估算。这样，可能出现因空间估算过小
而造成软件系统溢出，因空间估算过大而造成软件系统内存资源浪费的问题。

【例 2.1】 以顺序表求解约瑟夫环问题。

约瑟夫环（Josephus）问题：古代某法官要判决 n 个犯人的死刑，他有一条荒唐的法律，
将犯人站成一个圆圈，从第 s 个人开始数起，每数到第 d 个犯人，就拉出来处决，然后再数
个，数到的人再处决……直到剩下的最后一个可赦免。

当 n=5，s=1，d=2 时，约瑟夫环问题执行过程如图 2.4 所示。

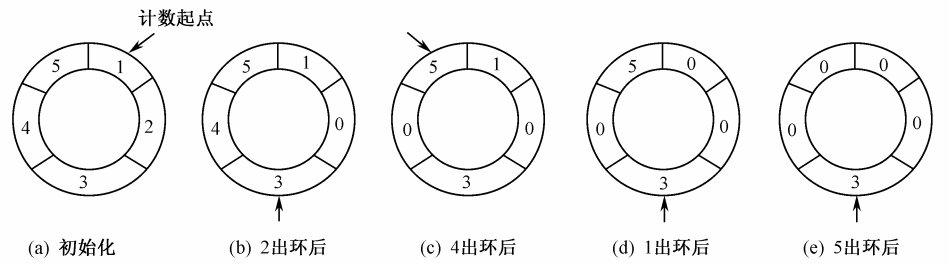


图 2.4　约瑟夫环

算法描述如下：

- 建立一个具有 n 个数据元素的线性表对象 ring1。
- 从第 s 个数据元素开始，依次计数，每数到 d，就将该数据元素设置为空。
- 重复上一步，依次选中 n-1 个数据元素出环。
- 输出环中所剩的最后一个数据元素。

从逻辑结构上看，该算法应该多次使用删除操作，即当每次一个人出环时，删除线性表相应位置的数据元素，这时必须移动其他元素。下面程序中，为减少数据移动次数，没有使用删除操作，采取的是一种变通办法，将相应位置的数据元素值设为空，这就使得计数时必须跳过值为空的数据元素。程序如下：

```
import ds_java.LinearList1;          //声明引用 ds_java 包中的 LinearList1 类
   public class Josephus1
   {
      public static void main(String args[])
      {
         (new Josephus1()).display(5,1,2);
      }
      public void display(int N,int S,int D)
      {
         final int NULL=0;
         LinearList1 ring1=new LinearList1(N);
         int i,j,k,m;
         for(i=1;i<=N;i++)                    //n 个人依次插入线性表
            ring1.insert(i);
         ring1.output();
         i=S-1;                               //从第 s 个人开始计数
         k=N;
         while(k>1)                           //n-1 个人依次出环
         {
            j=0;
            while(j<D)
            {
```

```
                i=i%N+1;                    //将线性表看成环形
                if(ring1.get(i)!=NULL)
                   j++;                     //计数
             }
             System.out.println("out :  "+ring1.get(i));
             ring1.set(i,NULL);             //第 i 个人出环,设置第 i 个位置为空
             k--;
             ring1.output();
          }
          i=1;
          while(i<=N && ring1.get(i)==NULL)//寻找最后一个人
               i++;
          System.out.println("The final person is "+ring1.get(i));
       }
   }
```

顺序表类 LinearList1 **的成员变量和成员方法声明在源程序** LinearList1.java **中，其第一条语句为**

```
    package ds_java;                        //声明包
```

将源程序 LinearList1.java **和** Josephus1.java **保存在** D:\myjava **文件夹中。在** DOS **窗口中用** JDK 1.3 **编译源程序，命令如下：**

```
D:\myjava>javac LinearList1.java
```

将编译后生成的 LinearList1.class **文件复制到** D:\ds_java **文件夹（包）中，再编译：**

```
D:\myjava>javac Josephus1.java
```

运行 Josephus1.class **文件：**

```
D:\myjava>java Josephus1
```

程序运行结果如下：

```
table:  1 2 3 4 5
out :  2
table:  1 0 3 4 5
out :  4
table:  1 0 3 0 5
out :  1
table:  0 0 3 0 5
out :  5
table:  0 0 3 0 0
The final person is 3
```

为突出重点问题，降低算法复杂程度，本例淡化了输入方式，以常数为例运行程序，本章及以后各章仍多采用常数。读者可根据需要使用命令行参数 args[]**或使用标准输入流** System.in **从键盘输入数据。**

思考题：调用在顺序表中删除数据元素的算法，改写例 2.1**。**

2.1.3 线性表的链式存储结构

线性表的链式存储结构是把线性表的数据元素存放在结点中。结点（node）由数据元素域和一个或若干个指针域组成，指针是用来指向其他结点地址的。这样，线性表数据元素之间的逻辑次序就由结点间的指针来实现。

由于 Java 没有类似 C 中的指针类型，所以，不能用指针方式实现链表，可以使用 Java“引用的类”表示链式结构。

1）声明自引用的类

自引用的类（self-referential class）包含一个指向同一类的引用成员。例如，

```
public class Node
{
   int data;
   Node next;
}
```

Node 类声明了两个成员变量：data 和 next。data 用于记载数据，next 引用 Node 类的对象。Node 构成自引用的类。next 称为链（link），即 next 可以替代指针将一个 Node 类的对象与另一个同类型的对象“连接”起来，实现结点间的链接。

在 Java 中，将链表中的结点设计成独立的 Node 类，一个结点就是 Node 类的一个对象。Java 的类是引用类型，通过将自引用对象链接起来，就可以实现多种动态的数据结构，如链表和二叉树等。

2）创建并使用对象

创建和维护动态数据结构需要动态内存分配（dynamic memory allocation），即一个程序在运行时申请所需的内存空间，系统分配内存后程序可以使用，使用完后释放不需要的空间。

Java 使用操作符 new 创建对象并为之分配内存。例如，

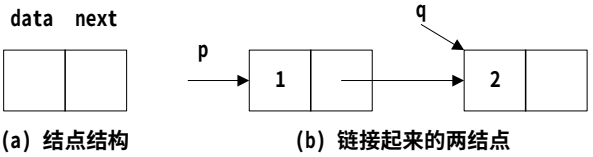
```
Node p,q;                    //声明 p 和 q 是 Node 类的对象
p=new Node();                //创建 Node 类的一个对象，由 p 引用
q=new Node();                //创建 Node 类的一个对象，由 q 引用
```

如果没有可用内存，new 产生一个异常 OutOfMemoryException。

由 p 引用对象中两个成员变量的格式为 p.data 和 p.next。通过下述语句可将 p、q 两个对象链接起来：

```
p.next=q;
```

这时，称对象 p 的 next 成员变量指向对象 q。链表的结点结构和链接起来的两结点如图 2.5 所示。



(a) 结点结构　　(b) 链接起来的两结点

图 2.5　链接起来的两结点

2.2 线性链表

用链式存储结构实现的线性表称为线性链表（linear linked list）或链表。指向线性链表第一个结点的指针称为线性链表的头指针，一个线性链表由头指针指向第一个结点，每个结点由链指向其后继结点，最后一个结点的链为空（null）。链表的结点个数称为链表的长度，长度为 0 时称为空表。

线性链表根据链的个数分为单向链表和双向链表两种。

2.2.1 单向链表

在线性链表中，如果每个结点只有一个链，则称为单向链表（singly linked list）。单向链表各结点的链通常指向其后继结点。

图 2.6 是一个单向链表的结构示意图，图中 head 是指向链表第一个结点的头指针，“∧”表示空指针值。

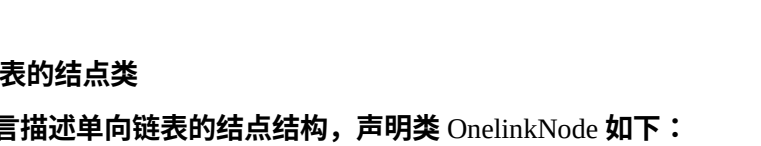


图 2.6 单向链表结构示意图

在单向链表中，从头指针 head 开始，沿链表的方向前进，就可以顺序访问链表中的每个结点。

1．单向链表的结点类

用 Java 语言描述单向链表的结点结构，声明类 OnelinkNode 如下：

```
package  ds_java;                  //声明包
public class OnelinkNode           //单向链表的结点类
{
   public int data;                //存放结点值
   public OnelinkNode next;        //后继结点的引用
   public OnelinkNode(int k)       //构造值为 k 的结点
   {
      data=k;
      next=null;
   }
   public OnelinkNode()            //无参数时构造值为 0 的结点
   {
      this(0);
   }
}
```

OnelinkNode 类声明了单向链表的结点结构，一个 OnelinkNode 类的对象只表示链表中的一个结点，通过成员变量 next 的自引用方式实现线性表中各数据元素的逻辑关系。OnelinkNode 类中成员变量设计为公有的（public），允许包之外的其他类直接访问。将 OnelinkNode 类编译后

放在 ds_java 包中，可供其他类引用。

2．单向链表类

用 Java 语言描述单向链表，声明类 Onelink1 如下：

```
package  ds_java;
import ds_java.OnelinkNode;                    //引用单向链表的结点类
public class Onelink1
{
    protected OnelinkNode head;                //指向链表的第一个结点
}
```

Onelink1 类的一个对象表示一条单向链表，成员变量 head 作为链表的头指针，指向链表的第一个结点。head 为被保护的（protected），可被其子类继承。当 head 为 null 时，表示链表为空，元素个数为 0。

3．单向链表的操作

将下述对单向链表的操作作为 Onelink1 类中的成员方法，并将 Onelink1 类编译后放在 ds_java 包中供其他类引用。

1）单向链表的初始化

用 Onelink1 类的构造方法建立一条链表，算法如下：

```
public Onelink1()                              //构造空的单向链表
{
   head=null;
}
public Onelink1(OnelinkNode h1)                //构造由 h1 指向的单向链表
{
   head=h1;
}
```

2）判断链表是否为空状态

```
public boolean isEmpty()                       //判断单向链表是否为空
{
   return head==null;
}
```

3）建立单向链表

使用 new 创建 OnelinkNode 类的对象，并依次链入链表的末尾，如图 2.7 所示。

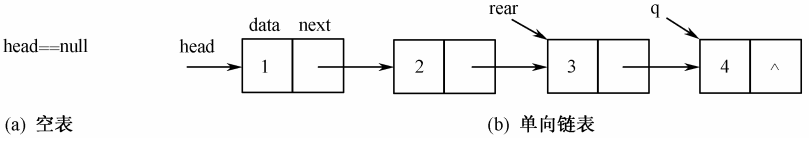


图 2.7　建立单向链表

设 rear 指向原链表的最后一个结点，q 指向新创建的结点，则将结点 q 链在 rear 结点之后的语句是：

```
rear.next=q;
rear=q;
```

这样就将结点 q 作为最后一个结点链入，重复上述操作可以建立一条单向链表，算法如下：

```
public Onelink1(int n)                    //以 n 个随机值建立单向链表
{
   OnelinkNode rear,q;
   if(n>0)
   {
      int k=(int)(Math.random()*100);     //产生随机数
      head=new OnelinkNode(k);
      rear=head;                          //rear 指向链表尾结点
      for(int i=1;i<n;i++)
      {
         k=(int)(Math.random()*100);
         q=new OnelinkNode(k);            //建立值为 k 的结点 q
         rear.next=q;                     //结点 q 链入原链表尾
         rear=q;                          //rear 指向新链尾结点
      }
   }
}
```

4）返回链表长度

```
public int length()                       //返回链表的长度
{
   int n=0;
   OnelinkNode p=head;
   while(p!=null)
   {
      n++;
      p=p.next;
   }
   return n;
}
```

5）输出单向链表

将已建立的单向链表按顺序输出各结点值。从 head 所指向的第一个结点开始，首先访问结点，再沿着链方向到达后继结点，再访问结点，直至到达链表的最后一个结点。设 p 指向链表中的某结点，由结点 p 到达 p 的后继结点的语句是：

```
p=p.next;
```

算法如下：

```
public void output()                      //输出 head 指向的单向链表
{
   this.output(head);
```

```
    }
    public void output(OnelinkNode p)        //输出 p 指向的单向链表，重载
    {
       System.out.print(this.getClass().getName()+":  ");//显示类名
       while(p!=null)
       {
           System.out.print(p.data);
           p=p.next;
           if(p!=null)
              System.out.print(" -> ");
       }
       System.out.println();
    }
```

【例 2.2**】 单向链表逆转。**

设已建立一条单向链表，现欲将各结点的 next **链改为指向其前驱结点，使得单向链表逆转过来。算法描述如图** 2.8 **所示。**

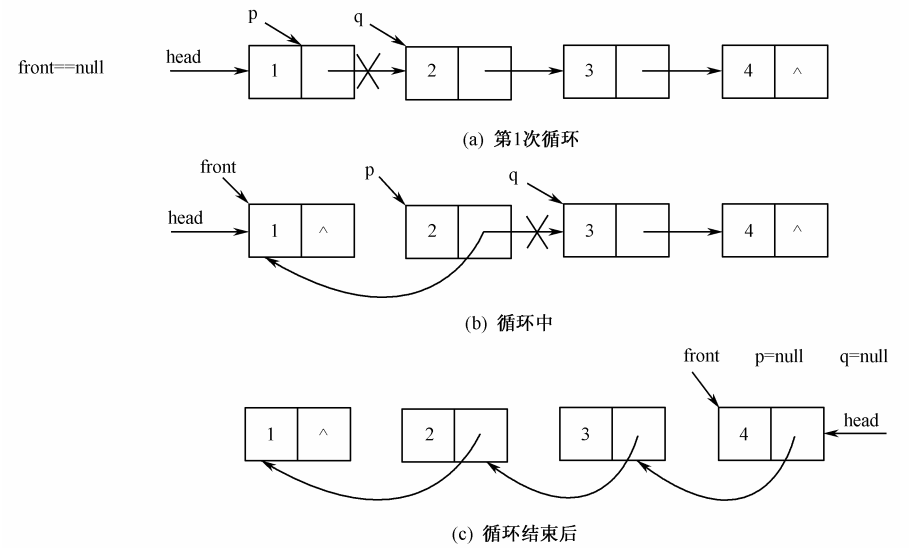


图 2.8 **单向链表逆转**

设 p **指向链表的某一结点，**front **和** q **分别指向** p **的前驱和后继结点，则使** p.next **指向其前驱结点的语句是：**

```
p.next=front;
```

算法描述如下：

- **第** 1 **次循环时，**front==null**，**p **指向链表的第一个结点，执行上述语句。**
- **以** p!=null **为循环条件，**front**、**p **和** q **沿链表方向依次前进，对于** p **指向的每一个结点，执行上述语句。**
- **循环结束后，**front **指向原链表的最后一个结点，该结点应成为新链表的第** 1 **个结点。**

由 head 指向，语句为：

```
head=front;
```

下面的程序声明 Onelink2 类继承单向链表 Onelink1 类，即继承了 Onelink1 类的成员变量 head 和 output()等方法，结点都是 OnelinkNode 类。程序如下：

```
import ds_java.OnelinkNode;
import ds_java.Onelink1;
public class Onelink2 extends Onelink1  //单向链表逆转
{
   public Onelink2()                    //构造空的单向链表
   {
      super();                          //执行超类的构造方法
   }
   public Onelink2(int n)               //以 n 个随机值建立单向链表
   {
      super(n);
   }
   public void reverse()                //将单向链表逆转
   {
      OnelinkNode p=this.head,q=null,front=null;
      while(p!=null)
      {
         q=p.next;
         p.next=front;                  //p.next 指向 p 结点的前驱结点
         front=p;
         p=q;
      }
      this.head=front;
   }
   public static void main(String args[])
   {
      Onelink2 h2=new Onelink2(8);
      h2.output();
      System.out.println("Reverse!");
      h2.reverse();
      h2.output();
   }
}
```

程序运行结果如下：

```
Onelink2:  7 -> 90 -> 43 -> 89 -> 34 -> 78 -> 96 -> 90
Reverse!
Onelink2:  90 -> 96 -> 78 -> 34 -> 89 -> 43 -> 90 -> 7
```

6）插入结点

在单向链表中插入新的结点，根据插入位置的不同，分 4 种情况讨论，如图 2.9 所示。

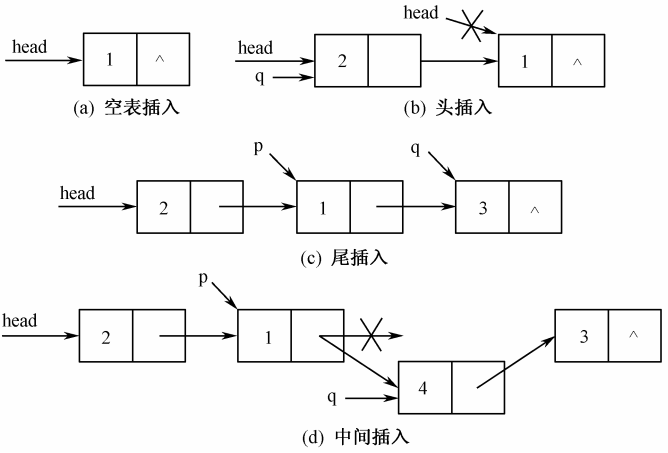


图 2.9　单向链表插入结点

① 空表插入。假定原链表为空（head==null），插入结点后，链表只有一个结点，由 hea
指向。语句如下：

```
head=new OnelinkNode(1);
```

② 头插入。设 head 指向非空链表的第一个结点，在结点 head 之前插入结点 q，插入
结点 q 作为链表的第一个结点，再由 head 指向。语句如下：

```
q=new OnelinkNode(2);
q.next=head;
head=q;
```

③ 尾插入。设 p 指向非空链表的最后一个结点，在结点 p 之后插入结点 q，形成新的
表。语句如下：

```
q=new OnelinkNode(3) ;
q.next=null;
p.next=q;
```

④ 中间插入。设 p 指向非空链表中的某结点，在结点 p 之后插入结点 q，形成新的链表
语句如下：

```
q=new OnelinkNode(4);
q.next=p.next;
p.next=q;
```

事实上，尾插入的操作是中间插入的特例。只是在插入前，p 指向最后一个结点，即

```
p.next=null
```

由此可见，在单向链表中插入结点，只要修改相关的几条链，而不需移动数据元素。

由于单向链表中的每个结点只有一个指向后继结点的链，而没有指向前驱结点的链，所
对于任意一个结点 p，可以很方便地访问 p 的后继结点。而访问 p 的前驱结点则较麻烦，必
从 head 开始重新查找。所以，如果需要在结点 p 之前插入结点，则必须修改 p 前驱结点的 ne
链，操作上较麻烦。此处算法从略，请读者思考。

【例 2.3】 单向链表插入排序。

本例演示在单向链表中的各种不同位置处插入结点。在已排序（sorted）的单向链表中插入 k 值，k 值的插入位置取决于 k 值的大小，插入后的链表仍然是排序的。

在已排序的单向链表中插入 k 值的算法（insert 方法）描述如下：

- 当 head==null 时，链表为空，创建值为 k 的结点作为链表的第一个结点，由 head 指向，这是空链表插入结点情况。
- 当 head!=null 时，比较 k 与 head.data 的值。如果 k 值小，将该结点插入到第一个结点之前，由 head 指向，这是表头插入结点情况。
- 否则，该结点需要插入在链表中间（尾）。设 q 引用从 head 开始依次查找，当 k>q.data 时，继续沿链表方向前进；当循环结束时，k 值结点应该插入在 q 结点之前，即 p 结点之后（由于单向链表无法直接访问结点的后继，所以算法中增设 p 作为 q 的前驱结点，q 每前进一步，p 也跟随前进），如图 2.10 所示。

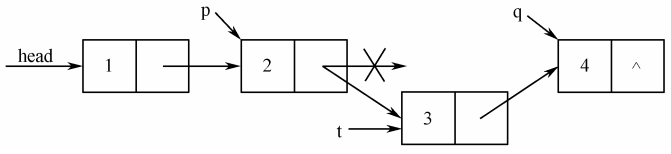


图 2.10 在已排序的单向链表中插入结点

Onelink3 类继承单向链表 Onelink1 类，继承超类的 head 成员变量和 output()等方法，结点结构是 OnelinkNode 类。程序如下：

```
import ds_java.OnelinkNode;
import ds_java.Onelink1;
public class Onelink3 extends Onelink1  //单向链表插入排序
{
   Onelink3()                           //建立空链表
   {
      super();
   }
   Onelink3(int n)                      //n 个随机值插入单向链表
   {
      System.out.print("insert:   ");
      for(int i=0;i<n;i++)
      {
         int k=(int)(Math.random()*100);
         System.out.print(k+"  ");
         insert(k);
      }
      System.out.println();
   }
   public void insert(int k)            //k 值插入已排序的单向链表
   {
```

```
        OnelinkNode p=null,q=null,t;
        t=new OnelinkNode(k);              //创建 k 值结点 t
        if(head==null)
           head=t;                         //空表插入
        else
        {
           if(k<head.data)                 //头插入
           {
              t.next=head;
              head=t;
           }
           else                            //表中、表尾插入
           {
              q=head;                      //查找 t 应插入的位置
              while(q!=null && k>q.data)
              {
                 p=q;                      //p 是 q 的前驱结点
                 q=q.next;
              }
              t.next=p.next;               //将 t 插入在 p 结点之后
              p.next=t;
           }
        }
     }
     public static void main(String args[])
     {
        (new Onelink3(10)).output();
     }
  }
```

程序运行结果如下：

```
insert:   68  45  11  50  15  97  15  85  35  87
Onelink3:  11 -> 15 -> 15 -> 35 -> 45 -> 50 -> 68 -> 85 -> 87 -> 97
```

7）删除结点

在单向链表中删除给定位置的结点，需要从逻辑上改变链接关系并将该结点所用的存储单元归还给系统。单向链表中某结点被删除后，该结点所占的存储单元由 Java 系统自动回收。

根据删除结点位置的不同，分 3 种情况讨论，如图 2.11 所示。

① 删除单结点链表，只要使链表的引用 head 为空即可。语句如下：

```
head=null;
```

② 删除链表第 1 个结点，只要将 head 指向链表第 1 个结点的后继结点即可。语句如下：

```
head=head.next;
```

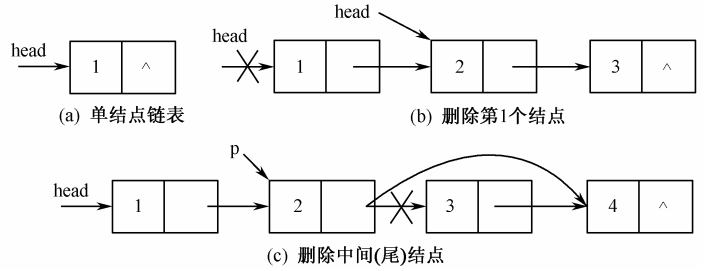


图 2.11　单向链表删除结点

③ 删除链表中间（尾）结点，设 p 指向单向链表中的某一结点，删除 p 的后继结点的语句如下：

```
p.next=(p.next).next;
```

执行上述语句之后，则建立了新的链接关系，替代了原链接关系。因此，在单向链表中删除结点，只要修改相关的几条链，而不需移动数据。

如果需要删除结点 p 自己，则必须修改 p 前驱结点的 next 链。由于单向链表中的结点没有指向前驱结点的链，无法直接修改 p 前驱结点的 next 链。所以，在单向链表中，要删除 p 的后继结点，操作简单；而要删除结点 p 自己，则操作比较麻烦。

4．两种存储结构性能的比较

线性表两种存储结构性能的比较主要从以下几个方面进行：

1）直接访问元素的性能

顺序表能够直接访问数据元素，即顺序结构只要给出数组的下标就可以引用到数组中的任何一个数据元素；而链式结构不能直接访问任意一个数据元素，只能从链表的第一个结点开始沿着链的方向，依次查找后继结点，直至到达所需访问的数据元素结点，才可以访问该结点的数据元素。

2）存储空间的利用

顺序表存在使用空间的浪费与溢出问题。顺序表进行插入或删除操作时，要判断顺序表是否为空，或是否已满。当顺序表所占的存储空间已满时，则无法插入。而链表只要判断是否为空就可以了，无需判断是否已满。因为链表每插入一个结点，就向系统申请一个存储单元，只要系统资源够用，系统就会分配存储单元。

3）存储密度

顺序表的全部空间都用来存放数据元素，因此存储密度高；而链表中每个结点都要包含其后继结点的地址，因此存储密度较低。

4）插入和删除操作

顺序表的插入和删除操作很不方便，插入和删除操作有时需要移动大量元素；而链表则容易进行插入和删除操作，只要简单地改动相关结点的链即可，不需移动数据元素。

5）查找和排序

顺序表和链表都可以采用查找和排序的一些简单算法，包括顺序查找、插入排序、选择排序等。此外，顺序表还可以采用多种复杂的查找和排序算法，包括折半查找、快速排序、堆排序等（本书第 3 章和第 7 章将详细介绍各种查找和排序算法）。

综上所述，两种存储结构各有所长，使用时可根据具体问题而有所选择，或采用两者综合

起来的复杂结构。本书后面几章将予以介绍。

2.2.2 单向循环链表

单向链表中，将最后一个结点的链设置为指向链表的第一个结点，则该链表成为环状，称为单向循环链表（circular linked list），如图 2.12 所示。

由图 2.12（c）可以看出，单向循环链表的所有结点链接成一条回路。即从链表中任何一个结点出发，沿着链的方向，访问链表中所有结点之后，又回到出发点。

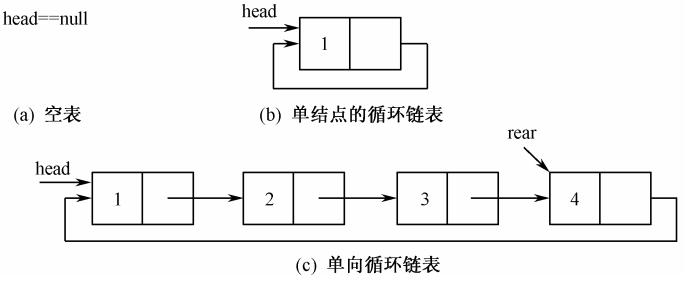


图 2.12 单向循环链表

为了对循环链表进行操作，必须定义一个首次访问的结点，习惯上仍用 head 指向。设尾指针 rear 指向循环链表的最后一个结点（相对于 head 而言），则有

```
rear.next=head;
```

当 head.next==head 时，循环链表只有一个结点。当 head==null 时，循环链表为空。

【例 2.4】 以单向循环链表求解约瑟夫环问题。

在例 2.1 中，我们已用顺序表解出约瑟夫环问题，本例将用单向循环链表来解约瑟夫环问题，借此表现：对于同一个算法，采用两种不同存储结构的两种程序实现。

当 n=5，s=1，d=2 时，以单向循环链表求解约瑟夫环问题的执行过程如图 2.13 所示。

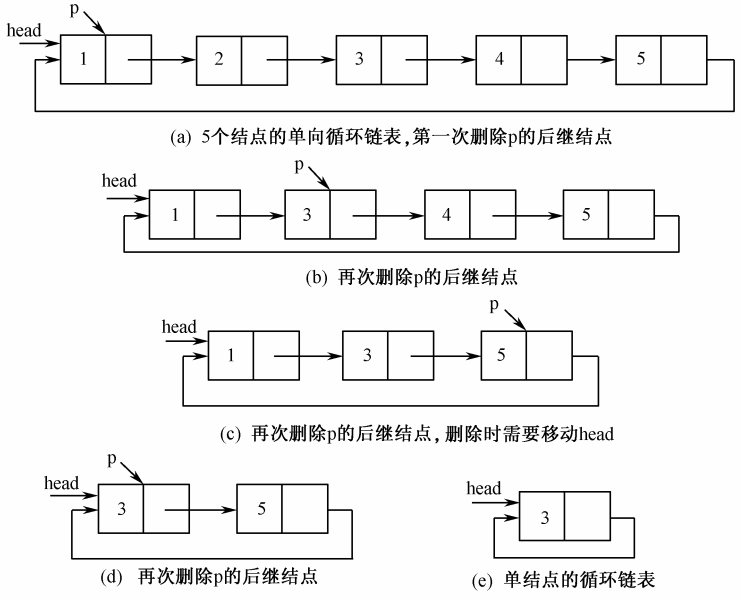


图 2.13 以单向循环链表求解约瑟夫环问题的执行过程

算法描述如下：

- **建立一个具有 n 个结点的单向循环链表，结点元素值为 1～n。**
- **从第 s 个结点开始，依次计数，每数到 d，就删除该结点。**
- **重复上一步，依次选中 n-1 个结点出环。**
- **输出循环链表中所剩的最后一个结点值。**

由于采用循环链表结构，当一个结点出环时，就可删除相应结点，此时不必移动其他结点

程序如下：

```
import ds_java.OnelinkNode;
import ds_java.Onelink1;
public class Josephus2 extends Onelink1     //单向循环链表，解约瑟夫环
{
   Josephus2()                              //构造空的单向循环链表
   {
      super();
   }
   Josephus2(int n)                         //建立 n 个结点的单向循环链表
   {                                        //链表结点值为 1 到 n
      this();
      int i=1;
      OnelinkNode q,rear;
      if(n>0)
      {
         head=new OnelinkNode(i);
         head.next=head;
         rear=head;
         while(i<n)
         {
            i++;
            q=new OnelinkNode(i);
            q.next=head;
            rear.next=q;
            rear=q;
         }
      }
   }
   public static void main(String args[])
   {
      int n=5,s=1,d=2;
      Josephus2 h1=new Josephus2(n);
      h1.output();
      h1.display(s,d);
```

```
    }
    public void display(int s,int d)              //解约瑟夫环
    {
        int j=0;
        OnelinkNode p=head;
        while(j<s-1)                              //计数起始点
        {
            j++;
            p=p.next;
        }
        while(p.next!=p)                          //多于一个结点时循环
        {
            j=1;
            while(j<d-1)                          //计数
            {
                j++;
                p=p.next;
            }
            delete(p);                            //删除 p 的后继结点
            p=p.next;
            this.output();
        }
    }
    public void delete(OnelinkNode p)             //删除 p 的后继结点
    {
        OnelinkNode q=p.next;                     //q 是 p 的后继结点
        System.out.print("delete:  "+q.data+"  ");
        if(q==head)                               //欲删除 head 指向结点时,
            head=q.next;                          //要将 head 向后移动
        p.next=q.next;                            //删除 q
    }
    public void output()                          //输出单向链表的各个结点值
    {
        OnelinkNode p=head;
        System.out.print(this.getClass().getName()+":  ");
        do
        {
            System.out.print(p.data+" -> ");
            p=p.next;
        }while(p!=head);
        System.out.println();
    }
```

```
}
```

程序运行结果如下：

```
Josephus2:  1 -> 2 -> 3 -> 4 -> 5 ->
delete:  2  Josephus2:  1 -> 3 -> 4 -> 5 ->
delete:  4  Josephus2:  1 -> 3 -> 5 ->
delete:  1  Josephus2:  3 -> 5 ->
delete:  5  Josephus2:  3 ->
```

由此可见，在单向（循环）链表中，通常删除 p **的后继结点，而不直接删除结点** p **自己而且还要判断删除结点是否为** head **指向的结点，如果要删除** head **指向的结点，则必须将** hea **指向其后继结点。如果删除了** head **指向的结点，则在输出时，条件** p!=head **永远无法成立造成死循环。**

思考题：在 delete **方法中，只使用一个对象** p **来删除** p **的后继结点。**

2.2.3 双向链表

在单向链表中，每个结点只有一个链用于指向后继结点，而没有链记载前驱结点的信息此时若要查找前驱结点，必须从链表的头指针开始重新沿着链表方向逐个检测，这样操作很不方便。因此，如果经常需要进行既向前又向后操作，采用双向链表比较合适。

双向链表（doubly linked list**）的每个结点除了成员变量** data **之外，还有两个成员变量链** prior **指向前驱结点，**next **指向后继结点。图** 2.14 **给出了双向链表的结构示意图。**

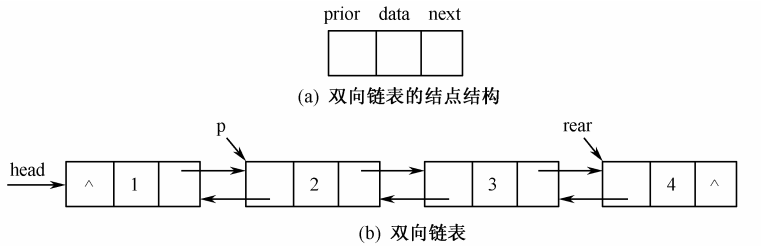


图 2.14 **双向链表结构示意图**

1．双向链表的结点类

用 Java **描述双向链表的结点，声明** TwolinkNode **类如下：**

```
package ds_java;
public class TwolinkNode                    //双向链表的结点类
{
    public int data;                        //存放结点值
    public TwolinkNode prior,next;          //前驱、后继结点的引用
    public TwolinkNode(int k)               //构造值为 k 的结点
    {
        data=k;
        prior=next=null;
    }
    public TwolinkNode()                    //构造方法，重载
```

```
    {
        this(0);
    }
}
```

TwolinkNode 类的一个对象表示双向链表中的一个结点。

2．双向链表类

用 Java 语言描述双向链表，声明 Twolink1 类如下：

```
import ds_java.TwolinkNode;                    //引用双向链表的结点类
public class Twolink1                          //双向链表类
{
    protected TwolinkNode head;
    public Twolink1()                          //构造空的双链表
    {
        head=null;                             //指向链表的第一个结点
    }
}
```

Twolink1 类的一个对象表示一条双向链表，其构造方法建立一条空链。

设 rear 指向双向链表的最后一个结点，由于线性表的第一个数据元素没有前驱数据元素最后一个数据元素没有后继数据元素，所以有

```
head.prior==null
rear.next==null
```

再设 p 指向双向链表中的某一结点（除首结点和最后一个结点），则双向链表的本质特征如下：

```
(p.prior).next=p
(p.next).prior=p
```

双向链表比单向链表在结点结构上增加了一个链，但给链表的操作带来很大的便利，能够沿链表向前走，从而直接找到前驱结点。

思考题：从以上声明可以看出，TwolinkNode 类比 OnelinkNode 类多一个链 prior，两者很相似，都有 data 和 next 成员，都是自引用的类。而此处却没有将 TwolinkNode 类声明为继承 OnelinkNode 类，也没有将 Twolink1 类声明为继承 Onelink1 类，请问为什么？

结论：TwolinkNode 类不需要继承 OnelinkNode 类，而 Twolink1 类不能继承 Onelink1 类。

如果 TwolinkNode 类继承 OnelinkNode 类，声明为：

```
public class TwolinkNode extends OnelinkNode
{
    public TwolinkNode prior;
}
```

则 TwolinkNode 类从超类继承了一个可用的 int 成员 data，但从超类继承的 next 链指向的却是 OnelinkNode，而不是我们希望的 TwolinkNode 类，必须重新声明 next 成员，所以没有必要继承。

同理，Onelink1 类中的 head 指向 OnelinkNode 类，Twolink1 类中的 head 应指向 TwolinkNod

类。如果 Twolink1 类继承 Onelink1 类，则继承来的成员 head 指向的是超类，而不是引用自己的类，这将引起混淆。所以不能继承。

3．双向链表的操作

将下述对双向链表的操作作为 Onelink1 类中的成员方法，并将 Onelink1 类编译后放在 ds_ja包中供其他类引用。

1）判断双向链表是否为空

当 head==null 时，为空链表，算法如下：

```
public boolean isEmpty()                    //判断双向链表是否为空
{
    return head==null;
}
```

双向链表的其他基本操作与单向链表类似，在此不重复讨论。

2）插入结点

根据插入位置的不同，分 3 种情况讨论，如图 2.15 所示。

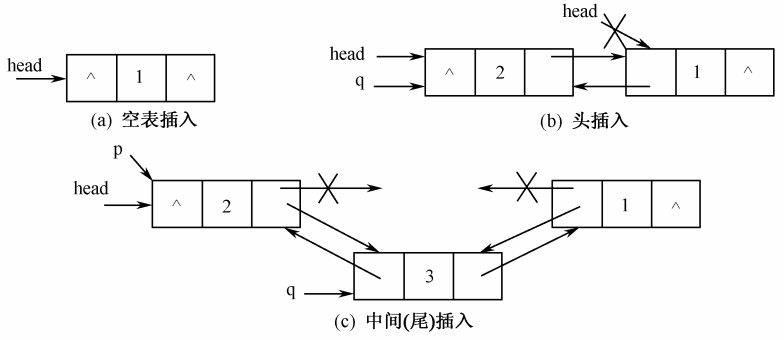


图 2.15　双向链表插入结点

① 空表插入。假定原链表为空（head==null），插入结点后，链表只有一个结点，由 hea指向。语句如下：

```
head=new TwolinkNode(1);
```

② 头插入。在非空链表的第一个结点之前插入 q 结点，插入后 q 结点作为链表的第一结点，由 head 指向。语句如下：

```
q=new TwolinkNode(2);
q.next=head;
head.prior=q;
head=q;
```

③ 中间（尾）插入。在非空链表的 p 结点之后插入 q 结点，形成新的链表。语句如下

```
q=new TwolinkNode(3);
q.prior=p;
q.next=p.next;
(p.next).prior=q;
p.next=q;
```

插入前，当 p 指向最后一个结点时，p.next==null，因此尾插入是中间插入的特例。

思考题：对于中间（尾）插入操作，如果在结点 p 之前插入结点 q 也很方便，应该怎样实现插入操作？

3）删除结点

在双向链表中删除给定位置的结点，根据删除结点位置的不同，分 3 种情况讨论，如图 2.16 所示。

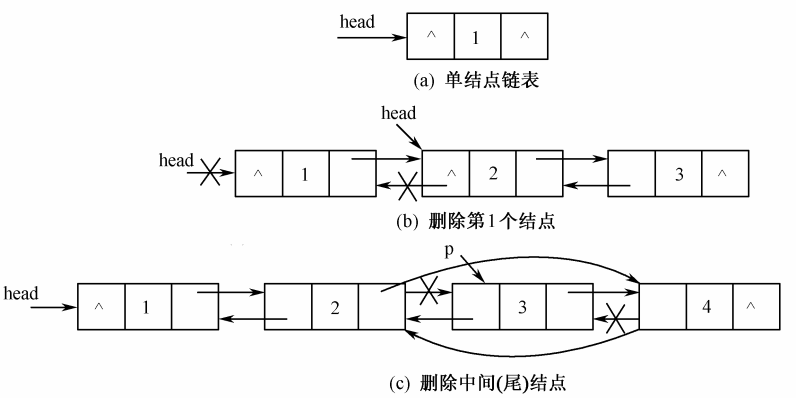


图 2.16　双向链表删除结点

① 删除单结点链表，只要使链表的引用 head 为空即可。语句如下：

```
head=null;
```

② 删除链表第 1 个结点，将 head 指向链表第 1 个结点的后继结点，并设置 head 指向结点的 prior 链为空值。语句如下：

```
head=head.next;
head.prior=null;
```

③ 删除链表中间（尾）结点，在双向链表中删除指定位置结点 p 的语句如下：

```
(p.prior).next=p.next;
(p.next).prior=p.prior;
```

在双向链表中可以删除结点 p 自己。因为删除结点 p 时，可以直接修改前驱和后继结点的链。

4）查找结点

在双向链表中可以沿 prior 和 next 移动，这使得查找操作很方便。假设某双向链表的结点值是按递增次序排列的，查找给定值 *k* 的过程为：从双向链表中的任何一个结点 p 开始，如果结点 p 的值等于 *k*，则查找成功。如果结点 p 的值小于 *k*，则 p 向前移动；否则 p 向后移动，直至找到 *k* 值；或走到链表两端时仍未找到 *k* 值，查找不成功。

思考题：请构造已排序的双向链表，并实现查找、插入、删除操作的方法。

2.2.4　双向循环链表

双向链表中，如果最后一个结点 rear 的 next 链指向链表的第 1 个结点 head，而链表第一个结点 head 的 prior 链指向最后一个结点，便形成双向循环链表，此时有下式成立：

```
rear.next==head 且 head.prior==rear
```

当 head==null 时，为空链表。当 head.next==head 且 head.prior==head 时，为单结点链表

如图 2.17 所示。

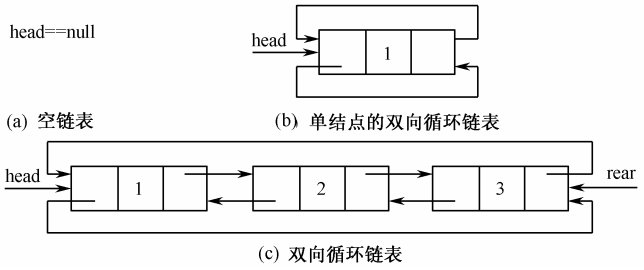


图 2.17　双向循环链表

双向循环链表的查找、插入和删除操作类似双向链表。

2.3　串

串（string）可以看成是每个结点仅由一个字符组成的线性表。串也称为字符串。

2.3.1　串的定义

串是由多个字符组成的有限序列。一个串中包含的字符个数，称为这个串的长度。长度为 0 的串称为空串，即空串不包含任何字符。Java 中用单引号将字符括起来，用双引号将串括起来。例如，

```
s1="data"                          //串长度为 4
s2="data  structure"
s3=""                              //空串，长度为 0
s4="  "                            //空格串，长度为 2
```

s1，s2，s3 和 s4 称为串名。

1．字符的比较

串中所能包含的字符依赖于所使用的具体字符集，按字符在字符集中的次序可以规定字符的大小。字母 A～Z，a～z 和数字 0～9 在字符集中是按顺序排列的。可对两个字符进行比较，如

```
'A'<'a'                            //比较结果为 true
'1'>='9'                           //比较结果为 false
```

目前通用的字符集有 ASCII 码，而 Java 采用的是 Unicode 码。

2．串的比较

按串中字符的次序，可以对两个字符串进行比较，从而定义两个串的大小，如

```
'ABC'<'BCD'                        //比较结果为 true
```

3．子串

由串中若干个连续的字符组成的一个子序列称为该串的一个子串（substring）。相应地，原串称为该子串的主串。空串是任何串的子串。一个串 s 也可看成是自身的子串，除本身外，它的其他子串都称为真子串。例如，上述 s1 是 s2 的子串。

4．子串在主串的序号

串 s 中的一个字符 c 的位置用一个整数表示，称为字符 c 在串 s 中的序号（index）。串的第一个字符的序号为 1。子串的序号是该子串的第一个字符在主串中的序号。例如，s1 在 s 中的序号为 1，s4 在 s2 中的序号为 5。如果串 sub 不是串 str 的子串，可以说，sub 在 str 中的序号为 0。

2.3.2 串的存储结构

由于串是由字符组成的线性表，所以可用顺序存储结构和链式存储结构两种方式来存储。

1．串的顺序存储结构

串的顺序存储结构就是用静态数组来存储串。将串中的字符依次存储在数组的相邻单元中，如图 2.18 所示。

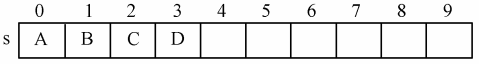


图 2.18 串的顺序存储结构

图 2.18 中，串 s=〃ABCD〃，s 的存储单元有 10 个，而 s 的长度只有 4。所以，用 Java 数组实现串，除了要申请一个字符数组表示串之外，还要有一个整型变量记载串的实际长度。

用数组存储串，优点是存储密度高，缺点是必须预先估计每个串可能的最大长度作为数组的容量。如果这个估计的数组长度不够，则在执行过程中将产生数据溢出错误，反之将可能造成空间的浪费。

另一种方法是在建立一个串时，按实际需要分配存储，即在执行过程中动态地分配，这就是链式存储结构。

2．串的链式存储结构

串的链式存储结构就是单向链表，每个结点的值是一个字符，如图 2.19 所示。

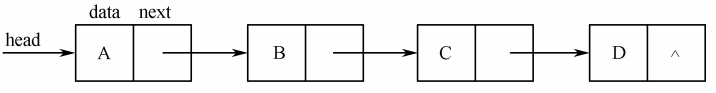


图 2.19 串的链式存储结构

图 2.19 中，串 s= "ABCD "，s 用单向链表存储，s 的长度是 4，相应的链表有 4 个结点。

2.3.3 串的操作

1．串的基本操作

串的基本操作有以下 5 种：

- 建立一个空串。
- 求串的长度。
- 连接两个串。
- 求子串。

- **查找子串。**

串的其他操作都可以用这些操作来实现。

2．**串的顺序存储结构实现基本操作**

串的顺序存储结构与顺序表相似，只是数据元素的类型不同而已。以顺序存储结构实现串的 String1 **类声明如下：**

```
public class String1                    //串的数组实现
{
   private char table[];                //用字符数组表示串
   private int number=0;                //记载串的实际长度
   public String1()                     //构造空串
   {
      table=new char[80];
      number=0;
   }
}
```

String1 **类表示串，成员变量有** table **和** number。table **将串存储在一个字符数组中，**numb**表示串的长度。**String1 **类实现串的** 5 **种基本操作的算法如下。**

（1**）构造方法创建一个串。**

```
public String1(int n)                   //构造 n 个存储单元的空串
{
   table=new char [n];
   number=0;
}
public String1()                        //构造 10 个存储单元的空串
{
   this(10);
}
public String1(char c)                  //构造一个字符的串
{
   this();
   table[0]=c;
   number=1;
}
public String1(char c[])                //以一个字符数组构造串
{
   this(c.length);
   System.arraycopy(c,0,table,0,c.length);      //复制数组
   number=c.length;
}
```

（2**）**length1()**方法返回串的长度。**

```
public int length1()                    //求串的长度
```

```
    {
        return number;
    }
```

（3）连接两个串的 concat()方法有两种重载的实现，分别是连接一个串与一个字符和连接两个串。

```
    public void concat(char c)                    //连接一个串与一个字符
    {
        this.table[number]=c;
        number++;
    }
    public void concat(String1 s2)                //连接两个串，将另一串 s2 连接在串之后
    {
        for(int i=0;i<s2.length1();i++)
            this.concat(s2.table[i]);
    }
```

（4）substring()方法返回串的子串。

substring()方法返回串中从序号 i 开始的长度为 n 的子串。设串的长度为 this.length1()，应满足 $1 \leq i \leq i+n \leq$ this.lenght1()，否则返回空串；若 $n=0$，也返回空串。

```
    public String1 substring(int i,int n)
    {                                             //返回串中从序号 i 开始的长度为 n 的子
        String1 sub=new String1();
        int j=0;
        if(i>=1 && i<=this.length1())
        {
            i--;                                  //下标 i 比序号 i 小 1
            while(j+i<this.length1() && j<n)
            {
                sub.table[j]=this.table[j+i];
                j++;
            }
            sub.number=j;
        }
        return sub;
    }
```

（5）查找子串的 indexof()方法返回子串 sub 的序号。

在串中查找与串 sub 相同的子串，若查找成功，返回子串的序号，即子串首次出现时第一个字符的序号，否则返回 0。

```
    public int indexof(String1 sub)               //返回子串 sub，子串的序号
    {
        int i=0,j=0;
        boolean yes=false;
```

```
        while(sub.length1()>0 && i<number && !yes)
        {
            j=0;
            while(j<sub.length1() && this.table[i+j]==sub.table[j])
                j++;
            if(j>=sub.length1())
                yes=true;
            else
                i++;
        }
        if(yes)
            return i+1;                          //序号为下标 i+1
        else
            return 0;
    }
```

3．串的链式存储结构实现基本操作

串的链式存储结构的结点类与单向链表的结点类相似，只是数据元素的类型不同而已。串的链式存储结构的结点类 StringNode 声明如下：

```
    package ds_java;                             //声明包
    public class StringNode                      //单向链表的结点结构
    {
        public char data;                        //存放结点值
        public StringNode next;                  //后继结点的引用
        public StringNode(char c)                //构造值为 c 的结点
        {
            data=c;
            next=null;
        }
        public StringNode()
        {
            this('?');
        }
    }
```

以链式存储结构实现串的 String2 类声明如下：

```
    import ds_java.StringNode;
    public class String2                         //串的链式存储结构
    {
        private StringNode head,rear;            //单向链表的首、尾结点引用
        private int number=0;                    //记载串的实际长度
    }
```

String2 类用单向链表实现串，成员变量有 head，rear 和 number。head 和 rear 分别指向

向链表的第一个和最后一个结点，number 表示串的长度。String2 类实现串的基本操作的方法及参数与 String1 类相同，算法如下。

（1）构造方法创建一个串。

```
public String2()                          //构造空串
{
    head=rear=null;
    number=0;
}
public String2(char c)                    //构造一个字符的串
{
    number=1;
}
```

（2）length2()方法返回串的长度。

```
public int length2()                      //求串的长度
{
    return number;
}
```

（3）连接两个串的 concat()方法有两种重载的实现，分别是连接一个串与一个字符和连接两个串。

```
public void concat(char c)                //连接一个串与一个字符
{
    StringNode p,q;
    if(this.head==null)
    {
        this.head=new StringNode(c);
        this.rear=head;
    }
    else
    {
        q=new StringNode(c);
        this.rear.next=q;
        this.rear=q;
    }
    number++;
}
public void concat(String2 s2)            //连接两个串
{
    StringNode q=s2.head;
    while(q!=null)
    {
        this.concat(q.data);
```

```
            q=q.next;
        }
    }
```

思考题：在 String2 **类中增加下列方法：**

- **写出构造方法，以一个字符数组构造串。**

```
public String2(char c[])
```

- **写出** substring **方法，返回串中从序号** *i* **开始的长度为** *n* **的子串。**

```
public String2 substring(int i,int n)
```

- **写出** indexof **方法，查找子串** sub**，返回子串的序号。**

```
public int indexof(String2 sub)
```

4．**串的其他操作**

对串的操作，除了前面讲过的 5 **种基本操作外，还有插入、删除、替换、逆转等其他操作这些操作都可用前面** 5 **种操作实现。**

（1**）串的插入：**

```
public String3 insert(int i,String3 s2)
```

将串 s2 **插入到串的第** *i* **位置处，** $1 \le i \le \text{this.length1()}$ **。算法描述如下：**

- **用** substring **操作将串以** *i* **为界分成两个子串，前** *i*-1 **个字符组成** sub1**，后** this.length1() $-i+1$ **个字符组成** sub2**。**
- **再用** concat **操作将** sub1**、**s2 **和** sub2 **依次连接起来构成一个新串** news1**。**

串的插入算法描述如图 2.20 **所示。**

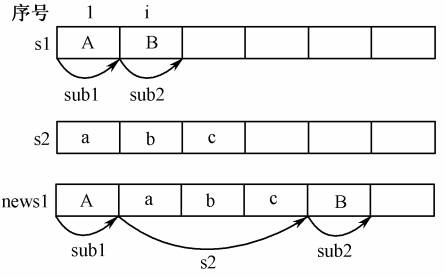


图 2.20　**串的插入**

串的插入算法如下：

```
public String3 insert(int i,String3 s2)
{                                          //将 s2 插入到主串第 i 位置处
   String1 sub1,sub2;
   sub1=this.substring(1,i-1);
   sub2=this.substring(i,this.length1()-i+1);
   String3 news1=new String3();
   news1.concat(sub1);
   news1.concat(s2);
   news1.concat(sub2);
   return news1;
```

```
}
```

（2）串的删除：

```
public String3 delete(int i,int n)
```

删除串中从第 i **位置开始的长度为** n **的子串，**$1 \le i \le i+n \le \text{this.length1()}$ **。算法描述如下**

- **用** substring **操作将串分成** 3 **个子串** sub1、sub2 **和** sub3，**前** $i-1$ **个字符组成** sub1，**从第** i **个字符开始的长度为** n **的子串为** sub2，**后** this.length()$-i-n+1$ **个字符组成** sub3。
- **用** concat **操作将** sub1 **和** sub3 **依次连接起来构成一个新串** news1。

串的删除算法描述如图 2.21 **所示。**

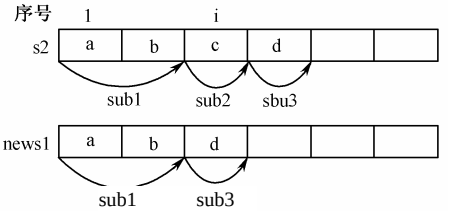


图 2.21　**串的删除**

串的删除算法如下：

```
public String3 delete(int i,int n)
{                                          //删除串中第 i 位置开始的长度为 n 的子
   String1 sub1,sub2,sub3;
   sub1=this.substring(1,i-1);
   sub2=this.substring(i,n);
   sub3=this.substring(i+n,this.length1()-i-n+1);
   String3 news1=new String3();
   news1.concat(sub1);
   news1.concat(sub3);
   return news1;
}
```

（3）串的替换：

```
public String3 replace(String3 oldstr,String3 newstr)
```

将串中 oldstr **子串替换成** newstr **子串。算法描述如下：**

- **用** substring **操作将串分成** 3 **个子串，前** $i-1$ **个字符组成** sub1，**中间子串** sub2 **与参数串** oldstr **相同，之后子串组成** sub3。
- **用** concat **操作将** sub1, newstr **和** sub3 **依次连接起来构成一个新串** news1。

串的替换算法描述如图 2.22 **所示。**

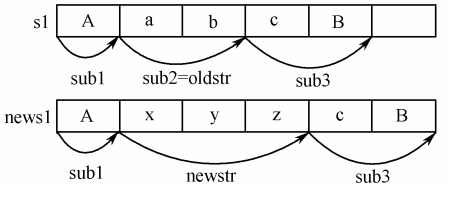


图 2.22　**串的替换**

串的替换算法如下：

```
public String3 replace(String3 oldstr,String3 newstr)
{                                          //将串中 oldstr 子串替换成 newstr 子
   int i,n;
   String1 sub1,sub2,sub3;
   String3 news1=new String3();
   i=indexof(oldstr);
   if(i!=-1)
   {
      sub1=this.substring(1,i-1);
      sub2=this.substring(i,this.length1()-i+1);
      n=oldstr.length1();
      sub3=this.substring(i+n,this.length1()-i-n+1);
      news1.concat(sub1);
      news1.concat(newstr);
      news1.concat(sub3);
   }
   return news1;
}
```

（4）串的逆转：

```
public String3 reverse()
```

将串逆转，算法描述如下：

- **初始化** news1 **为空串。**
- **设** *i* **为最后一个字符的位置。**
- **用** substring **操作取得串中的第** *i* **个字符组成的子串** sub1**。**
- **用** concat **操作将** sub1 **连接到一个新串** news1 **之后。**
- **重复以上两步，循环次数为串的长度。**

串的逆转算法描述如图 2.23 **所示。**

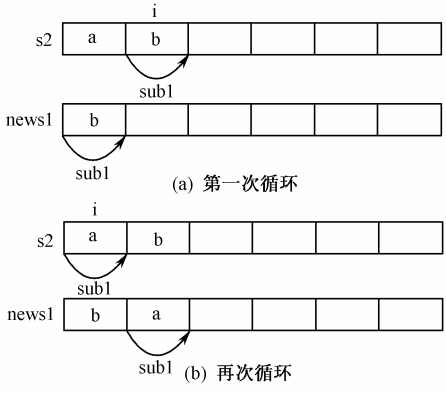


图 2.23　**串的逆转**

串的逆转算法如下：

```
public String3 reverse()                    //将串逆转
{
    int i;
    String1 sub1;
    String3 news1=new String3();
    for(i=this.length1();i>=1;i--)
    {
        sub1=this.substring(i,1);
        news1.concat(sub1);
    }
    return news1;
}
```

习　题　2

2.1　**在单向链表中：**

1．**返回第** *i* **个结点的值。**

2．**求各结点的数值之和。**

3．**以一个数组中的多个值构造单向链表，构造方法声明如下：**

```
public Onelink1(int table[])
```

4．**查找值为** *k* **的结点，返回值为** boolean **或** OnelinkNode **类型。**

5．**删除值为** *x* **的结点。**

6．**将两条单向链表连接起来，形成一条单向链表。**

7．**将两条单向链表首尾相接，合并成一条单向循环链表。**

2.2　**在双向线性链表中：**

1．**查找值为** *x* **的结点。**

2．**删除值为** *x* **的结点。**

3．**实现插入排序。**

2.3　**在** String2 **类中，增加求子串、查找子串的方法。**

2.4　**在** String3 **类中，用串的插入与删除操作实现串的替换。**

2.5　**用链式存储的串实现插入、删除、替换和逆转功能。**

2.6　**寻找两个字符串中的最长公共子串。**

实　习　2

1．**实验目的**

理解线性表的基本概念，熟练运用 Java **自引用类的方式实现线性链表的基本操作。**

2．**题意**

在一条双向链表中，已知每个结点的 next **链指向后继结点，而** prior **链为空。设计一个方法，使每个结点的** prior **指向它的前驱结点，形成双向循环链表。**

