**数据结构课程设计报告**

班级：

学号：

姓名：

指导老师： 孙涵

报告

目录

[1、系统进程统计（必做）（链表） 1](#_Toc155138504)

[2、迷宫问题（必做）（栈与队列） 7](#_Toc155138505)

[3、家谱管理系统（必做）（树） 13](#_Toc155138506)

[4、Huffman编码与解码 (必做)（Huffman编码、二叉树） 32](#_Toc155138507)

[5、地铁修建 (必做) （图） 41](#_Toc155138508)

[6、公交线路提示 (必做) （图） 44](#_Toc155138509)

[7、B-树应用 （必做） （查找） 53](#_Toc155138510)

[8、排序算法比较 （必做）（排序） 68](#_Toc155138511)

[14、【2】公共钥匙盒（选做）（线性表，栈，队列） 88](#_Toc155138512)

[15、【3】URL映射（选做）（字符串） 93](#_Toc155138513)

[16、【2】行车路线 (选做)（图） 97](#_Toc155138514)

[17、【2】算术表达式求值 (选做) （栈） 102](#_Toc155138515)

[19、【2】Hash表应用 （选做） （查找） 117](#_Toc155138516)

[21、【3】树的应用 (选做)（树） 128](#_Toc155138517)

# 1、系统进程统计（必做）（链表）

目标1：数据结构类型和算法选择分析

针对所提供的案例，需要有效地管理当前系统进程信息和已结束进程信息。在这种情况下，适当的数据结构和算法选择是至关重要的，以便能够高效地组织、存储和处理数据。

数据结构选择：

单向链表：用于存储当前活动进程。单向链表的优点在于插入和删除操作的效率较高，适用于频繁更新的场景。

双向链表：用于存储已结束进程。双向链表允许双向遍历，便于按照持续时间排序，同时具有快速的节点插入和删除。

算法思想选择：

插入排序：根据进程的内存使用量和持续时间对链表进行排序。插入排序适合链表结构，且在数据量较小的情况下表现良好。

目标2：数据处理和算法分析

数据组织和处理：

插入排序：根据内存使用量和持续时间将进程插入到相应的链表中，并保持链表有序。

算法评估：

插入排序对于小规模数据的排序是相对高效的，但在数据规模较大时可能会受到性能影响。在大规模数据下，可能需要考虑更高效的排序算法，比如归并排序或快速排序。

程序代码：

#include<Windows.h>

#include <stdio.h>

#include <stdlib.h>

#include <psapi.h>

#include <TlHelp32.h>

#include <iostream>

#include <string.h>

#include <wchar.h>

#include <atlcomtime.h>

#include <string>

using namespace std;

typedef struct processInof {

WCHAR pName[500]; //进程名称

unsigned long pMemory; //内存

unsigned time; //持续时间

unsigned id;

int on;

}processInfo;

typedef struct Node {

struct Node\* next;

struct Node\* prev;

processInfo value;

}Node, \* pNode;

typedef struct SNode {

struct SNode\* next;

processInfo value;

}SNode, \*sNode;

int CreateDbLinkList(pNode& pHead,sNode& phead)

{

HANDLE hSnapshot;

HANDLE hProcess;

PROCESSENTRY32 pe32;

PROCESS\_MEMORY\_COUNTERS pmc;

FILETIME ftCreateing, ftExit, ftKernel, ftUser;

// 获取进程快照

hSnapshot = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS, 0);

if (hSnapshot == INVALID\_HANDLE\_VALUE)

{

printf("CreateToolhelp32Snapshot (of processes) 失败");

return 0;

}

// 设置输入参数，结构的大小

pe32.dwSize = sizeof(PROCESSENTRY32);

// 开始列举进程信息

if (!Process32First(hSnapshot, &pe32))

{

printf("Process32First() 失败");

CloseHandle(hSnapshot); // 关闭句柄

return 0;

}

int length = 0;

pNode p\_new, p\_curr, p\_pre;

processInfo value;

pHead = (pNode)malloc(sizeof(Node));

if (pHead == NULL) {

printf("内存分配出错\n");

exit(0);

}

pHead->next = NULL;

pHead->next = NULL;

do {

//打开对应id的进程

hProcess = OpenProcess(PROCESS\_ALL\_ACCESS, FALSE, pe32.th32ProcessID);

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

GetProcessTimes(hProcess, &ftCreateing, &ftExit, &ftKernel, &ftUser);

wcscpy\_s(value.pName, pe32.szExeFile);

value.pMemory = pmc.WorkingSetSize / 1024 / 1024;

value.time = 0;

value.id = pe32.th32ProcessID;

value.on = 1;

//创建新的节点

p\_new = (pNode)malloc(sizeof(Node));

if (p\_new == NULL) {

printf("分配内存出错");

exit(0);

}

p\_curr = pHead->next;

p\_pre = pHead;

p\_new->value = value;

while (p\_curr != NULL && p\_curr->value.pMemory >= value.pMemory) {

p\_pre = p\_curr;

p\_curr = p\_curr->next;

}

p\_new->next = p\_curr;

p\_pre->next = p\_new;

p\_new->prev = p\_pre;

if (p\_curr != NULL)

p\_curr->prev = p\_new;

} while (Process32Next(hSnapshot, &pe32));

CloseHandle(hSnapshot); //关闭句柄

//创建一个单向链表

phead = (sNode)malloc(sizeof(SNode));

if (phead == NULL) {

printf("内存分配错误\n");

exit(0);

}

phead->next = NULL;

return 1;

}

void Traverse(pNode& pHead,sNode &phead)

{

pNode p;

sNode s;

int count = 0;

p = pHead->next;

s = phead->next;

printf("====================正在进行的进程信息================ ========================已经结束的进程==================\n");

printf("进程名称\t 进程持续时间\t 占用内存 进程名称\t 进程结束时间\t 占用内存\n");

while (p || s) {

count++;

if (p) {

printf("%-30ls", p->value.pName);

printf("%-8u\t", p->value.time);

printf("%8ld MB\t", p->value.pMemory);

p = p->next;

}

if (s) {

printf("%-30ls\t", s->value.pName);

printf("%-8u\t", s->value.time);

printf("%8ld MB", s->value.pMemory);

s = s->next;

}

printf("\n");

if (count == 25)

return;

}

}

//双向链表删除

void DeleteDbLinkList(pNode pHead, processInfo value)

{

pNode head = pHead;

pNode pre, cur;

pre = head, cur = head->next;

while (cur) {

if (cur->value.id == value.id) {

pre->next = cur->next;

if (cur->next) {

cur->next->prev = pre;

}

free(cur);

cur = NULL;//防止uaf漏洞

return;

}

pre = cur;

cur = cur->next;

}

printf("双向链表删除失败\n");

}

//双链表插入

void InsertDbLinkList(pNode &pHead ,processInfo value)

{

pNode pre = pHead,cur,p\_new;

cur = pHead->next;

while (cur && cur->value.pMemory >= value.pMemory) {

pre = cur;

cur = cur->next;

}

p\_new = (pNode)malloc(sizeof(Node));

if (p\_new == NULL) {

printf("内存分配失败\n");

exit(0);

}

p\_new->value.id = value.id;

p\_new->value.pMemory = value.pMemory;

wcscpy\_s(p\_new->value.pName,value.pName);

p\_new->value.time = value.time;

pre->next = p\_new;

p\_new->prev = pre;

p\_new->next = cur;

if (cur != NULL)

cur->prev = p\_new;

}

//初始化双向链表

void SetDbLinkList(pNode pHead)

{

pNode p = pHead;

while (p) {

p->value.on = 0;

p = p->next;

}

return;

}

//判断点该元素是否在单向链表内

int isInLinkList(sNode& phead, processInfo value)

{

sNode p = phead->next;

while (p) {

if (p->value.id == value.id)

return 1;

p = p->next;

}

return 0;

}

//从单项连链表中删除元素

int DeleteLinkList(sNode& phead, processInfo value)

{

sNode p = phead, temp;

while (p->next) {

if (p->next->value.id == value.id) {

temp = p->next;

p->next = p->next->next;

free(temp);

return 1;

}

}

printf("单项链表删除失败\n");

return 0;

}

//向单项链表中插入元素

int InsertLinkList(sNode& phead, processInfo value)

{

sNode p\_new;

p\_new = (sNode)malloc(sizeof(SNode));

if (p\_new == NULL) {

printf("内存分配出错\n");

exit(0);

}

p\_new->value.id = value.id;

p\_new->value.pMemory = value.pMemory;

int length = wcslen(value.pName);

if (length > 500)

return 0;

wcscpy\_s(p\_new->value.pName, value.pName);

p\_new->value.time = value.time;

p\_new->next = phead->next;

phead->next = p\_new;

return 1;

}

int UpdateLinkList(pNode pHead, sNode phead, int time)

{

pNode p;

unsigned int id, flag;

processInfo value;

HANDLE hSnapshot;

HANDLE hProcess;

PROCESSENTRY32 pe32;

PROCESS\_MEMORY\_COUNTERS pmc;

// 获取进程快照

hSnapshot = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS, 0);

if (hSnapshot == INVALID\_HANDLE\_VALUE)

{

printf("Cr eateToolhelp32Snapshot (of processes) 失败");

return 0;

}

// 设置输入参数，结构的大小

pe32.dwSize = sizeof(PROCESSENTRY32);

// 开始列举进程信息

if (!Process32First(hSnapshot, &pe32))

{

printf("Process32First() 失败");

CloseHandle(hSnapshot); // 关闭句柄

return 0;

}

//初始化双向链表标志置位

SetDbLinkList(pHead);

do {

hProcess = OpenProcess(PROCESS\_ALL\_ACCESS, FALSE, pe32.th32ProcessID);

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

flag = 0;

id = pe32.th32ProcessID;

//遍历双向链表看是否有相同id的进程

p = pHead->next;

while (p) {

if (p->value.id == id) {

p->value.on = 1;

flag = 1;

p->value.time = time;

break;

}

p = p->next;

}

//如果没有就增添该进程

if (flag == 0) {

value.id = pe32.th32ProcessID;

value.pMemory = pmc.WorkingSetSize/1024/1024;

wcscpy\_s(value.pName, pe32.szExeFile);

InsertDbLinkList(pHead, value);

if (isInLinkList(phead, value)) {

DeleteLinkList(phead, value);

}

}

} while (Process32Next(hSnapshot, &pe32));

CloseHandle(hSnapshot); //关闭句柄

p = pHead;

//寻找结束了的进程

while (p) {

//如果找到该进程

if (p->value.on == 0) {

//添加到单项链表中

InsertLinkList(phead, p->value);

}

p = p->next;

}

return 1;

}

int main()

{

pNode pHead;

sNode phead;

int time = 0;

CreateDbLinkList(pHead, phead);

while (1) {

Traverse(pHead, phead);

UpdateLinkList(pHead, phead, time);

Sleep(1000);

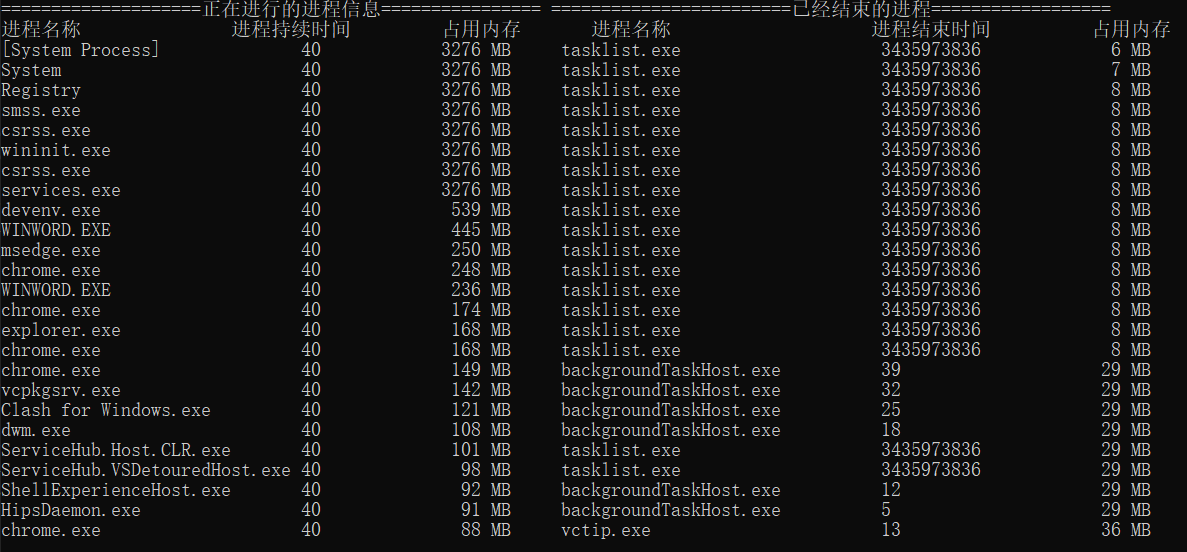
system("cls");

time++;

}

return 0;

}



目标4：设计思路、问题和解决方法的清晰表达

设计思路：

利用单向链表和双向链表分别存储当前活动进程和已结束进程，根据进程的内存使用量和持续时间进行排序。

出现的问题及解决方法：

性能问题：插入排序可能在大规模数据下效率较低。

解决方法：针对大规模数据可以考虑其他更高效的排序算法，如归并排序或快速排序。

时间复杂度分析

插入排序：对于单向链表和双向链表的插入排序时间复杂度为 O(n)。

遍历并打印链表：遍历单向链表和双向链表的时间复杂度为 O(n)。

可改进之处

算法优化：针对大规模数据，考虑使用更高效的排序算法。

# 2、迷宫问题（必做）（栈与队列）

目标1: 数据结构类型和算法选择分析

为解决迷宫问题，需要考虑以下因素：

数据结构类型：

二维数组（地图表示）：用于存储迷宫地图数据。

栈（Stack）：用于记录迷宫的路径。

递归：作为算法的一部分，用于搜索迷宫路径。

算法类型：

深度优先搜索（DFS）：通过栈或递归，沿着一个路径探索到不能再探索为止，然后回溯并尝试下一个路径。

标记回溯法：标记已访问过的路径，避免重复访问。

目标2: 算法改进及评价

数据处理与算法改进：

栈数据结构改进：使用标准库 std::stack 替代数组作为路径记录的栈，以更好地管理路径节点。

路径标记与回溯逻辑优化：确保对迷宫路径的标记和回溯逻辑正确，避免死循环或错误的路径标记。

测试数据和结果：

测试数据：使用不同大小的迷宫地图和不同入口出口位置。

结果：检查算法输出，确认路径是否正确找到，检验标记回溯逻辑是否正确。

时间复杂度评估：

时间复杂度：基于深度优先搜索的算法，对于每个可能的路径都会进行一次搜索，因此时间复杂度为 O(V + E)，其中 V 是顶点数，E 是边数。

程序代码：

#include <iostream>

#include <fstream>

using namespace std;

typedef struct point

{

int r,c;

}point;

point stack[256];

int num=0;

int find1(int row,int col,int path[][50],int a,int b,int x2,int y2)

{

if((a==x2) && (b==y2))

{

return 1;

}

else

{

if((b!=col-1)&&(path[a][b+1]==0)) //判断是否能向右走

{

path[a][b]=2;

stack[num].r=a;

stack[num].c=b;

num++;

find1(row,col,path,a,b+1,x2,y2);

}

else

{

if((a!=row-1)&&(path[a+1][b]==0)) //不能向右时向下

{

path[a][b]=2;

stack[num].r=a;

stack[num].c=b;

num++;

find1(row,col,path,a+1,b,x2,y2);

}

else

{

if((a!=0) && (path[a-1][b]==0)) //不能向右向下时向上

{

path[a][b]=2;

stack[num].r=a;

stack[num].c=b;

num++;

find1(row,col,path,a-1,b,x2,y2);

}

else

{

if((b!=0) && (path[a][b-1]==0)) //不能向右向上向下时向左

{

path[a][b]=2;

stack[num].r=a;

stack[num].c=b;

num++;

find1(row,col,path,a,b-1,x2,y2);

}

else //无法移动时，标记此格，后退一步

{

path[a][b]=9;

num--;

if(num==0)

{

return 0;

}

int x,y;

x=stack[num].r;

y=stack[num].c;

find1(row,col,path,x,y,x2,y2);

}

}

}

}

}

}

int main()

{

ifstream infile;

infile.open("test.txt");

if(infile)

{

int row,col;

infile>>row;

infile>>col;

int path[row][50];

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

{

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

{

infile>>path[i][j];

}

}

infile.close();

printf("迷宫地图如下图所示：\n");

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

{

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

{

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

}

printf("\n");

}

printf("\n请输入入口和出口位置：\n");

printf("例：1 1 7 7（入口为第1行第1列，出口为第7行第7列）\n");

int x1,x2,y1,y2;

scanf("%d %d %d %d",&x1,&y1,&x2,&y2);

x1--;

x2--;

y1--;

y2--;

if((path[x1][y1]==1) || (path[x2][y2]==1))

{

printf("输入有误，请重新输入！");

}

else

{

int flag;

flag=find1(row,col,path,x1,y1,x2,y2);

if(flag==1)

{

printf("迷宫线路存在：\n");

path[x2][y2]=2;

for(int i=0;i<row;i++) //输出迷宫路线

{

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

{

if((path[i][j]==1))

{

printf("# ");

}

else

{

if(path[i][j]==9||(path[i][j]==0))

{

printf(" ");

}

else

{

printf("o ");

}

}

}

printf("\n");

}

}

else

{

printf("无线路。\n");

}

}

}

else

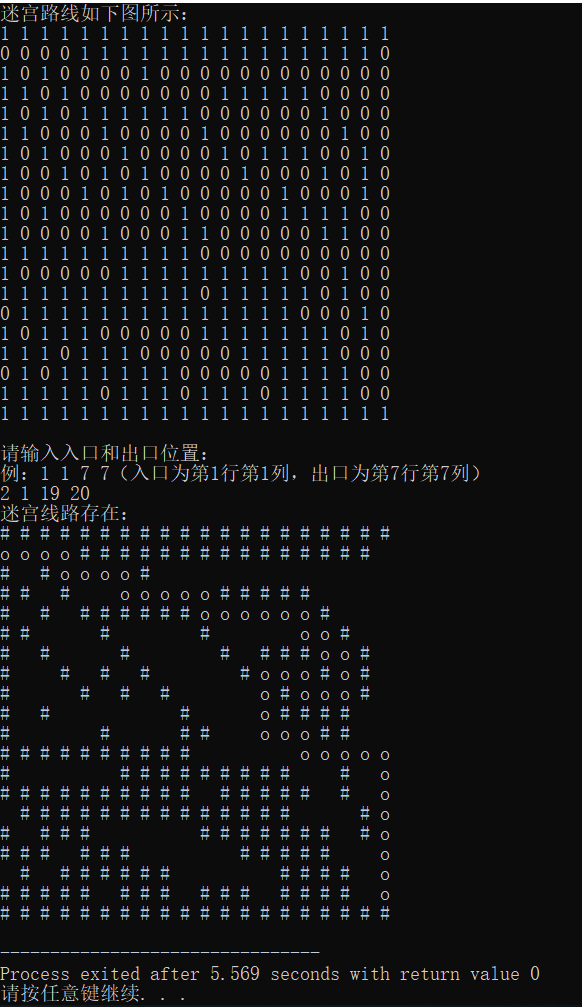
{

printf("文件打开失败");

}

return 0;

}



目标4: 设计思路、问题与解决方法

设计思路：

读取数据：从文件中读取迷宫地图数据。

算法实现：使用递归方式进行深度优先搜索，记录路径信息，并使用栈或标记回溯法解决迷宫问题。

出现的问题及解决方法：

栈操作问题：原始代码中栈操作不完整，无法有效记录路径。

解决方法：改进为使用 std::stack 这样的数据结构，确保正确的入栈和出栈操作。

递归终止条件不完整：递归调用中存在终止条件不明确的问题。

解决方法：确保在递归调用中有明确的结束条件，并在达到条件时正确地结束递归。

路径标记逻辑问题：迷宫路径的标记和回溯逻辑可能有错误，导致路径无法正确识别。

解决方法：优化路径标记和回溯逻辑，确保正确标记和回溯路径，避免重复访问节点。

可改进的地方：

性能优化：对算法进行进一步优化以提高性能，例如避免重复标记或访问已探索的节点。

异常处理：加入对输入数据的异常情况进行更全面的处理，提高程序的健壮性。

# 3、家谱管理系统（必做）（树）

目标1：数据结构类型和算法选择分析

对于给出的实际案例——族谱管理系统，选择了以下数据结构和算法：

数据结构类型：孩子兄弟二叉树

这种树形结构适用于表示家族关系，每个节点代表一个人，拥有指向其孩子和兄弟节点的指针，能有效表示家族中各个成员之间的关系。

算法类型：递归、队列遍历、字符串处理

递归用于树的构建、遍历和搜索。

队列遍历用于获取特定世代的信息，通过层级遍历来检索族谱中特定层次的成员。

字符串处理用于处理个人信息的存储和显示。

目标2：数据处理和算法改进分析

针对具体案例，在处理数据和实现功能需求时采取了以下措施和算法改进：

数据组织与存储：

使用孩子兄弟二叉树结构存储族谱信息，每个节点包含个人信息和指向孩子/兄弟节点的指针。

将族谱信息存储在文件中，通过文件读写操作实现数据的持久化。

算法设计和改进：

采用递归实现族谱树的创建、遍历、搜索和修改等功能，利用树结构的特点进行操作。

使用队列遍历来获取特定世代的信息，从树的根节点开始层级遍历并输出信息。

实现了搜索、添加孩子、保存修改、删除成员、修改成员信息等功能，利用字符串处理函数进行数据处理。

程序代码：

#pragma warning( disable : 4996)

#include <iostream>

#include <fstream>

#include <cstdlib>

#include <queue>

#include <algorithm>

#include <time.h>

#include <graphics.h>

using namespace std;

//定义数据的类型

typedef struct ElemType {

char name[30];

char birth[30];

char marrige[2];

char addr[40];

char isLive[2];

char deadDay[30];

}ElemType;

//定义孩子兄弟二叉树

typedef struct CSNode {

ElemType data;

struct CSNode\* firstchild;

struct CSNode\* nextsibling;

}CSNode, \*CSTree;

typedef struct node

{

CSTree p;

int sx;

int sy;

int ex;

int ey;

}widget;

fstream dataFile;

void findFather(CSTree T,char \*name, CSTree &fa)

{

if (T) {

CSTree son = T->firstchild;

while (son) {

if (!strcmp(son->data.name, name)) {

fa = T;

return;

}

son = son->nextsibling;

}

findFather(T->firstchild, name, fa);

findFather(T->nextsibling, name, fa);

}

}

//打开数据文件，文件中#表示结束

bool OpenFile(string& fileName)

{

dataFile.open(fileName, ios::in|ios::out);

if (!dataFile.is\_open())

return false;

return true;

}

//创建孩子兄弟二叉树

bool CreateCSTree(CSTree& T)

{

char name[30], birth[30], marrige[2],addr[40],isLive[2],deadDay[30];

dataFile >> name;

if (!strcmp(name, "#")) {

T = NULL;

}

else {

T = (CSTree)malloc(sizeof(CSNode));

if (!T) {

cout << "分配内存失败" << endl;

exit(1);

}

dataFile >> birth;

dataFile >> marrige;

dataFile >> addr;

dataFile >> isLive;

if (!strcmp(isLive, "n") || !strcmp(isLive, "N")) {

dataFile >> deadDay;

strcpy(T->data.deadDay, deadDay);

