

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： **信息安全2103班**

学 号： **U202112149**

姓 名： **李 翔**

指导教师： **邓贤君**

报告日期： **2022.11.1**

网络空间安全学院

目 录

**[1](#_bookmark2)** [基于链表的建图](#_bookmark2)  **[1](#_bookmark2)**

1.1 需求分析 [1](#_bookmark3)

1.2 总体设计 [1](#_bookmark4)

1.3 算法设计 [1](#_bookmark5)

1.4 系统实现 [2](#_bookmark6)

1.5 系统测试 [3](#_bookmark1)

1.6 结果分析 [3](#_bookmark1)

1.7 实验小结 [3](#_bookmark1)

**2** 增加站点，删除  **4**

2.1 需求分析 4

2.2 总体设计 4

2.3 算法设计 4

2.4 系统实现 7

2.5 系统测试 7

2.6 结果分析 7

2.7 实验小结 8

**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  **9**

3.1 需求分析 9

3.2 总体设计 9

3.3 算法设计 9

3.4 系统实现 13

3.5 系统测试 13

3.6 结果分析 14

3.7 实验小结 14

参考文献  **20**

附录 基于链式存储结构线性表实现的源程序  **21**

**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**

总体设计包括读入输入的数据，并存入n条双向链表之中，并且利用双向链表输出相关线路的信息。后利用双向链表作为中间数据的存储方式，通过相应的步骤构建出该地铁线路有向图的邻接表。

**1.3 算法设计**

**1.4 系统实现**

本程序在 MacOS系统上的 Clion IDE 编写、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。

|  |  |
| --- | --- |
| **函数名** | **主要功能** |
| DoubleNode \*Doubly\_list\_init(); | 初始化一个双向链表 |
| void Add\_rear(DoubleNode \*head, char \*name, float length)； | 在双向链表的尾部添加新结点 |
| void Add\_front(DoubleNode \*head, char \*name, float length)； | 在双向链表的首部添加新结点 |
| void print\_Doubly\_list(DoubleNode \*head)； | 从头至尾打印双向链表 |
| void Destroy\_Doubly\_list(DoubleNode \*head); | 双向链表的摧毁 |
| MetroGraph \*graph\_init(); | 初始化一个地铁有向图 |
| void graph\_destroy(MetroGraph \*G); | 销毁已经创建的图结构 |
| void Station\_insert(MetroGraph \*G, DoubleNode \*head); | 依据双向链表中的信息创建顶点表结点 |
| void EdgeNode\_insert(MetroGraph \*G, DoubleNode \*head); | 依据双向链表中的信息创建边表结点 |
| bool existed(MetroGraph \*G, char name[]); | 判断station是否已经存入了顶点表结点 |
| int locate(MetroGraph \*G, char name[]); | 查找图G中与name相同的顶点表中的元素，并返回stations数组的下标 |
| int main()； | 数据的读入输出，相关函数的调用 |

