数据结构课程设计报告

班级： 1619401

学号： 061910322

姓名： 殷圣权

指导老师： 孙 涵

目录

**课设程序分析及样例分析3**

1. 总体完成情况3
2. 系统进程统计（必做）4
3. 迷宫问题（必做）27
4. 家谱管理系统（必做）38
5. Huffman编码与解码（必做）58
6. 地铁修建（必做）73
7. 公交线路提示（必做）81
8. 平衡二叉树操作的演示（必做）104
9. 排序算法比较（必做）127
10. 消除类游戏（选做）153
11. 公共钥匙盒（选做）159
12. 行车路线（选做）174
13. 算术表达式求值 （选做）188
14. 树的应用 （选做）206
15. 最小生成树（选做）216
16. 心得体会229

# 总体完成情况

1. 完成情况

1、必做题完成情况：全部完成

2、选做题完成情况：

消除类游戏：2分（CSP）

公共钥匙盒：2分（CSP）

行车路线：2分（CSP）

算术表达式求值：2分

树的应用：3分

最小生成树：3分

总计：2+2+2+2+3+3（分） = 14（分）

1. 代码行数

系统进程统计：422

迷宫问题：195

家谱管理系统：361

Huffman编码与解码：242

地铁修建：124

公交线路提示：409

平衡二叉树操作的演示：468

排序算法比较：439

消除类游戏：65

公共钥匙盒:249

行车路线：185

算术表达式求值：344

树的应用：121

最小生成树：210

总代码行数：3834

# 系统进程统计

[问题描述]

　　设计一个程序，每秒统计一次当前系统的进程状况，并按照内存使用自多到少排序打印输出相关信息。对已经结束的进程，另外给出一个列表，并显示该进程的结束时间和持续时间。

[基本要求]

（1） 该题目要求使用两个链式线性表。一个链表存储当前活动进程，要求使用单向链表，并按照内存使用自多到少排序。另外一个链表存储已结束进程，要求使用双向链表，按照持续时间自少到多排序。

（2） 每秒在窗口内更新一次当前系统进程情况，输出内容包括：进程名，持续时间，内存使用情况。

（3） 每秒在窗口内更新一次已结束进程情况，输出内容包括：进程名，持续时间，结束时间。

（4） 注意进程在这两个链表中的切换，一个进程既可被结束，也可以过一段时间后再被运行。

1. 数据结构：

单链表和双链表

1. 算法设计思想：

使用两个单链表分别存放当前进程信息以及所有进程信息（作为对照），使用一个双链表存放已结束进程信息。

当前进程经对照后得出进程运行时间，若有新进程出现，插入对照链表，然后再与当前进程链表对比得出已结束进程，并在对照链表中删除。

1. 源程序：

#include<iostream>

#include<ctime>

#include<stdlib.h>

#include<stdio.h>

#include<vector>

#include<iomanip>

#include<cstring>

#include <windows.h>

#define Ok 1

#define Error 0

using namespace std;

vector<char\*> vl;

class task

{

public:

char name[128];

int memory;

SYSTEMTIME start;

SYSTEMTIME end;

double time;

task(char\* t) : memory(0)

{

char\* b = t + 1, \*p = t + 1;

while (\*p)

{

if (\*p == '"')

{

\*p = 0;

strcpy(name, b);

break;

}

p++;

}

for (b = p + 1; p[1] != 'K'; p++)

if (\*p == '"')

b = p;

for (char\* c = b + 1; c != p; c++)

if (\*c == ',')

continue;

else

memory = 10 \* memory + \*c - '0';

}

bool operator<(task& t) { return this->memory < t.memory; }//排序

};

typedef struct DLNode//双链表结点

{

task end;

struct DLNode\* prior, \* next;

}DLNode, \* DLinkList;

typedef struct LNode//单链表结点

{

task run;

struct LNode\* next;

}LNode, \* LinkList;

int InitDList(DLinkList& L);//初始化双链表

int InitList(LinkList& L);//初始化单链表

int ClearDList(DLinkList& L);//清空双链表

int ClearList(LinkList& L);//清空单链表

int insertDList(DLinkList& L, DLNode\* p);//插入双链表

int insertList(LinkList& L, LNode\* p);//插入单链表

void TraverseDList(DLinkList L);//遍历双链表

void TraverseList(LinkList L);//遍历单链表

int showpresent(LinkList& L, LinkList LL, DLinkList& Lt);//显示当前进程

int showend(DLinkList& L, LinkList LL, LinkList Lt);//显示结束进程

int InitLL(LinkList& L);//初始化对照链表

int circuit(DLinkList& L, LinkList L1);//根据当前删除结束进程

int circuitll(LinkList& pre, LinkList now);//根据现在校验对照组

int main()

{

LinkList L, LL;//LL-对照,L-当前进程

DLinkList L2;//已结束进程

InitList(L);

InitList(LL);

InitDList(L2);

InitLL(LL);

long t = time(NULL);

while (1)

{

ClearList(L);

showpresent(L, LL, L2);

circuit(L2, L);

showend(L2, LL, L);

circuitll(LL, L);

//getchar();//暂停服务

}

}

int InitDList(DLinkList& L)//初始化双链表

{

L = (DLNode\*)malloc(sizeof(struct DLNode));

if (!L)

return Error;

L->next = NULL;

L->prior = NULL;

return Ok;

}

int InitList(LinkList& L)//初始化单链表

{

L = (LNode\*)malloc(sizeof(struct LNode));

if (!L)

return Error;

L->next = NULL;

return 1;

}

int ClearDList(DLinkList& L)//清空双链表

{

free(L->next);

L->next = NULL;

return 1;

}

int ClearList(LinkList& L)//清空单链表

{

free(L->next);

L->next = NULL;

return 1;

}

int insertDList(DLinkList& L, DLNode\* p)//插入双链表

{

DLNode\* temp = L->next, \* temp2 = L->next;

if (!temp)

{

L->next = p;

p->prior = L;

return Ok;

}

while (temp)

{

if (temp->end.end.wHour > p->end.end.wHour || (temp->end.end.wHour == p->end.start.wHour && temp->end.end.wMinute > p->end.end.wMinute) || (temp->end.end.wHour == p->end.end.wHour && temp->end.end.wMinute == p->end.end.wMinute && temp->end.end.wSecond > p->end.end.wMinute))

temp = temp->next;

else

break;

}

if (!temp)

{

while (temp2->next != NULL)

temp2 = temp2->next;

temp2->next = p;

p->prior = temp2;

return Ok;

}

temp->prior->next = p;

p->next = temp;

p->prior = temp->prior;

temp->prior = p;

return Ok;

}

int insertList(LinkList& L, LNode\* p)//插入单链表

{

LNode\* temp = L->next, \* temp2=L;

if (L->next == NULL)

{

L->next = p;

return 1;

}

while (temp)

{

if (temp->run.memory >= p->run.memory)

temp = temp->next;

else

break;

}

if (!temp)

{

while (temp2->next != NULL)

temp2 = temp2->next;

temp2->next = p;

return 1;

}

while (temp2->next != temp)

temp2 = temp2->next;

temp2->next = p;

p->next = temp;

return 1;

}

void TraverseDList(DLinkList L)//遍历双链表

{

DLNode\* temp = L->next;

cout << setw(50) << left << "\n已结束进程名称";

cout << setw(15) << left << "内存调用";

cout << setw(10) << left << "持续时间";

cout << setw(10) << left << "结束时间" << endl;

cout << "===================================================================================================" << endl;

while (temp)

{

cout << setw(50) << left << temp->end.name;

cout << setw(15) << left << temp->end.memory;

cout << left << temp->end.time << "s" << '\t';

cout << left << temp->end.start.wHour << "h" << temp->end.start.wMinute << "min" << temp->end.start.wSecond << "s" << endl;

temp = temp->next;

}

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

}

void TraverseList(LinkList L)//遍历单链表

{

LNode\* temp = L->next;

cout << setw(70) << left << "\n当前进程名称";

cout << setw(15) << left << "内存调用";

cout << setw(10) << left << "持续时间" << endl;

cout << "=======================================================================================================" << endl;

while (temp)

{

cout << setw(70) << left << temp->run.name;

cout << setw(15) << left << temp->run.memory;

cout << setw(10) << left << (double)temp->run.time << "s" << endl;

temp = temp->next;

}

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

}

int showpresent(LinkList& L, LinkList LL, DLinkList& Lt)//显示当前进程

{

int flag = 0;

char buffer[128], cmd[] = "tasklist /FO CSV";

FILE\* pipe = \_popen(cmd, "r");

if (!pipe)

return Error;

fgets(buffer, 128, pipe);

while (!feof(pipe))

{

if (fgets(buffer, 128, pipe))

{

flag = 0;

LNode\* p = (LNode\*)malloc(sizeof(struct LNode));

LNode\* temp;

temp = LL->next;

p->run = task(buffer);

p->next = NULL;

GetLocalTime(&p->run.start);

while (temp)

{

if (strcmp(temp->run.name, p->run.name) == 0)

{

flag = 1;

p->run.time = (p->run.start.wMinute - temp->run.start.wMinute) \* 60.0 + (p->run.start.wSecond - temp->run.start.wSecond) \* 1.0 + (p->run.start.wMilliseconds - temp->run.start.wMilliseconds) \* 0.001;

break;

}

temp = temp->next;

}

if (flag == 0)

p->run.time = 0;

insertList(L, p);

}

}

\_pclose(pipe);

TraverseList(L);

}

int showend(DLinkList& L, LinkList LL, LinkList Lt)//显示结束进程

{

int i = 0;

int flag = 0;

LNode\* temp = LL->next, \* temp2 = Lt->next;

vector<char\*> vnow;

while (temp2)

{

vnow.push\_back(temp2->run.name);

temp2 = temp2->next;

}

while (temp)

{

int flag = 0;

vector<char\*>::iterator iter;

iter = vnow.begin();

while (iter != vnow.end())//LL在Lt中找不到

{

if (strcmp(\*iter, temp->run.name) == 0)

{

flag = 1;

break;

}

iter++;

}

if (flag == 0)

{

DLNode\* s = (DLNode\*)malloc(sizeof(struct DLNode));

strcpy(s->end.name, temp->run.name);

s->end.memory = temp->run.memory;

s->end.start = temp->run.start;

s->next = NULL;

SYSTEMTIME now;

GetLocalTime(&now);

GetLocalTime(&s->end.end);

s->end.time = (now.wMinute - s->end.start.wMinute) \* 60.0 + (now.wSecond - s->end.start.wSecond) \* 1.0 + (now.wMilliseconds - s->end.start.wMilliseconds) \* 0.001;

s->end.start = now;

insertDList(L, s);

vl.push\_back(s->end.name);

}

temp = temp->next;

}

TraverseDList(L);

return 1;

}

int InitLL(LinkList& L)//初始化对照链表

{

char buffer[128], cmd[] = "tasklist /FO CSV";

FILE\* pipe = \_popen(cmd, "r");

if (!pipe)

return Error;

fgets(buffer, 128, pipe);

while (!feof(pipe))

{

if (fgets(buffer, 128, pipe))

{

LNode\* p = (LNode\*)malloc(sizeof(struct LNode));

p->run = task(buffer);

p->next = NULL;

GetLocalTime(&p->run.start);

insertList(L, p);

}

}

\_pclose(pipe);

}

int circuit(DLinkList& L, LinkList L1)//根据present删除结束进程

{

LNode\* temp = L1->next;

while (temp)

{

vector<char\*>::iterator iter;

iter = vl.begin();

while (iter != vl.end())

{

if (strcmp(\*iter, temp->run.name) == 0)

{

DLNode\* temp3 = L->next;

while (strcmp(\*iter, temp3->end.name) != 0)

{

temp3 = temp3->next;

