**《数据结构课程设计》报告**

班级:1620001班

学号:162020321

姓名:朱家震

指导教师:孙涵

目录

1. 区块链（必做）（链表）……………………………………3
2. 迷宫问题（必做）（栈与队列）……………………………16
3. JSON查找（必做)（树）…………………………………24
4. 公交线路提示 (必做)（图）………………………………32
5. Hash表应用（必做）（查找）……………………………52
6. 排序算法比较（必做）（排序）……………………………66
7. 【2】地铁修建 (选做)（图）……………………………79
8. 【2】URL映射（选做）（字符串）………………………82
9. 【3】连连看（选做）（图）………………………………86
10. 【4】B-树应用（选做）（查找）…………………………94
11. 【1】算术表达式求值 (选做)（栈）…………………108

十二、结束语……………………………………………………116

1. 区块链（必做）（链表）
2. 数据结构

该题是明显的链表结构实现相关功能。

**struct** informa

{

**int** number;

    string info;

**int** CheckCode;

    informa()

    {

        number = 0;

        info = "aaa";

        CheckCode = 0;

    }

};

**typedef** **struct** Block

{

    informa inf;

    Block\* next;

}Block,\*BlockChain;

这里用informa表示每一个结点的基本信息，包括编号、字符串以及校验码。

1. 算法设计思想

这里判断是否有效，只需要第一个结点判断校验码是否等于字符串ASCII码之和mod113，之后的依次判断校验码是否等于(上一个结点的校验码+本结点的结点编号+本结点信息中字符串ASCII码）mod 113即可。第一个不符合的结点即为首个无效结点的编号。

添加信息时，添加在链表末尾，加上并计算校验码即可。

篡改某结点信息时，需要先找到篡改结点的编号，如果没有相应的编号，则输出没有相应的结点。如果找到了，则读取修改后的结点信息，并修改之后的校验码，修改校验码的方法与创建区块链的方法一致，直到第一个符合规则的结点，篡改结束。

1. 源程序

BlockChain.h文件：

#include<iostream>

#include<fstream>

#include<string>

**using** **namespace** std;

**struct** informa

{

**int** number;

    string info;

**int** CheckCode;

    informa()

    {

        number = 0;

        info = "aaa";

        CheckCode = 0;

    }

};

**typedef** **struct** Block

{

    informa inf;

    Block\* next;

}Block,\*BlockChain;

**int** CalAscll(string s);

**void** ReadData(fstream& file, BlockChain& c);

**void** UpdataData(BlockChain c);

BlockChain Find(BlockChain c, **int** des);

**void** Is\_Valid(BlockChain c);

**void** Insert(BlockChain& c);

**void** Revise(BlockChain& c);

**void** Print\_All(BlockChain c);

#pragma once

BlockChain.cpp文件：

#include<iostream>

#include<fstream>

#include<string>

#include"Blockchain.h"

**using** **namespace** std;

//计算ascll值

**int** CalAscll(string s)

{

**int** sum = 0;

**for** (**int** i = 0; i < s.length(); i++)

        sum += s[i];

**return** sum;

}

//判断有效

**void** Is\_Valid(BlockChain c)

{

    BlockChain p = c;

**if** (CalAscll(p->inf.info) % 113 == p->inf.CheckCode)

    {

**while** (p->next)

        {

**int** temp;

            temp = p->inf.CheckCode + p->next->inf.number + CalAscll(p->next->inf.info);

**if** (temp % 113 == p->next->inf.CheckCode) p = p->next;

**else**

            {

                cout <<"第一个无效结点的编号为:"<< p->next->inf.number<<endl;

                cout << "校验码应该为:" << temp % 113<<endl;

                cout << "是否进行自动修改?(y/n)";

**char** q;

                cin >> q;

**if** (q == 'y')

                {

                    cout << "正在修改中，请稍后....." << endl;

                    BlockChain f = p;

                    p = p->next;

                    temp = (f->inf.CheckCode + p->inf.number + CalAscll(p->inf.info));//篡改结点使合理

**while** (temp % 113 != p->inf.CheckCode)

                    {

                        p->inf.CheckCode = temp % 113;

                        f = p; p = p->next;

**if** (p == NULL) **break**;

                        temp = (f->inf.CheckCode + p->inf.number + CalAscll(p->inf.info));

                    }

                    cout << "修改完成！" << endl;

**return**;

                }

**else** **return**;

            }

        }

        cout << "文件无误，有效！" << endl;

**return**;

    }

**else**

    {

        cout <<"第一个无效结点的编号为:"<< p->inf.number << endl<<"校验码应该为:"<< CalAscll(p->inf.info) % 113 << endl;

        cout << "是否进行自动修改?(y/n)";

**char** q;

        cin >> q;

**int** temp= CalAscll(p->inf.info) % 113;

        p->inf.CheckCode = temp;

**if** (q == 'y')

        {

            cout << "正在修改中，请稍后....." << endl;

            BlockChain f = p;

            p = p->next;

            temp = (f->inf.CheckCode + p->inf.number + CalAscll(p->inf.info));//篡改结点使合理

**while** (temp % 113 != p->inf.CheckCode)

            {

                p->inf.CheckCode = temp % 113;

                f = p; p = p->next;

**if** (p == NULL) **break**;

                temp = (f->inf.CheckCode + p->inf.number + CalAscll(p->inf.info));

            }

            cout << "修改完成！" << endl;

**return**;

        }

**else** **return**;

    }

}

//读取数据

**void** ReadData(fstream& file, BlockChain& head)

{

    file.open("data.txt", ios::in | ios::out);

**if** (file.fail())

    {

        cout << "文件打开失败!";

        exit(0);

    }

    file >> head->inf.number;

    file.ignore();

    getline(file, head->inf.info);

    file >> head->inf.CheckCode;

    BlockChain c = head;

**while** (!file.eof())

    {

        BlockChain p = **new** Block;

        p->next = NULL;

        file >> p->inf.number;

        file.ignore();

        getline(file, p->inf.info);

        file >> p->inf.CheckCode;

        c->next = p;

        c = p;

    }

    file.close();

}

//将更新的数据写入文件

**void** UpdataData(BlockChain c)

{

    fstream file;

    file.open("temp.txt", ios::in | ios::out | ios::app);

    BlockChain p = c;

**while** (p->next)

    {

        file << p->inf.number << endl;

        file << p->inf.info << endl;

        file << p->inf.CheckCode << endl;

        p = p->next;

    }

    file << p->inf.number << endl;

    file << p->inf.info << endl;

    file << p->inf.CheckCode ;

    file.close();

    system("del data.txt");

    system("ren temp.txt data.txt");

}

//查找编号相同的结点

BlockChain Find(BlockChain c, **int** des)

{

    BlockChain p = c;

**int** sign = 0;

**while** (p)

    {

**if** (p->inf.number == des) { sign = 1; **break**; }

        p = p->next;

    }

**if** (sign)

**return** p;

**else** **return** NULL;

}

//添加

**void** Insert(BlockChain& c)

{

    BlockChain p = c;

**while** (p->next) p = p->next;//找到末尾结点

    BlockChain n =**new** Block;

    cout << "请输入新插入的结点的编号:";

**int** num;

    cin >> num;//输入结点编号

**while** (Find(c, num) != NULL)

    {

        cout << "编号重复！请重新输入编号:";

        cin >> num;

    }

    cout << "请输入新插入的结点的信息：";

    string information;

    cin.ignore();

    getline(cin, n->inf.info);//输入结点信息

    n->inf.number = num;

    n->inf.CheckCode = (p->inf.CheckCode + n->inf.number + CalAscll(n->inf.info)) % 113;

    cout << "该结点的校验码为" << n->inf.CheckCode << endl;

    p->next = n;

    n->next = NULL;

}

//修改

**void** Revise(BlockChain& c)

{

**int** num;

    cout << "输入需要更改的信息编号:";

    cin >> num;

    BlockChain p,f=c;

    p = Find(c, num);

**while** (!p)

    {

        cout << "未找到该编号！请重新输入编号:";

        cin >> num;

        p = Find(c, num);

    }

**while** (f->next != p && f->next)f = f->next;

    string s;

    cout << "请输入修改后的结点信息:";

    getchar();

    getline(cin, s);

    p->inf.info = s;

**int** temp = (f->inf.CheckCode + p->inf.number + CalAscll(p->inf.info));//篡改结点使合理

**while** (p && temp % 113 != p->inf.CheckCode)

    {

        p->inf.CheckCode = temp % 113;

        f = p; p = p->next;

**if**(p)

            temp = (f->inf.CheckCode + p->inf.number + CalAscll(p->inf.info));

    }

}

//打印所有

**void** Print\_All(BlockChain c)

{

    BlockChain p = c;

    cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

**while** (p)

    {

        cout << "编号为:" << p->inf.number <<endl ;

        cout << "信息为:" << p->inf.info << endl;

        cout << "校验码为:" << p->inf.CheckCode << endl;

        p = p->next;

        cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

    }

}

区块链.cpp文件：

#include<iostream>

#include<string>

#include<stdlib.h>

#include<stdio.h>

#include <windows.h>

#include"Blockchain.h"

**using** **namespace** std;

**int** main()

{

    fstream datafile;

    BlockChain c=**new** Block;

    ReadData(datafile, c);

**while** (1)

    {

        cout << "请输入需要进行的操作:" << endl;

        cout << "1、判断是否有效" << endl;

        cout << "2、添加信息" << endl;

        cout << "3、篡改信息" << endl;

        cout << "4、显示所有内容" << endl;

        cout << "5、退出并保存" << endl;

**int** cho;

        cin >> cho;

**switch** (cho)

        {

**case** 1: {Is\_Valid(c); system("pause"); system("cls"); **break**; }

**case** 2: {Insert(c); system("pause"); system("cls"); **break**; }

**case** 3: {Revise(c);  system("pause"); system("cls"); **break**; }

**case** 4: {Print\_All(c); system("pause"); system("cls"); **break**; }

**case** 5: {UpdataData(c); **return** 0; }

        }

    }

**delete** c;

**return** 0;

}

1. 测试数据和结果

所用的data.txt：

1

b

98

2

b

85

3

d

75

4

zzz

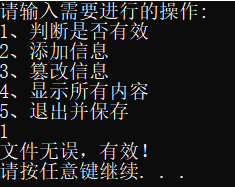
106

6

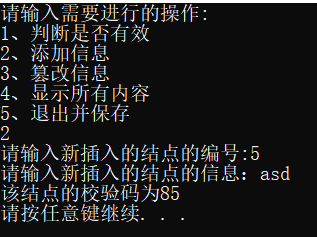
l

107

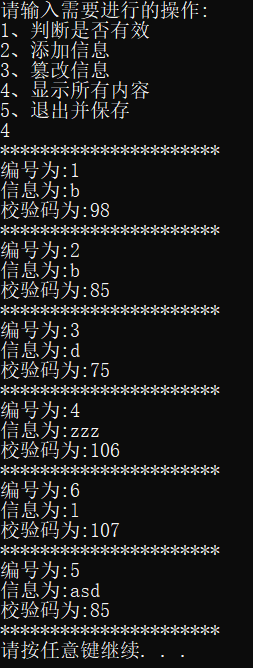
判断有效操作：



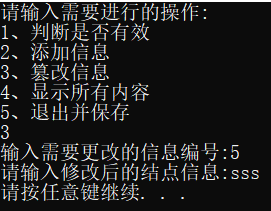
添加操作：



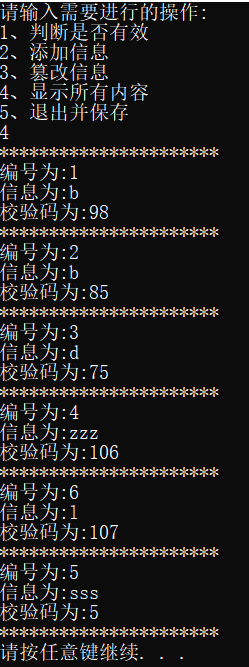
显示所有操作：



篡改结点操作：



篡改之后所有结点信息：



1. 算法的时间复杂度分析

该题中，无论是篡改、添加还是有效性检查都是遍历一遍所有的链表结点实现的，因此时间复杂度均为O(N)。

1. 算法的改进方法

我认为在有效性检查中如果无效，加入自动修改功能可能可以增加程序功能的完善性与人性化。

比如把数据文件改成:

1

b

98

2

b

83

3

d

75

4

zzz

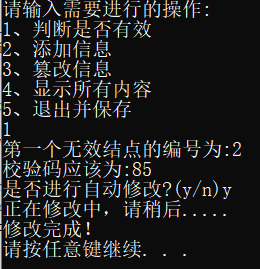
106

6

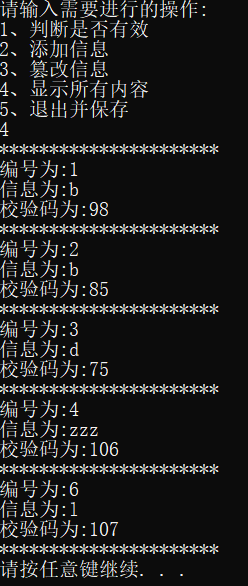
l

107

再进行有效性检查就为：



这时，修改完成之后文件内容就修改成了：



1. 迷宫问题（必做）（栈与队列）
2. 数据结构

该题中采取的数据结构主要是栈的运用。

**typedef** **struct** Point

{

**int** x;

**int** y;

**int** ap;

}Point;

**typedef** Point ElemType;

**typedef** **struct** SeqStack

{

    ElemType\* pBase;

    ElemType\* pTop;

**int** stacksize;

}SeqStack;

这里的Point是迷宫的位置坐标，x行y列。Ap表示该点之后往哪个方向前进。如果ap=0，就是往上走；ap=1，往右走；ap=2，往下走；ap=3，往左走。一开始的时候ap都为0，假设一开始均为向上走。

1. 算法设计思想

利用栈的思想。首先将n\*n的矩阵存入(n+1)\*(n+1)的数组中，为0的起始位置不放置，方便理解与表达。

首先输入起点，然后从起点开始寻找上右下左哪个方位是可以访问的，如果可以访问就将其迷宫数组置为2，表示已访问，如果一个方位不行，就换下一个方位。如果有一个方位可以访问，则将该位置入栈；如果没有位置可以访问，就将该位置出栈。直到找到终点结束程序。

1. 源代码

Stack.h文件：

#define STACKINITSIZE 256

#define STACKINCREMENT 128

**typedef** **struct** Point

{

**int** x;

**int** y;

**int** ap;

}Point;

**typedef** Point ElemType;

**typedef** **struct** SeqStack

{

    ElemType\* pBase;

    ElemType\* pTop;

**int** stacksize;

}SeqStack;

**void** InitStack(SeqStack& S);

**void** DestroyStack(SeqStack &S);

**void** ClearStack(SeqStack&S);

**bool** StackEmpty(SeqStack &S);

**int** StackLength(SeqStack&S);

**void** GetTop(SeqStack S,ElemType &e);

**void** Push(SeqStack &S,ElemType e);

**void** Pop(SeqStack &S,ElemType &e);

#pragma once

Stack.cpp文件：

#include<iostream>

#include"Stack.h"

**using** **namespace** std;

//初始化栈

**void** InitStack(SeqStack &S)

{

    S.pBase = (ElemType\*)malloc(STACKINITSIZE \* **sizeof**(ElemType));

**if** (S.pBase == NULL) { cout << "申请内存失败！|0"; **return**; }

    S.pTop = S.pBase;

    S.stacksize = STACKINITSIZE;

}

//顺序栈销毁

**void** DestroyStack(SeqStack& S)

{

**if** (S.pBase != NULL)

    {

        free(S.pBase);

        S.pBase = NULL;

    }

    S.pTop = NULL;

    S.stacksize = 0;

}

//顺序栈清空

**void** ClearStack(SeqStack& S)

{

    S.pTop = S.pBase;

}

//判断是否为空

**bool** StackEmpty(SeqStack& S)

{

**return** S.pBase == S.pTop;

}

//返回栈中元素个数

**int** StackLength(SeqStack& S)

{

**return** S.pTop - S.pBase;

}

//返回栈顶元素

**void** GetTop(SeqStack S, ElemType& e)

{

**if** (S.pBase == S.pTop) { cout << "空栈，无栈顶元素!\0"; **return**; }

    e = \*(S.pTop - 1);

}

//压栈

**void** Push(SeqStack& S, ElemType e)

{

**if** (S.pTop - S.pBase >= S.stacksize)

    {

        S.pBase = (ElemType\*)realloc(S.pBase, (S.stacksize + STACKINCREMENT) \* **sizeof**(ElemType));

**if** (S.pBase == NULL) { cout << "申请内存失败！\0"; **return**; }

        S.pTop = S.pBase + S.stacksize;

        S.stacksize += STACKINCREMENT;

    }

    \*S.pTop = e;

    S.pTop++;

}

//弹栈

**void** Pop(SeqStack& S, ElemType& e)

{

**if** (S.pBase == S.pTop) { cout << "空栈，无法弹出！\0"; **return**; }

    e = \*(S.pTop - 1);

    S.pTop--;

}

迷宫问题.cpp文件：

#include<iostream>

#include<fstream>

#include<windows.h>

#include"Stack.h"

**using** **namespace** std;

#define N 50

**int** DATA[N][N];

**int** col, row;

**int** start\_x, start\_y, end\_x, end\_y;

//读取迷宫

**void** ReadFile(fstream& file, **char**\* filename)

{

**int** i = 1, j = 1;

    file.open(filename, ios::in);

**if** (file.fail())

    {

        cout << "读取文件失败!";

        exit(0);

    }

    file >> row >> col;

**while** (!file.eof())

    {

        file >> DATA[i][j] ;

**if** (j == col)

        {

            i++;

            j = 1;

        }

**else** j++;

    }

    file.close();

}

//寻找路径

**void** GetPath()

{

    SeqStack s;

    InitStack(s);

    Point start = { start\_x,start\_y ,0};

    Point end = { end\_x,end\_y ,0};

    DATA[start.x][start.y] = 2;             //起点记为已访问

    Point t;

    Push(s, start);

**while** (!StackEmpty(s))

    {

        Pop(s, t);

**int** adjacent = t.ap;

**if** (t.x == end.x && t.y == end.y)   //到终点

        {

            DestroyStack(s);

