严蔚敏版数据结构所有算法代码

----------线性数据结构----------

2012 年 7 月

```
//线性表、链表
//栈、队列
//数组、广义表
//串
----------线性表----------
typedef struct
{
    char name[20];//注意如果应用指针的形式
                  //在初始化每个结点时一定要先为结点中的每个变量开辟内存空间
    char sex;
    char addr[100];
    unsigned int age;
    char phonenum[20];
}node;//结点描述
typedef struct
{
    node *p;
    int length;//当前顺序表长度
    int listsize;//当前分配的线性表长度
}list;//线性表描述
list L;//定义一个线性表
int initlist(list &l)//构造一个空的线性表
{
    l.p=(node*)malloc(LIST_INIT_SIZE*sizeof(node));
    if(!(l.p))
        exit(1);
    l.length=0;
    l.listsize=LIST_INIT_SIZE;
    return true;
}
void destroylist(list &l)//销毁线性表操作
{
    if(l.p!=NULL)
    {
        free(l.p);
        printf("销毁成功!\n");
    }
    else
```

```
        printf("线性表不存在！\n");
}
int clearlist(list &l)//将线性表置空操作
{
    if(l.p==NULL)
    {
        printf("线性表不存在！\n");
        return false;
    }
    else
        {
            free(l.p);
            l.p=(node*)malloc(l.listsize*sizeof(node));
            l.length=0;
        }
    return true;
}
int listempty(list &l)//判断线性表是否为空表
{
    if(l.p==NULL)
        return true;
    else
        return false;
}
int getelem(list &l,int i,node &e)//用 e 返回表中第 i 个数据元素
{
    if(l.p==NULL)
        return false;
    else
        e=l.p[i-1];
         return true;
}
int priorelem(list &l,int i,node &pre_e)//得到第 i 个元素的前驱元素
{
    if(i==0||l.p==NULL)
        return false;
    else
        pre_e=l.p[i-1];
        return true;
}
int nextelem(list &l,int i,node &next_e)//得到表中第 i 个元素的后继元素
{
    if(i>=l.length||l.p==NULL)
        return false;
    else
```

```
        next_e=l.p[i+1];
    return true;
}
int insertlist(list &l,int i,node &e)//将元素 e 插入到表 l 中第 i 个元素的后面
{
    node *q,*k;
    if(i<1||i>l.length+1)
        return false;
    if(l.length>=l.listsize)
    {
        l.p=(node *)realloc(l.p,(l.listsize+LISTINCREMENT)*sizeof(node));
        if(!l.p)
            exit(1);
        l.listsize+=LISTINCREMENT;
    }
    k=&l.p[i-1];
    for(q=&l.p[l.length-1];q>k;q--)
        *(q+1)=*q;
    *k=e;
    l.length++;
    return true;
}
int deletelist(list &l,int i,node &e)//删除表中第 i 个元素并用 e 返回其值
{
    node *q;
    int j=i-1;
    if(i<1||i>l.length)
        return false;
    e=l.p[i-1];
    for(q=&l.p[i-1];j<l.length-1;j++)
        *q=*(++q);
    l.length--;
    return true;
}
void mergerlist(list la,list lb,list &lc)//归并两个按非递减排列的线性表
{
    int la_len,lb_len,i=1,j=1,k=0;
    node ai,bj;
    la_len=la.length;
    lb_len=lb.length;
    while(i<=la_len&&j<=lb_len)
    {
        getelem(la,i,ai);
        getelem(lb,j,bj);
        if(ai.a<=bj.a)
```

```
        {
            insertlist(lc,++k,ai);
            i++;
        }
        else
        {
            insertlist(lc,++k,bj);
            j++;
        }
    }
    while(i<=la_len)
    {
        getelem(la,i,ai);
        insertlist(lc,++k,ai);
        i++;
    }
    while(j<=lb_len)
    {
        getelem(lb,j,bj);
        insertlist(lc,++k,bj);
        j++;
    }
}
int ListAscendingOrder(list &l)//按结点中某一元素的比较升序排列线性表中的结点
{
    node e;
    int i,j;
    if(l.p==NULL||l.length==1)
        return ERROR;
    for(i=0;i<l.length-1;i++)
        for(j=i+1;j<l.length;j++)
            if(l.p[i].num>=l.p[j].num)
            {
                e=l.p[i];
                l.p[i]=l.p[j];
                l.p[j]=e;
            }
            return OK;
}//省略降序排列
void MergerList(list la,list lb,list &lc)//将两线性表升序排列后再归并
{
    node *q,*k,e1;
    int i=0,j=0,m=0,n;
    ListAscendingOrder(la);
    ListAscendingOrder(lb);
```

```
    printf("表 a 升序排列后为: \n");
    for(i=0;i<la.length;i++)
        printf("%d ",la.p[i].num);
    printf("\n");
    printf("表 b 升序排列后为: \n");
    for(i=0;i<lb.length;i++)
        printf("%d ",lb.p[i].num);
    printf("\n");
    i=0;
    while(i<la.length&&j<lb.length)
    {
        if(la.p[i].num<=lb.p[j].num)
        {
            e1=la.p[i];
            i++;
        }
        else
        {
            e1=lb.p[j];
            j++;
        }
        if(e1.num!=lc.p[lc.length-1].num)
            InsertList(lc,++m,e1);
    }
    if(i<la.length)
        while(i<la.length)
        {
            if(la.p[i].num!=lc.p[lc.length-1].num)
                InsertList(lc,++m,la.p[i]);
            i++;
        }
    if(j<lb.length)
        while(j<lb.length)
        {
            if(lb.p[j].num!=lc.p[lc.length-1].num)
                InsertList(lc,++m,lb.p[j]);
            j++;
        }
        printf("按升序排列再归并两表为: \n");
        for(n=0;n<lc.length;n++)
            printf("%d ",lc.p[n].num);
        printf("\n");
}
----------------------链表-----------------------------
typedef struct
```

```
{
    int num;
}node;
typedef struct LIST
{
    node data;
    struct LIST *next;
}list,*slist;
int CreatList(slist &head)//此处应为只针对的引用
{
    head=(list *)malloc(sizeof(list));
    if(!head)
    return ERROR;
    head->next=NULL;
    return OK;
}
void InvertedList(slist &head1,slist &head2)
{//构造新表逆置单链表函数
    list *p,*q;
    p=head1->next;
    q=p->next;
    if(p==NULL)
        printf("链表为空无法实现逆置操作\n");
    else
    {
        while(q!=NULL)
        {
            p->next=head2->next;
             head2->next=p;
            p=q;
            q=q->next;
        }
        p->next=head2->next;
        head2->next=p;
        printf("逆置成功！?\n");
    }
}
void InsertList(slist &head,node &e)//此处应为指针的引用
{//而不应该是 list *head
    list *p,*q;
    p=(list *)malloc(sizeof(list));
    q=head;
    while(q->next!=NULL)
        q=q->next;
    p->next=q->next;
```

```
    q->next=p;
    p->data=e;
}
void InvertedList(sqlist &head)
{//-------不构造新表逆置单链表函数---------//
    list *p,*q,*k;
    p=head->next;
    q=p->next;
    k=q->next;
    p->next=NULL;
    while(k!=NULL)
    {
        q->next=p;
        p=q;
        q=k;
        k=k->next;
    }
    q->next=p;
    head->next=q;
}
//----交换链表中第 i 个和第 j 个结点，函数实现如下——//
int SwapListNode(sqlist &head,int i,int j)
{
    int m,n,m1,n1,sum=0;
    list *p,*q,*k,*c,*d,*ba;
    ba=head->next;
    while(ba!=NULL)
    {
        sum++;
        ba=ba->next;
    }
    if(i==j||i>sum||j>sum||i<1||j<1)
    {
        printf("所要交换的两个结点有误！\n");
        return  ERROR;
    }
    if(i<j)
    { m=i; n=j;}
    else
    {  m=j;n=i;}
    p=head;q=head;
    for(m1=1;m1<=m;m1++)
        p=p->next;
    for(n1=1;n1<=n;n1++)
        q=q->next;
```

```
    if(p->next==q)
    {//如果结点相邻
        k=head;
        while(k->next!=p)
            k=k->next;
        //相邻两结点的交换
        p->next=q->next;
        q->next=p;
        k->next=q;
    }
    else
    {//如果结点不相邻
        k=head;c=head;
        while(k->next!=p)
            k=k->next;
        while(c->next!=q)
            c=c->next;
        d=p->next;
        //不相邻两结点之间的交换
        p->next=q->next;
        c->next=p;
        k->next=q;
        q->next=d;
    }
    return OK;
}
//-----将链表中结点按结点中某一项大小升序排列，函数实现如下-----//
int AscendingList(sqlist &head)
{
    int m,n,sum=0,i,j;
    list *p,*q,*k;
    k=head->next;
    while(k!=NULL)
    {
        sum++;
        k=k->next;
    }
    for(i=1;i<sum;i++)
        for(j=i+1;j<=sum;j++)
        {
            p=head->next;
            m=1;
            while(m!=i)
            {
                m++;
```

```
            p=p->next;
        }
        q=head->next;
        n=1;
        while(n!=j)
        {
            n++;
            q=q->next;
        }
        if(p->data.exp>q->data.exp)//如果按 exp 降序排列，则应将>改为<;
            SwapListNode(head,i,j);
    }
    return OK;
}
//-----将两链表合并为一个链表------//
int AddList(sqlist &head1,sqlist &head2,sqlist &head3)
{//已将表 head1 和表 head2 按某一项升序排列过
    sqlist p,q;
    node e;
    p=head1->next;
    q=head2->next;
    while(p!=NULL&&q!=NULL)
    {
        if(p->data.exp<q->data.exp)
        {
            InsertList(head3,p->data);
            p=p->next;
        }
        else
        if(p->data.exp>q->data.exp)
        {
            InsertList(head3,q->data);
            q=q->next;
        }
        else
        if(p->data.exp==q->data.exp)
        {
            e.coefficient=p->data.coefficient+q->data.coefficient;
            e.exp=p->data.exp;//e.exp=q->data.exp;
            InsertList(head3,e);
            p=p->next;
            q=q->next;
        }
    }
    if(p!=NULL)
```

```
        while(p!=NULL)
        {
            InsertList(head3,p->data);
            p=p->next;
        }//如果 p 中有剩余，则直接将 p 中剩余元素插入 head3 中
    if(q!=NULL)
        while(q!=NULL)
        {
            InsertList(head3,q->data);
            q=q->next;
        }//如果 q 中有剩余，则直接将 q 中的剩余元素插入 head3 中
        return 0;
}
----------------------栈-----------------------------
//---------利用栈结构实现数制之间的转换------书3.2.1//
typedef struct
{
    int num;
}node;
typedef struct
{
    node *base;
    node *top;
    int stacksize;
}stack;//顺序栈结构定义
int CreatStack(stack &stackll)
{
    stackll.base=(node *)malloc(INITSTACKSIZE*sizeof(node));
    if(!stackll.base)
        exit(OVERFLOW);
    stackll.top=stackll.base;
    stackll.stacksize=INITSTACKSIZE;
    return OK;
}
void push(stack &s,node e)
{//进栈操作
    if(s.top-s.base>=s.stacksize)
    {   s.base=(node *)realloc(s.base,(s.stacksize+INCRESTACKMENT)*sizeof(node));
    if(!s.base)
        exit(OVERFLOW);
    s.top=s.base+s.stacksize;//可以不写此语句;
    s.stacksize+=INCRESTACKMENT;
    }
    *(s.top++)=e;//*s.top++=e;
}
```

```
void pop(stack &s,node &e)
{//出栈操作
    if(s.top==s.base||s.base==NULL)
        printf("信息有误！\n");
    else
        e=*--s.top;
}
//-------取栈顶元素函数------//
void gettop(stack &s,node &e)
{
    if(s.base==s.top)
        printf("栈为空，无法取得栈顶元素！\n");
    else
    {
        e=*(s.top-1);
    }
}
//-----栈的应用：括号匹配的检验------书3.2.2//
//省略了大部分上述已有代码//
int zypd(char c)
{//判断是否为左括号字符
    if(c=='['||c=='{'||c=='(')
        return OK;
    else
        return ERROR;
}
int main(void)
{
    stack s;
    node e1,e2,e3;
    char st[INITSTACKSIZE];
    int i=0,j;
    CreatStack(s);
    printf("请输入括号字符，以'#'做结束符：");
    scanf("%c",&st[i]);
    while(st[i]!='#')
    {
        i++;
        scanf("%c",&st[i]);
    }
    if(!zypd(st[0]))
        printf("输入字符不合法！\n");
    else
    {
        for(j=0;j<i;j++)
```

```
        {
            if(zypd(st[j]))
            {//如果是左括号则将此字符压入栈中
                e1.c=st[j];
                push(s,e1);
            }
            else
            {//如果当前st[j]元素不是左括号
                //则取出栈顶元素，比较栈顶元素与当前st[j]元素是否项匹配
                //如果匹配，则栈顶元素出栈
                gettop(s,e2);
                if(e2.c=='['&&st[j]==']'||e2.c=='{'&&st[j]=='}'||e2.c=='('&&st[j]==')')
                    pop(s,e3);
                else
                {
                    printf("括号验证失败！\n");
                    break;
                }
            }
        }
        if(s.top==s.base)//当循环结束时，如果栈为空栈说明输入的括号合法
            printf("括号验证成功！\n");
    }
        getchar();
        system("pause");
        return 0;
}
//----------链栈描述---------//
typedef struct Node
{
    int num;
    struct Node *next;
}node;
typedef struct
{
    Node *top;
    Node *base;
}stack;
void InitStack(stack &s)
{
    s.base=(Node *)malloc(sizeof(node));
    if(!s.base)
        exit(1);
    else
    {
```

```
        s.base->next=NULL;
        s.top=s.base;
    }
}
void InsertStack(stack &s,node e)
{
    node *p;
    p=(node *)malloc(sizeof(node));
    if(!p)
        exit(1);
    else
    {
        *p=e;
        p->next=s.top;
        s.top=p;
    }
}
void DeleteStack(stack &s,node &e)
{
    node *p;
    if(s.top==s.base)
        printf("栈为空！\n");
    else
    {
        p=s.top;
        s.top=s.top->next;
        e=*p;
        free(p);
    }
}
--------------------队列----------------------
//------链队列的描述及操作-------//
typedef struct Node
{
    int a;
    struct Node *next;
}Qnode,*QueuePtr;
typedef struct
{
    QueuePtr front;
    QueuePtr rear;
}LinkQueue;
void InitQueue(LinkQueue &Q)
{
    Q.front=(Qnode *)malloc(sizeof(Qnode));
```

```
    if(!Q.front)
        exit(1);
    Q.rear=Q.front;
    Q.front->next=NULL;
}
void InsertQueue(LinkQueue &Q,Qnode e)
{
    QueuePtr p;
    p=(Qnode *)malloc(sizeof(Qnode));
    if(!p)
        exit(1);
    *p=e;
    p->next=NULL;
    Q.rear->next=p;
    Q.rear=p;
}
void DeleteQueue(LinkQueue &Q,Qnode &e)
{
    Qnode *p;
    if(Q.front==Q.rear)
        printf("队列为空！\n");
    else
    {
        p=Q.front->next;
        e=*p;
        Q.front->next=p->next;
        if(p==Q.rear)
            Q.rear=Q.front;
        free(p);
    }
}
//-------------循环队列---------------//
typedef struct node
{
    int data;
    struct node *next;
}node;
typedef struct queue
{
    node *base;
    int front;
    int rear;
}Queue;
int tag;
void InitQueue(Queue &Q)
```

```
{
    Q.base=(node *)malloc(MAX*sizeof(node));
    if(!Q.base)
        exit(1);
    Q.front=Q.rear=0;
    tag=0;
}
void InsertQueue(Queue &Q,node e)
{
    if(tag==1&&Q.front==Q.rear)
        printf("循环队列已满！\n");
    else
    {
        Q.base[Q.rear]=e;
        Q.rear=(Q.rear+1)%MAX;
        if(Q.rear==Q.front)
            tag=1;
    }
}
void DeleteQueue(Queue &Q,node &e)
{
    if(tag==0&&Q.front==Q.rear)
        printf("队列为空！\n");
    else
    {
        e=Q.base[Q.front];
        Q.front=(Q.front+1)%MAX;
        if(Q.front==Q.rear)
            tag=0;
    }
}
int EmptyQueue(Queue &Q)
{
    if(Q.front==Q.rear&&tag==0)
        return 1;
    else
        return 0;
}

-------------------------串-------------------------------------
//-----------串：堆分配存储形式的一些操作-----------------//
typedef struct string
{
    char *ch;
    int length;
```


