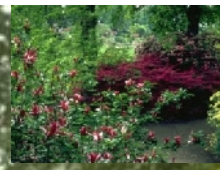


目 录

第1章 绪 论

1.1 数据结构的基本概念和术语

1.1.1 引例

首先分析学籍档案类问题。设一个班级有50个学生，这个班级的学籍表如表1.1所示。

表1.1 学 籍 表

序号	学号	姓名	性别	英语	数学	物理
01	20030301	李明	男	86	91	80
02	20030302	马琳	男	76	83	85
50	20030350	刘薇薇	女	88	93	90

我们可以把表中每个学生的信息看成一个记录，表中的每个记录又由7个数据项组成。该学籍表由50个记录组成，记录之间是一种顺序关系。这种表通常称为线性表，数据之间的逻辑结构称为线性结构，其主要操作有检索、查找、插入或删除等。

又如，对于学院的行政机构，可以把该学院的名称看成树根，把下设的若干个系看成它的树枝中间结点，把每个系分出的若干专业方向看成树叶，这样就形成一个树型结构，如图1.1所示。

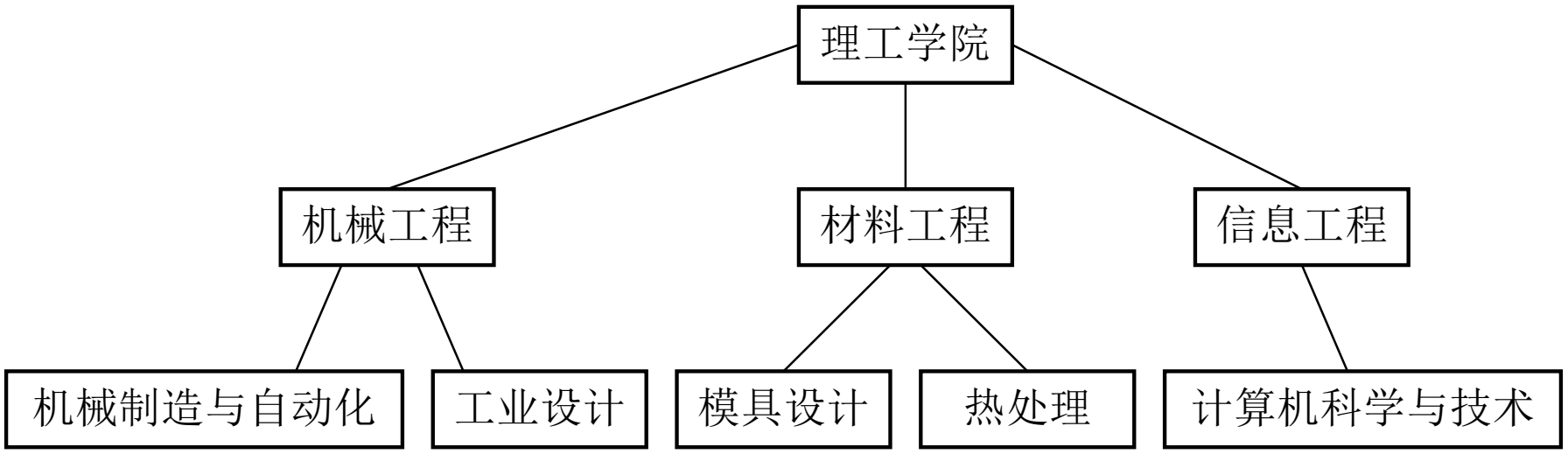


图1.1 专业设置

树中的每个结点可以包含较多的信息，结点之间的关系不再是顺序的，而是分层、分叉的结构。树型结构的主要操作有遍历、查找、插入或删除等。

最后分析交通问题。如果把若干个城镇看成若干个顶点，再把城镇之间的道路看成边，它们可以构成一个网状的图(如图1.2所示)，这种关系称为图型结构或网状结构。在实际应用中，假设某地区有5个城镇，有一调查小组要对该地区每个城镇进行调查研究，并且每个城镇仅能调查一次，试问调查路线怎样设计才能以最高的效率完成此项工作？这是一个图论方面的问题。交通图的存储和管理确实不属于单纯的数值计算问题，而是一种非数值的信息处理问题。

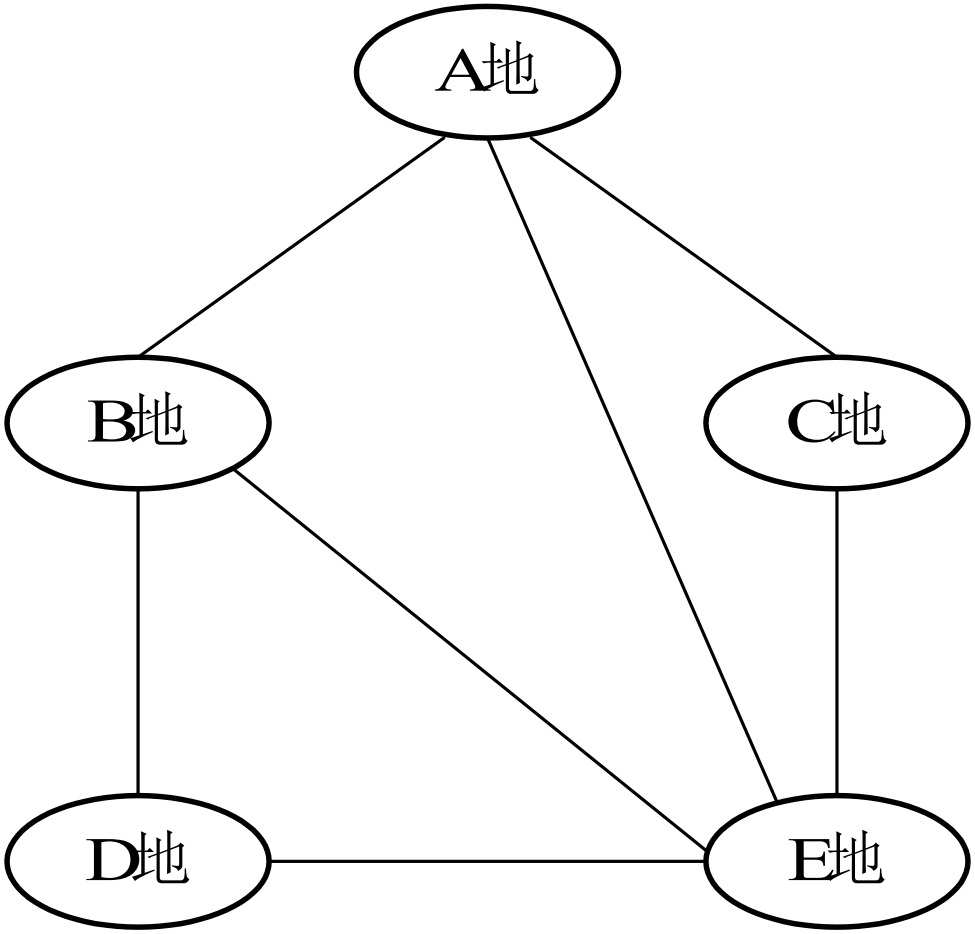


图1.2 交通示意图

1.1.2 数据结构有关概念及术语

一般来说，数据结构研究的是一类普通数据的表示及其相关的运算操作。数据结构是一门主要研究怎样合理地组织数据，建立合适的数据结构，提高计算机执行程序所用的时间效率和空间效率的学科。1968年，美国的D.E.Knuth教授开创了数据结构的最初体系，他的名著《计算机程序设计技巧》较为系统地阐述了数据的逻辑结构和存储结构及其操作。

“数据结构”是计算机专业的一门专业基础课。它为操作系统、数据库原理、编译原理等后继专业课程的学习奠定了基础。数据结构涉及到各方面的知识，如计算机硬件范围中的存储装置和存取方法，计算机软件范围中的文件系统、数据的动态存储与管理、信息检索，数学范围中的许多算法知识。

在计算机科学中，数据(Data)是描述客观事物的数字、字符以及所有能够输入到计算机中并被计算机处理的信息的总称。除了数字、字符之外，还有用英文、汉字或其他语种字母组成的词组、语句，以及表示图形、图像和声音等的信息，这些也可称为数据。

数据元素(Data Element)是数据的基本单位，在计算机中通常作为一个整体进行考虑和处理。例如，图1.1“专业设置树”中的一个专业，图1.2“交通图”中的一个城镇都可称为一个数据元素。数据元素除了可以是一个数字或一个字符串以外，它也可以由一个或多个数据项组成。例如，表1.1中每个学生的学籍信息作为一个数据元素，在表中占一行。每个数据元素由序号、学号、姓名、性别、英语成绩等7个数据项组成。数据项(Data Item)是有独立含义的数据的最小单位，有时也称为字段(Field)。

数据对象(Data Object)是具有相同性质的数据元素的集合，是数据的一个子集。例如，整数数据对象是集合N={0，±1, ±2,…}，字母字符数据对象是集合C={'A', 'B', …, 'Z'}。本节表1.1中的学籍表也可看成一个数据对象。

数据结构(Data Structure)是带有结构的数据元素的集合，它是指数据元素之间的相互关系，即数据的组织形式。我们把数据元素间的逻辑上的联系，称为数据的逻辑结构。常见的数据结构有前文所介绍的线性结构、树型结构、图型结构。数据的逻辑结构体现数据元素间的抽象化相互关系，并不涉及数据元素在计算机中具体的存储方式，是独立于计算机的。

然而，讨论数据结构的目的是为了在计算机中实现对数据的操作，因此还需要研究如何在计算机中表示数据。数据的逻辑结构在计算机存储设备中的映像被称为数据的存储结构，也可以说数据的存储结构是逻辑结构在计算机存储器中的实现，又称物理结构。数据的存储结构是依赖于计算机的。常见的存储结构有顺序存储结构、链式存储结构等。关于它们的详细解释将在以后的章节中逐步给出。

通常所谓的“数据结构”是指数据的逻辑结构、数据的存储结构以及定义在它们之上的一组运算。不论是存储结构的设计，还是运算的算法设计，都必须考虑存储空间的开销和运行时间的效率。因此，“数据结构”课程不仅讲授数据信息在计算机中的组织和表示方法，同时也训练学生高效地解决复杂问题程序设计的能力。

1.2 算法描述与分析

1.2.1 什么是算法

在解决实际问题时，当确定了数据的逻辑结构和存储结构之后，需进一步研究与之相关的一组操作(也称运算)，主要有插入、删除、排序、查找等。为了实现某种操作(如查找)，常常需要设计一种算法。算法(Algorithm)是对特定问题求解步骤的一种描述，是指令的有限序列。描述算法需要一种语言，可以是自然语言、数学语言或者是某种计算机语言。算法一般具有下列5个重要特性：

(1) 输入：一个算法应该有零个、一个或多个输入。

(2) 有穷性：一个算法必须在执行有穷步骤之后正常结束，而不能形成无穷循环。

(3) 确定性：算法中的每一条指令必须有确切的含义，不能产生多义性。

(4) 可行性：算法中的每一个指令必须是切实可执行的，即原则上可以通过已经实现的基本运算执行有限次来实现。

(5) 输出：一个算法应该至少有一个输出，这些输出是同输入有某种特定关系的量。

1.2.2 算法描述工具—— C语言

如何选择描述数据结构和算法的语言是十分重要的问题。传统的描述方法是用PASCAL语言。在Windows环境下涌现出一系列功能强大、面向对象的描述工具，如Visual C++，Borland C++，Visual Basic，Visual FoxPro等。近年来在计算机科学研究、系统开发、教学以及应用开发中，C语言的使用越来越广泛。因此，本教材采用C语言进行算法描述。为了能够简明扼要地描述算法，突出算法的思路，而不拘泥于语言语法的细节，本书有以下约定：

(1) 问题的规模尺寸用MAXSIZE表示，约定在宏定义中已经预先定义过，例如：

```
#define MAXSIZE 100
```

(2) 数据元素的类型一般写成ELEMTP，可以认为在宏定义中预先定义过，例如：

```
#define ELEMTP int
```

在上机实验时根据需要，可临时用其他某个具体的类型标识符来代替。

(3) 一个算法要以函数形式给出：

类型标识符 函数名(带类型说明的形参表)

{语句组}

例如：

```
int add (int a,int b)
  {int c;
   c=a+b;
   return(c);
  }
```

除了形参类型说明放在圆括号中之外，在描述算法的函数中其他变量的类型说明一般省略不写，这样使算法的处理过程更加突出明了。

(4) 关于数据存储结构的类型定义以及全局变量的说明等均应在写算法之前进行说明。

下面的例子给出了书写算法的一般步骤。

例1.1 有n个整数，将它们按由大到小的顺序排序，并且输出。

分析：n个数据的逻辑结构是线性表(a1,a2,a3,…,an)；选用一维数组作存储结构。每个数组元素有两个域：一个是数据的序号域，一个是数据的值域。

```
struct node
{int num;                                   /*序号域*/
  int data;                                 /*值域*/
}
/*算法描述1.1*/
void simsort(struct node a [MAXSIZE], int n)/*数组a的数据由主函数提供*/
{int i,j, m;
for(i=1;i<n;i++)
 for(j=1;j<=n;j++)
  if(a[i].data<a[j].data)
   {m=a[i];a[i]=a[j];a[j]=m;}
  for(i=i;i<=n;i++)
  printf("%8d %8d %10d\n",i,a[i].num,a[j].data);
}
```

1.2.3 算法分析技术初步

著名的计算机科学家N.沃思提出了一个有名的公式：算法+数据结构=程序。由此可见，数据结构和算法是程序的两大要素，二者相辅相成，缺一不可。一种数据结构的优劣是在实现其各种运算的算法中体现的。对数据结构的分析实质上也就是对实现其多种运算的算法的分析。评价一个算法应从四个方面进行：正确性、简单性、运行时间、占用空间。但主要看这个算法所要占用机器资源的多少。而在这些资源中时间和空间是两个最主要的方面，因此算法分析中最关心的也就是算法所需的时间代价和空间代价。

1. 空间

所谓算法的空间代价(或称空间复杂性)，是指当问题的规模以某种单位由1增至n时，解决该问题的算法实现所占用的空间也以某种单位由1增至f(n)，并称该算法的空间代价是f(n)。

2. 时间

(1) 语句频度(Frequency Count)：指的是在一个算法中该语句重复执行的次数。

(2) 算法的渐近时间复杂度(Asymptotic Time Complexity)：算法中基本操作重复执行的次数依据算法中最大语句频度来估算，它是问题规模n的某个函数f(n)，算法的时间量度记作T(n)=O(f(n))，表示随着问题规模n的增大，算法执行时间的增长率和f(n)的增长率相同，称作算法的渐近时间复杂度，简称时间复杂度。时间复杂度往往不是精确的执行次数，而是估算的数量级。它着重体现的是随着问题规模的增大，算法执行时间增长的变化趋势。

1.3 实习：常用算法实现及分析

例如，在下列三个程序段中：

(a) x=x+1;

(b) for(i=1;i<=n;i++) x=x+1;

(c) for(j=1;j<=n;j++)

for(k=1;k<=n;k++) x=x+1;

语句x=x+1的频度分别为1、n和n2，则(a)、(b)、(c)的时间复杂度分别是O(1)、O(n)、O(n2)。由此可见，随着问题规模的增大，其时间消耗也在增大。

下面以(c)程序段为例，进行时间复杂度的分析。

步骤1：先把所有语句改为基本操作。

```
    j=1;                          1
a: if j<=n                        n+1
    {k=1;                         n*1
     b: if k<=n                   n*(n+1)
     {x=x+1;                      n*n
      k++;                        n*n
      goto b;
      }
     j++;                         n*1
    goto a;
}
```

步骤2：分析每一条语句的语句频度，标到每条语句后边，如上。

步骤3：统计总的语句频度：1+n+1+n+n(n+1)+ n^2+ n^2+n=3n^2+4n+2。

步骤4：判断最大语句频度为n^2，所以时间复杂度为O(n^2)。其中O表示等价无穷小。

现在来分析例1.1中算法1.1的时间复杂度。算法中有一个二重循环，if语句的执行频度为

$$n+(n-1)+(n-2)+\ldots+3+2+1=\frac{n(n+1)}{2}$$

数量级为$O(n^2)$。算法中输出语句的频度为n，数量级为O(n)。该算法的时间复杂度以if语句的执行频度来估算(忽略输出部分)，则记为$O(n^2)$。算法还可能呈现的时间复杂度有指数阶O(lbn)等。

习 题 1

1. 简述下列术语：数据元素，数据，数据对象，数据结构，存储结构。

2. 试写一算法，自大至小依次输出顺序读入的3个整数x、y和z的值。分析算法的元素比较次数和元素移动次数。

3. 举出一个数据结构的例子，叙述其逻辑结构、存储结构、运算等三方面的内容。

4. 叙述算法的定义及其重要特性。

5. 分析并写出下面的各语句组所代表的算法的时间复杂度。

(1) {for(i=1;i<=n;i++)

for(j=1;j<=i;j++)

for(k=1;k<=j;k++)

{s=i+k;printf("%d",s);}

}

```
(2)  {i=1;k=0;
while(i<=n-1)
{k=k+10*i;
   i++;
}
}

(3)  {i=1;k=0;n=100;
do{k=k+10*i;
   i++;
}while(i= =n);
}
```

```
(4)  {i=1;j=0;
      while(i+j<=n)
      {if(i>j) j++;
       else i++;
      }
     }

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

(6) {m=91;n=100;

while(n>0)

{if(m>0){m=m-1;n--;}

else m++;

}

}

第2章 线性表

2.1 线性表引例

例2.1 某大学欲进行一次数学竞赛，约有200名学生报名参赛。现将报名登记表(如表2.1所示)存入计算机以便完成如下工作：

(1) 能正确录入学生记录；

(2) 按成绩对该表进行重新排序；

(3) 按学号或姓名查询学生成绩。

表2.1 报名登记表

学 号	姓 名	性 别	成 绩
2003	张三	男	84
2024	李四	男	79
2035	王五	女	75
⋮	⋮	⋮	⋮

2.2 线性表的定义和基本运算

2.2.1 线性表的概念

线性表是指n(n≥0)个具有相同类型数据元素(或称结点)的有限序列，可表示为(a1,a2,...,ai,...,an)。其中，ai代表一个数据元素，a1称为表头(或头结点)，an称为表尾(或尾结点)，ai(0≤i＜n)称为ai+1的直接前驱，ai+1称为ai的直接后继。线性表中数据元素的个数称为线性表的长度，长度为0的线性表称为空表，记为()。

在不同的问题中，数据元素代表的具体含义不同，它可以是一个数字、一个字符，也可以是一句话，甚至其他更复杂的信息。例如:

线性表L1: (12, 58, 45, 2, 45, 46), 其元素为数字；

线性表L2: (a, g, r, d, s, t), 其元素为字母。

表2.1也是一个线性表，其数据元素较为复杂，每个学生的学号、姓名、性别、成绩构成一个数据元素。这种由若干数据项构成的数据元素常称为记录，含有大量记录的线性表称为文件。

2.2.2 表的基本运算

线性表是一种相当灵活的数据结构，对其数据元素可以进行各种运算(操作)。如对表2.1，应不仅能查询成绩，还能根据需要增加或删除学生记录。下面给出线性表一些基本运算的含义，这些运算的实现算法后面将具体讨论。

(1) Initiate (L)：初始化运算。该函数用于设定一个空的线性表L。

(2) Length (L)：求长度函数。该函数返回给定线性表L中数据元素的个数。

(3) Get (L, i)：取元素操作。若1≤i≤Length(L)，则函数值为给定线性表中第i个数据元素，否则为空元素NULL。

(4) Prior (L, x)：求前驱函数。当x在线性表L中，且其位序大于1，则函数值为x的直接前驱，否则为空元素。

(5) Next (L, x)：求后继函数。当x在线性表L中，且其位序小于Length(L), 则函数值为x的直接后继，否则为空元素。

(6) Locate (L,x)：定位操作。如线性表中存在和x 相等的数据元素，则返回该数据元素的位序。若满足条件的元素不惟一，则返回最小的位序。

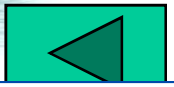
(7) Insert (L, i, x): 前插操作。若1≤i≤Length(L)+1，则在线性表L中第i个元素之前插入新结点x。

(8) Delete (L, i): 删除操作。若1≤i≤Length(L)，则删除线性表L中第 i个元素。

(9) Empty (L)：判空函数。若L为空表，则返回值为1(表示“真”)，否则返回值为0(表示“假”)。

(10) Clear (L)：置空操作。将线性表L 值为空表。

利用这些基本运算还可实现对线性表的各种复杂操作。如将两个线性表进行合并，重新复制一个线性表，对线性表中的元素按某个数据项递增(或递减)的顺序进行重新排序等。读者可将以上基本运算应用于表2.1，理解在具体问题中各种运算的具体含义。



2.3 线性表的顺序存储结构

2.3.1 向量的存储特点

在计算机内，线性表可以用不同的方式来存储。其中最简单、最常用的方式就是顺序存储，即用一组连续的存储单元依次存放线性表中的元素。这种顺序存储的线性表称为顺序表，又叫向量。

假设线性表每个元素占s个存储单元，并以其所占的第一个单元的存储地址作为数据元素的存储位置, 则线性表中第i+1个元素的存储位置$Loc(a_{i+1})$和第i个数据元素的存储位置$Loc(a_i)$之间满足下列关系：$Loc(a_{i+1}) = Loc(a_i) + s$

设线性表的起始位置(或称基址)是$Loc(a_1)$，因每个元素所占用的空间大小相同，则元素a_i的存放位置为：

$$Loc(a_i)= Loc(a_1)+s*(i-1)$$

由此可见，线性表的顺序存储结构是用数据元素在计算机内“物理位置相邻”来表示数据元素之间的逻辑相邻关系，其特点是向量中逻辑上相邻的结点在计算机的存储结构中也相邻，如图2.1所示。而且，只要知道了向量的基地址，由上式即可确定向量中任一数据元素的地址，从而对其可随机存取。

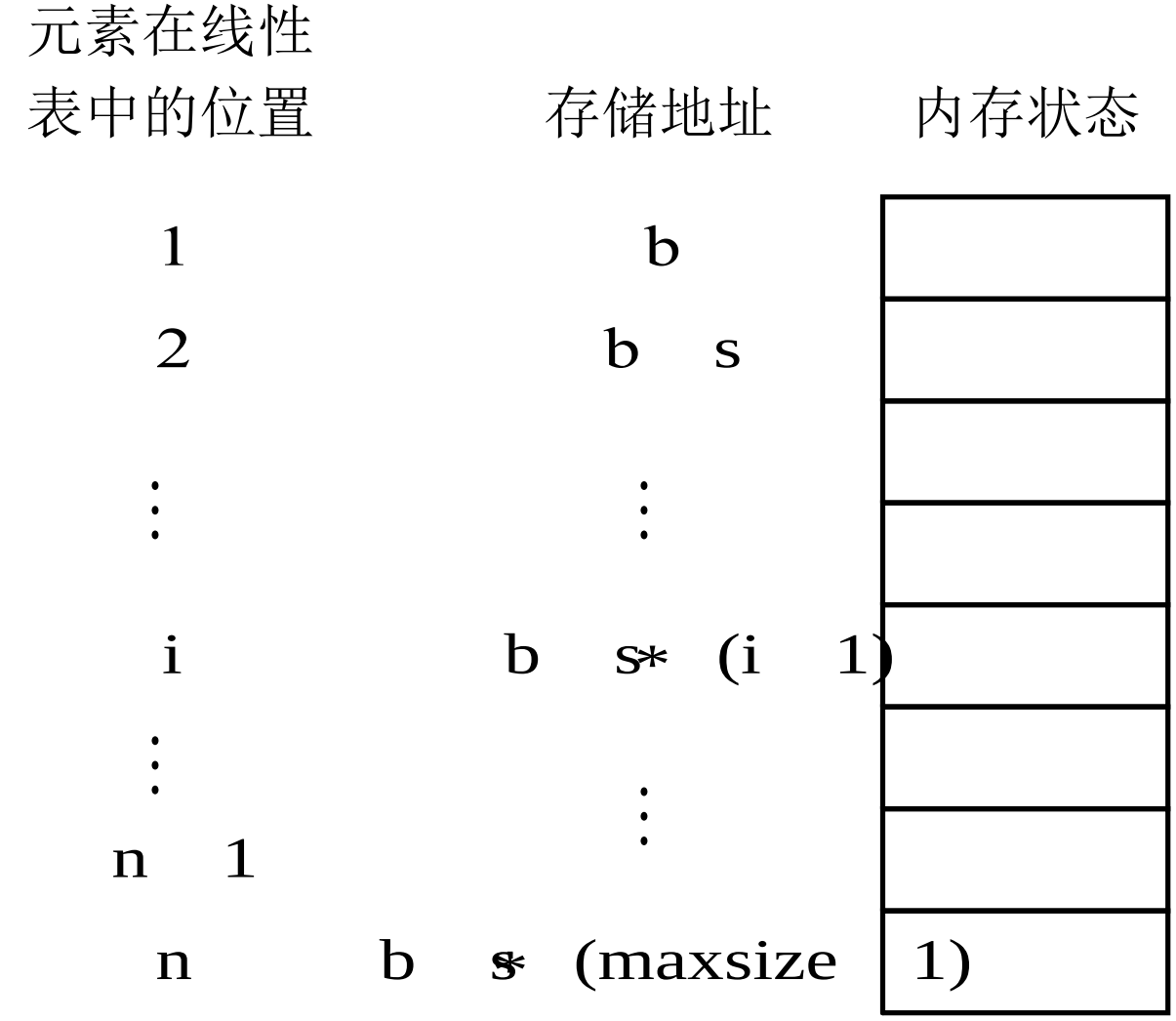


图2.1 线性表的顺序存储结构示意图

在C语言中，可以用一维数组来描述向量。

```
# define  maxsize  N;        /*设置线性表的最大长度为N, N为整数*/

typedef  struct

        {datatype  data[maxsize+1];    /*datatype为元素的数据类
型，它可是Turbo C中*/

                                        /*允许的任何数据类型*/

        int  last;                      /*记录当前表中元素的个数*/

        } Sqlist;
```

上述描述方法，将线性表顺序存储结构中的信息封装隐藏在类型Sqlist结构中。data数组描述了线性表中数据元素占用的空间，数组中第i个分量就是线性表中第i个数据元素。last描述了当前表中数据元素的个数即表长。

说明：在C语言中，数组的下标是从0开始的，但为了算法描述方便，本书中凡涉及数组的算法，规定下标从1开始，这样，读者可不必考虑下标为0的数组元素。

2.3.2 向量中基本运算的实现

1. 定位操作Locate (L, x)

定位操作返回线性表L中值和x相同的第一个元素的位置。算法如下：

```
/*算法描述2.1*/
Int Locate (Sqlist L, Datatype x)
{int  i=1;
    while ((i<L.last) && (L.data[i]!=x))  i++;
    if (i<=L.last)  return (i);
    else  return (0 );
}
```

也可按数据元素的某个关键数据项进行数据元素的定位。例如，表2.1所示的学生成绩表可按学号或姓名进行定位，只需将上面算法中data[i]和x换成相应数据项即可，请读者参阅后面实例。

算法描述2.1的基本操作是“进行两个元素之间的比较”。若线性表L中存在值和x相等的数据元素a_i，则需进行i (1≤i≤L.last) 次比较，否则，进行L.last次比较，直至i超出表长。所以该算法的时间复杂度为O(n)。

2. 向量的插入运算Insert (L, i, x)

线性表的插入操作是指在线性表的第i－1个元素和第i 元素之间插入一个新的数据元素，使长度为n的线性表(a_1, a_2, …, a_i, a_{i+1}, …, a_n)，变成长度为n+1的线性表(a_1, a_2, …, a_i, x, a_{i+1}, …, a_{n+1})。显然数据元素a_i和a_{i+1}的逻辑关系发生了变化，对向量而言，逻辑上相邻的数据元素在物理位置上也相邻，因此，必须将第i (i<n+1)至第n个元素依次向后移一个位置，空出位置放入x，才能反映这个逻辑关系上的变化。其具体算法如下：

```
/*算法描述2.2*/
void  Insert (Sqlist L, int i, Datatype  x )
{int  j;
 if (i<1 || i >L.last ) printf ("infeasible position! \n");
 else {  if ( L.last+1>maxsize ) printf("overflow!\n");
       else {for (j=L.last; j>=i; j--) L.data[j+1]=L.data[j];
            L.data[i]=x;  L.last++;
           }
     }
}
```

3. 向量的删除运算 Delete (L, x)

与向量的插入运算道理相同，当删除线性表中第i 个元素时，也改变了原数据间的逻辑关系，故需将第i+1(i<n+1)至第n个元素依次向前移一个位置来反映这个变化。算法如下：

```
/*算法描述2.3*/
void  Delete (Sqlist L, int i)
{int  j;
 if (i<1 || i>L.last )  printf ("infeasible \n");
 else {for (j=i; j<L.last; j++)
     L.data[j]=L.data[j+1];
     L.last--; }
 }
```

从算法2.2和2.3可以看出，在向量中某个位置插入或删除一个数据元素时，其时间主要耗费在移动元素上，故应将移动元素的操作作为预估算法时间复杂度的基本操作。假定在线性表中任意位置插入元素的概率相等，即p=1/(n+1), 那么在长度为n 的线性表中插入一个元素时所需移动元素的平均次数为

$$E_{insert}=\sum p_i*(n-i+1)$$

$$=\frac{1}{n+1}\sum(n-i+1)$$

$$=\frac{n}{2}$$

同理, 在线性表中删除任意一个元素时所需移动元素的平均次数为

$$Edelete=\sum p_d*(n-i+1)$$

$$=\frac{1}{n}\sum(n-i+1)$$

$$=\frac{n-1}{2}$$

所以, 对于长度为n的线性表, 算法Insert和算法Delete的时间复杂度均为O(n)。

2.4 线性表的链式存储结构

2.4.1 线性链表

与线性表的顺序存储结构不同，链式存储结构用一组任意的存储单元(可以是连续的，也可以是不连续的)来存储线性表的数据元素。为表示相邻数据元素之间的逻辑关系，将每个存储结点分为两个域：数据域用来存放一个数据元素的自身信息；指针域用来存放该数据元素直接后继的存储位置。这样，可以通过指针域中存放的信息(称为指针或链)将n个结点连接成一个链表，即成为线性表的链式存储结构。由于这种存储结构中每个结点只有一个指针域，故又将其称为线性单链表或单向链表。

图2.2给出了线性表(A,B,C,D)的链式存储结构。由于最后一个元素没有直接后继，则其结点的指针域应为“空”(NULL)。另外，由图2.2可以看出，头指针指向链表中第一个结点的存储位置，每个元素的存储位置都包含在其直接前驱结点的指针域中，因此，单向链表的存取必须从头指针开始，它是一种非随机存取的存储结构。

用线性链表表示线性表时，数据元素之间的逻辑关系由结点中的指针指示，故逻辑上相邻的数据元素其物理位置不要求紧邻，这与线性表的顺序存储结构完全不同。

头指针head　　存储地址　　内存状态

2000H

存储地址	数据域	指针域
2000H	A	2002H
2002H	B	2006H
⋮	⋮	⋮
2006H	C	3205H
⋮	⋮	⋮
3205H	D	NULL

图2.2 单向链表的存储结构示意图

我们在使用链表时往往只关心它所表示的数据元素之间的逻辑顺序，而不是每个元素在存储器中的实际位置。因此，为了分析方便，把链表画成用箭头相连接的结点的序列，结点之间的箭头表示链域中的指针，如图2.2可画成图2.3所示的形式。

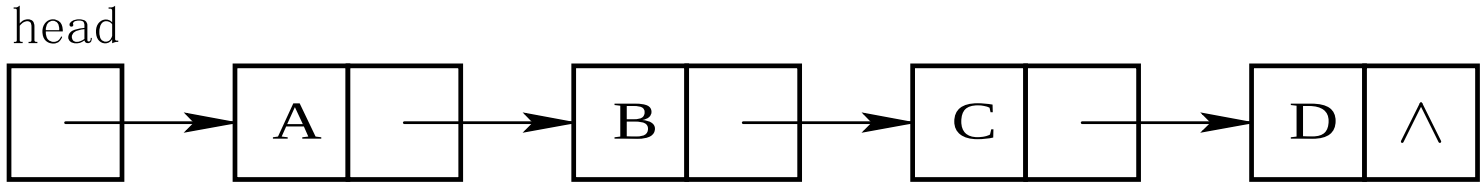


图2.3 单向链表的逻辑状态图

根据上述分析，单向链表可由头指针惟一确定，故在C语言中可用指针数据类型来描述。

```
Typedef struct Node
{Datatype  data;
 struct Node  *next;
}Node,  *LList;
```

一个单向链表对应一个头指针head，head是一个LList类型的变量，即它是一个指向Node类型结点的指针变量，并指向单向链表的第1个结点，通过它可以访问该单向链表。若头指针为“空”(即head=NULL)，则表示一个空表。

一般在单向链表中附加一个头结点，其指针域指向链表的第一个结点，而其数据域可以存储一些如链表长度之类的附加信息，也可以什么都不存储。这样，链表的头指针将指向头结点。如图2.4所示，表空的条件是头结点的指针域为“空”，即head->next=NULL。

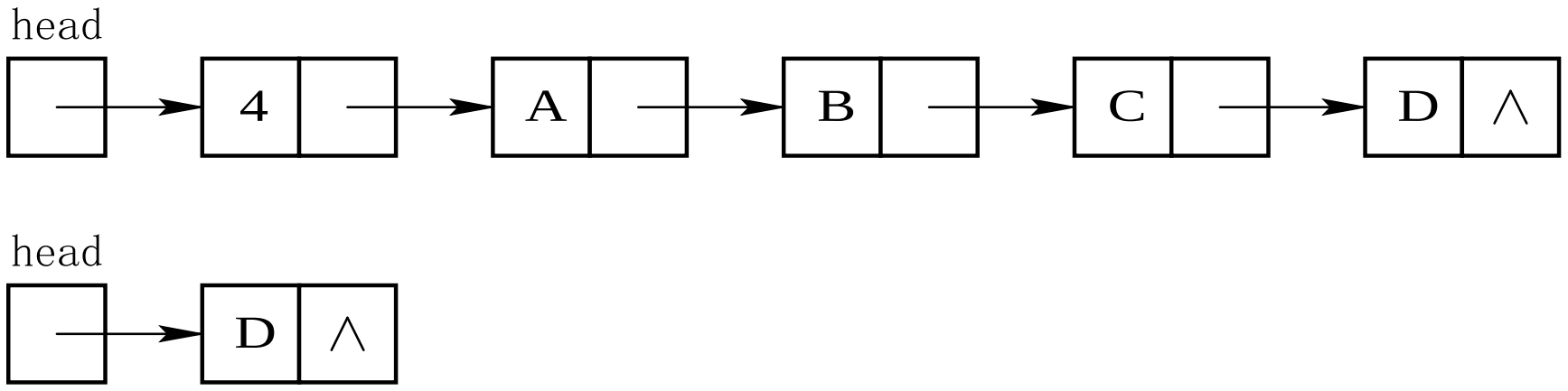


图2.4 带表头的单链表

2.4.2 单向链表基本运算的实现

1. 取单链表中元素Get (L , i)

该函数返回线性表L中第 i个数据元素的值。算法思路：从头指针出发，借用指针p，从第1个结点开始，顺着后继指针向后寻找第i个元素。若存在第i 个元素，即1≤i＜Length(L)，则通过p返回该元素的值。

```
/*算法描述2.4*/
Datatype  GetLList(LList L, int  i)
{ LList  p;
int  j =1;
p=L->next;
while( p!=NULL && j<i ) {p=p->next; j++; }
if (p= = NULL || j>i ) printf("no this data!\n");
else  return(p->data);
}
```

该算法的基本操作是比较j和i 并后移指针，若第i 个元素存在，则需执行基本操作i－1次，否则执行n次，故算法2.4的时间复杂度均为O(n)。

2. 定位函数Locate(L, x)

该函数在线性链表中寻找值与x相等的数据元素，若有，则返回其存储位置，否则返回NULL。其算法2.5思路与算法2.4相似, 其时间复杂度均也为O(n)。

```
/*算法描述2.5*/

LList  Locate (LList L, Datatype x)

{LList  p;

p=L->next;

while (p!=NULL && p->data!=x ) p=p->next;

return (p);

}
```

3. 单链表的插入Insert (L, i,x)

该函数在线性链表第i个元素之前插入一个数据元素x。算法思路：先生成一个包含数据元素x的新结点(用s指向它)，再找到链表中第i－1个结点(用p指向它)，修改这两个结点的指针即可。指针修改如图2.5所示，用语句描述为：

```
s->next=p->next;

p->next=s;
```

注意：修改指针的顺序，若先修改第i－1个结点的指针，使其指向待插结点，那么，第i个结点的地址将丢失，链表“断开”，待插结点将无法与第i个结点链接。

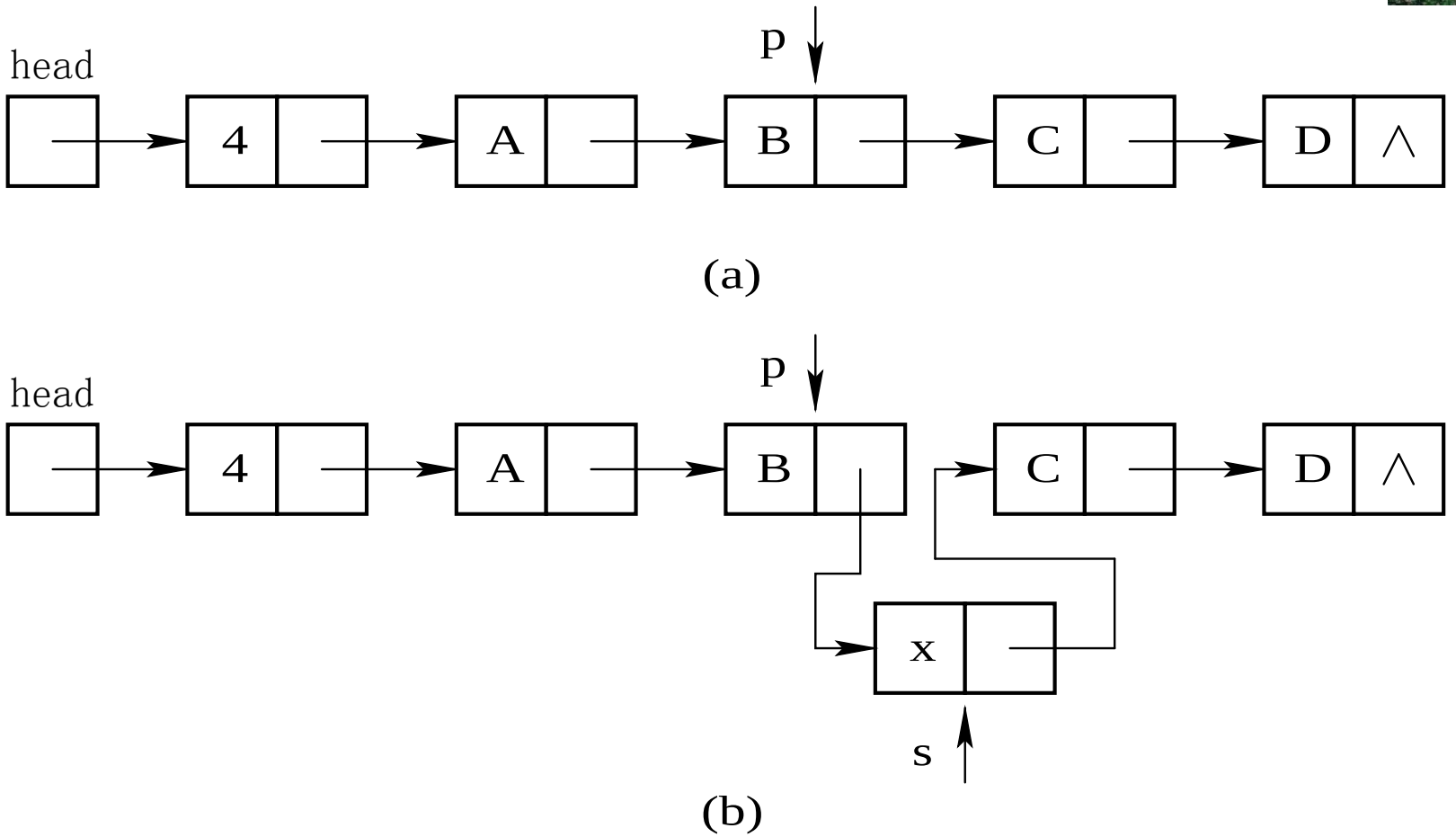


图2.5 单向链表中插入结点时指针的变化情况
(a) 插入前；(b) 插入后

```
/*算法描述2.6*/
void InsetLList(LList L, int i, Datatype x)
{LList  p, s;
 int  j=0;
 p= L;
 while (p!=NULL && j<i-1 ) {p=p->next;  j++; }
 if (p= = NULL || j>i-1) printf("No this position!\n");
 else  {s=(LList) malloc (sizeof (Node) );
      s->data=x;
      s->next=p->next;
      p->next=s;}
}
```

4. 单链表的删除Delete(L, i)

该函数删除线性链表中第i个数据结点。显然，只要找到第i－1个结点修改其指针使它跳过第i个结点，而直接指向第i+1个结点即可。但要注意，删除的结点应及时向系统释放，以便系统再次利用。指针变化如图2.6所示，语句描述为

```
p->next=p->next->next ;
```

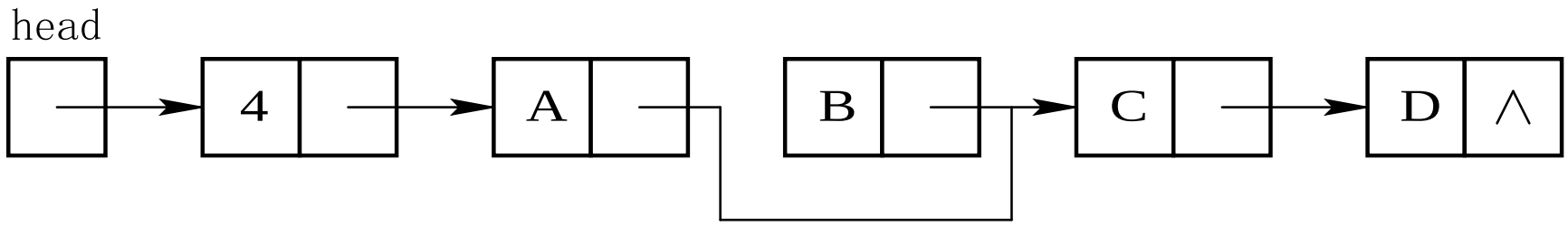


图2.6 单向链表中删除结点时指针的变化情况

其具体算法描述如下：

```
/*算法描述2.7*/
void Delete (LList L, int i)
{LList  p, q;
 int  j=0;
 p=L;
 while ( p!=NULL &&  j<i-1 ) {p=p->next; j++; }
 if ( p= = NULL || j>i-1) printf(" No this data!\n");
 else { q=p->next;
      p->next =p->next->next;
      free (q);
    }
}
```

由于在单向链表中插入和删除结点时，仅需修改相应结点的指针，而不需移动元素，该程序的执行时间主要耗费在查找结点上，由算法2.4知访问结点的时间复杂度为O(n), 所以算法2.6和算法2.7的时间复杂度均为O(n)。

5. 单链表的建立Crt-LList(L,n)

建立线性表的链式存储结构的过程就是一个动态生成链表的过程，即从“空表”的初始状态起，依次建立各元素结点，并逐个插入链表。下面是一个从表尾到表头建立单链表的算法，其时间复杂度是O(n)。

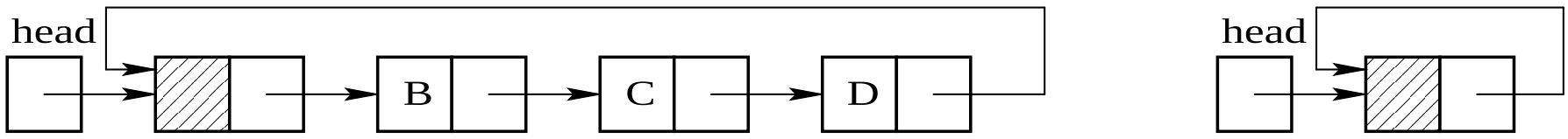
```
/*算法描述2.8*/
void Crt-LList(LList h, int n)
{LList p，  q;
 int i;
 h=(LList)malloc(sizeof (Node) );
 h->next=NULL;  p=h;
 for(i=1;i<=n;i++)
     {q=(LList)malloc(sizeof(Node));
      scanf("%&",&q->data);  q->next=NULL;
      p->next=q;   p=q;}
}
```

说明：上面算法中分别引用了Turbo C 语言的两个标准函数malloc()和free()。设p为LList 型变量，则执行p=(LList)malloc(sizeof (Node)) 的作用是向系统申请一个Node型的结点，同时让p指向该结点；执行free(p)的作用是向系统释放一个由p所指的Node型的结点，已释放的空间可供系统再次使用。

2.5 循环链表和双向链表

2.5.1 循环链表

循环链表是另一种形式的链式存储结构。其特点是表中最后一个结点的指针域指向头结点，整个链表呈环状。从表中任意结点出发都可到达其他结点，如图2.7所示为单循环链表。



(a) (b)

图2.7 单循环链表

(a) 非空表；(b) 空表

循环链表和单链表算法实现基本相同，差别仅在于前者算法中的循环条件是判p或p->next是否为空，而后者是判它们是否等于头指针。有时为了简化某些操作在链表中设立尾指针，而不是头指针。例如，将两个用循环链表存储的线性表合并成一个线性表，此时仅需将一个表的表尾和另一个表的表头相连即可。指针变化如图2.8所示，用语句描述为：

```
p=A->next;

A->next =B->next->next ;

B->next=p;
```

操作只改变了两个指针值，其算法的时间复杂度均为O(1)。

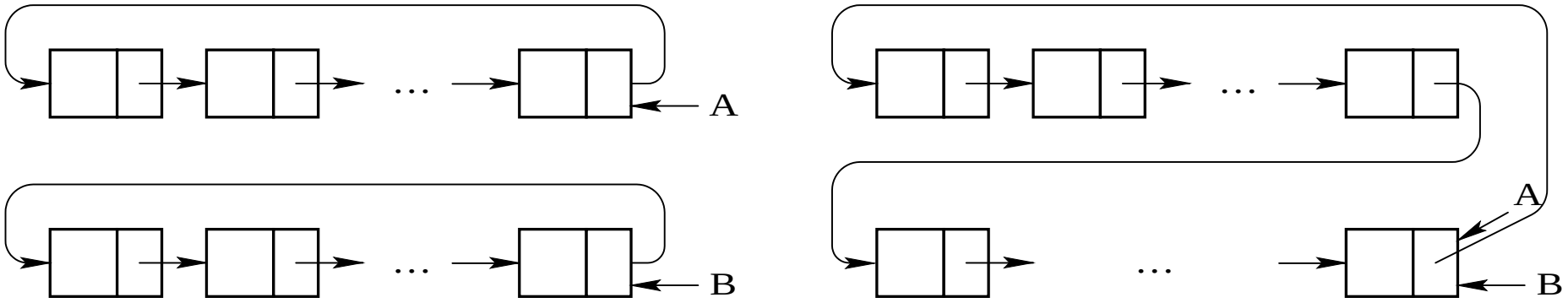


图2.8 循环链表合并示意图

(a) 合并前；(b) 合并后

2.5.2 双向链表

单向链表的结点只有一个指示其直接后继的指针域, 顺着某结点的指针可很容易地访问其后诸结点。但若要访问某结点的直接前驱，前驱虽与该结点相邻却无法直达，此时需从表头出发，且寻访时要记录相关信息。为克服单向链表这种访问方式的单向性，特设计了双向链表, 如图2.9(b)所示。

显然，在双向链表的结点中应有两个指针域，一个指向直接后继，一个指向直接前驱，如图2.9(a)所示。双向链表在Turbo C语言中可描述如下：

```
typedef  struct dnode
    {datatype  data;
struct  dnode  *prior;
struct  dnode  *next;
}DNode, *DList;
```

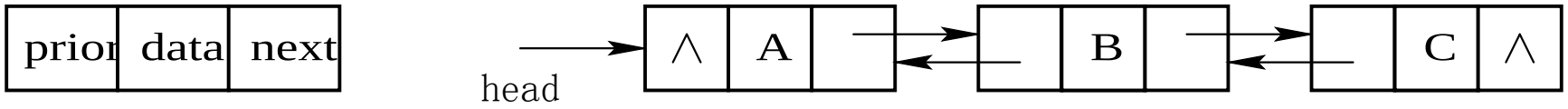


图2.9 双向链表示例

(a) 结点结构；(b) 双向链表

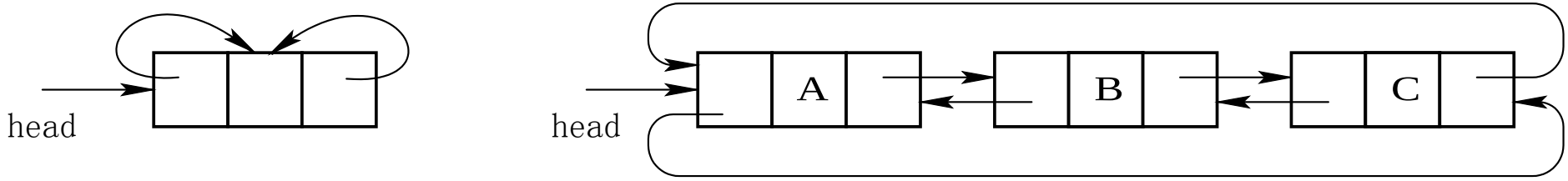


图2.10 双向循环链表示例

(a) 空表；(b) 非空表

在双向链表中，Length(L)，Get(L,i)，Locate(L,x)等操作仅涉及一个方向的指针，其算法描述与单链表相同。但插入和删除操作有所不同，在双向链表中需同时修改两个方向的指针，图2.11和图2.12分别显示了删除和插入结点时指针的修改情况，其具体算法读者可自己完成。

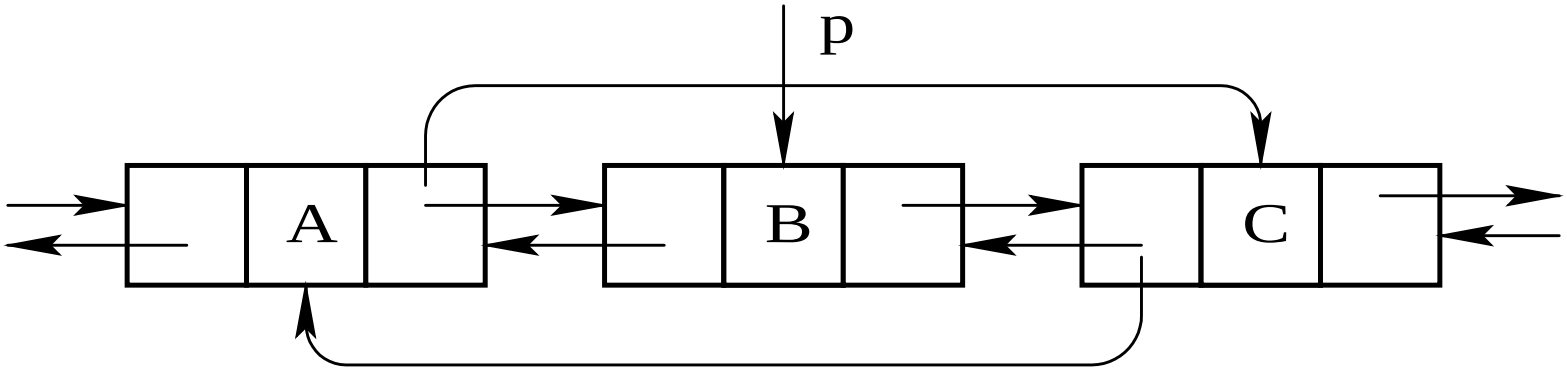


图2.11 双向链表中删除结点时指针的修改情况

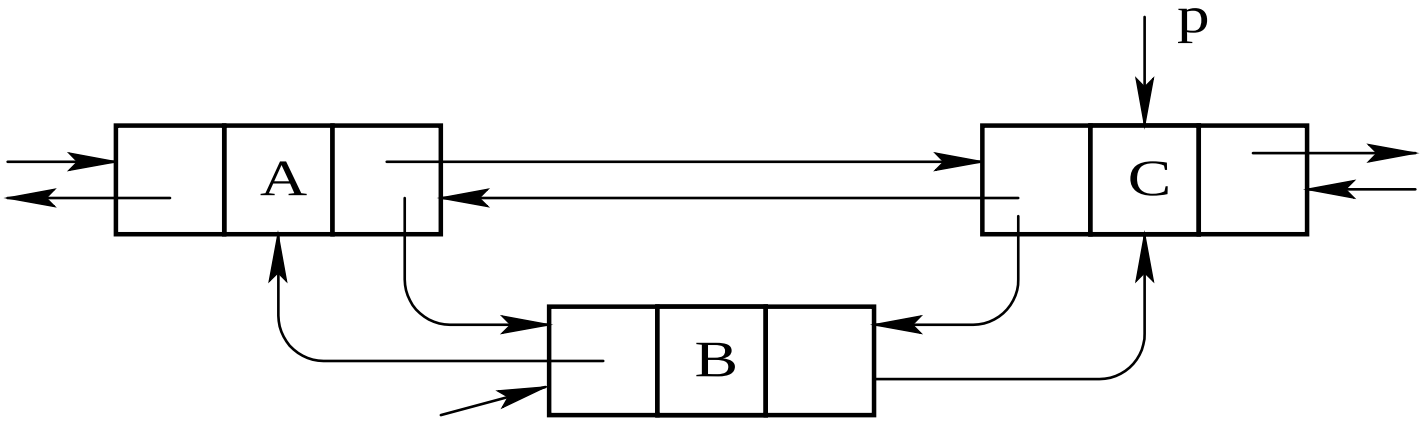


图2.12 双向链表中插入结点时指针的修改情况

在双向链表中删除结点时指针的变化用语句描述为:

```
p->prior->next=p->next;
p->next->prior=p->prior;
free(p);
```

在双向链表中插入结点时指针的变化用语句描述为:

```
s->prior=p->prior;
p->prior->next=s;
s->next=p;
p->prior=s;
```

2.5.3 线性表的顺序存储结构和链式存储结构的比较

在计算机中，线性表有两类不同的存储结构：顺序存储结构和链式存储结构，它们各有特点。顺序表的特点是逻辑上相邻的结点在存储结构中也相邻，它是一种可随机存取存储结构，在C语言中，用一维数组来描述，有以下三方面的缺点：

(1) 在插入和删除结点时，需移动大量元素；

(2) 在对长度较大的线性表预先分配空间时，必须按最大空间分配，从而使存储空间得不到充分利用；

(3) 表的容量难以扩充。

链式存储结构的特点是逻辑上相邻的数据元素其物理位置不要求紧邻，它是一种非随机存取存储结构，在C语言中用“结点指针”来描述。它克服了顺序表上述的三个缺点，但却不具备像顺序表那样随机存取的优点。

在实践中应仔细分析，根据不同研究对象的特点和经常进行的操作选择合适的存储结构。

2.6 实习：线性表的应用实例

实例1：利用顺序表实现例2.1的完整C语言程序。

```
# def   maxsize   250
typedef struct
        { struct   student
            {int  num;
             char *name;
             char  gender;
             float  score;
            }  data[maxsize+1];
          int  last;
        } Sqlist;
```

```
# include  "stdio.h"
void Initiate(L)
Sqlist  L;
{L.last=0;}
int Locate ( L, x)
Sqlist  L;
int  x ;
{int  i=1;
while (i<L.last && L.data[i].num<>x)  i++;
if (i<=L.last)  return (i);
else  return (0 );
}
```

```
void sort(L)
Sqlist  L;
{int i, j;
student x;
for(i=2; i<L.last;i++)
   {L.data[0]=L.data[1];  j=i-1;  x=L.data[i].num;
    while (x<L.data[j].num) {L.data[j+1]=L.data[j];j=j-1;}
    L.data[j+1]=L.data[0];}
}
main()
{ Sqlist L;
int i=1, num;
Initiate(L);
```

```
printf("input data,  please!( "-1"-- ----------end)\n");
scanf("num=%d", &L.data[i].num);
While(L.data[i].num<>-1&&i<=maxsize)
{scanf("name=%s,sex=%c,score=%f\n",&L.data[i].name,&L.data[i].
sex,&L.data[i].score);
   i++;
   scanf("num=%d", &Ldata[i].num);}
Sort(L);
for(i=1; i<=L.last;i++)
printf("%d %s %c %f\n", L.data[i].num, L.data[i].name, L.data[i].sex,
L.data[i].score);
```

```
printf("do you want to find a student?(y\n) ");
while(getchar()=='y')
{printf(" input the number of the student ! ");
    scanf("%d", &num);
    i=Locate(L, num);
    printf("%d, %s %c %f\n", L.data[i].num, L.data[i].name,
L.data[i].sex, L.data[i].score);}
}
```

实例2：多项式相加问题。

1) 存储结构的选取

任一一元多项式可表示为$P_n(x)=P_0+P_1x+P_2x^2+...+P_nx^n$，显然，由其n+1个系数可惟一确定该多项式。故一元多项式可用一个仅存储其系数的线性表来表示，多项式指数i隐含于P_i的序号中。

$$P=(P_0, P_1, P_2, ..., P_n)$$

若采用顺序存储结构来存储这个线性表，那么多项式相加的算法实现十分容易，同位序元素相加即可。

但当多项式的次数很高而且变化很大时，采用这种顺序存储结构极不合理。例如，多项式$S(x)= 1+ 3x+12x^{999}$需用一长度为1000的线性表来表示，而表中仅有三个非零元素，这样将大量浪费内存空间。此时可考虑另一种表示方法，如线性表S(x)可表示成S=((1,0)，(3,1), (12,999))，其元素包含两个数据项：系数项和指数项。

这种表示方法在计算机内对应两种存储方式：当只对多项式进行访问、求值等不改变多项式指数(即表的长度不变化)的操作时，宜采用顺序存储结构；当要对多项式进行加法、减法、乘法等改变多项式指数的操作时，宜采用链式存储结构。

2) 一元多项加法运算的实现

采用单链表结构来实现多项加法运算，无非是前述单向链表基本运算的综合应用。其数据结构描述如下，

