信息科学技术学院

课程（实习）设计

课程实习名称：数据结构实习

专　　业：计算机科学与技术

学　　号：2351610105

学 生 姓 名：方泽宇

成　　绩：

批 改 日 期：

指 导 教 师：

完成任务汇总表

序号	模块	任务
1	线性结构	约瑟夫环
2	线性结构	纸牌游戏
3	线性结构	一元多项式计算
4	线性结构	迷宫求解
5	线性结构	八皇后问题
6	线性结构	运动会分数统计
7	线性结构	订票系统
8	线性结构	文章编辑
9	树形结构	二叉树
10	树形结构	哈夫曼编码
11	树形结构	并查集
12	图形结构	最小代价生成树
13	图形结构	拓扑排序及关键路径
14	图形结构	交通咨询系统
15	查找技术	查找技术
16	排序技术	排序技术

目录（自动生成）

（宋体，五号字）

1. 约瑟夫环

任务

一堆猴子都有编号，编号是 1，2，3 . . . n ,这群猴子(n 个)按照 1 ~ n 的顺序围坐一 圈，从第 1 开始数，每数到第 m 个，该猴子就要离开此圈，这样依次下来，直到圈中只剩 下最后一只猴子，则该猴子为大王。请设计算法编写程序输出为大王的猴子的编号。

（1）需求分析

分析总结逻辑结构：

这是一个约瑟夫环问题，n 只猴子构成了一个环，每数到 m 只猴子就让这只猴子出圈，然后再从下一只猴子开始数，一直数到只剩下 1 只猴子。这样的问题可以使用递归进行解决，也可以不使用递归，循环数数即可。需要初始化猴子的编号，将 1-n 放在循环链表当中。由于使用头插法建表的速度比较快，在设计算法过程中可以从 n 到 1 倒序使用头插法建表。从链表的第一个结点开始，每次向前移动 m-1 个单位，并将该节点删除即可。当循环链表中只剩下一个结点的时候，说明只剩下了一只猴子，可以直接输出结果。

分析总结运算集合：

使用头插法向单链表中插入一个元素

删除单链表中的一个元素

（2）概要设计

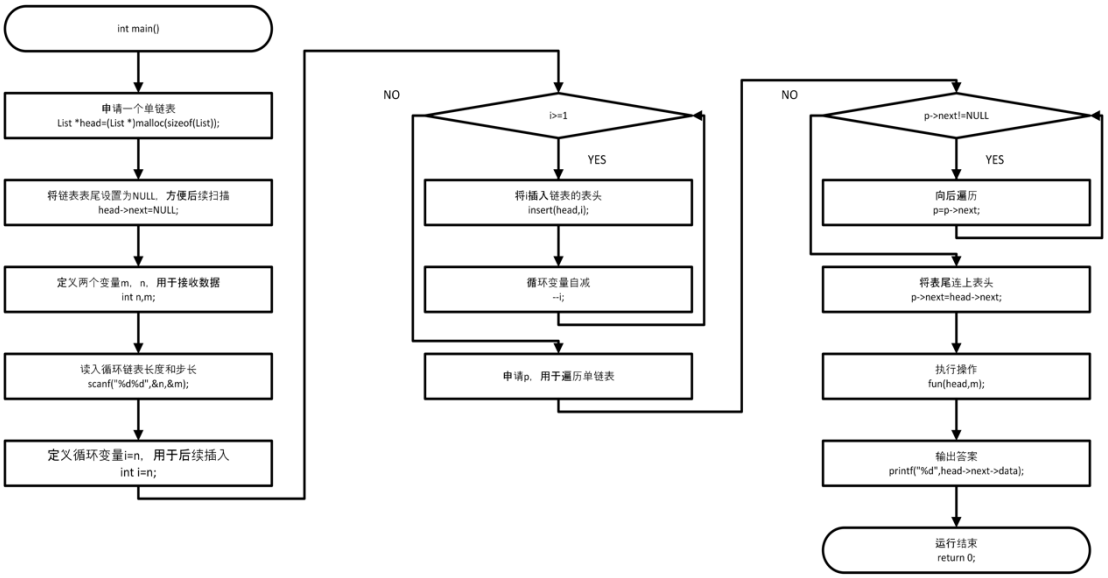
存储结构设计：

采用循环链表的数据结构，让 head 指针能够指向第一支猴子的位置，完成建表以后，将单链表最后的 NULL 指针改成指向头节点的指针。在删除过程中如果删除了头节点，那么就需要将 head 指针的指向发生改变。需要设计至少 2 个链表的操作函数，insert(*L,x)使用头插法将 x 插入链表，fun(*L,m)执行删除操作，直到链表中只剩下一个元素。

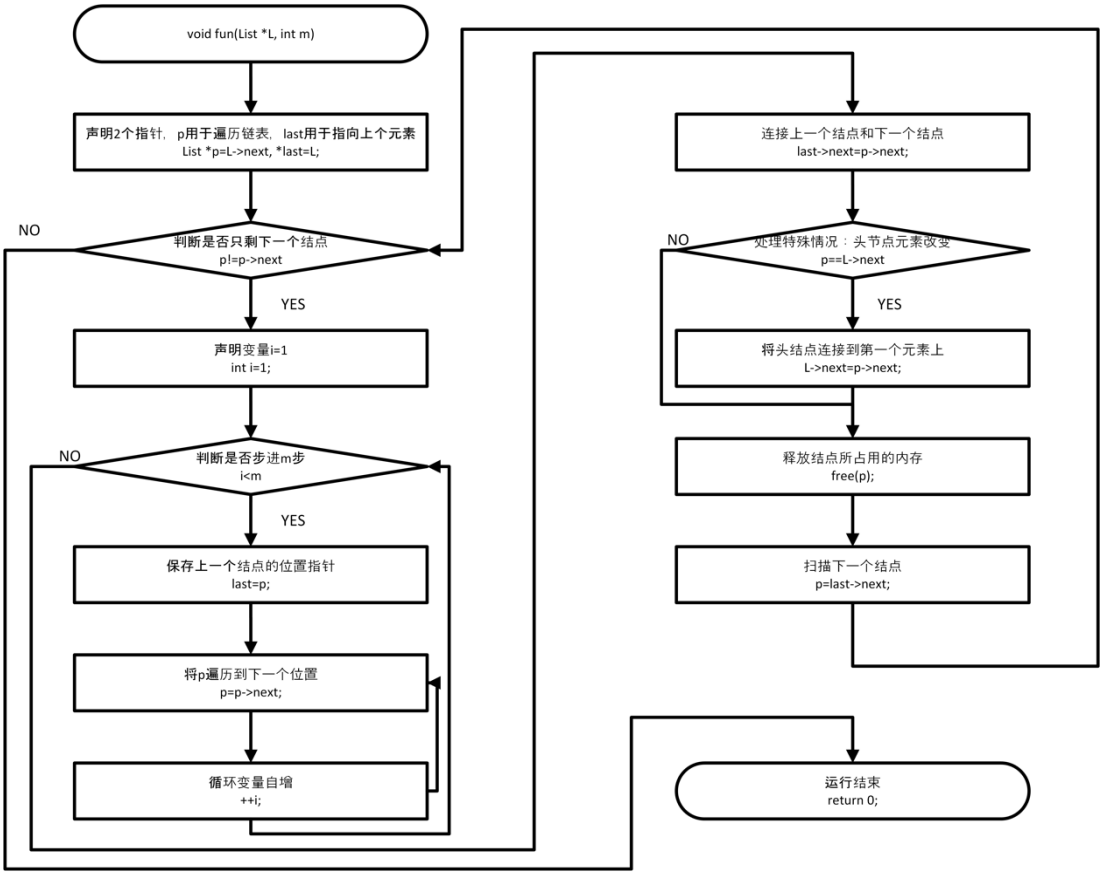
算法设计（流程图）：

采用循环链表的方法，每次经过 m-1 个结点，删除掉这个结点即可。

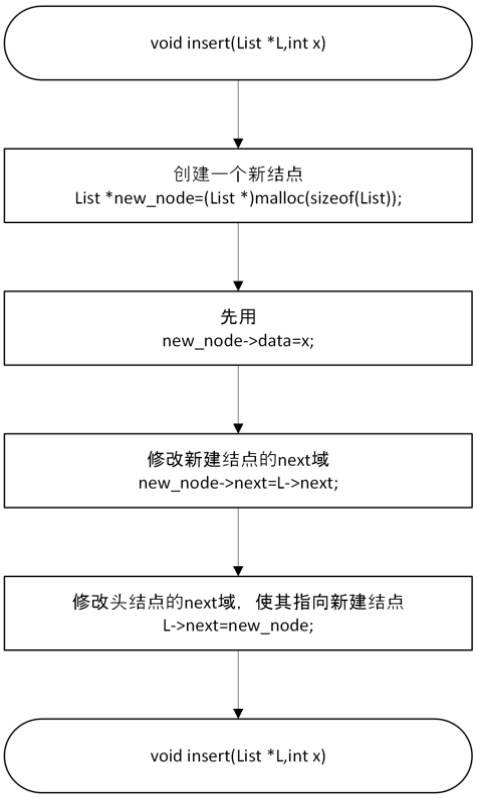
Int main()的函数实现：



Void fun(List *L,int m)的函数实现：



Insert(List *L,int x)的函数实现：



（3）详细设计

源代码（注释）

```
#include<stdio.h>
#include<stdlib.h>//因为要使用 malloc，所以需要 include<stdlib.h>
typedef struct List
{
    int data;
    struct List *next;
}List;//创建链表的结点结构题
void insert(List *L,int x)//使用头插法插入数字 x
{
    List *new_node=(List *)malloc(sizeof(List));//申请一个空间，插入链表的表
头
    new_node->data=x;//先用
    new_node->next=L->next;//后改
    L->next=new_node;//修改头指针指向的位置
}
void fun(List *L, int m)
{
    List *p=L->next, *last=L;
    while (p!=p->next)// 仅剩一个节点时停止
    {
        for (int i = 1; i < m; ++i)
        {
            last=p;//保存上一个结点的位置，方便后面修改 next 指针
            p=p->next;//向后扫描一个结点
        }
        last->next=p->next;
        if(p==L->next) L->next=p->next;//处理一个特殊情况，如果表头所指向的元
素发生改变
        free(p);//释放内存，节省内存占用
        p=last->next;//下一个结点
    }
}
int main()
{
    List *head=(List *)malloc(sizeof(List));//申请一个单链表
    head->next=NULL;//将头节点的 next 域赋值为 NULL，方便后面扫描单链表的表尾
    int n,m;
    scanf("%d%d",&n,&m);
    for(int i=n;i>=1;--i) insert(head,i);//倒着将数据使用头插法将数据插入单链
表
    List *p=head->next;//遍历单链表
    while(p->next!=NULL) p=p->next;//访问链表中的最后一个元素
    p->next=head->next;//将单链表的尾结点连上头节点所连接的结点，创建循环链表
```

```
    fun(head,m);//执行删除操作
    printf("%d",head->next->data);//输出结果
    return 0;
}
```

（4）调试分析

样例 1:

输入 10，3，输出 4

```
10 3
4%
```

以下是对样例 1 输出结果的分析：

第一次删除：从编号为 1 的猴子开始数，删除第 3 只猴子。即删除编号为 3 的猴子。剩下的猴子是：1，2，4，5，6，7，8，9，10。

第二次删除：从编号为 4 的猴子开始数，再数到第 3 只猴子，删除编号为 6 的猴子。剩下的猴子是：1，2，4，5，7，8，9，10。

第三次删除：从编号为 7 的猴子开始数，删除编号为 9 的猴子。剩下的猴子是：1，2，4，5，7，8，10。

第四次删除：从编号为 10 的猴子开始数，删除编号为 2 的猴子。剩下的猴子是：1，4，5，7，8，10。

第五次删除：从编号为 4 的猴子开始数，删除编号为 7 的猴子。剩下的猴子是：1，4，5，8，10。

第六次删除：从编号为 8 的猴子开始数，删除编号为 1 的猴子。剩下的猴子是：4，5，8，10。

第七次删除：从编号为 4 的猴子开始数，删除编号为 8 的猴子。剩下的猴子是：4，5，10。

第八次删除：从编号为 10 的猴子开始数，删除编号为 5 的猴子。剩下的猴子是：4，10。

第九次删除：从编号为 10 的猴子开始数，删除编号为 10 的猴子。剩下的猴子是：4。

样例 2:

输入 30 7，输出 23

```
30 7
23%
```

（5）小结

在本次约瑟夫环实习的过程中，关于仅剩一个结点的判断，我进行了很长时间的调试，我尝试过很多判断条件均没有用。后来我尝试手动更改循环，发现问题出现在如果 head 指向的元素被删除，head->next 没有被及时更新，于是对代码进行了修改。

2. 纸牌游戏

任务

编号为 1-52 张牌，正面向上，从第 2 张开始，以 2 为基数，是 2 的倍数的牌翻一次， 直到最后一张牌;然后，从第 3 张开始，以 3 为基数，是 3 的倍数的牌翻一次，直到最后一张牌;然后从第 4 张开始，以 4 为基数，是 4 的倍数的牌翻一次，直到最后一张牌;.....直 到以 52 为基数的牌翻过，这时正面向上的牌有哪些?请设计算法编写程序输出最终正面向 上的纸牌的编号。

（1）需求分析

分析总结逻辑结构：

这是一个翻牌问题，因为一张牌有 2 面，如果一张牌被翻了奇数次，那么这张牌的状态和初始状态不一致，如果一张牌被翻了偶数次，那么这张牌的状态会和初始状态相一致。所以我们只需要依次扫描这些牌，统计被翻牌的次数即可，如果被翻了奇数次，那么这些牌是反面朝上，如果被翻了偶数次，那么就是反面朝上。初始化一个顺序表，清空顺序表中的数据项，从 2 到 52 依次将 n 到倍数翻牌，最后统计被翻牌的次数即可。

分析总结运算集合：

初始化顺序表(数组)

翻牌(将该数组元素+1)

判断该牌被翻的次数是否是偶数

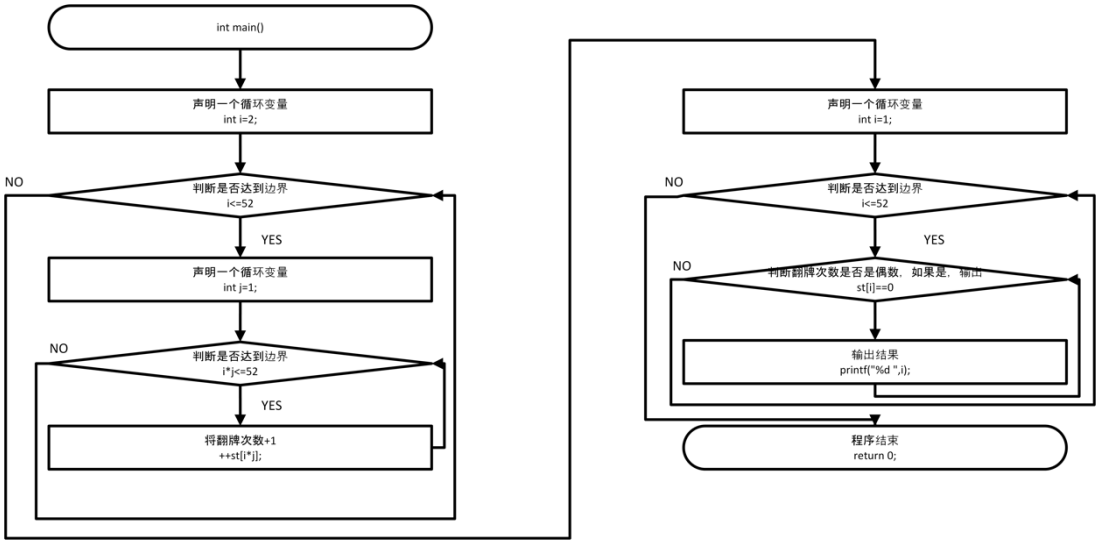
（2）概要设计

存储结构设计

因为需要很多次随机访问，为加快程序运行速度，采用顺序表的方式存储。

算法设计（流程图）

Int mian()的函数实现：



（3）详细设计

源代码（注释）

```
#include<stdio.h>
int st[53];
int main()
{
    for(int i=2;i<=52;++i)//以 1-52 为基数
        for(int j=1;i*j<=52;++j)//遇到基数的整数倍就翻牌一次
            ++st[i*j];//记录翻牌的次数
    for(int i=1;i<=52;++i)//检查被翻牌的牌
        if(st[i]%2==0)//如果被翻了偶数次，那么还是正面朝上
            printf("%d ",i);//输出结果
    return 0;//程序结束
}
```

（4）调试分析

输出结果：1 4 9 16 25 36 49

```
1 4 9 16 25 36 49
```

以下是对输出结果的分析：

1 没有被翻牌，所以正面朝上

4 被 2 和 4 翻牌 2 次，所以正面朝上

9 被 3 和 9 翻牌 2 次，所以正面朝上

16 被 2、4、8、16 翻牌 4 次，所以正面朝上

25 被 5 和 25 翻牌 2 次，所以正面朝上

36 被 2、3、4、6、9、12、18、36 翻牌 8 次，所以正面朝上

49 被 7 和 49 翻牌 2 次，所以正面朝上

类似地，

所有的质数均只会被自己翻牌 1 次，所以反面朝上

8 被 2、4、8 翻牌 3 次，所以反面朝上

10 被 2、5、10 翻牌 3 次，所以反面朝上

12 被 2、3、4、6、12 翻牌 5 次，所以反面朝上

等等……

因此，只有完全平方数的翻牌次数为偶数次，因此输出结果均为完全平方数

（5）小结

在本次实习中，我们使用顺序表来存储纸牌被翻牌的次数和状态，并使用随机访问的形式来更新每一个结点，最后顺序访问输出结果。通过对结果的观察不难发现，所有的答案均为完全平方数，因此我们可以直接输出所有的完全平方数，也可以得到本题的答案。

3. 一元多项式计算

任务

设计合适的存储结构，完成一元多项式的相关运算。

要求:(1)能够按照指数降序排列建立并输出多项式;(2)能够完成两个多项式的相加、相 减，并将结果输出。

（1）需求分析

分析总结逻辑结构：

题目中所说到的降序排序，其实是按照降序排序的规则，对一个表达式中变量次数的排序。因为这道题中经常会用到对线性表的插入和删除的操作，所以采用单链表的形式存储会更加便捷。

分析总结运算集合：

初始化线性表(链表)

基于值查找，并插入链表元素

将两个有序链表归并

对串的处理：从串中分离指数和系数

（2）概要设计

存储结构设计：

因为需要多次在线性表中插入元素，所以采用链式存储结构

算法设计（流程图）：

初始化结果链表：创建一个空的结果链表 result，用来存储两个多项式相加的结果。

定义指针遍历链表：pa 指向第一个多项式链表 A 的头节点，pb 指向第二个多项式链表 B 的头节点。这两个指针用来遍历两个多项式链表。

遍历两个链表：使用 while 循环同时遍历两个链表 A 和 B。在每次循环中，比较 pa 和 pb 所指向节点的指数 power 大小，分以下几种情况处理：情况 1：pa->power > pb->power。如果 A 中当前节点的指数大于 B 中当前节点的指数，将 A 中当前节点插入结果链表 result 中。然后，pa 指针向后移动，指向下一个节点。情况 2：pa->power < pb->power 如果 B 中当前节点的指数大于 A 中当前节点的指数，将 B 中当前节点插入结果链表 result 中。然后，pb 指针向后移动，指向下一个节点。情况 3：pa->power == pb->power 如果 A 和 B 中当前节点的指数相同，则将它们的系数相加，生成一个新的节点插入结果链表 result 中。然后同时移动 pa 和 pb 指针，分别指向下一个节点。

处理剩余项：当一个链表遍历结束后（pa 或 pb 为 NULL），继续处理另一个链表中的剩余项：将剩余的 pa 链表中的项插入结果链表 result。将剩余的 pb 链表中的项插入结果链表 result。

返回结果链表：addPoly 函数返回存储加法结果的链表 result。

（流程图实在太复杂了，这题就不画了，因为实习要求是画 4 个题目）

（3）详细设计

源代码（注释）

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
typedef struct PolyNode
{
    int factor;//系数
    int power;//指数
    struct PolyNode *next;
    //next 指针，指向下一个结点
}PolyNode, *PolyList;
//创建结点类型和结点指针类型
void insertTerm(PolyList *L, int factor, int power)
{
    PolyNode *newNode = (PolyNode *)malloc(sizeof(PolyNode));
    //创建一个新的结点
    newNode->factor = factor;
    //传入新结点的系数
    newNode->power = power;
    //传入新结点的指数
    newNode->next = NULL;
    //新建的结点没有下一个指针位置
    if (*L==NULL||(*L)->power<power)
    //如果说发现需要插入的链表为空，或者是新插入的这个结点的次数高于原有的最高次数
    {
        newNode->next=*L;
        //那么直接将该结点插入头节点即可
        *L=newNode;
        //修改头指针的位置
    }
    else
    {
        PolyNode *p=*L;
        //新建一个 p 指针，用于遍历链表
        while(p->next&&p->next->power>power) p=p->next;
        //扫描链表，直到次数相等或者次数刚好大于这个点
        if (p->power == power)
        //如果恰巧停下来的时候，次数相等，那么合并
        {
            p->factor+=factor;
            //将系数直接相加
            free(newNode);
            //释放这个已经合并同类项的点
            if (p->factor == 0)
```

```
            //如果次数干好为 0，那么把合并以后的点也直接释放掉
            {
                PolyNode *temp = p->next;
                //删除操作，先保存后面一个结点的指针，先连接后删除
                free(p);
                *L = temp;
            }
        }
        else
        {
            newNode->next = p->next;
            //否则，将新的结点插入链表
            p->next = newNode;
        }
    }
}
void print(PolyList L)
//这个函数用于输出表达式
{
    if (L == NULL)
    //如果表达式保存的链表为空，说明表达式不存在，直接输出 0 即可
    {
        printf("0\n");
        return;
    }
    PolyNode *p = L;
    //声明一个指针类型 p，用于遍历链表
    while (p)
    {
        if (p->factor>0&&p!=L) printf("+");
        //在非头节点的情况底下，如果系数大于 0，那么需要输出正号
        if (p->power==0) printf("%d",p->factor);
        //如果次数正号为 0，那么直接输出系数即可
        else if(p->power==1&&p->factor==1) printf("x");
        //如果系数和次数都恰好为 1，那么直接输出 x
        else if(p->power==1&&p->factor==-1) printf("-x");
        //如果次数为 1，系数为-1，那么直接输出-x
        else if(p->power==1&&p->factor!=1&&p->factor!=-1)
            printf("%dx",p->factor);
            //如果次数恰好为 1，系数无特殊输出系数和 x
        else if(p->factor==1) printf("x%d", p->power);
        //如果系数恰好为 1，且次数无特殊，那么输出 x 和次数
        else if(p->factor==-1) printf("-x%d",p->power);
        //如果系数为-1，次数无特殊，那么输出-x 和次数
```

```
        else printf("%dx%d", p->factor, p->power);
        //否则，输出系数、x、次数
        p = p->next;
        //遍历下一个结点
    }
    printf("\n");
    //输出换行，输出完成
}
PolyList addPoly(PolyList A, PolyList B)
//新建加法运算
{
    PolyList result=NULL;
    //首先先清空答案 result 链表
    PolyNode *pa=A,*pb=B;
    //创建两个指针类型变量 pa，pb，用于遍历两个链表
    while (pa&&pb)
    //如果在 pa 和 pb 都没有走到头掉情况底下，需要比较两个链表的次数
    {
        if (pa->power>pb->power)
        //如果说 a 链表的次数大于 b 链表的次数，那么需要先计算 a 链表里值
        {
            insertTerm(&result,pa->factor,pa->power);
            //调用 insert 函数将 a 链表中 pa 指向的系数和次数插入 result 链表中
            pa=pa->next;
            //pa 走向下一个结点
        }
        else if(pa->power<pb->power)
        //同样，如果说 b 链表的次数大于 a 链表的次数，那么需要先计算 b 链表里值
        {
            insertTerm(&result,pb->factor,pb->power);
            //调用 insert 函数将 b 链表中 pb 指向的系数和次数插入 result 链表中
            pb=pb->next;
            //pb 走向下一个结点
        }
        else
        //最后只剩下了两者次数相等的情况
        {
            int sum=pa->factor+pb->factor;
            //这种情况下，把两个系数相加即可
            if(sum!=0) insertTerm(&result,sum,pa->power);
            //如果说相加得到的结果为 0，也没有必要自己给自己找麻烦，直接跳过插
入操作
            //如果相加得到的结果不为 0，那么直接将这样一个系数和次数插入到
result 链表当中
```