购课认准惊呼网

```
}sstring;
void CreatString(sstring &T)
{
    T.ch=(char*)malloc(sizeof(char));
    T.length=0;
}
void StringAssign(sstring &T,char *s)
{//将串s的值赋值给串T
    if(T.ch)
        free(T.ch);
    T.ch=(char*)malloc(strlen(s)*sizeof(char));//或者T.ch=(char*)malloc(sizeof(char));
    //动态开辟空间不同于静态内存开辟之处
    if(!T.ch)
    {
        printf("ERROR");
        exit(1);
    }
    strcpy(T.ch,s);
    T.length=strlen(s);
}
void ClearString(sstring &T)
{
    if(T.ch)
        free(T.ch);
    T.length=0;
}
void ConcatString(sstring &T,sstring s1,sstring s2)
{//串连接
    if(T.ch)
        free(T.ch);
    T.ch=(char*)malloc((strlen(s1.ch)+strlen(s2.ch))*sizeof(char));
    if(!T.ch)
    {
        printf("ERROR\n");
        exit(1);
    }
    strcpy(T.ch,s1.ch);
    strcat(T.ch,s2.ch);
    T.length=strlen(s1.ch)+strlen(s2.ch);
}
void SubString(sstring &sub,sstring s,int pos,int len)
{//取子串操作，取串s中位置从pos至len处的子串于sub中
    int i,j=0;
    if(sub.ch)
        free(sub.ch);
```

```
    sub.ch=(char *)malloc((len-pos+1+1)*sizeof(char));
    if(!sub.ch)
    {
        printf("ERROR\n");
        exit(1);
    }
    for(i=pos-1;i<len;i++)
        sub.ch[j++]=s.ch[i];
    sub.ch[j]='\0';
    sub.length=strlen(sub.ch);
}
int CountString(sstring s1,sstring s2)
{//判断子串s2在母串s1中出现的次数
    int i,j,k,count=0;
    if(s1.length==0||s2.length==0||s2.length>s1.length)
    {
        printf("ERROR\n");
        return 0;
    }
    else
    {
        for(i=0;i<s1.length;i++)
        {
            k=1;
            for(j=0;j<s2.length;j++)
            {
                if(s2.ch[j]!=s1.ch[i+j])
                {
                    k=0;
                    break;
                }
            }
            if(k)
                count++;
        }
    }
    return count;
}
void Deletestring(sstring &s,int pos,int len)
{//删除s串中位置从pos到len处的元素
    int i,j,k;
    if(s.length==0)
        printf("ERROR\n");
    else
    {
```

```
        for(i=pos-1,j=len;j<s.length;i++,j++)
            s.ch[i]=s.ch[j];
        s.ch[i]='\0';
        s.length-=(len-pos)+1;
    }
}
void DeleteSub(sstring &s1,sstring s2)
{//删除母串s1中的子串s2
    int i,j,k,tag=0;
    for(i=0;i<s1.length;i++)
    {
        k=1;
        if(tag)
        i--;
        for(j=0;j<s2.length;j++)
            if(s2.ch[j]!=s1.ch[i+j])
            {
                k=0;
                break;
            }
            if(k)
            {
                Deletestring(s1,i+1,i+s2.length);
                tag=1;
            }
    }

}
----------------KMP算法----------------
int index_kmp(string T,string S,int pos)
{
    int i=pos,j=1;
    while(i<=S.length&&j<=T.length)
    {
        if(S.ch[i]==T.ch[j])
        {
            i++;
            j++;
        }
        else
            j=next[j+1];
    }
    if(j>T.length)
        return i-T.length;
    else
```

```
        return 0;
}
void get_next(string T)
{
    int i=1,j=0;
    next[1]=0;
    while(i<=T.length)
    {
        if(j-1==0||T.ch[i]==T.ch[j])
        {
            i++;
            j++;
            if(T.ch[i]!=T.ch[j])
                next[i]=j;
            else
                next[i]=next[j];
        }
        else
            j=next[j];
    }
}
```

————————————————————数组————————————————————

----------矩阵转置的经典算法---------

```
for(i=0;i<row;i++)
        for(j=0;j<col;j++)
            b[j][i]=a[i][j];
```

时间复杂度为O（row*col),每个元素都要存储，相对于稀疏矩阵来说比较浪费存储空间。

----------矩阵转置---利用三元组实现---------

```
#define MAX 12500//假设一个稀疏矩阵最多有12500个非零元
typedef struct
{
    int i,j;//i,j用于存储矩阵中元素的行、列下标
    int num;//num为矩阵中非零元的值
}Triple;//定义一个三元组
typedef struct
{
    Triple data[MAX+1];
    int mu,nu,tu;//mu,nu非别表示一个稀疏矩阵的行数和列数
    //tu表示该稀疏矩阵的非零元个数
}TSMatrix;
//矩阵转置，核心算法如下：
    t.mu=m.nu;t.nu=m.mu;t.tu=m.tu;
    for(i=0;i<m.nu;i++)
        for(j=0;j<m.tu;j++)//按列遍历三元组
```

```
        {
            if(m.data[j].j==i)//按列升序存入数组
            {
                t.data[p].i=m.data[j].j;
                t.data[p].j=m.data[j].i;
                t.data[p].num=m.data[j].num;
                p++;
            }
        }
```

该算法时间复杂度为O(nu*tu),即与该矩阵的列数和非零元个数有关，当tu=mu*nu时，时间复杂度为O(nu2*tu),此时的时间复杂度比一般算法的时间复杂度还要大，因此此算法适用于tu<<mu*nu的情况，此算法相比一般算法节省了存储空间。

----------快速转置算法<改进了时间复杂度>----------

```
t.tu=m.tu;t.mu=m.nu;t.nu=m.mu;
    for(i=1;i<=m.nu;i++)
        num[i]=0;//先使每列上元素的个数为0
    for(i=0;i<m.tu;i++)
        num[m.data[i].j]++;//遍历三元组求得每列上元素的个数
    for(i=2;i<=m.nu;i++)
        cpot[i]=cpot[i-1]+num[i-1];//求得每列上第一个元素在转置矩阵三元组中的存储序号
    for(i=0;i<m.tu;i++)
    {
        j=m.data[i].j;
        q=cpot[j];
        t.data[q].i=m.data[i].j;
        t.data[q].j=m.data[i].i;
        t.data[q].num=m.data[i].num;
        cpot[j]++;//当该列上一个元素存储完时序号加1
    }
```

该算法时间复杂度O(nu+tu)，这种算法称为快速转置算法。

----------利用行逻辑连接顺序表实现矩阵相乘----------

```
typedef struct
{
    int i,j;
    int num;
}Triple;
typedef struct
{
    Triple data[MAXSIZE];
    int rpos[MAXRC];//存放每行中首个非零元的位置
    int mu,nu,tu;
}RLSMatrix;//行逻辑连接顺序表的定义
int MultMatrix(RLSMatrix m,RLSMatrix n,RLSMatrix &q)
{//矩阵相乘函数、核心算法
```

```
    int arow,ctemp[MAXRC],i,tp,p,brow,t,ql,ccol;
    if(m.nu!=n.mu)
        return ERROR;
    q.mu=m.mu;q.nu=n.nu;q.tu=0;
    if(m.tu*n.tu!=0)
    {
        for(arow=1;arow<=m.mu;arow++)
        {//按 m 矩阵中每行进行处理
            for(i=1;i<=n.nu;i++)
                ctemp[i]=0;//每行处理开始，使得 ctemp[]置零
            q.rpos[arow]=q.tu+1;//求矩阵 q 中 rpos 的值
            if(arow<m.mu)
                tp=m.rpos[arow+1];
            else
                tp=m.tu+1;//求得 arow 下一行第一个非零所在的位置
            for(p=m.rpos[arow];p<tp;p++)//依次处理矩阵 m 中每行上所有的非零元
            {
                brow=m.data[p].j;
                if(brow<n.mu)
                    t=n.rpos[brow+1];
                else
                    t=n.tu+1;
                for(ql=n.rpos[brow];ql<t;ql++)
                {
                    ccol=n.data[ql].j;
                    ctemp[ccol]+=m.data[p].num*n.data[ql].num;
                }

            }
            for(ccol=1;ccol<=n.nu;ccol++)
            {
                if(ctemp[ccol])
                {
                    if(++q.tu>MAXSIZE) return ERROR;
                    q.data[q.tu].i=arow;
                    q.data[q.tu].j=ccol;
                    q.data[q.tu].num=ctemp[ccol];
                }
            }
        }
    }
    return OK;
}
void getrpos(RLSMatrix &m)
{
```

```
    int i,num[MAXRC],j;
    for(i=1;i<=m.mu;i++)
        num[i]=0;//先使每行上元素的个数为 0
    for(i=1;i<=m.tu;i++)
        num[m.data[i].i]++;//遍历三元组求得每行上元素的个数
    for(j=1;j<=m.mu;j++)
    {
        if(num[j]!=0)
            break;
    }
    m.rpos[j]=1;//j 代表第一个非零元数不为零的行的下标
    for(i=j+1;i<=m.mu;i++)
        m.rpos[i]=m.rpos[i-1]+num[i-1]; //求得每列上第一个元素在转置矩阵三元组中的存储序号

}
---------------十字链表-----------------
//定义
typedef struct OLNode{
    int i,j;//行列下标
    int e;
    struct OLNode *right,*dowm;
}OLNode;
typedef struct ListMatrix{
    OLNode *rhead,*chead;
    int mu,nu,tu;//行数、列数、非零元个数
}ListMatrix;

//将结点插入十字链表表示的矩阵中
void InsertMatrix(OLNode *t,ListMatrix &q)
{
    OLNode *rpre,*cpre,*p;
    int i,tag;
    p=(OLNode *)malloc(sizeof(OLNode));
    p->i=t->i;p->j=t->j;p->e=t->e;
    rpre=&q.rhead[p->i];cpre=&q.chead[p->j];
    for(i=1;i<q.nu+1;i++)
    {
        tag=1;
        if(rpre->right==NULL||rpre->right->j>p->j)  break;
        if(rpre->right->j==p->j) {tag=0;break;}
        rpre=rpre->right;
    }//找到指针 rpre 的位置
    while(1)
    {
        if(cpre->dowm==NULL||cpre->dowm->i>i)  break;
```

```
        cpre=cpre->dowm;
    }//找到 cpre 的位置
    if(tag)//判断该要出入的结点所在的行是否已经存在元素
    {
        p->right=rpre->right;rpre->right=p;
        p->dowm=cpre->dowm;cpre->dowm=p;
    }
    else
    {
        if(rpre->right->e+p->e==0)
        {
            if(rpre->right!=NULL)
            rpre->right=rpre->right->right;
            if(cpre->dowm!=NULL)
            cpre->dowm=cpre->dowm->dowm;
        }
        if(rpre->right->e+p->e!=0)
            rpre->right->e+=p->e;
    }
}
//用十字链表存储矩阵
void CreatMatrix(ListMatrix &m)
{
    int m1,n,t,i;
    OLNode *p,*rpre,*cpre;
    printf("请输入矩阵的行数、列数、非零元个数: ");
    scanf("%d%d%d",&m1,&n,&t);
    m.mu=m1;m.nu=n;m.tu=t;
    m.rhead=(OLNode *)malloc((m1+1)*sizeof(OLNode));
    m.chead=(OLNode *)malloc((n+1)*sizeof(OLNode));
    for(i=1;i<m1+1;i++)
        m.rhead[i].right=NULL;
    for(i=1;i<n+1;i++)//初始化指针的值
        m.chead[i].dowm=NULL;
    printf("请输入这%d 个非零元: \n",m.tu);
    for(i=0;i<m.tu;i++)
    {
        p=(OLNode *)malloc(sizeof(OLNode));
        if(!p) exit(1);
        scanf("%d%d%d",&p->i,&p->j,&p->e);
        InsertMatrix(p,m);
    }
}
----------广义表----------2012/09/01
----------广义表的构造及递归遍历----------
```

```
//广义表的定义用到串的一些操作，上述已有串的定义在此不再叙述。
typedef enum{ATOM,LIST}ElemTag;
typedef struct GLNode{
    ElemTag tag;
    union{
        char atom;
        struct{struct GLNode *hp,*tp;}ptr;
    };//若 atom 占用内存则表明为原子结点，否则 ptr 占用内存为表结点
}*Glist;//广义表结点结构的定义
void SubString(Sstring &A,Sstring &B,int i,int n)
{//取子串操作
    //将 B 串中从第 i 个字符起长度为 n 的字符串复制到 A 中
    int j=0,m=0;
        for(j=i;j<i+n;j++)
            A.ch[m++]=B.ch[j];
        A.ch[m]='\0';
        A.length=m;
}
void SeverGlist(Sstring &str,Sstring &hstr)
{//将非空串（广义表形式的串）str 中第一个逗号之前的
    //字符置给 hstr，剩余的字符（除去该逗号）留在 str 中
    Sstring Ch;int i=0,k=0,n=str.length;
    InitString(Ch);
    do
    {
        i++;
        ClearString(Ch);InitString(Ch);
        SubString(Ch,str,i-1,1);//每次取一个字符至 Ch 串中
        if(Ch.ch[0]=='(')
            k++;
        else
            if(Ch.ch[0]==')')
                k--;
    }while(i<n&&(Ch.ch[0]!=','||k!=0));
        if(i<n)
        {
            SubString(hstr,str,0,i-1);
            SubString(str,str,i,n-i);
        }
        else
        {
            strcpy(hstr.ch,str.ch);
            hstr.length=str.length;
            ClearString(str);
        }
```

```
}
void CreatGlist(Glist &L,Sstring &S)
{//广义表的构造
    //采用头尾链表存储结构,由广义表书写形式串 S 定义广义表
    Glist p,q;Sstring sub,hsub;
    InitString(sub);InitString(hsub);
    if(strcmp(S.ch,"()")==0) L=NULL;
    else
    {
        if(!(L=(Glist)malloc(sizeof(GLNode))))
            exit(1);
        if(S.length==1)
        {//若为单个字符即原子
            L->tag=ATOM;
            L->atom=S.ch[0];
        }
        else
        {
            L->tag=LIST;p=L;//L 为表头
            SubString(sub,S,1,S.length-2);//脱去外层括号
            do
            {
                SeverGlist(sub,hsub);//从 sub 中分离出表头串 hsub
                CreatGlist(p->ptr.hp,hsub);q=p;
                if(!StringEmpty(sub))
                {
                    p=(Glist)malloc(sizeof(GLNode));
                    p->tag=LIST;q->ptr.tp=p;
                }
            }while(!StringEmpty(sub));
            q->ptr.tp=NULL;
        }
    }
    printf("广义表构造成功! \n");
}
void TraverseGlist(Glist &L)
{//递归遍历广义表
    Glist p;
    if(L)
    {
        p=L;
        if(p->tag==ATOM)
            printf("%c ",p->atom);
        else
        {
```

```
            while(p)
            {
                TraverseGlist(p->ptr.hp);
                p=p->ptr.tp;
            }
        }
    }
}
```

