

**课 程 实 验 报 告**

**课程名称： 数据结构实验**

实验名称： 基于链表的树、图实现

**专业班级：**

**学 号：**

**姓 名：**

**指导教师：**

**报告日期： 2022年 月 日**

**网络空间安全学院**

**目 录**

**[1 基于链表的建图 1](#_Toc118025690)**

[1.1 需求分析 1](#_Toc118025691)

[1.2 总体设计 1](#_Toc118025692)

[1.3 数据结构 2](#_Toc118025693)

[1.4 算法设计 3](#_Toc118025694)

[1.5 系统实现 5](#_Toc118025695)

[1.6 系统测试 5](#_Toc118025696)

[1.7 复杂度分析 6](#_Toc118025697)

[1.8 结果分析 6](#_Toc118025698)

[1.9 实验小结 6](#_Toc118025699)

**[2 增加站点，删除 7](#_Toc118025700)**

[2.1 需求分析 7](#_Toc118025701)

[2.2 总体设计 7](#_Toc118025703)

[2.3 数据结构 8](#_Toc118025704)

[2.4 算法设计 9](#_Toc118025705)

[2.5 系统实现 12](#_Toc118025706)

[2.6 系统测试 12](#_Toc118025707)

[2.7 复杂度分析 13](#_Toc118025708)

[2.8 结果分析 13](#_Toc118025709)

[2.9实验小结 13](#_Toc118025710)

**[3 从指定站点出发，计算出到另一个站点的最短距离和途径的地铁站序列 14](#_Toc118025711)**

[3.1 需求分析 14](#_Toc118025712)

[3.2 总体设计 14](#_Toc118025713)

[3.3 数据结构 15](#_Toc118025714)

[3.4 算法设计 16](#_Toc118025715)

[3.5 系统实现 18](#_Toc118025716)

[3.6 系统测试 18](#_Toc118025717)

[3.7 复杂度分析 19](#_Toc118025718)

[3.8 结果分析 19](#_Toc118025719)

[3.9 实验小结 19](#_Toc118025720)

**[参考文献 20](#_Toc118025731)**

**[附录 基于链式存储结构线性表实现的源程序 21](#_Toc118025733)****1**

# 1 基于链表的建图

## 1.1 需求分析

### 1.1.1 功能需求

使用邻接表构成有向图来表达地铁线路，存储武汉地铁1号线、2号线、6号线和7号线在前两站的站点信息。其中，地铁线路均为双向线路，相同站名的地铁站为转乘车站；

### 1.1.2 输入输出需求

使用邻接表构成有向图来表达地铁线路，存储武汉地铁1号线、2号线、6号线和7号线在汉口区域的站点信息。

输入形式：

总线路条数n

线路号1 站名1 到下一站的距离 站名2 …… 到下一站的距离 站名n 0(到下一站距离为0，代表该站是线路最后一站)

线路号n 站名1 到下一站的距离 站名2 …… 到下一站的距离 站名n 0(到下一站距离为0，代表该站是线路最后一站)

输出形式：

线路号 站名1 到下一站的距离 站名2 …… 到下一站的距离 站名n

## 1.2 总体设计

整个程序分为使用邻接表建立有向图、向有向图中添加边和节点以及遍历有向图三个部分。

## 1.3 数据结构

使用双向链表储存有向图中与顶点相邻的节点，即有向图中的边节点

1. **typedef** **struct** ArcNode  // 双向链表链接的边节点
2. {
3. **int** adjvex;  // 相邻节点在数组中的位置下标
4. **float** distance;  // 相邻节点与顶点的距离
5. **int** lineId;  // 所处的线路
6. **struct** ArcNode \*prior;  // 前一个与顶点相邻的边节点
7. **struct** ArcNode \*next;   // 后一个与顶点相邻的边节点
8. } ArcNode;

使用结构体储存顶点节点站点名和邻接边节点

1. **typedef** **struct** VNode    // 顶点节点
2. {
3. **char** name[20];  // 节点的站名
4. ArcNode \*firstArc;  // 第一个相邻的边节点
5. } VNode;

使用邻接表的方式存储有向图，vertices是存储顶点节点的数组，heads中存储地铁线路起始站点

1. **typedef** **struct** ALGraph  // 邻接表储存的有向图
2. {
3. VNode \*vertices[MAX\_VEX\_NUM];    // 顶点节点
4. **int** heads[MAX\_VEX\_NUM]; // 起始节点下标
5. **int** vexnum; // 顶点节点的个数
6. **int** headnum;    // 起始节点的数量
7. } ALGraph;

