**南京航空航天大学**

**计算机科学与技术/人工智能学院**

**《数据结构》课程设计报告**

**班 级： 1619303**

**专 业： 软件工程**

**学 号： 061920125**

**姓 名： 郑伟林**

**指导教师： 高航**

**目录**

[一、系统进程统计 4](#_Toc61029942)

[1. 数据结构 4](#_Toc61029943)

[2. 算法思想 4](#_Toc61029944)

[3. 源代码 5](#_Toc61029945)

[4. 测试数据和结果 19](#_Toc61029946)

[5. 小结 20](#_Toc61029947)

[二、迷宫问题 21](#_Toc61029948)

[1. 数据结构 21](#_Toc61029949)

[2. 算法思想 21](#_Toc61029950)

[3. 源代码 21](#_Toc61029951)

[4. 测试数据和结果 26](#_Toc61029952)

[5. 小结 27](#_Toc61029953)

[三、家谱管理系统 27](#_Toc61029954)

[1. 数据结构 27](#_Toc61029955)

[2. 算法思想 28](#_Toc61029956)

[3. 源代码 28](#_Toc61029957)

[4. 测试数据和结果 50](#_Toc61029958)

[5. 小结 52](#_Toc61029959)

[四、Huffman编码与解码 52](#_Toc61029960)

[1. 数据结构 52](#_Toc61029961)

[2. 算法思想 53](#_Toc61029962)

[3. 源代码 53](#_Toc61029963)

[4. 测试数据和结果 60](#_Toc61029964)

[5. 小结 62](#_Toc61029965)

[五、地铁修建 62](#_Toc61029966)

[1. 数据结构 62](#_Toc61029967)

[2. 算法思想 62](#_Toc61029968)

[3. 源代码 63](#_Toc61029969)

[4. 测试数据和结果 64](#_Toc61029970)

[5. 小结 65](#_Toc61029971)

[六、公交线路提示 65](#_Toc61029972)

[1. 数据结构 65](#_Toc61029973)

[2. 算法思想 65](#_Toc61029974)

[3. 源代码 66](#_Toc61029975)

[4. 测试数据和结果 75](#_Toc61029976)

[5. 小结 75](#_Toc61029977)

[七、B-树的应用 75](#_Toc61029978)

[1. 数据结构 75](#_Toc61029979)

[2. 算法思想 76](#_Toc61029980)

[3. 源代码 77](#_Toc61029981)

[4. 测试数据和结果 87](#_Toc61029982)

[5. 小结 88](#_Toc61029983)

[八、排序算法速率比较 89](#_Toc61029984)

[1. 数据结构 89](#_Toc61029985)

[2. 算法思想 89](#_Toc61029986)

[3. 源代码 90](#_Toc61029987)

[4. 测试数据和结果 99](#_Toc61029988)

[5. 小结 100](#_Toc61029989)

[九、数字排序【1】 100](#_Toc61029990)

[1. 数据结构 100](#_Toc61029991)

[2. 算法思想 100](#_Toc61029992)

[3. 源代码 101](#_Toc61029993)

[4. 测试数据和结果 102](#_Toc61029994)

[5. 小结 102](#_Toc61029995)

[十、魔法优惠券【1】 102](#_Toc61029996)

[1. 数据结构 102](#_Toc61029997)

[2. 算法思想 102](#_Toc61029998)

[3. 源代码 103](#_Toc61029999)

[4. 测试数据和结果 105](#_Toc61030000)

[5. 小结 105](#_Toc61030001)

[十一、消除类游戏【2】 105](#_Toc61030002)

[1. 数据结构 105](#_Toc61030003)

[2. 算法思想 105](#_Toc61030004)

[3. 源代码 105](#_Toc61030005)

[4. 测试数据和结果 107](#_Toc61030006)

[5. 小结 108](#_Toc61030007)

[十二、最小二叉树【3】 108](#_Toc61030008)

[1. 数据结构 108](#_Toc61030009)

[2. 算法思想 108](#_Toc61030010)

[3. 源代码 109](#_Toc61030011)

[4. 测试数据和结果 113](#_Toc61030012)

[5. 小结 113](#_Toc61030013)

[十三、课程设计总结 114](#_Toc61030014)

# 系统进程统计

## 数据结构

实现进程存储的包括两个结构：单链表和双链表。其中单链表CurrentProcessList类用以存储当前活动的进程，其包含链表头指针L，而单链表结点由LNode结构体组成，一个LNode中包含结点的数据ProInfo类型的data和指向下一个结点的指针next；双链表EndedProcessList类用以存储已结束的进程，其包含头指针L和尾指针tail，双链表结点由DNode结构体组成，其包含结点数据ProInfo类型的data，指向下一个结点的指针next和指向上一个结点的指针pre。

头文件Process\_ZX.h中的结构主要是表示进程信息的ProcessInfo结构体，其包含进程ID DWORD类型的ID变量，进程名称TCHAR类型的Name，进程占用内存Memory，进程开始时间BeginTime，进程持续时间DuraTime和进程结束时间EndTime。

## 算法思想

在主程序中，先利用头文件Process\_ZX.h中的GetCurrentProcesses函数获取当前系统的活动程序，在调用FreshList函数用当前获取的信息L更新活动进程列表CurList和结束进程列表DelList。

FreshList函数的主要思想为：如果新的进程列表中有进程位于结束进程列表D中，则将其从D中删去；在从原活动进程列表C中找到不在新列表L中的进程，将其放入D中，以达到更新D列表，最后将L覆盖C即可。

## 源代码

**头文件：**

#pragma once

#ifndef PROCESS\_ZX\_H

#define PROCESS\_ZX\_H

/\*头文件包含\*/

#include <windows.h>

#include <stdio.h>

#include <psapi.h>

#include <iostream>

#include <time.h>

using namespace std;

#define  ProcessNum 1024

/\*数据结构定义\*/

typedef struct MSTime

{

    int minute;

    int second;

} MSTime ;

typedef struct Time

{

    int Year;

    int Month;

    int Day;

    int Hour;

    int Minute;

    int Second;

} Time ;

typedef struct ProcessInfo

{

    DWORD  ID ;                              //进程ID,DWORD

    TCHAR  Name[MAX\_PATH];                   //进程名称,TCHAR

    SIZE\_T Memory;                           //进程占用内存,以K为单位

    Time   BeginTime;                        //进程开始时间,Time

    MSTime DuraTime;                         //进程持续时间,int (min)

    Time   EndTime;                          //进程结束时间,Time

} ProInfo ;

//提升进程权限

#pragma comment(lib, "psapi.lib")

bool AdjustPurview()

{

    TOKEN\_PRIVILEGES TokenPrivileges;

    BOOL bRet;

    HANDLE hToken;

    LookupPrivilegeValue(NULL, SE\_DEBUG\_NAME, &TokenPrivileges.Privileges[0].Luid);

    OpenProcessToken(GetCurrentProcess(), TOKEN\_ADJUST\_PRIVILEGES, &hToken);

    TokenPrivileges.PrivilegeCount = 1;

    TokenPrivileges.Privileges[0].Attributes = SE\_PRIVILEGE\_ENABLED;

    bRet = AdjustTokenPrivileges(hToken, FALSE, &TokenPrivileges, 0, NULL, NULL);

    CloseHandle(hToken);

    return bRet == TRUE;

}

//获取信息

bool GetCurrentProcesses(ProInfo pil[] ,  int& pronum , int sizeofpil = 1024 )  //获取进程信息，成功返回true,失败返回false

{//pil 进程信息数组  pronum 返回进程数量 sizeofpil 进程信息的数组元素个数

    HANDLE hProcess ;                       //进程句柄，指向进程

    DWORD ProID;                            //进程ID

    PROCESS\_MEMORY\_COUNTERS pmc ;           //进程信息库（使用其中的内存信息）

    FILETIME begintime, endtime , ipker , ipuser ;  //进程的开始时间，持续时间和结束时间(后面是处于核心、用户状态时间)

    //获取进程列表

    DWORD Processes[1024], cbNeeded , cProcess ;

    if ( !EnumProcesses( Processes , sizeof(Processes), &cbNeeded) )

        return false;

    cProcess = cbNeeded / sizeof(DWORD);

    //按进程列表依次打开进程获取信息

    LPSYSTEMTIME pSystemTime = new SYSTEMTIME;//用来将filetime转换为systemtime，systemtime可以让人看懂

    DWORD btime = 0, etime = 0;               //DWORD形式的开始结束时间，用来运算持续时间

    int i = 0;

    //获取系统时间

    LPSYSTEMTIME systime = new SYSTEMTIME;

    ::GetSystemTime(systime);

    systime->wHour = (systime->wHour + 8) % 24; //转换UTC时间为背景时间

    DWORD stime = systime->wHour \* 60 \* 60 + systime->wMinute \* 60 + systime->wSecond ;//转换为DWORD型,单位S

    //开始循环寻找

    int j = -1;                  //有效数据的记录index

    for ( i = 0; i < cProcess ; i++ )

    {

        //打开进程

        ProID = Processes[i];

        if (ProID == 0)

            continue;

        hProcess = OpenProcess( PROCESS\_QUERY\_INFORMATION|PROCESS\_VM\_READ ,FALSE , ProID ); //打开ID指向的进程，获得查看信息、读取内存的权限

        if ( NULL == hProcess)  //如果未能成功打开进程，返回错误

            continue;

        j++;

        //获取进程ID

        pil[j].ID = ProID;

        //获取进程名称

        GetModuleFileNameEx( hProcess , NULL, pil[j].Name , MAX\_PATH );

        //获取进程内存

        GetProcessMemoryInfo( hProcess , &pmc , sizeof(pmc) );

        pil[j].Memory = pmc.PeakWorkingSetSize / 1024;

        //获取进程时间

        GetProcessTimes( hProcess, &begintime, &endtime, &ipker, &ipuser);//开始、结束

        //开始时间

        FileTimeToSystemTime( &begintime , pSystemTime );

        pSystemTime->wHour = ( pSystemTime->wHour + 8) % 24;  //将UTC时间(时间标准时间)转换为北京时间

        btime = pSystemTime->wHour \* 60 \* 60 + pSystemTime->wMinute \* 60 + pSystemTime->wSecond ; //运算DWORD形式开始时间，单位为S

        pil[j].BeginTime.Year  = pSystemTime->wYear;

        pil[j].BeginTime.Month = pSystemTime->wMonth;

        pil[j].BeginTime.Day = pSystemTime->wDay;

        pil[j].BeginTime.Hour = pSystemTime->wHour;

        pil[j].BeginTime.Minute = pSystemTime->wMinute;

        pil[j].BeginTime.Second = pSystemTime->wSecond;

        //结束时间

        FileTimeToSystemTime( &endtime, pSystemTime );

        pSystemTime->wHour = (pSystemTime->wHour + 8) % 24;  //将UTC时间(时间标准时间)转换为北京时间

        etime = pSystemTime->wHour \* 60 \* 60 + pSystemTime->wMinute \* 60 + pSystemTime->wSecond;//运算DWORD形式结束时间，单位为S

        pil[j].EndTime.Year = pSystemTime->wYear;

        pil[j].EndTime.Month = pSystemTime->wMonth;

        pil[j].EndTime.Day = pSystemTime->wDay;

        pil[j].EndTime.Hour = pSystemTime->wHour;

        pil[j].EndTime.Minute = pSystemTime->wMinute;

        pil[j].EndTime.Second = pSystemTime->wSecond;

        //持续时间

        pil[j].DuraTime.minute = (stime - btime) / 60;

        pil[j].DuraTime.second = (stime - btime) % 60;

    }

    pronum = j + 1;

    return true;

}

//打印信息

void PrintTime(Time T)

{

    cout << T.Year << "年" << T.Month << "月" << T.Day << "日"<< T.Hour << "时" << T.Minute << "分" << T.Second << "秒" <<"\n";

}

void PrintProInfo(ProInfo PI)

{

    if (PI.ID == 0) return;

    cout << "------------------------------------------------" << "\n";

    cout << "进程ID：" << PI.ID << "\n";

    printf("进程名称：%ws", PI.Name);

    cout << "\n";

    cout << "进程内存：" << PI.Memory << "K" << "\n" ;

    cout << "开始时间：";

    PrintTime(PI.BeginTime);

    cout << "持续时间：" << PI.DuraTime.minute << " min "  << PI.DuraTime.second << " s"<< "\n";

    cout << "结束时间：";

    if ( PI.EndTime.Year == 1601 )  //起始时间，即未结束

    {

        cout << "---" << "\n";

    }

    else

    {

        PrintTime(PI.EndTime);

    }

}

//获取系统时间

void GetSystime(Time& time)

{

    LPSYSTEMTIME systime = new SYSTEMTIME;

    ::GetSystemTime(systime);

    systime->wHour = (systime->wHour + 8) % 24; //转换UTC时间为背景时间

    time.Year = systime->wYear;

    time.Month = systime->wMonth;

    time.Day = systime->wDay;

    time.Hour = systime->wHour;

    time.Minute = systime->wMinute;

    time.Second = systime->wSecond;

}

//复制Time

void TimeCPY(Time& mbt, const Time sourcet)

{

    mbt.Year = sourcet.Year;

    mbt.Month = sourcet.Month;

    mbt.Day = sourcet.Day;

    mbt.Hour = sourcet.Hour;

    mbt.Minute = sourcet.Minute;

    mbt.Second = sourcet.Second;

}

//复制ProInfo

void ProInfoCPY(ProInfo& mbp, const ProInfo sourcep)

{

    mbp.ID = sourcep.ID;

    mbp.Memory = sourcep.Memory;

    ua\_tcscpy\_s(mbp.Name, sourcep.Name);

    TimeCPY(mbp.BeginTime, sourcep.BeginTime);

    mbp.DuraTime.minute = sourcep.DuraTime.minute;

    mbp.DuraTime.second = sourcep.DuraTime.second;

    TimeCPY(mbp.EndTime, sourcep.EndTime);

}

//比较MSTime

int MSTimeCMP(MSTime time\_a, MSTime time\_b)//比较time\_a和time\_b,如果a大于b返回1，如果a小于b返回-1;相等返回0

{

    if (time\_a.minute > time\_b.minute)

    {

        return 1;

    }

    else

    {

        if (time\_a.minute < time\_b.minute)

            return -1;

        else

        {

            if (time\_a.second > time\_b.second)

            {

                return 1;

            }

            else

            {

                if (time\_a.second < time\_b.second)

                {

                    return -1;

                }

                else

                {

                    return 0;

                }

            }

        }

    }

}

#endif // !PROCESS\_ZX\_H

**主程序**

//1.系统进程统计

#include <iostream>

using namespace std;

#include "Process\_ZX.h"

ProInfo prolist[1024];

int pronum;

struct LNode //定义一个单链表结点结构体

{

    ProInfo data;

    struct LNode\* next;

};

struct DNode //定义一个双链表结点的结构体

{

    ProInfo data;

    struct DNode\* next, \* pre;

};

class EndedProcessList //抽象一个双链表类

{

    DNode\* L; //头指针

    DNode\* tail;//尾指针

public:

    void InitList()

    {

        L = new DNode;

        //L->data = 0;

        L->next = NULL;

        L->pre = NULL;

        tail = L;

    }

    int ListLength()

    {

        DNode\* p;

        int count = 0;

        if (L)

        {

            p = L->next;

            while (p != NULL)

            {

                count++;

                p = p->next;

            }

        }

        //else

        //cout << "List isn't exists->" << endl;

        return count;

    }

    int LocateElem(ProInfo e)

    {

        int i = 0; //计数

        DNode\* p;

        if (L)

        {

            p = L->next;

            while (p && \_tcscmp(p->data.Name , e.Name)!=0) //当e不是链表中数据继续循环

            {

                ++i;

                p = p->next;

            }

            if (i >= ListLength())

            {

                //cout << "NotFound";

                return 0;

            }

            return i + 1;

        }

        //else

        //cout << "LocateElem failed" << endl;

        return 0;

    }

    void ListTraverse()

    {

        DNode\* p;

        if (L)

        {

            p = L->next;

            while (p) //遍历所有有效的节点

            {

                PrintProInfo(p->data);

                p = p->next;

            }

            cout << endl;

        }

        //else

        //cout << "List isn't exists" << endl;

    }

    void InsertElem( ProInfo e) //插入最后面

    {

        DNode\* s;

            s = new DNode;

            s->data = e; //创建新节点

            s->next = tail->next;

            s->pre = tail;

            tail->next = s;

            tail = tail->next;

    }

    void DeleteElem(int i, ProInfo& e)

    {

        DNode\* p, \* q;

        int j = 1;

        if (L != NULL)

        {

            p = L->next;

            q = L;

            while (j < i)

            {

                q = p;

                p = p->next;

                j++;

            }

            if (p->next)

            {

                p->next->pre = q;

                q->next = p->next;

            }

            else

                q->next = NULL;

            e = p->data;

            delete[] p;