**return**;

        }

**int** find\_flag = 0;

**int** next\_i, next\_j;

**while** (adjacent < 4 && find\_flag == 0)               //寻找可以访问的位置

        {

**switch** (adjacent)

            {

**case** 0: {next\_i = t.x - 1; next\_j = t.y; **break**; }//上

**case** 1: {next\_i = t.x; next\_j = t.y + 1; **break**; }//右

**case** 2: {next\_i = t.x + 1; next\_j = t.y; **break**; }//下

**case** 3: {next\_i = t.x; next\_j = t.y - 1; **break**; }//左

            }

**if** (DATA[next\_i][next\_j] == 0 && next\_i >= 1 && next\_i <= row && next\_j >= 1 && next\_j <= col)//如果没出界并且可以走

                find\_flag = 1;

            adjacent++;//记录下一个可以走的方位

        }

**if** (find\_flag)//找到可以走的方向

        {

            t.ap = adjacent;

            Push(s, t);

            DATA[next\_i][next\_j] = 2;//将可以走的方位标记为2，并入栈

            Point temp = { next\_i,next\_j,0 };

            Push(s, temp);

        }

**else**

        {

            DATA[t.x][t.y] = 0;//不可以走，重置为0

        }

    }

    DestroyStack(s);

}

**int** main()

{

    fstream file;

**char** filename[20] = "Labyrinth.txt";

    ReadFile(file, filename);

    cout << "请输入起点坐标:";

    cin >> start\_x >> start\_y;

    cout << "请输入终点坐标:";

    cin >> end\_x >> end\_y;

    cout << row << " " << col << endl;

    GetPath();

**for** (**int** i = 1; i <= row; i++) {

**for** (**int** j = 1; j <= col; j++) {

**if** (DATA[i][j] == 2)

            {

                SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED);

                cout << "0 ";

            }

**else**

            {

                SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);//设置三色相加

                cout << DATA[i][j] << " ";

            }

        }

        printf("\n");

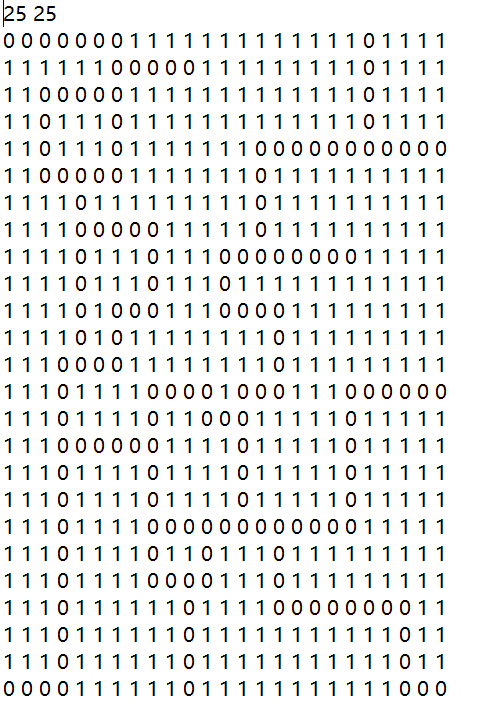
    }

**return** 0;

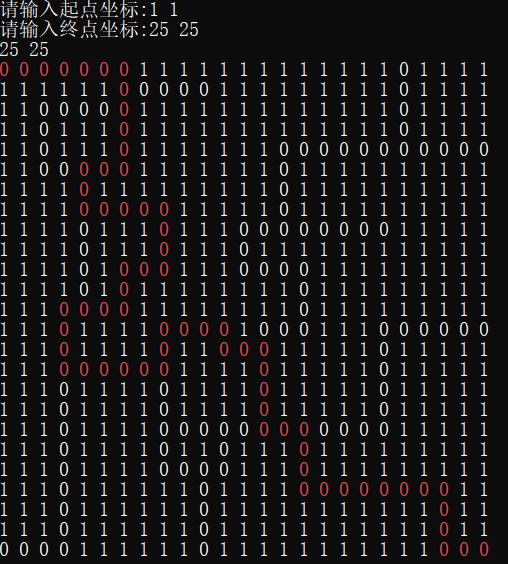
}

1. 测试数据和结果

迷宫数据文件内容为（0代表可以走，1代表不能走）：



运行结果：



1. 算法的时间复杂度

本题的时间复杂度主要来源于计算哪个方位可以走，每个点最多判断四次，因此时间最复杂的情况为4\*n^2,因此时间复杂度为O(n^2)。

1. 算法的改进方法

这个方法算出来的迷宫显然不是最优解，要想得到最优的迷宫解法，我觉得可以每走一步将长度加一，计算长度。到达终点之后回溯，到上一个岔路口，判断该岔路口下一个方向能否到达终点，再比较两次的路径长度，并将较短的路径经过的点存入一个栈中，最后打印的时候只要根据这个栈的路径打印即可获得迷宫的最优解。

1. JSON查找（必做）（树）

1.数据结构

本题采用孩子-兄弟树的方法来存储数据值。

#include <iostream>

#include<string>

**using** **namespace** std;

**typedef** **struct** TreeNode

{

    string type;//类型

    string keyvalue;//键值

**struct** TreeNode\* firstchild, \* nextbrother, \* farther;

}TreeNode,\*Tree;

**void** InitTree(Tree& T);

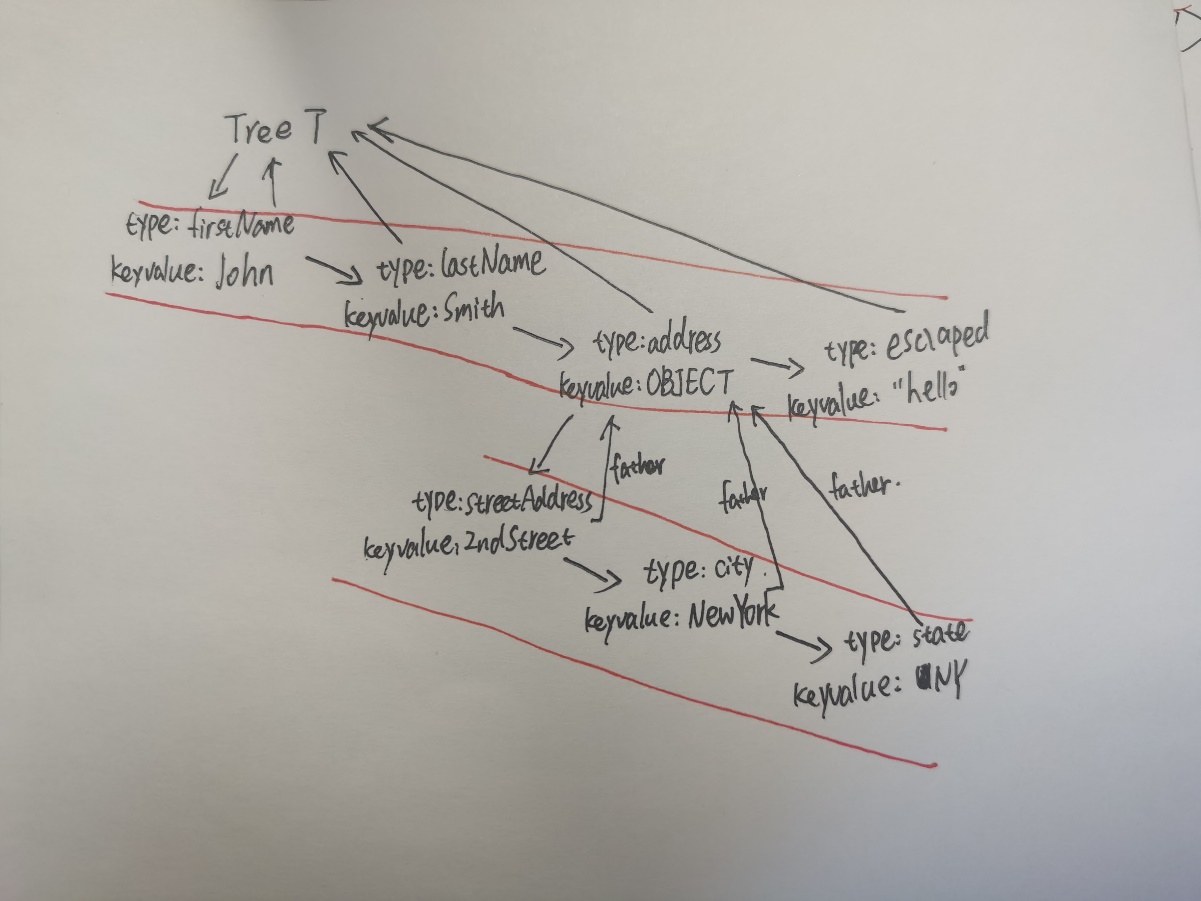
**void** format(string& s);

**void** Search\_json(Tree T, string s);

**void** CreatTree(Tree& T);

#pragma once

用图画出来大致是：



2.算法设计思想

这题我将同属于一个对象下面的类型作为树一层的兄弟，将对象中的对象作为结点的孩子。

分析所给数据，可以大致分为五种类型：

1.{ 只有一个左括号，这一定是第一行的数据，所以遇到时，需要建立树的一个孩子。

2.”type”:”keyvalue”,这是正常的键值，冒号前面是它的属性名，冒号后面的引号是他的属性内容，这样的类型需要我们新建一个兄弟结点将这一行的内容存进兄弟中。

3.”type”:{ 这说明这是一个对象属性，需要我们新建一个兄弟结点，将type存到结点的type之中，将结点的keyvalue记为”BOJECT”，并标记，告诉下一个循环时，应该在该结点的孩子之下插入操作。

4. }, 这代表着一个对象的结束，说明一个结点的孩子以及读取完成，需要重新返回上一层新建它的兄弟结点。

5. } 这代表着全部读取完成。

对于查找，输入一行数据时可以查找是否有进行分割的“.“如果有点，说明需要先在本层次中找到type等于点之前的结点，进入该结点的孩子查找，如果没有了点，那么就说明就在本层次中查找即可。如果到最后查找的指针是NULL，说明没有找到；如果没有孩子指针，说明是 “STRING”；如果有孩子，说明是个对象”OBJECT”。

3.源代码

Tree.h文件：

#include <iostream>

#include<string>

**using** **namespace** std;

**typedef** **struct** TreeNode

{

    string type;//类型

    string keyvalue;//键值

**struct** TreeNode\* firstchild, \* nextbrother, \* farther;

}TreeNode,\*Tree;

**void** InitTree(Tree& T);

**void** format(string& s);

**void** Search\_json(Tree T, string s);

**void** CreatTree(Tree& T);

#pragma once

Tree.cpp文件：

#include"Tree.h"

#include"Queue.h"

#include <iostream>

#include<string>

#include<fstream>

**using** **namespace** std;

//构造一个空二叉树

**void** InitTree(Tree& T)

{

    T = **new** TreeNode;

    T->farther = NULL;

    T->type = "OBJECT";

    T->firstchild = NULL;

    T->nextbrother = NULL;

**if** (T == NULL) { cout << "新建时空间分配错误！\n" << endl; **return**; }

}

//删除转义字符

**void** format(string& s)

{

    //把字符串s中的转义字符\ 删除，并保留其下一个字符

**for** (**int** i = 0; i < s.size(); i++)

    {

**if** (s[i] == '\\')

        {

            s.erase(s.begin() + i);

        }

    }

}

**void** Search\_json(Tree T, string s)

{

**int** pos = s.find('.', 0);

    Tree json = T;

    string s\_part;

**while** (pos != -1)//循环寻找分割的  .  然后找到的话就说明是进入孩子

    {

        json = json->firstchild;//进入孩子寻找 . 后面的类型

        s\_part = s.substr(0, pos); //将  .  前面的字符串切割出来，寻找相应的类型

        //cout << s\_part << " ";

**while** (json)//寻找与  .  前面一致的类型

**if** (json->type == s\_part)

**break**;

**else** json = json->nextbrother;

        s.erase(0, pos+1);//将  . 以及前面的都扔掉，寻找  .  后面的类型

        pos = s.find('.', 0);

    }

    //由于最后一遍是最后一个  .  后面的还有没有查找的，所以需要再来最后一遍

    json = json->firstchild;

    s\_part = s.substr(0, pos);

    //cout << s\_part << " ";

**while** (json)

**if** (json->type == s\_part)

**break**;

**else** json = json->nextbrother;

**if** (json != NULL && json->firstchild == NULL) // 没有孩子，也就是不是个类

        cout << "STRING " << json->keyvalue << endl;

**else** **if** (json != NULL && json->firstchild != NULL) //有孩子，是个类

        cout << json->keyvalue << endl;

**else** cout << "NOTEXIST" << endl;  // 不存在

    //cout << s << endl;

}

//输入树

**void** CreatTree(Tree& T)

{

    string s;

    fstream file;

    file.open("JSON.txt", ios::in);

**if** (file.fail()) { cout << "open failed!" << endl; exit(0); }

**int** n, m;

    file >> n >> m;

    file.ignore();

**int** sign = 0; // 判断是否是一个对象

    Tree p = T;

    Tree fa = T;// 记录父亲是哪个

**for** (**int** i = 0; i < n; i++)//n行数据读取

    {

        getline(file, s, '\n');

**if** (s.length() == 1 && s[0] == '{')//第一行，只有一个{

        {

            sign = 1;

**continue**;

        }

**if** (s.length() == 2 && s[0] == '}')//中间的对象结束有},两个,那么返回上一层

        {

            p = fa;

            fa = fa->farther;

**continue**;

        }

**if** (s.length() == 1 && s[0] == '}')//最后一行结束 有 }

**break**;

**int** k;

        string KeyType;//读取引号中的属性名

        string Key;//读取属性内容

**for** (**int** j = 0; j < s.length(); j++)//一行字符串中分割类型和值

        {

**if** (s[j] == '"')//从引号开始

            {

                k = s.find(':', j + 1);//查找: 的位置

                KeyType = s.substr(j + 1, k - j - 2);//截取类型字符串

                format(KeyType);

**int** len = s.length();

**int** pos = s.find('"', k);//从:开始找键值的" 如果为-1说明后面没有" 也就是后面是一个对象

**if** (pos == -1)//没有,这是个对象,新建一个对象结点

                {

                    sign = 1;

                    Tree bro = **new** TreeNode;

                    p->nextbrother = bro;

                    bro->farther = fa;

                    bro->type = KeyType;

                    bro->keyvalue = "OBJECT";

                    bro->firstchild = NULL;

                    bro->nextbrother = NULL;

                    p = bro;

                }

**else**//这不是对象

                {

**int** pos\_end = len;//键值的位置

**while** (s[pos\_end] != '"') pos\_end--;

                    Key = s.substr(pos + 1, pos\_end - pos - 1);//截取键值字符串

                    format(Key);

**if** (sign)//前一个是{ 要新建一个孩子

                    {

                        Tree child = **new** TreeNode;

                        child->farther = p;

                        child->firstchild = NULL;

                        child->keyvalue = Key;

                        child->nextbrother = NULL;

                        child->type = KeyType;

                        fa = p;

                        p->firstchild = child;

                        p = child;

                        sign = 0;

                    }

**else**//前一个不是{ 新建一个兄弟

                    {

                        Tree temp = **new** TreeNode;

                        temp->farther = fa;

                        temp->firstchild = NULL;

                        temp->keyvalue = Key;

                        temp->nextbrother = NULL;

                        temp->type = KeyType;

                        p->nextbrother = temp;

                        p = temp;

                    }

                }

**break**;

            }

        }

    }

    //进行查找

    string f;

**for** (**int** i = 0; i < m; i++)

    {

        getline(file, f, '\n');

        Search\_json(T, f);

    }

    file.close();

}

JSON查找.cpp文件：

#include<iostream>

#include<string>

#include<fstream>

#include"Tree.h"

**using** **namespace** std;

**int** main()

{

    Tree T;

    InitTree(T);

    CreatTree(T);

**return** 0;

}

4.测试数据和结果

样例输入

10 5

{

"firstName": "John",

"lastName": "Smith",

"address": {

"streetAddress": "2ndStreet",

"city": "NewYork",

"state": "NY"

},

"esc\\aped": "\"hello\""

}

firstName

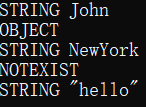
address

address.city

address.postal

esc\aped

输出：



5.算法的时间复杂度

该算法的时间复杂度主要由遍历查找以及新建产生的，因此时间复杂度为O(n)。

1. 公交线路提示（必做）（图）

1.数据结构

这题的数据结构用的是图的邻接表表示法。

**struct** Node\_Arc //存储边

{

**int** adjvex;//边到达的站点

**int** road;//公交路数

    Node\_Arc\* Next\_Arc;//下一条边

};

**struct** Bus\_Station//存储公交站台

{

**char**\* name=**new** **char**[50];

    Node\_Arc\* First\_Arc;//存储第一条边

};

**typedef** **struct** line\_sta//记录第i路公交的所有站点

{

**char**\* name = **new** **char**[50];

    line\_sta\* next;

    line\_sta\* pre;

}line\_sta;

**typedef** **struct** BusGraph

{

    Bus\_Station\* Station=**new** Bus\_Station[5200];

    line\_sta\* lu[1000];

**int** vexnum, arcnum;

    BusGraph()

    {

        vexnum = 0; arcnum = 0;

**for** (**int** i = 0; i < 1000; i++) lu[i] = NULL;

    }

}BusGraph;

**typedef** **struct** Node\_Line//公交站路线的邻接

{

    Node\_Arc\* firstarc;

}Node\_Line;

**typedef** **struct** Line//求最短换乘用这个

{

    Node\_Line l[1000];

