

一、目的：

1、掌握、巩固图的邻接表存储方法和最短路径算法。

2、通过实际例子模拟一个交通查询系统，学会运用所学知识解决实际问题。处于不同目的的旅客对交通工具具有不同的要求。例如，因公出差的旅客希望在旅途中的时间尽可能短，出门旅游的旅客则期望旅费尽可能省，而老年旅客则要求中转次数最少。

二、内容和要求：

1、内容：

处于不同目的的旅客对交通工具具有不同的要求。例如，因公出差的旅客希望在旅途中的时间尽可能短，出门旅游的旅客则期望旅费尽可能省，而老年旅客则要求中转次数最少。编制一个全国城市间的交通咨询程序，为旅客提供两种或三种最优决策的交通咨询。

2、基本要求：

(1) 提供对城市信息进行编辑（如：添加或删除）的功能。

(2) 城市之间有两种交通工具：火车和飞机。提供对列车时刻表和飞机航班进行编辑（增设或删除）的功能；

(3) 提供两种最优决策：最快到达或最省钱到达。全程只考虑一种交通工具。旅途中耗费的总时间应该包括中转站的等候时间。

(4) 咨询以用户和计算机的对话方式进行。由用户输入起始站、终点站、最优决策原则和交通工具，输出信息：最快需要多长时间才能到达或者最少需要多少旅费才能到达，并详细说明依次于何时乘坐哪一趟列车或哪一次班机到何地。

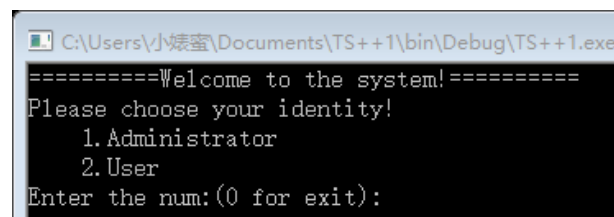
(5) 对全国城市交通图和列车时刻表及飞机航班表的编辑，应该提供文件形式输入和键盘输入两种方式。飞机航班表的信息应包括：起始站的出发时间、终点站的到达时间和票价；列车时刻表则需根据交通图给出各个路段的详细信息，例如：基于教科书 7.6 节图 7.33 的交通图，对从北京到上海的火车，需给出北京至天津、天津至徐州及徐州至上海各段的出发时间、到达时间及票价等信息。

(6) 以邻接表作交通图的存储结构，表示边的结点内除含有邻接点的信息外，还应包括交通工具、路程中消耗的时间和花费以及出发和到达的时间等多项属性。

三、需求分析：

1、输入形式：（根据列表选择数据即可，‘0’用于返回上一级）

(1) 提供用户和管理员选择。



```
C:\Users\小婊宝\Documents\TS++1\bin\Debug\TS++1.exe
=====Welcome to the system!=====
Please choose your identity!
1.Administrator
2.User
Enter the num:(0 for exit):
```

(2) 其中只有管理员通过账号密码登录才允许对城市、线路（即交通图）进行编辑和修改，同时可以对存储城市、线路文件进行更新同步，也可以修改密码。

```
C:\Users\小婊蜜\Documents\TS++1\bin\Debug\TS++1.exe
Now is the Administrator!

Please input account:Administrator
Please input password:Administrator

Login OK!

Please choose your operation:
    1.Change Password
    2.Operation about Flight
    3.Operation about Train
Enter the num:(0 for identity):2

Now is operations about Flight
Please choose the function!
    1.Add Flight city from the file.
    2.Add Flight city which you enter.
    3.Delete Flight city.
    4.Add Flight from the file.
    5.Add Flight which you enter.
    6.Delete Flight .
    7.Show all the cities!
    8.Show all the lines!
Enter the num:(0 for AdminOpr)
搜狗拼音输入法 全 :
```

(3) 用户只有查询功能，可以选择交通工具以及希望的最优决策。

```
C:\Users\小婊蜜\Documents\TS++1\bin\Debug\TS++1.exe
Now is the User!

Please choose your tool!
    1.Flight
    2.Train
Enter the num:(0 for identity):1

Now is operations about Flight
Please choose the function!
    1.Search cheapest way!
    2.Search fastest way!
    3.Show all the cities!
    4.Show all the lines!
Enter the num:(0 for tool)
```

(4) 管理员：其中从文件中导入城市列表和对应交通工具时刻表、同步信息到相应的文件，这些功能只需要输入对应的功能序号即可调用。如：

```

Now is operations about Flight
Please choose the function!
1.Add Flight city from the file.
2.Add Flight city which you enter.
3.Delete Flight city.
4.Add Flight from the file.
5.Add Flight which you enter.
6.Delete Flight .
7.Show all the cities!
8.Show all the lines!
Enter the num:(0 for AdminOpr)1

1.Add Flight city from the file.
从FCity.txt文件中导入城市!
城市导入完毕!

```

(5) 管理员：添加、删除城市，添加、删除线路都是需要根据相应的提示进行输入的，如添加线路：

```

Now is operations about Flight
Please choose the function!
1.Add Flight city from the file.
2.Add Flight city which you enter.
3.Delete Flight city.
4.Add Flight from the file.
5.Add Flight which you enter.
6.Delete Flight .
7.Show all the cities!
8.Show all the lines!
Enter the num:(0 for AdminOpr)5

5.Add Flight which you enter.
请输入起点城市：青岛
请输入终点城市：济南
请输入班次名：D234
请输入出发时间：(hh:mm, +d)11:47, +0
请输入到达时间：(hh:mm, +d)15:34, +0
请输入票价：104.5

```

(6) 用户：查询路径只需要输入出发城市和到达城市即可。

```

Now is operations about Flight
Please choose the function!
1.Search cheapest way!
2.Search fastest way!
3.Show all the cities!
4.Show all the lines!
Enter the num:(0 for tool)1

1.Search cheapest way!
Please enter start city:成都
Please enter end city:乌鲁木齐
最省钱路径：
成都 西安 CZ1015 07:20, +0 08:40, +0 01:20, +0 560.000
需要中转等待 825分钟!
西安 兰州 CZ1013 22:25, +0 23:30, +0 01:05, +0 540.000
需要中转等待 640分钟!
兰州 乌鲁木齐 CZ1010 10:10, +0 12:50, +0 02:40, +0 890.000
一共花费1990.00元和1770分钟!

