北京航空航天大学计算机研究生考试必备（七） 数据结构

第一章 绪论

学习数据结构的目的是为了了解计算机处理对象的特性，将实际问题中所涉及的处理对象在计算机中表示出来并对它们进行处理。

数据（Data）是信息的载体，它能够被计算机识别、存储和加工处理。它是计算机程序加工的原料，应用程序处理各种各样的数据。计算机科学中，所谓数据就是计算机加工处理的对象，它可以是数值数据，也可以是非数值数据。

数据元素（Data Element）是数据的基本单位。在不同的条件下，数据元素又可称为元素、结点、顶点、记录等。

有时，一个数据元素可由若干个数据项（Data Item）组成，数据项可以分为两种：一种叫做初等项，这些数据项是在数据处理时不能再分割的最小单位；另一种叫做组合项，可以再划分为更小的项。

数据对象（Data Object）或数据元素类（Data Element Class）是具有相同性质的数据元素的集合。在某个具体问题中，数据元素都具有相同的性质（元素值不一定相等），属于同一数据对象（数据元素类），数据元素是数据元素类的一个实例。

数据结构（Data Structure）是指互相之间存在着一种或多种关系的数据元素的集合。在任何问题中，数据元素之间都不会是孤立的，在它们之间都存在着这样或那样的关系，这种数据元素之间的关系称为结构。根据数据元素间关系的不同特性，通常有下列四类基本的结构：

⑴集合结构。在集合结构中，数据元素间的关系是“属于同一个集合”。集合是元素关系极为松散的一种结构。

⑵线性结构。该结构的数据元素之间存在着一对一的关系。

⑶树型结构。该结构的数据元素之间存在着一对多的关系。

⑷图形结构。该结构的数据元素之间存在着多对多的关系，图形结构也称作网状结构。

由于集合是数据元素之间关系极为松散的一种结构，因此也可用其他结构来表示它。

从上面所介绍的数据结构的概念中可以知道，一个数据结构有两个要素。一个是数据元素的集合，另一个是关系的集合。在形式上，数据结构通常可以采用一个二元组来表示。

数据结构的形式定义为：数据结构是一个二元组

$$Data\_Structure = (D, R)$$

其中，D 是数据元素的有限集，R 是 D 上关系的有限集。

数据结构包括数据的逻辑结构和数据的物理结构。

数据的逻辑结构可以看作是从具体问题抽象出来的数学模型，它与数据的存储无关。我们研究数据结构的目的是为了在计算机中实现对它的操作，为此还需要研究如何在计算机中表示一个数据结构。

数据结构在计算机中的标识（又称映像）称为数据的物理结构，或称存储结构。它所研究的是数据结构在计算机中的实现方法，包括数据结构中元素的表示及元素间关系的表示。

数据的存储结构可采用顺序存储或链式存储的方法。

顺序存储方法是把逻辑上相邻的元素存储在物理位置相邻的存储单元中，由此得到的存储表示称为顺序存储结构。顺序存储结构是一种最基本的存储表示方法，通常借助于程序设计语言中的数组来实现。

链式存储方法对逻辑上相邻的元素不要求其物理位置相邻，元素间的逻辑关系通过附设的指针字段来表示，由此得到的存储表示称为链式存储结构，链式存储结构通常借助于程序设计语言中的指针类型来实现。

除了通常采用的顺序存储方法和链式存储方法外，有时为了查找的方便还采用索引存储方法和散列存储方法。

算法特性

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。其中每一条指令表示一个或多个操作。一个算法应该具有下列特性：

⑴有穷性。一个算法必须在有穷步之后结束，即必须在有限时间内完成。

⑵确定性。算法的每一步必须有确切的定义，无二义性。算法的执行对应着的相同的输入仅有唯一的一条路经。

⑶可行性。算法中的每一步都可以通过已经实现的基本运算的有限次执行得以实现。

⑷输入。一个算法具有零个或多个输入，这些输入取自特定的数据对象集合。

⑸输出。一个算法具有一个或多个输出，这些输出同输入之间存在某种特定的关系。

要设计一个好的算法通常要考虑以下的要求。

⑴正确。算法的执行结果应当满足预先规定的功能和性能要求。

⑵可读。一个算法应当思路清晰、层次分明、简单明了、易读易懂。

⑶健壮。当输入不合法数据时，应能作适当处理，不至引起严重后果。

⑷高效。有效使用存储空间和有较高的时间效率。

我们可以从一个算法的时间复杂度与空间复杂度来评价算法的优劣。

当我们将一个算法转换成程序并在计算机上执行时，其运行所需要的时间取决于下列因素：

⑴硬件的速度。例如使用 486 机还是使用 586 机。

⑵书写程序的语言。实现语言的级别越高，其执行效率就越低。

⑶编译程序所生成目标代码的质量。对于代码优化较好的编译程序其所生成的程序质量较高。

⑷问题的规模。例如，求 100 以内的素数与求 1000 以内的素数其执行时间必然是不同的。

显然，在各种因素都不能确定的情况下，很难比较出算法的执行时间。也就是说，使用执行算法的绝对时间来衡量算法的效率是不合适的。为此，可以将上述各种与计算机相关的软、硬件因素都确定下来，这样一个特定算法的运行工作量的大小就只依赖于问题的规模（通常用正整数 n 表示），或者说它是问题规模的函数。

1.时间复杂度

一个程序的时间复杂度（Time complexity）是指程序运行从开始到结束所需要的时间。

一个算法是由控制结构和原操作构成的，其执行时间取决于两者的综合效果。为了便于比较同一问题的不同的算法，通常的做法是：从算法中选取一种对于所研究的问题来说是基本运算的原操作，以该原操作重复执行的次数作为算法的时间度量。一般情况下，算法中原操作重复执行的次数是规模 n 的某个函数 T(n)。

许多时候要精确地计算 T(n)是困难的，我们引入渐进时间复杂度在数量上估计一个算法的执行时间，也能够达到分析算法的目的。

定义（大 O 记号）：如果存在两个正常数 c 和 n_0，使得对所有的 n，$n \geqslant n_0$，有：

$$f(n) \leqslant cg(n)$$

则有：

$$f(n) = O(g(n))$$

使用大 O 记号表示的算法的时间复杂度，称为算法的渐进时间复杂度（Asymptotic Complexity）。

通常用 O(1)表示常数计算时间。常见的渐进时间复杂度有：

$$O(1) < O(\log_2 n) < O(n) < O(n\log_2 n) < O(n^2) < O(n^3) < O(2^n)$$

2.空间复杂度

一个程序的空间复杂度（Space complexity）是指程序运行从开始到结束所需的存储量。

程序运行所需的存储空间包括以下两部分：

⑴固定部分。这部分空间与所处理数据的大小和个数无关，或者称与问题的实例的特征无关。主要包括程序代码、常量、简单变量、定长成分的结构变量所占的空间。

⑵可变部分。这部分空间大小与算法在某次执行中处理的特定数据的大小和规模有关。例如 100 个数据元素的排序算法与 1000 个数据元素的排序算法所需的存储空间显然是不同的。

第二章　线性表

线性表是最简单、最基本、也是最常用的一种线性结构。它有两种存储方法：顺序存储和链式存储，它的主要基本操作是插入、删除和检索等。

2.1.1　线性表的定义

线性表是一种线性结构。线性结构的特点是数据元素之间是一种线性关系，数据元素“一个接一个的排列”。在一个线性表中数据元素的类型是相同的，或者说线性表是由同一类型的数据元素构成的线性结构。

线性表是具有相同数据类型的n(n>=0)个数据元素的有限序列，通常记为：

$$(a_1, a_2, \cdots a_{i-1}, a_i, a_{i+1}, \cdots a_n)$$

其中n为表长， n＝0 时称为空表。

表中相邻元素之间存在着顺序关系。将 a_{i-1}称为 a_i的直接前趋，a_{i+1}称为 a_i的直接后继。就是说：对于a_i，当 i=2，...，n 时，有且仅有一个直接前趋 a_{i-1}，当i=1，2，...，n-1 时，有且仅有一个直接后继 a_{i+1}，而 a_1是表中第一个元素，它没有前趋，a_n是最后一个元素无后继。

需要说明的是：a_i为序号为 i 的数据元素（i=1,2,…,n），通常我们将它的数据类型抽象为datatype。

2.1.2　线性表的基本操作

在第一章中提到，数据结构的运算是定义在逻辑结构层次上的，而运算的具体实现是建立在存储结构上的，因此下面定义的线性表的基本运算作为逻辑结构的一部分，每一个操作的具体实现只有在确定了线性表的存储结构之后才能完成。

线性表上的基本操作有：

⑴ 线性表初始化：Init_List(L)

初始条件：表L不存在

操作结果：构造一个空的线性表

⑵ 求线性表的长度：Length_List(L)

初始条件：表L存在

操作结果：返回线性表中的所含元素的个数

⑶ 取表元：Get_List(L,i)

初始条件：表L存在且1<=i<=Length_List(L)

操作结果：返回线性表L中的第 i 个元素的值或地址

⑷ 按值查找：Locate_List(L,x)， x 是给定的一个数据元素。

初始条件：线性表L存在

操作结果：在表L中查找值为 x 的数据元素，其结果返回在L中首次出现的值为 x 的那个元素的序号或地址，称为查找成功；否则，在L中未找到值为 x 的数据元素，返回一特殊值表示查找失败。

⑸ 插入操作：Insert_List(L,i,x)

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

操作结果：在线性表L的第 i 个位置上插入一个值为 x 的新元素，这样使原序号为 i , i+1, ... , n 的数据元素的序号变为 i+1,i+2, ... , n+1，插入后表长=原表长+1。

⑹ 删除操作：Delete_List(L,i)

初始条件：线性表L存在，1<=i<=n。

操作结果：在线性表L中删除序号为i的数据元素，删除后使序号为 i+1, i+2,..., n 的元素变为序号为 i, i+1,...,n-1，新表长＝原表长－１。

2.2 线性表的顺序存储及运算实现

2.2.1 顺序表

线性表的顺序存储是指在内存中用地址连续的一块存储空间顺序存放线性表的各元素，用这种存储形式存储的线性表称其为顺序表。

设 a_1的存储地址为Loc(a_1)，每个数据元素占d个存储地址，则第i个数据元素的地址为：

$$Loc(a_i)=Loc(a_1)+(i-1)*d \qquad 1<=i<=n$$

顺序表具有按数据元素的序号随机存取的特点。

在程序设计语言中，一维数组在内存中占用的存储空间就是一组连续的存储区域，因此，用一维数组来表示顺序表的数据存储区域是再合适不过的。考虑到线性表的运算有插入、删除等运算，即表长是可变的，因此，数组的容量需设计的足够大，设用：data[MAXSIZE]来表示，其中MAXSIZE是一个根据实际问题定义的足够大的整数，线性表中的数据从 data[0] 开始依次顺序存放，但当前线性表中的实际元素个数可能未达到MAXSIZE多个，因此需用一个变量 last 记录当前线性表中最后一个元素在数组中的位置，即 last 起一个指针的作用，始终指向线性表中最后一个元素，因此，表空时last=-1。这种存储思想的具体描述可以是多样的。如可以是：

```
datatype  data[MAXSIZE];
int  last;
```

从结构性上考虑，通常将 data 和 last 封装成一个结构作为顺序表的类型：

```
typedef  struct
        {   datatype  data[MAXSIZE];
            int  last;
    } SeqList;
```

定义一个顺序表：SeqList L ;

由于我们后面的算法用C语言描述，根据C语言中的一些规则，有时定义一个指向SeqList 类型的指针更为方便：

```
SeqList  *L ;
```

L是一个指针变量，线性表的存储空间通过 L=malloc(sizeof(SeqList)) 操作来获得。

L中存放的是顺序表的地址，表长表示为 (*L).last或 L－＞last+1，线性表的存储区域为 L->data ，线性表中数据元素的存储空间为： L－＞data[0] ~ L－＞data[L－＞last]。

2.2.2 顺序表上基本运算的实现

1.顺序表的初始化

顺序表的初始化即构造一个空表，这对表是一个加工型的运算，因此，将L设为指针参数，首先动态分配存储空间，然后，将表中 last 指针置为－1，表示表中没有数据元素。算法如下：

```
SeqList *init_SeqList( )
  { SeqList *L;
    L=malloc(sizeof(SeqList));
     L->last=-1;   return L;
  }
```

算法2.1

设调用函数为主函数，主函数对初始化函数的调用如下：

```
main()
{SeqList *L;
 L=Init_SeqList();
  . . .
  }
```

2. 插入运算

(1) 将a_i～a_n顺序向下移动，为新元素让出位置；

(2) 将 x 置入空出的第i个位置；

(3) 修改 last 指针(相当于修改表长)，使之仍指向最后一个元素。

算法如下：

```
int  Insert_SeqList(SeqList *L, int i, datatype x)
{  int j;
   if (L->last==MAXSIZE－1)
       {  printf(" 表满 "); return(-1); }  /*表空间已满，不能插入*/
   if (i<1 || i>L->last+2)    /*检查插入位置的正确性*/
     {  printf(" 位置错 ");return(0); }
   for(j=L->last;j>=i-1;j--)
       L->data[j+1]=L->data[j];      /* 结点移动 */
   L->data[i-1]=x;          /*新元素插入*/
   L->last++;     /*last仍指向最后元素*/
   return (1);      /*插入成功，返回*/
}
```

算法2.2

本算法中注意以下问题：

(1) 顺序表中数据区域有MAXSIZE个存储单元，所以在向顺序表中做插入时先检　查表空间是否满了，在表满的情况下不能再做插入，否则产生溢出错误。

(2) 要检验插入位置的有效性，这里 i 的有效范围是：1<=i<=n+1，其中 n 为原表长。

(3) 注意数据的移动方向。

插入算法的时间性能分析：

顺序表上的插入运算，时间主要消耗在了数据的移动上，在第i个位置上插入 x ，从 a_i 到 a_n 都要向下移动一个位置，共需要移动 n－i＋1个元素，而 i 的取值范围为 ：

1<= i<= n+1，即有 n＋1个位置可以插入。设在第i个位置上作插入的概率为P_i，则平均移动数据元素的次数：

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

设：$P_i=1/(n+1)$，即为等概率情况，则：

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

这说明：在顺序表上做插入操作需移动表中一半的数据元素。显然时间复杂度为O(n)。

3. 删除运算 DeleteList(L,i)

顺序表上完成这一运算的步骤如下：

(1) 将a_{i+1}～a_n顺序向上移动。

(2) 修改last指针(相当于修改表长)使之仍指向最后一个元素。

算法如下：

```
int Delete_SeqList(SeqList *L;int i)
   { int  j;
   if(i<1 || i>L->last+1)   /*检查空表及删除位置的合法性*/
         { printf ("不存在第i个元素"); return(0); }
   for(j=i;j<=L->last;j++)
         L->data[j-1]=L->data[j]; /*向上移动*/
    L->last--;
    return(1);         /*删除成功*/
   }
```

算法2.3

本算法注意以下问题：

(1) 删除第i个元素，i的取值为 1<=i<=n ,否则第i个元素不存在，因此，要检查删除位置的有效性。

(2) 当表空时不能做删除，因表空时 L->last的值为-1，条件（i<1 || i>L->last+1）也包括了对表空的检查。

(3) 删除 a_i 之后，该数据已不存在，如果需要，先取出 a_i，再做删除。

删除算法的时间性能分析：

与插入运算相同，其时间主要消耗在了移动表中元素上，删除第i个元素时，其后面的元素 a_{i+1}～a_n 都要向上移动一个位置，共移动了 n-i 个元素，所以平均移动数据元素的次数：

$$E_{de}=\sum_{i=1}^{n}p_i(n-i)$$

在等概率情况下，$p_i=1/n$，则：

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

这说明顺序表上作删除运算时大约需要移动表中一半的元素，显然该算法的时间复杂度为O(n)。

4. 按值查找

线性表中的按值查找是指在线性表中查找与给定值x相等的数据元素。在顺序表中完成该运算最简单的方法是：从第一个元素 a_1 起依次和x比较，直到找到一个与x相等的数据元素，则返回它在顺序表中的存储下标或序号（二者差一）；或者查遍整个表都没有找到与 x 相等的元素，返回-1。

算法如下：

```
int Location_SeqList(SeqList *L, datatype x)
   {  int i=0;
     while(i<=L.last && L->data[i]!= x)
        i++;
     if (i>L->last)  return -1;
     else     return i;  /*返回的是存储位置*/
}
```

算法2.4

本算法的主要运算是比较。显然比较的次数与x在表中的位置有关，也与表长有关。当 $a_1=x$ 时，比较一次成功。当 $a_n=x$ 时比较 n 次成功。平均比较次数为（n+1）/2，时间性能为O(n)。

2.3 线性表的链式存储和运算实现

2.3.1 单链表

为建立起数据元素之间的线性关系,对每个数据元素a_i,除了存放数据元素的自身的信息 a_i 之外,还需要和a_i一起存放其后继 a_{i+1} 所在的存贮单元的地址，这两部分信息组成一个“结点”，存放数据元素信息的称为数据域，存放其后继地址的称为指针域。因此n个元素的线性表通过每个结点的指针域拉成了一个“链子”，称之为链表。因为每个结点中只有一个指向后继的指针，所以称其为单链表。

链表是由一个个结点构成的，结点定义如下：

```
typedef struct node
      {  datatype data;
          struct node *next;
  } LNode，*LinkList;
```

定义头指针变量：

```
LinkList  H;
```

通常我们用“头指针”来标识一个单链表，如单链表L、单链表H等，是指某链表的第一个结点的地址放在了指针变量 L、H 中，头指针为“NULL”则表示一个空表。

需要进一步指出的是：上面定义的LNode是结点的类型，LinkList是指向Lnode类型结点的指针类型。 为了增强程序的可读性，通常将标识一个链表的头指针说明为LinkList类型的变量，如LinkList L ；当L有定义时，值要么为NULL，则表示一个空表；要么为第一个结点的地址，即链表的头指针；将操作中用到指向某结点的指针变量说明为LNode *类型，如LNode *p； 则语句：

```
p=malloc(sizeof(LNode));
```

则完成了申请一块 Lnode 类型的存储单元的操作，并将其地址赋值给变量p。P所指的结点为*p，*p的类型为LNode型，所以该结点的数据域为 (*p).data 或p->data，指针域为 (*p).next 或 p->next。free(p)则表示释放 p 所指的结点。

2.3.2 单链表上基本运算的实现

1. 建立单链表

(1)在链表的头部插入结点建立单链表

链表与顺序表不同，它是一种动态管理的存储结构，链表中的每个结点占用的存储空间不是预先分配，而是运行时系统根据需求而生成的，因此建立单链表从空表开始，每读入一个数据元素则申请一个结点，然后插在链表的头部，因为是在链表的头部插入，读入数据的顺序和线性表中的逻辑顺序是相反的。

算法如下：

```
LinkList  Creat_LinkList1( )
{   LinkList L=NULL; /*空表*/
   Lnode *s;
    int x;        /*设数据元素的类型为int*/
     scanf(" %d ",&x);
     while (x!=flag)
       { s=malloc(sizeof(LNode));
         s->data=x;
         s->next=L;  L=s;
        Scanf (" %d ",&x);
       }
    return L;
}
```

算法2.8

(2)在单链表的尾部插入结点建立单链表

因为每次是将新结点插入到链表的尾部，所以需加入一个指针 r 用来始终指向链表中的尾结点，以便能够将新结点插入到链表的尾部

初始状态：头指针H=NULL，尾指针 r=NULL；按线性表中元素的顺序依次读入数据元素，不是结束标志时，申请结点，将新结点插入到 r 所指结点的后面，然后 r 指向新结点（但第一个结点有所不同，读者注意下面算法中的有关部分）。

算法如下：

```
LinkList  Creat_LinkList2( )
  { LinkList L=NULL;
    Lnode  *s,*r=NULL;
     int x;       /*设数据元素的类型为int*/
      scanf(" %d ",&x);
      while (x!=flag)
        { s=malloc(sizeof(LNode));      s->data=x;
          if  (L==NULL)  L=s;  /*第一个结点的处理*/
          else  r->next=s;       /*其它结点的处理*/
          r=s;         /*r 指向新的尾结点*/
         scanf(" %d ",&x);
        }
     if ( r!=NULL)  r->next=NULL; /*对于非空表，最后结点的指针域放空指针*/
     return L;
}
```

算法2.9

在上面的算法中，第一个结点的处理和其它结点是不同的，原因是第一个结点加入时链表为空，它没有直接前驱结点，它的地址就是整个链表的指针， 需要放在链表的头指针变量中；而其它结点有直接前驱结点，其地址放入直接前驱结点的指针域。“第一个结点”的问题在很多操作中都会遇到，如在链表中插入结点时，将结点插在第一个位置和其它位置是不同的，在链表中删除结点时，删除第一个结点和其它结点的处理也是不同的，等等，为了方便操作，有时在链表的头部加入一个“头结点”，头结点的类型与数据结点一致，标识链表的头指针变量L中存放该结点的地址，这样即使是空表，头指针变量L也不为空了。头结点的加入使得“第一个结点”的问题不再存在，也使得“空表”和“非空表”的处理成为一致。

头结点的加入完全是为了运算的方便，它的数据域无定义，指针域中存放的是第一个数据结点的地址，空表时为空。

2. 求表长

算法思路：设一个移动指针 p 和计数器 j ，初始化后， p 所指结点后面若还有结点， p 向后移动，计数器加1。

(1)设L是带头结点的单链表(线性表的长度不包括头结点)。

算法如下：

```
   int  Length_LinkList1 (LinkList L)
        { Lnode  * p=L;   /* p指向头结点*/
          int  j=0;
          while (p->next)
           { p=p->next; j++ }   /* p所指的是第 j 个结点*/
          return  j;
        }
```

算法2.10(a)

(2)设L是不带头结点的单链表。

算法如下：

```
  int  Length_LinkList2 (LinkList L)
        {  Lnode  * p=L;
           int  j;
           if (p==NULL) return  0;  /*空表的情况*/
           j=1;          /*在非空表的情况下，p所指的是第一个结点*/;
```

```
        while (p->next )
         { p=p->next;  j++ }
        return  j;
    }
```

算法2.10(b)

从上面两个算法中看到，不带头结点的单链表空表情况要单独处理，而带上头结点之后则不用了。在以后的算法中不加说明则认为单链表是带头结点的。算法2.10(a)、(b)的时间复杂度均为O(n)。

3. 查找操作

(1) 按序号查找 Get_Linklist(L,i)

算法思路：从链表的第一个元素结点起，判断当前结点是否是第i个，若是，则返回该结点的指针，否则继续后一个，表结束为止。没有第 i 个结点时返回空。

算法如下：

```
Lnode  * Get_LinkList(LinkList  L, Int  i);
  /*在单链表L中查找第i个元素结点，找到返回其指针，否则返回空*/
 {  Lnode  * p=L;
     int  j=0;
     while (p->next !=NULL && j<i )
      {  p=p->next;  j++;  }
     if (j==i) return p;
     else  return NULL;
  }
```

算法2.11(a)

(2) 按值查找即定位 Locate_LinkList(L,x)

算法思路：从链表的第一个元素结点起，判断当前结点其值是否等于x，若是，返回该结点的指针，否则继续后一个，表结束为止。找不到时返回空。

算法如下：

```
Lnode  * Locate_LinkList( LinkList  L, datatype  x)
    /*在单链表L中查找值为x的结点，找到后返回其指针，否则返回空*/
 {  Lnode  * p=L->next;
    while ( p!=NULL && p->data != x)
       p=p->next;
    return p;
  }
```

算法2.11(b)

算法2.11(a)、(b)的时间复杂度均为O(n)。

4.插入

(1)后插结点：设p指向单链表中某结点，s指向待插入的值为x的新结点，将*s插入到*p的后面，插入示意图如图2.13。

操作如下：

①s->next=p->next;

②p->next=s;

注意：两个指针的操作顺序不能交换。

(2)前插结点：设 p 指向链表中某结点， s 指向待插入的值为x的新结点，将*s插入到*p的前面，插入示意图如图2.14，与后插不同的是：首先要找到*p的前驱*q，然后再完成在*q之后插入*s，设单链表头指针为L，操作如下：

```
q=L;
```

```
while (q->next!=p)
  q=q->next;        /*找*p的直接前驱*/
s->next=q->next;
q->next=s;
```

后插操作的时间复杂性为O(1)，前插操作因为要找 *p 的前驱，时间性能为O(n)；其实我们关心的更是数据元素之间的逻辑关系，所以仍然可以将 *s 插入到 *p 的后面，然后将 p->data与s->data交换即可，这样即满足了逻辑关系，也能使得时间复杂性为O(1)。

(3)插入运算 Insert_LinkList(L,i,x)

算法思路：1.找到第i-1个结点；若存在继续2，否则结束

2.申请、填装新结点；

3.将新结点插入。结束。

算法如下：

```
int  Insert_LinkList( LinkList  L, int i, datatype  x)
     /*在单链表L的第i个位置上插入值为x的元素*/
 { Lnode  * p,*s;
    p=Get_LinkList(L,i-1);   /*查找第i-1个结点*/
    if (p==NULL)
        { printf(" 参数i错 ");return 0; } /*第i-1个不存在不能插入*/
    else {
          s=malloc(sizeof(LNode)); /*申请、填装结点*/
          s->data=x;
          s->next=p->next;        /*新结点插入在第i-1个结点的后面*/
          p->next=s
         return 1;
  }
```

算法2.12

算法2.12的时间复杂度为O(n)。

5. 删除

(1)删除结点：设p指向单链表中某结点，删除*p。要实现对结点*p的删除，首先要找到*p的前驱结点*q，然后完成指针的操作即可。指针的操作由下列语句实现：

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

显然找*p前驱的时间复杂性为O(n)。

若要删除*p的后继结点(假设存在)，则可以直接完成：

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

该操作的时间复杂性为O(1) 。

(2)删除运算：Del_LinkList(L,i)

算法思路：1.找到第i-1个结点；若存在继续2，否则结束；

2.若存在第i个结点则继续3，否则结束；

3.删除第i个结点，结束。

算法如下：

```
int  Del_LinkList(LinkList  L, int i)
   /*删除单链表L上的第i个数据结点*/
 {  LinkList  p,s;
```

```
    p=Get_LinkList(L,i-1);   /*查找第i-1个结点*/
    if (p==NULL)
        { printf(" 第i-1个结点不存在 ");return -1; }
    else {  if (p->next==NULL)
              { printf(" 第i个结点不存在 ");return 0; }
          else
              { s=p->next;  /*s指向第i个结点*/
                p->next=s->next;        /*从链表中删除*/
                free(s);                /*释放*s */
                return 1;
  }
```

算法2.13

算法2.13的时间复杂度为O(n)。

通过上面的基本操作我们得知：

(1) 在单链表上插入、删除一个结点，必须知道其前驱结点。

(2) 单链表不具有按序号随机访问的特点，只能从头指针开始一个个顺序进行。

2.3.3 循环链表

对于单链表而言，最后一个结点的指针域是空指针，如果将该链表头指针置入该指针域，则使得链表头尾结点相连，就构成了单循环链表。

在单循环链表上的操作基本上与非循环链表相同，只是将原来判断指针是否为 NULL 变为是否是头指针而已，没有其它较大的变化。

对于单链表只能从头结点开始遍历整个链表，而对于单循环链表则可以从表中任意结点开始遍历整个链表，不仅如此，有时对链表常做的操作是在表尾、表头进行，此时可以改变一下链表的标识方法，不用头指针而用一个指向尾结点的指针 R 来标识，可以使得操作效率得以提高。

例如对两个单循环链表 H1 、H2 的连接操作，是将 H2 的第一个数据结点接到 H1 的尾结点，如用头指针标识，则需要找到第一个链表的尾结点，其时间复杂性为 O(n)，而链表若用尾指针 R1 、R2 来标识，则时间性能为 O(1)。操作如下：

```
p= R1 ->next;    /*保存 R1 的头结点指针*/
R1->next=R2->next->next;  /*头尾连接*/
free(R2->next);  /*释放第二个表的头结点*/
R2->next=p;    /*组成循环链表*/
```

2.3.4 双向链表

以上讨论的单链表的结点中只有一个指向其后继结点的指针域 next，因此若已知某结点的指针为 p，其后继结点的指针则为 p->next，而找其前驱则只能从该链表的头指针开始，顺着各结点的 next 域进行，也就是说找后继的时间性能是 O(1)，找前驱的时间性能是 O(n)，如果也希望找前驱的时间性能达到 O(1)，则只能付出空间的代价：每个结点再加一个指向前驱的指针域，用这种结点组成的链表称为双向链表。

双向链表结点的定义如下：

```
typedef struct  dlnode
{ datatype data;
  struct dlnode *prior,*next;
}DLNode,*DLinkList;
```

和单链表类似，双向链表通常也是用头指针标识，也可以带头结点和做成循环结构。显然通过某结点的指针 p 即可以直接得到它的后继结点的指针 p->next，也可以直接得到它的前驱结点的的指针 p->prior。这样在有些操作中需要找前驱时，则勿需再用循环。从下面的插入删除运算中可以看到这一点。

设 p 指向双向循环链表中的某一结点，即 p 中是该结点的指针，则 p->prior->next 表示的是*p 结点之前驱结点的后继结点的指针，即与 p 相等；类似，p->next->prior 表示的是*p 结点之后继结点的前驱结点的指针，也与 p 相等，所以有以下等式：

$$p->prior->next = p = p->next->prior$$

双向链表中结点的插入：设 p 指向双向链表中某结点，s 指向待插入的值为 x 的新结点，将*s 插入到*p 的前面。

操作如下：

① s->prior=p->prior;

② p->prior->next=s;

③ s->next=p;

④ p->prior=s;

指针操作的顺序不是唯一的，但也不是任意的，操作①必须要放到操作④的前面完成，否则*p 的前驱结点的指针就丢掉了。读者把每条指针操作的涵义搞清楚，就不难理解了。

双向链表中结点的删除：

设 p 指向双向链表中某结点，删除*p。

操作如下：

①p->prior->next=p->next;

②p->next->prior=p->prior;

free(p);

2.3.5 静态链表

数组sd的定义如下：

```
#define  MAXSIZE  …  /*足够大的数*/
typedef  struct
 {datatype  data;
  int       next;
 }SNode;  /*结点类型*/
SNode  sd[MAXSIZE];
int SL,AV;   /*两个头指针变量*/
```

这种链表的结点中也有数据域data和指针域next，与前面所讲的链表中的指针不同的是，这里的指针是结点的相对地址(数组的下标)，称之为静态指针，这种链表称之为静态链表，空指针用-1表示，因为上面定义的数组中没有下标为-1的单元。

2.3.6 单链表应用举例

例 2.5 已知单链表 H，写一算法将其倒置。即实现如图 2.22 的操作。(a)为倒置前，(b)为倒置后。

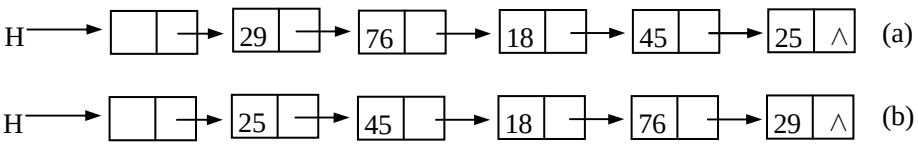


图2.22 单链表的倒置

算法思路：依次取原链表中的每个结点，将其作为第一个结点插入到新链表中去，指针p用来指向当前结点，p为空时结束。算法如下：

```
void  reverse (Linklist H)
 { LNode  *p;
   p=H->next;  /*p指向第一个数据结点*/
   H->next=NULL;  /*将原链表置为空表H*/
   while (p)
    {  q=p;   p=p->next;
       q->next=H->next;   /*将当前结点插到头结点的后面*/
       H->next=q;
    }
  }
```

算法2.15

该算法只是对链表中顺序扫描一边即完成了倒置，所以时间性能为O(n)。

例 2.6 已知单链表 L，写一算法，删除其重复结点，即实现如图 2.23 的操作。(a)为删除前，(b)为删除后。

算法思路：

用指针 p 指向第一个数据结点，从它的后继结点开始到表的结束，找与其值相同的结点并删除之；p 指向下一个；依此类推，p 指向最后结点时算法结束。

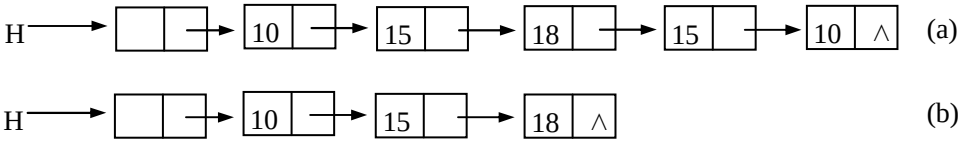


图2.23　删除重复结点

算法如下：

```
void pur_LinkList(LinkList H)
 { LNode  *p,*q,*r;
  p=H->next;  /*p指向第一个结点*/
  if(p==NULL) return;
  while (p->next)
   {  q=p;
       while (q->next)   /* 从*p的后继开始找重复结点*/
        { if (q->next->data==p->data)
           { r=q->next;  /*找到重复结点，用r指向，删除*r */
            q->next=r->next;
            free(r);
           }  /*if*/
         else q=q->next;
        } /*while(q->next)*/
     p=p->next;       /*p指向下一个，继续*/
   }  /*while(p->next)*/
  }
```

算法2.16

该算法的时间性能为$O(n^2)$。

例2.7 设有两个单链表A、B，其中元素递增有序，编写算法将A、B归并成一个按元素值递减（允许有相同值）有序的链表C，要求用A、B中的原结点形成，不能重新申请结点。

算法思路：利用A、B两表有序的特点，依次进行比较，将当前值较小者摘下，插入到C表的头部，得到的C表则为递减有序的。

算法如下：

```
LinkList  merge(LinkList A,LinkList B)
      /*设A、B均为带头结点的单链表*/
   { LinkList C;  LNode  *p,*q;
     p=A->next;q=B->next;
     C=A;   /*C表的头结点*/
     C->next=NULL;
      free(B);
      while (p&&q)
       { if (p->data<q->data)
          { s=p;p=p->next; }
         else
           {s=q;q=q->next;}  /*从原AB表上摘下较小者*/
          s->next=C->next;     /*插入到C表的头部*/
         C->next=s;
        }  /*while */
```

```
    if (p==NULL) p=q;
    while (p)       /* 将剩余的结点一个个摘下，插入到C表的头部*/
      { s=p;p=p->next;
        s->next=C->next;
       C->next=s;
      }
  }
```

算法2.17

该算法的时间性能为O(m+n)。

2．4 顺序表和链表的比较

顺序存储有三个优点：

(1) 方法简单，各种高级语言中都有数组，容易实现。

(2) 不用为表示结点间的逻辑关系而增加额外的存储开销。

(3) 顺序表具有按元素序号随机访问的特点。

两个缺点：

(1) 在顺序表中做插入删除操作时，平均移动大约表中一半的元素，因此对n较大的顺序表效率低。

(2) 需要预先分配足够大的存储空间，估计过大，可能会导致顺序表后部大量闲置；预先分配过小，又会造成溢出。

链表的优缺点恰好与顺序表相反。

在实际中怎样选取存储结构呢？通常有以下几点考虑：

1．基于存储的考虑

顺序表的存储空间是静态分配的，在程序执行之前必须明确规定它的存储规模，也就是说事先对＂MAXSIZE＂要有合适的设定，过大造成浪费，过小造成溢出。可见对线性表的长度或存储规模难以估计时，不宜采用顺序表；链表不用事先估计存储规模，但链表的存储密度较低，存储密度是指一个结点中数据元素所占的存储单元和整个结点所占的存储单元之比。显然链式存储结构的存储密度是小于 1 的。

2．基于运算的考虑

在顺序表中按序号访问a_i的时间性能时O(1)，而链表中按序号访问的时间性能O(n)，所以如果经常做的运算是按序号访问数据元素，显然顺序表优于链表；而在顺序表中做插入、删除时平均移动表中一半的元素，当数据元素的信息量较大且表较长时，这一点是不应忽视的；在链表中作插入、删除，虽然也要找插入位置，但操作主要是比较操作，从这个角度考虑显然后者优于前者。

3．基于环境的考虑

顺序表容易实现，任何高级语言中都有数组类型，链表的操作是基于指针的，相对来讲前者简单些，也是用户考虑的一个因素。

总之，两中存储结构各有长短，选择那一种由实际问题中的主要因素决定。通常“较稳定”的线性表选择顺序存储，而频繁做插入删除的即动态性较强的线性表宜选择链式存储。

第三章　栈和队列

栈和队列是在软件设计中常用的两种数据结构，它们的逻辑结构和线性表相同。其特点在于运算受到了限制：栈按“后进先出”的规则进行操作，队按“先进先出”的规则进行操作，故称运算受限制的线性表。

3．1　栈

3．1．1　栈的定义及基本运算

栈是限制在表的一端进行插入和删除的线性表。允许插入、删除的这一端称为栈顶，另一个固定端称为栈底。当表中没有元素时称为空栈。栈又称为后进先出的线性表（Last In First Out），简称 LIFO 表。

在程序设计中，常常需要栈这样的数据结构，使得与保存数据时相反顺序来使用这些数据，这时就需要用一个栈来实现。对于栈，常做的基本运算有：

⑴ 栈初始化：Init_Stack(s)

初始条件：栈 s 不存在

操作结果：构造了一个空栈。

⑵ 判栈空：Empty_Stack(s)

初始条件：栈 s 已存在

操作结果：若 s 为空栈返回为 1，否则返回为 0。

⑶ 入栈： Push_Stack(s，x)

初始条件：栈 s 已存在

操作结果：在栈 s 的顶部插入一个新元素 x， x 成为新的栈顶元素。栈发生变化。

⑷ 出栈：Pop_Stack(s)

初始条件：栈 s 存在且非空

操作结果：栈 s 的顶部元素从栈中删除，栈中少了一个元素。栈发生变化。

⑸ 读栈顶元素：Top_Stack(s)

初始条件：栈 s 存在且非空

操作结果：栈顶元素作为结果返回，栈不变化。

3.1.2 栈的存储实现和运算实现

由于栈是运算受限的线性表，因此线性表的存储结构对栈也是适用的，只是操作不同而已。

1. 顺序栈

利用顺序存储方式实现的栈称为顺序栈。栈中的数据元素用一个预设的足够长度的一维数组来实现：datatype data[MAXSIZE]，栈底位置可以设置在数组的任一个端点，而栈顶是随着插入和删除而变化的，用一个 int top 来作为栈顶的指针，指明当前栈顶的位置，同样将 data 和 top 封装在一个结构中，顺序栈的类型描述如下：

```
#define MAXSIZE  1024
typedef  struct
  {datatype  data[MAXSIZE];
   int  top;
  }SeqStack
```

定义一个指向顺序栈的指针：

```
    SeqStack  *s;
```

通常 0 下标端设为栈底，这样空栈时栈顶指针 top=-1; 入栈时，栈顶指针加 1 ，即 s->top++; 出栈时，栈顶指针减 1 ，即 s->top--。

⑴ 置空栈：首先建立栈空间，然后初始化栈顶指针。

```
SeqStack  *Init_SeqStack()
 { SeqStack  *s;
   s=malloc(sizeof(SeqStack));
   s->top= -1;  return s;
 }
```

⑵ 判空栈

```
int Empty_SeqStack(SeqStack *s)
  {  if (s->top= = -1)  return 1;
     else  return 0;
   }
```

⑶ 入栈

```
 int Push_SeqStack (SeqStack *s, datatype  x)
  {if (s->top= =MAXSIZE-1)  return 0; /*栈满不能入栈*/
   else {  s->top++;
         s->data[s->top]=x;
         return 1;
         }
  }
```

⑷ 出栈

```
  int  Pop_SeqStack(SeqStack *s, datatype *x)
    {  if  (Empty_SeqStack ( s ) )  return 0; /*栈空不能出栈 */
       else  {  *x=s->data[s->top];
          s->top--;  return 1; }        /*栈顶元素存入*x，返回*/
     }
```

⑸ 取栈顶元素

```
datatype  Top_SeqStack(SeqStack *s)
  {  if ( Empty_SeqStack ( s ) ) return 0;  /*栈空*/
     else return (s->data[s->top] );
  }
```

以下几点说明：

1. 对于顺序栈，入栈时，首先判栈是否满了，栈满的条件为：s->top= =MAXSIZE-1，栈满时，不能入栈；否则出现空间溢出，引起错误，这种现象称为上溢。

2. 出栈和读栈顶元素操作，先判栈是否为空，为空时不能操作，否则产生错误。通常栈空时常作为一种控制转移的条件。

2. 链栈

用链式存储结构实现的栈称为链栈。通常链栈用单链表表示，因此其结点结构与单链表的结构相同，在此用 LinkStack 表示，即有：

```
typedef  struct node
    { datatype data;
      struct node *next;
    }StackNode，* LinkStack;
```

说明 top 为栈顶指针： LinkStack top ;

因为栈中的主要运算是在栈顶插入、删除，显然在链表的头部做栈顶是最方便的，而且没有必要象单链表那样为了运算方便附加一个头结点。通常将链栈表示成图 3.3 的形式。链栈基本操作的实现如下：

⑴ 置空栈

```
LinkStack  Init_LinkStack ()
    {  return  NULL;
    }
```

⑵ 判栈空

```
int  Empty_LinkStack (LinkStack  top )
   {  if (top==-1)  return 1;
   else  return  0;
   }
```

⑶ 入栈

```
LinkStack  Push_LinkStack (LinkStack  top, datatype x)
  { StackNode  *s;
    s=malloc (sizeof (StackNode)) ;
    s->data=x;
    s->next=top;
    top=s;
    return top;
  }
```

⑷ 出栈

```
LinkStack   Pop_LinkStack (LinkStack  top, datatype  *x)
   {  StackNode  *p;
      if  (top= =NULL)  return NULL;
     else { *x = top->data;
            p = top;
            top = top->next;
            free (p);
            return  top;
        }
     }
```

3．2　栈的应用举例

由于栈的“先进先出”特点，在很多实际问题中都利用栈做一个辅助的数据结构来进行求解，下面通过几个例子进行说明。

例 3.1 简单应用：数制转换问题

所转换的 8 进制数按底位到高位的顺序产生的，而通常的输出是从高位到低位的，恰好与计算过程相反，因此转换过程中每得到一位 8 进制数则进栈保存，转换完毕后依次出栈则正好是转换结果。

算法思想如下：当 N>0 时重复 1，2

1. 若 N≠0，则将 N % r 压入栈 s 中 ，执行 2;若 N=0，将栈 s 的内容依次出栈，算法结束。
2. 用 N / r 代替 N

算法如下：

```
typedef  int datatype;
void conversion(int N, int r)
{ SeqStack  s;
  datetype   x;
  Init_SeqStack(&s);
   while ( N )
      { Push_SeqStack ( &s , N % r );
          N=N / r ;
      }
  while  ( Empty_SeqStack(& s ) )
     {  Pop_SeqStack (&s , &x ) ;
        printf ( “ %d ”, x ) ;
     }
  }
```

