华中科技大学

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： 2022.11.15

网络空间安全学院

目 录

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 数据结构

首先给出相关结构体的声明与定义（三题共用该数据结构）：

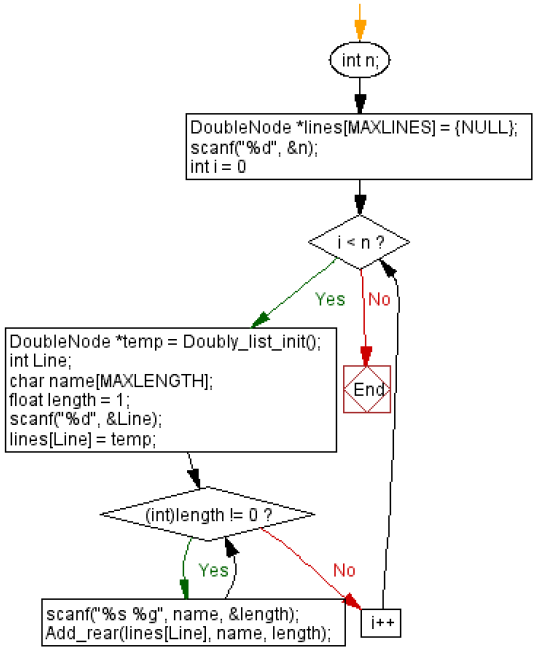
```
typedef struct DoubleNode /*双向链表用于存储线路*/
{
    char name[MAXLENGTH];    //站点名称
    float length;            //与下一个站点的距离
    struct DoubleNode *next; //后继
    struct DoubleNode *prev; //前驱
} DoubleNode;

typedef struct EdgeNode /*边表结点*/
{
    int line_number;       //地铁线路号
    char name[MAXLENGTH];  //用于存放站点的名称
    float distance;        //用于存储站点之间的距离
    struct EdgeNode *next; //链域，指向下一个邻接点
} EdgeNode;

typedef struct Station /*顶点表结点*/
{
    char Station_Name[MAXLENGTH]; //顶点域用于存储顶点的信息
    EdgeNode *firstedge;          //边表头指针
} Station, Stations_List[MAXSIZE];

typedef struct /*地铁线路图的定义*/
{
    Stations_List Stations; //图的点集，即站点信息
    int numNodes, numEdges; //结点的个数和边的个数
} MetroGraph;
```

1.4 算法设计

1.4.1 双向链表的创建与输入的读取

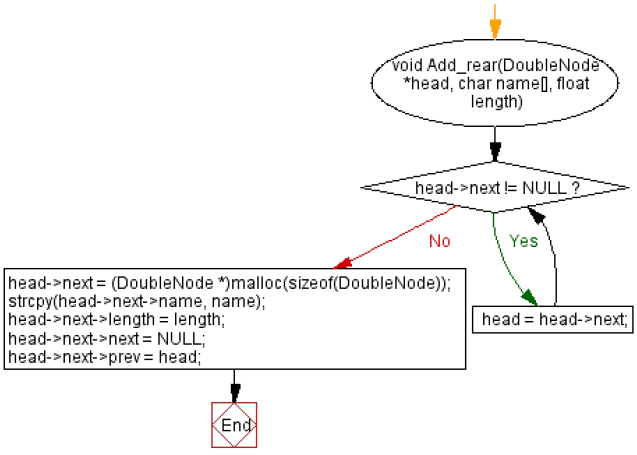
利用一个指针数组 lines 指向 n 条双向链表的头节点，且用数组的下标天然的代表线路号 Line，每条双向链表均使用了一个虚拟头节点以便创建后序的结点，而结点的加入则使用 Add_rear 函数把新建立的结点加入到双向链表的尾部。

（1）数据的读入和双向链表的创建



（2）相关函数的实现流程图

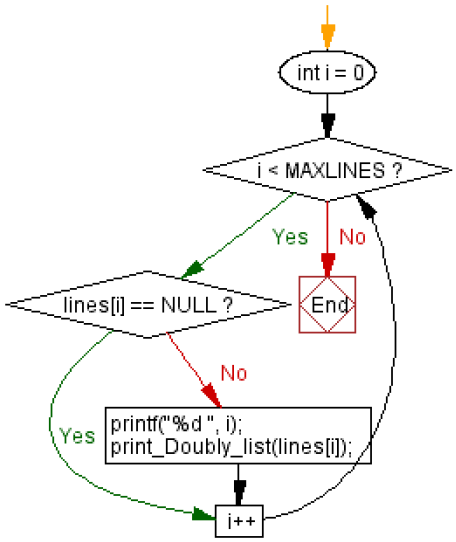
a）Add_rear 函数的实现（双向链表尾插法）



1.4.2 根据双向链表的打印某条特定的线路

首先遍历 lines 数组，如果是 NULL 则跳过，否则则根据该指针调用 print_Doubly_list 函数打印相应的双向链表。在程序的最后调用 Destroy_Doubly_list 函数销毁整个双向链表。

