# 数据结构

## （C 语言版）（第 2 版）

## 习题解析

揭安全　　李云清　　杨庆红


江西师范大学计算机信息工程学院

联系方式：janquan@163.com

（习题答案仅供参考）

# 第 1 章　绪论

1.1 什么是数据结构？

【答】：数据结构是指按一定的逻辑结构组成的一批数据，使用某种存储结构将这批数据存储于计算机中，并在这些数据上定义了一个运算集合。

1.2 数据结构涉及哪几个方面？

【答】：数据结构涉及三个方面的内容，即数据的逻辑结构、数据的存储结构和数据的运算集合。

1.3 两个数据结构的逻辑结构和存储结构都相同，但是它们的运算集合中有一个运算的定义不一样，它们是否可以认作是同一个数据结构？为什么？

【答】：不能，运算集合是数据结构的重要组成部分，不同的运算集合所确定的数据结构是不一样的，例如，栈与队列它们的逻辑结构与存储结构可以相同，但由于它们的运算集合不一样，所以它们是两种不同的数据结构。

1.4 线性结构的特点是什么？非线性结构的特点是什么？

【答】：线性结构元素之间的关系是一对一的，在线性结构中只有一个开始结点和一个终端结点，其他的每一个结点有且仅有一个前驱和一个后继结点。而非线性结构则没有这个特点，元素之间的关系可以是一对多的或多对多的。

1.5 数据结构的存储方式有哪几种？

【答】：数据结构的存储方式有顺序存储、链式存储、散列存储和索引存储等四种方式。

1.6 算法有哪些特点？它和程序的主要区别是什么？

【答】：算法具有（1）有穷性（2）确定性（3）0 个或多个输入（4）1 个或多个输出（5）可行性等特征。程序是算法的一种描述方式，通过程序可以在计算机上实现算法。

1.7 抽象数据类型的是什么？它有什么特点？

【答】：抽象数据类型是数据类型的进一步抽象，是大家熟知的基本数据类型的延伸和发展。抽象数据类型是与表示无关的数据类型，是一个数据模型及定义在该模型上的一组运算。对一个抽象数据类型进行定义时，必须给出它的名字及各运算的运算符名，即函数名，并且规定这些函数的参数性质。一旦定义了一个抽象数据类型及具体实现，程序设计中就可以像使用基本数据类型那样，十分方便地使用抽象数据类型。抽象数据类型的设计者根据这些描述给出操作的具体实现，抽象数据类型的使用者依据这些描述使用抽象数据类型。

1.8 算法的时间复杂度指的是什么？如何表示？

【答】：算法执行时间的度量不是采用算法执行的绝对时间来计算的，因为一个算法在不同的机器上执行所花的时间不一样，在不同时刻也会由于计算机资源占用情况的不同，使得算法在同一台计算机上执行的时间也不一样，另外，算法执行的时间还与输入数据的状态有关，所以对于算法的时间复杂性，采用算法执行过程中其基本操作的执行次数，称为计算量来度量。算法中基本操作的执行次数一般是与问题规模有关的，对于结点个数为 n 的数据处理问题，用 T(n)表示算法基本操作的执行次数。为了评价算法的执行效率，通常采用大写 O 符号表示算法的时间复杂度，大写 O 符号给出了函数 f 的一个上限。其它义如下：

定义：f (n)=O (g (n)) 当且仅当存在正的常数 c 和 $n_0$，使得对于所有的 $n \geq n_0$，有 $f(n) \leq c\, g(n)$。

上述定义表明，函数 f 顶多是函数 g 的 c 倍，除非 n 小于 $n_0$。因此对于足够大的 n (如 $n \geq n_0$)，g 是 f 的一个上限（不考虑常数因子 c）。在为函数 f 提供一个上限函数 g 时，通常使用比较简单的函数形式。比较典型的形式是含有 n 的单个项（带一个常数系数）。表 1-1 列出了一些常用的 g 函数及其名称。对于表 1-1 中的对数函数 logn，没有给出对数基，原因是对于任何大于 1 的常数 a 和 b 都有 $\log_a n = \log_b n / \log_b a$,所以 $\log_a n$ 和 $\log_b n$ 都有一个相对的乘法系数 $1/\log_b a$，其中 a 是一个常量。

表 1-1 常用的渐进函数

| 函　数 | 名　称 |
|---|---|
| 1 | 常数 |
| *logn* | 对数 |
| *n* | 线性 |
| *nlogn* | *n*个*logn* |
| $n^2$ | 平方 |
| $n^3$ | 立方 |
| $2^n$ | 指数 |
| *n*! | 阶乘 |

1.9 算法的空间复杂度指的是什么？如何表示？

【答】：算法的空间复杂度是指算法在执行过程中占用的额外的辅助空间的个数。可以将它表示为问题规模的函数，并通过大写O符号表示空间复杂度。

1.10 对于下面的程序段，分析带下划线的语句的执行次数，并给出它们的时间复杂度 T(n)。

（1） i++;

（2） for(i=0;i<n;i++)
　　if (a[i]<x) x=a[i];

（3）for(i=0;i<n;i++)
　　for(j=0;j<n;j++)
　　printf("%d",i+j);

（4）for (i=1;i<=n-1;i++)
　{ k=i;
　　for(j=i+1;j<=n;j++)
　　if(a[j]>a[j+1]) k=j;
　　t=a[k]; a[k]=a[i]; a[i]=t;
　}

（5）for(i=0;i<n;i++)
　　for(j=0;j<n;j++)
　　{++x;s=s+x;}

【答】：（1）O（1）；（2）O（n）；（3）O（$n^2$）；（4）O（n）；（5）O（$n^2$）

# 第 2 章　线性表及其顺序存储

2.1　选择题

（1）表长为 $n$ 的顺序存储的线性表，当在任何位置上插入或删除一个元素的概率相等时，插入一个元素所需移动元素的平均个数为（　E　），删除一个元素所需移动元素的平均个数为（　A　）。

A．$(n-1)/2$　　B．$n$　　C．$n+1$　　D．$n-1$

E．$n/2$　　F．$(n+1)/2$　　G．$(n-2)/2$

（2）设栈 $S$ 和队列 $Q$ 的初始状态为空，元素 $e1$、$e2$、$e3$、$e4$、$e5$ 和 $e6$ 依次通过栈 $S$，一个元素出栈后即进入队列 $Q$，若 6 个元素出队的序列为 $e2$、$e4$、$e3$、$e6$、$e5$ 和 $e1$，则栈 $S$ 的容量至少应该为（　C　）。

A．6　　B．4　　C．3　　D．2

（3）设栈的输入序列为 1、2、3 …$n$，若输出序列的第一个元素为 $n$，则第 $i$ 个输出的元素为（　B　）。

A．不确定　　B．$n-i+1$　　C．$i$　　D．$n-i$

（4）在一个长度为 n 的顺序表中删除第 i 个元素（1<=$i$<=$n$）时，需向前移动（　A　）个元素。

A．$n-i$　　B．$n-i+1$　　C．$n-i-1$　　D．$i$

（5）若长度为 $n$ 的线性表采用顺序存储结构存储，在第 $i$ 个位置上插入一个新元素的时间复杂度为（　A　）。

A．$O(n)$　　B．$O(1)$　　C．$O(n^2)$　　D．$O(n^3)$

（6）表达式 a*(b+c)−d 的后缀表达式是（　B　）。

A．*abcd*+−　　B．*abc*+**d*−　　C．*abc*+**d*−　　D．−+**abcd*

（7）队列是一种特殊的线性表，其特殊性在于（　C　）。

A．插入和删除在表的不同位置执行　B．插入和删除在表的两端位置执行

C．插入和删除分别在表的两端执行　D．插入和删除都在表的某一端执行

（8）栈是一种特殊的线性表，具有（　B　）性质。

A．先进先出　　B．先进后出　　C．后进后出　　D．顺序进出

（9）顺序循环队列中（数组的大小为 $n$），队头指示 front 指向队列的第 1 个元素，队尾指示 rear 指向队列最后元素的后 1 个位置，则循环队列中存放了 $n-1$ 个元素，即循环队列满的条件为（　B　）。

A．(rear + 1)%$n$ = front − 1　　B．(rear + 1)%$n$ = front

C．(rear)%$n$ = front　　D．rear + 1 = front

（10）顺序循环队列中（数组的大小为 6），队头指示 front 和队尾指示 rear 的值分别为 3 和 0，当从队列中删除 1 个元素，再插入 2 个元素后，front 和 rear 的值分别为（　D　）。

A．5 和 1　　B．2 和 4　　C．1 和 5　　D．4 和 2

2.2 什么是顺序表？什么是栈？什么是队列？

【答】：当线性表采用顺序存储结构时，即为顺序表。栈是一种特殊的线性表，它的特殊性表现在约定了在这种线性表中数据的插入与删除操作只能在这种线性表的同一端进行(即栈顶)，因此，栈具有先进后出、后进先出的特点。队列也是一种特殊的线性表，它的特殊性表现在约定了在这种线性表中数据的插入在表的一端进行，数据的删除在表的另一端进行，因此队列具有先进先出，后进后出的特点。

2.3 设计一个算法，求顺序表中值为 x 的结点的个数。

【答】：顺序表的存储结构定义如下（文件名 seqlist.h）：

```
#include <stdio.h>
#define N 100                   /*预定义最大的数据域空间*/
typedef    int datatype;        /*假设数据类型为整型*/
typedef struct {
      datatype data[N];         /*此处假设数据元素只包含一个整型的关键字域*/
      int length;               /*线性表长度*/
} seqlist;                      /*预定义的顺序表类型*/
```

算法 countx（L,x）用于求顺序表 L 中值为 x 的结点的个数。

```
int countx(seqlist *L,datatype x)
{     int c=0;
      int i;
      for (i=0;i<L->length;i++)
          if (L->data[i]==x) c++;
      return c;
}
```

2.4 设计一个算法，将一个顺序表倒置。即，如果顺序表各个结点值存储在一维数组 a 中，倒置的结果是使得数组 a 中的 a[0]等于原来的最后一个元素，a[1] 等于原来的倒数第 2 个元素，…，a 的最后一个元素等于原来的第一个元素。

【答】：顺序表的存储结构定义同题 2.3，实现顺序表倒置的算法程序如下：

```
void verge(seqlist *L)
{int t,i,j;
 i=0;
 j=L->length-1;
 while (i<j)
       {   t=L->data[i];
           L->data[i++]=L->data[j];
           L->data[j--]=t;
       }
}
```

2.5 已知一个顺序表中的各结点值是从小到大有序的，设计一个算法，插入一个值为 x 的结点，使顺序表中的结点仍然是从小到大有序。

【答】：顺序表的定义同题 2.3，实现本题要求的算法程序如下：

```
void insertx(seqlist *L,datatype x)
{int j;
 if (L->length<N)
 {   j=L->length-1;
     while (j>=0 && L->data[j]>x)
           {   L->data[j+1]=L->data[j];
               j--;
           }
     L->data[j+1]=x;
     L->length++;
 }
}
```

2.6 将下列中缀表达式转换为等价的后缀表达式。

(1) 5+6*7
(2) (5-6)/7
(3) 5-6*7*8
(4) 5*7-8
(5) 5*(7-6)+8/9
(6) 7*(5-6*8)-9

【答】:

(7) 5+6*7　　后缀表达式：5 6 7*+
(8) (5-6)/7　　后缀表达式：5 6-7/
(9) 5-6*7*8　　后缀表达式：5 6 7*8*-
(10)5*7-8　　后缀表达式：5 7* 8-
(11)5*(7-6)+8/9　　后缀表达式：5 7 6-*8 9/+
(12)7*(5-6*8)-9　　后缀表达式：7 5 6 8*-*9-

2.7 循环队列存储在一个数组中，数组大小为 n，队首指针和队尾指针分别为 front 和 rear，请写出求循环队列中当前结点个数的表达式。

【答】：循环队列中当前结点个数的计算公式是：(n+rear-front)%n

2.8 编号为 1,2,3,4 的四列火车通过一个栈式的列车调度站，可能得到的调度结果有哪些？如果有 n 列火车通过调度站，请设计一个算法，输出所有可能的调度结果。

【答】:

解题思路：栈具有先进后出、后进先出的特点，因此，任何一个调度结果应该是 1，2，3，4 全排列中的一个元素。由于进栈的顺序是由小到大的，所以出栈序列应该满足以下条件：对于序列中的任何一个数其后面所有比它小的数应该是倒序的，例如 4321 是一个有效的出栈序列，1423 不是一个有效的出栈结果（4 后面比它小的两个数 2，3 不是倒序）。据此，本题可以通过算法产生 n 个数的全排列，然后将满足出栈规则的序列输出。

产生 n 个数的全排列有递归与非递归两种实现算法。

产生全排列的递归算法：

设 R={$r_1$,$r_2$,…,$r_n$}是要进行排列的 n 个元素，$R_i$=R-{$r_i$}。集合 X 中元素的全排列记为 perm(X)。($r_i$)perm(X)表示在全排列 perm(X)的每一个排列前加上前缀 $r_i$ 得到的排列。R 的全排列可归纳定义如下：

当 n=1 时，perm(R)=(r)，其中 r 是集合 R 中惟一的元素；

当 n>1 时，perm(R)由($r_1$)perm($R_1$),($r_2$)perm($R_2$),…,($r_n$)perm($R_n$)构成。

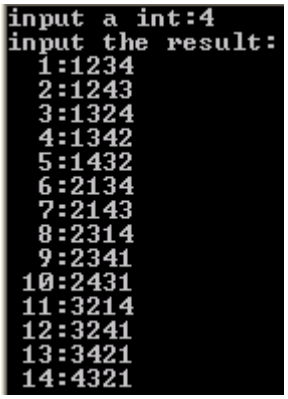
依此递归定义，可设计产生 perm(R)的递归算法如下：

**递归解法：(2_8_1.c)**

```
#include<stdio.h>
int cont=1;             /*全局变量，用于记录所有可能的出栈序列个数*/
void print(int str[],int n);
/*求整数序列 str[]从 k 到 n 的全排列*/
void perm(int str[],int k,int n)
{int i,temp;
   if (k==n-1) print(str,n);
     else
    { for (i=k;i<n;i++)
      {temp=str[k];        str[k]=str[i];        str[i]=temp;
       perm(str,k+1,n);   /*递归调用*/
       temp=str[i];        str[i]=str[k];        str[k]=temp;
       }
      }
}
/*本函数判断整数序列 str[]是否满足进出栈规则,若满足则输出*/
void print(int str[],int n)
{int i,j,k,l,m,flag=1,b[2];
   for(i=0;i<n;i++)        /*对每个 str[i]判断其后比它小的数是否为降序序列*/
    { m=0;
      for(j=i+1;j<n&&flag;j++)
           if (str[i]>str[j]) {if (m==0) b[m++]=str[j];
                                  else {if (str[j]>b[0]) {flag=0;}
                                            else b[0]=str[j];
                                       }
                                }
    }
   if(flag)                 /*满足出栈规则则输出 str[]中的序列*/
     {    printf(" %2d:",cont++);
          for(i=0;i<n;i++)
               printf("%d",str[i]);
          printf("\n");
```

```
    }
}
int main()
{int str[100],n,i;
 printf("input a int:"); /*输出排列的元素个数*/
 scanf("%d",&n);
 for(i=0;i<n;i++)   /*初始化排列集合*/
     str[i]=i+1;
 printf("input the result:\n");
 perm(str,0,n);
 printf("\n");
 return 0;
}
```

当参与进出栈的元素个数为 4 时，输出的结果如下图所示。




该算法执行的时间复杂度为 O（n!）。随着 n 的增大，算法的执行效率非常的低。

**非递归解法：（2_7_8.c）**

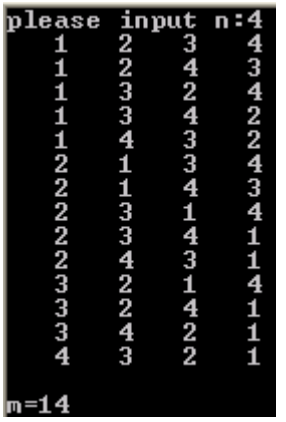
对一组数穷尽所有排列，还可一种更直接的方法，将一个排列看作一个长整数，则所有排列对应着一组整数，将这组整数按从小到大的顺序排成一个数列，从对应最小的整数开始，按数列的递增顺序逐一列举每个排列对应的每一个整数，这能更有效地完成排列的穷举。从一个排列找出对应数列的下一个排列可在当前排列的基础上作部分调整来实现。倘若当前排列为 1,2,4,6,5,3，并令其对应的长整数为 124653。要寻找比长整数 124653 更大的排列，可从该排列的最后一个数字顺序向前逐位考察，当发现排列中的某个数字比它前一个数字大时，如本例中的 6 比它的前一位数字 4 大，则说明还有可能对应更大整数的排列。但为顺序从小到大列举出所有的排列，不能立即调整得太大，如本例中将数字 6 与数字 4 交换得到的排列为 126453 就不是排列 124653 的下一个排列。为得到排列 124653 的下一个排列，应从已考察过的那部分数字中选出比数字 4 大，但又是它们中最小的那一个数字，比如数字 5，与数字 4 交换。该数字也是从后向前考察过程中第一个比 4 大的数字，5 与 4 交换后，得到排列 125643。在前面数字 1，2，5 固定的情况下，还应选择对应最小整数的那个排列，为此还需将后面那部分数字的排列颠倒，如将数字 6，4，3 的排列顺序颠倒，得到排列 1,2,5,3,4,6，这才是排列 1，2，4，6，

5，3 的下一个排列。按照以上想法可以编写非递归程序实现 n 个数的全排列，对满足进出栈规则的排列则计数并输出。

```
/*本程序输出 1 2 ... n 个序列进栈出栈的序列*/
#include <stdio.h>
int pl(int n)
{ int a[100];                          /*最大处理范围为 99 个数*/
   int flag=1,flag1=0;
   FILE *rf ;
   int i,j,k,x,count=0;
   rf = fopen("stack.txt", "w") ;      /*pl.txt 用于存放进出栈序列结果*/
   for (i=1;i<=n;i++)                  /*初始序列*/
              a[i]=i;
   while (flag)                        /*   还剩余未输出的排列*/
     {   flag1=1;                      /*   判断本次排列是否符合进栈出栈序列   */
         for (i=1;i<=n;i++)
               {   j=i+1;
                   while (j<=n && a[j]>a[i]) j++;     /*  找 a[i]后第一个比 a[i]小的元素 a[j]*/
                   k=j+1;
                   while (k<=n)    /*  如果 a[j]后还有比 a[i]小且比 a[j]大的元素,则此排列无效*/
                        {if ( a[k] <a[i]    && a[k]>a[j])    flag1=0;
                         k++;
                        }
               }
         if (flag1)
         { for (i=1;i<=n;i++)              /*输出当前排列*/
           {   printf("%4d",a[i]); fprintf(rf,"%4d",a[i]);}
               printf("\n"); fprintf(rf,"\n");
               count++;                  /*计数器加 1*/
            }
            i=n;                         /*从后向前找相邻位置后大前小的元素值*/
            while (i>1 && a[i]<a[i-1])   i--;
            if (i==1)    flag=0;     /*未找到则结束*/
              else
                {j=i-1;i=n;/*  若找到,则在该位置的后面从右向左找第一个比该元素大的值*/
                while (i>j && a[i]<a[j]) i--;
                  k=a[j];                    /*交换两元素的值*/
                 a[j]=a[i];
                 a[i]=k;
                  k=j+1;                     /*对交换后后面的数据由小到大排序*/
```

```
                for ( i=k+1;i<=n;i++)     /*插入排序*/
                    {j=i-1;
                     x=a[i];
                     while (j>=k && x<a[j])    { a[j+1]=a[j]; j--;}
                     a[j+1]=x;
                     }
                }
    }
    fclose(rf);
    return count;            /*返回排列总个数*/
}
void main()
{int n,m=0;
 printf("please input n:");          /*输入排列规模*/
 scanf("%d",&n);
 m=pl(n);
 printf("\nm=%d",m);                /*输出满足进出栈的排列总个数*/
}
```

程序运行时如果输入 4，则输出的结果如下图所示。




该算法的时间复杂度也是 O(n!)。

结论：如果 n 个数按编号由小到大的顺序进栈，进栈的过程中可以出栈，则所有可能的出栈序列的总数为：$\frac{(2n)!}{(n+1)n!n!}$

# 第 3 章　线性表的链式存储

3.1　选择题

（1）两个有序线性表分别具有 $n$ 个元素与 $m$ 个元素且 $n \le m$，现将其归并成一个有序表，其最少的比较次数是（　A　）。

A．$n$　　B．$m$　　C．$n-1$　　D．$m+n$

（2）非空的循环单链表 head 的尾结点（由 p 所指向）满足（　C　）。

A．p->next==NULL　B．p==NULL C．p->next==head D．p==head

（3）在带头结点的单链表中查找 $x$ 应选择的程序体是（　C　）。

A．node *p=head->next;　while (p && p->info!=x)　p=p->next;
　　if (p->info==x)　　return p else return NULL;

B．node *p=head;　　while (p&& p->info!=x) p=p->next;　return p;

C．node *p=head->next;　while (p&&p->info!=x) p=p->next;　return　p;

D．node *p=head;　　while (p->info!=x) p=p->next ;　　return p;

（4）线性表若采用链式存储结构时，要求内存中可用存储单元的地址（　D　）。

A．必须是连续的　　B．部分地址必须是连续的

C．一定是不连续的　　D．连续不连续都可以

（5）在一个具有 $n$ 个结点的有序单链表中插入一个新结点并保持单链表仍然有序的时间复杂度是（　B　）。

A．$O(1)$　　B．$O(n)$　　C．$O(n^2)$　　D．$O(n\log_2^n)$

（6）用不带头结点的单链表存储队列时，其队头指针指向队头结点，其队尾指针指向队尾结点，则在进行删除操作时（　D　）。

A．仅修改队头指针　　B．仅修改队尾指针

C．队头、队尾指针都要修改　　D．队头,队尾指针都可能要修改

（7）若从键盘输入 $n$ 个元素，则建立一个有序单向链表的时间复杂度为（　B　）。

A．$O(n)$　　B．$O(n^2)$　　C．$O(n^3)$　　D．$O(n \times \log 2^n)$

（8）下面哪个术语与数据的存储结构无关（　D　）。

A．顺序表　　B．链表　　C．散列表　　D．队列

（9）在一个单链表中，若删除 p 所指结点的后续结点，则执行（　A　）。

A．p->next=p->next->next;　　B．p=p->next; p->next=p->next->next;

C．p->next=p->next;　　D．p =p->next->next;

（10）在一个单链表中，若 p 所指结点不是最后结点，在 p 之后插入 s 所指结点，则执行（ B ）。

A．s->next=p;p->next=s;　　B．s->next=p->next;p->next=s;

