

《数据结构》

课程设计报告

姓 名： 李昭熹 班号： 193162

学 号：20161002844 组长： 李昭熹

院（系）： 计算机学院 专业： 网络工程

指导教师： 郭 艳 职称： 副教授

2017 年 12 月

# 需求分析

## §1.1 引言

### 1.1.1 课程设计背景

在自选数据结构题目时，我和另一名同学选择了无向带权图的数据结构设计以及应用。既然课程是数据结构，那么在进行结构设计时，就务必要考虑到数据结构的效率，即实际运行时的时间复杂度与空间复杂度，在对我们的课题进行观察时发现，我们要设计的这个数据结构，是以空间来换取时间的，所以我们的主要任务就是极力减小时间复杂度，在保证程序快速运行的前提下，再尽我所能地减小空间的占用。

在确立我们数据结构设计的方向后，我们开始了对《数据结构课程设计要求》的研究，起初看到复杂的要求，我们都是一头雾水，能查到的资料也都是Java版本，但是经过对所给资料的研究，课程设计有了基础的框架。

### 1.1.2 成员组成及任务分配

成员：

|  |  |  |
| --- | --- | --- |
| 姓名 | 性别 | 班级 |
| 李昭熹 | 女 | 193162 |
| 周锦浩 | 男 | 193161 |

图表 一‑1

1.Graphlnk

|  |  |  |
| --- | --- | --- |
| 函数 | 函数功能 | 负责人 |
| Initiate(int sz) | 新建一个无边无顶点的图 | 周锦浩 |
| int vertexCount() | 返回图中的顶点数 | 李昭熹 |
| int edgeCount（） | 返回图中的边数 | 李昭熹 |
| Void getVertices() | 返回包含所有顶点的数组 | 李昭熹 |
| void addVertex(Vertex \*x) | 添加顶点 | 周锦浩 |
| void removeVertex(string a) | 删除顶点 | 周锦浩 |
| int isVertex(string a) | 判断该顶点是否在图中 | 周锦辉 |
| int degree(string a) | 顶点的度 | 李昭熹 |
| string getFirstNeighbor(string a) | 返回第一个邻接顶点 | 李昭熹 |
| String getNextNeighbor(int v1,int v2) | 返回下一个邻接顶点 | 李昭熹 |
| void addEdge(Edge \*edge) | 添加边 | 李昭熹 |
| Void removeEdge(string v1,string v2) | 删除边 | 周锦浩 |
| int isEdge(string u,string v) | 判断是否为边 | 李昭熹 |
| int getweight(string u,string v) | 求某边的权值 | 李昭熹 |
| void dijstra(string a, string b, int \*path, int type) | Dijstra算法求路径 | 李昭熹 |
| int weightcost(string u, string v) | 最小花费路径 | 李昭熹 |
| int weightdist(string u, string v) | 最短距离路径 | 李昭熹 |
| VerNode\*FindPos(const int num) | 找到对象所对应位置 | 李昭熹 |

图表 一‑2

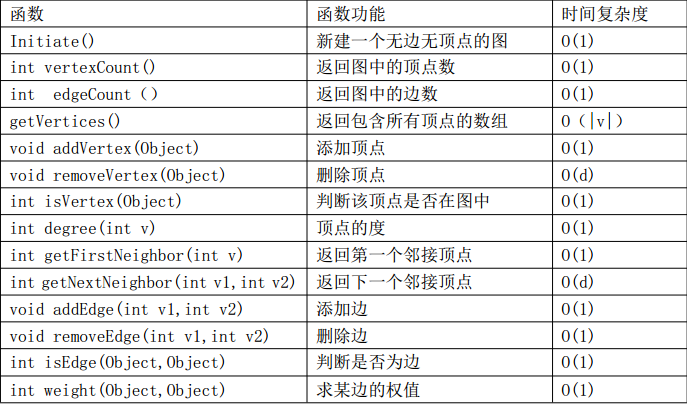
2.Hash

|  |  |  |
| --- | --- | --- |
| 函数 | 函数功能 | 负责人 |
| bool Remove(string a) | 删除hash表的一个元素 | 周锦浩 |
| bool Search(string a); | 寻找哈希表中的元素 | 周锦浩 |
| bool Insert(Vertex\* vertex); | 向hash表中插入一个元素 | 周锦浩 |