----------求广义表的深度、利用递归----------2012/09/02

```
int GlistDepth(Glist &L)
{//递归求广义表的深度
    Glist p;int depth,max=0;
    if(!L) return 1;//空表的深度为 1
    if(L->tag==ATOM) return 0;//原子的深度为 0
    for(p=L;p;p=p->ptr.tp)
    {
        depth=GlistDepth(p->ptr.hp);
        if(max<depth)
            max=depth;//找出各子表中深度的最大值
    }
    return max+1;//广义表的深度为各子表中深度最大的加 1
}
```

----------广义表的复制----------2012、09、02

```
void CopyGlist(Glist &T,Glist L)
{//广义表的复制, 将表 L 复制到表 T 中/利用递归实现
    if(!L) T=NULL;//若为空
    else
    {
        T=(Glist)malloc(sizeof(GLNode));
        T->tag=L->tag;
        if(L->tag==ATOM)
        {//若为原子结点
            T->atom=L->atom;
        }
        else
            if(L->tag==LIST)
            {//若为表结点
                CopyGlist(T->ptr.hp,L->ptr.hp);
                CopyGlist(T->ptr.tp,L->ptr.tp);
            }
    }
}
```

----------非线性数据结构----------

```
//---树、二叉树
//----图
//-----------------------------------------------------------------
2012\08\16
//用二叉链表描述二叉树、并用递归实现二叉树的三种遍历操作
typedef struct BTree{
    int data;
    struct BTree *lchild,*rchild;
}BiTNode,*BiTree;
void CreatBinaryTree(BiTree &T)
{//构造一个二叉树（递归定义）
    int ch;//以 0 表示空树
    printf("输入结点: ");
    scanf("%d",&ch);
    if(ch==0) T=NULL;//约定值为零的结点为空
    else
    {
        T=(BiTNode *)malloc(sizeof(BiTNode));
        if(!T) exit(1);
        T->data=ch;
        CreatBinaryTree(T->lchild);//递归
        CreatBinaryTree(T->rchild);
    }
}
void VistTree(BiTree &T)
{//输出结点
    if(T!=NULL)
        printf("%d ",T->data);
}
void PreOrderTraverse(BiTree &T)
{//递归实现先序遍历
    if(T!=NULL)
        VistTree(T);
    if(T->lchild)
    PreOrderTraverse(T->lchild);
    if(T->rchild)
    PreOrderTraverse(T->rchild);
}
void MiddleOrderTraverse(BiTree &T)
{//递归实现中序遍历
    if(T->lchild!=NULL)
        MiddleOrderTraverse(T->lchild);
    VistTree(T);
    if(T->rchild!=NULL)
        MiddleOrderTraverse(T->rchild);
```

```
}
void LastOrderTraverse(BiTree &T)
{//递归实现后序遍历
    if(T->lchild!=NULL)
        LastOrderTraverse(T->lchild);
    if(T->rchild!=NULL)
        LastOrderTraverse(T->rchild);
    VistTree(T);
}
-----------用遍历实现二叉树的几种简单操作------------
void CountBinaryLeaf(BiTree T,int &count)
{//计算二叉树的叶子结点数<先序遍历>
    if(T)
    {
        if((T->lchild==NULL)&&(T->rchild==NULL))
            count++;//叶子结点数加 1
        CountBinaryLeaf(T->lchild,count);
        CountBinaryLeaf(T->rchild,count);
    }
}
int BinaryTreeDepth(BiTree T)
{//计算二叉树的深度<后序遍历>
    int depth,depthleft,depthright;
    if(T==NULL)
        depth=0;
    else
    {
        depthleft=BinaryTreeDepth(T->lchild);//计算左子树深度
        depthright=BinaryTreeDepth(T->rchild);//计算右子树深度
        depth=(depthleft>=depthright?depthleft:depthright)+1;
        //树的深度为左右子树中深度较大的加上 1
    }
    return depth;
}
BiTree CopyBinaryTree(BiTree T)
{//复制二叉树<后序遍历>
    BiTree leftlink,rightlink,p;
    if(T==NULL)
        return NULL;
    if(T->lchild!=NULL)
    leftlink=CopyBinaryTree(T->lchild);
    else
        leftlink=NULL;//得到左子树指针
    if(T->rchild!=NULL)
    rightlink=CopyBinaryTree(T->rchild);
```

```
    else
        rightlink=NULL;//得到右子树指针
    p=(BiTree)malloc(sizeof(BiTNode));
    if(!p)
        exit(1);
    p->data=T->data;
    p->lchild=leftlink;
    p->rchild=rightlink;//复制结点
    return p;
}

----------用栈结构实现二叉树的非递归遍历----------
typedef struct {
    BiTree *top,*base;
    int length;
}stack;//栈的结构定义
void InitStack(stack &s)
{//初始化一个栈
    s.base=(BiTree *)malloc((MAXSIZE)*sizeof(BiTree));
    if(!s.base)
        exit(1);
    s.top=s.base;
    s.length=0;
    printf("InitStack Successfully!\n");
}
int StackEmpty(stack &s)
{//判断栈是否为空
    if(s.length==0||s.base==NULL)
        return 0;
    else
        return  1;
}
void PushStack(stack &s,BiTree &e)//注意参数类型
{//入栈操作
    //栈中依次存放的是二叉树中各结点，而且不打乱各结点之间的连接关系
    if(s.length>=MAXSIZE)
        s.base=(BiTree *)realloc(s.base,(MAXSIZE+STACK_INCRE_SIZE)*sizeof(BiTree));
    if(!s.base)
        exit(1);
    *s.top=e;
    s.top++;
    s.length++;
}
void PopStack(stack &s,BiTree &e)//注意参数类型
{//出栈操作
```

```
    if(StackEmpty(s))
    {
        s.top--;
        e=*s.top;
        s.length--;
    }
}
void PreOrderTraverse(BiTree &T)
{//先序遍历非递归实现
    stack S;BiTree p;
    InitStack(S);p=T;
    while(p!=NULL||StackEmpty(S))
    {
        if(p!=NULL)
        {
            VistTree(p);
            PushStack(S,p);
            p=p->lchild;
        }
        else
        { PopStack(S,p);p=p->rchild; }
    }
}
void MiddleTree(BiTree &T)
{//中序遍历二叉树非递归实现
    stack S;BiTree p;
    InitStack(S);p=T;
    while(p!=NULL||StackEmpty(S))
    {
        if(p!=NULL)
        {
            PushStack(S,p);
            p=p->lchild;
        }
        else
        {
            PopStack(S,p);
            VistTree(p);
            p=p->rchild;
        }
    }
}
void LastOrderTraverse(BiTree &T)
{//非递归后序遍历
    //用 tag 数组标记对应的结点是否被访问过
```

```
    int sign=-1,tag[MAXSIZE];
    stack S;
    BiTree p,p1;
    p=T;
    InitStack(S);
    while(p!=NULL||StackEmpty(S))
    {
        if(p!=NULL)
        {
            PushStack(S,p);
            tag[++sign]=0;
            p=p->lchild;
        }
        else
        {
            if(tag[sign]==0)
            {//如果结点未被访问则取出栈顶元素，访问栈顶结点的右子树
                Gettop(S,p);
                tag[sign]=1;
                p=p->rchild;
            }
            else
            {//如果该结点被访问过则元素出栈输出
                PopStack(S,p1);
                sign--;
                VistTree(p1);
            }
        }
    }
}
----------二叉树层序遍历：用队列实现-------------
typedef struct {
    BiTree qu[MAX];
    int front,rear;
}Queue;
void InitQueue(Queue &q)
{q.front=0;q.rear=0;}
void EnterQueue(Queue &q,BiTree &p)
{//元素入队
    if((q.rear+1)%MAX!=q.front)
    {
        q.qu[q.rear]=p;
        q.rear=(q.rear+1)%MAX;
    }
    else
```

```
        printf("队列满！\n");
}
void QuitQueue(Queue &q,BiTree &p)
{//元素出队
    if(q.front==q.rear)
        printf("队列空！\n");
    else
    {
        p=q.qu[q.front];
        q.front=(q.front+1)%MAX;
    }
}
void FloorTraverse(BiTree &T)
{//层序遍历
    Queue q;BiTree p;
    InitQueue(q);p=T;
    if(p)
    {
        EnterQueue(q,p);//入队
        while(q.front!=q.rear)
        {
            QuitQueue(q,p);//出队并输出
            printf("%d ",p->data);
            if(p->lchild)
            EnterQueue(q,p->lchild);
            if(p->rchild)
            EnterQueue(q,p->rchild);//左右子树分别入队
        }
    }
}
----------------二叉树中序线索化及遍历-----------
typedef enum pointtag{link,thread};//link=0 表示指针, thread=1 表示线索
typedef struct BTree{
    int data;
    struct BTree *lchild,*rchild;
    pointtag ltag,rtag;
}BiTNode,*BiTree;//线索二叉树的定义
BiTree pre;//定义其为全局变量
void CreatBinaryTree(BiTree &T)
{//构造二叉树
    int ch;
    printf("输入结点: ");
    scanf("%d",&ch);
    if(ch==0) T=NULL;
    else
```

```
    {
        T=(BiTNode *)malloc(sizeof(BiTNode));
        if(!T)
            exit(1);
        T->data=ch;
        T->ltag=link;T->rtag=link;//线索二叉树的构造
        CreatBinaryTree(T->lchild);
        CreatBinaryTree(T->rchild);
    }
}
void InThreading(BiTree &p)
{//中序线索化核心过程
    if(p!=NULL)
    {
        InThreading(p->lchild);
        if(p->lchild==NULL) { p->lchild=pre;p->ltag=thread;}
        if(pre->rchild==NULL) { pre->rchild=p;pre->rtag=thread;}
        pre=p;
        InThreading(p->rchild);
    }
}
void InOrderThreading(BiTree &Thr,BiTree &T)
{//线索化二叉树
    Thr=(BiTree)malloc(sizeof(BiTNode));//建立头结点
    if(!Thr)
        exit(1);
    Thr->ltag=link;
    Thr->rchild=Thr;
    Thr->rtag=thread;
    if(T==NULL) { Thr->lchild=Thr;}
    else
    {
        Thr->lchild=T;//令头结点的左指针指向非空的根结点
        pre=Thr;
        InThreading(T);
        pre->rchild=Thr;
        pre->rtag=thread;
        Thr->rchild=pre;
    }

}
void VistTree(BiTree T)
{
    if(T!=NULL)
        printf("%d ",T->data);
```

```
}
void InOrderTraverse(BiTree &T)
{//遍历中序线索二叉树
    BiTree p;
    p=T->lchild;//p 指向根
    while(p!=T)//当 p 不为头结点时
    {//线索化的二叉树就像一个双向循环链表
        while(p->ltag==link)
            p=p->lchild;//向左到底
        VistTree(p);
        while(p->rtag==thread&&p->rchild!=T)
        {  p=p->rchild; VistTree(p); }
        //若 p 的右指针为线索则 p->rchild 为 p 的后继
        //因此可以直接令 p=p->rchild 然后访问
        p=p->rchild;
        //后继访问完毕则遍历右子树
    }
}
--------------树结构的构造以及实现树的先根和后根遍历----------
typedef struct CSNode{//树的二叉链表定义
    int data;
    struct CSNode *firstchild,*nextslibing;
}CSNode,*CSTree;
typedef struct {
    CSTree *base;
    int front,rear;
}Queue;//队列定义
//省略初始化队列函数
//省略插入队列结点函数
void DeleteQueue(Queue &q)
{//删除队列队头元素
    if(q.front==q.rear)
        printf("队列为空, 无法删除元素! \n");
    else
    {
        q.front=(q.front+1)%MAX;
    }
}
void GetQueueHead(Queue &q,CSTree &e)
{//返回队头元素
    if(q.front==q.rear)
        printf("队列为空! \n");
    else
        e=q.base[q.front];
}
```

```
void CreatTree(CSTree &T)
{//构造树结构
    T=NULL;
    Queue Q;
    CSTree p,s,r;
    InitQueue(Q);
    int fa,ch;//我们约定 0 代表空树,-1 为根结点的根结点
    for(scanf("%d%d",&fa,&ch);ch!=0;scanf("%d%d",&fa,&ch))
    {
        p=(CSTree)malloc(sizeof(CSNode));
        p->data=ch;
        p->firstchild=NULL;
        p->nextslibing=NULL;
        EnterQueue(Q,p);
        if(fa==-1)
            T=p;
        else
        {
            GetQueueHead(Q,s);
            while(s->data!=fa)
            {DeleteQueue(Q);GetQueueHead(Q,s);}
            if(!s->firstchild)
            {s->firstchild=p;r=p;}
            else
            {r->nextslibing=p;r=p;}
        }
    }
}
void PreOrderTraverse(CSTree &T)
{//递归实现二叉树先序遍历、即树的先根遍历
    if(T!=NULL)
        VistTree(T);
    if(T->firstchild)
    PreOrderTraverse(T->firstchild);
    if(T->nextslibing)
    PreOrderTraverse(T->nextslibing);
}
void MiddleOrderTraverse(CSTree &T)
{//递归实现二叉树中序遍历、即树的后根遍历
    if(T->firstchild!=NULL)
        MiddleOrderTraverse(T->firstchild);
    VistTree(T);
    if(T->nextslibing!=NULL)
        MiddleOrderTraverse(T->nextslibing);
}
```

```
//树的层次遍历只需依次输出上述队列中元素即可

------------求树的深度--------------
int TreeDepth(CSTree T)
{//求树的深度
    int h1,h2;
    if(!T) return 0;
    else
    {
        h1=TreeDepth(T->firstchild);
        h2=TreeDepth(T->nextslibing);
        if(h1+1>h2) return h1+1;
        else        return h2;
    }
}
-----------树结构中输出根到叶子结点的所有路径-----------2012、08、27
void AllpathTree(CSTree &T)
{//找到并输出根到叶子结点的所有路径
    CSTree p;
    if(T)
    {
        PushStack(S,T);//结点若不为空则入栈
        if(T->firstchild)//左链若不为空则遍历左链
            AllpathTree(T->firstchild);
        else
        {
            PrintStack(S);//左链若为空则说明为叶子结点，输出当前栈中的元素
             //若此处有 count++;语句则输出的 count 为叶子结点数
            if(StackEmpty(S))
            {
                PopStack(S,p);//退出当前栈顶元素，遍历右链
                AllpathTree(p->nextslibing);
            }
        }
    }
    else
    {
        if(StackEmpty(S)) { PopStack(S,p); AllpathTree(p->nextslibing);}
    }
}
----------构造哈夫曼树及逆向求该树的哈夫曼编码----------2012、08、29
typedef struct {
    int weight;
    int parent,lchild,rchild;
}HTNode,*HuffmanTree;//哈夫曼树结点的定义
```

```
typedef char* *HuffmanCode;//定义字符型二维间址
void CreatHuffmanTree(HuffmanTree &HT,int *w,int n)
{//构造哈夫曼树
    //w 中存放的是 n 个叶子结点的权值
    int i,j,min1,min2,minl,minr;
    HT=(HuffmanTree)malloc((2*n-1)*sizeof(HTNode));
    if(!HT)
        exit(1);
    for(i=0;i<n;i++)
        HT[i].weight=w[i];//HT 前 n 个结点为叶子结点
    for(i=n;i<2*n-1;i++)
        HT[i].weight=0;//初始化剩余结点
    for(i=0;i<2*n-1;i++)
    {//初始化
        HT[i].parent=-1;
        HT[i].lchild=-1;
        HT[i].rchild=-1;
    }
    for(i=n;i<2*n-1;i++)
    {
        min1=min2=MAX;
        for(j=0;j<i;j++)
        {
            if(HT[j].parent==-1)
            {
                if(HT[j].weight<min1) {min1=HT[j].weight;minl=j;}
            }
        }//找到第一个最小的数
        HT[minl].parent=i;
        for(j=0;j<i;j++)
        {
            if(HT[j].parent==-1)
            {
                if(HT[j].weight<min2)  { min2=HT[j].weight;  minr=j;}
            }
        }//找到第二个最小的数
        HT[minr].parent=i;
        HT[i].weight=HT[minl].weight+HT[minr].weight;
        HT[i].lchild=minl;
        HT[i].rchild=minr;//建立树关系
    }
}
void HuffmanTreeCode(HuffmanTree &HT,HuffmanCode &HC,int n)
{//从叶子结点到根结点逆向求哈夫曼编码
    char *cd;int i,j,start,f;
```

```
    cd=(char *)malloc(MAXCHAR*sizeof(char));
    HC=(HuffmanCode)malloc(n*sizeof(char*));`
    cd[MAXCHAR-1]='\0';
    for(i=0;i<n;i++)
    {
        start=MAXCHAR-1;
        for(j=i,f=HT[j].parent;f!=-1;j=f,f=HT[f].parent)
        {   if(HT[f].lchild==j)
                cd[--start]='0';
            else
                cd[--start]='1';
        }
        HC[i]=(char *)malloc((MAXCHAR-start)*sizeof(char));
        strcpy(HC[i],&cd[start]);
    }
}
---------------无栈非递归遍历哈夫曼树求哈夫曼编码-------------
void BLHuffmanTreeCode(HuffmanTree &HT,HuffmanCode &HC,int n)
{//无栈非递归遍历哈夫曼树求哈夫曼编码
    char *cd;
    int cdlen=0,i,p=2*n-2;
    HC=(HuffmanCode)malloc(n*sizeof(char*));
    cd=(char*)malloc((MAXCHAR)*sizeof(char));
    for(i=0;i<=p;i++)
        HT[i].weight=0;//将其权值全部初始化为 0，不影响哈夫曼编码的求取
    while(p!=-1)
    {
        if(HT[p].weight==0)
        {
            HT[p].weight=1;
            if(HT[p].lchild!=-1)
            {
                p=HT[p].lchild;//向左
                cd[cdlen++]='0';
            }
            else
                if(HT[p].rchild==-1)
                {//判断若为叶子结点则求得该结点的哈夫曼编码
                    cd[cdlen]='\0';
                    HC[p]=(char*)malloc((cdlen+1)*sizeof(char));
                    strcpy(HC[p],cd);
                }
        }
        else
            if(HT[p].weight==1)
```

```
            {
                HT[p].weight=2;//第二次访问该结点时将其状态改为 2
                if(HT[p].rchild!=-1)
                {
                    p=HT[p].rchild;
                    cd[cdlen++]='1';
                }
            }
            else
            {
                HT[p].weight=0;
                p=HT[p].parent;//退回父亲结点
                cdlen--;
            }
    }
}
```