表 1-1 主要函数及功能

**1.5 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-

2 所示。

**1.6 结果分析**

成功通过所有的给定测试用例，表明该双向链表的操作程序设计成功实

现使用需求，并且正确存储了数据，在自己的IDE调试中也能够成功的创建一个完整的邻接图。

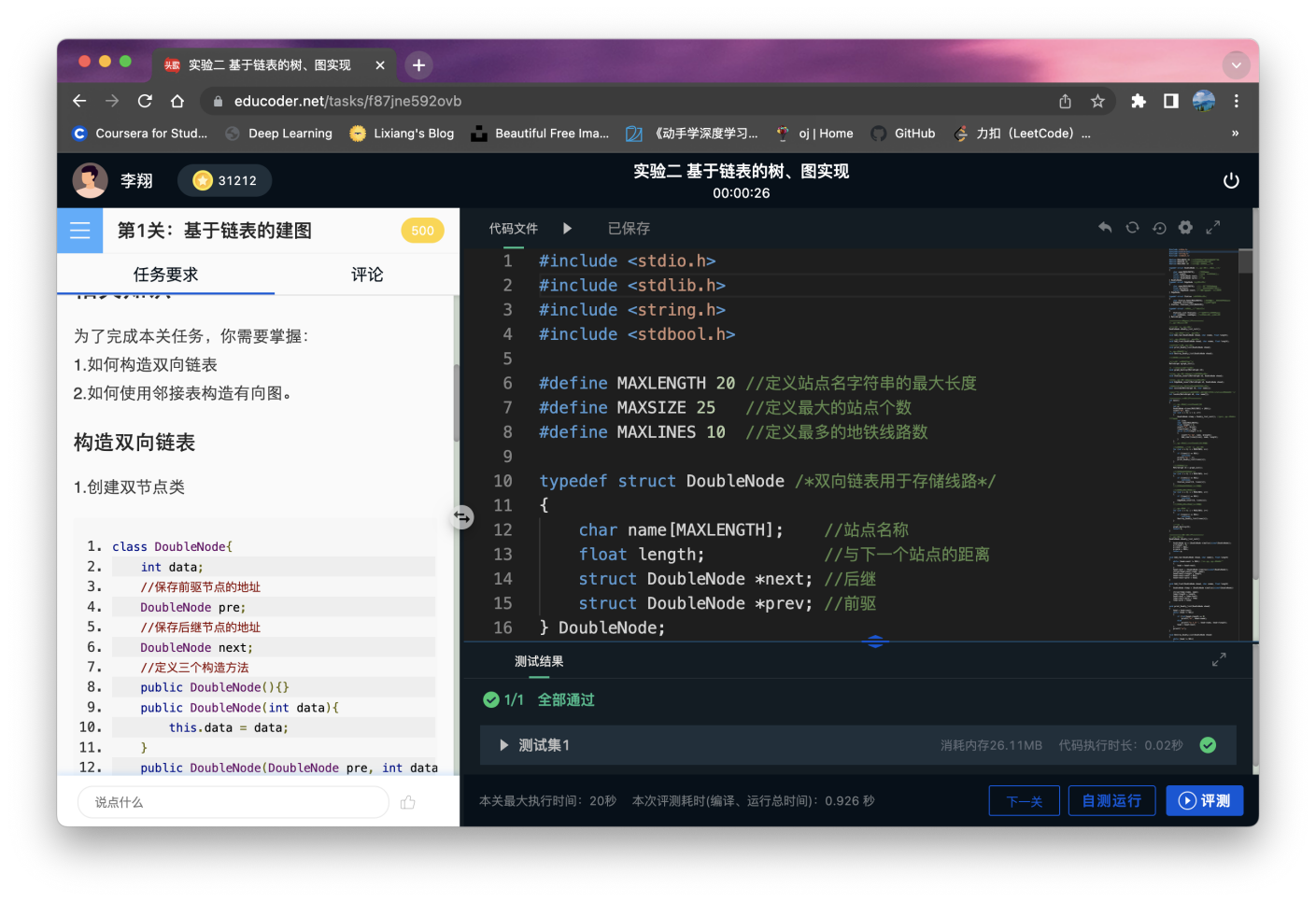


图 1-2 通过所有测试

**1.7 实验小结**

通过该次实验我学会了如何创建双向链表和邻接图。其实刚开始拿到这个题目的时候，有点不知所云，我认为此题并不需要使用到双向链表，意识到双向链表存储一条线路具有天然的优势（无论是输入还是输出方面），于是迎合题目使用了双向链表去暂存各条线路的信息，其次打印相关线路的信息，最后通过相关函数配合n条双向链表去构建了一个邻接图。初次构建邻接图也是花费了不小的力气，但明晰分步创建表头结点和边表结点之后，最终还是成功完成了本题。

**2 增加站点、删除**

**2.1 需求分析**

**2.1.1 功能需求**

使用第一关的邻接表构成有向图来表达地铁线路，并且能够实现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**

**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**

在第一关已有代码的基础上，进行后续功能的实现。包括邻接图中对于相应结点的添加和删除。

**2.3 算法设计**

**2.4 系统实现**

本程序在 MacOS系统上的 Clion IDE 编写、编译、调试、运行，并

最终在 Educoder 平台上运行通过。

主要函数（只列出相比第一关增加的函数）以及功能如表 2-1 所示。

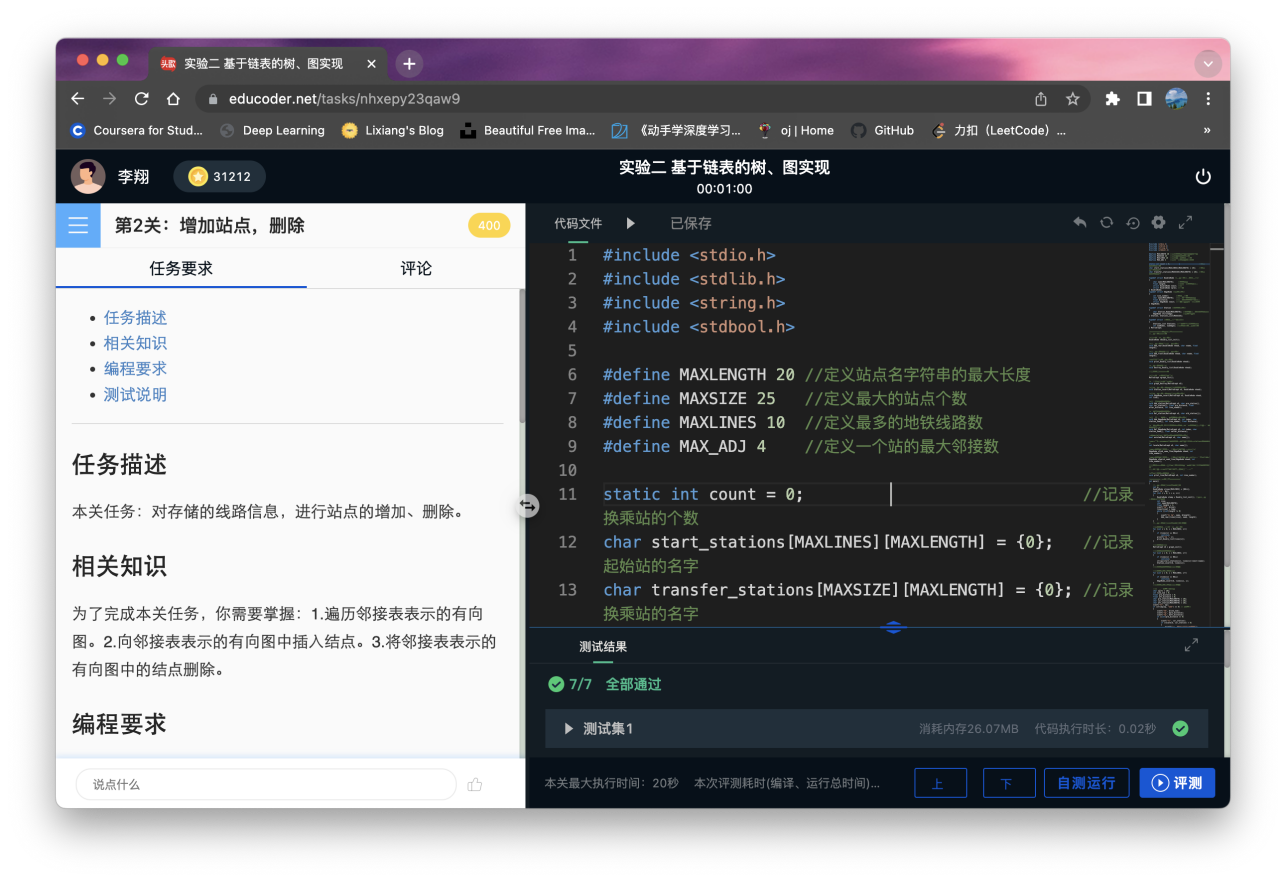
|  |  |
| --- | --- |
| **函数名** | **主要功能** |
| void Add\_station(MetroGraph \*G, char pre\_station[], char cur\_station[], float pre\_distance, float after\_distance, int line\_number); | 添加有效的地铁站点 |
| void Del\_station(MetroGraph \*G, char old\_station[]); | 删除有效的地铁站点 |
| void Add\_EdgeNode(MetroGraph \*G, int index, char station\_name[], int Line\_number, float Distance); | 添加由于删除或增加新产生的边表结点 |
| void Del\_EdgeNode(MetroGraph \*G, int index, char station\_name[], float \*after\_distance); | 删除边表结点,并且直接存储结点的离后一个站点的距离，因为删除之后要合并距离 |
| EdgeNode \*Find\_same\_line(EdgeNode \*head, int Line\_number); | 查找顶点的邻接点中线路号相同的元素,配对作用 |
| EdgeNode \*Search\_same\_line(EdgeNode \*head, int Line\_number); | 查找顶点的邻接点中线路号相同的元素,不配对，找出所有相同的 |
| void print\_line(MetroGraph \*G, int Line\_number); | 根据邻接表打印地铁线路 |