**int** vexnum, arcnum;

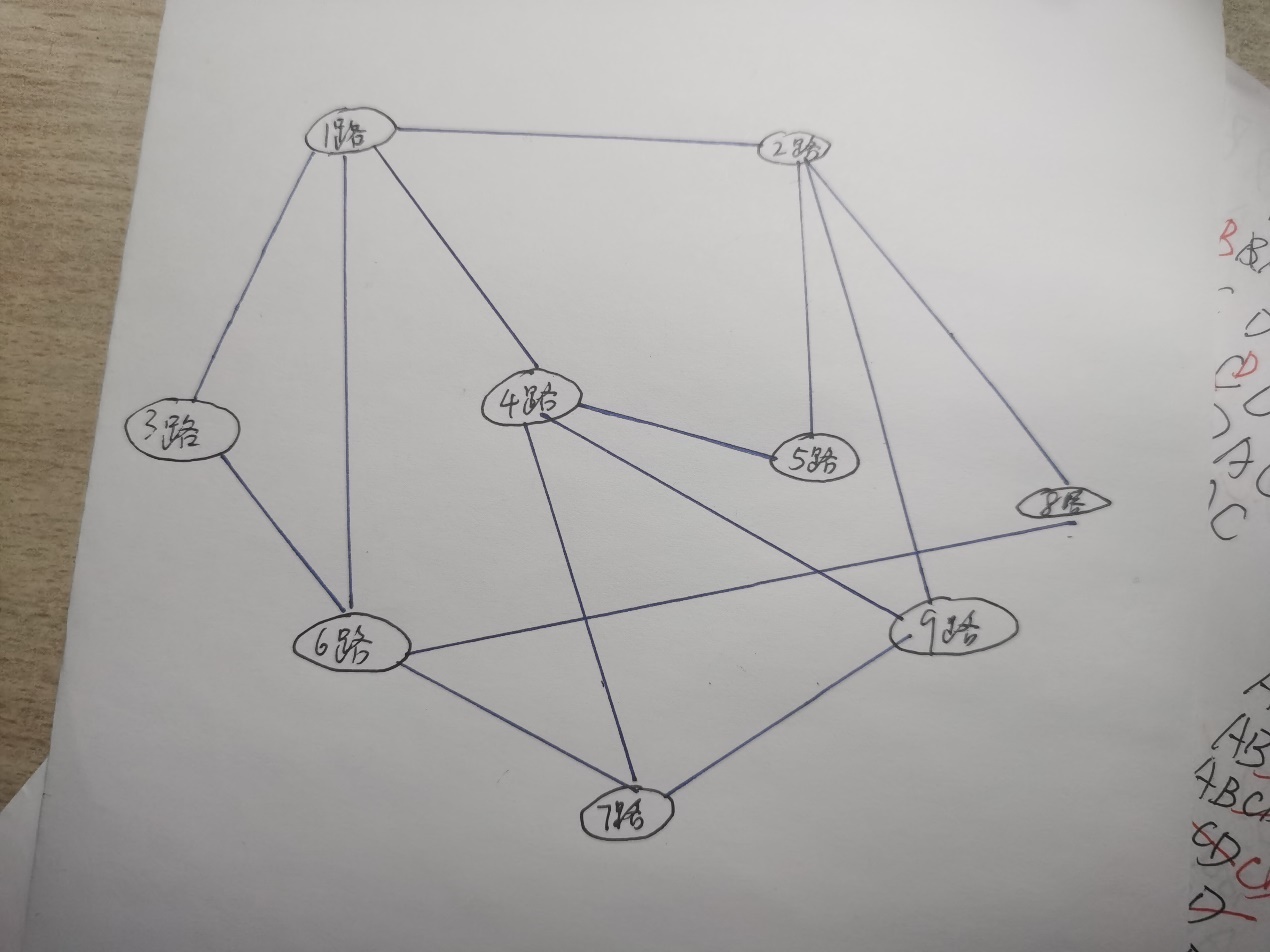
**int**\* visit\_lint;

}Line;

2.算法设计思想

对于经过站点最少的方案，很明显是广度优先算法，当到达终点时就输出。

对于转车次数最少的方案，首先需要建立关于公交路线的邻接表图，如果两个公交路线可以互相转乘，那么就在这两个路线之间连一条边。比如：



假如一路公交和二路公交存在同一个站点，那么他们就是可以互相转乘的，就在他们之间连一条边，以此类推，假设有上面的路线图。

我们需要的起点可以乘坐：1，2，3路公交；终点可以乘坐:7,9路公交。

那么我们可以知道转乘的方案有很多种：

1->4->7、2->5->4->9、2->9等等。

显然可以看出最少换乘的就是2->9的方案。

因此，最短换乘可以利用公交路线的邻接表的广度优先算法先求得从起点站先乘坐哪一路车，最后通过哪一路车到终点换乘最少。然后根据这两个路线广度优先中间的换乘路线找到相应的方案，具体方法如下：

首先第一个路线是起点站所在路线，找到下一个路线，在起点站所在路线中找到可以换乘到下一条路线的站点，在该站点换乘即可。

3.源代码

BusGraph.h文件：

#include<iostream>

#include<string.h>

#include<fstream>

#define STACKINITSIZE 256

#define STACKINCREMENT 128

**struct** Node\_Arc //存储边

{

**int** adjvex;//边到达的站点

**int** road;//公交路数

    Node\_Arc\* Next\_Arc;//下一条边

};

**struct** Bus\_Station//存储公交站台

{

**char**\* name=**new** **char**[50];

    Node\_Arc\* First\_Arc;//存储第一条边

};

**typedef** **struct** line\_sta//记录第i路公交的所有站点

{

**char**\* name = **new** **char**[50];

    line\_sta\* next;

    line\_sta\* pre;

}line\_sta;

**typedef** **struct** BusGraph

{

    Bus\_Station\* Station=**new** Bus\_Station[5200];

    line\_sta\* lu[1000];

**int** vexnum, arcnum;

    BusGraph()

    {

        vexnum = 0; arcnum = 0;

**for** (**int** i = 0; i < 1000; i++) lu[i] = NULL;

    }

}BusGraph;

**typedef** **struct** Node\_Line//公交站路线的邻接

{

    Node\_Arc\* firstarc;

}Node\_Line;

**typedef** **struct** Line//求最短换乘用这个

{

    Node\_Line l[1000];

**int** vexnum, arcnum;

**int**\* visit\_lint;

}Line;

**void** Add\_Arc(BusGraph& Graph, **int** station\_a, **int** station\_b, **int** road);

BusGraph Init\_BusGraph();

**int** Find\_Station(BusGraph G, **char**\* des);

**void** Destroy\_BusGraph(BusGraph& Graph);

**bool** Is\_Adj(BusGraph G,**int** i, **int** j);

**void** Less\_Station(BusGraph Graph, **char**\* begin, **char**\* end);

**void** Create\_arc(**int** i, **int** j, Line& r);

**void** InitLine(BusGraph G, Line& r);

**void** Destory\_Line(Line& r);

**bool** Is\_Adj(Line r, **int** i, **int** j);

**void** Print\_Min\_Change(BusGraph G, Line r, **int** i, **int** j, **char**\* start, **char**\* end);

**void** Min\_Change(Line r, BusGraph G, **char**\* beg, **char**\* en);

#pragma once

BusGraph.cpp文件：

#pragma warning(disable : 4996)

#include<iostream>

#include<string.h>

#include<fstream>

#include"BusGraph.h"

#include"Queue.h"

#include"Stack.h"

**using** **namespace** std;

**int**\* visit;

//添加边

**void** Add\_Arc(BusGraph& Graph, **int** station\_a, **int** station\_b, **int** road)

{

    Node\_Arc\* Arc;

    Arc = Graph.Station[station\_a].First\_Arc;

**if** (Arc == NULL)//如果没有邻接点

    {

        Graph.Station[station\_a].First\_Arc = **new** Node\_Arc;

        Graph.Station[station\_a].First\_Arc->adjvex = station\_b;

        Graph.Station[station\_a].First\_Arc->road = road;

        Graph.Station[station\_a].First\_Arc->Next\_Arc = NULL;

    }

**else**

    {

**while** (Arc->Next\_Arc != NULL) Arc = Arc->Next\_Arc;

        Arc->Next\_Arc = **new** Node\_Arc;

        Arc = Arc->Next\_Arc;

        Arc->adjvex = station\_b;

        Arc->Next\_Arc = NULL;

        Arc->road = road;

    }

    //无向图插两处

    Arc = Graph.Station[station\_b].First\_Arc;

**if** (Arc == NULL)

    {

        Graph.Station[station\_b].First\_Arc = **new** Node\_Arc;

        Graph.Station[station\_b].First\_Arc->adjvex = station\_a;

        Graph.Station[station\_b].First\_Arc->road = road;

        Graph.Station[station\_b].First\_Arc->Next\_Arc = NULL;

    }

**else**

    {

**while** (Arc->Next\_Arc != NULL) Arc = Arc->Next\_Arc;

        Arc->Next\_Arc = **new** Node\_Arc;

        Arc = Arc->Next\_Arc;

        Arc->adjvex = station\_a;

        Arc->Next\_Arc = NULL;

        Arc->road = road;

    }

    Graph.arcnum++;

}

//初始化图，读文件

BusGraph Init\_BusGraph()

{

    BusGraph p;

    p.arcnum = 0;

    p.vexnum = 0;

    fstream datafile;

    datafile.open("南京公交线路.txt", ios::in);

**if** (datafile.fail())

    {

        cout << "文件打开失败!" << endl;

        exit(0);

    }

**char** str[3000];

**int** road = 0;

**int** count = 0;

**int** pre\_p;

**int** n = 0;

**while** (!datafile.eof())

    {

        datafile >> road;

        datafile.getline(str, 3000, '\n');

**int** i = 5;

        //读取一路公交车的数据

**while** (i < strlen(str))

        {

**int** j;

            line\_sta\* tp = **new** line\_sta;

            //记录下站点的名字

**for** (j = 0; str[i] != ',' && i < strlen(str); i++, j++)

            {

                p.Station[p.vexnum].name[j] = str[i];

                tp->name[j] = str[i];

            }

            p.Station[p.vexnum].name[j] = '\0';

            tp->name[j] = '\0';

            tp->next = NULL;

            line\_sta\* pz = p.lu[road];

**if** (pz == NULL)

            {

                p.lu[road] = tp;

                tp->pre = NULL;

            }

**else**

            {

**while** (pz->next != NULL) pz = pz->next;

                tp->pre = pz;

                pz->next = tp;

            }

            //该站点已经在图里

**for** (j = 0; j < p.vexnum; j++)

            {

**if** (strcmp(p.Station[j].name, p.Station[p.vexnum].name) == 0)

                {

**if** (count > 0)

                    {

                        Add\_Arc(p, j, pre\_p, road);

                    }

                    pre\_p = j;

                    count++;

**break**;

                }

            }//end for (j = 0; j < p.vexnum; j++)

            //站点不在图中

**if** (j == p.vexnum)

            {

                p.Station[p.vexnum].First\_Arc = NULL;

**if** (count > 0) Add\_Arc(p, pre\_p, p.vexnum, road);

                pre\_p = p.vexnum;

                p.vexnum++;

                count++;

            }// end if(j==p.vexnum)

**if** (i < strlen(str)) i++;

**else** **break**;

        }//end while (i < strlen(str))

        pre\_p = 0;

        count = 0;

    }//end while (!datafile.eof())

    datafile.close();

    visit = **new** **int**[p.vexnum];

**return** p;

}

//销毁图

**void** Destroy\_BusGraph(BusGraph& Graph)

{

    Node\_Arc\* Arc, \* Pre\_Arc;

**for** (**int** i = 0; i < Graph.vexnum; i++)

    {

**delete** Graph.Station[i].name;

        Arc = Graph.Station[i].First\_Arc;

**if** (Arc != NULL)

        {

            Pre\_Arc = Arc;

**while** (1)

            {

                Arc = Arc->Next\_Arc;

**if** (Arc == NULL) **break**;

**delete** Pre\_Arc;

                Pre\_Arc = Arc;

            }

        }

    }

**delete** visit;

}

//寻找站的编号

**int** Find\_Station(BusGraph G, **char**\* des)

{

**for** (**int** i = 0; i < G.vexnum; i++)

    {

**if** (strcmp(G.Station[i].name, des) == 0)

**return** i;

    }

**return** -1;

}

//判断是否相邻

**bool** Is\_Adj(BusGraph G,**int** i, **int** j)

{

    Node\_Arc\* p = G.Station[i].First\_Arc;

**if** (p == NULL) **return** **false**;

**else**

    {

**while** (p != NULL)

        {

**if** (p->adjvex == j)

**return** **true**;

            p = p->Next\_Arc;

        }

    }

**return** **false**;

}

//经过站点最少的，广度优先

**void** Less\_Station(BusGraph Graph, **char**\* begin, **char**\* end)

{

**int** sta = 0, en = 0;//sta记录起点的编号，en记录终点的编号

**for** (**int** i = 0; i < Graph.vexnum; i++) visit[i] = 0;

    sta = Find\_Station(Graph, begin);

    en = Find\_Station(Graph, end);

**if** (sta == -1 || en == -1)

    {

        cout << "输入错误站点" << endl;

        exit(0);

    }

    LinkQueue q;

    InitQueue(q);

**int** j = sta;//起始点

    visit[j] = 1;

    EnQueue(q, j);//起始点入队

**int** flag = 0;

    Node\_Arc\* Arc = NULL;

**while** (!QueueEmpty(q))

    {

**if** (flag) **break**;

        DeQueue(q, j);

**for** (Arc = Graph.Station[j].First\_Arc; Arc != NULL; Arc = Arc->Next\_Arc)

        {

**if** (visit[Arc->adjvex] == 0)

            {

                visit[Arc->adjvex] = visit[j] + 1;//记录访问并算过了多少站

**if** (Arc->adjvex == en)//到达终点则退出

                {

                    flag = 1;

**break**;

                }

                EnQueue(q, Arc->adjvex);

            }

        }

    }

    Node\_Arc\* a;

    SeqStack s\_station;

    InitStack(s\_station);

    Push(s\_station, Arc->adjvex);

**int** i = Arc->adjvex;

    cout << "经过站点最少的方案：" << endl;

    cout << "一共:" << visit[Arc->adjvex] << "站" << endl;

**for** (**int** deep = visit[Arc->adjvex] - 1; deep > 1; deep--)//deep是广度的圈数

    {

**for** (j = 0; j < Graph.vexnum; j++)

        {

**if** (visit[j] == deep && Is\_Adj(Graph, i, j))//如果是上一圈且与本站点相邻的可以入栈

            {

                Push(s\_station, j);

                i = j;

**break**;

            }

        }

    }

    i = sta;

    a = Graph.Station[j].First\_Arc;

**while** (a->adjvex != i)

        a = a->Next\_Arc;

    cout << Graph.Station[sta].name << "(" << a->road << "路车)";

**while** (!StackEmpty(s\_station))

    {

        cout << "->";

        Pop(s\_station, j);

        a = Graph.Station[j].First\_Arc;

**while** (a->adjvex != i)

            a = a->Next\_Arc;

        cout << Graph.Station[j].name<<"("<<a->road<<"路车)";

        i = j;

    }

    cout << endl;

}

//插边

**void** Create\_arc(**int** i, **int** j, Line& r)

{

**int** flag;

    Node\_Arc\* p;

    p = r.l[i].firstarc;

**if** (p == NULL)                          //第一个空，新建一个

    {

        r.l[i].firstarc = **new** Node\_Arc;

        r.l[i].firstarc->adjvex = j;

        r.l[i].firstarc->Next\_Arc = NULL;

    }

**else**

    {

        flag = 0;

**while** (p->Next\_Arc != NULL)          //寻找有没有重复的

        {

**if** (p->adjvex == j) flag = 1;

            p = p->Next\_Arc;

**if** (p->adjvex == j) flag = 1;

        }

**if** (!flag)                          //没有重复的就新加一个

        {

            p->Next\_Arc = **new** Node\_Arc;

            p = p->Next\_Arc;

            p->adjvex = j;

            p->Next\_Arc = NULL;

        }

    }

    p = r.l[j].firstarc;

**if** (p == NULL)                          //第一个空，新建一个

    {

        r.l[j].firstarc = **new** Node\_Arc;

        r.l[j].firstarc->adjvex = i;

        r.l[j].firstarc->Next\_Arc = NULL;

    }

**else**

    {

        flag = 0;

**while** (p->Next\_Arc != NULL)          //寻找有没有重复的

        {

**if** (p->adjvex == i) flag = 1;

            p = p->Next\_Arc;

**if** (p->adjvex == i) flag = 1;

        }

**if** (!flag)                          //没有重复的就新加一个

        {

            p->Next\_Arc = **new** Node\_Arc;

            p = p->Next\_Arc;

            p->adjvex = i;

            p->Next\_Arc = NULL;

        }

    }

    r.arcnum++;

}

//线路初始化

**void** InitLine(BusGraph G, Line& r)

{

    r.vexnum = 0;

    Node\_Arc\* arc;

**int** i, j, k;

**int** count;

**int** Car\_bs\_all\_bus[500];

**for** (**int** kk = 0; kk < 1000; kk++) r.l[kk].firstarc = NULL;

**for** (i = 0; i < G.vexnum; i++)                               //把经过同一站点的路线相连

    {

        arc = G.Station[i].First\_Arc;

**for** (j = 0; j < 500; j++) Car\_bs\_all\_bus[j] = 0;

**if** (arc == NULL) **continue**;

        count = 0;

**while** (arc != NULL)

        {

**for** (k = 0; k < count; k++)                          //这一条路线是否已经存过

**if** (arc->road == Car\_bs\_all\_bus[k]) **break**;

**if** (k == count)                                     //没存过的路线，新加入

            {

                Car\_bs\_all\_bus[count++] = arc->road;

**if** (arc->road > r.vexnum)r.vexnum = arc->road;

            }

            arc = arc->Next\_Arc;

        }

**for** (k = 0; k < count - 1; k++)

**for** (j = k + 1; j < count; j++)

**if** (Car\_bs\_all\_bus[k] != Car\_bs\_all\_bus[j])     //不是同一路线，就相连

                    Create\_arc(Car\_bs\_all\_bus[j], Car\_bs\_all\_bus[k], r);

    }

    r.visit\_lint = **new** **int**[r.vexnum + 1];

}

//销毁线路

**void** Destory\_Line(Line& r)

{

    Node\_Arc\* p, \* prep;

**for** (**int** i = 1; i <= r.vexnum; i++)

    {

        p = r.l[i].firstarc;

**if** (p != NULL)

        {

            prep = p;

**while** (1)

            {

                p = p->Next\_Arc;

**if** (p == NULL)**break**;

**delete** prep;

                prep = p;

            }

        }

    }

}

//判断相邻

**bool** Is\_Adj(Line r, **int** i, **int** j)

{

    Node\_Arc\* p = r.l[i].firstarc;

**if** (p == NULL) **return** **false**;

**else**

    {

**while** (p != NULL)

        {

**if** (p->adjvex == j)

**return** **true**;

            p = p->Next\_Arc;

        }

    }

**return** **false**;

}

//获取更小换乘数,广度优先算法

**int** Less\_Change(Line r, **int** i, **int** j)

{

**if** (i == j)                                                 //同一站

    {

**return** 0;

    }

**for** (**int** k = 1; k < r.vexnum + 1; k++)

        r.visit\_lint[k] = 0;

**int** v, u;

    Node\_Arc\* p= NULL;

**int** flag = 0;

    LinkQueue q;

    InitQueue(q);

    v = i;

    r.visit\_lint[v] = 1;

    EnQueue(q, v);

**while** (!QueueEmpty(q))                                      //广度优先遍历

    {

**if** (flag) **break**;

        DeQueue(q, u);

**for** (p = r.l[u].firstarc; p != NULL; p = p->Next\_Arc)

**if** (r.visit\_lint[p->adjvex] == 0)

            {

                r.visit\_lint[p->adjvex] = r.visit\_lint[u] + 1;

