第 1 章　绪论

1.1 简述下列术语：数据，数据元素、数据对象、数据结构、存储结构、数据类型和抽象数据类型。

解：数据是对客观事物的符号表示。在计算机科学中是指所有能输入到计算机中并被计算机程序处理的符号的总称。

数据元素是数据的基本单位，在计算机程序中通常作为一个整体进行考虑和处理。

数据对象是性质相同的数据元素的集合，是数据的一个子集。

数据结构是相互之间存在一种或多种特定关系的数据元素的集合。

存储结构是数据结构在计算机中的表示。

数据类型是一个值的集合和定义在这个值集上的一组操作的总称。

抽象数据类型是指一个数学模型以及定义在该模型上的一组操作。是对一般数据类型的扩展。

1.2 试描述数据结构和抽象数据类型的概念与程序设计语言中数据类型概念的区别。

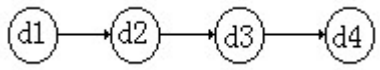
解：抽象数据类型包含一般数据类型的概念，但含义比一般数据类型更广、更抽象。一般数据类型由具体语言系统内部定义，直接提供给编程者定义用户数据，因此称它们为预定义数据类型。抽象数据类型通常由编程者定义，包括定义它所使用的数据和在这些数据上所进行的操作。在定义抽象数据类型中的数据部分和操作部分时，要求只定义到数据的逻辑结构和操作说明，不考虑数据的存储结构和操作的具体实现，这样抽象层次更高，更能为其他用户提供良好的使用接口。

1.3 设有数据结构(D,R)，其中

$$D=\{d1,d2,d3,d4\},\ R=\{r\},\ r=\{(d1,d2),(d2,d3),(d3,d4)\}$$

试按图论中图的画法惯例画出其逻辑结构图。

解：



1.4 试仿照三元组的抽象数据类型分别写出抽象数据类型复数和有理数的定义（有理数是其分子、分母均为自然数且分母不为零的分数）。

解：

```
ADT Complex{
    数据对象：D={r,i|r,i 为实数}
    数据关系：R={<r,i>}
    基本操作：
        InitComplex(&C,re,im)
            操作结果：构造一个复数 C，其实部和虚部分别为 re 和 im
        DestroyCmoplex(&C)
            操作结果：销毁复数 C
        Get(C,k,&e)
            操作结果：用 e 返回复数 C 的第 k 元的值
        Put(&C,k,e)
            操作结果：改变复数 C 的第 k 元的值为 e
        IsAscending(C)
            操作结果：如果复数 C 的两个元素按升序排列，则返回 1，否则返回 0
```

IsDescending(C)

操作结果：如果复数 C 的两个元素按降序排列，则返回 1，否则返回 0

Max(C,&e)

操作结果：用 e 返回复数 C 的两个元素中值较大的一个

Min(C,&e)

操作结果：用 e 返回复数 C 的两个元素中值较小的一个

}ADT Complex

ADT RationalNumber{

数据对象：D={s,m|s,m 为自然数，且 m 不为 0}

数据关系：R={<s,m>}

基本操作：

InitRationalNumber(&R,s,m)

操作结果：构造一个有理数 R，其分子和分母分别为 s 和 m

DestroyRationalNumber(&R)

操作结果：销毁有理数 R

Get(R,k,&e)

操作结果：用 e 返回有理数 R 的第 k 元的值

Put(&R,k,e)

操作结果：改变有理数 R 的第 k 元的值为 e

IsAscending(R)

操作结果：若有理数 R 的两个元素按升序排列，则返回 1，否则返回 0

IsDescending(R)

操作结果：若有理数 R 的两个元素按降序排列，则返回 1，否则返回 0

Max(R,&e)

操作结果：用 e 返回有理数 R 的两个元素中值较大的一个

Min(R,&e)

操作结果：用 e 返回有理数 R 的两个元素中值较小的一个

}ADT RationalNumber

1.5 试画出与下列程序段等价的框图。

(1)
```
product=1; i=1;
while(i<=n){
    product *= i;
    i++;
}
```

(2)
```
i=0;
do {
    i++;
} while((i!=n) && (a[i]!=x));
```

(3)
```
switch {
    case x<y: z=y-x; break;
    case x=y: z=abs(x*y); break;
    default: z=(x-y)/abs(x)*abs(y);
}
```

1.6 在程序设计中，常用下列三种不同的出错处理方式：

(1) 用 exit 语句终止执行并报告错误；

(2) 以函数的返回值区别正确返回或错误返回；

(3) 设置一个整型变量的函数参数以区别正确返回或某种错误返回。

试讨论这三种方法各自的优缺点。

解：(1)exit 常用于异常错误处理，它可以强行中断程序的执行，返回操作系统。

(2)以函数的返回值判断正确与否常用于子程序的测试，便于实现程序的局部控制。

(3)用整型函数进行错误处理的优点是可以给出错误类型，便于迅速确定错误。

1.7 在程序设计中，可采用下列三种方法实现输出和输入：

(1) 通过 scanf 和 printf 语句；

(2) 通过函数的参数显式传递；

(3) 通过全局变量隐式传递。

试讨论这三种方法的优缺点。

解：(1)用 scanf 和 printf 直接进行输入输出的好处是形象、直观，但缺点是需要对其进行格式控制，较为烦琐，如果出现错误，则会引起整个系统的崩溃。

(2)通过函数的参数传递进行输入输出，便于实现信息的隐蔽，减少出错的可能。

(3)通过全局变量的隐式传递进行输入输出最为方便，只需修改变量的值即可，但过多的全局变量使程序的维护较为困难。

1.8 设 n 为正整数。试确定下列各程序段中前置以记号@的语句的频度：

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

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

(3)
```
i=1; k=0;
while (i<=n-1) {
     i++;
    @ k += 10*i;
}
```

(4)
```
k=0;
for(i=1; i<=n; i++) {
    for(j=i; j<=n; j++)
        @  k++;
}
```

(5)
```
for(i=1; i<=n; i++) {
    for(j=1; j<=i; j++) {
        for(k=1; k<=j; k++)
            @  x += delta;
}
```

(6) i=1; j=0;

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

(7) x=n; y=0; // n 是不小于 1 的常数

```
while(x>=(y+1)*(y+1)) {
    @  y++;
}
```

(8) x=91; y=100;

```
while(y>0) {
    @  if(x>100) { x -= 10; y--; }
       else x++;
}
```

解：(1) n-1

(2) n-1

(3) n-1

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

(5) $1+(1+2)+(1+2+3)+\ldots+(1+2+3+\ldots+n)=\sum_{i=1}^{n}\frac{i(i+1)}{2}$

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

$$=\frac{1}{12}n(n+1)(2n+1)+\frac{1}{4}n(n+1)=\frac{1}{12}n(n+1)(2n+3)$$

(6) n

(7) $\lfloor\sqrt{n}\rfloor$ 向下取整

(8) 1100

1.9 假设 n 为 2 的乘幂，并且 n>2，试求下列算法的时间复杂度及变量 count 的值（以 n 的函数形式表示）。

```
int Time(int n) {
    count = 0;    x=2;
    while(x<n/2) {
        x *= 2;  count++;
    }
    return count;
}
```

解： $o(\log_2 n)$

count=$\log_2 n-2$

1.11 已知有实现同一功能的两个算法，其时间复杂度分别为 $O(2^n)$ 和 $O(n^{10})$，假设现实计算机可连续

运算的时间为10^7秒（100多天），又每秒可执行基本操作（根据这些操作来估算算法时间复杂度）10^5次。试问在此条件下，这两个算法可解问题的规模（即n值的范围）各为多少？哪个算法更适宜？请说明理由。

解： $2^n = 10^{12}$　　n=40

$n^{10} = 10^{12}$　　n=16

则对于同样的循环次数 n，在这个规模下，第二种算法所花费的代价要大得多。故在这个规模下，第一种算法更适宜。

1.12 设有以下三个函数：

$f(n) = 21n^4 + n^2 + 1000$，$g(n) = 15n^4 + 500n^3$，$h(n) = 500n^{3.5} + n\log n$

请判断以下断言正确与否：

(1) f(n)是O(g(n))

(2) h(n)是O(f(n))

(3) g(n)是O(h(n))

(4) h(n)是$O(n^{3.5})$

(5) h(n)是O(nlogn)

解： (1)对 (2)错 (3)错 (4)对 (5)错

1.13 试设定若干n值，比较两函数n^2和$50n\log_2 n$的增长趋势，并确定n在什么范围内，函数n^2的值大于$50n\log_2 n$的值。

解： n^2的增长趋势快。但在n较小的时候，$50n\log_2 n$的值较大。

当n>438时，$n^2 > 50n\log_2 n$

1.14 判断下列各对函数$f(n)$和$g(n)$，当$n \to \infty$时，哪个函数增长更快？

(1) $f(n) = 10n^2 + \ln(n! + 10^{n^3})$，$g(n) = 2n^4 + n + 7$

(2) $f(n) = (\ln(n!) + 5)^2$，$g(n) = 13n^{2.5}$

(3) $f(n) = n^{2.1} + \sqrt{n^4 + 1}$，$g(n) = (\ln(n!))^2 + n$

(4) $f(n) = 2^{(n^3)} + (2^n)^2$，$g(n) = n^{(n^2)} + n^5$

解： (1)g(n)快 (2)g(n)快 (3)f(n)快 (4) f(n)快

1.15 试用数学归纳法证明：

(1) $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6 \qquad (n \geq 0)$

(2) $\sum_{i=0}^{n} x^i = (x^{n+1} - 1)/(x - 1) \qquad (x \neq 1, n \geq 0)$

(3) $\sum_{i=1}^{n} 2^{i-1} = 2^n - 1 \qquad (n \geq 1)$

(4) $\sum_{i=1}^{n} (2i-1) = n^2 \qquad (n \geq 1)$

1.16 试写一算法，自大至小依次输出顺序读入的三个整数 X，Y 和 Z 的值

解：

```
int max3(int x,int y,int z)
{
    if(x>y)
        if(x>z) return x;
        else return z;
    else
        if(y>z) return y;
        else return z;
}
```

1.17 已知 k 阶斐波那契序列的定义为

$$f_0 = 0，\ f_1 = 0，\ ...，\ f_{k-2} = 0，\ f_{k-1} = 1；$$

$$f_n = f_{n-1} + f_{n-2} + \cdots + f_{n-k}，\ n = k, k+1, \cdots$$

试编写求 k 阶斐波那契序列的第 m 项值的函数算法，k 和 m 均以值调用的形式在函数参数表中出现。

解：k>0 为阶数，n 为数列的第 n 项

```
int Fibonacci(int k,int n)
{
    if(k<1) exit(OVERFLOW);
    int *p,x;
    p=new int[k+1];
    if(!p) exit(OVERFLOW);
    int i,j;
    for(i=0;i<k+1;i++){
            if(i<k-1) p[i]=0;
            else p[i]=1;
        }
        for(i=k+1;i<n+1;i++){
            x=p[0];
            for(j=0;j<k;j++) p[j]=p[j+1];
            p[k]=2*p[k-1]-x;
        }
        return p[k];
}
```

1.18 假设有 A，B，C，D，E 五个高等院校进行田径对抗赛，各院校的单项成绩均已存入计算机，并构成一张表，表中每一行的形式为

项目名称	性别	校名	成绩	得分

编写算法，处理上述表格，以统计各院校的男、女总分和团体总分，并输出。

解：

```
typedef enum{A,B,C,D,E} SchoolName;
typedef enum{Female,Male} SexType;
typedef struct{
     char event[3];  //项目
     SexType sex;
     SchoolName school;
     int score;
} Component;
typedef struct{
     int MaleSum;       //男团总分
     int FemaleSum;//女团总分
     int TotalSum; //团体总分
} Sum;
Sum SumScore(SchoolName sn,Component a[],int n)
{
     Sum temp;
     temp.MaleSum=0;
     temp.FemaleSum=0;
     temp.TotalSum=0;
     int i;
     for(i=0;i<n;i++){
          if(a[i].school==sn){
               if(a[i].sex==Male) temp.MaleSum+=a[i].score;
               if(a[i].sex==Female) temp.FemaleSum+=a[i].score;
          }
     }
     temp.TotalSum=temp.MaleSum+temp.FemaleSum;
     return temp;
}
```

1.19 试编写算法，计算$i!*2^i$的值并存入数组 a[0..arrsize-1]的第 i-1 个分量中(i=1,2,...,n)。假设计算机中允许的整数最大值为 maxint，则当 n>arrsize 或对某个$k(1 \le k \le n)$，使$k!\bullet 2^k > \max\text{int}$时，应按出错处理。注意选择你认为较好的出错处理方法。

解：

```
#include<iostream.h>
#include<stdlib.h>
#define MAXINT 65535
#define ArrSize 100
int fun(int i);

int main()
```

```
{
    int i,k;
    int a[ArrSize];
    cout<<"Enter k:";
    cin>>k;
    if(k>ArrSize-1) exit(0);
    for(i=0;i<=k;i++){
        if(i==0) a[i]=1;
        else{
            if(2*i*a[i-1]>MAXINT) exit(0);
            else a[i]=2*i*a[i-1];
        }
    }
    for(i=0;i<=k;i++){
        if(a[i]>MAXINT) exit(0);
        else cout<<a[i]<<" ";
    }
    return 0;
}
```

1.20 试编写算法求一元多项式的值 $P_n(x)=\sum_{i=0}^{n}a_i x^i$ 的值 $P_n(x_0)$，并确定算法中每一语句的执行次数和整个算法的时间复杂度。注意选择你认为较好的输入和输出方法。本题的输入为 $a_i(i=0,1,\cdots,n)$，x_0 和 n，输出为 $P_n(x_0)$。

解：

```
#include<iostream.h>
#include<stdlib.h>
#define N  10
double polynomail(int a[],int i,double x,int n);
int main()
{
    double x;
    int n,i;
    int a[N];
    cout<<"输入变量的值 x:";
    cin>>x;
    cout<<"输入多项式的阶次 n:";
    cin>>n;
    if(n>N-1) exit(0);
    cout<<"输入多项式的系数 a[0]--a[n]:";
    for(i=0;i<=n;i++) cin>>a[i];
    cout<<"The polynomail value is "<<polynomail(a,n,x,n)<<endl;
```

```
    return 0;
}
double polynomail(int a[],int i,double x,int n)
{
    if(i>0) return a[n-i]+polynomail(a,i-1,x,n)*x;
    else return a[n];
}
```

本算法的时间复杂度为o(n)。

第2章　线性表

2.1 描述以下三个概念的区别：头指针，头结点，首元结点（第一个元素结点）。

解：头指针是指向链表中第一个结点的指针。首元结点是指链表中存储第一个数据元素的结点。头结点是在首元结点之前附设的一个结点，该结点不存储数据元素，其指针域指向首元结点，其作用主要是为了方便对链表的操作。它可以对空表、非空表以及首元结点的操作进行统一处理。

2.2 填空题。

解：(1) 在顺序表中插入或删除一个元素，需要平均移动***表中一半***元素，具体移动的元素个数与***元素在表中的位置***有关。

(2) 顺序表中逻辑上相邻的元素的物理位置***必定***紧邻。单链表中逻辑上相邻的元素的物理位置***不一定***紧邻。

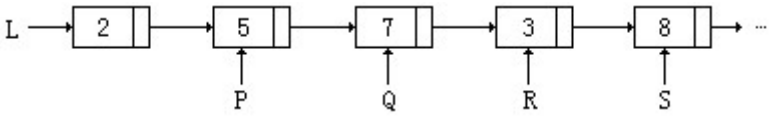
(3) 在单链表中，除了首元结点外，任一结点的存储位置由***其前驱结点的链域的值***指示。

(4) 在单链表中设置头结点的作用是***插入和删除首元结点时不用进行特殊处理***。

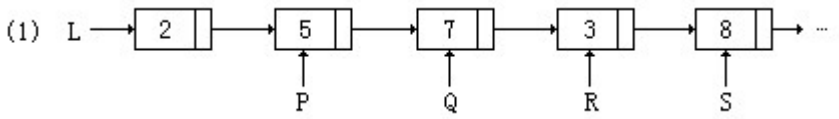
2.3 在什么情况下用顺序表比链表好？

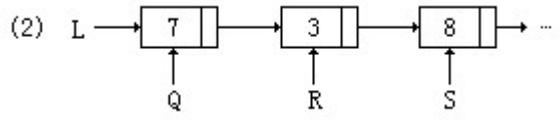
解：当线性表的数据元素在物理位置上是连续存储的时候，用顺序表比用链表好，其特点是可以进行随机存取。

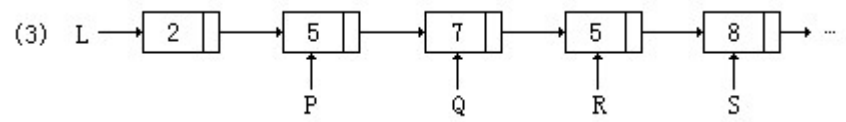
2.4 对以下单链表分别执行下列各程序段，并画出结果示意图。

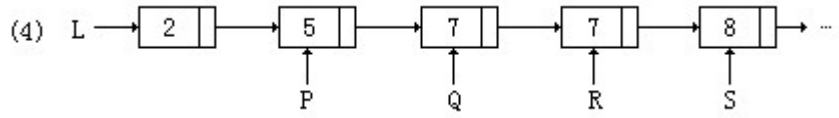


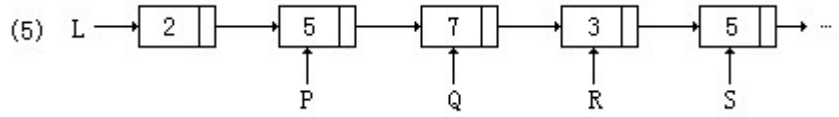
解：

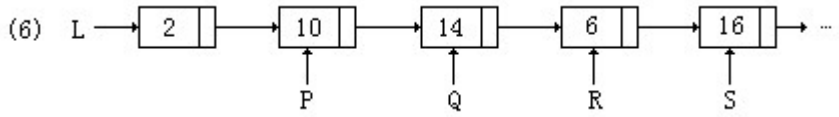


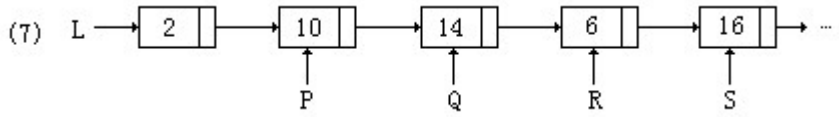








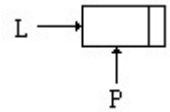


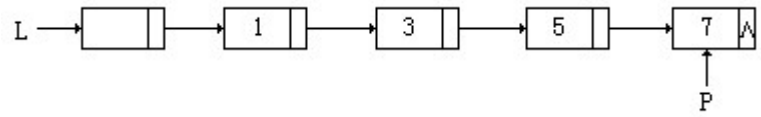


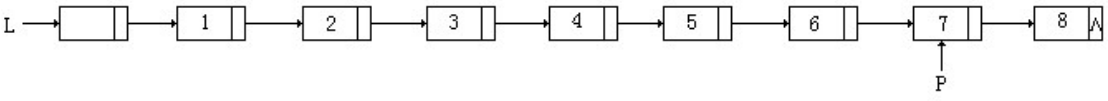
2.5 画出执行下列各行语句后各指针及链表的示意图。

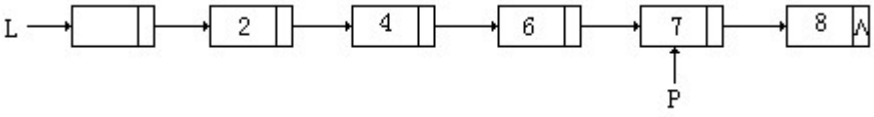
```
L=(LinkList)malloc(sizeof(LNode));    P=L;
for(i=1;i<=4;i++){
    P->next=(LinkList)malloc(sizeof(LNode));
    P=P->next;    P->data=i*2-1;
}
P->next=NULL;
for(i=4;i>=1;i--) Ins_LinkList(L,i+1,i*2);
for(i=1;i<=3;i++) Del_LinkList(L,i);
```

解：









2.6 已知 L 是无表头结点的单链表，且 P 结点既不是首元结点，也不是尾元结点，试从下列提供的答案中选择合适的语句序列。

a. 在 P 结点后插入 S 结点的语句序列是_____。

b. 在 P 结点前插入 S 结点的语句序列是_____。

c. 在表首插入 S 结点的语句序列是_____。

d. 在表尾插入 S 结点的语句序列是_____。

(1) P->next=S;

(2) P->next=P->next->next;

(3) P->next=S->next;

(4) S->next=P->next;

(5) S->next=L;

(6) S->next=NULL;

(7) Q=P;

(8) while(P->next!=Q) P=P->next;

(9) while(P->next!=NULL) P=P->next;

(10) P=Q;

(11) P=L;

(12) L=S;

(13) L=P;

解：a. (4) (1)

b. (7) (11) (8) (4) (1)

c. (5) (12)

d. (9) (1) (6)

2.7 已知 L 是带表头结点的非空单链表，且 P 结点既不是首元结点，也不是尾元结点，试从下列提供的答案中选择合适的语句序列。

a. 删除 P 结点的直接后继结点的语句序列是_____。

b. 删除 P 结点的直接前驱结点的语句序列是_____。

c. 删除 P 结点的语句序列是_____。

d. 删除首元结点的语句序列是_____。

e. 删除尾元结点的语句序列是_____。

(1) P=P->next;

(2) P->next=P;

(3) P->next=P->next->next;

(4) P=P->next->next;

(5) while(P!=NULL) P=P->next;

(6) while(Q->next!=NULL) { P=Q; Q=Q->next; }

(7) while(P->next!=Q) P=P->next;

(8) while(P->next->next!=Q) P=P->next;

(9) while(P->next->next!=NULL) P=P->next;

(10) Q=P;