表 2-1 主要函数及功能

**2.5 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-

2 所示。

图 2-2 通过所有测试

**2.6 结果分析**

成功的通过了Educoder 平台所有的测试用例，表明该程序对于邻接图结点的增加和删除操作正确，且对于所有不同的情况判断也均正确

**2.7 实验小结**

通过该次实验我学会了如何对邻接图进行插入与删除结点和如何遍历打印邻接图的特定的结点。本题的难点首先在于插入和删除前的条件判断，只有判定增加或者删除有效之后才能进行后续的操作。其次在于增加和删除后的一些细节问题，例如删除后前后距离的求和、增加（或删除）头结点和边表结点以及若删除的是换乘站，要修改的数据甚多。但经过一天的细细打磨也都实现了。在打印线路的函数中也遇到了不知到如何依赖邻接图并按照特定线路去输出，通过巧妙的设置起始站数组与在EdgeNode结构体中新增line\_number数据去区分不同线路，最终能够得以正常的运行。

**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**

在前两关的代码基础上实现从指定站点出发，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，需要输出最短路径的值和该最短路径上的所有地铁站点。

**3.1.2 输入、输出需求**

**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

**参考信息：**

1号线 六渡桥 5.00 循礼门 4.00 大智路 4.00 三阳路 5.00 黄浦路 0

2号线 汉口火车站 10.00 范湖 2.00 王家墩东 2.00 青年路 4.00 中山公园 2.00 循礼门 2.00 江汉路 5.00 积玉桥 0

6号线 三眼桥 3.00 香港路 2.00 苗栗路 2.00 大智路 5.00 江汉路 1.00 汉正街 0

7号线 武汉商务区 1.00 王家墩东 2.00 取水楼 2.00 香港路 6.00 三阳路 9.00 徐家棚 3.00 三角路 3.00 三层楼 3.00 积玉桥 0

**3.2 总体设计**

在前两关代码的基础上只需利用迪杰斯特拉算法（Dijkstra）求得最短路径即可。

**3.3 算法设计**

**3.4 系统实现**

本程序在 MacOS系统上的 Clion IDE 编写、编译、调试、运行，并

最终在 Educoder 平台上运行通过。

主要函数（只列出相比前两关关增加的函数）以及功能如表 3-1 所示。

|  |  |
| --- | --- |
| **函数名** | **主要功能** |
| void ShortestPath\_Dijkstra(MetroGraph \*G, int v0, int Patharc[], float ShortPathTable[]); | 利用迪杰斯特拉算法计算v0到其他所有点的最短路径 |
| void Print\_ShortestPath(MetroGraph \*G, int Vend, float Sum, const int Patharc[], const float ShortPathTable[h]); | 根据最短路径值表（到vi的最短路径值）和最短路径表（存储最短路径经过点的下标）打印最短路径 |