**if** (p->adjvex == j)

                {

                    flag = 1;

**break**;

                }

                EnQueue(q, p->adjvex);

            }

    }

**return** r.visit\_lint[p->adjvex];

}

//输出换乘最小方案

**void** Print\_Min\_Change(BusGraph G,Line r, **int** i, **int** j, **char**\* start, **char**\* end)

{

    line\_sta\* sta = G.lu[i];

**if** (i == j)

    {

        cout << "两站位于" << i << "路车上" << endl;

**while** (strcmp(sta->name, start) != 0)

            sta = sta->next;

**while** (strcmp(sta->name, end) != 0)

        {

            cout << sta->name << "("<<i<<"路)->";

            sta = sta->next;

        }

        cout << end;

**return**;

    }

**for** (**int** k = 1; k < r.vexnum + 1; k++) r.visit\_lint[k] = 0;

    //广度优先，遍历从路线i到路线j的最短换乘

**int** v, u;

    Node\_Arc\* p = NULL;

**int** flag = 0;

    LinkQueue q;

    InitQueue(q);

    v = i;

    r.visit\_lint[v] = 1;

    EnQueue(q, v);

**while** (!QueueEmpty(q))

    {

**if** (flag) **break**;

        DeQueue(q, u);

**for**(p=r.l[u].firstarc;p!=NULL;p=p->Next\_Arc)

**if** (r.visit\_lint[p->adjvex] == 0)

            {

                r.visit\_lint[p->adjvex] = r.visit\_lint[u] + 1;

**if** (p->adjvex == j)

                {

                    flag = 1;

**break**;

                }

                EnQueue(q, p->adjvex);

            }

    }

    Node\_Arc\* t;

    SeqStack s;

    InitStack(s);

    Push(s,p->adjvex);

**int** m = p->adjvex;

    cout << "转乘最少方案:\n";

**if** (r.visit\_lint[p->adjvex] <= 1)

        cout << "不用转乘\n";

**else**

        cout << "一共转乘了:" << r.visit\_lint[p->adjvex] - 1 << "次" << endl;

**for**(**int** deep = r.visit\_lint[p->adjvex]-1;deep>1;deep--)

**for**(j=0;j<r.vexnum;j++)

**if** (r.visit\_lint[j] == deep && Is\_Adj(r, m, j))

            {

                Push(s, j);

                m = j;

**break**;

            }

    cout << start<<"("<<i<<"路车)->";

    m = i;                                                  //m为本站的路线

**char**\* st = **new** **char**[50];

    strcpy(st, start);

**int** temp;

**while** (!StackEmpty(s))

    {

**int** flag = 0;

                                                    //temp是后一站的路线

        Pop(s, temp);

**int** pos = Find\_Station(G, st);

        Node\_Arc\* tp = G.Station[pos].First\_Arc;

**int** sum = -1;

**int** p[100];

**int** sum\_p = 0;

        p[0] = pos; sum\_p++;

        line\_sta\* te = G.lu[m];

**int** adj;

**while** (te != NULL)

        {

            //cout << te->name << endl;

**int** op = Find\_Station(G, te->name);

            Node\_Arc\* opt = G.Station[op].First\_Arc;

**while** (opt != NULL && opt->road!=temp)

            {

                opt = opt->Next\_Arc;

            }

**if** (opt == NULL)te = te->next;

**else** { adj = opt->adjvex; **break**; }

        }

        cout << G.Station[adj].name << "(" << temp << "路车)->";

        m = temp;

    }

    cout <<end<<"("<<temp<<"路车)" << endl;

}

//对每一个点广度优先

**void** Min\_Change(Line r, BusGraph G, **char**\* beg, **char**\* en)

{

**int** begin = Find\_Station(G, beg), end = Find\_Station(G, en);

**for** (**int** i = 1; i < r.vexnum + 1; i++)   r.visit\_lint[i] = 0;

    Node\_Arc\* p;

**int** s\_road[100],e\_road[100],s\_count=0,e\_count=0;//s\_road[] 是起点可以乘的公交路全部

**char** s[50], e[50];

    p = G.Station[begin].First\_Arc;

**while** (p != NULL)

    {

**if**(s\_count != 0 && s\_road[s\_count-1]!=p->road )//不是同一线路，也不能跳两个(起点)

            s\_road[s\_count++] = p->road;

**if** (s\_count == 0)

            s\_road[s\_count++] = p->road;

        p = p->Next\_Arc;

    }

    p = G.Station[end].First\_Arc;

**while** (p != NULL)

    {

**if** (e\_count != 0 && e\_road[e\_count - 1] != p->road)

            e\_road[e\_count++] = p->road;

**if** (e\_count == 0)

            e\_road[e\_count++] = p->road;

        p = p->Next\_Arc;

    }

**int** min = 999999;

**int** tmp = 0;

**int** min\_s = 0;

**int** min\_e = 0;

**int** sign = 0;                                           //记录有没有经过同一站

**for** (**int** i = 0; i < s\_count; i++)

    {

        sign = 0;

**for** (**int** j = 0; j < e\_count; j++)

        {

**if** (s\_road[i] == e\_road[j])                     //两站点位于同一条线路

            {

                sign = 1;

                min = 1;

                min\_s = min\_e = s\_road[i];

**break**;

            }

            tmp = Less\_Change(r, s\_road[i], e\_road[j]);

**if** (tmp < min)

            {

                min = tmp;

                min\_s = s\_road[i];

                min\_e = e\_road[j];

            }

        }

**if** (sign)**break**;

    }

    Print\_Min\_Change(G, r, min\_s, min\_e, beg, en);

}

公交路线.cpp文件：

#include<iostream>

#include<string.h>

#include<fstream>

#include<stack>

#include<queue>

#include<math.h>

#include"BusGraph.h"

**using** **namespace** std;

**const** **int** inf = 0x3f3f3f3f;

**int** main()

{

    BusGraph Graph=Init\_BusGraph();

    cout << "输入起点:";

**char** Start[50];

**char** End[50];

    cin >> Start;

    cout << "输入终点:";

    cin >> End;

    Less\_Station(Graph, Start, End);

    Line r;

    InitLine(Graph, r);

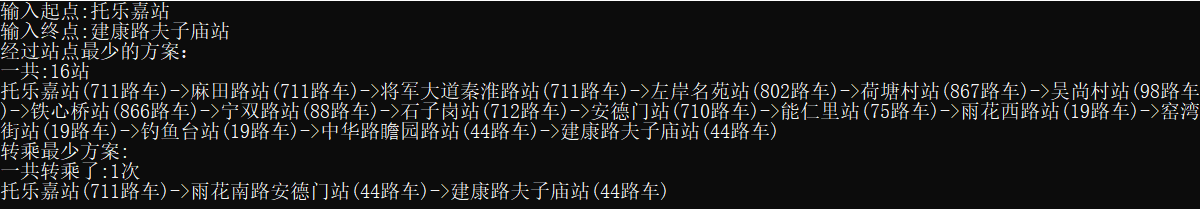
    Min\_Change(r, Graph, Start, End);

    Destory\_Line(r);

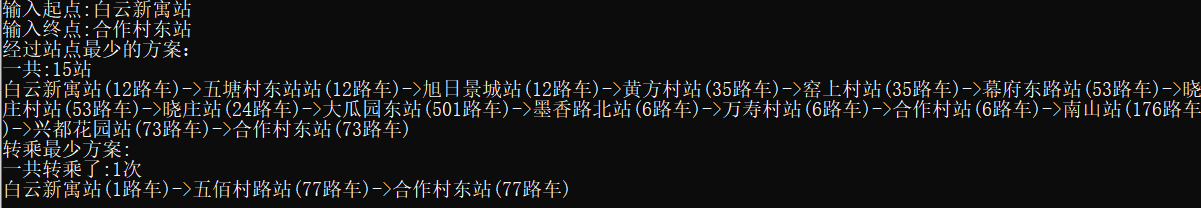
    Destroy\_BusGraph(Graph);

**return** 0;

}

4.测试数据和结果

转乘方案解释：从托乐嘉站乘坐711路车，到雨花南路安德门站转乘44路车最后到达建康路夫子庙站。



5.算法的时间复杂度

该算法的时间复杂度主要由广度优先算法消耗，其时间复杂度为O(n+e)，其中n为顶点数，e为边数。

6.算法的改进方法

该算法是用了比较复杂但是通俗易懂的方法。可以从搜索方面再度改进。我认为可以起点和终点双向查找路线，这样肯定会比从起点开始找快一点。

另外也可以进行一些站点的合并。比如将一些不可以与其他路线换乘的连续站点合并为一个站点，然后就可以减少顶点的数量，以此减少算法的复杂度。

除此之外也可以将站点分区块，分区块进行查询乘车路线。

1. Hash表应用（必做）（查找）

1.数据结构

用链地址存储哈希表时就是用一组数组，后面跟着链表，一个元素存储多个记录。

用开放地址存储哈希表时用的就是一组数组，一个元素存储一个记录。

//客户

**typedef** **struct** Client {

    string id;   //身份证

    string name; //姓名

    string flight;  //航班号

    string date; //航班日期

**double** mile; //里程

}Client;

//其他信息

**typedef** **struct** information {

    string flight;  //航班号

    string date; //航班日期

    information\* next;

}information;

//存放里程数、乘机频率的客户信息

**class** ClientInfo {

**public**:

    string id;

    string name;

    information inf;

**double** Sum\_Mile; //里程总数

**int** Freq;        //乘机频率

**static** **int** Num\_Used;

**int**  reLocNum;   //解决冲突重新定位的次数

    ClientInfo() {

        Freq = 0;

        Sum\_Mile = 0;

        inf.flight = '\0';

        reLocNum = 0;

    }

};

//链地址解决冲突的结点

**typedef** **struct** ConflictLink {

    string id;

    string name;

    information inf;

**int** Freq;

**double** Sum\_Mile;

**struct** ConflictLink\* next;

    ConflictLink() {

        next = NULL;

        Freq = 0;

        Sum\_Mile = 0;

        id = "\0";

        inf.date = '\0';

        inf.flight = '\0';

        inf.next = NULL;

    }

}ConflictLink;

2.算法设计思想

链地址解决冲突就是将产生冲突的所有关键字保留到一个链表中。此时，哈希表的每个存储单元存放的是产生冲突所有关键字构成的链表的表头。

这里采用的开放定址法用的是线性探测法。该方法就是以发生冲突的哈希地址开始，依次探测其下一个地址，直到找到一个空闲的哈希地址为止。

3.源代码

Client.h文件：

#include<iostream>

#include<string>

#include<Windows.h>

#include<algorithm>

#include<fstream>

#include<vector>

**using** **namespace** std;

#define MAXH 35

#define MAXN 150

//客户

**typedef** **struct** Client {

    string id;   //身份证

    string name; //姓名

    string flight;  //航班号

    string date; //航班日期

**double** mile; //里程

}Client;

//其他信息

**typedef** **struct** information {

    string flight;  //航班号

    string date; //航班日期

    information\* next;

}information;

//存放里程数、乘机频率的客户信息

**class** ClientInfo {

**public**:

    string id;

    string name;

    information inf;

**double** Sum\_Mile; //里程总数

**int** Freq;        //乘机频率

**static** **int** Num\_Used;

**int**  reLocNum;   //解决冲突重新定位的次数

    ClientInfo() {

        Freq = 0;

        Sum\_Mile = 0;

        inf.flight = '\0';

        reLocNum = 0;

    }

};

//链地址解决冲突的结点

**typedef** **struct** ConflictLink {

    string id;

    string name;

    information inf;

**int** Freq;

**double** Sum\_Mile;

**struct** ConflictLink\* next;

    ConflictLink() {

        next = NULL;

        Freq = 0;

        Sum\_Mile = 0;

        id = "\0";

        inf.date = '\0';

        inf.flight = '\0';

        inf.next = NULL;

    }

}ConflictLink;

**void** Read\_File(vector<Client>& ClientV);

**int** Get\_ID\_Num(string s);

**void** Hash\_Link(ConflictLink\* c, vector<Client> ClientV);

**void** Hash\_Linear(ClientInfo c\_INFO[], vector<Client> ClientV);

**void** Find(ConflictLink c[]);

#pragma once

Client.cpp文件：

#include<iostream>

#include<algorithm>

#include<fstream>

#include<vector>

#include<string>

#include"cilent.h"

**using** **namespace** std;

**int** ClientInfo::Num\_Used = 0;

// 中文转换编码

string UTF8ToGB(**const** **char**\* str)

{

    string rst;

**WCHAR**\* strSrc;

**LPSTR** szRes;

    // 获取临时变量大小

**int** i = MultiByteToWideChar(CP\_UTF8, 0, str, -1, NULL, 0);

    strSrc = **new** **WCHAR**[i + 1];

    MultiByteToWideChar(CP\_UTF8, 0, str, -1, strSrc, i);

    i = WideCharToMultiByte(CP\_ACP, 0, strSrc, -1, NULL, 0, NULL, NULL);

    szRes = **new** **CHAR**[i + 1];

    WideCharToMultiByte(CP\_ACP, 0, strSrc, -1, szRes, i, NULL, NULL);

    rst = szRes;

**delete**[] strSrc;

**delete**[] szRes;

**return** rst;

}

//读取文件内容

**void** Read\_File(vector<Client>& ClientV)

{

    fstream datafile;

    Client cli;

    datafile.open("Hash.txt", ios::in);

**if** (datafile.fail())

    {

        cout << "open fail!\n";

        exit(0);

    }

**while** (!datafile.eof())

    {

        datafile >> cli.id;

        datafile >> cli.name;

        datafile >> cli.flight;

        datafile >> cli.date;

        datafile >> cli.mile;

        cli.name = UTF8ToGB(cli.name.c\_str());

        //cout << cli.id << " " << cli.name << " " << cli.flight << " " << cli.date << " " << cli.mile << endl;

        ClientV.push\_back(cli);

    }

    datafile.close();

}

//计算身份证的数

**int** Get\_ID\_Num(string s)

{

**int**  sum = 0;

**for** (**int** i = 0; i < s.length() ; i++)

    {

        sum = sum + s[i] -48;

    }

**return** sum;

}

//链地址解决冲突

**void** Hash\_Link(ConflictLink\* c,vector<Client> ClientV)

{

**int** s = 0;

    ClientInfo Client\_Info[MAXH];

**int** hash;

**int** Hash\_Table[MAXH] = { 0 };

    ConflictLink\* cl;

**for** (**int** i = 0; i < ClientV.size(); i++)

    {

        hash = Get\_ID\_Num(ClientV[i].id) % MAXH;

**if** (Hash\_Table[hash] == 0)//这是第一次，之前没有数据

        {

            ConflictLink\* p = **new** ConflictLink;

            p->Freq++;

            p->id = ClientV[i].id;

            p->name = ClientV[i].name;

            p->Sum\_Mile += ClientV[i].mile;

            p->next = NULL;

            //添加一次航班及时间信息

            information\* n = **new** information;

            p->inf.next = n;//第一个位置放弃使用，方便之后的while

            n->date = ClientV[i].date;

            n->flight = ClientV[i].flight;

            n->next = NULL;

            //同样弃用第一个，方便后面的遍历

            c[hash].next = p;

            Hash\_Table[hash]++;

        }

**else** //已出现，判断是否有重复的，有的话就合并，没有就新建

        {

            ConflictLink\* p = c[hash].next;

**if** (p->id == ClientV[i].id)//第一个就是同一个人

            {

                //将频次加1，里程累计

                p->Freq++; p->Sum\_Mile += ClientV[i].mile;

                //新建一个信息记录

                information\* mation = p->inf.next;

                //cout << mation->flight << endl;

                //找到最后一个信息地址

**while** (mation->next != NULL) mation = mation->next;

                //新建信息

                information\* new\_record = **new** information;

                new\_record->date = ClientV[i].date;

                new\_record->flight = ClientV[i].flight;

                new\_record->next = NULL;

                mation->next = new\_record;

            }

**else**//第一个不是

            {

                ConflictLink\* fa = p; //fa记录p的前一个指针

                p = p->next;

                //寻找一样ID的记录

**while** (p != NULL && p->id != ClientV[i].id) { fa = p; p = p->next; }

**if** (p == NULL)//没找到

                {

                    //在c[hash]后面新加一个结点，添加新人

                    ConflictLink\* new\_con = **new** ConflictLink;

                    new\_con->Freq++;

                    new\_con->id = ClientV[i].id;

                    new\_con->name = ClientV[i].name;

                    new\_con->Sum\_Mile += ClientV[i].mile;

                    new\_con->next = NULL;

                    //新建该结点的一个信息

                    information\* new\_record = **new** information;

                    new\_record->date = ClientV[i].date;

                    new\_record->flight = ClientV[i].flight;

                    new\_record->next = NULL;

                    new\_con->inf.next = new\_record;

                    fa->next = new\_con;

                    Hash\_Table[hash]++;

                }

**else**//找到了

                {

                    p->Freq++; p->Sum\_Mile += ClientV[i].mile;

                    //新建一个信息记录

                    information\* mation = p->inf.next;

**while** (mation->next != NULL) mation = mation->next;

                    information\* new\_record = **new** information;

                    new\_record->date = ClientV[i].date;

                    new\_record->flight = ClientV[i].flight;

                    new\_record->next = NULL;

                    mation->next = new\_record;

                }

            }

        }

    }

**int** sum = 0;

**for** (**int** i = 0; i < MAXH; i++)

    {

        sum += Hash\_Table[i];

**if** (Hash\_Table[i] != 0)

        {

            s++;

            cout << "第" << i << "个位置有" << Hash\_Table[i] << "个:"<<endl;

            ConflictLink\* p = c[i].next;

**while**(p!=NULL)

            {

                cout <<"\t姓名:"<< p->name << " ID:" << p->id << " 频次:"<<p->Freq<<" 里程数:" << p->Sum\_Mile << endl;

                p = p->next;

            }

        }

    }

    cout << "共有"<<sum << "人  哈希表的装填因子为" << (**double**)s/MAXH;

}

//线性探测法

**void** Hash\_Linear(ClientInfo c\_INFO[], vector<Client> ClientV)

{

**int** Hash\_Table[MAXN] = { 0 };

**int** pos;

**for** (**int** i = 0; i < ClientV.size(); i++)

    {

        pos = Get\_ID\_Num(ClientV[i].id);

**if** (Hash\_Table[pos] == 0)//该位置没有数据放过

        {

            c\_INFO[pos].Freq++;

            c\_INFO[pos].id = ClientV[i].id;

            c\_INFO[pos].name = ClientV[i].name;

            c\_INFO[pos].reLocNum = 0;

            c\_INFO[pos].Sum\_Mile += ClientV[i].mile;

            information\* new\_info = **new** information;

            new\_info->date = ClientV[i].date;

            new\_info->flight = ClientV[i].flight;

            new\_info->next = NULL;

            c\_INFO[pos].inf.next = new\_info;