}

DLNode\* temp2 = L;

while (temp2->next != temp3)

{

temp2 = temp2->next;

}

temp2->next = temp3->next;

vl.erase(iter);

break;

}

iter++;

}

temp = temp->next;

}

return 1;

}

int circuitll(LinkList& pre, LinkList now)//根据现在校验对照组

{

vector<char\*> prev;

LNode\* temp = now->next, \* temp2 = pre->next;

while (temp2)

{

prev.push\_back(temp2->run.name);

temp2 = temp2->next;

}

temp = now->next, temp2 = pre->next;

while (temp2)//寻找now中不存在的删掉

{

temp = now->next;

int flag = 0;

while (temp)

{

if (strcmp(temp->run.name, temp2->run.name) == 0)

{

flag = 1;

break;

}

temp = temp->next;

}

if (flag == 0)

{

LNode\* temp3 = pre->next;

while (temp3->next != temp2)

{

temp3 = temp3->next;

}

temp2 = temp2->next;

temp3->next = temp2;

continue;

}

temp2 = temp2->next;

}

temp = now->next, temp2 = pre->next;

while (temp)//now中存在但LL不存在的添加

{

int flag = 0;

vector<char\*>::iterator it;

it = prev.begin();

while (it != prev.end())

{

if (strcmp(\*it, temp->run.name) == 0)//LL中存在的

{

flag = 1;

break;

}

it++;

}

if (flag == 0)

{

LNode\* s = (LNode\*)malloc(sizeof(struct LNode));

strcpy(s->run.name, temp->run.name);

s->run.memory = temp->run.memory;

s->run.start = temp->run.start;

s->run.time = temp->run.time;

insertList(pre, s);

}

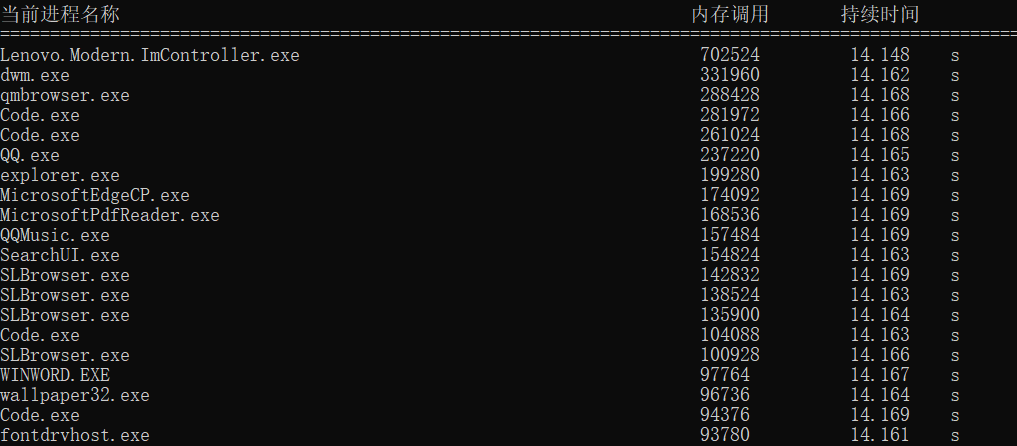
temp = temp->next;

}

return 1;

}

1. 测试数据和结果：





1. 时间复杂度分析：

遍历算法的时间复杂度为 O(n)，根据对照链表链表修改另一个链表信息的时间复杂度为 O(n^2)

# 迷宫问题

[问题描述]

利用栈操作实现迷宫问题求解。

[基本要求]

（1）从文件中读取数据，生成模拟迷宫地图，不少于20行20列。

（2）给出任意入口和出口，显示输出迷宫路线。

1. 数据结构：

顺序存储的栈结构

1. 算法设计思想：

判断在某位置时下一次移动的方向有哪些，并标记哪些方向已经走过。每走到一个位置将此位置信息压栈，直至走到终点，或者栈空时表示没有通路。

1. 源程序：

#include <iostream>

#include <cstdlib>

#include <cstring>

#include <cmath>

using namespace std;

#define MAX 500

struct Point//迷宫上的位置结构（栈类型）

{

int x, y;//坐标

char c;//位置上的符号

int vector[4], sum;///0 1 2 3 表示上下左右

int visit[4];//访问标记

}point[MAX][MAX];

typedef struct//栈结构

{

Point base[MAX];

int top;

}SqStack;

SqStack SqStackInit()//初始化栈

{

SqStack S;

S.top = -1;

return S;

}

bool StackEmpty(SqStack S)//判断栈空

{

if(S.top == -1)

return true;

else

return false;

}

void SqStackPush(SqStack &S, Point x)//入栈

{

if(S.top == MAX-1)

{

cout << "栈满！" << endl;

return ;

}

S.base[++S.top] = x;

}

void SqStackPop(SqStack &S)//出栈

{

if(S.top == -1)

{

cout << "栈空！" << endl;

return ;

}

S.top--;

}

Point GetSqStackTop(SqStack S)//取栈顶元素

{

return (S.base[S.top]);

}

void init(int n, int m)//对迷宫数据进行初始化

{

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

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

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

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

{

point[i][j].sum = 0;//表示共有几个方向可以走，初始化为0

for(int k=0;k<4;k++)//k取0、1、2、3时，依次表示上、下、左、右

if(point[i][j].c == '.' && point[i+a[k]][j+b[k]].c == '.')

{

point[i][j].vector[k] = 1;//该方向标记

point[i][j].sum++;

point[i][j].visit[k] = 0;//访问标记，可走

}

else//死路，不走

{

point[i][j].vector[k] = 0;

point[i][j].visit[k] = 1;//访问标记，不能走

}

}

}

void show(int n, int m, char answer[][500])//输出迷宫路径图

{

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

{

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

cout << answer[i][j];

cout << endl;

}

}

void solve(int n, int m)//寻找迷宫路径

{

char answer[MAX][MAX];//结果迷宫线路图

SqStack S = SqStackInit();//初始化栈

memset(answer, '.', sizeof(answer));//初始化结果迷宫图

int x, y, xx, yy;

cout << "\n请输入入口位置：";

cin >> x >> y;

cout << "请输入出口位置：";

cin >> xx >> yy;

SqStackPush(S, point[x][y]);//将入口压栈

while(!StackEmpty(S))//当栈非空时

{

Point top = GetSqStackTop(S);//取栈顶元素

//show2(n,m);

if(point[x][y].visit[0] == 0 && point[x][y].vector[0] == 1)//可以向上走

{

point[x++][y].visit[0] = 1;

point[x][y].visit[1] = 1;

SqStackPush(S,point[x][y]);

}

else if(point[x][y].visit[1] == 0 && point[x][y].vector[1] == 1)//可以向下走

{

point[x--][y].visit[1] = 1;

point[x][y].visit[0] = 1;

SqStackPush(S,point[x][y]);

}

else if(point[x][y].visit[2] == 0 && point[x][y].vector[2] == 1)//可以向左走

{

point[x][y--].visit[2] = 1;

point[x][y].visit[3] = 1;

SqStackPush(S,point[x][y]);

}

else if(point[x][y].visit[3] == 0 && point[x][y].vector[3] == 1)//可以向右走

{

point[x][y++].visit[3] = 1;

point[x][y].visit[2] = 1;

SqStackPush(S,point[x][y]);

}

//上下左右都走过了

else if(point[x][y].visit[0] == 1 && point[x][y].visit[1] == 1 && point[x][y].visit[2] == 1 && point[x][y].visit[3] == 1)

{

SqStackPop(S);//弹栈

if(StackEmpty(S))///当前点上下左右全访问过并且此点不是终点

{

cout << "没有可行路径！\n";

break;

}

}

x = GetSqStackTop(S).x;

y = GetSqStackTop(S).y;

if(point[x][y].x == xx && point[x][y].y == yy)//抵达终点

{

cout<<"\n找到迷宫路线：\n";

while(!StackEmpty(S))

{

Point ans;

ans.x = GetSqStackTop(S).x;

ans.y = GetSqStackTop(S).y;

SqStackPop(S);

x = ans.x;

y = ans.y;

answer[x][y] = '@';

}

show(n, m, answer);

break;

}

}

}

int main()

{

int n, m;

memset(point, 'x', sizeof(point));//初始化全墙壁

cout << "欢迎来到迷宫问题\n--------------------------------------------------------\n生成迷宫如下：\n";

FILE \*fp = fopen("迷宫路.txt", "r");

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

int k;

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

{

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

{

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

if(k == 0)

{

point[i][j].c = '.';

cout << point[i][j].c;

}

else

cout << '#';

point[i][j].x = i;

point[i][j].y = j;

}

cout << endl;

}

init(n, m);

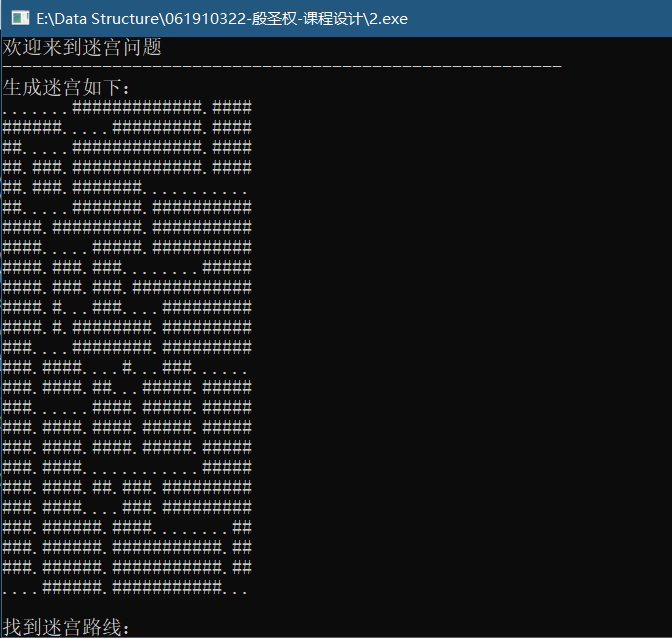
solve(n, m);

fclose(fp);

return 0;

}

1. 测试数据和结果：





1. 时间复杂度分析：

寻找迷宫路径时的时间复杂度为O(n)，初始化迷宫以及录入迷宫各点数据时为O(n^2)

# 家谱管理系统

[问题描述]

实现具有下列功能的家谱管理系统。

[基本要求]

（1）输入文件以存放最初家谱中各成员的信息，成员的信息中均应包含以下内容：姓名、出生日期、婚否、地址、健在否、死亡日期（若其已死亡），也可附加其它信息、但不是必需的。

（2）实现数据的文件存储和读取。

（3）以图形方式显示家谱。

（4）显示第n 代所有人的信息。

（5）按照姓名查询，输出成员信息（包括其本人、父亲、孩子的信息）。

（6）按照出生日期查询成员名单。

（7）输入两人姓名，确定其关系。

（8）某成员添加孩子。

（9）删除某成员（若其还有后代，则一并删除）。

（10）修改某成员信息。

（11）要求建立至少40个成员的数据，以较为直观的方式显示结果，并提供文稿形式以便检查。

（12）界面要求：有合理的提示，每个功能可以设立菜单，根据提示，可以完成相关的功能要求。

（13）存储结构：根据系统功能要求自行设计，但是要求相关数据要存储在数据文件中。测试数据：要求使用1、全部合法数据；2、局部非法数据。进行程序测试，以保证程序的稳定。

1. 数据结构：

孩子兄弟二叉树

1. 算法设计思想：

在孩子兄弟二叉树的基础上在每个树结点增加一个指向父亲的指针，用于寻找该孩子的父亲。

1. 源程序：

#include<iostream>

#include<string.h>

#include<stdlib.h>

#include<stdio.h>

#include<fstream>

#include<cstring>

#include <iomanip>

#define MAXSIZE 10

#define INFINITY 65535

#define LOCAL

using namespace std;

fstream fp;

typedef struct TNode//树结点信息

{

struct TNode \*child;//孩子

struct TNode \*brother;//兄弟

struct TNode \*parent;//父亲

char Name[20];

int Year,Month,Day;

int Marry;

char Located[100];

int Death;

int DYear,DMonth,DDay;

int Level;

