第 2 章　线性表

2．算法设计题

（1）将两个递增的有序链表合并为一个递增的有序链表。要求结果链表仍使用原来两个链表的存储空间，不另外占用其它的存储空间。表中不允许有重复的数据。

```
void MergeList_L(LinkList &La,LinkList &Lb,LinkList &Lc){
   pa=La->next;  pb=Lb->next;
   Lc=pc=La;              //用 La 的头结点作为 Lc 的头结点
   while(pa && pb){
      if(pa->data<pb->data){ pc->next=pa;pc=pa;pa=pa->next;}
      else if(pa->data>pb->data) {pc->next=pb; pc=pb; pb=pb->next;}
      else {// 相等时取 La 的元素，删除 Lb 的元素
               pc->next=pa;pc=pa;pa=pa->next;
               q=pb->next;delete pb ;pb =q;}
   }
   pc->next=pa?pa:pb;     //插入剩余段
   delete Lb;              //释放 Lb 的头结点}
```

（2）将两个非递减的有序链表合并为一个非递增的有序链表。要求结果链表仍使用原来两个链表的存储空间，不另外占用其它的存储空间。表中允许有重复的数据。

```
void union(LinkList& La, LinkList& Lb, LinkList& Lc, ) {
  pa = La->next;  pb = Lb->next;              // 初始化
  Lc=pc=La; //用 La 的头结点作为 Lc 的头结点
  Lc->next = NULL;
  while ( pa || pb ) {
     if  ( !pa )  { q = pb;  pb = pb->next; }
     else if  ( !pb )  { q = pa;  pa = pa->next; }
     else if (pa->data <= pb->data )      { q = pa;  pa = pa->next; }
     else  { q = pb;  pb = pb->next; }
     q->next = Lc->next;  Lc->next = q;     // 插入
  }
  delete Lb;              //释放 Lb 的头结点}
```

（3）已知两个链表 A 和 B 分别表示两个集合，其元素递增排列。请设计算法求出 A 与 B 的交集，并存放于 A 链表中。

```
void Mix(LinkList& La, LinkList& Lb, LinkList& Lc, ) {
    pa=la->next;pb=lb->next;//设工作指针 pa 和 pb;
    Lc=pc=La; //用 La 的头结点作为 Lc 的头结点
    while(pa&&pb)
      if(pa->data==pb->data)//交集并入结果表中。
       { pc->next=pa;pc=pa;pa=pa->next;
         u=pb;pb=pb->next; delete u;}
      else if(pa->data<pb->data) {u=pa;pa=pa->next; delete u;}
```

```
        else {u=pb; pb=pb->next; delete u;}
    while(pa){ u=pa; pa=pa->next; delete u;}// 释放结点空间
    while(pb) {u=pb; pb=pb->next; delete u;}//释放结点空间
    pc->next=null;//置链表尾标记。
    delete Lb;   //注:  本算法中也可对B表不作释放空间的处理
```

(4) 已知两个链表A和B分别表示两个集合，其元素递增排列。请设计算法求出两个集合A和B 的差集（即仅由在A中出现而不在B中出现的元素所构成的集合），并以同样的形式存储，同时返回该集合的元素个数。

```
 void  Difference (LinkedList  A, B, *n)
 //A和B均是带头结点的递增有序的单链表，分别存储了一个集合，本算法求两集合的差集，存储于单链表A中，*n是结果集合中元素个数，调用时为0
{p=A->next;                    //p和q分别是链表A和B的工作指针。
 q=B->next;  pre=A;            //pre为A中p所指结点的前驱结点的指针。
 while (p!=null && q!=null)
   if (p->data<q->data) {pre=p; p=p->next; *n++; } // A链表中当前结点指针后移。
   else if (p->data>q->data) q=q->next;       //B链表中当前结点指针后移。
   else {pre->next=p->next;           //处理A，B中元素值相同的结点，应删除。
         u=p;  p=p->next;  delete u; }   //删除结点
```

(5) 设计算法将一个带头结点的单链表A分解为两个具有相同结构的链表B、C，其中B表的结点为A表中值小于零的结点，而C表的结点为A表中值大于零的结点（链表A的元素类型为整型，要求B、C表利用A表的结点）。

(6) 设计一个算法，通过一趟遍历在单链表中确定值最大的结点。

```
ElemType Max (LinkList L ){
    if(L->next==NULL) return NULL;
    pmax=L->next; //假定第一个结点中数据具有最大值
    p=L->next->next;
    while(p != NULL ){//如果下一个结点存在
        if(p->data > pmax->data) pmax=p;
        p=p->next;
    }
    return pmax->data;
```

(7) 设计一个算法，通过遍历一趟，将链表中所有结点的链接方向逆转，仍利用原表的存储空间。

```
void  inverse(LinkList &L) {
    // 逆置带头结点的单链表 L
    p=L->next;  L->next=NULL;
    while ( p) {
        q=p->next;     // q指向*p的后继
        p->next=L->next;
        L->next=p;        // *p插入在头结点之后
        p = q;
```

```
    }
}
```