```
typedef stuct Pnode
 {float   coef;
 int  exp;
 struct pnode  *next;
 }Pnode,  *Ploytp;
```

图2.13给出了多项式A(x) = 15+ 6x+ $9x^7$+$3x^{18}$ 和B(x)= 4x+$5x^6$+ $16x^7$的链式存储结构(设一元多项式均按升幂形式存储，首指针为－1)。

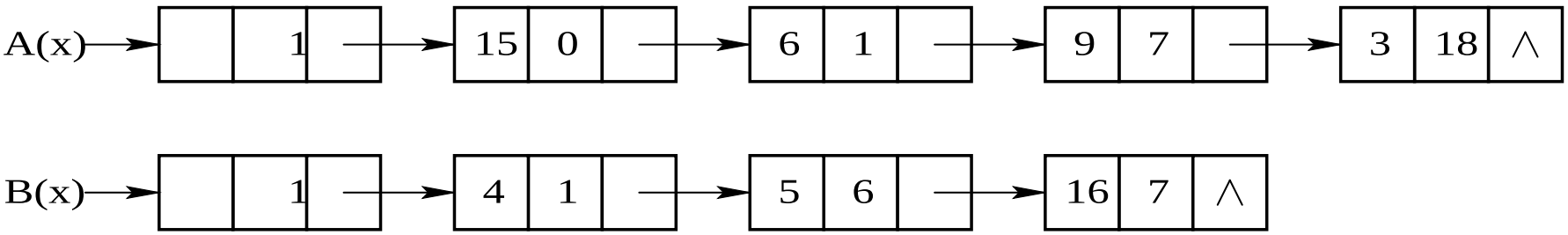


图2.13 一元多项式的存储

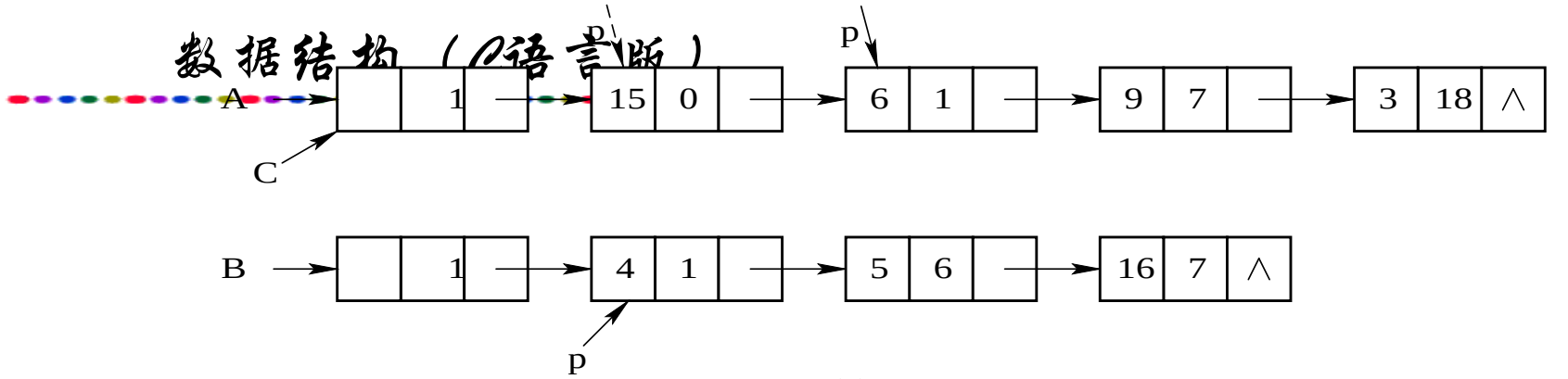
若上例A+B结果仍存于A中，根据一元多项式相加的运算规则，其实质是将B逐项按指数分情况合并于“和多项式”A中。设p, q分别指向A, B的第一个结点，如图2.14所示，其算法思路如下：

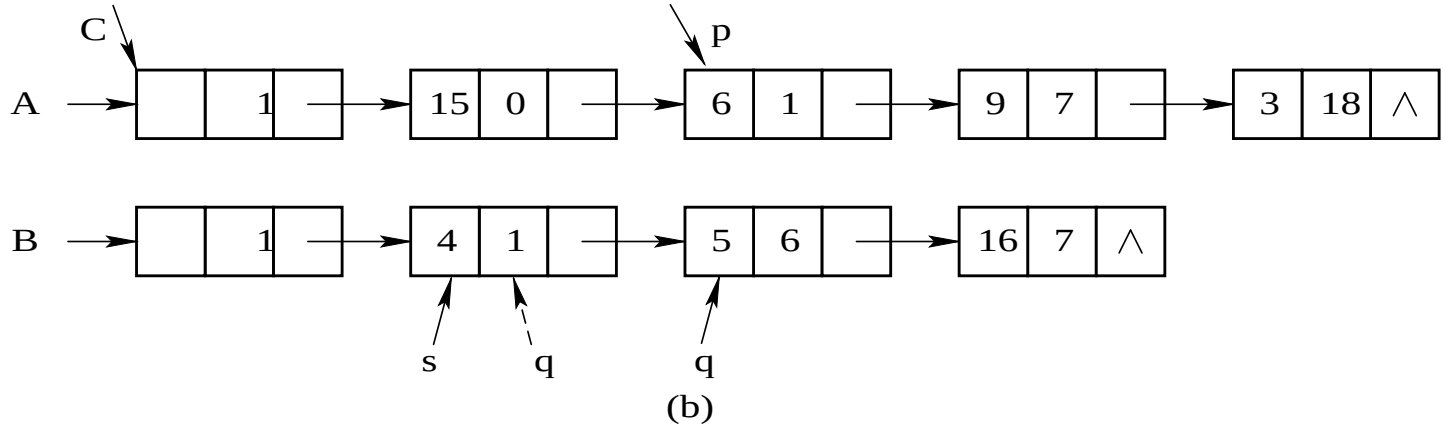
(1) p->exp<q->exp, 应使指针后移p=p->next，如图2.14(a)所示。

(2) p->exp=q->exp, 将两个结点系数相加，若系数和不为零，则修改p->ceof，并借助s释放当前q结点, 而使q指向多项式B的下一个结点，如图2.14(b)所示；若系数和为零，则应借助s释放p, q结点, 而使p, q分别指向多项式A,B的下一个结点。

(3) p->exp > q->exp，将q结点在p结点之前插入A中, 并使q指向多项式B的下一个结点，如图2.14(c)所示。

直到q=NULL为止或p=NULL，将B的剩余项链到A尾为止。最后释放B的头结点。





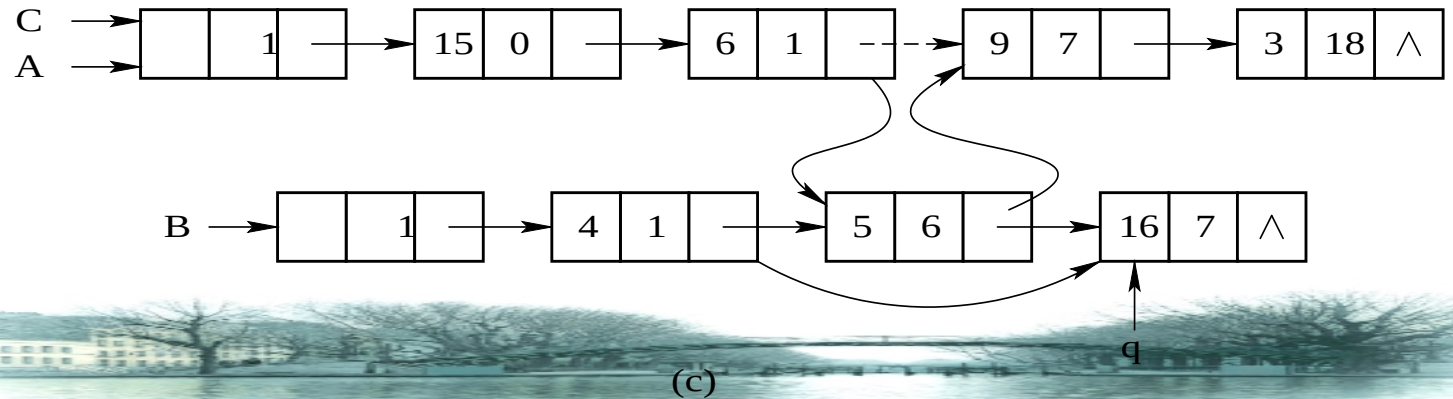


图 2.14 多项式相加运算示例

下面给出从接收多项式到完成多项式相加运算的完整C语言程序。

```
# include  "stdio.h"
void Crt-Polytp (h, n)
Polytp h;
int n;
{ Polytp，  q;
   int i;
   h=( Polytp)malloc(sizeof (Pnode) );
   h->next=NULL;  p=h;
      for(i=1;i<=n;i++)
         {q=( Polytp)malloc(sizeof(Pnode));
         scanf("%f,%d",&q->ceof,&q->exp);  q->next=NULL;
         p->next=q;   p=q;}
}
```

```
int Cmp(a,b)
float a,b;
{ if(a<b)return(-1);
  if(a=b)return(0);
  if(a>b)return(1);
}
void  Add Poly(pa, pb，  pc)
Polytp pa,pb,pc;
{ Polytp p,q,pre,s;
p=pa->next; q=pb->next;
pre=pa; pc=pa;
while(p!=NULL & q<> NULL)
```

```
{w=cmp(p->exp, q->exp);
 switch(w)
      {case -1:  pre=p; p=p->next; break;
       case 0:   sum=p->coef+q->coef;
                 if (sum<>0) {p->coef=sum;  pre=p; }
                  else {pre ->next=p->next; free(p); }
                 p=pre->next;
                 s=q; q=q->next; free(s); break;
       case 1:   s=q->next; q->next=p; pre->next=q;
                  pre=q; q=s; break;
    }
}
```

```
if (pb<>NULL) pre->next=q;
free(pb);
}
main()
{Ploytp pa,pb.pc,q;
 int  n1 , n2;
 printf(" input the length of pa and pb");
 scanf("n1= %d, n2=%d", &n1, &n2);
```

```
Crt-Polytp (pa,n1);
 Crt-Polytp (pb,n2);
 AddPolytp(pa,pb,pc);
 p=pc->next;
 printf("pc=pa+pb=");
 while(p<>NULL)
    {printf(" %f,%d", p->ceof, p->exp); p=p->next;}
 }
```

习 题 2

1．判断下列概念的正确性：

(1) 线性表在物理存储空间中也一定是连续的。

(2) 链表的物理存储结构具有与链表一样的顺序。

(3) 链表的删除算法很简单，因为当删去链表中某个结点后，计算机会自动地将后继的各个单元向前移动。

2．试比较顺序存储结构和链式存储结构的优缺点。

3．试写出一个计算链表中数据元素结点个数的算法，其中指针p指向该链表的第一个结点。

4．试设计实现在单链表中删去值相同的多余结点的算法。

5．有一个性表(a_1，a_2，…，a_n)，它存储在有附加表头结点的单链表中，写一个算法，求出该线性表中值为x的元素的序号。如果x不存在，则输出序号为0。

6. 写一个算法将一单链表逆置。要求操作在原链表上进行。

7．设有两个链表A、B，它们的数据元素分别为(x_1，x_2，…，x_m)和(y_1，y_2，…，y_n)。写一个算法将它们合并为一个线性表C，使得：

当m≥n时，C＝x_1，y_1，x_2，y_2，…，x_n，y_n，…，x_m；

当m<n时，C＝y_1，x_1，y_2，x_2，…，y_m，x_m，…，y_n。

8．在一个非递减有序线性表中，插入一个值为x的元素，使插入后的线性仍为非递减有序。试分别用向量和单链表编写算法。

9．写一个算法将值为b的结点插在链表中值为a的结点之后。如果值为a的结点不存在，则插入在表尾。

第3章 栈和队列

3.1栈和队列引例

任一表达式都可看成是由操作数，运算符和界限符组成的一个串。其中，操作数可以是常数也可以是变量或常量的标识符，运算符可以是算术运算符，关系运算符和逻辑运算符等，界限符包括左右括号和表达式结束符等，例表达式7+4*（8-3）。计算机要完成表达式的求值，必须正确的解释表达式，将其翻译成正确的机器指令序列。要了解计算机的求值过程，必须先研究栈的性质。

3.2 栈

3.2.1栈的定义和基本运算

栈是限定只能在表尾进行插入和删除的线性表，并将表尾称为栈顶，表头称为栈底。图3—1给出了非空栈s=(A,B,C,D)的示意图。

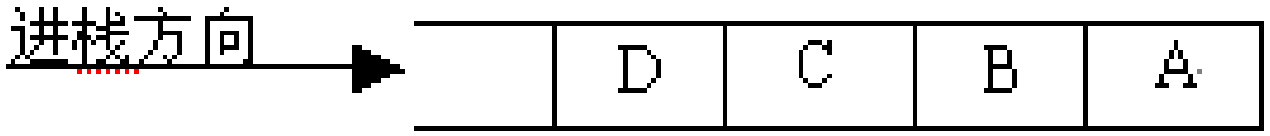


图3—1 非空栈示意图

由于限定只能在栈顶进行操作,所以栈中元素必按A,B,C,D次序进栈,,按D,C,B,A次序出栈,即按"后进先出"(或“先进后出")的原则进行操作的。这一特点可用生活中的实例形象说明。例如每次只能容一个人进出的死胡同就相当一个栈，胡同口相当于栈顶，而胡同的另一端则为栈底。

3.2.2栈的基本运算

（1）判栈空Empty(S). 若栈为空则返回“真“，否则返回”假“；

（2）入栈操作（压栈）Push(S,x) 将新元素压入栈中，使其成为栈顶元素；

（3）出栈操作（弹栈）Pop(S,x) 若栈不空则返回栈顶元素，并从栈中删除栈顶元素，否则返回NULL；

（4）取栈顶元素 Pettop(s,x) 若栈不空则返回栈顶元素，否则返回NULL；

（5）置空栈Clear(s) 将当前栈设定为空栈；

（6）求元素个数 CurrenSize(s) 求当前栈中元素的个数。

3.3 顺序栈的存储结构及算法实现

3.3.1顺序栈

顺序栈利用一组连续的存储单元存放从栈底到栈顶的诸元素。同时用一指针top指示当前栈顶元素的位置， C语言中用一维数组来描述。

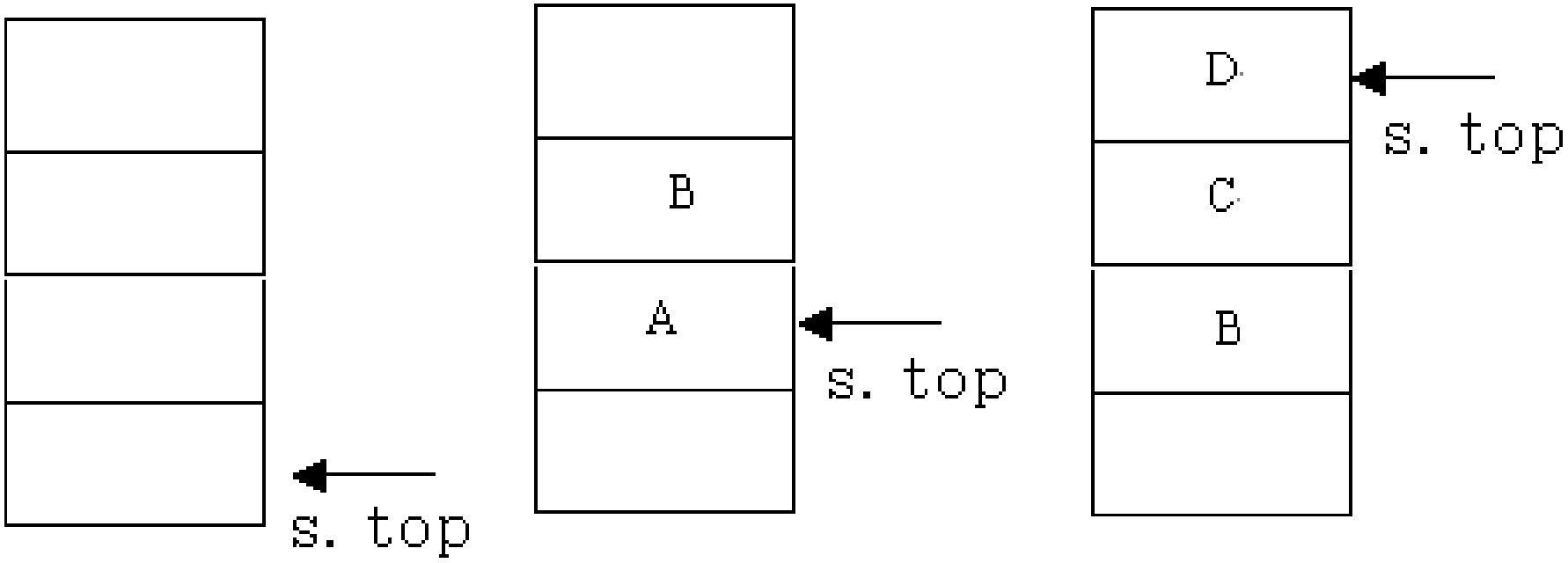
```
#define maxsize  N

typedef struct

  {Datatype   data[maxsize+1];

int top;

} Stack;
```



(a)空栈　　(b)一般情况　　(c)满栈

图3—2栈中元素和栈顶指针之间的关系

3.3.2顺序栈的基本运算的实现

判栈空，置空栈，求元素个数算法容易实现，只要判断或改变s.top的值即可。下面仅给出进栈，出栈，取栈顶元素等函数的算法实现。

1. 压栈

```
/*算法描述3-1*/
int Push(Stack S, Datatype x)
{if (S.top==maxsize) {printf("overflow\n");  return(0);}
 S.data[++S.top]=x;  return(1);
}
```

2. 出栈

```
/*算法描述3-2*/
int Pop(Stack s, Datatype x)
{if(S.top==0){printf("nudertflow\n");return(0)};
 x=S.data[S.top];  S.top--;
return(1);
}
```

3. 取栈顶元素

```
/*算法描述3-3*/

int Gettop(Stack S, Datatype x)

{if(S.top==0){printf("nodata\N");return(0);}

 x=S.data[top];  return(1);

}
```

3.4 链式栈

链式栈的组织形式和单链表类似, 其类型说明如下

```
Tyoedef struct Node
  {Datatype  data;
   struct Node  *next;
  }Node,  *LStack;
```

一个链栈由其栈顶指针唯一确定.设TOP是LStack形变量,则TOP指向栈顶元素。图3—3为链栈示意图 。top=NULL为判断栈空的条件。对链栈除非整个可用空间都被占满，否则不会出现栈满的情形。其操作是线性链表操作的特例，易实现. 请读者自行补充。

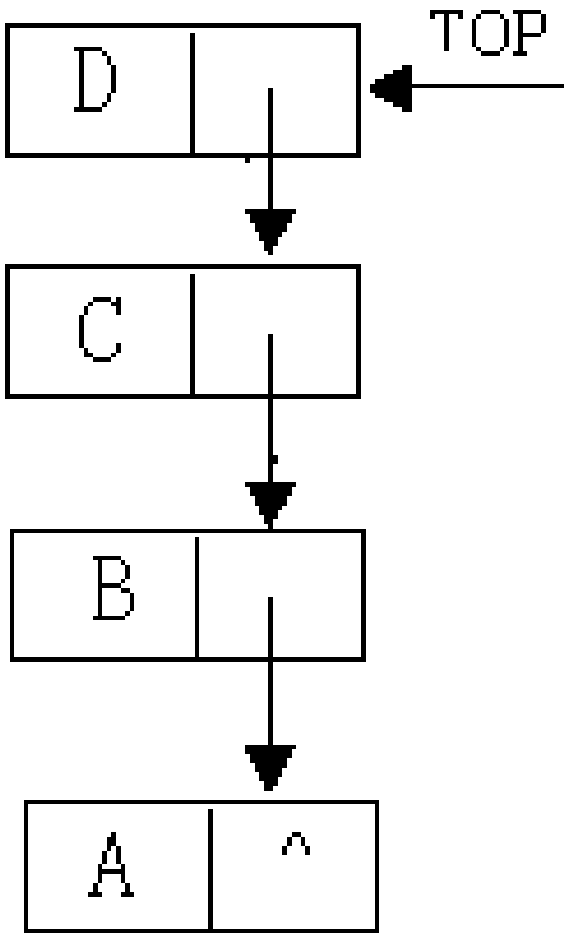


图3.3 链栈示意图

3.5 队列

3.5.1队列的定义和运算

和栈相反，队列是一种“先进先出”的线性表。他只允许再表的一端（称表尾）插入元素，在表的另一端（表头）删除元素。队列和日常生活中的排队是一致的，最早进入队列的元素最早得到服务。图3—4给出可队列示意图。

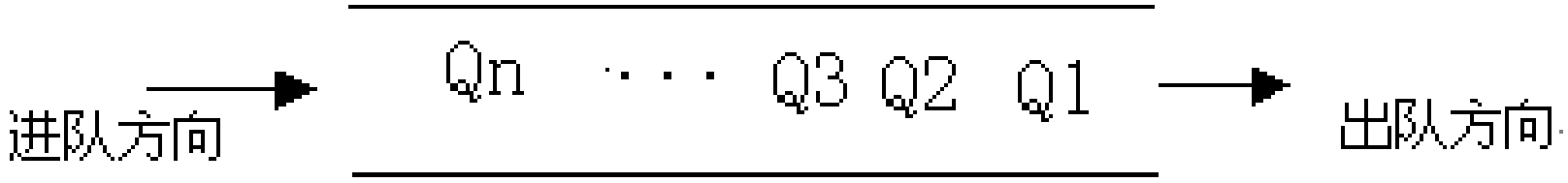


图3.4 队列示意图

队列的操作与栈的操作类似，不同的是删除运算是在表头进行。

(1)判队空 EmptyQueue(Q) 若队列为空则返回“真“，否则返回”假“;

(2)入队 InQueue(Q,x) 将新元素x插入到队尾;

(3) 出队 OutQueue(Q,x) 若队列不空则返回队首的第一个元素元素，并从队列中删除该元素，否则返回NULL;

（4）置空队列 InitQueue(Q) 将当队列设定为空队列;

（5）求元素个数 CurrentSize(Q) 返回当前队列中元素的个数 。

3.5.2.队列的存储结构及其算法实现

和栈类似，队列的顺序存储结构用一个向量来存储队列中的元素，不过还需两个指针来指示队头元素和对尾元素在队列中的当前位置。C语言描述如下:

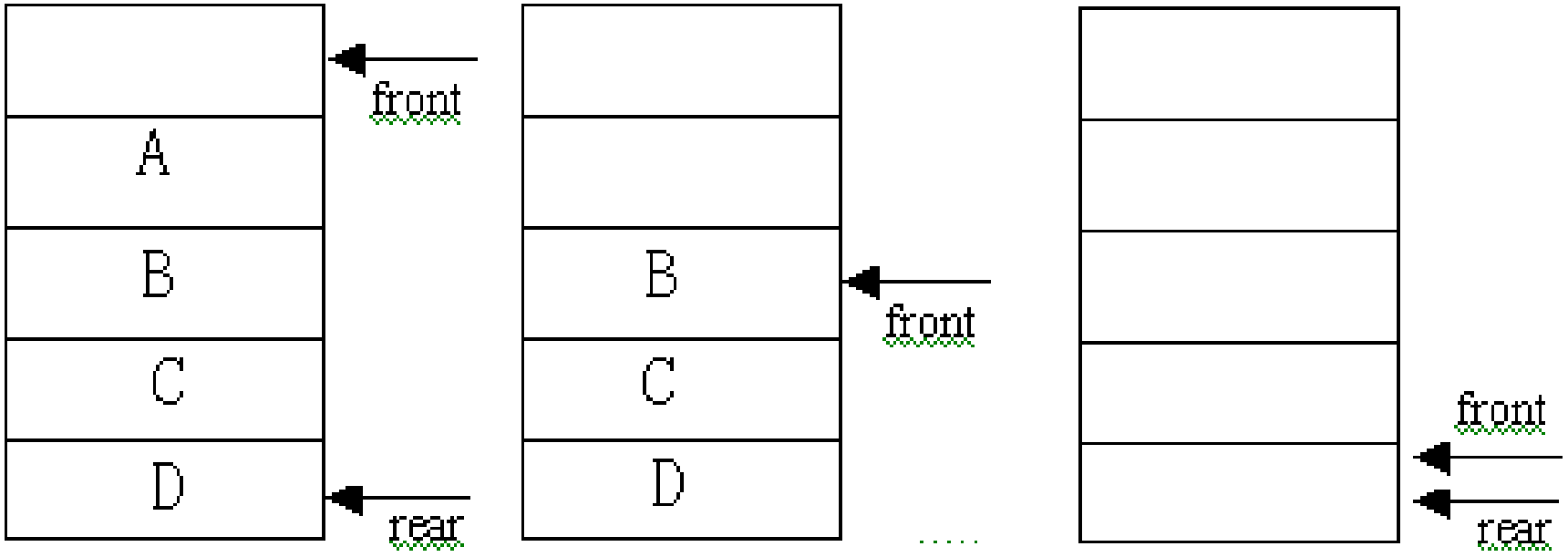
```
# define maxsize  N

typedef  stuct

    {Datatype  adta[maxsize+1];

     int  front, rear;

     } Queue;
```



(a) 满队　(b)一般情况　(c) 空队

图3—5顺序队列示意图

3.5.3 顺序队列的基本运算

(1)判队空

```
/*算法描述3-4*/

int EmptyQueue(Queue Q)

{if (Q.front==Q.rear) return(1);

 else return(0);}
```

2. 入队

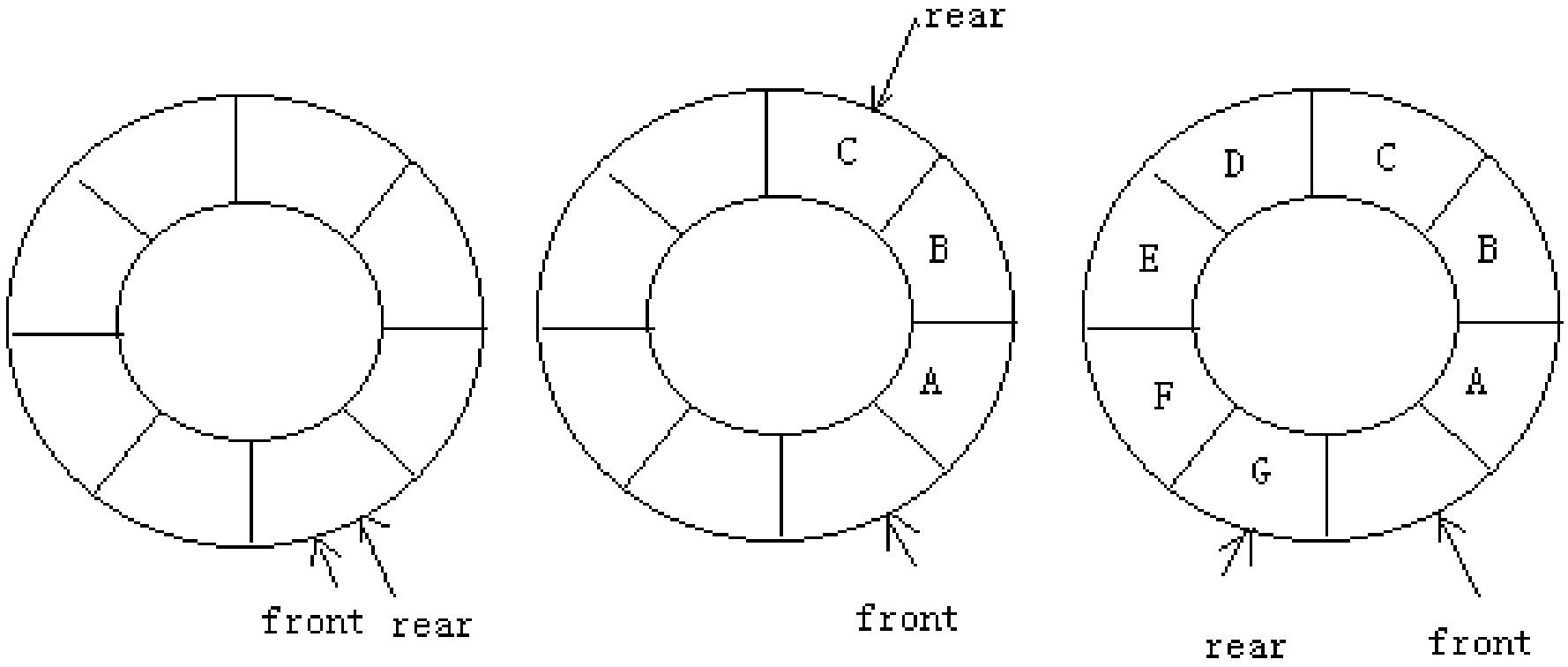
```
/*算法描述3.5*/
 int InQueue(Queue Q, Datatype x)
{if (Q.rear==maxsize) {printf("overflow!"); return(0);}
 Q.rear++;
 Q.data[Q.rear]=x;
 return(1);}
```

3．出队

```
/*算法描述3-6*/
int OutQueue(Queue Q, Datatype x)
{if(EmptyQueue(Q)){printf("underflow!"); return(0);}
x=Q.data[Q.front+1];
Q.front++;
if(EmptyQueue(Q)){Q.front=0; Q.rear=0}
return(1);}
```

3.5.4 循环队列

上述算法中判队列满的条件为Q.rear==maxsize. 用它来分析一下图3—5所示的队列状态, 此时,maxsize=5, Q.front=3,Q.rear=5, 显然不能作入队操作. 但队列中实际元素个数并未达到maxsize, 这种现象称之为假溢出. 解决这一问题的一个办法是在发生假溢出时将队列中全部元素向前移动指头指针为零, 但这样很浪费时间. 另一种办法是设想一个循环队列, 假想Q.data[1]接在Q.data[maxsize]之后, 如图3—6所示(由于循环队列的特点,队列中的元素可以存放在数组中下标从0到maxsize-1的共maxsize各位置).



(a) 空队　(b)非空队　(c)满队

图3.6 循环队列示意图

从上图不难看出, 无论空队还是满队都有Q.front=Q.rear. 从而无法判断属于那一种情况. 为此可少用一个元素空间, 而以队尾指针加1等于头指针来作为满队的标志.即,

队空： Q.front=Q.rear;

队满： Q.front=(Q.rear.+1)%maxsize;

循环队列的置空算法和非循环队列的置空算法相同, 其入队和出队算法如下,

(1) 入队

/*算法描述3-7*/

```
int InQueue(Queue Q, Datatype x)
{if ((Q.rear+1)% maxsize ==Q.front) {printf("overflow!"); return(0);}
 Q.rear=(Q.rear++)% maxsize
 Q.data[Q.rear]=x;
 Return(1);}
```

(2) 出队

```
/*算法描述3-8*/
nt OutQueue(Queue Q, Datatype x)
{if(EmptyQueue(Q)){printf("underflow!"); return(0);}
Q.front= Q.front++% maxsize
x=Q.data[Q.front];
return(1);}
```

3.6 实习：栈的应用实例

1．表达式的构成

任一表达式都可看成是由操作数，运算符和界限符组成的一个串。其中，操作数可以是常数也可以是变量或常量的标识符，运算符可以是算术运算符，关系运算符和逻辑运算符等，界限符包括左右括号和表达式结束符等，例表达式7+4*（8-3）。为论述方便，这里仅介绍简单算术表达式的求值问题。

2. 算符的优先关系

要对表达式正确求值，必须正确的解释表达式，将其翻译成正确的机器指令序列。例如，要对表达式3*（7-2）求值，首先要了解算术运算的规则。即

1）从左到右；

2）先括号内，后括号外；

3）先乘除，后加减；

故，该表达式的计算步骤应为3*（7-2）=3*5=15。

表3.1　算符优先级表

p1	+	-	*	/	(	)	@
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
@	<	<	<	<	<		=

表中空白表示运算符p1,p2不可能相遇的情况，若相遇则表明出现了语法错误。“#”是表达式的结束符，为算法方便在表达式的左边也虚设一个“#”使其配对。这样，当“（”=“）”时表示左右括号相遇，括号内运算已经结束；同理，当“#”=“#”时表示整个表达式求值完毕。

3．算法思路

使用两个栈S1和S2， 其中，S1为运算符栈，用以寄存运算符，而S2为操作数栈，用以寄存操作数或运算结果。其算法思路如下，

1）首先设置两栈为空，将“#”作为表达式起始符压入运算符栈S1作为栈底元素；

2）依次读入表达式的每个字符，若是操作数则进入操作数栈S2；若是运算符，则与S1的栈顶运算符比较优先级，若栈顶运算符优先级低，则进入栈S1，若栈顶运算符优先级高，则弹出S1的栈顶运算符，并从栈S2中弹出两个操作数，作相应运算后，将结果压入操作数栈S2，然后再次与S1的栈顶运算符比较优先级，直至栈顶运算符优先级低为止

3）当S1的栈顶运算符为“#”时，表达式求值结束，操作数栈S2中的数即为表达式的值。

表3.2求表达式3*(7-2)时的栈变化

步骤	OPTR 栈	OPND 栈	输入字符	主要操作
1	#		_3*(7-2)#	PUSH(OPND,'3')
2	#	3	_*(7-2)#	PUSH(OPTR,'*')
3	#*	3	_(7-2)#	PUSH(OPTR,'(')
4	#*(	3	_7-2)#	PUSH(OPND,'7')
5	#*(	3 7	_-2)#	PUSH(OPTR,'-')
6	#*(-	3 7	_2)#	PUSH(OPND,'2')
7	#*(-	3 7 2	_)#	operate('7','-','2')
8	#*(	3 5	_)#	POP(OPTR){消去一对括号}
9	#*	3 5	_#	operate('3','*','5')
10	#	1 5	_#	RETURN(GETTOP(OPND))

4.完整的C语言程序

file1

```
#define maxsize  30
typedef struct
{char   data[maxsize+1];
int top;
} Stack;
   int Push(S,x)
   Stack  S;
   Char  x;
   {if (S.top==maxsize) {printf("overflow\n");  return(0);}
     S.data[++S.top]=x;  return(1);
   }
```

```
int Pop(s,x)
Stack  S;
Char  x;
{if(S.top==0){printf("nudertflow\n");return(0)};
   x=S.data[S.top];  S.top--;
return(1);
}
file2
char  readtop(S)
Stack S;
{char  a;
```

```
Pop(S, a);
Push(S, a);
Return( a);
}
double operate(ch, x, y)
char  ch;
double x,y;
{double  z;
switch (ch)
{case  '+' :  z=x+y;  break;
 case  '-'  : z=x-y; break;
 case  '*'  : z=x*y; break;
```

```
    case  '/' :  z=x/y; break;}
return(z);}
int precede(p1,p2)
{ int  flag;
switch (p1)
{case  '+' :  if(p2=='*' || p2=='/' || p2== '(' ) flag=-1;
                else flag=1;
                break;
 case  '-'  : if(p2=='*' || p2=='/' || p2== '(' ) flag=-1;
                else flag=1;
                break;
 case  '*'  : if(p2=='(' ) flag=-1;
                else flag=1;
                break;
```

```
case  '/' : if(p2=='(' ) flag=-1;
            else flag=1;
            break;
case  '(' :  if(p2==')') flag=0;
             else  if(p2=='#')printf("error  operator!\n");
                   else flag=-1;
             break;
 case  ')'  : if(p2=='(' )printf("error  operator!\n");
            else flag=1;
            break;
```

```
case  '# '  : if(p2==')' )printf("error  operator!\n");
                else  if(p2=='#' ) flag=0;
                    else flag=-1;
                break;'}
return(flag);}
double calcul( a);
char  a[];
{Stack  S1, S2;
double  x, y, z;
char  r, ch;
int  I;
```

```
push(S1,'#');    r=a[I];
while(r<>'#' || readtop(S1)<>'#')
{if(r<='9' &&  r>='0'){ x=0;
                    while(r<='9' &&  r>='0')
                    {x=x*10+r-'0';
                     r=a[++I];}
                    push(S2,x);}
 else switch(precede(S1,r))
```

```
    {case  -1:  push(S1,r); r=a[++I]; break;
      case   0:   pop(S1,ch);  r=a[++I]; break;
      case   1: pop(S1, ch); pop(S2, x1); pop(S2, x2); push(S2,
operate(ch, x1.x2)); r=a[++I]; break;}}
return(read(S2));
}
```

习题3

1.假定有编号为A、B、C、D的4辆列车，顺序开进一个栈式结构的站台，请写出开出车站站台的列车顺序(注：每一列车由站台开出时均可进栈，出栈开出站台，但不允许出栈后回退)。写出每一种可能的序列。

2．已知堆栈采用链式存储结构，初始时为空，试画出a、b、c、d 4个元素依次以后堆栈的状态，然后再画出此时的栈顶元素出栈后的状态。

3.写出链栈的取栈顶元素和置栈空的算法。

4．写出多个链表栈中取第j个链表栈顶元素值的算法。

5．写出计算表达式3+4／25，8—6时操作数栈和运算符栈的变化情况。

6．课文中规定：无论是循环队列还是链表队列，队头指针总是指向队头元素的前一位置，队尾指针指向队尾元素。试画出有两个元素A、B的不同存储结构的图示，及将这两个元素出对后循环队列和链表队列的状态示意图。

7．对于一个具有m个单元的循环队列，写出求队列中元素个数的公式。

8．对于一个具有n个单元(n≥2)的循环队列，若从进入第一个元素开始，每隔T1个时间单位进入下一个元素，同时从进入第一个元素开始，每隔t2(t2≥t1)个时间单位处理一个元素并令其出队。试编写一个算法，求出在第几个元素进队时将发生溢出。

9．假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素结点（注意不设头指针），试编写出相应的置空队列、入队列和出队列的算法。

10．设有两栈共享一存储空间 stack[n] ，如图3-8所示，每个栈的大小不定试编写对任一栈进栈和出栈的算法push(x,i)和pop(i),I=1,2分别表示左右两个栈。上溢条件应该是整个存储空间全满。

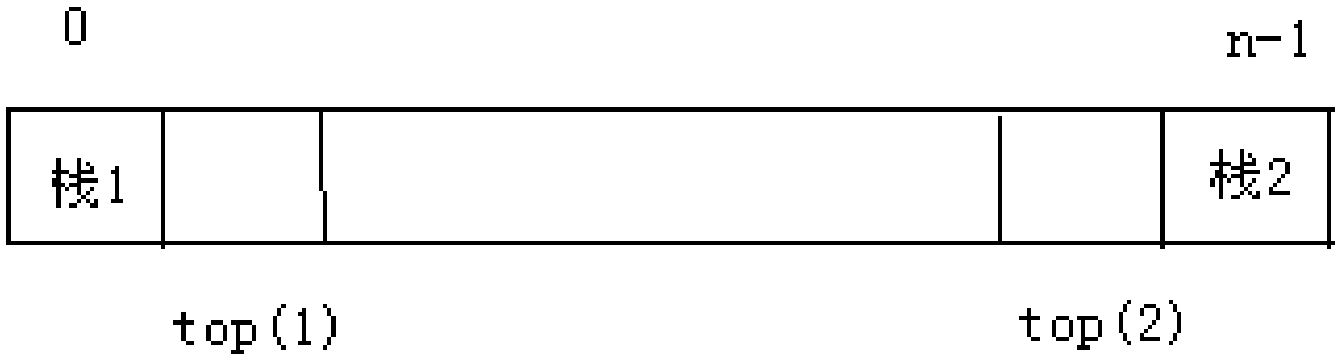


图3.7 第10题图

第4章　串

4.1 串的基本概念

串(String)是由零个或多个字符组成的有限序列。一般记作：

$$S= " a_1\ a_2 \ldots a_n " \quad (n\geq 0)$$

其中S是串名，用双引号括起来的字符序列为串值，引号是界限符，$a_i(1\leq i\leq n)$是一个任意字符(字母、数字或其他字符)，它称为串的元素，是构成串的基本单位，串中所包含的字符个数n称为串的长度，当n=0时，称为空串。

将串值括起来的双引号本身不属于串，它的作用是避免串与常数或与标识符混淆。

例如，A="123"是数字字符串，长度为3，它不同于整常数123；B= " lx " 是长度为2的字符串，而lx通常表示一个标识符。

常将仅由一个或多个空格组成的串称为空白串。注意空串和空白串的不同，例如""和""分别表示长度为1的空白串和长度为0的空串。

串中任意连续的字符组成的子序列称为该串的子串。包含子串的串相应地称为主串。

通常称字符在序列中的序号为该字符在串中的位置。子串在主串中的位置则以子串的第一个字符首次出现在主串中的位置来表示。

例如，设有两个字符串C和D：

C= " This is a string. "

D= " is "

则它们的长度分别为17、2；D是C的子串，C为主串。D在C中出现了两次，其中首次出现所对应的主串位置是3。因此，称D在C中的序号(或位置)为3。

若两个串的长度相等且对应字符都相等，则称两个串是相等的。当两个串不相等时，可按“字典顺序”区分大小。

串也是线性表的一种，因此串的逻辑结构和线性表极为相似，区别仅在于串的数据对象限定为字符集。

串的运算有很多，下面介绍部分基本运算。

(1) 串赋值 StrAssign(&s,chars)：已知串常量chars，生成一个值等于chars的串s。

(2) 求串长 StrLength(s)：已知串s，返回串s的长度。

(3) 串连接 StrConcat (&s1,s2)：已知串s1,s2，在s1的后面连接s2的串值。

(4) 求子串 SubStr(s,i,len)：已知串s，返回从串s的第i个字符开始的长度为len的子串。

(5) 串比较 StrCmp(s1,s2)：已知串s1,s2，若s1==s2，操作返回值为0；若s1<s2, 返回值小于0；若s1>s2, 返回值大于0。

(6) 子串定位 StrIndex(s,t)：已知串s,t，找子串t在主串s中首次出现的位置，即若t∈s，则操作返回t在s中首次出现的位置，否则返回值为－1。

(7) 串插入 StrInsert(&s,i,t)：已知串s,t，将串t插入到串s 的第i个字符位置上。

(8) 串删除 StrDelete(&s,i,len)：已知串s，删除串s 中从第i个字符开始的长度为len的子串。

(9) 串替换 StrRep(&s,t,r)：已知串s,t,r，用串r 替换串s中出现的所有与串t相等的不重叠的子串。

(10) 销毁串 StrDestroy(&s)：已知串s，销毁串s。

以上是串的一些基本操作。其中前 5 个操作是最为基本的，它们不能用其他的操作来合成，因此通常将这 5 个基本操作称为最小操作集，反之，其他操作可在这个最小操作集上实现。

4.2 串的存储结构

4.2.1 串的顺序存储

串的顺序存储结构简称为顺序串。

与顺序表类似，顺序串是用一组地址连续的存储单元来存储串中的字符序列的。因此可用高级语言的字符数组来实现，按其存储分配的不同可将顺序串分为静态存储分配的顺序串和动态存储分配的顺序串两类。

1. 静态存储分配的顺序串

所谓静态存储分配，是指按预定义的大小为每一个串变量分配一个固定长度的存储区，即串值空间的大小在编译时刻就已确定，是静态的。所以串空间最大值为MAXSIZE时，最多只能放MAXSIZE－1个字符。其类型描述如下：

```
#define MAXSIZE  256          /*该值依赖于应用，由用户定义*/

typedef struct

{char ch[MAXSIZE];            /*256个字符依次存储在
                                ch[0]..ch[MAXSIZE－1]中*/

int len;

}SString;                     /*SString是顺序串类型，则串的最
                                大长度不能超过255*/

SString  s;                   /*定义串变量s*/
```

在直接使用定长的字符数组存放串内容外，一般可使用一个不会出现在串中的特殊字符放在串值的末尾来表示串的结束。例如，C语言中以字符'\0'表示串值的终结。

C语言中串的静态存储结构如图4.1所示。

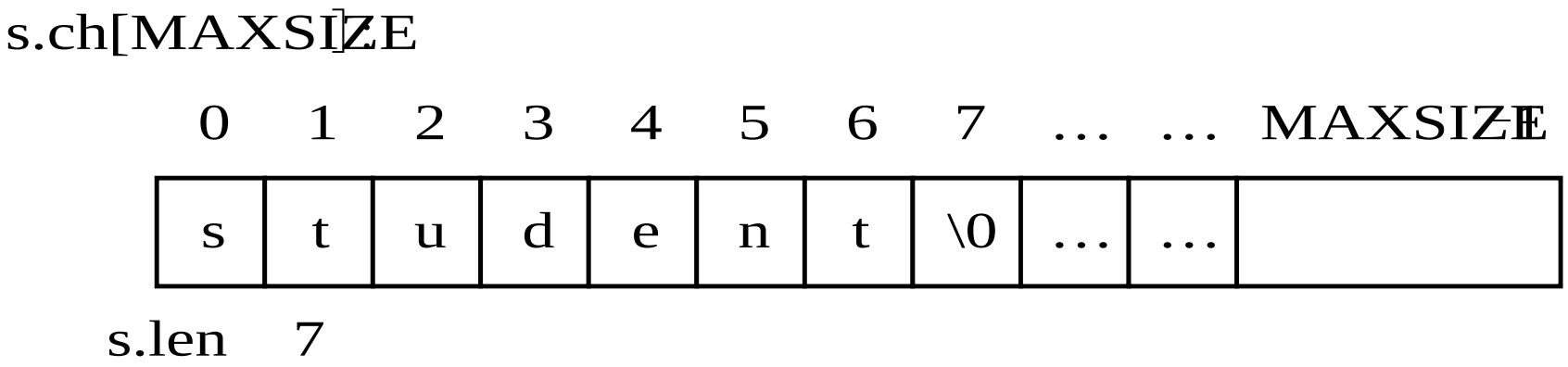


图4.1 C语言中串的静态存储结构

2. 动态存储分配的顺序串(堆串)

这种存储表示的特点是，仍以一组地址连续的存储单元存放串值字符序列，但它们的存储空间是在程序执行过程中动态分配而得的。系统将一个地址连续、容量很大的存储空间作为字符串的可用空间，每当建立一个新串时，系统就从这个空间中分配一个大小和字符串长度相同的空间存储新串的串值。

假设以一维数组heap[MAXSIZE]表示可供字符串进行动态分配的存储空间，并设一整型变量free指向heap中未分配区域的开始地址。在程序执行过程中，当生成一个新串时，就从free指示的位置起为新串分配一个所需大小的存储空间，同时记录该串的相关信息。这种存储结构称为堆结构。动态分配存储空间的顺序串也叫堆串。堆串可定义如下：

```
typedef struct
{ int length;
  int start;
}HeapString;
```

在C语言中，已经有一个称为“堆”的自由存储空间，并可利用malloc()和free()等动态存储管理函数，根据实际需要动态分配和释放字符数组空间，如图4.2所示。其类型可描述如下：

```
typedef struct
{ char *ch;   /*指示串的起始地址，可按实际的串长分配存储区
*/
  int len;
}HString;
HString s;   /*定义一个串变量*/
```

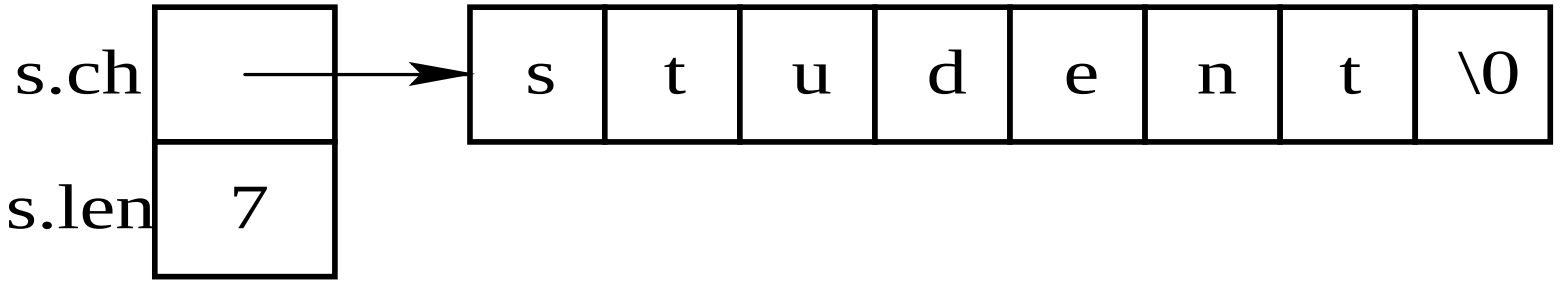


图4.2 顺序串的动态存储结构

在程序中，可根据实际需求为这种类型的串变量动态分配存储空间，这样做非常有效、方便，只是在程序执行过程中要不断地生成新串和销毁旧串。

4.2.2 串的链式存储

顺序串上的插入和删除操作极不方便，需要移动大量的字符。因此，我们可用单链表方式来存储串值，串的这种链式存储结构简称为链串，如图4.3所示。

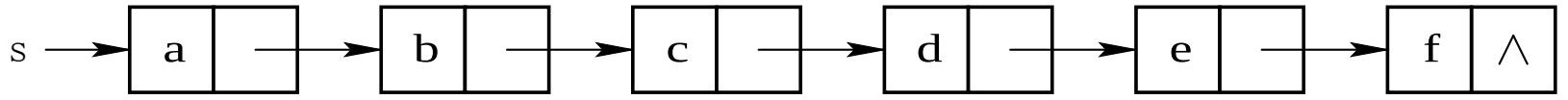


图4.3 结点大小为1的链串s

链串的类型可描述如下：

```
typedef struct node
{ char ch;
  struct node *next;      /*next为指向结点的指针*/
}LString;
LString  s;               /*定义一个串变量s*/
```

一个链串由头指针惟一确定。

这种结构便于进行插入和删除运算，但存储空间利用率太低。

为了提高存储密度，可使每个结点存放多个字符。如图4.4所示，通常将结点数据域存放的字符个数定义为结点的大小，显然，当结点大小大于1时，串的长度不一定正好是结点的整数倍，因此要用特殊字符来填充最后一个结点，以表示串的终结。

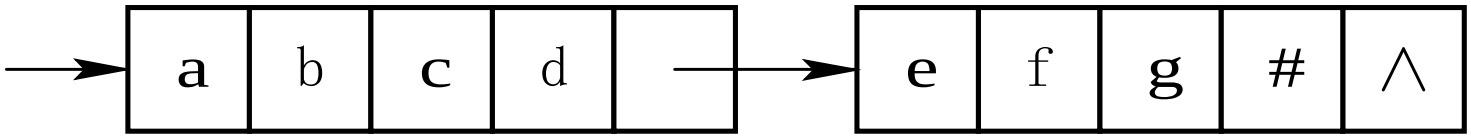


图4.4 结点大小为4的链串

对于结点大小不为1的链串，其类型定义只需对上述的结点类型做简单的修改即可。

```
#define nodesize 80
typedef struct node
{ char data[nodesize];
  struct node *next;
}LString;
```

虽然增大结点的数据域使得存储密度增大，但是做插入、删除运算时，需要考虑结点的分拆与合并，可能会引起大量字符的移动，给运算带来不便。图4.5所示为在图4.4所示链串的第三个字符之后插入“xyz”后的链串。

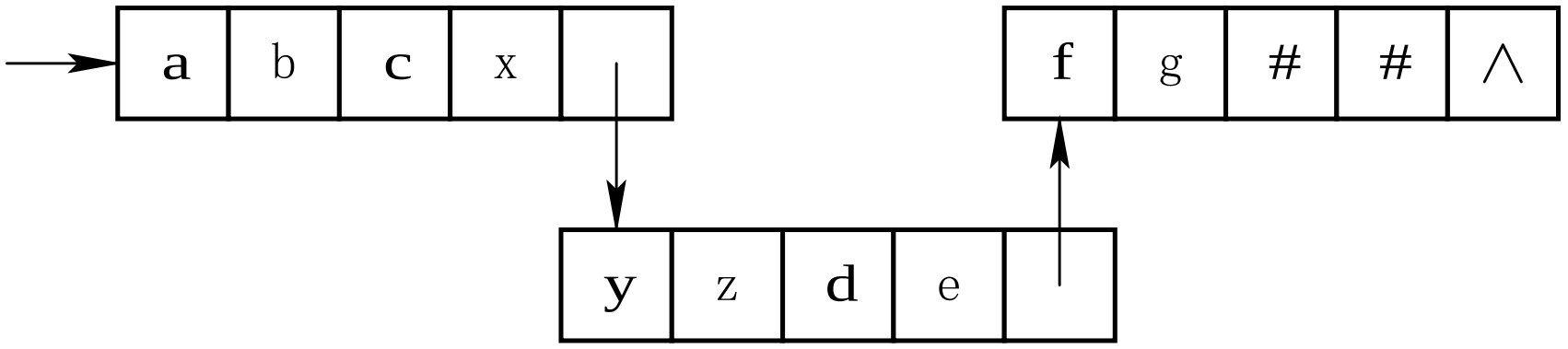


图4.5 链串的插入

4.3 串的基本运算的实现

1. 求子串运算(采用静态存储顺序串)

```
/*算法描述4.1  求子串*/
int StrSub(SString *sub, SString s, int pos, int len)
/* 用sub返回串s中序号pos开始的长度为len 的子串*/
{ int i;
if (pos<0 || pos>s.len || len<1 || len>s.len-pos)
{ sub->len=0;  return(0); }                    /*子串起始位置及长度是否合适*/
else{
for(i=0;i<len;i++) sub->ch[i]=s.ch[i+pos];
sub->[len]= '\0';                              /*子串结束*/
sub->len=len;
return(1);
}
}
```

2. 定位运算(采用静态存储顺序串)

串的定位运算也称为串的模式匹配，是一种重要的串运算。

设s和t是给定的两个串，在主串s中找到等于子串t的过程称为模式匹配，如果在s中找到等于t的子串，则称匹配成功，函数返回t在s中的首次出现的存储位置(或序号)，否则匹配失败，返回－1。t也称为模式。

算法思想如下：首先将s_0与t_0进行比较，若不同，就将s_1与t_0进行比较……，直到s的某一个字符s_i和t_0相同，再将它们之后的字符进行比较，若也相同，则如此继续往下比较，当s的某一个字符s_i与t的字符t_j不同时，则s返回到本趟开始字符的下一个字符，即s_{i-j+1}，t返回到t_0，继续开始下一趟的比较，重复上述过程。若t中的字符全部比较完，则说明本趟匹配成功，本趟的起始位置是i－j，否则，匹配失败。

设主串s= " ababcabcacbab " ，模式t= " abcac " ，匹配过程如图4.6所示。

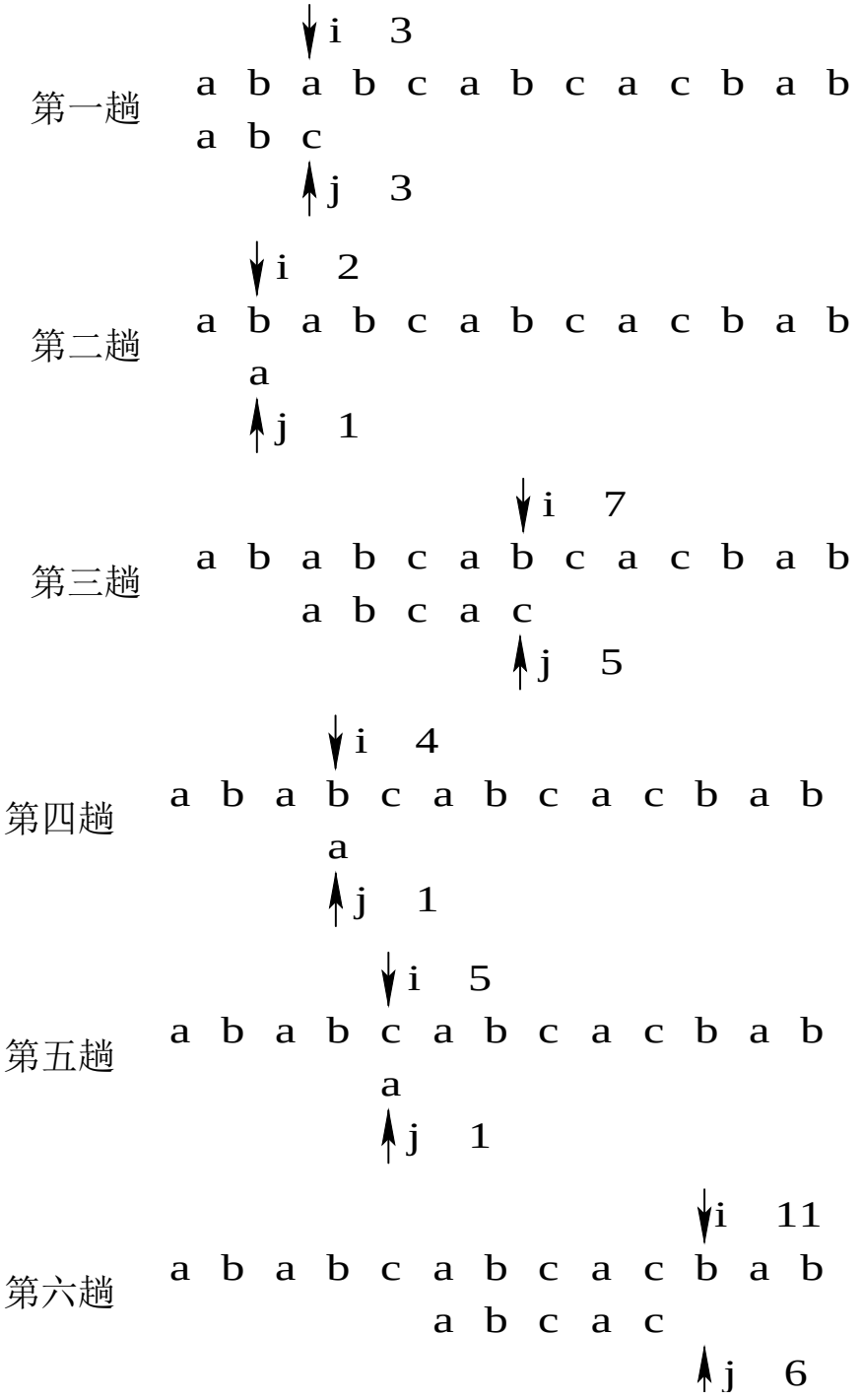


图4.6 简单模式匹配的匹配过程

```
/*算法描述4.2  模式匹配*/
int StrIndex(SString s,int pos,SString t)
   /*求串t在串s中的位置*/
{ int i,j;
  if(t.len==0) return(0);
  i=pos;j=0;
  while(i<s.len && j<t.len)              /*都没遇到结束符*/
        if(s.ch[i]==t.ch[j])
           { i++;j++; }                  /*继续*/
         else
           {i=i-j+1; j=0; }              /*回溯*/
   if (j>=t.len)  return(i-j);           /*匹配成功，返回存储位置*/
   else  return(-1);
}
```

3．插入运算(采用动态存储串)

```
/*算法描述4.3  串插入*/
StrInsert(HString *s, int pos, HString t)
   /*在串s的第pos个字符之前插入串t*/
{ char *temp;
  int i;
  if(pos<0 || pos>s->len) return(0);              /*pos不合理*/
  if(t.len)                                       /*t非空*/
{ temp=(char*)malloc((s->len+t.len)*sizeof(char));
                  /*临时变量，用来暂存插入后的结果*/
```

```
    if(temp==NULL) return(0);
    for(i=0;i<pos;i++) temp[i]=s->ch[i];
    for(i=0;i<t.len;i++) temp[i+pos]=t.ch[i];
    for(i=pos;i<=s->len;i++) temp[i+t.len]=s->ch[i];
    s->len+=t.len;
    free(s->ch);                              /*释放原串s*/
    s->ch=temp;                               /*s获得相加结果*/
    return(1);
  }
}
```

4．连接运算(采用动态存储串)

```
/*算法描述4.4  串连接*/
StrCat(HString *s,HString t)            /*将串t连接在串s的后面*/
{ char *temp;
  int i;
  temp=(char *) malloc((s->len+t.len)*sizeof(char));
  if(temp==NULL) return(0);
  for(i=0;i<=s->len;i++) temp[i]=s->ch[i];     /*复制s串*/
  for(i=s->len;i<=s->len+t.len;i++)            /*复制t串*/
    temp[i]=t.ch[i-s->len];
  s->len+=t.len;
  free(s->ch);
  s->ch=temp;
  return(1);
}
```

5．串定位(采用链串存储)

```
/*算法描述4.5  串定位*/
LString *lindex(LString s,LString t)
  /*求串t在串s中的位置，返回指向t串起始位置的指针*/
{ LString *loc,*p,*q;
loc=s;
p=loc;q=t;
while(p && q)                          /*当t、s串均未结束时*/
   { if(p->data==q->data)              /*字符匹配时，指针后移*/
```

```
        { p=p->next;
          q=q->next;
         }
         else                                  /*字符不匹配时，回溯*/
        { loc=loc->next;
          p=loc;
          q=t;
        }
     }
   if(q==NULL) return(loc);                    /*匹配完成，返回*/
          else  return(NULL);
   }
```

4.4 实习：串运算实例

文本编辑程序用于源程序的输入和修改，公文书信、报刊和书籍的编辑排版等。常用的文本编辑程序有Word、WPS等。文本编辑的实质是修改字符数据的形式和格式，虽然各个文本编辑程序的功能强弱不同，但基本操作是一样的，都包括串的查找、插入和删除等操作。

这一节我们来实现一个简单的文本编辑操作演示程序，包括字符串的部分基本运算：赋值、比较、连接、插入、删除和清除运算。字符串采用动态存储结构(即堆分配)存储。

