实验报告：

1.2 约瑟夫环 & 5.6题 最小生成树问题

钟 赟 2016K8009915009

魏旭晨 2016K8009908010

黄梦依 2016K8009922012

2018.07.08

PART1：1.2题 最小生成树问题

1. **需求分析**
2. **问题描述**

约瑟夫（Joseph）问题的一种描述为：编号为1，2，…，n的n个人按顺时针方向围坐一圈，每人持有一个密码（正整数），一开始任选一个正整数作为报数上限值m，从第一个人开始按顺时针方向自1开始顺序报数，报到m时停止报数。报m的人出列，将他的密码作为新的m值，从他在顺时针方向上的下一个人开始重新从1报数，如此下去，直至所有人全部出列为止。

1. 本实验程序施行时，要求用户指定初始报数上限值，然后读入各人的密码，要求人数上限设为n≤30.
2. 约瑟夫环是一个古老的数学问题，在计算机实验实现中，要求利用程序的简介迅速，实现数学问题的解法。本次实验中，仅需使用单向循环链表即可解决，不要求“头结点”。而在建立循环链表是，由于约瑟夫问题中的人数，即链表上的元素只会越来越少。通过不断删除链表中的元素的同时记录下它原先所处的位置，从而解决这个问题。
3. 用指针num指向当前的结点，用指针next指向链表的下一个结点。由于每个人都密码都已提前确定，可由编程实现一一对应的关系。从而实现，结点，位置，密码和循环方向的一一对应。实验中，建立每个结点的前结点与后结点的联系。由于是循环计数，所以才采用循环列表这个线性表方式，将链表头尾相连。
4. 程序执行的命令包括：
5. 构造单向循环链表；
6. 将链表节点与密码对应；
7. 按照出列的顺序依次打印出个人的标号
8. 测试数据
9. m初值取20；n=7；7个人的密码依次为：3，1，7，2，4，8，4

出列的顺序为6，1，4，7，2，3，5 演算正确

1. m初值取14；n=14;14个人的密码依次为：4，2，6，7，8，4，2，5，7，14，25，2，1，8

出列的顺序为14，8，13，1，5，3，11，6，12，4，10，7，2，9 演算正确。

1. **概要设计**
2. 抽象数据类型定义为：

**ADT LNode**{

**数据对象：**D={ai|ai∈intset,i=1,2,…,n,n≥0}

**数据关系：**R1={<ai-1,ai>|ai-1,ai∈D,i=1,2,…,n}

**基本操作：**

Createlinklist(&J, n)

操作结果：构造一个有n个结点的单向环表J。

Free(q)

初始条件：单向环表已存在且q为表上一结点。

操作结果：将环表上q值释放并依旧构成一个单向环表。

calculate(head,m,n);

初始条件：单向环表已存在，head为头指针>0，m>0，n=环表节点数

操作结果：返回约瑟夫环的计算结果。

Print()：

初始条件：单向环表已存在，程序通过，进行输入。

操作结果：进行输出结果打印。

}**ADT LNode**

2、本程序包含以下几个模块：

（1）主程序模块

int main()

{

初始化；

输入数据；

构建clinklist循环链表；

调用joseph函数演算；

打印结果；

}

1. 约瑟夫演算模块——实现约瑟夫问题的求值和演算；
2. 结点结构单元模块——定义链表循环和依次的结点构建。
3. 输入输出模块——对计算结果进行打印输出。
4. 模块调用关系

主程序模块

约瑟夫演算模块 结点结构单元模块

输入输出模块

1. **详细设计**
2. 元素类型、结点类型和指针类型

struct LNode

{

int num; /\*s声明数据类型\*/

struct LNode \*next; /\*声明next指针\*/

}

LNode;

typedef struct LNode node;

node \*createlinklist(int n) /\*构建单向环表\*/

{

node \*head,\*p,\*q; /\*声明head指针，p，q指针用于记录数据\*/

int i=1;

head=p=(struct L\*)malloc(sizeof(struct LNode));

head->num=i;

p=head;

for(i=2;i<=n;i++) /\*头结点后依次构建链表\*/

{

q=(struct LNode\*)malloc(sizeof(struct LNode));

if(q==NULL)

return 0; /\*实现空表退出游戏\*/

p->next=q; /\*构建表中数据相关关系\*/

p=q;

p->num=i;