}

strcpy(T->data.name, name);

strcpy(T->data.birth, birth);

strcpy(T->data.marrige, marrige);

strcpy(T->data.addr, addr);

strcpy(T->data.isLive, isLive);

CreateCSTree(T->firstchild);

CreateCSTree(T->nextsibling);

}

return true;

}

//显示这个人的个人信息

void ShowInformation(CSTree T)

{

cout << T->data.name << " " << T->data.birth << " " << T->data.marrige << " " << T->data.addr << " " << T->data.isLive ;

if (!strcmp(T->data.isLive, "n") || !strcmp(T->data.isLive, "N"))

cout << " " << T->data.deadDay;

cout << endl;

}

//获取第n代的信息

bool GetNGenration(CSTree& T, int n)

{

if (!T) {

cout << "族谱为空" << endl;

return false;

}

int depth = 1;

queue<CSTree> q1, q2;

q1.push(T);

while (!q1.empty() || !q2.empty()) {

CSTree temp;

if (!q1.empty()) {

while (!q1.empty()) {

temp = q1.front();

if (depth == n) //如果该层是要输出的层，输出

ShowInformation(temp);

q1.pop();

if (temp->firstchild) q2.push(temp->firstchild); //如果孩子不为空，就放到另一个队列

if (temp->nextsibling) { //如果兄弟不为空，还是把兄弟放进这个队列，继续遍历该层

q1.push(temp->nextsibling);

}

}

}

else {

while (!q2.empty()) {

temp = q2.front();

if (depth == n)

ShowInformation(temp);

q2.pop();

if (temp->firstchild) q1.push(temp->firstchild);

if (temp->nextsibling) {

q2.push(temp->nextsibling);

}

}

}

depth++;

}

return true;

}

//先序遍历

bool InOrderTraverse(CSTree T)

{

if (T) {

ShowInformation(T);

InOrderTraverse(T->firstchild);

InOrderTraverse(T->nextsibling);

}

return true;

}

//获取这个人是第几代

int GetDepth(CSTree T, char \*name)

{

if (!T) {

cout << "族谱为空" << endl;

return 0;

}

int depth = 1;

queue<CSTree> q1, q2;

q1.push(T);

while (!q1.empty() || !q2.empty()) {

CSTree temp;

if (!q1.empty()) {

while (!q1.empty()) {

temp = q1.front();

if (!strcmp(temp->data.name, name)) //如果该元素是要寻找的元素

return depth;

q1.pop();

if (temp->firstchild) q2.push(temp->firstchild); //如果孩子不为空，就放到另一个队列

if (temp->nextsibling) { //如果兄弟不为空，还是把兄弟放进这个队列，继续遍历该层

q1.push(temp->nextsibling);

}

}

}

else {

while (!q2.empty()) {

temp = q2.front();

if (!strcmp(temp->data.name, name)) //如果该元素是要寻找的元素

return depth;

q2.pop();

if (temp->firstchild) q1.push(temp->firstchild);

if (temp->nextsibling) {

q2.push(temp->nextsibling);

}

}

}

depth++;

}

cout << "没有找到" <<name << endl;

}

//查找这个人和它的父亲儿子的信息

bool SearchInformation(CSTree T, char \*name)

{

if (T) {

CSTree father;

CSTree son;

father = T->firstchild;

son = T->firstchild;

if (!strcmp(T->data.name, name)) {

cout << "他本人的信息："<< endl;

ShowInformation(T);

if(son)

cout << "他孩子的信息: " << endl;

while (son) {

ShowInformation(son);

son = son->nextsibling;

}

return true;

}

while (father) {

if (!strcmp(father->data.name, name)) {

cout << "他父亲的信息： "<<endl;

ShowInformation(T);

}

father = father->nextsibling;

}

if (SearchInformation(T->firstchild, name) || SearchInformation(T->nextsibling, name))

return true;

}

return false;

}

//按生日查询

bool SearchForBirth(CSTree T, char\* birth)

{

if (T) {

if (!strcmp(T->data.birth, birth)) {

ShowInformation(T);

return true;

}

if (SearchForBirth(T->firstchild, birth) || SearchForBirth(T->nextsibling, birth))

return true;

}

return false;

}

//确定两人关系

void DeterRela(CSTree T, char\* name1, char\* name2)

{

int depth1 = GetDepth(T, name1);

int depth2 = GetDepth(T, name2);

if (depth1 == depth2) {

cout << name1 << "是" << name2 << "的兄弟" << endl;

}

if (depth1 < depth2) {

if (depth2 - depth1 >= 2)

cout << name1 << "是" << name2 << "的祖父" << endl;

else if (depth2 - depth1 == 1) {

cout << name1 << "是" << name2 << "的父辈" << endl;

}

}

if (depth1 > depth2) {

if (depth1 - depth2 >= 2)

cout << name2 << "是" << name1 << "的祖父" << endl;

else if (depth1 - depth2 == 1) {

cout << name2 << "是" << name1 << "的父辈" << endl;

}

}

}

//添加孩子

int AddChild(CSTree &T, char\* name) {

CSTree p\_new;

if (T) {

if (!strcmp(T->data.name, name)) {

p\_new = (CSTree)malloc(sizeof(CSNode));

if (p\_new == NULL) {

cout << "内存分配出错" << endl;

exit(1);

}

p\_new->firstchild = NULL;

p\_new->nextsibling = NULL;

cout << "请输入要添加孩子的姓名: ";

cin >> p\_new->data.name;

cout << "请输入要添加孩子的生日: ";

cin >> p\_new->data.birth;

cout << "请输入要添加孩子的婚恋状况：";

cin >> p\_new->data.marrige;

cout << "请输入要添加孩子的地址：";

cin >> p\_new->data.addr;

cout << "请输入要添加的孩子的建在状况：";

cin >> p\_new->data.isLive;

if (!strcmp(p\_new->data.isLive, "n")) {

cout << "请输入要添加的孩子的死亡日期：" << endl;

cin >> p\_new->data.deadDay;

}

if (T->firstchild == NULL) {

T->firstchild = p\_new;

return 1;

}

else {

CSTree anotherSon = T->firstchild;

while (anotherSon->nextsibling) {

anotherSon = anotherSon->nextsibling;

}

anotherSon->nextsibling = p\_new;

return 1;

}

}

if (AddChild(T->firstchild, name) || AddChild(T->nextsibling, name))

return 1;

}

return 0;

}

void subPrint(CSTree A, widget\* data, int\* cnt)

{

//清空画布

cleardevice();

\*cnt = 0;

wchar\_t name[20];

//画出退出按钮

RECT r\_exit = { 535,405,600,440 };

rectangle(535, 405, 600, 440);

drawtext(\_T("退出"), &r\_exit, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

//画出返回按钮

RECT r\_return = { 430, 405, 495, 440 };

rectangle(430, 405, 495, 440);

drawtext(\_T("返回"), &r\_return, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

//画出父亲结点

RECT r\_pa = { 260,100,380,150 };

rectangle(260, 100, 380, 150);

//信息转换，utf-8转为Unicode，char转为wchar

int nRet = MultiByteToWideChar(CP\_ACP, 0, A->data.name, 20, name, 20);

drawtext(name, &r\_pa, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

//画出所有子结点

CSTree p = A->firstchild;

while (p)

{

data[\*cnt].p = p;

(\*cnt)++;

p = p->nextsibling;

}

int x = (640 - 60 \* (\*cnt)) / ((\*cnt) + 1);

int i, sx = x;

for (i = 0; i < \*cnt; i++)

{

data[i].sx = sx;

data[i].sy = 270;

data[i].ex = sx + 60;

data[i].ey = 300;

RECT r\_son = { sx,270,sx + 60,300 };

rectangle(sx, 270, sx + 60, 300);

line(sx + 30, 270, 320, 150);

int nRet = MultiByteToWideChar(CP\_ACP, 0, data[i].p->data.name, 20, name, 20);

drawtext(name, &r\_son, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

sx += x + 60;

}

}

void Print(CSTree T)

{

// 设置随机函数种子

srand((unsigned)time(NULL));

// 初始化图形模式

initgraph(640, 480);

// 设置字体

settextstyle(16, 8, \_T("Courier"));

// 设置颜色

COLORREF color = RGB(252, 253, 248);

setbkcolor(color);

settextcolor(BLACK);

setlinecolor(BLACK);

// 定义鼠标参数

MOUSEMSG m;

CSTree tree = T;

CSTree fa = NULL;

widget data[20];

int cnt = 0, i;

subPrint(T, data, &cnt);

while (1)

{

m = GetMouseMsg();

fa = NULL;

if (m.uMsg == WM\_LBUTTONDOWN)

{

//退出

if (m.x <= 600 && m.x >= 535 && m.y >= 405 && m.y <= 440)

{

break;

}

//返回

if (m.x <= 495 && m.x >= 430 && m.y >= 405 && m.y <= 440)

{

findFather(T, tree->data.name, fa);

if (fa != NULL)

{

subPrint(fa, data, &cnt);

tree = fa;

}

}

//选择子结点

for (i = 0; i < cnt; i++)

{

if (m.x >= data[i].sx && m.x <= data[i].ex && m.y >= data[i].sy && m.y <= data[i].ey)

{

tree = data[i].p;

subPrint(tree, data, &cnt);

}

}

}

}

// 关闭图形模式

closegraph();

}

//菜单

void Menue()

{

cout << "\n==========================================族谱管理系统===================================================\n";

cout << " 1.显示n代人的所有信息" << endl;

cout << " 2.按姓名查找" << endl;

cout << " 3.按生日查找" << endl;

cout << " 4.查找两人关系" << endl;

cout << " 5.添加孩子" << endl;

cout << " 6.保存修改" << endl;

cout << " 7.移出族谱" << endl;

cout << " 8.修改成员信息" << endl;

cout << " 9.绘制成员信息" << endl;

cout << "\n========================================================================================================\n";

}

//保存修改

void SaveChange(CSTree& T)

{

if (T) {

dataFile << T->data.name << " " << T->data.birth << " " << T->data.marrige << " " << T->data.addr << " " << T->data.isLive;

if (!strcmp("n", T->data.isLive) || !strcmp("N", T->data.isLive))

dataFile << " " << T->data.deadDay;

dataFile << endl;

SaveChange(T->firstchild);

SaveChange(T->nextsibling);

}

else {

dataFile << "#" << endl;

}

}

//删除成员

void Delete(CSTree& T)

{

CSTree child, sibling;

if (T) {

child = T->firstchild;

sibling = T->nextsibling;

free(T);

T = NULL;

if (child)

Delete(child);

if (sibling)

Delete(sibling);

}

}

int SearchForDelete(CSTree& T, char \*name)

{

if (T) {

if (!strcmp(T->data.name, name)) {

Delete(T->firstchild);

free(T);

T = NULL;

return 1;

}

if(SearchForDelete(T->firstchild, name) || SearchForDelete(T->nextsibling, name))

return 1;

}

}

//修改成员信息

void Change(CSTree& T)

{

int choice;

while (1) {

cout << "请输入要修改信息的序号" << endl;

cout << "1.姓名" << endl;

cout << "2.生日" << endl;

cout << "3.婚否" << endl;

cout << "4.居住地" << endl;

cout << "5.健在" << endl;

cout << "6.死亡日期" << endl;

cout << "7.结束修改" << endl;

cin >> choice;

switch (choice)

{

case 1:

cout << "请输入要修改成的姓名: " << endl;

cin >> T->data.name;

break;

case 2:

cout << "请输入要修改成的生日：" << endl;

cin >> T->data.birth;

break;

case 3:

cout << "请输入新的婚姻状态：" << endl;

cin >> T->data.marrige;

case 4:

cout << "请输入新的住址：" << endl;

cin >> T->data.addr;

case 5:

cout << "请输入新的健在状态：" << endl;

cin >> T->data.isLive;

case 6:

cout << "请输入新的死亡日期：" << endl;

cin >> T->data.deadDay;

case 7:

goto END;

break;

default:

break;

}

}

END:return;

}

bool SearchForChange(CSTree& T, char\* name)

{

if (T) {

if (!strcmp(T->data.name, name)) {

Change(T);

return true;

}

if (SearchForChange(T->firstchild, name) || SearchForChange(T->nextsibling, name))

return true;

}

return false;

}

int main()

{

CSTree T = NULL;

string fileName = "data.txt";

char str[100];

int choice;

if (!OpenFile(fileName)) {

cout << "文件不存在" << endl;

exit(0);

}

CreateCSTree(T);

dataFile.close();

while (1) {

Menue();

cout << "请输入你的选择：";

scanf("%s", str);

choice = atoi(str);

switch (choice)

{

case 1:

int n;

cout << "请输入要查找第几代（一个整数）：";

cin >> n;

GetNGenration(T, n);

break;

case 2:

char name[30];

cout << "请输入要查找的姓名：";

cin >> name;

if (!SearchInformation(T, name))

cout << "没有找到相应信息" << endl;

break;

case 3:

char birth[30];

cout << "请输入要查找的生日：";

cin >> birth;

if(!SearchForBirth(T, birth))

cout << "没有找到相应信息" << endl;

break;

case 4:

char name1[30], name2[30];

cout << "请输入要查询的两人的姓名：" << endl;

cin >> name1 >> name2;

DeterRela(T, name1, name2);

break;

case 5:

cout << "请输入要添加孩子的父亲的姓名：" << endl;

cin >> name;

if (!AddChild(T, name))

cout << "没有找到父亲的信息：" << endl;

break;

case 6:

if (!OpenFile(fileName)) {

cout << "文件不存在" << endl;

exit(0);

}

SaveChange(T);

dataFile.close();

break;

case 7:

cout << "请输入要移出族谱者的姓名：" << endl;

cin >> name;

if (!SearchForDelete(T, name))

cout << "未找到要移出族谱者" << endl;

break;

case 8:

cout << "请输入要修改信息者的姓名：" << endl;

cin >> name;

if (!SearchForChange(T, name))

cout << "未找到要修改信息者" << endl;

break;

case 9:

Print(T);

default:

break;

}

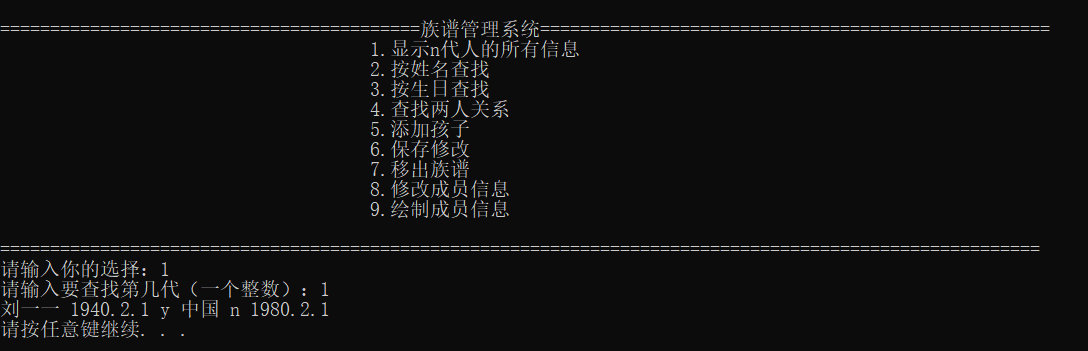
system("pause");

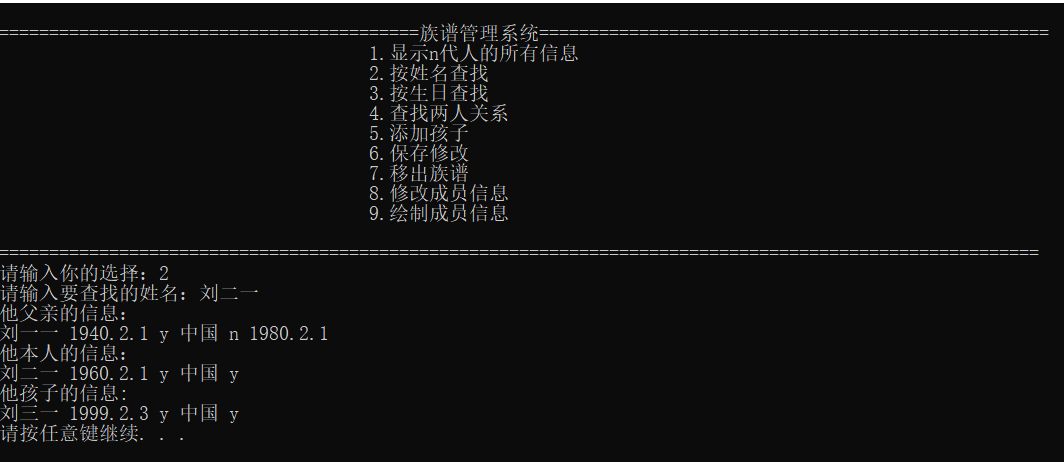
system("cls");

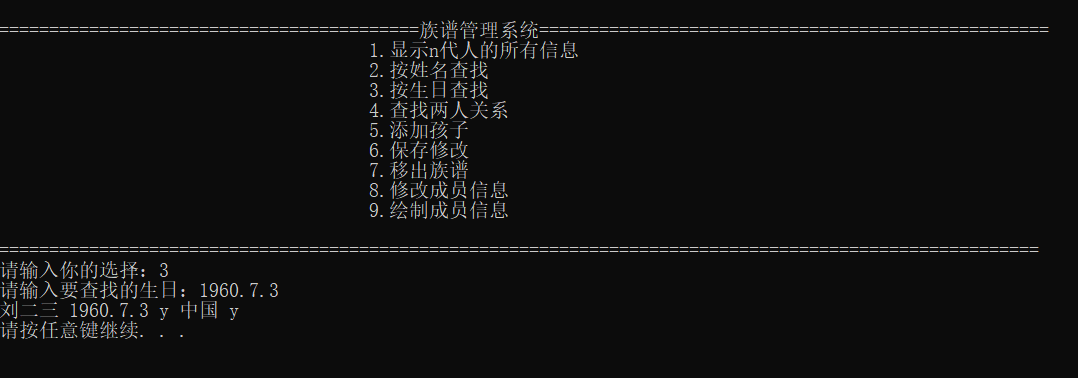
}

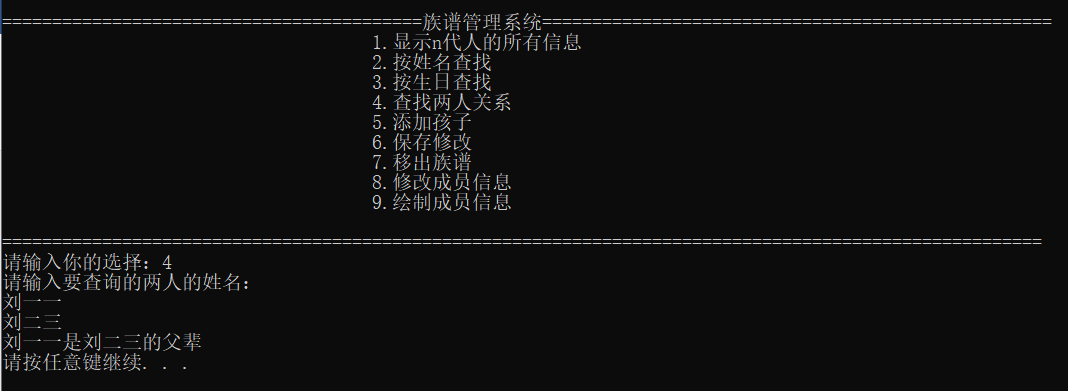
return 0;

}

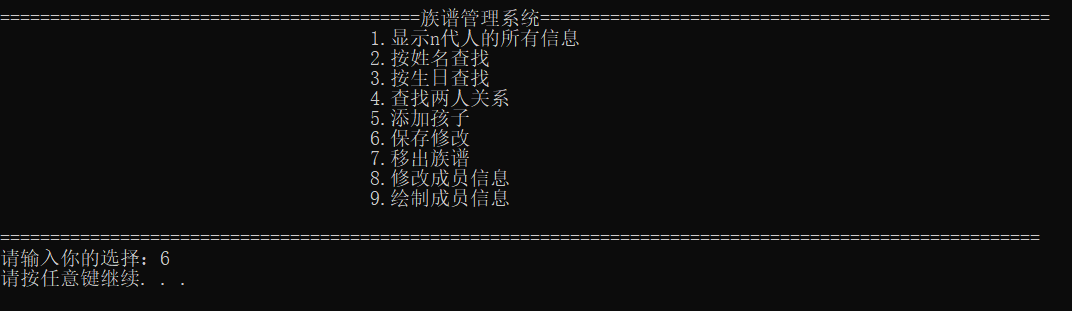


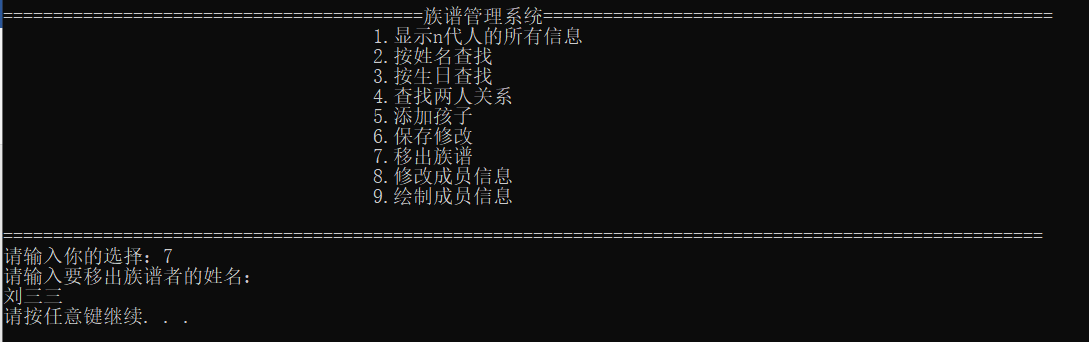


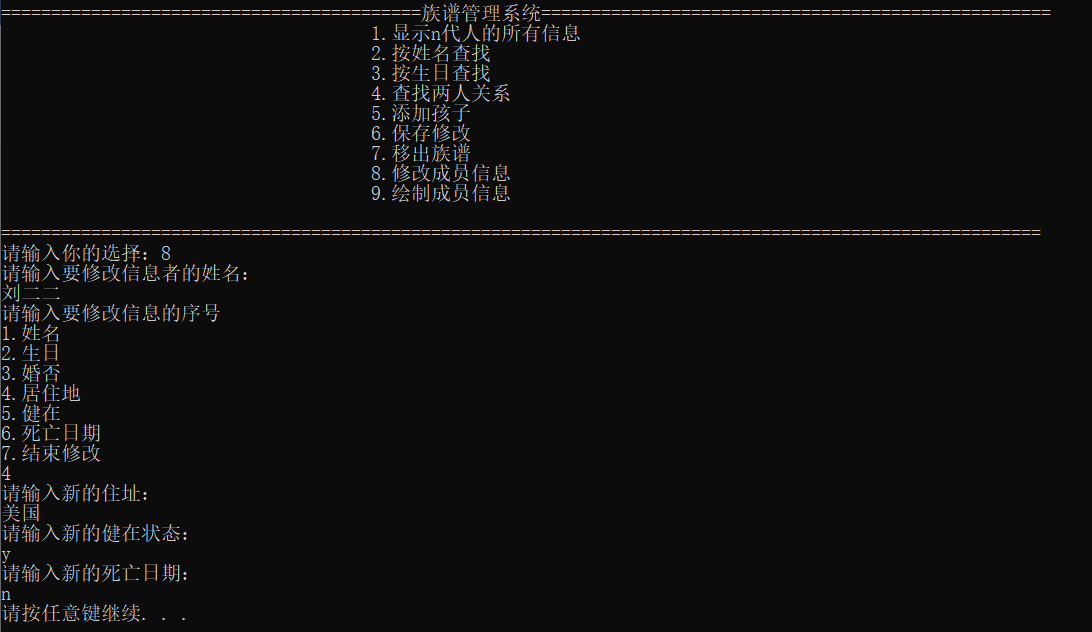


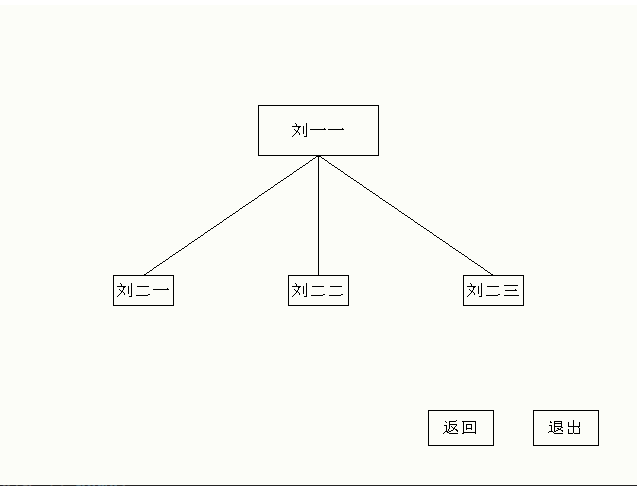
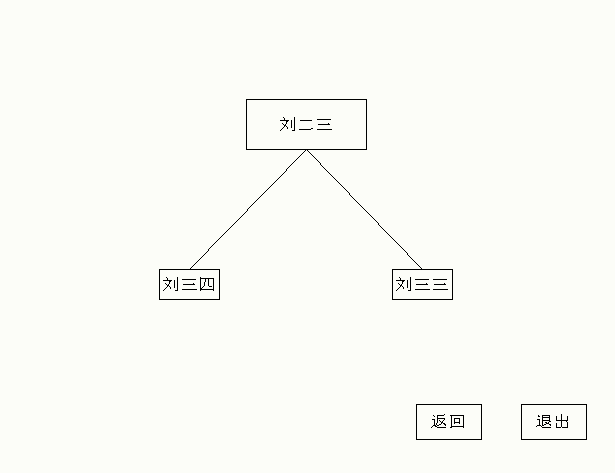