表 3-1 主要函数及功能

**3.5 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- 2 所示。

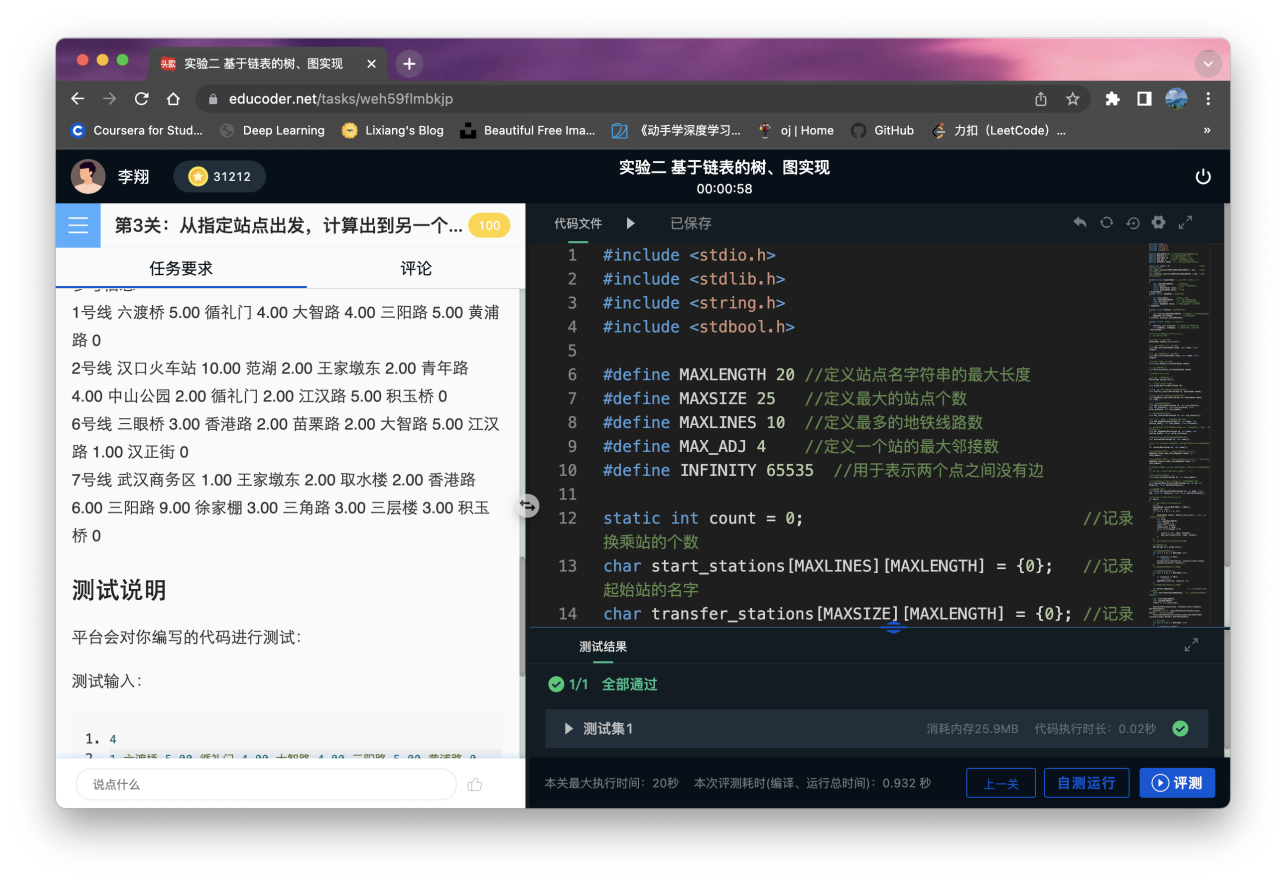


图 3-2 通过所有测试

**3.6 结果分析**

成功的通过了Educoder 平台所有的测试用例，在IDE的调试过程中也认真观察了中间变量的值，并且能够根据最短路径值表（到vi的最短路径值）和最短路径表（存储最短路径经过点的下标）打印最短路径。说明迪杰斯特拉算法（Dijkstra）的运用是成功的。

**3.7 实验小结**

通过该次实验我学会了如何利用迪杰斯特拉算法进行最短路径的求解，同时自己也根据课本上的邻接矩阵的版本自己修改成了邻接图的版本，深刻理解了该算法通过贪心策略从局部最优到全局最优的算法思想。输出方面遇到的难题是通过终点和Patharc数组（用于存储最短路径上的站点）逆推得到完整的路径，通过递归的压栈后序输出才能够实现从起点到终点的输出。

**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] 严蔚敏等.数据结构题集(C 语言版). 清华大学出版社

[3] 程杰.大话数据结构. 清华大学出版社

**附录 基于链表的树、图实现的源程序**

为了观看方面，下面的代码可以通过如下链接进行在线的访问。

[程序源代码](https://github.com/hustlixiang21/Hust_Data_Structure/blob/main/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84%E5%AE%9E%E9%AA%8C%E8%AF%BE/2_3/main.c)