        }

    }

    void Sort()

    {

        DNode\* t, \* p, \* q = L;

        p = L->next;

        L->next = NULL;

        while (p)

        {

            q = L;

            t = p->next;

            while (q->next && MSTimeCMP(p->data.DuraTime, q->next->data.DuraTime) == 1)

                q = q->next;

            p->next = q->next;

            p->pre = q;

            if (q->next)

                q->next->pre = p;

            q->next = p;

            p = t;

        }

    }

};

class CurrentProcessList //抽象一个单链表类

{

    LNode\* L; //头指针

public:

    void InitList()

    {

        L = new LNode;

        //L->data = 0;

        L->next = NULL;

    }

    void ClearList()

    {

        LNode\* p, \* q;

        p = L->next;

        while (p) //逐个删除节点

        {

            q = p->next;

            delete[] p;

            p = q;

        }

        L->next = NULL; //头指针指向空，链表清空

    }

    int ListLength()

    {

        LNode\* p;

        int count = 0;

        if (L)

        {

            p = L->next;

            while (p != NULL)

            {

                count++;

                p = p->next;

            }

        }

        else

            cout << "List isn't exists->" << endl;

        return count;

    }

    void GetElem(int i, ProInfo& e)

    {

        LNode\* p;

        int j = 1;

        if (L && (i > 0 && i <= ListLength())) //判断L存在且i符合定义域

        {

            p = L->next;

            while (j < i)

            {

                p = p->next;

                j++;

            }

            e = p->data;

        }

        else

            cout << "GetElem failed!" << endl;

    }

    int LocateElem(ProInfo e)

    {

        int i = 0; //计数

        LNode\* p;

        if (L)

        {

            p = L->next;

            while (p && (p->data.ID != e.ID)) //当e不是链表中数据继续循环

            {

                ++i;

                p = p->next;

            }

            if (i >= ListLength())

            {

                //cout << "NotFound";

                return 0;

            }

            return i + 1;

        }

        else

            //cout << "LocateElem failed" << endl;

            return 0;

    }

    void ListTraverse()

    {

        LNode\* p;

        if (L)

        {

            p = L->next;

            while (p) //遍历所有有效的节点

            {

                PrintProInfo(p->data);

                p = p->next;

            }

            cout << endl;

        }

        else

            cout << "List isn't exists" << endl;

    }

    void InsertElem(int i, ProInfo e)

    {

        LNode\* s;

        LNode\* p;

        int j = 1;

        if (L && (i >= 0 && i <= ListLength()))

        {

            s = new LNode;

            s->data = e; //创建新节点

            p = L;

            while (j < i) //定位i

            {

                p = p->next;

                j++;

            }

            s->next = p->next;

            p->next = s;

        }

    }

    void DeleteElem(int i, ProInfo& e)

    {

        LNode\* p, \* q;

        int j = 1;

        if (L != NULL)

        {

            p = L->next;

            q = L;

            while (j < i)

            {

                q = p;

                p = p->next;

                j++;

            }

            q->next = p->next;

            e = p->data;

            delete[] p;

        }

    }

    void Sort()

    {

        LNode\* t, \* p, \* q = L;

        p = L->next;

        L->next = NULL;

        while (p)

        {

            q = L;

            t = p->next;

            while (q->next && p->data.Memory < q->next->data.Memory)

                q = q->next;

            p->next = q->next;

            q->next = p;

            p = t;

        }

    }

};

void FreshList(ProInfo L[], int n, CurrentProcessList C, EndedProcessList D)

{

    int temp;

    ProInfo T;

    for (int i = 0; i < n; i++)

        if (temp = D.LocateElem(L[i]))       //如果新的进程列表里有进程在D中

            D.DeleteElem(temp, T);           //则将其从D中删去

    for (int i = 0; i < C.ListLength(); i++) //找已结束的进程放进D里

    {

        C.GetElem(i + 1, T);

        for (int j = 0; j < n; j++)

        {

            if (T.ID == L[j].ID)

                break;

            else if (j == n - 1)

            {

                D.InsertElem(T); //将结束的进程放进D

            }

        }

    }

    C.ClearList(); //清空C

    for (int i = 0; i < n; i++)

        C.InsertElem(0, L[i]); //将新的进程列表插入C中

    C.Sort(); //对C进行排序

    D.Sort();

}

int main()

{

    CurrentProcessList CurList; //当前进程列表

    EndedProcessList DelList; //已结束进程列表

    //初始化

    AdjustPurview();

    CurList.InitList();

    DelList.InitList();

    GetCurrentProcesses(prolist, pronum, 1024);

    for (int i = 0; i < pronum; i++)

        CurList.InsertElem(0, prolist[i]); //将新的进程列表插入C中

    while (1)

    {

        //执行

        GetCurrentProcesses(prolist, pronum, 1024);

        FreshList(prolist, pronum, CurList, DelList);

        system("cls");

        cout << "----------------当前活动的进程----------------" << endl;

        CurList.ListTraverse();

        cout << endl << "----------------已结束的进程------------------" << endl;

        DelList.ListTraverse();

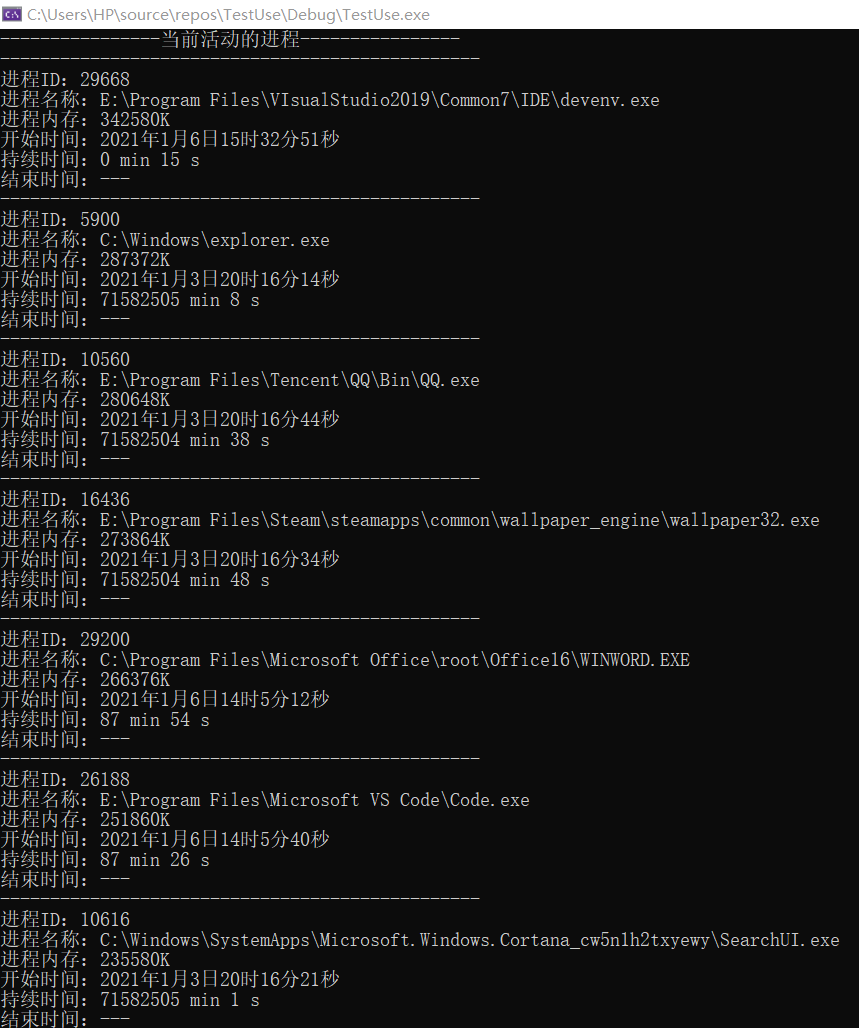
        //system("pause");

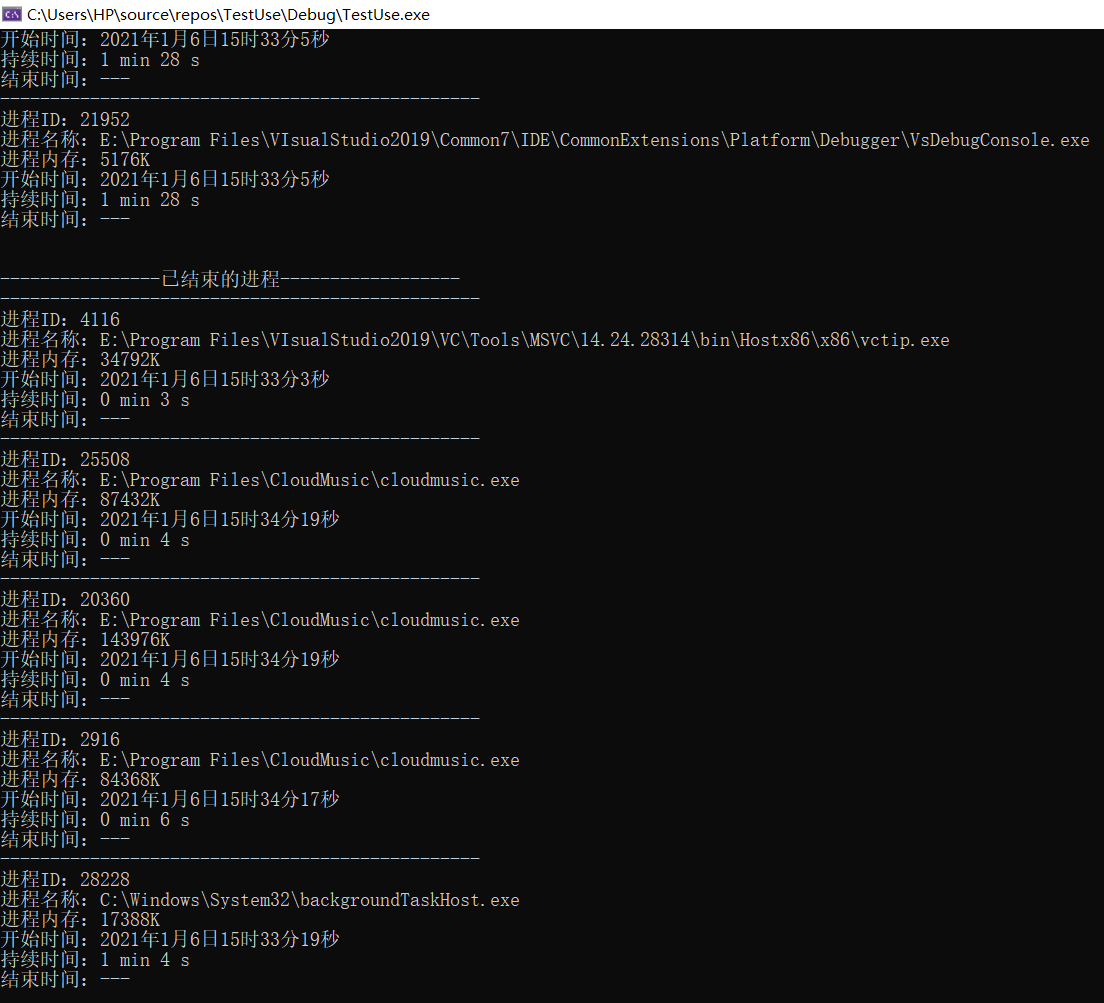
Sleep(1000);

    }

}

## 测试数据和结果





## 小结

链表操作中，排序操作时间复杂度为O（n²），其它均为O（n）。刷新函数FreshList的时间复杂度为O（n²）。

优化方面可以在双链表的排序上，可以选用更快的排序算法。

# 迷宫问题

## 数据结构

地图存储采用二维数组map[N][N]记录，1为不可走，0为可走。求解过程运用栈结构来实现非递归深搜，栈元素类型为Point结构体，其包含某个点的坐标i与j。栈类中包含栈底指针pBase和栈顶指针pTop以及栈分配空间stacksize。

## 算法思想

核心算法FindWay函数来寻找位置a到位置b的一条迷宫通路。其主要思想是利用栈操作实现深度优先搜索。具体步骤：先将起点位置a入栈并标记其位置已走过，然后开始当栈不为空时循环：获取栈顶元素cur为当前位置，尝试走cur四个方向，如果可走且没走过，则将其入栈并标记走过，如果四个方向位置都不可走，则判断是否为终点，是的话输出路线，否则将栈顶元素出栈。

## 3. 源代码

//2.迷宫问题

#include <iostream>

#include <iomanip>

#include <fstream>

#define N 100

#define STACKINITSIZE 512

#define STACKINCREMENT 128

using namespace std;

struct Point

{

    int x;

    int y;

    //bool flag;

};

int n, m;

class Stack

{

    Point \*pBase;

    Point \*pTop;

    int stacksize; //栈已分配的空间

public:

    void InitStack()

    {

        pBase = new Point[STACKINITSIZE];

        pTop = pBase;

        stacksize = STACKINITSIZE;

    }

    bool IsEmpty()

    {

        if (pBase)

        {

            if (pBase == pTop)

                return true;

            else

                return false;

        }

        return true;

    }

    void GetTop(Point &e)

    {

        if (pBase && (pBase != pTop))

        {

            e.x = (pTop - 1)->x;

            e.y = (pTop - 1)->y;

        }

    }

    void Pop()

    {

        if (pBase && pBase != pTop)

            pTop--;

    }

    void Push(int a, int b)

    {

        Point \*temp;

        if (pBase)

        {

            if (pTop - pBase >= stacksize) //栈满了动态增加空间

            {

                temp = new Point[stacksize + STACKINCREMENT];

                for (int j = 0; (pBase + j) < pTop; j++)

                    \*(temp + j) = \*(pBase + j);

                delete[] pBase;

                pBase = temp;

                pTop = pBase + stacksize;

                stacksize += STACKINCREMENT;

            }

            pTop->x = a;

            pTop->y = b;

            pTop++;

        }

    }

    void Print(int M[N][N])

    {

        int i = 0;

        char map[N][N];

        for (int j = 0; j < n; j++)

            for (int k = 0; k < m; k++)

            {

                if (M[j][k])

                    map[j][k] = '#';

                else

                    map[j][k] = ' ';

            }

        if (pBase)

            while ((pBase + i) < pTop)

            {

                map[(pBase + i)->x][(pBase + i)->y] = '@';

                i++;

            }

        cout << endl

             << "-------------------------------------------------" << endl;

        for (int j = 0; j < n; j++)

        {

            for (int k = 0; k < m; k++)

                cout << map[j][k] << ' ';

            cout << endl;

        }

    }

};

void FindWay(Point a, int M[N][N], Point b)

{

    Stack way;

    Point cur;

    int tx, ty; //下一个点坐标

    int next[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    int flag[N][N] = {0}; //标记已走过的点

    bool temp;            //标记某次有路可走

    way.InitStack();

    way.Push(a.x, a.y);

    flag[a.x][a.y] = 1;

    while (!way.IsEmpty()) //栈不为空继续循环

    {

        way.GetTop(cur); //获取栈顶元素

        temp = false;

        for (int i = 0; i < 4; i++) //尝试走四个方向

        {

            tx = cur.x + next[i][0];

            ty = cur.y + next[i][1];

            if (tx < 0 || tx >= n || ty < 0 || ty >= m || M[tx][ty] == 1) //判断不可走的点

                continue;

            if (flag[tx][ty] == 0) //如果没走过

            {

                flag[tx][ty] = 1; //标记该点走过

                way.Push(tx, ty); //入栈

                temp = true;      //标记有路

                break;

            }

        }

        if (temp == false) //没路走了，出栈

        {

            if ((cur.x == b.x && cur.y == b.y)) //如果是终点，输出(cur.x != a.x && cur.y != a.y) &&

            {

                way.Print(M);

                return;

            }

            way.Pop();

        }

    }

    cout << "未找到通路" << endl;

}

int main()

{

    ifstream fin("Maze.txt");

    int map[N][N];

    Point pbegin, pend;

    fin >> n >> m;

    for (int i = 0; i < n; i++)

    {

        for (int j = 0; j < m; j++)

            fin >> map[i][j];

    }

    for (int i = -1; i < n; i++)

    {

        if (i > -1)

            cout << setw(2) << i;

        for (int j = -1; j < m; j++)

        {

            if (i == -1)

                cout << setw(2) << j;

            else if (map[i][j] == 1)

                cout << "# ";

            else

                cout << "  ";

        }

        cout << endl;

    }

    cout << "输入起点位置(空格为间隔)：";

    cin >> pbegin.x >> pbegin.y;

    cout << "输入终点位置(空格为间隔)：";

    cin >> pend.x >> pend.y;