C．s->next=p->next;p=s;　　D．p->next=s;s->next=p;

3.2 设计一个算法，求一个单链表中的结点个数。

【答】：单链表存储结构定义如下（相关文件：linklist.h）

```
#include <stdlib.h>
```

```
#include <stdio.h>
typedef struct node
{    int data;
     struct node *next;
}linknode;
typedef linknode *linklist;
/*尾插法创建带头结点的单链表*/
linklist creatlinklist()
{    linklist head,r,x,s;
     head=r=(linklist)malloc(sizeof(linknode));
     printf("\n 请输入一组以 0 结束的整数序列：\n");
     scanf("%d",&x);
     while (x)
     {    s=(linklist)malloc(sizeof(linknode));
          s->data=x;
          r->next=s;
          r=s;
          scanf("%d",&x);
     }
     r->next=NULL;
     return head;
}
/*输出带头结点的单链表*/
void print(linklist head)
{    linklist p;
     p=head->next;
     printf("List is:\n");
     while(p)
     {    printf("%5d",p->data);
          p=p->next;
     }
     printf("\n");
}
```

基于上述结构定义，求单链表中的结点个数的算法程序如下：

```
int count(linklist head)
{    int c=0;
     linklist p=head;
     while (p)
     {c++;
```

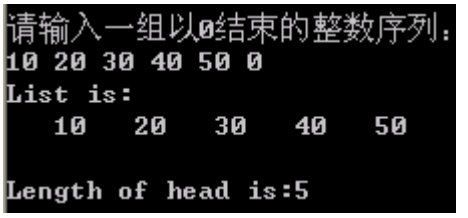
```
        p=p->next;
      }
      return c;
}
```

3.3 设计一个算法，求一个带头结点单链表中的结点个数。

【答】：带头结点的单链表的存储结构定义同题 3.2，实现本题功能的算法程序如下（3_3.c）

```
#include "linklist.h"
int count(linklist head)
{     int c=0;
      linklist p=head->next;
      while (p)
      {c++;
       p=p->next;
      }
      return c;
}
main()          /*测试函数*/
{linklist head;
 head=creatlinklist();
 print(head);
 printf("\nLength of head is:%d",count(head));
 getch();
}
```

当输入 5 个数据时，产生的输出结果如下图所示：



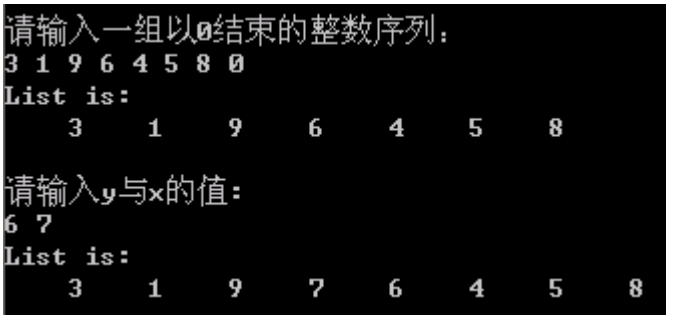

3.4 设计一个算法，在一个单链表中值为 y 的结点前面插入一个值为 x 的结点。即使值为 x 的新结点成为值为 y 的结点的前驱结点。

【答】：

```
#include "linklist.h"
void insert(linklist head,int y,int x)
{/*在值为 y 的结点前插入一个值为 x 的结点*/
 linklist pre,p,s;
 pre=head;
 p=head->next;
```

```
  while (p && p->data!=y)
  {   pre=p;
      p=p->next;
  }
  if (p)/*找到了值为 y 的结点*/
  {   s=(linklist)malloc(sizeof(linknode));
      s->data=x;
      s->next=p;
      pre->next=s;
  }
}
void main()                    /*测试程序*/
{linklist head;
  int y,x;
  head=creatlinklist();        /*创建单链表*/
  print(head);                 /*输出单链表*/
  printf("\n 请输入 y 与 x 的值:\n");
  scanf("%d %d",&y,&x);
  insert(head,y,x);
  print(head);
}
```

程序的一种运行结果如下图所示：



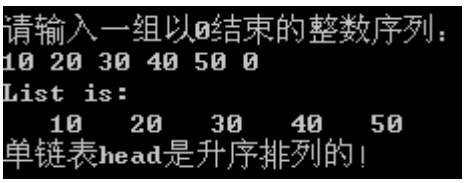

3.5 设计一个算法，判断一个单链表中各个结点值是否有序。

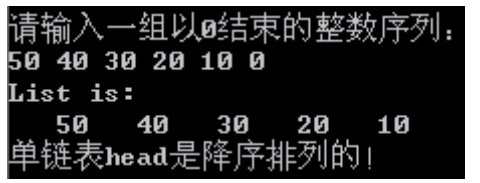
【答】:

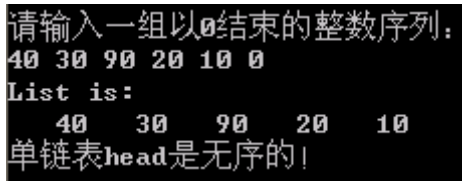
```
#include "linklist.h"
int issorted(linklist head,char c)
/*当参数 c='a'时判断链表是否为升序，当参数 c='d'是判断链表是否为降序*/
{   int flag=1;
    linklist p=head->next;
    switch (c)
    {case 'a':/*判断带头结点的单链表 head 是否为升序*/
```

```
            while (p &&p->next && flag)
            {if (p->data<=p->next->data) p=p->next;
              else flag=0;
            }
            break;
      case 'd':/*判断带头结点的单链表 head 是否为降序*/
            while (p &&p->next && flag)
            {if (p->data>=p->next->data) p=p->next;
              else flag=0;
            }
      break;
      }
    return flag;
}
int main()       /*测试程序*/
{ linklist head;
  head=creatlinklist();
  print(head);
  if (issorted(head,'a')) printf("单链表 head 是升序排列的！\n");
  else
  if (issorted(head,'d')) printf("单链表 head 是降序排列的！\n");
  else printf("单链表 head 是无序的！\n");
}
```

程序运行时的三种输出结果如下图所示：




3.6 设计一个算法，利用单链表原来的结点空间将一个单链表就地转置。

【答】:

```
#include "linklist.h"
void verge(linklist head)
{/*本函数的功能是就地倒置带头结点的单链表*/
```

```
 linklist p,q;
 p=head->next;
 head->next=NULL;
 while (p)            /*每次从原表取一个结点插入到新表的最前面*/
 {q=p;
  p=p->next;
  q->next=head->next;
  head->next=q;
 }
}
int main()                    /*测试函数*/
{linklist head;
 head=creatlinklist();        /*创建单链表*/
 print(head);                 /*输出原单链表*/
 verge(head);                 /*就地倒置单链表*/
 print(head);                 /*输出倒置后的单链表*/
}
```

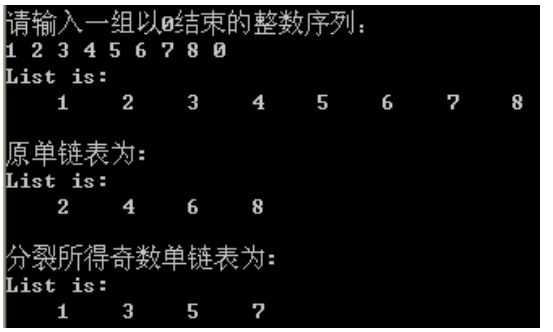
3.7 设计一个算法，将一个结点值自然数的单链表拆分为两个单链表，原表中保留值为偶数的结点，而值为奇数的结点按它们在原表中的相对次序组成一个新的单链表。

【答】:

```
#include "linklist.h"
linklist sprit(linklist head)
{    /*将带头结点的单链表 head 中的奇数值结点删除生成新的单链表并返回*/
     linklist L,pre,p,r;
     L=r=(linklist)malloc(sizeof(linknode));
     r->next=NULL;
     pre=head;
     p=head->next;
     while (p)
     {    if (p->data%2==1)          /*删除奇数值结点*/
          {    pre->next=p->next;
               r->next=p;
               r=p;
               p=pre->next;
          }
          else                       /*保留偶数值结点*/
          {    pre=p;
               p=p->next;
          }
```

```
    }
    r->next=NULL;                    /*置链表结束标记*/
    return L;
}
int main()                  /*测试函数*/
{linklist head,L;
 head=creatlinklist();     /*创建单链表*/
 print(head);              /*输出原单链表*/
 L=sprit(head);            /*分裂单链表 head*/
 printf("\n 原单链表为:\n");
 print(head);              /*输出倒置后的单链表*/
 printf("\n 分裂所得奇数单链表为:\n");
 print(L);
}
```

本程序的一组测试情况如下图所示。




3.8 设计一个算法，对一个有序的单链表，删除所有值大于 x 而不大于 y 的结点。

【答】:

```
#include "linklist.h"
void deletedata(linklist head,datatype x,datatype y)
{    /*删除带头结点单链表中所有结点值大于 x 而不大于 y 的结点*/
     linklist pre=head,p,q;
     p=head->next;       /*初始化*/
     while (p && p->data<=x)     /*找第 1 处大于 x 的结点位置*/
          {    pre=p;
               p=p->next;
          }
     while (p && p->data<=y)     /*找第 1 处小于 y 的位置*/
          p=p->next;
     q=pre->next;                /*删除大于 x 而小于 y 的结点*/
     pre->next=p;
     pre=q->next;
```

```
    while (pre!=p)                  /*释放被删除结点所占用的空间*/
    {free(q);
     q=pre;
     pre=pre->next;
    }
}
void main()                         /*测试函数*/
{   linklist head,L;
    datatype x,y;
    head=creatlinklist();       /*创建单链表*/
    print(head);                /*输出原单链表*/
    printf("\n 请输入要删除的数据区间:\n");
    scanf("%d%d",&x,&y);
    deletedata(head,x,y);
    print(head);                /*输出删除后的单链表*/
}
```

3.9 设计一个算法，在双链表中值为 y 的结点前面插入一个值为 x 的新结点。即使值为 x 的新结点成为值为 y 的结点的前驱结点。

【答】:

首先定义双链表的数据结构，相关文件 dlink.h，内容如下：

```
typedef int datatype;                       /*预定义的数据类型*/
typedef struct dlink_node{                  /*双链表结点定义*/
    datatype data;
    struct dlink_node *llink,*rlink;
    }dnode;
typedef dnode* dlinklist;                       /*双链表结点指针类型定义*/

/*尾插法创建带头结点的双链表*/
dlinklist creatdlinklist(void)
{   dlinklist head,r,s;
    datatype x;
    head=r=(dlinklist) malloc(sizeof(dnode));    /*建立双链表的头结点*/
    head->llink=head->rlink=NULL;
    printf("\n 请输入双链表的内容：（整数序列，以 0 结束）\n");
    scanf("%d",&x);
    while (x)                   /*输入结点值信息，以 0 结束*/
    {   s=(dlinklist ) malloc(sizeof(dnode));
        s->data=x;
        s->rlink=r->rlink;      /*将新结点 s 插入到双链表链尾*/
```

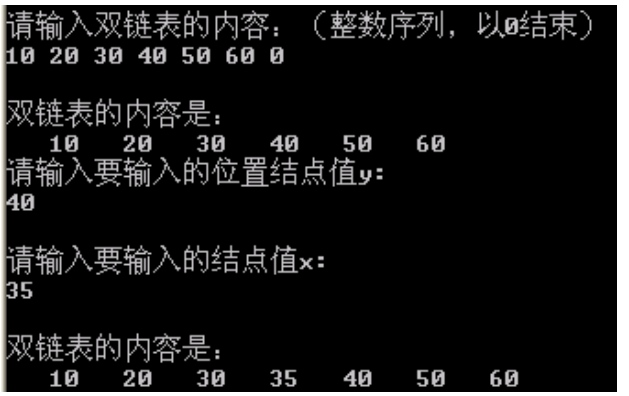
```
            s->llink=r;
            r->rlink=s;
            r=s;
            scanf("%d",&x);
        }
        return head;
}
/*输出双链表的内容*/
void print(dlinklist head)
{   dlinklist p;
    p=head->rlink;
    printf("\n 双链表的内容是：\n");
    while (p)
    {   printf("%5d",p->data);
        p=p->rlink;
    }
}
```

本题的求解程序如下：

```
#include <stdio.h>
#include "dlink.h"
void insertxaty(dlinklist head,datatype y,datatype x)
{ dlinklist s,p;
  /*首先在双链表中找 y 所在的结点，然后在 y 前面插入新结点*/
  p=head->rlink;
  while (p && p->data!=y)
    p=p->rlink;
  if (!p)   printf("\n 双链表中不存在值为 y 的结点,无法插入新结点!\n");
  else   /*插入值为 x 的新结点*/
      { s=(dlinklist)malloc(sizeof(dnode));
        s->data=x;
        s->rlink=p;
        s->llink=p->llink;
        p->llink->rlink=s;
        p->llink=s;
      }
}
void main()          /*测试函数*/
{   dlinklist head;
    datatype x,y;
```

```
    head=creatdlinklist();
    print(head);
    printf("\n 请输入要输入的位置结点值 y:\n");
    scanf("%d",&y);
    printf("\n 请输入要输入的结点值 x:\n");
    scanf("%d",&x);
    insertxaty(head,y,x);/*在值为 y 的结点前插入值为 x 的新结点*/
    print(head);/*输出新的双链表*/
    getch();
}
```

本程序的一组测试情况如下图所示。




3.10 设计一个算法，从右向左打印一个双链表中各个结点的值。

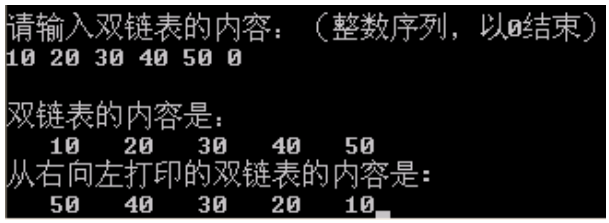
【答】:

本题的双链表定义同题 3.9，实现从右向左打印双链表的各个结点的值可以用递归程序实现如下：

```
#include <stdio.h>
#include "dlink.h"
void vprint(dlinklist head)
{   /*递归方法从右向左打印双链表的值*/
    if (head->rlink)
        {vprint(head->rlink);
         printf("%5d",head->rlink->data);
        }
}
void main()          /*测试函数*/
{    dlinklist head;
     head=creatdlinklist();
     print(head);
     printf("\n 从右向左打印的双链表的内容是:\n");
```

```
    vprint(head);
    getch();
}
```

本程序的一组测试情况如下图所示。




3.11 设计一个算法，将一个双链表改建成一个循环双链表。

【答】:

```
#include <stdio.h>
#include "dlink.h"
/*将一个双链表改成循环双链表*/
void dlinktocdlink(dlinklist head)
{ dlinklist r;
  r=head;
  while (r->rlink)          /*寻找尾结点*/
        r=r->rlink;
  head->llink=r;
  r->rlink=head;
}
void printcdlink(dlinklist head)
{ /*打印双链表*/
  dlinklist p;
  p=head->rlink;
  while (p!=head)
     {printf("%5d",p->data);
      p=p->rlink;
     }
}
int   main()                /*测试函数*/
{     dlinklist head;
      head=creatdlinklist();
      dlinktocdlink(head);
      printf("\n 循环双链表的内容是：\n");
      printcdlink(head);
}
```

# 第 4 章　字符串、数组和特殊矩阵

4.1　稀疏矩阵常用的压缩存储方法有（　三元组顺序存储　）和（ 十字链表　）两种。

4.2　设有一个 10 × 10 的对称矩阵 ***A*** 采用压缩方式进行存储，存储时以按行优先的顺序存储其下三角阵，假设其起始元素 $a_{00}$ 的地址为 1，每个数据元素占 2 个字节，则 $a_{65}$ 的地址为（　53　）。

4.3　若串 S =“software”，其子串的数目为（　36　）。

4.4　常对数组进行的两种基本操作为（　访问数据元素　）和（　修改数组元素　）。

4.5　要计算一个数组所占空间的大小，必须已知（数组各维数）和（每个元素占用的空间）。

4.6　对于半带宽为 b 的带状矩阵，它的特点是：对于矩阵元素 $a_{ij}$ ，若它满足（|i-j|>b），则 $a_{ij} = 0$。

4.7　字符串是一种特殊的线性表，其特殊性体现在（　该线性表的元素类型为字符　）。

4.8　试编写一个函数，实现在顺序存储方式下字符串的 strcompare（$S_1$，$S_2$）运算。

【答】:

```
#include <stdio.h>
#include <string.h>
#define MAXSIZE 100
typedef struct{
      char str[MAXSIZE];
      int length;
}seqstring;
/*  函数 strcompare()的功能是：当 s1>s2 时返回 1，当 s1==s2 时返回 0，当 s1<s2 时返回-1*/
int strcompare(seqstring s1,seqstring s2)
{     int i,m=0,len;
      len=s1.length<s2.length ?s1.length:s2.length;
      for(i=0;i<=len;i++)
            if(s1.str[i]>s2.str[i])
                  {m=1;break;}
            else if(s1.str[i]<s2.str[i])
                  {m=-1;break;}
       return m;
}
int   main()
{     seqstring s1,s2;
      int i,m;
      printf("input    char to s1:\n");
```

```
        gets(s1.str);
        s1.length=strlen(s1.str);
        printf("input    char to s2:\n");
        gets(s2.str);
        s2.length=strlen(s2.str);
        m=strcompare(s1,s2);
        if(m==1) printf("s1>s2\n");
        else if(m==-1) printf("s2>s1\n");
        else if(m==0) printf("s1=s2\n");
    }
```

4.9　试编写一个函数，实现在顺序存储方式下字符串的 replace（$S$，$T_1$，$T_2$）运算。

【参考程序 1】:

```
/*本程序用来在顺序存储下用快速匹配模式实现在字符串 S 中用 T2 替换所有 T1 出现的可能*/
#include<malloc.h>
#include<string.h>
#include<stdio.h>
# define maxsize 100
typedef struct{
        char str[maxsize];
          int length ;
} seqstring;
//求 next[]函数
void getnext(seqstring*p,int next[])
{int i,j;
 next[0]=-1;
 i=0;j=-1;
 while(i<p->length)
 {
  if(j==-1||p->str[i]==p->str[j])
  {++i;++j;next[i]=j;}
  else
    j=next[j];
  }
 }

//快速模式匹配算法
int kmp(seqstring*t,seqstring*p,int next[],int temppos)
{int i,j;
```

```
  i=temppos,j=0;
  while (i<t->length && j<p->length)
  {
    if(j==-1||t->str[i]==p->str[j])
      {i++; j++;}
    else    j=next[j];
  }
    if (j==p->length) return (i-p->length);
    else return(-1);
  }
//替换函数
void takeplace(seqstring*S,seqstring*T1,seqstring*T2)
{
       int next[maxsize],temppos=0,pos,i;
       int lent1,lent2;
       lent1=strlen(T1->str);          //求 T1 串长度
       lent2=strlen(T2->str);          //求 T2 串长度
       getnext(T1,next);//求 T1 串的 next[]向量
       do
       {
          pos=kmp(S,T1,next,temppos);        //快速模式匹配
          temppos=pos+T1->length;
          if (pos!=-1)                       //匹配成功
          {
               if (lent1>lent2)              //T1 串长度大于 T2 串长度
                  {
                  for (i=temppos;i<S->length;i++)   //前移
                        S->str[i-lent1+lent2]=S->str[i];
                  for(i=0;i<T2->length;i++)         //替换
                        S->str[pos+i]=T2->str[i];
                  S->length-=lent1-lent2;           //修改长度

                  }
              else
                  if (lent2>lent1)                  //T2 串长度大于 T1 串长度
                  {
                        for (i=S->length-1;i>=temppos;i--)       //后移留空
                              S->str[i+lent2-lent1]=S->str[i];
                        for(i=0;i<T2->length;i++)                //替换
```

```
                    S->str[pos+i]=T2->str[i];
                S->length+=lent2-lent1;                    //修改长度
            }
        else      //T1 长度与 T2 长度相等
            { for(i=0;i<T2->length;i++)
                S->str[pos+i]=T2->str[i];
            }
        temppos=pos+T2->length;   //修改下一次模式匹配的起点位置
    }
}while(pos!=-1);
S->str[S->length]='\0';
}
int main()
{    seqstring*S,*T1,*T2;
    printf("\n\n 本程序用来实现字符串替换，将 S 中用 T2 替换 T1 所有出现\nThis program is used for characters batch takeing place,The  T1 characters batch will take place all the T2's appearances in characters batch S: \n\n");
    printf("请输入 S 中的字符(plese input characters batch S:)\n");
    S=(seqstring*)malloc(sizeof(seqstring));
    T1=(seqstring*)malloc(sizeof(seqstring));
    T2=(seqstring*)malloc(sizeof(seqstring));
    scanf("%s",S->str);
    S->length=strlen(S->str);
    printf("输入要被替换的串(input T1):\n");
    scanf("%s",T1->str);
    T1->length=strlen(T1->str);
    printf("输入替换串(input T2):\n");
    scanf("%s",T2->str);
    T2->length=strlen(T2->str);
    takeplace(S,T1,T2);
    printf("替换后的字符串为：\n");
    printf("%s\n",S->str);
    free(S);
    free(T1);
    free(T2);
}
```

【参考程序 2】:

```
#include<stdio.h>
```

```
#define MAXSIZE 100
typedef struct{
            char str[MAXSIZE];
            int length;
            }seqstring;
void getnext(seqstring p,int next[])            //求模式的 next 函数
   {int i,j;
    next[0]=-1;
    i=0;
    j=-1;
    while(i<p.length)
      {if(j==-1||p.str[i]==p.str[j])
          {++i;++j;next[i]=j;}
        else j=next[j];
      }
    }
 void replace(seqstring *s,seqstring t1,seqstring t2,int next[])
    {int i,j,k,c,m;
     i=0;j=0;k=0;
     while(i<s->length)
     {     j=0;
           while(i<s->length&&j<t1.length)
           {if(j==-1||s->str[i]==t1.str[j])
               {i++;       j++;}
             else j=next[j];
           }
           if(j==t1.length)                          //匹配成功
            {    c=i-t1.length;
                 if(t1.length==t2.length)            //两串长度相等直接替换
                       for(k=0;k<t2.length;k++)
                             s->str[c+k]=t2.str[k];
                 else if(t1.length<t2.length)
                             {  for(m=s->length-1;m>i-1;m--)
                                     s->str[t2.length-t1.length+m]=s->str[m];      //后移留空
                                for(k=0;k<t2.length;k++)
                                     s->str[c+k]=t2.str[k];
                                s->length=s->length-t1.length+t2.length;
                                s->str[s->length]='\0';
                             }
```

```
                else
                    {   for(m=i-1;m<s->length;m++)          //前移
                            s->str[m-t1.length+t2.length]=s->str[m];
                        for(k=0;k<t2.length;k++)
                            s->str[c+k]=t2.str[k];
                        s->length=s->length-t1.length+t2.length;
                        s->str[s->length]='\0';
                    }
                i=i+t2.length-t1.length;
            }
            i++;
        }
    }
int main()
    {   int next[MAXSIZE];
        seqstring s,t1,t2;
        printf("Input string s:");              //输入主串
        gets(s.str);
        printf("\nInput string t1:");           //输入模式串
        gets(t1.str);
        printf("\nInput string t2:");           //输入拟替换的串
        gets(t2.str);
        s.length=strlen(s.str);
        t1.length=strlen(t1.str);
        t2.length=strlen(t2.str);
        getnext(t1,next);
        replace(&s,t1,t2,next);
        puts(s.str);
    }
```