```
            pa=pa->next;
            pb=pb->next;
            //两个指针移向下位
        }
    }
    while (pa)
    //收尾，如果 pb 已经走到了链表的表尾，将 pa 走到底就行了
    {
        insertTerm(&result,pa->factor,pa->power);
        pa=pa->next;
    }
    while (pb)
    //收尾，如果 pa 已经走到了链表的表尾，将 pb 走到底就行了
    {
        insertTerm(&result,pb->factor,pb->power);
        pb=pb->next;
    }
    return result;
    //把答案返回给主调函数
}
PolyList subPoly(PolyList A,PolyList B)
//这是一个减法的过程，实现原理和上面是类似的
{
    PolyList result=NULL;
    //首先定义一个结果链表，这个链表原始是空的，因为里面什么也没有
    PolyNode *pa=A,*pb=B;
    //定义两个用于循环的指针 pa 和 pb
    while (pa&&pb)
    //如果指向两个链表的指针都没有走到表尾
    {
        if (pa->power>pb->power)
        //如果 pa 指向的结点的次数高于 pb 指向结点的次数
        {
            insertTerm(&result,pa->factor,pa->power);
            //将 pa 的结点直接插入链表中即可
            pa = pa->next;
            //pa 走向下一个结点
        }
        else if (pa->power<pb->power)
        //相反的，如果 pb 指向的结点的次数高于 pa 指向的结点的次数
        {
            insertTerm(&result,-pb->factor,pb->power);
            //那么 pb 指向的结点，取相反数以后插入 result 链表中即可
            //注意，这里和上面的加法不一样，因为 b 是被减数，所以需要取负数
```

```
            pb = pb->next;
            //pb 走向下一个结点
        }
        else
        //最后只剩下了一种情况，就是两者的次数恰好相等
        {
            int ans=pa->factor-pb->factor;
            //那么将两者的系数相减
            if(ans!=0) insertTerm(&result,ans,pa->power);
            //如果两者系数相减正号为 0，也没必要白费功夫了，直接跳过即可
            //将结点插入 result 当中
            pa=pa->next;
            pb=pb->next;
            //pa，pb 均可以走向下一个结点
        }
    }
    while (pa)//收尾工作
    {
        insertTerm(&result,pa->factor,pa->power);
        pa = pa->next;
        //将 pa 剩下的结点全部插入到 result 中
    }
    while (pb)
    {
        insertTerm(&result,-pb->factor,pb->power);
        pb=pb->next;
        //将 pb 剩下的结点全部插入道 result 中
    }
    return result;
    //将结果返回给主调函数
}
void add(PolyList *L)
//这是一个读入字符串分离字符建立链表的字符串操作函数
{
    char ch[100];
    //本程序可以接受的最大表达式长度为 100
    scanf("%s",&ch);
    //将字符串读入程序中
    int factor=0,sign=1,power=0;
    //定义三个变量，factor 用于指示系数
    //sign 用于指示符号
    //power 用于指示次数
    for(int i=0;i<strlen(ch);++i)
    //依次遍历整个字符串
```

```
    {
        if(ch[i]=='-')
        //如果遇到了负号，将系数取相反数
        {
            sign=-1;
            continue;
        }
        else if(ch[i]=='+') continue;
        //如果遇到了加号，说明不需要处理
        else if(ch[i]=='x')
        //如果遇到了 x，那就说明遇到了系数和次数的分界点
        {
            if(factor==0) factor=1;
            //如果系数为 0，那就说明 x 前面什么都没有，那系数就是 1
            for(++i;ch[i]!='+'&&ch[i]!='-';++i)
            //通过一个循环读取次数
            {
                power*=10;
                power+=ch[i]-'0';
            }
            if(power==0&&(ch[i]=='+'||ch[i]=='-')) power=1;
            //如果还没读就结束了，说明次数也是 1
            insertTerm(L,sign*factor,power);
            //将这个结点插入道链表当中
            --i;
            //多读了一个符号位，说不定后面需要特殊处理，返回去重新处理
            power=0;factor=0;sign=1;//将标志还原
        }
        else
        {
            factor*=10;
            factor+=ch[i]-'0';
            //这两个代码用来读取系数
        }
    }
    if(factor!=0) insertTerm(L,factor*sign,0);
    //收尾工作，如果最后没有读到 x，那就说明存在 0 次方项，将 0 次方项插入链表
}
int main()
{
    PolyList A=NULL;//新建两个链表
    PolyList B=NULL;
    add(&A);//读入 A
    add(&B);
```

```
    PolyList C = addPoly(A,B);//C 是答案，AB 相加
    printf("A+B:");
    print(C);
    C=subPoly(A,B);//C 是答案，AB 相减
    printf("A-B:");
    print(C);
    return 0;//程序运行结束
}
```

（4）调试分析

样例 1 输入：3x3+2x2+1，3x3+4

输出：A+B:6x3+2x2+5，A-B:2x2-3

样例分析：在第一个表达式中，x^3 项为 3，与第二个表达式相一致，所以在结果中，没有 x^3 项。而其他项均存在，显然输出答案为 6x3+2x2+5，A-B:2x2-3

```
3x3+2x2+1
3x3+4
A+B:6x3+2x2+5
A-B:2x2-3
```

样例 2 输入：0，0

输出：A+B:0，A-B:0

```
0 0
A+B:0
A-B:0
```

样例 3 输入：0，-3x5-6

输出：A+B:3x5+6，A-B:-3x5-6

```
0
-3x5-6
A+B:-3x5-6
A-B:3x5+6
```

（6）小结

在本次实习过程中，我遇到了很多的困难，首先是在字符串处理方面。需要从段字符串文本中分离出系数、x 和次数，然后再插入链表中进行运算。通过这次实习，我更加深入理解了链表的链式存储的原理，以及两个有序链表归并的问题。

4. 迷宫求解

任务

（1）需求分析

分析总结逻辑结构：

采取数组坐标系，从入口(1,1)走到出口(n,m)所可能走过的路径。可以通过搜索和遍历来解决这样一个问题。遍历有 2 种常见的方式，一种是深度优先搜索(DFS，Depth First Search)，另一种是广度优先搜索(BFS,Breadth First Search)。前者是基于栈来实现的，后者是基于队列来实现的。为了在这道题中体现栈和队列两种数据结构，本题采用广度优先搜索遍历的方式搜索，其特点是可以搜索到从起点到终点的最短路径。回溯路径的方法是，到达终点之后逐个返回寻找路径即可。

分析总结运算集合：

初始化队列

将数据从队尾加入队列

从队首取出元素，并删除该元素

初始化栈

将数据加入栈，并更新栈顶

将栈顶元素去除，并删除栈顶元素

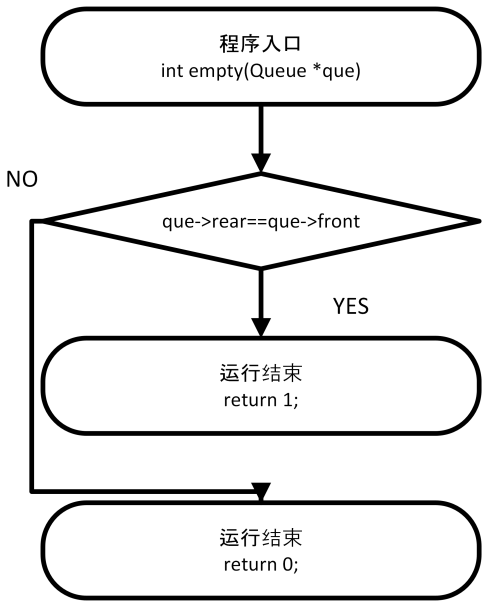
（2）概要设计

存储结构设计：

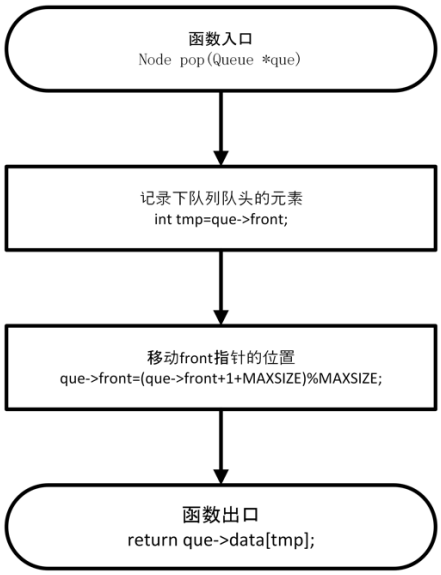
采用队列和栈两种数据结构来完成运算，通过队列完成广度优先搜索(BFS)的搜索过程，通过栈对答案进行逆序输出。现将终点入栈，再依次将所经过的点加入栈中，然后再逐个出栈输出。

算法设计（流程图）：

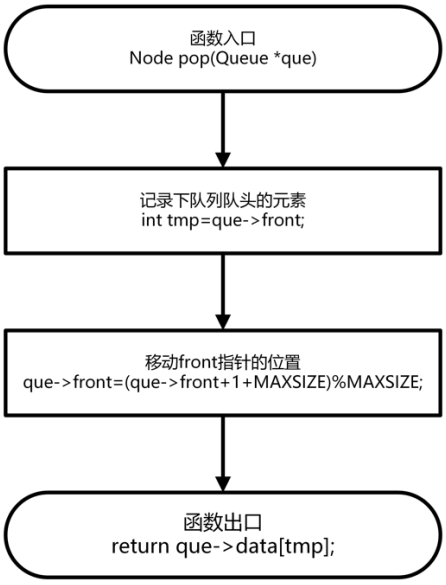
Int empty(Queue *que)的函数实现：



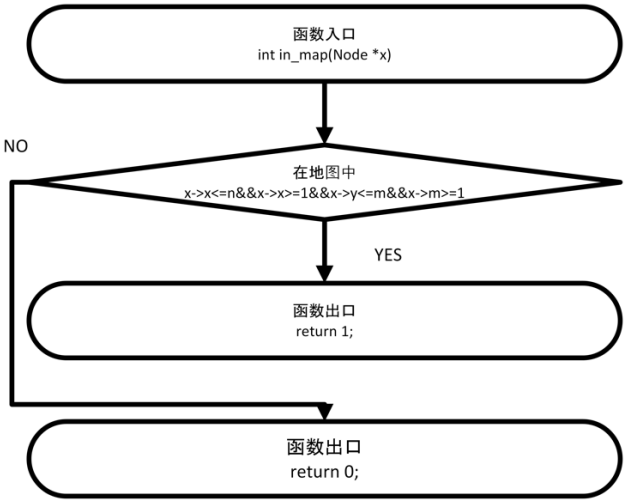
Node pop(Queue *que)的函数实现：



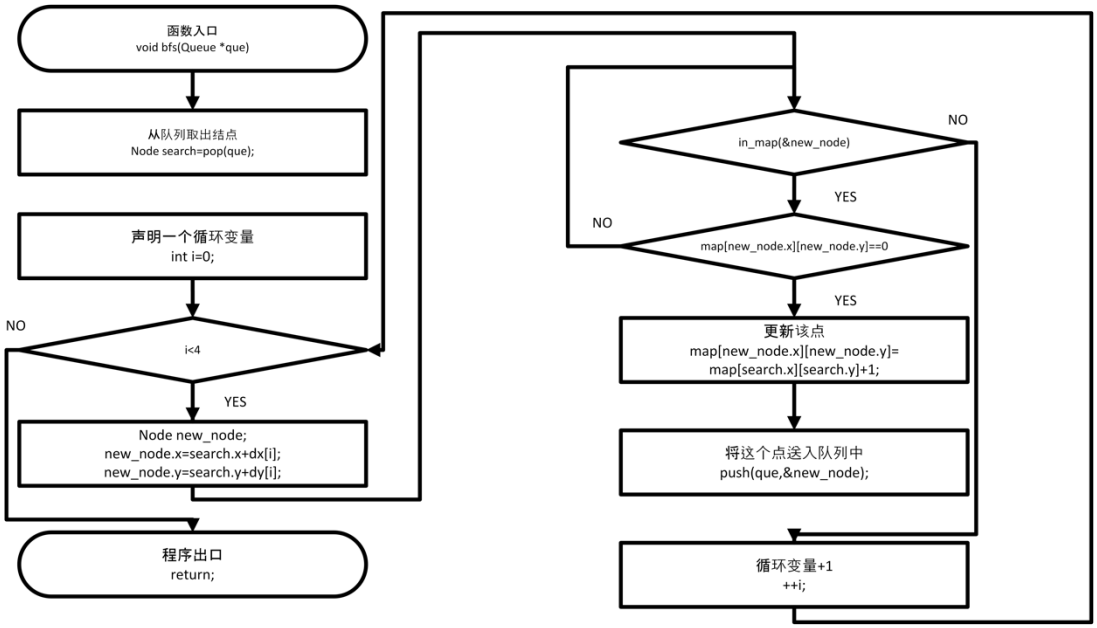
Void push(Queue *que,Node *x)的函数实现：



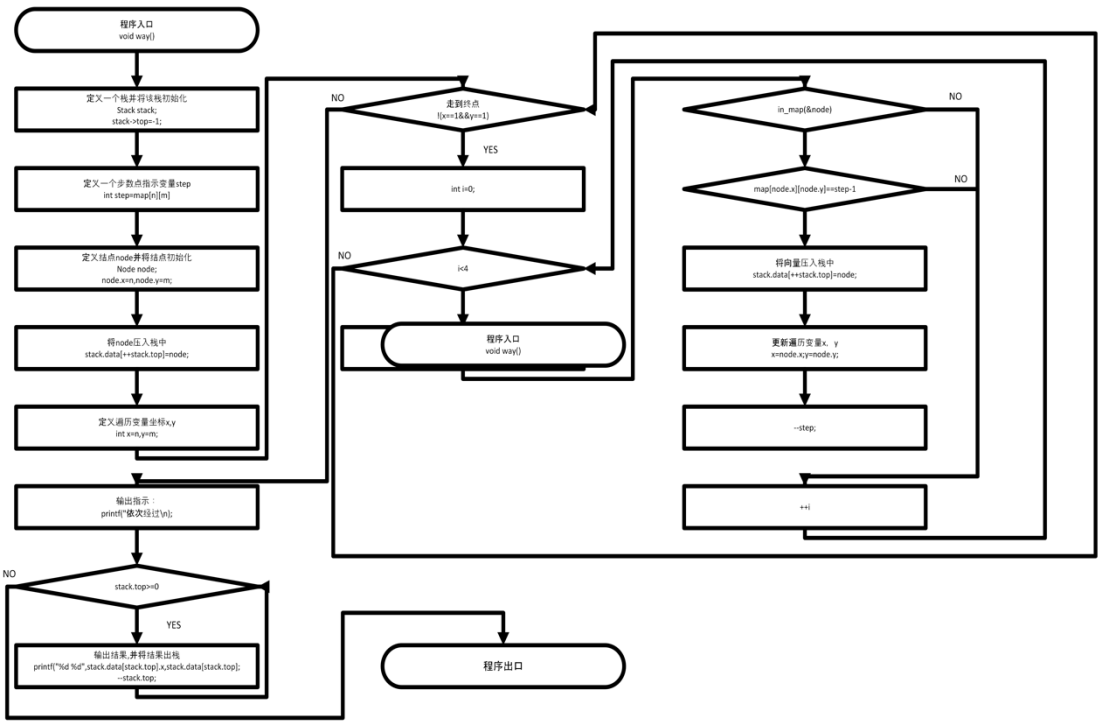
Int in_map(Node *x)的函数实现：



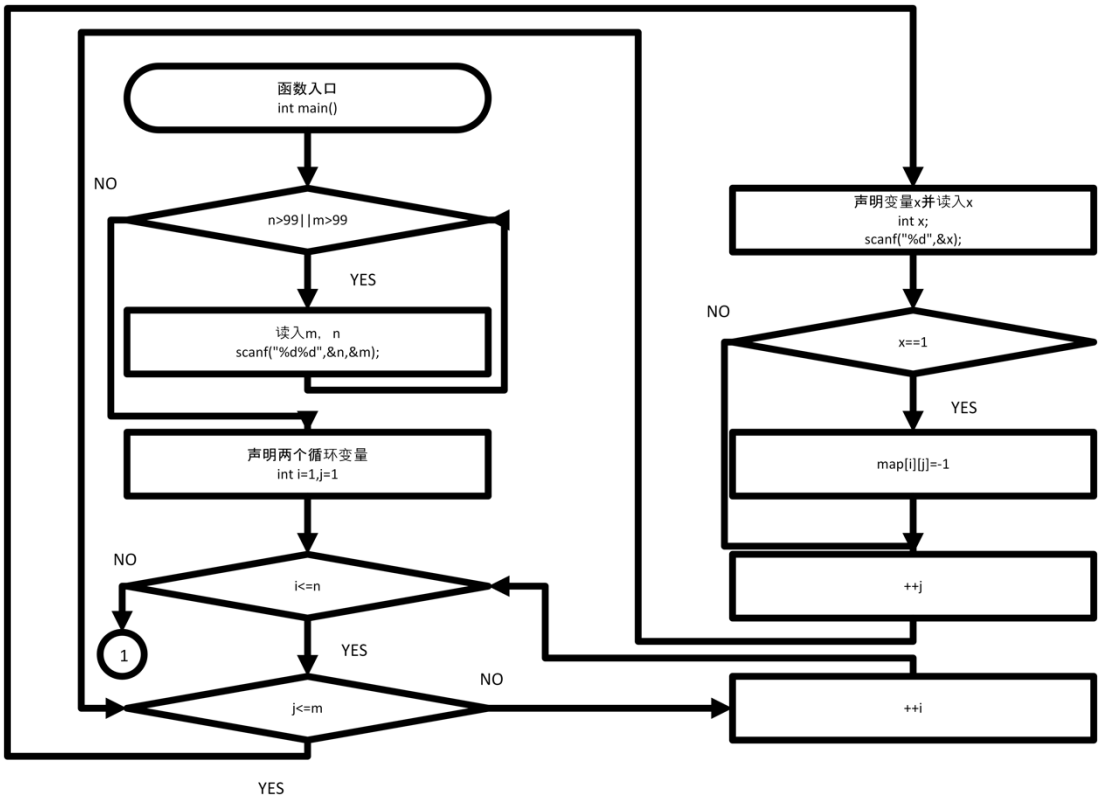
Void bfs(Queue *que)的函数实现：

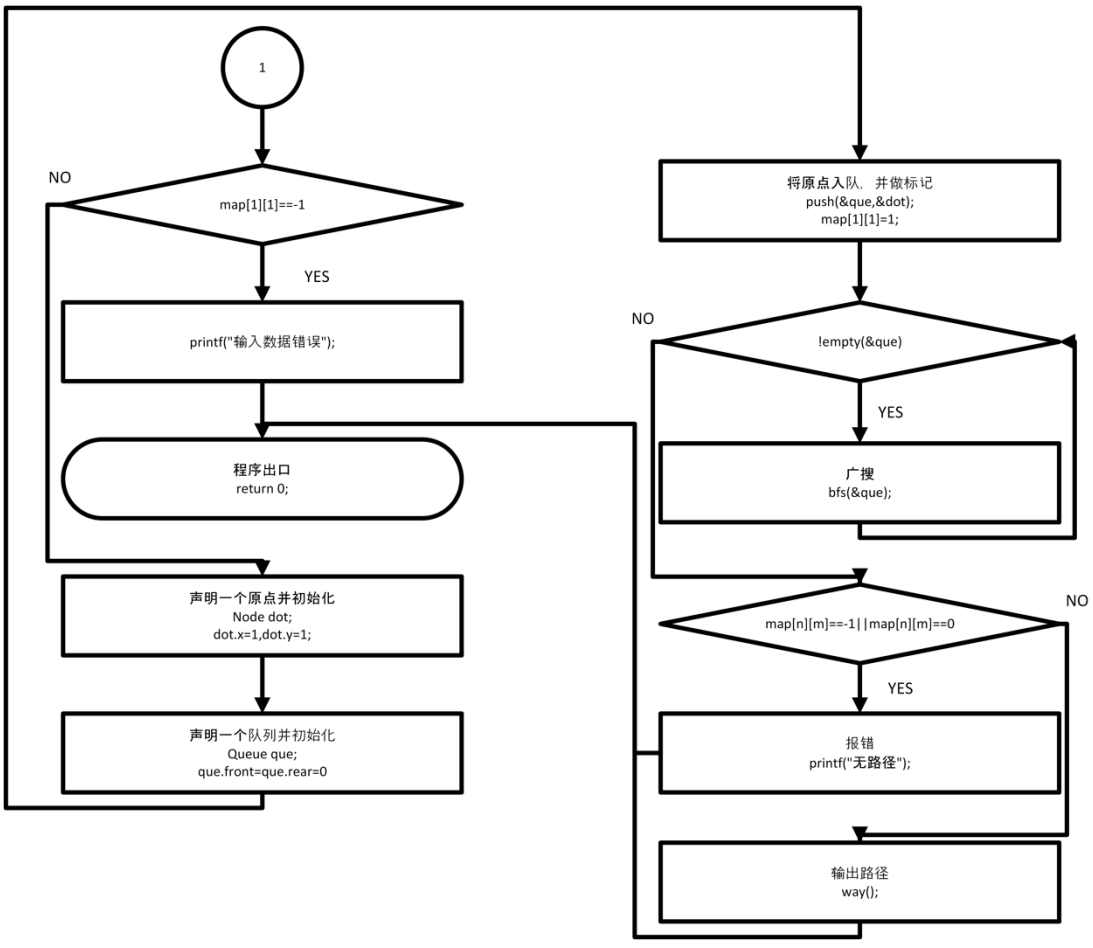


Void way()的函数实现：



Int main()的函数实现：





（3）详细设计

源代码（注释）

```
#include<stdio.h>
const int MAXSIZE=100;
//用一个全局的常量去定义地图的最大大小
int map[MAXSIZE][MAXSIZE],n=0x3f3f3f3f,m=0x3f3f3f3f;
//定义一个地图，最大的大小为 100*100
const int dx[4]={0,0,1,-1};
const int dy[4]={1,-1,0,0};
//定义两个向量，方便以后想下一步走
typedef struct Node
{
    int x,y;
}Node;
//为每一个点的坐标定义一个结点类型
typedef struct Queue
{
    Node data[MAXSIZE*MAXSIZE];
    int front,rear;
}Queue;
//定义一个队列，用于存放广度优先搜索(bfs)
typedef struct Stack
{
    Node data[MAXSIZE*MAXSIZE];
    int top;
}Stack;
//定义一个栈，用于逆序输出路径
int empty(Queue *que)
{
    if(que->rear==que->front) return 1;
    else return 0;
}
//判断队列是否为空(用处：如果队列为空说明 bfs 结束)
Node pop(Queue *que)
{
    int tmp=que->front;
    //保存临时变量，指向需要输出的结点的位置
    que->front=(que->front+1)%MAXSIZE;
    //front 指针向前移动
    return que->data[tmp];
}
//将队列中的最前面的一个元素返回，并且删除掉这个元素
void push(Queue *que,Node *x)
{
```