（8）设计一个算法，删除递增有序链表中值大于 mink 且小于 maxk 的所有元素（mink 和 maxk 是给定的两个参数，其值可以和表中的元素相同，也可以不同 ）。

```
void delete(LinkList &L, int mink, int maxk) {
   p=L->next; //首元结点
   while (p && p->data<=mink)
      { pre=p;  p=p->next; } //查找第一个值>mink 的结点
   if (p) {
       while (p && p->data<maxk)  p=p->next;
                        // 查找第一个值 ≥maxk 的结点
      q=pre->next;   pre->next=p;  // 修改指针
      while (q!=p)
         { s=q->next;  delete q;  q=s; } // 释放结点空间
   }//if
}
```

（9）已知 p 指向双向循环链表中的一个结点，其结点结构为 data、prior、next 三个域，写出算法 change(p)，交换 p 所指向的结点和它的前缀结点的顺序。

知道双向循环链表中的一个结点，与前驱交换涉及到四个结点（p 结点，前驱结点，前驱的前驱结点，后继结点）六条链。

```
void  Exchange (LinkedList p)
//p 是双向循环链表中的一个结点，本算法将 p 所指结点与其前驱结点交换。
{q=p->llink;
 q->llink->rlink=p;     //p 的前驱的前驱之后继为 p
 p->llink=q->llink;     //p 的前驱指向其前驱的前驱。
 q->rlink=p->rlink;     //p 的前驱的后继为 p 的后继。
 q->llink=p;            //p 与其前驱交换
 p->rlink->llink=q;     //p 的后继的前驱指向原 p 的前驱
 p->rlink=q;            //p 的后继指向其原来的前驱
}//算法 exchange 结束。
```

（10）已知长度为 n 的线性表 A 采用顺序存储结构，请写一时间复杂度为 O(n)、空间复杂度为 O(1)的算法，该算法删除线性表中所有值为 item 的数据元素。

[题目分析] 在顺序存储的线性表上删除元素，通常要涉及到一系列元素的移动（删第 i 个元素，第 i+1 至第 n 个元素要依次前移）。本题要求删除线性表中所有值为 item 的数据元素，并未要求元素间的相对位置不变。因此可以考虑设头尾两个指针（i=1，j=n），从两端向中间移动，凡遇到值 item 的数据元素时，直接将右端元素左移至值为 item 的数据元素位置。

```
void  Delete (ElemType A[ ], int  n)
//A 是有 n 个元素的一维数组，本算法删除 A 中所有值为 item 的元素。
{i=1; j=n; //设置数组低、高端指针（下标）。
 while (i<j)
   {while (i<j && A[i]!=item) i++;          //若值不为 item，左移指针。
    if (i<j) while (i<j && A[j]==item) j--; //若右端元素值为 item，指针左移
```

```
        if (i<j) A[i++]=A[j--];
      }
```

[算法讨论] 因元素只扫描一趟，算法时间复杂度为 O（n）。删除元素未使用其它辅助空间，最后线性表中的元素个数是 j。

第 3 章　栈和队列

（2）回文是指正读反读均相同的字符序列，如“abba”和“abdba”均是回文，但“good”不是回文。试写一个算法判定给定的字符向量是否为回文。(提示：将一半字符入栈)

```
   根据提示，算法可设计为：
//以下为顺序栈的存储结构定义
#define StackSize 100 //假定预分配的栈空间最多为 100 个元素
typedef char DataType;//假定栈元素的数据类型为字符
typedef struct{
 DataType data[StackSize];
 int top;
}SeqStack;

int IsHuiwen( char *t)
 {//判断 t 字符向量是否为回文，若是，返回 1，否则返回 0
  SeqStack s;
  int i , len;
  char temp;
  InitStack( &s);
  len=strlen(t); //求向量长度
  for ( i=0; i<len/2; i++)//将一半字符入栈
   Push( &s, t[i]);
  while( !EmptyStack( &s))
   {// 每弹出一个字符与相应字符比较
    temp=Pop (&s);
    if( temp!=S[i])  return 0 ;// 不等则返回 0
```

```
    else i++;
   }
  return 1 ; // 比较完毕均相等则返回 1
 }
```

（3）设从键盘输入一整数的序列：a_1, a_2, a_3，…，a_n，试编写算法实现：用栈结构存储输入的整数，当 $a_i \neq -1$ 时，将 a_i 进栈；当 $a_i=-1$ 时，输出栈顶整数并出栈。算法应对异常情况（入栈满等）给出相应的信息。

```
#define maxsize 栈空间容量
   void InOutS(int s[maxsize])
     //s 是元素为整数的栈，本算法进行入栈和退栈操作。
   {int top=0;                 //top 为栈顶指针，定义 top=0 时为栈空。
    for(i=1; i<=n; i++)      //n 个整数序列作处理。
     {scanf("%d",&x);     //从键盘读入整数序列。
      if(x!=-1)               // 读入的整数不等于-1 时入栈。
       if(top==maxsize-1){printf("栈满\n");exit(0);}else s[++top]=x; //x 入栈。
       else   //读入的整数等于-1 时退栈。
       {if(top==0){printf("栈空\n");exit(0);}  else  printf("出栈元素
       是%d\n",s[top--]); }}
         }//算法结束。
```