            Hash\_Table[pos]++;

            c\_INFO[pos].Num\_Used++;

        }

**else** **if** (c\_INFO[pos].id == ClientV[i].id)//放过了，但是是同一个

        {

            c\_INFO[pos].Freq++;

            c\_INFO[pos].Sum\_Mile += ClientV[i].mile;

            information\* mation = c\_INFO[pos].inf.next;

            //cout << mation->flight << endl;

            //找到最后一个信息地址

**while** (mation->next != NULL) mation = mation->next;

            //新建信息

            information\* new\_record = **new** information;

            new\_record->date = ClientV[i].date;

            new\_record->flight = ClientV[i].flight;

            new\_record->next = NULL;

            mation->next = new\_record;

            Hash\_Table[pos]++;

        }

**else** //不是同一个，查下一个地址

        {

**int** Empty\_Pos = i;

**int** j = 1;

**bool** flag = **false**;

**for** (j = 1; (j + i) % MAXN != i; j++)//查找之后的位置里有没有这个人

            {

**int** p = (j + i) % MAXN;

**if** (Empty\_Pos == i && Hash\_Table[p] == 0) Empty\_Pos = p;

**if** (c\_INFO[p].id == ClientV[i].id)

                {

                    flag = **true**;

**break**;

                }

            }

**if**(flag)//有一样的人在后面

            {

                pos = (j + i) % MAXN;

                Hash\_Table[pos]++;

                c\_INFO[pos].Freq++;

                c\_INFO[pos].Sum\_Mile += ClientV[i].mile;

            }

**else** //没有一样的人

            {

                c\_INFO[Empty\_Pos].Freq++;

                c\_INFO[Empty\_Pos].id = ClientV[i].id;

                c\_INFO[Empty\_Pos].name = ClientV[i].name;

                c\_INFO[Empty\_Pos].reLocNum = 0;

                c\_INFO[Empty\_Pos].Sum\_Mile += ClientV[i].mile;

                information\* new\_info = **new** information;

                new\_info->date = ClientV[i].date;

                new\_info->flight = ClientV[i].flight;

                new\_info->next = NULL;

                c\_INFO[Empty\_Pos].inf.next = new\_info;

                Hash\_Table[Empty\_Pos]++;

                c\_INFO[Empty\_Pos].Num\_Used++;

**if** (Empty\_Pos > i) c\_INFO[Empty\_Pos].reLocNum = Empty\_Pos - i;

**else** c\_INFO[Empty\_Pos].reLocNum = Empty\_Pos + MAXN - i;

            }

        }

    }

**int** sum = 0, s = 0, c = 0;;

**for** (**int** i = 0; i < MAXN; i++)

    {

        sum += Hash\_Table[i];

**if** (Hash\_Table[i] != 0)

        {

**if** (c\_INFO[i].reLocNum > 0) c++;

            s++;

            cout << "姓名为 " << c\_INFO[i].name << " id为 " << c\_INFO[i].id << " 乘机频率为 " << c\_INFO[i].Freq << " 里程数为 " << c\_INFO[i].Sum\_Mile << " 重新定位次数为 " << c\_INFO[i].reLocNum << endl;

        }

    }

    cout << "共有" << sum << "组数据 ， 有" << s << "人，  冲突次数为" << c << "次";

}

//查VIP

**void** Find(ConflictLink c[])

{

    cout << "输入VIP用户的频次(>):";

**int** f;

    cin >> f;

    cout << "输入VIP用户的里程数(>):";

**double** m;

    cin >> m;

**for** (**int** i = 0; i < MAXH; i++)

    {

        ConflictLink\* p = c[i].next;

**while** (p != NULL)

        {

**if** (p->Freq != 0 && p->Freq > f && p->Sum\_Mile > m)

            {

                cout << "姓名:" << p->name << "  id:" << p->id << "  频次:" << p->Freq << "  里程数:" << p->Sum\_Mile << endl;

            }

            p = p->next;

        }

    }

}

Hash.cpp文件：

#include <iostream>

#include <string>

#include <windows.h>

#include <algorithm>

#include <fstream>

#include <vector>

#include"cilent.h"

**using** **namespace** std;

**int** main()

{

    vector<Client> ClientV;

    Read\_File(ClientV);

    ConflictLink l[MAXH];

    cout << "1.链地址\n2.开放定址\n3.VIP用户(如需查找，请先选择1)\n4.退出\n请输入选项:";

**int** cho;

    cin >> cho;

    ClientInfo Info[MAXN];

**while** (1)

    {

**switch** (cho)

        {

**case** 1:

            Hash\_Link(l, ClientV);

**break**;

**case** 2:

            Hash\_Linear(Info, ClientV);

**break**;

**case** 3:

            Find(l); **break**;

**case** 4:

**delete** []Info;

            free(l);

**return** 0;

        }

        cout << endl;

        system("pause");

        system("cls");

        cout << "1.链地址\n2.开放定址\n3.VIP用户(如需查找，请先选择1)\n4.退出\n请输入选项:";

        cin >> cho;

    }

**return** 0;

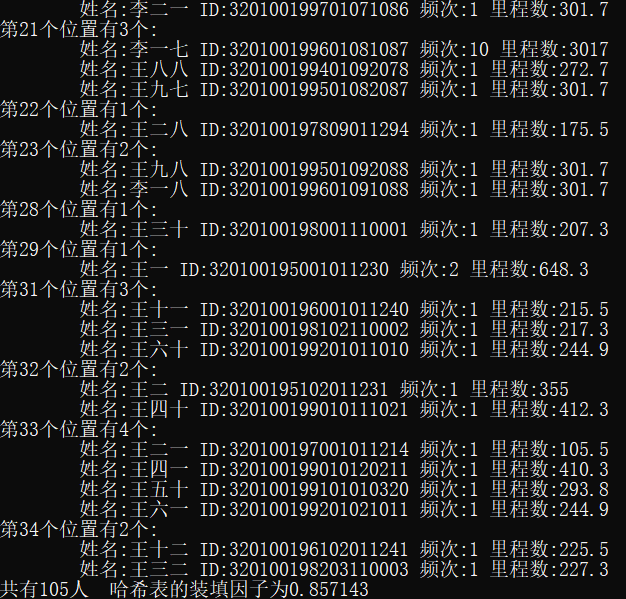
}

4.测试数据和结果

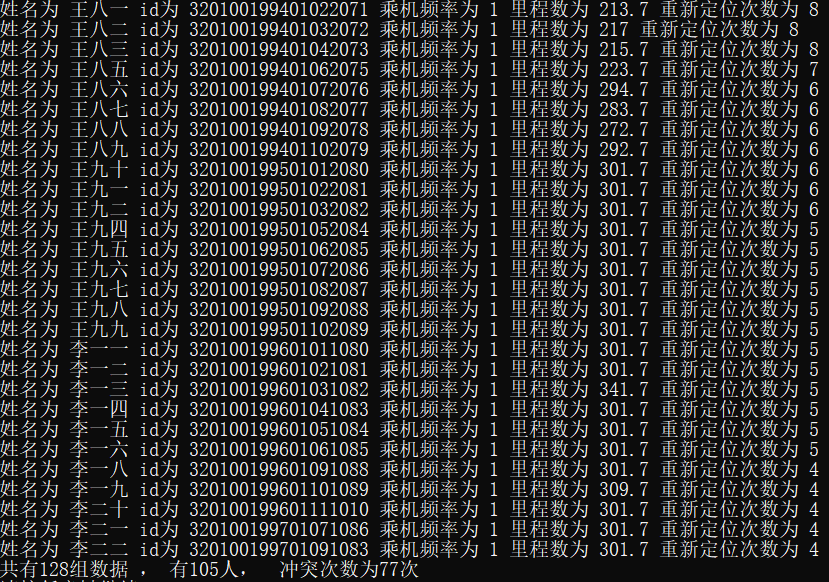
测试数据为Hash.txt文件内容

由于结果过长，只截取一小部分。

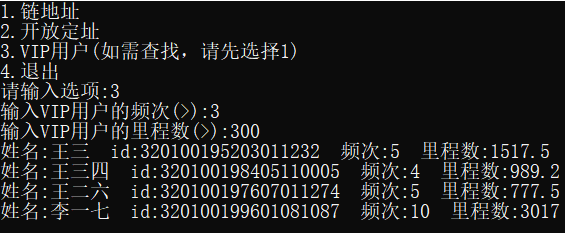
链地址存储截图：



开放定址截图：



查找VIP用户截图：



5.算法的时间复杂度

该算法时间复杂度主要由创建时的时间复杂度决定，时间复杂度为O(N)。

1. 排序算法比较（必做）（排序）

1.数据结构

本题数据结构是十组数组，用不同的排序方法计算所用的时间。

2.算法设计思想

用Time.h头文件里的clock函数计算排序所花费的时间，之后正常排序就可以了。

3.源代码

排序算法比较.cpp：

#include<iostream>

#include<time.h>

#include<algorithm>

#include<Windows.h>

#include<fstream>

#include"Queue.h"

#include<queue>

**using** **namespace** std;

**const** **int** AMOUNT = 50000;

string Rand\_File[10] = {

    "rand\_ascend\_1.txt","rand2.txt","rand3.txt","rand4.txt","rand5.txt",

    "rand6.txt","rand7.txt","rand8.txt","rand9.txt","rand\_descend\_10.txt"

};

//获取随机数

**void** GetRand()

{

**int** temp[AMOUNT + 1] = { 0 };

    srand((**int**)time(0));

    system("del rand\_ascend\_1.txt");

    system("del rand2.txt");

    system("del rand3.txt");

    system("del rand4.txt");

    system("del rand5.txt");

    system("del rand6.txt");

    system("del rand7.txt");

    system("del rand8.txt");

    system("del rand9.txt");

    system("del rand\_descend\_10.txt");

    fstream datafile;

    cout << "正在生成第1组数据....."<<endl;

    //datafile.open("rand\_ascend\_1.txt", ios::in | ios::out | ios::app);

    datafile.open(Rand\_File[0].c\_str(), ios::app | ios::in);

**for** (**int** i = 0; i < AMOUNT; i++)

    {

        datafile << i+1 << " ";

        //cout << t << " ";

    }

    cout << "第1组数据生成完成" << endl;

    datafile.close();;

**for** (**int** i = 1; i < 9; i++)

    {

        cout << "正在生成第" << i << "组数据....." << endl;

        datafile.open(Rand\_File[i].c\_str(), ios::app | ios::in);

**for** (**int** j = 0; j < AMOUNT; j++)

        {

            datafile << rand() << " ";

        }

        cout << "第" << i << "组数据生成完成" << endl;

        datafile.close();

    }

    cout << "正在生成第10组数据....." << endl;

    datafile.open(Rand\_File[9].c\_str(), ios::app | ios::in);

**for** (**int** i = AMOUNT; i > 0; i--)

    {

        datafile << i  << " ";

        //cout << t << " ";

    }

    cout << "第10组数据生成完成" << endl;

    datafile.close();

    system("pause");

    system("cls");

}

//判断是否已经有序

**bool** Is\_Order(**int** a[])

{

**for** (**int** i = 1; i < AMOUNT; i++)

**if** (a[i] > a[i + 1]) **return** **false**;

**return** **true**;

}

//读取数据到ORIG数组中

**void** Read\_Data(**int** a[],string filename)

{

**int** max = 0;

    fstream file;

    file.open(filename.c\_str(), ios::in);

**if** (file.fail())

    {

        cout << "文件打开失败!\n";

        exit(0);

    }

**for** (**int** i = 1; i <= AMOUNT; i++)

    {

        file >> a[i];

        //if (a[i] > max) max = a[i];

    }

    //cout << "最大值为" << max << endl;

    file.close();

}

//直接插入排序

**void** Insert\_Sort(**int** a[])

{

**for** (**int** i = 2; i <= AMOUNT; i++)

    {

**int** j;

        a[0] = a[i];

**for** (j = i; j > 1; j--)

**if** (a[0] < a[j - 1]) a[j] = a[j - 1];

**else** **break**;

        a[j] = a[0];

    }

    //for (int i = 1; i <= AMOUNT; i++)

    //  cout << ORIG[i] << " ";

}

//希尔排序

**void** Shell\_Insert(**int** a[], **int** d)

{

**for** (**int** i = d + 1; i <= AMOUNT; i++)

    {

**int** j = i;

**if** (a[i] < a[i - d])

        {

            a[0] = a[i];

**for** (j = i; j > d; j = j - d)

            {

**if** (a[0] < a[j - d]) a[j] = a[j - d];

**else** **break**;

            }

            a[j] = a[0];

        }

    }

}

**void** Shell(**int** a[], **int** d[], **int** t)

{

**for** (**int** k = 0; k < t; k++)

    {

        Shell\_Insert(a, d[k]);

    }

}

**void** Shell\_Sort(**int** a[])

{

**int** d[20];

**int** cnt = AMOUNT / 2;

**int** i = 0;

**while** (cnt > 0)

    {

        d[i++] = cnt;

        cnt /= 2;

    }

    Shell(a, d, i);

}

//冒泡排序

**void** Bubble\_Sort(**int** a[])

{

**for** (**int** i = 1; i < AMOUNT; i++)

    {

**for**(**int** j=1;j<=AMOUNT-i;j++)

**if** (a[j] > a[j + 1])

            {

**int** temp = a[j];

                a[j] = a[j + 1];

                a[j + 1] = temp;

            }

    }

}

//快速排序

**int** Partition(**int** a[], **int** low, **int** high)

{

    a[0] = a[low];//作为枢轴

**while** (low < high)

    {

**while** (low < high && a[high] >= a[0]) high--;

        a[low] = a[high];//将比枢轴小的数移到前面

**while** (low < high && a[low] <= a[0]) low++;

        a[high] = a[low];//将比枢轴大的数移到后面

    }

    a[low] = a[0];//将枢轴放到该放的位置

**return** low;//返回枢轴

}

**void** Qsort(**int** a[], **int** low, **int** high)//递归再次快排

{

**if** (low < high)

    {

**int** p = Partition(a, low, high);//一趟快速排序

        Qsort(a, low, p - 1);//对前部进行快排

        Qsort(a, p + 1, high);//对后部进行快排

    }

}

**void** QuickSort(**int** a[])

{

    Qsort(a, 1, AMOUNT);

}

//选择排序

**void** SelectionSort(**int** a[])

{

**for** (**int** i = 1; i <= AMOUNT; i++)

    {

**int** min = a[i], pos = i;

**for** (**int** j = i; j <= AMOUNT; j++)

**if** (a[j] < min)

            {

                min = a[j]; pos = j;

            }

        swap(a[i], a[pos]);

    }

}

//堆排序

**void** HeapAdjust(**int** a[], **int** low, **int** high)

{

**int** temp = a[low];

**int** i = low, j = 2 \* i;//i记录应插入的位置，j是其左子节点的位置

**while** (j <= high)

    {

**if** (j + 1 <= high && a[j] < a[j + 1]) j++;//记录子节点较大的位置

**if** (temp >= a[j]) **break**;

        a[i] = a[j]; i = j; j = 2 \* i;//循环其子树

    }

    a[i] = temp;

}

**void** HeapSort(**int** a[])

{

**for** (**int** i = AMOUNT / 2; i > 0; i--)//第一个非叶子结点为L.length/2

        HeapAdjust(a, i, AMOUNT);

**for** (**int** i = AMOUNT; i > 1; i--)

    {

**int** temp = a[1];

        a[1] = a[i]; a[i] = temp;//将根结点移到最后一个结点，再调整堆

        HeapAdjust(a, 1, i - 1);

    }

}

//归并排序

**void** Merge(**int** a[], **int** low, **int** mid, **int** high)//合并两个集合

{

**int** b[AMOUNT + 1] = { 0 };

    //T.r = (int\*)malloc((high - low + 1) \* sizeof(int));

**int** i = low, j = mid + 1, k = 0;

**while** (i <= mid && j <= high)

    {

**if** (a[i] <= a[j]) { b[k] = a[i]; i++; k++; }

**else** { b[k] = a[j]; j++; k++; }

    }

**while** (i <= mid) { b[k] = a[i]; i++; k++; }

**while** (j <= high) { b[k] = a[j]; j++; k++; }

**for** (**int** k = 0, i = low; i <= high; k++, i++)

        a[i] = b[k];

}

**void** Msort(**int** a[], **int** len)

{

**int** i = 1;

**while** (i + 2 \* len <= AMOUNT)//有两个集合能够合并

    {

        Merge(a, i, i + len - 1, i + 2 \* len - 1);

        i = i + 2 \* len;

    }

**if** (i + len <= AMOUNT)//有一个len长度加其他长度的集合

        Merge(a, i, i + len - 1, AMOUNT);

}

**void** MergeSort(**int** a[])

{

**for** (**int** len = 1; len <= AMOUNT; len \*= 2)//分成的集合的元素个数

    {

        Msort(a, len);

    }

}

//基数排序

**int** GetMax(**int** a[])

{

**int** max = a[1];

**for** (**int** i = 2; i < AMOUNT + 1; i++)

**if** (a[i] > max)max = a[i];

**int** n=0;

**while** (max > 0)

    {

        n++; max /= 10;

    }

**return** n;

}

**void** radixsort(**int** a[])

{

    queue<**int**> Q[10];

**int** r = GetMax(a);

**int** radix = 1;

**int** i, j, k;

**int** m;

**for** (k = 1; k <= r; k++)

    {

        radix \*= 10;

**for** (i = 1; i < AMOUNT + 1; i++)

        {

            m = (a[i] % radix) / (radix / 10);

            Q[m].push(a[i]);

        }

**for** (i = 1, m = 0; m < 10; m++)

        {

**while** (!Q[m].empty())

            {

                a[i] = Q[m].front();

                Q[m].pop();

                i++;

            }

        }

    }

}

//获取各个排序花费的时间

**void** Get\_Time()

{

**int**\* a = **new** **int**[AMOUNT + 1];

**clock\_t** start, end;

**double** duration;

    cout << "1.直接排序算法:" << endl;

    duration = 0;

**for** (**int** i = 0; i < 10; i++) {

        cout << "\t正在处理第" << i + 1 << "组数据....." << endl;

        Read\_Data(a,Rand\_File[i]);

        start = clock();

        Insert\_Sort(a);

        end = clock();

        duration = (**double**)end - start;

**if** (Is\_Order(a)) cout << "\t\t排序完成\n";

**else** cout << "\t\t排序失败\n";

        printf("\t第%d组数据处理时间为%.3lf毫秒\n\n", i + 1, duration);