(11) Q=P->next;

(12) P=L;

(13) L=L->next;

(14) free(Q);

解：a. (11) (3) (14)

b. (10) (12) (8) (3) (14)

c. (10) (12) (7) (3) (14)

d. (12) (11) (3) (14)

e. (9) (11) (3) (14)

2.8 已知P结点是某双向链表的中间结点，试从下列提供的答案中选择合适的语句序列。

a. 在P结点后插入S结点的语句序列是_____。

b. 在P结点前插入S结点的语句序列是_____。

c. 删除P结点的直接后继结点的语句序列是_____。

d. 删除P结点的直接前驱结点的语句序列是_____。

e. 删除P结点的语句序列是_____。

(1) P->next=P->next->next;

(2) P->priou=P->priou->priou;

(3) P->next=S;

(4) P->priou=S;

(5) S->next=P;

(6) S->priou=P;

(7) S->next=P->next;

(8) S->priou=P->priou;

(9) P->priou->next=P->next;

(10) P->priou->next=P;

(11) P->next->priou=P;

(12) P->next->priou=S;

(13) P->priou->next=S;

(14) P->next->priou=P->priou;

(15) Q=P->next;

(16) Q=P->priou;

(17) free(P);

(18) free(Q);

解：a. (7) (3) (6) (12)

b. (8) (4) (5) (13)

c. (15) (1) (11) (18)

d. (16) (2) (10) (18)

e. (14) (9) (17)

2.9 简述以下算法的功能。

(1)
```
Status A(LinkedList L) { //L是无表头结点的单链表
    if(L && L->next) {
        Q=L;      L=L->next;     P=L;
        while(P->next) P=P->next;
        P->next=Q;     Q->next=NULL;
    }
    return OK;
}
```

(2)
```
void BB(LNode *s, LNode *q) {
    p=s;
    while(p->next!=q) p=p->next;
    p->next =s;
```

```
}
void AA(LNode *pa, LNode *pb) {
    //pa 和 pb 分别指向单循环链表中的两个结点
    BB(pa,pb);
    BB(pb,pa);
}
```

解：(1) 如果 L 的长度不小于 2，将 L 的首元结点变成尾元结点。

(2) 将单循环链表拆成两个单循环链表。

2.10 指出以下算法中的错误和低效之处，并将它改写为一个既正确又高效的算法。

```
Status DeleteK(SqList &a,int i,int k)
{
    //本过程从顺序存储结构的线性表 a 中删除第 i 个元素起的 k 个元素
    if(i<1||k<0||i+k>a.length) return INFEASIBLE;//参数不合法
    else {
        for(count=1;count<k;count++){
            //删除第一个元素
            for(j=a.length;j>=i+1;j--) a.elem[j-i]=a.elem[j];
            a.length--;
        }
    return OK;
}
```

解：

```
Status DeleteK(SqList &a,int i,int k)
{
    //从顺序存储结构的线性表 a 中删除第 i 个元素起的 k 个元素
    //注意 i 的编号从 0 开始
    int j;
    if(i<0||i>a.length-1||k<0||k>a.length-i) return INFEASIBLE;
    for(j=0;j<=k;j++)
        a.elem[j+i]=a.elem[j+i+k];
    a.length=a.length-k;
    return OK;
}
```

2.11 设顺序表 va 中的数据元素递增有序。试写一算法，将 x 插入到顺序表的适当位置上，以保持该表的有序性。

解：

```
Status InsertOrderList(SqList &va,ElemType x)
{
    //在非递减的顺序表 va 中插入元素 x 并使其仍成为顺序表的算法
    int i;
    if(va.length==va.listsize)return(OVERFLOW);
    for(i=va.length;i>0,x<va.elem[i-1];i--)
        va.elem[i]=va.elem[i-1];
    va.elem[i]=x;
```

```
    va.length++;
    return OK;
}
```

2.12 设 $A=(a_1,\cdots,a_m)$和 $B=(b_1,\cdots,b_n)$均为顺序表， A' 和 B' 分别为 A 和 B 中除去最大共同前缀后的子表。若 $A'=B'=$空表，则 $A=B$；若 A'=空表，而 $B'\neq$ 空表，或者两者均不为空表，且 A' 的首元小于 B' 的首元，则 $A<B$；否则 $A>B$。试写一个比较 A， B 大小的算法。

解：

```
Status CompareOrderList(SqList &A,SqList &B)
{
    int i,k,j;
    k=A.length>B.length?A.length:B.length;
    for(i=0;i<k;i++){
        if(A.elem[i]>B.elem[i]) j=1;
        if(A.elem[i]<B.elem[i]) j=-1;
    }
    if(A.length>k) j=1;
    if(B.length>k) j=-1;
    if(A.length==B.length) j=0;
    return j;
}
```

2.13 试写一算法在带头结点的单链表结构上实现线性表操作 Locate(L,x)；

解：

```
int LocateElem_L(LinkList &L,ElemType x)
{
    int i=0;
    LinkList p=L;
    while(p&&p->data!=x){
        p=p->next;
        i++;
    }
    if(!p) return 0;
    else return i;
}
```

2.14 试写一算法在带头结点的单链表结构上实现线性表操作 Length(L)。

解：

```
//返回单链表的长度
int ListLength_L(LinkList &L)
{
    int i=0;
    LinkList p=L;
    if(p) p=p-next;
    while(p){
        p=p->next;
```

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

2.15 已知指针 ha 和 hb 分别指向两个单链表的头结点，并且已知两个链表的长度分别为 m 和 n。试写一算法将这两个链表连接在一起，假设指针 hc 指向连接后的链表的头结点，并要求算法以尽可能短的时间完成连接运算。请分析你的算法的时间复杂度。

解：

```
void MergeList_L(LinkList &ha,LinkList &hb,LinkList &hc)
{
    LinkList pa,pb;
    pa=ha;
    pb=hb;
    while(pa->next&&pb->next){
        pa=pa->next;
        pb=pb->next;
    }
    if(!pa->next){
        hc=hb;
        while(pb->next) pb=pb->next;
        pb->next=ha->next;
    }
    else{
        hc=ha;
        while(pa->next) pa=pa->next;
        pa->next=hb->next;
    }
}
```

2.16 已知指针 la 和 lb 分别指向两个无头结点单链表中的首元结点。下列算法是从表 la 中删除自第 i 个元素起共 len 个元素后，将它们插入到表 lb 中第 i 个元素之前。试问此算法是否正确？若有错，请改正之。

```
Status DeleteAndInsertSub(LinkedList la,LinkedList lb,int i,int j,int len)
{
    if(i<0||j<0||len<0) return INFEASIBLE;
    p=la;    k=1;
    while(k<i){   p=p->next;    k++;}
    q=p;
    while(k<=len){q=q->next;    k++;}
    s=lb; k=1;
    while(k<j){   s=s->next;    k++;}
    s->next=p;    q->next=s->next;
    return OK;
}
```

解：

```
Status DeleteAndInsertSub(LinkList &la,LinkList &lb,int i,int j,int len)
```

```
{
    LinkList p,q,s,prev=NULL;
    int k=1;
    if(i<0||j<0||len<0) return INFEASIBLE;
    // 在 la 表中查找第 i 个结点
    p=la;
    while(p&&k<i){
        prev=p;
        p=p->next;
        k++;
    }
    if(!p)return INFEASIBLE;
    // 在 la 表中查找第 i+len-1 个结点
    q=p; k=1;
    while(q&&k<len){
        q=p->next;
        k++;
    }
    if(!q)return INFEASIBLE;
    // 完成删除，注意，i=1 的情况需要特殊处理
    if(!prev) la=q->next;
    else prev->next=q->next;
    // 将从 la 中删除的结点插入到 lb 中
    if(j=1){
        q->next=lb;
        lb=p;
    }
    else{
        s=lb;    k=1;
        while(s&&k<j-1){
            s=s->next;
            k++;
        }
        if(!s)return INFEASIBLE;
        q->next=s->next;
        s->next=p; //完成插入
    }
    return OK;
}
```

2.17 试写一算法，在无头结点的动态单链表上实现线性表操作 Insert(L,i,b)，并和在带头结点的动态单链表上实现相同操作的算法进行比较。

2.18 试写一算法，实现线性表操作 Delete(L,i)，并和在带头结点的动态单链表上实现相同操作的算法进行比较。

2.19 已知线性表中的元素以值递增有序排列，并以单链表作存储结构。试写一高效的算法，删除表中所有

值大于 mink 且小于 maxk 的元素（若表中存在这样的元素），同时释放被删结点空间，并分析你的算法的时间复杂度（注意，mink 和 maxk 是给定的两个参变量，它们的值可以和表中的元素相同，也可以不同）。

解：

```
Status ListDelete_L(LinkList &L,ElemType mink,ElemType maxk)
{
    LinkList p,q,prev=NULL;
    if(mink>maxk)return ERROR;
    p=L;
    prev=p;
    p=p->next;
    while(p&&p->data<maxk){
        if(p->data<=mink){
            prev=p;
            p=p->next;
        }
        else{
            prev->next=p->next;
            q=p;
            p=p->next;
            free(q);
        }
    }
    return OK;
}
```

2.20 同 2.19 题条件，试写一高效的算法，删除表中所有值相同的多余元素（使得操作后的线性表中所有元素的值均不相同），同时释放被删结点空间，并分析你的算法的时间复杂度。

解：

```
void ListDelete_LSameNode(LinkList &L)
{
    LinkList p,q,prev;
    p=L;
    prev=p;
    p=p->next;
    while(p){
        prev=p;
        p=p->next;
        if(p&&p->data==prev->data){
            prev->next=p->next;
            q=p;
            p=p->next;
            free(q);
        }
    }
}
```

2.21 试写一算法，实现顺序表的就地逆置，即利用原表的存储空间将线性表$(a_1,\cdots,a_n)$逆置为$(a_n,\cdots,a_1)$。

解：

```
// 顺序表的逆置
Status ListOppose_Sq(SqList &L)
{
    int i;
    ElemType x;
    for(i=0;i<L.length/2;i++){
        x=L.elem[i];
        L.elem[i]=L.elem[L.length-1-i];
        L.elem[L.length-1-i]=x;
    }
    return OK;
}
```

2.22 试写一算法，对单链表实现就地逆置。

解：

```
// 带头结点的单链表的逆置
Status ListOppose_L(LinkList &L)
{
    LinkList p,q;
    p=L;
    p=p->next;
    L->next=NULL;
    while(p){
        q=p;
        p=p->next;
        q->next=L->next;
        L->next=q;
    }
    return OK;
}
```

2.23 设线性表$A=(a_1,a_2,\cdots,a_m)$，$B=(b_1,b_2,\cdots,b_n)$，试写一个按下列规则合并 A，B 为线性表 C 的算法，即使得

$$C=(a_1,b_1,\cdots,a_m,b_m,b_{m+1},\cdots,b_n) \quad \text{当} m\le n \text{时；}$$

$$C=(a_1,b_1,\cdots,a_n,b_n,a_{n+1},\cdots,a_m) \quad \text{当} m>n \text{时。}$$

线性表 A，B 和 C 均以单链表作存储结构，且 C 表利用 A 表和 B 表中的结点空间构成。注意：单链表的长度值 m 和 n 均未显式存储。

解：

```
// 将合并后的结果放在C表中，并删除B表
Status ListMerge_L(LinkList &A,LinkList &B,LinkList &C)
{
     LinkList pa,pb,qa,qb;
     pa=A->next;
     pb=B->next;
     C=A;
     while(pa&&pb){
          qa=pa;              qb=pb;
          pa=pa->next;  pb=pb->next;
          qb->next=qa->next;
          qa->next=qb;
     }
     if(!pa)qb->next=pb;
     pb=B;
     free(pb);
     return OK;
}
```

2.24 假设有两个按元素值递增有序排列的线性表A和B，均以单链表作存储结构，请编写算法将A表和B表归并成一个按元素值递减有序（即非递增有序，允许表中含有值相同的元素）排列的线性表 C，并要求利用原表（即A表和B表）的结点空间构造C表。

解：

```
// 将合并逆置后的结果放在C表中，并删除B表
Status ListMergeOppose_L(LinkList &A,LinkList &B,LinkList &C)
{
     LinkList pa,pb,qa,qb;
     pa=A;
     pb=B;
     qa=pa;    // 保存pa的前驱指针
     qb=pb;    // 保存pb的前驱指针
     pa=pa->next;
     pb=pb->next;
     A->next=NULL;
     C=A;
     while(pa&&pb){
          if(pa->data<pb->data){
               qa=pa;
               pa=pa->next;
               qa->next=A->next;  //将当前最小结点插入A表表头
               A->next=qa;
          }
          else{
               qb=pb;
               pb=pb->next;
```

```
                qb->next=A->next;  //将当前最小结点插入 A 表表头
                A->next=qb;
            }
        }
        while(pa){
            qa=pa;
            pa=pa->next;
            qa->next=A->next;
            A->next=qa;
        }
        while(pb){
            qb=pb;
            pb=pb->next;
            qb->next=A->next;
            A->next=qb;
        }
        pb=B;
        free(pb);
        return OK;
    }
```

2.25 假设以两个元素依值递增有序排列的线性表 A 和 B 分别表示两个集合（即同一表中的元素值各不相同），现要求另辟空间构成一个线性表 C，其元素为 A 和 B 中元素的交集，且表 C 中的元素有依值递增有序排列。试对顺序表编写求 C 的算法。

解：

```
// 将 A、B 求交后的结果放在 C 表中
Status ListCross_Sq(SqList &A,SqList &B,SqList &C)
{
    int i=0,j=0,k=0;
    while(i<A.length && j<B.length){
        if(A.elem[i]<B.elem[j]) i++;
        else
            if(A.elem[i]>B.elem[j]) j++;
            else{
                ListInsert_Sq(C,k,A.elem[i]);
                i++;
                k++;
            }
    }
    return OK;
}
```

2.26 要求同 2.25 题。试对单链表编写求 C 的算法。

解：

```
// 将 A、B 求交后的结果放在 C 表中，并删除 B 表
Status ListCross_L(LinkList &A,LinkList &B,LinkList &C)
```

```
{
    LinkList pa,pb,qa,qb,pt;
    pa=A;
    pb=B;
    qa=pa;   // 保存 pa 的前驱指针
    qb=pb;   // 保存 pb 的前驱指针
    pa=pa->next;
    pb=pb->next;
    C=A;
    while(pa&&pb){
        if(pa->data<pb->data){
            pt=pa;
            pa=pa->next;
            qa->next=pa;
            free(pt);
        }
        else
            if(pa->data>pb->data){
                pt=pb;
                pb=pb->next;
                qb->next=pb;
                free(pt);
            }
            else{
                qa=pa;
                pa=pa->next;
            }
    }
    while(pa){
        pt=pa;
        pa=pa->next;
        qa->next=pa;
        free(pt);
    }
    while(pb){
        pt=pb;
        pb=pb->next;
        qb->next=pb;
        free(pt);
    }
    pb=B;
    free(pb);
    return OK;
}
```

2.27 对 2.25 题的条件作以下两点修改，对顺序表重新编写求得表 C 的算法。

(1) 假设在同一表（A 或 B）中可能存在值相同的元素，但要求新生成的表 C 中的元素值各不相同；

(2) 利用 A 表空间存放表 C。

解：

(1)

```
// A、B 求交，然后删除相同元素，将结果放在 C 表中
Status ListCrossDelSame_Sq(SqList &A,SqList &B,SqList &C)
{
     int i=0,j=0,k=0;
     while(i<A.length && j<B.length){
          if(A.elem[i]<B.elem[j]) i++;
          else
               if(A.elem[i]>B.elem[j]) j++;
               else{
                    if(C.length==0){
                         ListInsert_Sq(C,k,A.elem[i]);
                         k++;
                    }
                    else
                         if(C.elem[C.length-1]!=A.elem[i]){
                              ListInsert_Sq(C,k,A.elem[i]);
                              k++;
                         }
                    i++;
               }
     }
     return OK;
}
```

(2)

```
// A、B 求交，然后删除相同元素，将结果放在 A 表中
Status ListCrossDelSame_Sq(SqList &A,SqList &B)
{
     int i=0,j=0,k=0;
     while(i<A.length && j<B.length){
          if(A.elem[i]<B.elem[j]) i++;
          else
               if(A.elem[i]>B.elem[j]) j++;
               else{
                    if(k==0){
                         A.elem[k]=A.elem[i];
                         k++;
                    }
                    else
                         if(A.elem[k]!=A.elem[i]){
```

```
                    A.elem[k]=A.elem[i];
                    k++;
                }
            i++;
        }
    }
    A.length=k;
    return OK;
}
```

2.28 对 2.25 题的条件作以下两点修改，对单链表重新编写求得表 C 的算法。

(1) 假设在同一表（A 或 B）中可能存在值相同的元素，但要求新生成的表 C 中的元素值各不相同；

(2) 利用原表（A 表或 B 表）中的结点构成表 C，并释放 A 表中的无用结点空间。

解：

(1)

```
// A、B 求交，结果放在 C 表中，并删除相同元素
Status ListCrossDelSame_L(LinkList &A,LinkList &B,LinkList &C)
{
    LinkList pa,pb,qa,qb,pt;
    pa=A;
    pb=B;
    qa=pa;   // 保存 pa 的前驱指针
    qb=pb;   // 保存 pb 的前驱指针
    pa=pa->next;
    pb=pb->next;
    C=A;
    while(pa&&pb){
        if(pa->data<pb->data){
            pt=pa;
            pa=pa->next;
            qa->next=pa;
            free(pt);
        }
        else
            if(pa->data>pb->data){
                pt=pb;
                pb=pb->next;
                qb->next=pb;
                free(pt);
            }
            else{
                if(pa->data==qa->data){
                    pt=pa;
                    pa=pa->next;
                    qa->next=pa;
```

```
                    free(pt);
                }
                else{
                    qa=pa;
                    pa=pa->next;
                }
            }
    }
    while(pa){
        pt=pa;
        pa=pa->next;
        qa->next=pa;
        free(pt);
    }
    while(pb){
        pt=pb;
        pb=pb->next;
        qb->next=pb;
        free(pt);
    }
    pb=B;
    free(pb);
    return OK;
}
```

(2)

```
// A、B 求交，结果放在 A 表中，并删除相同元素
Status ListCrossDelSame_L(LinkList &A,LinkList &B)
{
    LinkList pa,pb,qa,qb,pt;
    pa=A;
    pb=B;
    qa=pa;   // 保存 pa 的前驱指针
    qb=pb;   // 保存 pb 的前驱指针
    pa=pa->next;
    pb=pb->next;
    while(pa&&pb){
        if(pa->data<pb->data){
            pt=pa;
            pa=pa->next;
            qa->next=pa;
            free(pt);
        }
        else
            if(pa->data>pb->data){
```

```
                pt=pb;
                pb=pb->next;
                qb->next=pb;
                free(pt);
            }
            else{
                if(pa->data==qa->data){
                    pt=pa;
                    pa=pa->next;
                    qa->next=pa;
                    free(pt);
                }
                else{
                    qa=pa;
                    pa=pa->next;
                }
            }
        }
        while(pa){
            pt=pa;
            pa=pa->next;
            qa->next=pa;
            free(pt);
        }
        while(pb){
            pt=pb;
            pb=pb->next;
            qb->next=pb;
            free(pt);
        }
        pb=B;
        free(pb);
        return OK;
    }
```

2.29 已知 A，B 和 C 为三个递增有序的线性表，现要求对 A 表作如下操作：删去那些既在 B 表中出现又在 C 表中出现的元素。试对顺序表编写实现上述操作的算法，并分析你的算法的时间复杂度（注意：题中没有特别指明同一表中的元素值各不相同）。

解：

```
// 在 A 中删除既在 B 中出现又在 C 中出现的元素，结果放在 D 中
Status ListUnion_Sq(SqList &D,SqList &A,SqList &B,SqList &C)
{
    SqList Temp;
    InitList_Sq(Temp);
    ListCross_L(B,C,Temp);
```

```
    ListMinus_L(A,Temp,D);
}
```

2.30 要求同 2.29 题。试对单链表编写算法，请释放 A 表中的无用结点空间。

解：

```
// 在 A 中删除既在 B 中出现又在 C 中出现的元素，并释放 B、C
Status ListUnion_L(LinkList &A,LinkList &B,LinkList &C)
{
    ListCross_L(B,C);
    ListMinus_L(A,B);
}
// 求集合 A-B，结果放在 A 表中，并删除 B 表
Status ListMinus_L(LinkList &A,LinkList &B)
{
    LinkList pa,pb,qa,qb,pt;
    pa=A;
    pb=B;
    qa=pa;    // 保存 pa 的前驱指针
    qb=pb;    // 保存 pb 的前驱指针
    pa=pa->next;
    pb=pb->next;
    while(pa&&pb){
        if(pb->data<pa->data){
            pt=pb;
            pb=pb->next;
            qb->next=pb;
            free(pt);
        }
        else
            if(pb->data>pa->data){
                qa=pa;
                pa=pa->next;
            }
            else{
                pt=pa;
                pa=pa->next;
                qa->next=pa;
                free(pt);
            }
    }
    while(pb){
        pt=pb;
        pb=pb->next;
        qb->next=pb;
        free(pt);
```

```
    }
    pb=B;
    free(pb);
    return OK;
}
```

2.31 假设某个单向循环链表的长度大于 1，且表中既无头结点也无头指针。已知 s 为指向链表中某个结点的指针，试编写算法在链表中删除指针 s 所指结点的前驱结点。

解：

```
// 在单循环链表 S 中删除 S 的前驱结点
Status ListDelete_CL(LinkList &S)
{
    LinkList p,q;
    if(S==S->next)return ERROR;
    q=S;
    p=S->next;
    while(p->next!=S){
        q=p;
        p=p->next;
    }
    q->next=p->next;
    free(p);
    return OK;
}
```