```

2、输出形式：

(1) 无论是用户还是管理员都可以打印当前系统中（交通图）所有城市和线路，以线路为例，先告知有多少条线路，然后逐条打印，打印格式如下：（篇幅问题，只截图部分）

```
8.Show all the lines!
There are 60 line(s) in the system:
出发城市|到达城市|班次名|出发时间|到达时间|||用时|||票价
广州      深圳      T101 06:00,+0 07:26,+0 01:26,+0 143.500
广州      株洲      T102 07:48,+0 15:16,+0 07:28,+0 158.000
株洲      广州      T103 00:38,+0 08:05,+0 07:27,+0 158.000
株洲      南昌      T104 04:05,+0 09:00,+0 04:55,+0 99.5000
株洲      武汉      T105 03:37,+0 08:47,+0 05:10,+0 110.500
株洲      贵阳      T106 00:31,+0 13:42,+0 13:11,+0 200.000
株洲      柳州      T107 05:16,+0 13:53,+0 08:37,+0 152.000
南昌      株洲      T108 00:28,+0 04:57,+0 04:29,+0 99.5000
南昌      福州      T109 06:48,+0 10:25,+0 03:37,+0 133.000
南昌      上海      T110 16:59,+0 04:55,+1 11:56,+0 184.000
福州      南昌      T111 11:30,+0 18:12,+0 06:42,+0 133.000
上海      南昌      T112 06:25,+0 10:12,+0 03:47,+0 184.000
上海      徐州      T113 13:44,+0 22:48,+0 09:04,+0 158.000
徐州      上海      T114 00:14,+0 09:33,+0 09:19,+0 158.000
徐州      天津      T115 01:42,+0 08:05,+0 06:23,+0 158.000
徐州      郑州      T116 19:03,+0 23:43,+0 04:40,+0 91.5000
天津      徐州      T117 06:04,+0 15:27,+0 09:23,+0 163.000
天津      沈阳      T118 01:49,+0 09:07,+0 07:18,+0 163.000
天津      北京      T119 01:42,+0 04:26,+0 02:44,+0 67.5000
沈阳      天津      T120 01:22,+0 10:05,+0 08:43,+0 163.000
```

(2) 管理员：登录时如果账号或密码错误，会进行提示，并重新输入；修改密码时需要两次输入密码，和现实中的修改密码一样。

```
C:\Users\小媛室\Documents\TS++1\bin\Debug\TS++1.exe
Now is the Administrator!

Please input account:Administrator
Please input password:Administrator888
Account or Password Error!

Try again:
Please input account:Administrator
Please input password:Administrator

Login OK!

Please choose your operation:
1.Change Password
2.Operation about Flight
3.Operation about Train
Enter the num:(0 for identity):1

Please input new password:Administrator888
Please input again:Administrator888
Change Password OK!
```

(3) 管理员：删除线路时，输入起点和终点城市后，会列出该城市间已有的路线，若无则输出 NULL，然后再提示输入线路名字。

```

6.Delete Train .
请输入起点城市: 广州
请输入终点城市: 深圳
该城市间的线路有:
起点城市 终点城市 班次名 出发时间 到达时间 用时 票价
广州      深圳      T101 06:00,+0 07:26,+0 01:26,+0 143.500
广州      深圳      T123 12:40,+0 13:20,+0 00:40,+0 30.0000
请输入你要删除的航班号:

```

(4) 管理员: 删除城市时若输入城市时不存在的, 会提示没找到该城市。删除线路时同理。

```

3.Delete Flight city.
Please enter the city you want to delete:青岛
Can't find out the city!

```

(5) 用户: 在查询最优路径时, 若输入城市不存在, 系统会打印所有城市出来供用户参考:

```

1.Search cheapest way!
Please enter start city:崂山
Can not find out the city!
There are 14 city(s) in the system:
0南昌
1上海
2天津
3沈阳
4北京
5郑州
6呼和浩特
7兰州
8乌鲁木齐
9西安
10成都
11昆明
12贵阳
Please enter again:

```

(6) 查询最优路径时, 会把线路信息输出, 若要中转, 会告知等待时间, 最后会告知总时间和费用。如上图(6)所示。

3、实现功能: 如要求和上述所示。

四、概要设计:

1、数据结构:

(1) 采用邻接表的存储结构, 除了题目要求的原因外, 还有一点是邻接表结构中的弧是以链表的形式存储, 这样增加和删除线路的操作会方便很多。

(2) 头结点包含城市名字(string)、城市序号(int)、该点的出度(int), 即以该城市为起点的线路的条数, 还有该城市指向第一个结点的链域:

```

typedef struct VNode{//头结点
    string CityName;//地名
    int CityOrd;//编号
    LineNode *FirstLine;//指向第一条依附该顶点的弧的指针
    int Amount;//交通工具的班次
}VNodeDat;

```

(3) 表结点包含该弧的终点城市名字(string)、该弧的基本信息和指示下一条弧的链域(及同样城市起点的下一条线路), 其中基本信息包括车次号(string)、出发和到达时间、所花费的时间, 所花费的金钱(float)。

```

typedef struct InfoType{
    string LineName;//航班或车次号, 默认最长为8 (算上'\0')
    Time StartTime, EndTime;//出发、到达时间
    Time SpendTime;//所花费的时间
    float SpendMoney;//所花费的金钱
}TrafficLine;

typedef struct ArcNode{//表结点
    string EndCityName;//该弧所指向的城市, 即终点城市
    struct ArcNode *NextLine;//指向下一条弧的指针
    TrafficLine *Info;//该弧相关信息的指针 (该线路的信息)
}LineNode;