## 1.4 算法设计

添加顶点和边的系统如图1-1所示。

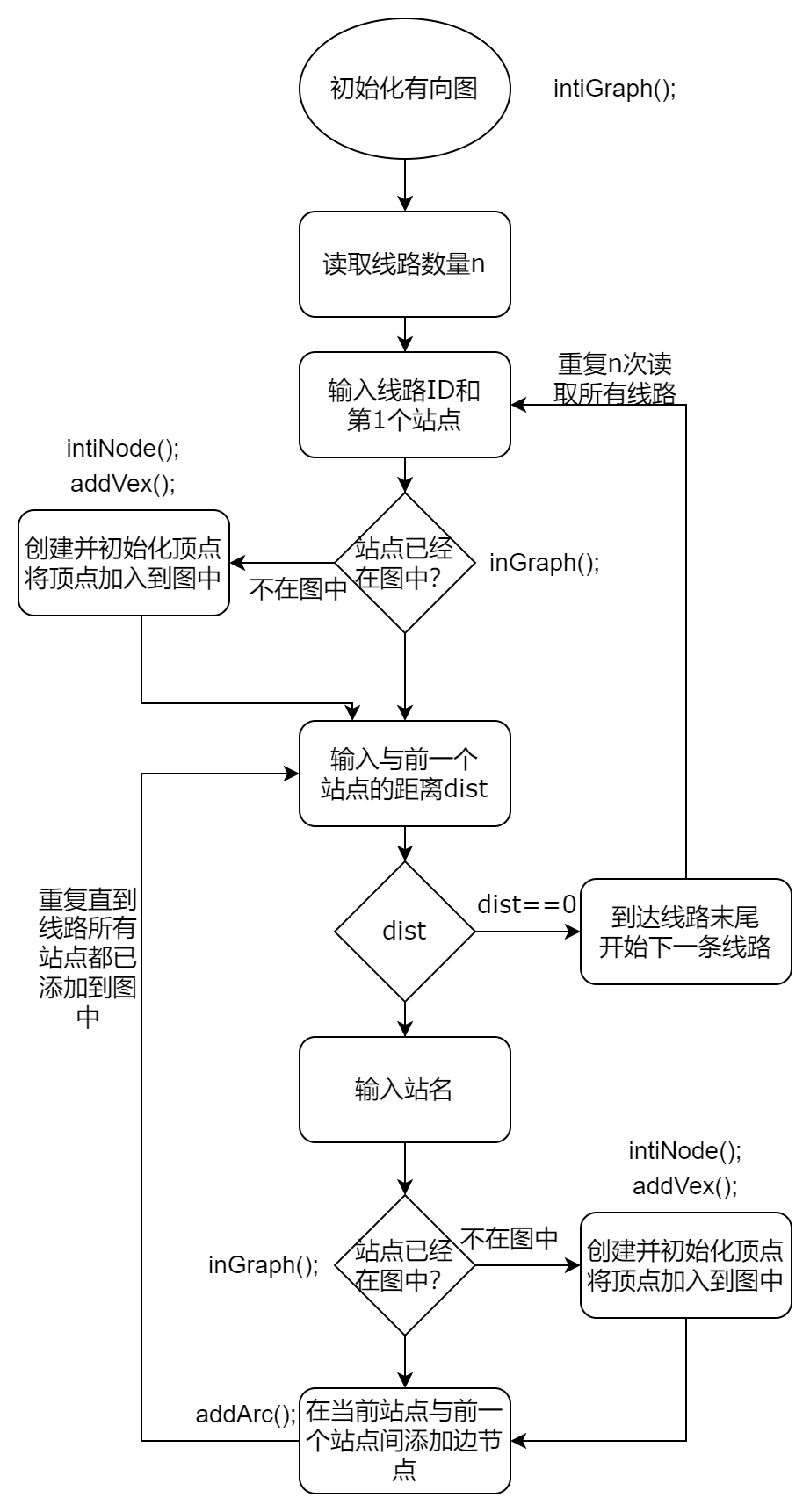


图1-1 添加顶点和边节点

遍历和输出有向图的系统如图1-2所示。

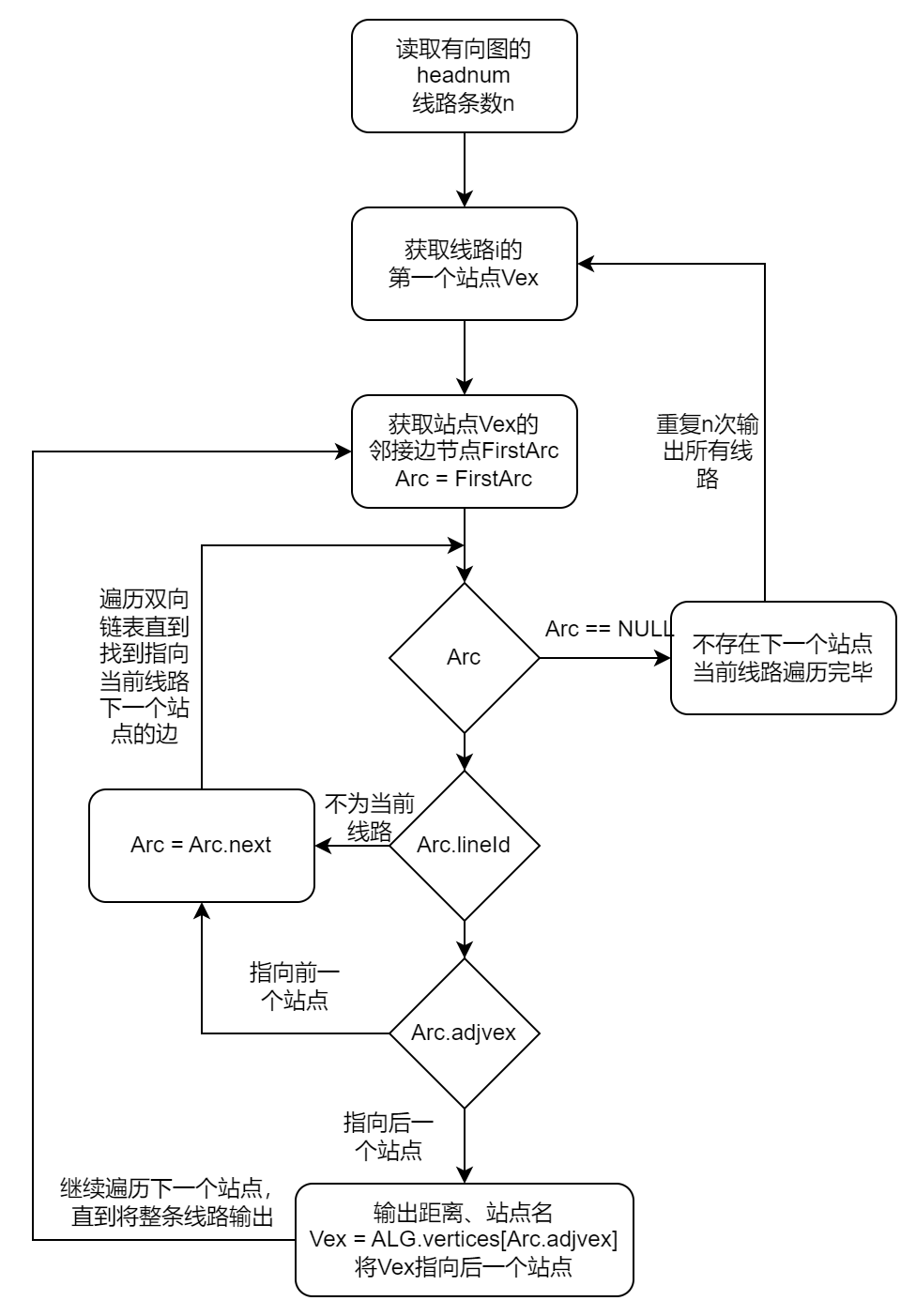


图1-2 遍历输出有向图

## 1.5 系统实现

本程序全程在Ubuntu 22.04上编写，使用GCC 11.2.0编译、调试、运行，并最终在Educoder平台上运行通过。

主要函数以及功能如表1-1所示。

表1-1主要函数及功能

|  |  |
| --- | --- |
| 函数名 | 主要功能 |
| int main() | 实现有向图顶点的增加和有向图的输出 |
| void initGraph() | 完成有向图的初始化 |
| int inGraph() | 判断某一站点是否在图中 |
| void addVex() | 将新的顶点加入有向图中 |
| Void addArc() | 在有向图的两个站点间添加边节点 |

## 1.6 系统测试

支持Educoder平台的所有可见测试用例与隐藏测试用例，均通过，如图1-3所示。

图1-3 通过所有测试

## 1.7 复杂度分析

创建和初始化有向图只需要O(1)的时间；记录站点时，对于全部n个站点，需要对每一个站点进行初始化，还需要建立一个指向前一个站点的边节点以及一个指向后一个站点的边节点，因此记录地铁站线路的时间复杂度为O(n)；输出线路图时，需要对每一个站点的边进行遍历，最坏的情况是每次都在最后一个链表节点中找到下一个符合条件的顶点，遍历的次数与图中边节点的数量相同，为2n，即输出的时间复杂度为O(n).

存储有向图时，每一个站点需要建立一个顶点节点，每个顶点节点需要建立边节点储存邻接信息，站点数为n时，边节点数目为2n，故该程序的空间复杂度为O(n)。

## 1.8 结果分析

成功通过所有的给定测试用例，表明该邻接表储存有向图的方式满足实验要求，能够完成对地铁线路图的存储。

## 1.9 实验小结

在存储站点时，每一个顶点节点需要储存与其邻接的所有节点的信息，因此当后一个节点与前一个节点之间相邻接时，不仅要在后一个节点中建立边节点，还需要在前一个节点中建立边节点，最开始的程序中，忽略了更新前一个节点的边节点信息，导致后续两个实验中判断两个站点是否邻接时出现了问题。因此设计了addArc()函数，在两个站点中添加给定长度的边。

由于地铁线路图中不同的线路存在交叉的情况，即有换乘站点，最开始选择的方式为对每一条线路分别存储，没有考虑换乘站点所在线路情况，在遍历站点时出现了遍历到其它的线路上，导致输出结果错误，因此选择了在边节点上添加线路标记lineId，而不是对站点进行标记线路，在遍历时只需要确保边节点的lineId与当前遍历线路的lineId相同即可。

# 2 增加站点、删除

## 2.1 需求分析

### 2.1.1 功能需求

对存储的线路信息，进行站点的增加、删除。

### 2.1.2 输入输出需求

使用邻接表构成有向图来表达地铁线路，进行站点的增加、删除。

输入格式：add（选择进行增加操作） 线路号 要增加站点的距前一个站点的距离 要增加站点的距后一个站点的距离 要增加位置的前一个站点名称 站点名称 （若增加的站点是该线路上的第一个，则前一个距离为0，不需要输入增加位置的前一个站点名称；若增加的站点是该线路上的最后一个，则后一个距离为0）

delete（选择进行删除操作） 线路号 要删除的站点名称

输出格式：操作成功，线路号 站名1 到前一站的距离 站名2 …… 到前一站的距离 站名n/增加失败，已有同名站点/增加失败，没有与输入的增加位置前一站点同名的站点/删除失败，没有同名站点

## 2.2 总体设计

整个程序包括有向图的创建、站点的添加和遍历，以及在有向图上添加新的站点、删除特定站点几个部分，其中有向图的创建、站点的添加和遍历在实验1中以及完成，本实验只需要增加添加站点、删除站点两个功能即可，并完成对指定线路的完整遍历。

## 2.3 数据结构

使用双向链表储存有向图中与顶点相邻的节点，即有向图中的边节点

1. **typedef** **struct** ArcNode  // 双向链表链接的边节点
2. {
3. **int** adjvex;  // 相邻节点在数组中的位置下标
4. **float** distance;  // 相邻节点与顶点的距离
5. **int** lineId;  // 所处的线路
6. **struct** ArcNode \*prior;  // 前一个与顶点相邻的边节点
7. **struct** ArcNode \*next;   // 后一个与顶点相邻的边节点
8. } ArcNode;

使用结构体储存顶点节点站点名和邻接边节点

1. **typedef** **struct** VNode    // 顶点节点
2. {
3. **char** name[20];  // 节点的站名
4. ArcNode \*firstArc;  // 第一个相邻的边节点
5. } VNode;

使用邻接表的方式存储有向图，vertices是存储顶点节点的数组，heads中存储地铁线路起始站点

1. **typedef** **struct** ALGraph  // 邻接表储存的有向图
2. {
3. VNode \*vertices[MAX\_VEX\_NUM];    // 顶点节点
4. **int** heads[MAX\_VEX\_NUM]; // 起始节点下标
5. **int** vexnum; // 顶点节点的个数
6. **int** headnum;    // 起始节点的数量
7. } ALGraph;