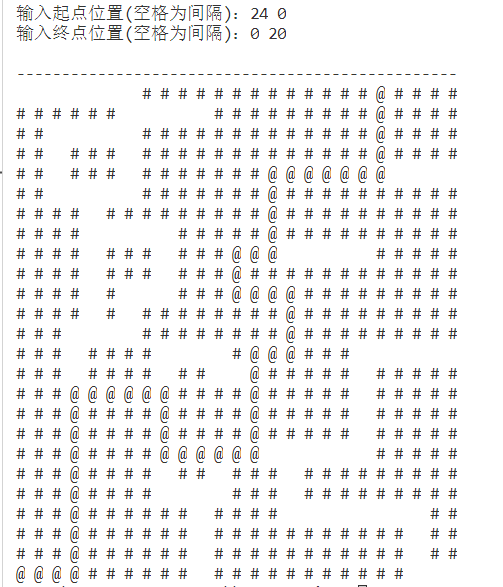
    FindWay(pbegin, map, pend);

}

## 4. 测试数据和结果



迷宫文件采用群文件里的，运行时命名为Maze.txt



## 5. 小结

非递归的深度优先搜索时间复杂度为O（n²）。

# 家谱管理系统

## 1. 数据结构

存储家谱的数据结构为多叉树。建立家谱树类FamilyTree，其包含数根结点指针root以及家谱的各类操作。树结点BiTNode结构体包含成员数据Member类型的data，结点代数generation，父亲指针parent，孩子数childnum和孩子指针数组child。Member结构体中包含字符串型姓名name和地址address，date型生日birth和忌日death，布尔型婚否merry和健在否alive。

家谱文件数据存储时，一个人信息在一行里，而信息前的制表符个数表示父子关系

## 2. 算法思想

利用储存时的制表符以确定父子关系，以此往树中添加成员。基于以上的数据结构设置，所有的查询类功能以及输出导出功能均可通过深度优先搜索以递归的方法实现。添加孩子、修改信息可利用定位函数定位到后直接添加或修改，删除成员也是定位到后调用释放函数Relase通过类似后序遍历来删除。确定两人关系，因为是双向树，所以直接匹配两者长辈即可。

## 3. 源代码

//3.家谱管理系统

#pragma warning(disable : 4996)

#include <iostream>

#include <stdlib.h>

#include <fstream>

#include <string>

#include <cstring>

#define CHILD 5 //最大孩子数

using namespace std;

ofstream out; //导出文件

struct date   //日期结构

{

    int year;  //年

    int mouth; //月

    int day;   //日

};

struct Member //成员数据结构

{

    char \*name;    //姓名

    date birth;    //生日

    bool merry;    //婚否

    char \*address; //地址

    bool alive;    //健在否

    date death;    //忌日

};

typedef struct BiTNode //树结点结构

{

    Member data;                  //成员数据

    int generation;               //代数

    BiTNode \*parent;              //父亲指针

    int childnum;                 //孩子数

    struct BiTNode \*child[CHILD]; //孩子指针

} BiTNode, \*BiTree;

void InitBiNode(BiTNode \*a) //初始化成员结点

{

    for (int i = 0; i < CHILD; i++)

        a->child[i] = NULL;

    a->data.name = new char[20];

   a->data.address = new char[30];

}

class FamilyTree //家谱树类

{

    BiTree root; //树根结点指针

public:

    void Creat()

    {

        BiTNode \*q;

        int y, m, d, temp;

        string s;

        q = new BiTNode[1];

        InitBiNode(q);

        root = q; //设置根结点

        q->parent = NULL;

        q->generation = 1;

        q->childnum = 0;

        cout << "创建第一个成员：" << endl;

        cout << "输入姓名：";

        cin >> s;

        strcpy(q->data.name, s.c\_str());

        cout << "是否健在（0为否，1为是）：";

        while (1)

        {

            cin >> temp;

            if (temp == 0)

                q->data.alive = false;

            else if (temp == 1)

                q->data.alive = true;

            else

            {

                cout << "输入有误，重新输入：";

                continue;

            }

            break;

        }

        cout << "输入出生年月日（以空格为分隔）：";

        while (1)

        {

            cin >> y >> m >> d;

            if (m < 0 || m > 12 || d < 0 || d > 31)

            {

                cout << "输入有误，重新输入：";

                continue;

            }

            else

            {

                q->data.birth.year = y;

                q->data.birth.mouth = m;

                q->data.birth.day = d;

            }

            break;

        }

        if (q->data.alive == false)

        {

            cout << "输入去世年月日（以空格为分隔）：";

            while (1)

            {

                cin >> y >> m >> d;

                if (m < 0 || m > 12 || d < 0 || d > 31 || y < q->data.birth.year)

                {

                    cout << "输入有误，重新输入：";

                    continue;

                }

                else

                {

                    q->data.birth.year = y;

                    q->data.birth.mouth = m;

                    q->data.birth.day = d;

                }

                break;

            }

        }

        cout << "是否已婚（0为否，1为是）：";

        while (1)

        {

            cin >> temp;

            if (temp == 0)

                q->data.merry = false;

            else if (temp == 1)

                q->data.merry = true;

            else

            {

                cout << "输入有误，重新输入：";

                continue;

            }

            break;

        }

        cout << "输入地址：";

        cin >> s;

        strcpy(q->data.address, s.c\_str());

    }

    void Import(char \*file) //导入家谱

    {

        //数据格式：姓名,已故,YYYY.MM.DD-YYYY.MM.DD,已婚,地址

        BiTNode \*father[20]; //记录当前各代父亲

        BiTNode \*p;          //新结点

        int i, k, num;       //计数用

        int rad;             //日期转换基数

        int ymd;             //年月日记录,0年1月2日

        char \*a;             //文件一行

        char \*temp;          //临时字符串

        char \*c;             //存储中文

        ifstream fin(file);

        c = new char[4];

        a = new char[200];

        temp = new char[30];

        while (!fin.eof())

        {

            fin.getline(a, 200);

            p = new BiTNode[1];

            InitBiNode(p);

            i = 0;

            while (a[i] == '\t') //吃掉制表符

                i++;

            if (i == 0) //如果是第一代

            {

                //root = new BiTNode;

                root = p; //设置根结点

                p->parent = NULL;

                father[0] = p;

                p->generation = 1;

                p->childnum = 0;

            }

            else //否则不是第一代

            {

                father[i] = p;

                p->parent = father[i - 1];                           //确定父亲

                father[i - 1]->child[father[i - 1]->childnum++] = p; //确定孩子

                p->generation = p->parent->generation + 1;           //确定代数

                p->childnum = 0;

            }

            //姓名

            strcpy(temp, "");

            while (a[i] != ',')

            {

                c[0] = a[i];

                c[1] = a[i + 1];

                c[2] = '\0';

                strcat(temp, c); //读入一个中文汉字

                i += 2;

            }

            p->data.name = new char[sizeof(char) \* (strlen(temp) + 1)];

            strcpy(p->data.name, temp); //读入姓名

            i++;                        //下一个字符

            //健在否

            strcpy(temp, "");

            while (a[i] != ',')

            {

                c[0] = a[i];

                c[1] = a[i + 1];

                //c[2] = a[i + 2];

                c[2] = '\0';

                strcat(temp, c); //读入一个汉字

                i += 2;

            }

            if (strcmp(temp, "已故") == 0)

                p->data.alive = false;

            else

                p->data.alive = true;

            i++; //下一个字符

            //出生日期

            k = 0;

            ymd = 0;

            strcpy(temp, "");

            while (a[i] != '-')

            {

                if (a[i] == '.')

                {

                    rad = 1; //基数

                    num = 0;

                    for (int j = strlen(temp) - 1; j >= 0; j--)

                    {

                        num = num + rad \* (temp[j] - '0');

                        rad \*= 10;

                    }

                    if (ymd == 0)

                        p->data.birth.year = num; //出生年

                    if (ymd == 1)

                        p->data.birth.mouth = num; //出生月

                    strcpy(temp, "");

                    k = 0;

                    ymd++;

                    i++;

                }

                temp[k++] = a[i]; //读入一个数字

                temp[k] = '\0';

                i++;

            }

            rad = 1; //基数

            num = 0;

            for (int j = strlen(temp) - 1; j >= 0; j--)

            {

                num = num + rad \* (temp[j] - '0');

                rad \*= 10;

            }

            p->data.birth.day = num; //出生日

            i++;                     //下一个字符

            //死亡日期

            if (p->data.alive == false) //如果已故

            {

                k = 0;

                ymd = 0;

                strcpy(temp, "");

                while (a[i] != ',')

                {

                    if (a[i] == '.')

                    {

                        rad = 1; //基数

                        num = 0;

                        for (int j = strlen(temp) - 1; j >= 0; j--)

                        {

                            num = num + rad \* (temp[j] - '0');

                            rad \*= 10;

                        }

                        if (ymd == 0)

                            p->data.death.year = num; //死亡年

                        if (ymd == 1)

                            p->data.death.mouth = num; //死亡月

                        strcpy(temp, "");

                        k = 0;

                        ymd++;

                        i++;

                    }

                    temp[k++] = a[i]; //读入一个数字

                    temp[k] = '\0';

                    i++;

                }

                rad = 1; //基数

                num = 0;

                for (int j = strlen(temp) - 1; j >= 0; j--)

                {

                    num = num + rad \* (temp[j] - '0');

                    rad \*= 10;

                }

                p->data.death.day = num; //死亡日

            }

            i++;

            //婚否

            strcpy(temp, "");

            while (a[i] != ',')

            {

                c[0] = a[i];

                c[1] = a[i + 1];

                c[2] = '\0';

                strcat(temp, c); //读入一个汉字

                i += 2;

            }

            if (strcmp(temp, "未婚") == 0)

                p->data.merry = false;

            else

                p->data.merry = true;

            i++; //下一个字符

            //地址

            strcpy(temp, "");

            while (i < strlen(a))

            {

                c[0] = a[i];

                c[1] = a[i + 1];

                c[2] = '\0';

                strcat(temp, c); //读入一个中文汉字

                i += 2;

            }

            p->data.address = new char[sizeof(char) \* (strlen(temp) + 1)];

            strcpy(p->data.address, temp); //读入地址

            //for (int i = 0; i < p->generation - 1; i++)

            //    cout << '\t';

            //printf("%s,%d,%d.%d.%d-\n", p->data.name, p->data.alive, p->data.birth.year, p->data.birth.mouth, p->data.birth.day);

        }

        //读入结束

        fin.close();

    }

    void Export(BiTNode \*p) //导出家谱

    {

        int len;

        if (p == NULL)

            return;

        for (int i = 0; i < p->generation - 1; i++)

            out << '\t';

        out << p->data.name << ',';

        if (p->data.alive)

            out << "健在," << p->data.birth.year << '.' << p->data.birth.mouth << '.' << p->data.birth.day << '-';

        else

            out << "已故," << p->data.birth.year << '.' << p->data.birth.mouth << '.' << p->data.birth.day << '-' << p->data.death.year << '.' << p->data.death.mouth << '.' << p->data.death.day;

        if (p->data.merry)

            out << ",已婚," << p->data.address;

        else

            out << ",未婚," << p->data.address;

        out << endl;

        for (int i = 0; i < p->childnum; i++)

            Export(p->child[i]);

    }

    void PrintInfo(BiTNode \*p)

    {

        printf("%s\t", p->data.name);

        if (p->data.alive)

            printf("健在\t%d.%d.%d-\t\t", p->data.birth.year, p->data.birth.mouth, p->data.birth.day);

        else

            printf("已故\t%d.%d.%d-%d.%d.%d\t", p->data.birth.year, p->data.birth.mouth, p->data.birth.day, p->data.death.year, p->data.death.mouth, p->data.death.day);

        if (p->data.merry)

            printf("已婚\t%s\n", p->data.address);

        else

            printf("未婚\t%s\n", p->data.address);

    }

    void Print(BiTNode \*p) //显示家谱

    {

        if (p == NULL)

            return;

        for (int i = 0; i < p->generation - 1; i++)

            cout << '\t';

        printf("%s\n", p->data.name);

        for (int i = 0; i < p->childnum; i++)

            Print(p->child[i]);

    }

    BiTNode \*GetRoot()

    {

        return root;

    }

    void DisplayNGenera(BiTNode \*p, int n) //显示第n代所有成员

    {

        if (p == NULL)

            return;

        if (p->generation == n)

            PrintInfo(p);

        for (int i = 0; i < p->childnum; i++)

            DisplayNGenera(p->child[i], n);

    }

    void SearchByName(char \*a) //按照姓名查询

    {

        BiTNode \*p;

        p = Locate(root, a);

        if (p == NULL)

        {

            cout << "未找到该成员。" << endl;

            return;

        }

        cout << "本人：" << endl;

        PrintInfo(p);

        if (p->parent)

        {

            cout << "父亲：" << endl;

            PrintInfo(p->parent);

        }

        for (int i = 0; i < p->childnum; i++)

        {

            cout << "第" << i + 1 << "个孩子" << endl;

            PrintInfo(p->child[i]);

        }

    }

    void SearchByBirth(BiTNode \*p, int y, int m) //按照出生年月查询成员

    {

        if (p == NULL)

            return;

        if (p->data.birth.year == y && p->data.birth.mouth == m)

            PrintInfo(p);

        for (int i = 0; i < p->childnum; i++)

            SearchByBirth(p->child[i], y, m);

    }

    void SearchByBirth(BiTNode \*p, int y) //按照出生年份查询成员

    {

        if (p == NULL)

            return;

        if (p->data.birth.year == y)

            PrintInfo(p);

        for (int i = 0; i < p->childnum; i++)

            SearchByBirth(p->child[i], y);

    }

    void Relation(BiTNode \*a, BiTNode \*b) //确定a、b关系

    {

        //a是b的XXX。

        if (a->parent) //如果a不是根结点，即a有父亲,且a辈分小于等于b时

        {

            if (a->parent == b->parent)

            {

                printf("%s是%s的兄弟\n", a->data.name, b->data.name);

                return;

            }

            if (a->parent == b)

            {

                printf("%s是%s的儿子\n", a->data.name, b->data.name);

                return;

            }

            if (a->parent->parent == b)

            {

                printf("%s是%s的孙子\n", a->data.name, b->data.name);

                return;

            }

            if (a->parent->parent && a->parent->parent->parent == b)

            {

                printf("%s是%s的曾孙子\n", a->data.name, b->data.name);

                return;

            }

        }

        if (b->parent) //如果b不是根结点，即b有父亲,且a辈分大于b时

        {

            if (b->parent == a)

            {

                printf("%s是%s的父亲\n", a->data.name, b->data.name);

                return;

            }

            if (b->parent->parent == a)

            {

                printf("%s是%s的祖父\n", a->data.name, b->data.name);

                return;

            }

            if (b->parent->parent && b->parent->parent->parent == a)

            {

                printf("%s是%s的曾祖父\n", a->data.name, b->data.name);

                return;

            }

        }

        printf("%s与%s为非直系关系或关系超出四代\n", a->data.name, b->data.name);

    }

    BiTNode \*Locate(BiTNode \*p, char \*a) //根据姓名定位结点地址

    {

        BiTNode \*q = NULL;

        if (p == NULL)

            return NULL;

        if (strcmp(p->data.name, a) == 0)

            return p;

        for (int i = 0; i < p->childnum; i++)

        {

            q = Locate(p->child[i], a);

            if (q != NULL)

                return q;

        }

        return NULL;

    }

    int AddChild(char \*a) //为a添加孩子

    {

        BiTNode \*p, \*q;

        string s;

        int temp, y, m, d;

        p = Locate(root, a);

        if (p == NULL)

        {

            cout << "未找到该人。" << endl;

            return 0;

        }

        q = new BiTNode;

        InitBiNode(q);

        q->parent = p;

        q->generation = p->generation + 1;

        q->childnum = 0; //初始化新新结点q

        p->child[p->childnum] = q;

        p->childnum++; //更新p结点

        cout << "输入姓名：";

        cin >> s;

        strcpy(q->data.name, s.c\_str());

        cout << "是否健在（0为否，1为是）：";

        while (1)

        {

            cin >> temp;

            if (temp == 0)

                q->data.alive = false;

            else if (temp == 1)

                q->data.alive = true;

            else

            {

                cout << "输入有误，重新输入：";

                continue;

            }

            break;

        }

        cout << "输入出生年月日（以空格为分隔）：";

        while (1)

        {

            cin >> y >> m >> d;

            if (m < 0 || m > 12 || d < 0 || d > 31)

            {

                cout << "输入有误，重新输入：";

                continue;

            }

            else

            {

                q->data.birth.year = y;

                q->data.birth.mouth = m;

                q->data.birth.day = d;

            }

            break;

        }

        if (q->data.alive == false)

        {

            cout << "输入去世年月日（以空格为分隔）：";

            while (1)

            {

                cin >> y >> m >> d;

                if (m < 0 || m > 12 || d < 0 || d > 31 || y < q->data.birth.year)

                {

                    cout << "输入有误，重新输入：";

                    continue;

                }

                else

                {

                    q->data.birth.year = y;

                    q->data.birth.mouth = m;

                    q->data.birth.day = d;

                }

                break;