图表 一‑3

## §1.2 具体分析

### 1.2.1 数据结构部分需求分析



图表 一‑4

以下是一些具体的思想来实现特定操作来达到上述目的（可参考图 一-4） ：  
（1） 对于图中顶点的存储， 最好的方式是采用哈希结构来存储。 哈希存储可以使得  
isVertex()的时间复杂度为 O(1)。

*哈希表结构选择使用二维哈希。*  
（2） 为使 getVertices()时间复杂度为 O(|V|)， 则应设计链表来存储顶点序列； 为使  
removeVertex()时间复杂度为 O(d)， 则该链表应该是双向的；

*getVertices()是返回包含所有顶点的数组，时间复杂度应当是遍历一遍顶点，所以是链表；而删除函数，为了使得删除时直接可以删除指定节点，所以选择双向链表，使得待删除结点的前一个结点可以和后一个结点直接相连，不需要另外再查找；犹豫过是否要使用双向循环链表，而后发现使用双向链表与头指针和尾指针就可以完美解决问题。*（3） 为使得 getNeighbors()时间复杂度为 O(d)， 需要对每个顶点创建邻接表来存放边；  
为使 removeEdge()是 O(1)， 则此表应该是双向的。

*设计时应当使用双向邻接链表。*  
（4） 在无向图中， 边（u， v） 必须出现在两个链表中（u 顶点和 v 顶点的边邻接表） 。  
若删除 u， 则必须删除 u 的所有相关联的边， 即使包含在 v 的链表中。 为使 removeVertex()时间复杂度为 O(d)， 不能遍历所有的邻接表， 可以尝试用下列方法来得到 O(d） 的时间。既然(u,v)在图中出现于两个表中， 则可以用两个结点来表示（u， v） ， 分别在 u 表和 v表中。 这些结点每一个可称为“半条边” ， 其中一条边为另一条的“伙伴” 。 每一个半条边有向前和向后的指针链入邻接链表中， 此外该半条边和其伙伴由“伙伴指针” 相关联。 采用这种方式， 我们可以从图中删除 u 时， 可遍历 u 的邻接链表， 并使用伙伴指针来删除另外半条边只使用 O(1)的时间。

*根据提示应当设计伙伴指针，在半边邻接链表中再加一个存储另一个半边的伙伴指针的空间。*  
（5） 为使 removeEdge()、 isEdge()、 weight()时间复杂度为 O(1)， 你需要第二个哈希  
表来存储边， 这个哈希表是成对出现的顶点在图中的表现形式。 为使 removeVertex()时间  
复杂度为 O(d)， 你需要从哈希表和顶点的邻接表上删除边。

*删除结点需要删除指定的半边和结点信息，涉及到哈希表的删除以及邻接表的删除，结点不同的位置会对结点删除的函数逻辑有较大的影响，需要分多钟情况讨论。*  
（6） 为使 vertexCount(),edgeCount(),degree()时间复杂度为 O(1)， 你需要添加边和  
顶点的计数器、 每个顶点的度数， 并且在每个操作注意其动态更新。

*只需要在结点数据结构中添加一个度数信息，在添加和删除边的时候对度数信息进行修改即可。*

图片包含 文字, 地图

已生成极高可信度的说明

图表 一‑5

### 1.2.2 应用部分需求分析

设想题目： 旅行导航

随着科技不断的进步，卫星已经遍布全球。人们可以在任何时间任何地点拿出手机或是打开任何电子设备，输入目标，即可显示相关路径。在众多路径中，又可根据自身条件，选择诸如：最短路径，即不计费用等，只根据距离长短作判断依据；最少花费，即根据行程所需费用，制定花费最少的路线。