4.10　试编写一个函数，实现在链式存储方式下字符串的 strcompare（$S_1$，$S_2$）运算。

【参考程序】:

```
/*建立字符串链式存储结构文件 linkstring.h*/
#include <stdio.h>
#include <stdlib.h>
typedef struct node
    {
        char data;
        struct node *next;
     } linkstrnode;
```

```
typedef linkstrnode *linkstring;
/*---------------------------------------*/
/*        尾插法建立单链表        */
/*---------------------------------------*/
void strcreate(linkstring *S)
{ char ch;
   linkstrnode *p,*r;
   *S=NULL; r=NULL;
   while ((ch=getchar())!='\n')
    { p=(linkstrnode *)malloc(sizeof(linkstrnode));
      p->data=ch;          /*产生新结点*/
      if (*S==NULL)    /*新结点插入空表*/
          { *S=p;     r=p;}
      else {r->next=p;
                r=p;}
    }
    if (r!=NULL)    r->next=NULL;      /*处理表尾结点指针域*/
}
/*-----------------------------------------------*/
/*            输出单链表的内容            */
/*-----------------------------------------------*/
void print(linkstring head)
{
 linkstrnode *p;
 p=head;
 while (p)
  {printf("%c-->",p->data);
   p=p->next;
  }
  printf("\n");
 }

#include “linkstring.h”
int strcompare(linkstrnode*S1,linkstrnode*S2)
{
      while(S1&&S2)
      {     if(S1->data>S2->data)
                return 1;
            else if(S1->data<S2->data)
```

```
                return -1;
            S1=S1->next;
            S2=S2->next;
        }
if(S1)          return 1;
else if(S2)     return -1;
return 0;
}
int main()
{   linkstring s1,s2;
    int k;
    printf("\nPlease input s1:");
    strcreate(&s1);     /*建立字符串 s1*/
    print(s1);
    printf("\nPlease input s2:");
    strcreate(&s2);     /*建立字符串 s2*/
    print(s2);
    k=strcompare(s1,s2);
    if (k==1) printf("s1>s2");
    else
        if (k==-1)printf("s1<s2");
    else
        printf("s1==s2");
}
```

4.11　试编写一个函数，实现在链式存储方式下字符串的 replace（$S$，$T_1$，$T_2$）运算。

/*题目：链式存储方式下字符串的 replace(S,T1,T2）运算*/

【参考程序】:

```
#include "linkstring.h"
/*单链表拷贝函数*/
linkstring   copy(linkstring head)
{ linkstring L=NULL,r=NULL,s,p;
  p=head;
  while(p)
     {s=(linkstring)malloc(sizeof(linkstrnode));
      s->data=p->data;
      if (L==NULL)   L=r=s;
      else
          {r->next=s;
```

```
                r=s;
                }
          p=p->next;
       }
     if (r!=NULL)      r->next=NULL;
     return L;
}
/*-------------------------------------------------*/
/*        在字符串 S 中用 T2 替换 T1      */
/*-------------------------------------------------*/
linkstring replace(linkstring S,linkstring T1,linkstring T2)
{linkstring p,q,r,s,pre,temp,pos;
 int flag=1;
 if (S==NULL|| T1==NULL ) return    S;       /*若 S 为空或 T1 为空,则原串不变*/
 pre=NULL;
 pos=S;       /*pos 表示可能的 T1 串在 S 中的起始位置*/
 while (pos && flag)                    /*flag 表示 S 中存在 T1*/
 {    p=pos;
      q=T1;
      while ( p&&q )                          /*从 pos 开始找子串 T1*/
      {    if (p->data==q->data)
                { p=p->next;q=q->next;}
           else
                {pre=pos;
                 pos=pos->next;
                 p=pos;
                 q=T1;
                }
       }
      if (q!=NULL)    flag=0;         /*未找到子串 T1*/
           else
           {       flag=1;                  /*找到了子串 T1,用 T2 代替 T1*/
                   r=pos;
                   while (r!=p)    /*首先删除子串 T1,最后 p 指向删除串 T1 的下一个结点*/
                   {s=r;
                    r=r->next;
                    free(s);
                   }
                if (T2!=NULL)                       /*T2 不为空*/
```

```
                {   temp=r=copy(T2);            /*复制一个 T2 串*/
                    while (r->next!=NULL)       /*找 temp 串的尾结点*/
                        r=r->next;
                    r->next=p;
                    if (pre==NULL)              /*如果 T1 出现在 S 的链头*/
                        S=temp;                 /*新串成为链首*/
                    else                        /*T1 不是链首串*/
                        pre->next=temp;
                    pre=r;
                    pos=p;              /*从 p 开始继续找子串 T1*/
                    }
                else                            /*原 T2 子串为空*/
                {   if (pre==NULL)              /*T1 为链首串*/
                        S=p;
                    else
                        pre->next=p;
                    pos=p;
                }
            }
        }
        return S;
}
int main()
{linkstring S,T1,T2;
   printf("\nPlease input S:");
   strcreate(&S);          /*建立字符串 S*/
   print(S);
   printf("\nPlease input T1:");
   strcreate(&T1);         /*建立字符串 T1*/
   print(T1);
   printf("\nPlease input T2:");
   strcreate(&T2);         /*建立字符串 T2*/
   print(T2);
   S=replace(S,T1,T2);
   printf("\nThe result is:\n");
   print(S);
}
```

4.12　已知如下字符串，求它们的 next 数组值：

（1）“bbdcfbbdac”

【答】：-1010001230

（2）“aaaaaaa”

【答】：-1012345

（3）“babbabab”

【答】：-10011232

4.13　已知正文 $t$ =“ababbaabaa”，模式 $p$ =“aab”，试使用 KMP 快速模式匹配算法寻找 $p$ 在 $t$ 中首次出现的起始位置，给出具体的匹配过程分析。

【答】：模式 p 的 next 向量如下表所示：

| 0 | 1 | 2 |
|---|---|---|
| a | a | b |
| -1 | 0 | 1 |

第一趟匹配：

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| a | b | a | b | b | a | a | b | a | a |
| a | a | | | | | | | | |

i=1，j=1，匹配失败，此时 j 取 next[1]=0，即下一趟匹配从 i=1，j=0 开始；

第二趟匹配：

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| a | b | a | b | b | a | a | b | a | a |
| | a | | | | | | | | |

i=1，j=0，匹配失败，此时 j=next[0]=-1，下一趟匹配从 i=2，j=0 开始；

第三趟匹配：

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| a | b | a | b | b | a | a | b | a | a |
| | | a | a | | | | | | |

i=3，j=1，匹配失败，此时 j=next[1]=0，下一趟匹配从 i=3，j=0 开始；

第四趟匹配：

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| a | b | a | b | b | a | a | b | a | a |
| | | | a | | | | | | |

i=3，j=0，匹配失败，此时 j=next[0]=-1，下一趟匹配从 i=4，j=0 开始；

第五趟匹配：

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| a | b | a | b | b | a | a | b | a | a |
| | | | | a | | | | | |

i=4，j=0，匹配失败，此时 j=next[0]=-1，下一趟匹配从 i=5，j=0 开始；

第五趟匹配：

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

| a | b | a | b | b | a | a | b | a | a |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | a | a | b | | |

i=8，j=3，匹配完成，表明匹配成功的位置是：i-j=5。

4.14　已知三维数组 ***A***[3][2][4]，数组首地址为 100，每个元素占用 1 个存储单元，分别计算数组元素 ***A***[0][1][2]在按行优先和按列优先存储方式下的地址。

【答】:

***A***[0][1][2]按行优先方式在内存的存储地址为：100+0*8+1*4+2=106

***A***[0][1][2]按列优先方式在内存的储储地址为：100+2*6+1*3+0*8=115

4.15　已知两个稀疏矩阵 ***A*** 和 ***B***，其行数和列数均对应相等，编写一个函数，计算 ***A*** 和 ***B*** 之和，假设稀疏矩阵采用三元组表示。

【参考程序 1】:

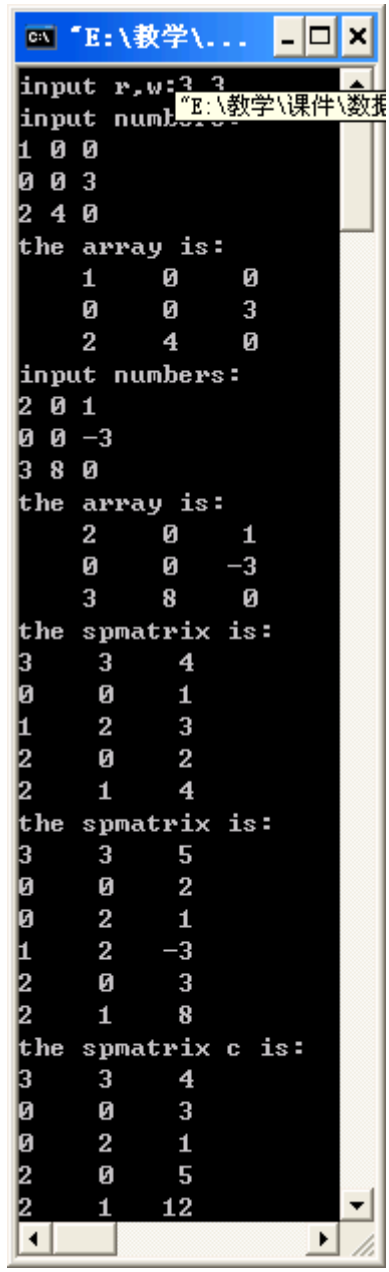
```
#include<stdio.h>
typedef struct {
int    data[100][100];
int    m,n;              /*分别存放稀疏矩阵的行数、列数和非零元素的个数*/
 } matrix;
typedef    int    spmatrix[100][3];   //三元组存储结构
spmatrix c;

void add(spmatrix a,spmatrix b,spmatrix c)          //基于三元组结构的矩阵相加算法
 {int i,j,k,t,r;
  i=j=k=1;
  while(i<=a[0][2]&&j<=b[0][2])
  {if(a[i][0]<b[j][0]||(a[i][0]==b[j][0]&&a[i][1]<b[j][1]))
      {    c[k][0]=a[i][0];
           c[k][1]=a[i][1];
           c[k][2]=a[i][2];
           i++;k++;
       }
       else
   if(a[i][0]>b[j][0]||(a[i][0]==b[j][0]&&a[i][1]>b[j][1]))
           {     c[k][0]=b[j][0];
                 c[k][1]=b[j][1];
                 c[k][2]=b[j][2];
                 j++;k++;
    }
   else
   if(a[i][0]==b[j][0]&&(a[i][1]==b[j][1]) )
           {     c[k][0]=a[i][0];
```

```
                c[k][1]=a[i][1];
                c[k][2]=a[i][2]+b[j][2];
                i++;j++;
                if (c[k][2]!=0 )     k++;       /*如果相加的和不为 0,则生成一个有效项*/
            }
        }
        while(j<=b[0][2])
        {       c[k][0]=b[j][0];
                c[k][1]=b[j][1];
                c[k][2]=b[j][2];
                j++;k++;
        }
        while(i<=a[0][2])
          {
                c[k][0]=a[i][0];
                c[k][1]=a[i][1];
                c[k][2]=a[i][2];
                i++;k++;
          }
         c[0][0]=a[0][0];
         c[0][1]=a[0][1];
         c[0][2]=k-1;
}

//函数 compressmatrix 将普通矩阵存储转换为三元组存储结构
void compressmatrix(matrix *A , spmatrix B)
  {    int i, j, k;
       k=1;
       for ( i=0; i<A->m; i++)
             for (j=0;j<A->n; j++)
             if (A->data[i][j] !=0)
                   { B[k][0]=i;
                     B[k][1]=j;
                     B[k][2]=A->data[i][j];
                     k++;
                   }
       B[0][0]=A->m;
       B[0][1]=A->n;
       B[0][2]=k-1;
```

```
i=0;
printf("the spmatrix is:\n");
while(i<=B[0][2])
   { printf("%d%5d%5d\n",B[i][0],B[i][1],B[i][2]);
     i++;
   }
}
void creat(int r,int w,matrix *s)
  {   int i,j,data;
      printf("input numbers:\n");
      for(i=0;i<r;i++)
          for(j=0;j<w;j++)
            {
             scanf("%d",&data);
             s->data[i][j]=data;
            }
      printf("the array is:\n");
      for(i=0;i<r;i++)
         {  for(j=0;j<w;j++)
            printf("%5d",s->data[i][j]);
            printf("\n");
   }
      s->m=r;
      s->n=w;
  }
int main()
  {matrix p,q;
   spmatrix a,b,c;
   int r,w,i;
   i=0;
   printf("input r,w:");          /*输入行和列*/
   scanf("%d%d",&r,&w);
   creat(r,w,&p);
   creat(r,w,&q);
   compressmatrix(&p,a);
   compressmatrix(&q,b);
   i=0;
   add(a,b,c);
   printf("the spmatrix c is:\n");
```

```
    while(i<=c[0][2])
    { printf("%d%5d%5d\n",c[i][0],c[i][1],c[i][2]);
      i++;
    }
}
```

程序运行时先输入矩阵的行和列，然后按普通矩阵格式输入矩阵值，一组程序测试用例运行结果如下图：




【参考程序 2】:

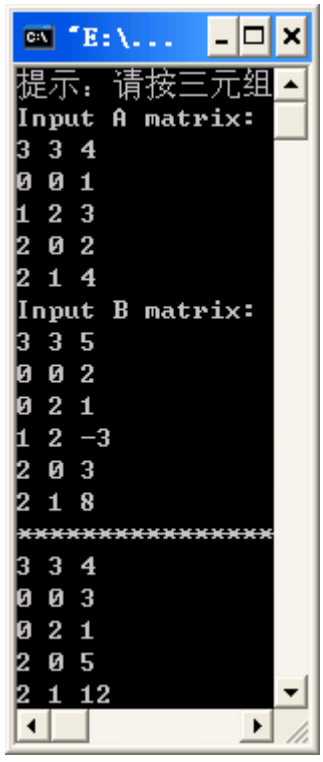
```
#include <stdio.h>
#include <malloc.h>
#define    MAX    100
typedef    struct{
        int    data[MAX][3];
        int    m,n,value;
}matrix;
matrix *Input(matrix *A)
{       int i,j;
        A = (matrix*)malloc(sizeof(matrix));
        scanf("%d%d%d",&A->m,&A->n,&A->value);
        A->data[0][0] = A->m;
           A->data[0][1] = A->n;
        A->data[0][2] = A->value;
        for(i = 1;i <= A->value;i ++)
        {
                for(j = 0;j < 3; j ++)
                        scanf("%d",&A->data[i][j]);
        }
        return A;
}
void Output(matrix *A)
{       int i,j;
        printf("************************************\n");
        for(i = 0;i <= A->value;i ++)
        {
                for(j = 0;j < 3;j ++)
                        printf("%d ",A->data[i][j]);
                printf("\n");
        }
        printf("************************************\n");
}
matrix *addmatrix(matrix *A,matrix *B,matrix *C)
{       int    i = 1,j =1,k = 1;
        C = (matrix*)malloc(sizeof(matrix));
           while(i <= A->value && j <= B->value)
        {
                if(A->data[i][0] == B->data[j][0])
                {
```

```
        if(A->data[i][1] == B->data[j][1])
        {
                C->data[k][0] = A->data[i][0];
                C->data[k][1] = A->data[i][1];
                C->data[k][2] = A->data[i][2] + B->data[j][2];
                if(C->data[k][2] != 0)    k ++;
                i ++;
                j ++;
        }
        else if(A->data[i][1] < B->data[j][1])
        {
                C->data[k][0] = A->data[i][0];
                C->data[k][1] = A->data[i][1];
                C->data[k][2] = A->data[i][2];
                   k ++;
                i ++;
        }
        else
        {
                C->data[k][0] = B->data[j][0];
                C->data[k][1] = B->data[j][1];
                C->data[k][2] = B->data[j][2];
                   k ++;
                j ++;
        }
}
else if(A->data[i][0] < B->data[j][0])
{
                C->data[k][0] = A->data[i][0];
                C->data[k][1] = A->data[i][1];
                C->data[k][2] = A->data[i][2];
                   k ++;
                i ++;
}
else
{
                C->data[k][0] = B->data[j][0];
                C->data[k][1] = B->data[j][1];
                C->data[k][2] = B->data[j][2];
```

```
                k ++;
            j ++;
        }
    }
    while(i <= A->value)
        {
            C->data[k][0] = A->data[i][0];
            C->data[k][1] = A->data[i][1];
            C->data[k][2] = A->data[i][2];
                k ++;
            i ++;
        }
    while(j <= B->value)
        {
            C->data[k][0] = B->data[j][0];
            C->data[k][1] = B->data[j][1];
            C->data[k][2] = B->data[j][2];
                k ++;
            j ++;
        }

    C->data[0][0] = (A->data[0][0]>=B->data[0][0])?A->data[0][0]:B->data[0][0];
    C->data[0][1] = (A->data[0][1]>=B->data[0][1])?A->data[0][1]:B->data[0][1];
        C->data[0][2] = k-1;
    C->value = k-1;
    return C;
}
int main()
{
    matrix * A = NULL,*B = NULL,*C = NULL;
    printf("提示：请按三元组格式输入矩阵内容。\n");
    printf("Input A matrix:\n");
    A = Input(A);
    printf("Input B matrix:\n");
    B = Input(B);
    C = addmatrix(A,B,C);
    Output(C);
    free(A);
    free(B);
```

```
    free(C);
    return 0;
}
```

程序运行时按照三元组的格式输入矩阵信息，程序运行结果如下图所示：




4.16　写出两个稀疏矩阵相乘的算法，计算：

$$\boldsymbol{C}_{pn} = \boldsymbol{A}_{pm} * \boldsymbol{B}_{mn}$$

其中，***A***、***B*** 和 ***C*** 都采用三元组表示法存储。

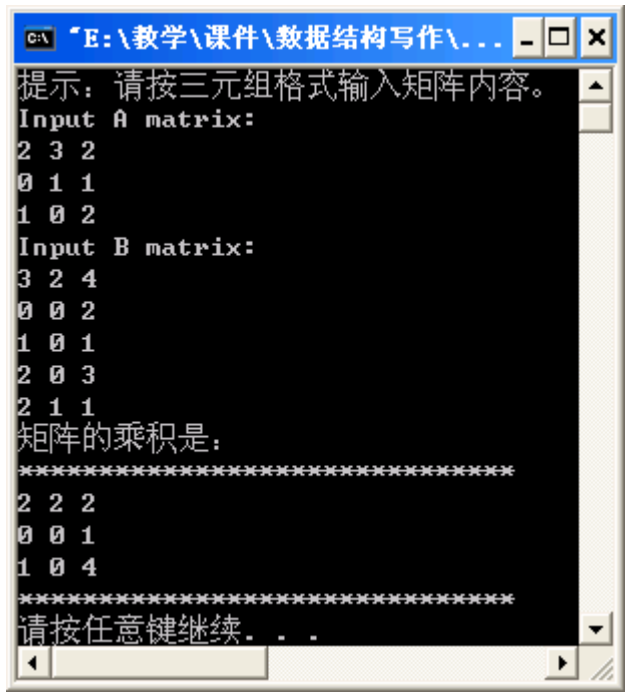
【答】:

```
#include <stdio.h>
#include <malloc.h>
#define   MAX   100
typedef   struct{
        int   data[MAX][3];
        int   m,n,value;
}matrix;
matrix *Input(matrix *A)
{     int i,j;
      A = (matrix*)malloc(sizeof(matrix));
      scanf("%d%d%d",&A->m,&A->n,&A->value);
      A->data[0][0] = A->m;
```

```
    A->data[0][1] = A->n;
    A->data[0][2] = A->value;
    for(i = 1;i <= A->value;i ++)
    {
        for(j = 0;j < 3; j ++)
            scanf("%d",&A->data[i][j]);
    }
    return A;
}
void Output(matrix *A)
{
    int i,j;
    printf("*******************************\n");
    for(i = 0;i <= A->value;i ++)
    {
        for(j = 0;j < 3;j ++)
            printf("%d ",A->data[i][j]);
        printf("\n");
    }
    printf("*******************************\n");
}
/*   基于三元组存储结构的矩阵相乘算法   */
matrix *MulMatrix(matrix *A,matrix *B,matrix *C)
{     int   i ,j ,k ,r=1,p,q,s;
      C = (matrix*)malloc(sizeof(matrix));
      C->m=A->m;
      C->n=B->n;
      C->data[0][0]=C->m;
      C->data[0][1]=C->n;
      for (i=0;i<C->m;i++)
          for (j=0;j<C->n;j++)
          {    s=0;
               for (k=0;k<A->n;k++)
               //找 A[i][k];
               {
               p=1;
               while (p<=A->value)
                  if (A->data[p][0]==i&&A->data[p][1]==k)
                     break;
```

```
                else p++;
                    //找 B[k][j];
                q=1;
                while (q<=B->value)
                    if (B->data[q][0]==k&&B->data[q][1]==j)
                        break;
                    else q++;
                if (p<=A->value && q<=B->value)
                        s=s+A->data[p][2]*B->data[q][2];
                }
                if (s!=0)
                {     C->data[r][0]=i;
                      C->data[r][1]=j;
                      C->data[r][2]=s;
                      r++;
                }
            }
        C->data[0][2]=r-1;
        C->value=r-1;
        return C;
}

int main()
{     matrix * A = NULL,*B = NULL,*C = NULL;
      printf("提示：请按三元组格式输入矩阵内容。\n");
      printf("Input A matrix:\n");
      A = Input(A);
      printf("Input B matrix:\n");
      B = Input(B);
      C = MulMatrix(A,B,C);
      Output(C);
      free(A);
      free(B);
      free(C);
      return 0;
}
```

程序运行时，要求按照三元组的格式输入矩阵信息。


"E:\教学\课件\数据结构写作\...
提示：请按三元组格式输入矩阵内容。
Input A matrix:
2 3 2
0 1 1
1 0 2
Input B matrix:
3 2 4
0 0 2
1 0 1
2 0 3
2 1 1
矩阵的乘积是：
**********
2 2 2
0 0 1
1 0 4
**********
请按任意键继续. . .