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <stdbool.h>
5. #define MAXLENGTH 20 *//定义站点名字符串的最大长度*
6. #define MAXSIZE 25   *//定义最大的站点个数*
7. #define MAXLINES 10  *//定义最多的地铁线路数*
8. #define MAX\_ADJ 4    *//定义一个站的最大邻接数*
9. #define INFINITY 65535  *//用于表示两个点之间没有边*
10. static int count = 0;                           *//记录换乘站的个数*
11. char start\_stations[MAXLINES][MAXLENGTH] = {0}; *//记录起始站的名字*
12. char transfer\_stations[MAXSIZE][MAXLENGTH] = {0};*//记录换乘站的名字*
13. typedef struct DoubleNode /\*双向链表用于存储线路\*/
14. {
15. char name[MAXLENGTH];    *//站点名称*
16. float length;            *//与下一个站点的距离*
17. struct DoubleNode \*next; *//后继*
18. struct DoubleNode \*prev; *//前驱*
19. } DoubleNode;
20. typedef struct EdgeNode /\*边表结点\*/
21. {
22. int line\_number;       *//地铁线路号*
23. char name[MAXLENGTH];  *//用于存放站点的名称*
24. float distance;        *//用于存储站点之间的距离*
25. struct EdgeNode \*next; *//链域，指向下一个邻接点*
26. } EdgeNode;
27. typedef struct Station /\*顶点表结点\*/
28. {
29. char Station\_Name[MAXLENGTH]; *//顶点域用于存储顶点的信息*
30. EdgeNode \*firstedge;          *//边表头指针*
31. } Station, Stations\_List[MAXSIZE];
32. typedef struct /\*地铁线路图的定义\*/
33. {
34. Stations\_List Stations; *//图的点集，即站点信息*
35. int numNodes, numEdges; *//结点的个数和边的个数*
36. } MetroGraph;
37. */\*\*\*\*\*\*\*\*\*\*函数声明部分\*\*\*\*\*\*\*\*\*\*/*
38. *//双向链表相关函数*
39. */\*初始化一个双向链表\*/*
40. DoubleNode \*Doubly\_list\_init();
41. */\*在双向链表的尾部添加新结点\*/*
42. void Add\_rear(DoubleNode \*head, char \*name, float length);
43. */\*在双向链表的首部添加新结点\*/*
44. void Add\_front(DoubleNode \*head, char \*name, float length);
45. */\*从头至尾打印双向链表\*/*
46. void print\_Doubly\_list(DoubleNode \*head);
47. */\*双向链表的摧毁\*/*
48. void Destroy\_Doubly\_list(DoubleNode \*head);
49. *//邻接表创建相关函数*
50. */\*初始化一个地铁有向图\*/*
51. MetroGraph \*graph\_init();
52. */\*销毁已经创建的图结构\*/*
53. void graph\_destroy(MetroGraph \*G);
54. */\*依据双向链表中的信息创建顶点表结点\*/*
55. void Station\_insert(MetroGraph \*G, DoubleNode \*head);
56. */\*依据双向链表中的信息创建边表结点\*/*
57. void EdgeNode\_insert(MetroGraph \*G, DoubleNode \*head, int line);
58. */\*添加有效的地铁站点\*/*
59. void Add\_station(MetroGraph \*G, char pre\_station[], char cur\_station[], float pre\_distance, float after\_distance, int line\_number);
60. */\*删除有效的地铁站点\*/*
61. void Del\_station(MetroGraph \*G, char old\_station[]);
62. */\*添加由于删除或增加新产生的边表结点\*/*
63. void Add\_EdgeNode(MetroGraph \*G, int index, char station\_name[], int Line\_number, float Distance);
64. */\*删除边表结点,并且直接存储结点的离后一个站点的距离，因为删除之后要合并距离\*/*
65. void Del\_EdgeNode(MetroGraph \*G, int index, char station\_name[], float \*after\_distance);
66. */\*判断station是否已经存入了顶点表结点\*/*
67. bool existed(MetroGraph \*G, char name[]);
68. */\*查找图G中与name相同的顶点表中的元素，并返回stations数组的下标\*/*
69. int locate(MetroGraph \*G, char name[]);
70. */\*查找顶点的邻接点中线路号相同的元素,配对作用\*/*
71. EdgeNode \*Find\_same\_line(EdgeNode \*head, int Line\_number);
72. */\*查找顶点的邻接点中线路号相同的元素,不配对，找出所有相同的\*/*
73. EdgeNode \*Search\_same\_line(EdgeNode \*head, int Line\_number);
74. *//上面两个函数理论上应该是可以整合到一起的，但是我想了很久还是不太会*
75. *//主要是因为程序太长了，要改逻辑的话过于复杂*
76. */\*根据邻接表打印地铁线路\*/*
77. void print\_line(MetroGraph \*G, int Line\_number);
78. */\*利用迪杰斯特拉算法计算v0到其他所有点的最短路径\*/*
79. void ShortestPath\_Dijkstra(MetroGraph \*G, int v0, int Patharc[], float ShortPathTable[]);
80. */\*根据最短路径值表（到vi的最短路径值）和最短路径表（存储最短路径经过点的下标）打印最短路径\*/*
81. void Print\_ShortestPath(MetroGraph \*G, int Vend, float Sum, const int Patharc[], const float ShortPathTable[]);
82. */\*\*\*\*\*\*\*\*\*\*主函数部分\*\*\*\*\*\*\*\*\*\*/*
83. int main()
84. {
85. *//双向链表的创建与输入的读取*
86. int n;
87. DoubleNode \*lines[MAXLINES] = {NULL};
88. scanf("%d", &n);
89. for (int i = 0; i < n; i++)
90. {
91. DoubleNode \*temp = Doubly\_list\_init(); *//新建双向链表的虚拟头节点*
92. int Line;
93. char name[MAXLENGTH];
94. float length = 1;
95. scanf("%d", &Line);
96. lines[Line] = temp;
97. while ((int)length != 0)
98. {
99. scanf("%s %g", name, &length);
100. Add\_rear(lines[Line], name, length);
101. }
102. }
103. *//双向链表的创建与输入的读取(成功)*
104. */\*\*\*\*\*第一关代码开始\*\*\*\*\*/*
105. *//打印地铁线路(即一个双向链表)*
106. for (int i = 0; i < MAXLINES; i++)
107. {
108. if (lines[i] == NULL)
109. continue;
110. printf("%d ", i);
111. print\_Doubly\_list(lines[i]);
112. }
113. *//邻接表的创建*
114. MetroGraph \*G = graph\_init();
115. *//邻接表的顶点表的创建*
116. for (int i = 0; i < MAXLINES; i++)
117. {
118. if (lines[i] == NULL)
119. continue;
120. strcpy(start\_stations[i], lines[i]->next->name);
121. Station\_insert(G, lines[i]);
122. }
123. *//邻接表的顶点表的创建(成功)*
124. *//邻接表边表结点的创建*
125. for (int i = 0; i < MAXLINES; i++)
126. {
127. if (lines[i] == NULL)
128. continue;
129. EdgeNode\_insert(G, lines[i], i);
130. }
131. *//邻接表边表结点的创建(成功)*
132. */\*\*\*\*\*第一关代码结束\*\*\*\*\*/*
133. */\*\*\*\*\*第二关代码开始\*\*\*\*\*/*
134. *//添加和删除操作的解析*
135. char op[7] = {0};