```
    que->data[que->rear].x=x->x;
    que->data[que->rear].y=x->y;
    //将结点的两个域分别更新
    que->rear=(que->rear+1)%MAXSIZE;
    //rear 指针后移
}
int in_map(Node *x)
{
    if(x->x<=n&&x->x>=1&&x->y<=m&&x->y>=1) return 1;
    //如果没有超出地图边界，那么返回 1，否则返回 0
    else return 0;
}
void bfs(Queue *que)
//这是一个广度优先搜索遍历的算法，当然我们也可以对这张图进行深度优先搜索
//但是 bfs 搜索的是最短路径，这道题我只输出一种最短路径
{
    Node search=pop(que);
    //从队列里取出一个结点
    for(int i=0;i<4;++i)
    {
        Node new_node;
        new_node.x=search.x+dx[i];
        new_node.y=search.y+dy[i];
        //使用向量，更新待搜索的结点
        if(in_map(&new_node))
        //如果没有超出边界
        {
            if(map[new_node.x][new_node.y]!=0) continue;
            //如果这个地方是障碍物，或者已经更新过到达这个点的最短路径，那么直
接 continue
            map[new_node.x][new_node.y]=map[search.x][search.y]+1;
            //如果满足要求，则这个点到(1,1)点的最短路径就是其上一个结点+1
            push(que,&new_node);
            //将这个结点送入队列中
        }
    }
}
void print()
{
    for(int i=1;i<=n;++i)
    {
        for(int j=1;j<=m;++j)
        {
            printf("%d ",map[i][j]);
```

```
            //依次输出每一个点到(1,1)的最短路，输出-1 代表该点无法到达(1,1)
        }
        printf("\n");
    }
}
void way()
{
    Stack stack;
    //定义一个栈
    //因为 bfs 到终点以后，可以倒着推出路径，但是正着无法推出，所以需要用一个栈来将结果逆序
    stack.top=-1;
    //初始化栈
    int step=map[n][m];
    //将最终步数放入临时变量 step 中，上一次走过的结点的步数一定是 step-1
    Node node;
    //创建一个新的结点
    node.x=n;node.y=m;
    stack.data[++stack.top]=node;
    //将结果放入栈中
    int x=n,y=m;
    while(!(x==1&&y==1))
    {
        for(int i=0;i<4;++i)
        {
            node.x=x+dx[i];
            node.y=y+dy[i];
            //通过向量更新出上一步有可能走过的点
            if(in_map(&node))
            //这个点是否在地图内
            {
                if(map[node.x][node.y]==step-1)
                //如果步数正好是 step-1，则说明这是路径上的一点
                {
                    stack.data[++stack.top]=node;
                    //将这个点加入栈
                    x=node.x;
                    y=node.y;
                    --step;
                    //循环，继续寻找上一个结点
                    break;
                }
            }
        }
```

```
    }
    printf("依次经过\n");
    //输出结果
    while(stack.top>=0)
    //依次出栈
    {
        printf("%d %d\n",stack.data[stack.top].x,stack.data[stack.top].y);
        --stack.top;
    }
}
int main()
{
    while(n>99||m>99) scanf("%d%d",&n,&m);
    //输入地图的大小，保证输入的大小是在一个合法的范围内，防止超出地图的边界
    for(int i=1;i<=n;++i)
    {
        for(int j=1;j<=m;++j)
        {
            int x;
            scanf("%d",&x);
            if(x==1) map[i][j]=-1;
            //输入地图，0 代表可以走，1 代表是墙壁，但是 1 和第一步冲突，为了防
止冲突，读入数据以后改成-1，代表走不了
        }
    }
    if(map[1][1]==-1)
    //如果起点为-1，墙壁
    {
        printf("输入数据错误");
        //出错返回
        return 0;
    }
    Node dot;
    //新建一个点，这个点是原点
    Queue que;
    //新建一个队列，用于存储 bfs 序列
    que.front=que.rear=0;
    //初始化队列，方式 segmentation fault
    dot.x=dot.y=1;
    push(&que,&dot);
    //将原点推入队列中
    map[1][1]=1;
    while(!empty(&que)) bfs(&que);
    if(map[n][m]==-1||map[n][m]==0)
```

```
    {
        printf("无路径\n");
        //最后一个点无法到达，出错返回
        return 0;
    }
    //print();
    way();
    return 0;
}
```

（4）调试分析

样例 1:

输入：

4 5
0 1 1 1 1
0 0 1 1 1
0 0 1 1 1
1 0 0 0 0

输出：

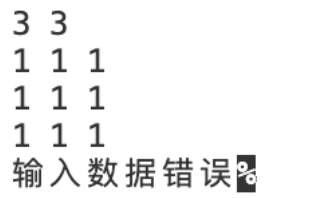
依次经过
1 1
2 1
3 1
3 2
4 2
4 3
4 4
4 5

4 5
0 1 1 1 1
0 0 1 1 1
0 0 1 1 1
1 0 0 0 0
依次经过
1 1
2 1
3 1
3 2
4 2
4 3
4 4
4 5

解释：通过 bfs 扩展得到的结果如下：

1 -1 -1 -1 -1
2 3 -1 -1 -1

3 4 -1 -1 -1
-1 5 6 7 8
其中-1 表示无法到达的点，正整数表示从坐标(1,1)走到该点所需要的步数(设走到点(1,1)需要 1 步)
样例 2:
输入：
9 6
0 0 0 0 0 0
1 1 1 1 1 0
0 0 0 0 0 0
0 1 1 1 1 1
0 0 0 0 0 0
1 1 1 1 1 0
0 0 0 0 0 0
0 1 1 1 1 1
0 0 0 0 0 0
输出：
依次经过
1 1，1 2，1 3，1 4，1 5，1 6，2 6，3 6，3 5，3 4，3 3，3 2，3 1，4 1，5 1，5 2，5 3，5 4，5 5，5 6，6 6，7 6，7 5，7 4，7 3，7 2，7 1，8 1，9 1，9 2，9 3，9 4，9 5，9 6
（为排版方便，将左右的换行改成了逗号”,”）
样例 3(错误样例 1):
输入：
3 3
1 1 1
1 1 1
1 1 1
输出：
输入数据错误



样例 4(错误样例 2):
输入：
5 4
0 1 1 1
0 0 1 1
1 1 0 1
1 1 1 1
1 1 0 0
输出：

无路径

```
5 4
0 1 1 1
0 0 1 1
1 1 0 1
1 1 1 1
1 1 0 0
无路径
```

（5）小结

在本次实习的过程中，我采用 BFS 广度优先搜索的方法遍历图，并逆序将结果输出到一个栈中，再利用栈先进后出的特性输出了迷宫所走过的路径。其中加上了异常处理的程序。一开始程序运行中出现段错误(segmentation fault)，后来经过仔细的检查发现错误出现在队列没有被初始化上，把队列初始化后这样的错误就没有再次发生。这段代码只能找到从坐标原点到右下角的最短路径，并不能输出所有的路径，并且时间复杂度较高。为了输出所有的路径，我们可以尝试改用基于栈/递归的 DFS 深度优先搜索算法，基于堆优化的启发式搜索 A*搜索算法，这段代码仍然有很多改进的空间。

5. 八皇后问题

任务

国际西洋棋棋手马克斯·贝瑟尔于 1848 年提出在 8X8 格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种 摆法。请设计算法编写程序解决。

（1）需求分析

分析总结逻辑结构：

这道题需要在一个 8x8 的地图上生成 8 个皇后，让他在每一行，每一列和每一个对角线上均有且只有一个皇后，对于这样的一个问题，我们可以对 64 个位置上生成 8 个皇后，并且一次判断。但是这样做的话时间复杂度会来到 2 的 64 次方的级别，显然超出了计算机最大的计算规模。因此需要对这样的一个计算进行剪枝。其中一个剪枝的方法，就是首先生成一个长度为 8 的全排列，分别表示在每一列上皇后所在的行数，这样可以在生成的过程中就完成了对不在同一行、不在同一列上的判断，完成了剪枝。而生成全排列的算法，就是 dfs。生成好全排列以后，只需要对对角线进行判断。通过观察我们发现，在同一主对角线上的元素，其横纵坐标的差值是一定的，而在同一副对角线上的元素，其横纵坐标之和是一定的。因此我们只需要对横纵坐标的和、差进行判重即可。

分析总结运算集合：

Dfs 生成全排列

计算和差进行判重

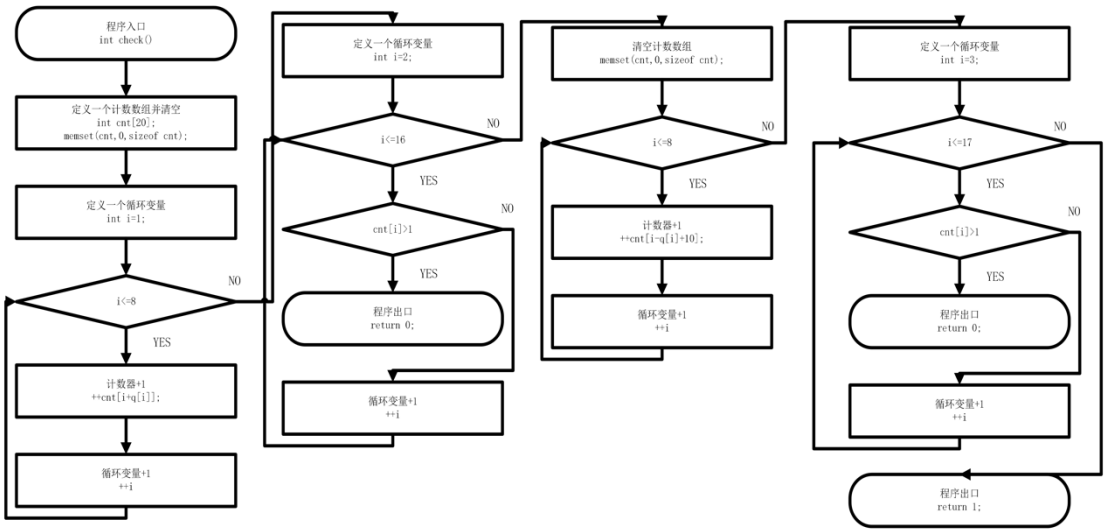
（2）概要设计

存储结构设计：

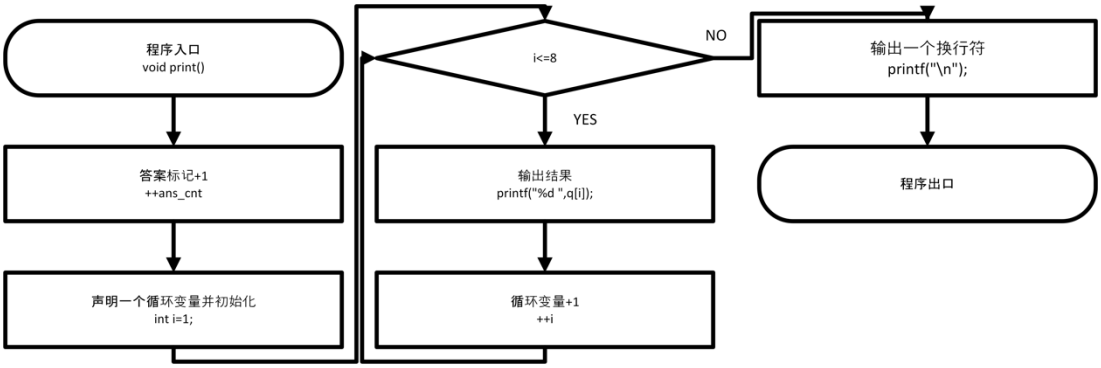
通过一个长度为 8 的一维数组生成全排列，代表八皇后的位置。通过递归的方式模拟出一个栈，完成 dfs 的过程。通过一个长度为 14 的一位数组对八皇后的位置进行判重，但是对于差运算，需要计算一个偏移值。

算法设计（流程图）：

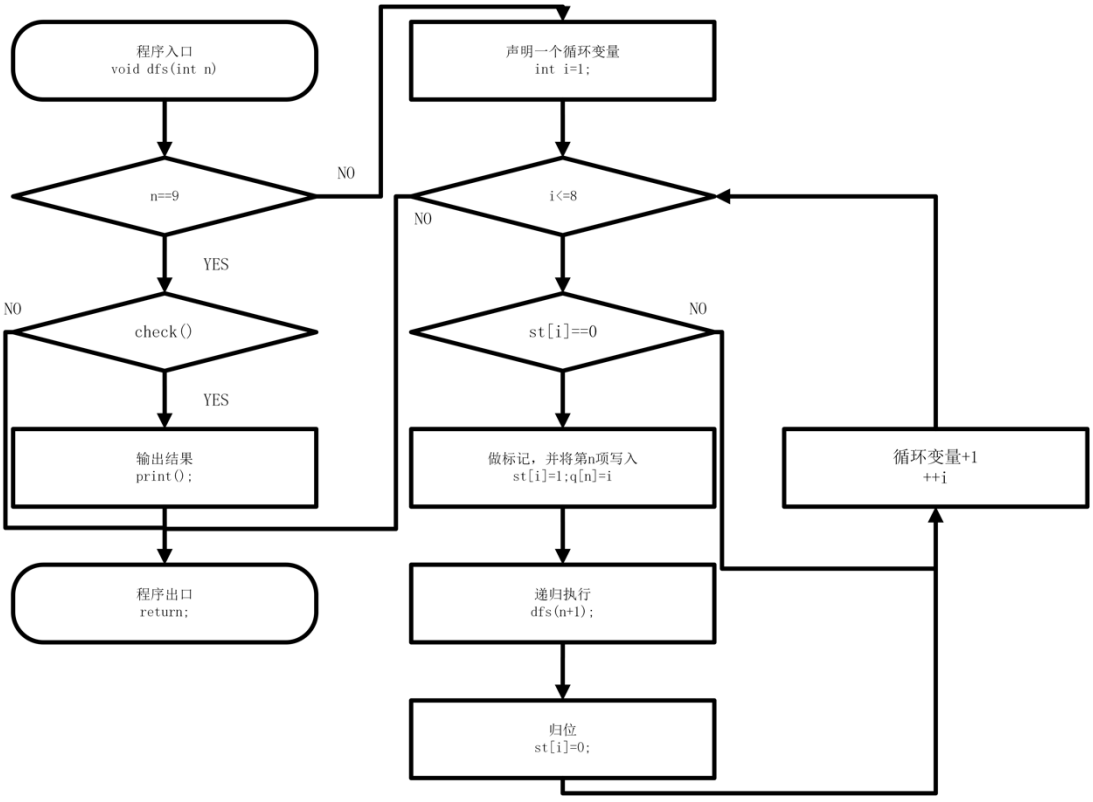
Int check()函数的实现过程：



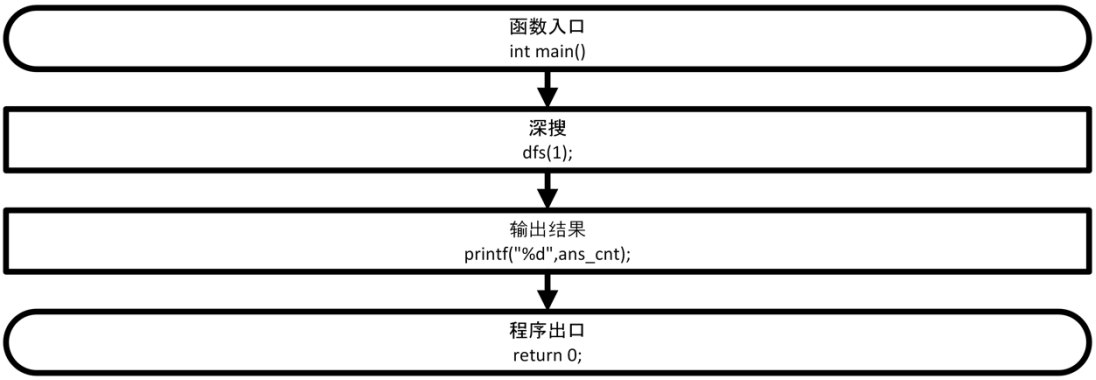
Void print()函数的实现过程：



Void dfs(int n)的函数实现过程：



Int main()函数的实现过程：



（3）详细设计

源代码（注释）

```
#include<stdio.h>
#include<string.h>
int q[9],st[9],ans_cnt;
//q 数组用于存放答案，st 代表这个数字是否被使用过，ans_cnt 统计答案的个数
int check()
{
    //先统计序号和 q 中值之和
    int cnt[20];
    memset(cnt,0,sizeof cnt);
    for(int i=1;i<=8;++i) ++cnt[i+q[i]];
    //统计每一个皇后所在位置的横坐标和纵坐标之和
    for(int i=2;i<=16;++i)
        if(cnt[i]>1) return 0;
    //如果存在两个数字的横坐标和纵坐标之和相等
    //那就说明存在两个或者更多的皇后出现在同一个副对角线上
    //先统计序号和 q 中值之差(带偏移值)
    memset(cnt,0,sizeof cnt);
    for(int i=1;i<=8;++i) ++cnt[i-q[i]+10];
    //统计每一个皇后坐在位置横坐标和纵坐标位置之差(算偏移值)
    for(int i=3;i<=17;++i)
        if(cnt[i]>1) return 0;
    //如果存在两个数字其横坐标和纵坐标点差相等
    //那就说明存在两个或更多的皇后出现在同一个主对角线上
    return 1;
}
void print()
{
    ans_cnt++;//print 答案说明答案正确
    for(int i=1;i<=8;++i) printf("%d ",q[i]);
    //将答案数组 q 里的元素全部输出
    printf("\n");
    //输出结束
}
void dfs(int n)
{
    if(n==9)
    //如果已经搜索到了第九层，说明全排列已经生成完毕
    {
        if(check()) print();
        //如果满足要求，那么输出结果
        return;//递归退层
    }
```

```
    for(int i=1;i<=8;++i)
    //生成全排列
    {
        if(st[i]==0)
        {
            st[i]=1;
            q[n]=i;
            //如果这个数字没有被使用过，那么使用
            dfs(n+1);
            st[i]=0;
            //归位
        }
    }
}
int main()
{
    dfs(1);
    //从第一位开始生成
    printf("%d",ans_cnt);
    //输出结果的总数
    return 0;
}
```

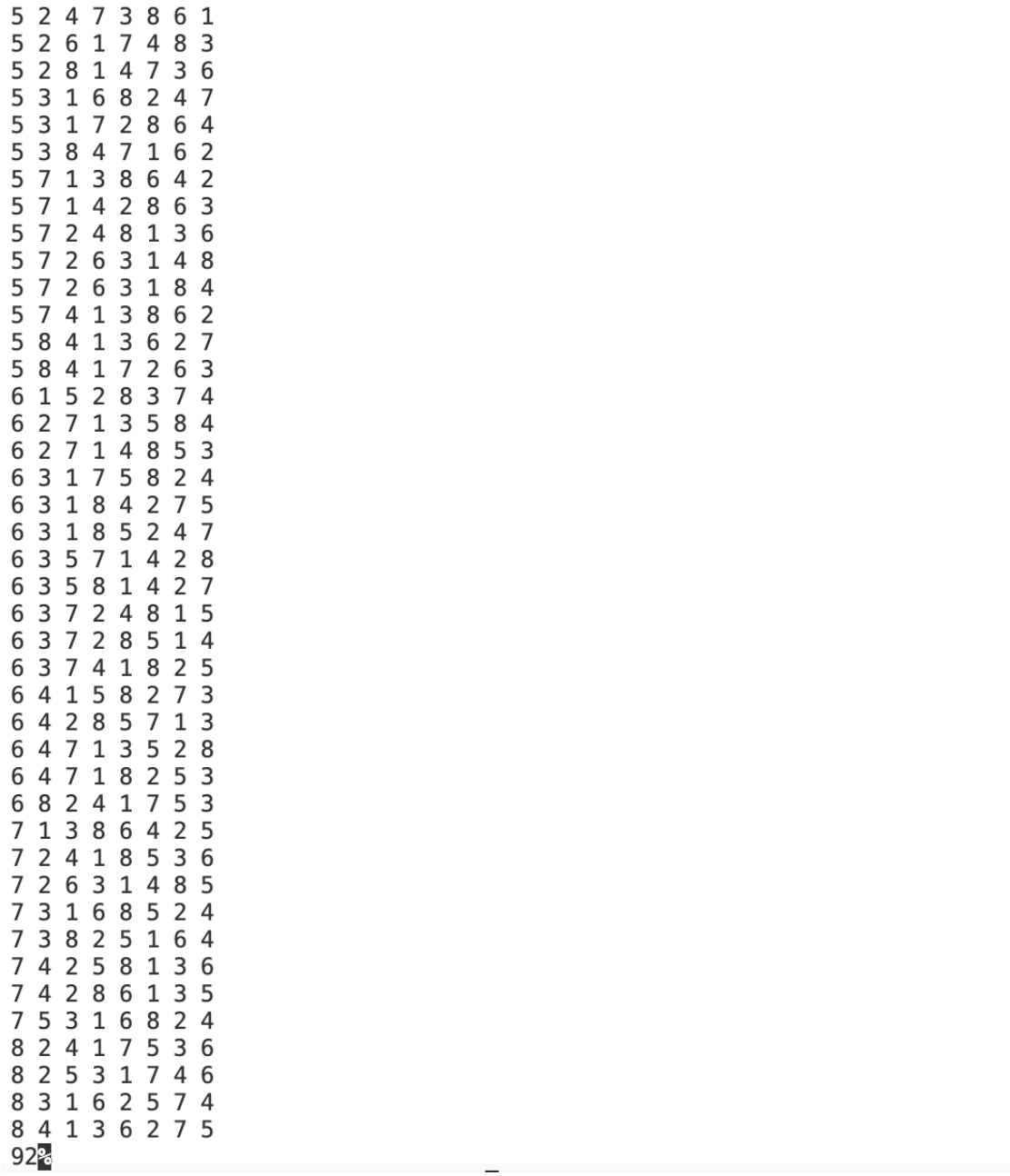
（4）调试分析

输出结果如下：

1 5 8 6 3 7 2 4	3 6 2 7 5 1 8 4	4 6 8 3 1 7 5 2
1 6 8 3 7 4 2 5	3 6 4 1 8 5 7 2	4 7 1 8 5 2 6 3
1 7 4 6 8 2 5 3	3 6 4 2 8 5 7 1	4 7 3 8 2 5 1 6
1 7 5 8 2 4 6 3	3 6 8 1 4 7 5 2	4 7 5 2 6 1 3 8
2 4 6 8 3 1 7 5	3 6 8 1 5 7 2 4	4 7 5 3 1 6 8 2
2 5 7 1 3 8 6 4	3 6 8 2 4 1 7 5	4 8 1 3 6 2 7 5
2 5 7 4 1 8 6 3	3 7 2 8 5 1 4 6	4 8 1 5 7 2 6 3
2 6 1 7 4 8 3 5	3 7 2 8 6 4 1 5	4 8 5 3 1 7 2 6
2 6 8 3 1 4 7 5	3 8 4 7 1 6 2 5	5 1 4 6 8 2 7 3
2 7 3 6 8 5 1 4	4 1 5 8 2 7 3 6	5 1 8 4 2 7 3 6
2 7 5 8 1 4 6 3	4 1 5 8 6 3 7 2	5 1 8 6 3 7 2 4
2 8 6 1 3 5 7 4	4 2 5 8 6 1 3 7	5 2 4 6 8 3 1 7
3 1 7 5 8 2 4 6	4 2 7 3 6 8 1 5	5 2 4 7 3 8 6 1
3 5 2 8 1 7 4 6	4 2 7 3 6 8 5 1	5 2 6 1 7 4 8 3
3 5 2 8 6 4 7 1	4 2 7 5 1 8 6 3	5 2 8 1 4 7 3 6
3 5 7 1 4 2 8 6	4 2 8 5 7 1 3 6	5 3 1 6 8 2 4 7
3 5 8 4 1 7 2 6	4 2 8 6 1 3 5 7	5 3 1 7 2 8 6 4
3 6 2 5 8 1 7 4	4 6 1 5 2 8 3 7	5 3 8 4 7 1 6 2
3 6 2 7 1 4 8 5	4 6 8 2 7 1 3 5	5 7 1 3 8 6 4 2

5 7 1 4 2 8 6 3
5 7 2 4 8 1 3 6
5 7 2 6 3 1 4 8
5 7 2 6 3 1 8 4
5 7 4 1 3 8 6 2
5 8 4 1 3 6 2 7
5 8 4 1 7 2 6 3
6 1 5 2 8 3 7 4
6 2 7 1 3 5 8 4
6 2 7 1 4 8 5 3
6 3 1 7 5 8 2 4
6 3 1 8 4 2 7 5
6 3 1 8 5 2 4 7
6 3 5 7 1 4 2 8
6 3 5 8 1 4 2 7
6 3 7 2 4 8 1 5
6 3 7 2 8 5 1 4
6 3 7 4 1 8 2 5
6 4 1 5 8 2 7 3
6 4 2 8 5 7 1 3
6 4 7 1 3 5 2 8
6 4 7 1 8 2 5 3
6 8 2 4 1 7 5 3
7 1 3 8 6 4 2 5
7 2 4 1 8 5 3 6
7 2 6 3 1 4 8 5
7 3 1 6 8 5 2 4
7 3 8 2 5 1 6 4
7 4 2 5 8 1 3 6
7 4 2 8 6 1 3 5
7 5 3 1 6 8 2 4
8 2 4 1 7 5 3 6
8 2 5 3 1 7 4 6
8 3 1 6 2 5 7 4
8 4 1 3 6 2 7 5
92

运行结果的截屏：



关于输出结果的解释(以 1 5 8 6 3 7 2 4)为例：
这样一个输出结果代表在第 1 列的第 1 行有一个皇后，在第 2 列的第 5 行有一个皇后。如果用 1 代表皇后，用 0 代表其他的点，那么这样一个输出结果可以表示为：
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
通过观察我们可以发现，上图中，在任何一列，任何一行，任何一条对角线上，都不存在两个皇后。

（5）小结
在本次实习的过程中，八皇后问题的存储结构设计让我印象尤其深刻。如果直接存储八皇后问题的图，则需要对行列和对角线均进行判断，但是如果先生成全排列，再再全排列中进行判断，只需要判断对角线即可，大大减少了程序的计算量，提高运算效率。

6. 运动会分数统计

任务

参加运动会有 n 个学校，学校编号为 1.....n。比赛分成 m 个男子项目，和 w 个女子 项目。项目编号为男子 1.....m，女子 m+1.....m+w。不同的项目取前五名或前三名积分; 取前五名的积分分别为:7、5、3、2、1，前三名的积分分别为:5、3、2;哪些取前五名或 前三名由学生自己设定。(n<=20 ， m<=20)

要求:(1)可以输入各个项目的前三名或前五名的成绩;(2)能统计各学校总分;(3)可以按学校编号、学校总分、男女团体总分排序输出;(4)可以按学校编号查询学校某个项目的情况;可以按项目编号查询取得前三或前五名的学校。

（1）需求分析

分析总结逻辑结构：

分析总结运算集合：

（2）概要设计

存储结构设计：

算法设计（流程图）：

（3）详细设计

源代码（注释）

（4）调试分析

……

（5）小结

7. 订票系统

任务

请根据以下要求，设计航班信息、订票信息的存储结构，设计程序完成订票系统的相 关功能。

要求:

(1)录入:可以录入航班情况，数据可以存储在一个数据文件中，数据结构自定;

(2)查询:可以查询某个航线的情况(如，输入航班号，查询起降时间，起飞抵达城市，航班票价，票价折扣，确定航班是否满仓);可以输入起飞抵达城市，查询飞机航班情况;

(3)订票:(订票情况可以存在一个数据文件中，结构自己设定)可以订票，如果该航班已 经无票，可以提供相关可选择航班;

(4)退票: 可退票，退票后修改相关数据文件;客户资料有姓名，证件号，订票数量及航 班情况，订单要有编号;

(5)修改航班信息:当航班信息改变可以修改航班数据文件。

（1）需求分析

分析总结逻辑结构：

分析总结运算集合：

（2）概要设计

存储结构设计：

算法设计（流程图）：

（3）详细设计

源代码（注释）

（4）调试分析

……

（5）小结

8. 文章编辑

任务

静态存储一页文章，每行最多不超过 80 个字符，共 N 行。统计文中所出现的英文字 母的个数、数字的个数、空格的个数、总字数等。

要求:

(1)分别统计出其中英文字母数和空格数及整篇文章总字数;

(2)统计某一字符串在文章中出现的次数，并输出该次数;

(3)删除某一子串，并将后面的字符前移。

(4)输入数据的形式和范围:可以输入大写、小写的英文字母、任何数字及标点符号。

(5)输出形式:分行输出用户输入的各行字符;分 4 行输出"全部字母数"、"数字个数"、" 空格个数"、"文章总字数";输出某一单词在文章中出现的次数;输出删除某一字符串后的 文章。

（1）需求分析

分析总结逻辑结构：

分析总结运算集合：

（2）概要设计

存储结构设计：

算法设计（流程图）：

（3）详细设计

源代码（注释）

（4）调试分析

……

（5）小结

9. 树形结构

任务

建立二叉树，并实现二叉树的先序、中序、后序、层序遍历，求二叉树的宽度，统计 二叉树中度为 1 的结点的个数，求根结点到指定结点的路径。

功能要求:

1) 建立二叉树;

2) 对二叉树进行先序(递归、非递归)、中序(递归、非递归)、后序、层序遍历，并输出对应遍历序列(遍历可基于栈或队列来完成);

3) 求二叉树的高度、宽度;

4) 统计二叉树中度各类结点的个数;

5) 求根结点到指定结点的路径。

界面要求:程序运行后，给出菜单项的内容和输入提示:

1. 建立二叉树

2. 遍历二叉树

3. 二叉树的高度、宽度

4. 各类结点的个数

5. 根结点到指定结点的路径 0. 退出

请选择 0-8:

（1）需求分析

分析总结逻辑结构：

树是一种常见的非线性结构，其具有一对多的关系。一个根结点会连接多个结点。而二叉树是较为特殊的一种，其最多只有两个孩子结点。因此在表示二叉树的过程当中，我们可以使用两个指针来表示二叉树的孩子结点，如果有需要寻找二叉树的双亲结点，只需要再多存储一个双亲结点的指针即可。为了完成题目所给出的对二叉树操作的任务，我们只需要使用孩子表示法即可。

分析总结运算集合：

建立二叉树，遍历二叉树，统计二叉树的高度、宽度，统计二叉树不同入度结点的个数。

（2）概要设计

存储结构设计：

定义一个结构题类型，用于存储每一个结点的信息。结点的值、左孩子指针、右孩子指针等。使用链式存储的方式，动态分配存储空间。因为题目中没有对二叉树结点的删除操作，所以无需对内存进行回收。

算法设计（流程图）：

(本题实现过于复杂，不设计流程图)

（3）详细设计

源代码（注释）

文件 1.stack.h