```

(4) 其中还定义一个时间结构体, 用于表示时间, 并重载了“-”运算符, 用于计算两时间之前的时间差。

```

//关于时间的格式hour:minute,+day
typedef struct Time{
    int day;
    int hour;
    int minute;

    //重载
    //给出出发时间和到达时间, 计算所用时间, 默认到达时间“大于”出发时间
    //默认结束时间的天数输入有误, 即 本应该加1, 但是却加零
    friend Time operator- (Time &et, Time &st){

```

2、类的定义:

由于一个交通图, 包含城市个数(顶点数), 线路条数(弧数), 城市列表, 而且可以增加、删除城市(顶点), 也可以增加、删除线路(弧), 同时根据需求还要求最短路径等操作, 进而可以把交通图抽象成一个类, 既有成员, 又有操作。

```

class ALGraph{
public:
    ALGraph(int size);
    int SearchCityNum(const string Cityname); //查询城市编号
    void AddCity(const string Cityname); //手动添加城市
    void AddCityFromFile(const char FileName[MAX_FILE_NAME_LEN]); //从文件中导入城市
    void DelCity(const string Cityname); //删除城市
    void InsertHead(string StartName, LineNode *temp, string EndName); //头插线路
    void AddLine(); //手动添加线路
    void AddLineFromFile(const char FileName[MAX_FILE_NAME_LEN]); //从文件中的导入线路
    void DelCityLine(int i); //删除以该城市为起点的航班
    void DelLine(); //删除线路
    void ReSize(int size); //重新开辟城市列表
    void UpdateFile(const char FileName[MAX_FILE_NAME_LEN], const string type); //同步文件
    void ShowAllLine(); //输出所有线路
    void ShowCity(); //输出城市
    void Dijkstra_Money(int v0, int *path, Node *dist); //计算最省钱路径
    void Dijkstra_Time(int v0, int *path, Node1 *dist); //计算最省时路径
    void SearchShortestPath(const string type); //调用并打印最短路径
    ~ALGraph();
private:
    VNodeDat *CityList; //城市列表, 即头结点数组
    int MaxCityNum; //当前数组的长度
    int CityNum; //城市个数, 即图的顶点数
    int arcnum; //图的弧数
};

```

其中私有成员有城市列表, 是一个指针, 调用构造函数会为其开辟空间, 析构函数则会将其释放, 同时有一个数组大小的变量, 因为数组大小不一定是城市数量。

而共有成员都是函数, 除了添加、删除城市、线路等操作外, 还有一些操作如 DelLine() 函数是用于删除城市时把以该城市为起点的线路全部删去并释放空间, InsertHead() 函数则是新增数量是调用的, 为的是让程序根据可读性; 还有 ReSize() 函数则是考虑到新增城市时可能空间不足而需要扩大城市列表数组, 由于多个函数都会调用增加城市的函数, 所以单独写一个出来, 类似数组类的思想, 这些函数等都是为了让程序根据可读性。

2、全局变量:

(1) 初始化类: 飞机和火车的图。因为不是所有城市间都有飞机和火车线路, 所以分开个图, 且题设是用户全程只考虑一种交通工具, 所以分开两个图方便代码的实现。

(2) 二维数组 tool 是为了输出提示时让程序使用者知道当前所操作的是飞机还是火车的图, 两个二维数组 file 则是调用与文件操作相关函数时作为参数传递的。

(3) 这样就能通过数组 (第一维) 下标去直接调用相应工具函数, 即将各种信息转化为数字, 有利于代码的重用。

(4) 之所以设置为全局变量而不是在主函数中, 是因为实际使用这些函数的是主函数所调用的函数, 这些函数并不是类中的函数, 所以和主函数同在一个文件。这些变量作为参数传递时, 几乎都要一起传递; 为了减少参数的传递, 所以设计为全局变量。

```

8 // (第一维) 下标为0代表飞机, 下标为1代表火车
9 ALGraph ALG[2] = {ALGraph(10), ALGraph(20)};
10 char tool[2][7] = {"Flight", "Train"};
11 char cityfile[2][10] = {"FCity.txt", "TCity.txt"};
12 char linefile[2][11] = {"Flight.txt", "Train.txt"};
13 string Account = "Administrator"; //管理员账号名
14 string Password = "Administrator"; //管理员初始秘密

```


3、主函数流程：欢迎界面、选择身份，根据身份进入相应的操作（调用相应的函数）。

```
int main(){
    int identity = 1;
    while(identity){
        cout<<"=====Welcome to the system!===== "<<endl;
        cout<<"Please choose your identity!"<<endl;
        cout<<"    1.Administrator"<<endl<<"    2.User"<<endl;
        cout<<"Enter the num: (0 for exit):";
        cin>>identity;
        if(!identity)
            break;
        system("cls");
        switch(identity){
            case 1:
                AdminOpra();
                break;
            case 2:
                UserOpra();
                break;
            default:
                break;
        }//switch(identity)
        system("cls");
    }
    cout<<endl<<"Thank you! Goodbye!"<<endl;
    return 0;
}
```

4、各程序模块之间的调用关系

(1) 图类成员中的函数：（截图见上）（描述时省略参数）

SearchCityNum(const string Cityname);//查询城市编号

InsertHead(string StartName, LineNode *temp, string EndName);//头插线路

DelCityLine(int i);//删除以该城市为起点的航班

ReSize(int size);//重新开辟城市列表

UpdateFile(const char FileName[MAX_FILE_NAME_LEN], const string type);//同步文件

ShowAllLine();//输出所有线路

ShowCity();//输出城市

Dijkstra_Money(int v0, int *path, Node *dist);//计算最省钱路径

Dijkstra_Time(int v0, int *path, Node1 *dist);//计算最省时路径

- AddCity(const string Cityname);//手动添加城市，先调用 SearchCityNum()函数查看城市是否存在，若不存在才添加；若空间不足，会调用 ReSize()函数
- AddCityFromFile(const char FileName[MAX_FILE_NAME_LEN]);//从文件中导入城市，从文件中获得一个城市名字后，直接调用 AddCity()函数。
- DelCity(const string Cityname);//删除城市，先调用 SearchCityNum()函数查看城市是否存在，若存在才删除，删除时会调用 DelCityLine()把以该城市为起点的线路全删

