#include
#include<stdio.h>
#include
using namespace std;
//线性表（逻辑类型）又分为（物理类型）顺序存储结构和链式存储结构
/*
表里存的是抽象数据类型 ElemType
Operation 包括：
 InitList(*L)建立空线性表，初始化操作*
 ListEmpty(L)判断线性表是否为空，为空则返回 false
 *ClearList(*L)清空线性表
 GetElem(L,i,*e)将线性表 L 中第 i 个元素赋给 e*
 LocateElem(L,e)L 中查找与 e 相同的元素，找的则返回下标，未找到则返回 0
 *ListInsert(*L,i,e)在线性表 L 中第 i 个位置插入新元素 e
 ListDelete(*L,i,*e)删除线性表 L 中第 i 个位置的元素，并用 e 保存其值
 ListLength(L)返回线性表 L 元素的个数
*/

#define ElemType int //假设数据元素是 int

//顺序存储 以下标为中心操作
 //1.顺序存储结构封装 (整个表当作一个完整类型)

```
#define MAXSIZE 20
    typedef struct     //封装数组
    {
        int data[MAXSIZE];
        int length;   //线性表当前长度(不是maxsize)
    }SqList;



//2.地址计算
LOC(i + 1) = LOC(i) + sizeof(ElemType)  //loc 获得该数对应下标的地址
LOC(i) = LOC(1) + (i - 1) * sizeof(ElemType)

//3.取              无需对表本身进行什么操作，故传入是L
bool GetElem(SqList L, int i, ElemType* e)   //用e返回取得的第i个元素
{
    if (L.length == 0 || i<1 || i>L.length)  //防止i超范围
    {
        return 0;
    };
    *e = L.data[i - 1];   //顺序存储里都是一个挨一个，按顺序存的
    return 1;
```

```
}

//4.插入（插入到第 i 个位置前）    需要对表内数据进行操作，故传入*L
bool ListInsert(SqList* L, int i, ElemType e)
{
    int k;
    if (L->length == MAXSIZE || i<1 || i>L->length + 1)   /*当表已经满了，
或者 i 不在范围*/
    {
        return 0;
    };
    if (i <= L->length)   //当想要插入的位置不在最后面时 需要把它后面所有元
素都后移一位
    {
        for (k = L->length - 1; k >= i - 1; k--)  //从最后一个开始往后挪
        {
            L->data[k + 1] = L->data[k];
        };
    };
    L->data[i - 1] = e;    /*插入到第 i 个位置之前，即成为第 i 个元素，即插到
下标 i-1 处，第 i 个元素( 下标 i-1 处的原元素)往后移*/

    L->length++;
    return 1;
}
//5.删除（删除第 i 个元素）
bool ListDelete(SqList* L, int i, ElemType* e)
{
    int k;
    if (L->length == MAXSIZE || i<1 || i>L->length + 1)   /*当表已经满了，
或者 i 不在范围*/
    {
        return 0;
    };
    *e = L->data[i - 1];
    for (k = i - 1; k < L->length; k++)   //当不是删除最后一个人元素时，需
要把后面的都往前移；当是删除最后一个人元素时，最后一个被空白覆盖.length-1 是最
后一个元素的下标
    {
        L->data[k] = L->data[k + 1];
    };

    L->length--;
    return 1;
}
```

//链式存储 以指针为中心操作 本质上是在用指针对死表进行编辑，查找操作，每一个结点其实都是相同的，在操作时都是用指针来具体指向谁，来代表某个节点

//存储数据元素的区域成为数据域，存储后继位置的域称为指针域。这两部分组成结点。n 个结点连接成链表
//头指针：指向链表第一个结点（有头结点就指向头节点）的指针
//头结点：为操作统一设立的，这样在第一个结点之前插入结点和删除第一个结点的操作就和其他结点统一了
//头结点不必要，头指针很必要，常常把头指针冠以整个链表的名字

```
//1.单链表存储结构
struct Node;
typedef struct Node
{
    ElemType data;   //数据域
    struct Node* Next;  //指针域
}Node;
typedef struct Node* LinkList;

//2.单链表的读取
bool GetElem(LinkList L, int i, ElemType* e)
{
    int j;
    LinkList p;                 //用L 直接挪，会导致丢失记载着头结点位置的L。要避免 L 被改变，L 只作为一个地址传进来给 q 用于调用链表，但是 L 本身存的地址不能改
    p = L->Next;  //p 此时为第一个结点的地址
    j = 1;
    while (p && j < i)              //得到第 i 个元素的位置
    {
        p = p->Next;
        ++j;
    };
    if (!p || j > i)        //如果第 i 个为空或者输入的 i 小于 1 了
    {
        return 0;
    };
    *e = p->data;
    return 1;
};
//单链表读取就需要遍历，不像顺序存储直接有下标就能取

//3.单链表的插入
bool ListInsert(LinkList* L, int i, ElemType e)   //把 e 插入到第 i 个位置
{
    int j;
    LinkList p,s;
    p = *L;           //和 2 中初始化不同，经过下面 while 循环得到的是第 i-1 个元素的位置
    j = 1;
```

```
    while (p && j < i)
    {
        p = p->Next;
        j++;
    }
    if (!p || j > i)
    {
        return 0;
    };
    s = (LinkList)malloc(sizeof(Node));
    s->data = e;
    s->Next = p->Next;
    p->Next = s;
    return 1;

}
//4.单链表的删除
bool ListDelete(LinkList* L, int i, ElemType* e)
{
    int j;
    LinkList p,q;
    p = *L;             //从头指针开始往后找，经 while 循环找到第 i-1 个结点位
置
    j = 1;
    while (p->Next && j < i)
    {
        p = p->Next;
        ++j;
    }
    if (!(p->Next) || j > i)     //若第 i 个位置上为空或输入的 i<1
    {
        return 0;
    };
    q = p->Next;
    p->Next = q->Next;
    *e = q->data;
    free(q);
    return 1;
}

//5.头插法建立单链表   结点都插入第一个位置
void CreateListHead(LinkList* L, int n)
/*因为函数内部要修改指针，故而传入指向指针的指针。同理，要修改变量，就传入指向
变量的指针。
如果你仅仅把 L 传入进去，传入的是地址，不是指针变量，在函数里面重分配空间得到新
的地址后，函数外面 L 还是刚开始创建时初始分配地址，
一出函数，函数内新建链表的头指针就丢了*/
{
    LinkList p;
```

```
    int i;

    srand(time(0));    //随机数种子
    *L = (LinkList)malloc(sizeof(Node));          //创建头结点，用*L 记录位
置
    (*L)->Next = NULL;                            //头结点初始化 Next 为NULL


    for (i = 0; i < n; i++)
    {
        p = (LinkList)malloc(sizeof(Node));       //创建新结点
        p->data = rand() % 100 + 1;
        p->Next = (*L)->Next;
        (*L)->Next = p;
    }
}

//6.尾插法建立单链表
void CreateListTail(LinkList* L, int n)
{
    LinkList p, r;
    int i;
    srand(time(0));
    *L = (LinkList)malloc(sizeof(Node));          //创建头结点，用*L 记录位
置
    r = *L;    //r 永远指向最后一个有值结点

    for (i = 0; i < n; i++)
    {
        p = (LinkList)malloc(sizeof(Node));
        p->data = rand() % 100 + 1;
        r->Next = p;
        r = p;
    }
    r->Next = NULL;        //最后一个结点 Next 为空
}

//7.单链表的整表删除
bool ClearList(LinkList* L)
{
    LinkList p, q;
    p = (*L)->Next;  //p 指向 L 中第一个有值结点
    while (p)
    {
        q = p->Next;
        free(p);
        p = q;
    }
```

```
    (*L)->Next = NULL;  //保留头结点
    return 1;
}
```

//线性表的静态链表存储结构
/*为了无需指针，但是可以随意插入删除创建的数组
使用游标实现法，每个结点包含（数据+游标），游标即下一个元素的下标
通常把未使用的数组元素称为备用链表
下标为 0 的结点不存放数据，游标=备用链表的第一个结点的下标
下标为 n-1(即全数组的最后一个结点)也不存东西，游标为第一个存了东西的结点的下标，起头指针的作用
很明显，结点的顺序不是按着数组 012345 来存的，所以可以随便插入删除，数组中每个结点不论有没有存数值，其游标必然不为空，记录着下一个结点位置
最后一个有值结点的游标为 0
*/

//1.静态链表存储结构

#define MAXSIZE 1000
typedef struct //每个结点结构
{
ElemType data; //存储的数据
int cur; //游标
}Component,StaticLinkList[MAXSIZE]; //创建链表

```
//2.链表初始化
bool InitList(StaticLinkList space)
{
    int i;
    for (i = 0; i < MAXSIZE - 1; i++)            //一直到下标为i-2，即倒数第二个结点都更新游标为下一位，倒数第二个的游标为倒数第一个的下标（MAXSIZE-1）

        space[i].cur = i + 1;

    space[MAXSIZE - 1].cur = 0;   //目前是初始化，所以整个数组最后一个结点的游标为0
    return 1;
}

   //一个获得链表内有值结点个数的函数
int ListLength(StaticLinkList space)
{
    int j = 0;          //用于记载个数
    int i = space[MAXSIZE - 1].cur;   //i 为第一个有值结点的下标
    while (i)              //最后一个有值结点的游标为0
```

```
    {
        j++;
        i = space[i].cur;
    }
    return j;
}

//3.静态链表的插入
int Malloc_SLL(StaticLinkList space)            //获取第一个空闲分量下标的函数，顺手把space[0]的游标也改一下
{
    int i = space[0].cur;   //space[0]里存了第一个空闲分量的下标，我们直接获取
    //当此空闲分量后面还有空闲分量时
    if (space[0].cur)        //反正这个空闲分量很快就不空闲了，我们顺手就帮忙把space[0]里游标改了，改成做完插入操作后的第一个空闲分量的下标
        space[0].cur = space[i].cur;
                                    //也即目前空闲分量的下一个
    return i;
}
//本函数执行完后，就得到了一个被排除出备用链表体系之外的，下标为函数返回值的可怜结点

    //在第i个元素之前插入，即挤到第i个元素的位置
bool ListInsert(StaticLinkList L, int i, ElemType e)
{
    int j, k, l;

    k = MAXSIZE - 1;  //k为整个数组最后一个节点的下标
    if (i<1 || i>ListLength(L) + 1)   //i=Length+1时，表示要在最后插入
    {
        return 0;
    }
    j = Malloc_SLL(L);           //不仅得到第一个空闲分量的下标，还将它从备用链表中取了出来，备用链表失去它后再次回归完整
    if (j)                  //只要j不等于0
    {
        L[j].data = e;
        for (l = 1; l < i; l++)           //无法由第i个结点得到其上一个结点的下标，故需要再遍历得到第i-1个结点的位置
        {
            k = L[k].cur;       //******得到第i-1个结点的下标
        }
        L[j].cur = L[k].cur;
        L[k].cur = j;
        return 1;
    }
    return 0;
```

```
}

//4.静态链表删除操作
void Free_SLL(StaticLinkList space, int k)
{
    space[k].cur = space[0].cur;
    space[0].cur = k;
}

bool ListDelete(StaticLinkList L, int i)
{
    int j, k;
    if (i<1 || i>ListLength(L))
        return 0;
    k = MAXSIZE - 1;
    for (j = 1; j < i; j++)
    {
        k = L[k].cur;        //******得到第i-1 个结点的下标
    }
    j = L[k].cur;       //j 为第i 个结点，即要被删除的结点的下标
    L[k].cur = L[j].cur;        //将第i 个结点排除出有值链表体系

    Free_SLL(L, j);       //将第i 个结点放入备用链表体系
    return 1;
}
```