        //cout << "\t第"<<i+1<<"组数据插入时间为:" << duration << " 毫秒" << endl<<endl;

    }

    cout << "2.希尔排序算法:" << endl;

    duration = 0;

**for** (**int** i = 0; i < 10; i++) {

        cout << "\t正在处理第" << i + 1 << "组数据....." << endl;

        Read\_Data(a, Rand\_File[i]);

        start = clock();

        Shell\_Sort(a);

        end = clock();

        duration = (**double**)end - start;

**if** (Is\_Order(a)) cout << "\t\t排序完成\n";

**else** cout << "\t\t排序失败\n";

        printf("\t第%d组数据处理时间为%.3lf毫秒\n\n", i + 1, duration);

        //cout << "\t第" << i + 1 << "组数据处理时间为:" << duration << " 毫秒" << endl << endl;

    }

    cout << "3.冒泡排序算法:" << endl;

    duration = 0;

**for** (**int** i = 0; i < 10; i++) {

        cout << "\t正在处理第" << i + 1 << "组数据....." << endl;

        Read\_Data(a, Rand\_File[i]);

        start = clock();

        Bubble\_Sort(a);

        end = clock();

        duration = (**double**)end - start;

**if** (Is\_Order(a)) cout << "\t\t排序完成\n";

**else** cout << "\t\t排序失败\n";

        printf("\t第%d组数据处理时间为%.3lf毫秒\n\n", i + 1, duration);

        //cout << "\t第" << i + 1 << "组数据处理时间为:" << duration << " 毫秒" << endl << endl;

    }

    cout << "4.快速排序算法:" << endl;

    duration = 0;

**for** (**int** i = 0; i < 10; i++) {

        cout << "\t正在处理第" << i + 1 << "组数据....." << endl;

        Read\_Data(a, Rand\_File[i]);

        start = clock();

        QuickSort(a);

        end = clock();

        duration = (**double**)end - start;

        //duration = t\_end - t\_start;

**if** (Is\_Order(a)) cout << "\t\t排序完成\n";

**else** cout << "\t\t排序失败\n";

        printf("\t第%d组数据处理时间为%.3lf毫秒\n\n", i + 1, duration);

        //cout << "\t第" << i + 1 << "组数据处理时间为:" << duration << " 毫秒" << endl << endl;

    }

    cout << "5.选择排序算法:" << endl;

    duration = 0;

**for** (**int** i = 0; i < 10; i++) {

        cout << "\t正在处理第" << i + 1 << "组数据....." << endl;

        Read\_Data(a, Rand\_File[i]);

        start = clock();

        SelectionSort(a);

        end = clock();

        duration = (**double**)end - start;

        //duration = t\_end - t\_start;

**if** (Is\_Order(a)) cout << "\t\t排序完成\n";

**else** cout << "\t\t排序失败\n";

        printf("\t第%d组数据处理时间为%.3lf毫秒\n\n", i + 1, duration);

        //cout << "\t第" << i + 1 << "组数据处理时间为:" << duration << " 毫秒" << endl << endl;

    }

    cout << "6.堆排序算法:" << endl;

    duration = 0;

**for** (**int** i = 0; i < 10; i++) {

        cout << "\t正在处理第" << i + 1 << "组数据....." << endl;

        Read\_Data(a, Rand\_File[i]);

        start = clock();

        HeapSort(a);

        end = clock();

        duration = (**double**)end - start;

        //duration = t\_end - t\_start;

**if** (Is\_Order(a)) cout << "\t\t排序完成\n";

**else** cout << "\t\t排序失败\n";

        printf("\t第%d组数据处理时间为%.3lf毫秒\n\n", i + 1, duration);

        //cout << "\t第" << i + 1 << "组数据处理时间为:" << duration << " 毫秒" << endl << endl;

    }

    cout << "7.归并排序算法:" << endl;

    duration = 0;

**for** (**int** i = 0; i < 10; i++) {

        cout << "\t正在处理第" << i + 1 << "组数据....." << endl;

        Read\_Data(a, Rand\_File[i]);

        start = clock();

        MergeSort(a);

        end = clock();

        duration = (**double**)end - start;

        //duration = t\_end - t\_start;

**if** (Is\_Order(a)) cout << "\t\t排序完成\n";

**else** cout << "\t\t排序失败\n";

        printf("\t第%d组数据处理时间为%.3lf毫秒\n\n", i + 1, duration);

        //cout << "\t第" << i + 1 << "组数据处理时间为:" << duration << " 毫秒" << endl << endl;

    }

    cout << "8.基数排序算法:" << endl;

    duration = 0;

**for** (**int** i = 0; i < 10; i++) {

        cout << "\t正在处理第" << i + 1 << "组数据....." << endl;

        Read\_Data(a, Rand\_File[i]);

        start = clock();

        radixsort(a);

        end = clock();

        duration = (**double**)end - start;

        //duration = t\_end - t\_start;

**if** (Is\_Order(a)) cout << "\t\t排序完成\n";

**else** cout << "\t\t排序失败\n";

        printf("\t第%d组数据处理时间为%.3lf毫秒\n\n", i + 1, duration);

        //cout << "\t第" << i + 1 << "组数据处理时间为:" << duration << " 毫秒" << endl << endl;

    }

**delete** a;

}

**int** main()

{

    //GetRand();

**while** (1)

    {

        cout << "1.重新生成数据" << endl << "2.计算时间\n3.退出\n";

        cout << "请输入:";

**int** cho;

        cin >> cho;

**switch** (cho)

        {

**case** 1:GetRand(); system("pause"); system("cls"); **break**;

**case** 2:Get\_Time(); system("pause"); system("cls"); **break**;

**case** 3:**return** 0;

        }

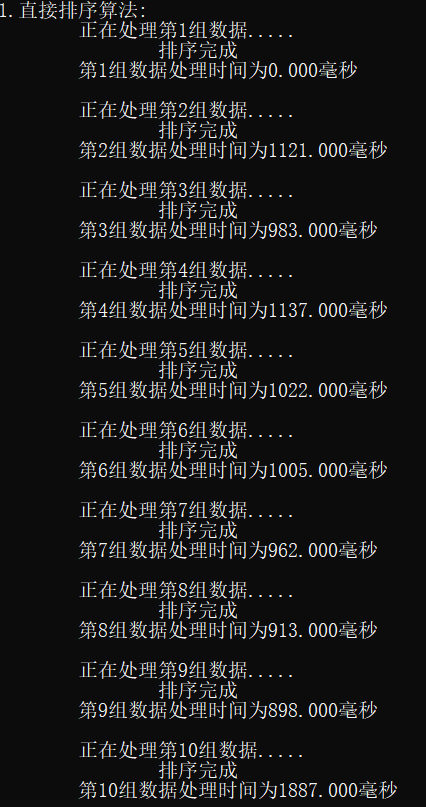
    }

**return** 0;

}

4测试数据和结果

测试数据为随机生成的十组数据，每组数据有50000个数。结果太长，只截取其中一部分：



5.算法的时间复杂度

该算法的时间复杂度就是各个排序所需的时间复杂度，如下表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 时间复杂度 | 最坏情况 | 空间复杂度 | 稳定性 |
| 直接插入排序 | O(n^2) | O(n^2) | O(1) | 稳定 |
| 希尔排序 | O(n^1.5) | O(n^1.5) | O(1) | 不稳定 |
| 冒泡排序 | O(n^2) | O(n^2) | O(1) | 稳定 |
| 快速排序 | O(nlogn) | O(n^2) | O(logn) | 不稳定 |
| 简单选择排序 | O(n^2) | O(n^2) | O(1) | 不稳定 |
| 堆排序 | O(nlogn) | O(nlogn) | O(1) | 不稳定 |
| 归并排序 | O(nlogn) | O(nlogn) | O(n) | 稳定 |
| 基数排序 | O(dn) | O(dn) | O(n) | 稳定 |

1. 【2】地铁修建（选做）（图）

1.数据结构

这题用的数据结构是图的基本数据结构，只包含边以及边的权值。

**struct** arc

{

**int** a;

**int** b;

**int** c;

}arc[maxx];

2.算法设计思想

这题使用了并查集的方法。首先将边按照权值从小到大排序，然后按照升序顺序依次插入并查集中，直到起点和终点都被插入了同一个集合，那么那个最后插入的边一定是最少需要的天数

3.源代码

#include<iostream>

#include<algorithm>

#include<vector>

**using** **namespace** std;

**const** **int** maxx = 2e5 + 10;

**int** parent[maxx];

**struct** arc

{

**int** a;

**int** b;

**int** c;

}arc[maxx];

**int** cmp(**struct** arc a, **struct** arc b)

{

**return** a.c < b.c;

}

**int** find\_root(**int** x)

{

**if** (parent[x] == x) **return** x;

**else** **return** parent[x] = find\_root(parent[x]);

}

**void** union\_arc(**int** x, **int** y)

{

**int** x\_root = find\_root(x);

**int** y\_root = find\_root(y);

    parent[x\_root] = y\_root;

}

**int** main()

{

**int** n, m, min,num=0;

    cin >> n >> m;

**for** (**int** i = 0; i < m; i++)

        cin >> arc[i].a >> arc[i].b >> arc[i].c;

    sort(arc, arc + m, cmp);

**for** (**int** i = 1; i <= n; i++)

        parent[i] = i;

**for** (**int** i = 0; i < m; i++)

    {

**int** x = arc[i].a;

**int** y = arc[i].b;

        union\_arc(x, y);

        //cout << find\_root(1) << " " << find\_root(n) << endl;

**if** (find\_root(1) == find\_root(n))

        {

            cout << arc[i].c;

**break**;

        }

    }

**return** 0;

}

4.测试数据和结果

样例输入

6 6

1 2 4

2 3 4

3 6 7

1 4 2

4 5 5

5 6 6

输出：



5.算法的时间复杂度

在并查集算法中的时间复杂度主要由结合时查找根过程生成，因此时间复杂度在最坏情况为O(N)。

1. 【2】URL映射（选做）（字符串）

1.数据结构

该题主要采用字符串匹配的方法，匹配是否有相同的url。

2.算法设计思想

从头开始匹配，匹配到的第一个不相同的字符一定是映射规则里的“<”，如果不是，说明这个URL和这个映射不规则不匹配，需要匹配下一个映射规则。如果“<”匹配了，那么就需要判断映射规则里的类型是int，str还是path。如果是int，那么就需要匹配URL里的数字，注意需要去掉前导零；如果是str，那么就匹配URL中的字符串，如果是path，那么一定可以输出。

如果匹配成功了，那么就先输出匹配的参数，之后按匹配的顺序输出最后的参数。

3.源代码

#include<iostream>

#include<string>

#include<vector>

**using** **namespace** std;

**const** **int** n = 200;

string p[n], r[n], des[n];

**bool** match(string& p, string& s,**int** mathod)//p为已知，s为待判断

{

**int** len1 = p.length(), len2 = s.length();

**int** i = 0, j = 0;

**while** (i < len1 && j < len2){                     //长度判断

**if** (p[i] == s[j]){                              //如果两个相等，下一个继续判断

            i++; j++;

        }

**else** {

**if** (p[i++] != '<') **return** **false**;         //不一样了，开始判断是不是<> ，如果是就开始判断类型相等不

**if** (mathod) cout << " ";                  //method=1， 是输出模式，先输出

**int** flag = 0;

**if** (p[i] == 'i') {                          //《int》类型

**while** (s[j] && isdigit(s[j]))           //是数字

                {

**if** (s[j] != '0') flag = 1;          //不是0 记录

**if** (flag && mathod) cout << s[j];

                    j++;

                }

**if** (!flag) **return** **false**;                //如果都是0，没有非0数字，就返回错误

                i += 4;

            }

**else** **if** (p[i] == 's') {

**while** (s[j] && s[j] != '/')

                {

                    flag = 1;

**if** (flag && mathod) cout << s[j];

                    j++;

                }

**if** (!flag) **return** **false**;

                i += 4;

            }

**else** **if** (p[i] == 'p')

            {

**if** (mathod)

                {

**while** (s[j]) {

                        cout << s[j];

                        j++;

                    }

                }

**return** **true**;

            }

        }

    }

**return** i == len1 && j == len2;

}

**int** main()

{

**int** n, m;

    cin >> n >> m;

**for** (**int** i = 0; i < n; i++) cin >> p[i] >> r[i];

**for** (**int** i = 0; i < m; i++)

        cin >> des[i];

**for** (**int** i = 0; i < m; i++)

    {

**int** sign = 0;

**for** (**int** j = 0; j < n; j++)

        {

**if** (match(p[j], des[i], 0)) {

                cout << r[j];

                match(p[j], des[i], 1);

                sign = 1;

**break**;

            }

        }

**if** (!sign) cout << "404";

        cout << endl;

    }

**return** 0;

}

4.测试数据和结果

样例输入：

5 4

/articles/2003/ special\_case\_2003  
/articles/<int>/ year\_archive  
/articles/<int>/<int>/ month\_archive  
/articles/<int>/<int>/<str>/ article\_detail  
/static/<path> static\_serve  
/articles/2004/  
/articles/1985/09/aloha/  
/articles/hello/  
/static/js/jquery.js

输出：



5.算法的时间复杂度

该算法的时间复杂度主要由匹配以及匹配后的输出消耗。设有n个字符串，平均长度为m，那么时间复杂度为o(n\*m)。

6.算法的改进方法

可以在先判断的时候就将输出的模式存入一个字符串中，如果返回true，直接可以输出，如果是false，那么将输出字符串清空，继续判断下一条是否匹配。这样的话可以减少一次进入函数的次数。

除此之外，也可以优化匹配的方式，比如以“/”为分割，将映射规则与待匹配字符串分割为若干小块，将若干小块进行比较。

1. 【3】连连看（选做）（图）

1.数据结构

该题运用矩阵存储图，矩阵为12\*22，存储方块为[1~11][1~21]，周围一圈置为-1，-1代表可以连。

2.算法设计思想

连连看的连线可以分为三种：不转折，转折一次，转折两次。

不转折就是行相同或者列相同，从行（列）较小的点开始往较大的查询，如果有不是-1的点就说明不能连通。

转折一次就是如下图，转化成两个星星到AB两点能否不转折连通。

A

B

转折两次就是A的横向或纵向上是否存在一点（星号），可以与A不转折相连并与B转折一次相连。

A

B

3.源代码

#include<iostream>

#include<vector>

#include<Windows.h>

#include<algorithm>

#include<iomanip>

#include<ctime>

**using** **namespace** std;

**int** map[12][22];

//随机创建地图

**void** Get\_New\_Map()

{

**int** a[200];

**int** num[10] = { 0 };

**int** sum[10];

**int** pos;

**int** pattern;

    srand(time(NULL));

**for** (**int** i = 0; i < 200; i++) a[i] = 1;

**for** (**int** i = 0; i < 12; i++)

**for** (**int** j = 0; j < 22; j++) map[i][j] = -1;

**for** (**int** i = 0; i < 100; i++)

    {

        pos = rand() % 200;//随机生成位置，前两位是第几列，个位是第几行

**while** (a[pos] == 0)//如果这个位置没有放过了数据，那么重新生成

            pos = rand() % 200;

        pattern = rand() % 10;//随机生成类型

        a[pos] = 0;

**int** x = pos % 10+1, y = pos / 10+1;//计算第几行第几列

        map[x][y] = pattern;//将点的类型赋值给图

        //由于需要是偶数个点，一个类型要生成两个位置

        pos = rand() % 200;

**while** (a[pos] == 0)

            pos = rand() % 200;

        x = pos % 10 + 1; y = pos / 10 + 1;

        map[x][y] = pattern;

        a[pos] = 0;

    }

}

//打印图

**void** Print()

{

    cout << "    ";

    SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED);

**for** (**int** i = 0; i < 20; i++)

        cout  << setw(4) << std::left << i+1;

    cout << endl;

    SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);

**for** (**int** i = 1; i <= 10; i++)

    {

        SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED);

        cout << setw(4) << std::left << i;

        SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);

**for** (**int** j = 1; j <= 20; j++)

        {

**if** (map[i][j] != -1)

                cout << setw(4) << std::left << map[i][j];

**else** cout << "    ";

        }

        SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED);

        cout << setw(4) << std::left << i;

        cout << endl;

    }

    cout << "    ";

    SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED);

**for** (**int** i = 0; i < 20; i++)

        cout << setw(4) << std::left << i + 1;

    cout << endl;

    SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);

}

//判断行相等时能否消去

**bool** Row\_Equal(**int** y1, **int** y2, **int** x1,**int** x2)

{

**if** (x1 != x2) **return** **false**;

    //cout << "行相等" << endl;

**if** (y1 == y2 - 1) **return** **true**;

    //cout << "非相邻" << endl;

**int** sign = 0;

    //判断能否从左到右一条直线连通

**for** (**int** i = y1 + 1; i < y2; i++)

    {

**if** (map[x1][i] != -1)

        {

**return** **false**;//sign = 1;

        }

    }

**return** **true**;

}

//判断列相等时能否消去

**bool** Col\_Equal(**int** x1, **int** x2, **int** y1,**int** y2)

{

**if** (y1 != y2) **return** **false**;

**if** (x2 - x1 == 1) **return** **true**;

**int** sign = 0;

**for** (**int** i = x1 + 1; i < x2; i++)

    {

        //cout << "map[" << i << "][" << y1 << "]=" << map[i][y1] << endl;

**if** (map[i][y1] != -1)

        {

**return** **false**;//sign = 1;

        }

    }

**return** **true**;

}

//不转折

**bool** Is\_Zero\_Turn(**int** x1, **int** x2, **int** y1, **int** y2)

{

**if** (Row\_Equal(min(y1, y2),max(y1,y2), x1, x2))  **return** **true**;

**if** (Col\_Equal(min(x1,x2),max(x1, x2), y1, y2))**return** **true**;

**return** **false**;

}

//转折一次

**bool** Is\_One\_Turn(**int** x1, **int** x2, **int** y1, **int** y2)

{

**if** (Is\_Zero\_Turn(x1, x1, y2, y1) && Is\_Zero\_Turn(x1, x2, y2, y2)) **return** **true**;

**if** (Is\_Zero\_Turn(x2, x1, y1, y1) && Is\_Zero\_Turn(x2, x2, y1, y2)) **return** **true**;

**return** **false**;

}

//转折两次

**bool** Is\_Two\_Turn(**int** x1, **int** y1, **int** x2, **int** y2)

{

    //向上找

**int** sign = 0;

**for** (**int** i = x1 - 1; i >= 0; i--)

    {

**if** (map[i][y1] != -1) **break**;

**if** (Is\_One\_Turn(i, x2, y1, y2)) **return** **true**;

    }

    //向下找

**for** (**int** i = x1 + 1; i <= 11; i++)

    {

**if** (map[i][y1] != -1)**break**;

**if** (Is\_One\_Turn(i, x2, y1, y2)) **return** **true**;

    }

    //向左找

**for** (**int** i = y1 - 1; i >= 0; i--)

    {

**if** (map[x1][i] != -1) **break**;

**if** (Is\_One\_Turn(x1, x2, i, y2))**return** **true**;

    }

    //向右找

**for** (**int** i = y1 + 1; i <= 21; i++)

    {

**if** (map[x1][i] != -1) **break**;

**if** (Is\_One\_Turn(x1, x2, i, y2)) **return** **true**;

    }

**return** **false**;

}

//判断能否消去

**void** Erase(**int** x1,**int** y1,**int** x2,**int** y2)

{

    //不进行转折

**if** (Is\_Zero\_Turn(x1,x2,y1,y2))

    {

        cout << "能够消去" << endl;

        map[x1][y1] = -1; map[x2][y2] = -1;