去（释放空间）

- d. `AddLine();`//手动添加线路，调用 `SearchCityNum()`函数查看城市是否存在；插入使用头插，把起终点相同的线路放一起，调用 `InsertHead()`。
- e. `AddLineFromFile(const char FileName[MAX_FILE_NAME_LEN]);`//从文件中的导入线路，同上
- f. `DelLine();`//删除线路，调用 `SearchCityNum()`函数查看城市序号，通过输入的线路名进行匹配删除
- g. `SearchShortestPath(const string type);`//根据参数 `type` 来调用对应的 `Dijkstra()`算法求得最短路径并打印最短路径
- h. `~ALGraph();`//析构函数，通过多次调用 `DelCityLine()`把以每个城市为起点的线路全删去，最后释放城市列表数组这个空间。

5、使用 C++标准模板库中的两个顺序容器适配器，分别是优先队列和栈。优先队列用于辅助 Dijkstra 算法的实现，栈则用于辅助还原最短路径。在详细设计中会介绍使用情况。

```
10     #include<queue>
11     #include<stack>
```

五、详细设计（主要展示邻接表 Dijkstra 算法的实现、最短路径的还原）

1、`Dijkstra_Money(int v0, int *path, Node *dist);`//计算最省钱路径

（1）首先定义一个节点结构体，用于存储当前线路的起点城市序号和权值，在此函数中权值是费用。引入这个结构体是因为原本表结点中信息太多，在求最短路径时这些信息是没用的。

（2）使用优先队列进行辅助，系统优先队列默认是弹出容器中“最大”的元素。比较运算符本身是不适用于结构体的，所以需要重载。而且求最短路径是，我们是需要权值较少的元素先弹出容器，所以重载小于符号，但是返回大于的真值。

（3）实际上优先队列可以自己写：使用链表存储，压入容器时使用插入排序的思想把元素插入实际的位置即可，但为了减少代码量，我们使用标准库中的模板。

（3）和书上描述的 Dijkstra 算法思想是一致的，但书上给出的伪代码是邻接矩阵的存储结构。以下是邻接表存储结构实现的描述：

- a. 参数：`v0` 是起点城市序号，`path` 数组用于记录下标对应节点的父亲节点，通过这个数组能“顺藤摸瓜”对路径进行还原。`Dist` 数组是起点到该下标对应城市所用的最少金钱，`visited` 数组记录该下标对应城市是否已找到最短路径，和书上说的 `S` 集合同理，找到则并入 `S` 集，即该下标对应的元素值为 1。
- b. 初始化：到所有顶点的的最少金钱是无限大，均没有父亲节点，且没找到最短路径，起点除外。
- c. 优先队列中的元素均为与当前没有求得最短路径且在 `S` 集中有它的父亲节点，即已求得最短路径城市中能直接到达该点的线路都会在优先队列中。
- d. Dijkstra 算法的核心正是找出 `V-S` 集中距离 `S` 集中权值最小的点，让其并入 `S` 集合，即按照路径长度递增的次序产生最短路径。在邻接矩阵中，通过遍历某一维的元素

找出这个点，因为除了这一维，还有之间就已经与 S 集有联系的元素，数组能快速定位这些元素；用邻接表的话遍历某个城市的线路并不是大问题，问题在于定位已有的与 S 集有关的线路是十分麻烦的。引入优先队列就能有效解决这个问题。

(4) 源代码截图如下：

```
//顶点节点，保存城市序号和到源顶点的估算金额，优先队列需要的类型
struct Node{
    int id;//源顶点id
    float money;//估算距离

    //因要实现最小堆，按升序排列，因而需要重载运算符，重定义优先级，以小为先
    friend bool operator < (struct Node a, struct Node b){
        return a.money > b.money;
    }
};

void ALGraph::Dijkstra_Money(int v0, int *path, Node *dist){
    priority_queue<Node>q;//P425
    //path[]记录下标对应节点的父节点。
    int visited[MaxCityNum]; //该下标对应的顶点是否已经算出最短距离，是则为1
    //初始化
    int i;
    for(i = 0; i < CityNum; i++){
        dist[i].id = i;
        dist[i].money = INF;
        path[i] = -1; //每个顶点都无父节点
        visited[i] = 0; //都未找到最短路
    }
    dist[v0].money = 0;
    q.push(dist[v0]);

    while(!q.empty()){
        Node cd = q.top();
        q.pop();
        int u = cd.id;

        if(visited[u])
            continue;
        visited[u] = 1;
        LineNode *p = CityList[u].FirstLine;

        while(p){
            int tempv = SearchCityNum(p->EndCityName);
            float tempm = p->Info->SpendMoney; //金钱为权值
            if(!visited[tempv] && dist[tempv].money > dist[u].money + tempm){
                dist[tempv].money = dist[u].money + tempm;
                path[tempv] = u;
                q.push(dist[tempv]);
            }
            p = p->NextLine;
        } //while (p)
    } //while (!q.empty())
} //Dijkstra_Money
```

2、Dijkstra_Time(int v0, int *path, Node1 *dist); //计算最省时路径

(1) 理论上该算法和上述求最省钱的路径的算法的原理是相同的，唯一的难点在于这个算法的权值除了该线路所花的时间，若需要中转，还要加上中转等待的时间。对于该问题，我们小组的处理方法是若需要中转，则把中转时间算入下一条线路的时间，把这两个时间加起来作为下一条线路的真正权值。举例：A->B->C，则在 B 等待的时间算入 B->C 上。这样做