//循环链表
/单链表到了尾部就停止了向后链的操作，故而想要遍历链表必须从头结点开始，而且只能索引前驱结点，不能索引后继结点
将单链表中尾结点的next 由空指针改为指向头节点的指针，进而使整个单链表构成一个环，称之为单循环链表，简称循环链表
若为空链表，则head->next=head
用尾指针rear 指向终端结点，rear->next 为头结点地址，rear->next->next 为第一个结点地址/

```
//1.初始化
typedef struct CLinkList
{
    ElemType data;
    struct CLinkList* next;
}node;

//参数为存着链表第一个结点地址的地址
//*pNode，是第一个结点地址，作为我们在计算机存储区域找到这个链表的唯一的线，如果第一个结点的地址改了，而外部并未记录下来，我们就会无法找到链表
void ds_init(node** pNode)                  //pNode 指向整个链表的指针，pNod
```

```
e 里面存的是整个链表的地址，*pNode 指向链表第一个结点的指针
{                                                   //pNode=》*pNode=》第一个结点
   pNode 作为存着 头结点地址的地方 的地址 ，它里面存的地址，即*pNode，为头结
点地址
    int item;                                       //为什么要用二重指针呢，还是和上面
一样，你在函数里要更改指向头结点的指针，那当然就得用指向头节点指针的指针
    node* temp;
    node* target;
    printf("输入结点的值，输入 0 为结束");

    while (1)
    {
        scanf("%d", &item);
        fflush(stdin);          //清除缓冲区

        if (item == 0)
            return;
        if ((*pNode) == NULL)       //若 pNode 有名无体，即里面没有第一个结点
的地址，即*pNode 本来应该是第一个结点地址，现在*pNode 为空
        {
            *pNode = (node*)malloc(sizeof(struct CLinkList));          //
现在是使用 pNode 指针更改它里面存的头结点的地址，malloc 一个结点
            if (!(*pNode))
                exit(0);
            (*pNode)->data = item;              //*pNode 就是具体一个结点的
地址了
            (*pNode)->next = *pNode;
        }
        else
        {
            for (target = (*pNode); target->next != (*pNode); target = ta
rget->next);
            //找到最后一个结点的地址给 target
            temp = (node*)malloc(sizeof(struct CLinkList));        //先给
我们新结点分配个空间，然后把地址给 temp

            if (!temp)
                exit(0);
            //在最后一个结点即 target 之后以及第一个结点也即*pNode 之前插入一
个新结点，这个新结点所在地址为 temp
            temp->data = item;
            temp->next = *pNode;
            target->next = temp;
        }
    }
}

//2.插入  参数为存着链表的第一个结点的地址的地址，以及要求插入的位置 i
```

```
void ds_insert(node** pNode, int i)
{
    node* temp;
    node* target;           //target 是要存入地址的前一个结点的地址
    node* p;
    int item;
    int j = 1;

    printf("输入要插入的结点的值:");
    scanf("%d", &item);

    if (i == 1)        //每个新插入结点作为第一个结点，相当于插入最后一个结点
后面，故而target 就是最后一个结点的地址
    {
        temp = (node*)malloc(sizeof(CLinkList));
        if (!temp)
            exit(0);
        temp->data = item;

        for (target = (*pNode); target->next != (*pNode); target = targe
t->next);   //老规矩，找最后一个结点的位置
        temp->next = target->next;                //相当于temp->next=(*p
Node)   即使temp 的next 指向原来的第一个结点
        target->next = temp;            //最后一个人结点的next 指向temp
（现第一结点）
        *pNode = temp;                        //*****更新操作:  *pNode 永远是
第一个结点的地址
    }
    else           //正常随便插到哪个地方          则*pNode 无需更新
    {
        target = *pNode;
        for (; j < (i - 1); j++)
            target = target->next;

        temp = (node*)malloc(sizeof(CLinkList));
        if (!temp)
            exit(0);
        temp->data = item;

        p = target->next;
        target->next = temp;
        temp->next = p;
    }
}

//3.删除结点
void ds_delete(node** pNode, int i)
{
```

```
    node* temp;      //temp 是要删的结点的地址
    node* target;    //target 是要删的结点的前一个结点的地址
    int j = 1;

    if (i == 1)                 //如果删除的是第一个结点
    {
        for (target = (*pNode); target->next != (*pNode); target = targe
t->next);   //找最后一个结点的地址然后存 target
        temp = *pNode;
        *pNode = (*pNode)->next;          //*pNode 更新
        target->next = *pNode;
        free(temp);
    }
    else
    {
        target = *pNode;
        for (; j < (i - 1); j++)
            target = target->next;        //同插入操作时一样，找到要删的结点
的前一个结点的地址
        temp = target->next;
        target->next = temp->next;
        free(temp);
    }
}

//4.查找节点所在位置
int ds_search(node* pNode, ElemType elem)   //因为不需要对结点的地址进行更
改操作，故而无需传入二级指针 ，此时 pNode 为指向第一个结点的指针
{
    node* target;
    int i = 1;
    for (target = pNode; (target->data) != elem && (target->next)!=pNode;
 i++)         //若仅有一个结点 pNode，而正好存的是 elem，不用担心，&操作直接
左边为 0 不判断右边，跳出循环
        target = target->next;
    if (target->next == pNode&& (target->data) != elem)
        return 0;
    else
        return i;
}
//5.约瑟夫问题（隔 3 个死一个）p18 线性表 13
/*思路其实就是依据人数先创建一个循环链表，data 存那个人的编号，然后指针 p 从 1
开始，指针向后挪 3 尾，输出这个结点的 data，然后删除这个结点
  然后不断循环数数儿删除，直到 p->next=p,这就是剩下的最后一个人，再输出他的编
号就行*/

//有的时候会直接不用头指针，只用尾指针 rear 指向最后的结点，这样访问尾结点就不
需要遍历整个循环链表了，而是直接用 rear，而且头结点也可以直接用 rear->next,第
```

```
一个结点也可以用rear->next->next
//比如连接两个链表时，效率就会大大提高，例如：(假设AB为两个不为空的循环单链表)

LinkList Connect(LinkList A, LinkList B)          //不变头指针（即B的尾指针），故而不用二级指针       AB为两个循环链表尾结点的地址，作为AB链表的大门
{
    LinkList p = A->Next;
    A->Next = B->Next->Next;
    free(B->Next);  //释放B的空的头结点    A的空头结点作为新链表的头结点

    B->Next = p;
    return B;     //返回B的尾指针作为新链表的尾指针
}

//循环链表中还有特殊情况：不首尾相连的环，即尾结点与链上某个结点相连
//检测是否有环的方法：（不论是首尾环还是特殊环）
    //法一：比较步数
    /*设两个指针，1号一步一步往前走，2号在一号每往前走一步时就重新从头开始走到1号目前所在位置，对比两个指针所用步数
    * 当步数不同时可得有环
    * 因为cur2走的步数必然是正正经经顺12345走到那的步数，而cur1因为一直往前走，如果有环可能就绕回哪个结点了，但是此时步数一直在积累，肯定就大于正经走到那的cur2
    */
    int HasLoop1(LinkList L)
    {
        LinkList cur1=L;     //1号指针，一步步往前走那个
        int pos1 = 0;        //初始步数为0
        while (cur1)
        {
            LinkList cur2 = L;    //每次循环2号指针都要初始化到第一个节点位置
            int pos2 = 0;
            while (cur2)      //cur2开始走
            {
                if (cur2 == cur1)          //当cur2走到cur1所在位置
                {
                    if (pos1 == pos2)       //步数相等，未检测到环，cur1继续往前走，cur2重新开始从头走
                        break;
                    else
                    {
                        printf("环在第%d个位置", pos2);
                        return 1;
                    }
                }
                cur2 = cur2->Next;
```

```
            pos2++;
        }
        cur1 = cur1->Next;
        pos1++;
    }
    return 0;      //若没有环，则总有cur1变为NULL的时候
}

//法二：快慢指针
//p走一步，q走两步，但凡有环，pq总有相遇之时
int HasLoop2(LinkList L)
{
    int step1 = 0;
    int step2 = 0;
    LinkList p = L;
    LinkList q = L;          //q是一下走两步那个
    while (p != NULL && q != NULL && q->Next != NULL)
    {
        p = p->Next;
        if (q->Next != NULL)
            q = q->Next->Next;
        if (p == q)
            return 1;
    }
    return 0;
}

//求魔术牌顺序   p20 线性表15（黑桃13张问题）
//先使用单链表createlist，结点数为13，data全部初始化为0，最后加个尾结点next指向头结点使成为循环链表
    void Magician(LinkList head)
    {
        LinkList p;
        int j;
        int Countnumber = 2;

        p = head;
        p->data = 1;  //第一张牌肯定放第一个

        while (1)
        {
            for (j = 1; j <=Countnumber; j++)   //使指针p挪到每轮要放牌的位置上
            {
                p = p->Next;
                if (p->data != 0)
                {
                    j--;           //如果p转移到的结点已经有存放了，本次转移没有意义不算数，j--
```

```
            }
        }

        p->data = Countnumber;
        Countnumber++;
        if (Countnumber == 14)
            break;
    }
}
//拉丁方阵问题
/*n 个元素生成 n*n 的表格，每行每列都由 n 个元素组成且元素不重复
思路:
n 个元素全部存在长度为 n 的循环链表中，分别从不同结点为起始打印整个链表，每个算一行*/
```

//双向链表：能够双向寻址，具有前驱结点地址和后继节点地址
// 头结点没有前驱结点，除头结点之外所以结点形成闭环，头结点的 prior 指向第一个结点地址
//1.双向链表结构
typedef struct DualNode
{
ElemType data;
struct DualNode* prior; //前驱结点
struct DualNode* next; //后继结点
}DualNode,*DuLinkList; //DuLinkList 表示 指向 DualNode 类型的指针 的数据类型 的 typedef 出来的新名字
//空的双向循环链表，两个指针都是指向自己
//创建(一个一个往最后插那种) 有空的头结点 *L 是空的头结点的地址*
bool InitList(DuLinkList L,int n) //老番再看 L 还是二级指针
{
DualNode* p, * q;
int i;

```
    *L = (DuLinkList)malloc(sizeof(DualNode));
    if (!(*L))
        return 0;       //防止 malloc 失败

    (*L)->next = (*L)->prior = NULL;     //有头结点后第一次初始化
    p = (*L);

    for (i = 0; i < n; i++)
    {
        q = (DuLinkList)malloc(sizeof(DualNode));
        if (!q)return 0;
        q->data = 0;          //data 初始化为0 吧全
```

```
        q->prior = p;          //先只管q和前面p的连接，以及q后继结点和头结点的连接
        q->next = p->next;     //其实按着前面的设定来看，和q->next=NULL一个意思.....
        p->next = q;

        p = q;       //p永远是最后一个，永远往p后面插，q后面跟的永远是头结点；这样也能保证链表创建连续下去，别创着创着现在的结点地址丢了
    }
    p->next = (*L)->next;         //首尾相连（空的头结点不参与循环链表，只作为敲门砖使）
    (*L)->next->prior = p;

    return 1;
}



//2.插入操作
/*简写一下思路吧，就不打代码了
* 要把新建的结点new插到结点s后面
* new->next=s->next;
* new->prior=s;
* s->next->prior=new;
* s->next=new;
* 或者插到结点p前面
* new->next=p;
* new->prior=p->prior;
* p->prior->next=new;
* p->prior=new;
*/

//3.删除
/*删掉p
* p->prior->next=p->next;
* p->next->prior=p->prior;
* free(p);
*/

//4.p22 线性表17例题
//本质上就是把头指针根据用户输入的数进行挪动，然后从头指针开始输出
//用上面的create的函数创建一个单链表，然后用下面的函数进行一下头指针挪动，然后在主函数里直接用头指针遍历输出即可
void Caeser(DuLinkList* L, int i)
{
    *L = (*L)->next;   //把*L挪到第一个节点的位置，进入循环
    if (i > 0) {
        i--;           //从第一个结点正着跳，跳i-1次即可跳到第i个结点
```

```
        do
        {
            (*L) = (*L)->next;
        } while (--i);       //会循环 i 次
    }
    if (i < 0) {
        i = -i;    //从第一个结点负着跳，跳 i 次到倒数第 i 个结点
        do
        {
            (*L)=(*L)->prior;
        } while (--i);
    }
}            //如此之后*L 即为要输出的第一个结点的头结点的地址

//也能这么改，反正最后*L 是第一个需要输出的结点的头结点的地址，即要输出时是先*
L=(*L)->next 得到第一个要被输出的结点的地址
void Caeser(DuLinkList* L, int i)
{
    if (i > 0) {
        do
        {
            (*L) = (*L)->next;
        } while (--i);
    }
    if (i < 0) {
        (*L) = (*L)->next;    //头结点没有前驱结点，没法往前走进入循环
        i = -i;
        do
        {
            (*L) = (*L)->prior;
        } while (--i);
    }
}
```

//栈
//特点：后进先出的线性表，只在表尾进行删除和插入操作
/表尾称为栈的栈顶（top），表头称为栈的栈底（bottom）
栈的插入操作（push）称为进栈/入栈/压栈
栈的删除操作（pop）称为出栈
栈也分顺序存储与链式存储
最开始没有任何数据的栈称为空栈，top=bottom/

```
//（一）栈的顺序存储结构 （数组形式，地址连续）
typedef struct              //这整个 struct 就是一个栈，故而不像前面的 struc
t 作结点一样有 data 元素
{
```

```
    ElemType* top;     //指向栈顶的指针        尾指针        指向栈后面第一个空
位置
    ElemType* base;    //指向栈底的指针        头指针
    int stackSize;     //指示栈当前可用最大容量      人话：数组大小
}sqStack;

/*也有这么搞的，其实没啥差
* typedef struct
* {
*    ElemType data[MAXSIZE];        //栈本体数组
*    int top;                       //栈顶下标
*    int stackSize;
* }
*/

//1.创建一个栈
#define STACK_INIT_SIZE 100
void initStack(sqStack*s)          //s 就是一个指向栈结构的指针，可以调用栈
结构内的对象
{
    s->base = (ElemType*)malloc(STACK_INIT_SIZE * sizeof(ElemType));
     //s->base 就是数组的头指针，顺序栈本质就是一个数组
    if (!s->base)
        exit(0);
    s->top = s->base;
    s->stackSize = STACK_INIT_SIZE;
}

//2.入栈操作
#define SATCKINCREMENT 10

void Push(sqStack* s, ElemType e)
{
    //如果栈满，追加空间
    if (s->top - s->base >= s->stackSize)
    {
        s->base = (ElemType*)realloc(s->base, (s->stackSize + SATCKINCRE
MENT) * sizeof(ElemType));       //指针名=（数据类型*）realloc（要改变内存
大小的指针名，新的大小）。
        if (!s->base)
                              //新的大小一定要大于原来的大小，不然的话会导致数据
丢失!
            exit(0);

        s->top = s->base + s->stackSize;                    //栈尾地址更新
了，自然原来的栈顶地址也不做数了。
        s->stackSize = s->stackSize + SATCKINCREMENT;          //记录最大
容量增加
```

```
    }
    *(s->top) = e;
    s->top++;
}
//3.出栈操作     从栈顶取出数据，即从表尾取出数据
void Pop(sqStack* s, ElemType *e)
{
    if (s->top == s->base)
        return;
    *e = *--(s->top);
}

//4.清空栈   清空内存元素，但是数组空间不被销毁
void ClearStack(sqStack* s)
{
    s->top = s->base;
}

//5.销毁栈
void DestroyStack(sqStack* s)
{
    int i, len;
    len = s->stackSize;
    for (i = 0; i < len; i++) {
        free(s->base);
        s->base++;
    }
    s->base = s->top=NULL;
    s->stackSize = 0;
}

//6.计算栈的当前容量
int StackLen(sqStack s)       //不修改结构体s，故不需要使用指向s的指针
{
    return (s.top - s.base);
}

//（二）栈的链式存储结构
/*因为只是栈顶来做插入和删除操作，所以一般都是栈顶放在单链表头部，使栈顶指针相
当于单链表头指针
即top指针不是指向第一个空的，而是指向最后一个结点*/
//1.结构
typedef struct StackNode   //结点
{
    ElemType data;
    struct StackNode* next;
}StackNode,*LinkStackPrt;

typedef struct LinkStack   //栈体
```

```
{
    LinkStackPrt top;  //栈顶、头指针
    int count;    //计数
};

//2.进栈操作
bool Push(LinkStack* s, ElemType e)    //s，指向整个栈对象的指针
{
    LinkStackPrt p = (LinkStackPrt)malloc(sizeof(StackNode));     //新建
结点
    p->data = e;
    p->next = s->top;      //p->next 指向前一个元素
    s->top = p;
    s->count++;
    return 1;
}

//3.出栈操作
bool Pop(LinkStack* s, ElemType *e)
{
    LinkStackPrt p;
    if (StackEmpty(*s))   //函数判断栈是否为空
        return 0;
    *e = s->top->data;
    p = s->top;
    s->top = p->next;
    free(p);
    s->count--;
    return 1;
}
```

//插曲：关于使用户连续输入数字，并以空格隔开，输入#结束输入 ，输入时为字符串，但是存储时需为 int 这回事

```
 char c;
 char str[10]; //用于将连续输入的数字字符存储，然后一下转化成一个完整的多位 int
 scanf("%c", &c);
 int i = 0,num;
 while (c != '#') {
 while (isdigit(c) || c == '.') //isdigit 函数可以判断 c 是否是数字字符  包含在 ctype.h 里  以及如果输入为小数，则会有小数点
 {
 str[i++] = c;
 str[i] = '\0';
 scanf("%c", &c);
 if (c == ' ')
 num = atof(str); //将数字组成的字符串转化为 flout  包含在 stdlib.h 里
```

```
 }
 //
 //输入其他字符的相应情况处理
 //可以用 switch
 }
```

```
//中缀表达式转化为后缀表达式
/*要求客户输入正常计算式，我们能打印出对应的后缀表达式
* 操作：
* 建一个栈存运算符
* 输入数字直接打印，输入符号则存入栈中
* 左括号（也存入栈中
* 当即将存入的符号比栈顶符号优先级低或者相等时（如栈顶此时是*而即将存入的符号是+）则该栈顶符号与它下面所有符号出栈打印（到左括号为止），再把即将存入的符号存入
* 当即将存入的符号为右括号）时，检测与其匹配的挨得最近的栈内左括号，并且把括号内符号依次出栈打印，左括号也出栈*/

sqStack s;

initStack(&s);
scanf("%c", &c);
while (c != '#')
{
    while (c >= '0' && c <= '9')
    {
        printf("%c", c);
        scanf("%c", &c);            //只要输入的是数字字符就不断输出，这样就能形成多位的数字
        if (c < '0' || c > '9')
            printf(" ");
    }
    if (c == ')')
    {
        Pop(&s, &c);
        while (c != '(')      //只要没到（就不断打印  最后一次取出（，正好停止循环，而且（也不在栈内了
        {
            printf("%c", c);
            Pop(&s, &c);
        }
    }
    else if (c == '+' || c == '-')
    {
        if (!stackLen(s))
        {
            Push(&s, c);
        }       //如果栈是空的，那就直接入栈（因为不会存在下面有更高优先级的情
```

```
况）
        else        //如果栈不是空的，就得判断下面有没有优先级更高或相等的运
算符
        {
            do
            {
                Pop(&s, &e);
                if (e == '(')
                    Push(&s, e);
                else {            //由于是+-操作，下面除了（之外都是优先级和
它一样或者更高的，所以下面的只要不是（就输出
                    printf("%c", e);
                }
            } while (StackLen(s) && e != '(')
                push(&s, c);     //等下面没有能输出的了，在把它放进去
        }
    }
    else if (c == '*' || c == '/' || c == '(')
        Push(&s, c);

    scanf("%c", &c);
}
while (StackLen(s)) {      //防止最后栈内还有符号
    Pop(&s, &e);
    printf("%c", e);
}
```