} TNode,\*Tree;

void InitTree( Tree &T)

{

T=(TNode \*)malloc (sizeof (TNode));

T->child=NULL;

T->brother=NULL;

T->Level=1;

}

void CreateTree( Tree &T)//建立家谱树（先序创建）

{

char name[20];

char b[20]={"#"};

fp>>name;

if(strcmp(name,b)==0)

{

T=NULL;

}

else

{

if(!T) exit(1);

strcpy(T->Name,name);

int year,month,day;

fp>>year>>month>>day;

T->Year=year;

T->Month=month;

T->Day=day;

int marry;

fp>>marry;

T->Marry=marry;

char locate[100];

fp>>locate;

strcpy(T->Located,locate);

int death;

fp>>death;

T->Death=death;

if(death)

{

fp>>year>>month>>day;

T->DYear=year;

T->DMonth=month;

T->DDay=day;

}

InitTree(T->brother);

T->brother->Level=T->Level;

T->brother->parent=T->parent;

CreateTree(T->brother);

InitTree(T->child);

T->child->parent=T;

T->child->Level=T->Level+1;

CreateTree(T->child);

}

}

TNode \*FindNodeName(Tree T,char name[20])//寻找对应名字的结点

{

TNode \*p = NULL;

if(T == NULL)

{

return NULL;

}

else if(strcmp(name,T->Name)==0)

{

return T;

}

else

{

p=FindNodeName(T->child,name);

if(p==NULL)

{

return (FindNodeName(T->brother,name));

}

else return p;

}

}

void Imformation(Tree T)//输出信息

{

cout<<T->Name<<'\t'<<T->Year<<" "<<T->Month<<" "<<T->Day<<'\t'<<T->Marry<<'\t'<<T->Located<<'\t'<<T->Death<<'\t';

if(T->Death==1)

cout<<T->DYear<<'\t'<<T->DMonth<<'\t'<<T->DDay<<endl;

else cout<<endl;

}

void FindNodeBirth(Tree T,int year,int month,int day)//寻找对应生日结点

{

if(T!=NULL)

{

if(T->Year==year and T->Month==month and T->Day==day)

Imformation(T);

FindNodeBirth(T->brother,year,month,day);

FindNodeBirth(T->child,year,month,day);

}

}

void OutPut(Tree T,int i)//显示家谱树

{

if(T!=NULL)

{

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

cout<<'\t';

cout<<T->Name;

if(T->child!=NULL)

cout<<endl;

OutPut(T->child,i+1);

if(T->brother!=NULL)

cout<<endl;

OutPut(T->brother,i);

}

}

void OutNum(Tree T,int n)//输出第n代人；

{

if(T!=NULL)

{

if(T->Level==n)

Imformation(T);

OutNum(T->brother,n);

OutNum(T->child,n);

}

}

void RelationShip(TNode \*p,TNode \*q)//判断关系

{

if(p->Level<q->Level)

cout<<p->Name<<"是"<<q->Name<<"的长辈"<<endl;

else if(p->Level==q->Level)

cout<<p->Name<<"和"<<q->Name<<"是一辈人"<<endl;

else if(p->Level>q->Level)

cout<<p->Name<<"是"<<q->Name<<"的晚辈"<<endl;

}

void CreateChild(Tree &C)//新添加孩子的信息

{

InitTree(C);

cout<<"请输入孩子的姓名，出生日期:"<<endl;

char name[20];

char a[20];

scanf("%c%[^\n]",&a,name);

strcpy(C->Name,name);

int year,month,day;

cin>>year>>month>>day;

C->Year=year;

C->Month=month;

C->Day=day;

}

void AddNumber(Tree &T,TNode \*p)//添加成员

{

TNode \*q;

q=p;

Tree C;

CreateChild(C);

if(p->child==NULL)

{

p->child=C;

C->parent=p;

C->Level=p->Level+1;

strcpy(C->Located,p->Located);

}

else

{

p=p->child;

while(p->brother!=NULL)

{

p=p->brother;

}

p->brother=C;

C->parent=q;

C->Level=p->Level;

strcpy(C->Located,q->Located);

}

cout<<"已完成"<<endl;

}

Tree DestoryTree(Tree &T)

{

if(T)

{

T->child =DestoryTree(T->child);

T->brother=DestoryTree(T->brother);

T=NULL;

}

}

void Alter(Tree &T,TNode \*p)//修改成员信息

{

cout<<"请输入修改信息:"<<endl;

cout<<"姓名:" ;

char name[20];

char a[20];

scanf("%c%[^\n]",&a,name);

strcpy(p->Name,name);

cout<<"出生日期:";

int year,month,day;

cin>>year>>month>>day;

p->Year=year;

p->Month=month;

p->Day=day;

cout<<"婚姻情况:";

int marry;

cin>>marry;

p->Marry=marry;

cout<<"住址:";

char locate[20];

scanf("%c%[^\n]",&a,locate);

strcpy(T->Located,locate);

int death;

cout<<"是否去世:";

cin>>death;

T->Death=death;

if(death==1)

{

cout<<"死亡日期:";

cin>>year>>month>>day;

T->DYear=year;

T->DMonth=month;

T->DDay=day;

}

}

int main()

{

Tree T;

int i=INFINITY;

cout<<"请选择需要的操作："<<endl;

cout<<"1：创建家谱。"<<endl;

cout<<"2：显示家谱。"<<endl;

cout<<"3：显示第n代人的信息"<<endl;

cout<<"4：按照姓名查询，输出成员信息。"<<endl;

cout<<"5：按照出生日期查询成员名单。"<<endl;

cout<<"6：输入两人姓名，确定其关系。"<<endl;

cout<<"7：某成员添加孩子。"<<endl;

cout<<"8：删除某成员。"<<endl;

cout<<"9：修改某成员信息。"<<endl;

cout<<"0：退出程序"<<endl;

while(i)

{

cin>>i;

switch(i)

{

case 1:

fp.open("family.txt");

InitTree (T);

T->parent=NULL;

CreateTree(T);

fp.close();

cout<<"家谱已创建完成"<<endl;

break;

case 2:

OutPut(T,1);

cout<<endl;

break;

case 3:

int n;

cout<<"请选择需要输出第几代人:";

cin>>n;

cout<<"姓名"<<'\t'<<"生日"<<'\t'<<" 婚姻情况"<<'\t'<<"住址"<<'\t'<<"是否去世"<<'\t'<<"死亡日期"<<endl;

OutNum(T,n);

cout<<"该代人的信息已输出"<<endl;

break;

case 4:

cout<<"请输入查找的人名";

char name[20];

char a[20];

scanf("%c%[^\n]",&a,name);

TNode \*p,\*q;

p=new TNode;

q=new TNode;

p=FindNodeName(T,name);

cout<<"姓名"<<'\t'<<"生日"<<'\t'<<" 婚姻情况"<<'\t'<<"住址"<<'\t'<<"是否去世"<<'\t'<<"死亡日期"<<endl;

Imformation(p);

q=p->child;

cout<<"孩子信息为："<<endl;

while(q!=NULL)

{

Imformation(q);

q=q->brother;

}

q=p->parent;

cout<<"父亲信息为:"<<endl;

Imformation(q);

break;

case 5:

cout<<"请输入需要查找的人的生日:"<<endl;

int y,m,d;

cin>>y>>m>>d;

cout<<"姓名"<<'\t'<<"生日"<<'\t'<<" 婚姻情况"<<'\t'<<"住址"<<'\t'<<"是否去世"<<'\t'<<"死亡日期"<<endl;

FindNodeBirth(T,y,m,d);

break;

case 6:

cout<<"请输入需查找的两个人的姓名："<<endl;

char N1[20];

char N2[20];

scanf("%c%[^\n]",&a,N1);

scanf("%c%[^\n]",&a,N2);

p=FindNodeName(T,N1);

q=FindNodeName(T,N2);

RelationShip(p,q);

break;

case 7:

cout<<"请选择新增加孩子的成员:"<<endl;

scanf("%c%[^\n]",&a,name);

p=FindNodeName(T,name);

AddNumber(T,p);

break;

case 8:

cout<<"请选择需删除的成员:"<<endl;

scanf("%c%[^\n]",&a,name);

p=FindNodeName(T,name);

q=p->parent;

if(q->child==p)

{

DestoryTree(p->child);

p->parent->child=p->brother;

free(p);

}

else

{

TNode \*x;

x=new TNode;

x=q->child;

while(x->brother!=p)

{

x=x->brother;

}

x->brother=p->brother;

DestoryTree(p->child);

free(p);

}

break;

case 9:

cout<<"请选择需要修改信息的成员:"<<endl;

scanf("%c%[^\n]",&a,name);

p=FindNodeName(T,name);

Alter(T,p);

break;

case 0:

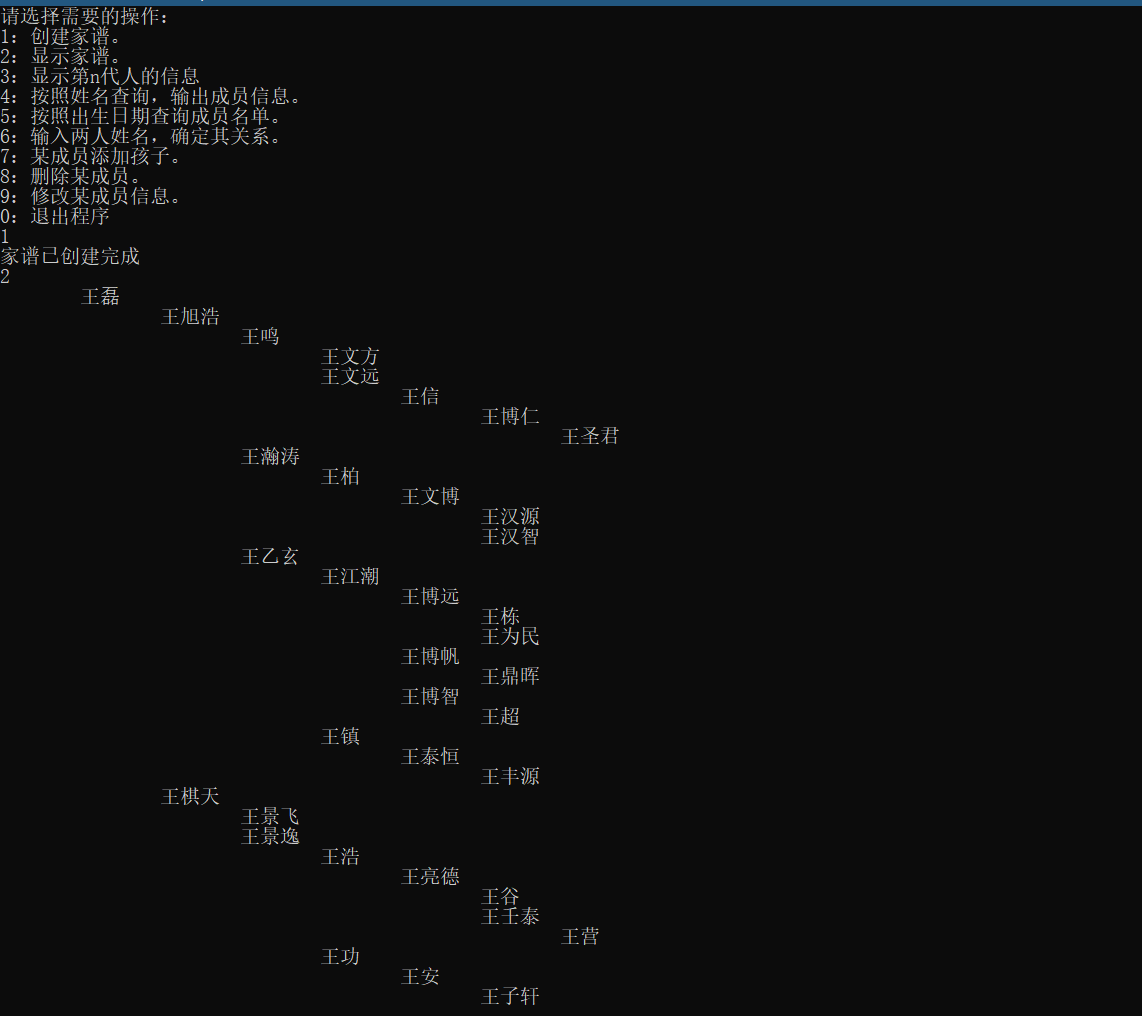
break;