2.32 已知有一个单向循环链表，其每个结点中含三个域：pre，data 和 next，其中 data 为数据域，next 为指向后继结点的指针域，pre 也为指针域，但它的值为空，试编写算法将此单向循环链表改为双向循环链表，即使 pre 成为指向前驱结点的指针域。

解：

```
// 建立一个空的循环链表
Status InitList_DL(DuLinkList &L)
{
    L=(DuLinkList)malloc(sizeof(DuLNode));
    if(!L) exit(OVERFLOW);
    L->pre=NULL;
    L->next=L;
    return OK;
}
// 向循环链表中插入一个结点
Status ListInsert_DL(DuLinkList &L,ElemType e)
{
    DuLinkList p;
    p=(DuLinkList)malloc(sizeof(DuLNode));
    if(!p) return ERROR;
    p->data=e;
    p->next=L->next;
```

```
    L->next=p;
    return OK;
}
// 将单循环链表改成双向链表
Status ListCirToDu(DuLinkList &L)
{
    DuLinkList p,q;
    q=L;
    p=L->next;
    while(p!=L){
        p->pre=q;
        q=p;
        p=p->next;
    }
    if(p==L) p->pre=q;
    return OK;
}
```

2.33 已知由一个线性链表表示的线性表中含有三类字符的数据元素（如：字母字符、数字字符和其他字符），试编写算法将该线性表分割为三个循环链表，其中每个循环链表表示的线性表中均只含一类字符。

解：

```
// 将单链表 L 划分成 3 个单循环链表
Status ListDivideInto3CL(LinkList &L,LinkList &s1,LinkList &s2,LinkList &s3)
{
    LinkList p,q,pt1,pt2,pt3;
    p=L->next;
    pt1=s1;
    pt2=s2;
    pt3=s3;
    while(p){
        if(p->data>='0' && p->data<='9'){
            q=p;
            p=p->next;
            q->next=pt1->next;
            pt1->next=q;
            pt1=pt1->next;
        }
        else
            if((p->data>='A' && p->data<='Z') ||
                    (p->data>='a' && p->data<='z')){
                q=p;
                p=p->next;
                q->next=pt2->next;
                pt2->next=q;
                pt2=pt2->next;
```

```
            }
            else{
                q=p;
                p=p->next;
                q->next=pt3->next;
                pt3->next=q;
                pt3=pt3->next;
            }
        }
        q=L;
        free(q);
        return OK;
    }
```

在 2.34 至 2.36 题中，“异或指针双向链表”类型 XorLinkedList 和指针异或函数 XorP 定义为：

```
typedef  struct   XorNode {
    char data;
    struct   XorNode *LRPtr;
} XorNode, *XorPointer;
typede   struct {      //无头结点的异或指针双向链表
    XorPointer   Left, Right;       //分别指向链表的左侧和右端
} XorLinkedList;
XorPointer XorP(XorPointer p, XorPointer q);
// 指针异或函数 XorP 返回指针 p 和 q 的异或值
```

2.34 假设在算法描述语言中引入指针的二元运算“异或”，若 a 和 b 为指针，则 a⊕b 的运算结果仍为原指针类型，且

a⊕(a⊕b)=(a⊕a)⊕b=b

(a⊕b)⊕b=a⊕(b⊕b)=a

则可利用一个指针域来实现双向链表 L。链表 L 中的每个结点只含两个域：data 域和 LRPtr 域，其中 LRPtr 域存放该结点的左邻与右邻结点指针（不存在时为 NULL）的异或。若设指针 L.Left 指向链表中的最左结点，L.Right 指向链表中的最右结点，则可实现从左向右或从右向左遍历此双向链表的操作。试写一算法按任一方向依次输出链表中各元素的值。

解：

```
Status TraversingLinkList(XorLinkedList &L,char d)
{
    XorPointer p,left,right;
    if(d=='l'||d=='L'){
        p=L.Left;
        left=NULL;
        while(p!=NULL){
            VisitingData(p->data);
            left=p;
            p=XorP(left,p->LRPtr);
        }
    }
```

```
    else
        if(d=='r'||d=='R'){
            p=L.Right;
            right=NULL;
            while(p!=NULL){
                VisitingData(p->data);
                right=p;
                p=XorP(p->LRPtr,right);
            }
        }
        else return ERROR;
    return OK;
}
```

2.35 采用 2.34 题所述的存储结构，写出在第 i 个结点之前插入一个结点的算法。

2.36 采用 2.34 题所述的存储结构，写出删除第 i 个结点的算法。

2.37 设以带头结点的双向循环链表表示的线性表 $L=(a_1,a_2,\cdots,a_n)$。试写一时间复杂度 O(n)的算法，将 L 改造为 $L=(a_1,a_3,\cdots,a_n,\cdots,a_4,a_2)$。

解：

```
// 将双向链表 L=(a1,a2,...,an)改造为(a1,a3,...,an,...,a2)
Status ListChange_DuL(DuLinkList &L)
{
    int i;
    DuLinkList p,q,r;
    p=L->next;
    r=L->pre;
    i=1;
    while(p!=r){
        if(i%2==0){
            q=p;
            p=p->next;
            // 删除结点
            q->pre->next=q->next;
            q->next->pre=q->pre;
            // 插入到头结点的左面
            q->pre=r->next->pre;
            r->next->pre=q;
            q->next=r->next;
            r->next=q;
        }
        else p=p->next;
        i++;
    }
```

```
    return OK;
}
```

2.38 设有一个双向循环链表，每个结点中除有 pre，data 和 next 三个域外，还增设了一个访问频度域 freq。在链表被起用之前，频度域 freq 的值均初始化为零，而每当对链表进行一次 Locate(L,x)的操作后，被访问的结点（即元素值等于 x 的结点）中的频度域 freq 的值便增 1，同时调整链表中结点之间的次序，使其按访问频度非递增的次序顺序排列，以便始终保持被频繁访问的结点总是靠近表头结点。试编写符合上述要求的 Locate 操作的算法。

解：

```
DuLinkList ListLocate_DuL(DuLinkList &L,ElemType e)
{
    DuLinkList p,q;
    p=L->next;
    while(p!=L && p->data!=e)   p=p->next;
    if(p==L) return NULL;
    else{
        p->freq++;
        // 删除结点
        p->pre->next=p->next;
        p->next->pre=p->pre;
        // 插入到合适的位置
        q=L->next;
        while(q!=L && q->freq>p->freq) q=q->next;
        if(q==L){
            p->next=q->next;
            q->next=p;
            p->pre=q->pre;
            q->pre=p;
        }
        else{
            // 在 q 之前插入
            p->next=q->pre->next;
            q->pre->next=p;
            p->pre=q->pre;
            q->pre=p;
        }
        return p;
    }
}
```

在 2.39 至 2.40 题中，稀疏多项式采用的顺序存储结构 SqPoly 定义为

```
typedef  struct {
    int  coef;
    int exp;
} PolyTerm;
typedef struct {        //多项式的顺序存储结构
```

```
    PolyTerm  *data;
    int last;
} SqPoly;
```

2.39 已知稀疏多项式 $P_n(x)=c_1x^{e_1}+c_2x^{e_2}+\cdots+c_mx^{e_m}$，其中 $n=e_m>e_{m-1}>\cdots>e_1\geq 0$，$c_i\neq 0(i=1,2,\cdots,m)$，$m\geq 1$。试采用存储量同多项式项数 m 成正比的顺序存储结构，编写求 $P_n(x_0)$ 的算法（x_0 为给定值），并分析你的算法的时间复杂度。

解：

```
typedef struct{
    int coef;
    int exp;
} PolyTerm;
typedef struct{
    PolyTerm *data;
    int last;
} SqPoly;
// 建立一个多项式
Status PolyInit(SqPoly &L)
{
    int i;
    PolyTerm *p;
    cout<<"请输入多项式的项数:";
    cin>>L.last;
    L.data=(PolyTerm *)malloc(L.last*sizeof(PolyTerm));
    if(!L.data) return ERROR;
    p=L.data;
    for(i=0;i<L.last;i++){
        cout<<"请输入系数:";
        cin>>p->coef;
        cout<<"请输入指数:";
        cin>>p->exp;
        p++;
    }
    return OK;
}
// 求多项式的值
double PolySum(SqPoly &L,double x0)
{
    double Pn,x;
    int i,j;
    PolyTerm *p;
    p=L.data;
```

```
    for(i=0,Pn=0;i<L.last;i++,p++){
        for(j=0,x=1;j<p->exp;j++) x=x*x0;
        Pn=Pn+p->coef*x;
    }
    return Pn;
}
```

2.40 采用 2.39 题给定的条件和存储结构，编写求 $P(x)=P_{n1}(x)-P_{n2}(x)$ 的算法，将结果多项式存放在新辟的空间中，并分析你的算法的时间复杂度。

解：

```
// 求两多项式的差
Status PolyMinus(SqPoly &L,SqPoly &L1,SqPoly &L2)
{
    PolyTerm *p,*p1,*p2;
    p=L.data;
    p1=L1.data;
    p2=L2.data;
    int i=0,j=0,k=0;
    while(i<L1.last&&j<L2.last){
        if(p1->exp<p2->exp){
            p->coef=p1->coef;
            p->exp=p1->exp;
            p++; k++;
            p1++;    i++;
        }
        else
            if(p1->exp>p2->exp){
                p->coef=-p2->coef;
                p->exp=p2->exp;
                p++;     k++;
                p2++;    j++;
            }
            else{
                if(p1->coef!=p2->coef){
                    p->coef=(p1->coef)-(p2->coef);
                    p->exp=p1->exp;
                    p++;     k++;
                }
                p1++;    p2++;
                i++;     j++;
            }
    }
    if(i<L1.last)
        while(i<L1.last){
```

```
            p->coef=p1->coef;
            p->exp=p1->exp;
            p++;      k++;
            p1++;     i++;
        }
    if(j<L2.last)
        while(j<L2.last){
            p->coef=-p2->coef;
            p->exp=p2->exp;
            p++;      k++;
            p2++;     j++;
        }
    L.last=k;
    return OK;
}
```

在 2.41 至 2.42 题中，稀疏多项式采用的循环链表存储结构 LinkedPoly 定义为

```
typedef struct PolyNode {
    PolyTerm data;
    struct PolyNode *next;
} PolyNode, *PolyLink;
typedef PolyLink LinkedPoly;
```

2.41 试以循环链表作稀疏多项式的存储结构，编写求其导函数的方法，要求利用原多项式中的结点空间存放其导函数多项式，同时释放所有无用结点。

解：

```
Status PolyDifferential(LinkedPoly &L)
{
    LinkedPoly p,q,pt;
    q=L;
    p=L->next;
    while(p!=L){
        if(p->data.exp==0){
            pt=p;
            p=p->next;
            q->next=p;
            free(pt);
        }
        else{
            p->data.coef=p->data.coef*p->data.exp;
            p->data.exp--;
            q=p;
            p=p->next;
        }
    }
    return OK;
```

```
}
```

2.42 试编写算法，将一个用循环链表表示的稀疏多项式分解成两个多项式，使这两个多项式中各自仅含奇次项或偶次项，并要求利用原链表中的结点空间构成这两个链表。

解：

```
// 将单链表 L 划分成 2 个单循环链表
Status ListDivideInto2CL(LinkedPoly &L,LinkedPoly &L1)
{
    LinkedPoly p,p1,q,pt;
    q=L;
    p=L->next;
    p1=L1;
    while(p!=L){
        if(p->data.exp%2==0){
            pt=p;
            p=p->next;
            q->next=p;
            pt->next=p1->next;
            p1->next=pt;
            p1=p1->next;
        }
        else{
            q=p;
            p=p->next;
        }
    }
    return OK;
}
```

第 3 章 栈和队列

3.1 若按教科书 3.1.1 节中图 3.1(b)所示铁道进行车厢调度（注意：两侧铁道均为单向行驶道），则请回答：

(1) 如果进站的车厢序列为 123，则可能得到的出站车厢序列是什么？

(2) 如果进站的车厢序列为 123456，则能否得到 435612 和 135426 的出站序列，并请说明为什么不能得到或者如何得到（即写出以 'S'表示进栈和以 'X'表示出栈的栈操作序列）。

解：(1) 123　231　321　213　132

(2) 可以得到 135426 的出站序列，但不能得到 435612 的出站序列。因为 4356 出站说明 12 已经在栈中，1 不可能先于 2 出栈。

3.2 简述栈和线性表的差别。

解：线性表是具有相同特性的数据元素的一个有限序列。栈是限定仅在表尾进行插入或删除操作的线性表。

3.3 写出下列程序段的输出结果（栈的元素类型 SElemType 为 char）。

```
void main()
{
```

```
    Stack S;
    char x,y;
    InitStack(S);
    x= 'c'; y= 'k';
    Push(S,x);    Push(S, 'a');  Push(S,y);
    Pop(S,x);Push(S, 't');  Push(S,x);
    Pop(S,x);Push(S, 's');
    while(!StackEmpty(S)) { Pop(S,y); printf(y); }
    printf(x);
}
```

解：stack

3.4 简述以下算法的功能（栈的元素类型 SElemType 为 int）。

```
(1) status algo1(Stack S)
    {
        int i,n,A[255];
        n=0;
        while(!StackEmpty(S)) { n++; Pop(S,A[n]); }
        for(i=1;i<=n;i++) Push(S,A[i]);
    }
(2) status algo2(Stack S,int e)
    {
        Stack T; int d;
        InitStack(T);
        while(!StackEmpty(S)){
            Pop(S,d);
            if(d!=e) Push(T,d);
        }
        while(!StackEmpty(T)){
            Pop(T,d);
            Push(S,d);
        }
    }
```

解：(1) 栈中的数据元素逆置　　(2) 如果栈中存在元素 e，将其从栈中清除

3.5 假设以 S 和 X 分别表示入栈和出栈的操作，则初态和终态均为空栈的入栈和出栈的操作序列可以表示为仅由 S 和 X 组成的序列。称可以操作的序列为合法序列（例如，SXSX 为合法序列，SXXS 为非法序列）。试给出区分给定序列为合法序列或非法序列的一般准则，并证明：两个不同的合法（栈操作）序列（对同一输入序列）不可能得到相同的输出元素（注意：在此指的是元素实体，而不是值）序列。

解：任何前 n 个序列中 S 的个数一定大于 X 的个数。

设两个合法序列为：

T1=S……X……S……

T2=S……X……X……

假定前 n 个操作都相同，从第 n+1 个操作开始，为序列不同的起始操作点。由于前 n 个操作相同，故此时两个栈（不妨为栈 A、B）的存储情况完全相同，假设此时栈顶元素均为 a。

第 n+1 个操作不同，不妨 T1 的第 n+1 个操作为 S，T2 的第 n+1 个操作为 X。T1 为入栈操作，假设将 b

压栈，则 T1 的输出顺序一定是先 b 后 a；而 T2 将 a 退栈，则其输出顺序一定是先 a 后 b。由于 T1 的输出为……ba……，而 T2 的输出顺序为……ab……，说明两个不同的合法栈操作序列的输出元素的序列一定不同。

3.6 试证明：若借助栈由输入序列 12...n 得到的输出序列为 $p_1 p_2 \cdots p_n$（它是输入序列的一个排列），则在输出序列中不可能出现这样的情形：存在着 i<j<k 使 $p_j < p_k < p_i$。

解：这个问题和 3.1 题比较相似。因为输入序列是从小到大排列的，所以若 $p_j < p_k < p_i$，则可以理解为通过输入序列 $p_j\ p_k\ p_i$ 可以得到输出序列 $p_i\ p_j\ p_k$，显然通过序列 123 是无法得到 312 的，参见 3.1 题。所以不可能存在着 i<j<k 使 $p_j < p_k < p_i$。

3.7 按照四则运算加、减、乘、除和幂运算(↑)优先关系的惯例，并仿照教科书 3.2 节例 3-2 的格式，画出对下列算术表达式求值时操作数栈和运算符栈的变化过程：

A-B×C/D+E↑F

解：BC=G　G/D=H　A-H=I　E^F=J　I+J=K

步骤	OPTR 栈	OPND 栈	输入字符	主要操作
1	#		A-B*C/D+E^F#	PUSH(OPND,A)
2	#	A	-B*C/D+E^F#	PUSH(OPTR,-)
3	#-	A	B*C/D+E^F#	PUSH(OPND,B)
4	#-	A B	*C/D+E^F#	PUSH(OPTR,*)
5	#-*	A B	C/D+E^F#	PUSH(OPND,C)
6	#-*	A B C	/D+E^F#	Operate(B,*,C)
7	#-	A G	/D+E^F#	PUSH(OPTR,/)
8	#-/	A G	D+E^F#	PUSH(OPND,D)
9	#-/	A G D	+E^F#	Operate(G,/,D)
10	#-	A H	+E^F#	Operate(A,-,H)
11	#	I	+E^F#	PUSH(OPTR,+)
12	#+	I	E^F#	PUSH(OPND,E)
13	#+	I E	^F#	PUSH(OPTR,^)
14	#+^	I E	F#	PUSH(OPND,F)
15	#+^	I E F	#	Operate(E,^,F)
16	#+	I J	#	Operate(I,+,J)
17	#	K	#	RETURN

3.8 试推导求解 n 阶梵塔问题至少要执行的 move 操作的次数。

解： $2^n - 1$

3.9 试将下列递推过程改写为递归过程。

```
void ditui(int n)
{
    int i;
    i = n;
    while(i>1)
        cout<<i--;
}
```

解：

```
void ditui(int j)
```

```
{
    if(j>1){
        cout<<j;
        ditui(j-1);
    }
    return;
}
```

3.10 试将下列递归过程改写为非递归过程。

```
void test(int &sum)
{
    int x;
    cin>>x;
    if(x==0) sum=0;
    else
    {
        test(sum);
        sum+=x;
    }
    cout<<sum;
}
```

解：

```
void test(int &sum)
{
    Stack s;
    InitStack(s);
    int x;
    do{
        cin>>x;
        Push(s,x);
    }while(x>0);
    while(!StackEmpty(s)){
        Pop(s,x);
        sum+=x;
        cout<<sum<<endl;
    }
    DestoryStack(s);
}
```

3.11 简述队列和堆栈这两种数据类型的相同点和差异处。

解：栈是一种运算受限的线性表，其限制是仅允许在表的一端进行插入和删除运算。

队列也是一种运算受限的线性表，其限制是仅允许在表的一端进行插入，而在表的另一端进行删除。

3.12 写出以下程序段的输出结果（队列中的元素类型 QElemType 为 char）。

```
void main()
{
    Queue Q;
```

```
InitQueue(Q);
char x= 'e', y= 'c';
EnQueue(Q, 'h');
EnQueue(Q, 'r');
EnQueue(Q, y);
DeQueue(Q, x);
EnQueue(Q, x);
DeQueue(Q, x);
EnQueue(Q, 'a');
While(!QueueEmpty(Q))
{
    DeQueue(Q,y);
    cout<<y;
}
cout<<x;
}
```

解：char

3.13 简述以下算法的功能（栈和队列的元素类型均为 int）。

```
void algo3(Queue &Q)
{
    Stack S;
    int d;
    InitStack(S);
    while(!QueueEmpty(Q))
    {
        DeQueue(Q, d);
        Push(S, d);
    }
    while(!StackEmpty(S))
    {
        Pop(S, d);
        EnQueue(Q, d);
    }
}
```

解：队列逆置

3.14 若以 1234 作为双端队列的输入序列，试分别求出满足以下条件的输出序列：

(1) 能由输入受限的双端队列得到，但不能由输出受限的双端队列得到的输出序列。

(2) 能由输出受限的双端队列得到，但不能由输入受限的双端队列得到的输出序列。

(3) 既不能由输入受限的双端队列得到，也不能由输出受限的双端队列得到的输出序列。

3.15 假设以顺序存储结构实现一个双向栈，即在一维数组的存储空间中存在着两个栈，它们的栈底分别设在数组的两个端点。试编写实现这个双向栈 tws 的三个操作：初始化 inistack(tws)、入栈 push(tws,i,x)和出栈 pop(tws,i)的算法，其中 i 为 0 或 1，用以分别指示设在数组两端的两个栈，并讨论按过程(正/误状态变量可设为变参)或函数设计这些操作算法各有什么有缺点。

解：

```
class DStack{
     ElemType *top[2];
     ElemType *p;
     int stacksize;
     int di;
public:
     DStack(int m)
     {
          p=new ElemType[m];
          if(!p) exit(OVERFLOW);
          top[0]=p+m/2;
          top[1]=top[0];
          stacksize=m;
     }
     ~DStack(){delete p;}
     void Push(int i,ElemType x)
     {
          di=i;
          if(di==0){
               if(top[0]>=p) *top[0]--=x;
               else cerr<<"Stack overflow!";
          }
          else{
               if(top[1]<p+stacksize-1) *++top[1]=x;
               else cerr<<"Stack overflow!";
          }
     }
     ElemType Pop(int i)
     {
          di=i;
          if(di==0){
               if(top[0]<top[1]) return *++top[0];
               else cerr<<"Stack empty!";
          }else{
               if(top[1]>top[0]) return *top[1]--;
               else cerr<<"Stack empty!";
          }
          return OK;
     }
};

// 链栈的数据结构及方法的定义
typedef struct NodeType{
     ElemType data;
```

```
    NodeType *next;
}NodeType,*LinkType;
typedef struct{
    LinkType top;
    int size;
}Stack;

void InitStack(Stack &s)
{
    s.top=NULL;
    s.size=0;
}

void DestroyStack(Stack &s)
{
    LinkType p;
    while(s.top){
        p=s.top;
        s.top=p->next;
        delete p;
        s.size--;
    }
}

void ClearStack(Stack &s)
{
    LinkType p;
    while(s.top){
        p=s.top;
        s.top=p->next;
        delete p;
        s.size--;
    }
}

int StackLength(Stack s)
{
    return s.size;
}

Status StackEmpty(Stack s)
{
    if(s.size==0) return TRUE;
    else return FALSE;
```

```
}

Status GetTop(Stack s,ElemType &e)
{
     if(!s.top) return ERROR;
     else{
          e=s.top->data;
          return OK;
     }
}

Status Push(Stack &s,ElemType e)
{
     LinkType p;
     p=new NodeType;
     if(!p) exit(OVERFLOW);
     p->next=s.top;
     s.top=p;
     p->data=e;
     s.size++;
     return OK;
}

Status Pop(Stack &s,ElemType &e)
{
     LinkType p;
     if(s.top){
          e=s.top->data;
          p=s.top;
          s.top=p->next;
          delete p;
          s.size--;
     }
     return OK;
}
// 从栈顶到栈底用Visit()函数遍历栈中每个数据元素
void StackTraverse(Stack s,Status (*Visit)(ElemType e))
{
     LinkType p;
     p=s.top;
     while(p) Visit(p->data);
}
```