## 2.4 算法设计

增加站点的程序设计如图3-1所示。

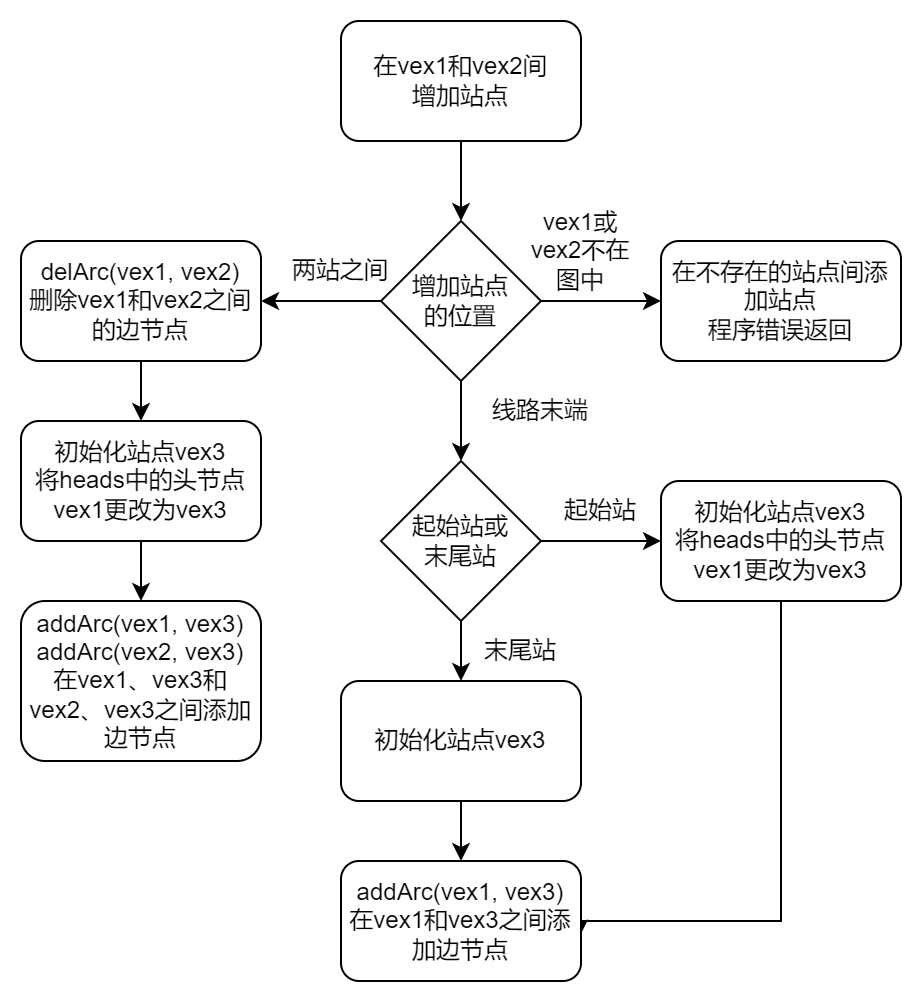


图2-1 增加站点部分

删除站点的程序设计如图3-2所示。

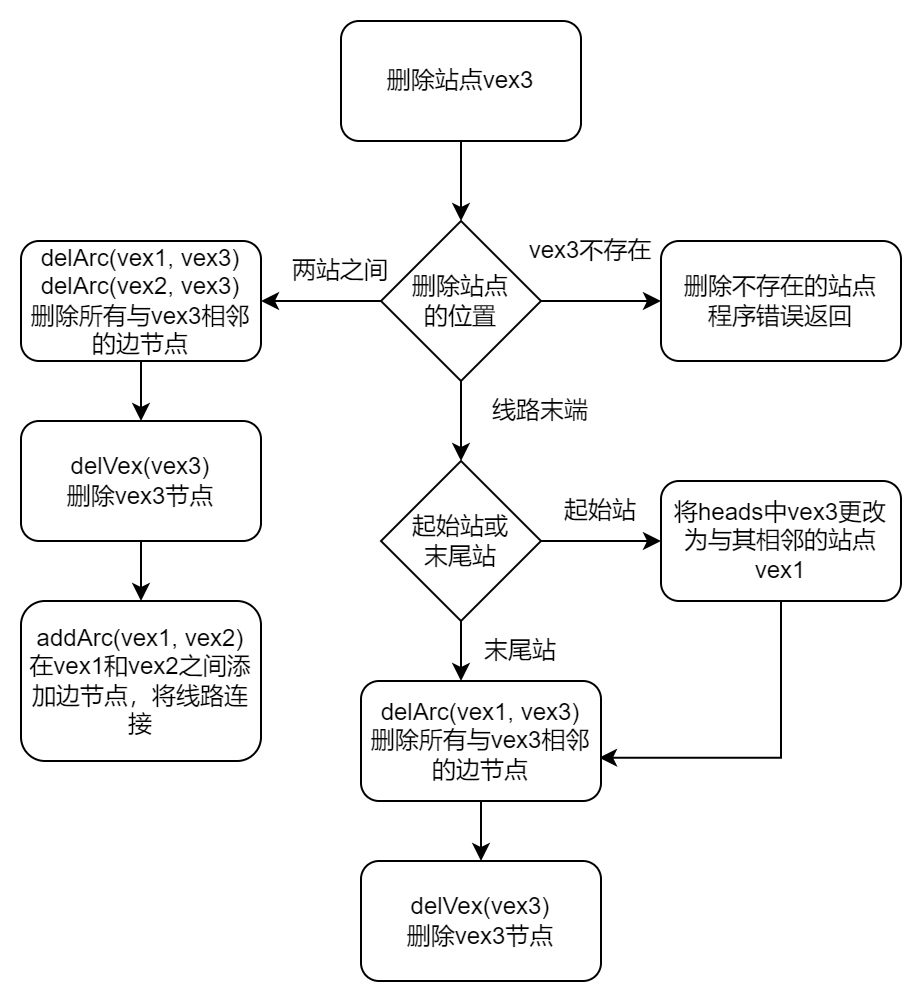


图2-2 删除站点部分

遍历输出指定线路程序如图3-3所示。

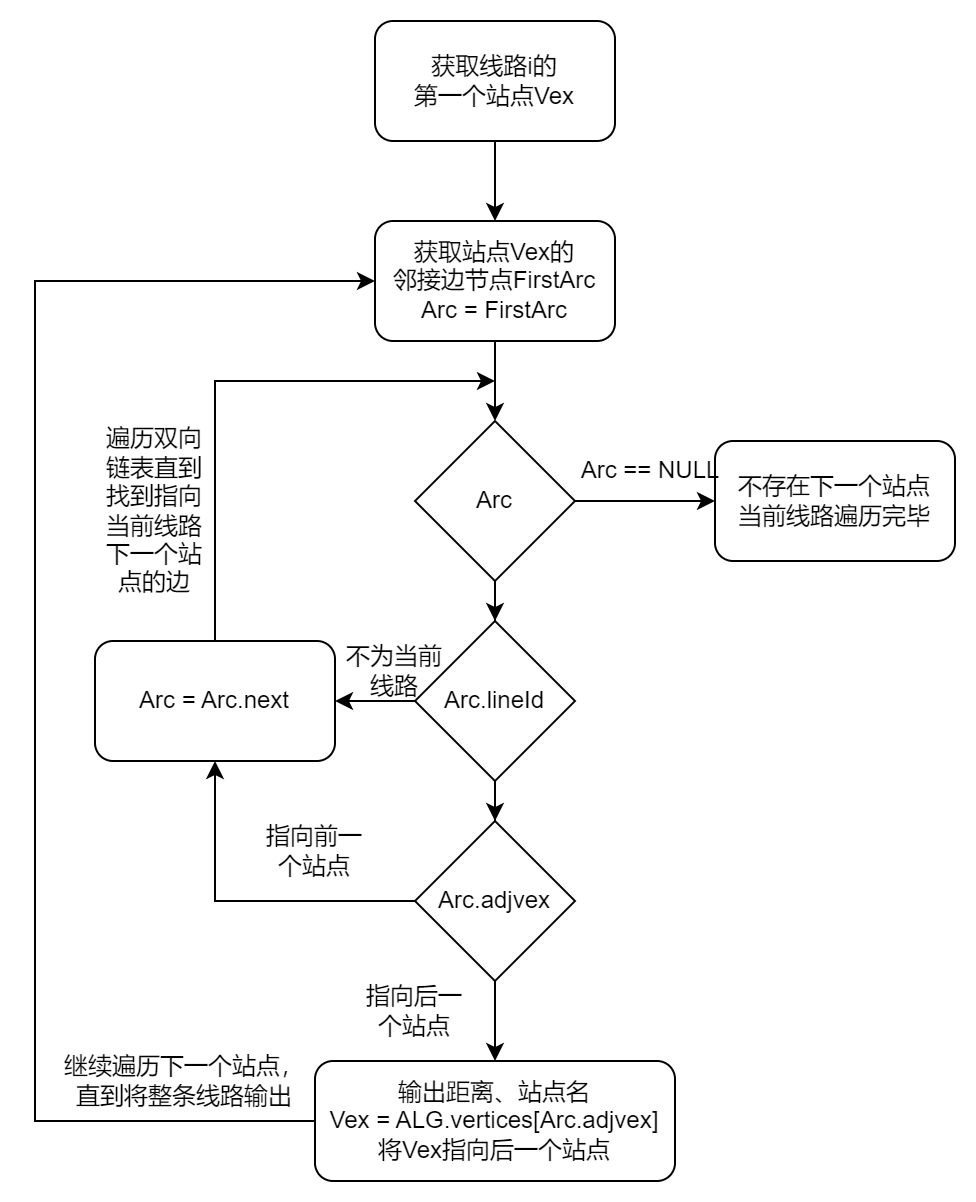


图2-3 输出线路部分

## 2.5 系统实现

本程序全程在Ubuntu 22.04上编写，使用GCC 11.2.0编译、调试、运行，并最终在Educoder平台上运行通过。

主要函数以及功能如表2-1所示。

表2-1 主要函数及功能

|  |  |
| --- | --- |
| 函数名 | 主要功能 |
| int main() | 实现有向图顶点的增加和有向图的输出 |
| void initGraph() | 完成有向图的初始化 |
| int inGraph() | 判断某一站点是否在图中 |
| void addVex() | 将新的顶点加入有向图中 |
| void addArc() | 在有向图的两个站点间添加边节点 |
| void printLine() | 输出线路信息 |
| void deleteVex() | 在线路上删除站点 |
| int addStation() | 在线路上删除站点 |

## 2.6 系统测试

支持Educoder平台的所有可见测试用例与隐藏测试用例，均通过，如图2-4所示。

图2-4 测试通过

## 2.7 复杂度分析

同实验1，有向图的站点添加和遍历的时间复杂度为O(n)；在站点间添加新站点时，只需要将原本站点间的边节点删除，并在新节点与原先两节点间添加线路，最坏的情况，需要修改的边节点个数为6，即删除原先两节点间的两个边节点，并在新插入的站点与原有两个站点各添加两个边节点，时间复杂度为O(1)；删除站点时，只需要将顶点节点删除，删除与顶点节点相邻的边节点，以及在断开的两个顶点节点间添加边节点，最好的情况下，需要修改的边节点数为6，时间复杂度为O(1)。

有向图的存储所需的空间复杂度为O(n)；增删站点时，空间复杂度为O(1)。

## 2.8 结果分析

成功通过所有的给定测试用例，表明该邻接表存储有向图的方式满足实验需要，能够对地铁线路图进行存储的同时完成站点的增删。

## 2.9实验小结

在新增、删除站点时，需要考虑操作的站点的所处位置，如果位于两个站点之间，需要对两端分别进行增删操作，修改边节点的情况使路线连续；如果操作的节点位于路径末端，还需要考虑操作节点所处位置是路线的头部还是尾部，否则会影响路线的输出，如果新增、删除的节点是路线的头节点，需要修改有向图的heads数组，使路线的第一个节点指向正确的位置，如果修改的节点是路线末尾的节点，则不需要修改有向图的heads数组。

修改顶点节点的边节点时，需要为新的边节点添加距离和线路id，否则会导致线路不连续。为了方便遍历图的操作，最好将边节点的存储顺序与原顺序向同，避免在遍历时出现死循环或者无法完整遍历整条线路的情况。