            }

        }

        cout << "是否已婚（0为否，1为是）：";

        while (1)

        {

            cin >> temp;

            if (temp == 0)

                q->data.merry = false;

            else if (temp == 1)

                q->data.merry = true;

            else

            {

                cout << "输入有误，重新输入：";

                continue;

            }

            break;

        }

        cout << "输入地址：";

        cin >> s;

        strcpy(q->data.address, s.c\_str());

        return 1;

    }

    void Delete(char \*a) //删除成员

    {

        BiTNode \*p;

        int i;

        p = Locate(root, a);

        if (p == NULL)

        {

            cout << "未找到该成员。" << endl;

            return;

        }

        if (p->parent)

        {

            for (i = 0; i < p->parent->childnum; i++)

                if (p->parent->child[i] == p)

                    break;

            for (i; i < p->parent->childnum - 1; i++)

                p->parent->child[i] = p->parent->child[i + 1]; //删掉p父结点指向p的孩子指针

            p->parent->childnum--;

        }

        Relase(p);

    }

    int Modify(char \*a) //修改信息

    {

        BiTNode \*q;

        int choose;

        string s;

        int temp, y, m, d;

        q = Locate(root, a);

        while (1)

        {

            system("cls");

            cout << "当前信息：";

            PrintInfo(q);

            cout << "\n1.姓名\n2.健在否\n3.出生日期\n4.去世日期\n5.婚否\n6.地址\n0.退出\n输入要修改的选项：";

            cin >> choose;

            switch (choose)

            {

            case 1:

                cout << "输入姓名：";

                cin >> s;

                strcpy(q->data.name, s.c\_str());

                system("pause");

                break;

            case 2:

                cout << "是否健在（0为否，1为是）：";

                while (1)

                {

                    cin >> temp;

                    if (temp == 0)

                        q->data.alive = false;

                    else if (temp == 1)

                        q->data.alive = true;

                    else

                    {

                        cout << "输入有误，重新输入：";

                        continue;

                    }

                    break;

                }

                system("pause");

                break;

            case 3:

                cout << "输入出生年月日（以空格为分隔）：";

                while (1)

                {

                    cin >> y >> m >> d;

                    if (m < 0 || m > 12 || d < 0 || d > 31)

                    {

                        cout << "输入有误，重新输入：";

                        continue;

                    }

                    else

                    {

                        q->data.birth.year = y;

                        q->data.birth.mouth = m;

                        q->data.birth.day = d;

                    }

                    break;

                }

                system("pause");

                break;

            case 4:

                if (q->data.alive == false)

                {

                    cout << "输入去世年月日（以空格为分隔）：";

                    while (1)

                    {

                        cin >> y >> m >> d;

                        if (m < 0 || m > 12 || d < 0 || d > 31 || y < q->data.birth.year)

                        {

                            cout << "输入有误，重新输入：";

                            continue;

                        }

                        else

                        {

                            q->data.birth.year = y;

                            q->data.birth.mouth = m;

                            q->data.birth.day = d;

                        }

                        break;

                    }

                }

                else

                {

                    cout << "无法设置该选项。" << endl;

                }

                system("pause");

                break;

            case 5:

                cout << "是否已婚（0为否，1为是）：";

                while (1)

                {

                    cin >> temp;

                    if (temp == 0)

                        q->data.merry = false;

                    else if (temp == 1)

                        q->data.merry = true;

                    else

                    {

                        cout << "输入有误，重新输入：";

                        continue;

                    }

                    break;

                }

                system("pause");

                break;

            case 6:

                cout << "输入地址：";

                cin >> s;

                strcpy(q->data.address, s.c\_str());

                system("pause");

                break;

            case 0:

                break;

            }

            if (choose == 0)

                break;

        }

        return 1;

    }

    void Relase(BiTNode \*p)

    {

        if (p == NULL)

            return;

        for (int i = 0; i < p->childnum; i++)

            Relase(p->child[i]);

        delete p;

    }

    void SetEmpty()

    {

        root = NULL;

    }

};

int menu\_sel() //测试菜单

{

    char \*m[15] = {(char \*)"1.创建新家谱",

                   (char \*)"2.导入家谱",

                   (char \*)"3.导出家谱",

                   (char \*)"4.显示家谱",

                   (char \*)"5.显示第N代成员",

                   (char \*)"6.按姓名查询",

                   (char \*)"7.按出生年/年月查询成员",

                   (char \*)"8.确定成员与成员的关系",

                   (char \*)"9.为成员添加孩子",

                   (char \*)"10.删除成员",

                   (char \*)"11.修改成员信息",

                   (char \*)"12.删除家谱",

                   (char \*)"0.退出"};

    int i, choice;

    do

    {

        system("cls");

        cout << "-------家谱管理系统主菜单-------" << endl;

        for (i = 0; i < 13; i++)

            cout << m[i] << endl;

        cout << "请输入选择：";

        cin >> choice;

    } while (choice < 0 || choice > 12);

    return choice;

}

int main()

{

    FamilyTree T;

    string s;

    int sel, choose, y, m;

    BiTNode \*a, \*b;

    T.Import((char \*)"FamilyTree.txt"); //导入初始家谱文件

    while (1)

    {

        if (T.GetRoot() == NULL)

        {

            cout << "当前家谱未创建，请导入或创建！\n";

            system("pause");

        }

        switch (sel = menu\_sel())

        {

        case 1:

            system("cls");

            cout << "---------创建新家谱---------\n";

            T.Relase(T.GetRoot());

            cout << "初始化完毕\n";

            T.Creat();

            cout << "创建结束\n";

            system("pause");

            break;

        case 2:

            system("cls");

            cout << "---------导入家谱---------\n";

            cout << "输入导入文件名：";

            cin >> s;

            T.Import((char \*)s.c\_str());

            cout << "导入结束" << endl;

            system("pause");

            break;

        case 3:

            system("cls");

            cout << "---------导出家谱---------\n";

            cout << "输入导出文件名：";

            cin >> s;

            out.open(s.c\_str());

            T.Export(T.GetRoot());

            cout << "导出结束" << endl;

            system("pause");

            break;

        case 4:

            system("cls");

            cout << "---------显示家谱---------\n";

            T.Print(T.GetRoot());

            system("pause");

            break;

        case 5:

            system("cls");

            cout << "---------显示第N代成员---------\n";

            cout << "输入你想输出的N：";

            cin >> choose;

            cout << "第" << choose << "代成员有：\n";

            T.DisplayNGenera(T.GetRoot(), choose);

            cout << "以上\n";

            system("pause");

            break;

        case 6:

            system("cls");

            cout << "---------按姓名查询---------\n";

            cout << "输入姓名：";

            cin >> s;

            T.SearchByName((char \*)s.c\_str());

            system("pause");

            break;

        case 7:

            system("cls");

            cout << "---------按出生年/年月查询---------\n";

            cout << "输入年份：";

            cin >> y;

            cout << "输入月份(模糊请输入0)：";

            while (1)

            {

                cin >> m;

                if (m < 0 || m > 12)

                    cout << "输入有误,请重输：" << endl;

                else

                    break;

            }

            if (m == 0)

                T.SearchByBirth(T.GetRoot(), y);

            else

                T.SearchByBirth(T.GetRoot(), y, m);

            cout << "以上" << endl;

            system("pause");

            break;

        case 8:

            system("cls");

            cout << "--------确定成员与成员的关系----------\n";

            cout << "输入成员1姓名：";

            while (1)

            {

                cin >> s;

                a = T.Locate(T.GetRoot(), (char \*)s.c\_str());

                if (a == NULL)

                    cout << "输入有误，请重输：";

                else

                    break;

            }

            cout << "输入成员2姓名：";

            while (1)

            {

                cin >> s;

                b = T.Locate(T.GetRoot(), (char \*)s.c\_str());

                if (b == NULL || b == a)

                    cout << "输入有误，请重输：";

                else

                    break;

            }

            T.Relation(a, b);

            system("pause");

            break;

        case 9:

            system("cls");

            cout << "---------为成员添加孩子---------\n";

            cout << "输入该成员姓名：";

            cin >> s;

            T.AddChild((char \*)s.c\_str());

            system("pause");

            break;

        case 10:

            system("cls");

            cout << "---------删除成员---------\n";

            cout << "输入成员姓名：";

            while (1)

            {

                cin >> s;

                a = T.Locate(T.GetRoot(), (char \*)s.c\_str());

                if (a == NULL)

                    cout << "输入有误，请重输：";

                else

                    break;

            }

            cout << "你确定要删除" << a->data.name << "及其所有后代?\n1确定，0取消：";

            cin >> choose;

            if (choose == 1)

            {

                T.Delete((char \*)s.c\_str());

                cout << "删除完成\n";

            }

            system("pause");

            break;

        case 11:

            system("cls");

            cout << "---------修改信息---------\n";

            cout << "输入成员姓名：";

            while (1)

            {

                cin >> s;

                a = T.Locate(T.GetRoot(), (char \*)s.c\_str());

                if (a == NULL)

                    cout << "输入有误，请重输：";

                else

                    break;

            }

            T.Modify((char \*)s.c\_str());

            system("pause");

            break;

        case 12:

            system("cls");

            cout << "-----------------删除家谱-------------------\n";

            cout << "你确定要删除该家谱?\n1确定，0取消：";

            cin >> choose;

            if (choose == 1)

            {

                T.Relase(T.GetRoot());

                T.SetEmpty();

                cout << "删除完成\n";

            }

            system("pause");

            break;

        case 0:

            break;

        }

        if (sel == 0)

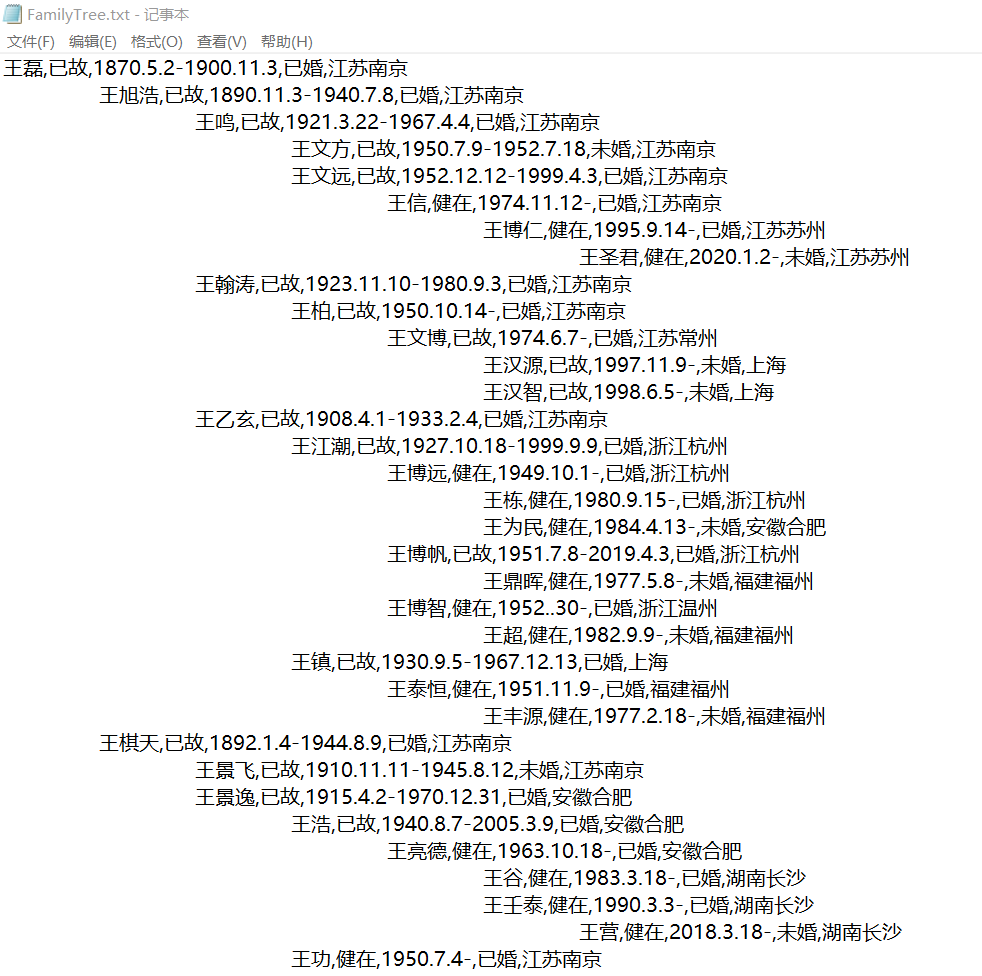
            break;

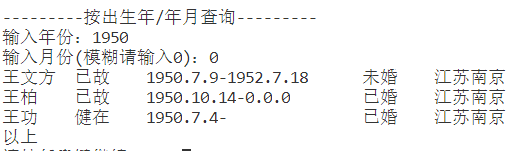
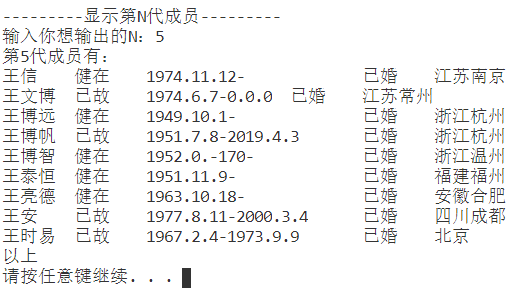
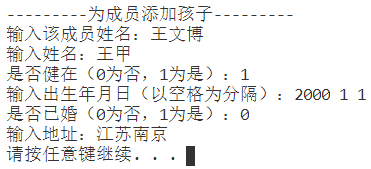
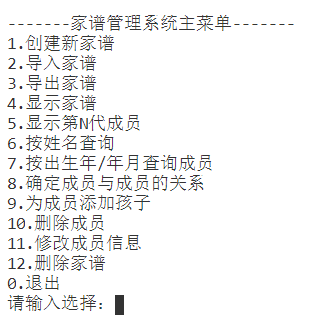
    }

}

## 4. 测试数据和结果

原始家谱的数据：





## 5. 小结

所有基于深搜的算法时间复杂度均为O（n）。

该程序的优化可在查询第N代所有成员，可以改为广搜，这样就减少无用功，提高效率。

# Huffman编码与解码

## 1. 数据结构

存储结构是霍夫曼树（即二叉树），建立霍夫曼树类Huffman，包含霍夫曼树数组HT，结点数量n，字符的编码数组HC。树结点定义为结构体HTNode，其包含结点权重weight，父亲指针、左右孩子指针（以数组下标为指针）。

## 2. 算法思想

编码算法：先统计各个字符出现次数，生成树结点的权重到霍夫曼树数组HT中。再每次从数组中选取两个无双亲最小的权重结点，生成他们的父结点，如此重复直到只剩一个结点，即根结点。接下来进行编码，从底部每一个叶子结点根据父亲指针向上走，判断当前编0还是1，最后将生成的01字符串倒置后放入HC数组中。

转码算法：逐个从source.txt中读取字符，转为对应01编码，存在bin数组的第1到7个单元中（0号空着），满了后将bin数组对应二进制转为十进制再转为ASCII字符写入文件，并清空bin数组，如此重复。

解码算法：从code.dat文件中逐个读取字符，将其转为ASCII对应十进制数，然后将其转为二进制从高位一位一位存到temp数组中，存一位对比一次temp数组对应01字符串是否为预设编码，是的话转为对应字符写入recode.txt，并清空temp继续上述操作。

## 3. 源代码

//4.Huffman编码与解码

#include <iostream>

#include <cstring>

#include <cmath>

#include <fstream>

#include <string.h>

#define N 256

using namespace std;

typedef struct

{

    unsigned int weight;

    unsigned int parent, lchild, rchild;

    char word;

} HTNode, \*HufTree;

class Huffman

{

    HufTree HT;  //霍夫曼树数组

    int n;       //结点数n

    char \*HC[N]; //字符的编码

public:

    void InitHufTree()

    {

        cout << "输入叶结点个数：";

        cin >> n;

        HT = new HTNode[2 \* n];         //动态分配HT数组空间

        for (int i = 0; i < 2 \* n; i++) //HT初始化

        {

            HT[i].weight = 0;

            HT[i].parent = 0;

            HT[i].lchild = 0;

            HT[i].rchild = 0;

        }

        cout << "输入" << n << "个权值：";

        for (int i = 0; i < n; i++)

            cin >> HT[i + 1].weight;

    }

    //选择两个最小的数

    void sel\_tow\_min(int k, int &s1, int &s2)

    {

        int t = 1;

        while (HT[t].parent != 0)

            t++;

        s1 = t++;

        while (HT[t].parent != 0)

            t++;

        s2 = t++;

        if (HT[s2].weight < HT[s1].weight)

        {

            t = s1;

            s1 = s2;

            s2 = t;

        }

        for (int i = 1; i <= k; i++)

        {

            if (HT[i].parent == 0 && i != s1 && i != s2)

                if (HT[i].weight < HT[s1].weight)