3.16 假设如题 3.1 所属火车调度站的入口处有 n 节硬席或软席车厢（分别以 H 和 S 表示）等待调度，试编写算法，输出对这 n 节车厢进行调度的操作（即入栈或出栈操作）序列，以使所有的软席车厢都被调整到

硬席车厢之前。

解：

```
int main()
{
    Stack s;
    char Buffer[80];
    int i=0,j=0;
    InitStack(s);
    cout<<"请输入硬席(H)和软席车厢(S)序列：";
    cin>>Buffer;
    cout<<Buffer<<endl;
    while(Buffer[i]){
        if(Buffer[i]=='S'){
            Buffer[j]=Buffer[i];
            j++;
        }
        else Push(s,Buffer[i]);
        i++;
    }
    while(Buffer[j]){
        Pop(s,Buffer[j]);
        j++;
    }
    cout<<Buffer<<endl;
    return 0;
}
```

3.17 试写一个算法，识别一次读入的一个以@为结束符的字符序列是否为形如‘序列 1&序列 2’模式的字符序列。其中序列 1 和序列 2 中都不含字符‘&’，且序列 2 是序列 1 的逆序列。例如，‘a+b&b+a’是属该模式的字符序列，而‘1+3&3-1’则不是。

解：

```
BOOL Symmetry(char a[])
{
    int i=0;
    Stack s;
    InitStack(s);
    ElemType x;
    while(a[i]!='&' && a[i]){
        Push(s,a[i]);
        i++;
    }
    if(a[i]) return FALSE;
    i++;
    while(a[i]){
        Pop(s,x);
```

```
        if(x!=a[i]){
            DestroyStack(s);
            return FALSE;
        }
        i++;
    }
    return TRUE;
}
```

3.18 试写一个判别表达式中开、闭括号是否配对出现的算法。

解：

```
BOOL BracketCorrespondency(char a[])
{
    int i=0;
    Stack s;
    InitStack(s);
    ElemType x;
    while(a[i]){
        switch(a[i]){
        case '(':
            Push(s,a[i]);
            break;
        case '[':
            Push(s,a[i]);
            break;
        case ')':
            GetTop(s,x);
            if(x=='(')    Pop(s,x);
            else return FALSE;
            break;
        case ']':
            GetTop(s,x);
            if(x=='[') Pop(s,x);
            else return FALSE;
            break;
        default:
            break;
        }
        i++;
    }
    if(s.size!=0) return FALSE;
    return TRUE;
}
```

3.20 假设以二维数组 g(1...m, 1...n)表示一个图像区域，g[i,j]表示该区域中点(i,j)所具颜色，其值为从 0 到 k 的整数。编写算法置换点(i_0,j_0)所在区域的颜色。约定和(i_0,j_0)同色的上、下、左、右的邻接点

为同色区域的点。

解：

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

typedef struct{
     int x;
     int y;
}PosType;
typedef struct{
     int Color;
     int Visited;
     PosType seat;
}ElemType;

#include "d:\VC99\Stack.h"

#define M 8
#define N 8

ElemType g[M][N];

void CreateGDS(ElemType g[M][N]);
void ShowGraphArray(ElemType g[M][N]);
void RegionFilling(ElemType g[M][N],PosType CurPos,int NewColor);

int main()
{
     CreateGDS(g);
     ShowGraphArray(g);

     PosType StartPos;
     StartPos.x=5;
     StartPos.y=5;
     int FillColor=6;
     RegionFilling(g,StartPos,FillColor);
     cout<<endl;
     ShowGraphArray(g);
     return 0;
}

void RegionFilling(ElemType g[M][N],PosType CurPos,int FillColor)
{
     Stack s;
```

```
    InitStack(s);
    ElemType e;
    int OldColor=g[CurPos.x][CurPos.y].Color;

    Push(s,g[CurPos.x][CurPos.y]);
    while(!StackEmpty(s)){
        Pop(s,e);
        CurPos=e.seat;
        g[CurPos.x][CurPos.y].Color=FillColor;
        g[CurPos.x][CurPos.y].Visited=1;

        if(CurPos.x<M &&
                !g[CurPos.x+1][CurPos.y].Visited &&
                g[CurPos.x+1][CurPos.y].Color==OldColor
        )
            Push(s,g[CurPos.x+1][CurPos.y]);
        if(CurPos.x>0 &&
            !g[CurPos.x-1][CurPos.y].Visited &&
            g[CurPos.x-1][CurPos.y].Color==OldColor
        )
            Push(s,g[CurPos.x-1][CurPos.y]);
        if(CurPos.y<N &&
            !g[CurPos.x][CurPos.y+1].Visited &&
            g[CurPos.x][CurPos.y+1].Color==OldColor
        )
            Push(s,g[CurPos.x][CurPos.y+1]);
        if(CurPos.y>0 &&
            !g[CurPos.x][CurPos.y-1].Visited &&
            g[CurPos.x][CurPos.y-1].Color==OldColor
        )
            Push(s,g[CurPos.x][CurPos.y-1]);
    }
}

void CreateGDS(ElemType g[M][N])
{
    int i,j;
    for(i=0;i<M;i++)
        for(j=0;j<N;j++){
            g[i][j].seat.x=i;
            g[i][j].seat.y=j;
            g[i][j].Visited=0;
            g[i][j].Color=0;
        }
```

```
    for(i=2;i<5;i++)
        for(j=2;j<4;j++)
            g[i][j].Color=3;
    for(i=5;i<M-1;i++)
        for(j=3;j<6;j++)
            g[i][j].Color=3;
}

void ShowGraphArray(ElemType g[M][N])
{
    int i,j;
    for(i=0;i<M;i++){
        for(j=0;j<N;j++)
            cout<<g[i][j].Color;
        cout<<endl;
    }
}
```

3.21 假设表达式有单字母变量和双目四则运算符构成。试写一个算法，将一个通常书写形式且书写正确的表达式转换为逆波兰表达式。

解：

```
// 输入的表达式串必须为#...#格式
void InversePolandExpression(char Buffer[])
{
    Stack s;
    InitStack(s);
    int i=0,j=0;
    ElemType e;

    Push(s,Buffer[i]);
    i++;
    while(Buffer[i]!='#'){
        if(!IsOperator(Buffer[i])){ // 是操作数
            Buffer[j]=Buffer[i];
            i++;
            j++;
        }
        else{    // 是操作符
            GetTop(s,e);
            if(Prior(e,Buffer[i])){// 当栈顶优先权高于当前序列时，退栈
                Pop(s,e);
                Buffer[j]=e;
                j++;
            }
            else{
```

```
                Push(s,Buffer[i]);
                i++;
            }
        }
    }
    while(!StackEmpty(s)){
        Pop(s,e);
        Buffer[j]=e;
        j++;
    }
}

Status IsOpertor(char c)
{
    char *p="#+-*/";
    while(*p){
        if(*p==c)
            return TRUE;
        p++;
    }
    return FALSE;
}

Status Prior(char c1,char c2)
{
    char ch[]="#+-*/";
    int i=0,j=0;
    while(ch[i] && ch[i]!=c1) i++;
    if(i==2) i--; // 加和减可认为是同级别的运算符
    if(i==4) i--; // 乘和除可认为是同级别的运算符
    while(ch[j] && ch[j]!=c2) j++;
    if(j==2) j--;
    if(j==4) j--;
    if(i>=j) return TRUE;
    else return FALSE;
}
```

3.22 如题 3.21 的假设条件，试写一个算法，对以逆波兰式表示的表达式求值。

解：

```
char CalVal_InverPoland(char Buffer[])
{
    Stack Opnd;
    InitStack(Opnd);
    int i=0;
    char c;
```

```
    ElemType e1,e2;

    while(Buffer[i]!='#'){
        if(!IsOperator(Buffer[i])){
            Push(Opnd,Buffer[i]);
        }
        else{
            Pop(Opnd,e2);
            Pop(Opnd,e1);
            c=Cal(e1,Buffer[i],e2);
            Push(Opnd,c);
        }
        i++;
    }
    return c;
}

char Cal(char c1,char op,char c2)
{
    int x,x1,x2;
    char ch[10];
    ch[0]=c1;
    ch[1]='\0';
    x1=atoi(ch);

    ch[0]=c2;
    ch[1]='\0';
    x2=atoi(ch);

    switch(op){
    case '+':
        x=x1+x2;
        break;
    case '-':
        x=x1-x2;
        break;
    case '*':
        x=x1*x2;
        break;
    case '/':
        x=x1/x2;
        break;
    default:
        break;
```

```
    }
    itoa(x,ch,10);
    return ch[0];
}
```

3.23 如题 3.21 的假设条件，试写一个算法，判断给定的非空后缀表达式是否为正确的逆波兰表达式，如果是，则将它转化为波兰式。

解：

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include "d:\VC99\DSConstant.h"

typedef char ARRAY[30];
typedef ARRAY ElemType;
typedef struct NodeType{
    ElemType data;
    NodeType *next;
}NodeType,*LinkType;
typedef struct{
    LinkType top;
    int size;
}Stack;

void InitStack(Stack &s);
Status Push(Stack &s,ElemType e);
Status Pop(Stack &s,ElemType e);
Status IsOperator(char c);
Status StackEmpty(Stack s);

Status InvToFroPoland(char a[]);

int main()
{
    char a[30];
    cout<<"请输入逆波兰算术表达式字符序列：";
    cin>>a;
    if(InvToFroPoland(a)) cout<<a<<endl;
    else cout<<"输入逆波兰算术表达式字符序列错误!";
    return 0;
}

Status InvToFroPoland(char a[])
{
    Stack s;
```

```
    InitStack(s);
    int i=0;
    ElemType ch;
    ElemType c1;
    ElemType c2;

    while(a[i]!='#'){
        if(!IsOperator(a[i])){
            if(a[i]>='0' && a[i]<='9'){
                ch[0]=a[i];   ch[1]='\0';
                Push(s,ch);
            }
            else return FALSE;
        }
        else{
            ch[0]=a[i];
            ch[1]='\0';
            if(!StackEmpty(s)){
                Pop(s,c2);
                if(!StackEmpty(s)){
                    Pop(s,c1);
                    strcat(ch,c1);
                    strcat(ch,c2);
                    Push(s,ch);
                }
                else return FALSE;
            }
            else return FALSE;
        }
        i++;
    }
    if(!StackEmpty(s)){
        Pop(s,c1);
        strcpy(a,c1);
    }
    else return FALSE;
    if(!StackEmpty(s)) return FALSE;
    return OK;
}
void InitStack(Stack &s)
{
    s.top=NULL;
    s.size=0;
}
```

```
Status Push(Stack &s,ElemType e)
{
     LinkType p;
     p=new NodeType;
     if(!p) exit(OVERFLOW);
     p->next=s.top;
     s.top=p;
     strcpy(p->data,e);
     s.size++;
     return OK;
}

Status Pop(Stack &s,ElemType e)
{
     LinkType p;
     if(s.top){
          strcpy(e,s.top->data);
          p=s.top;
          s.top=p->next;
          delete p;
          s.size--;
     }
     return OK;
}

Status StackEmpty(Stack s)
{
     if(s.size==0) return TRUE;
     else return FALSE;
}

Status IsOperator(char c)
{
     char *p="#+-*/";
     while(*p){
          if(*p==c)
               return TRUE;
          p++;
     }
     return FALSE;
}
```

3.24 试编写如下定义的递归函数的递归算法，并根据算法画出求 g(5,2)时栈的变化过程。

$$g(m,n)=\begin{cases}0 & m=0,n\geq 0\\ g(m-1,2n)+n & m>0,n\geq 0\end{cases}$$

解：

```
int g(int m,int n);
int main()
{
    int m,n;
    cout<<"请输入 m 和 n 的值：";
    cin>>m>>n;
    if(n>=0) cout<<g(m,n)<<endl;
    else cout<<"No Solution!";
    return 0;
}
int g(int m,int n)
{
    if(m>0)
        return(g(m-1,2*n)+n);
    else return 0;
}
```

假设主函数的返回地址为 0，递归函数 3 条语句的地址分别为 1、2、3。

3	0	64
3	1	32
3	2	16
3	3	8
3	4	4
0	5	2

3.25 试写出求递归函数 F(n)的递归算法，并消除递归：

$$F(n)=\begin{cases}n+1 & n=0\\ n\cdot F(n/2) & n>0\end{cases}$$

解：

```
#include <iostream.h>
#define N  20
int main()
{
    int i;
    int a[N];
    int n;
    cout<<"请输入 n:";
    cin>>n;
    for(i=0;i<n+1;i++){
        if(i<1) a[i]=1;
        else a[i]=i*a[i/2];
    }
    cout<<a[n]<<endl;
```

```
    return 0;
}
```

3.26 求解平方根$\sqrt{A}$的迭代函数定义如下：

$$sqrt(A,p,e)=\begin{cases} p & |p^2-A|<e \\ sqrt\left(A,\frac{1}{2}\left(p+\frac{A}{p}\right),e\right) & |p^2-A|\geq e \end{cases}$$

其中，p是A的近似平方根，e是结果允许误差。试写出相应的递归算法，并消除递归。

解：

```
#include <iostream.h>
double Sqrt(double A,double p,double e);
int main()
{
    double A,p,e;
    cout<<"请输入 A p e:";
    cin>>A>>p>>e;
    cout<<Sqrt(A,p,e)<<endl;
    return 0;
}
double Sqrt(double A,double p,double e)
{
    if((p*p-A)>-e  && (p*p-A)<e)
        return p;
    else
        return Sqrt(A,(p+A/p)/2,e);
}
```

3.27 已知Ackerman函数的定义如下：

$$akm(m,n)=\begin{cases} n+1 & m=0 \\ akm(m-1,1) & m\neq 0,n=0 \\ akm(m-1,akm(m,n-1)) & m\neq 0,n\neq 0 \end{cases}$$

(1) 写出递归算法；

(2) 写出非递归算法；

(3) 根据非递归算法，画出求akm(2,1)时栈的变化过程。

解：

```
unsigned int akm(unsigned int m,unsigned int n)
{
    unsigned int g;
    if(m==0)
        return n+1;
    else
        if(n==0) return akm(m-1,1);
        else{
```

```
            g=akm(m,n-1);
            return akm(m-1,g);
        }
}
```

非递归算法：

```
int akm1(int m,int n)
{
    Stack s;
    InitStack(s);
    ElemType e,e1,d;
    e.mval=m;     e.nval=n;
    Push(s,e);
    do{
        while(e.mval){
            while(e.nval){
                e.nval--;
                Push(s,e);
            }
            e.mval--;e.nval=1;
        }
        if(StackLength(s)>1){
            e1.nval=e.nval;
            Pop(s,e);
            e.mval--;
            e.nval=e1.nval+1;
        }
    }while(StackLength(s)!=1||e.mval!=0);
    return e.nval+1;
}
```

```
0,akm(2,1)    0 2 1          g=akm(2,0)
1,akm(2,0)    6 2 0          akm=akm(m-1,1)=akm(1,1)
2,akm(1,1)    4 1 1          g=akm(m,n-1)=akm(1,0)
3,akm(1,0)    6 1 0          akm=akm(m-1,1)=akm(0,1)
4,akm(0,1)    4 0 1          akm=n+1=2     退栈

0,akm(2,1)    0 2 1          g=akm(2,0)
1,akm(2,0)    6 2 0          akm=akm(m-1,1)=akm(1,1)
2,akm(1,1)    4 1 1          g=akm(m,n-1)=akm(1,0)
3,akm(1,0)    6 1 0          akm=akm(m-1,1)=akm(0,1)=2 退栈

0,akm(2,1)    0 2 1          g=akm(2,0)
1,akm(2,0)    6 2 0          akm=akm(m-1,1)=akm(1,1)
2,akm(1,1)    4 1 1          g=akm(m,n-1)=akm(1,0)=2; akm=akm(m-1,g)=akm(0,2);
```

3,akm(0,2)　7 0 2　　akm=n+1=3 退栈

0,akm(2,1)　0 2 1　　g=akm(2,0)
1,akm(2,0)　6 2 0　　akm=akm(m-1,1)=akm(1,1)
2,akm(1,1)　4 1 1　　g=akm(m,n-1)=akm(1,0)=2; akm=akm(m-1,g)=akm(0,2)=3; 退栈

0,akm(2,1)　0 2 1　　g=akm(2,0)
1,akm(2,0)　6 2 0　　akm=akm(m-1,1)=akm(1,1)=3 退栈

0,akm(2,1)　0 2 1　　g=akm(2,0)=3; akm=akm(1,3)
1,akm(1,3)　6 1 3　　g=akm(1,2); akm(m-1,g)
2,akm(1,2)　6 1 2　　g=akm(1,1); akm(m-1,g)
3,akm(1,1)　6 1 1　　g=akm(1,0); akm(m-1,g)
4,akm(1,0)　6 1 0　　akm=akm(0,1)
5,akm(0,1)　4 0 1　　akm(0,1)=2 退栈

0,akm(2,1)　0 2 1　　g=akm(2,0)=3; akm=akm(1,3)
1,akm(1,3)　6 1 3　　g=akm(1,2); akm(m-1,g)
2,akm(1,2)　6 1 2　　g=akm(1,1); akm(m-1,g)
3,akm(1,1)　6 1 1　　g=akm(1,0); akm(m-1,g)
4,akm(1,0)　6 1 0　　akm=akm(0,1)=2 退栈

0,akm(2,1)　0 2 1　　g=akm(2,0)=3; akm=akm(1,3)
1,akm(1,3)　6 1 3　　g=akm(1,2); akm(m-1,g)
2,akm(1,2)　6 1 2　　g=akm(1,1); akm(m-1,g)
3,akm(1,1)　6 1 1　　g=akm(1,0)=2; akm(m-1,g)=akm(0,2)
4,akm(0,2)　7 0 2　　akm=n+1=3 退栈

0,akm(2,1)　0 2 1　　g=akm(2,0)=3; akm=akm(1,3)
1,akm(1,3)　6 1 3　　g=akm(1,2); akm(m-1,g)
2,akm(1,2)　6 1 2　　g=akm(1,1); akm(m-1,g)
3,akm(1,1)　6 1 1　　g=akm(1,0)=2; akm(m-1,g)=akm(0,2)=3 退栈

0,akm(2,1)　0 2 1　　g=akm(2,0)=3; akm=akm(1,3)
1,akm(1,3)　6 1 3　　g=akm(1,2); akm(m-1,g)
2,akm(1,2)　6 1 2　　g=akm(1,1)=3; akm(m-1,g)=akm(0,3)
3,akm(0,3)　7 0 3　　akm=n+1=4 退栈

0,akm(2,1)　0 2 1　　g=akm(2,0)=3; akm=akm(1,3)
1,akm(1,3)　6 1 3　　g=akm(1,2); akm(m-1,g)
2,akm(1,2)　6 1 2　　g=akm(1,1)=3; akm(m-1,g)=akm(0,3)=4 退栈

0,akm(2,1)　0 2 1　　g=akm(2,0)=3; akm=akm(1,3)

1,akm(1,3)　　6 1 3　　g=akm(1,2)=4; akm(m-1,g)=akm(0,4)

2,akm(0,4)　　7 0 4　　akm=n+1=5 退栈

0,akm(2,1)　　0 2 1　　g=akm(2,0)=3; akm=akm(1,3)

1,akm(1,3)　　6 1 3　　g=akm(1,2)=4; akm(m-1,g)=akm(0,4)=5 退栈

0,akm(2,1)　　0 2 1　　g=akm(2,0)=3; akm=akm(1,3)=5 退栈

akm(2,1)=5;

3.28 假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素结点（注意不设头指针），试编写相应的队列初始化、入队列何处队列的算法。

解：

```
typedef int ElemType;
typedef struct NodeType{
    ElemType data;
    NodeType *next;
}QNode,*QPtr;
typedef struct{
    QPtr rear;
    int size;
}Queue;
Status InitQueue(Queue& q)
{
    q.rear=NULL;
    q.size=0;
    return OK;
}
Status EnQueue(Queue& q,ElemType e)
{
    QPtr p;
    p=new QNode;
    if(!p) return FALSE;
    p->data=e;
    if(!q.rear){
        q.rear=p;
        p->next=q.rear;
    }
    else{
        p->next=q.rear->next;
        q.rear->next=p;
        q.rear=p;
    }
    q.size++;
    return OK;
```

```
}
Status DeQueue(Queue& q,ElemType& e)
{
     QPtr p;
     if(q.size==0)return FALSE;
     if(q.size==1){
          p=q.rear;
          e=p->data;
          q.rear=NULL;
          delete p;
     }
     else{
          p=q.rear->next;
          e=p->data;
          q.rear->next=p->next;
          delete p;
     }
     q.size--;
     return OK;
}
```

3.29 如果希望循环队列中的元素都能得到利用，则需设置一个标志域 tag，并以 tag 的值为 0 和 1 来区分，尾指针和头指针值相同时的队列状态是“空”还是“满”。试编写与此结构相应的入队列和出队列的算法，并从时间和空间角度讨论设标志和不设标志这两种方法的使用范围（如当循环队列容量较小而队列中每个元素占的空间较多时，哪一种方法较好）。