----------**图**----------

----邻接矩阵存储表示，实现有向网的构造及各顶点度的计算----2012 年 09 月 11

```
/*邻接矩阵存储表示既是数组存储表示，即可用于有向图（有向网），也可用于无向图（无向网）
优点：便于存储，便于计算图结构中各顶点的度；
缺点：当图中顶点数目较多切顶点之间的联系较少时采用这种存储结构比较浪费存储空间。*/
#include<stdio.h>
#define MAX_NUM 20//最大顶点数目为 20
typedef enum {DG,DN,UDG,UDN} GraphKind;//图类型标志，有向图、有向网、无向图、无向网。
typedef enum {Border,UnBorder} Graph_Border;
typedef struct {
    int data;
}VertexNode;//图中顶点结构定义
typedef struct {
    union {
        Graph_Border adj;//适用于图，表示是否相邻
        int Value;//网，权值(这里定义权值类型为 int 型)
    }Graph_Net;
    char *info;//存储边或者弧的相关信息
}ArcCell,AdjMatrix[MAX_NUM][MAX_NUM];//边或弧结构的定义
typedef struct {
    VertexNode Vertex[MAX_NUM];//存储顶点数组
    AdjMatrix arcs;//存储顶点之间的关系，即边或弧(邻接矩阵)
    int Vexnum,Arcnum;//当前途中顶点数目、边或弧数目
    GraphKind kind;//图的类型
}MGraph;//图结构的定义
void CreatGraph_DN(MGraph &G)
{//构造有向网结构
    int i,j,isign,jsign;
    VertexNode v1,v2;ArcCell arc;
```

```
    printf("构造有向网: \n");
    G.kind=DN;//说明图的结构
    printf("请输入网中顶点数目和弧的数目: ");
    scanf("%d%d",&G.Vexnum,&G.Arcnum);
    printf("请初始化这些顶点: ");
    for(i=0;i<G.Vexnum;i++)
        scanf("%d",&G.Vertex[i].data);
    for(i=0;i<G.Vexnum;i++)
        for(j=0;j<G.Vexnum;j++)
            G.arcs[i][j].Graph_Net.Value=-1;//先初始化邻接矩阵
    printf("请初始化顶点之间的联系 (即弧的信息): \n");
    printf("格式: 顶点 1 顶点 2 权值 (表示顶点 1 邻接到顶点 2) \n");
    for(i=0;i<G.Arcnum;i++)
    {
        scanf("%d%d%d",&v1.data,&v2.data,&arc.Graph_Net.Value);
        isign=-1;jsign=-1;
        for(j=0;j<G.Vexnum;j++)
        {
            if(G.Vertex[j].data==v1.data)
                isign=j;
            if(G.Vertex[j].data==v2.data)
                jsign=j;//找到两顶点之间的关系即弧所在的邻接矩阵中的位置
        }
        if(isign==-1||jsign==-1)
            printf("顶点输入有误! \n");
        else
        {
            G.arcs[isign][jsign].Graph_Net.Value=arc.Graph_Net.Value;
            //初始化邻接矩阵中该位置弧权值的信息
        }
    }

}
int *CountVexNum_DN(MGraph G)
{//有向网各顶点度的计算
    int D[MAX_NUM],ID[MAX_NUM],OD[MAX_NUM];//分别存放顶点的度、入度、出度
    int i,j,k;
    for(i=0;i<G.Vexnum;i++)
    {//初始化
        D[i]=0;
        ID[i]=0;
        OD[i]=0;
    }
    for(i=0;i<G.Vexnum;i++)
    {
```

```
        for(j=0;j<G.Vexnum;j++)
            if(G.arcs[i][j].Graph_Net.Value!=-1)
                OD[i]++;//计算顶点出度
        for(k=0;k<G.Vexnum;k++)
            if(G.arcs[k][i].Graph_Net.Value!=-1)
                ID[i]++;//计算顶点入度
        D[i]=ID[i]+OD[i];//有向图（有向网）中顶点的度为入度加出度
    }
    return D;
}
-----------------邻接表存储表示有向网--------------------
/*-----------------------------------------------------------------------
邻接表存储表示，相对于邻接矩阵存储结构来说当为稀疏图时较为节省存储空间
但用邻接表存储有向图（有向网）时，不便于计算各结点的度
-------------------------------------------------2012 年 09 月 11*/
#define MAX_NUM 20//最大顶点数目
typedef enum {DG,DN,UDG,UDN} GraphKind;//图类型
typedef struct Acnode{
    int adjvex;//存放顶点在数组中的位置
    int value;//权值，这里定义为整型
    struct Acnode *next;
    char *info;//存放弧的信息
}Acnode;//弧的定义
typedef struct {
    int data;
    Acnode *firstarc;
}VertexNode;//顶点定义
typedef struct{
    VertexNode Vertex[MAX_NUM];
    int Vexnum,Arcnum;//当前顶点数目、弧数目
    GraphKind kind;//图的类型
}ALGraph;//图的定义
void CreatGraph_DN(ALGraph &G)
{//有向网的构造，利用邻接表实现
    Acnode *p,*q;int i,j,sign1,sign2,Value;
    VertexNode v1,v2;
    printf("利用邻接矩阵存储表示构造有向网\n");
    G.kind=DN;
    printf("请输入该有向网的顶点数目和弧的数目：");
    scanf("%d%d",&G.Vexnum,&G.Arcnum);
    printf("请初始化这%d 个顶点：",G.Vexnum);
    for(i=0;i<G.Vexnum;i++)
    {
        scanf("%d",&G.Vertex[i].data);
        G.Vertex[i].firstarc=NULL;
```

```
    }
    printf("请初始化顶点之间的关系（%d 条弧）：\n",G.Arcnum);
    printf("形式：顶点 1 顶点 2 权值（表示顶点 1 邻接到顶点 2）\n");
    for(i=0;i<G.Arcnum;i++)
    {
        sign1=-1;sign2=-1;
        scanf("%d%d%d",&v1.data,&v2.data,&Value);
        for(j=0;j<G.Vexnum;j++)
        {
            if(G.Vertex[j].data==v1.data)
                sign1=j;
            if(G.Vertex[j].data==v2.data)
                sign2=j;
        }
        if(sign1==-1||sign2==-1)
            printf("信息输入有误！\n");
        else
        {
            p=(Acnode*)malloc(sizeof(Acnode));
             p->next=NULL;
            p->adjvex=sign2;
            p->value=Value;//构造弧表结点
            q=G.Vertex[sign1].firstarc;
            if(q==NULL)//链入弧表结点至顶点链表中
                G.Vertex[sign1].firstarc=p;
            else
            {
                while(q->next!=NULL)
                q=q->next;
                 q->next=p;
            }
        }
    }
}
int *CountVerNum_DN(ALGraph G)
{//计算各结点的度
    //邻接表存储的有向网，当计算顶点的度时，出度为顶点链表中弧表结点的数目
    //入度则需遍历整个邻接表求得相应的顶点的入度，然后与出度相加得到个顶点的度
    int D[MAX_NUM],ID[MAX_NUM],OD[MAX_NUM];int i;
    Acnode *p;
    for(i=0;i<G.Vexnum;i++)
    {
        D[i]=0;ID[i]=0;OD[i]=0;
    }
    for(i=0;i<G.Vexnum;i++)
```

```
    {
        p=G.Vertex[i].firstarc;
        if(p==NULL)
            OD[i]=0;
        else
            while(p!=NULL)
            {
                OD[i]++;
                ID[p->adjvex]++;
                p=p->next;
            }
    }
    for(i=0;i<G.Vexnum;i++)
        D[i]=ID[i]+OD[i];
    return D;
}
------------------十字链表存储有向图---------------
/*-------------------------------------------------
十字链表存储结构是有向图（有向网）的另一种链式存储结构、
可以把十字链表看成是邻接矩阵的链式存储形式
-----------------------------------------2012 年 9 月 12*/
typedef struct ArcBox{
    int tailvex,headvex;
    struct ArcBox *hlink,*tlink;
    char *info;
}ArcBox;//弧表结点的定义
typedef struct VexNode{
    int data;
    ArcBox *firstin,*firstout;//指向第一个入结点和第一个出结点
}VexNode;//顶点结点的定义
typedef struct {
    VexNode xlist[MAX_NUM];
    int vexnum,arcnum;
}OLGraph;//图的定义
void CreatOLGraph_DN(OLGraph &G)
{//构造十字链表形式的有向图
    int i,j,sign1,sign2;VexNode v1,v2;ArcBox *p,*q;
    printf("--------------有向图十字链表-------------\n");
    printf("请输入该有向图顶点的数目和弧的数目： ");
    scanf("%d%d",&G.vexnum,&G.arcnum);
    if(G.vexnum==0)
        printf("图为空图！\n");
    else
    {
        printf("请初始化这%d 个顶点： ",G.vexnum);
```

```
        for(i=0;i<G.vexnum;i++)
        {
            scanf("%d",&G.xlist[i].data);
            G.xlist[i].firstin=NULL;
            G.xlist[i].firstout=NULL;//初始化指针
        }
    if(G.arcnum==0)
        printf("图为零图！\n");
    else
    {
        printf("请初始化这%d 条弧：",G.arcnum);
        printf("\n 格式：顶点 1 顶点 2（表示顶点 1 邻接到顶点 2）\n");
        for(i=0;i<G.arcnum;i++)
        {
            scanf("%d%d",&v1.data,&v2.data);
            for(j=0;j<G.vexnum;j++)
            {   if(G.xlist[j].data==v1.data)
                    sign1=j;
            if(G.xlist[j].data==v2.data)
                sign2=j;
            }
            q=(ArcBox*)malloc(sizeof(ArcBox));
            q->tailvex=sign1;q->headvex=sign2;
            q->hlink=NULL;q->tlink=NULL;//初始化
            p=G.xlist[sign1].firstout;
            if(p==NULL)
                G.xlist[sign1].firstout=q;//注意不能写成 p=q
            else
            {
                while(p->tlink!=NULL)
                    p=p->tlink;
                p->tlink=q;
            }//链接 firstout 指向的链表
            p=G.xlist[sign2].firstin;
            if(p==NULL)
                G.xlist[sign2].firstin=q;//注意不能写成 p=q
            else
            {
                while(p->hlink!=NULL)
