铁路交通查询程序解题报告

张鹏浩 00901001 赵靖康 00901002
王冬午 00901026　李骋 00901164

一、　问题描述和需求分析

题目要求简述：

自己设计数据结构和算法，开发一个全国城市间的铁路交通查询程序，为旅客提供 2-3 种最优决策的铁路交通查询。要求至少满足下列功能：

提供对城市信息、列车时刻表进行编辑的功能

至少提供两种最优决策选择：最快到达和最省钱到达，并输出各种有关信息

问题分析：

题目要求做出全国城市间的铁路交通查询程序，能够实现为用户提供乘坐列车的最优路线。由于用户对乘坐列车的要求可能不同，有些人可能希望快捷，而有些人则希望价格便宜或者换乘次数少，所以需要提供不同情况下的最优路线。这一部分问题的本质就是按照城市网络和线路图，并且结合用户的实际需求，寻找出最佳的路线然后输出给用户供用户参考。

另一部分问题是进行资料的更新，即更新列车和城市资料信息。应该能够管理所有的列车时刻信息和城市信息，而实际上有意义的城市应该是由列车而决定的，所以即是要求能够管理列车信息以及根据列车信息改变城市信息，并且当列车时刻信息发生改变的时候能够给用户提供改动之后的最优路线供用户选择。

设计思路：

为了实现提供给用户上述各种最优信息，采用的思路就是将全国的城市网络视作图，然后针对不同的需求采取不同方法的搜索方式：针对换乘最少，也就是经过结点最少的方式采用广度优先搜索，而针对时间最少和费用最少、距离最短，则采取深度优先搜索。最终根据对图的遍历得出最优解提供给用户。在呈现给用户的时候采用的是 MFC 界面的方式。

而为了实现管理列车信息，可以通过修改保存着列车时刻表信息的文本本件来完成。由于所有的信息都是存储在文本文件（相当于一个资料库）中的，所以改动了文本文件才能真正完成修改列车时刻信息和城市信息的工作。

二、 设计（数据结构和算法）

由于我们对算法进行了明确分工，因此对算法的设计也是各具特点。下面给出 5 方面具体的算法设计。

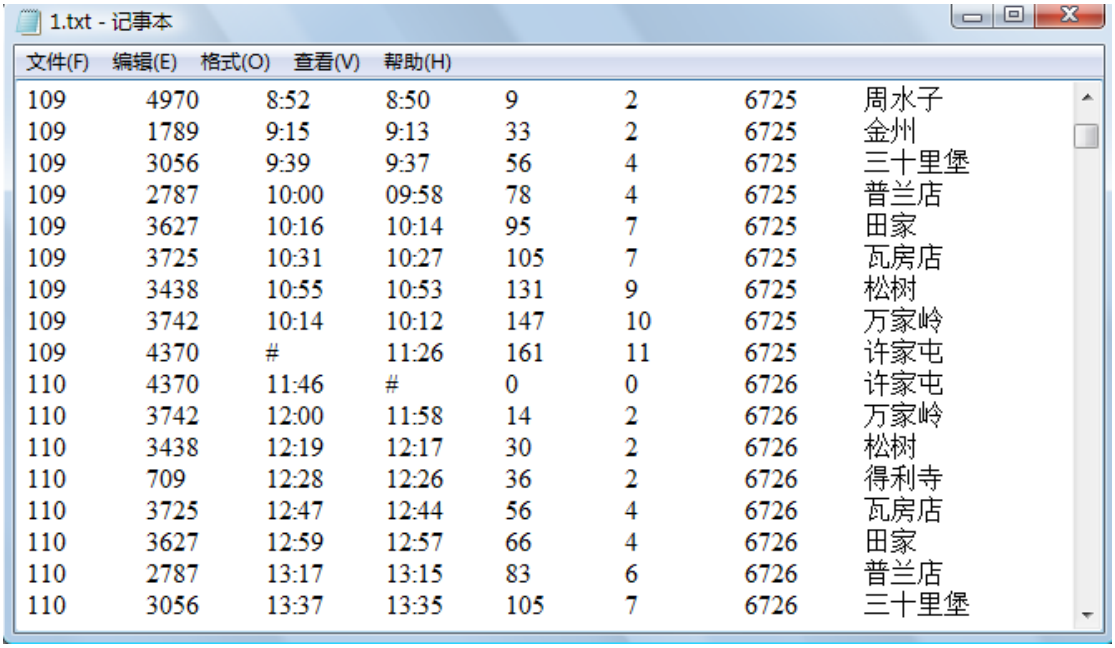
（1） 读入算法设计：

要成功将数据读入并存储，必须事先对数据库进行整理。我们首先从互联网上下载了一份信息十分全面的列车时刻表，再根据需要确定了要保留下来的信息，然后我们利用 EXCEL 对这份列车时刻表的数据进行了格式的规范，使之更方便我们读入，最后我们把 EXCEL 文件转换为文本文件使得它可以被程序所接受，修改后的文本文件如下图：

在读入数据时我们采用按字节读入的方式，由于在之前的准备工作中我们在不同信息之间用了制表符分隔，列车各站之间用了换行符分隔，并且在各起始以及终点站做了标记，所以读入过程比较方便，我们用类对数据进行了封装，具体形式如下：

```
class Railway{//车次，如 T501 次列车
public:int stationnum; //该车次所经过的站的总数
        int starthour[CITYNUM]; //从第 CITYNUM 站出发的小时
        int startminute[CITYNUM]; //从第 CITYNUM 站出发的分钟
        int arrivehour[CITYNUM]; //到达第 CITYNUM 站的小时
        int arriveminute[CITYNUM]; //到达第 CITYNUM 站的分钟
        double distance[CITYNUM]; //从起始站到第 CITYNUM 站的里程
        double price[CITYNUM]; //从起始站到第 CITYNUM 站的价格
        int time[CITYNUM]; //从起始站到第 CITYNUM 站的时间
        char id[20]; //车次名，如 T501
        int cityid[CITYNUM]; //途径第 CITYNUM 站的城市代号，其实就是下面 City[]的下标
（以下在函数部分说明）
        Railway(int num=0); //构造函数
        void getArrivetime(int& hour,int& minute,int CityID);
        int getStation(int CityID);
        int getTransitionTime(int fromid,int toid);
        void AddCity(int cn);
};
Class City//城市
{
public:char name[20]; //城市名，如北京
        int railway[600]; //途径该城市的火车，注意数组中存储的是火车的代号不是车次名，其实就是 Railway[]
的下标
        int rwnum; //途径该城市的火车数目
```

```
（以下在函数部分说明）
        City(); //构造函数
        void AddRailway(int rw);
};
```