算法 3.1(a)

```
#define L  10
  void conversion(int N, int r)
{  int  s[L],top;      /*定义一个顺序栈*/
   int   x;
        top =-1;   /*初始化栈*/
    while ( N )
       {  s[++top]=N%r;        /*余数入栈 */
         N=N / r ;   /* 商作为被除数继续 */
      }
    while (top!=-1)
    { x=s[top--];
         printf(“%d”,x);
    }
}
```

算法 3.1(b)

算法 3.1(a)是将对栈的操作抽象为模块调用，使问题的层次更加清楚。而算法 3.1(b)中的直接用 int 向量 S 和 int 变量 top 作为一个栈来使用，往往初学者将栈视为一个很复杂的东西，不知道如何使用，通过这个例子可以消除栈的“神秘”，当应用程序中需要一个与数据保存时相反顺序使用数据时，就要想到栈。通常用顺序栈较多，因为很便利。

在后面的例子中，为了在算法中表现出问题的层次，有关栈的操作调用了的相关函数，如象算法 3.1(a)那样，对余数的入栈操作：Push_SeqStack (&s , N % r); 因为是用 c 语言描述，第一个参数是栈的地址才能对栈进行加工。在后面的例子中，为了算法的清楚易读，在不至于混淆的情况下，不再加地址运算符，请读者注意。

例 3.2　利用栈实现迷宫的求解。

问题：把一只老鼠从一个无顶盖的大盒子的入口处赶进迷宫。迷宫中设置很多隔壁，对前进方向形成了多处障碍，心理学家在迷宫的唯一出口处放置了一块奶酪，吸引老鼠在迷宫中寻找通路以到达出口。

求解思想：回溯法是一种不断试探且及时纠正错误的搜索方法。下面的求解过程采用回溯法。从入口出发，按某一方向向前探索，若能走通（未走过的），即某处可以到达，则到达新点，否则试探下一方向 ；若所有的方向均没有通路，则沿原路返回前一点，换下一个方向再继续试探，直到所有可能的通路都探索到，或找到一条通路，或无路可走又返回到入口点。

在求解过程中，为了保证在到达某一点后不能向前继续行走（无路）时，能正确返回前一点以便继续从下一个方向向前试探，则需要用一个栈保存所能够到达的每一点的下标及从该点前进的方向。

需要解决的四个问题：

1. 表示迷宫的数据结构：

设迷宫为 m 行 n 列，利用 maze[m][n]来表示一个迷宫，maze[i][j]=0 或 1; 其中：0 表示通路，1 表示不通，当从某点向下试探时，中间点有 8 个方向可以试探，（见图 3.4）而四个角点有 3 个方向，其它边缘点有 5 个方向，为使问题简单化我们用 maze[m+2][n+2]来表示迷宫，而迷宫的四周的值全部为 1。这样做使问题简单了，每个点的试探方向全部为 8，不用再判断当前点的试探方向有几个，同时与迷宫周围是墙壁这一实际问题相一致。

迷宫的定义如下：

```
#define  m   6  /* 迷宫的实际行   */
#define  n   8   /*   迷宫的实际列 */
```

```
int maze [m+2][n+2] ;
```

2. 试探方向：

在上述表示迷宫的情况下，每个点有 8 个方向去试探，如当前点的坐标(x,y)，与其相邻的 8 个点的坐标都可根据与该点的相邻方位而得到。因为出口在（m，n)，因此试探顺序规定为：从当前位置向前试探的方向为从正东沿顺时针方向进行。为了简化问题，方便的求出新点的坐标，将从正东开始沿顺时针进行的这 8 个方向的坐标增量放在一个结构数组 move [8]中，在 move 数组中，每个元素有两个域组成，x：横坐标增量，y：纵坐标增量。move 数组如图 3.6 所示。

Move 数组定义如下：

```
typedef  struct
   { int x，y
   } item ;
item move[8] ;
```

这样对 move 的设计会很方便地求出从某点 (x，y) 按某一方向 v (0<=v<=7) 到达的新点（i，j）的坐标：i=x+move[v].x ；j=y+move[v].y ；

3. 栈的设计：

当到达了某点而无路可走时需返回前一点，再从前一点开始向下一个方向继续试探。因此，压入栈中的不仅是顺序到达的各点的坐标，而且还要有从前一点到达本点的方向。对于图 3.4 所示迷宫，依次入栈为：

栈中每一组数据是所到达的每点的坐标及从该点沿哪个方向向下走的，对于图 3.4 迷宫，走的路线为：$(1,1)_1 \rightarrow (2,2)_1 \rightarrow (3,3)_0 \rightarrow (3,4)_0 \rightarrow (3,5)_0 \rightarrow (3,6)_0$（下脚标表示方向），当从点(3,6)沿方向 0 到达点(3,7)之后，无路可走，则应回溯，即退回到点(3,6)，对应的操作是出栈，沿下一个方向即方向 1 继续试探，方向 1 、 2 试探失败，在方向 3 上试探成功，因此将(3,6,3)压入栈中，即到达了（4,5）点。

栈中元素是一个由行、列、方向组成的三元组，栈元素的设计如下：

```
typedef struct
   {int x , y , d ;/* 横纵坐标及方向*/
   }datatype ;
```

栈的定义仍然为： SeqStack s ;

4. 如何防止重复到达某点，以避免发生死循环：

一种方法是另外设置一个标志数组 mark[m][n]，它的所有元素都初始化为 0，一旦到达了某一点 (i , j)之后，使 mark[i][j] 置 1，下次再试探这个位置时就不能再走了。另一种方法是当到达某点（i , j）后使 maze[i][j] 置 -1，以便区别未到达过的点，同样也能起到防止走重复点的目的，本书采用后者方法，算法结束前可恢复原迷宫。

迷宫求解算法思想如下：

1. 栈初始化;
2. 将入口点坐标及到达该点的方向（设为－１）入栈
3. while (栈不空)

```
        { 栈顶元素＝＞（x , y , d）
          出栈 ;
          求出下一个要试探的方向 d++ ;
         while  （还有剩余试探方向时）
           {  if  （d 方向可走）
                 则 {  （x , y , d）入栈 ;
                       求新点坐标  (i, j ) ;
                        将新点（i , j）切换为当前点（x , y） ;
                         if  ( (x , y )= =(m,n) ) 结束 ;
                        else 重置 d=0 ;
                    }
            else  d++ ;
           }
         }
```

算法如下：

```
int  path(maze, move)
 int maze[m][n] ;
 item move[8] ;
{ SeqStack  s ;
  datetype  temp ;
  int x, y, d, i, j ;
  temp.x=1 ;  temp.y=1 ;  temp.d=-1 ;
  Push_SeqStack (s, temp) ;
  while (! Empty_SeqStack (s ) )
     {  Pop_SeqStack (s,&temp) ;
           x=temp.x ;  y=temp.y ;  d=temp.d+1 ;
           while  (d<8)
            {  i=x+move[d].x ;   j=y+move[d].y ;
               if  ( maze[i][j]= =0 )
                {  temp={x, y, d} ;
                   Push_SeqStack ( s, temp ) ;
                   x=i ;  y=j ;  maze[x][y]= -1 ;
                   if  (x==m&&y= =n)  return 1 ; /*迷宫有路*/
                   else  d=0 ;
                 }
               else  d++ ;
           } /*while (d<8)*/
     }  /*while */
  return  0 ;/*迷宫无路*/
}
```

算法 3.2

栈中保存的就是一条迷宫的通路。

例 3.3　表达式求值

表达式是由运算对象、运算符、括号组成的有意义的式子。运算符从运算对象的个数上分，有单目运算符和双目运算符；从运算类型上分，有算术运算、关系运算、逻辑运算。在此仅限于讨论只含二目运算符的算术表达式。

1.　中缀表达式求值：

中缀表达式：每个二目运算符在两个运算量的中间，假设所讨论的算术运算符包括：+ 、- 、*、/、%、^（乘方）和括号（）。设运算规则为：

. 运算符的优先级为：（）——>　^　——>*、/、%——>　+、-　；

. 有括号出现时先算括号内的，后算括号外的，多层括号，由内向外进行；

. 乘方连续出现时先算最右面的；

表达式作为一个满足表达式语法规则的串存储，如表达式“3*2^（4+2*2-1*3）-5”,它的的求值过程为：自左向右扫描表达式，当扫描到 3*2 时不能马上计算，因为后面可能还有更高的运算，正确的处理过程是：需要两个栈：对象栈 s1 和算符栈 s2。当自左至右扫描表达式的每一个字符时，若当前字符是运算对象，入对象栈，是运算符时，若这个运算符比栈顶运算符高则入栈，继续向后处理，若这个运算符比栈顶运算符低则从对象栈出栈两个运算量，从算符栈出栈一个运算符进行运算，并将其运算结果入对象栈，继续处理当前字符，直到遇到结束符。

根据运算规则，左括号“（”在栈外时它的级别最高，而进栈后它的级别则最低了；乘方运算的结合性是自右向左，所以，它的栈外级别高于栈内；就是说有的运算符栈内栈外的级别是不同的。当遇到右括号“）”时，一直需要对运算符栈出栈，并且做相应的运算，直到遇到栈顶为左括号“(”时，将其出栈，因此右括号“）”级别最低但它是不入栈的。对象栈初始化为空，为了使表达式中的第一个运算符入栈，算符栈中预设一个最低级的运算符“（”。根据以上分析，每个运算符栈内、栈外的级别如下：

算符	栈内级别	栈外级别
^	3	4
*、/、%	2	2
+、-	1	1

(	0	4
)	-1	-1

中缀表达式表达式 “3*2^（4+2*2-1*3）-5”求值过程中两个栈的状态情况见图 3.7 所示。

读字符	对象栈 s1	算符栈 s2	说明
3	3	(	3 入栈 s1
*	3	(*	*入栈 s2
2	3，2	(*	2 入栈 s1
^	3，2	(*^	^入栈 s2
(	3，2	(*^(	(入栈 s2
4	3，2，4	(*^(	4 入栈 s1
+	3，2，4	(*^(+	+入栈 s2
2	3，2，4，2	(*^(+	2 入栈 s1
*	3，2，4，2	(*^(+*	*入栈 s2
2	3，2，4，2，2	(*^(+*	2 入栈 s1
-	3，2，4，4	(*^(+	做 2+2=4，结果入栈 s1
	3，2，8	(*^(	做 4+4=8，结果入栈 s2
	3，2，8	(*^(-	-入栈 s2
1	3，2，8，1	(*^(-	1 入栈 s1
*	3，2，8，1	(*^(-*	*入栈 s2
3	3，2，8，1，3	(*^(-*	3 入栈 s1
)	3，2，8，3	(*^(-	做 1*3，结果 3 入栈 s1
	3，2，5	(*^(	做 8-3，结果 5 入栈 s2
	3，2，5	(*^	(出栈
-	3，32	(*	做 2^5，结果 32 入栈 s1
	96	(	做 3*32，结果 96 入栈 s1
	96	(-	-入栈 s2
5	96，5	(-	5 入栈 s1
结束符	91	(	做 96-5，结果 91 入栈 s1

图 3.7　中缀表达式 3*2^（4+2*2-1*3）-5　的求值过程

为了处理方便，编译程序常把中缀表达式首先转换成等价的后缀表达式，后缀表达式的运算符在运算对象之后。在后缀表达式中，不在引入括号，所有的计算按运算符出现的顺序，严格从左向右进行，而不用再考虑运算规则和级别。中缀表达式“3*2^（4+2*2-1*3）-5 ”的后缀表达式为：“32422*+13*-^*5-”。

2.　后缀表达式求值

计算一个后缀表达式，算法上比计算一个中缀表达式简单的多。这是因为表达式中即无括号又无优先级的约束。具体做法：只使用一个对象栈，当从左向右扫描表达式时，每遇到一个操作数就送入栈中保存，每遇到一个运算符就从栈中取出两个操作数进行当前的计算，然后把结果再入栈，直到整个表达式结束，这时送入栈顶的值就是结果。

下面是后缀表达式求值的算法，在下面的算法中假设，每个表达式是合乎语法的，并且假设后缀表达式已被存入一个足够大的字符数组 A 中，且以‘#’为结束字符，为了简化问题，限定运算数的位数仅为一位且忽略了数字字符串与相对应的数据之间的转换的问题。

```
typedef  char datetype ;
double  calcul_exp(char *A)
{    /*本函数返回由后缀表达式 A 表示的表达式运算结果*/
  Seq_Starck  s ;
  ch=*A++ ; Init_SeqStack(s) ;
  while ( ch != ’ #’ )
     {
       if  (ch!=运算符)  Push_SeqStack (s , ch) ;
      else { Pop_SeqStack (s , &a) ;
                Pop_SeqStack (s , &b) ; /*取出两个运算量*/
              switch (ch).
               { case ch= =’ +’ :    c =a+b ; break ;
                 case ch= =’ -’ :    c=a-b ; break ;
                 case ch= =’ *’ :   c=a*b ; break ;
                 case ch= =’ / ’ :   c=a/b ; break ;
```

```
                case ch= =’%’ :   c=a%b ; break ;
               }
             Push_SeqStack (s, c) ;
           }
      ch=*A++ ;
     }
  Pop _SeqStack ( s , result ) ;
  return  result ;
}
```

例 3.4　栈与递归：

栈的一个重要应用是在程序设计语言中实现递归过程。现实中，有许多实际问题是递归定义的，这时用递归方法可以使许多问题的结果大大简化，以 n!为例：

n!的定义为：

$$n! = \begin{cases} 1 & n=0 \quad /*递归终止条件*/ \\ n*(n-1) & n>0 \quad /*递归步骤*/ \end{cases}$$

根据定义可以很自然的写出相应的递归函数

```
int fact (int n)
  {  if (n= =0)  return 1 ;
   else return  (n* fact (n-1) ) ;
  }
```

递归函数都有一个终止递归的条件，如上例 n=0 时，将不再继续递归下去。

递归函数的调用类似于多层函数的嵌套调用，只是调用单位和被调用单位是同一个函数而已。在每次调用时系统将属于各个递归层次的信息组成一个活动记录（ActivaTion Record），这个记录中包含着本层调用的实参、返回地址、局部变量等信息，并将这个活动记录保存在系统的“递归工作栈”中，每当递归调用一次，就要在栈顶为过程建立一个新的活动记录，一旦本次调用结束，则将栈顶活动记录出栈，根据获得的返回地址信息返回到本次的调用处。下面以求 3!为例说明执行调用时工作栈中的状况。

为了方便将求阶乘程序修改如下：

```
main ()
{ int m，n=3 ;
 m=fact (n) ;
 R1：
printf (“%d!=%d\n”，n，m) ;
}

int fact (int n)
{  int  f ;
   if (n= =0)  f=1 ;
   else f=n*fact (n-1) ;
 R2:
   return f  ;
}
```

算法 3.4

其中 R1 为主函数调用 fact 时返回点地址，R2 为 fact 函数中递归调用 fact (n -1)时返回点地址。

3.3　队列

3.3.1　队列的定义及基本运算

实际问题中还经常使用一种“先进先出” (FIFO---First In First Out)的数据结构：即插入在表一端进行，而删除在表的另一端进行，我们将这种数据结构称为队或队列，把允许插入的一端叫队尾(rear) ，把允许删除的一端叫队头(front)。队列也是一种运算受限制的线性表，所以又叫先进先出表。

在队列上进行的基本操作有：

⑴ 队列初始化：Init_Queue(q)

初始条件： 队 q 不存在。

操作结果： 构造了一个空队。

⑵ 入队操作： In_Queue(q,x),

初始条件： 队 q 存在。

操作结果： 对已存在的队列 q，插入一个元素 x 到队尾，队发生变化。

⑶ 出队操作： Out_Queue(q,x)

初始条件: 队 q 存在且非空

操作结果： 删除队首元素，并返回其值，队发生变化。

⑷ 读队头元素：Front_Queue(q,x)

初始条件: 队 q 存在且非空

操作结果： 读队头元素，并返回其值，队不变;

⑸ 判队空操作：Empty_Queue(q)

初始条件： 队 q 存在

操作结果： 若 q 为空队则返回为 1，否则返回为 0。

3.3.2 队列的存储实现及运算实现

队列也有顺序存储和链式存储两种存储方法。

1 顺序队

顺序存储的队称为顺序队。因为队的队头和队尾都是活动的，因此，除了队列的数据区外还有队头、队尾两个指针。顺序队的类型定义如下：

```
define  MAXSIZE   1024   /*队列的最大容量*/
typedef  struct
 {datatype   data[MAXSIZE];  /*队员的存储空间*/
  int rear,front;  /*队头队尾指针*/
 }SeQueue;
```

定义一个指向队的指针变量：

```
 SeQueue  *sq;
```

申请一个顺序队的存储空间：

```
 sq=malloc(sizeof(SeQueue));
```

队列的数据区为：

```
  sq->data[0]---sq->data[MAXSIZE -1]
```

队头指针：sq->front

队尾指针：sq->rear

设队头指针指向队头元素前面一个位置，队尾指针指向队尾元素（这样的设置是为了某些运算的方便，并不是唯一的方法）。

置空队则为：sq->front=sq->rear=-1;

在不考虑溢出的情况下，入队操作队尾指针加 1，指向新位置后，元素入队。

操作如下：

```
    sq->rear++;
    sq->data[sq->rear]=x;  /*原队头元素送 x 中*/
```

在不考虑队空的情况下，出队操作队头指针加 1，表明队头元素出队。

操作如下：

```
    sq->front++;
    x=sq->data[sq->front];
```

队中元素的个数：m=(sq->rear)-(q->front);

队满时：m= MAXSIZE; 队空时：m=0。

解决假溢出的方法之一是将队列的数据区 data[0..MAXSIZE-1]看成头尾相接的循环结构，头尾指针的关系不变，将其称为“循环队”。

因为是头尾相接的循环结构，入队时的队尾指针加 1 操作修改为：

```
sq->rear=(sq->rear+1) % MAXSIZE;
```

出队时的队头指针加 1 操作修改为：

```
sq->front=(sq->front+1) % MAXSIZE;
```

即在队空情况下也有：front==rear。就是说“队满”和“队空”的条件是相同的了。这显然是必须要解决的一个问题。

方法之一是附设一个存储队中元素个数的变量如 num，当 num==0 时队空，当 num==MAXSIZE 时为队满。

另一种方法是少用一个元素空间，把图（d）所示的情况就视为队满，此时的状态是队尾指针加 1 就会从后面赶上队头指针，这种情况下队满的条件是：(rear+1) % MAXSIZE==front，也能和空队区别开。

下面的循环队列及操作按第一种方法实现。

循环队列的类型定义及基本运算如下：

```
typedef  struct     {
    datatype data[MAXSIZE];  /*数据的存储区*/
    int front,rear;  /*队头队尾指针*/
    int num;          /*队中元素的个数*/
  }c_SeQueue;    /*循环队*/
```

⑴ 置空队

```
c_SeQueue*  Init_SeQueue()
{  q=malloc(sizeof(c_SeQueue));
    q->front=q->rear=MAXSIZE-1;
    q->num=0;
    return q;
 }
```

⑵ 入队

```
int   In_SeQueue ( c_SeQueue *q , datatype  x)
   {  if  (num==MAXSIZE)
     { printf(" 队满 ");
         return  -1;   /*队满不能入队*/
      }
     else
      { q->rear=(q->rear+1) % MAXSIZE;
        q->data[q->rear]=x;
        num++;
        return 1;    /*入队完成*/
        }
  }
```

⑶ 出队

```
int   Out_SeQueue (c_SeQueue *q , datatype  *x)
  {  if  (num==0)
       {  printf(" 队空 ");
           return  -1;   /*队空不能出队*/
        }
     else
      { q->front=(q->front+1) % MAXSIZE;
        *x=q->data[q->front]; /*读出队头元素*/
        num--;
        return 1;    /*出队完成*/
        }
```

```
    }
```

(4) 判队空

```
int   Empty_SeQueue(c_SeQueue  *q)
    {  if  (num==0)  return 1;
       else return 0;
     }
```

2. 链队

链式存储的队称为链队。和链栈类似，用单链表来实现链队，根据队的 FIFO 原则，为了操作上的方便，我们分别需要一个头指针和尾指针。

头指针 front 和尾指针 rear 是两个独立的指针变量，从结构性上考虑，通常将二者封装在一个结构中。

链队的描述如下：

```
typedef struct node
 { datatype  data;
   struct  node *next;
 } QNode;     /*链队结点的类型*/
typedef struct
 { QNnode   *front,*rear;
 }LQueue;     /*将头尾指针封装在一起的链队*/
```

定义一个指向链队的指针：

```
LQueue  *q;
```

链队的基本运算如下：

(1) 创建一个带头结点的空队：

```
LQueue  *Init_LQueue()
    {  LQueue *q,*p;
       q=malloc(sizeof(LQueue)); /*申请头尾指针结点*/
       p=malloc(sizeof(QNode));  /*申请链队头结点*/
       p->next=NULL;   q->front=q->rear=p;
       return q;
    }
```

(2) 入队

```
  void In_LQueue(LQueue *q , datatype  x)
     {  QNode *p;
        p=malloc(sizeof(QNnode));  /*申请新结点*/
        p->data=x;   p->next=NULL;
        q->rear->next=p;
        q->rear=p;
     }
```

(3) 判队空

```
int  Empty_LQueue( LQueue *q)
{ if (q->front==q->rear)   return 0;
 else  return 1;
}
```

(4) 出队

```
  int Out_LQueue(LQueue *q , datatype  *x)
     {  QNnode *p;
        if (Empty_LQueue(q) )
```

```
        { printf ("队空"); return 0;
        }   /*队空，出队失败*/
    else
       { p=q->front->neat;
         q->front->next=p->next;
         *x=p->data;/*队头元素放 x 中*/
         free(p);
         if (q->front->next==NULL)
         q->rear=q->front;
              /*只有一个元素时，出队后队空，此时还要要修改队尾指针参考图 3.16(c)*/
        return 1;
       }
  }
```

3.4 队列应用举例

例 3.5 求迷宫的最短路径：现要求设计一个算法找一条从迷宫入口到出口的最短路径。

本算法要求找一条迷宫的最短路径，算法的基本思想为：从迷宫入口点（1,1）出发，向四周搜索，记下所有一步能到达的坐标点；然后**依次**再从这些点出发，再记下所有一步能到达的坐标点，…，依此类推，直到到达迷宫的出口点(m,n)为止，然后从出口点沿搜索路径回溯直至入口。这样就找到了一条迷宫的最短路径，否则迷宫无路径。

有关迷宫的数据结构、试探方向、如何防止重复到达某点以避免发生死循环的问题与例 3.2 处理相同，不同的是：如何存储搜索路径。在搜索过程中必须记下每一个可到达的坐标点，以便从这些点出发继续向四周搜索。由于先到达的点先向下搜索，故引进一个“先进先出”数据结构------队列来保存已到达的坐标点。到达迷宫的出口点(m,n)后，为了能够从出口点沿搜索路径回溯直至入口，对于每一点，记下坐标点的同时，还要记下到达该点的前驱点，因此，用一个结构数组 sq[num]作为队列的存储空间，因为迷宫中每个点至多被访问一次，所以 num 至多等于 m*n。sq 的每一个结构有三个域：x,y 和 pre，其中 x,y 分别为所到达的点的坐标，pre 为前驱点在 sq 中的坐标，是一个静态链域。除 sq 外，还有队头、队尾指针：front 和 rear 用来指向队头和队尾元素。

队的定义如下：

```
typedef  struct
   {  int x,y;
      int pre;
   }sqtype;
sqtype sq[num];
int front,rear;
```

初始状态，队列中只有一个元素 sq[1]记录的是入口点的坐标（1，1），因为该点是出发点，因此没有前驱点，pre 域为－1，队头指针 front 和队尾指针 rear 均指向它，此后搜索时都是以 front 所指点为搜索的出发点，当搜索到一个可到达点时，即将该点的坐标及 front 所指点的位置入队，不但记下了到达点的坐标，还记下了它的前驱点。front 所指点的 8 个方向搜索完毕后，则出队，继续对下一点搜索。搜索过程中遇到出口点则成功，搜索结束，打印出迷宫最短路径，算法结束；或者当前队空即没有搜索点了，表明没有路径算法也结束。

算法如下：

```
void path(maze, move)
 int maze[m][n]; /*迷宫数组*/
 item move[8]; /*坐标增量数组*/
{ sqtype sq[NUM];
 int front,rear;
 int x,y,i,j,v;
front=rear=0;
 sq[0].x=1; sq[0].y=1; sq[0].pre=-1;  /*入口点入队*/
 maze[1,1]=-1;
 while (front<=rear)  /*队列不空*/
     {  x=sq[front].x ;  y=sq[front ].y ;
```

```
        for (v=0;v<8;v++)
          { i=x+move[v].x;  j=x+move[v].y;
            if (maze[i][j]==0)
              {  rear++;
                 sq[rear].x=i; sq[rear].y=j;  sq[rear].pre=front;
                 maze[i][j]=-1;
              }
            if (i==m&&j==n)
              {  printpath(sq,rear);  /*打印迷宫*/
                 restore(maze);    /*恢复迷宫*/
                 return 1;
              }
          }  /*for v*/
       front++;  /*当前点搜索完，取下一个点搜索  */
     }  /*while*/
   return 0;
  } /*path*/

  void printpath(sqtype sq[],int rear)  /*打印迷宫路径*/
  {  int   i;
     i=rear;
      do {  printf("  (%d,%d)← ",sq[i].x , sq[i].y) ;
          i=sq[i].pre;  /*回溯*/
        } while (i!=-1);
   } /*printpath*/
```

算法 3.5

在上面的例子中，不能采用循环队列，因为在本问题中，队列中保存了探索到的路径序列，如果用循环队列，则把先前得到的路径序列覆盖掉。而在有些问题中，如持续运行的实时监控系统中，监控系统源源不断的收到监控对象顺序发来的信息如报警，为了保持报警信息的顺序性，就要按顺序一一保存，而这些信息是无穷多个，不可能全部同时驻留内存，可根据实际问题，设计一个适当大的向量空间，用作循环队列，最初收到的报警信息一一入队，当队满之后，又有新的报警到来到时，新的报警则覆盖掉了旧的报警，内存中始终保持当前最新的若干条报警，以便满足快速查询。

第四章　串

串（即字符串）是一种特殊的线性表，它的数据元素仅由一个字符组成，计算机非数值处理的对象经常是字符串数据，如在汇编和高级语言的编译程序中，源程序和目标程序都是字符串数据；在事物处理程序中，顾客的姓名、地址、货物的产地、名称等，一般也是作为字符串处理的。另外串还具有自身的特性，常常把一个串作为一个整体来处理，因此，在这一章我们把串作为独立结构的概念加以研究，介绍串的串的存储结构及基本运算。

4.1　串及其基本运算

4.1.1 串的基本概念

1．串的定义

串是由零个或多个任意字符组成的字符序列。一般记作：s=＂s_1 s_2… s_n＂

其中 s 是串名；在本书中，用双引号作为串的定界符，引号引起来的字符序列为串值，引号本身不属于串的内容；a_i(1<=i<=n)是一个任意字符，它称为串的元素，是构成串的基本单位，i 是它在整个串中的序号； n 为串的长度，表示串中所包含的字符个数，当 n=0 时，称为空串，通常记为 Φ。

2．几个术语

子串与主串：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。

子串的位置：子串的第一个字符在主串中的序号称为子串的位置。

串相等：称两个串是相等的，是指两个串的长度相等且对应字符都相等。

4.1.2 串的基本运算

1． 求串长 StrLength(s)

操作条件：串 s 存在

操作结果：求出串 s 的长度。

2． 串赋值 StrAssign(s1,s2)

操作条件： s1 是一个串变量，s2 或者是一个串常量，或者是一个串变量（通常 s2 是一个串常量时称为串赋值，是一个串变量称为串拷贝）。

操作结果：将 s2 的串值赋值给 s1， s1 原来的值被覆盖掉。

3． 连接操作 ：StrConcat (s1,s2,s) 或 StrConcat (s1,s2)

操作条件：串 s1,s2 存在。

操作结果：两个串的联接就是将一个串的串值紧接着放在另一个串的后面，连接成一 个串。前者是产生新串 s，s1 和 s2 不改变; 后者是在 s1 的后面联接 s2 的串值，s1 改变， s2 不改变。

例如： s1=＂he＂，s2=＂ bei＂，前者操作结果是 s=＂he bei＂；后者操作结果是 s1=＂he bei＂。

4． 求子串 SubStr (s,i,len)：

操作条件：串 s 存在，1≤i≤StrLength(s)，0≤len≤StrLength(s)-i+1。

操作结果：返回从串 s 的第 i 个字符开始的长度为 len 的子串。len=0 得到的是空串。

例如：SubStr(＂abcdefghi＂,3,4)= ＂cdef＂

5． 串比较 StrCmp(s1,s2)

操作条件：串 s1,s2 存在。

操作结果：若 s1==s2，操作返回值为 0；若 s1<s2, 返回值<0；若 s1>s2, 返回值>0。

6． 子串定位 StrIndex(s,t)：找子串 t 在主串 s 中首次出现的位置

操作条件：串 s,t 存在。

操作结果：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。

如：StrIndex(＂abcdebda＂, ＂bc＂)=2

StrIndex(＂abcdebda＂, ＂ba＂)=-1

7． 串插入 StrInsert(s,i,t)

操作条件：串 s,t 存在，1≤i≤StrLength(s)+1。

操作结果：将串 t 插入到串 s 的第 i 个字符位置上，s 的串值发生改变。

8． 串删除 StrDelete(s,i,len)

操作条件：串 s 存在，1≤i≤StrLength(s)，0≤len≤StrLength(s)-i+1。

操作结果：删除串 s 中从第 i 个字符开始的长度为 len 的子串，s 的串值改变。

9． 串替换 StrRep(s,t,r)

操作条件：串 s,t,r 存在，t 不为空。

操作结果：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，s 的串值改变。

以上是串的几个基本操作。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 个基本操称为最小操作集。

4.2 串的定长顺序存储及基本运算

4.2.1 串的定长顺序存储

类似于顺序表，用一组地址连续的存储单元存储串值中的字符序列，所谓定长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，如：

```
#define MAXSIZE  256
char  s[MAXSIZE];
```

则串的最大长度不能超过 256。

如何标识实际长度？

1. 类似顺序表，用一个指针来指向最后一个字符，这样表示的串描述如下：

```
typedef struct
 {  char  data[MAXSIZE];
     int   curlen;
```

} SeqString;

定义一个串变量：SeqString s;

这种存储方式可以直接得到串的长度：s.curlen+1。

2. 在串尾存储一个不会在串中出现的特殊字符作为串的终结符，以此表示串的结尾。比如 C 语言中处理定长串的方法就是这样的，它是用’\0’来表示串的结束。这种存储方法不能直接得到串的长度，是用判断当前字符是否是’\0’来确定串是否结束，从而求得串的长度。

```
char  s[MAXSIZE];
```

3. 设定长串存储空间：char s[MAXSIZE+1]; 用 s[0]存放串的实际长度，串值存放在 s[1]~s[MAXSIZE]，字符的序号和存储位置一致，应用更为方便。

4.2.2 定长顺序串的基本运算

串定位在下一小节讨论，设串结束用′\0′来标识。

1. 串联接：把两个串 s1 和 s2 首尾连接成一个新串 s ，即：s<=s1+s2。

```
int StrConcat1(s1,s2,s)
  char s1[],s2[],s[];
{ int i=0 , j, len1, len2;
  len1= StrLength(s1); len2= StrLength(s2)
  if  (len1+ len2>MAXSIZE-1)  return  0 ; /* s 长度不够*/
  j=0;
  while(s1[j]!=’\0’)  { s[i]=s1[j];i++; j++; }
  j=0;
  while(s2[j]!=’\0’)  { s[i]=s2[j];i++; j++; }
  s[i]=’\0’;   return 1;
}
```

算法 4.1

2．求子串

```
int StrSub (char *t, char *s, int i, int len)
  /* 用 t 返回串 s 中第个 i 字符开始的长度为 len 的子串 1≤i≤串长*/
 { int slen;
  slen=StrLength(s);
  if ( i<1 || i>slen || len<0 || len>slen-i+1)
        { printf(" 参数不对 "); return 0; }
  for (j=0; j<len; j++)
    t[j]=s[i+j-1];
  t[j]=’\0’;
  return 1;
  }
```

算法 4.2

3． 串比较

```
int StrComp(char *s1, char *s2)
{ int i=0;
 while (s1[i]==s2[i] && s1[i]!=’\0’)  i++;
 return (s1[i]-s2[i]);
}
```

算法 4.3

4.2.3 模式匹配

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。为了运算方便，设字符串的长度存放在 0 号单元，串值从 1 号单元存放，这样字符序号与存储位置一致。

1．简单的模式匹配算法

算法思想如下：首先将 s_1 与 t_1 进行比较，若不同，就将 s_2 与 t_1 进行比较，...，直到 s 的某一个字符 s_i 和 t_1 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+2}，t 返回到 t_1，继续开始下一趟的比较，重复上述过程。若 t 中的字符全部比完，则说明本趟匹配成功，本趟的起始位置是 i-j+1 或 i-t[0]，否则，匹配失败。

依据这个思想，算法描述如下:

```
int  StrIndex_BF  (char *s,char *t)
    /*从串 s 的第一个字符开始找首次与串 t 相等的子串*/
{  int i=1,j=1;
   while (i<=s[0] && j<=t[0] )  /*都没遇到结束符*/
        if (s[i]==t[j])
          { i++;j++; }    /*继续*/
        else
          {i=i-j+2; j=1; } /*回溯*/
   if (j>t[0])  return (i-t[0]);  /*匹配成功，返回存储位置*/
   else  return  -1;
}
```

算法 4.4

该算法简称为 BF 算法。下面分析它的时间复杂度，设串 s 长度为 n，串 t 长度为 m。

匹配成功的情况下，考虑两种极端情况：

在最好情况下，每趟不成功的匹配都发生在第一对字符比较时：

例如：s=" aaaaaaaaaabc "

t=" bc "

设匹配成功发生在 s_i 处，则字符比较次数在前面 i-1 趟匹配中共比较了 i-1 次，第 i 趟成功的匹配共比较了 m 次，所以总共比较了 i-1+m 次，所有匹配成功的可能共有 n-m+1 种，设从 s_i 开始与 t 串匹配成功的概率为 p_i，在等概率情况下 $p_i=1/(n-m+1)$，因此最好情况下平均比较的次数是：

即 最 $\sum_{i=1}^{n-m+1} p_i\times(i-1+m)=\sum_{i=1}^{n-m+1}\frac{1}{n-m+1}\times(i-1+m)=\frac{(n+m)}{2}$ 好情况下的时间复杂度是 O(n+m)。

在最坏情况下，每趟不成功的匹配都发生在 t 的最后一个字符：

例如：s=" aaaaaaaaaaab "

t=" aaab "

设匹配成功发生在 s_i 处，则在前面 i-1 趟匹配中共比较了(i-1)*m 次，第 i 趟成功的匹配共比较了 m 次，所以总共比较了 i*m 次，因此最坏好情况下平均比较的次数是：

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

即 最 坏情况下的时间复杂度是 O(n*m)。

上述算法中匹配是从 s 串的第一个字符开始的，有时算法要求从指定位置开始，这时算法的参数表中要加一个位置参数 pos：StrIndex(shar *s,int pos,char *t)，比较的初始位置定位在 pos 处。算法 4.4 是 pos=1 的情况。

*2．改进后的模式匹配算法

(1) KMP 算法的思想

分析算法 4.4 的执行过程，造成 BF 算法速度慢的原因是回溯，即在某趟的匹配过程失败后，对于 s 串要回到本趟开始字符的下一个字符，t 串要回到第一个字符。而这些回溯并不是必要的。如图 4.3 所示的匹配过程，在第三趟匹配过程中，s_3～s_6 和 t_1～t_4 是匹配成功的，$s_7\neq t_5$ 匹配失败，因此有了第四趟，其实这一趟是不必要的：由图可看出，因为在第三趟中有 $s_4=t_2$，而 $t_1\neq t_2$，肯定有 $t_1\neq s_4$。同理第五趟也是没有必要的，所以从第三趟之后可以直接到第六趟，进一步分析第六趟中的第一对字符 s_6 和 t_1 的比较也是多余的，因为第三趟中已经比过了 s_6 和 t_4，并且 $s_6=t_4$，而 $t_1=t_4$，必有 $s_6=t_1$，因此第六趟的比较可以从第二对字符 s_7 和 t_2 开始

进行，这就是说，第三趟匹配失败后，指针 i 不动，而是将模式串 t 向右“滑动”，用 t_2 “对准” s_7继续进行，依此类推。这样的处理方法指针 i 是无回溯的。

综上所述，希望某趟在 s_i和 t_j匹配失败后，指针 i 不回溯，模式 t 向右“滑动”至某个位置上，使得 t_k 对准 s_i继续向右进行。显然，现在问题的关键是串 t“滑动”到哪个位置上？不妨设位置为 k，即 s_i和 t_j匹配失败后，指针 i 不动，模式 t 向右“滑动”，使 t_k和 s_i对准继续向右进行比较，要满足这一假设，就要有如下关系成立：

$$" t_1\ t_2\ \cdots\ t_{k-1}\ " = " s_{i-k+1}\ s_{i-k+2}\ \cdots\ s_{i-1}\ " \qquad (4.1)$$

(4.1)式左边是 t_k前面的 k-1 个字符，右边是 s_i 前面的 k-1 个字符。

而本趟匹配失败是在 s_i和 t_j之处，已经得到的部分匹配结果是：

$$" t_1\ t_2\ \cdots\ t_{j-1}\ " = " s_{i-j+1}\ s_{i-j+2}\ \cdots\ s_{i-1}\ " \qquad (4.2)$$

因为 k<j，所以有：

$$" t_{j-k+1}\ t_{j-k+2}\ \cdots\ t_{j-1}\ " = " s_{i-k+1}\ s_{i-k+2}\ \cdots\ s_{i-1}\ " \qquad (4.3)$$

(4.3)式左边是 t_j前面的 k-1 个字符，右边是 s_i 前面的 k-1 个字符，

通过(4.1)和(4.3)得到关系：

$$" t_1\ t_2\ \cdots\ t_{k-1}\ " = " t_{j-k+1}\ t_{j-k+2}\ \cdots\ t_{j-1}\ " \qquad (4.4)$$

结论：某趟在 s_i和 t_j匹配失败后，如果模式串中有满足关系(4)的子串存在，即：模式中的前 k-1 个字符与模式中 t_j字符前面的 k-1 个字符相等时，模式 t 就可以向右“滑动”至使 t_k和 s_i对准，继续向右进行比较即可。

（2）next 函数

模式中的每一个 t_j都对应一个 k 值，由(4.4)式可知，这个 k 值仅依赖与模式 t 本身字符序列的构成，而与主串 s 无关。我们用 next[j]表示 t_j对应的 k 值，根据以上分析，next 函数有如下性质：

① next[j]是一个整数，且 0≤next[j]<j

② 为了使 t 的右移不丢失任何匹配成功的可能，当存在多个满足(4.4)式的 k 值时，应取最大的，这样向右“滑动”的距离最短，“滑动”的字符为 j-next[j]个。

③ 如果在 t_j前不存在满足(4.4)式的子串，此时若 $t_1 \neq t_j$，则 k=1；若 $t_1=t_j$，则 k=0；这时“滑动”的最远，为 j-1 个字符即用 t_1和 s_{j+1}继续比较。

因此，next 函数定义如下 ：

$$\text{next}[j]=\begin{cases} 0 & \text{当 } j=1 \\ \max\{k \mid 1<=k<j \text{ 且 } " t_1\ t_2\ \cdots\ t_{k-1}\ " = " t_{j-k+1}\ t_{j-k+2}\ \cdots\ t_{j-1}\ "\} & \\ 1 & \text{当不存在上面的 k 且 } t_1 \neq t_j \\ 0 & \text{当不存在上面的 k 且 } t_1=t_i \end{cases}$$

设有模式串：t=" abcaababc "，则它的 next 函数值为：

j	1	2	3	4	5	6	7	8	9
模式串	a	b	c	a	a	b	a	b	c
next[j]	0	1	1	0	2	1	3	1	1

(3) KMP 算法

在求得模式的 next 函数之后，匹配可如下进行：假设以指针 i 和 j 分别指示主串和模式中的比较字符，令 i 的初值为 pos，j 的初值为 1。若在匹配过程中 $s_i \neq t_j$，则 i 和 j 分别增 1，若 $s_i \neq t_j$ 匹配失败后，则 i 不变，j 退到 next[j]位置再比较，若相等，则指针各自增 1，否则 j 再退到下一个 next 值的位置，依此类推。直至下列两种情况：一种是 j 退到某个 next 值时字符比较相等，则 i 和 j 分别增 1 继续进行匹配；另一种是 j 退到值为零（即模式的第一个字符失配），则此时 i 和 j 也要分别增 1，表明从主串的下一个字符起和模式重新开始匹配。

在假设已有 next 函数情况下，KMP 算法如下：

```
int StrIndex_KMP(char *s,char *t,int pos)
    /*从串 s 的第 pos 个字符开始找首次与串 t 相等的子串*/
  {  int i=pos,j=1,slen,tlen;
     while (i<=s[0] && j<=t[0] ) /*都没遇到结束符*/
          if (j==0||s[i]==t[j]) { i++; j++; }
        else  j=next[j];   /*回溯*/
     if (j>t[0])  return  i-t[0];  /*匹配成功，返回存储位置*/
     else  return  -1;
```

```
}
```

算法 4.5

图 4.4 利用模式 next 函数进行匹配的过程示例

(4)如何求 next 函数

由以上讨论知，next 函数值仅取决于模式本身而和主串无关。我们可以从分析 next 函数的定义出发用递推的方法求得 next 函数值。

由定义知：

$$next[1]=0 \tag{4.5}$$

设 next[j]=k，即有：

$$"t_1\ t_2\ \cdots\ t_{k-1}"="t_{j-k+1}\ t_{j-k+2}\ \cdots\ t_{j-1}" \tag{4.6}$$

next[j+1]=? 可能有两种情况：

第一种情况：若 $t_k=t_j$ 则表明在模式串中

$$"t_1\ t_2\ \cdots\ t_k"="t_{j-k+1}\ t_{j-k+2}\ \cdots\ t_j" \tag{4.7}$$

这就是说 next[j+1]=k+1，即

$$next[j+1]=next[j]+1 \tag{4.8}$$

第二种情况：若 $t_k\neq t_j$ 则表明在模式串中

$$"t_1\ t_2\ \cdots\ t_k"\neq"t_{j-k+1}\ t_{j-k+2}\ \cdots\ t_j" \tag{4.9}$$

此时可把求 next 函数值的问题看成是一个模式匹配问题，整个模式串既是主串又是模式，而当前在匹配的过程中，已有(4.6)式成立，则当 $t_k\neq t_j$ 时应将模式向右滑动，使得第 next[k]个字符和“主串”中的第 j 个字符相比较。若 next[k]=k′，且 $t_{k'}=t_j$，则说明在主串中第 j+1 个字符之前存在一个最大长度为 k′ 的子串，使得

$$"t_1\ t_2\ \cdots\ t_{k'}"="t_{j-k'+1}\ t_{j-k'+2}\ \cdots\ t_j" \tag{4.10}$$

因此： $$next[j+1]=next[k]+1 \tag{4.11}$$

同理若 $t_{k'}\neq t_j$，则将模式继续向右滑动至使第 next[k′]个字符和 t_j 对齐，依此类推，直至 t_j 和模式中的某个字符匹配成功或者不存在任何 k′(1< k′<k <…<j) 满足 (4.10)，此时若 $t_1\neq t_{j+1}$，则有：next[j+1]=1 (4.12)