# 3 从指定站点出发，计算出到另一个站点的最短距离和途径的地铁站序列

## 3.1 需求分析

### 3.1.1 功能需求

从指定站点出发，计算出到另一个站点的最短距离和途径的地铁站序列。不同的线路之间可能存在相同的站点作为换乘车站，因此在两个站点之间路线不唯一。使用邻接表构成有向图来表达地铁线路，存储武汉地铁1号线、2号线、6号线和7号线的部分站点信息。其中，地铁线路均为双向线路，相同站名的地铁站为转乘车站；

### 3.1.2 输入输出需求

使用邻接表构成有向图来表达地铁线路，存储武汉地铁1号线、2号线、6号线和7号线的部分站点信息。尝试利用我们提供的部分站点信息，用关卡一的代码生成有向图后，查找两个站点之间的最短路径。

输入形式：

总线路条数n

线路号1 站名1 到下一站的距离 站名2 …… 到下一站的距离 站名n 0(到下一站距离为0，代表该站是线路最后一站)

线路号n 站名1 到下一站的距离 站名2 …… 到下一站的距离 站名n 0(到下一站距离为0，代表该站是线路最后一站)

站名i 站名j（要查找的两个站点）

输出形式：

最短距离s 站名i 到下一站的距离 站名i+1 …… 站名j-1 到下一站的距离 站名j

## 3.2 总体设计

整个程序包括有向图的创建、添加站点、遍历有向图以及在指定两个站点间查找最优路径，有向图的创建等操作已经在实验1中完成，本实验中添加寻找最优路径的程序即可。

寻找最优路径时，需要考虑换乘情况，即涉及到不同线路站点的遍历，同时需要避免遍历已经遍历过的站点形成死循环，在找到最优路径后还需要能够找到最优路径所经过的站点，基于以上的需要，程序中使用深度优先搜索完成路径的查找。

## 3.3 数据结构

使用双向链表储存有向图中与顶点相邻的节点，即有向图中的边节点

1. **typedef** **struct** ArcNode  // 双向链表链接的边节点
2. {
3. **int** adjvex;  // 相邻节点在数组中的位置下标
4. **float** distance;  // 相邻节点与顶点的距离
5. **int** lineId;  // 所处的线路
6. **struct** ArcNode \*prior;  // 前一个与顶点相邻的边节点
7. **struct** ArcNode \*next;   // 后一个与顶点相邻的边节点
8. } ArcNode;

使用结构体储存顶点节点站点名和邻接边节点

1. **typedef** **struct** VNode    // 顶点节点
2. {
3. **char** name[20];  // 节点的站名
4. ArcNode \*firstArc;  // 第一个相邻的边节点
5. } VNode;

使用邻接表的方式存储有向图，vertices是存储顶点节点的数组，heads中存储地铁线路起始站点

1. **typedef** **struct** ALGraph  // 邻接表储存的有向图
2. {
3. VNode \*vertices[MAX\_VEX\_NUM];    // 顶点节点
4. **int** heads[MAX\_VEX\_NUM]; // 起始节点下标
5. **int** vexnum; // 顶点节点的个数
6. **int** headnum;    // 起始节点的数量
7. } ALGraph;