1.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

109	4970	8:52	8:50	9	2	6725	周水子
109	1789	9:15	9:13	33	2	6725	金州
109	3056	9:39	9:37	56	4	6725	三十里堡
109	2787	10:00	09:58	78	4	6725	普兰店
109	3627	10:16	10:14	95	7	6725	田家
109	3725	10:31	10:27	105	7	6725	瓦房店
109	3438	10:55	10:53	131	9	6725	松树
109	3742	10:14	10:12	147	10	6725	万家岭
109	4370	#	11:26	161	11	6725	许家屯
110	4370	11:46	#	0	0	6726	许家屯
110	3742	12:00	11:58	14	2	6726	万家岭
110	3438	12:19	12:17	30	2	6726	松树
110	709	12:28	12:26	36	2	6726	得利寺
110	3725	12:47	12:44	56	4	6726	瓦房店
110	3627	12:59	12:57	66	4	6726	田家
110	2787	13:17	13:15	83	6	6726	普兰店
110	3056	13:37	13:35	105	7	6726	三十里堡

接下来我们只要编写函数 ReadData 将数据按字节读入并按类存储即可。

（2） 编辑算法设计：

a.基本分析：编辑是列车时刻表功能的重要组成部分，编辑功能的强弱直接决定了程序的可扩展性。本程序编辑功能采取直接对数据源（txt）编辑的方法，直接读取数据源（txt）文本中中内容，通过适当的处理整合成新的文本，再将其存回数据源（txt）。每次编辑后，只需重新通过 ReadData 函数读入数据源数据即可

b.编辑功能及基本原理：本程序提供四种编辑功能：删除指定车次；删除指定车次的指定站；向指定车次插入站；插入指定车次。

b.1 删除指定车次过程：原理相对最为简单，通过用户输入车次信息，将其与数据源中各行数据中车次项比较，相同则删除。

b.2 删除指定车次的指定站：在删除指定这次基础上改进而来，在其基础上对车站名一项同时进行比较，当两数据都符合时删除行。

b.3 向指定车次插入站：首先根据用户输入车次信息将相应车次所在行提出。对其他无关行进行简单复制操作；对提出行操作首先根据用户输入插入站的有关信息（车次中位置，与前后站距离、时间等）制作出插入站所在行并将其放

置于相应位置，然后对所在行后各行的距离、时间项依次进行调整。

b.4 插入指定车次：首先对列车编号进行检索，将空编号赋给新加列车，然后根据用户输入生成起始站行和终点站行，插于 txt 数据的最后。中间站各行数据可通过功能 3（向指定车次插入站）完成。

c.具体实现：

```
功能：添加列车//将 1.txt 写入 2.txt
用 fstream 将 1.txt 逐行读入并写入 2.txt//寻找线路，起止站的编号
ReadData();
ReadData();
for()
{       循环遍历所有路线找到第一个没有被占据的列车编号
}
记录编号
ReadData();
for()
{       遍历所有城市找到始发站编号}
记录编号
for(){   遍历所有城市找到终点站编号}
记录编号//写入 2.txt 具体更改内容
写入列车编号，起始站编号
将出发时间逐字符赋值到临时 char 数组并写入 2.txt，再写入“#”
写入距离和价格 0 0
将车次逐字符赋值到临时 char 数组并写入 2.txt
写入始发站编号对应的城市
写入列车编号，起始站编号
写入“#”，再将出发时间逐字符赋值到临时 char 数组并写入 2.txt
写入距离和价格 0
将车次逐字符赋值到临时 char 数组并写入 2.txt
写入终点站编号对应的城市//将 2.txt 写回 1.txt
用 fstream 将 2.txt 逐行读入并写入 1.txt

功能：删除指定列车的指定车站//查找列车和城市的编号
将输入的列车车次逐字符赋值成 char 字符数组
将输入的要删除的车站逐字符赋值成 char 字符数组
ReadData();
for(){   循环遍历所有路线找到要删除的车次对应的编号}
记录编号
ReadData();
for()
{       遍历所有城市找到要删除的城市对应的编号}
记录编号//判断并写入 2.txt
for{
```

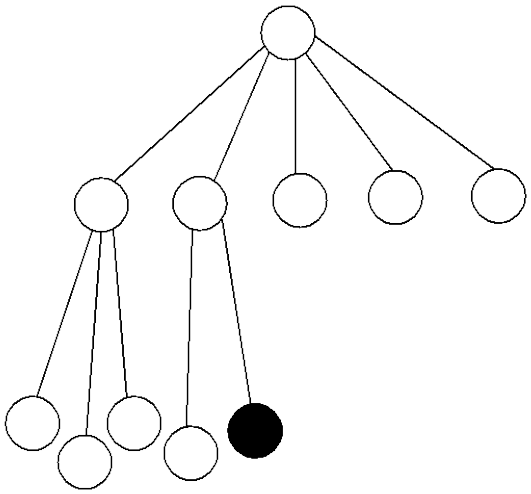
```
        用 fstream 读入 1.txt 中每一行
        if(如果城市字符串和车次字符串都不是一行的字串)
                写入 2.txt
}//将 2.txt 写回 1.txt
用 fstream 将 2.txt 逐行读入并写入 1.txt

功能：删除列车//判断并写入 2.txt
for{
        用 fstream 读入 1.txt 中每一行
        if(如果车次字符串不是一行的字串)
                写入 2.txt
}//将 2.txt 写回 1.txt
用 fstream 将 2.txt 逐行读入并写入 1.txt

功能：添加车站//定义
定义结构体 Read read;
内容为每一项的每一行存储的信息，有些信息有两种记录方式//找到目标车次
for{
        从 1.txt 逐行读入时刻表中的每一项内容
        if(车次是目标车次){
                将 read[1]记录为这一行
                跳出循环
        }
        else 写入 2.txt
}//找到目标城市为止和时间存储
用 trans_time 函数将 read[1]的时间字符串转化成具体时间并存入 read[1]中
for(kk){
        从 1.txt 逐行读入每一项内容并存入 read[kk]中
        if(是目标车次)
        {
                用 trans_time 函数将 read[kk]的时间字符串转化成具体时间并存入 read[kk]中
                if(是目标城市位置)
                        记录位置
                else
                        记录（最终记录成为这列车的最后一个站）
        }
}//找到新添加的城市编号或者新建编号
ReadData();
for(遍历所有城市){
        if(城市是目标城市)
                跳出
}
if(如果没有找到目标城市){
        for(遍历所有城市)
```