目标4：设计思路、问题及解决方法表达

设计思路：

使用孩子兄弟二叉树结构存储族谱信息，利用递归、队列遍历等算法实现族谱树的操作和功能。

通过文件读写操作实现数据的持久化，将族谱信息存储在文件中，保证数据的长期保存和修改。

出现问题及解决方法：

问题1：数据的有效组织和关系表示。

解决方法：选择孩子兄弟二叉树来表示家族关系，每个节点包含个人信息和指向孩子/兄弟节点的指针，有效地表示了家族成员之间的关系。

问题2：功能实现中的算法复杂度和效率。

解决方法：通过递归和队列遍历等算法，有效实现了对族谱的搜索、遍历、修改等功能，同时根据具体操作采用了字符串处理来操作个人信息，提高了算法的效率。

改进和优化方面：

改进1：优化数据存储方式。

可考虑采用数据库存储方式，提高数据的管理和检索效率。

改进2：改善用户交互和图形界面。

引入更友好的用户界面，提供更直观的族谱图可视化功能。

测试数据和结果：

测试数据涉及族谱成员的姓名、生日、婚姻状况、地址等信息。

结果包括根据特定世代输出的成员信息、个人信息的修改、孩子的添加、两人关系的确定等。

时间复杂度：

数据构建、遍历和搜索操作的时间复杂度取决于树的深度和节点数量，通常为 O(n)。

搜索特定世代信息和确定两人关系的时间复杂度也为 O(n)。

# 4、Huffman编码与解码 (必做)（Huffman编码、二叉树）

目标1：选择合适的数据结构类型和算法类型

数据结构类型选择：

数组：用于存储字符出现的次数。

结构体数组：表示哈夫曼树节点。

栈和队列：辅助进行编码和解码的操作。

算法类型选择：

哈夫曼树构建：采用贪心算法，选择权重最小的两个节点合并成新节点，构建哈夫曼树。

哈夫曼编码：利用哈夫曼树的特性，从叶节点向根节点追溯路径，并生成编码。

目标2：有效组织、存储和处理数据，设计满足功能需求的算法

数据组织和存储：

字符出现次数统计：使用数组记录每个字符出现的次数。

哈夫曼树节点：结构体数组存储每个节点的权重、字符等信息。

编码和解码数据：利用栈和队列辅助处理编码和解码过程。

算法设计与改进：

哈夫曼树构建：通过贪心算法找到最小的两个节点合并构建哈夫曼树。

哈夫曼编码：根据哈夫曼树的特性生成对应的编码。

改进空间：优化哈夫曼树构建过程的效率，提高编码和解码的速度。

程序代码：

#include<cstdio>

#include<cstring>

#include<cstdlib>

#include<iostream>

#include<vector>

#include<stack>

#include<queue>

using namespace std;

#define CharCount 300

// 位操作的数据格式

#define dataType unsigned int

#define dataLen 32

typedef struct {

int count; //字符出现的次数

dataType code;

char codeLen;

}Char;

typedef struct

{

char ch;

int weight;

int parent;

int lchild, rchild;

}HuffmanTree;

FILE\* fp;

char fileName[100];

//打开文件

void fileOpen(const char\* method)

{

fopen\_s(&fp, fileName, method);

if (fp == NULL) {

printf("打开文件失败\n");

return;

}

}

void fileClose()

{

if (fclose(fp) != 0) printf("关闭文件失败\n");

return;

}

void CreateHuffmanTree(HuffmanTree\*& hTree, int n, int m)

{

// 经过n-1次找两个最小值

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

{

int fMin = INT\_MAX, sMin = INT\_MAX - 1;

int fMinP = 0, sMinP = 0;

int j = 1;

//遍历

while (j < m && hTree[j].weight) {

//如果该节点没有双亲,且权值小于第二小的权值

if ((!hTree[j].parent) && (hTree[j].weight < sMin)) {

if (hTree[j].weight < fMin) {

sMin = fMin;

sMinP = fMinP;

fMin = hTree[j].weight;

fMinP = j;

}

else {

sMin = hTree[j].weight;

sMinP = j;

}

}

j++;

}

if (j < m) {

hTree[fMinP].parent = j;

hTree[sMinP].parent = j;

hTree[j].lchild = fMinP;

hTree[j].rchild = sMinP;

hTree[j].weight = fMin + sMin;

}

}

}

//设置x这个数第n比特位的值

void SetBit(dataType& x, int n, int bo)

{

if (bo)

x = x | 1U << n;

else x = x & ~(1U << n);

}

//获取比特位

dataType GetBit(dataType x, int n, int m)

{

return (x & ~(~0U << (m - n + 1)) << n) >> n;

}

//将01字符串形式的二进制转化成整数

void getCode(stack<char> hCode, dataType& code, char& codeLen)

{

codeLen = hCode.size();

if (codeLen > dataLen)

{

return;

}

//字符中的栈顶是二进制的高位

int i = codeLen - 1;

code = 0;

while (!hCode.empty()) {

char bitCh = hCode.top();

printf("%c", bitCh);

SetBit(code, i, bitCh - '0');

hCode.pop();

i--;

}

}

//将整数转化成字符形式的01串的二进制

stack<char> setCode(dataType code, char codeLen)

{

stack<char> hCode;

int i = 0;

while (i < codeLen) {

hCode.push(GetBit(code, i, i) + '0');

i++;

}

return hCode;

}

//哈夫曼编码

void HuffmanEnCode(HuffmanTree\* hTree, int n, int m, Char\* chs)

{

vector<stack<char>> hCodes;

stack<char> temp;

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

int son = i;

int parent = hTree[son].parent;

while (parent) {

if (hTree[parent].lchild == son) temp.push('0');

else temp.push('1');

son = parent;

parent = hTree[son].parent;

}

hCodes.push\_back(temp);

while (!temp.empty()) temp.pop();

}

printf("\n=============Huffman编码结果=================\n");

for (int i = 0; i < hCodes.size(); i++) {

printf("字符：%c----权重：%d----编码：", hTree[i + 1].ch, hTree[i + 1].weight);

getCode(hCodes[i], chs[hTree[i + 1].ch].code, chs[hTree[i + 1].ch].codeLen);

printf("--->%d(\*len:%d\n", chs[hTree[i + 1].ch].code, chs[hTree[i + 1].ch].codeLen);

}

printf("\n");

}

char HuffmanDeCode(HuffmanTree\* hTree, int n, int m, stack<char>hCode)

{

char ch, top;

int pos = m - 1;

while (!hCode.empty()) {

top = hCode.top();

hCode.pop();

if (top == '0') pos = hTree[pos].lchild;

else pos = hTree[pos].rchild;

}

if (hTree[pos].lchild || hTree[pos].rchild) return 0;

return hTree[pos].ch;

}

int main()

{

Char chs[256];

for (int i = 0; i < 256; i++) {

chs[i].count = 0;

}

//读取文本文件，并统计出现次数

strcpy(fileName, "source.txt");

fileOpen("r");

char source[10000];

int count = 0;

int n = 0;

while (!feof(fp)) {

fscanf(fp, "%c", &source[count]);

printf("%c", source[count]);

if (chs[source[count]].count == 0) n++;

chs[source[count]].count++;

count++;

}

fileClose();

//初始化Huffman树

int m = 2 \* n;

HuffmanTree\* hTree = (HuffmanTree\*)malloc(sizeof(HuffmanTree) \* m);

for (int i = 1; i < m; i++) {

hTree[i].weight = hTree[i].parent = hTree[i].lchild = hTree[i].rchild = 0;

}

//将Huffmans树线性表中从1到n位置赋值

int j = 1;

for (int i = 0; i < 256; i++) {

if (chs[i].count) {

hTree[j].parent = hTree[j].lchild = hTree[j].rchild = 0;

hTree[j].weight = chs[i].count;

hTree[j].ch = i;

j++;

}

}

CreateHuffmanTree(hTree, n, m);

HuffmanEnCode(hTree, n, m, chs);

//将字符出现次数和对应的二进制编码保存到文件

strcpy\_s(fileName, "Huffman.txt");

fileOpen("w");

stack<char> hCode;

for (int i = 0; i < 256; i++) {

if (chs[i].count) {

fprintf\_s(fp, "字符：%c---出现次数：%d---二进制编码：", i, chs[i].count);

stack<char> hCode = setCode(chs[i].code, chs[i].codeLen);

while (!hCode.empty()) {

fprintf(fp, "%c", hCode.top());

hCode.pop();

}

fprintf\_s(fp, "\n");

}

}

fileClose();

//对文本进行编码并输出到二进制文件

strcpy(fileName, "code.dat");

fileOpen("wb");

queue<Char> q; //队列

for (int i = 0; i < count;) {

int lenSum = 0;

//取多个字符，他们二进制编码长度加起来小于dataType数据二进制编码长度

while (1) {

lenSum += chs[source[i]].codeLen;

if (lenSum > dataLen) break;

q.push(chs[source[i]]);

i++;

}

//将这多个字符编码后的二进制数据，从dataType高位开始存放

dataType data = 0;

int pos = dataLen - 1; //pos指向要存入文件数据的高位

while (!q.empty()) {

Char temp = q.front();

q.pop();

int j = temp.codeLen - 1, bo = 0;

while (j >= 0) {

bo = GetBit(temp.code, j, j); //获取该位置的二进制数

SetBit(data, pos, bo);

j--;

pos--;

}

}

fwrite(&data, sizeof(dataType), 1, fp);

}

fileClose();

//读取二进制文件并解码

strcpy\_s(fileName, "code.dat");

fileOpen("rb");

char recode[10000];

int i = 0;

dataType data;

//每次从文件中读取一个dataType数据类型的数据

while (fread(&data, sizeof(dataType), 1, fp)) {

//从高位开始

int pos = dataLen - 1, j = 0;

while (pos - j >= 0) {

dataType temp = GetBit(data, pos - j, pos);

char res = HuffmanDeCode(hTree, n, m, setCode(temp, j + 1));

if (res) {

recode[i] = res;

i++;

printf("%c", res);

pos -= (j + 1);

j = 0;

continue;

}

j++;

}

}

fileClose();

strcpy\_s(fileName, "recode.txt");

fileOpen("w");

for (int i = 0; i < count; i++) {

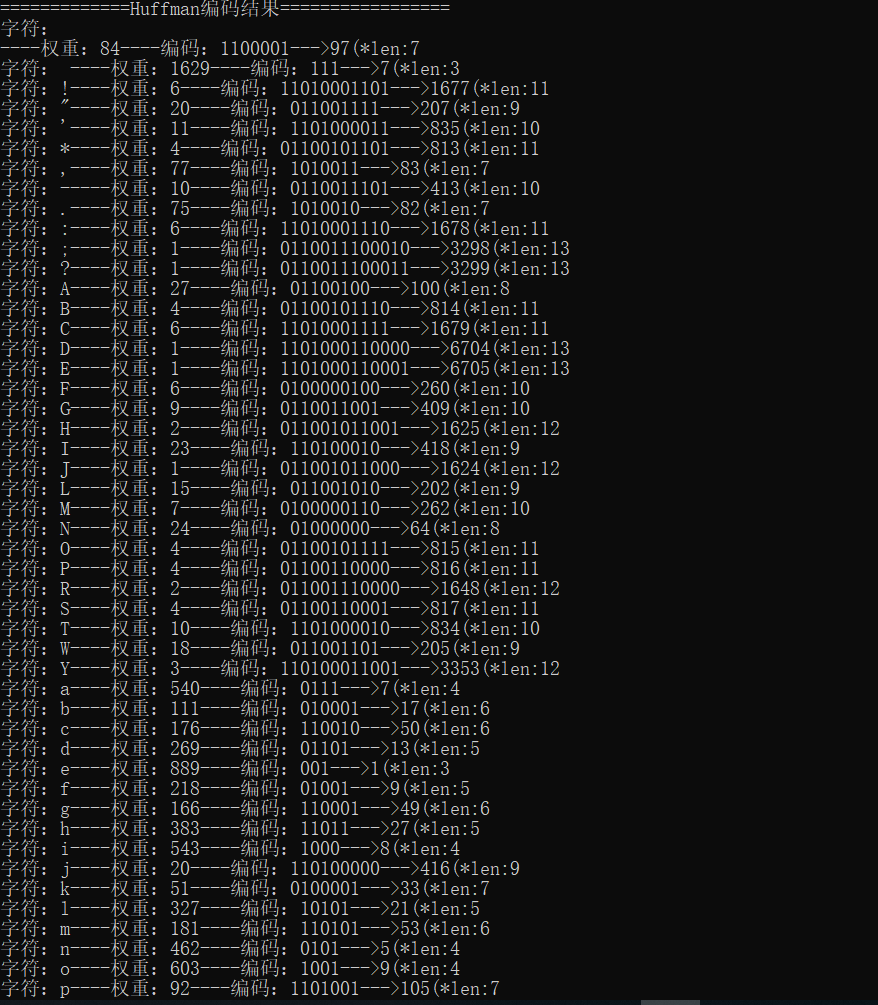
fprintf(fp, "%c", recode[i]);

}

fileClose();

free(hTree);

}



目标4：设计思路、问题及解决方法

设计思路：

通过统计字符出现次数构建哈夫曼树，生成相应编码进行压缩和解压缩。

利用栈和队列辅助进行编码和解码操作。

出现的问题及解决方法：

问题：可能存在编码长度超过数据类型长度的情况。

解决方法：在编码阶段需要检查编码长度，避免超出数据类型的长度范围。

使用的数据结构、算法思想、测试数据和结果、时间复杂度

数据结构：

数组、结构体数组、栈、队列等。

算法思想：

贪心算法用于构建哈夫曼树。

通过追溯哈夫曼树节点生成对应的编码。

测试数据和结果：

测试数据为文本文件内容，根据字符频率构建哈夫曼树，并将编码结果输出到文件。

解码二进制文件并恢复原始文本内容。

时间复杂度：

构建哈夫曼树的时间复杂度：O(nlogn)，n为字符种类数。

编码和解码的时间复杂度与数据长度有关，最坏情况下为O(n)。

改进的地方

性能优化：可以针对哈夫曼树构建、编码和解码的过程进行性能优化，提高算法效率。

容错处理：对于可能的编码长度溢出问题进行更严格的判断和处理。

# 5、地铁修建 (必做) （图）

目标1：分析数据结构类型和算法类型

数据结构类型

结构体：使用结构体 E 表示边，包含边的起点、终点和权重。

并查集：利用数组 father 实现并查集操作，用于判断是否形成环路。

算法类型

排序算法：使用了排序算法对边按权值从小到大进行排序。

并查集：通过并查集实现了判断是否存在环路的操作。

目标2：有效组织、存储和处理数据，改进满足功能需求的算法

数据处理

输入数据：接受用户输入的点的数量和边的数量以及每条边的起点、终点和权重。

数据存储：使用结构体 Edge 存储每条边的起点、终点和权重。

算法改进

边排序：对边按权重进行排序，使得在后续的最小生成树算法中能够按权重递增的顺序选择边。

并查集判断环路：通过并查集实现了判断是否存在环路的功能，从而实现了 Kruskal 算法的最小生成树构建。

程序代码：

#include <iostream>

#include <algorithm>

using namespace std;

#define MAXN 200000

struct E

{

int start;//边的顶点

int end;//边的终点

int weight;//边的权重

}Edge[MAXN+10];

bool cmp(E a,E b)

{

return a.weight<b.weight;

}

int father[100010];

inline int find(int x)//并查集

{

while(father[x]!=x)

{

x = father[x];

}

return x;

}

inline void unite(int x,int y)

{

father[find(x)] = find(y);

}

int main()

{

std::ios::sync\_with\_stdio(false);

std::cin.tie(NULL),cout.tie(NULL);

int n,m;

cin>>n>>m; //输入点的数量和边的数量

int start,end,weight;

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

{

cin>>start>>end>>weight;

Edge[i].start = start;

Edge[i].end = end;

Edge[i].weight = weight;

}

sort(Edge+1,Edge+m+1,cmp);

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

{

father[i] = i;

}

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

{

if(find(Edge[i].start) != find(Edge[i].end))

{

unite(find(Edge[i].start),find(Edge[i].end));

}

if(find(1) == find(n))

{

cout<<Edge[i].weight;

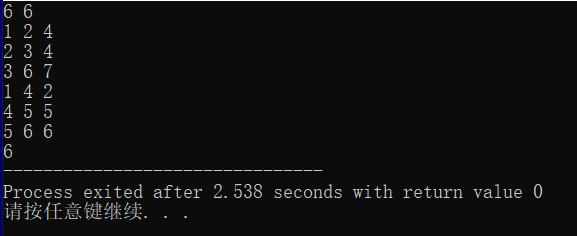
break;

}

}

return 0;

}



目标4：设计思路、出现问题及解决方法

设计思路

数据输入：根据用户输入的点的数量、边的数量以及每条边的起点、终点和权重，构建图。

最小生成树构建：使用 Kruskal 算法，按权重递增的顺序选择边，如果不形成环路则加入最小生成树中。

出现的问题及解决方法

环路检测：通过并查集实现环路检测，但在大规模数据或特定情况下，可能存在性能问题，需要进行优化或改进。

输入数据校验：未进行输入数据的合法性校验，应添加边界条件的检查和异常处理。

数据结构和算法思想

数据结构：利用结构体存储图的边信息，使用并查集判断是否形成环路。

算法思想：Kruskal 算法思想，按边的权重从小到大进行选择和构建最小生成树。

测试数据和结果

测试数据：根据用户输入的点的数量、边的数量以及每条边的起点、终点和权重，构建图并进行测试。

结果：输出了构建的最小生成树的权重，该权重代表了最小生成树的总权值。

时间复杂度

排序：对边按权重排序的时间复杂度为 O(E log E)，其中 E 为边的数量。

并查集操作：并查集的操作时间复杂度近似为 O(E)。

整体时间复杂度：取决于排序操作和并查集操作的时间复杂度，最终整体时间复杂度较优为 O(E log E)。

可改进的地方

性能优化：针对大规模数据或特定情况下并查集的性能问题进行优化，减少不必要的重复查找。

输入数据校验：增加对用户输入数据的边界条件检查和异常处理，以提高程序的健壮性和稳定性。

# 6、公交线路提示 (必做) （图）

目标1：分析数据结构类型和算法类型

数据结构类型

结构体：使用 struct 构建了数据结构 info, arcnode, MyGraph 等，用于表示站点信息、图的节点和边的信息。

动态内存分配：使用 malloc 函数对节点和边动态分配内存。

容器类：使用 priority\_queue, stack 等 STL 容器类，存储节点或辅助数据结构。

算法类型

Dijkstra 算法：用于求解两点之间的最短路径。

图的遍历：利用图的遍历方式寻找最短路径。

目标2：有效组织、存储和处理数据，改进满足功能需求的算法

数据处理

文件读取：从文件中读取数据，获取车站信息和路线信息，构建图的相关数据结构。

数据存储：使用结构体、动态内存分配等方式存储站点信息、图的节点和边的信息。

算法改进

Dijkstra 算法：应用 Dijkstra 算法求解最短路径问题，在两种情况下（转线最少、经过站点最少）进行求解。

程序代码：

#pragma warning( disable : 4996)

#include <stdio.h>

#include <stdlib.h>

#include <Windows.h>

#include <string.h>

#include <vector>

#include <queue>

#include <stack>

using namespace std;

#define maxn 99999 //最大顶点个数

#define INF 99999999 //权值上限

int parent[maxn];

bool visited[maxn];

typedef struct info {

char\* stationName;

char\* rodeName;

}info;

typedef struct arcnode {

int index;

int val;

struct arcnode\* next;

}AR;

typedef struct MyGraph {

int arcnum, vexnum;

info \*vexname;

AR\* N;

}GH;

struct node

{

int id;

int w;

friend bool operator < (node a, node b)

{

return a.w > b.w;

}

};

node d[maxn];

int findvex(char\* s,char \*t, GH\* G)

{

int i;

for (i = 0; i < G->vexnum; i++) {

if (!strcmp(s, G->vexname[i].stationName) && !strcmp(t,G->vexname[i].rodeName))

return i;

}

return -1;

}

int findStation(char\* s, GH\* G)

{

int i;

for (int i = 0; i < G->vexnum; i++) {

if (!strcmp(s, G->vexname[i].stationName))

return i;

}

return -1;

}

int side[maxn];

void Insert(GH\* G, int a, int b)

{

AR\* q = (AR\*)malloc(sizeof(AR));

if (q == NULL) {

printf("error\n");

exit(0);

}

q->index = b;

q->val = 1;

q->next = G->N[a].next;

G->N[a].next = q;

}

void ShowGraph(GH\* G)

{

for (int i = 0; i < G->vexnum; i++) {

printf("%s :", G->vexname[i].stationName);

AR\* p = G->N[i].next;

while (p) {

printf("%s %s->",G->vexname[p->index].rodeName, G->vexname[p->index].stationName);

p = p->next;

}

printf("\n");

}

}

void CreateGH1(GH\* G)

{

int i, j = 0, pos,num = 0;

FILE\* fp;

char roadName[40], str[50], buf[3000];

memset(str, 0, sizeof(str));

memset(buf, 0, sizeof(buf));

memset(side, 0, sizeof(side));

fp = fopen("data.txt", "r");

if (fp == NULL)

printf("open file erro\n");

//初始化图

G->arcnum = 0;

G->vexnum = 0;

G->vexname = (info\*)malloc(maxn \* sizeof(info));

G->N = (AR\*)malloc(maxn \* sizeof(AR));