第 3 章　排　　序

排序（sorting）是将若干个数据元素按关键字的值以递增（或递减）次序排列的过程。我们在生活中经常用到排序，如查看学生情况时，如果预先按班级或学号将学生数据元素排好序，则可以大大提高查看效率。

排序可以提高查找效率，因此排序在计算机数据处理中有着广泛的应用。本章首先介绍排序的基本概念，然后讨论插入、交换、选择、归并等多种经典排序算法，并讨论各种排序算法所适用的存储结构，以及比较各排序算法的性能。

建议本章授课 6 学时，实验 4 学时。

3.1　排序的基本概念

1．数据序列

数据序列（datalist）是待排序的数据元素的有限集合。例如，一个班级的学生信息组成一个待排序的数据序列，每个学生信息则是待处理的数据元素。

2．关键字

通常数据元素由多个数据项组成，以其中某个数据项作为排序依据，则该数据项称为关键字（key）。例如，学生信息由系别、专业、班级、学号、姓名、家庭地址等多个数据项组成，如果按学号进行排序，则学号成为关键字。

在数据序列中，如果数据元素不同则关键字一定不同，则这样的关键字称为主关键字（primary key）。主关键字是数据元素的惟一标识。用主关键字进行排序得到的结果一定是惟一的。例如，学号是惟一的，可以作为主关键字，而姓名则可能重复，依据姓名排序的结果不惟一。表 3-1 所示的是按学号排序的学生成绩表，其中学号是惟一的，姓名、成绩则不是惟一的。

表 3-1　学生成绩表

学号	姓名	成绩
20020001	王红	85
20020002	张明	90
20020003	吴宁	78
20020004	张明	85

3．排序算法的稳定性

在数据序列中，如果有两个数据元素 r_i 和 r_j，它们的关键字 k_i 等于 k_j，且在未排序时，r_i 位于 r_j 之前。如果排序后，元素 r_i 仍在 r_j 之前，则称这样的排序算法是稳定的（stable），否则是不稳定的。

4．内排序与外排序

排序问题一般分为内排序和外排序两类：

- 内排序——在待排序的数据序列中，数据元素个数较少，整个排序过程中所有的数据元素都可以保留在内存，则这样的排序称为内排序。
- 外排序——待排序的数据元素非常多，以至于它们必须存储在磁盘等外部存储介质上，则这样的排序称为外排序。

5．排序算法的性能评价

衡量排序算法的性能好坏的重要标志是算法的时间复杂度和空间复杂度：

- 排序算法的时间复杂度——指算法执行中的数据比较次数、数据移动次数与待排序数据序列的元素个数之间的关系。
- 排序算法的空间复杂度——指算法执行中，除待排序数据序列本身所占用的内存空间外，需要的附加内存空间与待排序数据序列的元素个数之间的关系。

因此，好的排序算法应该尽可能减少运行时间，占用较少的额外空间。

本章仅讨论内排序算法，并且主要考虑按关键字递增的排序。

3.2 插入排序

插入排序（insertion sort）的基本思想是：每趟将一个待排序的数据元素，按其关键字大小，插入到已排序的数据序列中，使得插入后的数据序列仍是已排序的，直到全部元素插入完毕。

插入排序算法有 3 种：直接插入排序，折半插入排序和希尔排序。其中，折半插入排序算法将在第 7 章中介绍。

3.2.1 直接插入排序

1．直接插入排序算法

直接插入排序（straight insertion sort）的基本思想是：当插入第 i 个数据元素 k 时，由前 $i-1$ 个数据元素组成已排序的数据序列，将 k 与数据序列中各数据元素依次进行比较后，插入到数据序列的适当位置，使得插入后的数据序列仍是排序的。

设有一个待排序的数据序列，其关键字序列为{5，3，2，4，7，1}，直接插入排序算法描述如下：

（1）将第 1 个元素 k=5 插入，作为初始的已排序的子序列{5}。

（2）欲插入元素值 k=3，首先在已排序的子序列{5}中进行顺序查找，找到 k 值应插入的位置 i=1。

（3）将从 i 位置到序列末尾的子序列{5}依次向后移动，移动次序是从序列末尾开始到 i 位置，空出第 i 位置。

（4）将 k 值插入第 i 位置。插入 3 之后的子序列为{3，5}。

（5）重复步骤 2、3 和 4，依次将其他数据元素插入到已排序的子序列中。

2．顺序存储结构线性表的直接插入排序

设待排序的数据序列保存在数组中，为简单起见，假定每个数据元素只含关键字。顺序表的直接插入排序算法如图 3.1 所示。

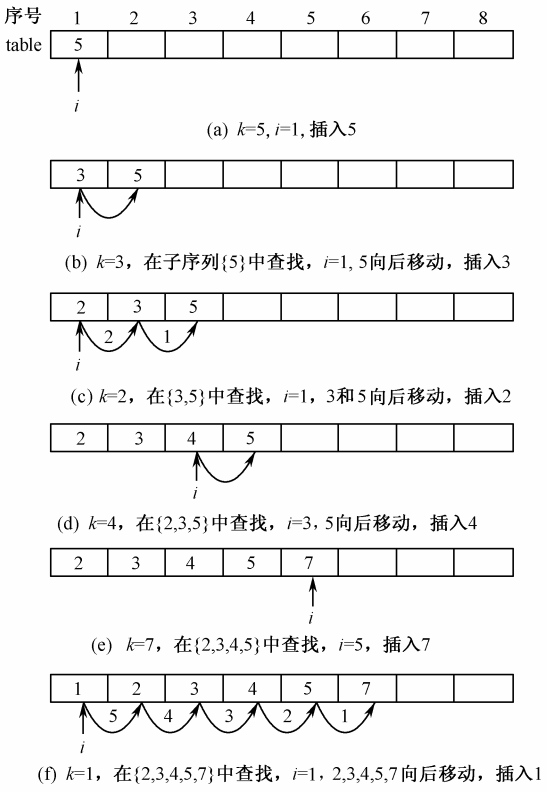


图 3.1　**直接插入排序算法描述**

（1**）在已排序的顺序表中，插入一个数据元素。**

```
public void insertdata(int k)                    //插入 k 值
{
    int i=search(k);                             //顺序查找
    for(int j=this.length();j>=i;j--)
        this.set(j+1,this.get(j));               //将第 j 个元素值后移一位
    this.set(i,k);                               //设置第 i 个元素值为 k
}
```

其中，search(k)**方法在顺序表中依次查找** *k* **值，若找到（称为查找成功），返回** *k* **值所在的位置；若没找到（称为查找不成功），返回** *k* **值应插入的位置。对于直接插入排序算法，每次插入** *k* **值之前，首先要在已排序的顺序表中查找** *k* **值以确定插入位置，这时大多数都是不成功的查找。所以，对于不成功的查找，**search(*k*)**方法必须返回** *k* **应插入的位置。**search(*k*)**算法实现详见例** 3.1**。**

（2**）顺序表的直接插入排序。**

```
public InsertSort1(int table[])    //将 table 数组元素依次插入已排序顺序表
{
    for(int i=0;i<table.length;i++)
        insertdata(table[i]);
}
```

在第 2 **章中，我们已设计了一个线性表的顺序存储结构** LinearList1 **类，其中成员变量** tab

数组是一个私有成员，我们可以通过以下的一些公共方法访问数据元素：

- 构造方法——创建一个具有 n 个存储单元的线性表，其中没有数值。
- length()——返回数组实际的元素个数。
- get(i)——返回数组中第 i 个元素值。
- set(i,k)——设置第 i 个元素值为 k。
- insert(int k)——添加 k 值。
- swap(int i,int j)——交换第 i 和 j 个位置上的元素。
- output()——输出 table 数组中的元素值。

注意：其中的 i 和 j 表示第 i 和 j 个元素，而不是下标。线性表第 i 个元素是 table[i-1]。由于 table 是私有成员，我们通过 LinearList1 类的方法访问线性表中的元素，而不必关心每个元素的下标是多少。

【例 3.1】 顺序表的直接插入排序的算法实现与测试。

本例定义 InsertSort1 类继承 LinearList1 类，将已排序的数据元素存放在顺序表中，并使用超类声明的若干方法。

search(k)方法在顺序表中依次查找 k 值，insertdata()方法将 k 值插入到已排序的顺序表中，构造方法调用 insertdata()方法，将 table 数组元素依次插入到已排序顺序表中，实现直接插入排序。程序如下：

```
import ds_java.LinearList1;                          //顺序存储结构的线性表类
public class InsertSort1 extends LinearList1         //直接插入排序
{
   public InsertSort1(int table[])     //将 table 数组元素依次插入已排序顺序表
   {
      super(table.length);
      for(int i=0;i<table.length;i++)
         insertdata(table[i]);
   }
   public void insertdata(int k)                     //插入 k 值
   {
      int i=search(k);                               //顺序查找
      System.out.print("k="+k+"  i="+i+"  ");
      for(int j=this.length();j>=i;j--)
          this.set(j+1,this.get(j));                 //将第 j 个元素值后移一位
      this.set(i,k);                                 //设置第 i 个元素值为 k
      this.output();
   }
   public int search(int k)                          //顺序查找 k 值
   {                                                 //返回 k 值应在的位置
      int i=1;
      while(i<=this.length() && k>this.get(i)) //短路计算
         i++;
      return i;
```

```
    }
    public static void main(String args[])
    {
        int table[]={5,3,2,4,7,1,8,6};
        new InsertSort1(table);
    }
}
```

程序运行结果如下：

```
k=5  i=1  table:  5 0 0 0 0 0 0 0
k=3  i=1  table:  3 5 0 0 0 0 0 0
k=2  i=1  table:  2 3 5 0 0 0 0 0
k=4  i=3  table:  2 3 4 5 0 0 0 0
k=7  i=5  table:  2 3 4 5 7 0 0 0
k=1  i=1  table:  1 2 3 4 5 7 0 0
k=8  i=7  table:  1 2 3 4 5 7 8 0
k=6  i=6  table:  1 2 3 4 5 6 7 8
```

3．**算法分析**

由第 2 **章可知，在具有** n **个数据元素的顺序存储结构的线性表中，考虑等概率情况，插入一个数据元素平均需要移动线性表全部数据元素的一半（**$n/2$**）。与此类似，在线性表中查找一个数据元素的平均比较次数为**$(n+1)/2$**。所以，直接插入排序的平均比较次数为**

$$\overline{C}=\sum_{i=1}^{n}\frac{i+1}{2}=\frac{1}{4}n^2+\frac{3}{4}n+1\approx\frac{n^2}{4}$$

平均移动次数为

$$\overline{M}=\sum_{i=1}^{n}\frac{i}{2}=\frac{n(n+1)}{4}\approx\frac{n^2}{4}$$

由此可知，直接插入排序的时间复杂度为 $O(n^2)$**。**

直接插入排序算法是稳定的。

4．**链表的直接插入排序**

在例 2.3 **中，我们已用单向链表实现了直接插入排序算法。下面用双向链表实现直接插入排序，首先讨论在双向链表中的各种不同位置处插入结点。为简单起见，假定每个结点只含关键字。**

设 rear **指向双向链表的最后一个结点，在已排序的双向链表中插入** k **值，**k **值的插入位置取决于** k **值的大小，插入后的链表仍然是排序的。相应算法（**insert()**方法）描述如下：**

- **当** head==null **时，链表为空，创建值为** k **的结点** q **作为链表的第一个结点，由** head **指向。这是空链表插入结点情况。此时，**rear **与** head **指向同一个结点，即** rear==head
- **当** head!=null **时，比较** k **与** head.data **的值，如果** k **值小，将结点** q **插入到第一个结点之前，由** head **指向。这是表头插入情况。**
- **当** k>head.data **时，结点** q **需要插入到链表中间（尾）。设** p **引用从** head **开始依次查找，当** k>p.data **时，沿链表方向前进。循环结束时，结点** q **应该插入在结点** p **之前。这是**

表中插入情况。

- 如果循环结束时，p 引用为空，则表示已到链表尾，q 应该插入在 rear 结点之后。这是表尾插入情况。

算法描述如图 3.2 所示。

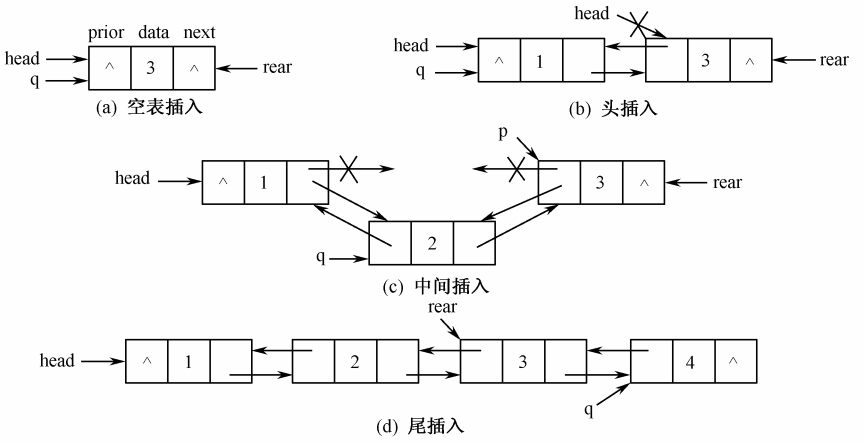


图 3.2　双向链表的插入排序算法描述

双向链表的直接插入排序算法的实现与测试见例 3.2。

【例 3.2】　双向链表的直接插入排序。

Twolink2 类继承在 2.2.3 节中定义的双向链表 Twolink1 类，继承超类的 head 成员变量和 output()等方法，结点是 TwolinkNode 类。程序如下：

```
import ds_java.TwolinkNode;                    //双向链表的结点类
import ds_java.Twolink1;                       //双向链表类
public class Twolink2 extends Twolink1         //双向链表插入排序
{
   protected TwolinkNode rear;                 //引用链表最后一个结点
   Twolink2()                                  //建立空链表
   {
      super();                                 //head==null
      rear=null;
   }
   Twolink2(int n)                             //n 个随机值插入双向链表
   {
      int i=0,k;
      System.out.print("insert:   ");
      for(i=0;i<n;i++)
      {
         k=(int)(Math.random()*100);
         System.out.print(k+"  ");
         insert(k);
      }
```

```
    System.out.println();
}
public void insert(int k)                    //k 值插入已排序的双向链表
{
    TwolinkNode p=null,q=null;
    try
    {
        q=new TwolinkNode(k) ;               //创建 k 值结点 q
        if(head==null)
        {
            head=q ;                         //空表插入
            rear=head;
        }
        else
        {
            if(k<head.data)                  //头插入
            {
                q.next=head;
                head.prior=q;
                head=q;
            }
            else
            {
                p=head;                      //查找 q 应插入的位置
                while(p!=null && k>p.data)
                    p=p.next;                //向后走
                if(p!=null)                  //表中插入
                {
                    q.prior=p.prior;         //将 q 插入在结点 p 之前
                    q.next=p;
                    (p.prior).next=q;
                    p.prior=q;
                }
                else                         //表尾插入
                {
                    rear.next=q;             //将 q 插在 rear 结点之后
                    q.prior=rear;
                    rear=q;
                }
            }
        }
    }
```

```
            catch(Exception e)                    //捕获异常
            {
                System.out.println(e);            //显示异常信息
                output();                         //输出双向链表
            }
        }
        public static void main(String args[])
        {
            (new Twolink2(8)).output();
        }
    }
```

程序运行结果如下：

```
insert:   77  76  73  87  89  22  64  1
Twolink2:  1 -> 22 -> 64 -> 73 -> 76 -> 77 -> 87 -> 89
```

用链表实现插入排序，算法的平均比较次数与数组实现的一样，约为 $n^2/4$，减少的是元素移动次数。所以，算法的时间复杂度为 $O(n^2)$。

本例演示异常处理，将创建结点的语句 new TwolinkNode(k)写在 try 后，程序运行时，由系统动态申请一个结点的存储单元，如果没有可用内存，new 产生一个异常 OutOfMemoryException，系统捕获后通过 Exception 对象 e 传递给 catch 语句处理。

3.2.2 希尔排序

在直接插入排序中，每次比较仅在相邻的数据元素间进行，一趟排序后数据元素最多移动一个位置。设待排序序列的数据元素个数为 n，假定序列中第 1 个数据元素的数值最大，排序后的最终位置应该是序列的最后一个元素，则将它从头移动到尾需要 $n-1$ 步。如果该值一次就能跳到最后，排序的速度就快得多。因此，每次进行比较的是相隔较远的数据元素，使得数据元素移动时能够跨过多个位置，然后逐渐减少被比较数据元素间的距离（又称增量或跳步），直至距离为 1 时，各数据元素都已按序排好。这就是由 D.L. SHELL 提出的渐减增量排序（diminishing increment sort），又称希尔排序。

希尔排序算法的思想是：排序之初，允许数据元素做较大的移动，而当数据元素接近目的地时，再做较小的移动。这样做可使排序过程加快。希尔排序的算法思想非常简单，但如何选择增量以产生最好的排序效果，至今仍没定论。

以增量选择法实现希尔排序的算法（按递增次序排序）如下：

```
    public void shellsort()                       //对顺序表对象进行希尔排序
    {                                             //数据元素已存储在 table 数组中
        int i,j,jump,n=this.length();
        jump=n/2;
        while(jump>0)                             //控制增量
        {
            for(i=jump;i<=n;i++)                  //一轮比较、交换
            {
                j=i-jump;
```

```
            while(j>0)
                if(this.get(j)>this.get(j+jump))
                {                                   //与相隔 jump 的元素比较、交换
                    swap(j,j+jump);                 //反序时，交换
                    j=j-jump;                       //继续与前面的元素比较
                }
                else
                  j=-1;                             //退出循环
            }
            System.out.print("jump="+jump+"  ");
            this.output();
            jump=jump/2;                            //增量减半
        }
    }
```

希尔排序算法共有三重循环：

- **最外层循环** while **语句控制增量** jump**，初值为数组长度** *n* **的一半，以后逐步减半**（jump=jump / 2)**，直至为** 1**。**
- **中间循环** for **语句进行一轮相隔** jump **的元素进行比较、交换。**
- **最内层循环** while **语句，将第** *j* **个元素值** this.get(j)**与相隔** jump **的第** j - jump **个元素值** this.get(j+jump)**进行比较，如果两者是反序的，则执行** swap(j, j+jump)**交换两个值。重复往前与相隔** jump **的元素再比较、交换，** *j*=*j*–jump**，当** this.get(j)>this.get(j+jump)**时表示该元素** this.get(j)**已在这趟排序后的位置，不需交换，则退出最内层循环。**

例如，对于一个待排序的数据序列，其关键字序列为{8，3，2，7，9，1，6，5}**，数据序列保存在数组中，数据元素个数** *n*=8**，增量** jump=*n*/2**。希尔排序算法描述如图** 3.3 **所示。希尔排序算法描述如下：**

（1**）**jump=4**，** *j* **从第** 1 **个位置元素开始，将相隔** jump **远的第** *j* **个元素与第** *j*+jump **个元素进行比较。如果反序（数值小的数在前，数值大的数在后），则交换，将较小的数交换到前面，依次重复进行完一趟排序，如图** 3.3（a）**所示。具体比较情况如下：**

- **第** 1 **次比较，第** 1 **个元素（**8**）小于第** 5 **个元素（**9**），递增次序，不交换。**
- **第** 2 **次比较，第** 2 **个元素（**3**）大于第** 6 **个元素（**1**），反序，交换两元素。**
- **第** 3 **次比较，第** 3 **个元素（**2**）小于第** 7 **个元素（**6**），不交换。**
- **第** 4 **次比较，第** 4 **个元素（**7**）大于第** 8 **个元素（**5**），交换。**

（2**）**jump=2**，比较第** *j* **个元素与第** *j*+jump **个元素，如果反序，则交换，依次重复。如图** 3.3（b）～（g）**所示。具体比较情况如下：**

- **第** 1 **次比较，第** 1 **个元素（**8**）大于第** 3 **个元素（**2**），交换。**
- **第** 2 **次比较，第** 2 **个元素（**1**）小于第** 4 **个元素（**5**），不交换。**
- **第** 3 **次比较，第** 3 **个元素（**8**）小于第** 5 **个元素（**9**），不交换。**
- **第** 4 **次比较，第** 4 **个元素（**5**）大于第** 6 **个元素（**3**），交换。**
- **第** 5 **次比较，第** 5 **个元素（**9**）大于第** 7 **个元素（**6**），交换。再比较，第** 3 **个元素（**8**）大于第** 5 **个元素（**6**），交换。**
- **第** 6 **次比较，第** 6 **个元素（**5**）小于第** 8 **个元素（**7**），不交换。**

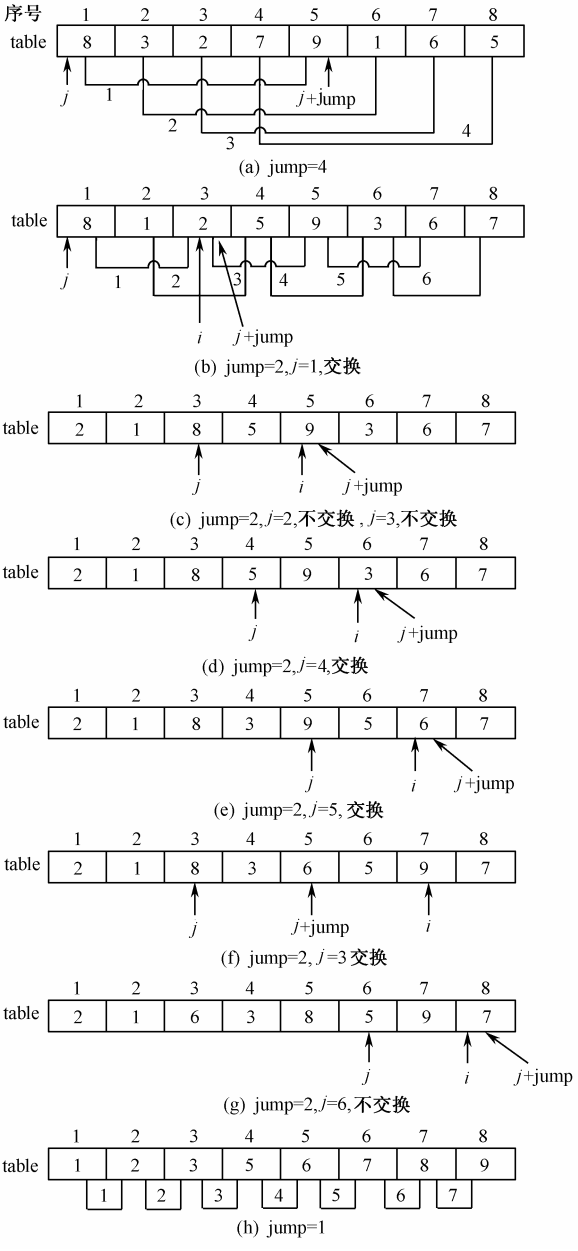


图 3.3　希尔排序算法描述

（3）jump=1，排序原理同上，略。比较、交换规则如图 3.3（h）所示。

算法测试的程序运行结果如下：

```
jump=4  table:  8 1 2 5 9 3 6 7
jump=2  table:  2 1 6 3 8 5 9 7
jump=1  table:  1 2 3 5 6 7 8 9
```

希尔排序算法的时间复杂度分析比较复杂，实际所需的时间取决于每次排序时增量的个数和增量的取值。研究证明，若增量的取值比较合理，希尔排序算法的时间复杂度约为 $O(n(\log_2 n)^2)$**。**

希尔排序算法的空间复杂度为 $O(1)$。由于希尔排序算法是按增量分组进行的排序，所以希尔排序算法是一种不稳定的排序算法。

3.3 交换排序

基于交换的排序算法有两种：冒泡排序（bubble sort）和快速排序（quick sort）。

3.3.1 冒泡排序

冒泡排序是最简单的交换排序算法。

1．冒泡排序算法

冒泡排序的基本思想是：将相邻的两个数据元素按关键字进行比较，如果反序，则交换。对于一个待排序的数据序列，经一趟排序后，最大值数据元素移到最后位置，其他值较大的数据元素也向最终位置移动，此过程称为一趟起泡。

对于有 n 个数据元素的数据序列，共需 $n-1$ 趟排序，第 i 趟对从 $1\sim n-i$ 个数据元素进行比较、交换，因此该排序算法可用二重循环实现。算法如下：

```
public void bubblesort()
{
   int i,j,n=this.length();
   for(i=1;i<n;i++)                          //n-1 趟排序
   {
      for(j=1;j<=n-i;j++)                    //一轮比较、交换
         if(this.get(j)>this.get(j+1))
            this.swap(j,j+1);                //反序时，交换
      System.out.print("第"+i+"趟  ");
      this.output();
   }
}
```

bubblesort()方法用两重循环实现冒泡排序算法。外层 for 循环控制 $n-1$ 趟排序，内层 for 循环对相邻两元素进行比较，反序时，交换它们。

设有一个待排序的数据序列，其关键字序列为{5，3，7，4，6，8，2，1}，算法测试的程序运行结果如下：

```
第1趟  table:  3 5 4 6 7 2 1 8
第2趟  table:  3 4 5 6 2 1 7 8
第3趟  table:  3 4 5 2 1 6 7 8
第4趟  table:  3 4 2 1 5 6 7 8
第5趟  table:  3 2 1 4 5 6 7 8
第6趟  table:  2 1 3 4 5 6 7 8
第7趟  table:  1 2 3 4 5 6 7 8
```

冒泡排序算法描述如图 3.4 所示。

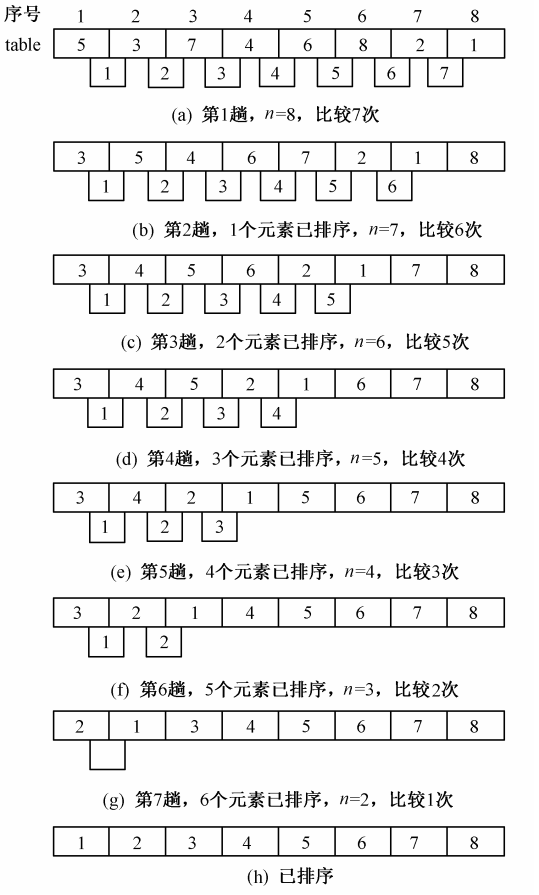


图 3.4　**冒泡排序算法描述**

2．算法分析

bubblesort()**方法中，以如下形式的二重循环实现冒泡排序：**

```
for(i=1;i<n;i++)                    //n-1 趟排序
    for(j=1;j<=n-i;j++)             //一轮比较、交换
```

由此可知，算法的时间复杂度为 $O(n^2)$**。**

冒泡排序中，因交换两个数据元素需要一个辅助空间，故空间复杂度为 $O(1)$**。**

冒泡排序是稳定的。

3．改进的冒泡排序

实际上，n **个数据元素的排序并不一定都需要进行** $n-1$ **趟。例如，对于以下已排序的数据序列**

{1，2，3，4，5，6，7，8}

如果一趟排序中未发生数据元素交换，则表示已完成排序工作，不必再进行比较。因此对前述的冒泡排序算法做以下两点改进：

- **增加一个标记** exchange**，表示本趟起泡结果是否发生了交换。**
- **增加一个比较起始位置的标记** index**，以减少重复的比较次数。**

改进的冒泡排序算法如下：

```
public void bubblesort()
```

```
    {
        int i=1,j,n=this.length(),index=1;
        boolean exchange=true;                    //是否交换的标记
        while(i<n && exchange)                    //最多 n-1 趟排序
        {
            System.out.print("第"+i+"趟  index="+index+" n-i="+(n-i)+"  ");
            exchange=false;                       //假定元素未交换
            j=index;                              //起始比较位置
            while(j<=n-i)                         //一轮比较、交换
            {
                System.out.print(this.get(j)+"<"+this.get(j+1)+"? ");
                if(this.get(j)>this.get(j+1))
                {
                    this.swap(j,j+1);             //反序时，交换
                    exchange=true;                //更改交换标记
                    System.out.print("swap ");
                }
                j++;
                if(!exchange && this.get(j)<this.get(j+1))
                    index++;
            }
            this.output();
            i++;
        }
    }
```

设有一个待排序的数据序列，其关键字序列为{1，2，3，4，8，7，6，5}，算法测试的程序运行结果如下：

```
    第 1 趟  index=1 n-i=7  1<2? 2<3? 3<4? 4<8? 8<7? swap 8<6? swap 8<5? swap table:
1 2 3 4 7 6 5 8
    第 2 趟  index=4 n-i=6  4<7? 7<6? swap 7<5? swap table:  1 2 3 4 6 5 7 8
    第 3 趟  index=4 n-i=5  4<6? 6<5? swap table:  1 2 3 4 5 6 7 8
    第 4 趟  index=4 n-i=4  4<5? table:  1 2 3 4 5 6 7 8
```

改进的冒泡排序算法描述如图 3.5 所示。

可见，该算法最好情况是已排序的数据序列，只需一趟排序即可，比较次数为 n，没有移动，因此算法的时间复杂度是 $O(n)$。最坏情况是已排序但反序的数据序列，需要 $n-1$ 趟排序，比较次数和移动次数都是 $O(n^2)$，因此算法的时间复杂度是 $O(n^2)$。

3.3.2 快速排序

快速排序是目前平均性能较好的一种排序算法。

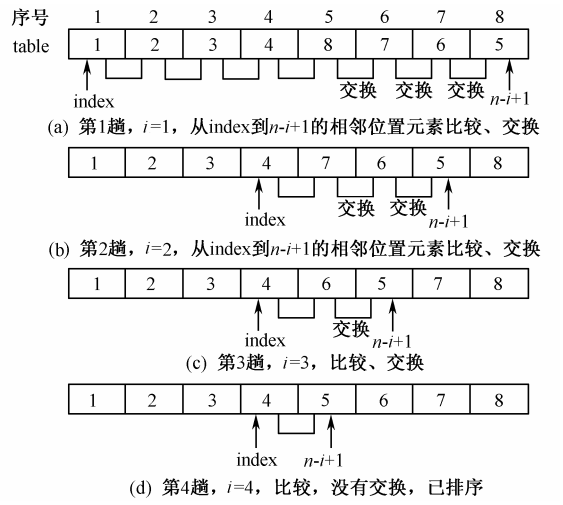


图 3.5　改进的冒泡排序算法描述

1．快速排序算法

快速排序的基本思想是：在待排序的数据序列中任意选择一个值作为基准值，由序列的两端交替地向中间进行比较、交换，使得所有比基准值小的元素都处于序列的左端，比基准值大的元素都处于序列的右端，这样序列就被划分成两个子序列。再对两个子序列分别进行同样的操作，直到子序列的长度为 1 时，则已排好序。每趟排序完，作为基准值的数据元素需要找到它在排好序的序列中的最终位置。

快速排序的算法如下：

```
public void quicksort(int left,int right)    //实现一趟快速排序
{
   int i,j,n,vot;
   if(left<right)                             //左边界小于右边界，序列有效
   {
       i=left;
       j=right;
       vot=this.get(i);                       //第一个值作为基准值
       while(i!=j)                            //进行一轮比较
       {
          while(vot<this.get(j) && i<j)
             j--;                             //从右向左寻找较小值
          if(i<j)
          {
             this.set(i,this.get(j));         //较小值元素向左移动
             i++;
          }
          while(this.get(i)<vot && i<j)
             i++;                             //从左向右寻找较大值
          if (i<j)
```

```
            {
                this.set(j,this.get(i));        //较大值元素向右移动
                j--;
            }
        }
        this.set(i,vot);                        //基准值的最终位置
        i++;
        j--;
        quicksort(left,j);                      //左边子序列再排序
        quicksort(i,right);                     //右边子序列再排序
    }
}
```

quicksort()方法是实现快速排序的递归算法。设元素个数为 *n*，left 和 right 分别表示待排序的子序列的左、右边界，*i* 和 *j* 分别表示子序列的第一个元素和最后一个元素，基准值 vot 选取子序列的第 1 个值，即 *i* 位置的元素，table[i]表示数组第 *i* 个位置的元素。this.get(i)获得 *i* 位置的元素，this.set(i,vot)设置 *i* 位置的元素值为 vot。

算法中，while(i!=j)循环进行一轮比较，*i*、*j* 分别从序列的最左、右端开始，如果左端的数据元素值较大（this.get(i)>vot），则该数据元素向右移动；如果右端的数据元素值较小（this.get(j)<vot），则该数据元素向左移动；直至 *i*==*j*，则已将原序列分成两个子序列，比基准值小的数据元素在左端，比基准值大的数据元素在右端，则 *i* 位置就是基准值 vot 的最终位置。

一趟排序后，原序列分为两个子序列，边界分别为(left，j)和(i，right)。对两个子序列再分别调用 quicksort()方法进行快速排序。

设有一个待排序的数据序列，其关键字序列为{5，6，2，4，8，3，1，7}，元素个数 *n*=8，快速排序算法的描述如下：

（1）第 1 趟：1 个子序列，left=1，right=8，*i*=1，*j*=8，基准值 vot=table[i]。排序基本思想是：将小于基准值的数据元素向左移动，将大于基准值的数据元素向右移动，即将 *i* 和 *j* 位置上的数据元素依次与 vot 比较：

- 如果 table[j]<vot，将 table[j]值赋给 table[i]，*i*++。
- 如果 table[i]>vot，将 table[i]值赋给 table[j]，*j*--。
- 依次重复直到 *i*==*j* 时，table[i]左端数据元素都比 vot 小，右端数据元素都比 vot 大，则 table[i]应该是 vot 的最终位置。

（2）第 1 趟排序后，以 vot 为界，将原序列分为两个子序列，左端数据元素值比 vot 小，边界是(left，j)；右端数据元素值比 vot 大，边界是(i，right)。

（3）第 2 趟：对上述两个子序列再分别进行快速排序，直至 left==right，即子序列长度为 1。

快速排序算法描述如图 3.6 所示。

算法测试的程序运行结果如下：

```
left=1  right=8  vot=5  table:  1 3 2 4 5 8 6 7
left=1  right=4  vot=1  table:  1 3 2 4 5 8 6 7
left=2  right=4  vot=3  table:  1 2 3 4 5 8 6 7
left=6  right=8  vot=8  table:  1 2 3 4 5 7 6 8
```

```
left=6  right=7  vot=7  table:  1 2 3 4 5 6 7 8
```

2．**算法分析**

快速排序的平均比较次数为 $O(n\log_2 n)$，**时间复杂度为** $O(n\log_2 n)$。

由于快速排序是递归算法，系统调用时需要设立一个栈（stack**）来存放参数，最大递归调用层次数为** $\lfloor \log_2(n+1) \rfloor$。**因此，算法的空间复杂度为** $O(n\log_2 n)$。

快速排序所用的执行时间与序列的初始排列及基准值的选取有关。最好情况是，每趟排序后，将序列分成两个长度相同的子序列。最坏情况是，当序列已排序时，如

$$\{1，2，3，4，5，6，7，8\}$$

选取序列的第一个值作为基准值，分成的两个子序列长度为 1 **与** $n-1$：

$$1，\{2，3，4，5，6，7，8\}$$

这样必须经过 $n-1$ **趟才能完成排序。因此，总的比较次数为：**

$$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}n(n-1)\approx\frac{n^2}{2}$$

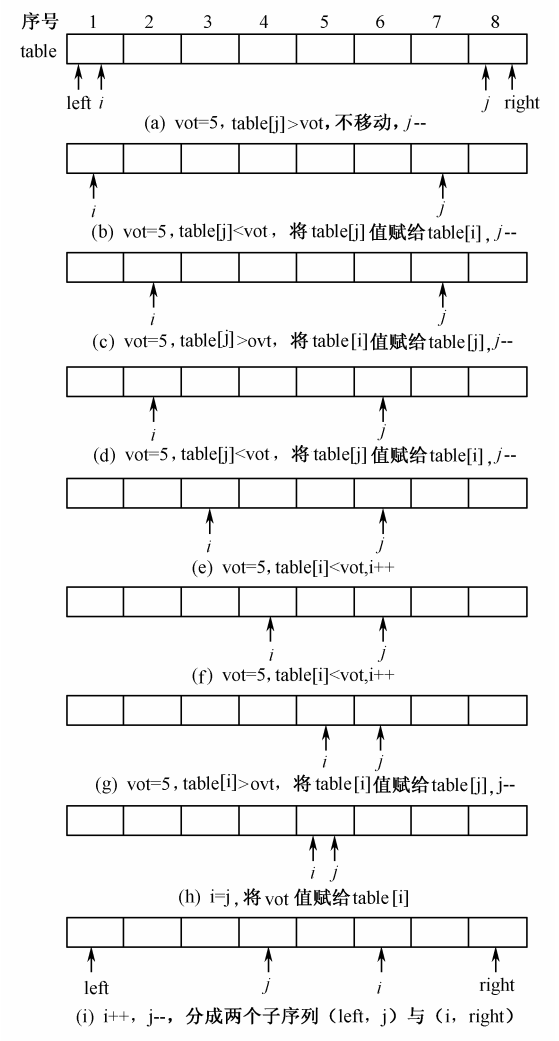


图 3.6　**快速排序算法描述**

这种情况下，所花费的比较次数与运行时间最多，时间复杂度为 $O(n^2)$。其排序速度已退化，比直接插入排序算法还慢。

基准值的选择还有其他多种方法，如可以选取序列的中间值等。但由于序列的初始排列是随机的，不管如何选择基准值，总会存在最坏情况。

快速排序算法是不稳定的。对于关键字相同的元素，排序后的次序可能会颠倒。

总之，对于 n 较大的平均情况而言，快速排序是“快速”的。但是当 n 很小时，或基准值选取不合适时，也会使快速排序的时间复杂度退化为 $O(n^2)$。

3.4 选择排序

选择排序算法有两种：直接选择排序（select sort）和堆排序（heap sort）。本节介绍直接选择排序，堆排序将在第 6 章中介绍。

1．顺序表的直接选择排序算法

直接选择排序的基本思想是：设待排序的数据序列有 n 个元素，第 1 趟排序，比较 n 个元素，选择关键字最小的元素，将其交换到序列的第 1 个位置上；第 2 趟排序，在余下的 $n-1$ 个元素中，再选取关键字最小的元素，交换到序列的第 2 个位置上……经过 $n-1$ 趟排序，n 个元素的数据序列则按递增次序排序完成。

直接选择排序算法既适用于顺序表，也适用于链表。

顺序表的直接选择排序算法如下：

```
public void selectsort()
{
   int i,j,min,k,n=this.length();
   for(i=1;i<n;i++)                          //n-1 趟排序
   {
       min=i;                                //记下本趟最小值位置
       for(j=i+1;j<=n;j++)                   //一轮比较、交换
          if(get(j)<get(min))
             min=j;                          //记下新的最小值位置
       if(min!=i)
          swap(i,min);                       //本趟最小值交换到左边
       System.out.print("min="+min+"  ");
       this.output();
   }
}
```

selectsort()方法用二重循环实现数组的直接选择排序。外层 for 循环控制 $n-1$ 趟排序，内层 for 循环进行每趟的比较，以 min 记录最小值的位置，然后与第 i 个位置的元素交换。

设有一个待排序的数据序列，其关键字序列为{5，3，7，4，6，8，2，1}，元素个数 n=8，i 表示起始比较的元素位置，min 表示关键字值最小元素位置。直接选择排序算法描述如图 3 所示。

算法测试的程序运行结果如下：

```
min=8  table:  1 3 7 4 6 8 2 5
min=7  table:  1 2 7 4 6 8 3 5
min=7  table:  1 2 3 4 6 8 7 5
min=4  table:  1 2 3 4 6 8 7 5
min=8  table:  1 2 3 4 5 8 7 6
min=8  table:  1 2 3 4 5 6 7 8
min=7  table:  1 2 3 4 5 6 7 8
```

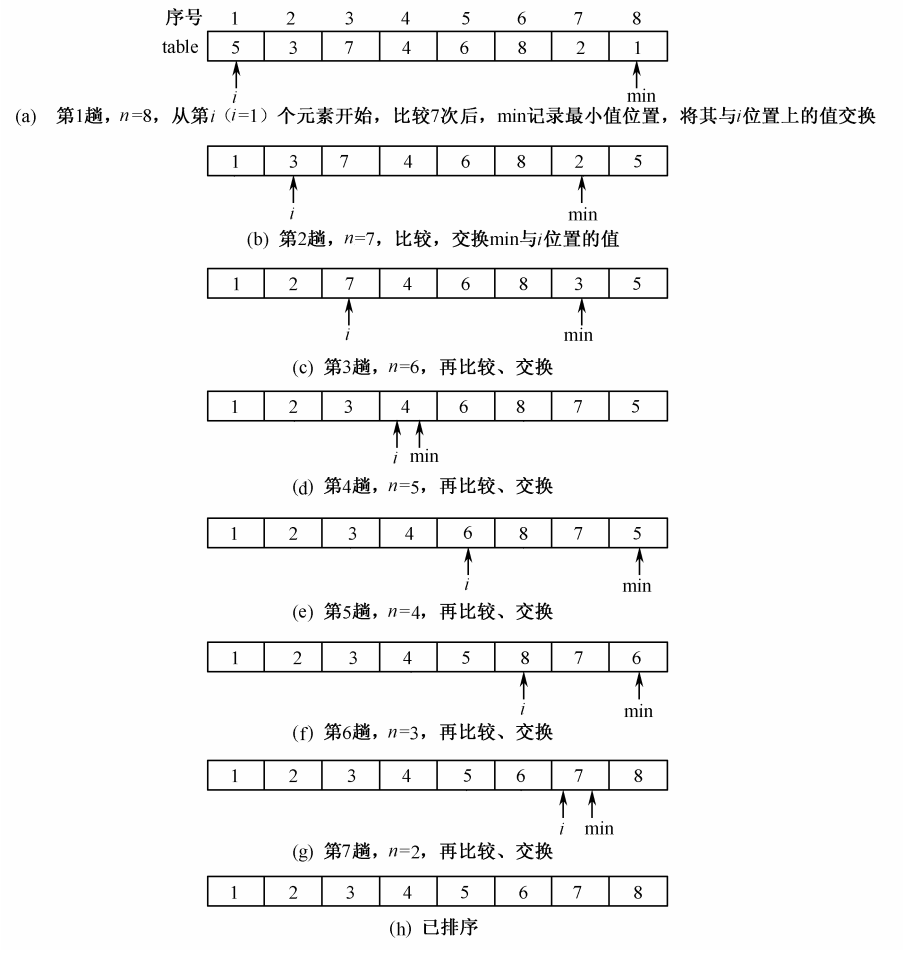


图 3.7　直接选择排序算法描述

2．算法分析

直接选择排序的比较次数与数据序列的初始排列无关。设待排序的数据序列有 n 个数据元素，第 i 趟排序，则选择最小值数据元素所需的比较次数是 $n-i-1$ 次。所以，算法总的比较次数为

$$C=\sum_{i=0}^{n-2}(n-i-1)=\frac{1}{2}n(n-1)\approx\frac{n^2}{2}$$

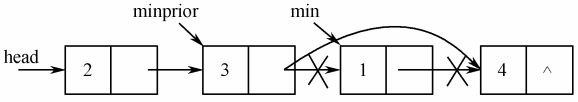
数据元素的移动次数与序列的初始排列有关。当数据序列的初始状态按数据元素的关键字递增排列时，数据元素的移动次数最少，$M=0$。最坏情况是，每一趟排序都要进行交换，总的

数据元素移动次数为 $M=3\times(n-1)$。所以，直接选择排序的时间复杂度为 $O(n^2)$。

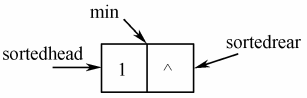
直接选择排序算法是不稳定的。

3．单向链表的直接选择排序

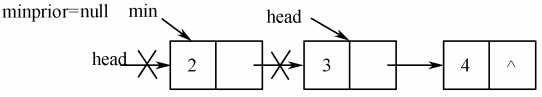
下面以单向链表为例，讨论单向链表的直接选择排序算法。算法描述如图 3.8 所示。



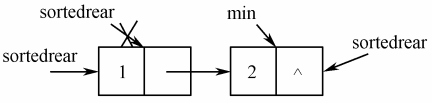
(a) 选择值最小的结点（由min指向），将min结点删除



(b) 将min结点添加到已排序的单向链表之后



(c) 选择值最小的结点（由min指向），将 min结点删除



(d) 将min结点添加到已排序的单向链表之后

图 3.8　单向链表的直接选择排序算法描述

对单向链表进行直接选择排序的算法描述如下：

- 设已建立一条由 head 指向的单向链表。sortedhead、sortedrear 指向已排序链表的首尾结点，sortedhead 和 sortedrear 的初值为空。
- 每一趟，在单向链表中寻找到最小值结点 min，删除结点 min，而后将结点 min 加入到由 sortedhead 指向的已排序链表中。由于单向链表无法直接访问结点的前驱，所以在查找过程中，同时设结点 q 为结点 p 的前驱，minprior 为结点 min 的前驱。
- 当 head==null 时，sortedhead 指向的是已排序成功的单向链表。

所以，对于单向链表，只要调整结点的链接关系，不需移动数据，就可实现直接选择排序。

单向链表的直接选择排序算法如下：

```
public void selectsort()
{
   OnelinkNode sortedhead=null,sortedrear=null;
   OnelinkNode p=null,q=null,min=null,minprior=null;
   do
   {
      min=head;
      p=head.next;
      q=head;
      while(p!=null)
```

```
            {
                if(p.data<min.data)            //比较,min 记住最小值位置
                {
                    minprior=q;                //minprior 是 min 的前驱结点
                    min=p;
                }
                q=p;                           //q 是 p 的前驱结点
                p=p.next;
            }
            System.out.print("min="+min.data+"    ");
            if(min==head)                      //从 head 链表中删除 min 结点
                head=head.next;
            else
                minprior.next=min.next;
            min.next=null;
            if(sortedhead==null)               //在已排序链表中插入 min 结点
                sortedhead=min;                //头插入
            else
                sortedrear.next=min;           //尾插入
            sortedrear=min;
            output();
        }while(head!=null);
        this.head=sortedhead;                  //返回已排序链表的引用
    }
```

算法测试的程序运行结果如下：

```
n=8    Selectlink1:  53 -> 62 -> 36 -> 28 -> 86 -> 17 -> 11 -> 14
min=11    Selectlink1:  53 -> 62 -> 36 -> 28 -> 86 -> 17 -> 14
min=14    Selectlink1:  53 -> 62 -> 36 -> 28 -> 86 -> 17
min=17    Selectlink1:  53 -> 62 -> 36 -> 28 -> 86
min=28    Selectlink1:  53 -> 62 -> 36 -> 86
min=36    Selectlink1:  53 -> 62 -> 86
min=53    Selectlink1:  62 -> 86
min=62    Selectlink1:  86
min=86    Selectlink1:
Selectlink1:  11 -> 14 -> 17 -> 28 -> 36 -> 53 -> 62 -> 86
```

思考题：以双向链表实现直接选择排序。

3.5 归并排序

1．归并排序算法描述

所谓归并，就是将两个已排序的数据序列合并，形成一个已排序的数据序列，又称两路归并

设有一个待排序的数据序列，其关键字序列为{8，3，2，7，9，1，6，5，0，4}，数据序列有 n 个元素，两路归并排序算法的描述如下：

- 将初始序列看成是由 n 个长度为 1 的已排序子序列组成。
- 反复将两个子序列合并成一个已排序的序列。
- 重复上步，直到合并成一个序列时，排序完成。

归并排序算法描述如图 3.9 所示。

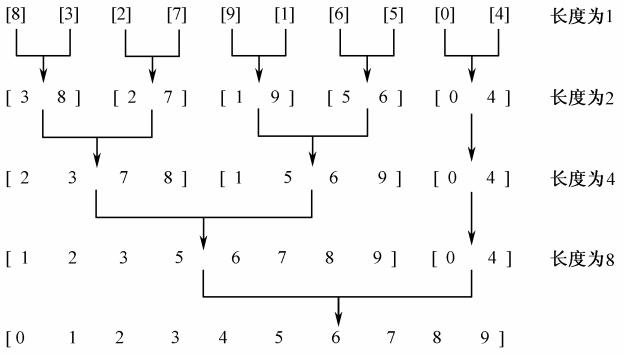


图 3.9　归并排序算法描述

2．顺序表的归并排序算法实现

以顺序表实现归并排序的算法如下：

```
MergeSort1 temp;                              //辅助线性表
public void mergesort()                       //归并排序算法
{
   MergeSort1 x=this;
   int len=1;                                 //已排序的序列长度,初始值为 1
   temp=new MergeSort1(x.length());           //temp 所需空间与 x 一样
   do
   {
      x.mergepass(temp,len);                  //将 x 中元素归并到 temp 中
      temp.output();
      len=len*2;
      temp.mergepass(x,len);                  //将 temp 中元素归并到 x 中
      x.output();
      len=len*2;
   }while(len<x.length());
}
public void mergepass(MergeSort1 y,int len)   //一趟归并排序
{                                             //将 this 中元素归并到 y 中
   MergeSort1 x=this;
   int i=1,j;
   System.out.print("len="+len+"  ");
   while(i<=x.length()-2*len+1)
```

```
        {
            x.onemerge(y,i,i+len,len);
            i=i+2*len;
        }
        System.out.print(" i="+i+" i+len="+(i+len)+"  ");
        if(i+len<x.length())
            x.onemerge(y,i,i+len,len);          //再一次归并
        else
            for(j=i;j<x.length();j++)           //将 x 中余下的值复制到 y 中
                y.set(j,x.get(j));
    }
    public void onemerge(MergeSort1 y,int m,int r,int n)   //一次归并
    {                                           //将 this 中的两个子序列归并到 y
        MergeSort1 x=this;
        int i=m,j=r,k=m;
        while(i<r && j<r+n && j<=x.length())
        {
            System.out.print(x.get(i)+"<"+x.get(j)+"? ");
            if(x.get(i)<x.get(j))
            {                                   //较小的值送到 y 中
                y.set(k,x.get(i));
                k++;
                i++;
            }
            else
            {
                y.set(k,x.get(j));
                k++;
                j++;
            }
        }
        while(i<r)                              //将一子序列余下的值复制到 y 中
        {
            y.set(k,x.get(i));
            k++;
            i++;
        }
        while(j<r+n && j<=x.length())           //将另一子序列余下的值复制到 y 中
        {
            y.set(k,x.get(j));
            k++;
            j++;
```

```
            }
        }
    }
```

其中，mergesort()**方法实现两路归并排序算法。设待排序的数据序列存放在线性表** *x* **中，**temp **表示排序中使用的辅助线性表，其所需空间的大小与** *x* **相同。**len **表示已排序的序列的长度，初始值为** 1**。一次循环中通过调用** mergepass()**方法完成两趟归并排序，从** *x* **到** temp**，再从** temp **到** *x*，**使得排序后的数据序列仍保存在线性表** *x* **中。每次归并后，已排序的序列长度** len **的值成倍增加。**

mergepass()**方法完成一趟归并排序。调用** onemerge()**方法，依次将** *x* **中相邻两个已排序的子序列归并到线性表** *y* **中，子序列的长度为** len**。如果相邻的子序列已归并完，** *x* **中仍有数据，则将其复制到** *y* **中。**

onemerge()**方法完成一次归并，将** *x* **中相邻的两个子序列（起始位置分别为** *m* **和** *r*，**长度为** *n*）

$$x_m, \cdots, x_{r-1} \text{ 和 } x_r, \cdots, x_{r+n}$$

归并到 *y* **中**

$$y_m, \cdots, y_{r+n}$$

顺序表的归并排序算法描述如图 3.10 **所示。**

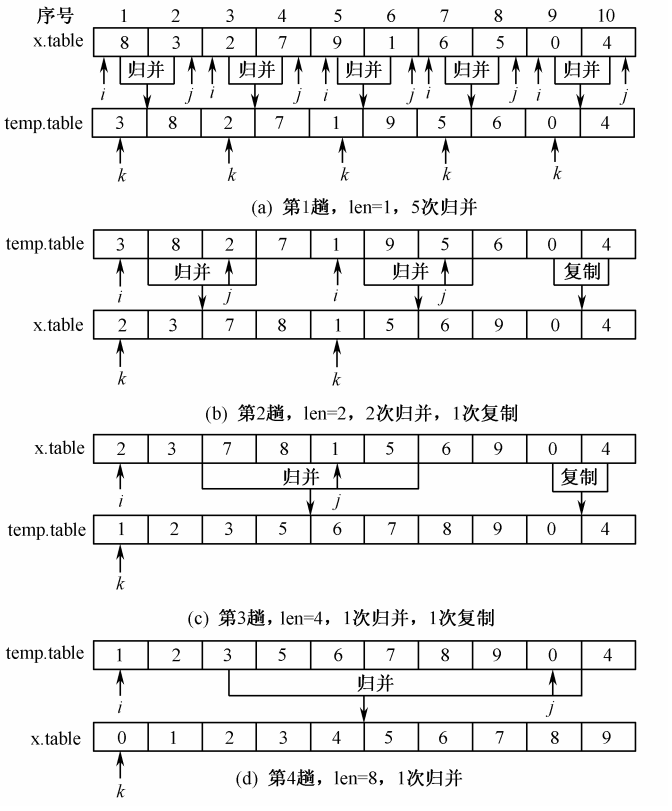


图 3.10　**顺序表的归并排序算法描述**

算法测试的程序运行结果如下：

```
n=10   table:  8 3 2 7 9 1 6 5 0 4
len=1  8<3? 2<7? 9<1? 6<5? 0<4?  i=11 i+len=12  table:  3 8 2 7 1 9 5 6 0 4
len=2  3<2? 3<7? 8<7? 1<5? 9<5? 9<6?  i=9 i+len=11  table:  2 3 7 8 1 5 6 9 0 4
len=4  2<1? 2<5? 3<5? 7<5? 7<6? 7<9? 8<9?  i=9 i+len=13 table: 1 2 3 5 6 7 8 9 0 4
len=8   i=1 i+len=9  1<0? 1<4? 2<4? 3<4? 5<4? table:  0 1 2 3 4 5 6 7 8 9
```

在归并排序算法中：

- onemerge()**方法完成一次归并，需要进行** $O(\text{len})$**次比较。**
- mergepass()**方法完成一趟归并排序，需要调用** onemerge()**方法** $\lceil n/(2\times\text{len})\rceil \approx O(n/\text{len})$ **次。**
- mergesort()**方法实现归并排序算法，需要调用** mergepass()**方法** $\lceil \log_2 n \rceil$**次。**

所以，归并排序算法的时间复杂度为 $O(n\log_2 n)$ **。**
此外，归并排序算法需要容量为 $O(n)$**的辅助空间，与存储数据序列的空间量相等。**
归并排序算法是稳定的。