}

p->next=head;

return head; /\*实现链表循环\*/

}

2、约瑟夫核心算法

void joseph(node \*p,int m,int n)

{

int k,i,j,l;

node \*q;

int a[100];

for (k=0;k<n;k++){

scanf("%d",&a[k]); /\*对应密码和链表\*/

while (a[k]<=0){

printf("密码错误，重新输入：");

scanf("%d",&a[k]); /\*排除数据错误情况\*/

}

for(i=1;i<=n;i++){

if(m==1){m=m+n-i+1;}

else{m=m;} /\*排除特殊值的干扰\*/

for(j=1;j<m-1;j++){p=p->next;} /\*指针移动\*/

q=p->next;

p->next=q->next;

p=p->next; /\*出列后构建新的单向循环链表\*/

printf("%d ",q->num); /\*打印每次的出列数\*/

m=a[q->num-1]; /\*获得新密码\*/

free(q);

} /\*实现出列数的释放\*/

printf("为依次出列人顺序。\n"); /\*出列顺序输出值打印和观察\*/

}

3、输入输出模块

void print()

{

printf("输入m值、圆桌人数："); /\*提醒输入初始值\*/

scanf("%d%d",&m,&n);

while(m<1 || n<1 || n>30 )

{

printf("数据不适合，重新输入：");/\*提醒数据的范围\*/

scanf("%d%d",&m,&n);

}

printf("输入依次的密码："); /\*提醒输入密码\*/

}

4、主函数模块

int main()

{

int n, m;

node \*head=NULL; /\*确定头指针\*/

print();

head=createlinklist(n); /\*构建单向循环链表\*/

joseph(head,m,n); /\*进行Joseph计算\*/

system("pause");

}

**四、调试分析**

1、早期程序在密码值的输入上需要一个个手动单独输入，极不方便也不符合实验要求，且导致输出数值没法直观看出。在完成逻辑基础后对操作界面进行简化。

2、实验过程中对数据有条件上和常识上的要求，早期程序没有对数据的筛选功能，之后做了补充。

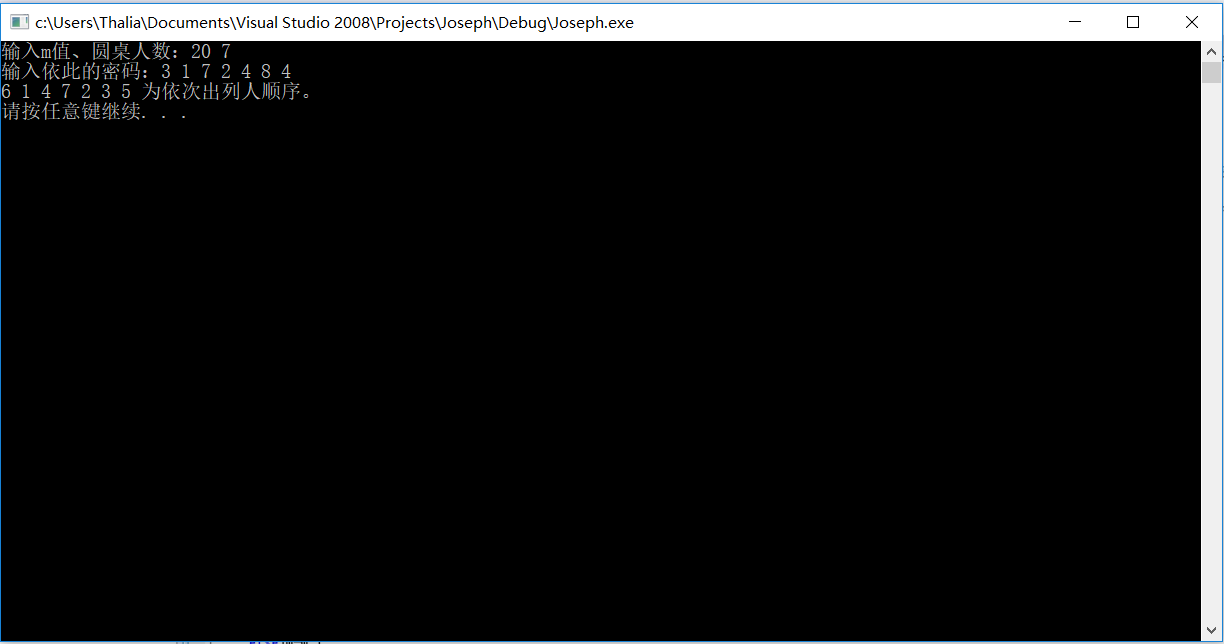
3、约瑟夫模块部分，为了实现数据在链表中的转移需要用到while循环，在循环外进行指针的转移。问题出现在用1 1 1……一组数据测试时，数据的特殊性导致逻辑重复。在这个基础上我们对特殊数据进行了处理，实现了逻辑的严密性。