找到空的城市编号并记录
}
最终赋值目标城市
ReadData();
for(遍历所有列车){
if(目标列车)跳出
}
对 read[0]结构体逐项赋值，其中包括计算时间
for(目标位置以后的本车次城市){
计算结构体中每一项的变化
}//将修改后的目标车次写入 2.txt
for(目标车次的目标位置前的每一个城市){
写入 2.txt
}
read[0]写入 2.txt
for(目标车次的目标位置后的每一个城市){
写入 2.txt
}//将其余的写入 2.txt
for(目标车次以后的所有列车的所有行){
写入 2.txt
}//将 2.txt 写回 1.txt
用 fstream 将 2.txt 逐行读入并写入 1.txt

（3）最少转站算法设计：

a.基本分析：最少转站策略即经过最少的城市到达目的地。如果将出发地与目的地连线，即要使其连线上的城市结点尽量少。由于本程序的基本数据结构包含铁路和城市双节点，因此最少转站问题就简化成了一个简单的将城市作为元素的宽度优先的搜索问题。

b.逻辑结构：以城市为结点，出发城市的直达城市为子结点建立树。（如图）



第一层即为出发地，第二层为出发地的直达城市（即转站次数为 0 的城市），第三层为第二层的直达城市（即转站次数为 1 的城市）依次类推。因为采用的是宽搜方式，因此搜索将按层进行。即当目的地入队列时，此时记录的

通往目的地策略一定是最少转乘。

c.存储结构：首先，作为宽搜的基本储存结构，本程序建立了一个lesschangequeue队列进行数据存储，其存储元素信息（info）为城市编号，每次针对队头元素城市进行处理，将其直达城市入队列。同时，为了回溯方便，本程序针对城市设立“变形”循环单链表链表结构。每个链表节点存储信息（info）为城市编号。除单链表基础链接link1外加入链接link2进行过程记录。每个结点的初始Link2指向头结点。在这一链表条件下，只需在队列运算中改变link2的指向，使其指向上一中转站，通过回溯即可找到指定线路。最后，为了在记录经过城市的过程中直接记录下列车编号节省输出时的处理量，本程序设立了记录数组（putout）记录最少转站策略结果中到达每一换乘站（包括终点站）的列车编号。综上，本程序最少转站存储部分包括一个队列，一个变形单链表和一个记录数组。

```
//队列
struct Node;
typedef struct Node *PNode;
struct Node{ /* 结点结构*/
   int info;
   PNode link;
};
struct LinkQueue { /* 链接队列类型定义*/
   PNode f; /* 头指针*/
   PNode r; /* 尾指针*/
};
typedef struct LinkQueue *PLinkQueue;
/*链接队列类型的指针类型*/

//单链表
struct Nodeline; /*单链表结点类型*/
typedef struct Nodeline * PNodeline; /*结点指针类型*/
struct Nodeline { /*单链表结点结构*/
   int infoline;
   PNodeline link1;
   PNodeline link2;
};
typedef struct Nodeline * LinkList ; /*单链表类型*/
LinkList llist; /*llist是一个链表的头指针*/
```

d.基本操作：首先，将出发地城市入队列。之后针对队头元素进行以下操作：

提取队首元素（城市）-》游历经过此元素的全部列车-》针对每次列车将队首元素下标后的所有元素入队列，同时将以上元素的链表链接 link2 指向队首元素-》进行出队操作。重复上述操作，直至队首元素为目的地位置。

e.特殊处理：以上循环有一明显缺陷，即过程中将存在重复搜索。同一城市将可能在不同层出现并造成循环。这一过程将产生了巨大计算量。同时，初对该城市的第一次游历外，其余游历皆为无用计算（因为算法按层计算，因此之后游历所求得的换乘次数一定大于等于第一次游历的换乘次数）。为了解决这一问题，针对城市建立对应的开关数组 cityteam。初始数值为 0。当城市第一次游历时改为 1。同时在游历城市时加入判断，当相应 cityteam 数值为 1 时停止。有次即可解决以上问题，同时当两车站无法到达时在程序游历全部出发地可到达城市后也会因队列空而终止。

f.具体实现过程：（伪码）

```
主函数部分
int lesschange(参数含始发站和终点站城市编号) {
//预处理部分
        建立相应参量及数组
        读入数据
        创建单链表
        创建队列
        将城市编号排入单链表（单链表初始化）
        将数组赋值为 0（数组初始化）
        出发地入队列（队列初始化）
//处理部分
    while () {
          取队头元素
          if (队头元素非终点) {
                对此元素进行处理（eachcity 函数，后有此部分伪码说明）
          else 出 while 循环;
    }
//输出部分
        P=目的地（单链表中目的地城市定位）
         while (p 不为出发地) {
               p=p->link2(回溯)
         输出}

主要函数（eachcity）部分
void eachcitydeal (参数包括队列，单链表，记录数组，当前处理元素城市编号 citynum,记录列车数组 trainteam 及城市游历情况开关数组 cityteam) {
```

```
参数定义
单链表中处理城市定位
        while（依次考察经过该城市的各列列车）{
                当前处理城市在该次列车线路中定位
 for（依次处理列车线路中当前处理城市的后置城市（直达城市））{
 if（该城市未被游历过）{
 该城市入队列
        修改该城市在单链表中的 link2 链接至队首城市
        记录游历至该城市火车编号
                开关数组记录该城市以被游历
                }}}
队首（已处理元素）出队列}
```