                {

                    s2 = s1;

                    s1 = i;

                }

                else if (HT[i].weight < HT[s2].weight)

                    s2 = i;

        }

    }

    //判断字符a是否已在霍夫曼树数组中

    int Exist(char a)

    {

        for (int i = 0; i < n; i++)

        {

            if (HT[i + 1].word == a)

                return i + 1;

        }

        return 0;

    }

    //统计文档字符数

    void CountCharacters()

    {

        char a;

        int pos;

        ifstream fin("source.txt");

        HT = new HTNode[N]; //动态分配HT数组空间

        n = 0;

        for (int i = 0; i < N; i++) //HT初始化

        {

            HT[i].weight = 0;

            HT[i].parent = 0;

            HT[i].lchild = 0;

            HT[i].rchild = 0;

        }

        fin.get(a);

        while (!fin.eof())

        {

            if (pos = Exist(a)) //如果字符读取过

            {

                HT[pos].weight++;

            }

            else //如果未读取过

            {

                n++;

                HT[n].word = a;

                HT[n].weight++;

            }

            fin.get(a);

        }

    }

    //创建霍夫曼树

    void HufCreat()

    {

        int s1, s2; //存放最小两个数

        if (n <= 1)

            return; //如果只有一个结点，直接返回

        for (int i = n + 1; i < 2 \* n; i++) //建立霍夫曼树

        {

            sel\_tow\_min(i - 1, s1, s2);

            HT[s1].parent = HT[s2].parent = i;

            HT[i].weight = HT[s1].weight + HT[s2].weight;

            HT[i].lchild = s1;

            HT[i].rchild = s2;

        }

        /\*cout << "num" << '\t' << "weight" << '\t' << "parent" << '\t' << "lchild" << '\t' << "rchild" << endl;

        for (int i = 1; i < 2 \* n; i++) //输入霍夫曼树结构

            cout << i << '\t' << HT[i].weight << '\t' << HT[i].parent << '\t' << HT[i].lchild << '\t' << HT[i].rchild << endl;\*/

    }

    //对创建的霍夫曼树进行编码

    void HufCode()

    {

        char \*temp;

        int start;

        ofstream out("Huffman.txt");

        temp = new char[n];

        for (int i = 1; i <= n; i++)

        {

            start = 0;

            for (int c = i, f = HT[i].parent; f != 0; c = f, f = HT[f].parent)

            {

                if (HT[f].lchild == c)

                    temp[start++] = '0';

                else

                    temp[start++] = '1';

            }

            temp[start] = '\0';

            strrev(temp);

            HC[i] = new char[start];

            strcpy(HC[i], temp);

        }

        delete temp;

        for (int i = 1; i <= n; i++)

            out << "字符：" << HT[i].word << "\t出现次数：" << HT[i].weight << "\t编码：" << HC[i] << endl;

    }

    //对文档进行转码

    void HufTransferToStr() //存为字符文件

    {

        char a;

        ifstream fin("source.txt");

        ofstream out("code01.dat");

        fin.get(a);

        while (!fin.eof())

        {

            out << HC[Exist(a)]; //查到a的编码，将编码写入code.dat

            fin.get(a);

        }

    }

    void HufTransfer() //存为二进制文件

    {

        char a;

        char \*b;

        int bin[8] = {0};

        int pos, k = 1;

        int count = 0;

        ifstream fin("source.txt");

        ofstream out("code.dat", ios::binary);

        b = new char;

        while (!fin.eof())

        {

            fin.get(a);

            count++;

            pos = Exist(char(a));

            for (int i = 0; i < strlen(HC[pos]); i++)

            {

                bin[k++] = HC[pos][i] - '0'; //一位一位存编码

                if (k == 8)

                {

                    //存到八位后转为十进制再转ASCII字符写入文档

                    b[0] = char(bin[7] + bin[6] \* 2 + bin[5] \* 4 + bin[4] \* 8 + bin[3] \* 16 + bin[2] \* 32 + bin[1] \* 64);

                    out.write(b, sizeof(char));

                    k = 1; //清空bin数组

                    for (int j = 0; j < 8; j++)

                        bin[j] = 0;

                }

            }

            //fin.get(a);

        }

        fin.close();

        out.close();

    }

    //对压缩数据文件进行解码

    /\*void HufDecode()

    {

        char \*temp;

        char a;

        int start = 0;

        ofstream out("recode.txt");

        ifstream fin("code.dat", ios::binary);

        temp = new char[n];

        fin.get(a);

        while (!fin.eof())

        {

            temp[start++] = a; //一个bit一个bit输入编码temp

            temp[start] = '\0';

            for (int i = 1; i <= n; i++)

            {

                if (strcmp(temp, HC[i]) == 0) //直到当前的编码temp存在，将编码解为字符输出

                {

                    out << HT[i].word;

                    start = 0;

                }

            }

            fin.get(a);

        }

        out.close();

        fin.close();

    }\*/

    void HufDecode()

    {

        char \*temp;

        char a;

        char \*b;

        int int\_a;

        int start = 0;

        int bin[8] = {0};

        int k = 1;

        ofstream out("recode.txt");

        ifstream fin("code.dat", ios::binary);

        b = new char;

        temp = new char[n];

        while (!fin.eof())

        {

            fin.read(b, sizeof(char));

            a = b[0];

            int\_a = int(a);

            k = 1;

            while (k < 8)

            {

                if (int\_a >= pow(2, 7 - k))

                {

                    int\_a = int\_a - pow(2, 7 - k);

                    temp[start++] = '1';

                }

                else

                    temp[start++] = '0';

                k++;

                temp[start] = '\0';

                for (int i = 1; i <= n; i++)

                {

                    if (strcmp(temp, HC[i]) == 0) //直到当前的编码temp存在，将编码解为字符输出

                    {

                        out << HT[i].word;

                        start = 0;

                    }

                    if (fin.eof())

                        break;

                }

                if (fin.eof())

                    break;

            }

        }

        out.close();

        fin.close();

    }

};

int main()

{

    Huffman T; //霍夫曼树

    T.CountCharacters(); //统计字符

    T.HufCreat();        //创建霍夫曼树

    cout<<"霍夫曼树创建完成。\n";

    T.HufCode();         //编码

    cout<<"文档编码完成,内容存储在：Huffman.txt\n";

    T.HufTransfer();     //转码

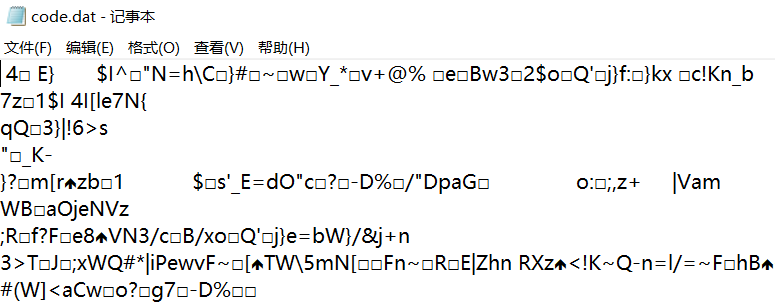
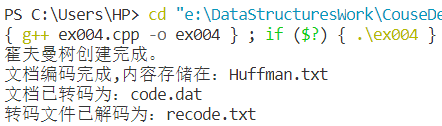
    cout<<"文档已转码为：code.dat\n";

    T.HufDecode();       //解码

    cout<<"转码文件已解码为：recode.txt\n";

}

## 4. 测试数据和结果





## 5. 小结

统计权重和霍夫曼树创建及编码解码算法时间复杂度均为O（n）

# 地铁修建

## 1. 数据结构

采用邻接矩阵存储地图。

## 2. 算法思想

核心算法为利用Prim算法，从1号点开始生成一棵最小生成树，然后记录这颗最小生成树中最大的一条边，当生成树包括最后一号点时输出最大边。

## 3. 源代码

//5.地铁修建

#include <iostream>

#define INF 999999

using namespace std;

int dis[1000];

bool flag[1000];

int ans = 0;

int n, m;

int G[100][100]; //本地编译器有问题暂时为100\*100

int x, y;

void Prim()

{

    int min = INF;

    int temp;

    flag[0] = true;

    for (int i = 0; i < n; i++) //第一个点开始更新dis

        dis[i] = G[0][i];

    for (int i = 0; i < n - 1; i++)

    {

        min = INF;

        for (int j = 0; j < n; j++)

        {

            if (flag[j] == false && dis[j] < min)

            {

                min = dis[j];

                temp = j;

            }

        }

        if (min > ans)

            ans = min;

        flag[temp] = true;

        for (int j = 0; j < n; j++)

        {

            if (flag[j] == false && G[temp][j] < INF)

                if (G[temp][j] < dis[j])

                    dis[j] = G[temp][j];

        }

        if (temp == n - 1)

            cout << ans;

    }

}

int main()

{

    cin >> n >> m;

    for (int i = 0; i < n; i++) //初始化邻接矩阵

    {

        dis[i] = INF;

        flag[i] = false;

        for (int j = 0; j < n; j++)

            G[i][j] = INF;

    }

    for (int i = 0; i < m; i++) //输入边

    {

        cin >> x >> y;

        cin >> G[x - 1][y - 1];

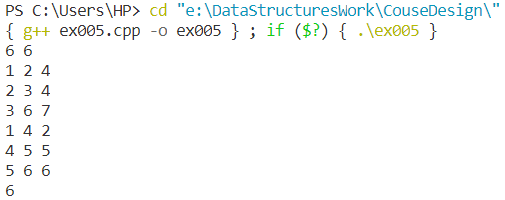
        G[y - 1][x - 1] = G[x - 1][y - 1];

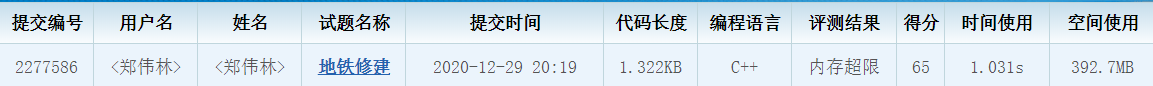
    }

    Prim();

}

## 4. 测试数据和结果





## 5. 小结

普通Prim算法时间复杂度为O（n²）。因此可以对Prim进行优化，可以利用并查集来减少搜索点时的时间。

# 公交线路提示

## 1. 数据结构

建立地图类Map，包含站点数组sta、公交数组bus，以及对应的总站点数allstanum、总公交数allbugnum；站点结构为Station结构体，其包含站点名称name，经过的公交车编号数组pass，及经过公交车线路数passnum；公交车结构为Bus结构体，包含公交车编号num（即几路车），经过的站点编号route数组，及经过的站点数stanum；最后建立一个Path结构体，用以记录搜索过程中的路径，包含当前站点所属公交线bus\_num，前一站点编号pre。

## 2. 算法思想

核心搜索算法为广度优先搜索。

搜索最少站点路线：广搜由队列实现。先将起点入队，标记为访问过，然后当队不为空时循环：队头出队，先判断是否为终点，是的话输出路线，否则将出队站所涉及的所有公交车中其前后两个未被访问过的站点入队，并标记访问过。

搜索最少转站路线：同样为广搜，只不过将入队条件改为出队站所涉及的所有公交车线路中的未访问过的站点。

## 3. 源代码

//6.公交线路提示

#pragma warning(disable : 4996)

#include <iostream>

#include <stdlib.h>

#include <cstring>

#include <fstream>

#define OK 1

#define OVERFLOW 0

#define ERROR 0

using namespace std;

struct Station //站点结构体

{

    char name[30];      //站点名称

    int pass[50]; //经过的公交车编号数组

    int passnum;        //经过公交车线路数

};

struct Bus //公交结构体

{

    int num;        //公交车编号

    int route[100]; //经过的站点编号

    int stanum;     //经过站点数;

};

struct Path

{

    int bus\_num; //当前站点在哪条公交线

    int pre;     //前一站点编号

};

typedef struct LNode //队列结点

{

    int data;

    struct LNode \*next;

} LNode, \*QueuePtr;

class LinkQueue

{

    QueuePtr front;

    QueuePtr rear;

public:

    //初始化操作

    int InitQueue()

    {

        front = new LNode;

        if (front == NULL)

            exit(OVERFLOW);

        front->next = NULL;

        rear = front;

        return OK;

    }

    bool QueueEmpty()

    {

        if (front)

        {

            if (front == rear)

            {

                return true;

            }

            else

            {

                return false;

            }

        }

        return true;

    }

    //入队

    int EnQueue(int e)

    {

        LNode \*s;

        if (front)

        {

            s = new LNode;

            if (s == NULL)

                exit(OVERFLOW);

            s->data = e;

            rear->next = s;

            rear = s;

            rear->next = NULL;

        }

    }

    //出队

    int DeQueue(int &e)

    {

        LNode \*p;

        if (front && (front != rear))

        {

            p = front->next;

            e = p->data;

            front->next = p->next;

            if (rear == p)

                rear = front;

            delete p;

            return OK;

        }

        return ERROR;

    }

};

int \*visited;

Path \*path;

void InitVisited()

{

    visited = new int[10000];

    path = new Path[10000];

    for (int i = 0; i < 10000; i++)

    {

        visited[i] = 0;

        path[i].bus\_num = path[i].pre = -1;

    }

}

class Map //地图类

{

    Station \*sta;

    int allstanum; //总站点数

    Bus bus[1000];

    int allbusnum; //总公交数

public:

    void InitMap() //初始化

    {

        sta = new Station[10000];

        for (int i = 0; i < 10000; i++)

            sta[i].passnum = 0;

        for (int i = 0; i < 1000; i++)

            bus[i].stanum = 0;

        allstanum = allbusnum = 0;

    }

    void ReadMap()

    {

        int i, pos;

        ifstream fin("南京公交线路.txt");

        char s[1000];

        char c[3];

        char temp[50];

        while (!fin.eof())

        {

            fin >> bus[allbusnum].num;

            fin.getline(s, 1000);

            if (strlen(s) < 3)

                return;

            i = 5;

            while (i < strlen(s)) //为该公交添加经过站点

            {

                strcpy(temp, "");

                while (i < strlen(s) && s[i] != ',') //读站点名

                {

                    c[0] = s[i];

                    c[1] = s[i + 1];

                    c[2] = '\0';

                    strcat(temp, c);

                    i += 2;

                }

                if ((pos = LocateStation(temp)) != -1) //如果站点已存在

                {

                    sta[pos].pass[sta[pos].passnum++] = bus[allbusnum].num; //记录站点经过当前公交

                }

                else //如果不存在，新建站点

                {

                    strcpy(sta[allstanum].name, temp);                                  //拷贝名字

                    sta[allstanum].pass[sta[allstanum].passnum++] = bus[allbusnum].num; //记录站点经过当前公交

                    pos = allstanum;

                    allstanum++; //站点数+1

                }

                if (i <= strlen(s))

                {

                    bus[allbusnum].route[bus[allbusnum].stanum++] = pos;

                    i++;

                }

            }

            allbusnum++; //完成一条公交线路添加

        }

    }

    int LocateStation(char a[30]) //定位XX站在sta中位置

    {

        for (int i = 0; i < allstanum; i++)

        {

            if (strcmp(a, sta[i].name) == 0)

                return i;

        }

        return -1;

    }

    void PrintMap()

    {

        for (int i = 0; i < allbusnum; i++)

        {

            cout << bus[i].num << "路  ";

            for (int j = 0; j < bus[i].stanum; j++)

            {

                cout << sta[bus[i].route[j]].name << ",";

            }

            cout << endl;

        }

    }

    void SearchMinStation(Station start, Station dest) //搜索最少站点路线

    {

        int pre, later; //当前站点在某条线路中前一站和后一站编号

        int bus\_num;    //某公交编号

        int count = 1;  //经过站点统计

        int temp;

        LinkQueue Q;

        int p;

        Q.InitQueue();

        InitVisited();

        Q.EnQueue(LocateStation(start.name)); //起点入队

        visited[LocateStation(start.name)] = 1;

        while (!Q.QueueEmpty())

        {

            Q.DeQueue(temp);

            if (strcmp(sta[temp].name, dest.name) == 0)

            {

                //找到终点

                printf("%s(起点,%d路)-->", sta[temp].name, path[temp].bus\_num);

                p = path[temp].pre;

                while (p != LocateStation(start.name))

                {

                    count++;

                    printf("%s(%d路)-->", sta[p].name, path[p].bus\_num);

                    p = path[p].pre;

                }

                printf("%s(终点)\n", sta[p].name);

                cout << "共" << count + 1 << "站\n";

                return;

            }

            for (int i = 0; i < sta[temp].passnum; i++) //将出队站所涉及所有公交车中其前后两个站点入队

            {

                bus\_num = sta[temp].pass[i];

                FindPreLater(bus\_num, sta[temp].name, pre, later); //找到bus中a站点前后站点编号

                if (visited[pre] == 0 && pre != -1)                //如果未被访问且存在。入队

                {

                    Q.EnQueue(pre);

                    visited[pre] = visited[temp] + 1;

                    path[pre].pre = temp;

                    path[pre].bus\_num = bus\_num;