能使代码重用，因为如果不需要中转，按照上述逻辑也不会产生中转时间。

(2) 但是该思路就必须：若该城市是求得最短路径，则要把到达该城市的到达时间记录好。因为在链表中定位元素是很不方便的，而且实际上并不能通过遍历来获得到达该城市的时间，因为终点为该城市的线路可能不止一条。通过增加内存开销能有效降低复杂度。Dist 结构体需要小改动，截图见下：

(3) 为了方便比较和记录，引入一个时间转化函数，把 Time 结构的时间转化为 int 型的分钟。

```
//顶点节点，保存城市序号和到源顶点的估算时间，优先队列需要的类型
struct Node1{
    int id;//城市序号id
    int tt;//估算时间
    Time et;//到达时间
    //因要实现最小堆，按升序排列，因而需要重载运算符，重定义优先级，以小为大
    friend bool operator < (struct Node1 a, struct Node1 b){
        return a.tt > b.tt;
    }
};

int TransformMinute(const Time &t){
    return ((t.day * 24 + t.hour) * 60 + t.minute);
}
```

(4) 在进优先队列的 while 循环中为了加上中转等待时间，作以下改动：

```
while(p){
    int tempv = SearchCityNum(p->EndCityName);
    int tempt = TransformMinute(p->Info->SpendTime); //时间为权值
    Time st = p->Info->StartTime; //本条线的开始时间
    if(u != v0){ //起点到任何点都没有换乘时间这一说法
        int change = TransformMinute(st - dist[u].et); //换乘时间
        tempt += change;
    }
    if(!visited[tempv] && dist[tempv].tt > dist[u].tt + tempt){
        dist[tempv].tt = dist[u].tt + tempt;
        dist[tempv].et = p->Info->EndTime;
        path[tempv] = u;
        q1.push(dist[tempv]);
    }
    p = p->NextLine;
} //while(p)
```

3、SearchShortestPath(const string type); //根据参数 type 来调用对应的 Dijkstra() 算法求得最短路径并打印最短路径

(1) 起终点城市的输入：若输入城市不载城市列表，则告知用户，并打印城市列表，让用户方便输入正确。

(2) 根据传参 type 决定定义的局部变量 Node 结点和所调用的函数：（一定程度上节省内存开销，但效果不大）

```

int path[MaxCityNum];
int t_m = 0; //时间, 分钟
float m_y = 0; //费用, 元

if(type == "Money"){
    Node dist[MaxCityNum];
    Dijkstra_Money(StartNum, path, dist);
    m_y = dist[EndNum].money;
}
else{
    Node1 dist[MaxCityNum];
    Dijkstra_Time(StartNum, path, dist);
    t_m = dist[EndNum].tt;
}

```

(3) 若所求终点的父亲节点为-1 说明没有最短路径, 提示用户并结束程序即可。

```

if(path[EndNum] == -1){
    cout<<"Sorry, NULL!"<<endl;
    return;
}
else{

```

(4) 还原路径, 无论是最省钱还是最省时, 因为如果需要中转则要打印中转等待信息, 所以无论如何都要引入如 Dijkstra_Time()函数中的中转等待思想去还原路径。

(5) path 数组仅仅是能帮助我们获取该最短路径经过哪些结点, 还有经过顺序的倒序。但是打印的时候需要正序打印, 所以引入栈, 为了减少代码量, 所以是用标准库中的模板。

(6) 同城市间的线路可能不止一条, 所以需要遍历才能知道所求最短路径中的这一段是哪一条, 找到后需要定位(链表需要遍历才能查找), 为了方便定位寻找时引入计数变量:

```

LineNode *p = CityList[father].FirstLine;
float mm = INF; //当前记录到的最少金钱, 不是总金钱
int i = 0, count; //指针移动的次数, 方便定位
while(p){
    if(p->EndCityName == CityList[child].CityName && mm >= p->Info->SpendMoney){
        mm = p->Info->SpendMoney;
        count = i;
    }
    p = p->NextLine;
    i++;
} //while(p)
p = CityList[father].FirstLine;
i = 0;
while(i++ != count)
    p = p->NextLine;

```

(↑ 还原最省钱路径的 ↑)

```

LineNode *p = CityList[father].FirstLine;
int tt = INF, ot = 0; // tt 当前记录到的最少时间，不是总时间；ot 是该条线的真正时间（算上换乘）
int i = 0, count; // 指针移动的次数，方便定位
while(p){
    if(p->EndCityName == CityList[child].CityName){
        if(!s.empty() && child != EndNum)
            ot = TransformMinute(p->Info->SpendTime) + TransformMinute(p->Info->StartTime - et);
        else
            ot = TransformMinute(p->Info->SpendTime);
        if(tt >= ot){
            tt = ot;
            count = i;
        }
    }
    p = p->NextLine;
    i++;
}
p = CityList[father].FirstLine;
i = 0;
while(i++ != count)
    p = p->NextLine;

```

(↑还原最省时路径的，所以若有中转，还需要算等待时间↑)

(7) 中转时间的计算，该线开始时间减去上条线的到达时间，对 Time 结构体“-”运算符的重载这里也能用：

```

tt += TransformMinute(p->Info->SpendTime);
if(father != StartNum){
    tt = tt + TransformMinute(p->Info->StartTime - et);
    cout << "需要中转等待 " << TransformMinute(p->Info->StartTime - et) << "分钟!" << endl;
}

```

(8) 关于 Time 结构体“-”运算符的重载，可能出现的情况是：计算不同线路的时间差，原来的开始时间和结束时间都是+0 天（即没有跨天），但是如果按照分钟，时钟分别借位相减则可能出现天数变成-1，明显是不正确的，所以为了使结果正确，若果相减后天数<0，则默认传参进来的结束时间是错误的，把结果的天数加一处理。

```

//重载
//给出出发时间和到达时间，计算所用时间，默认到达时间“大于”出发时间
//默认结束时间的天数输入有误，即...本应该加1，但是却加零
friend Time operator- (Time &et, Time &st){
    Time t = {0, 0, 0};
    t.minute = et.minute - st.minute; //计算分
    if(t.minute < 0){ //相当于借位
        t.minute += 60;
        t.hour--;
    }

    t.hour = t.hour + et.hour - st.hour; //计算时
    if(t.hour < 0){
        t.hour += 24;
        t.day--;
    }

    if(et.day <= st.day) //此时如果天数相等，那么说明录入的时候结束时间的天数是错误的
        t.day++;
    t.day = t.day + et.day - st.day; //计算天
    return t;
}