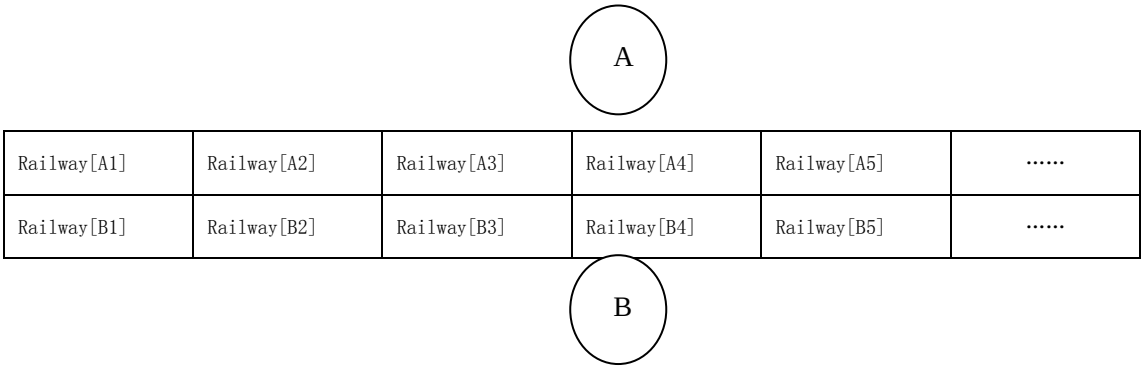
（4）最低花费算法设计：

所谓最省钱到达，就是乘客从指定的起始站到终点站所花费的价钱最少。其中，在价钱相等的情况下，乘客肯定倾向于选择换乘次数少的方案。基于此，我们设计了我们组的最省钱到达算法。

首先我们给出基本假设，假设数据中任意两个站之间最多经过两次换乘必可到达（即最多只需乘坐三列火车抵达目的地）。这是基于目前我国的铁路系统已经比较发达，我们可以直观的认为任意两个城市之间的行车方案可以按照如下所示：起始站——枢纽城市 A——枢纽城市 B——终点站。所以我们的算法中，分别考虑了直达、换乘一次、换乘两次的情况。我们定义了一个 double 类型变量 min，用来存储不同方案的最少价钱。而且由于在价钱相等的情况下，乘客倾向于选择换乘次数更少的方案，所以我们的算法中先初始化 min 为一个最大值 MAXMONEY=5000，之后从直达的方案开始算起，再是换乘一次和换乘两次，每次用得到的价钱与 min 比较，如果比 min 小，就更新 min 的值，并存储对应的方案。

下面考虑直达的算法：

将起始站 A 中的所有火车（所有途经 A 的火车）和终点站 B 中的所有火车进行比较，如果两车编号相同，则表明该火车途径 A、B 两城市（如下图）。再考虑 A、B 在该列火车中的位置，如果 A 在 B 前，则表明火车先经过 A 后经过 B，符合条件。反之，如果 B 不在 A 后，那么火车先经过 B 后经过 A，不符合条件。当两车编号相同且先后顺序正确的话，计算该列火车中 A、B 的价格差，与 min 比较。如果比 min 小，则更新 min 并记录该火车的信息。




```
double leastmoney0(int begin, int end, track train_result){//不用换乘的最少价钱算法
    double min = MAXMONEY;//初始化最少价钱
    for(起始站中查找火车){
        for(终点站中查找火车){
            if(两城市有同样的列车经过){
                如果 begin 在 railway[j]中的编号在 end 的编号之前且价格差小于 min
                更新 min 并记录列车信息
            }
        }
    }
}
```

接下来讨论换乘一次的算法：

考虑从起始城市出发查找其能直接通往的所有城市，按编号顺序存入一个数组中。同理，从终点站出发查找能够直接到达终点站的所有城市，也存入一个数组中。分别遍历这两个数组，如果二者编号相同，说明该城市可作为中转站。分别计算两段的价格，将其和与 min 进行比较，如果小于 min，则将 min 更新，并记录两列火车和中转站的信息。

首先需要编辑一个函数能够查找一个城市能直接通往的所有城市(或者能够直接到达该城市的所有城市)。从该城市的第一列火车开始遍历该站以后的所有站，如果不在要存入的数组中，就将其存入数组，直到找完经过该城市的所有火车。存入时可用二分插入法存储，这样在后面搜索对比时可以节省时间。

```
int adjacentcity1(int cityid, int* a){//计算一个城市能到达的所有城市，按城市编号由小到大记录在数组 a 中
    初始化数组
    找到第一个能到达的城市，并将其放入数组
    if(没找到这样的城市)return 失败;
    继续存入其他城市（二分法插入）
```

同理可得出查找能直接到达某个城市的所有城市的算法

得到了起始站和终点站的直达城市数组，就可以将它们一一比较。如果相同

则计算两段分别的价钱，并与 min 比较。小于 min 则更新 min 并存储列车和中转站记录。

```
if(adjacentcity1(begin, zhongzhuanzhan1)+adjacentcity2(end, zhongzhuanzhan2)<=0)
//起始站都是经过该站的火车的终点或者终点站都是经过该站火车的起点
    for(两个数组一起查找){//一次换乘到达
        if(中转城市相同){
            比较两段价钱之和与 min
        if(小于 min){
          更新最小价钱并记录火车和中转站
            }++i; ++j;
        }
        else if(数组 1 中城市编号小于 2 中城市编号)++i;
        else ++j;
    }（注：上面并没有单独用一个函数给出，而是属于 leastmoney 函数的一部分）
```

下面来讨论两次换乘的情况：

与一次换乘类似，先找出起始站和终点站的直达城市数组，之后数组之间任两城市用直达算法计算，将三段价格加起来与 min 比较，如果小于 min 则更新 min 并且记录三列火车和两个中转站。

```
（前面紧接一次换乘到达，所以直达数组已给出）
for(搜索起始城市的直达城市中){//两次换乘到达
        for(在终点站的直达城市中搜索){
            if 起点到中转城市 1 的价钱+中转城市 2 到终点的价钱已经超过最小价钱，不用继续考虑
            else{
            考虑三段价钱之和与 min 的大小
            if(小于 min){
                    更新最小值并记录
                }
            }
        }
}
```