**return**;

    }

    //进行一次转折

**if** (Is\_One\_Turn(x1, x2, y1, y2))

    {

        cout << "能够消去" << endl;

        map[x1][y1] = -1; map[x2][y2] = -1;

**return**;

    }

    //进行两次转折

**if** (Is\_Two\_Turn(x1, y1, x2, y2))

    {

        cout << "可以消去" << endl;

        map[x1][y1] = -1; map[x2][y2] = -1;

**return**;

    }

    cout << "无法消去！" << endl;

}

//判断是否完成

**bool** Is\_Finished()

{

**for** (**int** i = 0; i < 12; i++)

**for** (**int** j = 0; j < 22; j++)

**if** (map[i][j] != -1)**return** **false**;

**return** **true**;

}

**int** main()

{

    Get\_New\_Map();

    Print();

**int** x1, x2, y1, y2;

**while** (!Is\_Finished())

    {

        system("cls");

        Print();

        //cout << map[0][0] << endl;

        cout << "请输入第一个图案的坐标:";

        cin >> x1 >> y1;

        cout << "请输入第二个图案的坐标:";

        cin >> x2 >> y2;

**if** (x1 == x2 && y1 == y2) { cout << "请不要输入一样的坐标!\n" << endl; system("pause"); **continue**; }

**if**(map[x1][y1]!=map[x2][y2]) { cout << "图案不一样，无法消除!\n" << endl; system("pause"); **continue**; }

**if** (x1 < 1 || x1>10 || y1 < 1 || y1>20 || x2 < 1 || x2>10 || y1 < 1 || y2>20) { cout << "请输入有效坐标!\n" << endl; system("pause"); **continue**; }

        Erase(x1, y1, x2, y2);

        system("pause");

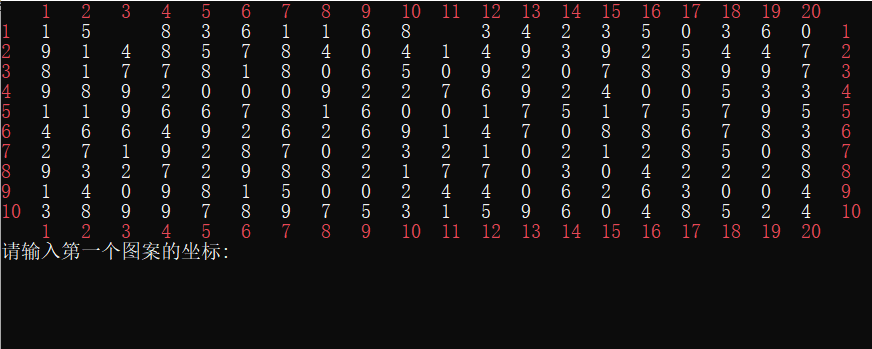
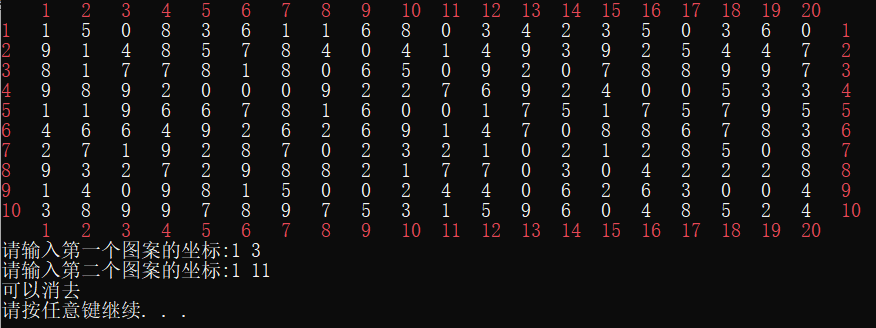
    }

    cout << "You Win!!!!!";

**return** 0;

}

4.测试数据和结果



5.算法的时间复杂度

算法的复杂度主要由寻找是否存在相连的线划分。最多是转折两次的情况下，将每一个横向的点都进行了一次转折一次的判断。因此时间复杂度为O(N^2)。

1. 【4】B-树应用（选做）（查找）

1.数据结构

本题主要由两个结构体，一个是树的结点与B-树类型，另一个是B-树的查找类型。其中查找类型具体是如果查找成功就返回所在结点以及下标，如果查找不成功，就返回数据的插入位置。

**typedef** **struct** BTNode {

**int** keynum;                 //关键字个数，即结点的大小

**struct** BTNode\* parent;      //指向双亲结点

    KeyType key[M + 1];         //关键字向量，0号地址未用

**struct** BTNode\* ptr[M + 1];  //子树指针向量

    BTNode()

    {

        keynum = 0;

        parent = NULL;

**for** (**int** i = 0; i < M + 1; i++) {

            ptr[i] = NULL;

        }

    }

}BTNode,\* BTree;

//B-树的查找结果类型

**typedef** **struct** {

    BTNode\* pt;         //指向找到的结点

**int** i;              //1....M，在结点中的关键字序号

**int** tag;            //1:查找成功 0:查找失败

}Result;

2.算法设计思想

查找：

由B-树的定义可以知道，B-树的查找过程与二叉排序树的查找过程类似。首先从根结点开始找，如果有相同的，那么就返回该位置；如果没有相同的，找到查找值在哪两个关键字数组之间，然后根据下标进入这个子树继续查找。

插入：

首先通过查找确定插入的位置。插入之后如果关键字数量<m-1（m为阶数），那么插入成功；如果=m-1，那么就是需要分裂结点。将中间的结点插入父节点，左半部分和右半部分分裂成两个结点连接在父节点之下。如果父节点的关键字数量仍旧超了，那么继续分裂。

删除：

首先通过查找确定删除的关键字的位置。如果插入的不是最底层的关键字，而是上层的Ki，那么就将Pi子树中的最小值换到Ki，转化成删除最底层结点。下面讲述怎么删除最底层结点。

如果删除关键字后，结点数>，那么可以直接删除；如果删除后结点数<，但是它的左（右）兄弟>，那么可以将其左（右）兄弟的最大（小）关键字移到父节点中，将父节点上的关键字下移到该删除的结点中；如果删除关键字后，结点以及其左右兄弟关键字数都<，那么需要将其左（右）结点以及父结点中用于分割这两个结点的关键字合并成一个结点，如果这样使得父节点也<，那么仍然需要进行上述处理。

3.源代码

#include<iostream>

#include<string>

#include<vector>

#include<fstream>

#include<algorithm>

**using** **namespace** std;

#define KeyType int

#define Record string

#define M 3

**typedef** **struct** BTNode {

**int** keynum;                 //关键字个数，即结点的大小

**struct** BTNode\* parent;      //指向双亲结点

    KeyType key[M + 1];         //关键字向量，0号地址未用

**struct** BTNode\* ptr[M + 1];  //子树指针向量

    BTNode()

    {

        keynum = 0;

        parent = NULL;

**for** (**int** i = 0; i < M + 1; i++) {

            ptr[i] = NULL;

        }

    }

}BTNode,\* BTree;

BTree T = NULL;

//B-树的查找结果类型

**typedef** **struct** {

    BTNode\* pt;         //指向找到的结点

**int** i;              //1....M，在结点中的关键字序号

**int** tag;            //1:查找成功 0:查找失败

}Result;

//在p->key[1...keynun]中查找i 使得 p->key[i-1]<K<=p->key[i]

**int** Search(BTree p, **int** K)

{

**int** i = 1;

**while** (i <= p->keynum && K > p->key[i]) i++;

**return** i;

}

//查找树T中关键字K的位置

Result SearchBTree(BTree T, KeyType K)

{

    BTree p = T,q = NULL;                                   //p为插入的结点，q为其双亲结点

**bool** found = **false**;

**int** i = 0;

**while** (p && !found)

    {

        i = Search(p, K);

**if** (i <= p->keynum && p->key[i] == K) found = **true**;

**else** { q = p; p = p->ptr[i - 1]; }                   //结点下移

    }

**if** (found) **return**{ p, i, 1 };

**else** **return** { q, i, 0 };                                //查找不成功，返回K的插入位置信息

}

//将x和ap分别插入到q->key[i+1]和q->ptr[i+1]

**void** Insert(BTree& q, **int** i, KeyType x, BTree ap)

{

**int** j, n = q->keynum;

**for** (j = n; j >= i; j--)         //插入位置后的数据后移

    {

        q->key[j + 1] = q->key[j];

        q->ptr[j + 1] = q->ptr[j];

    }

    q->key[i] = x;

    q->ptr[i] = ap;

**if** (ap != NULL) ap->parent = q;

    q->keynum++;

}

//分裂,将前一半留在原结点，后一半移入ap

**void** Split(BTree& q, **int** s, BTree& ap)

{

**int** i, j, n = q->keynum;

    ap = **new** BTNode;

    ap->ptr[0] = q->ptr[s];

**for** (i = s + 1, j = 1; i <= n; i++, j++) {           //将后一半移入ap结点里

        ap->key[j] = q->key[i];

        ap->ptr[j] = q->ptr[i];

    }

    ap->keynum = n - s;                                  //修改结点关键字的数量

    ap->parent = q->parent;

**for** (i = 0; i <= n - s; i++)                     //后一半结点的孩子的父节点指向该结点

**if** (ap->ptr[i] != NULL) ap->ptr[i]->parent = ap;

    q->keynum = s - 1;                                   //s前面的数

}

//创建结点

**void** NewRoot(BTree& T, BTree p, **int** x, BTree ap)

{

    T = **new** BTNode;

    T->keynum = 1;

    T->ptr[0] = p;

    T->ptr[1] = ap;

    T->key[1] = x;

**if** (p) p->parent = T;

**if** (ap) ap->parent = T;

    T->parent = NULL;

}

//插入,将关键字K插入q结点的第i个位置

**void** InsertBTree(BTree& T, KeyType K, BTree q, **int** i)

{

**int** x = K, s;

    BTree ap = NULL;

**bool** finished = **false**;

**if** (q == NULL)                          //q为空，需要重新建立一个根结点

    { NewRoot(T, NULL, K, NULL); **return**; }

**while** (q && !finished)

    {

        Insert(q, i, x, ap);

**if** (q->keynum < M) finished = **true**; //插入完成。关键字个数没有超过m-1

**else** {                              //分裂结点

            s = (M + 1) / 2;                //中间位置

            Split(q, s, ap);                //分裂，将q->key[s+1...m],q->ptr[s...m]移入新节点ap

            x = q->key[s];                   //x为中间位置，分裂之后插入上层双亲结点中

**if** (q->parent != NULL)

            {

                q = q->parent;

                i = Search(q, x);           //查找在双亲结点里插入的位置

            }

**else** **break**;                     //需要新的结点，T是空树或者根结点也分裂成了q和ap

        }

    }

**if** (!finished) NewRoot(T, q, x, ap);

}

//删除关键字,将key[i]与ptr[i]删除

**void** Remove(BTree& p, **int** i)

{

**for** (**int** j = i; i < p->keynum; i++)

    {

        p->key[j] = p->key[j + 1];

        p->ptr[j] = p->ptr[j + 1];

    }

    p->keynum--;

}

//用结点q中的ptr[i]所指子树中的最小关键字来覆盖

**void** PassOver(BTree& q, **int** i)

{

    BTree child = q->ptr[i];

**while** (child->ptr[0] != NULL) child = child->ptr[0];

    q->key[i] = child->key[1];

    q = child;

}

//数据数量少了，需要合并/借

**void** Adjust(BTree& T, BTree& p)

{

    BTree pa = p->parent;

**if** (pa == NULL)

    {               //空的根结点，需要删除，树的深度减一

        T = p;

        p = p->parent;

**return**;

    }

    BTree leftchild, rightchild, pr;//记录p结点的左右兄弟

**int** finished = 0, r = 0;        //r用来记录p的位置是双亲结点的哪个下标

**while** (!finished)

    {

        r = 0;

**while** (pa->ptr[r] != p) r++;                                     //确定p在pa的位置

**if** (r == 0)

        {

            r++;

            leftchild = NULL;

            rightchild = pa->ptr[r];

        }

**else** **if** (r == pa->keynum)

        {

            leftchild = pa->ptr[r - 1];

            rightchild = NULL;

        }

**else**

        {

            leftchild = pa->ptr[r - 1];

            rightchild = pa->ptr[r + 1];

        }

**if** (r > 0 && leftchild != NULL && (leftchild->keynum > (M - 1) / 2)) //向左兄弟借值

        {

            p->keynum++;

**for** (**int** i = p->keynum; i > 1; i--)                               //关键字右移

            {

                p->key[i] = p->key[i - 1];

                p->ptr[i] = p->ptr[i - 1];

            }

            p->key[1] = pa->key[r];                                           //父结点的下移

            p->ptr[1] = p->ptr[0];

            p->ptr[0] = leftchild->ptr[leftchild->keynum];

**if** (p->ptr[0] != NULL)                                           //修改子树的父结点

                p->ptr[0]->parent = p;

            pa->key[r] = leftchild->key[leftchild->keynum];                    //左兄弟的上移

            leftchild->keynum--;

            finished = 1;

**break**;

        }

**else** **if** (pa->keynum > r && rightchild != NULL && (rightchild->keynum > (M - 1) / 2))//向右兄弟借

        {

            p->keynum++;

            p->key[p->keynum] = pa->key[r];                        //父结点下移

            p->ptr[p->keynum] = rightchild->ptr[0];

**if** (p->ptr[p->keynum] != NULL)                        //修改子树的父结点

                p->ptr[p->keynum]->parent = p;

            pa->key[r] = rightchild->key[1];                  //右兄弟上移

            rightchild->ptr[0] = rightchild->ptr[1];

**for** (**int** i = 1; i < rightchild->keynum; i++)      //右兄弟结点关键字左移

            {

                rightchild->key[i] = rightchild->key[i + 1];

                rightchild->ptr[i] = rightchild->ptr[i + 1];

            }

            rightchild->keynum--;

            finished = 1;

**break**;

        }

        r = 0;

**while** (pa->ptr[r] != p) r++;

**if** (r > 0 && (pa->ptr[r - 1]->keynum <= (M - 1) / 2))//与左兄弟合并

        {

            leftchild = pa->ptr[r - 1];

            p->keynum++;

**for** (**int** i = p->keynum; i > 1; i--)               //将p结点关键字右移

            {

                p->key[i] = p->key[i - 1];

                p->ptr[i] = p->ptr[i - 1];

            }

            p->key[1] = pa->key[r];                           //将父结点的下移

            p->ptr[1] = p->ptr[0];

            pa->ptr[r] = leftchild;

**for** (**int** i = 1; i <= leftchild->keynum; i++)  //将p的关键字移到左兄弟中

            {

                leftchild->key[leftchild->keynum+i] = p->key[i];

                leftchild->ptr[leftchild->keynum + i] = p->ptr[i];

            }

**if** (p->ptr[0])                                   //将子树双亲结点设置

**for** (**int** i = 1; i <= leftchild->keynum; i++)

**if** (p->ptr[p->keynum + i])p->ptr[p->keynum + i]->parent = leftchild;

            leftchild->keynum += p->keynum;

**for** (**int** i = r; i < pa->keynum; i++)          //将双亲结点的关键字移位

            {

                pa->key[i] = pa->key[i + 1];

                pa->ptr[i] = pa->ptr[i + 1];

            }

            pa->keynum--;

            pr = p;

            free(pr);

            pr = NULL;

            p = leftchild;

        }

**else** //与右兄弟合并

        {

            rightchild = pa->ptr[r + 1];

**if** (r == 0) r++;

            p->keynum++;

            p->key[p->keynum] = pa->key[r];                                //父结点的关键字与p合并

            p->ptr[p->keynum] = rightchild->ptr[0];                        //从右兄弟左移一个指针

            rightchild->keynum += p->keynum;

            pa->ptr[r - 1] = rightchild;

**for** (**int** i = 1; i <= rightchild->keynum - p->keynum; i++)  //将p右兄弟关键字右移

            {

                rightchild->key[p->keynum + i] = rightchild->key[i];

                rightchild->ptr[p->keynum + i] = rightchild->ptr[i];

            }

**for** (**int** i = 1; i <= p->keynum; i++)                      //将p的关键字插入右兄弟

            {

                rightchild->key[i] = p->key[i];

                rightchild->ptr[i] = p->ptr[i];

            }

            rightchild->ptr[0] = p->ptr[0];                               //修改p中子树双亲为右兄弟

**if** (p->ptr[0])

**for** (**int** i = 1; i <= p->keynum; i++)

**if** (p->ptr[p->keynum + i]) p->ptr[p->keynum + i]->parent = rightchild;

**for** (**int** i = r; i < pa->keynum; i++)                      //将双亲关键字左移

            {

                pa->key[i] = pa->key[i + 1];

                pa->ptr[i] = pa->ptr[i + 1];

            }

            pa->keynum--;

            pr = p;

            free(pr);

            pr = NULL;

            p = rightchild;

        }

        pa = pa->parent;

**if** (p->parent->keynum >= (M - 1) / 2 || (pa == NULL && p->parent->keynum > 0))

            finished = 1;

**else** **if** (pa == NULL)                                            //根结点空，删除

        {

            pr = T;

            T = p;

            free(pr);

            pr = NULL;

            finished = 1;

        }

        p = p->parent;

    }

}

//删除p节点的第i个关键字

**void** DeleteBTree(BTree& T, BTree& p, **int** i)

{

**if** (p->ptr[i] != NULL) {     //不是最下层的点

        PassOver(p, i);             //用结点p中的ptr[i]子树的最小关键字覆盖key[i]