//从文件中读数据

while (!feof(fp)) {

fscanf(fp, "%s", roadName); //读取是几路车

fgets(buf, sizeof(buf), fp); //获取这一路车所有的站点

for (i = 0; buf[i]; i++) {

if (buf[i] != ',' && buf[i] != ' ' && buf[i] != '\n')

{

str[j] = buf[i];

j++;

}

else if (buf[i] == ',' || buf[i] == '\n') {

pos = findStation(str, G);

if (pos == -1) { //如果该车站以前并没有，那么在图中加入这个车站

G->vexname[G->vexnum].stationName = (char\*)malloc((strlen(str) + 1) \* sizeof(char));

G->vexname[G->vexnum].rodeName = (char\*)malloc((strlen(roadName) + 1) \* sizeof(char));

strcpy(G->vexname[G->vexnum].stationName, str);

strcpy(G->vexname[G->vexnum].rodeName, roadName);

G->N[G->vexnum].next = NULL;

side[num] = G->vexnum;

G->vexnum++;

}

else { //如果这个车站有，就保存它的位置在side中

side[num] = pos;

}

memset(str, 0, sizeof(str));

j = 0;

num++; //站点数量+1

}

}

for (i = 0; i < num-1; i++) {

for (int k = i+1; k < num; k++) {

Insert(G, side[i],side[k]);

Insert(G, side[k], side[i]);

}

}

num = 0;

memset(side, 0, sizeof(side)); //清空side

}

}

void CreateGH2(GH\* G)

{

int i, j = 0, pos, num = 0,pre = -1;

FILE\* fp;

char roadName[40], str[50], buf[3000];

memset(str, 0, sizeof(str));

memset(buf, 0, sizeof(buf));

memset(side, 0, sizeof(side));

fp = fopen("data.txt", "r");

if (fp == NULL)

printf("open file erro\n");

//初始化图

G->arcnum = 0;

G->vexnum = 0;

G->vexname = (info\*)malloc(maxn \* sizeof(info));

G->N = (AR\*)malloc(maxn \* sizeof(AR));

//从文件中读数据

while (!feof(fp)) {

fscanf(fp, "%s", roadName); //读取是几路车

fgets(buf, sizeof(buf), fp); //获取这一路车所有的站点

for (i = 0; buf[i]; i++) {

if (buf[i] != ',' && buf[i] != ' ' && buf[i] != '\n')

{

str[j] = buf[i];

j++;

}

else if (buf[i] == ',' || buf[i] == '\n') {

pos = findStation(str, G);

if (pos == -1) { //如果该车站以前并没有，那么在图中加入这个车站

G->vexname[G->vexnum].stationName = (char\*)malloc((strlen(str) + 1) \* sizeof(char));

G->vexname[G->vexnum].rodeName = (char\*)malloc((strlen(roadName) + 1) \* sizeof(char));

strcpy(G->vexname[G->vexnum].stationName, str);

strcpy(G->vexname[G->vexnum].rodeName, roadName);

G->N[G->vexnum].next = NULL;

if (pre == -1)

pre = G->vexnum;

else {

Insert(G, pre, G->vexnum);

Insert(G, G->vexnum, pre);

pre = G->vexnum;

}

G->vexnum++;

}

else {

if (pre == -1)

pre = pos;

else {

Insert(G, pre, pos);

Insert(G, pos, pre);

pre = pos;

}

}

memset(str, 0, sizeof(str));

j = 0;

num++; //站点数量+1

}

}

num = 0;

pre = -1;

memset(side, 0, sizeof(side)); //清空side

}

}

priority\_queue<node> q;

void Dijkstra(GH \* G,int s)

{

for (int i = 0; i < G->vexnum; i++) {

//初始化

d[i].id = i;

d[i].w = INF;

parent[i] = -1;

visited[i] = false;

}

d[s].w = 0; //源点到源点最短路径是0

q.push(d[s]); //压入队列中

while (!q.empty()) {

node cd = q.top();

q.pop();

int u = cd.id;

if (visited[u])

continue;

visited[u] = true;

AR\* p = G->N[u].next;

//松弛操作

while (p != NULL) {

//找所有与他相邻的顶点，进行松弛操作，更新估算距离，压入队列

int v = p->index;

if (!visited[v] && d[v].w > d[u].w + p->val) {

d[v].w = d[u].w + p->val;

parent[v] = u;

q.push(d[v]);

}

p = p->next;

}

}

}

int main()

{

char start[30],end[30],choice\_str[100];

int ch, st, ed;

GH G1, G2;

CreateGH1(&G1);

CreateGH2(&G2);

while (1) {

memset(d, 0, sizeof(d));

printf("\n================================公交线路系统================================\n");

printf("请输入您要查询的选项： 1.转线最少路线。2.经过站点最少路线。\n");

scanf("%s", choice\_str);

ch = atoi(choice\_str);

printf("请输入起点和终点：(两个站点名称用空格隔开)\n");

scanf("%s %s", start, end);

if (ch == 1) {

st = findStation(start, &G1);

ed = findStation(end, &G1);

if (st == -1 || ed == -1) {

printf("不存在该站");

continue;

}

Dijkstra(&G1, st);

stack<int> s;

if (d[ed].w != INF) {

printf("最少转线次数为 %d次\n", d[ed].w - 1);

for (int i = parent[ed]; i != -1; i = parent[i]) {

s.push(i);

}

printf("路径为：");

while (!s.empty()) {

int temp = s.top();

s.pop();

printf("%s--->", G1.vexname[temp]);

}

printf("%s\n", G1.vexname[ed]);

}

else

printf("不存在最短线路");

}

else if (ch == 2) {

st = findStation(start, &G2);

ed = findStation(end, &G2);

if (st == -1 || ed == -1) {

printf("不存在该站");

continue;

}

Dijkstra(&G2, st);

stack<int> s;

if (d[ed].w != INF) {

printf("最少经过站点数为 %d个\n", d[ed].w + 1);

for (int i = parent[ed]; i != -1; i = parent[i]) {

s.push(i);

}

printf("路径为：");

while (!s.empty()) {

int temp = s.top();

s.pop();

printf("%s--->", G2.vexname[temp]);

}

printf("%s\n", G2.vexname[ed]);

}

else

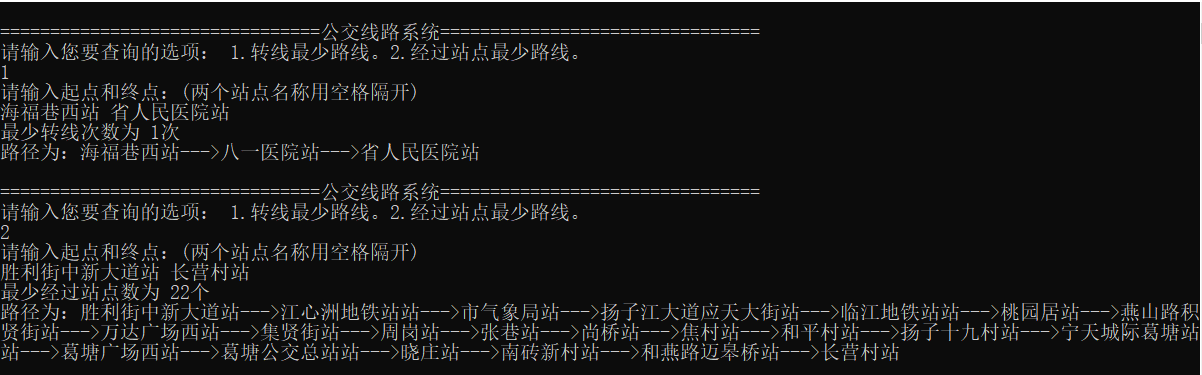
printf("不存在最短线路");

}

}

return 0;

}



目标4：设计思路、出现的问题及解决方法

设计思路

数据读取和构建：从文件读取数据，根据所需需求构建不同的图结构。

Dijkstra 算法：应用 Dijkstra 算法解决最短路径问题，并存储相关路径信息。

用户交互：通过命令行等方式与用户进行交互，接收用户输入查询的选项和站点名称，返回相应的结果。

出现的问题及解决方法

内存泄漏问题：代码中使用了动态分配内存，但没有在程序结束时释放分配的内存，可能存在内存泄漏问题。

逻辑问题：在图的构建过程中，边的连接可能存在逻辑错误，需要进行仔细的调试和验证。

用户输入验证：未对用户输入进行合法性校验，需要加入输入检查和异常处理。

数据结构和算法思想

数据结构：使用了结构体表示图的节点、边以及相关信息。

算法思想：基于 Dijkstra 算法思想，寻找最短路径，采用优先队列和栈等数据结构辅助求解。

测试数据和结果

测试数据：从文件中读取站点信息和路线信息，模拟不同的查询情况。

结果：输出经过验证的最短路径和站点信息。

时间复杂度

图的构建：图的构建时间复杂度可能受到节点和边的数量影响，为 O(V+E)。

Dijkstra 算法：最坏情况下的时间复杂度为 O(V^2)，使用优先队列可优化至 O((V+E)logV)。

可改进的地方

内存管理：增加对内存的合理释放，避免内存泄漏，提高程序的稳定性。

算法优化：对于 Dijkstra 算法，可以考虑使用堆优化或其他更高效的实现方式来提高算法效率。

用户输入验证：加入输入合法性校验，处理用户异常输入，提高程序的健壮性和稳定性。

# 7、B-树应用 （必做） （查找）

目标1: 数据结构类型和算法选择

针对实际案例中的B-树动态查找需求，我们选择了以下数据结构类型和算法：

数据结构类型：

B-树：用于实现动态查找。B-树是一种自平衡的树状数据结构，适用于大量数据的动态存储和查找。其节点可以容纳多个键和子节点，使得每个节点可以存储更多的数据，减少I/O操作。

算法类型：

插入算法：用于在B-树中插入新的关键字。实现了B-树的分裂和合并操作，确保树的平衡性。

查找算法：用于在B-树中查找特定关键字。通过逐层比较节点中的键值，确定要查找的关键字所在的位置。

删除算法：用于在B-树中删除特定关键字。实现了删除关键字并对树进行调整，保持B-树的平衡性。

目标2: 数据处理和算法设计

数据处理：

输入数据：从文件中读取关键字数据。

组织数据：将关键字按照B-树的规则插入到树中，并进行查找和删除操作。

结果处理：每次操作后，输出B-树的内容以验证操作的正确性。

算法设计：

插入算法：当节点关键字数目达到最大值时进行节点的分裂，并将新关键字插入到合适的位置。

查找算法：递归地在节点中比较关键字以找到要查找的关键字，同时判断是否在树中。

删除算法：删除关键字后，根据需要进行节点的合并操作，保持B-树的平衡性。

程序代码：

#include <iostream>

#include <stdlib.h>

#include <string.h>

#include <conio.h>

using namespace std;

#define TURE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASSIBLE -1

#define OVERFLOW -2

#define STACK\_INIT\_SIZE 100

#define STACKINCREMENT 10

#define m 3 //B-树的阶，暂设为3

#define n 17 //关键字个数，暂设为17

typedef int KeyType;

typedef int Status;

typedef struct BTNode

{

int keynum; //结点中关键字个数

struct BTNode \*parent; //指向双亲结点

KeyType key[m + 1]; //关键字向量，0号单位元未用

struct BTNode \*ptr[m + 1]; //子树指针向量

}BTNode, \*BTree; //B-树结点和B-树的类型

typedef struct

{

BTNode \*pt; //指向找到的结点

int i; //1...m,在结点中的关键字序号

int tag; //1:查找成功；0：查找失败

}Result; //B-树的查找结果类型

typedef struct QNode

{

BTree data;

struct QNode \*next;

}QNode, \*QueuePtr;

typedef struct

{

QueuePtr front;

QueuePtr rear;

}LinkQueue;

Status InitQueue(LinkQueue \*Q)

//创建一个队列

{

Q->front = Q->rear = (QueuePtr)malloc(sizeof(QNode));

if (!Q->front)exit(OVERFLOW);

Q->front->next = NULL;

return OK;

}

Status EnQueue(LinkQueue \*Q, BTree e)

//入队

{

QueuePtr p;

p = (QueuePtr)malloc(sizeof(QNode));

if (!p)

{

exit(OVERFLOW);

}

p->data = e;

p->next = NULL;

Q->rear->next = p;

Q->rear = p;

return OK;

}

Status DeQueue(LinkQueue \*Q, BTree \*e)

//出队

{

QueuePtr p;

if (Q->front == Q->rear)

{

return ERROR;

}

p = Q->front->next;

\*e = p->data;

Q->front->next = p->next;

if (Q->rear == p)

{

Q->rear = Q->front;

}

free(p);

return OK;

}

Status QueueEmpty(LinkQueue Q)

//判断是否队空

{

if (Q.front == Q.rear)

return OK;

else

return ERROR;

}

int Search(BTree p, KeyType K)

//在p->key[1...keynum]中查找i，使得p->key[i]<=k<p->key[i+1]

{

int i, j = 0;

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

{

if (p->key[i] <= K)

{

j = i;

}

}

return j;

}

Result SearchBTree(BTree T, KeyType K)

//在m阶B-树T上查找关键字K，返回结果（pt，i，tag）.

//若查找成功，则特征值tag=1，指针pt所指结点中第i个关键字等于K；否则特征值tag=0，等于K的关键字应插入在指针pt所指结点中第i和第i+1个关键字之间

{

BTree p = T, q = NULL;

int found = FALSE;

Result r;

int i = 0;

while (p && (!found))

{

i = Search(p, K); //在p->key[1..keynum]中查找i使得：p->key[i]<=k<p->key[i+1]

if ((i > 0) && (p->key[i] == K))

{

found = TURE;

}

else

{

q = p;

p = p->ptr[i];

}

}

r.i = i;

if (found)

//查找成功

{

r.pt = p;

r.tag = 1;

}

else

//查找不成功，返回K的插入位置信息

{

r.pt = q;

r.tag = 0;

}

return r;

}

Status NewRoot(BTree &T,KeyType x,BTree ap)

//生成含信息（T，x，ap）的新的根结点\*T，原T和ap为子树指针

{

BTree p;

p = (BTree)malloc(sizeof(BTNode));

p->ptr[0] = T;

T = p;

if (T->ptr[0])

{

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

}

T->parent = NULL;

T->keynum = 1;

T->key[1] = x;

T->ptr[1] = ap;

if (T->ptr[1])

{

T->ptr[1]->parent = T;

}

return OK;

}

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

//将x和ap分别插入到q->key[i+1]和q->ptr[i+1]

{

int j;

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

{

q->key[j + 1] = q->key[j];

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

}

q->key[i + 1] = x;

q->ptr[i + 1] = ap;

q->keynum++;

return OK;

}

Status split(BTree &q,int s, BTree &ap)

//将q->key[s+1..m]和q->ptr[s..m]移入新结点\*ap

{

int i, j;

ap = (BTree)malloc(sizeof(BTNode)); //生成新的结点

ap->ptr[0] = q->ptr[s]; //原来结点中间位置关键字相应指针指向的子树放到新生成结点的0棵子树中去

for (i = s + 1, j = 1; i <= q->keynum; i++,j++)

{

ap->key[j] = q->key[i];

ap->ptr[j] = q->ptr[i];

}

ap->keynum = q->keynum - s;

ap->parent = q->parent;

for (i = 0; i <= q->keynum - s;i++)

{

if (ap->ptr[i])

{

ap->ptr[i]->parent = ap;

}

}

q->key[q->keynum] = 0;

q->keynum = s - 1;

return OK;

}

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

//在m阶B-树T上\*q的key[i]与key[i+1]之间插入关键字K

//若结点中已无多余位置，则对结点进行分裂处理

{

KeyType x = K;

int finished = FALSE;

BTree ap = NULL;

int s;

while ( q && !finished )

{

Insert(q, i, x, ap); //将x和ap分别插入到q->key[i+1]和q->ptr[i+1]

if (q->keynum < m) //插入完成

finished = TURE;

else

//节点无多余位置，需要分裂 //分裂结点\*q

{

//将q->key[s+1..m]和q->ptr[s..m]移入新结点\*ap

s = (m + 1) / 2;

split(q, s, ap);

x = q->key[s];

q = q->parent;

if (q)

{

i = Search(q, x); //在双亲结点\*q中查找x的插入位置

}

}

}

if (!finished) //T是空树（参数q初值为NULL）或者根结点已分裂为结点\*q和\*ap

NewRoot(T, x, ap); //生成新的根结点\*T，原T和ap为子树指针

return OK;

}

static Status MergeBTree(BTree &T, BTree node);

static Status Merge\_BTree(BTree &T, BTree left, BTree right, int mid)

//递归调用MergeBTree来合并结点

{

int x = 0;

BTree parent = left->parent;

//合并到左兄弟结点

left->key[++left->keynum] = parent->key[mid];

memcpy(left->key + left->keynum + 1, right->key + 1, right->keynum\*sizeof(int));

memcpy(left->ptr + left->keynum, right->ptr, (right->keynum + 1)\*sizeof(BTree));

for (x = 0; x <= right->keynum;x++)

{

if (right->ptr[x] != NULL)

{

right->ptr[x]->parent = left;

}

}

left->keynum += right->keynum;

for (x = mid; x < parent->keynum ; x++)

{

parent->key[x] = parent->key[x + 1];

parent->ptr[x] = parent->ptr[x + 1];

}

parent->key[x + 1] = 0;

parent->ptr[x + 1] = NULL;

parent->keynum--;

free(right);

if (parent->keynum <= (m / 2 - 1))

{

return MergeBTree(T, parent);

}

return OK;

}

static Status MergeBTree(BTree &T, BTree node)

//删除某关键字后结点中无数据，对其进行结点合并调整

{

int i = 0, mid = 0;

BTree parent = node->parent, right = NULL, left = NULL;

//node是根结点, 不必进行合并处理

if (parent == NULL)

{

if (node->keynum == 0)

{

if (node->ptr[0] != NULL)

{

T = node->ptr[0];

node->ptr[0]->parent = NULL;

}

else

{

T = NULL;

}

free(node);

}

return OK;

}

//查找node为parent的第几个结点

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

{

//printf("%d,%d ", i, parent->ptr[0]->key[i]);

if (parent->ptr[i] == node)break;

}

if (i > parent->keynum)

{

return ERROR;

}

else if (i == parent->keynum)

{

mid = i;

left = parent->ptr[mid - 1];

//合并结点，当两个节点有一个不存在关键字时

if ((node->keynum + left->keynum) < (m - 1))

{

return Merge\_BTree(T, left, node, mid);

}

//借用结点left->key[keynum]

for (int x = node->keynum; x>1; x--)

{

node->key[x] = node->key[x - 1];

node->ptr[x] = node->ptr[x - 1];

}

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

node->key[1] = parent->key[mid];

node->keynum++;

node->ptr[0] = left->ptr[left->keynum];

if (left->ptr[left->keynum] != NULL)

{

left->ptr[left->keynum]->parent = node;

}

parent->key[mid] = left->key[left->keynum];

left->key[left->keynum] = 0;

left->ptr[left->keynum] = NULL;

left->keynum--;

return OK;

}

mid = i + 1;

right = parent->ptr[mid];

//合并结点

if ((node->keynum + right->keynum) < (m - 1))

{

return Merge\_BTree(T, node, right, mid);

}

//借用结点right->ptr[1]

node->key[++node->keynum] = parent->key[mid];

node->ptr[node->keynum] = right->ptr[0];

if (right->ptr[0] != NULL)

{

right->ptr[0]->parent = node;

}

parent->key[mid] = right->key[1];

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

for (int x = 1; x < right->keynum; x++)

{

right->key[x] = right->key[x + 1];

right->ptr[x] = right->ptr[x + 1];

}

right->key[right->keynum] = 0;

right->ptr[right->keynum--] = NULL;

return OK;

}

static Status DeleteBTree(BTree &T, BTree node, int i)

//在m阶B-树T上找到并删除关键字K

//若删除后结点中无数据，则沿双亲链进行结点合并调整

{

BTree child = node->ptr[i - 1], orig = node;

//使用node->ptr[i-1]中最大值替代被删除的关键字

while (child != NULL)

{

node = child;

child = node->ptr[child->keynum];

}

orig->key[i] = node->key[node->keynum];

//处理最底层被删除的关键字

node->key[node->keynum--] = 0;

//若删除后结点中无数据，则沿双亲链进行结点合并调整

if (node->keynum <= (m / 2 - 1))

{

MergeBTree(T, node);

}

return OK;

}

Status TraverseBTree(BTree T, LinkQueue Q, int newline, int sum)

//按关键字的顺序对DT的每个结点调用函数Visit()一次且至多一次(层次遍历)

{

BTree p;

int i;

if (T)

{

printf("[ ");

EnQueue(&Q, T->ptr[0]);

for (i = 1; i <= T->keynum; i++)

{

printf("%d ", T->key[i]);

EnQueue(&Q, T->ptr[i]);

}

sum += T->keynum + 1;

printf("]");

if (newline == 0)

{

printf("\n");

newline = sum - 1;

sum = 0;

}

else

--newline;

if (!QueueEmpty(Q))

{

DeQueue(&Q, &p);

TraverseBTree(p, Q, newline, sum);

}

}

return OK;

}

void print(BTree T)

{

LinkQueue Q;

if (!T)

{

printf("[ ]\n");

return ;

}

InitQueue(&Q);

TraverseBTree(T, Q, 0, 0);

return;

}

void InputKey(BTree &T, KeyType K[])

{

Result s;

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

{

s = SearchBTree(T, K[i]);

if (!s.tag)

{

InsertBTree(T, K[i], s.pt, s.i);

system("cls");

printf("待插入关键字有%d个:\n", n - i -1);

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

printf("%d ", K[j]);

printf("\n");

printf("\n");

printf("插入关键字%d:", K[i]);

printf("\n");

print(T);

\_getch();

}

}

}

Status DeleteKey(BTree &T, KeyType K)

{

int i = 0;

BTree node = T;

Result s;

while (NULL != node)

{

s = SearchBTree(T, K);

node = s.pt;

i = s.i;

if (K == node->key[i])

return DeleteBTree(T, node, i);

else if(K < node->key[i])

break;

}

return OK;

}

int main()

{

BTree T = NULL;

Result s;

FILE \*fp;

fp = fopen("BTree.txt", "r");

if(!fp)

{

cout << "打开文件失败!";

exit(0);

}

KeyType key[n];

for(int i = 0; !feof(fp); i++)

fscanf(fp, "%d", &key[i]);

fclose(fp);

InputKey(T, key); //生成B-树

KeyType k[4] = { 12, 80, 5, 37 };

for (int i = 0; i < 4; i++) //查找关键字

{

system("cls");

printf("%d阶B-树为:", m);

printf("\n");

print(T);

printf("\n");

printf("查找关键字%d:", k[i]);

getchar();

s = SearchBTree(T, k[i]);

if (s.tag)

{

printf("查找成功!\n");

}

else

{

printf("查找失败!\n");

}

getchar();

}

for (int i = 0; i < n; i++) //删除B-树

{

system("cls");

printf("%d阶B-树为:", m);

printf("\n");

print(T);

printf("\n");

printf("删除关键字%d\n",key[i]);

DeleteKey(T, key[i]);

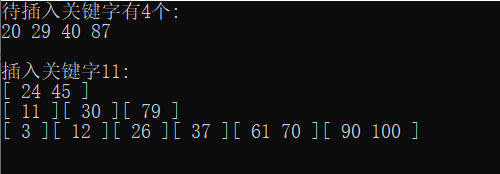
print(T);