最后将上面三种算法和在一起，再加上对结果的处理，就构成了 leastmoney 函数。

（5）最少时间算法设计：

采用城市辐射搜索法，从起始城市开始，按经过该城市的列车搜索。每辆列车经过该城市后能到达的站再以此类推，继续搜索。这种方法的原始算法很慢，但是经过合理的剪枝以后运行速度能提高很多。剪枝方法就是判断累积时间+两城市间的行驶时间+等待时间是否大于最小时间，如果大于则继续搜索火车或者

城市，小于则更新变量。

下面是伪码实现（以两次换乘情况为例）：

```
int Treetimes(int from,int to,int starthour,int startminute,unitInfo* save,int accutime){
 初始化最小时间
   for(从起始城市搜索每列火车){
      for(从该列火车的该站往后搜索)     {
            if(等待时间+行驶时间+累积时间>最小时间)continue;
            该时间+=由中转站到终点站的最小时间
            if(该时间<最小时间){
                  更新最小时间，记录火车、中转城市
            }
      }
 }
}
```

三、 具体程序实现

（1） 读入程序实现：

由于我们是按字节读入数据，而具体存储时的数据类型显然不都是字符，因此我们需要对数据类型进行转换。

```
double str_to_db(char* a) //将字符串转换为实数
void turn(char *a,int start,int end) //倒转两个字符间的字符的顺序
void db_to_str(double b,char* a) //将实数转换为字符串
void TranstoTime(int& h,int& m,char *a) //将字符串转换为时和分
void ReadData() //读入数据并把信息存到 City 类和 Railway 类中去
```

（2） 编辑程序实现：

```
int trans_time(char *p);//将时间字符串转化成分钟数
ReadData();//读入数据，目的是查找城市和列车的编号以及输出城市和列车的字符串
```

（3） 最少转站程序实现：

（a）队列操作

```
PLinkQueue createEmptyQueue_link( void ) //创建空队列
int isEmptyQueue_link( PLinkQueue plqu（目标队列） )//判断队列是否为空
void enQueue_link( PLinkQueue plqu（目标队列）, int x（要插入信息） ) //进队列
void deQueue_link( PLinkQueue plqu（目标队列）) //出队列
int frontQueue_link( PLinkQueue plqu（目标队列）) //取队列头部元素的值
```

（b）单链表操作

```
LinkList createNullList_link(void) //创建一个带头结点的空单链表
int isNullList_link(目标单链表) //判断单链表是否为空
```

（c）实际操作函数

```
void relistlink(LinkList llist（目标链表）,int citynum（城市总量）) //单链表预处理（对所有城市建立单
   链表链接）
void eachcitydeal (LinkList llist（单链表）,int citynumzph（目标城市编号,PLinkQueue plqu（队列）,int
   cityteamzph[]（记录各城市游历情况开关数组）,int trainteamzph[]（记录经过城市列车数组）) //对单个
   城市的处理
```

int LeastChange(int startcity(出发地城市编号), int finalcity（目的地城市编号）) //完成整个计算并输出

（4） 最低花费程序实现：

int zhongzhuanzhan1[CITY], zhongzhuanzhan2[CITY];//存放直达城市的两个数组

int Railway::getStation(int CityID);//在某条铁路中查找编号为 CityID 的城市的位置

double leastmoney0(int begin, int end, track train_result);//不用换乘的最少价钱算法（起始站编号，终点站编号，记录火车的指针）

int adjacentcity1(int cityid, int a[]);//计算一个城市能到达的所有城市，按城市编号由小到大记录在数组 a 中

int adjacentcity2(int cityid, int a[]);//计算能到达该城市的所有城市，按城市编号由小到大记录在数组 a 中

double LeastPrice(int begin, int end, track train_result1, track train_result2, track train_result3, city_wdw stop1, city_wdw stop2);//计算最少价钱（起始站编号，终点站编号，第一列火车信息，第二列火车信息，第三列火车信息，中转站 1 信息，中转站 2 信息）

（5） 最少时间程序实现：

int getTimeInterval(int starthour,int startminute,int endhour,int endminute)//两时间点之间的时间差

int Railway::getTransitionTime(int fromid,int toid)//两站间的行驶时间（起始城市，终点城市）

int Once(int from,int to,int starthour,int startminute,unitInfo *save,int accutime)//不换乘最短时间算法（起始站编号，终点站编号，出发小时，出发分钟，存储结果指针，累积时间（剪枝用））