3．**链表的归并排序算法**

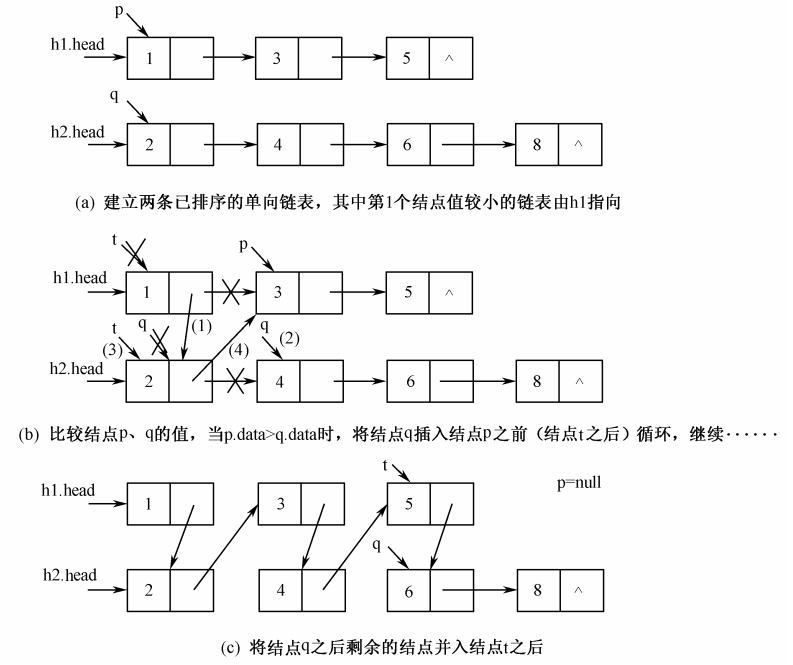
对于两条已排序的单向链表，使用一趟归并排序算法可以将两者合并成一条已排序的链表。下面算法归并两条已排序的单向链表，多次调用该算法即可实现链表的归并排序。

```
public void mergelink(Mergelink1 h2)     //归并两条已排序的单向链表
{
   Mergelink1 h1=this;
   OnelinkNode p,q,t=null;
   if(h2==null || h2.head==null)         //具有短路计算功能
      return;
   if(h1.head==null)
   {
      h1.head=h2.head;
      return;
   }
   p=h1.head;                            //p 在 h1 链表中移动
   q=h2.head;                            //q 在 h2 链表中移动
   if(p.data>q.data)                     //比较两个链表第 1 个结点的值
   {
      q=h1.head;
      p=h2.head;
      h1.head=h2.head;                   //h1 仍指向第 1 个结点值较小的链表
   }
   t=p;
   while(p!=null && q!=null)
   {
      if(p.data<=q.data)
      {
         t=p;
```

```
            p=p.next;
        }
        else
        {
            t.next=q;
            q=q.next;
            t=t.next;
            t.next=p;
        }
    }
    if(q!=null)                          //另一链表中多余结点并入 h1 链表之尾
      t.next=q;
}
```

其中，设 h1 **和** h2 **分别指向两条已排序的单向链表，**mergelink()**方法首先选择两条链表中第** 1 **个结点值较小的链表，由** h1 **指向；然后将另一链表（**h2**）中各结点依次分别插入到** h1 **链表中；最后** h2 **链表中所有多余的结点，全部链入** h1 **链表的尾部。**

算法描述如图 3.11 **所示。**



(a) 建立两条已排序的单向链表，其中第1个结点值较小的链表由h1指向

(b) 比较结点p、q的值，当p.data>q.data时，将结点q插入结点p之前（结点t之后）循环，继续……

(c) 将结点q之后剩余的结点并入结点t之后

图 3.11　**归并两条已排序的单向链表算法描述**

算法测试的程序运行结果如下：

```
Mergelink1:  1 -> 3 -> 5 -> 7 -> 9
Mergelink1:  0 -> 2 -> 4 -> 8 -> 10 -> 12
```

两链表归并后:

```
Mergelink1:  0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 7 -> 8 -> 9 -> 10 -> 12
```

习　题　3

3.1　设有一个待排序的数据序列，其关键字序列如下：

{03，17，12，61，08，70，97，75，53，26，54，61}

（1）画出直接插入排序的过程。

（2）画出希尔排序的过程。

（3）画出冒泡排序的过程。

（4）画出快速排序的过程。

（5）画出选择排序的过程。

（6）画出归并排序的过程。

3.2　在一个待排序的数据序列中，求第 k 个最小的数据元素，不要将所有数据元素全部排序。

3.3　对于数据序列{5，7，3，4，6，2，1，8}，请问选取什么样的排序算法只经过一趟排序就能够排序？

3.4　在一个含有正负数的待排序序列中，欲将正负数分类，使负数全部排在序列的前半段。如何实现？

3.5　写一个非递归的快速排序算法。

3.6　一种改进的归并排序算法。对给定的表，元素用数组存储：A[0]，A[1]，…，A[n-1]，若其中 A[i].key≤A[i+1].key≤…≤A[j].key，0≤i≤j≤n-1，则把 A[i]，…，A[j]作为一个初始已排序表，用这种方法，重归并排序算法。

3.7　设被排序的表为 R，奇偶交换排序算法如下：第一趟对所有奇数 i，将 R[i]和 R[i+1]进行比较；第二趟对所有偶数 i，将 R[i]和 R[i+1]进行比较，若 R[i]＞R[i+1]，则将两者交换；第三趟对奇数 i；第四趟对奇数 i，……依次类推，直至整个表排好序为止。

（1）试问这种排序方法的结束条件是什么？

（2）编写奇偶交换排序的算法。

（3）分析当初始序列为无序、正序或逆序时，排序算法所需的关键字比较次数。

3.8　比较本章各排序算法的时间代价和空间代价，并指出各种方法的特点。

3.9　总结本章的各种排序方法。哪些是稳定的？哪些是不稳定的？对不稳定的方法举例说明。

实　习　3

1．实习目的：

学习多种排序算法，灵活运行排序算法解实际问题。

2．题意：

（1）九宫排序问题

在一个方框盘中放上 8 个方块，分别写上 1，2，…，8，另有一个位置空着，如图 3.12（a）所示。做游戏时，只能将与空格相邻的方块移入空格内。要求对任意给定的初始状态，判断是否能够移成如图 3.12（的目标状态。若能，则给出移动步伐，否则输出不能移动信息。

（2）多项式相加问题

用单向链表可以表示含有一个未知数的多项式。每个结点包含 3 个成员：指数，系数和指向后继结点链。例如，多项式 $3x^4-6x^2+5x-10$ 可表示为图 3.13 所示的链表。

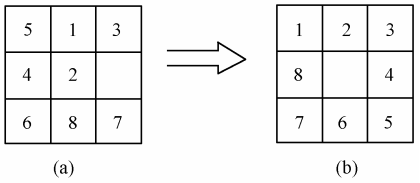


图 3.12　九宫排序图

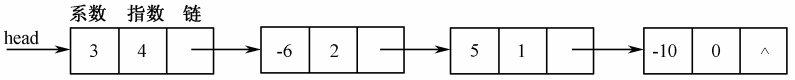


图 3.13　多项式链表

对于两个给定多项式，建立两条多项式链表，求解两个多项式相加问题。

（3）实现静态链表结构的插入排序

欲对顺序表{e，g，c，b，d，h，a，f}**进行插入排序，要求不能移动数据元素。设顺序表的每个数据元素由两个域组成：**data **表示数据元素值，**next **表示下一个数据元素的位置。这种存储方式称为静态链表，如图** 3.14 **所示。当插入排序成功后，**first **表示第** 1 **个数据元素的下标（**first=6**）。**

	data	next
0	e	7
1	g	5
2	c	4
3	b	2
4	d	0
5	h	-1
6	a	3
7	f	1

图 3.14　静态链表

第 4 章　栈与队列

栈和队列是两种特殊的线性表。与线性表一样，栈和队列的数据元素之间具有顺序的逻辑关系，都可以采用顺序存储结构和链式存储结构；与线性表不同的是，线性表的插入和删除操作不受限制，可以在任意位置进行，而栈和队列的插入和删除操作受到限制：栈的插入和删除操作只允许在线性表的一端进行，队列的插入和删除操作则分别在线性表的两端进行。

栈的特点是后进先出，队列的特点是先进先出，两者在实际问题中有着广泛的应用。

递归不是一种数据结构，而是一种有效的算法设计方法。存在自调用的算法称为递归算法。递归算法是解决许多复杂问题的重要方法。

本章介绍栈和队列的定义、抽象数据类型、存储结构和应用举例，以及递归的定义、算法设计和数据结构举例。

建议本章授课 6 学时，实验 4 学时。

4.1　栈

4.1.1　栈的定义

栈（stack）是一种特殊的线性表，其插入和删除操作只允许在线性表的一端进行。允许操作的一端称为栈顶（top），不允许操作的一端称为栈底（bottom）。栈顶的当前位置是动态的，标识栈顶当前位置的变量称为栈顶指针。栈中插入数据元素的过程称为入栈（push），删除数据元素的过程称为出栈（pop）。当栈中没有数据元素时称之为空栈。

由于只允许在栈顶进行插入和删除操作，也就是说，每次删除的数据元素总是最后插入的数据元素，因此栈又称为“后进先出表”（Last In First Out）。就像一堆书，每次只允许一本一本地往上堆，一本一本往下取，不允许从中间插进或抽出。放书和取书的操作就像入栈和出栈。栈结构如图 4.1 所示。

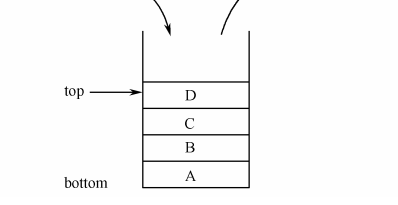


图 4.1　栈结构

图 4.1 中，数据元素的入栈次序为 A→B→C→D，出栈次序为 D→C→B→A。由于栈结构的特点，对于一种顺序入栈的数据元素序列，通过控制入栈和出栈时机，可以得到多种出栈排列。

思考题：图 4.1 中还有哪些出栈排列？哪些排列是不可能得到的出栈排列？

4.1.2 栈的抽象数据类型

1．栈的数据元素

栈的数据元素和线性表的数据元素完全相同，即栈的数据元素是 n（$n \geq 0$）个相同类型的数据元素 a_1，a_2，…，a_n 组成的有限序列，记为：

$$\text{Stack}=\{a_1，a_2，\cdots，a_n\}$$

其中，n 表示栈中数据元素的个数，称为栈的长度。若 $n=0$，则称为空栈。

2．栈的基本操作

栈的基本操作有：

- 栈的初始化，设置栈状态为空。
- 判断栈的状态是否为空。
- 判断栈的状态是否已满。
- 入栈——将数据元素插入栈中作为新的栈顶数据元素的过程。在入栈之前必须判断栈的状态是否已满，如果不满，则接收新数据元素入栈，否则产生上溢错误（overflow）。
- 出栈——取出当前栈顶数据元素，下一个数据元素成为新的栈顶数据元素的过程。在出栈之前，必须判断栈的状态是否为空。如果栈的状态为空，产生下溢错误（underflow）。
- 获得栈顶数据元素，此时该数据元素未出栈。

例如，对于数据序列

$$\{1，2，3，4\}$$

依次进行

{入栈，入栈，出栈，入栈，入栈，出栈}

的操作，栈状态的变化如图 4.2 所示。

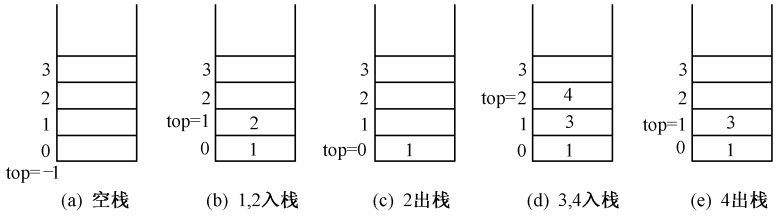


图 4.2 栈状态的变化

用 Java 语言将栈的基本操作声明在接口 StackInterface 中。声明如下：

```
package  ds_java;
public interface StackInterface      //栈的接口
{
   public boolean isEmpty();         //判断栈的状态是否为空
   public boolean isFull();          //判断栈的状态是否已满
   public boolean push(Object k);    //k 对象入栈
   public Object  pop();             //出栈
   public Object  get();             //获得栈顶数据元素，未出栈
}
```

此处将栈作为抽象数据类型，数据元素的类型设置为 Object，所以入栈的数据元素是 Object 对象，可以是 Integer 或 String 等对象，而出栈的数据元素需用 Object 对象保存，再转化为 Integer 或 String 等对象。

4.1.3 栈的存储结构及实现

栈作为一种特殊的线性表，同样可以采用顺序存储结构和链式存储结构。顺序存储结构的栈称为顺序栈。链式存储结构的栈称为链式栈。

1．栈的顺序存储结构及操作实现

栈的顺序存储结构就是定义一组连续的存储空间存放栈的数据元素。以下声明的 Stack1 类实现栈接口 StackInterface。

```
package ds_java;
import ds_java.StackInterface;
public class Stack1 implements StackInterface   //栈的顺序存储结构
{
   private Object table[];
   private final int empty=-1;                  //声明 empty 为整数常量
   private int top=empty;                       //top 为栈顶数据元素下标
}
```

Stack1 类用数组 table 存储栈的数据元素，变量 top 指示当前栈顶数据元素的下标。Stack1 类的一个对象就是一个栈。

顺序栈的操作实现如下，这些算法写在 Stack1 类中，作为 Stack1 类的方法。

（1）栈的初始化。构造方法申请 table 数组的存储空间来存放栈的数据元素，设置栈初始状态为空。

```
   public Stack1(int n)              //带参数时，构造具有 n 个存储单元的空栈
   {
      table=new Object[n];
      top=empty;                     //设置栈初始状态为空
   }
   public Stack1()                   //不带参数时，构造具有 10 个存储单元的空栈
   {
      this(10);
   }
```

（2）判断栈状态是否为空。

```
   public boolean  isEmpty()         //判断栈的状态是否为空
   {
      return top==empty;
   }
```

（3）判断栈是否已满。当 top>=table.length 时，栈已满。

```
   public boolean isFull()           //判断栈是否已满
   {
```

```
    return top>=table.length;
}
```

（4）入栈。当栈不满时，栈顶数据元素下标 top 加 1，将 *k* 放入 top 位置，作为新的栈顶数据元素。此时入栈的数据元素是 Object 对象，可以是 Integer 或 String 等对象。

```
public boolean push(Object k)          //k 对象入栈
{
    if(!isFull())                      //栈不满
    {
        top++;
        table[top]=k;
        return true;
    }
    else
    {
        System.out.println("栈已满，"+k+"值无法入栈！");
        return false;                  //栈溢出
    }
}
```

（5）出栈。当栈不空时，取走 top 位置处的数据元素，top 减 1，下一位置上的数据元素作为新的栈顶数据元素。此时出栈的数据元素是 Object 对象，可以转化为 Integer 或 String 等对象。

```
public Object pop()                    //出栈
{
    Object k=null;
    if(!isEmpty())                     //栈不空
    {
        k=table[top];                  //取得栈顶数据元素的值
        table[top]=null;
        top--;
    }
    return k;                          //栈空时返回 null
}
```

（6）获得栈顶数据元素的值。当栈非空时，获得 top 位置处的数据元素，此时该数据元素未出栈，top 值不变。

```
public Object  get()                   //获得栈顶数据元素，未出栈
{
    if(!isEmpty())
        return table[top];
    else
        return null;
}
```

【例 4.1】 使用顺序栈的基本操作。

源程序 Stack1_ex.java **引用** ds_java.Stack1 **类，使用顺序栈的基本操作。程序如下：**

```
import ds_java.Stack1;
class Stack1_ex
{
   public static void main(String args[])
   {
      int i=0,n=4;
      Stack1 s1=new Stack1(20);
      System.out.print("Push: ");
      while(i<args.length)
      {
         s1.push(args[i]);
         System.out.print(args[i]+"  ");
         i++;
      }
      s1.output();                          //输出栈中各元素值
      for(i=1;i<=n;i+=2)
      {
         s1.push(new Integer(i));           //值为 i 的 Integer 对象入栈
         s1.push(new Integer(i+1));
         System.out.print("Push: "+i+" "+(i+1)+"\t");
         s1.output();
         System.out.print("Pop:  "+s1.pop()+"\t\t");
         s1.output();
      }
      System.out.print("Pop : ");
      Object obj;
      while(!s1.isEmpty())                  //全部出栈
      {
         obj=s1.pop();
         System.out.print(obj+"  ");
      }
      System.out.println();
   }
}
```

编译并运行 Stack1_ex.java**，运行时从命令行输入参数：**

```
D:\myjava\javac  Stack1_ex.java
D:\myjava\java   Stack1_ex  a  b  c
```

程序运行结果如下：

```
Push: a  b  c    ds_java.Stack1:  a b c
```

```
Push: 1 2        ds_java.Stack1:  a b c 1 2
Pop:  2          ds_java.Stack1:  a b c 1
Push: 3 4        ds_java.Stack1:  a b c 1 3 4
Pop:  4          ds_java.Stack1:  a b c 1 3
Pop : 3  1  c  b  a
```

2．栈的链式存储结构及操作实现

栈的链式存储结构可用单向链表实现。设 top 指向链表的第 1 个结点（即栈顶结点），入栈和出栈操作都是针对 top 所指向的结点进行的。栈的链式存储结构如图 4.3 所示。

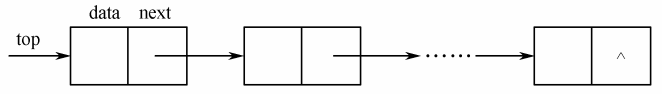


图 4.3　栈的链式存储结构

以下声明的 Stack2 类实现栈的链式存储结构：

```
package ds_java;
import ds_java.OnelinkNode;                    //引用单向链表的结点类
import ds_java.Onelink1;
public class Stack2 extends Onelink1           //栈的链式存储结构
{
   private OnelinkNode top;
}
```

Stack2 类的一个对象就是一个栈。其中，成员变量 top 指向栈顶数据元素结点，结点类型为单向链表的结点类 OnelinkNode，结点数据域的类型为 int。

链式栈的操作实现如下，这些算法写在 Stack2 类中，作为 Stack2 类的方法。

（1）栈的初始化。构造方法创建一条单向链表用做栈，设置栈初始状态为空。

```
public Stack2()                    //构造空栈
{
   top=null;
}
```

（2）判断栈状态是否为空。当 top==null 时，栈为空。

```
public boolean isEmpty()           //判断栈状态是否为空
{
   return  top=null;
}
```

（3）判断栈状态是否已满。采用动态分配方式为每个结点分配内存空间，当有一个数据元素需要入栈时，向系统申请一个结点的存储空间，程序中认为系统所提供的可用空间是足够大的，因此不必判断栈是否已满。如果空间已用完，系统无法分配新的存储单元，则产生运行时异常。

（4）入栈。在 top 指向的栈顶结点之前插入一个结点来存放数据元素值 k，并使 top 指向新的栈顶结点。

```
public boolean push(int k)          //k 值入栈
{
    OnelinkNode q=new OnelinkNode(k);
    q.next=top;                     //q 结点作为新的栈顶结点
    top=q;
    return true;
}
```

（5）出栈。当栈不为空时，取走 top 指向的栈顶结点的值，并删除该结点，使 top 指向新的栈顶结点。

```
public int pop()                    //出栈
{
    int k=-1;
    if(!isEmpty())                  //栈不空
    {
        k=top.data;                 //取得栈顶结点数据元素值
        top=top.next;               //删除栈顶结点
    }
    return k;                       //栈空时返回-1
}
```

（6）获得栈顶数据元素值。

```
public int get()                    //获得栈顶数据元素值，未出栈
{
    if(!isEmpty())
        return top.data;
    else
        return -1;
}
```

链式栈的基本操作示意如图 4.4 所示。

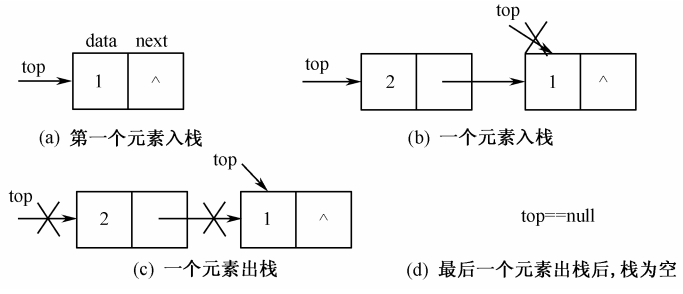


图 4.4　链式栈的基本操作

由以上分析可知，不管是顺序存储结构还是链式存储结构，都实现了栈的基本操作。当需要设立一个栈时，可以创建一个 Stack1 或 Stack2 的对象，通过栈对象调用入栈或出栈方法进行相应操作。对于程序员来说，只要知道栈中设计的基本方法就可以使用栈，而不必关注栈的存储结构及其实现细节。

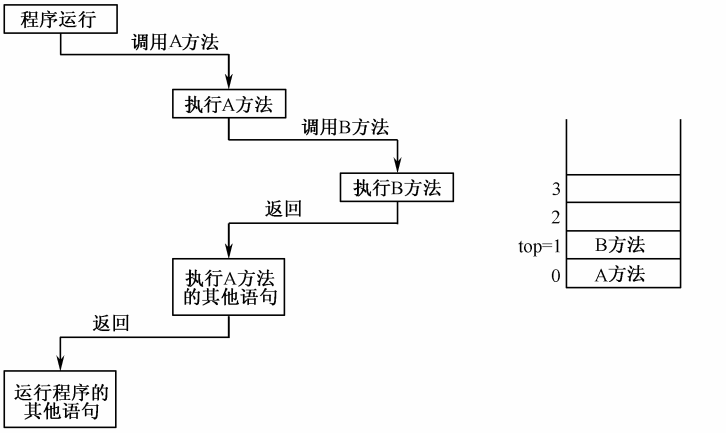
4.1.4 栈的应用举例

栈是一种特殊的线性表，具有后进先出的特性。当求解具有后进先出特性的问题时，栈为算法设计的有力工具。

1．栈是嵌套调用机制的实现基础

程序运行时，系统允许在一个方法中调用另一个方法，称为方法的嵌套调用。例如，执行方法 A 时，又调用方法 B，此时 A 和 B 两方法均未执行完，仍然占用系统资源，当方法 B 执行完后，系统能够返回调用 B 的方法 A 中。根据嵌套调用规则，每个方法在执行完后应返回到调用它的方法中。所以方法调用的次序与返回的次序正好相反。

方法嵌套调用时的系统栈如图 4.5 所示。



(a) 方法的嵌套调用与返回　　(b) 执行方法B时的系统栈

图 4.5　方法嵌套调用时的系统栈

栈是系统实现嵌套调用机制的基础。当方法被调用时，系统将该方法的有关信息（地址参数、局部变量等状态）保存在栈中（入栈），称为保护现场；方法调用完返回时，取出保存在栈中的信息（出栈），称为恢复现场，程序继续运行。

由此可知，系统实现嵌套调用或递归调用时，设立一个栈结构是必不可少的。

2．实现深度遍历算法时使用栈

实现深度遍历算法，如遍历二叉树，以深度优先算法遍历图，都需要使用栈。详细算法将在以后章节中介绍。

3．利用栈以非递归方式实现递归算法

如果程序设计语言中出现的表达式的定义本身是递归的，则计算表达式的值也有相应的递归算法。

那么，系统是如何将表达式编译成能够正确求值的指令，以及如何求值的呢？我们通过以下两个例题，理解栈的作用以及系统编译、运行程序的过程。

【例 4.2】 判断表达式中括号是否匹配。

在 Java 语言的表达式中，只能出现圆括号用以改变运算次序，而且圆括号应该是左右匹配的。对于一个给定的表达式，可使用栈来判断其中括号是否匹配。

本例声明 Exp_bracket **类，对于一个字符串** expstr**，方法** isValid()**判断其中的括号是否匹配**
例如，当字符串 expstr **保存表达式“** ((1+2)∗(3+4)) **”时，** isValid()**方法的算法描述如图** 4.6 **所示**

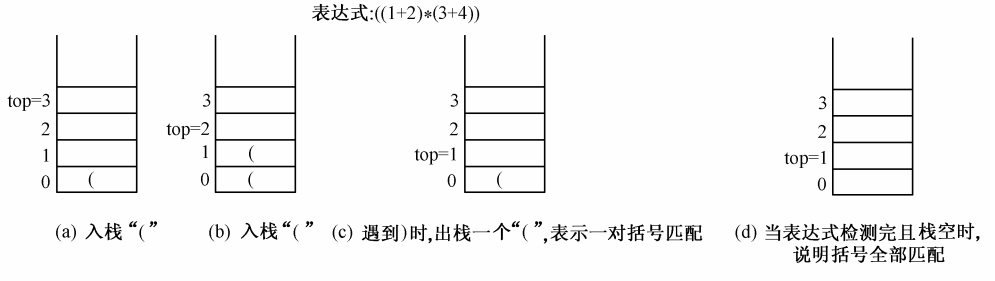


图 4.6　**判断表达式中括号是否匹配的算法描述**

isValid()**算法描述如下：**

（1**）设** ch **是待检测字符串** expstr **的当前字符：**

- **若** ch **是左括号，则** ch **入栈。**
- **若** ch **是右括号，则出栈一个值。若出栈值为左括号，表示括号匹配。若出栈值为空（**null**或不为左括号时，则表示缺少左括号，期望右括号。**

（2**）重复执行上一步。当** expstr **检测结束后，若栈为空，表示括号匹配，否则表示缺少左括号，期望右括号。**

程序中使用前面已声明的栈 Stack1**，数据元素的类型是** Object **类。入栈的数据元素是** String **对象，出栈数据元素也由** Object **对象保存。程序如下：**

```
import ds_java.Stack1;
public class Exp_bracket
{
    public static void main(String args[])
    {
        String expstr1="((1+2)*(3+4))=";
        if(args.length>0)                    //有参数时，
            expstr1=args[0];                 //获得表达式字符串
        System.out.println(expstr1+"  isValid  "+isValid(expstr1));
    }
    public static String isValid(String expstr)
    {
        Stack1 s1=new Stack1(30);            //创建空栈
        char ch;
        int i=0;
        boolean yes=true;
        String out;
        while(yes && i<expstr.length())
        {
            ch=expstr.charAt(i);
            i++;
```

```
            switch(ch)
            {
                case '(': s1.push(ch+"");          //遇见左括号时,入栈
                        break;
                case ')': out=(String)s1.pop();    //遇见右括号时,出栈
                          if(out==null || !out.equals("("))
                              yes=false;           //判断出栈的是否为左括号
            }
        }
        if(yes)
            if(s1.isEmpty())
                return "OK!";
            else
                return "期望)!";
        else
            return "期望(!";
    }
}
```

程序运行结果如下：

```
((1+2)*(3+4))=  isValid  OK!
```

思考题：本例为什么必须设置栈？如果用循环判断统计左右括号的个数是否相等，是否能解决问题？

【例 4.3**】 使用栈计算表达式的值。**

当我们在程序中写一个算术表达式，例如，

$$1+2*(3-4)+5 \qquad (4\text{-}1)$$

假定该表达式已通过编译程序的语法检查，程序运行时，系统将计算该表达式的值。那么系统是怎样对表达式求值的呢？

我们平常所写的表达式，将运算符写在两个操作数中间，称为中缀表达式。中缀表达式中运算符具有不同的优先级，而且圆括号还可以改变运算符的实际运算次序。这两点使得运算规律较复杂，求值过程不能从左到右顺序进行。

将运算符写在两个操作数之后的表达式称为后缀表达式。后缀表达式中没有括号，而且运算符没有优先级。后缀表达式的求值过程能够严格地从左到右顺序进行，符合运算器的求值规律。例如，式（4-1**）可以转化为如下的后缀表达式：**

$$1\ \ 2\ \ 3\ \ 4\ \ -\ \ *\ \ +\ \ 5\ \ + \qquad (4\text{-}2)$$

从左到右按顺序进行运算，遇到运算符时，则对它前面的两个操作数求值，如图 4.7 **所示**

为简化问题，本例对整型表达式求值，输入字符串形式的合法的中缀表达式，表达式由双目运算符＋、-、*和圆括号“(”、“)”组成。

表达式求值的算法分为两步进行：首先将中缀表达式转换为后缀表达式，再求后缀表达式的值。

1）**将中缀表达式转换为后缀表达式**

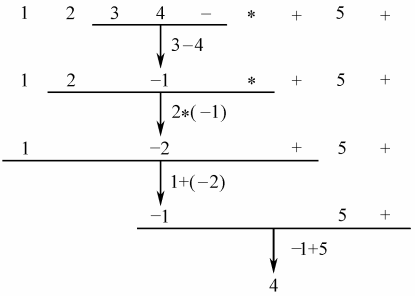


图 4.7　后缀表达式求值过程

对于字符串形式的合法的中缀表达式，“（”的运算优先级最高，“＊”次之，“＋”、“-最低，同级运算符从左到右按顺序运算。

在中缀表达式中，当前看到的运算符不能立即参与运算。例如式（4-1）中，第 1 个出现的运算符是“＋”，此时另一个操作数没有出现，而且后出现的“＊”运算符的优先级较高，应该先运算，所以不能进行“＋”运算，必须将“＋”运算符保存起来。式（4-1）中“＋”、“＊的出现次序与实际运算次序正好相反，因此将中缀表达式转换为后缀表达式时，运算符的次序可能改变，必须设立一个栈来存放运算符。转化过程的算法描述如下：

- 从左到右对中缀表达式进行扫描，每次处理一个字符。
- 若遇到左括号“（”，入栈。
- 若遇到数字，原样输出。
- 若遇到运算符，如果它的优先级比栈顶数据元素的优先级高，则直接入栈，否则栈顶数据元素出栈，直到新栈顶数据元素的优先级比它低，然后将它入栈。
- 若遇到右括号“）”，则运算符出栈，直到出栈的数据元素为左括号，表示左右括号相互抵消。
- 重复以上步骤，直至表达式结束。
- 若表达式已全部结束，将栈中数据元素全部出栈。

将中缀表达式（4-1）转换为后缀表达式（4-2）时，运算符栈状态的变化情况如图 4.8 所示。

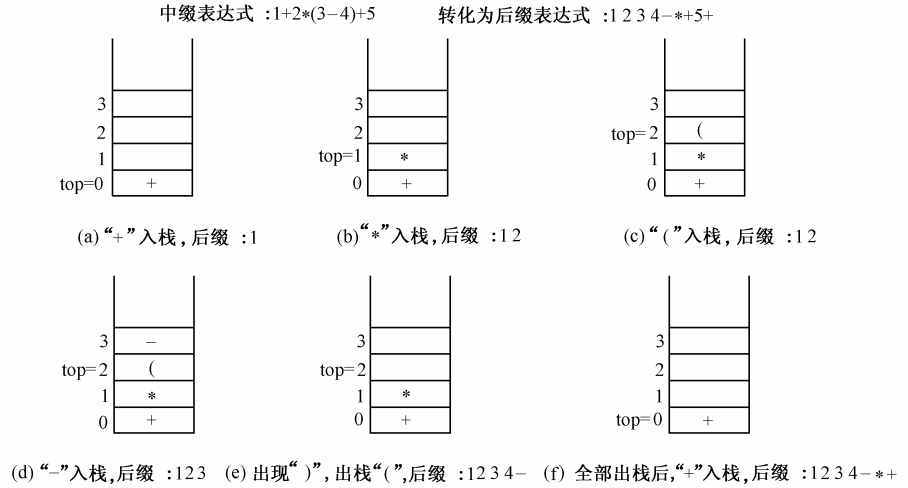


图 4.8　将中缀表达式变为后缀表达式时运算符栈状态的变化情况

2）后缀表达式求值

由于后缀表达式没有括号，且运算符没有优先级，因此求值过程中，当运算符出现时，只要取得前两个操作数就可以立即进行运算。而当两个操作数出现时，却不能立即求值，必须先保存等待运算符。所以，后缀表达式的求值过程中也必须设立一个栈，用于存放操作数。

后缀表达式求值算法描述如下：

- 从左到右对后缀表达式字符串进行扫描，每次处理一个字符。
- 若遇到数字，入栈。
- 若遇到运算符，出栈两个值进行运算，运算结果再入栈。
- 重复以上步骤，直至表达式结束，栈中最后一个数据元素就是所求表达式的结果。

在后缀表达式（4-2）的求值过程中，操作数栈状态的变化情况如图 4.9 所示。

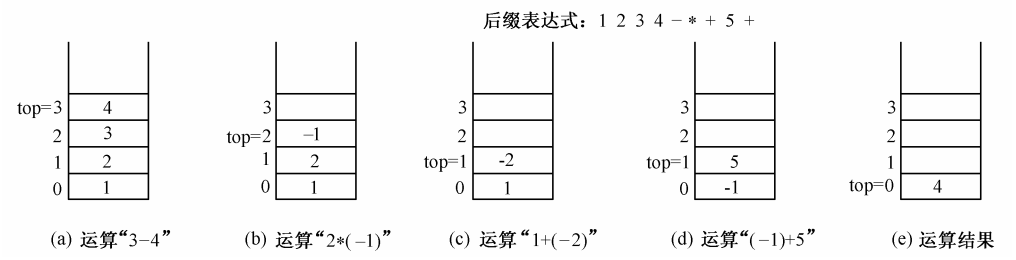


图 4.9　后缀表达式求值过程中数据栈状态的变化情况

例 4.3 声明的 Expression1 类对算术表达式求值。其中，构造方法以字符串 str 构造表达式对象，expstr 和 pstr 分别表示表达式的中缀和后缀形式。

postfix()方法将 expstr 中的中缀表达式转换为后缀表达式，保存在 pstr 中，转换时设立运算符栈 s1。s1 是 ds_java.Stack1 类的对象，数据元素类型为 Object。

value()方法对 pstr 中的后缀表达式求值，设立操作数栈 s2。s2 是 ds_java.Stack2 类的对象，数据元素类型为 int。程序如下：

```
import ds_java.Stack1;                  //顺序存储结构的栈，数据元素类型为 Object
import ds_java.Stack2;                  //链式存储结构的栈，数据元素类型为 int
public class Expression1
{
   String expstr="";                    //中缀表达式
   String pstr="";                      //后缀表达式
   public Expression1(String str)
   {
      expstr=str;
   }
   public static void main(String args[])
   {
      String str="((1+2)*(3-4)+5-6)*(7/7)";
      if(args.length>0)                 //有参数时，
         str=args[0];                   //获得表达式字符串
      Expression1 exp1=new Expression1(str);
      System.out.println("expstr:  "+exp1.expstr);
```

```
        System.out.println("postfix: "+exp1.postfix());
        System.out.println("value: "+exp1.value());
    }
    public String postfix()
    {
        Stack1 s1=new Stack1(30);                    //创建空栈
        char ch;
        String out;
        int i=0;
        while(i<expstr.length())
        {
            ch=expstr.charAt(i);
            switch(ch)
            {
                case '+':                            //遇到+、-时
                case '-': while(!s1.isEmpty() && !(s1.get()).equals("("))
                          {
                              out=(String)s1.pop();
                              pstr+=out;
                          }
                          s1.push(ch+"");
                          i++;
                          break;
                case '*':                            //遇到+、-时
                case '/': while(!s1.isEmpty() && ((s1.get()).equals("*") ||
                               (s1.get()).equals("/")))
                          {
                              out=(String)s1.pop();
                              pstr+=out;
                          }
                          s1.push(ch+"");
                          i++;
                          break;
                case '(': s1.push(ch+"");            //遇到左括号时，入栈
                          i++;
                          break;
                case ')': out=(String)s1.pop();      //遇到右括号时，出栈
                          while(!s1.isEmpty() && (out==null || !out.equals("(")
                          {
                              pstr+=out;
                              out=(String)s1.pop();
                          }
```

```
                    i++;
                    break;
            default:  while(ch>='0' && ch<='9')    //遇到数字时
                    {
                        pstr+=ch;
                        i++;
                        if(i<expstr.length())
                            ch=expstr.charAt(i);
                        else
                            ch='=';
                    }
                    pstr+=" ";
                    break;
        }
    }
    while (!s1.isEmpty())
    {
        out=(String)s1.pop();
        pstr=pstr+out;
    }
    return pstr;
}
public int value()
{
    Stack2 s2=new Stack2();                        //创建空栈
    char ch;
    int i=0,x,y,z=0;
    while(i<pstr.length())
    {
        ch=pstr.charAt(i);
        if(ch>='0' && ch<='9')
        {
            z=0;
            while(ch!=' ')
            {
                z=z*10+Integer.parseInt(ch+"");
                i++;
                ch=pstr.charAt(i);
            }
            i++;
            s2.push(z);
        }
```

```
            else
            {
                y=s2.pop();
                x=s2.pop();
                switch(ch)
                {
                    case '+': z=x+y; break;
                    case '-': z=x-y; break;
                    case '*': z=x*y; break;
                    case '/': z=x/y; break;
                }
                s2.push(z);
                i++;
            }
        }
        return  s2.pop();
    }
}
```

程序运行结果如下：

```
expstr:  ((1+2)*(3-4)+5-6)*(7/7)
postfix: 1 2 +3 4 -*5 +6 -7 7 /*
value: -4
```

4.2 队列

4.2.1 队列的定义

队列（queue**）是一种特殊的线性表，其插入和删除操作分别在线性表的两端进行。向队列中插入元素的过程称为入队（**enqueue**），删除元素的过程称为出队（**dequeue**）。允许入队的一端为队尾（**rear**），允许出队的一端为队头（**front**）。标识队头和队尾当前位置的变量称为队头指针和队尾指针。当队列中没有数据元素时称为空队列。**

队列的特点是当前出队的数据元素一定是队列中最先入队的数据元素（如图 4.10 **所示），因此队列又称为“先进先出表”（**First In First Out**），就像生活中排队购物一样。计算机系统运行时也需要许多队列，如多个进程在就绪时要排队等待运行等。**

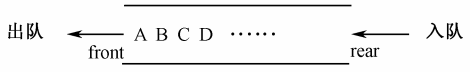


图 4.10　**队列**

设有数据元素 A，B，C，D **依次入队，则出队次序只有一种：**A→B→C→D**。**

栈和队列都是特殊的线性表，两者的区别在于：栈的插入和删除操作只允许在线性表的一端进行，而队列的插入和删除操作则分别在线性表的两端进行。

4.2.2 队列的抽象数据类型

1．队列的数据元素

队列的数据元素和线性表的数据元素完全相同，即队列的数据元素是 n（$n \geq 0$）个相同类型的数据元素 a_1，a_2，…，a_n 组成的有限序列，记为：

$$\text{Queue}=\{a_1, a_2, \cdots, a_n\}$$

其中，n 表示队列的元素个数，称为队列的长度。若 $n=0$，则称为空队列。

2．队列的基本操作

队列的基本操作有：

- 队列的初始化，设置队列状态为空。
- 判断队列的状态是否为空。
- 判断队列的状态是否为满。
- 入队——将数据元素从队尾处加入队列的过程。在入队之前必须判断队列的状态是否已满，如果队列不满，则接收新数据元素入队；队列满时则不能入队，产生上溢错误（overflow）。
- 出队——从队头处取出数据元素的过程。在出队之前，必须判断队列的状态是否为空，队列空时，取不到元素，产生下溢错误（underflow）。

将队列的基本操作声明在接口 QueueInterface 中，声明如下：

```
package ds_java;
public interface QueueInterface          //队列的接口
{
    public boolean isEmpty();            //判断队列状态是否为空
    public boolean isFull();             //判断队列状态是否已满
    public boolean enqueue(Object k);    //k 对象入队
    public Object  dequeue();            //出队
}
```

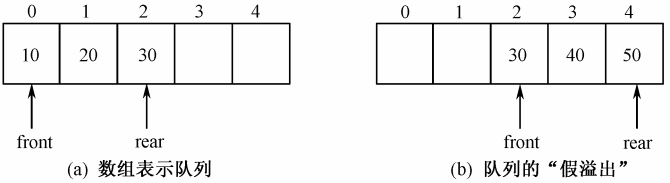
此处将队列作为抽象数据类型，数据元素的类型设置为 Object，所以入队的数据元素是 Object 对象，出队的数据元素是 Object 对象。

4.2.3 队列的存储结构及实现

同栈一样，队列也有顺序和链式两种存储结构。顺序存储结构的队列称为顺序队列，链式存储结构的队列称为链式队列。

1．队列的顺序存储结构

队列的顺序存储结构就是定义一组连续的存储空间存放队列的数据元素，如图 4.11 所示。



(a) 数组表示队列　　(b) 队列的“假溢出”

图 4.11　队列的顺序存储结构

设数组 table 长度 max=5。front 和 rear 分别是队头、队尾数据元素的下标。先有 3 个数据元素（10，20，30）入队，front=0，rear=2，如图 4.11（a）所示。再有 2 个数据元素（40 和 50）入队，2 个数据元素 10 和 20 出队后，front=2，rear=4，如图 4.11（b）所示。此时，如果再有数据元素 60 入队，应该存放于 rear 的下一位置处，rear=rear+1，那么 rear=5，说明队尾下标 rear 因数组下标越界而引起溢出，而此时顺序队列的头部已空出许多存储单元。因此，这时的“溢出”并不是由于存储空间不够而产生的。

顺序队列因多次入队和出队操作后出现的有存储空间但不能进行入队操作的溢出称为假溢出。顺序队列之所以会产生假溢出这样严重的缺陷，是因为顺序队列的存储单元没有重复使用机制。解决的办法是将顺序队列设计成环形结构。

2．顺序循环队列及操作实现

顺序循环队列是把顺序队列所使用的存储空间构造成一个逻辑上首尾相连的环形队列。为实现顺序循环队列，front 和 rear 不能简单地加 1，而应该按照如下规律进行移动：

```
front=(front+1)%table.length;
rear=(rear+1)%table.length;
```

其中，table.length 表示数组 table 的长度。当 rear 和 front 达到 table.length 后，再前进一个位置就自动为 0。这样，rear 和 front 就可以在数组中循环移动，使得数组的存储单元可以重复使用，而不会出现顺序队列数组的头部已空出许多存储单元，但队列的队尾指针却因数组下标越界而引起溢出的假溢出问题。

顺序循环队列中，front 定义为当前队头数据元素的下标，rear 定义为下一个入队数据元素位置，即当前队尾数据元素的下一位置的下标。顺序循环队列如图 4.12 所示。

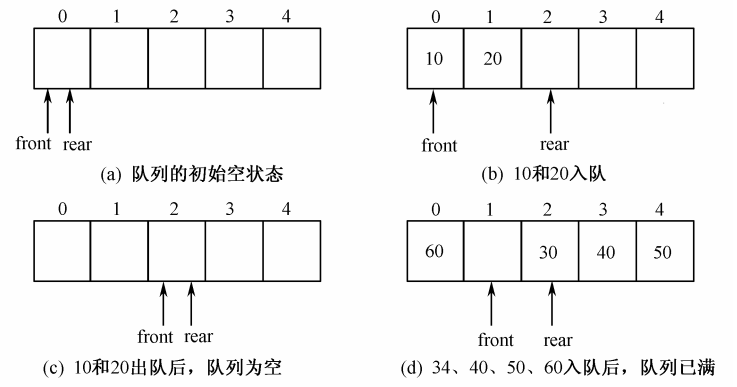


图 4.12　顺序循环队列

下面声明 Queue1 类表示顺序循环队列，实现队列接口 QueueInterface。其中，成员变量 table 用数组存储队列的数据元素，front 和 rear 分别作为队头数据元素下标和下一个入队数据元素位置的下标。

```
package  ds_java;
import ds_java.QueueInterface;
public class Queue1 implements QueueInterface   //顺序循环队列
{
   private Object table[];
```

```
    private int front,rear;                    //front 和 rear 为队列头尾下标
  }
```

Queue1 类的一个对象就是一个队列。队列数据元素的类型是 Object。

顺序循环队列的操作实现如下，这些算法写在 Queue1 类中，作为 Queue1 类的方法。

（1）队列的初始化。构造方法申请 table 数组的存储空间准备存放队列的数据元素，设置队列初始状态为空，即 front =0 且 rear=0。

```
    public Queue1(int n)                       //构造长度为 n 的空队列
    {
       table=new Object[n];
       front=rear=0;
    }
```

（2）判断队列是否为空。当 front==rear 时，队列为空。

```
    public boolean  isEmpty()                  //判断队列状态是否为空
    {
       return front==rear;
    }
```

（3）判断队列是否已满。当 front=(rear+1)% table.length 时，队列已满，此时 table 数组中仍有一个空位置。

```
    public boolean isFull()                    //判断队列状态是否已满
    {
       return front==rear%table.length+1;
    }
```

（4）入队。当队列不满时，将 *k* 对象存放在 rear 位置，作为新的队尾数据元素，rear 循环加 1。此时入队的数据元素是 Object 对象。

```
    public boolean enqueue(Object k)           //k 值入队
    {
       if(!isFull())                           //队列不满
       {
          table[rear]=k;
          rear=(rear+1) % table.length;
          return true;
       }
       else
       {
          System.out.println("队列已满，"+k+"值无法入队！");
          return false;                        //队列溢出
       }
     }
```

（5）出队。当队列不空时，取走 front 位置上的队首数据元素，front 循环加 1，front 位置上的数据元素成为新的队首数据元素。此时出队的数据元素是 Object 对象。

```
     public Object dequeue()                   //出队
```

```
    {
        Object k=null;
        if(!isEmpty())                    //队列不空
        {
            k=table[front];               //取得队头数据元素
            table[front]=null;
            front=(front+1) % table.length;
        }
        return k;                         //队列空时返回 null
    }
```

由此可见，顺序循环队列在以下几方面对顺序队列进行了改动：

- 下标 front 和 rear 循环移动，取值范围是 0~table.length，这使得存储单元可以重复使用，避免“假溢出”情况。
- 将 rear 定义为下一个入队数据元素位置的下标，使得入队时只要改变 rear，出队时只要改变 front 即可。如果 rear 仍定义为当前队尾数据元素的下标，则必须设置队列空状态为 front=-1 且 rear=-1。当第一个数据元素入队后，front=0 且 rear=0。也就是说，在入队前，不但要判断队列是否已满，还要判断队列是否为空，入队时不但要改变 rear，还要改变 front。同样，出队时也要改变 rear。
- 改变了队列已满条件。如果不设立一个空位置，则队列满的条件也是 front==rear，那么就无法与队列空的条件（front==rear）相区别。

【例 4.4】 顺序循环队列的基本操作。

源程序 Queue1_ex.java 使用 ds_java.Queue1 类。程序如下：

```
import ds_java.Queue1;
class Queue1_ex
{
    public static void main(String args[])
    {
        int i=0,n=2;
        Queue1 q1=new Queue1(20);
        while(i<args.length)
        {
            q1.enqueue(args[i]);          //入队
            System.out.print("Enqueue: "+args[i]+"\t");
            q1.output();
            i++;
        }
        for(i=0;i<n;i++)
        {
            Integer j=new Integer(i+1);
            System.out.print("EnQueue: "+(i+1)+"\t");
            q1.enqueue(j);                //入队
```

```
            q1.output();
        }
        System.out.print("DeQueue: ");
        while(!q1.isEmpty())          //全部出队
        {
            Object obj=q1.dequeue();
            System.out.print(obj.toString()+"  ");
        }
        System.out.println();
    }
}
```

运行如下命令：

```
D:\myjava>java  Queue1_ex  a  b
```

程序结果如下：

```
Enqueue: a   ds_java.Queue1:  a
Enqueue: b   ds_java.Queue1:  a b
Enqueue: 1   ds_java.Queue1:  a b 1
Enqueue: 2   ds_java.Queue1:  a b 1 2
Dequeue: a  b  1  2
```

3．队列的链式存储结构及操作实现

队列的链式存储结构以单向链表实现，如图 4.13 **所示，设** front **和** rear **分别指向队头和队尾结点。**

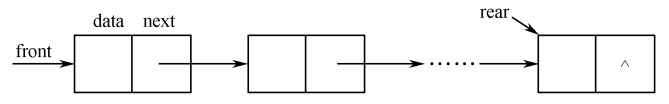


图 4.13　**队列的链式存储结构**

下面声明 Queue2 **类实现链式队列。**

```
package  ds_java;
import ds_java.OnelinkNode;
import ds_java.Onelink1;
public class Queue2 extends Onelink1//队列的链式存储结构
{
    private OnelinkNode front,rear;
}
```

Queue2 **类的一个对象就是一个队列。其中，成员变量** front **和** rear **分别指向队头和队尾结点，结点类型为单向链表的结点类** OnelinkNode**，结点数据域的类型为** int**。**

链式队列的基本操作实现如下，这些算法在 Queue2 **类中，作为** Queue2 **类的方法。**

（1**）队列的初始化。构造方法创建一条单向链表用做队列，设置队列初始状态为空。**

```
public Queue2()                    //构造空队列
```

```
{
    front=rear=null;
}
```

（2）判断队列状态是否为空。当 front==null 且 rear==null，队列为空。

```
public boolean isEmpty()                    //判断队列是否为空
{
    return (front==null) & (rear==null);
}
```

（3）判断队列是否已满。与链式栈一样，链式队列采用动态分配方式为每个结点分配内存空间，所以不需要判断队列是否已满。

（4）入队。在 rear 指向的队尾结点之后插入一个结点存放 k，并使 rear 指向新的队尾结点。

```
public boolean enqueue(int k)               //k 值入队
{
    OnelinkNode  q;
    q=new OnelinkNode(k) ;
    if(!isEmpty())                          //队列不空
        rear.next=q;                        //q 结点作为新的队尾结点
    else
        front=q;
    rear=q;
    return true;
}
```

（5）出队。当队列不空时，取走 front 指向的队首结点的数据元素，并删除该结点，使 front 指向新的队首结点。

```
public int dequeue()                        //出队
{
    int k=-1;
    if(!isEmpty())                          //队列不空
    {
        k=front.data;                       //取得队头结点数据元素
        front=front.next;                   //删除队头结点
        if(front==null)
            rear=null;
    }
    return k;                               //队列空时返回-1
}
```

链式队列的基本操作如图 4.14 所示。

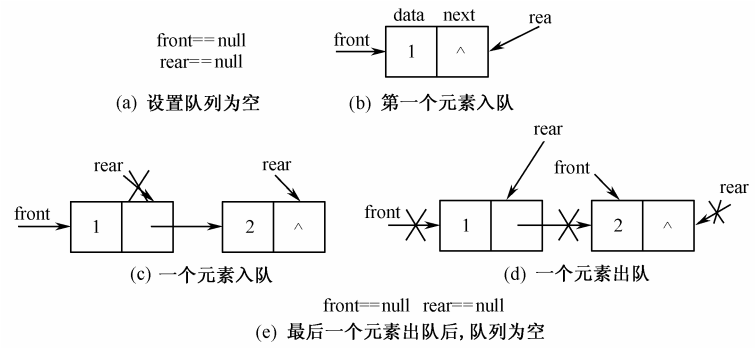


图 4.14　链式队列的基本操作

思考题：分别用单向循环链表、双向循环链表结构实现队列，并讨论其差别。

4.2.4　队列的应用举例

队列是一种特殊的线性表，具有先进先出的特性。当求解具有先进先出特性的问题时，需要使用队列。

1．处理等待问题时系统设立队列

队列具有“先进先出”的特性，当需要按一定次序等待时，系统需设立一个队列。

2．实现广度遍历算法时使用队列

实现广度遍历算法，如按层次遍历二叉树、以广度优先算法遍历图，都需要使用队列。详细算法将在以后的章节中介绍。

【例 4.5】　解素数环问题。

将 n 个数（1～n）排列成环形，使得每相邻两数之和为素数，构成一个素数环。

本例引用顺序存储结构的线性表 LinearList1 类和链式队列 Queue2 类。创建 LinearList 类的一个对象 ring1 存放素数环的数据元素，创建 Queue2 类的一个对象 q1 作为队列。静态方法 isPrime(k)判断 k 是否为素数，若是，返回 true，否则返回 false。

首先将 2～n 的数全部入队，将出队数据元素 k 与素数环最后一个数据元素相加，若两数之和是素数，则将 k 加入到素数环中，否则说明 k 暂时无法处理，必须再次入队等待。重复上述操作，直到队列为空。

```
import ds_java.Queue2;                  //引用链式存储结构的队列，元素为 int 型
import ds_java.LinearList1;             //引用顺序存储结构的线性表
public class Primering                  //求素数环
{
   public static boolean isPrime(int k)      //判断 k 是否为素数
   {
      int j=2;
      if(k==2)
         return true;
      if(k<2 || k>2 && k%2==0)
         return false;
```

```
            else
            {
                j=3;
                while(j<k && k%j!=0)
                    j=j+2;
                if(j>=k)
                    return true;
                else
                    return false;
            }
        }
        public static void main(String args[])
        {
            int i,j,k,n=10;
            Queue2 q1=new Queue2();                 //创建一个队列 q1
            LinearList1 ring1=new LinearList1(n);   //创建一个线性表 ring1 表示素数环
            ring1.add(1);                           //1 添加到素数环中
            for(i=2;i<=n;i++)                       //2~n 全部入队
                q1.enqueue(i);
            q1.output();                            //输出队列中全部数据元素
            i=1;
            while(!q1.isEmpty())
            {
                k=q1.dequeue();                     //出队
                System.out.print("Enqueue: "+k+"\t");
                j=ring1.get(i)+k;
                if(isPrime(j))                      //判断 j 是否为素数
                {
                    i++;
                    ring1.add(k);                   //k 添加到素数环中
                }
                else
                    q1.enqueue(k);                  //k 再次入队
                q1.output();
            }
            ring1.output();
        }
    }
```

程序运行结果如下：

```
ds_java.Queue2:  2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10
Enqueue: 2   ds_java.Queue2:  3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 1
```

```
Enqueue: 3   ds_java.Queue2:  4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10
Enqueue: 4   ds_java.Queue2:  5 -> 6 -> 7 -> 8 -> 9 -> 10
Enqueue: 5   ds_java.Queue2:  6 -> 7 -> 8 -> 9 -> 10 -> 5
Enqueue: 6   ds_java.Queue2:  7 -> 8 -> 9 -> 10 -> 5 -> 6
Enqueue: 7   ds_java.Queue2:  8 -> 9 -> 10 -> 5 -> 6
Enqueue: 8   ds_java.Queue2:  9 -> 10 -> 5 -> 6 -> 8
Enqueue: 9   ds_java.Queue2:  10 -> 5 -> 6 -> 8 -> 9
Enqueue: 10  ds_java.Queue2:  5 -> 6 -> 8 -> 9
Enqueue: 5   ds_java.Queue2:  6 -> 8 -> 9 -> 5
Enqueue: 6   ds_java.Queue2:  8 -> 9 -> 5 -> 6
Enqueue: 8   ds_java.Queue2:  9 -> 5 -> 6 -> 8
Enqueue: 9   ds_java.Queue2:  5 -> 6 -> 8
Enqueue: 5   ds_java.Queue2:  6 -> 8 -> 5
Enqueue: 6   ds_java.Queue2:  8 -> 5 -> 6
Enqueue: 8   ds_java.Queue2:  5 -> 6
Enqueue: 5   ds_java.Queue2:  6
Enqueue: 6   ds_java.Queue2:
table:  1 2 3 4 7 10 9 8 5 6
```

4.3 递归

递归（recursion）是数学计算中的一种思维方式，是数学和计算机科学中一种强有力的工具。程序设计中也用递归算法来实现一些问题的求解。

在数学及程序设计方法中，递归定义为：若一个对象部分地包含它自己，或用它自己定义自己，则称这个对象是递归的；若一个函数直接或间接地调用自己，则称这个函数是递归函数。递归可以出现在定义、算法和数据结构中。

1．问题的定义是递归的

数学上常用的阶乘函数、幂函数、Fibonacci 序列等，它们的定义和计算都是递归的。例如，阶乘函数 $f(n)=n!$，定义为

$$n!=\begin{cases}1 & n=0,\ 1\\ n\times(n-1)! & n\geqslant 2\end{cases} \tag{4-3}$$

式（4-3）称为阶乘函数 $n!$ 的递归定义式。像式（4-3）这样，给出被定义函数在某些自变量处的值，又给出由已知的被定义函数值逐步计算未知的被定义函数值的规则，来规定一个函数的方式，称为函数的递归定义（recursive definition）。

再如，Fibonacci 序列是指首两项为 0 和 1，以后各项的值是其前两项值之和：

{0，1，1，2，3，5，8，…}

其数学中的递归定义为：

$$f(n)=\begin{cases}n & n=0，1\\ f(n-1)+f(n-2) & n\geqslant 2\end{cases}$$

2．算法是递归的

存在自调用的算法称为递归算法（recursive arithmetic）。递归算法就是为了得到问题的解将问题推到比原问题更简单的解，然后再回到原问题上来。例如，求 $f(n)=n!$，为计算 $f(n)$，将它推到 $f(n-1)$，即

$$f(n)=n\times f(n-1)$$

而计算 $f(n-1)$的算法与 $f(n)$是一样的。

由此可见，对于一个较为复杂的问题：

- 如果能够分解成几个相对简单且解法相同或类似的子问题时，只要解决了子问题，那么原问题就迎刃而解，这就是递归求解。例如，$5!=5\times 4!$。
- 当分解后的子问题可以直接解决时，就停止分解。这些可以直接求解的问题称为递归结束条件。例如，$1!=1$。
- 根据递归定义，编写能够直接反映递归定义的递归函数来求解。