4、实验后有关于在顺序结构中实现约瑟夫环的思考，顺序储存结构在每一此指针移动到链表的末尾不能向循环单链表结构一样自然地指向链表的前端。事实上可以利用计算机便于计算的优点，计算机计算有补码的概念，在知道当下的实际人数和移动距离是，是可以通过取同余来拒绝溢出的。

**五、用户手册**

1、本次实验的运行环境为VS2008编译器，执行文件Joseph.exe

2、进入演示程序后即显示文本方式的用户界面：



1. **测试结果**

1、m初值取20；n=7；7个人的密码依次为：3，1，7，2，4，8，4

出列的顺序为6，1，4，7，2，3，5

2、m初值取14；n=14;14个人的密码依次为：4，2，6，7，8，4，2，5，7，14，25，2，1，8

出列的顺序为14，8，13，1，5，3，11，6，12，4，10，7，2，9

3、m初值取3；n=5；5个人的密码依次为：2，3，4，5，6

出列的顺序为3，2，1，5，4

4、m初值取3；n=5；5个人的密码依次为：2，3，4，5，6

出列的顺序为3，2，1，5，4

5、m初值取7；n=2；2个人的密码依次为：1，9

出列的顺序为1，2

6、m初值取1；n=6；6个人的密码依次为：1，1，1，1，1，1

出列的顺序为1，2，3，4，5，6

7、m初值取1；n=6；6个人的密码依次为：1，1，1，1，1，1

出列的顺序为1，2，3，4，5，6

8、m初值取3；n=3；3个人的密码依次为：1，0，3

输出结果为：密码错误，请重新输入。

9、m初值取0；n=5；

输出结果为：数据不适合，请重新输入。

1. m初值取4；n=40；

输出结果为：数据不适合，请重新输入。

**七、附录**

实验心得：

实现约瑟夫算法的程序和思考并不复杂，但是其中存在着很多细节需要把控。实验不算高难度，但对于数据结构的基础知识依然需要把握要点，而且从多方面考察将基础知识和实际操作联系在了一起。虽然这个实验内容不算很完备，但对于数据结构主要框架的构建仍然可以管中窥豹可见一斑。

同时，实验用一个古老的数学问题引发现代计算机知识的思考，灵活地将古老人文故事、课堂的理论知识和现代的计算思维结合在了一起，其中蕴含了很多值得深思的地方。依然记得曾经大量的计算是对人类脑力的禁锢和对发展的限制，但日益高超的现代技术使得思维的瓶颈被打破，而创新的能力也稳步开花。

其中实验源码见代码joseph.c。

PART2：5.6题 最小生成树问题

题目：编制一个以最低经济代价建设城市之间的通信网的程序

1. **需求分析**
2. 将n个城市抽象成具有n个顶点的无向图，边上的权值代表两地之间的经济损耗。
3. 本程序的目的是提供经济代价最低的通信网络。利用克鲁斯卡尔算法，根据不同城市之间的经济代价，输出经济消耗最低的通信网络图。
4. 测试数据（附后）。
5. **概要设计**
6. **抽象数据类型图的定义如下：**

**ADT MFset** {

**数据对象：**若设S是MFset型的集合，则它由n（n > 0）个子集Si（i = 1, 2, …, n）构成，每个子集的成员都是子界[-maxnumber, maxnumber]内的整数；

**数据关系：**

**基本操作：**

Initial(&S, n, x1, x2, …, xn);

操作结果：初始化操作。构造一个由n个子集（每个子集只含单个成员xi）构成的集合S。

Find(S, x);

初始条件：S是已存在的集合，x是S中某个子集的成员。

操作结果：查找函数。确定S中x所属子集Si。

Merge(&S, i, j);

初始条件：Si和Sj是S中互不相交的两个非空集合。

操作结果：归并操作。将Si和Sj中的一个并入另一个。

} **ADT** MFset;