## 3.4 算法设计

寻找最优路径的主程序如图3-1所示。

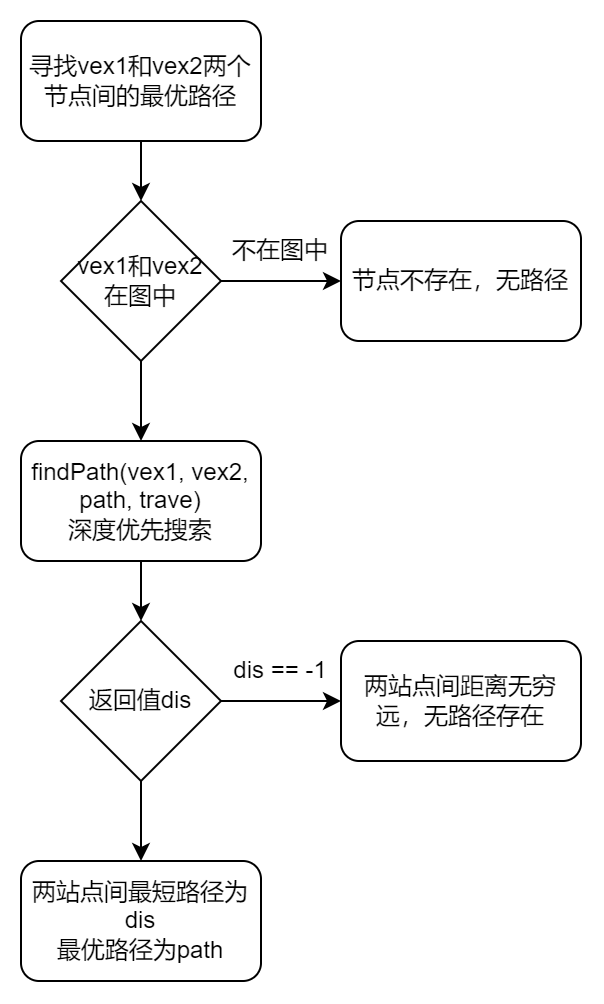


图3-1 主程序部分

使用深度优先搜索的过程如图3-2所示。

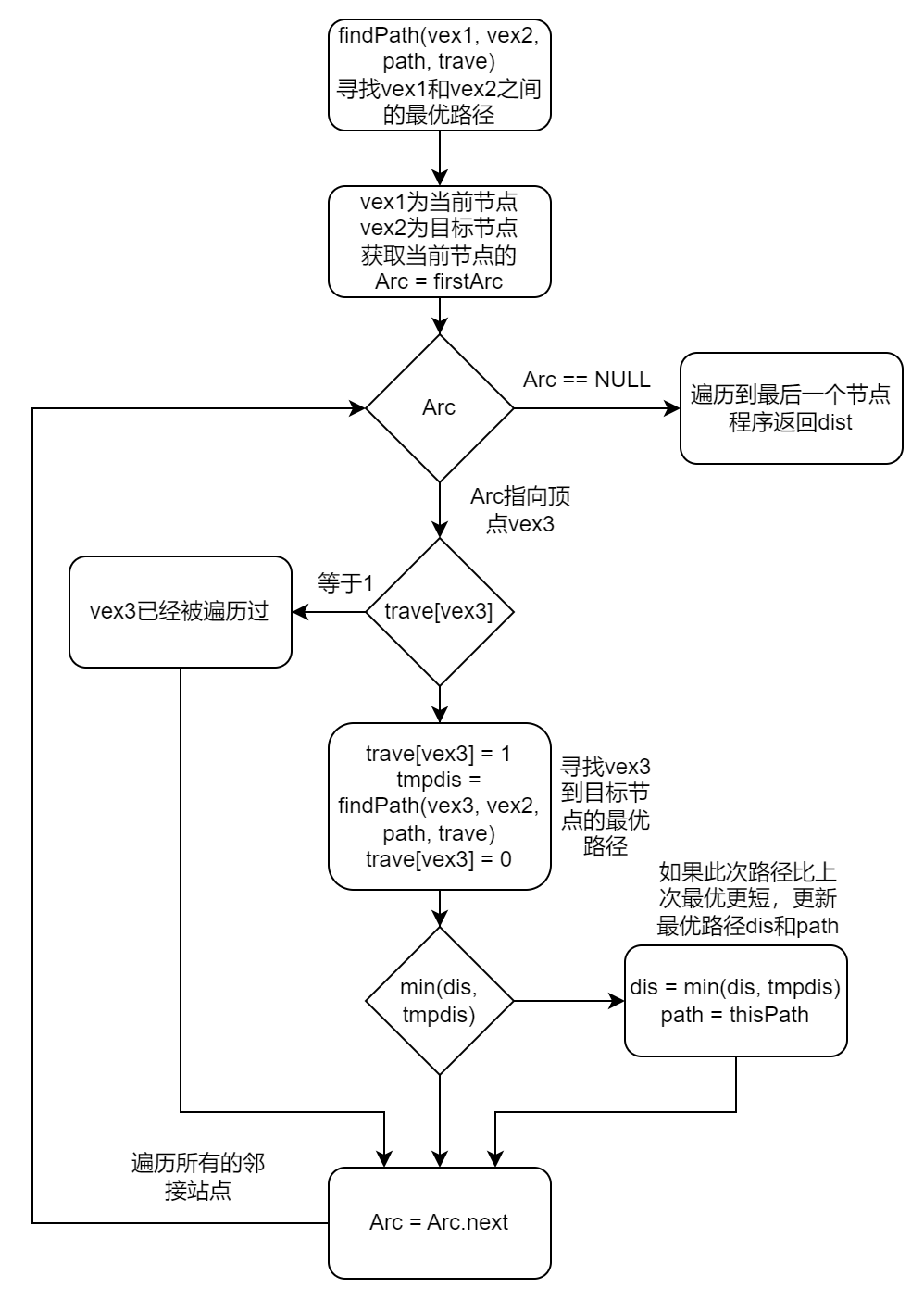


图3-2 DFS程序

## 3.5 系统实现

本程序全程在Ubuntu 22.04上编写，使用GCC 11.2.0编译、调试、运行，并最终在Educoder平台上运行通过。

主要函数以及功能如表3-1所示。

表3-1 主要函数及功能

|  |  |
| --- | --- |
| 函数名 | 主要功能 |
| int main() | 实现有向图顶点的增加和有向图的输出 |
| void initGraph() | 完成有向图的初始化 |
| int inGraph() | 判断某一站点是否在图中 |
| void addVex() | 将新的顶点加入有向图中 |
| void addArc() | 在有向图的两个站点间添加边节点 |
| void printLine() | 输出线路信息 |
| int inline() | 判断某一站点是否在某一线路上 |
| float findPath() | 使用深度优先搜索查找两站点间的最短距离并找到最短路径 |

## 3.6 系统测试

支持Educoder平台的所有可见测试用例与隐藏测试用例，均通过，如图3-3所示。

图3-3 测试通过

## 3.7 复杂度分析

同实验1，有向图的创建、添加站点和遍历站点操作所需要的时间复杂度为O(n)； 使用深度优先搜索遍历邻接表，有向图中的顶点节点个数为n，边节点个数e=2n，因此时间复杂度为O(n+e)，即O(n)。

有向图的存储所需的空间复杂度为O(n)；使用深度优先搜索，需要一个链表用来储存路径，还需要一个长度为n的数组记录每一个站点是否被遍历过，因此空间复杂度为O(n)。

## 3.8 结果分析

成功通过所有的给定测试用例，表明该邻接表存储有向图的方式满足实验需要，能够成功计算地铁线路图中任意两站点间的最短距离以及最短路径。

## 3.9 实验小结

在对顶点的所有边节点进行遍历时，会进入到边节点相连的顶点节点进行下一次深度优先搜索，当没有标记已遍历过的节点时，就会出现从当前节点再次遍历到前一个已经遍历过的节点，程序出现死循环，因此需要添加辅助数组来标记每个节点在当前线路的遍历情况。

在通常情况下，在一条线路中只会向一个方向进行遍历，在遍历到线路末尾的顶点节点或到达目的节点时停止遍历，但是当遍历到换乘站点时，即与一个顶点相邻的多个边节点处于不同线路时，就需要对所有未遍历的方向进行遍历。由于未考虑到换乘情况，仅在遍历一条线路后就结束遍历，造成当两个站点不再同一条线路上时无法找到路径。修改程序使其遍历所有符合遍历条件的站点后，成功解决上述问题。

使用DFS遍历有向图，需要考虑递归结束的条件，当遍历到目标节点后或者遍历到线路末尾时结束遍历。由于需要寻找最优路径，除了需要对所有路线进行遍历，比较路线长度外，还需要辅助链表来记录上一次优先路径，否则最后程序得到的路径为最后一次可行路径，而非最优路径。

# 参考文献

[1] 严蔚敏等. 数据结构(C语言版). 清华大学出版社

[2] [Larry Nyhoff](http://www.calvin.edu/~nyhl/index.html). [ADTs, Data Structures, and Problem Solving with C++.](http://vig.prenhall.com/catalog/academic/product/0,1144,0131409093,00.html)Second Edition, [Calvin College](http://cs.calvin.edu/), 2005