```
// 顺序栈类型定义
typedef struct
{
    StackElementType elem[maxsize];
    // 存储栈元素的数组
    int top;
    // 栈顶指针，指向栈顶元素的位置
} SeqStack;
// 初始化顺序栈
void InitStack(SeqStack *s)
{
    s->top = -1;
    // 初始化栈顶指针为-1，表示栈为空
}
// 判断顺序栈是否为空
int Stack_IsEmpty(SeqStack *s)
{
    return (s->top == -1);
    // 如果栈顶指针为-1，栈为空
}
// 判断顺序栈是否已满
int Stack_IsFull(SeqStack *s)
{
    return (s->top == maxsize - 1);
    // 如果栈顶指针等于最大容量减 1，栈满
}
// 入栈操作，将元素 e 压入栈中
int Stack_Push(SeqStack *s, StackElementType e)
{
    if (Stack_IsFull(s)) return 0;
    // 如果栈已满，入栈失败
    ++s->top; // 栈顶指针加 1
    s->elem[s->top] = e;
    // 将元素 e 压入栈顶
    return 1; // 入栈成功
}
// 出栈操作，从栈中弹出栈顶元素，并将其赋值给 e
int Stack_Pop(SeqStack *s, StackElementType *e)
{
    if (Stack_IsEmpty(s)) return 0;
    // 如果栈为空，出栈失败
```

```
    *e = s->elem[s->top--];
    // 获取栈顶元素并将栈顶指针减 1
    return 1; // 出栈成功
}
// 读取栈顶元素，但不弹出它
int Stack_GetTop(SeqStack *s, StackElementType *e)
{
    if (Stack_IsEmpty(s)) return 0;
    // 如果栈为空，读取失败
    *e = s->elem[s->top];
    // 将栈顶元素赋值给 e
    return 1; // 读取成功
}
```

文件 2:queue.h

```
// 循环队列类型定义
typedef struct
{
    QueueElementType elem[maxsize];
    // 存储队列元素的数组
    int rear;
    // 队尾指针，指向下一个插入的位置
    int front;
    // 队头指针，指向下一个删除的位置
} SeqQueue;
// 初始化循环队列
void InitQueue(SeqQueue *q)
{
    q->front = 0;
    // 队头指针初始值
    q->rear = 0;
    // 队尾指针初始值
}
// 判断循环队列是否为空
int Queue_IsEmpty(SeqQueue *q)
{
    return (q->front == q->rear);
    // 如果队头指针等于队尾指针，队列为空
}

// 判断循环队列是否已满
int Queue_IsFull(SeqQueue *q)
{
    return ((q->rear + 1) % maxsize == q->front);
    // 如果队尾指针加 1 后的位置等于队头指针，队列满
```

```
}
// 入队操作，将元素 e 插入到循环队列中
int Queue_Push(SeqQueue *q, QueueElementType e)
{
    if (Queue_IsFull(q)) return 0;
    // 如果队列已满，插入失败
    q->elem[q->rear] = e;
    // 将元素 e 插入到队尾
    q->rear = (q->rear + 1) % maxsize;
    // 更新队尾指针，采用模运算实现循环
    return 1; // 插入成功
}
// 出队操作，从循环队列中删除元素，并将其赋值给 e
int Queue_Pop(SeqQueue *q, QueueElementType *e)
{
    if (Queue_IsEmpty(q)) return 0;
    // 如果队列为空，删除失败
    *e = q->elem[q->front];
    // 获取队头元素
    q->front = (q->front + 1) % maxsize;
    // 更新队头指针，采用模运算实现循环
    return 1; // 删除成功
}
// 读取队头元素，将其赋值给 e，但不删除
int Queue_Front(SeqQueue *q, QueueElementType *e)
{
    if (Queue_IsEmpty(q)) return 0;
    // 如果队列为空，读取失败
    *e = q->elem[q->front];
    // 获取队头元素
    return 1; // 读取成功
}
// 读取队尾元素，将其赋值给 e，但不删除
int Queue_Back(SeqQueue *q, QueueElementType *e)
{
    if (Queue_IsEmpty(q)) return 0;
    // 如果队列为空，读取失败
    *e = q->elem[(q->rear - 1 + maxsize) % maxsize];
    // 获取队尾元素，采用模运算处理负数情况
    return 1; // 读取成功
}
```

文件 3:树形结构.c

```
#include<stdio.h>
#include<stdlib.h>
```

```c
#include<string.h>
#define maxsize 100
#define TElemType char
// 将数据元素类型定义为 char
typedef struct BiTree
{
    TElemType data;
    // 节点的数据部分
    struct BiTree *lchild,*rchild;
    // 左右孩子指针
}BiTree;
#define StackElementType BiTree*
// 定义栈元素类型为指向 BiTree 的指针
#define QueueElementType BiTree*
// 定义队列元素类型为指向 BiTree 的指针
#include"queue.h"
// 引入队列的头文件
#include"stack.h"
// 引入栈的头文件
BiTree *tree;
// 定义全局的树指针
SeqStack s;
// 定义全局的栈
int max(int a,int b)// 返回两个数中的最大值
{
    return a>b?a:b;
}
BiTree *build()//使用先序遍历创建一棵树
{
    char str;
    do str = getchar();while(str=='\n');
    // 跳过换行符
    if(str=='#') return NULL;
    // 结束条件，返回 NULL 表示空节点
    BiTree *new_node=(BiTree*)malloc(sizeof(BiTree));
    // 分配新节点空间
    if(new_node==NULL)
    {
        printf("申请空间失败");
        // 如果分配失败，输出错误信息
        return NULL;
    }
    new_node->data=str;
    // 设置节点数据
```

```
    new_node->lchild=build();
    // 递归构建左子树
    new_node->rchild=build();
    // 递归构建右子树
    return new_node;
    // 返回构建好的节点
}
void pre_recursion(BiTree *tree)//先序遍历 tree
{
    if(tree==NULL) return;
    // 如果当前节点为空，直接返回
    printf("%c",tree->data);
    // 输出当前节点数据
    pre_recursion(tree->lchild);
    // 递归遍历左子树
    pre_recursion(tree->rchild);
    // 递归遍历右子树
}
void mid_recursion(BiTree *tree)//中序遍历 tree
{
    if(tree==NULL) return;
    // 如果当前节点为空，直接返回
    mid_recursion(tree->lchild);
    // 递归遍历左子树
    printf("%c",tree->data);
    // 输出当前节点数据
    mid_recursion(tree->rchild);
    // 递归遍历右子树
}
void post_recursion(BiTree *tree)//后序遍历 tree
{
    if(tree==NULL) return;
    // 如果当前节点为空，直接返回
    post_recursion(tree->lchild);
    // 递归遍历左子树
    post_recursion(tree->rchild);
    // 递归遍历右子树
    printf("%c",tree->data);
    // 输出当前节点数据
}
int height(BiTree *tree)//统计二叉树的高度
{
    if(tree==NULL) return 0;
    // 如果树为空，高度为 0
```

```
    return max(height(tree->lchild),height(tree->rchild))+1;
    // 返回左、右子树中高度较大的值+1
}
void cnt(BiTree *tree,int t[])
{
    if(tree->lchild==tree->rchild&&tree->lchild==NULL) ++t[0];
    // 如果节点是叶子节点，叶子节点计数加 1
    else if(tree->rchild!=NULL&&tree->lchild!=NULL) ++t[2];
    // 如果节点有两个子节点，度为 2 的节点计数加 1
    else ++t[1];  // 如果节点有一个子节点，度为 1 的节点计数加 1
    if(tree->lchild!=NULL) cnt(tree->lchild,t);
    // 递归统计左子树
    if(tree->rchild!=NULL) cnt(tree->rchild,t);
    // 递归统计右子树
}
void Inorder(BiTree *root)//中序遍历
{
    SeqStack stack;
    InitStack(&stack);// 初始化栈
    BiTree *p=root;
    while(p!=NULL||!Stack_IsEmpty(&stack))
    {
        if(p!=NULL)
        {
            Stack_Push(&stack,p);
            // 当前节点不为空时，入栈
            p=p->lchild;
            // 移动到左子节点
        }
        else
        {
            Stack_Pop(&stack,&p);
            // 当前节点为空时，出栈
            printf("%c",p->data);
            // 输出节点数据
            p=p->rchild;
            // 移动到右子节点
        }
    }
}
int dfs(BiTree *root, char x)// 深度优先搜索路径
{
    if (root == NULL) return -1;
    // 如果当前节点为空，返回-1 表示未找到
```

```
    if (x == root->data)
    {
        Stack_Push(&s, root);
        // 找到目标节点，入栈
        return 1;  // 返回 1 表示找到
    }
    if (dfs(root->lchild, x)== 1)
    {
        Stack_Push(&s, root);
        // 在左子树中找到目标节点，当前节点入栈
        return 1;  // 返回 1 表示找到
    }
    if (dfs(root->rchild, x)==1)
    {
        Stack_Push(&s, root);
        // 在右子树中找到目标节点，当前节点入栈
        return 1;  // 返回 1 表示找到
    }
    return -1;  // 未找到，返回-1
}
int width(BiTree *root)
{
    if (root == NULL) return 0;
    // 定义一个队列来进行层序遍历
    SeqQueue queue;
    InitQueue(&queue);
    Queue_Push(&queue, root);
    int max_width = 0;
    while (!Queue_IsEmpty(&queue))
    {
        // 当前层的节点数
        int level_size = (queue.rear - queue.front + maxsize) % maxsize;
        if (level_size > max_width) max_width = level_size;  // 更新最大宽度
        // 遍历当前层的所有节点
        for (int i = 0; i < level_size; i++)
        {
            BiTree *node;
            Queue_Pop(&queue, &node);
            // 将子节点加入队列
            if (node->lchild != NULL) Queue_Push(&queue, node->lchild);
            if (node->rchild != NULL) Queue_Push(&queue, node->rchild);
        }
    }
    return max_width;  // 返回最大宽度
```

```
}
void Preorder(BiTree *root)//先序遍历
{
    SeqStack stack;
    InitStack(&stack);// 初始化栈
    BiTree *p=root;
    while(p!=NULL||!Stack_IsEmpty(&stack))
    {
        if(p!=NULL)
        {
            printf("%c",p->data);
            // 输出当前节点数据
            Stack_Push(&stack,p);
            // 当前节点入栈
            p=p->lchild;  // 移动到左子节点
        }
        else
        {
            Stack_Pop(&stack,&p);
            // 当前节点为空时，出栈
            p=p->rchild;  // 移动到右子节点
        }
    }
}
void Levelorder(BiTree *root)//层序遍历
{
    SeqQueue queue;
    InitQueue(&queue);
    // 初始化队列
    BiTree *p=root;
    Queue_Push(&queue,root);
    // 根节点入队
    while(!Queue_IsEmpty(&queue))
    {
        Queue_Pop(&queue,&p);  // 出队
        if(p!=NULL)
        {
            printf("%c",p->data);  // 输出当前节点数据
            Queue_Push(&queue,p->lchild);
            // 左子节点入队
            Queue_Push(&queue,p->rchild);
            // 右子节点入队
        }
    }
```

```
}
//1. 建立二叉树
//2. 遍历二叉树
//3. 二叉树的高度、宽度
//4. 各类结点的个数
//5. 根结点到指定结点的路径
//0. 退出
//1. 建立二叉树
//2. 遍历二叉树
//3. 二叉树的高度、宽度
//4. 各类结点的个数
//5. 根结点到指定结点的路径
//0. 退出
int menu()
{
    printf("1.建立二叉树\n2.遍历二叉树\n3.二叉树的高度、宽度\n4.统计二叉树中各类结点的个数\n5.根结点到指定结点到路径\n6.退出\n");
    int op;
    scanf("%d",&op);  // 获取用户选择的操作
    if(op==1) tree=build();  // 选项 1：创建二叉树
    else if(op==2)
    {
        printf("递归先序遍历：");
        pre_recursion(tree);  // 递归实现先序遍历
        printf("\n 递归中序遍历：");
        mid_recursion(tree);  // 递归实现中序遍历
        printf("\n 递归后序遍历：");
        post_recursion(tree);  // 递归实现后序遍历
        printf("\n 非递归先序遍历：");
        Preorder(tree);  // 非递归实现先序遍历
        printf("\n 非递归中序遍历：");
        Inorder(tree);  // 非递归实现中序遍历
        printf("\n");
    }
    else if(op==3)
    {
        printf("二叉树高度：%d\n",height(tree));
        // 输出二叉树的高度
        printf("二叉树宽度：%d\n",width(tree));
        // 输出二叉树的宽度
    }
    else if(op==4)
    {
        int c[3]={0};  // 初始化节点计数数组
```

```
        cnt(tree,c);  // 统计各类节点的个数
        for(int i=0;i<3;++i)
            printf("度为%d 的结点有%d 个\n",i,c[i]);
            // 输出度为 0、1、2 的节点个数
    }
    else if(op==5)
    {
        printf("需要查找的字符：");
        char x;
        scanf("\n%c",&x);  // 获取用户输入的字符
        InitStack(&s);  // 初始化栈，用于存储路径
        int v=dfs(tree,x);  // 深度优先搜索目标节点的路径
        if(v==-1) printf("未找到\n");
        // 如果未找到，输出提示信息
        else
        {
            while(!Stack_IsEmpty(&s))
            {
                BiTree *t;
                Stack_Pop(&s,&t);  // 依次输出路径上的节点
                printf("%c",t->data);
            }
            printf("\n");
        }
    }
    else if(op==6) return 0;  // 选项 6：退出程序
    return 1;
}

int main()
{
    int v=1;
    while(v) v=menu();  // 循环显示菜单并处理用户输入
    return 0;
}
```

（4）调试分析

样例 1:

首先建立一棵树，这棵树为：AB##CDF#E####

然后输入 2 遍历这棵树，输出结果为：

递归先序遍历：ABCDFE

递归中序遍历：BAFEDC

递归后序遍历：BEFDCA

非递归先序遍历：ABCDFE

非递归中序遍历：BAFEDC

运行结果截图：

```
1.建立二叉树
2.遍历二叉树
3.二叉树的高度、宽度
4.统计二叉树中各类结点的个数
5.根结点到指定结点到路径
6.退出
1
AB##CDF#E####
1.建立二叉树
2.遍历二叉树
3.二叉树的高度、宽度
4.统计二叉树中各类结点的个数
5.根结点到指定结点到路径
6.退出
2
递归先序遍历： ABCDFE
递归中序遍历： BAFEDC
递归后序遍历： BEFDCA
非递归先序遍历： ABCDFE
非递归中序遍历： BAFEDC
```

读取树的高度、宽度，输出为：

二叉树的高度：5

二叉树的宽度：2

运行结果截图：

```
3
二叉树高度： 5
二叉树宽度： 2
```

统计二叉树各类结点的个数，输出为：

度为 0 到结点有 2 个

度为 1 到结点有 3 个

度为 2 到结点有 1 个

运行结果截图：

```
4
度为0到结点有2个
度为1到结点有3个
度为2到结点有1个
```

输出根结点到指定结点(比如输入 E)的路径，输出:

ACDFE

运行结果截图：