# 第 5 章　递归

5.1　试述递归程序设计的特点。

【答】:

（1）具备递归出口。递归出口定义了递归的终止条件，当程序的执行使它得到满足时，递归执行过程便终止。有些问题的递归程序可能存在几个递归出口。

（2）在不满足递归出口的情况下，根据所求解问题的性质，将原问题分解成若干子问题，子问题的求解通过以一定的方式修改参数进行函数自身调用加以实现，然后将子问题的解组合成原问题的解。递归调用时，参数的修改最终必须保证递归出口得以满足。

5.2　试简述简单递归程序向非递归程序转换的方法。

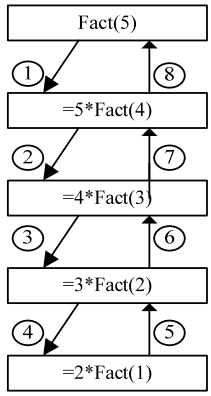
【答】：简单递归程序转换为非递归程序可以采用算法设计中的递推技术。递归方法与递推方法共同采用的分划技术，为使用递推技术解决递归问题奠定了基础。由于简单递归问题求解过程中无需回溯，因而要转换成非递归方式实现完全可以使用递推技术。为了使用自底向上的递推方式来解决递归问题，利用子问题已有的解构造规模较大子问题的解，进而构造原问题的解，必须建立原问题与子问题解之间的递推关系，然后定义若干变量用于记录求解过程中递推关系的每个子问题的解；程序的执行便是根据递推关系，不断修改这些变量的值，使之成为更大子问题的解的过程；当得到原问题解时，递推过程便可结束了。

5.3　试简述复杂递归程序向非递归程序转换的方法，并说明栈在复杂递归程序转换成非递归程序的过程中所起的作用。

【答】：复杂递归问题在求解的过程中无法保证求解动作一直向前，需要设置一些回溯点，当求解无法进行下去或当前处理的工作已经完成时，必须退回到所设置的回溯点，继续问题的求解。因此，在使用非递归方式实现一个复杂递归问题的算法时，经常使用栈来记录和管理所设置的回溯点。

5.4　试给出例题 5.1 中 Fact(5)的执行过程分析。

【答】：Fact(5)的执行过程如下图所示：




5.5　已知多项式 $p_n(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ 的系数按顺序存储在数组 $a$ 中，试：

（1）编写一个递归函数，求 $n$ 阶多项式的值；

（2）编写一个非递归函数，求 *n* 阶多项式的值。

【答】:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
double pnx1(double a[],int n,double x)        //（1）递归函数
{    if (n==0) return a[0];
     else
     return pnx1(a,n-1,x)+a[n]*pow(x,n);
}
double pnx2(double a[],int n,double x)        //（2）非递归迭代函数
{
     double t;
     int i;
     t=a[n];
     for (i=n-1;i>=0;i--)
          t=a[i]+t*x;
     return t;
}
int main()
{    double *a,x,p;
     int n,i;
     printf("请输入 n,x:\n");
     scanf("%d%lf",&n,&x);
     a=(double*)malloc((n+1)*sizeof(double));
     printf("请输入%d 项系数：\n",n+1);
     for (i=0;i<=n;i++)
          scanf("%lf",&a[i]);
     p=pnx1(a,n,x);
     printf("%f\n",p);
     p=pnx2(a,n,x);
     printf("%f\n",p);
     free(a);
}
```

5.6　已知两个一维整型数组 *a* 和 *b*，分别采用递归和非递归方式编写函数，求两个数组的内积（数组的内积等于两个数组对应元素相乘后再相加所得到的结果）。

【答】:

```
#include <iostream>
using namespace std;
```

```
long sum1(int a[],int b[],int n)       //非递归函数
{     long s=0;
      int i;
      for (i=0;i<n;i++)
            s+=a[i]*b[i];
      return s;
}
long sum2(int a[],int b[],int n)       //递归函数
{
      if (n==1)
            return a[0]*b[0];
      else
            return sum2(a,b,n-1)+a[n-1]*b[n-1];
}
int main()
{
      int i,n,*a,*b;
      long s;
      cout<<"输入数组的长度:"<<endl;
      cin>>n;
      a=new int [n];
      b=new int [n];
      cout<<"请输入数组 a 的值"<<endl;
      for (i=0;i<5;i++)
            cin>>a[i];
      cout<<"请输入数组 b 的值"<<endl;
      for (i=0;i<5;i++)
            cin>>b[i];
      s=sum1(a,b,5);
      cout<<"非递归计算的和="<<s<<endl;
      s=sum2(a,b,5);
      cout<<"递归计算的和="<<s<<endl;
      delete []a;
      delete []b;
      return 0;
}
```

5.7　写出求 Ackerman 函数 Ack($m$，$n$)值的递归函数，Ackerman 函数在 $m \geq 0$ 和 $n \geq 0$ 时的定义为：

Ack(0，$n$)=$n$+1;

Ack($m$，0)=Ack($m$−1，1);

Ack($m$，$n$)=Ack($m$−1，Ack($m$，$n$−1))　　$n$>0 且 $m$>0

【答】:

```
// 输入　m,和 n 的值，计算 Ack (m,n)
#include <stdio.h>
int Ack (int m,int n)
{    if (m == 0)
     {        return n + 1;
     }
     else if (n == 0)
     {        return Ack (m - 1,1);
     }
     else
     {        return Ack (m - 1,Ack (m,n - 1));
     }
}
int main ()
{    int m,n;
     printf ("Please input m n\n");
     scanf ("%d%d",&m,&n);
     printf ("Ack (m,n) = %d\n",Ack (m,n));
     return 0;
}
```

5.8　已知多项式 $F_n(x)$的定义如下：

$$F_n(x)=\begin{cases}1 & n=0\\ 2x & n=1\\ 2xF_{n-1}(x)-2(n-1)F_{n-2}(x) & n>1\end{cases}$$

试写出计算 $F_n(x)$值的递归函数。

【答】:

```
// 输入 n，x　计算 F(x)
#include <stdio.h>
// 为了符合递归函数的表达这里将 x 作为参数传递，在实际运用中建议使用全局变量。
// C++中也可以传递引用  int Function (int n,const int &x)
int Function (int n,int x)
{    if (n == 0)
     {    return 1;
     }
     else if (n == 1)
```

```
    {    return 2 * x;
    }
    else
    {    return 2 * x * Function (n - 1,x) - 2 * (n - 1) * Function (n - 2,x);
    }
}
int main ()
{    int n,x;
    printf ("input n,x:\n");
    scanf ("%d%d",&n,&x);
    printf ("F(x) = %d\n",Function (n,x));
    return 0;
}
```

5.9　*n* 阶 Hanoi 塔问题：设有 3 个分别命名为 X，Y 和 Z 的塔座，在塔座 X 上从上到下放有 *n* 个直径各不相同、编号依次为 1，2，3，…，*n* 的圆盘（直径大的圆盘在下，直径小的圆盘在上），现要求将 X 塔座上的 *n* 个圆盘移至塔座 Z 上，并仍然按同样的顺序叠放，且圆盘移动时必须遵循以下规则：

（1）每次只能移动一个圆盘；

（2）圆盘可以插在塔座 X，Y 和 Z 中任何一个塔座上；

（3）任何时候都不能将一个大的圆盘压在一个小的圆盘之上。

试编写一个递归程序实现该问题。

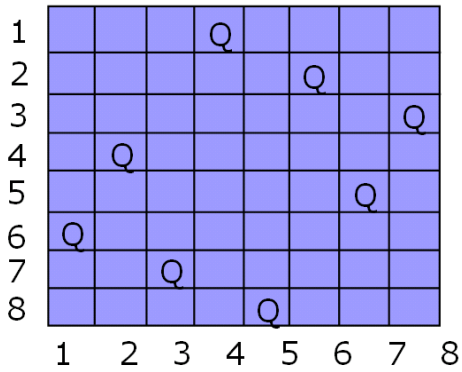
【答】:

```
#include <stdio.h>
void Hanoi(int n,char x,char y,char z)
{
if (n==1)
    printf("%c-->%c\n",x,z);
else
{
    Hanoi(n-1,x,z,y);
    printf("%c-->%c\n",x,z);
    Hanoi(n-1,y,x,z);
}
}
int main()
{
int n;
printf("请输入盘子个数：");
```

```
scanf("%d",&n);
Hanoi(n,'X','Y','Z');
return 0;
}
```

5.10　八皇后问题：在一个 8×8 格的国际象棋棋盘上放上 8 个皇后，使其不能相互攻击，即任何两个皇后不能处于棋盘的同一行、同一列和同一斜线上。试编写一个函数实现八皇后问题。

【答】:




解法一：（程序 place.cpp）

解向量：($x_1$，$x_2$，…，$x_n$)

显约束：$x_i$=1，2，…，n

隐约束：

1）不同列：$x_i \neq x_j$

2）不处于同一正、反对角线：$|i-j| \neq |x_i-x_j|$

```
#include <stdio.h>
#include <math.h>
#define n 8
int x[n+1]={0};      //x[i]的含义表示第 i 行放置在 x[i]列
int sum=0;           //sum 用于记录 n 后问题的所有解
void print();        //输出 n 后问题的解
int Place(int k)     //检测第 k 行的皇后放置是否符后规则
{ for (int j=1;j<k;j++)
      if ((abs(k-j)==abs(x[j]-x[k]))||(x[j]==x[k]))
      return 0;
   return 1;
}
void Backtrack(int t)     //放置 n 后问题的第 t 行
{   if (t>n) {   sum++;
                 print();   //输出一种解方案
                 getchar();
```

```
        }
    else
    for (int i=1;i<=n;i++)
    {   x[t]=i;          //试探在第 t 行放在第 i 列
        if (Place(t))    Backtrack(t+1);
    }
 }
void print()             //输出 n 后问题的解
{   int i;
    printf("%d 皇后问题的第%d 种解为:\n",n,sum);
    for (i=1;i<=n;i++)
        printf("%d:%d\n",i,x[i]);
}
int main()
{
    Backtrack(1);
}
```

解法二：

程序思想：

用 3 个数组分别存放 8 列，15 个左对角线和 15 个右对角线的值,同时用一个 8 元素大小的数组记录下放置的皇后的位置。

初始时将三个数组全部标记为 0。

一，从第一行开始，从左向右开始查找。当找到第一个能够放下棋子的点 x,y（其对应的列，左对角，右对角全部为 0）后，将此棋子所对应的列，左对角，右对角全部设置为 1，然后继续放置下一行——x + 1 行（调用自身的子函数）。

二，当其子函数执行完毕后，将放在 x,y 上的点拿起，然后将这点对应的列，左对角，右对角元素全部置 0，然后继续在此行的 y 后查找能放置的点。

三，如果当前要放置的棋子为第 9 个棋子的时候，打印结果。

程序见 5_10.c （递归法） 5_10_byStack.c (用栈回溯法)

```
/*************************************************/
/*                  八皇后问题递归解法             */
/*************************************************/
#include<stdio.h>
int  left[16],right[16],row[9],queen[9],count=0; /*分别用来存放左对角线和右对角线的使用情况,row 数组记录行数的棋子摆放情况，未使用的话就赋予 0，使用的话就赋予 1*/
/*row  的数组用来存放列的使用情况，当第 n 行，第 m 列被使用的时候在 row[n]=m*/
/*其中 queen 数组用来专门存放当前皇后的摆放情况*/
```

```
int prin()/*打印出皇后的放置情况*/
{int m;
    count++;
    printf(" The %d    th",count);
    for(m=1;m<=8;m++)
       printf("    %d",queen[m]);
       printf("\n");
  return 0;
}
int kaiserin(int line)/*line  表示当前放棋子的列数,第一次放的时候为 1 列*/
{     int i;/*本变量用来记录当前行放子的列数*/
      for(i=1;i<=8;i++)
      if((row[i]||left[i+line-1]||right[8-i+line])==0)/*看当前的列是否可以放*/
        {
            row[i]=1;left[i+line-1]=1;
            right[8-i+line]=1;queen[line]=i;
            if(line==8)/*如果当前找到第 8 行了，说明找成功了，打印*/
            prin();
            else
            kaiserin(line+1);/*找到了，但是还没找完，将这点放上，然后尝试下一列*/
            left[i+line-1]=0;row[i]=0;
            right[8-i+line]=0;queen[line]=0;
            /*不管找到没找到，重新初始化当前放子的点，开始右移放子*/
          }
      return 0;
}
void init()/*将三个数组进行初始化*/
{     int i;
      for(i=1;i<16;i++)
            left[i]=right[i]=0;
      for(i=1;i<9;i++)
            queen[i]=row[i]=0;
}
int main()
{     init();
      kaiserin(1);
      return 0;
}
```

```
/*****************************************/
/*                八皇后问题回溯法                */
/*****************************************/
#include<stdio.h>
#define max 10
typedef struct{/*用来存放回溯点的信息，row 用来存放回溯点所放的行数，line 用来存放所放的列数*/
        int row;
        int line;
}node;
int  left[16],right[16],row[9],queen[9],count=0;/*分别用来存放左对角线和右对角线的使用情况,row 数组记录行数的棋子摆放情况，未使用的话就赋予 0，使用的话就赋予 1*/
/*row  的数组用来存放列的使用情况，当第 n 行，第 m 列被使用的时候在 row[n]=m*/
/*其中 queen 数组用来专门存放当前皇后的摆放情况*/
void print()                /*打印出皇后的放置情况*/
{   int m;
    count++;
    printf("The %d    th",count);
    for(m=1;m<=8;m++)
        printf("    %d",queen[m]);
    printf("\n");
}
void init()/*将三个数组进行初始化*/
{       int i;
        for(i=1;i<16;i++)
            queen[i]=left[i]=right[i]=0;
        for(i=1;i<9;i++)
            row[i]=0;
}
int kaiserin()
{     node back[max];
      int opline=1,oprow=1,i,find,top=0;
      while(1)/*循环条件用真，用 return 来结束循环和整个函数*/
      {   find=0;
          for(i=oprow;i<=8;i++)
              if((row[i]||left[i+opline-1]||right[8-i+opline])==0)/*判断当前的点是否可放*/
                  {           find=1;/*找到的话结束循环*/
                              break;
```

```
            }
            if(find)/*找到了这点*/
            {    oprow=i;
                 row[oprow]=1; left[oprow+opline-1]=1;
                 right[8-oprow+opline]=1;queen[opline]=oprow;/*把这个点放上*/
                 top++;
                 back[top].row=oprow;back[top].line=opline;
                  /*记录回溯点*/
                 opline++;oprow=1;/*开始找下一行*/
             }
              else /*需要进行回溯*/
              {    if(opline==9)
                       print();
                   do{
                       if(top==0)
                         return 0;/*如果栈取空了，说明查找结束*/
                   oprow=back[top].row;/*如果栈中还有元素可取*/
                   opline=back[top].line;/*取出栈顶元素*/
                   top--;
                   row[oprow]=0; left[oprow+opline-1]=0;/*同时将这个棋子拿起*/
                   right[8-oprow+opline]=0;queen[opline]=0;
                   }while(oprow==8);/*当取到的 oprow 的右边可以放子时，开始下一步操作
*/
                   oprow++;
                  }
        }
}
main()
{       init();
        kaiserin();
}
```

# 第 6 章　树

6.1　树最适合用来表示具有（　有序　）性和（　层次　）性的数据。

6.2　在选择存储结构时，既要考虑数据值本身的存储，还需要考虑（　数据间关系　）的存储。

6.3　对于一棵具有 *n* 个结点的树，该树中所有结点的度数之和为（　n-1　）。

6.4　已知一棵树如图 6.11 所示，试回答以下问题：

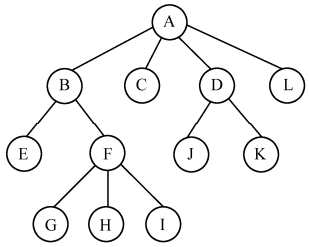



**图** 6.11　**一棵树**

（1）树中哪个结点为根结点？哪些结点为叶子结点？

【答】：A 是根结点；E，G，H，I，C，J，K，L 为叶结点。

（2）结点 B 的双亲为哪个结点？其子女为哪些结点？

【答】：B 的双亲结点是 A，其子女结点为 E 和 F。

（3）哪些结点为结点 I 的祖先？哪些结点为结点 B 的子孙？

【答】：F，B，A 是结点 I 的祖先结点；E，F，G，H，I 是 B 的子孙结点。

（4）哪些结点为结点 D 的兄弟？哪些结点为结点 K 的兄弟？

【答】：B，C，L 是结点 D 的兄弟结点，J 是结点 K 的兄弟结点。

（5）结点 J 的层次为多少？树的高度为多少？

【答】：结点 J 的层次为 3，树的高度为 4。

（6）结点 A、C 的度分别为多少？树的度为多少？

【答】：结点 A 的度为 4，结点 C 的度是 0，树的度是 4。

（7）以结点 B 为根的子树的高度为多少？

【答】：以结点 B 为根的子树的高度是 3。

（8）试给出该树的括号表示及层号表示形式。

【答】：该树的括号表示为：A（B（E，F（G，H，I）），C，D（J，K），L），该树的层号表示为：10A，20B，30E，30F，40G，40H，40I，20C，20D，25J，25K，20L

6.5　试写出图 6.11 所示树的前序遍历、后序遍历和层次遍历的结果。

【答】：　前序遍历：ABEFGHICDJKL

后序遍历：EGHIFBCJKDLA

层次遍历：ABCDLEFJKGHI

6.6　试给出图 6.11 所示树的双亲表示法和数组方式孩子表示法的表示。

【答】:

双亲表示法：

| | data | parent |
|---|---|---|
| 0 | A | -1 |
| 1 | B | 0 |
| 2 | C | 0 |
| 3 | D | 0 |
| 4 | E | 1 |
| 5 | F | 1 |
| 6 | G | 5 |
| 7 | H | 5 |
| 8 | I | 5 |
| 9 | J | 3 |
| 10 | K | 3 |
| 11 | L | 0 |

数组方式的孩子表示法：

| | data | Child[0] | Child[1] | Child[2] | Child[3] |
|---|---|---|---|---|---|
| 0 | A | 1 | 2 | 3 | 11 |
| 1 | B | 4 | 5 | -1 | -1 |
| 2 | C | -1 | -1 | -1 | -1 |
| 3 | D | 9 | 10 | -1 | -1 |
| 4 | E | -1 | -1 | -1 | -1 |
| 5 | F | 6 | 7 | 8 | -1 |
| 6 | G | -1 | -1 | -1 | -1 |
| 7 | H | -1 | -1 | -1 | -1 |
| 8 | I | -1 | -1 | -1 | -1 |
| 9 | J | -1 | -1 | -1 | -1 |
| 10 | K | -1 | -1 | -1 | -1 |
| 11 | L | -1 | -1 | -1 | -1 |

6.7　已知一棵度为 $m$ 的树中有 $n_1$ 个度为 1 的结点，$n_2$ 个度为 2 的结点，……，$n_m$ 个度为 $m$ 的结点，问该树中有多少个叶子结点？

【答】:

树中结点总数 $n=n_0+n_1+n_2+\ldots+n_m$

树中结点的度数之和为 n-1，且有：$n-1=n_1+2n_2+3n_3+\ldots+mn_m$

所以叶子结点个数为：$n_0=n_2+2n_3+\ldots+(m-1)n_m$

6.8　假设树采用指针方式的孩子表示法表示，试编写一个非递归函数，实现树的前序遍历算法。

【答】：本程序在运行时请按树的括号表示法输入拟遍历的树，例如，对习题 6.4 图 6.11 所示的树为例，应输入 A（B（E，F（G，H，I）），C，D（J，K），L）

定义树的链式存储结构及建立树的存储结构 CreateTree 函数（tree.h）内容如下：

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>
// 树的最大度定义为 5
#define MAXN 5
#define MAXLEN 100
typedef char dataType;
typedef struct node
{    dataType key;
     struct node *child[MAXN];
}TreeNode;
/*----------------------------------------------------------------*
     树的括号法转化到树指针方式的孩子表示法
*----------------------------------------------------------------*/
TreeNode *CreateTree (TreeNode *root,char string[MAXLEN])
{    int length;
     int i,j,top;
     TreeNode *stack[MAXLEN],*temp = NULL,*n;
     int childSeq[MAXLEN];      // 其第几个孩子
     top = -1;
     length = strlen (string);
     for (i = 0;i < length;i++)
     {
          if (string[i] == ',')
          {
               continue;
          }
          else if (string[i] == '(')
          {
               stack[++top] = temp;
               childSeq[top] = 0;
          }
          else if (string[i] == ')')
          {
               top--;
          }
```

```
            else if (top != -1)
            {
                n = (TreeNode*)malloc (sizeof (TreeNode));
                n->key = string[i];

                for (j = 0;j < MAXN;j++)
                {
                    n->child[j] = NULL;
                }
                  temp = n;
                stack[top]->child[childSeq[top]++] = temp;
            }
            else
            {
                root = (TreeNode *)malloc (sizeof (TreeNode));
                root->key = string[i];
                for (j = 0;j < MAXN;j++)
                {
                    root->child[j] = NULL;
                }
                temp = root;
            }
        }
        return root;
}

void PreOrder (TreeNode *root)
{   int i;
    if (root != NULL)
    {
        printf ("%5c",root->key);
        for (i = 0;i < MAXN;i++)
        {   PreOrder (root->child[i]);
        }
    }
}
```

树的前序非递归遍历算法及其测试程序（6_8.c）如下所示：

```
#include "tree.h"
/*          前序遍历（非递归）    */
```

```
void    PreOrder1(TreeNode *root)
{       TreeNode *stack[MAXLEN];
        int top=-1,i;
        while (root || top!=-1)
        {
        if (root)
        {       printf("%c",root->key);
                for (i=MAXN-1;i>0;i--)
                        if (root->child[i])
                          stack[++top]=root->child[i];
                root=root->child[0];
        }
        else
        if (top>-1)
                {root=stack[top--];
                }
      }
}
int main ()
{       char string[MAXLEN];
        TreeNode *root = NULL;
        printf ("请用树的括号表示法输入一棵树:\n");
        scanf ("%s",string);
        root = CreateTree (root,string);
        PreOrder1(root);
        return 0;
}
```