136. int line\_num = 0;
137. float pre\_distance = 0;
138. float aft\_distance = 0;
139. char pre\_station[MAXLENGTH] = {0};
140. char cur\_station[MAXLENGTH] = {0};
141. char aft\_station[MAXLENGTH] = {0};
142. scanf("%s", op);
143. if (strcmp(op, "add") == 0) *// add操作*
144. {
145. scanf("%d", &line\_num);
146. scanf("%g", &pre\_distance);
147. scanf("%g", &aft\_distance);
148. if ((int)pre\_distance == 0)
149. {
150. scanf("%s", cur\_station);
151. if (locate(G, cur\_station) > 0)
152. {
153. printf("增加失败，已有同名站点");
154. }
155. else
156. {
157. strcpy(aft\_station, start\_stations[line\_num]);
158. strcpy(start\_stations[line\_num], cur\_station);
159. Add\_EdgeNode(G, locate(G, aft\_station), cur\_station, line\_num, aft\_distance);
160. strcpy(G->Stations[G->numNodes++].Station\_Name, cur\_station);
161. Add\_EdgeNode(G, locate(G, cur\_station), aft\_station, line\_num, aft\_distance);
162. print\_line(G, line\_num);
163. }
164. }
165. else
166. {
167. scanf("%s", pre\_station);
168. scanf("%s", cur\_station);
169. if (locate(G, pre\_station) == -1)
170. {
171. printf("增加失败，没有与输入的增加位置前一站点同名的站点");
172. }
173. else if (locate(G, cur\_station) > 0)
174. {
175. printf("增加失败，已有同名站点");
176. }
177. else
178. {
179. Add\_station(G, pre\_station, cur\_station, pre\_distance, aft\_distance, line\_num);
180. print\_line(G, line\_num);
181. }
182. }
183. }
184. else
185. {
186. scanf("%d", &line\_num);
187. scanf("%s", cur\_station);
188. if (locate(G, cur\_station) == -1)
189. {
190. printf("删除失败，没有同名站点");
191. }
192. else
193. {
194. Del\_station(G, cur\_station);
195. print\_line(G, line\_num);
196. }
197. }
198. */\*\*\*\*\*第二关代码结束\*\*\*\*\*/*
199. */\*\*\*\*\*第三关代码开始\*\*\*\*\*/*
200. int Patharc[MAXSIZE];             *//用于存储最短路径下标的数组*
201. float ShortPathTable[MAXSIZE];  *//用于存储到各点最短路径的权值和*
202. char start[MAXLENGTH];
203. char end[MAXLENGTH];
204. scanf("%s %s",start,end);
205. ShortestPath\_Dijkstra(G, locate(G,start),Patharc,ShortPathTable);
206. printf("%.2f %s ",ShortPathTable[locate(G,end)],start);  *//先打印路径最短*
207. Print\_ShortestPath(G,locate(G,end),ShortPathTable[locate(G,end)],Patharc,ShortPathTable);
208. */\*\*\*\*\*第三关代码结束\*\*\*\*\*/*
209. *//双向链表的销毁*
210. for (int i = 0; i < MAXLINES; i++)
211. {
212. if (lines[i] == NULL)
213. continue;
214. Destroy\_Doubly\_list(lines[i]);
215. }
216. *//图的销毁*
217. graph\_destroy(G);
218. return 0;
219. }
220. */\*\*\*\*\*\*\*\*\*\*函数实现部分\*\*\*\*\*\*\*\*\*\*/*
221. *//双向链表相关*
222. DoubleNode \*Doubly\_list\_init()
223. {
224. DoubleNode \*p = (DoubleNode \*)malloc(sizeof(DoubleNode));
225. p->length = 0;
226. p->next = NULL;
227. p->prev = NULL;
228. return p;
229. }
230. void Add\_rear(DoubleNode \*head, char name[], float length)
231. {
232. while (head->next != NULL) *//定位到双向链表的末尾*
233. {
234. head = head->next;
235. }
236. head->next = (DoubleNode \*)malloc(sizeof(DoubleNode));
237. strcpy(head->next->name, name);
238. head->next->length = length;
239. head->next->next = NULL;
240. head->next->prev = head;
241. }
242. void Add\_front(DoubleNode \*head, char \*name, float length)
243. {
244. DoubleNode \*temp = (DoubleNode \*)malloc(sizeof(DoubleNode));
245. strcpy(temp->name, name);
246. temp->length = length;
247. head->next = temp->next;
248. temp->next->prev = temp;
249. temp->prev = head;
250. }
251. void print\_Doubly\_list(DoubleNode \*head)
252. {
253. head = head->next;
254. while (head != NULL)
255. {
256. if ((int)head->length == 0)
257. printf("%s", head->name);
258. else
259. printf("%s %.2f ", head->name, head->length);
260. head = head->next;
261. }
262. printf("\n");
263. }
264. void Destroy\_Doubly\_list(DoubleNode \*head)
265. {
266. while (head != NULL)
267. {
268. DoubleNode \*temp = head;
269. head = head->next;
270. free(temp);
271. }
272. }
273. *//有向图邻接表相关*
274. MetroGraph \*graph\_init()
275. {
276. MetroGraph \*G = (MetroGraph \*)malloc(sizeof(MetroGraph));
277. G->numEdges = 0;
278. G->numNodes = 0;
279. for (int i = 0; i < MAXSIZE; i++)
280. {
281. G->Stations[i].firstedge = NULL;
282. }
283. return G;
284. }
285. void graph\_destroy(MetroGraph \*G)
286. {
287. for (int i = 0; i < G->numNodes; i++)
288. {
289. EdgeNode \*temp;
290. while (G->Stations[i].firstedge != NULL)
291. {
292. temp = G->Stations[i].firstedge;
293. G->Stations[i].firstedge = G->Stations[i].firstedge->next;
294. free(temp);
295. }
296. }
297. free(G);
298. }
299. void Station\_insert(MetroGraph \*G, DoubleNode \*head)
300. {
301. head = head->next; *//跳过双向链表的虚拟头结点*
302. while (head != NULL)
303. {
304. if (!existed(G, head->name)) *//如果站名在顶点表里不存在就加入*
305. {
306. strcpy(G->Stations[G->numNodes++].Station\_Name, head->name);
307. }
308. else
309. {
310. strcpy(transfer\_stations[count++], head->name);
311. }
312. head = head->next;