```

六、算法复杂度分析

1. `ALGraph(int size);`//构造函数
`O(size)`
2. `int SearchCityNum(const string Cityname);`//查询城市编号
`O(CityNum)`
3. `void AddCity(const string Cityname);`//手动添加城市
`O(1)`
4. `void AddCityFromFile(const char FileName[MAX_FILE_NAME_LEN]);`//从文件中导入城市
`O(文件的行数)`
5. `void DelCity(const string Cityname);`//删除城市
`O(CityNum) + SearchCityNum() + DelCityLine()`
6. `void InsertHead(string StartName, LineNode *temp, string EndName);`//头插线路
`O(CityList[StartNum]. Amount) + 2 * SearchCityNum()`//遍历一条链表
7. `void AddLine();`//手动添加线路
`InsertHead()`
8. `void AddLineFromFile(const char FileName[MAX_FILE_NAME_LEN]);`//从文件中的导入线路
`O(1 + (文件行数 - 1) * InsertHead())`//第一行是目录，不需要调用插入函数
9. `void DelCityLine(int i);`//删除以该城市为起点的航班
`O(CityList[i]. Amount)`
10. `void DelLine();`//删除线路
`O(3 * CityList[StartNum]. Amount) + SearchCityNum()`//遍历链表寻找同起终点的线路
+遍历链表匹配同班次名的线路+遍历链表进行删除
11. `void ReSize(int size);`//重新开辟城市列表
`O(size)`
12. `void UpdateFile(const char FileName[MAX_FILE_NAME_LEN], const string type);`//同步文件
`O(CityNum)` 或 `O(arcnum)`//同步城市则是前者，同步线路则是后者
13. `void ShowAllLine();`//输出所有线路
`O(arcnum)`
14. `void ShowCity();`//输出城市
`O(CityNum)`
15. `void Dijkstra_Money(int v0, int *path, Node *dist);`//计算最省钱路径
`O(CityNum * CityNum)`//确切来说是 `CityNum + CityNum * (CityNum - 1)`
16. `void Dijkstra_Time(int v0, int *path, Node1 *dist);`//计算最省时路径
`O(CityNum * CityNum)`//同上
17. `void SearchShortestPath(const string type);`//调用并打印最短路径
`O(arcnum) + 2 * SearchCityNum() + Dijkstra_()`
18. `~ALGraph();`//析构函数
`O(CityNum * DelCityLine())`

七、课程设计中遇到的问题及解决的办法

1、审题：一开始设计结构体的时候认为表结点应当包含距离，实际上题目并不是要求找距离最短的路径，而是最省钱和省时，也就是说权值是金钱和时间，在实际生活中与距离不一定有关系，所以不需要记录距离的。

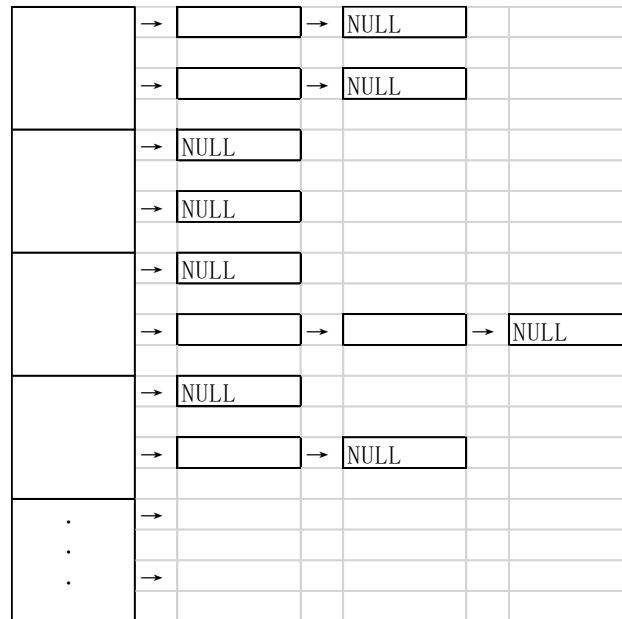
2、我们一开始的设计思路是先完成最短路径的模块，再文件读写等其他模块，但是发现如果手动创建固定的交通图，信息录入量很大。所以后来我们改变了思路，把顺序反过来，先完成除了求最短路径的模块，也即创建和修改图，这样确保每个内存中的信息是正确的，最后调用迪杰斯特拉算法时直接就能使用内存中需要的值。

3、关于链表，以往我们学习的时候链表都有一个头结点用于记录链表的信息，传参时由于头结点和其他结点的结构是一样的，所以对链表的操作代码的重用性很好，但是邻接表中不能说没有头结点，只是头结点格式和表结点的格式是不同的，所以在增加和删除结点时要注意第一个结点和其他结点的情况，这样编写增加和删除的时候分的情况就相对较多，代码量也大，不过认真耐心点还是没有问题的。

4、某些空类型函数如 `void UpdateFile()`，会根据参数的值选择不同的文件和文字格式进行同步，同步完即可结束函数而不进行其他操作，一开始用 `exit(0)`，但是发现这个函数是用来终止正在执行的进程而不仅仅是某个函数。所以得要用 `return;`即可。

5、在删除城市的函数中 `void DelCity()`一开始删除城市后，把该城市以下的所有城市上移，序号都减一，但是最后发现其他函数可能会崩溃，通过对内存内容的输出，我们发现原来在线路(表结点)中记录的序号是不变，这就造成即使是把以该城市为起点终点的线路删除后，但涉及该城市的所有线路起点或目的地在内存中已经发生了改变。实际上这也是逻辑上的一种漏洞。为了解决这个问题，我们把表结点中记录的序号换成了城市，所以表结点中不在含有终点城市序号，而是直接记录终点城市的名字，若需要用到序号时，通过调用 `int SearchCityNum()`函数即可。