6.9　假设树采用指针方式的孩子表示法表示，试编写一个非递归函数，实现树的后序遍历算法。

**【答】:**

```
#include "tree.h"
int PostOrderByStack (TreeNode *root)
{
        TreeNode *temp;
        TreeNode *stack[MAXLEN];
        // childSeq 表示当前打印到了此树第几个孩子，
        int top,childSeq[MAXLEN];
        int i;
        top = -1;          //初始化空栈
```

```
temp = root;
while (1)
{
    while (temp != NULL)
    {
        for (i = 0;i < MAXN;i++)
        {   if (temp->child[i] != NULL)
            {   childSeq[++top] = i + 1;
                stack[top] = temp;
                temp = temp->child[i];
                break;
            }
        }
        // 如果此节点是叶子节点，则输出该结点
        if (i == MAXN)
        {   printf ("%5c",temp->key);
            temp = NULL;
            break;
        }
    }
    while (top != -1 )
    {
        for (i = childSeq[top];i < MAXN;i++)
        {
            if (stack[top]->child[i] != NULL)
            {
                temp = stack[top]->child[i];
                childSeq[top] = i + 1;
                break;
            }
        }
        if (i == MAXN)
        {
            printf ("%5c",stack[top]->key);
            top--;
        }
        if (temp != NULL)
        {
            break;
```

```
                }
            }
            if (top == -1)
            {
                return 1;
            }
        }
    }
    int main ()
    {   char string[MAXLEN];
        TreeNode *root = NULL;
        printf ("请用树的括号表示法输入一棵树:\n");
        scanf ("%s",string);
        root = CreateTree (root,string);
        PostOrderByStack (root);
        return 0;
    }
```

6.10　假设树采用指针方式的孩子表示法表示，试编写一个函数，判断两棵给定的树是否等价（两棵树等价当且仅当其根结点的值相等且其对应的子树均相互等价）。

```
    #include "tree.h"
    #define TRUE    1
    #define FALSE 0
    /*-------------------------------------------------*
            递归比较两棵树是否相等
    *-------------------------------------------------*/
    int PreCmp (TreeNode *tree1,TreeNode *tree2)
    {   int i;
        if (tree1 == NULL && tree2 == NULL)
        {
            return TRUE;
        }
        else if (tree1 == NULL && tree2 != NULL   ||   tree1 != NULL && tree2 == NULL)
        {
            return FALSE;
        }
        else
        {
            if (tree1->key != tree2->key)
            {
```

```
            return FALSE;
        }
        for (i = 0;i < MAXN;i++)
        {
            if (PreCmp (tree1->child[i],tree2->child[i]) == FALSE)
            {
                return FALSE;
            }
        }
        return TRUE;
    }
}

int main ()
{
    char string[MAXLEN];
    TreeNode *tree1 = NULL,*tree2 = NULL;

    printf ("请用树的括号表示法输入第一棵树:\n");
    scanf ("%s",string);
    tree1 = CreateTree (tree1,string);
    printf ("请用树的括号表示法输入第二棵树:\n");
    scanf ("%s",string);
    tree2 = CreateTree (tree2,string);
    if (PreCmp (tree1,tree2) == TRUE)
    {
        printf ("两树相等\n");
    }
    else
    {
        printf ("两树不相等\n");
    }
    return 0;
}
```

# 第 7 章　二叉树

7.1　选择题。

（1）前序遍历和中序遍历结果相同的二叉树为（　F　）；前序遍历和后序遍历结果相同的二叉树为（　B　）。

A．一般二叉树　　B．只有根结点的二叉树

C．根结点无左孩子的二叉树　　D．根结点无右孩子的二叉树

E．所有结点只有左子树的二叉树　　F．所有结点只有右子树的二叉树。

（2）以下有关二叉树的说法正确的是（　B　）。

A．二叉树的度为 2　　B．一棵二叉树的度可以小于 2

C．二叉树中至少有一个结点的度为 2　　D．二叉树中任一个结点的度均为 2

（3）一棵完全二叉树上有 1001 个结点，其中叶子结点的个数为（　D　）。

A．250　　B．500　　C．254　　D．501

（4）一棵完全二叉树有 999 个结点，它的深度为（　B　）。

A．9　　B．10　　C．11　　D．12

（5）一棵具有 5 层的满二叉树所包含的结点个数为（　B　）。

A．15　　B．31　　C．63　　D．32

7.2　用一维数组存放完全二叉树：ABCDEFGHI，则后序遍历该二叉树的结点序列为（　HIDEBFGCA　）。

7.3　有 $n$ 个结点的二叉树，已知叶结点个数为 $n_0$，则该树中度为 1 的结点的个数为（　$n-2n_0+1$　）；若此树是深度为 $k$ 的完全二叉树，则 $n$ 的最小值为（　$2^{k-1}$　）。

7.4　设 F 是由 T1、T2 和 T3 三棵树组成的森林，与 F 对应的二叉树为 B。已知 T1、T2 和 T3 的结点数分别是 n1、n2 和 n3，则二叉树 B 的左子树中有（　n1-1　）个结点，二叉树 B 的右子树中有（　n2+n3　）结点。

7.5　高度为 $k$ 的二叉树的最大结点数为（　$2^k-1$　），最小结点数为（　k　）。

7.6　对于一棵具有 $n$ 个结点的二叉树，该二叉树中所有结点的度数之和为（　n-1　）。

7.7　已知一棵二叉树如图 7.12 所示，试求：

（1）该二叉树前序、中序和后序遍历的结果；

【答】：前序：abdgecfh；中序：dgbcafhc；后序：gdebhfca

（2）该二叉树是否是满二叉树？是否是完全二叉树？

【答】：该二叉树不是满二叉树，也不是完全二叉树。

（3）将它转换成对应的树或森林；

【答】：

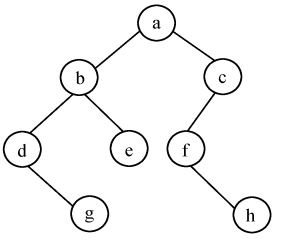
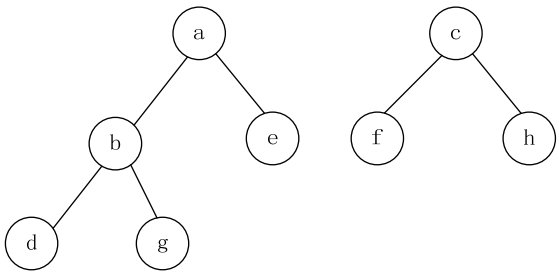



图 7.12　一棵二叉树

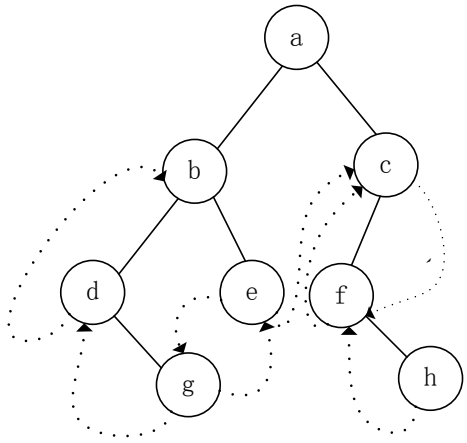



（4）这棵二叉树的深度为多少？

【答】：该二叉树的深度为 4。

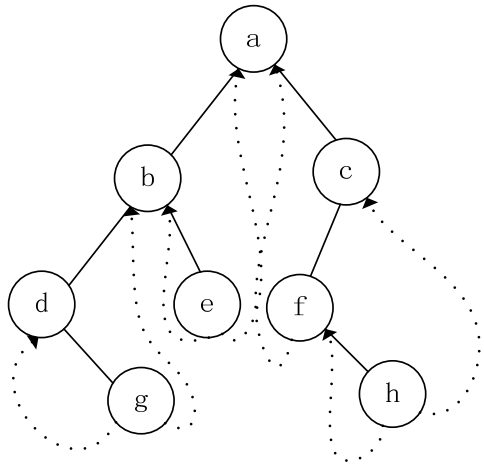
（5）试对该二叉树进行前序线索化；

【答】：




（6）试对该二叉树进行中序线索化。

【答】：




7.8　试述树和二叉树的主要区别。

【答】:

（1）树的结点个数至少为 1，而二叉树的结点个数可以为 0。

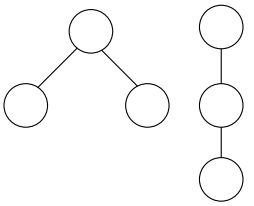
（2）树中结点的最大度数没有限制，而二叉树结点的最大度数为 2。

（3）树分为有序树与无序树，而二叉树一定是有序树，它的结点有左，右之分。

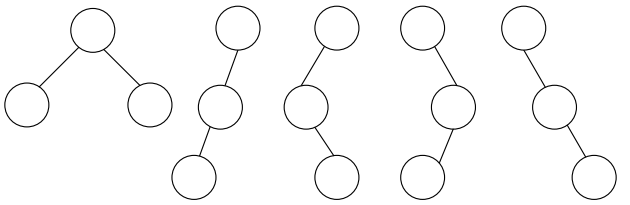
7.9　试分别画出具有 3 个结点的树和具有 3 个结点的二叉树的所有不同形态。

【答】:

三个结点的树有两种形态：

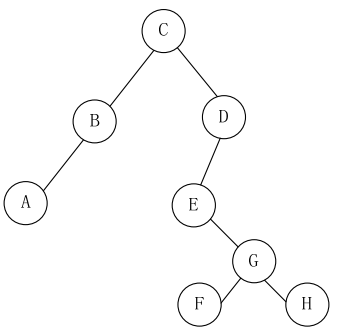


三个结点的二叉树具有五种形态：



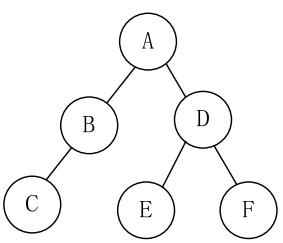
7.10　已知一棵二叉树的中序遍历的结果为 ABCEFGHD，后序遍历的结果为 ABFHGEDC，试画出此二叉树。

【答】:



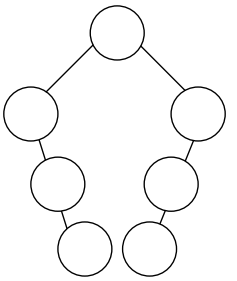

7.11　已知一棵二叉树的前序遍历的结果为 ABCDEF，中序遍历的结果为 CBAEDF，试画出此二叉树。

【答】:



7.12　若一棵二叉树的左、右子树均有 3 个结点，其左子树的前序序列与中序序列相同，右子树的中序序列与后序序列相同，试画出该二叉树。

【答】:



7.13　分别采用递归和非递归方式编写两个函数，求一棵给定二叉树中叶子结点的个数。

【答】：为方便测试二叉树相关程序，定义二叉树头文件 bintree.h 如下：

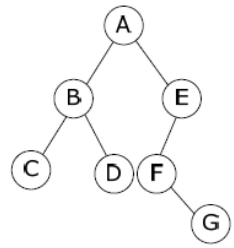
```
#include <stdio.h>
#define N 100
extern char *a;              /*存放扩充二叉树的前序序列*/
typedef struct node          /*二叉树结构定义*/
{
    char data;
    struct node *lchild,*rchild;
}binnode;
typedef binnode *bintree;

/*函数 creat(根据扩充二叉树的前序序列(字符串 a)建立二叉树 t 的存储结构*/
void creat(bintree * t)
{
    char ch=*a++;
    if   (ch==' ')   *t=NULL;
    else
    { *t=(bintree)malloc(sizeof(binnode));
      (*t)->data=ch;
      creat(&(*t)->lchild);
      creat(&(*t)->rchild);
    }
}

void preorder(bintree t)   /*前序递归遍历二叉树*/
{
```

```
    if (t)
    {   printf("%c",t->data);
        preorder(t->lchild);
        preorder(t->rchild);
    }
}

/*顺序栈定义*/
typedef struct
{   bintree data[N];
    int top;
}seqstack;

void init(seqstack *s)    /*初始化空栈*/
{
    s->top=-1;
}
int empty(seqstack *s)    /*判断栈是否为空*/
{
    if (s->top>-1) return 0;
    else return 1;
}
int full(seqstack *s)    /*判断栈是否为满*/
{
    if (s->top==N-1) return 1;
    else return 0;
}
void push(seqstack *s ,bintree x)    /*进栈*/
{
    if (!full(s))
        s->data[++s->top]=x;
}
bintree pop(seqstack *s)            /*出栈*/
{
    if (!empty(s))
        return s->data[s->top--];
}
```

基于上述结构，递归方法与非递归方法求二叉树中叶子结点的个数的算法分别如 leaf1()

与 leaf2()所示：

```
#include "bintree.h"
char *a="ABC    D    EF G      ";          /*扩充二叉树序树 t 的前序序列*/
int leaf1(bintree t)                       /*递归方法求二叉树叶子结点的个数*/
{
      if (t==NULL) return 0;
      else
       if (!t->lchild && !t->rchild)
            return 1;
            else
              return leaf1(t->lchild)+leaf1(t->rchild);
}
int leaf2(bintree t)                       /*非递归方法求二叉树叶子结点的个数*/
{
      seqstack s;                          /*顺序栈*/
      int count=0;                         /*叶子结点计数变量*/
      init(&s);                            /*初始化空栈*/
      while (t || !empty(&s))
      {
            if (t)
            {if (!t->lchild && !t->rchild) count++;
             push(&s,t);
             t=t->lchild;
            }
            else
            {t=pop(&s);
             t=t->rchild;
            }
      }
      return count;
}
int main()
{     bintree t;
      creat(&t);     /*建立二叉树 t 的存储结构*/
      printf("\n 该二叉树一共有%d 个叶子结点！ \n",leaf1(t));
      printf("\n 该二叉树一共有%d 个叶子结点！ \n",leaf2(t));
}
```




7.14　试编写一个函数，将一棵给定二叉树中所有结点的左、右子女互换。

【答】:

```
#include "bintree.h"
char *a="ABC  D  EF G   ";  /*扩充二叉树序树 t 的前序序列*/
void change(bintree t)
{   bintree p;
    if (t)
    {
        p=t->lchild;
        t->lchild=t->rchild;
        t->rchild=p;
        change(t->lchild);
        change(t->rchild);
    }
}
int main()
{   bintree t;
    creat(&t);   /*建立二叉树 t 的存储结构*/
    preorder(t);
    change(t);
    printf("\n");
    preorder(t);
}
```

7.15　试编写一个函数，返回一棵给定二叉树在中序遍历下的最后一个结点。

【答】:

```
#include "bintree.h"
char *a="ABC  D  EF G   ";  /*扩充二叉树序树 t 的前序序列*/
bintree inlast(bintree t)
{   bintree p=t;
    while (p && p->rchild)
        p=p->rchild;
    return p;
}
int main()
{   bintree t;
    creat(&t);   /*建立二叉树 t 的存储结构*/
    printf("二叉树中序遍历最后一个结点是%c\n",inlast(t)->data);
}
```

7.16　试编写一个函数，返回一棵给定二叉树在前序遍历下的最后一个结点。

【答】:

```
#include "bintree.h"
char *a="ABC  D  EF G   ";  /*扩充二叉树序树 t 的前序序列*/
/*求二叉树前序遍历的最后一个结点*/
bintree prelast(bintree t)
{   bintree p;
    if (t)
    {
        p=t;
        while (p && p->lchild || p->rchild)
        if (p->rchild)
            p=p->rchild;
        else
            p=p->lchild;
    }
    return p;
}
int main()
{   bintree t;
    creat(&t);   /*建立二叉树 t 的存储结构*/
    printf("二叉树前序遍历的最后一个结点是%c\n",prelast(t)->data);
}
```

7.17　试编写一个函数，返回一棵给定二叉树在后序遍历下的第一个结点。

【答】:

```
#include "bintree.h"
char *a="ABC  D  EF G   ";  /*扩充二叉树序树 t 的前序序列*/
/*求二叉树后序遍历的第一个结点*/
bintree succfirst(bintree t)
{   bintree p;
    if (t)
    {   p=t;
        while (p && p->lchild || p->rchild)
        if (p->lchild)
            p=p->lchild;
        else
            p=p->rchild;
    }
    return p;
}
```

```
int main()
{    bintree t;
     creat(&t);    /*建立二叉树 t 的存储结构*/
     printf("二叉树后序遍历的第一个结点是%c\n",succfirst(t)->data);
}
```

7.18　假设二叉树采用链式方式存储，root 为其根结点，p 指向二叉树中的任意一个结点，编写一个求从根结点到 p 所指结点之间路径长度的函数。

【答】：在后序遍历非递归算法中，当访问的结点为 p 时，其祖先点正好全部在栈中，此时栈中结点的个数就是根结点到 p 所指结点之间路径长度。

```
#include<stdio.h>
#include<stdlib.h>
typedef char datatype;
typedef struct node                    /*二叉树结点定义*/
{    datatype data;
     struct node *lchild,*rchild;
}bintnode;
typedef bintnode *bintree;

typedef struct stack
{
    bintree data[100];
    int tag[100];
    int top;
}seqstack;
void creat(bintree *t) ;
/*
      函数 Depth，功能：求根结点到 x 的路径长度
*/
int Depth(bintree t,datatype x)
{
    seqstack   s;
    int i=0,j;
    s.top=0;
    while(t || s.top!=0)
    {
        while(t)
            {
            s.data[s.top]=t;
            s.tag[s.top]=0;
```

```
            s.top++;
            t=t->lchild;
            }
        while(s.top>0 && s.tag[s.top-1])
        {
            s.top--;
            t=s.data[s.top];
            if(t->data==x) return s.top;          //此时栈中的结点都是 x 的祖先结点
        }
        if(s.top>0)
        {
            t=s.data[s.top-1];
            s.tag[s.top-1]=1;
            t=t->rchild;
        }
        else   t=NULL;
    }
}

/*函数 creat(根据扩充二叉树的前序序列建立二叉树 t 的存储结构*/
void creat(bintree *t)
{
  char ch;
  scanf("%c",&ch);
  if (ch==' ')
          *t=NULL;
  else {
        *t=(bintnode *)malloc(sizeof(bintnode));
             (*t)->data=ch;
        creat(&(*t)->lchild);
        creat(&(*t)->rchild);
            }
  }
int main()
{   bintree root,p=NULL;
    datatype x;
    int k=0;
    printf("请输入扩充二叉树的前序序列：\n");
    creat(&root);
```

```
    printf("请输入树中的 1 个结点值：\n");
    scanf("%1s",&x);
    k=Depth(root,x);
    printf("根结点到%c 的长度是%d\n",x,k);
}
```

7.19　假设二叉树采用链式方式存储，root 为其根结点，p 和 q 分别指向二叉树中任意两个结点，编写一个函数，返回 p 和 q 最近的共同祖先。

【答】：方法同上题，利用后序遍历非递归算法分别求出 p 和 q 的公共祖先，然后再查找它们的最近公共祖先结点。

```
#include<stdio.h>
#include<stdlib.h>
typedef char datatype;
typedef struct node              /*二叉树结点定义*/
  {    datatype data;
       struct node *lchild,*rchild;
   }bintnode;
typedef bintnode *bintree;

typedef struct stack             //顺序栈结构定义
{   bintree data[100];
    int tag[100];
    int top;
}seqstack;
void creat(bintree *t) ;         //创建二叉树结构函数声明

/*函数 SeekAncestor 的功能是返回二叉树 t 中结点 x 与结点 y 的最近公共祖先结点*/
void SeekAncestor(bintree t,datatype x,datatype y,bintree *antor)
{   seqstack   s;
    bintree data[100]={0};
    int i=0,j;
    s.top=-1;
    while(t || s.top>-1)
    {    while(t)
           {
                    s.data[++s.top]=t;
                    s.tag[s.top]=0;
                    t=t->lchild;
                    }
```

```
                while(s.top>-1 && s.tag[s.top])
                  {
                    t=s.data[s.top];
                    if(t->data==x)
                    while(i<=s.top)            //记录 x 结点的所有祖先结点
                    {
                      data[i]=s.data[i];
                      i++;
                    }
                   else if(t->data==y)
                     {
                       while(s.top>-1)
                       {
                        j=i-1;
                        while(j>=0&&t!=data[j])//查找 y 与 x 的最近公共祖先结点
                        j--;
                        if(j>=0)
                        {
                        *antor=data[j];     //返回公共祖先结点地址
                        return;
                        }
                        t=s.data[--s.top];
                       }
                     }
                    --s.top;
                  }
                  if(s.top>-1)
                  {
                    t=s.data[s.top];
                    s.tag[s.top]=1;
                    t=t->rchild;
                  }
                  else    t=NULL;
    }
}
/*函数 creat(根据扩充二叉树的前序序列建立二叉树 t 的存储结构*/
void creat(bintree *t)
{
  char ch;
```

```
    scanf("%c",&ch);
    if (ch==' ')
          *t=NULL;
    else {
            *t=(bintnode *)malloc(sizeof(bintnode));
            (*t)->data=ch;
            creat(&(*t)->lchild);
            creat(&(*t)->rchild);
          }
    }
  int main()
  {
     bintree root,p=NULL;
     datatype x='B',y='C';
     printf("请输入扩充二叉树的前序序列：\n");
     creat(&root);
     printf("请输入树中的两个结点值：\n");
     scanf("%1s%c",&x,&y);
     SeekAncestor(root,x,y,&p);
     if (p)printf("%c 和%c 的最近公共祖先是：%c\n",x,y,p->data);
}
```

# 第 8 章　图

8.1　选择题

（1）如果某图的邻接矩阵是对角线元素均为零的上三角矩阵，则此图是（　D　）。

A．有向完全图　B．连通图　C．强连通图　D．有向无环图

（2）若邻接表中有奇数个表结点，则一定（　D　）。

A．图中有奇数个顶点　B．图中有偶数个顶点

C．图为无向图　D．图为有向图

（3）下列关于无向连通图特性的叙述中，正确的是（　A　）。

Ⅰ．所有顶点的度之和为偶数

Ⅱ．边数大于顶点个数减 1

Ⅲ．至少有一个顶点的度为 1

A．只有Ⅰ　B．只有Ⅱ　C．Ⅰ和Ⅱ　D．Ⅰ和Ⅲ

（4）假设一个有 $n$ 个顶点和 $e$ 条弧的有向图用邻接表表示，则删除与某个顶点 $v_i$ 相关的所有弧的时间复杂度是（　B　）。

A．$O(n)$　B．$O(e)$　C．$O(n+e)$　D．$O(n*e)$

（5）已知一个有向图 8.30 所示，则从顶点 a 出发进行深度优先偏历，不可能得到的 DFS 序列为（　A　）。

A．a d b e f c　B．a d c e f b　C．a d c b f e　D．a d e f c b

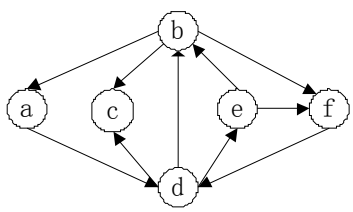



图 8.30　有向图

（6）无向图 G=（V，E），其中：V={a,b,c,d,e,f}，E={（a,b），（a,e），（a,c），（b,e），（c,f），（f,d），（e,d）}，对该图进行深度优先遍历，得到的顶点序列正确的是（　D　）。