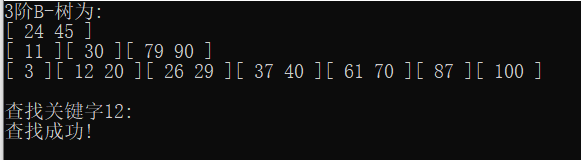
\_getch();

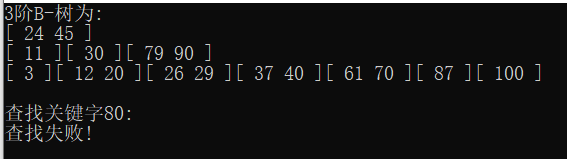
}

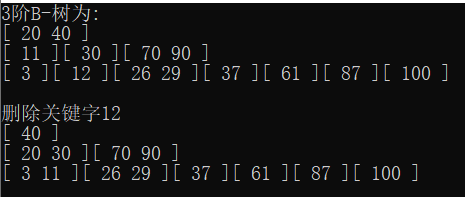
return 0;

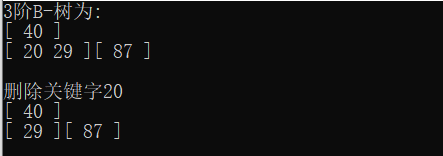
}

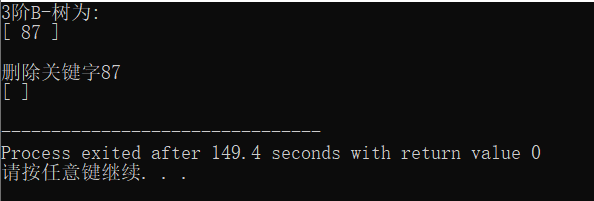












目标4: 设计思路、问题与解决方法

设计思路：

实现B-树的基本结构和操作函数。

从文件中读取关键字数据构建B-树，并进行插入、查找和删除操作。

每次操作后输出B-树的内容，验证操作的正确性。

出现的问题及解决方法：

节点分裂和合并：在节点操作中需要确保分裂和合并的正确性，以维持树的平衡。解决方法是在插入和删除操作时适时地对节点进行分裂和合并。

数据验证：为了确保每次操作的正确性，需要输出B-树的内容进行验证，发现问题及时调试和修复。

时间复杂度

插入、查找、删除操作的时间复杂度为O(log n)，其中n为B-树中的关键字数量。

节点分裂和合并的时间复杂度为O(m)，其中m为B-树的阶数。

可改进的地方

性能优化：可进一步优化节点的分裂和合并策略，以提高B-树的性能。

错误处理：增加更多的错误处理机制，提高程序的健壮性和稳定性。

# 8、排序算法比较 （必做）（排序）

结构设计

该程序设置了 2 维数组 a[10][50000] 来存储每组样本，通过 rand() 函数随机生成样本，并依次实现了 8 种排序算法的比较。

算法构思

该程序一共实现了以下 8 种排序算法的比较：

• 冒泡排序： 采用两个 for 循环，每次将第 i 个和后续 n-i-1 个晾凉交换，可以优化为，当未发生交换时说明排序完成，因此正样本时间为 0。

• 选择排序： 依然采用两个 for 循环，每次在第 i+1 个到最后一个中选择一个最小的放到表头实现升序排列。

• 归并排序： 采用分治思想，二分法递归分裂数组一直到只有两个，对两个数进行升序排列，最后再合并。（原谅我的电脑归并只能跑 25000 个样本即 5 组样本，不然就爆栈）

• 快速排序： 首先找到一个基准使得序列中基准前的都比它小，基准后的都比它大，然后将基准左边和右边的数组分开递归再次寻找新的基准，一直到 left 指针与right 指针相等。

• 插入排序： 每次选择一个数，将其插入到升序序列中应在的位置上。

• 希尔排序： 插入排序的升级版，不再遍历数组进行插入，从一个较大的步长比如 5开始遍历数组，将选中的数放入升序数组中（其实可以插入优先队列, 偷懒），然后再逐步缩小增量至 1 即可完成排序。

• 堆排序： 首先实现大顶堆的构建，对一个无序序列，从 n/2 的位置开始，也就是第一个非叶子结点开始向前递归构建大顶堆。然后将堆顶和末尾元素换位，每次将末尾元素前移一个，并构建新的大顶堆，重复 n 次即可。

基数排序： 首先根据个位数的大小排序，再根据十位数的大小排序，再根据百位数的大小排序。（假设最大是百位数）

代码：

#include <iostream>

#include <algorithm>

#include <cstring>

#include <cstdlib>

#include <time.h>

using namespace std;

#define MAXN 0x3f3f3f //定义数的最大值

#define col 50000

int arr[10][col]; //定义二维指针数组

void Random()

{

srand(time(0)); //设置时间种子

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

{

arr[0][k] = rand()%MAXN+1;

}

//memset(arr[0],rand()%MAXN+1,50000);

for(int i=1;i<10;++i)

{

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

{

arr[i][j] = arr[0][j];

}

}

}

void swap(int &a,int &b)

{

a = a+b;

b = a-b;

a = a-b;

}

void Bubble(int c)

{

if(c<0 || c>10)

{

return;

}

if(col<=0)

{

return;

}

int \*a = arr[c];

int len = col;

bool flag;

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

{

flag = false;

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

{

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

{

swap(a[j],a[j+1]);

flag = true;

}

}

if(!flag) break;

}

/\*

cout<<"排序后的数组为："<<endl;

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

{

cout<<arr[c][i]<<" ";

}

\*/

}

void Select(int c)

{

if(c<0 || c>10)

{

return;

}

if(col<=0)

{

return;

}

int index;

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

{

index = i;

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

{

if(arr[c][index]>arr[c][j])

{

index = j;

}

}

if(index != i)//不加这个会导致空间丢失

{

swap(arr[c][index],arr[c][i]);

}

}

/\*

cout<<"排序后的数组为："<<endl;

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

{

cout<<arr[c][i]<<" ";

}

\*/

}

int merge(int r[],int s[],int left,int mid,int right)

//s是你要存储的数组，而r是临时数组

{

int i,j,k;

i = left;

j = mid+1;

k = left;

while(i<=mid && j<=right)

//将r的两个数组按大小归并

{

if(r[i]<=r[j])

{

s[k] = r[i];

i++;

k++;

}

else

{

s[k] = r[j];

j++;

k++;

}

}

while(i<=mid)

{

s[k++] = r[i++];

}

while(j<=right)

{

s[k++] = r[j++];

}

return 0;

}

int merge\_sort(int r[],int s[],int left,int right)

{

int mid;

int\* t = new int[right+1];

//临时数组

if(left==right)

{

s[left] = r[right];

}

else

{

mid = (left+right)/2;

merge\_sort(r,t,left,mid);

merge\_sort(r,t,mid+1,right);

merge(t,s,left,mid,right);

}

return 0;

}

void MMerge(int c)

{

if(col<=0)

{

return;

}

if(c<0 || c>10)

{

return;

}

int len = col;

int \*a = arr[c];

merge\_sort(a,a,0,len-1);

}

void Quick\_sort(int \*arr,int low,int high);

void Quick(int c)

{

if(col<=0)

{

return;

}

if(c<0 || c>10)

{

return;

}

int \*a = arr[c];

Quick\_sort(a,0,col-1);

}

int partition(int \*arr ,int l,int r)

// 编程实现arr[l, r]分区：选定一个基准，左边比基准小，右边比基准大

// 返回基准所处位置

{

int pivot = arr[l];

while(l<r)

{

while(l<r && arr[r]>=pivot)

{

--r;

}

if(l<r)

{

arr[l] = arr[r];

}

while(l<r && arr[l]<=pivot)

{

++l;

}

if(l<r)

{

arr[r] = arr[l];

}

}

arr[l] = pivot;

return l;

}

void Quick\_sort(int \*arr,int low,int high)

{

if(low<high)

{

int mid = partition(arr,low,high);

Quick\_sort(arr,low,mid);

Quick\_sort(arr,mid+1,high);

}

}

void adjustHeap(int \*arr, int root, int n)

// 编程实现堆的调整

{

int left = 2\*root+1;

if(left < n)

{

int flag = left;

int right = left+1;

if(right<n)

{

if(arr[right]>arr[flag])

{

flag = right;

}

}

if(arr[root] < arr[flag])

{

swap(arr[root],arr[flag]);

adjustHeap(arr,flag,n);

}

}

}

int\* Heap\_sort(int \*arr, int len)

// 基于adjustHeap函数编程实现堆排序

{

for(int i=len/2;i>=0;--i)

{

adjustHeap(arr,i,len);

}

for(int j=len-1;j>0;--j)

{

swap(arr[0],arr[j]);

adjustHeap(arr,0,j);

}

return arr;

}

void Heap(int c)

{

if(col<=0)

{

return;

}

if(c<0 || c>10)

{

return;

}

int \*a = arr[c];

a = Heap\_sort(a,col);

/\*

cout<<"排序后的数组为："<<endl;

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

{

cout<<arr[c][i]<<" ";

}

\*/

}

void Heal\_sort(int \*arr, int n)

{

int step[3]={5,2,1};//希尔排序的步长

int s=0;

while(s<3)

{

for(int i=step[s];i<n;++i)

{

int key = \*(arr+i);

int j = i-step[s];

while(j>=0 && key<\*(arr+j))

{

\*(arr+j+step[s]) = \*(arr+j);

j-=step[s];

}

\*(arr+j+step[s]) = key;

}

s++;

}

}

void Heal(int c)

{

if(col<=0)

{

return;

}

if(c<0 || c>10)

{

return;

}

int \*a = arr[c];

Heal\_sort(a,col);

/\*

cout<<"排序后的数组为："<<endl;

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

{

cout<<arr[c][i]<<" ";

}

\*/

}

void Insert(int c)

{

if(col<=0)

{

return;

}

if(c<0 || c>10)

{

return;

}

int \*a = arr[c];

int len = col;

int temp;

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

{

int key = a[i];

int j = i-1;

while(j>=0 && a[j]>key)

{

a[j+1] = a[j];

--j;

}

a[j+1] = key;

}

/\*

cout<<"排序后的数组为："<<endl;

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

{

cout<<arr[c][i]<<" ";

}

\*/

}

int\* jishu\_sort(int \*arr, int n)

{

int max = \*max\_element(arr,arr+n);

int \*radix = new int[n\*10];

fill(radix,radix+n\*10,0);

int \*tmparr = new int[n];

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

{

tmparr[i] = arr[i];

}

int jishu[10];

fill(jishu,jishu+10,-1);

int wei = 0;

while(max>0)

{

max /= 10;

wei ++;

}

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

{

int dev = 1;

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

{

dev \*= 10;

}

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

{

int index = arr[j]/(dev)%(10);

//tmparr[j] = tmparr[j]/(10);

jishu[index]++;

radix[index\*n + jishu[index]] = arr[j];

}

int ind = 0;

for(int k=0;k<10;k++)

{

if(jishu[k]>=0)

{

sort(radix + k\*n,radix + k\*n + jishu[k]+1);

for(int j=0;j<jishu[k]+1;j++)

{

arr[ind] = radix[k\*n+j];

ind ++;

}

jishu[k] = -1;

}

}

fill(radix,radix+n\*10,0);

}

return arr;

}

void jishu(int c)

{

if(col<=0)

{

return;

}

if(c<0 || c>10)

{

return;

}

int \*a = arr[c];

int len = col;

a = jishu\_sort(a,len);

/\*

cout<<"排序后的数组为："<<endl;

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

{

cout<<arr[c][i]<<" ";

}

\*/

}

int main()

{

double cost\_time[10] = {0};

while(1)

{

cout<<"该程序支持的排序方法(第一组样本为已排好样本，第二组样本为倒序样本):"<<endl;

cout<<"1.冒泡排序"<<endl;

cout<<"2.选择排序"<<endl;

cout<<"3.归并排序"<<endl;

cout<<"4.快速排序"<<endl;

cout<<"5.插入排序"<<endl;

cout<<"6.希尔排序"<<endl;

cout<<"7.堆排序"<<endl;

cout<<"8.基数排序"<<endl;

cout<<"9.输出所有排序的平均耗时（未计时为0）"<<endl;

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

cout<<"请输入你想要使用的排序方法:"<<endl;

int c;cin>>c;

time\_t begin,end;

double result;

double total;

switch(c)

{

case 1:

total = 0;

Random();

sort(arr[0],arr[0]+col,less<int>());

sort(arr[1],arr[1]+col,greater<int>());

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

{

begin = clock();

Bubble(i);

end = clock();

result = double(end -begin)\*1000/CLOCKS\_PER\_SEC;

total += result;

cout<<"第"<<i+1<<"组样本的代码运行时间为:"<<result<<endl;

}

cost\_time[c] = total/10;

system("pause");

system("cls");

break;

case 2:

total = 0;

Random();

sort(arr[0],arr[0]+col,less<int>());

sort(arr[1],arr[1]+col,greater<int>());

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

{

begin = clock();

Select(i);

end = clock();

result = double(end -begin)\*1000/CLOCKS\_PER\_SEC;

total += result;

cout<<"第"<<i+1<<"组样本的代码运行时间为:"<<result<<endl;

}

cost\_time[c] = total/10;

system("pause");

system("cls");

break;

case 3:

total = 0;

Random();

sort(arr[0],arr[0]+col,less<int>());

sort(arr[1],arr[1]+col,greater<int>());

// for(int i=0;i<5;++i)

// {

// begin = clock();

// MMerge(i);

// end = clock();

// result = double(end -begin)\*1000/CLOCKS\_PER\_SEC;

// total += result;

// cost\_time[c] = total/10;

// cout<<"第"<<i+1<<"组样本的代码运行时间为:"<<result<<endl;

// }

// system("pause");

for(int i=5;i<10;++i)

{

begin = clock();

MMerge(i);

end = clock();

result = double(end -begin)\*1000/CLOCKS\_PER\_SEC;

total += result;

cout<<"第"<<i+1<<"组样本的代码运行时间为:"<<result<<endl;

}

cost\_time[c] = total/5;

system("pause");

system("cls");

break;

case 4:

total = 0;

Random();

sort(arr[0],arr[0]+col,less<int>());

sort(arr[1],arr[1]+col,greater<int>());

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

{

begin = clock();

//Quick(i);//会爆栈

sort(arr[i],arr[i]+col,less<int>());

end = clock();

result = double(end -begin)\*1000/CLOCKS\_PER\_SEC;

total += result;

cout<<"第"<<i+1<<"组样本的代码运行时间为:"<<result<<endl;

}

cost\_time[c] = total/10;

system("pause");

system("cls");

break;

case 7:

total = 0;

Random();

sort(arr[0],arr[0]+col,less<int>());

sort(arr[1],arr[1]+col,greater<int>());

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

{

begin = clock();

Heap(i);

end = clock();

result = double(end -begin)\*1000/CLOCKS\_PER\_SEC;

total += result;

cout<<"第"<<i+1<<"组样本的代码运行时间为:"<<result<<endl;

}

cost\_time[c] = total/10;

system("pause");

system("cls");

break;

case 6:

total = 0;

Random();

sort(arr[0],arr[0]+col,less<int>());

sort(arr[1],arr[1]+col,greater<int>());

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

{

begin = clock();

Heal(i);

end = clock();

result = double(end -begin)\*1000/CLOCKS\_PER\_SEC;

total += result;

cout<<"第"<<i+1<<"组样本的代码运行时间为:"<<result<<endl;

}

cost\_time[c] = total/10;

system("pause");

system("cls");

break;

case 5:

total = 0;

Random();

sort(arr[0],arr[0]+col,less<int>());

sort(arr[1],arr[1]+col,greater<int>());

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

{

begin = clock();

Insert(i);

end = clock();

result = double(end -begin)\*1000/CLOCKS\_PER\_SEC;

total += result;

cout<<"第"<<i+1<<"组样本的代码运行时间为:"<<result<<endl;

}

cost\_time[c] = total/10;

system("pause");

system("cls");

break;

case 8:

total = 0;

Random();

sort(arr[0],arr[0]+col,less<int>());

sort(arr[1],arr[1]+col,greater<int>());

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

{

begin = clock();

jishu(i);

end = clock();

result = double(end -begin)\*1000/CLOCKS\_PER\_SEC;

total += result;

cout<<"第"<<i+1<<"组样本的代码运行时间为:"<<result<<endl;

}

cost\_time[c] = total/10;

system("pause");

system("cls");

break;

case 9:

cout<<"1.冒泡排序的运行时间为:"<<cost\_time[1]<<"ms"<<endl;

cout<<"2.选择排序的运行时间为:"<<cost\_time[2]<<"ms"<<endl;

cout<<"3.归并排序的运行时间为:"<<cost\_time[3]<<"ms"<<endl;

cout<<"4.快速排序的运行时间为:"<<cost\_time[4]<<"ms"<<endl;

cout<<"5.插入排序的运行时间为:"<<cost\_time[5]<<"ms"<<endl;

cout<<"6.希尔排序的运行时间为:"<<cost\_time[6]<<"ms"<<endl;

cout<<"7.堆排序的运行时间为:"<<cost\_time[7]<<"ms"<<endl;

cout<<"8.基数排序的运行时间为:"<<cost\_time[8]<<"ms"<<endl;

system("pause");

system("cls");

break;

default:

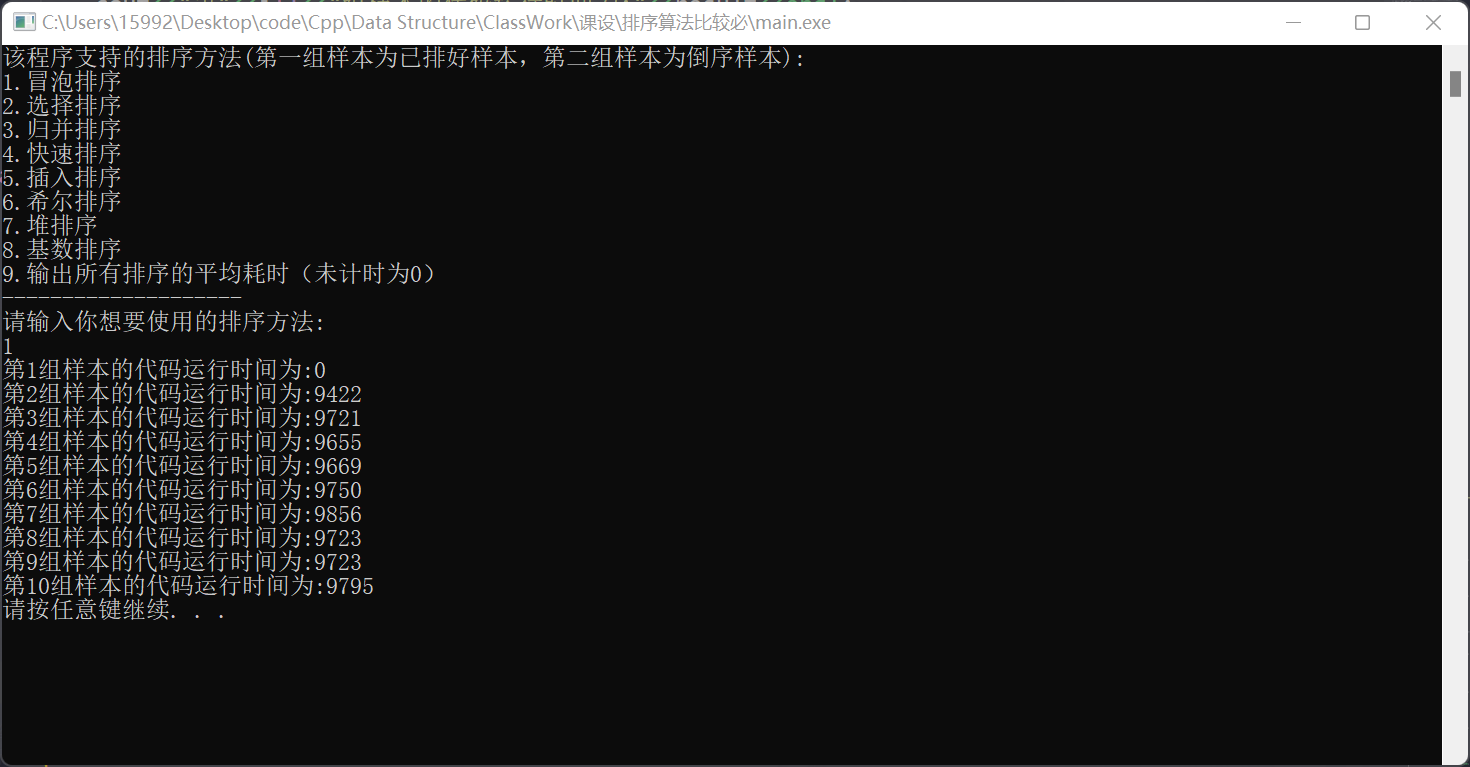
exit(0);