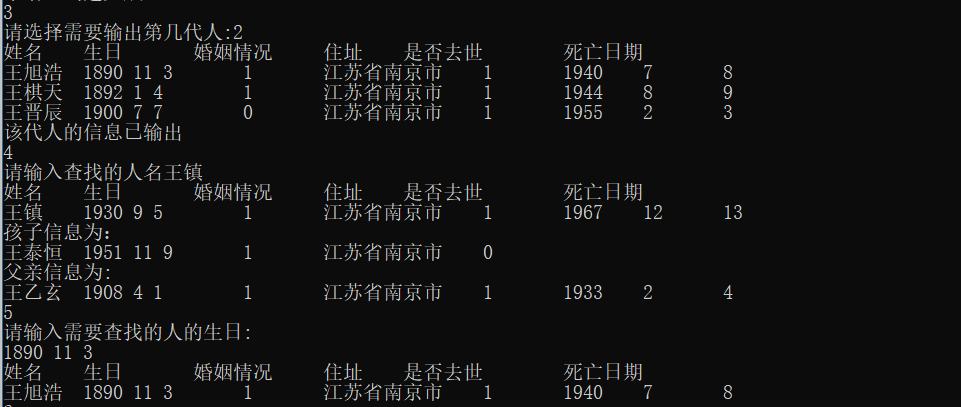
}

}

}

1. 测试数据和结果：





1. 时间复杂度分析：

先序遍历操作为：O(n)，其他操作建立在遍历的基础之上。

# Huffman编码与解码

[问题描述]

对一篇不少于5000字符的英文文章（source.txt），统计各字符出现的次数，实现Huffman编码(code.dat)，以及对编码结果的解码(recode.txt)。

[基本要求]

（1） 输出每个字符出现的次数和编码,并存储文件(Huffman.txt)。

（2） 在Huffman编码后，英文文章编码结果保存到文件中(code.dat)，编码结果必须是二进制形式，即0 1的信息用比特位表示，不能用字符’0’和’1’表示。

（3） 实现解码功能。

1. 数据结构：

Huffman树

1. 算法设计思想：

由读取到的各字符频率建立Huffman树，即把每个字符看成一棵树，这些字符构成一片森林，每次寻找两个权值最小的结点进行连接形成一棵树，重复此操作直至森林中只剩下一颗树，以左子树标0，右子树标1的形式进行各字符的编码。另外在写入文件时注意对齐规则，采用满八位写入。

1. 源程序：

#include<iostream>

#include<stdlib.h>

#include<stdio.h>

#include<string>

#include<fstream>

#include<errno.h>

#include<assert.h>

#include<bitset>

using namespace std;

typedef struct HTNode

{

int position;

int weight;

int flag;

string code;

struct HTNode\* parent, \* lchild, \* rchild;

}HTNode, \* HuffmanTree;

int word = 0;

int wordnum=0;//文章中的字符数

int words[300];//各字符频率

string code[100000];//存储编码

char article[100000];//存储文章字符

void Select(HuffmanTree\* HF, int& s1, int& s2, int n); //选择权值较小的两个

void HuffmanCoding(HuffmanTree\* HF, int n); //建立霍夫曼树

void HuffmanCode(HuffmanTree HF); //编码

void createcode(HuffmanTree\* HF); //对文章编码

void WriteBit(char x, FILE\* p); //八位存入

void WriteCode(HuffmanTree \*HF); //编码二进制存储在code.dat

void recode(HuffmanTree\* HF); //解码

void shouhuffmancode(HuffmanTree\* HF); //输出频率和编码

int main()

{

int i, n = 256;

HuffmanTree\* HF;

HTNode\* p;

HF = (HuffmanTree\*)malloc((2 \* n + 1) \* sizeof(HuffmanTree));

createcode(HF);

shouhuffmancode(HF);

WriteCode(HF);

recode(HF);

}

void Select(HuffmanTree\* HF, int& s1, int& s2, int n)

{

int k, w,i;

w = INT\_MAX;

for (k = 1; k <= n; k++)

{

if (HF[k]->weight < w && HF[k]->flag == 0 && HF[k]->weight != 0)

{

w = HF[k]->weight;

s1 = k;

}

}

int j = s1;

w = INT\_MAX;

for (k = 1; k <= n; k++)

{

if (HF[k]->weight < w && k != j && HF[k]->flag == 0 && HF[k]->weight != 0)

{

w = HF[k]->weight;

s2 = k;

}

}

}

void HuffmanCoding(HuffmanTree\* HF, int n)//建立霍夫曼树

{

int s1, s2, i, j;

for (i = n + 1; i <= 2 \* n; i++)

{

Select(HF, s1, s2, i);

HF[s1]->flag = 1;

HF[s2]->flag = 1;

HF[s1]->parent = HF[i];

HF[s2]->parent = HF[i];

HF[i]->lchild = HF[s1];

HF[i]->rchild = HF[s2];

HF[i]->weight = HF[s1]->weight + HF[s2]->weight;

HF[i]->flag = 0;

}

}

void HuffmanCode(HuffmanTree HF)//建立霍夫曼编码

{

if (HF->lchild != NULL)

{

code[HF->lchild->position] = code[HF->position] + '0';

HuffmanCode(HF->lchild);

}

if (HF->rchild != NULL)

{

code[HF->rchild->position] = code[HF->position] + '1';

HuffmanCode(HF->rchild);

}

return;

}

void createcode(HuffmanTree\* HF)//初始化数据

{

fstream file;

file.open("source.txt", ios::in);

assert(file.is\_open());

char c;

int k = 1;

int t = 0;

file >> noskipws;

while (!file.eof())

{

file >> c;

words[c]++;

article[wordnum] = c;

wordnum++;

}

for (int i = 1; i <= 256; i++)

{

if (words[i])

word++;

}

for (int i = 1; i <= 256; i++)

{

if (words[i])

{

HTNode\* p = (HTNode\*)malloc(sizeof(struct HTNode));

p->position = i;

p->weight = words[i];

p->lchild = NULL;

p->rchild = NULL;

p->flag = 0;

HF[k] = p;

k++;

}

}

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

{

HTNode\* p = (HTNode\*)malloc(sizeof(struct HTNode));

p->position = i+10000;

p->weight = 0;

p->lchild = NULL;

p->rchild = NULL;

p->flag = 0;

HF[i] = p;

}

HuffmanCoding(HF, word);

HuffmanCode(HF[2 \* word-1]);

file.close();

}

int s = 0;

char b = 0;

void WriteBit(char x, FILE\* p)//八位存入

{

if (x == '1')

b |= 1 << 7 - s;

else if (x == '0')

b |= 0 << 7 - s;

s++;

if (s == 8)

{

fwrite(&b, sizeof(char), 1, p);

b = 0;

s = 0;

}

}

void WriteCode(HuffmanTree \*HF)//编码二进制存储在code.dat

{

FILE\* file=fopen("code.dat","w");;

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

{

for (int j = 0; j < code[article[i]].size(); j++)

{

WriteBit(code[article[i]][j], file);

}

}

fclose(file);

}

void recode(HuffmanTree\* HF)//解码

{

cout << "-------------------------------------解码如下-----------------------------------------" << endl;

HTNode\* maxn, \* temp;

maxn = (HTNode\*)malloc(sizeof(struct HTNode));

temp = (HTNode\*)malloc(sizeof(struct HTNode));

maxn->weight = 0;

int i = 0, m = 0;

maxn = HF[2 \* word - 1];

FILE\* fp= fopen("code.dat", "rb");

string rcode;

char t;

temp = maxn;

while (fread(&t, sizeof(char), 1, fp))//读二进制文件

{

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

{

if (((t >> 7) & 1) == 1)

rcode += '1';

else

rcode += '0';

t <<= 1;

}

}

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

{

if (rcode[i] == '0')

{

temp = temp->lchild;

if (temp->lchild == NULL && temp->rchild == NULL)

{

cout << (char)temp->position;

temp = maxn;

}

}

else if (rcode[i] == '1')

{

temp = temp->rchild;

if (temp->lchild == NULL && temp->rchild == NULL)

{

cout << (char)temp->position;

temp = maxn;

}

}

}

fclose(fp);

}

void shouhuffmancode(HuffmanTree\* HF)//输出频率和编码

{

cout << "-------------------------------------编码如下-----------------------------------------" << endl;

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

{

if (HF[i]->weight == 0)

continue;

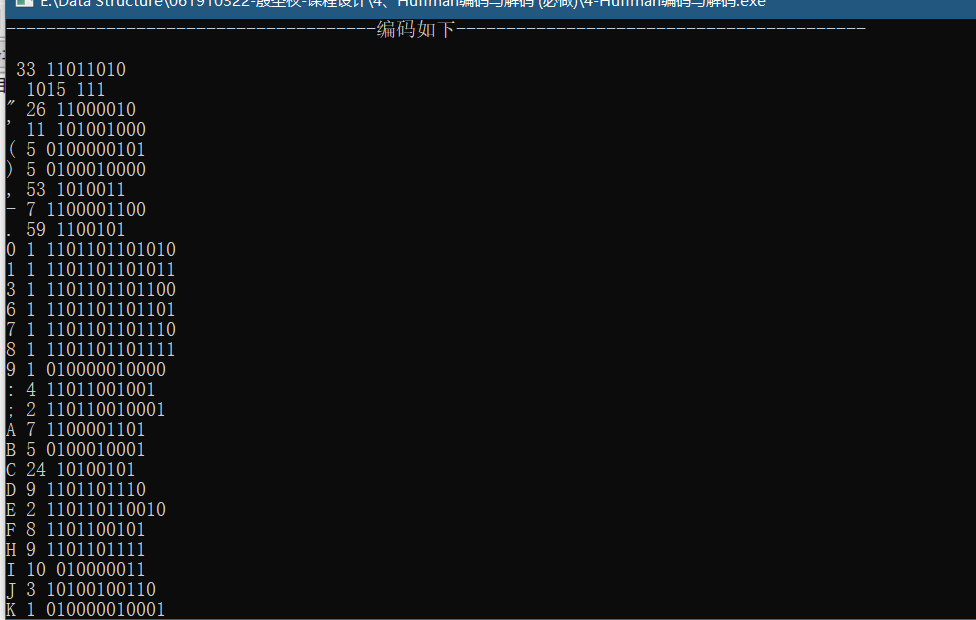
char c = HF[i]->position;