（4）从键盘上输入一个后缀表达式，试编写算法计算表达式的值。规定：逆波兰表达式的长度不超过一行，以$符作为输入结束，操作数之间用空格分隔，操作符只可能有+、-、*、/四种运算。例如：234 34+2*$。

[题目分析]逆波兰表达式(即后缀表达式)求值规则如下：设立运算数栈 OPND，对表达式从左到右扫描(读入)，当表达式中扫描到数时，压入 OPND 栈。当扫描到运算符时，从 OPND 退出两个数，进行相应运算，结果再压入 OPND 栈。这个过程一直进行到读出表达式结束符$，这时 OPND 栈中只有一个数，就是结果。

```
      float expr( )
   //从键盘输入逆波兰表达式，以'$'表示输入结束，本算法求逆波兰式表达式的值。
    {float OPND[30];   // OPND 是操作数栈。
    init(OPND);        //两栈初始化。
     float num=0.0;    //数字初始化。
     scanf ("%c",&x);//x 是字符型变量。
     while(x!='$')
      {switch
         {case'0'<=x<='9':while((x>='0'&&x<='9')||x=='.')  //拼数
                                  if(x!='.')   //处理整数
                                   {num=num*10+ (ord(x)-ord('0')) ; scanf("%c",&x);}
                                  else              //处理小数部分。
                                   {scale=10.0; scanf("%c",&x);
                                    while(x>='0'&&x<='9')
```

```
                    {num=num+(ord(x)-ord('0')/scale;
                     scale=scale*10;  scanf("%c",&x); }
                  }//else
                  push(OPND,num); num=0.0;//数压入栈, 下个数初始
化
          case x=' ':break;  //遇空格, 继续读下一个字符。
          case x='+':push(OPND,pop(OPND)+pop(OPND));break;
          case x='-':x1=pop(OPND);x2=pop(OPND);push(OPND,x2-x1);break;
          case x='*':push(OPND,pop(OPND)*pop(OPND));break;
          case x='/':x1=pop(OPND);x2=pop(OPND);push(OPND,x2/x1);break;
          default:        //其它符号不作处理。
        }//结束 switch
        scanf("%c",&x);//读入表达式中下一个字符。
      }//结束 while (x!='$')
     printf("后缀表达式的值为%f",pop(OPND));
  }//算法结束。
```

[算法讨论]假设输入的后缀表达式是正确的，未作错误检查。算法中拼数部分是核心。若遇到大于等于'0'且小于等于'9'的字符，认为是数。这种字符的序号减去字符'0'的序号得出数。对于整数，每读入一个数字字符，前面得到的部分数要乘上 10 再加新读入的数得到新的部分数。当读到小数点，认为数的整数部分已完，要接着处理小数部分。小数部分的数要除以 10（或 10 的幂数）变成十分位，百分位，千分位数等等，与前面部分数相加。在拼数过程中，若遇非数字字符，表示数已拼完，将数压入栈中，并且将变量 num 恢复为 0，准备下一个数。这时对新读入的字符进入'+'、'-'、'*'、'/'及空格的判断，因此在结束处理数字字符的 case 后，不能加入 break 语句。

（5）假设以 I 和 O 分别表示入栈和出栈操作。栈的初态和终态均为空，入栈和出栈的操作序列可表示为仅由 I 和 O 组成的序列，称可以操作的序列为合法序列，否则称为非法序列。

①下面所示的序列中哪些是合法的?

A. IOIIOIOO　　B. IOOIOIIO　　C. IIIOIOIO　　D. IIIOOIOO

②通过对①的分析，写出一个算法，判定所给的操作序列是否合法。若合法，返回 true，否则返回 false（假定被判定的操作序列已存入一维数组中）。

①A 和 D 是合法序列，B 和 C 是非法序列。

②设被判定的操作序列已存入一维数组 A 中。

```
int Judge(char A[])
//判断字符数组 A 中的输入输出序列是否是合法序列。如是，返回 true，否则返
  回 false。
{i=0;                  //i 为下标。
 j=k=0;                //j 和 k 分别为 I 和字母 O 的的个数。
 while(A[i]!='\0') //当未到字符数组尾就作。
   {switch(A[i])
     {case'I': j++; break; //入栈次数增 1。
      case'O': k++; if(k>j){printf("序列非法\n"); exit(0);}
```

```
        }
      i++; //不论 A[i]是'I'或'O', 指针 i 均后移。}
    if(j!=k) {printf("序列非法\n"); return(false);}
    else {printf("序列合法\n"); return(true);}
  }//算法结束。
```