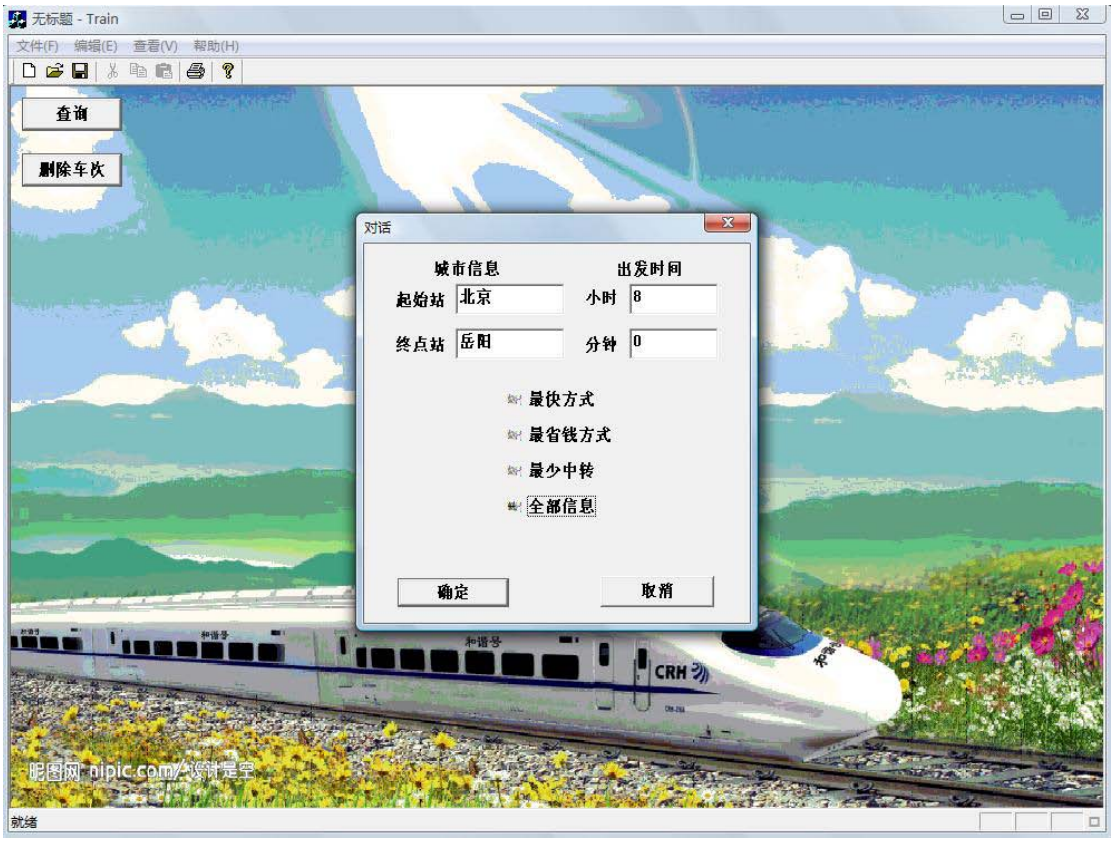
int Twice(int from,int to,int starthour,int startminute,unitInfo* save,int accutime)//一次换乘最短时间算法（起始站编号，终点站编号，出发小时，出发分钟，存储结果指针，累积时间（剪枝用））

int Treetimes(int from,int to,int starthour,int startminute,unitInfo* save,int accutime)两次换乘最短时间算法（起始站编号，终点站编号，出发小时，出发分钟，存储结果指针，累积时间（剪枝用））

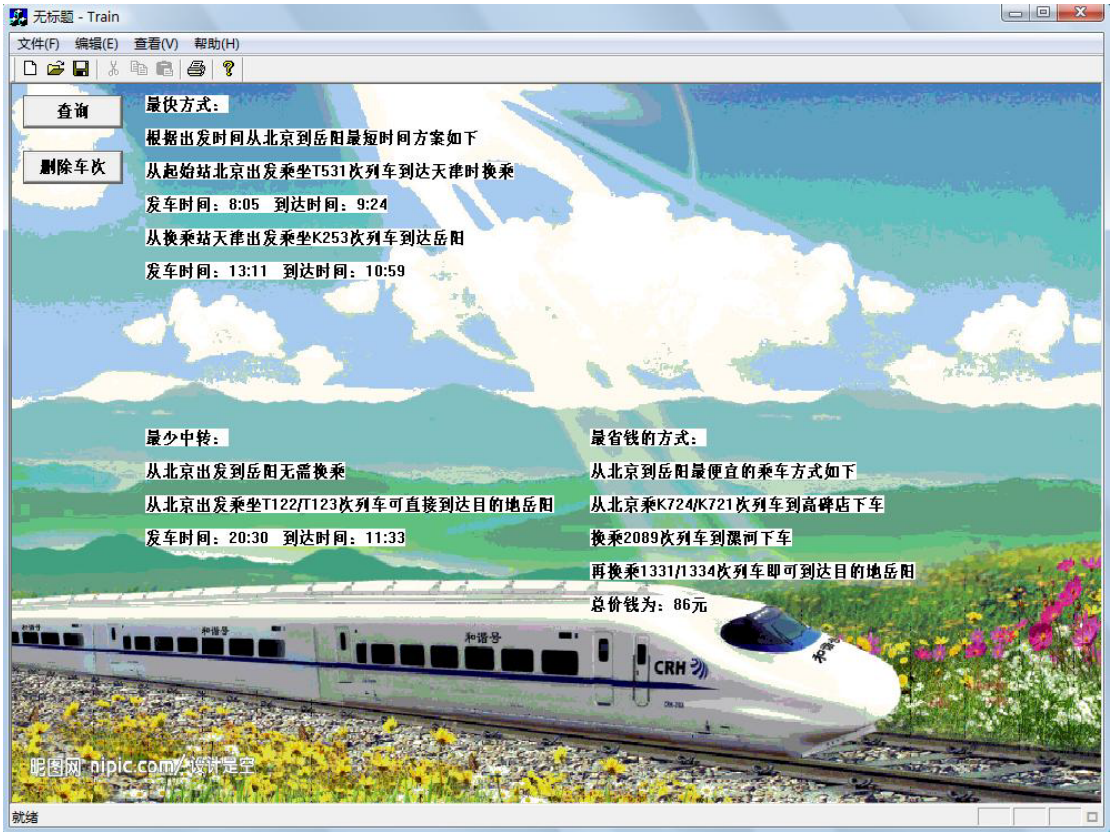
四、 开发环境的界面设计

为了增强程序的可操作性，我们利用 MFC 的开发环境进行了界面设计，主要用到了按钮和对话功能，用户打开程序，首先看到的界面如下：

点击查询进入查询对话框



输入相关数据后，我们即可得到相应结果，这里我们选择全部结果查看



这样我们就得到美观而清晰的结果。

五、 调试情况

由于我们本身的编程能力有限，对 c 语言的掌握也有不足之处，初次编完的程序往往含有大量的语法错误，这方面的修正就花费了大量时间。当然，其中也有一些更加典型的修正过程。如在编辑最省钱算法过程中，刚开始我们采用的是从开始城市起辐射搜索法，即从起始城市搜索每一列火车，每一列火车中再搜索从该城市之后的城市，再继续搜索下去。比如不换乘搜索就是搜索经过始发站的所有火车，每一列火车中从始发站往后搜索看有没有终点站，如果有，将价钱差和 min 比较，小于 min 则更新。一次换乘和两次换乘同理可得。后来读入数据调试的时候发现，这个算法花费的时间实在是太长了，常常要搜索 1 分钟左右才能给出结果。后来发现是这个算法本身时间代价太高（详见第六部分），所以经过了我们的讨论换成了现在的算法。我们组有位同学曾评价说：“这两个算法就像是改革开放前后的中国一样，可谓是发生了翻天覆地的变化。”另外，在编辑功能开发过程中我们也经历了一段艰难的调试过程。由于我们的界面设计应用到了 mfc，由此就牵扯到了大量的格式转换和输入要求的问题，而这方便问题在不停需要读入输出并与用户交互的编辑功能中显得格外突出。比如在读入用户输入数