//队列
//特点：只允许一端进行插入操作，另一端进行删除操作的线性表，与栈相反，队列崇尚先进先出
//队列同样分为顺序存储结构与链式存储结构
/从队头出队列，队尾入队列
栈一般用顺序表实现，而队列则一般用链表实现，我们称之为链队列/

```
//1.队列的链式存储结构
typedef struct QNode
{
    ElemType data;
    struct QNode* next;
}QNode, *QueuePtr;

typedef struct     //包含了头指针与尾指针的一个结构
{
    QueuePtr front, rear;    //队头指针指向链队列的头结点（真正的，空的那种
头结点），队尾指针指向终端节点
}LinkQueue;
```

```
//2.创建一个队列
bool initQueue(LinkQueue* q)
{
    q->front = q->rear = (QueuePtr)malloc(sizeof(QNode));   //创建头结点
    if (!q->front)
        exit(0);
    q->front->next = NULL;   //q->front 表示 q 结构中 front 对象存储的地址，即新建的头结点的地址，而结点 QNode 内部都具有 next 指针，头结点的 next 指针由于是初始化故为 NULL
}

//3.入队操作
void InsertQueue(LinkQueue* q, ElemType e)
{
    QueuePtr p;
    p = (QueuePtr)malloc(sizeof(QNode));       //创建新结点
    if (p == NULL)
        exit(0);
    p->data = e;
    p->next = NULL;
    q->rear->next = p;
    q->rear = p;
}

//4.出队操作
void DeleteQueue(LinkQueue* q, ElemType* e)
{
    QueuePtr p;
    if (q->front == q->rear)  //当只有一个头结点
        return;
    p = q->front->next;
    *e = p->data;
    q->front->next = p->next;      //即使只有一个头结点一个尾结点，正好头节点的 next 也会初始化为尾结点的 next 即 null
    if (q->rear == p)   //当只有一个头结点与一个尾结点，即头结点后面跟的就是尾结点时
        q->rear = q->front;
    free(p);
}

//5.销毁队列
void DestoryQueue(LinkQueue* q)
{
    while (q->front)
    {
        q->rear = q->front->next;
        free(q->front);
```

```
        q->front = q->rear;
    }
}
```

//队列如果用顺序结构存储的话，假设数组末尾当队尾插入元素，那么队头就是数组下标为 0 处，从队头出队，要么所有元素往前挪一格，要么就当队头往后挪了，即要么浪费时间要么浪费空间
//故而为了解决这个问题，较好的解决方法为使用循环队列，本质还是用数组，但是队头指针和队尾指针在数组上动，使数组看起来像是头尾相连的表一样
//其中，front 指向第一个有值结点，rear 指向第一个无值结点
/*使 front 和 rear 不断加一，超出地址范围则自动从头开始，就实现了数组的“循环”，我们可以采取取模运算处理：

- （rear+1）%QueeuSize
  - （front+1）%QueueSize

*/

```
//循环队列
//1.定义一个循环队列
#define MAXSIZE 100
typedef struct
{
    ElemType* base;     //内存分配基地址，也可以当作一个数组
    int front;
    int rear;
}cycleQueue;

//2.初始化循环队列
void initQueue(cycleQueue* q)
{
    q->base = (ElemType*)malloc(MAXSIZE * sizeof(ElemType));
    if (!q->base)
        exit(0);
    q->front = q->rear = 0;
}

//3.入队列操作
void InsertQueue(cycleQueue* q, ElemType e)
{
    if((q->rear+1)%MAXSIZE==q->front )   //当队列已满
        return;
    q->base[q->rear] = e;
    q->rear = (q->rear + 1) % MAXSIZE;
}

//4.出队列操作
void DeleteQueue(cycleQueue* q, ElemType *e)
```

```
{
    if ((q->rear + 1) % MAXSIZE == q->front)   //当队列已满
        return;
    *e = q->base[q->front];
    q->front = (q->front + 1) % MAXSIZE;
}
```

//递归
//迭代与递归的区别在于迭代是通过循环来控制，递归是通过选择来控制。一般递归中会有一个选择来作为突破口，当传入参数符合这个条件，就开始往回 return 数据
//斐波那契数列

```
 int Fib(int i)
 {
 if (i == 0)
 return 0;
 else if (i == 1)
 return 1;
 return Fib(i-1) + Fib(i - 2);
 }
```

//同时也有一个特点就在于，递归会不断扩大空间，故而在每个分配的空间内参数都存在里面，不会随着其内嵌套的自己函数的执行而删除
//例：输入未知长度字符串，然后倒着输出（输入以#为结束）

```
 char print() //利用了递归调用的顺序与回退的顺序是相反的这个特性
 {
 char c; //每一轮输入的 c 都保存在对应的空间，没有丢失，直到有返回来的，才开始往下执行，然后才会被删除
 scanf("%c", &c);
 if (c != '#')print();
 if(c!='#')printf("%c",c)
 }//上例也告诉我们不一定调用就是要返回值，也可以是单纯开出空间，不追求让一层一层返回来值，也可能只是为了让函数空间保存且暂停
```

```
//折半查找法：不断缩小一半的查找范围
//迭代实现：
int cuthalf(int low, int high, int num, int* a) {
    if (low <= high) {
        if (num < a[(low + high) / 2])return cuthalf(low, (low + high) / 2 - 1, num, a);
        else if (num == a[(low + high) / 2])return (low + high) / 2;
        else if (num > a[(low + high) / 2])return cuthalf((low + high) / 2 + 1, high, num, a);
    }
    else {
        return -1;
```

```
    }
}

//汉诺塔移动问题
void move(int n, char x, char y, char z) {   //(n,x,y,z)的意思是是将n 个片片从x 借助y 移动到z 上
    if (n == 1) {
        printf("%c-->%c", x, z);
    }
    else {
        move(n - 1, x, z, y);
        printf("%c-->%c", x, z);
        move(n - 1, y, x, z);
    }
}

//八皇后问题
//8*8 的棋盘上摆放八个皇后，皇后与皇后不能处于同一行，同一列，同一斜线上，求有几种摆法
int numm = 1;
int chess[8][8] = { 0 };

bool notDanger(int row, int j, int(*chess)[8]) {  //以判断在传入的chess上坐标为（row,j）的点横竖斜线是否有皇后为目的
    int i,k,f1=0,f2=0;
    //同列上是否有皇后
    for (i = 0; i < 8; i++)
    {
        if (chess[i][j] == 1) {
            f1 = 1;
            break;
        }
    }
    //同行上是否有皇后
    for (i = 0; i < 8; i++)
    {
        if (chess[row][i] == 1) {
            f1 = 1;
            break;
        }
    }
    //左上
    for (i = row, k = j; i >= 0&& k >= 0; i--, k--)
    {
        if (chess[i][k] == 1) {
            f2 = 1;
            break;
        }
    }
```

```
    //右上
    for (i = row, k = j; i >= 0&&k<8; i--,k++)
    {
        if (chess[i][k] == 1) {
            f2 = 1;
            break;
        }
    }
    //左下
    for (i = row, k = j; i < 8 && k >= 0; i++, k--)
    {
        if (chess[i][k] == 1) {
            f2 = 1;
            break;
        }
    }
    //右下
    for (i = row, k = j; k < 8 && i <8; k++, i++)
    {
        if (chess[i][k] == 1) {
            f2 = 1;
            break;
        }
    }
        if (f1 == 1 || f2 == 1)return 0;
        else return 1;
}

//此递归函数的操作的范围是每行，递归行为则是以“下一行”为线索，来递归下一行下一行，在递归结束时得到所有行总和即一个完整的棋盘
    //以行为单位进行递归，而对每列的确定是在函数内通过 for 循环针对每列再次使用 eightqueen 函数实现的，对每行的确定是在调用函数时输入的
    void EightQueen(int row, int n, int(*chess)[8]) {   //chess 指向行的指针，每行 8 列；row 表示起始行；n 表示列数
        int chess2[8][8], i, j;
        for (i = 0; i < 8; i++) {       //新建一个数组，并且先复制传入的数组上的数据，既能保证在函数内在数组上操作，又可以使传入函数的数组维持原状，来支撑被下一个函数调用
            for (j = 0; j < 8; j++) {
                chess2[i][j] = chess[i][j];
            }
        }
        //开始输出
        if (row == 8) //row 从 0 开始代表行的下标的，当 row 为 8 时，代表到了第九行，即前八行棋盘（整个棋盘）已经全部下完，可以开始输出
        {
            printf("第%d 种方法\n", numm++);
            for (i = 0; i < 8; i++) {
```

```
                for (j = 0; j < 8; j++) {
                    printf("%d ", chess2[i][j]);    //*(*(chess2+i)+j)
                }
                printf("\n");
            }
            printf("\n");
        }
        else {
            for (j = 0; j < 8; j++) {   //遍历第row行的各列      对于每一个能下皇后的位置都会遍历到，然后每一个位置都会各自往下延申出递归递归递归直到最后一行
                if (notDanger(row, j, chess))
                {
                    for (i = 0; i < 8; i++)
                    {   //先把row行上一次下棋留下来的痕迹全部清零
                        chess2[row][i] = 0;
                    }
                    chess2[row][j] = 1;   //把检测到的这个row行的可下位置下皇后
                    EightQueen(row + 1, n, chess2);      //在下一行找第一个能下皇后的位置
                }
            }
        }
    }
```

//字符串
/*字符串的存储结构与线性表相同，也分顺序存储结构与链式存储结构。
字符串的顺序存储结构是用一组地址连续的存储单元来存储串中的字符序列的
按照预定义的大小，为每个定义的字符串变量分配一个固定长度的存储区，一般用定长数组来定义
与线性表相似，既然是固定长度的存储区，就存在一个空间分配不灵活的问题，就会考虑用链式存储结构
但由于字符串一般都是一整句，断章取义的情况并不常见，故平常还是用顺序存储结构较多
*/
//BF 算法 野兽暴力算法
//核心思想：连个字符串 S T，先比 S[1]T[1],相等就继续比较 S[2]T[2]，不等就 s 向右挪一位再比，即比较 S[2]T[1]，当比到半中间发现 s[4]t[3]不等了，就返回去再比较 s[3]t[1]，即 s 的每一位都要与 t[1]匹配一下而且有可能往后比对很多位发现有对不上的，就返回去找下一个 s[]与 t[1]重开始匹配 即不断回溯
//其中 S 为主串，T 为子串（如 assfa 是 assfadddse 的子串），这种子串的定位操作我们称之为串的模式匹配
//意思就是 BF 算法的目的就是为了从主串中找到子串的位置...

```
//KMP 算法
//目的：为了避免不必要的回溯
//核心：问题由模式匹配决定，不是由目标决定（模式匹配指的是模式匹配串即子串）
/*算法实现：
基于子串建一个 k 数组，在 k 中与子串中元素对应下标的位置存放“若在此处匹配失败，则将字符串中下标为几的字符移到断处继续匹配”
而 k 中填写是基于对子串中每个字符的前后缀集中对应前后缀进行比较
从大的前后缀开始比，到找到相等的前后缀，根据此前/后缀的长度来填写（其实也就是直接填进去前缀的后一个位置的下标）即填入（前缀长度+1）

而且因为是从最长的前后缀到逐渐短的前后缀进行比较，就是在一点点审核每一种前移代替可能，一旦有一个不同，代表无法代替，则再进行下一次判断

                                （因为在断点之前，母串与子串是一一对应完全一样的，故可以用后缀代指母串断点前的字符串与子串进行匹配）
后缀与前缀进行对比，后缀就相当于被比对的长字符串在比对中断处之前的字符，分别从长到短与前缀，即短字符的开头一段比较，进而锁定能够确定的已匹配的字符
例如，开头 3 个与中断前 3 个重叠了，自然会直接将短字符开头三个与长字符中断前 3 个字符上下对齐，然后再继续往后一一比较*/

//先写一个与子串元素一一对应的 next 数组，以及子串格式需要修改，下标 0 存放字串元素数，下标 1 存放第一个元素。next 数组内下标 1 对应子串数组下标 1 存放字符对应的转移匹配的下标
  void get_next(string T, int* next)
  {
      int j = 0, i = 1;                    //j 为此时比对的前缀下标，i 为此时比对的后缀下标
      //这里他用了一个小技巧，从开头比对，此时前缀后缀长度只能为 1，且相差唯一，即往后走的话，前缀后缀都是当做最长的来进行比对，而且直接利用了之前比对的结果，比如比到了 5 之前，那即 123 与 234 都一轮一轮比下来了，就当作专门为了求 5 对应 next 比的，但实际上是边比边填，但是前面的比对也是为后面的比对保留了过程
      next[1] = 0;
      while (i < T[0]) //只要后缀没有比到 T 串的最后面，就没有结束
      {
          if (j == 0 || T[j] == T[i])      //此时 i 作后缀，用于参与判断 i+1 位置上的元素对应的 next 标
          {
              j++;
              i++;                         //此时 j i 都是新的，等下一轮循环来比较
              if (T[i] != T[j])            //此时 i 与 j 不是为了填 next 而比较，ij 没++之前就是为了 目前这个 i 对应的元素的 next 值而进行比较，此时比较仍是为了 i 这个下标对应的元素的 next 值，如果此时比对不成功这个元素和它对应 next 值的对应元素一样，那不肯定也不匹配吗，那它的 next 就不能按着常规写了，就得再找 next 的 next
              {//正常情况，即断点处子串元素与它 next 对应元素不同
```

```
                next[i] = j;            //刚更新到的，还没进行比较的后缀元素的next 下标仅与此元素前面的元素有关，它对应的应该是与它前面元素相等的元素们的下一个
            }
            else            //断点处子串元素与它的next 对应元素相同
            {
                next[i] = next[j];      //他的next 填上next 的next（而这个next 的next 肯定在之前写的时候也是填的next 的next（不是从前往后填next 的嘛，前面的a1 与a0 一样，它next 填了1，之后和a1 一样的a2 填a1 的next，肯定也是1，再之后一样的填都是1），故最后一定是指向第一个一样的元素）
            }
        }
        else
        {
            j = next[i];      //没有执行if 语句中的i++，此时i 对应的元素已经有了对应的next 标（根据上一轮i-1 填了）即此时i 作当前在做对比的元素时，已经知道了如果i 处比对不符，该挪到哪重新开始匹配
        }                     //但是i 作为后缀，它和前缀不符了，则需要把前缀挪到能和此时后缀比对上的地方，（为什么挪前缀不是挪后缀？因为后缀就模拟母串，前缀模拟子串，你说比对时候是挪子串进行比对还是挪母串进行比对？）
                              //而已知匹配在i 处中断，在i 处匹配中断则子串挪到next 处再重开比对的next 数值已知为next[i]，前缀（代子串）应该挪到哪呢？废话，当然是next[i]处
    }
}

//开始查找   返回子串t 在母串s 第pos 个字符之后的位置(即规定了在pos 之后找有没有与子串相同的)，若不存在，则返回0
int index_KMP(string s, string t, int pos)
{
    int i = pos;
    int j = 1;
    int next[255];
    get_next(t, next);
    while (j <= t[0] && i <= s[0])
    {
        if (j==0||s[i] == t[j])
        {
            i++;
            j++;
        }
        else
        {
            j = next[j];        //j 是子串的下标，next 中存了在此下标处比对失败后要把子串的哪个下标处挪到断点处重新一轮比对
        }
    }
```

```
    if (j > t[0])      //子串全部比对完成,此时 i 在比对完成后的第一个位置
        return i - t[0];              //得到子母串开始相等的第一个位置
    else          //子串都没遍历完，结果跳出来了，肯定是母串已经遍历完了，
那就是没比对成功
        return 0;

  }
```