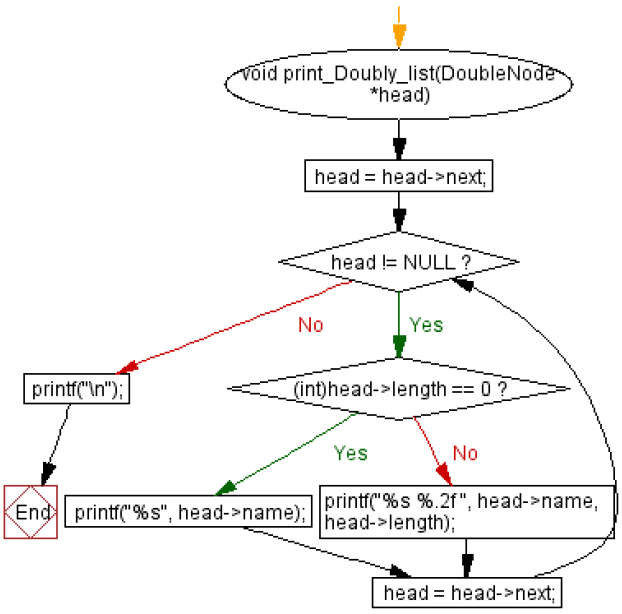
（1）打印所有的线路



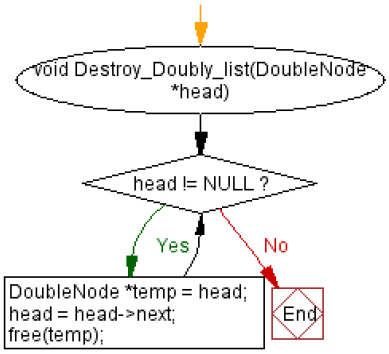
（2）相关函数的实现流程图

a）print_Doubly_list 函数的实现

该算法的本质就是一个遍历双向链表并输出的过程。



b) Destroy_Doubly_list 函数的实现

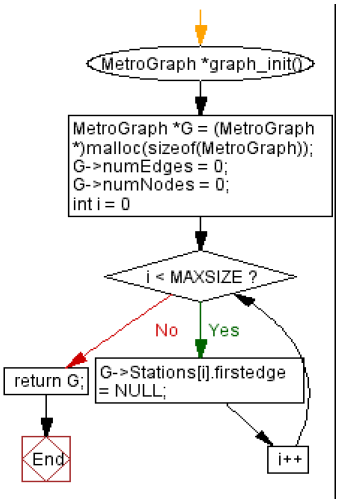


1.4.3 根据双向链表的数据创建邻接图

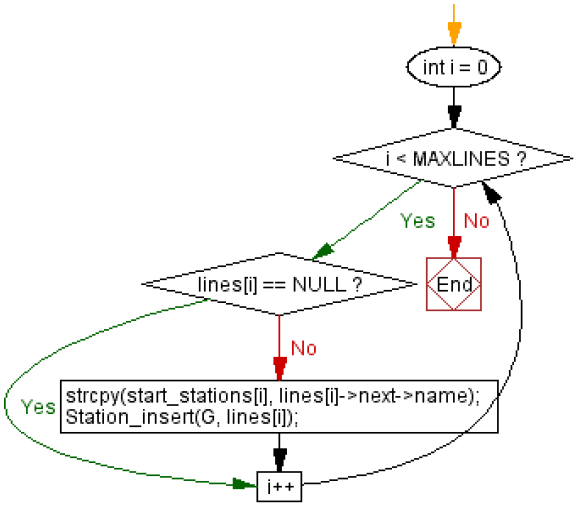
图的邻接表创建包括头节点表和边表结点的创建。

(1) 邻接表的初始化

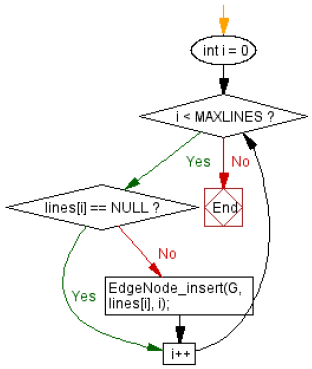
调用 graph_init 函数对图进行相关的初始化操作



(2) 邻接表的顶点表的创建

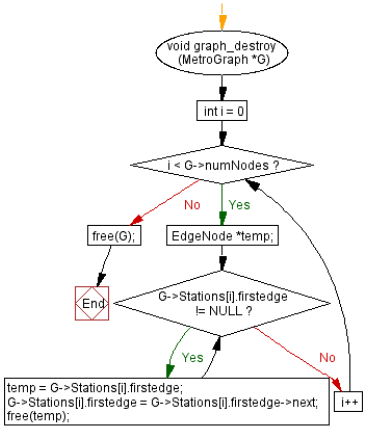


(3) 邻接表的边表结点的创建



(4) 图的销毁

调用 graph_destroy 函数销毁图：



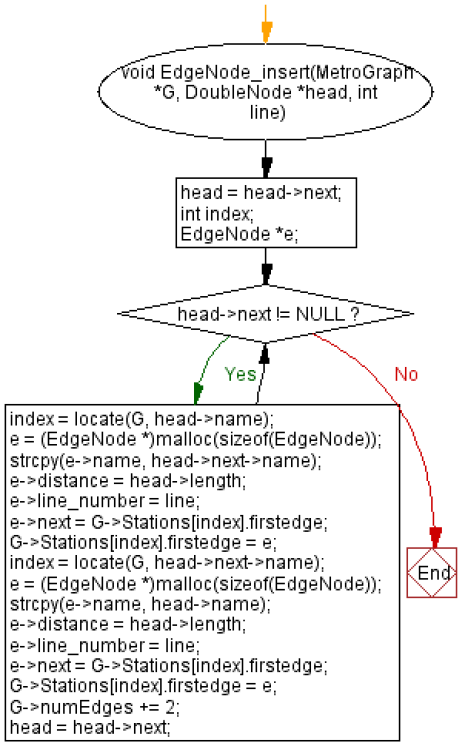
(5) 相关函数的实现

a) Station_insert 函数实现

对于已经在头结点表里存过的站点（即换乘站）该函数需要判断这个站点是否已经被存过（该功能封装成函数 existed），如果存过则不用在头结点表中插入该站点。

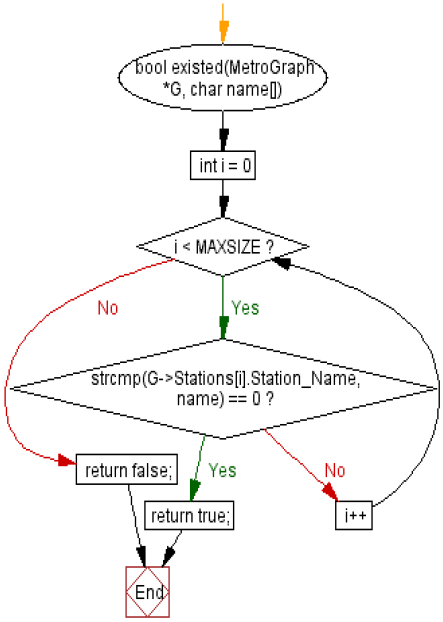
b) Edgenode_insert 函数实现

需要注意的是该图为有向图，所以对于一个站点需要将其前后站点（如果有的话）全部存入该结点的边表中，如何找到这个站点在头结点表中的位置呢，既然头结点表本质上是一个数组，所以我直接利用数组的下标去定位这个站点，本质上就是一个遍历比对站点名称的过程（该功能封装成函数 locate）。