```
5
需要查找到字符： E
ACDFE
```

（5）小结

在本次实习中，我更加理解了树这样一种数据结构的作用。学会了树的构造、遍历、寻找路径等算法。

本次实习中遇到了几个困难：

首先是在建立一棵树以后，程序无法自行跳出的问题。一开始我还认为是我代码的问题，后来经过我的排查没有找到代码中存在的不正确的情况，最后发现是我构造了一棵没有完成的树。

然后是在路径输出时遇到的问题，一开始我想再初始化一个字符栈，但是程序中引用的我自己所写的头文件 stack.h 是一个结点类型的栈。后来我认为字符类型可以用结点类型去代替，解决了这个问题。

本次实习是我第一次使用指针去指向指针类型，使用了一个较为复杂的数据结构。

以下是对本次实习，每一个函数的简介：

- 树的构建：代码通过递归先序遍历方式构建二叉树，每个节点以字符形式存储。
- 递归遍历：提供先序、中序、后序三种递归遍历方法，依次访问节点数据。
- 非递归遍历：实现了非递归的先序和中序遍历，利用栈存储临时节点信息。
- 高度计算：递归计算二叉树的高度，即从根节点到最远叶节点的最长路径。
- 节点计数：统计二叉树中度为 0、1、2 的节点数量，并输出结果。
- 路径查找：通过深度优先搜索(DFS)找到指定节点，从根节点输出路径。
- 菜单操作：用户可以通过菜单选择不同的操作，包括建立树、遍历树、计算高度、查找路径等。
- 主程序循环：主函数循环显示菜单，并根据用户输入执行相应操作，直到用户选择退出。

10.哈夫曼编码

任务

根据给定的若干权值来构造哈夫曼树，实现对应的哈夫曼编码以及译码。 功能要求:

1）按传输文本中字符的个数以及字符的频率来建立并输出哈夫曼树的存储结构;

2）设计并输出各字符对应的哈夫曼编码;

3）将传输的文本转换成对应的哈夫曼编码 01 序列;

4）将哈夫曼编码 01 序列翻译成原来的文本字符。

界面要求:程序运行后，给出菜单项的内容和输入提示:

1.建立并输出哈夫曼树

2.设计并输出哈夫曼树

3.将文本转换成 01 编码

4.将 01 编码翻译成文本

0.退出

请选择 0-4:

建立并输出哈夫曼树 设计并输出哈夫曼编码 将文本转换成 01 编码 将 01 编码翻译成文本

（1）需求分析

分析总结逻辑结构：

哈夫曼树是通过维护一棵二叉树，生成二进制序列，给字符编码，使其编码之后的文本长度最小。它以字符出现的频度为权值，字符出现的频率越高，其编码长度越短，字符出现的频率越低，编码越长，这样可以大大降低字符串编码之后的序列长度。

分析总结运算集合：

建立二叉树

根据频率创建二叉树

建立二叉树上的一个结点

根据文本输出二进制序列

根据二进制序列转换回文本

（2）概要设计

存储结构设计：

动态分配内存，使用 malloc 回收和释放

建立二叉树的存储结构

算法设计（流程图）：

（因为本题较为复杂，不设计流程图）

（3）详细设计

源代码（注释）

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>
// 定义二叉树节点结构
typedef struct BiTree
{
    char data; // 节点数据，字符
    int weight; // 节点权重
    struct BiTree *lchild, *rchild; // 左右子树指针
} BiTree;
// 定义哈夫曼编码表的结构
typedef struct
{
    char data; // 字符
    char code[100]; // 对应的哈夫曼编码
} HuffmanCode;
// 创建新的树节点
BiTree* createNode(char data, int weight)
{
    BiTree *node = (BiTree *)malloc(sizeof(BiTree)); // 动态分配内存
    node->data = data; // 设置节点数据
    node->weight = weight; // 设置节点权重
    node->lchild = node->rchild = NULL; // 初始化左右子树为空
    return node; // 返回新节点指针
}
// 合并两个节点为一个新的父节点
BiTree* merge(BiTree *left, BiTree *right)
{
    // 创建一个新的父节点，其权重为左右子节点权重之和
    BiTree *node = createNode('\0', left->weight + right->weight);
    node->lchild = left; // 左子树指向较小的权重节点
    node->rchild = right; // 右子树指向较大的权重节点
    return node; // 返回新的父节点
}
// 比较函数，用于按权重排序
int compare(const void *a, const void *b)
{
    // 比较两个节点的权重大小，qsort 会根据这个结果进行排序
    return (*(BiTree **)a)->weight - (*(BiTree **)b)->weight;
}
// 生成哈夫曼树
BiTree* create(BiTree *nodes[], int n)
{
    // 不断合并最小的两个节点，直到只剩下一个节点
    while (n > 1)
    {
```

```
        qsort(nodes, n, sizeof(BiTree *), compare); // 按权重排序
        BiTree *left = nodes[0]; // 最小权重节点
        BiTree *right = nodes[1]; // 次小权重节点
        BiTree *newNode = merge(left, right); // 合并这两个节点
        nodes[1] = newNode; // 将新节点放置在数组中
        nodes++; // 数组指针前移，忽略掉已合并的节点
        n--; // 数组中有效节点数减少
    }
    return nodes[0]; // 返回哈夫曼树的根节点
}
// 递归生成哈夫曼编码
void build(BiTree *root, char *code, int length, HuffmanCode huffCodes[], int
*idx)
{
    if (root->lchild == NULL && root->rchild == NULL)
    {
        // 当前节点为叶子节点，保存编码
        code[length] = '\0'; // 结束符
        huffCodes[*idx].data = root->data; // 保存字符
        strcpy(huffCodes[*idx].code, code); // 保存编码
        (*idx)++; // 增加编码表索引
        return;
    }
    if (root->lchild)
    {
        // 递归处理左子树
        code[length] = '0'; // 左子树编码为 0
        build(root->lchild, code, length + 1, huffCodes, idx);
    }
    if (root->rchild)
    {
        // 递归处理右子树
        code[length] = '1'; // 右子树编码为 1
        build(root->rchild, code, length + 1, huffCodes, idx);
    }
}
// 根据输入字符串生成哈夫曼编码
void encode(char *str, HuffmanCode huffCodes[], int n)
{
    // 遍历输入字符串中的每个字符
    for (int i = 0; str[i] != '\0'; i++)
    {
        // 查找字符对应的哈夫曼编码
        for (int j = 0; j < n; j++)
```

```
        {
            if (str[i] == huffCodes[j].data)
            {
                printf("%s", huffCodes[j].code); // 输出编码
                break;
            }
        }
    }
    printf("\n");
}
// 根据哈夫曼编码解码为字符串
void decode(char *encodedStr, BiTree *root)
{
    BiTree *current = root; // 从根节点开始
    // 遍历编码字符串的每个字符
    for (int i = 0; encodedStr[i] != '\0'; i++)
    {
        // 根据当前编码字符移动到左或右子树
        if (encodedStr[i] == '0')
            current = current->lchild;
        else if (encodedStr[i] == '1')
            current = current->rchild;

        // 如果到达叶子节点，输出对应的字符
        if (current->lchild == NULL && current->rchild == NULL)
        {
            printf("%c", current->data);
            current = root; // 回到根节点，继续解码
        }
    }
    printf("\n");
}
// 菜单界面函数
void menu() {
    printf("输入结点的个数：");
    int len;
    scanf("%d", &len);
    // 动态分配节点数组
    BiTree *nodes[len];
    printf("输入字符串内容+频度：\n");
    // 输入每个字符及其频度
    for (int i = 0; i < len; ++i)
    {
        char str;
```

```
    int fre;
    scanf(" %c %d", &str, &fre);
    nodes[i] = createNode(str, fre); // 创建叶子节点
}
// 构建哈夫曼树
BiTree *root = create(nodes, len);
// 生成哈夫曼编码
HuffmanCode huffCodes[len];
char code[100];
int idx = 0;
build(root, code, 0, huffCodes, &idx);
// 输出生成的哈夫曼编码
printf("生成的哈夫曼编码:\n");
for (int i = 0; i < len; i++)
{
    printf("%c: %s\n", huffCodes[i].data, huffCodes[i].code);
}
// 菜单选项
int choice;
do {
    printf("\n 请选择操作:\n");
    printf("1. 编码字符串\n");
    printf("2. 解码字符串\n");
    printf("3. 退出\n");
    printf("输入您的选择: ");
    scanf("%d", &choice);

    if (choice == 1) {
        // 编码字符串
        char str[100];
        printf("输入要编码的字符串: ");
        scanf("%s", str);
        printf("编码结果: ");
        encode(str, huffCodes, len);

    } else if (choice == 2) {
        // 解码字符串
        char encodedStr[100];
        printf("输入要解码的哈夫曼编码: ");
        scanf("%s", encodedStr);
        printf("解码结果: ");
        decode(encodedStr, root);

    } else if (choice != 3) {
```

```
            printf("无效的选择，请重新输入。\n");
        }

    } while (choice != 3);

    printf("已退出程序。\n");
}
// 主函数
int main()
{
    menu(); // 启动程序
    return 0;
}
```

（4）调试分析

测试样例 1：

```
输入结点的个数： 5
输入字符串内容+频度：
a 1
b 2
c 3
d 4
 e 5
生成的哈夫曼编码：
a: 000
b: 001
c: 01
d: 10
e: 11

请选择操作：
1. 编码字符串
2. 解码字符串
3. 退出
输入您的选择： 1
输入要编码的字符串： abedced
编码结果： 0000011110011110

请选择操作：
1. 编码字符串
2. 解码字符串
3. 退出
输入您的选择： 2
输入要解码的哈夫曼编码： 0000011110011110
解码结果： abedced
```

测试样例 2:
输入频度：
q 1 w 2 e 3 r 4 t 5 y 6 u 7 i 8 o 9 p 10 a 11 s 12 d 13 f 14 g 15 h 16 j 17 k 18 l 19 z 20 x 21 c 22 v 23 b 24 n 25 m 26
编码后的结果：
h: 0000
j: 0001
k: 0010
o: 00110
r: 001110
t: 001111
l: 0100
z: 0101
x: 0110
p: 01110
a: 01111
c: 1000
v: 1001
s: 10100
q: 10101000
w: 10101001
e: 1010101
y: 101011
b: 1011
n: 1100
m: 1101
d: 11100
f: 11101
g: 11110
u: 111110
i: 111111

```
请选择操作：
1. 编码字符串
2. 解码字符串
3. 退出
输入您的选择: 1
输入要编码的字符串: njfuisthebestschoolinnanjingcity
编码结果: 11000001111011111101111111010000111100001010101101110101011010000111110100
10000000001100011001001111111100110001111110000011111111100111101000111111001111101011
```

```
请选择操作：
1. 编码字符串
2. 解码字符串
3. 退出
输入您的选择: 2
输入要解码的哈夫曼编码: 11000001111011111101111111010000111100001010101101110101011010
10000111111010010000000001100011001001111111100110001111110000011111111100111101000111
1111001111101011
解码结果: njfuisthebestschoolinnanjingcity
```

```
请选择操作：
1. 编码字符串
2. 解码字符串
3. 退出
输入您的选择：1
输入要编码的字符串：thisisacprogram
编码结果：00111100001111111010011111111010001111100001110001110001101111000111001111
101

请选择操作：
1. 编码字符串
2. 解码字符串
3. 退出
输入您的选择：00111100001111111010011111111010001111100001110001110001101111000111001
1111101
无效的选择，请重新输入。

请选择操作：
1. 编码字符串
2. 解码字符串
3. 退出
输入您的选择：2
输入要解码的哈夫曼编码：00111100001111111010011111111010001111100001110001110001101111
10001110011111101
解码结果：thisisacprogram
```

（5）小结

在本次实习过程中，我一开始对哈夫曼编码的了解不是很透彻，导致在编程中出现了一些错误。但是经过我重新学习和巩固，最终我对哈夫曼编码有了更加深入的认识，经过调整我才成功编写出上述代码。赫夫曼编码的具体方法：先按出现的概率大小排队，把两个最小的概率相加，作为新的概率 和剩余的概率重新排队，再把最小的两个概率相加，再重新排队，直到最后变成 1。每次相 加时都将“0”和“1”赋与相加的两个概率，读出时由该符号开始一直走到最后的“1”， 将路线上所遇到的“0”和“1”按最低位到最高位的顺序排好，就是该符号的赫夫曼编码。

11. 并查集

任务

用树形结构来表示集合，树的每个结点代表一个集合元素。实现查找(判定)元素所 属集合的运算、实现集合的并运算。

功能要求:

1) 建立并输出集合的存储结构;

2) 设计并实现查找(判定)元素所属集合的运算;

3) 设计并实现集合的并运算。

界面要求:程序运行后，给出菜单项的内容和输入提示:

1. 建立集合

2. 查找(判定)元素所属集合

3. 集合的并运算

0. 退出 请选择 0-3:

（1）需求分析

分析总结逻辑结构：

并查集是通过维护一棵树，判断两个数字是否属于同一个集合的数据结构。如果要将两个集合连接起来，只要把他的树根连接到另一个集合的树根上即可。如果需要查询两个数字是否在一个集合内，那么只需要根据这两个数字逐层向上寻找双亲，如果其根结点相同，则两个数字在同一个集合内。

分析总结运算集合：

将两个集合合并——将其中一棵树的树根连接到另一棵树的树根上即，合并：将两个不同集合合并成一个集合。通过合并操作，可以将两个不相交的集合合并为一个新的集合。

查询两个数字是否属于同一个集合——分别递归寻找树根，如果树根不一致，那么不在同一个集合，反之则在。在递归寻找树根的过程中进行路径压缩。即，查找：确定某个元素所属的集合。具体来说，就是找到该元素所在树的根节点，根节点可以作为该集合的代表元素。

（2）概要设计

存储结构设计：

维护一棵树，最开始每一个结点代表自己所属于的集合，如果需要将两个集合合并，那么就将这个结点的数据废弃，让其 parent 指针指向另一个结点，方便查询。存储过程中没有必要记录每一个结点的孩子，因为并查集总是向上访问的，不会向下访问。

算法设计（流程图）：

（此题较为复杂，不再设计流程图）

（3）详细设计

源代码（注释）

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
// 定义集合中的节点结构
```

```c
typedef struct Node
{
    int data;
    // 节点数据，即元素值
    struct Node* parent;
    // 指向父节点，用于路径压缩
} Node;
Node* nodes[MAX_SIZE];
// 定义一个全局数组，用于存储集合中的所有节点
// 初始化集合中的单个元素，使其自成一个集合
void InitSet(int x) {
    nodes[x] = (Node*)malloc(sizeof(Node));
    // 动态分配内存给节点
    nodes[x]->data = x;
    // 设置节点的数据为元素值
    nodes[x]->parent = nodes[x];
    // 初始时，每个节点的父节点是它自己
    nodes[x]->rank = 0;
    // 初始时，节点的秩为 0
}
// 查找元素所在集合的代表元（根节点），带路径压缩
Node* findSet(Node* node)
{
    if (node->parent != node)
    {
        // 如果节点的父节点不是它自己，递归查找根节点，并进行路径压缩
        node->parent = findSet(node->parent);
    }
    return node->parent;
    // 返回根节点
}

// 合并两个集合，按秩合并
void unionSets(int x, int y) {
    Node* rootX = findSet(nodes[x]);
    // 查找 x 所在集合的根节点
    Node* rootY = findSet(nodes[y]);
    // 查找 y 所在集合的根节点
    // 如果两个元素的根节点不同，进行合并操作
    if (rootX != rootY)
    {
        rootY->parent = rootX;
        // 将 y 的根节点的父节点设置为 x 的根节点，表示两个集合合并
    }
```

```
}
// 输出集合的存储结构
void printSets(int n)
{
    for (int i = 0; i < n; i++)
    {
        Node* root = findSet(nodes[i]);
        // 查找每个元素所在集合的根节点
        printf("Element %d -> Set Representative %d\n", i, root->data);
        // 输出每个元素及其集合代表元
    }
}
int main()
{
    int n, ch, x, y;
    printf("请输入集合中元素的数量: ");
    scanf("%d", &n);
    // 输入集合中元素的数量
    while (1)
    {
        // 显示操作菜单
        printf("\n1. 建立集合\n");
        printf("2. 查找(判定)元素所属集合\n");
        printf("3. 集合的并运算\n");
        printf("0. 退出\n");
        printf("请选择 0-3: ");
        scanf("%d", &ch);
        if (ch == 1)
        {
            // 建立集合：对每个元素调用 InitSet 函数，初始化集合
            for (int i = 0; i < n; i++) InitSet(i);
        }
        else if (ch == 2)
        {
            // 查找元素所属集合的代表元
            printf("请输入要查找的元素: ");
            scanf("%d", &x);
            printf("元素 %d 所属集合的代表元是: %d\n", x, findSet(nodes[x])->data);
        }
        else if (ch == 3)
        {
            // 集合的并运算：将两个元素所在的集合合并
            printf("请输入要联合的两个元素: ");
            scanf("%d %d", &x, &y);
```

```
            unionSets(x, y);
        }
        else if (ch == 0)
        {
            // 退出程序
            return 0;
        }
    }
    return 0;
}
```

（4）调试分析

1. 建立集合
2. 查找(判定)元素所属集合
3. 集合的并运算
0. 退出

（输入）请选择 0-3：1
（输入）请选择 0-3：2
（输入）请输入要查找的元素：1
（输出）元素 1 所属集合的代表元是：1
（输入）请选择 0-3：2
（输入）请输入要查找的元素：2
（输出）元素 2 所属集合的代表元是：2
（输入）请选择 0-3：3
（输入）请输入要联合的两个元素：1 3
（输入）请选择 0-3：3
（输入）请输入要联合的两个元素：3 5
（输入）请选择 0-3：2
（输入）请输入要查找的元素：1
（输出）元素 1 所属集合的代表元是：1
（输入）请选择 0-3：2
（输入）请输入要查找的元素：3
（输出）元素 1 所属集合的代表元是：1
（输入）请选择 0-3：2
（输入）请输入要查找的元素：5
（输出）元素 1 所属集合的代表元是：1
（输入）请输入要联合的两个元素：2 4
（输入）请选择 0-3：2
（输入）请输入要查找的元素：4
（输出）元素 1 所属集合的代表元是：2

运行截图：

```
1. 建立集合
2. 查找(判定)元素所属集合
3. 集合的并运算
0. 退出
请选择 0-3: 3
请输入要联合的两个元素: 3 5

1. 建立集合
2. 查找(判定)元素所属集合
3. 集合的并运算
0. 退出
请选择 0-3: 2
请输入要查找的元素: 1
元素 1 所属集合的代表元是: 1

1. 建立集合
2. 查找(判定)元素所属集合
3. 集合的并运算
0. 退出
请选择 0-3: 2
请输入要查找的元素: 3
元素 3 所属集合的代表元是: 1

1. 建立集合
2. 查找(判定)元素所属集合
3. 集合的并运算
0. 退出
请选择 0-3: 2
请输入要查找的元素: 5
元素 5 所属集合的代表元是: 1

1. 建立集合
2. 查找(判定)元素所属集合
3. 集合的并运算
0. 退出
请选择 0-3: 3
请输入要联合的两个元素: 2 4

1. 建立集合
2. 查找(判定)元素所属集合
3. 集合的并运算
0. 退出
请选择 0-3: 2
请输入要查找的元素: 4
元素 4 所属集合的代表元是: 2
```

（5）小结

并查集有 2 个较为常见的优化方法，一个是在每一次递归查询的过程中，都将并查集的子树插入到根结点上，实现路径压缩。另外一种优化思路是按秩合并，在合并两个集合时，总是将高度较小的树附加到高度较大的树上，从而保持树的高度尽可能低。在本次实习中，我实现了路径压缩的功能，但是没有写按秩合并，这是本次实习中一个可以优化的点。

并查集的应用非常广泛，其经常用于检测网络连通性，即判断网络中两个节点是否连接在一起；以及最小生成树 Kruskal 算法：用于检测边的连通性并合并连通分量。

以前我从来都是使用一个顺序表来写并查集，这是我第一次使用动态分配的树形结构来写并查集，对我来说也是一种新知识。

12.最小代价生成树

任务

建立图并求图的最小代价生成树。 功能要求:

(1) 建立图的存储结构(邻接矩阵或邻接表)，能够输入图的顶点、边、以及边上的权值的信息，存储到相应存储结构中，并输出图的相关信息。

(2) 求图的最小代价生成树。

（1）需求分析

分析总结逻辑结构：

最小代价生成树是在图上进行的一种操作，有 krustkal 和 prim 两大经典算法。这两种算法都需要构建一张图，然后从途中选择比结点数少 1 的点将所有的结点都连接起来，构成一个最小的生成树。为了输出整个图的相关信息，我们定义了 bfs 和 dfs 函数，分别用于广度优先搜索图和深度优先搜索图，这是描述图的两个维度。

分析总结运算集合：

建立一张图（往往是无向图）

寻找图的最小生成树

深度优先搜索遍历 DFS

广度优先搜索遍历 BFS

（2）概要设计

存储结构设计：

采用邻接表的形式存储边的信息

采用并查集的形式存储 krustkal 算法执行过程中的集合信息

采用映射的形式记录顶点的值

算法设计（流程图）：

（因为本题较为复杂，所以不再绘制流程图）

（3）详细设计

源代码（注释）

文件 1.queue.h

```
// 循环队列类型定义
typedef struct
{
    QueueElementType elem[maxsize];
    // 存储队列元素的数组
    int rear;
    // 队尾指针，指向下一个插入的位置
    int front;
    // 队头指针，指向下一个删除的位置
} SeqQueue;
```

```c
// 初始化循环队列
void InitQueue(SeqQueue *q)
{
    q->front = 0;
    // 队头指针初始值
    q->rear = 0;
    // 队尾指针初始值
}
// 判断循环队列是否为空
int Queue_IsEmpty(SeqQueue *q)
{
    return (q->front == q->rear);
    // 如果队头指针等于队尾指针，队列为空
}
// 判断循环队列是否已满
int Queue_IsFull(SeqQueue *q)
{
    return ((q->rear + 1) % maxsize == q->front);
    // 如果队尾指针加 1 后的位置等于队头指针，队列满
}
// 入队操作，将元素 e 插入到循环队列中
int Queue_Push(SeqQueue *q, QueueElementType e)
{
    if (Queue_IsFull(q)) return 0;
    // 如果队列已满，插入失败
    q->elem[q->rear] = e;
    // 将元素 e 插入到队尾
    q->rear = (q->rear + 1) % maxsize;
    // 更新队尾指针，采用模运算实现循环
    return 1; // 插入成功
}
// 出队操作，从循环队列中删除元素，并将其赋值给 e
int Queue_Pop(SeqQueue *q, QueueElementType *e)
{
    if (Queue_IsEmpty(q)) return 0;
    // 如果队列为空，删除失败
    *e = q->elem[q->front];
    // 获取队头元素
    q->front = (q->front + 1) % maxsize;
    // 更新队头指针，采用模运算实现循环
    return 1; // 删除成功
}
// 读取队头元素，将其赋值给 e，但不删除
int Queue_Front(SeqQueue *q, QueueElementType *e)
```

```
{
    if (Queue_IsEmpty(q)) return 0;
    // 如果队列为空，读取失败
    *e = q->elem[q->front];
    // 获取队头元素
    return 1; // 读取成功
}
// 读取队尾元素，将其赋值给 e，但不删除
int Queue_Back(SeqQueue *q, QueueElementType *e)
{
    if (Queue_IsEmpty(q)) return 0;
    // 如果队列为空，读取失败
    *e = q->elem[(q->rear - 1 + maxsize) % maxsize];
    // 获取队尾元素，采用模运算处理负数情况
    return 1; // 读取成功
}
```

文件 2.Krustkal.c

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
// 定义队列相关常量与函数（需要 queue.h 文件）
#define QueueElementType int
#define maxsize 100
#include"queue.h"
#define MAX_VERTEX_NUM 20
#define INFNITY 32768
// 定义顶点数据类型
#define VertexData char
// 图的种类：DG-有向图，DN-有向网，UDG-无向图，UDN-无向网
typedef enum { DG, DN, UDG, UDN } GraphKind;
// 边结点结构定义
typedef struct ArcNode
{
    int adjvex;             // 该边指向的顶点下标
    int weight;             // 边的权重
    struct ArcNode *nextarc; // 指向下一个边结点
} ArcNode;
// 顶点结点结构定义
typedef struct VertexNode
{
    VertexData data;        // 顶点数据
    ArcNode *firstarc;      // 指向第一条依附该顶点的边
} VertexNode;
// 图的邻接表结构定义
```

```
typedef struct
{
    VertexNode vertex[MAX_VERTEX_NUM]; // 顶点数组
    int vexnum, arcnum;              // 顶点数，边数
    GraphKind kind;                 // 图的种类
} AdjList;
// Kruskal 算法中使用的边结构
typedef struct
{
    int u, v;            // 边的两个顶点
    int weight;          // 边的权重
} Edge;
int visited[MAX_VERTEX_NUM]; // 访问标志数组
// 根据顶点数据查找其在图中的下标
int get_num(AdjList *graph, VertexData target)
{
    for (int i = 1; i <= graph->vexnum; ++i)
    {
        if (graph->vertex[i].data == target)
            return i;
    }
    return -1;
}
// 构建图的邻接表结构
void build(AdjList *graph)
{
    printf("请输入图的顶点数:");
    scanf("%d", &(graph->vexnum));

    printf("请输入图各个顶点的名称:(不要带有空格)");
    for (int i = 1; i <= graph->vexnum; ++i) {
        char input;
        do scanf("%c", &input); while (input == '\n'); // 过滤掉换行符
        graph->vertex[i].data = input;
        graph->vertex[i].firstarc = NULL; // 初始化每个顶点的边链表为空
    }
    printf("输入边的数量:");
    scanf("%d", &(graph->arcnum));
    printf("输入各个顶点之间的关系(边 a) (边 b) (边权):\n");
    for (int i = 1; i <= graph->arcnum; ++i) {
        char a, b;
        int c;
        scanf("\n%c %c %d", &a, &b, &c); // 输入两个顶点及其边的权重
        int a_num = get_num(graph, a);
```

```
        int b_num = get_num(graph, b);
        if (a_num == -1 || b_num == -1) {
            printf("输入数据错误,建图失败\n");
            return;
        }
        // 创建从 a 到 b 的边
        ArcNode *new_node = (ArcNode*)malloc(sizeof(ArcNode));
        if (new_node == NULL)//处理内存分配失败的异常
        {
            printf("内存分配失败\n");
            return;
        }
        //建立第一条边
        new_node->nextarc = graph->vertex[a_num].firstarc;
        graph->vertex[a_num].firstarc = new_node;
        new_node->adjvex = b_num;
        new_node->weight = c;
        // 创建从 b 到 a 的边（因为是无向图）
        new_node = (ArcNode*)malloc(sizeof(ArcNode));
        if (new_node == NULL)
        {
            printf("内存分配失败\n");
            return;
        }
        //建立第二个边
        new_node->nextarc = graph->vertex[b_num].firstarc;
        graph->vertex[b_num].firstarc = new_node;
        new_node->adjvex = a_num;
        new_node->weight = c;
    }
}
// 深度优先搜索（DFS）遍历
void dfs(AdjList *graph, int v)
{
    printf("%c ", graph->vertex[v].data);
    // 输出当前顶点
    visited[v] = 1;
    // 标记当前顶点为已访问
    ArcNode *arc = graph->vertex[v].firstarc;
    while (arc != NULL)
    {
        int adjVex = arc->adjvex;
        if (!visited[adjVex]) dfs(graph, adjVex);
        // 递归访问相邻顶点
```