否则若 $t_1=t_{j+1}$，则有：next[j+1]=0 (4.13)

综上所述，求 next 函数值过程的算法如下：

```
void GetNext(char *t,int next[ ])
     /*求模式 t 的 next 值并寸入 next 数组中*/
  {  int i=1,j=0;
     next[1]=0;
     while (i<t[0])
       { while (j>0&&t[i]!=t[j])  j=next[j];
         i++;  j++;
        if (t[i]==t[j])  next[i]=next[j];
        else  next[i]=j;
       }
  }
```

算法 4.6

算法 4.6 的时间复杂度是 O(m)；所以算法 4.5 的时间复杂度是 O(n*m)，但在一般情况下，实际的执行时间是 O(n+m)。

第五章 数组、特殊矩阵和广义表

5.1.1 数组的逻辑结构

数组是我们很熟悉的一种数据结构，它可以看作线性表的推广。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

数组是一个具有固定格式和数量的数据有序集，每一个数据元素有唯一的一组下标来标识，因此，在数组上不能做插入、删除

数据元素的操作。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。在数组中通常做下面两种操作：

(1) 取值操作：给定一组下标，读其对应的数据元素。

(2) 赋值操作：给定一组下标，存储或修改与其相对应的数据元素。

5.1.2 数组的内存映象

通常，数组在内存被映象为向量，即用向量作为数组的一种存储结构，这是因为内存的地址空间是一维的，数组的行列固定后，通过一个映象函数，则可根据数组元素的下标得到它的存储地址。

对于一维数组按下标顺序分配即可。

对多维数组分配时，要把它的元素映象存储在一维存储器中，一般有两种存储方式：一是以行为主序（或先行后列）的顺序存放，另一种是以列为主序（先列后行）的顺序存放即一列一列地分配。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。

设有 m×n 二维数组 A_{mn}，下面我们看按元素的下标求其地址的计算：

以“以行为主序”的分配为例：设数组的基址为 $LOC(a_{11})$，每个数组元素占据 l 个地址单元，那么 a_{ij} 的物理地址可用一线性寻址函数计算：

$$LOC(a_{ij}) = LOC(a_{11}) + ((i-1)*n + j-1) * l$$

这是因为数组元素 a_{ij} 的前面有 i-1 行，每一行的元素个数为 n，在第 i 行中它的前面还有 j-1 个数组元素。

在 C 语言中，数组中每一维的下界定义为 0，则：

$$LOC(a_{ij}) = LOC(a_{00}) + (i*n + j) * l$$

推广到一般的二维数组：$A[c_1..d_1]\ [c_2..d_2]$，则 a_{ij} 的物理地址计算函数为：

$$LOC(a_{ij})=LOC(a_{c1\ c2})+((i- c_1) *(d_2 - c_2 + 1)+ (j- c_2))*l$$

同理对于三维数组 A_{mnp}，即 m×n×p 数组，对于数组元素 a_{ijk} 其物理地址为：

$$LOC(a_{ijk})=LOC(a_{111})+((i-1) *n*p+ (j-1)*p +k-1)*l$$

推广到一般的三维数组：$A[c_1..d_1]\ [c_2..d_2]\ [c_3..d_3]$，则 a_{ijk} 的物理地址为：

$$LOC(i,j)=LOC(a_{c1\ c2\ c3})+((i- c_1) *(d_2 - c_2 + 1)* (d_3 - c_3 + 1)+ (j- c_2) *(d_3 - c_3 + 1)+(k- c_3))*l$$

例 5．1 若矩阵 $A_{m\times n}$ 中存在某个元素 a_{ij} 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，然后判断该元素它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：

```
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);
              } /* if */
          } /*for i*/
      }
```

算法的时间性能为 O(m*(n+m*n))。

5.2 特殊矩阵的压缩存储

5.2.1 对称矩阵

对称矩阵的特点是：在一个 n 阶方阵中，有 $a_{ij}=a_{ji}$ ，其中 1≤i ， j≤n，对称矩阵关于主对角线对称，因此只需存储上三角或下三角部分即可，这样，原来需要 n*n 个存储单元，现在只需要 n(n+1)/2 个存储单元了，节约了 n(n-1)/2 个存储单元，当 n 较大时，这是可观的一部分存储资源。

如何只存储下三角部分呢？对下三角部分以行为主序顺序存储到一个向量中去，在下三角中共有 n*(n+1)/2 个元素，因此，不失一般性，设存储到向量 SA[n(n+1)/2]中，存储顺序可用图 5.6 示意，这样，原矩阵下三角中的某一个元素 a_{ij} 则具体对应一个 sa_k，下面的问题是要找到 k 与 i、j 之间的关系。

对于下三角中的元素 a_{ij}，其特点是：i≥j 且 1≤i≤n，存储到 SA 中后，根据存储原则，它前面有 i-1 行，共有 1+2+…+i-1=i*(i-1)/2 个元素，而 a_{ij} 又是它所在的行中的第 j 个，所以在上面的排列顺序中，a_{ij} 是第 i*(i-1)/2+j 个元素，因此它在 SA 中的下标 k 与 i、j 的关系为：

$$k=i*(i-1)/2+j-1 \quad (0 \leqslant k < n*(n+1)/2)$$

若 i<j，则 a_{ij} 是上三角中的元素，因为 $a_{ij}=a_{ji}$ ，这样，访问上三角中的元素 a_{ij} 时则去访问和它对应的下三角中的 a_{ji} 即可，因此将上式中的行列下标交换就是上三角中的元素在 SA 中的对应关系：

$$k=j*(j-1)/2+i-1 \quad (0 \leqslant k < n*(n+1)/2)$$

综上所述，对于对称矩阵中的任意元素 a_{ij}，若令 I=max(i,j)，J=min(i,j)，则将上面两个式子综合起来得到： k=I*(I-1)/2+J-1。

5.2.2 三角矩阵

形如图 5.7 的矩阵称为三角矩阵，其中 c 为某个常数。其中 5.7(a)为下三角矩阵：主队角线以上均为同一个常数；(b)为上三角矩阵，主队角线以下均为同一个常数；下面讨论它们的压缩存储方法。

1. 下三角矩阵

与对称矩阵类似，不同之处在于存完下三角中的元素之后，紧接着存储对角线上方的常量，因为是同一个常数，所以存一个即可，这样一共存储了 n*(n+1)+1 个元素，设存入向量：SA[n*(n+1)+1]中，这种的存储方式可节约 n*(n-1)-1 个存储单元，sa_k 与 a_{ji} 的对应关系为：

$$k=\begin{cases} i*(i-1)/2+j-1 & \text{当 } i \geqslant j \\ n*(n+1)/2-1 & \text{当 } i<j \end{cases}$$

2. 上三角矩阵

对于上三角矩阵，存储思想与下三角类似，以行为主序顺序存储上三角部分，最后存储对角线下方的常量。对于第 1 行，存储 n 个元素，第 2 行存储 n-1 个元素，…，第 p 行存储(n-p+1)个元素，a_{ij} 的前面有 i-1 行，共存储：

$$n+(n-1)+\cdots+(n-i+1)=\sum_{p=1}^{i-1}(n-p)+1 \quad =(i-1)*(2n-i+2)/2$$

个元素，而 a_{ij} 是它所在的行中要存储的第（j-i+1）个；所以，它是上三角存储顺序中的第 (i-1)*(2n-i+2)/2+(j-i+1)个，因此它在 SA 中的下标为：k=(i-1)*(2n-i+2)/2+j-i。

综上， sa_k 与 a_{ji} 的对应关系为：

$$k=\begin{cases} (i-1)*(2n-i+2)/2+j-i & \text{当 } i \leqslant j \\ n*(n+1)/2 & \text{当 } i>j \end{cases}$$

5.3 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。

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（i,j,v），然后再按某种规律存储这些三元组。

5.3.1 稀疏矩阵的三元组表存储

将三元组按行优先的顺序，同一行中列号从小到大的规律排列成一个线性表，称为三元组表，采用顺序存储方法存储该表。显然，要唯一的表示一个稀疏矩阵，还需要存储三元组表的同时存储该矩阵的行、列，为了运算方便，矩阵的非零元素的个数也同时存储。

```
define SMAX  1024  /*一个足够大的数*/
typedef  struct
  {  int i,j;   /*非零元素的行、列*/
     datatype  v;   /*非零元素值*/
  }SPNode;   /*三元组类型*/
typedef  struct
 {  int mu,nu,tu;    /*矩阵的行、列及非零元素的个数*/
    SPNode  data[SMAX];  /*三元组表*/
 } SPMatrix; /*三元组表的存储类型*/
```

1. 稀疏矩阵的转置

设 SPMatrix A; 表示一 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 SPMatrix B; 由 A 求 B 需要：

A 的行、列转化成 B 的列、行；

将 A.data 中每一三元组的行列交换后转化到 B.data 中；

看上去以上两点完成之后，似乎完成了 B，没有。因为我们前面规定三元组的是按一行一行且每行中的元素是按列号从小到大的规律顺序存放的，因此 B 也必须按此规律实现，A 的转置 B 如图 5.13 所示，图 5.14 是它对应的三元组存储，就是说，在 A 的三元组存储基础上得到 B 的三元组表存储（为了运算方便，矩阵的行列都从 1 算起，三元组表 data 也从 1 单元用起）。

算法思路：

①A 的行、列转化成 B 的列、行；

②在 A.data 中依次找第一列的、第二列的、直到最后一列，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。

算法如下：

```
void TransM1 (SPMatrix *A)
  {  SPMatrix *B;
     int p,q,col;
     B=malloc(sizeof(SPMatrix));  /*申请存储空间*/
     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*/
      {  q=0;
        for (col=1; col<=(A->nu); col++)  /*按 A 的列序转换*/
          for (p=1; p<= (A->tu); p++)  /*扫描整个三元组表*/
             if  (A->data[p].j==col )
               {  B->data[q].i= A->data[p].j ;
                  B->data[q].j= A->data[p].i ;
                     B->data[q].v= A->data[p].v;
                     q++;     }/*if*/
        } /*if(B->tu>0)*/
     return B;    /*返回的是转置矩阵的指针*/
   }   /*TransM1*/
```

算法 5.1 稀疏矩阵转置

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 $O(m*n^2)$，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。

算法 5.1 的效率低的原因是算法要从 A 的三元组表中寻找第一列、第二列、…，要反复搜索 A 表，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，如此类推，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。

根据这个想法，需引入两个向量来实现 ：num[n+1]和 cpot[n+1]，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），cpot [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 cpot 的初始值为：

cpot[1]=1；

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

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 cpot[col]位置上，cpot[col]加 1 ，cpot[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如下：

```
SPMatrix * TransM2 (SPMatrix *A)
{  SPMatrix *B;
   int  i,j,k;
   int num[n+1],cpot[n+1];
   B=malloc(sizeof(SPMatrix));  /*申请存储空间*/
   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*/
    {  for (i=1;i<=A->nu;i++)  num[i]=0;
       for (i=1;i<=A->tu;i++)     /*求矩阵 A 中每一列非零元素的个数*/
         { j= A->data[i].j;
           num[j]++;
         }
        cpot[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 (i=2;i<=A->nu;i++)
            cpot[i]= cpot[i-1]+num[i-1];
       for (i=1; i<= (A->tu); i++)     /*扫描三元组表*/
          { j=A->data[i].j;  /*当前三元组的列号*/
            k=cpot[j];     /*当前三元组在 B.data 中的位置*/
            B->data[k].i= A->data[i].j ;
            B->data[k].j= A->data[i].i ;
             B->data[k].v= A->data[i].v;
              cpot[j]++;
            } /*for i */
    }  /*if  (B->tu>0)*/
   return B;     /*返回的是转置矩阵的指针*/
}   /*TransM2*/
```

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向量。

2．稀疏矩阵的乘积

已知稀疏矩阵 A(m_1× n_1)和 B(m_2× n_2)，求乘积 C(m_1× n_2)。

由矩阵乘法规则知：

C（i，j）=A(i,1)×B(1,j)+A(i,2)×B(2,j)+…+A(i,n)×B(n,j)

$$=\sum_{k=1}^{n} A(i,k)*B(k,j)$$

这就是说只有 A(i,k)与 B(k,p)（即 A 元素的列与 B 元素的行相等的两项）才有相乘的机会，且当两项都不为零时，乘积中的这一项才不为零。

矩阵用二维数组表示时，传统的矩阵乘法是：A 的第一行与 B 的第一列对应相乘累加后得到 c_{11}，A 的第一行再与 B 的第二列对应相乘累加后得到 c_{12}，…，因为现在按三元组表存储，三元组表是按行为主序存储的，在 B.data 中，同一行的非零元素其三元组是相邻存放的，同一列的非零元素其三元组并未相邻存放，因此在 B.data 中反复搜索某一列的元素是很费时的，因此改变一下求值的顺序，以求 c_{11}和 c_{12}为例，因为 ：

即 a_{11}只有可能和 B 中第 1 行的非零元素相乘，a_{12}只有可能和 B 中第 2 行的非零元素相乘，…，而同一行的非零元是相邻存放的，所以求 c_{11}和 c_{12}同时进行：求 $a_{11}*b_{11}$累加到 c_{11}，求 $a_{11}*b_{12}$累加到 c_{12}，再求 $a_{12}*b_{21}$累加到 c_{11}，再求 $a_{12}*b_{22}$累加到 c_{22}，…，当然只有 a_{ik}和 b_{kj}(列号与行号相等)且均不为零（三元组存在）时才相乘，并且累加到 c_{ij}当中去。

为了运算方便，设一个累加器：datatype temp[n+1]；用来存放当前行中 c_{ij}的值，当前行中所有元素全部算出之后，再存放到 C.data 中去。

为了便于 B.data 中寻找 B 中的第 k 行第一个非零元素，与前面类似，在此需引入 num 和 rpot 两个向量。num[k]表示矩阵 B 中第 k 行的非零元素的个数；rpot[k]表示第 k 行的第一个非零元素在 B.data 中的位置。于是有

rpot[1]=1

rpot[k]=rpot[k-1]+num[k-1]　　2≤k≤n

根据以上分析，稀疏矩阵的乘法运算的粗略步骤如下：

⑴初始化。清理一些单元，准备按行顺序存放乘积矩阵;

⑵求 B 的 num，rpot;

⑶做矩阵乘法。将 A.data 中三元组的列值与 B.data 中三元组的行值相等的非零元素相乘，并将具有相同下标的乘积元素相加。

算法如下：

```
SPMatrix *MulSMatrix (SPMatrix *A, SPMatrix *B)
            /*稀疏矩阵 A(m1× n1)和 B(m2× n2) 用三元组表存储，求 A×B */
{  SPMatrix  *C; /* 乘积矩阵的指针 */
 int p,q,i,j,k,r;
 datatype temp[n+1];
 int num[B->nu+1],rpot[B->nu+1];
 if (A->nu!=B->mu) return NULL; /*A 的列与 B 的行不相等*/
 C=malloc(sizeof(SPMatrix));  /*申请 C 矩阵的存储空间*/
 C->mu=A->mu; C->nu=B->nu;
 if (A->tu*B->tu==0)  {C->tu=0; return C;  }
  for (i=1;i<= B->mu;i++)  num[i]=0;  /*求矩阵 B 中每一行非零元素的个数*/
    for (k=1;k<=B->tu;k++)
     { i= B->data[k].i;
         num[i]++;
      }
   rpot[1]=1;  /*求矩阵 B 中每一行第一个非零元素在 B.data 中的位置*/
    for (i=2;i<=B->mu;i++)
          rpot[i]= rpot[i-1]+num[i-1];

   r=0;   /*当前 C 中非零元素的个数*/
    p=1;  /*指示 A.data 中当前非零元素的位置*/
   for ( i= 1;i<=A->mu; i++)
       {  for (j=1;j<=B->nu;j++)  temp[j]=0;  /*cij的累加器初始化*/
            while (A->data[p].i==i )  ./*求第 i 行的*/
             {  k=A->data[p].j;  /*A 中当前非零元的列号*/
                if (k<B->mu)  t=rpot[k+1];
                   else  t=B->tu+1; /*确定 B 中第 k 行的非零元素在 B.data 中的下限位置*/
                   for (q=rpot[k]; q<t; q++;)  /* B 中第 k 行的每一个非零元素*/
                 {  j=B->data[q].j;
                    temp[j]+=A->data[p].v * B->data[q].v
                 }
                    p++;
            } /* while */
```

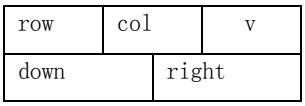
```
            for (j=1;j<=B->nu;j++)
            if (temp[j] )
               { r++;;
               C->data[r]={i,j,temp[j] };
               }
        }  /*for i*/
      C->tu=r;
      return C;
    }/* MulSMatrix */
```

算法 5.3　稀疏矩阵的乘积

分析上述算法的时间性能如下：（1）求 num 的时间复杂度为 O(B->nu+B->tu)；（2）求 rpot 时间复杂度为 O(B->mu)；（3）求 temp 时间复杂度为 O(A->mu*B->nu)；（4）求 C 的所有非零元素的时间复杂度为 O(A->tu*B->tu/B->mu)；（5）压缩存储时间复杂度为 O(A->mu*B->nu)；所以总的时间复杂度为 O(A->mu*B->nu+(A->tu*B->tu)/B->nu)。

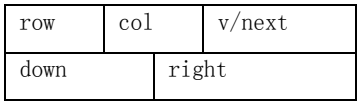
5.3.2　稀疏矩阵的十字链表存储

用十字链表表示稀疏矩阵的基本思想是：对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 5.19 表示，其中：row 域存储非零元素的行号，col 域存储非零元素的列号，v 域存储本元素的值，right，down 是两个指针域。



稀疏矩阵中每一行的非零元素结点按其列号从小到大顺序由 right 域链成一个带表头结点的循环行链表，同样每一列中的非零元素按其行号从小到大顺序由 down 域也链成一个带表头结点的循环列链表。即每个非零元素 a_{ij}既是第 i 行循环链表中的一个结点，又是第 j 列循环链表中的一个结点。行链表、列链表的头结点的 row 域和 col 域置 0。每一列链表的表头结点的 down 域指向该列链表的第一个元素结点，每一行链表的表头结点的 right 域指向该行表的第一个元素结点。由于各行、列链表头结点的 row 域、col 域和 v 域均为零，行链表头结点只用 right 指针域，列链表头结点只用 right 指针域，故这两组表头结点可以合用，也就是说对于第 i 行的链表和第 i 列的链表可以共用同一个头结点。为了方便地找到每一行或每一列，将每行（列）的这些头结点们链接起来，因为头结点的值域空闲，所以用头结点的值域作为连接各头结点的链域，即第 i 行（列）的头结点的值域指向第 i+1 行（列）的头结点，… ，形成一个循环表。这个循环表又有一个头结点，这就是最后的总头结点，指针 HA 指向它。总头结点的 row 和 col 域存储原矩阵的行数和列数。

因为非零元素结点的值域是 datatype 类型，在表头结点中需要一个指针类型，为了使整个结构的结点一致，我们规定表头结点和其它结点有同样的结构，因此该域用一个联合来表示；改进后的结点结构如图 5.20 所示。



综上，结点的结构定义如下:

```
 typedef  struct   node
  { int  row, col;
    struct node *down , *right;
    union  v_next
            {  datatype  v;
               struct node  *next;
            }
   }  MNode, *MLink;
```

1. 建立稀疏矩阵 A 的十字链表

首先输入的信息是：m（A 的行数），n（A 的列数），r（非零项的数目），紧跟着输入的是 r 个形如（i,j,a_{ij}）的三元组。

算法的设计思想是：首先建立每行（每列）只有头结点的空链表，并建立起这些头结点拉成的循环链表；然后每输入一个三元组（i，j，a_{ij}），则将其结点按其列号的大小插入到第 i 个行链表中去，同时也按其行号的大小将该结点插入到第 j 个列链表中去。在算法中将利用一个辅助数组 MNode *hd[s+1]; 其中 s=max(m , n) ， hd [i]指向第 i 行(第 i 列)链表的头结点。这样做可以在建立链表时随机的访问任何一行（列），为建表带来方便。

算法如下：

```
MLink CreatMLink( ) /* 返回十字链表的头指针*/
 {
    MLink H;
     Mnode *p,*q,*hd[s+1];
    int i,j,m,n,t;
     datatype v;
     scanf(“%d,%,%d”,&m,&n,&t);
    H=malloc(sizeof(MNode));    /*申请总头结点*/
    H->row=m; H->col=n;
   hd[0]=H;
   for(i=1; i<S; i++)
     {  p=malloc(sizeof(MNode));  /*申请第 i 个头结点*/
          p->row=0; p->col=0;
          p->right=p; p->down=p;
          hd[i]=p;
          hd[i-1]->v_next.next=p;
    }
    hd[S]->v_next.next=H; /*将头结点们形成循环链表*/
    for (k=1;k<=t;k++)
      { scanf (“%d,%d,%d”,&i,&j,&v);  /*输入一个三元组，设值为 int*/
        p=malloc(sizeof(MNode));
        p->row=i ; p->col=j; p->v_next.v=v
         /*以下是将*p 插入到第 i 行链表中去，且按列号有序*/
       q=hd[i];
       while ( q->right!=hd[i]  &&  (q->right->col)<j )  /*按列号找位置*/
            q=q->right;
       p->right=q->right;  /*插入*/
       q->right=p;
          /*以下是将*p 插入到第 j 行链表中去，且按行号有序*/
       q=hd[i];
       while ( q->down!=hd[j]  &&  (q->down->row)<i )   /*按行号找位置*/
            q=q->down;
       p-> down =q-> down; /*插入*/
       q-> down =p;
      } /*for k*/
    return H;
} /* CreatMLink */
```

算法 5.4　建立稀疏矩阵的十字链表

上述算法中，建立头结点循环链表时间复杂度为 O(S)，插入每个结点到相应的行表和列表的时间复杂度是 O(t*S)，这是因为每个结点插入时都要在链表中寻找插入位置，所以总的时间复杂度为 O(t*S)。该算法对三元组的输入顺序没有要求。如果我们输入三元组时是按以行为主序（或列）输入的，则每次将新结点插入到链表的尾部的，改进算法后，时间复杂度为 O(S+t)。

2．两个十字链表表示的稀疏矩阵的加法

已知两个稀疏矩阵 A 和 B，分别采用十字链表存储，计算 C=A+B，C 也采用十字链表方式存储，并且在 A 的基础上形成 C。

由矩阵的加法规则知，只有 A 和 B 行列对应相等，二者才能相加。C 中的非零元素 c_{ij} 只可能有 3 种情况：或者是 $a_{ij}+b_{ij}$，或者是 a_{ij} (b_{ij}=0)，或者是 b_{ij} (a_{ij}=0)，因此当 B 加到 A 上时，对 A 十字链表的当前结点来说，对应下列四种情况：或者改变结点的值（$a_{ij}+b_{ij}\neq 0$），或者不变（$b_{ij}=0$），或者插入一个新结点（$a_{ij}=0$），还可能是删除一个结点（$a_{ij}+b_{ij}=0$）。整个运算从矩阵的第一行起逐行进行。对每一行都从行表的头结点出发，分别找到 A 和 B 在该行中的第一个非零元素结点后开始比较，然后按 4 种不同情况分别处

理。设 pa 和 pb 分别指向 A 和 B 的十字链表中行号相同的两个结点， 4 种情况如下：

(1) 若 pa->col=pb->col 且 pa->v+pb->v≠0，则只要用 $a_{ij}+b_{ij}$ 的值改写 pa 所指结点的值域即可。

(2) 若 pa->col=pb->col 且 pa->v+pb->v=0，则需要在矩阵 A 的十字链表中删除 pa 所指结点，此时需改变该行链表中前趋结点的 right 域，以及该列链表中前趋结点的 down 域。

(3) 若 pa->col < pb->col 且 pa->col≠0（即不是表头结点），则只需要将 pa 指针向右推进一步，并继续进行比较。

(4) 若 pa->col > pb->col 或 pa->col＝0（即是表头结点），则需要在矩阵 A 的十字链表中插入一个 pb 所指结点。

由前面建立十字链表算法知，总表头结点的行列域存放的是矩阵的行和列，而各行（列）链表的头结点其行列域值为零，当然各非零元素结点的行列域其值不会为零，下面分析的 4 种情况利用了这些信息来判断是否为表头结点。

综上所述，算法如下：

```
MLink AddMat (Ha,Hb)
 MLink Ha,Hb;
{  Mnode  *p,*q,*pa,*pb,*ca,*cb,*qa;
   if (Ha->row!=Hb->row || Ha->col!=Hb->col) return NULL;
   ca=Ha->v_next.next; /*ca 初始指向 A 矩阵中第一行表头结点*/
   cb=Hb->v_next.next; /*cb 初始指向 B 矩阵中第一行表头结点*/
   do { pa=ca->right;  /*pa 指向 A 矩阵当前行中第一个结点*/
       qa=ca;  /*qa 是 pa 的前驱*/
        pb=cb->right;  /*pb 指向 B 矩阵当前行中第一个结点*/
       while (pb->col!=0)  /*当前行没有处理完*/
        {
          if  (pa->col < pb->col && pa->col !=0 ) /*第三种情况*/
            { qa=pa;
              pa=pa->right;
            }
          else
            if (pa->col > pb->col || pa->col ==0 ) /*第四种情况*/
              {p=malloc(sizeof(MNode));
               p->row=pb->row; p->col=pb->col; p->v=pb->v;
               p->right=pa;qa->right=p; /* 新结点插入*pa 的前面*/
                  pa=p;
                      /*新结点还要插到列链表的合适位置，先找位置，再插入*/
               q=Find_JH(Ha,p->col); /*从列链表的头结点找起*/
               while(q->down->row!=0 && q->down->row<p->row)
                  q=q->down;
               p->down=q->down; /*插在*q 的后面*/
               q->down=p;
               pb=pb->right;
              } /* if */
            else /*第一、二种情况*/
              {x= pa->v_next.v+ pb->v_next.v;
               if (x==0)  /*第二种情况*/
                { qa->right=pa->right;  ./*从行链中删除*/
                     /*还要从列链中删除，找*pa 的列前驱结点*/
                     q= Find_JH (Ha,pa->col); /*从列链表的头结点找起*/
                  while ( q->down->row < pa->row  )
                     q=q->down;
                  q->down=pa->down;
                  free (pa);
                  pa=qa;
```

```
                } /*if (x==0)*/
               else  /*第一种情况*/
                         { pa->v_next.v=x;
                        qa=pa;
                       }
              pa=pa->right;
              pb=pb->right;
             }
          } /*while*/
        ca=ca->v_next.next;  /*ca 指向 A 中下一行的表头结点*/
        cb=cb->v_next.next;  /*cb 指向 B 中下一行的表头结点*/
      } while (ca->row==0)  /*当还有未处理完的行则继续*/
    return  Ha;
}
```

为了保持算法的层次，在上面的算法，用到了一个函数 findjH。

函数 Mlink Find_JH(MLink H, int j)的功能是：返回十字链表 H 中第 j 列链表的头结点指针。

5.4　　广义表

5.4.1　　广义表的定义和基本运算

1.广义表的定义和性质

广义表（Generalized Lists）是 n（n≥0）个数据元素 a_1，a_2，…，a_i，…，a_n的有序序列，一般记作：

$$ls=(a_1, a_2, \cdots, a_i, \cdots, a_n)$$

其中：ls 是广义表的名称，n 是它的长度。每个 a_i（1≤i≤n）是 ls 的成员，它可以是单个元素，也可以是一个广义表，分别称为广义表 ls 的单元素和子表。当广义表 ls 非空时，称第一个元素 a_1为 ls 的表头（head），称其余元素组成的表（a_2，…，a_i，…，a_n）为 ls 的表尾（tail）。

显然，广义表的定义是递归的。

为书写清楚起见，通常用大写字母表示广义表，用小写字母表示单个数据元素，广义表用括号括起来，括号内的数据元素用逗号分隔开。

2.广义表的性质

⑴广义表是一种多层次的数据结构。广义表的元素可以是单元素，也可以是子表，而子表的元素还可以是子表，…。

⑵广义表可以是递归的表。广义表的定义并没有限制元素的递归，即广义表也可以是其自身的子表。例如表 E 就是一个递归的表。

⑶广义表可以为其他表所共享。

当二维数组的每行（或每列）作为子表处理时，二维数组即为一个广义表。

另外，树和有向图也可以用广义表来表示。

由于广义表不仅集中了线性表、数组、树和有向图等常见数据结构的特点，而且可有效地利用存储空间，因此在计算机的许多应用领域都有成功使用广义表的实例。

3.广义表基本运算

广义表有两个重要的基本操作，即取头操作（Head）和取尾操作（Tail）。

根据广义表的表头、表尾的定义可知，对于任意一个非空的列表，其表头可能是单元素也可能是列表，而表尾必为列表。

此外，在广义表上可以定义与线性表类似的一些操作，如建立、插入、删除、拆开、连接、复制、遍历等。

CreateLists(ls)：根据广义表的书写形式创建一个广义表 ls。

IsEmpty(ls)：若广义表 ls 空，则返回 True；否则返回 False。

Length(ls)：求广义表 ls 的长度。

Depth(ls)：求广义表 ls 的深度。

Locate(ls，x)：在广义表 ls 中查找数据元素 x。

Merge(ls1，ls2)：以 ls1 为头、ls2 为尾建立广义表。

CopyGList(ls1，ls2)：复制广义表，即按 ls1 建立广义表 ls2。

Head(ls)：返回广义表 ls 的头部。

Tail(ls)：返回广义表的尾部。

……

5.4.2 广义表的存储

通常都采用链式的存储结构来存储广义表。在这种表示方式下，每个数据元素可用一个结点表示。

按结点形式的不同，广义表的链式存储结构又可以分为不同的两种存储方式。一种称为头尾表示法，另一种称为孩子兄弟表示法。

1.头尾表示法

若广义表不空，则可分解成表头和表尾；反之，一对确定的表头和表尾可惟一地确定一个广义表。

由于广义表中的数据元素既可能是列表也可能是单元素，相应地在头尾表示法中结点的结构形式有两种：一种是表结点，用以表示列表；另一种是元素结点，用以表示单元素。在表结点中应该包括一个指向表头的指针和指向表尾的指针；而在元素结点中应该包括所表示单元素的元素值。为了区分这两类结点，在结点中还要设置一个标志域，如果标志为 1，则表示该结点为表结点；如果标志为 0，则表示该结点为元素结点。其形式定义说明如下：

```
typedef  enum {ATOM, LIST} Elemtag;     /*ATOM=0：单元素；LIST=1：子表*/
typedef  struct  GLNode {
    Elemtag  tag;                       /*标志域，用于区分元素结点和表结点*/
    union {                             /*元素结点和表结点的联合部分*/
      datatype  data;                    /*data 是元素结点的值域*/
      struct {
         struct GLNode  *hp, *tp
       }ptr;                            /*ptr 是表结点的指针域，ptr.hp 和 ptr.tp 分别*/
                                        /*指向表头和表尾*/
    };
  }*GList;                              /*广义表类型*/
```

头尾表示法的结点形式如图 5.21 所示。

（a）表结点　　（b）元素结点

图 5.21　头尾表示法的结点形式

从上述存储结构示例中可以看出，采用头尾表示法容易分清列表中单元素或子表所在的层次。

2.孩子兄弟表示法

在孩子兄弟表示法中，也有两种结点形式：一种是有孩子结点，用以表示列表；另一种是无孩子结点，用以表示单元素。在有孩子结点中包括一个指向第一个孩子（长子）的指针和一个指向兄弟的指针；而在无孩子结点中包括一个指向兄弟的指针和该元素的元素值。为了能区分这两类结点，在结点中还要设置一个标志域。如果标志为 1，则表示该结点为有孩子结点；如果标志为 0，则表示该结点为无孩子结点。其形式定义说明如下：

```
typedef  enum {ATOM, LIST} Elemtag;     /*ATOM=0：单元素；LIST=1：子表*/
typedef  struct  GLENode {
    Elemtag  tag;                       /*标志域，用于区分元素结点和表结点*/
    union {                             /*元素结点和表结点的联合部分*/
      datatype  data;                    /*元素结点的值域*/
      struct GLENode  *hp;              /*表结点的表头指针*/
      };
    struct GLENode  *tp;                /*指向下一个结点*/
  }*EGList;                             /*广义表类型*/
```

孩子兄弟表示法的结点形式如图 5.23 所示。

（a）有孩子结点　　（b）无孩子结点

图 5.23　孩子兄弟表示法的结点形式

从图 5.24 的存储结构示例中可以看出，采用孩子兄弟表示法时，表达式中的左括号“（”对应存储表示中的 tag=1 的结点，且最高层结点的 tp 域必为 NULL。

5.5.3 广义表基本操作的实现

由于广义表的定义是递归的，因此相应的算法一般也都是递归的。

1.广义表的取头、取尾

```
GList Head(GList ls)
  {
   if ls->tag = = 1
    then  p = ls->hp;
   return  p;
  }
```

算法 5.6

```
GList Tail(GList ls)
  {
   if ls->tag = = 1
    then  p = ls->tp;
   return  p;
  }
```

算法 5.7

2.建立广义表的存储结构

```
int  Create(GList *ls, char * S)
  { Glist  p;  char  *sub;
   if StrEmpty(S)  *ls = NULL;
   else {
    if (!(*ls = (GList)malloc(sizeof(GLNode))))  return  0;
    if (StrLength(S) = = 1) {
      (*ls)->tag = 0;
      (*ls)->data = S;
     }
    else {
      (*ls)->tag = 1;
      p = *ls;
      hsub =SubStr(S,2,StrLength(S)-2);
      do {
        sever(sub,hsub);
        Create(&(p->ptr.hp), sub);
        q = p;
        if (!StrEmpty(sub)){
          if (!(p = (GList)malloc(sizeof(GLNode))))  return 0;;
          p->tag = 1;
          q->ptr.tp = p;
         }
       }while (!StrEmpty(sub));
       q->ptr.tp = NULL;
      }
     }
   return 1;
  }
```

算法 5.8

```
int  sever(char *str, char *hstr)
```

```
  {
   int  n = StrLength(str);
    i= 1; k = 0;
   for (i = 1, k = 0; i <= n || k != 0; ++i)
     {
      ch=SubStr(str,i,1);
      if (ch = = '(')  ++k;
      else  if (ch = = ')')  --k;
   }
   if (i <= n)
    {
     hstr =SubStr(str,1,i-2);
     str= SubStr(str,i,n-i+1);
    }
   else {
     StrCopy(hstr,str);
     ClearStr(str);
    }
  }                                  算法 5.9
```

3.以表头、表尾建立广义表

```
int  Merge(GList ls1,GList ls2, Glist *ls)
  {
   if (!(*ls = (GList)malloc(sizeof(GLNode))))  return 0;
   *ls->tag = 1;
   *ls->hp = ls1;
   *ls->tp = ls2;
   return 1;
  }
```

算法 5.10

4.求广义表的深度

```
int Depth(GList ls)
  {
   if (!ls)
     return  1;                /*空表深度为 1*/
   if (ls->tag = = 0)
     return  0;                /*单元素深度为 0*/
   for (max = 0,p = ls; p; p = p->ptr.tp) {
     dep = Depth(p->ptr.hp);     /*求以 p->ptr.hp 尾头指针的子表深度*/
     if (dep > max)  max = dep;
    }
   return max+1;             /*非空表的深度是各元素的深度的最大值加 1*/
  }
```

算法 5.11

5.复制广义表

```
int  CopyGList(GList ls1, GList *ls2)
  {
   if (!ls1)  *ls2 = NULL;         /*复制空表*/
   else {
     if (!(*ls2 = (Glist)malloc(sizeof(Glnode)))) return 0;     /*建表结点*/
```

```
    (*ls2)->tag = ls1->tag;
    if (ls1->tag = = 0) (*ls2)->data = ls1->data;     /*复制单元素*/
    else {
      CopyGList(&((*ls2)->ptr.hp), ls1->ptr.hp);     /*复制广义表 ls1->ptr.hp 的一个副本*/
      CopyGList(&((*ls2)->ptr.tp) , ls1->ptr.tp);      /*复制广义表 ls1->ptr.tp 的一个副本*/
    }
  }
  return 1;
}
```

算法 5.12

第六章 二叉树

6.1.1 二叉树的基本概念

1. 二叉树

二叉树（Binary Tree）是个有限元素的集合，该集合或者为空、或者由一个称为根(root)的元素及两个不相交的、被分别称为左子树和右子树的二叉树组成。当集合为空时，称该二叉树为空二叉树。在二叉树中，一个元素也称作一个结点。

二叉树是有序的，即若将其左、右子树颠倒，就成为另一棵不同的二叉树。即使树中结点只有一棵子树，也要区分它是左子树还是右子树。因此二叉树具有五种基本形态。

2. 二叉树的相关概念

（1）结点的度。结点所拥有的子树的个数称为该结点的度。

（2）叶结点。度为 0 的结点称为叶结点，或者称为终端结点。

（3）分枝结点。度不为 0 的结点称为分支结点，或者称为非终端结点。一棵树的结点除叶结点外，其余的都是分支结点。

（4）左孩子、右孩子、双亲。树中一个结点的子树的根结点称为这个结点的孩子。这个结点称为它孩子结点的双亲。具有同一个双亲的孩子结点互称为兄弟。

（5）路径、路径长度。如果一棵树的一串结点 $n_1,n_2,\cdots,n_k$有如下关系：结点 n_i是 n_{i+1}的父结点（$1\leqslant i<k$），就把 $n_1,n_2,\cdots,n_k$称为一条由 n_1至 n_k的路径。这条路径的长度是 k-1。

（6）祖先、子孙。在树中，如果有一条路径从结点 M 到结点 N，那么 M 就称为 N 的祖先，而 N 称为 M 的子孙。

（7）结点的层数。规定树的根结点的层数为 1，其余结点的层数等于它的双亲结点的层数加 1。

（8）树的深度。树中所有结点的最大层数称为树的深度。

（9）树的度。树中各结点度的最大值称为该树的度。

（10）满二叉树。在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子结点都在同一层上，这样的一棵二叉树称作满二叉树。

（11)完全二叉树。一棵深度为 k 的有 n 个结点的二叉树,对树中的结点按从上至下、从左到右的顺序进行编号,如果编号为 i(1≤i≤n)的结点与满二叉树中编号为 i 的结点在二叉树中的位置相同，则这棵二叉树称为完全二叉树。完全二叉树的特点是：叶子结点只能出现在最下层和次下层，且最下层的叶子结点集中在树的左部。显然，一棵满二叉树必定是一棵完全二叉树，而完全二叉树未必是满二叉树。

6.1.2 二叉树的主要性质

性质 1　一棵非空二叉树的第 i 层上最多有 2^{i-1}个结点（$i\geqslant 1$）。

性质 2　一棵深度为 k 的二叉树中，最多具有 2^k-1 个结点。

性质 3　对于一棵非空的二叉树，如果叶子结点数为 n_0，度数为 2 的结点数为 n_2，则有:

$n_0=n_2+1$。

性质 4　具有 n 个结点的完全二叉树的深度 k 为$[\log_2 n]+1$。

性质 5　对于具有 n 个结点的完全二叉树，如果按照从上至下和从左到右的顺序对二叉树中的所有结点从 1 开始顺序编号，则对于任意的序号为 i 的结点，有：

（1）如果 i>1，则序号为 i 的结点的双亲结点的序号为 i/2(“/”表示整除)；如果 i＝1，则序号为 i 的结点是根结点，无双亲结点。

（2）如果 2i≤n，则序号为 i 的结点的左孩子结点的序号为 2i；如果 2i>n，则序号为 i 的结点无左孩子。

（3）如果 2i＋1≤n，则序号为 i 的结点的右孩子结点的序号为 2i＋1；如果 2i＋1>n，则序号为 i 的结点无右孩子。

此外，若对二叉树的根结点从 0 开始编号，则相应的 i 号结点的双亲结点的编号为（i－1）/2，左孩子的编号为 2i＋1，右孩子的编号为 2i＋2。

6.2 基本操作与存储实现

6.2.1 二叉树的存储

1．顺序存储结构

所谓二叉树的顺序存储，就是用一组连续的存储单元存放二叉树中的结点。一般是按照二叉树结点从上至下、从左到右的顺序存储。这样结点在存储位置上的前驱后继关系并不一定就是它们在逻辑上的邻接关系，然而只有通过一些方法确定某结点在逻辑上的前驱结点和后继结点，这种存储才有意义。因此，依据二叉树的性质，完全二叉树和满二叉树采用顺序存储比较合适，树中结点的序号可以唯一地反映出结点之间的逻辑关系，这样既能够最大可能地节省存储空间，又可以利用数组元素的下标值确定结点在二叉树中的位置，以及结点之间的关系。

对于一般的二叉树，如果仍按从上至下和从左到右的顺序将树中的结点顺序存储在一维数组中，则数组元素下标之间的关系不能够反映二叉树中结点之间的逻辑关系，只有增添一些并不存在的空结点，使之成为一棵完全二叉树的形式，然后再用一维数组顺序存储。如图 6.5 给出了一棵一般二叉树改造后的完全二叉树形态和其顺序存储状态示意图。显然，这种存储对于需增加许多空结点才能将一棵二叉树改造成为一棵完全二叉树的存储时，会造成空间的大量浪费，不宜用顺序存储结构。最坏的情况是右单支树。

二叉树的顺序存储表示可描述为：

```
#define MAXNODE                          /*二叉树的最大结点数*/
typedef elemtype SqBiTree[MAXNODE]       /*0 号单元存放根结点*/
SqBiTree bt;
```

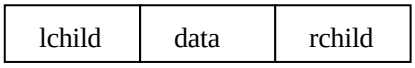
即将 bt 定义为含有 MAXNODE 个 elemtype 类型元素的一维数组。