                    p=p->hlink;
                p->hlink=q;
            }//链接顶点中 firstin 指向的链表
        }
    }
}
```

```
}
void TaverseOLGraph(OLGraph &G)
{//输出十字链表形式的有向图
    int i,sign;ArcBox *p;
    printf("\n 输出有向图中结点间的关系: \n");
    for(i=0;i<G.vexnum;i++)
    {
        p=G.xlist[i].firstout;
        while(p)
        {
            sign=p->headvex;
            printf("顶点%d 邻接到顶点%d\n",G.xlist[i].data,G.xlist[sign].data);
            p=p->tlink;
        }
    }
}
-----------------邻接多重表存储无向图-----------------
/*-------------------------------------------------------------------------------
邻接多重表是无向图（无向网）的另一种链式存储结构、邻接多重表是邻接表的一种扩展，
在邻接表中一条边用两个边表结点表示，而在邻接多重表中则用一的边结点表示，则自然邻
接多重表中的链域比邻接多表，而出去 mark 标志域外，邻接多重表和邻接表占用的空间相同
----------------------2012/09/12*/
typedef struct EBox{
    int mark;//标记边是否已被访问
    int ivex,jvex;//顶点在图中的位置域
    struct EBox *ilink,*jlink;
    char *info;
}EBox;//边结点定义
typedef struct VexBox{
    int data;
    EBox *firstedge;//指向第一条依附于该顶点的边
}VexBox;
typedef struct{
    VexBox adjmulist[MAX_NUM];
    int vexnum,edgenum;
}AMLGraph;//图定义
void CreatAMLGraph(AMLGraph &G)
{//邻接多重表无向图的构造，过程类似于十字链表有向图的构造
    int i,j,sign1,sign2;EBox *p,*q,*k;VexBox v1,v2;
    printf("------------邻接多重表构造无向图------------\n");
    printf("请输入无向图的顶点数以及边数: ");
    scanf("%d%d",&G.vexnum,&G.edgenum);
    if(G.vexnum==0)
        printf("该图为空图！\n");
    else
```

```
{
    printf("请初始化这%d 个顶点: ",G.vexnum);
    for(i=0;i<G.vexnum;i++)
    {   scanf("%d",&G.adjmulist[i].data);
    G.adjmulist[i].firstedge=NULL;
    }
    if(G.edgenum==0)
        printf("该图为零图\n");
    else
    {
        printf("请初始化这%d 条边: \n",G.edgenum);
        printf("格式: 顶点 1 顶点 2 (表示顶点 1 与顶点 2 共边) \n");
        for(i=0;i<G.edgenum;i++)
        {
            scanf("%d%d",&v1.data,&v2.data);
             for(j=0;j<G.vexnum;j++)
            {
            if(G.adjmulist[j].data==v1.data)
               sign1=j;
             if(G.adjmulist[j].data==v2.data)
              sign2=j;
             }
        q=(EBox*)malloc(sizeof(EBox));
        q->ivex=sign1;q->jvex=sign2;
        q->ilink=NULL;q->jlink=NULL;
        p=G.adjmulist[sign1].firstedge;
        if(p==NULL)//构造过程重点在于边链表的建立
            G.adjmulist[sign1].firstedge=q;
        else
        {
            while(p)
            {
                k=p;
                if(sign1==p->ivex)//判断路径
                    p=p->ilink;
                else
                    p=p->jlink;
            }
            if(sign1==k->ivex)
                k->ilink=q;
            else
                k->jlink=q;
        }
        p=G.adjmulist[sign2].firstedge;
        if(p==NULL)
```

```
                G.adjmulist[sign2].firstedge=q;
            else
             {
                while(p)
                {
                    k=p;
                    if(sign2==p->ivex)
                        p=p->ilink;
                    else
                        p=p->jlink;
                }
                if(sign2==k->ivex)
                    k->ilink=q;
                else
                    k->jlink=q;
             }
            }
        }
    }
}
```

------------------深度优先搜索（邻接表存储图结构）----------------------2012、09、13

```
int visted[MAX_NUM];//定义为全局变量，存放各顶点是否已被访问标志
void VistGraph(VertexNode V)
{//访问顶点操作
    printf("%d ",V.data);
}
int FirstAdjvex(ALGraph &G,int v)
{//得到图 G 中第 v 个顶点的第一个邻接顶点所在位置
    Acnode *p;p=G.Vertex[v].firstarc;
    if(p==NULL)
        return -1;
    return p->adjvex;
}
int NextAdjvex(ALGraph &G,int v,int w)
{//得到 w 顶点下一邻接点的位置
    Acnode *p;p=G.Vertex[v].firstarc;
    while(p->adjvex!=w&&p!=NULL)
        p=p->next;
    p=p->next;
    if(p==NULL)
        return -1;
    return p->adjvex;
}
void DFS(ALGraph &G,int v)
{//深度优先搜索核心过程。参数 v 表示从图中第 v 个结点开始深度优先遍历
```

```
    int w;
    VertexNode V;V=G.Vertex[v];
    visted[v]=1;VistGraph(V);//令该顶点所在标志为 1 表示已访问，然后访问该顶点
    for(w=FirstAdjvex(G,v);w>=0;w=NextAdjvex(G,v,w))
        if(!visted[w]) DFS(G,w);
}
void DFSTraverse(ALGraph &G)
{//深度优先搜索
    int i;
    for(i=0;i<G.Vexnum;i++) visted[i]=0;//首先初始化访问标志
    for(i=0;i<G.Vexnum;i++)
        if(!visted[i]) DFS(G,i);
}
-----------------广度优先搜索-----------------2012、09、13
void BFS(ALGraph &G,int v)
{
    int w,sign,i;node e,e1;Queue q;VertexNode vv;
    InitQueue(q);
    visted[v]=1;//标志域更换表示已被访问
    e.data=G.Vertex[v].data;InsertQueue(q,e);
    while(!EmptyQueue(q))
    {
        for(i=0;i<G.Vexnum;i++)
            if(G.Vertex[i].data==q.base[q.front].data)
                sign=i;//找到表头结点在图中的位置
        for(w=FirstAdjvex(G,sign);w>=0;w=NextAdjvex(G,sign,w))
            if(!visted[w])//循环的作用是找到表中顶点所有邻接点依次插入队列
            {
                e.data=G.Vertex[w].data;InsertQueue(q,e);
                visted[w]=1;
            }
        DeleteQueue(q,e1);vv.data=e1.data;VistGraph(vv);
//找完顶点的所以邻接点后该顶点出队，然后访问输出
    }
}
void BFSTraverse(ALGraph &G)
{//广度优先搜索
    int i;
    for(i=0;i<G.Vexnum;i++) visted[i]=0;
    for(i=0;i<G.Vexnum;i++)
        if(!visted[i]) BFS(G,i);
}
-------------------深度优先生成树的构造-------------2012、09、15
void DFSTree(ALGraph &G,int v,Tree &T)
{//从图中第 v 个顶点构造深度优先生成树 T
```

```
    int w;Tree p,q;
    visted[v]=1;
    for(w=FirstAdjvex(G,v);w>=0;w=NextAdjvex(G,v,w))
        if(!visted[w])
        {
            p=(Tree)malloc(sizeof(TNode));
            p->Tdata=G.Vertex[w].Gdata;
            p->firstchild=NULL;p->nextsibling=NULL;
            if(T->firstchild==NULL)//First 用来判断该结点的是否已存在左子树
            { T->firstchild=p; }
            else q->nextsibling=p;
            q=p;
            DFSTree(G,w,q);//递归调用
        }
}
void DFSForest(ALGraph &G,Tree &T)
{//构造图 G 的深度优先生成树 T
    int i;Tree p,q;
    T=NULL;
    for(i=0;i<G.Vexnum;i++)
        visted[i]=0;//初始时置顶点为未被访问状态
    for(i=0;i<G.Vexnum;i++)
    if(!visted[i])
    {
        p=(Tree)malloc(sizeof(TNode));
        p->Tdata=G.Vertex[i].Gdata;
        p->firstchild=NULL;p->nextsibling=NULL;
        if(T==NULL) T=p;//如果树为空则令 p 为树根结点
        else q->nextsibling=p;//若为连通图则 else 语句不会执行,构造的
深度优先生成树也只有左子树
        //若为非连通图即通过一个顶点不能遍历图中所有顶点，则构造的
深度优先生成树会有右子树
        //可根据一定的规则遍历右子树求该无向图的连通分量
        q=p;
        DFSTree(G,i,p);//调用构造树函数，构造 p 的深度优先生成树
    }
}
//------------逆向深度优先搜索----------------------2012、09、18
Finashed[]数组中存放的是深度优先搜索时从某个顶点出发调用 DFS 函数完毕时所访问的顶点顺序
详见数据结构笔记讲解
int C_FirstAdjvex(OLGraph &G,int v)
{
    ArcBox *p;p=G.xlist[v].firstin;
    if(p==NULL)
        return -1;
```

```
    return p->tailvex;
}
int C_NextAdjvex(OLGraph &G,int v,int w)
{
    ArcBox *p;p=G.xlist[v].firstin;
    while(p->tailvex!=w&&p!=NULL)
        p=p->hlink;
    p=p->hlink;
    if(p==NULL)
        return -1;
    return p->tailvex;
}
void C_DFS(OLGraph &G,int v)
{//有向图逆向深度优先搜索
    //沿着以该顶点为弧头的弧进行逆向的深度优先搜索
    int w;
    visted[v]=1;
    printf("%d ",G.xlist[v].data);
    for(w=C_FirstAdjvex(G,v);w>=0;w=C_NextAdjvex(G,v,w))
        if(!visted[w]) C_DFS(G,w);
}
void ContraryDFS(OLGraph &G)
{//逆向深度优先搜索
    int i;
    for(i=0;i<G.vexnum;i++) visted[i]=0;
    for(i=G.vexnum-1;i>=0;i--)
        if(!visted[finashed[i]])
        {C_DFS(G,finashed[i]);printf("\n");}
}
```