解：

```
#define MaxQSize   4
typedef int ElemType;
typedef struct{
     ElemType *base;
     int front;
     int rear;
     Status tag;
}Queue;
Status InitQueue(Queue& q)
{
     q.base=new ElemType[MaxQSize];
     if(!q.base) return FALSE;
     q.front=0;
     q.rear=0;
     q.tag=0;
     return OK;
}
Status EnQueue(Queue& q,ElemType e)
{
```

```
    if(q.front==q.rear&&q.tag) return FALSE;
    else{
        q.base[q.rear]=e;
        q.rear=(q.rear+1)%MaxQSize;
        if(q.rear==q.front)q.tag=1;
    }
    return OK;
}
Status DeQueue(Queue& q,ElemType& e)
{
    if(q.front==q.rear&&!q.tag)return FALSE;
    else{
        e=q.base[q.front];
        q.front=(q.front+1)%MaxQSize;
        q.tag=0;
    }
    return OK;
}
```

设标志节省存储空间，但运行时间较长。不设标志则正好相反。

3.30 假设将循环队列定义为:以域变量 rear 和 length 分别指示循环队列中队尾元素的位置和内含元素的个数。试给出此循环队列的队满条件，并写出相应的入队列和出队列的算法（在出队列的算法中要返回队头元素）。

解：

```
#define MaxQSize   4
typedef int ElemType;
typedef struct{
    ElemType *base;
    int rear;
    int length;
}Queue;
Status InitQueue(Queue& q)
{
    q.base=new ElemType[MaxQSize];
    if(!q.base) return FALSE;
    q.rear=0;
    q.length=0;
    return OK;
}
Status EnQueue(Queue& q,ElemType e)
{
    if((q.rear+1)%MaxQSize==(q.rear+MaxQSize-q.length)%MaxQSize)
        return FALSE;
    else{
        q.base[q.rear]=e;
```

```
        q.rear=(q.rear+1)%MaxQSize;
        q.length++;
    }
    return OK;
}
Status DeQueue(Queue& q,ElemType& e)
{
    if((q.rear+MaxQSize-q.length)%MaxQSize==q.rear)
        return FALSE;
    else{
        e=q.base[(q.rear+MaxQSize-q.length)%MaxQSize];
        q.length--;
    }
    return OK;
}
```

3.31 假设称正读和反读都相同的字符序列为“回文”，例如，‘abba’和‘abcba’是回文，‘abcde’和‘ababab’则不是回文。试写一个算法判别读入的一个以‘@’为结束符的字符序列是否是“回文”。

解：

```
Status SymmetryString(char* p)
{
    Queue q;
    if(!InitQueue(q)) return 0;
    Stack s;
    InitStack(s);
    ElemType e1,e2;
    while(*p){
        Push(s,*p);
        EnQueue(q,*p);
        p++;
    }
    while(!StackEmpty(s)){
        Pop(s,e1);
        DeQueue(q,e2);
        if(e1!=e2) return FALSE;
    }
    return OK;
}
```

3.32 试利用循环队列编写求 k 阶菲波那契序列中前 n+1 项的算法，要求满足：$f_n \le \max$ 而 $f_{n+1} > \max$，其中 max 为某个约定的常数。（注意：本题所用循环队列的容量仅为 k，则在算法执行结束时，留在循环队列中的元素应是所求 k 阶菲波那契序列中的最后 k 项）

解：

```
int Fibonacci(int k,int n)
{
```

```
    if(k<1) exit(OVERFLOW);
    Queue q;
    InitQueue(q,k);
    ElemType x,e;
    int i=0;
    while(i<=n){
        if(i<k-1){
            if(!EnQueue(q,0)) exit(OVERFLOW);
        }
        if(i==k-1){
            if(!EnQueue(q,1)) exit(OVERFLOW);
        }
        if(i>=k){
            // 队列求和
            x=sum(q);
            DeQueue(q,e);
            EnQueue(q,x);
        }
        i++;
    }
    return q.base[(q.rear+q.MaxSize-1)%q.MaxSize];
}
```

3.33 在顺序存储结构上实现输出受限的双端循环队列的入列和出列（只允许队头出列）算法。设每个元素表示一个待处理的作业，元素值表示作业的预计时间。入队列采取简化的短作业优先原则，若一个新提交的作业的预计执行时间小于队头和队尾作业的平均时间，则插入在队头，否则插入在队尾。

解：

```
// Filename:Queue.h
typedef struct{
    ElemType *base;
    int front;
    int rear;
    Status tag;
    int MaxSize;
}DQueue;
Status InitDQueue(DQueue& q,int size)
{
    q.MaxSize=size;
    q.base=new ElemType[q.MaxSize];
    if(!q.base) return FALSE;
    q.front=0;
    q.rear=0;
    q.tag=0;
    return OK;
}
```

```
Status EnDQueue(DQueue& q,ElemType e)
{
     if(q.front==q.rear&&q.tag) return FALSE;
     if(q.front==q.rear&&!q.tag){ // 空队列
          q.base[q.rear]=e;
          q.rear=(q.rear+1)%q.MaxSize;
          if(q.rear==q.front)q.tag=1;
     }
     else{ // 非空非满
          if(e<(q.base[q.front]+q.base[(q.rear+q.MaxSize-1)%q.MaxSize])/2){
               // 从队头入队
               q.front=(q.front+q.MaxSize-1)%q.MaxSize;
               q.base[q.front]=e;
               if(q.rear==q.front)q.tag=1;
          }
          else{ // 从队尾入队
               q.base[q.rear]=e;
               q.rear=(q.rear+1)%q.MaxSize;
               if(q.rear==q.front)q.tag=1;
          }
     }
     return OK;
}
Status DeDQueue(DQueue& q,ElemType& e)
{
     if(q.front==q.rear&&!q.tag)return FALSE;
     else{ // 非空队列
          e=q.base[q.front];
          q.front=(q.front+1)%q.MaxSize;
          q.tag=0;
     }
     return OK;
}
// Filename:XT333.cpp 主程序文件
#include <iostream.h>
#include <stdlib.h>
typedef int ElemType;
#include "D:\VC99\Queue.h"

int main()
{
     int t1,t2,t3,t4;
     ElemType e;
     cout<<"请输入作业 a1、a2、a3、a4 的执行时间: ";
```

```
    cin>>t1>>t2>>t3>>t4;
    DQueue dq;
    InitDQueue(dq,5);
    EnDQueue(dq,t1);
    EnDQueue(dq,t2);
    EnDQueue(dq,t3);
    EnDQueue(dq,t4);
    while(dq.front!=dq.rear||dq.tag){
        DeDQueue(dq,e);
        cout<<e<<endl;
    }
    return 0;
}
```

3.34 假设在如教科书 3.4.1 节中图 3.9 所示的铁道转轨网的输入端有 n 节车厢：硬座、硬卧和软卧（分别以 P,H 和 S 表示）等待调度，要求这三种车厢在输出端铁道上的排列次序为：硬座在前，软卧在中，硬卧在后。试利用输出受限的双端队列对这 n 节车厢进行调度，编写算法输出调度的操作序列：分别以字符'E'和'D'表示对双端队列的头端进行入队列和出队列的操作；以字符 A 表示对双端队列的尾端进行入队列的操作。

解：

```
int main()
{
    ElemType e;
    DQueue dq;
    InitDQueue(dq,20);
    char ch[20];
    cout<<"请输入待调度的车厢字符序列(仅限 PHS)："";
    cin>>ch;
    int i=0;
    while(ch[i]){
        if(ch[i]=='P') cout<<ch[i];
        if(ch[i]=='S') EnDQueue(dq,ch[i],0);// 从队头入队
        if(ch[i]=='H') EnDQueue(dq,ch[i],1);// 从队尾入队
        i++;
    }
    while(dq.front!=dq.rear||dq.tag){
        DeDQueue(dq,e);
        cout<<e;
    }
    cout<<endl;
    return 0;
}
```

第 4 章　串

4.1 解：空格串是指一个或多个空格字符(ASCII 码为 20H)组成的串，而空串中没有任何字符。

4.2 解：串赋值(StrAssign)、串比较(StrCompare)、求串长(StrLength)、串连接(Concat)、求子串(SubString)这五种基本操作构成串类型的最小操作子集。

4.6 解：

```
s1=SubString(s,3,1)
s2=SubString(s,6,1)
Replace(s,s1,s2)
Concat(s3,s,s1)
Concat(t,SubString(s3,1,5),SubString(s3,7,2))
```

算法设计题：

```
// Filename: String.h
#include <stdlib.h>
#include <string.h>
#define MaxSize 128
class String{
    char *ch;
    int curlen;
public:
    String(const String& ob);
    String(const char* init);
    String();
    ~String();
    void StrAssign(String t);
    int StrCompare(String t);
    int StrLength();
    void Concat(String t);
    String SubString(int start,int len);
    void show();
};
String::String(const String& ob)
{
    ch=new char[MaxSize+1];
    if(!ch) exit(1);
    curlen=ob.curlen;
    strcpy(ch,ob.ch);
}
String::String(const char* init)
{
        ch=new char[MaxSize+1];
    if(!ch) exit(1);
    curlen=strlen(init);
```

```
     strcpy(ch,init);
}
String::String()
{
     ch=new char[MaxSize+1];
     if(!ch) exit(1);
     curlen=0;
     ch[0]='\0';
}
String::~String()
{
     delete ch;
     curlen=0;
}
void String::StrAssign(String t)
{
     strcpy(ch,t.ch);
     curlen=t.curlen;
}
int String::StrCompare(String t)
{
     return strcmp(ch,t.ch);
}
int String::StrLength()
{
     return curlen;
}
void String::Concat(String t)
{
     strcat(ch,t.ch);
     curlen=curlen+t.curlen;
}
String String::SubString(int start,int len)
{
     String temp;
     int i,j;
     if(start>=0 && start+len<=curlen && len>0){
          temp.curlen=len;
          for(i=0,j=start;i<len;i++,j++)
               temp.ch[i]=ch[j];
          temp.ch[len]='\0';
     }
     return temp;
}
```

```
void String::show()
{
    cout<<ch<<endl;
}
```

4.10 解：

```
void StrReverse(String& s)
{
    String t;
    int i,j;
    j=s.StrLength();
    for(i=j-1;i>=0;i--)
        t.Concat(s.SubString(i,1));
    s.StrAssign(t);
}
```

4.11 解：

第 5 章 数组与广义表

5.1 解：

(1) 6×8×6=288 Byte

(2) LOC(5,7)=1000+(5×8+7)×6=1282

(3) LOC(1,4)=1000+(1×8+4)×6=1072

(4) LOC(4,7)=1000+(7×6+4)×6=1276

5.2 解：

(1) LOC(0,0,0,0)=100

(2) LOC(1,1,1,1)=100+(1×3×5×8 + 1×5×8 + 1×8 + 1)×4=776

(3) LOC(3,1,2,5)=100+(3×3×5×8 + 1×5×8 + 2×8 + 5)×4=1784

(4) LOC(8,2,4,7)=100+(8×3×5×8 + 2×5×8 + 4×8 + 7)×4=4416

5.3 解：

(0,0,0,0) (1,0,0,0) (0,1,0,0) (1,1,0,0) (0,0,1,0) (1,0,1,0) (0,1,1,0) (1,1,1,0)
(0,0,2,0) (1,0,2,0) (0,1,2,0) (1,1,2,0) (0,0,0,1) (1,0,0,1) (0,1,0,1) (1,1,0,1)
(0,0,1,1) (1,0,1,1) (0,1,1,1) (1,1,1,1) (0,0,2,1) (1,0,2,1) (0,1,2,1) (1,1,2,1)
(0,0,0,2) (1,0,0,2) (0,1,0,2) (1,1,0,2) (0,0,1,2) (1,0,1,2) (0,1,1,2) (1,1,1,2)
(0,0,2,2) (1,0,2,2) (0,1,2,2) (1,1,2,2)

5.4 解： $k=\frac{a(a+1)}{2}+b$

其中，a=Max(i,j)，b=Min(i,j)

5.5 解：$k=ni-(n-j)-\frac{i(i-1)}{2}-1$（$i\geq 1$，$j\geq 1$，$i\geq j$）

$f_1(i)=(n+\frac{1}{2})i-\frac{1}{2}i^2$　　$f\ (j)=j$　　$c=-(n+1)$

5.6 解：u=i-j+1　　v=j-1

5.7 解：(1) k=2(i-1)+j-1　　（$|i-j|\leq 1$）

(2) i=(k+1) DIV 3 + 1　（$0\leq k\leq 3n-1$）

j=k+1-2(k DIV 3)

5.8 解：i 为奇数时，k=i+j-2

j 为偶数时，k=i+j-1

k=2(i DIV 2)+j-1

5.9 解：设稀疏矩阵为 n 行 n 列，其中的非零元为 m 个，m 远小于 n^2。从时间上来说，采用二维数组存储稀疏矩阵需要 n^2-1 次加法运算，而用三元组只需 m-1 次加法运算。从空间上来说，用二维数组需要 n^2 个基本存储单元，而三元组需要 m 个基本存储单元外加 2m 个整型存储单元。由于 n^2 远远大于 m，故实际存储空间也较大。

5.10 解：

(1) GetHead【(p,h,w)】= p

(2) GetTail【(b,k,p,h)】= (k,p,h)

(3) GetHead【((a,b),(c,d))】= (a,b)

(4) GetTail【((a,b),(c,d))】= ((c,d))

(5) GetHead【GetTail【((a,b),(c,d))】】= GetHead【((c,d))】= (c,d)

(6) GetTail【GetHead【((a,b),(c,d))】】= GetTail【(a,b)】= (b)

(7) GetHead【GetTail【GetHead【((a,b),(c,d))】】】= GetHead【(b)】= b

(8) GetTail【GetHead【GetTail【((a,b),(c,d))】】】= GetTail【(c,d)】= (d)

5.11 解：

(1) GetHead【GetTail【GetTail【L1】】】

(2) GetHead【GetHead【GetTail【L2】】】

(3) GetHead【GetHead【GetTail【GetTail【GetHead【L3】】】】】

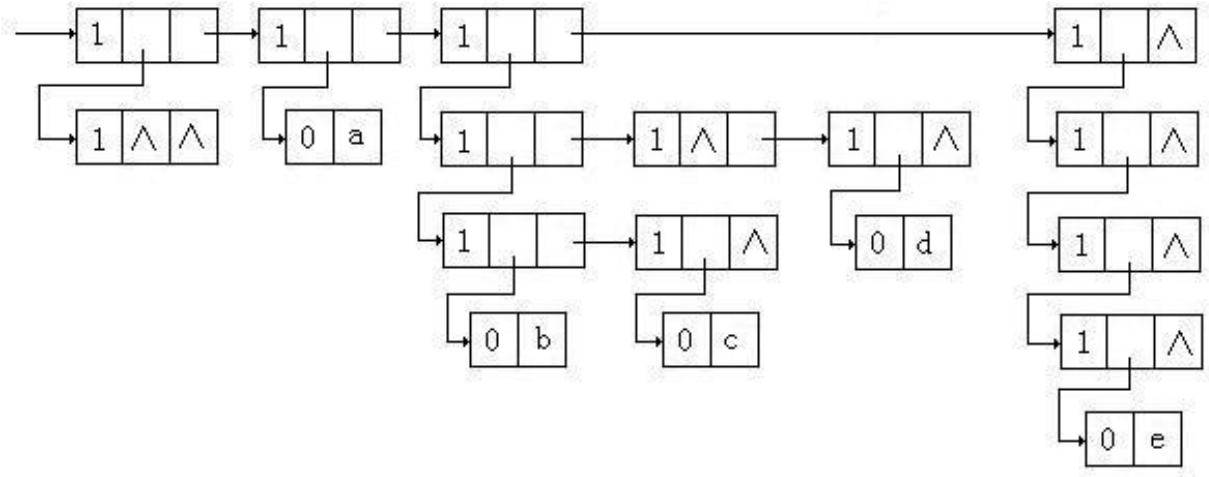
(4) GetHead【GetHead【GetHead【GetTail【GetTail【L4】】】】】

(5) GetHead【GetHead【GetTail【GetTail【L5】】】】

(6) GetHead【GetTail【GetHead【L6】】】

(7) GetHead【GetHead【GetTail【GetHead【GetTail【L7】】】】】

5.12 解：



5.13 解：

(1) List=((x,(y)),((())),(()),(z)))

(2) List=(((a,b,()),()),(a,(b)),())

5.14 解： $s(n)=s(n-1)+a_n=s(n-1)+a_1+(n-1)d$ (n>=1)

```
ElemType s(int i)
{
    if(i>1)
        return s(i-1)+a1+(i-1)*d;
    else
        return a1;
}
```

5.16 解： $add(a,b)=\begin{cases} a & b=0 \\ add(++a,--b) & b>0 \end{cases}$

5.17 解：

```
int Max(SqList &L,int k)
{
    if(k<L.length-1)
        if(L.elem[k]<Max(L,k+1))
            return Max(L,k+1);
        else
            return L.elem[k];
    else
        return L.elem[k];
}
int Min(SqList &L,int k)
{
    if(k<L.length-1)
        if(L.elem[k]>Min(L,k+1))
            return Min(L,k+1);
        else
            return L.elem[k];
    else
        return L.elem[k];
}
int Sum(SqList &L,int k)
{
    if(k==0)
        return L.elem[0];
    else
        return L.elem[k]+Sum(a,k-1);
}
int Product(SqList &L,int k)
{
    if(k==0)
```

```
        return L.elem[0];
    else
        return L.elem[k]*Sum(a,k-1);
}
double Avg(SqList &L,int k)
{
    if(k==0)
        return L.elem[0];
    else
        return (Avg(a,k-1)*k+L.elem[k])/(k+1);
}
```

5.18 解：算法的基本思想是将数组分成 k 组，将第一组与第二组进行两两交换，再将第一组与第三组进行两两交换，...，总共需进行 n-k 次交换。注意最后一组可能出现不足 k 个元素的情况，此时最后一组为剩余元素加第一组的前几个元素共 k 个构成最后一组。

```
void RRMove(ElemType A[],int k,int n)
{
    ElemType e;
    int i=0,j,p;
    while(i<n-k){
        p=i/k+1;
        for(j=0;j<k;j++){
            e=A[j];
            A[j]=A[(p*k+j)%n];
            A[(p*k+j)%n]=e;
            i++;
        }
    }
}
```

5.19 解：

```
#include <iostream.h>
#define RS 4
#define CS 4

typedef int ElemType;
typedef struct{
    ElemType e;
    int i,j;
    int Flags;
}NodeType;

void Initialize(NodeType a[RS][CS],ElemType A[RS][CS]);
void SaddlePoint(NodeType a[RS][CS]);
ElemType RowMin(NodeType a[RS][CS],int k);
ElemType ColMax(NodeType a[RS][CS],int k);
```

```
void Show(NodeType a[RS][CS]);

int main()
{
    ElemType A[RS][CS]={
        {2,1,3,4},
        {1,3,1,2},
        {2,7,1,3},
        {3,2,4,1}
    };
    NodeType a[RS][CS];
    Initialize(a,A);
    SaddlePoint(a);
    Show(a);
    return 0;
}

void Initialize(NodeType a[RS][CS],ElemType A[RS][CS])
{
    int i,j;
    for(i=0;i<RS;i++){
        for(j=0;j<CS;j++){
            a[i][j].e=A[i][j];
            a[i][j].i=i;
            a[i][j].j=j;
            a[i][j].Flags=0;
        }
    }
}

void SaddlePoint(NodeType a[RS][CS])
{
    int i,j;
    ElemType x,y;
    for(i=0;i<RS;i++){
        x=RowMin(a,i);
        for(j=0;j<CS;j++){
            y=ColMax(a,j);
            if(a[i][j].e==x&&a[i][j].e==y)
                a[i][j].Flags=1;
        }
    }
}
```

```
ElemType RowMin(NodeType a[RS][CS],int k)
{
     ElemType x;
     x=a[k][0].e;
     int i;
     for(i=1;i<CS;i++)
          if(x>a[k][i].e){
               x=a[k][i].e;
          }
     return x;
}

ElemType ColMax(NodeType a[RS][CS],int k)
{
     ElemType x;
     x=a[0][k].e;
     int i;
     for(i=1;i<RS;i++)
          if(x<a[i][k].e){
               x=a[i][k].e;
          }
     return x;
}

void Show(NodeType a[RS][CS])
{
     for(int i=0;i<RS;i++)
          for(int j=0;j<CS;j++)
               if(a[i][j].Flags)
                    cout<<i<<" "<<j<<" is a saddle point"<<endl;
}
```