//树
//树是 n>=0 的结点的有限集，当 n=0 时，称之为空树，有且仅有一个特定为根的结点；n>1 时，其余结点又可分为 m 个不可相交的有限集，其中每一个集合本身又是一棵树，称为根的子树
//树的根节点是唯一的，子树个数没有限制，但子树之间一定不可能相交。
//出度又称为度，结点的度即为结点的出度，树的度即为树中度的最大值
//指向此节点的个数称为入度，从此节点伸出的称为出度。出度为零的称为根结点，出度不为 0 的称为分支结点，其中除根结点外其他结点也叫内部结点
//结点出度指向的称为结点的孩子（child），相应的，该结点称为孩子的双亲（parent），同一双亲的孩子结点互称为兄弟（sibling），从根到该结点所经路上所有的结点称为该结点的祖先（包括 parent）
//结点的层次从根开始称为第一层，根的孩子称为第二层；双亲在同一层的结点互称为堂兄弟；树中结点的最大层次称为树的深度（depth）或高度
//若将树中结点各子树看作从左至右有次序，不能互换的，则称该树为有序树，否则称为无序树。
//森林为 m 棵互不相交的树的集合。而对树中每个结点而言，其子树的集合即为森林
 //树的存储结构有三种不同表示法：双亲表示法，孩子表示法，孩子兄弟表示法
 //（一）双亲表示法
 //以双亲为索引关键词，每个结点知道它自己是谁，同时知道它双亲是谁
 //定义如下
#define MAX_TREE_SIZE 100
 typedef struct PTNode
 {
 ElemType data;
 int parent; //双亲位置，存了双亲的下标，因为是存储在一个数组里，不是动态分配内存，不需要指针；根的 parent=-1
 //改进
 //int child1,child2; //存储孩子位置，使可以直接查找孩子，不用遍历数组才能查找孩子是谁
 //int rightsib; //存储兄弟的位置，右边没有兄弟就填-1
 }PTNode;
 typedef struct
 {
 PTNode nodes[MAX_TREE_SIZE];

int r; //根的位置
int n; //结点数目
};

```
//（二）孩子表示法
//每个结点的孩子个数是不确定的，针对此问题，我们进行了如下思想上的演化：
/*每个结点都分配树的度（最大度）个位置存放子结点->空间浪费
* 升级->通过数组与链表的结合。数组存本结点data与指向他的孩子下标链表的头指针，它的孩子的下标存放在链表中
* 再升级->双亲孩子表示法：数组每个单元内存data，双亲的下标，指向孩子链表的头指针
*/
//双亲孩子表示法定义如下：
typedef struct CTNode        //构造每个结点对应的链表中的孩子结点
{
    int  Child;      //孩子下标
    struct CTNode* next;        //指向下一个存储孩子下标的结点
}*Childptr;

typedef struct     //构造结点
{
    ElemType data;
    int parent;
    Childptr firstchild;
}CTBox;

typedef struct   //存放结点的数组
{
    CTBox nodes[MAX_TREE_SIZE];
    int r, n;
};

//二叉树
//二叉树或为空树仅一个结点，或由一个根节点，与两颗互不相交，分别被称为根节点的左子树与右子树的二叉树组成
//每个结点出度必须小于等于2，且左右子树不可调换位置；即使此结点只有一棵子树，也需要区分它是左子树还是右子树，仅有一个左子树与仅有一个右子树是不同的。比如三个结点的树，若为二叉树，则有5种情况
//特殊二叉树
    //1. 斜树：子树全在一边
    //2. 满二叉树：每个结点出度都为2，且叶子只能出现在最下面一层
    //3. 完全二叉树：从上至下对每层结点从左到右编号（中间跳了就直接跳了继续编，忽略本来应该有节点的位置），其编号与满二叉树编号一一对应    （它的叶结点只可能出现在倒数1 2层）
    //满二叉树一定是完全二叉树，但完全二叉树不一定是满二叉树
//二叉树的性质：
/*1.第i层最多有2^(i-1)个结点
```

```
 * 2.深度为k 的二叉树至多有2^k-1 个结点
 * 3.对任意一棵二叉树t，若其终端结点数为n0，度为2 的节点数为n2，则n0=n2+1

 * 4.具有n 个结点的完全二叉树深度为（log2 n）+1，满二叉树的深度为log2（n+1）
 * 5.对一棵有n 个结点的完全二叉树的结点按层序编号，对任一结点i 有以下性质：
 *     i=1，则此结点为根，没有双亲；i>1，则此结点双亲为[i/2]（取整）
 *     2i>n，则结点i 为叶结点，无左孩子；否则其左孩子为2i
 *     2i+1>n，则结点i 无右孩子；否则其右孩子为结点2i+1
 *     总结一下，结点i（i 不等于1），双亲必为[i/2]，左孩子必为2i，右孩子必为2i+1，2i 或2i+1>n 则没有左孩子或右孩子，而没有左孩子则必没有右孩子，故2i>n 则结点i 无孩子即为叶结点
 */

//二叉树的存储结构
/*（一）顺序存储结构，即用一维数组存储结点，并且结点的存储位置可以体现结点之间的逻辑关系
 *       即使是非完全二叉树，也可以用^来表示无结点位置；但是如果缺的结点很多，但为了使数组能根据存储结构反应逻辑关系，就会空很多位置，空间利用率过低，故而推荐链式存储结构
 * （二）链式存储结构，每个结点设计一个数据域与两个指针域
 *                    *LChild   data   *RChild
 *                   我们称这样的链表为二叉链表
 */
        typedef struct BiTNode
        {
            ElemType data;
            struct BiTNode* lchild, * rchild;

        }BiTNode,*BiTree;

//二叉树的遍历
/*从根结点出发，按照某种次序依次访问二叉树中所有结点，使得每个结点被访问一次且仅被访问一次
遍历方式被分为以下四种：前序遍历，中序遍历，后序遍历，层序遍历（一层一层从左到右）

1.前序遍历：若二叉树为空，则空操作返回，若不为空，则先访问根结点，后左子树（左子树又可以作为一棵完整树，先访问此树的根，再访问此树的左子树..），左子树全访问完了，访问右子树
            过程中若遇到某棵可爱小树没有左子树，则按序顺推先根后右
2.中序遍历：左子树 根 右子树
3.后序遍历：左子树 右子树 根
*/

//二叉树的创建(用户遵照前序遍历输入数据)
```

```
void CreateBiTree(BiTree* T)       //创建结点需要赋予结点地址，故要用存有节点地址的地址来更改
{
    char c;
    cin>>c;    //char 类型只能存取一个字符，输入一串字符时挨个儿读取,故你输入一串儿，它读完一个开始判断，然后再进入 createbitree（lchild）函数，再往下读一个 char
    if (' ' == c)  //此结点不存在
    {
        *T = NULL;
    }
    else       //存在结点，则开始创建一个 bitnode 对象，并编写内容
    {
        *T = (BiTNode*)malloc(sizeof(BiTNode));        //先建根
        (*T)->data = c;
        CreateBiTree(&((*T)->lchild));                 //后建左
        CreateBiTree(&((*T)->rchild));                 //后建右
    }
}
//上面这个创建一定要记住按着前序遍历输入，没有结点的地方也需要输入‘ ’

void visit(BiTree T) {
    cout << T->data;
    //...

}
//遍历二叉树(中序)    想要别的顺序，到时候改一下函数内部顺序就行
void PreOrderTraverse(BiTree T)      //遍历不修改，故传入结点地址即可
{
    if (T) //防止二叉树为空
    {
        PreOrderTraverse(T->lchild);
        visit(T);          //遍历时对每个结点的操作，比如输出啥啊之类的
        PreOrderTraverse(T->rchild);
    }
}

//但即使是二叉链表，仍存在空间浪费的情况，如某个结点因为没有子结点而将两个 lchild rchild 结点置为空
//故而我们想用可被利用的‘空着的 lchild 和 rchild’记录该结点的前驱和后继（遍历过程中的前驱和后继，会随遍历的方法改变而改变）
//区别于存放指向左右子树的指针，在本应空着的结构中存放的 指向遍历过程中此结点前驱与后继的指针 称为存放线索。
//于是我们选择了一种更浪费空间的结构:
//                lchild ltag data rtag rchild
//没错，一个结点，存它的左右子树的地址/前驱后继的地址，再存它两边存的是地址还是线索的判断‘tag’
```

```
  //ltag 为0 时 lchild 指向该结点的左孩子，为1 时 lchild 指向该结点的前驱；rta
g 同理，为1 时 rchild 指向该结点的后继



  //线索二叉树创建
  typedef struct BiThrNode
  {
      ElemType data;
      struct BiThrNode* lchild,*rchild;
      int ltag,rtag;
  }BiThrNode,*BiThrTree;

  void CreateBiThrTree(BiThrTree* T)    //此时创建出来的二叉树只要是存在的
结点，ltag rtag 都等于0，左右存的要么是指针要么是空，还没有线索化
  {
      char c;
      cin >> c;
      if (' ' == c)
      {
          *T = NULL;
      }
      else
      {
          *T = (BiThrNode*)malloc(sizeof(BiThrNode));
          (*T)->data = c;
          (*T)->ltag = 0;
          (*T)->rtag = 0;
          CreateBiThrTree(&((*T)->lchild));
          CreateBiThrTree(&((*T)->rchild));
      }
  }

  BiThrTree pre;    //全局变量，始终指向访问的前一个结点

  void InThreading(BiThrTree T)         //中序遍历，使刚刚创建的二叉树线索
化,只需要改变结点中的东西，不需要改变结点的地址，故仅传入指向结点的指针
  {
      if (T)           //判断这个存储在上个结点的 lchild 或 rchild 里的地址 T
是不是空的，不是空的才能继续下一步处理
      {
          InThreading(T->lchild);   //递归左孩子线索化            中序遍历先
左

          //对根结点的处理                                          中序遍历后中
          if (!T->lchild)          //如果没有左孩子，他的 lchild 就存的是线
索，他的 ltag 就得是 1
          {
              T->ltag = 1;      //表明左边存的是前驱不是孩子
```

```
            T->lchild = pre;  //lchild 指向它的前驱节点
        }
        if (!pre->rchild)     //如果上一个结点（即先在 T 的前驱结点）没有右孩子，则正好给这个前驱节点的 rchild 填上 T
        {
            pre->rtag = 1;
            pre->rchild = T;
        }
        pre = T;
        InThreading(T->rchild);   //递归右孩子线索化            中序遍历后右
    }
}
void createpre(BiThrTree* p, BiThrTree T)
{
    *p = (BiThrTree)malloc(sizeof(BiThrNode));
    (*p)->ltag = 0;         //左指针指向根节点，相当于他的孩子
    (*p)->rtag = 1;         //右指针指向树根据中序遍历的最后一个结点，相当于他的前驱；之后使这个结点后继（rchild）填 pre 指针，就能使整个遍历过程形成一个环
    if (!T)       //若 t 为空树，pre 直接指向自己吧，甭白费功夫
    {
        (*p)->lchild = *p;
    }
    else
    {
        (*p)->lchild = T;
        pre = *p;         //pre 指针初始化为指向树 T 的根
        InThreading(T);    //有了 pre，开始线索化 T
        pre->rchild = *p;   //经过 inthreading 线索化，pre 已经变成了指向最后一个结点的指针，而 p 此时作为函数内变量，仍表示指向树的根的头指针
        pre->rtag = 1;      //最后一个结点的 rtag 为 1，预示着它也有了后继结点，就是，当当当当，头结点！
        (*p)->rchild = pre;
    }
}

void visit(BiThrTree T) {
    cout << T->data;
}
//建好以后，中序遍历二叉树，使用迭代方法
void travelBiThrTree(BiThrTree T)       //这个 t 可不是根结点，而是指向根结点的头结点，独立于树之外的那个
{
    BiThrTree p;   //p 是用于遍历树的指针，而 T 是固定不动的根结点地址
    p = T->lchild;   //p 初始为根结点地址
    while (p!=T)        //当 p 遍历到最后一个结点，而最后一个节点的 lchild
```

```
指向头结点（不是根结点），此时 p=T 即等于头结点，遍历结束
    {
        while (p->ltag == 0)        //将 p 指向最左边那个，即遍历的第一个结点
        {
            p = p->lchild;
        }
        visit(p);    //访问
        while (p->rtag == 1 && p->rchild != T)      //p 的右边放了后继地址且不是头结点地址（即存了下一个可遍历地址的结点，也不是指回头结点）
        {
            p = p->rchild;
            visit(p);
        }                           //执行完后 p 必然是右边存着后继的结点了
        p = p->rchild;     //访问叶结点的后继结点
    }
}
```

//树，森林二叉树的转换
//树变二叉树：所有兄弟之间加一条线，再去掉每个结点与除左儿子之外儿子的连线。变更根。
//森林变二叉树：先将各树根据上法变二叉树，再将得到的各二叉树的根从左至右连起来
//二叉树变树和森林：若有结点 x 是其双亲 y 的左孩子，则将 x 的右孩子，右孩子的右孩子..都与 y 连起来；再去除掉所有双亲与右孩子之间的连线（不包括刚刚画上去的）

//树与森林的遍历
//树的遍历：先根遍历和后根遍历
//先根：先遍历根后遍历其子树；后跟：先子树后根
//森林的遍历：前序遍历和后序遍历
//前序：以先根的方式遍历所有树
//树与森林的先根（前序）遍历与二叉树前序遍历结果相同；后根遍历与二叉树中序遍历结果相同

//哈夫曼树
/*叶子结点带权
结点的路径长度：从根结点到该结点的路径上的连线数（跟到这个结点走了几根线）
树的路径长度：树中每个叶子结点的路径长度之和（整棵树有几根线）
结点带权路径长度：结点的权值与结点的路径长度的乘积
树的带权路径长度（WPL）：树中所有叶子结点的带权路径长度之和
WPL 值越小，说明构造出的二叉树性能越优

构造哈夫曼树过程：
给你一堆带权结点，先从结点中选出两个权值最小的，其中最小的在左边，大一点的在右边，为他俩创造一个根结点，权值为他俩之和，然后再在被创造出来的根与其他结点中继续选权值最小的两个，再建根…*/

//哈夫曼编码
/*定长编码：码的长度都一样
变长编码：单个编码长度不一致，根据整体出现频率调节
前缀码：不是其他码的前缀（其他码没有前缀与之相等的）

哈夫曼编码：给每条连线赋 1 或 0，左 0 右 1，对于某个结点，从根到该结点的线路上的 1 和 0 组合起来（从根到结点=从左到右），即为这个结点的哈夫曼编码
结点代表元素出现次数越多，离哈夫曼树的根越近，哈夫曼编码越短。
编码过程：
1.创建一个队列，使所有结点依照权值从小到大排列。
2.创建哈夫曼树：从左到右取队列中两个结点，相加得新结点，新结点指向这两个结点，再根据新结点权值插入队列中，再从左到右取两个结点…
3.创建哈夫曼表：用于存放每个结点的哈夫曼编码
3.encode：打印编码
4.decode（解码）：通过哈夫曼编码寻找它对应的结点（0 就往左找一步，1 就往右跨一步），找到结点后解码的过程/

/*比如以下为编码字符串的过程

- 会把字符串中每个字符出现次数统计一下，根据每个字符出现次数来为每个字符编出它对应的哈夫曼编码。
  编码长度不一，编出的哈夫曼树由根结点到叶结点，自上而下出现次数由高到低，哈夫曼编码长度由短到长
  而编码的过程使用的就是哈夫曼编码法，即建队列（按出现次数由低到高排），建树（队列辅助建树实现哈夫曼建树法，使用哈夫曼法建树的后果就是：只有叶结点是代表字符的结点，其他结点都是为了建树新建的），用树建编码表（链表存储，每个已出现过的字符有且只有一个对应的编码，出现次数越多编码长度越短，（短了）主要是便于使用，毕竟要用很多次，每次都那么长谁也吃不消）
  得到每个字符的编码后就可以通过使用此码指引（010：往左往右往左）遍历刚建的哈夫曼树，来一个一个找到对应的字符结点然后输出
  由于是根据每个字符在给出字符串中出现次数生成的哈夫曼编码，故而字符串改变，两字符串中相同的字符的编码也可能不同
  **对每个字符的编码过程与每个字符在字符串中出现顺序毫无关系！编码仅与字符出现次数有关！