A．a,b,e,c,d,f　B．a,c,f,e,b,d　C．a,e,b,c,f,d　D．a,e,d,f,c,b

（7）下列哪一个选项不是图 8.31 所示有向图的拓扑排序结果（ C　）。

A．AFBCDE　B．FABCDE　C．FACBDE　D．FADBCE

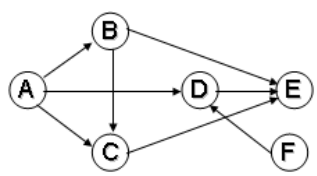



图 8.31　AOV 网

（8）判断一个有向图是否存在回路除了可以利用拓扑排序方法外，还可以利用（　D　）。

A．单源最短路 Dijkstra 算法　　B．所有顶点对最短路 Floyd 算法

C．广度优先遍历算法　　D．深度优先遍历算法

（9）在一个带权连通图 G 中，权值最小的边一定包含在 G 的（　A　）。

A．最小生成树中　　B．深度优先生成树中

C．广度优先生成树中　　D．深度优先生成森林中

（10）下图所示带权无向图的最小生成树的权为（　C　）。

A．14　　B．15　　C．17　　D．18

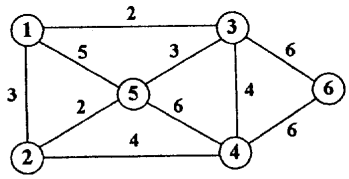



图 8.32　带权无向图

8.2　对于如图 8.33 所示的无向图，试给出：

（1）图中每个顶点的度；

（2）该图的邻接矩阵；

（3）该图的邻接表与邻接多重表；

（4）该图的连通分量。

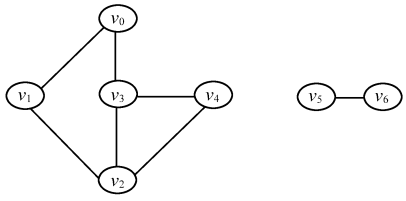



图 8.33　无向图

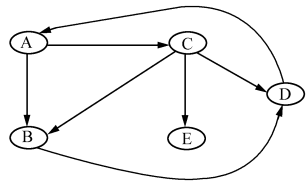



图 8.34　有向图

【答】:

（1）D($V_0$)=2；D（$V_1$）=2；D（$V_2$）=3；D（$V_3$）=3；D（$V_4$）=2；D（$V_5$）=2；D（$V_6$）=1。

（2）该图的邻接矩阵如图 8.33.1 所示。

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

图 8.33.1　邻接矩阵

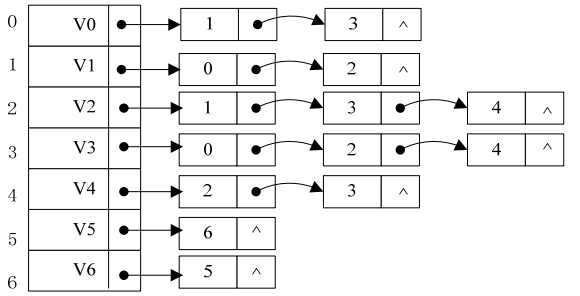



图 8.33.2　邻接表

（3）该图的邻接表如图 8.30.2 所示；该图的邻接多重表如图 8.30.3 所示。

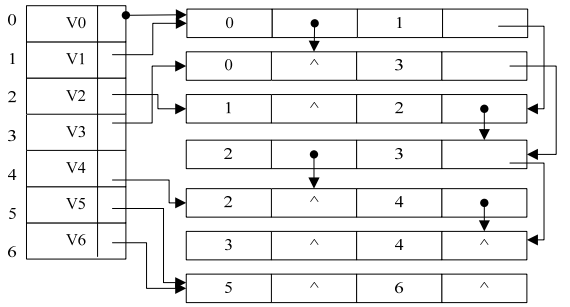



图 8.33.3　邻接多重表

（4）该图的两个连通分量如图 8.33.4 所示。

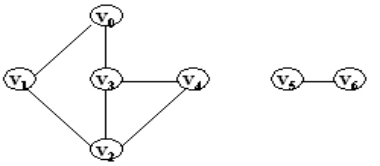



图 8. 33. 4　两个连通分量

8.3　对于如图 8.34 所示的有向图，试给出：

（1）顶点 D 的入度与出度；

（2）该图的出边表与入边表；

（3）该图的强连通分量。

【答】:

（1）顶点 D 的入度是 2；顶点 D 的出度为 1。

（2）该图的出边表如图 8.34.1 所示，该图的入边表如图 8.34.2 所示。

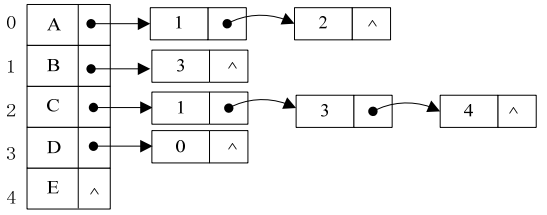



图 8.34.1　出边表

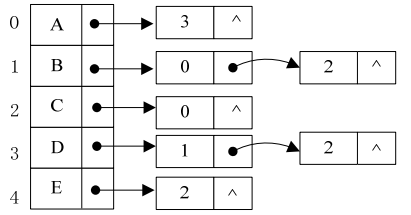



图 8.34.2　入边表

（3）该图的两个强连通分量如图 8.34.3 所示。

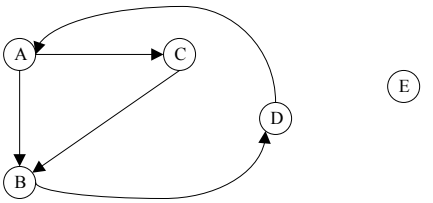



图 8.34.3 两个强连通分量

8.4　试回答下列关于图的一些问题。

（1）有 *n* 个顶点的有向强连通图最多有多少条边？最少有多少条边？

（2）表示一个有 500 个顶点，500 条边的有向图的邻接矩阵有多少个非零元素？

（3）G 是一个非连通的无向图，共有 28 条边，则该图至少有多少个顶点？

【答】:

（1）有 n 个顶点的有向强连通图昨多有 n(n-1)条边；最少有 n-1 条边。

（2）500 个。

（3）9 个顶点。

8.5　图 8.35 所示的是某个无向图的邻接表，试：

（1）画出此图；

（2）写出从顶点 A 开始的 DFS 遍历结果；

（3）写出从顶点 A 开始的 BFS 遍历结果。

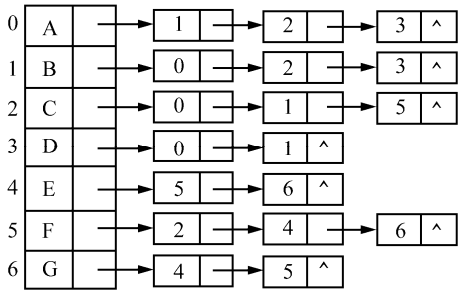



**图 8.35　题 8.5 的邻接表**

【答】:

（1）图 8.35 邻接表对应的无向图如图 8.35.1 所示。

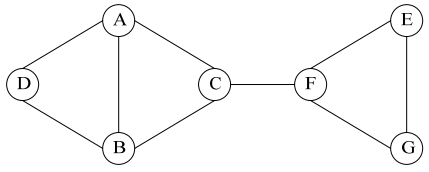



图 8.35.1

（2）从顶点 A 开始的 DFS 遍历结果是：A，B，C，F，E，G，D

（3）从顶点 A 开始的 BFS 遍历结果是：A，B，C，D，F，E，G

8.6　证明，用 Prim 算法能正确地生成一棵最小生成树。

【证明】:

Prim 算法是按照逐边加入的方法来构造最小生成树过程的。记构造过程中生成的子图为G'，显然 G'始终是一棵树。这是因为初始时 V(G')={$u_0$}，E（G'）=Φ，是一棵树。随后每一步都是向 G'中添加一个顶点同时添加一条边，始终保持 G'中所有顶点连通。这样，当 G'有 n 个顶点时是仅有 n-1 条边的连通图，所以 G'是一棵树。Prim 算法在执行过程中，始终能保证 G'=（V'，E'）是无向连通网络 G=（V，E）上某个最小代价生成树的连通图，如果该结论正确，则随着 V(G')顶点不断增加，当 V(G')=V(G)时，G'=(V',E')就是 G=(V,E)上的最小代价生成树。

下面证明 Prim 算法的每一步都能保证 G'=(V',E')是无向连通网络 G=(V,E)上某个代价生成树的子连通图。

初始时，G'仅有一个顶点 V（G'）={$u_0$}，E（G'）=Φ，设该树为 $T_1$，此时 G'显然是 G 的某个最小生成树的子连通图。现假设第 i 步 G'=(V',E')中含有 i 个顶点，不妨设该树为 $T_i$，在 G（V，E）中存在一棵最小生成树包含着 $T_i$，在第 i+1 步，存在 u∈$T_i$，v∉$T_i$,且(u,v)是最小两栖边，将顶点{v}，边（u,v）添加到树 $T_i$ 中，所得树 $T_{i+1}$ 是包含 V（$T_i$）+{v}顶点集的生成树，且具有 i+1 个顶点。根据 MST 性质，此时在 G=(V,E)中必存在一棵最小生成树包含着 $T_{i+1}$。由此可知，当 i=n 时，G'($T_n$)即为无向图 G 含有 n 个顶点的最小生成树。

当然在进行最小两栖边选择时，如果同时存在几个具有相同代价的最小两栖边，则可任选一条，因些 Prim 算法构造的最小生成树不是唯一的。但它们的最小（代价）总和是相等的。

8.7　证明，用 Kruskal 算法能正确地生成一棵最小生成树。

【证明】:

算法首先构造具有 n 个不同顶点的 n 个连通分量，然后在 E（G）中选择边(u,v)，若 u,v 顶点属于不同的两个连通分量，则把该边添加到树 T 中，每添加一条边就减少一个连通分量。当添加了共 n-1 条边时，就把 n 个连通分量变成一个连通分量，此时 T 就是具有 n 个顶点 n-1 条边的树。由于 n 是有限数，E（G）中边数也是有限的，所以算法具有有穷性。

不妨设 T 的边为（u1,v1）,(u2,v2)，…，(un-1,vn-1)，按权值从小到大排列，若存在一棵树 T'的代价总和小于 T 的代价总和，则必在 T'中存在一条边（u',v'）∈TE'，（u',v'） ∉TE，且(u',v')的代价小于(un-1,vn-1)的代价，这就说明(u',v')边没有选取的原因是因为 u',v'在同一个连通分量，添加（u',v'）将产生回路，说明 T'不可能是树（添加一条边没有减少一个连通分量，这样 T'的边数将大于 n-1）。这与 T'是一棵树的假设相矛盾。证毕。

8.8　对如图 8.36 所示的连通图，分别用 Prim 和 Kruskal 算法构造其最小生成树。

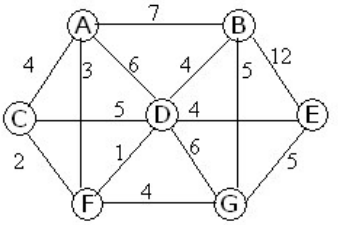
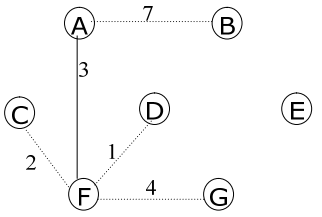


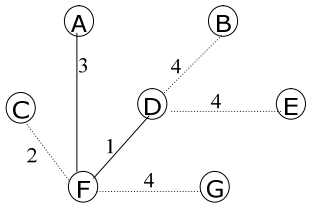
图 8.36　无向连通网

【答】:

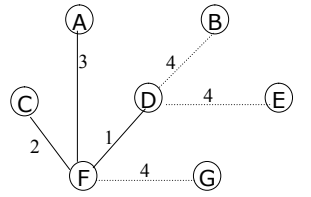
（1）采用 Prim 算法求解最小生成树的过程如图 8.36.1 所示。



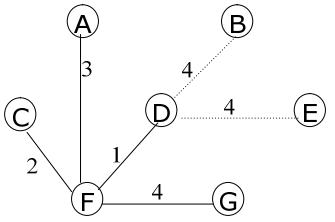

（a）选取（A，F）



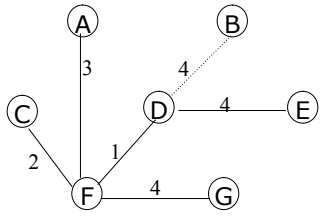

（b）选取（D,F）



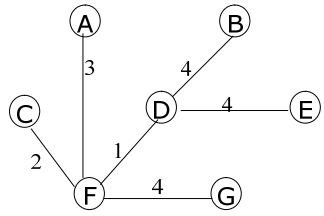

（c）选取（C,F）




（d）选取（F,G）



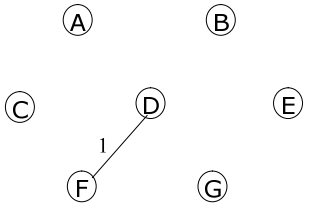

（e）选取（D,E）



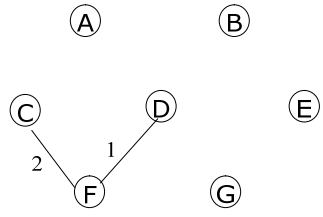

（f）选取（D,B）

图 8.36.1 Prim 算法求解最小生成树过程

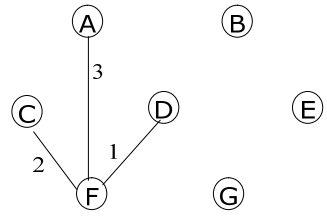
（2）采用 Kruskal 算法求解最小生成树时首先要对边进行由小到大进行排序，本题对边进行排序的结果是：（D,F）$_1$、（C,F）$_2$、（A,F）$_3$、（A,C）$_4$、（F,G）$_4$、（D,E）$_4$、（D,B）$_4$、（C,D）$_5$、（E,G）$_5$、（A,D）$_6$、（D,G）$_6$、（A,B）$_7$。根据 Kruskal 算法，构造最小生成树的过程如图 8.33.2 所示。



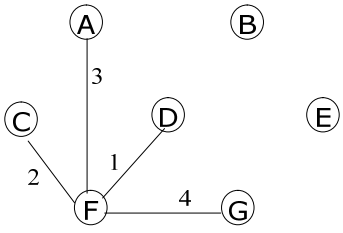

（a）选取（D，F）



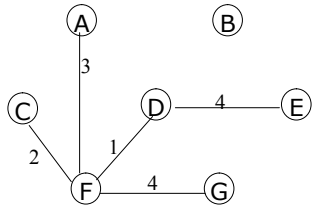

（b）选取（C,F）



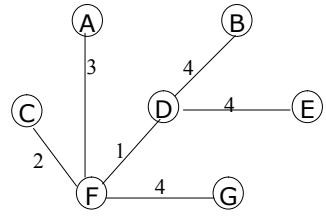

（c）选取（A,F）




（d）选取（F,G）




（e）选取（D,E）




（f）选取（D,B）

图 8.36.2　Kruskal 算法求解最小生成树过程

8.9　对于如图 8.37 所示的有向网，用 Dijkstra 方法求从顶点 A 到图中其他顶点的最短路径，并写出执行算法过程中距离向量 *d* 与路径向量 *p* 的状态变化情况。

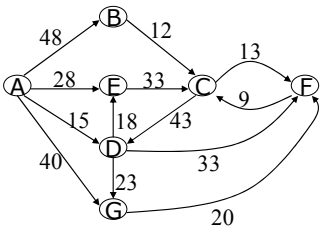



图 8.37　有向网

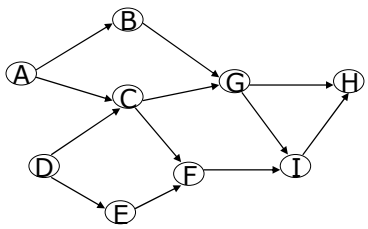



图 8.38　题 8.10 的 AOV 网

【答】：对于如图 8.37 所示的有向网，Dijkstra 方法求从顶点 A 到图中其它顶点的最短径时，距离向量与路径向量的状态变化如下表示：

| 循环 | 集合 S | v | 距离向量 d | | | | | | | 路径向量 p | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 初始化 | {A } | - | 0 | 48 | ∞ | 15 | 28 | ∞ | 40 | -1 | 0 | -1 | 0 | 0 | -1 | 0 |
| 1 | { AD} | 3 | 0 | 48 | ∞ | 15 | 28 | 48 | 38 | -1 | 0 | -1 | 0 | 0 | 3 | 3 |
| 2 | {ADE } | 4 | 0 | 48 | 61 | 15 | 28 | 48 | 38 | -1 | 0 | 4 | 0 | 0 | 3 | 3 |
| 3 | { ADG} | 6 | 0 | 48 | 61 | 15 | 28 | 48 | 38 | -1 | 0 | 4 | 0 | 0 | 3 | 3 |
| 4 | {ADGB } | 1 | 0 | 48 | 60 | 15 | 28 | 48 | 38 | -1 | 0 | 1 | 0 | 0 | 3 | 3 |
| 5 | {ADGBF} | 5 | 0 | 48 | 57 | 15 | 28 | 48 | 38 | -1 | 0 | 5 | 0 | 0 | 3 | 3 |
| 6 | {ADGBFC } | 2 | 0 | 48 | 57 | 15 | 28 | 48 | 38 | -1 | 0 | 5 | 0 | 0 | 3 | 3 |

从表中可以看出源点 A 到其它各顶点的最短距离及路径为：

A→B：48　路径：A→B

A→C：57　路径：A→D→F→C

A→D：15　路径：A→D

A→E：28　路径：A→E

A→F：48　路径：A→D→F

A→G：38　路径：A→D→G

8.10　试写出如图 8.38 所示的 AOV 网的 4 个不同的拓扑序列。

【答】：图 8.38 所示的 AOV 网的 4 个不同的拓扑序列为：

（1）ABDCEFGIH

（2）ABDECFGIH

（3）DABCEFGIH

（4）DAECBFGIH

8.11　计算如图 8.39 所示的 AOE 网中各顶点所表示的事件的发生时间 ve(j)，vl(j)，各边所表示的活动的开始时间 e(i)，l(i)，并找出其关键路径。

【答】：为描述方便，将 AOE 网中的所有边事件记为 $a_0$-$a_7$，如图 8.39 所示。按照关键路径算法，求得各顶事件的最早与最晚开始时间如下表所示。

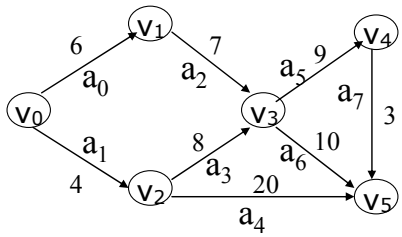



图 8.39　题 8.10 的 AOE 网

| 顶点 | $v_e$ | $v_l$ | 活动 | e | l | l-e | 关键活动 |
|---|---|---|---|---|---|---|---|
| $v_0$ | 0 | 0 | $a_0$ | 0 | 0 | 0 | √ |
| $v_1$ | 6 | 6 | $a_1$ | 0 | 1 | 1 | |
| $v_2$ | 4 | 5 | $a_2$ | 6 | 6 | 0 | √ |
| $v_3$ | 13 | 13 | $a_3$ | 4 | 5 | 1 | |
| $v_4$ | 22 | 22 | $a_4$ | 4 | 5 | 1 | |
| $v_5$ | 25 | 25 | $a_5$ | 13 | 13 | 0 | √ |
| | | | $a_6$ | 13 | 15 | 2 | |
| | | | $a_7$ | 22 | 22 | 0 | √ |

可见，该 AOE 网的关键路径是 $a_0,a_2,a_5,a_7$。　　（注：图中箭头指示求解的顺序）

8.12 无向图采用邻接表作为存储结构，试写出以下算法

（1）求一个顶点的度；

（2）往图中插入一个顶点；

（3）往图中插入一条边；

（4）删去图中某顶点；

（5）删去图中某条边。

【答】：本题的参考解答基于以下的存储结构：

```
#define m 20                    /*预定义图的最大顶点数*/
typedef char datatype;          /*顶点信息数据类型*/
typedef struct node{            /*边表结点*/
    int adjvex;                 /*邻接点*/
    struct node *next;
}edgenode;
typedef struct vnode{           /*头结点类型*/
    datatype vertex;            /*顶点信息*/
    edgenode *firstedge;        /*邻接链表头指针*/
}vertexnode;
```

```
typedef struct{                    /*邻接表类型*/
  vertexnode adjlist [m];          /*存放头结点的顺序表*/
  int n,e;                         /*图的顶点数与边数*/
}adjgraph;
```

（1）求一个顶点的度；

```
/*求无向图 g 的第 i 个顶点的度*/
int d(adjgraph g ,int i)
{int k=0;
  edgenode *p;
  p=g.adjlist[i].firstedge;
  while (p)
  {k++;
    p=p->next;
  }
  return k;
}
```

（2）往图中插入一个顶点；

```
void insertvex(adjgraph *g,datatype x)
{ int i=0,flag=0;
    /*查找待插入的结点是否存在*/
    while (!flag&&i<g->n)
    {if (g->adjlist[i].vertex==x) flag=1;
     i++;
     }
    if (flag)       {     printf("结点已存在!");
                          getch();
                          exit(0);
                    }
    if    (g->n>m)      { printf("邻接表已满！ ");
                          exit(0);
                    }
    g->adjlist[g->n].vertex=x;
    g->adjlist[g->n].firstedge=NULL;
    g->n++;
}
```

（3）往图中插入一条边；

```
/*在无向图 g 中插入无向边（i,j）*/
void insertedge(adjgraph *g,int i,int j)
{ edgenode *p,*s;
```

```
    int flag=0;
    if (i<g->n&&j<g->n)
      { p=g->adjlist[i].firstedge;
        while (!flag&&p)
          {if (p->adjvex==j) flag=1;
           p=p->next;
          }
        if (flag) {printf("边已存在!");
                   getch();
                   exit(0);
                   }
        /*插入无向边(i,j)*/
        s=(edgenode *)malloc(sizeof(edgenode));
        s->adjvex=j;                          /*邻接点序号为 j*/
        s->next=g->adjlist[i].firstedge;
        g->adjlist[i].firstedge=s;            /*将新结点*s 插入顶点 vi 的边表头部*/

        s=(edgenode *)malloc(sizeof(edgenode));
        s->adjvex=i;                          /*邻接点序号为 i*/
        s->next=g->adjlist[j].firstedge;
        g->adjlist[j].firstedge=s;            /*将新结点*s 插入顶点 vj 的边表头部*/
        }
    else
         printf("插入边不合法!");
  }
```