【例 4.6】 求 $n!$。

例如，求 $5!$的递归调用及返回值的情况如图 4.15 所示。

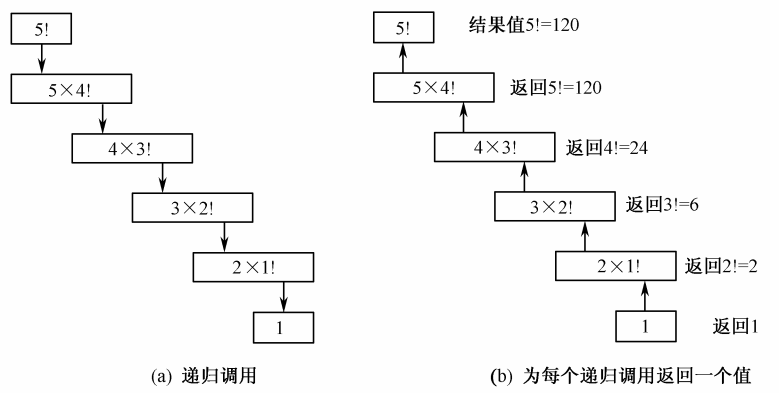


图 4.15 递归调用与返回

程序如下：

```
public class Factorial
{
    static int f(int n)                    //递归方法
    {
        if(n==0 || n==1)
            return 1;
        else
        {
            System.out.println(n+"!="+n+"*"+(n-1)+"!");
            return n*f(n-1);
        }
    }
    public static void main(String args[])
```

```
    {
        int i=5;
        if(args.length>0)
            i=Integer.parseInt(args[0]);   //将 args[0]转化为 int 值
        System.out.println(i+"!="+f(i));
    }
}
```

程序中用 Integer **类的静态方法** parseInt()**将命令行参数** args[0]**转化为** int **值。带参数的运行命令为：**

```
D:\myjava\>java  Nmul  6
```

程序运行结果如下：

```
6!=6*5!
5!=5*4!
4!=4*3!
3!=3*2!
2!=2*1!
6!=720
```

【例 4.7**】 打印数字塔。**

打印如下形式的数字塔：

```
                    1
                  1 2 1
                1 2 3 2 1
              1 2 3 4 3 2 1
            1 2 3 4 5 4 3 2 1
          1 2 3 4 5 6 5 4 3 2 1
        1 2 3 4 5 6 7 6 5 4 3 2 1
      1 2 3 4 5 6 7 8 7 6 5 4 3 2 1
    1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1
```

题目本身虽不是递归定义的，但可以用递归算法求解。程序如下：

```
public class Dig9_r
{
    static void count(int i,int n)            //递归方法
    {
        if(i<n)
        {
            System.out.print(i+"  ");
         count(i+1,n);
        }
        System.out.print(i+"  ");
    }
```

```
    public static void main(String args[])
    {
       int i,j,n=9;
       for(i=1;i<=n;i++)
       {
          for(j=n;j>i;j--)
              System.out.print("   ");        //前导空格
          count(1,i);
          System.out.println();
       }
    }
}
```

3．**数据结构是递归的**

有些数据结构是递归的。例如，链表就是一种递归的数据结构，如图 4.16 **所示。**

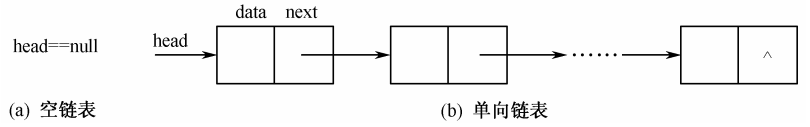


图 4.16　**单向链表**

可以将单向链表结点类的 next **链与指向链表第** 1 **个结点的** head **递归定义为：**

$$\text{next}\text{及}\text{head}=\begin{cases}\text{null} & \text{指向一个空链表}\\ \text{非null} & \text{指向一个单向链表}\end{cases}$$

【例 4.8**】　单向链表递归定义的实现。**

本例声明的 OnelinkNode2 **类，成员变量** data **为** char **类型。**output()**方法按递归算法输出链表中所有结点的数据元素。**

main()**方法中使用标准输入流** System.in**，以** read(buffer)**方法将从键盘输入的字符读取到缓冲区** buffer **中，此时必须引用** java.io **包中的类。程序如下：**

```
import java.io.*;                                    //引用输入输出流包中的类
public class OnelinkNode2                            //单向链表的结点类
{
   public char data;                                 //存放结点数据元素
   public OnelinkNode2 next;                         //指向单向链表
   public OnelinkNode2(char ch,OnelinkNode2 q)       //构造 q 的前驱结点
   {
      this.data=ch;
      this.next=q;
   }
   public OnelinkNode2(char ch)                      //构造值为 ch 的结点
   {
      this(ch,null);
   }
```

```
    public OnelinkNode2()
    {
        this('?');
    }
    public void output(OnelinkNode2 p)          //输出链表中所有结点的值
    {                                           //递归算法，必须有参数
        if(p!=null)
        {
            System.out.print(p.data+" -> ");
            output(p.next);
        }
        else
            System.out.println("null");
    }
    public static void main(String args[]) throws IOException
    {                                           //抛出异常
        System.out.print("Input: ");
        byte buffer[]=new byte[512];            //输入缓冲区
        int i=0,count=System.in.read(buffer);   //读取标准输入流
        System.out.print("Output: ");
        for(i=0;i<count;i++)                    //输出 buffer 元素值
            System.out.print("  "+buffer[i]);
        System.out.println();
        for(i=0;i<count;i++)                    //按字符方式输出 buffer
            System.out.print((char)buffer[i]);
        System.out.println("count="+count);     //buffer 实际长度
        OnelinkNode2 head=null;
        for(i=0;i<count-2;i++)                  //创建链表
        {
            head=new OnelinkNode2((char)buffer[i],head);
            head.output(head);                  //输出链表
        }
    }
}
```

程序运行时，从键盘输入“1234”，结果如下：

```
Input: 1234
Output:
  49  50  51  52  13  10
1234
count = 6
1 -> null
```

```
2 -> 1 -> null
3 -> 2 -> 1 -> null
4 -> 3 -> 2 -> 1 -> null
```

习　题　4

4.1　分别设计用单向循环链表、双向循环链表结构实现的队列，并讨论它们之间的差别。

4.2　用递归算法计算 Fibonacci 序列的前 20 项元素值。

4.3　用递归算法求两个整数的最大公因数。

设有不全为 0 的整数 a、b，记 gcd(a，b)为它们的最大公因数，即同时能整除 a 和 b 的公因数中的最者。按照 Euclid 的辗转相除算法，gcd(a，b)具有如下性质：

$$\begin{cases} \gcd(a,b)=\gcd(b,a) \\ \gcd(a,b)=\gcd(-a,b) \\ \gcd(a,0)=|a| \\ \gcd(a,b)=\gcd(b,a\ \%\ b), \end{cases} \qquad 0\le a\%b<b$$

4.4　用递归算法实现表达式中的括号匹配问题。

4.5　用递归算法计算表达式。

在表达式的数学定义中，表达式中允许用括号将子表达式括起来，从而改变表达式的运算次序，而子达式的定义与表达式相同。这种用子表达式来定义表达式的方式就是一种递归定义。

实　习　4

1．实验目的：

使用栈、队列或递归算法求解问题。

2．题意

（1）走迷宫

一个迷宫可以看成一个二维数组，如图 4.17 所示。每个迷宫都有一个入口和一个出口，其中白色单元示通路，黑色单元表示路不通。试寻找一条通过迷宫的路径，从入口进入迷宫，每次移动只能从一个白色元移到相邻的白色单元，直到移至出口时为止。

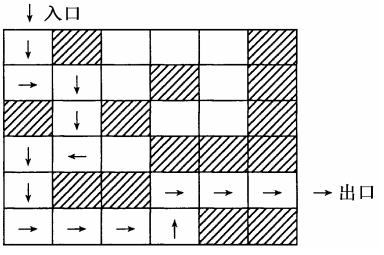


图 4.17　迷宫

（2）骑士游历

在国际象棋的棋盘（8 行 8 列）上放置一“马”，按“马走日字”的规则，马要到达棋盘上的每一格，且每格只到达一次。若给定起始位置（x_0，y_0），编程探索出一条路线，沿这条路线“马”能遍历棋盘上的

有格子。这就是“骑士游历”问题。

按国际象棋的规则，马在棋盘的某一位置（x，y）上，下一步有 8 个方向可走，如图 4.18 所示。

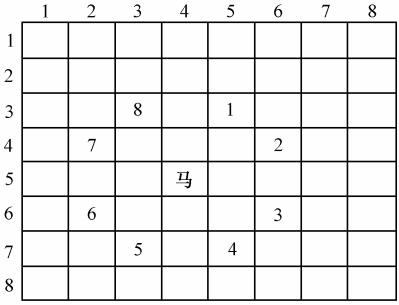


图 4.18　骑士游历

3．实验要求

选择使用栈、队列或递归算法求解以上问题。

第 5 章　数组和广义表

在高级程序设计语言中，数组是一种数据类型。由前几章讨论的线性结构可知，数组是线性结构及其他数据结构实现顺序存储的基础。

一维数组可以看成一个顺序存储结构的线性表，二维数组定义为“其数据元素为一维数组”的线性表。矩阵一般采用二维数组存储，但对特殊矩阵和稀疏矩阵可采用一些特殊方法进行压缩存储。

广义表是一种复杂的数据结构，它是线性表结构的扩展。

本章介绍数组、稀疏矩阵和广义表的基本概念，以及稀疏矩阵和广义表的存储结构。

建议本章授课 4 学时，实验 3 学时。

5.1　数组

数组（array）是一组相同数据类型的数据元素的集合，数组元素按次序存储于一个地址连续的内存空间中。数组元素在数组中的位置通常称为数组的下标，通过元素的下标，可以找到存放该元素的存储地址，从而可以访问该数组元素的值。就此意义，数组可以看成二元组〈下标，值〉的一个集合。

数组下标的个数就是数组的维数，有一个下标是一维数组，有两个下标就是二维数组，以此类推。

5.1.1　一维数组

在数据结构中，数组是 n（$n>1$）个相同数据类型的数据元素 a_1，a_2，…，a_n 构成的，占用一块地址连续的内存单元的有限序列。其中，n 称为数组长度。

一维数组中，每两个相邻数据元素之间都有直接前驱和直接后继的关系。当系统为一个数组分配内存空间时，数组的首地址就确定了。假设数组的首地址为 $\mathrm{Loc}(a_1)$，每个数据元素占用 c 个存储单元，则第 i 个数据元素的地址为：

$$\mathrm{Loc}(a_i)=\mathrm{Loc}(a_1)+(i-1)\times c$$

也就是说，已知数组元素的位置（即下标），系统就可计算出该数组元素的存储地址。具有这种特性的存储结构通常称为随机存储结构。数组是一种随机存储结构。所以，通过数组名加下标的形式，可以访问数组中任意一个指定的数组元素。

根据系统为数组分配内存空间的方式不同，可以将数组分为两种：静态数组和动态数组。

- 静态数组——声明时给出数组元素个数。当程序开始运行时，数组即获得系统分配的

一块地址连续的内存空间。静态数组所占用的内存空间由系统自动管理。

- 动态数组——声明时不指定数组长度。当程序运行中需要使用数组时，向系统申请数组的存储单元空间，并给出数组长度。当数组使用完之后，需要向系统归还所占用的内存空间。

在 Java 中，数组元素既可以是简单数据类型，也可以是引用类型。而且 Java 中的数组都是动态数组，即声明数组变量时不指定数组长度，使用 new 运算符为数组分配空间后，数组才真正占用一片地址连续的存储单元空间。而当数组使用完之后，不需要向系统归还所占用的内存空间。因为 Java 的垃圾回收机制将自动判断对象是否在使用，并能够自动销毁不再使用的对象，收回对象所占的资源。

5.1.2 多维数组

1．多维数组的概念

多维数组是线性表的推广，二维数组定义为“其数据元素为一维数组”的线性表。例如二维数组可以表示一个矩阵

$$\boldsymbol{A}_{m\times n}=\begin{bmatrix} a_{1,1} & a_{1,2} & \Lambda & a_{1,n} \\ a_{2,1} & a_{2,2} & \Lambda & a_{2,n} \\ \Lambda & \Lambda & \Lambda & \Lambda \\ a_{m,1} & a_{m,2} & \Lambda & a_{m,n} \end{bmatrix}$$

$\boldsymbol{A}_{m\times n}$ 定义为由 $m\times n$ 个元素 $a_{i,j}$ 组成的矩阵，也可以看成是由 m 行（一维数组）一维数组组成的，或是 n 列（一维数组）一维数组组成的。

$\boldsymbol{A}_{m\times n}$ 中的每个元素 $a_{i,j}$ 同时属于两个线性表：第 i 行的线性表和第 j 列的线性表。一般情况下，$a_{i,j}$ 有 2 个前驱（行前驱 $a_{i-1,j}$ 和列前驱 $a_{i,j-1}$）以及 2 个后继（行后继 $a_{i+1,j}$ 和列后继 $a_{i,j+1}$），但 $a_{1,1}$ 是起点，没有前驱；$a_{m,n}$ 是终点，没有后继。而且边界上的元素 $a_{1,j}$（j=1，…，n），$a_{i,1}$（i=1，…，m），$a_{m,j}$（j=1，…，n）和 $a_{i,n}$（i=1，…，m），只有一个后继或只有一个前驱。

同样，三维数组 $\boldsymbol{A}_{m\times n\times p}$ 中的每个元素 $a_{i,j,k}$ 最多可以有 3 个前驱和 3 个后继。推而广之，n 维数组的每个元素可以有 m 个前驱和 m 个后继。

2．多维数组的遍历

按照某种次序访问一个数据结构中的所有元素，并且每个数据元素恰好访问一次，称为对该数据结构的遍历。遍历一种数据结构，将得到一个所有数据元素的线性序列。

一维数组只有一种遍历次序，而二维数组则有两种次序：行优先和列优先。

将数组元素按行排列，第 $i+1$ 行紧跟在第 i 行后面。对于 $\boldsymbol{A}_{m\times n}$，可以得到如下线性序列：

$$a_{1,1}，a_{1,2}，\cdots，a_{1,n}，a_{2,1}，a_{2,2}，\cdots，a_{2,n}，\cdots，a_{m,1}，a_{m,2}，\cdots，a_{m,n}$$

将数组元素按列排列，第 $j+1$ 列紧接在第 j 列后面。对于 $\boldsymbol{A}_{m\times n}$，可以得到如下线性序列：

$$a_{1,1}，a_{2,1}，\cdots，a_{m,1}，a_{1,2}，a_{2,2}，\cdots，a_{m,2}，\cdots，a_{1,n}，a_{2,n}，\cdots，a_{m,n}$$

3．多维数组的顺序存储结构

对于多维数组，可以按行优先或列优先的次序进行顺序存储。例如，将二维数组 $\boldsymbol{A}_{m\times n}$ 按行优先次序存储在内存以后，元素 $a_{i,j}$ 的地址为：

$$\mathrm{Loc}(a_{i,j}) = \mathrm{Loc}(a_{1,1}) + (i-1)n + (j-1)$$

其中，$\mathrm{Loc}(a_{i,j})$ 为 $a_{i,j}$ 的存储地址，$\mathrm{Loc}(a_{1,1})$ 为 $a_{1,1}$ 的存储地址。

按列优先次序存储数组时，元素 $a_{i,j}$ 的地址为：

$$\mathrm{Loc}(a_{i,j}) = \mathrm{Loc}(a_{1,1}) + (j-1)m + (i-1)$$

所以，可以用数组元素的下标值通过简单的函数关系，计算出该数组元素的存储地址。也就是说，二维数组的顺序存储结构也是随机的存储结构，可以对数组元素进行随机存取，其存储密度为 1。

这种存储方式适合于数组元素大多数不为零的情况。如果数组元素大多数为零，仍然用顺序的方法把每个元素都存起来就不合算，看起来存储密度为 1，实际上占用了许多单元去存储重复的零信息，造成浪费。

如果数组中有很多零且非零元素具有某种分布规律时，可以只对非零元素进行顺序存储，此时仍可以进行随机存取。例如，下三角矩阵

$$\boldsymbol{A}_{m\times n} = \begin{bmatrix} a_{1,1} & 0 & \Lambda & 0 \\ a_{2,1} & a_{2,2} & \Lambda & 0 \\ \Lambda & \Lambda & \Lambda & \Lambda \\ a_{m,1} & a_{m,2} & \Lambda & a_{m,n} \end{bmatrix}$$

当 $i<j$ 时，$a_{i,j}=0$。如果按行优先次序遍历矩阵中的下三角元素，便得到如下的线性序列：

$$a_{1,1}，a_{2,1}，a_{2,2}，\cdots，a_{m,1}，a_{m,2}，\cdots，a_{m,n}$$

如果按行优先次序只将矩阵中的下三角元素顺序存储，第 1 行到第 $i-1$ 行元素的个数为：

$$\sum_{k=1}^{i-1} k = \frac{i(i-1)}{2}$$

因此，元素 $a_{i,j}$（$i\geq j$）的地址可用下式计算：

$$\mathrm{Loc}(a_{i,j}) = \mathrm{Loc}(a_{1,1}) + \frac{i(i-1)}{2} + (j-1),\ 1\leq j \leq i \leq n$$

如果数组中大多数元素值为零且非零元素的分布没有规律时，可以用 5.2 节介绍的稀疏矩阵存储方法。

【例 5.1】 矩阵相加。

本例声明 Matrix 类表示矩阵，成员 table 是一个元素类型为整型的二维数组。add()方法实现两个矩阵相加。程序如下：

```
package  ds_java;
public class Matrix
{
   private int table[][];
   public Matrix(int n)
   {
      table=new int[n][n];
   }
   public Matrix()
   {
      this(10);
```

```
    }
    public Matrix(int mat[][])
    {
        this(mat.length);
        System.arraycopy(mat,0,table,0,mat.length);
    }
    public int get(int i,int j)          //获得第 i 行第 j 列的元素
    {
        return  table[i-1][j-1];
    }
    public void set(int i,int j,int k)   //设置第 i 行第 j 列的元素值为 k
    {
        table[i-1][j-1]=k;
    }
    public void add(Matrix b)            //两个矩阵相加
    {
        Matrix a=this;
        int i,j;
        for(i=0;i<a.table.length;i++)
            for(j=0;j<a.table[i].length;j++)
               a.table[i][j]=a.table[i][j]+b.table[i][j];
    }
    public void output()                 //遍历,输出各元素值
    {
        int i,j;
        for(i=0;i<table.length;i++)
        {
            for(j=0;j<table[i].length;j++)
                System.out.print("  "+table[i][j]);
            System.out.println();
        }
        System.out.println();
    }
}
```

源程序 Matrix_ex.java**，引用** Matrix **类对两个矩阵相加。程序如下：**

```
import ds_java.Matrix;
class Matrix_ex
{
    public static void main(String args[])
    {
        int m1[][]={{1,2,3},
```

```
                    {4,5,6},
                    {7,8,9}};
        Matrix a=new Matrix(m1);
        int m2[][]={{1,0,0},
                    {0,1,0},
                    {0,0,1}};
        Matrix b=new Matrix(m2);
        a.add(b);
        a.output();
    }
}
```

程序运行结果如下：

```
2  2  3
4  6  6
7  8  10
```

思考题：请在 Matrix **类中增加下列功能：**

- **求一个矩阵的转置矩阵。**
- **判断一个矩阵是否为上三角矩阵。**
- **计算两个矩阵的乘积。**

5.2 稀疏矩阵

设 $\boldsymbol{A}_{m\times n}$ **中有** t **个非零元素，则称** $\delta=t(m\times n)^{-1}$ **为矩阵的稀疏因子，当** $\delta\leq 0.05$ **时，称为稀疏矩阵（**sparse matrix**）。**

在存储稀疏矩阵时，为了节省存储单元，最好只存储其中的非零元素。采用压缩存储方式可以压缩掉零元素的存储空间，但会失去数组的随机存取功能。

稀疏矩阵的非零元素由 3 **部分组成：行下标、列下标和矩阵元素值，这称为稀疏矩阵的三元组。设** $i, j, a_{i,j}$ **分别表示稀疏矩阵非零元素的行下标、列下标和矩阵元素值，则一个稀疏矩阵的三元组集合可以惟一地确定一个稀疏矩阵。例如，稀疏矩阵**

$$\boldsymbol{A}=\begin{bmatrix}1&0&0&0\\0&0&0&0\\2&0&7&0\\0&0&8&9\end{bmatrix}$$

可以用稀疏矩阵的三元组序列表示为：{{1,1,1},{3,1,2},{3,3,7},{4,3,8},{4,4,9}}**。**

要压缩存储稀疏矩阵，就可以只存储稀疏矩阵的非零元素，即只存储稀疏矩阵的三元组。存储稀疏矩阵三元组的方法有顺序存储结构和链式存储结构两种。

5.2.1 三元组的顺序存储结构

三元组的顺序存储结构是按照行优先（或列优先）的原则，将稀疏矩阵三元组存储在一个线性表中，线性表的每个数据元素对应稀疏矩阵的一个三元组。例如，上述稀疏矩阵 $\boldsymbol{A}$ **的顺序存储结构如表** 5-1 **所示。**

表 5-1　稀疏矩阵三元组的顺序存储结构

数组下标	行下标	列下标	数据元素值
0	1	1	1
1	3	1	2
2	3	3	7
3	4	3	8
4	4	4	9

1．声明顺序存储结构的稀疏矩阵的三元组类

将顺序存储结构的稀疏矩阵的三元组声明为如下的 SparseNode1 类：

```
public class SparseNode1                    //稀疏矩阵的三元组表示的结点结构
{
    public int row;                         //行下标
    public int column;                      //列下标
    public int data;                        //值
    public SparseNode1(int i,int j,int k)
    {
        row=i;
        column=j;
        data=k;
    }
    public SparseNode1()
    {
        this(0,0,0);
    }
    public void output()                //输出一个元素的三元组值
    {
        System.out.println("\t"+row+"\t"+column+"\t"+data);
    }
}
```

SparseNode1 类的一个对象表示稀疏矩阵的一个三元组，它只记录了稀疏矩阵中的一个元素的位置和值。

2．声明顺序存储结构的稀疏矩阵类

下面声明的 Sparse1 类的一个对象则表示一个稀疏矩阵，成员 table 是一个数组表示的线性表，元素类型为 SparseNode1 类。构造方法将一个稀疏矩阵转换成三元组表示法，output 方法输出 table 数组的元素值。

```
public class Sparse1                            //稀疏矩阵的三元组顺序存储结构
{
    protected SparseNode1 table[];              //数组，元素为三元组
```

```
    public Sparse1(int mat1[][])            //建立三元组表示
    {
        System.out.println("稀疏矩阵:");
        int n=mat1.length;
        table=new SparseNode1 [n*2];        //估计数组长度
        int i,j,k=0;
        for(i=0;i<mat1.length;i++)
        {
            for(j=0;j<mat1[i].length;j++)
            {
               System.out.print("  "+mat1[i][j]);
               if(mat1[i][j]!=0)
               {                    //mat1[i][j]是矩阵中第 i+1 行第 j+1 列的元素
                   table[k]=new SparseNode1(i+1,j+1,mat1[i][j]);
                   k++;
               }
            }
            System.out.println();
        }
    }
    public Sparse1()
    {  }
    public void output()          //输出一个稀疏矩阵中所有元素的三元组值
    {
        int i,j;
        System.out.println("稀疏矩阵三元组的顺序表示:");
        System.out.println("\t\t 行下标\t 列下标\t 值");
        for(i=0;i<table.length;i++)
        {
            System.out.print("table["+i+"] = ");
            if(table[i]!=null)
               table[i].output();    //调用 SparseNode1 类方法输出三元组值
            else
               System.out.println("null");
        }
    }
}
```

【例 5.2】 稀疏矩阵三元组的顺序存储结构。

下面程序调用 Sparse1 类实现稀疏矩阵三元组的顺序存储结构。程序如下：

```
public class Sparse1_ex
{
```

```
    public static void main(String args[])
    {
       int mat1[][]={{1,0,0,0},          //稀疏矩阵
                     {0,0,0,0},
                     {2,0,7,0},
                     {0,0,8,9}};
       Sparse1 s1=new Sparse1(mat1);     //一个对象表示一个稀疏矩阵
       s1.output();
    }
}
```

程序运行结果如下：

稀疏矩阵:

```
 1  0  0  0
 0  0  0  0
 2  0  7  0
 0  0  8  9
```

稀疏矩阵三元组的顺序表示:

```
            行下标  列下标  值
table[0] =   1       1    1
table[1] =   3       1    2
table[2] =   3       3    7
table[3] =   4       3    8
table[4] =   4       4    9
table[5] = null
table[6] = null
table[7] = null
```

从数学角度看，矩阵的下标应该从 1 **开始计数。由于** Java **数组的下标从** 0 **开始，本例构造方法中将** mat1[i][j]**中的下标转换成从** 1 **开始计数，即** mat1[i][j]**对应矩阵中第** *i*+1 **行第** *j*+1 **列的元素。**

SparseNode1 **类中有** output()**方法，**Sparse1 **类中也有** output()**方法，两者虽然同名，但表示不同的含义。**SparseNode1 **类中的** output()**方法输出一个矩阵元素的三元组值，**Sparse1 **类中的** output()**方法输出一个稀疏矩阵中所有元素的三元组值，其中每个元素均调用** SparseNode1 **类的** output()**方法输出一个三元组值。**

顺序存储结构的稀疏矩阵有两个缺点：

- **数组长度不易设定，可能存在溢出与浪费问题。**
- **插入、删除操作不方便。若矩阵元素的值发生变化，一个为零的元素变成非零元素，就要向线性表中插入一个三元组；若非零元素变成零元素，就要从线性表中删除一个三元组。为了保持线性表元素间的相对次序，进行插入和删除操作时，就必须移动元素。**

思考题：

- **返回** table **数组的实际长度。**

- 求稀疏矩阵的转置矩阵。
- 实现三元组表示的两个稀疏矩阵相加。

5.2.2 三元组的链式存储结构

以顺序存储结构存储稀疏矩阵的三元组可以节省很多存储单元，但缺点是在非零元素增加或减少时，插入和删除操作不方便。以链式存储结构存储稀疏矩阵的三元组则可以克服这一不足，常用的链式存储结构有两种：行（列）的单链表示和十字链表示。

1．行的单链表示

将稀疏矩阵每行上的若干个非零元素作为结点链接成一个单向链表，每条链表第 1 个结点的引用存放在数组中。对于前述稀疏矩阵 A，其行的单链表示如图 5.1 所示。

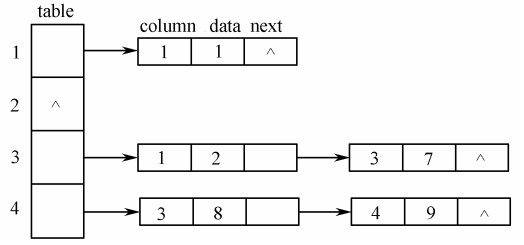


图 5.1 稀疏矩阵的行的单链表示

其中，链表的每个结点由 3 个成员组成：column（列下标），data（值）和 next（后继结点的引用）。table 数组元素存放每条链表第 1 个结点的引用。声明稀疏矩阵以行的单链表示方式如下的 SparseNode2 类：

```
public class SparseNode2                    //稀疏矩阵行的单链表示
{
   public int  column;                      //列下标
   public int  data;                        //值
   public SparseNode2 next;                 //后继结点的引用
   public SparseNode2(int i,int j)
   {
      column=i;
      data=j;
      next=null;
   }
   public SparseNode2()
   {
      this(0,0);
   }
   public void output(SparseNode2 p)        //输出链表所有结点的值
   {                                        //递归算法，必须有参数
      if(p!=null)
      {
```

```
            System.out.print("  "+p.column+"  "+p.data+" -> ");
            output(p.next);
        }
        else
            System.out.println("null");
     }
}
```

SparseNode2 **类的一个对象表示链表中的一个结点，对应稀疏矩阵中的一个元素。**SparseNode2 **类中的** output()**方法以递归算法输出链表中所有结点的值，功能与** Onelink1 **中的** output()**方法相同。**

下面的 Sparse2 **类实现稀疏矩阵的行的单链表示，成员** table **是一个数组，元素类型为** SparseNode2 **类。构造方法是将一个稀疏矩阵转换成行的单链表示，**output()**方法输出稀疏矩阵若干条链表中的全部结点值。**

```
public class Sparse2                    //稀疏矩阵行的单链表示
{
   protected SparseNode2 table[];       //数组元素引用链表的第 1 个结点
   public Sparse2(int mat1[][])         //建立稀疏矩阵行的单链表示
   {
      int n=mat1.length;
      table=new SparseNode2 [n+1];
      int i,j,k=0;
      SparseNode2 p=null,q;
      for(i=0;i<n;i++)
      {
         p=table[i+1];
         for(j=0;j<mat1[i].length;j++)
           if(mat1[i][j]!=0)
           {
              q=new SparseNode2(j+1,mat1[i][j]);
              if(p==null)
                 table[i+1]=q;
              else
                 p.next=q;
              p=q;
           }
      }
   }
   public Sparse2()
   {  }
   public  void output()                        //输出稀疏矩阵行的单链表示
   {
```

```
        int i,j;
        System.out.println("稀疏矩阵行的单链表示:");
        for(i=0;i<table.length;i++)
        {
            System.out.print("table["+i+"]=");
            if(table[i]!=null)
               table[i].output(table[i]);    //输出一条链表中所有结点的值
            else
               System.out.println("null");
        }
    }
}
```

使用行的单链表示，存取一个元素的时间复杂度为 $O(s)$，其中 s 为一行中非零元素的个数

【例 5.3】 稀疏矩阵行的单链表示。

下面程序调用 Sparse2 类实现稀疏矩阵行的单链表示。

```
public class Sparse2_ex
{
    public static void main(String args[])
    {
        int mat1[][]={{1,0,0,0},
                      {0,0,0,0},
                      {2,0,7,0},
                      {0,0,8,9}};
        Sparse2 s1=new Sparse2(mat1);
        s1.output();
    }
}
```

程序运行结果如下：

```
  稀疏矩阵行的单链表示:
table[0]=null
table[1]=  1  1 -> null
table[2]=null
table[3]=  1  2 ->   3  7 -> null
table[4]=  3  8 ->   4  9 -> null
```

为使数组下标（从 0 开始计数）与矩阵行列（从 1 开始计数）表示一致，本例的 table 数组多申请了一个单元，以 table[1]到 table[4]实现稀疏矩阵行的单链表示，没有使用 table[0]元素。

按行的单链表示的稀疏矩阵，每个结点可以很容易地找到行的后继结点，但很难找到列的后继结点。为充分表示行和列的后继结点，可以采用十字链表示。

思考题：

- 求稀疏矩阵的转置矩阵。
- 实现行的单链表示的两个稀疏矩阵相加。

2．十字链表示

将行的单链表示和列的单链表示结合起来存储稀疏矩阵称为十字链表示，如图 5.2 所示

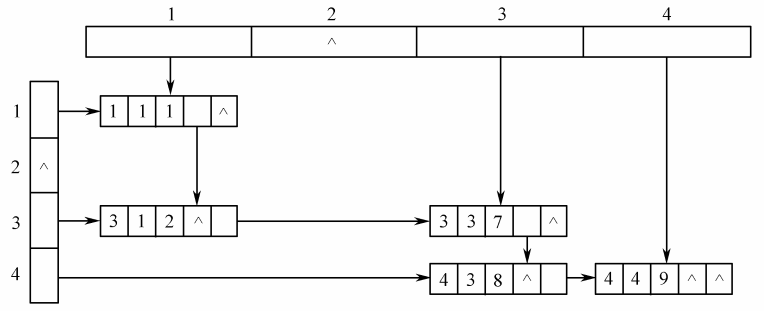


图 5.2　稀疏矩阵的十字链表示

每个结点表示一个非零元素。每个结点有 5 个成员：行下标，列下标，值，行后继引用以及列后继引用。

从行的角度看，需要一个数组存放行的单链的第 1 个结点；同样，从列的角度看，还需要一个数组存放列的单链的第 1 个结点。使用这种表示，各行的非零元素和各列的非零元素都分别链接在一起，最多有 $m+n$ 条链。对元素的查找可顺着所在行的链进行，也可以顺着所在列的链进行。查找一个元素的最大时间为 $O(s)$，其中 s 为某行或某列上非零元素的个数。

十字链表示法的优点是结构灵活，且便于使用。

5.3　广义表

5.3.1　广义表的概念

1．广义表的定义

广义表（general list）是 n（$n\geq 0$）个数据元素 $a_1, a_2, \cdots, a_n$ 组成的有限序列，记为：

$$\text{List}=(a_1, a_2, \cdots, a_n)$$

其中，a_i 或为不可分的单元素（称为原子），或为可再分的广义表（称为子表）。数据元素个数 n 称为广义表的长度，当 $n=0$ 时，为空表。为了区分原子和表，在下面的书写中约定用大写字母表示表，用小写字母表示原子。例如，

```
L=(a，b)                              //线性表，长度为 2
T=(c，L)=(c，(a，b))                  //L 为 T 的子表，T 的长度为 2
G=(d，L，T)=(d，(a，b)，(c，(a，b))) //L、T 为 G 的子表，G 的长度为 3
S=()                                  //空表，长度为 0
S1=(S)=(())                           //非空表，元素是一个空表，长度为 1
Z=(e，Z)=(e，(e，(e，(……))))         //递归表，Z 的长度为 2
```

广义表是一种递归定义的数据结构。广义表的深度是指表中所含括号的层数。例如，L 的

深度为 1，T 的深度为 2，G 的深度为 3。而空表的深度为 1，原子的深度为 0。

如果广义表的子表是其自身，则称该广义表为递归表。递归表的深度是无穷值，长度是有限值。

如果规定任何表都是有名字的表，则为了既标明每个表的名字，又说明它的组成，可以将表名写在本表对应的括号前，于是上例的各表又可以写成：

```
L(a，b)
T(c，L(a，b))
G(d，L(a，b)，T(c，L(a，b)))
S()
S1(())
Z(e，Z(e，Z(e，Z(……))))
```

2．广义表的特性

1）广义表是一种线性结构

广义表的数据元素之间是线性关系，即广义表的数据元素之间有着固定的相对次序，如同线性表。但广义表并不等价于线性表，仅当广义表的数据元素全部是原子时，该广义表为线性表。所以广义表是线性表的扩展，而线性表是广义表的特例。例如，广义表 L(a,b)其实就是线性表。

2）广义表也是一种多层次的结构

当广义表的数据元素中包含子表时，该广义表就是一种多层次的结构。例如，T(c,L(a,b))表示一种树形的层次结构。本章在此用到树结构和图结构的一些概念，详细内容将在以后章节中介绍。

所以，广义表也可以看成树的推广。如果限制表中成分的共享和递归，所得到的结构就是树结构，树中的叶结点对应广义表中原子，非叶结点对应子表。

3）广义表可为其他广义表共享

例如，上述广义表 L 同时作为广义表 T 和 G 的子表。在 T 和 G 中不必列出子表的值，而通过子表的名称来引用。在应用问题中，利用广义表的共享特性可以减少存储结构中的数据冗余，以节约存储空间。

4）广义表可以是一个递归表

广义表中有共享或递归成分的子表就是图结构。对应的图中结点入度可能大于 1，并且可能出现环。

通常将与树对应的广义表称为纯表，将允许数据元素共享的广义表称为再入表，将允许递归的广义表称为递归表，它们之间的关系满足：

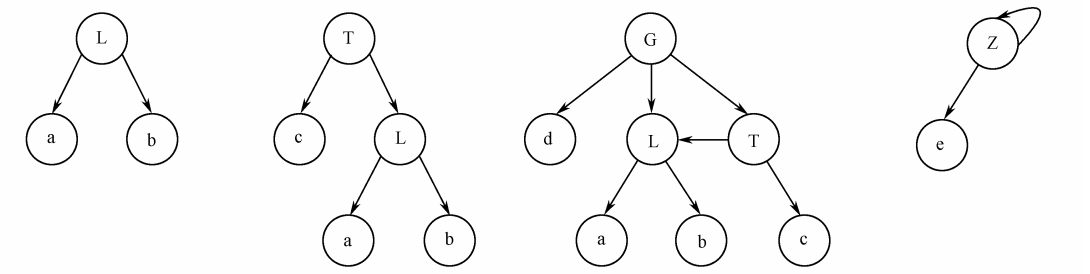
递归表 ⊃ 再入表 ⊃ 纯表 ⊃ 线性表

3．广义表的图形表示

广义表可以与有根有序的有向图建立对应关系，即可以用广义表的形式表达线性表、树和图等基本的数据结构。

对应的方法如下：主表对应于图的根，广义表中的数据各元素顺序对应于根邻接到的各顶点，如果某个数据元素是原子，则对应的结点称为原子结点；否则可继续上述对应的过程来处理子表，直到全部对应完毕。用广义表的形式表达线性表、树和图等基本的数据

结构如图 5.3 所示。



(a) 线性结构的线性表L(a,b)　(b) 树结构的纯表T(c,L(a,b))　(c) 图结构的再入表G(d,L(a,b),T(c,L(a,b)))　(d) 图结构的递归表Z(e,Z)

图 5.3　广义表表示的多种结构对应的图形

- 当广义表 L(a，b)的数据元素全部是原子时，该广义表 L 为线性结构的线性表。
- 当广义表 T(c，L)的数据元素中有子表，但没有共享和递归成分时，该广义表 T 为树结构的纯表。
- 当广义表 G(d，L，T)的数据元素中有子表，并且有共享成分时，该广义表 G 为图结构的再入表。
- 当广义表 Z(e，Z)的数据元素中有子表且有递归成分时，该广义表 Z 为图结构的递归表。

图 5.3（a）所示的线性表，表面上看像一棵树，实质上只是一个链表，将它转化成等价的二叉树就非常清楚（原理详见第 6 章），如图 5.4 所示。

(a) 线性表L(a,b)　(b) 线性表 L(a,b) 转换成二叉树结构就是一个链表

图 5.4　线性表 L(a,b)与其对应的二叉树

4．广义表的操作

用广义表的形式可以表示线性表、树和图等基本的数据结构，广义表的操作与线性表、树和图中讨论的操作类似，主要有：

- 建立一个广义表。
- 在广义表中插入一个数据元素。
- 删除一个数据元素。
- 复制一个广义表。
- 判别某数据元素是否为原子。
- 判别两个广义表是否相等。
- 判别广义表是否为空表。

5.3.2　广义表的存储结构

线性表有顺序存储结构和链式存储结构两种，非线性结构的广义表大多采用链式存储结

构。树结构和图结构的存储表示将在第 6 章和第 8 章中讨论。此处主要讨论适合再入表和递归表的链式存储结构。

1．广义表的单链表示

类似线性表的单向链表结构，广义表可以用单链结构存储。每个结点由如下 3 个域组成：

atom	data	next

其中，atom 是一个标志位，表示该数据元素是否为原子。取值如下：

$$\text{atom}=\begin{cases}0 & (\text{本结点为子表})\\ 1 & (\text{本结点为原子})\end{cases}$$

当 atom=1 时，data 存放本原子的信息；当 atom＝0 时，data 存放子表中第一个数据元素所对应结点的地址。next 成员存放与本数据元素同层的下一个数据元素所对应结点的地址，当本数据元素是所在层的最后一个数据元素时，next==null。用单链方式表示的再入表和递归表如图 5.5 所示。

当广义表中有共享成分时，共享的结点将重复出现。例如，再入表 G 中有子表 L 和 T，T 中也有子表 L，所以在图中子表 L 的结点就出现两次，出现的位置不同，在结构中所起的作用也就不同。

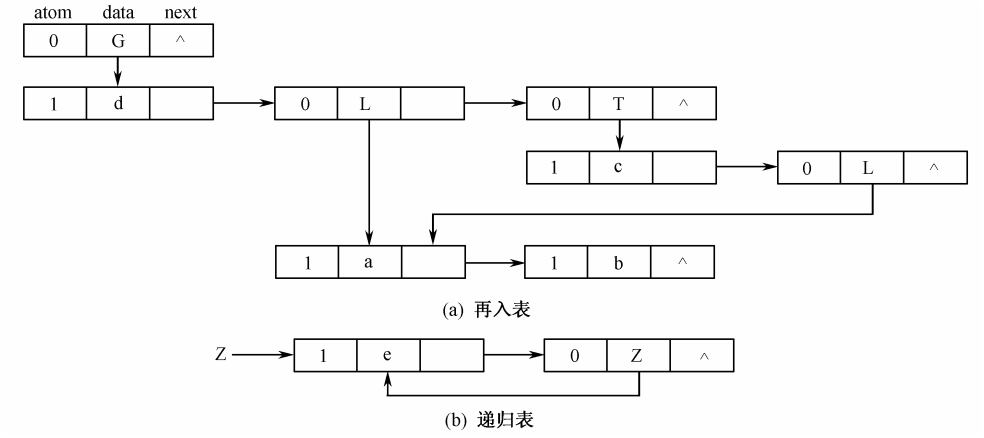


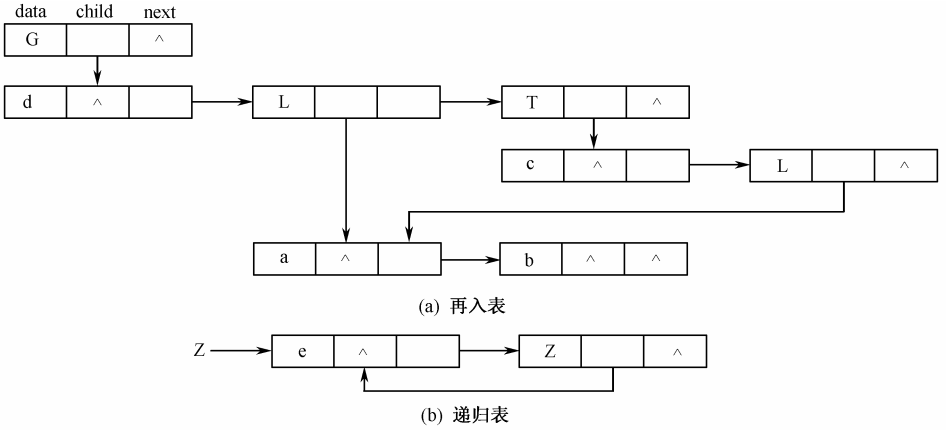
图 5.5　广义表的单链表示

2．广义表的双链表示

采用双向链表的结构存储广义表，每个结点由如下 3 个域组成：

data	child	next

其中，data 存放数据元素信息，child 存放子表中第一个数据元素所对应结点的地址，next 存放与本数据元素同层的下一个数据元素所对应结点的地址。当本数据元素是所在层的最后一个数据元素时，next==null。所以，本结点是原子的充要条件是 child==null。用双链方式表示的再入表和递归表如图 5.6 所示。



(a) 再入表

(b) 递归表

图 5.6　广义表的双链表示

习　题　5

5.1　**在** Matrix **类中增加以下功能：**

（1）求一个矩阵的转置矩阵。

（2）判断一个矩阵是否为上三角矩阵。

（3）计算两个矩阵相乘。

5.2　**在三元组表示的稀疏矩阵** Sparse1 **类中，增加以下功能：**

（1）稀疏矩阵的转置矩阵。

（2）两个稀疏矩阵相加。

5.3　**在行的单链表示的稀疏矩阵** Sparse2 **类中，增加以下功能：**

（1）稀疏矩阵的转置矩阵。

（2）两个稀疏矩阵相加。

5.4　**定义用十字链表示的稀疏矩阵的结点及类。**

实　习　5

1．实验目的：应用类的方法对二维数组进行操作。

2．题意：找出一个二维数组的鞍点，即该位置上的元素在该行上最大，在该列上最小。一个二维数组可能没有鞍点；如果有，那么它只有一个。

第 6 章　树和二叉树

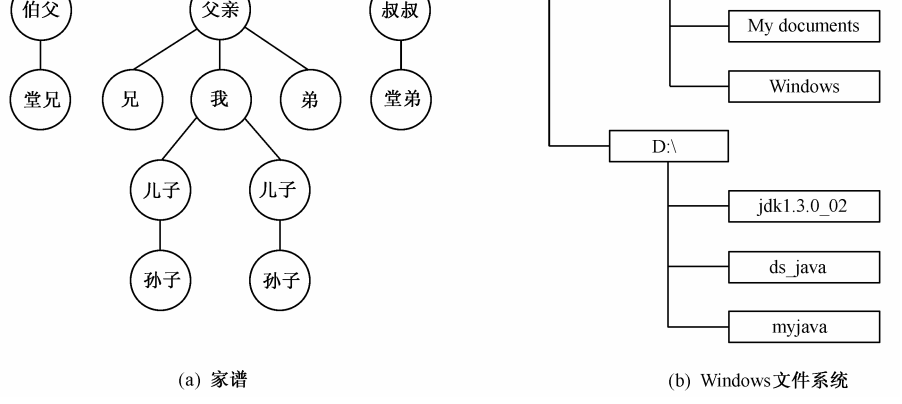
树结构定义为数据元素之间具有层次关系的非线性结构。树结构除根结点外，每个数据元素只有一个前驱数据元素，可有零个或若干个后继数据元素，根结点没有前驱数据元素。树结构有树和二叉树两种。二叉树是最多只有两个子树且两个子树有左右之分的有序树。

本章介绍具有层次关系的树结构，重点讨论二叉树的定义、性质、存储结构和遍历算法，并讨论线索二叉树的定义、存储结构和遍历算法，以及作为完全二叉树应用的堆排序。

建议本章授课 10 学时，实验 4 学时。

6.1　树

树结构从自然界中的树抽象而来，有树根、从根发源类似枝杈的分支关系以及作为分支结点的叶子等。生活中存在大量的树结构，如家谱、Windows 文件系统等，虽然表现形式各不相同，但本质上结构是相同的，如图 6.1 所示。



(a) 家谱　　(b) Windows文件系统

图 6.1　树结构举例

6.1.1 树的定义

树（tree）是由 n（$n \geq 0$）个结点组成的有限集合。$n=0$ 的树称为空树；$n>0$ 的树 T 的特点如下：

- 有一个特殊的结点称为根结点（root），它只有直接后继结点，没有直接前驱结点。
- 当 $n>1$ 时，除根结点之外的其他结点分为 m（$m \geq 0$）个互不相交的集合 T_1，T_2，…，T_m，每个集合 T_m（$1 \leq i \leq m$）本身又是一棵结构与树类同的子树（subtree）。每棵子树的根结点有且仅有一个直接前驱结点，但可以有零或多个直接后继结点。

树的递归定义显示了树的固有特性，树中每个结点都是该树中某一棵子树的根。图 6.2 表示树结构。图 6.2（a）中，$n=0$，表示一棵空树。图 6.2（b）中，$n=1$，树中只有一个结点 A，A 就是根结点。图 6.2（c）中，$n=10$，A 为树的根结点。其他结点分为 3 棵互不相交的子集 T_1、T_2 和 T_3 作为 A 的子树，$T_1=\{B, E, F\}$，$T_2=\{C, G\}$，$T_3=\{D, H, I, J\}$，子树的根分别为 B、C 和 D。

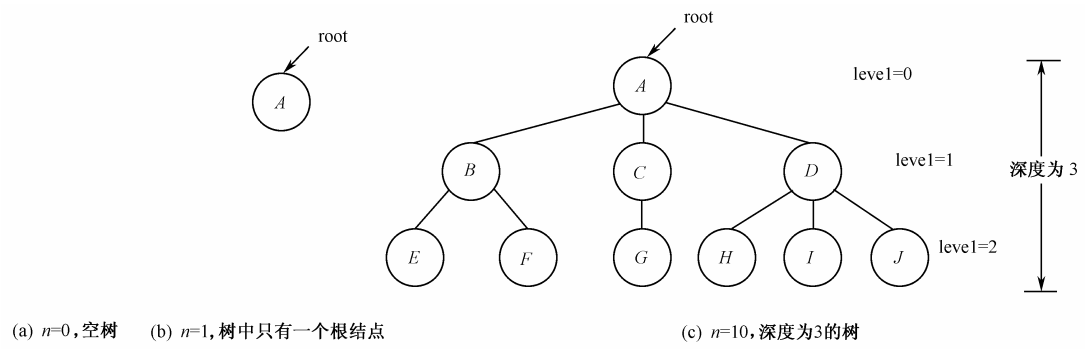


图 6.2　树结构

6.1.2 树的术语

下面以图 6.2（c）为例，说明与树有关的术语。

1．结点

结点（node）由数据元素以及指向子树的地址构成。例如，图 6.2（c）表示一棵具有 10 个结点的树。

2．孩子结点与双亲结点

若 X 结点有子树，则子树的根结点称为 X 的孩子（child）结点，又称子女结点。相应地，X 称为其孩子的双亲（parents）结点，又称父母结点。一棵树中，根没有双亲结点。例如，B、C、D 是 A 的子树的根，所以 A 的孩子结点是 B、C、D，即 A 是 B、C、D 的双亲结点，但 A 没有双亲结点。

3．兄弟结点

同一双亲的孩子结点之间互称兄弟（sibling）结点。例如，B、C 和 D 是兄弟，E 和 F 也是兄弟，但 F 和 G 不是兄弟。

4．祖先结点与后代结点

X 结点的祖先（ancestor）是指从根到 X 所经过的所有结点。X 结点的后代（descendant）是

指 X 的所有孩子结点，以及孩子结点的孩子。例如，E 的祖先是 B 和 A，E 则是 A 的后代结点

5．结点的度

结点的度（degree）定义为结点所拥有子树的棵数。例如，A 的度是 3，E 的度是 0。

6．叶子结点与分支结点

叶子（leaf）结点是指度为 0 的结点，又称为终端结点。除叶子结点之外的其他结点，称为分支结点或非叶结点，又称为非终端结点。例如，E 和 F 是叶子结点，B、C 和 D 是非叶结点。

7．树的度

树的度是指树中各结点度的最大值。例如，图 6.2（c）中树的度为 3。

8．边

设树中 X 结点是 Y 结点的双亲结点，有序对〈X，Y〉称为连接这两个结点的边（edge）。例如，〈A，B〉和〈B，E〉都是树的边。

9．路径与路径长度

如果（X_1, X_2, …, X_k）是由树中结点组成的一个序列，且〈X_i, X_{i+1}〉（$1 \leq i \leq k-1$）都是树的边，则该序列称为从 X_1 到 X_k 的一条路径（path）。路径长度（path length）为路径上边的条数。例如，从 A 到 E 的路径是（A, B, E），路径长度是 2。

10．结点的层次

X 结点的层次（level）是指从根到 X 结点的路径长度。令根结点的层次为 1，其余结点的层次等于它双亲结点的层次加上 1。显然，兄弟结点的层次相同。例如，A 的层次为 1，B 的层次为 2，E 的层次为 3。F、G 不是兄弟，称为同一层上的结点。

11．树的深度或高度

树中结点的最大层次数称为树的深度（depth）或高度（height）。例如，图 6.2（c）中树的深度为 3。

12．无序树与有序树

由树的定义可知，若结点的子树 T_0，T_1，…之间没有次序，称为无序树。如果树中结点的子树 T_0，T_1，…从左至右是有次序的（即不能交换），则称该树为有序树（orderd tree）。通常所说的树是指无序树。

13．森林

若干棵互不相交的树的集合称为森林（forest）。给森林加上一个根结点就变成一棵树，将树的根结点删除就变成森林。

6.1.3 树的广义表形式表示

除了直观的图示法可以表示树结构之外，还可以用广义表的形式表示树结构。例如，图 6.2（c）所示树的广义表表示形式为：

$$A（B（E，F），C（G），D（H，I，J））$$

6.2 二叉树的定义及性质

6.2.1 二叉树的定义

1．二叉树的递归定义

二叉树（binary tree）是 n（$n \geq 0$）个结点组成的有限集合，$n=0$ 时称为空二叉树；$n>0$ 时二叉树由一个根结点和两棵互不相交的、分别称为左子树和右子树的子二叉树构成。

二叉树也是递归定义的。

二叉树是一种有序树，因为二叉树中每个结点的两棵子树有左、右之分，即使只有一个子树，也要区分是左子树还是右子树。

二叉树的结点最多只有两棵子树，所以二叉树的度最多为 2。但二叉树与度为 2 的树的结构是不等价 的。根本区别在于有序和无序。例如图 6.3 中的两棵树，如果看成是树结构，则图（a）和（b）表示同一棵树；如果看成是二叉树结构，则图（a）和（b）表示两棵不同的二叉树。

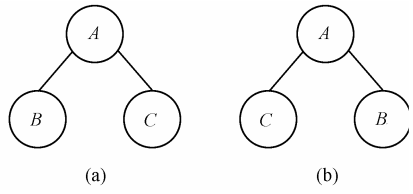


图 6.3 二叉树与度为 2 的树

树中定义的度、层次等术语，同样适用于二叉树。

2．二叉树的基本形态

二叉树有 5 种基本形态，如图 6.4 所示（设二叉树的结点个数为 n）。

- $n=0$，二叉树为空。
- $n=1$，二叉树只有一个结点作为根结点。
- $n=2$，二叉树由根结点、非空的左子树和空的右子树组成。
- $n=2$，二叉树由根结点、空的左子树和非空的右子树组成。
- $n=3$，二叉树由根结点、非空的左子树和非空的右子树组成。

其中，图 6.4（c）和（d）是两种不同形态的二叉树。

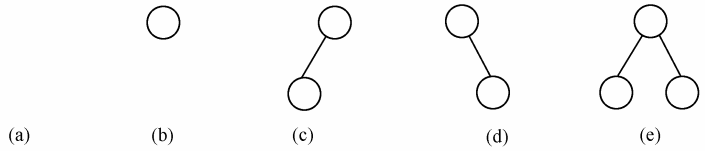


图 6.4 二叉树的基本形态

由此可知，3 个结点的树与二叉树具有不同的基本形态，如图 6.5 所示。

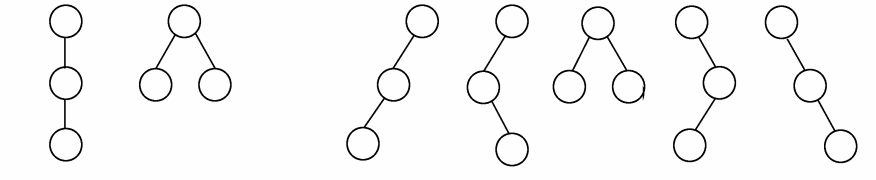


图 6.5 3 个结点的树与二叉树的基本形态

6.2.2 二叉树的性质

1．性质一

若根结点的层次为 1，则二叉树第 i 层的结点数目最多为 2^{i-1}（$i \geq 1$）个。

证明：用归纳法证明。

归纳基础：根是 $i=1$ 层上的惟一结点，故 $2^{i-1}=2^0=1$，命题成立。

归纳假设：对所有 j（$1 \leq j < i$），j 层上的最大结点数为 2^{j-1}。

归纳步骤：根据归纳假设，第 $i-1$ 层上的最大结点数为 2^{i-2}；由于二叉树中每个结点的度最大为 2，故第 i 层上的最大结点数为 $2 \times 2^{i-2} = 2^{i-1}$。命题成立。

2．性质二

在深度为 k 的二叉树中，至多有 2^k-1 个结点（$k \geq 0$）。

由性质一可知，在深度为 k 的二叉树中，最大结点数为 $\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$。