在此之外，若用户有意向自己制定一份旅游计划，对于路径的选择，此系统便是必不可少的了。

只需要输入起点和终点，即可显示出所有的路线安排，用户可以根据具体需要增加或者删减其中的任意景点，而系统会根据用户的具体需求作出相应的调整，真正做到“有求必应”。

系统同时囊括世界各国地图，无论是境内游或是境外游，都能包君满意。

从智能制定旅游计划，囊括各大景点，结合路程远近，制定最优计划。当然，优秀的系统必须是人性化的。在旅行途中，或者最初之时，任何时间任何地点只要用户有需要，系统都能配合。

增删查改，随时更新，随时调整。一键知晓当前景点库中所有景点，输入一个景点系统就会显示周边所有景点，同一目标，多条路径供用户选择，等同于用户在定制一个专属于自己的旅游导航系统。

分析：

首先，如果真的能够实现上诉功能，就需要一个非常大的数据库和存储空间，但是这在现阶段是不可实现的，所以，我们将范围缩小，做了一个国内的城市旅游线路推荐，录入城市和城市之间的距离，以及路程花费（这里的花费其实可以对交通工具分为很多类别，但是处于对结构的应用，就只用一个值为代替。

这里要求对最短路径，与最小花费做两个路径推荐，实际上可以运用迪杰斯特拉（djstra）算法，分别将路程长度和路程花费当作关键值进行替换即可。

增删改查方面，增需要划分为增加城市，增加路径，删除也要分为删除城市，删除路径等…根据需求，可能还需要有管理员系统负责信息维护与修改，和用户系统负责推荐和投诉等，可以依照最终情况，选择性实现。

# 程序设计

## §2.1 数据结构部分设计

### 2.1.1 类设计

1.邻接表结点类

class Vertex

{

public:

int key=0; //关键值

int code=0; //点的实际值即编号

string name=""; //点的name

int degree=0; //点的度数

int number; //点的序号

Edge \*adj=new Edge();

Vertex \*rLink=nullptr, \*lLink=nullptr;

};

邻接表是双向链表，所以需要有l结点和r结点两个结点。



图表 二‑1

！后期又添加了序号 number 结点

2.邻接表边结点类

class Edge

{

public:

int dest=0; //边的另一个顶点的真实code

int cost=0; //边上的权值

int dist = 0; //边的长度

int key=0; //边的关键码

int head=0; //存终点

string d\_name="";

string h\_name = "";

Edge \*lLink=nullptr, \*rLink=nullptr; //下一条边链指针

Edge \* partner=nullptr;//伙伴指针

Edge() { cost = maxWeight; }

Edge(int num, int weight ,string name) : dest(num), cost(weight), lLink(nullptr), rLink(nullptr),d\_name(name) {};

bool operator != (Edge& R)const

{

return (dest != R.dest) ? true : false;

}

};

边结点存储的信息较多，包括边的两端点的结点信息以及伙伴边指针信息，还包括边本身的长度以及花费等。大致结构同Vertex结点。

3. 哈希表结点类

struct VerNode//点的哈希表结点

{

Vertex\* data= nullptr;

VerNode \*link=nullptr;

VerNode\* last=nullptr;//指向前一个结点；

};

由于我们选择开散列，所以需要前指针与后指针方便删除与连接同key值结点。

4.哈希表边结点类

struct EdgNode//边的哈希表结点

{

Edge\* data= nullptr;

//int head;//边的起点 //在边结点里增加量

EdgNode \*link=nullptr;

EdgNode \*last=nullptr;

};

5.带权无向图类

class Graphlnk

{

public:

Graphlnk(int sz);