[算法讨论]在入栈出栈序列（即由'I'和'O'组成的字符串）的任一位置，入栈次数（'I'的个数）都必须大于等于出栈次数（即'O'的个数），否则视作非法序列，立即给出信息，退出算法。整个序列（即读到字符数组中字符串的结束标记'\0'），入栈次数必须等于出栈次数（题目中要求栈的初态和终态都为空），否则视为非法序列。

（6）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站点(注意不设头指针），试编写相应的置空队、判队空 、入队和出队等算法。

算法如下：

```
//先定义链队结构:
typedef struct queuenode{
  Datatype data;
  struct queuenode *next;
 }QueueNode; //以上是结点类型的定义

typedef struct{
  queuenode *rear;
 }LinkQueue; //只设一个指向队尾元素的指针

(1)置空队
 void InitQueue( LinkQueue *Q)
  { //置空队：就是使头结点成为队尾元素
   QueueNode *s;
   Q->rear = Q->rear->next;//将队尾指针指向头结点
   while (Q->rear!=Q->rear->next)//当队列非空，将队中元素逐个出队
    {s=Q->rear->next;
     Q->rear->next=s->next;
     free(s);
```

```
    }//回收结点空间
  }
```

(2)判队空

```
 int EmptyQueue( LinkQueue *Q)
  { //判队空
    //当头结点的 next 指针指向自己时为空队
    return Q->rear->next->next==Q->rear->next;
  }
```

(3)入队

```
 void EnQueue( LinkQueue *Q, Datatype x)
  { //入队
    //也就是在尾结点处插入元素
    QueueNode *p=(QueueNode *) malloc (sizeof(QueueNode));//申请新结点
    p->data=x; p->next=Q->rear->next;//初始化新结点并链入
    Q-rear->next=p;
    Q->rear=p;//将尾指针移至新结点
  }
```

(4)出队

```
 Datatype DeQueue( LinkQueue *Q)
  {//出队,把头结点之后的元素摘下
    Datatype t;
    QueueNode *p;
    if(EmptyQueue( Q ))
      Error("Queue underflow");
    p=Q->rear->next->next; //p 指向将要摘下的结点
    x=p->data; //保存结点中数据
    if (p==Q->rear)
```

```
{//当队列中只有一个结点时，p 结点出队后，要将队尾指针指向头结点
  Q->rear = Q->rear->next; Q->rear->next=p->next;}
else
  Q->rear->next->next=p->next;//摘下结点 p
free(p);//释放被删结点
return x;
}
```

（7）假设以数组 Q[*m*]存放循环队列中的元素，同时设置一个标志 *tag*，以 *tag* == 0 和 *tag* == 1 来区别在队头指针(*front*)和队尾指针(*rear*)相等时，队列状态为“空”还是“满”。试编写与此结构相应的插入(*enqueue*)和删除(*dlqueue*)算法。

【解答】

循环队列类定义

```
#include <assert.h>
    template <class Type> class Queue {                    //循环队列的类定义
    public:
        Queue ( int=10 );
        ~Queue ( ) { delete [ ] Q; }
        void EnQueue ( Type & item );
        Type DeQueue ( );
        Type GetFront ( );
        void MakeEmpty ( ) { front = rear = tag = 0; }              //置空队列
        int IsEmpty ( ) const { return front == rear && tag == 0; }    //判队列空否
        int IsFull ( ) const { return front == rear && tag == 1; } //判队列满否
    private:
        int rear, front, tag;                          //队尾指针、队头指针和队满标志
        Type *Q;                                       //存放队列元素的数组
        int m;                                         //队列最大可容纳元素个数
    }
    构造函数
    template <class Type>
    Queue<Type>:: Queue ( int sz ) : rear (0), front (0), tag(0), m (sz) {
    //建立一个最大具有 m 个元素的空队列。
        Q = new Type[m];                               //创建队列空间
        assert ( Q != 0 );                             //断言：动态存储分配成功与否
    }
    插入函数
    template<class Type>
    void Queue<Type> :: EnQueue ( Type &item ) {
          assert ( ! IsFull ( ) );                     //判队列是否不满，满则出错处理
          rear  = ( rear + 1 ) % m;                    //队尾位置进 1，队尾指针指示实际队尾位置
```

```
        Q[rear] = item;                    //进队列
        tag = 1;                           //标志改 1，表示队列不空
    }
    删除函数
    template<class Type>
    Type Queue<Type> :: DeQueue ( ) {
        assert ( ! IsEmpty ( ) );          //判断队列是否不空，空则出错处理
        front = ( front + 1 ) % m;         //队头位置进 1，队头指针指示实际队头的前一
位置
        tag = 0;                           //标志改 0，表示栈不满
        return Q[front];                   //返回原队头元素的值
    }
    读取队头元素函数
    template<class Type>
    Type Queue<Type> :: GetFront ( ) {
        assert ( ! IsEmpty ( ) );          //判断队列是否不空，空则出错处理
        return Q[(front + 1) % m];         //返回队头元素的值
    }
```