  - 非要扯和字符串中字符顺序有啥关系，也就输出字符串的编码的时候，是根据字符串从左到右一个一个输出字符对应编码的（所以输出

编码时如果有字符出现很多次，他的编码自然也会输出好多次咯，他的码在整体字符串的码中的位置就根据他字符本身在字符串中位置定咯）
*/

//以下为哈夫曼编码字符串的过程
/*使每个字符在树中，然后输出字符串则通过输出哈夫曼树每个结点得到（即解码）*/

```
//头文件
typedef struct _htNode {    //哈夫曼树的结点的结构
    char symbol;
    struct _htNode* left, * right;
}htNode;
typedef struct _htTree {       //哈夫曼树的头指针
    htNode* root;              //内存哈夫曼树的根结点地址
}htTree;
typedef struct _hlNode {
    char symbol;
    char* code;
    struct _hlNode* next;
}hlNode;
typedef struct _hlTable {     //用于指示哈夫曼编码表第一个结点地址与最后一个结点地址
    hlNode* first;
    hlNode* last;
}hlTable;

typedef struct _pQueueNode {       //队列的结点结构(这个队列是个链表)
    htNode* val;        //指向对应的树结点的指针
    unsigned int priority;  //优先级 unsign 表示必>0
    struct _pQueueNode* next;   //指向下一个结点的指针
}pQueueNode;

typedef struct _pQueue {       //指向队列的头指针的结构
    unsigned int size;         //队列的当前长度
    struct _pQueueNode* first;        //指向队列的头指针
}pQueue;

void initpQueue(pQueue** queue) //队列头指针的初始化
{
    (*queue) = (pQueue*)malloc(sizeof(pQueue));
    (*queue)->first = NULL;
    (*queue)->size = 0;
};

void addPQueue(pQueue** queue, htNode* val, unsigned int priority) {
//在队列中插入结点   （队列结点排序依照结点优先级从小到大排）
```

```
    if ((*queue)->size == 256) {       //撑死了256个字符，那队列就算每个字符都有，撑死了不也就长256吗
        printf("queue is full!");
        return ;
    }
    pQueueNode* aux = (pQueueNode*)malloc(sizeof(pQueueNode));     //建立对应的队列中结点
    aux->priority = priority;
    aux->val = val;

    if ((*queue)->size == 0 || (*queue)->first == NULL) {  //如果队列目前为空
        aux->next = NULL;
        (*queue)->first = aux;
        (*queue)->size++;        //队列长度加一
        return;
    }
    else {
        if (priority <= (*queue)->first->priority) {     //如果新结点的优先级比列表中第一个结点优先级还小
            aux->next = (*queue)->first;
            (*queue)->first = aux;
            (*queue)->size++;
            return;
        }
        else
        {
            pQueueNode* iterator = (*queue)->first;         //iterator指向列表第一个结点,作为遍历列表用的指针(iteratoe 迭代器)
            while (iterator->next != NULL) {
                if (priority <= iterator->next->priority) {        //若找到新结点的位置----iterator 与 iteratoe->next 中间
                    aux->next = iterator->next;
                    iterator->next = aux;
                    (*queue)->size++;
                    return;
                }
                iterator = iterator->next;        //没找的就继续找
            }
            if (iterator->next == NULL) {       //遍历完了但是没有return才会执行到这==》新结点优先级比列表里都大（故而没有满足上面遍历中 if 函数的）
                iterator->next = aux;
                aux->next = NULL;
                (*queue)->size++;
                return;
            }
        }
    }
```

```
};

htNode* popQueue(pQueue** queue)      //从队列中取出目前队列的第一个结点建树
{
    htNode* returnvalue;

    if ((*queue)->size > 0)
    {
        returnvalue = (*queue)->first->val;
        (*queue)->first = (*queue)->first->next;
        (*queue)->size--;
    }
    else
    {
        printf("队列为空无法取结点");
    }
    return returnvalue;
}

void traverseTree(htNode* treeNode, hlTable** table, int k, char code[256])   //treeNode 是树的根结点，code 作为不断更新的存储着路上的编码的数组（每一次递归会复制新建一个 code，不要担心又批了）
{
    if (treeNode->left == NULL && treeNode->right == NULL)    //只有当一个结点没有左右子树时，它才配有哈夫曼码
    {
        code[k] = '\0';
        hlNode* aux = (hlNode*)malloc(sizeof(hlNode));         //新建一个对应此时 treeNode 结点的哈夫曼编码表结点
        aux->code =(char*)malloc(sizeof(char)*(strlen(code)+1));
 //哈夫曼码就等于目前 code 数组中存储的哈夫曼编码
        strcpy(aux->code, code);
        aux->symbol = treeNode->symbol;
        aux->next = NULL;

        if((*table)->first==NULL)    //若哈夫曼编码表还空着
        {
            (*table)->first = aux;
            (*table)->last = aux;
        }
        else
        {
            (*table)->last->next = aux;    //将新建的哈夫曼编码结点与哈夫曼编码表最后一个结点连起来
            (*table)->last = aux;         //此时 aux 变成哈夫曼编码表的最后一个
        }
```

```
    }
    if (treeNode->left != NULL)        //左边分支继续走
    {
        code[k] = '0';
        traverseTree(treeNode->left, table, k + 1, code);
    }
    if (treeNode->right != NULL)       //右边分支继续走
    {
        code[k] = '1';
        traverseTree(treeNode->right, table, k + 1, code);
    }
}

//准备齐全，开始建树
htTree* bulidTree(char* inputString)
{
    //创建用于存储每个字符出现次数的数组
    int* probability = (int*)malloc(sizeof(int) * 256);  //asc 编码能表示的字符也就 256 个,256 个下标分别对应其 asc 码对应的字符，存储的值是 asc 码为该下标的字符 在传入的字符串中出现的次数
    probability = { 0 };//初始化
    //数出现次数
    for (int j = 0; inputString[j] != '\0'; j++)
    {
        probability[(unsigned char)inputString[j]]++;    //从头开始遍历字符串，得到 256 个字符在传入的字符串中 每个字符出现的次数
    }
    //建一个指向队列的头指针的指针
    pQueue* haffumanTree;
    initpQueue(&haffumanTree);     //haffumanTree 初始化   通过改变固定的与 haffumanTree 对应的存储地址里的东西，达到改变 haffumanTree 的目的，因为你使用某个变量本质上是使用它对应的存储区域里的东西

    //填充队列
    for (int k = 0; k < 256; k++) {    //找到数组中出现次数不为 0 的字符，建树
        if (probability[k] != 0) {          //若该字符在字符串中出现过
 由于 probability【】数组中每个下标对应不同的字符，故而每个在字符串中出现的字符只会在列表中创建一个结点
            htNode* aux = (htNode*)malloc(sizeof(htNode));    //新建对应树结点并初始化（为什么先建好树节点？因为下一步新建的列表结点初始化时不是有这个结点对应的树节点嘛）
            aux->symbol = (char)k;
            aux->left = NULL;
            aux->right = NULL;

            addPQueue(&haffumanTree, aux, probability[k]);    //队列 haf
```

```
fumanTree 中新加入结点
            }
        }
        free(probability);

        //生成哈夫曼树
        while (haffumanTree->size != 1)    //循环到队列中只剩最后一个根结点
        {
            int priority = haffumanTree->first->priority;
            priority += haffumanTree->first->next->priority;      //新结点
优先度等于两个优先度最小的结点的优先度之和

            htNode* newNode = (htNode*)malloc(sizeof(htNode));    //新建一
个结点（作为结点中最小的两个的父母结点，就是值等于选出的最小的两个结点权值之和
的那个）
            newNode->left = popQueue(&haffumanTree);      //从队列中取出前两
个结点，作为他的两个孩子
            newNode->right= popQueue(&haffumanTree);

            addPQueue(&haffumanTree, newNode, priority);       //新结点插入
队列中
        }
        htTree* tree = (htTree*)malloc(sizeof(htTree));
        tree->root = popQueue(&haffumanTree);
        return tree;      //返回最后生成的根结点
    }

    //创建哈夫曼编码表格
    hlTable* buildTable(htTree* tree)
    {
        hlTable* table = (hlTable*)malloc(sizeof(hlTable));
        table->first = NULL;
        table->last = NULL;

        char code[256];
        int k = 0;

        traverseTree(tree->root, &table, k, code);  //传入根结点 tree->roo
t，传入待填的编码表 table，k 是递归式目前进展到第几层，code 是每层用于存储目前
编码的数组
        return table;
    }
    //把编码表中所有字符与对应编码都打印出来
    void encode(hlTable* table, char* stringtoencode)
    {
        hlNode* traversal;
        printf("Encoding......\n\nInput string:\n%s\n\nEncoded String: \n",
 stringtoencode);
```

```
    for (int i = 0; stringtoencode[i] != '\0'; i++)
    {
        traversal = table->first;     //每一次都从编码表第一个结点开始遍历

        while (traversal->symbol != stringtoencode[i])        //找到与
当前字符对应的编码表结点
            traversal = traversal->next;
        printf("%s", traversal->code);       //输出编码
    }                                        //循环后可以得到从上到下所有结
点编码排列在一起的结果
    printf("\n");
}

//解码
void decode(htTree* tree, char* stringtodecode)
{
    htNode* traversal = tree->root;
    printf("\n\nDecoding.....\n\nInput String:\n%s\n\nDecoded string:\
n", stringtodecode);
    for (int i = 0; stringtodecode[i] != '\0'; i++)
    {
        if (traversal->left == NULL && traversal->right == NULL)
//若找到根据编码找的叶子结点了
        {
            printf("%c", traversal->symbol);
            traversal = tree->root;          //则下一个1或0肯定是下一个
字符的编码的开头，故要重新从树的根开始顺着编码指示找
        }
        if (stringtodecode[i] == '0')    //顺着编码的指示往下找叶子结点
            traversal = traversal->left;
        if (stringtodecode[i] == '1')
            traversal = traversal->right;

        if (stringtodecode[i] != '0' && stringtodecode[i] != '1')
        {
            printf("你交给我的编码中混入了什么奇怪的东西");
            return;
        }
    }
    if (traversal->left == NULL && traversal->right == NULL)
    {
        printf("%c", traversal->symbol);
    }
}
```

//图（多对多）

//通常由顶点的有穷非空集合和顶点之间边的集合组成，通常表示为：G（V,E）v 是顶点集合，e 是边的集合

线性表中我们把数据元素称之为元素，树中称之为结点，图中称之为顶点
线性表可以没有元素，称之为空表，树可以没有结点，称之为空树，但图的点集应为有穷非空
无向边：vi 与 vj 间的没有方向的边，用（vi，vj）表示
有向边（弧）：vi 到 vj 的有方向的边，用<vi，vj>表示，vi 为弧尾，vj 为弧头
简单图：不存在自己到自己的边，且不出现重复的边
无向完全图：无向图中，任意两个顶点之间都存在边。n 个顶点则有 n(n-1)/2 条边
有向完全图：有向图中，任意两个顶点之间都存在方向相反的两条弧。n 个顶点有 n*(n-1)*条边
*稀疏图与稠密图：边或弧数<n*logn 的图为稀疏图，反之为稠密图
边或弧上如有与之相关的数字，称为权，带权图一般称为网
G1 为 G2 的子图：G1 的边集与点集都是 G2 的子集
若图中存在边(vi，vj)，则称 vi 与 vj 相邻接，边（vi，vj）依附于点 vi，vj，或（vi，vj）与 vi，vj 相关联
若为有向图<vi,vj>，则称 vi 邻接到 vj，vj 邻接自 vi
顶点 v 的度指与 v 相关联的边的数目
若为有向图，以 v 为弧尾的弧数（从 v 出发的）称为 v 的出度，以 v 为弧头（指向 v 的）的称为 v 的入度
从顶点到顶点的路径的长度是路径上边或弧的数目
第一个顶点到最后一个顶点相同路径称为回路或环
简单路径：路径中没有重复的顶点
简单回路：除路径上第一个顶点与最后一个顶点之外没有重复顶点的路径
连通：若存在从 v1 到 v2 的路径，则称 v1 与 v2 连通；若对于途中任意两个点都连通，则称图为连通图
无向图（不一定连通）中极大连通子图称为连通分量：1.连通的子图 2.子图含有极大顶点数 3.具有极大顶点数的连通子图包含依附于这些顶点的所有边 连通图的连通分量就是自己

对有向图来说，任意两顶点 vivj 都有从 vi 到 vj 的路径，称此图为强连通图
有向图（不一定是强连通图）中的极大强连通子图，称为有向图的强连通分量
连通图的生成树：是一个极小的连通子图，包含图中所有顶点，但仅包含足以构成一棵树的 n-1 条边
如果一个有向图恰有一个顶点入度为 0，其余顶点入度为 1，则是一颗有向树*/

//图的存储结构 5 种
//1.邻接矩阵 边数较小的图较浪费存储空间
/*顶点与边分开存放
**顶点信息用一维数组来存储，边的信息用二维数组来存储（坐标来表示它是哪两个点之间的边）
无向图
两点之间有边，两点对应数组位置存 1，无边存 0 数组第几行

对称矩阵：ai=aj，即以矩阵对角线为轴对称
以上用于存储边信息的邻接矩阵即为对称矩阵
要求某点 vi 的度，即为邻接矩阵第 i 行的 1 之和
有向图
不再是对称矩阵 数组 i，由 vi 指向 vj 的边 行箭头出发点，列是箭头指向点
vi 顶点的出度：数组第 i 行的 1 总和
vi 顶点的入度：数组第 i 列的 1 总和

网：每条边上带有权
除有权边填权值以外，其他填正无穷
*/

```
//2.邻接表      仅可统计点的出度
/*数组与链表一起使用
**顶点用一个一维数组存储，内存有指针指向它的第一个邻接点    例如结构为{data与*first}
    每个顶点的所有邻接点构成一条条链表，拴在存有顶点的一维数组对应位置上    链表中每个邻接点结构可以为（此邻接点在数组中对应下标 与 顶点的另一个邻接点的地址）
    对于边上带权值的网图，可以在邻接表结点结构中增加一个数据域存储权值
*/

//3.十字链表 （针对有向图）
/*在邻接表的架构基础上，重新定义顶点数组中每个结点的结构，变为{data，*firstIn，*firstOut}
* firstin 为第一个指向此顶点的边的指针，firstout 为此第一个从此顶点指出的边的指针
* 而链表中每个结点不再表示一个与数组中对应顶点相邻接的顶点，而是一条完整的弧

* 其结构为{tailvex  headvex  headlink  taillink}
* tailvex 与 headvex 分别存储 这条弧的弧尾顶点的数组下标（这条弧必串于顶点数组中 tailvex 对应的链表中） 与 这条弧的弧头顶点的数组下标
* headlink 与 taillink 分别存储 下一条指向 hesdvex 的边的地址  与下一条从 tailvex 指出的边的地址的地址
*
* 由于同时存储了我指向谁和谁指向我，十字链表将邻接表与逆邻接表整合在了一起，可以同时关注到出度与入度
*/

//4.邻接多重表（无向图）
/*如果我们不以顶点为中心，而以边为操作对象，如删除一条边，邻接表就比较麻烦（要遍历两个结点的链表找到对应的那条边（如 1-2 的边，1 的链表里找到 1-2，2 的链表里找到 2-1，然后删除），还要做两次收尾工作）
* 因此我们仿照十字链表，对边结点进行改装
* {ivex ilink jvex jlink }
```

```
  * ivex 与 jvex 为此边依附的两顶点的下标，ilink 指向依附于顶点 i 的下一条边，j
link 指向依附于顶点 j 的下一条边
  */

  //5.边集数组（有向图）
  /*由两个一维数组组成
  * 一个数组存储顶点信息，一个数组存储每条边的信息，数组中每条边结构包括 起点
下标，终点下标，边的权值
  */
```

//图的遍历（目的是遍历每个点，而不是走每条边） 深度优先遍历与广度优先遍历

//深度优先遍历

/*又称深度优先搜索，简称 DFS

* 从某个点开始，固定一个选择优先方向（能走最左边就走最左边），然后开始往下走，没走到一个点在此点做一个标记，一旦遍历到已有标记的点，则退回到上一个结点，从第一列开始再找 1