1. **主程序**

**int main**()

{

初始化；

接受命令（）

处理命令；

return 0；

}

1. **模块调用关系**

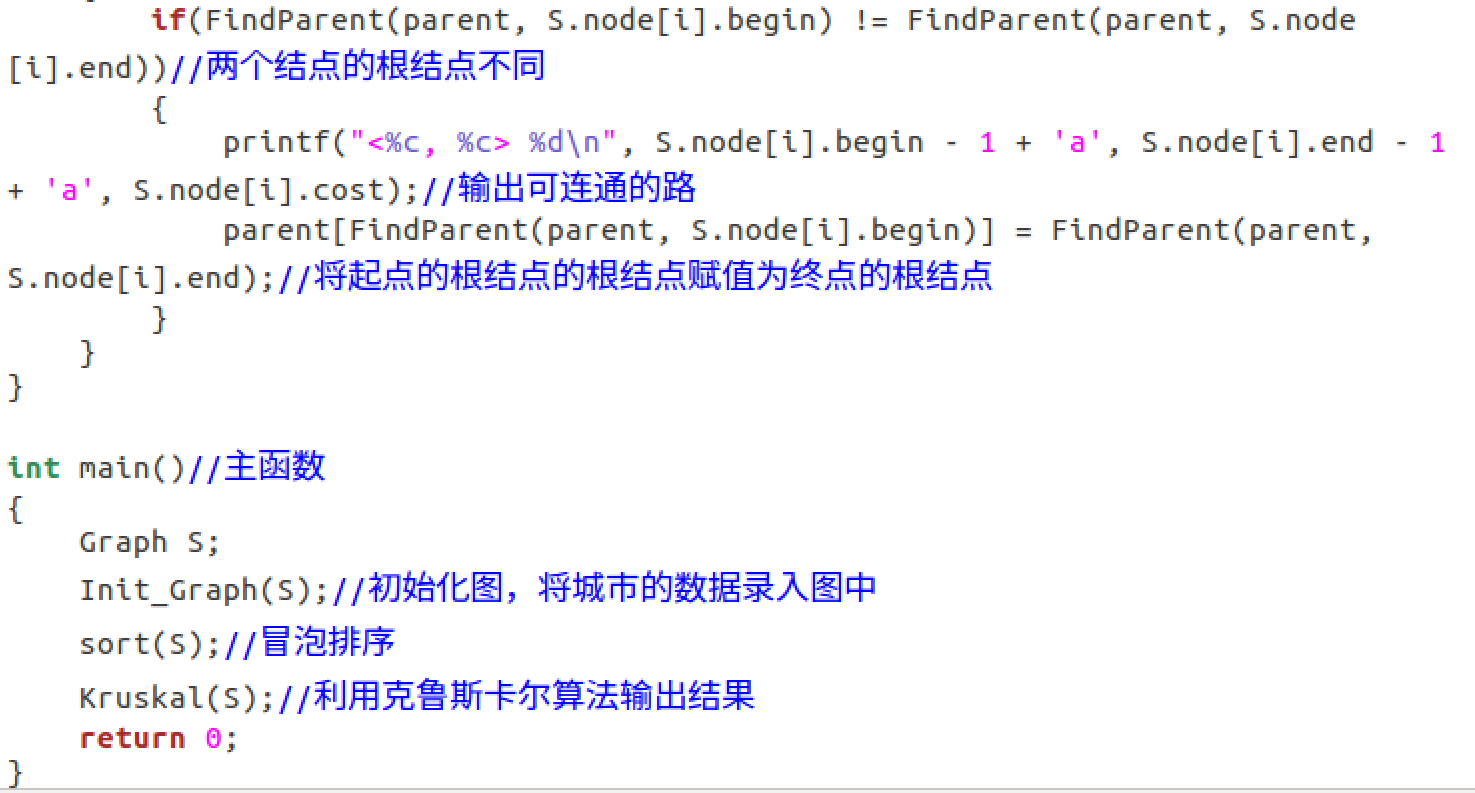
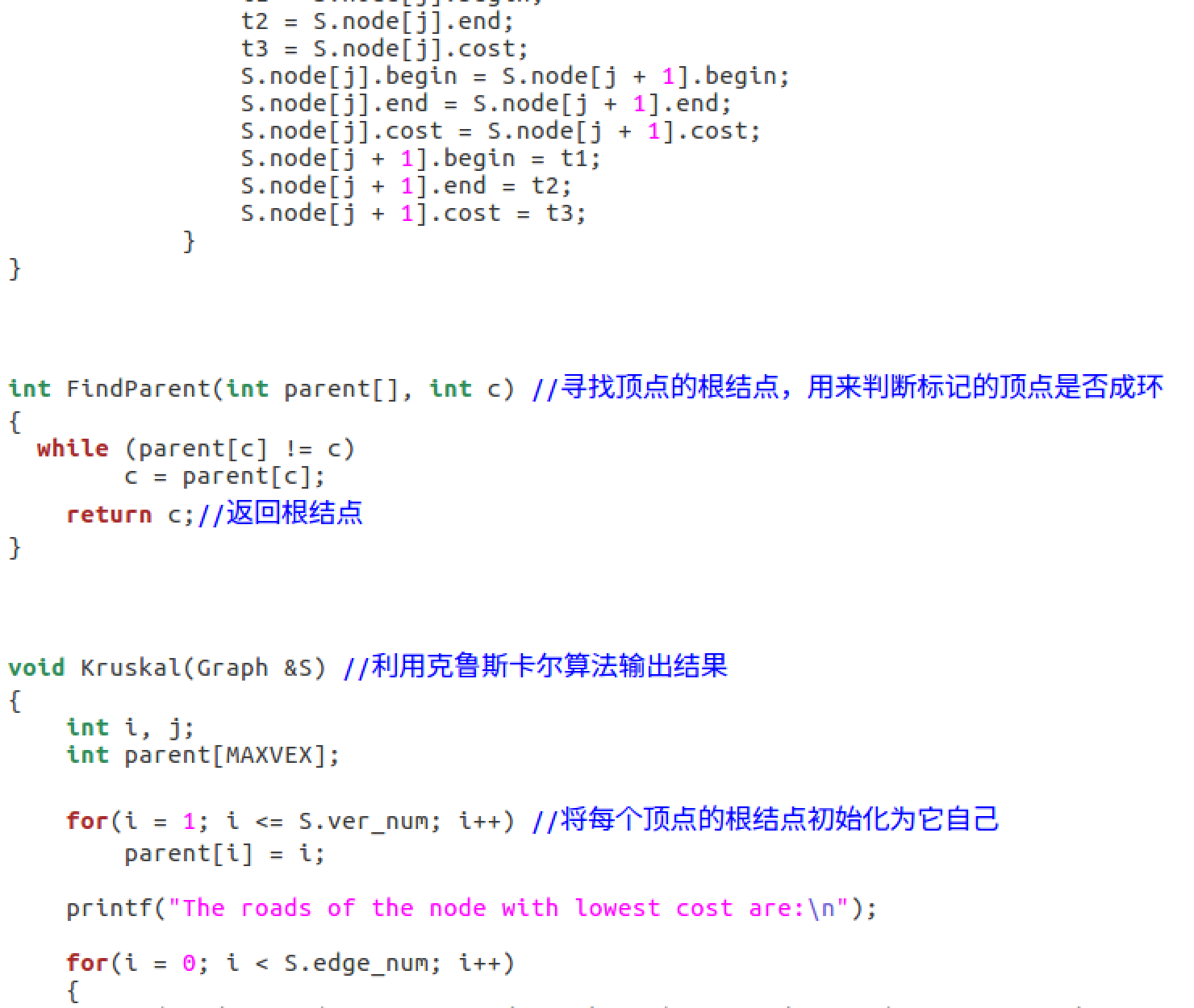
主程序