(8) 如果允许在循环队列的两端都可以进行插入和删除操作。要求:

① 写出循环队列的类型定义;

② 写出“从队尾删除”和“从队头插入”的算法。

[题目分析] 用一维数组 v[0..M-1]实现循环队列，其中 M 是队列长度。设队头指针 front 和队尾指针 rear，约定 front 指向队头元素的前一位置，rear 指向队尾元素。定义 front=rear 时为队空，(rear+1)%m=front 为队满。约定队头端入队向下标小的方向发展，队尾端入队向下标大的方向发展。

```
(1) #define M  队列可能达到的最大长度
typedef struct
  { elemtp data[M];
    int front,rear;
  } cycqueue;
(2) elemtp delqueue ( cycqueue Q)
//Q 是如上定义的循环队列，本算法实现从队尾删除，若删除成功，返回被删除元素，
  否则给出出错信息。
    { if (Q.front==Q.rear)  {printf("队列空"); exit(0);}
    Q.rear=(Q.rear-1+M)%M;              //修改队尾指针。
        return(Q.data[(Q.rear+1+M)%M]); //返回出队元素。
}//从队尾删除算法结束
void enqueue (cycqueue Q, elemtp x)
// Q 是顺序存储的循环队列，本算法实现“从队头插入”元素 x。
{if (Q.rear==(Q.front-1+M)%M)  {printf("队满"; exit(0);)
 Q.data[Q.front]=x;          //x 入队列
  Q.front=(Q.front-1+M)%M;     //修改队头指针。
}// 结束从队头插入算法。
```

(9) 已知 Ackermann 函数定义如下：

$$Ack(m,n)=\begin{cases} n+1 & \text{当 } m=0 \text{ 时} \\ Ack(m-1,1) & \text{当 } m\neq 0, n=0 \text{ 时} \\ Ack(m-1,Ack(m,n-1)) & \text{当 } m\neq 0, n\neq 0 \text{ 时} \end{cases}$$

① 写出计算 Ack(m,n)的递归算法，并根据此算法给出出 Ack(2,1)的计算过程。

② 写出计算 Ack(m,n)的非递归算法。

```
int Ack(int m,n)
   {if (m==0) return(n+1);
    else if(m!=0&&n==0) return(Ack(m-1,1));
        else return(Ack(m-1,Ack(m,m-1));
    }//算法结束
```

(1) Ack(2,1)的计算过程

```
Ack(2,1)=Ack(1,Ack(2,0))              //因 m<>0,n<>0 而得
        =Ack(1,Ack(1,1))              //因 m<>0,n=0 而得
        =Ack(1,Ack(0,Ack(1,0)))       // 因 m<>0,n<>0 而得
        = Ack(1,Ack(0,Ack(0,1)))      // 因 m<>0,n=0 而得
        =Ack(1,Ack(0,2))              // 因 m=0 而得
        =Ack(1,3)                     // 因 m=0 而得
        =Ack(0,Ack(1,2))              //因 m<>0,n<>0 而得
        = Ack(0,Ack(0,Ack(1,1)))      //因 m<>0,n<>0 而得
        = Ack(0,Ack(0,Ack(0,Ack(1,0)))) //因 m<>0,n<>0 而得
        = Ack(0,Ack(0,Ack(0,Ack(0,1)))) //因 m<>0,n=0 而得
        = Ack(0,Ack(0,Ack(0,2)))      //因 m=0 而得
        = Ack(0,Ack(0,3))             //因 m=0 而得
        = Ack(0,4)                    //因 n=0 而得
        =5                            //因 n=0 而得
```

(2)

```
int Ackerman( int m, int n)
   {int akm[M][N];int i,j;
    for(j=0;j<N;j++) akm[0][j];=j+1;
    for(i=1;i<m;i++)
      {akm[i][0]=akm[i-1][1];
       for(j=1;j<N;j++)
         akm[i][j]=akm[i-1][akm[i][j-1]];
       }
      return(akm[m][n]);
    }//算法结束
```