                }

                if (visited[later] == 0 && later != -1)

                {

                    Q.EnQueue(later);

                    visited[later] = visited[temp] + 1;

                    path[later].pre = temp;

                    path[later].bus\_num = bus\_num;

                }

            }

        }

    }

    void FindPreLater(int busnum, char na[30], int &pre, int &later)

    {

        for (int i = 0; i < allbusnum; i++)

            if (bus[i].num == busnum)

                for (int j = 0; j < bus[i].stanum; j++)

                    if (strcmp(na, sta[bus[i].route[j]].name) == 0)

                    {

                        if (j - 1 >= 0)

                            pre = bus[i].route[j - 1];

                        else

                            pre = -1; //如果该站点是该条线路头一站，则没有前驱

                        if (j + 1 < bus[i].stanum)

                            later = bus[i].route[j + 1];

                        else

                            later = -1; //最后一站，则没有后继

                        return;

                    }

    }

    int FindBusNum(int num)

    {

        for (int i = 0; i < allbusnum; i++)

            if (bus[i].num == num)

                return i;

    }

    void SearchMinTransfer(Station start, Station dest) //搜索最少转站路线

    {

        int bus\_num;   //某公交编号

        int count = 0; //经过站点统计

        int temp;

        LinkQueue Q;

        int p;

        Q.InitQueue();

        InitVisited();

        Q.EnQueue(LocateStation(start.name)); //起点入队

        visited[LocateStation(start.name)] = 1;

        while (!Q.QueueEmpty())

        {

            Q.DeQueue(temp);

            if (strcmp(sta[temp].name, dest.name) == 0)

            {

                //找到终点

                printf("%s(起点,%d路)-->", sta[temp].name, path[temp].bus\_num);

                p = path[temp].pre;

                while (p != LocateStation(start.name))

                {

                    count++;

                    printf("%s(换乘%d路)-->", sta[p].name, path[p].bus\_num);

                    p = path[p].pre;

                }

                printf("%s(终点)\n", sta[p].name);

                cout << "共换乘" << count << "次\n";

                //

                return;

            }

            for (int i = 0; i < sta[temp].passnum; i++) //将出队站所涉及所有公交车中所有站点入队

            {

                bus\_num = FindBusNum(sta[temp].pass[i]);

                //将当前线路bus\_num的所有站点入队

                for (int j = 0; j < bus[bus\_num].stanum; j++)

                {

                    if (visited[bus[bus\_num].route[j]] == 0) //未访问过时入队

                    {

                        Q.EnQueue(bus[bus\_num].route[j]);

                        visited[bus[bus\_num].route[j]] = visited[temp] + 1;

                        path[bus[bus\_num].route[j]].pre = temp;

                        path[bus[bus\_num].route[j]].bus\_num = bus[bus\_num].num;

                    }

                }

            }

        }

    }

    Station FindStation(char na[30])

    {

        return sta[LocateStation(na)];

    }

};

int main()

{

    Map M;

    Station a, b;

    string s;

    int sel = 1;

    M.InitMap();

    M.ReadMap();

    while (sel)

    {

        system("cls");

        cout << "-------------南京公交线路提示系统(输入0可退出)--------------\n";

        cout << "输入起点站：";

        while (1)

        {

            cin >> s;

            if (strcmp("0",s.c\_str())==0)

                exit(0);

            if (M.LocateStation((char \*)s.c\_str()) == -1)

                cout << "输入有误请重输:";

            else

                break;

        }

        a = M.FindStation((char \*)s.c\_str());

        cout << "输入终点站：";

        while (1)

        {

            cin >> s;

            if (strcmp("0",s.c\_str())==0)

                exit(0);

            if (M.LocateStation((char \*)s.c\_str()) == -1||strcmp(s.c\_str(),a.name)==0)

                cout << "输入有误请重输:";

            else

                break;

        }

        b = M.FindStation((char \*)s.c\_str());

        cout<<"最少站点路线：\n";

        M.SearchMinStation(b,a);

        cout<<"\n最少转车路线：\n";

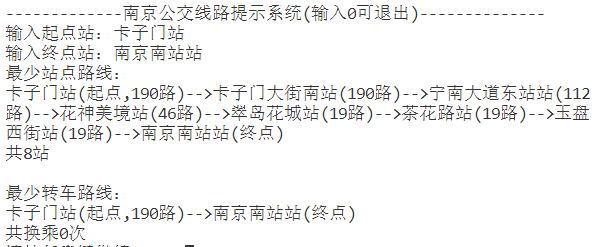
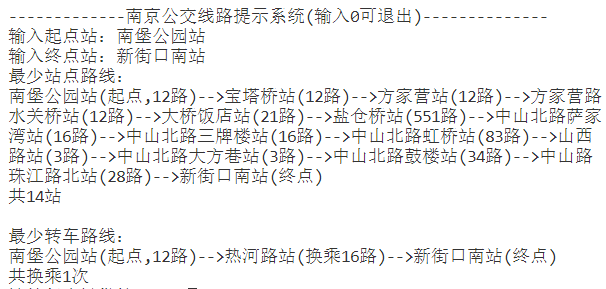
        M.SearchMinTransfer(b, a);

        system("pause");

    }

}

## 4. 测试数据和结果



## 5. 小结

广搜时间复杂度为O（（n）

# B-树的应用

## 1. 数据结构

B-树是一棵M阶树，即它结点最多M个孩子。

存储结构为BTree类，其包含B-树根结点指针root。结点为BTNode结构体，包含结点关键字个数keynum，关键字数组key，父结点指针parent，孩子结点指针数组ptr。

## 2. 算法思想

查询算法：首先将key与根结点中的ki比较，如果key=ki，则查找成功；如果key<ki，则沿着p0指向的子树继续查找；如果ki<key<ki+1，则沿着pi指向的子树继续查找；如果key>kn，则沿着pn指向的子树继续查找；如果此时pi指向空，则查找不成功。

插入算法：第一步，先查找插入的关键字key在B-树底层结点中合适的位置。第二部，插入：如果插入该关键字后当前分支结点关键字个数n<m-1则直接插入到合适位置；如果插入后n=m-1，则需进行分裂。分裂步骤为将key插入该结点且排好序后，取中间位置[m/2]（向上取整）右侧所有关键字放入新建结点中，左侧不动，中间关键字和指向新结点的指针放入原结点的父结点中。如果父结点插入后n=m-1，则继续向上分裂，直到满足n<m-1或到根结点为止。

删除算法：第一步，先查找关键字所处结点q和其位置i。第二步，如果q是底层的结点，则直接删除该最底层分支节点中的该关键字；如果q不是底层结点，则找到q右支中最小的关键字（该关键字必在底层结点），将其覆盖q中的第i个关键字即所要删除的关键字，再将q右支中所找到的最小关键字删除。第三步，删除底层分支结点中的关键字：如果删除后关键字个数不少于[m/2]-1(向上取整)，则直接删除；如果删除后少于[m/2]-1，先向左（或右）兄弟借，如果可以借（即其关键字数大于[m/2]-1），则将左（右）兄弟中最大（小）关键字上移到父结点，再把父结点中大于（小于）上移关键字的关键字下移到q中；如果左右兄弟都借不了了，则需合并结点，即将删除ki后的q与其左（无左时选右）兄弟以及父结点中分割二者的关键字合并为一个结点，如果此操作使得父结点关键字也少于[m/2]-1则继续上诉第三步操作。

## 3. 源代码

//7.B-树的应用

#include <iostream>

#include <fstream>

#include <cmath>

#define MAX\_M 3

using namespace std;

typedef struct node

{

    int keynum;           //结点包含关键字个数

    int key[MAX\_M + 1];   //结点关键字数组，MAX\_M号单元放空

    node \*parent;         //父结点指针

    node \*ptr[MAX\_M + 1]; //孩子结点指针数组

} BTNode;                 //B-树结点结构体

class BTree

{

private:

    BTNode \*root;

public:

    void Creat()

    {

        root = new BTNode;

        root->keynum = 0;

        root->parent = NULL;

        for (int i = 0; i < MAX\_M; i++)

        {

            root->ptr[i] = NULL;

        }

    }

    void Read()

    {

        ifstream fin("B-Tree.txt");

        int temp;

        Creat();

        while (!fin.eof())

        {

            fin >> temp;

            Insert(temp);

        }

    }

    BTNode \*GetRoot()

    {

        return root;

    }

    BTNode \*Search(int e, int &num)

    {

        node \*p;

        int k = 0;

        p = root; //从根结点开始

        while (p != NULL)

        {

            if (e == p->key[k])

            {

                //cout << e << " 查找成功！" << endl;

                num = k;

                return p;

            }

            if (e < p->key[k])

            {

                p = p->ptr[k];

                k = 0;

            }

            else

            {

                k++;

                if (k == p->keynum)

                {

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

                    k = 0;

                }

            }

        }

        return NULL;

    }

    BTNode \*FindPosition(int e)

    {

        node \*p;

        int k = 0;

        p = root;

        while (p->ptr[0] != NULL)

        {

            if (e < p->key[k])

                p = p->ptr[k];

            else

            {

                k++;

                if (k == p->keynum)

                {

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

                    k = 0;

                }

            }

        }

        return p;

    }

    void InsertKey(BTNode \*cur, int e, BTNode \*nex)

    {

        int i, j, mid;

        BTNode \*newnode, \*newroot;

        for (i = 0; i < cur->keynum; i++) //先将该关键字及下一支地址存入当前结点合适的位置

            if (e < cur->key[i])

                break;

        for (j = cur->keynum; j > i; j--)

        {

            cur->key[j] = cur->key[j - 1]; //将关键字和指针向后移

            cur->ptr[j + 1] = cur->ptr[j];

        }

        cur->key[i] = e; //放入新的关键字和指针

        cur->ptr[i + 1] = nex;

        if (cur->keynum < MAX\_M - 1) //当当前分支结点的关键字数小于m-1，直接插入不分裂

        {

            cur->keynum++;

            return; //直接退出，不分裂

        }

        else //否则，分裂处理

        {

            //当cur为根结点时

            if (cur == root)

            { //新开一个根结点,并更新相关项

                newroot = new BTNode;

                newroot->keynum = 0;

                newroot->parent = NULL;

                for (j = 0; j < MAX\_M; j++)

                    newroot->ptr[j] = NULL;

                cur->parent = newroot;

                newroot->ptr[0] = cur;

                root = newroot;

            }

            newnode = new BTNode;

            newnode->parent = cur->parent; //初始化新结点父亲指针

            for (j = 0; j < MAX\_M; j++)

                newnode->ptr[j] = NULL; //初始化新结点孩子指针数组

            i = floor((MAX\_M - 1) / 2) + 1;

            mid = i - 1;

            newnode->keynum = MAX\_M - i;               //初始化新结点关键字数

            cur->keynum = MAX\_M - newnode->keynum - 1; // 更新当前结点剩余关键字数

            j = 0;

            while (i < MAX\_M)

            {

                newnode->key[j] = cur->key[i]; //转移cur中间关键字之后的关键字

                newnode->ptr[j] = cur->ptr[i]; //转移相应孩子指针

                if (newnode->ptr[j])

                    newnode->ptr[j]->parent = newnode;

                cur->ptr[i] = NULL;

                i++;

                j++;

            }

            newnode->ptr[j] = cur->ptr[i]; //转移最后一个指针

            if (newnode->ptr[j])

                newnode->ptr[j]->parent = newnode;

            InsertKey(cur->parent, cur->key[mid], newnode); //中间关键字及指向新结点指针插入cur的父结点中

        }

    }

    void Insert(int e)

    {

        BTNode \*p;

        p = FindPosition(e);

        InsertKey(p, e, NULL);

    }

    BTNode \*FindChildMin(BTNode \*p, int &min)

    {

        if (p->ptr[0] == NULL) //p为最底层

        {

            min = p->key[0]; //覆盖

            return p;

        }

        else

        {

            return FindChildMin(p->ptr[0], min);

        }

    }

    BTNode \*Merge(BTNode \*cur, BTNode \*bro, int num)

    {

        int temp;                //存父结点中下移的关键字

        int flag\_l = 0;          //bro为cur左兄弟标记，1为左兄弟，0为右兄弟

        BTNode \*newnode, \*other; //新结点，存合并后的结点

        int i = 0, j = 0, k = 0;

        if (bro != cur->parent->ptr[num + 1]) //bro为cur左兄弟

        {

            num -= 1; //更新num为父结点下移关键字位置

            flag\_l = 1;

        }

        temp = cur->parent->key[num];

        for (int i = num; i < cur->parent->keynum; i++)

        {

            cur->parent->key[i] = cur->parent->key[i + 1];

            cur->parent->ptr[i + 1] = cur->parent->ptr[i + 2];

        }

        cur->parent->keynum--;

        if (flag\_l) //bro为cur左兄弟

        {

            bro->key[bro->keynum] = temp;            //放temp

            bro->ptr[bro->keynum + 1] = cur->ptr[0]; //把右边的结点头孩子指针转移

            if (cur->ptr[0])

                cur->ptr[0]->parent = bro;

            bro->keynum++;

            newnode = bro;

            other = cur;

        }

        else //否则bro为cur右兄弟

        {

            cur->key[cur->keynum] = temp; //放temp

            cur->ptr[cur->keynum + 1] = bro->ptr[0];

            if (bro->ptr[0])

                bro->ptr[0]->parent = cur;

            cur->keynum++;

            newnode = cur;

            other = bro;

        }

        for (int i = newnode->keynum; i < newnode->keynum + other->keynum; i++)

        {

            newnode->key[i] = other->key[i - newnode->keynum]; //转移另一个节点中的数据

            newnode->ptr[i + 1] = other->ptr[i - newnode->keynum + 1];

            if (newnode->ptr[i + 1] != NULL)

                newnode->ptr[i + 1]->parent = newnode; //更新新孩子指针的父结点

        }

        newnode->keynum = cur->keynum + bro->keynum;

        other = NULL;

        return newnode;

    }

    void DeleteButtonKey(BTNode \*cur, int num)

    {

        int k, temp;

        BTNode \*l, \*r;

        for (int i = num; i < cur->keynum; i++)

        {

            cur->key[i] = cur->key[i + 1]; //删除关键字

            //删除指针?

        }

        //如果cur是根，直接删

        cur->keynum--; //关键字数-1

        if (cur->keynum >= (ceil(MAX\_M / 2.0) - 1) || cur == root) //如果删去后关键字数仍不小于[m/2]-1

        {

            return; //直接删除，返回

        }

        else

        {

            while (cur->keynum < (ceil(MAX\_M / 2.0) - 1))

            {

                temp = 0;

                if (cur->parent == NULL) //如果上溯到根结点

                {

                    if (cur->keynum == 0) //如果根结点空了

                    {

                        if (cur->ptr[0] != NULL) //如果根结点还有孩子

                        {

                            root = cur->ptr[0]; //更改根节点

                            root->parent = NULL;

                        }

                        else

                            root = NULL; //否则B树清空

                        delete cur;

                        break;

                    }

                }

                for (int i = 0; i <= cur->parent->keynum; i++)

                    if (cur->parent->ptr[i] == cur)

                        temp = i;

                if (temp - 1 >= 0 && cur->parent->ptr[temp - 1]->keynum > ceil(MAX\_M / 2.0) - 1) //如果左兄弟的关键字数大于[m/2]-1

                {

                    l = cur->parent->ptr[temp - 1];

                    k = 0;

                    while (k < cur->parent->keynum && l->key[l->keynum - 1] > cur->parent->key[k]) //找到父结点中下移关键字位置

                        k++;

                    for (int i = cur->keynum; i > 0; i--) //下移父结点的关键字,全部右移,放在第一位

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

                    cur->key[0] = cur->parent->key[k];

                    cur->parent->key[k] = l->key[l->keynum - 1]; //上移左兄弟中最大的关键字

                    l->keynum--;                                 //左兄弟少一个关键字

                    cur->keynum++;                               //cur关键字+1

                }

                else

                {

                    if (temp + 1 <= cur->parent->keynum && cur->parent->ptr[temp + 1]->keynum > ceil(MAX\_M / 2.0) - 1) //如果右兄弟的关键字数大于[m/2]-1

                    {

                        r = cur->parent->ptr[temp + 1];

                        k = 0;

                        while (k < cur->parent->keynum && r->key[0] > cur->parent->key[k]) //找到父结点中下移关键字位置

                            k++;

                        k -= 1;

                        cur->key[cur->keynum] = cur->parent->key[k]; //下移父结点的关键字，放最后一位

                        cur->parent->key[k] = r->key[0];             //上移右兄弟中最小的关键字

                        r->keynum--;                                 //右兄弟少一个关键字

                        cur->keynum++;                               //cur关键字+1

                    }

                    else

                    {

                        //如果左右兄弟的关键字数都不够，则进行合并

                        if (temp + 1 <= cur->parent->keynum)