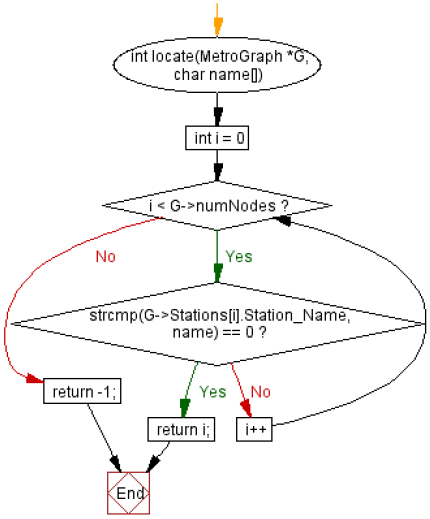
c）existed 函数实现

通过传入一个目标站点名称，然后遍历比较的过程判断是否存在该结点。



d）locate 函数实现

传入一个站点名称，返回该站点在表头结点表数组的索引。本质就是一个遍历比较的过程。



1.5 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

1.6 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 1-2 所示。

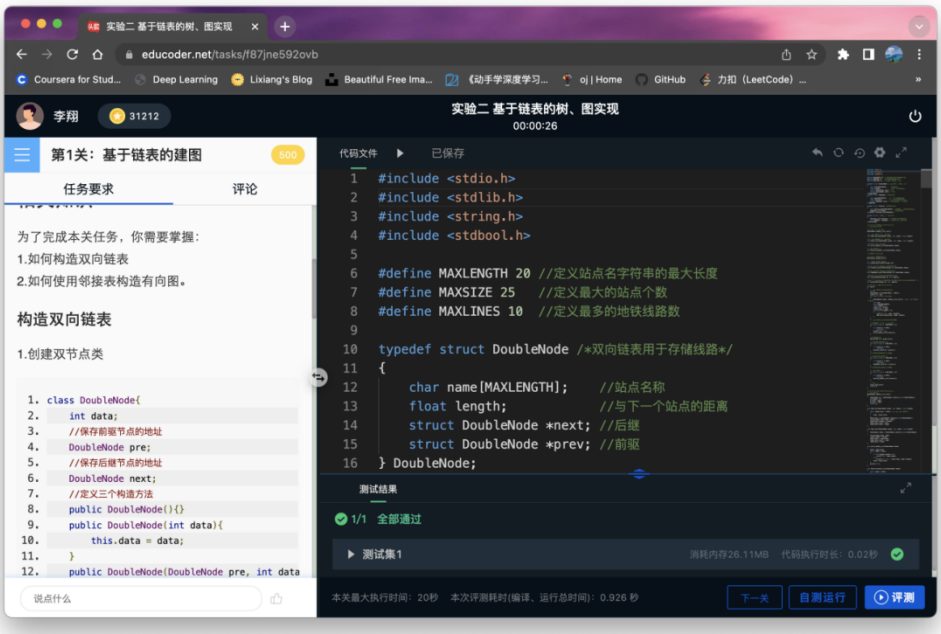


图 1-2 通过所有测试

1.7 复杂度分析

(1) 双向链表的创建与数据的读取过程

设线路的总条数为 n, 每条线路的站点数数据规模量为 m，则在该过程中利用了一个 for 循环按照线路号存储相应的线路，同时在内部有个 while 循环 m 次用于读取一条线路中的所有站点。故算法的时间复杂度为$O(m*n)$,而空间复杂度则开辟了了 n 条 m 长度的双向链表空间，则空间复杂度也为$O(m*n)$。

(2) 根据双向链表打印所有线路的过程

本质上是遍历 n 条 m 长的双向链表，故算法的时间复杂度为$O(m*n)$,而该算法并没有新开辟空间，则空间复杂度仍$O(1)$。

(3) 邻接表创建的过程

邻接表的创建过程中，对于 n 个顶点和 e 条边且需要查找来插入结点的算法来说，正常情况下的算法间复杂度为$O(n*e)$，但在本题中，其实我构建头结点表的时候，由于换乘站的存在，所以我需要确定在头结点表中该站是否已经被存过了，这个过程是一个遍历并且比较的过程，故时间复杂度为$O(n^2*e)$，空间复杂度显然为$O(n+e)$。

1.8 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。

1.9 实验小结

通过该次实验我学会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，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实现邻接图中对于相应结点的添加和删除。

2.3 数据结构

与 1.3 的数据结构相同。

2.4 算法设计

本题的要求是在邻接图的基础上进行一系列的增加和删除的操作，首先读取数据，第一步是对数据进行分析，是增加还是删除，以及所操作的结点是否有效（例如站点是否存在），这些条件的判断在主程序中区分开来并输出相应的结果和调用相应的函数，分离情况的过程较为的简单在此不多赘述。

(1) 增加站点（Add_station 函数）

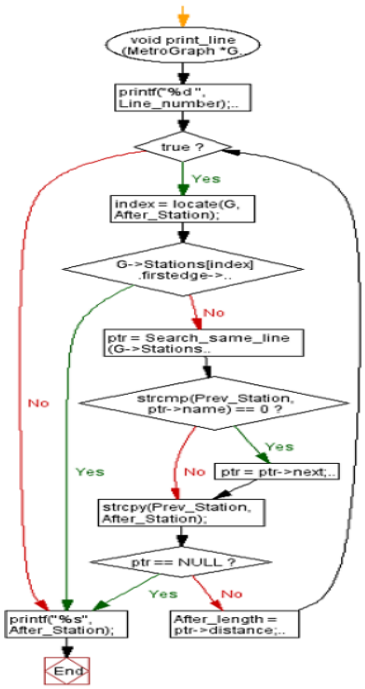
本算法需要实现的功能是结点的增加，在图的邻接表中，如果想要先增加结点的话，那么需要首先找到需要插入的位置并且断开相应的邻接表的连接，后在邻接表中添加该结点。这两个功能分别封装成了 Add_EdgeNode 函数和 Del_EdgeNode 函数，其本质上就是一个对于单向链表的结点增加或者删除的过程。并且，此处的增加并不需要遍历，而是直接使用头插法加入即可。

(2) 删除站点（Del_station 函数）

删除之后需要建立新的连接，删除的情况比增加更为的复杂在于如果想要删除的是换乘站，那么就需要新建立多对新的邻接关系并且还需要更改新建立的邻接关系之间的距离数值。依据题意，其实一个换乘站最多也就连接了两条线路，但是为了该算法的普适性，我选择了使用 visited 数组记录已经连接过的线路。

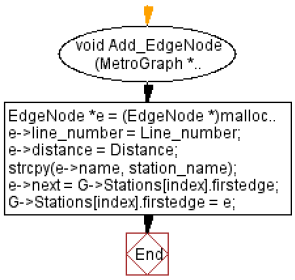
(3) 根据线路号打印整条线路

在边表结点的设计之初就在其中加入了 line_number 数据项以区分不同的线路站点，并且定义了一个名为 Start_Station 的数组以存储每条线路的起始站的站名（数组的下标相应的线路）。然后就是一个对于邻接表表示的图的遍历的过程，只不过只有满足线路号相同时才会输出该结点的信息。结束的条件是遇到 distance=0 的情况。