(10) 已知 f 为单链表的表头指针，链表中存储的都是整型数据，试写出实现下列运算的递归算法：

① 求链表中的最大整数；

② 求链表的结点个数；

③ 求所有整数的平均值。

```
#include <iostream.h>                    //定义在头文件"RecurveList.h"中
```

```
class List;
class ListNode {                                    //链表结点类
friend class List;
private:
    int data;                                       //结点数据
    ListNode *link;                                 //结点指针
    ListNode ( const int item ) : data(item), link(NULL) { }    //构造函数
};
class List {                                        //链表类
private:
    ListNode *first, current;
    int Max ( ListNode *f );
    int Num ( ListNode *f );
    float Avg ( ListNode *f,    int& n );
public:
    List ( ) : first(NULL), current (NULL) { }      //构造函数
    ~List ( ){ }                                    //析构函数
    ListNode* NewNode ( const int item );           //创建链表结点，其值为 item
    void NewList ( const int retvalue );            //建立链表，以输入 retvalue 结束
    void PrintList ( );                             //输出链表所有结点数据
    int GetMax ( ) { return Max ( first ); }        //求链表所有数据的最大值
    int GetNum ( ) { return Num ( first ); }        //求链表中数据个数
    float GetAvg ( ) { return Avg ( first ); }      //求链表所有数据的平均值
};

ListNode* List :: NewNode ( const int item ) {          //创建新链表结点
    ListNode *newnode = new ListNode (item);
    return newnode;
}

void List :: NewList ( const int retvalue ) {           //建立链表，以输入 retvalue 结束
    first = NULL;    int value;    ListNode *q;
    cout << "Input your data:\n";                       //提示
    cin >> value;                                       //输入
    while ( value != retvalue ) {                       //输入有效
        q = NewNode ( value );                          //建立包含 value 的新结点
        if ( first == NULL ) first = current = q;       //空表时，新结点成为链表第一个结点
        else { current->link = q;    current = q; }     //非空表时，新结点链入链尾
        cin >> value;                                   //再输入
    }
    current->link = NULL;                               //链尾封闭
}

void List :: PrintList ( ) {                            //输出链表
```

```
        cout << "\nThe List is : \n";
        ListNode *p = first;
        while ( p != NULL ) {  cout << p->data << '   ';   p = p->link;      }
        cout << '\n';
}

int List :: Max ( ListNode *f ) {                    //递归算法 ： 求链表中的最大值
        if ( f ->link == NULL ) return f ->data;              //递归结束条件
        int temp = Max ( f ->link );                 //在当前结点的后继链表中求最大值
        if ( f ->data > temp ) return f ->data;      //如果当前结点的值还要大，返回当前检点值
        else return temp;                          //否则返回后继链表中的最大值
}

int List :: Num ( ListNode *f ) {                    //递归算法 ： 求链表中结点个数
        if ( f == NULL ) return 0;                   //空表，返回 0
        return 1+ Num ( f ->link );                  //否则，返回后继链表结点个数加 1
}
float List :: Avg ( ListNode *f , int& n ) {         //递归算法 ： 求链表中所有元素的平均值
        if ( f ->link == NULL )                      //链表中只有一个结点，递归结束条件
                { n = 1;    return ( float ) ( f ->data ); }
        else { float Sum = Avg ( f ->link, n ) * n;    n++;    return ( f ->data + Sum ) / n; }
}
#include "RecurveList.h"                             //定义在主文件中
int main ( int argc, char* argv[ ] ) {
        List test;      int finished;
        cout << "输入建表结束标志数据 ： ";
        cin >> finished;                             //输入建表结束标志数据
        test.NewList ( finished );                   //建立链表
        test.PrintList ( );                          //打印链表
        cout << "\nThe Max is : " << test.GetMax ( );
        cout << "\nThe Num is : " << test.GetNum ( );
        cout << "\nThe Ave is : " << test.GetAve () << '\n';
        printf ( "Hello World!\n" );
        return 0;
}
```

第 4 章　串、数组和广义表

(1)已知模式串 t='abcaabbabcab'写出用 KMP 法求得的每个字符对应的 next 和 nextval 函数值。

模式串 t 的 next 和 nextval 值如下:

j	1	2	3	4	5	6	7	8	9	10	11	12
t 串	a	b	c	a	a	b	b	a	b	c	a	b
next[j]	0	1	1	1	2	2	3	1	2	3	4	5
nextval[j]	0	1	1	0	2	1	3	0	1	1	0	5

(2) 设目标为 t="abcaabbabcabaacbacba",模式为 p="abcabaa"

① 计算模式 p 的 naxtval 函数值;

② 不写出算法,只画出利用 KMP 算法进行模式匹配时每一趟的匹配过程。

① p 的 nextval 函数值为 0110132。(p 的 next 函数值为 0111232)。

② 利用 KMP(改进的 nextval)算法, 每趟匹配过程如下:

第一趟匹配: abcaabbabcabaacbacba

abcab(i=5, j=5)

第二趟匹配: abcaabbabcabaacbacba

abc(i=7, j=3)

第三趟匹配: abcaabbabcabaacbacba

a(i=7, j=1)

第四趟匹配: abcaabbabcabaac bacba

(成功) abcabaa(i=15, j=8)

(3) 数组 A 中, 每个元素 A[i,j]的长度均为 32 个二进位,行下标从-1 到 9, 列下标从 1 到 11, 从首地址 S 开始连续存放主存储器中, 主存储器字长为 16 位。求:

① 存放该数组所需多少单元?