                            Merge(cur, cur->parent->ptr[temp + 1], temp); //右兄弟存在

                        else

                            Merge(cur, cur->parent->ptr[temp - 1], temp); //左兄弟存在

                        cur = cur->parent;                                //上溯

                    }

                }

            }

        }

    }

    void Delete(int e)

    {

        int num;

        BTNode \*q;

        if (q = Search(e, num)) //查找，如果关键字存在

        {

            if (q->ptr[0] == NULL) //关键字位于底层结点

            {

                DeleteButtonKey(q, num); //直接删除

            }

            else //否则从q结点num+1子树中取最小关键字来覆盖，再将这个最小关键字删除

            {

                q = FindChildMin(q->ptr[num + 1], q->key[num]); //查找num+1子树中的最小关键字，覆盖当前，并更新q为所找到的底层结点

                DeleteButtonKey(q, 0);                          //删除底层结点最小关键字

            }

        }

        else //关键字不存在

            cout << "关键字不存在！" << endl;

    }

    void Print(BTNode \*cur)

    {

        if (cur == NULL)

            return;

        if (cur->parent)

            cout << "parent:" << cur->parent->key[0] << '\t';

        cout << "keys:";

        for (int i = 0; i < cur->keynum; i++)

            cout << cur->key[i] << "\t";

        cout << endl;

        for (int i = 0; i <= cur->keynum; i++)

            Print(cur->ptr[i]);

    }

};

int main()

{

    BTree T;

    int e, temp, sel;

    T.Read();

    while (1)

    {

        system("cls");

        cout << "---------B-树管理--------\n";

        cout << "1.查询数据\n2.添加数据\n3.删除数据\n4.显示关系表\n请输入选择：";

        cin >> sel;

        switch (sel)

        {

        case 1:

            cout << "\n请输入你要查询的数据:";

            cin >> e;

            if (T.Search(e, temp))

                cout << '\n'

                     << e << "查找成功！\n";

            else

                cout << '\n'

                     << e << "查找失败！\n";

            system("pause");

            break;

        case 2:

            cout << "\n请输入你要添加的数据:";

            cin >> e;

            if (T.Search(e, temp))

            {

                cout << '\n'

                     << e << "已存在！\n当前关系表为：\n";

                T.Print(T.GetRoot());

            }

            else

            {

                T.Insert(e);

                cout << '\n'

                     << e << "添加成功\n当前关系表为：\n";

                T.Print(T.GetRoot());

            }

            system("pause");

            break;

        case 3:

            cout << "\n请输入你要删除的数据:";

            cin >> e;

            if (!T.Search(e, temp))

            {

                cout << '\n'

                     << e << "不存在！\n当前关系表为：\n";

                T.Print(T.GetRoot());

            }

            else

            {

                T.Delete(e);

                cout << '\n'

                     << e << "删除成功\n当前关系表为：\n";

                T.Print(T.GetRoot());

            }

            system("pause");

            break;

        case 4:

            cout << "\n当前关系表为：\n";

            T.Print(T.GetRoot());

system("pause");

            break;

        case 0:

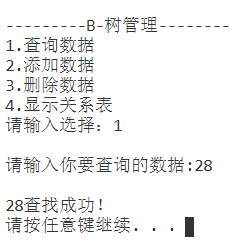
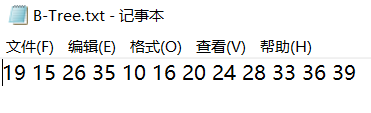
            exit(0);

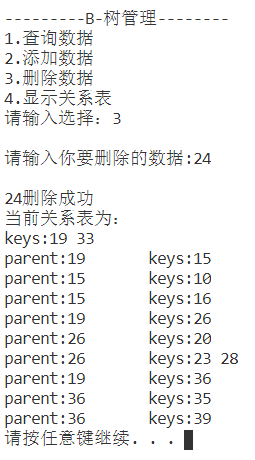
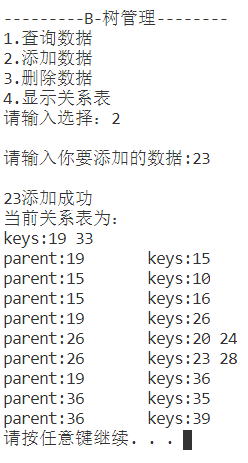
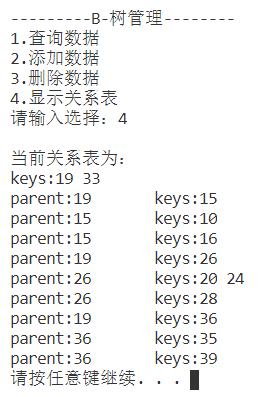
        }

    }

}

## 4. 测试数据和结果





## 5. 小结

三个算法时间复杂度均为O（n²）。

# 排序算法速率比较

## 1. 数据结构

采用二维数组存储十组每组50000个的数据。

## 2. 算法思想

插入排序：将一个记录插入到已经排好队一组有序表的合适位置，如此重复，直到所有记录完成插入。

希尔排序：先将整个待排记录分成若干个子序列，分别进行直接插入排序，待整个序列中记录基本有序（即子序列均呈有序趋势），这个过程可换着分法排若干次，再对全体记录进行直接插入排序。

冒泡排序：从第一个数据开始，左右两个逐一比较交换，往后到最后可将最大（小）排到最后一个记录，如次重复。

快速排序：取一个基准数为枢轴（以第一个记录为例），将大于基准数的放置到枢轴之后，小于的放之前，再递归快速排序前半部分和后半部分。如此重复。

选择排序：每一趟选择剩余记录中最小（大）的数交换到这段序列第一个记录，如此继续对剩下记录排序。

堆排序：先将所有记录建立成一颗完成二叉树即堆，经过调整使其达成大顶堆，再将堆顶记录与最后一个元素交换，使得最大记录存到末尾，再将堆进行调整，重复上述操作。

归并排序：设初始序列有n个记录，可将其看成n个有序的子序列，每个序列长度为1，然后两两归并，得到n/2个长度为2的有序子序列集，如此重复最后得到1个长度为n的有序序列。

基数排序：将记录按照各位的数值分配到编号0~9的十个队列中，再将数据从这十个队列中按0~9顺序取出，如此对十位百位……完成上诉操作，完成排序。

## 3. 源代码

//8.排序算法速率比较

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <Windows.h>

#include <fstream>

#define N 50000 //样例大小，五万个数据

#define RADIX 5 //基数排序中最大基数位数

#define OK 1

#define OVERFLOW 0

#define ERROR 0

using namespace std;

typedef struct LNode

{

    int data;

    struct LNode \*next;

} LNode, \*QueuePtr;

//队列类

class LinkQueue

{

    QueuePtr front;

    QueuePtr rear;

public:

    //初始化、销毁和清空操作

    int InitQueue()

    {

        front = new LNode;

        if (front == NULL)

            exit(OVERFLOW);

        front->next = NULL;

        rear = front;

        return OK;

    }

    //访问型操作

    bool QueueEmpty()

    {

        if (front)

        {

            if (front == rear)

            {

                return true;

            }

            else

            {

                return false;

            }

        }

        return true;

    }

    //加工型操作

    void EnQueue(int e)

    {

        LNode \*s;

        if (front)

        {

            s = new LNode;

            if (s == NULL)

                exit(OVERFLOW);

            s->data = e;

            rear->next = s;

            rear = s;

            rear->next = NULL;

        }

    }

    int DeQueue(int &e)

    {

        LNode \*p;

        if (front && (front != rear))

        {

            p = front->next;

            e = p->data;

            front->next = p->next;

            if (rear == p)

                rear = front;

            delete p;

            return OK;

        }

        return ERROR;

    }

};

//int S[10][N + 1]; //存储样例用数组

int E[10][N + 1]; //排序用数组

//插入排序

void InsertSort(int a[N + 1])

{

    int j, temp;

    for (int i = 2; i <= N; i++)

    {

        temp = a[i];

        for (j = i; j > 1; j--)

        {

            if (temp < a[j - 1])

                a[j] = a[j - 1];

            else

                break;

        }

        a[j] = temp;

    }

}

//希尔排序

void ShellSort(int a[N + 1])

{

    int d[3] = {5, 3, 1};

    int temp, j;

    for (int k = 0; k < 3; k++)

        for (int i = d[k] + 1; i <= N; i++)

            if (a[i] < a[i - d[k]])

            {

                temp = a[i];

                for (j = i; j > d[k]; j = j - d[k])

                {

                    if (temp < a[j - d[k]])

                        a[j] = a[j - d[k]];

                    else

                        break;

                }

                a[j] = temp;

            }

}

//冒泡排序

void BubbleSort(int a[N + 1])

{

    int temp;

    for (int i = 1; i <= N - 1; i++)

        for (int j = 1; j <= N - i; j++)

            if (a[j + 1] < a[j])

            {

                temp = a[j];

                a[j] = a[j + 1];

                a[j + 1] = temp;

            }

}

//快速排序

void QuickSort(int k, int head, int tail)

{

    if (head >= tail)

        return; //当头大于等于尾，直接返回

    int randbase = (rand() % (tail - head + 1)) + head;

    int base = E[k][head];

    E[k][head] = E[k][randbase];

    E[k][randbase] = base; //取随机位置做基数

    base = E[k][head]; //设置基数

    int i = head, j = tail;

    if (head >= tail)

        return;   //当头大于等于尾，直接返回

    while (i < j) //否则调整数据

    {

        while (i < j && E[k][j] >= base) //找到右边比基数小的

            j--;

        if (i < j)

            E[k][i] = E[k][j];           //放入左边的坑

        while (i < j && E[k][i] <= base) //找到左边比基数大的

            i++;

        if (i < j)

            E[k][j] = E[k][i]; //放入右边的坑

    }

    E[k][i] = base; //将基数放入中间位置

    QuickSort(k, head, i - 1); //递归排左边

    QuickSort(k, i + 1, tail); //递归排右边

}

//倒序快速排序

void DeQuickSort(int k, int head, int tail)

{

    if (head >= tail)

        return; //当头大于等于尾，直接返回

    int randbase = (rand() % (tail - head + 1)) + head;

    int base = E[k][head];

    E[k][head] = E[k][randbase];

    E[k][randbase] = base; //取随机位置做基数

    base = E[k][head]; //设置基数

    int i = head, j = tail;

    while (i < j) //否则调整数据

    {

        while (i < j && E[k][j] <= base) //找到右边比基数小的

            j--;

        if (i < j)

            E[k][i] = E[k][j];           //放入左边的坑

        while (i < j && E[k][i] >= base) //找到左边比基数大的

            i++;

        if (i < j)

            E[k][j] = E[k][i]; //放入右边的坑

    }

    E[k][i] = base; //将基数放入中间位置

    DeQuickSort(k, head, i - 1); //递归排左边

    DeQuickSort(k, i + 1, tail); //递归排右边

}

//选择排序

void SeleteSort(int a[N + 1])

{

    int min;

    int temp;

    for (int i = 1; i <= N - 1; i++)

    {

        min = i;

        for (int j = i + 1; j <= N; j++)

            if (a[j] < a[min])

                min = j;

        if (min != i)

        {

            temp = a[i];

            a[i] = a[min];

            a[min] = temp;

        }

    }

}

//大顶堆调整

void HeapAdjust(int a[N + 1], int low, int high)

{

    int temp = a[low];

    int i = low, j = 2 \* i; //j为i的左孩子

    while (j <= high)

    {

        if (j + 1 <= high && a[j + 1] > a[j]) //挑选左右孩子较大者

            j++;

        if (temp >= a[j]) //如果孩子不必根大

            break;

        a[i] = a[j]; //否则继续往下筛选

        i = j;

        j = 2 \* i;

    }

    a[i] = temp; //将原根结点放入合适位置

}

//堆排序

void HeapSort(int a[N + 1])

{

    int temp;

    for (int i = N / 2; i > 0; i--)

        HeapAdjust(a, i, N); //将初始数据调整为大顶堆

    for (int i = N; i > 1; i--)

    {

        temp = a[1];

        a[1] = a[i];

        a[i] = temp;             //交换堆顶和当前位置数据

        HeapAdjust(a, 1, i - 1); //继续调整堆为大顶堆

    }

}

//归并两组数据

void Merge(int a[N + 1], int low, int mid, int high)

{

    int \*T;

    int i, j, k = 1;

    T = new int[high - low + 1];

    i = low;

    j = mid + 1;

    while (i <= mid && j <= high) //将两组数中较小者依次放入T中

    {

        if (a[i] < a[j])

            T[k++] = a[i++];

        else

            T[k++] = a[j++];

    }

    while (i <= mid)

        T[k++] = a[i++]; //剩余数据存入T数组

    while (j <= high)

        T[k++] = a[j++]; //剩余数据存入T数组

    for (k = 1, i = low; i <= high; k++, i++)

        a[i] = T[k];

}

//归并排序

void MergeSort(int a[N + 1])

{

    for (int len = 1; len <= N; len \*= 2)

    {

        int i = 1;

        while (i + 2 \* len <= N)

        {

            Merge(a, i, i + len - 1, i + 2 \* len - 1); //归并两组长度为len的数据

            i = i + 2 \* len;                           //跳到下一组

        }

        if (i + len <= N)

            Merge(a, i, i + len - 1, N); //归并剩下长度不足2个len的数据

    }

}

//基数排序

void RadixSort(int a[N + 1])

{

    LinkQueue Q[10]; //辅助队列

    int radix = 1;   //起始基数

    int m;           //存某位上的数字

    for (int i = 0; i < 10; i++)

        Q[i].InitQueue();

    for (int k = 1; k < RADIX; k++)

    {

        radix \*= 10;

        for (int i = 1; i <= N; i++)

        {

            m = (a[i] % radix) / (radix / 10); //取基数位上的数

            Q[m].EnQueue(a[i]);                //将其入队

        }

        for (int i = 1, m = 0; m < 10; m++) //将所有数出队，完成一次排序

        {

            while (!Q[m].QueueEmpty())

            {

                Q[m].DeQueue(a[i]);

                ++i;

            }

        }

    }

}

//测试类

class SortTest

{

public:

    //创建样例

    void CreatSample()

    {

        ofstream fout("sample.txt");

        int temp;

        for (int i = 0; i < 10; i++)

            for (int j = 1; j <= N; j++)

                E[i][j] = rand(); //生成随机数组

        QuickSort(0, 1, N);   //第一组调正序

        DeQuickSort(1, 1, N); //第二组调倒序

        for (int i = 0; i < 10; i++) //写入文件

        {

            for (int j = 1; j <= N; j++)

            {

                fout << E[i][j] << ' ';

                if (j % 100 == 0)

                    fout << endl;

            }

            fout << endl;

        }

        fout.close();

    }

    //测试函数

    void Test(void (\*Sort)(int a[N + 1]))

    {

        ifstream fin("sample.txt");

        int start\_time;

        for (int i = 0; i < 10; i++)

            for (int j = 1; j <= N; j++)

                fin >> E[i][j]; //每次从样例文件中获取数据到E数组

        start\_time = clock(); //计时开始

        for (int i = 0; i < 10; i++)

            Sort(E[i]);                                             //排序十组数据

        cout << "用时：\t" << clock() - start\_time << "ms" << endl; //计时结束，输出时间

    }

    void Test(void (\*Sort)(int k, int head, int tail)) //Test函数的重载，快排用

    {

        ifstream fin("sample.txt");

        int start\_time;

        for (int i = 0; i < 10; i++)

            for (int j = 1; j <= N; j++)

                fin >> E[i][j];

        start\_time = clock();

        for (int i = 0; i < 10; i++)

            Sort(i, 1, N);

        cout << "用时：\t" << clock() - start\_time << "ms" << endl;

        /\* ofstream fout("out.txt"); //排序输出测试

        for (int i = 0; i < 10; i++)

        {    for (int j = 1; j <= N; j++)

            {

                fout<<E[i][j]<<' ';

                if (j%100==0) fout<<endl;

            }

            fout<<endl;

        }

        fout.close();\*/

    }

};

int main()

{

    SortTest T;

    T.CreatSample();

    cout << "插入排序";

    T.Test(InsertSort);

    cout << "希尔排序";

    T.Test(ShellSort);

    cout << "冒泡排序";

    T.Test(BubbleSort);

    cout << "快速排序";

    T.Test(QuickSort);

    cout << "选择排序";

    T.Test(SeleteSort);

    cout << "  堆排序";

    T.Test(HeapSort);

    cout << "归并排序";

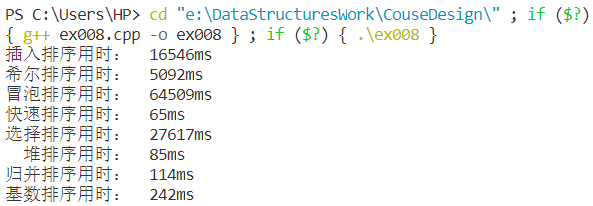
    T.Test(MergeSort);

    cout << "基数排序";