[3] 严蔚敏等.数据结构题集(C语言版). 清华大学出版社

# **附录 基于链表的树、图实现的源程序**

一、基于链表的建图

1. **int** main()
2. {
3. **int** n, i;
4. scanf("%d", &n);
5. ALGraph \*StationGraph = malloc(**sizeof**(ALGraph));  // 创建地铁线路图
6. initGraph(StationGraph);
7. **for** (i = 0; i < n; i++)
8. {
9. **int** lineId; // 线路ID
10. **char** name[20];  // 站点名字
11. **float** distance = -1.0; // 距离
12. VNode \*priorNode = NULL; // 前一个站点，用于连接后一个站点
13. scanf("%d", &lineId);
14. scanf("%s", name);
15. **while** (distance != 0)
16. {
17. VNode \*curNode; // 当前站点
18. **if** (inGraph(StationGraph, name) != -1)  // 如果name站点已经在图中
19. {
20. curNode = StationGraph->vertices[inGraph(StationGraph, name)];  // 从图中取出该站点
21. }
22. **else**    // 当name站点不在图中时
23. {
24. curNode = malloc(**sizeof**(VNode)); // 创建新的站点
25. initNode(curNode, name);
26. addVex(StationGraph, curNode);  // 将节点加入到图中
27. }
28. addArc(StationGraph, priorNode, curNode, lineId, distance);   // 在当前站点与前一个站点见添加线路
29. priorNode = curNode;
30. scanf("%f", &distance);
31. **if** (distance == 0)
32. {
33. **break**;
34. }
35. scanf("%s", name);
36. }
37. }
38. // 输出图
39. **for** (i = 0; i < StationGraph->headnum; i++)
40. {
41. **int** priorvex = -1;  // 前一个站点的下标
42. **int** vex = StationGraph->heads[i];   // 当前站点的下标
43. **int** lineId = StationGraph->vertices[vex]->firstArc->lineId;   // 取出lineId
44. VNode \*curNode = StationGraph->vertices[vex];   // 当前节点
45. printf("%d", lineId);
46. **char** \*name = curNode->name;
47. **float** distance;
48. printf(" %s", name);
49. **while** (1)
50. {
51. ArcNode \*tmp = curNode->firstArc;   // 当前节点的邻接站点
52. **while** (tmp != NULL) // 找到lineId与当前线路lineId相同，且指向后的邻接站点
53. {
54. **if** (tmp->lineId != lineId || tmp->adjvex == priorvex)
55. {
56. tmp = tmp->next;
57. }
58. **else**
59. {
60. **break**;
61. }
62. }
64. **if** (tmp != NULL)    // 若当前站点不为结尾站点，则存在指向后的路线，输出该路线信息
65. {
66. priorvex = vex;
67. vex = tmp->adjvex;
68. curNode = StationGraph->vertices[vex];
69. name = curNode->name;
70. distance = tmp->distance;
71. printf(" %.02f %s", distance, name);
72. }
73. **else**    // 如果为结尾站点，遍历下一条线路
74. {
75. printf("\n");   // 换行下一条线路
76. **break**;
77. }
78. }
79. }
80. **return** 0;
81. }

二、增加站点，删除

1. **int** main()
2. {
3. **int** n, i;
4. scanf("%d", &n);
5. ALGraph \*StationGraph = malloc(**sizeof**(ALGraph)); // 创建地铁线路图
6. initGraph(StationGraph);
7. **for** (i = 0; i < n; i++)
8. {
9. **int** lineId;              // 线路ID
10. **char** name[20];           // 站点名字
11. **float** distance = -1.0;   // 距离
12. VNode \*priorNode = NULL; // 前一个站点，用于连接后一个站点
13. scanf("%d", &lineId);
14. scanf("%s", name);
15. **while** (distance != 0)
16. {
17. VNode \*curNode;                        // 当前站点
18. **if** (inGraph(StationGraph, name) != -1) // 如果name站点已经在图中
19. {
20. curNode = StationGraph->vertices[inGraph(StationGraph, name)]; // 从图中取出该站点
21. }
22. **else** // 当name站点不在图中时
23. {
24. curNode = malloc(**sizeof**(VNode)); // 创建新的站点
25. initNode(curNode, name);
26. addVex(StationGraph, curNode); // 将节点加入到图中
27. }
28. addArc(StationGraph, priorNode, curNode, lineId, distance); // 在当前站点与前一个站点见添加线路
29. priorNode = curNode;
30. scanf("%f", &distance);
31. **if** (distance == 0)
32. {
33. **break**;
34. }
35. scanf("%s", name);
36. }
37. }
38. // 输出图
39. **for** (i = 0; i < StationGraph->headnum; i++)
40. {
41. printLine(StationGraph, i);
42. }
44. // 增、删站点
45. **char** mode[20]; // 操作模式
46. scanf("%s", mode);
47. **int** lineId;
48. scanf("%d", &lineId);
49. **if** (strcmp(mode, "add") == 0) // 添加站点
50. {
51. **int** ret = addStation(StationGraph, lineId);
52. **if** (ret == -1)
53. {
54. printf("增加失败，没有与输入的增加位置前一站点同名的站点\n");
55. **return** 0;
56. }
57. **else** **if** (ret == -2)
58. {
59. printf("增加失败，已有同名站点\n");
60. **return** 0;
61. }
62. }
63. **else** **if** (strcmp(mode, "delete") == 0) // 删除站点
64. {
65. **char** name[20];
66. scanf("%s", name);
67. **if** (inLine(StationGraph, name, lineId) == -1) // 线路中没有name站点
68. {
69. printf("删除失败，没有同名站点\n");
70. **return** 0;
71. }
72. deleteVex(StationGraph, name, lineId); // 删除线路上的该站点
73. }
74. **for** (i = 0; i < StationGraph->headnum; i++) // 输出该被修改的线路
75. {
76. **int** vex = StationGraph->heads[i];                         // 当前站点的下标
77. **int** line = StationGraph->vertices[vex]->firstArc->lineId; // 取出lineId
78. **if** (line == lineId)
79. {
80. printLine(StationGraph, i);
81. }
82. }
83. **return** 0;
84. }

三、从指定站点出发，计算出到另一个站点的最短距离和途径的地铁站序列

1. **int** main()
2. {
3. **int** n, i;
4. scanf("%d", &n);
5. ALGraph \*StationGraph = malloc(**sizeof**(ALGraph)); // 创建地铁线路图
6. initGraph(StationGraph);
7. **for** (i = 0; i < n; i++)
8. {
9. **int** lineId;              // 线路ID
10. **char** name[20];           // 站点名字
11. **float** distance = -1.0;   // 距离
12. VNode \*priorNode = NULL; // 前一个站点，用于连接后一个站点
13. scanf("%d", &lineId);
14. scanf("%s", name);
15. **while** (distance != 0)
16. {
17. VNode \*curNode;                        // 当前站点
18. **if** (inGraph(StationGraph, name) != -1) // 如果name站点已经在图中
19. {
20. curNode = StationGraph->vertices[inGraph(StationGraph, name)]; // 从图中取出该站点
21. }
22. **else** // 当name站点不在图中时
23. {
24. curNode = malloc(**sizeof**(VNode)); // 创建新的站点
25. initNode(curNode, name);
26. addVex(StationGraph, curNode); // 将节点加入到图中
27. }
28. addArc(StationGraph, priorNode, curNode, lineId, distance); // 在当前站点与前一个站点见添加线路
29. priorNode = curNode;
30. scanf("%f", &distance);
31. **if** (distance == 0)
32. {
33. **break**;
34. }
35. scanf("%s", name);
36. }
37. }
38. // 输出图
39. // for (i = 0; i < StationGraph->headnum; i++)
40. // {
41. //     printLine(StationGraph, i);
42. // }
44. **char** name1[20] = {'\0'};
45. **char** name2[20] = {'\0'};
46. scanf("%s %s", name1, name2);
47. **float** distance = -1;
48. **if** (inGraph(StationGraph, name1) != -1 && inGraph(StationGraph, name2) != -1)
49. {
50. ArcNode \*path = malloc(**sizeof**(ArcNode));   // 使用双向链表存储路线信息
51. **int** trave[MAX\_VEX\_NUM] = {0};
52. trave[inGraph(StationGraph, name1)] = 1;
53. distance = findPath(StationGraph, path, name1, name2, trave);
54. **if** (distance != -1)
55. {
56. printf("%.02f ", distance);
57. **while** (path->next != NULL)
58. {
59. **char** \*name = StationGraph->vertices[path->adjvex]->name;
60. printf("%s %.02f ", name, path->distance);
61. path = path->next;
62. }
63. **char** \*name = StationGraph->vertices[path->adjvex]->name;
64. printf("%s", name);
65. }
66. **else**
67. {
68. printf("%s与%s之间没有相连的线路", name1, name2);
69. }
70. }
71. **else**
72. {
73. printf("同名站点不存在");
74. }
75. **return** 0;
76. }