带权无向图模块

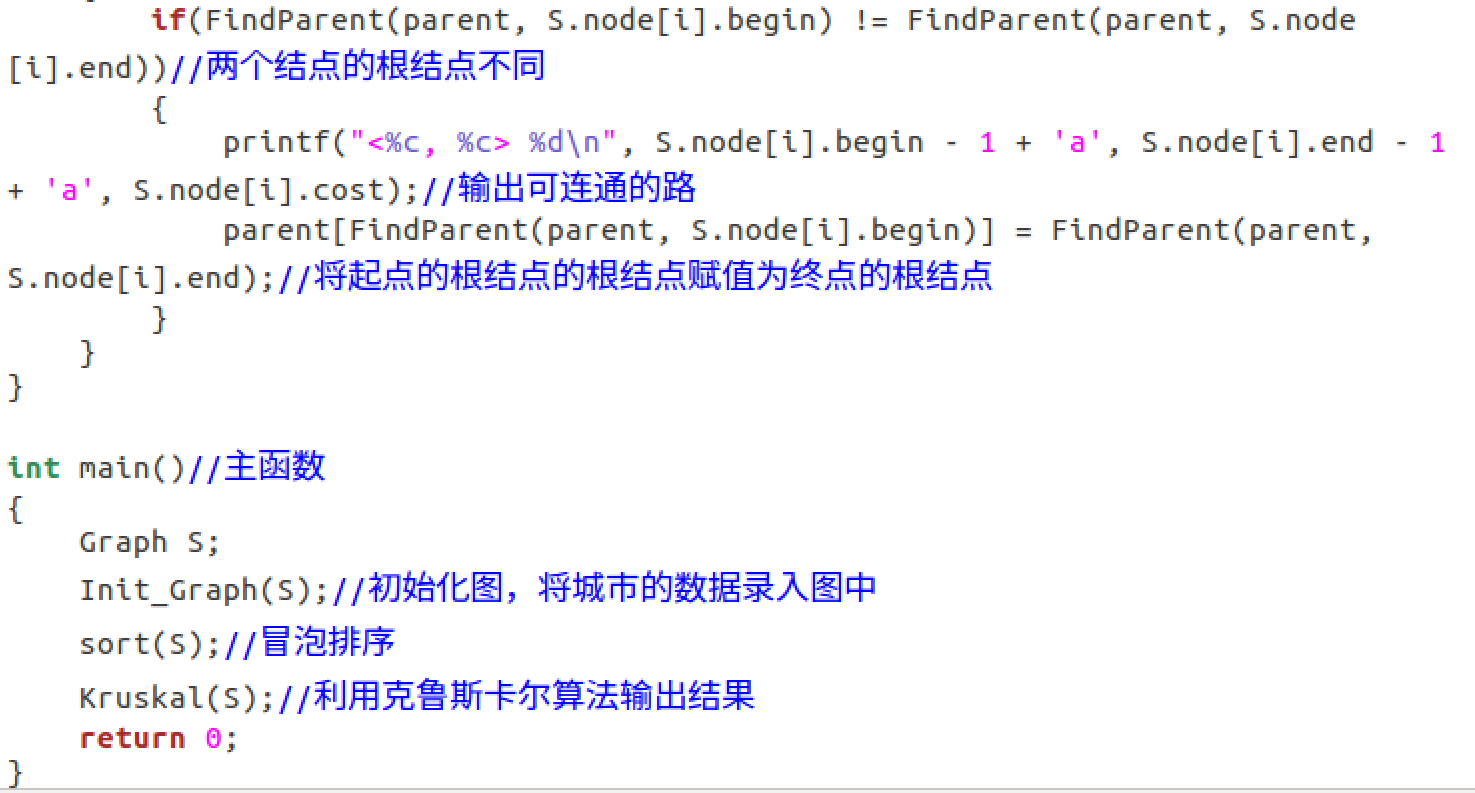
1. **详细设计**
2. 根据题目要求和克鲁斯卡尔算法，克鲁斯卡尔算法使用并查集（Union-Find）算法。