2．链式存储结构

所谓二叉树的链式存储结构是指，用链表来表示一棵二叉树，即用链来指示着元素的逻辑关系。

（1）二叉链表存储

链表中每个结点由三个域组成，除了数据域外，还有两个指针域，分别用来给出该结点左孩子和右孩子所在的链结点的存储地址。结点的存储的结构为：



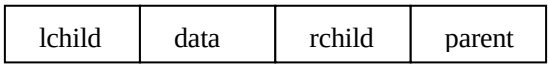
其中，data 域存放某结点的数据信息；lchild 与 rchild 分别存放指向左孩子和右孩子的指针，当左孩子或右孩子不存在时，相应指针域值为空（用符号∧或 NULL 表示）。

图 6.7(a)给出了图 6.3(b)所示的一棵二叉树的二叉链表示。

二叉链表也可以带头结点的方式存放，如图 6.7(b)所示。

（2）三叉链表存储

每个结点由四个域组成，具体结构为：



其中，data、lchild 以及 rchild 三个域的意义同二叉链表结构；parent 域为指向该结点双亲结点的指针。这种存储结构既便于查找孩子结点，又便于查找双亲结点；但是，相对于二叉链表存储结构而言，它增加了空间开销。

尽管在二叉链表中无法由结点直接找到其双亲，但由于二叉链表结构灵活，操作方便，对于一般情况的二叉树，甚至比顺序存储结构还节省空间。因此，二叉链表是最常用的二叉树存储方式。本书后面所涉及到的二叉树的链式存储结构不加特别说明的都是指二叉链表结构。

二叉树的二叉链表存储表示可描述为：

```
typedef struct BiTNode{
      elemtype data;
      struct BiTNode *lchild;*rchild;     /*左右孩子指针*/
     }BiTNode,*BiTree;
```

即将 BiTree 定义为指向二叉链表结点结构的指针类型。

6.2.2 二叉树的基本操作及实现

1．二叉树的基本操作

二叉树的基本操作通常有以下几种：

（1） Initiate（bt）建立一棵空二叉树。

（2）Create（x，lbt，rbt）生成一棵以 x 为根结点的数据域信息，以二叉树 lbt 和 rbt 为左子树和右子树的二叉树。

（3）InsertL（bt，x，parent）将数据域信息为 x 的结点插入到二叉树 bt 中作为结点 parent 的左孩子结点。如果结点 parent 原来有左孩子结点，则将结点 parent 原来的左孩子结点作为结点 x 的左孩子结点。

（4）InsertR（bt，x，parent）将数据域信息为 x 的结点插入到二叉树 bt 中作为结点 parent 的右孩子结点。如果结点 parent 原来有右孩子结点，则将结点 parent 原来的右孩子结点作为结点 x 的右孩子结点。

（5）DeleteL（bt，parent）在二叉树 bt 中删除结点 parent 的左子树。

（6）DeleteR（bt，parent）在二叉树 bt 中删除结点 parent 的右子树。

（7）Search（bt，x）在二叉树 bt 中查找数据元素 x。

（8）Traverse（bt）按某种方式遍历二叉树 bt 的全部结点。

2．算法的实现

下面讨论基于二叉链表存储结构的上述操作的实现算法。

（1）Initiate（bt）初始建立二叉树 bt，并使 bt 指向头结点。在二叉树根结点前建立头结点，就如同在单链表前建立的头结点，可以方便后边的一些操作实现。

```
int Initiate (BiTree  *bt)
{/*初始化建立二叉树*bt 的头结点*/
  if((*bt=(BiTNode *)malloc(sizeof(BiTNode)))==NULL)
    return 0;
  *bt->lchild=NULL;
  *bt->rchild=NULL;
  return 1;
 }
```

（2）Create（x，lbt，rbt）建立一棵以 x 为根结点的数据域信息，以二叉树 lbt 和 rbt 为左右子树的二叉树。建立成功时返回所建二叉树结点的指针；建立失败时返回空指针。

```
BiTree Create（elemtype x，BiTree lbt，BiTree rbt）
{/*生成一棵以 x 为根结点的数据域值以 lbt 和 rbt 为左右子树的二叉树*/
  BiTree  p;
  if ((p=(BiTNode *)malloc(sizeof(BiTNode)))==NULL) return NULL;
  p->data=x;
  p->lchild=lbt;
  p->rchild=rbt;
  return p;
 }
```

（3）InsertL（bt，x，parent）

```
BiTree InsertL（BiTree bt，elemtype x，BiTree parent）
 {/*在二叉树 bt 的结点 parent 的左子树插入结点数据元素 x*/
  BiTree  p;
  if (parent==NULL)
   { printf(“\n 插入出错”);
     return NULL;
   }
  if ((p=(BiTNode *)malloc(sizeof(BiTNode)))==NULL) return NULL;
  p->data=x;
  p->lchild=NULL;
  p->rchild=NULL;
  if (parent->lchild==NULL) parent->lchild=p;
  else {p->lchild=parent->lchild;
        parent->lchild=p;
       }
```

```
  return bt;
}
```

（4）InsertR（bt，x，parent）功能类同于（3），算法略。

（5）DeleteL（bt，parent）在二叉树 bt 中删除结点 parent 的左子树。当 parent 或 parent 的左孩子结点为空时删除失败。删除成功时返回根结点指针；删除失败时返回空指针。

```
BiTree  DeleteL（BiTree bt，BiTree parent）
{/*在二叉树 bt 中删除结点 parent 的左子树*/
  BiTree  p;
  if (parent==NULL||parent->lchild==NULL)
   { printf(“\n 删除出错”);
     return NULL’
   }
  p=parent->lchild;
  parent->lchild=NULL;
  free(p);  /*当 p 为非叶子结点时，这样删除仅释放了所删子树根结点的空间，*/
          /*若要删除子树分支中的结点，需用后面介绍的遍历操作来实现。*/
  return br;
}
```

（6）DeleteR（bt，parent）功能类同于（5），只是删除结点 parent 的右子树。算法略。

操作 Search（bt，x）实际是遍历操作 Traverse（bt）的特例，关于二叉树的遍历操作的实现，将在下一节中重点介绍。

6.3　二叉树的遍历

6.3.1　二叉树的遍历方法及递归实现

二叉树的遍历是指按照某种顺序访问二叉树中的每个结点，使每个结点被访问一次且仅被访问一次。

由二叉树的定义可知，一棵由根结点、根结点的左子树和根结点的右子树三部分组成。因此，只要依次遍历这三部分，就可以遍历整个二叉树。若以 D、L、R 分别表示访问根结点、遍历根结点的左子树、遍历根结点的右子树，则二叉树的遍历方式有六种：DLR、LDR、LRD、DRL、RDL 和 RLD。如果限定先左后右，则只有前三种方式，即 DLR（称为先序遍历）、LDR（称为中序遍历）和 LRD（称为后序遍历）。

1．先序遍历（DLR）

先序遍历的递归过程为：若二叉树为空，遍历结束。否则，

（1） 访问根结点；

（2） 先序遍历根结点的左子树；

（3） 先序遍历根结点的右子树。

先序遍历二叉树的递归算法如下：

```
void PreOrder（BiTree bt）
{/*先序遍历二叉树 bt*/
  if (bt==NULL) return;      /*递归调用的结束条件*/
  Visite（bt->data）;        /*访问结点的数据域*/
  PreOrder（bt->lchild）;    /*先序递归遍历 bt 的左子树*/
  PreOrder（bt->rchild）;    /*先序递归遍历 bt 的右子树*/
}
```

2．中序遍历（LDR）

中序遍历的递归过程为：若二叉树为空，遍历结束。否则，

（1）中序遍历根结点的左子树；

（2）访问根结点；

（3）中序遍历根结点的右子树。

中序遍历二叉树的递归算法如下：

```
void InOrder（BiTree bt）
{/*中序遍历二叉树 bt*/
  if (bt==NULL) return;      /*递归调用的结束条件*/
```

```
  InOrder (bt->lchild) ;    /*中序递归遍历 bt 的左子树*/
  Visite (bt->data) ;       /*访问结点的数据域*/
  InOrder (bt->rchild) ;    /*中序递归遍历 bt 的右子树*/
}
```

3. 后序遍历（LRD）

后序遍历的递归过程为：若二叉树为空，遍历结束。否则，

（1）后序遍历根结点的左子树；

（2）后序遍历根结点的右子树。

（3）访问根结点；

后序遍历二叉树的递归算法如下：

```
void PostOrder (BiTree bt)
{/*后序遍历二叉树 bt*/
  if (bt==NULL) return;     /*递归调用的结束条件*/
  PostOrder (bt->lchild) ;   /*后序递归遍历 bt 的左子树*/
  PostOrder (bt->rchild) ;   /*后序递归遍历 bt 的右子树*/
  Visite (bt->data) ;       /*访问结点的数据域*/
}
```

4. 层次遍历

所谓二叉树的层次遍历，是指从二叉树的第一层（根结点）开始，从上至下逐层遍历，在同一层中，则按从左到右的顺序对结点逐个访问。

下面讨论层次遍历的算法。

由层次遍历的定义可以推知，在进行层次遍历时，对一层结点访问完后，再按照它们的访问次序对各个结点的左孩子和右孩子顺序访问，这样一层一层进行，先遇到的结点先访问，这与队列的操作原则比较吻合。因此，在进行层次遍历时，可设置一个队列结构，遍历从二叉树的根结点开始，首先将根结点指针入队列，然后从对头取出一个元素，每取一个元素，执行下面两个操作：

（1） 访问该元素所指结点；

（2） 若该元素所指结点的左、右孩子结点非空，则将该元素所指结点的左孩子指针和右孩子指针顺序入队。

此过程不断进行，当队列为空时，二叉树的层次遍历结束。

在下面的层次遍历算法中，二叉树以二叉链表存放，一维数组 Queue[MAXNODE]用以实现队列，变量 front 和 rear 分别表示当前对首元素和队尾元素在数组中的位置。

```
void LevelOrder (BiTree bt)
/*层次遍历二叉树 bt*/
{ BiTree Queue[MAXNODE];
  int front,rear;
  if (bt==NULL) return;
  front=-1;
  rear=0;
  queue[rear]=bt;
  while(front!=rear)
     {front++;
      Visite(queue[front]->data);    /*访问队首结点的数据域*/
      if (queue[front]->lchild!=NULL)   /*将队首结点的左孩子结点入队列*/
       { rear++;
         queue[rear]=queue[front]->lchild;
        }
      if (queue[front]->rchild!=NULL)   /*将队首结点的右孩子结点入队列*/
       { rear++;
         queue[rear]=queue[front]->rchild;
        }
     }
```

}

6.3.2 二叉树遍历的非递归实现

如图 6.3(b)所示的二叉树，对其进行先序、中序和后序遍历都是从根结点 A 开始的，且在遍历过程中经过结点的路线是一样的，只是访问的时机不同而已。图 6.9 中所示的从根结点左外侧开始，由根结点右外侧结束的曲线，为遍历图 6.3(b)的路线。沿着该路线按△标记的结点读得的序列为先序序列，按*标记读得的序列为中序序列，按⊕标记读得的序列为后序序列。

然而，这一路线正是从根结点开始沿左子树深入下去，当深入到最左端，无法再深入下去时，则返回，再逐一进入刚才深入时遇到结点的右子树，再进行如此的深入和返回，直到最后从根结点的右子树返回到根结点为止。先序遍历是在深入时遇到结点就访问，中序遍历是在从左子树返回时遇到结点访问，后序遍历是在从右子树返回时遇到结点访问。

在这一过程中，返回结点的顺序与深入结点的顺序相反，即后深入先返回，正好符合栈结构后进先出的特点。因此，可以用栈来帮助实现这一遍历路线。其过程如下。

在沿左子树深入时，深入一个结点入栈一个结点，若为先序遍历，则在入栈之前访问之；当沿左分支深入不下去时，则返回，即从堆栈中弹出前面压入的结点，若为中序遍历，则此时访问该结点，然后从该结点的右子树继续深入；若为后序遍历，则将此结点再次入栈，然后从该结点的右子树继续深入，与前面类同，仍为深入一个结点入栈一个结点，深入不下去再返回，直到第二次从栈里弹出该结点，才访问之。

（1）先序遍历的非递归实现

在下面算法中，二叉树以二叉链表存放，一维数组 stack[MAXNODE]用以实现栈，变量 top 用来表示当前栈顶的位置。

```
void NRPreOrder (BiTree bt)
{/*非递归先序遍历二叉树*/
  BiTree stack[MAXNODE],p;
  int top;
  if (bt==NULL) return;
  top=0;
  p=bt;
  while(!(p==NULL&&top==0))
     { while(p!=NULL)
          { Visite(p->data);    /*访问结点的数据域*/
            if (top<MAXNODE-1)  /*将当前指针 p 压栈*/
             { stack[top]=p;
               top++;
             }
            else { printf(“栈溢出”);
                   return;
                 }
            p=p->lchild;          /*指针指向 p 的左孩子*/
           }
          if (top<=0) return;     /*栈空时结束*/
          else{ top--;
                p=stack[top];            /*从栈中弹出栈顶元素*/
                p=p->rchild;             /*指针指向 p 的右孩子结点*/
              }
      }
}
```

（2）中序遍历的非递归实现

中序遍历的非递归算法的实现，只需将先序遍历的非递归算法中的 Visite(p->data)移到 p=stack[top]和 p=p->rchild 之间即可。

（3）后序遍历的非递归实现

由前面的讨论可知，后序遍历与先序遍历和中序遍历不同，在后序遍历过程中，结点在第一次出栈后，还需再次入栈，也就是说，结点要入两次栈，出两次栈，而访问结点是在第二次出栈时访问。因此，为了区别同一个结点指针的两次出栈，设置一标志 flag，令：

$$\text{flag}=\begin{cases}1 & \text{第一次出栈，结点不能访问}\\ 2 & \text{第二次出栈，结点可以访问}\end{cases}$$

当结点指针进、出栈时，其标志 flag 也同时进、出栈。因此，可将栈中元素的数据类型定义为指针和标志 flag 合并的结构体类型。定义如下：

```
typedef struct {
     BiTree  link;
     int  flag;
    }stacktype;
```

后序遍历二叉树的非递归算法如下。在算法中，一维数组 stack[MAXNODE]用于实现栈的结构，指针变量 p 指向当前要处理的结点，整型变量 top 用来表示当前栈顶的位置，整型变量 sign 为结点 p 的标志量。

```
void NRPostOrder(BiTree  bt)
/*非递归后序遍历二叉树 bt*/
{ stacktype stack[MAXNODE];
  BiTree p;
  int top,sign;
  if (bt==NULL) return;
  top=-1              /*栈顶位置初始化*/
  p=bt;
  while (!(p==NULL && top==-1))
    { if (p!=NULL)           /*结点第一次进栈*/
       { top++;
         stack[top].link=p;
         stack[top].flag=1;
         p=p->lchild;          /*找该结点的左孩子*/
       }
       else { p=stack[top].link;
            sign=stack[top].flag;
            top--;
            if (sign==1)              /*结点第二次进栈*/
             {top++;
              stack[top].link=p;
              stack[top].flag=2;         /*标记第二次出栈*/
              p=p->rchild;
             }
            else { Visite(p->data);       /*访问该结点数据域值*/
                 p=NULL;
                }
          }
    }
}
```

6.3.3 由遍历序列恢复二叉树

任意一棵二叉树结点的先序序列和中序序列都是唯一的。反过来，若已知结点的先序序列和中序序列，能否确定这棵二叉树呢？这样确定的二叉树是否是唯一的呢？回答是肯定的。

根据定义，二叉树的先序遍历是先访问根结点，其次再按先序遍历方式遍历根结点的左子树，最后按先序遍历方式遍历根结点的右子树。这就是说，在先序序列中，第一个结点一定是二叉树的根结点。另一方面，中序遍历是先遍历左子树，然后访问根结点，最后再遍历右子树。这样，根结点在中序序列中必然将中序序列分割成两个子序列，前一个子序列是根结点的左子树的中序序列，而后一个子序列是根结点的右子树的中序序列。根据这两个子序列，在先序序列中找到对应的左子序列和右子序列。在先序序列中，左子序列的第一个结点是左子树的根结点，右子序列的第一个结点是右子树的根结点。这样，就确定了二叉树的三个结点。同时，

左子树和右子树的根结点又可以分别把左子序列和右子序列划分成两个子序列，如此递归下去，当取尽先序序列中的结点时，便可以得到一棵二叉树。

同样的道理，由二叉树的后序序列和中序序列也可唯一地确定一棵二叉树。因为，依据后序遍历和中序遍历的定义，后序序列的最后一个结点，就如同先序序列的第一个结点一样，可将中序序列分成两个子序列，分别为这个结点的左子树的中序序列和右子树的中序序列，再拿出后序序列的倒数第二个结点，并继续分割中序序列，如此递归下去，当倒着取取尽后序序列中的结点时，便可以得到一棵二叉树。

上述过程是一个递归过程，其递归算法的思想是：先根据先序序列的第一个元素建立根结点；然后在中序序列中找到该元素，确定根结点的左、右子树的中序序列；再在先序序列中确定左、右子树的先序序列；最后由左子树的先序序列与中序序列建立左子树，由右子树的先序序列与中序序列建立右子树。

下面给出用 C 语言描述的该算法。假设二叉树的先序序列和中序序列分别存放在一维数组 preod[ ]与 inod[ ]中，并假设二叉树各结点的数据值均不相同。

```
void ReBiTree (char preod[ ],char inod[ ],int n,BiTree root)
/*n 为二叉树的结点个数，root 为二叉树根结点的存储地址*/
{ if (n≤0) root=NULL;
  else PreInOd(preod,inod,1,n,1,n,&root);
}
```

算法 6.11

```
void PreInOd (char preod[ ],char inod[ ],int i,j,k,h,BiTree *t)
{* t=(BiTNode *)malloc(sizeof(BiTNode));
  *t->data=preod[i];
  m=k;
  while (inod[m]!=preod[i])  m++;
  if (m==k) *t->lchild=NULL
  else PreInOd(preod,inod,i+1,i+m-k,k,m-1,&t->lchild);
  if (m==h) *t->rchild=NULL
  else PreInOd(preod,inod,i+m-k+1,j,m+1,h,&t->rchild);
}
```

6.3.4 不用栈的二叉树遍历的非递归方法

还有一类二叉树的遍历算法，就是不用栈也不用递归来实现。常用的不用栈的二叉树遍历的非递归方法有以下三种：

（1）对二叉树采用三叉链表存放，即在二叉树的每个结点中增加一个双亲域 parent，这样，在遍历深入到不能再深入时，可沿着走过的路径回退到任何一棵子树的根结点，并再向另一方向走。由于这一方法的实现是在每个结点的存储上又增加一个双亲域，故其存储开销就会增加。

（2）采用逆转链的方法，即在遍历深入时，每深入一层，就将其再深入的孩子结点的地址取出，并将其双亲结点的地址存入，当深入不下去需返回时，可逐级取出双亲结点的地址，沿原路返回。虽然此种方法是在二叉链表上实现的，没有增加过多的存储空间，但在执行遍历的过程中改变子女指针的值，这既是以时间换取空间，同时当有几个用户同时使用这个算法时将会发生问题。

（3）在线索二叉树上的遍历，即利用具有 n 个结点的二叉树中的叶子结点和一度结点的 n+1 个空指针域，来存放线索，然后在这种具有线索的二叉树上遍历时，就可不需要栈，也不需要递归了。有关线索二叉树的详细内容，将在下一节中讨论。

6.4 线索二叉树

6.4.1 线索二叉树的定义及结构

1. 线索二叉树的定义

按照某种遍历方式对二叉树进行遍历，可以把二叉树中所有结点排列为一个线性序列。在该序列中，除第一个结点外，每个结点有且仅有一个直接前驱结点；除最后一个结点外，每个结点有且仅有一个直接后继结点。但是，二叉树中每个结点在这个序列中的直接前驱结点和直接后继结点是什么，二叉树的存储结构中并没有反映出来，只能在对二叉树遍历的动态过程中得到这些信息。为了保留结点在某种遍历序列中直接前驱和直接后继的位置信息，可以利用二叉树的二叉链表存储结构中的那些空指针域来指示。这些指向直接前驱结点和指向直接后继结点的指针被称为线索（thread），加了线索的二叉树称为线索二叉树。

线索二叉树将为二叉树的遍历提供许多遍历。

2. 线索二叉树的结构

一个具有 n 个结点的二叉树若采用二叉链表存储结构，在 2n 个指针域中只有 n－1 个指针域是用来存储结点孩子的地址，而另

外 n＋1 个指针域存放的都是 NULL。因此，可以利用某结点空的左指针域（lchild）指出该结点在某种遍历序列中的直接前驱结点的存储地址，利用结点空的右指针域（rchild）指出该结点在某种遍历序列中的直接后继结点的存储地址；对于那些非空的指针域，则仍然存放指向该结点左、右孩子的指针。这样，就得到了一棵线索二叉树。

由于序列可由不同的遍历方法得到，因此，线索树有先序线索二叉树、中序线索二叉树和后序线索二叉树三种。把二叉树改造成线索二叉树的过程称为线索化。

对图 6.3 (b)所示的二叉树进行线索化，得到先序线索二叉树、中序线索二叉树和后序线索二叉树分别如图 6.11(a)、(b)、(c)所示。图中实线表示指针，虚线表示线索。

那么，下面的问题是在存储中，如何区别某结点的指针域内存放的是指针还是线索？通常可以采用下面两种方法来实现。

（1） 为每个结点增设两个标志位域 ltag 和 rtag。

（2）不改变结点结构，仅在作为线索的地址前加一个负号，即负的地址表示线索，正的地址表示指针。

6.4.2 线索二叉树的基本操作实现

在线索二叉树中，结点的结构可以定义为如下形式：

```
typedef char elemtype;
typedef struct BiThrNode {
    elemtype data;
    struct BiThrNode *lchild;
    struct BiThrNode *rchild;
    unsigned ltag:1;
    unsigned rtag:1;
   }BiThrNodeType,*BiThrTree;
```

下面以中序线索二叉树为例，讨论线索二叉树的建立、线索二叉树的遍历以及在线索二叉树上查找前驱结点、查找后继结点、插入结点和删除结点等操作的实现算法。

1．建立一棵中序线索二叉树

建立线索二叉树，或者说对二叉树线索化，实质上就是遍历一棵二叉树。在遍历过程中，访问结点的操作是检查当前结点的左、右指针域是否为空，如果为空，将它们改为指向前驱结点或后继结点的线索。为实现这一过程，设指针 pre 始终指向刚刚访问过的结点，即若指针 p 指向当前结点，则 pre 指向它的前驱，以便增设线索。

另外，在对一棵二叉树加线索时，必须首先申请一个头结点，建立头结点与二叉树的根结点的指向关系，对二叉树线索化后，还需建立最后一个结点与头结点之间的线索。

下面是建立中序线索二叉树的递归算法，其中 pre 为全局变量。

```
int  InOrderThr(BiThrTree *head,BiThrTree T)
{/*中序遍历二叉树 T，并将其中序线索化，*head 指向头结点。*/
  if (!(*head =(BiThrNodeType*)malloc(sizeof(BiThrNodeType))))  return 0;
  (*head)->ltag=0;    (*head)->rtag=1;           /*建立头结点*/
  (*head)->rchild=*head;                        /*右指针回指*/
  if (!T) (*head)->lchild =*head;                 /*若二叉树为空，则左指针回指*/
  else { (*head)->lchild=T;  pre= head;
       InThreading(T);                         /*中序遍历进行中序线索化*/
       pre->rchild=*head;  pre->rtag=1;         /*最后一个结点线索化*/
       (*head)->rchild=pre;
      }
  return 1;
}
void InTreading(BiThrTree p)
{/*中序遍历进行中序线索化*/
  if (p)
   { InThreading(p->lchild);         /*左子树线索化*/
     if (!p->lchild)                 /*前驱线索*/
       { p->ltag=1;   p->lchild=pre;
       }
```

```
    if (!pre->rchild)              /*后继线索*/
      { pre->rtag=1;   pre->rchild=p;
       }
    pre=p;
    InThreading(p->rchild);              /*右子树线索化*/

  }
}
```

2．在中序线索二叉树上查找任意结点的中序前驱结点

对于中序线索二叉树上的任一结点，寻找其中序的前驱结点，有以下两种情况：

（1） 如果该结点的左标志为 1，那么其左指针域所指向的结点便是它的前驱结点；

（2）如果该结点的左标志为 0，表明该结点有左孩子，根据中序遍历的定义，它的前驱结点是以该结点的左孩子为根结点的子树的最右结点，即沿着其左子树的右指针链向下查找，当某结点的右标志为 1 时，它就是所要找的前驱结点。

在中序线索二叉树上寻找结点 p 的中序前驱结点的算法如下：

```
BiThrTree InPreNode（BiThrTree p）
{/*在中序线索二叉树上寻找结点 p 的中序前驱结点*/
  BiThrTree pre;
  pre=p->lchild;
  if (p->ltag!=1)
     while (pre->rtag==0) pre=pre->rchild;
 return(pre);
}
```

3．在中序线索二叉树上查找任意结点的中序后继结点

对于中序线索二叉树上的任一结点，寻找其中序的后继结点，有以下两种情况：

（1） 如果该结点的右标志为 1，那么其右指针域所指向的结点便是它的后继结点；

（2）如果该结点的右标志为 0，表明该结点有右孩子，根据中序遍历的定义，它的前驱结点是以该结点的右孩子为根结点的子树的最左结点，即沿着其右子树的左指针链向下查找，当某结点的左标志为 1 时，它就是所要找的后继结点。

在中序线索二叉树上寻找结点 p 的中序后继结点的算法如下：

```
BiThrTree InPostNode（BiThrTree p）
{/*在中序线索二叉树上寻找结点 p 的中序后继结点*/
  BiThrTree post;
  post=p->rchild;
  if (p->rtag!=1)
     while (post->rtag==0) post=post->lchild;
 return(post);
}
```

4．在中序线索二叉树上查找任意结点在先序下的后继

这一操作的实现依据是：若一个结点是某子树在中序下的最后一个结点，则它必是该子树在先序下的最后一个结点。该结论可以用反证法证明。

下面就依据这一结论，讨论在中序线索二叉树上查找某结点在先序下后继结点的情况。设开始时，指向此某结点的指针为 p。

（1） 若待确定先序后继的结点为分支结点，则又有两种情况：

① 当 p->ltag=0 时，p->lchild 为 p 在先序下的后继；

② 当 p->ltag=1 时，p->rchild 为 p 在先序下的后继。

（2） 若待确定先序后继的结点为叶子结点，则也有两种情况：

① 若 p->rchild 是头结点，则遍历结束；

② 若 p->rchild 不是头结点，则 p 结点一定是以 p->rchild 结点为根的左子树中在中序遍历下的最后一个结点，因此 p 结点也是在该子树中按先序遍历的最后一个结点。此时，若 p->rchild 结点有右子树，则所找结点在先序下的后继结点的地址为 p->rchild->rchild；若 p->rchild 为线索，则让 p＝p->rchild，反复情况（2）的判定。

在中序线索二叉树上寻找结点 p 的先序后继结点的算法如下：

```
BiThrTree IPrePostNode (BiThrTree head,BiThrTree p)
{/*在中序线索二叉树上寻找结点 p 的先序的后继结点,head 为线索树的头结点*/
  BiThrTree post;
  if (p->ltag==0) post=p->lchild;
  else { post=p;
        while (post->rtag==1&&post->rchild!=head) post=post->rchild;
        post=post->rchild;
       }
 return(post);
}
```

5．在中序线索二叉树上查找任意结点在后序下的前驱

这一操作的实现依据是：若一个结点是某子树在中序下的第一个结点，则它必是该子树在后序下的第一个结点。该结论可以用反证法证明。

下面就依据这一结论，讨论在中序线索二叉树上查找某结点在后序下前驱结点的情况。设开始时，指向此某结点的指针为 p。

（1）若待确定后序前驱的结点为分支结点，则又有两种情况：

① 当 p->ltag=0 时，p->lchild 为 p 在后序下的前驱；

② 当 p->ltag=1 时，p->rchild 为 p 在后序下的前驱。

（2）若待确定后序前驱的结点为叶子结点，则也有两种情况：

① 若 p->lchild 是头结点，则遍历结束；

② 若 p->lchild 不是头结点，则 p 结点一定是以 p->lchild 结点为根的右子树中在中中序遍历下的第一个结点，因此 p 结点也是在该子树中按后序遍历的第一个结点。此时，若 p->lchild 结点有左子树，则所找结点在后序下的前驱结点的地址为 p->lchild->lchild；若 p->lchild 为线索，则让 p＝p->lchild，反复情况（2）的判定。

在中序线索二叉树上寻找结点 p 的后序前驱结点的算法如下：

```
BiThrTree IPostPretNode (BiThrTree head,BiThrTree p)
{/*在中序线索二叉树上寻找结点 p 的先序的后继结点，head 为线索树的头结点*/
  BiThrTree pre;
  if (p->rtag==0) pre=p->rchild;
  else { pre=p;
        while (pre->ltag==1&& post->rchild!=head) pre=pre->lchild;
        pre=pre->lchild;
       }
 return(pre);
}
```

6．在中序线索二叉树上查找值为 x 的结点

利用在中序线索二叉树上寻找后继结点和前驱结点的算法，就可以遍历到二叉树的所有结点。比如，先找到按某序遍历的第一个结点，然后再依次查询其后继；或先找到按某序遍历的最后一个结点，然后再依次查询其前驱。这样，既不用栈也不用递归就可以访问到二叉树的所有结点。

在中序线索二叉树上查找值为 x 的结点，实质上就是在线索二叉树上进行遍历，将访问结点的操作具体写为那结点的值与 x 比较的语句。下面给出其算法：

```
BiThrTree Search (BiThrTree head,elemtype x)
{/*在以 head 为头结点的中序线索二叉树中查找值为 x 的结点*/
  BiThrTree p;
  p=head->lchild;
  while (p->ltag==0&&p!=head) p=p->lchild;
  while(p!=head && p->data!=x) p=InPostNode(p);
  if (p==head)
    { printf(“Not Found the data!\n”);
      return(0);
    }
```

```
  else  return(p);
 }
```

7．在中序线索二叉树上的更新

线索二叉树的更新是指，在线索二叉树中插入一个结点或者删除一个结点。一般情况下，这些操作有可能破坏原来已有的线索，因此，在修改指针时，还需要对线索做相应的修改。一般来说，这个过程的代价几乎与重新进行线索化相同。这里仅讨论一种比较简单的情况，即在中序线索二叉树中插入一个结点 p，使它成为结点 s 的右孩子。

下面分两种情况来分析：

（1）若 s 的右子树为空，如图 6.13 (a)所示，则插入结点 p 之后成为图 6.13 (b)所示的情形。在这种情况中，s 的后继将成为 p 的中序后继，s 成为 p 的中序前驱，而 p 成为 s 的右孩子。二叉树中其它部分的指针和线索不发生变化。

（2）若 s 的右子树非空，如图 6.14 (a)所示，插入结点 p 之后如图 6.14 (b)所示。S 原来的右子树变成 p 的右子树，由于 p 没有左子树，故 s 成为 p 的中序前驱，p 成为 s 的右孩子；又由于 s 原来的后继成为 p 的后继，因此还要将 s 原来的本来指向 s 的后继的左线索，改为指向 p。

下面给出上述操作的算法。

```
void InsertThrRight(BiThrTree s,BiThrTree p)
{/*在中序线索二叉树中插入结点 p 使其成为结点 s 的右孩子*/
 BiThrTree w;
  p->rchild=s->rchild;
  p->rtag=s->rtag;
  p->lchild=s;
  p->ltag=1;           /*将 s 变为 p 的中序前驱*/
  s->rchild=p;
  s->rtag=0;           /*p 成为 s 的右孩子*/
  if(p->rtag==0) /*当 s 原来右子树不空时，找到 s 的后继 w，变 w 为 p 的后继，p 为 w 的前驱*/
     { w=InPostNode(p);
       w->lchild=p;
     }
}
```

6.5　二叉树的应用

6.5.1　二叉树遍历的应用

在以上讨论的遍历算法中，访问结点的数据域信息，即操作 Visite（bt->data）具有更一般的意义，需根据具体问题，对 bt 数据进行不同的操作。下面介绍几个遍历操作的典型应用。

1．查找数据元素

Search（bt，x）在 bt 为二叉树的根结点指针的二叉树中查找数据元素 x。查找成功时返回该结点的指针；查找失败时返回空指针。

算法实现如下，注意遍历算法中的 Visite(bt->data)等同于其中的一组操作步骤。

```
BiTree  Search（BiTree bt，elemtype x）
{/*在 bt 为根结点指针的二叉树中查找数据元素 x*/
  BiTree  p;
  if (bt->data==x) return bt;       /*查找成功返回*/
  if (bt->lchild!=NULL) return(Search(bt->lchild,x));
  /*在 bt->lchild 为根结点指针的二叉树中查找数据元素 x*/
  if (bt->rchild!=NULL) return(Search(bt->rchild,x));
  /*在 bt->rchild 为根结点指针的二叉树中查找数据元素 x*/
  return NULL;      /*查找失败返回*/
}
```

2．统计出给定二叉树中叶子结点的数目

（1）顺序存储结构的实现

```
int CountLeaf1（SqBiTree bt，int k）
{/*一维数组 bt[2k-1]为二叉树存储结构，k 为二叉树深度，函数值为叶子数。*/
```

```
 total=0;
 for(i=1;i<=2^k-1;i++)
   { if (bt[i]!=0)
      { if ((bt[2i]==0 && bt[2i+1]==0) || (i>(2^k-1)/2))
        total++;
      }
   }
return(total);
}
```

（2）二叉链表存储结构的实现

```
int CountLeaf2（BiTree  bt）
{/*开始时，bt 为根结点所在链结点的指针，返回值为 bt 的叶子数*/
 if (bt==NULL) return(0);
 if (bt->lchild==NULL && bt->rchild==NULL) return(1);
 return(CountLeaf2(bt->lchild)+CountLeaf2(bt->rchild));
}
```

3．创建二叉树二叉链表存储，并显示。

设创建时，按二叉树带空指针的先序次序输入结点值，结点值类型为字符型。输出按中序输出。

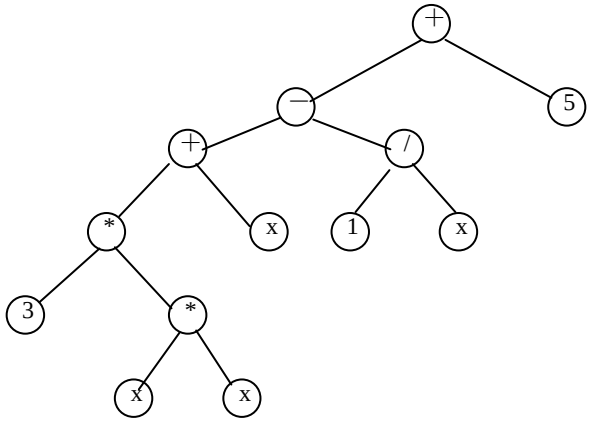
CreateBinTree（BinTree *bt）是以二叉链表为存储结构建立一棵二叉树 T 的存储，bt 为指向二叉树 T 根结点指针的指针。InOrderOut（bt）为按中序输出二叉树 bt 的结点。

算法实现如下，注意在创建算法中，遍历算法中的 Visite(bt->data)被读入结点、申请空间存储的操作所代替；在输出算法中，遍历算法中的 Visite(bt->data)被 c 语言中的格式输出语句所代替。

```
void CreateBinTree(BinTree *T)
{/*以加入结点的先序序列输入，构造二叉链表*/
 char ch;
 scanf("\n%c",&ch);
 if (ch=='0')  *T=NULL;              /*读入 0 时，将相应结点置空*/
 else {*T=(BinTNode*)malloc(sizeof(BinTNode));  /*生成结点空间*/
      (*T)->data=ch;
      CreateBinTree(&(*T)->lchild);      /*构造二叉树的左子树*/
      CreateBinTree(&(*T)->rchild);      /*构造二叉树的右子树*/
     }
}
void InOrderOut(BinTree T)
{/*中序遍历输出二叉树 T 的结点值*/
 if (T)
   { InOrderOut(T->lchild);                    /*中序遍历二叉树的左子树*/
    printf("%3c",T->data);                  /*访问结点的数据*/
    InOrderOut(T->rchild);                    /*中序遍历二叉树的右子树*/
   }
}
main()
{BiTree bt;
 CreateBinTree(&bt);
 InOrderOut(bt);
}
```

4．表达式运算

我们可以把任意一个算数表达式用一棵二叉树表示，图 6.15 所示为表达式 $3x^2+x-1/x+5$ 的二叉树表示。在表达式二叉树中，每个叶结点都是操作数，每个非叶结点都是运算符。对于一个非叶子结点，它的左、右子树分别是它的两个操作数。



对该二叉树分别进行先序、中序和后序遍历，可以得到表达式的三种不同表示形式。

前缀表达式　　+−+*3*xxx/1x5

中缀表达式　　3*x*x+x−1/x+5

后缀表达式　　3xx**x+1x/−5+

6.5.2　最优二叉树——哈夫曼树

1．哈夫曼树的基本概念

最优二叉树，也称哈夫曼（Haffman）树，是指对于一组带有确定权值的叶结点，构造的具有最小带权路径长度的二叉树。

在前面我们介绍过路径和结点的路径长度的概念，而二叉树的路径长度则是指由根结点到所有叶结点的路径长度之和。如果二叉树中的叶结点都具有一定的权值，则可将这一概念加以推广。设二叉树具有 n 个带权值的叶结点，那么从根结点到各个叶结点的路径长度与相应结点权值的乘积之和叫做二叉树的带权路径长度，记为：

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

其中 W_k 为第 k 个叶结点的权值，L_k 为第 k 个叶结点的路径长度。如图 6.16 所示的二叉树，它的带权路径长度值 WPL＝2×2＋4×2＋5×2＋3×2＝28。

在给定一组具有确定权值的叶结点，可以构造出不同的带权二叉树。例如，给出 4 个叶结点，设其权值分别为 1，3，5，7，我们可以构造出形状不同的多个二叉树。这些形状不同的二叉树的带权路径长度将各不相同。图 6.17 给出了其中 5 个不同形状的二叉树。

这五棵树的带权路径长度分别为：

（a）WPL＝1×2＋3×2＋5×2＋7×2＝32

（b）WPL＝1×3＋3×3＋5×2＋7×1＝29

（c）WPL＝1×2＋3×3＋5×3＋7×1＝33

（d）WPL＝7×3＋5×3＋3×2＋1×1＝43

（e）WPL＝7×1＋5×2＋3×3＋1×3＝29

根据哈夫曼树的定义，一棵二叉树要使其 WPL 值最小，必须使权值越大的叶结点越靠近根结点，而权值越小的叶结点越远离根结点。哈夫曼（Haffman）依据这一特点提出了一种方法，这种方法的基本思想是：

（1）由给定的 n 个权值{W1，W2，…，Wn}构造 n 棵只有一个叶结点的二叉树，从而得到一个二叉树的集合 F＝{T1，T2，…，Tn}；

（2）在 F 中选取根结点的权值最小和次小的两棵二叉树作为左、右子树构造一棵新的二叉树，这棵新的二叉树根结点的权值为其左、右子树根结点权值之和；

（3）在集合 F 中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到集合 F 中；

（4）重复（2）（3）两步，当 F 中只剩下一棵二叉树时，这棵二叉树便是所要建立的哈夫曼树。

2．哈夫曼树的构造算法

在构造哈夫曼树时，可以设置一个结构数组 HuffNode 保存哈夫曼树中各结点的信息，根据二叉树的性质可知，具有 n 个叶子结点的哈夫曼树共有 2n－1 个结点，所以数组 HuffNode 的大小设置为 2n－1，数组元素的结构形式如下：

weight	lchild	rchild	parent

其中，weight 域保存结点的权值，lchild 和 rchild 域分别保存该结点的左、右孩子结点在数组 HuffNode 中的序号，从而建立起结点之间的关系。为了判定一个结点是否已加入到要建立的哈夫曼树中，可通过 parent 域的值来确定。初始时 parent 的值为

－1，当结点加入到树中时，该结点 parent 的值为其双亲结点在数组 HuffNode 中的序号，就不会是－1 了。

构造哈夫曼树时，首先将由 n 个字符形成的 n 个叶结点存放到数组 HuffNode 的前 n 个分量中，然后根据前面介绍的哈夫曼方法的基本思想，不断将两个小子树合并为一个较大的子树，每次构成的新子树的根结点顺序放到 HuffNode 数组中的前 n 个分量的后面。

下面给出哈夫曼树的构造算法。

```
#define MAXVALUE 10000        /*定义最大权值*/
#define MAXLEAF 30            /*定义哈夫曼树中叶子结点个数*/
#define MAXNODE  MAXLEAF*2-1
typedef struct {
    int weight;
    int parent;
    int lchild;
    int rchild;
   }HNodeType;
void  HaffmanTree(HNodeType HuffNode [ ])
 {/*哈夫曼树的构造算法*/
  int i,j,m1,m2,x1,x2,n;
  scanf(“%d”,&n);             /*输入叶子结点个数*/
  for (i=0;i<2*n-1;i++)          /*数组 HuffNode[ ]初始化*/
    { HuffNode[i].weight=0;
      HuffNode[i].parent=-1;
      HuffNode[i].lchild=-1;
      HuffNode[i].rchild=-1;
     }
  for (i=0;i<n;i++) scanf(“%d”,&HuffNode[i].weight);  /*输入 n 个叶子结点的权值*/
  for (i=0;i<n-1;i++)             /*构造哈夫曼树*/
    { m1=m2=MAXVALUE;
      x1=x2=0;
      for (j=0;j<n+i;j++)
       { if (HuffNode[j].weight<m1 && HuffNode[j].parent==-1)
          { m2=m1;     x2=x1;
            m1=HuffNode[j].weight;    x1=j;
          }
         else if (HuffNode[j].weight<m2 && HuffNode[j].parent==-1)
              { m2=HuffNode[j].weight;
                x2=j;
              }
       }
     /*将找出的两棵子树合并为一棵子树*/
     HuffNode[x1].parent=n+i;         HuffNode[x2].parent=n+i;
     HuffNode[n+i].weight= HuffNode[x1].weight+HuffNode[x2].weight;
     HuffNode[n+i].lchild=x1;  HuffNode[n+i].rchild=x2;
    }
}
```

7．1．1　树的定义及相关术语

1．树的定义

树（Tree）是n（n≥0）个有限数据元素的集合。当n＝0时，称这棵树为空树。在一棵非树T中：

（1）有一个特殊的数据元素称为树的根结点，根结点没有前驱结点。

（2）若n>1，除根结点之外的其余数据元素被分成m（m>0）个互不相交的集合T_1，T_2，…，T_m，其中每一个集合T_i（1≤i≤m）本身又是一棵树。树T_1，T_2，…，T_m称为这个根结点的子树。

