严蔚敏《数据结构（C语言版）习题集》答案  
  
说明:   
1. 本文是对严蔚敏《数据结构(c语言版)习题集》一书中所有算法设计题目的解决方案,主要作者为一具.以下网友:biwier,szm99,siice,龙抬头,iamkent,zames,birdthinking,lovebuaa等为答案的修订和完善工作提出了宝贵意见,在此表示感谢;  
2. 本解答中的所有算法均采用类c语言描述,设计原则为面向交流、面向阅读,作者不保证程序能够上机正常运行(这种保证实际上也没有任何意义);  
3. 本解答原则上只给出源代码以及必要的注释,对于一些难度较高或思路特殊的题目将给出简要的分析说明,对于作者无法解决的题目将给出必要的讨论.目前尚未解决的题目有: 5.20, 10.40;  
4. 请读者在自己已经解决了某个题目或进行了充分的思考之后,再参考本解答,以保证复习效果;  
5. 由于作者水平所限,本解答中一定存在不少这样或者那样的错误和不足,希望读者们在阅读中多动脑、勤思考,争取发现和纠正这些错误,写出更好的算法来.请将你发现的错误或其它值得改进之处向作者报告:  [email]yi-ju@263.net[/email]   
  
  
第一章 绪论  
  
1.16   
void print\_descending(int x,int y,int z)//按从大到小顺序输出三个数  
{  
  scanf("%d,%d,%d",&x,&y,&z);  
  if(x<y) x<->y; //<->为表示交换的双目运算符,以下同  
  if(y<z) y<->z;  
  if(x<y) x<->y; //冒泡排序  
  printf("%d %d %d",x,y,z);  
}//print\_descending   
1.17   
Status fib(int k,int m,int &f)//求k阶斐波那契序列的第m项的值f  
{  
   int tempd;  
  if(k<2||m<0) return ERROR;   
  if(m<k-1) f=0;  
  else if (m==k-1 || m==k) f=1;  
  else  
  {  
    for(i=0;i<=k-2;i++) temp[i]=0;  
    temp[k-1]=1;temp[k]=1; //初始化  
    sum=1;  
    j=0;  
    for(i=k+1;i<=m;i++,j++) //求出序列第k至第m个元素的值  
      temp[i]=2\*sum-temp[j];  
    f=temp[m];  
  }  
  return OK;  
}//fib  
分析: k阶斐波那契序列的第m项的值f[m]=f[m-1]+f[m-2]+......+f[m-k]  
        =f[m-1]+f[m-2]+......+f[m-k]+f[m-k-1]-f[m-k-1]  
        =2\*f[m-1]-f[m-k-1]  
所以上述算法的时间复杂度仅为O(m). 如果采用递归设计,将达到O(k^m). 即使采用暂存中间结果的方法,也将达到O(m^2).      
1.18   
typedef struct{  
                    char \*sport;  
                    enum{male,female} gender;  
                    char schoolname; //校名为'A','B','C','D'或'E'  
                    char \*result;  
                    int score;  
                  } resulttype;   
typedef struct{  
                    int malescore;  
                    int femalescore;  
                    int totalscore;  
                  } scoretype;   
void summary(resulttype result[ ])//求各校的男女总分和团体总分,假设结果已经储存在result[ ]数组中  
{  
  scoretype score[MAXSIZE];  
  i=0;  
  while(result[i].sport!=NULL)  
  {  
    switch(result[i].schoolname)  
    {  
      case 'A':  
        score[ 0 ].totalscore+=result[i].score;  
        if(result[i].gender==0) score[ 0 ].malescore+=result[i].score;  
        else score[ 0 ].femalescore+=result[i].score;  
        break;  
      case 'B':  
        score[ 0 ].totalscore+=result[i].score;  
        if(result[i].gender==0) score[ 0 ].malescore+=result[i].score;  
        else score[ 0 ].femalescore+=result[i].score;  
        break;  
      ……    ……    ……  
    }  
    i++；  
  }  
  for(i=0;i<5;i++)  
  {  
    printf("School %d:\n",i);  
    printf("Total score of male:%d\n",score[i].malescore);  
    printf("Total score of female:%d\n",score[i].femalescore);  
    printf("Total score of all:%d\n\n",score[i].totalscore);  
  }  
}//summary   
1.19   
Status algo119(int a[ARRSIZE])//求i!\*2^i序列的值且不超过maxint  
{  
  last=1;  
  for(i=1;i<=ARRSIZE;i++)  
  {  
  a[i-1]=last\*2\*i;  
   if((a[i-1]/last)!=(2\*i)) reurn OVERFLOW;  
   last=a[i-1];  
   return OK;  
  }  
}//algo119  
分析:当某一项的结果超过了maxint时,它除以前面一项的商会发生异常.   
1.20   
void polyvalue()  
{  
  float temp;  
  float \*p=a;  
  printf("Input number of terms:");  
  scanf("%d",&n);  
  printf("Input value of x:");  
  scanf("%f",&x);  
  printf("Input the %d coefficients from a0 to a%d:\n",n+1,n);  
  p=a;xp=1;sum=0; //xp用于存放x的i次方  
  for(i=0;i<=n;i++)  
  {  
    scanf("%f",&temp);   
    sum+=xp\*(temp);  
    xp\*=x;  
  }  
  printf("Value is:%f",sum);  
}//polyvalue

 第二章 线性表  
  
2.10   
Status DeleteK(SqList &a,int i,int k)//删除线性表a中第i个元素起的k个元素  
{  
  if(i<1||k<0||i+k-1>a.length) return INFEASIBLE;  
  for(count=1;i+count-1<=a.length-k;count++) //注意循环结束的条件  
    a.elem[i+count-1]=a.elem[i+count+k-1];  
  a.length-=k;  
  return OK;  
}//DeleteK   
2.11  
Status Insert\_SqList(SqList &va,int x)//把x插入递增有序表va中  
{  
  if(va.length+1>va.listsize) return ERROR;  
  va.length++;  
  for(i=va.length-1;va.elem[i]>x&&i>=0;i--)  
    va.elem[i+1]=va.elem[i];  
  va.elem[i+1]=x;  
  return OK;  
}//Insert\_SqList   
2.12   
int ListComp(SqList A,SqList B)//比较字符表A和B,并用返回值表示结果,值为1,表示A>B;值为-1,表示A<B;值为0,表示A=B  
{  
  for(i=1;i<=A.length&&i<=B.length;i++)  
    if(A.elem[i]!=B.elem[i])  
      return A.elem[i]>B.elem[i]?1:-1;  
  if(A.length==B.length) return 0;  
  return A.length>B.length?1:-1;    //当两个字符表可以互相比较的部分完全相同时,哪个较长,哪个就较大  
}//ListComp   
2.13   
LNode\* Locate(LinkList L,int x)//链表上的元素查找,返回指针  
{  
  for(p=l->next;p&&p->data!=x;p=p->next);  
  return p;  
}//Locate   
2.14   
int Length(LinkList L)//求链表的长度  
{  
  for(k=0,p=L;p->next;p=p->next,k++);  
  return k;  
}//Length   
2.15   
void ListConcat(LinkList ha,LinkList hb,LinkList &hc)//把链表hb接在ha后面形成链表hc  
{  
  hc=ha;p=ha;  
  while(p->next) p=p->next;  
  p->next=hb;  
}//ListConcat   
2.16   
见书后答案.   
2.17   
Status Insert(LinkList &L,int i,int b)//在无头结点链表L的第i个元素之前插入元素b  
{  
  p=L;q=(LinkList\*)malloc(sizeof(LNode));  
  q.data=b;  
  if(i==1)  
  {  
    q.next=p;L=q; //插入在链表头部  
  }  
  else  
  {  
    while(--i>1) p=p->next;  
    q->next=p->next;p->next=q; //插入在第i个元素的位置  
  }  
}//Insert   
2.18   
Status Delete(LinkList &L,int i)//在无头结点链表L中删除第i个元素  
{  
  if(i==1) L=L->next; //删除第一个元素  
  else  
  {  
    p=L;  
    while(--i>1) p=p->next;  
    p->next=p->next->next; //删除第i个元素  
  }  
}//Delete   
2.19   
Status Delete\_Between(Linklist &L,int mink,int maxk)//删除元素递增排列的链表L中值大于mink且小于maxk的所有元素  
{  
  p=L;  
  while(p->next->data<=mink) p=p->next; //p是最后一个不大于mink的元素  
  if(p->next)    //如果还有比mink更大的元素  
  {  
    q=p->next;  
    while(q->data<maxk) q=q->next; //q是第一个不小于maxk的元素  
    p->next=q;  
  }  
}//Delete\_Between   
2.20   
Status Delete\_Equal(Linklist &L)//删除元素递增排列的链表L中所有值相同的元素  
{  
  p=L->next;q=p->next; //p,q指向相邻两元素  
  while(p->next)  
  {  
    if(p->data!=q->data)  
    {  
      p=p->next;q=p->next; //当相邻两元素不相等时,p,q都向后推一步  
    }  
    else  
    {  
      while(q->data==p->data)   
   {  
     free(q);  
     q=q->next;   
   }  
      p->next=q;p=q;q=p->next; //当相邻元素相等时删除多余元素  
    }//else  
  }//while  
}//Delete\_Equal   
2.21   
void reverse(SqList &A)//顺序表的就地逆置  
{  
  for(i=1,j=A.length;i<j;i++,j--)  
    A.elem[i]<->A.elem[j];  
}//reverse   
2.22   
void LinkList\_reverse(Linklist &L)//链表的就地逆置;为简化算法,假设表长大于2  
{  
  p=L->next;q=p->next;s=q->next;p->next=NULL;  
  while(s->next)  
  {  
    q->next=p;p=q;  
    q=s;s=s->next; //把L的元素逐个插入新表表头  
  }  
  q->next=p;s->next=q;L->next=s;  
}//LinkList\_reverse  
分析:本算法的思想是,逐个地把L的当前元素q插入新的链表头部,p为新表表头.   
2.23   
void merge1(LinkList &A,LinkList &B,LinkList &C)//把链表A和B合并为C,A和B的元素间隔排列,且使用原存储空间  
{  
  p=A->next;q=B->next;C=A;  
  while(p&&q)  
  {  
    s=p->next;p->next=q; //将B的元素插入  
    if(s)  
    {  
      t=q->next;q->next=s; //如A非空,将A的元素插入  
    }  
    p=s;q=t;  
  }//while  
}//merge1   
2.24   
void reverse\_merge(LinkList &A,LinkList &B,LinkList &C)//把元素递增排列的链表A和B合并为C,且C中元素递减排列,使用原空间  
{  
  pa=A->next;pb=B->next;pre=NULL; //pa和pb分别指向A,B的当前元素  
  while(pa||pb)  
  {  
    if(pa->data<pb->data||!pb)  
    {  
      pc=pa;q=pa->next;pa->next=pre;pa=q; //将A的元素插入新表  
    }  
    else  
    {  
      pc=pb;q=pb->next;pb->next=pre;pb=q; //将B的元素插入新表  
    }  
    pre=pc;  
  }  
  C=A;A->next=pc; //构造新表头  
}//reverse\_merge  
分析:本算法的思想是,按从小到大的顺序依次把A和B的元素插入新表的头部pc处,最后处理A或B的剩余元素.   
2.25   
void SqList\_Intersect(SqList A,SqList B,SqList &C)//求元素递增排列的线性表A和B的元素的交集并存入C中  
{  
  i=1;j=1;k=0;  
  while(A.elem[i]&&B.elem[j])  
  {  
    if(A.elem[i]<B.elem[j]) i++;  
    if(A.elem[i]>B.elem[j]) j++;  
    if(A.elem[i]==B.elem[j])  
    {  
      C.elem[++k]=A.elem[i]; //当发现了一个在A,B中都存在的元素,  
      i++;j++; //就添加到C中  
    }  
  }//while  
}//SqList\_Intersect   
2.26   
void LinkList\_Intersect(LinkList A,LinkList B,LinkList &C)//在链表结构上重做上题  
{  
  p=A->next;q=B->next;  
  pc=(LNode\*)malloc(sizeof(LNode));   
C=pc;  
  while(p&&q)  
  {  
    if(p->data<q->data) p=p->next;  
    else if(p->data>q->data) q=q->next;  
    else  
    {  
      s=(LNode\*)malloc(sizeof(LNode));  
      s->data=p->data;  
      pc->next=s;pc=s;  
      p=p->next;q=q->next;  
    }  
  }//while  
}//LinkList\_Intersect   
2.27   
void SqList\_Intersect\_True(SqList &A,SqList B)//求元素递增排列的线性表A和B的元素的交集并存回A中  
{  
  i=1;j=1;k=0;  
  while(A.elem[i]&&B.elem[j])  
  {  
    if(A.elem[i]<B.elem[j]) i++;  
    else if(A.elem[i]>B.elem[j]) j++;  
    else if(A.elem[i]!=A.elem[k])  
    {  
      A.elem[++k]=A.elem[i]; //当发现了一个在A,B中都存在的元素  
      i++;j++; //且C中没有,就添加到C中  
    }  
    else {i++;j++;}  
  }//while  
  while(A.elem[k]) A.elem[k++]=0;  
}//SqList\_Intersect\_True   
2.28   
void LinkList\_Intersect\_True(LinkList &A,LinkList B)//在链表结构上重做上题  
{  
  p=A->next;q=B->next;pc=A;  
  while(p&&q)  
  {  
    if(p->data<q->data) p=p->next;  
    else if(p->data>q->data) q=q->next;  
    else if(p->data!=pc->data)  
    {  
      pc=pc->next;  
      pc->data=p->data;  
      p=p->next;q=q->next;  
    }  
  }//while  
}//LinkList\_Intersect\_True   
2.29   
void SqList\_Intersect\_Delete(SqList &A,SqList B,SqList C)   
{  
  i=0;j=0;k=0;m=0;        //i指示A中元素原来的位置,m为移动后的位置  
  while(i<A.length&&j<B.length&& k<C.length)   
  {  
    if(B.elem[j]<C.elem[k]) j++;  
    else if(B.elem[j]>C.elem[k]) k++;  
    else  
    {  
      same=B.elem[j];                        //找到了相同元素same  
      while(B.elem[j]==same) j++;  
      while(C.elem[k]==same) k++;        //j,k后移到新的元素  
      while(i<A.length&&A.elem[i]<same)   
        A.elem[m++]=A.elem[i++];                //需保留的元素移动到新位置  
      while(i<A.length&&A.elem[i]==same) i++;        //跳过相同的元素  
    }  
  }//while  
  while(i<A.length)   
    A.elem[m++]=A.elem[i++];        //A的剩余元素重新存储。  
  A.length=m;   
}// SqList\_Intersect\_Delete  
分析:先从B和C中找出共有元素,记为same,再在A中从当前位置开始, 凡小于same的  
元素均保留(存到新的位置),等于same的就跳过,到大于same时就再找下一个same.   
2.30   
void LinkList\_Intersect\_Delete(LinkList &A,LinkList B,LinkList C)//在链表结构上重做上题  
{  
  p=B->next;q=C->next;r=A-next;  
  while(p&&q&&r)  
  {  
    if(p->data<q->data) p=p->next;  
    else if(p->data>q->data) q=q->next;  
    else  
    {  
      u=p->data; //确定待删除元素u  
      while(r->next->data<u) r=r->next; //确定最后一个小于u的元素指针r  
      if(r->next->data==u)  
      {  
        s=r->next;  
        while(s->data==u)  
        {  
          t=s;s=s->next;free(t); //确定第一个大于u的元素指针s  
        }//while  
        r->next=s; //删除r和s之间的元素  
      }//if  
      while(p->data=u) p=p->next;  
      while(q->data=u) q=q->next;  
    }//else  
  }//while  
}//LinkList\_Intersect\_Delete   
2.31   
Status Delete\_Pre(CiLNode \*s)//删除单循环链表中结点s的直接前驱  
{  
  p=s;  
  while(p->next->next!=s) p=p->next; //找到s的前驱的前驱p  
  p->next=s;  
  return OK;  
}//Delete\_Pre   
2.32   
Status DuLNode\_Pre(DuLinkList &L)//完成双向循环链表结点的pre域  
{  
  for(p=L;!p->next->pre;p=p->next) p->next->pre=p;  
  return OK;  
}//DuLNode\_Pre   
2.33   
Status LinkList\_Divide(LinkList &L,CiList &A,CiList &B,CiList &C)//把单链表L的元素按类型分为三个循环链表.CiList为带头结点的单循环链表类型.  
{  
  s=L->next;  
  A=(CiList\*)malloc(sizeof(CiLNode));p=A;  
  B=(CiList\*)malloc(sizeof(CiLNode));q=B;  
  C=(CiList\*)malloc(sizeof(CiLNode));r=C; //建立头结点  
  while(s)  
  {  
    if(isalphabet(s->data))  
    {  
      p->next=s;p=s;  
    }  
    else if(isdigit(s->data))  
    {  
      q->next=s;q=s;  
    }  
    else  
    {  
      r->next=s;r=s;  
    }  
  }//while  
  p->next=A;q->next=B;r->next=C; //完成循环链表  
}//LinkList\_Divide   
2.34   
void Print\_XorLinkedList(XorLinkedList L)//从左向右输出异或链表的元素值  
{  
  p=L.left;pre=NULL;  
  while(p)  
  {  
    printf("%d",p->data);  
    q=XorP(p->LRPtr,pre);  
    pre=p;p=q; //任何一个结点的LRPtr域值与其左结点指针进行异或运算即得到其右结点指针  
  }  
}//Print\_XorLinkedList   
2.35   
Status Insert\_XorLinkedList(XorLinkedList &L,int x,int i)//在异或链表L的第i个元素前插入元素x  
{  
  p=L.left;pre=NULL;  
  r=(XorNode\*)malloc(sizeof(XorNode));  
  r->data=x;  
  if(i==1) //当插入点在最左边的情况  
  {  
    p->LRPtr=XorP(p.LRPtr,r);  
    r->LRPtr=p;  
    L.left=r;  
    return OK;  
  }  
  j=1;q=p->LRPtr; //当插入点在中间的情况  
  while(++j<i&&q)  
  {  
    q=XorP(p->LRPtr,pre);  
    pre=p;p=q;  
  }//while //在p,q两结点之间插入  
  if(!q) return INFEASIBLE; //i不可以超过表长  
  p->LRPtr=XorP(XorP(p->LRPtr,q),r);  
  q->LRPtr=XorP(XorP(q->LRPtr,p),r);  
  r->LRPtr=XorP(p,q); //修改指针  
  return OK;  
}//Insert\_XorLinkedList   
2.36   
Status Delete\_XorLinkedList(XorlinkedList &L,int i)//删除异或链表L的第i个元素  
{  
  p=L.left;pre=NULL;  
  if(i==1) //删除最左结点的情况  
  {  
    q=p->LRPtr;  
    q->LRPtr=XorP(q->LRPtr,p);  
    L.left=q;free(p);  
    return OK;  
  }  
  j=1;q=p->LRPtr;  
  while(++j<i&&q)  
  {  
    q=XorP(p->LRPtr,pre);  
    pre=p;p=q;  
  }//while //找到待删结点q  
  if(!q) return INFEASIBLE; //i不可以超过表长  
  if(L.right==q) //q为最右结点的情况  
  {  
    p->LRPtr=XorP(p->LRPtr,q);  
    L.right=p;free(q);  
    return OK;  
  }  
  r=XorP(q->LRPtr,p); //q为中间结点的情况,此时p,r分别为其左右结点  
  p->LRPtr=XorP(XorP(p->LRPtr,q),r);  
  r->LRPtr=XorP(XorP(r->LRPtr,q),p); //修改指针  
  free(q);  
  return OK;  
}//Delete\_XorLinkedList   
2.37   
void OEReform(DuLinkedList &L)//按1,3,5,...4,2的顺序重排双向循环链表L中的所有结点  
{  
  p=L.next;  
  while(p->next!=L&&p->next->next!=L)  
  {  
    p->next=p->next->next;  
    p=p->next;  
  } //此时p指向最后一个奇数结点  
  if(p->next==L) p->next=L->pre->pre;  
  else p->next=l->pre;  
  p=p->next; //此时p指向最后一个偶数结点  
  while(p->pre->pre!=L)  
  {  
    p->next=p->pre->pre;  
    p=p->next;  
  }  
  p->next=L; //按题目要求调整了next链的结构,此时pre链仍为原状  
  for(p=L;p->next!=L;p=p->next) p->next->pre=p;  
  L->pre=p; //调整pre链的结构,同2.32方法  
}//OEReform  
分析:next链和pre链的调整只能分开进行.如同时进行调整的话,必须使用堆栈保存偶数结点的指针,否则将会破坏链表结构,造成结点丢失.   
2.38   
DuLNode \* Locate\_DuList(DuLinkedList &L,int x)//带freq域的双向循环链表上的查找  
{  
  p=L.next;  
  while(p.data!=x&&p!=L) p=p->next;  
  if(p==L) return NULL; //没找到  
  p->freq++;q=p->pre;  
  while(q->freq<=p->freq&&p!=L) q=q->pre; //查找插入位置  
  if(q!=p->pre)  
  {  
    p->pre->next=p->next;p->next->pre=p->pre;  
    q->next->pre=p;p->next=q->next;  
    q->next=p;p->pre=q; //调整位置  
  }  
  return p;  
}//Locate\_DuList   
2.39   
float GetValue\_SqPoly(SqPoly P,int x0)//求升幂顺序存储的稀疏多项式的值  
{  
  PolyTerm \*q;  
  xp=1;q=P.data;  
  sum=0;ex=0;  
  while(q->coef)  
  {  
    while(ex<q->exp) xp\*=x0;  
    sum+=q->coef\*xp;  
    q++;  
  }  
  return sum;  
}//GetValue\_SqPoly   
2.40   
void Subtract\_SqPoly(SqPoly P1,SqPoly P2,SqPoly &P3)//求稀疏多项式P1减P2的差式P3  
{  
  PolyTerm \*p,\*q,\*r;  
  Create\_SqPoly(P3); //建立空多项式P3  
  p=P1.data;q=P2.data;r=P3.data;  
  while(p->coef&&q->coef)  
  {  
    if(p->exp<q->exp)  
    {  
      r->coef=p->coef;  
      r->exp=p->exp;  
      p++;r++;  
    }  
    else if(p->exp<q->exp)  
    {  
      r->coef=-q->coef;  
      r->exp=q->exp;  
      q++;r++;  
    }  
    else  
    {  
      if((p->coef-q->coef)!=0) //只有同次项相减不为零时才需要存入P3中  
      {  
        r->coef=p->coef-q->coef;  
        r->exp=p->exp;r++;  
      }//if  
      p++;q++;  
    }//else  
  }//while  
  while(p->coef) //处理P1或P2的剩余项  
  {  
    r->coef=p->coef;  
    r->exp=p->exp;  
    p++;r++;  
  }  
  while(q->coef)  
  {  
    r->coef=-q->coef;  
    r->exp=q->exp;  
    q++;r++;  
  }  
}//Subtract\_SqPoly   
2.41   
void QiuDao\_LinkedPoly(LinkedPoly &L)//对有头结点循环链表结构存储的稀疏多项式L求导  
{  
  p=L->next;  
  if(!p->data.exp)  
  {  
    L->next=p->next;p=p->next; //跳过常数项  
  }  
  while(p!=L)  
  {  
    p->data.coef\*=p->data.exp--;//对每一项求导  
    p=p->next;  
  }  
}//QiuDao\_LinkedPoly   
2.42   
void Divide\_LinkedPoly(LinkedPoly &L,&A,&B)//把循环链表存储的稀疏多项式L拆成只含奇次项的A和只含偶次项的B  
{  
  p=L->next;  
  A=(PolyNode\*)malloc(sizeof(PolyNode));  
  B=(PolyNode\*)malloc(sizeof(PolyNode));  
  pa=A;pb=B;  
  while(p!=L)  
  {  
    if(p->data.exp!=2\*(p->data.exp/2))  
    {  
      pa->next=p;pa=p;  
    }  
    else  
    {  
      pb->next=p;pb=p;  
    }  
    p=p->next;  
  }//while  
  pa->next=A;pb->next=B;   
}//Divide\_LinkedPoly