调用 ContraryDFS 函数时，输出的行数即为该有向图的强连通分量数目，每行所输出的顶点即为一个强连通分量的顶点集。

----------关节点的求取----------2012 年 9 月 19

```
void DFSArticul(ALGraph G,int v)
{//从图中第 v 个顶点出发深度优先搜索求关结点
    int min,w;AcNode *p;
    visted[v]=min=++count;
    for(p=G.Vertex[v].firstarc;p;p=p->next)
    {
        w=p->adjvex;
        if(visted[w]==0)
        {
            DFSArticul(G,w);
            if(low[w]<min) min=low[w];
            if(low[w]>=visted[v]&&!finash[v])//finash[]表示若该结点已经输出过则不再重复输出
            {
```

```
                printf("%d ",G.Vertex[v].Gdata.data);
                finash[v]=1;
            }
        }
        else
            if(visted[w]<min) min=visted[w];//w 为其回边结点
    }
    low[v]=min;
}
void FindArticul(ALGraph G)
{//找图中所有的关节点并输出这些结点
    int i,v,v1=0;AcNode *p;
    count=1;
    visted[0]=1;low[0]=1;//假设从邻接表中第一个结点开始深度搜索,该结点为深度优先生成树的树根
    for(i=1;i<G.Vexnum;i++) { visted[i]=0;finash[i]=0;}//初始化其他还未访问的结点序列
    p=G.Vertex[0].firstarc;v=p->adjvex;
    DFSArticul(G,v);
    if(count<G.Vexnum)
    { printf("%d ",G.Vertex[0].Gdata.data);
    //一次搜索未能遍历图中所以的顶点则说明树根结点至少存在两棵子树, 说明树根为关节点
     while(p->next)
     {
        p=p->next;v=p->adjvex;
        if(visted[v]==0)
        DFSArticul(G,v);
     }
    }
}//求取 low[]的原则详见数据结构笔记。
----------------Prim 算法------------------2012、09、20
typedef struct {
    int adjvex;
    int lowcost;
}close[MAX_NUM];//辅助数组的定义
void MinispanTree_PRIM(MGraph G,MGraph &T)
{//Prim 算法最小生成树的构造
    int k=0,i,j,m,lowcost;//规定从第 0 个顶点开始
    close closedge;length=G.Vexnum;//辅助数组的声明
    T.Vexnum=G.Vexnum;
    for(i=0;i<G.Vexnum;i++)
        T.Vertex[i].data=G.Vertex[i].data;
    for(i=0;i<G.Vexnum;i++)
        for(j=0;j<G.Vexnum;j++)
            T.arcs[i][j].Graph_Net.Value=-1;
    for(i=0;i<G.Vexnum;i++)
    {
```

```
        closedge[i].adjvex=k;
        closedge[i].lowcost=G.arcs[k][i].Graph_Net.Value;
    }
    closedge[k].lowcost=0;//把第 0 个结点并入最小生成树
    for(m=1;m<G.Vexnum;m++)
 {
     lowcost=MAX;
    for(i=1;i<G.Vexnum;i++)
    {
        if(lowcost>closedge[i].lowcost&&closedge[i].lowcost!=0&&closedge[i].lowcost!=-1)
        {
            lowcost=closedge[i].lowcost;
            k=i;
        }
    }
    T.arcs[closedge[k].adjvex][k].Graph_Net.Value=lowcost;
    T.arcs[k][closedge[k].adjvex].Graph_Net.Value=lowcost;
    closedge[k].lowcost=0;//相当于把第 k 个结点并入最小生成树
    for(i=1;i<G.Vexnum;i++)
    {
    if((G.arcs[k][i].Graph_Net.Value<closedge[i].lowcost&&G.arcs[k][i].Graph_Net.Value!=-1)|
|closedge[i].lowcost==-1)
        {
            closedge[i].lowcost=G.arcs[k][i].Graph_Net.Value;
            closedge[i].adjvex=k;
        }
    }
 }
}
----------------Kruskal 算法---------------------2012 年 9 月 22
int pfun(ALGraph &G,int i,int j)
{//判断图 G 中第 i 个顶点和第 j 个顶点是否在同一个连通分量上
    int k;
    for(k=0;k<G.Vexnum;k++)
        visted[k]=0;
    DFS(G,i);
    if(visted[j]==1)
        return 1;
    return 0;
}
void Kruskal(ALGraph G,ALGraph &T)
{//克鲁斯卡尔算法构造最小生成树
    closedge Edge;int i,j=0,flag=1,k,sign,lowcost;AcNode *p,*q;
    Edge.length=0;
    for(i=0;i<G.Vexnum;i++)
```

```
    {//初始化 T(最小生成树中包含图中所有顶点)
        T.Vertex[i].Gdata=G.Vertex[i].Gdata;
        T.Vertex[i].firstarc=NULL;
    }
    T.Vexnum=G.Vexnum;T.Arcnum=0;
    for(i=0;i<G.Vexnum;i++)
    {//将图中的所有的边加入 Edge 中
        p=G.Vertex[i].firstarc;
        while(p)
        {
            for(k=0;k<Edge.length;k++)
            {
                flag=1;
    if((Edge.edge[k].i==i&&Edge.edge[k].j==p->adjvex)||(Edge.edge[k].i==p->adjvex&&Edge.edge
[k].j==i))
                {   flag=0;break;  }
            }
            if(flag)
            {
                Edge.edge[j].i=i;Edge.edge[j].j=p->adjvex;
                Edge.edge[j].value=p->value;j++;Edge.length++;
            }
            p=p->next;
        }
    }
    while(T.Arcnum!=T.Vexnum-1)
    {
    lowcost=MAX;
    for(j=0;j<Edge.length;j++)
    {
        if(lowcost>Edge.edge[j].value&&Edge.edge[j].value!=0)
        {sign=j;lowcost=Edge.edge[j].value;}
    }
    printf("        找        到        最        小        边        为        :        <%d,%d,%d>
",Edge.edge[sign].i,Edge.edge[sign].j,Edge.edge[sign].value);
    if(!pfun(T,Edge.edge[sign].i,Edge.edge[sign].j))
        {//如果该边所依附的两个顶点不在同一个连通分量上则把该边加入最小生成树中
            q=(AcNode *)malloc(sizeof(AcNode));
            q->adjvex=Edge.edge[sign].j;
            q->value=Edge.edge[sign].value;
            q->next=NULL;
            p=T.Vertex[Edge.edge[sign].i].firstarc;
            if(p==NULL)
                T.Vertex[Edge.edge[sign].i].firstarc=q;
            else
```

```
            {
                while(p->next!=NULL)
                    p=p->next;
                p->next=q;
            }
            q=(AcNode *)malloc(sizeof(AcNode));
            q->adjvex=Edge.edge[sign].i;
            q->value=Edge.edge[sign].value;
            q->next=NULL;
            p=T.Vertex[Edge.edge[sign].j].firstarc;
            if(p==NULL)
                T.Vertex[Edge.edge[sign].j].firstarc=q;
            else
            {
                while(p->next!=NULL)
                    p=p->next;
                p->next=q;
            }
            T.Arcnum++;
            Edge.edge[sign].value=0;//表示该边已并入最小生成树
            printf("  成  功  将  边  <%d,%d,%d>  并  入  最  小  生  成  树
\n",Edge.edge[sign].i,Edge.edge[sign].j,lowcost);
    }
    }
}
--------------------------------拓扑排序---------------------------------
/*AOV 网拓扑排序
AOV 网是有向无环网的一种，网中顶点表示活动，弧表示活动执行的优先顺序
拓扑排序输出 AOV 网中所有顶点----2012 年 9 月 21*/
void TopologicalSort(ALGraph G)
{//拓扑排序
    int indegree[MAX_NUM],count=0,k,i;Acnode *p;node e;
    stack s;
    CreatStack(s);
    for(i=0;i<G.Vexnum;i++)
        indegree[i]=0;//初始化顶点入度信息
    for(i=0;i<G.Vexnum;i++)
    {//遍历邻接表求得图中每个顶点的入度
        p=G.Vertex[i].firstarc;
        while(p)
        {
            indegree[p->adjvex]++;
            p=p->next;
        }
    }
```

```
    for(i=0;i<G.Vexnum;i++)//找到 indegree 数组中入度为 0 的顶点然后入栈
        if(!indegree[i]){ e.adjvex=i;   push(s,e);}
    while(!EmptyStack(s))
    {
        pop(s,e);printf("%d ",G.Vertex[e.adjvex]);
        count++;//对输出的顶点计数
        for(p=G.Vertex[e.adjvex].firstarc;p!=NULL;p=p->next)
        {
            k=p->adjvex;
            indegree[k]--;
            if(!indegree[k]) { e.adjvex=k;  push(s,e);}
            //删除下标为 p->adjvex 的顶点相关的边，并将度为 0 的顶点入栈
        }
    }
    if(count<G.Vexnum)
        printf("未能输出图中所有顶点，图中定存在环。\n");
}
----------------关键路径---------------2012 年 9 月 24
int TopologicalSort(ALGraph G,stack &T)
{//拓扑排序、求得各顶点的最早发生时间，并把逆拓扑排序顺序的顶点存入栈 T 中
    ////省略部分内容，省略内容与上述拓扑排序相同
    for(i=0;i<G.Vexnum;i++)//找到 indegree 数组中入度为 0 的顶点然后入栈
    {E[i]=0;//初始化最早发生时间}
    while(!EmptyStack(s))
    {
        pop(s,e);push(T,e);
        count++;//对输出的顶点计数
        for(p=G.Vertex[e.adjvex].firstarc;p!=NULL;p=p->next)
        {
            k=p->adjvex;
            indegree[k]--;
            if(!indegree[k]) { e1.adjvex=k; push(s,e1);}
            if((E[e.adjvex]+p->value)>E[k]){ E[k]=(E[e.adjvex]+p->value);}
            //拓扑排序求得每个顶点的最早发生时间
        }
    }
    return count;
}
void CriticalPath(ALGraph G)
{//关键路径的求取、适用于图中关键路径唯一
    stack T;int i,j,min;node e;Acnode *p;
    CreatStack(T);
    if(TopologicalSort(G,T)<G.Vexnum)
        printf("图 G 中存在环，无法求取关键路径！\n");
    else
```

```
    {
        for(i=0;i<G.Vexnum;i++)
        {   L[i]=E[i];//初始化每个顶点的最迟发生时间   }
        while(!EmptyStack(T))
        {
            pop(T,e);
            if(G.Vertex[e.adjvex].firstarc!=NULL)
            {
                p=G.Vertex[e.adjvex].firstarc;
                min=L[p->adjvex]-p->value;
             for(p=G.Vertex[e.adjvex].firstarc;p!=NULL;p=p->next)
             {
                 if(min>L[p->adjvex]-p->value)
                     min=L[p->adjvex]-p->value;
             }
             L[e.adjvex]=min;//求得各顶点的最迟发生时间
    //最迟发生时间为该顶点邻接点的最迟发生时间减去该顶点到其邻接点的活动所用时间的最小值
            }
        }
        printf("关键路径上的顶点为: ");
        for(j=0;j<G.Vexnum;j++)
            if(visted[j]==0)
            DFS(G,j);
//DFS 经过改进输出图 G 中的关键路径上的顶点.若 E[i]==L[i]则 i 是关键路径上的顶点
        printf("\n");
    }
}
----------------最短路径: Dijkstra 算法--------------------2012 年 9 月 25
Dijkstra 算法适用于求图中的一个源点至图中其他顶点的最短路径。
注意: 此算法图 G 中邻接矩阵当两顶点间不共边时用 MAX 表示, 而不用-1 表示
typedef int Path[MAX_NUM][MAX_NUM];//存放路径
typedef int Dist[MAX_NUM];//存放最短路径长度
void ShortPath_Dijkstra(MGraph G,VertexNode v,Path &P,Dist &D)
{//最短路径_迪杰斯特拉算法
    //以图 G 中顶点 v 为源点求源点到其他顶点的最短路径,路径存放在二维数组 p 中
    //源点到各顶点最短路径长度存放在一维数组 D 中
    int i,v0,j,w,k,n,Final[MAX_NUM],min;
    for(i=0;i<G.Vexnum;i++)
        if(v.data==G.Vertex[i].data) { v0=i; break; }
    for(i=0;i<G.Vexnum;i++)
    {//初始化
        Final[i]=0;
        D[i]=G.arcs[v0][i].Value;
        for(j=0;j<G.Vexnum;j++)
            P[i][j]=0;//首先置路径为空
```

```
        if(D[i]<MAX)
        {
            P[i][v0]=1;//起点
            P[i][i]=1;//终点
        }
    }
    D[v0]=0;Final[v0]=1;
    for(i=1;i<G.Vexnum;i++)
    {//求其他 n-1 个顶点的最短路径
        min=MAX;
        for(j=0;j<G.Vexnum;j++)
            if(min>D[j]&&Final[j]==0)
            { min=D[j];w=j;}
            Final[w]=1;//表示源点到该顶点的最短路径已经找到
        for(k=0;k<G.Vexnum;k++)
            if(Final[k]==0&&D[k]>min+G.arcs[w][k].Value)
            {//修改 D 数组中最短路径长度的值
                D[k]=min+G.arcs[w][k].Value;
                for(n=0;n<G.Vexnum;n++)
                P[k][n]=P[w][n];
                P[k][k]=1;//路径
            }
    }
}
---------------最短路径：Floyd 算法-----------------2012 年 9 月 25
Floyd 算法适用于求图中一对顶点即顶点与顶点之间最短路径的算法。
typedef int Path[MAX_NUM][MAX_NUM][MAX_NUM];//存放路径
typedef int Dist[MAX_NUM][MAX_NUM];//存放最短路径长度
void ShortPath_Floyd(MGraph G,Path &p,Dist &d)
{//Floyd 算法，适用于求图中一对顶点之间的最短路径
    //路径存放在三维数组 p 中，最短路径长度存放在二维数组 d 中
    int i,u,v,w;
    for(v=0;v<G.Vexnum;v++)//初始化
        for(w=0;w<G.Vexnum;w++)
        {
            d[v][w]=G.arcs[v][w].Value;
            for(u=0;u<G.Vexnum;u++)
                p[v][w][u]=0;
                if(d[v][w]<MAX)
                {
                    p[v][w][v]=1;//起点
                    p[v][w][w]=1;//终点
                }
        }
        for(u=0;u<G.Vexnum;u++)
```

```
            for(v=0;v<G.Vexnum;v++)
                for(w=0;w<G.Vexnum;w++)
                    if(d[v][u]+d[u][w]<d[v][w])
                    {
                        d[v][w]=d[v][u]+d[u][w];
                        for(i=0;i<G.Vexnum;i++)
                            p[v][w][i]=p[v][u][i]||p[u][w][i];
                    }
}
```

