#include <stdio.h> // 标准输入输出库

#include <stdlib.h> // 标准库，包含内存分配函数

#include <limits.h> // 包含数据类型限制的常量定义

#include <stdbool.h> // 包含布尔类型定义（不加一堆warning）

#define NUM\_CITIES 26 // 定义城市数量

#define INF INT\_MAX // 定义无穷大为整型的最大值

//元素：src-出发地，dest-目的地，edge-路径/边，

// 定义边的结构体

typedef struct Edge

{

int dest; // 目的地

int weight; // 边的权重

struct Edge\* next; // 指向下一条边的指针

} Edge;

// 定义图的邻接表

typedef struct Graph

{

Edge\* adj[NUM\_CITIES]; // 邻接表数组

} Graph;

// 定义城市的名称

const char\* cities[NUM\_CITIES] = {"乌鲁木齐", "兰州", "西宁", "呼和浩特", "北京","天津","沈阳","长春","哈尔滨","大连","西安","郑州","徐州","拉萨","成都","武汉","上海","昆明","贵阳","株洲","南昌","福州","柳州","南宁","广州","深圳"};

//只有 char\*的话地址不变，但指针的数值可能改变，必须用const char\*表示一个限定不会被改变的指针变量

// 添加内存分配检查

Edge\* createEdge(int dest, int weight)

{

Edge\* edge = (Edge\*)malloc(sizeof(Edge)); // 为边分配内存

if (!edge) // 检查内存分配是否成功

{

perror("内存分配错误!"); // 打印错误信息

exit(EXIT\_FAILURE); // 退出程序

}

edge->dest = dest; // 设置目的地

edge->weight = weight; // 设置权重

edge->next = NULL; // 初始化下一条边为NULL

return edge; // 返回创建的边

}

// 初始化图

Graph\* createGraph()

{

Graph\* graph = (Graph\*)malloc(sizeof(Graph)); // 为图分配内存

if (!graph) // 检查内存分配是否成功

{

perror("图记忆分配失败"); // 如果失败，输出错误信息

exit(EXIT\_FAILURE); // 退出程序

}

for (int i = 0; i < NUM\_CITIES; i++) // 初始化邻接表

{

graph->adj[i] = NULL; // 将所有邻接表元素初始化为NULL

}

return graph; // 返回创建的图

}

// 添加边（双向边）

void addEdge(Graph\* graph, int src, int dest, int weight)

{

Edge\* edge = createEdge(dest, weight); // 创建新的边-指向目的地

edge->next = graph->adj[src]; // 将正向边插入到邻接表的前面

graph->adj[src] = edge; // 更新邻接表

// 添加反向边（可在main()里算两次算法来使得顺序和逆序都可以计算，反过来计算；不过这里直接算了）

edge = createEdge(src, weight); // 创建反向边-指向出发地

edge->next = graph->adj[dest]; // 将反向边插入到邻接表的前面

graph->adj[dest] = edge; // 更新邻接表

}

// 释放图的内存

void freeGraph(Graph\* graph)

{

for (int i = 0; i < NUM\_CITIES; i++) // 遍历每个城市

{

Edge\* temp = graph->adj[i]; // 获取邻接表

while (temp) // 遍历邻接表中的所有边

{

Edge\* toDelete = temp; // 保存当前边

temp = temp->next; // 移动到下一条边

free(toDelete); // 释放当前边的内存

}

}

free(graph); // 释放图的内存

}

// 找到尚未处理的顶点中距离最小的顶点的索引

int minDistance(int dist[], bool sptSet[], int V)

{

int min = INT\_MAX, min\_index = -1; // 初始化最小值

for (int v = 0; v < V; v++) { // 遍历所有顶点

if (!sptSet[v] && dist[v] <= min) // 找到未处理且距离最小的顶点

{

min = dist[v]; // 更新最小值

min\_index = v; // 更新最小值的索引

}

}

return min\_index; // 返回最小值的索引

}

// 打印从源点到目标点的最短路径

void printPath(int parent[], int j) //举例: 令j是单节点

{

if (parent[j] == -1) //若j是源节点

{

return;

}

printPath(parent, parent[j]); // 递归打印父节点

printf(" -> %s", cities[j]); // 打印当前节点

}

// 打印从出发点到目标点的最短路径及其距离

void printSolution(int dist[], int parent[], int src, int dest)

{

printf("从 %s 到 %s 的最短路径距离是 %d\n", cities[src], cities[dest], dist[dest]); // 打印最短路径距离

printf("路径: %s", cities[src]); // 打印起点

printPath(parent, dest); // 打印路径

printf("\n");

}

// Dijkstra算法 （迪杰斯特拉）

void dijkstra(Graph\* graph, int src, int dest)