第三章 栈与队列  
  
3.15   
typedef struct{  
                    Elemtype \*base[2];  
                    Elemtype \*top[2];  
                  }BDStacktype; //双向栈类型   
Status Init\_Stack(BDStacktype &tws,int m)//初始化一个大小为m的双向栈tws  
{  
  tws.base[0]=(Elemtype\*)malloc(sizeof(Elemtype));  
  tws.base[1]=tws.base[0]+m;  
  tws.top[0]=tws.base[0];  
  tws.top[1]=tws.base[1];  
  return OK;  
}//Init\_Stack   
Status push(BDStacktype &tws,int i,Elemtype x)//x入栈,i=0表示低端栈,i=1表示高端栈  
{  
  if(tws.top[0]>tws.top[1]) return OVERFLOW; //注意此时的栈满条件  
  if(i==0) \*tws.top[0]++=x;  
  else if(i==1) \*tws.top[1]--=x;  
  else return ERROR;  
  return OK;  
}//push   
Status pop(BDStacktype &tws,int i,Elemtype &x)//x出栈,i=0表示低端栈,i=1表示高端栈  
{  
  if(i==0)  
  {  
    if(tws.top[0]==tws.base[0]) return OVERFLOW;  
    x=\*--tws.top[0];  
  }  
  else if(i==1)  
  {  
    if(tws.top[1]==tws.base[1]) return OVERFLOW;  
    x=\*++tws.top[1];  
  }  
  else return ERROR;  
  return OK;  
}//pop   
3.16   
void Train\_arrange(char \*train)//这里用字符串train表示火车,'H'表示硬席,'S'表示软席  
{  
  p=train;q=train;  
  InitStack(s);  
  while(\*p)  
  {  
    if(\*p=='H') push(s,\*p); //把'H'存入栈中  
    else \*(q++)=\*p; //把'S'调到前部  
    p++;  
  }  
  while(!StackEmpty(s))  
  {  
    pop(s,c);  
    \*(q++)=c; //把'H'接在后部  
  }  
}//Train\_arrange   
3.17   
int IsReverse()//判断输入的字符串中'&'前和'&'后部分是否为逆串,是则返回1,否则返回0  
{  
  InitStack(s);  
  while((e=getchar())!='&')  
  {  
    if(e==’@’) return 0;//不允许在’&’之前出现’@’  
    push(s,e);   
  }  
  while( (e=getchar())!='@')  
  {  
    if(StackEmpty(s)) return 0;  
    pop(s,c);  
    if(e!=c) return 0;  
  }  
  if(!StackEmpty(s)) return 0;  
  return 1;  
}//IsReverse   
3.18   
Status Bracket\_Test(char \*str)//判别表达式中小括号是否匹配  
{  
  count=0;  
  for(p=str;\*p;p++)  
  {  
    if(\*p=='(') count++;  
    else if(\*p==')') count--;  
    if (count<0) return ERROR;  
  }  
  if(count) return ERROR; //注意括号不匹配的两种情况  
  return OK;  
}//Bracket\_Test   
3.19   
Status AllBrackets\_Test(char \*str)//判别表达式中三种括号是否匹配  
{  
  InitStack(s);  
  for(p=str;\*p;p++)  
  {  
    if(\*p=='('||\*p=='['||\*p=='{') push(s,\*p);  
    else if(\*p==')'||\*p==']'||\*p=='}')  
    {  
      if(StackEmpty(s)) return ERROR;  
      pop(s,c);  
      if(\*p==')'&&c!='(') return ERROR;  
      if(\*p==']'&&c!='[') return ERROR;  
      if(\*p=='}'&&c!='{') return ERROR; //必须与当前栈顶括号匹配  
    }  
  }//for  
  if(!StackEmpty(s)) return ERROR;  
  return OK;  
}//AllBrackets\_Test   
3.20   
typedef struct {  
.             int x;   
              int y;   
            } coordinate;  
  
  
void Repaint\_Color(int g[m][n],int i,int j,int color)//把点(i,j)相邻区域的颜色置换为color  
{  
  old=g[i][j];  
  InitQueue(Q);  
  EnQueue(Q,{I,j});  
  while(!QueueEmpty(Q))  
  {  
    DeQueue(Q,a);  
  x=a.x;y=a.y;  
    if(x>1)  
      if(g[x-1][y]==old)  
      {  
        g[x-1][y]=color;  
        EnQueue(Q,{x-1,y}); //修改左邻点的颜色  
      }  
    if(y>1)  
      if(g[x][y-1]==old)  
      {  
        g[x][y-1]=color;  
        EnQueue(Q,{x,y-1}); //修改上邻点的颜色  
      }  
    if(x<m)  
      if(g[x+1][y]==old)  
      {  
        g[x+1][y]=color;  
        EnQueue(Q,{x+1,y}); //修改右邻点的颜色  
      }  
    if(y<n)  
      if(g[x][y+1]==old)  
      {  
        g[x][y+1]=color;  
        EnQueue(Q,{x,y+1}); //修改下邻点的颜色  
      }  
  }//while  
}//Repaint\_Color  
分析:本算法采用了类似于图的广度优先遍历的思想,用两个队列保存相邻同色点的横坐标和纵坐标.递归形式的算法该怎么写呢?   
3.21   
void NiBoLan(char \*str,char \*new)//把中缀表达式str转换成逆波兰式new  
{  
  p=str;q=new; //为方便起见,设str的两端都加上了优先级最低的特殊符号  
  InitStack(s); //s为运算符栈  
  while(\*p)  
  {  
    if(\*p是字母)) \*q++=\*p; //直接输出  
    else  
    {  
      c=gettop(s);  
      if(\*p优先级比c高) push(s,\*p);  
      else  
      {  
        while(gettop(s)优先级不比\*p低)  
        {  
          pop(s,c);\*(q++)=c;  
        }//while  
        push(s,\*p); //运算符在栈内遵循越往栈顶优先级越高的原则  
      }//else  
    }//else  
    p++;  
  }//while  
}//NiBoLan //参见编译原理教材   
3.22   
int GetValue\_NiBoLan(char \*str)//对逆波兰式求值  
{  
  p=str;InitStack(s); //s为操作数栈  
  while(\*p)  
  {  
    if(\*p是数) push(s,\*p);  
    else  
    {  
      pop(s,a);pop(s,b);  
      r=compute(b,\*p,a); //假设compute为执行双目运算的过程  
      push(s,r);  
    }//else  
    p++;  
  }//while  
  pop(s,r);return r;  
}//GetValue\_NiBoLan   
3.23   
Status NiBoLan\_to\_BoLan(char \*str,stringtype &new)//把逆波兰表达式str转换为波兰式new  
{  
  p=str;Initstack(s); //s的元素为stringtype类型  
  while(\*p)  
  {  
    if(\*p为字母) push(s,\*p);  
    else  
    {  
      if(StackEmpty(s)) return ERROR;  
      pop(s,a);  
      if(StackEmpty(s)) return ERROR;  
      pop(s,b);  
      c=link(link(\*p,b),a);  
      push(s,c);  
    }//else  
    p++;  
  }//while  
  pop(s,new);  
  if(!StackEmpty(s)) return ERROR;  
  return OK;  
}//NiBoLan\_to\_BoLan  
分析:基本思想见书后注释.本题中暂不考虑串的具体操作的实现,而将其看作一种抽象数据类型stringtype,对其可以进行连接操作:c=link(a,b).   
3.24   
Status g(int m,int n,int &s)//求递归函数g的值s  
{  
  if(m==0&&n>=0) s=0;  
  else if(m>0&&n>=0) s=n+g(m-1,2\*n);  
  else return ERROR;  
  return OK;  
}//g   
3.25   
Status F\_recursive(int n,int &s)//递归算法  
{  
  if(n<0) return ERROR;  
  if(n==0) s=n+1;  
  else  
  {  
    F\_recurve(n/2,r);  
    s=n\*r;  
  }  
  return OK;  
}//F\_recursive   
Status F\_nonrecursive(int n,int s)//非递归算法  
{  
  if(n<0) return ERROR;  
  if(n==0) s=n+1;  
  else  
  {  
    InitStack(s); //s的元素类型为struct {int a;int b;}  
    while(n!=0)  
    {  
      a=n;b=n/2;  
      push(s,{a,b});  
      n=b;  
    }//while  
    s=1;  
    while(!StackEmpty(s))  
    {  
      pop(s,t);  
      s\*=t.a;  
    }//while  
  }  
  return OK;  
}//F\_nonrecursive   
3.26   
float Sqrt\_recursive(float A,float p,float e)//求平方根的递归算法  
{  
  if(abs(p^2-A)<=e) return p;  
  else return sqrt\_recurve(A,(p+A/p)/2,e);  
}//Sqrt\_recurve   
float Sqrt\_nonrecursive(float A,float p,float e)//求平方根的非递归算法  
{  
  while(abs(p^2-A)>=e)  
    p=(p+A/p)/2;  
  return p;  
}//Sqrt\_nonrecursive   
3.27   
这一题的所有算法以及栈的变化过程请参见《数据结构(pascal版)》,作者不再详细写出.   
3.28   
void InitCiQueue(CiQueue &Q)//初始化循环链表表示的队列Q  
{  
  Q=(CiLNode\*)malloc(sizeof(CiLNode));  
  Q->next=Q;  
}//InitCiQueue   
void EnCiQueue(CiQueue &Q,int x)//把元素x插入循环链表表示的队列Q,Q指向队尾元素,Q->next指向头结点,Q->next->next指向队头元素  
{  
  p=(CiLNode\*)malloc(sizeof(CiLNode));  
  p->data=x;  
  p->next=Q->next; //直接把p加在Q的后面  
  Q->next=p;  
  Q=p;  //修改尾指针  
}   
Status DeCiQueue(CiQueue &Q,int x)//从循环链表表示的队列Q头部删除元素x  
{  
  if(Q==Q->next) return INFEASIBLE; //队列已空  
  p=Q->next->next;  
  x=p->data;  
  Q->next->next=p->next;  
  free(p);  
  return OK;  
}//DeCiQueue   
3.29   
Status EnCyQueue(CyQueue &Q,int x)//带tag域的循环队列入队算法  
{  
  if(Q.front==Q.rear&&Q.tag==1) //tag域的值为0表示"空",1表示"满"  
    return OVERFLOW;  
  Q.base[Q.rear]=x;  
  Q.rear=(Q.rear+1)%MAXSIZE;  
  if(Q.front==Q.rear) Q.tag=1; //队列满  
}//EnCyQueue   
Status DeCyQueue(CyQueue &Q,int &x)//带tag域的循环队列出队算法  
{  
  if(Q.front==Q.rear&&Q.tag==0) return INFEASIBLE;  
  Q.front=(Q.front+1)%MAXSIZE;  
  x=Q.base[Q.front];  
  if(Q.front==Q.rear) Q.tag=1; //队列空  
  return OK;  
}//DeCyQueue  
分析:当循环队列容量较小而队列中每个元素占的空间较多时,此种表示方法可以节约较多的存储空间,较有价值.   
3.30   
Status EnCyQueue(CyQueue &Q,int x)//带length域的循环队列入队算法  
{  
  if(Q.length==MAXSIZE) return OVERFLOW;  
  Q.rear=(Q.rear+1)%MAXSIZE;  
  Q.base[Q.rear]=x;  
  Q.length++;  
  return OK;  
}//EnCyQueue   
Status DeCyQueue(CyQueue &Q,int &x)//带length域的循环队列出队算法  
{  
  if(Q.length==0) return INFEASIBLE;  
  head=(Q.rear-Q.length+1)%MAXSIZE; //详见书后注释  
  x=Q.base[head];  
  Q.length--;  
}//DeCyQueue   
3.31   
int Palindrome\_Test()//判别输入的字符串是否回文序列,是则返回1,否则返回0  
{  
  InitStack(S);InitQueue(Q);  
  while((c=getchar())!='@')  
  {  
    Push(S,c);EnQueue(Q,c); //同时使用栈和队列两种结构  
  }  
  while(!StackEmpty(S))  
  {  
    Pop(S,a);DeQueue(Q,b));  
    if(a!=b) return ERROR;  
  }  
  return OK;  
}//Palindrome\_Test   
3.32   
void GetFib\_CyQueue(int k,int n)//求k阶斐波那契序列的前n+1项  
{  
  InitCyQueue(Q); //其MAXSIZE设置为k  
  for(i=0;i<k-1;i++) Q.base[i]=0;  
  Q.base[k-1]=1; //给前k项赋初值  
  for(i=0;i<k;i++) printf("%d",Q.base[i]);  
  for(i=k;i<=n;i++)  
  {  
    m=i%k;sum=0;  
    for(j=0;j<k;j++) sum+=Q.base[(m+j)%k];  
    Q.base[m]=sum; //求第i项的值存入队列中并取代已无用的第一项  
    printf("%d",sum);  
  }  
}//GetFib\_CyQueue   
3.33   
Status EnDQueue(DQueue &Q,int x)//输出受限的双端队列的入队操作  
{  
  if((Q.rear+1)%MAXSIZE==Q.front) return OVERFLOW; //队列满  
  avr=(Q.base[Q.rear-1]+Q.base[Q.front])/2;  
  if(x>=avr) //根据x的值决定插入在队头还是队尾  
  {  
    Q.base[Q.rear]=x;  
    Q.rear=(Q.rear+1)%MAXSIZE;  
  } //插入在队尾  
  else  
  {  
    Q.front=(Q.front-1)%MAXSIZE;  
    Q.base[Q.front]=x;  
  } //插入在队头  
  return OK;  
}//EnDQueue   
Status DeDQueue(DQueue &Q,int &x)//输出受限的双端队列的出队操作  
{  
  if(Q.front==Q.rear) return INFEASIBLE; //队列空  
  x=Q.base[Q.front];  
  Q.front=(Q.front+1)%MAXSIZE;  
  return OK;  
}//DeDQueue   
3.34   
void Train\_Rearrange(char \*train)//这里用字符串train表示火车,'P'表示硬座,'H'表示硬卧,'S'表示软卧,最终按PSH的顺序排列  
{  
  r=train;  
  InitDQueue(Q);  
  while(\*r)  
  {  
    if(\*r=='P')  
    {  
      printf("E");  
      printf("D"); //实际上等于不入队列,直接输出P车厢  
    }  
    else if(\*r=='S')  
    {  
      printf("E");  
      EnDQueue(Q,\*r,0); //0表示把S车厢从头端入队列  
    }  
    else  
    {  
      printf("A");  
      EnDQueue(Q,\*r,1); //1表示把H车厢从尾端入队列  
    }  
  }//while  
  while(!DQueueEmpty(Q))  
  {  
    printf("D");  
    DeDQueue(Q);  
  }//while //从头端出队列的车厢必然是先S后H的顺序   
}//Train\_Rearrange

第四章 串  
  
4.10   
void String\_Reverse(Stringtype s,Stringtype &r)//求s的逆串r  
{  
  StrAssign(r,''); //初始化r为空串  
  for(i=Strlen(s);i;i--)  
  {  
    StrAssign(c,SubString(s,i,1));  
    StrAssign(r,Concat(r,c)); //把s的字符从后往前添加到r中  
  }  
}//String\_Reverse   
4.11   
void String\_Subtract(Stringtype s,Stringtype t,Stringtype &r)//求所有包含在串s中而t中没有的字符构成的新串r  
{  
  StrAssign(r,'');  
  for(i=1;i<=Strlen(s);i++)  
  {  
    StrAssign(c,SubString(s,i,1));  
    for(j=1;j<i&&StrCompare(c,SubString(s,j,1));j++); //判断s的当前字符c是否第一次出现  
    if(i==j)  
    {  
      for(k=1;k<=Strlen(t)&&StrCompare(c,SubString(t,k,1));k++); //判断当前字符是否包含在t中  
      if(k>Strlen(t)) StrAssign(r,Concat(r,c));  
    }  
  }//for  
}//String\_Subtract   
4.12   
int Replace(Stringtype &S,Stringtype T,Stringtype V);//将串S中所有子串T替换为V,并返回置换次数  
{  
  for(n=0,i=1;i<=Strlen(S)-Strlen(T)+1;i++) //注意i的取值范围  
    if(!StrCompare(SubString(S,i,Strlen(T)),T)) //找到了与T匹配的子串  
    { //分别把T的前面和后面部分保存为head和tail  
      StrAssign(head,SubString(S,1,i-1));  
      StrAssign(tail,SubString(S,i+Strlen(T),Strlen(S)-i-Strlen(T)+1));  
      StrAssign(S,Concat(head,V));  
      StrAssign(S,Concat(S,tail)); //把head,V,tail连接为新串  
      i+=Strlen(V); //当前指针跳到插入串以后  
      n++;  
    }//if  
  return n;  
}//Replace  
分析:i+=Strlen(V);这一句是必需的,也是容易忽略的.如省掉这一句,则在某些情况下,会引起不希望的后果,虽然在大多数情况下没有影响.请思考:设S='place', T='ace', V='face',则省掉i+=Strlen(V);运行时会出现什么结果?   
4.13   
int Delete\_SubString(Stringtype &s,Stringtype t)//从串s中删除所有与t相同的子串,并返回删除次数  
{  
  for(n=0,i=1;i<=Strlen(s)-Strlen(t)+1;i++)  
    if(!StrCompare(SubString(s,i,Strlen(t)),t))  
    {  
      StrAssign(head,SubString(S,1,i-1));  
      StrAssign(tail,SubString(S,i+Strlen(t),Strlen(s)-i-Strlen(t)+1));  
      StrAssign(S,Concat(head,tail)); //把head,tail连接为新串  
      n++;  
    }//if  
  return n,  
}//Delete\_SubString   
4.14   
Status NiBoLan\_to\_BoLan(Stringtype str,Stringtype &new)//把前缀表达式str转换为后缀式new  
{  
  Initstack(s); //s的元素为Stringtype类型  
  for(i=1;i<=Strlen(str);i++)  
  {  
    r=SubString(str,i,1);  
    if(r为字母) push(s,r);  
    else  
    {  
      if(StackEmpty(s)) return ERROR;  
      pop(s,a);  
      if(StackEmpty(s)) return ERROR;  
      pop(s,b);  
      StrAssign(t,Concat(r,b));  
      StrAssign(c,Concat(t,a)); //把算符r,子前缀表达式a,b连接为新子前缀表达式c  
      push(s,c);  
    }  
  }//for  
  pop(s,new);  
  if(!StackEmpty(s)) return ERROR;  
  return OK;  
}//NiBoLan\_to\_BoLan  
分析:基本思想见书后注释3.23.请读者用此程序取代作者早些时候对3.23题给出的程序.   
4.15   
void StrAssign(Stringtype &T,char chars&#;)//用字符数组chars给串T赋值,Stringtype的定义见课本  
{  
  for(i=0,T[0]=0;chars[i];T[0]++,i++) T[i+1]=chars[i];  
}//StrAssign   
4.16   
char StrCompare(Stringtype s,Stringtype t)//串的比较,s>t时返回正数,s=t时返回0,s<t时返回负数  
{  
  for(i=1;i<=s[0]&&i<=t[0]&&s[i]==t[i];i++);  
  if(i>s[0]&&i>t[0]) return 0;  
  else if(i>s[0]) return -t[i];  
  else if(i>t[0]) return s[i];  
  else return s[i]-t[i];  
}//StrCompare   
4.17   
int String\_Replace(Stringtype &S,Stringtype T,Stringtype V);//将串S中所有子串T替换为V,并返回置换次数  
{  
  for(n=0,i=1;i<=S[0]-T[0]+1;i++)  
  {  
    for(j=i,k=1;T[k]&&S[j]==T[k];j++,k++);  
    if(k>T[0]) //找到了与T匹配的子串:分三种情况处理  
    {  
      if(T[0]==V[0])  
        for(l=1;l<=T[0];l++) //新子串长度与原子串相同时:直接替换  
          S[i+l-1]=V[l];  
      else if(T[0]<V[0]) //新子串长度大于原子串时:先将后部右移  
      {  
        for(l=S[0];l>=i+T[0];l--)  
          S[l+V[0]-T[0]]=S[l];  
        for(l=1;l<=V[0];l++)  
          S[i+l-1]=V[l];  
      }  
      else //新子串长度小于原子串时:先将后部左移  
      {  
        for(l=i+V[0];l<=S[0]+V[0]-T[0];l++)  
          S[l]=S[l-V[0]+T[0]];  
        for(l=1;l<=V[0];l++)  
          S[i+l-1]=V[l];  
      }  
      S[0]=S[0]-T[0]+V[0];  
      i+=V[0];n++;  
    }//if  
  }//for  
  return n;  
}//String\_Replace   
4.18   
typedef struct {  
                     char ch;  
                     int num;  
                   } mytype;  
void StrAnalyze(Stringtype S)//统计串S中字符的种类和个数  
{  
  mytype T[MAXSIZE]; //用结构数组T存储统计结果  
  for(i=1;i<=S[0];i++)  
  {  
    c=S[i];j=0;  
    while(T[j].ch&&T[j].ch!=c) j++; //查找当前字符c是否已记录过  
    if(T[j].ch) T[j].num++;  
    else T[j]={c,1};  
  }//for  
  for(j=0;T[j].ch;j++)  
    printf("%c:  %d\n",T[j].ch,T[j].num);  
}//StrAnalyze   
4.19   
void Subtract\_String(Stringtype s,Stringtype t,Stringtype &r)//求所有包含在串s中而t中没有的字符构成的新串r  
{  
  r[0]=0;  
  for(i=1;i<=s[0];i++)  
  {  
    c=s[i];  
    for(j=1;j<i&&s[j]!=c;j++); //判断s的当前字符c是否第一次出现  
    if(i==j)  
    {  
      for(k=1;k<=t[0]&&t[k]!=c;k++); //判断当前字符是否包含在t中  
      if(k>t[0]) r[++r[0]]=c;  
    }  
  }//for  
}//Subtract\_String   
4.20  
int SubString\_Delete(Stringtype &s,Stringtype t)//从串s中删除所有与t相同的子串,并返回删除次数  
{  
  for(n=0,i=1;i<=s[0]-t[0]+1;i++)  
  {  
    for(j=1;j<=t[0]&&s[i+j-1]==t[i];j++);  
    if(j>m) //找到了与t匹配的子串  
    {  
      for(k=i;k<=s[0]-t[0];k++) s[k]=s[k+t[0]]; //左移删除  
      s[0]-=t[0];n++;  
    }  
  }//for  
  return n;  
}//Delete\_SubString   
4.21   
typedef struct{  
                char ch;  
                LStrNode \*next;  
              } LStrNode,\*LString; //链串结构   
void StringAssign(LString &s,LString t)//把串t赋值给串s  
{  
  s=malloc(sizeof(LStrNode));  
  for(q=s,p=t->next;p;p=p->next)  
  {  
    r=(LStrNode\*)malloc(sizeof(LStrNode));  
    r->ch=p->ch;  
    q->next=r;q=r;  
  }  
  q->next=NULL;  
}//StringAssign   
void StringCopy(LString &s,LString t)//把串t复制为串s.与前一个程序的区别在于,串s业已存在.  
{  
  for(p=s->next,q=t->next;p&&q;p=p->next,q=q->next)  
  {  
    p->ch=q->ch;pre=p;  
  }  
  while(q)  
  {  
    p=(LStrNode\*)malloc(sizeof(LStrNode));  
    p->ch=q->ch;  
    pre->next=p;pre=p;  
  }  
  p->next=NULL;  
}//StringCopy   
char StringCompare(LString s,LString t)//串的比较,s>t时返回正数,s=t时返回0,s<t时返回负数  
{  
  for(p=s->next,q=t->next;p&&q&&p->ch==q->ch;p=p->next,q=q->next);  
  if(!p&&!q) return 0;  
  else if(!p) return -(q->ch);  
  else if(!q) return p->ch;  
  else return p->ch-q->ch;  
}//StringCompare   
int StringLen(LString s)//求串s的长度(元素个数)  
{  
  for(i=0,p=s->next;p;p=p->next,i++);  
  return i;  
}//StringLen   
LString \* Concat(LString s,LString t)//连接串s和串t形成新串,并返回指针  
{  
  p=malloc(sizeof(LStrNode));  
  for(q=p,r=s->next;r;r=r->next)  
  {  
    q->next=(LStrNode\*)malloc(sizeof(LStrNode));  
    q=q->next;  
    q->ch=r->ch;  
  }//for //复制串s  
  for(r=t->next;r;r=r->next)  
  {  
    q->next=(LStrNode\*)malloc(sizeof(LStrNode));  
    q=q->next;  
    q->ch=r->ch;  
  }//for //复制串t  
  q->next=NULL;  
  return p;  
}//Concat   
LString \* Sub\_String(LString s,int start,int len)//返回一个串,其值等于串s从start位置起长为len的子串  
{  
  p=malloc(sizeof(LStrNode));q=p;  
  for(r=s;start;start--,r=r->next); //找到start所对应的结点指针r  
  for(i=1;i<=len;i++,r=r->next)  
  {  
    q->next=(LStrNode\*)malloc(sizeof(LStrNode));  
    q=q->next;  
    q->ch=r->ch;  
  } //复制串t  
  q->next=NULL;  
  return p;  
}//Sub\_String   
4.22   
void LString\_Concat(LString &t,LString &s,char c)//用块链存储结构,把串s插入到串t的字符c之后  
{  
  p=t.head;  
  while(p&&!(i=Find\_Char(p,c))) p=p->next; //查找字符c  
  if(!p) //没找到  
  {  
    t.tail->next=s.head;  
    t.tail=s.tail; //把s连接在t的后面  
  }  
  else  
  {  
    q=p->next;  
    r=(Chunk\*)malloc(sizeof(Chunk)); //将包含字符c的节点p分裂为两个  
    for(j=0;j<i;j++) r->ch[j]='#'; //原结点p包含c及其以前的部分  
    for(j=i;j<CHUNKSIZE;j++) //新结点r包含c以后的部分  
    {  
      r->ch[j]=p->ch[j];  
      p->ch[j]='#'; //p的后半部分和r的前半部分的字符改为无效字符'#'  
    }  
    p->next=s.head;  
    s.tail->next=r;  
    r->next=q; //把串s插入到结点p和r之间  
  }//else  
  t.curlen+=s.curlen; //修改串长  
  s.curlen=0;  
}//LString\_Concat   
int Find\_Char(Chunk \*p,char c)//在某个块中查找字符c,如找到则返回位置是第几个字符,如没找到则返回0  
{  
  for(i=0;i<CHUNKSIZE&&p->ch[i]!=c;i++);  
  if(i==CHUNKSIZE) return 0;  
  else return i+1;  
}//Find\_Char  
  