查找

----------查找表----------

2012 年 9 月 28

静态查找表：顺序查找

当 n 很大时查找效率很低，但使用范围很广

```
int Sequence_search(OrderList l,int num)
{//num 为所要查找的元素的值，返回该值在顺序表中的位置
    int i;
    for(i=0;i<l.length;i++)
        if(l.a[i]==num)
            return i;
}
```

静态查找表：折半查找

只适用于有序的顺序查找表，效率明显高于顺序查找.

```
int Binary_search(OrderList l,int num)
{//折半查找。查找 num 在顺序表 l 中的位置
    //只适用于有序的顺序表
    int i,low,high,mid;
    low=0;//顺序表中的元素位置是从 0 开始的
    high=l.length-1;
    while(low<=high)
    {
        mid=(high+low)/2;
        if(l.a[mid]==num)
            return mid;
        else
            if(l.a[mid]>num)
                high=mid-1;
            else
                low=mid+1;
    }
    return 0;
}
```

----------次优查找树的构造----------2012 年 9 月 28

本算法时间复杂度为 O(nlog2n),其性能略低于最优查找树，但构造最优查找树的时间复杂度较大。

```
int sw[MAXLENGTH];//全局变量，存放记录的累加权值
void NearlyOptimal(BiTree &T,OrderList &L,int low,int high)
{//次优查找树的构造,数组 sw 中存放表 L 记录的累加权值，sw 为全局变量
    int i,j=0, min,dw;
    min=sw[high];dw=sw[high];
    for(i=low;i<=high;i++)
     if(abs(dw-sw[i]-sw[i-1])<min)
     {//abs(dw-sw[i]-sw[i-1])相当于 P(见详解)
         j=i;
         min=abs(dw-sw[i]-sw[i-1]);
     }
     T=(BiTree)malloc(sizeof(BiTNode));
     T->data=L.l[j].num;
     if(j==low)
         T->lchild=NULL;
     else//构造左子树
         NearlyOptimal(T->lchild,L,low,j-1);
     if(j==high)
         T->rchild=NULL;
     else//构造右子树
         NearlyOptimal(T->rchild,L,j+1,high);
}
void CreatOptimalTree(BiTree &T,OrderList &L)
{
    int i;
    if(L.length==0)
       T=NULL;
    else
    {
        sw[0]=L.l[0].weight;
        for(i=1;i<L.length;i++)
        {//求得 sw
            sw[i]=sw[i-1]+L.l[i].weight;//权值累加
        }
        NearlyOptimal(T,L,0,L.length-1);
    }
}
--------------------二叉排序树（二叉查找树）的构造-----------------2012 年 9 月 29
void BinarySortTree(BiTree &T,OrderList L)
{//以顺序表 L 中的结点构造二叉排序树
    int i;BiTree p,q,k;T=NULL;
     for(i=0;i<L.length;i++)
    {
       p=(BiTree)malloc(sizeof(BiTNode));
        p->data=L.l[i].num;
```

```
        p->lchild=NULL;p->rchild=NULL;
        if(T==NULL) T=p;
        else
        {
            if(p->data<T->data)
            {//p 结点为左子树的结点
                q=T->lchild;
                if(q==NULL)   T->lchild=p;
                else
                {
                    while(q!=NULL)
                    {
                        k=q;
                        if(p->data<q->data) q=q->lchild;
                        else  q=q->rchild;
                    }
                    if(p->data<k->data) k->lchild=p;
                    else   k->rchild=p;
                }
            }
            else
            {//p 结点为右子树上的结点
                q=T->rchild;
                if(q==NULL)  T->rchild=p;
                else
                {
                    while(q!=NULL)
                    {
                        k=q;
                        if(p->data<q->data)  q=q->lchild;
                        else   q=q->rchild;
                    }
                    if(p->data<k->data)  k->lchild=p;
                    else   k->rchild=p;
                }
            }
        }
    }
}
----------------------动态查找树在二叉排序树中进行插入操作--------------2012 年 9 月 29
int SearchBST(BiTree T,BiTree f,int key,BiTree &p)
{//查找二叉排序树 T，f 为 T 的根结点初始时为 NULL
    //若查找成功，则 p 指向 key 所在的结点并返回 TRUE
    //否则 p 指向该路径上最后一个结点并返回 FALSE
    if(T==NULL) {p=f;return FALSE;}
```

```
    else
        if(key==T->data)
        {
            p=T;
            return TRUE;
        }
        else
            if(key<T->data)
                return SearchBST(T->lchild,T,key,p);
            else
                return SearchBST(T->rchild,T,key,p);
}
int InsertBST(BiTree &T,int key)
{//当二叉排序树 T 中不存在关键字 key 则将 key 插入其中并返回 TRUE
    //若存在关键字 key 则返回 FALSE
    BiTree p,s;
    if(!SearchBST(T,NULL,key,p))
    {
        s=(BiTree)malloc(sizeof(BiTNode));
        s->data=key;
        s->lchild=NULL;s->rchild=NULL;
        if(!p) T=s;
        else
            if(s->data<p->data)
                p->lchild=s;
            else
                p->rchild=s;
        return TRUE;
    }
    return FALSE;
}
-------------二叉排序树中删除结点操作-------------2012 年 9 月 29
int Deletefun(BiTree &p)
{//从二叉排序树中删除指针 p 所指向的结点
    BiTree q,s;
    if(p->rchild==NULL)
    {//当右子树为空则只需要重接其左子树
        q=p;p=p->lchild;
        //函数形参为指针的引用因此 p=p->lchild 相当于 f->lchild=p->lchild
        //其中 f 为指针 p 所指结点的父母
        free(q);
    }
    else
        if(p->lchild==NULL)
        {//左子树为空则只需要重接其右子树
```

```
            q=p;p=p->rchild;
            free(q);
        }
        else
        {//左右子树都不为空，详见笔记
            q=p;s=p->lchild;
            while(s->rchild!=NULL)
            {
                q=s;
                s=s->rchild;
            }
            p->data=s->data;
            if(q!=p)
                q->rchild=s->lchild;
            else
                q->lchild=s->lchild;
            free(s);
        }
        return TRUE;
}
int DeleteBST(BiTree &T,int key)
{
    if(T==NULL)
        return FALSE;
    else
        if(key==T->data)
            return Deletefun(T);
        else
            if(key<T->data)
                return DeleteBST(T->lchild,key);
            else
                return DeleteBST(T->rchild,key);
}
-----------------平衡二叉排序树的构建-------------2012 年 9 月 30
#define LH +1//左边高
#define EH 0//一样高
#define RH -1//右边高
typedef struct BSTNode{
    int data;
    int bf;//平衡因子
    struct BSTNode *lchild,*rchild;
}BSTNode,*BSTree;//平衡二叉排序树结构的定义
void R_Rotate(BSTree &p)
{//对以*p 为根的二叉排序树做右旋处理，处理之后 p 指向新的树根结点
    BSTree lc;
```

```
    lc=p->lchild;
    p->lchild=lc->rchild;
    lc->rchild=p;p=lc;
}
void L_Rotate(BSTree &p)
{//对以*p 为根的二叉排序树做左旋处理，处理之后 p 指向新的树根结点
    BSTree rc;
    rc=p->rchild;
    p->rchild=rc->lchild;
    rc->lchild=p;p=rc;
}
void LeftBalance(BSTree &T)
{//对已*T 为根的二叉排序树做左平衡旋转处理
    BSTree lc,rd;
    lc=T->lchild;
    switch(lc->bf)
    {
    case LH:T->bf=lc->bf=EH;
        R_Rotate(T);
        break;
    case RH:rd=lc->rchild;
        switch(rd->bf)
        {
        case LH:T->bf=RH;lc->bf=EH;
            break;
        case EH:T->bf=lc->bf=EH;
            break;
        case RH:T->bf=EH;lc->bf=RH;
            break;
        }
        rd->bf=EH;
        L_Rotate(T->lchild);
        R_Rotate(T);
    }
}
void RightBalance(BSTree &T)
{//对已*T 为根的二叉排序树做右平衡旋转处理
    BSTree lc,rd;
    lc=T->rchild;
    switch(lc->bf)
    {
    case RH:T->bf=lc->bf=EH;
        L_Rotate(T);
        break;
    case LH:rd=lc->lchild;
```

```
        switch(rd->bf)
        {
        case RH:T->bf=LH;lc->bf=EH;
            break;
        case LH:T->bf=EH;lc->bf=RH;
            break;
        case EH:T->bf=lc->bf=EH;
            break;
        }
        rd->bf=EH;
        R_Rotate(T->rchild);
        L_Rotate(T);
    }
}
int InsertAVL(BSTree &T,int key,bool &taller)
{//若在平衡二叉排序树中不存在与关键字 key 相等的结点，则将关键字插入树中
    //布尔变量 taller 表示树是否“长高”
    if(T==NULL)
    {
        T=(BSTree)malloc(sizeof(BSTNode));
        T->data=key;T->bf=EH;//叶子结点其平衡因子肯定为 0
        T->lchild=T->rchild=NULL;
        taller=1;//树长高了
    }
    else
    {
        if(key==T->data)
        {//如果树中已存放此关键字则不予插入
            taller=0;
            return 0;
        }
        if(key<T->data)
        {//关键字小于根结点则插入其左子树中
            if(!InsertAVL(T->lchild,key,taller))
                return 0;
            if(taller)
            {//如果树长高了，判断是否平衡
                switch(T->bf)
                {
                case LH:LeftBalance(T);//不平衡时调用左平衡函数，使左子树平衡
                    taller=0; break;
                case EH:T->bf=LH;taller=1;
                    break;
                case RH:T->bf=EH;taller=0;
                    break;
```

```
            }
        }
    }
    else
    {//插入右子树中
        if(!InsertAVL(T->rchild,key,taller))
            return 0;
        if(taller)
        {
            switch(T->bf)
            {
            case LH:T->bf=EH; taller=0;
                break;
            case EH:T->bf=RH;taller=1;
                break;
            case RH:RightBalance(T);taller=0;
                break;
            }
        }
    }
}
return 1;
}
----------------双链树（键树）的插入、删除及查找操作----------------2012 年 10 月 3
#define MAXKEY 16//关键字最大长度
typedef struct {
    char ch[MAXKEY];//关键字域
    int num;//关键字长度
}KeyType;//字符型关键字结构定义
typedef struct DLTNode{
    char symbol;//关键字中的单个字符
    struct DLTNode *first,*next;
}DLTNode,*DLTree;//用二叉链表表示键树的存储结构
int SearchDLTree(DLTree T,KeyType k)
{//在双链树 T 中查找关键字 k，查找成功返回 1，否则返回 0
    DLTree p;int i;
    if(T==NULL)
    {printf("树空查找失败！\n");return 0;}
    else
    {
        p=T->first;i=0;
        while(p&&i<k.num)
        {
            while(p&&p->symbol!=k.ch[i])
                p=p->next;
```

```
            if(p&&i<k.num-1)
                p=p->first;
             i++;
        }
        if(p==NULL) return 0;
        else  return 1;
    }
}
void InsertDLTree(DLTree &T,KeyType k)
{//把关键字插入双链树 T 中
    DLTree p,q,pt;int i=0;
    if(k.num==1||k.ch[k.num-1]!='#')
        printf("请以正确的格式输入关键字！\n");
    else
    {
        if(T==NULL)
        {
            T=(DLTree)malloc(sizeof(DLTNode));
            T->first=T->next=NULL;
            p=T;q=p;
            while(i<k.num)
            {
                p=(DLTree)malloc(sizeof(DLTNode));
                p->first=p->next=NULL;
                p->symbol=k.ch[i];
                q->first=p;
                q=p;i++;
            }
            printf("关键字%s 插入成功！\n",k.ch);
        }
        else
        {
            if(!SearchDLTree(T,k))
            {//若 T 中无关键字 k 则插入之
                i=0;p=T->first;q=p;
                while(p)
                {
                    while(p&&p->symbol!=k.ch[i])
                    {q=p;p=p->next;}
                    if(p) {q=p;p=p->first;}
                    i++;
                }//第 i 个结点应插入到 q 指针所指向的结点之后
                i--;
                p=(DLTree)malloc(sizeof(DLTNode));
                p->first=p->next=NULL;
```

```
                p->symbol=k.ch[i];
                q->next=p;q=p;
                i++;
                while(i<k.num)
                {
                    p=(DLTree)malloc(sizeof(DLTNode));
                    p->first=p->next=NULL;
                    p->symbol=k.ch[i];
                    q->first=p;q=p;
                    i++;
                }
                printf("关键字%s 插入成功! \n",k.ch);
            }
            else
                printf("插入失败, 关键字已存在! \n");
        }
    }
}
int Judge(DLTree &p)
{//判断*p 结点以下结点的 next 域是否为空, 如果为空则返回 1
    DLTree q;
    if(p->first==NULL)
        return 1;
    q=p->first;
    while(q)
    {
        if(q->next!=NULL)
            return 0;
        q=q->first;
    }
    return 1;
}
void DeleteDLTree(DLTree &T,KeyType k)
{//删除双链树中的关键字 k
    int i;DLTree p,q,pt;
    if(!SearchDLTree(T,k))
        printf("树中不存在此关键字! \n");
    else
    {
        p=T->first;q=p;i=0;
        while(i<k.num)
        {
            while(p->symbol!=k.ch[i])
            {q=p;p=p->next;}
            if(Judge(p))
```

```
                break;
            else
            {q=p;p=p->first;}
            i++;
        }
        if(q->next==p) q->next=p->next;
        else
            q->first=p->next;//重新链接剩余部分
        pt=p->first;
        while(pt)//删除以结点*p 为头的结点链接
        {
            free(p);p=pt;pt=pt->first;
        }
        printf("关键字%s 已删除! \n",k.ch);
    }
}
```