② 存放数组第 4 列所有元素至少需多少单元?

③ 数组按行存放时, 元素 A[7,4]的起始地址是多少?

④ 数组按列存放时, 元素 A[4,7]的起始地址是多少?

每个元素 32 个二进制位, 主存字长 16 位, 故每个元素占 2 个字长, 行下标可平移至 1 到 11。

(1) 242 (2) 22 (3) s+182 (4) s+142

(4)请将香蕉 banana 用工具 H( )—Head( ), T( )—Tail( )从 L 中取出。

L=(apple,(orange,(strawberry,(banana)),peach),pear)

H (H (T (H (T (H (T (L)))))))

(5) 写一个算法统计在输入字符串中各个不同字符出现的频度并将结果存入文件 (字符串中的合法字符为 A-Z 这 26 个字母和 0-9 这 10 个数字) 。

```
void Count ()
//统计输入字符串中数字字符和字母字符的个数。
 {int i, num[36];
  char ch;
 for (i=0; i<36; i++) num[i]=0; // 初始化
```

```
while ((ch=getchar ()) !='#')   //'#'表示输入字符串结束。
  if ('0'<=ch<='9') {i=ch-48;num[i]++;}          // 数字字符
   else if ('A'<=ch<='Z') {i=ch-65+10;num[i]++;} // 字母字符

for (i=0; i<10; i++)   // 输出数字字符的个数
   printf ("数字%d 的个数=%d\n", i, num[i]);
for (i=10; i<36; i++) // 求出字母字符的个数
   printf ("字母字符%c 的个数=%d\n", i+55, num[i]);
} // 算法结束。
```

(6) 写一个递归算法来实现字符串逆序存储，要求不另设串存储空间。

[题目分析]实现字符串的逆置并不难，但本题"要求不另设串存储空间"来实现字符串逆序存储，即第一个输入的字符最后存储，最后输入的字符先存储，使用递归可容易做到。

```
void InvertStore(char A[])
//字符串逆序存储的递归算法。
    { char ch;
  static int i = 0;//需要使用静态变量
  scanf ("%c",&ch);
  if (ch!= '.')    //规定'.'是字符串输入结束标志
        {InvertStore(A);
         A[i++] = ch;//字符串逆序存储
        }
  A[i] = '\0';  //字符串结尾标记
   }//结束算法 InvertStore。
```

(7) 编写算法，实现下面函数的功能。函数 void insert(char*s,char*t,int pos)将字符串 t 插入到字符串 s 中，插入位置为 pos。假设分配给字符串 s 的空间足够让字符串 t 插入。(说明：不得使用任何库函数)

[题目分析]本题是字符串的插入问题，要求在字符串 s 的 pos 位置，插入字符串 t。首先应查找字符串 s 的 pos 位置，将第 pos 个字符到字符串 s 尾的子串向后移动字符串 t 的长度，然后将字符串 t 复制到字符串 s 的第 pos 位置后。

对插入位置 pos 要验证其合法性，小于 1 或大于串 s 的长度均为非法，因题目假设给字符串 s 的空间足够大，故对插入不必判溢出。

```
   void  insert(char *s,char *t,int pos)
  //将字符串 t 插入字符串 s 的第 pos 个位置。
    {int i=1,x=0;  char *p=s,*q=t;  //p, q 分别为字符串 s 和 t 的工作指针
     if(pos<1) {printf("pos 参数位置非法\n");exit(0);}
 while(*p!='\0'&&i<pos) {p++;i++;} //查 pos 位置
       //若 pos 小于串 s 长度，则查到 pos 位置时，i=pos。
     if(*p == '/0') {printf("%d 位置大于字符串 s 的长度",pos);exit(0);}
     else      //查找字符串的尾
       while(*p!= '/0') {p++; i++;}  //查到尾时，i 为字符'\0'的下标，p 也指向'\0'。
     while(*q!= '\0') {q++; x++; }   //查找字符串 t 的长度 x，循环结束时 q 指向'\0'。
```

```
        for(j=i;j>=pos  ;j--){*(p+x)=*p;  p--;}//串 s 的 pos 后的子串右移，空出串
    t 的位置。
        q--;  //指针 q 回退到串 t 的最后一个字符
        for(j=1;j<=x;j++) *p--=*q--;  //将 t 串插入到 s 的 pos 位置上
    [算法讨论] 串 s 的结束标记('\0')也后移了，而串 t 的结尾标记不应插入到 s 中。
```