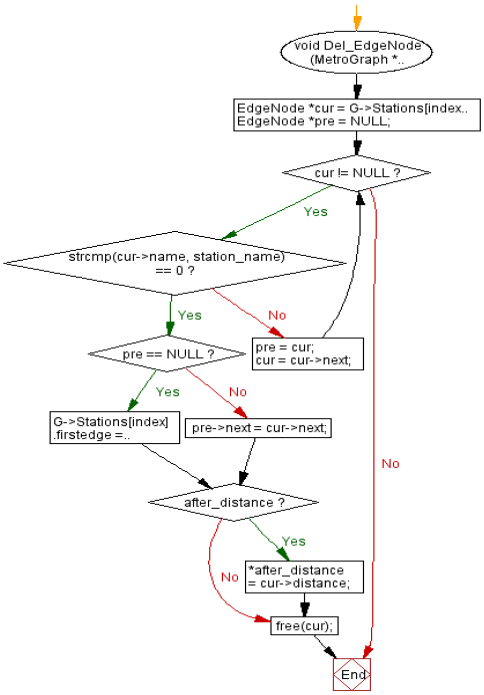


（4）相关函数的实现

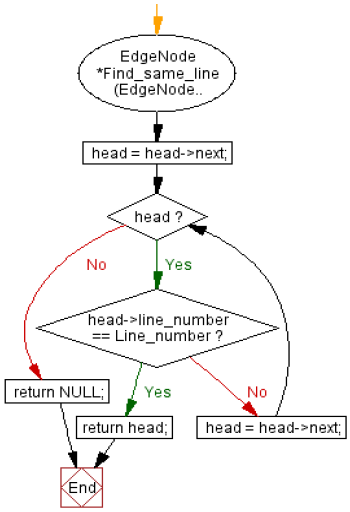
a. Add_EdgeNode



b. Del_EdgeNode



c. Find_same_line



2.5 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

2.6 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 2-2 所示。

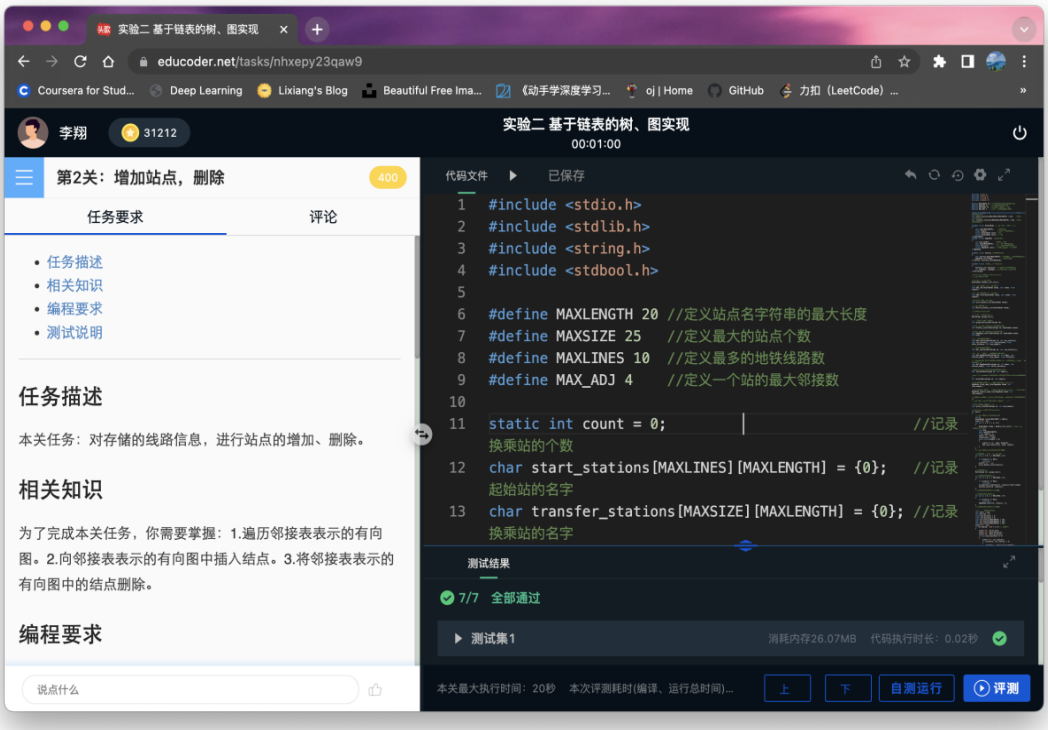


图 2-2 通过所有测试

2.7 复杂度分析

在图中增加结点和删除结点的过程，需要遍历去定位需要增加或者删除的位置，其时间复杂度为$O(n)$，而所增加减少的空间只是常数个内存空间，故空间复杂度为$O(1)$。

2.8 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

2.9 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 数据结构

与 1.3 的数据结构相同。

3.4 算法设计

该题目的本质是就是利用迪杰斯特拉算法中利用到的贪心策略去编写算法，我把该算法封装成了函数 ShortestPath_Dijkstra，由于需要进行最短路径的输出，又需要根据路径表去输出最短路径上的结点，该功能封装成 Print_ShortestPath。

3.4.1 迪杰斯特拉算法的实现

该算法采用贪心策略，从起点出发，散发路径，每次选取最短的一条，并且利用该条最短的线路（作为一个最优的）继续发散，并且与已有的路径对比及时更改成最短的路径。而且需要注意的是，本题是是以邻接表作为图的存储结构，所以在已经掌握邻接矩阵的基础上自己写出了邻接表的版本。

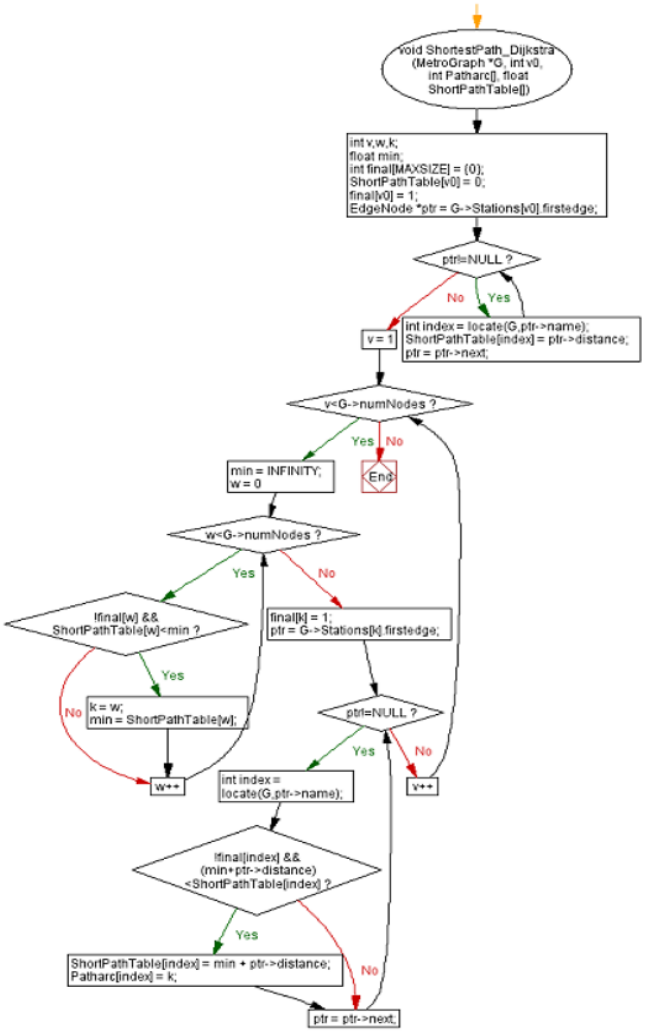
(1) 调用 ShortestPath_Dijkstra 函数求得 Patharc 和 ShortestPathTable 数组。

算法基本思想：

1. 通过Dijkstra计算图G中的最短路径时，需要指定起点s(即从顶点s开始计算)。

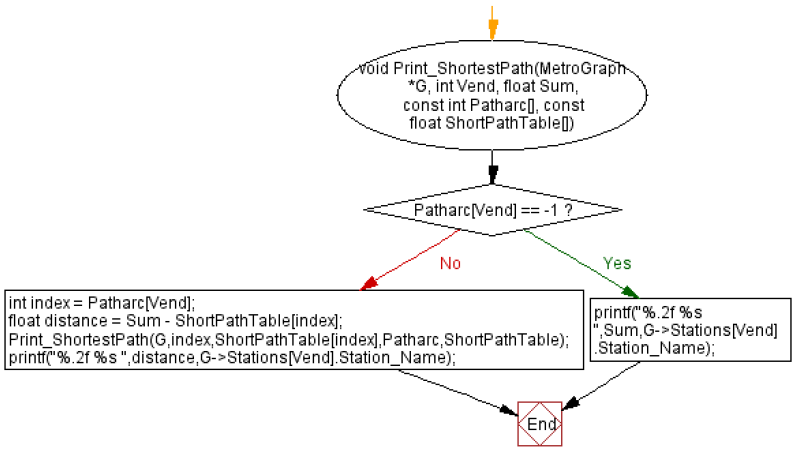
2. 此外，引进两个集合S和U。S的作用是记录已求出最短路径的顶点(以及相应的最短路径长度)，而U则是记录还未求出最短路径的顶点(以及该顶点到起点s的距离)。