据时，mfc 仅能提供包括 cstring 在内的部分输入格式，而缺乏对 char 类型的支持，虽然可以通过循环依次读入数组元素的方式进行赋值后进行比较，但这种方法对中文就失效了。因此在制作输出行等步骤中我们饱受折磨，不得不多次通过 ReadData 函数中的城市编号与名称的转换功能进行操作。虽然繁琐，但我们最终还是达到了目标。

六、 算法与数据结构的分析与评价

为了进一步认识评估自己的程序效率，我们对其中几个主要部分进行了细致的测试分析与评价。

（1） 读入程序部分：

这部分的算法没有什么特殊的，就是最简单的按字节读入并存储以及一些常见的转换函数。我们在数据存储方面为了节约空间，事先对数据进行统计从而保证了在开数组时不会浪费大量存储空间。

（2） 编辑程序部分：

（a）理论分析：由于编辑过程主要涉及到两次数据源的整体移动过程，因此这 2o(n)的处理量构成了编辑程序的时间代价主体。同时，4 项编辑功能中 3 项应用了 ReadData 函数，其会带来 o(n)的运算量。而 ReadData 后得到的数据主要应用于城市名与城市编号以及火车车次与火车编号的配对上，每一次配对的 for 循环带来 o(n)的代价。由于对每行数据的处理都不牵扯到循环过程，因此在其基础上不会进一步扩充代价。故编辑程序的总时间代价大概在 a*o(n)(a<10)，不同编辑更能所需代价不同，但都在 o(n)的层次上。

(b)数据选择：选取位于 txt 文件上中下三种位置的数据进行编辑

（c）检测方式：对各种编辑方式插入时间函数 clock()进行计时，并通过任务管理器观察 cpu 和内存变化

（d）试验结果：

上部数据	所花时间	Cpu 变化	内存变化
插入站	4.8700s	51%	50MB
插入车次	1.6300s	48%	54MB
删除站	0.8000s	54%	59MB
删除车次	0.3200s	54%	48MB

中部数据	所花时间	Cpu 变化	内存变化
插入站	4.3400s	56%	67MB
插入车次	1.4300s	48%	64MB
删除站	0.9000s	46%	68MB
删除车次	0.4400s	55%	60MB
下部数据	所花时间	Cpu 变化	内存变化
插入站	6.8300s	58%	59MB
插入车次	1.5200s	56%	62MB
删除站	0.9800s	56%	60MB
删除车次	0.4400s	55%	62MB

（3） 最少转站程序部分：

(a)理论推理：设循环一次数据量为 o(n)，分析结构建立和处理过程，结构建立中用到数组赋值，数据量为 o(n)；处理过程中 while 循环由转站次数决定，设为 a，在每个 while 循环中，if 语句为破除循环功能，无数据量作用，之后的 eachcity 函数中包含一个 while 语句，while 中又包含两个 for 语句，因此数据量为 $o(n)^2$。因此总的数据量应为 a $o(n)^2$。因为转站次数远小于数据量，因此数据量级等于 $o(n)^2$。

(b)数据选择：测试采用针对最少转站算法的单独测试。选取数据包括以下几组：（1）北京到上海（从交通枢纽到交通枢纽，检测第一层大容量的直达情形工作效率）（2）北京到龙家堡（从交通枢纽到地方小站，实现转站功能检测）（3）龙家堡到北京（从地方小站到交通枢纽，与第 2 组对照，检测效率差异）（4）龙家堡到舍利屯（从地方小站到地方小站，检测转站情况单一但转站次数偏多的工作效率）

(c)检测过程：

时间代价：设置 startcity（出发地）与 finalcity（目的地）参数，将各组数据带入，通过在读入完成，建立队列与链表完成，处理完成及输出完成四个点建立时间函数进行测试。

空间代价：设置 startcity（出发地）与 finalcity（目的地）参数，将各组数据带入，通过任务管理器监控 cpu 及内存变化。

(d)实际测试结果：

时间代价：

原始数据：(记录数据均为测试 5 次均值)

	起始	至读完数据	至定义完结构	至处理完	至输出
第一组	0.0000s	2.7616s	2.8766s	2.9126s	2.9140s
第二组	0.0000s	2.6362s	2.7508s	2.9056s	2.9086s
第三组	0.0000s	2.5818s	2.6876s	2.7112s	2.7138s
第四组	0.0000s	2.8126s	2.9180s	3.0696s	3.0730s

经整理：

	起始	读数据	定义结构	处理	输出
第一组	0.0000s	2.7616s	0.1150s	0.0360s	0.0014s
第二组	0.000s	2.6362s	0.1146s	0.1548s	0.0030s
第三组	0.000s	2.5818s	0.1058s	0.0236s	0.0026s
第四组	0.000s	2.8126s	0.1054s	0.1516s	0.0034s