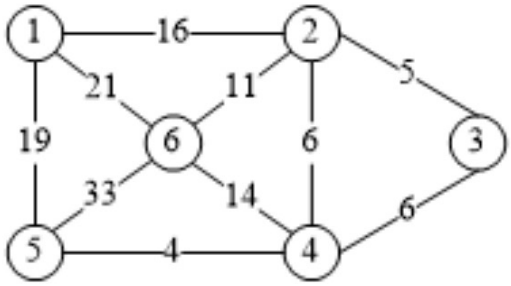
```
        arc = arc->nextarc;
        //还原标记
    }
}
// 广度优先搜索（BFS）遍历
void bfs(AdjList *graph, int start)
{
    memset(visited, 0, sizeof visited);
    // 重置访问标志数组
    printf("广度优先搜索遍历:\n");
    SeqQueue queue;
    //定义一个队列实现广搜
    InitQueue(&queue); // 初始化队列
    Queue_Push(&queue, start);
    visited[start] = 1;
    //第一个数据点开始
    while (!Queue_IsEmpty(&queue))
    {
        int v;
        Queue_Pop(&queue, &v); // 弹出队列头部元素
        printf("%c ", graph->vertex[v].data);
        for (ArcNode *p = graph->vertex[v].firstarc; p != NULL; p = p->nextarc)
        {
            if (!visited[p->adjvex])
            {
                Queue_Push(&queue, p->adjvex);
                //将这一层的数据推入队列
                visited[p->adjvex] = 1;
            }
        }
    }
    printf("\n");
}
// 并查集查找操作：查找元素的根节点
int find(int parent[], int i)
{
    while (parent[i] != i) i = parent[i];
    //实现路径压缩过程
    return i;
}
// 并查集合并操作：合并两个集合
void union_sets(int parent[], int rank[], int x, int y)
{
    int xroot = find(parent, x);
```

```
    int yroot = find(parent, y);
    parent[xroot] = yroot;
    //将两个集合合并
}
// Kruskal 算法实现：计算最小生成树（MST）的边权之和
int kruskal(AdjList *graph)
{
    int parent[MAX_VERTEX_NUM];
    int rank[MAX_VERTEX_NUM];
    int edge_count = 0, min_cost = 0;
    Edge edges[MAX_VERTEX_NUM * MAX_VERTEX_NUM];
    //定义两个数组，用来存储边的信息，避免访问全局变量
    int edge_index = 0;
    // 初始化并查集
    for (int i = 1; i <= graph->vexnum; ++i)
    {
        parent[i] = i;
        rank[i] = 0;
        // 将图中的所有边存储到 edges 数组中
        for (ArcNode *arc = graph->vertex[i].firstarc; arc != NULL; arc =
arc->nextarc)
        {
            edges[edge_index].u = i;
            edges[edge_index].v = arc->adjvex;
            edges[edge_index].weight = arc->weight;
            edge_index++;
        }
    }
    // 按边权升序排序
    for (int i = 0; i < edge_index - 1; i++)
    {
        for (int j = 0; j < edge_index - i - 1; j++)
        {
            if (edges[j].weight > edges[j + 1].weight)
            {
                Edge temp = edges[j];
                edges[j] = edges[j + 1];
                edges[j + 1] = temp;
            }
        }
    }
    // Kruskal 算法选择边构建最小生成树
    for (int i = 0; i < edge_index && edge_count < graph->vexnum - 1; i++)
    {
```

```
        int u = find(parent, edges[i].u);
        int v = find(parent, edges[i].v);
        if (u != v)
        {
            union_sets(parent, rank, u, v);
            min_cost += edges[i].weight;
            edge_count++;
            printf("选择的边: %c-%c, 权重: %d\n", graph->vertex[edges[i].u].data,
graph->vertex[edges[i].v].data, edges[i].weight);
        }
    }
    //返回答案
    return min_cost;
}
int main()
{
    AdjList g;
    //初始化一张图
    build(&g);
    //输入图的数据
    memset(visited, 0, sizeof(visited));
    //清空已经被访问的数组
    printf("深度优先搜索遍历：\n");
    for (int i = 1; i <= g.vexnum; ++i)
        if (!visited[i]) dfs(&g, i);
    //这么做是为了防止出现第 1 个点不在图中，导致图无法被遍历
    //但事实上不会存在这种可能性，因为后面需要建立最小生成树
    //如果这个点不在途中，那么最小生成树一定不存子
    printf("\n");
    //深搜的结果已经输出，输出一个换行，接下来开始广搜
    bfs(&g, 1);
    printf("最小生成树序列：\n");
    int mst_cost = kruskal(&g);
    printf("最小生成树的边权之和：%d\n", mst_cost);
    //输出结果
    return 0;
}
```

（4）调试分析:

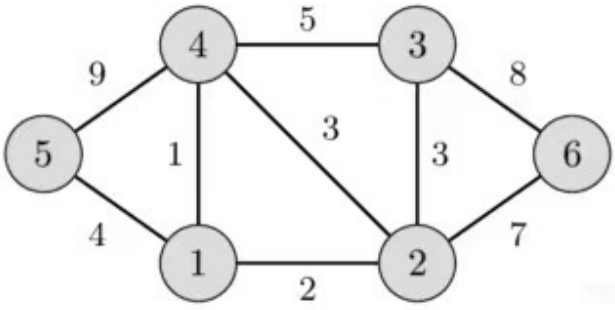
样例 1:



样例 1 的输入和输出截图:

```
请输入图的顶点数:6
请输入图各个顶点的名称:(不要带有空格)123456
输入边的数量:10
输入各个顶点之间的关系(边a)(边b)(边权):
1 2 16
1 5 19
5 6 33
1 6 21
2 6 11
6 4 14
2 4 6
2 3 5
3 4 6
4 5 4
深度优先搜索遍历:
1 6 4 5 3 2
广度优先搜索遍历:
1 6 5 2 4 3
u,v:4 5 4
u,v:2 3 5
u,v:2 4 6
u,v:2 6 11
u,v:1 2 16
最小生成树的边权之和: 42
```

样例 2:



样例 2 的输入和输出截图：

```
请输入图的顶点数:6
请输入图各个顶点的名称:(不要带有空格)123456
输入边的数量:9
输入各个顶点之间的关系(边a)(边b)(边权):
5 4 9
5 1 4
4 3 5
4 1 1
1 2 2
4 2 3
3 2 3
3 6 8
2 6 7
深度优先搜索遍历:
1 2 6 3 4 5
广度优先搜索遍历:
1 2 4 5 6 3
最小生成树序列:
u,v:1 4 1
u,v:1 2 2
u,v:2 3 3
u,v:1 5 4
u,v:2 6 7
最小生成树的边权之和: 17
```

（5）小结

Kruskal 算法从另一途径求网的最小生成树。其基本思想是：假设连通网 G=（V，E），令最小生成树的初始状态为只有 n 个顶点而无边的非连通图，概述图中每个顶点自成一个连通分量。在 E 中选择代价最小的边，若该边依附的顶点分别在 T 中不同的连通分量上，则将此边加入到 T 中；否则，舍去此边而选择下一条代价最小的边。依此类推，直至 T 中所有顶点构成一个连通分量为止。这段代码比较适合求解系数图的 MST，对于稠密图我们更多地会采用 Prim 算法，因为篇幅限制和精力有限没有在此给出。

13.拓补排序及关键路径

任务

建立图并实现图的拓扑排序和关键路径。 功能要求:

(1) 建立图的存储结构(邻接矩阵或邻接表)，能够输入图的顶点和边以及边上的权值等信息，存储到相应存储结构中，并输出图的结构。

(2) 对图进行拓扑排序并输出结果。

(3) 在(2)的基础上求出图的关键路径并输出关键路径以及源点到汇点的最长路径长度。

（1）需求分析

分析总结逻辑结构：

拓扑排序是图中一种经典的算法，常用于工程问题。由于有些工程必须要在另外几个工程结束以后才可以开工，所以才有了这样一个问题。拓扑排序的存储是以是以图为主的，每一次删除掉一个入度为 0 的结点，更新所有以这个结点为开始的结点的 earliest 值，直到图中不再存在结点。如果出现环，那么说明这张图的拓扑排序序列不存在。否则一定会存在关键路径，以及从源点到汇点的最长路径长度。最长路径长度确定之后，我们可以从汇点倒推，推理出每一个点的最晚开工时间，以及关键路径。

分析总结运算集合：

创建一个图

在图中新增一个结点

在途中新增一条边

删除入度为 0 的点

更新所有与该点连接点点 earliest 数组

从汇点倒推最晚开工时间

输出关键路径

（2）概要设计

存储结构设计：

使用动态分配内存，使用邻接表存储图

结构体定义：

ArcNode: 表示图中的一条边，包含三个成员：adjvex: 邻接顶点的索引。weight: 边的权重。nextarc: 指向下一条边的指针，用于在邻接表中链式存储边。

VertexNode: 表示图中的一个顶点,包含两个成员:data: 顶点的数据(字符类型)。firstarc: 指向与该顶点相邻的第一条边的指针。

AdjList: 图的邻接表结构，包含两个成员：

vertex: 顶点数组,每个元素是一个 VertexNode 结构体。vexnum: 图中顶点的数量。arcnum: 图中边的数量。

图的初始化：initGraph(AdjList *graph): 初始化图，将顶点数和边数都设置为 0。

添加边：addEdge(AdjList *graph, int from, int to, int weight): 向图中添加一条边。创建一个新的 ArcNode 结构体，将其插入到指定顶点的边链表中。

图的创建：buildGraph(AdjList *graph)：从用户输入中读取顶点和边的信息，构建图的邻接表。
拓扑排序：topologicalSort(AdjList *graph, int *topoOrder)：使用 Kahn 算法进行拓扑排序。计算每个顶点的入度，将入度为 0 的顶点加入队列，逐步处理队列中的顶点，更新拓扑排序结果。
关键路径计算：findCriticalPath(AdjList *graph)：计算关键路径。通过拓扑排序，计算每个顶点的最早开始时间和最晚开始时间，然后通过 printCriticalPath 函数输出关键路径。
输出关键路径：printCriticalPath(AdjList *graph, int *topoOrder, int *earliest, int *latest, int ans)：遍历图的每条边，如果边的最早开始时间和最晚开始时间相同，则输出这条边作为关键路径的一部分。

算法设计（流程图）：
（因为此题较为复杂，这里不设计流程图）

（3）详细设计
源代码（注释）

```
#include <stdio.h>
#include <stdlib.h>
#define VertexData char
// 定义顶点数据类型为字符
#define MAX_VERTEX_NUM 20
// 最大顶点数量
#define INF 32767
// 表示无穷大，用于初始化最晚开始时间
// 边节点结构体
typedef struct ArcNode
{
    int adjvex;
    // 邻接顶点的索引
    int weight;
    // 边的权重
    struct ArcNode *nextarc;
    // 指向下一条边的指针
} ArcNode;
// 顶点节点结构体
typedef struct VertexNode
{
    VertexData data;
    // 顶点的数据（字符）
    ArcNode *firstarc;
    // 指向第一条边的指针
} VertexNode;
// 邻接表结构体
```

```
typedef struct
{
    VertexNode vertex[MAX_VERTEX_NUM];
    // 顶点数组
    int vexnum, arcnum;
    // 顶点数和边数
} AdjList;
// 获取指定顶点在图中的索引
int get_num(AdjList *graph, VertexData target)
{
    for (int i = 1; i <= graph->vexnum; ++i)
    {
        if (graph->vertex[i].data == target)
            return i;
            // 找到目标顶点，返回索引
    }
    return -1;
    // 没找到目标顶点，返回-1
}

// 初始化图
void initGraph(AdjList *graph)
{
    graph->vexnum = 0;
    // 初始化顶点数为 0
    graph->arcnum = 0;
    // 初始化边数为 0
}

// 添加边到图中
void addEdge(AdjList *graph, int from, int to, int weight)
{
    ArcNode *newNode = (ArcNode*)malloc(sizeof(ArcNode));
    // 为新边分配内存
    newNode->adjvex = to;
    // 设置邻接顶点
    newNode->weight = weight;
    // 设置边的权重
    newNode->nextarc = graph->vertex[from].firstarc;
    // 将新边插入到当前顶点的边表中
    graph->vertex[from].firstarc = newNode;
    // 更新顶点的第一条边
}
// 创建图
```

```c
void buildGraph(AdjList *graph)
{
    printf("请输入顶点数: ");
    scanf("%d", &(graph->vexnum));
    // 输入顶点数量
    printf("请输入各个顶点的名称(不要带空格): ");
    for (int i = 0; i < graph->vexnum; i++)
    {
        scanf(" %c", &(graph->vertex[i].data));
        // 输入每个顶点的名称
        graph->vertex[i].firstarc = NULL;
        // 初始化每个顶点的边表为空
    }
    printf("请输入边的数量: ");
    scanf("%d", &(graph->arcnum));
    // 输入边的数量
    printf("请输入每条边的信息(起点 终点 权重):\n");
    for (int i = 0; i < graph->arcnum; i++)
    {
        char u, v;
        int w;
        scanf(" %c %c %d", &u, &v, &w);
        // 输入每条边的起点、终点和权重
        int from = -1, to = -1;
        for (int j = 0; j < graph->vexnum; j++)
        {
            if (graph->vertex[j].data == u) from = j;
            // 获取起点的索引
            if (graph->vertex[j].data == v) to = j;
            // 获取终点的索引
        }
        if (from != -1 && to != -1)
        {
            addEdge(graph, from, to, w);
            // 如果起点和终点都存在，添加这条边
        }
    }
}
// 拓扑排序（Kahn 算法）
void topologicalSort(AdjList *graph, int *topoOrder)
{
    int inDegree[MAX_VERTEX_NUM] = {0};
    // 初始化所有顶点的入度为 0
    int queue[MAX_VERTEX_NUM], front = 0, rear = 0;
```

```
// 队列用于存储入度为 0 的顶点
int count = 0;
// 计算每个顶点的入度
for (int i = 0; i < graph->vexnum; i++)
{
    ArcNode *p = graph->vertex[i].firstarc;
    while (p != NULL)
    {
        inDegree[p->adjvex]++;
        // 邻接顶点的入度加 1
        p = p->nextarc;
        // 遍历所有边
    }
}
// 将入度为 0 的顶点入队
for (int i = 0; i < graph->vexnum; i++)
{
    if (inDegree[i] == 0)
    {
        queue[rear++] = i;
        // 入度为 0，入队
    }
}
// 处理队列
while (front < rear)
{
    int v = queue[front++];
    // 出队一个顶点
    topoOrder[count++] = v;
    // 将其加入拓扑排序结果
    ArcNode *p = graph->vertex[v].firstarc;
    while (p != NULL)
    {
        inDegree[p->adjvex]--;
        // 邻接顶点的入度减 1
        if (inDegree[p->adjvex] == 0)
        {
            queue[rear++] = p->adjvex;
            // 如果入度为 0，入队
        }
        p = p->nextarc;
        // 继续处理下一个邻接顶点
    }
}
```

```
    // 检查是否存在环
    if (count < graph->vexnum)
    {
        printf("图中存在环，无法进行拓扑排序。\n");
    }
    else
    {
        printf("拓扑排序结果: ");
        for (int i = 0; i < count; i++)
        {
            printf("%c ", graph->vertex[topoOrder[i]].data);
            // 输出拓扑排序结果
        }
        printf("\n");
    }
}
// 查找关键路径
void findCriticalPath(AdjList *graph)
{
    int topoOrder[MAX_VERTEX_NUM];
    int earliest[MAX_VERTEX_NUM] = {0};
    // 初始化最早开始时间为 0
    int latest[MAX_VERTEX_NUM];
    // 最晚开始时间数组
    for (int i = 0; i < graph->vexnum; i++)
    {
        latest[i] = INF;
        // 初始化最晚开始时间为无穷大
    }
    topologicalSort(graph, topoOrder);
    // 对图进行拓扑排序
    // 计算最早开始时间
    for (int i = 0; i < graph->vexnum; i++)
    {
        int u = topoOrder[i];
        ArcNode *p = graph->vertex[u].firstarc;
        while (p != NULL)
        {
            int v = p->adjvex;
            if (earliest[u] + p->weight > earliest[v])
            {
                earliest[v] = earliest[u] + p->weight;
                // 更新最早开始时间
            }
```

```
            p = p->nextarc;
        }
    }
    // 计算最晚开始时间
    for (int i = graph->vexnum - 1; i >= 0; i--)
    {
        int u = topoOrder[i];
        ArcNode *p = graph->vertex[u].firstarc;
        while (p != NULL)
        {
            int v = p->adjvex;
            if (latest[v] - p->weight < latest[u])
            {
                latest[u] = latest[v] - p->weight;
                // 更新最晚开始时间
            }
            p = p->nextarc;
        }
    }
    // 输出最长路径长度
    printf("最长路径长度: %d\n", earliest[topoOrder[graph->vexnum - 1]]);
    printCriticalPath(graph, topoOrder, earliest, latest,
earliest[topoOrder[graph->vexnum - 1]]);
}

// 输出关键路径
void printCriticalPath(AdjList *graph, int *topoOrder, int *earliest, int
*latest, int ans)
{
    printf("关键路径:\n");
    for (int i = 0; i < graph->vexnum; i++)
    {
        int u = topoOrder[i];
        ArcNode *p = graph->vertex[u].firstarc;
        while (p != NULL)
        {
            int v = p->adjvex;
            int e = earliest[u];
            // 当前顶点的最早开始时间
            int l = latest[v] - p->weight - INF + ans;
            // 邻接顶点的最晚开始时间减去边的权重
            if (e == l)
                printf("%c->%c \n", graph->vertex[u].data,
graph->vertex[v].data);
```

```
                // 如果最早和最晚开始时间相等，则是关键路径
                p = p->nextarc;
            }
        }
        printf("\n");
}
int main()
{
    AdjList graph;
    initGraph(&graph);
    buildGraph(&graph);
    findCriticalPath(&graph);
    return 0;
}
```

（4）调试分析

样例 1:

输出结果：

拓扑排序结果: 1 3 2 5 4 6

最长路径长度: 43

关键路径:

1->3

3->2

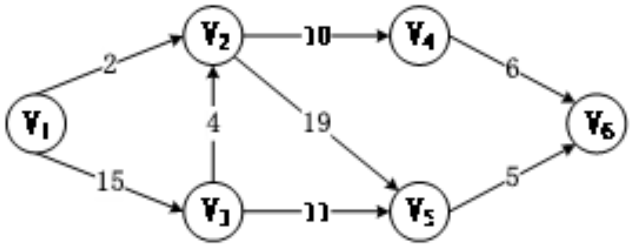
2->5

5->6

输出结果截图：

```
请输入顶点数: 6
请输入各个顶点的名称(不要带空格): 123456
请输入边的数量: 8
请输入每条边的信息(起点 终点 权重):
1 2 2
1 3 15
3 2 4
2 4 10
2 5 19
3 5 11
5 6 5
4 6 6
拓扑排序结果: 1 3 2 5 4 6
最长路径长度: 43
关键路径:
1->3
3->2
2->5
5->6
```

样例 1 的图：



样例 2(错误样例):

显然，若在样例 1 的基础上新增一条边，其边为：2 1 2，则构成了一个环，不存在拓扑序列

输出结果：图中存在环，无法进行拓扑排序。

输出结果截图：

```
请输入顶点数：6
请输入各个顶点的名称(不要带空格)：123456
请输入边的数量：9
请输入每条边的信息(起点 终点 权重)：
1 2 2
1 3 15
3 2 4
2 4 10
2 5 19
3 5 11
5 6 5
4 6 6
2 1 2
图中存在环，无法进行拓扑排序。
```

程序监测到环以后，直接跳出，无法输出拓扑序列

样例 3:

输出结果：

拓扑排序结果: 1 3 2 4 5 6

最长路径长度: 19

关键路径:

1->3

3->2

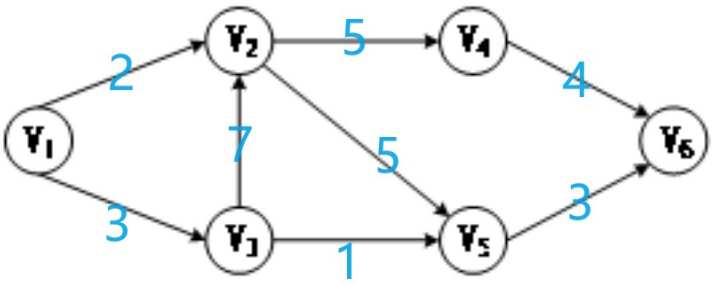
2->4

4->6

输出结果截图：

```
请输入顶点数： 6
请输入各个顶点的名称(不要带空格)： 123456
请输入边的数量： 8
请输入每条边的信息(起点 终点 权重)：
1 2 2
1 3 3
3 2 7
2 5 5
3 5 1
2 4 5
4 6 4
5 6 3
拓扑排序结果： 1 3 2 4 5 6
最长路径长度： 19
关键路径：
1->3
3->2
2->4
4->6
```

样例 3 的图：



（5）小结

结构体定义：ArcNode 表示边，VertexNode 表示顶点，AdjList 表示图的邻接表结构。

图的初始化：initGraph 函数初始化图，将顶点数和边数设置为 0。

添加边：addEdge 函数将一条边添加到图中，链式插入到目标顶点的边链表中。

图的创建：buildGraph 函数从用户输入中读取顶点和边的信息，构建图的邻接表。

拓扑排序：topologicalSort 函数使用 Kahn 算法进行拓扑排序，计算顶点的入度并生成拓扑顺序。

关键路径计算：findCriticalPath 函数通过拓扑排序计算最早开始时间和最晚开始时间，并调用 printCriticalPath 输出关键路径。

输出关键路径：printCriticalPath 函数遍历边，检查边的最早开始时间和最晚开始时间是否相同，输出关键路径。

14.交通咨询系统

任务

设计一个简易交通咨询系统，能让旅客咨询从一个城市到另一个城市之间的最短路径。
功能要求:
(1) 建立交通网络图的存储结构，并输出;
(2) 求单源最短路径(Dijkstra 算法)，并输出;
(3) 求任一对城市之间的最短路径(Floyd 算法)，并输出。

（1）需求分析
分析总结逻辑结构：
首先需要创建一个图，然后对这张图进行深度优先搜索(DFS)和广度优先搜索(BFS)，然后输入开始的点和结束的点，先进行一轮 Dijkstra 算法，寻找从起点到终点的最短路，然后基于动态规划的思想，在进行一遍 Floyd 的扫描，并输出 Floyd 算法矩阵。
分析总结运算集合：
建立一张图
创建图的顶点和边
利用 DFS 和 BFS 分别进行深度优先搜索遍历和广度优先搜索遍历
利用 Dijkstra 计算两个给定点之间的最短路
利用 Floyd 算法计算每两个点之间的最短路

（2）概要设计
存储结构设计：
采用邻接表的形式存储边的信息
采用并查集的形式存储 kruskal 算法执行过程中的集合信息
采用映射的形式记录顶点的值

算法设计（流程图）：
（因本题较为复杂，不再制作流程图）

（3）详细设计
源代码（注释）
文件 1.Queue.h

```
//构建循环队列类型
// 1.实现循环队列的初始化
// 2.循环队列的判空
// 3.循环队列的入队
// 4.循环队列的出队
// 5.读取队头元素
// 6.读取队尾元素内容
// 7.基于循环队列实现杨辉三角形 N 行数据输出
```

```c
// @右手 fang   2024-08-19
typedef struct
{
    QueueElementType elem[maxsize];
    int rear;
    int front;
}SeqQueue;
void InitQueue(SeqQueue *q)//循环链表的初始化操作
{
    q->front=0;
    q->rear=0;
}
int Queue_IsEmpty(SeqQueue *q)//循环队列的判空
{
    if(q->front==q->rear) return 1;
    else return 0;
}
int Queue_IsFull(SeqQueue *q)
{
    return (q->rear+1)%maxsize==q->rear;
}
int Queue_Push(SeqQueue *q,QueueElementType e)
{
    if(Queue_IsFull(q)) return 0;
    q->elem[q->rear]=e;
    q->rear=(q->rear+1)%maxsize;
    return 1;
}
int Queue_Pop(SeqQueue *q,QueueElementType *e)
{
    if(Queue_IsEmpty(q)) return 0;
    *e=q->elem[q->front];
    q->front=(q->front+1)%maxsize;
    return 1;
}
int Queue_Front(SeqQueue *q,QueueElementType *e)
{
    if(Queue_IsEmpty(q)) return 0;
    *e=q->elem[q->front];
    return 1;
}
int Queue_Back(SeqQueue *q,QueueElementType *e)
{
    if(Queue_IsEmpty(q)) return 0;
```

```
    *e=q->elem[(q->rear-1+maxsize)%maxsize];
    return 1;
}
```