3. 初始时，S中只有起点s；U中是除s之外的顶点，并且U中顶点的路径是”起点s到该顶点的路径”。然后，从U中找出路径最短的顶点，并将其加入到S中；接着，更新U中的顶点和顶点对应的路径。 然后，再从U中找出路径最短的顶点，并将其加入到S中；接着，更新U中的顶点和顶点对应的路径。重复该操作，直到遍历完所有顶点。



（2）调用 Print_ShortestPath 函数打印最短路径以及距离

由于迪杰斯特拉得到的 ShortestPathTable 数组中存的是源点到达该点的路径上的上一个结点的索引值，故需要倒推出最短路径，采用递归压栈的方式，进行后序的输出操作，可以实现把线路倒序输出的算法。



3.5 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

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 3-2 所示。

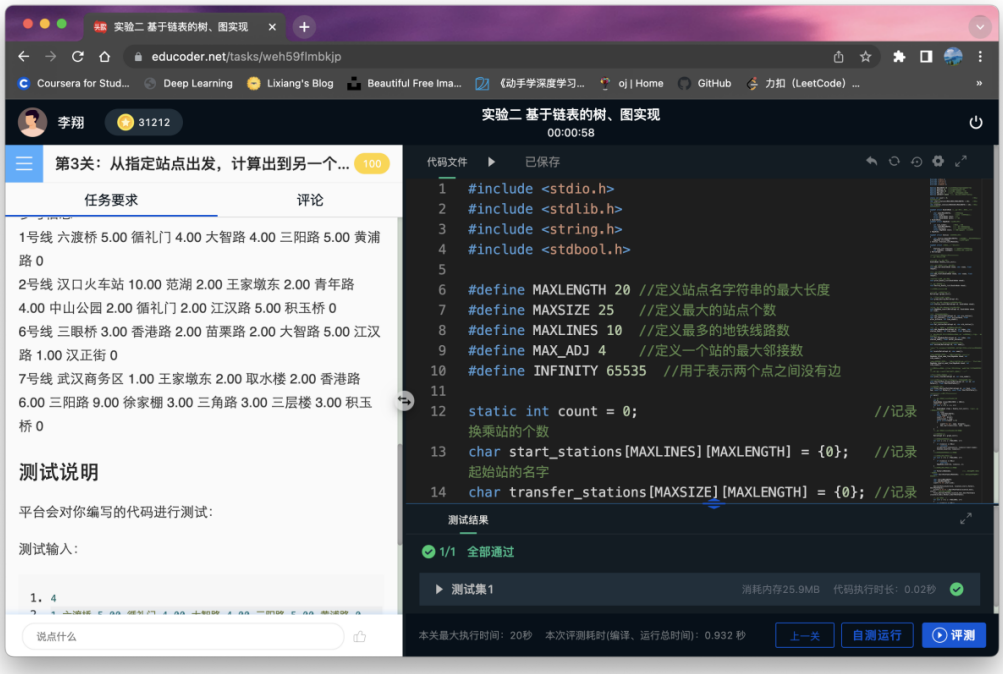


图 3-2 通过所有测试

3.7 复杂度分析

本题所用的算法即为迪杰斯特拉算法，由源点出发求到所有点的最短距离，从算法中可以显而易见的看出其本质为一个嵌套的循环，故算法的时间复杂度为$O(n^2)$，在空间方面，其开辟了两个数组，分别是最短路径值数组，和一个路径索引的数组，长度均为 n，所以空间复杂度为$O(n)$。

输出过程中，所用到的是一个线性递归的过程，时间复杂度为$O(n)$，而递归过程中是一个压栈的过程，空间复杂度与递归的深度有关，同样为$O(n)$。

3.8 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。

3.9 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] 程杰.大话数据结构. 清华大学出版社

[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

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#define MAXLENGTH 20 //定义站点名字符串的最大长度
#define MAXSIZE 25   //定义最大的站点个数
#define MAXLINES 10  //定义最多的地铁线路数
#define MAX_ADJ 4    //定义一个站的最大邻接数
#define INFINITY 65535  //用于表示两个点之间没有边

static int count = 0;                                   //记录换乘站的个数
char start_stations[MAXLINES][MAXLENGTH] = {0};   //记录起始站的名字
char transfer_stations[MAXSIZE][MAXLENGTH] = {0}; //记录换乘站的名字

typedef struct DoubleNode /*双向链表用于存储线路*/
{
    char name[MAXLENGTH];     //站点名称
    float length;             //与下一个站点的距离
    struct DoubleNode *next; //后继
    struct DoubleNode *prev; //前驱
} DoubleNode;
typedef struct EdgeNode /*边表结点*/
{
    int line_number;        //地铁线路号
    char name[MAXLENGTH];  //用于存放站点的名称
    float distance;         //用于存储站点之间的距离
    struct EdgeNode *next; //链域，指向下一个邻接点
} EdgeNode;

typedef struct Station /*顶点表结点*/
```

```
{
    char Station_Name[MAXLENGTH]; //顶点域用于存储顶点的信息
    EdgeNode *firstedge;          //边表头指针
} Station, Stations_List[MAXSIZE];

typedef struct /*地铁线路图的定义*/
{
    Stations_List Stations; //图的点集，即站点信息
    int numNodes, numEdges; //结点的个数和边的个数
} MetroGraph;

/**********函数声明部分**********/
//双向链表相关函数

/*初始化一个双向链表*/
DoubleNode *Doubly_list_init();

/*在双向链表的尾部添加新结点*/
void Add_rear(DoubleNode *head, char *name, float length);

/*在双向链表的首部添加新结点*/
void Add_front(DoubleNode *head, char *name, float length);

/*从头至尾打印双向链表*/
void print_Doubly_list(DoubleNode *head);

/*双向链表的摧毁*/
void Destroy_Doubly_list(DoubleNode *head);

//邻接表创建相关函数

/*初始化一个地铁有向图*/
MetroGraph *graph_init();

/*销毁已经创建的图结构*/
void graph_destroy(MetroGraph *G);

/*依据双向链表中的信息创建顶点表结点*/
```

```
70. void Station_insert(MetroGraph *G, DoubleNode *head);
71.
72. /*依据双向链表中的信息创建边表结点*/
73. void EdgeNode_insert(MetroGraph *G, DoubleNode *head, int line);
74.
75. /*添加有效的地铁站点*/
76. void Add_station(MetroGraph *G, char pre_station[], char cur_station[], float pre_distance, float after_distance, int line_number);
77.
78. /*删除有效的地铁站点*/
79. void Del_station(MetroGraph *G, char old_station[]);
80.
81. /*添加由于删除或增加新产生的边表结点*/
82. void Add_EdgeNode(MetroGraph *G, int index, char station_name[], int Line_number, float Distance);
83.
84. /*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*/
85. void Del_EdgeNode(MetroGraph *G, int index, char station_name[], float *after_distance);
86.
87. /*判断station 是否已经存入了顶点表结点*/
88. bool existed(MetroGraph *G, char name[]);
89.
90. /*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*/
91. int locate(MetroGraph *G, char name[]);
92.
93. /*查找顶点的邻接点中线路号相同的元素,配对作用*/
94. EdgeNode *Find_same_line(EdgeNode *head, int Line_number);
95.
96. /*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*/
97. EdgeNode *Search_same_line(EdgeNode *head, int Line_number);
98.
99. //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
```