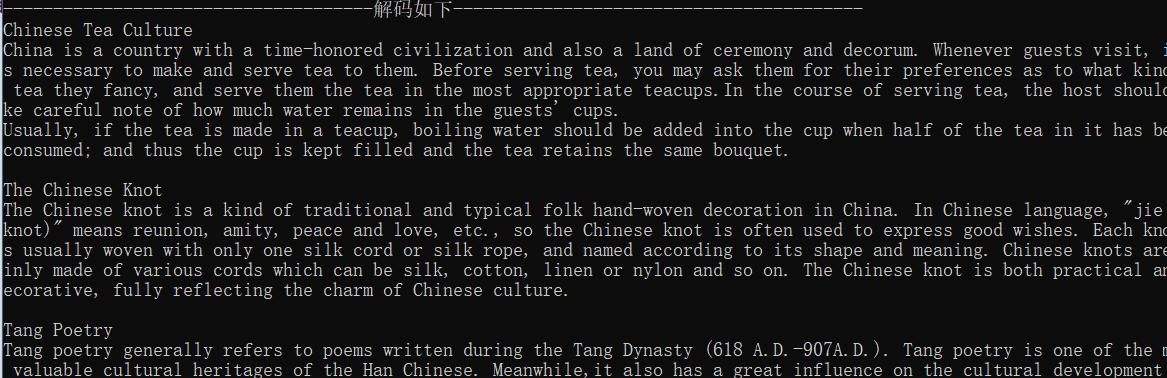
cout << c << " " << HF[i]->weight << " " << code[HF[i]->position] << endl;

}

}

1. 测试数据和结果：









1. 时间复杂度分析：

时间复杂度为：O(n^2)

# 地铁修建

[问题描述]

　　A市有n个交通枢纽，其中1号和n号非常重要，为了加强运输能力，A市决定在1号到n号枢纽间修建一条地铁。

　　地铁由很多段隧道组成，每段隧道连接两个交通枢纽。经过勘探，有m段隧道作为候选，两个交通枢纽之间最多只有一条候选的隧道，没有隧道两端连接着同一个交通枢纽。

　　现在有n家隧道施工的公司，每段候选的隧道只能由一个公司施工，每家公司施工需要的天数一致。而每家公司最多只能修建一条候选隧道。所有公司同时开始施工。

　　作为项目负责人，你获得了候选隧道的信息，现在你可以按自己的想法选择一部分隧道进行施工，请问修建整条地铁最少需要多少天。

输入格式

　　输入的第一行包含两个整数n, m，用一个空格分隔，分别表示交通枢纽的数量和候选隧道的数量。

第2行到第m+1行，每行包含三个整数a, b, c，表示枢纽a和枢纽b之间可以修建一条隧道，需要的时间为c天。

[基本要求]

输出格式

　　输出一个整数，修建整条地铁线路最少需要的天数。

样例输入

6 6

1 2 4

2 3 4

3 6 7

1 4 2

4 5 5

5 6 6

样例输出

6

样例说明

　　可以修建的线路有两种。

　　第一种经过的枢纽依次为1, 2, 3, 6，所需要的时间分别是4, 4, 7，则整条地铁线需要7天修完；

　　第二种经过的枢纽依次为1, 4, 5, 6，所需要的时间分别是2, 5, 6，则整条地铁线需要6天修完。

第二种方案所用的天数更少。

1. 数据结构：

图和数组

1. 算法设计思想：

先求出从1到n的所有路径，每求出一条路径就求出该条路径上的最大权值的边，然后和当前的最小天数对比，这样求出来的天数即为题目所求。

1. 源程序：

#include <iostream>

#include <cstdlib>

using namespace std;

#define MAX\_V 256

typedef struct edgenode

{

struct edgenode \*nextedge;

int adjvcode;

int weight;

}edgenode;

typedef struct vexnode

{

edgenode \*firstedge;

int data;

}vexnode;

typedef struct graph

{

vexnode adjlist[MAX\_V];

int vexnum, edgenum;

int type;

}graph;

void create\_graph(graph &G)

{

G.type = 0;

cin >> G.vexnum >> G.edgenum;

int i, j, k, weight;

edgenode \*p;

for(k=1; k<=G.vexnum; k++)

{

G.adjlist[k].data = k;

G.adjlist[k].firstedge = NULL;

}

for(k=1; k<=G.edgenum; k++)

{

cin >> i >> j >> weight;

p = (edgenode \*) malloc(sizeof(edgenode));

if(!p) exit(0);

p->adjvcode = j;

p->weight = weight;

p->nextedge = G.adjlist[i].firstedge;

G.adjlist[i].firstedge = p;

if(G.type == 0)

{

p = (edgenode \*) malloc(sizeof(edgenode));

if(!p) exit(0);

p->adjvcode = i;

p->weight = weight;

p->nextedge = G.adjlist[j].firstedge;

G.adjlist[j].firstedge = p;

}

}

}

int minnn = 9999;//当前最短天数

//求从顶点u出发到顶点v的所有简单路径

void DFS(graph G, int u, int v, int d, int \*visited, int \*path)

{

path[d] = u;

visited[u] = 1;

int i;

if(u == v)//找到路径，判断最长时间是否小于当前的最长时间，若是则替换

{

int day\_max = 0;

//for(i=1; i<=d; i++)

// cout << path[i];

//cout << endl;

i = 1;

while(i != d)

{

edgenode \*p = G.adjlist[path[i++]].firstedge;

while(p)

{

if(p->adjvcode == path[i])

{

break;

}

p = p->nextedge;

}

if(day\_max < p->weight )

day\_max = p->weight;//当前路径最长天数

}

if(day\_max < minnn)

minnn = day\_max;

}

else

{

edgenode \*p = G.adjlist[u].firstedge;

while(p)

{

if(!visited[p->adjvcode])

{

DFS(G, p->adjvcode, v, d+1, visited, path);

visited[p->adjvcode] = 0;

}

p = p->nextedge;

}

}

}

void all\_simple\_path(graph G)

{

int visited[MAX\_V], path[MAX\_V];

int i, u, v;

for(i=1; i<=G.vexnum; i++)

visited[i] = 0;

DFS(G, 1, G.vexnum, 1, visited, path);

}

int main()

{

graph G;

create\_graph(G);

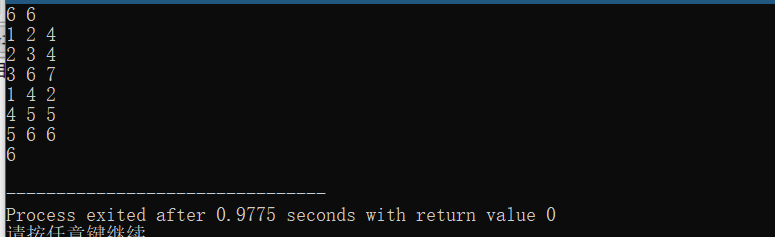
all\_simple\_path(G);

cout << minnn << endl;

return 0;

}

1. 测试数据和结果：



1. 时间复杂度分析：

基于广度优先遍历，最小天数算法的时间复杂度为O(n^2)

# 公交线路提示

[问题描述]

上网下载真实南京公交线路图，建立南京主要公交线路图的存储结构。

[基本要求]

（1）输入任意两站点，给出转车次数最少的乘车路线。

（2）输入任意两站点，给出经过站点最少的乘车路线。

1. 数据结构：

图、队列

1. 算法设计思想：

求转车次数最少和经过站点数量最少均可以使用广度优先遍历来求，而广度优先遍历用队列实现；

求转车数最少，即判断当前路线中是否有终点；

求经过站点数最少，即判断当前站点的前后站点是否有终点。

1. 源程序：

#include <stdio.h>

#include <windows.h>

typedef int Status;

typedef struct BusInfo

{

int No; //该公交车的编号

int passStationNum; //该公交车经过的站点数量

int line[80]; //该公交车的路线

} Bus;

typedef struct StationInfo

{

short int bus[35]; //经过该站点的公交车的编号

int passBusNum; //经过该站点的公交车数量

char name[60]; //站点名称

} Station;

typedef struct Map

{

Bus bus[1000]; //该城市的公交车

Station station[10000]; //该城市的站点

int busNum; //公交车数量

int stationNum; //站点数量

} Map;

typedef struct element//队列元素

{

int bus;

int before;

int cur;

} element;

typedef struct QNode

{

element data;

QNode \*next;

} QNode, \*QueuePtr;

typedef struct

{

QueuePtr front;

QueuePtr rear;

} LinkQueue;

Status InitQueue(LinkQueue &Q);

Status EnQueue(LinkQueue &Q, element e);

element DeQueue(LinkQueue &Q);

Status GetMap(Map &M);

Status InitMap(Map &M);

int locate\_station(Map &M, char name[30]);

int locate\_bus(Map &M, int No);

Status GetBusStation(Map &M, char temp[60]);

Status LeastTransfers(Map &M, char name\_1[60], char name\_2[60]);//最少转车

Status TraversalSite\_CD(Map &M, int v1, int v2);

Status LeastSite(Map &M, char name\_1[60], char name\_2[60]);//最少站点

Status TraversalSite\_AD(Map &M, int v1, int v2);

void Print(Map M);

int visited[10000] = {0};

int main()

{

Map M;

InitMap(M);

GetMap(M);

while (1)

{

system("cls");

char name\_1[60], name\_2[60];

printf("欢迎使用南京公交线路查询系统(如要退出系统,请输入退出)\n");

printf("请输入起点:");

scanf("%s", name\_1);

if (strcmp(name\_1, "退出") == 0)

break;

if (!locate\_station(M, name\_1))

{

printf("未找到该站点!\n");

system("pause");

continue;

}

printf("请输入终点:");

scanf("%s", name\_2);

if (strcmp(name\_2, "退出") == 0)

break;

if (!locate\_station(M, name\_2))

{

printf("未找到该站点!\n");

system("pause");

continue;

}

printf("\n最少站数路线\n");

LeastSite(M, name\_1, name\_2);

printf("\n最少转车路线\n");

LeastTransfers(M, name\_1, name\_2);

system("pause");

}

system("pause");

return 0;

}

//求经过站数最少 采用广度优先遍历算法，判断起点所涉及的每辆公交车经过的起点前后的站点，

//如果是终点，则经过站数最少。

//若无终点，则让起点所涉及的每辆公交车经过的起点前后的站点进入队列，

//通过Visited数组判断该站点是否已经访问过。

//队列元素出队，继续判断该点所涉及的每辆公交车经过的该点前后的站点，

//重复以上步骤，直至找到所需终点。

Status LeastSite(Map &M, char name\_1[60], char name\_2[60])

{

int v1 = locate\_station(M, name\_1);

for (int i = 1; i < 10000; i++)

{

visited[i] = 0;

}

int v2 = locate\_station(M, name\_2);

TraversalSite\_AD(M, v2, v1);

}

Status TraversalSite\_AD(Map &M, int v1, int v2)

{

element e[8000];

for (int i = 1; i < 8000; i++)

{

e[i].cur = i;

e[i].bus = 0;

}

element e0;

int count = 1;//统计站点数

LinkQueue Q;

InitQueue(Q);

EnQueue(Q, e[v1]);

visited[v1] = 1;

while (1)

{

int k = 1;

count++;

e0.cur = -1;

EnQueue(Q, e0);

while (Q.front != Q.rear)

{

v1 = DeQueue(Q).cur;

if (v1 == -1)

break;

//判断该站点前后的所有站点是否有终点，若有则得出最少站点路线

for (int i = 1; i <= M.station[v1].passBusNum; i++)

{

int bus1 = locate\_bus(M, M.station[v1].bus[i]);//v1经过的站点的公交车编号

for (int j = 1; j <= M.bus[bus1].passStationNum; j++)

{

if (M.bus[bus1].line[j] == v1)//该公交车路线上有v1

{

if (M.bus[bus1].line[j + 1] == v2 || M.bus[bus1].line[j - 1] == v2)//找到路线

{

int bus = e[v1].bus;

printf("最少经过%d个站点\n", count);

printf("%s(起点,乘坐%d路公交车)-->", M.station[v2].name, M.bus[bus1].No);

count--;

while (count--)