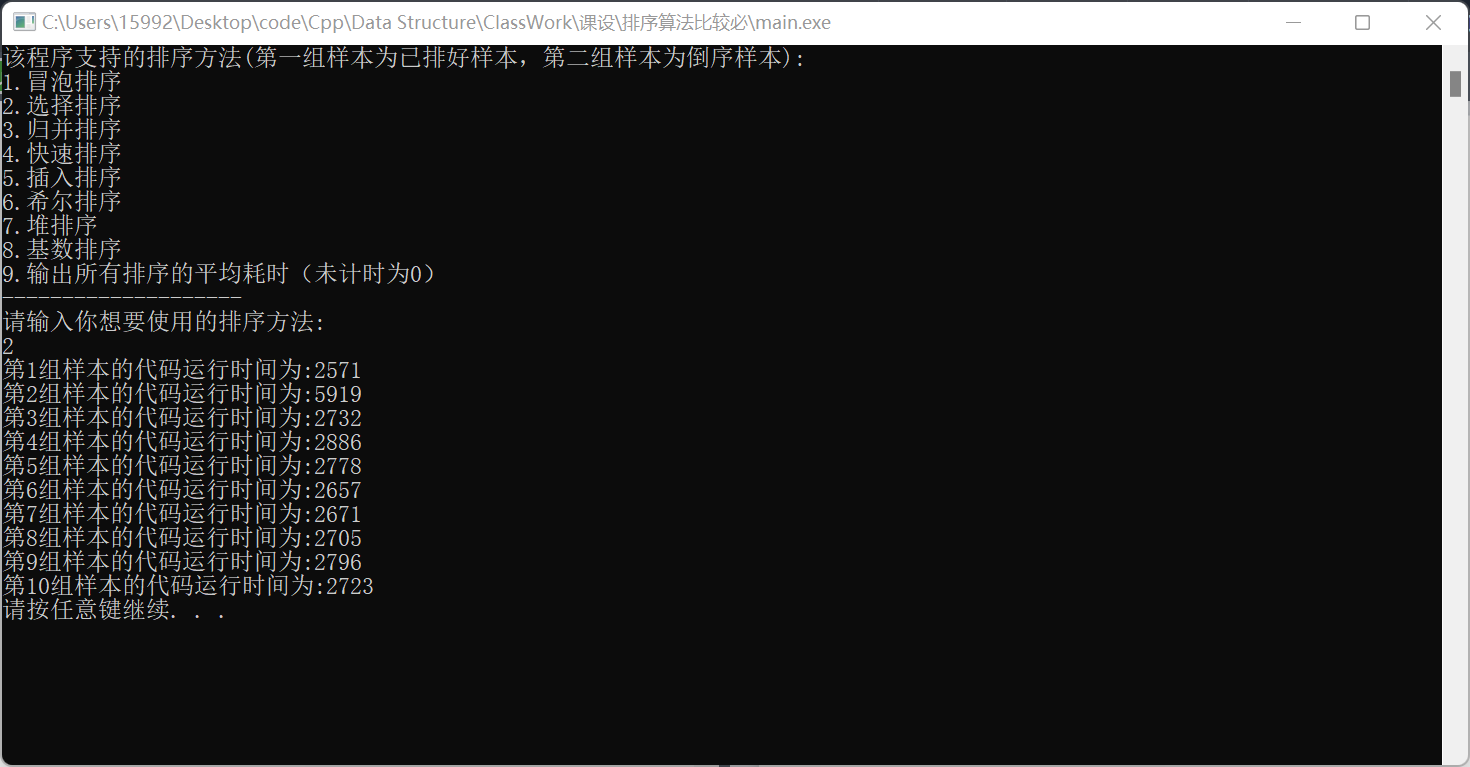
}

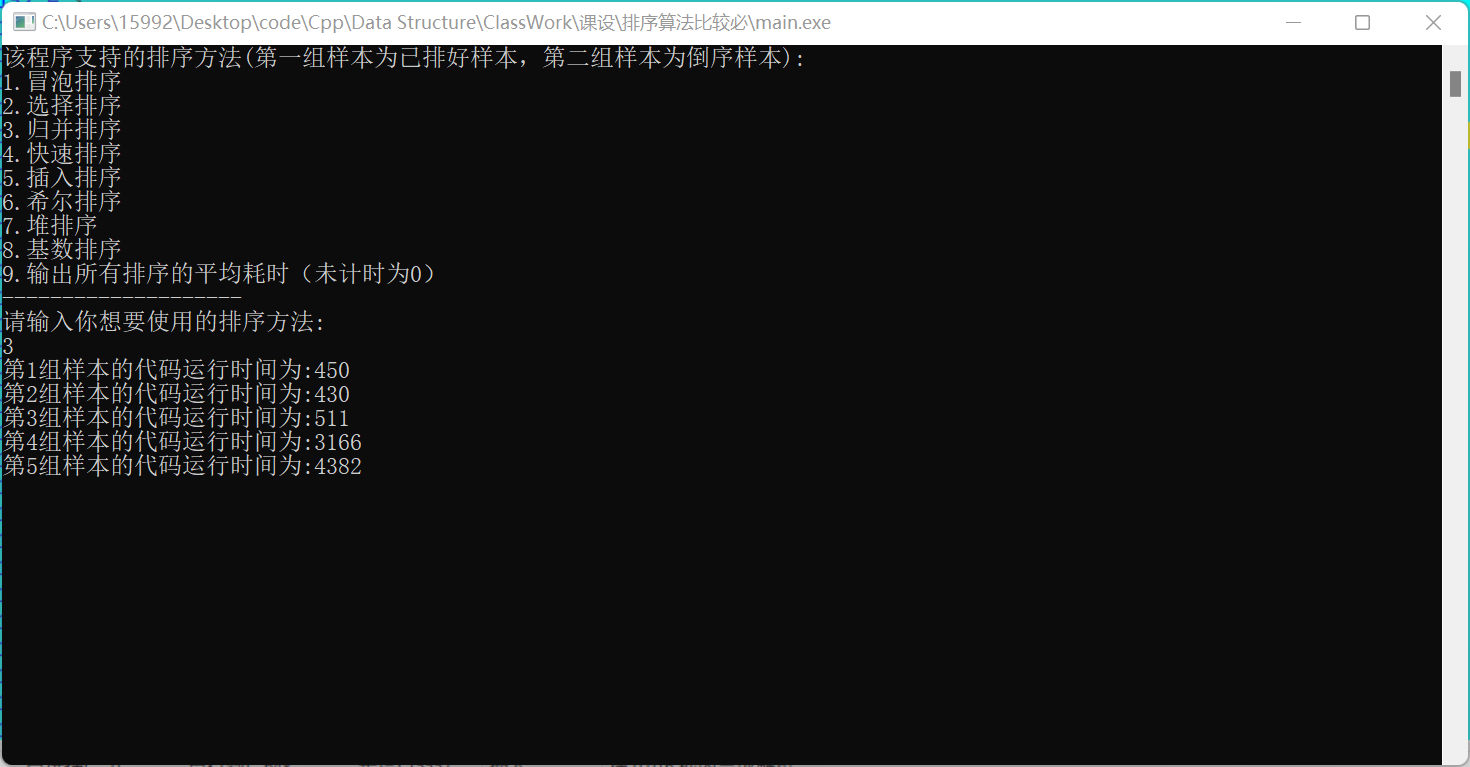
}

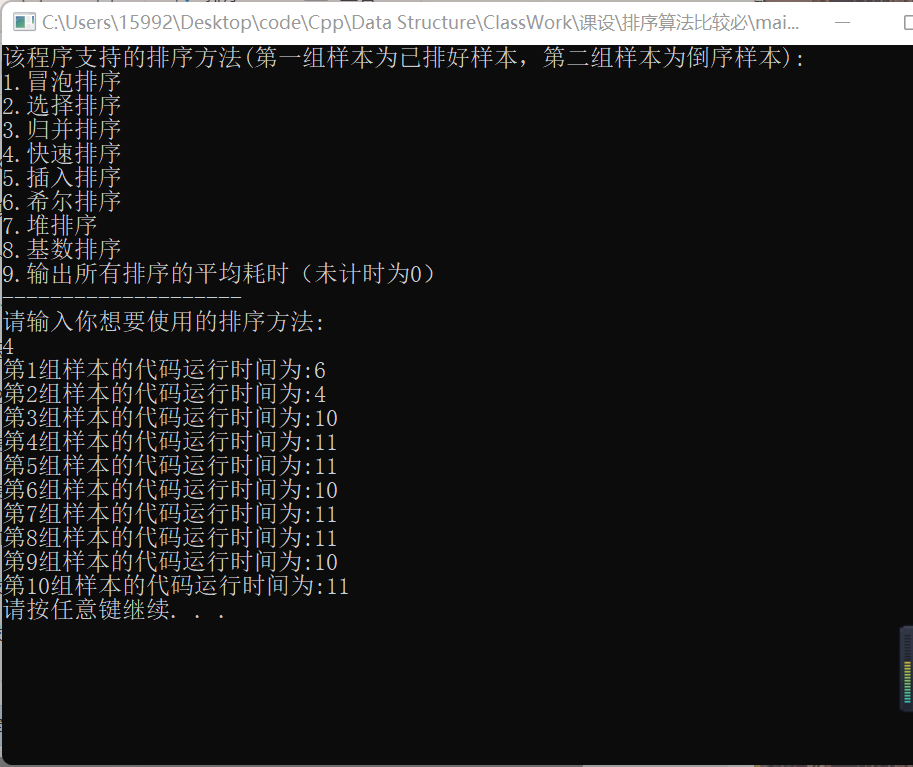
return 0;

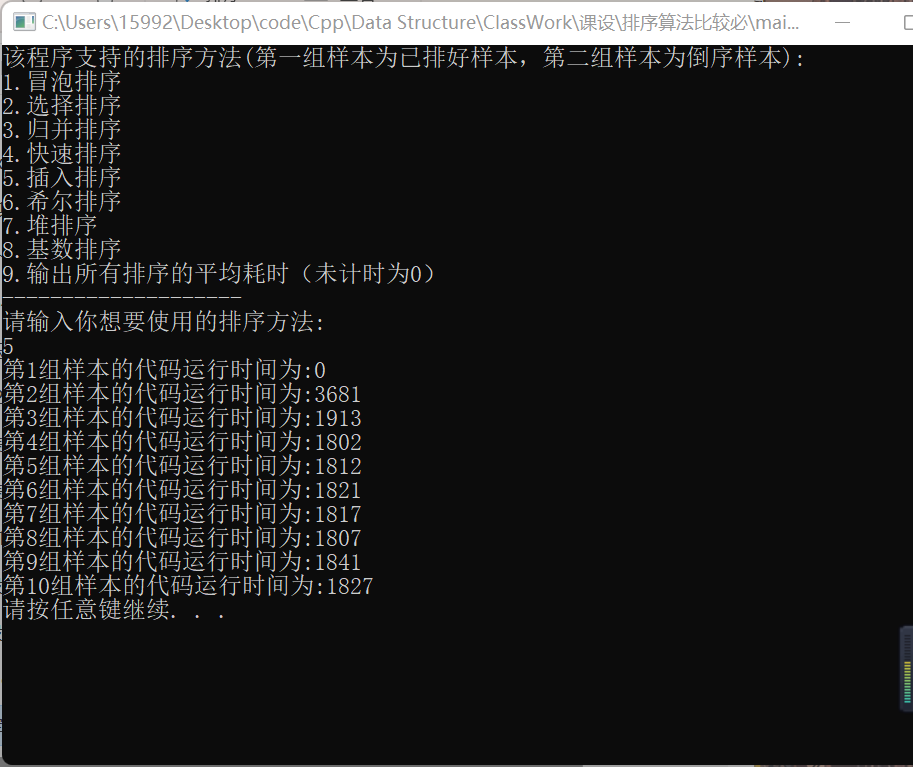
}

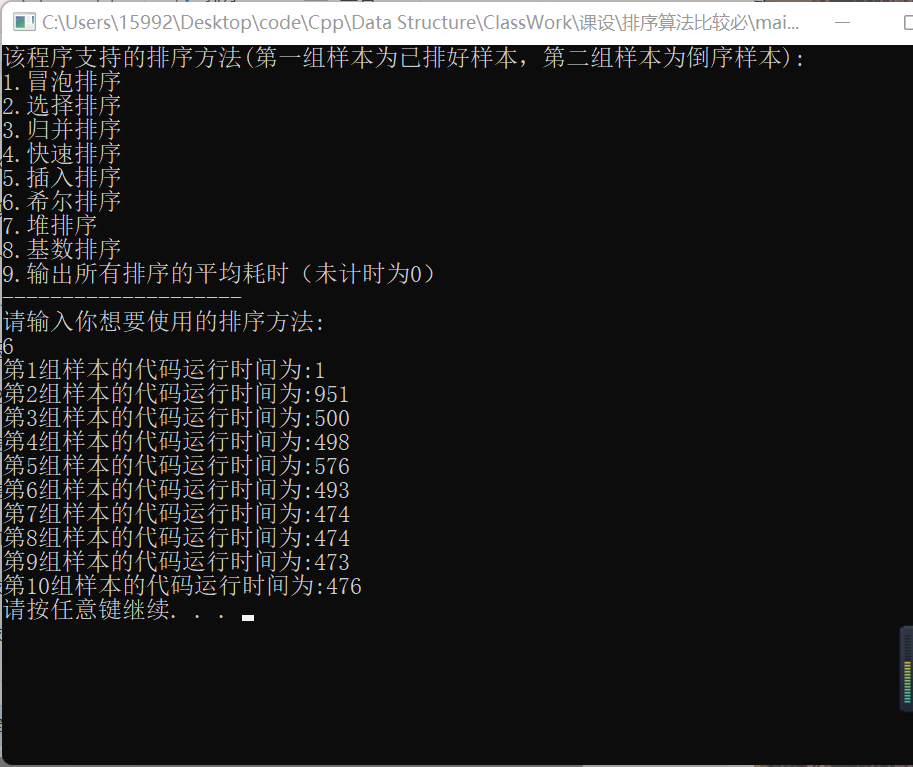


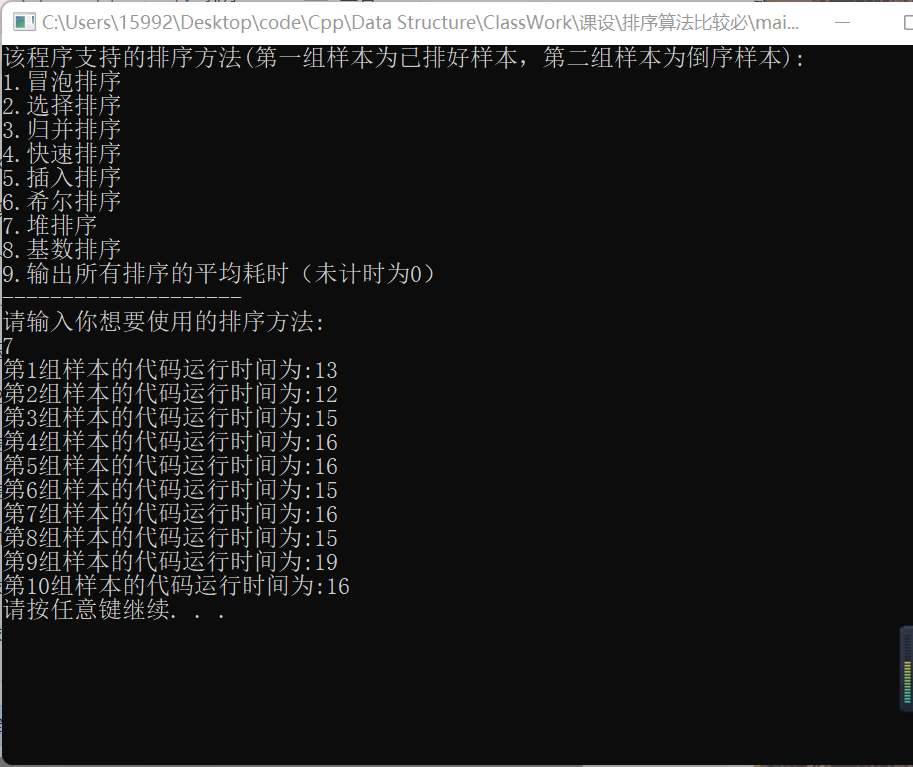
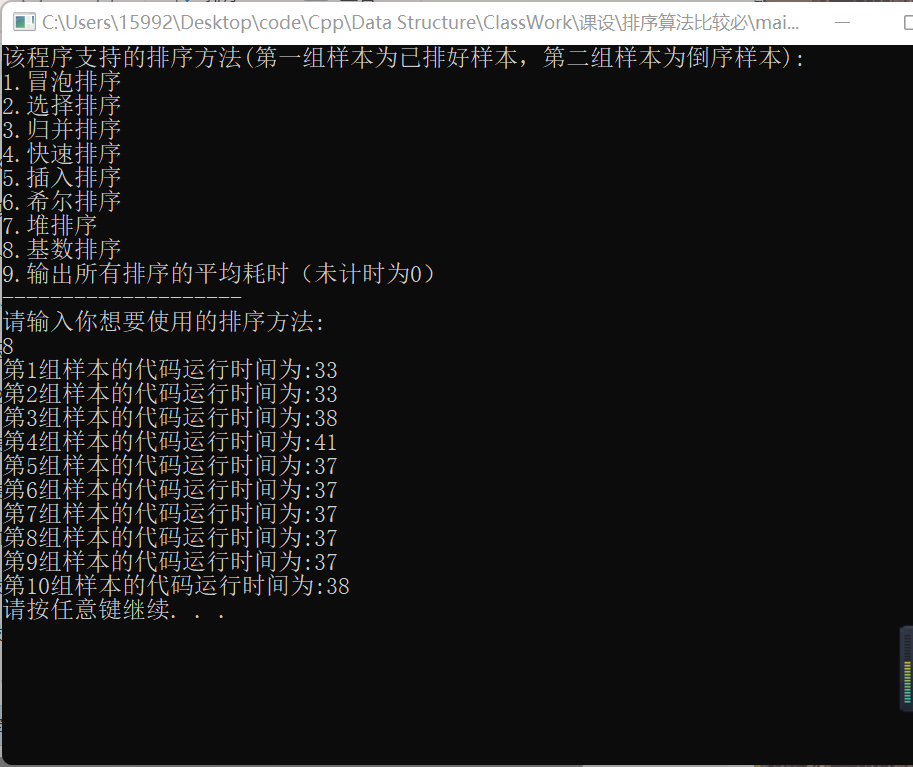










算法复杂度：冒泡、选择排序是 O(n2)，插入排序目前也是 O(n2)，但可以通过二分查找的形式优化为 O(nlogn)，基数排序的复杂度是 O(n+e)，其它算法的复杂度都是 O(nlogn)

# 14、【2】公共钥匙盒（选做）（线性表，栈，队列）

目标1: 数据结构类型和算法类型的分析和选择

数据结构类型：

使用了结构体 Key 来存储钥匙的信息，包括钥匙编号和时间。

使用了队列 (std::queue) 来存储借还钥匙的事件，并按时间顺序处理。

算法类型：

对借出和归还的钥匙按时间进行排序。

通过模拟事件流程，在模拟的时间流逝中处理钥匙的借还。

目标2: 组织、存储和处理数据的算法分析

读取输入文件中的数据，按时间对借出和归还的钥匙进行排序，存储在 borrow 和 ret 数组中。

使用队列 q 存储按时间排序的借还事件，通过模拟时间处理事件并更新钥匙的状态。

程序代码：

#include<cstdio>

#include<cstdlib>

#include<windows.h>

#include<iostream>

#include<queue>

//想写一个计时器

using namespace std;

int timecount=0;

typedef struct {

int keyId;

int time;

}Key;

// 全局变量

FILE \*fp;

char fileName[] = "input.txt";

// 打开文件

void fileOpen(char \*method) {

if((fp=fopen(fileName,method)) == NULL) {

printf("file cannot be opened\n");

return;

}

//else printf("file has been opened\n");

}

// 关闭文件

void fileClose() {

if(fclose(fp)!=0) printf("file cannot be closed\n");

//else printf("file has been closed\n");

return;

}

// 显示图形界面

void ShowBox(int n, int \*box, int time, int \*key) {

system("cls");

cout<<"正在盒子里的钥匙"<<endl;

cout<<" ";

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

printf("-----");

}

printf("\n");

cout<<" ";

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

if (box[i]) printf("| %2d ",box[i]);

else printf("| ");

}

printf("|\n");

cout<<" ";

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

printf("-----");

}

printf("\n\n");

cout<<"在老师手上的钥匙"<<endl;

cout<<" ";

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

if (key[i] == 0) printf(" %d ", i);

}

printf("\n\n\n\n\n\n\n");

while(time>0)

{

Sleep(1000);

time--;

cout<<" 第："<<timecount++<<"秒"<<endl;

}

}

int main() {

fileOpen("r");

int n = 0, k = 0;

fscanf(fp, "%d%d", &n, &k);

int w = 0, s = 0, c = 0;

Key \*borrow = (Key\*)malloc(sizeof(Key) \* k), \*ret = (Key\*)malloc(sizeof(Key) \* k);//借出和返回的储存

for (int i=0; i<k; i++) {

fscanf(fp, "%d%d%d", &w, &s, &c);

// 借出的钥匙号和时间

borrow[i].keyId = w;

borrow[i].time = s;

// 归还的钥匙号和时间

ret[i].keyId = w;

ret[i].time = s + c;

}

// 对借出和归还的钥匙，根据时间从小到大排序

for (int i=0; i<k; i++) {

for (int j=i; j<k; j++) {

if (borrow[j].time < borrow[i].time || (borrow[j].time == borrow[i].time && borrow[j].keyId < borrow[i].keyId)) {

Key temp = borrow[j];

borrow[j] = borrow[i];

borrow[i] = temp;

}

if (ret[j].time < ret[i].time || (ret[j].time == ret[i].time && ret[j].keyId < ret[i].keyId)) {

Key temp = ret[j];

ret[j] = ret[i];

ret[i] = temp;

}

}

}

// 将事件放进队列中

queue<Key> q;

int i = 0, j = 0;

while (i<k && j<k) {

if (borrow[i].time < ret[j].time) {

q.push(borrow[i]);

i++;

}

else if (borrow[i].time > ret[j].time) {

q.push(ret[j]);

j++;

}

else {

q.push(ret[j]);

j++;

}

}

while (i<k) {

q.push(borrow[i]);

i++;

}

while (j<k) {

q.push(ret[j]);

j++;

}

// 对key钥匙和box钥匙盒数组初始化。key数组：下标为编号，值为在box中的位置

int key[n+1], box[n+1];

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

box[i] = i;

key[i] = i;

}

// 根据队列进行事件安排

int time = 0;

while (!q.empty()) {

Key cur = q.front();

q.pop();

ShowBox(n, box, cur.time - time, key);

time = cur.time;

// 如果该钥匙已经发在一个位置了，就取出

if (key[cur.keyId]) {

box[key[cur.keyId]] = 0;

key[cur.keyId] = 0;

}

else {

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

if (box[i] == 0) {

key[cur.keyId] = i;

box[i] = cur.keyId;

break;

}

}

}

}

ShowBox(n, box, 1, key);

printf("\n所有老师已经下课\n");

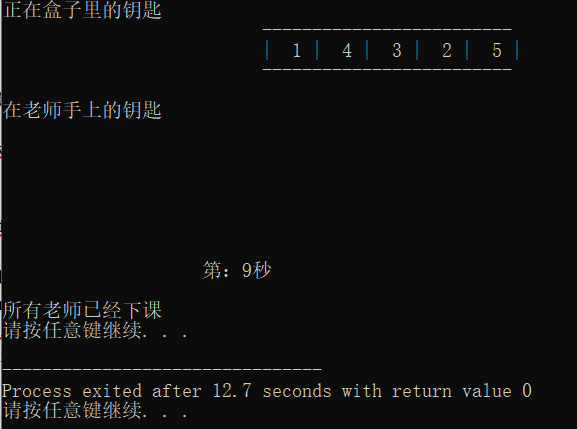
free(borrow);

free(ret);

fileClose();

system("pause");

}



目标4: 设计思路、问题和解决方法的清晰表达

设计思路：

代码按时间处理借还事件，并在命令行界面上实时显示钥匙盒和老师手上的钥匙情况。

使用队列进行事件处理，模拟时间流逝的过程。

问题及解决方法：

在模拟过程中，通过 ShowBox 函数实时展示钥匙盒和老师手上的钥匙状态，但由于界面实时刷新速度较慢，可能会造成显示不及时的问题。

使用 Sleep(1000) 函数每隔一秒更新界面，但在大量事件处理时，可能会导致界面卡顿或显示不准确。

可改进的地方

优化界面刷新：考虑使用多线程或异步方式更新界面，避免界面卡顿或显示延迟的情况。

优化算法和数据结构：通过使用更高效的数据结构和算法，优化钥匙借还的模拟过程，提高程序执行效率。

异常处理：添加对文件读取失败等异常情况的处理，以提高程序的健壮性。

时间复杂度：

代码中的主要操作包括文件读取、数据排序、模拟事件处理等。针对不同规模的输入数据，时间复杂度可能会有所不同，需要进一步分析。

# 15、【3】URL映射（选做）（字符串）

目标1: 数据结构类型和算法类型的分析和选择

数据结构类型：

该代码片段中使用了基本的数据结构，如整型变量、字符串数组等。

字符串操作是主要的处理方式，但没有使用标准库中更高级的字符串处理功能，例如 std::string 类的各种方法。

算法类型：

代码主要基于循环、条件语句，以及字符串处理的算法思想。

采用了逐字符比较和条件判断的方式，根据规则对输入字符串进行操作和匹配。

目标2: 组织、存储和处理数据的算法分析

数据存储和处理：

代码使用字符串数组 a[] 和 b[] 存储规则，并在查询时逐一匹配并处理输入字符串。

使用了循环嵌套和条件判断来遍历规则并处理输入字符串，以确定输出。

程序代码：

#include <bits/stdc++.h>

using namespace std;

int main(){

int n,m;

cin>>n>>m;

string a[100],b[100],s;

for(int i=0;i<n;i++){

cin>>a[i]>>b[i];

}

while(m--){

cin>>s;

bool flag=0;

int len2=s.length();

for(int i=0;i<n;i++){

int len1=a[i].length();

int x=0,y=0;

while(x<len1&&y<len2){

if(s[y]==a[i][x]){

x++;

y++;

continue;

}

if(a[i][x]!='<'){

break;