```
100.    //主要是因为程序太长了，要改逻辑的话过于复杂
101.
102.    /*根据邻接表打印地铁线路*/
103.    void print_line(MetroGraph *G, int Line_number);
104.
105.    /*利用迪杰斯特拉算法计算 v0 到其他所有点的最短路径*/
106.    void ShortestPath_Dijkstra(MetroGraph *G, int v0, int Pa
tharc[], float ShortPathTable[]);
107.
108.    /*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*/
109.    void Print_ShortestPath(MetroGraph *G, int Vend, float S
um, const int Patharc[], const float ShortPathTable[]);
110.
111.    /**********主函数部分**********/
112.    int main()
113.    {
114.        //双向链表的创建与输入的读取
115.        int n;
116.        DoubleNode *lines[MAXLINES] = {NULL};
117.        scanf("%d", &n);
118.        for (int i = 0; i < n; i++)
119.        {
120.            DoubleNode *temp = Doubly_list_init(); //新建双向
链表的虚拟头节点
121.            int Line;
122.            char name[MAXLENGTH];
123.            float length = 1;
124.            scanf("%d", &Line);
125.            lines[Line] = temp;
126.            while ((int)length != 0)
127.            {
128.                scanf("%s %g", name, &length);
129.                Add_rear(lines[Line], name, length);
130.            }
131.        }
132.        //双向链表的创建与输入的读取(成功)
133.
```

```
134.        /*****第一关代码开始*****/
135.        //打印地铁线路(即一个双向链表)
136.        for (int i = 0; i < MAXLINES; i++)
137.        {
138.            if (lines[i] == NULL)
139.                continue;
140.            printf("%d ", i);
141.            print_Doubly_list(lines[i]);
142.        }
143.
144.        //邻接表的创建
145.        MetroGraph *G = graph_init();
146.
147.        //邻接表的顶点表的创建
148.        for (int i = 0; i < MAXLINES; i++)
149.        {
150.            if (lines[i] == NULL)
151.                continue;
152.            strcpy(start_stations[i], lines[i]->next->name);
153.            Station_insert(G, lines[i]);
154.        }
155.        //邻接表的顶点表的创建(成功)
156.
157.        //邻接表边表结点的创建
158.        for (int i = 0; i < MAXLINES; i++)
159.        {
160.            if (lines[i] == NULL)
161.                continue;
162.            EdgeNode_insert(G, lines[i], i);
163.        }
164.        //邻接表边表结点的创建(成功)
165.        /*****第一关代码结束*****/
166.
167.        /*****第二关代码开始*****/
168.        //添加和删除操作的解析
169.        char op[7] = {0};
170.        int line_num = 0;
171.        float pre_distance = 0;
```

```
172.        float aft_distance = 0;
173.        char pre_station[MAXLENGTH] = {0};
174.        char cur_station[MAXLENGTH] = {0};
175.        char aft_station[MAXLENGTH] = {0};
176.        scanf("%s", op);
177.        if (strcmp(op, "add") == 0) // add 操作
178.        {
179.            scanf("%d", &line_num);
180.            scanf("%g", &pre_distance);
181.            scanf("%g", &aft_distance);
182.            if ((int)pre_distance == 0)
183.            {
184.                scanf("%s", cur_station);
185.                if (locate(G, cur_station) > 0)
186.                {
187.                    printf("增加失败，已有同名站点");
188.                }
189.                else
190.                {
191.                    strcpy(aft_station, start_stations[line_
num]);
192.                    strcpy(start_stations[line_num], cur_sta
tion);
193.                    Add_EdgeNode(G, locate(G, aft_station),
cur_station, line_num, aft_distance);
194.                    strcpy(G->Stations[G->numNodes++].Statio
n_Name, cur_station);
195.                    Add_EdgeNode(G, locate(G, cur_station),
aft_station, line_num, aft_distance);
196.                    print_line(G, line_num);
197.                }
198.            }
199.            else
200.            {
201.                scanf("%s", pre_station);
202.                scanf("%s", cur_station);
203.                if (locate(G, pre_station) == -1)
204.                {
```

```
205.                     printf("增加失败，没有与输入的增加位置前一站点同名的站点");
206.                 }
207.                 else if (locate(G, cur_station) > 0)
208.                 {
209.                     printf("增加失败，已有同名站点");
210.                 }
211.                 else
212.                 {
213.                     Add_station(G, pre_station, cur_station, pre_distance, aft_distance, line_num);
214.                     print_line(G, line_num);
215.                 }
216.             }
217.         }
218.         else
219.         {
220.             scanf("%d", &line_num);
221.             scanf("%s", cur_station);
222.             if (locate(G, cur_station) == -1)
223.             {
224.                 printf("删除失败，没有同名站点");
225.             }
226.             else
227.             {
228.                 Del_station(G, cur_station);
229.                 print_line(G, line_num);
230.             }
231.         }
232.         /*****第二关代码结束*****/
233.
234.         /*****第三关代码开始*****/
235.         int Patharc[MAXSIZE];              //用于存储最短路径下标的数组
236.         float ShortPathTable[MAXSIZE];  //用于存储到各点最短路径的权值和
237.
238.         char start[MAXLENGTH];
```

```
    239.         char end[MAXLENGTH];
    240.         scanf("%s %s",start,end);
    241.
    242.         ShortestPath_Dijkstra(G, locate(G,start),Patharc,ShortPathTable);
    243.         printf("%.2f %s ",ShortPathTable[locate(G,end)],start);  //先打印路径最短
    244.         Print_ShortestPath(G,locate(G,end),ShortPathTable[locate(G,end)],Patharc,ShortPathTable);
    245.         /*****第三关代码结束*****/
    246.
    247.
    248.         //双向链表的销毁
    249.         for (int i = 0; i < MAXLINES; i++)
    250.         {
    251.             if (lines[i] == NULL)
    252.                 continue;
    253.             Destroy_Doubly_list(lines[i]);
    254.         }
    255.
    256.         //图的销毁
    257.         graph_destroy(G);
    258.         return 0;
    259.     }
    260.
    261.     /**********函数实现部分**********/
    262.     //双向链表相关
    263.     DoubleNode *Doubly_list_init()
    264.     {
    265.         DoubleNode *p = (DoubleNode *)malloc(sizeof(DoubleNode));
    266.         p->length = 0;
    267.         p->next = NULL;
    268.         p->prev = NULL;
    269.         return p;
    270.     }
    271.
```