（4）删去图中某顶点；

/*下面的函数删除无向图中顶点编号为 i 的顶点。由于该顶点被删除，与这个顶点相关联的边也应该被删除，具体做法是，通过邻接表查找与顶点 i 邻接的其它所有顶点，在这些顶点的邻接边链表中删除编号为 i 的边结点，再将邻接表中最后一个顶点移动到第 i 个顶点在邻接表中的位置，相应地修改最后一个顶点在邻接表中的顶点编号为 i*/

```
    void deletevex(adjgraph *g,int i)
    {   int k;
        edgenode *pre,*p,*s;
        if (i>=0 && i<g->n)   /*顶点下标合法*/
          {   /*删除与原顶点 i 有关的所有边*/
              s=g->adjlist[i].firstedge;
              while (s)
              {k=s->adjvex;
               pre=NULL;
```

```
        p=g->adjlist[k].firstedge;
        while (p)
              {if   (p->adjvex==i)      /*结点编号是 i,应该删除*/
                if (!pre)                 /*边链表的第一个边结点*/
                    {g->adjlist[k].firstedge=p->next;
                     free(p);
                     p=g->adjlist[k].firstedge;
                    }
                else                      /*不是第一个边结点*/
                    {pre->next=p->next;
                     free(p);
                     p=pre->next;
                    }
               else       /*结点编号不是 i*/
                  {pre=p;
                   p=p->next;
                  }
              }
        g->adjlist[i].firstedge=s->next;
        free(s);          /*释放顶点 i 边链表上的结点*/
        s=g->adjlist[i].firstedge;
        }
        g->adjlist[i]=g->adjlist[g->n-1]; /*将最后一个结点换到第 i 个结点的位置*/
        p=g->adjlist[i].firstedge;
        /*由于第 n-1 个结点的编号被改为 i，所以调整所有与这个结点关联的边信息*/
        while (p)
        {     k=p->adjvex;
              s=g->adjlist[k].firstedge;
              while (s)
                      {if (s->adjvex==g->n-1)      /*将最后一个结点的编号改为 i*/
                                s->adjvex=i;
                       s=s->next;
                       }
               p=p->next;
        }
       g->n--;            /*结点个数减 1*/
     }
else
        printf("不存在指定的结点!\n");
```

```
    }
（5）删去图中某条边。
    void deleteedge(adjgraph *g,int i,int j)
    { edgenode *pre,*p;
      int k;
      if (i>=0 && i<g->n && j>=0 && j<g->n)       /*判断边是否有效*/
        {    pre=NULL;                             /*找无向边(i,j)并删除*/
             p=g->adjlist[i].firstedge;
             while (p && p->adjvex!=j)
                 {pre=p;
                  p=p->next;
                 }
             if (p)                          /*找到了被删除边结点*/
                 { if (!pre)                 /*是第一个边结点*/
                     g->adjlist[i].firstedge=p->next;
                   else
                     pre->next=p->next;
                   free(p);
                   pre=NULL;                 /*找无向边(j,i)并删除*/
                   p=g->adjlist[j].firstedge;
                   while (p&& p->adjvex!=i)
                     {pre=p;
                      p=p->next;
                     }
                   if (!pre)
                            g->adjlist[j].firstedge=p->next;
                   else
                            pre->next=p->next;
                   free(p);
                   g->e--;          /*边的个数减 1*/
                 }
             else {      printf("找不到指定的边!");
                         getch();
                         exit(0);
                  }
        }
        else
                {  printf("边不合法!\n");
                   exit(0);
```

```
                    }
        }
```

8.13 设有向图采用邻接表的存储结构（出边表），试写出求图中一个顶点的入度的算法。

【答】:

```
/*求有向图 g 中顶点 i 的入度，g 的存储结构为出边表，结构定义同题 8.11*/
int id(adjgraph *g,int i)
{int j,count=0;
 edgenode *p;
 for (j=0;j<g->n;j++)
   {p=g->adjlist[j].firstedge;
    while (p)
       {if (p->adjvex==i) count++;
        p=p->next;
       }
   }
 return count;
}
```

8.14 设计一个算法，不利用拓扑排序判断有向图中是否存在环。

【答】: 对于有向图进行深度优先遍历，如果从有向图上某个顶点 v 出发的遍历，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，则有向图中必定存在包含顶点 v 到顶点 u 的环。因此判断有一个有向图中是否有环可以借助图的深度优先遍历算法。

```
int visited[m];
void dfs(adjgraph g,int i)
{ edgenode *p;
   visited[i]=1;
   p=g.adjlist[i].firstedge;
   while (p)
      {   if (!visited[p->adjvex])
              dfs(g,p->adjvex);
           else                                  /* 深度优先遍历到已访问的结点，出现环*/
              {   printf(" found a circle!");
                  getch();
                  exit(0);
              }
          p=p->next;
      }
}
void findcircle(adjgraph g)
{   int i;
```

```
    for (i=0;i<g.n;i++)
        visited[i]=0;        /*初始化标志数组*/
    for (i=0;i<g.n;i++)
        if (!visited[i])     /*vi 未访问过*/
            dfs(g,i);
 }
```

8.15 编写一个非递归深度优先遍历图的函数。

【答】：图的深度优先遍历类似于二叉树的前序遍历，非递归实现时可以用一个栈保存已访问过的结点，这些结点的邻接点可能还没有全部访问完成，遍历过程中可能需要回溯。

参考程序如下：

```
#define MAXVEX 100     /*定义栈的最大容量*/
int visited[m];
void dfs(adjgraph g,int i)
{ /*以 vi 为出发点对邻接表表示的图 g 进行深度优先遍历*/
   edgenode *p;
   edgenode * stack[MAXVEX];   /*栈用来保存回溯点*/
   int top=-1;
   printf("visit vertex: %c ",g.adjlist[i].vertex);    /*访问顶点 i*/
   visited[i]=1;
   p=g.adjlist[i].firstedge;
   while (top>-1 || p)                  /*当栈不空或 p 不空时*/
        {   if   (p)                    /*优先处理 p 不空的情况*/
              if   (visited[p->adjvex])
                    p=p->next;
              else
                   {printf("%c ",g.adjlist[p->adjvex].vertex);
                    visited[p->adjvex]=1;
                    stack[++top]=p;
                    p=g.adjlist[p->adjvex].firstedge;
                   }
            else                        /*从栈中进行回溯*/
                   if (top>-1)
                        {p=stack[top--];
                         p=p->next;
                        }
        }
}
void dfstraverse(adjgraph g)
{   int i;
```

```
    for (i=0;i<g.n;i++)
        visited[i]=0;              /*初始化标志数组*/
    for (i=0;i<g.n;i++)
        if (!visited[i])           /*vi 未访问过*/
            dfs(g,i);
}
```

8.16 编写两个函数分别计算 AOE 网中所有活动的最早开始时间与最晚允许开始时间。

【答】：为了记录 AOE 网中所有活动的最早开始时间与最晚允许开始时间，定义边结点结构 edge 如下，其中 $v_i$,$v_j$ 存放边的起点与终点，e 存放该边表示的活动的最早开始时间，l 存放该边表示的活动的最晚允许开始时间。

```
typedef struct
{ int vi,vj;
  e,l;
} edge;      /*定义边结构类型，存放每个活动的最早开始时间与最晚允许开始时间*/
```

若活动 $a_k$ 的尾事件是 $v_i$，头事件是 $v_j$，则 e(k)就是 $v_i$ 的最早发生时间，l(k)是 $v_j$ 所允许的最晚开始时间减去活动 $a_k$ 的持续的时间。求解 AOE 网络事件的最早发生时间与最晚允许发生时间的算法参见教材算法 8.10。

```
/*求 AOE 网中各活动的最早开始时间*/
void e(aoegraph *gout,int ve[],edge active[])
{int i,k=0;
 edgenode* p;
 for (i=0;i<gout->n;i++)       /*对出边表中的每一条边进行求解*/
 {   p=gout->adjlist[i].firstedge;
     while (p)
     { active[k].vi=i;                 /*边的起点*/
       active[k].vj=p->adjvex;     /*边的终点*/
       active[k].e=ve[i];
       k++;
       p=p->next;
     }
 }
}

/*求 AOE 网中各活动的最晚允许开始时间*/
void l(aoegraph *gout,int vl[],edge active[])
{int i,k=0;
 edgenode *p;
 for (i=0;i<gout->n;i++)
 {   p=gout->adjlist[i].firstedge;
```

```
        while (p)
              {active[k].l=vl[p->adjvex]-p->len;
               p=p->next;
               k++;
              }
   }
}
```

# 第 9 章　检索

9.1　选择题

（1）在关键字序列（12，23，34，45，56，67，78，89，91）中二分查找关键字为 45、89 和 12 的结点时，所需进行的比较次数分别为（　B　）。

A．4，4，3　　B．4，3，3　　C．3，4，4　　D．3，3，4

（2）适用于折半查找的表的存储方式及元素排列要求为（　D　）。

A．链式方式存储，元素无序　　B．链式方式存储，元素有序

C．顺序方式存储，元素无序　　D．顺序方式存储，元素有序

（3）设顺序存储的线性表共有 123 个元素，按分块查找的要求等分成 3 块。若对索引表采用顺序查找来确定块，并在确定的块中进行顺序查找，则在查找概率相等的情况下，分块查找成功时的平均查找长度为（　B　）。

A．21　　B．23　　C．41　　D．62

（4）已知含 10 个结点的二叉排序树是一棵完全二叉树，则该二叉排序树在等概率情况下查找成功的平均查找长度等于（　B　）。

A．1.0　　B．2.9　　C．3.4　　D．5.5

（5）在图 9.27 所示的各棵二叉树中，二叉排序树是（　C　）。

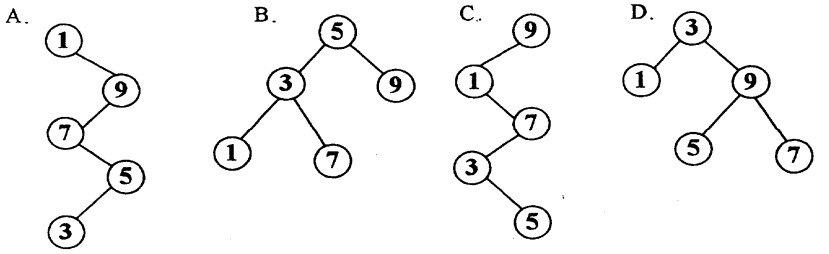



图 9.27　题（5）图

（6）由同一关键字集合构造的各棵二叉排序树（　B　）。

A．其形态不一定相同，但平均查找长度相同

B．其形态不一定相同，平均查找长度也不一定相同

C．其形态均相同，但平均查找长度不一定相同

D．其形态均相同，平均查找长度也都相同

（7）有数据{53，30，37，12，45，24，96}，从空二叉树开始逐步插入数据形成二叉排序树，若希望高度最小，则应该选择下列（　A　）的序列输入。

A．37,24,12,30,53,45,96　　B．45,24,53,12,37,96,30

C．12,24,30,37,45,53,96　　D．30,24,12,37,45,96,53

（8）若在 9 阶 B-树中插入关键字引起结点分裂，则该结点在插入前含有的关键字个数为（　C　）。

A．4　　B．5　　C．8　　D．9

（9）对于哈希函数 H(key) = key%13，被称为同义词的关键字是（　D　）。

A．35 和 41　　B．23 和 39　　C．15 和 44　　D．25 和 51

（10）下列叙述中，不符合 m 阶 B 树定义要求的是（　D　）。

A．根结点最多有 m 棵子树

B．所有叶结点都在同一层上

C．各结点内关键字均升序或降序排列

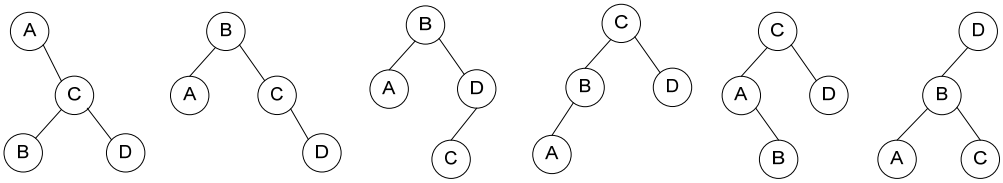
D．叶结点之间通过指针链接

9.2 在分块检索中，对 256 个元素的线性表分成多少块最好？每块的最佳长度是多少？若每块的长度为 8，其平均检索的长度是多少？

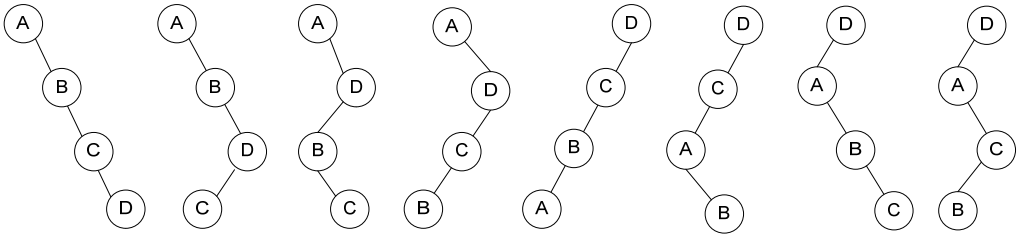
【答】：对 256 个元素的线性表分成 16 块，每块的最佳长度是 17；若每块的长度为 8，当采用顺序检索方法确定所在的块时平均检索长度是 21，当采用二分检索方法确定所在的块时平均检索长度是 9。

9.3 设有关键码 A、B、C 和 D，按照不同的输入顺序，共可能组成多少不同的二叉排序树。请画出其中高度较小的 6 种。

【答】：共可能组成 14 种不同形态的二叉排序树。其中高度较小的 6 种如图 9.28（a ）所示。其它 8 种分别是高度为 4 的二叉排序树如图 9.28(b)所示。




（a）　4 个结点组成的高度较小的 6 棵二叉排序树



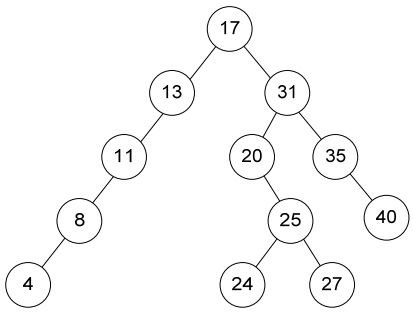

（b）　4 个结点组成的高度为 4 的二叉排序树

图 9.28　4 个结点输入序列构成的不同二叉排序树

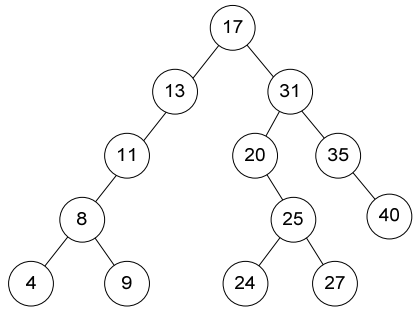
9.4 已知序列 17，31，13，11，20，35，25，8，4，11，24，40，27。请画出由该输入序列构成的二叉排序树，并分别给出下列操作后的二叉排序树。

（1）插入数据 9；（2）删除结点 17；（3）再删除结点 13

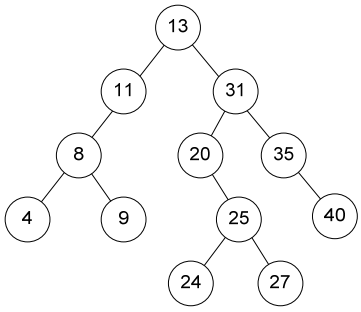
【答】：该序列的二叉排序树如图 9.29（a）所示，插入数据 9 后的二叉排序树如图 9.29（b）所示，删除结点 17 后的二叉排序树如图 9.29（c）所示，再删除结点 13 后的二叉排序树如图 9.29（d）所示。



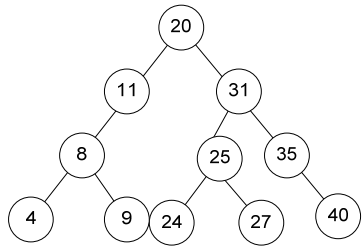

（a） 二叉排序树




（b） 插入 9 之后的二叉排序树




（c）删除 17 之后的二叉排序树




（d）删除 13 之后的二叉排序树

图 9.29　二叉树结点插入与删除操作示例

9.5 试写一算法判别给定的二叉树是否为二叉排序树，设此二叉树以二叉链表为存储结构，且树中结点的关键字均不相同。

【答】：判定二叉树是否为二叉排序树可以建立在二叉树中序遍历的基础上，在遍历中附设一指针 pre 指向树中当前访问结点的中序直接前驱，每访问一个结点就比较前驱结点 pre 和此结点是否有序；若遍历结束后各结点和其中序直接前驱均满足有序，则此二叉树即为二叉排序树，否则不是二叉排序树。

二叉树存储结构定义为：

```
typedef int datatype;
typedef struct node                    /*二叉树结点定义*/
  { datatype data;
     struct node *lchild,*rchild;
   }bintnode;
typedef bintnode *bintree;
```

函数 bisorttree()用于判断二叉树 t 是否为二叉排序树，初始时 pre=NULL；flag=1；结束时若 flag==1，则此二叉树为二叉排序树，否则此二叉树不是二叉排序树。

```
void bisorttree(bintree t,bintree *pre,int *flag)
{if (t&& *flag==1)
   {   bisorttree(t->lchild,pre,flag);       /*判断左子树*/
       if (pre==NULL)     /*访问中序序列的第一个结点时不需要比较*/
```

```
        { *flag=1;
          *pre=t;
          }
      else              /*比较 t 与中序直接前驱 pre 的大小（假定无相同关键字）*/
      {if   ((*pre)->data<t->data)
            {*flag=1;
             *pre=t;
             }
       else     /* pre 与 t 无序  */
            *flag=0;
      }
      bisorttree(t->rchild,pre,flag);        /*判断右子树*/
    }
  }
```

9.6 设 T 是一棵给定的查找树，试编写一个在树 T 中删除根结点值为 a 的子树的程序。要求在删除的过程中释放该子树中所有结点所占用的存储空间。这里假设树 T 中的结点采用二叉链表存储结构。

【答】：删除二叉树可以采用后序遍历方法，先删除左子树，再删除右子树，最后删除根结点。本题先在指定的树中查找值为 a 的结点，找到后删除该棵子树。相关函数实现如下（二叉排序树的结构定义同题 9.5）

```
/*删除以 t 为根的二叉树*/
void deletetree(bintree *t)
{if (*t)
    {deletetree(&(*t)->lchild);      /*递归删除左子树*/
     deletetree(&(*t)->rchild);      /*递归删除右子树*/
     free(*t);          /*删除根结点*/
    }
}
/*删除二叉树中以根结点值为 a 的子树*/
void deletea(bintree *t,datatype a)
{bintree pre=NULL,p=*t;
 while (p&&p->data!=a)   /*查找值为 a 的结点*/
   {pre=p;
    p=(a<p->data)?p->lchild:p->rchild;
   }
 if (!pre) *t=NULL;         /*树根*/
     else     /*非树根*/
         if (pre->lchild==p) pre->lchild=NULL;
              else pre->rchild=NULL;
```

```
    deletetree(&p);   /*删除以 p 为根的子树*/
}
```

9.7 含有 12 个节点的平衡二叉树的最大深度是多少（设根结点深度为 0），并画出一棵这样的树。

【答】：含有 12 个节点的平衡二叉树的最大深度是 4（设根结点深度为 0），如果根结点的深度为 1 则本题的答案为 5，该树如图 9.30 所示。

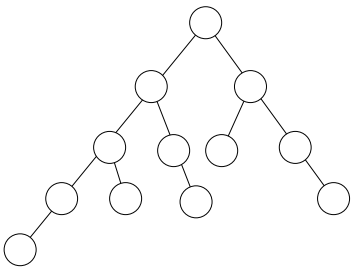
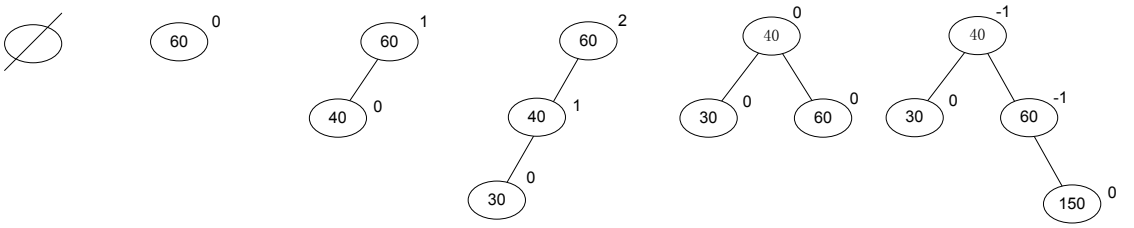


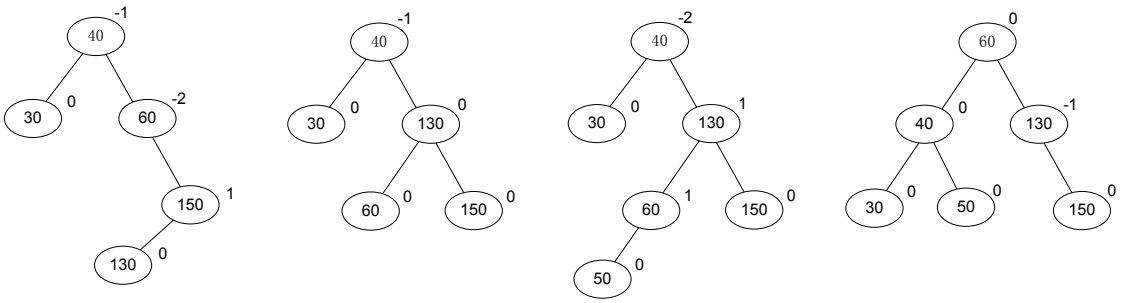
图 9.30　具有 12 个结点的深度为 4 的平衡二叉树

9.8 试用 Adelson 插入方法依次把结点值为 60，40，30，150，130，50，90，80，96，25 的记录插入到初始为空的平衡二叉排序树中，使得在每次插入后保持该树仍然是平衡查找树。请依次画出每次插入后所形成的平衡查找树。