{

if (count == 0)

printf("%s(终点)\n\n", M.station[v1].name);

else

printf("%s(乘坐%d路公交车)-->", M.station[v1].name, bus);

v1 = e[v1].before;

bus = e[v1].bus;

}

return 1;

}

if (!visited[M.bus[bus1].line[j - 1]] && j - 1 > 0)

{

e[M.bus[bus1].line[j - 1]].bus = M.bus[bus1].No;

e[M.bus[bus1].line[j - 1]].before = v1;

EnQueue(Q, e[M.bus[bus1].line[j - 1]]);

visited[M.bus[bus1].line[j - 1]] = 1;

}

if (!visited[M.bus[bus1].line[j + 1]] && j + 1 <= M.bus[bus1].passStationNum)

{

e[M.bus[bus1].line[j + 1]].bus = M.bus[bus1].No;

e[M.bus[bus1].line[j + 1]].before = v1;

EnQueue(Q, e[M.bus[bus1].line[j + 1]]);

visited[M.bus[bus1].line[j + 1]] = 1;

}

break;

}

}

}

}

}

}

//求转车次数最少 采用广度优先遍历算法，判断起点所涉及的每辆公交车经过的所有站点，如果有终点，则无需转车。

//若无终点，则让起点所涉及的每辆公交车经过的所有站点进入队列，通过Visited数组判断该站点是否已经访问过。

//队列元素出队，继续判断该点所涉及的每辆公交车经过的所有站点，

//重复以上步骤，直至找到所需终点。

Status LeastTransfers(Map &M, char name\_1[60], char name\_2[60])

{

int v1 = locate\_station(M, name\_1);

for (int i = 1; i < 10000; i++)

{

visited[i] = 0;

}

int v2 = locate\_station(M, name\_2);

TraversalSite\_CD(M, v2, v1);

}

Status TraversalSite\_CD(Map &M, int v1, int v2)

{

element e[8000];

for (int i = 1; i < 8000; i++)

{

e[i].cur = i;

e[i].bus = 0;

}

element e0;

int count = 1;

LinkQueue Q;

InitQueue(Q);

EnQueue(Q, e[v1]);

visited[v1] = 1;

while (1)

{

int k = 1;

count++;

e0.cur = -1;

EnQueue(Q, e0);

while (Q.front != Q.rear)

{

v1 = DeQueue(Q).cur;

if (v1 == -1)

break;

//判断当前路线中是否有终点

for (int i = 1; i <= M.station[v1].passBusNum; i++)

{

int bus1 = locate\_bus(M, M.station[v1].bus[i]);

for (int j = 1; j <= M.bus[bus1].passStationNum; j++)

{

if (M.bus[bus1].line[j] == v2)//找到路线

{

int bus2;

bus2 = locate\_bus(M, e[v1].bus);

printf("最少转%d次车\n", count - 1);

printf("%s(起点,乘坐%d路公交车)-->", M.station[v2].name, M.station[v1].bus[i]);

--count;

while (count--)

{

if (count == 0)

printf("%s(终点)\n\n", M.station[v1].name);

else

printf("%s(换乘%d路公交车)-->", M.station[v1].name, M.bus[bus2].No);

v1 = e[v1].before;

bus2 = locate\_bus(M, e[v1].bus);

}

return 1;

}

if (!visited[M.bus[bus1].line[j]])

{

e[M.bus[bus1].line[j]].bus = M.bus[bus1].No;

e[M.bus[bus1].line[j]].before = v1;

EnQueue(Q, e[M.bus[bus1].line[j]]);

visited[M.bus[bus1].line[j]] = 1;

}

}

}

}

}

}

Status GetBusStation(Map &M, char temp[60])

{

int index = locate\_station(M, temp);

if (!index)

{

strcpy(M.station[++M.stationNum].name, temp);

index = M.stationNum;

}

M.bus[M.busNum].line[M.bus[M.busNum].passStationNum] = index;

M.station[index].bus[++M.station[index].passBusNum] = M.bus[M.busNum].No;//经过该站点的公交车编号

for (int i = 0; i < 60; i++)//清空字符串

{

temp[i] = '\0';

}

return index;

}

int locate\_station(Map &M, char name[60])//查找站点

{

for (int i = 1; i <= M.stationNum; i++)

{

if (strcmp(M.station[i].name, name) == 0)

return i;

}

return 0;

}

int locate\_bus(Map &M, int No)//查找公交车

{

for (int i = 1; i <= M.busNum; i++)

{

if (M.bus[i].No == No)

return i;

}

return 0;

}

void Print(Map M)

{

for (int i = 1; i <= M.busNum; i++)

{

printf("%d路\t", M.bus[i].No);

for (int j = 1; j <= M.bus[i].passStationNum; j++)

{

printf("%s ", M.station[M.bus[i].line[j]].name);

}

printf("\n");

}

}

Status InitMap(Map &M)

{

M.busNum = 0;

M.stationNum = 0;

return 1;

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

{

M.bus[i].passStationNum = 1;

}

for (int i = 0; i < 15000; i++)

{

M.station[i].passBusNum = 0;

}

}

Status GetMap(Map &M)

{

int k = 0;

char temp[60];//存储上一个字符

char ch;

FILE \*fp;

if ((fp = fopen("南京公交线路.txt", "r")) == NULL)

{

printf("无法代开文件！\n");

exit(0);

}

for (int i = 1; i <= 1000; i++)

{

M.bus[i].passStationNum = 1;

}

fscanf(fp, "%d", &M.bus[++M.busNum].No);

while (ch != ' ')

ch = fgetc(fp);

while (1)

{

ch = fgetc(fp);

if (feof(fp))//读取到最后一条路线

{

GetBusStation(M, temp);

break;

}

if (ch == '\n')

{

GetBusStation(M, temp);

k = 0;

char ch2;

ch2 = fgetc(fp);

if (feof(fp))

break;

fseek(fp, -1L, 1);

fscanf(fp, "%d", &M.bus[++M.busNum].No);

while (ch != ' ')

ch = fgetc(fp);

continue;

}

if (ch == ' ')

continue;

if (ch == ',')

{

GetBusStation(M, temp);

k = 0;

M.bus[M.busNum].passStationNum++;

continue;

}

temp[k++] = ch;

}

fclose(fp);

}

Status InitQueue(LinkQueue &Q)

{

Q.front = (QueuePtr)malloc(sizeof(QNode));

if (!Q.front)

exit(-2);

Q.front->next = NULL;

Q.rear = Q.front;

return true;

}

Status EnQueue(LinkQueue &Q, element e)

{

QueuePtr p = (QueuePtr)malloc(sizeof(QNode)); //开辟新结点

if (!p)

exit(-2);

p->data = e;

p->next = NULL;

Q.rear->next = p;

Q.rear = p;

return true;

}

element DeQueue(LinkQueue &Q)

{

element e;

if (Q.front == Q.rear)

{

printf("栈空");

system("pause");

exit(0);

}

QueuePtr p = Q.front->next;

e = p->data;

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

if (Q.rear == p)

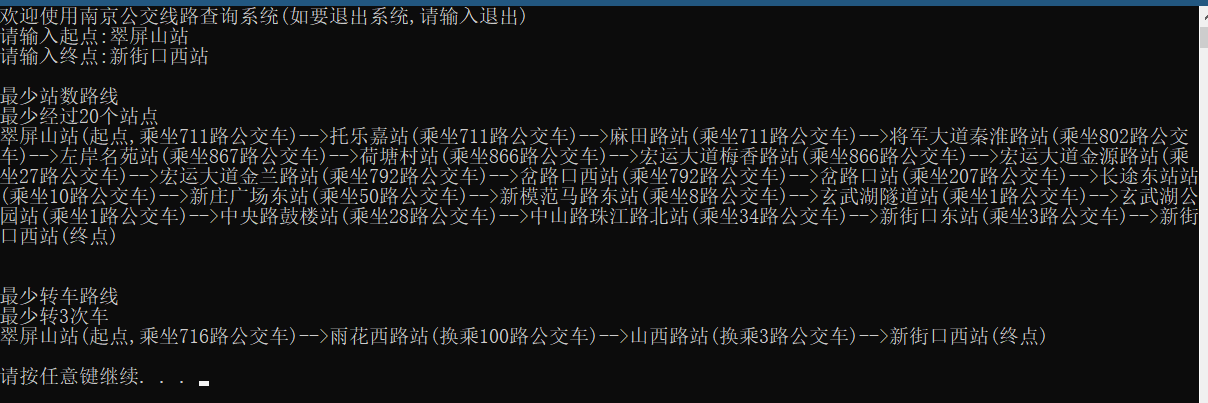
Q.rear = Q.front;

free(p);

return e;

}

1. 测试数据和结果：



1. 时间复杂度分析：

求解最少换乘次数算法以及最少乘车次数算法均为广度优先遍历的应用，判断当前路线上是否有终点和该站点前后是否有终点的时候时间复杂度为O(n^2)

# 平衡二叉树操作演示

[问题描述]

利用平衡二叉树实现一个动态查找表。

[基本要求]

（1） 从文件读取数据

（2）实现动态查找表的三种基本功能：查找、插入和删除。

（3）以可验证的方式输出结果

1. 数据结构：

平衡二叉树

1. 算法设计思想：

在树结点分配一个整型变量记录每个结点的平衡度，创建平衡二叉树以及插入或删除某元素的时候先找到每个结点在树上的位置，然后再根据平衡度是否满足平衡二叉树的定义来进行相应的左旋和右旋的操作。

1. 源程序：

#include<stdio.h>

#include<stdlib.h>

#define LH +1

#define EH 0

#define RH -1

#define NO 0

typedef struct BSTNode

{

int data;

int bf;//结点的平衡因子

struct BSTNode \*lchild,\*rchild;

}BSTNode,\*BSTree;

void CreatBST(BSTree &T);//创建二叉排序树

bool InsertAVL(BSTree &T,int e,bool &taller);//平衡化插入

void R\_Rotate (BSTree &p);//对以\*p为根的二叉排序树作右旋处理

void L\_Rotate(BSTree &p);//对以\*p为根的二叉排序树作左旋处理

void LeftBalance(BSTree &T);//对以指针Ｔ所指结点为根的二叉树作左平衡旋转处理

void RightBalance(BSTree &T);//对以指针Ｔ所指结点为根的二叉树作右平衡旋转处理

bool SearchBST(BSTree &T,int key);//查找

void LeftBalance\_div(BSTree &p,int &shorter);//删除结点时左平衡旋转处理

void RightBalance\_div(BSTree &p,int &shorter);//删除结点时右平衡旋转处理

void Delete(BSTree q,BSTree &r,int &shorter);//删除结点

int DeleteAVL(BSTree &p,int x,int &shorter);//平衡二叉树删除操作

void PrintBST(BSTree T,int depth);//打印树状图

int main()

{

BSTree T;

int sear,cmd,depth;

int ch;

int shorter=0;

bool taller=false;

T=(BSTree)malloc(sizeof(BSTNode));

T=NULL;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*菜单\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\* 1--创建 \*\n");

printf("\* 2--查找 \*\n");

printf("\* 3--插入 \*\n");

printf("\* 4--删除 \*\n");

printf("\* 0--退出 \*\n");

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

do

{

printf("\n请选择操作的编号：");

scanf("%d",&cmd);

getchar();

switch(cmd)

{

case 1:

CreatBST(T);break;

case 2:

printf("请输入您要查找的关键字：");

scanf("%d",&sear);

getchar();

if(SearchBST(T,sear))

{

printf("关键字%d存在，查找成功!\n",sear);

}

else printf("查找失败!\n");

break;

case 3:

printf("请输入您要插入的关键字：");

scanf("%d",&sear);

getchar();

InsertAVL(T,sear,taller);depth=0;