```
272.  void Add_rear(DoubleNode *head, char name[], float lengt
h)
273.  {
274.      while (head->next != NULL) //定位到双向链表的末尾
275.      {
276.          head = head->next;
277.      }
278.      head->next = (DoubleNode *)malloc(sizeof(DoubleNode)
);
279.      strcpy(head->next->name, name);
280.      head->next->length = length;
281.      head->next->next = NULL;
282.      head->next->prev = head;
283.  }
284.
285.  void Add_front(DoubleNode *head, char *name, float lengt
h)
286.  {
287.      DoubleNode *temp = (DoubleNode *)malloc(sizeof(Doubl
eNode));
288.      strcpy(temp->name, name);
289.      temp->length = length;
290.      head->next = temp->next;
291.      temp->next->prev = temp;
292.      temp->prev = head;
293.  }
294.
295.  void print_Doubly_list(DoubleNode *head)
296.  {
297.      head = head->next;
298.      while (head != NULL)
299.      {
300.          if ((int)head->length == 0)
301.              printf("%s", head->name);
302.          else
303.              printf("%s %.2f ", head->name, head->length)
;
304.          head = head->next;
```

```
305.        }
306.        printf("\n");
307.    }
308.
309.    void Destroy_Doubly_list(DoubleNode *head)
310.    {
311.        while (head != NULL)
312.        {
313.            DoubleNode *temp = head;
314.            head = head->next;
315.            free(temp);
316.        }
317.    }
318.
319.    //有向图邻接表相关
320.    MetroGraph *graph_init()
321.    {
322.        MetroGraph *G = (MetroGraph *)malloc(sizeof(MetroGra
ph));
323.        G->numEdges = 0;
324.        G->numNodes = 0;
325.        for (int i = 0; i < MAXSIZE; i++)
326.        {
327.            G->Stations[i].firstedge = NULL;
328.        }
329.        return G;
330.    }
331.
332.    void graph_destroy(MetroGraph *G)
333.    {
334.        for (int i = 0; i < G->numNodes; i++)
335.        {
336.            EdgeNode *temp;
337.            while (G->Stations[i].firstedge != NULL)
338.            {
339.                temp = G->Stations[i].firstedge;
340.                G->Stations[i].firstedge = G->Stations[i].fi
rstedge->next;
```

```
341.                 free(temp);
342.             }
343.         }
344.         free(G);
345.     }
346.
347.     void Station_insert(MetroGraph *G, DoubleNode *head)
348.     {
349.         head = head->next; //跳过双向链表的虚拟头结点
350.         while (head != NULL)
351.         {
352.             if (!existed(G, head->name)) //如果站名在顶点表里不存在就加入
353.             {
354.                 strcpy(G->Stations[G->numNodes++].Station_Name, head->name);
355.             }
356.             else
357.             {
358.                 strcpy(transfer_stations[count++], head->name);
359.             }
360.             head = head->next;
361.         }
362.     }
363.
364.     void EdgeNode_insert(MetroGraph *G, DoubleNode *head, int line)
365.     {
366.         head = head->next;
367.         int index;   //用于定位Stations中的站点
368.         EdgeNode *e; //主要用于malloc作为一个临时的指针
369.         while (head->next != NULL)
370.         {
371.             index = locate(G, head->name);
372.             e = (EdgeNode *)malloc(sizeof(EdgeNode));
373.             strcpy(e->name, head->next->name);
374.             e->distance = head->length;
```

```
375.            e->line_number = line;
376.            e->next = G->Stations[index].firstedge;
377.            G->Stations[index].firstedge = e;
378.
379.            index = locate(G, head->next->name);
380.            e = (EdgeNode *)malloc(sizeof(EdgeNode));
381.            strcpy(e->name, head->name);
382.            e->distance = head->length;
383.            e->line_number = line;
384.            e->next = G->Stations[index].firstedge;
385.            G->Stations[index].firstedge = e;
386.            G->numEdges += 2;
387.            head = head->next;
388.        }
389.    }
390.
391.    void Add_station(MetroGraph *G, char pre_station[], char
 cur_station[], float pre_distance, float after_distance, int line_
number)
392.    {     //首先通过对于线路起点至终点的遍历找到cur 与after
393.        char Start[MAXLENGTH];          //存放起始站名
394.        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
395.        char After_Station[MAXLENGTH]; //用于存放下一个站点
396.        EdgeNode *ptr = NULL;
397.        strcpy(Start, start_stations[line_number]);
398.        int index = locate(G, Start);
399.        strcpy(Prev_Station, Start);
400.        strcpy(After_Station, G->Stations[index].firstedge->
name);
401.        while (true)
402.        {
403.            if (strcmp(pre_station, Prev_Station) == 0)
404.                break;
405.            index = locate(G, After_Station);
406.            if (G->Stations[index].firstedge->next == NULL)
407.                break;
```

```
408.            ptr = Search_same_line(G->Stations[index].firste
dge, line_number);
409.            if (strcmp(Prev_Station, ptr->name) == 0) //如果
访问过直接从 ptr 开始寻找
410.            {
411.                ptr = ptr->next;
412.                ptr = Search_same_line(ptr, line_number);
413.            }
414.            strcpy(Prev_Station, After_Station);
415.            strcpy(After_Station, ptr->name);
416.        }
417.        //以上代码只为找到 pre 结点在原线路的下一结点
418.        strcpy(G->Stations[G->numNodes++].Station_Name, cur_
station);
419.        index = locate(G, Prev_Station);
420.        ptr = G->Stations[index].firstedge;
421.        while (ptr != NULL)
422.        {
423.            if (strcmp(ptr->name, After_Station) == 0)
424.            {
425.                strcpy(ptr->name, cur_station);
426.                ptr->distance = pre_distance;
427.                break;
428.            }
429.            ptr = ptr->next;
430.        }
431.        index = locate(G, After_Station);
432.        ptr = G->Stations[index].firstedge;
433.        while (ptr != NULL)
434.        {
435.            if (strcmp(ptr->name, Prev_Station) == 0)
436.            {
437.                strcpy(ptr->name, cur_station);
438.                ptr->distance = after_distance;
439.                break;
440.            }
441.            ptr = ptr->next;
442.        }
```

```
443.        Add_EdgeNode(G, locate(G, cur_station), Prev_Station
, line_number, pre_distance);
444.        Add_EdgeNode(G, locate(G, cur_station), After_Statio
n, line_number, after_distance);
445.        G->numEdges += 2;
446.    }
447.
448.    void Del_station(MetroGraph *G, char old_station[])
449.    {
450.        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；
451.        int index;                          //用于索引
452.        int visited[MAXLINES] = {0};        //用于标记已经访问
过的线路
453.        EdgeNode *ptr = NULL;               //用于遍历邻接点
454.        for (int i = 0; i < MAXLINES; i++) //寻找起始站点
455.        {
456.            if (start_stations[i][0] == 0)
457.                continue;
458.            if (strcmp(old_station, start_stations[i]) == 0)
459.            {
460.                flag = 1;
461.                break;
462.            }
463.        }
464.        //以上部分用于设置flag 以体现不同类型的站点
465.        //删除之后会产生新的邻接结点，需要连接起来
466.        index = locate(G, old_station);
467.        if (flag == 0)
468.        {
469.            char adj_stations[MAX_ADJ][MAXLENGTH] = {0};
470.            ptr = G->Stations[index].firstedge;
471.            strcpy(G->Stations[index].Station_Name, "\0"); /
/在顶点表结点中删除该结点
472.            for (int i = 0; i < MAX_ADJ && ptr; i++)
473.            {
474.                float distance2;
475.                strcpy(adj_stations[i], ptr->name);
```