~Graphlnk() {};

void Initial();

int vertexCount(); //返回图中的顶点数

int edgeCount(); //返回图中的边数

void getVertices(); //返回包含所有顶点的数组

void addVertex(Vertex \*x); //添加顶点

void removeVertex(string a); //删除顶点

int isVertex(string a); //判断该顶点是否在图中

int degree(string a); //顶点的度

string getFirstNeighbor(string a); //返回第一个邻接顶点

string getNextNeighbor(string v1, string v2); //返回下一个邻接顶点

void addEdge(Edge\* edge); //添加边

void removeEdge(string v1, string v2); //删除边

int isEdge(string u, string v); //判断是否为边

int weightcost(string u, string v); //求某边的权值

int weightdist(string u, string v); //获取距离

int getWeight(int a, int b, int type); //获取距离值或花费值

void dijstra(string a, string b, int \*path, int type);//迪杰斯特拉算法

private:

Verhash NodeHashtable{ 1000, MaxVertices };//点哈希表

Edghash EdgeHashtable{ 1000,MaxVertices }; //边哈希表

//VertexDblList NodeTable;

Vertex \*real;

Vertex \*first;

void print(int\* dist, int\* path, int u, int v,int type); //输出最短路径

string getvalue(int v);

int maxVertices;

int numEdges;

int numVertices;

};

## §2.2 核心算法设计

### 2.2.1 点结点添加与删除



图表 二‑2

添加函数算法设计：

如图表二-2首先录入待添加的城市名称和编号构成边结点a，先尝试在哈希表中添加结点，如果添加成功，再在邻接表中添加结点，将新结点加到邻接链表的尾部，再将尾指针指向新结点即完成插入操作；哈希表中的插入见图表 二-2，首先调用Find函数查找在记录中是否已经存在，若已经存在，则返回空指针，hash表的addhashnode函数检测到返回的nullptr输出城市已存在的提示信息，同时向addvertex函数返回false，addvertex函数检测到false则不进行邻接表节点添加操作，而直接返回。

删除函数算法设计：

如图，录入要删除的点，调用Find函数查找，查找成功后在hash表中删除结点的边，及结点，若删除成功，从邻接链表中将其删除，将邻接链表表尾元素拷贝到删除后空位，将原表尾置空，结点数减1，否则输出该点不存在，返回主菜单；hash表中的删除先是由城市名称，计算hash表中的key值，之后访问hash结点边结点以及其伙伴指针将之删除。



图表 二‑3

### 2.2.2 边结点添加与删除

添加函数算法设计：

如图二-4，首先录入边的信息，尝试在边的邻接链表中添加此边结点，具体方式为计算出此边的hash key值，访问hash表，看是否能够查找到该结点，若不能查找到该结点，返回hash对应桶的尾指针，将新边信息插入，返回true；如果能够查找到边信息，则输出边已存在的提示语；返回true 后，在邻接链表中对应城市结点插入这一对边，并用兄弟指针互指。



图表 二‑4

删除函数算法设计：

如图 二-5，首先录入边信息，在边hash表中查询边是否存在，若存在，则删除边信息，返回true，接着在邻接表中查找对应的某个城市，并且找到其对应的边进行删除，且通过其兄弟（伙伴）指针找到对应的另一个半边，完成快速删除；若未查询到相关信息，则会输出某城市不存在的提示样式并返回主菜单。



图表 二‑5

### 2.2.3 最短路径算法设计



图表 二‑6

迪杰斯特拉算法设计：

如图表二-2，这个算法是通过为每个顶点 v 保留目前为止所找到的从u到v的最短路径来工作的。初始时，原点 u 的路径权重被赋为 0 （d[u] = 0）。若对于顶点 u 存在能直接到达的边（u, m），则把d[m]设为w（u, m）,同时把所有其他（s不能直接到达的）顶点的路径长度设为无穷大，即表示我们不知道任何通向这些顶点的路径（对于所有顶点的集合 V 中的任意顶点 s， 若 s 不为 u 和上述 m 之一， d[v] = ∞）。当算法结束时，d[s] 中存储的便是从 u 到 s 的最短路径，或者如果路径不存在的话是无穷大。