------------------哈希表----------------------2012 年 10 月 4

```
typedef struct{
    int *elem;//定义关键字为整型
    int count;//当前哈希表中的关键字个数
    int length;//当前哈希表的容量
}HashTable;
HashTable H;
int Hash(int key)//哈希函数
{return key%11;}
int Collision(int p,int c)
{//冲突处理函数
    //以线性探测再散列作为冲突处理方式
    return (p+c)%H.length;//重新找到的哈希地址
}
int SearchHash(HashTable H,int key,int &p,int &c)
{//查找关键字 key 在哈希表中的位置, 如果查找成功则返回 SUCCESS
    //用 p 返回关键字在哈希表中的位置。否则返回 UNSUCCESS
    //用 p 返回关键字要插入的位置,c 为处理冲突的次数
    int q;
    p=q=Hash(key);//取得关键字的哈希地址
    while(H.elem[p]!=0&&H.elem[p]!=key)
        p=Collision(q,++c);//若该存储位置上已有关键字并且关键字值与 key 不相等
    //则进行冲突处理
    if(H.elem[p]==key)
        return SUCCESS;
    else
    return UNSUCCESS;
}
int InsertHash(HashTable &H,int key)
```

```
{//若哈希表 H 中无关键字 key 则插入之
    int c=0,p;
    if(SearchHash(H,key,p,c))
        return ERROR;
    if(c<H.length/2)
    {
        printf("哈希地址为%d ",p);
        H.elem[p]=key;
        H.count++;
        printf("冲突次数为%d ",c);
        return OK;
    }
    else
    {
        printf("冲突次数过多，请定义合适的哈希函数！\n");
        return ERROR;
    }
}
```

----------内部排序----------

----------直接插入排序----------

```
typedef struct {
    int key[MAXSIZE+1];//关键字域，0 单元起"哨兵"作用
    int length;//表长度
}Sqlist;//关键字顺序的定义
void InsertSort(Sqlist &L)
{//直接插入排序。非递减排列,时间复杂度为 O(n2)
    int i,j;
    for(i=2;i<=L.length;i++)
        if(L.key[i]<L.key[i-1])
        {
            L.key[0]=L.key[i];//更换哨兵
            L.key[i]=L.key[i-1];
            for(j=i-2;L.key[0]<L.key[j];j--)
                L.key[j+1]=L.key[j];
            L.key[j+1]=L.key[0];//哨兵入列
        }
}
```

----------折半插入排序----------

```
/*折半插入排序。相对直接插入排序来说改进了比较次数
记录移动次数并未改变、时间复杂度为 O(n2)*/
void BinarySort(Sqlist &L)
{//折半插入排序
    int low,high,i,m,j;
    for(i=2;i<=L.length;i++)
```

```
    {
        L.key[0]=L.key[i];
        low=1;high=i-1;
        while(low<=high)
        {//折半查找得到第 i 个结点在前面有序表中的插入位置
            m=(low+high)/2;
            if(L.key[i]<L.key[m]) high=m-1;
            else low=m+1;
        }
        for(j=i-1;j>=high+1;j--) L.key[j+1]=L.key[j];//记录向后移动
        L.key[high+1]=L.key[0];//插入结点
    }
}
--------------------2-路插入排序----------------------
void TwoRoadSort(Sqlist &L)
{//2-路插入排序。相对直接插入排序来讲减少了记录移动的次数
    //时间复杂度仍是 O(n2)
    int d[MAXSIZE],i,j,k,first,final;
    //first 指针指向比 d[1]小的最小的那个元素位置
    //final 指向比 d[1]大的最大的那个元素的位置
    //当所有关键字按非递减 2-路插入有序排列至数组 d 中后 first=final+1
    for(i=1;i<=L.length;i++)
        d[i]=0;//辅助数组的初始化
    first=1;final=2;d[1]=L.key[1];
    for(i=2;i<=L.length;i++)
    {
        if(L.key[i]<d[1])
        {//如果比第一个结点小则插入其前面，即从插入最"后面"
            for(j=L.length;j>=first;j--)
                if(d[j]!=0&&d[j]>L.key[i]);
                else break;
                if(d[j]==0)
                { d[j]=L.key[i];first=j;}
                else
                {
                    for(k=first-1;k<j;k++)
                        d[k]=d[k-1];
                    first=first-1;
                    d[k]=L.key[i];
                }
        }
        else
        {//如果比第一个结点小则插入到该结点后面
            for(j=2;j<=final;j++)
                if(d[j]!=0&&d[j]<L.key[i]);
```

```
                else break;
                if(d[j]==0)
                {d[j]=L.key[i];final=j;}
                else
                {
                    for(k=final+1;k>j;k--)
                        d[k]=d[k-1];
                    final=final+1;
                    d[k]=L.key[i];
                }
        }
    }
    k=1;
    for(i=first;i<=L.length;i++)
        L.key[k++]=d[i];
    for(i=1;i<=final;i++)
        L.key[k++]=d[i];//把数组 d 中有序记录重新复制到顺序表 L 中
}
```

------------------希尔排序------------------

希尔排序也是插入排序的一种，它是这几种插入排序中时间复杂度唯一比前几种插入排序时间复杂（O(n2)）度小的排序方法。只是增量序列应选的适当。

```
void ShellInsert(Sqlist &L,int dk)
{//按 dk 增量对顺序表做一趟希尔排序
    int i,j;
    for(i=dk+1;i<=L.length;i++)
        if(L.key[i]<L.key[i-dk])
        {
            L.key[0]=L.key[i];
            for(j=i-dk;j>0&&L.key[0]<L.key[j];j=j-dk)
                L.key[j+dk]=L.key[j];
            L.key[j+dk]=L.key[0];
        }
}
void ShellSort(Sqlist &L,int dlta[],int t)
{//按增量序列 dlta 做希尔排序
    int k;
    for(k=0;k<t;k++)
        ShellInsert(L,dlta[k]);
}
```

----------------------快速排序----------------------

快速排序是对起泡排序的一种改进。如果单看平均排序时间那快速排序是最好的排序方法，一般取第一个元素为枢轴记录，但当顺序表有序或基本有序时取第一个记录为枢轴会使得快速排序退化为起泡排序（时间复杂度为 O(n2)）。因此为避免快速排序退化为起泡排序通常在表中记录有序或基本有序时采用"三者取中"法来取得枢轴结点。

```
int Partition(Sqlist &L,int low,int high)
```

```
{//枢轴是随机选取的，通常选择第一个记录为枢轴
    //使序列位置从 low 到 high 位置的记录做以下排列：
    //比枢轴结点大的记录排列的枢轴之后，否则排列在其前面
    //返回排列后枢轴记录在顺序表中位置
    int pivotkey;
    L.key[0]=L.key[low];//选第一个元素为枢轴记录存储在 0 单元位置
    pivotkey=L.key[low];//取得枢轴记录关键字
    while(low<high)
    {
        while(low<high&&L.key[high]>=pivotkey) high--;
        L.key[low]=L.key[high];
        while(low<high&&L.key[low]<=pivotkey) low++;
        L.key[high]=L.key[low];
    }
    L.key[low]=L.key[0];//将枢轴记录存入排序后的枢轴所在的新位置
    return low;//返回枢轴结点所在的新位置
}
void QSort(Sqlist &L,int low,int high)
{//对顺序表 L 中从 low 到 high 的序列做快速排序
    int p;
    if(low<high)
    {
        p=Partition(L,low,high);
        QSort(L,low,p-1);//对低子表快速排序，p 是枢轴位置
        QSort(L,p+1,high);//对高子表快速排序
    }
}
void QuickSort(Sqlist &L)
{//快速排序
    QSort(L,1,L.length);
}
--------------选择排序-----------------
简单选择排序：
void SelectSort(Sqlist &L)
{//简单选择排序。时间复杂度为 O(n 的平方)
    int i,j,sign,change;
    for(i=1;i<L.length;i++)
    {
        sign=i;
        for(j=i;j<L.length+1;j++)
            if(L.data[sign]>L.data[j])
                sign=j;//选择较小关键字所在位置
        if(i!=sign)
        {
            change=L.data[i];
```

```
            L.data[i]=L.data[sign];
            L.data[sign]=change;
        }
    }
}
```

堆排序：

```
void HeapAdjust(Sqlist &L,int s,int m)
{//调整（筛选）L.data[s]使L.data[s....m]成为一个大顶堆
    int rc,j;rc=L.data[s];
    for(j=2;j<=m;j=j*2)
    {
        if(j<m&&L.data[j]>L.data[j+1]) j++;
        if(rc<L.data[j]) break;
        L.data[s]=L.data[j];s=j;
    }
    L.data[s]=rc;
}
void HeapSort(Sqlist &L)
{//堆排序。时间复杂度为O(nlog2n)
    int i,change;
    for(i=L.length/2;i>=0;i--)
        HeapAdjust(L,i,L.length);//将L.data[1...L.length]建成大顶堆
    for(i=L.length;i>1;i--)
    {
        change=L.data[1];
        L.data[1]=L.data[i];
        L.data[i]=change;//最后一个记录相互交换
        HeapAdjust(L,1,i-1);//将L.data[1...i-1]重新调整为大顶堆
    }
}
```

2-路归并排序：

```
void Merge(Sqlist L,Sqlist &T,int i,int m,int n)
{//将顺序表L中的[1....m-1]和[m...n]归并为顺序表T
    int j,k;
    for(j=m+1,k=i;i<=m&&j<=n;k++)
    {
        if(L.data[i]<L.data[j]) T.data[k]=L.data[i++];
        else T.data[k]=L.data[j++];
    }
    if(i<=m)
        for(;i<=m;)
            T.data[k++]=L.data[i++];
    if(j<=n)
        for(;j<=n;)
            T.data[k++]=L.data[j++];
```

```
}
void MSort(Sqlist L,Sqlist &T,int s,int t)
{//2-路归并排序利用递归实现效率较低。时间复杂度为O(nlog2n)
    int m;Sqlist S;S.length=0;
    if(s==t) T.data[s]=L.data[s];
    else
    {
        m=(s+t)/2;
        MSort(L,S,s,m);
        MSort(L,S,m+1,t);
        Merge(S,T,s,m,t);
    }
}
void MergeSort(Sqlist &L)
{
    MSort(L,L,1,L.length);
}
-------------基数排序---------------
void Distribute(SLList l,int i,ArrType &f,ArrType &e)
{//按LSD分配算法。
    int p,j;
    for(j=0;j<RADIX;j++) f[j]=0;//指针初始化
    for(p=l.r[0].next;p!=0;p=l.r[p].next)
    {
        j=ord(l.r[p].key,i);//取得低位关键字在记录中的下标
        if(f[j]==0) f[j]=p;//若第j个静态链表为空则头指针直接指向即可
        else l.r[e[j]].next=p;//若第j个静态链表不为空则从后面插入
        e[j]=p;//更改尾指针的指向
    }
}
void Collect(SLList &r,int i,ArrType f,ArrType e)
{//收集算法。与分配算法相对
    int j,t;
    for(j=0;!f[j];j++);//找到第一个非空子表
    r.r[0].next=f[j];t=e[j];//r.r[0]指向第一个非空子表的第一个结点
    while(j<RADIX)
    {
        for(j++;j<RADIX-1&&!f[j];j++);//找到下一个非空子表
        if(f[j]) {r.r[t].next=f[j];t=e[j];}
    }
    r.r[t].next=0;//收集完成后最后一个结点的next指针指向0
}
void RadixSort(SLList &L)
{//基数排序.时间复杂度为O(d(n+rd)),其中d为记录中关键字的个数
    //rd为基数的取值范围,n为记录的个数
```

```
    int i;ArrType f,e;
    for(i=0;i<L.recnum;i++) L.r[i].next=i+1;
    L.r[L.recnum].next=0;//最后一个记录的 next 指针指向 0
    for(i=0;i<L.keynum;i++)
    {
        Distribute(L,i,f,e);
        Collect(L,i,f,e);
    }
}
```