四、函数的定义、声明

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
5. #define MAX\_VEX\_NUM 100 // 最大节点数量
7. **typedef** **struct** ArcNode  // 双向链表链接与顶点相连的节点
8. {
9. **int** adjvex; // 相邻节点在数组中的位置下表
10. **float** distance; // 相邻节点与顶点的距离
11. **int** lineId; // 所处的线路
12. **struct** ArcNode \*prior;  // 前一个与顶点相邻的节点
13. **struct** ArcNode \*next;   // 后一个与顶点相邻的节点
14. } ArcNode;
16. **typedef** **struct** VNode    // 顶点节点
17. {
18. **char** name[20];  // 节点的站名
19. ArcNode \*firstArc;  // 第一个相邻的节点
20. } VNode;
22. **typedef** **struct** ALGraph  // 邻接表储存的有向图
23. {
24. VNode \*vertices[MAX\_VEX\_NUM];    // 顶点节点
25. **int** heads[MAX\_VEX\_NUM]; // 起始节点下标
26. **int** vexnum; // 顶点节点的个数
27. **int** headnum;    // 起始节点的数量
28. } ALGraph;
30. **void** initGraph(ALGraph \*Graph);   // 初始化图
31. **void** initNode(VNode \*Node, **char** \*name);     // 初始化节点
32. **int** inGraph(ALGraph \*Graph, **char** \*name); // 判断name站点是否在图中
33. **void** addVex(ALGraph \*Graph, VNode \*Node);   // 将站点加入图中
34. **void** addArc(ALGraph \*Graph, VNode \*Node1, VNode \*Node2, **int** lineId, **float** distance);    // 为相邻的两个站点添加线路
35. **void** printLine(ALGraph \*Graph, **int** headn);                                           // 输出线路
36. **int** inLine(ALGraph \*Graph, **char** \*name, **int** lineId);                                  // 判断name站点是否在线路lineId上
37. **void** deleteVex(ALGraph \*Graph, **char** \*name, **int** lineId);                              // 删除线路lineId上的name站点
38. **int** addStation(ALGraph \*Graph, **int** lineId);                                          // 添加站点
39. **float** findPath(ALGraph \*Graph, ArcNode \*path, **char** \*name1, **char** \*name2, **int** \*trave);
40. // 查找name1和name2之间的距离, trave指向遍历过的站点，避免重复遍历
42. **void** initGraph(ALGraph \*Graph) // 初始化图
43. {
44. Graph->vexnum = 0;  // 初始化图中储存的顶点数目
45. Graph->headnum = 0; // 初始化图中的起始站点数为0
46. }
48. **void** initNode(VNode \*Node, **char** \*name) // 初始化节点
49. {
50. strcpy(Node->name, name); // 站点名
51. Node->firstArc = NULL;    // 初始化时站点无邻接点
52. }
54. **int** inGraph(ALGraph \*Graph, **char** \*name) // 判断name站点是否在图中
55. {
56. **int** i;
57. **for** (i = 0; i < Graph->vexnum; i++) // 遍历图的所有节点
58. {
59. **if** (strcmp(name, Graph->vertices[i]->name) == 0) // 如果name在图中，返回其在数组中的下标
60. {
61. **return** i;
62. }
63. }
64. **return** -1; // 否则返回-1
65. }
67. **void** addVex(ALGraph \*Graph, VNode \*Node) // 将节点储存到图中
68. {
69. Graph->vertices[Graph->vexnum] = Node;
70. Graph->vexnum++; // 图中的顶点数量+1
71. }
73. **void** addArc(ALGraph \*Graph, VNode \*Node1, VNode \*Node2, **int** lineId, **float** distance) // 为相邻的两个站点添加线路
74. {
75. **if** (distance != -1.0) // 为-1是代表该节点为第一个节点，不存在前一个节点
76. {
77. **int** adjvex1 = inGraph(Graph, Node1->name);   // 前一个节点在数组中的下标
78. **int** adjvex2 = inGraph(Graph, Node2->name);   // 后一个节点在数组中的下标
79. ArcNode \*arcNode1 = malloc(**sizeof**(ArcNode)); // 前一个节点的邻接节点为后一个节点
80. ArcNode \*arcNode2 = malloc(**sizeof**(ArcNode)); // 后一个节点的邻接节点为前一个节点
81. arcNode1->adjvex = adjvex2;
82. arcNode1->lineId = lineId;
83. arcNode1->distance = distance;
84. arcNode1->next = NULL;
85. **if** (Node1->firstArc != NULL) // 将新的邻接节点添加到节点邻接链表末尾
86. {
87. ArcNode \*tmp = Node1->firstArc;
88. **while** (tmp->next != NULL)
89. tmp = tmp->next;
90. arcNode1->prior = tmp;
91. tmp->next = arcNode1;
92. }
93. **else**
94. {
95. arcNode1->prior = Node1->firstArc;
96. Node1->firstArc = arcNode1;
97. }
98. arcNode2->adjvex = adjvex1;
99. arcNode2->lineId = lineId;
100. arcNode2->distance = distance;
101. arcNode2->next = NULL;
102. **if** (Node2->firstArc != NULL) // 将新的邻接节点添加到节点邻接链表末尾
103. {
104. ArcNode \*tmp = Node2->firstArc;
105. **while** (tmp->next != NULL)
106. tmp = tmp->next;
107. arcNode2->prior = tmp;
108. tmp->next = arcNode2;
109. }
110. **else**
111. {
112. arcNode2->prior = Node2->firstArc;
113. Node2->firstArc = arcNode2;
114. }
115. }
116. **else** // 如果是第一个节点，将其加入到起始节点中
117. {
118. **int** adjvex = inGraph(Graph, Node2->name);
119. Graph->heads[Graph->headnum] = adjvex;
120. Graph->headnum++;
121. }
122. }
124. **int** inLine(ALGraph \*Graph, **char** \*name, **int** lineId) // 判断name站点是否在线路lineId上
125. {
126. **int** k = inGraph(Graph, name);
127. **if** (k == -1)
128. {
129. **return** -1;
130. }
131. **else**
132. {
133. ArcNode \*arc = Graph->vertices[k]->firstArc; // 取出该站点的邻接信息
134. **while** (arc != NULL)
135. {
136. **if** (arc->lineId == lineId) // 如果站点在线路上，则返回其存储下标
137. {
138. **return** k;
139. }
140. arc = arc->next;
141. }
142. }
143. **return** -1;
144. }
146. **void** printLine(ALGraph \*Graph, **int** headn)
147. {
148. **int** priorvex = -1;                                   // 前一个站点的下标
149. **int** vex = Graph->heads[headn];                       // 当前站点的下标
150. **int** lineId = Graph->vertices[vex]->firstArc->lineId; // 取出lineId
151. VNode \*curNode = Graph->vertices[vex];               // 当前节点
152. printf("%d", lineId);
153. **char** \*name = curNode->name;
154. **float** distance;
155. printf(" %s", name);
156. **while** (1)
157. {
158. ArcNode \*tmp = curNode->firstArc; // 当前节点的邻接站点
159. **while** (tmp != NULL)               // 找到lineId与当前线路lineId相同，且指向后的邻接站点
160. {
161. **if** (tmp->lineId != lineId || tmp->adjvex == priorvex)
162. {
163. tmp = tmp->next;
164. }
165. **else**
166. {
167. **break**;
168. }
169. }
171. **if** (tmp != NULL) // 若当前站点不为结尾站点，则存在指向后的路线，输出该路线信息
172. {
173. priorvex = vex;
174. vex = tmp->adjvex;
175. curNode = Graph->vertices[vex];
176. name = curNode->name;
177. distance = tmp->distance;
178. printf(" %.02f %s", distance, name);
179. }
180. **else** // 如果为结尾站点，遍历下一条线路
181. {
182. printf("\n"); // 换行下一条线路
183. **break**;
184. }
185. }
186. }
188. **void** deleteVex(ALGraph \*Graph, **char** \*name, **int** lineId) // 删除线路lineId上的name站点
189. {
190. **int** k = inGraph(Graph, name);
191. VNode \*curNode = Graph->vertices[k];
192. ArcNode \*arc = curNode->firstArc;
193. ArcNode \*arc1 = NULL, \*arc2 = NULL;
194. **while** (arc != NULL)
195. {
196. **if** (arc->lineId == lineId)
197. {
198. **if** (arc1 == NULL)
199. {
200. arc1 = arc;
201. }
202. **else**
203. {
204. arc2 = arc;
205. }
206. }
207. arc = arc->next;
208. }
209. **if** (arc2 == NULL) // 只有一个相邻的节点，该节点为头节点或者尾节点
210. {
211. **int** vexn = arc1->adjvex;
212. **int** i;
213. **for** (i = 0; i < Graph->headnum; i++) // 如果为头节点
214. {
215. **if** (k == Graph->heads[i])
216. {
217. Graph->heads[i] = vexn; // 将与其相邻的节点修改为头节点
218. }
219. }
220. arc = Graph->vertices[vexn]->firstArc;
221. **while** (arc != NULL)
222. {
223. **if** (arc->adjvex == k && arc->lineId == lineId) // 从邻接点中删除该条边
224. {
225. **if** (arc->prior != NULL)
226. {
227. arc->prior->next = arc->next;
228. }
229. **if** (arc->next != NULL)
230. {
231. arc->next->prior = arc->prior;