边的拓展是Dijkstra 算法的基础操作：如果存在一条从 m 到 v 的边，那么从 u 到 v 的最短路径可以通过将边（m, v）添加到尾部来拓展一条从 u 到 v 的路径。这条路径的长度是 d[m] + w(m, v)。如果这个值比目前已知的 d[v] 的值要小，我们可以用新值来替代当前 d[v] 中的值。拓展边的操作一直运行到所有的 d[v] 都代表从 u 到 v 的最短路径的长度值。此算法的组织令 d[u] 达到其最终值时，每条边（u, v）都只被拓展一次。

算法维护两个顶点集合 S 和 Q。集合 S 保留所有已知最小 d[v] 值的顶点 v ，而集合 Q 则保留其他所有顶点。集合S初始状态为空，而后每一步都有一个顶点从 Q 移动到 S。这个被选择的顶点是 Q 中拥有最小的 d[u] 值的顶点。当一个顶点 u 从 Q 中转移到了 S 中，算法对 u 的每条外接边 (u, v) 进行拓展。

通过上述算法就可以算出两城市之间最短距离。



图表 二‑7

最短路径输出函数设计：

如图表二-3，最短路径的输出函数依赖于path数组，dist数组与d数组，path数组用于记录每一点的上一点，直到到达起始城市，dist数组用于存储距离/花费，d数据则用于记录途经城市的序号，当用while循环统计完路径，再由d数组调用getvalue函数输出点的名称，完成路径输出；最后再输出dist数组对应的值。

# 程序实现

## §3.1 调试分析

### 3.1.1 调试问题

1.初始化问题：在读文件初始化时，没有掌握文件结束末尾指针的正确使用方法，在查阅部分资料后，能够正确初始化实例；

2.hash添加函数指针逻辑混乱：起初结点总是添加失败，后期删除也是，最后单步调试得出的原因是hash函数在编写初期有一种情况被漏掉，另有一种冗余情况，经过修正后能够正常添加。

3.find（）函数返回值有误：这个函数是基于findpos函数的，在结点存在的情况下，函数可以正常返回，但若当结点不存在是，就会造成访问冲突，后经过分类讨论，解决了这一问题。

4.细节优化：初期代码并没有很多提示语句，给使用者很不好的使用感受，后期在代码初步完成后在一些关键步骤添加了错误提示，以优化程序软性能。

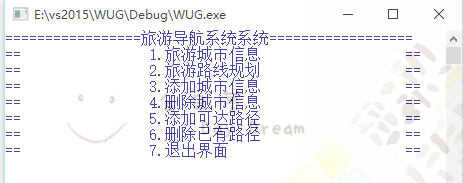
### 3.1.2 函数调用关系

### 3.3.3用户手册

1. 本程序运行在windows系列的操作系统下，执行文件为WUG.exe。
2. 请用户在使用前正确了解城市编号以及汉字写法，以免输入问题影响结果正确性。
3. 用户按照程序框提示输入，根据菜单后方或前方的数字进行功能选择。
4. 如有程序异常，请及时回馈。