4.23   
int LString\_Palindrome(LString L)//判断以块链结构存储的串L是否为回文序列,是则返回1,否则返回0  
{  
  InitStack(S);  
  p=S.head;i=0;k=1; //i指示元素在块中的下标,k指示元素在整个序列中的序号(从1开始)  
  for(k=1;k<=S.curlen;k++)  
  {  
    if(k<=S.curlen/2) Push(S,p->ch[i]); //将前半段的字符入串  
    else if(k>(S.curlen+1)/2)  
    {  
      Pop(S,c); //将后半段的字符与栈中的元素相匹配  
      if(p->ch[i]!=c) return 0; //失配  
    }  
    if(++i==CHUNKSIZE) //转到下一个元素,当为块中最后一个元素时,转到下一块  
    {  
      p=p->next;  
      i=0;  
    }  
  }//for  
  return 1; //成功匹配  
}//LString\_Palindrome  
4.24   
void HString\_Concat(HString s1,HString s2,HString &t)//将堆结构表示的串s1和s2连接为新串t  
{  
  if(t.ch) free(t.ch);  
  t.ch=malloc((s1.length+s2.length)\*sizeof(char));  
  for(i=1;i<=s1.length;i++) t.ch[i-1]=s1.ch[i-1];  
  for(j=1;j<=s2.length;j++,i++) t.ch[i-1]=s2.ch[j-1];  
  t.length=s1.length+s2.length;  
}//HString\_Concat   
4.25   
int HString\_Replace(HString &S,HString T,HString V)//堆结构串上的置换操作,返回置换次数  
{  
  for(n=0,i=0;i<=S.length-T.length;i++)  
  {  
    for(j=i,k=0;k<T.length&&S.ch[j]==T.ch[k];j++,k++);  
    if(k==T.length) //找到了与T匹配的子串:分三种情况处理  
    {  
      if(T.length==V.length)  
        for(l=1;l<=T.length;l++) //新子串长度与原子串相同时:直接替换  
          S.ch[i+l-1]=V.ch[l-1];  
      else if(T.length<V.length) //新子串长度大于原子串时:先将后部右移  
      {  
        for(l=S.length-1;l>=i+T.length;l--)  
          S.ch[l+V.length-T.length]=S.ch[l];  
        for(l=0;l<V.length;l++)  
          S[i+l]=V[l];  
      }  
      else //新子串长度小于原子串时:先将后部左移  
      {  
        for(l=i+V.length;l<S.length+V.length-T.length;l++)  
          S.ch[l]=S.ch[l-V.length+T.length];  
        for(l=0;l<V.length;l++)  
          S[i+l]=V[l];  
      }  
      S.length+=V.length-T.length;  
      i+=V.length;n++;  
    }//if  
  }//for  
  return n;  
}//HString\_Replace   
4.26   
Status HString\_Insert(HString &S,int pos,HString T)//把T插入堆结构表示的串S的第pos个字符之前  
{  
  if(pos<1) return ERROR;  
  if(pos>S.length) pos=S.length+1;//当插入位置大于串长时,看作添加在串尾  
  S.ch=realloc(S.ch,(S.length+T.length)\*sizeof(char));  
  for(i=S.length-1;i>=pos-1;i--)  
    S.ch[i+T.length]=S.ch[i]; //后移为插入字符串让出位置  
  for(i=0;i<T.length;i++)  
    S.ch[pos+i-1]=T.ch[pos]; //插入串T  
  S.length+=T.length;  
  return OK;  
}//HString\_Insert   
4.27   
int Index\_New(Stringtype s,Stringtype t)//改进的定位算法  
{  
  i=1;j=1;  
  while(i<=s[0]&&j<=t[0])  
  {  
    if((j!=1&&s[i]==t[j])||(j==1&&s[i]==t[j]&&s[i+t[0]-1]==t[t[0]]))  
    { //当j==1即匹配模式串的第一个字符时,需同时匹配其最后一个  
      i=i+j-2;  
      j=1;  
    }  
    else  
    {  
      i++;j++;  
    }  
  }//while  
  if(j>t[0]) return i-t[0];  
}//Index\_New   
4.28   
void LGet\_next(LString &T)//链串上的get\_next算法  
{  
  p=T->succ;p->next=T;q=T;  
  while(p->succ)  
  {  
    if(q==T||p->data==q->data)  
    {  
      p=p->succ;q=q->succ;  
      p->next=q;  
    }  
    else q=q->next;  
  }//while  
}//LGet\_next   
4.29   
LStrNode \* LIndex\_KMP(LString S,LString T,LStrNode \*pos)//链串上的KMP匹配算法,返回值为匹配的子串首指针  
{  
  p=pos;q=T->succ;  
  while(p&&q)  
  {  
    if(q==T||p->chdata==q->chdata)  
    {  
      p=p->succ;  
      q=q->succ;  
    }  
    else q=q->next;  
  }//while  
  if(!q)  
  {  
    for(i=1;i<=Strlen(T);i++)  
      p=p->next;  
    return p;  
  } //发现匹配后,要往回找子串的头  
  return NULL;  
}//LIndex\_KMP   
4.30   
void Get\_LRepSub(Stringtype S)//求S的最长重复子串的位置和长度  
{  
  for(maxlen=0,i=1;i<S[0];i++)//串S2向右移i格  
  {  
    for(k=0,j=1;j<=S[0]-i;j++)//j为串S2的当前指针,此时串S1的当前指针为i+j,两指针同步移动  
    {  
      if(S[j]==S[j+i]) k++; //用k记录连续相同的字符数  
      else k=0; //失配时k归零  
      if(k>maxlen) //发现了比以前发现的更长的重复子串  
      {  
        lrs1=j-k+1;lrs2=mrs1+i;maxlen=k; //作记录  
      }  
    }//for  
  }//for  
  if(maxlen)  
  {  
    printf("Longest Repeating Substring length:%d\n",maxlen);  
    printf("Position1:%d  Position 2:%d\n",lrs1,lrs2);  
  }  
  else printf("No Repeating Substring found!\n");  
}//Get\_LRepSub  
分析:i代表"错位值".本算法的思想是,依次把串S的一个副本S2向右错位平移1格,2格,3格,...与自身S1相匹配,如果存在最长重复子串,则必然能在此过程中被发现.用变量lrs1,lrs2,maxlen来记录已发现的最长重复子串第一次出现位置,第二次出现位置和长度.题目中未说明"重复子串"是否允许有重叠部分,本算法假定允许.如不允许,只需在第二个for语句的循环条件中加上k<=i即可.本算法时间复杂度为O(Strlen(S)^2).   
4.31   
void Get\_LPubSub(Stringtype S,Stringtype T)//求串S和串T的最长公共子串位置和长度  
{  
  if(S[0]>=T[0])  
  {  
    StrAssign(A,S);StrAssign(B,T);  
  }  
  else  
  {  
    StrAssign(A,T);StrAssign(B,S);  
  } //为简化设计,令S和T中较长的那个为A,较短的那个为B  
  for(maxlen=0,i=1-B[0];i<A[0];i++)  
  {  
    if(i<0) //i为B相对于A的错位值,向左为负,左端对齐为0,向右为正  
    {  
      jmin=1;jmax=i+B[0];  
    }//B有一部分在A左端的左边  
    else if(i>A[0]-B[0])  
    {  
      jmin=i;jmax=A[0];  
    }//B有一部分在A右端的右边  
    else  
    {  
      jmin=i;jmax=i+B[0];  
    }//B在A左右两端之间.  
     //以上是根据A和B不同的相对位置确定A上需要匹配的区间(与B重合的区间)的端点:jmin,jmax.  
    for(k=0,j=jmin;j<=jmax;j++)  
    {  
      if(A[j]==B[j-i]) k++;  
      else k=0;  
      if(k>maxlen)  
      {  
        lps1=j-k+1;lps2=j-i-k+1;maxlen=k;  
      }  
    }//for  
  }//for  
  if(maxlen)  
  {  
    if(S[0]>=T[0])  
    {  
      lpsS=lps1;lpsT=lps2;  
    }  
    else  
    {  
      lpsS=lps2;lpsT=lps1;  
    } //将A,B上的位置映射回S,T上的位置  
    printf("Longest Public Substring length:%d\n",maxlen);  
    printf("Position in S:%d  Position in T:%d\n",lpsS,lpsT);  
  }//if  
  else printf("No Repeating Substring found!\n");  
}//Get\_LPubSub  
分析:本题基本思路与上题同.唯一的区别是,由于A,B互不相同,因此B不仅要向右错位,而且还要向左错位,以保证不漏掉一些情况.当B相对于A的位置不同时,需要匹配的区间的计算公式也各不相同,请读者自己画图以帮助理解.本算法的时间复杂度是o（strlrn（s）\*strlen（t））。

第五章 数组和广义表  
  
5.18   
void RSh(int A[n],int k)//把数组A的元素循环右移k位,只用一个辅助存储空间  
{  
  for(i=1;i<=k;i++)  
    if(n%i==0&&k%i==0) p=i;//求n和k的最大公约数p  
  for(i=0;i<p;i++)   
  {  
    j=i;l=(i+n-k)%n;temp=A[i];  
    while(l!=i)  
    {  
      A[j]=A[l];  
      j=l;l=(j+n-k)%n;  
    }// 循环右移一步  
    A[j]=temp;  
  }//for  
}//RSh  
分析:要把A的元素循环右移k位,则A[0]移至A[k],A[k]移至A[2k]......直到最终回到A[0].然而这并没有全部解决问题,因为有可能有的元素在此过程中始终没有被访问过,而是被跳了过去.分析可知,当n和k的最大公约数为p时,只要分别以A[0],A[1],...A[p-1]为起点执行上述算法,就可以保证每一个元素都被且仅被右移一次,从而满足题目要求.也就是说,A的所有元素分别处在p个"循环链"上面.举例如下:  
n=15,k=6,则p=3.  
第一条链:A[0]->A[6],A[6]->A[12],A[12]->A[3],A[3]->A[9],A[9]->A[0].  
第二条链:A[1]->A[7],A[7]->A[13],A[13]->A[4],A[4]->A[10],A[10]->A[1].  
第三条链:A[2]->A[8],A[8]->A[14],A[14]->A[5],A[5]->A[11],A[11]->A[2].  
恰好使所有元素都右移一次.  
虽然未经数学证明,但作者相信上述规律应该是正确的.   
5.19   
void Get\_Saddle(int A[m][n])//求矩阵A中的马鞍点  
{  
  for(i=0;i<m;i++)  
  {  
    for(min=A[i][0],j=0;j<n;j++)  
      if(A[i][j]<min) min=A[i][j]; //求一行中的最小值  
    for(j=0;j<n;j++)  
      if(A[i][j]==min) //判断这个(些)最小值是否鞍点  
      {  
        for(flag=1,k=0;k<m;k++)  
          if(min<A[k][j]) flag=0;  
        if(flag)  
          printf("Found a saddle element!\nA[%d][%d]=%d",i,j,A[i][j]);  
      }  
  }//for  
}//Get\_Saddle   
5.20   
int exps[MAXSIZE]; //exps数组用于存储某一项的各变元的指数  
int maxm,n; //maxm指示变元总数,n指示一个变元的最高指数   
void Print\_Poly\_Descend(int \*a,int m)//按降幂顺序输出m元多项式的项，各项的系数已经按照题目要求存储于m维数组中，数组的头指针为a  
{  
  maxm=m;  
  for(i=m\*n;i>=0;i--) //按降幂次序,可能出现的最高项次数为mn  
    Get\_All(a,m,i,0); //确定并输出所有次数为i的项  
}//Print\_Poly\_Descend   
void Get\_All(int \*a,int m,int i,int seq)//递归求出所有和为i的m个自然数  
{  
  if(seq==maxm) Print\_Nomial(a,exps); //已经求完时,输出该项  
  else  
  {  
    min=i-(m-1)\*n; //当前数不能小于min  
    if(min<0) min=0;  
    max=n<i?n:i; //当前数不能大于max  
    for(j=min;j<=max;j++)  
    {  
      exps[seq]=j; //依次取符合条件的数  
      Get\_All(a,m-1,i-j,seq+1); //取下一个数  
    }  
  }//else  
  exps[seq]=0; //返回  
}//Get\_All   
void Print\_Nomial(int \*a,int exps[ ])//输出一个项,项的各变元的指数已经存储在数组exps中  
{  
  pos=0;  
  for(i=0;i<maxm;i++) //求出该项的系数在m维数组a中低下标优先的存储位置pos  
  {  
    pos\*=n;  
    pos+=exps[i];  
  }  
  coef=\*(a+pos); //取得该系数coef  
  if(!coef) return; //该项为0时无需输出  
  else if(coef>0) printf("+"); //系数为正时打印加号  
  else if(coef<0) printf("-"); //系数为负时打印减号  
  if(abs(coef)!=1) printf("%d",abs(coef)); //当系数的绝对值不为1时打印系数  
  for(i=0;i<maxm;i++)  
    if(exps[i]) //打印各变元及其系数  
    {  
      printf("x");  
      printf("%d",i);  
      printf("E");  
      if(exps[i]>1) printf("%d",exp[i]); //系数为1时无需打印  
    }  
}//Print\_Nomial  
分析:本算法的关键在于如何按照降幂顺序输出各项.这里采用了一个递归函数来找到所有满足和为i的m个自然数作为各变元的指数.只要先取第一个数为j,然后再找到所有满足和为i-j的m-1个自然数就行了.要注意j的取值范围必须使剩余m-1个自然数能够找到,所以不能小于i-(m-1)\*maxn,也不能大于i.只要找到了一组符合条件的数,就可以在存储多项式系数的数组中确定对应的项的系数的位置,并且在系数不为0时输出对应的项.  
  
5.21   
void TSMatrix\_Add(TSMatrix A,TSMatrix B,TSMatrix &C)//三元组表示的稀疏矩阵加法  
{  
  C.mu=A.mu;C.nu=A.nu;C.tu=0;  
  pa=1;pb=1;pc=1;  
  for(x=1;x<=A.mu;x++) //对矩阵的每一行进行加法  
  {  
    while(A.data[pa].i<x) pa++;  
    while(B.data[pb].i<x) pb++;  
    while(A.data[pa].i==x&&B.data[pb].i==x)//行列值都相等的元素  
    {  
      if(A.data[pa].j==B.data[pb].j)  
      {  
        ce=A.data[pa].e+B.data[pb].e;  
        if(ce) //和不为0  
        {  
          C.data[pc].i=x;  
          C.data[pc].j=A.data[pa].j;  
          C.data[pc].e=ce;  
          pa++;pb++;pc++;  
        }  
      }//if  
      else if(A.data[pa].j>B.data[pb].j)   
      {  
        C.data[pc].i=x;  
        C.data[pc].j=B.data[pb].j;  
        C.data[pc].e=B.data[pb].e;  
        pb++;pc++;  
      }  
      else  
      {  
        C.data[pc].i=x;  
        C.data[pc].j=A.data[pa].j;  
        C.data[pc].e=A.data[pa].e  
        pa++;pc++;  
      }  
    }//while  
    while(A.data[pa]==x) //插入A中剩余的元素(第x行)  
    {  
      C.data[pc].i=x;  
      C.data[pc].j=A.data[pa].j;  
      C.data[pc].e=A.data[pa].e  
      pa++;pc++;  
    }  
    while(B.data[pb]==x) //插入B中剩余的元素(第x行)  
    {  
      C.data[pc].i=x;  
      C.data[pc].j=B.data[pb].j;  
      C.data[pc].e=B.data[pb].e;  
      pb++;pc++;  
    }  
  }//for  
  C.tu=pc;  
}//TSMatrix\_Add   
5.22   
void TSMatrix\_Addto(TSMatrix &A,TSMatrix B)//将三元组矩阵B加到A上  
{  
  for(i=1;i<=A.tu;i++)  
    A.data[MAXSIZE-A.tu+i]=A.data[i];/把A的所有元素都移到尾部以腾出位置  
  pa=MAXSIZE-A.tu+1;pb=1;pc=1;  
  for(x=1;x<=A.mu;x++) //对矩阵的每一行进行加法  
  {  
    while(A.data[pa].i<x) pa++;  
    while(B.data[pb].i<x) pb++;  
    while(A.data[pa].i==x&&B.data[pb].i==x)//行列值都相等的元素  
    {  
      if(A.data[pa].j==B.data[pb].j)  
      {  
        ne=A.data[pa].e+B.data[pb].e;  
        if(ne) //和不为0  
        {  
          A.data[pc].i=x;  
          A.data[pc].j=A.data[pa].j;  
          A.data[pc].e=ne;  
          pa++;pb++;pc++;  
        }  
      }//if  
      else if(A.data[pa].j>B.data[pb].j)   
      {  
        A.data[pc].i=x;  
        A.data[pc].j=B.data[pb].j;  
        A.data[pc].e=B.data[pb].e;  
        pb++;pc++;  
      }  
      else  
      {  
        A.data[pc].i=x;  
        A.data[pc].j=A.data[pa].j;  
        A.data[pc].e=A.data[pa].e  
        pa++;pc++;  
      }  
    }//while  
    while(A.data[pa]==x) //插入A中剩余的元素(第x行)  
    {  
      A.data[pc].i=x;  
      A.data[pc].j=A.data[pa].j;  
      A.data[pc].e=A.data[pa].e  
      pa++;pc++;  
    }  
    while(B.data[pb]==x) //插入B中剩余的元素(第x行)  
    {  
      A.data[pc].i=x;  
      A.data[pc].j=B.data[pb].j;  
      A.data[pc].e=B.data[pb].e;  
      pb++;pc++;  
    }  
  }//for  
  A.tu=pc;  
  for(i=A.tu;i<MAXSIZE;i++) A.data[i]={0,0,0}; //清除原来的A中记录  
}//TSMatrix\_Addto   
5.23   
typedef struct{  
                    int j;  
                    int e;  
                  } DSElem;   
typedef struct{  
                DSElem data[MAXSIZE];  
                int cpot[MAXROW];//这个向量存储每一行在二元组中的起始位置  
                int mu,nu,tu;  
              } DSMatrix; //二元组矩阵类型   
Status DSMatrix\_Locate(DSMatrix A,int i,int j,int &e)//求二元组矩阵的元素A[i][j]的值e  
{  
  for(s=A.cpot[i];s<A.cpot[i+1]&&A.data[s].j!=j;s++);//注意查找范围  
  if(s<A.cpot[i+1]&&A.data[s].j==j) //找到了元素A[i][j]  
  {  
    e=A.data[s];  
    return OK;  
  }  
  return ERROR;  
}//DSMatrix\_Locate   
5.24   
typedef struct{  
                    int seq; //该元素在以行为主序排列时的序号  
                    int e;  
                  } SElem;   
typedef struct{  
                    SElem data[MAXSIZE];  
                    int mu,nu,tu;  
                  } SMatrix; //单下标二元组矩阵类型   
Status SMatrix\_Locate(SMatrix A,int i,int j,int &e)//求单下标二元组矩阵的元素A[i][j]的值e  
{  
  s=i\*A.nu+j+1;p=1;  
  while(A.data[p].seq<s) p++; //利用各元素seq值逐渐递增的特点  
  if(A.data[p].seq==s) //找到了元素A[i][j]  
  {  
    e=A.data[p].e;  
    return OK;  
  }  
  return ERROR;  
}//SMatrix\_Locate   
5.25   
typedef enum{0,1} bool;   
typedef struct{  
                    int mu,nu;  
                    int elem[MAXSIZE];  
                    bool map[mu][nu];  
                  } BMMatrix; //用位图表示的矩阵类型   
void BMMatrix\_Add(BMMatrix A,BMMatrix B,BMMatrix &C)//位图矩阵的加法  
{  
  C.mu=A.mu;C.nu=A.nu;  
  pa=1;pb=1;pc=1;  
  for(i=0;i<A.mu;i++) //每一行的相加  
    for(j=0;j<A.nu;j++) //每一个元素的相加  
    {  
      if(A.map[i][j]&&B.map[i][j]&&(A.elem[pa]+B.elem[pb]))//结果不为0  
      {  
        C.elem[pc]=A.elem[pa]+B.elem[pb];  
        C.map[i][j]=1;  
        pa++;pb++;pc++;  
      }  
      else if(A.map[i][j]&&!B.map[i][j])  
      {  
        C.elem[pc]=A.elem[pa];  
        C.map[i][j]=1;  
        pa++;pc++;  
      }  
      else if(!A.map[i][j]&&B.map[i][j])  
      {  
        C.elem[pc]=B.elem[pb];  
        C.map[i][j]=1;  
        pb++;pc++;  
      }  
    }  
}//BMMatrix\_Add   
5.26   
void Print\_OLMatrix(OLMatrix A)//以三元组格式输出十字链表表示的矩阵  
{  
  for(i=0;i<A.mu;i++)  
  {  
    if(A.rhead[i])  
      for(p=A.rhead[i];p;p=p->right) //逐次遍历每一个行链表  
        printf("%d %d %d\n",i,p->j,p->e;  
  }  
}//Print\_OLMatrix   
5.27   
void OLMatrix\_Add(OLMatrix &A,OLMatrix B)//把十字链表表示的矩阵B加到A上  
{  
  for(j=1;j<=A.nu;j++) cp[j]=A.chead[j]; //向量cp存储每一列当前最后一个元素的指针  
  for(i=1;i<=A.mu;i++)  
  {  
    pa=A.rhead[i];pb=B.rhead[i];pre=NULL;  
    while(pb)  
    {  
      if(pa==NULL||pa->j>pb->j) //新插入一个结点  
      {  
        p=(OLNode\*)malloc(sizeof(OLNode));  
        if(!pre) A.rhead[i]=p;  
        else pre->right=p;  
        p->right=pa;pre=p;  
        p->i=i;p->j=pb->j;p->e=pb->e; //插入行链表中  
        if(!A.chead[p->j])  
        {  
          A.chead[p->j]=p;  
          p->down=NULL;  
        }  
        else  
        {  
          while(cp[p->j]->down) cp[p->j]=cp[p->j]->down;  
          p->down=cp[p->j]->down;  
          cp[p->j]->down=p;  
        }  
        cp[p->j]=p; //插入列链表中  
      }//if  
      else if(pa->j<pb->j)  
      {  
        pre=pa;  
        pa=pa->right;  
      } //pa右移一步  
      else if(pa->e+pb->e)  
      {  
        pa->e+=pb->e;  
        pre=pa;pa=pa->right;  
        pb=pb->right;  
      } //直接相加  
      else  
      {  
        if(!pre) A.rhead[i]=pa->right;  
        else pre->right=pa->right;  
        p=pa;pa=pa->right; //从行链表中删除  
        if(A.chead[p->j]==p)  
          A.chead[p->j]=cp[p->j]=p->down;  
        else cp[p->j]->down=p->down; //从列链表中删除  
        free (p);  
      }//else  
    }//while  
  }//for  
}//OLMatrix\_Add  
分析:本题的具体思想在课本中有详细的解释说明.   
5.28   
void MPList\_PianDao(MPList &L)//对广义表存储结构的多元多项式求第一变元的偏导  
{  
  for(p=L->hp->tp;p&&p->exp;pre=p,p=p->tp)  
  {  
    if(p->tag) Mul(p->hp,p->exp);  
    else p->coef\*=p->exp; //把指数乘在本结点或其下属结点上  
    p->exp--;  
  }  
  pre->tp=NULL;  
  if(p) free (p); //删除可能存在的常数项  
}//MPList\_PianDao   
void Mul(MPList &L,int x)//递归算法,对多元多项式L乘以x  
{  
  for(p=L;p;p=p->tp)  
  {  
    if(!p->tag) p->coef\*=x;  
    else Mul(p->hp,x);  
  }  
}//Mul  
      