文件 2. 交通咨询系统. c

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define QueueElementType int
// 定义队列元素的类型为整型
#define maxsize 100
// 定义队列的最大尺寸
#include "Queue.h"
// 引入自定义的循环队列头文件
#define MAX_VERTEX_NUM 20
// 图的最大顶点数量
#define INFNITY 32768
// 定义一个表示无穷大的值
#define VertexData char
// 顶点的数据类型为字符类型
typedef enum { DG, DN, UDG, UDN } GraphKind;
// 图的类型：DG 有向图，DN 有向网，UDG 无向图，UDN 无向网
int visited[maxsize];
// 访问标记数组
// 图的邻接表中边的结点定义
typedef struct ArcNode
{
    int adjvex; // 该边所指向的顶点的位置
    int weight; // 边的权重
    struct ArcNode *nextarc; // 指向下一条边的指针
} ArcNode;
// 图的邻接表中顶点的结点定义
typedef struct VertexNode
{
    VertexData data; // 顶点的数据
    ArcNode *firstarc; // 指向该顶点第一条边的指针
} VertexNode;
// 图的邻接表定义
typedef struct
{
    VertexNode vertex[MAX_VERTEX_NUM]; // 顶点数组
    int vexnum, arcnum; // 图的顶点数和边数
    GraphKind kind; // 图的类型
} AdjList;
```

```
// 边的信息结构体，用于存储边的两个顶点和权重
typedef struct
{
    int u, v;
    int weight;
} Edge;
// 根据顶点数据查找其在图中的索引
int get_num(AdjList *graph, VertexData target)
{
    for (int i = 1; i <= graph->vexnum; ++i)
    {
        if (graph->vertex[i].data == target)
            return i;
    }
    return -1; // 如果找不到则返回-1
}
// 函数原型声明
// 构建图的邻接表
void build(AdjList *graph)
{
    printf("请输入图的顶点数:");
    scanf("%d", &(graph->vexnum));
    printf("请输入图各个顶点的名称:(不要带有空格)");
    for (int i = 1; i <= graph->vexnum; ++i)
    {
        char input;
        do scanf("%c", &input); while (input == '\n');
        // 读取顶点名称
        graph->vertex[i].data = input;
        graph->vertex[i].firstarc = NULL;
        // 初始化边表为空
    }
    printf("输入边的数量:");
    scanf("%d", &(graph->arcnum));
    printf("输入各个顶点之间的关系(边 a) (边 b) (边权):\n");
    for (int i = 1; i <= graph->arcnum; ++i) {
        char a, b;
        int c;
        scanf("\n%c %c %d", &a, &b, &c);
        // 读取边的起点、终点和权重
        int a_num = get_num(graph, a);
        int b_num = get_num(graph, b);
        if (a_num == -1 || b_num == -1)
        {
```

```
            printf("输入数据错误,建图失败\n");
            return;
        }
        // 为起点 a 添加边
        ArcNode *new_node = (ArcNode*)malloc(sizeof(ArcNode));
        if (new_node == NULL)
        {
            printf("内存分配失败\n");
            return;
        }
        new_node->nextarc = graph->vertex[a_num].firstarc;
        graph->vertex[a_num].firstarc = new_node;
        new_node->adjvex = b_num;
        new_node->weight = c;
        // 为终点 b 添加边（如果是无向图）
        new_node = (ArcNode*)malloc(sizeof(ArcNode));
        if (new_node == NULL)
        {
            printf("内存分配失败\n");
            return;
        }
        new_node->nextarc = graph->vertex[b_num].firstarc;
        graph->vertex[b_num].firstarc = new_node;
        new_node->adjvex = a_num;
        new_node->weight = c;
    }
}
// 辅助函数：从指定顶点 v 开始进行深度优先搜索
void dfs(AdjList *graph, int v)
{
    // 打印当前顶点并标记为已访问
    printf("%c ", graph->vertex[v].data);
    visited[v] = 1;
    // 遍历当前顶点的邻接表
    ArcNode *arc = graph->vertex[v].firstarc;
    while (arc != NULL)
    {
        int adjVex = arc->adjvex;
        if (!visited[adjVex]) dfs(graph, adjVex);  // 递归访问相邻顶点
        arc = arc->nextarc;  // 继续下一个邻接顶点
    }
}
// 广度优先搜索
void bfs(AdjList *graph, int start)
```

```
{
    memset(visited, 0, sizeof visited); // 初始化访问标记
    printf("广度优先搜索遍历:\n");
    SeqQueue queue; // 声明队列
    InitQueue(&queue); // 初始化队列
    Queue_Push(&queue, start); // 起点入队
    visited[start] = 1;
    while (!Queue_IsEmpty(&queue))
    {
        int v;
        Queue_Pop(&queue, &v); // 出队并访问
        printf("%c ", graph->vertex[v].data);
        // 遍历邻接表
        for (ArcNode *p = graph->vertex[v].firstarc; p != NULL; p = p->nextarc)
        {
            if (!visited[p->adjvex])
            {
                Queue_Push(&queue, p->adjvex); // 未访问的邻接点入队
                visited[p->adjvex] = 1; // 标记为已访问
            }
        }
    }
    printf("\n");
}
// Dijkstra 算法，计算源点 source 到目标点 target 的最短路径
void dijkstra(AdjList *graph, char t1, char t2)
{
    int source=get_num(graph,t1);
    int target=get_num(graph,t2);
    int dist[MAX_VERTEX_NUM];
    // 存储源点到各个顶点的最短路径
    int path[MAX_VERTEX_NUM];
    // 存储最短路径的前驱节点
    int visited[MAX_VERTEX_NUM];
    // 标记顶点是否已访问
    // 初始化
    for (int i = 1; i <= graph->vexnum; ++i)
    {
        dist[i] = INFNITY;
        // 设置初始距离为无穷大
        path[i] = -1;
        // 初始前驱节点为-1
        visited[i] = 0;
        // 所有顶点均未访问
```

```
    }
    dist[source] = 0;
    // 源点到自己的距离为 0
    // 执行 Dijkstra 算法
    for (int i = 1; i <= graph->vexnum; ++i)
    {
        // 找到未访问的顶点中距离源点最近的顶点
        int u = -1;
        for (int j = 1; j <= graph->vexnum; ++j)
        {
            if (!visited[j] && (u == -1 || dist[j] < dist[u])) u=j;
        }
        if (dist[u] == INFNITY) break;
        // 剩余顶点不可达
        visited[u] = 1; // 标记为已访问
        // 更新顶点 u 的邻接点的距离
        for (ArcNode *arc = graph->vertex[u].firstarc; arc != NULL; arc =
arc->nextarc)
        {
            int v = arc->adjvex;
            if (!visited[v] && dist[u] + arc->weight < dist[v])
            {
                dist[v] = dist[u] + arc->weight;
                path[v] = u; // 更新前驱节点
            }
        }
    }
    // 输出源点到目标点的最短路径及其距离
    if (dist[target] == INFNITY) printf("从顶点 %d 到顶点 %d 没有路径\n", source,
target);
    else
    {
        printf("从顶点 %d 到顶点 %d 的最短路径长度是 %d\n", source, target,
dist[target]);
        // 输出路径
        printf("路径: ");
        int node = target;
        while (node != -1)
        {
            printf("%c ",graph->vertex[node].data);
            node = path[node];
        }
        printf("\n");
    }
```

```
}
// Floyd-Warshall 算法，计算所有顶点对之间的最短路径
void floyd(AdjList *graph)
{
    int dist[MAX_VERTEX_NUM][MAX_VERTEX_NUM]; // 存储顶点之间的最短路径距离
    int i, j, k;
    // 初始化距离矩阵
    for (i = 1; i <= graph->vexnum; ++i)
    {
        for (j = 1; j <= graph->vexnum; ++j)
        {
            if (i == j) dist[i][j] = 0; // 自己到自己的距离为 0
            else dist[i][j] = INFNITY; // 初始化为无穷大
        }
    }
    // 根据图的邻接表设置初始距离
    for (i = 1; i <= graph->vexnum; ++i)
    {
        for (ArcNode *arc = graph->vertex[i].firstarc; arc != NULL; arc =
arc->nextarc)
        {
            dist[i][arc->adjvex] = arc->weight; // 邻接点的距离即为边的权重
        }
    }
    // Floyd-Warshall 核心算法，动态规划
    for (k = 1; k <= graph->vexnum; ++k)
    {
        for (i = 1; i <= graph->vexnum; ++i)
        {
            for (j = 1; j <= graph->vexnum; ++j)
            {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                {
                    dist[i][j] = dist[i][k] + dist[k][j]; // 更新最短路径
                }
            }
        }
    }
    // 输出结果矩阵
    printf("Floyd-Warshall 结果矩阵:\n");
    for (i = 1; i <= graph->vexnum; ++i)
    {
        for (j = 1; j <= graph->vexnum; ++j)
        {
```

```
            if (dist[i][j] == INFNITY) printf("INF ");
            else printf("%3d ", dist[i][j]);
        }
        printf("\n");
    }
}
int main()
{
    AdjList g;
    build(&g);
    // 选择源点和目标点进行测试
    int source, target;
    printf("请输入源点: ");
    do scanf("%c", &source);while(source=='\n');
    printf("请输入目标点: ");
    do scanf("%c", &target);while(target=='\n');
    dijkstra(&g, source, target);
    printf("\n 以下是 floyd 的输出结果\n");
    floyd(&g);
    return 0;
}
```

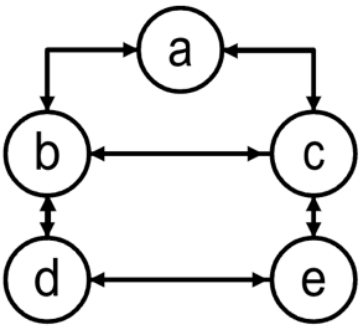
（4）调试分析

测试样例 1:

```
请输入图的顶点数:5
请输入图各个顶点的名称:(不要带有空格)abcde
输入边的数量:6
输入各个顶点之间的关系(边a) (边b) (边权):
a b 1
a c 1
c b 1
b d 1
c e 1
e d 1
请输入源点: a
请输入目标点: e
从顶点 1 到顶点 5 的最短路径长度是 2
路径: e c a

以下是floyd的输出结果
Floyd-Warshall 结果矩阵:
  0   1   1   2   2
  1   0   1   1   2
  1   1   0   2   1
  2   1   2   0   1
  2   2   1   1   0
```

样例 1 的图：

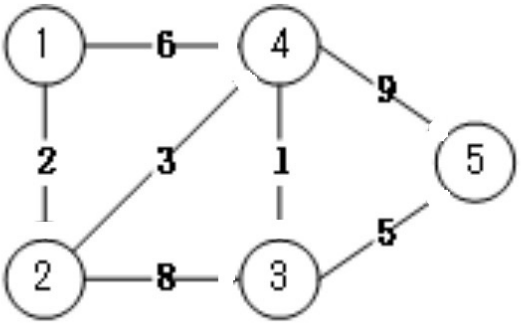


样例 2:

```
请输入图的顶点数:5
请输入图各个顶点的名称:(不要带有空格)12345
输入边的数量:7
输入各个顶点之间的关系(边a)(边b)(边权):
1 4 6
1 2 2
2 3 8
2 4 3
3 4 1
4 5 9
3 5 5
请输入源点: 1
请输入目标点: 5
从顶点 1 到顶点 5 的最短路径长度是 11
路径: 5 3 4 2 1

以下是floyd的输出结果
Floyd-Warshall 结果矩阵:
  0   2   6   5  11
  2   0   4   3   9
  6   4   0   1   5
  5   3   1   0   6
 11   9   5   6   0
```

样例 2 的图(但是建的是双向遍，)：



（5）小结

对于交通系统，我们最频繁需要的问题就是从一个点出发到另外一个点所需要的最短时间，而往往这两个点之间没有直接路径，或者存在堵车的情况。Dijkstra 起到了从一个点到另一个点所有的路径的最短路的计算，让我们能够在最短的时间或者权值内到达。Floyd 可以返回任意两个点之间的最短路，便于直观查看。

15. 查找技术

任务

利用相关查找方法，建立查找表，实现顺序查找、折半查找、二叉排序树上的查找等不同的查找算法。

要求:

1. 输入的数据有多个数据项,有主关键字和次关键字之分(比如学生信息有:学号、姓名、总成绩，学号是主关键字，姓名和总成绩是次关键字)。
2. 输出的形式:查找成功输出找到的数据的所有数据项，查找失败也要给出相应提示信息。
3. 比较不同查找算法的优缺点。

（1）需求分析

分析总结逻辑结构：

查找有多种查找方式，首先是建立在一维数组上的查找，他可以采用顺序查找的方式。而对于有序的一位数组，我们可以尝试使用二分查找，即折半查找的方式，提高查找的效率。也可以建立二叉排序树查找，可以提高平均查找效率(而对于已经有序的序列，不可以提高查找效率，查找效率会退化为O(n)。输入不同的关键字进行查找，那么我们只需要逐个比对即可。返回其他的信息。

分析总结运算集合：

建立查找表

顺序查找

二分查找(折半查找)

二叉排序树上的查找

对学号、姓名等信息的查找

（2）概要设计

存储结构设计：

对于顺序查找，我们可以采用顺序表进行存储，对于二叉排序树的查找，我们可以采用建立树的方式，对于学号、姓名等多关键字查找，我们需要建立结构题数组。

算法设计（流程图）：

（本题类型多样，这里不设计流程图）

（3）详细设计

源代码（注释）

- 查找方式1:顺序查找

```
#include <stdio.h>
// 定义数组大小的常量
const int N = 1e5 + 10;
int q[N];
// 存储数组元素的数组
// 函数：顺序查找数组中的指定值
```

```
int sequentialSearch(int size, int target)
{
    // 从数组末尾开始逐个向前查找目标值
    for (int i = size; i >= 1; --i)
    {
        if (q[i] == target) return i; // 找到目标值，返回索引
    }
}
int main()
{
    int target, n;
    // 读取数组大小
    scanf("%d", &n);
    // 读取数组元素
    for (int i = 1; i <= n; ++i) scanf("%d", &q[i]);
    // 提示用户输入要查找的值
    printf("请输入要查找的值: ");
    scanf("%d", &target);
    // 在数组的第一个位置设置目标值，以便顺序查找能够找到它
    q[0] = target;
    // 调用顺序查找函数
    int index = sequentialSearch(n, target);
    // 输出结果
    if (index) printf("值 %d 在数组中的索引是 %d\n", target, index);
    else printf("值 %d 未找到\n", target);
    return 0;
}
```

- 二叉排序树上的查找

```
#include <stdio.h>
#include <stdlib.h>
// 定义二叉排序树的节点结构
typedef struct TreeNode
{
    int value;
    // 节点值
    struct TreeNode *left;
    // 左子节点
    struct TreeNode *right;
    // 右子节点
} TreeNode;
// 创建一个新的节点
TreeNode* createNode(int value)
{
    TreeNode *newNode = (TreeNode*)malloc(sizeof(TreeNode));
```

```
    newNode->value = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
// 插入节点到二叉排序树
TreeNode* insert(TreeNode *root, int value)
{
    if (root == NULL)
    {
        // 树为空或到达叶子节点位置，创建新节点
        return createNode(value);
    }
    if (value < root->value)
    {
        // 如果值小于当前节点的值，插入到左子树
        root->left = insert(root->left, value);
    }
    else if (value > root->value)
    {
        // 如果值大于当前节点的值，插入到右子树
        root->right = insert(root->right, value);
    }
    // 如果值等于当前节点的值，不插入，返回原节点
    return root;
}
// 查找二叉排序树中的值
TreeNode* search(TreeNode *root, int target)
{
    if (root == NULL || root->value == target)
    {
        // 树为空或找到目标值
        return root;
    }
    if (target < root->value)
    {
        // 如果目标值小于当前节点的值，继续在左子树中查找
        return search(root->left, target);
    } else
    {
        // 如果目标值大于当前节点的值，继续在右子树中查找
        return search(root->right, target);
    }
}
```

```c
// 中序遍历树，输出节点值（用于测试）
void inorderTraversal(TreeNode *root)
{
    if (root != NULL)
    {
        inorderTraversal(root->left);
        printf("%d ", root->value);
        inorderTraversal(root->right);
    }
}

int main()
{
    TreeNode *root = NULL;  // 初始化空树
    int n, value, target;
    printf("请输入节点数量: ");
    scanf("%d", &n);
    printf("请输入 %d 个节点的值:\n", n);
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &value);
        root = insert(root, value);
    }
    printf("请输入要查找的值: ");
    scanf("%d", &target);
    TreeNode *result = search(root, target);
    if (result != NULL) printf("值 %d 在树中存在\n", target);
    else printf("值 %d 不存在于树中\n", target);
    return 0;
}
```

- 二分查找，折半查找

```c
#include <stdio.h>
const int N = 1e5 + 10; // 定义数组的最大容量
int arr[N], n; // 全局数组 arr 和数组大小 n
int main()
{
    // 读取数组的大小
    scanf("%d", &n);
    // 读取数组元素
    for (int i = 1; i <= n; ++i) scanf("%d", &arr[i]);
    // 读取要查找的目标值
    int target;
    scanf("%d", &target);
    // 初始化二分查找的左右边界
```

```
    int l = 1, r = n;
    // 执行二分查找
    while (l < r)
    {
        // 计算中间位置
        int mid = l + r >> 1;
        // 比较中间值与目标值
        if (arr[mid] >= target)
            // 如果中间值大于等于目标值，则目标值可能在右半部分
            l = mid + 1;
        else
            // 如果中间值小于目标值，则目标值可能在左半部分
            r = mid;
    }
    // 检查目标值是否存在于数组中
    if (arr[l] == target)
        // 如果目标值存在，输出其在数组中的位置
        printf("数字 %d 是第 %d 个数字\n", target, l);
    else
        // 如果目标值不存在，输出提示信息
        printf("数字不存在\n");
    return 0;
}
```

- 关键字查找

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define CLS "clear"
// 在 Windows 系统中改成"cls"，在其他系统中使用"clear"
const int MAXN = 1e3 + 10;
// 数据库最大记录数
int data_open;
// 日志模式开关，指示是否启用调试模式
int idx = 0;
// 当前记录数
// 数据库结构体定义
struct DataBase
{
    char num[30]; // 学号
    char name[30]; // 姓名
    int score; // 总成绩
} info[MAXN]; // 数据库数组
// 函数声明
```

```
int read();
// 读取数据函数
int write();
// 写入数据函数
int query_score_by_num(char num[]);
// 根据学号查询成绩
void query_score_output();
// 查询成绩的前端处理函数
void main_tab();
// 主菜单函数
int add_record(char num[], char name[], int score);
// 添加记录函数
// 读取数据函数
int read()
{
    FILE *fp = fopen("data.fzy", "r");
    // 打开数据文件以只读模式
    if (fp == NULL) return 0;
    // 如果文件打开失败，可能是首次运行
    fscanf(fp, "%d", &data_open);
    // 读取日志模式开关状态
    char num[100], name[100];
    // 临时缓冲区
    int score; // 临时缓冲区
    memset(info, 0, sizeof(info));
    // 清空原有数据
    idx = 0; // 重置记录数
    // 从文件中读取数据并存入 info 数组
    while (fscanf(fp, "%s %s %d", num, name, &score) == 3)
    {
        ++idx;
        if (idx >= MAXN - 10) printf("Error, 数据量过大\n");
        // 数据量过多时给出提示
        strcpy(info[idx].num, num);
        // 复制学号
        strcpy(info[idx].name, name);
        // 复制姓名
        info[idx].score = score;
        // 复制成绩
    }
    fclose(fp);
    // 关闭文件
    return 0;
}
```

```
// 写入数据函数
int write() {
    FILE *fp = fopen("data.fzy", "w+");
    // 打开数据文件以写模式
    fprintf(fp, "%d\n", data_open);
    // 写入日志模式开关状态
    // 将所有记录写入文件
    for (int i = 1; i <= idx; ++i)
    {
        fprintf(fp, "%s %s %d\n", info[i].num, info[i].name, info[i].score);
    }
    fclose(fp); // 关闭文件
    printf("数据保存成功\n");
    // 提示保存成功
    return 0;
}

// 根据学号查询成绩
int query_score_by_num(char num[])
{
    for (int i = 1; i <= idx; ++i)
    {
        if (strcmp(info[i].num, num) == 0)
        {
            return i;
            // 返回找到记录的索引
        }
    }
    return -1;
    // 学号不存在
}

// 添加记录函数
int add_record(char num[], char name[], int score)
{
    // 检查学号是否已经存在
    for (int i = 1; i <= idx; ++i)
    {
        if (strcmp(info[i].num, num) == 0) return -1;
        // 学号重复
    }

    // 增加新记录
```

```
    ++idx;
    if (idx >= MAXN) return -2;
    // 数据量超限
    if (strlen(num) > 25) return -3;
    // 学号长度超长
    if (strlen(name) > 25) return -4;
    // 姓名长度超长
    strcpy(info[idx].num, num);
    // 复制学号
    strcpy(info[idx].name, name);
    // 复制姓名
    info[idx].score = score;
    // 复制成绩
    write(); // 写入文件保存数据
    return 0; // 添加成功
}

// 查询成绩的前端处理函数
void query_score_output()
{
    char num[100];
    // 学号或姓名的输入缓冲区
    int choice;
    // 用户选择的查询方式
    printf("请选择查询方式:\n");
    printf("1. 通过学号查询\n2. 通过学生姓名查询\n");
    scanf("%d", &choice);
    // 读取用户选择
    if (choice == 1)
    {
        printf("请输入学号，回车键结束\n");
        scanf("%s", num); // 读取学号
        int id = query_score_by_num(num);
        // 查询学号对应的记录
        if (id == -1)
        {
            if (data_open) system(CLS);
            // 清屏
            printf("学号不存在\n");
            // 提示学号不存在
            main_tab();
            // 返回主菜单
            return;
        }
```

```
            if (data_open) system(CLS);
            // 清屏
            // 输出查询结果
            printf("学号[%s], 姓名[%s]的总成绩为[%d]\n", num, info[id].name,
info[id].score);
        }
        else if (choice == 2)
        {
            printf("请输入姓名，回车键结束\n");
            scanf("%s", num);
            // 读取姓名
            int score = -1;
            // 遍历记录查找姓名
            for (int i = 1; i <= idx; ++i)
            {
                if (strcmp(info[i].name, num) == 0)
                {
                    printf("学号[%s], 姓名[%s]的总成绩为[%d]\n", info[i].num,
info[i].name, info[i].score);
                    score = 0;
                    // 找到记录
                }
            }
            if (score == -1)
            {
                if (data_open) system(CLS);
                // 清屏
                printf("姓名不存在\n");
                // 提示姓名不存在
                main_tab();
                // 返回主菜单
                return;
            }
            if (data_open) system(CLS);
            // 清屏
        }
        else
        {
            printf("无效的选择\n");
            // 提示无效选择
        }
        main_tab();
        // 返回主菜单
}
```

```
// 主菜单函数
void main_tab() {
    // 打印主界面
    for (int i = 1; i <= 50; ++i) printf("#");
    for (int i = 1; i <= 2; ++i) printf("\n");
    printf("   学生成绩管理程序\n");
    printf("   请输入以下数字，选择您所要进行的操作：\n");
    printf("   1. 从磁盘中读取记录\n");
    printf("   2. 向磁盘中写入记录\n");
    printf("   3. 查询成绩\n");
    printf("   4. 添加记录\n");
    printf("   5. 退出系统\n");
    for (int i = 1; i <= 1; ++i) printf("\n");
    for (int i = 1; i <= 50; ++i) printf("#");
    printf("\n 请输入您的选择(回车键确认)： ");

    // 主菜单循环
    while (1)
    {
        int choose;
        scanf("%d", &choose);
        // 读取用户选择
        switch (choose)
        {
            case 1: read(); break;
            // 读数据文件
            case 2: write(); break;
            // 写数据文件
            case 3: query_score_output(); break;
            // 查询成绩
            case 4:
            {
                char num[30], name[30];
                int score;
                printf("请输入学号、姓名和总成绩，使用空格分开\n");
                scanf("%s %s %d", num, name, &score);
                // 读取用户输入
                int result = add_record(num, name, score);
                // 添加记录
                // 输出操作结果
                if (result == -1) printf("学号已存在\n");
                else if (result == -2) printf("数据量超限\n");
                else if (result == -3) printf("学号超长\n");
                else if (result == -4) printf("姓名超长\n");
```

```
                else printf("记录添加成功\n");
                break;
            }
            case 5: return;
            // 退出程序
            default: printf("无效的选择\n"); break;
            // 提示无效选择
        }
    }
}

// 主函数
int main()
{
    read();
    // 读取初始数据
    if (data_open) system(CLS);
    // 清屏（调试模式下）
    main_tab();
    // 进入主菜单
    printf("已退出系统\n");
    // 程序退出提示
    return 0;
}
```