6、一开始为了节省内存，所以设计邻接表的时候头结点拥有两个指向第一个结点的链域分别指向飞机和火车线路（见下图），但是最后发现这样代码就无法重用了。举个例子，删除线路的函数 `void DelLine()`如果希望代码重用，那么传参应该是传递弧的指针也就是第一个结点的链域，但是有可能删除的正是第一个结点，这样是无法让原城市列表中对应的指向第一个结点的链域在调用函数后指向正确的位置。同理增加删除线路、同步文件等函数都必须都要写两份，而且两份的代码几乎一样，仅仅是函数内定义头指针的赋值不一样而已。实际上头结点并没有占用很大的内存空间，因为它仅仅有城市名字(string)、城市序号(int)、该点的出度(int)，即以该城市为起点的线路的条数，还有该城市指向第一个结点的链域，这些加起来相对于程序来说并不会占用很大内存；而且也并不是火车和飞机的拥有的城市是相同的，数量上火车也明显比飞机多；综上，没有必要为了节省这些内存而使程序更复杂，更不好实现。



7、我们小组所采用的 IDE 是 codeblocks 13.12 版本，在 C++ 中，有对 ERROR 进行宏定义，在 int SearchCityNum() 函数中若查找不成功则会返回 ERROR，也就是我们所希望的 -1，很多函数都会调用查找城市序号函数，在测试代码的时候我们发现如果在增加、删除城市、线路的函数中，有些城市明明在城市列表中，但却告知“返回”的是 ERROR。后来建立工程进行单步调试（该 IDE 只有建立工程才能单步调试），发现 int SearchCityNum() 能正常返回，如果查询不到，也确实返回 -1，但是如果用判断语句如 if(SearchCityNum(StartName) == ERROR){...} 时，即使返回值为 0 也能进入 if 语块。尔后我们发现出现上述情况都是输入的城市在城市列表中的第 0 个城市。原因我们还没找到，但是是解决方案很明显，就是把所有 ERROR 用 -1 代替。

8、判断语句如果是多个并列的话需要考虑先后顺序，如，在 void InsertHead() 函数中：

```
p = CityList[StartNum].FirstLine;
if(p == NULL) { // 原本没有航班的情况
    CityList[StartNum].FirstLine = temp; // 不能 p = temp
    temp->NextLine = NULL;
} // if
else {
    q = p->NextLine;
    while(q != NULL && EndName != q->EndCityName) { // 把终点城市相同的航线放到一起
        // 前后顺序不能倒，而且应该是判断 p 空而不是 p->next
        p = q;
        q = q->NextLine;
    }
    p->NextLine = temp;
    temp->NextLine = q;
} // else
```

While 循环的判断语句两个条件顺序颠倒则可能 q 为空，此时若取其内存，必然导致程序崩溃。实际上如果两个条件是或关系的话，调整先后顺序能一定程度上提高程序执行效率，如把常能判断为真的条件放前面，这样总体上能减少判断次数。

9、开始还不是使用类的思想进行编写时候，重新分配空间的函数 void ReSize() 传递的是城市列表数组的地址，不带任何返回值。因为考虑到改变指针指向的值，可以不返回就能是主函

数中对应的值改变。实际上我们一开始学习指针的时候，最常用的例子是通过传递指针使得主函数的两个值通过指针传参在 `swap` 交换函数完成交换且不带返回值也能完成真正的交换，所以平时使用指针时，我们几乎没接触到需要返回指针的情况。这次设计同理。我们发现每次调用 `ReSize()` 函数后若使用新开部分的内存，都会是程序奔溃，但是该函数中设置了容错性代码，若分配不成功会进行提示，崩溃时没有一次会提示说明分配成功，但为啥还会崩溃呢？后来我们单步调试发现调用这个函数是能正常调用并结束的。这时候我们对调用前后城市列表指针进行地址输出发现竟然是一样的，也就是说经过分配后地址仍然一样，可是在函数里重新分配后立刻输出地址值，却有发现不一样。很明显，空间确实重新分配了，但函数结束后，原来的空间还是原来的空间，新增的空间被释放了或者说泄露了。查书发现：（当时是用 C 写的）永远不能试图通过改变形参的值去改变实参的值，`swap` 函数之所以成功是因为它改变的是指针所指向的内容，而不是指针本身。在调用 `realloc()` 重新开辟空间后，形参城市列表的指针已经发生变化，若想实参也发生变化就要返回它。其实道理和普通的 `int` 型一样，此时指针也是一个数，如果在函数中想让这个 `int` 型的变量改变并影响到上一层的函数，就需要返回值。所以在之后的版本中都加入了返回值或者说让城市列表指针重新指向新开辟的内存空间。如：（见黑色行）

```
//重新开辟城市列表，只考虑增多，不考虑减少
void ALGraph::ReSize(int size){
    if(size <= CityNum)//已经包括size为负数的情况
        return;
    VNodeDat *NewList = new VNodeDat[size];
    int i;
    for(i = 0; i < CityNum; i++)//不是CityNum-1
        NewList[i] = CityList[i];//结构体赋值会整体赋值，前提是后者都要有值。

    for(i = CityNum; i < size; i++){
        NewList[i].Amount = 0;
        NewList[i].CityOrd = i;
        NewList[i].FirstLine = NULL;
        NewList[i].CityName = "NULL";
    }
    delete []CityList;
    CityList = NewList;
} //ReSize
```

10、连续输入文字时，可能会出现如下情况。

```
add flight:
请输入起点城市: 青岛
请输入终点城市: 西宁
请输入航班名: CA2345
请输入起飞时间: (hh:mm, +d) 12:20, +0
请输入降落时间: (hh:mm, +d) 13:59, +0
请输入票价: 2345
del flight:
请输入起点城市: 请输入终点城市:
```