可以看出，在树的定义中用了递归概念，即用树来定义树。因此，树结构的算法类同于二叉树结构的算法，也可以使用递归方法。

树的定义还可形式化的描述为二元组的形式：

T＝（D，R）

其中D为树T中结点的集合，R为树中结点之间关系的集合。

当树为空树时，D＝Φ；当树T不为空树时有：

$D=\{Root\}\cup D_F$

其中，Root为树T的根结点，DF为树T的根Root的子树集合。D_F可由下式表示：

$D_F=D_1\cup D_2\cup\cdots\cup D_m$且$D_i\cap D_j=\Phi$（i≠j，1≤i≤m，1≤j≤m）

当树T中结点个数n≤1时，R＝Φ；当树T中结点个数n>1时有：

$R=\{<Root, r_i>, i=1, 2, \cdots, m\}$

其中，Root为树T的根结点，r_i是树T的根结点Root的子树T_i的根结点。

树定义的形式化，主要用于树的理论描述。

图7.1(a)是一棵具有9个结点的树，即T＝{A，B，C，…，H，I}，结点A为树T的根结点，除根结点A之外的其余结点分为两个不相交的集合：T_1＝{B,D,E,F,H,I}和T_2={C,G}，T_1和T_2构成了结点A的两棵子树，T_1和T_2本身也分别是一棵树。例如，子树T_1的根结点为B，其余结点又分为两个不相交的集合：T_{11}＝{D}，T_{12}＝{E,H,I}和T_{13}＝{F}。T_{11}、T_{12}和T_{13}构成了子树T_1的根结点B的三棵子树。如此可继续向下分为更小的子树，直到每棵子树只有一个根结点为止。

从树的定义和图7.1(a)的示例可以看出，树具有下面两个特点：

（1） 树的根结点没有前驱结点，除根结点之外的所有结点有且只有一个前驱结点。

（2） 树中所有结点可以有零个或多个后继结点。

2．相关术语

（1）有序树和无序树。如果一棵树中结点的各子树从左到右是有次序的，即若交换了某结点各子树的相对位置，则构成不同的树，称这棵树为有序树；反之，则称为无序树。

（2）森林。零棵或有限棵不相交的树的集合称为森林。自然界中树和森林是不同的概念，但在数据结构中，树和森林只有很小的差别。任何一棵树，删去根结点就变成了森林。

7．1．2　树的表示

1．直观表示法

树的直观表示法就是以倒着的分支树的形式表示，图7.1(a)就是一棵树的直观表示。其特点就是对树的逻辑结构的描述非常直观。是数据结构中最常用的树的描述方法。

2．嵌套集合表示法

所谓嵌套集合是指一些集合的集体，对于其中任何两个集合，或者不相交，或者一个包含另一个。用嵌套集合的形式表示树，就是将根结点视为一个大的集合，其若干棵子树构成这个大集合中若干个互不相交的子集，如此嵌套下去，即构成一棵树的嵌套集合表示。图7.2 (a)就是一棵树的嵌套集合表示。

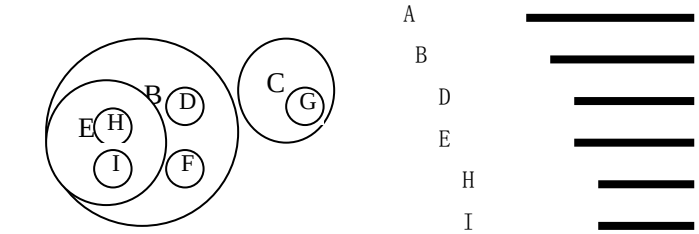
3．凹入表示法

树的凹入表示法如图7.2 (c)所示。

树的凹入表示法主要用于树的屏幕和打印输出。

4．广义表表示法

树用广义表表示，就是将根作为由子树森林组成的表的名字写在表的左边，这样依次将书表示出来。图7.2 (b)就是一棵树的广义表表示。



(a)

(A(B(D,E(H,I),F),C(G)))

(b)

(c)

图 7.2　对图 7.1(a)所示树的其它三种表示法示意

7．2　树的基本操作与存储

7．2．1　树的基本操作

树的基本操作通常有以下几种：

（1） Initiate（t）初始化一棵空树 t。

（2） Root（x）求结点 x 所在树的根结点。

（3） Parent（t，x）求树 t 中结点 x 的双亲结点。

（4） Child（t，x，i）求树 t 中结点 x 的第 i 个孩子结点。

（5） RightSibling（t，x）求树 t 中结点 x 的第一个右边兄弟结点。

（6） Insert（t，x，i，s）把以 s 为根结点的树插入到树 t 中作为结点 x 的第 i 棵子树。

（7） Delete（t，x，i）在树 t 中删除结点 x 的第 i 棵子树。

（8） Tranverse（t）是树的遍历操作，即按某种方式访问树 t 中的每个结点，且使每个结点只被访问一次。

7．2．2　树的存储结构

1．双亲表示法

由树的定义可以知道，树中的每个结点都有唯一的一个双亲结点，根据这一特性，可用一组连续的存储空间（一维数组）存储树中的各个结点，数组中的一个元素表示树中的一个结点，数组元素为结构体类型，其中包括结点本身的信息以及结点的双亲结点在数组中的序号，树的这种存储方法称为双亲表示法。

```
#define MAXNODE <树中结点的最大个数>
typedef struct {
    elemtype  data;
    int  parent;
   }NodeType;
NodeType  t[MAXNODE];
```

图中用 parent 域的值为-1 表示该结点无双亲结点，即该结点是一个根结点。

树的双亲表示法对于实现 Parent（t，x）操作和 Root（x）操作很方便，但若求某结点的孩子结点，即实现 Child（t，x，i）操作时，则需要查询整个数组。此外，这种存储方式不能反映各兄弟结点之间的关系，所以实现 RightSibling（t，x）操作也比较困难。在实际中，如果需要实现这些操作，可在结点结构中增设存放第一个孩子的域和存放第一个右兄弟的域，就能较方便地实现上述操作了。

2．孩子表示法

（1）多重链表法

由于树中每个结点都有零个或多个孩子结点，因此，可以令每个结点包括一个结点信息域和多个指针域，每个指针域指向该结点的一个孩子结点，通过各个指针域值反映出树中各结点之间的逻辑关系。在这种表示法中，树中每个结点有多个指针域，形成了多条链表，所以这种方法又常称为多重链表法。

在一棵树中，各结点的度数各异，因此结点的指针域个数的设置有两种方法：

① 每个结点指针域的个数等于该结点的度数；

② 每个结点指针域的个数等于树的度数。

对于方法①，它虽然在一定程度上节约了存储空间，但由于树中各结点是不同构的，各种操作不容易实现，所以这种方法很少采用；方法②中各结点是同构的，各种操作相对容易实现，但为此付出的代价是存储空间的浪费。图 7.4 是图 7.1(a)所示的树采用

这种方法的存储结构示意图。显然，方法②适用于各结点的度数相差不大的情况。

树中结点的存储表示可描述为：

```
#define MAXSON <树的度数>
typedef struct TreeNode {
     elemtype data;
     struct TreeNode  *son[MAXSON];
   }NodeType;
```

对于任意一棵树 t，可以定义：NodeType *t;使变量 t 为指向树的根结点的指针。

（2）孩子链表表示法

其主体是一个与结点个数一样大小的一维数组，数组的每一个元素有两个域组成，一个域用来存放结点信息，另一个用来存放指针，该指针指向由该结点孩子组成的单链表的首位置。单链表的结构也由两个域组成，一个存放孩子结点在一维数组中的序号，另一个是指针域，指向下一个孩子。

在孩子表示法中查找双亲比较困难，查找孩子却十分方便，故适用于对孩子操作多的应用。

这种存储表示可描述为：

```
#define MAXNODE <树中结点的最大个数>
typedef struct ChildNode{
    int childcode;
    struct ChildNode *nextchild;
   }
typedef struct {
    elemtype  data;
    struct ChildNode *firstchild;
   }NodeType;
NodeType  t[MAXNODE];
```

3．双亲孩子表示法

双亲表示法是将双亲表示法和孩子表示法相结合的结果。其仍将各结点的孩子结点分别组成单链表，同时用一维数组顺序存储树中的各结点，数组元素除了包括结点本身的信息和该结点的孩子结点链表的头指针之外，还增设一个域，存储该结点双亲结点在数组中的序号。

4．孩子兄弟表示法

其方法是这样的：在树中，每个结点除其信息域外，再增加两个分别指向该结点的第一个孩子结点和下一个兄弟结点的指针。在这种存储结构下，树中结点的存储表示可描述为：

```
typedef struct TreeNode {
    elemtype data;
    struct TreeNode  *son;
    struct TreeNode  *next;
   }NodeType;
```

7．3　树、森林与二叉树的转换

7．3．1　树转换为二叉树

对于一棵无序树，树中结点的各孩子的次序是无关紧要的，而二叉树中结点的左、右孩子结点是有区别的。为避免发生混淆，我们约定树中每一个结点的孩子结点按从左到右的次序顺序编号。如图 7.8 所示的一棵树，根结点 A 有 B、C、D 三个孩子，可以认为结点 B 为 A 的第一个孩子结点，结点 C 为 A 的第二个孩子结点，结点 D 为 A 的第三个孩子结点。

将一棵树转换为二叉树的方法是：

（1） 树中所有相邻兄弟之间加一条连线。

（2） 对树中的每个结点，只保留它与第一个孩子结点之间的连线，删去它与其它孩子结点之间的连线。

（3） 以树的根结点为轴心，将整棵树顺时针转动一定的角度，使之结构层次分明。

可以证明，树作这样的转换所构成的二叉树是唯一的。

由上面的转换可以看出，在二叉树中，左分支上的各结点在原来的树中是父子关系，而右分支上的各结点在原来的树中是兄弟关系。由于树的根结点没有兄弟，所以变换后的二叉树的根结点的右孩子必为空。

事实上，一棵树采用孩子兄弟表示法所建立的存储结构与它所对应的二叉树的二叉链表存储结构是完全相同的。

7．3．2　森林转换为二叉树

由森林的概念可知，森林是若干棵树的集合，只要将森林中各棵树的根视为兄弟，每棵树又可以用二叉树表示，这样，森林也同样可以用二叉树表示。

森林转换为二叉树的方法如下：

（1） 将森林中的每棵树转换成相应的二叉树。

（2）第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树根结点的右孩子，当所有二叉树连起来后，此时所得到的二叉树就是由森林转换得到的二叉树。

这一方法可形式化描述为：

如果 F＝{ T_1，T_2，…，T_m }是森林，则可按如下规则转换成一棵二叉树 B＝（root，LB，RB）。

（1） 若 F 为空，即 m＝0，则 B 为空树；

（2）若 F 非空，即 m≠0，则 B 的根 root 即为森林中第一棵树的根 Root(T1)；B 的左子树 LB 是从 T1 中根结点的子树森林 F1＝{ T_{11}，T_{12}，…，T_{1m1} }转换而成的二叉树；其右子树 RB 是从森林 F’＝{ T_2，T_3，…，T_m }转换而成的二叉树。

7．3．3　二叉树转换为树和森林

树和森林都可以转换为二叉树，二者不同的是树转换成的二叉树，其根结点无右分支，而森林转换后的二叉树，其根结点有右分支。显然这一转换过程是可逆的，即可以依据二叉树的根结点有无右分支，将一棵二叉树还原为树或森林，具体方法如下：

（1）若某结点是其双亲的左孩子，则把该结点的右孩子、右孩子的右孩子……都与该结点的双亲结点用线连起来；

（2）删去原二叉树中所有的双亲结点与右孩子结点的连线；

（3）整理由（1)、(2）两步所得到的树或森林，使之结构层次分明。

这一方法可形式化描述为：

如果 B＝（root，LB，RB）是一棵二叉树，则可按如下规则转换成森林 F＝{ T_1，T_2，…，T_m }。

（1）若 B 为空，则 F 为空；

（2）若 B 非空，则森林中第一棵树 T1 的根 ROOT（T1）即为 B 的根 root；T1 中根结点的子树森林 F1 是由 B 的左子树 LB 转换而成的森林；F 中除 T1 之外其余树组成的森林 F’＝{ T_2，T_3，…，T_m }是由 B 的右子树 RB 转换而成的森林。

7．4　树和森林的遍历

7．4．1　树的遍历

1．先根遍历

先根遍历的定义为：

（1） 访问根结点；

（2） 按照从左到右的顺序先根遍历根结点的每一棵子树。

2．后根遍历

后根遍历的定义为：

（1） 按照从左到右的顺序后根遍历根结点的每一棵子树。

（2） 访问根结点；

根据树与二叉树的转换关系以及树和二叉树的遍历定义可以推知，树的先根遍历与其转换的相应二叉树的先序遍历的结果序列相同；树的后根遍历与其转换的相应二叉树的中序遍历的结果序列相同。因此树的遍历算法是可以采用相应二叉树的遍历算法来实现的。

7．4．2　森林的遍历

1．前序遍历

前序遍历的定义为：

（1） 访问森林中第一棵树的根结点；

（2） 前序遍历第一棵树的根结点的子树；

（3） 前序遍历去掉第一棵树后的子森林。

2．中序遍历

中序遍历的定义为：

（1） 中序遍历第一棵树的根结点的子树；

（2） 访问森林中第一棵树的根结点；

（3） 中序遍历去掉第一棵树后的子森林。

根据森林与二叉树的转换关系以及森林和二叉树的遍历定义可以推知，森林的前序遍历和中序遍历与所转换的二叉树的先序遍历和中序遍历的结果序列相同。

第八章　　图

在树状结构中，结点间具有分支层次关系，每一层上的结点只能和上一层中的至多一个结点相关，但可能和下一层的多个结点相关。而在图状结构中，任意两个结点之间都可能相关，即结点之间的邻接关系可以是任意的。

8.1　图的基本概念

8.1.1　　图的定义和术语

1．图的定义

图(Graph)是由非空的顶点集合和一个描述顶点之间关系－－边（或者弧）的集合组成，其形式化定义为：

G＝（V，E）

V＝{vi｜ vi∈dataobject}

E＝{(vi,vj)｜ vi, vj ∈V ∧P(vi, vj)}

其中，G 表示一个图，V 是图 G 中顶点的集合，E 是图 G 中边的集合，集合 E 中 P(vi,vj)表示顶点 vi 和顶点 vj 之间有一条直接连线，即偶对(vi,vj)表示一条边。

2．图的相关术语

（1）无向图。在一个图中，如果任意两个顶点构成的偶对（vi, vj）∈E 是无序的，即顶点之间的连线是没有方向的，则称该图为无向图。

（2）有向图。在一个图中，如果任意两个顶点构成的偶对（vi, vj）∈E 是有序的，即顶点之间的连线是有方向的，则称该图为有向图。

G2=(V2,E2)

V2={v1,v2,v3,v4}

E2={<v1,v2>,<v1,v3>,<v3,v4>,<v4,v1>}

（3）顶点、边、弧、弧头、弧尾。图中，数据元素 vi 称为顶点(vertex)；P(vi, vj)表示在顶点 vi 和顶点 vj 之间有一条直接连线。如果是在无向图中，则称这条连线为边；如果是在有向图中，一般称这条连线为弧。边用顶点的无序偶对（vi, vj）来表示，称顶点 vi 和顶点 vj 互为邻接点，边（vi, vj）依附于顶点 vi 与顶点 vj；弧用顶点的有序偶对<vi, vj>来表示，有序偶对的第一个结点 vi 被称为始点（或弧尾），在图中就是不带箭头的一端；有序偶对的第二个结点 vj 被称为终点（或弧头），在图中就是带箭头的一端。

（4）无向完全图。在一个无向图中，如果任意两顶点都有一条直接边相连接，则称该图为无向完全图。可以证明，在一个含有 n 个顶点的无向完全图中，有 n(n-1)/2 条边。

（5）有向完全图。在一个有向图中，如果任意两顶点之间都有方向互为相反的两条弧相连接，则称该图为有向完全图。在一个含有 n 个顶点的有向完全图中，有 n(n-1)条边。

（6）稠密图、稀疏图。若一个图接近完全图，称为稠密图；称边数很少的图为稀疏图。

（7）顶点的度、入度、出度。顶点的度（degree）是指依附于某顶点 v 的边数，通常记为 TD (v)。在有向图中，要区别顶点的入度与出度的概念。顶点 v 的入度是指以顶点 为终点的弧的数目。记为 ID (v)；顶点 v 出度是指以顶点 v 为始点的弧的数目，记为 OD (v)。有 TD (v)=ID (v)＋OD (v)。

（8）边的权、网图。与边有关的数据信息称为权（weight）。

（9）路径、路径长度。顶点 v_p 到顶点 v_q 之间的路径（path）是指顶点序列 $v_p, v_{i1}, v_{i2}, \cdots, v_{im}, v_q$。其中，（$v_p, v_{i1}$），($v_{i1}, v_{i2}$)，…,($v_{im}$,.$v_q$)分别为图中的边。路径上边的数目称为路径长度。

（10）回路、简单路径、简单回路。称 vi 的路径为回路或者环（cycle）。序列中顶点不重复出现的路径称为简单路径。在图 8.1 中，前面提到的 v1 到 v5 的两条路径都为简单路径。除第一个顶点与最后一个顶点之外，其他顶点不重复出现的回路称为简单回路，或者简单环。

（11）子图。对于图 G=（V，E），G’=（V’，E’），若存在 V’是 V 的子集 ，E’是 E 的子集 ，则称图 G’是 G 的一个子图。

（12）连通的、连通图、连通分量。在无向图中，如果从一个顶点 vi 到另一个顶点 vj(i≠j)有路径，则称顶点 vi 和 vj 是连通的。如果图中任意两顶点都是连通的，则称该图是连通图。无向图的极大连通子图称为连通分量。

（13）强连通图、强连通分量。对于有向图来说，若图中任意一对顶点 vi 和 vj(i≠j)均有从一个顶点 vi 到另一个顶点 vj 有路径，也有从 vj 到 vi 的路径，则称该有向图是强连通图。有向图的极大强连通子图称为强连通分量。

（14）生成树。所谓连通图 G 的生成树，是 G 的包含其全部 n 个顶点的一个极小连通子图。它必定包含且仅包含 G 的 n-1 条边。在生成树中添加任意一条属于原图中的边必定会产生回路，因为新添加的边使其所依附的两个顶点之间有了第二条路径。若生成树中减少任意一条边，则必然成为非连通的。

（15）生成森林。在非连通图中，由每个连通分量都可得到一个极小连通子图，即一棵生成树。这些连通分量的生成树就组成了一个非连通图的生成森林。

8.1.2 图的基本操作

（1） CreatGraph（G）输入图 G 的顶点和边，建立图 G 的存储。

（2）DestroyGraph（G）释放图 G 占用的存储空间。

（3）GetVex（G，v）在图 G 中找到顶点 v，并返回顶点 v 的相关信息。

（4）PutVex（G，v，value）在图 G 中找到顶点 v，并将 value 值赋给顶点 v。

（5）InsertVex（G，v）在图 G 中增添新顶点 v。

（6）DeleteVex（G，v）在图 G 中，删除顶点 v 以及所有和顶点 v 相关联的边或弧。

（7）InsertArc（G，v，w）在图 G 中增添一条从顶点 v 到顶点 w 的边或弧。

（8）DeleteArc（G，v，w）在图 G 中删除一条从顶点 v 到顶点 w 的边或弧。

（9）DFSTraverse（G，v）在图 G 中，从顶点 v 出发深度优先遍历图 G。

（10）BFSTtaverse（G，v）在图 G 中，从顶点 v 出发广度优先遍历图 G。

在一个图中，顶点是没有先后次序的，但当采用某一种确定的存储方式存储后，存储结构中顶点的存储次序构成了顶点之间的相对次序，这里用顶点在图中的位置表示该顶点的存储顺序；同样的道理，对一个顶点的所有邻接点，采用该顶点的第 i 个邻接点表示与该顶点相邻接的某个顶点的存储顺序，在这种意义下，图的基本操作还有：

（11）LocateVex（G，u）在图 G 中找到顶点 u，返回该顶点在图中位置。

（12）FirstAdjVex（G，v）在图 G 中，返回 v 的第一个邻接点。若顶点在 G 中没有邻接顶点，则返回“空”。

（13）NextAdjVex（G，v，w）在图 G 中，返回 v 的（相对于 w 的）下一个邻接顶点。若 w 是 v 的最后一个邻接点，则返回“空”。

8.2 图的存储表示

一个图的信息包括两部分，即图中顶点的信息以及描述顶点之间的关系－－边或者弧的信息。

8.2.1 邻接矩阵

所谓邻接矩阵（Adjacency Matrix）的存储结构，就是用一维数组存储图中顶点的信息，用矩阵表示图中各顶点之间的邻接关系。假设图 G＝（V，E）有 n 个确定的顶点，即 V＝{$v_0, v_1, \cdots, v_{n-1}$},则表示 G 中各顶点相邻关系为一个 n×n 的矩阵，矩阵的元素为：

$$A[i][j]=\begin{cases}1 & 若(v_i,v_j)或<v_i,v_j>是\ E(G)中的边\\ 0 & 若(v_i,v_j)或<v_i,v_j>不是\ E(G)中的边\end{cases}$$

若 G 是网图，则邻接矩阵可定义为：

$$A[i][j]=\begin{cases}w_{ij} & 若(v_i,v_j)或<v_i,v_j>是\ E(G)中的边\\ 0\ 或\infty & 若(v_i,v_j)或<v_i,v_j>不是\ E(G)中的边\end{cases}$$

其中，w_{ij}表示边(v_i, v_j)或$<v_i, v_j>$上的权值；∞表示一个计算机允许的、大于所有边上权值的数。

从图的邻接矩阵存储方法容易看出这种表示具有以下特点：

① 无向图的邻接矩阵一定是一个对称矩阵。因此，在具体存放邻接矩阵时只需存放上（或下）三角矩阵的元素即可。

② 对于无向图，邻接矩阵的第 i 行（或第 i 列）非零元素（或非∞元素）的个数正好是第 i 个顶点的度 TD(vi)。

③ 对于有向图,邻接矩阵的第 i 行(或第 i 列)非零元素(或非∞元素)的个数正好是第 i 个顶点的出度 OD(vi)(或入度 ID(vi))。

④用邻接矩阵方法存储图，很容易确定图中任意两个顶点之间是否有边相连；但是，要确定图中有多少条边，则必须按行、按列对每个元素进行检测，所花费的时间代价很大。这是用邻接矩阵存储图的局限性。

在用邻接矩阵存储图时，除了用一个二维数组存储用于表示顶点间相邻关系的邻接矩阵外，还需用一个一维数组来存储顶点信息，另外还有图的顶点数和边数。故可将其形式描述如下：

```
#define MaxVertexNum 100          /*最大顶点数设为 100*/
typedef char VertexType;             /*顶点类型设为字符型*/
typedef int EdgeType;                /*边的权值设为整型*/
typedef struct {
   VertexType vexs[MaxVertexNum]; /*顶点表*/
   EdeType edges[MaxVertexNum][MaxVertexNum]; /*邻接矩阵，即边表*/
   int n,e;                          /*顶点数和边数*/
  }Mgragh;                          /*Maragh 是以邻接矩阵存储的图类型*/
```

建立一个图的邻接矩阵存储的算法如下：

```
 void CreateMGraph(MGraph *G)
 {/*建立有向图 G 的邻接矩阵存储*/
```

```
    int i,j,k,w;
    char ch;
    printf("请输入顶点数和边数(输入格式为:顶点数,边数):\n");
    scanf("%d,%d",&(G->n),&(G->e));/*输入顶点数和边数*/
    printf("请输入顶点信息(输入格式为:顶点号<CR>):\n");
    for (i=0;i<G->n;i++)  scanf("\n%c",&(G->vexs[i]));   /*输入顶点信息，建立顶点表*/
    for (i=0;i<G->n;i++)
     for (j=0;j<G->n;j++)  G->edges[i][j]=0;              /*初始化邻接矩阵*/
    printf("请输入每条边对应的两个顶点的序号(输入格式为:i,j):\n");
    for (k=0;k<G->e;k++)
      {scanf("\n%d,%d",&i,&j);        /*输入 e 条边，建立邻接矩阵*/
       G->edges[i][j]=1;              /*若加入 G->edges[j][i]=1;，*/
                                        /*则为无向图的邻接矩阵存储建立*/
      }
  }/*CreateMGraph*/
```

8.2.2 邻接表

邻接表(Adjacency List)是图的一种顺序存储与链式存储结合的存储方法。邻接表表示法类似于树的孩子链表表示法。就是对于图 G 中的每个顶点 vi，将所有邻接于 vi 的顶点 vj 链成一个单链表，这个单链表就称为顶点 vi 的邻接表，再将所有点的邻接表表头放到数组中，就构成了图的邻接表。

一种是顶点表的结点结构，它由顶点域（vertex）和指向第一条邻接边的指针域（firstedge）构成，另一种是边表（即邻接表）结点，它由邻接点域(adjvex)和指向下一条邻接边的指针域(next)构成。对于网图的边表需再增设一个存储边上信息（如权值等）的域（info）。

邻接表表示的形式描述如下：

```
#define MaxVerNum 100          /*最大顶点数为 100*/
typedef struct node{               /*边表结点*/
      int adjvex;                   /*邻接点域*/
      struct node  * next;          /*指向下一个邻接点的指针域*/
      /*若要表示边上信息，则应增加一个数据域 info*/
     }EdgeNode;
typedef struct vnode{              /*顶点表结点*/
      VertexType vertex;           /*顶点域*/
      EdgeNode  * firstedge;       /*边表头指针*/
     }VertexNode;
typedef VertexNode AdjList[MaxVertexNum];  /*AdjList 是邻接表类型*/
typedef struct{
     AdjList adjlist;                 /*邻接表*/
     int n,e;                        /*顶点数和边数*/
    }ALGraph;                       /*ALGraph 是以邻接表方式存储的图类型*/
```

建立一个有向图的邻接表存储的算法如下：

```
 void CreateALGraph(ALGraph *G)
 {/*建立有向图的邻接表存储*/
  int i,j,k;
  EdgeNode * s;
  printf("请输入顶点数和边数(输入格式为:顶点数,边数)：\n");
  scanf("%d,%d",&(G->n),&(G->e));  /*读入顶点数和边数*/
  printf("请输入顶点信息(输入格式为:顶点号<CR>)：\n");
  for (i=0;i<G->n;i++)               /*建立有 n 个顶点的顶点表*/
 {scanf("\n%c",&(G->adjlist[i].vertex));  /*读入顶点信息*/
  G->adjlist[i].firstedge=NULL;       /*顶点的边表头指针设为空*/
```

```
    }
    printf("请输入边的信息(输入格式为:i,j): \n");
    for (k=0;k<G->e;k++)                /*建立边表*/
   {scanf("\n%d,%d",&i,&j);              /*读入边<Vi,Vj>的顶点对应序号*/
    s=(EdgeNode*)malloc(sizeof(EdgeNode));  /*生成新边表结点 s*/
    s->adjvex=j;                        /*邻接点序号为 j*/
    s->next=G->adjlist[i].firstedge;       /*将新边表结点 s 插入到顶点 Vi 的边表头部*/
    G->adjlist[i].firstedge=s;
    }
}/*CreateALGraph*/
```

若无向图中有 n 个顶点、e 条边，则它的邻接表需 n 个头结点和 2e 个表结点。显然，在边稀疏(e<<n(n-1)/2)的情况下，用邻接表表示图比邻接矩阵节省存储空间，当和边相关的信息较多时更是如此。

在无向图的邻接表中，顶点 vi 的度恰为第 i 个链表中的结点数；而在有向图中，第 i 个链表中的结点个数只是顶点 vi 的出度，为求入度，必须遍历整个邻接表。在所有链表中其邻接点域的值为 i 的结点的个数是顶点 vi 的入度。有时，为了便于确定顶点的入度或以顶点 vi 为头的弧，可以建立一个有向图的逆邻接表，即对每个顶点 vi 建立一个链接以 vi 为头的弧的链表。

在建立邻接表或逆邻接表时，若输入的顶点信息即为顶点的编号，则建立邻接表的复杂度为 O（n+e），否则，需要通过查找才能得到顶点在图中位置，则时间复杂度为 O（n·e)。

在邻接表上容易找到任一顶点的第一个邻接点和下一个邻接点，但要判定任意两个顶点（vi 和 vj）之间是否有边或弧相连，则需搜索第 i 个或第 j 个链表，因此，不及邻接矩阵方便。

8.2.3 十字链表

十字链表（Orthogonal List）是有向图的一种存储方法，它实际上是邻接表与逆邻接表的结合，即把每一条边的边结点分别组织到以弧尾顶点为头结点的链表和以弧头顶点为头顶点的链表中。

在弧结点中有五个域：其中尾域(tailvex)和头(headvex)分别指示弧尾和弧头这两个顶点在图中的位置，链域 hlink 指向弧头相同的下一条弧，链域 tlink 指向弧尾相同的下一条弧，info 域指向该弧的相关信息。弧头相同的弧在同一链表上，弧尾相同的弧也在同一链表上。它们的头结点即为顶点结点，它由三个域组成：其中 vertex 域存储和顶点相关的信息，如顶点的名称等；firstin 和 firstout 为两个链域，分别指向以该顶点为弧头或弧尾的第一个弧结点。例如，图 8.14(a)中所示图的十字链表如图 8.14（b）所示。若将有向图的邻接矩阵看成是稀疏矩阵的话，则十字链表也可以看成是邻接矩阵的链表存储结构，在图的十字链表中，弧结点所在的链表非循环链表，结点之间相对位置自然形成，不一定按顶点序号有序，表头结点即顶点结点，它们之间而是顺序存储。

有向图的十字链表存储表示的形式描述如下：

```
#define  MAX_VERTEX_NUM 20
typedef struct ArcBox {
int tailvex,headvex;  /*该弧的尾和头顶点的位置*/
struct  ArcBox  * hlink, tlink; /分别为弧头相同和弧尾相财的弧的链域*/
InfoType   info;   /*该弧相关信息的指针*/
}ArcBox;
typedef struct VexNode {
VertexType vertex:
ArcBox  fisrin, firstout;  /*分别指向该顶点第一条入弧和出弧*/
}VexNode;
typedef struct {
VexNode xlist[MAX_VERTEX_NUM]; /*表头向量*/
int  vexnum,arcnum;  /*有向图的顶点数和弧数*/
 }OLGraph;
```

下面给出建立一个有向图的十字链表存储的算法。通过该算法，只要输入 n 个顶点的信息和 e 条弧的信息，便可建立该有向图的十字链表，其算法内容如下。

```
 void CreateDG(LOGraph **G)
 /*采用十字链表表示，构造有向图 G(G.kind=DG)*/
 {  scanf (&(*G->brcnum),&(*G->arcnum),&IncInfo);   /*IncInfo 为 0 则各弧不含其实信息*/
   for (i=0;i<*G->vexnum;++i)                    /*构造表头向量*/
```

```
    { scanf(&(G->xlist[i].vertex));                    /*输入顶点值*/
      *G->xlist[i].firstin=NulL;*G->xlist[i].firstout =NULL;  /*初始化指针*/
    }
   for(k=0;k<G.arcnum;++k)                          /*输入各弧并构造十字链表*/
    { scanf(&v1,&v2);                                /*输入一条弧的始点和终点*/
     i=LocateVex(*G,v1);  j=LocateVex(*G,v2);        /*确定 v1 和 v2 在 G 中位置*/
     p=(ArcBox*) malloc (sizeof(ArcBox));            /*假定有足够空间*/
     *p={ i,j,*G->xlist[j].fistin,*G->xlist[i].firstout,NULL} /*对弧结点赋值*/
                                          /*{tailvex,headvex,hlink,tlink,info}*/
     *G->xlist[j].fisrtin=*G->xlist[i].firstout=p;          /*完成在入弧和出弧链头的插入*/
     if (IncInfo) Input( p->info);                    /*若弧含有相关信息，则输入*/
    }
}/*CreateDG*/
```

在十字链表中既容易找到以为尾的弧，也容易找到以 vi 为头的弧，因而容易求得顶点的出度和入度（或需要，可在建立十字链表的同时求出）。

8.2.4 邻接多重表

邻接多重表(Adjacency Multilist)主要用于存储无向图。因为，如果用邻接表存储无向图，每条边的两个边结点分别在以该边所依附的两个顶点为头结点的链表中，这给图的某些操作带来不便。例如，对已访问过的边做标记，或者要删除图中某一条边等，都需要找到表示同一条边的两个结点。因此，在进行这一类操作的无向图的问题中采用邻接多重表作存储结构更为适宜。

邻接多重表的存储结构和十字链表类似，也是由顶点表和边表组成，每一条边用一个结点表示。

其中，顶点表由两个域组成，vertex 域存储和该顶点相关的信息 firstedge 域指示第一条依附于该顶点的边；边表结点由六个域组成，mark 为标记域，可用以标记该条边是否被搜索过；ivex 和 jvex 为该边依附的两个顶点在图中的位置；ilink 指向下一条依附于顶点 ivex 的边;jlink 指向下一条依附于顶点 jvex 的边，info 为指向和边相关的各种信息的指针域。

在邻接多重表中，所有依附于同一顶点的边串联在同一链表中，由于每条边依附于两个顶点，则每个边结点同时链接在两个链表中。可见，对无向图而言，其邻接多重表和邻接表的差别，仅仅在于同一条边在邻接表中用两个结点表示，而在邻接多重表中只有一个结点。因此，除了在边结点中增加一个标志域外，邻接多重表所需的存储量和邻接表相同。在邻接多重表上，各种基本操作的实现亦和邻接表相似。邻接多重表存储表示的形式描述如下：

```
#define MAX_VERTEX_NUM 20
typedef emnu{ unvisited,visited} VisitIf;
typedef struct EBox{
VisitIf mark:             /*访问标记*/
int ivex,jvex;             /*该边依附的两个顶点的位置*/
struct EBox ilink, jlink;   /*分别指向依附这两个顶点的下一条边*/
InfoType  info;          /*该边信息指针*/
}EBox;
typedef struct VexBox{
VertexType data;
EBox  fistedge;          /*指向第一条依附该顶点的边*/
}VexBox;
typedef struct{
VexBox adjmulist[MAX_VERTEX_NUM];
int vexnum,edgenum;      /*无向图的当前顶点数和边数*/
}AMLGraph;
```

8．3 图的遍历

图的遍历是指从图中的任一顶点出发，对图中的所有顶点访问一次且只访问一次。图的遍历操作和树的遍历操作功能相似。

由于图结构本身的复杂性，所以图的遍历操作也较复杂，主要表现在以下四个方面：

① 在图结构中，没有一个“自然”的首结点，图中任意一个顶点都可作为第一个被访问的结点。

② 在非连通图中，从一个顶点出发，只能够访问它所在的连通分量上的所有顶点，因此，还需考虑如何选取下一个出发点以访问图中其余的连通分量。

③ 在图结构中，如果有回路存在，那么一个顶点被访问之后，有可能沿回路又回到该顶点。

④ 在图结构中，一个顶点可以和其它多个顶点相连，当这样的顶点访问过后，存在如何选取下一个要访问的顶点的问题。

图的遍历通常有深度优先搜索和广度优先搜索两种方式，下面分别介绍。

8.3.1 深度优先搜索

深度优先搜索（Depth_Fisrst Search）遍历类似于树的先根遍历，是树的先根遍历的推广。

假设初始状态是图中所有顶点未曾被访问，则深度优先搜索可从图中某个顶点发 v 出发，访问此顶点，然后依次从 v 的未被访问的邻接点出发深度优先遍历图，直至图中所有和 v 有路径相通的顶点都被访问到；若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

显然，这是一个递归的过程。为了在遍历过程中便于区分顶点是否已被访问，需附设访问标志数组 visited[0:n-1], ，其初值为 FALSE ，一旦某个顶点被访问，则其相应的分量置为 TRUE。

```
void DFS(Graph G,int v )
{ /*从第 v 个顶点出发递归地深度优先遍历图 G*/
 visited[v]=TRUE;VisitFunc(v);              /*访问第 v 个顶点*/
 for(w=FisrAdjVex(G,v);w; w=NextAdjVex(G,v,w))
 if (!visited[w]) DFS(G,w);                  /*对 v 的尚未访问的邻接顶点 w 递归调用 DFS*/
}
```

算法 8.5 和算法 8.6 给出了对以邻接表为存储结构的整个图 G 进行深度优先遍历实现的 C 语言描述。

```
void DFSTraverseAL(ALGraph *G)
{/*深度优先遍历以邻接表存储的图 G*/
 int i;
 for (i=0;i<G->n;i++)
    visited[i]=FALSE;                  /*标志向量初始化*/
 for (i=0;i<G->n;i++)
    if (!visited[i]) DFSAL(G,i);           /*vi 未访问过，从 vi 开始 DFS 搜索*/
}/*DFSTraveseAL*/
```

算法 8.5

```
void DFSAL(ALGraph *G,int i)
{/*以 Vi 为出发点对邻接表存储的图 G 进行 DFS 搜索*/
 EdgeNode *p;
 printf("visit vertex:V%c\n",G->adjlist[i].vertex);/*访问顶点 Vi*/
 visited[i]=TRUE;              /*标记 Vi 已访问*/
 p=G->adjlist[i].firstedge;        /*取 Vi 边表的头指针*/
 while(p)                      /*依次搜索 Vi 的邻接点 Vj， j=p->adjva*/
  {if (!visited[p->adjvex])         /*若 Vj 尚未访问，则以 Vj 为出发点向纵深搜索*/
      DFSAL(G,p->adjvex);
   p=p->next;                    /*找 Vi 的下一个邻接点*/
     }
}/*DFSAL*/
```

分析上述算法，在遍历时，对图中每个顶点至多调用一次 DFS 函数，因为一旦某个顶点被标志成已被访问，就不再从它出发进行搜索。因此，遍历图的过程实质上是对每个顶点查找其邻接点的过程。其耗费的时间则取决于所采用的存储结构。当用二维数组表示邻接矩阵图的存储结构时，查找每个顶点的邻接点所需时间为 $O(n^2)$ ，其中 n 为图中顶点数。而当以邻接表作图的存储结构时，找邻接点所需时间为 O(e)，其中 e 为无向图中边的数或有向图中弧的数。由此，当以邻接表作存储结构时，深度优先搜索遍历图的时间复杂度为 O(n+e) 。

8.3.2 广度优先搜索

广度优先搜索（Breadth_First Search） 遍历类似于树的按层次遍历的过程。

假设从图中某顶点 v 出发，在访问了 v 之后依次访问 v 的各个未曾访问过和邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问，直至图中所有已被访问的顶点的邻接点都被访问到。若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访

问到为止。换句话说，广度优先搜索遍历图的过程中以 v 为起始点，由近至远，依次访问和 v 有路径相通且路径长度为 1,2,…的顶点。

和深度优先搜索类似，在遍历的过程中也需要一个访问标 志数组。并且，为了顺次访问路径长度为 2、3、…的顶点，需附设队列以存储已被访问的路径长度为 1、2、… 的顶点。

从图的某一点 v 出发，递归地进行广度优先遍历的过程如算法 8.7 所示。

```
void  BFSTraverse(Graph G, Status(*Visit)(int v))
 {/*按广度优先非递归遍历图 G。使用辅助队列 Q 和访问标志数组 visited*/
 for (v=0;v<G,vexnum;++v)
 visited[v]=FALSE
 InitQueue(Q);                    /*置空的国债队列 Q*/
  if (!visited[v])                    /*v 尚未访问*/
   {EnQucue(Q,v);   /*v 入队列*/
    while (!QueueEmpty(Q))
      { DeQueue(Q,u);             /*队头元素出队并置为 u*/
       visited[u]=TRUE; visit(u);   /*访问 u*/
       for(w=FistAdjVex(G,u); w; w=NextAdjVex(G,u,w))
         if (!visited[w]) EnQueue(Q,w);   /*u 的尚未访问的邻接顶点 w 入队列 Q*/
       }
    }
 }/*BFSTraverse*/
```

算法 8.7

算法 8.8 和算法 8.9 给出了对以邻接矩阵为存储结构的整个图 G 进行深度优先遍历实现的 C 语言描述。

```
 void BFSTraverseAL(MGraph *G)
 {/*广度优先遍历以邻接矩阵存储的图 G*/
  int i;
  for (i=0;i<G->n;i++)
     visited[i]=FALSE;               /*标志向量初始化*/
  for (i=0;i<G->n;i++)
     if (!visited[i]) BFSM(G,i);         /* vi 未访问过，从 vi 开始 BFS 搜索*/
 }/*BFSTraverseAL*/
```

算法 8.8

```
 void BFSM(MGraph *G,int k)
 {/*以 Vi 为出发点，对邻接矩阵存储的图 G 进行 BFS 搜索*/
  int i,j;
  CirQueue Q;
  InitQueue(&Q);
  printf("visit vertex:V%c\n",G->vexs[k]);   /*访问原点 Vk*/
  visited[k]=TRUE;
  EnQueue(&Q,k);                          /*原点 Vk 入队列*/
  while (!QueueEmpty(&Q))
     {i=DeQueue(&Q);                         /*Vi 出队列*/
      for (j=0;j<G->n;j++)                      /*依次搜索 Vi 的邻接点 Vj*/
      if (G->edges[i][j]==1 && !visited[j]) /*若 Vj 未访问*/
       {printf("visit vertex:V%c\n",G->vexs[j]);  /*访问 Vj */
           visited[j]=TRUE;
        EnQueue(&Q,j);                            /*访问过的 Vj 入队列*/
       }
```

```
    }
}/*BFSM*/
```

算法 8.9

分析上述算法，每个顶点至多进一次队列。遍历图的过程实质是通过边或弧找邻接点的过程，因此广度优先搜索遍历图的时间复杂度和深度优先搜索遍历相同，两者不同之处仅仅在于对顶点访问的顺序不同。

8.4 图的连通性

8.4.1 无向图的连通性

在对无向图进行遍历时，对于连通图，仅需从图中任一顶点出发，进行深度优先搜索或广度优先搜索，便可访问到图中所有顶点。对非连通图，则需从多个顶点出发进行搜索，而每一次从一个新的起始点出发进行搜索过程中得到的顶点访问序列恰为其各个连通分量中的顶点集。

8.4.2 有向图的连通性

深度优先搜索是求有向图的强连通分量的一个有效方法。假设以十字链表作有向图的存储结构，则求强连通分量的步骤如下：

（1）在有向图 G 上，从某个顶点出发沿以该顶点为尾的弧进行深度优先搜索遍历，并按其所有邻接点的搜索都完成(即退出 DFS 函数)的顺序将顶点排列起来。此时需对 8.3.1 中的算法作如下两点修改：（a）在进入 DFSTraverseAL 函数时首先进行计数变量的初始化，即在入口处加上 count=0 的语句；（b）在退出 函数之前将完成搜索的顶点号记录在另一个辅助数组 finished[vexnum] 中，即在函数 DFSAL 结束之前加上 finished[++count]=v 的语句。

（2）在有向图 G 上，从最后完成搜索的顶点（即 finished[vexnum-1]中的顶点）出发，沿着以该顶点为头的弧作逆向的深度搜索遍历，若此次遍历不能访问到有向图中所有顶点，则从余下的顶点中最后完成搜索的那个顶点出发，继续作逆向的深度优先搜索遍历，依次类推，直至有向图中所有顶点都被访问到为止。此时调用 DFSTraverseAL 时需作如下修改：函数中第二个循环语句的边界条件应改为 v 从 finished[vexnum-1]至 finished[0]。