如果上一个结点所有路对应的结点都有标记了，就再往回退，再从第一列开始找 1，直到退到初始点，则遍历完成

哪个点和哪个点间有连线并不重要，重要的是这点已经遍历过，就无需再通过别的路径找它了

一般是遍历邻接矩阵，通过**递归**实现

对某顶点那一行进行遍历，遍历到值为 1 的结点，则此结点那一列都被设置为 0（所有通向这个已被访问顶点的边全部变为不通），然后接着遍历此顶点对应的那一行（从此新已标记的顶点出发找下一个顶点）找 1，直到到了某一行除刚刚用于找到它的那条边之外，其他边都通向已标记顶点（即那一行其他结点值全为 0），则回退到上一个顶点那一行重新找路

*/

//马踏棋盘算法（骑士周游问题）

/在 88 的棋盘中，将马放在任意位置，按照马的走棋规律将马进行移动，每个方格只能进入一次，要求最后马走遍 64 个方格

用 1-64 来标注马走的路径并进行输出

基本方法与深度优先遍历相同

哈密尔顿路径：经过图中所有顶点且每个点只经过一次的路径*/

//算法实现

```
    int chess[8][8];
    int nextxy(int* x, int* y, int count)      //找到基于（x，y）位置的下
一个可走位置
    {
        switch (count)              //每个点共有八种下一步走法，用 count 来
表示用哪种走法
```

```
    {
    case 0:
        if (*x + 2 <= 7 && y - 1 >= 0 && chess[*x + 2][*y - 1] == 0)
      //8-1=7   如果其最右上方的下一个位置不超出棋盘且未被标志过
        {
            *x += 2;   //将x，y 变为它要走的下一个位置
            *y -= 1;
            return 1;    //表示下一个顶点可以走，有戏
        }
        break;
    case 1:
        if (*x + 2 <= 7 && *y + 1 <= 7 && chess[*x + 2][*y + 1] == 0)
//走最右下方那个位置
        {
            *x += 2;
            *y += 1;
            return 1;
        }
        break;
    case 2:
        if (*x + 1 <= 7 && *y - 2 >=0 && chess[*x + 1][*y - 2] == 0)

        {
            *x += 1;
            *y -= 2;
            return 1;
        }
        break;
    case 3:
        if (*x + 1 <= 7 && *y + 2 <= 7 && chess[*x + 1][*y + 2] == 0)

        {
            *x += 1;
            *y += 2;
            return 1;
        }
        break;
    case 4:
        if (*x - 2 >= 0 && *y - 1 >= 0 && chess[*x - 2][*y - 1] == 0)

        {
            *x -= 2;
            *y -= 1;
            return 1;
        }
        break;
    case 5:
        if (*x - 2 >= 0 && *y + 1 <=7 && chess[*x - 2][*y + 1] == 0)
```

```
            {
                *x -= 2;
                *y += 1;
                return 1;
            }
            break;
        case 6:
            if (*x - 1 >= 0 && *y + 2 <= 7 && chess[*x - 1][*y + 2] == 0)

            {
                *x -= 1;
                *y += 2;
                return 1;
            }
            break;
        case 7:
            if (*x - 1 >= 0 && *y - 2 >=0 && chess[*x - 1][*y - 2] == 0)

            {
                *x -= 1;
                *y -= 2;
                return 1;
            }
            break;

    default:
        break;
    }
    return 0; //若无路可走则返回0
}

//打印棋盘
void printchess()
{
    int i, j;
    for (i = 0; i <= 7; i++)
    {
        for (j = 0; j <= 7; j++)
        {
            printf("%2d\t", chess[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

//深度优先遍历棋盘的算法  x,y 为位置坐标，tag 为标记变量,每走一步 tag 加一
int Travelchess(int x, int y, int tag)
{
```

```
    int x1 = x, y1 = y, flag = 0, count = 0;
    chess[x][y] = tag;              //相当于最后整出来是输出整个棋盘，棋盘每个格存的数就是这个格子是第几步遍历到的

    if (tag == 64)     //如果遍历到最后一个格子
    {       //打印棋盘
        printchess();
        return 1;
    }

    //若还未遍历到最后一个，则使用nextxy函数找到下一个可走坐标，若找到则flag=1，未找到flag=0
    flag = nextxy(&x1, &y1, count);      //经过此函数，x1y1已变成它能走的下一个位置的坐标
    while (flag == 0 && count < 7)
    {
        count++;
        flag = nextxy(&x1, &y1, count);       //把所有下一步位置都试一遍，直到试出能走的
    }

    //（x，y）点的每一个能走的位置都要通过递归往下走，直到这点所有能走的位置都走完了，即flag=0
    while (flag)       //while用于控制找到（x，y）点每一个可移动位置，每一次while中调用travelchess用于沿着此次while找到的（x，y）的可移动位置走下去
    {
        if (Travelchess(x1, y1, tag + 1))    //继续到下一个位置
        {
            return 1;
        }
        //然后退回x y点（x1 y1本来已经是x y下一个可走点的位置坐标了，现在x1 y1又翻回去x y的位置）
        x1 = x;                 //x，y为最初传入的点位置
        y1 = y;
        //找x y点的另一条能走的路
        count++;
        flag = nextxy(&x1,&y1,count);
        while (flag == 0 && count < 7)
        {
            count++;
            flag = nextxy(&x1, &y1, count);
        }
        //然后通过while（flag）在 沿着新路走 与 找另一个可走位置 同时进行
    }
    if (flag == 0)    //如果走不下去
    {
```

```
            chess[x][y] == 0;
        }

        return 0;
    }
```

//广度优先遍历 BFS
/*深度优先以往下遍历为主旨，广度优先以一层一层遍历为主旨。如从左到右广度优先，则从根开始，从左到右遍历根的子结点，再从左到右遍历每颗子结点的子结点（没有标记的），遍历过的就进行标记，直到全部标记为止
一般使用队列实现此遍历算法：
根入队列，取出根，将根的孩子结点从左到右入队列，孩子们开始出队列，出一个，就将他的孩子结点再从左到右入队列，
*/
//邻接矩阵的广度优先遍历

```
void BFS(MGraph G)  //传入邻接矩阵，包含G.pointNumber 结点数，G.point 存结点的数组，G.art 存边的二维数组
    {
        int i, j, visited[G.pointNumber];   //visited 用于标记每个点是否被遍历过
        Queue Q;                            //队列Q
        for (i = 0; i < G.pointNumber; i++)
        {
            visited[i] = 0;
        }
        initQueue(&Q);                      //初始化队列
        for (i = 0; i < G.pointNumber; i++)         //用于遍历一整个森林，防止存在树与树之间断开导致仅能遍历一颗的可能
        {
            if (!visited[i])            //若 i 下标对应结点还未被访问
            {
                printf("%c", G.point[i]);
                visited[i] = 1;         //访问后标记
                EnQueue(&Q, i);        //i 下标对应结点入队

                while (!QueueEmpty(Q))      //当队列不为空
                {
                    DeQueue(&Q, &i);       //i 下标对应结点出队，队列中 i 之后那个结点顺移到 i 的位置
                    for (j = 0; j < G.pointNumber; j++)       //访问所有 i 连接的未被访问过的结点并入队
                    {
                        if (G.art[i][j] == 1 && !visited[j])
                        {
                            printf("%c", G.point[j]);
                            visited[j] = 1;
```

```
                    EnQueue(&Q, j);
                }
            }
        }
    }
}


//最小生成树
//在图中找出一棵包含所有结点的树，满足路径权值相加最小
//普雷姆算法
//并非以点为层次一个点一个点找最短边，而是以点为线索，一层一层筛边并找到这一层边中最短的那个，但是找出来的最短边不一定与线索点相连
void Prim(MGragh G)      //图G以二维数组存储，下标表示结点，存储为两节点间的路径权值
{
    int min, i, j, k;
    int adjvex[MAXVEX];   //存储每个下标连接的最短路径对应的另一头的结点的下标，故而数组长度为图中结点的数量
    int lowcost[MAXVEX]; //用于存储在目前遍历过的边中，各个点的最短路径。故而数组长度为图中结点数量，ps：未相连则权值为INFINITY，自己与自己权值为0
    //初始化，以v0为根，开始找最小生成树
    lowcost[0] = 0;      //以0结点到自己的路径长度为0
    adjvex[0] = 0;

    for (i = 1; i < G.numVertexes; i++)
    {
        lowcost[i] = G.arc[0][i];     //将0结点到下标数字对应结点的距离输入lowcost数组
        adjvex[i] = 0;         //先初始化假装各结点都是到0路径最短
    }

    for (i = 1; i < G.numVertexes; i++)
    {
        min = INFINITY;
        j = 1;
        k = 0;

        while (j < G.numVertexes)        //遍历lowcost找到目前遍历过的边中的最短边，并将此边对应的顶点下标存在k中（此边必然是一端结点已被遍历过，另一端结点就是当前下标）
        {                                //lowcost不断筛选并存储各个还未遍历到的点的当前遍历过的边中最短路径长度
            if (lowcost[j] != 0 && lowcost[j] < min)
            {
                min = lowcost[j];
```

```
                k = j;
            }
            j++;
        }

        printf("(%d,%d)", adjvex[k], k);   //打印当前结点权值最小的边的权值，与这条边对应的另一个结点
        lowcost[k] = 0;          //存0，表示此顶点已完成任务，开始找k的最短路径，而k结点到自己的路径长度为0

        //再将k顶点到各顶点的边的权值输入lowcost中（筛一遍在目前遍历过的边中，各个点的最短路径）
        for (j = 1; j < G.numVertexes; j++)
        {
            if (lowcost[j] != 0 && G.arc[k][j] < lowcost[j])      //等于0的就是已经找出最短路径的点，不再找它
            {                                                     //由于这种已找到最短路径的点就不会再考虑与之相连的边的特性，比如找v1的最短边是与v3相连的那条，但是v3的边中最短的不是与v1相连的那条，此时就需要取其中权值最短的来作为v3的对应边，故而有判断条件G.arc[k][j] < lowcost[j]
                lowcost[j] = G.arc[k][j];
                adjvex[j] = k;
            }
        }
    }
}

//最小生成树：克鲁斯卡尔算法
/*上面的普利姆算法是以某顶点为起点，逐步找到各个顶点上最小权值的边来构建最小生成树
* 而现在克鲁斯卡尔算法则是以边为起点，找权值最小的边来构建最小生成树
* 精髓：加入的边不可以和现有的边构成环路
*/
typedef struct EDGE {
    int begin;
    int end;
    int weight;
}Edge;
int Find(int* parent, int f) {
    while (parent[f] > 0)
        f = parent[f];
    return f;
}
//Kruskal算法生成最小生成树
void MiniSpanTree_Kruskal(MGrapoh G) {
    int i, n, m;
    Edge edges[MAGEDGE];    //定义边集数组 ！！！！重点，在此数组中图的各条边是按权值从小到大存储的。
```

```
    int parent[MAXVEX];     //parent 数组用于判断边与边是否形成环路。i 点与对应的parent[i]点表示此两点不可相连，相连则树中出现环路

    for (i = 0; i < G.numVertexes; i++)
        parent[i] = 0;
    for (i = 0; i < G.numEdges; i++) {       //遍历边集数组中的每条边，从前往后，即从权值最小的边到最大的边依次遍历
        n = Find(parent, edges[i].begin);
        m = Find(parent, edges[i].end);

        if (n != m) {          //若n==m 则形成环路
            parent[n] = m;         //将此边结尾顶点方入下标为起点的parent数组中，表明此顶点已在生成树中，并且以后n 点与m 点都在【同一颗】生成树中，绝不可相连
            printf("(%d,%d) %d", edges[i].begin, edges[i].end, edges[i].weight);
        }
    }
}

//最短路径：迪杰斯特拉（直接跳毕竟学过很多次了）--一个顶点到所有顶点的最短路径

//最短路径：弗洛伊德算法----所以顶点到所有顶点的最短路径
typedef int Path[MAXVEX][MAXVEX];
typedef int Short[MAXVEX][MAXVEX];
/*这两个数组是灵魂
*/
void Floyd(MGragh G, Path* P, Short* D)
{
    int v, w, k;
    //初始化二维数组P 与D
    //D 里初始存各条路的权值，算法完毕后，如D[i][j]存的是i 到j 的最短路径的总长
    //P 里P[i]对应的P[i][0]P[i][1]P[i][2]P[i][3]对应存的都是列标，方便之后i 到每个点的路都遍历一遍
    for (v = 0; v < G.numVertexes; v++)
    {
        for (w = 0; w < G.numVertexes; w++)
        {
            (*D)[v][w] = G.matirx[v][w];
            (*P)[v][w] = w;
        }
    }
    //算法
    for (k = 0; k < G.numVertexes; k++)   //每一个k
    {
        for (v = 0; v < G.numVertexes; v++) //对应一个点
        {
```

```
            for (w = 0; w < G.numVertexes; w++)   //v 点到每条边
            {
                if ((*D)[v][w] > (*D)[v][k] + (*D)[k][w])   //以 k 点为中转，比较 v->w 与 v->k->w 的长度
                {
                    (*D)[v][w] = (*D)[v][k] + (*D)[k][w];
                    (*P)[v][w] = (*P)[v][k];        //若中转后更短，则 v->w 最短路上第二个点由原 p[v][w]变为 p[v][k]，即把 v->k 这段路插进 v->w 的最短路径指示
                }
            }
        }
    }
    //算法完成后 P[i][j]里存的是 i 点到 j 点的最短路径上，i 点连的下一个点是哪个
}
```

//排序算法
/*相关概念：

- DGA 图：无环图，也即一个无环的有向图
- AOV 网：一个表示工程的无权有向图，顶点表示活动，有向边表示活动的执行条件
- 可以理解为每个活动都有先行条件，并且也作为其他活动的先行条件，将这些活动按照这种条件规则，排列为一个 按序执行每件事并保证执行每件事时它的先行条件活动已经都执行过 的执行序列，就是拓扑排序，生成的序列就是拓扑序列
- 对 AOV 网进行拓扑排序的步骤：
- 1.从 AOV 网中选择所以没有前驱的点（即入度为 0），并输出后删除
- 2.删除从该顶点出发的所有有向边（由此又会有一批没有前驱结点的结点出现）
- 3.重复上述两步，直到网中不存在没有前驱结点的顶点
- AOV 网图一般用邻接表存储
- 表头结构为 下标结点对应的入度，该结点的 data，以及指向对应链表的指针

```
/
//拓扑排序
 //邻接表边结点结构
 typedef struct Edge
```

```
{
int adjvex;
struct Edge next;
}Edge;
//邻接表顶点表结点结构
typedef struct Ver
{
int in;
int data;
Edge* firstedge;
}Ver,Adjlist[MAXVEX];
//AOV 图
typedef struct
{
Adjlist adjList; //邻接表的顶点表
int numVertexes, numEdges; //分别是图中还剩的顶点数与边数
}graphAdjList,GraphAdjList;
//拓扑排序算法
//若 GL 无回路，输出拓扑排序序列，否则返回 error
void TopologicalSort(GraphAdjList GL)
{
Edge e;
int i, k, gettop;
int top = 0; //用于栈指针下标索引
int count = 0; //用于统计输出顶点的个数
int* stack; //用于存储入度为 0 的顶点

  stack = (int*)malloc(GL->numVertexes * sizeof(int));   //stack
大小可以存储所有顶点

  for (i = 0; i < GL->numVertexes; i++)
  {
      if (GL->adjList[i].in == 0)   //初始检查一遍图，入度为0 的顶点
入栈
          stack[++top] = i;
  }

  while (top != 0) {  //top=几，就有几个入度为0 的顶点
      gettop = stack[top--];    //在 gettop=top[1]后，top 又变回0
      printf("%d->", GL->adjList[gettop].data);
      count++;

      for (e = GL->adjList[gettop].firstedge; e; e = e->next)  //
遍历此入度为0 的点的所以出度边
      {
          k = e->adjvex;  //k 等于这条边指向的点
```

```
                if (!(--GL->adjList[k].in))  //k 点的入度-1
                    stack[++top] = k;    //若减一后（删除刚刚那条边后）k 点
        的入度也变为0，则k 点也入栈      相当于一条路走到黑，走不下去了再换 top-
        1 那条路，再一条黑走下去
            }
        }
        if (count < GL->numVertexes) {
            printf("ERROR");
        }

    }
```