5.29   
void MPList\_Add(MPList A,MPList B,MPList &C)//广义表存储结构的多项式相加的递归算法  
{  
  C=(MPLNode\*)malloc(sizeof(MPLNode));  
  if(!A->tag&&!B->tag) //原子项,可直接相加  
  {  
    C->coef=A->coef+B->coef;  
    if(!C->coef)  
    {  
      free(C);  
      C=NULL;  
    }  
  }//if  
  else if(A->tag&&B->tag) //两个多项式相加  
  {  
    p=A;q=B;pre=NULL;  
    while(p&&q)  
    {  
      if(p->exp==q->exp)  
      {  
        C=(MPLNode\*)malloc(sizeof(MPLNode));  
        C->exp=p->exp;  
        MPList\_Add(A->hp,B->hp,C->hp);  
        pre->tp=C;pre=C;  
        p=p->tp;q=q->tp;  
      }  
      else if(p->exp>q->exp)  
      {  
        C=(MPLNode\*)malloc(sizeof(MPLNode));  
        C->exp=p->exp;  
        C->hp=A->hp;  
        pre->tp=C;pre=C;  
        p=p->tp;  
      }  
      else  
      {  
        C=(MPLNode\*)malloc(sizeof(MPLNode));  
        C->exp=q->exp;  
        C->hp=B->hp;  
        pre->tp=C;pre=C;  
        q=q->tp;  
      }  
    }//while  
    while(p)  
    {  
      C=(MPLNode\*)malloc(sizeof(MPLNode));  
      C->exp=p->exp;  
      C->hp=p->hp;  
      pre->tp=C;pre=C;  
      p=p->tp;  
    }  
    while(q)  
    {  
      C=(MPLNode\*)malloc(sizeof(MPLNode));  
      C->exp=q->exp;  
      C->hp=q->hp;  
      pre->tp=C;pre=C;  
      q=q->tp;  
    } //将其同次项分别相加得到新的多项式,原理见第二章多项式相加一题  
  }//else if  
  else if(A->tag&&!B->tag) //多项式和常数项相加  
  {  
    x=B->coef;  
    for(p=A;p->tp->tp;p=p->tp);  
    if(p->tp->exp==0) p->tp->coef+=x; //当多项式中含有常数项时,加上常数项  
    if(!p->tp->coef)  
    {  
      free(p->tp);  
      p->tp=NULL;  
    }  
    else  
    {  
      q=(MPLNode\*)malloc(sizeof(MPLNode));  
      q->coef=x;q->exp=0;  
      q->tag=0;q->tp=NULL;  
      p->tp=q;  
    } //否则新建常数项,下同  
  }//else if  
  else  
  {  
    x=A->coef;  
    for(p=B;p->tp->tp;p=p->tp);  
    if(p->tp->exp==0) p->tp->coef+=x;  
    if(!p->tp->coef)  
    {  
      free(p->tp);  
      p->tp=NULL;  
    }  
    else  
    {  
      q=(MPLNode\*)malloc(sizeof(MPLNode));  
      q->coef=x;q->exp=0;  
      q->tag=0;q->tp=NULL;  
      p->tp=q;  
    }  
  }//else  
}//MPList\_Add   
5.30   
int GList\_Getdeph(GList L)//求广义表深度的递归算法  
{  
  if(!L->tag) return 0; //原子深度为0  
  else if(!L) return 1; //空表深度为1  
  m=GList\_Getdeph(L->ptr.hp)+1;  
  n=GList\_Getdeph(L->ptr.tp);  
  return m>n?m:n;  
}//GList\_Getdeph   
5.31   
void GList\_Copy(GList A,GList &B)//复制广义表的递归算法  
{  
  if(!A->tag) //当结点为原子时,直接复制  
  {  
    B->tag=0;  
    B->atom=A->atom;  
  }  
  else //当结点为子表时  
  {  
    B->tag=1;  
    if(A->ptr.hp)  
    {  
      B->ptr.hp=malloc(sizeof(GLNode));  
      GList\_Copy(A->ptr.hp,B->ptr.hp);  
    } //复制表头  
    if(A->ptr.tp)  
    {  
      B->ptr.tp=malloc(sizeof(GLNode));  
      GList\_Copy(A->ptr.tp,B->ptr.tp);  
    } //复制表尾  
  }//else  
}//GList\_Copy   
5.32   
int GList\_Equal(GList A,GList B)//判断广义表A和B是否相等,是则返回1,否则返回0  
{ //广义表相等可分三种情况:  
  if(!A&&!B) return 1; //空表是相等的  
  if(!A->tag&&!B->tag&&A->atom==B->atom) return 1;//原子的值相等  
  if(A->tag&&B->tag)  
    if(GList\_Equal(A->ptr.hp,B->ptr.hp)&&GList\_Equal(A->ptr.tp,B->ptr.tp))  
      return 1; //表头表尾都相等  
  return 0;  
}//GList\_Equal   
5.33   
void GList\_PrintElem(GList A,int layer)//递归输出广义表的原子及其所在层次,layer表示当前层次  
{  
  if(!A) return;  
  if(!A->tag) printf("%d %d\n",A->atom,layer);  
  else  
  {  
    GList\_PrintElem(A->ptr.hp,layer+1);  
    GList\_PrintElem(A->ptr.tp,layer); //注意尾表与原表是同一层次  
  }  
}//GList\_PrintElem   
5.34   
void GList\_Reverse(GList A)//递归逆转广义表A  
{  
  GLNode \*ptr[MAX\_SIZE];  
  if(A->tag&&A->ptr.tp) //当A不为原子且表尾非空时才需逆转  
  {  
    for(i=0,p=A;p;p=p->ptr.tp,i++)  
    {  
      if(p->ptr.hp) GList\_Reverse(p->ptr.hp);        //逆转各子表  
      ptr[i]=p->ptr.hp;   
    }  
    for(p=A;p;p=p->ptr.tp)        //重新按逆序排列各子表的顺序  
      p->ptr.hp=ptr[--i];  
  }  
}//GList\_Reverse   
5.35   
Status Create\_GList(GList &L)//根据输入创建广义表L,并返回指针  
{  
  scanf("%c",&ch);  
  if(ch==' ')  
  {  
    L=NULL;  
    scanf("%c",&ch);  
    if(ch!=')') return ERROR;  
    return OK;  
  }  
  L=(GList)malloc(sizeof(GLNode));  
  L->tag=1;  
  if(isalphabet(ch)) //输入是字母  
  {  
    p=(GList)malloc(sizeof(GLNode)); //建原子型表头  
    p->tag=0;p->atom=ch;  
    L->ptr.hp=p;  
  }  
  else if(ch=='(') Create\_GList(L->ptr.hp); //建子表型表头  
  else return ERROR;  
  scanf ("%c",&ch);  
  if(ch==')') L->ptr.tp=NULL;  
  else if(ch==',') Create\_GList(L->ptr.tp); //建表尾   
  else return ERROR;  
  return OK;  
}//Create\_GList  
分析:本题思路见书后解答.   
5.36   
void GList\_PrintList(GList A)//按标准形式输出广义表  
{  
  if(!A) printf("()"); //空表  
  else if(!A->tag) printf("%d",A->atom);//原子  
  else  
  {  
    printf("(");  
    for(p=A;p;p=p->ptr.tp)   
    {  
      GList\_PrintList(p->ptr.hp);  
       if(p->ptr.tp) printf(",");        //只有当表尾非空时才需要打印逗号  
    }   
    printf(")");  
  }//else  
}//GList\_PrintList   
5.37   
void GList\_DelElem(GList &A,int x)//从广义表A中删除所有值为x的原子  
{  
  if(A&&A->ptr.hp)   
  {  
    if(A->ptr.hp->tag) GList\_DelElem(A->ptr.hp,x);  
    else if(!A->ptr.hp->tag&&A->ptr.hp->atom==x)  
    {  
      q=A;  
      A=A->ptr.tp;    //删去元素值为x的表头  
      free(q);  
      GList\_DelElem(A,x);  
    }  
  }  
  if(A&&A->ptr.tp) GList\_DelElem(A->ptr.tp,x);  
}//GList\_DelElem   
5.39   
void GList\_PrintElem\_LOrder(GList A)//按层序输出广义表A中的所有元素  
{  
  InitQueue(Q);  
  for(p=L;p;p=p->ptr.tp) EnQueue(Q,p);  
  while(!QueueEmpty(Q))  
  {  
    DeQueue(Q,r);  
    if(!r->tag) printf("%d",r->atom);  
    else  
      for(r=r->ptr.hp;r;r=r->ptr.tp) EnQueue(Q,r);   
  }//while  
}//GList\_PrintElem\_LOrder   
分析:层序遍历的问题,一般都是借助队列来完成的,每次从队头取出一个元素的同时把它下一层的孩子插入队尾.这是层序遍历的基本思想.

第六章 树和二叉树  
  
6.33   
int Is\_Descendant\_C(int u,int v)//在孩子存储结构上判断u是否v的子孙,是则返回1,否则返回0  
{  
  if(u==v) return 1;  
  else  
  {  
    if(L[v])  
      if (Is\_Descendant(u,L[v])) return 1;  
    if(R[v])  
      if (Is\_Descendant(u,R[v])) return 1; //这是个递归算法  
  }  
  return 0;  
}//Is\_Descendant\_C   
6.34   
int Is\_Descendant\_P(int u,int v)//在双亲存储结构上判断u是否v的子孙,是则返回1,否则返回0  
{  
  for(p=u;p!=v&&p;p=T[p]);  
  if(p==v) return 1;  
  else return 0;  
}//Is\_Descendant\_P   
6.35   
这一题根本不需要写什么算法,见书后注释:两个整数的值是相等的.   
6.36   
int Bitree\_Sim(Bitree B1,Bitree B2)//判断两棵树是否相似的递归算法  
{  
  if(!B1&&!B2) return 1;  
  else if(B1&&B2&&Bitree\_Sim(B1->lchild,B2->lchild)&&Bitree\_Sim(B1->rchild,B2->rchild))  
    return 1;  
  else return 0;  
}//Bitree\_Sim   
6.37   
void PreOrder\_Nonrecursive(Bitree T)//先序遍历二叉树的非递归算法  
{  
  InitStack(S);  
  Push(S,T); //根指针进栈  
  while(!StackEmpty(S))  
  {  
    while(Gettop(S,p)&&p)  
    {  
      visit(p->data);  
      push(S,p->lchild);  
    } //向左走到尽头  
    pop(S,p);  
    if(!StackEmpty(S))  
    {  
     pop(S,p);  
     push(S,p->rchild); //向右一步  
    }  
  }//while  
}//PreOrder\_Nonrecursive   
6.38   
typedef struct {  
                     BTNode\* ptr;  
                     enum {0,1,2} mark;  
                   } PMType; //有mark域的结点指针类型   
void PostOrder\_Stack(BiTree T)//后续遍历二叉树的非递归算法,用栈  
{  
  PMType a;  
  InitStack(S); //S的元素为PMType类型  
  Push (S,{T,0}); //根结点入栈  
  while(!StackEmpty(S))  
  {  
    Pop(S,a);  
    switch(a.mark)  
    {  
      case 0:  
        Push(S,{a.ptr,1}); //修改mark域  
        if(a.ptr->lchild) Push(S,{a.ptr->lchild,0}); //访问左子树  
        break;  
      case 1:  
        Push(S,{a.ptr,2}); //修改mark域  
        if(a.ptr->rchild) Push(S,{a.ptr->rchild,0}); //访问右子树  
        break;  
      case 2:  
        visit(a.ptr); //访问结点,返回  
    }  
  }//while  
}//PostOrder\_Stack  
分析:为了区分两次过栈的不同处理方式,在堆栈中增加一个mark域,mark=0表示刚刚访问此结点,mark=1表示左子树处理结束返回,mark=2表示右子树处理结束返回.每次根据栈顶元素的mark域值决定做何种动作.   
6.39   
typedef struct {  
                     int data;  
                     EBTNode \*lchild;  
                     EBTNode \*rchild;  
                     EBTNode \*parent;  
                     enum {0,1,2} mark;  
                  } EBTNode,EBitree; //有mark域和双亲指针域的二叉树结点类型   
void PostOrder\_Nonrecursive(EBitree T)//后序遍历二叉树的非递归算法,不用栈  
{  
  p=T;  
  while(p)  
    switch(p->mark)  
    {  
      case 0:  
        p->mark=1;  
        if(p->lchild) p=p->lchild; //访问左子树  
        break;  
      case 1:  
        p->mark=2;  
        if(p->rchild) p=p->rchild; //访问右子树  
        break;  
      case 2:  
        visit(p);  
        p->mark=0; //恢复mark值  
        p=p->parent; //返回双亲结点  
    }  
}//PostOrder\_Nonrecursive  
分析:本题思路与上一题完全相同,只不过结点的mark值是储存在结点中的,而不是暂存在堆栈中,所以访问完毕后要将mark域恢复为0,以备下一次遍历.   
6.40   
typedef struct {  
                     int data;  
                     PBTNode \*lchild;  
                     PBTNode \*rchild;  
                     PBTNode \*parent;  
                   } PBTNode,PBitree; //有双亲指针域的二叉树结点类型   
void Inorder\_Nonrecursive(PBitree T)//不设栈非递归遍历有双亲指针的二叉树  
{  
  p=T;  
  while(p->lchild) p=p->lchild; //向左走到尽头  
  while(p)  
  {  
    visit(p);  
    if(p->rchild) //寻找中序后继:当有右子树时  
    {  
      p=p->rchild;  
      while(p->lchild) p=p->lchild; //后继就是在右子树中向左走到尽头  
    }  
    else if(p->parent->lchild==p) p=p->parent; //当自己是双亲的左孩子时后继就是双亲  
    else  
    {  
      p=p->parent;  
      while(p->parent&&p->parent->rchild==p) p=p->parent;  
      p=p->parent;  
    } //当自己是双亲的右孩子时后继就是向上返回直到遇到自己是在其左子树中的祖先  
  }//while  
}//Inorder\_Nonrecursive   
6.41   
int c,k; //这里把k和计数器c作为全局变量处理   
void Get\_PreSeq(Bitree T)//求先序序列为k的结点的值  
{  
  if(T)  
  {  
    c++; //每访问一个子树的根都会使前序序号计数器加1  
    if(c==k)  
    {  
      printf("Value is %d\n",T->data);  
      exit (1);  
    }  
    else  
    {  
      Get\_PreSeq(T->lchild); //在左子树中查找  
      Get\_PreSeq(T->rchild); //在右子树中查找  
    }  
  }//if  
}//Get\_PreSeq   
main()  
{  
  ...  
  scanf("%d",&k);  
  c=0; //在主函数中调用前,要给计数器赋初值0  
  Get\_PreSeq(T,k);  
  ...  
}//main   
6.42   
int LeafCount\_BiTree(Bitree T)//求二叉树中叶子结点的数目  
{  
  if(!T) return 0; //空树没有叶子  
  else if(!T->lchild&&!T->rchild) return 1; //叶子结点  
  else return Leaf\_Count(T->lchild)+Leaf\_Count(T->rchild);//左子树的叶子数加上右子树的叶子数  
}//LeafCount\_BiTree   
6.43   
void Bitree\_Revolute(Bitree T)//交换所有结点的左右子树  
{  
  T->lchild<->T->rchild; //交换左右子树  
  if(T->lchild) Bitree\_Revolute(T->lchild);  
  if(T->rchild) Bitree\_Revolute(T->rchild); //左右子树再分别交换各自的左右子树  
}//Bitree\_Revolute   
6.44   
int Get\_Sub\_Depth(Bitree T,int x)//求二叉树中以值为x的结点为根的子树深度  
{  
  if(T->data==x)  
  {  
    printf("%d\n",Get\_Depth(T)); //找到了值为x的结点,求其深度  
    exit 1;  
  }  
  else  
  {  
    if(T->lchild) Get\_Sub\_Depth(T->lchild,x);  
    if(T->rchild) Get\_Sub\_Depth(T->rchild,x); //在左右子树中继续寻找  
  }  
}//Get\_Sub\_Depth   
int Get\_Depth(Bitree T)//求子树深度的递归算法  
{  
  if(!T) return 0;  
  else  
  {  
    m=Get\_Depth(T->lchild);  
    n=Get\_Depth(T->rchild);  
    return (m>n?m:n)+1;  
  }  
}//Get\_Depth   
6.45   
void Del\_Sub\_x(Bitree T,int x)//删除所有以元素x为根的子树  
{  
  if(T->data==x) Del\_Sub(T); //删除该子树  
  else  
  {  
    if(T->lchild) Del\_Sub\_x(T->lchild,x);  
    if(T->rchild) Del\_Sub\_x(T->rchild,x); //在左右子树中继续查找  
  }//else  
}//Del\_Sub\_x   
void Del\_Sub(Bitree T)//删除子树T  
{  
  if(T->lchild) Del\_Sub(T->lchild);  
  if(T->rchild) Del\_Sub(T->rchild);  
  free(T);  
}//Del\_Sub   
6.46   
void Bitree\_Copy\_Nonrecursive(Bitree T,Bitree &U)//非递归复制二叉树  
{  
  InitStack(S1);InitStack(S2);  
  push(S1,T); //根指针进栈  
  U=(BTNode\*)malloc(sizeof(BTNode));  
  U->data=T->data;  
  q=U;push(S2,U);  
  while(!StackEmpty(S))  
  {  
    while(Gettop(S1,p)&&p)  
    {  
      q->lchild=(BTNode\*)malloc(sizeof(BTNode));  
      q=q->lchild;q->data=p->data;  
      push(S1,p->lchild);  
      push(S2,q);  
    } //向左走到尽头  
    pop(S1,p);  
    pop(S2,q);  
    if(!StackEmpty(S1))  
    {  
     pop(S1,p);pop(S2,q);  
     q->rchild=(BTNode\*)malloc(sizeof(BTNode));  
     q=q->rchild;q->data=p->data;  
     push(S1,p->rchild); //向右一步  
     push(S2,q);  
    }  
  }//while  
}//BiTree\_Copy\_Nonrecursive  
分析:本题的算法系从6.37改写而来.   
6.47   
void LayerOrder(Bitree T)//层序遍历二叉树  
{  
  InitQueue(Q); //建立工作队列  
  EnQueue(Q,T);  
  while(!QueueEmpty(Q))  
  {  
    DeQueue(Q,p);  
    visit(p);  
    if(p->lchild) EnQueue(Q,p->lchild);  
    if(p->rchild) EnQueue(Q,p->rchild);  
  }  
}//LayerOrder   
6.48   
int found=FALSE;   
Bitree\* Find\_Near\_Ancient(Bitree T,Bitree p,Bitree q)//求二叉树T中结点p和q的最近共同祖先  
{  
  Bitree pathp[ 100 ],pathq[ 100 ] //设立两个辅助数组暂存从根到p,q的路径  
  Findpath(T,p,pathp,0);  
  found=FALSE;  
  Findpath(T,q,pathq,0); //求从根到p,q的路径放在pathp和pathq中  
  for(i=0;pathp[i]==pathq[i]&&pathp[i];i++); //查找两条路径上最后一个相同结点  
  return pathp[--i];  
}//Find\_Near\_Ancient   
void Findpath(Bitree T,Bitree p,Bitree path[ ],int i)//求从T到p路径的递归算法  
{  
  if(T==p)  
  {  
    found=TRUE;  
    return; //找到  
  }  
  path[i]=T; //当前结点存入路径  
  if(T->lchild) Findpath(T->lchild,p,path,i+1); //在左子树中继续寻找  
  if(T->rchild&&!found) Findpath(T->rchild,p,path,i+1); //在右子树中继续寻找  
  if(!found) path[i]=NULL; //回溯  
}//Findpath   
6.49   
int IsFull\_Bitree(Bitree T)//判断二叉树是否完全二叉树,是则返回1,否则返回0  
{  
  InitQueue(Q);  
  flag=0;  
  EnQueue(Q,T); //建立工作队列  
  while(!QueueEmpty(Q))  
  {  
    DeQueue(Q,p);  
    if(!p) flag=1;  
    else if(flag) return 0;  
    else  
    {  
      EnQueue(Q,p->lchild);  
      EnQueue(Q,p->rchild); //不管孩子是否为空,都入队列  
    }  
  }//while  
  return 1;  
}//IsFull\_Bitree  
分析:该问题可以通过层序遍历的方法来解决.与6.47相比,作了一个修改,不管当前结点是否有左右孩子,都入队列.这样当树为完全二叉树时,遍历时得到是一个连续的不包含空指针的序列.反之,则序列中会含有空指针.   
6.50   
Status CreateBitree\_Triplet(Bitree &T)//输入三元组建立二叉树  
{  
  if(getchar()!='^') return ERROR;  
  T=(BTNode\*)malloc(sizeof(BTNode));  
  p=T;p->data=getchar();  
  getchar(); //滤去多余字符  
  InitQueue(Q);  
  EnQueue(Q,T);  
  while((parent=getchar())!='^'&&(child=getchar())&&(side=getchar()))  
  {  