3．性质三

二叉树中，若叶子结点数为 n_0，度为 2 的结点数为 n_2，则有 $n_0=n_2+1$。

证明：设二叉树结点数为 n，度为 1 的结点数为 n_1，则有

$$n = n_0 + n_1 + n_2 \tag{6-1}$$

而度为 1 的结点有 1 个子女，度为 2 的结点有 2 个子女，叶子结点没有子女，根结点不是任何结点的子女，从子女结点数角度看，有

$$n = 0 \times n_0 + 1 \times n_1 + 2 \times n_2 + 1 \tag{6-2}$$

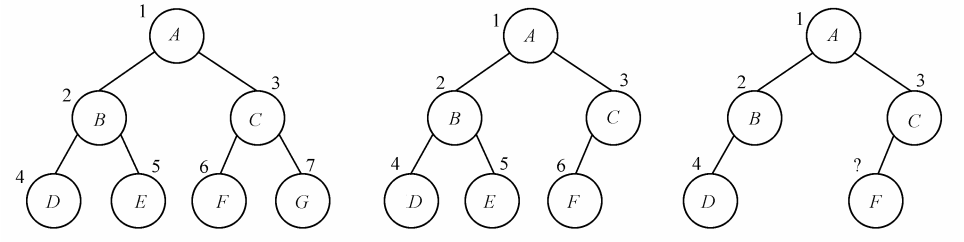
综合上述两式，可得 $n_0 = n_2 + 1$，即二叉树中叶子结点数比度为 2 的结点数多 1。

4．满二叉树与完全二叉树

一棵深度为 k 的满二叉树（full binary tree）是具有 2^k-1（$k \geq 0$）个结点的二叉树。从定义知，满二叉树每一层的结点数目都达到最大值。

对满二叉树的结点进行连续编号，约定编号从根结点开始，自上而下，每层自左至右，如图 6.6（a）所示。

一棵具有 n 个结点深度为 k 的二叉树，如果它的每一个结点都与深度为 k 的满二叉树中编号为 1～n 的结点一一对应，则称这棵二叉树为完全二叉树（complete binary tree），如图 6.6（b）所示。



(a) 满二叉树　(b) 完全二叉树　(c) 非完全二叉树

图 6.6　满二叉树与完全二叉树

由定义可知，完全二叉树具有下列特点：

- 满二叉树一定是完全二叉树，而完全二叉树不一定是满二叉树。完全二叉树只有最下面一层可以不满，其上各层都可看成满二叉树。
- 完全二叉树是具有满二叉树结构的而不一定满的二叉树。
- 完全二叉树至多只有最下面两层结点的度可以小于 2。
- 完全二叉树最下面一层的结点都集中在该层最左边的若干位置上。图 6.6（c）不是一棵完全二叉树。

5．性质四

如果一棵完全二叉树有 n 个结点，则其深度 $k=\lfloor \log_2 n \rfloor+1$。

6．性质五

若将一棵具有 n 个结点的完全二叉树按顺序表示，对于编号为 i（$1 \le i \le n$）的结点，有如下特点：

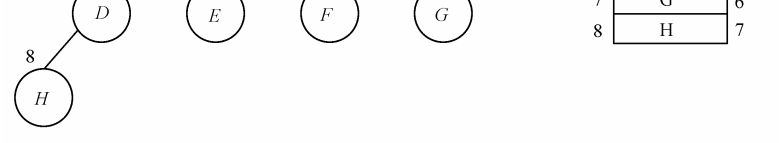
- 若 $i=1$，则 i 为根结点，无双亲；若 $i \ne 1$，则 i 的双亲是编号为 $i/2$ 的结点。
- 若 $2i \le n$，则 i 的左孩子是编号为 $2i$ 的结点；若 $2i > n$，则 i 无左孩子。
- 若 $2i+1 \le n$，则 i 的右孩子是编号为 $2i+1$ 的结点；若 $2i+1 > n$，则 i 无右孩子。这个性质在堆排序算法中将得到充分应用。

6.2.3 二叉树的存储结构

二叉树的存储结构有两种：顺序存储结构和链式存储结构。

1．二叉树的顺序存储结构

二叉树的顺序存储结构适用于完全二叉树，对完全二叉树进行顺序编号，将编号为 i 的结点存放在数组下标为 $i-1$ 的位置上。图 6.7 将完全二叉树的结点按顺序存放在数组中。



(a) 完全二叉树　　(b) 顺序存储完全二叉树

图 6.7　顺序存储结构的完全二叉树

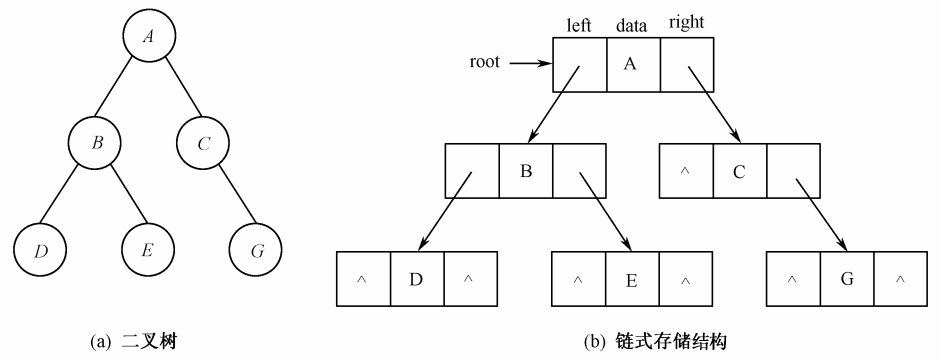
顺序存储完全二叉树时，根据二叉树的性质五，通过计算可以直接得到双亲、左孩子结点和右孩子结点的位置。

2．二叉树的链式存储结构

一般情况下，采用链式存储结构来存储二叉树。每个结点有 3 个域：

- data 表示结点的数据元素。
- left 指向该结点的左孩子结点，即左子树的根结点。
- right 指向该结点的右孩子结点，即右子树的根结点。

root 指向二叉树的根结点。若二叉树为空，则 root==null。若结点的左（右）子树为空，则其 left（right）链为空值（null），如图 6.8 所示。



(a) 二叉树　　(b) 链式存储结构

图 6.8　二叉树的链式存储结构

思考题：在一棵具有 *n* 个结点的链式存储的二叉树中，有多少个空链？

6.2.4　声明二叉树类

1．二叉树的结点类

将链式存储结构的二叉树的结点声明为 TreeNode1 类，其中有 3 个成员变量：data 用于表示数据元素，left 和 right 分别是指向左、右孩子结点的链。3 个成员变量都是公有的，可以被其他类访问。构造方法将创建一个有值的结点。

```
package ds_java;
public class TreeNode1                          //二叉树的结点类
{
   public String data;                          //数据元素
   public TreeNode1 left,right;                 //指向左、右孩子结点的链
   public TreeNode1()
   {
      this("?");
   }
   public TreeNode1(String d)                   //构造有值结点
   {
      data=d;
      left=right=null;
   }
```

2．二叉树类

声明 Tree1 类表示链式存储结构的二叉树，其中成员变量 root 指向二叉树的根结点。

```
package ds_java;
```

```
import ds_java.TreeNode1;
public class Tree1                          //以先根、中根次序遍历序列建立二叉树
{
    protected TreeNode1 root;               //指向二叉树的根结点
    public Tree1()                          //构造空二叉树
    {
        root=null;
    }
}
```

6.3 二叉树的遍历

6.3.1 二叉树遍历的概念

遍历（traversal）二叉树就是按照一定规则和次序访问二叉树中的所有结点，并且每个结点仅被访问一次。所谓访问，是指对每一个结点的数据元素进行查阅、修改等操作。一次完整的遍历按照一种规则，对二叉树中的结点产生一种线性次序。

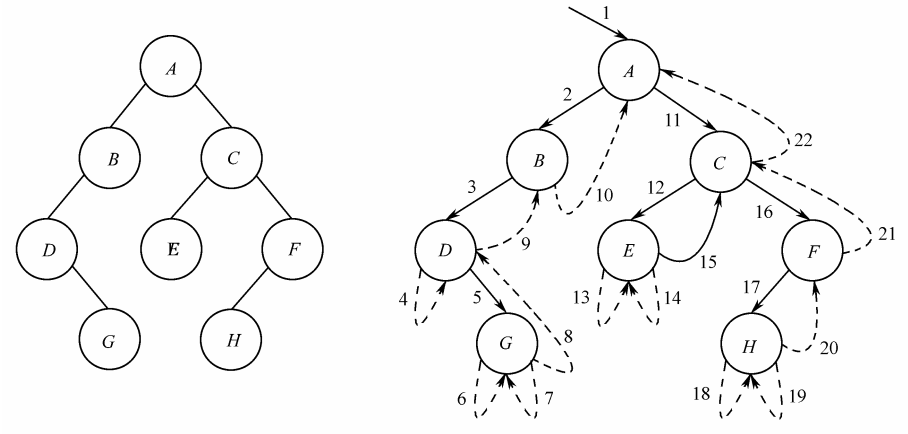
若规定对子树的访问按“先左后右”的次序进行，则遍历二叉树有 3 种次序：

- 先根次序——访问根结点，遍历左子树，遍历右子树。
- 中根次序——遍历左子树，访问根结点，遍历右子树。
- 后根次序——遍历左子树，遍历右子树，访问根结点。

按先根次序遍历二叉树的过程如下：

- 若二叉树为空，返回；否则，继续。
- 从根结点开始，访问当前结点。
- 若当前结点的左子树不空，则沿着 left 链进入该结点的左子树。
- 若当前结点的右子树不空，则沿着 right 链进入该结点的右子树。

图 6.9 中，二叉树的 3 种遍历产生的序列如下：



(a) 二叉树　　(b) 二叉树的遍历过程

图 6.9　二叉树的遍历过程

- **先根次序——***A, B, D, G, C, E, F, H*。
- **中根次序——***D, G, B, A, E, C, H, F*。
- **后根次序——***G, D, B, E, H, F, C, A*。

根据二叉树的遍历规则：在先根序列中，根结点是最先被访问的；而在后根序列中，根结点是最后被访问的；在中根序列中，排在根结点前面的都是左子树上的结点，在根结点后面的都是其右子树上的结点。所以，先根次序或后根次序反映双亲与孩子结点的层次关系，中根次序反映兄弟结点间的左右次序。

6.3.2 二叉树遍历的递归算法

1．按先根次序遍历二叉树的递归算法

按先根次序遍历二叉树的递归算法描述如下：

- **若二叉树为空，返回；否则，继续。**
- **从根结点开始，访问当前结点。**
- **按先根次序遍历当前结点的左子树。**
- **按先根次序遍历当前结点的右子树。**

在二叉树的结点类 TreeNode1 **中，增加按先根次序遍历二叉树的递归方法。**

```
public void preorder(TreeNode1 p)        //先根次序遍历二叉树
{
   if(p!=null)
   {
      System.out.print(p.data+" ");
      preorder(p.left);
      preorder(p.right);
   }
}
```

2．按中根次序遍历二叉树的递归算法

以中根或后根次序遍历二叉树也是递归算法，与先根次序遍历相似，只是访问根结点的时刻不同。在二叉树的结点类 TreeNode1 **中，增加按中根次序遍历二叉树的递归方法。**

```
public void inorder(TreeNode1 p)        //中根次序遍历二叉树
{
   if(p!=null)
   {
     inorder(p.left);
     System.out.print(p.data+" ");
     inorder(p.right);
   }
}
```

3．按后根次序遍历二叉树的递归算法

在二叉树的结点类 TreeNode1 **中，增加按后根次序遍历二叉树的递归方法。**

```
public void postorder(TreeNode1 p)      //后根次序遍历二叉树
{
   if(p!=null)
   {
      postorder(p.left);
      postorder(p.right);
      System.out.print(p.data+" ");
   }
}
```

4．调用 3 种次序遍历二叉树的递归方法

在二叉树类 Tree1 中，增加如下的 3 个方法，分别调用 TreeNode1 类中按 3 种次序遍历二叉树的递归方法。

```
public void preorderTraversal()         //先根次序遍历二叉树
{
   System.out.print("先根次序：  ");
   if(root!=null)
      root.preorder(root);
   System.out.println();
}
public void inorderTraversal()          //中根次序遍历二叉树
{
   System.out.print("中根次序：  ");
   if(root!=null)
      root.inorder(root);
   System.out.println();
}
public void postorderTraversal()            //后根次序遍历二叉树
{
   System.out.print("后根次序：  ");
   if(root!=null)
      root.postorder(root);
   System.out.println();
}
```

6.3.3 建立二叉树

由于二叉树是数据元素之间具有层次关系的非线性结构，而且二叉树中每个结点的两个子树有左右之分。所以，建立一棵二叉树必须明确以下两点：

- 结点与双亲结点及孩子结点间的层次关系。
- 兄弟结点间的左右子树的顺序关系。

对于给定的一棵二叉树，遍历产生的先根、中根、后根序列是惟一的；反之，已知二叉树的一种遍历序列，并不能惟一确定一棵二叉树。因为遍历序列反映的是：在某种条件下，将二叉树结构映射成的线性关系。先根次序或后根次序反映双亲与孩子结点的层次关系，中根次

反映兄弟结点间的左右次序。所以，已知先根和中根两种遍历序列，或中根和后根两种遍历序列才能够惟一确定一棵二叉树。而已知先根和后根两种遍历序列仍无法惟一确定一棵二叉树。

1．按先根和中根次序遍历序列建立二叉树

已知二叉树的先根及中根两种次序的遍历序列，可惟一确定一棵二叉树。

证明：设二叉树的先根及中根次序遍历序列分别为字符数组 prestr 和 instr。

（1）由先根次序知，该棵二叉树的根为 prestr[1]。

（2）在 instr 的下标中存在 k，使得 instr[k] = prestr[1]。

由中根次序知，instr[k]之前的结点在根的左子树上，instr[k]之后的结点在根的右子树上。因此，根的左子树由 k-1 个结点组成：

- 先根序列——prestr[2], …, prestr[k]。
- 中根序列——instr[1], …, instr[k – 1]。

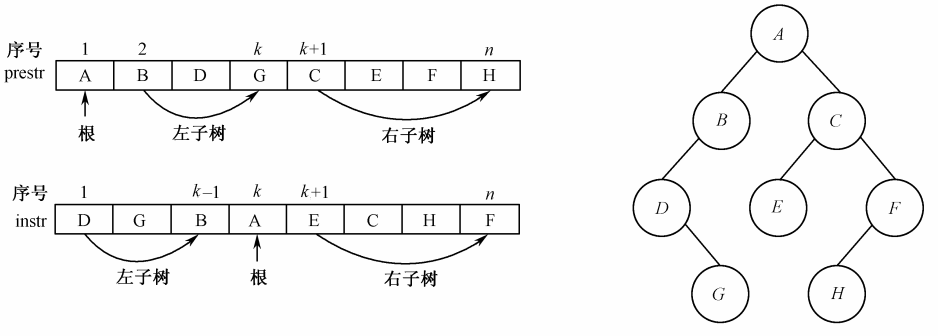
根的右子树由 n-k 个结点组成：

- 先根序列——prestr[k+1], …, prestr[n]。
- 中根序列——instr[k+1], …, instr[n]。

其中，n = prestr.length( )。

（3）以此递归，可建立惟一的一棵二叉树。

对于先根遍历次序 prestr="ABDGCEFH"，中根遍历次序 instr="DGBAECHF"，按先根次序和中根次序建立二叉树的过程如图 6.10 所示。



(a) 先根与中根遍历序列　　(b) 所建立的二叉树

图 6.10　按先根和中根次序遍历序列建立二叉树的过程

同理可证明：中根与后根次序遍历序列可惟一确定一棵二叉树。

思考题：举一反例，说明先根与后根序列不能惟一确定一棵二叉树。

在二叉树类 Tree1 中，增加 enter( )方法实现以先根次序遍历序列 prestr、中根次序遍历序列 instr 建立一棵二叉树的算法，并返回这棵二叉树的根结点。

```
public Tree1(String prestr,String instr)//构造方法
{
   root=enter(prestr,instr);
}
public TreeNode1 enter(String prestr,String instr)
{                                                    //返回所建立二叉树的根结点
   TreeNode1 p=null;
```

```
        int k,n;
        String first,presub,insub;
        n=prestr.length();
        if(n>0)
        {
           System.out.print("prestr="+prestr+"\t instr="+instr);
           first=prestr.substring(0,1);          //取串的第 1 个字符作为根
           p=new TreeNode1(first);               //建立结点
           k=instr.indexOf(first);               //确定根在中根序列中的位置
           System.out.println("\t first="+first+"\t  k="+k);
           presub=prestr.substring(1,k+1);
           insub=instr.substring(0,k);
           p.left=enter(presub,insub) ;          //建立左子树，递归调用
           presub=prestr.substring(k+1,n);
           insub=instr.substring(k+1,n);
           p.right=enter(presub,insub);          //建立右子树，递归调用
        }
        return p;                                //n=0 时为空串，返回 null
    }
```

【例 6.1**】 以先根和中根次序遍历序列建立二叉树。**

程序 Tree1_ex.java **调用** Tree1 **类，以先根、中根次序建立一棵二叉树。二叉树的先根和中根序列可以由参数** args[ ]**得到。**

```
    import ds_java.Tree1;
    public class Tree1_ex                          //以先根、中根次序建立二叉树
    {
        public static void main(String args[])
        {
            String prestr="ABDGCEFH";
            String instr="DGBAECHF";
            if(args.length>1)
            {
                prestr=args[0];
                instr=args[1];
            }
            Tree1 t1=new Tree1(prestr,instr);
            t1.preorderTraversal();
            t1.inorderTraversal();
            t1.postorderTraversal();
        }
    }
```

程序运行结果如下：

```
prestr=ABDGCEFH   instr=DGBAECHF   first=A   k=3
prestr=BDG        instr=DGB        first=B   k=2
prestr=DG         instr=DG         first=D   k=0
prestr=G          instr=G          first=G   k=0
prestr=CEFH       instr=ECHF       first=C   k=1
prestr=E          instr=E          first=E   k=0
prestr=FH         instr=HF         first=F   k=1
prestr=H          instr=H          first=H   k=0
```

先根次序： A B D G C E F H

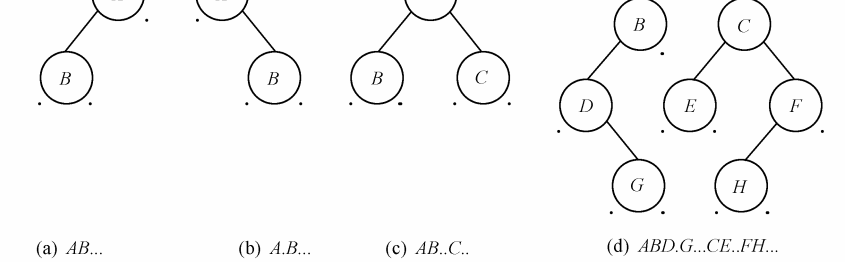
中根次序： D G B A E C H F

后根次序： G D B E H F C A

2．以标明空子树的先根次序遍历序列建立二叉树

由前面的分析我们知道，已知二叉树的先根遍历序列不能惟一确定一棵二叉树。例如，先根遍历序列为{*A*，*B*}的二叉树可以有两棵，如图 6.11（a）和（b）所示。

如果我们能够在先根遍历序列中加入反映兄弟结点间的左右次序的信息，例如，以'.'标明空子树，则可以惟一确定一棵二叉树。标明空子树的二叉树及其先根次序遍历的序列如图 6.11 所示。



(a) *AB...* (b) *A.B...* (c) *AB..C..* (d) *ABD.G...CE..FH...*

图 6.11　标明空子树的二叉树及其先根次序遍历序列

这种标明空子树的先根遍历序列，明确了二叉树中结点与双亲、孩子及兄弟间的关系，因此，它可以惟一确定一棵二叉树。

设 str 表示二叉树带空子树的先根遍历序列，以 str="*ABD...C.E..*"为例，建立二叉树的递归算法描述如下：

（1）str 的第 1 个字符一定是二叉树的根，建立 *A* 结点作为二叉树的根。此时，*A* 结点的两条链 left 和 rigth 初始化为空值。

（2）str 的第 2 个字符一定是根的左孩子，建立 *B* 结点，欲将 *B* 作为根 *A* 的左孩子结点，但此时 *A* 的 left 链还未得到指向 *B* 结点的值。

（3）下一字符是'*D*'，建立 *D* 结点，欲将 *D* 作为 *B* 的左孩子结点。

（4）下一字符是'.'，表示 *D* 结点的左子树为空，则 *D* 的 left 链为 null。

（5）下一字符是'.'，表示 *D* 结点的右子树为空，则 *D* 的 right 链为 null。

（6）以 *D* 结点为根的子树建成后，使 *B* 的 left 链指向 *D* 结点，则 *B* 的左子树建成。

（7）重复执行步骤 3～6，若下一字符是'.'，返回 null；否则建立相应结点。

（8）最后，以 *A* 结点为根的二叉树建成后，使 root 指向 *A* 结点。

建立二叉树的过程如图 6.12 所示。

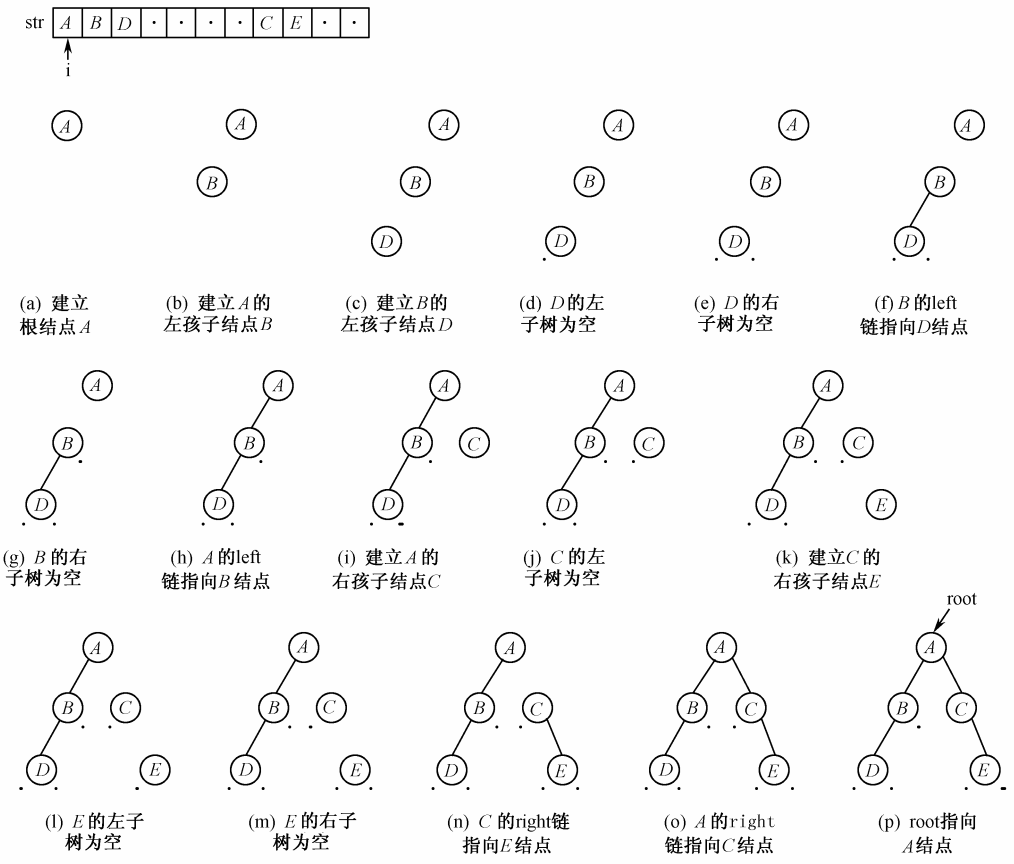


图 6.12　以标明空子树的先根次序建立二叉树的过程

【例 6.2】　以标明空子树的先根次序遍历序列建立二叉树。

Tree2 类继承 Tree1 类中的成员变量 root 和多个成员方法，增加私有成员变量 str 和 i，s 表示二叉树带空子树的先根次序，i 指出 str 中的当前位置。程序如下：

```
package ds_java;
import ds_java.TreeNode1;
import ds_java.Tree1;
public class Tree2 extends Tree1     //以标明空子树的先根次序建立二叉树
{
   private String str;               //表示二叉树带空子树的先根遍历序列
   private int i=0;
   public Tree2()                    //构造空二叉树
   {
      super();
   }
```

```
    public Tree2(String s)
    {
        str=s;
        System.out.println("str="+str);
        if(str.length()>0)
            root=enter();
    }
    public TreeNode1 enter()          //返回所建立二叉树的根结点
    {
        TreeNode1 p=null;
        char ch=str.charAt(i);
        i++;
        if(ch!='.')
        {
            p=new TreeNode1(ch+"");       //非"."时，建立结点
            p.left=enter();               //建立左子树，递归调用
            p.right=enter();              //建立右子树，递归调用
        }
        return p;                         //ch="."时，返回 null
    }
}
```

源程序 Tree2_ex.java **调用** Tree2 **类建立一棵二叉树。程序如下：**

```
import ds_java.Tree2;
public class Tree2_ex                     //以标明空子树的先根次序建立二叉树
{
    public static void main(String args[])
    {
        String prestr="ABD.G...CE..FH...";
        if (args.length>0)
            prestr=args[0];
        Tree2 t2=new Tree2(prestr);
        t2.preorderTraversal();
        t2.inorderTraversal();
        t2.postorderTraversal();
    }
}
```

程序运行结果如下：

```
str=ABD.G...CE..FH...
先根次序： A B D G C E F H
中根次序： D G B A E C H F
后根次序： G D B E H F C A
```

3．建立链式存储结构的完全二叉树

对于一棵顺序存储结构的完全二叉树（如图 6.7 所示），由二叉树的性质五可知，第 1 个结点为根结点，第 i 个结点的左孩子为第 $2i$ 个结点，右孩子为第 $2i+1$ 个结点。

利用性质五，我们可以将顺序存储结构的完全二叉树建成链式存储结构的完全二叉树。为简化算法，我们用字符串表示顺序存储结构的完全二叉树。

【例 6.3】 建立链式结构的完全二叉树。

Tree3 类继承 Tree1 类中的成员变量 root 和多个成员方法，增加私有成员变量 str 用以表示顺序存储的完全二叉树的结点序列。

由于 Java 字符串中的字符位置从 0 开始计数，所以取字符串中第 $i-1$ 个元素作为编号为 i 结点的元素值。程序如下：

```
import ds_java.TreeNode1;
import ds_java.Tree1;
public class Tree3 extends Tree1        //建立链式结构的完全二叉树
{
   private String str;
   public Tree3()                       //构造空二叉树
   {
      super();
   }
   public Tree3(String s)               //构造完全二叉树
   {
      str=s;
      System.out.println("str="+str);
      root=enter(1);            //以编号为 1 的元素作为根结点，建立一棵二叉树
   }
  public TreeNode1 enter(int i) //建立一棵子树，以编号为 i 的元素作为子树的根结点
   {
      TreeNode1 p=null;
      char ch;
      if(i<=str.length())
      {
         ch=str.charAt(i-1); //取字符串中第 i-1 个元素作为编号为 i 结点的元素值
          System.out.println("  i="+i+"  ch="+ch);
          p=new TreeNode1(ch+"");       //建立结点 p
          p.left=enter(2*i);            //建立 p 的左子树
          p.right=enter(2*i+1);         //建立 p 的右子树
      }
      return p;
   }
   public static void main(String args[])
```

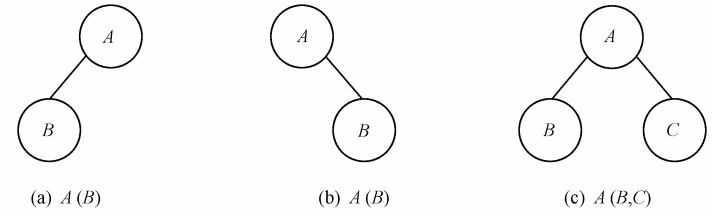
```
        {
            String varstr="ABCDEFGH";
            if(args.length>0)
               varstr=args[0];
            Tree3 t3=new Tree3(varstr);
            t3.preorderTraversal();
            t3.inorderTraversal();
            t3.postorderTraversal();
        }
    }
```

程序运行结果如下：

```
str=ABCDEFGH
  i=1  ch=A
  i=2  ch=B
  i=4  ch=D
  i=8  ch=H
  i=5  ch=E
  i=3  ch=C
  i=6  ch=F
  i=7  ch=G
先根次序： A B D H E C F G
中根次序： H D B E A F C G
后根次序： H D E B F G C A
```

4．以广义表表示建立二叉树

以广义表形式可以表示树结构，但不能惟一表示一棵二叉树，原因在于无法明确左右子树。例如，广义表 *A*(*B*)对应两棵二叉树，如图 6.13 所示。所以，以广义表形式惟一表示一棵二叉树必须重新定义。



(a) *A* (*B*)　　(b) *A* (*B*)　　(c) *A* (*B*,*C*)

图 6.13　广义表 *A*(*B*)对应两棵二叉树

如果用大写字母标识二叉树结点，则一棵二叉树可以用符合图 6.14 所示语法图的字符序列表示，“ # ”表示空子树，这种形式称为二叉树的广义表表示。

二叉树的广义表表示是递归定义的，二叉树的广义表表示如图 6.15 所示。

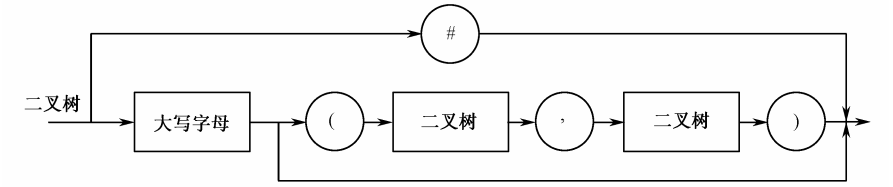


图 6.14　**广义表形式表示二叉树的语法图**

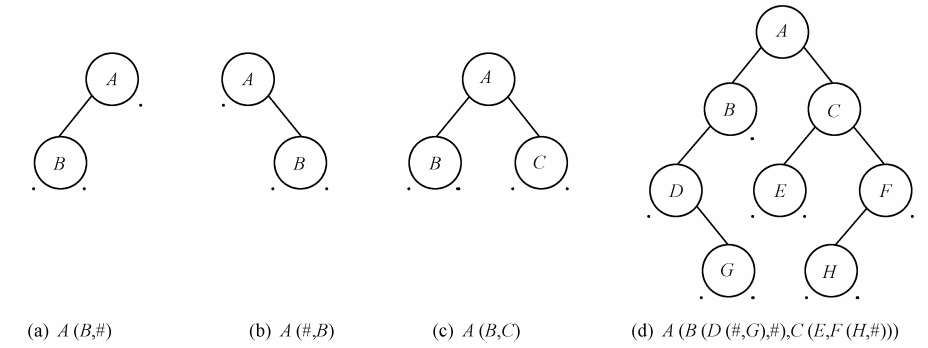


图 6.15　**二叉树的广义表表示**

反之，给定一棵二叉树的广义表表示，则能够惟一确定一棵二叉树。算法描述：依次识
二叉树的广义表表示序列中的每个字符，如果遇到“ # **”，表示空子树，返回** null **值；如果**
到大写字母，建立一个结点，则

- **如果遇到圆括号，圆括号括起的是该结点的左子树与右子树，再递归调用分别建立左**
  右子树，返回结点对象。
- **没有遇到圆括号，表示该结点是叶子结点。**

【例 6.4**】　以广义表表示建立二叉树。**

Tree4 **类继承** Tree2 **类中的成员变量** root **和多个成员方法，增加私有成员变量** str **和** i，s
存储二叉树的广义表形式，i **表示** str **的当前元素位置。**

enter( )**方法以递归算法将广义表表示的字符串** str **中的一个字符建成二叉树的一个结点。**
带参数的构造方法调用 enter( )**方法建立一棵二叉树。**

```
import ds_java.TreeNode1;
import ds_java.Tree2;
public class Tree4 extends Tree2          //以广义表表示建立二叉树
{
   private String str;
   private int i=0;
   public Tree4()                         //构造空二叉树
   {
      super();
   }
   public Tree4(String s)
   {
      str=s;
```

```
        System.out.println("str="+str);
        if(str.length()>0)
            root=enter();              //以 str 的全部元素建立一棵二叉树
    }
    public TreeNode1 enter()           //以 str 的部分元素（从 i 开始）建立一棵子
    {
        TreeNode1 p=null;
        char ch=str.charAt(i);
        if(ch>='A' && ch<='Z')                  //大写字母
        {
            p=new TreeNode1(ch+"");             //建立结点
            i++;                                //跳过大写字母
            ch=str.charAt(i);
            if(ch=='(')
            {
                i++;                            //跳过(
                p.left=enter();                 //建立左子树
                i++;                            //跳过#
                p.right=enter();                //建立右子树
                i++;                            //跳过)
            }
        }
        if(ch=='#')
            i++;                                //跳过#
        return p;                               //ch 非大写字母时，返回 null
    }
    public static void main(String args[])
    {
        String varstr="A(B(D(#,G),#),C(E,F(H,#)))";
        if(args.length>0)
            varstr=args[0];
        Tree4 t4=new Tree4(varstr);
        t4.preorderTraversal();
        t4.inorderTraversal();
        t4.postorderTraversal();
    }
}
```

程序运行结果如下：

```
str=A(B(D(#,G),#),C(E,F(H,#)))
先根次序： A B D G C E F H
中根次序： D G B A E C H F
```

后根次序： G D B E H F C A

6.3.4 二叉树遍历的非递归算法

二叉树的 3 种遍历的递归算法，通过设立一个栈，也可以用非递归算法实现。下面以中根次序遍历为例讨论二叉树的非递归遍历算法。

已知以中根次序遍历二叉树的规则是：遍历左子树，访问根结点，遍历右子树。

对于如图 6.16（a）所示的二叉树，从根结点 A 开始进行中根次序遍历，沿着指向孩子结点的两条链，可以依次访问 A、B、D 及叶子结点 G。这时，已遍历完 A 的左子树，需要再遍历 A 的右子树。但从 G 没有到达 A 的右孩子 C 的链。

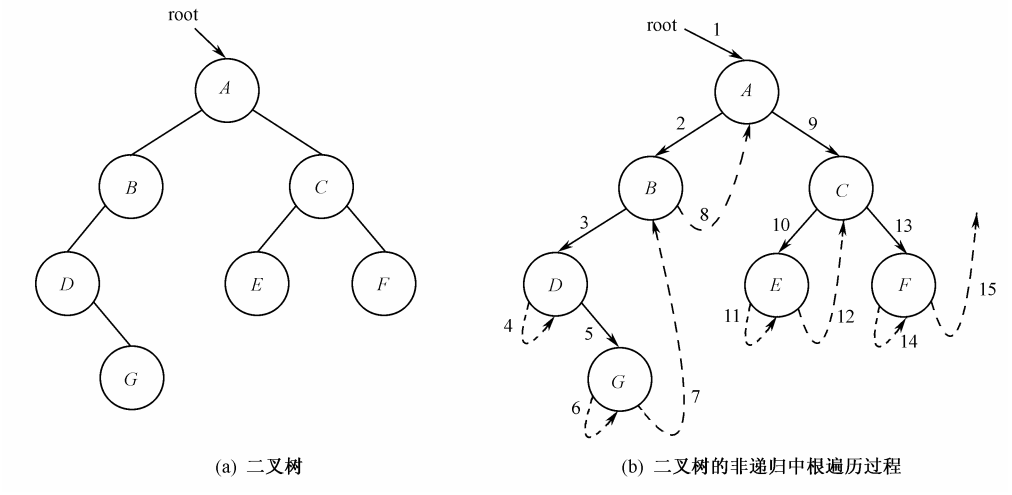
在链式存储的二叉树中，结点的两条链均是指向孩子结点的，没有指向父母结点的链。所以，从根结点开始，沿着链的方向可以走到任意一个叶子结点。而要遍历二叉树必须访问所有结点，走过所有路径。按照遍历规则，在每个结点处，先选择遍历左子树，当左子树遍历完后必须返回该结点，访问该结点后，再遍历右子树。但是二叉树中的任何结点均无法返回其父结点。这说明，二叉树本身提供的链已无法满足本题的要求，必须设立辅助的数据结构，指出下一个访问点是谁。

在从根到叶子结点的一条路径上，所经过结点的次序与返回结点的次序正好相反。如果能够依次保存路径上所经过的结点，只要按照相反次序就能找到返回的路径。因此，辅助结构应该选择具有“后进先出”特点的栈。

二叉树中根次序遍历的非递归算法描述如下：设置一个栈状态为空；从二叉树的根结点开始，如果 p 不空或栈不空时，循环执行以下操作，直到走完二叉树且栈为空状态。

- 如果 p 不空，表示刚刚到达一个结点，将结点 p 入栈，进入左子树。
- 如果 p 为空并且栈不空，表示已走出一条路径，此时必须返回寻找另一条路径。而返回的结点就是刚才经过的最后一个结点，它已保存在栈顶，所以由 p 指向出栈的结点访问结点 p，再进入 p 的右子树。

二叉树的非递归中根遍历栈状态的变化如图 6.17 所示。



(a) 二叉树　　(b) 二叉树的非递归中根遍历过程

图 6.16　二叉树的非递归中根遍历过程

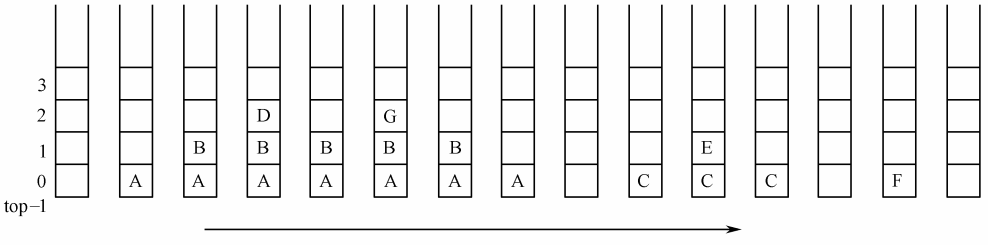


图 6.17　栈状态的变化

【例 6.5】　二叉树中根次序遍历的非递归算法。

Tree5 **类继承** Tree2 **类，以标明空子树的先根次序遍历建立一棵二叉树。**

将二叉树的中序遍历用非递归算法实现，需要设计一个栈，栈元素是二叉树的结点

TreeNode1**。本例引用第** 4 **章中设计的栈** Stack1**，元素是** Object **对象，程序中将出栈的** Obje

对象强制转换为 TreeNode1 **对象。程序如下：**

```
import ds_java.TreeNode1;
import ds_java.Tree2;
import ds_java.Stack1;
public class Tree5 extends Tree2          //二叉树中根次序遍历的非递归算法
{
   public Tree5()
   {
      super();
   }
   public Tree5(String s)                 //以标明空子树的先根次序构造二叉树
   {
      super(s);
   }
   public void inorderTraversal()         //中根次序遍历二叉树
   {
      Stack1 s1=new Stack1(20);
      TreeNode1 p=root;
      System.out.print("中根次序：  ");
      while(p!=null || !s1.isEmpty())    //p 非空或栈非空时
         if(p!=null)
         {
            s1.push(p);                  //p 结点入栈
            p=p.left;                    //进入左子树
         }
         else                            //p 为空且栈非空时
         {
            p=(TreeNode1)s1.pop();       //出栈的结点由 p 指向
            System.out.print(p.data+" ");      //访问结点
```

```
                p=p.right;                          //进入右子树
            }
        System.out.println();
    }
    public static void main(String args[])
    {
        String prestr="ABD.G...CE..F..";      //标明空子树的先根次序
        if(args.length>0)
            prestr=args[0];
        Tree5 t5=new Tree5(prestr);
        t5.inorderTraversal();
    }
}
```

程序运行结果如下：

```
str=ABD.G...CE..F..
中根次序： D G B A E C F
```

思考题：二叉树先序遍历非递归算法是怎样的？怎样修改中序遍历的非递归算法得到先序遍历非递归算法？为什么后序遍历的非递归算法不能通过修改中序遍历的非递归算法而得到？

6.3.5 层次遍历二叉树

二叉树的遍历过程，除了前述的先、中、后三种次序，还可以按层次进行，即从根结点开始，逐层深入，同层从左至右依次访问结点。如图 6.18 所示的二叉树，其层次遍历序列为 *A, B, C, D, E, F, G*。

由层次遍历结果可知，首先访问根结点 *A*，再访问根的孩子 *B* 和 *C*，然后应该访问 *B* 的孩子 *D*，再访问 *C* 的孩子 *E* 结点……

在链式存储结构的二叉树中，每个结点存在指向左右孩子的两条链。所以，从根 *A* 可以到达 *B* 和 *C*，而从 *C* 却无法到达下一层的 *D*，从 *D* 也无法到达同层的 *E*。这说明，二叉树本身提供的链已无法满足本题的要求，必须设立辅助的数据结构，指出下一个访问点是谁。

如果 *B* 在 *C* 之前访问，则 *B* 的孩子结点均在 *C* 的孩子结点之前访问。因此，辅助结构应该选择具有“先进先出”特点的队列。

按层次遍历二叉树的非递归算法描述如下：设置一个队列状态为空。从二叉树的根结点开始，当 *p* 不空时，循环顺序执行以下操作：

- 访问结点 *p*。
- 如果 *p* 的 left 链不空，将结点 *p* 的左孩子入队。
- 如果 *p* 的 right 链不空，将结点 *p* 的右孩子入队。
- 由 *p* 指向出队的结点，继续。

当队列为空时，出队方法返回 null，循环停止。

按层次遍历二叉树的队列状态的变化如图 6.19 所示。

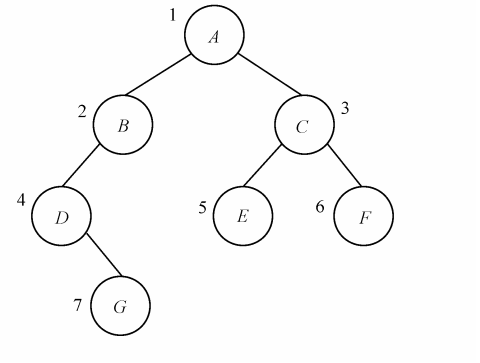


图 6.18　二叉树的层次遍历

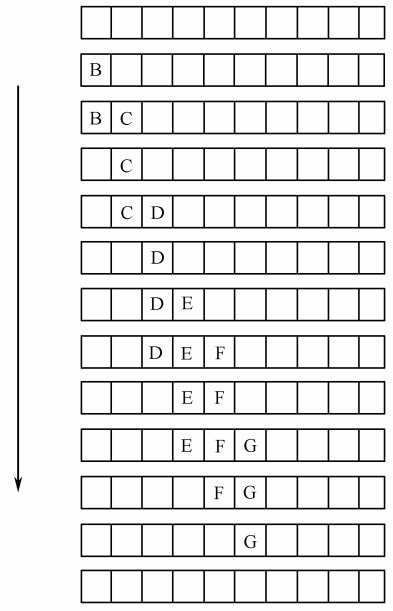


图 6.19　队列状态的变化

【例 6.6】　层次遍历二叉树。

Tree6 **类继承** Tree2 **类，以标明空子树的先根次序遍历序列建立一棵二叉树。**

实现二叉树的层次遍历，需要设计一个队列，队列元素是二叉树的结点类 TreeNode1**。本例使用第** 4 **章中设计的队列** Queue1**，元素是** Object **对象，程序中将出队的** Object **对象强制转换为** TreeNode1 **对象。程序如下：**

```
import ds_java.TreeNode1;
import ds_java.Tree2;
import ds_java.Queue1;
public class Tree6 extends Tree2          //按层次遍历二叉树
{
   public Tree6()
   {
      super();
   }
   public Tree6(String s)                 //以标明空子树的先根次序构造二叉树
   {
      super(s);
   }
   public void level()                    //按层次遍历二叉树
   {
      Queue1 q1=new Queue1(10);           //设立一个空队列
      TreeNode1 p=root;
      System.out.print("层次遍历：  ");
      while(p!=null)
```

```
        {
            System.out.print(p.data+" ");
            if(p.left!=null)
                q1.enqueue(p.left);         //p 的左孩子结点入队
            if(p.right!=null)
                q1.enqueue(p.right);        //p 的右孩子结点入队
            p=(TreeNode1)q1.dequeue();      //p 指向出队的结点
        }
        System.out.println();
    }
    public static void main(String args[])
    {
        String prestr="ABD.G...CE..F..";
        if(args.length>0)
            prestr=args[0];
        Tree6  t6=new Tree6(prestr);
        t6.level();
    }
}
```

程序运行结果如下：

```
str=ABD.G...CE..F..
层次遍历： A B C D E F G
```

6.4 线索二叉树

按照一定规则遍历二叉树得到的是二叉树结点的一种线性次序，每个结点（除第一个和最后一个外）在这些线性序列中有且只有一个前驱结点和一个后继结点。

在链式存储结构的二叉树中，每个结点很容易到达其左、右孩子结点，而不能直接到达该结点在任意一个序列下的前驱或后继结点。当需要得到结点在一种线性序列中的前驱和后继结点信息时，解决的办法有以下两种：

- **再进行一次遍历，这需要花费很多执行时间，效率很低。**
- **采用多重链表结构，即每个结点设立 5 个域，除原有的数据元素、指向左、右孩子结点的链以外，再增加 2 个分别，指向前驱结点和后继结点的链。当需要到达某结点的前驱或后继结点时，只要沿着结点的前驱链或后继链前进即可。这种办法的缺点是存储密度太低，浪费空间太多。**

实际应用中上述两种方法都不太好，而采用下面介绍的线索树结构，能够很好地解决直接访问前驱结点和后继结点的问题。

6.4.1 线索二叉树的定义

1．定义线索二叉树

在链式存储结构的二叉树中，若结点的子树为空，则指向孩子的链就为空值。因此，具有

n 个结点的二叉树，在总共 $2n$ 个链中，只需要 $n-1$ 个链来指明各结点间的关系，其余 $n+1$ 个链均为空值。如果利用这些空链来指明结点在某种遍历次序下的前驱和后继结点，就构成线索二叉树。指向前驱或后继结点的链称为线索。对二叉树以某种次序进行遍历并加上线索的过程称为线索化。按先（中、后）根次序进行线索化的二叉树称为先（中、后）序线索二叉树。

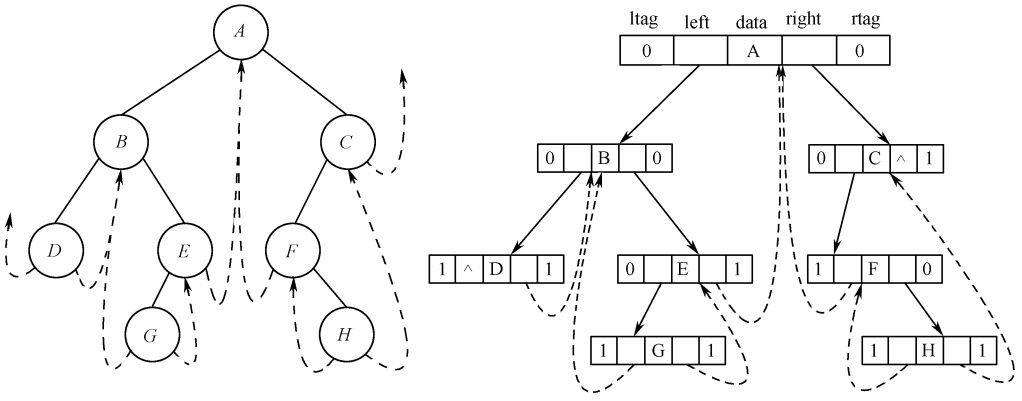
线索二叉树中，原非空的链保持不变，仍然指向该结点的左、右孩子结点。使用原先空的 left 链指向该结点的前驱结点，原先空的 right 链指向后继结点。为了区别每条链到底是指向孩子结点还是线索，需要在每个结点增加设置两个状态位 ltag 和 rtag，用来标记链的状态。ltag 与 rtag 定义如下：

$$\text{ltag}=\begin{cases}0 & \text{left指向左孩子}\\ 1 & \text{left为线索，指向前趋结点}\end{cases}$$

$$\text{rtag}=\begin{cases}0 & \text{right指向右孩子}\\ 1 & \text{right为线索，指向后继结点}\end{cases}$$

因此，每个结点就由 5 个域构成：data，left，right，ltag 和 rtag。

图 6.20 给出中序线索二叉树及其链式结构，图中的虚线表示线索。G 的前驱是 B，后继是 E。D 没有前驱，C 没有后继，相应的链为空，此时约定 ltag=1 或 rtag=1。因此，在线索二叉树中可以直接找到结点的前驱或后继结点。



(a) 中序线索二叉树　　(b) 中序线索二叉树的链式结构

图 6.20　中序线索二叉树

2．声明线索二叉树的结点类

下面声明的 ThreadTreeNode 类表示线索二叉树结点结构，它也是自引用的类，比二叉树结点结构 TreeNode1 类多了两个标记：ltag 和 rtag。

```
package ds_java;
public class ThreadTreeNode                    //线索二叉树的结点类
{
    public String data;
    public ThreadTreeNode left;
    public ThreadTreeNode right;
    public int ltag;                           //左线索标志
    public int rtag;                           //右线索标志
```

```
    public ThreadTreeNode()                 //构造空结点
    {
        this("");
    }
    public ThreadTreeNode(String d)         //构造有值结点
    {
        data=d;
        left=right=null;
        ltag=rtag=0;
    }
}
```

虽然 ThreadTreeNode **类和** TreeNode1 **类很相似，都有** data、left **和** right **成员变量，都是自引用的类。但** ThreadTreeNode **类不能继承** TreeNode1 **类，否则，继承来的成员变量** left **和** right **指向的是超类，而不是引用自己的类，将引起混淆。**

3．**声明线索二叉树类**

下面声明的 ThreadTree1 **类表示线索二叉树。**

```
package ds_java;
import ds_java.ThreadTreeNode;
public class ThreadTree1                    //中序线索二叉树
{
    protected  ThreadTreeNode root;
    public ThreadTree1()                    //构造空二叉树
    {
        root=null;
    }
    public ThreadTree1(String s)            //以标明空子树的先根次序构造二叉树，
}
```

6.4.2 中序线索二叉树

1．**二叉树的中序线索化**

对二叉树进行中序线索化的递归算法描述如下：

设 root **指向一棵二叉树的根结点，** *p* **指向某个结点，**front **指向** *p* **的前驱结点。**front **的初值为空（**null**）。** *p* **从根** root **开始，当** *p* **非空时，执行以下操作：**

- **中序线索化** *p* **结点的左子树。**
- **如果** *p* **的左子树为空，设置** *p* **的** left **链为指向前驱结点** front **的线索，**p.ltag **标记为** 1
- **如果** *p* **的右子树为空，设置前驱结点** front **的** right **链为指向** *p* **的线索，**p.rtag **标记为** 1
- **使** front **指向结点** *p*。
- **中序线索化** *p* **结点的右子树。**

中序线索化二叉树的过程如图 6.21 **所示。**

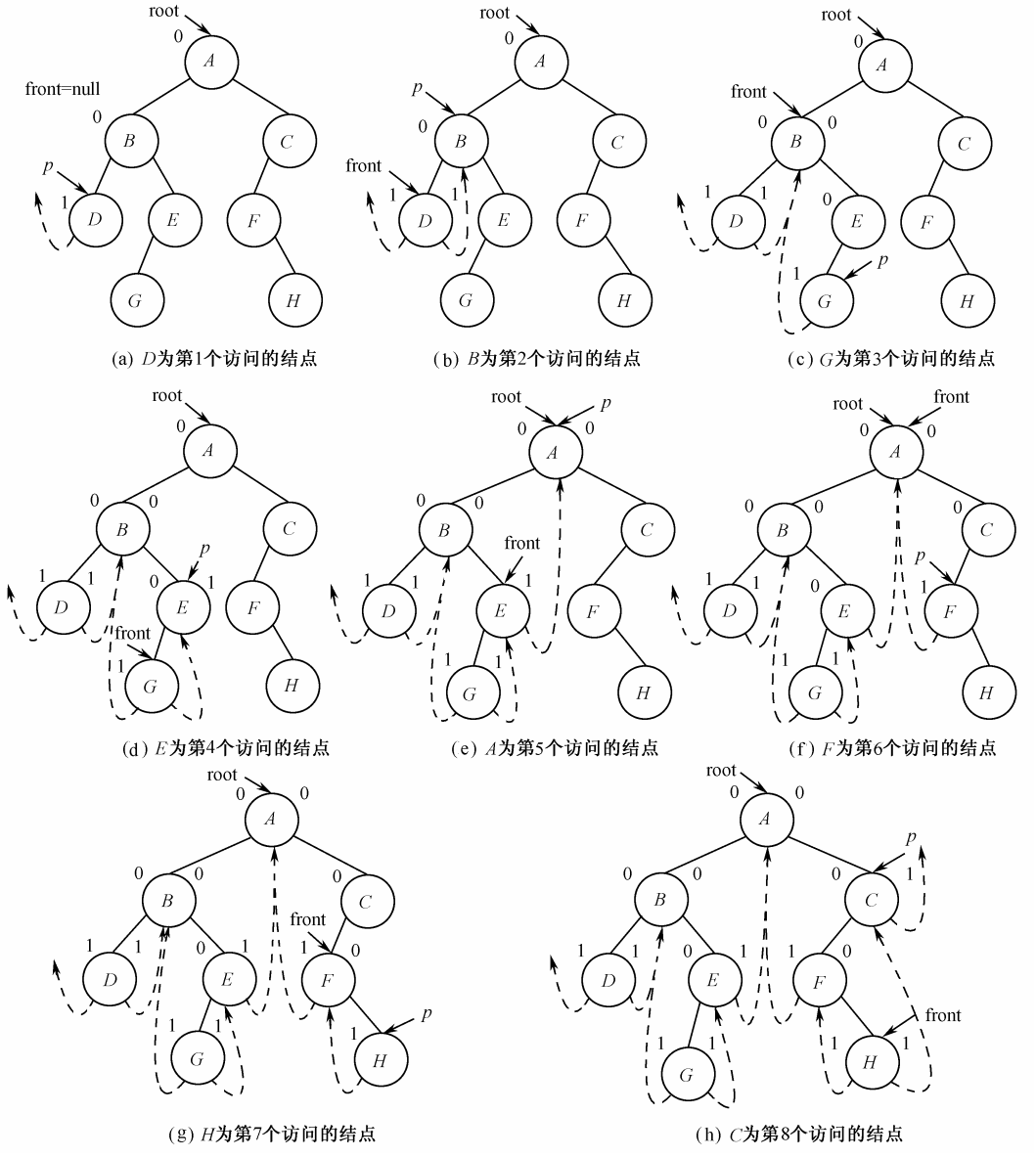


图 6.21　中序线索化二叉树的过程

在线索二叉树 ThreadTree1 **类中，增加以下方法对二叉树进行中序线索化。**

```
protected  ThreadTreeNode front=null;//中序下的前驱结点的引用
public void inThreadTreeNode(ThreadTreeNode p)
{                                          //中序线索化以 p 结点为根的子树
   if(p!=null)
   {
      inThreadTreeNode(p.left);            //中序线索化 p 的左子树
      if(p.left==null)                     //p 的左子树为空时
      {                                    //设置 p.left 为指向 front 的线索
```

```
            p.ltag=1;
            p.left=front;
        }
        if(p.right==null)                    //p 的右子树为空时
            p.rtag=1;                        //设置 p.right 为线索的标志
        if(front!=null && front.rtag==1)
            front.right=p;                   //设置front.right为指向p的线

        if(front!=null)                      //显示中间结果
            instr=instr+"front="+front.data+"\t";
        else
            instr=instr+"front=null";
        instr=instr+"  p="+p.data+"\r\n";
        front=p;
        inThreadTreeNode(p.right);           //中序线索化 p 的右子树
    }
}
public void inorderThread()                  //中序线索化二叉树
{
    front=null;
    inThreadTreeNode(root);
}
```

2．中根次序遍历中序线索二叉树

在中序线索二叉树中，容易查找到某结点在中根次序下的前驱或后继结点，而不必遍历整棵二叉树。

1）查找中根次序下的前驱或后继结点

已知按中根次序遍历二叉树的规则：遍历左子树，访问根结点，遍历右子树。

下面以图 6.21 的中序线索二叉树为例，说明查找中根次序下的前驱或后继结点的过程。

查找前驱结点的过程描述如下：

- 如果结点的左子树为空（如 *G*），则 *G* 的 left 链指向其前驱结点（*B*）。
- 如果结点的左子树非空（如 *A*），则 *A* 的前驱结点是 *A* 左子树上最后一个中序访问的结点（*E*）。或者说，*E* 是 *A* 左孩子 *B* 的最右边的子孙结点。

查找后继结点的过程描述如下：

- 如果结点的右子树为空（如 *G*），则 *G* 的 right 链指向 *G* 的后继结点（*E*）。
- 如果结点的右子树非空（如 *A*），则 *A* 的后继结点是 *A* 右子树上第一个中序访问的结点（*F*）。或者说，*F* 是 *A* 右孩子 *C* 的最左边的子孙结点。