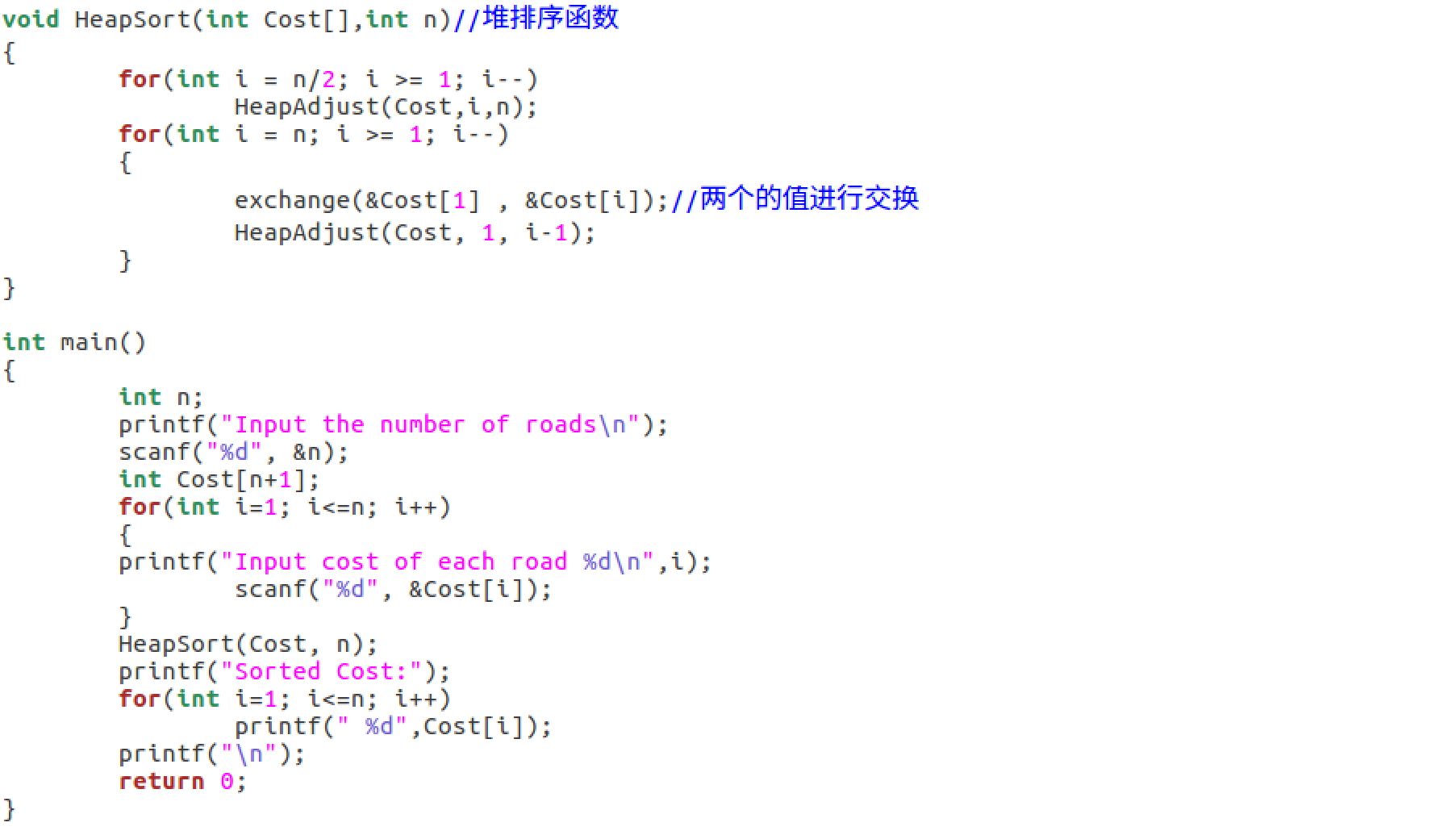


1. **主程序**

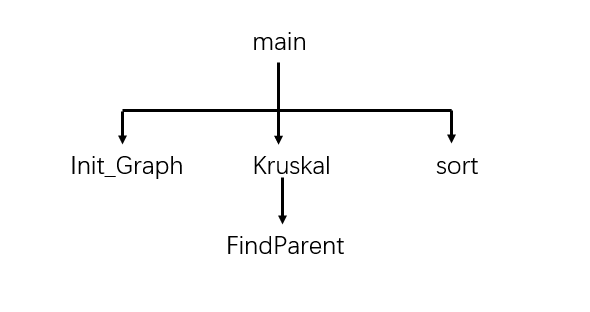


1. **选做内容—堆排序**

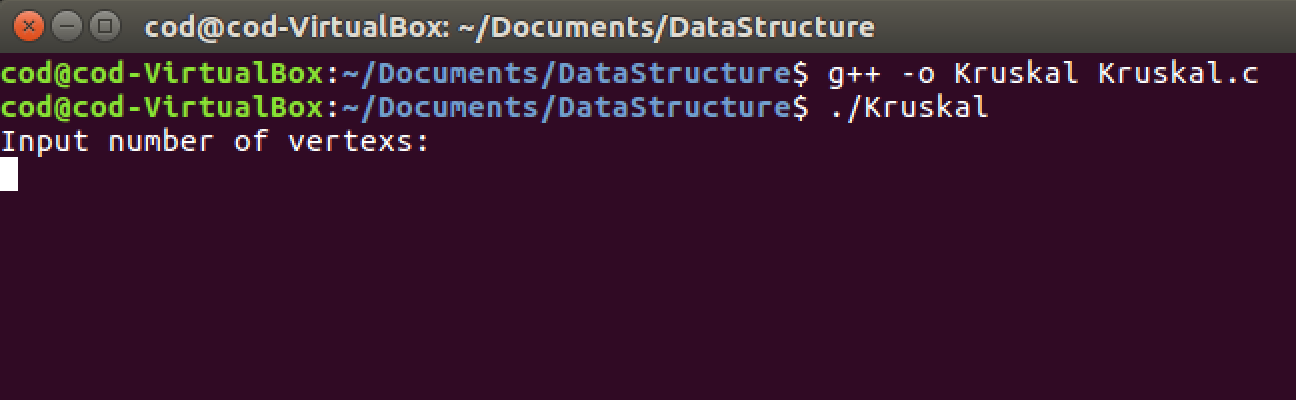




1. **函数的调用关系图**



1. **设计和调试分析**
2. 在Kruskal函数中，我们在parent的从属关系方面抉择良久，如何连接parent关系，parent关系的连接顺序是怎么样的，如何利用好这个关系等等。开始我们犯了一个致命的错误，即当两个结点的FindParent值不相等时，我们直接为这两个结点定义了parent的从属关系。这样做的结果是得不到一个稳定的parent关系，自然得不到正确的结果。
3. 也是在Kruskal函数中，一开始我们将每个结点的parent初始化为0，这样做的结果是每个结点的FindParent的函数值也相等了，此时程序相当于做了无用功。发现这个bug之后，我们将每个结点的parent初始化为它自己，这样就解决了这个问题。
4. 我们在输入输出的方式进行了多次优化，一开始我们采用的输入方式是需要用户把结点的字母转化为字母在字母表中的序数输入，输出时也只能以字母序数代表字母，这样一来还需要进行一步手动转化，把字母序数对应成字母。经过优化，我们的输入和输出程序能自行实行字母和字母序数的转化，方便用户使用。