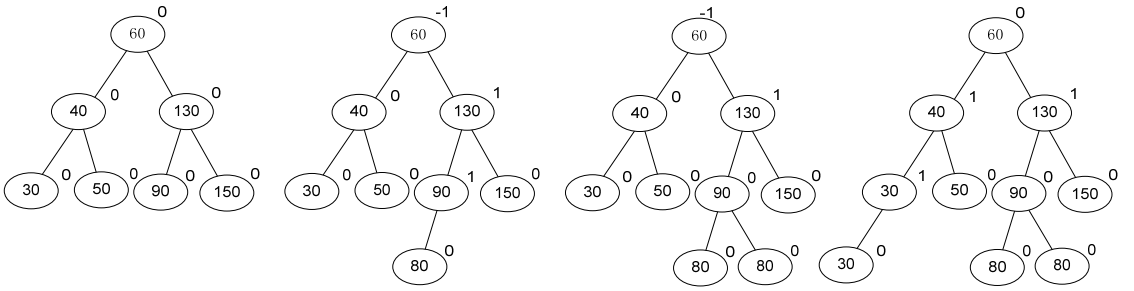
【答】：由结点序列构成的平衡二叉排序树如图 9.31 所示。




（a）空树（b）插入 60（c）插入 40　（d）插入 30　（e）LL 型调整　　（f）插入 150




（g）插入 130　　（h）RL 型调整　　（i）插入 50　　（j）RL 型调整




（k）插入 90　　（l）插入 80　　（m）插入 96　　（n）插入 25

图 9.31　AVL 树的插入过程

9.9 结点关键字 $k_1$，$k_2$，$k_3$，$k_4$，$k_5$ 为一个有序序列，它们的相对使用频率分别为 $p_1$=6，$p_2$=8，$p_3$=12，$p_4$=2，$p_5$=16，外部结点的相对使用频率分别为 $q_0$=4，$q_1$=9，$q_2$=8，$q_3$=12，$q_4$=3，$q_5$=2。试构造出有序序列 $k_1$，$k_2$，$k_3$，$k_4$，$k_5$ 所组成的最优查找树。

【答】:（略）

9.10 证明 Huffman 算法能正确地生成一棵具有最小带权外部路枝长度的二叉树。

【答】: 哈夫曼提出了一种构造最优前缀编码的贪心算法，由此产生的编码方案称为哈夫曼算法。哈夫曼算法以自底向上的方式构造表示最优前缀码的二叉树 T。算法以 C 个叶结点开始，执行 C-1 次的“合并”运算后产生最终所要求的树 T。要证明哈夫曼算法的正确性，只要证明最优前缀码问题具有贪心选择性质和最优子结构性质。

（1）贪心选择性质

设 C 是编码字符集，C 中字符 c 的频率为 f（c）。又设 x 和 y 是 C 中具有最小频率的两个字符，则存在 C 的一个最优前缀编码使 x 和 y 具有相同码长且仅最后一们编码不同。

证明：设二叉树 T 表示 C 的任意一个最优前缀码。我们要证明可以对 T 作适当修改后得到一棵新的二叉树 T"，使得在新树中，x 和 y 是最深中子且为兄弟。同时新树 T"表示的前缀码也是 CS 的一个最优前缀码。如果我们能做到这一点，则 x 和 y 在 T"表示的最优前缀码中就具有相同的码长且仅最后一位编码不同。

设 b 和 c 是二叉树 T 的最深叶子且为兄弟。不失一般性，可设 f（b）≤f（c）,f（x）≤f（y）。由于 x 和 y 是 C 中具有最小频率的两个字符，故 f（x）≤f（b），f（y）≤f（c）。首先在树 T 中交换叶子 b 和 x 位置得到树 T'，然后在树 T"再交换叶子 c 和 y 的位置，得到树 T"。如下图所示。

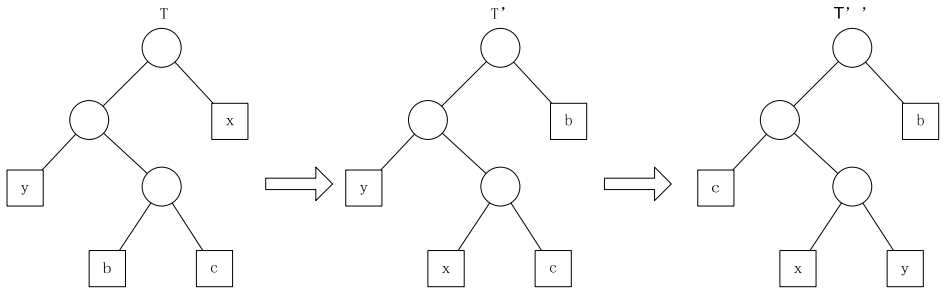



图　　编码树 T 的变换

由此可知，树 T 和 T'表示的前缀码的平均码长之差为

B（T）-B（T'）$= \sum_{c \in CS} f(c)d_T(c) - \sum_{c \in CS} f(c)d_{T'}(c)$

$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b)$

$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_T(b) - f(b)d_T(x)$

$= (f(b) - f(x))(d_T(b) - d_T(x)) \geq 0$

最后一个不等式是因为 f（b）-f（x）和 $d_T$（b）-$d_T$（x）均为非负。

类似地，可以证明在 T'中交换 y 与 c 的位置也不增加平均码长，即 B（T'）-B（T"）也是非负的。由此可知 B（T"）≤B（T'）≤B（T）。另一方面，由于 T 所表示的前缀码是最优的，故 B（T）≤B（T"）。因此，B（T）=B（T"），即 T"表示的前缀码也是最优前缀码，且 x 和 y 具有最长的码长，同时仅最后一位编码不同。

（2）最优子结构性质

设 T 是表示字符集 C 的一个最优前缀码的二叉树。C 中字符 c 的出现频率为 f（c）。设 x 和 y 是树 T 中的两个叶子且为兄弟，z 是它们的双亲。若将 z 看作是具有频率 f（z）=f（x）+f（y）的字符，则树 T'=T-{x，y}表示字符集 C'=C-{x,y}∪{z}的确一个最优前缀码。

证明：我们首先证明 T 的平均码长 B（T）可用 T'的平均码长 B（T'）来表示。

事实上，对任意 c∈C-{x，y}有，$d_T(c)=d_{T'}$（c），故有 f（c）$d_T$（c）=f（c）$d_{T'}$（c）。

另一方面，$d_T$（x）=$d_T$（y）=$d_{T'}$（z）+1，故

f（x）$d_T$（x）+f（y）$d_T$（y）=（f（x）+f（y））($d_{T'}$（z）+1)

=f（x）+f（y）+f（z）$d_{T'}$（z）

由此即知，B（T）=B（T'）+f（x）+f（y）。

由 T'所表示的字符集 C'的前缀码不是最优的，则有 T"表示的 C'的前缀码使得 B（T"）<B（T'）。由于 z 被看作是 C'中的一个字符，故 z 在 T"中是一树叶。若将 x 和 y 加入 T"中作为 z 的儿子，则得到表示字符集 C 的前缀码的二叉树 T"'，且有

B（T"'）=B（T"）+f（x）+f（y）<B（T'）+f（x）+f（y）=B（T）

这与 T 的最优性矛盾。故 T'所表示的 C'的前缀码是最优的。

由贪心选择性质和最优子结构性质立即可推出：哈夫曼算法是正确的，Huffman 算法能正确地生成一棵具有最小带权外部路枝长度的二叉树。Huffman Tree 产生 C 的确一棵最优前缀编码树。

9.11 假设通讯电文中只用到 A，B，C，D，E，F 六个字母，它们在电文中出现的相对频率分别为：8，3，16，10，5，20，试为它们设计 Huffman 编码。

【答】：由 A，B，C，D，E，F 建立的 Huffman 树如下：

各字符对应的 Huffman 编码为：

A：001

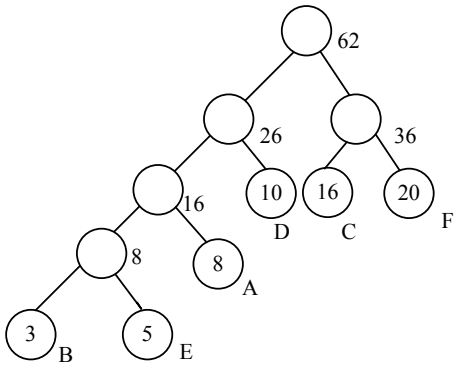
B：0000

C：10

D：01

E：0001

F：11




9.12 含有 9 个叶子结点的 3 阶 B-树中至少有多少个非叶子结点？含有 10 个叶子结点的 3 阶 B-树中至少有多少个非叶子结点？

【答】:（略）

9.13 编写在 B-树中插入结点与删除结点的算法程序。

【答】:（略）

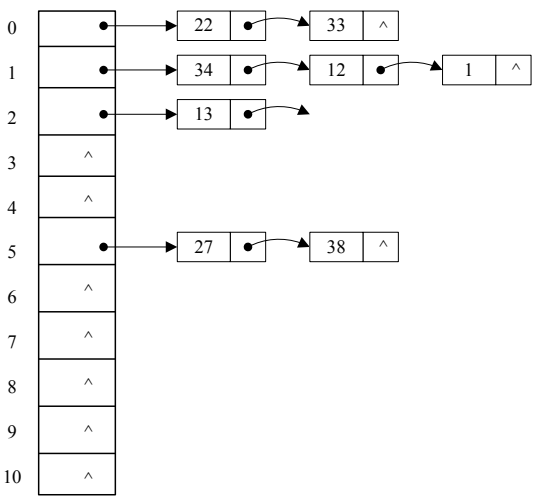
9.14 用依次输入的关键字 23、30、51、29、27、15、11、17 和 16 建一棵 3 阶 B-树，画出建该树的变化过程示意图（每插入一个结点至少有一张图）。

【答】:（略）

9.15 设散列表长度为 11，散列函数 H（x）=x % 11，给定的关键字序列为：1，13，12，34，38，33，27，22。试画出分别用拉链法和线性探测法解决冲突时所构造的散列表，并求出在等概率的情况下，这两种方法查找成功和失败时的平均查找长度。

【答】:

（1）拉链法构造的散列表如下：



查找成功时的平均查找长度为：13/8

查找失败时的平均查找长度为：8/11

（2）线性探测法构造的散列表如下：

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 33 | 1 | 13 | 12 | 34 | 38 | 27 | 22 | | | |

查找成功时的平均查找长度为：（1+1+3+4+1+1+2+8）/8=21/8

查找失败时的平均查找长度为：（1+2+3+...+11）/11=6 次

9.16 设散列表为 T[0..12]，即表的大小 m=13。现采用再哈希法（双散列法）解决冲突。散列函数和再散列函数分别为：

$H_0$（k）=k % 13、 $H_i$=（$H_{i-1}$+REV（k+1）%11+1）%13；i=1，2，…，m-1

其中，函数 REV（x）表示颠倒 10 进制数的各位，如 REV（37）=73，REV（1）=1 等。

若插入的关键码序列为{2，8，31，20，19，18，53，27}。

（1）试画出插入这 8 个关键码后的散列表。

（2）计算检索成功的平均查找长度 ASL。

【答】:（1）

$H_0$（2）=2，2 存入 2 号。

$H_0$（8）=8，8 存入 8 号。

$H_0$（31）=5，31 存入 5 号。

$H_0$（20）=7，20 存入 7 号。

$H_0$（19）=6，19 存入 6 号。

$H_0$（18）=5，出现碰撞，$H_1$（18）=（5+91%11+1）%13=9，18 存入 9 号。

$H_0$（53）=1，53 存入 1 号。

$H_0$（27）=1，出现碰撞，$H_1$（27）=（1+82%11+1）%13=7，发生碰撞，$H_2$（27）=（7+82%11+1）%13=0，27 存入 0 号。

插入 8 个关键码序列以后的散列表如下所示：

| 位置 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 关键码 | 27 | 53 | 2 | | | 31 | 19 | 20 | 8 | 5 | | | |

（2）检索成功的平均查找长度是 1.375。

# 第 10 章　内排序

10.1　选择题。

（1）下列序列中，（　A　）是执行第一趟快速排序后得到的序列。

A．[da,ax,eb,de,bb]ff[ha,gc]　　B．[cd,eb,ax,da]ff[ha,gc,bb]

C．[gc,ax,eb,cd,bb]ff[da,ha]　　D．[ax,bb,cd,da]ff[eb,gc,ha]

（2）下列说法错误的是（　D　）

A．冒泡排序在数据有序的情况下具有最少的比较次数。

B．直接插入排序在数据有序的情况下具有最少的比较次数。

C．二路归并排序需要借助 $O(n)$ 的存储空间。

D．基数排序适合于实型数据的排序。

（3）下面的序列中初始序列构成最小堆（小根堆）的是（　D　）。

A．10、60、20、50、30、26、35、40

B．70、40、36、30、20、16、28、10

C．20、60、50、40、30、10、8、72

D．10、30、20、50、40、26、35、60

（4）在下列算法中，（　C　）算法可能出现下列情况：在最后一趟开始之前，所有的元素都不在其最终的位置上。

A．堆排序　　B．插入排序　　C．冒泡排序　　D．快速排序

（5）若需在 $O(n\log n)$ 的时间内完成对数组的排序，且要求排序算法是稳定的，则可选择的排序方法是（　A　）。

A．归并排序　　B．堆排序　　C．快速排序　　D．直接插入排序

（6）以下排序方法中，不稳定的排序方法是（　AC　）。

A．直接选择排序　　B．二分法插入排序

C．堆排序　　D．基数排序

（7）一个序列中有 10000 个元素，若只想得到其中前 10 个最小元素，最好采用（　B　）方法。

A．快速排序　　B．堆排序　　C．插入排序　　D．二路归并排序

（8）若要求尽可能快地对实数数组进行稳定的排序，则应选（　C　）

A．快速排序　　B．堆排序　　C．归并排序　　D．基数排序

（9）排序的趟数与待排序元素的原始状态有关的排序方法是（　A　）。

A．冒泡排序　　B．快速排序　　C．插入排序　　D．选择排序

（10）直接插入排序在最好情况下的时间复杂度为（　A　）。

A．$O(n)$　　B．$O(\log n)$　　C．$O(n\log n)$　　D．$O(n^2)$

10.2　给出初始待排序码{27，46，5，18，16，51，32，26}使用下面各种排序算法的状态变化示意图：

（1）直接插入排序；

【答】:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| [27] | 46 | 5 | 18 | 16 | 51 | 32 | 26 |
| [27 | 46] | 5 | 18 | 16 | 51 | 32 | 26 |
| [5 | 27 | 46] | 18 | 16 | 51 | 32 | 26 |
| [5 | 18 | 27 | 46] | 16 | 51 | 32 | 26 |
| [5 | 16 | 18 | 27 | 46] | 51 | 32 | 26 |
| [5 | 16 | 18 | 27 | 46 | 51] | 32 | 26 |
| [5 | 16 | 18 | 27 | 32 | 46 | 51] | 26 |
| [5 | 16 | 18 | 26 | 27 | 32 | 46 | 51] |

（2）表插入排序；

【答】:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| key | | 27 | 46 | 5 | 18 | 16 | 51 | 32 | 26 |
| link | | | | | | | | | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

（a）初始存储状态

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| key | | 27 | 46 | 5 | 18 | 16 | 51 | 32 | 26 |
| link | 1 | 0 | | | | | | | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

（b）

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| key | | 27 | 46 | 5 | 18 | 16 | 51 | 32 | 26 |
| link | 1 | 2 | 0 | | | | | | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

（c）

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| key | | 27 | 46 | 5 | 18 | 16 | 51 | 32 | 26 |
| link | 3 | 2 | 0 | 1 | | | | | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

（d）

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| key | | 27 | 46 | 5 | 18 | 16 | 51 | 32 | 26 |
| link | 3 | 2 | 0 | 4 | 1 | | | | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

（e）

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| key | | 27 | 46 | 5 | 18 | 16 | 51 | 32 | 26 |
| link | 3 | 2 | 0 | 5 | 1 | 4 | | | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

（f）

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| key | | 27 | 46 | 5 | 18 | 16 | 51 | 32 | 26 |
| link | 3 | 2 | 6 | 5 | 1 | 4 | 0 | | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

（g）

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| key | | 27 | 46 | 5 | 18 | 16 | 51 | 32 | 26 |
| link | 3 | 7 | 6 | 5 | 1 | 4 | 0 | 2 | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

(h)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| key | | 27 | 46 | 5 | 18 | 16 | 51 | 32 | 26 |
| link | 3 | 7 | 6 | 5 | 8 | 4 | 0 | 2 | 1 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

(i)

（3）二分法插入排序；

【答】：这里只给出最后一趟将 26 二分插入前面的有序序列的过程。

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 16 | 18 | 27 | 32 | 46 | 51 | **26** |
| low | | | mid | | | high | |

26<27,将 high=mid-1;

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 16 | 18 | 27 | 32 | 46 | 51 | **26** |
| low | mid | high | | | | | |

26>16,将 low=mid+1;

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 16 | 18 | 27 | 32 | 46 | 51 | **26** |
| | | low,high | | | | | |

26>16，将 low=mid+1;

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 16 | 18 | 27 | 32 | 46 | 51 | **26** |
| | | high | low | | | | |

此时，high<low,二分定位结束，将 4..7 的所有元素后移一位，将 26 插入到 low 指定的位置。

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 16 | 18 | 27 | 32 | 46 | 51 | **26** |
| | | high | low | | | | |

最终排序结果为：

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 16 | 18 | ***26*** | 27 | 32 | 46 | 51 |

（4）直接选择排序；

【答】:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 27 | 46 | ***5*** | 18 | 16 | 51 | 32 | 26 |
| i=1 | | k=3 | | | | | |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 46 | 27 | 18 | ***16*** | 51 | 32 | 26 |
| | i=2 | | | k=5 | | | |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 16 | 27 | ***18*** | 46 | 51 | 32 | 26 |
| | | i=3 | k=4 | | | | |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 16 | 18 | 27 | 46 | 51 | 32 | ***26*** |
| | | | i=4 | | | | k=8 |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 16 | 18 | 26 | 46 | 51 | 32 | ***27*** |
| | | | | i=5 | | | k=8 |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 16 | 18 | 26 | 27 | 51 | ***32*** | 46 |
| | | | | | i=6 | k=7 | |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 16 | 18 | 26 | 27 | 32 | 51 | ***46*** |
| | | | | | | i=7 | k=8 |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 16 | 18 | 26 | 27 | 32 | 46 | 51 |
| | | | | | | | i=8 |

（5）冒泡排序；

【答】:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 27 | 46 | 5 | 18 | 16 | 51 | 32 | 26 |
| 27 | 5 | 18 | 16 | 46 | 32 | 26 | [51] |
| 5 | 18 | 16 | 27 | 32 | 26 | [46 | 51] |
| 5 | 16 | 18 | 27 | 26 | [32 | 46 | 51] |
| 5 | 16 | 18 | 26 | [27 | 32 | 46 | 51] |
| 5 | 16 | 18 | [26 | 27 | 32 | 46 | 51] |

（6）快速排序；

【答】:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 27 | 46 | 5 | 18 | 16 | 51 | 32 | 26 |
| [26 | 16 | 5 | 18] | 27 | [51 | 32 | 46] |
| [18 | 16 | 5] | 26 | 27 | [51 | 32 | 46] |
| [5 | 16] | 18 | 26 | 27 | [51 | 32 | 46] |
| 5 | [16] | 18 | 26 | 27 | [51 | 32 | 46] |
| 5 | 16 | 18 | 26 | 27 | [46 | 32] | 51 |
| 5 | 16 | 18 | 26 | 27 | [32] | 46 | 51 |
| 5 | 16 | 18 | 26 | 27 | 32 | 46 | 51 |

（7）二路归并排序；

【答】:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| [27] | [46] | [5] | [18] | [16] | [51] | [32] | [26] |
| [27 | 46] | [5 | 18] | [16 | 51] | [26 | 32] |
| [5 | 18 | 27 | 46] | [16 | 26 | 32 | 51] |
| [5 | 16 | 18 | 26 | 27 | 32 | 46 | 51] |

（8）基数排序。

【答】:

10.3　在冒泡排序过程中，有的排序码在某一次起泡中可能朝着与最终排序相反的方向移动，试举例说明。在快速排序过程中是否也会出现这种现象？

【答】:

例如，80，70，40，50 在第一趟冒泡后序列变为 70，40，50，80，70 朝着与最终排序相反的方向移动了。

在快速排序中也会出现这种情况，例如，对序列[90，32，25，50，60]以 90 划分时，序列变为[60，32，25，50]，90，其中 60 也朝与最终排序相反的方向移动了。

10.4　修改冒泡排序算法，使第一趟把排序序码最大的记录放到最末尾，第二趟把排序码最小的记录在最前面，如此反复进行，达到排序的目的。

【答】:

```
#include "table.h"
/*冒泡排序*/
```

```
void bubblesort(table *L)
{     int i,j,k,done;
i=1;j=L->length;done=1;
while (done)
{
      done=0;
      for (k=i;k<=j;k++)
            if (L->r[k+1].key<L->r[k].key)
            {
                  L->r[0]=L->r[k];
                  L->r[k]=L->r[k+1];
                  L->r[k+1]=L->r[0];
                  done=1;
            }
      j--;
      done=0;
      for (k=j;k>=i;k--)
            if (L->r[k].key<L->r[k-1].key)
            {
                  L->r[0]=L->r[k];
                  L->r[k]=L->r[k-1];
                  L->r[k-1]=L->r[0];
                  done=1;
            }
      i++;
}
}

int main()
{ table L;                  /*定义表 L*/
   input(&L,"in.txt");      /*从文件 in.txt 中读入排序数据*/
   print(L);                /*输出原表*/
   bubblesort(&L);
   print(L);                /*输出排序后的表*/
}
```

10.5　对习题 10.2 中给出的初始数列，给出堆排序中建堆过程示意图。

10.6 ［计数排序］一个记录在已排序的文件中的位置，可由此文件中比该记录排序码小的记录的个数而定，由此得到一个简单的排序方法，对于每一个记录，增加一个 count 字段确

定在已排序的文件中位于该记录之前的记录的个数，写一个算法，确定一个无序文件中的每个记录的 count 的值，并证明若文件有 $n$ 个记录，则至多进行 $n(n-1)/2$ 次排序码比较，即可确定所有记录的 count 值。

10.7　设计一个算法，重新排列一组整数位置，使所有负值的整数位于正值的整数之前（不要对这一组整数进行排序，要求尽量减少算法中的交换次数）。

10.8　对本章中的各种排序算法，说明哪些是稳定的？哪些是不稳定的？对不稳定的排序算法举例说明。

10.9　总结本章中各种排序算法的特点，分析比较各算法的时间、空间复杂度及附加存储空间情况。

10.10　一油田欲建设一条连接油田内 $n$ 口油井的主输油管道，管道由东向西，从每一口油井都有一条支输油管道和主输油管道相连。如果知道每口油井的具体位置，应该如何确定主输油管道的建设位置，使得所有支输油管道的长度之和最小。

10.11　某大学一、二、三年级的学生报名参加一知识竞赛，报名信息包括年级和姓名，已知这 3 个年级都有学生报名，报名信息中的年级用 1、2、3 表示，设计一个算法对所有报名参赛学生按年级排序，要求排序算法的时间复杂度是线性的。