232. **if** (arc->prior == NULL) // arc为firstArc，需要将next设置为firstArc
233. {
234. Graph->vertices[vexn]->firstArc = arc->next;
235. }
236. }
237. free(arc);
238. **break**;
239. }
240. arc = arc->next;
241. }
242. **if** (arc1->prior != NULL)
243. {
244. arc1->prior->next = arc1->next;
245. }
246. **if** (arc1->next != NULL)
247. {
248. arc1->next->prior = arc1->prior;
249. }
250. free(arc1);
251. }
252. **else** // 存在两个相邻节点
253. {
254. **int** vexn1 = arc1->adjvex, vexn2 = arc2->adjvex;
255. **float** distance = arc1->distance + arc2->distance; // 新的距离
256. ArcNode \*tmparc1 = Graph->vertices[vexn1]->firstArc;
257. **while** (tmparc1 != NULL)
258. {
259. **if** (tmparc1->adjvex == k && tmparc1->lineId == lineId)
260. {
261. tmparc1->adjvex = vexn2;      // 连接断开的线路
262. tmparc1->distance = distance; // 为新连接的线路设置长度
263. **break**;
264. }
265. tmparc1 = tmparc1->next;
266. }
267. ArcNode \*tmparc2 = Graph->vertices[vexn2]->firstArc;
268. **while** (tmparc2 != NULL)
269. {
270. **if** (tmparc2->adjvex == k && tmparc2->lineId == lineId)
271. {
272. tmparc2->adjvex = vexn1;      // 连接断开的线路
273. tmparc2->distance = distance; // 为新连接的线路设置长度
274. **break**;
275. }
276. tmparc2 = tmparc2->next;
277. }
278. **if** (arc1->prior != NULL)
279. {
280. arc1->prior->next = arc1->next;
281. }
282. **if** (arc1->next != NULL)
283. {
284. arc1->next->prior = arc1->prior;
285. }
286. free(arc1);
287. **if** (arc2->prior != NULL)
288. {
289. arc2->prior->next = arc2->next;
290. }
291. **if** (arc2->next != NULL)
292. {
293. arc2->next->prior = arc2->prior;
294. }
295. free(arc2);
296. }
298. }
300. **int** addStation(ALGraph \*Graph, **int** lineId) // 添加站点
301. {
302. **float** priorDistance, nextDistance;
303. scanf("%f %f", &priorDistance, &nextDistance); // 输入到前一站和后一站的距离
304. **char** name[20], priorName[20] = {"\0"};
305. **if** (priorDistance != 0) // 如果前一站存在
306. {
307. scanf("%s", priorName);
308. **if** (inLine(Graph, priorName, lineId) == -1) // 前一站不在线路中
309. {
310. **return** -1;
311. }
312. }
313. scanf("%s", name);
314. **if** (inLine(Graph, name, lineId) != -1) // 站点name已经在线路中
315. {
316. **return** -2;
317. }
318. VNode \*curNode;
319. curNode = malloc(**sizeof**(VNode)); // 创建新的站点
320. initNode(curNode, name);
321. addVex(Graph, curNode); // 将节点加入到图中
322. **if** (priorDistance == 0) // 添加节点为头节点
323. {
324. **int** i;
325. **for** (i = 0; i < Graph->headnum; i++)
326. {
327. // 找到该条线路的头节点，替换之
328. VNode \*headNode = Graph->vertices[Graph->heads[i]];
329. **if** (inLine(Graph, headNode->name, lineId) != -1)
330. {
331. Graph->heads[i] = inGraph(Graph, name);                 // 修改头节点指向
332. addArc(Graph, curNode, headNode, lineId, nextDistance); // 在新节点和原来头节点间添加边
333. }
334. }
335. }
336. **else** // 为普通节点
337. {
338. **int** priorVexn = inGraph(Graph, priorName);
339. VNode \*priorNode = Graph->vertices[priorVexn];
340. ArcNode \*tmparc = priorNode->firstArc;
341. **int** nextVexn = -1;
342. **int** first = -1;
343. **while** (tmparc != NULL)
344. {
345. nextVexn = tmparc->adjvex; // 找到下一个站点
346. **if** (inLine(Graph, Graph->vertices[nextVexn]->name, lineId) != -1)
347. {
348. // printf("\n%s\n", Graph->vertices[nextVexn]->name);
349. **if** (first == -1)
350. {
351. first = nextVexn;
352. }
353. **else**{
354. **break**;
355. }
356. }
357. nextVexn = -1;
358. tmparc = tmparc->next;
359. }
360. **if** (first != -1 && nextVexn == -1)
361. {
362. nextVexn = first;
363. }
364. **if** (nextVexn == -1) // 没有下一个站点，向末尾添加
365. {
366. addArc(Graph, curNode, priorNode, lineId, priorDistance); // 在新节点和上一个节点间添加边
367. }
368. **else** // 两头都有节点
369. {
370. VNode \*nextNode = Graph->vertices[nextVexn]; // 下一个节点
371. tmparc = priorNode->firstArc;
372. **while** (tmparc != NULL)
373. {
374. **if** (tmparc->adjvex == nextVexn && tmparc->lineId == lineId) // 找到指向下一个节点的边，让其指向新节点
375. {
376. tmparc->adjvex = inGraph(Graph, name); // 指向新节点
377. tmparc->distance = priorDistance;
378. **break**;
379. }
380. tmparc = tmparc->next;
381. }
382. addArc(Graph, curNode, priorNode, lineId, priorDistance); // 在新节点和上一个节点间添加边
383. tmparc = nextNode->firstArc;
384. **while** (tmparc != NULL)
385. {
386. **if** (tmparc->adjvex == priorVexn && tmparc->lineId == lineId) // 找到指向下一个节点的边，让其指向新节点
387. {
388. tmparc->adjvex = inGraph(Graph, name); // 指向新节点
389. tmparc->distance = nextDistance;
390. **break**;
391. }
392. tmparc = tmparc->next;
393. }
394. addArc(Graph, curNode, nextNode, lineId, nextDistance); // 在新节点和上一个节点间添加边
395. }
396. }
397. **return** 1;
398. }
400. **float** findPath(ALGraph \*Graph, ArcNode \*path, **char** \*name1, **char** \*name2, **int** \*trave) // 查找name1和name2之间的距离
401. {
402. // printf("==%s==", name1);
403. **int** vexn1 = inGraph(Graph, name1);
404. **int** vexn2 = inGraph(Graph, name2);
405. path->adjvex = vexn1;
406. path->prior = NULL;
407. **if** (strcmp(name1, name2) == 0)
408. {
409. path->distance = 0;
410. path->lineId = 0;
411. path->next = NULL;
412. **return** 0;
413. }
414. **int** thisTrave[MAX\_VEX\_NUM] = {0};
415. **int** i;
416. **for** (i = 0; i < MAX\_VEX\_NUM; i++)   // 复制遍历过的节点
417. {
418. thisTrave[i] = \*(trave+i);
419. }
420. // 查找线路
421. **float** minDistance = -1;
422. **float** tmpDistance = -1;
423. ArcNode \*minPath = NULL;
424. ArcNode \*tmp = Graph->vertices[vexn1]->firstArc;
425. **while** (tmp != NULL) // 先将这一层所有站点加入thisTrave，防止下层再次遍历此层
426. {
427. thisTrave[tmp->adjvex] = 1;
428. tmp = tmp->next;
429. }
430. tmp = Graph->vertices[vexn1]->firstArc;
431. **while** (tmp != NULL)
432. {
433. **if** (\*(trave+tmp->adjvex) == 1) // 已经遍历过的站点，跳过遍历
434. {
435. tmp = tmp->next;
436. **continue**;
437. }
438. ArcNode \*tmpPath = malloc(**sizeof**(ArcNode));
439. **char** \*tmpName = Graph->vertices[tmp->adjvex]->name;
440. tmpDistance = findPath(Graph, tmpPath, tmpName, name2, thisTrave);  // 递归遍历相邻节点
441. **if** (tmpDistance != -1 && (minDistance == -1 || tmpDistance + tmp->distance < minDistance)) // 新的距离为最短距离，交换min值
442. {
443. minDistance = tmpDistance + tmp->distance;  // tmp->distance为遍历节点到name1节点的距离
444. path->distance = tmp->distance; // 记录name1站点到该站点的距离
445. path->lineId = tmp->lineId;
446. **while** (minPath != NULL)
447. {
448. ArcNode \*tmpNode = minPath->next;
449. free(minPath);
450. minPath = tmpNode;
451. }
452. minPath = tmpPath;
453. }
454. tmp = tmp->next;
455. }
456. **if** (minDistance != -1)  // 存在路线
457. {
458. path->next = minPath;
459. **if** (minPath != NULL)    // minPath为NULL时代表遍历到末尾
460. {
461. minPath->prior = path;
462. }
463. // printf("\n%s==%.02f\n",Graph->vertices[path->adjvex]->name, minDistance);
464. **return** minDistance;
465. }
466. **else**    // 线路不存在
467. {
468. path = NULL;
469. // printf("-1\n");
470. **return** -1;
471. }
472. }