//关键路径
/*AOE 网：表示工程的带权有向图，顶点表示事件，有向边表示活动，边上的权值表示活动持续时间
* 将 AOE 网中没有入边的顶点称为始点或源点，没有出边的顶点称为终点或汇点
* AOV 网与 AOE 网比较：
* AOV 顶点表示活动，AOE 边表示活动，顶点表示结果事件
* AOV 边没有权值，AOE 边有权值表示活动持续时间
* 关键路径：加起来权值最大的那条，即所有活动能同时进行的都同时进行后最短时间路径
* etv：事件（顶点）最早发生时间，再早就没法开始
* ltv：事件（顶点）最晚发生时间，再晚会延误工期
* ete：活动（边）最早开始时间
* lte：活动最晚开始时间
*/

```
//AVE 图存储和上面拓扑排序一摸一样，不用改，直接挪下来
//edge 的改一下，加个权
typedef struct Edge
{
    int adjvex;
    int weight;
    struct Edge* next;
}Edge;
int* etv, * ltv;
int* stack2;   //用于存储拓扑序列的栈
int top2;      //用于stack2 的栈顶指针

//先拓扑排序
void TopSort2(GraphAdjList GL)
{
    Edge* e;
    int i, k, gettop;
```

```
    int top = 0;      //用于栈指针下标索引
    int count = 0;    //用于统计输出顶点的个数
    int* stack;       //用于存储入度为0 的顶点

    stack = (int*)malloc(GL->numVertexes * sizeof(int));   //stack 大小
可以存储所有顶点

    for (i = 0; i < GL->numVertexes; i++)
    {
        if (GL->adjList[i].in == 0)   //初始检查一遍图，入度为0 的顶点入栈

            stack[++top] = i;
    }
    //初始化 etv 都为0
    top2 = 0;
    etv = (int*)malloc(GL->numVertexes * sizeof(int));
    for (i = 0; i < GL->numVertexes; i++)
    {
        etv[i] = 0;
    }
    stack2 = (int*)malloc(GL->numVertexes * sizeof(int));

    while (top != 0)
    {
        gettop = stack[top--];
        stack2[++top2] = gettop;    //保存拓扑顺序
        count++;

        for (e = GL->adjList[gettop].firstedge; e; e = e->next)   //遍历
此入度为0 的点的所以出度边
        {
            k = e->adjvex;
            if (!(--GL->adjList[k].in))
            {
                stack[++top] = k;
            }
            if ((etv[gettop] + e->weight) > etv[k])  //如果沿 gettop 到 k
这条路比原来存着的到 k 点的路耗时长，则延后 k 点最早发生时间到 gettop 到 k 这条路
走完
            {
                etv[k] = etv[gettop] + e->weight;
            }
        }
    }
    if (count < GL->numVertexes)
        printf("ERROR");
}
//求关键路径
```

```
void CriticalPath(GraphAdjList GL)
{
    Edge* e;
    int i, gettop, k, j;
    int ete, lte;

    //调用改进后的拓扑排序，求出etv 和stack2
    TopSort2(GL);

    //初始化ltv 都为汇点时间
    ltv = (int*)malloc(GL->numVertexes * sizeof(int));
    for (i = 0; i < GL->numVertexes; i++)
    {
        ltv[i] = etv[GL->numVertexes - 1];
    }
    //从汇点倒过来逐个计算ltv
    while (0 != top2)
    {
        gettop = stack2[top2--];    //第一个出栈为汇点
        for (e = GL->adjList[gettop].firstedge; e; e = e->next)
        {
            k = e->adjvex;
            if ((ltv[k] - e->weight) < ltv[gettop])           // 【getto
p】--(e->weight)--> 【k】
                ltv[gettop] = ltv[k] - e->weight;     //最晚发生时间只能往
前推，不能往后推
        }
    }
    //通过etv 和ltv 求ete 和lte
    for (j = 0; j < GL->numVertexes; j++)   //顶点j
    {
        for (e = GL->adjList[j].firstedge; e; e = e->next)    //遍历每个
顶点的每条边,求每条边的ete 与lte。每条边遍历时为e
        {
            k = e->adjvex;
            ete = etv[j];   //ete 与对应点的etv 必相等
            lte = ltv[k] - e->weight;   //lte 与对应点的ltv 之间有函数关系
            if (ete == lte)  //相等，则此时e 边代表的活动为关键活动，关键活
动组成的路径为关键路径
                printf("<v%d,v%d> length: %d ,", GL->adjList[j].data, GL
->adjList[k].data, e->weight);
        }
    }
}
```

//查找

/静态查找：数据集合稳定，不需要添加、删除元素的查找操作

动态查找：在查找过程中需要同时删除、添加元素的查找操作/

//静态查找
//1.顺序查找
/*又称线性查找，是最基本的查找技术，他的查找过程是：从第一个或者最后一个元素开始，按序一个一个比对值查找*/
//2.插值查找（按比例查找）
/*折半查找的衍生物，折半查找每一次 mid=(high+low)/2，而插值查找每一次 mid 的值不是单纯取中间，而是根据比率取位置的，mid=low+(key-list[low]/list[high]-list[low])*(high-low)
折半查找适用于数据量大，且增长均匀的数据集合的查找/
//3.斐波那契查找（黄金比例查找）0.618
/斐波那契额数列：

- F[k]：1，1，2，3，5，8，13，21，34，55……随着数字增大，前一个数字与后一个数字的比值越来越接近 0.618，且 F[k]=F[k-1]+F[k-2]
  将数据集合扩充为最接近的斐波那契数列中的元素，再往前根据斐波那契数列 F[k]=F[k-1]+F[k-2]分割它,取 F[k-1]与 F[k-2]的夹缝作为 mid。
  mid=left+F[n-1]-1 （left=low）
  - 相等，则查找成功，返回中间位置即可；
    中间值小于目标值，则说明目标值位于中间值到右边界之间（即右区间），右区间含有 F(n-2)个元素，所以 n 应该更新为 n=n-2；
    中间值大于目标值，这说明目标值位于左边界和中间值之间（即左区间），左区间含有 F(n-1)个元素，所以 n 应更新为 n=n-1；
    /
    //以上两个个查找方法前提都是数据已经有序排好存放
    //4.线性索引查找
    /稠密索引：有一个索引表，包含关键码和指向关键码对应数据的指针，查找时只需要使关键码相同，即可找到对应数据；仅适用于数据量小的数据集查找
    - 分块索引：将数据集分块，索引表仅存各块中最大元素与各块内起始地址字；各块之间有排序，但块内无排序；用一个元素代表整块中所有元素参与查找
      倒排索引：
      在搜索引擎中每个文件都对应一个文件 ID，文件内容被表示为一系列关键词的集合（实际上在搜索引擎索引库中，关键词也已经转换为关键词 ID）。例如“文档 1”经过分词，提取了 20 个关键词，每个关键词都会记录它在文档中的出现次数和出现位置。
      得到正向索引的结构如下：
      “文档 1”的 ID > 单词 1：出现次数，出现位置列表；单词 2：

出现次数，出现位置列表；……。
“文档 2”的 ID > 此文档出现的关键词列表。

当用户在主页上搜索关键词“华为手机”时，假设只存在正向索引，那么就需要扫描索引库中的所有文档，找出所有包含关键词“华为手机”的文档，再根据打分模型进行打分，排出名次后呈现给用户。因为互联网上收录在搜索引擎中的文档的数目是个天文数字，这样的索引结构根本无法满足实时返回排名结果的要求。

所以，搜索引擎会将正向索引重新构建为倒排索引，即把文件 ID 对应到关键词的映射转换为关键词到文件 ID 的映射，每个关键词都对应着一系列的文件，这些文件中都出现这个关键词。

得到倒排索引的结构如下：

“关键词 1”：“文档 1”的 ID，“文档 2”的 ID，……。

“关键词 2”：带有此关键词的文档 ID 列表。

从词的关键字，去找文档。
*
*/

```
//二叉排序树
/*又称为二叉查找树，它满足：
* 根结点的值永远大于他的左子树上的任意结点的值
* 根结点的值永远小于他的右子树上的任意结点的值
* 取树的任意左右子树也必为二叉排序树
*
* 二叉排序树根据中序遍历得到的就是整个数集从小到大的排序
*/
//二叉排序树的二叉链表的结点定义
typedef struct BiTNode
{
    int data;
    struct BiTNode* lchild, * rchild;
}BiTNode,*BiTree;
//二叉排序树创建过程省略
/*接下来递归查找二叉排序树 T 中是否存在 key，指针 f 指向 T 的双亲，其初始值调用为 NULL；
 若查找成功，则指针 p 指向该数据元素结点，并返回 True，否则指针 p 指向查找路径上访问的最后一个结点，并返回 False*/
bool SearchBST(BiTree T,int key,BiTree f,BiTree *p ) {
    if (!T)   //查找失败
    {
        *p = f;
        return false;
    }
```

```
    else if (key == T->data)   //查找成功
    {
        *p = T;
        return true;
    }
    else if (key < T->data) {
        return SearchBST(T->lchild, key, T, p); //在左子树继续查找
    }
    else {
        return SearchBST(T->rchild, key, T, p);  //在右子树继续查找
    }
}
//二叉排序树插入操作
//当二叉排序树 T 中不存在关键字等于 key 的数据元素时，插入 key 并返回 TRUE，若已存在 key，返回 FALSE
bool InsertBST(BiTree* T, int key) {
    BiTree p, s;
    if (!SearchBST(*T, key, NULL, &p))  //若在书中不存在 key，则开始插入操作
    {
        s = (BiTree)malloc(sizeof(BiTree));   //创建新结点
        s->data = key;
        s->lchild = s->rchild = NULL;
        if (!p) {     //若原树也为空，则插入 s 作为根结点
            *T = s;
        }
        else if (key < p->data) {
            p->lchild = s;  //插入 s 作为左孩子
        }
        else {
            p->rchild = s; //插入 s 作为右孩子
        }
        return true;
    }
    else
    {
        return false;   //树中找到关键字相同的结点，不插入
    }
}
//使用 insertbst 和 searchbst 算法结合，可以用于创建二叉排序树：建一个根结点，然后每输入一个元素，就使用 insert 插入树中
//二叉排序树结点的删除
/*难点在于在删除此结点后怎么连接，使树仍保持二叉排序树特性。
 以及此处删除不一定是 delete 释放这个结点然后再重新把它周边的结点连连看，也可以把要用于替换此 被删除结点的位置 的 结点的 data 赋给此即将被删除结点，然后 delete 用于替换的结点的原位置即可*/

bool Delete(BiTree* p);//这才是真正的删除操作使用的函数，包括了删除此结点与
```

```
删除后的收尾
 bool DeleteBST(BiTree* T, int key)//用于递归找到我们要删除的那个结点的位置并打开对他的删除操作的开关
 {
    if (!T)
    {
        return false;
    }
    else
    {
        if (key == (*T)->data)   //找到我们要删除的点
        {
            return Delete(T);
        }
        else if (key < (*T)->data)
        {
            return DeleteBST(&(*T)->lchild, key);
        }
        else
        {
            return DeleteBST(&(*T)->rchild, key);
        }
    }
 }

 bool Delete(BiTree* p)
 {
    BiTree q, s;
    if ((*p)->rchild == NULL)   //若要删除的点没有右子树，那就把左子树的根顶上来就行，可以直接删
    {
        q = *p;
        *p = (*p)->lchild;
        free(q);
    }
    else if ((*p)->lchild == NULL)   //没有左子树同理
    {
        q = *p;
        *p = (*p)->rchild;
        free(q);
    }
    else        //若要删除的点既有左子树又有右子树，则找到它的前驱节点（中序遍历整颗排序树，在他前面的那个结点就是它的前驱节点），替换他们的数据，然后删除前驱结点
    {
        q = *p;
        s = (*p)->lchild;

        while (s->rchild)   //寻找前驱结点
```

```
        {
            q = s;              //q 存储了这个前驱结点的根
            s = s->rchild;      //s 最终存储了它的前驱结点
        }
        (*p)->data = s->data;   //热知识：已知 s 是 p 的前驱节点，则 s 必没有右子树
        //根据 p 与它的前驱结点 s 不同的位置关系，有不同的善后方法
        if (q != *p)  //若 p 不是 s 的根
        {
            q->rchild = s->lchild;
        }
        else    //若 p 就是 s 的根
        {
            q->lchild = s->lchild;
        }
        free(s);
    }
}

//但是有些二叉排序树查找效率很低，比如一直都是往左子树上拼
//有没有办法生成一个均衡的二叉排序树？
//平衡二叉排序树（avl）：所有左子树深度减去对应右子树深度得数**绝对值**小于等于一。（**左减右，得正数与负数也会有不同的处理方式）
/*如何生成平衡二叉树？
 * 在排序二叉树的基础上，每输入一个数都要检测是否满足平衡，不满足则立即进行调整
 */

#define LH 1 //lefthigh 左子树高了
#define EH 0 //两子树等高
#define RH -1 //右子树高了
typedef struct BiTNode
{
int data;
int bf; //平衡因子
struct BiTNode* lchild, * rchild;
}BiNTode,*BiTree;

void R_Rotate(BiTree* p)
{
    BiTree L;
    L = (*p)->lchild;
    (*p)->lchild = L->rchild;
    L->rchild = (*p);
    *p = L;
}
void L_Rotate(BiTree* p)
{
```

```
    BiTree L;
    L = (*p)->rchild;
    (*p)->rchild = L->lchild;
    L->lchild = (*p);
    *p = L;
}

void LeftBalance(BiTree* T) {
    BiTree L, Lr;
    L = (*T)->lchild;

    switch (L->bf)
    {
    case LH:
        (*T)->bf = L->bf = EH;
        R_Rotate(T);
        break;
    case RH:
        Lr = L->rchild;
        switch (Lr->bf)
        {
        case LH:
            (*T)->bf = RH;
            L->bf = EH;
            break;
        case EH:
            (*T)->bf = L->bf = EH;
            break;
        case RH:
            (*T)->bf = EH;
            L->bf = LH;
            break;
        }
        Lr->bf = EH;
        L_Rotate(&(*T)->lchild);
        R_Rotate(T);
    }
}

int InsertAVL(BiTree* T, int e, int* taller)  //要插入元素e   taller 表示这个元素插入后树会不会长高   T 为此时递归到的结点
{
    if (!*T)  //当递归到底到了NULL 时，就新建结点
    {
        *T = (BiTree)malloc(sizeof(BiTNode));
        (*T)->data = e;
        (*T)->lchild = (*T)->rchild = NULL;
        (*T)->bf = EH;   //新建的结点还没有左右子树，当然左右子树就是等高
 //taller 表示是否长高，bf 表示向哪边长高
```

```
        *taller = true;    //表示是否长高了
    }
    else
    {
        if (e == (*T)->data)
        {
            *taller = false;    //没能长高
            return false;
        }
        if (e < (*T)->data)  //若结点小于T，则会往T的左子树上加
        {
            if (!InsertAVL(&(*T)->lchild, e, taller))  //小于则向左走，同时T的左子树高度增加
            {
                return false;
            }
            if (*taller)
            {
                switch ((*T)->bf)   //bf表示T本来不加这个新结点时的状态是什么样
                {
                case LH:    //若本来T就是左子树比较长
                    LeftBalance(T);   //此时要再往左边加，就需要特殊的左平衡处理
                    *taller = false;  //左平衡处理后，会相当于这个节点加上去，整棵树没长高
                    break;
                case EH:   //若T本来左右子树一样长，此时左边加上这个结点，就会左子树变长
                    (*T)->bf = LH;  //更改现在T的状态
                    *taller = true;  //这个结点会导致树长高
                    break;
                case RH:   //若T本来右子树长，此时左子树一加上这个节点，T左右子树反而平衡了
                    (*T)->bf = EH;
                    *taller = false;   //这个结点不会导致树长高
                    break;
                }
            }
        }
        else   //e>T，要往T的右子树上加
        {
            if (!InsertAVL(&(*T)->rchild, e, taller))
            {
                return false;
            }
            if (*taller)
            {
```

```
                switch ((*T)->bf)
                {
                case LH:
                    (*T)->bf = EH;
                    *taller = false;
                    break;
                case EH:
                    (*T)->bf = RH;
                    *taller = true;
                    break;
                case RH:     //右上加右
                    RightBalance(T);
                    *taller = false;
                    break;
                }
            }
        }
    }
}
```