在线索二叉树 ThreadTree1 类中，增加 innext( )方法，以查找结点 *p* 在中根次序下的后继结点。

```
public ThreadTreeNode innext(ThreadTreeNode p)
{                                            //返回中根次序下的后继结点
```

```
        if(p.rtag==1)                    //右子树为空时
          p=p.right;                     //right 就是指向后继结点的线索
        else                             //右子树非空时
        {
          p=p.right;                     //进入右子树
          while(p.ltag==0)               //找到最左边的子孙结点
            p=p.left;
        }
        return p;
    }
```

2）中根次序遍历

以中根次序遍历中序线索二叉树的非递归算法描述如下：

- 寻找第一个访问结点，它是根（*A*）的左子树上（*B*）最左边的子孙结点（*D*），由指向 *D*。
- 访问 *p* 结点，之后再找到 *p* 的后继结点。
- 重复执行上一步，就可以遍历整棵二叉树。

在线索二叉树 ThreadTree1 类中，增加 inorderTraversal( )方法，调用 innext( )方法在中序线索二叉树中实现中根次序遍历。

```
    public void inorderTraversal()       //中根次序遍历中序线索二叉树
    {
        ThreadTreeNode p=root;
        if(p!=null)
        {
          System.out.print("中根次序： ");
          while(p.ltag==0)
              p=p.left;                  //找到根的最左边子孙结点
           do
           {
              System.out.print(p.data+" ");
              p=innext(p);               //返回 p 的后继结点
           } while(p!=null);
           System.out.println();
        }
    }
```

【例 6.7】 中序线索二叉树的线索化与中序遍历。

本程序调用 ThreadTree1 类演示中序线索二叉树的线索化与中序遍历。

```
    import ds_java.ThreadTree1;
    public class ThreadTree1_ex                  //中序线索二叉树
    {
        public static void main(String args[])
        {
```

```
        String prestr="ABD..EG...CF.H...";
        if (args.length>0)
           prestr=args[0];
        ThreadTree1 t1=new ThreadTree1(prestr);  //建立二叉树
        t1.inorderThread();                      //中序线索化二叉树
        t1.inorderTraversal();                   //中根次序遍历中序线索二叉树
     }
  }
```

程序运行结果如下：

```
str=ABD..EG...CF.H...
```

中序线索化：

```
front=null  p=D
front=D    p=B
front=B    p=G
front=G    p=E
front=E    p=A
front=A    p=F
front=F    p=H
front=H    p=C
```

中根次序： D B G E A F H C

思考题 本例将建立一棵中序线索二叉树的功能分两步进行 :先建一棵链式存储的二叉树
再线索化。请实现另一种建立中序线索二叉树的方法：在建立链式二叉树的同时进行线索化

在中序线索二叉树中，不但可以很方便地查找中根次序下的前驱和后继结点，还可以查找
先根、后根次序下的前驱和后继结点。因此，以先根和后根次序能够遍历中序线索二叉树。

3．**先根次序遍历中序线索二叉树**

已知按先根次序遍历二叉树的规则：访问根结点，遍历左子树，遍历右子树。

下面以图 6.21 **的中序线索二叉树为例，说明查找先根次序下的后继结点及先根次序遍历**
的过程。

1）**查找先根次序下的后继结点**

在中序线索二叉树中，查找先根次序下后继结点的过程描述如下：

- **如果结点的左子树非空（如** *A*），**则** *A* **的左孩子** *B* **即为** *A* **的后继结点。**
- **如果结点的左子树为空且右子树非空（如** *F*），**则** *F* **的右孩子** *H* **即为** *F* **的后继结点。**
- **如果结点的左、右子树均为空（叶子结点），则后继结点为它右边的兄弟（如** *D* **的后**
  继为 *E*），**或为它所在子树的右边兄弟（如** *G* **是以** *B* **为根的子树上最后一个访问的结**
  点， *B* **的右边兄弟** *C* **即是** *G* **的后继结点）。通过** right **线索（** *G* **的** right **和** *E* **的** right）
  可以找到 *G* **的某个祖先结点** *A*，**再找到** *A* **的右孩子** *C*。

上述算法实现如下：

```
public ThreadTreeNode prenext(ThreadTreeNode p)
{                                          //返回先根次序下的后继结点
```

```
        if(p.ltag==0)                          //左子树非空时
          p=p.left;                            //左孩子就是 p 的后继结点
        else
        {
          if(p.rtag==0)                        //左子树为空而右子树非空时
            p=p.right;                         //右孩子是 p 的后继
          else
          {
            while(p.rtag==1 && p.right!=null)  //叶子结点
               p=p.right;                      //后继是其某个中序祖先的右孩子
            p=p.right;
          }
        }
        return p;
    }
```

2）先根次序遍历

以先根次序遍历中序线索二叉树的非递归算法描述如下：

- 寻找第一个访问结点，它是根（A），由 p 指向结点 A。
- 访问结点 p，之后再找到 p 的后继结点。
- 重复执行上一步，就可以遍历整棵二叉树。

上述算法实现如下：

```
    public void preorderTraversal()          //先根次序遍历中序线索二叉树
    {
        ThreadTreeNode p=root;
        if(p!=null)
        {
          System.out.print("先根次序：  ");
          do
          {
             System.out.print(p.data+" ");
                                                  p=prenext(p);
                                              }while(p!=null);
                                              System.out.println();
                                          }
                                      }
```

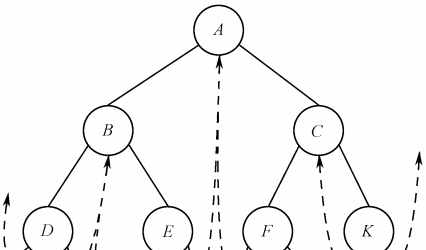


图 6.22　一棵中序线索二叉树

4．后根次序遍历中序线索二叉树

下面以图 6.22 的中序线索二叉树为例，说明查找后根次序下的前驱结点及后根次序遍历的过程。

1）查找后根次序下的前驱结点

已知按后根次序遍历二叉树的规则：遍历左子树，遍历右子树，访问根结点。

在中序线索二叉树中，查找后根次序下前驱结点的过程描述如下：

- 如果结点的右子树非空（如 A），则 A 的右孩子 C 即为 A 的前驱结点。
- 如果结点的右子树为空且左子树非空（如 E），则 E 的左孩子 G 即为 E 的前驱结点。
- 如果结点的左、右子树均为空（叶子结点），则前驱结点为它左边的兄弟（如 K 的前驱为 F），或为它所在子树的左边兄弟（如 H 是以 C 为根的子树上第一个访问的结点，C 的左边兄弟 B 即是 H 的前驱结点）。通过 left 线索（H 的 left 和 F 的 left），可以找到 H 的某个祖先结点 A，再找到 A 的左孩子 B。

2）后根次序遍历

以后根次序遍历中序线索二叉树的非递归算法描述如下：

- 首先由 p 指向根结点（A）。
- 访问结点 p，之后再找到 p 的前驱结点。
- 重复上一步，就可以遍历整棵二叉树。所得到的序列与后根遍历次序正好相反。

思考题：编写后根次序遍历中序线索二叉树的方法。

6.5 堆排序

作为完全二叉树的应用，本节讨论堆排序（heap sort）算法。

在每次选择最小值时，如果能利用以前的比较结果，就可提高排序速度。堆排序就是根据这个思想提出的一种排序的方法。

1．堆的定义

堆是一个关键字序列$\{k_1, k_2, \cdots, k_n\}$，它具有如下特性：

$k_i \leq k_{2i}$ 且 $k_i \leq k_{2i+1}$（$i=1, 2, \cdots, n/2$）

根据二叉树的性质五，对于完全二叉树中的第 i 个结点，其左孩子为第 $2i$ 个结点，右孩子为第 $2i+1$ 个结点。因此，可以将堆序列理解成具有下列特性的一棵完全二叉树结点的层次序列：二叉树中，任意一个结点的关键字的值都小于或等于它的孩子结点的关键字的值。

由此可知，根结点具有最小值。图 6.23 是一棵具有堆性质的完全二叉树。

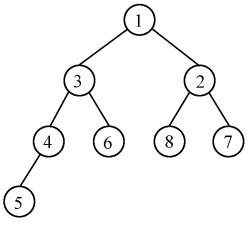


图 6.23　具有堆性质的完全二叉树

2．堆排序算法描述

堆排序的思路是：用筛选法将一个待排序的数据序列建成堆，得到最小值——根结点的值，将根结点值向后交换，再将余下的值调整成堆，重复进行，直到排序完成。

设有一个待排序的数据序列，其关键字序列为{5，3，7，4，6，8，2，1}，堆排序的算法描述如下：

- 以顺序存储结构的线性表存储待排序的数据序列，设数据元素个数 $n=8$，如图 6.24（a）所示。
- 将数据序列看成一棵完全二叉树结点的层次序列，此时它不具有堆的特性，如图 6.24（b）所示。
- 将这棵完全二叉树调整成堆。从其中最深的一棵子树开始，该子树的根结点是堆序

的第 i 个元素（$i=n/2$）。

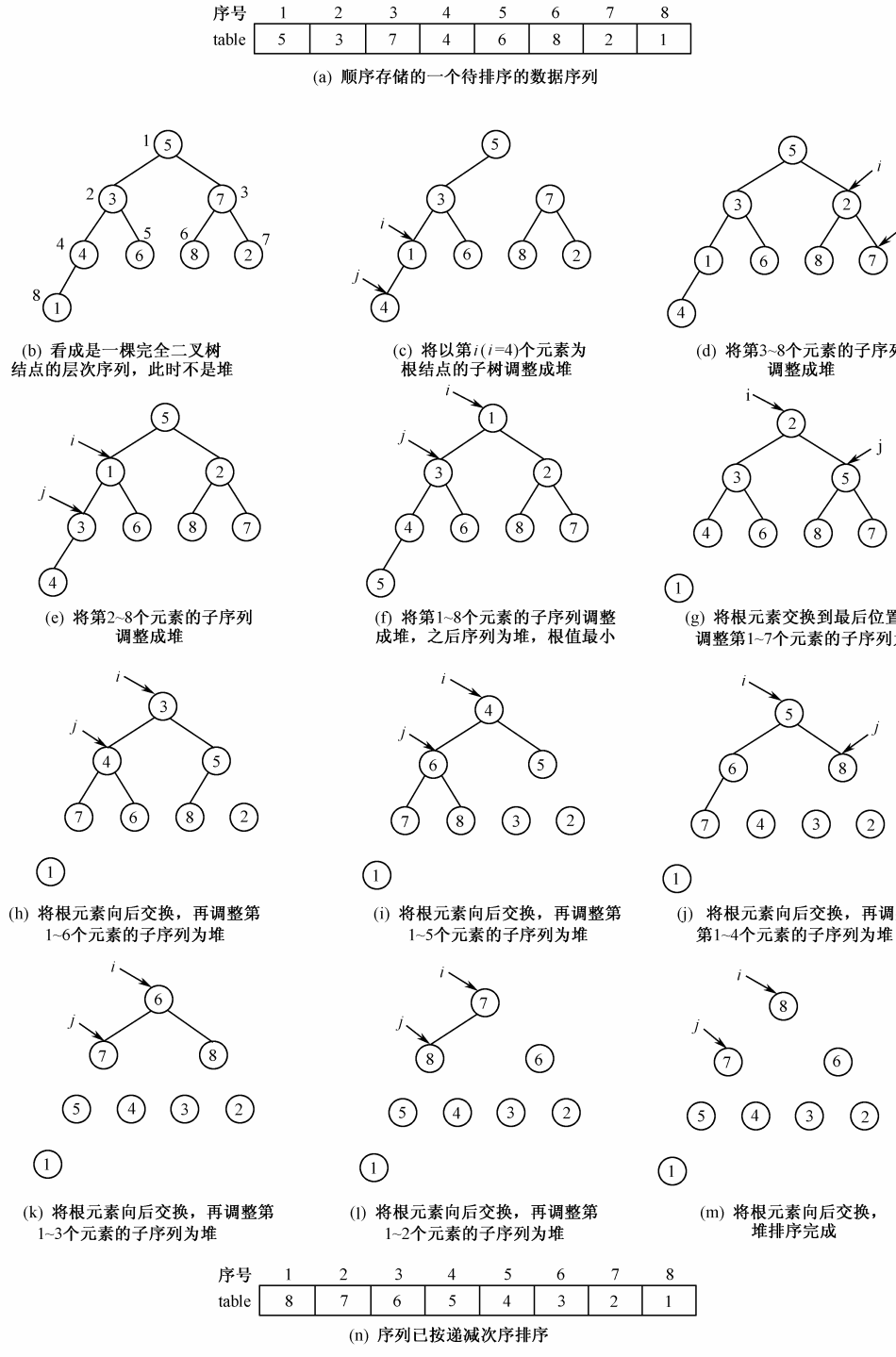


图 6.24　堆排序过程

- **将一棵子树调整成堆，在第 i 个元素的两个孩子结点（左孩子为第 $2i$ 个结点，右孩子为第 $2i+1$ 个结点）中，选择较小的值（j 表示其序号）作为根值。再将 i 降低一个层**

次，与孩子结点（j）的值比较、交换，直到叶子结点。这样从每棵子树的角度看，值最小的元素成为根结点，就像筛子，把值较大的元素沉在下面，因而建成堆后根结点值最小，称为筛选法，如图 6.24（c）所示。

- 重复上一步，从下向上依次将子树调整成堆，如图 6.24（d）~（f）所示。对于一棵子树而言，如果根值较大，需要将它向下滑动，如图 6.24（f）中的 5。当以序列的第 1 个元素为根结点的子树被调整后，这棵完全二叉树结点的层次序列就成为一个堆序列，如图 6.24（f）所示。
- 这时根结点的值最小，将根结点值交换到最后位置，再将序列的前 n-1 个元素调整成堆，如图 6.24（g）~（m）所示。
- 重复上一步，将子序列调整成堆，直到子序列长度为 1，堆排序完成。最终数据序列排列为降序，如图 6.24（n）所示。

3．堆排序算法实现

【例 6.8】 堆排序算法的实现与测试。

设待排序的数据序列已存储在顺序存储结构的线性表（LinearList1 类）中，sift( )方法用筛选法将以 left 和 right 为左右边界的子序列调整成堆。

heapsort( )方法实现堆排序。它先调用 sift( )方法 $n/2$ 次，使待排序的数据序列成为堆。再调用 sift( )方法 n-1 次，每次将值最小的根（i=1）元素交换到第 n-i+1 个位置，将前 n-1 个元素调整成堆。依此重复进行，直到排序完成。

```
import ds_java.LinearList1;                    //顺序存储结构的线性表类
public class HeapSort1 extends LinearList1     //堆排序
{
   public HeapSort1(int table[])               //以数组构造线性表
   {
      super(table);
   }
   public HeapSort1(int n)                     //构造具有 n 个元素的线性表
   {
      super(n);
   }
   public void sift(int left,int right)
   {                              //将以第 left 个元素为根结点的子树调整成堆
      int i,j,x;
      i=left;
      j=2*i;                                   //第 j 个元素是第 i 个元素的左孩子
      x=this.get(i);                           //获得第 i 个元素的值
      while(j<=right)
      {
          if(j<right && this.get(j)>this.get(j+1))
             j++;                              //如果右孩子值较小时,j 表示右孩子
          if(x>this.get(j))                    //第 i 个元素值较大
```

```
            {
                this.set(i,this.get(j));      //设置第 i 个元素为 j 的值
                i=j;                          //i,j 向下滑动一层
                j=2*i;
            }
            else
                j=right+1;
        }
        this.set(i,x);
        System.out.print("sift  "+left+".."+right+"  ");
        this.output();
    }
    public void heapsort()
    {
        int j,n=this.length();
        for(j=n/2;j>=1;j--)              //建堆
            sift(j,n);
        for(j=n;j>1;j--)                 //调整成堆
        {
            swap(1,j);                   //根(最小)值交换到后面
            sift(1,j-1);
        }
    }
    public static void main(String args[])
    {
        int table[]={5,3,7,4,6,8,2,1};
        int n=table.length;
        HeapSort1 s1=new HeapSort1(table);
        System.out.print("n="+n+"   ");
        s1.output();
        s1.heapsort();
    }
}
```

程序运行结果如下：

```
n=8   table:  5 3 7 4 6 8 2 1
sift  4..8  table:  5 3 7 1 6 8 2 4
sift  3..8  table:  5 3 2 1 6 8 7 4
sift  2..8  table:  5 1 2 3 6 8 7 4
sift  1..8  table:  1 3 2 4 6 8 7 5
sift  1..7  table:  2 3 5 4 6 8 7 1
sift  1..6  table:  3 4 5 7 6 8 2 1
```

```
sift  1..5  table:  4 6 5 7 8 3 2 1
sift  1..4  table:  5 6 8 7 4 3 2 1
sift  1..3  table:  6 7 8 5 4 3 2 1
sift  1..2  table:  7 8 6 5 4 3 2 1
sift  1..1  table:  8 7 6 5 4 3 2 1
```

思考题：将堆定义为具有如下特性的一个关键字序列（k_1，k_2，…，k_n）：

$$k_i \geq k_{2i} \text{ 且 } k_i \geq k_{2i+1} \qquad (i=1，2，\cdots，n/2)$$

实现按递增次序的堆排序算法。

堆排序算法的时间复杂度为 $O(n\log_2 n)$。堆排序算法是不稳定的。

6.6 树与二叉树的转换

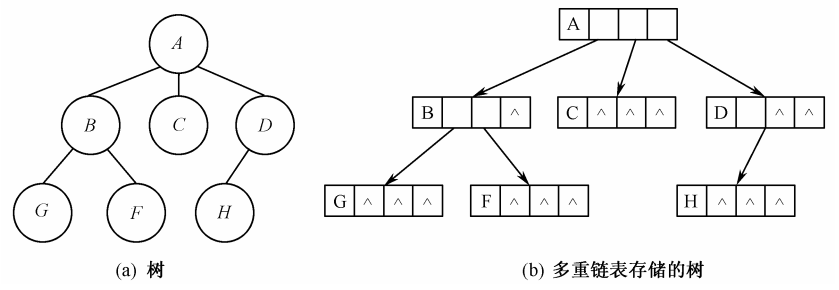
树和二叉树是两种不同的数据结构，树实现起来比较麻烦，二叉树实现起来比较容易。树可以转换为二叉树进行处理，处理完以后再从二叉树还原为树。

1．树转化为二叉树

如果用链式结构来存储具有 n 个结点的度为 k 的树，则每个结点需用 k 个链指向孩子结点，这种存储方式称为多重链表结构（如图 6.25 所示），总的链数为 $n\times k$，其中只有 $n-1$ 个非空的链指向除根以外的 $n-1$ 个结点，其余的链皆为空。空链与链总数之比为：

$$\frac{(n\times k)-(n-1)}{n\times k} \approx 1-\frac{1}{k}$$

当 k=10 时，则空链占 90%。由此可见，这样的结构有时非常浪费存储空间。



(a) 树　　(b) 多重链表存储的树

图 6.25　树的多重链表存储方式

通常将树（或森林）转化成二叉树形式来存储。设二叉树的结点有 3 个域：

- data 存放结点数据。
- child 指向该结点的第一个孩子结点。
- brother 指向该结点的下一个兄弟结点。

上述树的存储结构称为树的孩子兄弟存储结构。显然，树的孩子兄弟存储结构将一棵树转换成了一棵二叉树。由此可将一棵树按孩子兄弟存储结构存储。对于给定的一棵树，必定有惟一的一棵二叉树与之相对应。由于树的根结点没有兄弟，所以对应二叉树的根结点的右子树为空。图 6.26（a）是一棵树及其对应的二叉树。

森林也可转化成一棵二叉树的形式存储，转化过程如下：

- 将每棵树转化成二叉树。

- 由根的 brother 链将若干棵树连接成一棵二叉树。图 6.26（b）是森林及其对应的二叉树。

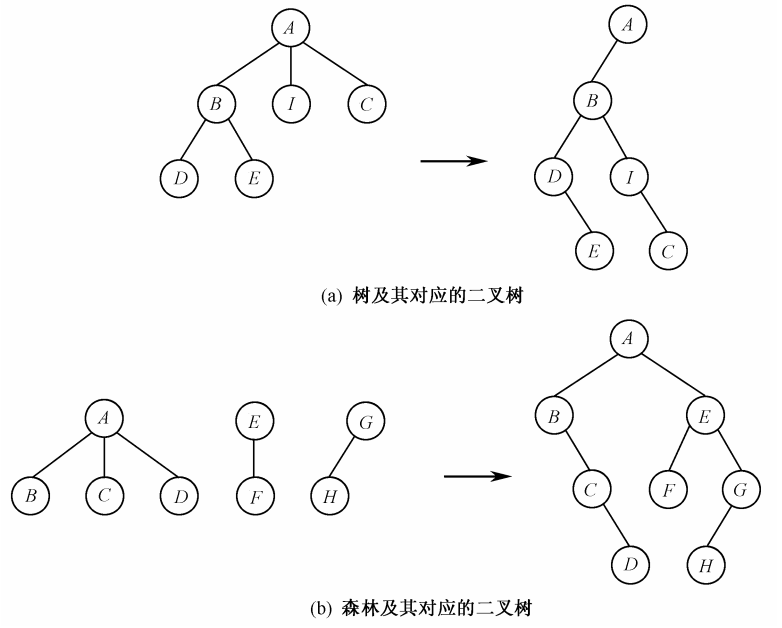


图 6.26　树和森林及其对应的二叉树

2．二叉树还原为树

二叉树还原为树的方法是：

- 若某结点是其双亲结点的左孩子，则把该结点的右孩子、右孩子的右孩子……都与该结点的双亲结点用线连起来。
- 删除原二叉树中所有双亲结点与右孩子结点的连线。
- 整理所有保留的和添加的连线，使每个结点的所有孩子结点位于相同层次高度。

习　题　6

6.1　二叉树是否就是度为 2 的树？为什么？

6.2　什么样的二叉树，它的先根与后根次序相同？

6.3　先根遍历序列为 *ABCD* 的二叉树有多少种形态？

6.4　对于如图 6.27 所示的二叉树，求先、中、后三种次序的遍历序列。

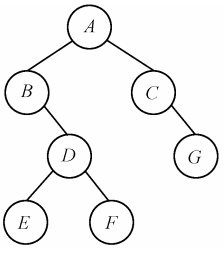


图 6.27　二叉树

6.5 给定一棵链式存储的二叉树，完成下列功能，并比较是否可用先、中、后三种次序的遍历算法：

（1）输出二叉树的叶子结点。

（2）统计二叉树的结点个数。

（3）验证二叉树的性质二：$n_0=n_2+1$。

（4）找出二叉树中值大于 k 的结点。

（5）将二叉树中所有结点的左右子树相互交换。

（6）求一棵二叉树的高度。

（7）求一棵二叉树的每个结点所在的层次。

（8）复制一棵二叉树。

（9）判断一棵二叉树是否为完全二叉树。

（10）实现二叉树后序遍历的非递归算法。

6.6 有关线索二叉树：

（1）对图 6.27 所示的二叉树，分别以先、中、后序进行线索化。

（2）直接建立线索二叉树，即在建树的同时进行线索化。

（3）编写后根次序遍历中序线索二叉树的方法。

6.7 设一棵二叉树以三重链表为其结点结构，如图 6.28 所示。其中，data 为表示数据元素的成员变量，left, right 和 parent 分别为指向左、右孩子结点及父母结点的成员变量。完成以下功能：

（1）建立一棵三重链表结构的二叉树。

（2）求该二叉树中某结点的所有祖先。

Left	data	parent	right

图 6.28 三重链表结构的二叉树

6.8 判断以下序列是否为堆，若不是，则调整为堆。

（1）{100，86，48，73，35，39，42，57，66，21}

（2）{12，70，33，65，24，56，48，92，86，33}

（3）{103，97，56，38，66，23，42，12，30，06}

（4）{05，56，20，23，40，38，29，61，35，76}

6.9 用堆排序方法对关键码序列{82，73，71，23，94，16，05，68}进行排序，画出排序过程的示意图

6.10 编程判断一个数据序列是否为堆序列。

实 习 6

1．实验目的：掌握二叉树的定义、性质、存储结构、遍历原理与实现方法。

2．题意：

在一棵二叉树中，求两结点最近的共同祖先结点。

3．实验要求：

建立链式存储的二叉树，并研究以下多种算法：

- 递归算法。
- 设立栈。
- 建立线索二叉树。
- 建立三重链表结构的二叉树。

第 7 章　查　　找

查找（searching）也称检索，是在数据结构中寻找满足某种给定条件的数据元素的过程。查找是数据处理中经常使用的一种重要运算，也是程序设计中的一项重要的基本技术。生活中经常用到查找，如在电话簿中查找电话号码，在字典中查找单词等。

本章介绍查找的基本概念，讨论多种经典查找技术，包括线性表的顺序、折半和分块查找算法，二叉排序树的查找算法以及哈希表的查找算法，并讨论各种算法适用于哪些数据存储结构，以及比较各算法的运行效率。

建议本章授课 8 学时，实验 4 学时。

7.1　查找的基本概念

1．查找表与关键字

查找表（search table）是同一种数据类型的数据元素的有限集合。关键字是数据元素中用于识别该元素的某个域，能够惟一识别数据元素的关键字称为主关键字。

2．查找操作与查找结果

查找操作是在查找表中，根据给定的某个值 k，确定关键字与 k 相同的数据元素的过程。

查找的结果有两种：查找成功与查找不成功。如果在查找表中，存在关键字与 k 相同的数据元素，则查找成功；否则，查找不成功。

表 7-1　电话簿

姓名	电话号码
王清	1234567
李放	7654321
朱小红	1236789
张明	3567856

例如，表 7-1 所示的电话簿是一个查找表，以姓名为关键字，设 k="李放"，从第 1 个数据元素开始依次比较，第 2 个数据元素的姓名与 k 相同，则查找成功；再设 k="李明"，电话簿中任何一个数据元素的姓名都不等于 k，则查找不成功。

3．查找表所具有的数据结构

一个如表 7-1 所示的电话簿，可以看成是一个顺序存储结构的线性表。此外，还可以在树型结构中查找。例如，Windows 的文件系统以树型结构组织和存储文件，文件夹是树中的非叶结点，文件是叶子结点。所以，Windows 的文件系统是一个树型结构的查找表，提供了以

文件名、日期等为关键字查找文件的功能，如图 7.1 所示。

Windows 在给定范围内查找相关文件。如果查找成功，给出文件路径、长度、日期等信息；如果查找不成功，给出“找到 0 个文件”的信息。

图 7.1　查找文件

4．查找方法

查找的方法有很多，不同的数据结构及条件可以采用不同的查找算法，以求快速高效地得到查找结果。

对于数据量较小的线性表，可以采用顺序查找算法。例如，从电话簿的第 1 个数据元素开始，依次将数据元素的关键字与 k 比较，进行查找操作。

当数据量较大时，采用分块查找算法。例如，在字典中查找单词 state，顺序查找算法是可实现的，但速度慢、效率低。由于字典是按字母顺序排列的，只要在以字母 s 开头的范围内，再根据 state 后几个字母的顺序，就可快速准确地定位 state，查阅 state 的含义。

5．查找算法的性能评价

衡量查找算法效率的最主要标准是平均查找长度。平均查找长度是指查找过程所需进行的关键字比较次数的平均值。平均查找长度通常记为 ASL：

$$\mathrm{ASL}=\sum_{i=1}^{n} p_i \times c_i$$

其中，p_i 是要查找数据元素的出现概率，c_i 是查找相应数据元素需进行的关键字比较次数。由于 p_i 很难通过分析给出，通常考虑概率相等的情况，即设 $p_i=1/n$。

查找成功和查找不成功的平均查找长度通常不同，分别用 $\mathrm{ASL}_{成功}$ 和 $\mathrm{ASL}_{不成功}$ 表示。

6．静态查找表与动态查找表

对查找表进行的操作有：

- 查询某个特定数据元素是否在查找表中。
- 检索查找表中某个特定数据元素的属性。
- 在查找表中插入一个数据元素。
- 从查找表中删除一个数据元素。

如果不需要对一个查找表进行插入、删除操作，则该查找表称为静态查找表（static searc

table）；反之，称为动态查找表（dynamic search table）。例如，字典是一个静态查找表，而电话簿则是一个动态查找表。

7．查找技术的实现

查找算法的性能直接依赖于数据结构和查找算法。因此，要实现一种查找技术，必须完成以下多方面的操作：

- 建立数据结构。
- 设计查找算法。
- 维护数据结构，实现插入、删除等操作。

7.2 线性表的查找

针对线性表的查找技术有 3 种：顺序查找，折半查找和分块查找。

7.2.1 顺序查找

顺序查找（sequential search）又称线性查找（linear search），是一种最简单的查找算法。对于给定值 k，从线性表的一端开始，依次比较每个数据元素的关键字，直到查找成功，或到达线性表的另一端，则查找不成功。

1．顺序表的顺序查找

设数据元素保存在顺序表中，对于一个给定值 k，顺序查找算法描述如下：

- 设 i=1，比较第 i 个数据元素的关键字是否等于 k，如果相等，则查找成功，查找过程结束；否则 i++，继续比较。
- 线性表中所有数据元素的关键字都不等于 k，则查找不成功。

顺序表的顺序查找过程如图 7.2 所示。

图 7.2 顺序存储线性表的顺序查找过程

【例 7.1】顺序表的顺序查找算法实现与测试。

本例定义 Search1 类继承顺序存储结构的线性表 LinearList1 类，继承超类的 table 数组以及 length( )、add( )和 output( )等方法。构造方法将含在含有 n 个数据元素的线性表中加满随机数；search( )方法查找 k 值，查找成功时返回 true，否则返回 false。

```
import ds_java.LinearList1;                    //线性表的顺序存储结构
public class Search1 extends LinearList1       //顺序查找
{
```

```
    public Search1(int n)                          //构造具有 n 个存储单元的线性表
    {
        super(n);
        create();                                  //线性表中加满随机数
    }
    public Search1()
    {
        this(0);
    }
    public boolean search(int k)                   //查找 k 值，查找成功时返回 true
    {                                              //不成功时返回 false。
        int j=1;
        boolean find=false;
        while(!find && j<=this.length())
        {
            System.out.print(this.get(j)+"="+k+"?  ");
            if(this.get(j)==k)                     //判断第 j 个元素值是否为 k
                find=true;
            else
                j++;
        }
        return find;
    }
    public static void main(String args[])
    {
        int n=8,k=20;
        Search1 s1=new Search1(n);
        s1.output();
        System.out.println("\r\nsearch("+k+")="+s1.search(k));
    }
}
```

程序运行结果：

```
table:  21 54 84 59 1 25 75 58
21=20?  54=20?  84=20?  59=20?  1=20?  25=20?  75=20?  58=20?
search(20)=false
```

2．单向链表中查找结点

在已建立的单向链表（如图 7.3 所示）中，对于给定值 k，采用顺序查找算法，查找关键字值为 k 的结点。查找成功时，返回该结点地址；否则返回空值（null）。

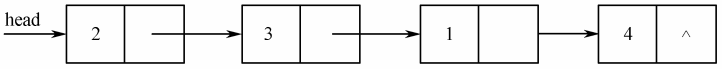


图 7.3　单向链表

下面是单向链表的顺序查找算法，search( )方法返回单向链表的结点类 OnelinkNode。

```
public OnelinkNode search(int k)          //查找 k 值，返回 k 值所在的结点
{                                         //查找不成功时返回 null
   OnelinkNode p=head;
   while(p!=null && k!=p.data)
      p=p.next;
   return p;
}
```

3．算法分析

如果线性表中第 i 个元素的关键字等于 k，进行 i 次比较即可找到该元素。设线性表中元素的个数为 n，查找第 i 个元素的概率为 p_i，在等概率情况下，$p_i=1/n$，对于成功的查找，其关键字的平均查找长度 $ASL_{成功}$ 为

$$ASL_{成功}=\sum_{i=1}^{n}p_i\times c_i=\frac{1}{n}\sum_{i=1}^{n}i=\frac{1}{n}\times\frac{n(n+1)}{2}=\frac{n+1}{2}$$

对于不成功的查找，其关键字的平均查找长度为 $ASL_{不成功}$ 为

$$ASL_{不成功}=\sum_{i=1}^{n}p_i\times c_i=\sum_{i=1}^{n}\frac{1}{n}\times n=n$$

由此可知，在等概率情况下，查找成功的平均查找长度为线性表长度的一半，查找不成功的平均查找长度为线性表的长度。所以，顺序查找算法的时间复杂度为 $O(n)$。

7.2.2　折半查找

折半查找（binary search）也称二分法查找，适用于顺序存储结构并且数据元素已经按照关键字大小排序的线性表。

1．折半查找算法

1）基本思想

假定线性表的数据元素是按照升序排列的，对于给定值 k，从线性表的中间位置开始比较，如果当前数据元素的关键字等于 k，则查找成功。否则，若 k 小于当前数据元素的关键字，则在线性表的前半部继续查找；反之，则在线性表的后半部继续查找。依此重复进行，直至获得查找成功或不成功的结果。

例如，对于如下数据元素的关键字序列：

{1，4，7，10，13，16，19，22}

设查找 k=13，left 和 right 为查找范围的左、右边界，将线性表的第 mid 个数据元素与 13 进行比较，mid 在由 left 和 right 标记范围的中间位置：

mid =(left + right)/ 2

折半查找算法描述如下：

- 设 left=1，right=8，此时线性表长度为 8，取中间位置的数据元素 mid=4，比较 *k* 与 table[mid]，如图 7.4（a）所示。
- 因 *k*>table[mid]，故只需要在线性表后半部继续比较，查找范围缩小为 left=mid＋1=5，right=8，则 mid=6，如图 7.4（b）所示。
- 再比较 *k* 与 table[mid]，有 *k*<table[mid]，说明只需要在线性表前半部继续比较，查找范围缩小为 left=5，right=mid-1=5，则 mid=5，如图 7.4（c）所示。此时，若 *k*=table[mid]，则查找成功，mid 为查找到的数据元素的序号；否则查找不成功。

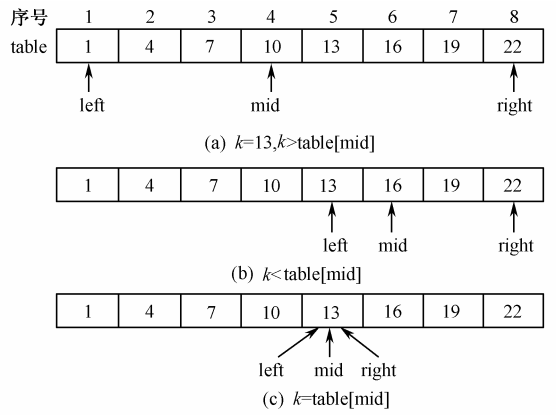


图 7.4　折半查找算法描述

2）算法实现

```
public boolean halfsearch(int k)     //查找 k 值,找到时返回 true
{                                    //不成功时返回 false。
   int left,right,mid;
   boolean find=false;
   left=1;
   right=this.length();              //获得线性表长度
   while(left<=right && !find)       //子序列边界有效
   {
      mid=(left+right)/2;            //子序列的中间位置
     System.out.print("left="+left+"  right="+right+"  mid="+mid+" ");
      System.out.println(this.get(mid)+"=="+k+"? "+(k==this.get(mid)));
      if(k==this.get(mid))
         find=true;                  //查找成功
      else
         if(k<this.get(mid))
            right=mid-1;             //子序列为原序列的前半部
         else
            left=mid+1;              //子序列为原序列的后半部
   }
   return find;
}
```

3）算法分析

折半查找的过程可以用一棵二叉判定树表示，设线性表的长度 n=8，如图 7.5 所示。

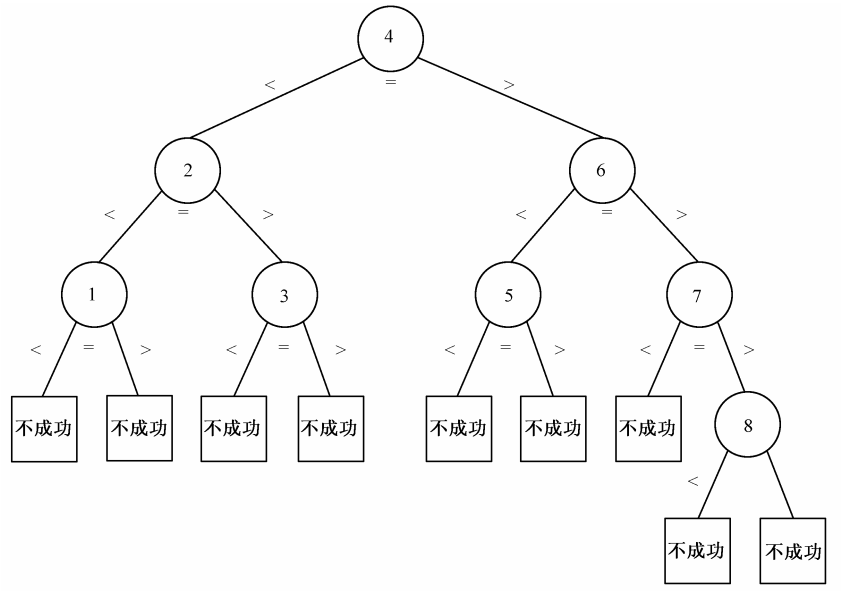


图 7.5　折半查找过程的二叉判定树

结点中的数字值表示线性表中数据元素的序号。二叉判定树反映了折半查找过程中，进行关键字比较的数据元素次序。当 n=8 时，线性表的边界为 1～8，第一次 k 与第 4 个数据元素比较，若 k 值较小，再与第 2 个数据元素比较，否则与第 6 个数据元素比较，以此继续。

设二叉判定树的高度为 h，则

$$2^h-1<n\leq 2^{h+1}-1$$

一次成功的查找恰好走过一条从根结点到某结点的路径。比较次数最少为 1 次，最多为 $\lceil\log_2(n+1)\rceil$，平均情况为 $O(\log_2(n+1))$。

不成功的查找路径则是从根结点到叶子结点，比较次数至少为 $\lfloor\log_2(n+1)\rfloor$，最多为树的高度 $\lceil\log_2(n+1)\rceil$，平均情况为 $O(\log_2(n+1))$。平均效率要比顺序查找算法的效率要高。

折半查找算法每比较一次，如果查找成功，算法结束；否则，将查找的范围缩小了一半，而顺序查找算法在每一次比较后，只将查找范围缩小了一个数据元素。因此，折半查找算法的平均效率要比顺序查找算法的高。

2．折半插入排序

在顺序表的直接插入排序算法中，当需要插入一个数据元素 k 时，首先使用顺序查找算法在已排序的顺序表中查找数据元素 k，以确定插入位置，此时的顺序表正好符合折半查找算法的条件。因此，直接插入排序算法中，用折半查找算法代替顺序查找算法，构成折半插入排序的算法。

例如，对于一个待排序的数据序列，其关键字如下：

{5，3，2，4，7，1}

折半插入排序算法描述如下：

（1）首先将第 1 个数据元素 k=5 插入，作为初始的已排序的子序列{5}。

（2）欲插入数据元素值 k=3，在已排序的子序列{5}中进行折半查找，找到 k 值应插入的位置 i=1。

（3）将从第 i 个数据元素开始到序列尾的子序列{5}依次向后移动，移动次序是从序列尾开始到第 i 个数据元素，空出第 i 个数据元素的位置。

（4）将 k 值插入第 i 数据元素的位置。插入 3 之后的子序列为{3，5}。

（5）重复步骤 2～4，依次将数据元素插入子序列中。

折半插入排序算法描述如图 7.6 所示。

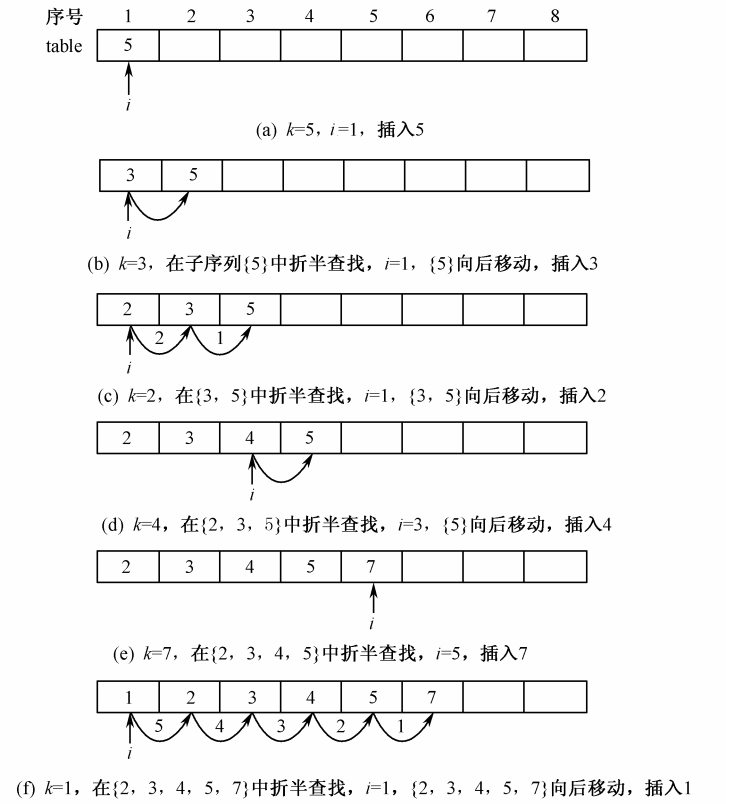


图 7.6　折半插入排序算法

【例 7.2】 折半插入排序。

本例声明的 HalfInsertSort1 类继承直接插入排序的 InsertSort1 类，用折半查找算法覆盖基类中的 search 顺序查找算法。

```
public class HalfInsertSort1 extends InsertSort1   //折半插入排序
{
    public HalfInsertSort1(int n)          //构造具有 n 个存储单元的线性表
    {
        super(n);                          //n 个随机数插入排序
    }
    public int search(int k)               //折半查找 k 值
    {                                      //不成功时返回 k 值应在的位置
        int left=1,right,mid=0;
```

```
        right=this.length();
        while(left<=right)                    //子序列边界有效
        {
            mid=(left+right)/2;
            if(k<this.get(mid))
                right=mid-1;
            else
                left=mid+1;
        }
        if(mid==0 || k>=this.get(mid))
            mid++;           //当 k 比第 mid 个元素大时,插入在下一位置
        return mid;
    }
    public static void main(String args[])
    {
        HalfInsertSort1 s1=new HalfInsertSort1(8);
    }
}
```

编译时将 InsertSort1.class 文件存放在当前文件夹中。程序运行结果如下：

```
k=22  i=1  table:  22 0 0 0 0 0 0 0
k=80  i=2  table:  22 80 0 0 0 0 0 0
k=33  i=2  table:  22 33 80 0 0 0 0 0
k=78  i=3  table:  22 33 78 80 0 0 0 0
k=4   i=1  table:  4 22 33 78 80 0 0 0
k=52  i=4  table:  4 22 33 52 78 80 0 0
k=95  i=7  table:  4 22 33 52 78 80 95 0
k=90  i=7  table:  4 22 33 52 78 80 90 95
```

折半插入排序算法的时间复杂度为 $O(n^2)$。它比直接插入排序减少了比较次数，但平均移动次数仍为 $n^2/4$。

7.2.3 分块查找

前面介绍了顺序查找和折半查找算法，顺序查找算法简单，对原始数据序列无排序要求，适用于顺序存储结构和链式存储结构；折半查找算法虽然减少了查找次数，速度较快，但条件严格，要求数据序列是顺序存储结构的并且已排序，而对数据序列进行排序也是要花费一定代价的。因此，当数据量较小时，这两种查找算法可行；当数据量较大时，查找速度慢、效率低，此时，采用分块查找算法更合适。

1．分块查找算法的基本思想

分块查找（blocking search）将数据序列中的数据元素存储在若干个数据块中，每一个数据块中的数据元素是顺序存放的，没有排序，但多个数据块之间必须按照数据元素的关键字大小排序，这种分块特性称为“块间有序、块内无序”。假定不同的数据块按照数据元素的递增

次序排列，也就是说，在第一块中任意一个数据元素的关键字都小于第二块中所有数据元素的关键字。为了记载数据块之间的次序，必须另外再设计一张索引表（index）。

2．静态查找表的分块查找

字典是我们经常使用的一种工具，在字典中使用最多的操作就是查找。除此之外，不需要进行诸如插入、删除等操作。所以，字典是一种静态查找表。

静态查找表只要能够存储数据元素即可，所以静态查找表可以采用顺序存储结构。

对于字典这样数据量较大的查找表，使用顺序查找算法进行查找操作显然不合适，而适宜采用分块查找技术。为使查找方便，字典中的关键字通常是按照一定规则排好序的，并且在正式文本之前，都有一个索引表指明关键字所在的页码。

下面我们以 Java 语言的关键字为例，说明如何在字典中实现分块查找算法。

设将 Java 的关键字排序后存放在数组 table 中，并为关键字表设计一个索引表 index，index 的每个数据元素由两部分组成：首字母和下标，它们分别对应于关键字的首字母和其所在 table 数组中的下标，如图 7.7 所示。

索引表index

	首字母	下标
0	a	0
1	b	1
2	c	4
3	d	9
4	e	12
5	f	14
6	i	19
7	l	25
8	n	26
9	p	29
10	r	33
11	s	34
12	t	39
13	v	45
14	w	46

Java关键字表table

	下标	关键字
0	0	abstract
1	1	boolean
2	2	break
3	3	byte
4	4	case
5	5	catch
6	6	char
7	7	class
8	8	continue
9	9	default
10	10	do
11	11	double
12	12	else
13	13	extends
14	14	false
15	15	final
16	16	finally
17	17	float
18	18	for
19	19	if
20	20	implements
21	21	import
22	22	instanceof
23	23	int
24	24	interface
25	25	long
26	26	native
27	27	new
28	28	null
29	29	package
30	30	private
31	31	protected
32	32	public
33	33	return
34	34	short
35	35	static
36	36	super
37	37	switch
38	38	synchonized
39	39	this
40	40	throw
41	41	throws
42	42	transient
43	43	true
44	44	try
45	45	void
46	46	while

图 7.7　Java 关键字表与索引表

通过索引表 index，将较长的关键字表 table 划分成若干个数据块，以首字母相同的若干关键字构成一个数据块，因而每个数据块的大小不等，每块的起始下标由 index 中对应“首字母”列的“下标”标明。例如，以’a’为首字母的关键字块，起始下标为 0；以’b’为首字母的关键字块，起始下标为 1；以’c’为首字母的关键字块，起始下标为 4，等等。

给定一个单词 str，如果能在关键字表 table 中查找成功，则说明 str 是关键字，否则 str 不是关键字。

使用分块查找算法，在关键字表 table 中查找给定的单词 str，必须分两步进行：

- 根据 str 的首字母，查找索引表 index，确定 str 应该在 table 中的哪一个数据块。
- 在相应数据块中，使用顺序或折半查找算法查找 str，得到查找成功与否的信息。

【例 7.3】　判断单词是否为 Java 的关键字。

本例声明 IsKey 类实现字典的分块查找算法。其中，table 数组存储已排序的 Java 关键字

index 是关键字表的索引表，其元素由“首字母”和“下标”两部分组成。

由于 Java 中没有类似 C 中的结构体（struct），程序中将 index 的元素类型设计成一个内部类 IndexNode，它有两个成员 first 和 subscript，它们分别对应关键字的首字母和其所在 table 中的下标。

halfsearch(ch)方法在 index 中查找字母 ch，查找成功时返回 ch 所在元素的下标，否则返回 -1。由于 index 中的元素是以“首字母”为关键字排序的，所以，此时可以使用折半查找算法。

search(str, left, right)方法在 table 中查找 str，left 和 right 分别表示待查找数据块的第一个元素和最后一个元素的下标。查找成功时返回 true，否则返回 false。

isKeyword(str)方法判断 str 是否为 Java 的关键字。以 str 的首字母 ch 为参数，调用 halfsearch(ch)方法，获得在 index 中元素的下标 *i*，index[i].subscript 是 str 可能在的那一个数据块的起始下标，index[i＋1].subscript - 1 是该块最后一个元素的下标。再调用 search(str, left, right)方法，在该块中使用顺序查找算法查找 str。程序如下：

```
public class IsKey
{
   String table[]={"abstract","boolean","break","byte","case",
                   "catch","char","class","continue","default",
                   "do","double","else","extends","false","final",
                   "finally","float","for","if","implements",
                   "import","instanceof","int","interface","long",
                   "native","new","null","package","private",
                   "protected","public","return","short","static",
                   "super","switch","synchronized","this","throw",
                   "throws","transient","true","try","void","while"};
   IndexNode index[]={new IndexNode('a',0),new IndexNode('b',1),
                      new IndexNode('c',4),new IndexNode('d',9),
                      new IndexNode('e',12),new IndexNode('f',14),
                      new IndexNode('i',19),new IndexNode('l',25),
                      new IndexNode('n',26),new IndexNode('p',29),
                      new IndexNode('r',33),new IndexNode('s',34),
                      new IndexNode('t',39),new IndexNode('v',45),
                      new IndexNode('w',46)};
   class IndexNode                          //内部类
   {
      char first;                           //关键字的首字母
      int subscript;                        //关键字所在 table 中的下标
      IndexNode(char ch,int i)
      {
         first=ch;
         subscript=i;
```

```
    }
}
public boolean isKeyword(String str)    //判断 str 是否为 Java 的关键字
{                                       //是则返回 true,否则返回 false
    char ch=str.charAt(0);
    int i=halfsearch(ch);               //查找 str 所在的块
    System.out.println("halfsearch("+ch+")="+i);
    int left=index[i].subscript;        //确定一块的位置
    int right=index[i+1].subscript-1;
    return search(str,left,right);      //在一块中查找 str
}
public int halfsearch(char ch)          //在 index 中查找 ch,折半查找
{
    int left=0,right,mid=0;
    right=index.length-1;
    mid=(left+right)/2;
    while(left<=right && ch!=index[mid].first)
    {                                   //子序列边界有效且未找到时
        System.out.print(ch+"=="+index[mid].first+"?\t");
        if(ch<index[mid].first)
            right=mid-1;
        else
            left=mid+1;
        mid=(left+right)/2;
    }
    System.out.println();
    if(left<=right)
        return mid;                     //查找成功时,返回元素的下标
    else
        return-1;                       //查找不成功时,返回-1
}
public boolean search(String str,int left,int right) //查找 str
{                                       //left、right 为块的下标边界
    int i=left;
    boolean find=false;
    while(!find && i<=right)
       {
        System.out.print(str+"=="+table[i]+"?\t");
       if(str.equals(table[i]))    //判断两个字符串 str 与 table[i]是否相等
            find=true;
```

```
            else
                i++;
        }
        return find;
    }
    public static void main(String args[])
    {
        String str="index";
        System.out.println("\r\nisKeyword("+str+")="
                                    +(new IsKey()).isKeyword(str));
    }
    }
```

程序运行结果：

```
   i==l?i==d?   i==f?
   halfsearch(i)=6
   index==if?   index==implements?  index==import?  index==instanceof?
index==int? index==interface?
   isKeyword(index)=false
```

3．**动态查找表的分块查找**

与字典不同，一本电话簿在使用过程中，经常需要增加或删除数据元素。所以，电话簿是一种动态查找表。

动态查找表的存储结构必须能够适应动态地进行插入或删除数据元素的操作。

如果以顺序存储结构保存电话簿的数据元素，则进行插入、删除操作时必须移动大量的数据元素，运行效率较低。如果以链式存储结构保存电话簿的数据元素，虽然插入、删除操作较方便，但花费的空间较多，查找的效率较低。那么，怎样才能既最大限度地利用空间，又有很高的运行效率呢？

下面我们以顺序存储结构和链式存储结构相结合的方式来存储数据元素，如图 7.8 **所示**

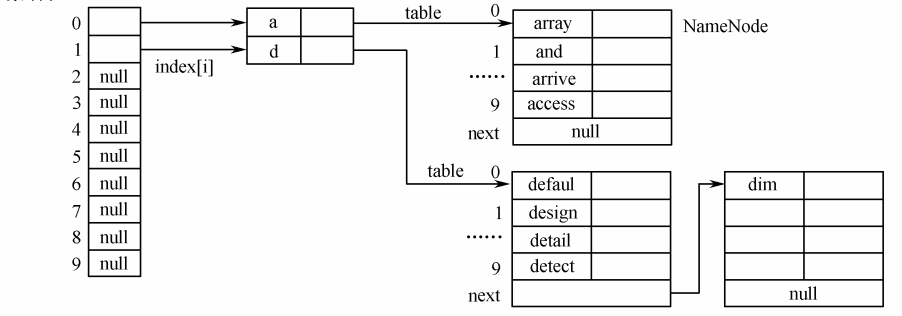


图 7.8　**电话簿中数据元素的分块存储结构**

设 NameNode **表示一个人的信息，包括姓名** name **和电话号码** telephone。BlockNode **表**

一个块，包括保存个人信息的数组 table、指向下一块的链 next。IndexNode 表示索引表中的一个元素，包括姓名的首字母 first、指向块的链 head。

索引表 index 的元素 index[i]保存由首字母相同的若干人所组成的若干块信息，每块的大小相等，如果一块的容量不够时，再申请一块，以链表方式将若干块连接起来。

【例 7.4】 动态查找表的分块查找。

为简化问题，设每个数据元素仅由关键字构成。对于如下的数据序列：

{94，8，17，97，2，92，1，9，10，0}

本例采用如图 7.9 所示的分块存储结构。本例声明的 BlockSearch1 类实现分块查找算法。

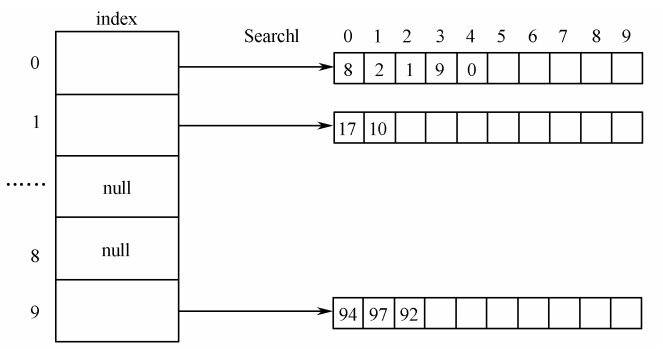


图 7.9 动态查找表的分块存储结构

我们利用在例 7.1 中声明的 Search1 类的一个对象表示一个顺序存储结构的数据块，其中的 search( )方法实现顺序查找算法，add(k)方法添加 *k* 值。

index 数组是各数据块的索引表，元素 index[i]指向一个数据块。设每个数据块可保存 10 个数据元素，约定 index[i]指向的数据块存储从 $i\times10$ 到 $(i+1)\times10-1$ 的数据元素。例如，index[0]指向的数据块范围为 0~9，index[1]指向的数据块范围为 10~19。

index 数组元素的初值为 null。根据实际需要，系统动态生成每个数据块，将对数据块的引用保存在 index[i]中。

insert(k)方法将 *k* 值添加到某个数据块中。首先根据 *k* 值，确定应该将 *k* 添加到由 index[k/10]指向的数据块中，再调用 Search1 类的 add(k)方法将 *k* 值添加到相应数据块中。

构造方法创建索引表 index，调用 insert( )方法依次将产生的 *n* 个随机数添加到相应数据块中。

output( )方法输出 index 数组及全部数据块的数据元素值。

blocksearch(k)方法实现分块查找算法。首先根据分块规则，*k* 应该在由 index[k/10]指向的数据块中，再调用 Search1 类的顺序查找方法 search(k)在相应数据块中求得查找结果。

```
public class BlockSearch1                    //分块查找
{
   protected Search1 index[];                //各数据块的索引表
   int size;
   public BlockSearch1(int n)                //创建 index,并添加 n 个随机数
   {
      index=new Search1[n];
      size=n;
```

```
        int i,j,k;
        System.out.print("k=");
        for(i=0;i<size;i++)
        {
            k=(int)(Math.random()*100);
            System.out.print(k+", ");
            insert(k);
        }
        System.out.println();
    }
    public void insert(int k)            //将 k 值添加到某个数据块中
    {
        int i,j;
        i=k/size;                        //确定 k 值应在第 i 个数据块
        if(index[i]==null)
            index[i]=new Search1(size);
        index[i].add(k);                 //在第 i 个数据块中添加 k 值
    }
    public boolean blocksearch(int k)    //查找 k 值，成功时返回 true
    {                                    //不成功时返回 false。
        int i,j;
        i=k/size;
        boolean find=false;
        if(index[i]!=null)
        {
            System.out.print("find k="+k+"  index["+i+"]\t");
            find=index[i].search(k);
        }
        return find;
    }
    public void output()             //输出索引表及全部数据块
    {
        int i;
        for(i=0;i<index.length;i++)
        {
            System.out.print("index["+i+"]");
            if(index[i]==null)
                System.out.println("=null");
            else
            {
                System.out.print("->");
                index[i].output();
```

```
            }
        }
        System.out.println();
    }
    public static void main(String args[])
    {
        int n=10,k=20;
        BlockSearch1 s1=new BlockSearch1(n);
        s1.output();
        System.out.println("\r\nblocksearch("+k+")="+s1.blocksearch(k));
    }
}
```