313. }
314. }
315. void EdgeNode\_insert(MetroGraph \*G, DoubleNode \*head, int line)
316. {
317. head = head->next;
318. int index;   *//用于定位Stations中的站点*
319. EdgeNode \*e; *//主要用于malloc作为一个临时的指针*
320. while (head->next != NULL)
321. {
322. index = locate(G, head->name);
323. e = (EdgeNode \*)malloc(sizeof(EdgeNode));
324. strcpy(e->name, head->next->name);
325. e->distance = head->length;
326. e->line\_number = line;
327. e->next = G->Stations[index].firstedge;
328. G->Stations[index].firstedge = e;
329. index = locate(G, head->next->name);
330. e = (EdgeNode \*)malloc(sizeof(EdgeNode));
331. strcpy(e->name, head->name);
332. e->distance = head->length;
333. e->line\_number = line;
334. e->next = G->Stations[index].firstedge;
335. G->Stations[index].firstedge = e;
336. G->numEdges += 2;
337. head = head->next;
338. }
339. }
340. void Add\_station(MetroGraph \*G, char pre\_station[], char cur\_station[], float pre\_distance, float after\_distance, int line\_number)
341. {
342. *//首先通过对于线路起点至终点的遍历找到cur与after*
343. char Start[MAXLENGTH];         *//存放起始站名*
344. char Prev\_Station[MAXLENGTH];  *//用于普通站点（即非换乘站和非起始站）的已访问记录*
345. char After\_Station[MAXLENGTH]; *//用于存放下一个站点*
346. EdgeNode \*ptr = NULL;
347. strcpy(Start, start\_stations[line\_number]);
348. int index = locate(G, Start);
349. strcpy(Prev\_Station, Start);
350. strcpy(After\_Station, G->Stations[index].firstedge->name);
351. while (true)
352. {
353. if (strcmp(pre\_station, Prev\_Station) == 0)
354. break;
355. index = locate(G, After\_Station);
356. if (G->Stations[index].firstedge->next == NULL)
357. break;
358. ptr = Search\_same\_line(G->Stations[index].firstedge, line\_number);
359. if (strcmp(Prev\_Station, ptr->name) == 0) *//如果访问过直接从ptr开始寻找*
360. {
361. ptr = ptr->next;
362. ptr = Search\_same\_line(ptr, line\_number);
363. }
364. strcpy(Prev\_Station, After\_Station);
365. strcpy(After\_Station, ptr->name);
366. }
367. *//以上代码只为找到pre结点在原线路的下一结点*
368. strcpy(G->Stations[G->numNodes++].Station\_Name, cur\_station);
369. index = locate(G, Prev\_Station);
370. ptr = G->Stations[index].firstedge;
371. while (ptr != NULL)
372. {
373. if (strcmp(ptr->name, After\_Station) == 0)
374. {
375. strcpy(ptr->name, cur\_station);
376. ptr->distance = pre\_distance;
377. break;
378. }
379. ptr = ptr->next;
380. }
381. index = locate(G, After\_Station);
382. ptr = G->Stations[index].firstedge;
383. while (ptr != NULL)
384. {
385. if (strcmp(ptr->name, Prev\_Station) == 0)
386. {
387. strcpy(ptr->name, cur\_station);
388. ptr->distance = after\_distance;
389. break;
390. }
391. ptr = ptr->next;
392. }
393. Add\_EdgeNode(G, locate(G, cur\_station), Prev\_Station, line\_number, pre\_distance);
394. Add\_EdgeNode(G, locate(G, cur\_station), After\_Station, line\_number, after\_distance);
395. G->numEdges += 2;
396. }
397. void Del\_station(MetroGraph \*G, char old\_station[])
398. {
399. int flag = 0;                      *//普通站点flag=0；起始站点flag=1；换乘站点和普通站点的处理方法一致；*
400. int index;                         *//用于索引*
401. int visited[MAXLINES] = {0};       *//用于标记已经访问过的线路*
402. EdgeNode \*ptr = NULL;              *//用于遍历邻接点*
403. for (int i = 0; i < MAXLINES; i++) *//寻找起始站点*
404. {
405. if (start\_stations[i][0] == 0)
406. continue;
407. if (strcmp(old\_station, start\_stations[i]) == 0)
408. {
409. flag = 1;
410. break;
411. }
412. }
413. *//以上部分用于设置flag以体现不同类型的站点*
414. *//删除之后会产生新的邻接结点，需要连接起来*
415. index = locate(G, old\_station);
416. if (flag == 0)
417. {
418. char adj\_stations[MAX\_ADJ][MAXLENGTH] = {0};
419. ptr = G->Stations[index].firstedge;
420. strcpy(G->Stations[index].Station\_Name, "\0"); *//在顶点表结点中删除该结点*
421. for (int i = 0; i < MAX\_ADJ && ptr; i++)
422. {
423. float distance2;
424. strcpy(adj\_stations[i], ptr->name);
425. index = locate(G, adj\_stations[i]);
426. Del\_EdgeNode(G, index, old\_station, &distance2);
427. G->numEdges -= 2;
428. if (!visited[ptr->line\_number]) *//对于未访问过的进行新边的添加*
429. {
430. EdgeNode \*new\_adj\_station = Find\_same\_line(ptr, ptr->line\_number);
431. float Sum\_of\_Distance = ptr->distance;
432. Sum\_of\_Distance += distance2;
433. Add\_EdgeNode(G, index, new\_adj\_station->name, new\_adj\_station->line\_number, Sum\_of\_Distance);
434. Add\_EdgeNode(G, locate(G, new\_adj\_station->name), ptr->name, ptr->line\_number, Sum\_of\_Distance);
435. G->numEdges += 2;
436. visited[ptr->line\_number] = 1;
437. }
438. ptr = ptr->next;
439. }
440. }
441. else *//删除起始结点，并且不需要进行连接（一般来说是这样的）*
442. {
443. for (int i = 0; i < count; i++)
444. {
445. if (strcmp(old\_station, start\_stations[i]) == 0)
446. {
447. strcpy(start\_stations[i], G->Stations[index].firstedge->name);
448. break;
449. }
450. }
451. index = locate(G, G->Stations[index].firstedge->name);
452. Del\_EdgeNode(G, index, old\_station, NULL);