5.21 解：

```
typedef int ElemType;
class Triple{
public:
     int row;
     int col;
     ElemType e;

     Triple(){}
     virtual ~Triple(){}
     BOOL operator<(Triple b);
     BOOL operator==(Triple b);
};
```

```
BOOL Triple::operator<(Triple b)
{
     if(row<b.row) return TRUE;
     if(row==b.row&&col<b.col) return TRUE;
     return FALSE;
}
BOOL Triple::operator==(Triple b)
{
     if(row==b.row && col==b.col)
          return TRUE;
     else
          return FALSE;
}

class CSparseMat
{
public:
     CSparseMat(){}
     virtual ~CSparseMat(){}
     CSparseMat(int r,int c,int n);
     CSparseMat operator+(CSparseMat B);
     void ShowSparse(CDC* pDC);

     Triple *m_pt; // 指向非零元的指针
     int m_nCol;        // 矩阵列数
     int m_nRow;        // 矩阵行数
     int m_nTrs;        // 非零元个数
};

CSparseMat::CSparseMat(int r, int c, int n)
{
     m_nRow=r;
     m_nCol=c;
     m_nTrs=n;
     m_pt=new Triple[m_nTrs];

     // 输入矩阵的所有三元组
     int i;
     for(i=0;i<m_nTrs;i++){
          CInputDlg dlg1;
          if(dlg1.DoModal()==IDOK){
               m_pt[i].row=dlg1.m_nRow;
               m_pt[i].col=dlg1.m_nCol;
```

```
                m_pt[i].e=dlg1.m_nElem;
            }
        }
}

void CSparseMat::ShowSparse(CDC *pDC)
{
     char str[10];
     int k=0;
     for(int i=0;i<m_nRow;i++){
          for(int j=0;j<m_nCol;j++){
               if(m_pt[k].row==i && m_pt[k].col==j){
                    itoa(m_pt[k].e,str,10);
                    k++;
               }
               else itoa(0,str,10);
               pDC->TextOut(0+j*20,0+i*20,str,strlen(str));
          }
     }
}

// 矩阵相加的运算符重载函数
CSparseMat CSparseMat::operator+(CSparseMat B)
{
     CSparseMat temp(m_nRow,m_nCol,0);
     if(m_nRow!=B.m_nRow || m_nCol!=B.m_nCol)
          return temp;

     temp.m_pt=new Triple[m_nTrs+B.m_nTrs];
     if(!temp.m_pt) return temp;
     temp.m_nTrs=m_nTrs+B.m_nTrs;

     int i=0;
     int j=0;
     int k=0;
     while(i<m_nTrs && j<B.m_nTrs){
          if(m_pt[i]<B.m_pt[j]){
               temp.m_pt[k].row=m_pt[i].row;
               temp.m_pt[k].col=m_pt[i].col;
               temp.m_pt[k].e=m_pt[i].e;
               i++;
          }
          else{
               if(m_pt[i]==B.m_pt[j]){
```

```
                    temp.m_pt[k].row=m_pt[i].row;
                    temp.m_pt[k].col=m_pt[i].col;
                    temp.m_pt[k].e=m_pt[i].e+B.m_pt[j].e;
                    i++; j++;
               }
               else{
                    temp.m_pt[k].row=B.m_pt[j].row;
                    temp.m_pt[k].col=B.m_pt[j].col;
                    temp.m_pt[k].e=B.m_pt[j].e;
                    j++;
               }
          }
          k++;
     }
     while(i<m_nTrs){
          temp.m_pt[k].row=m_pt[i].row;
          temp.m_pt[k].col=m_pt[i].col;
          temp.m_pt[k].e=m_pt[i].e;
          i++;
          k++;
     }
     while(j<B.m_nTrs){
          temp.m_pt[k].row=B.m_pt[j].row;
          temp.m_pt[k].col=B.m_pt[j].col;
          temp.m_pt[k].e=B.m_pt[j].e;
          j++;
          k++;
     }
     temp.m_nTrs=k;
     return temp;
}
```

5.23 解：

```
#include<iostream.h>
#include<stdlib.h>
#define Max 128

typedef int ElemType;
typedef struct{
     int col;
     ElemType e;
}Twin;

class CSparseMat
{
```

```
public:
    CSparseMat(){}
    CSparseMat(int r,int c,int n);
    virtual ~CSparseMat(){}
    void ShowSparse(int i,int j);

    Twin* m_pt;   // 指向非零元的指针
    int rpos[Max];
    int m_nCol;        // 矩阵列数
    int m_nRow;        // 矩阵行数
    int m_nTws;        // 非零元个数
};

CSparseMat::CSparseMat(int r, int c, int n)
{
    m_nRow=r;
    m_nCol=c;
    m_nTws=n;
    m_pt=new Twin[m_nTws];
    if(!m_pt) return;

    // 输入矩阵的所有二元组
    int i;
    for(i=0;i<m_nTws;i++){
        cout<<"请输入非零元二元组的列标和值：";
        cin>>m_pt[i].col>>m_pt[i].e;
    }
    for(i=0;i<m_nRow;i++){
        cout<<"请输入每行第一个非零元在二元组中的序号(没有输入-1)：";
        cin>>rpos[i]; // 该行没有非零元输入-1
    }
}

void CSparseMat::ShowSparse(int i,int j)
{
    if(i>m_nRow||j>m_nCol) return;

    ElemType x=0;
    int s,d;
    if(i==m_nRow){
        s=rpos[i];
        d=m_nTws;
    }
    else{
```

```
            s=rpos[i];
            int m=1;
            d=rpos[i+m];
            while(d<0){
                if(i+m<m_nRow){
                    m++;
                    d=rpos[i+m];
                }
                else
                    d=m_nTws;
            }
        }
        if(s>=0){
            int k=s;
            while(k<d){
                if(m_pt[k].col==j)
                    x=m_pt[k].e;
                k++;
            }
        }
        cout<<x<<endl;
}

int main()
{
        CSparseMat A(3,3,5);
        A.ShowSparse(2,1);
        return 0;
}
```

5.26 解：

```
typedef int ElemType;
typedef struct OLNode{
        int row;
        int col;
        ElemType e;
        struct OLNode *right,*down;
}OLNode,*OLink;

class CCrossListMat
{
public:
        OLink *RHead,*CHead;    // 行与列指针向量的头指针
        int m_nCol;         // 矩阵列数
        int m_nRow;         // 矩阵行数
```

```
    int m_nNum;        // 非零元个数
public:
    CCrossListMat(){}
    CCrossListMat(int r,int c,int n);
    virtual ~CCrossListMat(){}
    void ShowMat(int i,int j);
};

CCrossListMat::CCrossListMat(int r, int c, int n)
{
    m_nRow=r;
    m_nCol=c;
    m_nNum=n;
    int i;

    RHead=new OLink[m_nRow];
    if(!RHead) exit(-2);
    CHead=new OLink[m_nCol];
    if(!CHead) exit(-2);
    for(i=0;i<m_nRow;i++)
        RHead[i]=NULL;
    for(i=0;i<m_nCol;i++)
        CHead[i]=NULL;

    OLink p,q,qf;
    for(i=0;i<m_nNum;i++){
        p=new OLNode;
        if(!p) exit(-2);
        cout<<"请输入非零元的行标、列标和值：";
        cin>>p->row>>p->col>>p->e;
        q=RHead[p->row];
        if(!q){
            RHead[p->row]=p;
            p->right=NULL;
        }
        else{
            qf=q;
            while(q && q->col<p->col){
                qf=q;
                q=q->right;
            }
            p->right=qf->right;
            qf->right=p;
        }
```

```
            q=CHead[p->col];
            if(!q){
                CHead[p->col]=p;
                p->down=NULL;
            }
            else{
                qf=q;
                while(q && q->row<p->row){
                    qf=q;
                    q=q->down;
                }
                p->down=qf->down;
                qf->down=p;
            }
        }
}

void CCrossListMat::ShowMat(int i,int j)
{
    ElemType x=0;
    OLink p;
    p=RHead[i];
    while(p && p->col!=j)
        p=p->right;
    if(p)
        x=p->e;
    cout<<x<<endl;
}
```

5.27 解：

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

typedef int ElemType;
typedef struct OLNode{
    int row;
    int col;
    ElemType e;
    struct OLNode *right,*down;
}OLNode,*OLink;

class CCrossListMat
{
public:
    OLink *RHead,*CHead;    // 行与列指针向量的头指针
```

```
    int m_nCol;                 // 矩阵列数
    int m_nRow;                 // 矩阵行数
    int m_nNum;                 // 非零元个数
public:
    CCrossListMat(){}
    virtual ~CCrossListMat(){}
    CCrossListMat(int r,int c,int n);
    void Add(CCrossListMat B);
    void ShowMat();
};

CCrossListMat::CCrossListMat(int r, int c, int n)
{
    m_nRow=r;
    m_nCol=c;
    m_nNum=n;
    int i;

    RHead=new OLink[m_nRow];
    if(!RHead) exit(-2);
    CHead=new OLink[m_nCol];
    if(!CHead) exit(-2);
    for(i=0;i<m_nRow;i++)
        RHead[i]=NULL;
    for(i=0;i<m_nCol;i++)
        CHead[i]=NULL;

    OLink p,q,qf;
    for(i=0;i<m_nNum;i++){
        p=new OLNode;
        if(!p) exit(-2);
        cout<<"请输入非零元的行标、列标和值：";
        cin>>p->row>>p->col>>p->e;
        q=RHead[p->row];
        if(!q){
            RHead[p->row]=p;
            p->right=NULL;
        }
        else{
            qf=q;
            while(q && q->col<p->col){
                qf=q;
                q=q->right;
            }
```

```
                p->right=qf->right;
                qf->right=p;
            }
            q=CHead[p->col];
            if(!q){
                CHead[p->col]=p;
                p->down=NULL;
            }
            else{
                qf=q;
                while(q && q->row<p->row){
                    qf=q;
                    q=q->down;
                }
                p->down=qf->down;
                qf->down=p;
            }
        }
}

void CCrossListMat::Add(CCrossListMat B)
{
    int i,k=0;
    OLink pa,pb;
    OLink pre,p;  // 按行插入
    OLink qpre,q; // 按列插入

    for(i=0;i<m_nRow;i++){
        pa=RHead[i];
        pb=B.RHead[i];
        pre=NULL;

        while(pb){
            while(pa&&pa->col<pb->col){
                pre=pa;
                pa=pa->right;
            }
            if(pa&&pa->col==pb->col){
                pa->e=pa->e+pb->e;
                pb=pb->right;
                pre=pa;
                pa=pa->right;
            }
            else{
```

```
                    // 在 A 中插入一个新结点
                    p=new OLNode;
                    p->row=pb->row;
                    p->col=pb->col;
                    p->e=pb->e;
                    pb=pb->right;
                    if(!pre){
                         p->right=pa;
                         RHead[i]=p;
                    }
                    else{
                         p->right=pre;
                         pre->right=p;
                    }
                    // 处理列指针
                    qpre=NULL;
                    q=CHead[p->col];
                    while(q&&q->row<i){
                         qpre=q;
                         q=q->down;
                    }
                    if(!qpre){
                         p->down=q;
                         CHead[p->col]=p;
                    }
                    else{
                         p->down=pre;
                         pre->down;
                    }
                    k++;
               }
          } // end while(pb)
     } // end for
     m_nNum=m_nNum+k;
}

void CCrossListMat::ShowMat()
{
     int i,j;
     OLink p;
     for(i=0;i<m_nRow;i++){
          p=RHead[i];
          for(j=0;j<m_nCol;j++){
               if(p && p->row==i && p->col==j){
```

```
                cout<<p->e<<" ";
                p=p->right;
            }
            else
                cout<<0<<" ";
        }
        cout<<endl;
    }
}

int main()
{
    CCrossListMat A(3,3,4),B(3,3,2);
    A.Add(B);
    A.ShowMat();
    return 0;
}
```

以下是关于广义表算法涉及的描述及方法

```
#include "DSConst.h"    // 常量定义头文件
#include "StrStat.h"    // 字符串定义头文件

// 广义表数据结构声明
typedef char AtomType;
typedef enum{ATOM,LIST} ElemTag;
typedef struct GLNode{
    ElemTag tag;
    union{
        AtomType atom;
        struct GLNode *hp;
    };
    struct GLNode *tp;
}*GList;

// 将非空串 Str 分割成两部分，HStr 为第一个，TStr 为之后的子串
int StrDistrict(CString& Str,CString& HStr,CString& TStr)
{
    int n,i,k;
    CString s1;
    CString s2(","), s3("("), s4(")");    // 定义常量串
    n=Str.StrLength();
    i=1;
    k=0;
```

```
    while(i<=n && s1.StrCompare(s2) || k!=0){
        s1=Str.SubString(i,1);
        if(!s1.StrCompare(s3)) k++;
        else if(!s1.StrCompare(s4)) k--;
        i++;
    }
    if(i<=n){
        HStr=Str.SubString(1,i-2);
        TStr=Str.SubString(i,n-i+1);
    }
    else{
        HStr=Str;
        TStr.StrClear();
    }
    return OK;
}

// 用串 s 建立广义表 L
int CreateGList(GList& L,CString& s)
{
    CString Sub,HSub,TSub; // 子串，表头串，表尾串
    if(s.StrEmpty()) L=NULL;
    else{    // 非空串，建立广义表
        L=new GLNode; // 开辟一个结点
        if(!L) exit(OVERFLOW);

        if(s.StrLength()>1){    // 如果串长大于 1，说明是表结点
            L->tag=LIST;
            Sub=s.SubString(2,s.StrLength()-2);   // 取括号内子串
            if(!Sub.StrEmpty()){    // 建立子表
                StrDistrict(Sub,HSub,TSub);
                if(!HSub.StrEmpty())    // 表头不空
                    CreateGList(L->hp,HSub);
                else L->hp=NULL;
                if(!TSub.StrEmpty())    // 表尾不空
                    CreateGList(L->tp,TSub);
                else L->tp=NULL;
            }
            else{    // 空表
                L->hp=NULL;
                L->tp=NULL;
            }
        }
        else{    // 建立原子结点
```

```
            L->tag=ATOM;
            L->atom=s.GetStr()[0];
            L->tp=NULL;
        }
    }
    return OK;
}

// 显示广义表串
void ShowGList(GList &L)
{
    if(L){
        if(L->tag==LIST){
            cout<<"(";
            if(L->hp)
                ShowGList(L->hp);
            if(L->tp){
                cout<<",";
                ShowGList(L->tp);
            }
            cout<<")";
        }
        else cout<<L->atom;
    }
}
```

5.30 解：

```
// 求广义表深度的递归算法
int GListDepth(GList& L)
{
    int Depth=0;
    int HDepth,TDepth; // 表头深度，表尾深度
    if(!L) return Depth;    // 广义表不存在
    if(L->tag==ATOM) return Depth;   // 原子结点深度为0
    else{
        Depth++; // 表结点深度为1
        HDepth=Depth+GListDepth(L->hp);
        TDepth=Depth+GListDepth(L->tp);
        return HDepth>TDepth?HDepth:TDepth;
    }
}
```

5.31 解：

```
// 由广义表L复制广义表T
int CopyGList(GList& T,GList& L)
```

```
{
    if(!L) T=NULL;
    else{
        T=new GLNode;
        if(!T) exit(OVERFLOW);
        T->tag=L->tag;
        if(L->tag==ATOM) T->atom=L->atom;
        else{
            CopyGList(T->hp,L->hp);
            CopyGList(T->tp,L->tp);
        }
    }
    return OK;
}
```

5.32 解：

```
// 判两广义表是否相等，相等返回 OK，否则返回 FALSE
Status GListCompare(GList& L1,GList& L2)
{
    if(!L1 && !L2) return OK;   // L1 和 L2 均为空表
    if((!L1 && L2) || (L1 && !L2)) return FALSE;
    else{    // L1 和 L2 均非空表
        if(L1->tag==L2->tag){  // 表属性相同
            if(L1->tag==ATOM){ // 均为原子结点
                if(L1->atom==L2->atom) return OK;
                else return FALSE;
            }
            else{    // 均为表结点
                if(GListCompare(L1->hp,L2->hp) &&
                    GListCompare(L1->tp,L2->tp))
                    return OK;    // 表头、表尾均相同
                else return FALSE;
            }
        }
        else return FALSE; // 表属性不同
    }
}
```

5.33 解：

6 章 树和二叉树

6.1 已知一棵树边的集合为{<I,M>, <I,N>, <E,I>, <B,E>, <B,D>, <A,B>, <G,J>, <G,K>, <C,G>, <C,F>, <A,C>}，请画出这棵树，并回答下列问题：

(1) 哪个是根结点？

(2) 哪些是叶子结点？

(3) 哪个是结点 G 的双亲？

(4) 哪些是结点 G 的祖先？

(5) 哪些是结点 G 的孩子？

(6) 哪些是结点 E 的子孙？

(7) 那些是结点 E 的子孙？

(8) 结点 B 和 N 的层次号分别是什么？

(9) 树的深度是多少？

(10) 以结点 C 为根的子树的深度是多少？

6.2 一棵度为 2 的树与一棵二叉树有何区别？

解：二叉树是颗有序树，但度为 2 的树则未必有序。

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

6.4 一棵深度为 H 的满 k 叉树有如下性质：第 H 层上的结点都是叶子结点，其余各层上每个结点都有 k 棵非空子树。如果按层次顺序从 1 开始对全部结点编号，问：

(1) 各层的结点数目是多少？

(2) 编号为 p 的结点的父结点（若存在）的编号是多少？

(3) 编号为 p 的结点的第 i 个儿子结点（若存在）的编号是多少？

(4) 编号为 p 的结点有右兄弟的条件是什么？其右兄弟的编号是多少？

解：(1) $\frac{k^H-1}{k-1}$

(2)如果 p 是其双亲的最小的孩子（右孩子），则 p 减去根结点的一个结点，应是 k 的整数倍，该整数即为所在的组数，每一组为一棵满 k 叉树，正好应为双亲结点的编号。如果 p 是其双亲的最大的孩子（左孩子），则 p+k-1 为其最小的弟弟，再减去一个根结点，除以 k，即为其双亲结点的编号。

综合来说，对于 p 是左孩子的情况，i=(p+k-2)/k；对于 p 是右孩子的情况，i=(p-1)/k

如果左孩子的编号为 p，则其右孩子编号必为 p+k-1，所以，其双亲结点的编号为

$$i=\left\lfloor \frac{p+k-2}{k} \right\rfloor$$ 向下取整，如 1.5 向下取整为 1

(3)结点 p 的右孩子的编号为 kp+1，左孩子的编号为 kp+1-k+1=k(p-1)+2，第 i 个孩子的编号为 k(p-1)+2+i-1=kp-k+i+1。

(4)当(p-1)%k != 0 时，结点 p 有右兄弟，其右兄弟的编号为 p+1。

6.5 已知一棵度为 k 的树中有 n_1 个度为 1 的结点，n_2 个度为 2 的结点，…，n_k 个度为 k 的结点，问该树中有多少个叶子结点？

解：根据树的定义，在一颗树中，除树根结点外，每个结点有且仅有一个前驱结点，也就是说，每个结点与指向它的一个分支一一对应，所以除树根结点之外的结点树等于所有结点的分支数，即度数，从而可得树中的结点数等于所有结点的度数加 1。总结点数为

$$1+n_1+2n_2+3n_3+...+kn_k$$

而度为 0 的结点数就应为总结点数减去度不为 0 的结点数的总和，即

$$n_0=1+n_1+2n_2+3n_3+...+kn_k-(n_1+n_2+n_3+...+n_k)=1+\sum_{i=1}^{k}(i-1)n_i$$

6.6 已知在一棵含有 n 个结点的树中，只有度为 k 的分支结点和度为 0 的叶子结点。试求该树含有的叶子

节点数目。

解：利用上题结论易得结果。设度为 k 的结点个数为n_k，则总结点数为$n=1+n_k k$。叶子结点的数目应等于总结点数减去度不为 0 的结点的数目，即

$$n_0 = n - n_k = n - \frac{n-1}{k}$$

6.7 一棵含有 n 个结点的 k 叉树，可能达到的最大深度和最小深度各为多少？

解：能达到最大深度的树是单支树，其深度为 n。满 k 叉树的深度最小，其深度为

$\lceil \log_k (n(k-1)+1) \rceil$ （证明见徐孝凯著数据结构实用教程 P166）

6.8 证明：一棵满 k 叉树上的叶子结点数n_0和非叶子结点数n_1之间满足以下关系：

$$n_0 = (k-1)n_1 + 1$$

解：一棵满 k 叉树的最后一层(深度为 h)的结点数(叶子结点数)为$n_0 = k^{h-1}$，其总结点数为$\frac{k^h - 1}{k-1}$，

则非叶子结点数$n_1 = \frac{k^h - 1}{k-1} - n_0 = \frac{kn_0 - 1}{k-1} - n_0$，从而得

$$n_0 = (k-1)n_1 + 1$$

6.9 试分别推导含有 n 个结点和含n_0个叶子结点的完全三叉树的深度 H。

解：(1) 根据完全三叉树的定义

$$\frac{k^{H-1}-1}{k-1} < n \le \frac{k^H - 1}{k-1} \qquad 3^{H-1} - 1 < 2n \le 3^H - 1$$

$$2n+1 \le 3^H < 3(2n+1) \qquad \log_3(2n+1) \le H < 1 + \log_3(2n+1)$$

$$H = \log_3(2n+1)$$

(2) 设总的结点数为 n，非叶子结点数为n_1注意到每个非叶子结点的度均为 3，则

$$1 + 3n_1 = n \qquad \text{由 } n_0 + n_1 = n$$

$$n = \frac{3}{2}n_0 - \frac{1}{2} \quad H = \log_3(3n_0) = 1 + \log_3 n_0$$

6.10 对于那些所有非叶子结点均含有左右子数的二叉树：

(1) 试问：有 n 个叶子结点的树中共有多少个结点？

(2) 试证明：$\sum_{i=1}^{n} 2^{-(l_i - 1)} = 1$，其中 n 为叶子结点的个数，$l_i$表示第 i 个叶子结点所在的层次（设根节点所在层次为 1）。

解：（1）总结点数为$1+2n_1$，其中n_1为非叶子结点数，则叶子结点数为$n=n_1+1$，所以总结点数为$2n-1$ 。

（2）用归纳法证明。

i=1，说明二叉树只有一个叶子结点，则整棵树只有一个根结点，$l_1=1$，

$2^{-(l_1-1)}=1$，结论成立。

设有 n 个叶子结点时也成立，即$2^{-(l_1-1)}+2^{-(l_2-1)}+...+2^{-(l_p-1)}+...+2^{-(l_n+1)}=1$，现假设增加一个叶子结点，这意味着在某叶子结点 p 上新生两个叶子结点，而结点 p 则成为非叶子结点，可见，总结点数增 2，叶子结点数增 1。此时，所有叶子结点是原结点除去 p，然后加上两个深度为l_p+1的新叶子结点，由此，

$$2^{-(l_1-1)}+2^{-(l_2-1)}+...+2^{-(l_{p-1}-1)}+2^{-(l_{p+1}-1)}+...+2^{-(l_n+1)}+2^{-(l_p+1-1)}+2^{-(l_p+1-1)}$$

$$=2^{-(l_1-1)}+2^{-(l_2-1)}+...+2^{-(l_p-1)}+...+2^{-(l_n+1)}=1$$

则当 i=n+1 时，也成立，由此即得到证明。

6.11 在二叉树的顺序存储结构中，实际上隐含着双亲的信息，因此可和三叉链表对应。假设每个指针域占 4 个字节，每个信息域占 k 个字节。试问：对于一棵有 n 个结点的二叉树，且在顺序存储结构中最后一个节点的下标为 m，在什么条件下顺序存储结构比三叉链表更节省空间？

解：采用三叉链表结构，需要 n(k+12)个字节的存储空间。采用顺序存储结构，需要 mk 个字节的存储空间，则当 mk<n(k+12)时，即$k<\dfrac{12n}{m-n}$时，采用顺序存储比采用三叉链表更节省空间。

6.12 对题 6.3 所得各种形态的二叉树，分别写出前序、中序和后序遍历的序列。

6.13 假设 n 和 m 为二叉树中两结点，用 1、0 或#（分别表示肯定、恰恰相反或不一定）填写下表：

问 已知	前序遍历时 n 在 m 前？	中序遍历时 n 在 m 前？	后序遍历时 n 在 m 前？
n 在 m 左方			
n 在 m 右左方			
n 是 m 祖先			
n 是 m 子孙			

注：如果(1)离 a 和 b 最近的共同祖先 p 存在，且(2)a 在 p 的左子树中，b 在 p 的右子树中，则称 a 在 b 的左方（即 b 在 a 的右方）。

6.14 找出所有满足下列条件的二叉树：

(a) 它们在先序遍历和中序遍历时，得到的节点访问序列相同；

(b) 它们在后序遍历和中序遍历时，得到的结点访问序列相同；

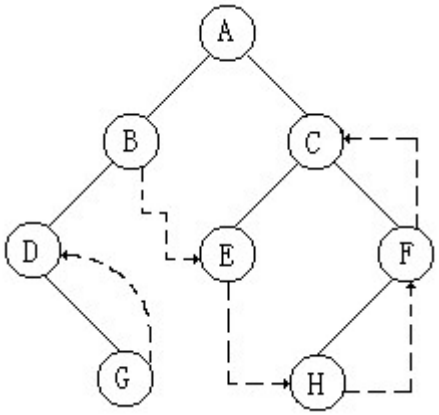
(c) 它们在先序遍历和后序遍历时，得到的节点访问序列相同。

解：(a) 不含左子树的二叉树。

(b) 不含右子树的二叉树。

(c) 即不含左子树，也不含右子树的二叉树。

6.15 解：



6.16 解： 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Info	A	B	C	D	E	F	G	H	I	J	K	L	M	N
Ltag	0	0	0	1	0	1	0	1	0	0	1	1	1	1
Lchild	2	4	6	2	7	3	10	14	12	13	13	9	10	11
Rtag	0	0	1	1	0	0	0	1	1	1	0	1	1	1
Rchild	3	5	6	5	8	9	11	3	12	13	14	0	11	8

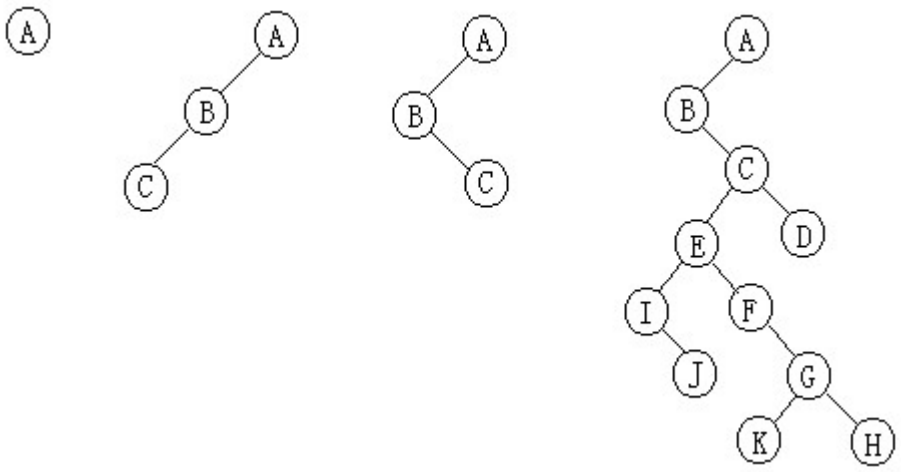
6.17 解： 其错误在于中序遍历应先访问其左子树，可做如下修改：