(8) 已知字符串 S1 中存放一段英文，写出算法 format(s1,s2,s3,n)，将其按给定的长度 n 格式化成两端对齐的字符串 S2，其多余的字符送 S3。

```
    [题目分析]本题要求字符串 s1 拆分成字符串 s2 和字符串 s3，要求字符串 s2"按给定长
度 n 格式化成两端对齐的字符串"，即长度为 n 且首尾字符不得为空格字符。算法从左到右
扫描字符串 s1，找到第一个非空格字符，计数到 n，第 n 个拷入字符串 s2 的字符不得为空
格，然后将余下字符复制到字符串 s3 中。
   void format (char *s1,*s2,*s3)
    //将字符串 s1 拆分成字符串 s2 和字符串 s3，要求字符串 s2 是长 n 且两端对齐
   {char *p=s1, *q=s2;
        int i=0;
        while(*p!= '\0' && *p== ' ') p++;//滤掉 s1 左端空格
        if(*p== '\0') {printf("字符串 s1 为空串或空格串\n");exit(0); }
        while( *p!='\0' && i<n){*q=*p; q++; p++; i++;}//字符串 s1 向字符串 s2 中
复制
        if(*p =='\0'){ printf("字符串 s1 没有%d 个有效字符\n",n); exit(0);}
        if(*(--q)==' ') //若最后一个字符为空格，则需向后找到第一个非空格字符
         {p-- ;           //p 指针也后退
          while(*p==' '&&*p!='\0') p++;//往后查找一个非空格字符作串 s2 的尾字符
          if(*p=='\0') {printf("s1 串没有%d 个两端对齐的字符串\n",n); exit(0);
    }
          *q=*p;          //字符串 s2 最后一个非空字符
          *(++q)='\0';    //置 s2 字符串结束标记
          }
        *q=s3;p++;       //将 s1 串其余部分送字符串 s3。
        while (*p!= '\0') {*q=*p; q++; p++;}
        *q='\0';         //置串 s3 结束标记
   }
```

(9)设二维数组 a[1..m, 1..n] 含有 m*n 个整数。

① 写一个算法判断 a 中所有元素是否互不相同?输出相关信息(yes/no);

② 试分析算法的时间复杂度。

```
    [题目分析]判断二维数组中元素是否互不相同，只有逐个比较，找到一对相等的元素，
就可结论为不是互不相同。如何达到每个元素同其它元素比较一次且只一次？在当前行，每
个元素要同本行后面的元素比较一次（下面第一个循环控制变量 p 的 for 循环），然后同第
i+1 行及以后各行元素比较一次，这就是循环控制变量 k 和 p 的二层 for 循环。
int JudgEqual(ing a[m][n],int m,n)
 //判断二维数组中所有元素是否互不相同，如是，返回 1；否则，返回 0。
```

```
{for(i=0;i<m;i++)
  for(j=0;j<n-1;j++)
   { for(p=j+1;p<n;p++) //和同行其它元素比较
      if(a[i][j]==a[i][p]) {printf("no"); return(0); }
  //只要有一个相同的，就结论不是互不相同
     for(k=i+1;k<m;k++)  //和第 i+1 行及以后元素比较
      for(p=0;p<n;p++)
         if(a[i][j]==a[k][p]) {printf("no"); return(0); }
}// for(j=0;j<n-1;j++)
printf(yes"); return(1);   //元素互不相同
}//算法 JudgEqual 结束
```

(2) 二维数组中的每一个元素同其它元素都比较一次，数组中共 m*n 个元素，第 1 个元素同其它 m*n-1 个元素比较，第 2 个元素同其它 m*n-2 个元素比较，……，第 m*n-1 个元素同最后一个元素(m*n)比较一次，所以在元素互不相等时总的比较次数为 (m*n-1)+(m*n-2)+…+2+1= (m*n) (m*n-1)/2。在有相同元素时，可能第一次比较就相同，也可能最后一次比较时相同，设在(m*n-1)个位置上均可能相同，这时的平均比较次数约为 (m*n) (m*n-1)/4，总的时间复杂度是 O(n^4)。

(10)设任意 n 个整数存放于数组 A(1:n)中，试编写算法，将所有正数排在所有负数前面（要求算法复杂性为 0(n)）。

[题目分析]本题属于排序问题，只是排出正负，不排出大小。可在数组首尾设两个指针 i 和 j，i 自小至大搜索到负数停止，j 自大至小搜索到正数停止。然后 i 和 j 所指数据交换，继续以上过程，直到 i=j 为止。

```
void Arrange(int A[],int n)
   //n 个整数存于数组 A 中，本算法将数组中所有正数排在所有负数的前面
  {int i=0,j=n-1,x;  //用类 C 编写，数组下标从 0 开始
   while(i<j)
    {while(i<j && A[i]>0)  i++;
     while(i<j && A[j]<0)  j--;
     if(i<j) {x=A[i]; A[i++]=A[j]; A[j--]=x; }//交换 A[i] 与 A[j]
    }
     }//算法 Arrange 结束.
```

[算法讨论]对数组中元素各比较一次，比较次数为 n。最佳情况(已排好，正数在前，负数在后)不发生交换，最差情况(负数均在正数前面)发生 n/2 次交换。用类 c 编写，数组界偶是 0..n-1。空间复杂度为 O(1).