```
#define NULL 0
typedef struct
    { char *ch;
    int len;
    }HString;
int StrAssign(HString *s,char *chars)
{ int i=0,slen;
  if(s->ch!=NULL) free(s->ch);
  while(chars[i]!='\0') i++;
  slen=i;
  if(slen){
```

```
    s->ch=(char *)malloc(slen*sizeof(char));
    if(s->ch==NULL) return(0);
    for(i=0;i<=slen;i++) s->ch[i]=chars[i];
   }
   else s->ch=NULL;
   s->len=slen;
   return(1);
}
int StrCompare(HString s,HString t)
{ int i;
   for(i=0;i<s.len&&i<t.len;i++)
```

```
        if (s.ch[i]==t.ch[i]) return(0);
        else if(s.ch[i]>t.ch[i]) return(1);
                else return(-1);
}
StrCat(HString *s,HString t)                    /*参见算法4.4*/
StrInsert(HString *s, int pos, HString t)       /*参见算法4.3*/
StrDelete(HString *s, int pos,int len)
{ int i;
  char *temp;
  if(pos<0 || pos>s->len-len) return(0);
  if(len){
```

```
    temp=(char*)malloc((s->len-len)*sizeof(char));
    if(temp==NULL) return(0);
    for(i=0;i<pos;i++) temp[i]=s->ch[i];
    for(i=pos;i<=s->len-len;i++) temp[i]=s->ch[i+len];
    s->len-=len;
    free(s->ch);
    s->ch=temp;
    return(1);
   }
  }
```

```
int ClearString(HString *s)
{ if(s->ch)
  { free(s->ch);
    s->ch=NULL;
    s->len=0;
  }
  return(0);
}
main()
{ int inp,flag=1;
  char *s,*t;
  HString s1,s2,*res;
```

```
int pos,len,ret;
printf("1-------StrAssing\n");
printf("2-------StrCompare\n");
printf("3-------StrCat\n");
printf("4-------StrInsert\n");
printf("5-------StrDelete\n");
printf("6-------ClearString\n");
printf("7-------Exit");
printf("please input 1--7\n\n");
while(flag)
{ scanf("%d",&inp);
  switch(inp){
    case 1:{
```

```
            scanf("%s",s);
             ret=StrAssign(&s1,s);
             if(ret!=0) printf("the string is:%s\n",s1.ch);
             else   printf("error\n");
             break;}
        case 2:{
             printf("s=");  scanf("%s",s);
             StrAssign(&s1,s);
             printf("t=");  scanf("%s",t);
             StrAssign(&s2,t);
             ret=StrCompare(s1,s2);
             switch(ret){
```

```
        case 0:  printf("two strings are equle\n"); break;
        case 1:  printf("the first string > the second string\n"); break;
        case -1:  printf("the first string < the second string\n"); break;
        }
        break;}
case 3:{
  printf("s=");  scanf("%s",s);
  StrAssign(&s1,s);
  printf("t=");  scanf("%s",t);
  StrAssign(&s2,t);
  StrCat(&s1,s2);
  printf("s1 cat s2 is:%s\n",s1);
  break;}
```

```
case 4:{
        printf("s=");  scanf("%s",s);
        StrAssign(&s1,s);
        printf("t=");  scanf("%s,",t);
        StrAssign(&s2,t);
        printf("pos=");  scanf("%d",&pos);
        StrInsert(&s1,pos,s2);
        printf("result:%s\n",s1);
        break;}
```

```
case 5:{
        printf("s=");  scanf("%s,",s);
        StrAssign(&s1,s);
        printf("pos=");  scanf("%d",&pos);
        printf("len=");  scanf("%d",&len);
        StrDelete(&s1,pos,len);
        printf("result: %s\n",s1);
        break;}
```

```
        case 6:{
              printf("s="); scanf("%s",s);
              StrAssign(&s1,s);
              ret=ClearString(&s1);
              if(ret==0) printf("the string is NULL\n");
              else printf("error\n");
              break;}
            case 7:flag=0;break;
             }
          }
        }
```

习 题 4

1. 简述下列每对术语的区别：空串和空白串，串常量和串变量，主串和子串，静态分配的顺序串和动态分配的顺序串。

2. 设s="I am a student"，t="good"，q="programer"。给出下列操作的结果：

(1) StrLength(s)

(2) SubString(sub1,s,1,7)

(3) StrIndex(s, 'a',4)

(4) StrReplace(s, 'student',q)

(5) Strcat(StrCat(sub1,t))

3. 利用C的库函数strlen,strcpy和strcat写一算法void StrInsert(char *S, char *T, int i),将串T插入到串S的第i个位置上。若i大于S的长度，则插入不执行。

4. 利用C的库函数strlen 和strcpy写一算法void StrDelete(char *S,int i, int m)，删去串S中从位置i开始的连续m个字符。若i≥strlen(S)，则没有字符被删除；若i+m≥strlen(S),则将S中从位置i开始直至末尾的字符均删去。

5. 以HString为存储表示，写一个求子串的算法。

6. 一个文本串可用事先给定的字母映射表进行加密。例如，设字母映射表为：

a	b	c	d	e	f	g	h	I
j	k	l	m	n	o	p	q	r
s	t	u	v	w	x	y	z	
n	g	z	q	t	c	o	b	
m	u	h	e	l	k	p	d	a
w	x	f	y	I	v	r	s	j

则字符串"encrypt"被加密为"tkzwsdf"。试写一算法将输入的文本串进行加密后输出；另写一算法，将输入的已加密的文本串进行解密后输出。

7. 写一算法void StrReplace(char *T, char *P, char *S)，将T中首次出现的子串P替换为串S。注意：S和P的长度不一定相等。

8. 若S和T是用结点大小为1的单链表存储的两个串，试设计一个算法找出S中第一个不在T中出现的字符。

第5章 数 组

5.1 数组的定义和运算

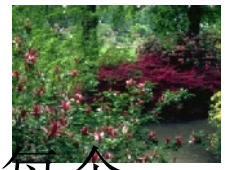
数组是我们很熟悉的一种数据类型，很多高级语言都支持这种数据类型。从逻辑结构上看，数组可以看作一般线性表的推广。数组作为一种数据结构，其特点是结构中的元素本身可以是具有某种结构的数据，但属于同一数据类型，比如：一维数组可以看作一个线性表，二维数组可以看作“数据元素是一维数组”的一维数组，三维数组可以看作“数据元素是二维数组”的一维数组，依次类推。

例如，对线性表A＝(A_0，A_1，A_2，…，A_{n-1})，如果其中的任意元素A_j($0 \le j \le n-1$)是简单类型，则A是一个一维数组，如果A_j本身也是一个线性表，即A_j＝(a_{1j}，a_{2j}，…，$a_{m-1,\ j}$)，则A是一个二维数组，如图5.1所示。

$$A\ (A_0,\ A_1,\ \cdots,\ A_j,\ \cdots,\ A_{n-1})$$

$$A_{m\times n}=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0j} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1j} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{ij} & \cdots & a_{i,n-1} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,j} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

图5.1 由线性表推广得到的二维数组



同理，三维数组可以看成是这样的一个线性表，其中每个数据元素均是一个二维数组。依次类推，可以得到多维数组。

从数组的特殊结构，我们可以看出，数组中的每一个元素由一个值和一组下标来描述。“值”代表数组中元素的数据信息，一组下标用来描述该元素在数组中的相对位置信息。数组的维数不同，描述其对应位置的下标的个数也不同。例如，在二维数组中，元素aij由两个下标值i，j来描述，其中i表示该元素所在的行号，j表示该元素所在的列号。同样，我们可以将这个特性推广到多维数组，对于n维数组而言，其元素由n个下标值来描述其在n维数组中的相对位置。

根据数组的结构特性可以看出，数组实际上是一组有固定个数的元素的集合。也就是说，一旦定义了数组的维数和每一维的上下限，数组中元素个数就固定了。例如二维数组$A_{3\times4}$，它有3行、4列，即由12个元素组成。由于这个性质，使得对数组的操作不像对线性表的操作那样可以在表中任意一个合法的位置插入或删除一个元素。通常在各种高级语言中数组一旦被定义，每一维的大小及上下界都不能改变。对于数组的操作一般只有以下几种：

(1) 构造数组；

(2) 撤消数组；

(3) 取值操作：给定一组下标，读取对应的数据元素值；

(4) 赋值操作：给定一组下标，存储或修改与其相对应的数据元素值。

我们着重研究二维数组，因为它的应用最为广泛。

5.2 数组的顺序存储和实现

对于数组A，一旦给定其维数n及各维长度bi(1≤i≤n)，则该数组中元素的个数是固定的，不能对数组做插入和删除操作，不涉及移动数据元素操作，因此对于数组而言，采用顺序存储方式比较合适。

我们知道，计算机内存储器的结构是一维的，因此对于一维数组按下标顺序分配即可，而对多维数组，就必须按照某种次序将数据元素排成一个线性序列，然后将这个线性序列存放在存储器中。

数组的顺序存储结构有两种：一是以行为主序(或先行后列)的顺序存放，如BASIC、PASCAL、COBOL、C等程序设计语言中用的是以行为主的顺序分配，即一行分配完了接着分配下一行；另一种是以列为主序(先列后行)的顺序存放，如FORTRAN语言中，用的是以列为主序的分配顺序，即一列一列地分配。以行为主序的分配规律是：最右边的下标先变化，即最右下标从小到大，循环一遍后，右边第二个下标再变，...，从右向左，最后是左下标。以列为主序分配的规律恰好相反：最左边的下标先变化，即最左下标从小到大，循环一遍后，左边第二个下标再变，...，从左向右，最后是右下标。

例如，二维数组$A_{m\times n}$以行为主序的存储序列为：

a_{00}，a_{01}，…，$a_{0,n-1}$，a_{10}，a_{11}，…，$a_{1,n-1}$，…，$a_{m-1,0}$，$a_{m-1,1}$，…，$a_{m-1,n-1}$

而以列为主序的存储序列为：

a_{00}，a_{10}，…，$a_{m-1,0}$，a_{01}，a_{11}，…，$a_{m-1,1}$，…，$a_{0,n-1}$，$a_{1,n-1}$，…，$a_{m-1,n-1}$

例如，一个2×3的二维数组，逻辑结构可以用图5.2(a)表示。以行为主序的内存映像如图5.2(b)所示，分配顺序为a_{00}，a_{01}，a_{02}，a_{10}，a_{11}，a_{12}；以列为主序的分配顺序为a_{00}，a_{10}，a_{01}，a_{11}，a_{02}，a_{12}，它的内存映像如图5.2(c)所示。

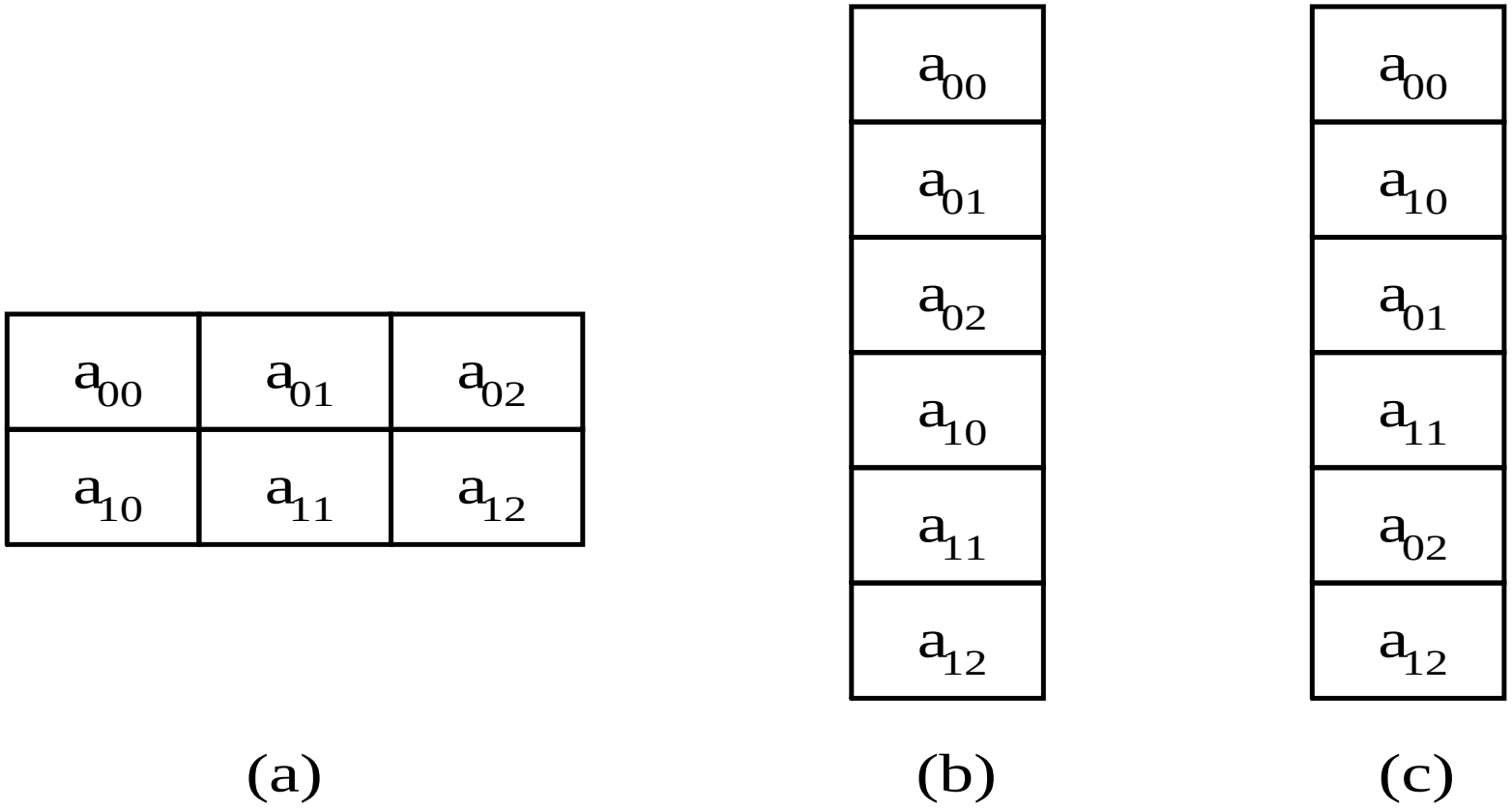


图5.2 2×3数组存储
(a) 逻辑状态；(b) 以行为主序；(c) 以列为主序

假设有一个3×4×2的三维数组A，共有24个元素，其逻辑结构如图5.3所示。

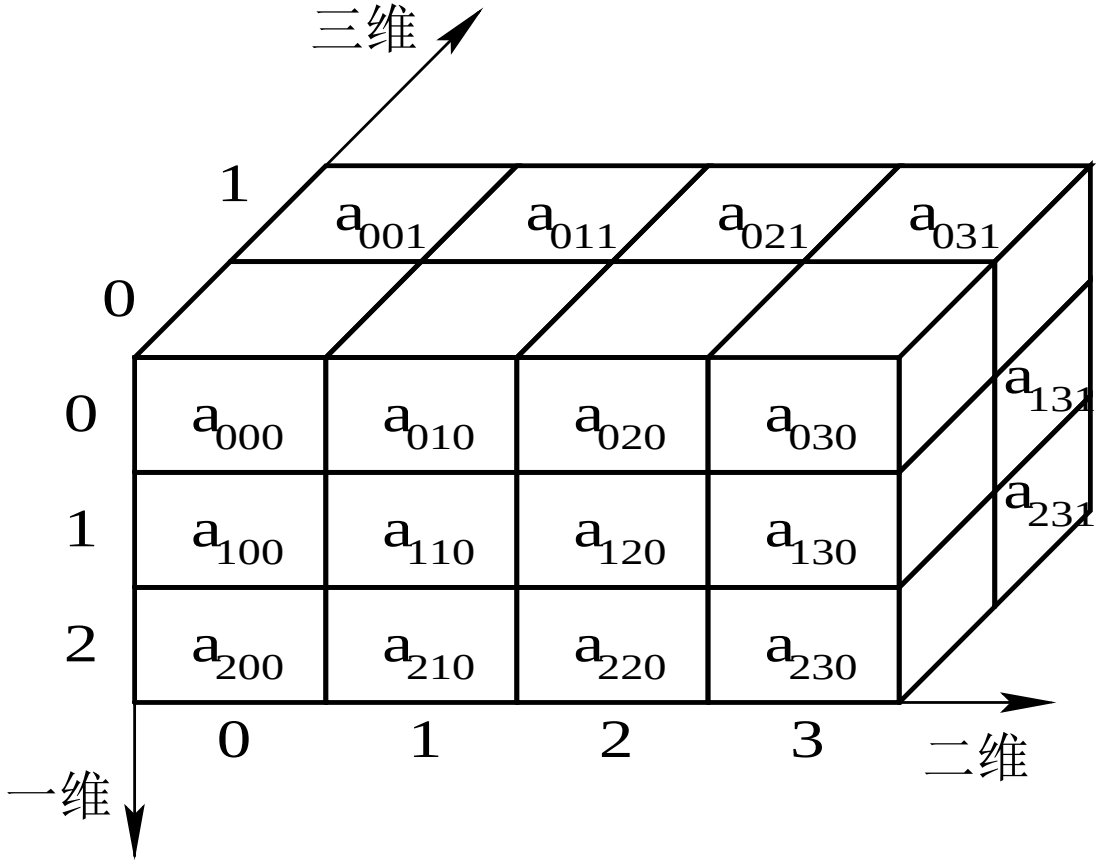


图5.3 三维数组的逻辑结构

三维数组元素的标号由三个数字表示。如果对$A_{3\times4\times2}$采用以行为主序的方式存放，则顺序为：

a_{000}，a_{001}，a_{010}，a_{011}，…，a_{220}，a_{221}，a_{230}，a_{231}

采用以列为主序的方式存放，则顺序为：

a_{000}，a_{100}，a_{200}，a_{010}，…，a_{221}，a_{031}，a_{131}，a_{231}

以上的存放规则可推广到多维数组的情况。总之，知道了多维数组的维数，以及每维的上下界，就可以方便地将多维数组按顺序存储结构存放在计算机中了。同时，根据数组的下标，可以计算出其在存储器中的位置。因此，数组的顺序存储是一种随机存取的结构。

下面，以“以行为主序”的分配为例，讨论数组中数据元素存储位置的计算。

设有二维数组$A_{m\times n}$，，下标从0开始，假设每个数组元素占size个存储单元，首元素a_{00}的存储地址为Loc[0，0]，对任意元素a_{ij}来说，因为a_{ij}是排在第i行、第j列，前面的i行有n×i个元素，第i行第j列个元素前面还有j个元素，所以，可得a_{ij}的地址计算公式如下：

$$Loc[i，j] = Loc[0，0] + (i\times n + j) \times size$$

同理，三维数组$A_{r\times m\times n}$可以看成是r个m×n的二维数组，若首元素的存储地址为Loc[0，0，0]，则元素a_{i11}的存储地址为Loc[i，1，1] = Loc[0，0，0] + (i×m×n) × size，这是因为在该元素之前，有i个m×n的二维数组。由a_{i11}的地址和二维数组的地址计算公式，不难得到三维数组任意元素a_{ijk}的地址计算公式如下：

Loc[i，j，k] = Loc[0，0，0] + (i×m×n + j×n＋k) × size

其中0≤i≤r－1，0≤j≤m－1，0≤k≤n－1。

数组是各种高级语言中已经实现的数据结构。在高级语言的应用层上，一般不会涉及到数据元素的存储地址的计算，这一计算内存地址的任务是由高级语言的编译系统完成的。当我们使用数组进行程序设计时，只需给出数组的下标范围，编译系统将根据用户提供的必要参数进行地址分配，存取数据元素时，也只要给出下标，而不必考虑其内存情况。

例5.1 若矩阵$A_{m\times n}$中存在某个元素a_{ij}，且满足a_{ij}是第i行中最小值且是第j列中的最大值，则称该元素为矩阵A的一个鞍点。试编写一个算法，找出A中的所有鞍点。

基本思想：在矩阵A中求出每一行的最小值元素，然后判断该元素是否是它所在列中的最大值，是则打印出，接着处理下一行。矩阵A用一个二维数组表示。

算法如下：

```
/*算法描述5.1  矩阵的鞍点*/
void  saddle (int A[][],int m, int n)
  /*m,n是矩阵A的行和列*/
{ int i,j,min;
  for (i=0;i<m;i++)     /*按行处理*/
    { min=A[i][0]
```

```
for (j=1; j<n; j++)
        if (A[i][j]<min ) min=A[i][j];    /*找第i行最小值*/
    for (j=0; j<n; j++)    /*检测该行中的每一个最小值是否是鞍点*/
        if (A[i][j]==min )
          { k=j;  p=0;
            while (p<m && A[p][j]<min)
                p++;
            if ( p>=m) printf ( " %d,%d,%d\n " , i ,k,min);
          }
    }
}
```

5.3 特殊矩阵的压缩存储

矩阵是科学计算、工程数学中大量研究的对象。对于一个矩阵结构显然用一个二维数组来表示是非常恰当的。但在有些情况下，例如有些高阶矩阵中，阶次很高，数据量很大，而非零元素非常少(远小于m×n)，若仍采用二维数组存放，就会有很多存储单元都存放的是0，这不仅造成存储空间的浪费，而且给运算带来了很大的不便，这时，就很希望能够只存储部分有效元素。另外，还有一些矩阵的数据元素分布有一定规律，如三角矩阵、对称矩阵、带状矩阵等，那么就可以利用这些规律，只存储部分元素，从而提高存储空间的利用率。这些问题都需要对矩阵进行压缩存储，一般的压缩原则是：对有规律的元素和值相同的元素只分配一个存储空间，对零元素不分配空间。

5.3.1 三角矩阵

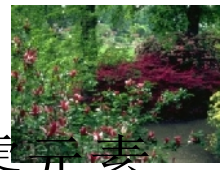
三角矩阵可以分为三类：下三角矩阵、上三角矩阵、对称矩阵。

对于一个n阶矩阵A来说，若当i<j时，有a_{ij}＝0或a_{ij}＝c(c为常数)，则称此矩阵为下三角矩阵；若当i>j时，有a_{ij}＝0或a_{ij}＝c(c为常数)，则称此矩阵为上三角矩阵；若矩阵中的所有元素均满足a_{ij}＝a_{ji}，则称此矩阵为对称矩阵。

例如，图5.4就是一个n×n的下三角矩阵，我们以此为例来讨论三角矩阵的压缩存储。

$$A \begin{bmatrix} a_{00} & & & & \\ a_{10} & a_{11} & & 0 & \\ a_{20} & a_{21} & a_{22} & & \\ \vdots & \vdots & \vdots & \ddots & \\ a_{n1,0} & a_{n1,1} & a_{n1,2} & \cdots & a_{n1,n1} \end{bmatrix}$$

图5.4 n×n的下三角矩阵A



对于下三角矩阵，只需存储下三角的非零元素，对于零元素则不存储。若以行序为主序进行存储，得到的序列是：

a_{00}，a_{10}，a_{11}，a_{20}，…，$a_{n-1,0}$，$a_{n-1,1}$，…，$a_{n-1,n-1}$

由于下三角矩阵的元素个数为n(n+1)/2，即

第0行：1个

第1行：2个

第2行：3个

……

第n－1行：n个

共有1+2+3+…+n=n(n+1)个，所以可将三角矩阵压缩到一个大小为n(n+1)/2的一维数组C中，如图5.5所示。

Loc[i,j]	0	1	2	3	4	5	…	n(n+1)/2－1
数组C	a_{00}	a_{10}	a_{11}	a_{20}	a_{21}	a_{22}	…	$a_{n-1,n-1}$

图5.5 三角矩阵的压缩存储

假设每个数据元素占size个存储单元，下三角矩阵中任意元素a_{ij}在一维数组C中的存储位置可通过下式计算：

$$Loc[i，j]=Loc[0，0]+\left(\frac{i(i+1)}{2}+j\right)\times size \quad (i\geq j)$$

也就是说，如果C中数据元素C[k]的下标为k，则k与下三角矩阵中数据元素a_{ij}的下标i、j之间的关系为：

$$k=\frac{i(i+1)}{2}+j$$

若上三角部分为一常数c，则数组的大小可设为C[n(n+1)/2+1]，其中C[n(n+1)/2]存放常数c。

例如，一个5×5的下三角矩阵A，我们可以用一维数组C[15]对其进行压缩存储，如图5.6所示。

$$A=\begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ 6 & 2 & 0 & 0 & 0 \\ 4 & 8 & 1 & 0 & 0 \\ 7 & 4 & 6 & 0 & 0 \\ 8 & 2 & 9 & 5 & 7 \end{bmatrix}$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	6	2	4	8	1	7	4	6	0	8	2	9	5	7

图5.6 5×5的下三角矩阵的压缩存储

同理，对上三角矩阵，只需存储上三角的非零元素，对于零元素则不存储，同样可以将其压缩存储到一个大小为n(n+1)/2的一维数组中。若以行序为主序进行存储，得到的序列是：

$$a_{00}，a_{01}，a_{02}，\ldots，a_{0,n-1}，a_{1,n-1}，\ldots，a_{n-1,n-1}$$

其中元素$a_{ij}(i\le j)$在数组C中的存储位置为：

$$k=\frac{i(2n-i+1)}{2}+j-i \quad (i\le j)$$

对于对称矩阵，则可以只存储上三角部分，也可只存储下三角部分，将其压缩存储到一个大小为n(n+1)/2的一维数组中。若只存储下三角部分，那么元素a_{ij}在数组C中的存储位置为：

$$k=\frac{i(i+1)}{2}+j \qquad (i\geq j)$$

$$k=\frac{j(j+1)}{2}+i \qquad (i\leq j)$$

5.3.2 稀疏矩阵

设m×n矩阵中有t个非零元素，且t<<m×n，这样的矩阵称为稀疏矩阵。很多科学管理及工程计算中，常会遇到阶数很高的大型稀疏矩阵。如果按常规分配方法顺序分配在计算机内，那将是相当浪费内存的，并且也给计算带来不便。如图5.7所示的矩阵M中，非零元素个数为8个，矩阵元素为42个，它显然是一个稀疏矩阵。

$$M_{6\times7}=\begin{bmatrix}0&12&0&0&0&0&0\\0&0&0&0&0&0&0\\3&0&0&0&0&14&0\\0&0&20&0&0&0&0\\0&18&0&0&0&0&0\\15&0&0&0&0&0&-7\end{bmatrix}$$

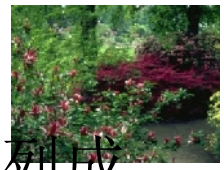
图5.7 稀疏矩阵

1. 稀疏矩阵的三元组表存储

对于稀疏矩阵，我们设想另外一种存储方法，即仅仅存放非零元素。但这类矩阵中零元素的分布通常没有规律，为了能找到相应的元素，所以仅存储非零元素的值是不够的，还要记下它所在的行和列。于是采取如下方法：将非零元素所在的行、列以及它的值构成一个三元组(row，col，value)，然后再按某种规律存储这些三元组，这就是稀疏矩阵的三元组表示法。每个非零元素在一维数组中的表示形式如图5.8所示。

行值	列值	元素值
row	col	value

图5.8 三元组的结构



将三元组按行优先的顺序，同一行中列号从小到大排列成的线性表称为三元组表。三元组表应采用顺序存储方法存储，如图5.7所示稀疏矩阵对应的三元组表为图5.9。

	row	col	value
1	1	2	12
2	3	1	3
3	3	6	14
4	4	3	20
5	5	2	18
6	6	1	15
7	6	7	−7

图5.9 稀疏矩阵的三元组表表示

显然，要惟一地表示一个稀疏矩阵，在存储三元组表的同时还需要存储该矩阵的总行数、总列数，为了运算方便，矩阵的非零元素的个数也同时存储。三元组表的类型说明如下：

```
define MAX  1024                    /*一个足够大的数*/
typedef  struct
  {  int row,col;                   /*非零元素的行、列*/
     datatype  v;                   /*非零元素值*/
  }Triple;                          /*三元组类型*/
typedef  struct
 {  int mu,nu,tu;                   /*矩阵的行、列及非零元素的个数*/
    Triple  data[MAX+1];            /*三元组表，data[0]未用*/
    } TSMatrix;                     /*三元组表的存储类型*/
```

2．用三元组表实现稀疏矩阵的转置

矩阵的三元组存储方法确实节约了存储空间，但矩阵的运算从算法上可能变得复杂些。下面以稀疏矩阵的转置运算为例介绍其实现方法。

所谓矩阵转置，是指把位于(row，col)位置上的元素转换到(col，row)位置上，也就是说，把元素的行列互换。如图5.7所示的6×7矩阵M，它的转置矩阵就是7×6的矩阵N(如图5.10所示)，M[row，col]＝N[col，row]。

$$N_{7\times6}=\begin{bmatrix}0&0&3&0&0&15\\12&0&0&0&18&0\\0&0&0&20&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&14&0&0&0\\0&0&0&0&0&-7\end{bmatrix}$$

图5.10 矩阵M的转置矩阵N

显然，稀疏矩阵的转置矩阵仍为稀疏矩阵，所以可以采用三元组表实现矩阵的转置。

设TSMatrix　A表示一个m×n的稀疏矩阵，其转置B则是一个n×m的稀疏矩阵，因此也有 TSMatrix　B。由A求B需要将A的行、列转化成B的列、行，将A.data中每一个三元组的行列交换后存放到B.data中，并且保证B.data中的数据也是以行序为主序进行存放(B.data中的行号即为A.data中的列号)。

要保证B.data中的数据也是以行序为主序进行存放，可以有两种方法，下面分别介绍。

方法一：

这种方法的思路是按照A.data中的列序进行转置，并依次送入B.data中，即在A.data中依次找第一列的、第二列的……直到最后一列的非零元素，并将找到的每个三元组的行、列交换后顺序存储到B.data中即可，转置过程如图5.11所示。

	row	col	value
1	1	2	12
2	3	1	3
3	3	6	14
4	4	3	20
5	5	2	18
6	6	1	15
7	6	7	7

	row	col	value
1	1	3	3
2	1	6	15
3	2	1	12
4	2	5	18
5	3	4	20
6	6	3	14
7	7	6	7

图5.11 矩阵的转置

附设一个位置计数器j，用于指向当前转置后元素应放入三元组表B.data中的位置。处理完一个元素后，j加1，j的初值为1。具体转置算法如下：

```
/*算法描述5.2    稀疏矩阵转置*/
void TransTSMatrix1(TSMatrix  A，TSMatrix  *B)
/*为了能使调用函数得到转置结果，将B设为指针类型*/
{ int i,j,k;
  B->mu=A.nu;  B->nu=A.mu;  B->tu=A.tu;
/*稀疏矩阵的行、列、元素个数*/
if (B->tu>0)                                  /*有非零元素则转换*/
```

```
{  j=1;
    for (k=1; col<=A.nu; k++)                    /*按A的列序转换*/
      for (i=1; i<= (A.tu); i++)                 /*扫描整个三元组表*/
        if  (A.data[i].col==k )
          { B->data[j].row= A.data[i].col ;
    B->data[j].col= A.data[i].row ;
    B->data[j].v= A.data[i].v;
    j++;
   }
}
}
```

分析该算法，其时间主要耗费在col和p的二重循环上，所以时间复杂性为O(n×t)，

(设m、n是原矩阵的行、列，t是稀疏矩阵的非零元素个数)，显然当非零元素的个数t和m×n同数量级时，算法的时间复杂度为$O(m\times n^2)$，和通常存储方式下矩阵转置算法相比，可能节约了一定量的存储空间，但算法的时间性能更差一些。

方法二：

算法5.2的效率低的原因是算法从A的三元组表中寻找第一列的、第二列的……直至最后一列的非零元素时，要反复搜索A表多次，若能直接确定A中每一三元组在B中的位置，则对A的三元组表扫描一次即可。这是可以做到的，因为A中第一列的第一个非零元素一定存储在B.data[1]，如果还知道第一列的非零元素的个数，那么第二列的第一个非零元素在B.data中的位置便等于第一列的第一个非零元素在B.data中的位置加上第一列的非零元素的个数，依次类推，因为A中三元组的存放顺序是先行后列，对同一行来说，必定先遇到列号小的元素，这样只需扫描一遍A.data即可。

根据这个思路，需引入两个辅助数组num[n+1]和pos[n+1]来实现，num[col]表示矩阵A中第col列的非零元素的个数(为了方便均从1单元用起)，pos[col]初始值表示矩阵A中的第col列的第一个非零元素在B.data中的位置。于是pos的初始值为：

pos[1]=1；

pos[col]=pos[col－1]+num[col－1]；　2≤col≤n

例如，对于图5.11所示的矩阵A，num 和pos的值如图5.12所示。

col	1	2	3	4	5	6	7
num[col]	2	2	1	0	0	1	1
pos[col]	1	3	5	6	6	6	7

图5.12 矩阵A的num与pos值

依次扫描A.data，当扫描到一个col列元素时，直接将其存放在B.data的pos[col]位置上，pos[col]加 1 ，pos[col]中始终是下一个col列元素在B.data中的位置。

下面按以上思路改进转置算法。

```
/*算法描述5.3　稀疏矩阵的快速转置*/

void TransTSMatrix2(TSMatrix  A，TSMatrix  *B)

{ int  i,j,k,col;

int num[n+1],pos[n+1];

    B->mu=A.nu;  B->nu=A.mu;  B->tu=A.tu;
```

```
/*稀疏矩阵的行、列、元素个数*/
if (B->tu)                          /*有非零元素则转换*/
{ for (col=1; col<=A.nu; col++)  num[j]=0;
 for (k=1; k<=A.tu; k++)        /*求矩阵A中每一列非零元素的个数*/
    num[A.data[i].col ]++;
 pos[1]=1;                  /*求矩阵A中每一列第一个非零元
                             素在B.data中的位置*/
 for (col=2; col<=A.nu; col++)
        pos[col]= pos[col-1]+num[col-1];
      for (i=1; i<= A.tu; i++)                  /*扫描三元组表*/
```

```
{ col=A.data[i].col;              /*当前三元组的列号*/
  j=pos[col];                     /*当前三元组在B.data中的位置*/
  B->data[j].col= A.data[i].row ;
  B->data[j].row= A.data[i].col;
  B->data[j].v= A.data[i].v;
   pos[col]++;
  }
    }
}
```

分析这个算法的时间复杂度：这个算法中有四个循环，分别执行n，t，n－1，t次，在每个循环中，每次迭代的时间是一常量，因此总的计算量是O(n+t)。当然它所需要的存储空间比前一个算法多了两个辅助数组。

3. 稀疏矩阵的十字链表存储

三元组表可以看作稀疏矩阵顺序存储，但是在做一些操作(如加法、乘法)时，非零项数目及非零元素的位置会发生变化，这时这种表示就十分不便。下面我们介绍稀疏矩阵的一种链式存储结构——十字链表，它同样具备链式存储的特点，因此，在某些情况下，采用十字链表表示稀疏矩阵是很方便的。

用十字链表表示稀疏矩阵时每个非零元素存储为一个结点，结点由5个域组成，其结构如图5.13表示。

(1) row：存储非零元素的行号；

(2) col：存储非零元素的列号；

(3) value：存储本元素的值；

(4) right：链接同一行的下一个非零元素；

(5) down：链接同一列的下一个非零元素。

row	col	value
down		right

图5.13 十字链表中的结点结构

稀疏矩阵中每一行的非零元素结点按其列号从小到大的顺序由right域链成一个带表头结点的单向行链表，同样每一列中的非零元素按其行号从小到大的顺序由down域也链成一个带表头结点的单向列链表，即每个非零元素a_{ij}既是第i行链表中的一个结点，又是第j列链表中的一个结点。这好像是处在一个十字交叉路上，所以称其为十字链表。同时再附设一个存放所有行链表的头指针的一维数组和一个存放所有列链表的头指针的一维数组。如图5.14是一个稀疏矩阵和其对应的十字链表。

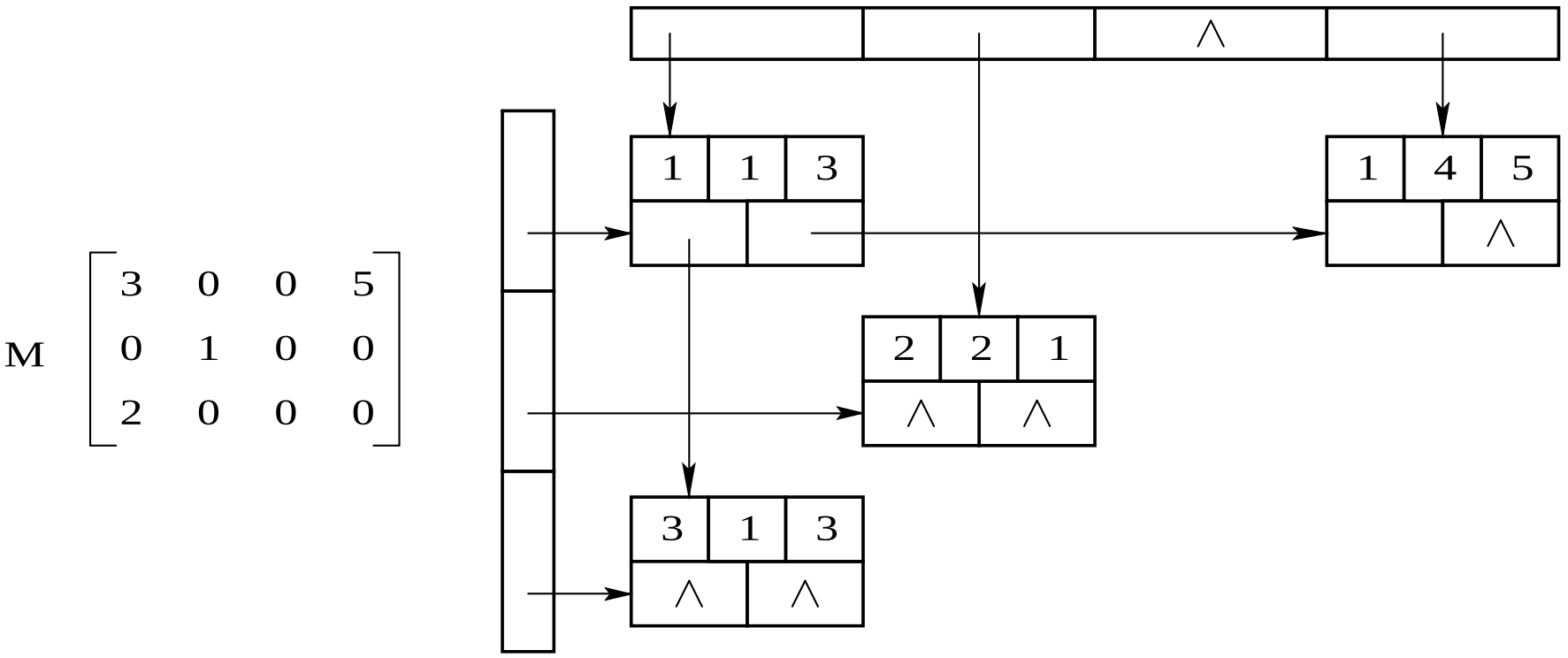


图5.14 稀疏矩阵和对应的十字链表

十字链表的结构类型说明如下：

```
typedef  struct  OLNode
{ int  row, col;                          /*非零元素的行和列下标*/
   datatype value;
   struct OLNode  *down , *right;    /*非零元素所在的行表、列表的后继链域*/
 }OLNode; *Olink;
typedef struct
 { Olink *row_head; *col_head;    /*行、列链表的头指针*/
   int  mu,nu,tu;                          /*稀疏矩阵的行数、列数和非零元素的个数*/
       }CrossLink;
```

5.4 实习：数组应用实例

我们可以用一个二维数组maze[m+2][n+2]表示迷宫，其中元素0表示走得通，1表示走不通。为了叙述上的方便，设定由maze [1][1]进入迷宫，由maze [m][n]走出迷宫。迷宫中任意一点的位置可由maze [row][col]来表示。在MAZE[row][col]的周围有八个方向可走，为了避免检测边界状态，二维数组maze [m+2][n+2]的零行、零列及m＋1行、n＋1列的值均为1。另外，为了简化计算，设一个二维数组move[8][2]记录可走的八个方向坐标增量。move [k][0]表示第k个方向上row的增量，move [k][1]表示第k个方向上col的增量。例如，从maze[row][col]出发，沿东南的方向达到下一个位置maze[i][j]为：

$$i = row + move[1][0] = row + 1$$

$$j = row + move[1][1] = col + 1$$

为了避免走重复路径，我们用mark[m+2][n+2]来记走过的路径。它的初值为0，一旦达到某个位置maze[row][col]，就把mark[row][col]置为1。

我们规定计算机走迷宫时，每次都从正东的方向起顺时针检测，当检测到某个方向下一个位置的值为0，并且没有走过这一位置时，就沿此方向走一步。这样依次重复检测，当某一步七个可前进方向都为1时，则退回一步，重新检测下一个方向。因此，我们必须让计算机记下所走过的路径和方向，以便退到刚走过的方位，并继续检测下一个方位。我们可以设置一个栈stack来记每一步的坐标row、col。

下面是一个7×10的迷宫实例。

```
#include <stdio.h>
main()
{ int maze[9][12] = {
                    1,  1,1,1,1,1,1,1,1,1,  1,
                    1,  0,1,0,0,0,1,1,0,1,1,  1,
                    1,  1,0,1,1,0,0,0,1,1,0,  1,
                    1,  1,0,1,1,0,1,1,0,0,1,  1,
                    1,  1,1,1,1,0,0,1,1,1,1,  1,
                    1,  0,0,1,1,1,1,0,1,1,0,  1,
                    1,  0,1,1,0,0,0,1,1,0,1,  1,
                    1,  1,0,1,1,0,1,0,0,1,0,  1,
                    1,  1,1,1,1,1,1,1,1,1,1,  1 };
```

```
int move[8][2] = { 0,1, 1,1, 1,0, 1,－1, 0,－1,－1,－1,－1,0,－1,1 };
  /*move为下一步行走方向，值为坐标增量*/
 int stack[64][2], mark[9][12] = { 0 };
  /*stack为栈，记录路径；mark为标记，记录已走过的点*/
 int top=1, row=1, col=1, k=0;  /*k标记方向，从正东开始*/
 int i,j;
 mark[row][col]=1;                                    /*第一步*/
stack[top][0]=row;  stack[top][1]=col;
 while (!(row==7&&col==10)&&!top==0)
```

```
i=row+move[k][0]; j=col+move[k][1];    /*下一步*/
  if (maze[i][j]==0&&mark[i][j]==0)
   { top=top+1; k=0; row=i; col=j;              /*可行走*/
  stack[top][0]=row; stack[top][1]=col; mark[row][col]=1; }
  else
   { k++;                                       /*不可行走, 看下一个方向*/
 if (k==8)
    { k=0; top--; row=stack[top][0];
     col=stack[top][1];
  }                                             /*八个方向全部不通*/
    }
  }
```

```
if (top==0)
    printf("NO PATH!!!\n");
  else
   { printf("ALL  PATH : ");
   for(i=1;i<top+1;i++) printf("%d, %d\n",stack[i][0],stack[i][1]);
   for(i=1;i<top+1;i++) maze[stack[i][0]][stack[i][1]]=2;                    /*
有通路标记为2*/
   for(i=1;i<8;i++)
          { for(j=1;j<11;j++)
         printf("%d ",maze[i][j]);
         printf("\n");
        }
    }
  }
```

习 题 5

1. 设有7行8列的二维数组A，每个数据元素占4个存储单元，已知A的基地址为1000，计算：

(1) 数组A共占用多少存储单元；

(2) 数组A的最后一个元素的地址；

(3) 按行存储时元素a_{46}的地址；

(4) 按列存储时元素a_{46}的地址。

2. 对上三角矩阵，将其压缩存储到一个一维数组C中。若以行序为主序进行存储，试求元素$a_{ij}(i \leq j)$在数组C中的存储位置。

3. 特殊矩阵和稀疏矩阵哪一种压缩存储后会失去随机存取的功能?为什么?

4. 当稀疏矩阵A和B均以三元组表作为存储结构时，试写出矩阵相加的算法，其结果存放在三元组表C中

5. 试给出图5.15所示矩阵的三元组，并根据所得三元组分别按照算法5.2和算法5.3求其转置矩阵的三元组。

$$A=\begin{bmatrix}0 & 3 & 0 & 0 & 0\\ 6 & 0 & 0 & 5 & 0\\ 0 & 2 & 0 & 0 & 1\\ 0 & 0 & 8 & 0 & 0\\ 0 & 0 & 0 & 7 & 0\end{bmatrix}$$

图5.15 第5、6题图

6. 给出图5.15所示矩阵的十字链表。

7. 用数组结构存储一元多项式，编写算法求解两个一元多项式相加结果。

第6章 树

6.1 树的应用实例

例6.1 如图6.1为某校计算机系领导结构图。

例6.2 如图6.2是Windows 2000系统的注册表，第一层是“我的电脑”，第二层分别是HKEY_CLASSES_ROOT、HKEY_CURRENT_USER、HKEY_LOCAL_MACHINE、HKEY_USERS和HKEY_CURRENT_CONFIG，第三层是“HKEY_LOCAL_MACHINE”下的HARDWARE、SAM、SECURITY、SOFTWARE及SYSTEM。

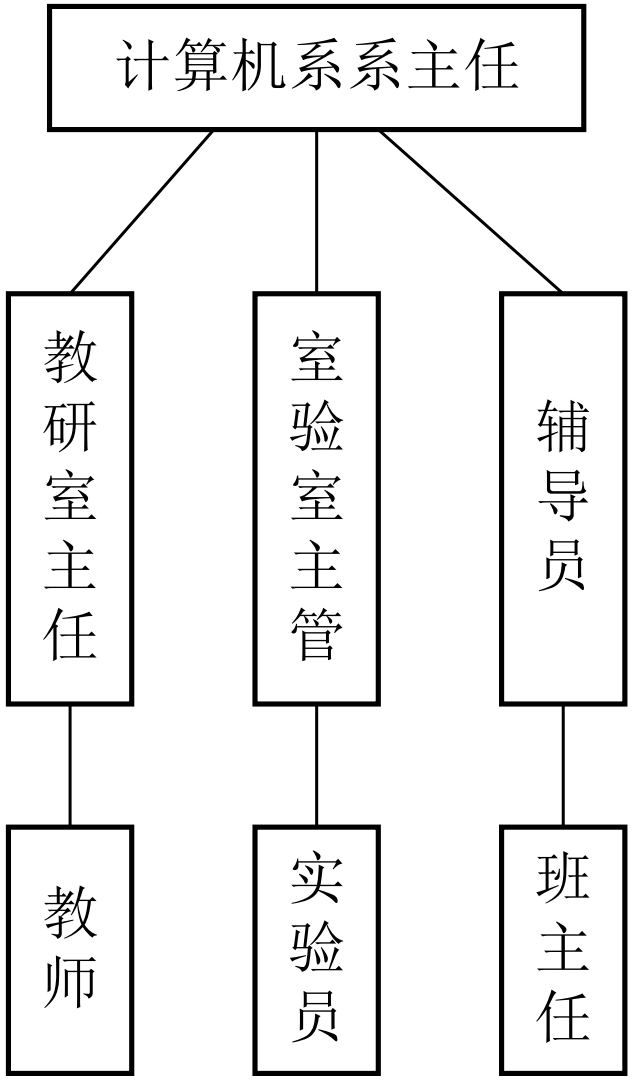


图6.1 某校计算机系领导结构图

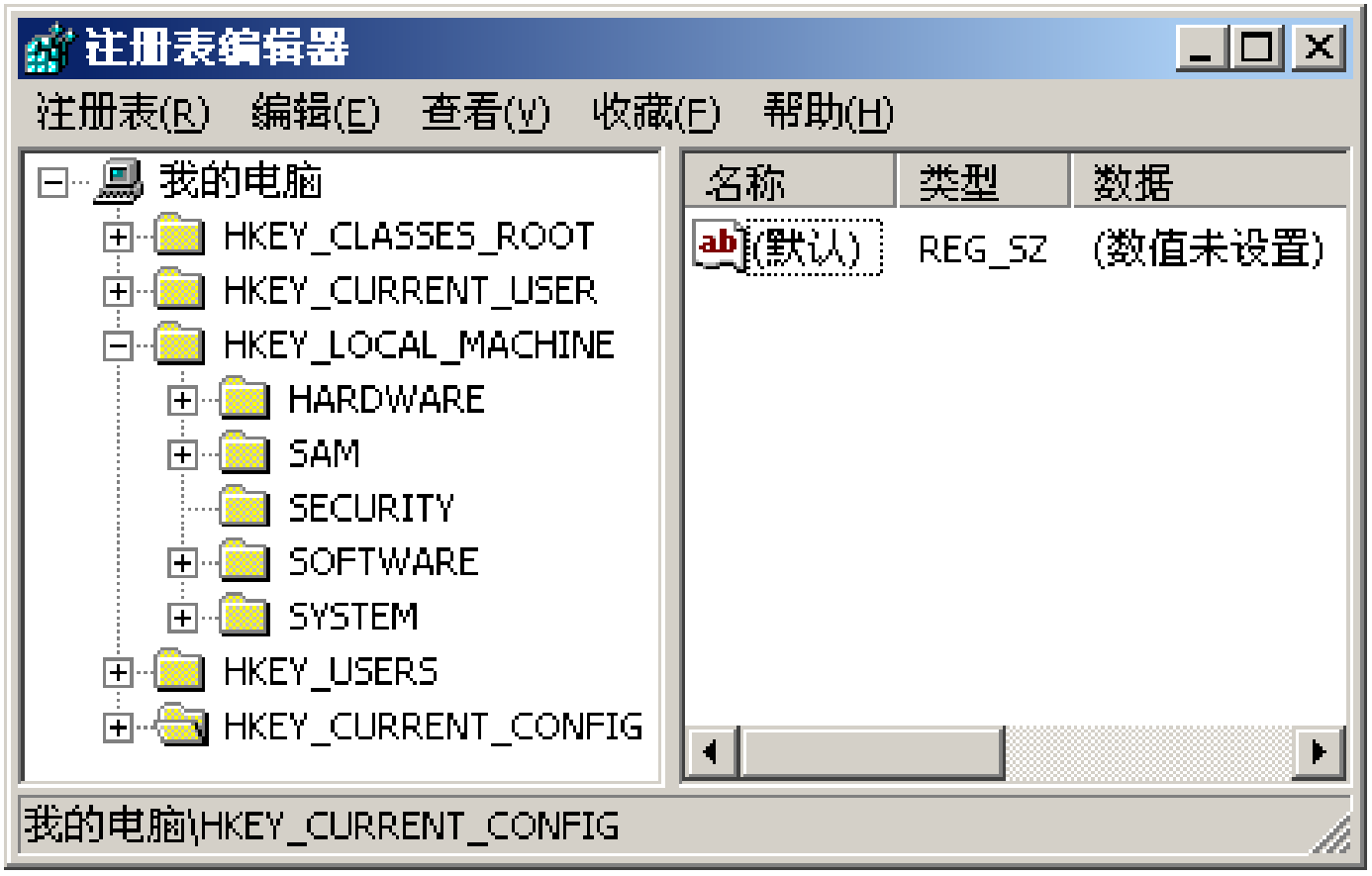


图6.2 Windows 2000系统的注册表

6.2 树的基本概念和术语

6.2.1 树的定义

树(Tree)是由一个或多个结点组成的有限集合T。其中：

(1) 有一个特定的结点称为该树的根(Root)结点；

(2) 除根结点之外的其余结点可分为m(m≥0)个互不相交的有限集合T_1，T_2，…，T_m，且其中每一个集合本身又是一棵树，称之为根的子树(Subtree)。

这是一个递归的定义，即在定义中又用到了树这个术语。它道出了树的固有特性。仅有一个根结点的树是最小树，树中结点较多时，每个结点都是某一棵子树的根。

图6.3是一棵由9个结点组成的树T。其中A是根结点，其余结点分为三个互不相交的子集：T_1={B,H,I}，T_2={C}，T_3={D,E,F,G}。T_1、T_2、T_3都是树根A的子树，这三棵子树的根结点分别是B、C、D。每棵子树本身也是一棵树，可继续划分。例如子树T_3以D为根结点，它的其余结点又可分为两个互不相交的子集：T_{31}={E}，T_{32}={F，G}，而其中T_{31}可以认为是仅有一个根结点的子树。

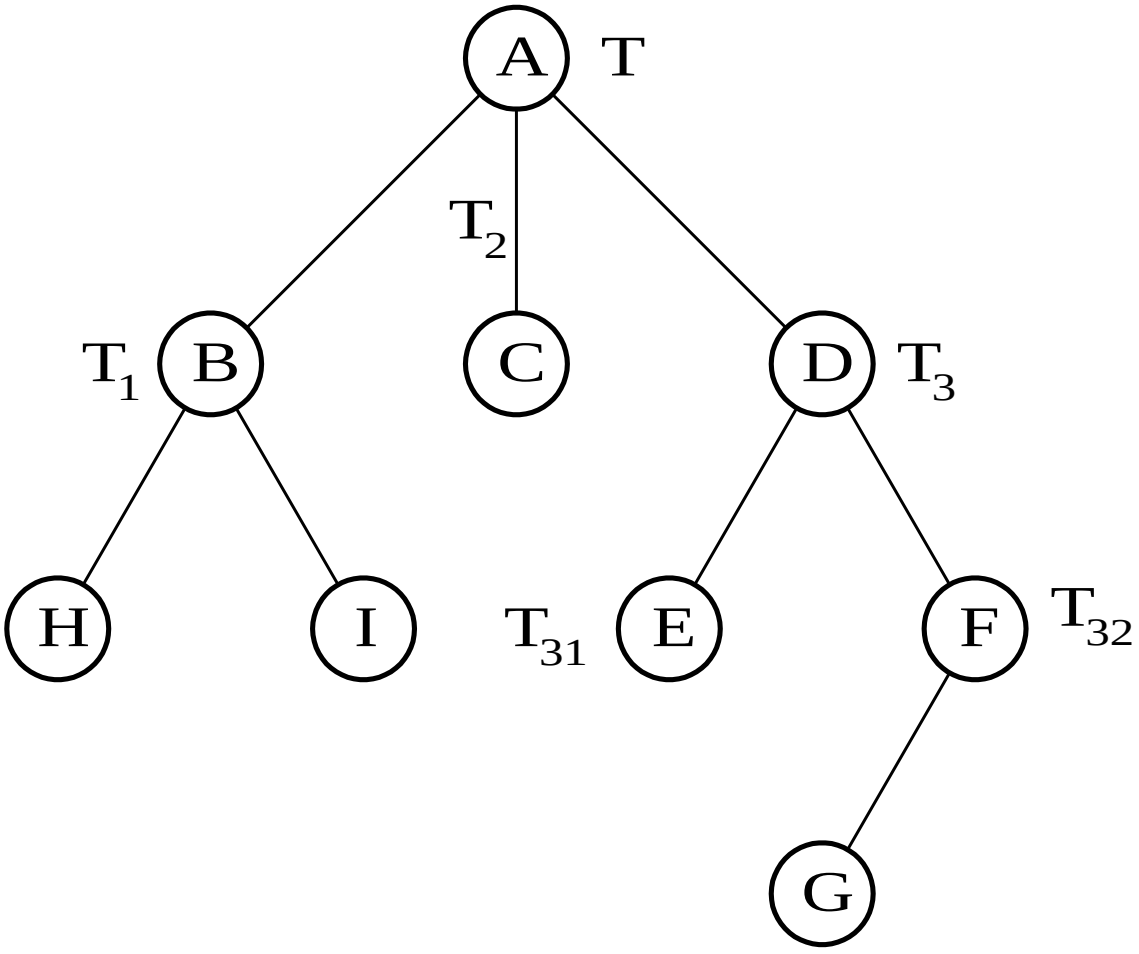


图6.3 树T

6.2.2 树的常用术语

1. 结点的度和树的度

树的结点包含一个数据元素及若干指向其子树的分支。结点拥有的子树数称为结点的度(Degree)。而树的度是指树中结点的度的最大值。如图6.3中，B结点有2个子树，则它的度为2。在树T中，A结点的度最大，值为3，也就是说，树T的度为3。

2. 分支结点和叶子结点

称度不为0的结点为分支结点，也叫非终端结点。称度为0的结点为叶子(Leaf)或终端结点。如图6.3中，分支结点分别为A、B、D、F，而叶子结点分别为H、I、C、E、G。

3. 孩子、双亲、兄弟、子孙、祖先

结点的子树的根称为该结点的孩子(Child)，相应地，该结点称为孩子的双亲(Parent)。同一个双亲的孩子之间互称兄弟(Sibling)。如图6.3中，B、C、D分别是根结点A的子树的根，三个都是A的孩子，相应地，A是它们的双亲，其中，B、C、D三者是兄弟。一棵树上除根结点以外的其它结点称为根结点的子孙。对于树中某结点，从根结点开始到该结点的双亲是该结点的祖先。

4. 结点的层次和树的高度

结点的层次(Level)从根结点开始定义，根为第一层，根结点的孩子为第二层，依次类推，其余结点的层次值为双亲结点层次值加1。树中结点的最大层次值称为树的高度或深度(Depth)。如图6.3所示的树T高度为4。

5. 无序树、有序树、森林

若树中结点的各子树看成从左至右是有次序的(即不能互换)，则称该树为有序树，否则称为无序树。在有序树中最左边的子树的根称为第一个孩子，最右边的称为最后一个孩子。

森林(Forest)是m(m≥0)棵互不相交的树的集合。对树中每个结点而言，其子树的集合即为森林。

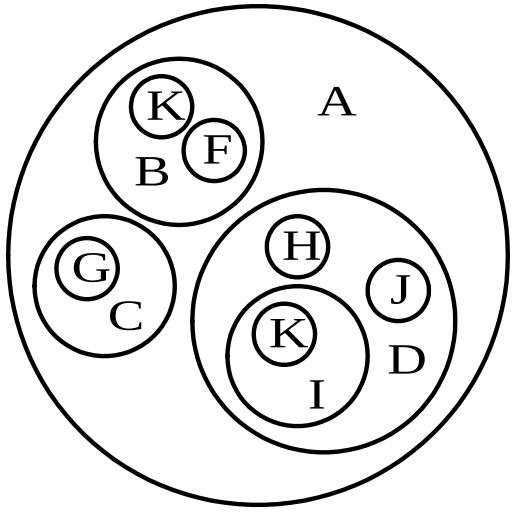
6.2.3 树的表示方法

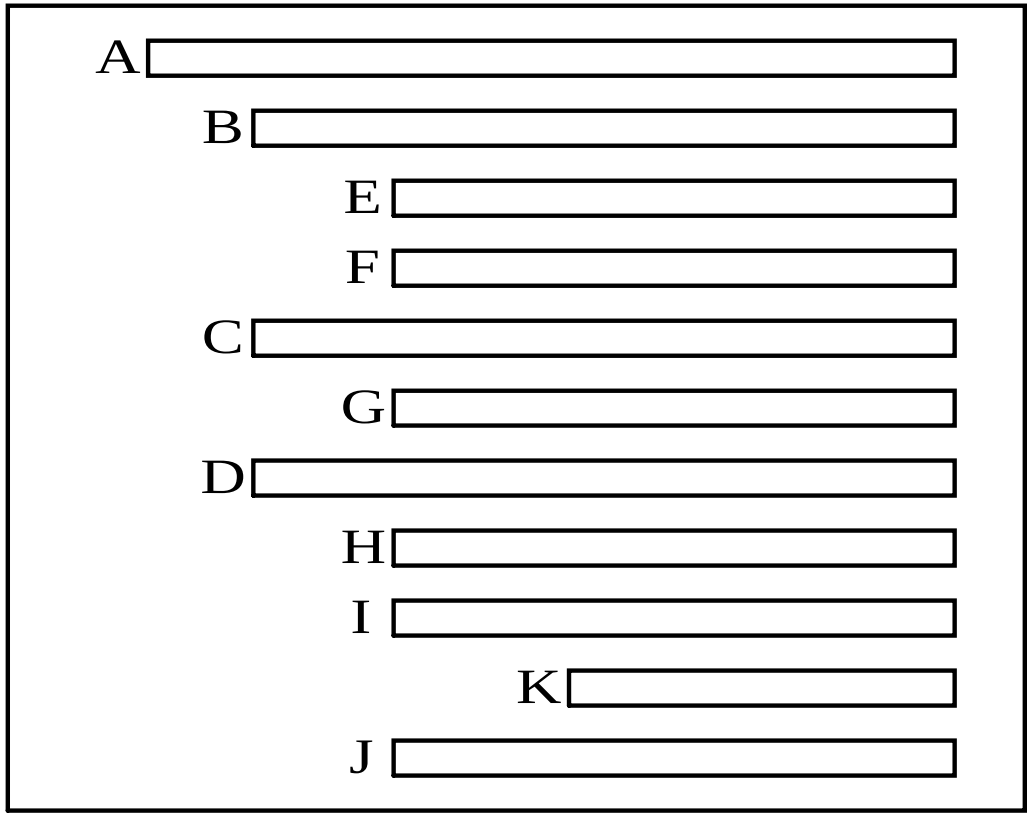
树的表示形式有多种，常见的三种方法是：

(1) 倒悬树法，如图6.3所示。

(2) 嵌套集合表示法，如图6.4(a)所示。

(3) 凹入表示法，如图6.4(b)所示。





(a) (b)

图6.4 树结构的不同表示方法

(a) 嵌套集合表示法；(b) 凹入表示法

6.3 二 叉 树

6.3.1 二叉树的定义

二叉树(Binary Tree)是n(n≥0)个结点的有限集合。它或为空树(n＝0)，或为非空树；对于非空树有：

(1) 有一个特定的称之为根的结点；

(2) 根结点以外的其余结点分别由两棵互不相交的称之为左子树和右子树的二叉树组成。

这个递归定义表明二叉树或为空，或是由一个根结点加上两棵分别称为左子树和右子树的互不相交的二叉树组成的。由于左、右子树也是二叉树，则由二叉树的定义，它们也可以为空。由此，二叉树可以有五种基本形态，如图6.5所示。

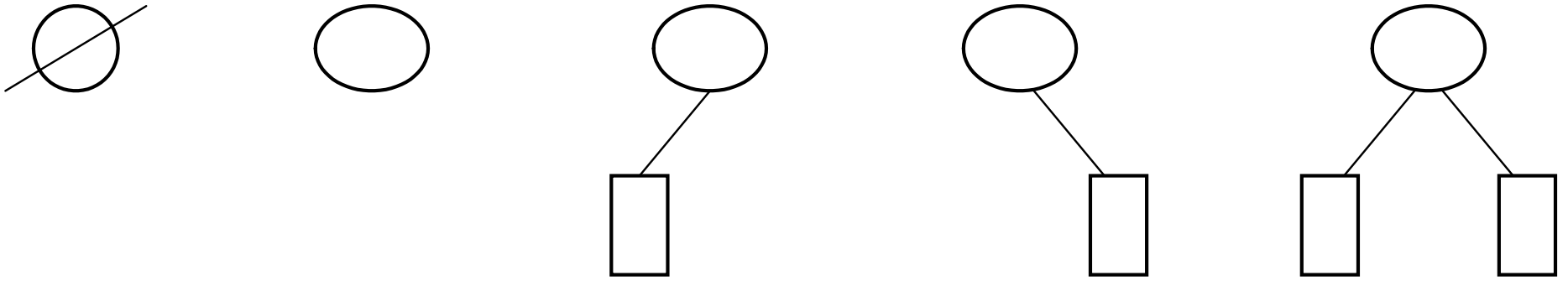


图6.5 二叉树的五种基本形态

(a) 空二叉树; (b) 只有一个根结点; (c) 有根结点和左子树;
(d) 有根结点和右子树; (e) 有根结点和左、右子树

从以上分析得知二叉树与普通树比较，有以下特点：

(1) 二叉树可以为空树。

(2) 二叉树的度不大于2(即每个结点至多只有两棵子树)。

(3) 二叉树是有序树，其左子树和右子树是严格区分且不能随意颠倒的。如图6.5(c)和图6.5(d)就是二棵不同的二叉树。

6.3.2 二叉树的重要性质

性质1 二叉树第i(i≥1)层上至多有2^{i-1}个结点。

根据二叉树和结点层次的定义可知，根结点在第一层上，这层结点数至多为1个，即2^0个；显然第二层上至多有2个结点，即2^1个…… 假设第i－1层的结点至多有2^{i-2}个，且每个结点最多有两个孩子，那么第i层上结点至多有$2\times 2^{i-2}=2^{i-1}$个。

性质2 深度为k(k≥1)的二叉树至多有2^k-1个结点。

由性质1可知，各层结点最多数目之和为$2^0+2^1+2^2+\ldots+2^{k-1}$；由二进制换算关系可知：$2^0+2^1+2^2+\ldots+2^{k-1}=2^k-1$，因此二叉树中结点的最大数目为$2^k-1$。

性质3 在任意二叉树中，若叶子结点(即度为零的结点)个数为n_0，度为1的结点个数为n_1，度为2的结点个数为n_2，那么$n_0=n_2+1$。

证明： 设n代表二叉树结点总数，那么

$$n=n_0+n_1+n_2 \qquad (1)$$

由于有n个结点的二叉树总分支数为n－1条，于是得

$$n-1=0\times n_0+1\times n_1+2\times n_2 \qquad (2)$$

将式(1)代入式(2)得

$$n_0=n_2+1$$

有两种特殊形态的二叉树，它们是满二叉树和完全二叉树。

满二叉树：深度为k且含有2^k-1个结点的二叉树为满二叉树，这种树的特点是每层上的结点数都是最大结点数，如图6.6(a)所示。对满二叉树的结点可以从根结点开始自上向下，自左至右顺序编号，图6.6(a)中每个结点边的数字即是该结点的编号。

完全二叉树：深度为k，含有n个结点的二叉树，当且仅当每个结点的编号与相应满二叉树结点顺序号从1到n相对应时，则称此二叉树为完全二叉树，如图6.6(b)所示。而图6.6(c)则不是完全二叉树。

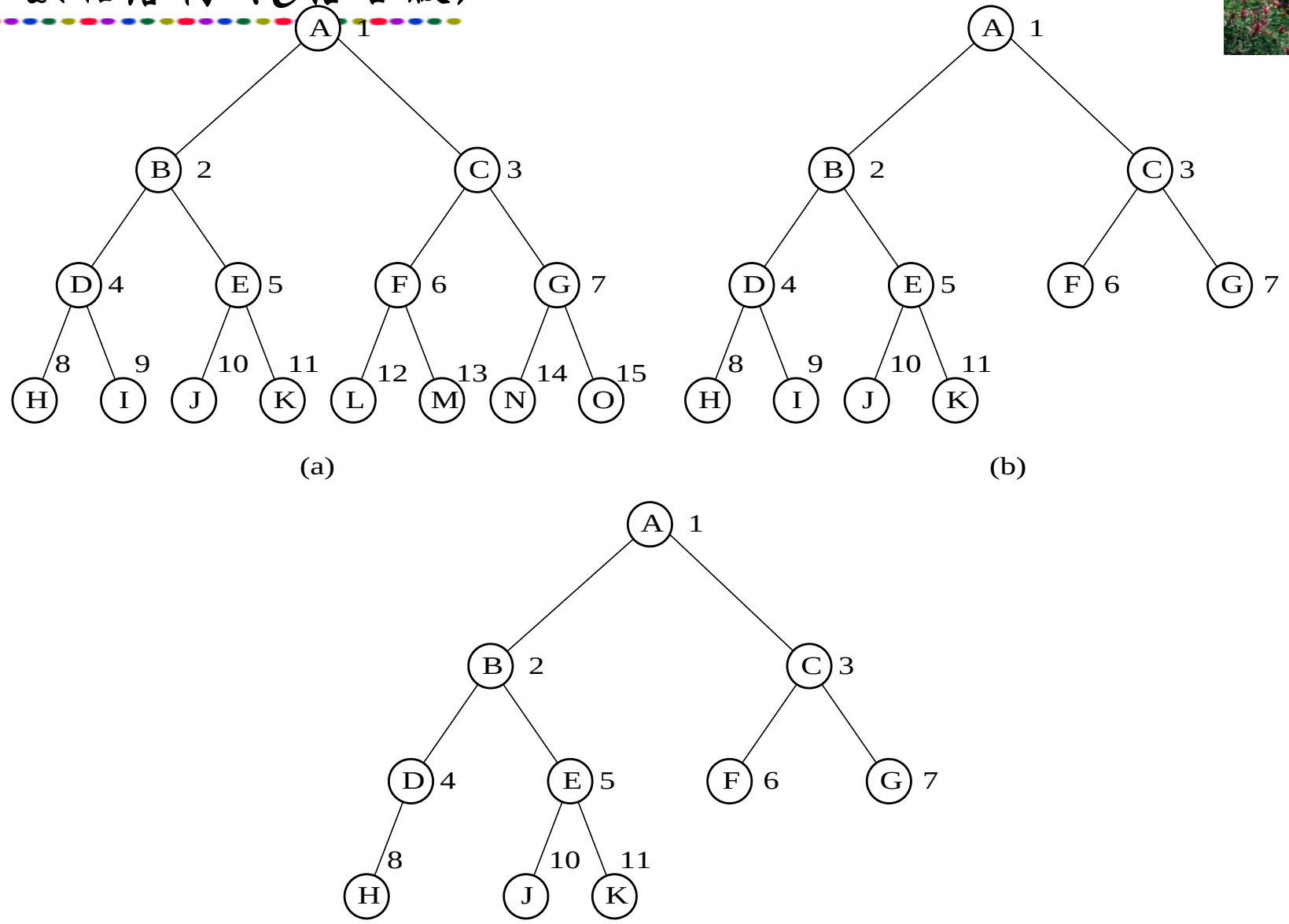


图6.6 满二叉树和完全二叉树

(a) 满二叉树；(b) 完全二叉树；(c) 非完全二叉树

性质4 具有n个结点的完全二叉树深度为⌊lbn⌋+1(其中⌊x⌋表示不大于x的最大整数)。

性质5 若对有n(1≤i≤n)个结点的完全二叉树进行顺序编号，那么，对于编号为i(i≥1)的结点：

当i=1时，该结点为根，它无双亲结点；

当i＞1时，该结点的双亲结点编号为⌊i/2⌋；

若2i≤n，则有编号为2i的左孩子，否则没有左孩子；

若2i+1≤n，则有编号为2i+1的右孩子，否则没有右孩子。

对照图6.6(a)或图6.6(b)，读者可看到由性质5所描述的结点与编号的对应关系。

6.3.3 二叉树的存储结构

二叉树常用的存储结构有两种，即顺序存储结构(向量)和链表存储结构。

(1) 顺序存储结构(向量)可以作为二叉树的存储结构。这种存储结构适用于完全二叉树和满二叉树。假设用一维数组a存放图6.6(a)所示的满二叉树。可以发现图6.6(a)中结点的编号恰好与数组元素的下标相对应，见图6.7。根据二叉树性质5，在a数组中可以方便地由某结点a[i]的下标i找到它们的双亲结点a[i/2]或左、右孩子结点a[2i]、a[2i+1]。在哈夫曼树构造算法中也用到顺序存储结构。一般二叉树较少采用顺序存储结构。

1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	C	D	E	F	G	H	J	K	L	M	N	O

图6.7 二叉树的顺序存储结构

(2) 链表存储结构通常用于二叉树存储。常见的有二叉链表和三叉链表。

二叉链表的每个结点都有一个数据域和两个指针域，一个指针指向左孩子，另一个指针指向右孩子。结点结构如图6.8(a)所示，可以描述为：