PrintBST(T,depth);

break;

case 4:

depth=0;

printf("请输入你要删除的关键字: ");

scanf("%d",&sear);

getchar();

DeleteAVL(T,sear,shorter);

PrintBST(T,depth);

break;

case 0:

printf("结束!\n");

return 0;

default:

printf("输入错误!\n");

}

}while(true);

}

void CreatBST(BSTree &T)

{

FILE \*fp = fopen("7.txt", "r");

if(!fp)

{

printf("文件打开失败！\n");

exit(0);

}

int depth;

int e;

bool taller = false;

T = NULL;

fscanf(fp, "%d", &e);

while(!feof(fp))

{

InsertAVL(T,e,taller);

fscanf(fp, "%d", &e);

taller = false;

}

depth = 0;

printf("您创建的二叉树为\n");

if(T)

PrintBST(T,depth);

else

printf("这是一棵空树!\n");

}

void R\_Rotate (BSTree &p)

{

BSTree lc;

lc=p->lchild;

p->lchild=lc->rchild;

lc->rchild=p;

p=lc;

}

void L\_Rotate(BSTree &p)

{

BSTree rc;

rc=p->rchild;

p->rchild=rc->lchild;

rc->lchild=p;

p=rc;

}

void LeftBalance(BSTree &T)

{

BSTree lc,rd;

lc=T->lchild;

switch(lc->bf)

{

case LH://原本左子树高于右子树

T->bf=lc->bf=EH;//现在设为等高

R\_Rotate(T);//右旋

break;

case RH://原本右子树高于左子树

rd=lc->rchild;//右子树

switch(rd->bf)

{

case LH://原本右子树的平衡度为1

T->bf=RH;

lc->bf=EH;

break;

case EH://原本右子树的平衡度为0

T->bf=lc->bf=EH;

break;

case RH://原本右子树的平衡度为-1

T->bf=EH;

lc->bf=LH;break;

}

rd->bf=EH;//将右子树平衡度设为0

L\_Rotate(T->lchild);//左旋左子树

R\_Rotate(T);//右旋

}

}

void RightBalance(BSTree &T)

{

BSTree rc,ld;

rc=T->rchild;

switch(rc->bf)

{

case RH:

T->bf=rc->bf=EH;

L\_Rotate(T);

break;

case LH:

ld=rc->lchild;

switch(ld->bf)

{

case RH:

T->bf=LH;

rc->bf=EH;

break;

case EH:

T->bf=rc->bf=EH;

break;

case LH:

T->bf=EH;

rc->bf=RH;

break;

}

ld->bf=EH;

R\_Rotate(T->rchild);

L\_Rotate(T);

}

}

bool InsertAVL(BSTree &T,int e,bool &taller)

{

if(!T)//若为空则直接插入，树长高，把taller设置为true

{

T=(BSTree)malloc(sizeof(BSTNode));

T->data=e;

T->lchild=T->rchild=NULL;

T->bf=EH;

taller=true;

}

else//否则

{

if(e==T->data)//若已存在该关键字，则不再插入

{

taller=false;

printf("已存在相同关键字的结点!\n");

return false;

}

if(e<T->data)//若值较小则继续在T的左子树中进行搜索

{

if(!InsertAVL(T->lchild,e,taller))//递归调用，在左子树中未插入

return false;//没有长高

if(taller)//已插入左子树并且左子树长高

switch(T->bf)//根据平衡度判断是否旋转

{

case LH://原本左子树比右子树高，需要作左平衡处理

LeftBalance(T);

taller=false;

break;

case EH: //原本左、右子树等高，现因左子树增高而使树增高，树依然在平衡状态

T->bf=LH;

taller=true;

break;

case RH://原本右子树比左子树高，现左、右子树等高，树依然在平衡状态

T->bf=EH;

taller=false;

break;

}

}

else//否则继续在T的右子树中进行搜索

{

if(!InsertAVL(T->rchild,e,taller))

return 0;

if(taller)

switch(T->bf)

{

case LH:

T->bf=EH;

taller=false;

break;

case EH:

T->bf=RH;

taller=true;

break;

case RH://原本右子树比左子树高，需要平衡化处理

RightBalance(T);

taller=false;

break;

}

}

}

return true;

}

bool SearchBST(BSTree &T,int key)

{

if(!T)

return false;

else if(key==T->data)

return true;

else if(key<T->data)

return SearchBST(T->lchild,key);

else

return SearchBST(T->rchild,key);

}

void LeftBalance\_div(BSTree &p,int &shorter)

{

BSTree p1,p2;

if(p->bf==1)

{

p->bf=0;

shorter=1;

}

else if(p->bf==0)

{

p->bf=-1;

shorter=0;

}

else

{

p1=p->rchild;

if(p1->bf==0)

{

L\_Rotate(p);

p1->bf=1;

p->bf=-1;

shorter=0;

}

else if(p1->bf==-1)

{

L\_Rotate(p);

p1->bf=p->bf=0;

shorter=1;

}

else

{

p2=p1->lchild;

p1->lchild=p2->rchild;

p2->rchild=p1;

p->rchild=p2->lchild;

p2->lchild=p;

if(p2->bf==0)

{

p->bf=0;

p1->bf=0;

}

else if(p2->bf==-1)

{

p->bf=1;

p1->bf=0;

}

else

{

p->bf=0;

p1->bf=-1;

}

p2->bf=0;

p=p2;

shorter=1;

}

}

}

void RightBalance\_div(BSTree &p,int &shorter)

{

BSTree p1,p2;

if(p->bf==-1)

{

p->bf=0;

shorter=1;

}

else if(p->bf==0)

{

p->bf=1;

shorter=0;

}

else

{

p1=p->lchild;

if(p1->bf==0)

{

R\_Rotate(p);

p1->bf=-1;

p->bf=1;

shorter=0;

}

else if(p1->bf==1)

{

R\_Rotate(p);

p1->bf=p->bf=0;

shorter=1;

}

else

{

p2=p1->rchild;

p1->rchild=p2->lchild;

p2->lchild=p1;

p->lchild=p2->rchild;

p2->rchild=p;

if(p2->bf==0)

{

p->bf=0;

p1->bf=0;

}

else if(p2->bf==1)

{

p->bf=-1;

p1->bf=0;

}

else

{

p->bf=0;

p1->bf=1;

}

p2->bf=0;

p=p2;

shorter=1;

}

}

}

void Delete(BSTree q,BSTree &r,int &shorter)

{

if(r->rchild==NULL)

{

q->data=r->data;

q=r;

r=r->lchild;

free(q);

shorter=1;

}

else

{

Delete(q,r->rchild,shorter);

if(shorter==1)

RightBalance\_div(r,shorter);

}

}

int DeleteAVL(BSTree &p,int x,int &shorter)

{

int k;

BSTree q;

if(p==NULL)

{

printf("不存在要删除的关键字!\n");

return 0;

}

else if(x<p->data)

{

k=DeleteAVL(p->lchild,x,shorter);

if(shorter==1)//树变矮

LeftBalance\_div(p,shorter);

return k;

}

else if(x>p->data)

{

k=DeleteAVL(p->rchild,x,shorter);

if(shorter==1)

RightBalance\_div(p,shorter);

return k;

}

else

{

q=p;

if(p->rchild==NULL)

{

p=p->lchild;

free(q);

shorter=1;

}

else if(p->lchild==NULL)

{

p=p->rchild;

free(q);

shorter=1;

}

else

{

Delete(q,q->lchild,shorter);

if(shorter==1)

LeftBalance\_div(p,shorter);

p=q;

}

return 1;

}

}

void PrintBST(BSTree T,int depth)

{

int i;

if(T->rchild)

PrintBST(T->rchild,depth+1);

for(i=1;i<=depth;i++)

printf(" ");

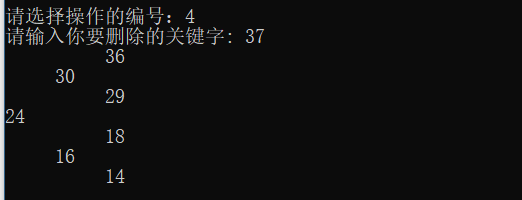
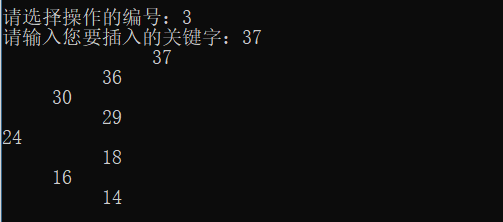
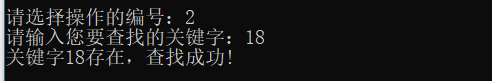
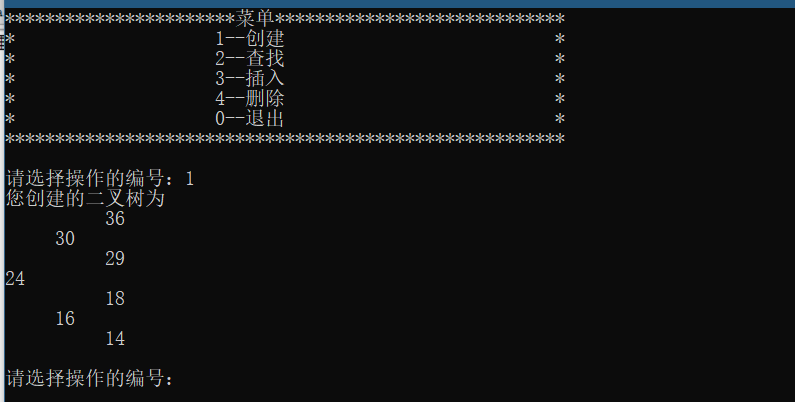
printf("%d\n",T->data);

if(T->lchild)

PrintBST(T->lchild,depth+1);

}

1. 测试数据和结果：



1. 时间复杂度分析：

时间主要耗费在左旋和右旋的操作上，这两个操作的时间复杂度均为O(n)

# 排序算法比较

[问题描述]

利用随机函数产生10个样本，每个样本有50000个随机整数（并使第一个样本是正序，第二个样本是逆序），利用直接插入排序、希尔排序，冒泡排序、快速排序、选择排序、堆排序，归并排序、基数排序8种排序方法进行排序（结果为由小到大的顺序），并统计每一种排序算法对不同样本所耗费的时间。

[基本要求]

（1） 原始数据存在文件中，用相同样本对不同算法进行测试；

（2） 屏幕显示每种排序算法对不同样本所花的时间；

1. 数据结构：

数组（用于存放待排序的序列）

1. 算法设计思想：

系统生成随机数形成随机序列样本，然后分别进行题目要求的排序算法，调用clock函数计算每个排序算法的时长，得出结果。

1. 源程序：

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

using namespace std;

void Swap\_SortUp(int \*a,int n)//交换排序排为升序

{

int i,j;

int temp;

for(i=0;i<n-1;i++)

{

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

{

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

{

temp=a[j];

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

a[j+1]=temp;

}

}

}

}

void Swap\_SortDown(int \*a,int n)//交换排序排为降序

{

int i,j;

int temp;

for(i=0;i<n-1;i++)

{

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

{

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

{

temp=a[j];

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

a[j+1]=temp;

}

}

}

}

void Specimen()//生成随机数

{

srand(time(NULL));

int i,j;

int a[20000];

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

{

FILE \*fp=NULL;

switch(i)

{

case 1:

fp=fopen("1.txt","w");

break;

case 2:

fp=fopen("2.txt","w");

break;

case 3:

fp=fopen("3.txt","w");

break;

case 4:

fp=fopen("4.txt","w");

break;

case 5:

fp=fopen("5.txt","w");

break;

case 6:

fp=fopen("6.txt","w");

break;

case 7:

fp=fopen("7.txt","w");

break;

case 8:

fp=fopen("8.txt","w");

break;

case 9:

fp=fopen("9.txt","w");

break;

case 10:

fp=fopen("10.txt","w");

break;