    while(QueueHead(Q)!=parent&&!QueueEmpty(Q)) DeQueue(Q,e);  
    if(QueueEmpty(Q)) return ERROR; //未按层序输入  
    p=QueueHead(Q);  
    q=(BTNode\*)malloc(sizeof(BTNode));  
    if(side=='L') p->lchild=q;  
    else if(side=='R') p->rchild=q;  
    else return ERROR; //格式不正确  
    q->data=child;  
    EnQueue(Q,q);  
  }  
  return OK;  
}//CreateBitree\_Triplet   
6.51   
Status Print\_Expression(Bitree T)//按标准形式输出以二叉树存储的表达式  
{  
  if(T->data是字母) printf("%c",T->data);  
  else if(T->data是操作符)  
  {  
    if(!T->lchild||!T->rchild) return ERROR; //格式错误  
    if(T->lchild->data是操作符&&T->lchild->data优先级低于T->data)  
    {  
      printf("(");  
      if(!Print\_Expression(T->lchild)) return ERROR;  
      printf(")");  
    } //注意在什么情况下要加括号  
    else if(!Print\_Expression(T->lchild)) return ERROR;  
    if(T->rchild->data是操作符&&T->rchild->data优先级低于T->data)  
    {  
      printf("(");  
      if(!Print\_Expression(T->rchild)) return ERROR;  
      printf(")");  
    }  
    else if(!Print\_Expression(T->rchild)) return ERROR;  
  }  
  else return ERROR; //非法字符  
  return OK;  
}//Print\_Expression   
6.52   
typedef struct{  
                    BTNode node;  
                    int layer;  
                  } BTNRecord; //包含结点所在层次的记录类型   
int FanMao(Bitree T)//求一棵二叉树的"繁茂度"  
{  
  int countd; //count数组存放每一层的结点数  
  InitQueue(Q); //Q的元素为BTNRecord类型  
  EnQueue(Q,{T,0});  
  while(!QueueEmpty(Q))  
  {  
    DeQueue(Q,r);  
    count[r.layer]++;  
    if(r.node->lchild) EnQueue(Q,{r.node->lchild,r.layer+1});  
    if(r.node->rchild) EnQueue(Q,{r.node->rchild,r.layer+1});  
  } //利用层序遍历来统计各层的结点数  
  h=r.layer; //最后一个队列元素所在层就是树的高度  
  for(maxn=count[0],i=1;count[i];i++)  
    if(count[i]>maxn) maxn=count[i]; //求层最大结点数  
  return h\*maxn;  
}//FanMao  
分析:如果不允许使用辅助数组,就必须在遍历的同时求出层最大结点数,形式上会复杂一些,你能写出来吗?   
6.53   
int maxh;   
Status Printpath\_MaxdepthS1(Bitree T)//求深度等于树高度减一的最靠左的结点  
{  
  Bitree pathd;  
  maxh=Get\_Depth(T); //Get\_Depth函数见6.44  
  if(maxh<2) return ERROR; //无符合条件结点  
  Find\_h(T,1);  
  return OK;  
}//Printpath\_MaxdepthS1   
void Find\_h(Bitree T,int h)//寻找深度为maxh-1的结点  
{  
  path[h]=T;  
  if(h==maxh-1)  
  {  
    for(i=1;path[i];i++) printf("%c",path[i]->data);  
    exit; //打印输出路径  
  }  
  else  
  {  
    if(T->lchild) Find\_h(T->lchild,h+1);  
    if(T->rchild) Find\_h(T->rchild,h+1);  
  }  
  path[h]=NULL; //回溯  
}//Find\_h   
6.54   
Status CreateBitree\_SqList(Bitree &T,SqList sa)//根据顺序存储结构建立二叉链表  
{  
  Bitree ptr[sa.last+1]; //该数组储存与sa中各结点对应的树指针  
  if(!sa.last)  
  {  
    T=NULL; //空树  
    return;  
  }  
  ptr[1]=(BTNode\*)malloc(sizeof(BTNode));  
  ptr[1]->data=sa.elem[1]; //建立树根  
  T=ptr[1];  
  for(i=2;i<=sa.last;i++)  
  {  
    if(!sa.elem[i]) return ERROR; //顺序错误  
    ptr[i]=(BTNode\*)malloc(sizeof(BTNode));  
    ptr[i]->data=sa.elem[i];  
    j=i/2; //找到结点i的双亲j  
    if(i-j\*2) ptr[j]->rchild=ptr[i]; //i是j的右孩子  
    else ptr[j]->lchild=ptr[i]; //i是j的左孩子  
  }  
  return OK;  
}//CreateBitree\_SqList   
6.55   
int DescNum(Bitree T)//求树结点T的子孙总数填入DescNum域中,并返回该数  
{  
  if(!T) return -1;  
  else d=(DescNum(T->lchild)+DescNum(T->rchild)+2); //计算公式  
  T->DescNum=d;  
  return d;  
}//DescNum  
分析:该算法时间复杂度为O(n),n为树结点总数.注意:为了能用一个统一的公式计算子孙数目,所以当T为空指针时,要返回-1而不是0.   
6.56   
BTNode \*PreOrder\_Next(BTNode \*p)//在先序后继线索二叉树中查找结点p的先序后继,并返回指针  
{  
  if(p->lchild) return p->lchild;  
  else return p->rchild;  
}//PreOrder\_Next  
分析:总觉得不会这么简单.是不是哪儿理解错了?   
6.57   
Bitree PostOrder\_Next(Bitree p)//在后序后继线索二叉树中查找结点p的后序后继,并返回指针  
{  
  if(p->rtag) return p->rchild; //p有后继线索  
  else if(!p->parent) return NULL; //p是根结点  
  else if(p==p->parent->rchild) return p->parent; //p是右孩子  
  else if(p==p->parent->lchild&&p->parent->rtag)  
    return p->parent; //p是左孩子且双亲没有右孩子  
  else //p是左孩子且双亲有右孩子  
  {  
    q=p->parent->rchild;  
    while(q->lchild||!q->rtag)  
    {  
      if(q->lchild) q=q->lchild;  
      else q=q->rchild;  
    } //从p的双亲的右孩子向下走到底  
  return q;  
  }//else  
}//PostOrder\_Next   
6.58   
Status Insert\_BiThrTree(BiThrTree &T,BiThrTree &p,BiThrTree &x)//在中序线索二叉树T的结点p下插入子树x  
{  
  if(p->ltag) //x作为p的左子树  
  {  
    s=p->lchild; //s为p的前驱  
    p->ltag=Link;   
    p->lchild=x;   
    q=x;   
    while(q->lchild&&!q->ltag) q=q->lchild;   
    q->lchild=s; //找到子树中的最左结点,并修改其前驱指向s  
    x->rtag=Thread;   
    x->rchild=p; //x的后继指向p  
  }  
  else if(p->rtag) //x作为p的右子树  
  {  
    s=p->rchild; //s为p的后继  
    p->rtag=Link;   
    p->rchild=x;   
    q=x;   
    while(q->lchild&&!q->ltag) q=q->lchild;   
    q->lchild=p; //找到子树中的最左结点,并修改其前驱指向p  
    x->rtag=Thread;   
    x->rchild=s; //x的后继指向p的后继  
  }  
  else//x作为p的左子树,p的左子树作为x的右子树  
  {  
    s=p->lchild;t=s;   
    while(t->lchild&&!t->ltag) t=t->lchild;   
    u=t->lchild; //找到p的左子树的最左节点t和前驱u  
    p->lchild=x;   
    x->rtag=Link;   
    x->rchild=s; //x作为p的左子树,p的左子树s作为x的右子树  
    t->lchild=x;   
    q=x;   
    while(q->lchild&&!q->ltag) q=q->lchild;   
    q->lchild=u; //找到子树中的最左结点,并修改其前驱指向u  
  }  
  return OK;   
}//Insert\_BiThrTree  
6.59   
void Print\_CSTree(CSTree T)//输出孩子兄弟链表表示的树T的各边  
{  
  for(child=T->firstchild;child;child=child->nextsib)  
  {  
    printf("(%c,%c),",T->data,child->data);  
    Print\_CSTree(child);  
  }  
}//Print\_CSTree   
6.60   
int LeafCount\_CSTree(CSTree T)//求孩子兄弟链表表示的树T的叶子数目  
{  
  if(!T->firstchild) return 1; //叶子结点  
  else  
  {  
    count=0;  
    for(child=T->firstchild;child;child=child->nextsib)  
      count+=LeafCount\_CSTree(child);  
    return count; //各子树的叶子数之和  
  }  
}//LeafCount\_CSTree   
6.61   
int GetDegree\_CSTree(CSTree T)//求孩子兄弟链表表示的树T的度  
{  
  if(!T->firstchild) return 0; //空树  
  else  
  {  
    degree=0;  
    for(p=T->firstchild;p;p=p->nextsib) degree++;//本结点的度  
    for(p=T->firstchild;p;p=p->nextsib)  
    {  
      d=GetDegree\_CSTree(p);  
      if(d>degree) degree=d; //孩子结点的度的最大值  
    }  
    return degree;  
  }//else  
}//GetDegree\_CSTree   
6.62   
int GetDepth\_CSTree(CSTree T)//求孩子兄弟链表表示的树T的深度  
{  
  if(!T) return 0; //空树  
  else  
  {  
    for(maxd=0,p=T->firstchild;p;p=p->nextsib)  
      if((d=GetDepth\_CSTree(p))>maxd) maxd=d; //子树的最大深度  
    return maxd+1;  
  }  
}//GetDepth\_CSTree   
6.63   
int GetDepth\_CTree(CTree A)//求孩子链表表示的树A的深度  
{  
  return SubDepth(A.r);  
}//GetDepth\_CTree   
int SubDepth(int T)//求子树T的深度  
{  
  if(!A.nodes[T].firstchild) return 1;  
  for(sd=1,p=A.nodes[T].firstchild;p;p=p->next)  
    if((d=SubDepth(p->child))>sd) sd=d;  
  return sd+1;  
}//SubDepth   
6.64   
int GetDepth\_PTree(PTree T)//求双亲表表示的树T的深度  
{  
  maxdep=0;  
  for(i=0;i<T.n;i++)  
  {  
    dep=0;  
    for(j=i;j>=0;j=T.nodes[j].parent) dep++; //求每一个结点的深度  
    if(dep>maxdep) maxdep=dep;  
  }  
  return maxdep;  
}//GetDepth\_PTree   
6.65   
char Pred,Ind; //假设前序序列和中序序列已经分别储存在数组Pre和In中   
Bitree Build\_Sub(int Pre\_Start,int Pre\_End,int In\_Start,int In\_End)//由子树的前序和中序序列建立其二叉链表  
{  
  sroot=(BTNode\*)malloc(sizeof(BTNode)); //建根  
  sroot->data=Pre[Pre\_Start];  
  for(i=In\_Start;In[i]!=sroot->data;i++); //在中序序列中查找子树根  
  leftlen=i-In\_Start;  
  rightlen=In\_End-i; //计算左右子树的大小  
  if(leftlen)  
  {  
    lroot=Build\_Sub(Pre\_Start+1,Pre\_Start+leftlen,In\_Start,In\_Start+leftlen-1);  
    sroot->lchild=lroot;  
  } //建左子树,注意参数表的计算  
  if(rightlen)  
  {  
    rroot=Build\_Sub(Pre\_End-rightlen+1,Pre\_End,In\_End-rightlen+1,In\_End);  
    sroot->rchild=rroot;  
  } //建右子树,注意参数表的计算  
  return sroot; //返回子树根  
}//Build\_Sub   
main()  
{  
  ...  
  Build\_Sub(1,n,1,n); //初始调用参数,n为树结点总数  
  ...  
}  
分析:本算法利用了这样一个性质,即一棵子树在前序和中序序列中所占的位置总是连续的.因此,就可以用起始下标和终止下标来确定一棵子树.Pre\_Start,Pre\_End,In\_Start和In\_End分别指示子树在前序子序列里的起始下标,终止下标,和在中序子序列里的起始和终止下标.   
6.66   
typedef struct{  
                    CSNode \*ptr;  
                    CSNode \*lastchild;  
                  } NodeMsg; //结点的指针和其最后一个孩子的指针   
Status Build\_CSTree\_PTree(PTree T)//由树T的双亲表构造其孩子兄弟链表  
{  
  NodeMsg Tree[MAXSIZE];  
  for(i=0;i<T.n;i++)  
  {  
    Tree[i].ptr=(CSNode\*)malloc(sizeof(CSNode));  
    Tree[i].ptr->data=T.node[i].data; //建结点  
    if(T.nodes[i].parent>=0) //不是树根  
    {  
      j=T.nodes[i].parent; //本算法要求双亲表必须是按层序存储  
      if(!(Tree[j].lastchild)) //双亲当前还没有孩子  
        Tree[j].ptr->firstchild=Tree[i].ptr; //成为双亲的第一个孩子  
      else //双亲已经有了孩子  
        Tree[j].lastchild->nextsib=Tree[i].ptr; //成为双亲最后一个孩子的下一个兄弟  
      Tree[j].lastchild=Tree[i].ptr; //成为双亲的最后一个孩子  
    }//if  
  }//for  
}//Build\_CSTree\_PTree   
6.67   
typedef struct{  
                char data;  
                CSNode \*ptr;  
                CSNode \*lastchild;  
              } NodeInfo; //结点数据,结点指针和最后一个孩子的指针   
Status CreateCSTree\_Duplet(CSTree &T)//输入二元组建立树的孩子兄弟链表  
{  
  NodeInfo Treed;  
  n=1;k=0;  
  if(getchar()!='^') return ERROR; //未按格式输入  
  if((c=getchar())=='^') T=NULL; //空树  
  Tree[0].ptr=(CSNode\*)malloc(sizeof(CSNode));  
  Tree[0].data=c;  
  Tree[0].ptr->data=c;  
  while((p=getchar())!='^'&&(c=getchar())!='^')  
  {  
    Tree[n].ptr=(CSNode\*)malloc(sizeof(CSNode));  
    Tree[n].data=c;  
    Tree[n].ptr->data=c;  
    for(k=0;Tree[k].data!=p;k++); //查找当前边的双亲结点  
    if(Tree[k].data!=p) return ERROR; //未找到:未按层序输入  
    r=Tree[k].ptr;  
    if(!r->firstchild)  
      r->firstchild=Tree[n].ptr;  
    else Tree[k].lastchild->nextsib=Tree[n].ptr;  
    Tree[k].lastchild=Tree[n].ptr; //这一段含义同上一题  
    n++;  
  }//while  
  return OK;  
}//CreateCSTree\_Duplet   
6.68   
Status CreateCSTree\_Degree(char node[ ],int degree[ ])//由结点的层序序列和各结点的度构造树的孩子兄弟链表  
{  
  CSNode \* ptr[MAXSIZE]; //树结点指针的辅助存储  
  ptr[0]=(CSNode\*)malloc(sizeof(CSNode));  
  i=0;k=1; //i为当前结点序号,k为当前孩子的序号  
  while(node[i])  
  {  
    ptr[i]->data=node[i];  
    d=degree[i];  
    if(d)  
    {  
      ptr[k]=(CSNode\*)malloc(sizeof(CSNode)); //k为当前孩子的序号  
      ptr[i]->firstchild=ptr[k]; //建立i与第一个孩子k之间的联系  
      for(j=2;j<=d;j++)  
      {  
        ptr[k]=(CSNode\*)malloc(sizeof(CSNode));  
        ptr[k-1]->nextsib=ptr[k]; //当结点的度大于1时,为其孩子建立兄弟链表  
        k++;  
      }//for  
     ptr[k-1]->nextsib=NULL;  
    }//if  
  i++;  
  }//while  
}//CreateCSTree\_Degree   
6.69   
void Print\_BiTree(BiTree T,int i)//按树状打印输出二叉树的元素,i表示结点所在层次,初次调用时i=0  
{  
  if(T->rchild) Print\_BiTree(T->rchild,i+1);  
  for(j=1;j<=i;j++) printf(" "); //打印i个空格以表示出层次  
  printf("%c\n",T->data); //打印T元素,换行  
  if(T->lchild) Print\_BiTree(T->rchild,i+1);  
}//Print\_BiTree  
分析:该递归算法实际上是带层次信息的中序遍历,只不过按照题目要求,顺序为先右后左.   
6.70   
Status CreateBiTree\_GList(BiTree &T)//由广义表形式的输入建立二叉链表  
{  
  c=getchar();  
  if(c=='#') T=NULL; //空子树  
  else  
  {  
    T=(CSNode\*)malloc(sizeof(CSNode));  
    T->data=c;  
    if(getchar()!='(') return ERROR;  
    if(!CreateBiTree\_GList(pl)) return ERROR;  
    T->lchild=pl;  
    if(getchar()!=',') return ERROR;  
    if(!CreateBiTree\_GList(pr)) return ERROR;  
    T->rchild=pr;  
    if(getchar()!=')') return ERROR; //这些语句是为了保证输入符合A(B,C)的格式  
  }  
  return OK;  
}//CreateBiTree\_GList   
6.71   
void Print\_CSTree(CSTree T,int i)//按凹入表形式打印输出树的元素,i表示结点所在层次,初次调用时i=0  
{  
  for(j=1;j<=i;j++) printf(" "); //留出i个空格以表现出层次  
  printf("%c\n",T->data); //打印元素,换行  
  for(p=T->firstchild;p;p=p->nextsib)  
    Print\_CSTree(p,i+1); //打印子树  
}//Print\_CSTree   
6.72   
void Print\_CTree(int e,int i)//按凹入表形式打印输出树的元素,i表示结点所在层次  
{  
  for(j=1;j<=i;j++) printf(" "); //留出i个空格以表现出层次  
  printf("%c\n",T.nodes[e].data); //打印元素,换行  
  for(p=T.nodes[e].firstchild;p;p=p->next)  
    Print\_CSTree(p->child,i+1); //打印子树  
}//Print\_CSTree   
main()  
{  
  ...  
  Print\_CTree(T.r,0); //初次调用时i=0  
  ...  
}//main   
6.73   
char c; //全局变量,指示当前字符   
Status CreateCSTree\_GList(CSTree &T)//由广义表形式的输入建立孩子兄弟链表  
{  
  c=getchar();  
  T=(CSNode\*)malloc(sizeof(CSNode));  
  T->data=c;  
  if((c=getchar())=='(') //非叶结点  
  {  
    if(!CreateCSTree\_GList(fc)) return ERROR; //建第一个孩子  
    T->firstchild=fc;  
    for(p=fc;c==',';p->nextsib=nc,p=nc) //建兄弟链  
      if(!CreateCSTree\_GList(nc)) return ERROR;  
    p->nextsib=NULL;  
    if((c=getchar())!=')') return ERROR; //括号不配对  
  }  
  else T->firtchild=NULL; //叶子结点  
  return OK;  
}//CreateBiTree\_GList  
分析:书后给出了两个间接递归的算法,事实上合成一个算法在形式上可能更好一些.本算法另一个改进之处在于加入了广义表格式是否合法的判断.   
6.74   
void PrintGlist\_CSTree(CSTree T)//按广义表形式输出孩子兄弟链表表示的树  
{  
  printf("%c",T->data);  
  if(T->firstchild) //非叶结点  
  {  
    printf("(");  
    for(p=T->firstchild;p;p=p->nextsib)  
    {  
      PrintGlist\_CSTree(p);  
      if(p->nextsib) printf(","); //最后一个孩子后面不需要加逗号  
    }  
    printf(")");  
  }//if  
}//PrintGlist\_CSTree   
6.75   
char c;  
int pos=0; //pos是全局变量,指示已经分配到了哪个结点   
Status CreateCTree\_GList(CTree &T,int &i)//由广义表形式的输入建立孩子链表  
{  
  c=getchar();  
  T.nodes[pos].data=c;  
  i=pos++; //i是局部变量,指示当前正在处理的子树根  
  if((c=getchar())=='(') //非叶结点  
  {  
    CreateCTree\_GList();  
    p=(CTBox\*)malloc(sizeof(CTBox));  
    T.nodes[i].firstchild=p;  
    p->child=pos; //建立孩子链的头  
    for(;c==',';p=p->next) //建立孩子链  
    {  
      CreateCTree\_GList(T,j); //用j返回分配得到的子树根位置  
      p->child=j;  
      p->next=(CTBox\*)malloc(sizeof(CTBox));  
    }  
    p->next=NULL;  
    if((c=getchar())!=')') return ERROR; //括号不配对  
  }//if  
  else T.nodes[i].firtchild=NULL; //叶子结点  
  return OK;  
}//CreateBiTree\_GList  
分析:该算法中,pos变量起着"分配"结点在表中的位置的作用,是按先序序列从上向下分配,因此树根T.r一定等于0,而最终的pos值就是结点数T.n.   
6.76   
void PrintGList\_CTree(CTree T,int i)//按广义表形式输出孩子链表表示的树  
{  
  printf("%c",T.nodes[i].data);  
  if(T.nodes[i].firstchild) //非叶结点  
  {  
    printf("(");  
    for(p=T->firstchild;p;p=p->nextsib)  
    {  
      PrintGlist\_CSTree(T,p->child);  
      if(p->nextsib) printf(","); //最后一个孩子后面不需要加逗号  
    }  
    printf(")");  
  }//if  
}//PrintGlist\_CTree