```
struct  treenode2
        {int  data;                          /*数据域*/
         struct   treenode2  *lch，*rch;    /*左、右指针域*/
        }
```

三叉链表的结点比二叉链表多了一个指向双亲的指针域。结点结构如图6.8(b)所示，可以描述为：

```
struct  treenode3
{int  data;                                  /*数据域*/
        struct  treenode3  *lch，*parent，*rch；/*parent是双亲指针域*/
   }
```

(a) (b)

图6.8 二叉树链表存储结构中的结点结构
(a) 二叉链表中的结点结构；(b) 三叉链表中的结点结构

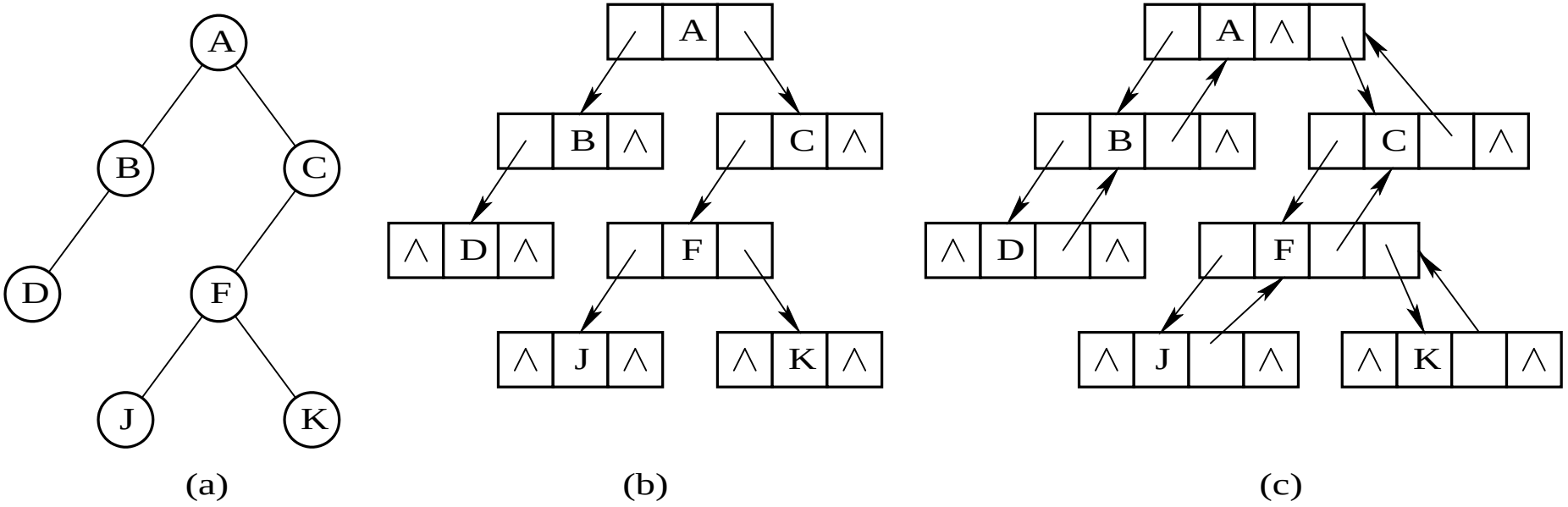


图6.9 二叉树的链表存储结构

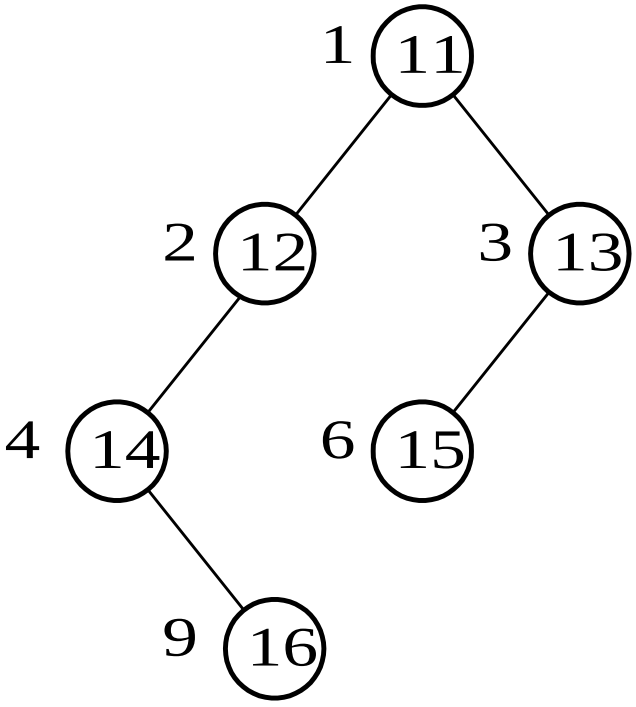
6.3.4 二叉树二叉链表的一个生成算法

算法思路分析：此方法主要利用二叉树的性质5。对任意二叉树，先按满二叉树对其进行编号，如图6.10(a)所示。由于此树并非完全二叉树，所以编号并不连续。算法中使用一个辅助向量s来存放指向树结点的指针，如s[i]中存放编号为i的结点的指针，即为该结点的地址。此例原始数据序列如图6.10(b)所示，照此一一输入即可生成二叉链表。

当结点编号i=1时，所产生的结点为根结点，同时将指向该结点的指针存入s[1]。

当　　结点编号i＞1时，产生一个新的结点之后，也要将指向该结点的指针存入s[i]。由

性质5可知：j=i/2为它的双亲结点编号。如果i为偶数，则它是双亲结点的左孩子，即让s [j]-> lch=s[i]；如果i为奇数，则它是双亲结点的右孩子，即让s[j]->rch =s[i]。这样就将新输入的结点逐一与其双亲结点相连，生成二叉树。



(a)

i	1	2	3	4	6	9
x	11	12	13	14	15	16

(b)

图6.10 二叉树及数据表

结点结构如前所描述，为struct　treenode2。辅助向量为struct treenode2 *s[20]。二叉树生成算法如下：

```
/*算法描述6.1 二叉树生成算法*/

stuct  treenode2  * creat()

 int i,x;

 struct treenode2 *q, *t;

 {printf("i,x="); scanf("%d,%d"，  &i,&x);

  while  ((i!=0)&&(x!=0))

  {q=(struct  treenode2 *) malloc(sizeof(struct  treenode2))
   /*产生一个结点*/
```

```
    q->data=x;q->lch=NULL;q->rch=NULL;
     s[i]=q;
     if(i= =1) t=q;                    /*t为局部变量，代表树根结点*/
        else{j=i/2;                                    /*双亲结点编号*/
            if(i%2)= =0) s[j]->lch=q;else  s[j]->rch=q;
           }
     printf("i,x=");scanf("%d%d",&i,&x);
    }
  return(t);
  }  /*creat end*/
```

6.4 遍历二叉树

在二叉树的应用中，常常需要在树中搜索具有某种特征的结点，或对树中全部的结点逐一进行处理。这就涉及到一个遍历二叉树的问题。遍历二叉树是指以一定的次序访问二叉树中的每个结点，并且每个结点仅被访问一次。所谓访问结点，就是指对结点进行各种操作。例如，查询结点数据域的内容，或输出它的值，或找出结点位置，或执行对结点的其他操作。遍历二叉树的过程实质是把二叉树的结点进行线性排列的过程。对于线性结构来说，遍历很容易实现，顺序扫描结构中的每个数据元素即可。但二叉树是非线性结构，遍历时是先访问根结点还是先访问子树，是先访问左子树还是先访问右子树必须有所规定，这就是遍历规则。采用不同的遍历规则会产生不同的遍历结果，因此必须人为设定遍历规则。

由于一棵非空二叉树是由根结点、左子树和右子树三个基本部分组成的，遍历二叉树时只要按顺序依次遍历这三部分即可。假定我们以D、L、R分别表示访问根结点、遍历左子树和遍历右子树，则可以有六种遍历形式：DLR、LDR、LRD、DRL、RDL、RLD，若依习惯规定先左后右，则上述六种形式可归并为三种形式，即：

DLR　先根遍历

LDR　中根遍历

LRD　后根遍历

二叉树的链表存储，其结点结构如下：

```
struct  treenode2
{char  data；  /*假设数据类型char，根据需要也可为其他类型*/
struct  treenode2  *lch，*rch；
    }
```

6.4.1 先根遍历

先根遍历可以递归地描述如下：

如果根不空，则依次执行① 访问根结点，② 按先根次序遍历左子树，③ 按先根次序遍历右子树，否则返回。

先根遍历的递归算法如下:

```
/*算法描述6.2  先根遍历的递归算法*/

      void preorder(struct treenode2 * p)

{ if(p!=NULL)

    {printf("%c\n"，p->data)；        /*访问根结点*/

preorder(p->lch)；                  /*按先根次序遍历左子树*/

preorder(p->rch)；                  /*按先根次序遍历右子树*/

}

}  /*preorder*/
```

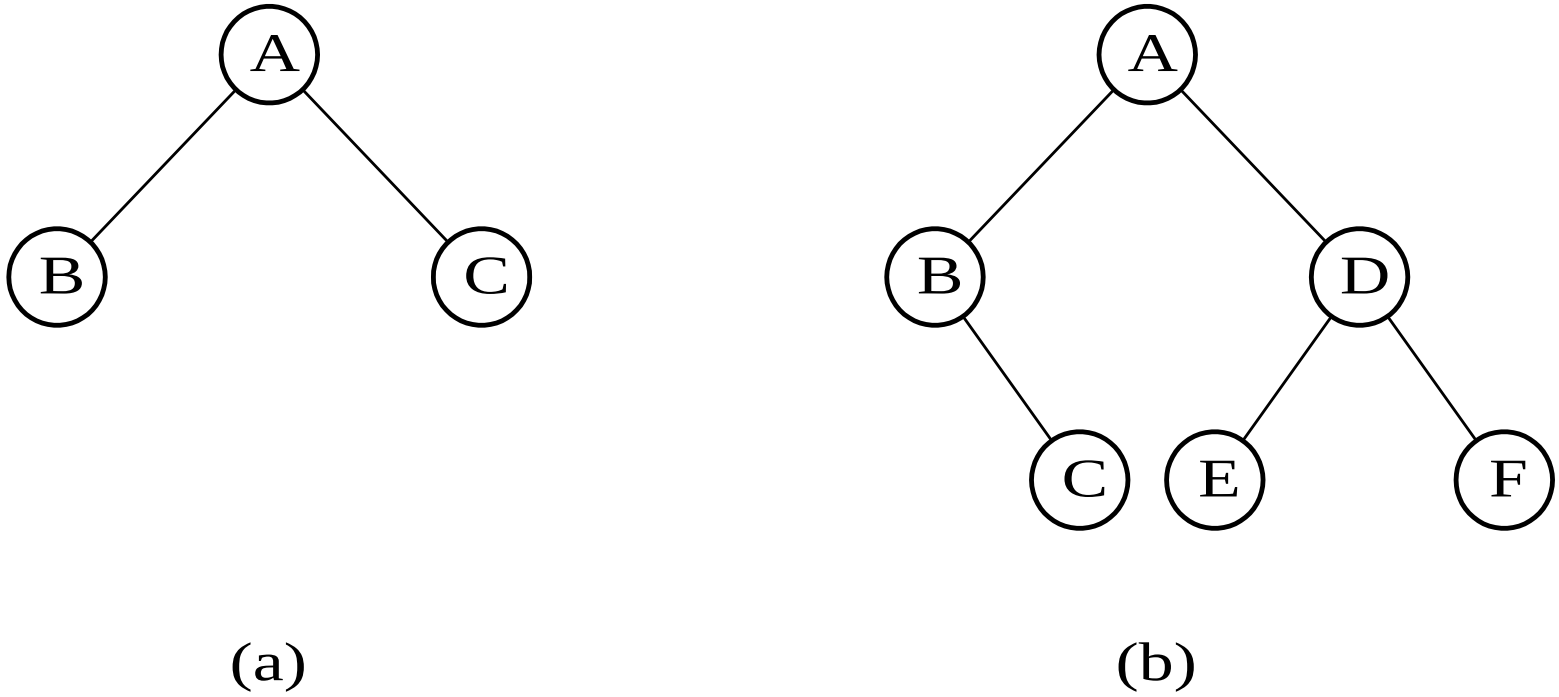


图6.11 遍历序列示例

6.4.2 中根遍历

中根遍历可以递归地描述如下：

如果根不空，则依次执行① 按中根次序遍历左子树，② 访问根结点，③ 按中根次序遍历右子树，否则返回。

中根遍历递归算法如下:

```
/*算法描述6.3  中根遍历的递归算法*/

  void inorder(struct treenode2 *p)

 {if (p!=NULL)
  {inorder(p->lch);                    /*中根遍历左子树*/
    printf("%c\n"，p->data);     /*访问根结点*/
    inorder(p->rch);                   /*中根遍历右子树*/
  }
}/*inorder*/
```

例如，图6.11(a)所示二叉树的中根遍历序列为BAC, 图6.11(b)所示二叉树的中根遍历序列为BCAEDF。

6.4.3 后根遍历

后根遍历可以递归地描述如下：

如果根不空，则依次执行① 按后根次序遍历左子树，② 按后根次序遍历右子树，③ 访问根结点，否则返回。

后根遍历递归算法如下：

```
/*算法描述6.4  后根遍历的递归算法*/
void postorder (stuct treenode2 * p )
{ if ( p!= NULL )
{ postorder ( p->lch);  /*后根遍历左子树*/
 postorder ( p->rch );              /*后根遍历右子树*/
 printf (" %c \ n"，  p->data);              /*访问根结点*/
}
} /*postorder*/
```

例如，图6.11(a)所示二叉树的后根遍历序列为BCA, 图6.11(b)所示二叉树的后根遍历序列为CBEFDA。

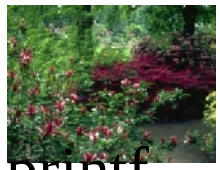
6.4.4 二叉树遍历算法的应用

1. 统计二叉树中结点个数m和叶子结点个数n

算法思路分析：在调用遍历算法时设计两个计数器变量m、n。我们知道，所谓遍历二叉树，即以某种次序去访问二叉树的每个结点，且每个结点仅访问一次，这就提供了方便的条件。每当访问一个结点时，在原访问语句printf 后边，加上一计数器语句m ++和一个判断该结点是否为叶子的语句，便可解决问题。在这里，所谓的访问结点操作已拓宽为一组语句，见下列算法的第4、5、6行。

假设用中根遍历方法统计叶子结点的个数，算法如下：

```
/*算法描述6.5  中根遍历方法统计叶子结点的个数*/
void injishu (struct treenode2 * t )
  { if (t != NULL)
     {injishu ( t->lch ) ;                          /*中根遍历左子树*/
      printf ("%c\n", t->data ) ;                   /*访问根结点*/
      m ++ ;                                        /*结点计数*/
       if (( t->lch = =NULL) && (t->rch = = NULL )) n ++ ;
                                                    /*叶子结点计数*/
       injishu ( t->rch ) ;                         /*中根遍历右子树*/
     }
  } /*injishu*/
```



如果数据域类型不是字符型而是整型，语句应该为“printf("%d＼n"，t->data);”。假设数据域类型更为复杂，则应结合具体实际重新设计输出模块。上面函数中m、n是全局变量，在主程序先置0，在调用injishu函数结束后，m值便是结点总个数，n值便是叶子结点的个数。主函数示意如下：

```
main ()
{t=creat();                  /*建立二叉树t，为全局变量*/
 m=0,n=0;                    /*全局变量m，n置初值*/
 injishu(t);                 /*求树中结点总数m，叶子结点个数n*/
 printf("m=%d,n=%d",m,n);            /*输出结果*/
}
```

当然，也可用先根或后根遍历方法统计结点个数。

2. 求二叉树的树深

首先看如下算法。

```
/*算法描述6.6  求二叉树的树深*/
void predeep (struct treenode2 * t ,int i)
  { if (t != NULL)
        {printf("%c＼n"，t->data)；      /*访问根结点*/
   i++;
   if(k＜i)k=i;
   predeep(t->lch,i);                    /*先根遍历左子树*/
   predeep(t->rch,i);                    /*先根遍历右子树*/
}
   } /*predeep*/
```

可以看出，此算法利用了先根遍历二叉树的思路，只是在这里“访问结点”操作复杂了一些，如算法中第3、4、5行。其中k为全局变量，在主程序中置初值0，在调用函数predeep之后，k值就是树的深度。形参i在主程序调用时，用一个初值为0的实参代入。当深度递归调用时，i会不断增大，k记下它的值。当返回时，退到哪一个调用层次，i会保持在本层次的原先较小值。而在返回时不论退到哪一个调用层次，k将保持较大值不变。这样，k值就是树深。相应主函数示意如下：

```
main()
{t=creat();                  /*建立二叉树t，为全局变量*/
  k=0;i=0;                   /*k,i置初值，其中k为全局变量*/
  predeep(t,i);              /*求树t的深度，i为一个辅助变量*/
  printf("k=%d",k);          /*输出树深k*/
}
```

6.5 线索二叉树

6.5.1 线索二叉树的基本概念

我们发现，具有n个结点的二叉树中有n – 1条边指向其左、右孩子，这意味着在二叉链表中的2n个孩子指针域中只用到了n –1个域，还有另外n+1个指针域是空的。我们可充分利用这些空指针来存放结点的线性前驱和后继信息。

试作如下规定：若结点有左子树，则其lch域指示其左孩子，否则令lch域指示其直接前驱；若结点有右子树，则其rch域指示其右孩子，否则令rch域指示其直接后继。为了严格区分结点的孩子指针域究竟指向孩子结点还是指向前驱或后继结点，需在原结点结构中增加两个标志域。新的结点结构为：

lch	ltag	data	rtag	rch

其中：

ltag=0　表示lch指示结点的左孩子

ltag=1　表示lch指示结点的直接前驱

rtag=0　表示rch指示结点的右孩子

rtag=1　表示rch指示结点的直接后继

算法描述为：

```
struct xtreenode
    {char data;
     struct xtreenode *lch,*rch;
     int ltag,rtag;                /*左、右标志域*/
       }
```

通常把指向前驱或后继的指针称做线索。对二叉树以某种次序进行遍历并且加上线索的过程称做线索化。经过线索化之后生成的二叉链表表示称为线索二叉树。

对一个已建好的二叉树的二叉链表进行线索化时规定(对p结点)：

(1) p有左孩子时，则令左特征域p->ltag = 0;

(2) p无左孩子时，令p->ltag = 1, 并且p->lch指向p的前驱结点;

(3) p有右孩子时，令p->rtag = 0;

(4) p无右孩子时，令p->rtag = 1,并且让p->rch指向p的后继结点。

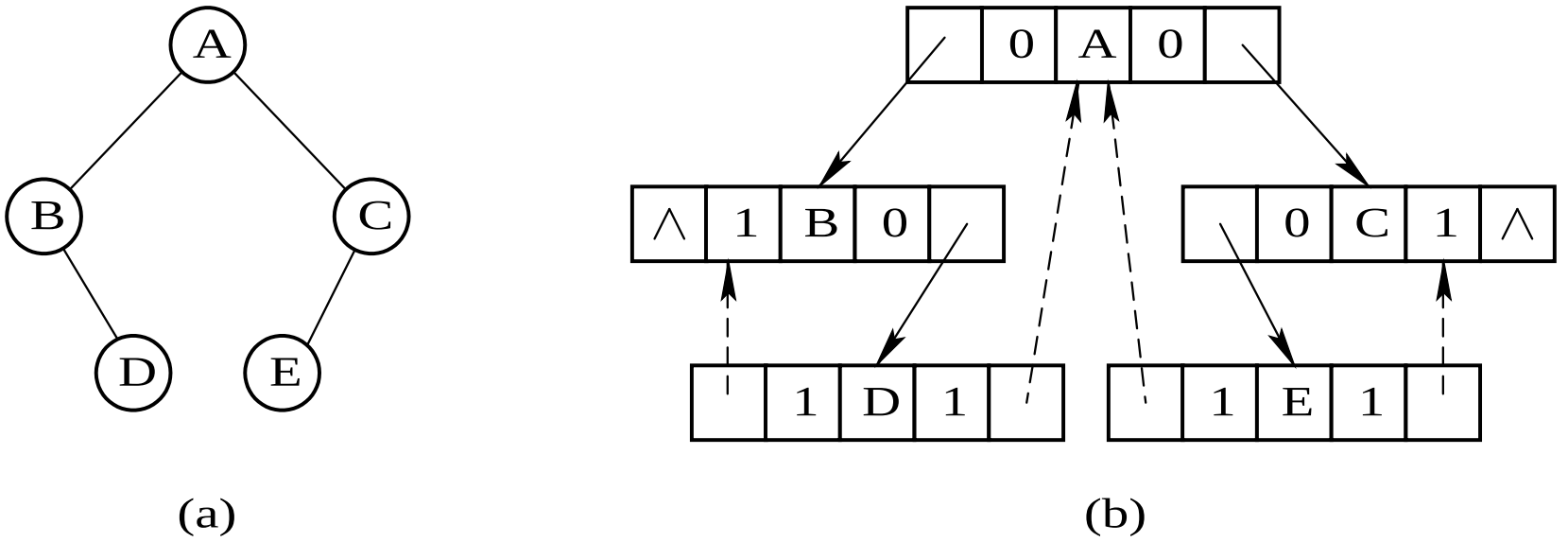


图6.12 中根次序线索树
(a) 二叉树；(b) 中根次序线索树

6.5.2 线索二叉树的逻辑表示图

按照不同的次序进行线索化，可得到不同的线索二叉树，即先根线索二叉树、中根线索二叉树和后根线索二叉树。对图6.13(a)所示的二叉树进行线索化，可得到图6.13(b)、(c)、(d)所示的三种线索二叉树的逻辑表示。

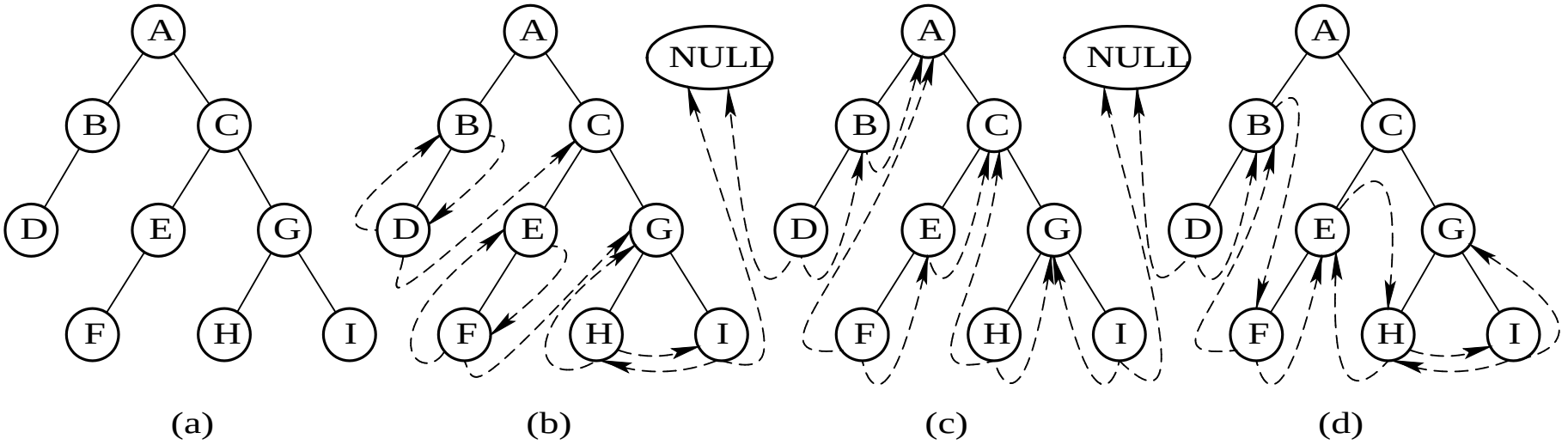


图6.13 线索二叉树的逻辑表示图
(a) 二叉树；(b) 先根线索二叉树；
(c) 中根线索二叉树；(d) 后根线索二叉树

6.5.3 中根次序线索化算法

这里重点介绍中根次序线索化的算法。中根次序线索化是在已建立好的二叉链表之上(每个结点有5个域)按中根遍历的方法在访问根结点时建立线索。

中根次序线索化递归算法如下：

```
/*算法描述6.7 中根次序线索化递归算法*/
void inthread(struct xtreenode *p)
    {if (p!=NULL)
{inthread(p->lch);
```

```
        printf ("%6c\t",p->data);

            if(p->lch!=NULL)p->ltag=0;
              else{p->ltag=1;
                 p->lch=pr;
                  }                    /*建p结点的左线索，指向前驱结点pr*/
            if(pr!=NULL)
              { if(pr->rch!=NULL) pr->rtag=0;
      else{ pr->rtag=1;
        pr->rch=p;
           }          /*前驱结点pr建右线索，指向结点p*/
   }
          pr=p;                    /*pr跟上p，以便p向后移动*/
          inthread(p->rch);
 }
}/*inthread*/
```

此算法中pr是全局变量，在主程序中置初值为空。在inthread函数中pr 始终作为当前结点p的前驱结点的指针。在线索化过程中，边判断二叉树的结点有无左、右孩子，边给相应标志域置0或1，边建立线索。在阅读此算法时，将inthread(p->lch)和inthread(p->rch)之间的一组语句看成一个整体，把这段语句理解为“访问”，很明显这里应用了典型的中根遍历算法思路。在递归调用结束时p为空，这表明pr已是最后一个结点，应该没有后继结点。所以在返回主程序后还要使pr->rch=NULL，至此整个线索化结束。主函数语句示意如下：

```
main()
  { pr=NULL;                          /*全局变量*/
    t=creat();                        /*建立二叉树*/
    inthread(t);                      /*中根线索化二叉树*/
    pr->rch=NULL;                     /*善后处理*/
  }
```

初学者在这里往往易犯错误，常把预处理pr=NULL和善后处理pr->rch=NULL放在线索化子函数void inthread (struct xtreenode * p)中，一个放最前面，另一个放最后面。这样每递归调用一次，pr就置一次空，无法记下p的前驱结点。而在从深度递归返回时，每返回一次就让pr->rch置一次空，这显然是错误的。因此，在描述递归算法时，提倡同时写出主函数来示意递归调用前的初始化处理和递归调用结束后的善后处理。

6.5.4 在中根线索树上检索某结点的前驱或后继

1. 已知q结点找出它的前驱结点

根据线索树的基本概念，当q->ltag=1时，q->lch就指向q的前驱。当q->ltag=0时，表明q有左孩子。由中根遍历的规律可知，作为根q的前驱结点(或者说以根结点为后继的结点)，它应是中根遍历q的左子树时访问的最后一个结点，即左子树的最右尾结点。图6.13(b)中，结点D是A的左子树的最右尾结点，它就是A的前驱结点。而D的后继指针指向了A，A就是D的后继结点。若用p记录q的前趋，则算法如下：

```
/*算法描述6.8 已知q结点，找出它的前驱结点*/
struct xtreenode *inpre(struct xtreenode *q)
  { if(q->ltag==1) p=q->lch;
        else { r=q->lch;
            while (r->rtag!==1) r=r->rch;
            p=r;
           }
   return(p);
  }
```

2. 已知q结点找出它的后继结点

当q->rtag=1时，q->rch即指向q的后继结点。若q->rtag=0，表明q有右孩子，那么q的后继应是中根遍历q的右子树时访问的第一个结点，即右子树的最左尾结点。图6.13(c)中，A的后继为F，C的后继为H。依照找前驱结点的算法请读者自己思考该算法的写法，这里就不再细讲。

6.5.5 在中根线索树上遍历二叉树

在中根线索树上遍历二叉树，首先从根结点开始查找二叉树的最左结点，对最左结点进行访问。然后，利用在中根线索树上求某结点后继的算法，逐一找出每个结点加以访问，直到某结点的右孩子指针域为空为止。

6.6 二叉树、树和森林

6.6.1 树的存储结构

树的存储结构有顺序结构和链表结构。顺序存储结构即向量，一般将树结点按自上而下、自左至右的顺序一一存放。如前文所介绍的完全二叉树就可以采用顺序存储结构。对于一般树结构更适合使用链表存储结构。常用的有结点定长的多叉链表和孩子－兄弟二叉链表。

图6.14所示的树是一个三叉树，可用三重链表来存储，其结点结构为：有一个数据域和三个指针域，指针域用于指向该结点的各个孩子。该树的三重链表如图6.15(a)所示。如果用孩子－兄弟链表作存储结构，其结点结构为：有一个数据域和两个指针域，一个指针指向它的长子，另一指针指向它的一个兄弟。孩子－兄弟链表如图6.15(b)所示。

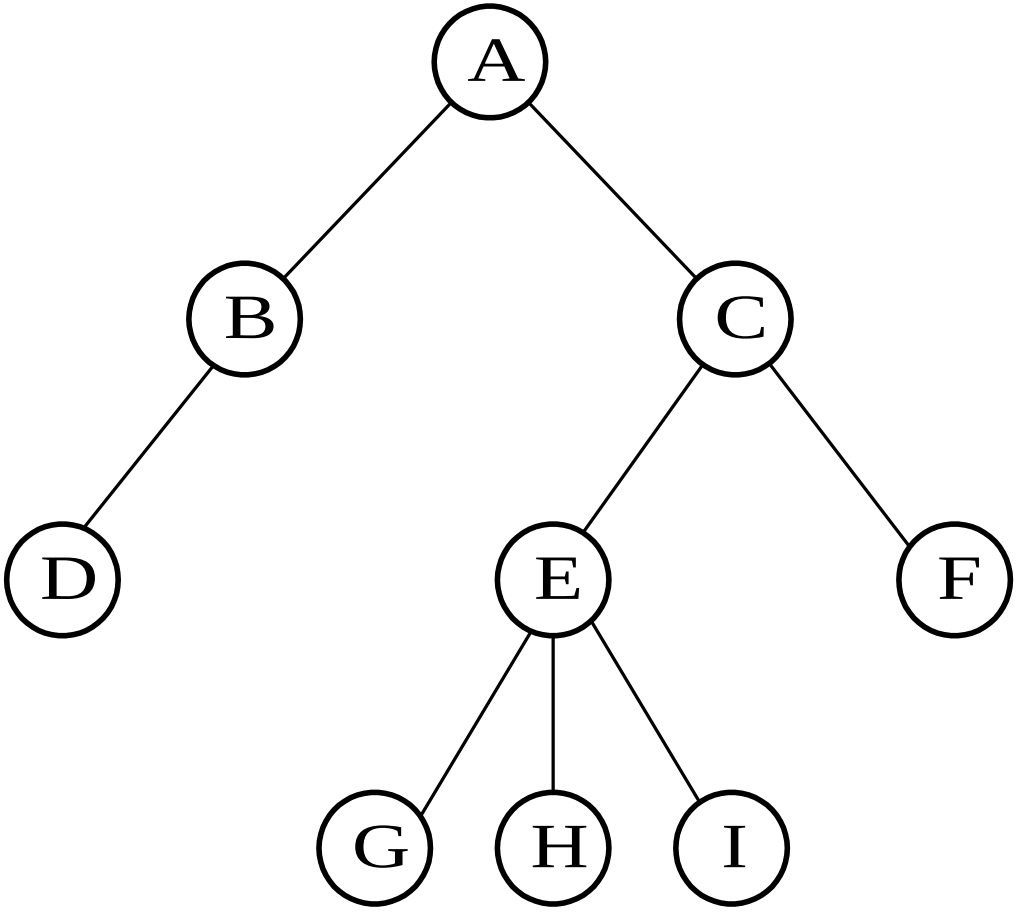


图6.14 树

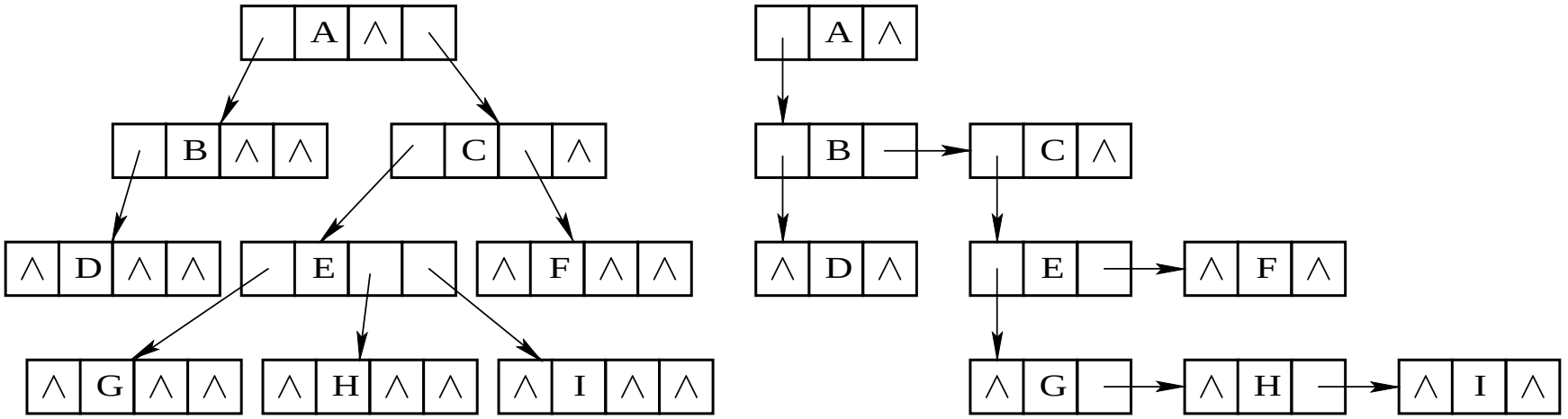


图6.15 树的存储结构
(a) 多重链表；(b) 孩子-兄弟链表

6.6.2 树与二叉树之间的转换

对于一般树，树中孩子的次序并不重要，只要双亲与孩子的关系正确即可。但在二叉树中，左、右孩子的次序是严格区分的。所以在讨论二叉树与一般树之间的转换时，为了不引起混淆，就约定按树上现有结点次序进行转换。

1. 一般树转化为二叉树

将一般树转化为二叉树的思路，主要根据树的孩子—兄弟存储方式而来，步骤是：

(1) 加线：在各兄弟结点之间用虚线相连。可理解为每个结点的兄弟指针指向它的一个兄弟。

(2) 抹线：对每个结点仅保留它与其最左一个孩子的连线，抹去该结点与其它孩子之间的连线。可理解为每个结点仅有一个孩子指针，让它指向自己的长子。

(3) 旋转：把虚线改为实线从水平方向向下旋转45°，成右斜下方向。原树中实线成左斜下方向。这样就形成一棵二叉树。

由于二叉树中各结点的右孩子都是原一般树中该结点的兄弟，而一般树的根结点又没有兄弟结点，因此所生成的二叉树的根结点没有右子树。在所生成的二叉树中某一结点的左孩子仍是原来树中该结点的长子，并且是它的最左孩子。图6.16是一个由一般树转为二叉树的实例。

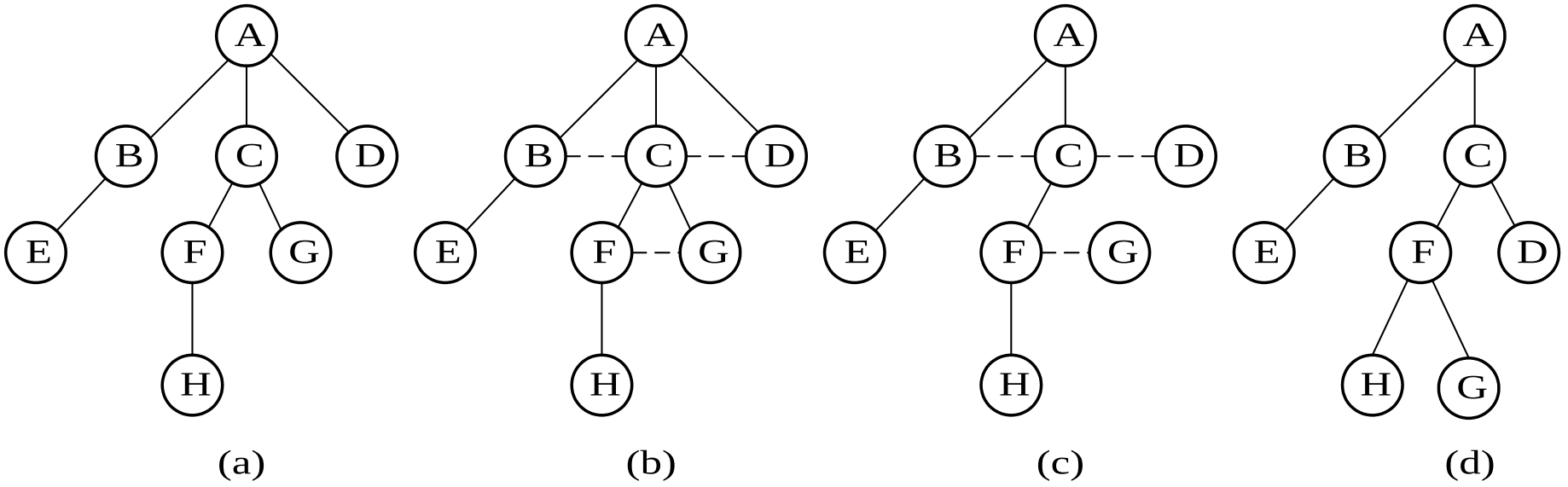


图6.16 一般树转换为二叉树
(a) 一般树；(b) 加线；(c) 抹线；(d) 旋转整理

2. 二叉树还原为一般树

二叉树还原为一般树时，二叉树必须是由某一树转换而来的且没有右子树。并非任意一棵二叉树都能还原成一般树。其还原过程也分为三步：

(1) 加线：若某结点i是双亲结点的左孩子，则将该结点i的右孩子以及当且仅当连续地沿着右孩子的右链不断搜索到所有右孩子都分别与结点i的双亲结点用虚线连接。

(2) 抹线：把原二叉树中所有双亲结点与其右孩子的连线抹去。这里的右孩子实质上是原一般树中结点的兄弟，抹去的连线是兄弟间的关系。

(3) 整理：把虚线改为实线，把结点按层次排列。

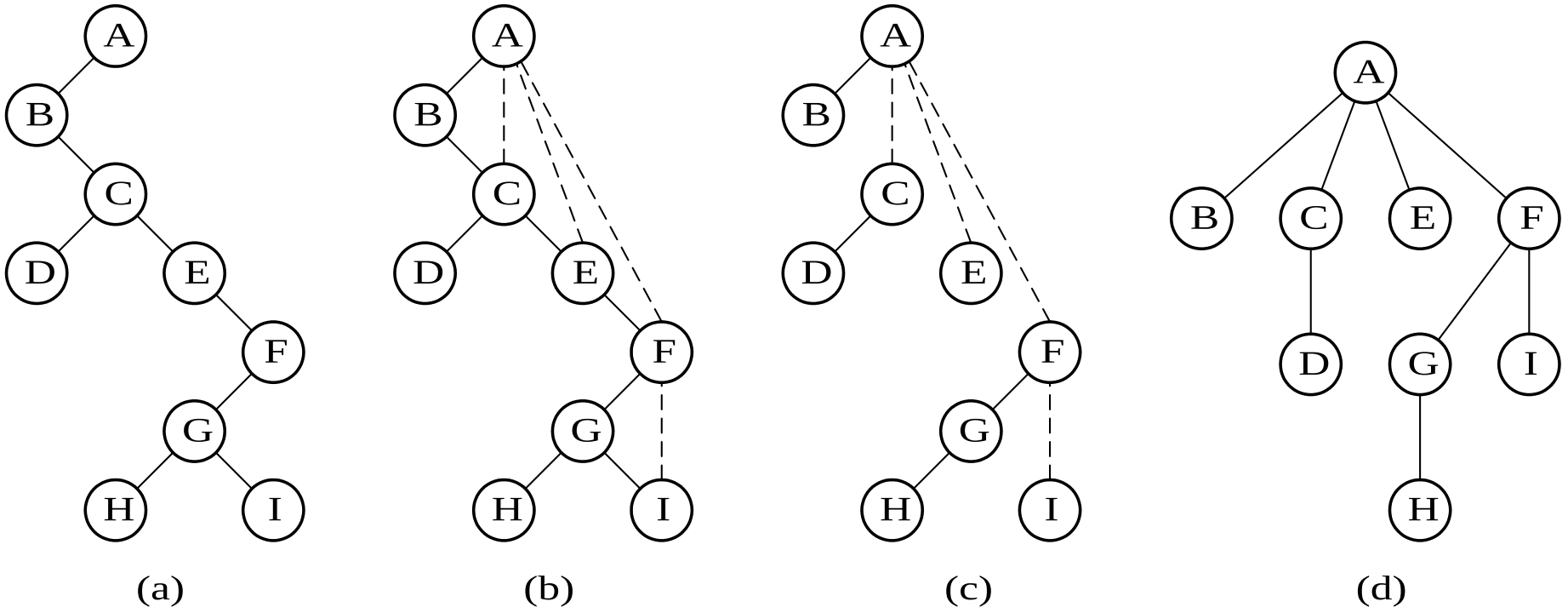
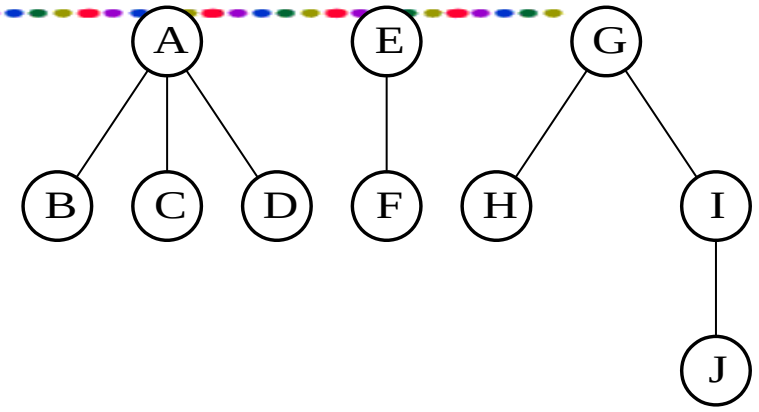


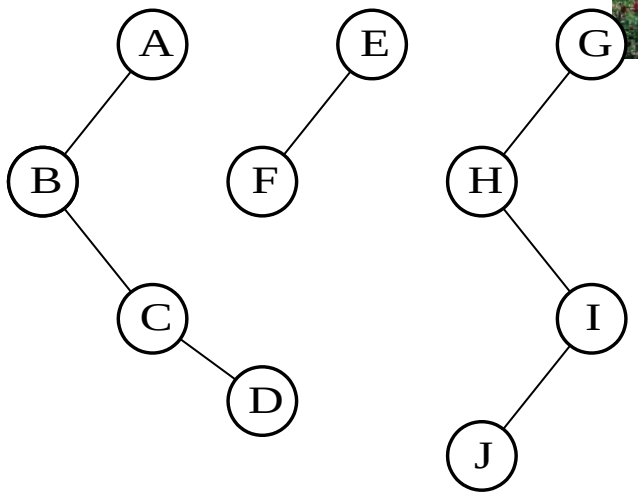
图6.17 二叉树还原为一般树

(a) 二叉树；(b) 还原加线；(c) 还原抹线；(d) 还原整理

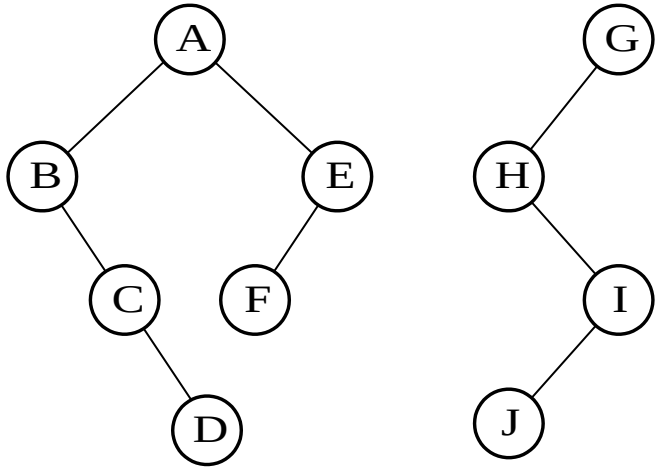
6.6.3 森林与二叉树之间的转换



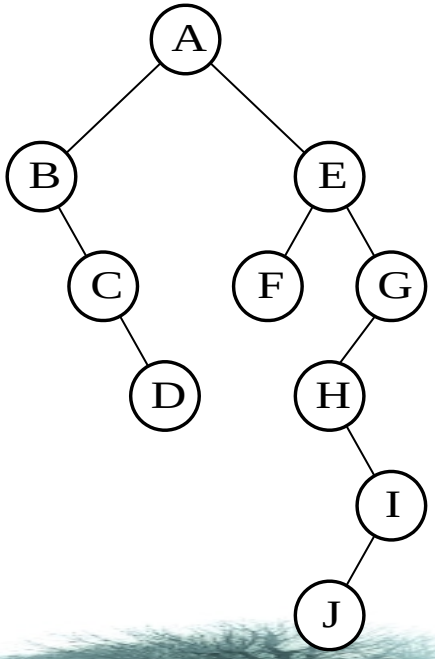
(a)



(b)



(c)



(d)

图6.18 森林转换为二叉树

1. 森林转换为二叉树

森林转换为二叉树的步骤为：

(1) 将森林中每棵子树转换成相应的二叉树，形成有若干二叉树的森林。

(2) 按森林图形中树的先后次序，依次将后边一棵二叉树作为前边一棵二叉树根结点的右子树，这样整个森林就生成了一棵二叉树，实际上第一棵树的根结点便是生成后的二叉树的根结点。图6.18是森林转化为二叉树的示例。

2. 二叉树还原为森林

将一棵由森林转换得到的二叉树还原为森林的步骤是：

(1) 抹线：将二叉树的根结点与其右孩子的连线以及当且仅当连续地沿着右链不断地搜索到的所有右孩子的连线全部抹去，这样就得到包含有若干棵二叉树的森林。

(2) 还原：将每棵二叉树按二叉树还原为一般树的方法还原为一般树，于是得到森林。

这部分的图示，请读者自己练习画出。

6.6.4 一般树或森林的遍历

一般树的遍历主要是先根和后根遍历，一般不讨论中根遍历。树的先根遍历首先访问树的根结点，然后从左至右逐一先根遍历每一棵子树。树的后根遍历首先后根遍历树的最左边的第一棵子树，接着从左至右逐一后根遍历每一棵子树，最后访问树的根结点。一般树转换为二叉树后，此二叉树没有右子树，对此二叉树的中根遍历结果与上述一般树的后根遍历结果相同。

6.7 树的应用

6.7.1 二叉排序树

1. 二叉排序树的定义和特点

定义：二叉排序树(Binary Sort Tree)或是空树，或是非空树。

对于非空树：

(1) 若左子树不空，则左子树上各结点的值均小于它的根结点的值；

(2) 若右子树不空，则右子树上各结点的值均大于等于它的根结点的值；

(3)它的左、右子树又分别是二叉排序树。

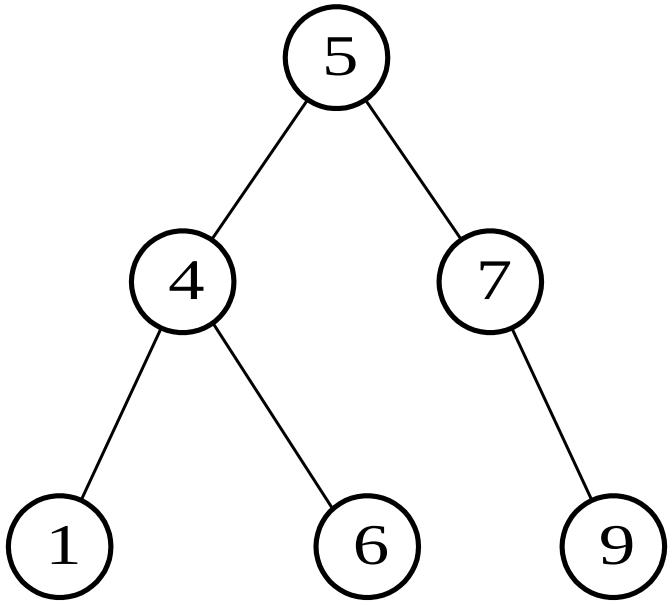


图6.19 二叉排序树

特点：对二叉排序树进行中根遍历，可得到一个由小到大的序列。例如，对图6.19所示的二叉排序树进行中根遍历，则得到序列：1，4，5，6，7，9。

2. 建立二叉排序树的算法

建立二叉排序树，实质上是不断地进行插入结点的操作。设有一组数据：K={k_1，k_2，…，k_n}，将它们一一输入建成二叉排序树。我们用二叉链表作为存储结构。结点结构如下：