结论：d.1 经过核对，输出信息为正确信息；

d.2 读数据和创建结构部分和输出部分时间差异不大；

d.3 处理过程 1,3 组效率明显高于 2,4 组。

空间代价：

第一组：初始 cpu 为 4%，内存 0.98GB；峰值 cpu 为 52%，内存 1.07GB；最终 cpu 为 4%，内存 0.98 GB

第二组：初始 cpu 为 4%，内存 0.98GB；峰值 cpu 为 51%，内存 1.07 GB；最终 cpu 为 4%，内存 0.98 GB

第三组：初始 cpu 为 4%，内存 0.98 GB；峰值 cpu 为 43%，内存 1.06 GB；最终 cpu 为 4%，内存 0.98 GB

第四组：初始 cpu 为 4%，内存 0.98 GB；峰值 cpu 为 48%，内存 1.07 GB；最终 cpu 为 4%，内存 0.98 GB

结论：d.4 运行过程对 cpu 及内存有影响。

d.5 内存变化四组数据基本一致。

d.6cpu 变化第三组数据低于其他三组数据。

(e)结果分析：

e.1 对时间代价的试验中，由于读数据和创建结构部分和输出部分为 4 组数据的通用时间。因此相差不大可以解释。处理过程中 1 组时间段在于存在直达，仅进行了 1 层运算；第 3 组时间低于第 2 组时间原因在于从小站出发直通点少，即每

层元素相对从大站出发的元素少。因此处理速度更快。第 4 组时间长的原因则在于转站次数较多，层数增加。因此，时间代价与转站次数及经过站的大小有关。总结起来：转乘少速度快；经过的站小速度快。

e.2 对空间代价的试验中，内存变化主要由读入数据引起，变化不大说明处理过程的空间占用量稳定。Cpu 第 3 组较低的原因为从小站出发直通点少，因此所需 cpu 少。

（4）最低花费程序部分：

在生成直达城市数组的过程中，二分法插入时每步时间代价为 o(lni/ln2)，总共时间代价为 o（lnn!/ln2）<o(nlnn/ln2)<o(n^2)，而 o(n^2)是乱序搜索时的时间代价，可见二分插入法生成直达城市数组有明显的时间代价。此外，最省钱方案搜索的时候的时间代价与两次换乘的时间代价有关，而两次换乘时最坏情况是，一个直达城市数组的每个元素与另一个数组中每个元素都有直达方案，总时间代价为 o(n^2)*o(n^2)=o(n^4)。

刚开始的时候我们用的是从开始城市起辐射搜索法，即从起始城市搜索每一列火车，每一列火车中再搜索从该城市之后的城市，再继续搜索下去。所以这种算法两次换乘的情况下时间代价为 o(n^6)，经过一些简化之后其时间代价仍超过了 o(n^5)。由此可见我们目前所采用的最省钱算法的优越性。(如下图对比)

最省钱算法(旧)								
	起始站	终点站	读入数据时间(s)	查找结果的时间(s)	CPU 使用率		物理内存	
					运行前	运行中	运行前 GB	运行中 GB
1	北京	武穴	4.676000	70.196000	9%	54%	1.05	1.11
2	北京	济南	4.571000	52.370000	1%	52%	1.04	1.10
3	鹰潭	齐齐哈尔	4.577000	97.521000	6%	55%	1.05	1.10

最省钱算法								
	起始站	终点站	读入数据时间(s)	查找结果的时间(s)	CPU 使用率		物理内存	
					运行前	运行中	运行前 MB	运行中 MB
第一次	北京 259	武穴 3947	4.699000	0.094000	4%	56%	934	989
第二次			4.665000	0.109000	5%	53%	933	988
第三次			4.649000	0.093000	4%	54%	933	989
平均值			4.671000	0.098667	4%	54%	933	989
第一次	北京	济南	4.682000	0.328000	4%	59%	930	991

第二次	259	1678	4.867000	0.343000	4%	65%	931	986
第三次			4.649000	0.328000	6%	54%	931	986
平均值			4.732667	0.333000	5%	59%	931	988
第一次	鹰潭 4609	齐齐哈尔 2821	4.692000	0.499000	5%	57%	881	935
第二次			4.634000	0.514000	6%	57%	883	939
第三次			4.634000	0.499000	5%	54%	883	937
平均值			4.653333	0.504000	5%	56%	882	937

唯一的一点不足是我们数据读入过程中每个城市的火车不是按照编号排列的，如果按照编号排列，在直达算法中我们的时间代价将会由 o(n^2)提高到 o(n)，总的时间代价也会提高为 o(n^3)。

（5）最少时间程序部分：

理论上来说，该算法的时间代价应该为 o(n^6)，但是经过一系列的剪枝，算法运行速度已经和前两种算法差不多了。但这其中的具体原因以及该算法和我们先前的最省钱算法的异同还需要我们进一步研究。

七、上机实习总结

从最初接到题目任务到最终完成任务，在思路方面都是四个人在一起进行讨论，对不同的意见进行分析然后确定最终的策略方法，而在具体操作过程中，四个人分工明确，在初期各自编写自己的程序部分并且各自调试，最后进行了程序的接合，最终完成了程序。

虽然四个人都尽所能的完成程序任务，程序还是有不尽人意的地方，比如在进行搜索的时候，虽然经过不断的努力优化算法，但是由于数据量庞大所以运行速度与成熟的列车时刻表查询系统还有一定的差距，如果继续进行数据结构与算法的学习并且不断改进程序应该能够不断地提高运行速度。

另一方面，提供给用户的界面虽然极大地方便了用户的操作，尽量的做到人性化，但是与成熟的界面相比也有一定的差距。比如有的时候有许多最优方案(比如同时是两次换乘最省价钱的方案)，而我们显示结果的时候只能显示一个最优方案，所有最优方案不能同时显示出来。下次再做类似的问题的时候可以把所有最优方案都输出出来，让用户自己去选择。当然继续学习 MFC 的相关内容或者采用更高级的编程方式也可以进一步优化操作界面。

由于时间所限，程序有一定不足之处可以改进，但是四个人在各自编程、相互讨论和最终程序接合的过程中共同提高，熟练了对数据结构与算法的掌握，并

对软件工程的思想有了一定的体会，获益颇丰。

八、合作分工

张鹏浩：共同问题分析，讨论数据结构，中转算法，添加删除城市、列车，报告相关部分，综合报告内容；

赵靖康：共同问题分析，讨论数据结构，最省时算法，读入算法设计，程序调试，MFC 实现，报告相关部分；

王冬午：共同问题分析，讨论数据结构，查找数据，最省钱算法，报告相关部分；

李骋：共同问题分析，讨论数据结构，设计数据结构，添加删除城市、列车，MFC 实现。