第七章 图  
  
7.14   
Status Build\_AdjList(ALGraph &G)//输入有向图的顶点数,边数,顶点信息和边的信息建立邻接表  
{  
  InitALGraph(G);  
  scanf("%d",&v);  
  if(v<0) return ERROR; //顶点数不能为负  
  G.vexnum=v;  
  scanf("%d",&a);  
  if(a<0) return ERROR; //边数不能为负  
  G.arcnum=a;  
  for(m=0;m<v;m++)  
    G.vertices[m].data=getchar(); //输入各顶点的符号  
  for(m=1;m<=a;m++)  
  {  
    t=getchar();h=getchar(); //t为弧尾,h为弧头  
    if((i=LocateVex(G,t))<0) return ERROR;  
    if((j=LocateVex(G,h))<0) return ERROR; //顶点未找到  
    p=(ArcNode\*)malloc(sizeof(ArcNode));  
    if(!G.vertices.[i].firstarc) G.vertices[i].firstarc=p;  
    else  
    {  
      for(q=G.vertices[i].firstarc;q->nextarc;q=q->nextarc);  
      q->nextarc=p;  
    }  
    p->adjvex=j;p->nextarc=NULL;  
  }//while  
  return OK;  
}//Build\_AdjList   
7.15   
//本题中的图G均为有向无权图,其余情况容易由此写出  
Status Insert\_Vex(MGraph &G, char v)//在邻接矩阵表示的图G上插入顶点v  
{  
  if(G.vexnum+1)>MAX\_VERTEX\_NUM return INFEASIBLE;  
  G.vexs[++G.vexnum]=v;  
  return OK;  
}//Insert\_Vex   
Status Insert\_Arc(MGraph &G,char v,char w)//在邻接矩阵表示的图G上插入边(v,w)  
{  
  if((i=LocateVex(G,v))<0) return ERROR;  
  if((j=LocateVex(G,w))<0) return ERROR;  
  if(i==j) return ERROR;  
  if(!G.arcs[i][j].adj)  
  {  
    G.arcs[i][j].adj=1;  
    G.arcnum++;  
  }  
  return OK;  
}//Insert\_Arc   
Status Delete\_Vex(MGraph &G,char v)//在邻接矩阵表示的图G上删除顶点v  
{  
  n=G.vexnum;  
  if((m=LocateVex(G,v))<0) return ERROR;  
  G.vexs[m]<->G.vexs[n]; //将待删除顶点交换到最后一个顶点  
  for(i=0;i<n;i++)  
  {  
    G.arcs[i][m]=G.arcs[i][n];  
    G.arcs[m][i]=G.arcs[n][i]; //将边的关系随之交换  
  }  
  G.arcs[m][m].adj=0;  
  G.vexnum--;  
  return OK;  
}//Delete\_Vex  
分析:如果不把待删除顶点交换到最后一个顶点的话,算法将会比较复杂,而伴随着大量元素的移动,时间复杂度也会大大增加.   
Status Delete\_Arc(MGraph &G,char v,char w)//在邻接矩阵表示的图G上删除边(v,w)  
{  
  if((i=LocateVex(G,v))<0) return ERROR;  
  if((j=LocateVex(G,w))<0) return ERROR;  
  if(G.arcs[i][j].adj)  
  {  
    G.arcs[i][j].adj=0;  
    G.arcnum--;  
  }  
  return OK;  
}//Delete\_Arc   
7.16   
//为节省篇幅,本题只给出Insert\_Arc算法.其余算法请自行写出.   
Status Insert\_Arc(ALGraph &G,char v,char w)//在邻接表表示的图G上插入边(v,w)  
{  
  if((i=LocateVex(G,v))<0) return ERROR;  
  if((j=LocateVex(G,w))<0) return ERROR;  
  p=(ArcNode\*)malloc(sizeof(ArcNode));  
  p->adjvex=j;p->nextarc=NULL;  
  if(!G.vertices[i].firstarc) G.vertices[i].firstarc=p;  
  else  
  {  
    for(q=G.vertices[i].firstarc;q->q->nextarc;q=q->nextarc)  
      if(q->adjvex==j) return ERROR; //边已经存在  
    q->nextarc=p;  
  }  
  G.arcnum++;  
  return OK;  
}//Insert\_Arc   
7.17   
//为节省篇幅,本题只给出较为复杂的Delete\_Vex算法.其余算法请自行写出.   
Status Delete\_Vex(OLGraph &G,char v)//在十字链表表示的图G上删除顶点v  
{  
  if((m=LocateVex(G,v))<0) return ERROR;  
  n=G.vexnum;  
  for(i=0;i<n;i++) //删除所有以v为头的边  
  {  
    if(G.xlist[i].firstin->tailvex==m) //如果待删除的边是头链上的第一个结点  
    {  
      q=G.xlist[i].firstin;  
      G.xlist[i].firstin=q->hlink;  
      free(q);G.arcnum--;  
    }  
    else //否则  
    {  
      for(p=G.xlist[i].firstin;p&&p->hlink->tailvex!=m;p=p->hlink);  
      if(p)  
      {  
        q=p->hlink;  
        p->hlink=q->hlink;  
        free(q);G.arcnum--;  
      }  
    }//else  
  }//for  
  for(i=0;i<n;i++) //删除所有以v为尾的边  
  {  
    if(G.xlist[i].firstout->headvex==m) //如果待删除的边是尾链上的第一个结点  
    {  
      q=G.xlist[i].firstout;  
      G.xlist[i].firstout=q->tlink;  
      free(q);G.arcnum--;  
    }  
    else //否则  
    {  
      for(p=G.xlist[i].firstout;p&&p->tlink->headvex!=m;p=p->tlink);  
      if(p)  
      {  
        q=p->tlink;  
        p->tlink=q->tlink;  
        free(q);G.arcnum--;  
      }  
    }//else  
  }//for  
  for(i=m;i<n;i++) //顺次用结点m之后的顶点取代前一个顶点  
  {  
    G.xlist[i]=G.xlist[i+1]; //修改表头向量  
    for(p=G.xlist[i].firstin;p;p=p->hlink)  
      p->headvex--;  
    for(p=G.xlist[i].firstout;p;p=p->tlink)  
      p->tailvex--; //修改各链中的顶点序号  
  }  
  G.vexnum--;  
  return OK;  
}//Delete\_Vex   
7.18   
//为节省篇幅,本题只给出Delete\_Arc算法.其余算法请自行写出.   
Status Delete\_Arc(AMLGraph &G,char v,char w)////在邻接多重表表示的图G上删除边(v,w)  
{  
  if((i=LocateVex(G,v))<0) return ERROR;  
  if((j=LocateVex(G,w))<0) return ERROR;  
  if(G.adjmulist[i].firstedge->jvex==j)  
    G.adjmulist[i].firstedge=G.adjmulist[i].firstedge->ilink;  
  else  
  {  
    for(p=G.adjmulist[i].firstedge;p&&p->ilink->jvex!=j;p=p->ilink);  
    if (!p) return ERROR; //未找到  
    p->ilink=p->ilink->ilink;  
  } //在i链表中删除该边  
  if(G.adjmulist[j].firstedge->ivex==i)  
    G.adjmulist[j].firstedge=G.adjmulist[j].firstedge->jlink;  
  else  
  {  
    for(p=G.adjmulist[j].firstedge;p&&p->jlink->ivex!=i;p=p->jlink);  
    if (!p) return ERROR; //未找到  
    q=p->jlink;  
    p->jlink=q->jlink;  
    free(q);  
  } //在i链表中删除该边  
  G.arcnum--;  
  return OK;  
}//Delete\_Arc   
7.19   
Status Build\_AdjMulist(AMLGraph &G)//输入有向图的顶点数,边数,顶点信息和边的信息建立邻接多重表  
{  
  InitAMLGraph(G);  
  scanf("%d",&v);  
  if(v<0) return ERROR; //顶点数不能为负  
  G.vexnum=v;  
  scanf(%d",&a);  
  if(a<0) return ERROR; //边数不能为负  
  G.arcnum=a;  
  for(m=0;m<v;m++)  
    G.adjmulist[m].data=getchar(); //输入各顶点的符号  
  for(m=1;m<=a;m++)  
  {  
    t=getchar();h=getchar(); //t为弧尾,h为弧头  
    if((i=LocateVex(G,t))<0) return ERROR;  
    if((j=LocateVex(G,h))<0) return ERROR; //顶点未找到  
    p=(EBox\*)malloc(sizeof(EBox));  
    p->ivex=i;p->jvex=j;  
    p->ilink=NULL;p->jlink=NULL; //边结点赋初值  
    if(!G.adjmulist[i].firstedge) G.adjmulist[i].firstedge=p;  
    else  
    {  
      q=G.adjmulist[i].firstedge;  
      while(q)  
      {  
        r=q;  
        if(q->ivex==i) q=q->ilink;  
        else q=q->jlink;  
      }  
      if(r->ivex==i) r->ilink=p;//注意i值既可能出现在边结点的ivex域中,  
      else r->jlink=p; //又可能出现在边结点的jvex域中  
    }//else //插入i链表尾部  
    if(!G.adjmulist[j].firstedge) G.adjmulist[j].firstedge=p;  
    else  
    {  
      q=G.adjmulist[i].firstedge;  
      while(q)  
      {  
        r=q;  
        if(q->jvex==j) q=q->jlink;  
        else q=q->ilnk;  
      }  
      if(r->jvex==j) r->jlink=p;  
      else r->ilink=p;  
    }//else //插入j链表尾部  
  }//for  
  return OK;  
}//Build\_AdjList   
7.20   
int Pass\_MGraph(MGraph G)//判断一个邻接矩阵存储的有向图是不是可传递的,是则返回1,否则返回0  
{  
  for(x=0;x<G.vexnum;x++)  
    for(y=0;y<G.vexnum;y++)  
      if(G.arcs[x][y])  
      {  
        for(z=0;z<G.vexnum;z++)  
          if(z!=x&&G.arcs[y][z]&&!G.arcs[x][z]) return 0;//图不可传递的条件  
      }//if  
  return 1;  
}//Pass\_MGraph  
分析:本算法的时间复杂度大概是O(n^2\*d).   
7.21   
int Pass\_ALGraph(ALGraph G)//判断一个邻接表存储的有向图是不是可传递的,是则返回1,否则返回0  
{  
  for(x=0;x<G.vexnum;x++)  
    for(p=G.vertices[x].firstarc;p;p=p->nextarc)  
    {  
      y=p->adjvex;  
      for(q=G.vertices[y].firstarc;q;q=q->nextarc)  
      {  
        z=q->adjvex;  
        if(z!=x&&!is\_adj(G,x,z)) return 0;  
      }//for  
    }//for  
}//Pass\_ALGraph   
int is\_adj(ALGraph G,int m,int n)//判断有向图G中是否存在边(m,n),是则返回1,否则返回0  
{  
  for(p=G.vertices[m].firstarc;p;p=p->nextarc)  
    if(p->adjvex==n) return 1;  
  return 0;  
}//is\_adj   
7.22   
int visited[MAXSIZE]; //指示顶点是否在当前路径上   
int exist\_path\_DFS(ALGraph G,int i,int j)//深度优先判断有向图G中顶点i到顶点j是否有路径,是则返回1,否则返回0  
{  
  if(i==j) return 1; //i就是j  
  else  
  {  
    visited[i]=1;  
    for(p=G.vertices[i].firstarc;p;p=p->nextarc)  
    {  
      k=p->adjvex;  
      if(!visited[k]&&exist\_path(k,j)) return 1;//i下游的顶点到j有路径  
    }//for  
  }//else  
}//exist\_path\_DFS   
7.23   
int exist\_path\_BFS(ALGraph G,int i,int j)//广度优先判断有向图G中顶点i到顶点j是否有路径,是则返回1,否则返回0  
{  
  int visited[MAXSIZE];  
  InitQueue(Q);  
  EnQueue(Q,i);  
  while(!QueueEmpty(Q))  
  {  
    DeQueue(Q,u);  
    visited[u]=1;  
    for(p=G.vertices[i].firstarc;p;p=p->nextarc)  
    {  
      k=p->adjvex;  
      if(k==j) return 1;  
      if(!visited[k]) EnQueue(Q,k);  
    }//for  
  }//while  
  return 0;  
}//exist\_path\_BFS   
7.24   
void STraverse\_Nonrecursive(Graph G)//非递归遍历强连通图G  
{  
  int visited[MAXSIZE];  
  InitStack(S);  
  Push(S,GetVex(S,1)); //将第一个顶点入栈  
  visit(1);  
  visited =1;  
  while(!StackEmpty(S))  
  {  
    while(Gettop(S,i)&&i)  
    {  
      j=FirstAdjVex(G,i);  
      if(j&&!visited[j])  
      {  
        visit(j);  
        visited[j]=1;  
        Push(S,j); //向左走到尽头  
      }  
    }//while  
    if(!StackEmpty(S))  
    {  
      Pop(S,j);  
      Gettop(S,i);  
      k=NextAdjVex(G,i,j); //向右走一步  
      if(k&&!visited[k])  
      {  
        visit(k);  
        visited[k]=1;  
        Push(S,k);  
      }  
    }//if  
  }//while  
}//Straverse\_Nonrecursive  
分析:本算法的基本思想与二叉树的先序遍历非递归算法相同,请参考6.37.由于是强连通图,所以从第一个结点出发一定能够访问到所有结点.   
7.25   
见书后解答.   
7.26   
Status TopoNo(ALGraph G)//按照题目要求顺序重排有向图中的顶点  
{  
  int new[MAXSIZE],indegree[MAXSIZE]; //储存结点的新序号  
  n=G.vexnum;  
  FindInDegree(G,indegree);  
  InitStack(S);  
  for(i=1;i<G.vexnum;i++)  
    if(!indegree[i]) Push(S,i); //零入度结点入栈  
  count=0;  
  while(!StackEmpty(S))  
  {  
    Pop(S,i);  
    new[i]=n--; //记录结点的拓扑逆序序号  
    count++;  
    for(p=G.vertices[i].firstarc;p;p=p->nextarc)  
    {  
      k=p->adjvex;  
      if(!(--indegree[k])) Push(S,k);  
    }//for  
  }//while  
  if(count<G.vexnum) return ERROR; //图中存在环  
  for(i=1;i<=n;i++) printf("Old No:%d New No:%d\n",i,new[i])  
  return OK;  
}//TopoNo  
分析:只要按拓扑逆序对顶点编号,就可以使邻接矩阵成为下三角矩阵.   
7.27   
int visited[MAXSIZE];   
int exist\_path\_len(ALGraph G,int i,int j,int k)//判断邻接表方式存储的有向图G的顶点i到j是否存在长度为k的简单路径  
{  
  if(i==j&&k==0) return 1; //找到了一条路径,且长度符合要求  
  else if(k>0)  
  {  
    visited[i]=1;  
    for(p=G.vertices[i].firstarc;p;p=p->nextarc)  
    {  
      l=p->adjvex;  
      if(!visited[l])  
        if(exist\_path\_len(G,l,j,k-1)) return 1; //剩余路径长度减一  
    }//for  
    visited[i]=0; //本题允许曾经被访问过的结点出现在另一条路径中  
  }//else  
  return 0; //没找到  
}//exist\_path\_len   
7.28   
int path[MAXSIZE],visited[MAXSIZE]; //暂存遍历过程中的路径   
int Find\_All\_Path(ALGraph G,int u,int v,int k)//求有向图G中顶点u到v之间的所有简单路径,k表示当前路径长度  
{  
  path[k]=u; //加入当前路径中  
  visited[u]=1;  
  if(u==v) //找到了一条简单路径  
  {  
    printf("Found one path!\n");  
    for(i=0;path[i];i++) printf("%d",path[i]); //打印输出  
  }  
  else  
    for(p=G.vertices[u].firstarc;p;p=p->nextarc)  
    {  
      l=p->adjvex;  
      if(!visited[l]) Find\_All\_Path(G,l,v,k+1); //继续寻找  
    }  
  visited[u]=0;  
  path[k]=0; //回溯  
}//Find\_All\_Path   
main()  
{  
  ...  
  Find\_All\_Path(G,u,v,0); //在主函数中初次调用,k值应为0  
  ...  
}//main   
7.29   
int GetPathNum\_Len(ALGraph G,int i,int j,int len)//求邻接表方式存储的有向图G的顶点i到j之间长度为len的简单路径条数  
{  
  if(i==j&&len==0) return 1; //找到了一条路径,且长度符合要求  
  else if(len>0)  
  {  
    sum=0; //sum表示通过本结点的路径数  
    visited[i]=1;  
    for(p=G.vertices[i].firstarc;p;p=p->nextarc)  
    {  
      l=p->adjvex;  
      if(!visited[l])  
        sum+=GetPathNum\_Len(G,l,j,len-1)//剩余路径长度减一  
    }//for  
    visited[i]=0; //本题允许曾经被访问过的结点出现在另一条路径中  
  }//else  
  return sum;  
}//GetPathNum\_Len   
7.30   
int visited[MAXSIZE];  
int path[MAXSIZE]; //暂存当前路径  
int cycles[MAXSIZE][MAXSIZE]; //储存发现的回路所包含的结点  
int thiscycle[MAXSIZE]; //储存当前发现的一个回路  
int cycount=0; //已发现的回路个数   
void GetAllCycle(ALGraph G)//求有向图中所有的简单回路  
{  
  for(v=0;v<G.vexnum;v++) visited[v]=0;  
  for(v=0;v<G.vexnum;v++)  
    if(!visited[v]) DFS(G,v,0); //深度优先遍历  
}//DFSTraverse   
void DFS(ALGraph G,int v,int k)//k表示当前结点在路径上的序号  
{  
  visited[v]=1;  
  path[k]=v; //记录当前路径  
  for(p=G.vertices[v].firstarc;p;p=p->nextarc)  
  {  
    w=p->adjvex;  
    if(!visited[w]) DFS(G,w,k+1);  
    else //发现了一条回路  
    {  
      for(i=0;path[i]!=w;i++); //找到回路的起点  
      for(j=0;path[i+j];i++) thiscycle[j]=path[i+j];//把回路复制下来  
      if(!exist\_cycle()) cycles[cycount++]=thiscycle;//如果该回路尚未被记录过,就添加到记录中  
      for(i=0;i<G.vexnum;i++) thiscycle[i]=0; //清空目前回路数组  
    }//else  
  }//for  
  path[k]=0;  
  visited[k]=0; //注意只有当前路径上的结点visited为真.因此一旦遍历中发现当前结点visited为真,即表示发现了一条回路  
}//DFS   
int exist\_cycle()//判断thiscycle数组中记录的回路在cycles的记录中是否已经存在  
{  
  int temp[MAXSIZE];  
  for(i=0;i<cycount;i++) //判断已有的回路与thiscycle是否相同  
  { //也就是,所有结点和它们的顺序都相同  
    j=0;c=thiscycle&#0;; //例如,142857和857142是相同的回路  
    for(k=0;cycles[i][k]!=c&&cycles[i][k]!=0;k++);//在cycles的一个行向量中寻找等于thiscycle第一个结点的元素  
    if(cycles[i][k]) //有与之相同的一个元素  
    {  
      for(m=0;cycles[i][k+m];m++)  
        temp[m]=cycles[i][k+m];  
      for(n=0;n<k;n++,m++)  
        temp[m]=cycles[i][n]; //调整cycles中的当前记录的循环相位并放入temp数组中  
      if(!StrCompare(temp,thiscycle)) //与thiscycle比较  
        return 1; //完全相等  
      for(m=0;m<G.vexnum;m++) temp[m]=0; //清空这个数组  
    }  
  }//for  
  return 0; //所有现存回路都不与thiscycle完全相等  
}//exist\_cycle  
分析:这个算法的思想是,在遍历中暂存当前路径,当遇到一个结点已经在路径之中时就表明存在一条回路;扫描路径向量path可以获得这条回路上的所有结点.把结点序列(例如,142857)存入thiscycle中;由于这种算法中,一条回路会被发现好几次,所以必须先判断该回路是否已经在cycles中被记录过,如果没有才能存入cycles的一个行向量中.把cycles的每一个行向量取出来与之比较.由于一条回路可能有多种存储顺序,比如142857等同于285714和571428,所以还要调整行向量的次序,并存入temp数组,例如,thiscycle为142857第一个结点为1,cycles的当前向量为857142,则找到后者中的1,把1后部分提到1前部分前面,最终在temp中得到142857,与thiscycle比较,发现相同,因此142857和857142是同一条回路,不予存储.这个算法太复杂,很难保证细节的准确性,大家理解思路便可.希望有人给出更加简捷的算法.   
7.31   
int visited[MAXSIZE];  
int finished[MAXSIZE];  
int count; //count在第一次深度优先遍历中用于指示finished数组的填充位置   
void Get\_SGraph(OLGraph G)//求十字链表结构储存的有向图G的强连通分量  
{  
  count=0;  
  for(v=0;v<G.vexnum;v++) visited[v]=0;  
  for(v=0;v<G.vexnum;v++) //第一次深度优先遍历建立finished数组  
    if(!visited[v]) DFS1(G,v);  
  for(v=0;v<G.vexnum;v++) visited[v]=0; //清空visited数组  
  for(i=G.vexnum-1;i>=0;i--) //第二次逆向的深度优先遍历  
  {  
    v=finished(i);  
    if(!visited[v])  
    {  
      printf("\n"); //不同的强连通分量在不同的行输出  
      DFS2(G,v);  
    }  
  }//for  
}//Get\_SGraph   
void DFS1(OLGraph G,int v)//第一次深度优先遍历的算法  
{  
  visited[v]=1;  
  for(p=G.xlist[v].firstout;p;p=p->tlink)  
  {  
    w=p->headvex;  
    if(!visited[w]) DFS1(G,w);  
  }//for  
  finished[++count]=v; //在第一次遍历中建立finished数组  
}//DFS1   
void DFS2(OLGraph G,int v)//第二次逆向的深度优先遍历的算法  
{  
  visited[v]=1;  
  printf("%d",v); //在第二次遍历中输出结点序号  
  for(p=G.xlist[v].firstin;p;p=p->hlink)  
  {  
    w=p->tailvex;  
    if(!visited[w]) DFS2(G,w);  
  }//for  
}//DFS2  
分析:求有向图的强连通分量的算法的时间复杂度和深度优先遍历相同,也为O(n+e).   
7.32   
void Forest\_Prim(ALGraph G,int k,CSTree &T)//从顶点k出发,构造邻接表结构的有向图G的最小生成森林T,用孩子兄弟链表存储  
{  
  for(j=0;j<G.vexnum;j++) //以下在Prim算法基础上稍作改动  
    if(j!=k)  
    {  
      closedge[j]={k,Max\_int};  
      for(p=G.vertices[j].firstarc;p;p=p->nextarc)  
        if(p->adjvex==k) closedge[j].lowcost=p->cost;  
    }//if  
  closedge[k].lowcost=0;  
  for(i=1;i<G.vexnum;i++)  
  {  
    k=minimum(closedge);  
    if(closedge[k].lowcost<Max\_int)  
    {  
      Addto\_Forest(T,closedge[k].adjvex,k); //把这条边加入生成森林中  
      closedge[k].lowcost=0;  
      for(p=G.vertices[k].firstarc;p;p=p->nextarc)  
        if(p->cost<closedge[p->adjvex].lowcost)  
          closedge[p->adjvex]={k,p->cost};  
    }//if  
    else Forest\_Prim(G,k); //对另外一个连通分量执行算法  
  }//for  
}//Forest\_Prim   
void Addto\_Forest(CSTree &T,int i,int j)//把边(i,j)添加到孩子兄弟链表表示的树T中  
{  
  p=Locate(T,i); //找到结点i对应的指针p,过程略  
  q=(CSTNode\*)malloc(sizeof(CSTNode));  
  q->data=j;  
  if(!p) //起始顶点不属于森林中已有的任何一棵树  
  {  
    p=(CSTNode\*)malloc(sizeof(CSTNode));  
    p->data=i;  
    for(r=T;r->nextsib;r=r->nextsib);  
    r->nextsib=p;  
    p->firstchild=q;  
  } //作为新树插入到最右侧  
  else if(!p->firstchild) //双亲还没有孩子  
    p->firstchild=q; //作为双亲的第一个孩子  
  else //双亲已经有了孩子  
  {  
    for(r=p->firstchild;r->nextsib;r=r->nextsib);  
    r->nextsib=q; //作为双亲最后一个孩子的兄弟  
  }  
}//Addto\_Forest   
main()  
{  
  ...  
  T=(CSTNode\*)malloc(sizeof(CSTNode)); //建立树根  
  T->data=1;  
  Forest\_Prim(G,1,T);  
  ...  
}//main  
分析:这个算法是在Prim算法的基础上添加了非连通图支持和孩子兄弟链表构建模块而得到的,其时间复杂度为O(n^2).   
7.33   
typedef struct {  
                     int vex; //结点序号  
                     int ecno; //结点所属的连通分量号  
                   } VexInfo;  
VexInfo vexs[MAXSIZE]; //记录结点所属连通分量号的数组   
void Init\_VexInfo(VexInfo &vexs[ ],int vexnum)//初始化  
{  
  for(i=0;i<vexnum;i++)  
    vexs[i]={i,i}; //初始状态:每一个结点都属于不同的连通分量  
}//Init\_VexInfo   
int is\_ec(VexInfo vexs[ ],int i,int j)//判断顶点i和顶点j是否属于同一个连通分量  
{  
  if(vexs[i].ecno==vexs[j].ecno) return 1;  
  else return 0;  
}//is\_ec   
void merge\_ec(VexInfo &vexs[ ],int ec1,int ec2)//合并连通分量ec1和ec2  
{  
  for(i=0;vexs[i].vex;i++)  
    if(vexs[i].ecno==ec2) vexs[i].ecno==ec1;  
}//merge\_ec   
void MinSpanTree\_Kruscal(Graph G,EdgeSetType &EdgeSet,CSTree &T)//求图的最小生成树的克鲁斯卡尔算法  
{  
  Init\_VexInfo(vexs,G.vexnum);  
  ecnum=G.vexnum; //连通分量个数  
  while(ecnum>1)  
  {  
    GetMinEdge(EdgeSet,u,v); //选出最短边  
    if(!is\_ec(vexs,u,v)) //u和v属于不同连通分量  
    {  
      Addto\_CSTree(T,u,v); //加入到生成树中  
      merge\_ec(vexs,vexs[u].ecno,vexs[v].ecno); //合并连通分量  
      ecnum--;  
    }  
    DelMinEdge(EdgeSet,u,v); //从边集中删除  
  }//while  
}//MinSpanTree\_Kruscal   
void Addto\_CSTree(CSTree &T,int i,int j)//把边(i,j)添加到孩子兄弟链表表示的树T中  
{  
  p=Locate(T,i); //找到结点i对应的指针p,过程略  
  q=(CSTNode\*)malloc(sizeof(CSTNode));  
  q->data=j;  
  if(!p->firstchild) //双亲还没有孩子  
    p->firstchild=q; //作为双亲的第一个孩子  
  else //双亲已经有了孩子  
  {  
    for(r=p->firstchild;r->nextsib;r=r->nextsib);  
    r->nextsib=q; //作为双亲最后一个孩子的兄弟  
  }  
}//Addto\_CSTree  
分析:本算法使用一维结构体变量数组来表示等价类,每个连通分量所包含的所有结点属于一个等价类.在这个结构上实现了初始化,判断元素是否等价(两个结点是否属于同一个连通分量),合并等价类(连通分量)的操作.   
7.34   
Status TopoSeq(ALGraph G,int new[ ])//按照题目要求给有向无环图的结点重新编号,并存入数组new中  
{  
  int indegree[MAXSIZE]; //本算法就是拓扑排序  
  FindIndegree(G,indegree);  
  Initstack(S);  
  for(i=0;i<G.vexnum;i++)  
    if(!indegree[i]) Push(S,i);  
  count=0;  
  while(!stackempty(S))  
  {  
    Pop(S,i);new[i]=++count; //把拓扑顺序存入数组的对应分量中  
    for(p=G.vertices[i].firstarc;p;p=p->nextarc)  
    {  
      k=p->adjvex;  
      if(!(--indegree[k])) Push(S,k);  
    }  
  }//while  
  if(count<G.vexnum) return ERROR;  
  return OK;  
}//TopoSeq   
7.35   
int visited[MAXSIZE];   
void Get\_Root(ALGraph G)//求有向无环图的根,如果有的话  
{  
  for(v=0;v<G.vexnum;v++)  
  {  
    for(w=0;w<G.vexnum;w++) visited[w]=0;//每次都要将访问数组清零  
    DFS(G,v); //从顶点v出发进行深度优先遍历  
    for(flag=1,w=0;w<G.vexnum;w++)  
      if(!visited[w]) flag=0; //如果v是根,则深度优先遍历可以访问到所有结点  
    if(flag) printf("Found a root vertex:%d\n",v);  
  }//for  
}//Get\_Root,这个算法要求图中不能有环,否则会发生误判   
void DFS(ALGraph G,int v)  
{  
  visited[v]=1;  
  for(p=G.vertices[v].firstarc;p;p=p->nextarc)  
  {  
    w=p->adjvex;  
    if(!visited[w]) DFS(G,w);  
  }  
}//DFS   
7.36   
void Fill\_MPL(ALGraph &G)//为有向无环图G添加MPL域  
{  
  FindIndegree(G,indegree);  
  for(i=0;i<G.vexnum;i++)  
    if(!indegree[i]) Get\_MPL(G,i);//从每一个零入度顶点出发构建MPL域  
}//Fill\_MPL   
int Get\_MPL(ALGraph &G,int i)//从一个顶点出发构建MPL域并返回其MPL值  
{  
  if(!G.vertices[i].firstarc)  
  {  
    G.vertices[i].MPL=0;  
    return 0; //零出度顶点  
  }  
  else  
  {  
    max=0;  
    for(p=G.vertices[i].firstarc;p;p=p->nextarc)  
    {  
      j=p->adjvex;  
      if(G.vertices[j].MPL==0) k=Get\_MPL(G,j);  
      if(k>max) max=k; //求其直接后继顶点MPL的最大者  
    }  
    G.vertices[i]=max+1;//再加一,就是当前顶点的MPL  
    return max+1;  
  }//else  
}//Get\_MPL   
7.37   
int maxlen,path[MAXSIZE]; //数组path用于存储当前路径  
int mlp[MAXSIZE]; //数组mlp用于存储已发现的最长路径   
void Get\_Longest\_Path(ALGraph G)//求一个有向无环图中最长的路径  
{  
  maxlen=0;  
  FindIndegree(G,indegree);  
  for(i=0;i<G.vexnum;i++)  
  {  
    for(j=0;j<G.vexnum;j++) visited[j]=0;  
    if(!indegree[i]) DFS(G,i,0);//从每一个零入度结点开始深度优先遍历  
  }  
  printf("Longest Path:");  
  for(i=0;mlp[i];i++) printf("%d",mlp[i]); //输出最长路径  
}//Get\_Longest\_Path   
void DFS(ALGraph G,int i,int len)  
{  
  visited[i]=1;  
  path[len]=i;  
  if(len>maxlen&&!G.vertices[i].firstarc) //新的最长路径  
  {  
    for(j=0;j<=len;j++) mlp[j]=path[j]; //保存下来  
    maxlen=len;  
  }  
  else  
  {  
    for(p=G.vertices[i].firstarc;p;p=p->nextarc)  
    {  
      j=p->adjvex;  
      if(!visited[j]) DFS(G,j,len+1);  
    }  
  }//else  
  path[i]=0;  
  visited[i]=0;  
}//DFS   
7.38   
void NiBoLan\_DAG(ALGraph G)//输出有向无环图形式表示的表达式的逆波兰式  
{  
  FindIndegree(G,indegree);  
  for(i=0;i<G.vexnum;i++)  
    if(!indegree[i]) r=i; //找到有向无环图的根  
  PrintNiBoLan\_DAG(G,i);  
}//NiBoLan\_DAG   
void PrintNiBoLan\_DAG(ALGraph G,int i)//打印输出以顶点i为根的表达式的逆波兰式  
{  
  c=G.vertices[i].data;  
  if(!G.vertices[i].firstarc) //c是原子  
    printf("%c",c);  
  else //子表达式  
  {  
    p=G.vertices[i].firstarc;  
    PrintNiBoLan\_DAG(G,p->adjvex);  
    PrintNiBoLan\_DAG(G,p->nexarc->adjvex);  
    printf("%c",c);  
  }  
}//PrintNiBoLan\_DAG   
7.39   
void PrintNiBoLan\_Bitree(Bitree T)//在二叉链表存储结构上重做上一题  
{  
  if(T->lchild) PrintNiBoLan\_Bitree(T->lchild);  
  if(T->rchild) PrintNiBoLan\_Bitree(T->rchild);  
  printf("%c",T->data);  
}//PrintNiBoLan\_Bitree   
7.40   
int Evaluate\_DAG(ALGraph G)//给有向无环图表示的表达式求值  
{  
  FindIndegree(G,indegree);  
  for(i=0;i<G.vexnum;i++)  
    if(!indegree[i]) r=i; //找到有向无环图的根  
  return Evaluate\_imp(G,i);  
}//NiBoLan\_DAG   
int Evaluate\_imp(ALGraph G,int i)//求子表达式的值  
{  
  if(G.vertices[i].tag=NUM) return G.vertices[i].value;  
  else  
  {  
    p=G.vertices[i].firstarc;  
    v1=Evaluate\_imp(G,p->adjvex);  
    v2=Evaluate\_imp(G,p->nextarc->adjvex);  
    return calculate(v1,G.vertices[i].optr,v2);  
  }  
}//Evaluate\_imp  
分析:本题中,邻接表的vertices向量的元素类型修改如下:  
struct {  
         enum tag{NUM,OPTR};  
         union {  
                 int value;  
                 char optr;  
               };  
         ArcNode \* firstarc;  
       } Elemtype;   
7.41   
void Critical\_Path(ALGraph G)//利用深度优先遍历求网的关键路径  
{  
  FindIndegree(G,indegree);  
  for(i=0;i<G.vexnum;i++)  
    if(!indegree[i]) DFS1(G,i); //第一次深度优先遍历:建立ve  
  for(i=0;i<G.vexnum;i++)  
    if(!indegree[i]) DFS2(G,i); //第二次深度优先遍历:建立vl  
  for(i=0;i<=G.vexnum;i++)  
    if(vl[i]==ve[i]) printf("%d",i); //打印输出关键路径  
}//Critical\_Path   
void DFS1(ALGraph G,int i)  
{  
  if(!indegree[i]) ve[i]=0;  
  for(p=G.vertices[i].firstarc;p;p=p->nextarc)  
  {  
    dut=\*p->info;  
    if(ve[i]+dut>ve[p->adjvex])  
      ve[p->adjvex]=ve[i]+dut;  
    DFS1(G,p->adjvex);  
  }  
}//DFS1   
void DFS2(ALGraph G,int i)  
{  
  if(!G.vertices[i].firstarc) vl[i]=ve[i];  
  else  
  {  
    for(p=G.vertices[i].firstarc;p;p=p->nextarc)  
    {  
      DFS2(G,p->adjvex);  
      dut=\*p->info;  
      if(vl[p->adjvex]-dut<vl[i])  
        vl[i]=vl[p->adjvex]-dut;  
    }  
  }//else  
}//DFS2   
7.42   
void ALGraph\_DIJ(ALGraph G,int v0,Pathmatrix &P,ShortestPathTable &D)//在邻接表存储结构上实现迪杰斯特拉算法  
{  
  for(v=0;v<G.vexnum;v++)  
    D[v]=INFINITY;  
  for(p=G.vertices[v0].firstarc;p;p=p->nextarc)  
    D[p->adjvex]=\*p->info; //给D数组赋初值  
  for(v=0;v<G.vexnum;v++)  
  {  
    final[v]=0;  
    for(w=0;w<G.vexnum;w++) P[v][w]=0; //设空路径  
    if(D[v]<INFINITY)  
    {  
      P[v][v0]=1;  
      P[v][v]=1;  
    }  
  }//for  
  D[v0]=0;final[v0]=1; //初始化  
  for(i=1;i<G.vexnum;i++)  
  {  
    min=INFINITY;  
    for(w=0;w<G.vexnum;w++)  
      if(!final[w])  
        if(D[w]<min) //尚未求出到该顶点的最短路径  
        {  
          v=w;  
          min=D[w];  
        }  
    final[v]=1;  
    for(p=G.vertices[v].firstarc;p;p=p->nextarc)  
    {  
      w=p->adjvex;  
      if(!final[w]&&(min+(\*p->info)<D[w])) //符合迪杰斯特拉条件  
      {  
        D[w]=min+edgelen(G,v,w);  
        P[w]=P[v];  
        P[w][w]=1; //构造最短路径  
      }  
    }//for  
  }//for  
}//ALGraph\_DIJ  
分析:本算法对迪杰斯特拉算法中直接取任意边长度的语句作了修改.由于在原算法中,每次循环都是对尾相同的边进行处理,所以可以用遍历邻接表中的一条链来代替.