```
struct  bstreenode
   {int  data;
    struct  bstreenode *lch,*rch;
   }
```

算法思路分析：

(1) 让k_1作根；

(2) 对于k_2，若$k_2<k_1$，令k_2做k_1的左孩子；否则令k_2做k_1的右孩子；

(3) 对于k_3，从根k_1开始比较。若$k_3<k_1$，到左子树中找，否则到右子树中找；找到适当位置进行插入；

(4) 对于k_4，k_5，…，k_n，重复第(3)步，直到k_n处理完为止。

在建立过程中，每输入一个数据元素，就插入一次。现把插入一个结点的算法单独编写，而在建立二叉排序树的函数中对其进行调用。

首先看建立二叉排序树的主体算法。

```
/*算法描述6.9  建立二叉排序树的主体算法*/
struct bstreenode creat()
      { printf("n=?"); scanf("%d",&n ); t==NULL;
for(i=1;i<=n; i++)
      {printf("k%d=?",i); scanf("%d", &k);
       s=(struct bstreenode * )malloc(sizeof(struct bstreenode));
       s->data=k; s->lch=NULL；  s->rch=NULL;
       insertl(t,s);          /*或调用insert2(t, s)*/
      }
return(t);
}
```

在二叉排序树中，插入一个结点s的递归算法如下：

```
/*算法描述6.10  在二叉排序树中，插入一个结点s的递归算法*/
void insertl(struct bstreenode *t, struct bstreenode *s)
    {if(t==NULL) t=s;
      else if(s->data<t->data) insert1(t->lch,s);
     /*将s插入t的左子树*/
      else insert1(t->rch,s);                    /*将s插入t的右子树*/
}
```

在二叉排序树中，插入一个结点s的非递归算法如下：

```
/*算法描述6.11  在二叉排序树中，插入一个结点s的非递归算法*/
void insert2(struct bstreenode *t, struct bstreenode *s)
      { if(t==NULL) t=s;
else { p=t;
       while(p!=NULL)
         { q=p;                    /*当p向子树结点移动时，q记p的双亲位置*/
          if(s->data<p->data) p=p->lch;
                    else p=p->rch;
         }
       if(s->data<q->data) q->lch=s;
         else q->rch=s;   /*当p为空时, q就是可插入的地方*/
       }
}
```

假设给出一组数据{5，3，7，6，9，2}，对照上述算法生成二叉排序树的过程如图6.20所示。

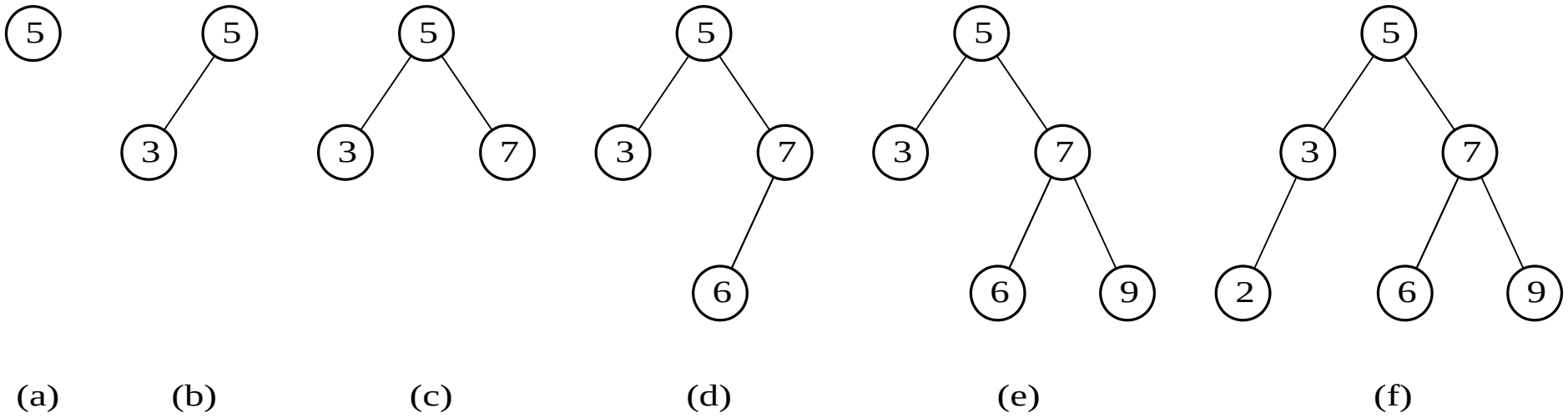


图6.20 二叉排序树的生成

由此可见，在二叉排序树上插入结点不需遍历树，仅需从根结点出发，走一条根到某个有空子树的结点的路径，使该结点的空指针指向被插入结点，使被插入结点成为新的叶子结点。如果仍使用前边6个数据，但输入先后顺序发生变化，那么生成的二叉排序树如何？请思考。

3. 在二叉排序树中删除结点

算法思路分析：在二叉排序树上删除一个结点，应该在删除之后仍保持二叉排序树的特点。要删除某结点p，p结点和它的双亲结点f都是已知条件，这里的关键是怎样找一个结点s来替换p结点。下面分三种情况来讨论。

(1) p结点无右孩子，则让p的左孩子s上移替代p结点；

(2) p结点无左孩子，则让p的右孩子s上移替代p结点；

(3) p结点有左孩子和右孩子，可用它的前驱(或后继)结点s代替p结点。现假定用它的后继结点来代替，这个结点s应是p的右子树中数据域值最小的结点，或者说是p的右子树中的最左结点。因其值域最小(在右子树中)，所以它一定没有左孩子。这时先让p结点取s结点的值，然后可按第(2)种情况处理，删除s结点，这就等效于删除了原p结点。具体算法这里不再细讲，请读者思考。

6.7.2 哈夫曼树及其应用

哈夫曼(Huffman)树，又称最优二叉树，是一类带权路径长度最短的树，有着广泛的应用。

1. 哈夫曼树的基本概念

首先我们要学习一些与哈夫曼树有关的术语。

两个结点之间的路径长度：树中一个结点到另一个结点之间的分支数目。

树的路径长度：从根结点到每个结点的路径长度之和。

树的带权路径长度：设一棵二叉树有n个叶子，每个叶子结点拥有一个权值W_1，W_2，…，W_n，从根结点到每个叶子结点的路径长度分别为L_1，L_2，…，L_n，那么树的带权路径长度为每个叶子的路径长度与该叶子权值乘积之和，通常记作：

$$WPL = \sum_{k=1}^{n} W_k L_k$$

为了直观起见，在图6.21中，把带权的叶子结点画成方形，其它非叶子结点仍为圆形。请看图6.21中的三棵二叉树以及它们的带权路径长度。

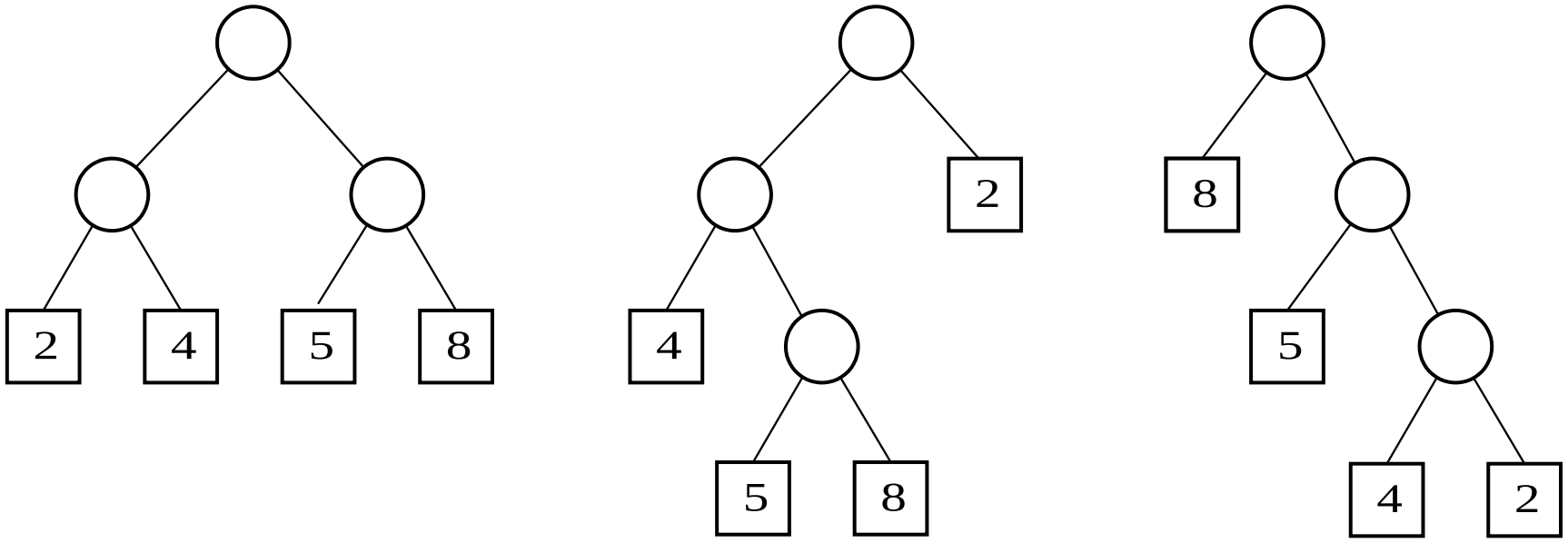
(a) WPL=2×2+4×2+5×2+8×2=38

(b) WPL=4×2+5×3+8×3+2×1=49

(c) WPL=8×1+5×2+4×3+2×3=36

注意: 这三棵二叉树叶子结点数相同，它们的权值也相同，但是它们的WPL带权路径长各不相同。图6.21(c)所示二叉树的WPL最小。它就是哈夫曼树，最优树。

哈夫曼树：在具有同一组权值的叶子结点的不同二叉树中，带权路径长度最短的树。



(a) (b) (c)

图6.21 具有不同带权路径长度的二叉树
(a) WPL=38；(b) WPL=49；(c) WPL=36

2. 哈夫曼树的构造及其算法

1) 构造哈夫曼树的方法

对于已知的一组叶子的权值W_1，W_2，…，W_n：

(1) 首先把n个叶子结点看作n棵树(有一个结点的二叉树)，把它们看作一个森林。

(2) 在森林中把权值最小和次小的两棵树合并成一棵树，该树根结点的权值是两棵子树权值之和。这时森林中还有n－1棵树。

(3) 重复第(2)步直到森林中只有一棵树为止。(此树就是哈夫曼树。)

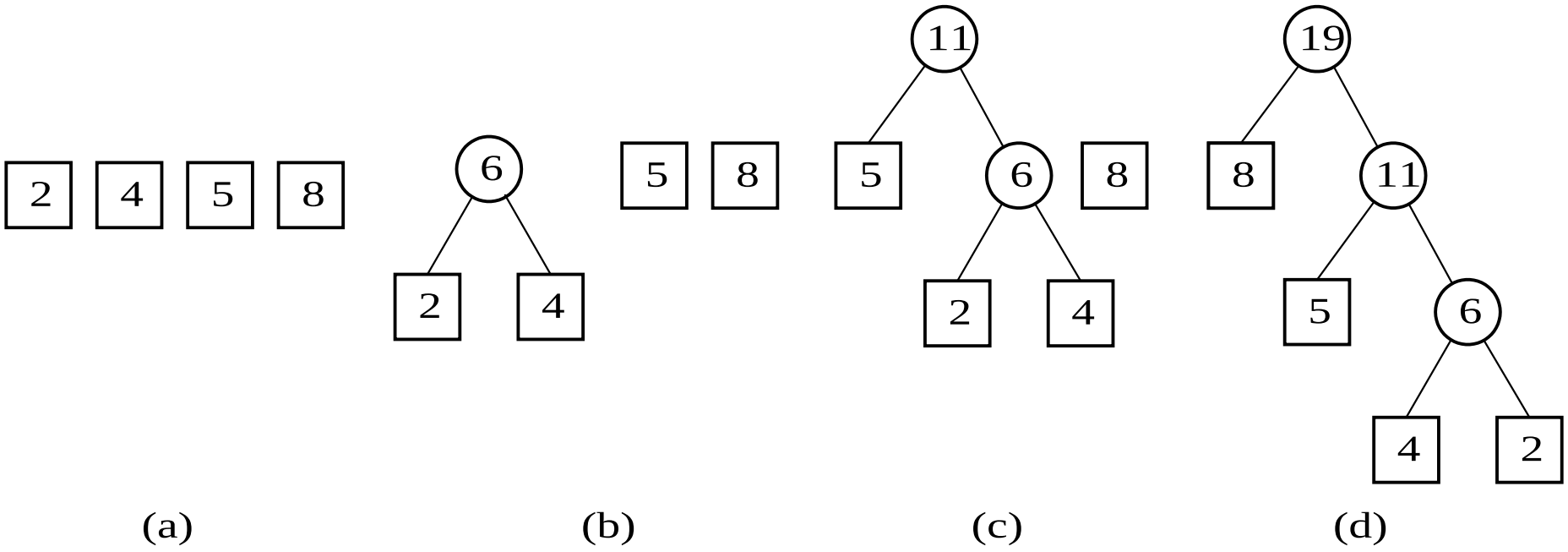


图6.22 哈夫曼树构造过程
(a) 森林中有四棵树；(b) 森林中有三棵树；
(c) 森林中有两棵树；(d) 生成一棵树

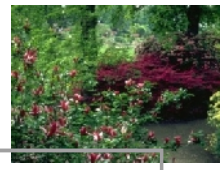
此时我们或许会问，n个叶子构成的哈夫曼树其带权路径长度唯一吗？答案是确实唯一。树形唯一吗？答案是不唯一。因为将森林中两棵权值最小和次小的子树合并时，哪棵做左子树，哪棵做右子树并不严格限制。图6.22之中的做法是把权值较小的当作左子树，权值较大的当作右子树。如果反过来也可以，画出的树形有所不同，但WPL值相同。

2) 哈夫曼算法实现

讨论算法实现需选择合适的存储结构，因为哈夫曼树中没有度为1的结点。这里选用顺序存储结构。由二叉树性质可知$n_0=n_2+1$，而现在总结点数为n_0+n_2，也即$2n_0-1$。叶子数n_0若用n表示则二叉树结点总数为2n－1，向量的大小就定义为2n－1。假设n<10，存储结构如下：

```
struct  hftreenode
 {int data;            /*权值域*/
  int lch,rch;            /*左、右孩子结点在数组中的下标*/
  int tag;                /*tag=0结点独立，tag= 1结点已并入树中*/
 }
struct  hftreenode r[20];
```

首先需将叶子权值输入r向量，lch,rch,tag域全置零，如果用前边的一组数值{2，4，5，8}初始化向量r(见图6.23(a))，然后执行算法，可得出如图6.23(b)所示的结果。设t为指向哈夫曼树的根结点(在此是数组元素)的指针，则算法如算法6.12所示。



0	0	2	0		1	0	2	0
0	0	4	0		1	0	4	0
0	0	5	0		1	0	5	0
0	0	8	0		1	0	8	0
					1	1	6	2
					1	3	11	5
					0	4	19	6

(a) (b)

图6.23 哈夫曼树向量存储结构示意图
(a) 初始状态；(b) 最终状态

```
/*算法描述6.12  哈夫曼算法*/
int huffman (struct hftreenode r[20])
{scanf("n=%d",&n);                          /*n为叶子结点的个数*/
    for(j=1;j<=n; j++)
      {scanf("%d", &r[j].data);
       r[j].tag=0;r[j].lch=0;r[j].rch=0;
      }
    i=0;
    while (i<n-1)                          /*合并n–1次*/
        {x1=0; m1=32767;          /*m1是最小值单元，x1为下标号*/
```

```
x2=0; m2=32767;     /*m2为次小值单元，x2为下标号*/
for(j=1;j<=n+i;j++)
        {if((r[j].data<m1)&&(r[j].tag==0))
            {m2=m1; x2=x1;
             m1=r[j].data;x1=j;
            }
         else if ((r[j].data<m2)&&(r[j].tag==0))
            {m2=r[j].data;
             x2=j;
            }
        }
```

```
        r[x1].tag=1;r[x2].tag=1;i++;
         r[n+i].data=r[x1].data+r[x2].data;      /*m1+m2*/
         r[n+i].tag=0;r[n+i].lch=x1;r[n+i].rch=x2;
      }
t=2*n-1;return(t);
   } /*end*/
```

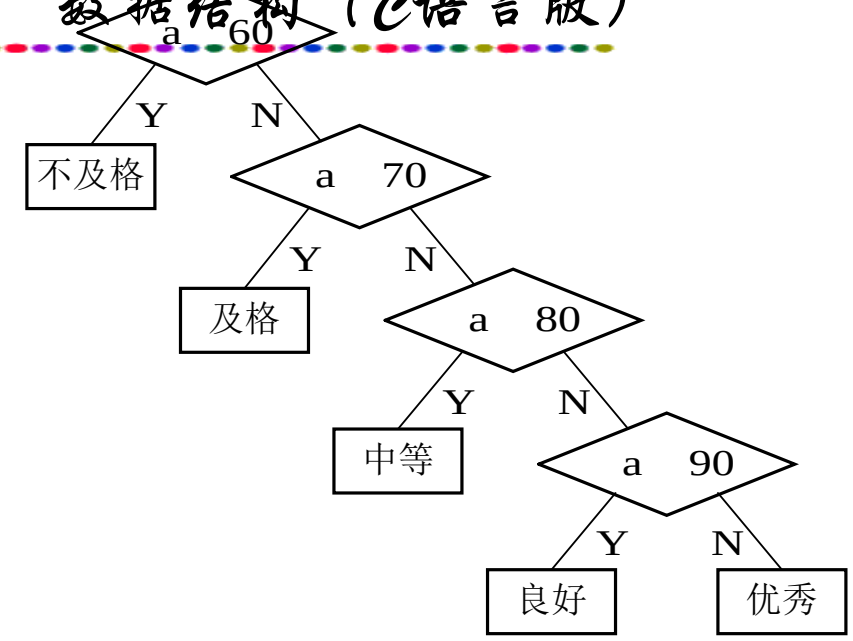
图6.23中仅给出了所用到的数组元素，其它略去未画。在算法6.12中主要有一个二重循环，内循环的平均循环次数均为O(n)，外循环大约n次，所以该算法的时间复杂度为$O(n^2)$。

3. 哈夫曼树的应用

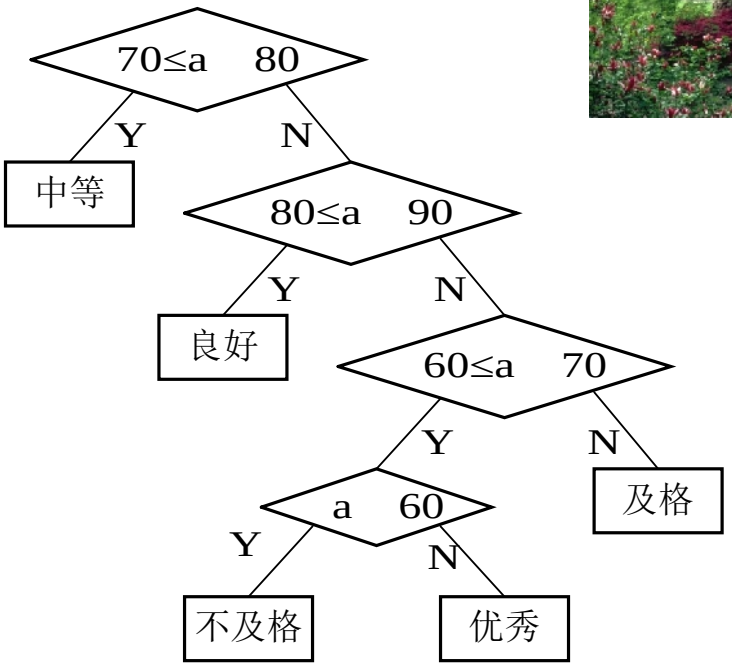
1) 判定树

在考查课记分时往往把百分制转换成优秀(x≥90)、良好(80≤x<90)、中等(70≤x<80)、及格(60≤x<70)、不及格(x<60)5个等级。若不考虑学生考试分数的分布概率，程序判定过程很容易写成图6.24(a)，可看出这种情况的x值要比较2～3次才能确定等级。而学生中考试不及格的人数很少，x值比较一次即可定等级。

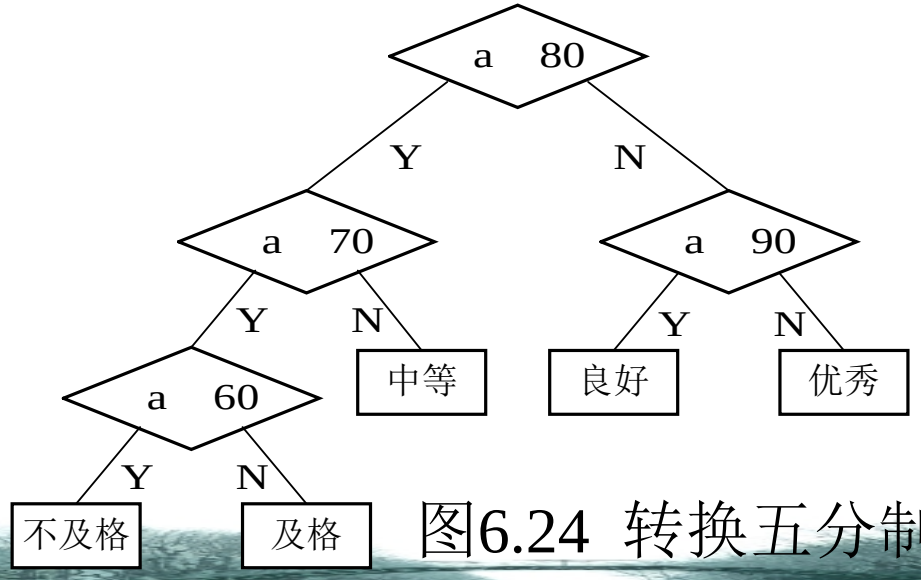
能否使出现次数多的在70～80分之间的x值比较次数减少些，而使很少出现的低于60分的x值比较次数多一些，以便提高程序的运行效率呢？假设学生成绩对于不及格、及格、中等、良好和优秀的分布概率分别为5%，15%，40%，30%，10%，以它们为叶子的权值来构造哈夫曼树，如图6.24(b)所示。此时带权路径长最短，其值为205。它可以使大部分的分数值经过较少的比较次数得到相应的等级。但是，事情往往不是绝对的，此时每个判断框内的条件较为复杂，需比较两次，反而降低了运行效率。所以我们采用折衷做法，调整后得图6.24(c)所示的判定树，它更加切合实际。



(a)



(b)



(c)

图6.24 转换五分制不同判定过程

2) 哈夫曼编码

在通信及数据传输中多采用二进制编码，即由0、1组成的字符串。接收方收到一系列0、1串后，再把它还原成文字，即译码。

例如：需传送的电文为“ACDACAB”，其间只用到了四个字符，则只需两个字符的串便足以分辨。令A、B、C、D的编码分别为00、01、10、11，则电文的二进制代码串为00101100100001，总码长14位。接收方按两位一组进行分割，便可译码。

但是，在传送电文时，总希望总码长尽可能的短。如果对每个字符设计长度不等的编码，且让电文中出现次数较多的字符采用尽可能短的编码，则传送电文的总长便可减少。上例电文中A和C出现的次数较多，我们可以再设计一种编码方案，即A、B、C、D的编码分别为0、01、1、11，此时，电文“ACDACAB”的二进制代码串为011101001，总码长9位，显然缩短了。

但是，接收方收到该代码串后无法进行译码。比如代码串的“01”是代表B还是代表AC呢？因此，若要设计长度不等的编码，必须是任一个字符的编码都不是另一个字符的编码的前缀，这个编码称为前缀编码。电话号码就是前缀码，例如110是报警电话的号码，其他的电话号码就不能以110开头了。

利用哈夫曼树不仅能构造出前缀码，而且还能使电文编码的总长度最短。方法如下：

假定电文中共用了n种字符，每种字符在电文中出现的次数为W_i(i=1，2，…，n)。以W_i作为哈夫曼树叶子结点的权值，用我们前面所介绍的哈夫曼算法构造出哈夫曼树，然后将每个结点的左分支标上“0”，右分支标上“1”，则从根结点到代表该字符的叶子结点之间，沿途路径上的分支号组成的代码串就是该字符的编码。

例如，在电文“ACDACAB”中，A、B、C、D四个字符出现的次数为3、1、2、1，则按上述方法可得到各字符的哈夫曼编码：A(0)，B(110)，C(10)，D(111)，见图6.25。

信息编码是一个复杂的问题，还应考虑其他一些因素。比如译码时较困难，还有检测、纠正等问题，都应考虑在内。这里仅对哈夫曼树举了一个应用实例。

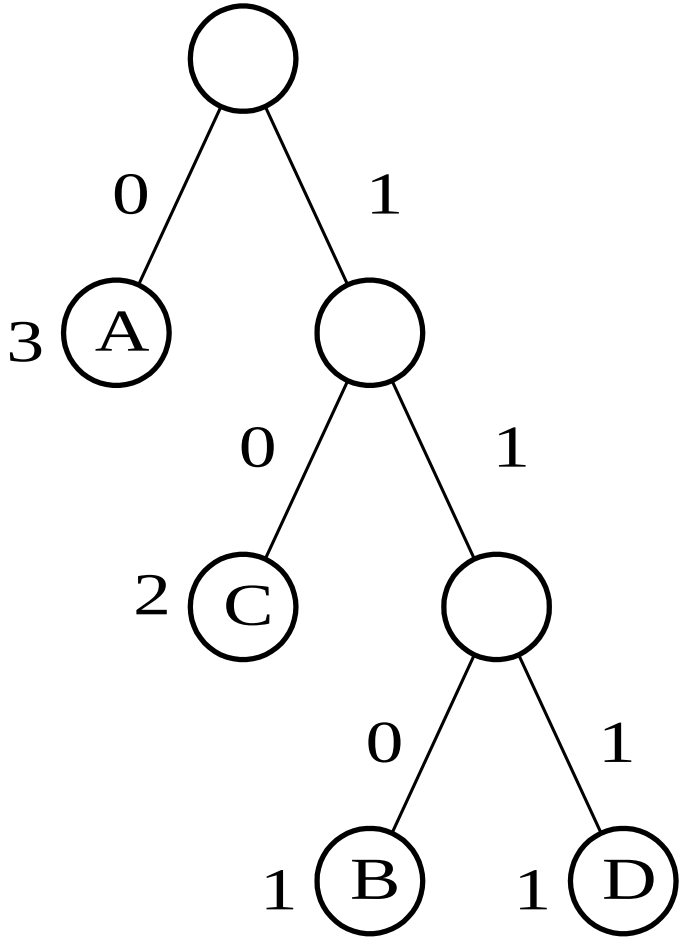


图6.25 哈夫曼编码

6.8 实习：二叉树的建立和遍历

二叉树的建立和遍历C语言源程序示例如下：

```
#include "stdio.h"
#include "stdib.h"
#define ELEMTP int;
struct node
      { ELEMTP data;
  struct node *lc, *rc;
      }
struct node *root;        /*根结点指针root定义为全局变量*/
  int m=0;                          /*统计叶子结点数，初值置零*/
```

```
main()
    {int cord;
struct node *creat();
void preorderz(struct node *t);          /*参数必须与下边函数说明一致*/
void inorder(struct node *t);
void postorder(struct node *t);
do {  printf("\n        主菜单                    \n");
printf("    1    建立二叉树                \n");
printf("    2    先根非递归遍历              \n");
printf("    3    中根递归遍历                \n");
printf("    4    后根递归遍历，求叶子结点数   \n");
      printf("    5    结束程序运行                \n");
```

```
printf("-------------------------------------\n");
printf("请输入您的选择(1，2，3，4) ");
    scanf("%d",&cord);
switch (cord)
{case 1: {root=creat();                          /*建立二叉树*/
        printf("建立二叉树程序已执行完\n");
        printf("请返回主菜单，用遍历算法验证程序的正确性\n");
      } break;
   case 2: {preorderz(root);
        printf("先根非递归遍历程序已执行完\n");
      } break;
```

```
        case 3: {  inorder(root);
                   printf("\n中根遍历二叉树程序已执行完\n");
                 } break;
        case 4: {  postorder(root);
                   printf("\n后根遍历二叉树程序已执行完\n");
                   printf("叶子结点数m=%3d\n",m;)
                 }
        case 5: {  printf("二叉树程序执行完，再见! \n");
                   exit(0);
                 }
      }  /*switch*/
    } while(cord<=5);
    } /*main*/
```

```
struct node * creat()      /*建立二叉树*/
{ struct node *t, *q, *s[30];
    int  i,j,x;
 printf("i, x=");scanf("%d%d",&i,&x);
        /* i是按满二叉树编号x结点应有的序号, x是结点数据*/
while ((i!=0)&&(x!=0))
      {q=(struct node *)malloc(sizeof(struct node )) /*产生一个结点*/
       q->data=x; q->lch=NULL;q->rch=NULL;
       s[i]=q;
        if(i==1)t=q;                          /*t代表树根结点*/
            else{j=i/2;                       /*双亲结点编号*/
```

```
            if((i%2)==0) {s[j]->lch=q;else s[j]->rch=q;}
                }
          printf("i,x=");scanf("&d, &d",&i, &x);
        }
    return(t);
   }/*creat*/


void preorderz(struct node *p)          /*先根非递归算法*/
   {struct node *q,*s[30];        /*s是辅助栈*/
     int top,bool;
      q=p; top=0;
         bool=1;     /*bool=1为真值继续循环;bool=0为假值栈空,结束循环*/
```

```
do {while(q!=NULL){ printf("%6d " ， q->data)；/*访问根结点*/
                top++;s[top]=q;q=q->lch;
               }
   if(top==0) bool=0;
   .       else{q=s[top];top--;q=q->rch;}
   }while (bool);
printf("/n");
      }/*preorder*/

void inorder(struct node *p)
      {if(p!=NULL) {inorder(p->lch);
```

```
            printf("%6d",p->data)
            inorder(p->rch);
        }
}/*inorder*/

oid postorder(struct node *p)
{if(p!=NULL) {postorder(p->lch);
            postorder(p->rch);
            printf("%6d",p->data);
            if(p->lch==NULL&&p->rch==NULL) m++;      /*统计叶结点*/
        }
    }/*postorder*/
```

上面有关二叉树的源程序在上机调试时有几点应注意：在主函数main中调用二叉树的建立函数creat时，指向根结点的指针root必须是全局变量；在调用递归遍历函数时，访问输出的格式不带换行符"/n"，当调用结束返回主调函数后，在主调函数中设一输出语句令其回车换行。

习 题 6

1. 找出图6.26所示树T的深度、度、分支结点和叶子结点。

2. 对于三个结点A、B、C，可组成多少种不同的二叉树？请画出。

3. 分别写出图6.27所示二叉树的三种遍历次序的序列。

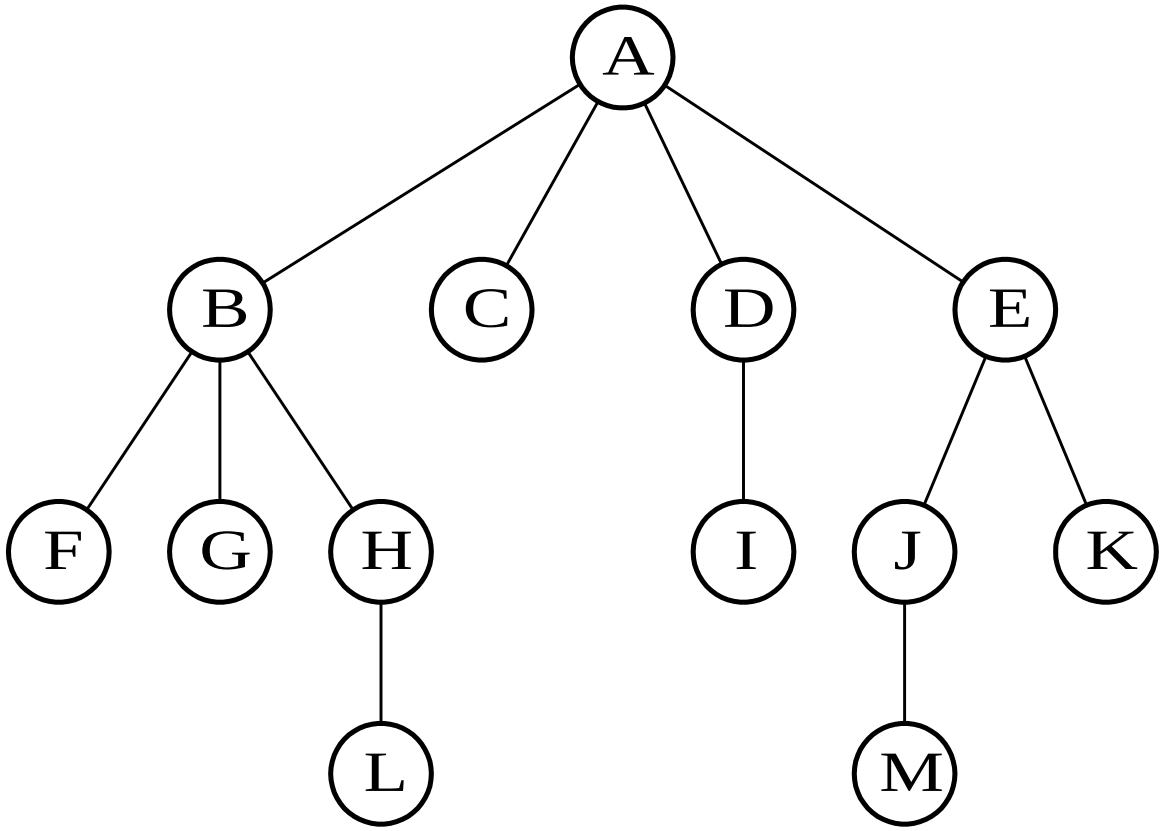


图6.26 树T

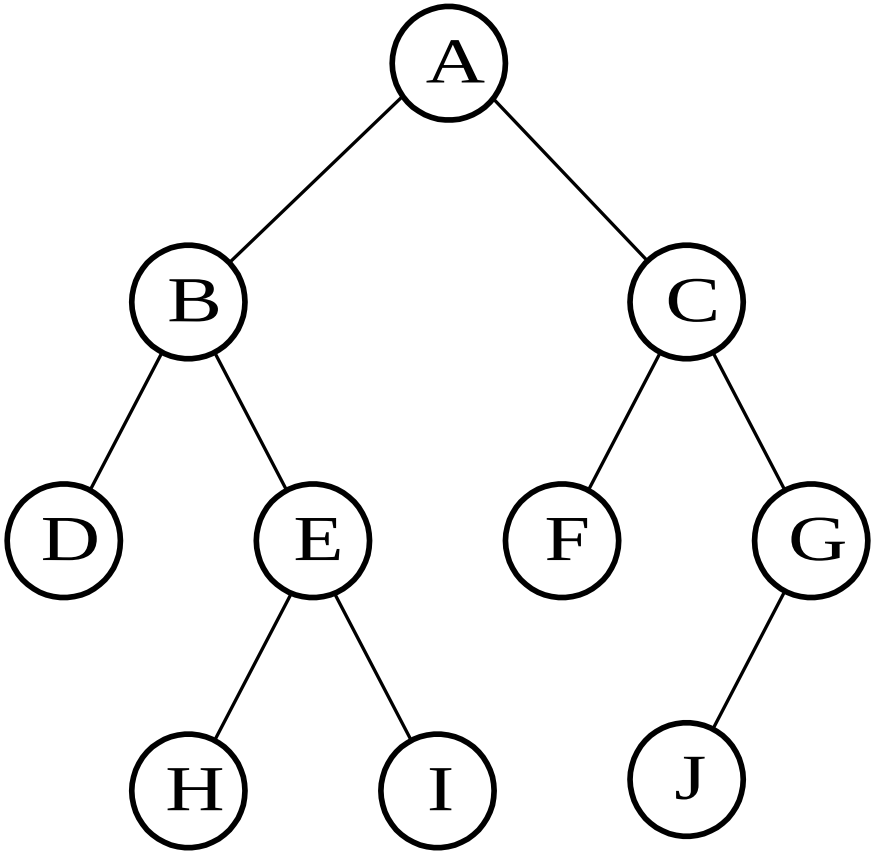


图6.27 二叉树

4. 写出先根遍历的非递归算法。

5. 现有按后根遍历二叉树的结果为C、B、A，有几种不同的二叉树可以得到这一结果？

6. 若已知某二叉树的两种遍历序列，可推断出二叉树的形状吗？结果是否唯一？为什么？

7. 画出图6.27所示二叉树先根、中根、后根线索化树的逻辑表示图。

8. 对于同一组数据，若以不同的顺序输入，所建立的二叉树树形、树深是否相同？中根遍历结果是否相同？为什么？

9. 已知三个互不相等的整数，若以这三个数来生成二叉排序树，可以生成几种不同形状的树？

10. 有一组数值14、21、32、15、28，画出哈夫曼树的生成过程及最后结果。

11. 用哈夫曼算法构造哈夫曼树时森林中两棵权值最小和次小的子树合并时，哪棵做左子树并不影响WPL？影响哈夫曼编码吗？

12. 试写出在二叉排序树中删除一个结点的算法。

13. 为什么说一般二叉树一般不采用顺序存储结构，而满二叉树和完全二叉树多采用顺序存储结构？

第7章　图

7.1 基本术语

图(Graph)：图G由两个集合V(G)和E(G)所组成，记作G=(V，E)，其中V(G)是图中顶点的非空有限集合，E(G)是图中边的有限集合。

有向图(Digraph)：如果图中每条边都是顶点有序对，即每条边在图示时都用箭头表示方向，则称此图为有向图。有向图的边也称为弧。如图7.1中G_1是有向图，它由$V(G_1)$和$E(G_1)$组成。其中：

$V(G_1)=\{v_1，v_2，v_3\}$

$E(G_1)=\{<v_1\ v_2>，<v_2\ v_3>，<v_1\ v_3>\}$

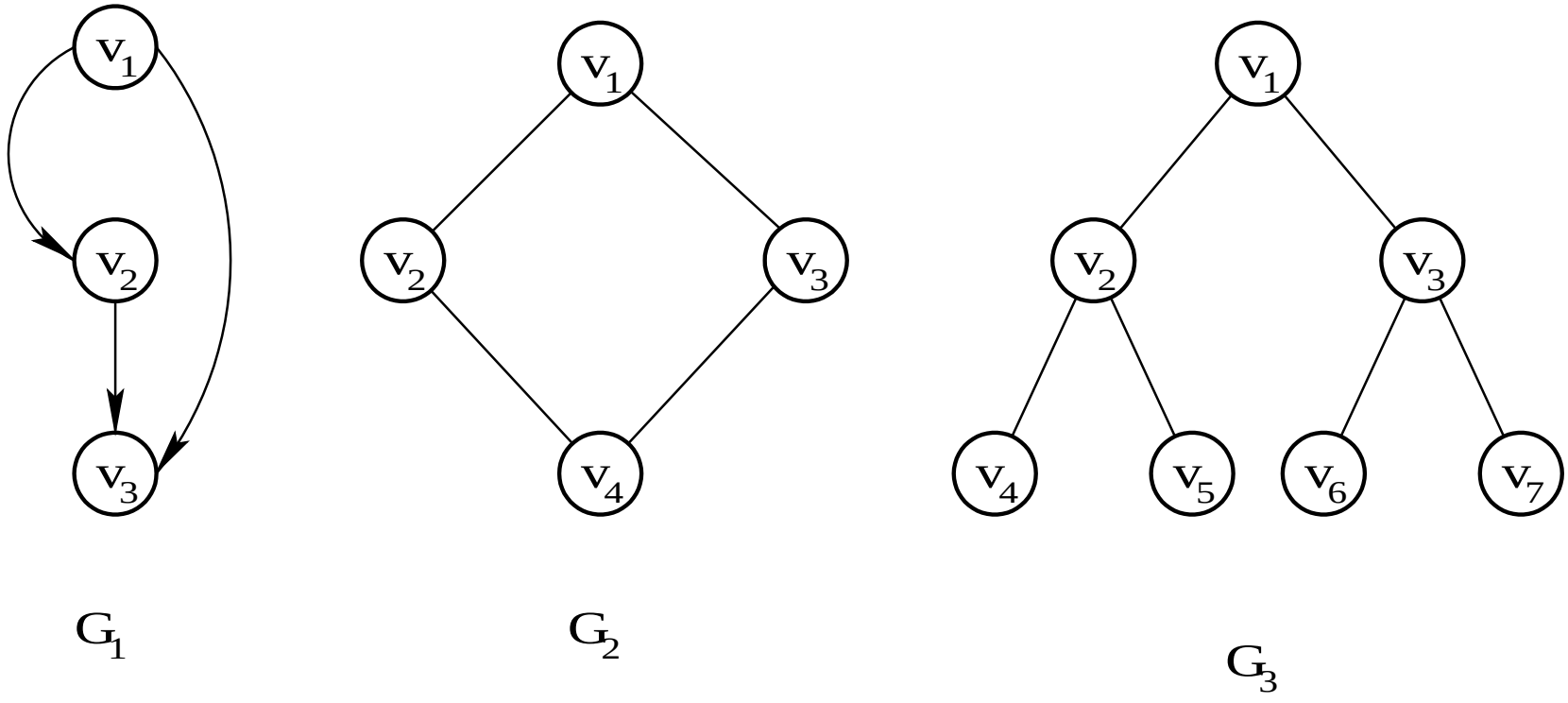


图7.1 图的示例

无向图(Undigraph)：如果图中每条边都是顶点无序对，则称此图为无向图。无向边用圆括号括起的两个相关顶点来表示。所以在无向图中，$(v_1\ v_2)$和$(v_2\ v_1)$表示的是同一条边。

如图7.1中的G_2和G_3都是无向图。其中

$V(G_2)=\{v_1,\ v_2,\ v_3,\ v_4\}$

$E(G_2)=\{(v_1,\ v_2),\ (v_1,\ v_3),\ (v_2,\ v_3),\ (v_2,\ v_4),\ (v_3,\ v_4)\}$

$V(G_3)=\{v_1,\ v_2,\ v_3,\ v_4,\ v_5,\ v_{16},\ v_7\}$

$E(G_3)=\{(v_1,\ v_2),\ (v_1,\ v_3),\ (v_2,\ v_4),\ (v_2,\ v_5),\ (v_3,\ v_6),\ (v_3,\ v_7)\}$

无向完全图(Completed Undigraph)和有向完全图(Completed Digraph)：若一个无向图有n个顶点，而每一个顶点与其他n－1个顶点之间都有边，这样的图称之为无向完全图，即共有n(n－1)/2条边。类似地，在有 n 个顶点的有向图中，若有n(n－1)条弧，即任意两点顶点之间都有双向相反的两条弧连接，则称此图为有向完全图。

子图(Subgraph)：设有两个图A和B且满足条件

V(B)V(A)

E(B)E(A)

则称B是A的子图。

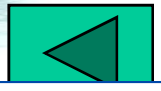
路径(Path)：在图G中，从顶点v_p到v_q的一条路径是顶点序列(v_p，v_{i1}，v_{i2}，…，v_{in}，v_q)且(v_p，v_{i1})，(v_{i1}，v_{i2})，…，(v_{in}，v_q)是E(G)中的边，路径上边的数目称之为该路径长度。对于有向图，其路径也是有向的，路径由弧组成。

简单路径(Simple Path)：如果一条路径上所有顶点除起始点和终止点外彼此都是不同的，则称该路径是简单路径。

回路(Cycle)和简单回路：在一条路径中，如果起始点和终止点是同一顶点，则称其为回路。简单路径相应的回路称为简单回路。

连通图(Connected Graph)和强连通图(Strongly Connected Graph)：在无向图G中，若从v_i到v_j有路径，则称v_i和v_j是连通的。若G中任意两顶点都是连通的，则称G是连通图。对于有向图而言，若G中每一对不同顶点v_i和v_j之间都有v_i到v_j和v_j到v_i的路径，则称G为强连通图。

度(Degree)、入度(Indegree)和出度(Outdegree)：若$(v_i\ v_j)$是E(G)中的一条边，则称顶点v_i和v_j是邻接的，并称边$(v_i\ v_j)$依附于顶点v_i和v_j。所谓顶点的度，就是依附于该顶点的边数。在有向图中，以某顶点为头，即终止于该顶点的弧的数目称为该顶点的入度；以某顶点为尾，即起始于该顶点的弧的数目称为该顶点的出度。该顶点的入度和出度之和称为该顶点的度。



7.2 图的存储结构

7.2.1 邻接矩阵

图的邻接矩阵表示法是用一个二维数组来表示图中顶点的相邻关系。设图G=(V，E)有n(n≥1)个顶点，则G的邻接矩阵是按如下定义的n阶方阵：

$$\text{cost}[i][j]=\begin{cases}1 & \text{若}(v_i，v_j)\text{或}<v_i，v_j>\in E(G)\\ 0 & \text{反之}\end{cases}$$

例如，图7.1中G_1，G_2，G_3的邻接矩阵分别表示为B_1，B_2和B_3，矩阵的行列号对应于图中结点的序号。

$$B_1=\begin{bmatrix}0&1&1\\0&0&1\\0&0&0\end{bmatrix}$$

$$B_2=\begin{bmatrix}0&1&1&0\\1&0&1&1\\1&1&0&1\\0&1&1&0\end{bmatrix}$$

$$B_3=\begin{bmatrix}0&1&1&0&0&0&0\\1&0&0&1&1&0&0\\1&0&0&0&0&1&1\\0&1&0&0&0&0&0\\0&1&0&0&0&0&0\\0&0&1&0&0&0&0\\0&0&1&0&0&0&0\end{bmatrix}$$

不难看出，无向图的邻接矩阵是对称的，而有向图的邻矩阵不一定对称。

在C语言中，图的邻接矩阵存储表示如下。

```
#define MAXVERTEXNUM 50
int cost[][MAXVERTEXNUM]
```

根据邻接矩阵的定义，可以得出邻接矩阵建立的算法如下：

```
/*算法描述7.1 邻接矩阵的建立*/
int creatadjmatrix (int cost[][MAXVERTEXNUM])  /*cost二维数组为建立的邻接矩阵*/
{int vexnum，arcnum，i，j，k，v1，v2；
 scanf ("%d，%d"，&vexnum，&arcnum)；
 /*输入图的顶点数和弧数或二倍边数*/
 for (i=0；i<vexnum；i++)
```

```
for (j=0；i<vexnum；j++)
cost[i][j]=0；/*给矩阵各元素赋初值0*/
for(k=1；k<=arcnum；k++)
{printf ("v1,v2=");
 scanf ("%d %d"，&v1，&v2);
 /* 输入所有边或所有弧的一对顶点v1,v2(无向图还要键入v2，v1)*/
cost[v1][v2]=1；/*cost[v1][v2]=1*/
}
 return(vexnum) ;
```

7.2.2 邻接链表

图的邻接链表存储结构是一种顺序分配和链式分配相结合的存储结构。它包括两个部分，一部分是向量，另一部分是链表。

原则上，在链表部分共有n个链表，即每个顶点一个链表。每个链表由一个表头结点和若干个表结点组成。表头结点用来指示第i个顶点v_i是否被访问过和所对应的链表指针；表结点由顶点域(Vertex)和链域(Next)所组成。顶点域指示了与v_i相邻接的顶点的序号，所以一个表结点实际代表了一条依附于v_i的边；链域指示了依附于v_i的另一条边的表结点。因此，第i个链表就表示了依附于v_i的所有边。对有向图来讲，第i个链表就表示了从v_i发出的所有弧。

邻接链表中的表头部分是向量，用来存储n个表头结点。向量的下标指示了顶点的 序号。

例如，对于图7.1中的G_1和G_2，其邻接链表分别如图7.2(a)、(b)所示。

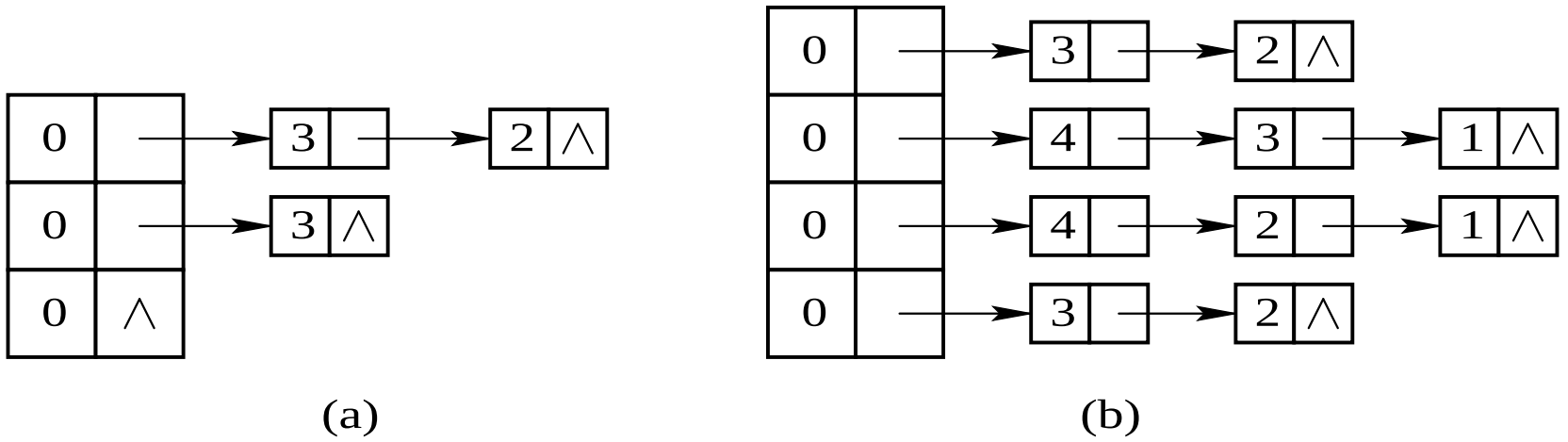


图7.2 邻接链表

在C语言中，图的邻接表存储表示如下:

```
#define MAXVERTEXNUM 50
 typedef struct arcnode
 {int vextex;
 struct arcnode *next;
 }arcnode;
  typedef struct vexnode
 {int data;
  arcnode  *firstarc;
 }vexnode, list[MAXVERTEXNUM];
```

由此可以得到建立邻接表的算法如下：

```
/*算法描述7.2  建立邻接表*/
int creatadjlist(vexnode list)                    /*建立邻接链表*/
{  arcnode *ptr;
   int vexnum, arcnum，k，v1，v2；
   printf("total vexnum、arcnum");
   scanf(“%d”，&vexnum，&arcnum)；                /*输入图的顶点数和
                                                  弧数或二倍边数*/
   for (k=0；k<vexnum；k++)
    list [k].firstarc=NULL；                      /*为邻接表的list数组
                                                  链域赋初值*/
```

```
    for(k=0；k<=arcnum；k++)      /*循环为list数组的各元素分
                                  别建立各自的链表*/
    {printf("v1，v2=");
     scanf ("%d %d"，&v1，&v2);
     ptr=(arcnode*)malloc(sizeof(arcnode));
     /*给结点v1的相邻结点v2分配内存空间*/
     ptr->vextex=v2；
     ptr->next=list[v1].firstarc；
     list [v1].firstarc=ptr；
     /*将相邻结点v2插入表头结点v1之后以形成链表*/
    }
    return(vexnum);
}
```

7.3 遍 历 图

给定一个无向连通图G，G=(V，E)，当从V(G)中的任一顶点V出发，去访问图中其余顶点，使每个顶点仅被访问一次，这个过程叫做图的遍历。

图的遍历和树的遍历相似，但要比树的遍历复杂得多。因为图中的任一顶点都可能和其余的顶点相邻接，所以在访问某个顶点后，可能沿着某条路径搜索之后，又回到该顶点。例如，由于图7.3中存在回路，因此在访问了v_1 v_2 v_4 v_3之后沿着边(v_3，v_1)又可访问到v_1。为了避免同一顶点被访问多次，在遍历图的过程中，必须记下每个已访问过的顶点。为此，我们设一个辅助数组，可以利用表头向量list，让其data域作为标志域，即

List[v_i].data=1　已访问过

List[v_i].data=0　未访问过

通常有两种遍历图的方法，一种是深度优先搜索法，另一种为广度优先搜索法。

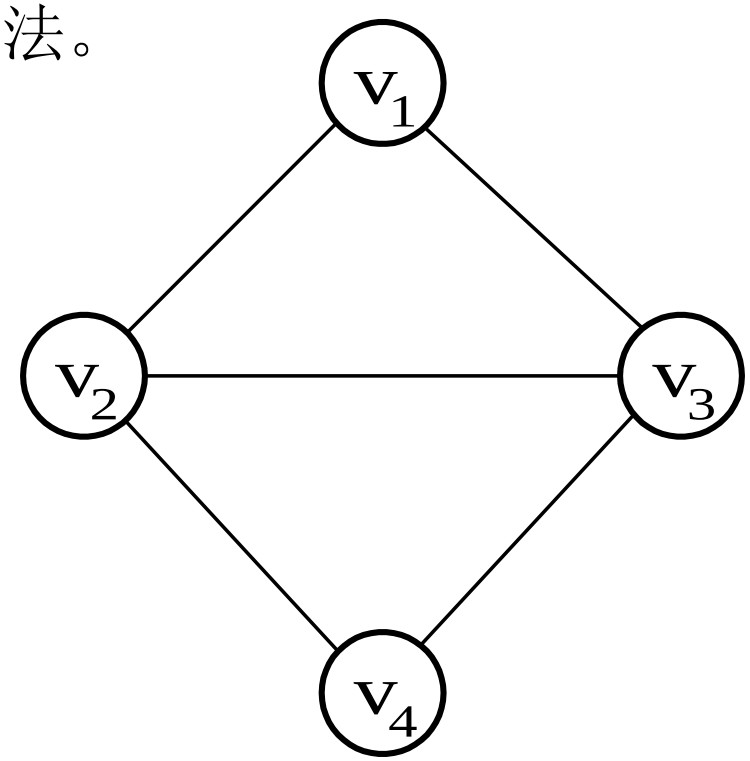


图7.3 图遍历示例

7.3.1 深度优先搜索法

设有无向连通图G(V，E)，从V(G)中任一顶点V出发深度优先搜索法进行遍历的步骤是：

先访问指定顶点v_1，然后访问与该顶点v_1相邻的顶点v_2，再从v_2出发，访问与v_2相邻且未被访问过的任意顶点v_3，然后从顶点v_3出发，重复上述过程，直到所有邻接点都被访问过为止，然后回溯到此顶点的直接前驱，从这儿出发再继续访问。显然搜索是一个递归过程。

对于图7.1中的G_2，邻接表为图7.2(b)，先选取v_1顶点作为搜索的起始点。顶点v_1的相邻顶点分别是v_3、v_2，所以沿着它的一个相邻顶点往下走。当走到顶点v_3时，它有三个相邻顶点v_4、v_2、v_1，沿着它的一个相邻顶点往下走到v_4，而v_4有两个相邻顶点v_3和v_2，因v_3已被访问过所以应原路返回，访问v_2，当走到v_2时，v_2有三个相邻顶点v_4、v_3、v_1，因v_4、v_3、v_1已被访问，原路返回上层，上层为空再返回，v_3、v_1被访问过，再返回上层，v_2被访问过，为空则结束，如图7.4所示。

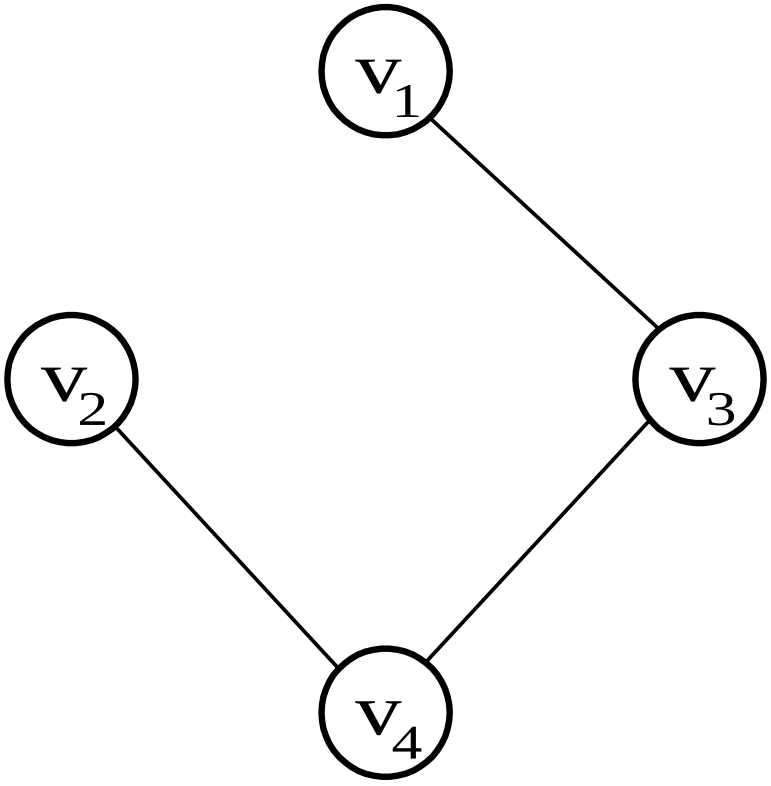


图7.4 深度优先搜索示意图

下面以邻接表为图的存储结构给出深度优先搜索算法。

```
/*算法描述7.3  深度优先搜索*/
vexnode list[MAXVERTEXNUM],*ptr[MAXVERTEXNUM];
int n;
      main()
{int v;
n=creatadjlist(list);                    /*建立邻接表，返回结点数*/
for(v=1;v<=n;v++)                        /*初始化指针数组及标志域*/
    {ptr [v]=list[v].firstarc;
     list[v].data=0;
     }
```

```
for(v=1;v<=n;v++)
if (list[v].data==0)
dfs(v);
}
dfs(int v)
   /*从某顶点v出发深度优先搜索子函数 */
   { int w;
     printf ("%d",v);                    /*输出顶点*/
     List [v].data=1;                    /*顶点标志域置1*/
           while  (ptr[v]!=NULL)
```