        DeleteBTree(T, p, 1);       //转换为删除最底层的分支节点中删除一个关键字

    }

**else** {                          //是最下层的结点

        Remove(p, i);

**if** (p->keynum < (M - 1) / 2)//关键字个数小于

            Adjust(T, p);

    }

}

// 打印树中结点信息

**void** ShowBtree(BTree& p)

{

**if** (p == NULL) {

        puts("B-树未创建!");

**return**;

    }

**bool** have\_child = **false**;

    printf("[");

**for** (**int** i = 1; i <= p->keynum; i++) {

**if** (i == 1);

**else** cout<<" ";

        cout<<p->key[i];

    }

    printf("]");

**for** (**int** i = 0; i <= p->keynum; i++) {

**if** (p->ptr[i] != NULL) {

**if** (i == 0) cout<<"<";

**else** cout<<",";

            ShowBtree(p->ptr[i]);

            have\_child = **true**;

        }

    }

**if** (have\_child) cout << ">";

}

// 创建树

**void** CreateBtree()

{

    T = **new** BTNode;

    T->keynum = 0;

}

// 销毁

**void** Destory(BTree& t)

{

**int** i = 0;

**if** (t != NULL) {

**while** (i < t->keynum) {

            Destory(t->ptr[i]);

            free(t->ptr[i]);

            i++;

        }

    }

    free(t);

    t = NULL;

}

/\*操作界面函数\*/

//插入关键字

**void** InsertKeytype()

{

    puts("请输入插入的数据:");

    KeyType temp;

    scanf\_s("%d", &temp);

    Result p;

    p=SearchBTree(T, temp);

**if** (p.tag == 0) {

        InsertBTree(T, temp, p.pt, p.i);

        puts("插入成功");

        ShowBtree(T);

        puts("");

    }

**else** puts("数据已存在.");

}

// 查找数据

**void** FindKeytype()

{

    puts("请输入查找的数据:");

    KeyType temp;

    scanf\_s("%d", &temp);

    Result p;

    p=SearchBTree(T, temp);

**if** (p.tag) {

        puts("查找成功");

    }

**else** puts("数据不存在!");

}

//删除

**void** DeleteKeytype()

{

    cout << "请输入需要删除的数据：";

    KeyType p;

    cin >> p;

    Result r;

    r = SearchBTree(T, p);

**if** (r.tag)

    {

        DeleteBTree(T, r.pt, r.i);

        cout << "删除成功!";

        ShowBtree(T);

    }

**else** cout << "数据不存在!";

}

**int** main()

{

    vector<**int**> testData;

**const** **char**\* filename = "btree.txt";

    // 读取文件

    ifstream infile(filename, ios::in);

**if** (!infile.is\_open()) { cout << "文件打开失败！\n"; **return** 0; }

**while** (!infile.eof())

    {

**int** temp;

        infile >> temp;

        Result p;

        p = SearchBTree(T, temp);

**if** (p.tag == 0) InsertBTree(T, temp, p.pt, p.i);

    }

**int** cmd = 0;

**while** (cmd != 7) {

        cout << "1. 插入数据\n";

        cout << "2. 查找数据\n";

        cout << "3. 删除数据\n";

        cout << "4. 显示全部数据\n";

        cout << "0. 退出\n";

        cout << "请选择(0-4): ";

        cin >> cmd;

**switch** (cmd) {

**case** 1:

            InsertKeytype();

**break**;

**case** 2:

            FindKeytype();

**break**;

**case** 3:

            DeleteKeytype();

**break**;

**case** 4:

            ShowBtree(T);

            //puts("");

**break**;

**case** 0:

**return** 0;

        }

        cout << endl;

        system("pause");

        system("cls");

    }

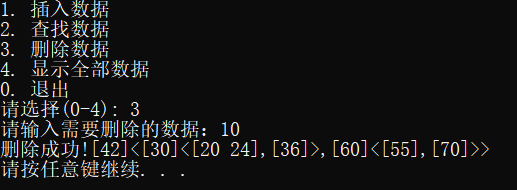
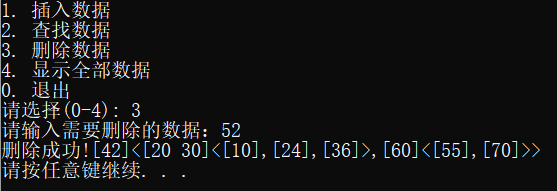
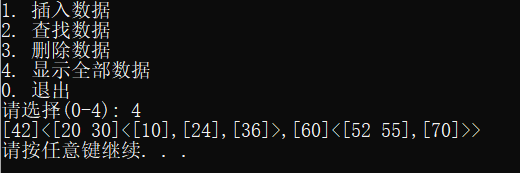
    Destory(T);

**return** 0;

}

4.测试数据和结果

70 10 30 42 60 36 20 55 24 52



5.算法的时间复杂度

B-树的删除操作需要执行查找操作，因此时间复杂度为O(mlognm)。插入过程与删除类似，最坏情况需要回溯O(h),因此时间复杂度为O(mlognm)。

十一、【1】算术表达式求值（选做）（栈）

1. 数据结构

这题数据结构用的是栈的基本操作。

// 栈

**template**<**class** T>

**class** Stack

{

**private**:

    T\* data;

    T\* base;

    T\* top;

**int** stacksize;

**public**:

    Stack(**void**) {

        data = **new** T[STC\_IN\_LEN];

        top = base = data;

        stacksize = STC\_IN\_LEN;

    }

**void** ClearStack() {  // 清空

        top = base;

    }

**bool** StackEmpty() { // 判空

**if** (top == base) **return** **true**;

**else** **return** **false**;

    }

**int** StackLength() {  // 获取栈中元素个数

**return** top - base;

    }

    T GetTop() {  // 获取栈顶

**return** \*(top - 1);

    }

    T\* GetBaseLoca() { // 获取栈底

**return** base;

    }

**void** Push(T num) { // 进栈

**if** (top - base != stacksize) {

            \*(top) = num;

            top++;

        }

**else** {  // 扩展空间

**int** i;

            T\* tmp;

            tmp = **new** T[stacksize + ST\_Cre];

**for** (i = 0; i < stacksize; i++) tmp[i] = data[i];

**delete**[]data;

            data = tmp;

            stacksize += ST\_Cre;

            \*(top) = num;

            top++;

        }

    }

    T Pop() { //出栈

**if** (top != base) {

            top--;

**return** \*(top);

        }

**else** **return** '\0';

    }

**void** ShowStack() {  // 打印栈中元素

**int** i;

**if** (top - base == 0) cout << "空栈" << endl;

**else** {

            cout << "栈底到栈顶：";

**for** (i = 0; i < top - base; i++) {

                cout << data[i] << " ";

            }

            cout << endl;

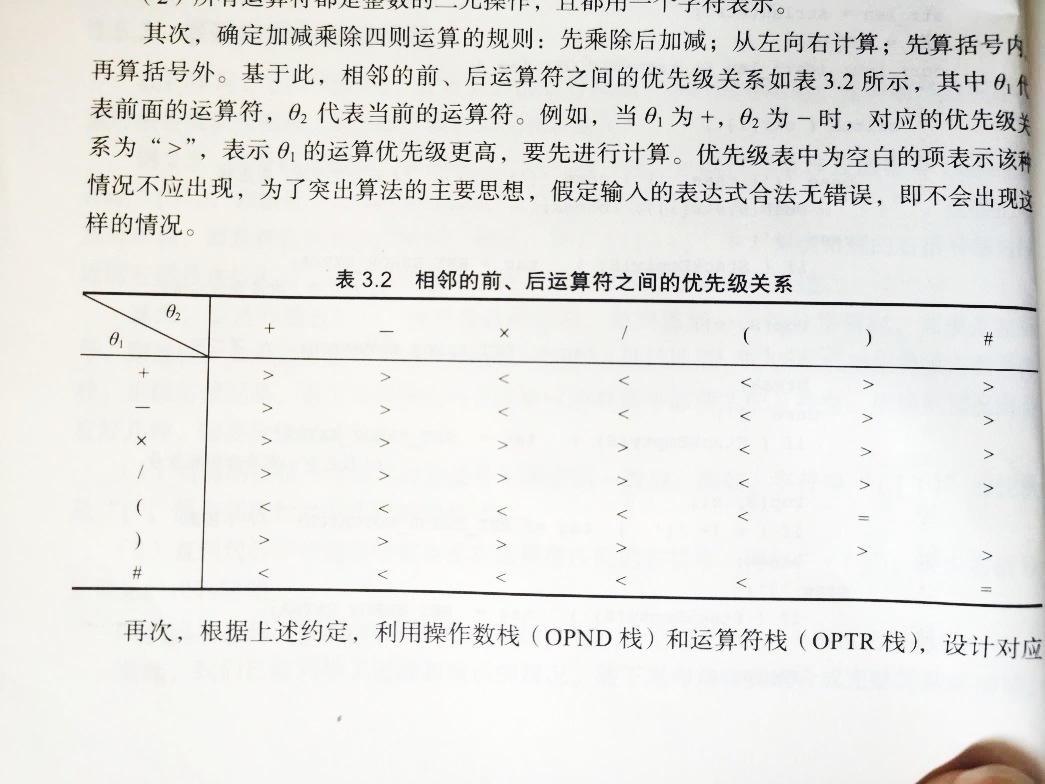
        }

    }

};

2.算法设计思想

首先需要判定各个操作符的优先级



接着读取表达式。如果是数字，那么需要读取相应的数字，将其压入OPND栈。

如果是操作符，那么需要和OPTR栈栈顶的操作符进行优先级比较。如果栈顶运算符优先级低于当前运算符，那么将当前运算符入栈；如果栈顶运算符优先级高于当前运算符，那么将栈顶运算符出栈，同时出栈两个运算数字，根据栈顶运算符运算，得到的新数字入栈，当前运算符也入栈；如果当前运算符优先级等于栈顶运算符，那么将栈顶运算符出栈。

对于运算表达式合理性判断，主要有以下几种情况：1.首尾不是‘#’字符；2.存在非法字符，不是数字，小数点，括号以及加减乘除号；3.运算符号后面还是运算符号（非括号）；4.左括号右边不是数字，右括号左边不是数字；5.小数点之后不是数字；6.括号不匹配。

3.源代码

#include<iostream>

#include<string>

#include<algorithm>

#include<vector>

#include"stack.h"

#include<Windows.h>

**using** **namespace** std;

//#6+15\*(21-8/4)#

//#3+4\*2-(1+1)#

//#9.1+(7+0.1)\*2+78/50#

//获取优先级

**char** Get\_Pririty(**char** a, **char** b)

{

**int** i=-1, j=-1;

**char** f[7][7] = {

        {'>','>','<','<','<','>','>'},

        {'>','>','<','<','<','>','>'},

        {'>','>','>','>','<','>','>'},

        {'>','>','>','>','<','>','>'},

        {'<','<','<','<','<','=','0'},

        {'>','>','>','>','0','>','>'},

        {'<','<','<','<','<','0','='}

    };

**switch** (a)

    {

**case** '+':i = 0; **break**;

**case** '-':i = 1; **break**;

**case** '\*':i = 2; **break**;

**case** '/':i = 3; **break**;

**case** '(':i = 4; **break**;

**case** ')':i = 5; **break**;

**case** '#':i = 6; **break**;

    }

**switch** (b)

    {

**case** '+':j = 0; **break**;

**case** '-':j = 1; **break**;

**case** '\*':j = 2; **break**;

**case** '/':j = 3; **break**;

**case** '(':j = 4; **break**;

**case** ')':j = 5; **break**;

**case** '#':j = 6; **break**;

    }

**if** (i == -1 && j == -1) **return** 'a';

**return** f[i][j];

}

//计算数字

**double** Cal\_Number(**double** x, **char** op, **double** y)

{

**double** q = 0;

**if** (op == '+') q = x + y;

**if** (op == '-') q = x - y;

**if** (op == '\*') q = x \* y;

**if** (op == '/') q = x / y;

    //cout << q << endl;

**return** q;

}

//计算表达式

**void** Cal\_Expression(string str)

{

    Stack<**double**> OPND;

    Stack<**char**> OPTR;

**int** i = 0;

    OPTR.Push(str[i]);

    i++;

**char** theta, thetal;

    thetal = OPTR.GetTop();

**int** step = 0;

**int** sign = 1;

**while** (str[i] != '#' || thetal != '#')

    {

        step++;

**if** (isdigit(str[i]))//是数字，就依次读取数据存入操作数栈

        {

**double** num = 0;

**double** x = 0.1;//如果有小数部分就用这个乘

**int** sign = 0;//记录有没有小数点

**while** (isdigit(str[i]) || str[i] == '.')

            {

**if** (isdigit(str[i]) && sign == 0)

                {

                    num = num \* 10 + str[i] - '0';

                }

**if** (str[i] == '.') sign = 1;

**if** (isdigit(str[i]) && sign)//是数字且经过小数点了

                {

                    num += (str[i] - '0') \* x;

                    x \*= 0.1;

                }

                i++;

            }

            OPND.Push(num);

        }

**else**

        {

**double** a, b;

            thetal = OPTR.GetTop();

            cout << endl;

            cout <<thetal<< Get\_Pririty(thetal, str[i]) <<str[i]<< " ";

**switch** (Get\_Pririty(thetal,str[i]))

            {

**case** '<':OPTR.Push(str[i]); thetal=OPTR.GetTop(); i++; **break**;//栈顶运算符优先级低，则当前运算符压栈

**case** '>'://栈顶运算符优先级高，则将运算数的两个数进行计算，压栈到运算数栈

                theta = OPTR.Pop();

                b = OPND.Pop(); a = OPND.Pop();

                OPND.Push(Cal\_Number(a, theta, b));

**break**;

**case** '='://是括号，脱括号

                theta = OPTR.Pop(); i++; **break**;

            }

        }

        cout << "第" << step << "步:" << endl;

        cout << "    "; OPND.ShowStack(); cout << "    "; OPTR.ShowStack();

        cout << endl;

**if** (OPTR.StackEmpty()) **break**;

    }

**double** result;

    result=OPND.GetTop();

    cout << "答案为" << result;

}

//指向错误

**void** To\_Error(string s,**int** pos)

{

    SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED);

**for** (**int** i = 0; i < s.length(); i++)

**if** (i != pos) cout << " ";

**else** cout << "↑";

    cout << endl;

    SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);

}

//判断表达式是否合理 0:首尾不是#  1：存在非法字符  2：运算符重复  3：括号错误  4：小数点错误 5：括号未成对 -1:正确

**int** Is\_Vaild(string s)

{

**int** len = s.length();

**if** (s[0] != '#' || s[len-1] != '#')

    {

        To\_Error(s,0);

**return** 0;

    }

**if** (!isdigit(s[1])) { To\_Error(s, 1); **return** 6; }

**int** sign = 0;

**for** (**int** i = 1; i < len - 1; i++)

    {

**if** (!isdigit(s[i]) && s[i] != '+' && s[i] != '-' && s[i] != '\*' && s[i] != '/' && s[i] != '(' && s[i] != ')' && s[i]!='.') { To\_Error(s, i); **return** 1; }

**if** ((s[i] == '+' || s[i] == '-' || s[i] == '\*' || s[i] == '/') && (s[i + 1] == '+' || s[i + 1] == '-' || s[i + 1] == '\*' || s[i + 1] == '/' || s[i + 1] == '.'))//运算符后面还是运算符

        {

            To\_Error(s, i + 1);

**return** 2;

        }

**if** ((s[i] == '(' && !isdigit(s[i + 1])) || (s[i] == ')' && !isdigit(s[i - 1])))// 左括号右边不是数字，右括号左边不是数字

        {

            To\_Error(s, i);

**return** 3;

        }

**if** (s[i] == '.' && !isdigit(s[i + 1])) { To\_Error(s, i); **return** 4; }

    }

    Stack<**int**> q;

**int** temp;

**for** (**int** i = 0; i < len; i++)

    {

**if** (s[i] == '(') q.Push(i);

**if** (s[i] == ')') temp = q.Pop();

    }

**if** (!q.StackEmpty()) { To\_Error(s, q.GetTop()); **return** 5; }

**return** -1;

}

**int** main()

{

    string expre;

    cin >> expre;

**int** pos = Is\_Vaild(expre);

**while** (pos != -1)

    {

**switch** (pos)

        {

**case** 0:cout << "首尾不是#" << endl; **break**;

**case** 1:cout << "存在非法字符" << endl; **break**;

**case** 2:cout << "运算符重复" << endl; **break**;

**case** 3:cout << "括号错误" << endl; **break**;

**case** 4:cout << "小数点错误" << endl; **break**;

**case** 5:cout << "括号未成对" << endl; **break**;

**case** 6:cout << "运算符错误" << endl; **break**;

        }

        cout << "\n表达式错误，请重新输入:"<<endl;

        cin >> expre;

        pos = Is\_Vaild(expre);

    }

    Cal\_Expression(expre);

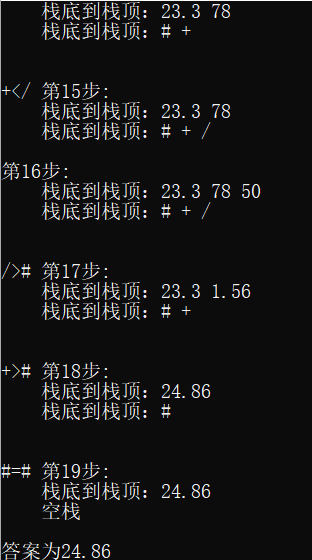
**return** 0;

}

4.测试数据和结果

样例输入：

#9.1+(7+0.1)\*2+78/50#



5.算法的时间复杂度

时间复杂度主要由读取字符串耗费，时间复杂度为O(N)。

十二、结束语

完成各题的全部要求，并在一定程度上对界面进行了美化，使得操作界面更加人性化，并加入了错误检测，使得代码更具健壮性。

第一题代码行数：299

第二题代码行数：214

第三题代码行数：200

第四题代码行数：924

第五题代码行数：410

第六题代码行数：587

第七题代码行数：55

第八题代码行数：81

第九题代码行数：228

第十题代码行数：467

第十一题代码行数：273

总代码行：3738

经过一个学期的数据结构学习，让我学会了如何合理的组织数据、有效的存储数据和处理数据、正确的设计算法以及对算法的复杂度进行分析与评价。通过这门课，我深入理解了各种基本数据结构的逻辑关系、物理存储和基本操作，会结合案例讨论不同数据结构的特点、适用范围以及基本算法复杂度的分析方法，为后续的专业课程学习提供了必要的基础知识。

本学期学习了线性表、栈和队列、数组和广义表、数和二叉树、图、查找和排序等内容。一系列的专项编程训练让我的抽象建模与表示能力大有提升