{

int dist[NUM\_CITIES]; // 保存最短路径的距离

bool sptSet[NUM\_CITIES]; // 记录顶点是否被处理

int parent[NUM\_CITIES]; // 记录路径中的父节点

// 初始化距离和已访问集合

for (int i = 0; i < NUM\_CITIES; i++)

{

dist[i] = INF; // 将所有距离初始化为无穷大

sptSet[i] = false; // 初始化所有顶点为未处理

parent[i] = -1; // 初始化父节点为-1

}

dist[src] = 0; // 源节点到自身的距离为0

// 查找最短路径

for (int count = 0; count < NUM\_CITIES - 1; count++)

{

int u = minDistance(dist, sptSet, NUM\_CITIES); // 找到距离最小的顶点

if (u == -1)

{

break;

}

sptSet[u] = true; // 将找到的顶点标记为已处理

// 更新所有相邻顶点的距离

Edge\* temp = graph->adj[u]; // 获取邻接表

while (temp)

{

int v = temp->dest; // 获取邻接顶点

if (!sptSet[v] && dist[u] != INF && dist[u] + temp->weight < dist[v])

{

parent[v] = u; // 更新父节点

dist[v] = dist[u] + temp->weight; // 更新距离

}

temp = temp->next; // 移动到下一条边

}

}

// 打印最短路径和距离

printSolution(dist, parent, src, dest);

}

int main()

{

Graph\* graph = createGraph(); // 创建图

// 添加城市之间的边的权重

addEdge(graph, 0, 1, 1892); //乌鲁木齐-兰州

addEdge(graph, 2, 1, 216); //西宁-兰州

addEdge(graph, 1, 3, 1145); //兰州-呼和浩特

addEdge(graph, 3, 4, 668); //呼和浩特-北京

addEdge(graph, 4, 5, 137); //北京-天津

addEdge(graph, 5, 6, 704); //天津-沈阳

addEdge(graph, 6, 7, 305); //沈阳-长春

addEdge(graph, 7, 8, 242); //长春-哈尔滨

addEdge(graph, 6, 9, 397); //沈阳-大连

addEdge(graph, 1, 10, 676); //兰州-西安

addEdge(graph, 10, 11, 511); //西安-郑州

addEdge(graph, 4, 11, 695); //北京-郑州

addEdge(graph, 11, 12, 349); //郑州-徐州

addEdge(graph, 5, 12, 674); //天津-徐州

addEdge(graph, 10, 14, 842); //西安-成都

addEdge(graph, 13, 14, 1990); //拉萨-成都

addEdge(graph, 11, 15, 534); //郑州-武汉

addEdge(graph, 12, 16, 651); //徐州-上海

addEdge(graph, 14, 18, 967); //成都-贵阳

addEdge(graph, 14, 17, 1100); //成都-昆明

addEdge(graph, 15, 19, 409); //武汉-株洲

addEdge(graph, 16, 20, 825); //上海-南昌

addEdge(graph, 17, 18, 639); //昆明-贵阳

addEdge(graph, 18, 24, 902); //贵阳-株洲

addEdge(graph, 19, 20, 367); //株洲-南昌

addEdge(graph, 20, 21, 622); //南昌-福州

addEdge(graph, 18, 22, 607); //贵阳-柳州

addEdge(graph, 19, 22, 672); //株洲-柳州

addEdge(graph, 19, 24, 675); //株洲-广州

addEdge(graph, 22, 23, 255); //柳州-南宁

addEdge(graph, 24, 25, 140); //广州-深圳

// 输入源点和目标点

int src, dest;

printf("[地址编号表] \n");

printf("0-乌鲁木齐, 1-兰州,2-西宁,3-呼和浩特,4-北京,5-天津,6-沈阳,7-长春,8-哈尔滨,9-大连,10-西安,11-郑州,12-徐州\n ");

printf("13-拉萨,14-成都,15-武汉,16-上海,17-昆明,18-贵阳,19-株洲,20-南昌,21-福州,22-柳州,23-南宁,24-广州,25-深圳\n");

printf("\n");

printf("请输入出发地的编号: ");

scanf("%d", &src); // 输入出发地编号

printf("请输入目的地的编号: ");

scanf("%d", &dest); // 输入目的地编号

printf("\n");

// 检查输入是否有效

if (src < 0 || src >= NUM\_CITIES || dest < 0 || dest >= NUM\_CITIES)

{

printf("输入的编号无效，请输入0到25之间的数字。\n"); // 输入无效时，输出错误信息

freeGraph(graph); // 释放图的内存

return EXIT\_FAILURE; // 退出程序

}

// 调用Dijkstra算法

dijkstra(graph, src, dest); // 计算最短路径

// 释放图的内存

freeGraph(graph);

return 0;

}