```
{ w=ptr[v]->vextex;          /*取出顶点v的某相邻顶点的序号*/
    if (list [w].data==0) dfs(w);
    /*如果该顶点未被访问过则递归调用，从该顶点w出发，沿着它的各相邻顶点向下*/
    /*搜索*/
    ptr[v]=ptr[v]->next;
    /*若从顶点v出发沿着某个相邻顶点的向下搜索已走到头，则换一个相邻顶点，沿着*/
  /*它往下搜索*/
    } /*从顶点v出发对各相邻顶点逐个搜索，直至从顶点v出发的所有并行路线已被搜索*/
}
```

7.3.2 广度优先搜索法

设无向图G(V，E)是连通的，从V(G)中的任一顶点v_1出发按广度优先搜索法遍历图的步骤是：

首先访问指定的起始顶点v_1，从v_1出发，依次访问与v_1相邻的未被访问过的顶点w_1，w_2，w_3，…，w_t，然后依次从w_1，w_2，w_3，…，w_t出发，重复上述访问过程，直到所有顶点都被访问过为止。以图7.5为例，如选顶点v_1为起始点先进行访问。顶点有三个相邻顶点，依次是v_4、v_3、v_2，则广度优先搜索时对这几个顶点依次访问，然后再将这些相邻顶点的第一相邻顶点v_4作为起始顶点重复上述步骤，再优先访问v_6。再将第二个相邻顶点v_3作为起始顶点，重复上述步骤，顶点v_3有三个相邻顶点依次为v_6、v_5、v_2，其中v_2、v_6被访问过(不再访问)，所以访问v_5。依次再判v_2、v_6、v_5，重复上述步骤，因相邻顶点都被访问过即结束，如图7.5所示。

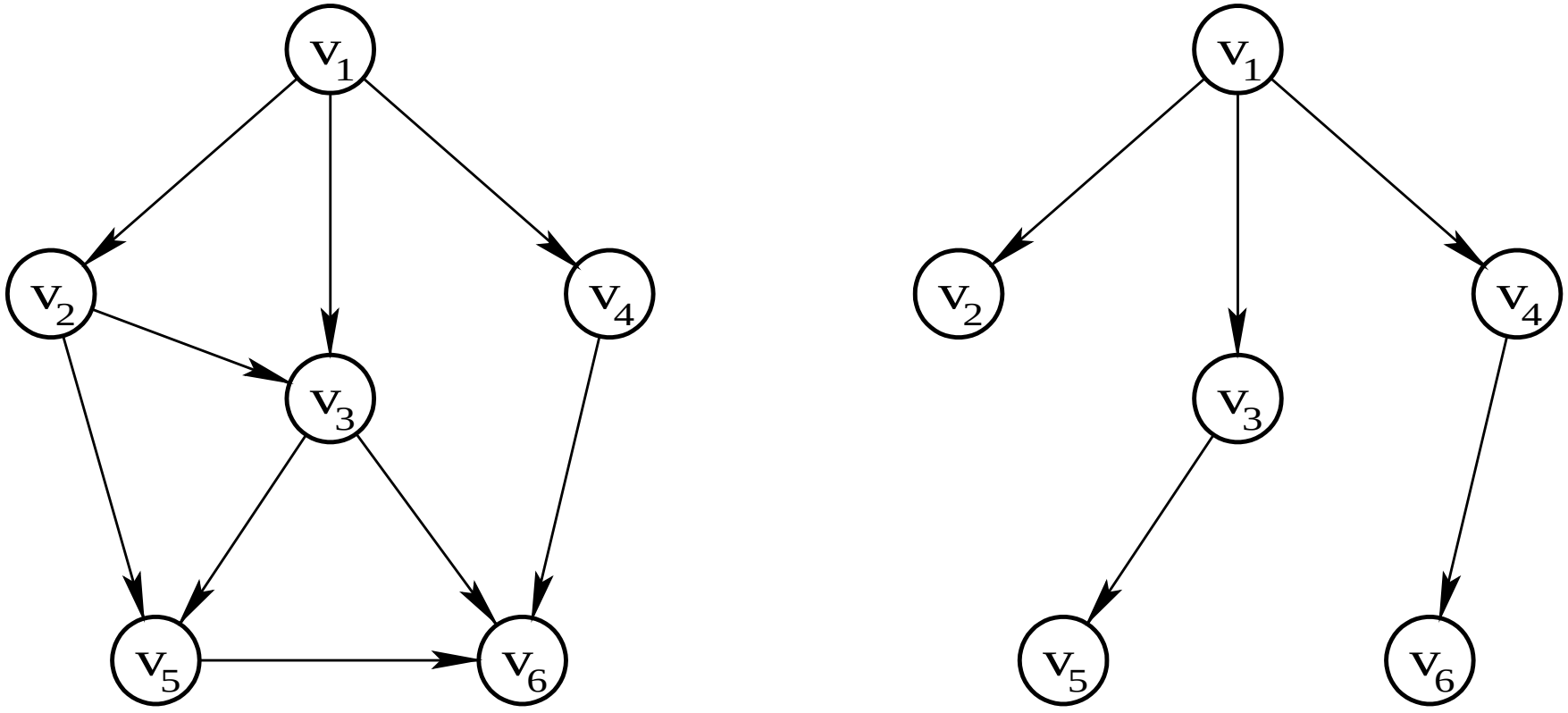


图7.5 广度优先搜索示意图

下面以邻接表为图的存储结构给出广度优先搜索算法。

```
/*算法描述7.4 广度优先搜索*/
  vexnode list[MAXVERTEXNUM];
  int n;
  main()
{int v;
n=creatadjlist(list);                    /*建立邻接表，返回结点数*/
for(v=1;v<=n;v++)                        /*初始化标志域*/
list[v].data=0;
for(v=1;v<=n;v++)
if (list[v].data==0)
bfs(v);
}
```

```
void bfs(int v)                    /*从v出发广度优先搜索函数*/
    {arcnode * ptr;
     int v1， w;
     printf ("%d"v);               /*输出该顶点*/
     list [v].data=1;              /*标志域置1*/
     enqueue(v);                   /*将该顶点入队尾*/
     while((v1=dequeue())!=EOF)
     /*while循环使属于同一层顶点的相邻顶点的依次出队。由于这些相邻顶点作为上一*/
/*层顶点均被访问过，出队的目的是访问它的下层相邻顶点*/
     { ptr=list[v1].firstarc；/*取出该顶点的第一个相邻顶点地址*/
```

```
        while (ptr!=NULL)     /*while循环依次访问各相邻顶点*/
         {w=ptr->vertex;      /*取出该顶点的序号*/
          ptr=ptr->next;
          /*从邻接表的相应链表中，取出下一个相邻顶点的地址，
以备访问*/
          if (list[w].data==0)  /*若该相邻顶点未被访问过*/
           {printf("%d",w)     /*则访问该相邻顶点*/
            list [w].data=1;    /*修改顶点标志域为1*/
            enqueue (w);        /*将访问过的顶点入队，以备在进入
下一层搜索时使用*/
             }
          }
        }
}
```

7.4 最短路径

我们先举一个例子来说明什么是最短路径。如果我们用顶点表示城市，用边表示城市之间的公路，则由这些顶点和边组成的图可以表示沟通各城市的公路网。若把两个城市之间的距离作为权值，赋给图中的边，就构成了带权图。

对于一个汽车司机来说，他一般关心两个问题：

(1) 从甲地到乙地是否有公路?

(2) 从甲地到乙地有多条公路可以到达时，那么，哪条公路路径最短或花费代价最小？

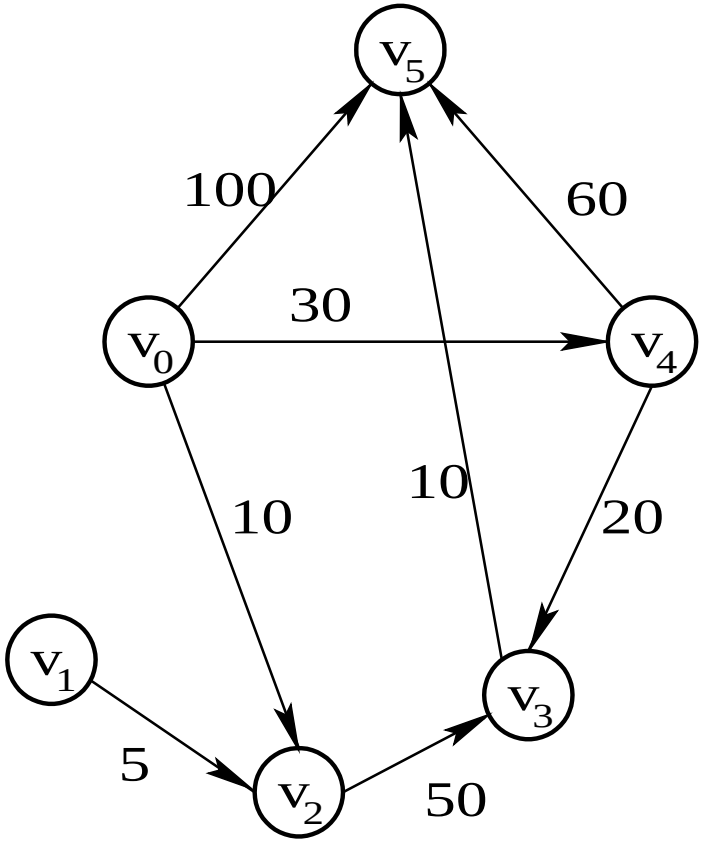
这就是本节要讨论的最短路径问题。这里所谓的最短路径，是指所经过的边的权值之和为最小的路径，而不是经过边的数目最少。考虑到公路的有向性，在讨论中结合有向带权图来进行。设定边的权值为正值，并将路径的开始顶点称为源点，路径的最后一个顶点称为终点。本节给出两个算法，一个是求从某个源点到其他顶点的最短路径，另一个是求每对顶点间的最短路径。

7.4.1 从某个源点到其他各顶点的最短路径

设有向带权图G=(V，E)，我们用cost[][]表示图G的邻接矩阵，其中

$$cost[i][j]=\begin{cases} w & \text{若}<v_i, v_j>\in E(G)，\ w\text{为边权值} \\ \infty & \text{反之} \end{cases}$$

例如，图7.6(a)所示带权图的邻接矩阵如图7.6(b)所示。



$$\begin{bmatrix} \infty & \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 5 & \infty & \infty & \infty \\ \infty & \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & \infty & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(a) (b)

图7.6 有向带权图及其邻接矩阵
(a) 有向带权图；(b) 带权邻接矩阵

对于这样的存储结构，如何能较方便地在计算机上求得最短路径呢?迪杰斯特拉(Dijkstra)提出了按路径长度递增的次序产生最短路径的算法。此算法把网中所有顶点分成两个集合。凡以v_0为源点已确定了最短路径的终点并入S集合，S集合的初态只包含v_0；另一个集合V－S为尚未确定最短路径的顶点的集合。按各顶点与v_0间的最短路径长度递增的次序，逐个把V－S集合中的顶点加入到S集合中去，使得从v_0到S集合中各顶点的路径长度始终不大于从v_0到V－S集合中各顶点的路径长度。为了能方便地求出从v_0到V－S集合中最短路径的递增次序，算法中引入一个辅助向量dist[]。它的某一分量dist[i]，表示当前求出的从v_0到v_i的最短路径长度。这个路径长度不一定是真正的路径长度。它的初始状态即是邻接矩阵cost[][]中v_0行内各列的值，显然，从v_0到各顶点的最短路径中最短的一条路径长度应为

$$dist[w]=mindist[i]/v_i \in V(G)$$

第一次求得的这条最短路径必然是<v_0，w>，这时顶点w应从V－S中删除而并入S集合中。每当选出一个顶点w并使之并入S集合之后，修改V－S集合中各顶点的最短路径长度dist。对于V－S集合中的某一顶点v_i来说，其当前的最短路径或者是<v_0，v_i>或者是<v_0，w，v_i>，而决不可能有其他选择。也就是说

如果 dist[w]+cost[w][v_i]<dist[i]

则 dist[i]=dist[w]+cost[w][v_i]

当V－S集合中各顶点的dist进行修改后，再从中挑选一个路径长度最小的顶点，从V－S中删除，并入S中，依次类推，就能求出到各顶点的最短路径长度。

对于带权邻接矩阵求单源最短路径的算法如下：

```
/*算法描述7.5  单源最短路径*/

dijkstra (int cost[][MAX]，int n)  /*求单源最短路径*/

{int s [MAX]，dist [MAX];

/*s[]用来表示顶点集合S，即S集合中的顶点是从源点v0出发到它们的最短路径已求出*/

/*的顶点。而dist[]用来记录从源点v0出发到各顶点的最短距离*/

int i，j，v，w，sum,v0 ;
```

```
        printf("%s","v0 ");
        scanf("%d",&v0 );
        for (i=0；i<=n-1；i++)
          {dist [i]=cost [v0 ][i];
           s[i]=0；
    }
    /*初始化dist[]和s[]，开始时，设所有的终点都不在S集合中。
    这里的i是顶点的序号*/
    /*0是一种状态，表示vi不在S集合中*/
        S[v0 ]=1；                    /*开始时，只有源点v0在集合中*/
        printf("choose vertex set|distance")；
```

```
trace (s，dist，n)  /*调用子程序，显示源点v0到各顶点的距离*/
for (i=0；i<=n-2；i++)
    {w=minicost (dist，s，n)；      /*调用函数从S集外找出距离
                                    源点v0最近的顶点w */
     s[w]=1；                       /*将顶点w加入S集中，即w成为已
                                    求出最短距离的顶点*/
     for (v=1；v<=n-1；v++)
     if (s[v]==0)
      {
      sum=dist[w]+cos[w][v]；       /*sum为从顶点v0出发，经过
                                    顶点w到达终点v的距离*/
```

```
            dis t[v]=(dist[v]<sum)?dist[v]：sum；          /*从原来到达
终点v的距离和现在经过w的距离*/
/*中选取一个较短的距离，作为当前的最短距离*/
                }/*若顶点不在S集合中，则调整顶点v距源点的距离*/
          trace (s，dist，n);                                  /*再次显示v0
到达各顶点的距离*/
        }
      putdist (dist，n)；   /*显示从源点v0到达各顶点的最短距离*/
 }
```

求S集合外找出距源点v_0最近的顶点的算法如下：

```
/* 算法描述7.6*/
int minicost (dist，s，n)  /*从S集合外找出距离源点v0最近的顶点的
                             子函数*/
    int dist []，s[]，n;
    {int i，tmp=9999，w=1;        /*tmp=9999为假定最大数*/
     for (i=1；i<=n-1 i++)
     if ((s[i]==0)&&(dist[i]<tmp))
       {tmp=dist [i];
        w=i;
       }  /*若顶点vi不在S集合中而且距源点v0的距离小于tmp，则改变tmp的
值，而且*/
          /*将顶点vi当作距离v0最近的顶点*/
   return(w);
}
```

显示最后结果的函数定义如下：

```
/*算法描述7.7*/
putdist (dist，n)
    int dist []，n；
      {int i;
        printf("the shortest path from v0  to each vertex：")
        for  (i=1；i<=n-1；i++)
        printf("%d：%d"，i，dist [i]);
      /*在for循环中显示各结点的序号和距源点v0的距离*/
        printf(" ");
      }
```

显示从源点v_0到达各顶点距离的函数定义如下：

```
/*算法描述7.8*/
trace (s，dist，n)
    int s[]，dist[]，n；
     {int  j；
      for (j=0；j<n-1；j++)
      if (s[j]==1) printf("%-4d"，j)；
       else printf("%-4d"，s[j])/*利用for循环显示S集合中顶点的序号*/
       printf("|");
       for (j=1；  j<=n-1；j++)
       printf ("%8d"，dist[j])；/*利用for循环显示当前为止各顶点距源点v0的距离*/
       printf (" ");
     }
```

7.4.2 求每一对顶点之间的最短路径

解决这个问题的一个办法是：每次以一个顶点为源点，重复执行迪杰斯特拉算法n次。这样，便可求得每一对顶点之间的最短路径。总的执行时间为$O(n^3)$。

这里介绍弗洛伊德(Floyd)提出的另一个算法。这个算法的时间复杂度也是$O(n^3)$，但形式更简单。

弗洛伊德算法仍从图的带权邻接矩阵cost出发，其基本思想是：

在算法中设立两个矩阵用来分别记录各顶点间的路径和相应的路径长度。矩阵P表示路径，矩阵A用来表示路径长度。

我们先讨论如何求得各顶点间的最短路径长度，初始时，复制网的代价矩阵cost为矩阵A的值，即顶点v_i到顶点v_j的最短路径长度A[i][j]就是弧<v_i，v_j>所对应的权值(若<v_i，v_j>不存在，则A[i][j]为∞)，我们不妨记为$A^{(-1)}$，$A^{(-1)}$的值不可能是最短路径长度。

如何求得最短路径?要进行n次试探。对于从顶点v_i到顶点v_j的最短路径长度，首先考虑让路径经过顶点v_0，比较路径<v_i，v_j>和<v_i，v_0，v_j>的长度，取其短者为当前求得的最短路径。对每一对顶点都作这样的试探，可求得$A^{(0)}$。然后，再考虑在$A^{(0)}$的基础上让路径经过顶点v_1，于是求得$A^{(1)}$。依次类推，一般地，如果从顶点v_i到顶点v_j的路径经过新顶点v_k能使路径缩短，则修改$A^{(k)}[i][j]=A^{(k-1)}[i][k]+A^{(k-1)}[k][j]$，所以，$A^{(k)}[i][j]$就是当前求得的从顶点$v_i$到顶点$v_j$的最短路径长度，且其路径上的顶点(除源点、终点外)序号均不大于k。

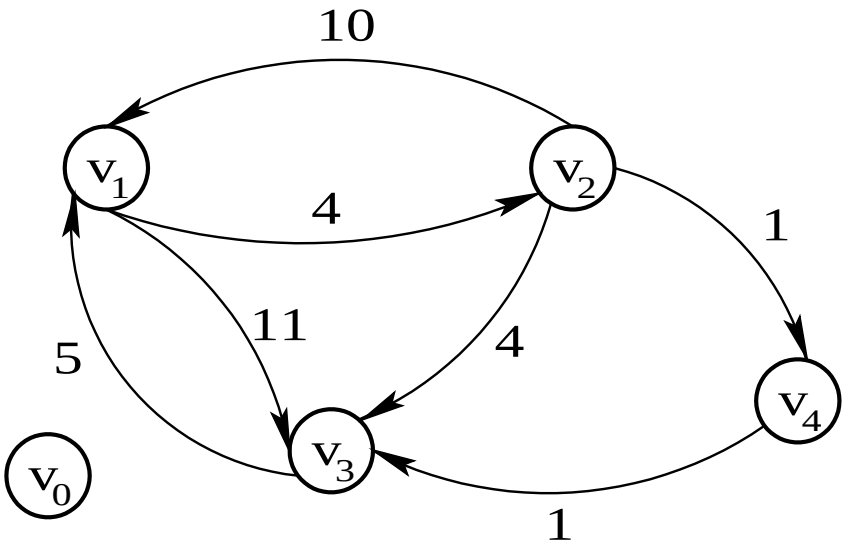
这样经过几次试探，就把几个顶点都考虑到相应的路径中去了。最后求得的$A^{(n-1)}$就一定是各顶点间的最短路径长度。综上所述，弗洛伊德算法的基本思想是递推地产生两个几阶的矩阵序列。其中，表示最短路径长度的矩阵序列是$A^{(-1)}$，$A^{(0)}$，$A^{(1)}$，$A^{(2)}$，…，$A^{(k)}$，…，$A^{(n-1)}$，其递推关系是

$A^{(-1)}[i][j]=cost[i][j]$

$A^{(k)}[i][j]=min\{A^{(k-1)}[i][j]，A^{(k-1)}[i][k]+A^{(k-1)}[k][j]\}$ $(i\geq 0，j\geq 0，k\leq n-1)$

现在我们再讨论如何求解最短路径长度的同时求解最短路径?初始时矩阵P的各元素都赋零。P[i][j]=0表示v_i到v_j的路径是直接到达，中间不经过其他顶点。以后，当考虑路径经过某个顶点v_k时，如果使路径更短，则修改$A^{(k-1)}$[i][j]的同时令P[i][j]=k，即P[i][j]中存放的是从v_i到v_j的路径上所经过的某个顶点(若P[i][j]≠0)。那么，如何求得从v_i到v_j的路径上的全部顶点呢? 这只需要编写一个递归过程即可解决，因为所有最短路径的信息都包含在矩阵P中了。设经过n次试探后，P[i][j]=k，即从v_i到v_j的最短路径经过顶点v_k(若k≠0)。该路径上还有哪些顶点呢? 只需去查P[i][k]和P[k][j]即可。依次类推，直到所查元素为零。

对于图7.7所示的有向带权图G，按照弗洛伊德算法由递推产生的两个矩阵序列如图7.8所示。



$$\begin{bmatrix} 0 & \infty & \infty & \infty & \infty \\ \infty & 0 & 4 & 11 & \infty \\ \infty & 10 & 0 & 4 & 1 \\ \infty & 5 & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & 0 \end{bmatrix}$$

图7.7 网G和它的邻接矩阵

$$A^{(-1)}=\begin{bmatrix}0&\infty&\infty&\infty&\infty\\\infty&0&4&11&\infty\\\infty&10&0&4&1\\\infty&5&\infty&0&\infty\\\infty&\infty&\infty&1&0\end{bmatrix}\qquad p^{(-1)}=\begin{bmatrix}0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\end{bmatrix}$$

$$A^{(0)}=\begin{bmatrix}0&\infty&\infty&\infty&\infty\\\infty&0&4&11&\infty\\\infty&10&0&4&1\\\infty&5&\infty&0&\infty\\\infty&\infty&\infty&1&0\end{bmatrix}\qquad p^{(0)}=\begin{bmatrix}0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\end{bmatrix}$$

图7.8 网G的各对顶点间最短路径及长度

$$A^{(1)}=\begin{bmatrix}0&\infty&\infty&\infty&\infty\\ \infty&0&4&11&\infty\\ \infty&10&0&4&1\\ \infty&5&9&0&10\\ \infty&\infty&\infty&1&0\end{bmatrix}\qquad p^{(1)}=\begin{bmatrix}0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&1&0&0\\ 0&0&0&0&0\end{bmatrix}$$

$$A^{(2)}=\begin{bmatrix}0&\infty&\infty&\infty&\infty\\ \infty&0&4&5&8\\ \infty&10&0&4&1\\ \infty&5&9&0&10\\ \infty&\infty&\infty&1&0\end{bmatrix}\qquad p^{(2)}=\begin{bmatrix}0&0&0&0&0\\ 0&0&0&2&2\\ 0&0&0&0&0\\ 0&0&1&0&2\\ 0&3&3&0&0\end{bmatrix}$$

图7.8 网G的各对顶点间最短路径及长度

$$A^{(3)}=\begin{bmatrix}0&\infty&\infty&\infty&\infty\\\infty&0&4&8&5\\\infty&9&0&4&1\\\infty&5&9&0&10\\\infty&6&10&1&0\end{bmatrix}\qquad p^{(3)}=\begin{bmatrix}0&0&0&0&0\\0&0&0&2&2\\0&3&0&0&0\\0&0&1&0&2\\0&3&3&0&0\end{bmatrix}$$

$$A^{(4)}=\begin{bmatrix}0&\infty&\infty&\infty&\infty\\\infty&0&4&6&5\\\infty&7&0&2&1\\\infty&5&9&0&10\\\infty&6&10&1&0\end{bmatrix}\qquad p^{(4)}=\begin{bmatrix}0&0&0&0&0\\0&0&0&4&2\\0&4&0&4&0\\0&0&1&0&2\\0&3&3&0&0\end{bmatrix}$$

图7.8 网G的各对顶点间最短路径及长度

由此可以得到如下的弗洛伊德算法描述：

```
/*算法描述7.9*/
void floyed (cost，a,p,n)
    int cost [][MAX]， a[][MAX]，p[][MAX]，n;
     /*a[][]表示最短路径长度，数组p[][]表示最短路径的数组*/
     {
     int i，j，k;
     for(i=0；i<n；i++)
       for(j=0；j<n；j++)
         {a[i][j]=cost[i][j];
          p[i][j]=0;
          }/*给A数组和P数组赋初值 */
```

```
for (k=0；k<n；k++)
        for (i=0；i<n；i++)
          for (j=0；j<n；j++)
            if (a[i][k]+a[k][j]<a[i][j])
              {a[i][j]=a[i][k]+a[k][j]；
               p[i][j]=k;
               }/*根据A(-1)[i][j]=cost[i][j]；A(k)[i][j]=minA(k-1)[i][j],A(k-1)[i][k]+A(k-1)[k][j]*/
                /*(i≥0,j≥0,k≤n-1)递推最短路径长度和路径*/
    }
```

7.5 拓扑排序

7.5.1 AOV网

在实际工作中，经常用有向图来表示工程的施工流程图或产品生产的流程图。一个工程一般可分为若干子工程，我们把子工程称为“活动”。在有向图中若以顶点表示“活动”，用有向边表示“活动”之间的优先关系，则这样的有向图称为以顶点表示“活动”的网(Activity On Vertex Network)，简称AOV网。

AOV网中的弧表示了“活动”之间的优先关系，也可以说是一种制约关系。例如，计算机专业学生必须学完一系列规定的课程后才能毕业。这可看作一个工程，我们用图7.9所示的AOV网加以表示，网中的顶点表示各门课程的教学活动，有向边表示各门课程的制约关系。如图7.9中有一条弧<c_3, c_9>，其中c_3和c_9分别表示“普通物理”和“计算机组成原理”的教学活动，这说明“普通物理”是“计算机组成原理”的直接前驱，“普通物理”教学活动一定要安排在“计算组成原理”教学活动之前。

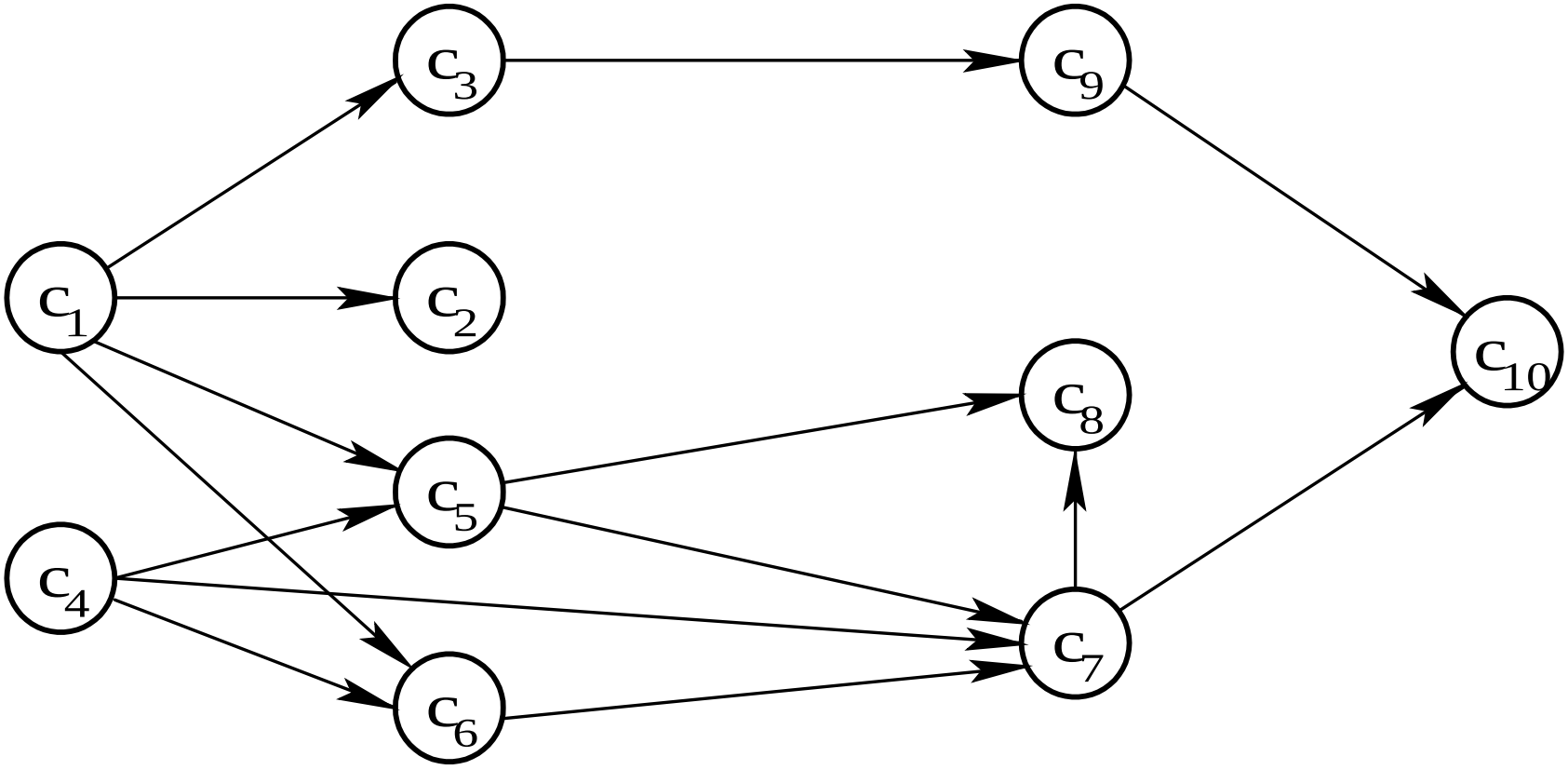


图7.9 表示课程之间优先关系的AOV网

课程代号	课程名称	先行课程
c_1	高等数学	无
c_2	工程数学	c_1
c_3	普通物理	c_1
c_4	程序设计基础	无
c_5	C语言程序设计	c_1 c_2 c_4
c_6	离散数学	c_1
c_7	数据结构	c_4 c_5 c_6
c_8	编译方法	c_5 c_7
c_9	计算机组成原理	c_3
c_{10}	操作系统	c_7 c_9

在图7.9中，顶点c_1，c_4是顶点c_5的直接前驱；顶点c_7是顶点c_4，c_5，c_6的直接后继；顶点c_1是顶点c_9的前驱，但不是直接前驱。显然，在AOV网中，由弧表示的优先关系有传递性，如顶点c_1是c_3的前驱，而c_3是c_9的前驱，则c_1也是c_9的前驱。在AOV网中不能出现有向回路，如果存在回路，则说明某个“活动”能否进行要以自身任务的完成作为先决条件，显然，这样的工程是无法完成的。如果要检测一个工程是否可行，首先就得检查对应的AOV网是否存在回路。检查AOV网中是否存在回路的方法就是拓扑排序。

7.5.2 拓扑排序

对于一个AOV网，构造其所有顶点的线性序列，使此序列不仅保持网中各顶点间原有的先后次序，而且使原来没有先后次序关系的顶点之间也建立起人为的先后关系，这样的序列称为拓扑有序序列。构造AOV网的拓扑有序序列的运算称为拓扑排序。

某个AOV网，如果它的拓扑有序序列被构造成功，则该网中不存在有向回路，其各子工程可按拓扑有序序列的次序进行安排。一个AOV网的拓扑有序序列并不是惟一的，例如，下面的两个序列都是图7.9所示AOV网的拓扑有序序列。

c_1 c_4 c_3 c_2 c_5 c_6 c_9 c_7 c_8 c_{10}

c_4 c_1 c_2 c_3 c_9 c_6 c_5 c_7 c_8 c_{10}

对AOV网进行拓扑排序的步骤是:

(1) 在网中选择一个没有前驱的顶点且输出之。

(2) 在网中删去该顶点，并且删去从该顶点发出的全部有向边。

(3) 重复上述两步，直到网中不存在没有前驱的顶点为止。

这样操作结果有两种：一种是网中全部顶点均被输出，说明网中不存在有向回路；另一种是网中顶点未被全部输出，剩余的顶点均有前驱顶点，说明网中存在有向回路。

拓扑排序的方法很多，主要有深度优先搜索排序和广度优先搜索排序两种，下面分别介绍。

1. 广度优先搜索拓扑排序

根据拓扑排序的方法，把入度为0的顶点插入一个队列，按顺序输出。本算法中将顶点的入度记录在邻接表数组的数据域中，即记录在list[v].data中。算法如下:

```
void topsort(vexnode list[])
{arcnode * ptr;
int v，  w，  n1=0;
for (v=1；  v<=n；  v++)
  if (list[v].data==0)
  enqueue (v);
```

```
/*利用循环检测入度为0的顶点并入队，即将无前驱的顶点加入队列*/
  while((v=dequeue())!=EOF)
   { printf("%-5d%c"，v，(++n1%10==0)? '/n':");
   /*显示无前驱顶点，即删除这些顶点，并计数*/
    ptr=list[v].firstarc;
   /*取上面被删除顶点所相邻顶点的地址，以便删除指向它的边*/
    while (ptr!=NULL)
     { w=ptr->vertex;                    /*取相邻顶点的序号*/
       if (--list[w].data==0)
          enqueue(w);
```

```
/*将相邻顶点的入度减1,即删除指向该相邻顶点的一条边。若该顶点的入度减1后，*/

/*入度为0，则成为无前驱顶点，所以入队列以便删除*/

        ptr=ptr->next;   /*取下一个相邻顶点的地址*/

        }

    }/*循环结构,无前驱的顶点逐个出队

if (n1<n)/*如果n1<n，则拓扑排序失败*/

printf("not a set of partial order");

}
```

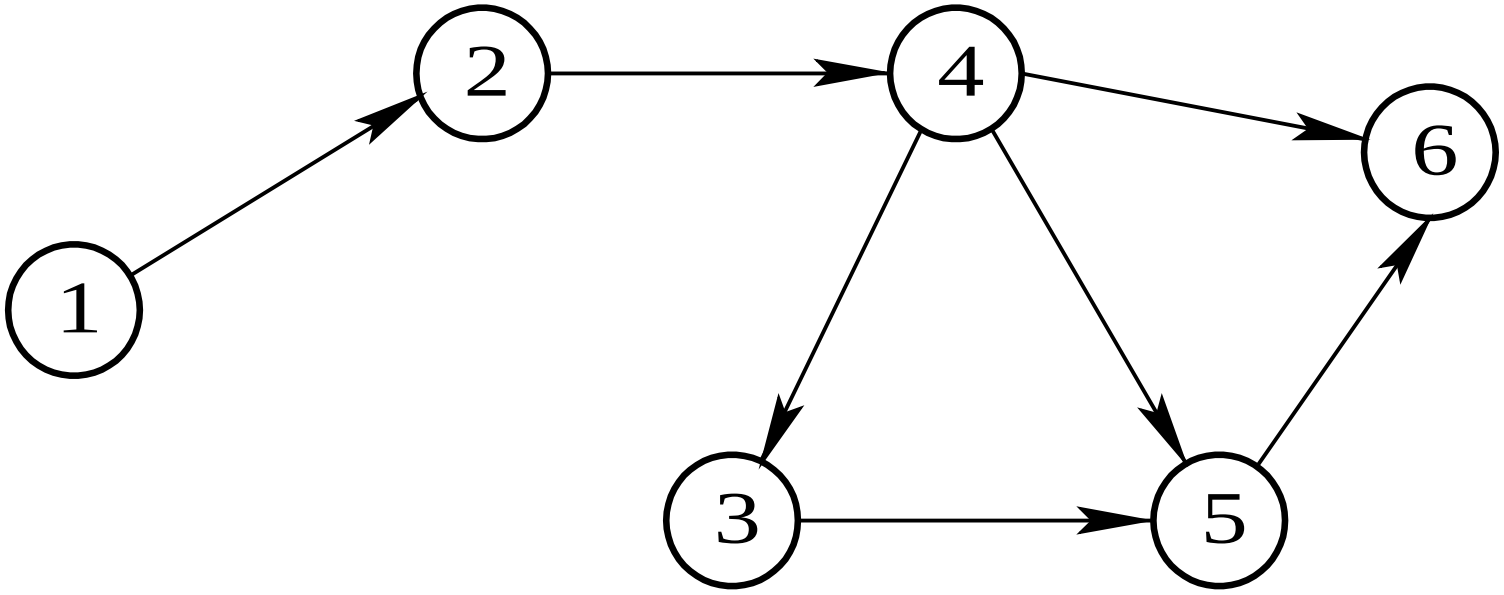


图7.10 拓扑排序

如图7.10的广度优先搜索拓扑排序序列为① ② ④ ③ ⑤ ⑥。

此算法中如果把enqueue(v)改为push(v)，把dequeae(v)改为pop(v)，则可得另外一种方式的拓扑序列。

2. 深度优先拓扑排序

根据拓扑排序的方法，先用深度优先搜索法向下走，直到无路可走为止。每走一步都伴随着顶点进栈。无路可走时出栈，并同时显示顶点序号。当退回一步后，换向再走,若无向可换，则出栈，即删除无前驱的顶点。算法如下：

```
topodfs(v)

    int v;

int w;
```

```
list[v].data=1;          /*对搜索过的顶点标志改为1，以免重复，并进栈*/
push(v);
while (ptr[v]!=NULL)
 w=ptr[v]->vertex;  /*则取出相邻顶序的序号，看相邻顶点是否搜索过*/
  if (list[w]．  data==0)  topodfs(w);
```

/*若相邻顶点未被搜索过，则递归调用拓扑排序函数，去深度优先搜索相邻顶点*/

```
   ptr[v]=ptr[v]->next;          /*取另一个相邻顶点*/
printf ("%5d"，  pop ());       /*按相反的拓扑序列显示拓朴序列的顶序*/
```

7.6 实习：最短路径的实现

用图表示出全国直辖市和省会城市的铁路网络，量出地图上直辖市和省会城市的距离，并输入计算机，要求能求出从任一城市出发到其他城市的最短路径并输出。

主程序如下:

```
#include <stdio.h>
main()
int cost[MAX][MAX],n;     /*cost[][]是带权邻接矩阵*/
 n=adjmatrix(cost);       /*调用子函数建立邻接矩阵*/
 prmatrix(cost,n);        /*调用子函数显示邻接矩阵*/
 dijkstra(cost,n);        /*调用子函数求单源最短路径*/
```

建立带权邻接矩阵的算法如下：

```
admatrix(matrix)
int matrix[][max];
int vexnum,arcnum,i,j v1,v2,weight;
/*v1,v2分别为边的两个顶点，weight是它们的权*/
printf ("total vexnum=");
scanf ("%d",&vexnum);                    /*键入矩阵的结点数*/
for (i=0 ;i<vexnum;i++)
 for (j=0 ; j<vexnum;j++)
   matrix[i][j]=32767;
   /*给矩阵各结点输初值*/
```

```
printf("total arcnum=");
scanf ("%d", &arcnum);
/*键入图的弧数*/
for (i=0;i<arcnum;i++)
printf("v1 v2 weight=");
  scanf ("%d,%d,%d",&v1,&v2,&weight);
  matrix[v1][v2]=weight;
return (vexnum);
```

邻接矩阵显示函数如下：

```
prmatrix(mat ,n)

int mat[][MAX],n;

 int i,j;

 for (i=0;i<n;i++)

  for (j=0;j<n;j++)

   printf((mat[i][j]==32767)?"":"%d",mat[i][j]);

   printf("");
```

其他算法参考第6章6.4节。

习 题 7

1. 图有哪两种常用的存储结构？它们是如何存储的？

2. 什么是图？什么是路径？什么是连通图和强连通图？

3. 对于给出的无向图7.11，求：

(1) 每个顶点的入度和出度；

(2) 图的邻接矩阵；

(3) 图的邻接链表。

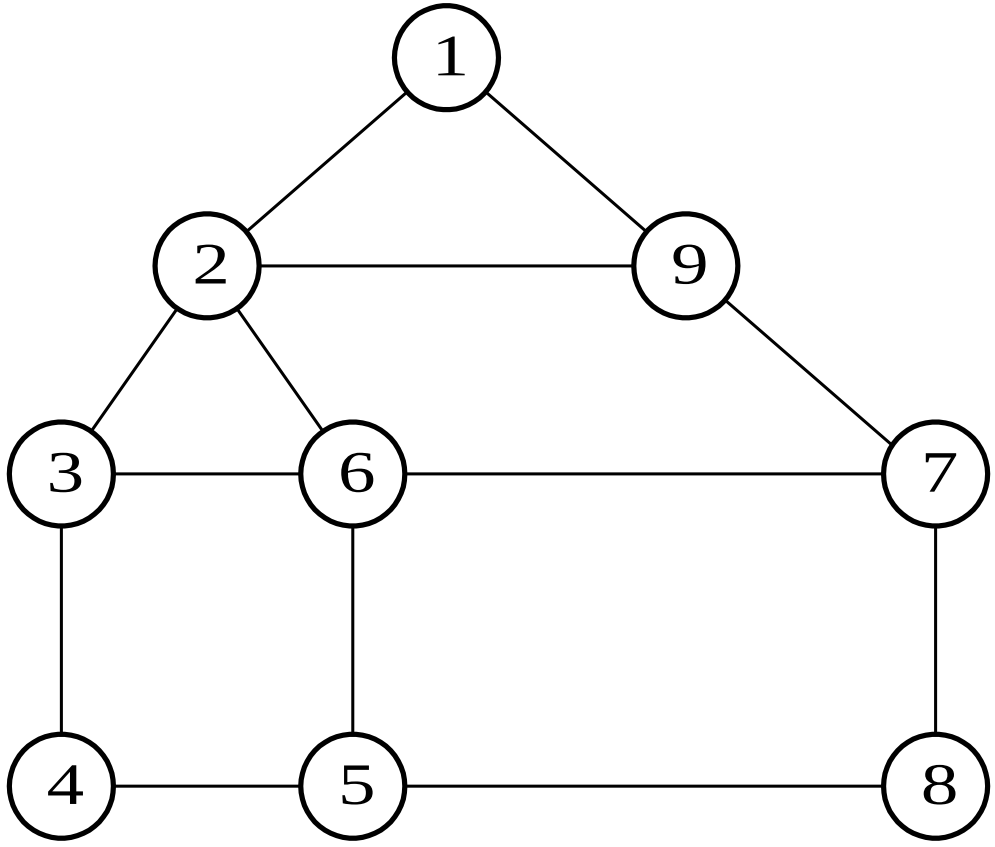


图7.11 题3图

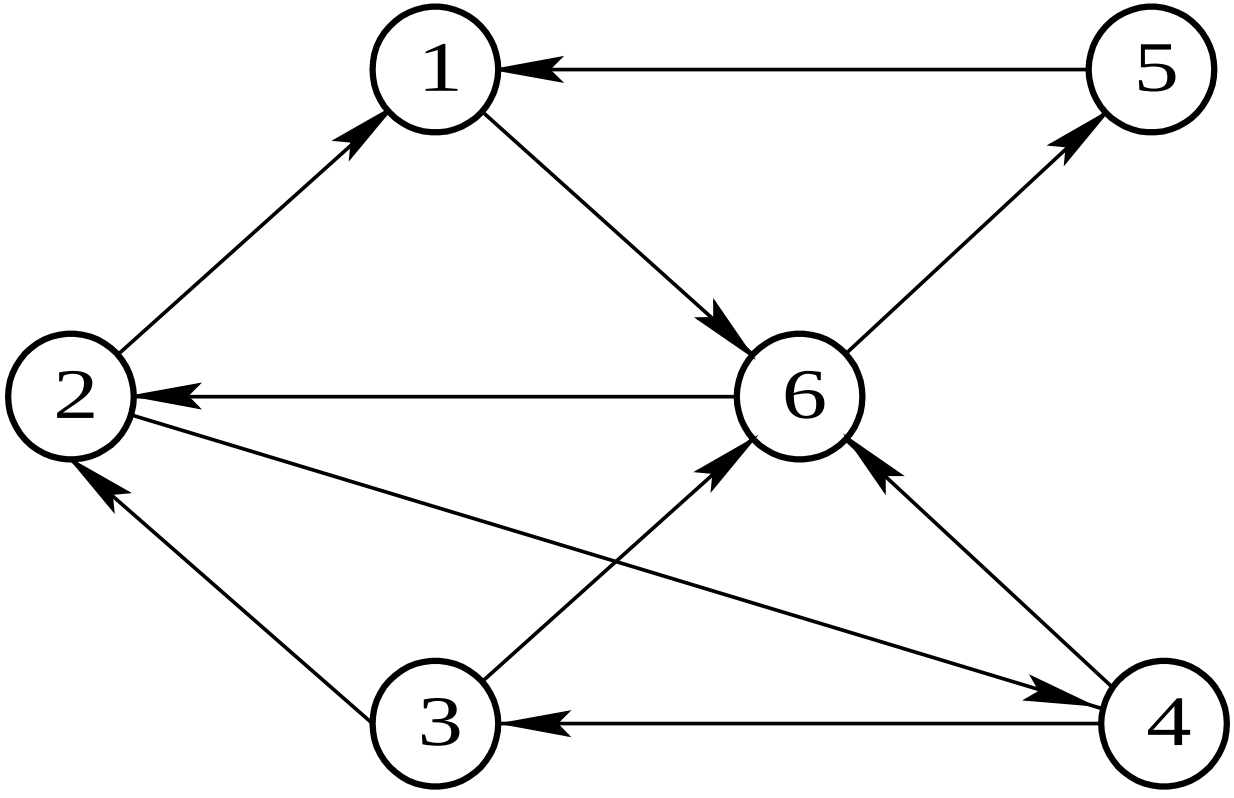


图7.12 题4图

4. 请写出对图7.12，按深度优先搜索和广度优先搜索从顶点①出发遍历图的结果。

5. 设用邻接矩阵来表示无向图，试用C语言写出从某一顶点出发，采用广度优先搜索遍历图的程序，并上机验证。

6. 什么是拓扑排序？对图7.13，试写出一种拓扑排序的序列。

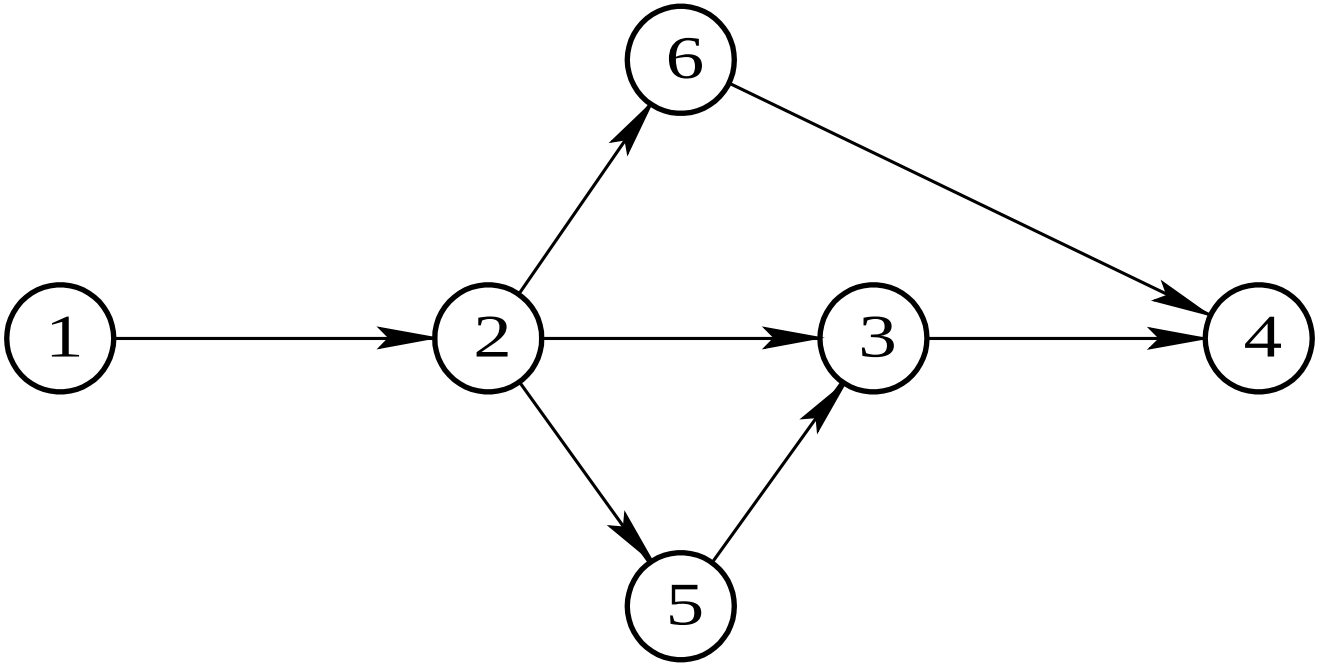


图7.13 题6图

7. 用迪杰斯特拉算法，求图7.14中从顶点①到其他各顶点的最短路径。要求写出：

(1) 图的带权邻接矩阵；

(2) 集合S到dist的变化过程；

(3) 从顶点①到其余各顶点的最短路径和最短路径长度。

8. 用弗洛伊德算法求图7.15中的每一对顶点间的最短路径，写出其计算过程。

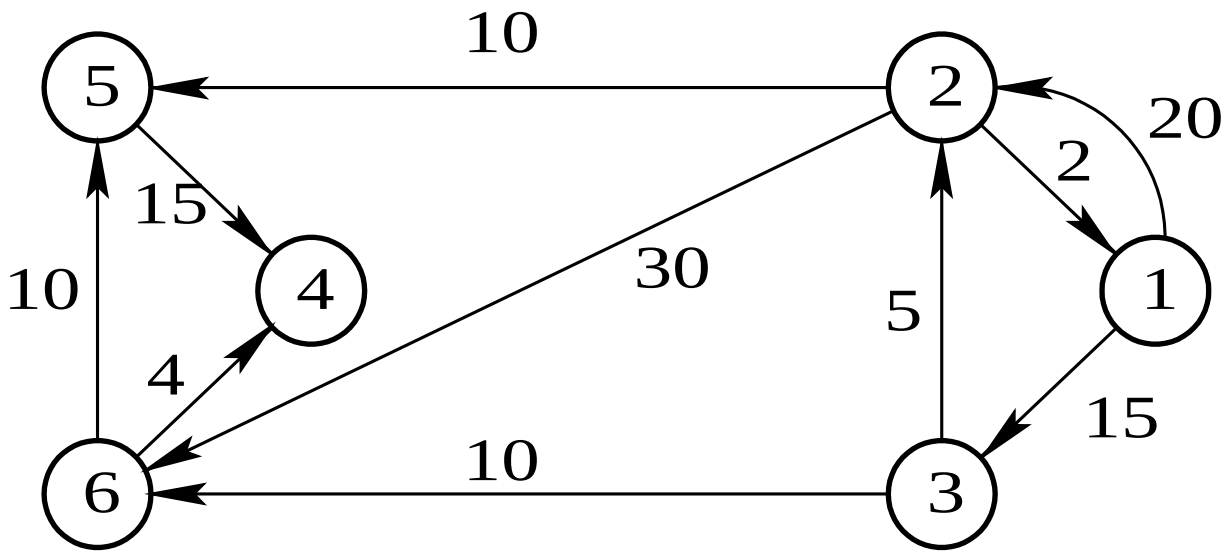
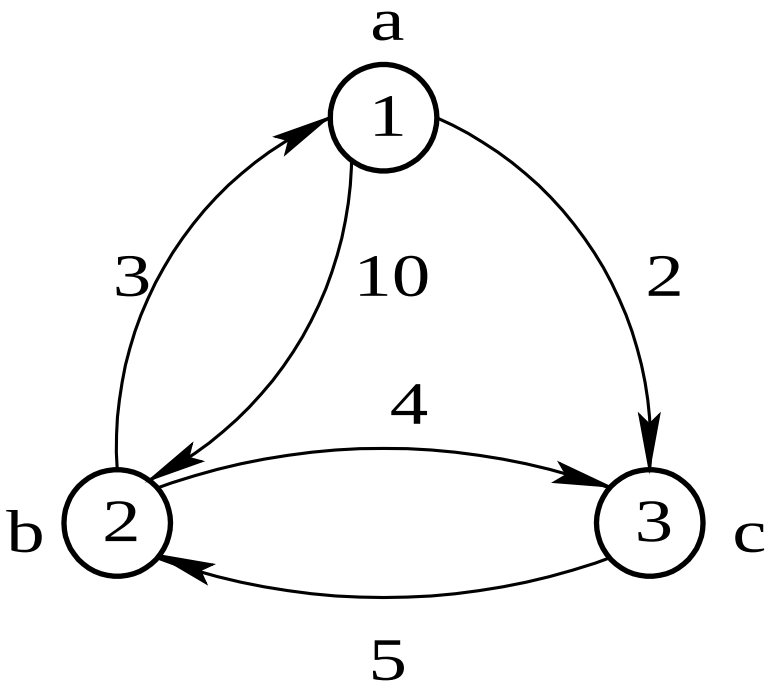


图7.14 题7图



9. 用C语言改写拓扑排序的程序，要求程序中使用队列而不使用堆栈，并上机验证。

第8章 查 找

8.1 静态查找表

8.1.1 顺序表的查找

顺序查找(Sequential Search)又称为线性查找，是一种最简单的查找方法。查找过程为：从线性表的一端开始顺序扫描线性表，依次将扫描到的结点关键字和给定值m 相比较，若当前扫描到的结点关键字与m相等，则查找成功；若扫描结束后，仍未找到关键字等于m 的结点，则查找失败。

例8.1 在关键字序列为{1，5，9，7，12，11，10，8，6，2，3}的线性表中查找关键字为8的元素。

顺序查找过程如图8.1所示。

顺序查找的线性表定义如下：

```
#define  MAXITEM  100      /*最多项数*/
struct element
{
KeyType  key;          /*关键字*/
  ElemType data;                /*其他数据*/
}
typedef struct element linelist [MAXITEM];
```

这里的KeyType和ElemType分别为关键字数据类型和其他数据的数据类型，可以是任何相应的数据类型，这里KeyType 默认为int型。

顺序查找的函数如算法8.1，其功能是在线性表r中顺序查找关键字为k的结点，若找到，则返回其位置i；若找不到，返回0。

开始：　1　5　9　7　12　11　10　8　6　2　3

第一次比较：　1　5　9　7　12　11　10　8　6　2　3

i　1

第二次比较：　1　5　9　7　12　11　10　8　6　2　3

i　2

第三次比较：　1　5　9　7　12　11　10　8　6　2　3

i　3

第四次比较：　1　5　9　7　12　11　10　8　6　2　3

i　4

第五次比较：　1　5　9　7　12　11　10　8　6　2　3

i　5

第六次比较：　1　5　9　7　12　11　10　8　6　2　3

i　6

第七次比较：　1　5　9　7　12　11　10　8　6　2　3

i　7

第八次比较：　1　5　9　7　12　11　10　8　6　2　3

i　8

图8.1　顺序查找过程

查找成功，返回序号8

```
/*算法描述8.1  顺序查找*/
int seqsearch (r，k，n)
linelist  r ;
int  k，n ;              /*n为线性表r中元素个数*/
{
    int i=1;
    while (r[i].key!=k) i++;
    if (i>n)   i=0;
    return (i);
}
```

算法分析：对于含有n 个结点的线性表，结点的查找在等概率的前提下，即$p_i=1/n$，平均查找长度为

$$ASL=\sum_{i=1}^{n}p_i c_i=\sum_{i=1}^{n}\left(\frac{1}{n}\right)\bullet i=\left(\frac{1}{n}\right)*(1+2+\cdots+n)=\frac{(n+1)}{2}$$

顺序查找和我们后面将要讨论到的其他查找算法相比，其缺点是平均查找长度较大，特别是当n很大时，查找效率较低，速度较慢，平均查找长度为O(n)。然而，它有很大的优点，即算法简单且适应面广。它对表的结构无任何要求，无论记录是否按关键字有序(若表中所有记录的关键字满足下列关系：r[i].key≤r[i+1].key(i=1,2,…,n－1)，则称表中记录按关键字有序)均可应用。

8.1.2 有序表的查找

以有序表表示静态查找表时，search 函数可用折半查找来实现，折半查找(Binary Search)的查找过程是：先确定待查记录所在的范围(区间)，然后逐步缩小范围直到找到或找不到该记录为止。也就是要求线性表中的结点必须已按关键字值的递增或递减顺序排列。它首先把要查找的关键字k与中间位置结点的关键字相比较，这个中间结点把线性表分成了两个子表，若比较结果相等则查找完成；若不相等，再根据k与该中间结点关键字的比较结果确定下一步查找哪个子表，这样递归进行下去，直到找到满足条件的结点或者该线性表中没有这样的结点。

例8.2　在关键字有序序列{3，5，6，8，9，12，17，23，30，35，39，42}中采用折半查找法查找关键字为8的元素。

指针low和high分别指示待查元素所在范围的下界和上界，指针mid指示区间的中间位置，即mid=[(low+high)/2]。在此例中，low 和high的初值分别为1和12，即[1，12]为待查范围。

折半查找过程如图8.2所示。

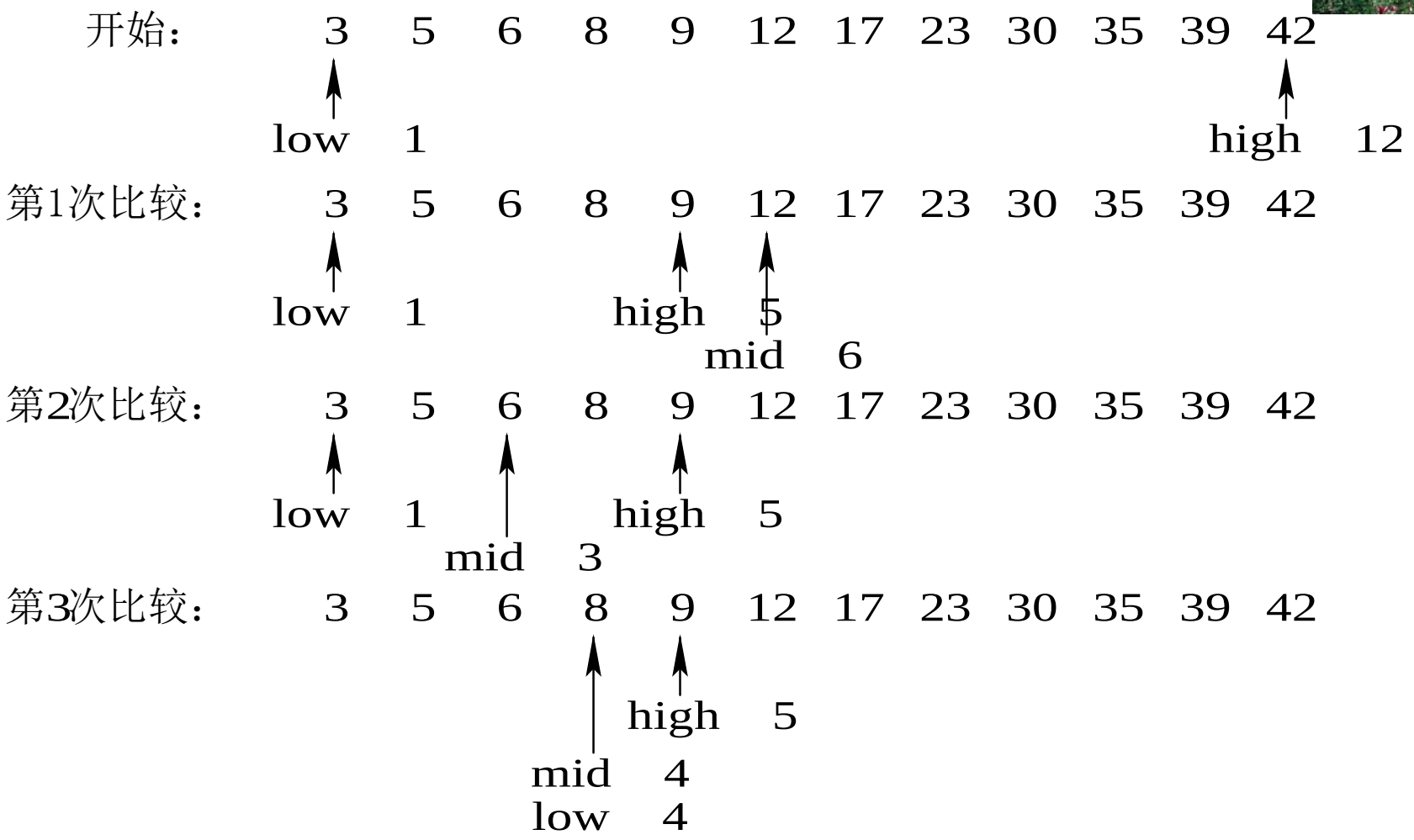


图8.2 折半查找过程

折半查找的函数如算法8.2，其功能是在线性表r中二分查找关键字为k的结点，查找到，则返回其位置i；若找不到返回0。

```
/*算法描述8.2  折半查找*/
  int binsearch (r，k，n)
sqlist  r;
int k，n;                    /*n为线性表r 中元素个数*/
{
   int i，low=1，high=n，mid;
   int find=0;               /*find=0表示未找到；find=1 表示已找到*/
   while (low <=high  && !find)
   {
```