5. **用户手册**
6. 本程序的运行环境为Linux虚拟机环境，执行文件为Kruskal.c。
7. 进入虚拟机后，将程序编译，运行，进入如下界面。



1. 根据屏幕上方的提示，依次输入顶点数、边数、每条边的顶点及其权值。其中每条边的顶点及其权值是输入格式为（输入括号内的内容）：<vi vj n\t>， 其中vi和vj 为顶点名称（小写字母），i, j ≤ 最大边数，n为边(vi , vj)上的权值。
2. 输入完毕后，键入回车键，显示结果.
3. **测试结果**

运行文件，输入测试数据：

8

14

a b 4

a c 3

b c 5

b e 9

b d 5

c d 5

c h 5

e d 7

d h 4

d f 6

d g 5

e f 3

f g 2

g h 6

输出结果为：

The roads of the city with lowest cost are:

<f, g> 2

<a, c> 3

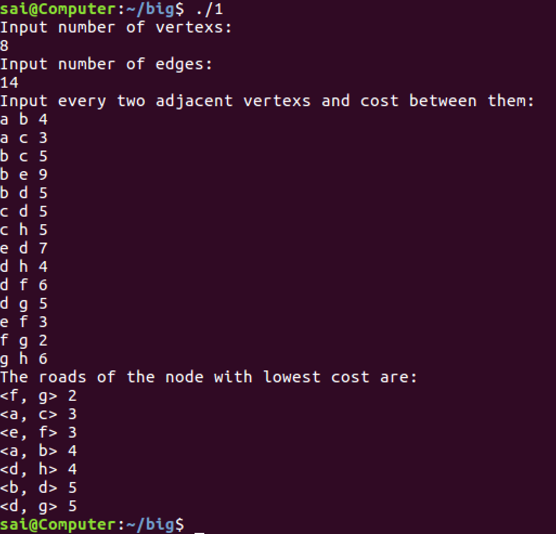
<e, f> 3

<a, b> 4

<d, h> 4

<b, d> 5

<d, g> 5



1. **附录**

源代码见附件“实验5.6.zip”，其中堆排序部分见“heapsort.c”。