（4）调试分析

对于顺序查找，样例如下：

数组长度：90

数组元素具体值：72 58 1 64 32 87 50 6 95 43 91 83 39 89 10 40 14 76 65 18 21 90 8 59 53 11 70 33 42 27 97 77 86 4 5 60 69 19 23 26 46 80 52 2 35 98 71 30 84 15 61 13 57 17 73 47 62 93 22 66 49 79 67 81 34 7 54 45 25 41 92 68 96 48 28 82 38 12 51 3 29 63 74 31 56 44 24 99 78 16

查找元素：77

输出结果截图：

```
90 72 58 1 64 32 87 50 6 95 43 91 83 39 89 10 40 14 76 65 18 21 90 8 59 53 11 70 33
42 27 97 77 86 4 5 60 69 19 23 26 46 80 52 2 35 98 71 30 84 15 61 13 57 17 73 47 62
93 22 66 49 79 67 81 34 7 54 45 25 41 92 68 96 48 28 82 38 12 51 3 29 63 74 31 56 44
 24 99 78 16
请输入要查找的值：77
值 77 在数组中的索引是 32
```

对于二叉排序树上的查找，样例如下：

结点数量：10

结点的值：12 32 14 56 23 20 98 26 38 28

查找的值：5 26

```
请输入节点数量: 10
请输入 10 个节点的值:
12 32 14 56 23 20 98 26 38 28
请输入要查找的值: 5
值 5 不存在于树中
```

```
请输入节点数量: 10
请输入 10 个节点的值:
12 32 14 56 23 20 98 26 38 28
请输入要查找的值: 26
值 26 在树中存在
```

对于二分查找，样例如下：

数据长度：10

数据：1 3 5 45 76 87 98 100 105 120

查找：98 97

```
10
1 3 5 45 76 87 98 100 105 120
98
数字 98 是第 7 个数字
```

```
10
1 3 5 45 76 87 98 100 105 120
97
数字不存在
```

对于关键字查找，样例如下：

下图是一个基于学号的错误查找

```
####################################################

    学生成绩管理程序
    请输入以下数字，选择您所要进行的操作：
    1. 从磁盘中读取记录
    2. 向磁盘中写入记录
    3. 查询成绩
    4. 添加记录
    5. 退出系统

####################################################
请输入您的选择(回车键确认): 3
请选择查询方式：
1.通过学号查询
2.通过学生姓名查询
1
请输入学号，回车键结束
123
学号不存在
```

此时 data.fzy 中的数据如下：

```
0
001 cwj 99
002 cjr 96
003 cxr 93
004 dsy 100
005 fzy 98
006 fw 92
007 gjl 95
008 fzy 94
```

下图是一个基于学号的正确查找：

```
##################################################

    学生成绩管理程序
    请输入以下数字，选择您所要进行的操作：
    1. 从磁盘中读取记录
    2. 向磁盘中写入记录
    3. 查询成绩
    4. 添加记录
    5. 退出系统

##################################################
请输入您的选择(回车键确认)： 3
请选择查询方式：
1.通过学号查询
2.通过学生姓名查询
1
请输入学号，回车键结束
002
学号[002],姓名[cjr]的总成绩为[96]
```

下图是一个基于姓名的正确查找：

```
##################################################

    学生成绩管理程序
    请输入以下数字，选择您所要进行的操作：
    1. 从磁盘中读取记录
    2. 向磁盘中写入记录
    3. 查询成绩
    4. 添加记录
    5. 退出系统

##################################################
请输入您的选择(回车键确认)： 3
请选择查询方式：
1.通过学号查询
2.通过学生姓名查询
2
请输入姓名，回车键结束
dsy
学号[004],姓名[dsy]的总成绩为[100]
```

下图是一个一对多关系的正确查找：

```
#####################################################

    学生成绩管理程序
    请输入以下数字，选择您所要进行的操作：
    1. 从磁盘中读取记录
    2. 向磁盘中写入记录
    3. 查询成绩
    4. 添加记录
    5. 退出系统

#####################################################
请输入您的选择(回车键确认)：3
请选择查询方式：
1.通过学号查询
2.通过学生姓名查询
2
请输入姓名，回车键结束
fzy
学号[005],姓名[fzy]的总成绩为[98]
学号[008],姓名[fzy]的总成绩为[94]
```

下图是一个基于姓名的错误查找：

```
#####################################################

    学生成绩管理程序
    请输入以下数字，选择您所要进行的操作：
    1. 从磁盘中读取记录
    2. 向磁盘中写入记录
    3. 查询成绩
    4. 添加记录
    5. 退出系统

#####################################################
请输入您的选择(回车键确认)：3
请选择查询方式：
1.通过学号查询
2.通过学生姓名查询
2
请输入姓名，回车键结束
1
姓名不存在
```

然后我们向数据库中添加几组数据：

009 bmy 100

010 ll 95

011 lsy 93

012 lyh 96

此时数据库变成如下状态：

```
0
001 cwj 99
002 cjr 96
003 cxr 93
004 dsy 100
005 fzy 98
006 fw 92
007 gjl 95
008 fzy 94
009 bmy 100
010 ll 95
011 lsy 93
012 lyh 96
```

以添加最后一个记录为例：

```
####################################################

    学生成绩管理程序
    请输入以下数字，选择您所要进行的操作：
    1. 从磁盘中读取记录
    2. 向磁盘中写入记录
    3. 查询成绩
    4. 添加记录
    5. 退出系统

####################################################
请输入您的选择(回车键确认)：4
请输入学号、姓名和总成绩，使用空格分开
012 lyh 96
数据保存成功
记录添加成功
```

当然记录的添加也支持中文输入

（5）小结

在基于顺序表的查找方面，我使用了一个哨兵作为查找终止的条件，如果说从最后一个数字向前查找，都没有查找到我们所需要查找的数值，那么程序可以直接返回 false，不需要每一次都判断是否已经查找到了表尾，有效提高了查找的效率，减少了比较量。

对于二分查找，这是一种基于线性表的快速查找方式，可以将线性阶的查找工作量降低到对阶，有效降低了时间复杂度。我认为二分查找不一定需要序列具有单调性，只需要一边符合某一个性质，另一边不符合某一个性质，那么就可以使用二分来寻找边界。

对于树形结构而言，树形结构可以降低查找和插入数据的时间复杂度，也可以用于降低存储空间，实现存储空间的动态分配。但是如果插入时候本身就是有序的数字，那么树形结构的查找效率会退化为 O(n)，此时我们应该引入平衡二叉树 AVL 来解决这个问题。

在基于关键字的查找方面，不同于传统的信息管理系统，在本次实习过程中对于学号我们也采用了 char 类型数组来声明，这样做可以避免因为精度问题导致的数值改变，更符合现代数据库中以字符串数组作为主键的模式。

16. 排序技术

任务

利用相关排序算法，将用户随机输入的一组整数(20<=个数<=50)按递增的顺序排好。

要求:

1. 输入的数据形式为整数。
2. 输出的形式:数字大小逐个递增的数列。
3. 比较不同排序算法的优缺点。

（1）需求分析

分析总结逻辑结构：

排序，用数组存储，将一些杂乱无章的数字依照升序排列即可

分析总结运算集合：

交换两个数的位置

（2）概要设计

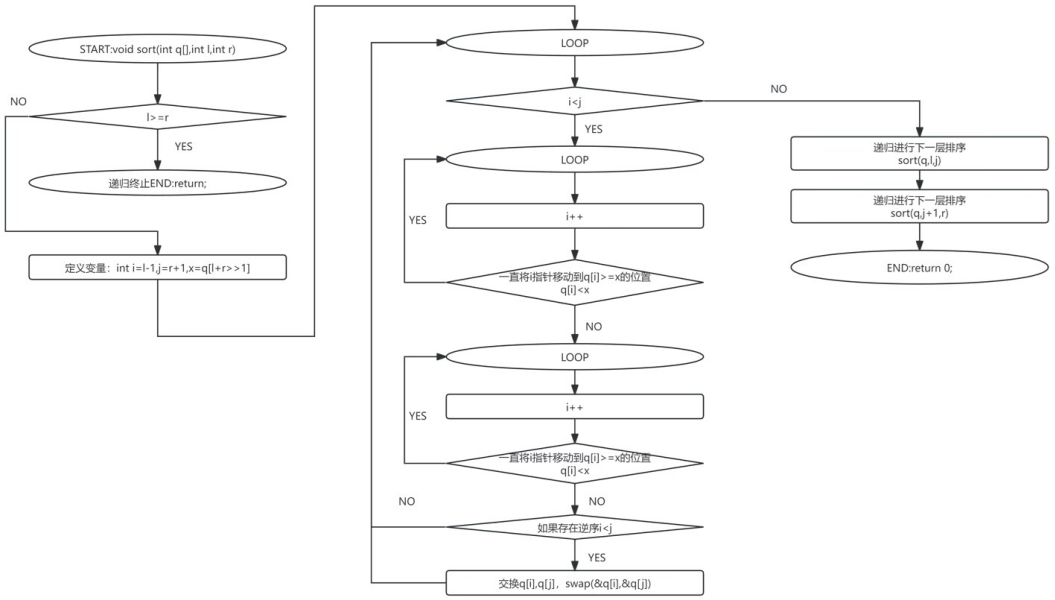
存储结构设计：

顺序存储：数据元素按顺序存放在一个连续的存储空间中，常用数组表示。适合直接访问，如快速排序、归并排序等。

辅助数组：在归并排序等算法中使用，用于暂存中间结果。

算法设计（流程图）：

以快速排序为例，快速排序利用双指针，从两边往中间靠，如果遇到了一个数字位于基准值错误的一边，那么停下来将两个值进行交换即可。若双指针相遇，则代表数组已经被分成了2边，一遍大于一个值，另一边小于一个值：



（3）详细设计

源代码（注释）

- 快速排序

```
#include<stdio.h>
// 交换两个整数的值
void swap(int *a, int *b)
{
    int tmp = *a;  // 临时变量存储 a 的值
    *a = *b;       // 将 b 的值赋给 a
    *b = tmp;      // 将临时变量的值（即原来的 a 值）赋给 b
}
// 实现快速排序的递归函数
void quick_sort(int q[], int l, int r)
{
    if(l >= r) return;
    // 如果子数组长度为 0 或 1，直接返回
    int mid = q[l + r >> 1];
    // 选择中间位置的元素作为基准值
    int i = l - 1, j = r + 1;
    // 初始化两个指针：i 指向左边界外，j 指向右边界外
    while(i < j)
    // 开始分区过程
    {
        do ++i; while(q[i] < mid);
        // 从左向右找到第一个大于等于基准值的元素
        do --j; while(q[j] > mid);
        // 从右向左找到第一个小于等于基准值的元素
        if(i < j) swap(&q[i], &q[j]);
        // 如果找到的一对元素的位置不正确，交换它们
    }
    // 递归地对分区后的左右部分进行快速排序
    quick_sort(q, l, i - 1);
    // 对左半部分排序
    quick_sort(q, j + 1, r);
    // 对右半部分排序
}
//主程序
int main()
{
    int n, num[100];
    // 声明变量：n 表示数组的大小，num 数组存储待排序的数
    scanf("%d", &n);
    // 读取数组的大小
    for(int i = 0; i < n; ++i) scanf("%d", &num[i]);
```

```
    // 读取数组中的 n 个元素
    quick_sort(num, 0, n - 1);
     // 对整个数组进行快速排序
    for(int i = 0; i < n; ++i) printf("%d ", num[i]);
    // 输出排序后的数组
    return 0;  // 程序结束
}
```

- 归并排序

```
#include<stdio.h>
// 声明全局变量
int num[100010], tmp[100010], n;
// 归并排序的递归函数
void merge_sort(int l, int r)
{
    if(l >= r) return;
    // 如果子数组长度为 0 或 1，直接返回，递归结束条件
    int mid=l+r>>1;
    // 计算中间位置，用于将数组分成两部分
    merge_sort(l, mid);
    // 递归排序左半部分
    merge_sort(mid + 1, r);
    // 递归排序右半部分
    int k = 1, i = l, j = mid + 1;
    // 初始化指针：k 用于临时数组，i 用于左半部分，j 用于右半部分
    // 合并两个有序的子数组
    while(i <= mid && j <= r)
    {
        if(num[i] < num[j]) tmp[k++] = num[i++];
        // 如果左半部分当前元素较小，放入临时数组
        else tmp[k++] = num[j++];
        // 否则放入右半部分当前元素
    }
    // 如果左半部分有剩余元素，继续放入临时数组
    while(i <= mid) tmp[k++] = num[i++];
    // 如果右半部分有剩余元素，继续放入临时数组
    while(j <= r) tmp[k++] = num[j++];
    // 将临时数组中的元素拷贝回原数组对应位置
    for(i = 1, j = l; j <= r; i++, j++) num[j] = tmp[i];
}
int main()
{
    scanf("%d", &n);
    // 读取数组大小 n
    for(int i = 1; i <= n; i++) scanf("%d", &num[i]);
```

```
    // 读取 n 个元素到 num 数组中
    merge_sort(1, n);
    // 对整个数组进行归并排序
    for(int i = 1; i <= n; i++) printf("%d ", num[i]);
    // 输出排序后的数组
    return 0;
}
```

● 桶排序

```
#include<stdio.h>
#include<string.h>
int main()
{
    int n,a[100],b;
    memset(a,0,sizeof(a));
    scanf("%d",&n);
    for(int i=1;i<=n;i++)
    {
        scanf("%d",&b);
        if(b>100||b<0)
        {
            printf("输入数据错误");
            return 0;
        }
        a[b]++;
    }
    for(int i=0;i<=99;i++)
    {
        while(a[i]>0)
        {
            printf("%d ",i);
            a[i]--;
        }
    }
    return 0;
}
```

● 冒泡排序

```
#include<stdio.h>
const int N=10;
int a[N];
int main()
{
    for(int i=0;i<10;i++) scanf("%d",&a[i]);
    for(int i=0;i<10;i++)
    {
```

```
        for(int j=1;j<10;j++)
        {
            if(a[j-1]>a[j])
            {
                int tmp=a[j-1];
                a[j-1]=a[j];
                a[j]=tmp;
            }
        }
    }
    for(int i=0;i<10;i++) printf("%d ",a[i]);
    return 0;
}
```

- 拓扑排序

```
#include<stdio.h>
#include<string.h>
const int N = 100010;
int e[N], ne[N], idx;
int h[N];
int q[N], hh = 0, tt = -1;
int n, m;
int d[N];
void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}
void topsort()
{
    for(int i = 1; i <= n; i++) if(d[i] == 0) q[++tt] = i;
    while(tt >= hh)
    {
        int a = q[hh++];
        for(int i = h[a]; i != -1; i = ne[i])
        {
            int b = e[i];
            d[b]--;
            if(d[b] == 0) q[++tt] = b;
        }
    }
    if(tt == n - 1)
    {
        for(int i = 0; i < n; i++) printf("%d ",q[i]);
    }
    else printf("-1");
```

```
}
int main(){
    scanf("%d%d",&n,&m);
    memset(h, -1, sizeof h);
    while (m -- )
    {
        int a, b;
        scanf("%d%d",&a,&b);
        d[b]++;
        add(a, b);
    }
    topsort();
    return 0;
}
```

- 选择排序

```
#include<stdio.h>
const int N=1e5+10;
int arr[N],n;
int main()
{
    scanf("%d",&n);
    for(int i=0;i<n;i++) scanf("%d",&arr[i]);
    for(int i=0;i<n;i++)
    {
        int tmp=i;
        for(int j=i+1;j<n;j++)
        {
            if(arr[j]<arr[tmp]) tmp=j;
        }
        int mid=arr[tmp];
        arr[tmp]=arr[i];
        arr[i]=mid;
    }
    for(int i=0;i<n;i++) printf("%d ",arr[i]);
    return 0;
}
```

- 插入排序

```
#include<stdio.h>
const int N=1e5+10;
int arr[N],n;
int main()
{
    scanf("%d",&n);
    for(int i=0;i<n;i++) scanf("%d",&arr[i]);
```

```
    for(int i=0;i<n;i++)
    {
        for(int j=i;j>=1;j--)
        {
            if(arr[j]<arr[j-1])
            {
                int tmp=arr[j];
                arr[j]=arr[j-1];
                arr[j-1]=tmp;
            }
        }
    }
    for(int i=0;i<n;i++) printf("%d ",arr[i]);
    return 0;
}
```

- 希尔排序

```
#include <stdio.h>
// 希尔排序函数
void shell_sort(int arr[], int n)
{
    // 选择初始步长，通常为数组长度的一半
    for (int gap = n / 2; gap > 0; gap /= 2)
    {
        // 对每个步长 gap 进行分组，并对每组进行插入排序
        for (int i = gap; i < n; i++)
        {
            int temp = arr[i];
            int j;
            // 插入排序部分，将 temp 插入到它在分组内的合适位置
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) arr[j] = arr[j
- gap];
            arr[j] = temp;
        }
    }
}
// 主函数
int main() {
    int n;
    printf("请输入数组的元素数量: ");
    scanf("%d", &n);
    int arr[n];
    printf("请输入数组的元素:\n");
    for (int i = 0; i < n; i++) scanf("%d", &arr[i]);
    shell_sort(arr, n);
```

```
    printf("排序后的数组:\n");
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    return 0;
}
```

（4）调试分析

所有排序算法的测试数据均相同：

测试数据 1:

20

23 23 28 29 19 29 39 20 10 9 18 29 57 97 45 99 26 38 28 39

测试数据 2:

50

64 13 78 52 16 89 28 4 85 66 12 19 73 45 9 39 82 96 6 35 57 7 24 61 92 77 41 33 22 69 3 83 94 25 48 59 1 30 53 98 74 2 50 88 34 5 87 17 31 46

测试数据 3:

40

77 15 42 89 61 37 81 23 54 68 10 90 28 34 76 2 55 92 43 60 4 99 7 21 85 17 72 39 57 63 30 18 11 74 3 86 25 50 96 8

对于桶排序，我给出了另一组特殊数据：

40

-24 58 -3 83 -71 48 -99 -53 100 -26 -16 -78 14 66 -34 -42 9 5 71 39 7 -11 -94 30 -17 24 4 93 -64 84 -81 50 -45 15 63 -18 80 -90 -7 12

测试数据 1:

```
20
23 23 28 29 19 29 39 20 10 9 18 29 57 97 45 99 26 38 28 39
9 10 18 19 20 23 23 26 28 28 29 29 29 38 39 39 45 57 97 99
```

测试数据 2:

```
50
64 13 78 52 16 89 28 4 85 66 12 19 73 45 9 39 82 96 6 35 57 7 24 61 92 77 41 33 22 69 3 83 94 25 48 59 1 30 53 98 74 2 50 88 34
 5 87 17 31 46
1 2 3 4 5 6 7 9 12 13 16 17 19 22 24 25 28 30 31 33 34 35 39 41 45 46 48 50 52 53 57 59 61 64 66 69 73 74 77 78 82 83 85 87 88
89 92 94 96 98
```

测试数据 3:

```
40
77 15 42 89 61 37 81 23 54 68 10 90 28 34 76 2 55 92 43 60 4 99 7 21 85 17 72 39 57 63 30 18 11 74 3 86 25 50 96 8
2 3 4 7 8 10 11 15 17 18 21 23 25 28 30 34 37 39 42 43 50 54 55 57 60 61 63 68 72 74 76 77 81 85 86 89 90 92 96 99
```

对于桶排序，测试数据 4:

```
40
-24 58 -3 83 -71 48 -99 -53 100 -26 -16 -78 14 66 -34 -42 9 5 71 39 7 -11 -94 30 -17 24 4 93 -64 84 -81 50 -45 15 63 -18 80 -9(
 -7 12
输入数据错误
```

（5）小结

不同的排序方法在时间复杂度、空间复杂度上都各有优劣。比较类排序往往拥有更高的时间复杂度，其他的排序方法往往有较高的空间复杂度，没有一种排序方法是两全其美的。

冒泡排序：是一种简单的交换排序，通过重复地遍历数组，将相邻的未排序部分逐步推到数组的末尾。它的时间复杂度是 O(n^2)，且由于每次只涉及相邻元素的交换，空间复杂度为 O(1)。它是稳定的，但性能较差，不适合大型数字。

选择排序：主要思路是每次从未排序部分选择最小（或最大）的元素，放到已排序部分的末尾。它的时间复杂度为O(n^2)，与冒泡排序类似，但它只进行n次交换，因此在某些情况下可能比冒泡排序更快。不过它同样不适合大规模数据，且不稳定。
插入排序：是通过将未排序的元素插入到已排序的部分中来实现的。它的时间复杂度为O(n^2)，但对几乎已排序的数据表现良好，达到O(n)的效率。空间复杂度为O(1)，且它是稳定的，常用于小规模或部分有序的数组。
希尔排序：是插入排序的改进版本，它通过选择不同的步长（gap）对数组进行分组，然后对每组进行插入排序。随着步长的减少，数组逐渐变得有序，最后一步是普通的插入排序。希尔排序的时间复杂度依赖于步长的选择，通常介于O(n^1.3)到O(n^2)之间，且不稳定。适合中等规模的数据集。
归并排序：是一种基于分治法的排序算法，它将数组递归地分成两半，分别进行排序，然后合并。归并排序的时间复杂度是O(n log n)，空间复杂度为O(n)，它是稳定的。由于其稳定性和时间复杂度，它适合处理大规模数据，但额外的空间开销较大。
快速排序：也是一种基于分治法的排序算法，通过选择一个基准元素将数组分成两部分，一部分比基准小，另一部分比基准大，然后递归地排序。它的平均时间复杂度为O(n log n)，但在最坏情况下退化为O(n^2)。快速排序通常是不稳定的，但由于其较小的常数因子和O(log n)的空间复杂度，它是实际中最常用的排序算法之一，尤其适合大数据集。
以下是通过一个表格直观反映各种排序方法优劣：

序号	排序名称	时间复杂度	空间复杂度	稳定性	缺点
1	快速排序	O(nlogn)	最优O(logn)，最坏O(n)	不稳定	空间复杂度高
2	归并排序	O(nlogn)	O(n)	稳定	空间复杂度高
3	桶排序	O(n)	由值域决定	稳定	空间复杂度高
4	冒泡排序	O(n^2)	O(1)	稳定	时间复杂度高
5	拓扑排序	O(n+e)	O(n+e)	不稳定	空间复杂度高
6	选择排序	O(n^2)	O(1)	不稳定	时间复杂度高
7	插入排序	O(n^2)	O(1)	稳定	时间复杂度高
8	希尔排序	O(n^1.3)	O(1)	不稳定	特殊情况下时间复杂度高