编译时将 Search1.class **文件存放在当前文件夹中。程序运行结果如下：**

```
k=34, 53, 86, 8, 56, 9, 47, 66, 51, 23,
index[0] ->  table:  8 9 0 0 0 0 0 0 0 0
index[1] = null
index[2] ->  table:  23 0 0 0 0 0 0 0 0 0
index[3] ->  table:  34 0 0 0 0 0 0 0 0 0
index[4] ->  table:  47 0 0 0 0 0 0 0 0 0
index[5] ->  table:  53 56 51 0 0 0 0 0 0 0
index[6] ->  table:  66 0 0 0 0 0 0 0 0 0
index[7] = null
index[8] ->  table:  86 0 0 0 0 0 0 0 0 0
index[9] = null

find k=20  index[2]  23=20?
blocksearch(20)=false
```

本例所采用的分块存储结构，将数据元素按其值分别存储在若干块较小的存储空间内，得查找范围缩小，提高了查找效率。

7.3 二叉排序树及其查找算法

本节以二叉排序树为例，介绍树结构的查找算法。

1．二叉排序树的定义

二叉排序树又称二叉查找树，它可以是一棵空树，若非空时具有下述性质：

- **若根结点的左子树非空，则左子树上所有结点的关键字值均小于等于根结点的关键字值**
- **若根结点的右子树非空，则右子树上所有结点的关键字值均大于等于根结点的关键字值**
- **根结点的左、右子树也是二叉排序树。**

显然，二叉排序树的中根遍历序列是按升序排列的。例如，以关键字序列

{5，8，3，2，4，7，9，1，5}

建立的一棵二叉排序树如图 7.10 所示，它的中根遍历序列是

{1，2，3，4，5，5，7，8，9}

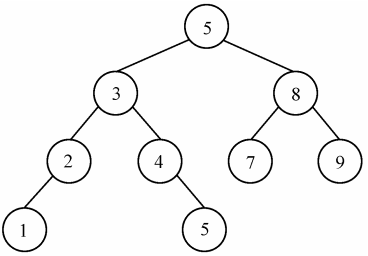


图 7.10　二叉排序树

二叉排序树与堆是不同的。二叉排序树是二叉树结构，通常采用链式存储结构，父母结点的关键字值比左孩子结点值大，而比右孩子结点值小。而堆序列是顺序存储结构的线性表，只看成是完全二叉树结点的层次序列，堆序列的性质强调对应完全二叉树中父母结点值比子女结点值小，而不考虑左右孩子值间的大小。

2．二叉排序树的建立

以关键字序列

{5，8，3，2，4，7，9，1，5}

为例，建立一棵二叉排序树的算法描述如下：

- 以序列的第 1 个值作为根结点的关键字，建立一棵二叉排序树。
- 将序列值 k 与根结点的关键字进行比较，如果 k 值较小，则将 k 插入根结点的左子树中；否则，将 k 插入根结点的右子树中。到达空子树时，以 k 为关键字建立新结点并链入二叉排序树。
- 重复上一步，将序列值依次插入到二叉排序树中。

建立一棵二叉排序树的过程如图 7.11 所示。由此可知，每次新插入的结点都是叶子结点，这样就不会破坏二叉树原有的形态。

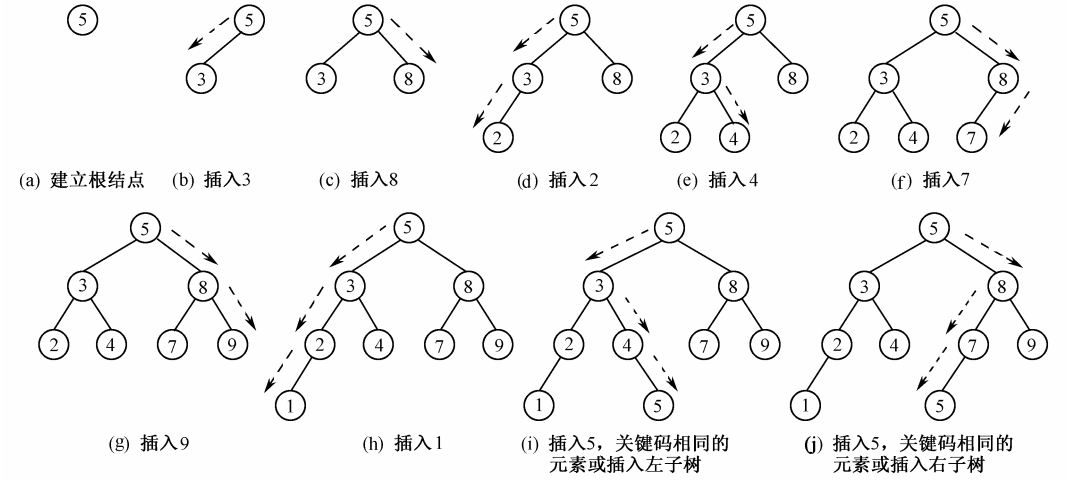


图 7.11　建立二叉排序树

【例 7.5】 建立二叉排序树的递归算法。

Tree7 类继承二叉树类 Tree1，结点仍是 TreeNode1 类。

Insert( )方法将 *d* 值插入到以 root 为根结点的二叉树中。若 root 为空，建立一个结点作为根结点；否则，将 *d* 值插入到根结点的子树中。

Insertnode( )方法将 *d* 值插入到以 *p* 为根的子树中。根据 *d* 值的大小，插入到根 *p* 的左子树或右子树中。若 *p* 的左（右）子树为空，则建立一个叶子结点作为 *p* 的左（右）孩子。

```
import ds_java.TreeNode1;
import ds_java.Tree1;
public class Tree7 extends Tree1          //建立二叉排序树，递归
{
   public Tree7()                         //构造空二叉树
   {
      super();
   }
   public Tree7(String str)               //以 str 字符串建立二叉排序树
   {
      this();
      int i;
      String ch ;
      System.out.print("建立二叉排序树： ");
      for(i=0;i<str.length();i++)
      {
         ch=str.substring(i,i+1);
         System.out.print(ch+" ");
         insert(ch);
      }
      System.out.println();
   }
   public void insert(String d)           //插入结点
   {
      if(root==null)
         root=new TreeNode1(d);           //建立根结点
      else
         insertnode(d,root);              //将 d 插入到以 root 为根结点的子树中
   }
   public void insertnode(String d,TreeNode1 p)
   {                          //将 d 插入到以 root 为根结点的子树中，递归
      if(d.compareTo(p.data)<=0)
      {
         if(p.left==null)
            p.left=new TreeNode1(d);      //建立叶子结点作为 p 的左孩子
```

```
            else
               insertnode(d,p.left);        //将 d 插入到 p 的左子树中
         }
         else
         {
            if(p.right==null)
               p.right=new TreeNode1(d);    //建立叶子结点作为 p 的右孩子
            else
               insertnode(d,p.right);       //将 d 插入到 p 的右子树中
         }
      }
      public static void main(String args[])
      {
         String varstr="583247915";
         if(args.length>0)
            varstr=args[0];
         Tree7 t7=new Tree7(varstr);
         t7.inorderTraversal();
      }
   }
```

程序运行结果如下：

```
建立二叉排序树： 5 8 3 2 4 7 9 1 5
中根次序： 1 2 3 4 5 5 7 8 9
```

建立二叉排序树的序列值也可由参数 args[0]**得到。**

3．在二叉排序树中查找给定值

在一棵二叉排序树中，查找给定值 *k* **的算法描述如下：**

- **设** *p* **指向根结点（**p=root**）。**
- **将** *k* **与** *p* **指向结点的关键字进行比较，若两者相等，则查找成功；若** *k* **值较小，则进入** *p* **的左子树继续查找；若** *k* **值较大，则进入** *p* **的右子树继续查找，直到查找成功或** *p* **为空。** *p* **为空时，表示查找不成功。**

查找算法如下：

```
   public boolean search(String d)          //查找结点
   {
      TreeNode1 p=root;
      System.out.print("find("+d+")=  ");
      while(p!=null && !d.equals(p.data))
      {                                     //比较两个字符串是否相等
        System.out.print(p.data+" ");
        if(d.compareTo(p.data)<0)           //比较两个字符串的大小
          p=p.left;                         //进入左子树
```

```
            else
               p=p.right;                          //进入右子树
         }
         if(p!=null)
            return true;                           //查找成功
         else
            return false;                          //查找不成功
      }
```

一次查找成功的路径是，从根结点到该结点所经过的结点序列。例如，在图 7.10 的二叉排序树中查找 8，比较的结点序列是{5，8}。而查找不成功的路径，是从根结点到某个叶子结点的结点序列。如查找 6，必须在访问过 5，8 和 7 之后才能肯定查找不成功，此时已比较过叶子结点 7。

对于一般二叉树而言，查找一个结点需要遍历一棵二叉树，查找过程就是遍历的过程。但在二叉排序树查找中，进行查找的比较序列只是一条路径，而非整棵树，不需要遍历整棵树，这时使用循环沿着一条路径就可以实现查找，不需要递归算法。

4．在二叉排序树中插入结点的非递归过程

在二叉排序树中插入一个结点，首先需要找到该结点应该插入的位置，这是一个查找问题，通常情况下是一次不成功的查找，用非递归算法可以实现。虽然是一次不成功的查找，查找结果会找到结点应该插入的位置，这正是所希望的。

插入结点的非递归算法实现如下：

```
   public void insertnode2(String d)   //插入结点的非递归算法
   {
      TreeNode1 p,q=null;
      if(root==null)
         root=new TreeNode1(d);         //建立根结点
      else
      {
         p=root;
         while(p!=null)
         {
            q=p;
            if(d.compareTo(p.data)<=0)
               p=p.left;                //进入左子树
            else
               p=p.right;               //进入右子树
         }
      p=new TreeNode1(d);               //建立叶子结点 p
      if(d.compareTo(p.data)<=0)
         q.left=p;                      //p 作为 q 的左孩子
      else
```

```
            q.right=p;                          //p 作为 q 的右孩子
        }
    }
```

7.4 哈希查找

哈希是 hash 的音译，意为“杂凑”，也称散列。哈希表是一种重要的存储方式，哈希查找技术是一种按关键字编址的检索方法。

为了在查找表中查找某个数据元素，用顺序、折半或分块查找算法，都必须经过一系列关键字比较后，才能确定数据元素在查找表中的位置，或者给出查找不成功的信息。因此，算法的平均查找长度总是与查找表中数据元素的总数有关，数据元素个数越多，查找所花费的平均比较次数就越多。而数据元素在查找表中的存储位置往往与数据元素的关键字无关。

1．哈希函数与哈希表

如果能在数据元素的关键字与数据元素的存储位置之间建立一种对应关系，就可以通过对关键字的操作直接得到数据元素的存储位置，而不需要进行多次比较，从而提高了查找的效率。哈希法就是基于这种思想设计的一种查找技术，其对应关系称为哈希函数（hash function），实质上是关键字集合到地址空间的映射。按此方法建立的一组数据元素的存储区域称为哈希表（hash table）。以哈希函数构造哈希表的过程称为哈希造表，以哈希函数在哈希表中查找的过程称为哈希查找。

2．哈希查找技术的设计思想

哈希查找技术的设计思想是，由数据元素的关键字决定数据元素的存储位置，即根据数据元素的关键字值 k 计算出相应的哈希函数值 hash(k)，这个值就是该数据元素的存储位置。哈希造表和哈希查找的过程都是基于这个思想实现的。

在计算哈希函数时，如果有两个不同的关键字 k_1 和 k_2，对应相同的哈希函数值，即 $k_1 \neq k_2$ 而 $\text{hash}(k_1) = \text{hash}(k_2)$，表示不同关键字的多个数据元素映射到同一个存储位置。这种现象称为冲突（collision），与 k_1 和 k_2 对应的数据元素称为同义词。

如果哈希表的存储空间足够大，使得所有数据元素的关键字与其存储位置是一一对应的，则不会产生冲突。这是理想情况，很难实现。因为理论上讲，数据量是无限的，而系统的存储空间资源是有限的，系统不能提供无限数量的空间，所以冲突是不可避免的。要考虑的问题是当冲突发生时如何解决冲突。

哈希查找技术的关键问题在于以下两点：

- 设计一个好的哈希函数，尽可能减少冲突。
- 因为冲突是不可避免的，发生冲突时，使用一种解决冲突的有效方法。

3．设计哈希函数

一个好的哈希函数的标准是，能将关键字值均匀地分布在整个空间中，使冲突的机会尽可能地减少。地址分布得越均匀，产生冲突的可能性就越小。因此，在设计哈希函数时，应该使组成关键字值的所有成分在实现到地址的转换中都起作用，从而反映不同关键字值之间数据差

素的差异。下面介绍设计哈希函数的几种常用方法。

1）除留余数法

设 $\text{hash}(k)=k\%p$，函数 $\text{hash}(k)$ 的取值为 0～$p-1$。

此法影响哈希函数好坏的关键在于 p 值的选取。一般地说，p 值有多种选法：

- 选 p 为 10 的幂次，如 $p=10^2$ 表示取关键字值的后两位作为地址。此时，冲突产生在后两位相同的关键字上，它们的哈希函数相同。如 321 与 521 是同义词，在哈希表中的地址都是 21，产生冲突。
- 取 p 为小于哈希表长度 max 的最大素数。max 与 p 的取值关系如表 7-2 所示。

表 7-2　max 与 p 的取值关系

max	8	16	32	64	128	256	…
P	7	13	31	61	127	251	…

2）平方取中法

将关键字值 k^2 的中间几位作为 $\text{hash}(k)$ 的值，位数取决于哈希表的长度。冲突的可能性较小。例如，$k=4731$，$k^2=22\,382\,361$，若表长为 100，取中间两位，则 $\text{hash}(k)=82$。

3）折叠法

将关键字分成几部分，按照某种约定把这几部分组合在一起。

每种关键字都有自己的特殊性。例如，以整数或字符串作为关键字时，处理方式不相同。因此，不存在一种哈希函数对任何关键字集合都是最好的。在实际应用中，应该根据具体情况分析关键字与地址空间之间可能的对应关系，构造不同的哈希函数，或将几种基本的哈希函数组合起来使用，以求达到最佳效果。

设计哈希函数时，应该考虑以下几方面的因素：

- 计算哈希函数所需的时间。
- 哈希表的大小。
- 关键字的性质和分布情况。
- 数据元素的查找频率。

综合以上因素与实际情况，设计一个最佳的哈希函数。

4．解决冲突的方法

虽然一个好的哈希函数能使关键字不同的数据元素分布较为均匀，但只能减少冲突，而不能从根本上避免冲突。所以，当冲突发生时必须解决冲突。解决冲突的方法有很多，这里介绍线性开放寻址法和拉链法。

1）线性开放寻址法

欲将关键字为 k 的数据元素存放在哈希表的 $i=\text{hash}(k)$ 位置上。如果该位置为空，则存入该数据元素；否则产生冲突，继续探测下一个空位置。当探测完整个哈希表而没有找到空位置时，说明哈希表已满，不能再存入新的数据元素，这种情况称为溢出。必须采取措施处理溢出情况，通常再建立一个溢出表，采用顺序查找。原来的哈希表称为哈希基表。例如，对于如下的关键字序列：

{9，9，24，44，32，86，36，3，62，56}

设哈希函数 $\text{hash}(k)=k\%10$，采用线性开放寻址法存储的哈希基表与溢出表如图 7.12 所示

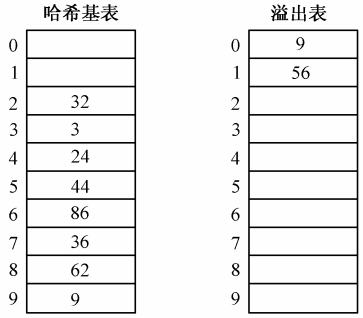


图 7.12　线性开放寻址法的哈希表

查找关键字为 k 的数据元素时，首先与哈希基表中 i=hash(k)位置的数据元素进行比较，如果该位置是 k，则查找成功，否则继续向后依次查找，此时哈希查找则蜕变为顺序查找。如果在哈希基表中没有找到，还要在溢出表中采用顺序查找算法继续查找。线性开放寻址法简单可行，实现方便，但存在严重缺陷：

- 容易产生堆积现象，即存入哈希表的数据元素连成一片，增大了产生冲突的可能性。
- 必须另行处理溢出现象，即对溢出表也要设计存储结构及查找算法。
- 所允许的操作只能是查找和插入数据元素，不能删除数据元素，否则探测序列将中断，无法查到具有相同哈希函数值的后继数据元素。

2）拉链法

拉链法是解决冲突既灵活又有效的方法。拉链法的基本思想是，根据关键字 k，将数据元素存放在哈希基表中的 i=hash(k)位置上。如果产生冲突，则创建一个结点存放该数据元素，并将该结点插入到一个链表中。这种由冲突的数据元素构成的链表称为哈希链表。一个哈希基表与若干条哈希链表相连。例如，对于如下的关键字序列：

{9，9，24，44，32，86，36，3，62，56}

设哈希函数 hash(k) = k % 10，hash(k)对应哈希基表 table 的下标值 i，采用拉链法的哈希表结构如图 7.13 所示。

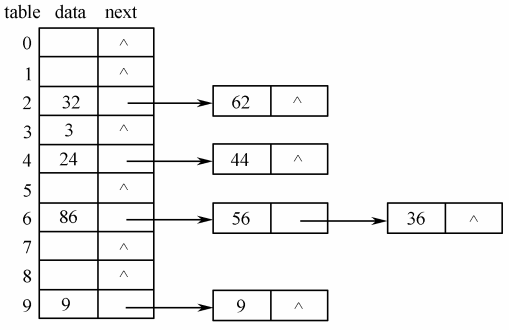


图 7.13　拉链法的哈希表

实际上，哈希链表是一条单向链表，哈希基表的数据元素结构与单向链表的结点结构相同。为便于使用本书已声明的单向链表 Onelink1 类及其结点 OnelinkNode 类，图中以 data 成员表示关键字。

拉链法的哈希查找算法描述如下：

- 设给定值为 k，若哈希基表中 table[hash(k)]位置数据元素的关键字等于 k，则查找成功
- 否则，产生冲突，在由 table[hash(k)].next 指向的哈希链表中按顺序查找。只要将这一条链查找完，就能确定查找是否成功。

拉链法克服了开放寻址法的缺陷，它不会产生堆积现象，不必另外考虑溢出问题，并可以随时对哈希表（包括哈希链表）进行插入、删除、修改等操作。因而拉链法是一种有效的存储结构和查找方法。

哈希链表是动态的，冲突越多，链表越长。因此，要设计哈希函数，使数据元素尽量均匀地分布在哈希基表中，哈希链表越短越好。如果哈希函数的均匀性较差，则会造成哈希基表中空闲单元较多，而某些哈希链表很长的情况，此时占用的存储空间较大，查找效率降低。

【例 7.6】 构造拉链法的哈希表并进行哈希查找。

本例声明 HashSearch1 类实现哈希查找。其中 table 数组作为哈希基表，元素是单向链表的结点 OnelinkNode 类。构造方法为 table 数组分配存储单元，并将产生的 n 个随机数 k 插入到哈希表中。

hash( )方法实现哈希函数。insert( )方法将 k 值插入到哈希基表中，发生冲突时，插入到哈希链表中。output( )方法输出哈希基表及所有哈希链表中的数据元素值。hashsearch( )方法查找 k 值，先在哈希基表中查找，发生冲突时再到哈希链表中查找，查找成功返回 true，否时返回 false。

```
import ds_java.OnelinkNode;
import ds_java.Onelink1;
public class HashSearch1                    //哈希查找
{
   private OnelinkNode table[];
   public HashSearch1(int n)                //产生 n 个随机数插入哈希表
   {
      int i=0,k=0;
      table=new OnelinkNode[n];
      System.out.print("k=");
      for(i=0;i<n;i++)
      {
         k=(int)(Math.random()*100);
         System.out.print(k+" ");
         insert(k);
      }
      System.out.println();
   }
   public int hash(int k)                   //哈希函数确定 k 值的位置
   {
      return k%10;
   }
   public void insert(int k)                //哈希基表中插入 k 值
```

```
{
    OnelinkNode q;
    int i=hash(k);
    if(table[i]==null)
        table[i]=new OnelinkNode(k);      //k 值存入哈希基表
    else
    {                                     //冲突时
        q=new OnelinkNode(k);             //k 值结点插入哈希链表
        q.next=table[i].next;             //作为哈希链表的第 1 个结点
        table[i].next=q;
    }
}
public void output()                      //输出哈希基表及全部哈希链表
{
    for(int i=0;i<table.length;i++)
    {
        System.out.print("table["+i+"]= ");
        Onelink1 link1=new Onelink1(table[i]);
        link1.output();
    }
}
public OnelinkNode hashsearch(int k)      //哈希查找
{
    int i=hash(k);
    if(table[i]==null)
        return null;
    else
        if(k==table[i].data)              //k 在哈希基表中
            return table[i];
        else                              //k 在哈希链表中
        {
            LinkSearch1 link2=new LinkSearch1(table[i]);
            return link2.search(k);       //返回在哈希链表中查找 k 的结果
        }
}
public static void main(String args[])
{
    int n=10,k=20;
    HashSearch1 h1=new HashSearch1(n);
    h1.output();
    System.out.print("hashsearch("+k+")=");
    OnelinkNode p=h1.hashsearch(k);
```

```
            if(p!=null)
               System.out.println(p.data);
            else
               System.out.println("null");
         }
      }
```

编译时将 LinkSearch1.class **文件存放在当前文件夹中。程序运行结果如下：**

```
k=69 62 75 88 61 58 7 46 70 87
table[0]=70
table[1]=61
table[2]=62
table[3]=
table[4]=
table[5]=75
table[6]=46
table[7]=7 -> 87
table[8]=88 -> 58
table[9]=69
search(20)=null
```

习　题　7

7.1　**比较各种查找算法的特点、性能及适用情况。**

7.2　**试分别画出在线性表（**a，b，c，d，e，f**）中，查找** e、f **和** g **的折半查找过程。**

7.3　**试编写折半查找的递归算法。**

7.4　**实现如图** 7.8 **所示电话簿数据元素的分块存储结构。**

7.5　**哈希查找的设计思想是怎样的？两个关键问题是什么？**

7.6　**以关键字序列**{Jan，Feb，Mar，Apr，May，Jun，Jul，Aug，Sep，Oct，Nov，Dec}**构造哈希表。**

7.7　**说明二叉排序树与堆的差别。**

7.8　**将自然数** 1 **到** 9 **填入图** 7.14 **所示的二叉树中，构成一棵二叉排序树。**

7.9　**画出以序列**{3，2，5，8，7，9，3，1，4，6}**建立的一棵二叉排序树。**

7.10　**编程判断一棵二叉树是否为二叉排序树。**

7.11　**建立学生档案，并设计查找算法。**

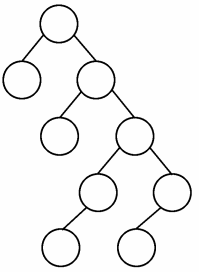


图 7.14　**二叉树**

实　习　7

1．**实验目的：哈希查找算法研究。**

2．**题意：在拉链法的哈希表中，删除给定关键字值为** x **的数据元素。**

第 8 章　图

生活中我们通常以图的方式表达文字难以描述的信息，如城市交通图、线路图、网络图等。

在离散数学中，图论（graphic theory）注重研究图的纯数学性质；在数据结构中，图结构侧重于计算机中如何存储图以及如何实现图的操作和应用等。

图是一种比线性表和二叉树更复杂的非线性数据结构。线性表的数据元素之间仅有顺序关系，树结构的数据元素之间存在层次关系，而在图结构中，数据元素之间的关系没有限制，任意两个数据元素之间都可以相邻，即每个数据元素可有多个前驱数据元素，多个后继数据元素。

对于离散结构的刻画，图是一种有力的工具。在运筹规划、网络研究和计算机程序流程分析中，都存在图结构的应用问题。

本章主要讨论图的概念、存储结构以及生成树、最短路径等。

建议本章授课 8 学时，实验 2 学时。

8.1　图的基本知识

8.1.1　图的定义

图（graph）是由结点集合及结点间的关系集合组成的一种数据结构。图中的结点又称为顶点，结点之间的关系称为边（edge）。一个图 G 记为

$$G=(V, E)$$

其中，V 是结点 x 的有限集合，E 是边的有限集合。即

$$V=\{x|x\in\text{某个数据元素集合}\}$$

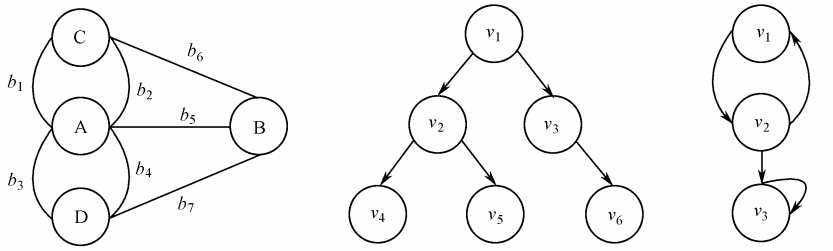
$$E=\{(x, y)|x, y\in V\} \text{ 或 } E=\{\langle x, y\rangle |x, y\in V\}$$

其中，(x, y)表示从结点 x 到 y 的一条双向通路，即(x, y)没有方向；$\langle x, y\rangle$ 表示从结点 x 到结点 y 的一条单向通路，即 $\langle x, y\rangle$ 是有方向的。图 8.1 中的 G_1、G_2 和 G_3 都是图结构。

1．无向图

一个图 G 中，如果用结点的无序偶对代表一条边，则称边是无向的，用圆括号将一对结点括起来表示无向边，如(A, B)和(B, A)表示同一条边，此时图 G 称为无向图（undirected graph）。

图 8.1 中的 G_1 是无向图，G_1 的结点集合 V 和边的集合 E 为：



(a) 哥尼斯堡七桥，无向图G_1　　(b) 有向图G_2　　(c) 有向图G_3

图 8.1　图结构

$$V(G_1)=\{A, B, C, D\}$$

$$E(G_1)=\{(C, A), (C, A), (A, D), (A, D), (A, B), (C, B), (B, D)\}$$

其中，$b_1=(C, A)$，$b_2=(C, A)$，称 b_1 和 b_2 为重边。

2．**有向图**

一个图 G 中，如果用结点的有序偶对代表一条边，则称边是有向的，用尖括号将一对结点括起来表示有向边，如〈v_1, v_2〉，v_1 称为边的起点（initial node**），v_2 称为边的终点（**terminal node**），所以〈v_1, v_2〉和〈v_2, v_1〉分别表示两条不同的边，此时图 G 称为有向图（**directed graph**）。图 8.1 中的 G_2 和 G_3 都是有向图，G_3 的结点集合 V 和边的集合 E 为：**

$$V(G_3)=\{v_1, v_2, v_3\}$$

$$E(G_3)=\{〈v_1, v_2〉, 〈v_2, v_1〉, 〈v_2, v_3〉, 〈v_3, v_3〉\}$$

有向图中的边又称为弧（arc**）。在示意图上用箭头表示弧的方向，箭头从起点指向终点。当边 $e=(v, v)$ 或 $e=〈v, v〉$ 时，称 e 为环（**loops**）。例如，G_3 中的〈v_3, v_3〉是环。**

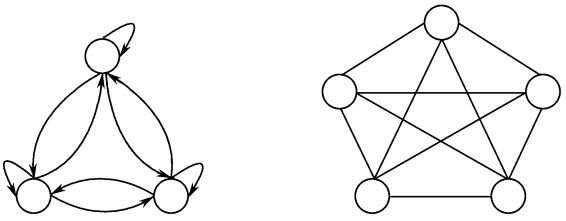
3．**完全图**

一个有 n 个结点的无向图，其边的最大数目为 $n\times(n-1)/2$；一个有 n 个结点的有向图，其弧的最大数目为 $n\times(n-1)$。如果一个图的边数达到最大值，则称该图为完全图（complete graph**），如图 8.2 所示。n 个结点的完全图通常记为 K_n。**

无环且无重边的无向图称为简单图。在下面的讨论中，我们仅考虑简单图。

4．**带权图**

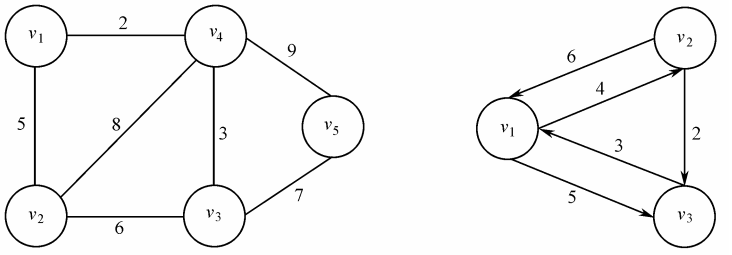
如果图的每条边上都加一个实数作为权（weight**），称为带权图（**weighted graph**）或网（**network**）。权可以表示从一个结点到另一个结点的距离、花费的代价、所需的时间等。**



(a) 有向完全图　　(b) 无向完全图

图 8.2　完全图

图 8.3 中是两个带权图，边（或弧）上标出的实数为权值。



(a) 带权的无向图G_4　　(b) 带权的无向图G_5

图 8.3　两个带权图

5．相邻结点

若(v_i, v_j)是一个无向图的一条边，则称 v_i 和 v_j 是相邻结点，边(v_i, v_j)是与结点 v_i 和 v_j 相关联的边，v_i 和 v_j 是边(v_i, v_j)相关联的结点。

若〈v_i，v_j〉是一个有向图的一条边，则称结点 v_i 邻接到结点 v_j，结点 v_j 邻接于结点 v_i，边〈v_i，v_j〉是与结点 v_i 和 v_j 相关联的边。

8.1.2　结点的度

1．度、入度、出度

图中与结点 v 相关联的边的数目称为结点的度（degree），记为 TD(v)。1 度的结点称为悬挂点（pendant nodes）。在 G_1 中，结点 B 的度 TD(B)=3。

在有向图中，以 v 为终点的弧的数目称为 v 的入度（in-degree），记为 ID(v)；以 v 为起点的弧的数目称为 v 的出度（out-degree），记为 OD(v)。出度为 0 的结点称为终端结点（或叶子结点）。结点的度是入度与出度之和，有

$$\mathrm{TD}(v)=\mathrm{ID}(v)+\mathrm{OD}(v)$$

在 G_3 中，结点 v_2 的入度 ID(v_2)=1，出度 OD(v_2)=2，度 TD(v_2)=3。

2．度与边的关系

对于任意一个图 G，设其边数为 e，结点集合为$\{v_1, v_2, \cdots, v_n\}$，那么

$$e=\frac{1}{2}\sum_{i=1}^{n}\mathrm{TD}(v_i)$$

当 G 为有向图时，上式可写为

$$\sum_{i=1}^{n}\mathrm{ID}(v_i)=\sum_{i=1}^{n}\mathrm{OD}(v_i)=e$$

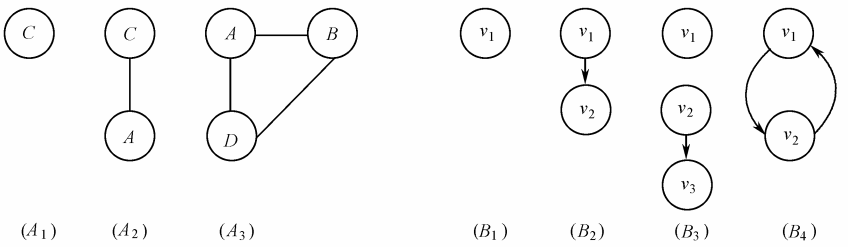
$$\sum_{i=1}^{n}\mathrm{TD}(v_i)=\sum_{i=1}^{n}\mathrm{ID}(v_i)+\sum_{i=1}^{n}\mathrm{OD}(v_i)=2e$$

8.1.3　子图

1．子图、真子图

设图 $G=(V, E)$，$G'=(V', E')$，若$V'\subseteq V$，$E'\subseteq E$，并且 E' 中的边所关联的结点都在 V' 中，则称图 G' 是 G 的子图（subgraph）。如果 $G'\neq G$，称图 G' 是 G 的真子图。图 8.4 列出了

G_1和 G_3的部分真子图。



(a) G_1的部分真子图　　(b) G_3的部分真子图

图 8.4　G_1和 G_3的部分真子图

2．生成子图

如果 G'是 G 的子图，且 $V'=V$，称图 G'是 G 的生成子图（spanning subgraph）。

8.1.4　路径、回路及连通性

1．路径、路径长度、回路

在图 $G=(V, E)$中，若从结点 v_i 出发，沿一些边经过一些结点 v_{p_1}，v_{p_2}，…，v_{p_m} 到达结点 v_j，则称结点序列$(v_i, v_{p_1}, v_{p_2}, \cdots, v_{p_m}, v_j)$是从结点 v_i 到 v_j 的一条路径（path）。它经过的边(v_i, v_{p_1})，$(v_{p_1}, v_{p_2}),\cdots,(v_{p_m}, v_j)$是属于 E 的边。

若 G 是有向图，则路径也是有向的，即〈v_i, v_{p_1}〉，〈v_{p_1}, v_{p_2}〉,…,〈v_{p_m}, v_j〉都在 E 中，v_i 为路径的起点，v_j 为终点。

路径长度（pathlength）定义为这条路径上边的数目。如果在一条路径中，除起点和终点外，其他结点都不相同，则此路径称为简单路径（simple path）。把路径(v_1, v_2)，(v_2, v_3)和(v_3, v_4)缩写成(v_1, v_2, v_3, v_4)。起点和终点相同且长度大于 1 的简单路径称为回路（cycle）。例如，图 G_3 中的路径(v_2, v_1, v_2)是一条回路。

带权图中，从起点到终点的路径上各条边的权值之和称为这条路径的路径长度。例如，图 8.3（a）中从结点 v_1 到 v_5 的一条路径(v_1, v_4, v_5)的路径长度为 2+9=11。

2．有根的图和图的根

一个有向图 G 中，若存在一个结点 v_0，从 v_0 有路径可以到达图 G 中其他所有结点，则称此有向图为有根的图，称 v_0 为图 G 的根。

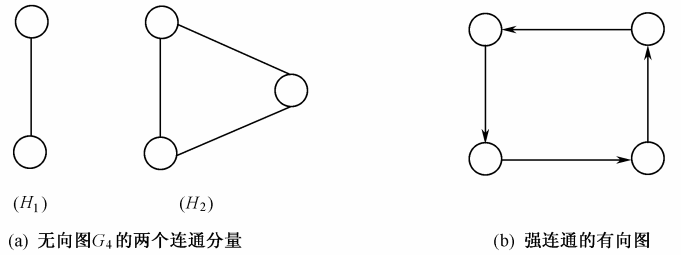
3．连通图

在无向图 G 中，若从结点 v_i 到 v_j 有一条路径，则称 v_i 和 v_j 是连通的（connected）。若图 G 中任意两个结点 v_i 和 v_j（$v_i \neq v_j$）都连通，则称 G 为连通图（connected graph）。非连通图的极大连通子图称为该图的连通分量（connected component）。图 8.1 中的 G_1 是连通图。图 8.5（a）不是连通图，它有两个连通分量 H_1 和 H_2。

4．强连通图

在有向图中，若任意两个结点 v_i 和 v_j（$v_i \neq v_j$）有一条从 v_i 到 v_j 的路径，同时还有一条从 v_j 到 v_i 的路径，则称该有向图是强连通的（strongly connected）。图 8.1 中的有向图 G_2 不是强

连通的，因为从结点 v_3 到 v_2 没有路径。一个有向图的强连通的最大子图定义为此图的强连通分支。图 8.5（b）是强连通的有向图。



(a) 无向图G_4的两个连通分量　　(b) 强连通的有向图

图 8.5　图的连通性

8.2　图的存储结构

与线性表和二叉树一样，图的存储结构也有顺序存储结构和链式存储结构两种。图的顺序存储结构为邻接矩阵，链式存储结构为邻接表。

8.2.1　邻接矩阵

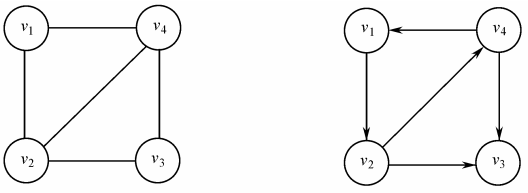
1．邻接矩阵的定义

图的邻接矩阵（adjacency matrix）是表示结点间相邻关系的矩阵。设 $G=(V, E)$是一个具有 n（$n\geq 1$）个结点的图，G 的邻接矩阵 $\boldsymbol{A}$ 是具有下列性质的 n 阶方阵：

$$a_{i,j}=\begin{cases}1 & \text{若}(v_i,v_j)\in E\text{或}<v_i,v_j>\in E\\ 0 & \text{若}(v_i,v_j)\notin E\text{或}<v_i,v_j>\notin E\end{cases}$$

例如，对于图 8.6 所示的无向图 G_6 与有向图 G_7，对应的邻接矩阵分别为 $\boldsymbol{A}_1$ 和 $\boldsymbol{A}_2$：

$$\boldsymbol{A}_1=\begin{bmatrix}0&1&0&1\\1&0&1&1\\0&1&0&1\\1&1&1&0\end{bmatrix}\quad \boldsymbol{A}_2=\begin{bmatrix}0&1&0&0\\0&0&1&1\\0&0&0&0\\1&0&1&0\end{bmatrix}$$



(a) 无向图G_6　　(b) 有向图G_7

图 8.6　无向图与有向图

可以看出，无向图的邻接矩阵是对称的，有向图的邻接矩阵不一定对称。用邻接矩阵表示一个有 n 个结点的图需要 n^2 个存储单元。由于无向图的邻接矩阵是对称的，可以只存储邻接矩阵的上三角数据元素或下三角数据元素。

2．邻接矩阵与结点的度

用邻接矩阵表示图，容易判定任意两个结点 v_i 和 v_j 之间是否有边相连，并容易求得各个结点的度。如果 $a_{i,j}=1$，则 v_i 和 v_j 之间有边相连；$a_{i,j}=0$，则 v_i 和 v_j 之间无边相连。

对于无向图，邻接矩阵第 i 行上各数据元素之和是结点 v_i 的度；对于有向图，矩阵第 i 行上各数据元素之和是结点 v_i 的出度，第 i 列上各数据元素之和是结点 v_i 的入度。

3．带权图的邻接矩阵

在带权图中，设 $w(v_i, v_j)$表示边（v_i, v_j）或〈v_i, v_j〉上的权值，其邻接矩阵定义如下：

$$a_{ij}=\begin{cases} w(v_i,v_j) & \text{若} v_i\neq v_j \text{且}(v_i,v_j)\in E\text{或}<v_i,v_j>\in E \\ \infty & \text{若} v_i\neq v_j \text{且}(v_i,v_j)\notin E\text{或}<v_i,v_j>\notin E \\ 0 & \text{若} v_i=v_j \end{cases}$$

图 8.3 中的两个带权图 G_4 和 G_5 的邻接矩阵分别为 $\boldsymbol{A}_4$ 和 $\boldsymbol{A}_5$：

$$\boldsymbol{A}_4=\begin{bmatrix} 0 & 5 & \infty & 2 & \infty \\ 5 & 0 & 6 & 8 & \infty \\ \infty & 6 & 0 & 3 & 7 \\ 2 & 8 & 3 & 0 & 9 \\ \infty & \infty & 7 & 9 & 0 \end{bmatrix} \quad \boldsymbol{A}_5=\begin{bmatrix} 0 & 4 & 5 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

4．声明以邻接矩阵存储的图类

下面声明 Graph1 类表示一个具有 n 个结点的以邻接矩阵存储的图，将图的邻接矩阵存储在一个二维数组 mat 中。

```
public class Graph1                          //以邻接矩阵存储的图类
{
    protected int n;                         //图的结点个数
    protected int mat[][];                   //二维数组存储图的邻接矩阵
}
```

8.2.2 邻接表

1．无向图的邻接表

用邻接矩阵表示图，占用的存储单元个数只与图中结点个数有关，而与边的数目无关。一个有 n 个结点的图需要 n^2 个存储单元，若其边数比 n^2 少得多，则它的邻接矩阵中就会有很多零元素，造成空间上的浪费。这时可以用邻接表（adjacency list）来存储图，占用的存储单元个数不但与图中结点个数有关，而且与边的数目也有关。同是 n 个结点的图，如果边数较少则占用的存储单元也较少。

邻接表包括两部分：结点表和边表。

结点表以顺序存储结构保存图中的所有结点。数组中的每个元素对应于一个结点，包括两个成员：data 和 next。data 表示结点数据元素信息，next 指向该结点的边表。

边表以链式存储结构保存与一个结点相关联的若干条边。图中的每个结点都有一个边表，边表中的每个结点对应于与该结点相关联的一条边，它有两个成员：data 和 next。data 表示结点数据元素信息，next 指向与该结点相关联的另一条边。图 8.6（a）中无向图 G_6 的邻接表如图 8.7 所示。

与无向图邻接矩阵的对称性质相类似，无向图的邻接表中也将每条边存储了两次，即每条边分别存储在与该边相关联的两个结点的边表中，因此，存储 n 个结点 m 条边的无向图需要占用 $n+2m$ 个结点存储单元。

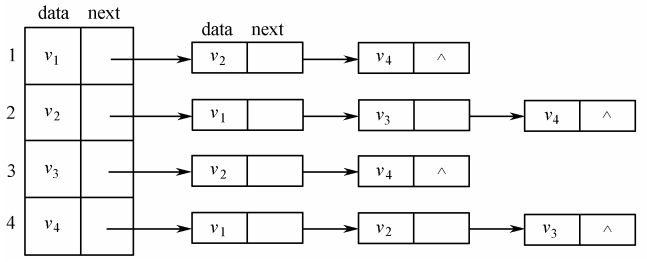


图 8.7　无向图 G_6 的邻接表

2．有向图的邻接表

用邻接表存储有向图，根据边的方向，边表可分为两种：出边表和入边表。出边表是以该结点为起点的边组成的边表。入边表是以该结点为终点的边组成的边表。

如果只保存入边表和出边表之一，则需要占用 $n+m$ 个结点存储单元。图 8.6（b）中有向图 G_7 的邻接表如图 8.8 所示。

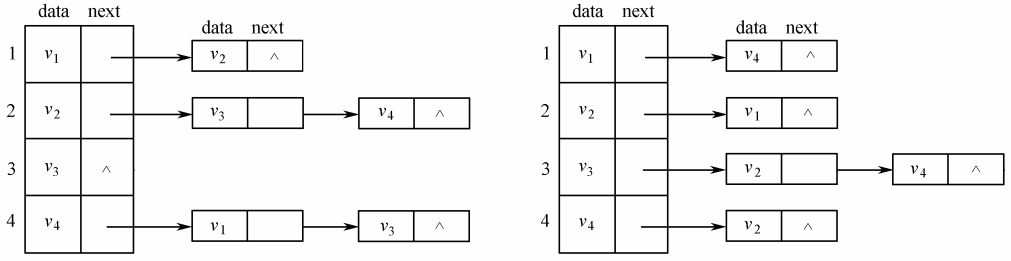


图 8.8　有向图 G_7 的邻接表

用邻接表存储图，不仅是当 $m \ll n^2$ 时节省了存储单元，而且因为与一个结点相关联的所有边都在同一个边表中，也给图的操作提供了方便。

3．声明以邻接表存储的图类

下面声明 Graph2 类表示一个具有 n 个结点的以邻接表存储的图，其中成员变量 table 数组表示图的邻接表。table 数组的元素类型为单向链表的结点类 OnelinkNode。

```
import ds_java.OnelinkNode;                    //单向链表的结点类
public class Graph2                            //以邻接表存储的图类
{
    private OnelinkNode table[];               //图的邻接表
}
```

8.3　图的遍历

与二叉树的遍历类似，遍历是图的一种基本操作。对于一个图，从其中的一个结点出发以某种次序顺序地访问图中的每个结点，并且每个结点只被访问一次，这一过程称为图的遍历（traversal）。

图的遍历通常有两种策略：深度优先搜索遍历和广度优先搜索遍历。图的深度优先搜索遍

历类似于树的先根遍历，图的广度优先搜索遍历类似于树的层次遍历。

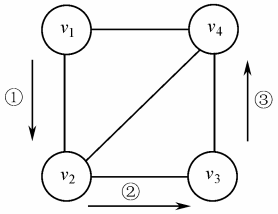
8.3.1 深度优先遍历

1．深度优先遍历算法描述

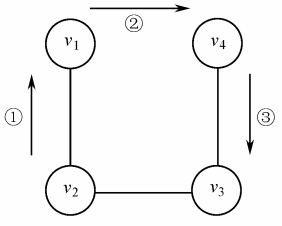
深度优先搜索（depth first search）遍历类似于树的先根遍历，可以看成是树的先根遍历的推广。其递归算法描述如下：

- 从图中选定的一个结点 v_0 出发，访问 v_0。
- 查找与 v_0 有边相连且未被访问过的另一个结点 v_j。
- 若有 v_j，从 v_j 出发继续进行深度优先搜索遍历。
- 若找不到 v_j，说明从 v_0 开始能够到达的所有结点都已被访问过，遍历结束。

对图 8.6 中无向图 G_6 进行深度优先搜索遍历，如图 8.9 所示。



(a) 从顶点 v_1 出发，一种可能的访问序列为 $\{v_1, v_2, v_3, v_4\}$



(b) 从顶点 v_2 出发,一种可能的访问序列为 $\{v_2, v_1, v_4, v_3\}$

图 8.9 无向图的深度优先搜索遍历

显然，对一个连通的无向图或一个强连通的有向图，从某一结点 v_0 出发执行一次深度优先搜索，可以访问图的每个结点；否则，只能访问图中的一个连通分量。

图的遍历要比二叉树的遍历复杂得多，因为图中一个结点可能与多个结点相邻接，在访问了一个结点后，可能沿着某条路径又回到该结点。为了避免同一结点重复多次访问，在遍历过程中必须对访问过的结点做标记。

2．以邻接矩阵存储的图的深度优先遍历算法实现

在以邻接矩阵存储的图 Graph1 类中，增加一个一维数组成员变量 visited 和一个成员方法 depthfs( )。

visited 用于标志图中结点是否已被访问。每个数组元素对应图的一个结点，若一个结点已被访问，则相应的数组元素被置为 1，否则为 0。构造方法中创建 visited 数组，其元素值全为 0，表示初始状态是图中所有结点均未被访问过。

depthfs( )方法实现图的深度优先遍历算法。增加成员后的 Graph1 类如下：

```
package ds_java;
public class Graph1                          //以邻接矩阵存储的图类
{
   protected int n;                          //图的结点个数
   protected int mat[][];                    //二维数组存储图的邻接矩阵
   protected int visited[];                  //访问标记数组
   public Graph1(int m1[][])
   {
```

```
        n=m1.length;
        mat=new int[n][n];
        System.arraycopy(m1,0,mat,0,n);   //System 类方法，复制数组
        visited=new int [n];
    }
    public Graph1()
    {  }
    public void depthFirstSearch()       //图的深度优先遍历
    {
        System.out.println("深度优先遍历 Depth first search:");
        for(int k=1;k<=n;k++)                //k 为结点序号,从 1 开始
        {
            depthfs(k);
            System.out.println();
            unvisited();
        }
    }
    public void depthfs(int k)               //从结点 k 开始的深度优先遍历
    {
        int i,j=0;
        System.out.print("  v"+k+" ->");
        i=k-1;                               //i 下标从 0 开始
        visited[i]=1;
        while(j<n)                           //查找与 k 相邻的其他结点
            if(mat[i][j]==1 && visited[j]==0)
                depthfs(j+1);                //递归
            else
                j++;
    }
    public void unvisited()                  //设置未访问标记
    {
        int i;
        for(i=0;i<visited.length;i++)
            visited[i]=0;
    }
}
```

【例 8.1】 以邻接矩阵表示的图的深度优先遍历算法测试。

```
import ds_java.Tree1;
public class Graph1_ex                       //图类的测试
{
    public static void main(String args[])
```

```
    {
        int mat1[][]={{0,1,0,1},                    //无向图 G6 的邻接矩阵
                      {1,0,1,1},
                      {0,1,0,1},
                      {1,1,1,0}};
        Graph1 g1=new Graph1(mat1);
        g1.depthFirstSearch();
    }
}
```

程序运行结果如下：

```
深度优先遍历 Depth first search:
v1 ->  v2 ->  v3 ->  v4 ->
v2 ->  v1 ->  v4 ->  v3 ->
v3 ->  v2 ->  v1 ->  v4 ->
v4 ->  v1 ->  v2 ->  v3 ->
```

在一个含有 *n* 个结点和 *e* 条边的图上进行深度优先遍历，对每个结点，depthfs( )**方法至**被调用一次。因为一旦在一个结点上调用 **depthfs( )**方法，便标志该结点“ 已被访问 ”，此后**不再对该结点调用 **depthfs( )**方法。**

3．**以邻接表存储的图的深度优先遍历算法实现**

在以邻接表存储的图 **Graph2 类中，同样需要成员变量 **visited** 数组和 *n*，所以 **Graph2** 类**计为继承 **Graph1** 类。**

构造方法中创建具有 *n*＋1 个元素的 **table 数组，并将邻接矩阵 **mat** 中表示的边转换成邻**表 **table** 中的出边表。其中，**table[0]**元素不用，使得结点序号 *i* 与 **table** 中的下标一致。**

output( )方法输出邻接表 **table** 及其出边表的各个结点数据元素值。**depthfs( )**方法覆盖超**Graph1** 中的同名方法。修改后的 **Graph2** 类如下：**

```
import ds_java.OnelinkNode;                 //单向链表的结点类
public class Graph2 extends Graph1          //以邻接表存储的图类
{
    private OnelinkNode table[];            //图的邻接表
    public Graph2(int mat[][])              //以邻接矩阵建立图的邻接表
    {
        n=mat.length;                       //继承 Graph1 类的成员
        table=new OnelinkNode[n+1];         //建立结点表，多一个元素
        OnelinkNode p=null,q;
        int i,j;
        for(i=1;i<=n;i++)                   //table[0]不用，
    {                                       //结点序号 i 与 table 中的下标一
            table[i]=new OnelinkNode(i);    //创建 i 在结点表中的元素
            p=table[i];                     //建立结点 i 的出边表
            for(j=1;j<=n;j++)               //查找与 i 相邻的其他结点 j
```

```
            if(mat[i-1][j-1]==1)
            {
                q=new OnelinkNode(j);       //出边表中添加结点
                p.next=q;
                p=q;
            }
        }
        visited=new int [n+1];              //继承 Graph1 类的成员
    }
    Graph2()
    {  }
    public void output()                    //输出邻接表
    {
        OnelinkNode p;
        System.out.println("邻接表 table:");
        for(int i=0;i<table.length;i++)
        {
            System.out.print("table["+i+"]= ");
            if(table[i]!=null)
            {
                System.out.print("v"+table[i].data);
                p=table[i].next;
                while(p!=null)
                {
                    System.out.print(" ->  v"+p.data);
                    p=p.next;
                }
            }
            System.out.println("  null");
        }
    }
    public void depthfs(int k)              //图的深度优先遍历
    {                                       //k 为结点序号
        int i,j=0;
        OnelinkNode p;
        System.out.print("  v"+k+" ->");
        i=k;                                //i 下标从 0 开始
        visited[i]=1;
        if(table[i]!=null)                  //查找有边相连的另一结点
        {
            p=table[i].next;
            while(p!=null)
```

```
            {
                j=p.data;
                if(visited[j]==0)
                    depthfs(j);                    //再递归访问
                else
                    p=p.next;
            }
        }
    }
}
```

在例 8.1 的 main 方法中，增加下列调用语句，可输出图的邻接表：

```
    g2.output();
```

对于有向图 G_7，程序运行结果如下：

```
邻接表 table:
table[0]=  null
table[1]= v1 ->  v2  null
table[2]= v2 ->  v3 ->  v4  null
table[3]= v3  null
table[4]= v4 ->  v1 ->  v3  null

深度优先遍历 Depth first search:
 v1 ->  v2 ->  v3 ->  v4 ->
 v2 ->  v3 ->  v4 ->  v1 ->
 v3 ->
 v4 ->  v1 ->  v2 ->  v3 ->
```

设图有 n 个结点和 e 条边（$e \geq n$），若用邻接矩阵存储，处理一行的时间为 $O(n)$，矩阵共有 n 行，故所需时间为 $O(n^2)$。若用邻接表存储，运行时间为 $O(e)$。

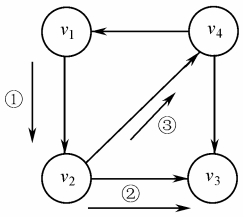
8.3.2 广度优先遍历

1．广度优先遍历算法描述

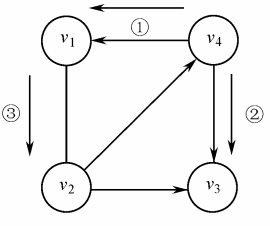
广度优先搜索（breadth first search）遍历类似于树的层次遍历。其中必须设立一个队列来保存访问过的结点，算法描述如下：

- 从图中选定的一个结点 v_0 作为出发点，访问 v_0。
- 将访问过的结点 v_0 入队。
- 当队列不空时，进入循环：
  - ✧ 有结点 v_i 出队。
  - ✧ 访问与 v_i 有边相连的且未被访问过的所有结点 v_j。
  - ✧ 访问过的结点 v_j 入队。
- 当队列空时，循环结束，说明从 v_0 开始能够到达的所有结点都已被访问过。

对图 8.6 中无向图 G_7 进行广度优先搜索，如图 8.10 所示。



(a) 从顶点v_1出发，一种可能的访问序列为{v_1，v_2，v_3，v_4}



(b) 从顶点v_4出发,一种可能的访问序列为{v_4，v_1，v_3，v_2}

图 8.10　有向图的广度优先搜索

在广度优先搜索遍历的算法中，由于使用队列保存访问过的结点，若结点 v_0 在结点 v_1 之前被访问，则与结点 v_0 相邻接的结点一定在与结点 v_1 相邻接的结点之前被访问。

如果图 G 是一个连通的无向图或强连通的有向图，从 G 的任一结点出发，进行一次广度优先搜索便可遍历图；否则，只能访问图中的一个连通分量。

2．以邻接矩阵存储的图的广度优先遍历算法实现

在以邻接矩阵存储的图 Graph1 **类中，增加** breadthfs( )**方法实现图的广度优先遍历算法。** breadthfs( )**方法如下：**

```
import ds_java.Queue2;                          //链式队列，元素类型为 int
   public void breadthFirstSearch()             //图的广度优先遍历
   {                                            //k 为起始结点序号
      System.out.println("广度优先遍历 Breadth first search:");
      for(int k=1;k<=n;k++)
      {
         breadthfs(k);
         System.out.println();
         unvisited();
      }
   }
   public void breadthfs(int k)                 //从结点 k 开始的广度优先遍历
   {                                            //k 为起始结点序号
      Queue2 q2=new Queue2();                   //设置空队列
      int i=k-1,j=0;
      System.out.print("  v"+k+"->");           //访问起始结点
      visited[i]=1;                             //设置访问标记
      q2.enqueue(k);                            //访问过的结点 k 入队
      while(!q2.isEmpty())                      //队列不空时
      {
         k=q2.dequeue();                        //出队
         i=k-1;                                 //i 是结点 k 的数组下标
         j=0;
         while(j<n)                             //查找与 k 相邻的其他结点
            if((mat[i][j]==1)&&(visited[j]==0))
```

```
                {
                    k=j+1;
                    System.out.print("v"+k+"->");
                    visited[j]=1;
                    q2.enqueue(k);
                }
                else
                    j++;
            }
        }
```