第八章 动态存储管理  
  
8.11   
typedef struct {  
                     char \*start;  
                     int size;  
                  } fmblock; //空闲块类型   
char \*Malloc\_Fdlf(int n)//遵循最后分配者最先释放规则的内存分配算法  
{  
  while(Gettop(S,b)&&b.size<n)  
  {  
    Pop(S,b);  
    Push(T,b); //从栈顶逐个取出空闲块进行比较  
  }  
  if(StackEmpty(S)) return NULL; //没有大小足够的空闲块  
  Pop(S,b);  
  b.size-=n;  
  if(b.size) Push(S,{b.start+n,b.size});//分割空闲块  
  while(!StackEmpty(T))  
  {  
    Pop(T,a);  
    Push(S,a);  
  } //恢复原来次序  
  return b.start;  
}//Malloc\_Fdlf   
mem\_init()//初始化过程  
{  
  ...  
  InitStack(S);InitStack(T); //S和T的元素都是fmblock类型  
  Push(S,{MemStart,MemLen}); //一开始,栈中只有一个内存整块  
  ...  
}//main   
8.12   
void Free\_Fdlf(char \*addr,int n)//与上一题对应的释放算法  
{  
  while(Gettop(S,b)&&b.start<addr)  
  {  
    Pop(S,b);  
    Push(T,b);  
  } //在按地址排序的栈中找到合适的插入位置  
  if(Gettop(T,b)&&(b.start+b.size==addr)) //可以与上邻块合并  
  {  
    Pop(T,b);  
    addr=b.start;n+=b.size;  
  }  
  if(Gettop(S,b)&&(addr+n==b.start)) //可以与下邻块合并  
  {  
    Pop(S,b);  
    n+=b.size;  
  }  
  Push(S,{addr,n}); //插入到空闲块栈中  
  while(!StackEmpty(T))  
  {  
    Pop(T,b);  
    Push(S,b);  
  } //恢复原来次序  
}//Free\_Fdlf   
8.13   
void Free\_BT(Space &pav,Space p)//在边界标识法的动态存储管理系统中回收空闲块p  
{  
  n=p->size;  
  f=p+n-1; //f指向空闲块底部  
  if((p-1)->tag&&(f+1)->tag) //回收块上下邻块均为占用块  
  {  
    p->tag=0;f->tag=0;  
    f->uplink=p;  
    if(!pav)  
    {  
      p->llink=p;  
      p->rlink=p;  
    }  
    else  
    {  
      q=pav->llink;  
      p->llink=q;p->rlink=pav;  
      q->rlink=p;pav->llink=p;  
    }  
    pav=p;  
  }//if  
  else if(!(p-1)->tag&&(f+1)->tag) //上邻块为空闲块  
  {  
    q=(p-1)->uplink;  
    q->size+=n;  
    f->uplink=q;  
    f->tag=0;  
  }  
  else if((p-1)->tag&&!(f+1)->tag) //下邻块为空闲块  
  {  
    q=f+1;  
    s=q->llink;t=q->rlink;  
    p->llink=s;p->rlink=t;  
    s->rlink=p;t->llink=p;  
    p->size+=q->size;  
    (q+q->size-1)->uplink=p;  
    p->tag=0;  
  }  
  else //上下邻块均为空闲块  
  {  
    s=(p-1)->uplink;  
    t=f+1;  
    s->size+=n+t->size;  
    t->llink->rlink=t->rlink;  
    t->rlink->llink=t->llink;  
    (t+t->size-1)->uplink=s;  
  }  
}//Free\_BT,该算法在课本里有详细的描述.   
8.14   
void Free\_BS(freelist &avail,char \*addr,int n)//伙伴系统的空闲块回收算法  
{  
  buddy=addr%(2\*n)?(addr-n):(addr+n); //求回收块的伙伴地址  
  addr->tag=0;  
  addr->kval=n;  
  for(i=0;avail[i].nodesize<n;i++); //找到这一大小的空闲块链  
  if(!avail[i].first) //尚没有该大小的空闲块  
  {  
   addr->llink=addr;  
   addr->rlink=addr;  
   avail[i].first=addr; //作为唯一一个该大小的空闲块  
  }  
  else  
  {  
    for(p=avail[i].first;p!=buddy&&p!=avail[i].first;p=p->rlink);//寻找伙伴  
    if(p==buddy) //伙伴为空闲块,此时进行合并  
    {  
      if(p->rlink==p) avail[i].first=NULL;//伙伴是此大小的唯一空闲块  
      else  
      {  
        p->llink->rlink=p->rlink;  
        p->rlink->llink=p->llink;  
      } //从空闲块链中删去伙伴  
      new=addr>p?p:addr; //合并后的新块首址  
      Free\_BS(avail,new,2\*n); //递归地回收新块  
    }//if  
    else //伙伴为占用块,此时插入空闲块链头部  
    {  
      q=p->rlink;  
      p->rlink=addr;addr->llink=p;  
      q->llink=addr;addr->rlink=q;  
    }  
  }//else  
}//Free\_BS   
8.15   
FBList \*MakeList(char \*highbound,char \*lowbound)//把堆结构存储的的所有空闲块链接成可利用空间表,并返回表头指针  
{  
  p=lowbound;  
  while(p->tag&&p<highbound) p++; //查找第一个空闲块  
  if(p>=highbound) return NULL; //没有空闲块  
  head=p;  
  for(q=p;p<highbound;p+=cellsize) //建立链表  
    if(!p->tag)  
    {  
      q->next=p;  
      q=p;  
    }//if  
  p->next=NULL;  
  return head; //返回头指针  
}//MakeList   
8.16   
void Mem\_Contract(Heap &H)//对堆H执行存储紧缩  
{  
  q=MemStart;j=0;  
  for(i=0;i<Max\_ListLen;i++)  
    if(H.list[i].stadr->tag)  
    {  
      s=H.list[i].length;  
      p=H.list[i].stadr;  
      for(k=0;k<s;k++) \*(q++)=\*(p++); //紧缩内存空间  
      H.list[j].stadr=q;  
      H.list[j].length=s; //紧缩占用空间表  
      j++;  
    }  
}//Mem\_Contract

第九章 查找  
  
9.25   
int Search\_Sq(SSTable ST,int key)//在有序表上顺序查找的算法,监视哨设在高下标端  
{  
  ST.elem[ST.length+1].key=key;  
  for(i=1;ST.elem[i].key>key;i++);  
  if(i>ST.length||ST.elem[i].key<key) return ERROR;  
  return i;  
}//Search\_Sq  
分析:本算法查找成功情况下的平均查找长度为ST.length/2,不成功情况下为ST.length.   
9.26   
int Search\_Bin\_Recursive(SSTable ST,int key,int low,int high)//折半查找的递归算法  
{  
  if(low>high) return 0; //查找不到时返回0  
  mid=(low+high)/2;  
  if(ST.elem[mid].key==key) return mid;  
  else if(ST.elem[mid].key>key)  
    return Search\_Bin\_Recursive(ST,key,low,mid-1);  
  else return Search\_Bin\_Recursive(ST,key,mid+1,high);  
  }  
}//Search\_Bin\_Recursive   
9.27   
int Locate\_Bin(SSTable ST,int key)//折半查找,返回小于或等于待查元素的最后一个结点号  
{  
  int \*r;  
  r=ST.elem;  
  if(key<r .key) return 0;  
  else if(key>=r[ST.length].key) return ST.length;  
  low=1;high=ST.length;  
  while(low<=high)  
  {  
    mid=(low+high)/2;  
    if(key>=r[mid].key&&key<r[mid+1].key) //查找结束的条件  
      return mid;  
    else if(key<r[mid].key) high=mid;  
    else low=mid;  
  } //本算法不存在查找失败的情况,不需要return 0;  
}//Locate\_Bin   
9.28   
typedef struct {  
                     int maxkey;  
                     int firstloc;  
                   } Index;   
typedef struct {  
                     int \*elem;  
                     int length;  
                     Index idx[MAXBLOCK]; //每块起始位置和最大元素,其中idx[0]不利用,其内容初始化为{0,0}以利于折半查找  
                     int blknum; //块的数目  
                   } IdxSqList; //索引顺序表类型   
int Search\_IdxSeq(IdxSqList L,int key)//分块查找,用折半查找法确定记录所在块,块内采用顺序查找法  
{  
  if(key>L.idx[L.blknum].maxkey) return ERROR; //超过最大元素  
  low=1;high=L.blknum;  
  found=0;  
  while(low<=high&&!found) //折半查找记录所在块号mid  
  {  
    mid=(low+high)/2;  
    if(key<=L.idx[mid].maxkey&&key>L.idx[mid-1].maxkey)  
      found=1;  
    else if(key>L.idx[mid].maxkey)  
      low=mid+1;  
    else high=mid-1;  
  }  
  i=L.idx[mid].firstloc; //块的下界  
  j=i+blksize-1; //块的上界  
  temp=L.elem[i-1]; //保存相邻元素  
  L.elem[i-1]=key; //设置监视哨  
  for(k=j;L.elem[k]!=key;k--); //顺序查找  
  L.elem[i-1]=temp; //恢复元素  
  if(k<i) return ERROR; //未找到  
  return k;  
}//Search\_IdxSeq  
分析:在块内进行顺序查找时,如果需要设置监视哨,则必须先保存相邻块的相邻元素,以免数据丢失.   
9.29   
typedef struct {  
                     LNode \*h; //h指向最小元素  
                     LNode \*t; //t指向上次查找的结点  
                  } CSList;   
LNode \*Search\_CSList(CSList &L,int key)//在有序单循环链表存储结构上的查找算法,假定每次查找都成功  
{  
  if(L.t->data==key) return L.t;  
  else if(L.t->data>key)  
    for(p=L.h,i=1;p->data!=key;p=p->next,i++);  
  else  
    for(p=L.t,i=L.tpos;p->data!=key;p=p->next,i++);  
  L.t=p; //更新t指针  
  return p;  
}//Search\_CSList  
分析:由于题目中假定每次查找都是成功的,所以本算法中没有关于查找失败的处理.由微积分可得,在等概率情况下,平均查找长度约为n/3.   
9.30   
typedef struct {  
                     DLNode \*pre;  
                     int data;  
                     DLNode \*next;  
                  } DLNode;   
typedef struct {  
                     DLNode \*sp;  
                     int length;  
                  } DSList; //供查找的双向循环链表类型   
DLNode \*Search\_DSList(DSList &L,int key)//在有序双向循环链表存储结构上的查找算法,假定每次查找都成功  
{  
  p=L.sp;  
  if(p->data>key)  
  {  
    while(p->data>key) p=p->pre;  
    L.sp=p;  
  }  
  else if(p->data<key)  
  {  
    while(p->data<key) p=p->next;  
    L.sp=p;  
  }  
  return p;  
}//Search\_DSList  
分析:本题的平均查找长度与上一题相同,也是n/3.   
9.31   
int last=0,flag=1;   
int Is\_BSTree(Bitree T)//判断二叉树T是否二叉排序树,是则返回1,否则返回0  
{  
  if(T->lchild&&flag) Is\_BSTree(T->lchild);  
  if(T->data<last) flag=0; //与其中序前驱相比较  
  last=T->data;  
  if(T->rchild&&flag) Is\_BSTree(T->rchild);  
  return flag;  
}//Is\_BSTree   
9.32   
int last=0;   
void MaxLT\_MinGT(BiTree T,int x)//找到二叉排序树T中小于x的最大元素和大于x的最小元素  
{  
  if(T->lchild) MaxLT\_MinGT(T->lchild,x); //本算法仍是借助中序遍历来实现  
  if(last<x&&T->data>=x) //找到了小于x的最大元素  
    printf("a=%d\n",last);  
  if(last<=x&&T->data>x) //找到了大于x的最小元素  
    printf("b=%d\n",T->data);  
  last=T->data;  
  if(T->rchild) MaxLT\_MinGT(T->rchild,x);  
}//MaxLT\_MinGT   
9.33   
void Print\_NLT(BiTree T,int x)//从大到小输出二叉排序树T中所有不小于x的元素  
{  
  if(T->rchild) Print\_NLT(T->rchild,x);  
  if(T->data<x) exit(); //当遇到小于x的元素时立即结束运行  
  printf("%d\n",T->data);  
  if(T->lchild) Print\_NLT(T->lchild,x); //先右后左的中序遍历  
}//Print\_NLT   
9.34   
void Delete\_NLT(BiTree &T,int x)//删除二叉排序树T中所有不小于x元素结点,并释放空间  
{  
  if(T->rchild) Delete\_NLT(T->rchild,x);  
  if(T->data<x) exit(); //当遇到小于x的元素时立即结束运行  
  q=T;  
  T=T->lchild;  
  free(q); //如果树根不小于x,则删除树根,并以左子树的根作为新的树根  
  if(T) Delete\_NLT(T,x); //继续在左子树中执行算法  
}//Delete\_NLT   
9.35   
void Print\_Between(BiThrTree T,int a,int b)//打印输出后继线索二叉排序树T中所有大于a且小于b的元素  
{  
  p=T;  
  while(!p->ltag) p=p->lchild; //找到最小元素  
  while(p&&p->data<b)  
  {  
    if(p->data>a) printf("%d\n",p->data); //输出符合条件的元素  
    if(p->rtag) p=p->rtag;  
    else  
    {  
      p=p->rchild;  
      while(!p->ltag) p=p->lchild;  
    } //转到中序后继  
  }//while  
}//Print\_Between   
9.36   
void BSTree\_Insert\_Key(BiThrTree &T,int x)//在后继线索二叉排序树T中插入元素x  
{  
  if(T->data<x) //插入到右侧  
  {  
    if(T->rtag) //T没有右子树时,作为右孩子插入  
    {  
      p=T->rchild;  
      q=(BiThrNode\*)malloc(sizeof(BiThrNode));  
      q->data=x;  
      T->rchild=q;T->rtag=0;  
      q->rtag=1;q->rchild=p; //修改原线索  
    }  
    else BSTree\_Insert\_Key(T->rchild,x);//T有右子树时,插入右子树中  
  }//if  
  else if(T->data>x) //插入到左子树中  
  {  
    if(!T->lchild) //T没有左子树时,作为左孩子插入  
    {  
      q=(BiThrNode\*)malloc(sizeof(BiThrNode));  
      q->data=x;  
      T->lchild=q;  
      q->rtag=1;q->rchild=T; //修改自身的线索  
    }  
    else BSTree\_Insert\_Key(T->lchild,x);//T有左子树时,插入左子树中  
  }//if  
}//BSTree\_Insert\_Key   
9.37   
Status BSTree\_Delete\_key(BiThrTree &T,int x)//在后继线索二叉排序树T中删除元素x  
{  
  BTNode \*pre,\*ptr,\*suc;//ptr为x所在结点,pre和suc分别指向ptr的前驱和后继  
  p=T;last=NULL; //last始终指向当前结点p的前一个(前驱)  
  while(!p->ltag) p=p->lchild; //找到中序起始元素  
  while(p)  
  {  
    if(p->data==x) //找到了元素x结点  
    {  
      pre=last;  
      ptr=p;  
    }  
    else if(last&&last->data==x) suc=p; //找到了x的后继  
    if(p->rtag) p=p->rtag;  
    else  
    {  
      p=p->rchild;  
      while(!p->ltag) p=p->lchild;  
    } //转到中序后继  
    last=p;  
  }//while //借助中序遍历找到元素x及其前驱和后继结点  
  if(!ptr) return ERROR; //未找到待删结点  
  Delete\_BSTree(ptr); //删除x结点  
  if(pre&&pre->rtag)  
    pre->rchild=suc; //修改线索  
  return OK;  
}//BSTree\_Delete\_key   
void Delete\_BSTree(BiThrTree &T)//课本上给出的删除二叉排序树的子树T的算法,按照线索二叉树的结构作了一些改动  
{  
  q=T;  
  if(!T->ltag&&T->rtag) //结点无右子树,此时只需重接其左子树  
    T=T->lchild;  
  else if(T->ltag&&!T->rtag) //结点无左子树,此时只需重接其右子树  
    T=T->rchild;  
  else if(!T->ltag&&!T->rtag) //结点既有左子树又有右子树  
  {  
    p=T;r=T->lchild;  
    while(!r->rtag)  
    {  
      s=r;  
      r=r->rchild; //找到结点的前驱r和r的双亲s  
    }  
    T->data=r->data; //用r代替T结点  
    if(s!=T)  
      s->rchild=r->lchild;  
    else s->lchild=r->lchild; //重接r的左子树到其双亲结点上  
    q=r;  
  }//else  
  free(q); //删除结点  
}//Delete\_BSTree  
分析:本算法采用了先求出x结点的前驱和后继,再删除x结点的办法,这样修改线索时会比较简单,直接让前驱的线索指向后继就行了.如果试图在删除x结点的同时修改线索,则问题反而复杂化了.   
9.38   
void BSTree\_Merge(BiTree &T,BiTree &S)//把二叉排序树S合并到T中  
{  
  if(S->lchild) BSTree\_Merge(T,S->lchild);  
  if(S->rchild) BSTree\_Merge(T,S->rchild); //合并子树  
  Insert\_Key(T,S); //插入元素  
}//BSTree\_Merge   
void Insert\_Node(Bitree &T,BTNode \*S)//把树结点S插入到T的合适位置上  
{  
  if(S->data>T->data)  
  {  
    if(!T->rchild) T->rchild=S;  
    else Insert\_Node(T->rchild,S);  
  }  
  else if(S->data<T->data)  
  {  
    if(!T->lchild) T->lchild=S;  
    else Insert\_Node(T->lchild,S);  
  }  
  S->lchild=NULL; //插入的新结点必须和原来的左右子树断绝关系  
  S->rchild=NULL; //否则会导致树结构的混乱  
}//Insert\_Node  
分析:这是一个与课本上不同的插入算法.在合并过程中,并不释放或新建任何结点,而是采取修改指针的方式来完成合并.这样,就必须按照后序序列把一棵树中的元素逐个连接到另一棵树上,否则将会导致树的结构的混乱.   
9.39   
void BSTree\_Split(BiTree &T,BiTree &A,BiTree &B,int x)//把二叉排序树T分裂为两棵二叉排序树A和B,其中A的元素全部小于等于x,B的元素全部大于x  
{  
  if(T->lchild) BSTree\_Split(T->lchild,A,B,x);  
  if(T->rchild) BSTree\_Split(T->rchild,A,B,x); //分裂左右子树  
  if(T->data<=x) Insert\_Node(A,T);  
  else Insert\_Node(B,T); //将元素结点插入合适的树中  
}//BSTree\_Split   
void Insert\_Node(Bitree &T,BTNode \*S)//把树结点S插入到T的合适位置上  
{  
  if(!T) T=S; //考虑到刚开始分裂时树A和树B为空的情况  
  else if(S->data>T->data) //其余部分与上一题同  
  {  
    if(!T->rchild) T->rchild=S;  
    else Insert\_Node(T->rchild,S);  
  }  
  else if(S->data<T->data)  
  {  
    if(!T->lchild) T->lchild=S;  
    else Insert\_Node(T->lchild,S);  
  }  
  S->lchild=NULL;  
  S->rchild=NULL;    
}//Insert\_Key   
9.40   
typedef struct {  
                     int data;  
                     int bf;  
                     int lsize; //lsize域表示该结点的左子树的结点总数加1  
                     BlcNode \*lchild,\*rchild;  
                  } BlcNode,\*BlcTree; //含lsize域的平衡二叉排序树类型   
BTNode \*Locate\_BlcTree(BlcTree T,int k)//在含lsize域的平衡二叉排序树T中确定第k小的结点指针  
{  
  if(!T) return NULL; //k小于1或大于树结点总数  
  if(T->lsize==k) return T; //就是这个结点  
  else if(T->lsize>k)  
    return Locate\_BlcTree(T->lchild,k); //在左子树中寻找  
  else return Locate\_BlcTree(T->rchild,k-T->lsize); //在右子树中寻找,注意要修改k的值  
}//Locate\_BlcTree   
9.41   
typedef struct {  
                 enum {LEAF,BRANCH} tag; //结点类型标识  
                 int keynum;  
                 BPLink parent; //双亲指针  
                 int key[MAXCHILD]; //关键字  
                 union {  
                         BPLink child[MAXCHILD];//非叶结点的孩子指针  
                         struct {  
                                  rectype \*info[MAXCHILD];//叶子结点的信息指针  
                                  BPNode \*next; //指向下一个叶子结点的链接  
                                  } leaf;  
                         }  
              } BPNode,\*BPLink,\*BPTree;//B+树及其结点类型   
Status BPTree\_Search(BPTree T,int key,BPNode \*ptr,int pos)//B+树中按关键字随机查找的算法,返回包含关键字的叶子结点的指针ptr以及关键字在叶子结点中的位置pos  
{  
  p=T;  
  while(p.tag==BRANCH) //沿分支向下查找  
  {  
    for(i=0;i<p->keynum&&key>p->key[i];i++); //确定关键字所在子树  
    if(i==p->keynum) return ERROR; //关键字太大  
    p=p->child[i];  
  }  
  for(i=0;i<p->keynum&&key!=p->key[i];i++); //在叶子结点中查找  
  if(i==p->keynum) return ERROR; //找不到关键字  
  ptr=p;pos=i;  
  return OK;  
}//BPTree\_Search     
9.42   
void TrieTree\_Insert\_Key(TrieTree &T,StringType key)//在Trie树T中插入字符串key,StringType的结构见第四章  
{  
  q=(TrieNode\*)malloc(sizeof(TrieNode));  
  q->kind=LEAF;  
  q->lf.k=key; //建叶子结点  
  klen=key[0];  
  p=T;i=1;  
  while(p&&i<=klen&&p->bh.ptr[ord(key[i])])  
  {  
    last=p;  
    p=p->bh.ptr[ord(key[i])];  
    i++;  
  } //自上而下查找  
  if(p->kind==BRANCH) //如果最后落到分支结点(无同义词):  
  {  
    p->bh.ptr[ord(key[i])]=q; //直接连上叶子  
    p->bh.num++;  
  }  
  else //如果最后落到叶子结点(有同义词):  
  {  
    r=(TrieNode\*)malloc(sizeof(TrieNode)); //建立新的分支结点  
    last->bh.ptr[ord(key[i-1])]=r; //用新分支结点取代老叶子结点和上一层的联系  
    r->kind=BRANCH;r->bh.num=2;  
    r->bh.ptr[ord(key[i])]=q;  
    r->bh.ptr[ord(p->lf.k[i])]=p; //新分支结点与新老两个叶子结点相连  
  }  
}//TrieTree\_Insert\_Key  
分析:当自上而下的查找结束时,存在两种情况.一种情况,树中没有待插入关键字的同义词,此时只要新建一个叶子结点并连到分支结点上即可.另一种情况,有同义词,此时要把同义词的叶子结点与树断开,在断开的部位新建一个下一层的分支结点,再把同义词和新关键字的叶子结点连到新分支结点的下一层.   
9.43   
Status TrieTree\_Delete\_Key(TrieTree &T,StringType key)//在Trie树T中删除字符串key  
{  
  p=T;i=1;  
  while(p&&p->kind==BRANCH&&i<=key[0]) //查找待删除元素  
  {  
    last=p;  
    p=p->bh.ptr[ord(key[i])];  
    i++;  
  }  
  if(p&&p->kind==LEAF&&p->lf.k=key) //找到了待删除元素  
  {  
    last->bh.ptr[ord(key[i-1])]=NULL;  
    free(p);  
    return OK;  
  }  
  else return ERROR; //没找到待删除元素  
}//TrieTree\_Delete\_Key   
9.44   
void Print\_Hash(HashTable H)//按第一个字母顺序输出Hash表中的所有关键字,其中处理冲突采用线性探测开放定址法  
{  
  for(i=1;i<=26;i++)  
    for(j=i;H.elem[j].key;j=(j+1)%hashsize[sizeindex]) //线性探测  
      if(H(H.elem[j].key)==i) printf("%s\n",H.elem[j]);  
}//Print\_Hash   
int H(char \*s)//求Hash函数  
{  
  if(s) return s[0]-96; //求关键字第一个字母的字母序号(小写)  
  else return 0;  
}//H   
9.45   
typedef \*LNode[MAXSIZE] CHashTable; //链地址Hash表类型   
Status Build\_Hash(CHashTable &T,int m)//输入一组关键字,建立Hash表,表长为m,用链地址法处理冲突.  
{  
  if(m<1) return ERROR;  
  T=malloc(m\*sizeof(WORD)); //建立表头指针向量  
  for(i=0;i<m;i++) T[i]=NULL;  
  while((key=Inputkey())!=NULL) //假定Inputkey函数用于从键盘输入关键字  
  {  
    q=(LNode\*)malloc(sizeof(LNode));  
    q->data=key;q->next=NULL;  
    n=H(key);  
    if(!T[n]) T[n]=q; //作为链表的第一个结点  
    else  
    {  
      for(p=T[n];p->next;p=p->next);  
      p->next=q; //插入链表尾部.本算法不考虑排序问题.  
    }  
  }//while  
  return OK;  
}//Build\_Hash   
9.46   
Status Locate\_Hash(HashTable H,int row,int col,KeyType key,int &k)//根据行列值在Hash表表示的稀疏矩阵中确定元素key的位置k  
{  
  h=2\*(100\*(row/10)+col/10); //作者设计的Hash函数  
  while(H.elem[h].key&&!EQ(H.elem[h].key,key))  
    h=(h+1)%20000;  
  if(EQ(H.elem[h].key,key)) k=h;  
  else k=NULL;  
}//Locate\_Hash  
分析:本算法所使用的Hash表长20000,装填因子为50%,Hash函数为行数前两位和列数前两位所组成的四位数再乘以二,用线性探测法处理冲突.当矩阵的元素是随机分布时,查找的时间复杂度为O(1).

