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 kaoyan.com 计算机版版主一具.以下网友:siice,龙抬头,iamkent,zames,birdthinking 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: yi-ju@263.net

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) f=1;
 else
 {
  for(i=0;i<=k-2;i++) temp[i]=0;
  temp[k-1]=1; //初始化
  for(i=k;i<=m;i++) //求出序列第 k 至第 m 个元素的值
  {
   sum=0;
   for(j=i-k;j<i;j++) sum+=temp[j];
   temp[i]=sum;
  }
  f=temp[m];
 }
 return OK;
}//fib
```

分析:通过保存已经计算出来的结果,此方法的时间复杂度仅为 O(m^2).如果采用递归编程(大多数人都会首先想到递归方法),则时间复杂度将高达 O(k^m).

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;
 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.totalscore+=result[i].score;
    if(result[i].gender==0) score.malescore+=result[i].score;
    else score.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 ad;
 float *p=a;
 printf("Input number of terms:");
 scanf("%d",&n);
 printf("Input the %d coefficients from a0 to a%d:\n",n,n);
 for(i=0;i<=n;i++) scanf("%f",p++);
 printf("Input value of x:");
 scanf("%f",&x);
 p=a;xp=1;sum=0; //xp 用于存放 x 的 i 次方
 for(i=0;i<=n;i++)
 {
  sum+=xp*(*p++);
  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,值为正,表示 A>B;值为负,表示 A<B;值为零,表示 A=B
{
 for(i=1;A.elem[i]||B.elem[i];i++)
  if(A.elem[i]!=B.elem[i]) return A.elem[i]-B.elem[i];
 return 0;
}//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));
 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
 C=pc;
}//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
  }
 }//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) 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())!='&')
  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
```


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+k)%n;temp=A[i];
  while(l!=i)
  {
   A[j]=temp;
   temp=A[l];
   A[l]=A[j];
   j=l;l=(j+k)%n;
  }// 循环右移一步
  A[i]=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
本题难度极大,故仅探讨一下思路.这一题的难点在于,在多项式的元数 m 未知的情况下,如何按照降幂次序输出各项.可以考虑采取类似于层序遍历的思想,从最高次的项开始,依次对其每一元的次数减一,入一个队列.附设访问标志 visited 以避免重复.

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=cpot[i];s<cpot[i+1]&&A.data[s].j!=j;s++);//注意查找范围
 if(A.data[s].i==i&&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
{
 if(A->tag&&A->ptr.tp) //当 A 不为原子且表尾非空时才需逆转
 {
  D=C->ptr.hp;
  A->ptr.tp->ptr.hp=A->ptr.hp;A->ptr.hp=D; //交换表头和表尾
  GList_Reverse(A->ptr.hp);
  GList_Reverse(A->ptr.tp); //逆转表头和表尾
 }
}//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("(");
  GList_PrintList(A->ptr.hp);
  if(A->ptr.tp)
  {
   printf(",");
   GList_PrintList(A->ptr.tp);
  } //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->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->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->tag)
  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->ltag||!q->rtag)
  {
   if(!q->ltag) 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&&!p->rtag) return INFEASIBLE; //无法插入
 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=q->lchild;
  q->lchild=s; //找到子树中的最左结点,并修改其前驱指向 s
  q=x;
  while(q->rchild) q=q->rchild;
  q->rchild=p; //找到子树中的最右结点,并修改其前驱指向 p
 }
 else //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->rchild) q=q->rchild;
  q->rchild=s; //找到子树中的最右结点,并修改其前驱指向 s
  q=x;
  while(q->lchild) 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
 }
 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 Bulid_CSTree_PTree(PTree T)//由树 T 的双亲表构造其孩子兄弟链表
{
 NodeMsg Treed;
 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
}//Bulid_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 * ptrd; //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
   }//for
  }//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).