//多路查找树***不考
//其每一个结点的孩子可以多于两个，且每一个结点可以存储多个数据
/*一个结点拥有两个孩子和一个元素我们称之为 2 结点，他跟二叉排序树类似，左子树包含的元素小于结点的元素，右子树包含的元素大于结点的元素。
不过与二叉排序树不同的是，2 结点要么没有孩子，要么就应该有两个孩子，不能只有一个孩子
3 结点同理，要么没有孩子，要么有三个孩子，不能只有 1 个或只有两个
2-3 树：树的每个结点要么是 2 结点要么是 3 结点，且由于 2 结点和 3 结点的特性，树的所有叶子必然都在同一个层次
而由于这种特性，造就了 2-3 独特的插入方法：尽量不添加新结点，而是拆分 3 结点或者将 2 结点变为 3 结点：插二就变 3，插三就往上找 2 结点然后下面 3 变二，上面 2 变 3
*/

//散列表查找（哈希表查找）
*/*散列技术，即在记录的存储位置与它的关键字之间建立一个确定的对应关系 f，使得每个关键字 key 对应一个存储位置 f(key)*
这里我们把这种对应关系 f 称为散列函数，又称为哈希函数，采用散列技术将记录存储在一块连续的存储空间中，这块连续存储空间称为散列表或哈希表
散列表的查找步骤
当存储记录时，通过散列函数计算出记录的散列地址，将此记录存在此地址
当查找记录时，我们通过同样的是散列函数计算记录的散列地址，并按此散列地址访问该记录
故而散列函数必须保证不同的记录通过 f 计算出的地址是不会相同的/
//散列函数的构造
*/*构造的两个基本原则：1.计算简单 2.分布均匀*
*

```
 * 直接定址法：
 * f(key)=key  (f(key)是相对于某确定地址的偏移量)
 * f(key)=key-1300
 * 取关键字的一个线性函数作为偏移量
 *
 * 数字分析法：
 * 适合用于处理数字关键字位数较大的情况，抽取其中某几位作为偏移量
 *
 * 平方取中法：
 * 将关键字平方后取中间若干位数字作为散列地址
 *
 * 折叠法：
 * 将关键字从左到右分割成位数相等的几部分，然后将这几部分叠加求和，并按散列表表长取后几位作为散列地址
 *
 * 除留余数法：
 * 此方法为最常用的构造函数的方法，对于散列表长为m 的散列函数计算公式为：
 * f(key)=key mod p(p<=m) 故p 的选择很重要
 *
 * 随机数法：
 * 选择一个随机数，取关键字的随机函数值为他的散列地址。
 * 即f(key)=random(key)
 * 此random 函数为随机函数，当关键字长度不等时，这个方法是比较合适的
 */
 //处理散列函数冲突的方法
 /*可能会出现不同key 经过f 后得到相同地址后，即出现冲突
 *
 开放定址法
 所谓开放定址法，就是一旦发生了冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将数据存入
 fi(key)=(f(key)+di) mod m (di=1,2,3...m-1)
 可以修改di 的取值方式，例如平方运算：di=1^2,(-1)^2,2^2,(-2)^2,3^2...q^2 =>二次探测法
 也可以使用随机函数获得di 的值    =>随即探测法

 再散列函数法
 多个散列函数，第一个不行换下一个

 链地址法
 使用邻接表存储结构，很多元素对应一个位置

 公共溢出区法
 一个基本表，如果有冲突，就放到另一个表--溢出表中
 */

 //排序算法
```

```
/*稳定排序与不稳定排序
* 稳定排序：当经过此排序后，key 相同的两个元素前后顺序不变，和排序前一样
*/
//1.冒泡排序
/*两两相邻关键字比大小，反序则交换，未反序则不交换，不断遍历直到没有反序为止*
/
void BubbleSort(int k[], int n) {
    int temp;
    bool flag = true;
    while (flag) {
        flag = false;
        for (int j = 0; j < n-1; j++) {
            if (k[j] > k[j + 1]) {
                temp = k[j];
                k[j] = k[j + 1];
                k[j + 1] = temp;
                flag = true;
            }
        }
    }
}
//2.选择排序
/*在第i 次循环中，从 i---（n-1）中选出最小的数字与i 交换*/
void ChooseSort(int k[], int n) {
    int p = 0;
    while (p < n) {
        int min = p;
        for (int i = p; i < n; i++) {
            if (k[i] < k[min])min = i;
        }
        int temp = k[p];
        k[p] = k[min];
        k[min] = temp;
        p++;
    }
}
//3.直接插入排序
/*将一个记录插入到一个已经排好序的有序列表中，从而得到一个新的，记录数加1 的
有序列表
k[0]永远是空出来的，为了插入与修改*/
void InsertSort(int k[], int n)
{
    int i, j, temp;
    for (i = 1; i < n; i++)
    {
        if (k[i] < k[i - 1])   //找见比前一个小的元素
        {
            temp = k[i];    //记录这个元素
```

```
            for (j = i - 1; k[j] > temp; j--)  //把这个元素之前的比这个元素大的都往后移一位
            {
                k[j + 1] = k[j];
            }
            k[j + 1] = temp; //把这元素插进得到的空出的位置j+1中
        }
    }
}
//4.希尔排序
/*改进后的直接插入排序，只不过原来元素比较的跨度是1，就是i和i+1比，现在跨度由大变小
* 如第一轮i和i+6比，直到i+6==n-1
* 第二轮i和i+3比，同上
* 第三轮...
*/
void ShallSort(int k[], int n)
{
    int i, j, temp;
    int gap = n;
    do {
        gap = gap / 3 + 1;
        for (i =gap; i < n; i++)
        {
            if (k[i] < k[i - gap])
            {
                temp = k[i];
                for (j = i - gap; k[j] > temp; j=j-gap)
                {
                    k[j + gap] = k[j];
                }
                k[j + gap] = temp;
            }
        }
    } while (gap > 1);
}
//5.堆排序
/*将数组虚拟化为完全二叉树，再将此完全二叉树构建成大顶堆或小顶堆
大顶堆：符合根值必然比左右孩子都大的特性的完全二叉树
必然满足k(i)>=k(2i)  k(i)>=k(2i+1)    1<=i<=n/2
小顶堆：根值必然小于或等于左右孩子
必然满足k(i)<=k(2i)  k(i)<=k(2i+1)
故将大顶堆或小顶堆根据层序遍历存入数组，其下标必然满足上式
*/
/*堆排序就是利用堆进行排序的算法，它的基本思想是：
将待排序的序列构造成一个大顶堆（或小顶堆）
此时整个序列的最大值就是堆顶的根节点，将他移走（就是将其与堆数组的末尾元素交换，此时末尾元素就是最大值）
```

```
 然后将剩余的n-1 个序列重新构造成一个堆，这样就会得到这n-1 个元素中的最大值
 如此反复*/
 //构建大顶堆
 /*从下往上，从右往左调整，即从最小的树开始慢慢往上构建一整个大树
 堆排序假设第一个元素下标为1，先作为堆的根，则下标为2*i 的是i 的左孩子，下标
为2*i+1 就是i 的右孩子*/
 void swap(int k[], int i, int j)
 {
     int temp;
     temp = k[i];
     k[i] = k[j];
     k[j] = temp;
 }
 void HeapAdjust(int k[], int s, int n)   //保证以下标s 存根，长度为n 的数
组是一个完美的大顶堆
 {
     int i, temp;
     temp = k[s];   //s 为此次构建的堆的根结点位置
     //temp 占用了根的位置，但是需要检测它能不能作根，不能就一直不断往下走找它
能做根的位置，而在往下的过程中，顺便也重新构建了大顶堆
     for (i = 2 * s; i <= n; i *= 2)    //2*s 即为根s 的左孩子  i=i*2 即此时
i 变为它的左孩子，以进行下一次temp 与孩子结点的孩子结点的比较，直到temp 找到它
应该在的位置（即左右孩子都比temp 小的根位置）
     {
         if (i < n && k[i] < k[i + 1])
         {
             i++;       //找到左右孩子里最大的，将其下标存i 中
         }
         if (temp >= k[i])   //如果根比它的这两个孩子都大
         {
             break;     //ok 没事儿了
         }
         k[s] = k[i];    //没有break；说明孩子里有比temp 大的，则temp 不适
合放在这个根位置上，则把这个比temp 大的数作根放到上面
         s = i;      //用s 记录：如果下一次temp 找到它该在的层次了，它应该放
在哪个具体位置
     }
     k[s] = temp;
 }
 void HeapSort(int k[], int n)
 {
     int i;
     for (i = n / 2; i > 0; i--)   //我们从最底层开始往上构建堆  n/2 对应最
低层的最左边的树   即i 表示这轮进行构建的树的根，i 一步步减小，即堆的构建一步
步往上
     {
         HeapAdjust(k, i, n);  //建大顶堆的函数
```

```
    }                   //一整个大顶堆构建完成
    for (i = n; i > 1; i--)
    {
        swap(k, 1, i);          //将大顶堆的根（即此时下标为1 的数，也即最大的数）与数组的最后一个数互换，即此时数组最后一个数是全数组最大的
        HeapAdjust(k, 1, i - 1);  //在前n-1 个数中重新构建堆  此时要构建的堆的根为下标为1
    }
}

//归并排序
/*2 路归并排序：
一个元素为n 的数组，将其分为n 个长度为1 的子序列，再将这些序列两两归并，得到n/2 个长度为2 的子序列，再两两归并，直到得到长度为n 的序列*/
  //递归实现

void merging(int* list1, int list1_size, int* list2, int list2_size)  //对传入的list1 与list2 进行归并,并把最后结果放到list1 里面
{
    int i, j, k,m;  //i,j 分别作为list1 和list2 的指针，k 作为temp 的指针
    int temp[10]; //用来存最后的结果，10 是瞎编的，反正就长度是整个数组长度
    i = j = k = 0;
    while (i < list1_size && j < list2_size)
    {
        if (list1[i] < list2[j])   //挨个往过比list1 和list2 的元素，而且由于list1 和list2 此时已经是排好序的，所以先是两个最小的里面取最小的往temp 里存，再依次往后取小，然后一个个往temp 里存
        {
            temp[k++] = list1[i++];
        }
        else
        {
            temp[k++] = list2[j++];
        }
    }
    //如果比完了之后list1 或者list2 还有剩下的元素，则继续依次往temp 里放
    while (i < list1_size)
    {
        temp[k++] = list1[i++];
    }
    while (j < list2_size)
    {
        temp[k++] = list2[j++];
    }
    for (m = 0; m < (list1_size + list2_size); m++)   //把最后结果存到list1 里面
    {
        list1[m] = temp[m];
```

```
    }
}
void MergeSort(int k[], int n)  //拆
{
    if (n > 1)
    {   //将k分为两半
        int* list1 = k;   //list1表示list的左半部分
        int list1_size = n / 2;  //左半部分长度
        int* list2 = k + n / 2;   //list2表示list的右半部分
        int list2_size = n - list1_size;

        MergeSort(list1, list1_size);   //继续把左半部分往下拆
        MergeSort(list2, list2_size);    //继续把右半部分往下拆

        merging(list1, list1_size, list2, list2_size);  //拆完了，现在开始一层层归并
    }

}
//迭代实现归并
/*递归是某种物理意义上真的把它先分成一个一个小序列然后再一对一对处理直到合成大序列
而迭代则可以直接第一次长度为2先分别排序，第二次长度为4进行2与2排序，第三次长度为8进行4与4排序*/
void MergeSort(int k[], int n)
{
    int i, left_min, left_max, right_min, right_max;
    int* temp = (int*)malloc(n * sizeof(int));

    for (i = 1; i < n; i *= 2)  //每一轮子序列长度为i
    {
        for (left_min = 0; left_min < n - 1; left_min = right_max)  //固定子序列长度下，开始一对一对处理合并子序列
        {
            right_min = left_max = left_min + i;  //初始化本轮要处理的两个子序列的指针left是左边那个序列的指针，right是右边那个序列的指针
            right_max = left_max + i;
            if (right_max > n)
            {
                right_max = n;
            }
            int next = 0;   //用于temp指针
            while (left_min < left_max && right_min < right_max)
            {
                if (k[left_min] < k[right_min])
                {
                    temp[next++] = k[left_min++];
                }
```

```
                else
                {
                    temp[next++] = k[right_min++];
                }
            }
            while (left_min < left_max)
            {
                k[--right_min] = k[--left_min];
            }
            while (next > 0)
            {
                k[--right_min] = temp[--next];   //最终结果存到右边序列里
            }
        }
    }
}

//快速排序
/*每一次排序设立一个基准点，比这个基准大的全放右边，比这个基准小的全放左边，再继续选基准，递归*/
void swap(int k[], int low, int high)
{
    int temp;
    temp = k[low];
    k[low] = k[high];
    k[high] = temp;
}

int Partition(int k[], int low, int high) //由于表中大于或小于基准点的数据多少不一定，故而不是基准点不动，其他点照着基准点所在位置为分界线而挪动，而是动态的，大了的数据往右放，小了的数据往左放，最后才能确定下来基准点也就是分界线到底在哪
{
    int point;
    point = k[low];        //k[low]的值作为基准
    while (low < high)  //low 与 high 是用于遍历的指针
    {
        while (low < high && k[high] >= point)
        {
            high--;
        }
        swap(k, low, high);  //交换
        while (low < high && k[low] <= point)
        {
            low++;
        }
        swap(k, low, high);
    }
    return low;  //返回此时基准点应该在表中的位置
```

```
}
void QSort(int k[], int low, int high)  //从low到high进行排序
{
    int point;  //首先设定一个分界值，通过该分界值将数组分成左右两部分
    if (low < high)
    {
        point = Partition(k,low,high);  //将大于或等于分界值的数据集中到数组右边，小于分界值的数据集中到数组的左边。此时，左边部分中各元素都小于或等于分界值，而右边部分中各元素都大于或等于分界值，使用此函数实现此对k的划分操作
        //然后，左边和右边的数据可以独立排序。对于左侧的数组数据，又可以取一个分界值，将该部分数据分成左右两部分，同样在左边放置较小值，右边放置较大值。右侧的数组数据也可以做类似处理
        QSort(k, low, point - 1);
        QSort(k, point + 1, high);
    }
}
void QuickSort(int k[], int n)
{
    QSort(k, 0, n - 1);
}

//快速排序的优化
//1.优化选取基准点
/*三数取中法
* 在Partition中,取point为k[low]，k[m]，k[high]的中间值
* int m=low+(high-low)/2;
* if(k[low]>k[high])
*     swap(k,low,high);
* if(k[m]>k[high])
*     swap(k,m,high);
* if(k[m]>k[low])
*     swap(k,m,low);
* point=k[low];
*/
//2.优化不必要的交换
/*把交换改成赋值，作为基准的point的值则一直由point存着，直到找到最终基准所在位置之后直接赋给这个位置
* while (low < high)
    {
        while (low < high && k[high] >= point)
        {
            high--;
        }
        k[low]=k[high];  //交换
        while (low < high && k[low] <= point)
        {
            low++;
        }
```

```
            k[high]=k[low];
            swap(k, low, high);
        }
        k[low]=point;
    */
    //3.优化小数组时的排序方案
    /*当数组比较小时用直接插入排序，大数组再用快速排序
    * 加个判断
    */
    //4.优化递归操作
    /*使用尾递归
    * 如果一个函数中所有递归形式的调用都出现在函数的末尾，我们称这个递归函数是尾递归的。
    */
```