在例 8.1 **的** main **方法中，增加下列调用图的广度优先遍历算法的语句：**

```
g2.breadthFirstSearch();
```

程序运行增加的结果如下：

```
广度优先遍历 Breadth first search:
v1 ->  v2 ->  v4 ->  v3 ->
v2 ->  v1 ->  v3 ->  v4 ->
v3 ->  v2 ->  v4 ->  v1 ->
v4 ->  v1 ->  v2 ->  v3 ->
```

用广度优先遍历图，由于每个被访问过的结点只进队列一次，因此对图的每个结点，breadthfs( )**方法中的循环只执行一次。对于有向图，每条弧〈**v_i,v_j**〉被检测一次，对于无向图每条边(**v_i, v_j**)被检测两次。**

3．**以邻接表存储的图的广度优先遍历算法实现**

在以邻接表存储的图 Graph2 **类中，增加** breadthfs( )**方法实现图的广度优先遍历算法。**breadthfs( )**方法将覆盖超类** Graph1 **中的同名方法。**breadthfs( )**方法如下：**

```
import ds_java.Queue2;                          //链式队列，元素类型为 int
    public void breadthfs(int k)                //图的广度优先遍历
    {                                           //k 为起始结点序号
        int i,j=0;
        Queue2 q2=new Queue2();                 //设置空队列
        OnelinkNode p;
        i=k;
        System.out.print("  v"+k+" ->");       //访问起始结点
        visited[i]=1;                           //设置访问标记
        q2.enqueue(i);                          //访问过的结点入队
        while(!q2.isEmpty())                    //队列不空时
        {
            i=q2.dequeue();                     //出队
            if(table[i]!=null)                  //查找有边相连的另一结点
            {
                p=table[i].next;
```

```
                while(p!=null)
                {
                    j=p.data;
                    if(visited[j]==0)
                    {
                        System.out.print("v"+j+"->");
                        visited[j]=1;
                        q2.enqueue(j);
                    }
                    else
                        p=p.next;
                }
            }
        }
    }
```

对于有向图 G_7，程序运行的结果如下：

```
广度优先遍历 Breadth first search:
v1 ->  v2 ->   v3 ->   v4 ->
v2 ->  v3 ->   v4 ->   v1 ->
v3 ->
v4 ->  v1 ->   v3 ->   v2 ->
```

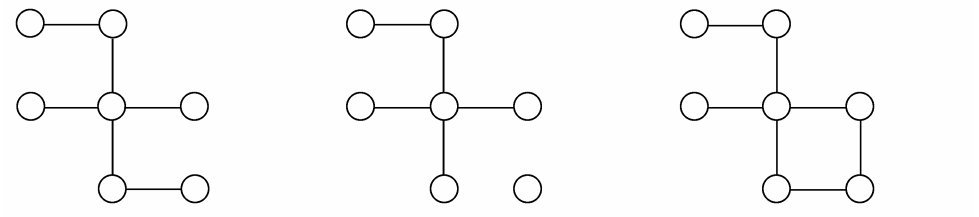
8.4 最小代价生成树

在讨论图的生成树之前，我们首先从图的角度来看树的基本概念。

8.4.1 树与图

连通的无回路的无向图称为无向树，简称树（tree）。树中的悬挂点又称为树叶（leaf），其他结点称为分支点（branched node）。诸连通分量均为树的图称为森林（forest），树是森林。

由于树无环也无重边（否则它有回路），因此树必定是简单图。若去掉树中的任意一条边，则树变为非连通图；若给树加上一条边，则形成图中的一条回路，如图 8.11 所示。



(a) 树　(b) 去掉一边成森林，非连通图　(c) 加上一边就不是树，而是有回路的图

图 8.11　树、森林与图

设图 T 为一棵树，其结点数为 n，边数为 m，那么 $n-m=1$。

【例 8.2】 以树结构描述测试假币的称重策略。

有 8 枚硬币，其中恰有一枚假币，假币比真币重。试用一架天平称出假币，使称重的次数尽可能地少。

图 8.12 描述了称重的策略。只要称两次便可测出假币。图中 8 个数字表示 8 枚硬币，结点处标记的两个集合为一次称量中两盘所放的硬币，假币一定在一次称量中较低的盘中。

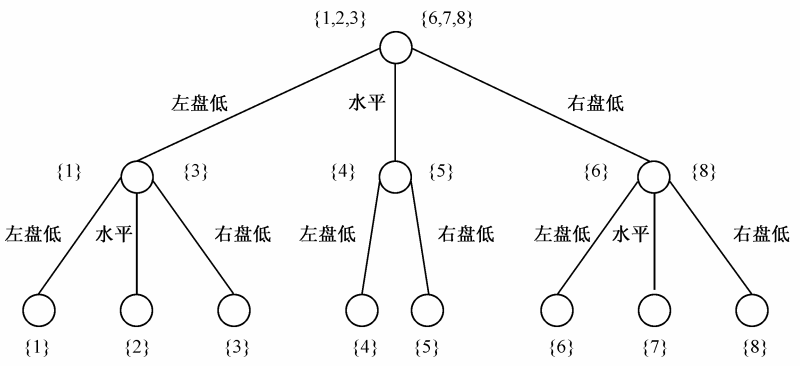


图 8.12　以树结构描述测试假币的称重策略

8.4.2　生成树

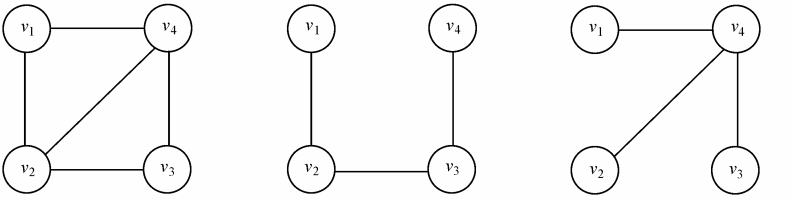
由树的特性可知，一个连通图 G 的“极小连通生成子树”应当是一棵树。

1．生成树的定义

如果图 T 是无向图 G 的生成子图，且 T 是树，则图 T 称为图 G 的生成树（spanning tree）。图 G 的生成树 T 包含 G 中的所有结点和尽可能少的边。任意一个连通图都至少有一棵生成树。

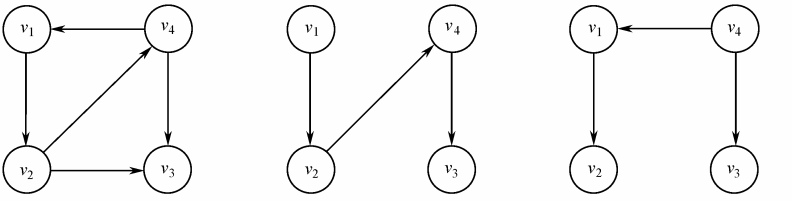
设图 $G=(V, E)$ 是一个连通的无向图，从 G 中的任意一个结点 v_0 出发进行一次遍历所经过的边的集合为 TE，则 $T=(V, \mathrm{TE})$ 是 G 的一个连通子图，即得到 G 的一棵生成树。

以深度优先遍历得到的生成树，称为深度优先生成树；以广度优先遍历得到的生成树，称为广度优先生成树。无向图的生成树如图 8.13 所示，有向图的生成树如图 8.14 所示。



(a) 无向图G_6　(b) 从v_1出发的深度优先生成树　(c) 从v_4出发的广度优先生成树

图 8.13　无向图 G_6 及其生成树



(a) 有向图G_7　(b) 从v_1出发的深度优先生成树　(c) 从v_4出发的广度优先生成树

图 8.14　有向图 G_7 及其生成树

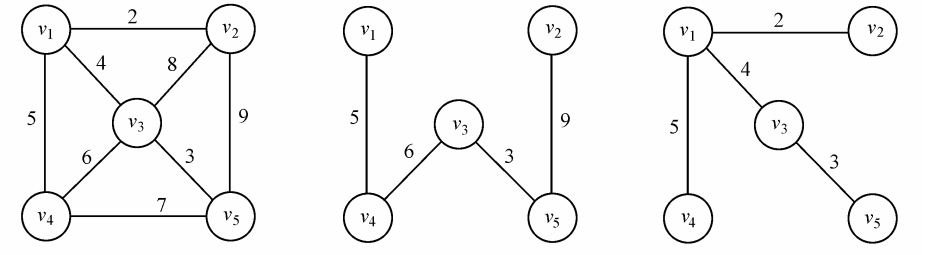
图的生成树不是惟一的，从不同的结点出发遍历可以得到不同的生成树。具有 n 个结点的连通无向图的生成树有 n 个结点和 $n-1$ 条边。对图的任意两个结点 v_i 和 v_j，在生成树中，v_i 和 v_j 之间只有惟一的一条路径。如果在生成树中加入一条边，则产生回路；如果删除生成树中的一条边，生成树将被分成两棵树。

2．生成森林

如果 G 是非连通的，则进行一次搜索只能遍历图的一个连通分支，各连通分支的生成树组成 G 的生成森林（spanning forest）。

3．带权图的生成树

带权图 G_8 及其深度优先、广度优先生成树如图 8.15 所示。



(a) 带权图G_8　　(b) 深度优先生成树　　(c) 广度优先生成树

图 8.15　带权图 G_8 及其生成树

一个带权图的生成树中，各边的权值之和称为生成树的代价（cost）。一般地，一个连通图的生成树不止一棵，图 8.15 中两棵生成树的代价分别为 23 和 14。

8.4.3　最小代价生成树

设 G 是一个连通的带权图，$w(e)$为边 e 上的权，T 为 G 的生成树，那么 T 中各边权之和为

$$w(T)=\sum_{e\in T} w(e)$$

上式称为生成树 T 的权，也称为生成树的代价（cost）。权最小的生成树称为最小生成树或最小代价生成树（minimum cost spanning tree）。

例如，设有一个通信网络图，图中结点表示城市，边表示连接两个城市的通信线路，边上的权表示相应的代价，则该图的一棵最小代价生成树给出连接每个城市的具有最小代价的通信网络线路。

按照生成树的定义，n 个结点的连通图的生成树有 n 个结点 $n-1$ 条边。因此，构造最小生成树的准则有 3 条：

- 必须只使用该图中的边来构造最小生成树。
- 必须使用且仅使用 $n-1$ 条边来连接图中的 n 个结点。
- 不能使用产生回路的边。

构造最小生成树有两种典型的算法：克鲁斯卡尔（Kruskal）算法和普里姆（Prim）算法，它们都采用了以下一种逐步求解的策略：

设连通带权图 $G=(V, E)$有 n 个结点和 m 条边。最初先构造一个包括全部 n 个结点和

条边的森林 $T=\{T_1, T_2,\cdots, T_n\}$，以后每一步向 T 中加入一条边，它应当是一端在 T 的某一棵树 T_i 上、而另一端不在 T_i 上的所有边中具有最小权值的边。由于边的加入，使 T 中的某两棵树合并为一棵，树的棵数减 1。经过 $n-1$ 步，最终得到一棵有 $n-1$ 条边的最小代价生成树。

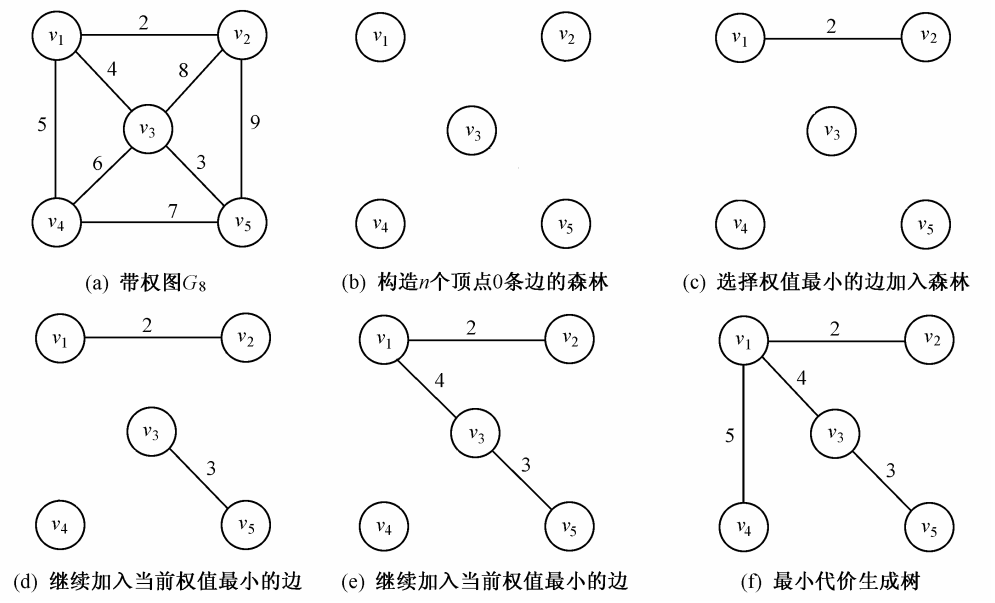
1．Kruskal 算法

Kruskal 算法的基本思想是，依照边的权从小到大的次序，逐边将它们放回到所关联的结点上，但删除会生成回路的边，直至产生一个 $n-1$ 条边的生成树。

Kruskal 算法描述如下：

- 构造 n 个结点和 0 条边的森林。
- 选择权值最小的边加入森林。
- 重复上一步，使森林中不产生回路，直到该森林变成一棵树为止。

以 Kruskal 算法构造连通带权图的最小生成树如图 8.16 所示。



(a) 带权图G_8　(b) 构造n个顶点0条边的森林　(c) 选择权值最小的边加入森林

(d) 继续加入当前权值最小的边　(e) 继续加入当前权值最小的边　(f) 最小代价生成树

图 8.16　以 Kruskal 算法构造连通带权图的最小生成树

构造最小生成树时，为使得生成树的权最小，每一步应选择权值尽可能小的边，所以从权值最小的边开始，选择 $n-1$ 条权值较小的边构成无回路的生成树。但是，并非每一条当前权值最小的边都可选。

2．Prim 算法

Prim 算法的基本思想是，从连通带权图 G 的某个结点 v_0 出发，选择与它相关的具有最小权值的边（v_0, v_i），将该边与结点 v_i 加入到生成树 T 中，直至产生一个 $n-1$ 条边的生成树。

以 Prim 算法构造连通带权图的最小生成树如图 8.17 所示。

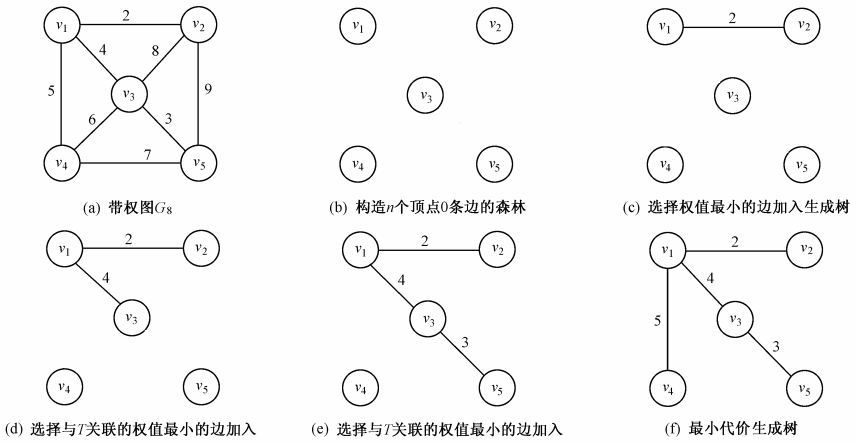


图 8.17　以 Prim 算法构造连通带权图的最小生成树

8.5　最短路径

设图 $G=(V, E)$ 是一个带权图，如果图中从结点 v 到结点 w 的一条路径为 $(v, v_1, \cdots v_i, w)$，其路径长度不大于从 v 到 w 的所有其他路径的路径长度，则该路径是从 v 到 w 的最短路径（shortest path**），v 称为源点，w 称为终点。**

1．单源最短路径

若给定一个带权图 G 与源点 v，限定各边上的权值大于或等于 0，求从 v 到 G 中其他结点的最短路径称为边上权值非负情形的单源最短路径问题。

例如，对于图 8.17（a）中的带权图，其邻接矩阵为

$$\boldsymbol{A}_8=\begin{bmatrix}0 & 2 & 4 & 5 & \infty\\ 2 & 0 & 8 & \infty & 9\\ 4 & 8 & 0 & 6 & 3\\ 5 & \infty & 6 & 0 & 7\\ \infty & 9 & 3 & 7 & 0\end{bmatrix}$$

以 v_1 为源点的单源最短路径如表 8-1 所示。

表 8-1　以 v_1 为源点的单源最短路径

源　点	终　点	路　径	路径长度	最短路径
v_1	v_2	(v_1, v_2)	2	✓
		(v_1, v_3, v_2)	12	
	v_3	(v_1, v_3)	4	✓
		(v_1, v_2, v_3)	10	
	v_4	(v_1, v_4)	5	✓
		(v_1, v_3, v_4)	10	
	v_5	(v_1, v_2, v_5)	11	
		(v_1, v_3, v_5)	7	✓
		(v_1, v_4, v_5)	12	

2．所有结点之间的最短路径

求每一对结点之间的最短路径的方法是，依次将图 G 中的每个结点作为源点，求每个结点的单源最短路径。

例如，对于图 8.17（a）中的带权图，其所有结点间的最短路径如表 8-2 所示。

表 8-2　最短路径

源　点	终　点	最短路径	路径长度
v_1	v_2	(v_1, v_2)	2
	v_3	(v_1, v_3)	4
	v_4	(v_1, v_4)	5
	v_5	(v_1, v_3, v_5)	7
v_2	v_3	(v_2, v_1, v_3)	6
	v_4	(v_2, v_1, v_4)	7
	v_5	(v_2, v_5)	9
v_3	v_4	(v_3, v_4)	6
	v_5	(v_3, v_5)	3
v_4	v_5	(v_4, v_5)	7

习　题　8

8.1　对于图 8.18（a）中的无向带权图 G_9，请给出：

（1）图中每个结点的度。

（2）图的邻接矩阵。

（3）从任一结点出发，进行深度优先和广度优先遍历所得到的结点序列和边的序列。

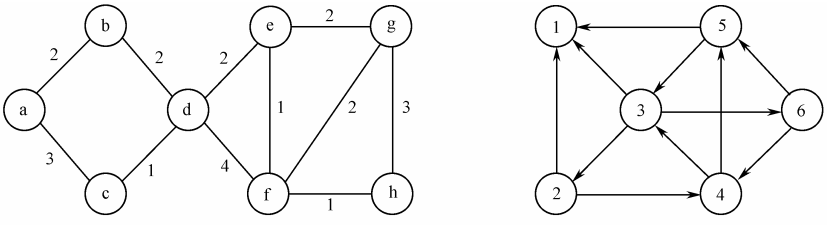
（4）以 a 为起点的一棵深度优先生成树和一棵广度优先生成树。

（5）以 d 为起点的一棵深度优先生成树和一棵广度优先生成树。

（6）分别以克鲁斯卡尔（Kruskal）算法和普里姆（Prim）算法构造最小生成树。

（7）图中所有结点间的最短路径。

8.2　对于图 8.18（b）中的有向图 G_{10}，请给出：



(a) 无向图G_9　　(b) 有向图G_{10}

图 8.18　无向图与有向图

（1）图中每个结点的度、入度和出度。

（2）图的邻接矩阵。

（3）图的邻接表。

（4）图的强连通分支。

（5）从任一结点出发，进行深度优先和广度优先遍历所得到的结点序列和边的序列。

（6）一棵深度优先生成树和一棵广度优先生成树。

（7）图中所有结点间的最短路径。

实　习　8

1．实习目的：掌握图的概念、存储与遍历。

2．题意：以邻接表存储带权图，并对图进行遍历。

第 9 章　综合应用设计

数据结构是计算机学科本科学生必修的一门专业基础课，它是培养学生软件设计能力的一门重要课程。

数据结构设计和算法设计是软件系统设计的核心。数据结构课程讨论的知识内容是软件设计的理论基础，数据结构课程介绍的技术方法是软件设计中使用的基本方法。

在前述各章中，我们以线性结构、树结构到图结构为主线，从数据的各种逻辑结构、在计算机中的存储结构以及各种操作的算法设计等角度，全面阐述了数据结构的基本理论。

数据结构课程不仅要注重理解基本知识，更要培养软件设计的基本技能，实践性环节是巩固所学理论知识、使理论与实际相结合的一项必不可少的重要环节。除了前述各章中安排的实验外，大多数学校还安排了课程设计。通过课程设计的训练，学生将体会软件设计的全过程，编写出高效率的应用程序，培养掌握数据处理的能力，提高软件设计的能力，从而为今后进行系统软件和应用软件的开发打下坚实的基础。

本章首先以求解骑士游历问题作为一个课程设计范例，描述从建立抽象数据结构模型、算法分析与设计、算法实现、程序动态调试到程序全方位静态测试的全过程，并实现软件设计。其后，给出若干综合应用实习题作为课程设计的选题。

本章设计为课程设计一周。

9.1　用“预见算法”解骑士游历问题

骑士游历问题是一个典型的趣味程序设计题。本节在常规算法的基础上，提出一个新的算法——“预见算法”，该算法能大幅提高程序的运行效率。

1．题意

在国际象棋的棋盘（8 行×8 列）上放置一个马，按照“马走日字”的规则，马要遍历棋盘，即到达棋盘上的每一格，并且每格只到达一次。若给定起始位置（x_0, y_0），编程探索出一条路径，沿着这条路径马能遍历棋盘上的所有单元格，这就是“骑士游历”问题。

设当前马在棋盘的某一位置（x, y）上，按照规则，下一步有 8 个方向可走，如图 9.1 所示。

	1	2	3	4	5	6	7	8
1								
2								
3			8		1			
4		7				2		
5				马				
6		6				3		
7			5		4			
8								

图 9.1　马下一步可走的 8 个方向

2．棋盘的存储结构

设二维数组 mat 表示棋盘，每个元素表示棋盘的一格，其值定义如下：

$$mat[i,j]=\begin{cases}0 & \text{表示马未到达过}\\ -1 & \text{表示棋盘边界}\\ \text{自然数} & \text{表示马到达该格的步数}\end{cases}$$

为了简化程序，在定义数组时，将数组的规模定得比实际棋盘多 4 行 4 列。初始化时将第 －1、0 行及第 9、10 行都置成 －1；将第 －1、0 列及第 9、10 列都设置成 －1，表示这些格是棋盘的边界，马不能到达。

设初始位置为(1, 1)，一次成功的遍历如图 9.2 所示。

1	16	27	22	3	18	47	56
26	23	2	17	46	57	4	19
15	28	25	62	21	48	55	58
24	35	30	45	60	63	20	5
29	14	61	34	49	44	59	54
36	31	38	41	64	53	6	9
13	40	33	50	11	8	43	52
32	37	12	65	42	51	10	7

图 9.2　从(1, 1)开始的一次成功的遍历

3．常规的“回溯算法”

1）设计思想

马从棋盘上的某一初始位置(x_0, y_0)开始，每次选择一个方向 k，向前走一格，直到走完 64 格为止。每走一格，设置数组中相应格的元素值为马走的步数。算法流程如图 9.3 所示。

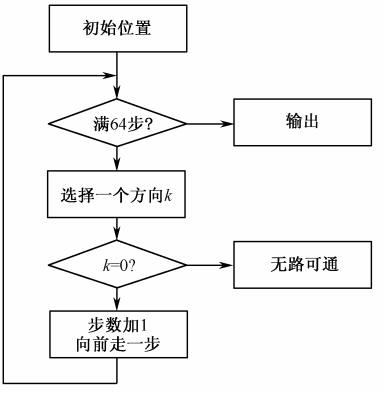


图 9.3　“回溯算法”流程图

2）辅助结构——栈

如果选择的方向 k=0，表示可能的 8 种走向都试探不通，不通的原因是走向超出了棋盘范围，或当前位置已经被马访问过。此时马已无路可走，说明前几步走得不对，应该退回去重新换一种走法。这时需要一个辅助的数据结构——栈，用以记录来时路径上每一步的有关数据，每当发现需要退回时，就可以从栈中取出必需的数据。这种逐步试探的设计思想称为“回溯算法”。

入栈的数据元素应该包括：从第几个方向走来的，当前位置所在行号和列号等。

3）性能评价

回溯算法在每一格上朝一个方向盲目地进行试探。遇到在某一格上所有方向都不能走通时，才回到前一格上来试探另一个方向。当每一格上的所有方向都试探过，不能走通时，才作出“走不通”的结论。因此该算法在探索时带有一定的盲目性和随机性，它只适用于探索空间不很大的情况，而在比较大的空间中使用是很费时的。若探索空间增大时，运行时间将按指数规律上升。

4.“预见算法”

1）设计思想

回溯算法的思路是可行的，但它的运行效率较低，原因在于每步试探的随机性和盲目性。如果能够找到一种克服这种随机性和盲目性的办法，按照一定规律选择前进的方向，则将增加成功的可能性，运行时间也大为缩短。本节提出的算法试图在这方面有所突破。

如果在每步选择方向时，不是任意选择一个方向，而是经过一定的测试和计算，“预见”每条路的“宽窄”，再选择一条最“窄”的路先走，成功的可能性较大。理由是先走“困难的路”，光明大道留在后面。因为每一格迟早都要走到，与其把困难留在后面，不如先走“困难的路”，这样路就会越走越宽，成功的机会就越大。这种方法称为启发式探索，不妨称它为“预见算法”。

2）实现手段

为每个方向测定一个值——可通路数，它表示该位置上还有多少条通路。在图 9.1 所示的棋盘上，马的位置在(5, 4)，下一步有 8 个方向可走，这 8 个方向的再下一步位置如何呢？表 9-1 表示测试出的该位置的可通路数情况。在每一格上对 8 个方向都进行试探，并分析比较，下一步应该选择可通路数值最小的方向走。

表 9-1　(5, 4)位置的可通路数情况

方向	下一位置	可通路数
1	(3,5)	7
2	(4,6)	7
3	(6,6)	7
4	(7,5)	5
5	(7,3)	5
6	(6,2)	5
7	(4,2)	5
8	(3,3)	7

3）性能评价

预见算法虽然在每一格上选择下一步的方向需要花费一定的时间，但它针对性强，减少了许多盲目的试探，总的选择次数少，从而缩短了运行时间。

【例 9.1】 用“预见算法”解骑士游历问题。

本例声明 Horse 类，成员变量 mat 以二维数组表示棋盘，show 表示是否显示中间结果，内部类 Position 中的成员变量 *x* 和 *y* 表示棋盘上一格的位置，*x* 和 *y* 从 1 开始计数。

构造方法中为 mat 数组分配空间，并初始化棋盘。get(Position p)方法获得 *p* 格的值，set(Position p, int i)设置 *p* 格的值为 *i*。isValid(Position p)方法判断 *p* 格是否在棋盘内且未被访过。goaStep(Position p，int k)方法由 *p* 格走到下一步。

play(int x, int y)方法从(*x*, *y*)位置开始遍历。算法流程如图 9.4 所示。

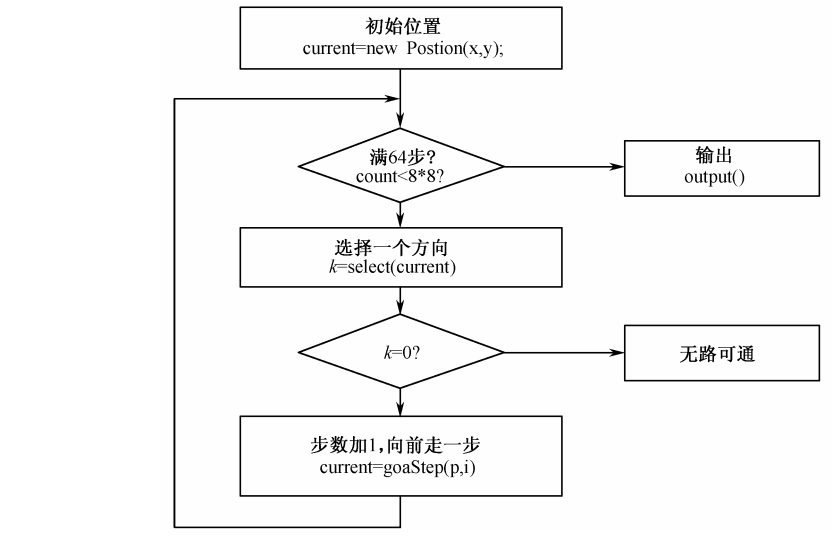


图 9.4 play()方法实现游历的算法流程

其中，select(Position p)方法用“预见算法”为位置 *p* 选择一个前进的方向 *k*。算法描述如下：

- 设位置 *p* 按方向 *i* 的下一位置是 next1，如果 next1 在棋盘内且未被访问过，则统计 next1 的 8 个方向的可通路数 road；否则 *i*++，进行下一个方向的测试。*i* 从 1 循环到 8。
- 对于每一个 next1 位置，设其按 *j* 方向的下一位置是 next2，如果 next2 在棋盘内且未被访问过，则该方向可通，next1 的可通路数 road 加 1。*j* 从 1 循环到 8。
- 在每一个 next1 位置的可通路数 road 中，选择最小值为 minroad，相应方向为 *k*，返回 *k*。

程序如下：

```
public class Horse
{
   private int mat[][];                    //二维数组表示棋盘
   boolean show;                           //是否显示中间结果
   class Position                          //内部类
   {
      int x,y;                             //表示棋盘上一格的位置
```

```
    Position(int x,int y)
    {
        this.x=x;
        this.y=y;
    }
    Position()
    {
        this(1,1);
    }
    Position(Position p)
    {
        this.x=p.x;
        this.y=p.y;
    }
}
public Horse(int n,int x,int y,boolean show)
{                          //数组比实际棋盘多顶边、底边各两行，左边、右边各两
    mat=new int[n+2*2][n+2*2];
    this.show=show;
    inition();                          //棋盘初始化
    play(x,y);
}
public Horse(int n,int x,int y)
{
    this(n,x,y,false);
}
public Horse(int n)
{
    this(n,1,1);
}
public Horse()
{
    this(8,1,1);
}
public void inition()                       //棋盘初始化
{
    int i,j;
    for(i=0;i<=1;i++)
    {
        for(j=0;j<mat.length;j++)           //顶边两行
            mat[i][j]=-1;
        for(j=0;j<mat.length;j++)           //底边两行
```

```
            mat[mat.length-1-i][j]=-1;
        for(j=0;j<mat.length;j++)          //左边两列
            mat[j][i]=-1;
        for(j=0;j<mat.length;j++)          //右边两列
            mat[j][mat.length-1-i]=-1;
    }
}
public int get(Position p)                 //获得 p 格的值
{
    return mat[p.x+1][p.y+1];
}
public void set(Position p,int i)          //设置 p 格的值为 i
{
    mat[p.x+1][p.y+1]=i;
}
public void play(int x,int y)              //从(x,y)格开始遍历
{
    int n=mat.length-4;                    //棋盘大小
    Position current=new Position(x,y);    //当前位置
    int count=1;                           //统计第几步,count 记录走过的格
    int i,j,k=1;
    while(count<=n*n&&k!=0)
    {
       set(current,count);
       if(this.show)
          System.out.print("第"+count+"步  ");
       k=select(current);                  //试探，选择一个方向
       if(k==0 && count<n*n)
          System.out.println("第"+count+"步无路可通!");
       else
       {
          count++;                         //步数加 1
          current=goaStep(current,k);      //向前走一步
       }
    }
}
public boolean isValid(Position p)         //判断 p 是否在棋盘内且未被访问过
{
    int n=mat.length-4;                    //棋盘大小
    if(p!=null && p.x>=1 && p.x<=n && p.y>=1 && p.y<=n && get(p)==0)
       return true;
    else
```

```
        return false;
    }
    public Position goaStep(Position p,int k) //返回 p 位置按 k 方向的下一位置
    {
        int x=p.x;
        int y=p.y;
        switch(k)
        {
            case 1: x-=2; y++;  break;
            case 2: x--;  y+=2; break;
            case 3: x++;  y+=2; break;
            case 4: x+=2; y++;  break;
            case 5: x+=2; y--;  break;
            case 6: x++;  y-=2; break;
            case 7: x--;  y-=2; break;
            case 8: x-=2; y--;  break;
        }
        return new Position(x,y);
    }
    public int select(Position p) //选择 p 位置到达下一位置 next1 应走的方向 k
       //试探 next1 的 8 个方向位置 next2 的可通路数 road,road 的最小值为 minroad
    {
        int i=0,j=0,k=0,road=0,minroad=8;
        Position next1=null,next2=null;
        if(this.show)
        {
            System.out.println("当前位置: ("+p.x+","+p.y+")");
            this.output();
            System.out.println("方向   下一位置   可通方向     可通路数");
        }
        for(i=1;i<=8;i++)          //试探位置 p(x,y)的 8 个方向的位置 next1
        {
            road=0;
            next1=goaStep(p,i);                 //next1 是 p 按 i 方向的下一位置
            if(isValid(next1))                  //next1 在棋盘内且未被访问过
            {
                if(this.show)
                    System.out.print(""+i+"\t("+next1.x+","+next1.y+")\t");
                for(j=1;j<=8;j++)               //统计 next1(x,y)的可通路数 roa
                {
                    next2=goaStep(next1,j);     //next2是next1按j 方向的下一位
                    if(isValid(next2))          //next2 在棋盘内且未被访问过
```

```
                    {
                        road++;
                        if(this.show)
                            System.out.print(j+",");
                    }
                }
                if(road<minroad)
                {
                    minroad=road;              //minroad 记载 road 的最小值
                    k=i;                       //k 记载 road 最小值的方向
                }
                if(this.show)
                    System.out.println("\t"+road);
            }
        }
        if(this.show)
            System.out.println("选定下一步方向 k="+k+"\r\n");
        return k;
    }
    public void output()                  //输出棋盘上所有位置的元素
    {
        int i,j,n=mat.length;
        for(i=0;i<n;i++)
        {
            for(j=0;j<n;j++)
            {
                if(mat[i][j]>=0&&mat[i][j]<10)
                    System.out.print("  "+mat[i][j]);
                else
                    System.out.print(" "+mat[i][j]);
            }
            System.out.println();
        }
        System.out.println();
    }
    public static void main(String args[])
    {
        Horse h1=new Horse(8,1,1,false);
        h1.output();
    }
}
```

起始位置在(1, 1)的运行结果如下：

```
-1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
-1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
-1  -1   1  16  27  22   3  18  47  56  -1  -1
-1  -1  26  23   2  17  46  57   4  19  -1  -1
-1  -1  15  28  25  62  21  48  55  58  -1  -1
-1  -1  24  35  30  45  60  63  20   5  -1  -1
-1  -1  29  14  61  34  49  44  59  54  -1  -1
-1  -1  36  31  38  41  64  53   6   9  -1  -1
-1  -1  13  40  33  50  11   8  43  52  -1  -1
-1  -1  32  37  12  39  42  51  10   7  -1  -1
-1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
-1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
```

使用下面的参数创建对象，可以在程序运行时查看每一步的中间结果，即马到达的位置及该位置下一步的可通路数表。

```
Horse h1=new Horse(8,1,1,true);
```

在 8×8 的棋盘上，每个初始位置都可走通。而在 5×5 的棋盘上，起始位置为(1，2)，至第 24 步时就走不通了。

由于篇幅所限，程序中没有用栈。一旦无路可通时，程序只给出相应的提示信息，没有再返回重新寻找一条新的路径。

5．进一步研究方向

- 程序运行时，随意设置棋盘的大小，最小为 5×5。
- 设置栈，在无路可通时，选择另一条路径。
- 设置队列，对于同一个初始位置，求得多条路径。
- 用图形描绘马在棋盘上的移动情况。
- 使用栈或队列时，跟踪程序运行，观察并描绘栈或队列的动态变化情况。

9.2 综合应用实习

本节列出多种综合应用实习题作为课程设计的选题。

1．求解骑士游历、迷宫等问题的多种算法

1）实习目的

掌握栈、队列的基本概念，熟练运用；掌握递归算法的设计思想。

2）题意

骑士游历、迷宫的题意参见第 4 章实习 4。

类似问题还有汉诺塔（Tower of Hanoi）问题和八皇后问题等。

① 汉诺塔（Tower of Hanoi）问题

传说婆罗门庙里有一个塔台，台上有 3 根用钻石做成的标号为 A, B 和 C 的柱子，在 A 柱上放着 64 个金盘，每一个金盘都比下面的略小一点。把 A 柱上的金盘全部移到 C 柱上的那一

天就是世界末日。移动的条件是，一次只能移动一个金盘，移动过程中大金盘不能放在小金盘上面。庙里的僧人一直在移个不停。因为全部移动次数是 $2^{64}-1$，如果每秒移动一次的话，需要 500 亿年。

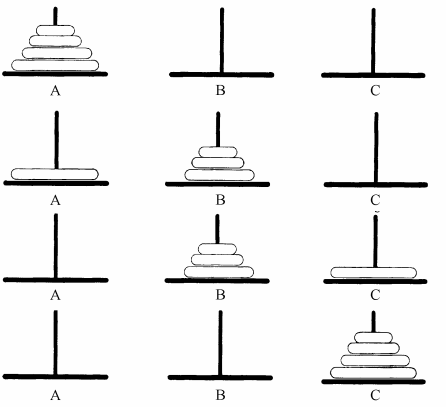


图 9.5　汉诺塔问题

② 八皇后问题

在国际象棋的棋盘（8 行×8 列）上，依次放置 8 个皇后，使其不在同一行、同一列和同一斜线上。图 9.6 显示八皇后的一种排列方式。

	1	2	3	4	5	6	7	8
1	Q							
2			Q					
3					Q			
4							Q	
5		Q						
6				Q				
7						Q		
8								Q

图 9.6　八皇后问题

3）实习要求

分别用以下算法实现：

- 递归算法。
- 使用栈作为辅助结构。
- 使用队列作为辅助结构（除汉诺塔问题）。

2．计算表达式值的多种算法

1）实习目的

掌握栈、递归算法、二叉树的设计技术。

2）题意

计算表达式的值。

3）实习要求

本书第 4 章例 4.4 中，先后使用两个栈计算表达式的值。还有其他方法，要求如下：

- 同时使用两个栈求值。

- 递归算法。
- 采用二叉树结构。

将表达式

$$1+2*(3-4)$$

建立一棵表达式二叉树，如图 9.7 所示。

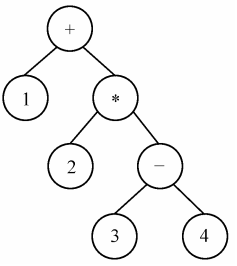


图 9.7　表达式二叉树

对表达式二叉树进行后根次序遍历，得到后缀表达式

1　2　3　4　−　*　+

以后根次序遍历表达式二叉树，求表达式的值。

3．利用线程比较多种查找、排序算法的运行时间

1）实习目的

利用线程技术，比较不同查找、排序算法的性能。

2）题意

- 将顺序、折半查找算法设计成线程，启动两个不同线程同时运行，并计算不同查找算法的运行时间。
- 将冒泡、快速等多个排序算法设计成线程，启动两个以上不同线程同时运行，并计算不同排序算法的运行时间。

4．管理信息系统中的算法设计

1）实习目的

以 Java 中的流技术，练习管理信息系统中常用的算法设计。

2）题意

以学生管理信息系统为例：

- 设计 Student 类的增加、删除、修改、查询、统计、自动编号等功能。
- 设计系统管理员、班主任、任课教师、学生、普通用户等多级用户管理权限。
- 设计系别、专业、班级等字典库，并进行维护。

5．经典问题求解

1）实习目的

综合运用所学知识，设计新算法并实现。

2）题意

对于以下的经典问题，设计相应的算法：

- 电梯调度算法。
- 自动排课算法。
- 电话号码簿的设计与查找算法。
- 数据字典的设计与查找算法。
- 城市道路交通网络的构架设计。

6．数据结构的算法设计与动态描述

1）实习目的

对常用数据结构和经典算法进行动态描述。将数据结构用图形方式显示在屏幕上，并对插入、删除等操作的每一步状态（当前结点等）进行动态演示。

2）题意

- 线性表：单向链表、双向链表的插入与删除操作，约瑟夫环问题，多项式相加。
- 排序：单向链表的简单选择排序（不新建链表）、归并排序，顺序表的希尔排序、快速排序、堆排序、归并排序的排序过程动态演示，九宫排序的排序过程动态演示，各种排序算法的比较演示。
- 栈与队列：计算表达式值时的栈、解素数环问题时的队列的动态演示。
- 稀疏矩阵与广义表：稀疏矩阵三元组的顺序存储结构、链式存储结构，广义表的存储结构。
- 树和二叉树：二叉树的建立与 3 种遍历算法（先根、中根、后根），表达式二叉树的建立与计算，广义表描述的二叉树的建立与遍历，线索二叉树的建立与遍历；二叉树中求两个结点最近的共同祖先；树与二叉树的转换。
- 查找：顺序链表的顺序查找算法，有序顺序表的折半查找算法，动态查找表的插入、删除及查找算法，二叉排序树的建立与查找算法，哈希表的插入、删除及查找算法，各种查找算法的比较。
- 图：图的两种存储结构和两种遍历算法（深度优先、广度优先），最小代价生成树，最短路径。

附录 A　ASCII 码表

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	NUL (^@)	32	20		64	40	@	96	60	‘
1	01	SOH (^A)	33	21	!	65	41	A	97	61	a
2	02	STX (^B)	34	22	"	66	42	B	98	62	b
3	03	ETX (^C)	35	23	#	67	43	C	99	63	c
4	04	EOT (^D)	36	24	$	68	44	D	100	64	d
5	05	ENQ (^E)	37	25	%	69	45	E	101	65	e
6	06	ACK (^F)	38	26	&	70	46	F	102	66	f
7	07	BEL (^G)	39	27	’	71	47	G	103	67	g
8	08	BS (^H)	40	28	(	72	48	H	104	68	h
9	09	HT (^I)	41	29	)	73	49	I	105	69	i
10	0A	LF (^J)	42	2A	*	74	4A	J	106	6A	j
11	0B	VT (^K)	43	2B	+	75	4B	K	107	6B	k
12	0C	FF (^L)	44	2C	,	76	4C	L	108	6C	l
13	0D	CR (^M)	45	2D	-	77	4D	M	109	6D	m
14	0E	SO (^N)	46	2E	.	78	4E	N	110	6E	n
15	0F	SI (^O)	47	2F	/	79	4F	O	111	6F	o
16	10	DLE (^P)	48	30	0	80	50	P	112	70	p
17	11	DC1 (^Q)	49	31	1	81	51	Q	113	71	q
18	12	DC2 (^R)	50	32	2	82	52	R	114	72	r
19	13	DC3 (^S)	51	33	3	83	53	S	115	73	s
20	14	DC4 (^T)	52	34	4	84	54	T	116	74	t
21	15	NAK (^U)	53	35	5	85	55	U	117	75	u
22	16	SYN (^V)	54	36	6	86	56	V	118	76	v
23	17	ETB (^W)	55	37	7	87	57	W	119	77	w
24	18	CAN (^X)	56	38	8	88	58	X	120	78	x
25	19	EM (^Y)	57	39	9	89	59	Y	121	79	y
26	1A	SUB (^Z)	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

附录 B　Java 关键字表

abstract	声明抽象类、抽象方法
boolean	声明一个布尔类型变量
break	中断一个循环，中止一个 switch 分支
byte	声明一个字节类型变量
case	创建一个 switch 分支
catch	捕获一个异常分支
char	声明一个字符类型变量
class	声明一个类
continue	进入下一个循环
default	创建一个默认的 switch 分支
do	创建一个 do 循环
double	声明一个双精度浮点数类型变量
else	创建一个 else 子句，给 if 语句提供一个可替换的路径
extends	从另一个类派生一个类
false	boolean 类型常量，“假”值
final	声明一个常量，声明一个最终类、最终方法
finally	捕获异常之后执行的语句
float	声明一个单精度浮点数类型变量
for	创建一个 for 循环
if	有条件地执行语句
implements	实现一个接口
import	导入一个包或类
instanceof	获得运行时对象信息
int	声明一个整数类型变量
interface	声明一个接口
long	声明一个长整数类型变量
native	声明一个本地方法
new	给对象分配内存空间
null	空值，引用类型常量
package	声明一个包
private	声明一个私有的成员
protected	声明一个保护的成员
public	声明一个公有的类、公有的成员

return	从一个方法返回，从一个方法返回一个值
short	声明一个短整数类型变量
static	声明类成员，或称静态成员
super	引用一个超类成员，调用超类的构造方法
switch	创建一个多分支语句
synchronized	声明一个同步方法
this	引用对象，在另一个构造方法中调用构造方法
throw	抛出一个异常
throws	指定一个方法可能抛出一个异常
transient	声明一个临时变量
true	boolean 类型常量，“真”值
try	封装可能抛出异常的代码
void	声明一个没有返回值的方法
while	创建一个 while 循环

附录 C　Java 部分类库表

本附录列出 java.lang、java.util、java.io **等包中的常用类及其方法。**

C.1　java.lang 包

C.1.1　Object 类

1）**类声明**

class Object

2）**构造方法**

public **Object**()

3）**实例方法**

public final Class **getClass**()　　//**返回该对象的类**

public boolean **equals**(Object obj)　　//**比较该对象与其他对象是否相等**

protected Object **clone**()　　//**创建并返回该对象的一个复制**

public String **toString**()　　//**返回该对象的字符串**

protected void **finalize**() throws Throwable　　//**对象的析构方法**

C.1.2　数据类型包装类

对应 Java **的** 8 **个基本数据类型，共有** 8 **个数据类型包装类：**Byte, Short, Integer, Long, Float, Doub Character **和** Boolean**。其中，**Byte, Short, Integer, Long, Float **和** Double **是数值类** Number **的子类。**

1．Integer 类

1）Integer **类层次结构**

```
java.lang.Object
  |
  +-java.lang.Number
        |
        +-java.lang.Integer
```

2）**类声明**

public final class **Integer** extends Number implements Comparable

3）**常量**

public static final int **MIN_VALUE**

public static final int **MAX_VALUE**

public static final Class **TYPE**

4）**构造方法**

public **Integer**(int value)

public **Integer**(String s) throws NumberFormatException

5）实例方法

public int **intValue**()　　//对象取值方法

public static int **parseInt**(String s) throws NumberFormatException　//字符串转成 int

其他数值类 Byte、Short、Long、Float 和 Double 的类层次与 Integer 相同，构造方法、基本操作也类似从略。

2．Boolean 类

1）类声明

public final class **Boolean** extends Object implements Serializable

2）常量

public static final Boolean **FALSE**

public static final Boolean **TRUE**

public static final Class **TYPE**

3）构造方法

public **Boolean**(boolean value)

public **Boolean**(String s)

3．Character 类

1）类声明

public final class **Character** extends Object implements Serializable, Comparable

2）构造方法

public **Character**(char value)

3）实例方法

public char **charValue**()　　//取值

public static char **toLowerCase**(char ch)　　//转换成小写字母

public static char **toUpperCase**(char ch)　　//转换成大写字母

C.1.3　String 类

1）类声明

public final class **String** extends Object implements Serializable, Comparable

2）构造方法

public **String**()

public **String**(String value)

public **String**(char[] value)

public **String**(char[] value，int offset，int count)

public **String**(byte[] bytes)

public **String**(byte[] bytes，int offset，int length)

3）实例方法

public int **length**()　　//返回字符串的长度

public String **concat**(String str)　　//连接字符串

4）比较字符串方法

public boolean **equals**(Object obj)　　//比较字符串是否相等

public boolean **equalsIgnoreCase** (String s) //**比较字符串，忽略字母大小写**

public int **compareTo**(String str) //**比较字符串的大小**

public int **compareToIgnoreCase**(String str) //**比较字符串，忽略字母大小写**

5）**求子串方法**

public String **substring**(int beginIndex) //**返回字符串从** beginIndex **开始的子串**

public String **substring**(int beginIndex，int endIndex)

//**返回从** beginIndex **开始至** endIndex **的子串**

public boolean **startsWith**(String prefix) //**判断是否以子串** prefix **开头**

public boolean **endsWith**(String suffix) //**判断是否以子串** suffix **结尾**

6）**查找与替换方法**

public char **charAt**(int index) //**返回** index **指定位置的字符**

public int **indexOf**(String str) //**返回** str **在字符串中第一次出现的位置**

public int **indexOf**(String str，int fromIndex) //**返回从** fromIndex **开始** str **在字符串中第一次出现的位置**

public int **lastIndexOf**(String str，int fromIndex) //**返回** str **在字符串中最后一次出现的位置**

public String **replace**(char oldc，char newc) //**以** newc **字符替换串中所有** oldc **字符**

7）**转换方法**

public String **toLowerCase**() //**全部转换成小写字母**

public String **toUpperCase**() //**全部转换成大写字母**

public String **toString**() //**返回其他类对象的字符串形式**

public String **trim**() //**删除字符串中所有空格**

C.1.4 Math **类**

1）**类声明**

public final class **Math** extends Object

Math **类没有构造方法，其中常量和方法都是静态的。**

2）**常量**

public static final double **PI**

public static final double **E**

3）**求绝对值方法**

public static double **abs**(double a)

public static float **abs**(float a)

public static int **abs**(int a)

public static long **abs**(long a)

4）**最大值与最小值**

public static double **max**(double a，double b)

public static float **max**(float a，float b)

public static int **max**(int a，int b)

public static long **max**(long a，long b)

public static double **min**(double a，double b)

public static float **min**(float a，float b)

public static int **min**(int a，int b)

public static long **min**(long a,long b)

5）**随机数与平方根**

public static double **random**() //**返回一个** 0.0~1.0 **之间的随机数**

public static double **sqrt**(double a) //**返回一个浮点数的平方根值**

6）**三角函数**

public static double **sin**(double a) //**返回一个浮点数的正弦值**

public static double **cos**(double a) //**返回一个浮点数的余弦值**

public static double **tan**(double a) //**返回一个浮点数的正切值**

public static double **acos**(double a) //**返回一个浮点数的反余弦值**

public static double **asin**(double a) //**返回一个浮点数的反正弦值**

public static double **atan**(double a) //**返回一个浮点数的反正切值**

7）**指数与对数函数**

public static double **exp**(double a) //**返回一个浮点数的以** e **为底的指数值**

public static double **log**(double a) //**返回一个浮点数的以** e **为底的对数值**

8）**转换函数**

public static long **round**(double a) //**将** double **浮点数转换成** long **值**

public static int **round**(float a) //**将** float **浮点数转换成** int **值**

C.1.5 System **类**

1）**类声明**

public final class **System** extends Object

System **类没有构造方法，其中常量和方法都是静态的。**

2）**常量**

public static final InputStream **in**

public static final PrintStream **out**

public static final PrintStream **err**

3）**静态方法**

public static viod **arraycopy**(Object src，int src_pos，Object dst，int dst_pos，int length)

//**复制一个数组**

public static void **exit**(int status) //**结束当前运行的程序**

C.1.6 Class **类**

1）**类声明**

public final class **Class** extends Object implements Serializable

Class **类没有构造方法。**

2）**实例方法**

public String **getName**() //**返回当前类名字符串**

public Class **getSuperclass**() //**返回当前类的超类**

C.1.7 异常处理类

错误和异常处理类有：Throwable, Exception **和** Error。

1）**类声明**

public class **Throwable** extends Object implements Serializable

public class **Error** extends Throwable

public class **Exception** extends Throwable

2）**构造方法**

public **Throwable**()

public **Throwable**(String message)

public **Error**()

public **Error**(String s)

public **Exception**()

public **Exception**(String s)

3）Throwable **类实例方法**

public String **getMessage**()　　//**获得异常信息**

public void **printStackTrace**()　　//**显示异常栈跟踪信息**

C.2 java.io 包

1．InputStream 类

1）**类声明**

public abstract class **InputStream** extends Object

2）**构造方法**

public **InputStream**()

3）**实例方法**

public abstract int **read**() throws IOException //**返回读入的一个字节**

public int **read**(byte[] b) throws IOException //**读入的多个字节返回缓冲区** b **中**

public int **read**(byte[] b，int off，int len)　throws IOException

public void **close**() throws IOException

2．OutputStream 类

1）**类声明**

public abstract class **OutputStream** extends Object

2）**构造方法**

public **OutputStream**()

3）**实例方法**

public abstract void **write**(int b) throws IOException

public void **write**(byte[] b) throws IOException

public void **write**(byte[] b，int off，int len) throws IOException

public void **flush**() throws IOException

public void **close**() throws IOException

3．FileInputStream 类

1）**类声明**

public class **FileInputStream** extends InputStream

2）**构造方法**

public **FileInputStream**(String name) throws FileNotFoundException

public **FileInputStream**(File file) throws FileNotFoundException

4．FileOutputStream 类

1）**类声明**

public class **FileOutputStream** extends OutputStream

2）**构造方法**

public **FileOutputStream**(String name) throws FileNotFoundException

public **FileOutputStream**(File file) throws FileNotFoundException

public **FileOutputStream**(String name，boolean append)
throws FileNotFoundException

5．PrintStream 类

1）**类声明**

public class **PrintStream** extends FilterOutputStream

2）**构造方法**

public **PrintStream**(OutputStream out)

public **PrintStream**(OutputStream out，boolean autoFlush)

3）**实例方法**

public void **print**(long l)

public void **println**()

参 考 文 献

1 许卓群等. 数据结构. 北京：高等教育出版社，1987.5

2 陈本林等编著. 数据结构. 南京：南京大学出版社，1998.6

3 严蔚敏等编著. 数据结构及应用算法教程. 北京：清华大学出版社，2001.2

4 殷人昆等编著. 数据结构（用面向对象方法与 C＋＋描述）. 北京：清华大学出版社，1999.7

5 王元元等编著. 离散数学导论. 北京：科学出版社，2002.2

6 赵致琢著. 计算科学导论. 北京：科学出版社，2000.2

7 Clifford A.Shaffer 著，张铭译. 数据结构与算法分析(Java 版). 北京：电子工业出版社，2001.2

8 傅清祥等编著.算法与数据结构（第二版）.北京：电子工业出版社，2001.8

9 Duane A.Bailey. Java™ Structures——Data Structures in Java for the Principled Programmer. 影印版. 北京：清华大学出版社，1999.12

10 Robert.Kruse. Data Structures and Program Design in C. 影印版. 北京：清华大学出版社，1998.7

阅 读 推 介

“数据结构”课程是计算机及其相关专业的一门核心必修课。本教材系统、全面地阐述了数据结构的基本理论，详细介绍了线性表、栈、队列、串、数组、广义表、树、二叉树、图等基本的数据结构以及查找和排序算法。全书内容取材恰当，组织合理，理论叙述准确、清楚，文字流畅。

“数据结构”课程既是一门理论课程，又是一门锻炼程序设计能力的实践课程。本教材既注重基本知识的传授，又注重对读者基本技能的培养。全书精心安排了大量有特色的例题和完整的实例程序，体现出作者丰富的教学经验和理论研究能力。

计算机软件开发方法是不断发展的，“数据结构”课程的内容也应随着软件开发方法的不断发展而发展。目前，面向对象的软件分析和设计技术已发展成为软件开发的主流方法，因此，用面向对象的方法描述数据结构就成为“数据结构”课程改革的必然。

用面向对象的观点描述具体的数据结构问题时，首先涉及选用哪种面向对象的高级语言的问题。

Java 语言是一种完全面向对象的语言。Java 语言具有平台无关性、分布式、可移植性、可重用性、健壮性、安全性等特点，这些优点是诸多传统程序设计语言所无法比拟的。所以 Java 语言能够在众多程序设计语言中脱颖而出，成为当今广泛使用的主流语言之一。

Java 语言继承了 C++语言的优秀特性，语法上类似 C++，易于学习。Java 语言比 C++语言简洁、精炼，类之间的继承关系简明清晰，程序的可读性更好。

采用 Java 语言既能够准确描述基本数据结构，充分表达算法的设计与实现，又能够进一步以面向对象的软件设计思想锻炼程序设计能力。所以，采用 Java 语言描述数据结构是一种发展趋势。

目前，除部分译著外，由国内作者编写的同类教材并不多见。

本教材采用 Java 语言以面向对象方法设计并实现了全部的数据结构及算法，具有一定的创新性、前瞻性，体现出作者扎实的理论功底和应用设计能力。

本教材适用于高等学校计算机及相关专业的本、专科学生。

2004 年 3 月