第十章 内部排序  
  
10.23   
void Insert\_Sort1(SqList &L)//监视哨设在高下标端的插入排序算法  
{  
  k=L.length;  
  for(i=k-1;i;--i) //从后向前逐个插入排序  
    if(L.r[i].key>L.r[i+1].key)  
    {  
      L.r[k+1].key=L.r[i].key; //监视哨  
      for(j=i+1;L.r[j].key>L.r[i].key;++j)  
        L.r[j-1].key=L.r[j].key; //前移  
      L.r[j-1].key=L.r[k+1].key; //插入  
    }  
}//Insert\_Sort1   
10.24   
void BiInsert\_Sort(SqList &L)//二路插入排序的算法  
{  
  int d[MAXSIZE]; //辅助存储  
  x=L.r .key;d =x;  
  first=1;final=1;  
  for(i=2;i<=L.length;i++)  
  {  
    if(L.r[i].key>=x) //插入前部  
    {  
      for(j=final;d[j]>L.r[i].key;j--)  
        d[j+1]=d[j];  
      d[j+1]=L.r[i].key;  
      final++;  
    }  
    else //插入后部  
    {  
      for(j=first;d[j]<L.r[i].key;j++)  
        d[j-1]=d[j];  
      d[(j-2)%MAXSIZE+1]=L.r[i].key;  
      first=(first-2)%MAXSIZE+1; //这种形式的表达式是为了兼顾first=1的情况  
    }  
  }//for  
  for(i=first,j=1;d[i];i=i%MAXSIZE+1,j++)//将序列复制回去  
    L.r[j].key=d[i];  
}//BiInsert\_Sort   
10.25   
void SLInsert\_Sort(SLList &L)//静态链表的插入排序算法  
{  
  L.r[0].key=0;L.r[0].next=1;  
  L.r[1].next=0; //建初始循环链表  
  for(i=2;i<=L.length;i++) //逐个插入  
  {  
    p=0;x=L.r[i].key;  
    while(L.r[L.r[p].next].key<x&&L.r[p].next)  
      p=L.r[p].next;  
    q=L.r[p].next;  
    L.r[p].next=i;  
    L.r[i].next=q;  
  }//for  
  p=L.r[0].next;  
  for(i=1;i<L.length;i++) //重排记录的位置  
  {  
    while(p<i) p=L.r[p].next;  
    q=L.r[p].next;  
    if(p!=i)  
    {  
      L.r[p]<->L.r[i];  
      L.r[i].next=p;  
    }  
    p=q;  
  }//for  
}//SLInsert\_Sort   
10.26   
void Bubble\_Sort1(int a[ ],int n)//对包含n个元素的数组a进行改进的冒泡排序  
{  
  change=n-1; //change指示上一趟冒泡中最后发生交换的元素  
  while(change)  
  {  
    for(c=0,i=0;i<change;i++)  
      if(a[i]>a[i+1])  
      {  
        a[i]<->a[i+1];  
        c=i+1; //c指示这一趟冒泡中发生交换的元素  
      }  
    change=c;  
  }//while  
}//Bubble\_Sort1   
10.27   
void Bubble\_Sort2(int a[ ],int n)//相邻两趟是反方向起泡的冒泡排序算法  
{  
  low=0;high=n-1; //冒泡的上下界  
  change=1;  
  while(low<high&&change)  
  {  
    change=0;  
    for(i=low;i<high;i++) //从上向下起泡  
      if(a[i]>a[i+1])  
      {  
        a[i]<->a[i+1];  
        change=1;  
      }  
    high--; //修改上界  
    for(i=high;i>low;i--) //从下向上起泡  
      if(a[i]<a[i-1])  
      {  
        a[i]<->a[i-1];  
        change=1;  
      }  
    low++; //修改下界  
  }//while  
}//Bubble\_Sort2   
10.28   
void Bubble\_Sort3(int a[ ],int n)//对上一题的算法进行化简,循环体中只包含一次冒泡  
{  
  int b[ 3 ]; //b[0]为冒泡的下界,b[ 2 ]为上界,b[1]无用  
  d=1;b[0]=0;b[ 2 ]=n-1; //d为冒泡方向的标识,1为向上,-1为向下  
  change=1;  
  while(b[0]<b[ 2 ]&&change)  
  {  
    change=0;  
    for(i=b[1-d];i!=b[1+d];i+=d) //统一的冒泡算法  
      if((a[i]-a[i+d])\*d>0) //注意这个交换条件  
      {  
        a[i]<->a[i+d];  
        change=1;  
      }  
    b[1+d]-=d; //修改边界  
    d\*=-1; //换个方向  
  }//while  
}//Bubble\_Sort3   
10.29   
void OE\_Sort(int a[ ],int n)//奇偶交换排序的算法  
{  
  change=1;  
  while(change)   
  {  
    change=0;  
    for(i=1;i<n-1;i+=2) //对所有奇数进行一趟比较  
      if(a[i]>a[i+1])  
      {  
        a[i]<->a[i+1];  
        change=1;  
      }  
    for(i=0;i<n-1;i+=2) //对所有偶数进行一趟比较  
      if(a[i]>a[i+1])  
      {  
        a[i]<->a[i+1];  
        change=1;  
      }  
  }//while  
}//OE\_Sort   
分析:本算法的结束条件是连续两趟比较无交换发生  
10.30   
typedef struct {  
                     int low;  
                     int high;  
                   } boundary; //子序列的上下界类型   
void QSort\_NotRecurve(int SQList &L)//快速排序的非递归算法  
{  
  low=1;high=L.length;  
  InitStack(S); //S的元素为boundary类型  
  while(low<high&&!StackEmpty(S)) //注意排序结束的条件  
  {  
    if(high-low>2) //如果当前子序列长度大于3且尚未排好序  
    {  
      pivot=Partition(L,low,high); //进行一趟划分  
      if(high-pivot>pivot-low)  
      {  
        Push(S,{pivot+1,high}); //把长的子序列边界入栈  
        high=pivot-1; //短的子序列留待下次排序  
      }  
      else  
      {  
        Push(S,{low,pivot-1});  
        low=pivot+1;  
      }  
    }//if  
    else if(low<high&&high-low<3)//如果当前子序列长度小于3且尚未排好序  
    {  
      Easy\_Sort(L,low,high); //直接进行比较排序  
      low=high; //当前子序列标志为已排好序  
    }  
    else //如果当前子序列已排好序但栈中还有未排序的子序列  
    {  
      Pop(S,a); //从栈中取出一个子序列  
      low=a.low;  
      high=a.high;  
    }  
  }//while  
}//QSort\_NotRecurve   
int Partition(SQList &L,int low,int high)//一趟划分的算法,与书上相同  
{  
  L.r[0]=L.r[low];  
  pivotkey=L.r[low].key;  
  while(low<high)  
  {  
    while(low<high&&L.r[high].key>=pivotkey)  
      high--;  
    L.r[low]=L.r[high];  
    while(low<high&&L.r[low].key<=pivotkey)  
      low++;  
    L.r[high]=L.r[low];  
  }//while  
  L.r[low]=L.r[0];  
  return low;  
}//Partition   
void Easy\_Sort(SQList &L,int low,int high)//对长度小于3的子序列进行比较排序  
{  
  if(high-low==1) //子序列只含两个元素  
    if(L.r[low].key>L.r[high].key) L.r[low]<->L.r[high];  
  else //子序列含有三个元素  
  {  
    if(L.r[low].key>L.r[low+1].key) L.r[low]<->L.r[low+1];  
    if(L.r[low+1].key>L.r[high].key) L.r[low+1]<->L.r[high];  
    if(L.r[low].key>L.r[low+1].key) L.r[low]<->L.r[low+1];  
  }  
}//Easy\_Sort   
10.31   
void Divide(int a[ ],int n)//把数组a中所有值为负的记录调到非负的记录之前  
{  
  low=0;high=n-1;  
  while(low<high)  
  {  
    while(low<high&&a[high]>=0) high--; //以0作为虚拟的枢轴记录  
    a[low]<->a[high];  
    while(low<high&&a[low]<0) low++;  
    a[low]<->a[high];  
  }  
}//Divide   
10.32   
typedef enum {RED,WHITE,BLUE} color; //三种颜色   
void Flag\_Arrange(color a[ ],int n)//把由三种颜色组成的序列重排为按照红,白,蓝的顺序排列  
{  
  i=0;j=0;k=n-1;  
  while(j<=k)  
    switch(a[j])  
    {  
      case RED:  
        a[i]<->a[j];  
        i++;  
        j++;  
        break;  
      case WHITE:  
        j++;  
        break;  
      case BLUE:  
        a[j]<->a[k];  
        k--; //这里没有j++;语句是为了防止交换后a[j]仍为蓝色的情况  
    }  
}//Flag\_Arrange  
分析:这个算法中设立了三个指针.其中,j表示当前元素;i以前的元素全部为红色;k以后的元素全部为蓝色.这样,就可以根据j的颜色,把其交换到序列的前部或者后部.   
10.33   
void LinkedList\_Select\_Sort(LinkedList &L)//单链表上的简单选择排序算法  
{  
  for(p=L;p->next->next;p=p->next)  
  {  
    q=p->next;x=q->data;  
    for(r=q,s=q;r->next;r=r->next) //在q后面寻找元素值最小的结点  
      if(r->next->data<x)  
      {  
        x=r->next->data;  
        s=r;  
      }  
    if(s!=q) //找到了值比q->data更小的最小结点s->next  
    {  
      p->next=s->next;s->next=q;  
      t=q->next;q->next=p->next->next;  
      p->next->next=t;  
    } //交换q和s->next两个结点  
  }//for  
}//LinkedList\_Select\_Sort   
10.34   
void Build\_Heap(Heap &H,int n)//从低下标到高下标逐个插入建堆的算法  
{  
  for(i=2;i<n;i++)  
  { //此时从H.r[1]到H.r[i-1]已经是大顶堆  
    j=i;  
    while(j!=1) //把H.r[i]插入  
    {  
      k=j/2;  
      if(H.r[j].key>H.r[k].key)  
        H.r[j]<->H.r[k];  
      j=k;  
    }  
  }//for  
}//Build\_Heap   
10.35   
void TriHeap\_Sort(Heap &H)//利用三叉树形式的堆进行排序的算法  
{  
  for(i=H.length/3;i>0;i--)  
    Heap\_Adjust(H,i,H.length);  
  for(i=H.length;i>1;i--)  
  {  
    H.r[1]<->H.r[i];  
    Heap\_Adjust(H,1,i-1);  
  }  
}//TriHeap\_Sort   
void Heap\_Adjust(Heap &H,int s,int m)//顺序表H中,H.r[s+1]到H.r[m]已经是堆,把H.r[s]插入并调整成堆  
{  
  rc=H.r[s];  
  for(j=3\*s-1;j<=m;j=3\*j-1)  
  {  
    if(j<m&&H.r[j].key<H.r[j+1].key) j++;  
    if(j<m&&H.r[j].key<H.r[j+1].key) j++;  
    H.r[s]=H.r[j];  
    s=j;  
  }  
  H.r[s]=rc;  
}//Heap\_Adjust  
分析:本算法与课本上的堆排序算法相比,只有两处改动:1.建初始堆时,i的上限从H.length/3开始(为什么?) 2.调整堆的时候,要从结点的三个孩子结点中选择最大的那一个,最左边的孩子的序号的计算公式为j=3\*s-1(为什么?)   
10.36   
void Merge\_Sort(int a[ ],int n)//归并排序的非递归算法  
{  
  for(l=1;l<n;l\*=2) //l为一趟归并段的段长  
    for(i=0;(2\*i-1)\*l<n;i++) //i为本趟的归并段序号  
    {  
      start1=2\*l\*i; //求出待归并的两段的上下界  
      end1=start1+l-1;  
      start2=end1+1;  
      end2=(start2+l-1)>(n-1)?(n-1):(start2+l-1);//注意end2可能超出边界  
      Merge(a,start1,end1,start2,end2); //归并  
    }  
}//Merge\_Sort   
void Merge(int a[ ],int s1,int e1,int s2,int e2)//将有序子序列a[s1]到a[e1]和a[s2]到a[e2]归并为有序序列a[s1]到a[e2]  
{  
  int b[MAXSIZE]; //设立辅助存储数组b  
  for(i=s1,j=s2,k=s1;i<=e1&&j<=e2;k++)  
  {  
    if(a[i]<a[j]) b[k]=a[i++];  
    else b[k]=a[j++];  
  }  
  while(i<=e1) b[k++]=a[i++];  
  while(j<=e2) b[k++]=a[j++]; //归并到b中  
  for(i=s1;i<=e2;i++) //复制回去  
    a[i]=b[i];  
}//Merge   
10.37   
void LinkedList\_Merge\_Sort1(LinkedList &L)//链表结构上的归并排序非递归算法  
{  
  for(l=1;l<L.length;l\*=2) //l为一趟归并段的段长  
    for(p=L->next,e2=p;p->next;p=e2)  
    {  
      for(i=1,q=p;i<=l&&q->next;i++,q=q->next);  
      e1=q;  
      for(i=1;i<=l&&q->next;i++,q=q->next);  
      e2=q; //求出两个待归并子序列的尾指针  
      if(e1!=e2) LinkedList\_Merge(L,p,e1,e2); //归并  
    }  
}//LinkedList\_Merge\_Sort1   
void LinkedList\_Merge(LinkedList &L,LNode \*p,LNode \*e1,LNode \*e2)//对链表上的子序列进行归并,第一个子序列是从p->next到e1,第二个是从e1->next到e2  
{  
  q=p->next;r=e1->next; //q和r为两个子序列的起始位置  
  while(q!=e1->next&&r!=e2->next)  
  {  
    if(q->data<r->data) //选择关键字较小的那个结点接在p的后面  
    {  
      p->next=q;p=q;  
      q=q->next;  
    }  
    else  
    {  
      p->next=r;p=r;  
      r=r->next;  
    }  
  }//while  
  while(q!=e1->next) //接上剩余部分  
  {  
    p->next=q;p=q;  
    q=q->next;  
  }  
  while(r!=e2->next)  
  {  
    p->next=r;p=r;  
    r=r->next;  
  }  
}//LinkedList\_Merge   
10.38   
void LinkedList\_Merge\_Sort2(LinkedList &L)//初始归并段为最大有序子序列的归并排序,采用链表存储结构  
{  
  LNode \*end[MAXSIZE]; //设立一个数组来存储各有序子序列的尾指针  
  for(p=L->next->next,i=0;p;p=p->next) //求各有序子序列的尾指针  
    if(!p->next||p->data>p->next->data) end[i++]=p;  
  while(end[0]->next) //当不止一个子序列时进行两两归并  
  {  
    j=0;k=0; //j:当前子序列尾指针存储位置;k:归并后的子序列尾指针存储位置  
    for(p=L->next,e2=p;p->next;p=e2) //两两归并所有子序列  
    {  
      e1=end[j];e2=end[j+1]; //确定两个子序列  
      if(e1->next) LinkedList\_Merge(L,p,e1,e2); //归并  
      end[k++]=e2; //用新序列的尾指针取代原来的尾指针  
      j+=2; //转到后面两个子序列  
    }  
  }//while  
}//LinkedList\_Merge\_Sort2   
void LinkedList\_Merge(LinkedList &L,LNode \*p,LNode \*e1,LNode \*e2)//对链表上的子序列进行归并,第一个子序列是从p->next到e1,第二个是从e1->next到e2  
{  
  q=p->next;r=e1->next;  
  while(q!=e1->next&&r!=e2->next)  
  {  
    if(q->data<r->data)  
    {  
      p->next=q;p=q;  
      q=q->next;  
    }  
    else  
    {  
      p->next=r;p=r;  
      r=r->next;  
    }  
  }//while  
  while(q!=e1->next)  
  {  
    p->next=q;p=q;  
    q=q->next;  
  }  
  while(r!=e2->next)  
  {  
    p->next=r;p=r;  
    r=r->next;  
  }  
}//LinkedList\_Merge,与上一题完全相同   
10.39   
void SL\_Merge(int a[ ],int l1,int l2)//把长度分别为l1,l2且l1^2<(l1+l2)的两个有序子序列归并为有序序列  
{  
  start1=0;start2=l1; //分别表示序列1和序列2的剩余未归并部分的起始位置  
  for(i=0;i<l1;i++) //插入第i个元素  
  {  
    for(j=start2;j<l1+l2&&a[j]<a[start1+i];j++); //寻找插入位置  
    k=j-start2; //k为要向右循环移动的位数  
    RSh(a,start1,j-1,k);//将a[start1]到a[j-1]之间的子序列循环右移k位  
    start1+=k+1;  
    start2=j; //修改两序列尚未归并部分的起始位置  
  }  
}//SL\_Merge   
void RSh(int a[ ],int start,int end,int k)//将a[start]到a[end]之间的子序列循环右移k位,算法原理参见5.18  
{  
  len=end-start+1;  
  for(i=1;i<=k;i++)  
    if(len%i==0&&k%i==0) p=i; //求len和k的最大公约数p  
  for(i=0;i<p;i++) //对p个循环链分别进行右移  
  {  
    j=start+i;l=start+(i+k)%len;temp=a[j];  
    while(l!=start+i)  
    {  
      a[j]=temp;  
      temp=a[l];  
      a[l]=a[j];  
      j=l;l=start+(j-start+k)%len; //依次向右移  
    }  
    a[start+i]=temp;  
  }//for  
}//RSh   
10.40   
书后给出的解题思路在表述上存在问题,无法理解.比如说,"把第一个序列划分为两个子序列,使其中的第一个子序列含有s1个记录,0<=s1<s,第二个子序列有s个记录."可是题目中并没有说明,第一个序列的长度<2s.请会做的朋友提供解法.   
10.41   
void Hash\_Sort(int a[ ])//对1000个关键字为四位整数的记录进行排序  
{  
  int b[10000];  
  for(i=0;i<1000;i++) //直接按关键字散列  
  {  
    for(j=a[i];b[j];j=(j+1)%10000);  
    b[j]=a[i];  
  }  
  for(i=0,j=0;i<1000;j++) //将散列收回a中  
    if(b[j])  
    {  
      for(x=b[j],k=j;b[k];k=(k+1)%10000)  
        if(b[k]==x)  
        {  
          a[i++]=x;  
          b[k]=0;  
        }  
    }//if  
}//Hash\_Sort   
10.42   
typedef struct {  
                     int gt; //大于该记录的个数  
                     int lt; //小于该记录的个数  
                   } place; //整个序列中比某个关键字大或小的记录个数   
int Get\_Mid(int a[ ],int n)//求一个序列的中值记录的位置  
{  
  place b[MAXSIZE];  
  for(i=0;i<n;i++) //对每一个元素统计比它大和比它小的元素个数gt和lt  
    for(j=0;j<n;j++)  
    {  
      if(a[j]>a[i]) b[i].gt++;  
      else if(a[j]<a[i]) b[i].lt++;  
    }  
  mid=0;  
  min\_dif=abs(b[0].gt-b[0].lt);  
  for(i=0;i<n;i++) //找出gt值与lt值最接近的元素,即为中值记录  
    if(abs(b[i].gt-b[i].lt)<min\_dif) mid=i;  
  return mid;  
}//Get\_Mid   
10.43   
void Count\_Sort(int a[ ],int n)//计数排序算法  
{  
  int c[MAXSIZE];  
  for(i=0;i<n;i++) //对每一个元素  
  {  
    for(j=0,count=0;j<n;j++) //统计关键字比它小的元素个数  
      if(a[j]<a[i]) count++:  
    c[i]=count;  
  }  
  for(i=0;i<n;i++) //依次求出关键字最小,第二小,...,最大的记录  
  {  
    min=0;  
    for(j=0;j<n;j++)  
      if(c[j]<c[min]) min=j; //求出最小记录的下标min  
    a[i]<->a[min]; //与第i个记录交换  
    c[min]=INFINITY; //修改该记录的c值为无穷大以便下一次选取  
  }  
}//Count\_Sort   
10.44   
void Enum\_Sort(int a[ ],int n)//对关键字只能取v到w之间任意整数的序列进行排序  
{  
  int number[w+1],pos[w+1];  
  for(i=0;i<n;i++) number[a[i]]++; //计数  
  for(pos[0]=0,i=1;i<n;i++)  
    pos[i]=pos[i-1]+num[i]; //pos数组可以把关键字的值映射为元素在排好的序列中的位置  
  for(i=0;i<n;i++) //构造有序数组c  
    c[pos[a[i]]++]=a[i];  
  for(i=0;i<n;i++)  
    a[i]=c[i];  
}//Enum\_Sort  
分析:本算法参考了第五章三元组稀疏矩阵转置的算法思想,其中的pos数组和那里的cpot数组起的是相类似的作用.   
10.45   
typedef enum {0,1,2,3,4,5,6,7,8,9} digit; //个位数类型  
typedef digit[3] num; //3位自然数类型,假设低位存储在低下标,高位存储在高下标   
void Enum\_Radix\_Sort(num a[ ],int n)//利用计数实现基数排序,其中关键字为3位自然数,共有n个自然数  
{  
  int number ,pos ;  
  num c[MAXSIZE];  
  for(j=0;j<3;j++) //依次对个位,十位和百位排序  
  {  
    for(i=0;i<n;i++) number[a[i][j]]++; //计数  
    for(pos[0]=0,i=1;i<n;i++)  
      pos[i]=pos[i-1]+num[i]; //把关键字的值映射为元素在排好的序列中的位置  
    for(i=0;i<n;i++) //构造有序数组c  
      c[pos[a[i][j]]++]=a[i];  
    for(i=0;i<n;i++)  
      a[i]=c[i];  
  }//for  
}//Enum\_Radix\_Sort  
分析:计数排序是一种稳定的排序方法.正因为如此,它才能够被用来实现基数排序.   
10.46   
typedef struct {  
                     int key;  
                     int pos;  
                   } Shadow; //影子序列的记录类型   
void Shadow\_Sort(Rectype b[ ],Rectype &a[ ],int n)//对元素很大的记录序列b进行排序,结果放入a中,不移动元素  
{  
  Shadow d[MAXSIZE];  
  for(i=0;i<n;i++) //生成影子序列  
  {  
    d[i].key=b[i].key;  
    d[i].pos=i;  
  }  
  for(i=n-1,change=1;i>1&&change;i--) //对影子序列执行冒泡排序  
  {  
    change=0;  
    for(j=0;j<i;j++)  
      if(d[j].key>d[j+1].key)  
      {  
        d[j]<->d[j+1];  
        change=1;  
      }  
  }//for  
  for(i=0;i<n;i++) //按照影子序列里记录的原来位置复制原序列  
    a[i]=b[d[i].pos];  
}//Shadow\_Sort