    T.Test(RadixSort);

}

## 4. 测试数据和结果



## 5. 小结

时间复杂度：

插入排序：O（n²） 希尔排序：O（n1.5）

冒泡排序：O（n²） 快速排序：O（nlog2n）

选择排序：O（n²） 堆排序：O（nlog2n）

归并排序：O（nlog2n） 基数排序：O（dn）

# 数字排序【1】

## 1. 数据结构

建立一个一维数组a，类型为自建的Data结构体，Data中包含元素值value以及计数count。

## 2. 算法思想

一个个数据输入，例如输入到temp中，则a[temp]中的count+1，然后再对a数组按count由大到小排序即可。

## 3. 源代码

//9.【1】数字排序   CSP 201503-2

#include <iostream>

#define N 1001

using namespace std;

struct Data

{

    int value;

    int count;

};

//冒泡

void BubbleSort(Data a[N])

{

    Data temp;

    for (int i = 0; i < N-1; i++)

        for (int j = 0; j < N-1 - i; j++)

            if (a[j + 1].count > a[j].count)

            {

                temp = a[j];

                a[j] = a[j + 1];

                a[j + 1] = temp;

            }

}

int main()

{

    Data a[N];

    int n,temp;

    for (int i = 0; i < N; i++)

    {

        a[i].value=i;

        a[i].count=0;

    }

    cin>>n;

    for (int i = 0; i < n; i++)

    {

        cin>>temp;

        a[temp].count++;

    }

    BubbleSort(a);

    for (int i = 0; i < N; i++)

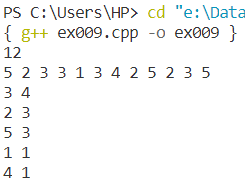
        if (a[i].count>0)

            cout<<a[i].value<<' '<<a[i].count<<endl;

}

## 4. 测试数据和结果





## 5. 小结

时间复杂度为排序的O（n²）。可换用效率更好且稳定的排序方法。

# 魔法优惠券【1】

## 1. 数据结构

建立队列来解决。

## 2. 算法思想

建立四个队列对应正优惠券、负优惠券、正商品和负商品。将其从大到小排序。然后将正优惠券和正商品队列一个个出队相乘直到一个队列为空，负优惠券和负商品队列同样操作。

## 3. 源代码

//11.【1】魔法优惠券    PTA 7-39

#include <iostream>

#include <algorithm>

#include <cmath>

using namespace std;

bool cmp(int a,int b)

{

    return a>b;

}

class quen

{

    int\* a;

    int head, tail;

public:

    quen(int n)

    {

        a=new int[n];

        head = 0;

        tail = 0;

    }

    void EnQ(int e)

    {

        a[tail] = e;

        tail++;

    }

    int DeQ()

    {

        head++;

        return a[head-1];

    }

    void Sort()

    {

        //sort(a+head,a+tail,cmp);

    int temp;

        for (int i = head; i < tail - head - 1; i++)

            for (int j = head; j < tail - head - 1 - i; j++)

                if (abs(a[j + 1]) > abs(a[j]))

                {

                    temp = a[j];

                    a[j] = a[j + 1];

                    a[j + 1] = temp;

                }

    }

    bool Empty()

    {

        if (head==tail)

            return true;

        else

            return false;

    }

};

int main()

{

    int n,m,e,ans=0;

    cin>>n; //输入优惠券

    quen QZ(n),QF(n);

    for (int i = 0; i < n; i++)

    {

        cin>>e;

        if (e>0)

            QZ.EnQ(e);//正的

        else

            QF.EnQ(e);//负的

    }

    cin>>m;//输入商品

    quen SZ(m),SF(m);

    for (int i = 0; i < m; i++)

    {

        cin>>e;

        if (e>0)

            SZ.EnQ(e);

        else

            SF.EnQ(e);

    }

    QF.Sort();QZ.Sort();SZ.Sort();SF.Sort();

    while (!QZ.Empty()&& !SZ.Empty())

    {

        ans+=QZ.DeQ()\*SZ.DeQ(); //正正相乘

    }

    while (!QF.Empty()&& !SF.Empty())

    {

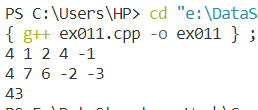
        ans+=QF.DeQ()\*SF.DeQ(); //负负相乘

    }

    cout<<ans;

}

## 4. 测试数据和结果



## 5. 小结

时间复杂度为排序的O（nlogn）。

# 消除类游戏【2】

## 1. 数据结构

建立二维数组map来存储数据，其为自建的Point结构体，包含该位置的值value和是否可以被消除status。

## 2. 算法思想

在读入数据后将所有位置设为不可被消除（即status为0），然后逐行判断有无可消除的序列，将其状态status改为1，再逐列判断一次，最后将状态为1的消除。

## 3. 源代码

//13.【2】消除类游戏    CSP 201512-2

#include <iostream>

using namespace std;

struct Point

{

    int value;

    int status; //是否可被消除

};

int main()

{

    int n, m, i, j;

    Point map[30][30];

    int last;

    int k;

    cin >> n >> m;

    for (int i = 0; i < n; i++)

        for (int j = 0; j < m; j++)

        {

            cin >> map[i][j].value;

            map[i][j].status = 0;

        }

    for (i = 0; i < n; i++) //逐行判断有无可消除，并先标记

    {

        k = 1; //连续个数计数

        last = 0;

        for (j = 0; j < m; j++) //判断每一行

        {

            if (map[i][j].value == last)

                k++; //连续出现计数

            else

            {               //连续中止

                if (k >= 3) //判断是否标记

                    for (int l = 0; l < k; l++)

                        map[i][j - l - 1].status = 1;

                k = 1;

            }

            last = map[i][j].value;

        }

        if (k >= 3) //行末判断有无需要标记

            for (int l = 0; l < k; l++)

                map[i][j - l - 1].status = 1;

    }

    for (j = 0; j < m; j++) //逐列判断有无可消除，并先标记

    {

        k = 1; //连续个数计数

        last = 0;

        for (i = 0; i < n; i++) //判断每一列

        {

            if (map[i][j].value == last)

                k++;

            else

            {

                if (k >= 3)

                    for (int l = 0; l < k; l++)

                        map[i - l - 1][j].status = 1;

                k = 1;

            }

            last = map[i][j].value;

        }

        if (k >= 3)

            for (int l = 0; l < k; l++)

                map[i - l - 1][j].status = 1;

    }

    for (int i = 0; i < n; i++)

    {

        for (int j = 0; j < m; j++)

            if (map[i][j].status == 0)

                cout << map[i][j].value << ' ';

            else

                cout << 0 << ' ';

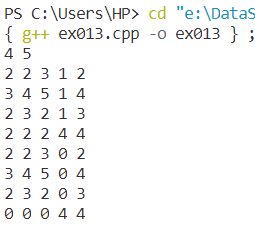
        cout << endl;

    }

}

## 4. 测试数据和结果





## 5. 小结

时间复杂度为O（n²）。

# 最小二叉树【3】

## 1. 数据结构

Prim算法用的是邻接矩阵的结构，Kruskal算法直接用边集输入即可（边集为edge型数组edg，edge结构体包含边起点i，边终点j和边权重w），还用到并查集以便判断回路。

## 2. 算法思想

Prim算法：先从一个点出发，找到距离当前最近的点加入形成一棵子树，再继续找距离当前子树最近的点并加入，直到所有点都再子树中，该树即为最小生成树。

Kruskal算法：先对边集从小到大排序。再对并查集初始化。然后遍历边集，如果两条边可以相连（即连接后的子树不会有回路），则进行合并，直到子树中包含n-1条边，此时子树为最小生成树。

## 3. 源代码

//22.【3】最小二叉树

#include <iostream>

#include <fstream>

#include <algorithm>

#include <cstring>

#define START 0        //起始点

#define INFINITY 99999 //自定义一个无穷大

#define MAX\_V 20       //最大顶点个数

#define MAX\_E 100      //最大边数量

using namespace std;

struct edge //边

{

    int i;

    int j;

    float w;

};

edge edg[MAX\_E];

int visit[MAX\_V] = {0}; //标记已走过的数组

string G\_name[MAX\_V];   //顶点信息

int f[MAX\_V];

int n, m;

//求最小生成树并将其存储到邻接矩阵T中

void Prim(float G[MAX\_V][MAX\_V]) //, int T[MAX\_V][MAX\_V]

{

    int flag[MAX\_V] = {0}; //标记已访问的数组

    int dis[MAX\_V];        //记录最短距离数组

    int path[MAX\_V];       //记录路径数组

    int min, temp, ans = 0;

    flag[0] = 1;                //将第一个顶点标记访问

    for (int i = 0; i < n; i++) //第一个点开始更新dis

    {

        dis[i] = G[0][i];

        if (dis[i] < INFINITY) //初始化路径数组path

            path[i] = 0;

        else

            path[i] = -1;

    }

    for (int i = 0; i < n - 1; i++) //循环n-1次找出最小生成树

    {

        min = INFINITY;

        for (int j = 0; j < n; j++)

        {

            if (flag[j] == 0 && dis[j] < min) //查找最短边

            {

                min = dis[j];

                temp = j; //记录最短边temp

            }

        }

        flag[temp] = 1; //标记走过

        ans += min;

        for (int j = 0; j < n; j++)

        {

            if (flag[j] == 0 && G[temp][j] < INFINITY) //更新dis数组

                if (G[temp][j] < dis[j])

                {

                    dis[j] = G[temp][j];

                    path[j] = temp;

                }

        }

    }

    /\*for (int i = 0; i < n; i++) //生成最小生成树

        for (int j = 0; j < n; j++)

            if (path[j] == i)

            {

                T[i][j] = G[i][j];

                T[j][i] = G[j][i]; //加边

            }\*/

    cout << "Prim算法最小生成树权重和：" << ans << endl;

}

int getfather(int a) //并查集找父亲

{

    if (f[a] == a)

        return a;

    else

    {

        f[a] = getfather(f[a]);

        return f[a];

    }

}

bool merge(int a, int b) //并查集合并两子集

{

    int f1, f2;

    f1 = getfather(a);

    f2 = getfather(b);

    if (f1 != f2) //如果两者没有共同祖先

    {

        f[f2] = f1;  //合并

        return true; //返回true表示可以合并

    }

    return false; //否则有共同祖先，返回false

}

bool cmp(edge a, edge b)

{

    return a.w < b.w;

}

void Kruskal()

{

    int count = 0; //边计数

    int ans = 0;

    edge \*path;

    path = new edge[n - 1];

    sort(edg, edg + m, cmp);    //对边集排序

    for (int i = 0; i < n; i++) //查并集初始化

        f[i] = i;

    for (int i = 0; i < m; i++)

    {

        if (merge(edg[i].i, edg[i].j)) //可以合并

        {

            path[count] = edg[i];

            count++;

            ans += edg[i].w;

        }

        if (count == n - 1)

            break;

    }

    cout << "Kruskal算法最小生成树权重和：" << ans << endl;

    cout << "\n该最小生成树所用边为：" << endl;

    for (int i = 0; i < n - 1; i++)

        cout << G\_name[path[i].i] << "<--->" << G\_name[path[i].j] << "\t权重" << path[i].w << endl;

}

void Read(float G[MAX\_V][MAX\_V])

{

    ifstream fin("matrix.txt");

    fin >> n;

    for (int i = 0; i < n; i++)

        fin >> G\_name[i];

    for (int i = 0; i < n; i++) //邻接矩阵初始化

        for (int j = 0; j < n; j++)

            G[i][j] = INFINITY;

    m = 0;

    while (!fin.eof())

    {

        fin >> edg[m].i >> edg[m].j >> edg[m].w;

        G[edg[m].i][edg[m].j] = edg[m].w;

        G[edg[m].j][edg[m].i] = edg[m].w;

        m++;

    }

    cout << "当前邻接矩阵为：" << endl;

    for (int i = 0; i < n; i++)

        cout << G\_name[i] << "\t";

    cout << "\n\n";

    for (int i = 0; i < n; i++)

    {

        for (int j = 0; j < n; j++)

            cout << G[i][j] << '\t';

        cout << endl;

    }

    cout << endl;

}

int main()

{

    float G[MAX\_V][MAX\_V]; //邻接矩阵

    Read(G);

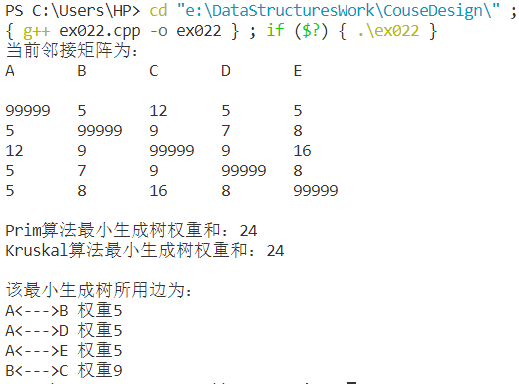
    Prim(G);

    Kruskal();

    return 0;

}

## 4. 测试数据和结果



## 5. 小结

Prim算法时间复杂度为O（n），Kruskal算法时间复杂度为O（nlogn）。

Prim算法可以用堆优化。

# 课程设计总结

**代码行数**

1.系统进程统计   355行  
2.迷宫问题   181行  
3.家谱管理系统    951行  
4.Huffman编码与解码    299行  
5.地铁修建    67行  
6.公交线路提示    372行  
7.B-树的应用    416行  
8.排序算法速率比较   416行  
9.数字排序【1】   46行  
11.魔法优惠券【1】   89行  
13.消除类游戏【2】   77行  
22.最小二叉树【3】   172行  
 共计3445行

**心得体会**

**终于结束了漫长的课程设计。虽然老师早早布置，但由于各种原因还是一拖再拖，最后只有十天才开始动手。于是乎开始了全天候编程模式，每天除了吃饭就是编程，朝九晚十一，紧赶慢赶终于将必做题做完并做了四题选做题。这算是人生到目前为止编程强度最大的一段时间，无论是工作量还是难度都是较大的一个水平。**

**回顾所做的课程设计，也发现不少当时赶工时没注意到问题，进行了稍加改进。印象最深的还是B-树的构建，花费了好几天不断改bug，最后才能够正常的运行并操作B-树，插入算法和删除算法中需要考虑到各种情况以及结构化设计，其中我的解决方法是将插入中的查询为第一部分，再调用插入关键字函数进行插入，而删除较为复杂分为三步，第一步查询，如果不是叶子结点则从右支寻找最大值来覆盖再删除右支最大值，如果是直接调用删除关键字函数，第二步删除关键字，第三步进行需要合并的操作。整个过程中情况稍为复杂，所以该程序标记了最多的中文注释，以便调试和之后的修改作为帮助，因此养成了编程中多做注释的习惯，也深深感受到良好的代码注释对调试和维护的重要性。而说到复杂，另外一题家谱是几个程序中行数最多的一个，将近一千行的代码量，而这里面最复杂的却是从文件导入数据这个过程，需要自己设置数据文件格式，并且考验字符串处理的操作，有一小小个错误都会使所读数据出现偏移而造成重大错误。这个过程中我经历了各种各样的报错，不断地在网上检索资料询问如何解决，最后总发现使字符串处理出错导致的一系列问题。同样的，在Huffman编码中，字符串的处理仍是很重要的一个部分，巧妙地利用ASCII码将八位二进制的01字符串变成一个字节输出，这样才能真正将空间压缩下来。因此还需多加练习字符串处理部分，csp考试中的第三题也往往会考到这部分。而公交线路这题也有字符串处理相关，不过在家谱锻炼之后，在这里的出错减少了许多。这题关键的部分在于数据结构的搭建，如果直接用邻接矩阵搭建一个图，由于其有五千多点站点，这个图将十分巨大，再利用dijkastra算法求解最短路径将十分不易。在他人的提示下，我将车和站点分开存储，但又互相包含相关信息，即车次中包含经过站点，站点中包含经过车次，这样将两者关联起来，再利用广度优先搜索即可很快速的求出答案。因此一个准确的数据结构配合合适的算法对问题求解来说将会有巨大的帮助。**

**由于时间原因没能做到选做题十二分，但在选做题中，有不少挺重要也挺有趣的题目，在寒假时间可以尝试尝试。尤其是系统设计类的题目，通过自己构思出一个系统并将其编写出来，这是一件十分有成就感的事情，同时还能锻炼自己的程序设计和数据结构设计能力。而CSP相关的题目，我还需要学习STL容器的运用，听说这个东西掌握之后做题将十分方便。**

**最后的最后，感谢高航老师的教学和指导。让我系统地学习了数据结构这门程序设计中最为重要的学科之一，并且对各个数据结构都有了较深的印象，能够在编程中较为清晰的分析和搭建。还有感谢帮助过我的同学们，百忙之中也为我解惑。**