由此，每一次调用 DFSAL 作逆向深度优先遍历所访问到的顶点集便是有向图 G 中一个强连通分量的顶点集。

例如图 8.14 (a) 所示的有向图，假设从顶点 v1 出发作深度优先搜索遍历，得到 finished 数组中的顶点号为(1,3,2,0) ；则再从顶点 v1 出发作逆向的深度优先搜索遍历，得到两个顶点集{ v1, v3, v4}和{ v2}，这就是该有向图的两个强连通分量的顶点集。

上述求强连通分量的第二步，其实质为：

（1）构造一个有向图 Gr，设 G=(V,{A})，则 Gr=(Vr，{Ar})对于所有< vi,,vj>∈A，必有< vj, vi >∈Ar。即 Gr 中拥有和 G 方向相反的弧；

（2）在有向图 Gr 上，从顶点 finished[vexnum-1] 出发作深度优先遍历。可以证明，在 Gr 上所得深度优先生成森林中每一棵树的顶点集即为 G 的强连通分量的顶点集。

显然，利用遍历求强连通分量的时间复杂度亦和遍历相同。

8.4.3 生成树和生成森林

在这一小节里，我们将给出通过对图的遍历，得到图的生成树或生成森林的算法。

设 E(G)为连通图 G 中所有边的集合，则从图中任一顶点出发遍历图时，必定将 E(G)分成两个集合 T(G)和 B(G)，其中 T(G)是遍历图过程中历经的边的集合；B(G)是剩余的边的集合。显然，T(G)和图 G 中所有顶点一起构成连通图 G 的极小连通子图。按照 8.1.2 节的定义，它是连通图的一棵生成树，并且由深度优先搜索得到的为深度优先生成树；由广度优先搜索得到的为广度优先生成树。

对于非连通图，通过这样的遍历，将得到的是生成森林。

假设以孩子兄弟链表作生成森林的存储结构，则算法 8.10 生成非连通图的深度优先生成森林，其中 DFSTree 函数如算法 8.11 所示。显然，算法 8.10 的时间复杂度和遍历相同。

```
void DESForest(Graph G, CSTree *T)
{ /*建立无向图 G 的深度优先生成森林的孩子兄弟链表 T*/
  T=NULL;
  for (v=0;v<G.vexnum;++v)
    if (!visited[v]=FALSE;
  for(v=0;v<G.vexnum;++v)
    if (!visited[v])                          /*顶点 v 为新的生成树的根结点*/
      { p=(CSTree)malloc(sixeof(CSNode)); /*分配根结点*/
       p={GetVex(G,v).NULL,NULL};      /*给根结点赋值*/
        if (!T)
```

```
            (*T)=p;                          /*T 是第一棵生成树的根*/
        else  q->nextsibling=p;                 /*前一棵的根的兄弟是其它生成树的根*/
        q=p;                                 /*q 指示当前生成树的根*/
        DFSTree(G,v,&p);                      /*建立以 p 为根的生成树*/
       }
}
```

算法 8.10

```
void  DFSTree(Graph G,int v ,CSTree *T)
{/*从第 v 个顶点出发深度优先遍历图 G，建立以*T 为根的生成树*/
 visited[v]=TRUE;
 first=TRUE;
 for(w=FirstAdjVex(G,v);  w;  w=NextAdjVex(G,v,w))
   if(!visited[w])
    { p=(CSTree)malloc(sizeof)CSNode));        /*分配孩子结点*/
      *p={GetVex(G,w),NULL,NULL};
      if (first)              /*w 是 v 的第一个未被访问的邻接顶点，作为根的左孩子结点*/
       { T->lchild=p;
         first=FALSE;
       }
      else { /*w 是 v 的其它未被访问的邻接顶点，作为上一邻接顶点的右兄弟*/
           q->nextsibling=p;
           }
      q=p;
      DFSTree(G,w,&q);    /*从第 w 个顶点出发深度优先遍历图 G，建立生成子树*q*/
     }
}
```

8.4.4 关节点和重连通分量

假若在删去顶点 v 以及和 v 相关联的各边之后，将图的一个连通分量分割成两个或两个以上的连通分量，则称顶点 v 为该图的一个关节点(articulation point) 。一个没有关节点的连通图称为重连通图(biconnected graph) 。在重连通图上，任意一对顶点之间至少存在两条路径，则在删去某个顶点以及依附于该顶点的各边时也不破坏图的连通性。若在连通图上至少删去 k 个顶点才能破坏图的连通性，则称此图的连通度为 k。关节点和重连通图在实际中较多应用。显然，一个表示通信网络的图的连通度越高，其系统越可靠，无论是哪一个站点出现故障或遭到外界破坏，都不影响系统的正常工作；又如，一个航空网若是重连通的，则当某条航线因天气等某种原因关闭时，旅客仍可从别的航线绕道而行；再如，若将大规模的集成电路的关键线路设计成重连通的话，则在某些元件失效的情况下，整个片子的功能不受影响，反之，在战争中，若要摧毁敌方的运输线，仅需破坏其运输网中的关节点即可。

利用深度优先搜索便可求得图的关节点，并由此可判别图是否是重连通的。

图 8.21 (b)所示为从顶点 A 出发深优先生成树，图中实线表示树边，虚线表示回边（即不在生成树上的边）。对树中任一顶点 v 而言，其孩子结点为在它之后搜索到的邻接点，而其双亲结点和由回边连接的祖先结点是在它之前搜索到的邻接点。由深度优先生成树可得出两类关节点的特性：

（1）若生成树的根有两棵或两棵以上的子树，则此根顶点必为关节点。因为图中不存在联结不同子树中顶点的边，因此，若删去根顶点，生成树便变成生成森林。如图 8.21(b) 中的顶点 A。

（2）若生成树中某个非叶子顶点 v，其某棵子树的根和子树中的其它结点均没有指向 v 的祖先的回边，则 v 为关节点。因为，若删去 v，则其子树和图的其它部分被分割开来。如图 8.21(b)中的顶点 B 和 G 。

若对图 Graph＝(V，{Edge}) 重新定义遍历时的访问函数 visited，并引入一个新的函数 low，则由一次深度优先遍历便可求得连通图中存在的所有关节点。

定义 visited[v]为深度优先搜索遍历连通图时访问顶点 v 的次序号；定义：

若对于某个顶点 v，存在孩子结点 w 且 low[w]≧visited[v]，则该顶点 v 必为关节点。因为当 w 是 v 的孩子结点时，low[w]≧visited[v]，表明 w 及其子孙均无指向 v 的祖先的回边。

由定义可知,visited[v]值即为v在深度优先生成树的前序序列的序号,只需将DFS函数中头两个语句改为visited[v0]=++count（在DFSTraverse中设初值count=1）即可；low[v]可由后序遍历深度优先生成树求得，而v在后序序列中的次序和遍历时退出DFS函数的次序相同，由此修改深度优先搜索遍历的算法便可得到求关节点的算法（见算法8.12和算法8.13)。

```
void FindArticul(ALGraph G)
{ /*连通图G以邻接表作存储结构，查找并输出G上全部关节点*/
  count=1;                          /*全局变量count用于对访问计数*/
  visited[0]=1;                      /*设定邻接表上0号顶点为生成树的根*/
  for(i=1;i<G.vexnum;++i)            /*其余顶点尚未访问*/
    visited[i]=0;
  p=G.adjlist[0].first;
  v=p->adjvex;
  DFSArticul(g,v);                   /*从顶点v出发深度优先查找关节点*/
  if(count<G.vexnum)                /*生成树的根至少有两棵子树*/
   {printf(0,G.adjlist[0].vertex);         /*根是关节点，输出*/
    while(p->next)
      { p=p->next;
       v=p->adjvex;
       if(visited[v]==0) DFSArticul(g,v);
      }
  }
} /*FindArticul*/
```

算法 8.12

```
void DFSArticul(ALGraph G,int v0)
/*从顶点v0出发深度优先遍历图G，查找并输出关节点*/
{ visited[v0]=min=++count;                  /*v0是第count个访问的顶点*/
  for(p=G.adjlist[v0].firstedge; p; p=p->next;)   /*对v0的每个邻接点检查*/
    { w=p->adjvex;                         /*w为v0的邻接点*/
      if(visited[w]==0)                     /*若w未曾访问，则w为v0的孩子*/
       { DFSArticul(G,w);                  /*返回前求得low[w]*/
         if(low[w]<min)min=low[w];
         if(low[w]>=visited[v0]) printf(v0,G.adjlist[v0].vertex);  /*输出关节点*/
        }
       else if(visited[w]<min) min=visited[w];  /*w已访问，w是v0在生成树上的祖先*/
     }
  low[v0]=min;
}
```

8.5 最小生成树

8.5.1 最小生成树的基本概念

由生成树的定义可知，无向连通图的生成树不是唯一的。连通图的一次遍历所经过的边的集合及图中所有顶点的集合就构成了该图的一棵生成树，对连通图的不同遍历，就可能得到不同的生成树。

可以证明，对于有n个顶点的无向连通图，无论其生成树的形态如何，所有生成树中都有且仅有n－1条边。

如果无向连通图是一个网，那么，它的所有生成树中必有一棵边的权值总和最小的生成树，我们称这棵生成树为最小生成树，简称为最小生成树。

8.5.2 构造最小生成树的Prim算法

假设G＝（V，E）为一网图，其中V为网图中所有顶点的集合，E为网图中所有带权边的集合。设置两个新的集合U和T，其中集合U用于存放G的最小生成树中的顶点，集合T存放G的最小生成树中的边。令集合U的初值为U＝{u1}（假设构造最小生成树时，从顶点u1出发），集合T的初值为T＝{}。Prim算法的思想是，从所有u∈U，v∈V－U的边中，选取具有最小权值的边（u，v），将

顶点 v 加入集合 U 中，将边（u，v）加入集合 T 中，如此不断重复，直到 U＝V 时，最小生成树构造完毕，这时集合 T 中包含了最小生成树的所有边。

Prim 算法可用下述过程描述，其中用 w_{uv} 表示顶点 u 与顶点 v 边上的权值。

（1）U＝{u1},T={};

（2）while (U≠V)do

(u，v)＝min{w_{uv}; u∈U，v∈V－U }

T＝T＋{(u，v)}

U＝U＋{v}

（3）结束。

为实现 Prim 算法，需设置两个辅助一维数组 lowcost 和 closevert，其中 lowcost 用来保存集合 V－U 中各顶点与集合 U 中各顶点构成的边中具有最小权值的边的权值；数组 closevertex 用来保存依附于该边的在集合 U 中的顶点。假设初始状态时，U＝{u1}(u1 为出发的顶点)，这时有 lowcost[0]=0，它表示顶点 u1 已加入集合 U 中，数组 lowcost 的其它各分量的值是顶点 u1 到其余各顶点所构成的直接边的权值。然后不断选取权值最小的边（ui，uk）(ui∈U，uk∈V－U)，每选取一条边，就将 lowcost（k）置为 0，表示顶点 uk 已加入集合 U 中。由于顶点 uk 从集合 V－U 进入集合 U 后，这两个集合的内容发生了变化，就需依据具体情况更新数组 lowcost 和 closevertex 中部分分量的内容。最后 closevertex 中即为所建立的最小生成树。

当无向网采用二维数组存储的邻接矩阵存储时，Prim 算法的 C 语言实现为：

```
void Prim（int gm[ ][MAXNODE], int n, int closevertex[ ])
{/*用 Prim 方法建立有 n 个顶点的邻接矩阵存储结构的网图 gm 的最小生成树*/
  /*从序号为 0 的顶点出发；建立的最小生成树存于数组 closevertex 中*/
  int lowcost[100],mincost;
  int i,j,k;
    for (i=1;i<n;i++)            /*初始化*/
     { lowcost[i]=gm[0][i];
      closevertex[i]=0;
     }
    lowcost[0]=0;                /*从序号为 0 的顶点出发生成最小生成树*/
    closevertex[0]=0;
    for (i=1;i<n;i++)              /*寻找当前最小权值的边的顶点*/
      {mincost=MAXCOST;      /*MAXCOST 为一个极大的常量值*/
       j=1;k=1;
       while (j<n)
         { if (lowcost[j]<mincost && lowcost[j]!=0)
            { mincost=lowcost[j];
              k=j;
            }
           j++;
         }
       printf(“顶点的序号＝%d 边的权值＝%d\n”,k,mincost);
       lowcost[k]=0;
       for (j=1;j<n;j++)           /*修改其它顶点的边的权值和最小生成树顶点序号*/
         if (gm[k][j]<lowcost[j])
          { lowcost[j]=gm[k][j];
            closevertex[j]=k;
          }
      }
    }
```

算法 8.14

图 8.24 给出了在用上述算法构造网图 8.23 (a)的最小生成树的过程中，数组 closevertex、lowcost 及集合 U，V－U 的变化情况，读者可进一步加深对 Prim 算法的了解。

在 Prim 算法中，第一个 for 循环的执行次数为 n－1，第二个 for 循环中又包括了一个 while 循环和一个 for 循环，执行次数为 $2(n-1)^2$，所以 Prim 算法的时间复杂度为 $O(n^2)$。

8.5.3 构造最小生成树的 Kruskal 算法

Kruskal 算法是一种按照网中边的权值递增的顺序构造最小生成树的方法。其基本思想是：设无向连通网为 G＝（V，E），令 G 的最小生成树为 T，其初态为 T＝（V，{}），即开始时，最小生成树 T 由图 G 中的 n 个顶点构成，顶点之间没有一条边，这样 T 中各顶点各自构成一个连通分量。然后，按照边的权值由小到大的顺序，考察 G 的边集 E 中的各条边。若被考察的边的两个顶点属于 T 的两个不同的连通分量，则将此边作为最小生成树的边加入到 T 中，同时把两个连通分量连接为一个连通分量；若被考察边的两个顶点属于同一个连通分量，则舍去此边，以免造成回路，如此下去，当 T 中的连通分量个数为 1 时，此连通分量便为 G 的一棵最小生成树。

对于图 8.23(a)所示的网，按照 Kruskal 方法构造最小生成树的过程如图 8.25 所示。在构造过程中，按照网中边的权值由小到大的顺序，不断选取当前未被选取的边集中权值最小的边。依据生成树的概念，n 个结点的生成树，有 n－1 条边，故反复上述过程，直到选取了 n－1 条边为止，就构成了一棵最小生成树。

下面介绍 Kruskal 算法的实现。

设置一个结构数组 Edges 存储网中所有的边，边的结构类型包括构成的顶点信息和边权值，定义如下：

```
#define MAXEDGE  〈图中的最大边数〉
typedef struct {
     elemtype v1;
      elemtype v2;
      int cost;
    } EdgeType;
EdgeType edges[MAXEDGE];
```

在结构数组 edges 中，每个分量 edges[i]代表网中的一条边，其中 edges[i].v1 和 edges[i].v2 表示该边的两个顶点，edges[i].cost 表示这条边的权值。为了方便选取当前权值最小的边，事先把数组 edges 中的各元素按照其 cost 域值由小到大的顺序排列。在对连通分量合并时，采用 7.5.2 节所介绍的集合的合并方法。对于有 n 个顶点的网，设置一个数组 father[n]，其初值为 father[i]=-1（i＝0，1，…，n－1），表示各个顶点在不同的连通分量上，然后，依次取出 edges 数组中的每条边的两个顶点，查找它们所属的连通分量，假设 vf1 和 vf2 为两顶点所在的树的根结点在 father 数组中的序号，若 vf1 不等于 vf2，表明这条边的两个顶点不属于同一分量，则将这条边作为最小生成树的边输出，并合并它们所属的两个连通分量。

下面用 C 语言实现 Kruskal 算法，其中函数 Find 的作用是寻找图中顶点所在树的根结点在数组 father 中的序号。需说明的是，在程序中将顶点的数据类型定义成整型，而在实际应用中，可依据实际需要来设定。

```
typedef int elemtype;
typedef struct {
     elemtype v1;
     elemtype v2;
     int cost;
    }EdgeType;
void Kruskal (EdgeType edges[ ], int n)
  /*用 Kruskal 方法构造有 n 个顶点的图 edges 的最小生成树*/
  { int father[MAXEDGE];
    int i,j,vf1,vf2;
    for (i=0;i<n;i++) father[i]=-1;
    i=0;j=0;
    while(i<MAXEDGE && j<n-1)
      { vf1=Find(father,edges[i].v1);
        vf2=Find(father,edges[i].v2);
        if (vf1!=vf2)
         { father[vf2]=vf1;
           j++;
           printf(“%3d%3d\n”,edges[i].v1,edges[i].v2);
         }
```

```
        i++;
      }
  }
```

算法 8.15

```
int Find (int father[ ], int v)
 /*寻找顶点 v 所在树的根结点*/
 { int t;
 t=v;
  while(father[t]>=0)
    t=father[t];
  return(t);
 }
```

算法 8.16

在 Kruskal 算法中，第二个 while 循环是影响时间效率的主要操作，其循环次数最多为 MAXEDGE 次数，其内部调用的 Find 函数的内部循环次数最多为 n，所以 Kruskal 算法的时间复杂度为 O（n•MAXEDGE)。

8.6 最短路径

在非网图中，最短路径是指两点之间经历的边数最少的路径。

8.6.1 从一个源点到其它各点的最短路径

本节先来讨论单源点的最短路径问题：给定带权有向图 G＝（V，E）和源点 v∈V，求从 v 到 G 中其余各顶点的最短路径。在下面的讨论中假设源点为 v0。

下面就介绍解决这一问题的算法。即由迪杰斯特拉（Dijkstra）提出的一个按路径长度递增的次序产生最短路径的算法。该算法的基本思想是：设置两个顶点的集合 S 和 T＝V－S，集合 S 中存放已找到最短路径的顶点，集合 T 存放当前还未找到最短路径的顶点。初始状态时，集合 S 中只包含源点 v0，然后不断从集合 T 中选取到顶点 v0 路径长度最短的顶点 u 加入到集合 S 中，集合 S 每加入一个新的顶点 u，都要修改顶点 v0 到集合 T 中剩余顶点的最短路径长度值，集合 T 中各顶点新的最短路径长度值为原来的最短路径长度值与顶点 u 的最短路径长度值加上 u 到该顶点的路径长度值中的较小值。此过程不断重复，直到集合 T 的顶点全部加入到 S 中为止。

Dijkstra 算法的正确性可以用反证法加以证明。假设下一条最短路径的终点为 x，那么，该路径必然或者是弧（v0，x)，或者是中间只经过集合 S 中的顶点而到达顶点 x 的路径。因为假若此路径上除 x 之外有一个或一个以上的顶点不在集合 S 中，那么必然存在另外的终点不在 S 中而路径长度比此路径还短的路径，这与我们按路径长度递增的顺序产生最短路径的前提相矛盾，所以此假设不成立。

下面介绍 Dijkstra 算法的实现。

首先，引进一个辅助向量 D，它的每个分量 D[i] 表示当前所找到的从始点 v 到每个终点 vi 的最短路径的长度。它的初态为：若从 v 到 vi 有弧，则 D[i]为弧上的权值；否则置 D[i]为∞。显然，长度为：

$$D[j]=Min\{D[i] \mid vi \in V\}$$

的路径就是从 v 出发的长度最短的一条最短路径。此路径为（v ，vj)。

那么，下一条长度次短的最短是哪一条呢？假设该次短路径的终点是 vk，则可想而知，这条路径或者是（v, vk)，或者是（v, vj, vk)。它的长度或者是从 v 到 vk 的弧上的权值，或者是 D[j]和从 vj 到 vk 的弧上的权值之和。

依据前面介绍的算法思想，在一般情况下，下一条长度次短的最短路径的长度必是：

$$D[j]=Min\{D[i] \mid vi \in V-S\}$$

其中，D[i] 或者弧（v, vi）上的权值，或者是 D[k](vk∈S 和弧（vk, vi）上的权值之和。

根据以上分析，可以得到如下描述的算法：

（1）假设用带权的邻接矩阵 edges 来表示带权有向图，edges[i][j] 表示弧〈vi, vj〉上的权值 。若〈vi, vj〉不存在，则置 edges[i][j]为∞（在计算机上可用允许的最大值代替)。S 为已找到从 v 出发的最短路径的终点的集合，它的初始状态为空集。那么，从 v 出发到图上其余各顶点（终点）vi 可能达到最短路径长度的初值为：

$$D[i]= edges[Locate\ Vex(G,v)][i] \quad vi \in V$$

（2）选择 vj，使得

$$D[j]=Min\{D[i] \mid vi \in V-S\}$$

vj 就是当前求得的一条从 v 出发的最短路径的终点。令

S＝S∪{j}

（3）修改从 v 出发到集合 V-S 上任一顶点 vk 可达的最短路径长度。如果

D[j]+ edges[j][k]<D[k]

则修改 D[k]为

D[k]=D[j]+ edges[j][k]

重复操作（2)、(3）共 n-1 次。由此求得从 v 到图上其余各顶点的最短路径是依路径长度递增的序列。

算法 8.17 为用 C 语言描述的 Dijkstra 算法。

```
void ShortestPath_1(Mgraph G,int v0,PathMatrix *p, ShortPathTable *D)
{ /*用 Dijkstra 算法求有向网 G 的 v0 顶点到其余顶点 v 的最短路径 P[v]及其路径长度 D[v]*/
  /*若 P[v][w]为 TRUE，则 w 是从 v0 到 v 当前求得最短路径上的顶点*/
  /*final[v] 为 TRUE 当且仅当 v∈S, ，即已经求得从 v0 到 v 的最短路径*/
  /*常量 INFINITY 为边上权值可能的最大值*/
  for (v=0;v<G.vexnum;++v)
     {fianl[v]=FALSE; D[v]=G.edges[v0][v];
      for (w=0; w<G.vexnum; ++w)  P[v][w]=FALSE;  /*设空路径*/
      if (D[v]<INFINITY) {P[v][v0]=TRUE; P[v][w]=TRUE;}
     }
  D[v0]=0; final[v0]=TRUE;          /*初始化，v0 顶点属于 S 集*/
   /*开始主循环，每次求得 v0 到某个 v 顶点的最短路径，并加 v 到集*/
  for(i=1; i<G.vexnum; ++i)           /*其余 G.vexnum-1 个顶点*/
    {min=INFINITY;                 /*min 为当前所知离 v0 顶点的最近距离*/
     for (w=0;w<G.vexnum;++w)
       if (!final[w])                  /*w 顶点在 V－S 中*/
        if (D[w]<min) {v=w; min=D[w];}
     final[v]=TRUE                                /*离 v0 顶点最近的 v 加入 S 集合*/
     for(w=0;w>G.vexnum;++w)                        /*更新当前最短路径*/
       if (!final[w]&&(min+G.edges[v][w]<D[w]))      /*修改 D[w]和 P[w],w∈V-S*/
        { D[w]=min+G.edges[v][w];
          P[w]=P[v]; P[w][v]=TRUE;                /*P[w]=P[v]+P[w]*/
        }
     }
}/*ShortestPath._1*/
```

若对 G8 施行 Dijkstra 算法，则所得从 v0 到其余各顶点的最短路径，以及运算过程中 D 向量的变化状况，如下所示：

下面分析一下这个算法的运行时间。第一个 for 循环的时间复杂度是 O(n)，第二个 for 循环共进行 n-1 次，每次执行的时间是 O(n)。所以总是的时间复杂度是 $O(n^2)$。如果用带权的邻接表作为有向图的存储结构，则虽然修改 D 的时间可以减少，但由于在 D 向量中选择最小的分量的时间不变，所以总的时间仍为 $O(n^2)$。

如果只希望找到从源点到某一个特定的终点的最短路径，但是，从上面我们求最短路径的原理来看，这个问题和求源点到其它所有顶点的最短路径一样复杂，其时间复杂度也是 $O(n^2)$。

8.6.2 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，重复招待迪杰斯特拉算法 次。这样，便可求得每一结顶点的最短路径。总的执行时间为 $O(n^3)$。

这里要介绍由弗洛伊德(Floyd)提出的另一个算法。这个算法的时间复杂度也是 $O(n^3)$,但形式上简单些。

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：

假设求从顶点 vi 到 vj 的最短路径。如果从 vi 到 vi 有弧，则从 vi 到 vj 存在一条长度为 edges[i][j]的路径，该路径不一定是最短路径，尚需进行 n 次试探。首先考虑路径（vi, v0, vj）是否存在（即判别弧（vi, v0）和(v0, vj)是否存在)。如果存在，则比较（vi, vj）和（vi, v0, vj）的路径长度取长度较短者为从 vi 到 vj 的中间顶点的序号不大于 0 的最短路径。假如在路径上再增加一个顶点 v1，也就是说，如果 （vi, …, v1）和（v1, …, vj）分别是当前找到的中间顶点的序号不大于 0 的最短路径，那么（vi, …, v1, … , vj） 就有可能是从 vi 到 vj 的中间顶点的序号不大于 1 的最短路径。将它和已经得到的从 vi 到 vj 中间顶

点序号不大于 0 的最短路径相比较，从中选出中间顶点的序号不大于 1 的最短路径之后，再增加一个顶点 v2，继续进行试探。依次类推。在一般情况下，若（vi, …, vk）和（vk, …, vj）分别是从 vi 到 vk 和从 vk 到 vj 的中间顶点的序号不大于 k-1 的最短路径，则将（vi, …, vk, …, vj）和已经得到的从 vi 到 vj 且中间顶点序号不大于 k-1 的最短路径相比较，其长度较短者便是从 vi 到 vj 的中间顶点的序号不大于 k 的最短路径。这样，在经过 n 次比较后，最后求得的必是从 vi 到 vj 的最短路径。

按此方法，可以同时求得各对顶点间的最短路径。

现定义一个 n 阶方阵序列。

$D^{(-1)}$, $D^{(0)}$, $D^{(1)}$, …, $D^{(k)}$, $D^{(n-1)}$

其中

$D^{(-1)}[i][j]=edges[i][j]$

$D^{(k)}[i][j]=Min\{D^{(k-1)}[i][j], D^{(k-1)}[i][k]+D^{(k-1)}[k][j]\}$ $0 \leqq k \leqq n-1$

从上述计算公式可见，$D^{(1)}[i][j]$是从 vi 到 vj 的中间顶点的序号不大于 1 的最短路径的长度；$D^{(k)}[i][j]$ 是从 vi 到 vj 的中间顶点的个数不大于 k 的最短路径的长度；$D^{(n-1)}[i][j]$ 就是从 vi 到 vj 的最短路径的长度。

由此得到求任意两顶点间的最短路径的算法 8.18。

```
void ShortestPath_2 (Mgraph G, PathMatrix *P[],DistancMatrix *D)
{/*用 Floyd 算法求有向网 G 中各对顶点 v 和 w 之间的最短路径 P[v][w]及其带权长度 D[v][w]。*/
 /*若 P[v][w][u]为 TRUE，则 u 是从 v 到 w 当前求得的最短路径上的顶点。*/
 for(v=0;v<G.vexnum;++v)                /*各对顶点之间初始已知路径及距离*/
  for(w=0;w<G,vexnum;++w)
   { D[v][w]=G.arcs[v][w];
     for(u=0;u<G,vexnum;++u)  P[v][w][u]=FALSE;
     if (D[v][w]<INFINITY)              /*从 v 到 w 有直接路径*/
      { P[v][w][v]=TRUE;
      }
   }
 for(u=0; u<G.vexnum; ++u)
   for(v=0; v<G.vexnum; ++v)
     for(w=0;w<G.vexnum;++w)
       if (D[v][u]+D[u][w]<D[v][w])      /*从 v 经 u 到 w 的一条路径更短*/
        {D[v][w]=D[v][u]+D[u][w];
         for(i=0;i<G.vexnum;++i)
         P[v][w][i]=P[v][u][i]||P[u][w][i];
        }
}/* ShortestPath_2*/
```

8.7 有向无环图及其应用

8.7.1 有向无环图的概念

一个无环的有向图称做有向无环图（directed acycline praph)。简称 DAG 图。 检查一个有向图是否存在环要比无向图复杂。对于无向图来说，若深度优先遍历过程中遇到回边（即指向已访问过的顶点的边），则必定存在环；而对于有向图来说，这条回边有可能是指向深度优先生成森林中另一棵生成树上顶点的弧。但是，如果从有向图上某个顶点 v 出发的遍历，在 dfs(v)结束之前出现一条从顶点 u 到顶点 v 的回边，由于 u 在生成树上是 v 的子孙，则有向图必定存在包含顶点 v 和 u 的环。

有向无环图是描述一项工程或系统的进行过程的有效工具。除最简单的情况之外，几乎所有的工程（project）都可分为若干个称作活动（activity）的子工程，而这些子工程之间，通常受着一定条件的约束，如其中某些子工程的开始必须在另一些子工程完成之后。对整个工程和系统，人们关心的是两个方面的问题：一是工程能否顺利进行：二是估算整个工程完成所必须的最短时间。以下两小节将详细介绍这样两个问题是如何通过对有向图进行拓扑排序和关键路径操作来解决的。

8.7.2 AOV 网与拓扑排序

1. AOV 网(Activity on vertex network)

所有的工程或者某种流程可以分为若干个小的工程或阶段，这些小的工程或阶段就称为活动。若以图中的顶点来表示活动，有向边表示活动之间的优先关系，则这样活动在顶点上的有向图称为 AOV 网。在 AOV 网中，若从顶点 i 到顶点 j 之间存在一条有向路径，称顶点 i 是顶点 j 的前驱，或者称顶点 j 是顶点 i 的后继。若<i,j>是图中的弧，则称顶点 i 是顶点 j 的直接前驱，顶点 j 是

顶点 i 的直接后驱。

AOV 网中的弧表示了活动之间存在的制约关系。

2．拓扑排序

首先复习一下离散数学中的偏序集合与全序集合两个概念。

若集合 A 中的二元关系 R 是自反的、非对称的和传递的，则 R 是 A 上的偏序关系。集合 A 与关系 R 一起称为一个偏序集合。

若 R 是集合 A 上的一个偏序关系，如果对每个 a、b∈A 必有 aRb 或 bRa ，则 R 是 A 上的全序关系。集合 A 与关系 R 一起称为一个全序集合。

AOV 网所代表的一项工程中活动的集合显然是一个偏序集合。为了保证该项工程得以顺利完成，必须保证 AOV 网中不出现回路；否则，意味着某项活动应以自身作为能否开展的先决条件，这是荒谬的。

测试 AOV 网是否具有回路（即是否是一个有向无环图）的方法，就是在 AOV 网的偏序集合下构造一个线性序列，该线性序列具有以下性质：

① 在 AOV 网中，若顶点 i 优先于顶点 j ，则在线性序列中顶点 i 仍然优先于顶点 j；

② 对于网中原来没有优先关系的顶点与顶点 ，如图 8.33 中的 C1 与 C13，在线性序列中也建立一个先后关系，或者顶点 i 优先于顶点 j ，或者顶点 j 优先于 i。

满足这样性质的线性序列称为拓扑有序序列。构造拓扑序列的过程称为拓扑排序。也可以说拓扑排序就是由某个集合上的一个偏序得到该集合上的一个全序的操作。

若某个 AOV 网中所有顶点都在它的拓扑序列中，则说明该 AOV 网不会存在回路，这时的拓扑序列集合是 AOV 网中所有活动的一个全序集合。以图 8.21 中的 AOV 网例，可以得到不止一个拓扑序列，C1、C12、C4、C13、C5、C2、C3、C9、C7、C10、C11、C6、C8 就是其中之一。显然，对于任何一项工程中各个活动的安排，必须按拓扑有序序列中的顺序进行才是可行的。

3．拓扑排序算法

对 AOV 网进行拓扑排序的方法和步骤是：

①从 AOV 网中选择一个没有前驱的顶点（该顶点的入度为 0）并且输出它；

②从网中删去该顶点，并且删去从该顶点发出的全部有向边；

③重复上述两步，直到剩余的网中不再存在没有前驱的顶点为止。

这样操作的结果有两种：一种是网中全部顶点都被输出，这说明网中不存在有向回路；另一种就是网中顶点未被全部输出，剩余的顶点均不前驱顶点，这说明网中存在有向回路。

为了实现上述算法，对 AOV 网采用邻接表存储方式，并且邻接表中顶点结点中增加一个记录顶点入度的数据域，即顶点结构设为：

其中，vertex、firstedge 的含义如前所述；count 为记录顶点入度的数据域。

顶点表结点结构的描述改为：

```
typedef struct vnode{              /*顶点表结点*/
        int   count                    /*存放顶点入度*/
      VertexType vertex;           /*顶点域*/
      EdgeNode  * firstedge;       /*边表头指针*/
     }VertexNode;
```

当然也可以不增设入度域，而另外设一个一维数组来存放每一个结点的入度。

算法中可设置了一个堆栈，凡是网中入度为 0 的顶点都将其入栈。为此，拓扑排序的算法步骤为：

①将没有前驱的顶点（count 域为 0）压入栈；

②从栈中退出栈顶元素输出，并把该顶点引出的所有有向边删去，即把它的各个邻接顶点的入度减 1；

③将新的入度为 0 的顶点再入堆栈；

④重复②～④，直到栈为空为止。此时或者是已经输出全部顶点，或者剩下的顶点中没有入度为 0 的顶点。

下面给出用 C 语言描述的拓扑排序算法的实现。

从上面的步骤可以看出，栈在这里的作用只是起到一个保存当前入度为零点的顶点，并使之处理有序。这种有序可以是后进先出，也可以是先进先出，故此也可用队列来辅助实现。在下面给出用 C 语言描述的拓扑排序的算法实现中，我们采用栈来存放当前未处理过的入度为零点的结点，但并不需要额外增设栈的空间，而是设一个栈顶位置的指针将当前所有未处理过的入度为零的结点连接起来，形成一个链式栈。

```
void Topo_Sort (AlGraph *G)
```

```
{/*对以带入度的邻接链表为存储结构的图 G，输出其一种拓扑序列*/
  int  top = -1;                    /* 栈顶指针初始化*/
  for (i=0; i<n; i++)                /* 依次将入度为 0 的顶点压入链式栈*/
    {  if ( G->adjlist[i]. Count = = 0)
         { G->adjlist[i].count = top;
           top = i;
         }
    }
   for (i=0;i<n;i++)
    { if (t0p= -1)
       {printf(“The network has a cycle”);
        return;
       }
       j=top;
       top=G->adjlist[top].count;       /* 从栈中退出一个顶点并输出*/
       printf(“% c”,G->adjlist[j].vertex);
       ptr=G->adjlist[j].firstedge;
       while (ptr!=null)
          { k=ptr->adjvex;
            G->adjlist[k].count--;        /*当前输出顶点邻接点的入度减 1*/
            if(G->adjlist[k].count= =0)  /*新的入度为 0 的顶点进栈*/
              {G->adjlist[k].count =top;
                top=k;
              }
            ptr=ptr->next;               /*找到下一个邻接点*/
          }
    }
}
```

算法 8.19

对一个具有 n 个顶点、e 条边的网来说，整个算法的时间复杂度 为 O（e+n)。

8.7.3 AOE 图与关键路径

1. AOE 网（Activity on edge network）

若在带权的有向图中，以顶点表示事件，以有向边表示活动，边上的权值表示活动的开销（如该活动持续的时间），则此带权的有向图称为 AOE 网。

如果用 AOE 网来表示一项工程，那么，仅仅考虑各个子工程之间的优先关系还不够，更多的是关心整个工程完成的最短时间是多少；哪些活动的延期将会影响整个工程的进度，而加速这些活动是否会提高整个工程的效率。因此，通常在 AOE 网中列出完成预定工程计划所需要进行的活动，每个活动计划完成的时间，要发生哪些事件以及这些事件与活动之间的关系，从而可以确定该项工程是否可行，估算工程完成的时间以及确定哪些活动是影响工程进度的关键。

AOE 网具有以下两个性质：

①只有在某顶点所代表的事件发生后，从该顶点出发的各有向边所代表的活动才能开始。

②只有在进入一某顶点的各有向边所代表的活动都已经结束，该顶点所代表的事件才能发生。

2. 关键路径

由于 AOE 网中的某些活动能够同时进行，故完成整个工程所必须花费的时间应该为源点到终点的最大路径长度（这里的路径长度是指该路径上的各个活动所需时间之和)。具有最大路径长度的路径称为关键路径。关键路径上的活动称为关键活动。关键路径长度是整个工程所需的最短工期。这就是说，要缩短整个工期，必须加快关键活动的进度。

利用 AOE 网进行工程管理时要需解决的主要问题是：

①计算完成整个工程的最短路径。

②确定关键路径，以找出哪些活动是影响工程进度的关键。

3．关键路径的确定

为了在 AOE 网中找出关键路径，需要定义几个参量，并且说明其计算方法。

（1）事件的最早发生时间 ve[k]

ve[k]是指从源点到顶点的最大路径长度代表的时间。这个时间决定了所有从顶点发出的有向边所代表的活动能够开工的最早时间。根据 AOE 网的性质，只有进入 vk 的所有活动< vj,vk>都结束时，vk 代表的事件才能发生；而活动< vj, vk>的最早结束时间为 ve[j]+dut(< vj, vk>)。所以计算 vk 发生的最早时间的方法如下：

$$\begin{cases} ve[1]=0 \\ ve[k]=Max\{ve[j]+dut(<vj, vk>)\} \quad <vj, vk>\in p[k] \end{cases} \qquad (8\text{-}1)$$

其中，p[k]表示所有到达 vk 的有向边的集合；dut(< vj, vk>)为有向边< vj,vk>上的权值。

（2）事件的最迟发生时间 vl[k]

vl[k]是指在不推迟整个工期的前提下，事件 vk 允许的最晚发生时间。设有向边< vk，vj>代表从 vk 出发的活动，为了不拖延整个工期,vk 发生的最迟时间必须保证不推迟从事件 vk 出发的所有活动< vk，vj>的终点 vj 的最迟时间 vl[j]。vl[k] 的计算方法如下：

$$\begin{cases} vl[n]=ve[n] \\ vl[k]=Min\{vl[j]-dut(<vk- vj>)\} \quad <vk, vj>\in s[k] \end{cases} \qquad (8\text{-}2)$$

其中，s[k]为所有从 vk 发出的有向边的集合。

（3）活动 ai 的最早开始时间 e[i]

若活动 ai 是由弧<vk,vj>表示，根据 AOE 网的性质，只有事件 vk 发生了，活动 ai 才能开始。也就是说，活动 ai 的最早开始时间应等于事件 vk 的最早发生时间。因此，有：

$$e[i]=ve[k] \qquad (8\text{-}3)$$

（4）活动 ai 的最晚开始时间 l[i]

活动 ai 的最晚开始时间指，在不推迟整个工程完成日期的前提下， 必须开始的最晚时间。若 由弧< vk,vj>>表示，则 ai 的最晚开始时间要保证事件 vj 的最迟发生时间不拖后。因此，应该有：

$$l[i]=vl[j]-dut(<vk,vj>) \qquad (8\text{-}4)$$

根据每个活动的最早开始时间 e[i]和最晚开始时间 l[i]就可判定该活动是否为关键活动，也就是那些 l[i]=e[i]的活动就是关键活动，而那些 l[i]>e[i]的活动则不是关键活动，l[i]-e[i]的值为活动的时间余量。关键活动确定之后，关键活动所在的路径就是关键路径。

由由上述方法得到求关键路径的算法步骤为：

（1）输入 e 条弧<j,k>,建立 AOE-网的存储结构；

（2）从源点 v_0出发，令 ve[0]＝0，按拓扑有序求其余各顶点的最早发生时间 ve[i](1≤i≤n-1)。如果得到的拓扑有序序列中顶点个数小于网中顶点数 n，则说明网中存在环，不能求关键路径，算法终止；否则执行步骤（3）。

（3）从汇点 v_n出发，令 vl[n-1]=ve[n-1],按逆拓扑有序求其余各顶点的最迟发生时间 vl[i](n-2≥i≥2)；

（4）根据各顶点的 ve 和 vl 值，求每条弧 s 的最早开始时间 e(s)和最迟开始时间 l（s)。若某条弧满足条件 e(s)=l(s),则为关键活动。

由该步骤得到的算法参看算法8.20和算法8.21。在算法8.20中,Stack 为栈的存储类型;引用的函数FindInDegree(G, indegree)用来求图 G 中各顶点的入度，并将所求的入度存放于一维数组 indegree 中。

```
int topologicalOrder(ALGraph G,Stack T)
{ /* 有向网 G 采用邻接表存储结构，求各顶点事件的最早发生时间 ve(全局变量)*/
  /* T 为拓扑序列顶点栈，S 为零入度顶点栈。*/
  /* 若 G 无回路，则用栈 T 返回 G 的一个拓扑序列，且函数值为 OK，否则为 ERROR。*/
  FindInDegree(G, indegree);                  /* 对各顶点求入度 indegree[0..vernum-1]*/
  InitStack(S);                               /*建零入度顶点栈 S*/
  count = 0;  ve[0..G.vexnum-1] = 0;          /*初始化 ve[ ]*/
  for (i=0; i<G.vexnum; i++)                  /*将初始时入度为 0 的顶点入栈*/
     {if (indegree[i]==0)  push(S,i); }
  while (! StackEmpty(S)) {
       Pop (S, j);  Push (T, j);  ++ count;      /* j 号顶点入 T 栈并计数*/
       for ( p=G.adjlist[j].firstedge; p; p=p->next)
```

```
        { k = p->adjvex;                          /*对 j 号顶点的每个邻接点的入度减 1*/
          if (- - indegree[k] = = 0)  Push(S, k);  /*若入度减为 0, 则入栈*/
          if (ve[j]+* (p->info)>ve[k])
            ve[k] = ve[j]+*(p->info);
        }
    }
    if (count<G. vexnum)  return 0;               /*该有向网有回路返回 0，否则返回 1*/
    else return 1;
} /* TopologicalOrder*/
```

算法 8.20

```
int Criticalpath(ALGraph G)
{ /* G 为有向网，输出 G 的各项关键活动。*/
  InitStack(T);                                  /*建立用于产生拓扑逆序的栈 T*/
  if (! TopologicalOrder (G,T) )   return 0;       /*该有向网有回路返回 0*/
  vl[0..G.vexnum-1] = ve [G.vexnum-1];           /* 初始化顶点事件的最迟发生时间*/
  while (! StackEmpty (T) )                      /*按拓扑逆序求各顶点的 vl 值*/
     for (Pop(T,j), p=G. adjlist[j].firstedge; p; p=p->next)
      { k=p->adjvex;  dut = * (p->info);
       if ( vl [k]-dut < vl [j] )  vl [j] = vl [k] - dut;
      }
  for ( j=0; j<G. vexnum; + +j)                  /*求 e、l 和关键活动*/
      for (p=G.adjlist [j].firstedge; p; p = p->next)
       { k = p->adjvex;   dut= * (p->indo);
        e = ve [j]; l = vl [k] - dut;
        tag = (e= =l) ? '*' :' ' ;
        printf ( j,k,dut,e,l,tag );                /*输出关键活动*/
       }
return 1;                                        /*求出关键活动后返回 1*/
} /*Criticalpath*/
```