没有输入起点城市，就提示输入终点城市了。实际上原因在于我们使用 `getline` 函数获得字符串该函数能以回车符作为结束符，但是保留在缓冲区等待适配，如果我们在这之前加入 `putchar()` 函数，会发现 `del flight:` 后会空出一行，才提示输入，并且不会出现上述情况，则说明缓冲区中多了一个回车符。如果用 `getchar()` 函数消去，那么放 `getchar()` 函数的地方是很难预知而且不一定每次都是固定地方消去回车的，如果设置不当，程序则会等待用户输入，但是 `getchar()` 函数是不接受用户输入的回车符的，它只把用户输入的回车符当做是结束符，不会把它写入缓冲区，这样就必须要让用户输入一个可视字符才能让程序继续运行，十分混乱，

而且加入 `getchar()` 函数也使得程序可读性降低，所以统一改成正常的 “`cin>>.....`” 输出就好了。

11、时间结构体 “-” 运算符重载函数的逻辑错误：其中判断 `if(et.day <= st.day)` 不应该放在 `if(t.hour < 0)` 里，因为在计算中转时间的时候，使用下一条线的出发时间减去到达该城市的时间，前者肯定是+0，但后者可能是+1，也可能+0，如果是前者的还，嵌套当然没问题，但如果是后者且时分是小于减数时间的话，就会出现计算出来的时间是一个负数，所以不应该把这句判断嵌套。

```
.....
if(t.hour < 0){
    t.hour += 24;
    t.day--;
    if(et.day <= st.day) //此时如果天数相等，那么说明录入的时候结束时间的天数是错误的
        t.day++;
}

t.day = t.day + et.day - st.day; //计算天
return t;
```

- 12、设计中还存在着几个已知的但未解决的问题，这些问题不影响程序的正常运行和使用：
- 删除城市的函数 `void DelCity()` 只是把以该城市为起点的线路删了，并没有把以该城市为终点的线路删去。解决办法是遍历整个图进行删除。
 - 部分函数没有容错功能，如删除线路 `void DelLine()` 函数，默认删除的线路间的城市是存在的，即默认输入正确
 - 部分函数的容错做得不好，如用户查询最短路径时，若输入的城市不再城市列表中，程序只会提示有哪些城市并告知用户继续输入直到输入正确未知，不能退出返回上一级。管理员输入密码也是。
 - 在求最省时路径中可能求出的路径并不是真的最省时，例如，若起点城市到第一个确定最短路径的城市中有好几条线路，出发时间不同但用时相同，则很有可能影响到之后确定最短路径的城市的正确性。目前该问题仍然没有想到解决方案。在实际生活中，火车不存在这种情况，但飞机一般都存在。

八、总结

1、更加熟知一些编程细节：如

- `Fprintf()` 函数原型有三个参数，第一个是文件的指针，第二个是字符串，第三个是匹配字符串的变量名，实际上第三个是不一定需要的，和 `printf` 函数同理，当我们直接输出双引号里的内容，不带任何变量的话是不需要第二个参数的。
- `Exit()` 函数是终止正在执行的所有进程，中间的参数是用来告知系统和用户因为什么原因终止，0 的话代表正常终止。如果想结束某个函数的话用 `return` 即可。
- 在该 IDE 不能随用用 C++ 中的 `ERROR`，它不一定代表 -1

2、更加理解指针：

- 若结构体 A 中有指针 b，而 A 也是一个指针，那么为 A 开辟空间时 b 是没有任何空间的，如果想要往 b 中放内容，可以另外定义统一类型的指针 c 并开辟空间放入内容，然后让 b 指向 c，该情况下不能直接为 b 开辟空间。

- b. 指针就是指向某个内容的地址，本质上是一个地址，整型“没区别”；所以在调用函数时若改变了这个值，就必须返回它才会生效。改变指针本身的值就是在改变它指向的地方。只要是同一类型的指针，他能指向任何地方。
- c. 所谓为指针开辟空间，是说在系统中按照规定开辟一个空间，这个空间只有对应类型的指针可以指向它，访问它。
- d. 而所谓释放空间也不是说释放之后这个指针变量就消失了，而是说这块空间被系统回收了，之前指向这个空间的指针变成空指针了。
- e. 良好的定义指针习惯应该是定义之后立刻为其赋值 `NULL` 或立刻开辟空间。

3、要开始学会调用标准库来辅助编程，这样能节省时间同时提高代码效率。

4、对于这次课程设计，最大的难点在于迪杰斯特拉算法的实现，以往是用邻接矩阵的存储结构，而且书上有基于该存储结构的伪代码，所以实现起来很方便。采用邻接矩阵的核心是如何找出 `V-S` 集中离 `S` 集权值最小的点。该算法我们上网查找资料才得知可以采用优先队列这个结构进行辅助。实际上，就是一条有序链表，每次“删除”都是“删除”头结点，而插入则是有序插入。

5、《数据结构与算法》这门课其实很实用，关键是能不能想到可以用。数组、链表、队列、栈等都是很常用的结构，问题是我们在面对实际问题的时候能否通过对问题特点的描述进而想到它们。就例如上述迪杰斯特拉算法的问题，如果想到可以通过设置一条有序链表，每次“删除”都是“删除”头结点，而插入则是插入排序，我们是否真的会想到这和队列有类似的地方。这些都是这么课中学过的东西，但却不会想到。这也正是需要多编程的原因，提高自身对数据结构和算法的敏感度。

6、在主函数的编写中，由于需要输出的目录太多，屏幕大小有限，所以编写时感觉很凌乱。以后写程序时碰到代码较多行的函数可以考虑分割成几个小函数进行结构化设计，这样也有利于调试和修改。

7、可以尝试使用面向对象的方法，这次设计最后之所以使用类的思想是因为想把需要的东西封装，并不是真的把某一事物进行抽象。可以设计用户类，分管理员和普通游客，不同的用户带有不同的操作和属性，如对图的操作。