}

x++;

if(a[i][x]=='i'){

while(y<len2&&'0'<=s[y]&&s[y]<='9'){

y++;

}

x+=4;

continue;

}

if(a[i][x]=='s'){

bool flag2=0;

while(y<len2&&s[y]!='/'){

flag2=1;

y++;

}

if(!flag2){

break;

}

x+=4;

continue;

}

if(a[i][x]=='p'){

flag=1;

break;

}

}

if(x==len1&&y==len2){

flag=1;

}

if(flag){

x=0,y=0;

cout<<b[i];

while(x<len1&&y<len2){

if(a[i][x]==s[y]){

x++;

y++;

continue;

}

x++;

cout<<" ";

if(a[i][x]=='i'){

bool bo=0;

while(y<len2&&'0'<=s[y]&&s[y]<='9'){

if(s[y]>'0'){

bo=1;

}

if(bo){

cout<<s[y];

}

y++;

}

x+=4;

continue;

}

if(a[i][x]=='s'){

bool bo=0;

while(y<len2&&s[y]!='/'){

bo=1;

cout<<s[y];

y++;

}

x+=4;

continue;

}

if(a[i][x]=='p'){

while(y<len2){

cout<<s[y];

y++;

}

}

}

}

if(flag) break;

}

if(!flag){

cout<<"404";

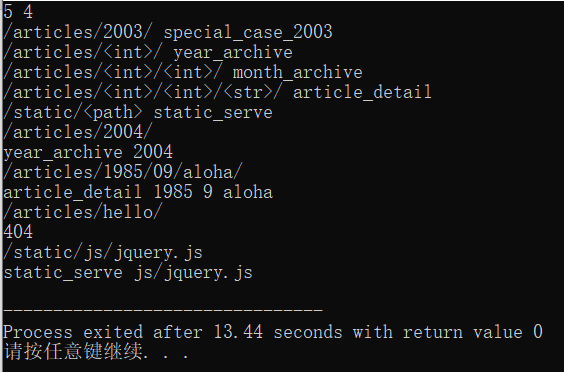
}

cout<<endl;

}

return 0;

}



目标4: 设计思路、问题和解决方法的清晰表达

设计思路：

代码设计了一种规则匹配和处理的方法，通过字符串规则定义对输入进行操作，并输出相应的结果。

问题及解决方法：

代码存在可读性较差的问题，例如缺乏注释和清晰的变量命名，使得代码难以理解。

另外，代码中的字符串匹配和处理方式较为复杂，可读性和可维护性较差。

可改进的地方

代码可读性：增加注释和清晰的变量命名，以提高代码的可读性。

采用标准库：使用标准库中提供的字符串处理函数，例如 std::string 类的成员函数，可以简化和优化字符串处理的过程。

优化算法：对算法进行重构和优化，以减少冗余的条件判断和循环，提高代码效率和可维护性。

时间复杂度

代码片段中的主要循环嵌套用于对规则进行匹配和处理，时间复杂度可能会随着输入数据大小的增加而增加，具体时间复杂度可能需要根据更多细节进行分析。这样的复杂字符串匹配和处理可能会导致较高的时间复杂度。

# 16、【2】行车路线 (选做)（图）

目标1：

在给定的实际案例中，选择了以下数据结构和算法：

数据结构类型：

邻接矩阵：用于表示图结构，存储节点之间的连接关系和道路类型。

结构体 Node：存储道路的长度和类型信息。

算法类型：

Dijkstra 算法：用于在图中找到单源最短路径，考虑了不同类型的道路（大道和小道）。

目标2：

数据组织和存储：

使用邻接矩阵存储图结构，记录节点间的连接关系和道路类型。

使用结构体 Node 存储道路长度和类型信息。

算法改进和设计：

根据道路类型（大道或小道），在 Dijkstra 算法中考虑不同的路径选择和更新规则。

实现了计算最小疲劳值的功能需求。

程序代码：

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <algorithm>

#include <iostream>

#include <fstream>

using namespace std;

#define MAX 999 //假设最多这么多站点

#define INF 9999999 //初始最初距离，表现为无穷大

struct Node

{

long long length;//记录道的长度

int type;//记录道的类型

};

struct Node Graph[MAX][MAX]; //邻接矩阵储存

//sum\_long、sum\_short,分别保存每个结点到出发点的大道路径之和、小道路径之和

long long sum\_short[MAX];

long long sum\_short\_square[MAX];

long long sum\_long[MAX];

long long d[MAX];

int pre[MAX]; //pre数组来保存路径上每个结点的前驱结点

int visited[MAX]; //储存已经访问的地方

int ans=INF;

long long square(long long x)

{

return x\*x;

}

void Dijkstra(int s,int n)

{

memset(visited,0,sizeof(visited));//初始化数据

fill(sum\_long,sum\_long+MAX,INF);

fill(sum\_short,sum\_short+MAX,INF);

fill(sum\_short\_square,sum\_short\_square+MAX,INF);

fill(pre,pre+MAX,0);

fill(d,d+MAX,INF);

d[s]=0;

sum\_long[s]=0;

sum\_short[s]=0;

while(true)

{

int u=-1;

long long mini=INF;//走到当前节点最小值

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

{

if(visited[i]==0&&d[i]<mini) //d[i]走到了i 就能赋值 否者break 相当于更新了mini,找到下一个点u

{

u=i;//遍历点 如果没走过就是u为（1，2，3，4，5，6）

mini=d[i];//记录当前的最小数 （0，9，25，45，66，76）

}

}

if(u==-1) break;

visited[u]=1; //此点走过

cout<<u<<"->";

for(int v=1;v<=n;v++) //到达u寻找v

{

if(Graph[u][v].length!=INF&&visited[v]==0) //有路 并且另外一个点没走过

{

if(Graph[u][v].type==1)//小路

{

long long d1=INF;

long long d2=INF;

if(sum\_short[u]!=INF)//如果有小道与源结点相连,即前面有小路

{

//需要处理一种特殊情况，就是大道、小道交错出现的情况

if(pre[u]==0) d1=square(sum\_short[u]+Graph[u][v].length);

else d1=sum\_long[pre[u]]+square(sum\_short[u]+Graph[u][v].length);//可能出现a通过大道与b相连，b通过小道与c相连

//c通过小道与d相连的情况

}

if(sum\_long[u]!=INF)//如果前面是大路

d2=square(Graph[u][v].length)+sum\_long[u];

if(d1<d2)

{

sum\_short[v]=min(sum\_short[v],sum\_short[u]+Graph[u][v].length);//更新结点对应的小道点权

sum\_short\_square[v]=min(d1,sum\_short\_square[v]);

}

else

{

sum\_short[v]=min(sum\_short[v],Graph[u][v].length);//由于上一条道路是大道，所以这里需要从头开始考虑小道的点权

sum\_short\_square[v]=min(d2,sum\_short\_square[v]);

pre[v]=u;//记录当前结点的前驱结点

}

d[v]=min(sum\_short\_square[v],sum\_long[v]);

}

else//当前道路是大道

{

long long d1=INF;

long long d2=INF;

if(sum\_long[u]!=INF) d1=sum\_long[u]+Graph[u][v].length;

if(sum\_short[u]!=INF) d2=sum\_short\_square[u]+Graph[u][v].length;

long long mini=min(d1,d2);

sum\_long[v]=min(sum\_long[v],mini);

d[v]=min(sum\_short\_square[v],sum\_long[v]);

}

}

}

}

return;

}

int main()

{

int n,m;

int t,a,b;

long long c;

ifstream infile("input.txt");

//scanf("%d %d",&n,&m);

infile>>n>>m;

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

{

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

{

Graph[i][j].length=INF;

Graph[j][i].length=INF;

}

}

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

{

infile>>t>>a>>b>>c;

//scanf("%d%d%d%lld",&t,&a,&b,&c);

if(t==1)//小道

{

Graph[a][b].length=c;

Graph[b][a].length=c;

Graph[a][b].type=1;

Graph[b][a].type=1;

}

else

{

Graph[a][b].length=c;

Graph[b][a].length=c;

Graph[a][b].type=0;

Graph[b][a].type=0;

}

}

infile.close();

cout<<"到达终点路线为：\n";

Dijkstra(1,n);

cout<<"\n"<<endl;

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

cout<<"到达第"<<i<<"个路口的最小疲劳值为："<<d[i]<<endl;

cout<<endl;

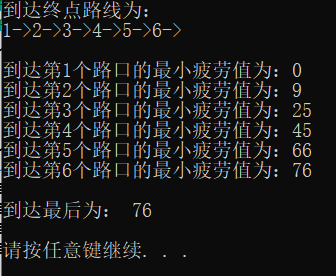
printf("到达最后为： %lld\n",d[n]);

cout<<endl;

system("pause");

return 0;

}



目标4：

设计思路：

使用邻接矩阵存储图数据，结合 Dijkstra 算法解决最短路径问题。

考虑了不同类型道路的特性，更新路径权重和选择路径的方式。

出现的问题及解决方法：

问题：需要在 Dijkstra 算法中考虑不同类型道路的影响。

解决方法：在算法中添加了条件判断，根据道路类型更新路径权重。

时间复杂度：

Dijkstra 算法的时间复杂度为 O(V^2)，其中 V 是节点数量。

算法在给定的最大节点数量下运行，能够在合理时间内找到最短路径。

可改进的地方：

优化算法以减少时间复杂度，例如使用堆优化的 Dijkstra 算法（优先队列）来减少时间开销。

考虑更多的边界情况和异常情况，增强代码的健壮性和鲁棒性。

# 17、【2】算术表达式求值 (选做) （栈）

目标1：

在这个实际案例中，使用了以下数据结构和算法：

数据结构类型：

双栈结构（doubleSqStack 和 charSqStack）：用于处理双栈计算器的输入和计算过程。

邻接矩阵：并未直接使用，而是利用堆栈实现了双栈计算器。

算法类型：

双栈计算器：根据后缀表达式进行计算。

目标2：

数据组织和存储：

使用文件读取和字符串数组存储表达式数据。

通过双栈结构对表达式进行处理和计算。

算法改进和设计：

设计了双栈计算器用于处理后缀表达式。

对输入的表达式进行了有效性检查，处理括号、小数点、运算符等异常情况。

时间复杂度分析：

输入的表达式经过处理后使用双栈计算器进行计算，时间复杂度约为 O(n)，n 为表达式长度。

程序代码：

#include<iostream>

#include<cstdio>

#include<cstdlib>

#include<string>

#include<fstream>

#include<cmath>

#include<cstring>

#include<string.h>

#include<iomanip>

#define OK 1

#define ERROR 0

#define TRUE 1

#define FALSE 0

#define STACK\_INIT\_SIZE 100

#define STACKINCREMENT 10

#define MAX\_N INT\_MAX

using namespace std;

typedef struct

{

double \*base;

double \*top;

int stacksize;

}doubleSqStack;

typedef struct

{

char \*base;

char \*top;

int stacksize;

}charSqStack;

void InitdoubleStack(doubleSqStack& S)

{

S.base=(double\*)malloc(STACK\_INIT\_SIZE\*sizeof(double));

if(!S.base)

exit(ERROR);

S.top=S.base;

S.stacksize=STACK\_INIT\_SIZE;

}

void doublePush(doubleSqStack &S,double &e)

{

if(S.top-S.base>=S.stacksize)

{

S.base=(double\*)realloc(S.base,(S.stacksize+STACKINCREMENT)\*sizeof(double));

if(!S.base)

exit(0);

S.top=S.base+S.stacksize;

S.stacksize+=STACKINCREMENT;

}

\*S.top = e;

++S.top;

}

bool doublePop(doubleSqStack &S,double &e)

{

if(S.top==S.base)

return false;

S.top--;

e=\*S.top;

return true;

}

double doubleGetTop(doubleSqStack S)

{

if(S.top==S.base)

{

cout<<"error";

exit(0);

}

return \*(S.top-1);

}

bool doubleTraverse(doubleSqStack &S)

{

if(!S.base || !S.top || S.base == S.top)

{

cout << "遍历失败！" << endl;

return ERROR;

}

double \*base = S.base;

double \*top = S.top;

while(top != base)

{

cout << "元素值为：" << \*(top-1)<<endl;

top--;

}

return OK;

}

void InitcharStack(charSqStack& S)

{

S.base=(char\*)malloc(STACK\_INIT\_SIZE\*sizeof(char));

if(!S.base)

exit(0);

S.top=S.base;

S.stacksize=STACK\_INIT\_SIZE;

}

void charPush(charSqStack& S,char e)

{

if(S.top-S.base>=S.stacksize)

{

S.base=(char\*)realloc(S.base,(S.stacksize+STACKINCREMENT)\*sizeof(char));

if(!S.base)

exit(0);

S.top=S.base+S.stacksize;

S.stacksize+=STACKINCREMENT;

}

\*S.top=e;

++S.top;

}

char charGetTop(charSqStack S)

{

if(S.top==S.base)

{

cout<<"error";

exit(0);

}

return \*(S.top-1);

}

bool charPop(charSqStack &S,char &e)

{

if(S.top==S.base)

return false;

S.top--;

e=\*S.top;

return true;

}

bool charTraverse(charSqStack &S)

{

if(!S.base || !S.top || S.base == S.top)

{

cout << "遍历失败！" << endl;

return ERROR;

}

char \*base = S.base;

char \*top = S.top;

while(top != base)

{

cout << "元素值为：" << \*(top-1)<<endl;

top--;

}

return OK;

}

char Precede(char a,char b)

{

char ch[7] = {'+','-','\*','/','(',')','#'};

int ch1,ch2;

char pre[7][10];

strcpy(pre[0],">><<<>>");

strcpy(pre[1],">><<<>>");

strcpy(pre[2],">>>><>>");

strcpy(pre[3],">>>><>>");

strcpy(pre[4],"<<<<<= ");

strcpy(pre[5],">>>> >>");

strcpy(pre[6],"<<<<< =");

for(int i = 0;i < 7;i++)

{

if(ch[i] == a)

ch1 = i;

if(ch[i] == b)

ch2 = i;

}

return pre[ch1][ch2];

}

bool isOPTR(char ch)

{

if(ch == '+' || ch == '-' || ch == '\*' || ch == '/')

return OK;

else

return ERROR;

}

bool isMARK(char ch)

{

if(ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '(' || ch == ')' || ch == '#' || ch == '.')

return OK;

else

return ERROR;

}

double Operate(double &a,char theta,double &b)

{

if(theta == '+')

return a+b;

else if(theta == '-')

return a-b;

else if(theta == '\*')

return a\*b;

else if(theta == '/')

{

if(b == 0)

{

cout << "ERROR :无法除0！" << endl;

return MAX\_N;

}

return a/b;

}

}

double Pow(double x,int y)

{

double result = 1;

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

{

result = result/x;

}

return result;

}

bool GetValue(string s)

{

doubleSqStack OPND;

charSqStack OPTR;

InitcharStack(OPTR);

InitdoubleStack(OPND);

charPush(OPTR,'#');

int count1 = 0;

int count2 = 0;

if(s[0] != '#' || s[s.length()-1] != '#')

{

cout << "界限符'#'错误（首尾应为#）！" << endl;

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

return ERROR;

}

if(isOPTR(s[1]))

{

cout << "运算符不能放在首位！" << endl;

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

return ERROR;

}

if(isOPTR(s[s.length()-2]))

{

cout << "运算符不能放在末位！" << endl;

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

return ERROR;

}

for(int i = 1;i <= s.length()-2;i++)

{

if(s[i] == '#')

{

cout << "界限符'#'错误（表达式内部不应有#）！" << endl;

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

return ERROR;

}

if(isOPTR(s[i]) && (isOPTR(s[i-1]) || isOPTR(s[i-1])))//运算符前后是运算符

{

cout << "运算符前后不能为运算符！" << endl;

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

return ERROR;

}

if(s[i] == '(')

{

count1++;

if(!isOPTR(s[i-1]) && s[i-1] != '#' && s[i-1] != '(')

{

cout << "左括弧前只能为运算符或'#'和'('！ " << endl;

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

return ERROR;

}

if(isOPTR(s[i+1]))

{

cout << "左括弧后不能直接接运算符！" << endl;

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

return ERROR;

}

else if(s[i+1] == ')')

{

cout << "括弧内不能为空！" << endl;

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

return ERROR;

}

}

else if(s[i] == ')')

{

count2++;

if(isOPTR(s[i-1]))

{

cout << "右括弧前不能直接接运算符！" << endl;

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

return ERROR;

}

if(!isOPTR(s[i+1]) && s[i+1] != '#' && s[i+1] != ')')

{

cout << "右括弧后只能接运算符或者'#'和')'！" << endl;

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

return ERROR;

}

}

if(s[i] == '.' && (isMARK(s[i+1]) || isMARK(s[i-1])))

{

cout << "小数点前后应为数字！" << endl;

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

return ERROR;

}

}

if(count1 != count2)

{

if(count1 > count2)

{

cout << "右括弧缺失！" << endl;

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

return ERROR;

}

else

{

cout << "左括弧缺失！" << endl;

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

return ERROR;

}

}

int count = 1;

char theta;

double a,b;

int flag = 0;

while((s[count] != '#' || charGetTop(OPTR)!='#') && count <= s.length()-1)

{

cout << endl;

if(!isMARK(s[count]))

{

double num1 = 0;

double num2 = 0;

while(!isMARK(s[count]))

{

num1 = num1\*10 + (s[count]-'0');

count++;

//cout << s[count];

}//当count位置上的元素为字符时停止

if(s[count] == '.')

{

count ++;

int i = -1;

while(!isMARK(s[count]))

{

num2 = num2 + (double)(s[count]-'0')\*Pow(10.0,i);

i--;

count++;

//cout << s[count];

}

}

double num = num1 + num2;

//cout <<endl<< "num:" <<num<<endl;

//cout << "num1:" << num1<<" num2:" << num2<<endl;

doublePush(OPND,num);

}

else

{

switch(Precede(charGetTop(OPTR),s[count]))

{

case '<':

{

charPush(OPTR,s[count]);

count++;

break;

}

case '=':

{

charPop(OPTR,theta);

count++;

break;

}

case '>':

{

charPop(OPTR,theta);

doublePop(OPND,b);

doublePop(OPND,a);

double result = Operate(a,theta,b);

if(result == MAX\_N)

return ERROR;

else

doublePush(OPND,result);

break;

}

}

}

cout << endl << "运算符栈：";

cout << "从栈顶到栈尾依次为：" << endl;

charTraverse(OPTR);

cout << "运算数栈：";

cout << "从栈顶到栈尾依次为：" << endl;

doubleTraverse(OPND);

}

double result = doubleGetTop(OPND);

cout << "运算结果为：" << result << endl;

return OK;

}

int main()

{

ifstream infile("data.txt",ios::in);

if(!infile.is\_open())

{

cout << "文件无法打开！" << endl;

exit(ERROR);

}

string expression[20];

int ExpressionNum = 1;

while(!infile.eof())

{

getline(infile,expression[ExpressionNum]);

cout << ExpressionNum << ": " << expression[ExpressionNum] << endl;

ExpressionNum++;

}

ExpressionNum--;

cout << "ExpressionNum:"<<ExpressionNum<<endl;

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

{

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

cout << i << ": ";

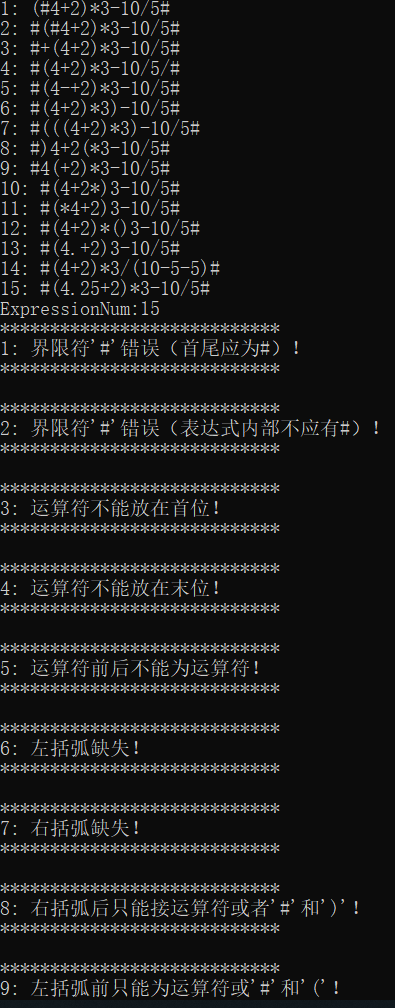
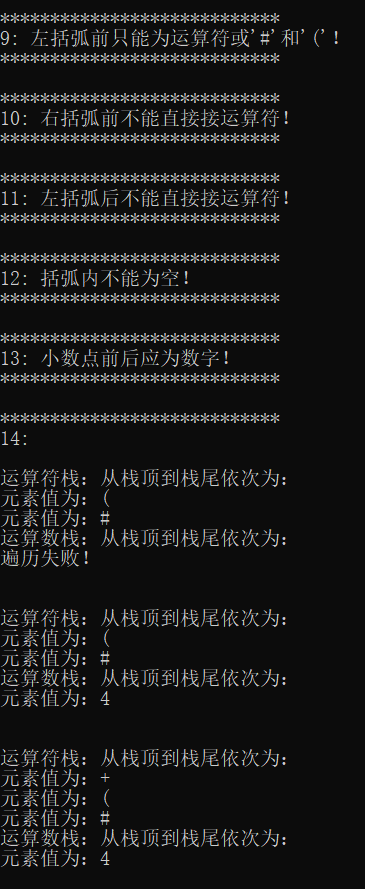
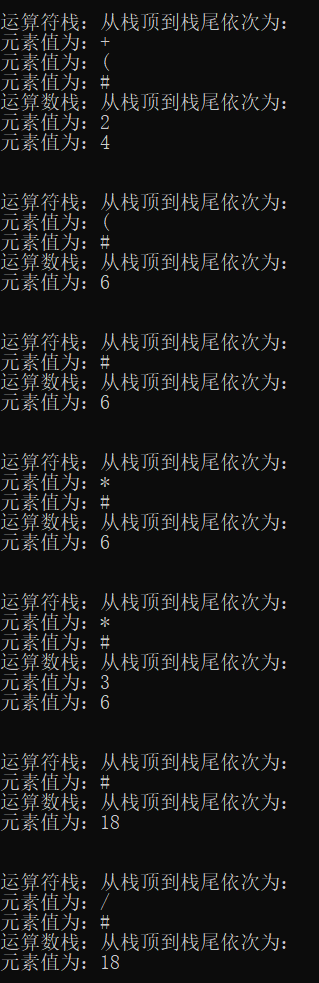
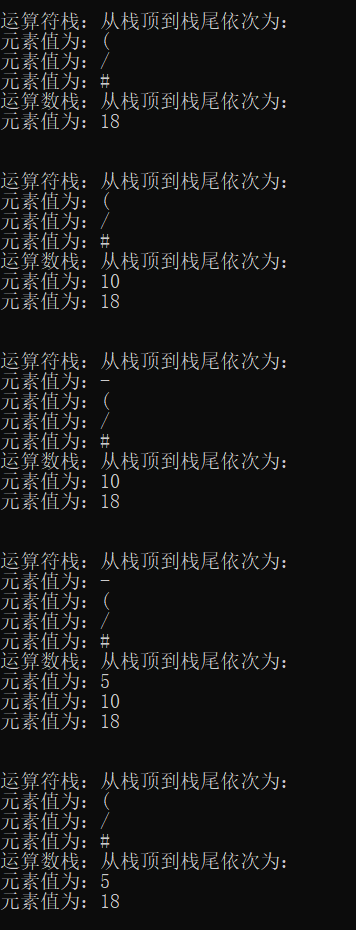
GetValue(expression[i]);