第九章　查找

9.1 基本概念与术语

1.数据项（也称项或字段）

项是具有独立含义的标识单位，是数据不可分割的最小单位。项有名和值之分，项名是一个项的标识，用变量定义，而项值是它的一个可能取值。项具有一定的类型，依项的取值类型而定。

2.组合项

由若干项、组合项构成。

3.数据元素（记录）

数据元素是由若干项、组合项构成的数据单位，是在某一问题中作为整体进行考虑和处理的基本单位。数据元素有型和值之分，表中项名的集合，也即表头部分就是数据元素的类型；而一个学生对应的一行数据就是一个数据元素的值，表中全体学生即为数据元素的集合。

4.关键码

关键码是数据元素（记录）中某个项或组合项的值，用它可以标识一个数据元素（记录）。能唯一确定一个数据元素（记录）的关键码，称为主关键码；而不能唯一确定一个数据元素（记录）的关键码，称为次关键码。

5.查找表

是由具有同一类型（属性）的数据元素（记录）组成的集合。分为静态查找表和动态查找表两类。

静态查找表：仅对查找表进行查找操作，而不能改变的表；

动态查找表：对查找表除进行查找操作外，可能还要进行向表中插入数据元素，或删除表中数据元素的表。

6.查找

按给定的某个值 kx，在查找表中查找关键码为给定值 kx 的数据元素（记录）。

关键码是主关键码时：由于主关键码唯一，所以查找结果也是唯一的，一旦找到，查找成功，结束查找过程，并给出找到的数据元素（记录）的信息，或指示该数据元素（记录）的位置。要是整个表检测完，还没有找到，则查找失败，此时，查找结果应给出一个“空”记录或“空”指针。

关键码是次关键码时：需要查遍表中所有数据元素（记录），或在可以肯定查找失败时，才能结束查找过程。

7.数据元素类型说明

在手工绘制表格时，总是根据有多少数据项，每个数据项应留多大宽度来确定表的结构，即表头的定义。然后，再根据需要的行数，画出表来。在计算机中存储的表与手工绘制的类似，需要定义表的结构，并根据表的大小为表分配存储单元。

9.2 静态查找表

9.2.1 静态查找表结构

静态查找表是数据元素的线性表，可以是基于数组的顺序存储或以线性链表存储。

```
/* 顺序存储结构 */
        typedef  struct{
                ElemType  *elem;          /* 数组基址 */
                int        length;         /* 表长度 */
                }S_TBL;
/* 链式存储结构结点类型 */
        typedef  struct  NODE{
                ElemType  elem;           /* 结点的值域 */
                struct  NODE   *next;      /* 下一个结点指针域 */
                }NodeType;
```

9.2.2 顺序查找

顺序查找又称线性查找，是最基本的查找方法之一。其查找方法为：从表的一端开始，向另一端逐个按给定值 kx 与关键码进行比较，若找到，查找成功，并给出数据元素在表中的位置；若整个表检测完，仍未找到与 kx 相同的关键码，则查找失败，给出失败信息。

【算法 9.1】以顺序存储为例，数据元素从下标为 1 的数组单元开始存放，0 号单元留空。

```
int   s_search(S_TBL  tbl, KeyType  kx)
{   /*在表 tbl 中查找关键码为 kx 的数据元素，若找到返回该元素在数组中的下标，否则返回 0 */
     tbl.elem[0].key = kx; /* 存放监测，这样在从后向前查找失败时，不必判表是否检测完，*/
                           /* 从而达到算法统一*/
     for( i = tbl.length ; tbl.elem[i].key < > kx ; i-- );  /* 从标尾端向前找 */
     return  i;
}
```

查找不成功时，关键码的比较次数总是 n+1 次。

算法中的基本工作就是关键码的比较，因此，查找长度的量级就是查找算法的时间复杂度，其为 O(n)。

许多情况下，查找表中数据元素的查找概率是不相等的。为了提高查找效率，查找表需依据查找概率越高，比较次数越少；查找概率越低，比较次数就较多的原则来存储数据元素。

顺序查找缺点是当 n 很大时，平均查找长度较大，效率低；优点是对表中数据元素的存储没有要求。另外，对于线性链表，只能进行顺序查找。

9.2.3 有序表的折半查找

有序表即是表中数据元素按关键码升序或降序排列。

折半查找的思想为：在有序表中，取中间元素作为比较对象，若给定值与中间元素的关键码相等，则查找成功；若给定值小于中间元素的关键码，则在中间元素的左半区继续查找；若给定值大于中间元素的关键码，则在中间元素的右半区继续查找。不断重复上述查找过程，直到查找成功，或所查找的区域无数据元素，查找失败。

【步骤】

① low=1；high=length;　　// 设置初始区间

② 当 low>high 时，返回查找失败信息　　// 表空，查找失败

③ low≤high，mid=(low+high)/2;　　// 取中点

a. 若 kx<tbl.elem[mid].key，high=mid-1；转②　// 查找在左半区进行

b. 若 kx>tbl.elem[mid].key，low=mid+1；转②　// 查找在右半区进行

c. 若 kx=tbl.elem[mid].key，返回数据元素在表中位置　// 查找成功

【算法 9.2】

```
int  Binary_Search(S_TBL  tbl，KEY  kx)
{  /* 在表 tbl 中查找关键码为 kx 的数据元素，若找到返回该元素在表中的位置，否则，返回 0  */
     int   mid，flag=0；
     low=1；high=length；                /* ①设置初始区间 */
     while(low<=high)                    /* ②表空测试 */
     {    /* 非空，进行比较测试 */
          mid=(low+high)/2；             /* ③得到中点 */
          if(kx<tbl.elem[mid].key)  high=mid-1；      /* 调整到左半区 */
          else if(kx>tbl.elem[mid].key)   low=mid+1；       /* 调整到右半区 */
               else   { flag=mid；break；}    /* 查找成功，元素位置设置到 flag 中 */
     }
     return  flag；
}
```

折半查找的时间效率为 $O(\log_2 n)$。

9.3 动态查找表

9.3.1 二叉排序树

一.二叉排序树定义

二叉排序树（Binary Sort Tree）或者是一棵空树；或者是具有下列性质的二叉树：

⑴若左子树不空，则左子树上所有结点的值均小于根结点的值；若右子树不空，则　右子树上所有结点的值均大于根结点的值。

⑵ 左右子树也都是二叉排序树。

由图 9.4 可以看出，对二叉排序树进行中序遍历，便可得到一个按关键码有序的序列，因此，一个无序序列，可通过构一棵二叉排序树而成为有序序列。

二.二叉排序树查找过程

从其定义可见，二叉排序树的查找过程为：

① 若查找树为空，查找失败。

② 查找树非空，将给定值 kx 与查找树的根结点关键码比较。

③ 若相等，查找成功，结束查找过程，否则，

a. 当给 kx 小于根结点关键码，查找将在以左子女为根的子树上继续进行，转①

b. 当给 kx 大于根结点关键码，查找将在以右子女为根的子树上继续进行，转①

以二叉链表作为二叉排序树的存储结构，则查找过程算法程序描述如下：

```
typedef   struct  NODE
          {    ElemType       elem;     /*数据元素字段*/
               struct NODE    *lc,*rc;   /*左、右指针字段*/
          }NodeType;                /*二叉树结点类型*/
```

【算法 9.4】

```
int SearchElem(NodeType *t,NodeType **p,NodeType **q,KeyType kx)
{    /*在二叉排序树 t 上查找关键码为 kx 的元素，若找到，返回 1，且 q 指向该结点，p 指向其父结点；*/
     /*否则，返回 0，且 p 指向查找失败的最后一个结点*/
     int  flag=0;   *q=t;
     while(*q)      /*从根结点开始查找*/
```

```
    {    if(kx>(*q)->elem.key)          /*kx 大于当前结点*q 的元素关键码*/
         {    *p=*q;    *q=(*q)->rc;  }    /*将当前结点*q 的右子女置为新根*/
         else
         {    if(kx<(*q)->elem.key)          /*kx 小于当前结点*q 的元素关键码*/
              {    *p=*q;    *q=(*q)->lc;}        /*将当前结点*q 的左子女置为新根*/
              else      {flag=1;break;}          /*查找成功，返回*/
         }
    }/*while*/
    return flag;
}
```

三.二叉排序树插入操作和构造一棵二叉排序树

先讨论向二叉排序树中插入一个结点的过程：设待插入结点的关键码为 kx，为将其插入，先要在二叉排序树中进行查找，若查找成功，按二叉排序树定义，待插入结点已存在，不用插入；查找不成功时，则插入之。因此，新插入结点一定是作为叶子结点添加上去的。

构造一棵二叉排序树则是逐个插入结点的过程。

【算法 9.5】

```
int InsertNode (NodeType **t,KeyType kx)
{    /*在二叉排序树*t 上插入关键码为 kx 的结点*/
NodeType *p=*t,*q,*s;   int flag=0;
    if(!SearchElem(*t,&p,&q,kx));     /*在*t 为根的子树上查找*/
    {    s=(NodeType *)malloc(sizeof(NodeType));    /*申请结点，并赋值*/
         s->elem.key=kx;s->lc=NULL;s->rc=NULL;
         flag=1;        /*设置插入成功标志*/
       if(!p) t=s;      /*向空树中插入时*/
         else
         {    if(kx>p->elem.key) p->rc=s;        /*插入结点为 p 的右子女*/
              else      p->lc=s;        /*插入结点为 p 的左子女*/
         }
    }
    return flag;
}
```

四.二叉排序树删除操作

从二叉排序树中删除一个结点之后，使其仍能保持二叉排序树的特性即可。

设待删结点为*p（p 为指向待删结点的指针），其双亲结点为*f，以下分三种情况进行讨论。

1. *p 结点为叶结点，由于删去叶结点后不影响整棵树的特性，所以，只需将被删结点的双亲结点相应指针域改为空指针。

2. *p 结点只有右子树 p_r 或只有左子树 p_l，此时，只需将 p_r 或 p_l 替换*f 结点的*p 子树即可。如图 9.7。

3. *p 结点既有左子树 P_l 又有右子树 P_r，可按中序遍历保持有序进行调整。

设删除*p 结点前，中序遍历序列为：

① P 为 F 的左子女时有：…，P_1 子树，P，P_j，S 子树，P_k，S_k子树，…，P_2，S_2子树，P_1，S_1子树，F，…

② P 为 F 的右子女时有：…，F，P_1 子树，P，P_j，S 子树，P_k，S_k子树，…，P_2，S_2子树，P_1，S_1子树，…

则删除*p 结点后，中序遍历序列应为：

① P 为 F 的左子女时有：…，P_1 子树，P_j，S 子树，P_k，S_k子树，…，P_2，S_2子树，P_1，S_1子树，F，…

② P 为 F 的右子女时有：…，F，P_1 子树，P_j，S 子树，P_k，S_k子树，…，P_2，S_2子树，P_1，S_1子树，…

有两种调整方法：

① 直接令 p_l 为*f 相应的子树，以 p_r 为 p_l 中序遍历的最后一个结点 p_k 的右子树；

② 令*p 结点的直接前驱 P_r 或直接后继（对 P_l 子树中序遍历的最后一个结点 P_k）替换*p 结点，再按 2.的方法删去 P_r 或 P_k。图 9.8 所示的就是以*p 结点的直接前驱 P_r 替换*p。

【算法 9.6】

```
int DeleteNode(NodeType **t,KeyType kx)
```

```
{    NodeType *p=*t,*q,*s,**f;
     int flag=0;
     if(SearchElem(*t,&p,&q,kx));
     {    flag=1;        /*查找成功，置删除成功标志*/
          if(p= =q)  f=&(*t);      /*待删结点为根结点时*/
          else      /*待删结点非根结点时*/
          {    f=&(p->lc); if(kx>p->elem.key)    f=&(p->rc);
          }          /*f 指向待删结点的父结点的相应指针域*/
          if(!q->rc)     *f=q->lc;      /*若待删结点无右子树，以左子树替换待删结点*/
          else
          {    if(!q->lc)     *f=q->rc;      /*若待删结点无左子树，以右子树替换待删结点*/
               else      /*既有左子树又有右子树*/
               {    p=q->rc;s=p;
                    while(p->lc)  {s=p;p=p->lc;} /*在右子树上搜索待删结点的前驱 p*/
                    *f=p;p->lc=q->lc;        /*替换待删结点 q，重接左子树*/
                    if(s!=p)
                    {    s->lc=p->rc;        /*待删结点的右子女有左子树时，还要重接右子树*/
                         p->rc=q->rc;
                    }
               }
          }
     free(q);
     }
     return flag;
}
```

对给定序列建立二叉排序树，若左右子树均匀分布，则其查找过程类似于有序表的折半查找。但若给定序列原本有序，则建立的二叉排序树就蜕化为单链表，其查找效率同顺序查找一样。因此，对均匀的二叉排序树进行插入或删除结点后，应对其调整，使其依然保持均匀。

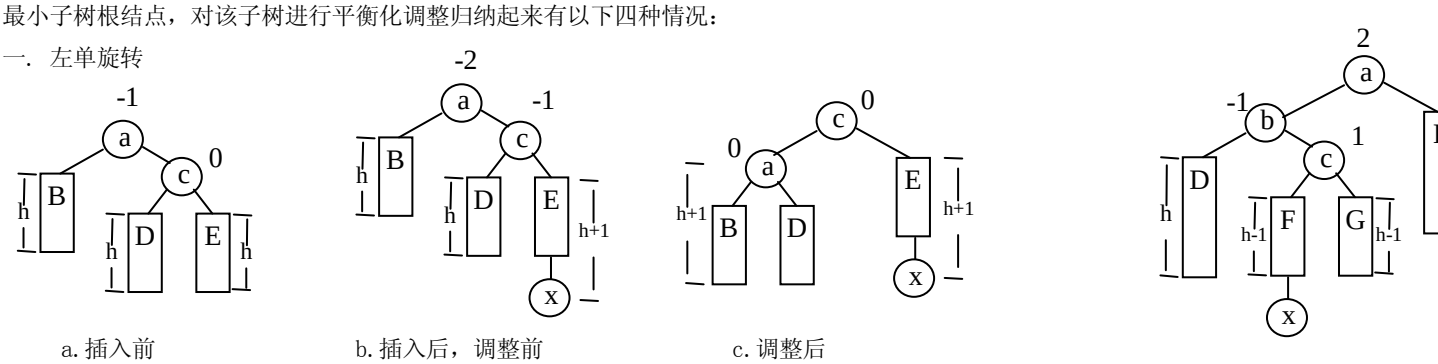
9.3.2 平衡二叉树(AVL 树)

平衡二叉树或者是一棵空树，或者是具有下列性质的二叉排序树：它的左子树和右子树都是平衡二叉树，且左子树和右子树高度之差的绝对值不超过 1。

图 9.9 给出了两棵二叉排序树，每个结点旁边所注数字是以该结点为根的树中，左子树与右子树高度之差，这个数字称为结点的平衡因子。由平衡二叉树定义，所有结点的平衡因子只能取-1，0，1 三个值之一。若二叉排序树中存在这样的结点，其平衡因子的绝对值大于 1，这棵树就不是平衡二叉树。如图 9.9 （a）所示的二叉排序树。

在平衡二叉树上插入或删除结点后，可能使树失去平衡，因此，需要对失去平衡的树进行平衡化调整。设 a 结点为失去平衡的最小子树根结点，对该子树进行平衡化调整归纳起来有以下四种情况：

一. 左单旋转



a.插入前　　b.插入后，调整前　　c.调整后

图 9.10

如图 9.10 的图(a)为插入前的子树。其中，B 为结点 a 的左子树，D、E 分别为结点 c 的左右子树，B、D、E 三棵子树的高均为 h。图(a)所示的子树是平衡二叉树。

在图(a)所示的树上插入结点 x，如图(b)所示。结点 x 插入在结点 c 的右子树 E 上，导致结点 a 的平衡因子绝对值大于 1，以结点 a 为根的子树失去平衡。

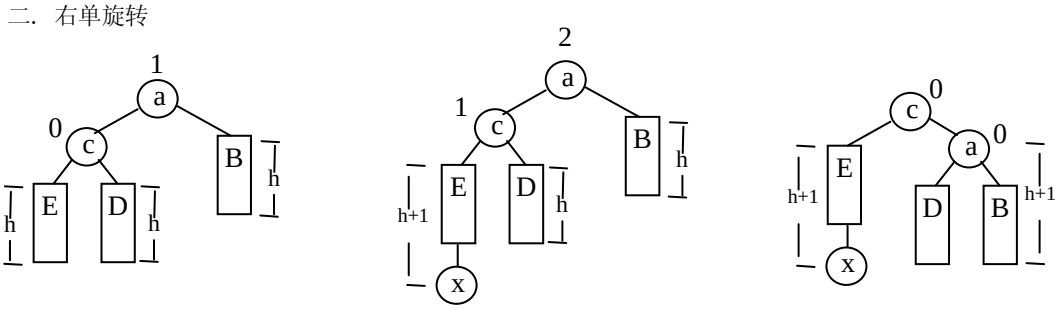
【调整策略】

调整后的子树除了各结点的平衡因子绝对值不超过 1，还必须是二叉排序树。由于结点 c 的左子树 D 可作为结点 a 的右子树，将结点 a 为根的子树调整为左子树是 B，右子树是 D，再将结点 a 为根的子树调整为结点 c 的左子树，结点 c 为新的根结点，如图(c)。

【平衡化调整操作判定】

沿插入路径检查三个点 a、c、E，若它们处于“\”直线上的同一个方向，则要作左单旋转，即以结点 c 为轴逆时针旋转。

二. 右单旋转



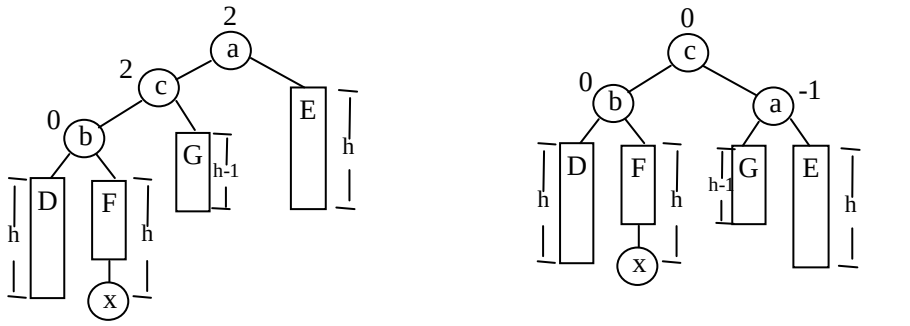
a. 插入前　　b. 插入后，调整前　　c. 调整后

图 9.11

右单旋转与左单旋转类似，沿插入路径检查三个点 a、c、E，若它们处于“/”直线上的同一个方向，则要作右单旋转，即以结点 c 为轴顺时针旋转，如图 9.11 所示。

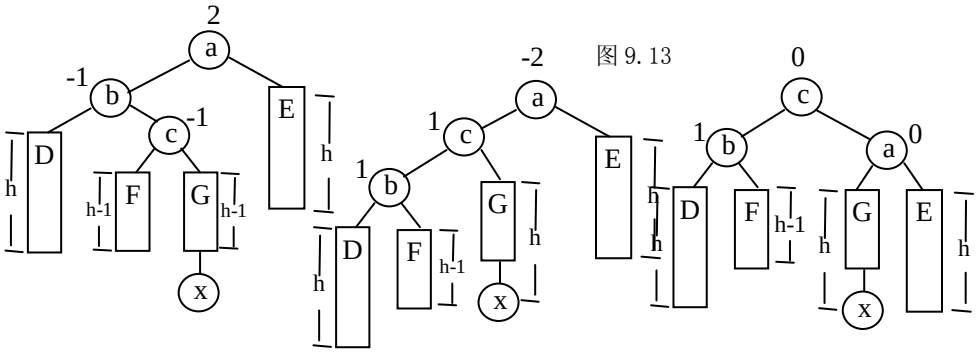
三. 先左后右双向旋转

如图 9.12 为插入前的子树，根结点 a 的左子树比右子树高度高 1，待插入结点 x 将插入到结点 b 的右子树上，并使结点 b 的右子树高度增 1，从而使结点 a 的平衡因子的绝对值大于 1，导致结点 a 为根的子树平衡被破坏。



a. 插入后，调整前　　b. 先左旋转　　c. 再右旋转

图 9.13



d. 插入后，调整前　　e. 先左旋转　　f. 再右旋转

图 9.14

沿插入路径检查三个点 a、b、c，若它们呈“<”字形，需要进行先左后右双向旋转：

1. 首先，对结点 b 为根的子树，以结点 c 为轴，向左逆时针旋转，结点 c 成为该子树的新根，如图(b、e)；
2. 由于旋转后，待插入结点 x 相当于插入到结点 b 为根的子树上，这样 a、c、b 三点处于“/”直线上的同一个方向，则要作右单旋转，即以结点 c 为轴顺时针旋转，如图(c、f)。

四. 先右后左双向旋转

先右后左双向旋转和先左后右双向旋转对称，请读者自行补充整理。

在平衡的二叉排序树 T 上插入一个关键码为 kx 的新元素，递归算法可描述如下：

1. 若 T 为空树，则插入一个数据元素为 kx 的新结点作为 T 的根结点，树的深度增 1；
2. 若 kx 和 T 的根结点关键码相等，则不进行插入；
3. 若 kx 小于 T 的根结点关键码，而且在 T 的左子树中不存在与 kx 有相同关键码的结点， 则将新元素插入在 T 的左子树上，并且当插入之后的左子树深度增加 1 时，分别就下列情况进行处理：

① T 的根结点平衡因子为-1(右子树的深度大于左子树的深度)，则将根结点的平衡因子更改为 0，T 的深度不变；

② T 的根结点平衡因子为 0(左、右子树的深度相等)，则将根结点的平衡因子更改为 1，T 的深度增加 1；

③ T 的根结点平衡因子为 1(左子树的深度大于右子树的深度)，则

若 T 的左子树根结点的平衡因子为 1，需进行单向右旋平衡处理，并且在右旋处理之后，将根结点和其右子树根结点的平衡因子更改为 0，树的深度不变；

若 T 的左子树根结点平衡因子为-1，需进行先左后右双向旋转平衡处理，并且在旋转处理之后，修改根结点和其左、右子树根结点的平衡因子，树的深度不变。

4. 若 kx 大于 T 的根结点关键码，而且在 T 的右子树中不存在与 kx 有相同关键码的结点，则将新元素插入在 T 的右子树上，并且当插入之后的右子树深度增加 1 时，分别就不同情况处理之。其处理操作和 (3.) 中所述相对称，读者可自行补充整理。

【算法 9.7】

```
typedef  struct  NODE{
          ElemType    elem;  /*数据元素*/
          int         bf;    /*平衡因子*/
         struct   NODE   *lc,*rc;   /*左右子女指针*/
     }NodeType;     /*结点类型*/

void  R_Rotate(NodeType **p)
{    /*对以*p 指向的结点为根的子树，作右单旋转处理，处理之后，*p 指向的结点为子树的新根*/
     lp=(*p)->lc;          /*lp 指向*p 左子树根结点*/
    (*p)->lc=lp->rc;     /*lp 的右子树挂接*p 的左子树*/
    lp->rc=*p; *p=lp;    /*  *p 指向新的根结点*/
}

void  L_Rotate(NodeType **p)
{    /*对以*p 指向的结点为根的子树，作左单旋转处理，处理之后，*p 指向的结点为子树的新根*/
     lp=(*p)->rc;          /*lp 指向*p 右子树根结点*/
    (*p)->rc=lp->lc;     /*lp 的左子树挂接*p 的右子树*/
    lp->lc=*p; *p=lp;   /* *p 指向新的根结点*/
}

#define  LH  1  /* 左高 */
#define  EH  0  /* 等高 */
#define  RH  1  /* 右高 */

void LeftBalance((NodeType **p)
{    /*对以*p 指向的结点为根的子树，作左平衡旋转处理，处理之后，*p 指向的结点为子树的新根*/
     lp=(*p)->lc;          /*lp 指向*p 左子树根结点*/
     switch((*p)->bf)      /*检查*p 平衡度，并作相应处理*/
```

```
    {case  LH:     /*新结点插在*p 左子女的左子树上，需作单右旋转处理*/
          (*p)->bf=lp->bf=EH;R_Rotate(p);break;
     case  EH:     /*原本左、右子树等高，因左子树增高使树增高*/
          (*p)->bf=LH;  *paller=TRUE;break;
     case  RH:     /*新结点插在*p 左子女的右子树上，需作先左后右双旋处理*/
          rd=lp->rc;     /*rd 指向*p 左子女的右子树根结点*/
          switch(rd->bf)          /*修正*p 及其左子女的平衡因子*/
          { case   LH:(*p)->bf=RH;lp->bf=EH;break;
            case   EH:(*p)->bf=lp->bf=EH;break;
            case   RH:(*p)->bf=EH;lp->bf=LH;break;
          }/*switch(rd->bf)*/
          rd->bf=EH;          L_Rotate(&((*p)->lc));       /*对*p 的左子树作左旋转处理*/
          R_Rotate(p);       /*对*t 作右旋转处理*/
     }/*switch((*p)->bf)*/
}/*LeftBalance*/

int  InsertAVL(NodeType **t,ElemType e,Boolean *taller)
{    /*若在平衡的二叉排序树 t 中不存在和 e 有相同关键码的结点，则插入一个数据元素为 e 的*/
     /*新结点，并反回 1，否则反回 0。若因插入而使二叉排序树失去平衡，则作平衡旋转处理，*/
     /*布尔型变量 taller 反映 t 长高与否*/
     if(!(*t))   /*插入新结点，树“长高”，置 taller 为 TURE*/
     {    *t=(NodeType *)malloc(sizeof(NodeType)); (*T)->elem=e;
          (*t)->lc=(*t)->rc=NULL;(*t)->bf=EH;*taller=TRUE;
     }/*if*/
     else
     {    if(e.key==(*t)->elem.key)     /*树中存在和 e 有相同关键码的结点，不插入*/
          {    taller=FALSE; return 0;}
          if(e.key<(*t)->elem.key)
          {    /*应继续在*t 的左子树上进行*/
               if(!InsertAVL(&((*t)->lc)),e,&taller)) return 0;  /*未插入*/
               if(*taller)    /*已插入到(*t)的左子树中，且左子树增高*/
                    switch((*t)->bf)    /*检查*t 平衡度*/
                    {case  LH:     /*原本左子树高，需作左平衡处理*/
                         LeftBalance(t);     *taller=FALSE;break;
                     case  EH:     /*原本左、右子树等高，因左子树增高使树增高*/
                         (*t)->bf=LH;  *taller=TRUE;break;
                     case  RH:     /*原本右子树高，使左、右子树等高*/
                         (*t)->bf=EH;  *taller=FALSE;break;
                    }
          }/*if*/
          else           /*应继续在*t 的右子树上进行*/
          {    if(!InsertAVL(&((*t)->rc)),e,&taller)) return 0;  /*未插入*/
               if(*taller)           /*已插入到(*t)的左子树中，且左子树增高*/
                    switch((*t)->bf)    /*检查*t 平衡度*/
                    {case  LH:      /*原本左子树高，使左、右子树等高*/
                         (*t)->bf=EH;  *taller=FALSE;break;
                     case  EH:     /*原本左、右子树等高，因右子树增高使树增高*/
                         (*t)->bf=RH;  *taller=TRUE;break;
                     case  RH: /*原本右子树高，需作右平衡处理*/
```

```
                    RightBalance(t);    *taller=FALSE;break;
                }
            }/*else*/
        }/*else*/
        return 1;
}/*InsertAVL*/
```

【平衡树的查找分析】

在平衡树上进行查找的过程和二叉排序树相同，因此，在查找过程中和给定值进行比较的关键码个数不超过树的深度。那么，含有 n 个关键码的平衡树的最大深度是多少呢？为解答这个问题，我们先分析深度为 h 的平衡树所具有的最少结点数。

假设以 N_h 表示深度为 h 的平衡树中含有的最少结点数。显然，$N_0=0$，$N_1=1$，$N_2=2$，并且 $N_h=N_{h-1}+N_{h-2}+1$。这个关系和斐波那契序列极为相似。利用归纳法容易证明：当 h≥0 时 $N_h = F_{h+2} - 1$，而 F_k 约等于 $\Phi^h/\sqrt{5}$(其中 $\Phi = \frac{1+\sqrt{5}}{2}$)，则 N_h 约等于 $\Phi^{h+2}/\sqrt{5}-1$。反之，含有 n 个结点的平衡树的最大深度为 $\log_\Phi(\sqrt{5}(n+1))-2$。因此，在平衡树上进行查找的时间复杂度为 O(logn)。

上述对二叉排序树和二叉平衡树的查找性能的讨论都是在等概率的提前下进行的。

9.3.3 B-树和 B+树

一.B-树及其查找

B-树是一种平衡的多路查找树，它在文件系统中很有用。

定义：一棵 m 阶的 B-树，或者为空树，或为满足下列特性的 m 叉树：

⑴树中每个结点至多有 m 棵子树；

⑵若根结点不是叶子结点，则至少有两棵子树；

⑶除根结点之外的所有非终端结点至少有⌈m/2⌉棵子树；

⑷所有的非终端结点中包含以下信息数据：(n，A_0，K_1，A_1，K_2，…，K_n，A_n)

其中：K_i（i=1,2,…,n）为关键码，且 $K_i<K_{i+1}$，A_i 为指向子树根结点的指针(i=0,1,…,n)，且指针 A_{i-1} 所指子树中所有结点的关键码均小于 K_i（i=1,2,…,n)，A_n 所指子树中所有结点的关键码均大于 K_n，⌈m/2⌉-1≤n≤m -1 ，n 为关键码的个数。

⑸所有的叶子结点都出现在同一层次上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空)。

B-树的查找类似二叉排序树的查找，所不同的是 B-树每个结点上是多关键码的有序表，在到达某个结点时，先在有序表中查找，若找到，则查找成功；否则，到按照对应的指针信息指向的子树中去查找，当到达叶子结点时，则说明树中没有对应的关键码，查找失败。即在 B-树上的查找过程是一个顺指针查找结点和在结点中查找关键码交叉进行的过程。比如，在图 9.12 中查找关键码为 93 的元素。首先，从 t 指向的根结点（a）开始，结点（a）中只有一个关键码，且 93 大于它，因此，按（a）结点指针域 A_1 到结点（c）去查找，结点（c）由两个关键码，而 93 也都大于它们，应按（c）结点指针域 A_2 到结点（i）去查找，在结点（i）中顺序比较关键码，找到关键码 K_3。

【算法 9.8】

```
#define   m    5          /*B 树的阶，暂设为 5*/
typedef   struct NODE{
    int  keynum;          /*结点中关键码的个数，即结点的大小*/
    struct  NODE  *parent;    /*指向双亲结点*/
    KeyType key[m+1];         /*关键码向量，0 号单元未用*/
    struct  NODE  *ptr[m+1];    /*子树指针向量*/
    record    *recptr[m+1];    /*记录指针向量*/
    }NodeType;                /*B 树结点类型*/

typedef   struct{
    NodeType    *pt;          /*指向找到的结点*/
    int       i;          /*在结点中的关键码序号，结点序号区间[1…m]*/
```

```
    int       tag;       /* 1:查找成功，0:查找失败*/
    }Result;             /*B 树的查找结果类型*/

Result  SearchBTree(NodeType *t,KeyType kx)
{   /*在 m 阶 B 树 t 上查找关键码 kx，反回(pt,i,tag)。若查找成功，则特征值 tag=1，*/
    /*指针 pt 所指结点中第 i 个关键码等于 kx；否则，特征值 tag=0,等于 kx 的关键码记录*/
    /*应插入在指针 pt 所指结点中第 i 个和第 i+1 个关键码之间*/
    p=t;q=NULL;found=FALSE;i=0;  /*初始化，p 指向待查结点，q 指向 p 的双亲*/
    while(p&&!found)
    {   n=p->keynum;i=Search(p,kx);  /*在 p-->key[1…keynum]中查找*/
        if(i>0&&p->key[i]= =kx) found=TRUE;  /*找到*/
        else {q=p;p=p->ptr[i];}
    }
    if(found) return (p,i,1);          /*查找成功*/
    else return (q,i,0);               /*查找不成功，反回 kx 的插入位置信息*/
}
```

【查找分析】

B-树的查找是由两个基本操作交叉进行的过程，即

⑴在 B-树上找结点；　⑵在结点中找关键码。

由于，通常 B-树是存储在外存上的，操作⑴就是通过指针在磁盘相对定位，将结点信息读入内存，之后，再对结点中的关键码有序表进行顺序查找或折半查找。因为，在磁盘上读取结点信息比在内存中进行关键码查找耗时多，所以，在磁盘上读取结点信息的次数，即 B-树的层次树是决定 B-树查找效率的首要因素。

那么，对含有 n 个关键码的 m 阶 B-树，最坏情况下达到多深呢？可按二叉平衡树进行类似分析。首先，讨论 m 阶 B-数各层上的最少结点数。

由 B-树定义：第一层至少有 1 个结点；第二层至少有 2 个结点；由于除根结点外的每个非终端结点至少有$\lceil m/2 \rceil$棵子树，则第三层至少有 2 ($\lceil m/2 \rceil$) 个结点；……；以此类推，第 k+1 层至少有 2 ($\lceil m/2 \rceil$) $^{k-1}$ 个结点。而 k+1 层的结点为叶子结点。若 m 阶 B-树有 n 个关键码，则叶子结点即查找不成功的结点为 n+1，由此有：

$$n+1 \geqslant 2*(\lceil m/2 \rceil)^{k-1}$$

即

$$k \leq \log_{\lceil m/2 \rceil}(\frac{n+1}{2})+1$$

这就是说，在含有 n 个关键码的 B-树上进行查找时，从根结点到关键码所在结点的路径上涉及的结点数不超过

二.B-树的插入和删除

$$\log_{\lceil m/2 \rceil}(\frac{n+1}{2})+1$$

1.插入

在 B-树上插入关键码与在二叉排序树上插入结点不同，关键码的插入不是在叶结点上进行的，而是在最底层的某个非终端结点中添加一个关键码，若该结点上关键码个数不超过 m-1 个，则可直接插入到该结点上；否则，该结点上关键码个数至少达到 m 个，因而使该结点的子树超过了 m 棵，这与 B-树定义不符。所以要进行调整，即结点的“分裂”。方法为：关键码加入结点后，将结点中的关键码分成三部分，使得前后两部分关键码个数

个结点将其插入到父结点中。若插入父结点而使父结点中关键码个数超过 m-1，则父结点继续分裂，直到插入某个父结点，其关键码均大于等于（$\lceil m/2 \rceil$-1），而中间部分只有一个结点。前后两部分成为两个结点，中间的一个数小于 m。可见，B-树是从底向上生长的。

【算法 9.9】

```
int InserBTree(NodeType **t,KeyType kx,NodeType *q,int i)
{   /*在 m 阶 B 树*t 上结点*q 的 key[i],key[i+1]之间插入关键码 kx*/
```

```
    /*若引起结点过大，则沿双亲链进行必要的结点分裂调整，使*t 仍为 m 阶 B 树*/
     x=kx;ap=NULL;finished=FALSE;
     while(q&&!finished)
     {    Insert(q,i,x,ap);    /*将 x 和 ap 分别插入到 q->key[i+1]和 q->ptr[i+1]*/
          if(q->keynum<m)   finished=TRUE;  /*插入完成*/
          else
          {    /*分裂结点*p*/
               s=m/2;split(q,ap);x=q->key[s];
          /*将 q->key[s+1…m],q->ptr[s…m]和 q->recptr[s+1…m]移入新结点*ap*/
               q=q->parent;
               if(q)      i=Search(q,kx);  /*在双亲结点*q 中查找 kx 的插入位置*/

          }/*else*/
     }/*while*/
     if(!finished)  /*(*t)是空树或根结点已分裂为*q*和 ap*/
          NewRoot(t,q,x,ap);  /*生成含信息(t,x,ap)的新的根结点*t,原*t 和 ap 为子树指针*/
}
```

2. 删除

分两种情况：

(1)删除最底层结点中关键码

a)若结点中关键码个数大于 $\lceil m/2 \rceil$-1，直接删去。

b)否则除余项与左兄弟(无左兄弟，则找左兄弟)项数之和大于等于 2($\lceil m/2 \rceil$-1) 就与它们父结点中的有关项一起重新分配。如删去图 9.16 (h)中的 76 得图 9.17

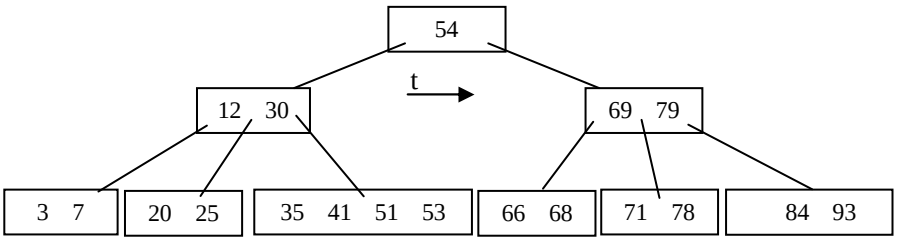


图 9.17

(c)若删除后，余项与左兄弟或右兄弟之和均小于 2(　　-1)，就将余项与左兄弟 (无左兄弟时，与右兄弟)合并。由于两个结点合并后，父结点中相关项不能保持，把相关项也并入合并项。若此时父结点被破坏，则继续调整，直到根。如删去图 9.16 (h)中 7，得图 9.18。

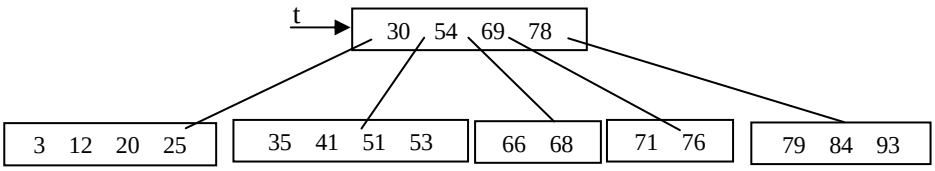


图 9.18

(2)删除为非底层结点中关键码

若所删除关键码非底层结点中的 K_i，则可以指针 A_i所指子树中的最小关键码 X 替代 K_i，然后，再删除关键码 X，直到这个 X 在最底层结点上，即转为(1)的情形。

删除程序，请读者自己完成。

三. B+树

B+树是应文件系统所需而产生的一种 B-树的变形树。一棵 m 阶的 B+树和 m 阶的 B-树的差异在于：

⑴有 n 棵子树的结点中含有 n 个关键码；

⑵所有的叶子结点中包含了全部关键码的信息，及指向含有这些关键码记录的指针，且叶子结点本身依关键码的大小自小而大的顺序链接。

⑶所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键码。

例如图 9.16 所示为一棵五阶的 B+树，通常在 B+树上有两个头指针，一个指向根结

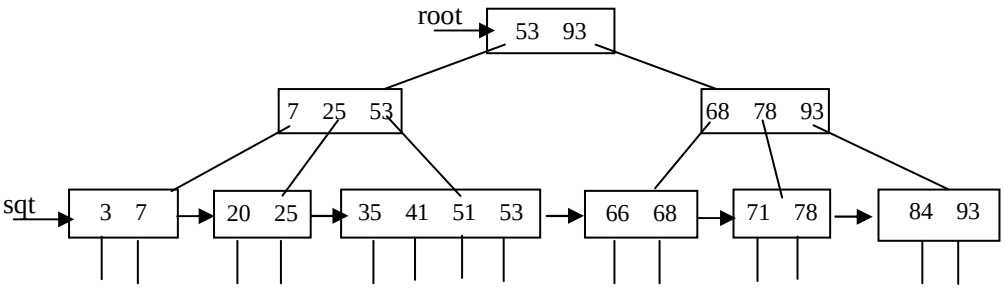


图 9.19 一棵 5 阶二叉

点，另一个指向关键码最小的叶子结点 。因此，可以对 B+树进行两种查找运算：一种是从最小关键码起顺序查找，另一种是根结点开始，进行随机查找。

$\lceil m/2 \rceil$

在 B+树上进行随机查找、插入和删除的过程基本上与 B-树类似。只是在查找时，若非终端结点上的关键码等于给定值，并不终止，而是继续向下直到叶子结点。因此，在 B+树，不管查找成功与否，每次查找都是走了一条从根到叶子结点的路径。B+树查找的分析类似于 B-树。B+树的插入仅在叶子结点上进行，当结点中的关键码个数大于 m 时要

分裂成两个结点，它们所含关键码的个数为 $\left\lceil \frac{m+1}{2} \right\rceil$ 和 $\left\lfloor \frac{m+1}{2} \right\rfloor$。并且，它们的双亲结点中应同时包含

这两个结点中最大关键码及指向这两个结点的指针。

在 B+树上进行删除也仅在叶子结点进行，当叶子结点中的最大关键码被删除时，其在非终端结点中的值可以作为一个“分界关键码”存在。若因删除

而使结点中关键码的个数少于 $\lceil m/2 \rceil$ 时，其和 $\lceil m/2 \rceil$ 兄弟结点的合并过程亦和 B-树类似。

9.4 哈希表查找(杂凑法)

9.4.1 哈希表与哈希方法

理想的情况是依据关键码直接得到其对应的数据元素位置，即要求关键码与数据元素间存在一一对应关系，通过这个关系，能很快地由关键码得到对应的数据元素位置。

哈希表与哈希方法：选取某个函数，依该函数按关键码计算元素的存储位置，并按此存放；查找时，由同一个函数对给定值 kx 计算地址，将 kx 与地址单元中元素关键码进行比，确定查找是否成功，这就是**哈希方法**(杂凑法)；哈希方法中使用的转换函数称为**哈希函数**(杂凑函数)；按这个思想构造的表称为**哈希表**(杂凑表)。

对于 n 个数据元素的集合，总能找到关键码与存放地址一一对应的函数。若最大关键为 m，可以分配 m 个数据元素存放单元，选取函数 f(key)=key 即可，但这样会造成存储空间的很大浪费，甚至不可能分配这么大的存储空间。通常关键码的集合比哈希地址集合大得多，因而经过哈希函数变换后，可能将不同的关键码映射到同一个哈希地址上，这种现象称为**冲突**(Collision)，映射到同一哈希地址上的关键码称为**同义词**。可以说，冲突不可能避免，只能尽可能减少。所以，哈希方法需要解决以下两个问题：

1. 构造好的哈希函数

（1）所选函数尽可能简单，以便提高转换速度。

（2）所选函数对关键码计算出的地址，应在哈希地址集中大致均匀分布，以减少空间浪费。

2. 制定解决冲突的方案。

9.4.2 常用的哈希函数

一. 直接定址法

Hash(key)=a • key+b　　(a、b 为常数)