453. G->numEdges -= 2;
454. }
455. G->numNodes--;
456. }
457. void Add\_EdgeNode(MetroGraph \*G, int index, char station\_name[], int Line\_number, float Distance)
458. {
459. EdgeNode \*e = (EdgeNode \*)malloc(sizeof(EdgeNode));
460. e->line\_number = Line\_number;
461. e->distance = Distance;
462. strcpy(e->name, station\_name);
463. e->next = G->Stations[index].firstedge;
464. G->Stations[index].firstedge = e;
465. }
466. void Del\_EdgeNode(MetroGraph \*G, int index, char station\_name[], float \*after\_distance)
467. {
468. EdgeNode \*cur = G->Stations[index].firstedge;
469. EdgeNode \*pre = NULL;
470. while (cur != NULL)
471. {
472. if (strcmp(cur->name, station\_name) == 0)
473. {
474. if (pre == NULL)
475. G->Stations[index].firstedge = cur->next;
476. else
477. pre->next = cur->next;
478. if (after\_distance)
479. \*after\_distance = cur->distance;
480. free(cur);
481. break;
482. }
483. pre = cur;
484. cur = cur->next;
485. }
486. }
487. bool existed(MetroGraph \*G, char name[])
488. {
489. for (int i = 0; i < MAXSIZE; i++)
490. {
491. if (strcmp(G->Stations[i].Station\_Name, name) == 0)
492. return true; *//如果存在相同的站名就返回true*
493. }
494. return false;
495. }
496. int locate(MetroGraph \*G, char name[])
497. {
498. for (int i = 0; i < G->numNodes; i++)
499. {
500. if (strcmp(G->Stations[i].Station\_Name, name) == 0)
501. return i;
502. }
503. return -1;
504. }
505. EdgeNode \*Find\_same\_line(EdgeNode \*head, int Line\_number)
506. {
507. head = head->next;
508. while (head)
509. {
510. if (head->line\_number == Line\_number)
511. return head;
512. head = head->next;
513. }
514. return NULL;
515. }
516. EdgeNode \*Search\_same\_line(EdgeNode \*head, int Line\_number)
517. {
518. while (head)
519. {
520. if (head->line\_number == Line\_number)
521. return head;
522. head = head->next;
523. }
524. return NULL;
525. }
526. void print\_line(MetroGraph \*G, int Line\_number)
527. {
528. printf("%d ", Line\_number);
529. *//在起始站数组中寻找该起始点*
530. char Start[MAXLENGTH];         *//存放起始站名*
531. char Prev\_Station[MAXLENGTH];  *//用于普通站点（即非换乘站和非起始站）的已访问记录*
532. char After\_Station[MAXLENGTH]; *//用于存放下一个站点*
533. EdgeNode \*ptr = NULL;
534. strcpy(Start, start\_stations[Line\_number]);
535. int index = locate(G, Start);
536. float After\_length = G->Stations[index].firstedge->distance;
537. printf("%s %.2f ", Start, After\_length);
538. strcpy(Prev\_Station, Start);
539. strcpy(After\_Station, G->Stations[index].firstedge->name);
540. while (true)
541. {
542. index = locate(G, After\_Station);
543. if (G->Stations[index].firstedge->next == NULL)
544. break;
545. ptr = Search\_same\_line(G->Stations[index].firstedge, Line\_number);
546. if (strcmp(Prev\_Station, ptr->name) == 0) *//如果访问过直接从ptr开始寻找*
547. {
548. ptr = ptr->next;
549. ptr = Search\_same\_line(ptr, Line\_number);
550. }
551. strcpy(Prev\_Station, After\_Station);
552. if(ptr == NULL)
553. break;
554. After\_length = ptr->distance;
555. printf("%s %.2f ", After\_Station, After\_length);
556. strcpy(After\_Station, ptr->name);
557. }
558. printf("%s", After\_Station);
559. }
560. void ShortestPath\_Dijkstra(MetroGraph \*G, int v0, int Patharc[], float ShortPathTable[])
561. {
562. int v,w,k;
563. float min;
564. int final[MAXSIZE] = {0};  *//final[w]=1表示求得顶点v0至vw的最短路径（因为存在不可达的点但本题无）*
565. *//初始化部分*
566. for(v = 0; v<G->numNodes; v++)
567. {
568. Patharc[v] = -1;
569. ShortPathTable[v] = INFINITY;
570. }
571. *//另外需要在最短路径数组里初始化与v0邻接的点的权值*
572. ShortPathTable[v0] = 0;  *//该点到自己的距离为0*
573. final[v0] = 1;  *//该源点不需要被访问*
574. EdgeNode \*ptr = G->Stations[v0].firstedge;
575. while(ptr!=NULL)
576. {
577. int index = locate(G,ptr->name);
578. ShortPathTable[index] = ptr->distance;
579. ptr = ptr->next;
580. }
581. for(v = 1; v<G->numNodes; v++)
582. {
583. min = INFINITY;
584. for(w = 0; w<G->numNodes; w++)
585. {
586. if(!final[w] && ShortPathTable[w]<min)
587. {
588. k = w;
589. min = ShortPathTable[w];
590. }
591. }
592. final[k] = 1;
593. ptr = G->Stations[k].firstedge;
594. while(ptr!=NULL)
595. {
596. int index = locate(G,ptr->name);
597. if(!final[index] && (min+ptr->distance)<ShortPathTable[index])
598. {
599. ShortPathTable[index] = min + ptr->distance;
600. Patharc[index] = k;
601. }
602. ptr = ptr->next;
603. }
604. }
605. }
606. void Print\_ShortestPath(MetroGraph \*G, int Vend, float Sum, const int Patharc[], const float ShortPathTable[])
607. {
608. if(Patharc[Vend] == -1)
609. {
610. printf("%.2f %s ",Sum,G->Stations[Vend].Station\_Name);
611. return;
612. }
613. int index = Patharc[Vend];
614. float distance = Sum - ShortPathTable[index];
615. Print\_ShortestPath(G,index,ShortPathTable[index],Patharc,ShortPathTable);
616. printf("%.2f %s ",distance,G->Stations[Vend].Station\_Name);
617. }