```
    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。

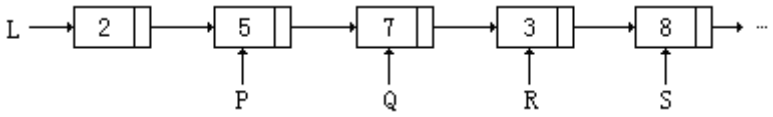
(3) 在单链表中，除了首元结点外，任一结点的存储位置由*其前驱结点的链域的值*指示。

(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？

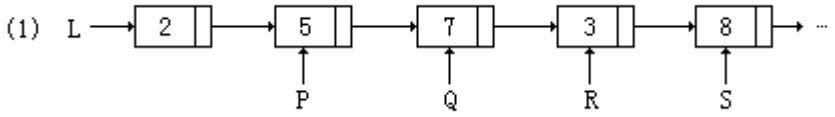
解：当线性表的数据元素在物理位置上是连续存储的时候，用顺序表比用链表好，其特点是可以进行随机存取。

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。

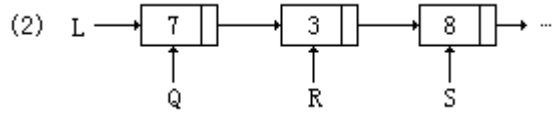


解：

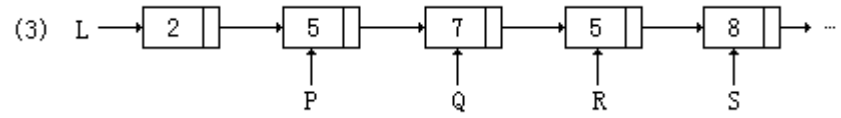
(1)

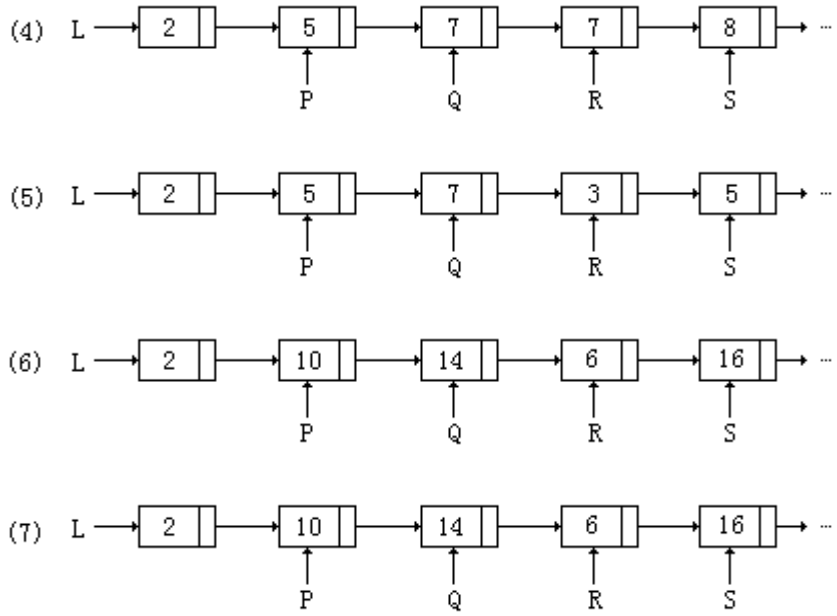


(2)



(3)

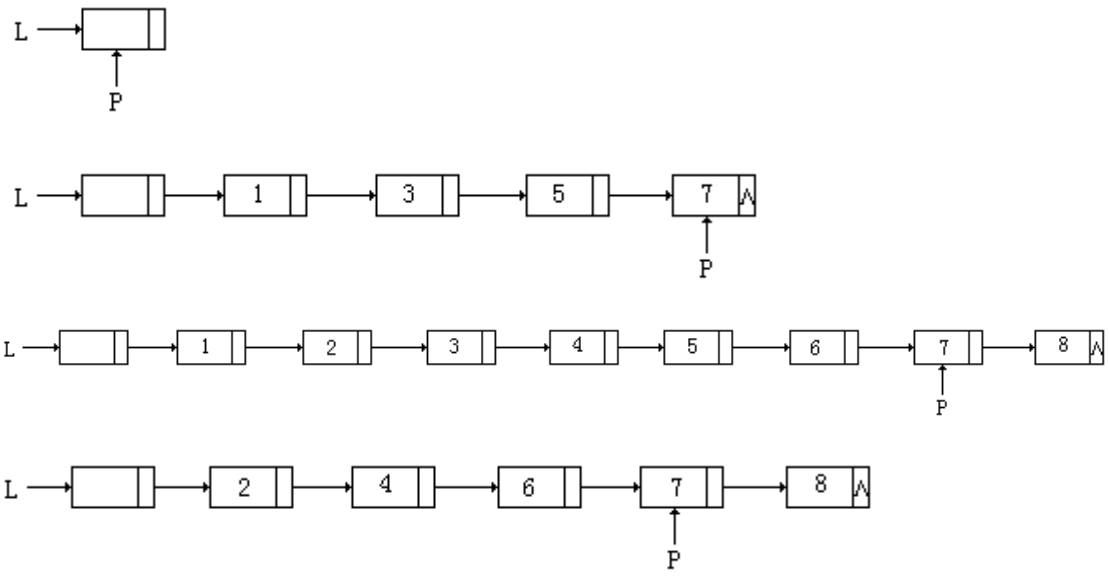




2.5 画出执行下列各行语句后各指针及链表的示意图。

```
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;
}
```