即取关键码的某个线性函数值为哈希地址，这类函数是一一对应函数，不会产生冲突，但要求地址集合与关键码集合大小相同，因此，对于较大的关键码集合不适用。

二. 除留余数法

Hash(key)=key mod p (p 是一个整数)

即取关键码除以 p 的余数作为哈希地址。使用除留余数法，选取合适的 p 很重要，若哈希表表长为 m，则要求 p≤m，且接近 m 或等于 m。p 一般选取质数，也可以是不包含小于 20 质因子的合数。

三. 乘余取整法

Hash(key)=⌊B*(A*key mod 1)⌋ (A、B 均为常数，且 0<A<1，B 为整数)

以关键码 key 乘以 A，取其小数部分(A*key mod 1 就是取 A*key 的小数部分)，之后再用整数 B 乘以这个值，取结果的整数部分作为哈希地址。

该方法 B 取什么值并不关键，但 A 的选择却很重要，最佳的选择依赖于关键码集合的特征。

四. 数字分析法

设关键码集合中，每个关键码均由 m 位组成，每位上可能有 r 种不同的符号。

数字分析法根据 r 种不同的符号，在各位上的分布情况，选取某几位，组合成哈希地址。所选的位应是各种符号在该位上出现的频率大致相同。

五. 平方取中法

对关键码平方后，按哈希表大小，取中间的若干位作为哈希地址。

六. 折叠法(Folding)

此方法将关键码自左到右分成位数相等的几部分，最后一部分位数可以短些，然后将这几部分叠加求和，并按哈希表表长，取后几位作为哈希地址。这种方法称为折叠法。

有两种叠加方法：

1. 移 位 法 —— 将各部分的最后一位对齐相加。
2. 间界叠加法 —— 从一端向另一端沿各部分分界来回折叠后，最后一位对齐相加。

9.4.3 处理冲突的方法

一. 开放定址法

所谓开放定址法，即是由关键码得到的哈希地址一旦产生了冲突，也就是说，该地址已经存放了数据元素，就去寻找下一个空的哈希地址，只要哈希表足够大，空的哈希地址总能找到，并将数据元素存入。

找空哈希地址方法很多，下面介绍三种：

1. 线性探测法

H_i=(Hash(key)+d_i) mod m (1≤i < m)

其中：

Hash(key)为哈希函数

m 为哈希表长度

d_i 为增量序列 1，2，……，m-1，且 d_i=i

线性探测法可能使第 i 个哈希地址的同义词存入第 i+1 个哈希地址，这样本应存入第 i+1 个哈希地址的元素变成了第 i+2 个哈希地址的同义词，……，因此，可能出现很多元素在相邻的哈希地址上“堆积”起来，大大降低了查找效率。为此，可采用二次探测法，或双哈希函数探测法，以改善“堆积”问题。

2. 二次探测法

H_i=(Hash(key)±d_i) mod m

其中：

Hash(key)为哈希函数

m 为哈希表长度，m 要求是某个 4k+3 的质数(k 是整数)

d_i 为增量序列 1^2，-1^2，2^2，-2^2，……，q^2，$-q^2$ 且 q≤ (m-1) $\frac{1}{2}$

3. 双哈希函数探测法

H_i=(Hash(key)+i*ReHash(key)) mod m (i=1，2，……，m-1)

其中：

Hash(key)，ReHash(key)是两个哈希函数，

m 为哈希表长度

双哈希函数探测法，先用第一个函数 Hash(key)对关键码计算哈希地址，一旦产生地址冲突，再用第二个函数 ReHash(key)确定移动的步长因子，最后，通过步长因子序列由探测函数寻找空的哈希地址。

比如，Hash(key)=a 时产生地址冲突，就计算 ReHash(key)=b，则探测的地址序列为

$H_1=(a+b) \bmod m$，$H_2=(a+2b) \bmod m$，……，$H_{m-1}=(a+(m-1)b) \bmod m$

二. 拉链法

设哈希函数得到的哈希地址域在区间[0，m-1]上，以每个哈希地址作为一个指针，指向一个链，即分配指针数组 ElemType *eptr[m]；

建立 m 个空链表，由哈希函数对关键码转换后，映射到同一哈希地址 i 的同义词均加入到*eptr[i]指向的链表中。

三. 建立一个公共溢出区

设哈希函数产生的哈希地址集为[0，m-1]，则分配两个表：

一个基本表 ElemType base_tbl[m]；每个单元只能存放一个元素；

一个溢出表 ElemType over_tbl[k]；只要关键码对应的哈希地址在基本表上产生冲突，则所有这样的元素一律存入该表中。查找时，对给定值 kx 通过哈希函数计算出哈希地址 i，先与基本表的 base_tbl[i]单元比较，若相等，查找成功；否则，再到溢出表中进行查找。

9.4.4 哈希表的查找分析

哈希表的查找过程基本上和造表过程相同。一些关键码可通过哈希函数转换的地址直接找到，另一些关键码在哈希函数得到的地址上产生了冲突，需要按处理冲突的方法进行查找。在介绍的三种处理冲突的方法中，产生冲突后的查找仍然是给定值与关键码进行比较的过程。所以，对哈希表查找效率的量度，依然用平均查找长度来衡量。

查找过程中，关键码的比较次数，取决于产生冲突的多少，产生的冲突少，查找效率就高，产生的冲突多，查找效率就低。因此，影响产生冲突多少的因素，也就是影响查找效率的因素。影响产生冲突多少有以下三个因素：

1. 哈希函数是否均匀； 2. 处理冲突的方法； 3. 哈希表的装填因子。

分析这三个因素，尽管哈希函数的“好坏”直接影响冲突产生的频度，但一般情况下，我们总认为所选的哈希函数是“均匀的”，因此，可不考虑哈希函数对平均查找长度的影响。就线性探测法和二次探测法处理冲突的例子看，相同的关键码集合、同样的哈希函数，但在数据元素查找等概率情况下，它们的平均查找长度却不同：

哈希表的装填因子定义为：$\alpha = \dfrac{\text{填入表中的元素个数}}{\text{哈希表的长度}}$

α 是哈希表装满程度的标志因子。由于表长是定值，α 与“填入表中的元素个数”成正比，所以，α 越大，填入表中的元素较多，产生冲突的可能性就越大；α 越小，填入表中的元素较少，产生冲突的可能性就越小。

实际上，哈希表的平均查找长度是装填因子 α 的函数，只是不同处理冲突的方法有不同的函数。以下给出几种不同处理冲突方法的平均查找长度：

处理冲突的方法	平均查找长度	
	查找成功时	查找不成功时
线性探测法	$S_{nl} \approx \frac{1}{2}(1+\frac{1}{1-\alpha})$	$U_{nl} \approx \frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
二次探测法 与双哈希法	$S_{nr} \approx -\frac{1}{\alpha}\ln(1-\alpha)$	$U_{nr} \approx \frac{1}{1-\alpha}$
拉链法	$S_{nc} \approx 1+\frac{\alpha}{2}$	$U_{nc} \approx \alpha + e^{-\alpha}$

图 9.22

哈希方法存取速度快，也较节省空间，静态查找、动态查找均适用，但由于存取是随机的，因此，不便于顺序查找。

第 10 章 排序

若对任意的数据元素序列，使用某个排序方法，对它按关键码进行排序：若相同关键码元素间的位置关系，排序前与排序后

保持一致，称此排序方法是**稳定**的；而不能保持一致的排序方法则称为**不稳定**的。

排序分为两类：内排序和外排序。

内排序：指待排序列完全存放在内存中所进行的排序过程，适合不太大的元素序列。

外排序：指排序过程中还需访问外存储器，足够大的元素序列，因不能完全放入内存，只能使用外排序。

10.2 插入排序

10.2.1 直接插入排序

设有 n 个记录，存放在数组 r 中，重新安排记录在数组中的存放顺序，使得按关键码有序。即

r[1].key≤r[2].key≤……≤r[n].key

先来看看向有序表中插入一个记录的方法：

设 1 < j ≤ n ， r[1].key≤r[2].key≤……≤r[j-1].key ， 将 r[j] 插 入 ， 重 新 安 排 存 放 顺 序 ， 使 得 r[1].key≤r[2].key≤……≤r[j].key，得到新的有序表，记录数增 1 。

【算法 10.1】

① r[0]=r[j]; //r[j]送 r[0]中，使 r[j]为待插入记录空位

i=j-1; //从第 i 个记录向前测试插入位置，用 r[0]为辅助单元， 可免去测试 i<1。

② 若 r[0].key≥r[i].key，转④。 //插入位置确定

③ 若 r[0].key < r[i].key 时，

r[i+1]=r[i]; i=i-1; 转②。 //调整待插入位置

④ r[i+1]=r[0]; 结束。 //存放待插入记录

直接插入排序方法：仅有一个记录的表总是有序的，因此，对 n 个记录的表，可从第二个记录开始直到第 n 个记录，逐个向有序表中进行插入操作，从而得到 n 个记录按关键码有序的表。

【算法 10.2】

```
void   InsertSort(S_TBL &p)
{   for(i=2; i<=p->length; i++)
        if(p->elem[i].key < p->elem[i-1].key)        /*小于时，需将 elem[i]插入有序表*/
        {   p->elem[0].key=p->elem[i].key;        /*为统一算法设置监测*/
            for(j=i-1; p->elem[0].key < p->elem[j].key; j--)
                p->elem[j+1].key=p->elem[j].key;    /*记录后移*/
            p->elem[j+1].key=p->elem[0].key;        /*插入到正确位置*/
        }
}
```

10.2.2 折半插入排序

直接插入排序的基本操作是向有序表中插入一个记录，插入位置的确定通过对有序表中记录按关键码逐个比较得到的。平均情况下总比较次数约为 $n^2/4$。既然是在有序表中确定插入位置，可以不断二分有序表来确定插入位置，即一次比较，通过待插入记录与有序表居中的记录按关键码比较，将有序表一分为二，下次比较在其中一个有序子表中进行，将子表又一分为二。这样继续下去，直到要比较的子表中只有一个记录时，比较一次便确定了插入位置。

二分判定有序表插入位置方法：

① low=1; high=j-1; r[0]=r[j]; // 有序表长度为 j-1，第 j 个记录为待插入记录
//设置有序表区间，待插入记录送辅助单元

② 若 low>high，得到插入位置，转⑤

③ low≤high，m=(low+high)/2; // 取表的中点，并将表一分为二，确定待插入区间*/

④ 若 r[0].key<r[m].key，high=m-1; //插入位置在低半区
否则，low=m+1; // 插入位置在高半区
转②

⑤ high+1 即为待插入位置，从 j-1 到 high+1 的记录，逐个后移，r[high+1]=r[0]；放置待插入记录。

【算法 10.3】

```
void   InsertSort(S_TBL *s)
```

```
{    /* 对顺序表 s 作折半插入排序 */
     for(i=2; i<=s->length; i++)
     {    s->elem[0]=s->elem[i];   /* 保存待插入元素 */
          low=i; high=i-1;     /* 设置初始区间 */
          while(low<=high)   /* 该循环语句完成确定插入位置 */
          {    mid=(low+high)/2;
               if(s->elem[0].key>s->elem[mid].key)
                    low=mid+1;       /* 插入位置在高半区中 */
               else  high=mid-1;       /* 插入位置在低半区中 */
          }/* while */
          for(j=i-1; j>=high+1; j--)      /* high+1 为插入位置 */
               s->elem[j+1]=s->elem[j];  /* 后移元素，留出插入空位 */
          s->elem[high+1]=s->elem[0];  /* 将元素插入 */
     }/* for */
}/* InsertSort */
```

【时间效率】

确定插入位置所进行的折半查找，关键码的比较次数至多为 $\lceil \log_2(n+1) \rceil$ 次，移动记录的次数和直接插入排序相同，故时间复杂度仍为 $O(n^2)$。是一个稳定的排序方法。

10.2.3 表插入排序

直接插入排序、折半插入排序均要大量移动记录，时间开销大。若要不移动记录完成排序，需要改变存储结构，进行表插入排序。所谓表插入排序，就是通过链接指针，按关键码的大小，实现从小到大的链接过程，为此需增设一个指针项。操作方法与直接插入排序类似，所不同的是直接插入排序要移动记录，而表插入排序是修改链接指针。用静态链表来说明。

```
#define  SIZE  200
typedef  struct{
           ElemType  elem;       /*元素类型*/
           int       next;       /*指针项*/
           }NodeType;            /*表结点类型*/
typedef    struct{
           NodeType  r[SIZE];    /*静态链表*/
           int       length;     /*表长度*/
           }L_TBL;               /*静态链表类型*/
```

假设数据元素已存储在链表中，且 0 号单元作为头结点，不移动记录而只是改变链指针域，将记录按关键码建为一个有序链表。首先，设置空的循环链表，即头结点指针域置 0，并在头结点数据域中存放比所有记录关键码都大的整数。接下来，逐个结点向链表中插入即可。

表插入排序得到一个有序的链表，查找则只能进行顺序查找，而不能进行随机查找，如折半查找。为此，还需要对记录进行重排。

重排记录方法：按链表顺序扫描各结点，将第 i 个结点中的数据元素调整到数组的第 i 个分量数据域。因为第 i 个结点可能是数组的第 j 个分量，数据元素调整仅需将两个数组分量中数据元素交换即可，但为了能对所有数据元素进行正常调整，指针域也需处理。

【算法 10.3】

```
1. j=l->r[0].next; i=1;      //指向第一个记录位置，从第一个记录开始调整
2. 若 i=l->length 时，调整结束；否则，
a. 若 i=j，j=l->r[j].next; i++; 转(2)  //数据元素应在这分量中，不用调整，处理下一个结点
b. 若 j>i，l->r[i].elem<-->l->r[j].elem;  //交换数据元素
       p=l->r[j].next;              // 保存下一个结点地址
       l->r[j].next=l->[i].next; l->[i].next=j;  // 保持后续链表不被中断
       j=p; i++; 转(2)              // 指向下一个处理的结点
c. 若 j<i，while(j<i) j=l->r[j].next; //j 分量中原记录已移走，沿 j 的指针域找寻原记录的位置
       转到(a)
```

【时效分析】

表插入排序的基本操作是将一个记录插入到已排好序的有序链表中，设有序表长度为 i，则需要比较至多 i+1 次，修改指针两次。因此，总比较次数与直接插入排序相同，修改指针总次数为 2n 次。所以，时间复杂度仍为 $O(n^2)$

10.2.4 希尔排序(Shell’s Sort)

希尔排序又称缩小增量排序，较前述几种插入排序方法有较大的改进。

直接插入排序算法简单，在 n 值较小时，效率比较高，在 n 值很大时，若序列按关键码基本有序，效率依然较高，其时间效率可提高到 O(n)。希尔排序即是从这两点出发，给出插入排序的改进方法。

希尔排序方法：

1. 选择一个步长序列 t_1，t_2，…，t_k，其中 $t_i>t_j$，$t_k=1$；
2. 按步长序列个数 k，对序列进行 k 趟排序；
3. 每趟排序，根据对应的步长 t_i，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅步长因子为 1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

【算法 10.5】

```
void   ShellInsert(S_TBL &p, int dk)
{    /*一趟增量为 dk 的插入排序，dk 为步长因子*/
     for(i=dk+1; i<=p->length; i++)
          if(p->elem[i].key < p->elem[i-dk].key)  /*小于时，需 elem[i]将插入有序表*/
          {    p->elem[0]=p->elem[i];         /*为统一算法设置监测*/
               for(j=i-dk; j>0&&p->elem[0].key < p->elem[j].key; j=j-dk)
                    p->elem[j+dk]=p->elem[j];     /*记录后移*/
               p->elem[j+dk]=p->elem[0];         /*插入到正确位置*/
          }
}
void   ShellSort(S_TBL *p, int dlta[], int t)
{    /*按增量序列 dlta[0, 1…, t-1]对顺序表*p 作希尔排序*/
     for(k=0; k<t; t++)
          ShellSort(p, dlta[k]);  /*一趟增量为 dlta[k]的插入排序*/
}
```

【时效分析】

希尔排序时效分析很难，关键码的比较次数与记录移动次数依赖于步长因子序列的选取，特定情况下可以准确估算出关键码的比较次数和记录的移动次数。目前还没有人给出选取最好的步长因子序列的方法。步长因子序列可以有各种取法，有取奇数的，也有取质数的，但需要注意：步长因子中除 1 外没有公因子，且最后一个步长因子必须为 1。希尔排序方法是一个不稳定的排序方法。

10.3 交换排序

最坏情况下：每次比较后均要进行三次移动，移动次数 $=\sum_{j=2}^{n}3(j-1)=\frac{3}{2}n(n-1)$

交换排序主要是通过两两比较待排记录的关键码，若发生与排序要求相逆，则交换之。

10.3.1 冒泡排序(Bubble Sort)

先来看看待排序列一趟冒泡的过程：设 1<j≤n，r[1],r[2],・・・,r[j]为待排序列，
通过两两比较、交换，重新安排存放顺序，使得 r[j]是序列中关键码最大的记录。一趟冒泡方法为：

① i=1;　　//设置从第一个记录开始进行两两比较

② 若 i≥j，一趟冒泡结束。

③ 比较 r[i].key 与 r[i+1].key，若 r[i].key≤r[i+1].key，不交换，转⑤

④ 当 r[i].key>r[i+1].key 时，　r[0]=r[i]; r[i]=r[i+1]; r[i+1]=r[0];
　　将 r[i]与 r[i+1]交换

⑤ i=i+1;　调整对下两个记录进行两两比较，转②

冒泡排序方法：对 n 个记录的表，第一趟冒泡得到一个关键码最大的记录 r[n]，第二趟冒泡对 n-1 个记录的表，再得到一个关键码最大的记录 r[n-1]，如此重复，直到 n 个记录按关键码有序的表。

【算法 10.6】

① j=n;　　//从 n 记录的表开始

② 若 j<2，排序结束

③ i=1;　　//一趟冒泡，设置从第一个记录开始进行两两比较，

④ 若 i≥j，一趟冒泡结束，j=j-1；冒泡表的记录数-1，转②

⑤ 比较 r[i].key 与 r[i+1].key，若 r[i].key≤r[i+1].key，不交换，转⑤

⑥ 当 r[i].key>r[i+1].key 时， r[i]<-->r[i+1]； 将 r[i]与 r[i+1]交换

⑦ i=i+1； 调整对下两个记录进行两两比较，转④

【效率分析】

空间效率：仅用了一个辅助单元。

时间效率：总共要进行 n-1 趟冒泡，对 j 个记录的表进行一趟冒泡需要 j-1 次关键码比较。

移动次数：

$$总比较次数=\sum_{j=2}^{n}(j-1)=\frac{1}{2}n(n-1)$$

最好情况下：待排序列已有序，不需移动。

10.3.2 快速排序

快速排序是通过比较关键码、交换记录，以某个记录为界(该记录称为支点)，将待排序列分成两部分。其中，一部分所有记录的关键码大于等于支点记录的关键码，另一部分所有记录的关键码小于支点记录的关键码。我们将待排序列按关键码以支点记录分成两部分的过程，称为一次划分。对各部分不断划分，直到整个序列按关键码有序。

一次划分方法：

设 1≤p<q≤n，r[p],r[p+1],...,r[q]为待排序列

① low=p；high=q； //设置两个搜索指针，low 是向后搜索指针，high 是向前搜索指针

r[0]=r[low]； //取第一个记录为支点记录，low 位置暂设为支点空位

② 若 low=high，支点空位确定，即为 low。

r[low]=r[0]; //填入支点记录，一次划分结束

否则，low<high，搜索需要交换的记录，并交换之

③ 若 low<high 且 r[high].key≥r[0].key //从 high 所指位置向前搜索，至多到 low+1 位置

high=high-1；转③ //寻找 r[high].key<r[0].key

r[low]=r[high]; //找到 r[high].key<r[0].key，设置 high 为新支点位置，

//小于支点记录关键码的记录前移。

④ 若 low<high 且 r[low].key<r[0].key //从 low 所指位置向后搜索，至多到 high-1 位置

low=low+1；转④ //寻找 r[low].key≥r[0].key

r[high]=r[low]; //找到 r[low].key≥r[0].key，设置 low 为新支点位置，

//大于等于支点记录关键码的记录后移。

转② //继续寻找支点空位

【算法 10.7】

```
int Partition(S_TBL *tbl,int low,int high) /*一趟快排序*/
{   /*交换顺序表 tbl 中子表 tbl->[low…high]的记录，使支点记录到位，并反回其所在位置*/
    /*此时，在它之前(后)的记录均不大(小)于它*/
    tbl->r[0]=tbl->r[low];   /*以子表的第一个记录作为支点记录*/
    pivotkey=tbl->r[low].key;  /*取支点记录关键码*/
    while(low<higu)              /*从表的两端交替地向中间扫描*/
    {    while(low<high&&tbl->r[high].key>=pivotkey)  high--;
         tbl->r[low]=tbl->r[high]; /*将比支点记录小的交换到低端*/
         while(low<high&&tbl-g>r[high].key<=pivotkey)  low++;
         tbl->r[low]=tbl->r[high]; /*将比支点记录大的交换到低端*/
    }
    tbl->r[low]=tbl->r[0];     /*支点记录到位*/
    return low;    /*反回支点记录所在位置*/
}
```

【算法 10.8】

```
void  QSort(S_TBL *tbl,int low,int high)    /*递归形式的快排序*/
{    /*对顺序表 tbl 中的子序列 tbl->[low…high]作快排序*/
     if(low<high)
     {    pivotloc=partition(tbl,low,high); /*将表一分为二*/
          QSort(tbl,low,pivotloc-1);  /*对低子表递归排序*/
          QSort(tbl,pivotloc+1,high);  /*对高子表递归排序*/
     }
}
```

【效率分析】

空间效率：快速排序是递归的，每层递归调用时的指针和参数均要用栈来存放，递归调用层次数与上述二叉树的深度一致。因而，存储开销在理想情况下为 $O(\log_2 n)$，即树的高度；在最坏情况下，即二叉树是一个单链，为 $O(n)$。

时间效率：在 n 个记录的待排序列中，一次划分需要约 n 次关键码比较，时效为 $O(n)$，若设 $T(n)$ 为对 n 个记录的待排序列进行快速排序所需时间。

理想情况下：每次划分，正好将分成两个等长的子序列，则

$$
\begin{aligned}
T(n) &\leqslant cn+2T(n/2) && c \text{ 是一个常数}\\
&\leqslant cn+2(cn/2+2T(n/4))=2cn+4T(n/4)\\
&\leqslant 2cn+4(cn/4+T(n/8))=3cn+8T(n/8)\\
&\cdots\cdots\\
&\leqslant cn\log_2 n+nT(1)=O(n\log_2 n)
\end{aligned}
$$

最坏情况下：即每次划分，只得到一个子序列，时效为 $O(n^2)$。

快速排序是通常被认为在同数量级（$O(n\log_2 n)$）的排序方法中平均性能最好的。但若初始序列按关键码有序或基本有序时，快排序反而蜕化为冒泡排序。为改进之，通常以“三者取中法”来选取支点记录，即将排序区间的两个端点与中点三个记录关键码居中的调整为支点记录。快速排序是一个不稳定的排序方法。

10.4 选择排序

选择排序主要是每一趟从待排序列中选取一个关键码最小的记录，也即第一趟从 n 个记录中选取关键码最小的记录，第二趟从剩下的 n-1 个记录中选取关键码最小的记录，直到整个序列的记录选完。这样，由选取记录的顺序，便得到按关键码有序的序列。

10.4.1 简单选择排序

操作方法：第一趟，从 n 个记录中找出关键码最小的记录与第一个记录交换；第二趟，从第二个记录开始的 n-1 个记录中再选出关键码最小的记录与第二个记录交换；如此，第 i 趟，则从第 i 个记录开始的 n-i+1 个记录中选出关键码最小的记录与第 i 个记录交换，直到整个序列按关键码有序。

【算法 10.9】

```
void  SelectSort(S_TBL *s)
{    for(i=1; i<s->length; i++)
     {    /* 作 length-1 趟选取 */
          for(j=i+1, t=i; j<=s->length; j++)
          {   /* 在 i 开始的 length-n+1 个记录中选关键码最小的记录 */
               if(s->elem[t].key>s->elem[j].key)
                    t=j;      /* t 中存放关键码最小记录的下标 */
          }
        s->elem[t]<-->s->elem[i];   /* 关键码最小的记录与第 i 个记录交换 */
     }
}
```

从程序中可看出，简单选择排序移动记录的次数较少，但关键码的比较次数依然是 $\frac{1}{2}n(n+1)$，所以时间复杂度仍为 $O(n^2)$。

10.4.2 树形选择排序

按照锦标赛的思想进行，将 n 个参赛的选手看成完全二叉树的叶结点，则该完全二叉树有 2n-2 或 2n-1 个结点。首先，两两进行比赛(在树中是兄弟的进行，否则轮空，直接进入下一轮)，胜出的再兄弟间再两两进行比较，直到产生第一名；接下来，将作为第一名的结点看成最差的，并从该结点开始，沿该结点到根路径上，依次进行各分枝结点子女间的比较，胜出的就是第二名。因为和他比赛的均是刚刚输给第一名的选手。如此，继续进行下去，直到所有选手的名次排定。

10.4.3 堆排序(Heap Sort)

设有 n 个元素的序列 k_1，k_2，…，k_n，当且仅当满足下述关系之一时，称之为堆。

若以一维数组存储一个堆，则堆对应一棵完全二叉树，且所有非叶结点的值均不大于(或不小于)其子女的值，根结点的值是最小(或最大)的。

设有 n 个元素，将其按关键码排序。首先将这 n 个元素按关键码建成堆，将堆顶元素输出，得到 n 个元素中关键码最小(或最大)的元素。然后，再对剩下的 n-1 个元素建成堆，输出堆顶元素，得到 n 个元素中关键码次小(或次大)的元素。如此反复，便得到一个按关键码有序的序列。称这个过程为堆排序。

因此，实现堆排序需解决两个问题：

1. 如何将 n 个元素的序列按关键码建成堆；
2. 输出堆顶元素后，怎样调整剩余 n-1 个元素，使其按关键码成为一个新堆。

首先，讨论输出堆顶元素后，对剩余元素重新建成堆的调整过程。

设树高为k，$k=\lfloor \log_2 n \rfloor+1$。从根到叶的筛选，关键码比较次数至多$2(k-1)$

调整方法：设有 m 个元素的堆，输出堆顶元素后，剩下 m-1 个元素。将堆底元素送入堆顶，堆被破坏，其原因仅是根结点不满足堆的性质。将根结点与左、右子女中较小(或小大)的进行交换。若与左子女交换，则左子树堆被破坏，且仅左子树的根结点不满足堆的性质；若与右子女交换，则右子树堆被破坏，且仅右子树的根结点不满足堆的性质。继续对不满足堆性质的子树进行上述交换操作，直到叶子结点，堆被建成。称这个自根结点到叶子结点的调整过程为筛选。

再讨论对 n 个元素初始建堆的过程。

建堆方法：对初始序列建堆的过程，就是一个反复进行筛选的过程。n 个结点的完全二叉树，则最后一个结点是第$\left\lfloor \frac{n}{2} \right\rfloor$个结点的子女。对第$\left\lfloor \frac{n}{2} \right\rfloor$个结点为根的子树筛选，使该子树成为堆，之后向前依次对各结点为根的子树进行筛选，使之成为堆，直到根结点。

堆排序：对 n 个元素的序列进行堆排序，先将其建成堆，以根结点与第 n 个结点交换；调整前 n-1 个结点成为堆，再以根结点与第 n-1 个结点交换；重复上述操作，直到整个序列有序。

【算法 10.10】

```
void HeapAdjust(S_TBL *h, int s, int m)
{/*r[s…m]中的记录关键码除r[s]外均满足堆的定义，本函数将对第s个结点为根的子树筛选，使其成为大顶堆*/
     rc=h->r[s];
     for(j=2*s; j<=m; j=j*2)  /* 沿关键码较大的子女结点向下筛选 */
     {    if(j<m&&h->r[j].key<h->r[j+1].key)
               j=j+1;  /* 为关键码较大的元素下标*/
          if(rc.key<h->r[j].key)  break;   /* rc 应插入在位置 s 上*/
          h->r[s]=h->r[j];  s=j;  /* 使 s 结点满足堆定义 */
     }
     h->r[s]=rc;       /* 插入 */
}

void HeapSort(S_TBL *h)
{    for(i=h->length/2; i>0; i--)          /* 将r[1..length]建成堆 */
          HeapAdjust(h, i, h->length);
     for(i=h->length; i>1; i--)
     {    h->r[1]<-->h->r[i];          /* 堆顶与堆低元素交换 */
          HeapAdjust(h, 1, i-1);         /*将r[1..i-1]重新调整为堆*/
     }
}
```

【效率分析】

次，交换记录至多 k 次。所以，在建好堆后，排序过程中的筛选次数不超过下式：

$2(\lfloor\log_2(n-1)\rfloor+\lfloor\log_2(n-2)\rfloor+\cdots+\log_2 2\rfloor) < 2n\log_2 n$

而建堆时的比较次数不超过 4n 次，因此堆排序最坏情况下，时间复杂度也为 $O(n\log_2 n)$。

10.5 二路归并排序

二路归并排序的基本操作是将两个有序表合并为一个有序表。

设 r[u…t]由两个有序子表 r[u…v-1]和 r[v…t]组成，两个子表长度分别为 v-u、t-v+1。合并方法为：

⑴ i=u; j=v; k=u; //置两个子表的起始下标及辅助数组的起始下标

⑵ 若 i>v 或 j>t，转⑷ //其中一个子表已合并完，比较选取结束

⑶ //选取 r[i]和 r[j]关键码较小的存入辅助数组 rf

如果 r[i].key<r[j].key，rf[k]=r[i]; i++; k++; 转⑵

否则，rf[k]=r[j]; j++; k++; 转⑵

⑷ //将尚未处理完的子表中元素存入 rf

如果 i<v，将 r[i…v-1]存入 rf[k…t] //前一子表非空

如果 j<=t，将 r[i…v]存入 rf[k…t] //后一子表非空

⑸ 合并结束。

【算法 10.11】

```
void    Merge(ElemType *r, ElemType *rf, int u, int v, int t)
{
     for(i=u, j=v, k=u; i<v&&j<=t; k++)
     {    if(r[i].key<r[j].key)
          {    rf[k]=r[i]; i++; }
          else
          {    rf[k]=r[j]; j++; }
     }
     if(i<v) rf[k…t]=r[i…v-1];
     if(j<=t) rf[k…t]=r[j…t];
}
```

一.两路归并的迭代算法

1 个元素的表总是有序的。所以对 n 个元素的待排序列，每个元素可看成 1 个有序子表。对子表两两合并生成$\left\lceil\frac{n}{2}\right\rceil$个子表，所得子表除最后一个子表长度可能为 1 外，其余子表长度均为 2。再进行两两合并，直到生成 n 个元素按关键码有序的表。

【算法 10.12】

```
void MergeSort(S_TBL *p, ElemType *rf)
{    /*对*p 表归并排序，*rf 为与*p 表等长的辅助数组*/
ElemType *q1, *q2;
     q1=rf; q2=p->elem;
     for(len=1; len<p->length; len=2*len)  /*从 q2 归并到 q1*/
     {    for(i=1; i+2*len-1<=p->length; i=i+2*len)
               Merge(q2, q1, i, i+len, i+2*len-1);    /*对等长的两个子表合并*/
          if(i+len-1<p->length)
               Merge(q2, q1, i, i+len, p->length);    /*对不等长的两个子表合并*/
          else      if(i<=p->length)
                         while(i<=p->length)      /*若还剩下一个子表，则直接传入*/
                              q1[i]=q2[i];
          q1<-->q2;          /*交换，以保证下一趟归并时，仍从 q2 归并到 q1*/
          if(q1!=p->elem)  /*若最终结果不在*p 表中，则传入之*/
               for(i=1; i<=p->length; i++)
                    p->elem[i]=q1[i];
```

```
    }
}
```

二.两路归并的递归算法

【算法 10.13】

```
void MSort(ElemType *p, ElemType *p1, int s, int t)
{   /*将 p[s…t]归并排序为 p1[s…t]*/
    if(s==t) p1[s]=p[s]
    else
    {   m=(s+t)/2;     /*平分*p 表*/
        MSort(p, p2, s, m);      /*递归地将 p[s…m]归并为有序的 p2[s…m]*/
        MSort(p, p2, m+1, t);    /*递归地将 p[m+1…t]归并为有序的 p2[m+1…t]*/
        Merge(p2, p1, s, m+1, t);    /*将 p2[s…m]和 p2[m+1…t]归并到 p1[s…t]*/
    }
}


void MergeSort(S_TBL *p)
{   /*对顺序表*p 作归并排序*/
    MSort(p->elem, p->elem, 1, p->length);
}
```

【效率分析】

需要一个与表等长的辅助元素数组空间，所以空间复杂度为 O(n)。

对 n 个元素的表，将这 n 个元素看作叶结点，若将两两归并生成的子表看作它们的父结点，则归并过程对应由叶向根生成一棵二叉树的过程。所以归并趟数约等于二叉树的高度-1，即 $\log_2 n$，每趟归并需移动记录 n 次，故时间复杂度为 $O(n\log_2 n)$。

10.6 基数排序

基数排序是一种借助于多关键码排序的思想，是将单关键码按基数分成“多关键码”进行排序的方法。

10.6.1 多关键码排序

扑克牌中 52 张牌，可按花色和面值分成两个字段，其大小关系为：

花色：　　梅花 < 方块 < 红心 < 黑心

面值：　　2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A

若对扑克牌按花色、面值进行升序排序，得到如下序列：

梅花 2,3,...,A，方块 2,3,...,A，红心 2,3,...,A，黑心 2,3,...,A

即两张牌，若花色不同，不论面值怎样，花色低的那张牌小于花色高的，只有在同花色情况下，大小关系才由面值的大小确定。这就是多关键码排序。

为得到排序结果，我们讨论两种排序方法。

方法 1：先对花色排序，将其分为 4 个组，即梅花组、方块组、红心组、黑心组。再对每个组分别按面值进行排序，最后，将 4 个组连接起来即可。

方法 2：先按 13 个面值给出 13 个编号组(2 号，3 号，...，A 号)，将牌按面值依次放入对应的编号组，分成 13 堆。再按花色给出 4 个编号组(梅花、方块、红心、黑心)，将 2 号组中牌取出分别放入对应花色组，再将 3 号组中牌取出分别放入对应花色组，……，这样，4 个花色组中均按面值有序，然后，将 4 个花色组依次连接起来即可。

设 n 个元素的待排序列包含 d 个关键码$\{k^1, k^2, \cdots, k^d\}$，则称序列对关键码$\{k^1, k^2, \cdots, k^d\}$有序是指：对于序列中任两个记录 r[i]和 r[j](1≤i≤j≤n)都满足下列有序关系：

$$(k_i^1, k_i^2, \cdots, k_i^d) < (k_j^1, k_j^2, \cdots, k_j^d)$$

其中 k^1称为最主位关键码，k^d称为最次位关键码。

多关键码排序按照从最主位关键码到最次位关键码或从最次位到最主位关键码的顺序

逐次排序，分两种方法：

最高位优先(Most Significant Digit first)法，简称 MSD 法：先按 k^1排序分组，同一组中记录，关键码 k^1相等，再对各组按 k^2排序分成子组，之后，对后面的关键码继续这样的排序分组，直到按最次位关键码 k^d对各子组排序后。再将各组连接起来，便得到一个有序序列。扑克牌按花色、面值排序中介绍的方法一即是 MSD 法。

最低位优先(Least Significant Digit first)法，简称 LSD 法：先从 k^d开始排序，再对 k^{d-1}进行排序，依次重复，直到对 k^1排序后便得到一个有序序列。扑克牌按花色、面值排序中介绍的方法二即是 LSD 法。

10.6.2 链式基数排序

将关键码拆分为若干项，每项作为一个“关键码”，则对单关键码的排序可按多关键码排序方法进行。比如，关键码为 4 位的整数，可以每位对应一项，拆分成 4 项；又如，关键码由 5 个字符组成的字符串，可以每个字符作为一个关键码。由于这样拆分后，每个关键码都在相同的范围内(对数字是 0～9，字符是'a'～'z')，称这样的关键码可能出现的符号个数为“基”，记作 RADIX。上述取数字为关键码的“基”为 10；取字符为关键码的“基”为 26。基于这一特性，用 LSD 法排序较为方便。

基数排序：从最低位关键码起，按关键码的不同值将序列中的记录“分配”到 RADIX 个队列中，然后再“收集”之。如此重复 d 次即可。链式基数排序是用 RADIX 个链队列作为分配队列，关键码相同的记录存入同一个链队列中，收集则是将各链队列按关键码大小顺序链接起来。

【算法 10.14】

```
#define   MAX_KEY_NUM    8     /*关键码项数最大值*/
#define   RADIX     10         /*关键码基数，此时为十进制整数的基数*/
#define   MAX_SPACE 1000       /*分配的最大可利用存储空间*/
typedef   struct{
     KeyType        keys[MAX_KEY_NUM]; /*关键码字段*/
     InfoType  otheritems;         /*其它字段*/
     int       next;          /*指针字段*/
     }NodeType;     /*表结点类型*/
typedef   struct{
     NodeType  r[MAX_SPACE];      /*静态链表，r[0]为头结点*/
     int  keynum;              /*关键码个数*/
     int  length;              /*当前表中记录数*/
          }L_TBL;              /*链表类型*/
typedef   int  ArrayPtr[radix];    /*数组指针，分别指向各队列*/

void  Distribute(NodeType *s, int i, ArrayPtr *f, ArrayPtr *e)
{    /*静态链表 ltbl 的 r 域中记录已按(kye[0], keys[1], …, keys[i-1])有序)*/
     /*本算法按第 i 个关键码 keys[i]建立 RADIX 个子表，使同一子表中的记录的 keys[i]相同*/
     /*f[0…RADIX-1]和 e[0…RADIX-1]分别指向各子表的第一个和最后一个记录*/
     for(j=0; j<RADIX; j++) f[j]=0;      /* 各子表初始化为空表*/
     for(p=r[0].next; p; p=r[p].next)
     {    j=ord(r[p].keys[i]);    /*ord 将记录中第 i 个关键码映射到[0…RADIX-1]*/
          if(!f[j])  f[j]=p;
     else   r[e[j]].next=p;
          e[j]=p;              /* 将 p 所指的结点插入到第 j 个子表中*/
     }
}

void  Collect(NodeType *r, int i, ArrayPtr f, ArrayPtr e)
{/*本算法按keys[i]自小到大地将f[0…RADIX-1]所指各子表依次链接成一个链表*e[0…RADIX-1]为各子表的尾指针*/
     for(j=0; !f[j]; j=succ(j));      /*找第一个非空子表，succ 为求后继函数*/
     r[0].next=f[j]; t=e[j];     /*r[0].next 指向第一个非空子表中第一个结点*/
     while(j<RADIX)
     {    for(j=succ(j); j<RADIX-1&&!f[j]; j=succ(j));  /*找下一个非空子表*/
```

```
        if(f[j])  {r[t].next=f[j]; t=e[j]; } /*链接两个非空子表*/
    }
    r[t].next=0;   /*t 指向最后一个非空子表中的最后一个结点*/
}
void  RadixSort(L_TBL *ltbl)
{   /*对 ltbl 作基数排序，使其成为按关键码升序的静态链表，ltbl->r[0]为头结点*/
    for(i=0; i<ltbl->length; i++)  ltbl->r[i].next=i+1;
    ltbl->r[ltbl->length].next=0;     /*将 ltbl 改为静态链表*/
    for(i=0; i<ltbl->keynum; i++) /*按最低位优先依次对各关键码进行分配和收集*/
    {   Distribute(ltbl->r, i, f, e);   /*第 i 趟分配*/
        Collect(ltbl->r, i, f, e);    /*第 i 趟收集*/
    }
}
```

由于 $\frac{k-1}{\log k}$ 随 k 的增加而增长，则内部归并的时间增大

行$\lfloor \log_2 k \rfloor$次比较，从而使总的归并时间变为

【效率分析】

时间效率:设待排序列为 n 个记录,d 个关键码,关键码的取值范围为 radix,则进行链式基数排序的时间复杂度为 O(d(n+radix))，其中，一趟分配时间复杂度为 O(n)，一趟收集时间复杂度为 O(radix)，共进行 d 趟分配和收集。

空间效率：需要 2*radix 个指向队列的辅助空间，以及用于静态链表的 n 个指针。

10.7 外排序

10.7.1 外部排序的方法

外部排序基本上由两个相互独立的阶段组成。首先，按可用内存大小，将外存上含 n 个记录的文件分成若干长度为 k 的子文件或段(segment)，依次读入内存并利用有效的内部排序方法对它们进行排序，并将排序后得到的有序子文件重新写入外存。通常称这些有序子文件为归并段或顺串；然后，对这些归并段进行逐趟归并，使归并段(有序子文件)逐渐由小到大，直至得到整个有序文件为止。

10.7.2 多路平衡归并的实现

从上式可见，增加 k 可以减少 s，从而减少外存读/写的次数。但是，从下面的讨论中又可发现，单纯增加 k 将导致增加内部归并的时间 ut_{mg}。那末，如何解决这个矛盾呢?

先看 2-路归并。令 u 个记录分布在两个归并段上，按 Merge 函数进行归并。每得到归并后的含 u 个记录的归并段需进行 u-1 次比较。

再看 k-路归并。令 u 个记录分布在 k 个归并段上，显然，归并后的第一个记录应是 k 个归并段中关键码最小的记录，即应从每个归并段的第一个记录的相互比较中选出最小者，这需要进行 k-1 次比较。同理，每得到归并后的有序段中的一个记录，都要进行 k-1 次比较。显然，为得到含 u 个记录的归并段需进行(u-1)(k-1)次比较。由此，对 n 个记录的文件进行外部排序时，在内部归并过程中进行的总的比较次数为 s(k-1)(n-1)。假设所得初始归并段为 m 个，则可得内部归并过程中进行比较的总的次数为

$$\lceil \log_k m \rceil (k-1)(n-1)t_{mg} = \left\lceil \frac{\log_2 m}{\log_2 k} \right\rceil (k-1)(n-1)t_{mg}$$

k 而减少外存信息读写时间所得效益，这是我们所不希望的。然而，若在进行 k-路归并时利用“败者树”(Tree of Loser)，则可使在 k 个记录中选出关键码最小的记录时仅需进

它不再随 k 的增长而增长。

何谓“败者树”？它是树形选择排序的一种变型。相对地，我们可称图 10.5 和图 10.6 中二叉树为“胜者树”，因为每个非终端结点均表示其左、右子女结点中“胜者”。反之，若在双亲结点中记下刚进行完的这场比赛中的败者，而让胜者去参加更高一层的比赛，便可得到一棵“败者树”。