## §3.2 多组测试结果

程序界面菜单：



图表 三‑1

1.旅游城市信息

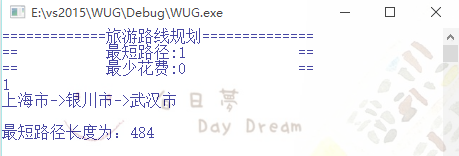


图表 三‑2

2.旅游路线规划

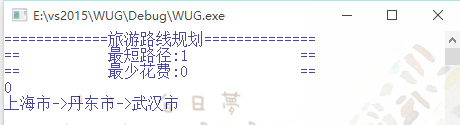
查找成功：

最短路径：



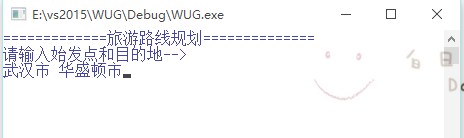
图表 三‑3

最少花费

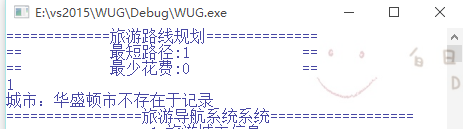


图表 三‑4

查找失败



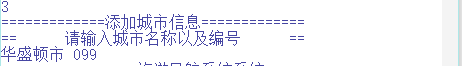
图表 三‑5



图表 三‑6

3.添加城市

添加成功：

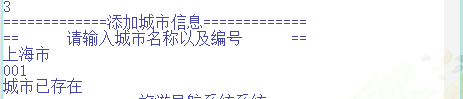


图表 三‑7



图表 三‑8

添加失败：



图表 三‑9

4.删除城市

删除成功：

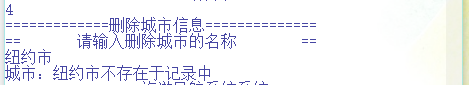


图表 三‑10



图表 三‑11

删除失败：



图表 三‑12

5.添加可达路径

添加失败：



图表 三‑13

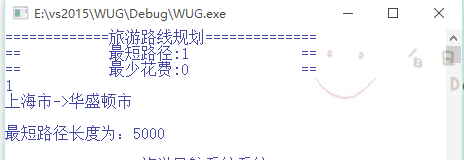


图表 三‑14

添加成功：



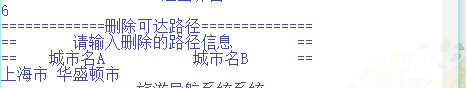
图表 三‑15



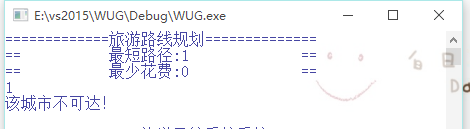
图表 三‑16

6.删除已有路径

删除成功：



图表 三‑17



图表 三‑18

# 心得体会

这一次的上机实习比以往更加具有挑战性，因为我们做的是一个没有参考的数据结构，我和小伙伴在第一眼看到它的时候也觉得很难实现，一张被画得密密麻麻的结构图成了挡在我们前面的拦路虎，但是我们没有气馁，而是一点一点去体会，去揣摩，去思考，到底怎样才能实现这样的结构？

功夫不负有心人，经过一周的讨论，我们最终破解了结构的谜题（虽然最终效率没有达到预想值，但也达到了百分之八十），并且做出了最初的结构设计，但是最初就是最初，虽然有了想法，但是逻辑还很不完整，有很多矛盾的地方，我和小伙伴就夜以继日得调试bug，尽自己所能得解决bug，这才有了现在的成品，一个虽然不很优秀，但是倾注了我们很多心血的成品。

这次实验我们采取了合作的方式，“人多力量大”是实话，每个人的分工明确后，个人的任务也比较容易完成，但是“一个和尚挑水喝，两个和尚抬水喝，三个和尚没水喝”也是不假。在做项目的这段日子里，我们很清晰的认识到时间的重要性，以及效率的重要性。

本次实习的主要考察部分是邻接表与哈希表的活用，散列表（Hash table，也叫哈希表），是根据键（Key）而直接访问在内存存储位置的数据结构。 也就是说，它通过计算一个关于键值的函数，将所需查询的数据映射到表中一个位置来访问记录，这加快了查找速度。邻接链表则是对图的每个顶点建立一个单链表，存储该顶点所有邻接顶点及其相关信息。每一个单链表设一个表头结点。这样的复合结构考验和巩固了我们数据结构的相关知识，我们收益颇丰。

总之，通过这次实习，我学到了很多，感谢老师的指导和助教学长的辛勤工作，你们辛苦了。