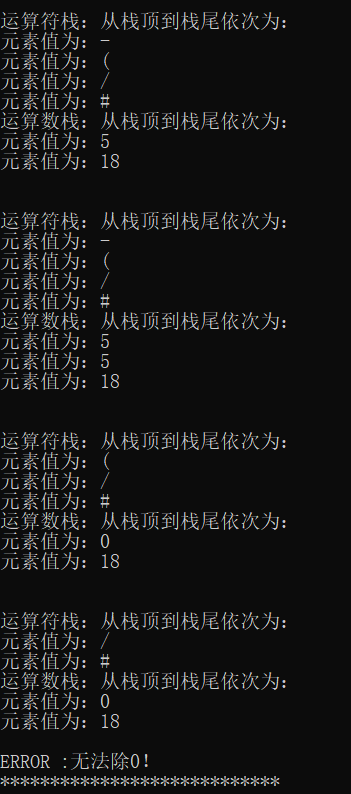
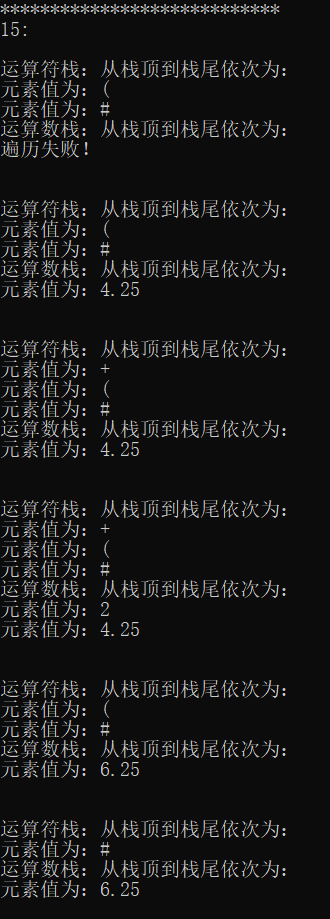
}

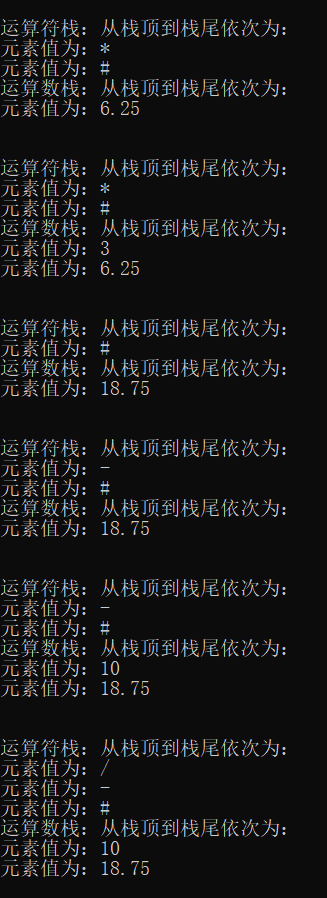
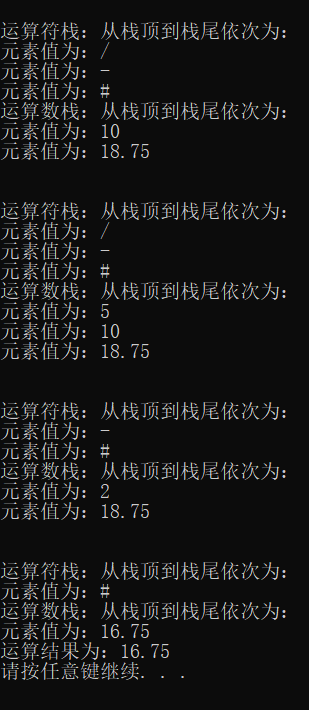
system("pause");

infile.close();

}

目标4：

设计思路：

双栈计算器： 采用双栈结构设计，一个用于操作数（doubleSqStack），另一个用于运算符（charSqStack）。利用后缀表达式进行计算，通过栈的特性实现运算过程。

输入合法性检查： 在处理表达式前，进行了有效性检查，检查括号匹配、运算符位置、小数点位置等异常情况，提高程序的健壮性。

数据结构的动态管理： 使用了动态内存分配进行栈空间的管理，确保栈的大小能够满足需求。

出现的问题及解决方法：

异常输入处理： 针对括号匹配、小数点位置、运算符位置等异常情况进行了处理，若出现异常，输出相应的错误信息，提高了程序的容错性和可靠性。

动态内存管理： 在动态内存分配过程中，出现内存分配失败的可能性，通过检查分配结果并进行合适的错误处理来解决此类问题。

可改进的地方：

内存管理优化： 代码中使用了动态内存分配（malloc 和 realloc），可以考虑使用 C++ 中的 std::vector 进行动态数组管理，更便捷且安全。

代码结构优化： 将计算逻辑进一步封装成函数，增加代码的模块化和可读性。

错误处理增强： 考虑更多的异常情况，提供更具体的错误信息和异常处理，使程序更健壮。

# 19、【2】Hash表应用 （选做） （查找）

目标1：分析数据结构类型和算法类型

数据结构类型

结构体：Person和Record结构体用于表示人员信息和飞行记录。

哈希表：使用哈希表来实现基于链地址法和平方探测法的冲突解决。

算法类型

哈希算法：基于链地址法和平方探测法的哈希算法，解决冲突问题，保证查找效率。

排序算法：使用std::sort对人员信息进行排序，选择合适的比较函数实现对个人信息的排序。

目标2：有效组织、存储和处理数据，改进满足功能需求的算法

数据处理

生成模拟数据：随机生成360个人的信息和1000条飞行记录，使用这些数据进行各种操作。

数据存储：使用结构体和哈希表，对人员信息和飞行记录进行有效的存储。

算法改进

哈希查找：实现了基于链地址法和平方探测法的哈希查找，并对哈希表进行了打印输出。

数据排序：通过自定义比较函数对人员信息进行排序，实现根据乘坐次数和总里程进行排序。

程序代码：

#include <iostream>

#include <fstream>

#include <string>

#include <cstring>

#include <map>

#include <queue>

#include <algorithm>

#include <sstream>

#include <random>

#include <iomanip>

#include <time.h>

using namespace std;

#define Person\_Num 360

float MAXT = 0;

float MAXKM = 0;

struct Person

{

string ID;//身份证号

string name;//姓名

long int Total\_KM=0;//总里程

int times = 0;//乘坐次数

}person[Person\_Num+1];

struct Record

{

int Flight\_ID;//航班号

string Flight\_Date;//航班日期

string ID;//乘坐人身份证号

int KM;//里程

}record[1000];

vector <string> split(string& str,const char space = ' ')

//字符串分割函数split

{

istringstream sstr(str);

vector <string> result;

string tmp;

while(getline(sstr,tmp,space))

{

result.push\_back(tmp);

}

return result;

}

vector<string> generate\_name();

void gennerate\_ID()

//生成ID和姓名

{

srand(time(0));

vector<string> nm = generate\_name();

for(int i=0;i<360;++i)

{

string str = to\_string(rand()%9+1);

for(int j=0;j<16;++j)

{

switch(rand()%10)

{

case 0:

str += '0';

break;

case 1:

str += '1';

break;

case 2:

str += '2';

break;

case 3:

str += '3';

break;

case 4:

str += '4';

break;

case 5:

str += '5';

break;

case 6:

str += '6';

break;

case 7:

str += '7';

break;

case 8:

str += '8';

break;

case 9:

str += '9';

break;

}

}

switch(rand()%10)

{

case 0:

str += 'X';

break;

case 1:

str += '1';

break;

case 2:

str += '2';

break;

case 3:

str += '3';

break;

case 4:

str += '4';

break;

case 5:

str += '5';

break;

case 6:

str += '6';

break;

case 7:

str += '7';

break;

case 8:

str += '8';

break;

case 9:

str += '9';

break;

}

person[i].ID = str;

person[i].name = to\_string(rand()%nm.size());

//person[i].name = nm[rand()%nm.size()];

}

}

vector<string> generate\_name()

//随机生成姓名 （中文编码紊乱，只能用数字了好吗）

{

fstream name;

name.open("name.txt",ios::in);

vector<string> nm;

string str;

while(getline(name,str))

{

vector<string> tmp;

tmp = split(str,'、');

vector<string>::iterator it;

for(it=tmp.begin();it!=tmp.end();++it)

{

//cout<<\*it<<endl;

nm.push\_back(\*it);

}

}

return nm;

}

void generate\_record()

//随机生成1000条飞行记录

{

srand(time(0));

vector<string> nm = generate\_name();

vector <int> Flight\_ID = {

12001,12002,12003,12004,12005,12006,12007,

13001,13002,13003,13004,13005,13006,13007,

14001,14002,14003,14004,14005,14006,14007,

15001,15002,15003,15004,15005,15006,15007

};

int Flight\_ID\_size = Flight\_ID.size();

int size = Person\_Num;

string YEAR = "2021";

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

{

string MONTH = to\_string(rand()%12 +1);

string DAY = to\_string(rand()%27 +1);

string date = YEAR + '-' + MONTH + '-' + DAY;

int fid = Flight\_ID[rand()%Flight\_ID\_size];

int pid = rand()%size;

person[pid].times++;

record[i].ID = person[pid].ID;

record[i].Flight\_ID = fid;

record[i].Flight\_Date = date;

record[i].KM = rand()%1000;//1000以内的里程数

person[pid].Total\_KM += record[i].KM;

}

}

void Write\_File()

{

fstream infile;

infile.open("data.txt",ios::out|ios::app);

cout<<"下面模拟生成360个人的信息:"<<endl;

gennerate\_ID();

cout<<"ID已生成！"<<endl;

generate\_name();

cout<<"姓名已生成！"<<endl;

cout<<"下面模拟生成1000条飞行记录:"<<endl;

generate\_record();

cout<<"记录已生成"<<endl;

cout<<"下面将信息写入文件日志:"<<endl;

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

{

infile

<<record[i].ID

<<" "<<record[i].Flight\_Date

<<" "<<record[i].Flight\_ID

<<" "<<record[i].KM<<endl;

}

cout<<"信息已写入!"<<endl;

infile.close();

}

struct HashNode

{

string key;//关键字

Record\* arr = new Record[1000];//链地址

int size=0;//链地址大小

}HashList[Person\_Num],HashList2[2000];

int search(int i)

{

for(int j=0;j<Person\_Num;++j)

{

if(record[i].ID==HashList[j].key)

{

return j;

}

}

return -1;

}

int search2(int i)

{

for(int j=0;j<Person\_Num;++j)

{

if(record[i].ID==person[j].ID)

{

return j;

}

}

return -1;

}

void Print\_Hash1();

void Hash1()

{

for(int i=0;i<Person\_Num;++i)

{

HashList[i].key = person[i].ID;

}

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

{

int j = search(i);//查找第i条record对应的ID在person数组中的位置

if(j == -1)

{

cout<<"查无此ID！"<<endl;

return;

}

HashList[j].arr[HashList[j].size++] = record[i];

}

cout<<"下面打印哈希表:"<<endl;

Print\_Hash1();

}

void Print\_Hash1()

{

cout<<setw(10)<<"ID"<<setw(24)<<"航班号"<<setw(18)<<"航班日期"<<endl;

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

for(int i=0;i<Person\_Num;++i)

{

if(HashList[i].size==0)

{

continue;

}

else

{

for(int j=0;j<HashList[i].size;++j)

{

cout<<setw(10)<<HashList[i].key<<setw(15)<<HashList[i].arr[j].Flight\_ID

<<setw(20)<<HashList[i].arr[j].Flight\_Date<<endl;

}

cout<<"------------------------------------------统计次数:"<<HashList[i].size<<endl;

}

}

}

void Hash2()

{

cout<<setw(10)<<"ID"<<setw(24)<<"航班号"<<setw(18)<<"航班日期"<<setw(15)<<"冲突次数"<<endl;

for(int i=0;i<2000;++i)

{

HashList2[i].key = "";

}

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

{

int count=0;

int tmp = search2(i);//查找第i条record对应的ID在person数组中的位置

if(tmp == -1)

{

cout<<"查无此ID"<<endl;

return;

}

while(HashList2[tmp].key != "")

{

++count;

tmp += count\*count;

tmp %= 2000;

if(HashList2[tmp].key != "")

{

tmp -= 2\*count\*count;

tmp %= 2000;

}

}

HashList2[tmp].key = record[i].ID;

cout<<setw(10)<<record[i].ID<<setw(15)<<record[i].Flight\_ID

<<setw(20)<<record[i].Flight\_Date<<setw(10)<<count<<endl;

}

}

float MAX\_T()

{

int max = 0;

for(int i=0;i<Person\_Num;++i)

{

if(person[i].times>max)

{

max = person[i].times;

}

}

return float(max);

}

float MAX\_KM()

{

int max = 0;

for(int i=0;i<Person\_Num;++i)

{

if(person[i].Total\_KM>max)

{

max = person[i].Total\_KM;

}

}

return float(max);

}

bool cmp(const Person &a,const Person &b)

{

float aa = float(a.times)/MAXT;

float ab = float(a.Total\_KM)/MAXKM;

float ba = float(b.times)/MAXT;

float bb = float(b.Total\_KM)/MAXKM;

//cout<<(aa+ab)\*0.5 <<" "<< (ba+bb)\*0.5<<endl;

return (aa+ab)\*0.5 > (ba+bb)\*0.5;

//return a.Total\_KM\*a.times>b.times\*b.Total\_KM;

}

int main()

{

Write\_File();

system("pause");

system("cls");

while(1)

{

cout<<"本程序提供以下功能:"<<endl;

cout<<"1.基于链地址法解决冲突的哈希查找"<<endl;

cout<<"2.基于平方探测法解决冲突的哈希查找"<<endl;

cout<<"3.乘机频数统计及VIP客户筛选"<<endl;

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

cout<<"请输入你要使用的功能:";

int c;

cin>>c;

Person sort\_person[Person\_Num];

for(int i=0;i<Person\_Num;++i)

{

sort\_person[i] = person[i];

}

MAXT = MAX\_T();

MAXKM = MAX\_KM();

switch(c)

{

case 1:

Hash1();

system("pause");

system("cls");

break;

case 2:

Hash2();

system("pause");

system("cls");

break;

case 3:

//memcpy(sort\_person,&person,sizeof person);

sort(sort\_person,sort\_person+Person\_Num,cmp);

cout<<MAXT<<" "<<MAXKM<<endl;

cout<<setw(10)<<"ID"<<setw(18)<<"姓名"<<setw(15)<<"乘坐次数"<<setw(11)<<"总里程"<<setw(11)<<"重要系数"<<endl;

for(int i=0;i<Person\_Num;++i)

{

float weight;

weight = (float(sort\_person[i].times)/MAXT) + float(sort\_person[i].Total\_KM)/MAXKM;

weight \*= 0.5;

cout<<setw(10)<<sort\_person[i].ID<<setw(10)<<sort\_person[i].name

<<setw(10)<<sort\_person[i].times<<setw(15)<<sort\_person[i].Total\_KM

<<setw(10)<<setiosflags(ios::fixed)<<setprecision(4)<<weight;

if(i<=10)

cout<<" ★"<<endl;

else

cout<<endl;

}

system("pause");

system("cls");

break;

default:

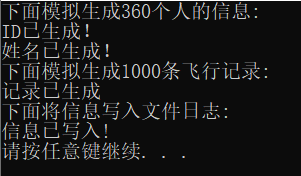
exit(0);

}

}

return 0;

}



目标4：设计思路、出现问题及解决方法

设计思路

数据生成：利用随机数生成身份证号和姓名，模拟生成了人员信息和飞行记录。

哈希表处理：实现了两种不同的哈希查找算法，对冲突进行了处理。

统计分析：根据乘坐次数和总里程，对人员信息进行了排序。

出现的问题及解决方法

哈希查找问题：出现了哈希冲突，使用链地址法和平方探测法解决了这一问题，保证了查找的准确性。

数据生成：生成的姓名存在编码问题，只能用数字表示姓名。

改进空间

姓名编码问题：解决姓名编码混乱的情况，使得生成的姓名更为真实可读。

优化算法效率：进一步优化哈希算法，提高查找效率。

时间复杂度

哈希表构建：哈希表构建的时间复杂度为 O(n)，其中 n 为记录数目。

哈希查找：平均情况下的哈希查找时间复杂度为 O(1)，最坏情况下为 O(n)。

# 21、【3】树的应用 (选做)（树）

目标1：分析数据结构类型和算法类型

数据结构类型

结构体：使用结构体 TNode 来表示树的节点，每个节点包含键、值以及指向子节点和兄弟节点的指针。

树：采用树形结构来构建和存储JSON数据。

算法类型

递归算法：在创建树和查找值的过程中使用了递归的思想，逐层建立树形结构并查找对应的值。

目标2：有效组织、存储和处理数据，改进满足功能需求的算法

数据处理

文件读取：通过文件操作读取JSON格式的数据，逐行解析。

字符串处理：实现了字符串的处理，根据冒号和逗号分割键值对。

算法改进

树的构建：利用递归的方式创建树形结构，遍历解析JSON数据。

查找值：根据输入的键值路径查找对应的值，通过递归实现快速查找。

程序代码：

#include<cstdio>

#include<cstdlib>

#include<cstring>

#include<iostream>

#include<windows.h>

using namespace std;

typedef struct TNode {

char key[100];

char value[100];

TNode \*child;

TNode \*brother;

}\*Tree;

// 全局变量

FILE \*fp;

char fileName[] = "input.txt";

// 打开文件

void fileOpen(char \*method) {

if((fp=fopen(fileName,method)) == NULL) {

printf("file cannot be opened\n");

return;

}

//else printf("file has been opened\n");

}

// 关闭文件

void fileClose() {

if(fclose(fp)!=0) printf("file cannot be closed\n");

//else printf("file has been closed\n");

return;

}

// 先序遍历

void TraverseTree(Tree tree) {

if (tree) {

printf("%s:%s\n", tree->key, tree->value);

TraverseTree(tree->brother);

TraverseTree(tree->child);

}

}

// 销毁

void DestroyTree(Tree &tree) {

if (tree) {

DestroyTree(tree->brother);

DestroyTree(tree->child);

free(tree);

}

}

// 字符串处理函数

void HandleStr(char \*source, int start, int end, char \*dist) {

int i = start+1;

int j = 0;

while (i != end) {

if (source[i] == '\\') i++;

dist[j] = source[i];

i++;

j++;

}

dist[j] = '\0';

}

// 建树函数，递归

void CreateTree(Tree &tree) {

char str[1000];

fscanf(fp, "%s", str); // 读取'{'的下一行的字符串

int i = 0;

TNode \*preNode = tree, \*node;

bool isFirstNode = true; // 判断是否为根节点

while (str[i] != '}') {

node = (TNode\*)malloc(sizeof(TNode));

printf("前一半======%s\n", str);

if (str[i] == '"') {

int j = i;

while (str[j] != ':') j++;

HandleStr(str, i, j-1, node->key);

i = j; // i位置为':'

}

i++;// 跳过':'到value部分

printf("后一半=====%s\n", str);

if (str[i] == '{') {

strcpy(node->value, "object");

CreateTree(node->child); // 值是一个对象，则递归操作建子树

}

else if (str[i] == '"') {

node->child = NULL;

int j = i;

// j<str字符串长度是为了结束字符'}'前没有','的情况

while (str[j] != ',' && str[j] != '}' && j < strlen(str)) j++;

HandleStr(str, i, j-1, node->value);

i = j; // i位置为','或'}'

}

// 为第一个结点时将node结点指针赋给tree指针

if (isFirstNode) {

tree = node;

isFirstNode = false;

}

// 不是第一个结点，则将上个结点的兄弟指针指向当前结点

else preNode->brother = node;

preNode = node;

fscanf(fp, "%s", str);

i = 0;

}

node->brother = NULL;

}

// 找value值

char\* FindValue(Tree tree, char \*str) {

int i = 0;

char key[1000];

while (str[i]!='.' && i<strlen(str)) {

key[i] = str[i];

i++;

}

key[i] = '\0';

TNode \*node = tree;

while (node) {

if (strcmp(node->key, key) == 0) {

if (str[i] == '.') {

int j = 0;

char nextStr[1000];

i++;

while (i<strlen(str)) {

nextStr[j] = str[i];

i++;

j++;

}

nextStr[j] = '\0';

return FindValue(node->child, nextStr);

}

else return node->value;

}

node = node->brother;

}

return NULL;

}

// 显示树

void ShowTree(Tree tree, int tab) {

TNode \*node = tree;

while (node) {

for (int i=0; i<tab; i++) printf("| ");

printf("|--%s", node->key);

if (node->child) {

printf("\n");

ShowTree(node->child, tab+1);

}

else printf(":%s\n", node->value);

node = node->brother;

}

}

int main() {

fileOpen("r");

int n = 0, m = 0;

fscanf(fp, "%d%d", &n, &m);

Tree tree = NULL;

char strLine[10000];

fscanf(fp, "%s", strLine);// 读取开始的'{'

CreateTree(tree);

system("cls"); // 清屏

cout<<"Json树的结构如下图所示"<<endl;

printf("+\n");

ShowTree(tree, 0);

cout<<"\n\n\n\n\n返回结果:"<<endl;

// 读取输入数据

for (int i=0; i<m; i++) {

fscanf(fp, "%s", strLine);

char \*ret = FindValue(tree, strLine);

char value[1000];

if (!ret) sprintf(value, "NOTEXIST");

else if (strcmp(ret, "object") == 0) sprintf(value, "OBJECT");

else sprintf(value, "STRING %s", ret);

printf("%s\n", value);

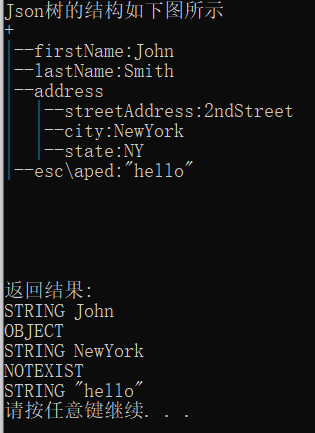
}

DestroyTree(tree);

system("pause");

fileClose();

}



目标4：设计思路、出现问题及解决方法

设计思路

文件读取和处理：通过逐行解析文件内容，根据JSON格式构建树形结构。

树的构建：利用递归方式实现树的构建和查找，提供了查找值的功能。

出现的问题及解决方法

字符串处理问题：在解析键值对时，字符串的处理可能存在潜在的边界情况导致错误，需要更精确地处理边界情况。

内存泄漏问题：在树的销毁过程中，需要确保每个节点被正确释放，避免内存泄漏。

数据结构和算法思想

数据结构：使用树形结构储存和处理JSON数据。

算法思想：利用递归构建树结构，实现对JSON数据的遍历和查找。

测试数据和结果

测试数据：通过读取 input.txt 文件中的数据进行测试。

结果：成功构建了树形结构并实现了根据键路径查找值的功能，结果输出为每个查询键路径对应的值或错误信息。

时间复杂度

树的构建：在遍历解析JSON数据时采用了递归方式，时间复杂度取决于JSON数据的大小和结构。

查找值：根据键值路径查找值的时间复杂度取决于树的深度和数据的存储结构，最坏情况下为树的高度。

可改进的地方

字符串处理：提高对字符串边界情况的处理能力，确保更加准确地提取键值对。

内存管理：确保在树的销毁过程中释放每个节点的内存，避免内存泄漏问题的发生。

代码行数：

1、331行

2、160行

3、612行

4、297行

5、67行

6、312行

7、564行

8、633行

14、170行

15、111行

16、158行

17、416行

19、418行

21、197行

总：4446行