```
    mid =(low+high)/2;                    /*整除*/
  if (k<r[mid].key)  high=mid-1;
  else if (k>r[mid].key)  low=mid+1;
      else {
           i=mid;
           find=1;
         }
}
if (!find)  i=0;
return (i);
}
```

算法分析：折半查找函数找到一个记录的过程恰好是一条从判定树的根到被查找结点的一条路径，而比较的次数恰好是树深。借助于二叉判定树很容易求得折半查找的平均查找长度。假设表长为n，树深h= lb(n+1)，则平均查找长度为

$$ASL=\sum_{i-1}^{n}p_i c_i=\frac{1}{n}\sum_{i=1}^{n}i\cdot 2^{i-1}=\frac{n+1}{n}lb(n+1)-1\approx lb(n+1)-1$$

因此，折半查找法的平均查找长度为O(lbn)，与顺序查找方法相比，折半查找的效率比顺序查找高，速度比顺序查找快，但折半查找只能适用于有序表，需要对n个元素预先进行排序，仅限于顺序存储结构(对线性链表无法进行折半查找)。

8.1.3 索引顺序表的查找

分块查找又称为索引顺序查找，是顺序查找的一种改进，其性能介于顺序查找和折半查找之间。分块查找把线性表分成若干块，每一块中的元素存储顺序是任意的，但块与块之间必须按关键字大小排序，即前一块中的最大关键字小于(或大于)后一块中的最小(或最大)关键字值。另外，还需要建立一个索引表，索引表中的一项对应线性表中的一块，索引项由关键字域和链域组成，关键字域存放相应块的最大关键字，链域存放指向本块第一个结点的指针。索引表按关键字值递增(或递减)顺序排列。

分块查找的查找函数分为两步进行：首先确定待查找的结点属于哪一块，即查找其所在的块；然后在块内查找要查的结点。由于索引表是递增有序的，采用折半查找时，块内元素个数较少，采用顺序法在块内查找，不会对执行速度有太大的影响。

例8.3 对于关键字序列为{8，20，13，17，40，42，45，32，49，58，50，52，67，70，78，80}的线性表采用分块查找法查找关键字为50的元素。假定分块索引和索引表如图8.3所示。

序号	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	8	20	13	17	40	42	45	32	49	58	50	52	67	79	78	80

(a)

key	low	high
20	1	4
45	5	8
58	9	12
80	13	16

图8.3 块表与索引表
(a) 块表；(b) 索引表

分块查找过程是：先在索引表中采用二分查找法查找关键字50所在的块，即在第3块中有4个元素，其关键字分别为49，58，50，52，在其中按顺序查找法进行查找，找到第3个元素(即总序号为11的元素)即关键字为50的元素。

索引表的定义如下:

```
struct indexterm
{
KeyType key;
int low，high;
}
typedef struct indexterm index[MAXITEM];
```

这里的KeyType是关键字的数据类型，可以是任何相应的数据类型，这里默认为int型。

分块查找的函数如算法8.3，其功能是在线性表r中分块查找关键字为k的结点，若找到，则返回其位置i；若找不到，返回0。

```
/*算法描述8.3 分块查找*/
int blksearch (r，idx，k，kug)
sqlist r;
index idx;
int k，kug;
{                                    /*kug为块的个数*/
int i，low 1=1，high1=kug，mid1，db，find=0;
while (low<=high1 && !find)          /*二分查找索引表*/
{
        mid1=(low1+high1)/2;
```

```
if (k<idx[mid1].key) high1=mid1-1;
  else if(k>idx[mid1].key) low1=mid1+1;
  else {
          high1=mid1-1；
          find=1；
   }
         }
if (low1<kug)                              /*k小于索引表内最大值*/
         {
```

```
 i=idx[low1].low;                    /*在索引表中定块起始地址*/
      db=idx[low1].high;             /*在索引表中定块终止地址*/
}                                    /*在指定的块内采用顺序方法进行查找*/
while (i<kug && r[i].key!=k) i++;
if (r[i].key!=k) i=0;
return (i);
}
```

算法分析：分块查找实际上进行了两次查找，故整个算法的平均查找长度是两次查找的平均查找长度之和。

假设有n个结点，分成kug块，每块有m个结点，即kug=n/m,每块的查找概率为1/kug，块内每个结点的查找概率为1/m，则

$$ASL \approx lb(kug+1)-1+\frac{m+1}{2} \approx lb\left(\frac{n}{m}+1\right)+\frac{m}{2}$$

分块查找的优点是：在线性表中插入或删除一个元素时，只要找到相应的块，然后在该块内进行插入或删除即可。由于块内元素个数相对较少，而且是任意存放的，因此插入或删除比较容易，不需要移动大量的元素。

8.2 动态查找表

8.2.1 二叉排序树

二叉排序数(Binary Sort Tree，又记为BST)或者是一棵空树，或者是具有如下性质的二叉树：

(1) 若它的左子树非空，则左子树上所有结点的值均小于它的根结点的值；

(2) 若它的右子树非空，则右子树上所有结点的值均大于它的根结点的值；

(3) 它的左、右子树本身又各是一棵二叉排序树。

图8.4所示的就是两棵二叉排序树。

二叉排序树又称二叉查找树，由上述定义的结构特点，再结合图8.4可以发现二叉树的重要性质：它的查找过程和次优二叉树类似，即当二叉排序树不空时，首先将给定值和根结点的关键字比较，若相等，则查找成功，否则将依据给定值和根结点的关键字之间的大小关系，分别在左子树或右子树上继续进行查找。

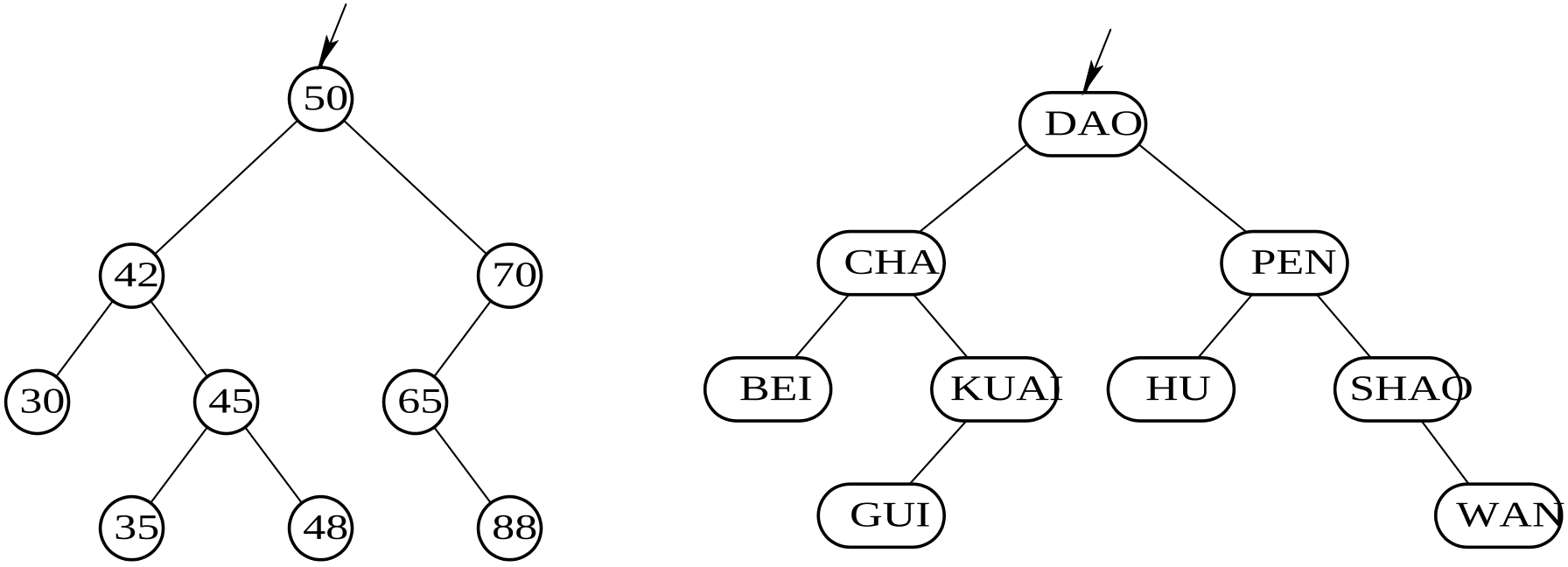


图8.4 二叉排序树示例

二叉排序树用二叉链表来存储，说明形式如下：

```
typedef  struct  bstnode
{ datatype  key;
   struct bsnode * lchild, * rchild;
   datatype  other;
    }bstnode;
```

二叉树的构造是通过依次输入数据元素，并把它们插入到二叉树的适当位置上来实现的。具体的过程是：每读入一个元素，建立一个新结点，若二叉排序树非空，则将新结点的值与根结点的值相比较，如果小于根结点的值，则插入到左子树中，否则插入到右子树中；若二叉排序树为空，则新结点作为二叉排序树的根结点。

由于二叉排序树是递归的，子树的插入过程与在树中的插入过程相同，因此，可以写出这个递归过程，算法如下：

```
/*算法描述8.4*/
procedure  bstinsert (bstnode *s,bstnode*t)
/*将新结点s插入到指针t所指的二叉排序树中*/
{   if t =NULL t=s;
    else
      if (s -> key < t->key)
         bstinsert(s,t->lchild);
           /*插入到左子树上*/
else bstinsert (s,t->rchild);
           /*插入到右子树上*/
} /*bstinsert*/
```

例如，在图8.4(a)上插入关键字为55的过程为：由于此时二叉树非空，则将55与根结点50相比较，因为55>50，则应将55插入到50的右子树上，又因为50的右子树非空，将55与右子树的根70比较，因55<70，则55应插入到70的左子树中，依次类推，当最后因55<65，且65的左子树为空，将55作为65的左子树插入到树中。

整棵树的构造过程的算法描述如下:

```
/*算法描述8.5*/
bstset (bstnode *T)
/*输入一个数据元素序列，建立一棵二叉排序树，endmark为输入结束标志*/
{ int x ;
bstnode *s;
  T=NULL;
  scanf("%d",&x);
while (x!=endmark ) do
```

```
{s=(bstnode*) malloc(sizeof(bstnode));
s->key=x;
scanf("%d",&other);
s->lchild= NULL;
s->rchild= NULL;                    /*建立新结点*/
bstinsert (s，p);          /*新结点插入*/
scanf("%d",&x)；}    /*读入下一个结点*/
   } /*bstset*/
```

例8.4 已知一关键字序列为{40，25，52，10，28，80}，则生成二叉排序树的过程如图8.5所示。

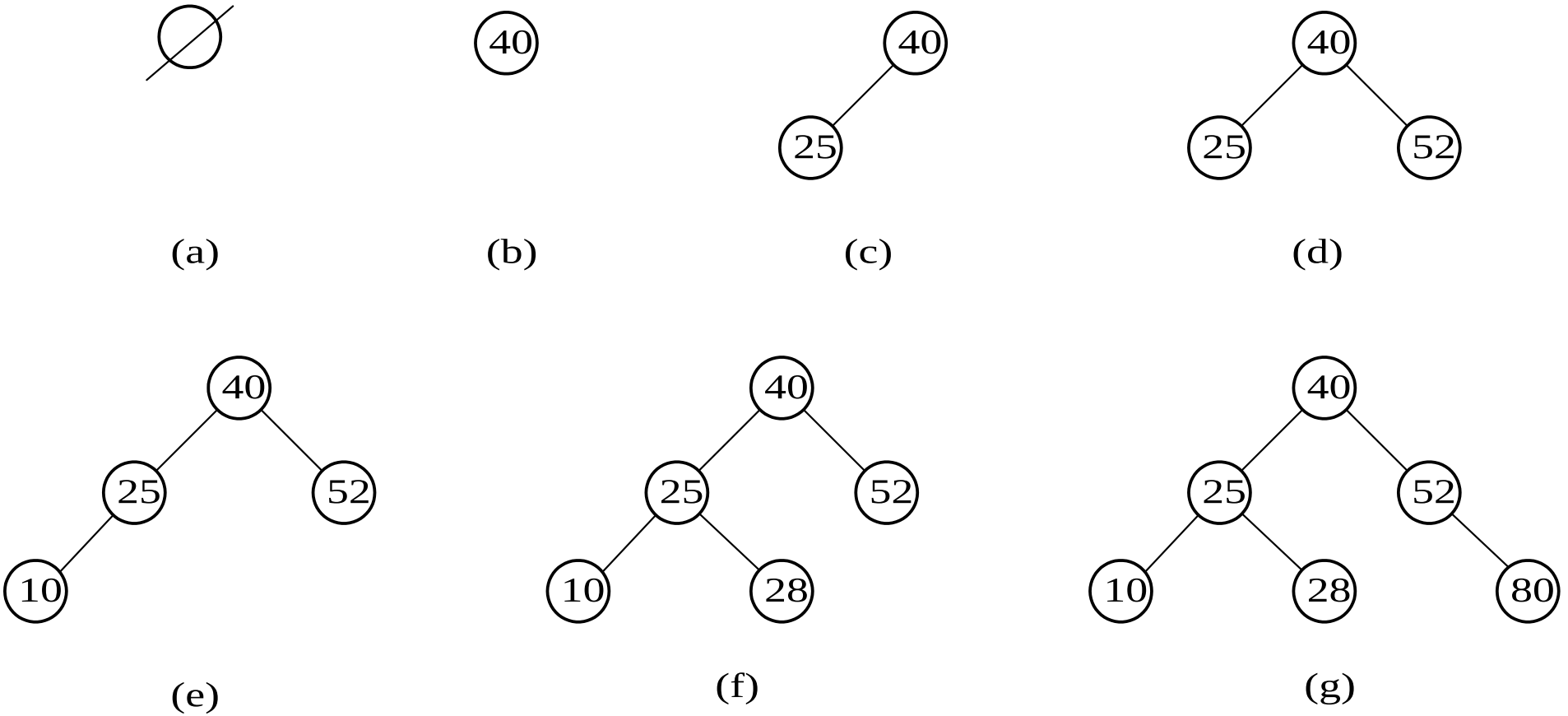


图8.5 二叉排序树的构造过程

由此可以看出：二叉排序树的形态完全由一个输入序列决定，一个无序序列可以通过构造一棵二叉排序树而变成一个有序序列。若输入序列为10，25，28，40，52，80，则所对应的二叉排序树为单支树，如图8.6所示。

此外，从图8.5所示的二叉排序树构造过程中还可以看出，每次插入的新结点都是作二叉排序树的叶子结点，在插入过程中不需要移动其他元素，只需将某个结点的指针由空变为非空即可，所以二叉排序树结构适合于元素的经常插入，那么如何在二叉排序树上删除一个结点呢?在二叉排序树中删除一个结点，不能把以该结点为根的子树都删去，只能删除这个结点并仍旧保证二叉排序树的特性，也就是说删除二叉排序树上一个结点相当于删除有序序列中的一个元素。

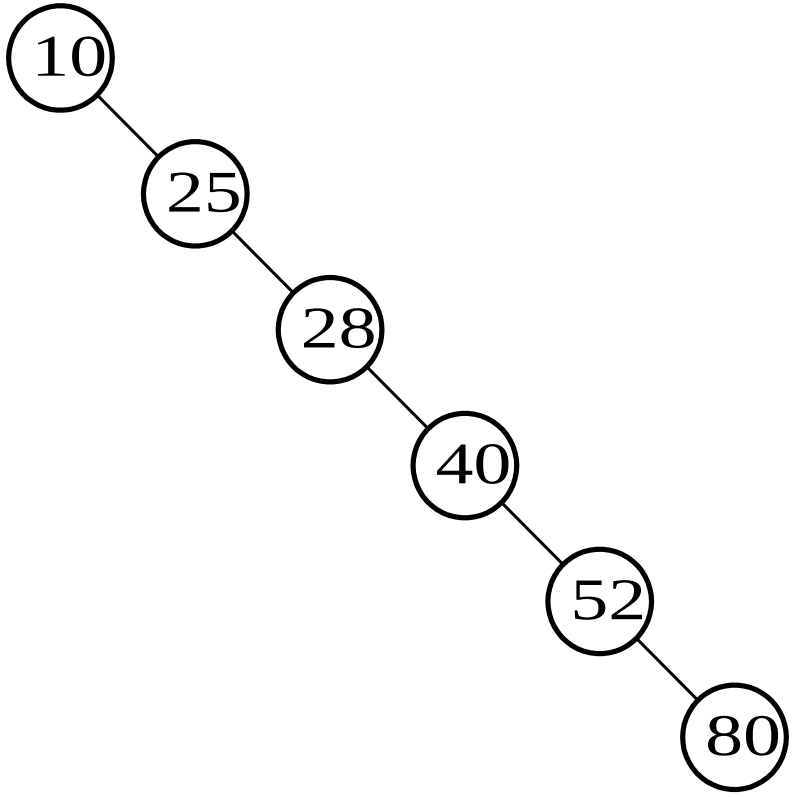


图8.6 单支二叉排序树

假设在二叉排序树上被删除结点为p(p指针指向被删除结点)，f为其双亲结点，则删除结点p的过程分三种情况讨论：

(1) 若p结点为叶子结点，即p->lchild及p->rchild均为空，则由于删去叶子结点后不破坏整棵树的结构，因此，只需修改p结点的双亲结点指针即可：

f->lchild (或f-> rchild) =NULL；

(2) 若p结点只有左子树或者只有右子树，此时只需用p的左子树或右子树的根结点取代p成为双亲f的左子树(或右子树)，即令

f->lchild (或f-> rchild)=p->lchild；

或　　f->lchild (或f->rchild)=p->rchild；

(3) 若p结点的左、右子树均不空，此时不能像上面那样简单处理，删除p结点时应考虑将p的左子树、右子树连接到适当的位置，并保证二叉排序树的特性。有两种方法，一是令p的左子树为双亲f的左子树，而p的右子树下接到p的中序遍历前驱结点s的右指针上；二是令p的中序前驱结点s结点代替p结点，然后删除s结点。

查找p结点中序前驱结点的操作为：

```
q=p;
s=p->lchild;
while  (s->rchild!=NULL) do
{q=s;  s= s->rchild;}
```

因为二叉排序树可以看成是一个有序表，在二叉排序树中左子树上所有结点的关键字均小于根结点的关键字，右子树上所有结点的关键字均大于或等于根结点的关键字，所以在二叉排序树上进行查找与折半查找类似。

查找过程为：若二叉排序树非空，将给定值与根结点值相比较，若相等，则查找成功；若不等，则当根结点值大于给定值时，到根的左子树中查找，否则在根的右子树中查找。这显然仍是一个递归过程。

```
/*算法描述8.6*/
bstnode *bstsrch (bstnode *t,int k)
/*在t 所指的二叉排序树中查找关键字等于给定值k的记录*/
{
  if t!=NULL
    if (t->key =k) return(t);
      /*查找成功，返回t指针*/
else if (k<t->key)
       bstsrch(t->lchild；k);
else bstsrch(t->rchild；k);
else return(t)；
} /*bstsrch*/
```

可见，在二叉排序树上查找其关键字等于给定值的结点的过程，恰是走了一条从根结点到该结点的路径的过程，和给定值进行比较的关键字个数等于路径长度加1(或结点所在层次数)。二叉排序数的平均查找长度为:

$$ASL_{bst} = \sum_{i-1}^{n} P_i C_i$$

长度为n的有序表其判定树惟一，故平均查找长度惟一；但二叉排序树的形态与关键字的输入有关。例如，图8.4(a)所示的二叉树的平均查找长度为：

$$ASL_a = \frac{1}{9} \times (1 + 2 \times 2 + 3 \times 3 + 4 \times 3) = \frac{26}{9} \approx 2.89$$

同样结点数的单支二叉排序树，其平均查找长度为：

$$ASL_{a单} = \frac{1}{9} \times (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9) = \frac{45}{9} = 5$$

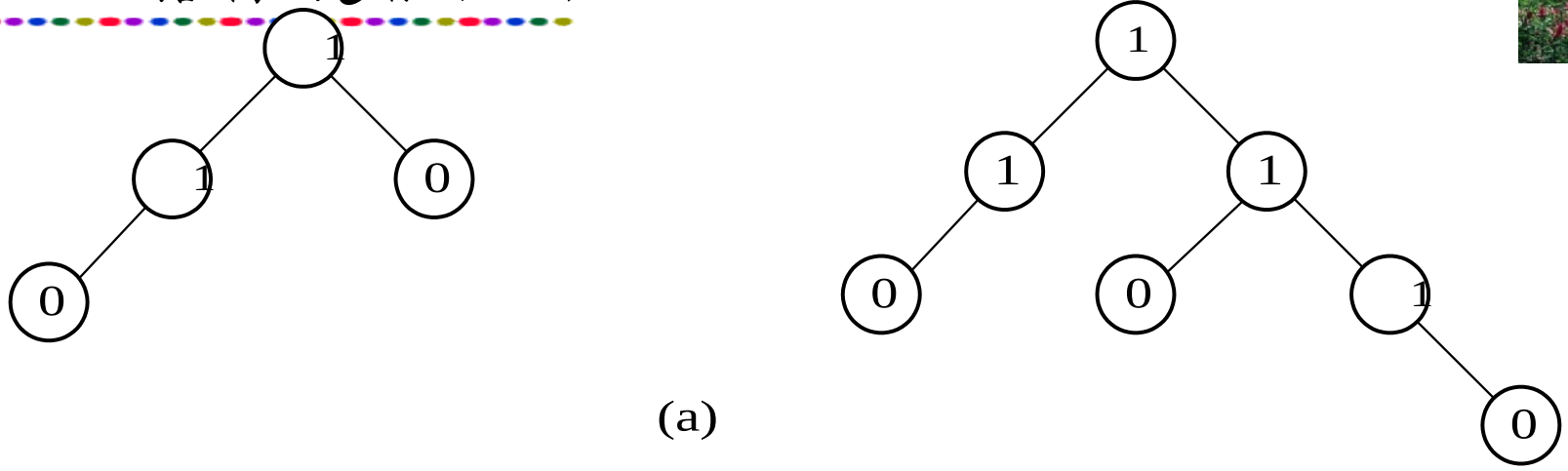
最坏的情况下，二叉排序树变为单支树，最好的情况是二叉排序树比较均匀。

二叉排序树的性能与折半查找相差不大，但二叉排序树对于结点的插入和删除十分方便，因此对于经常进行插入、删除和查找运算的表，应采用二叉排序树。

8.2.2 平衡二叉树

平衡二叉树(Balanced Binary Tree)又称AVL树。

它或者是一棵空树，或者是具有下列性质的二叉树：它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过1。二叉树上结点的平衡因子定义为该结点的左子树的深度减去它的右子树的深度。因此，平衡二叉树上所有结点的平衡因子为－1、0或1中的一个值，只要二叉树上有一个结点的平衡因子的绝对值大于1，则该二叉树就是不平衡的。图8.7(a)、(b)分别给出了两棵平衡二叉树和两棵不平衡二叉树，树中结点内的数字是该结点的平衡因子。



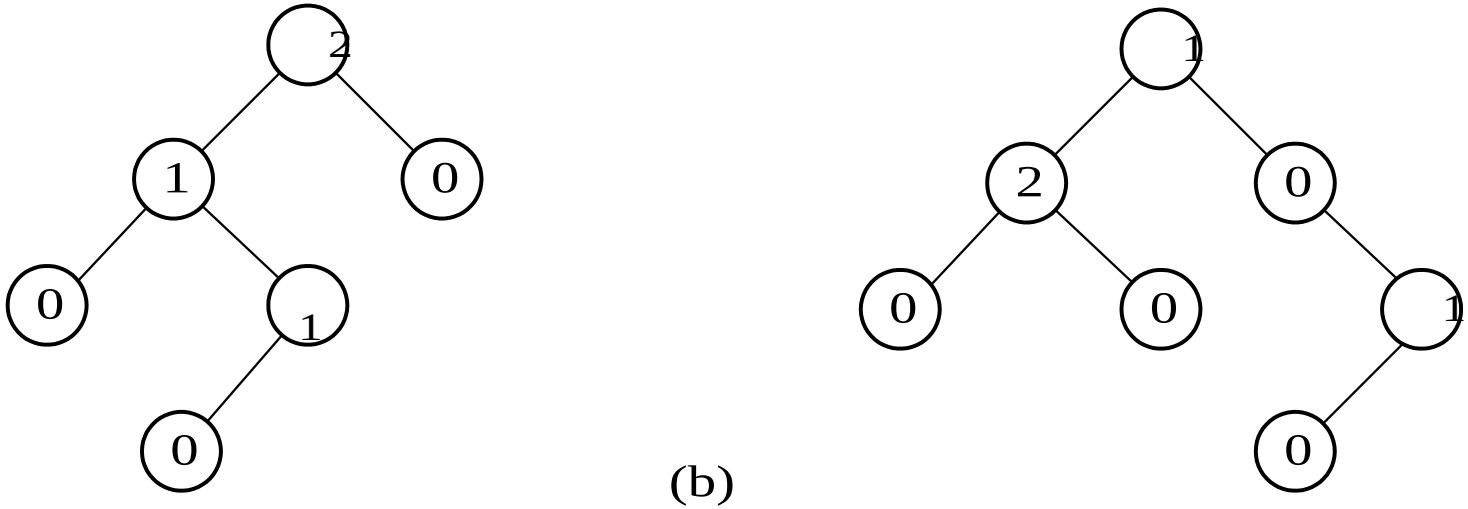


图8.7 平衡二叉树和不平衡二叉树
(a) 平衡二叉树；(b) 不平衡二叉树

由上节得知，二叉排序树形态均匀时性能才能最好，而形态为单支树时其性能与顺序查找相同，因此，我们希望在初始时及在有元素变动时的二叉排序树都是平衡二叉树，那么如何使构成的二叉排序树成为平衡树呢？

例如，输入关键字序列为(15，20，35，80，50)，构造一棵二叉排序树。

(1) 输入15，15作为根结点，见图8.8(a)。

(2) 输入20，20作为15的右子树，见图8.8(b)，此时仍为平衡树。

(3) 输入35，35作为20的右子树，见图8.8(c)，此时结点15的平衡因子是－2，该树为非平衡树，必须进行调整。显然要进行逆时针右转，20作为根结点，见图8.8(d)，此时又是一棵平衡二叉树。

(4) 输入80，80作为35的右子树，见图8.8(e)，此时树为一棵平衡二叉树。

(5) 输入50,50作为80的左子树，见图8.8(f)，由于树中又出现有绝对值大于1的平衡因子，所以必须进行调整。先将50作为35的右子树，80作为50的右子树，见图8.8(g)，然后令50作为20的右子树，35和80分别作50的左、右子树，见图8.8(h)。

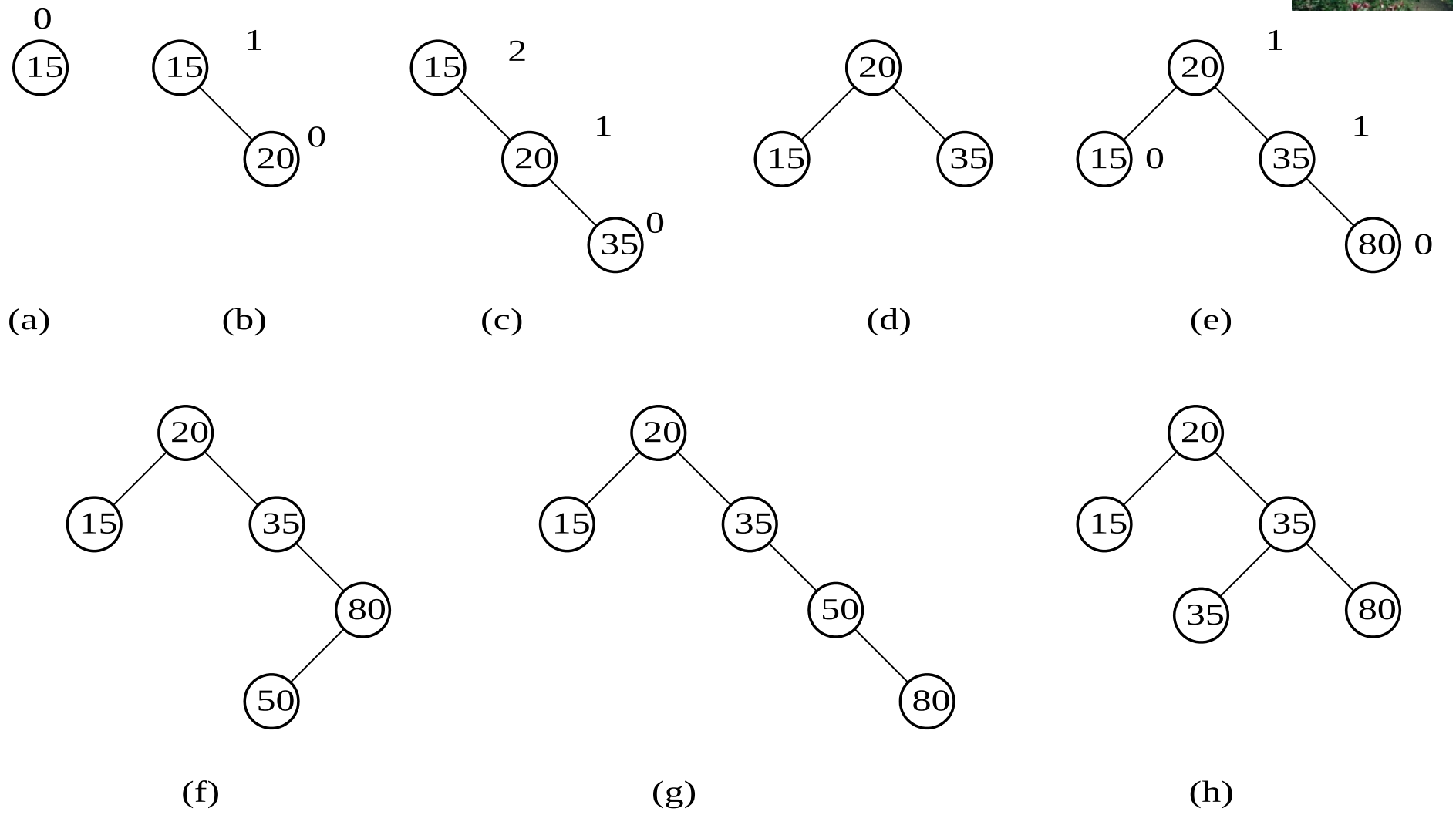


图8.8 平衡二叉树的构造示例

可见，保持二叉树为平衡二叉树的基本思想是：每当对二叉排序树插入一个结点时，首先检查是否因插入而破坏了树的平衡，若是，则找出其中最小的不平衡树，在保持二叉排序树的特性情况下，调整最小不平衡子树中结点之间的关系，以达到新的平衡。离插入结点最近且以平衡因子的绝对值大于1的结点作根的子树被称为最小平衡树。

失去平衡后调整方法可归纳为四种类型：① LL型平衡旋转；② RR型平衡旋转；③ LR型平衡旋转；④ RL型平衡旋转。

平衡二叉排序树上的查找与二叉排序树中的查找相同，由于平衡二叉树的形态不再变为单支树的情形，且比较均匀，故在AVL树上查找的时间复杂度为O(lbn)。由于在动态平衡的过程中所进行的调整操作仍需花费不少时间，因此AVL树的使用要视具体情况而定。

8.3 哈希表及其查找

8.3.1 哈希表与哈希函数

在前面讨论的各种结构(线性表、树等)中，记录在结构中的相对位置是随机的，和记录的关键字之间不存在确定的关系。因此，在结构中查找记录时需进行一系列和关键字的比较。这一类查找方法建立在“比较”的基础上。在顺序查找时，比较的结果为“=”与“≠”两种可能；在折半查找、二叉排序树查找时，比较的结果为“<”、“=”和“>“三种可能。查找的效率依赖于查找过程中所进行的比较次数。

理想的情况是希望不经过任何比较，一次存取便能得到所查记录，那就必须在记录的存储位置和它的关键字之间建立一个确定的对应关系f，使每个关键字和结构中一个惟一的存储位置相对应。因而在查找时，只要根据这个对应关系f找到给定值k的像f(k)即可。若结构中存在关键字和k相等的记录，则必定在f(k)的存储位置上，由此，不需要进行比较便可直接取得所查记录。在此，我们称这个对应关系f为哈希(Hash)函数，按这个设想建立的表为哈希表。

举一个哈希表的最简单的例子。假设要建立一张全国30个地区的各民族人口统计表，每个地区为一个记录，记录的各数据项为：

编号	地区名	总人口	汉族	回族	……

显然，可以用一个一维数组c(1:30)来存放这张表，其中c[i]是编号为i的地区的人口情况。编号i便为记录的关键字，由它惟一确定记录的存储位置c[i]。例如，假设北京市的编号为1，则若要查看北京市的各民族人口，只要取出c[1]的记录即可。假如把这个数组看成是哈希表，则哈希函数f(key)=key。然而很多情况下的哈希函数并不如此简单。不能简单地取哈希函数f(key)=key，而是首先要将它们转化为数字，有时还要做些简单的处理。例如，假设有这样的哈希函数：取关键字中第一个字母在字母表中的序号作为哈希函数，其中Beijing的哈希函数值为字母“B”在字母表中的序号，等于02，Tianjin的哈希函数值为字母“T”在字母表中的序号，等于20，见表8.1。

表8.1 简单的哈希函数示例

key	Beijing (北京)	Tianjin (天津)	Shanghai (上海)	Hebei (河北)	Henan (河南)	Shandong (山东)	Shanxi (山西)	Sichuan (四川)
f(key)	02	20	19	08	08	19	19	19

从这个例子可见：

(1) 哈希函数是一个映像，因此哈希函数的给定很灵活，只要使得任何关键字由此所得的哈希函数值都落在表长允许范围之内即可。

(2) 对不同的关键字可能得到同一哈希地址，即key1≠key2，而f(key1)=f(key2)，这种现象称为冲突(Collision)。

综上所述，可像如下这样描述哈希表：根据设定的哈希函数H(key)和处理冲突的方法将一组关键字映像到一个有限的连续的地址集(区间)上，并以关键字在地址集中的“像”作为记录在表中的存储位置，这种表便称为哈希表，这一映像过程称为哈希表或散列，所得存储位置称哈希地址或散列地址。

下面分别就哈希函数和处理冲突的方法进行讨论。

8.3.2 构造哈希函数的常用方法

构造哈希函数的方法很多，如何构造一个“好”的哈希函数是技术性和实践性都很强的问题。这里的“好”指的是哈希函数的构造比较简单，并且用此哈希函数产生的映像所发生冲突的可能性最小，换句话说，一个好的哈希函数能将给定的数据集合均匀地映射到所给定的地址区间中。

我们知道，关键字可以惟一地对应一个记录，因此，在构造哈希函数时，应尽可能地使关键字的各个成分都对它的哈希地址产生影响。下面介绍几种常用的构造哈希函数的方法。

1. 直接地址法

对于关键字是整数类型的数据，直接地址法的哈希函数H直接利用关键字求得哈希地址。

$$H(K_i)=aK_i+b，\quad (a，b为常量)$$

在使用时，为了使哈希地址与存储空间吻合，可以调整a和b。例如，取$H(K_i)=K_i+10$。

直接地址法的特点是：哈希函数简单，并且对于不同的关键字不会产生冲突。但在实际问题中，由于关键字集中的元素很少是连续的，用该方法产生的哈希表会造成空间的大量浪费。因此，这种方法很少使用。

2. 数字分析法

数字分析法是假设有一组关键字，每个关键字由几位数字组成，如K_1 K_2 K_3…K_n。数字分析法是从中提取数字分布比较均匀的若干位作为哈希地址。

例如，对于关键字K_1到K_8的序列{100011211，100011322，100011413，100011556，100011613，100011756，100011822，100011911}，可以取第6位和第7位作为哈希地址，即$H(K_1)=12$，$H(K_2)=13$，$H(K_3)=14$，$H(K_4)=15$，$H(K_5)=16$，$H(K_6)=17$，$H(K_7)=18$，$H(K_8)=19$。

3. 平方取中法

平方取中法是取关键字平方的中间几位作为散列地址的方法，具体取多少位视实际情况而定，即

$$H(K_i)=\text{“}K_i\text{的平方的中间几位”}$$

这也是一种常用的较好的设计哈希函数的方法。关键字平方后使得它的中间几位和组成关键字的多位值均有关，从而使哈希地址的分布更为均匀，减少了发生冲突的可能性。

4. 折叠法

折叠法是首先把关键字分割成位数相同的几段(最后一段的位数可少一些)，段的位数取决于哈希地址的位数，由实际情况而定，然后将它们的叠加和(舍去最高进位)作为哈希地址的方法。

与平方取中法类似，折叠法也使得关键字的各位值都对哈希地址产生影响。

5. 除留余数法

除留余数法是用关键字K_i除以一个合适的不大于哈希表长度的正整数P，所得余数作为哈希地址的方法。对应的哈希函数$H(K_i)$为：

$$H(K_i)=K_i \text{ MOD } P$$

这里的MOD表示求余数运算，用该方法产生的哈希函数的好坏取决于P值的选取。实践证明，当P取小于哈希表长的最大质数时，产生的哈希函数较好。

除留余数法是一种简单而行之有效的构造哈希函数的方法。

8.3.3 解决冲突的主要方法

1. 开放地址法

开放地址法又分为线性探测再散列、二次探测再散列和随机探测再散列。

假设哈希表空间为T(0..M－1)，哈希函数为$H(K_i)$。MOD表示求模运算。线性探测再散列解决冲突求“下一个”地址的公式是：

$$d_1=H(K_i)$$

$$d_j=(d_{j-1}+j) \text{ MOD } M \quad (j=1, 2,$$

二次探测再散列解决冲突求“下一个”地址的公式是：

$$d_1=H(K_i)$$

$$d_{2j}=(d_1+j^2)\ \text{MOD}\ M$$

$$d_{2j+1}=(d_1-j^2)\ \text{MOD}\ M\ \ (j=1, 2, \ldots, M/2)$$

随机探测再散列解决冲突求“下一个”地址的公式是：

$$d_1=H(K_i)$$

$$d_{j+1}=(d_1+R)\ \text{MOD}\ M$$

其中，R为伪随机数序列。

例8.5 使用的哈希函数为：

$$H(K_i)=3K_i \text{ MOD } 11$$

并采用开放地址法处理冲突，其求下一个地址的函数为：

$$d_1=H(K_i)$$

$$d_i=(d_{i-1}+(7K_i \text{ MOD}10)+1) \text{ MOD } 11 \quad (i=2, 3, \ldots)$$

试在0～10的散列地址空间中对关键字序列(22，41，53，46，30，13，01，67)构造哈希表，求等概率情况下查找成功的平均查找长度，并设计构造哈希表的完整函数。

本题的哈希表构造过程如下：

(1) H(22)=3*22 MOD 11=0

(2) H(41)=3*41 MOD 11=2

(3) H(53)=3*53 MOD 11=5

(4) H(46)=3*46 MOD 11=6

(5) H(30)=3*30 MOD 11=2 (冲突)

d_1=H(30)=2

H(30)=(2+(7*30 MOD 10)+1) MOD 11=3

(6) H(13)=3*13 MOD 11=6 (冲突)

d_1=H(13)=6

H(13)=(6+(7*13 MOD 10)+1) MOD 11=8

(7) H(1)=3*01 MOD 11=3 (冲突)

d_1=H(1)=3

H(1)=(3+(1*7 MOD 10)+1) MOD 11=0 (冲突)

d_2=H(1)=0

H(1)=(0+(1*7 MOD 10)+1)MOD 11=8 (冲突)

d_3=H(1)=8

H(1)=(8+(1*7 MOD 10)+1) MOD 11=5 (冲突)

d_4=H(1)=5

H(1)=(5+(1*7 MOD 10)+1)MOD 11=2 (冲突)

d_5=H(1)=2

H(1)=(2+(1*7 MOD 10)+1)MOD 11=10

(8) H(67)=3*67 MOD 11=3 （冲突）

d_1=H(67)=3

H(67)=(3+(7*67 MOD 10)+1) MOD 11=2 (冲突)

d_2=H(67)=2

H(67)=(2+(7*67 MOD 10)+1) MOD 11=1

查找成功的平均查找长度为：

$$ASL=(1\times4+2\times2+3\times1+6\times1)\times(1/8)=2.125$$

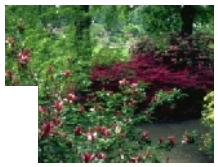
构造本哈希表的程序如下：

```
/*算法描述8.7*/
#include<stdio.h>
#define M 11
#define N 8
struct hterm
{
    int  key;                /*关键字值*/
    int   si;                /*散列次数*/
}
```

```
struct  hterm  hashlist[M+1];
int i, address, sum, d, x[N+1];
float average;
main()
{
for(i=1;i<=M;i++)                    /*置初值*/
{
    hashlist[i].key=0;
    hashlist[i].si=0;
}
```

```
x[1]=22;  x[2]=41;  x[3]=53;
x[4]=46;  x[5]=30;  x[6]=13;
x[7]=1;   x[8]=67;
for(i=1;i<=N,i++)
{
   sum=0;
address=(3 *x[i]) % M;
d=address;
if (hashlist [address].key==0)
{
hashlist[address].key=x[i];
hashlist[address].si=1;
}
```

```
else
{
do                                  /*处理冲突*/
{
d=(d+(x[i]*7) %10+1) % 11;
sum=sum+1;
}   while (hashlist[d].key!=0);
hashlist[d].key=x[i];
hashlist[d].si=sum+1;
}
}
```



```
printf("哈希表地址： ");
for (i=0; i<M; i++)  printf ("%-4d",i);
printf("\n");
printf("哈希表关键字： ");
for(i=0;i<M;i++)  printf ("%-4d", hashlist[i].key);
printf("搜索长度： ");
for (i=0; i<M; i++)  printf ("%-4d", hashlist[i].si);
printf("\n");
average=0;
for (i=0; i<M; i++)  average=average+hashlist[i].si;
average=average/N;
printf("平均搜索长度： ASL(%d)=%g", N,average);
}
```

2. 再哈希法

$$H_i=RH_i(key) \quad (i=1, 2, 3,\ldots, k)$$

RH_i序列均是不同的哈希函数，即在同义词产生地址冲突时计算另一个哈希函数地址，直到冲突不再发生，这种方法不易产生“聚集”，但增加了计算的时间。

3. 链地址法

链地址法是将所有关键字为同义词的记录存储在同一线性链表中。假设某哈希函数产生的哈希地址在区间[0，m－1]上，则设立一个指向记录类型element的指针型向量

element * chainhash[m] ;

其每个分量的初始状态都是空指针。凡哈希地址为i的记录都插入到头指针为chainhash[i]的链表中。在链表中的插入位置可以在表头或表尾，也可以在中间，以保持同义词在同一线性链表中按关键字有序。

8.4 实习：哈希表查找设计

实验程序

```
#include <stdio.h>
#define  MAX  100
int  ha[MAX], hlen[MAX ], n, m, p;
void creathash()
{
        int i, j, d, d1, odd, s, key[MAX];
        printf ("= = = = = 建立散列表= = = = =
        printf ("输入元素个数n：");
    scanf ("%d",&n);
```

```
printf ("输入哈希表长m：");
scanf ("%d",&m);
printf ("散列函数：h(k)=k mod p：");
scanf ("%d",&p);
    for (i=0; i<m; i++)  ha[i]=hlen[i]=0;
  /*hlen [i] 为第i个元素的查找长度*/
i=0 ;
  while (i<n )
{
  printf ("第%d个元素：",i+1);
      scanf ("%d",&key[i]);
```

```
odd =1;
if (key[i]<=0) printf ("输入错误，重新输入！\n");
else  i++;
}
i=0 ;
printf ("哈希表建立如下：\n");
while (i<n)
{
     odd =1;
     d =d1=key[i]%p;
     j=s=1;                              /*s记录比较次数*/
printf ("H (%d)=%d  MOD  %d=%d ",key [i],key[i], p, d);
while (ha[d]!=0)
    {
```

```
printf ("冲突 \n");
if (odd)
{
d =(d1+j *j) %m;
printf ("H(%d )=(%d +%d * %d )  MOD  %d =%d ",key[i], d1, i, j, m, d);
      odd =0;
}
else
{
d=(d1-j*j) %m;
        printf ("H(%d) =(%d -%d * %d )  MOD  %d =%d", key[i], d1, i, j, m, d );
```

```
                    odd=1;
                        j++;
                }
                s++;
        }
        printf ("比较%d次\n",s);
        ha[d]=key[i];
        hlen [d]=s;
        i++;
         }
           }
```

```
void disphash()
{
int  i,  s=0;
printf ("\n散列表H为：  \n");
    for (i=0; i<m; i++)
        printf ("%3d", i );
printf ("\n");
for (i=0;i<m;i++)
        printf ("%3d" ,ha[i] );
    printf ("\n");
        for (i=0;i<m;i++)
```

```
            printf("%3d",hlen[i]);
        printf("\n");
        for (i=0; i<m ; i++)  s=s+hlen[i];
      printf ("\t ASL (%d)=%6.2f \n", n, (1.0*s)/n);
}
void find hash()
{
   int x , j, d ,d1, odd=1;
   printf ("\n输入查找的值：");
   scanf ("%d”,&x);
   d=d1=x % p;
       j=1;
```

```
while (ha[d]!=0 && ha[d]!=x)
{
if (odd)
    {
  d=(d1+j*j)%m;
       odd=0;
 }
 else
{
   d=(d1-j*j)%m;
   odd=1;
   j++;
   }
       }
```

```
if (ha[d]==0) printf ("\t输入的查找值不正确！\n");
else printf ("\t查找值：ha[%d]=%d!\n",d,x );
}
main()
{
creathash ();
disphash ();
find hash ();
    }
```

实验结果

设数序为53,17,12,61,98,70,87,25,63,46,14,59,67,75，哈希表长M=18，哈希函数为

$$H(k)=k \text{ MOD } 17$$

建立对应的哈希表，采用开放地址法中的二次探测再散列解决冲突，并查找值为70的元素位置。

程序执行过程如下：

= = = = = = = 建立散列表= = = = = = = =

输入元素个数n：14<CR> (CR表示回车)

输入哈希表长m：18<CR>

散列函数：h(k)=k mod p ： 17<CR>

第 1 个元素：53<CR>

第2个元素：17<CR>

第3个元素：12<CR>

第4个元素：61<CR>

第5个元素：98<CR>

第6个元素：70<CR>

第7个元素：87<CR>

第8个元素：25<CR>

第9个元素：63<CR>

第10个元素：46<CR>

第11个元素：14<CR>

第12个元素：59<CR>

第13个元素：67<CR>

第14个元素：75<CR>

哈希表建立如下：

H(53)=53 MOD 17 =2　　比较1次

H(17)=17 MOD 17 =0　　比较1次

H(12)=12 MOD 17 =12　　比较1次

H(61)=61 MOD 17 =10　　比较1次

H(98)=98 MOD 17 =13　　比较1次

H(70)=70 MOD 17 =2　　冲突

H(70)=(2+1*1) MOD 18 =3　　比较2次

H(87)=87 MOD 17 =2　　冲突

H(87)= (2+1*1) MOD 18 =3　　冲突

H(87)= (2－1*1) MOD 18 =1　　比较3次

H(25)=25 MOD 17 =8　　比较1次

H(63)=63 MOD 17 =12　　冲突

H(63)= (12+ 1*1) MOD 18 =13　　冲突

H(63)= (12－1*1) MOD 18 =11　　比较3次

H(46)=46 MOD 17 =12　　冲突

H(46)= (12+1*1) MOD 18 =13　　冲突

H(46)= (12－1*1) MOD 18 =11　　冲突

H(46)= (12+2*2) MOD 18 =16	比较4次
H (14)= 14 MOD 17 =14	比较1次
H (59)= 59 MOD 17 =8	冲突
H (59)= (8+1*1) MOD 18 =9	比较2次
H (67)= 67 MOD 17 =16	冲突
H (67)= (16+1*1) MOD 18 =17	比较2次
H (75)= 75 MOD 17 =7	比较1次

散列表H为：

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
17	87	53	70	0	0	0	75	25	59	61	63	12	98	14	0	46	67
1	3	1	2	0	0	0	1	1	2	1	3	1	1	1	0	4	2

ASL(14)=1.71

输入查找的值：70<CR>

查找值：ha[3]=70!

习 题 8

1. 单项选择题：

(1) 顺序查找法适合于存储结构为_____的线性表。

A) 散列存储　　B) 顺序存储或链接存储

C) 压缩存储　　D) 索引存储

(2) 采用顺序查找方法查找长度为n的线性表时，每个元素的平均查找长度为_____。

A) n　　B) n/2　C) (n+1)/2　　D (n－1)/2

(3) 采用折半查找方法查找长度为n的线性表时，每个元素的平均查找长度为_____。

A) $O(n^2)$　　B) O(n lb n)　C) O(n)　　D) O(lb n)

(4) 有一个有序表为{1，3，9，12，32，41，45，62，75，77，82，95，100}，当二分查找值为82的结点时，_____次比较后查找成功。

A) 1　　B) 2　　C) 4　　D) 8

(5) 设哈希表长m=14，哈希函数H(key)=key%11。表中已有4个结点：

Addr(15)=4

Addr(38)=5

Addr (61)=6

Addr(84)=7

其余地址为空

如用二次探测再散列处理冲突，关键字为49的结点的地址是_____。

A) 8　　B) 3　　C) 5　　D) 9

2. 填空题：

(1) 顺序查找法的平均查找长度为_____；二分查找法的平均查找长度为_____；分块查找法(以顺序查找确定块)的平均查找长度为_____；分块查找法(以二分查找确定块)的平均查找长度_____；哈希表查找法采用链接法处理冲突时的平均查找长度为_____。

(2) 在多种查找方法中，平均查找长度与结点个数n无关的查找方法是_____。

(3) 二分查找(折半查找)的存储结构仅限于_____且是_____。

(4) 在分块查找方法中，首先查找_____，然后再查找相应的_____。

(5) 假设在有序线性表A［1··20］上进行二分查找，则比较一次查找成功的结点数为_____，比较两次查找成功的结点数为_____，比较三次查找成功的结点数为_____，比较四次查找成功的结点数为_____，则比较五次查找成功的结点数为_____，平均查找长度为_____。

3. 设有一组关健字{19, 01, 23, 14, 55, 20, 84, 27, 68, 11,10，77}，所用哈希函数为

$$H(k_i)=k_i \ \text{MOD}\ 13$$

采用开放地址法的线性探测再散列方法解决冲突，试在0～18的散列地址空间中对该关键字序列构造哈希表。

第9章 排 序

9.1 排序的基本概念

排序(Sorting)是把一个无序的数据元素序列按某个关键字进行有序(递增或递减)排列的过程。排序中经常把数据元素称为记录(Record)。把记录中作为排序依据的某个数据项称为排序关键字，简称关键字(Key)。

排序时选取哪一个数据项作为关键字，应根据具体情况而定。例如，表9.1为某次考试成绩表，表中每个学生的记录包括考号、姓名、三门课成绩以及这三门课的总分。

表9.1 学生成绩表

考号	姓名	英语	数学	微机	总分
010	李 明	89	80	91	260
202	王小萌	76	92	68	236
103	张 炎	78	85	77	240
204	赵 沼	94	82	98	274
011	王 丽	77	89	90	256

以“考号”作为关键字排序，可以快速查找到某个学生的成绩，因为考号可以惟一识别一个学生的记录。若想以“总分”排列名次，就应把“总分”作为关键字对成绩表进行排序。

待排序的记录可以是任意的数据类型，其关键字可以是整型、实型、实符型等基本数据类型，通过排序可以构造一种新的按关键字有序的排列。如果待排序的记录序列中存在关键字相同的记录，例如，有一序列(10,45,12,32,45,78)，其中45区别于45。排序前45在序列中的位置先于45，排序后的新序列若为(10,12,32,45,45,78)，45的位置仍先于45，则称这种排序方法是稳定的；反之，如果数据序列变为(10,12,32,45,45,78)，此排序方法是不稳定的。

排序的方式根据待排记录数量不同可以分为两类：

(1) 在排序过程中，只使用计算机的内存储器存放待排序的记录，称为内部排序。内部排序用于排序的记录个数较少时，全部排序可在内存中完成，不涉及外存储器，因此，排序速度快。

(2) 当排序的记录数很大时，全部记录不能同时存放在内存中，需要借助外存储器，也就是说排序过程中不仅要使用内存，还要使用外存，记录要在内、外存之间移动，这种排序称为外部排序。外部排序运行速度较慢。

本章只讨论内部排序，不涉及外部排序。

内部排序的方法很多，但不论哪种排序过程，通常都要进行两种基本操作：

(1) 比较两个记录关键字的大小。

(2) 根据比较结果，将记录从一个位置移到另一个位置。

所以，在分析排序算法的时间复杂度时，主要分析关键字的比较次数和记录的移动次数。

特别需要说明的是：本章介绍的排序算法都是采用顺序存储结构，即用数组存储，且按关键字递增排序。函数中记录类型及数组结构定义如下：

```
# define Maxsize 100

typedef struct Recordnode

{  keytype key;

  elemtype data;

} Recordnode;

Recordnode r[Maxsize]
```

这里的keytype和elemtype可以是任何相应的数据类型，如int，float及char等，在算法中规定keytype和elemtype默认为int型。

9.2 插入排序

9.2.1 直接插入排序

直接插入排序是最简单的排序方法之一。其基本思想是：在有序区中进行顺序查找，以确定插入的位置，然后移动记录腾出空间，以便插入关键字相应的记录。

例9.1　设有6个待排序的记录，它们排序的关键字序列为{20，6，15，7，3，6}。在顺序查找中，为了防止循环变量越界，在有序区前段增设了一个“岗哨”r[0]，暂存当前待插入的记录。具体的排序过程如图9.1所示。

	r[0]	r[1]	r[2]	r[3]	r[4]	r[5]	r[6]
初始序列:	岗哨	20	6	15	7	3	6
第一趟:	6	[6	20]	15	7	3	6
第二趟:	15	[6	15	20]	7	3	6
第三趟:	7	[6	7	15	20]	3	6
第四趟:	3	[3	6	7	15	20]	6
第五趟:	6	[3	6	6	7	15	20]

图9.1 直接插入排序示例

从例9.1看出：① 直接插入排序是从第二个记录开始的，对记录数为6的序列，需要进行5趟排序才能完成；② 6和6的相对位置没有变化。因此，直接插入排序是稳定的排序方法。

直接插入排序算法描述如算法9.1。

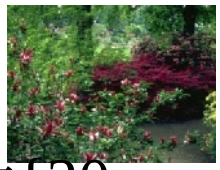
```
/*算法描述9.1  直接插入排序*/
Insertsort(Recordnode r[],int n)        /*对r[1]～r[n]进行插入排序*/
        {int  i，  j;
       for(i=2;i<=n;i++)
         { r[0]=r[i];                   /*待插入记录送入岗哨*/
        j=i-1;
       while (r[0].key<r[j].key)
          { r[j+1]=r[j];
               j--;
           }
        r[j+1]=r[0]                     /*r[0]插入合适位置*/
       }
          }
```

算法分析：直接插入排序的比较次数取决于原记录序列的有序程度。如果原始记录的关键字正好为递增顺序时，比较次数最少为n－1次；如果为递减顺序时，比较次数最多，为(n－1)(n+2)/2次，因此，直接插入排序的时间复杂度为$O(n^2)$。

9.2.2 折半插入排序

折半插入排序是直接插入排序的改进，它提高了查找插入位置的速度。基本思想是：若要插入一个记录，可先对有序的记录序列折半，确定要插入记录的位置在前一半序列，还是在后一半序列中，不断地对有序的记录序列折半，最后找到合适的插入位置。

具体过程为：设置一个头指针low和一个尾指针high，它们分别指向待查区间的头和尾，用low和high的值可以计算出中间记录的位置，用mid指示，mid=(low+high)/2。每次将待查记录的关键字r[i].key与r[mid].key作比较；若r[i].key>r[mid].key，令low=mid+1，继续查找；若r[i].key<r[mid].key，令 high=mid－1，继续查找，依次类推，直到查找到适当插入位置。



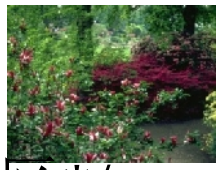
例9.2　设待排序的记录共有5个，它们的关键字分别为{20，6，15，7，3}，经四趟插入，完成全部排序工作，如图9.2所示。

	r[0]	r[1]	r[2]	r[3]	r[4]	r[5]
初始序列:	岗哨	[20]	6	15	7	3
第一趟(mid　1)	6	[6	20]	15	7	3
第二趟(mid　1)	15	[6	15	20]	7	3
第三趟(mid　2,1)	7	[6	7	15	20]	3
第四趟(mid　2,1)	3	[3	6	7	15	20]

图9.2 折半插入排序示例

折半插入排序算法描述如下：

```
/*算法描述9.2  折半插入排序*/
Binsort (Recordnode r[],int n)
{
int i,j,low,high,mid;
   for(i=2;i<=n;i++)
  { r[0]=r[i];
    low=1;high=i-1;          /*设置查找区间上、下界*/
     while(low<high)
          {m=(low+high)/2;
```



```
            if(r[0].key<r[m].key) high=mid-1;    /*插入点在前半区*/
          Else low=mid+1;                        /*插入点在后半区*/
        }
        for(j+1;j>=low;j--)
    r[j+1]=r[j];                                 /*记录后移*/
         r[low]=r[0];                                  /*插入*/
        }
    }
```

算法分析：折半插入排序的比较次数比直接插入排序的少，而移动次数相同，因此，总的时间复杂度仍为$O(n^2)$，另外，折半插入排序也是一种稳定的排序方法。

9.2.3 希尔排序

希尔排序(Shell's Sort)也称为“缩小增量法排序”，是由D.L.Shell对直接插入排序进行改进后提出来的。其基本思想是：不断地把待排序的一组记录按间隔值分成若干小组，分别进行组内直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

对间隔值(用d表示)的取法有多种，希尔提出的方法是：$d_1=[n/2]$，$d_{i+1}=[d_i/2]$，最后一次排序时间的间隔值必须为1，其中n为记录数。

例9.3 设待排序的记录数n为8，它们的关键字分别为{47，55，10，40，15，94，5，70}，间隔值序列取4、2、1。希尔排序过程如图9.3所示。

	r[1]	r[2]	r[3]	r[4]	r[5]	r[6]	r[7]	r[8]
初始关键字:	47	55	10	40	15	94	5	70

(1)第一趟排序(d　4):分4组, 组内采用直接插入排序法。

	{47				15}			
		{55				94}		
			{10				5}	
				{40				70}
结果:	15	55	5	40	47	94	10	70

(2)第二趟排序(d　2):分2组, 组内采用直接插入排序法。

	{15		5		47		10}	
		{55		40		94		70}
结果:	5	40	10	55	15	70	47	94

(3)第三趟排序(d=1):整个数据合成1组, 采用直接插入排序法。

	{5	40	10	55	15	70	47	94}
最后结果:	5	10	15	40	47	55	70	94

图9.3 希尔排序示例

希尔排序算法描述如下:

```
/*算法描述9.3  希尔排序*/
Shellsort (Recordnode r[],int n)
 { int i,j,d;
     struct recordnode x;
     d=n/2;                                  /*设置初值*/
     while(d>1)
       {for(i=d+1;i<=n;i++)
         {j=i-d;
            while(j>=0)
```

```
            if (r[j].key>r[j+d].key)
        {x=r[j];                    /*将r[j]与r[j+d]进行交换*/
         r[j]=r[j+d];
         r[j+d]=x;
         j=j-d;
              }
     else  j=0;
  }
  d=d/2;                            /*减小间隔值*/
  }
  }
```

该算法通过三重循环来实现，外循环由不同间隔值d来控制，直到d=1为止，中间循环是在某个d值下对各组进行排序，用变量i控制。

可以看出，希尔排序实际上是对直接插入排序的一种改进，它的排序速度一般要比直接插入排序快。希尔排序的时间复杂速度取决于所取的间隔，一般认为是O(n lb n)。希尔排序是一个较复杂的问题，并且它还是一种不稳定的排序方法。

9.3 交换排序

9.3.1 冒泡排序

冒泡排序是一种简单常用的排序方法。其排序思想是：通过相邻记录关键字间的比较和交换，使关键字最小的记录像气泡一样逐渐上浮。比较可以采用不同的方法，本算法是从最下面的记录开始，对两个相邻的关键字进行比较并且使关键字较小的记录换至关键字较大的记录之上，使得经过一次冒泡后，关键字最小的记录到达最上端，接着，再在剩下的记录中找关键字最小的记录，把它换到第二个位置上。依次类推，一直到所有记录都有序为止。一般情况下，记录数为n，需要做n－1次冒泡。

例9.4 设待排记录的关键字分别为{37，19，90，64，13，49，20，40}，进行冒泡排序的具体过程如图9.4所示。

初始(n=8)	i 1	i 2	i 3	i 4	i 5	i 6	i 7
r[1]	37	13	13	13	13	13	13
r[2]	19	37	19	19	19	19	19
r[3]	90	19	39	20	20	20	20
r[4]	64	90	20	37	37	37	37
r[5]	13	64	90	40	40	40	40
r[6]	49	20	64	90	49	49	49
r[7]	20	49	40	64	90	64	64
r[8]	40	40	49	49	64	90	90

图9.4 冒泡排序示例

从排序的过程看，记录数为8，需要做7次冒泡，但实际进行到第5次冒泡时，整个记录已经有序了，因此，不需要再进行6、7次冒泡了，也就是说，在某次比较过程中，如果设有交换记录，则排序可提前结束。这点在算法中给予考虑，可节省排序时间，为此在算法中设置一个变量F来监视排序情况，F=1时表示有记录交换，F=0时无记录交换。

冒泡排序算法描述如下:

```
/*算法描述9.4  冒泡排序*/
Bubblesort (Recordnode r[]，int n)
{int i，j，F；
 F=1；
 For(i=1;i<=n-1&&F==1；i++)
{F=0;
 for(j=n;j>=i+1;j--)
    if(r[j].key<r[j-1].key)
      {F=1;
       r[0]=r[j];                          /*r[0]用于中转*/
       r[j]=r[j-1];
       r[j-1]=r[0];
      }
}
}
```

算法分析：冒泡排序是一种稳定的排序方法，算法的执行时间与原始记录的有序程度有很大关系，如果原始记录已经是有序排列时，比较次数为n－1次，交换次数为0；如果原始记录是“逆序”排列时，则总的比较次数为n*(n－1)/2，交换次数为3*n*(n－1)/2。所以，算法的平均时间复杂度为$O(n^2)$。

9.3.2 快速排序

快速排序又叫分区交换排序，它是对冒泡排序的一种改进。其排序的基本思想是：取记录序列中一个合适的关键字(通常选取序列中的第一个)，以此关键字取对应的记录r_i作为基准，把一个序列分割成两个独立的子序列，使得基准位置前的所有记录的关键字都小于r_i key，而基准位置后的所有记录的关键字都大于r_i.key。这里把这样的一次过程称作一次快速排序，在第一次快速排序中，确定了所选取的基准记录r_i在序列中的最终排列位置，同时也把剩余的记录分成了两个序列，然后对每个序列再进行分割，重复上述过程，直到所有记录全部排好序为止。

下面具体介绍一次快速排序的过程：设置两个指示器i，j分别表示当前待排记录序列中的第一个记录和最后一个记录位置；将第一个记录作为基准关键字放到变量x中，使它所处的位置腾空，然后从序列的两端开始逐步向中间扫描:

(1) 在序列的右端扫描时，从序列的当前右端j处开始，把记录r_i.key与基准关键字x.key进行比较，若前者大于后者，令j=j－1，继续进行比较，直到前者小于或等于后者，此时，若j>i，则将r_j放到腾空的位置上，使r_j腾空，同时使i+1。然后进行(2)。

(2) 在序列的左端扫描时，从序列的当前左端i处开始将r_i.key与x.key进行比较，若前者小于后者，令i=i+1，继续向右扫描，直到前者大于等于后者，此时，若i<j，则将r_i放到记录r_j中(注：r_j已经腾空)，使j减1。然后继续进行(1)。

(1)和(2)交替反复执行，当i≥j时，扫描结束。

例9.5 设待排记录的关键字分别是{37，19，90，64，13，49，20，40}，第一次快速排序过程如图9.5所示。

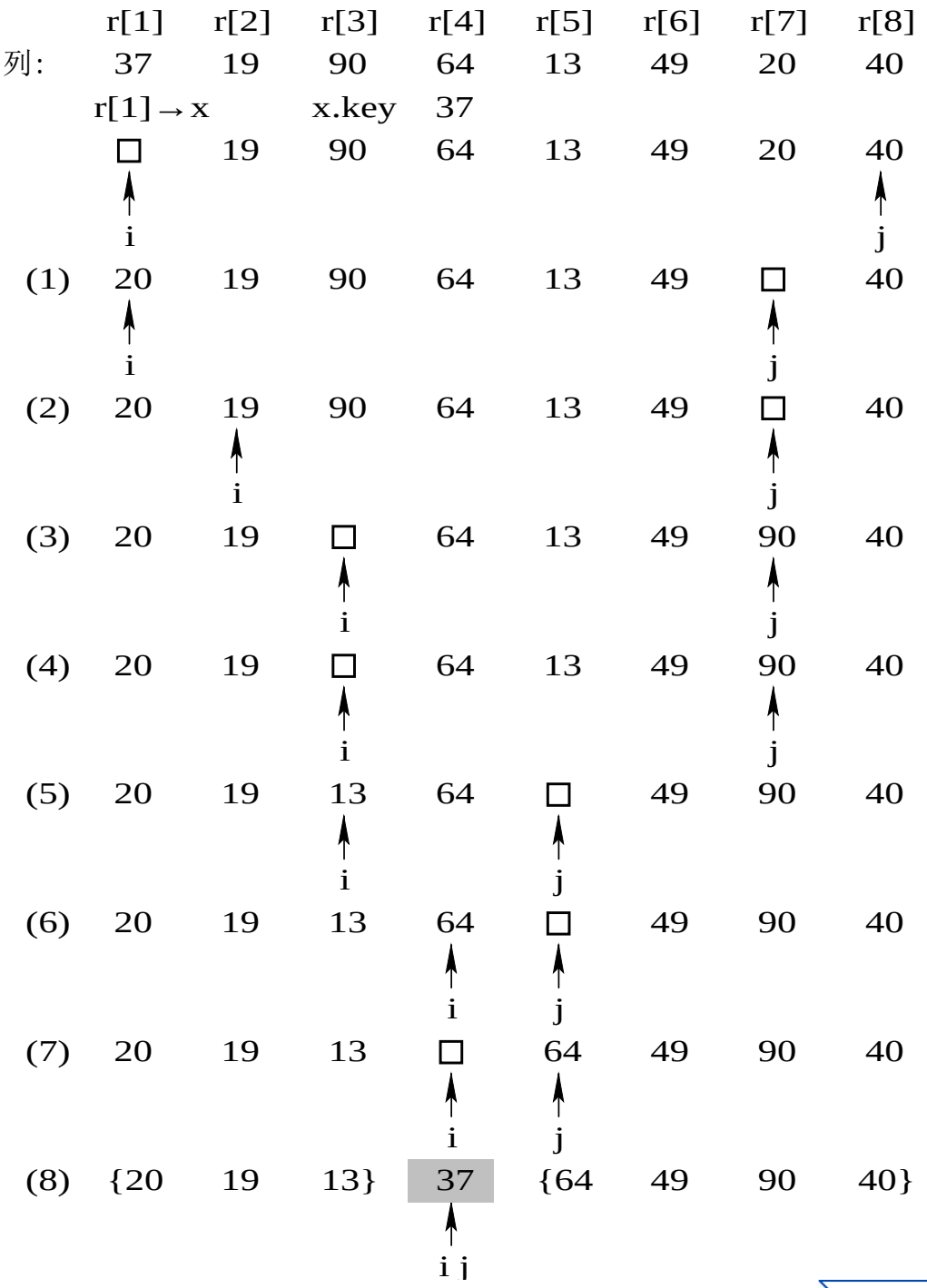


图9.5 快速排序示例

第一次快速排序后，用同样的方法对产生的两个子序列继续进行快速排序，下面给出整个排序过程：

初始关键字序列：{37 19 90 64 13 49 20 40}

第一次排序结果：{20 19 13} 37 {64 49 90 40}

第二次排序结果：{13 19} 20 37 {40 49} 64 {90}

第三次排序结果： 13 {19} 20 37 40 {49} 64 90

最后结果： 13 19 20 37 40 49 64 90

快速排序算法描述如下：

```
/*算法描述9.5 快速排序*/
Quicskort(Recordnode r[]，int low，int high)
        /*low和high为记录序列的下界和上界*/
{int i，j;
 struct Recordnode x;
 i=low;
 j=high;
 x=r[low];
 while(i<j)
 { /*在序列的右端扫描*/
      while(i<j&&r[j].key>=x.key)
```

```
                    j--;
                if(i<j) {r[i]=r[j];i++;}
                /*在序列的左端扫描*/
                while(i<j&&r[i].key<x.key)
                    i++;
                if(i<j) {r[j]=r[i];j--;}
    }
        r[i]=x;
          /*对序列进行快速排序，使用递归*/
        if(low<i)  Quicksort(r,low,i-1);
        if(i<high) Quicksort(r,j+1,high);
          }
```

在上述算法程序中，待排记录序列顺序存放在r[low]，r[low+1]，…，r[high]中，另外，该程序采用了递归的方法，快速排序也可用非递归的方法实现，有兴趣的读者不妨将该函数改写为非递归的形式。

快速排序并没有每次比较都进行交换，只是移动一个数据，待到所选出的元素找到合适的位置才存入，因此，排序速度快。

算法分析：快速排序的执行时间和基准记录的选取有关，若基准记录的关键字能把当前记录从中间分成两个相等的子区间，则运行效率最高；若基准记录的关键字在记录排序的首或尾时，即只有一个子区间，该算法就退化成冒泡排序法，最好和最坏情况的平均时间复杂度为O(n lb n)。快速排序也是一种不稳定的排序方法。

9.4 选择排序

9.4.1 直接选择排序

直接选择排序是一种简单直观的排序方法，它的做法是：从待排序的所有记录中，选取关键字最小的记录，把它与第一个记录交换，然后在其余的记录中再选出关键字最小的记录与第二个记录交换，如此重复下去，直到所有记录排序完成。

例9.6 设待排记录序列的关键字为{11，50，16，70，1，30}，直接选择排序过程如图9.6所示。

n　6	r[1]	r[2]	r[3]	r[4]	r[5]	r[6]
初始关键字:	11	50	16	70	1	3
第一次排序:	1	50	16	70	11	3
第二次排序:	1	3	16	70	11	50
第三次排序:	1	3	11	70	16	50
第四次排序:	1	3	11	16	70	50
第五次排序:	1	3	11	16	50	70
最后结果:	1	3	11	16	50	70

图9.6 直接选择排序示例

实现以上过程的算法如下：

```
/*算法描述9.6  直接选择排序*/
Selectsort(Recordnode r[],int n)
{int i,j,k;
  struct  Recordnode  x;
  for(i＝1；  I<＝n－1；  i++)
  {k＝i;
  for(j＝i+1；  j<＝n；  j++)
   if (r[ j ].key<r[ k ].key)  k=j;   /*记录最小数的位置*/
   if (k!=i)
     {x=r[i]；  r[i]=r[k];r[k]=x;}
}
}
```

9.4.2 堆排序

堆排序是借助于一种称为堆的完全二叉树结构进行排序的，排序过程中，将向量中存储的数据看成是一棵完全二叉树的顺序存储结构，利用完全二叉树中的父结点和孩子结点之间的内在关系来选择关键字最小的记录。

具体做法是：把待排序的记录存放在数组r[1‥n]中，将r看作一棵二叉树，每个结点表示一个记录，第一个记录r[1]作为二叉树的根，以后各记录r[2]，…，r[n]依次逐层从左到右顺序排列，构成一棵完全二叉树，任意结点r [i]的左孩子是r[2i]，右孩子是r[2i+1]，双亲是r[i/2]。

对这棵完全二叉树的结点进行调整，使各结点的关键字满足下列条件：

$$r[i] \le r[2i] \text{ 且 } r[i] \le r[2i+1]$$

即每个结点的值均大于或小于它的两个子结点的值，称满足这个条件的完全二叉树为堆树。显然在这个堆树中根结点的关键字最小，这种堆被称为“小根堆”，如图9.7所示。

各结点的值满足$r[i] \ge r[2i]$并且$r[i] \ge r[2i+1]$的堆，称为“大根堆”，如图9.8所示。大根堆中根结点的关键字值最大。

1	2	3	4	5	6	7
12	25	16	60	65	38	72

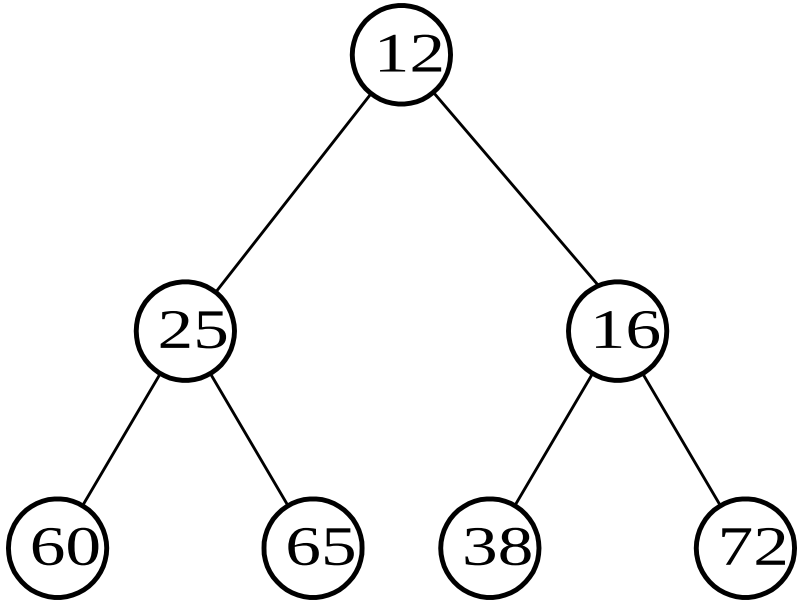


图9.7 小根堆示例

1	2	3	4	5	6	7
95	30	73	10	21	19	27

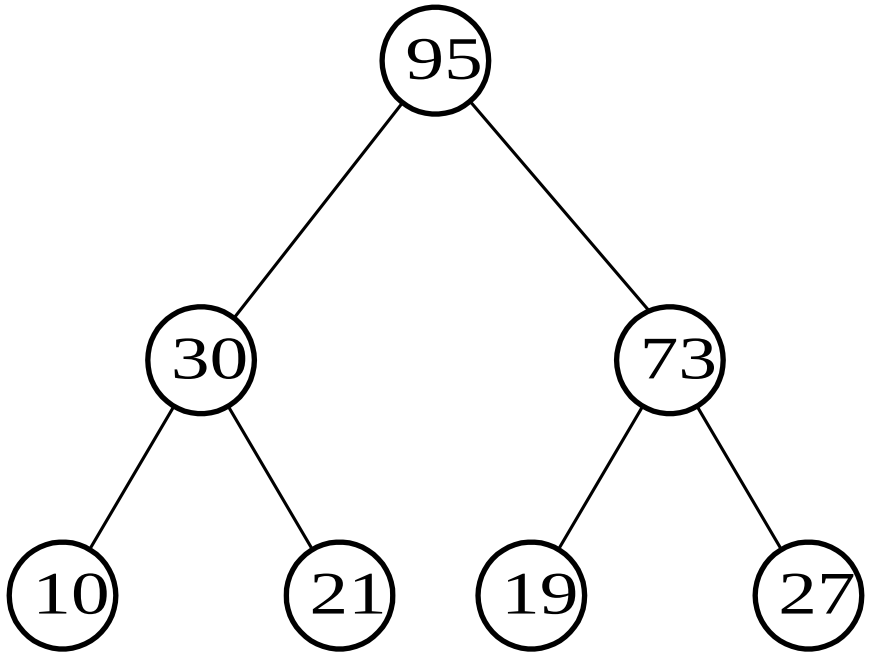


图9.8 大根堆示例

在堆中，根结点又被称为堆顶元素，当把二叉树转换成大根堆后，堆顶元素最大，把堆顶元素输出，重新调整二叉树的剩余结点，使其成为一个新堆，再输出堆顶元素，便可求得次最大值，如此反复进行，就可得到一个有序序列，这就是利用大根堆排序的基本方法。读者不难推导出小堆根的排序方法。

完成堆排序必须解决两个问题：

(1) 如何将原始记录序列构造成一个堆，即建立初始堆。

(2) 输出堆顶元素后，如何将剩余记录调整成一个新堆。

因此，堆排序的关键是构造初始堆，其实构造初始堆的过程就是将待排元素序列对应的完全二叉树调整形成一个堆，所以解决问题的关键在于如何调整元素间的关系使之形成初始堆。下面举例说明。

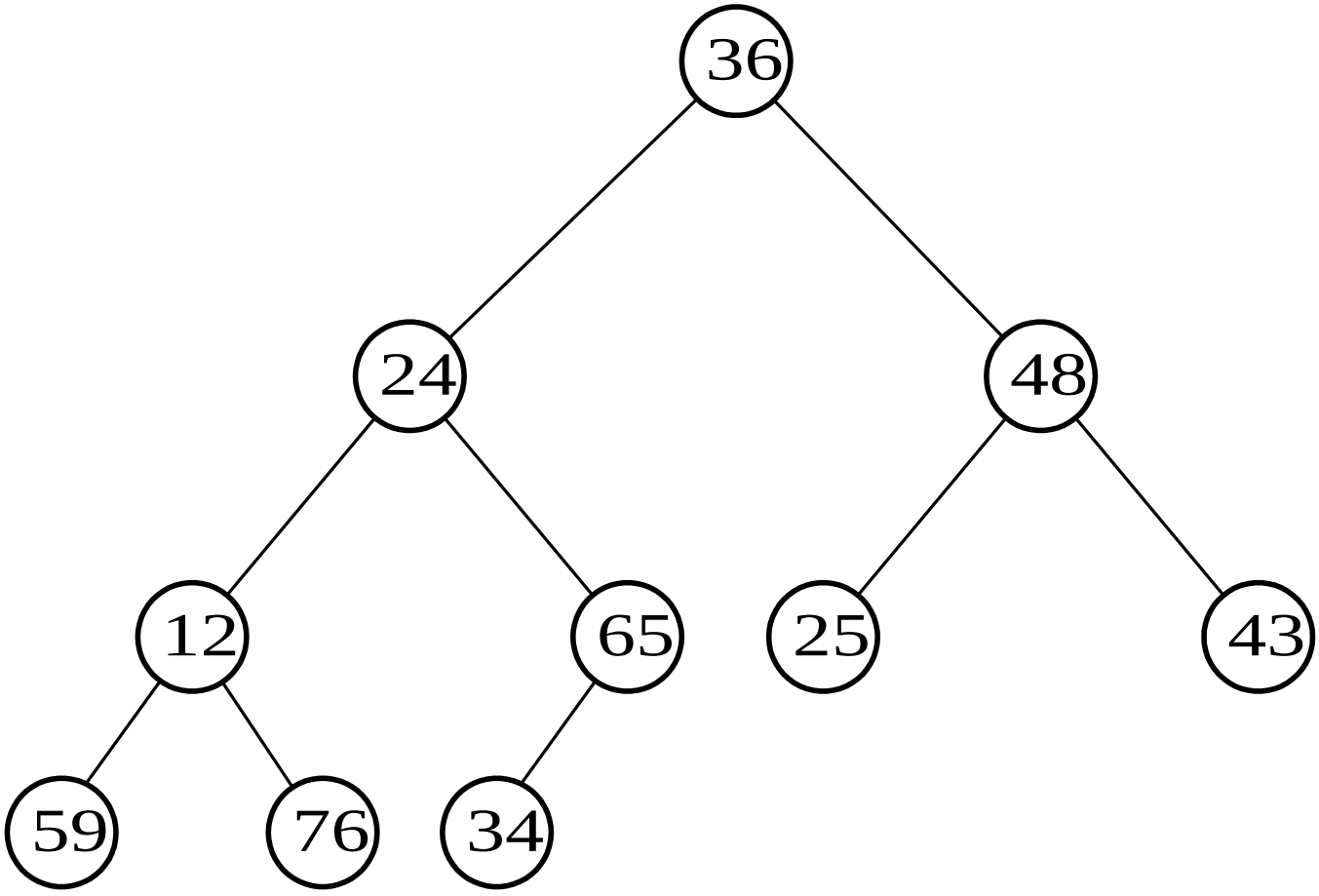


图9.9 原始数据的二叉树顺序存储形式

由二叉树的性质得知，二叉树中序号最大的一个非终点是[n/2]，即图中的5号结点65，序号最小的列终结点是序号为1的结点，即根结点36，对这些结点需一一进行调整，使其满足堆的条件。调整过程为：首先把5号结点元素65与其两个孩子中值大者进行比较，由于只有1个左孩子34，故只和34比较，因为65>34故不必交换，则以65为根的子树即为堆，接着用相同的步骤对第4个结点12进行调整，直至第1个结点36。如果在中间调整过程中，由于交换破坏了以其孩子为根的堆，则要对破坏了的堆进行调整，依次类推，直到父结点大于等于左、右孩子的元素结点或者孩子结点为空的元素结点。当这一系列调整过程完成时，图9.10所示的二叉树即成为一个堆树，这个调整过程也叫做“筛选”。

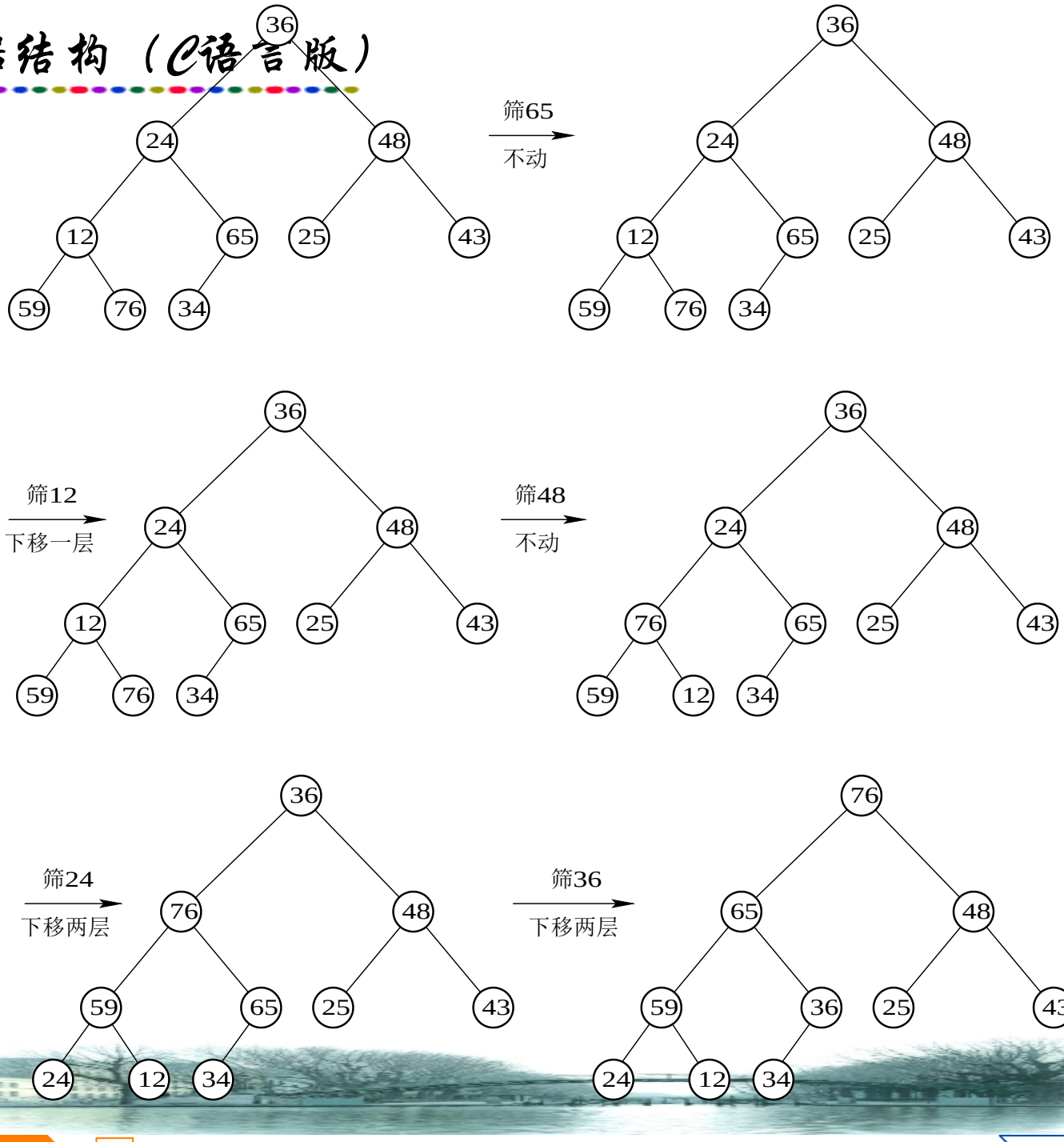


图9.10 建立初始堆示例

下面给出建立初始堆的筛选过程及其算法。

```
/*算法描述9.7  建立初始堆*/
sift (Recordnode  r[], int  L, int  m)
 /*对有n个元素的数组r中的第L个元素进行筛选*/
{int i, j ;
struct Recordnode  x;
i＝L；  j=2*i;              /*r[j]是r[i]的左孩子*/
x=r[i];                              /*把筛选结点的值存入变量 x中*/
while  (j<=m)
      { if (j<m && r[j].key<r[j+1].key)  j++;
```

```
 /*左孩子小于右孩子，沿右子树筛选*/
if(x<r[j])
 { r[i]=r[j];                /*大孩子上移到双亲的位置*/
   i=j;                      /*往下搜索，令r[j]为根结点*/
   i=2*i;                    /*下沉一层，求出左孩子*/
 }
else                         /*根不小于它的孩子时退出循环*/
   j=m+1;
}
r[i]=x;
}                            /*被筛选的结点放入最终位置*/
```

排序过程为：先输出堆顶元素76，把它放到原数组的最后位置，而原数组最后一个单元存放的是34，为了不破坏数据，则把76与34交换，即r[1]与r[n](n为10)交换，这时r[n]为最大，接着对r[1]与r[n－1]进行筛选又得到新堆，此时新堆的r[1]为当前最大的元素，再把r[1]与r[n－1]对调，使r[n－1]为次最大，这样，经过n－1次对调、筛选，数组r中的所有元素均按升序排列，堆排序全部完成，如图9.11所示。

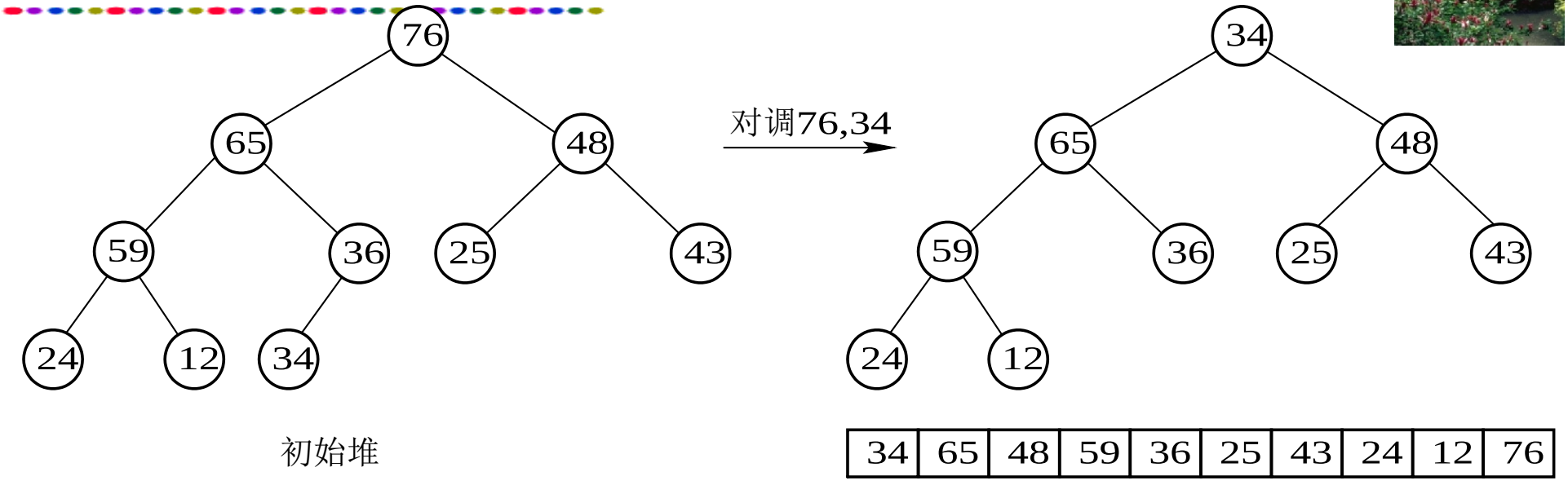


图9.11 堆排序过程示例

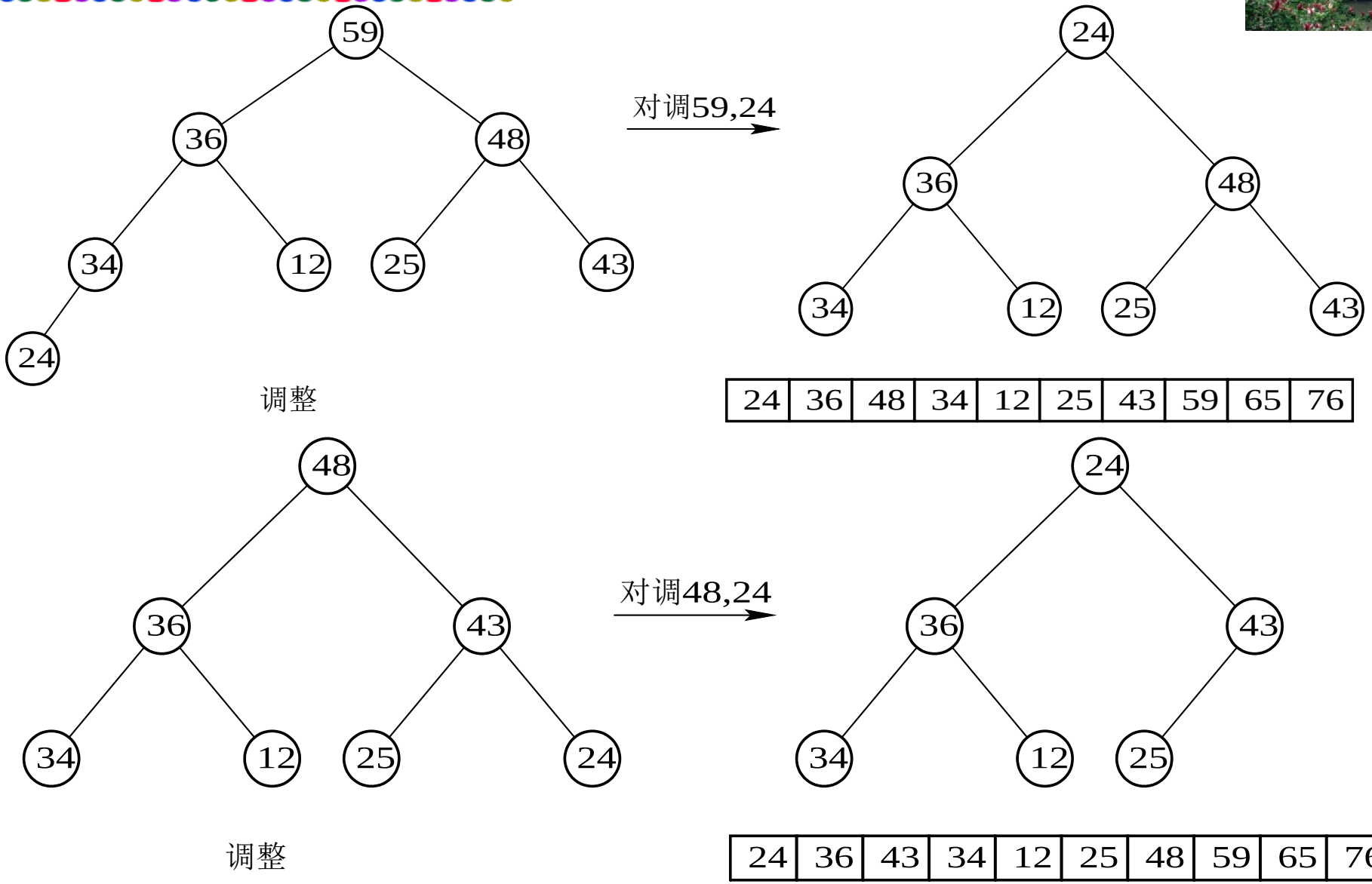


图9.11 堆排序过程示例

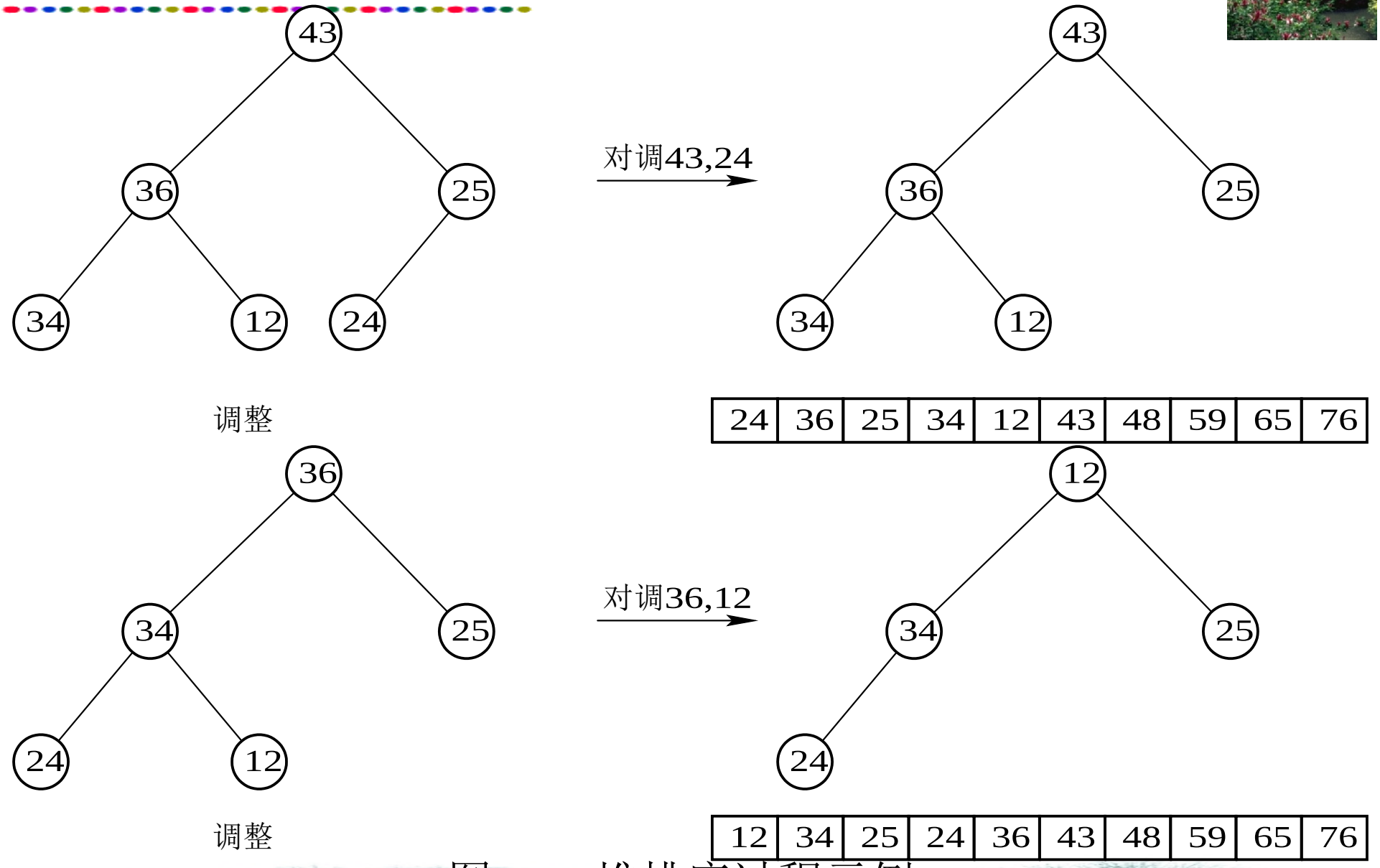


图9.11 堆排序过程示例

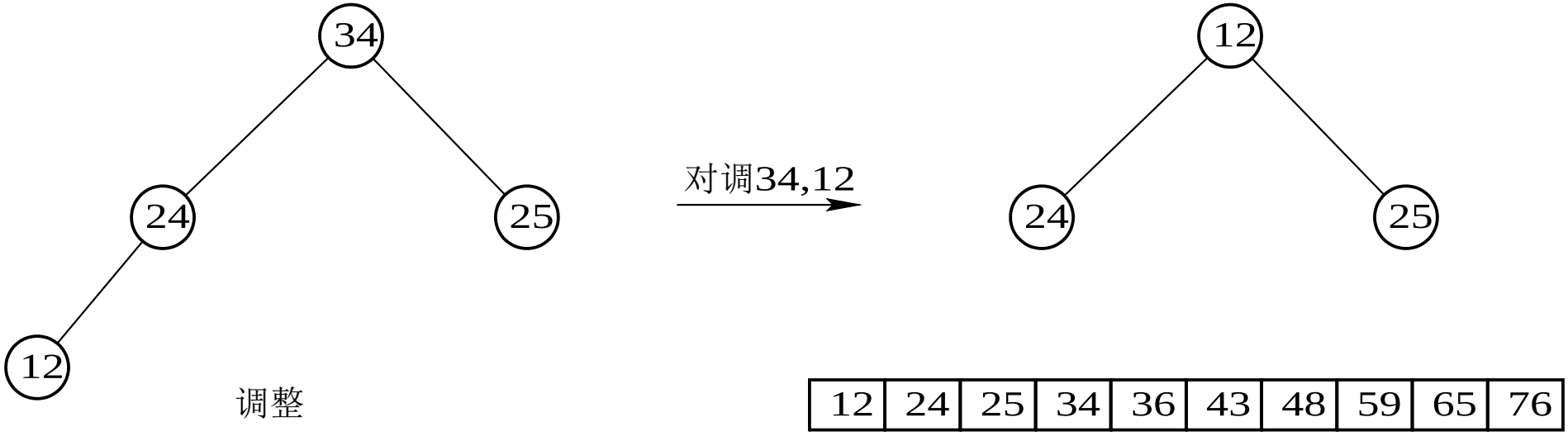


图9.11 堆排序过程示例

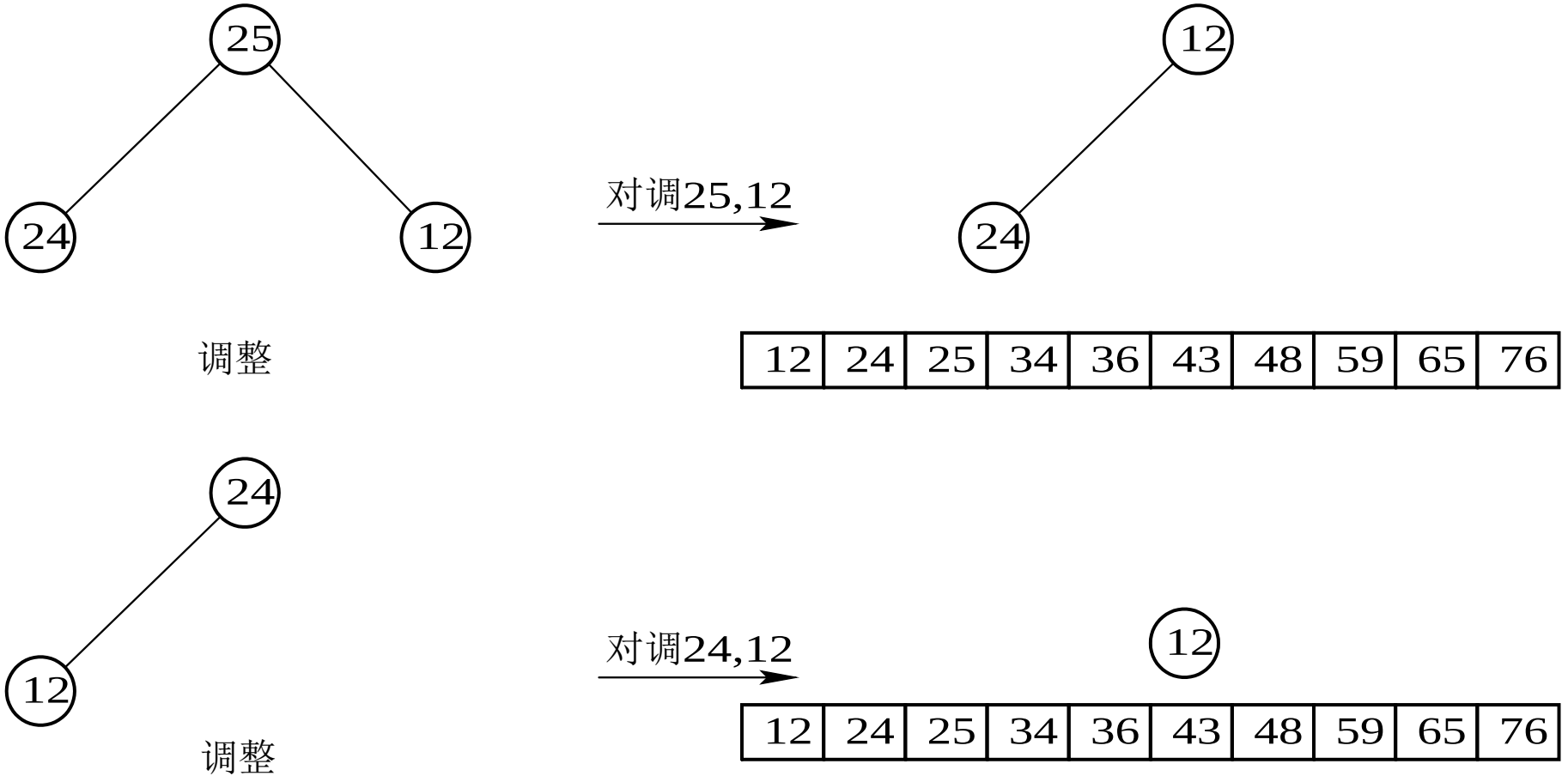


图9.11 堆排序过程示例(续二)

下面给出整个排序过程及算法描述。(说明：已经有序的子树部分省去。)

堆排序算法描述如下：

```
/*算法描述9.8  堆排序*/
heapsort (Recordnode  r[],int n)
{int i,k;
struck  Recordnode  x;
for(i=n/2;i>0;i--)              /*建立初始堆*/
sift(r,i,n);
for(i=n;i>1;i--)                /*进行n－1次循环，完成堆排序*/
```

```
{ x=r[1];    /*将第一个元素与当前区间的最后一个元素对调*/
 r[1]=r[i];
 r[i]=x;
sift(r,1,i-1);      /*调用筛选子函数*/
}
}
```

从堆排序的算法知道，堆排序所需的比较次数是建立初始堆与重新建堆所需的比较次数之和，其平均时间复杂度和最坏的时间复杂度均为O(n lbn)。它是一种不稳定的排序方法。

9.5 内部排序方法的比较

一个好的排序方法所需要的比较次数和占用存储空间应该要少。从表9.2可以看出，每种排序方法各有优缺点，不存在十全十美的排序方法，因此，在不同的情况下可选择不同的方法，选取排序方法时，一般需要考虑以下几点：

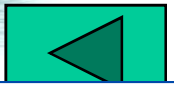
(1) 算法的简单性。它分为简单算法和改进后的算法，简单算法有直接插入、直接选择和冒泡法，这些方法都简单明了。改进的算法有希尔排序、快速排序和堆排序等，这些算法比较复杂。

(2) 待排序的记录多少。记录数越少越适合简单算法，记录数越多越适合改进算法，这样，效率可以明显提高。

(3) 记录本身所带的信息量的大小。记录信息量越大，用的字节越多，那么移动这些记录需花费的时间也越多，所以选择算法应尽量避免选用移动次数较多的算法。

表9.2 7种排序方法的比较

	排序方法	平均时间	最坏情况	辅助空间	稳定性
	直接插入排序	O (n^2)	O (n^2)	O(1)	√
	折半插入排序	O(n^2)	O (n^2)	O(1)	√
	希尔排序	O($n^{1.3}$)	O(n^2)	O (1)	×
	冒泡排序	O (n^2)	O (n^2)	O (1)	√
	快速排序	O(n lb n)	O(n^2)	O(lb n)	×
	直接选择排序	O(n^2)	O (n^2)	O(1)	√
	堆排序	O(n lb n)	O(n lb n)	O(1)	×



9.6 实习：排序算法的实现——学生成绩管理

给出n个学生的考试成绩表，每条信息由姓名与分数组成，试设计一个算法：① 按分数高低次序，求出每个学生在考试中获得的名次，分数相同的为同一名次；② 按名次列出每个学生的姓名与分数。

算法实现：下面给出的是用直接选择排序算法实现的C语言程序，仅供参考。在此基础上，读者可以尝试用直接插入、Shell排序、冒泡排序、快速排序等排序算法实现对该问题的求解。

```
#include  <stdio.h>
#define   maxsize  100
typedef   struct  Recordnode
{char     name[8];
 float    score;
}Recordnode;
Recordnode   r[maxsize];
void  selectsort  (Recordnode  r[], int n)                    /*函数说明*/
main()
{int  num,i,j,n;
   printf("\n请输入学生成绩:\n");
```

```
for (i=1;i<=n;i++)
{printf("姓名: ");
scanf("%s",r[i].name);
scanf("%4f",&r[i].score);
}
void  selectsort(r , n);                          /*调用selectsort子函数*/
num=1;                                            / *变量num表示名次*/
printf("%4d%s%4f\n",num,r[1].name,r[1].score);
for (i=2;i<=n;i++)                                /*按名次打印成绩表*/
{if (r[i].score < r[i-1].score)
      num=num+1;
```

```
printf("%4d%s%4f\n",num,r[ i ].name,r[ i ].score);
}
}
void selectsort (Recordnode r[] , int n)                    /*排序子函数*/
{int i , j , k ;
struct  Recordnode  x;
for(i=1;i<=n-1;i++)
      {k=i;
```

```
for( j = i + 1 ; j < = n ; j + +)
 if (r [ j ].score>r[ k ].score)  k=j;
if (k != i )
{ x = r [ i ];
  r[ i ]=r[ k ];
  r[ k ]=x;
}
}
}
```

习 题 9

1. 选择题

(1) 在所有排序方法中，关键字的比较次数与记录的初始排列无关的是___。

A) Shell排序　　B) 冒泡排序　C) 插入排序　D) 选择排序

(2) 直接插入排序和冒泡排序的时间复杂度为_____，若初始数据有序(即正序),则时间复杂度为___。

A) O(n)　　B) O(lb n)　C) O(n lb n)　D) $O(n^2)$

(3) 排序的方法有多种，____法从未排序序列中依次取出元素与已排序序列(初始时为空)中元素作比较，将其放入已排序序列中的正确位置上；____法从未排序序列中挑选元素，并将其依次放入已排序序列的一端；____法对序列中相邻元素进行一系列比较，当被比较的两个元素逆序时就进行交换。

A) 冒泡排序　　B) 插入排序　　C) 快速排序　　D) 选择排序

(4) 对n个元素的序列进行冒泡排序时, 最少的比较次数是__。

A) n　　B) n－1　　C) 0　　D) 1

(5) 一组记录的关键字为(45，80，55，40，42，85)，则利用堆排序的方法建立的初始堆为___。

A) (80，45，55，40，42，85) B) (85，80，55，40，42，45)

C) (85，80，55，45，42，40) D) (85，55，80，42，45，40)

(6) 用某种排序方法对线性表(25，84，21，47，15，27，68，35，20)进行排序时，元素序列的变化情况如下:

1 (25，84，21，47，15，27，68，35，20)

2 (20，15，21，25，47，27，68，35，84)

3 (15，20，21，25，35，27，47，68，84)

4 (15，20，21，25，27，35，47，68，84)

则所采用的排序方法是__。

A) 插入排序法　B) 选择排序法　C) 快速排序法　D) 堆排序法

2. 已知序列(10，18，4，3，6，12，1，9，18，8), 采用希尔排序法排序, 写出每一趟的结果。

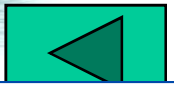
3. 已知序列(45，87，12，56，67，6，90，39，83), 采用快速排序法排序, 写出每一趟的结果。

4. 判别以下序列是否为堆，如果不是，则把它调整成堆。

(1) (150，86，48，73，35，39，42，57，66，22)

(2) (120，70，33，65，24，56，48，92，86，30)

5．输入若干国家名称，请按字典顺序将这些国家名称排序。(设所有的名称均用大写或小写表示。)



第10章 文 件

10.1 文件的基本概念

所谓文件，一般是指存储在外部存储介质上的数据的集合。这里所讨论的文件主要是数据库意义上的文件。数据库所研究的文件是带有结构的记录集合，每个记录可以由若干个数据项构成，数据项有时也称为字段。可以说，文件就是性质相同的记录的集合。文件所涉及数据量通常较大，因此被放置在外存中。

表10.1是一个简单的职工文件,，每个职工情况是一个记录，每个记录由6个数据项组成。

表10.1 职 工 文 件

编号	姓名	性别	职称	婚姻状况	工资
1001	王 华	女	工程师	已婚	1500
1002	赵小方	女	经理	未婚	1200
1003	张 弓	男	高工	已婚	2000
1004	李 明	男	工程师	已婚	1800

文件的操作有两类：检索和修改。

检索就是在文件中查找满足给定条件的记录，它可以根据记录的序号或记录的相对位置进行查找，也可以按关键字进行查找。

文件的修改包括记录的插入、删除和更新三种操作。

10.2 文件的组织

10.2.1 顺序文件

顺序文件是指文件中的物理记录按其在文件中的逻辑顺序依次存入存储介质而建立的文件。

顺序文件的存取是根据记录的序号或记录的相对位置进行检索，如果要存取第i个记录，则必须先搜索它前面的i－1个记录；如果要插入一个新记录，则只能添加在文件的末尾；如果要更新文件中的记录，则对要修改的记录用新记录代替。

磁带是一种典型的顺序存取设备，存储在磁带上的文件只能是顺序文件，其存储时间与数据在磁带上的位置及当前读写头所在位置有关。如果文件中的第i个记录被存取过，而下一个要存取的记录是第i+1个记录，则依次存取会很快完成；如果某个文件在磁带的末尾，而磁头当前位置在磁带首部，则必须空转磁带到尾部才能读取文件，此时，磁带的存取速度较慢。

磁盘是一种直接存储介质，存放于磁带上的文件可以是顺序文件，也可以是其他结构类型的文件。磁盘上的文件可以用顺序查找存取，也可以用分块查找或折半查找进行存取。

10.2.2 索引文件

为了提高文件的检索效率，可以建立一个索引表，给出关键字与相应记录地址之间的对应关系，这种包括文件记录数据和索引表两大部分的文件称为索引文件。索引表中的每一项称作索引项，不论主文件是否按关键字有序排列，索引项总是按关键字有序。若主文件的记录也按关键字有序排列，则称为索引顺序文件，反之，若主文件的记录不按关键字有序排列，则称为索引无序文件。

索引文件只能存放在直接存储的外存储器(例如磁盘)中，不能存放在只能顺序存取的磁带中。表10.2是一个学生档案数据文件。

表10.2 学生档案数据文件

物理记录号	学号	姓名	性别	入学总分
1	200203	王芳	女	535.0
2	200218	刘志	男	550.0
3	200227	李大成	男	530.0
4	200207	张民	男	540.0
5	200104	梁小平	女	520.0

表10.3 以总分为关键字排序后的索引表

总分	物理记录号(指针)
520.0	5
530.0	3
535.0	1
540.0	4
550.0	2

有了表10.3所示以总分为关键字排序后的索引表后，按总分查找学生时可先在索引表中查找，得到物理记录号后再到数据区取出对应的物理记录。

因此，索引文件的检索是分两步进行的，即先读索引表，根据索引表得到所要检索记录的物理地址，然后根据该物理地址读取记录。由于索引表已按关键字排序，因此对索引表的查找可采用效率较高的折半查找方法。

索引文件的其他操作也很简单。插入记录时，先将记录插入到主文件的尾部，然后在索引表中增加相应的索引项并重新排序；删除记录时，仅需要删去相应的索引项；更新记录时，若不更新记录的关键字，则不必修改索引表，只需检索到对应的记录后修改要求的数据项内容，若需要更新记录的关键字，则既要修改主文件记录，也要更新索引表。

10.2.3 索引顺序文件ISAM

ISAM是Index Sequential Access Method(索引顺序存取方法)的缩写，是一种专为磁盘存取设计的文件组织方式，采用的是静态索引结构。ISAM文件是采用索引顺序存取方法的文件。由于磁盘是以盘组、柱面和磁道三级地址存取的设备，因此ISAM方法对磁盘上的数据文件要建立主索引、柱面和磁道三级索引。

在ISAM文件中，存储的数据文件先按关键字排序。文件记录在同一盘组上存放时，应先集中放在一个柱面上，然后再顺序存放在相邻的柱面上，对于同一个柱面，则应按盘面的次序顺序存放。

图10.1为一个ISAM文件的结构示例。每个柱面建立一个磁道索引，每个磁道索引由两个部分组成：基本索引项和溢出索引项，每一部分都包括关键字和指针两项，关键字表示该磁道最末一个记录的关键字(最大关键字)，指针指示该磁道中第一个记录的位置。柱面索引的每一个索引项也由关键字和指针两部分组成，前者表示该柱面中最末一个记录的关键字，后者指示该柱面上的磁道索引首地址。柱面索引放在某一个柱面上，如果柱面索引较大，占用多个磁道时，则可以建立柱面索引的索引，称为主索引。

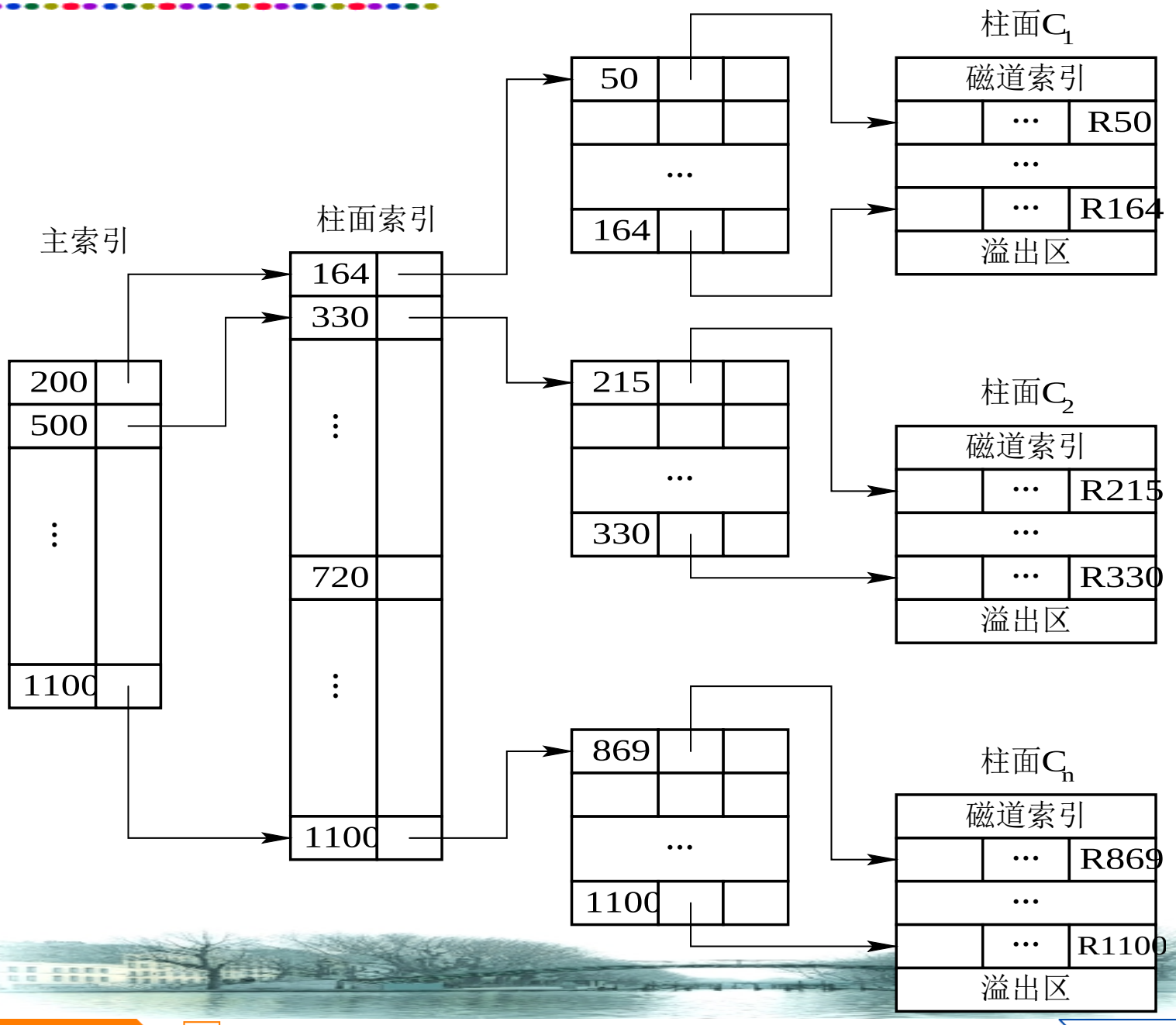
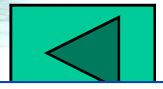


图10.1 ISAM文件的结构示例

在ISAM文件上查找一个记录时，先从主索引出发，再从柱面索引找到记录所在柱面的磁道索引，最后从磁道索引的基本项找到记录所在磁道第一个记录的位置，由此出发在该磁道上进行顺序查找，进而根据溢出索引项在溢出区查找，查到找到为止，反之，如果找遍该磁道及其溢出区不存在此记录，则表明该文件中无此记录。

每个柱面上还开辟有一个溢出区，并且磁道索引项中设有溢出索引项，这是为插入记录所设置的。由于ISAM文件中的记录是按关键字大小顺序存放的，则在插入记录时需移动记录，并将同一磁道上最后一个记录移至溢出区，同时修改磁道溢出项。每个柱面的基本区是顺序存储结构，而溢出区是链表结构。同一磁道溢出的记录由指针相链，该磁道索引的溢出索引项中的关键字指示该磁道溢出的记录的最大关键字，指针则指示在溢出区中的第一个记录。ISAM文件删除记录时，只需找到待删除的记录位置，并在其存储位置上作删除标记即可，而不需移动记录或改变指针。经过多次增删记录后，文件的结构有可能变得很不合理，此时，大量的记录进入溢出区，而基本区中又因存在被删除的记录而浪费很多存储空间。所以，有必要周期性地整理ISAM文件，将记录读入内存，重新排序，复制成一个新的ISAM文件，填满基本区而空出溢出区，以达到节约存储空间的目的。



习 题 10

1. 叙述顺序文件、索引文件的结构特点。

2. 设有一个职工文件如下表所示，由5个记录组织，其中职工号为关键字。

(1) 若该文件为顺序文件，请写出文件的存储结构。

(2) 若文件为索引文件，请写出索引表。

物理记录号	职工号	姓名	性别	工资
1	39	张珊	女	1000.00
2	50	王芳	女	1200.00
3	10	李斌	男	800.00
4	60	丁达	男	1500.00
5	25	赵刚	男	2000.00

3. 图10.2给出了一个ISAM文件的局部表示，其中记录用关键字代表。

柱面索引

磁道索引

图10.2 第3题图

柱面C_1

33	50	57	60	62	
65	67	71	72	80	
91	102	110	120	135	
溢出区					

图10.2 第3题图

(1) 画出相应柱面索引和磁道索引。

(2) 当插入75，40两个记录后，画出索引的变化情况和溢出区的情况。