```
BiTree InSucc(BiTree q){
    // 一直 q 是指向中序线索二叉树上某个结点的指针，
    // 本函数返回指向*q 的后继的指针。
    r=q->rchild;
    if(!r->ltag)
        while(!r->ltag) r=r->lchild;
    return r;
} // InSucc
```

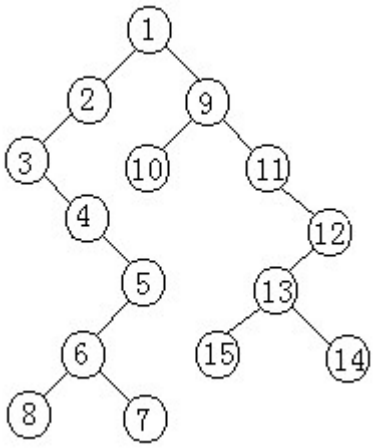
6.18 解： 如果 p 是根结点，则其后继为空。否则需查找 p 的双亲结点。从 p 结点开始中序线索遍历，如果某结点的左指针域等于 p，说明该结点是 p 的双亲结点，且 p 是它的左孩子；如果某结点的右指针域等于 p，说明该结点是 p 的双亲结点，且 p 是它的右孩子；如此即可确定访问次序。若是右孩子，其后继是双亲结点；若是左孩子，其后继是其兄弟最左下的子孙，如果兄弟不存在，其后继是其双亲结点。

6.19 分别画出和下列树对应的各个二叉树：

解：



6.20 解：



(1) 先序：1 2 3 4 5 6 8 7 9 10 11 12 13 15 14

(2) 中序：3 4 8 6 7 5 2 1 10 9 11 15 14 13 12

(3) 后序：8 7 6 5 4 3 2 10 15 14 13 12 11 9 1

6.23 解：树的先根序列为 GFKDAIEBCHJ，后根序列为 DIAEKFCJHBG，可以先转化成二叉树，再通过二叉树转换成树。注意二叉树的先根序列与等价树的先根序列相同，二叉树的中序序列对应着树的后根序列。

GFKDAIEBCHJ 为所求二叉树的先序序列，DIAEKFCJHBG 为二叉树的中序序列。通过观察先序序列，G 为二叉树的根结点，再由中序序列，G 的左子树序列为 DIAEKFCJHB，右子为空。可以表示成如下形式：

G(DIAEKFCJHB，NULL)

对于子树先序序列为 FKDAIEBCHJ，中序序列为 DIAEKFCJHB，显然子树根为 F。再由中序序列可以看到，F 的左子树是 DIAEK，右子树为 CJHB。进一步表示成：

G(F(DIAEK,CJHB),NULL)

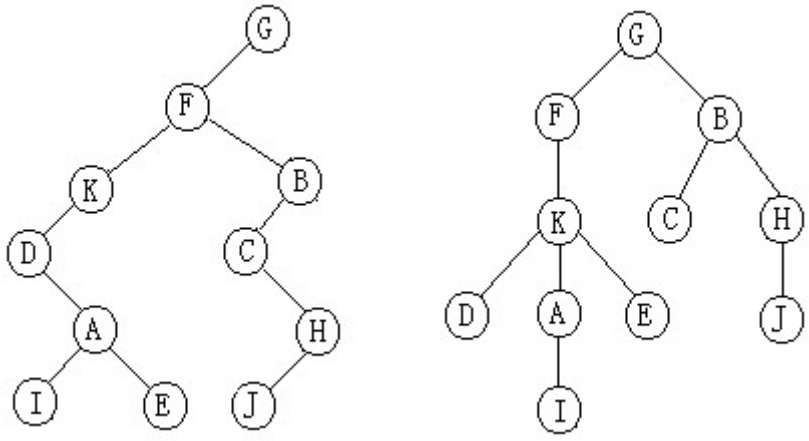
对于 DIAEK（中序表示），先序为 KDAIE，K 为根，左子为 DIAE，右子为空；对于 CJHB，B 为根，左子为 CJH，右子为空。进一步表示成：

G(F(K(DIAE,NULL),B(CJH,NULL)),NULL)

G(F(K(D(NULL,IAE),NULL),B(C(NULL,JH),NULL)),NULL)

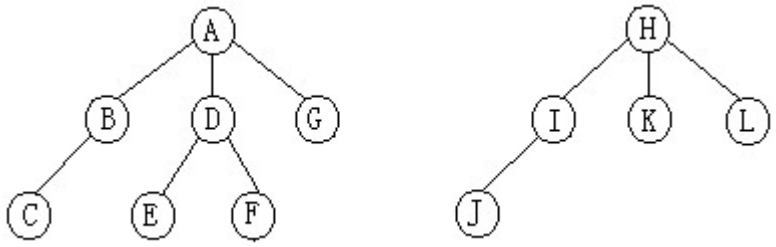
G(F(K(D(NULL,A(I,E)),NULL),B(C(NULL,H(J,NULL)),NULL)),NULL)

由此画出二叉树，进而画出树。



6.24 解：本题应注意下列转换：

树	森林	二叉树
先根	先序	先序
后根	中序	中序

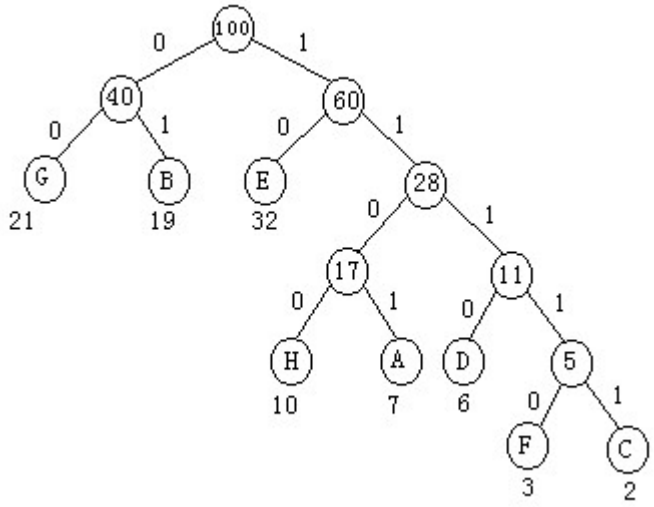


6.25 解：用归纳法证明。

当 n=2 时，要使其成为最优二叉树，必须使两个结点都成为叶子结点。

设 n=k 时成立，则当 n=k+1 时，要使其成为最优，必须用 k 个结点的哈夫曼树与第 k+1 个结点组成一个新的最优二叉树，所以 n=k+1 时也成立。

6.26 解：不妨设这 8 个结点为 A、B、C、D、E、F、G、H，其相应的权为 7、19、2、6、32、3、21、10。



A:1101 B:01 C:11111 D:1110 E:10 F:11110 G:00 H:1100

采用这种方式编码，电文最短。

6.27 解：

6.33 解：

```
int Visit(int u,int v)
{
    if(u==v) return 1;
    if(L[v]==0){//左子树不存在
        if(R[v]==0)//右子树也不存在
            return 0;
        else{// 右子树存在，继续访问右子树
            if(Visit(u,R[v])) return 1;
            else return 0;
        }
    }
```

```
    else{// 左子树存在
        if(Visit(u,L[v]))// 左子树中存在目标
            return 1;
        else{// 左子树中没有目标，需访问右子树
            if(R[v]==0)// 没有右子树
                return 0;
            else{// 右子树存在，继续访问右子树
                if(Visit(u,R[v])) return 1;
                else return 0;
            }
        }
    }
}
```

6.34 解：

```
int Visit(int u,int v)
{
    int Nv;
    Nv=NumElem(v); // 返回结点 v 的下标
    if(Nv==-1) exit(-2); // 结点 v 不存在，退出

    if(u==v) return 1;
    if(L[Nv]==0){//左子树不存在
        if(R[Nv]==0)//右子树也不存在
            return 0;
        else{// 右子树存在，继续访问右子树
            if(Visit(u,R[Nv])) return 1;
            else return 0;
        }
    }
    else{// 左子树存在
        if(Visit(u,L[Nv]))// 左子树中存在目标
            return 1;
        else{// 左子树中没有目标，需访问右子树
            if(R[Nv]==0)// 没有右子树
                return 0;
            else{// 右子树存在，继续访问右子树
                if(Visit(u,R[Nv])) return 1;
                else return 0;
            }
        }
    }
}
// 返回结点在数组 T 中的下标
int NumElem(int x)
```

```
{
    int i=0;
    while(i<N&&T[i]!=x) i++;
    if(T[i]==x) return i;
    else return -1;
}
```

6.35 解：

```
#define N 8
char T[N]={'0','a','b','c','d','e','f','g'};
// 用顺序数组存储，0 为头结点，
// a 为根结点，b 为左子树，c 为右子树，...

// 若某结点编号为奇数，
// 则它必为其双亲的右孩子，否则为左孩子
// 编号为 k 的结点的双亲结点的编号为 k/2

int NumTree(char c);// 求结点在树中的编号
int Exp(int a,int b);// 求 a 的 b 次方
int NumElem(char c);// 返回元素在数组中的下标

int main()
{
    char c;
    cout<<"请输入结点的值：";
    cin>>c;
    cout<<NumTree(c)<<endl;
    return 0;
}

int NumTree(char c)// 求结点在树中的编号
{
    int k,i=0;
    int Code[N];
    k=NumElem(c); // 返回结点 c 的下标
    if(k==-1) exit(-2); // 结点 c 不存在，退出

    while(k){
        // 如果 k 为偶数，记录一个代码 1
        if(k%2==1) Code[i]=1;
        else Code[i]=0;
        k=k/2;
        i++;
    }
    int x=0,j;
```

```
    for(j=0;j<i;j++)
        x=x+Code[j]*Exp(2,j);
    return x;
}

int Exp(int a,int b)
{
    int i;
    int x=1;
    for(i=0;i<b;i++)
        x=x*a;
    return x;
}

// 返回结点在数组 T 中的下标
int NumElem(char c)
{
    int i=0;
    while(i<N&&T[i]!=c) i++;
    if(T[i]==c) return i;
    else return -1;
}
```

6.36 解：

```
Status SimilarTree(BiTree& T1,BiTree& T2)
{
    if(!T1){// T1 是空树
        if(!T2) return TRUE;// T2 是空树
        else return FALSE;
    }
    else{// T1 是非空树
        if(!T2) return FALSE;
        else{// T2 是非空树
            if(SimilarTree(T1->lchild,T2->lchild)
                && SimilarTree(T1->rchild,T2->rchild))
                return TRUE;
            else return FALSE;
        }
    }
}
```

6.37 解：

```
// 先序遍历的非递归算法
Status POTraverse(BiTree& T,Status (*Visit)(TElemType e))
{
    BiTree p;
```

```
    Stack s;
    InitStack(s);
    p=T;
    while(p||!StackEmpty(s)){
        if(p){
            // 如果根指针不空，访问根结点，
            // 右指针域压栈，继续遍历左子树
            if(!Visit(p->data)) return ERROR;
            Push(s,p->rchild);
            p=p->lchild;
        } // 根指针已空，本子树已遍历完毕，
          // 退栈返回上一层，继续遍历未曾访问的结点
        else Pop(s,p);
    }
    return OK;
}
```

6.41 解：

```
// 求位于先序序列中第 k 个位置的结点的值，
// e 中存放结点的返回值，i 为计数器
Status PONodeK(TElemType& e,int& i,int k,BiTree& T)
{
    if(T){
        i++;
        if(i==k) e=T->data;
        else{
            PONodeK(e,i,k,T->lchild);
            PONodeK(e,i,k,T->rchild);
        }
    }
    return OK;
}
```

6.42 解：

```
// 求二叉树中叶子结点的数目
Status POLeafNodeNum(int& i,BiTree& T)
{
    if(T){
        if(!T->lchild && !T->rchild) i++;
        POLeafNodeNum(i,T->lchild);
        POLeafNodeNum(i,T->rchild);
    }
    return OK;
}
```

6.43 解：

```
// 按先序交换二叉树的左右子树
```

```
Status ExchangeBiTree(BiTree& T)
{
     BiTree p;
     if(T){
          p=T->lchild;
          T->lchild=T->rchild;
          T->rchild=p;
          ExchangeBiTree(T->lchild);
          ExchangeBiTree(T->rchild);
     }
     return OK;
}
```

6.44 解：

```
// 求二叉树中以元素值为 x 的结点为根的子树的深度
Status ChildTreeDepth(BiTree& T,TElemType x,int& depth)
{
     BiTree T1;
     if(PreOrderLocate(T,x,T1)){
          depth=BiTDepth(T1);
          return OK;
     }
     else return ERROR;
}

// 按先序在树中查找根为 x 的子树，T1 为指向子树根的指针
Status PreOrderLocate(BiTree& T,TElemType x,BiTree& T1)
{
     if(T){
          if(T->data==x){
               T1=T;
               return OK;
          }
          else{
               if(PreOrderLocate(T->lchild,x,T1))
                    return OK;
               else{
                    if(PreOrderLocate(T->rchild,x,T1))
                         return OK;
                    else return ERROR;
               }
          }
     }
     else return ERROR;
}
```

```
// 求二叉树的深度
int BiTDepth(BiTree& T)
{
     int ldep,rdep;
     if(!T) return 0;
     else{
          ldep=BiTDepth(T->lchild)+1;
          rdep=BiTDepth(T->rchild)+1;
          return ldep>rdep?ldep:rdep;
     }
}
```

6.45 解：

```
// 删除以元素值为 x 的结点为根的子树
Status DelChildTree(BiTree& T,TElemType x)
{
     if(T){
          if(T->data==x){
               DelBTree(T);
               T=NULL;
               return OK;
          }
          else{
               if(DelChildTree(T->lchild,x))
                    return OK;
               else{
                    if(DelChildTree(T->rchild,x))
                         return OK;
                    else return ERROR;
               }
          }
     }
     else return ERROR;
}
// 删除二叉树
Status DelBTree(BiTree& T)
{
     if(T){
          DelBTree(T->lchild);
          DelBTree(T->rchild);
          delete T;
          return OK;
     }
     else return ERROR;
```

```
}
```

6.46 解：

```
// 复制一棵二叉树
Status CopyBiTree(BiTree& T,BiTree& T1)
{
     BiTree p;
     if(T){
          p=new BiTNode;
          if(!p) return ERROR;
          p->data=T->data;
          T1=p;
          CopyBiTree(T->lchild,T1->lchild);
          CopyBiTree(T->rchild,T1->rchild);
     }
     else{
          T1=T;
     }
     return OK;
}
```

6.47 解：

```
typedef BiTree QElemType;
#include "c:\Yin\include\Queue.h"
Status LevelOrderTraverse(BiTree& T,Status (*Visit)(TElemType e))
{
     QElemType p;
     Queue q;
     InitQueue(q);

     if(T) EnQueue(q,T);
     while(!QueueEmpty(q)){
          DeQueue(q,p);
          Visit(p->data);
          if(p->lchild) EnQueue(q,p->lchild);
          if(p->rchild) EnQueue(q,p->rchild);
     }
     return OK;
}
```

6.48 解：

```
// 在二叉树 T 中求结点*p 和*q 的共同最小祖先 e
Status MinComAncst(BiTree& T,TElemType& e,TElemType *p,TElemType *q)
{
     if(!T) return ERROR;
     BiTree T1,T2,pt=NULL;
     if(!CopyBiTree(T,T1)) return ERROR;
```

```
    if(!CopyBiTree(T,T2)) return ERROR;
    if(!PathTree(T1,p))
        return ERROR;// 求根结点到结点 p 的路径树 T1
    else ShowBiTree(T1);
    cout<<endl;
    if(!PathTree(T2,q))
        return ERROR;// 求根结点到结点 q 的路径树 T2
    else ShowBiTree(T2);
    cout<<endl;
    while(T1 && T2 && T1->data==T2->data){
        pt=T1;
        if(T1->lchild){
            T1=T1->lchild;
            T2=T2->lchild;
        }
        else{
            if(T1->rchild){
                T1=T1->rchild;
                T2=T2->rchild;
            }
        }
    }
    if(!pt) return ERROR;
    else{
        e=pt->data;
        return OK;
    }
}
// 在二叉树 T 中求根到结点 p 的路径树，该操作将剪去除路径之外的所有分支
Status PathTree(BiTree& T,TElemType *p)
{
    if(!T || !p) return ERROR;
    if(T->data==*p){// 找到目标，删除目标的左右子树
        if(T->lchild) DelBiTree(T->lchild);
        if(T->rchild) DelBiTree(T->rchild);
        return OK;
    }
    else{// 没找到目标，继续递归查找
        if(PathTree(T->lchild,p)){// 目标在左子树中，删除右子树
            if(T->rchild) DelBiTree(T->rchild);
            return OK;
        }
        else
            if(PathTree(T->rchild,p)){// 目标在右子树中，删除左子树
```

```
                if(T->lchild) DelBiTree(T->lchild);
                return OK;
            }
            else return ERROR;// 找不到目标
    }
}
```

6.49 解：

```
Status CompleteBiTree(BiTree& T)
{
    int d;
    if(T){
        d=BiTDepth(T->lchild)-BiTDepth(T->rchild);
        if(d<0 || d>1) return ERROR;
        else{
            if(CompleteBiTree(T->lchild) &&
                CompleteBiTree(T->rchild)) return OK;
            else return ERROR;
        }
    }
    else return OK;
}
```

6.51 解：