```
476.                index = locate(G, adj_stations[i]);
477.                Del_EdgeNode(G, index, old_station, &distanc
e2);
478.                G->numEdges -= 2;
479.                if (!visited[ptr->line_number]) //对于未访问
过的进行新边的添加
480.                {
481.                    EdgeNode *new_adj_station = Find_same_li
ne(ptr, ptr->line_number);
482.                    float Sum_of_Distance = ptr->distance;
483.                    Sum_of_Distance += distance2;
484.                    Add_EdgeNode(G, index, new_adj_station->
name, new_adj_station->line_number, Sum_of_Distance);
485.                    Add_EdgeNode(G, locate(G, new_adj_statio
n->name), ptr->name, ptr->line_number, Sum_of_Distance);
486.                    G->numEdges += 2;
487.                    visited[ptr->line_number] = 1;
488.                }
489.                ptr = ptr->next;
490.            }
491.        }
492.        else //删除起始结点，并且不需要进行连接（一般来说是这样
的）
493.        {
494.            for (int i = 0; i < count; i++)
495.            {
496.                if (strcmp(old_station, start_stations[i]) =
= 0)
497.                {
498.                    strcpy(start_stations[i], G->Stations[in
dex].firstedge->name);
499.                    break;
500.                }
501.            }
502.            index = locate(G, G->Stations[index].firstedge->
name);
503.            Del_EdgeNode(G, index, old_station, NULL);
504.            G->numEdges -= 2;
```

```
505.        }
506.        G->numNodes--;
507.    }
508.
509.    void Add_EdgeNode(MetroGraph *G, int index, char station
_name[], int Line_number, float Distance)
510.    {
511.        EdgeNode *e = (EdgeNode *)malloc(sizeof(EdgeNode));
512.        e->line_number = Line_number;
513.        e->distance = Distance;
514.        strcpy(e->name, station_name);
515.        e->next = G->Stations[index].firstedge;
516.        G->Stations[index].firstedge = e;
517.    }
518.
519.    void Del_EdgeNode(MetroGraph *G, int index, char station
_name[], float *after_distance)
520.    {     EdgeNode *cur = G->Stations[index].firstedge;
521.        EdgeNode *pre = NULL;
522.        while (cur != NULL)
523.        {
524.            if (strcmp(cur->name, station_name) == 0)
525.            {
526.                if (pre == NULL)
527.                    G->Stations[index].firstedge = cur->next
;
528.                else
529.                    pre->next = cur->next;
530.                if (after_distance)
531.                    *after_distance = cur->distance;
532.                free(cur);
533.                break;
534.            }
535.            pre = cur;
536.            cur = cur->next;
537.        }
538.    }
539.
```

```
bool existed(MetroGraph *G, char name[])
{
    for (int i = 0; i < MAXSIZE; i++)
    {
        if (strcmp(G->Stations[i].Station_Name, name) == 0)
            return true; //如果存在相同的站名就返回true
    }
    return false;
}

int locate(MetroGraph *G, char name[])
{    for (int i = 0; i < G->numNodes; i++)
    {
        if (strcmp(G->Stations[i].Station_Name, name) == 0)
            return i;
    }
    return -1;
}

EdgeNode *Find_same_line(EdgeNode *head, int Line_number)
{    head = head->next;
    while (head)
    {
        if (head->line_number == Line_number)
            return head;
        head = head->next;
    }
    return NULL;
}

EdgeNode *Search_same_line(EdgeNode *head, int Line_number)
{    while (head)
    {
        if (head->line_number == Line_number)
```

```
574.                 return head;
575.             head = head->next;
576.         }
577.         return NULL;
578.     }
579.
580.     void print_line(MetroGraph *G, int Line_number)
581.     {    printf("%d ", Line_number);
582.         //在起始站数组中寻找该起始点
583.         char Start[MAXLENGTH];          //存放起始站名
584.         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
585.         char After_Station[MAXLENGTH]; //用于存放下一个站点
586.         EdgeNode *ptr = NULL;
587.         strcpy(Start, start_stations[Line_number]);
588.         int index = locate(G, Start);
589.         float After_length = G->Stations[index].firstedge->distance;
590.         printf("%s %.2f ", Start, After_length);
591.         strcpy(Prev_Station, Start);
592.         strcpy(After_Station, G->Stations[index].firstedge->name);
593.         while (true)
594.         {
595.             index = locate(G, After_Station);
596.             if (G->Stations[index].firstedge->next == NULL)
597.                 break;
598.             ptr = Search_same_line(G->Stations[index].firstedge, Line_number);
599.             if (strcmp(Prev_Station, ptr->name) == 0) //如果访问过直接从ptr 开始寻找
600.             {
601.                 ptr = ptr->next;
602.                 ptr = Search_same_line(ptr, Line_number);
603.             }
604.             strcpy(Prev_Station, After_Station);
605.             if(ptr == NULL)
606.                 break;
```

```
607.            After_length = ptr->distance;
608.            printf("%s %.2f ", After_Station, After_length);
609.            strcpy(After_Station, ptr->name);
610.        }
611.        printf("%s", After_Station);
612.    }
613.
614.    void ShortestPath_Dijkstra(MetroGraph *G, int v0, int Pa
tharc[], float ShortPathTable[])
615.    {    int v,w,k;
616.        float min;
617.        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）
618.        //初始化部分
619.        for(v = 0; v<G->numNodes; v++)
620.        {
621.            Patharc[v] = -1;
622.            ShortPathTable[v] = INFINITY;
623.        }
624.        //另外需要在最短路径数组里初始化与 v0 邻接的点的权值
625.        ShortPathTable[v0] = 0;  //该点到自己的距离为 0
626.        final[v0] = 1;  //该源点不需要被访问
627.        EdgeNode *ptr = G->Stations[v0].firstedge;
628.        while(ptr!=NULL)
629.        {
630.            int index = locate(G,ptr->name);
631.            ShortPathTable[index] = ptr->distance;
632.            ptr = ptr->next;
633.        }
634.        for(v = 1; v<G->numNodes; v++)
635.        {
636.            min = INFINITY;
637.            for(w = 0; w<G->numNodes; w++)
638.            {
639.                if(!final[w] && ShortPathTable[w]<min)
640.                {
641.                    k = w;
642.                    min = ShortPathTable[w];
```

```
643.                 }
644.             }
645.             final[k] = 1;
646.             ptr = G->Stations[k].firstedge;
647.             while(ptr!=NULL)
648.             {
649.                 int index = locate(G,ptr->name);
650.                 if(!final[index] && (min+ptr->distance)<ShortPathTable[index])
651.                 {
652.                     ShortPathTable[index] = min + ptr->distance;
653.                     Patharc[index] = k;
654.                 }
655.                 ptr = ptr->next;
656.             }
657.         }
658.     }
659.
660.     void Print_ShortestPath(MetroGraph *G, int Vend, float Sum, const int Patharc[], const float ShortPathTable[])
661.     {     if(Patharc[Vend] == -1)
662.         {
663.             printf("%.2f %s ",Sum,G->Stations[Vend].Station_Name);
664.             return;
665.         }
666.         int index = Patharc[Vend];
667.         float distance = Sum - ShortPathTable[index];
668.         Print_ShortestPath(G,index,ShortPathTable[index],Patharc,ShortPathTable);
669.         printf("%.2f %s ",distance,G->Stations[Vend].Station_Name);
670.     }
```