```
Status ShowBiTExpress(BiTree& T)
{
    if(T){
        if(T->lchild){
            if(Low(T->lchild->data,T->data)){
                cout<<'(';
                ShowBiTExpress(T->lchild);
                cout<<')';
            }
            else ShowBiTExpress(T->lchild);
        }
        cout<<T->data;
        if(T->rchild){
            if(Low(T->rchild->data,T->data)){
                cout<<'(';
                ShowBiTExpress(T->rchild);
                cout<<')';
            }
            else ShowBiTExpress(T->rchild);
        }
    }
    return OK;
```

```
}

Status Low(char a,char b)
{
     if((a=='+' || a=='-') && (b=='*' || b=='/')) return TRUE;
     else return FALSE;
}
```

6.52 解：

```
int BiTreeThrive(BiTree& T)
{
     int i,d,nn[20];
     d=BiTDepth(T);
     BiTree p=T;
     Stack s1,s2;
     InitStack(s1);InitStack(s2);
     for(i=0;i<20;i++){
          nn[i]=0; // 每层结点个数
     }

     if(p) Push(s1,p);
     else return 0;
     for(i=0;i<d;i++){
          if(!StackEmpty(s1) && StackEmpty(s2)){
               while(!StackEmpty(s1)){
                    Pop(s1,p);    nn[i]++;// s1 中存放第 i 层的结点
                    if(p->lchild) Push(s2,p->lchild);//s2 中存放第 i+1 层结点
                    if(p->rchild) Push(s2,p->rchild);
               }
          }
          else{
               if(StackEmpty(s1) && !StackEmpty(s2)){
                    while(!StackEmpty(s2)){
                         Pop(s2,p);    nn[i]++;
                         if(p->lchild) Push(s1,p->lchild);
                         if(p->rchild) Push(s1,p->rchild);
                    }
               }
          }
     }
     int max=nn[0];
     for(i=0;i<d;i++)
          if(max<nn[i]) max=nn[i];
     return max*d;
}
```

6.53 解：

```
// 所有从根到叶子最长路径树
Status MaxPathBiTree(BiTree& T)
{
     if(T){
          if(BiTDepth(T)-BiTDepth(T->lchild)!=1)
               DelBiTree(T->lchild);
          else
               MaxPathBiTree(T->lchild);
          if(BiTDepth(T)-BiTDepth(T->rchild)!=1)
               DelBiTree(T->rchild);
          else
               MaxPathBiTree(T->rchild);
     }
     return OK;
}
// 从根到叶子最长路径中最左方的路径树
Status LMaxPathBiTree(BiTree& T)
{
     if(T){
          if(BiTDepth(T)-BiTDepth(T->lchild)==1){
               DelBiTree(T->rchild);
               LMaxPathBiTree(T->lchild);
          }
          else{
               DelBiTree(T->lchild);
               if(BiTDepth(T)-BiTDepth(T->rchild)==1)
                    LMaxPathBiTree(T->rchild);
               else
                    DelBiTree(T->rchild);
          }
     }
     return OK;
}
```

6.54 解：

```
// 根据完全二叉顺序树创建完全二叉链表树
Status CreateCompleteBiTree(SqList<TElemType>& ST,BiTree& LT)
{
     BiTree p;
     int i=0,len;
     if(ST.Length==0) return OK;
     p=new BiTNode;
     if(!p) return ERROR;
     p->data=ST.Get(i);
```

```
    p->lchild=NULL;
    p->rchild=NULL;
    LT=p;

    Queue q;
    InitQueue(q);
    EnQueue(q,p);
    len=ST.Length();

    while(!QueueEmpty(q)&&i<len-1){
        DeQueue(q,p);
        if(i<len-1 && i%2==0){
            p->lchild=new BiTNode;
            if(!p->lchild) return ERROR;
            p->lchild->data=ST.Get(++i);
            p->lchild->lchild=NULL;
            p->lchild->rchild=NULL;
            EnQueue(q,p->lchild);
        }
        if(i<len-1 && i%2==1){
            p->rchild=new BiTNode;
            if(!p->rchild) return ERROR;
            p->rchild->data=ST.Get(++i);
            p->rchild->lchild=NULL;
            p->rchild->rchild=NULL;
            EnQueue(q,p->rchild);
        }
    }
    return OK;
}
```

6.55 解：

```
Status PreOrderTraverse(BiTree& T)
{
    if(T){
        T->DescNum=DescendNum(T);
        PreOrderTraverse(T->lchild);
        PreOrderTraverse(T->rchild);
    }
    return OK;
}

int DescendNum(BiTree& T)
{
    if(!T) return 0;
```

```
    if(!T->lchild){
        if(!T->rchild) return 0;
        else return DescendNum(T->rchild)+1;
    }
    else{
        if(!T->rchild) return DescendNum(T->lchild)+1;
        else return DescendNum(T->rchild)+DescendNum(T->lchild)+2;
    }
}
```

6.56 解：

先对二叉树 T 进行先序线索，得到先序线索二叉树 Thrt。然后再进行查找。

```
// 先序线索二叉树算法
Status PreOrderThreading(BiThrTree& Thrt,BiThrTree& T)
{
    BiThrTree pre;
    Thrt=new BiThrNode; // 为线索二叉树建立头结点
    if(!Thrt) exit(OVERFLOW);
    Thrt->LTag=Thread;
    Thrt->RTag=Link;
    Thrt->lchild=Thrt;// 左子树回指
    if(!T) Thrt->rchild=Thrt;// 若二叉树空，右子树回指
    else{
        Thrt->rchild=T;
        pre=Thrt;
        PreThreading(T,pre); // 先序遍历进行先序线索化
        pre->rchild=Thrt; // 最后一个结点线索化
        pre->RTag=Thread;
        Thrt->lchild=pre;
    }
    return OK;
}

Status PreThreading(BiThrTree& T,BiThrTree& pre)
{
    if(T){
        if(!T->lchild){
            T->LTag=Thread;
            T->lchild=pre;
        }
        if(pre && !pre->rchild){
            pre->RTag=Thread;
            pre->rchild=T;
        }
        pre=T;
```

```
        if(T->LTag==Link) PreThreading(T->lchild,pre);
        if(T->RTag==Link) PreThreading(T->rchild,pre);
    }
    return OK;
}

// 从二叉线索树上任一结点 q 开始查找结点*p。
// 如果找到，将*p 的后继结点指针存于 q 中，返回 TRUE；否则返回 FALSE
Status FindNextInBiThrTree(BiThrTree& q,TElemType *p)
{
    BiThrTree pt=q;
    if(!pt) return FALSE;
    if(pt->data==*p){
        if(pt->LTag==Link) q=pt->lchild;
        else q=pt->rchild;
        return OK;
    }
    pt=q->rchild;
    while(pt!=q && pt->data!=*p){
        if(pt->LTag==Link) pt=pt->lchild;
        else pt=pt->rchild;
    }
    if(pt==q) return FALSE;
    if(pt->data==*p){
        if(pt->LTag==Link) q=pt->lchild;
        else q=pt->rchild;
    }
    return OK;
}
```

6.57 解：

```
Status PostOrderThreading(BiThrTree& T,BiThrTree& pre);//首先建立后序线索树
Status FindNextInBiThrTree(BiThrTree& q,TElemType *p);//再进行查找

// 后序线索二叉树的算法
Status PostOrderThreading(BiThrTree& Thrt,BiThrTree& T)
{
    BiThrTree pre;
    Thrt=new BiThrNode; // 为线索二叉树建立头结点
    if(!Thrt) exit(OVERFLOW);
    Thrt->LTag=Link;
    Thrt->RTag=Thread;
    Thrt->rchild=Thrt;// 右子树回指
    if(!T) Thrt->lchild=Thrt;// 若二叉树空，左子树回指
    else{
```

```
        Thrt->lchild=T;
        pre=Thrt;
        PostThreading(T,pre); // 后序遍历进行后序线索化
        pre->rchild=Thrt;// 最后一个结点线索化
        pre->RTag=Thread;
        Thrt->rchild=pre;
    }
    return OK;
}

Status PostThreading(BiThrTree& T,BiThrTree& pre)
{
    if(T){
        if(T->LTag==Link) PostThreading(T->lchild,pre);
        if(T->RTag==Link) PostThreading(T->rchild,pre);
        if(!T->lchild){
            T->LTag=Thread;
            T->lchild=pre;
        }
        if(pre && !pre->rchild){
            pre->RTag=Thread;
            pre->rchild=T;
        }
        pre=T;
    }
    return OK;
}
```

6.58 解：

```
typedef char TElemType;
typedef struct CSNode{
    TElemType data;
    struct CSNode *firstchild,*nextsibling;
}CSNode,*CSTree;
// 建立树的二叉链表表示
Status CreateTree(CSTree& T)
{
    char ch;
    cout<<"输入结点的值(一个字符，'@'表示空树)";
    cin>>ch;
    if(ch=='@'){
        T=NULL;
    }
    else{
        T=new CSNode;
```

```
        if(!T) return ERROR;
        T->data=ch;
        CreateTree(T->firstchild);
        CreateTree(T->nextsibling);
    }
    return OK;
}
// 输出树的各边
Status ShowTree(CSTree& T,CSTree& Father)
{
    if(T && Father)
        cout<<"("<<Father->data<<","<<T->data<<")";
    if(T->firstchild) ShowTree(T->firstchild,T);
    if(T->nextsibling) ShowTree(T->nextsibling,Father);
    return OK;
}
```

6.60 解：

```
int LeafNum(CSTree& T)
{
    if(T){
        if(!T->firstchild)
            return 1+LeafNum(T->nextsibling);
        else
            return LeafNum(T->firstchild)+LeafNum(T->nextsibling);
    }
    else return 0;
}
```

6.61 解：

```
int DegreeNum(CSTree& T)
{
    int d,dl,dr;
    if(T){
        if(!T->firstchild) d=0;
        else d=1+RSiblingNum(T->firstchild);
        dl=DegreeNum(T->firstchild);
        dr=DegreeNum(T->nextsibling);
        return Max(d,dl,dr); // 三数中求最大者
    }
    else return 0;
}
// 返回当前结点的兄弟数
int RSiblingNum(CSTree& T)
{
    int i=0;
```

```
    while(T->nextsibling){
        i++;
        T=T->nextsibling;
    }
    return i;
}
```

6.62 解：

```
// 树的深度
int Depth(CSTree& T)
{
    int d1,d2;
    if(T){
        d1=1+Depth(T->firstchild);
        d2=Depth(T->nextsibling);
        return d1>d2?d1:d2;
    }
    else return 0;
}
```

第 7 章　图

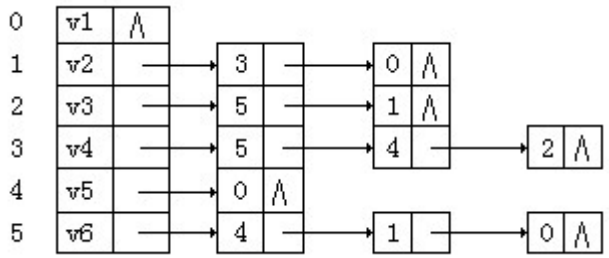
7.1 解： (1)

	ID(1)=3	OD(1)=0
	ID(2)=2	OD(2)=2
	ID(3)=1	OD(3)=2
	ID(4)=1	OD(4)=3
	ID(5)=2	OD(5)=1
	ID(6)=2	OD(6)=3

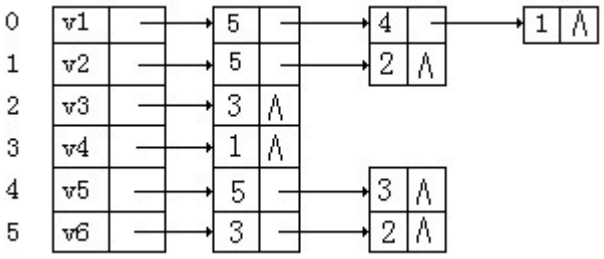
(2)

0	0	0	0	0	0
1	0	0	1	0	0
0	1	0	0	0	1
0	0	1	0	1	1
1	0	0	0	0	0
1	1	0	0	1	0

(3)



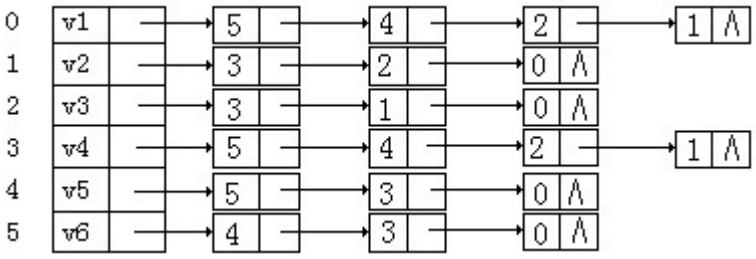
(4)



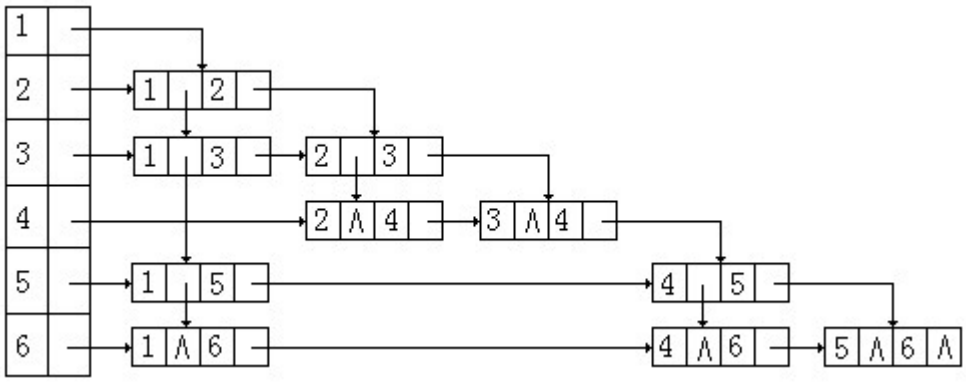
(5) 有三个连通分量 1、5、2346

7.2 解：k=1，说明了各结点之间的相互连通关系；k=2 说明了结点之间按路径长度为 2 的相互连通关系；...。

7.3 解：邻接表：



邻接多重表：



深度优先搜索的顺序为 1 5 6 4 3 2

广度优先搜索的顺序为 1 5 6 3 2 4，　　15 16 13 12 24

7.14 解：

```
Status CreateAG(ALGraph &G)
{
    int n,e,k,i,j;
    cout<<"请输入顶点数：";
    cin>>n;
    cout<<"请输入边数：";
    cin>>e;
    G.vernum=n;
    G.arcnum=e;

    // 建立顶点数组
    for(k=0;k<G.vernum;k++){
```

```
            cout<<"请输入顶点信息：";
            cin>>G.vertices[k].data;
            G.vertices[k].firstarc=NULL;
        }
        // 建立邻接表
        VertexType v1,v2;
        ArcNode *p,*q;
        for(k=0;k<G.arcnum;k++){
            cout<<"请输入弧的始点和终点信息，中间用空格分开:";
            cin>>v1>>v2;
            i=LocateVex(G,v1);
            if(i<0 || i>G.vernum-1) return ERROR;
            j=LocateVex(G,v2);
            if(j<0 || j>G.vernum-1) return ERROR;
            if(i==j) return ERROR;
            p=new ArcNode;
            if(!p) return ERROR;
            p->adjvex=j;
            p->nextarc=NULL;
            q=G.vertices[i].firstarc;
            if(!q) G.vertices[i].firstarc=p;
            else{
                while(q->nextarc) q=q->nextarc;  // 指针定位于邻接表的尾结点
                q->nextarc=p;
            }
        }
        return OK;
}
int LocateVex(ALGraph& G,VertexType v)
{
        int i=0;
        while(G.vertices[i].data!=v&&i<G.vernum) i++;
        if(G.vertices[i].data==v) return i;
        else return -1;
}
```

7.15 解

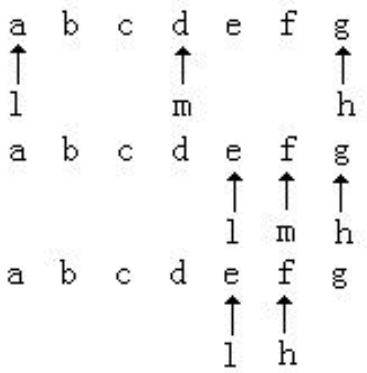
第 9 章　查找

9.1 解：(1)相同，平均查找长度为$\frac{3}{4}(n+1)$

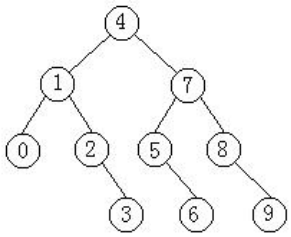
(2)相同，平均查找长度为$\frac{1}{2}(n+1)$

(3)对于有序顺序表，平均查找长度为$\frac{n-k+1}{2}$，对于无序顺序表，则为$\frac{1}{2}(n+1)$。

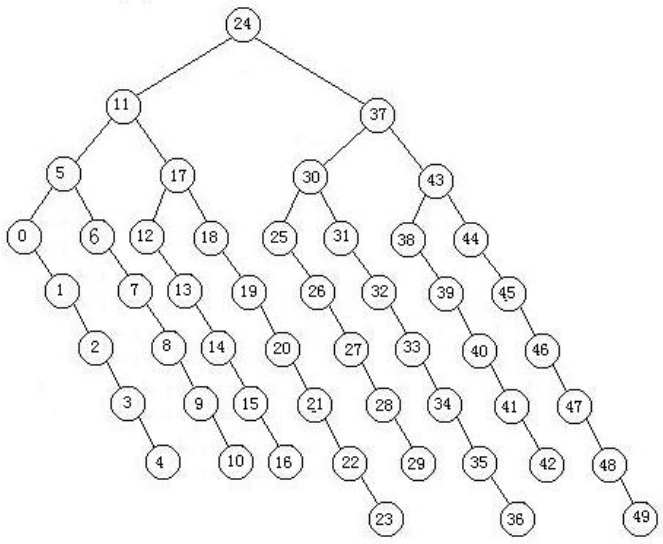
9.2 解：查找 e 的过程如下：



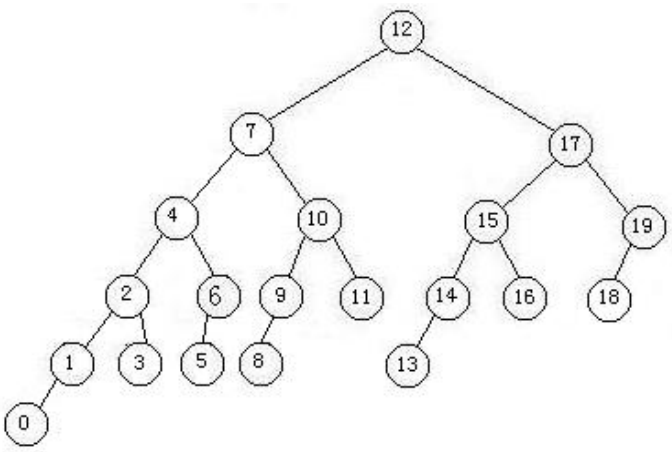
9.3 解：$ASL=\frac{1}{10}[1\times1+2\times2+4\times3+3\times4]=2.9$



9.4 解：$ASL=\frac{1}{50}[1+2\times2+4\times3+8\times(4+5+6+7+8)+9\times3]=5.68$



9.5 解：$ASL=\frac{1}{20}[1+2\times2+4\times3+7\times4+5\times5+6]=3.8$

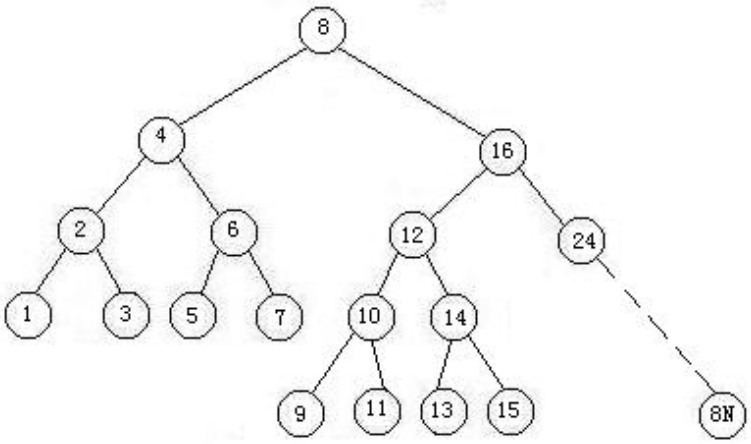


9.7 解：$a_1 = 1\times1+1\times2+2\times3+4\times4$

$$a_1 = 1\times1+1\times2+2\times3+4\times4$$

$$a_2 = 1\times2+1\times3+2\times4+4\times5$$

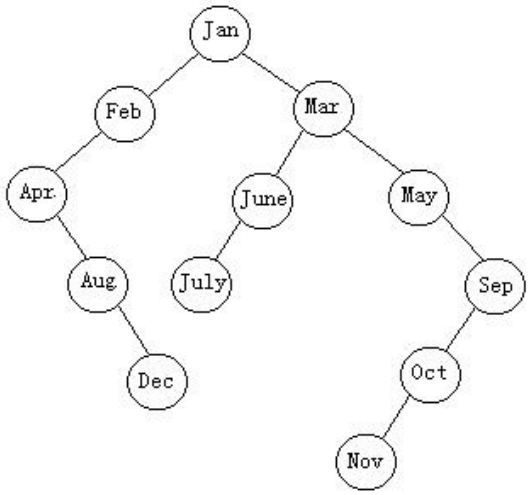
$$a_n = 1\times n+1\times(n+1)+2\times(n+2)+4\times(n+3)$$



$$s_N = \sum_{n=1}^{N} a_n = \sum_{n=1}^{N}[8N+17] = \frac{8N(N+1)}{2}+17N$$

$$ASL = \frac{N+1}{2}+\frac{17}{8}$$

9.9 解：(1) $ASL = \frac{1}{12}[1\times1+2\times2+3\times3+3\times4+2\times5+1\times6] = 3.5$



(3)