}

for(j=0;j<20000;j++)

{

a[j]=rand()%1000;

}

if(i==1) Swap\_SortUp(a,20000);

if(i==2) Swap\_SortDown(a,20000);

for(j=0;j<20000;j++)

{

fprintf(fp,"%d\n",a[j]);

}

fclose(fp);

}

}

//直接插入排序

void Insert\_Sort(int \*a,int n)

{

int i,j;

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

{

j = i;

while(j>0 && a[j]<a[j-1])

{

int temp;

temp=a[j];

a[j]=a[j-1];

a[j-1]=temp;

j--;

}

}

}

//希尔排序

void Shell\_Sort(int \*a,int len)

{

int i,j,temp,step;

for(step=len/2;step>0;step/=2)

{

for(i=step;i<len;i++)

{

temp=a[i];

j=i-step;

if(a[j]>temp)

{

a[j]=temp;

a[i]=a[j];

}

}

}

}

//冒泡排序

void Swap\_Sort(int \*a,int n)

{

int i,j;

int temp;

for(i=0;i<n-1;i++)

{

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

{

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

{

temp=a[j];

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

a[j+1]=temp;

}

}

}

}

//快速排序

void Quick\_Sort(int \*a,int left,int right)

{

if(left>=right)

{

return;

}

int i=left;

int j=right;

int key=a[left];

while(i<j)

{

while(i<j && key<=a[j])

{

j--;

}

a[i]=a[j];

while(i<j && key>=a[i])

{

i++;

}

a[j]=a[i];

}

a[i]=key;

Quick\_Sort(a,left,i-1);

Quick\_Sort(a,i+1,right);

}

//选择排序

void Select\_Sort(int \*a,int n)

{

int i,j,min,temp;

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

{

min=i;

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

{

if(a[j]<a[i])

min=j;

}

if(min!=i)

{

temp=a[i];

a[i]=a[min];

a[min]=temp;

}

}

}

//堆排序

void Swap(int \*a,int \*b)

{

int temp;

temp=\*a;

\*a=\*b;

\*b=temp;

}

void Max\_Heapify(int \*a,int start,int end)

{

int dad=start;

int son=dad\*2+1;

while(son<=end)

{

if(son+1<=end && a[son]<=a[son+1])

{

son++;

}

if(a[dad]>=a[son])

{

return;

}

else

{

a[dad]=a[son];

dad=son;

son=dad\*2+1;

}

}

}

void Heap\_Sort(int \*a,int len)

{

int i;

for(i=len/2-1;i>=0;i--)

{

Max\_Heapify(a,i,len-1);

}

for(i=len-1;i>=0;i--)

{

Swap(&a[0],&a[i]);

Max\_Heapify(a,0,i-1);

}

}

//归并排序

void Merge(int a[],int start,int mid,int end)

{

int \*temp=(int \*)malloc((end-start+1)\*(sizeof(int)));

int i=start;

int j=mid+1;

int k=0;

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

{

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

temp[k++]=a[i++];

else

temp[k++]=a[j++];

}

while(i<=mid)

temp[k++]=a[i++];

while(j<=end)

temp[k++]=a[j++];

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

{

a[start+i]=temp[i];

}

free(temp);

}

void Merge\_Sort(int a[], int start , int end)

{

if(a==NULL || start>=end)

return ;

int mid;

mid=(start+end)/2;

Merge\_Sort(a,start,mid);

Merge\_Sort(a,mid+1,end);

Merge(a,start,mid,end);

}

//基数排序

int Get\_max(int \*a,int n)

{

int i,max=a[0];

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

{

if(a[i]>max)

max=a[i];

}

return max;

}

void Count\_Sort(int \*a,int n,int exp)

{

int tmp[n];

int i,bucket[10]={0};

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

{

bucket[(a[i]/exp)%10]++;

}

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

{

bucket[i]+=bucket[i-1];

}

for(i=n-1;i>=0;i--)

{

tmp[bucket[(a[i]/exp)%10]-1]=a[i];

bucket[(a[i]/exp)%10]--;

}

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

{

a[i]=tmp[i];

}

}

void Radix\_Sort(int \*a,int n)

{

int exp;

int max=Get\_max(a,n);

for(exp=1;max/exp>0;exp \*=10)

{

Count\_Sort(a,n,exp);

}

}

int main()

{

Specimen();

int i,j,num;

clock\_t start,end;

int a1[20000],a2[20000],a3[20000],a4[20000],a5[20000],a6[20000],a7[20000],a8[20000];

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

{

FILE \*fp=NULL;

switch(i)

{

case 1:

printf("样本1各排序算法所花时间如下：\n");

fp = fopen("1.txt","r");

break;

case 2:

printf("样本2各排序算法所花时间如下：\n");

fp = fopen("2.txt","r");

break;

case 3:

printf("样本3各排序算法所花时间如下：\n");

fp = fopen("3.txt","r");

break;

case 4:

printf("样本4各排序算法所花时间如下：\n");

fp = fopen("4.txt","r");

break;

case 5:

printf("样本5各排序算法所花时间如下：\n");

fp = fopen("5.txt","r");

break;

case 6:

printf("样本6各排序算法所花时间如下：\n");

fp = fopen("6.txt","r");

break;

case 7:

printf("样本7各排序算法所花时间如下：\n");

fp = fopen("7.txt","r");

break;

case 8:

printf("样本8各排序算法所花时间如下：\n");

fp = fopen("8.txt","r");

break;

case 9:

printf("样本9各排序算法所花时间如下：\n");

fp = fopen("9.txt","r");

break;

case 10:

printf("样本10各排序算法所花时间如下：\n");

fp = fopen("10.txt","r");

break;

}

printf("\n");

for(j=0;j<20000;j++)

{

fscanf(fp,"%d",&num);

a1[j]=num;

a2[j]=num;

a3[j]=num;

a4[j]=num;

a5[j]=num;

a6[j]=num;

a7[j]=num;

a8[j]=num;

}

fclose(fp);

printf("直接插入排序：");

start=clock();//开始时间

Insert\_Sort(a1,20000);

end=clock();//结束时间

printf("%fms\n",(float)(end-start));

printf("希尔排序：");

start=clock();

Shell\_Sort(a2,20000);

end=clock();

printf("%fms\n",(float)(end-start));

printf("冒泡排序：");

start=clock();

Swap\_Sort(a3,20000);

end=clock();

printf("%fms\n",(float)(end-start));

printf("快速排序：");

start=clock();

Quick\_Sort(a4,0,19999);

end=clock();

printf("%fms\n",(float)(end-start));

printf("选择排序：");

start=clock();

Select\_Sort(a5,20000);

end=clock();

printf("%fms\n",(float)(end-start));

printf("堆排序：");

start=clock();

Heap\_Sort(a6,20000);

end=clock();

printf("%fms\n",(float)(end-start));

printf("归并排序：");

start=clock();

Merge\_Sort(a7,0,19999);

end=clock();

printf("%fms\n",(float)(end-start));

printf("基数排序：");

start=clock();

Radix\_Sort(a8,20000);

end=clock();

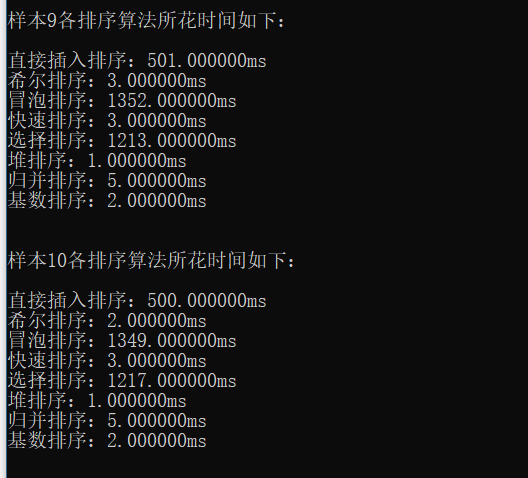
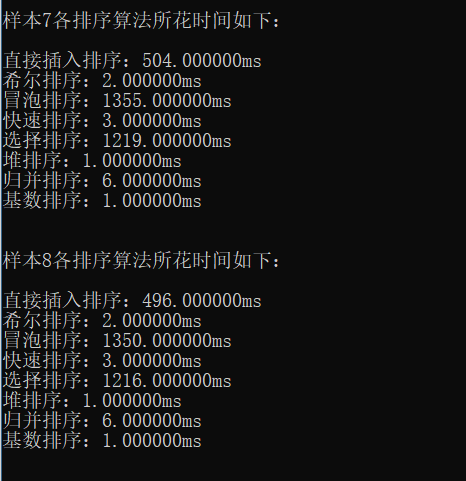
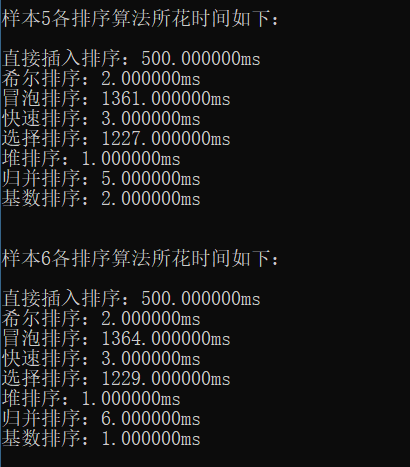
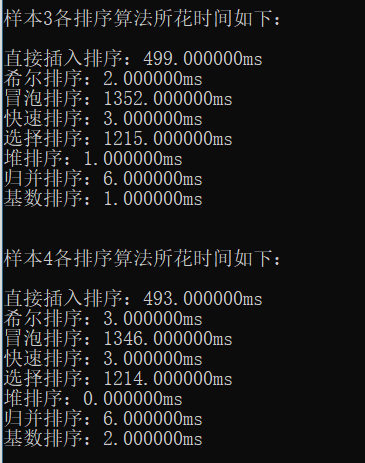
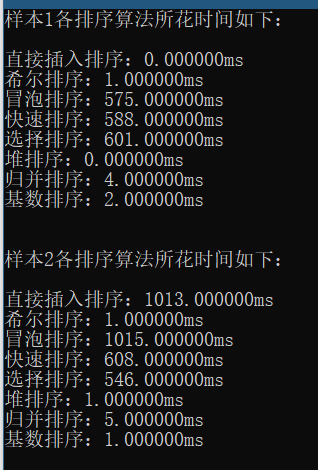
printf("%fms\n\n\n",(float)(end-start));

}

return 0;

}

1. 测试数据和结果：



1. 时间复杂度分析：

直接插入排序：将一个记录插入已排好序的有序表，从而得到新表。当待排序列为正序时，时间复杂度为O(1)；其他时候为O(n^2)

希尔排序：较直接插入排序更优，时间复杂度与其增量有关，大量研究表明当增量取到特定的值时，时间复杂度为O(n^1.5)

冒泡排序：进行n-1次冒泡，每次冒泡把最大值移至表尾，在待排序列不是有序的情况下冒泡排序是最低效的，时间复杂度为O(n^2)

快速排序：冒牌排序的改进，一次快速排序将序列分成两部分，然后再次快速排序直至最后序列有序。在同时间复杂度数量级的排序算法中平均性能最好，时间复杂度为O(nlogn)

简单选择排序：每次选择序列中最小值放在有序序列后。与冒泡排序性能差不多，但比冒泡排序略优，时间复杂度为O(n^2)

堆排序：简单选择排序的改进，省去了许多比较的次数，时间复杂度为O(nlogn)

归并排序：需要进行logn次归并，每次将两两子序列归并，时间复杂度为O(nlogn)

基数排序：按位数对每个记录进行分配和收集，时间复杂度为O(dn)，其中d为位数。

# 消除类游戏

[问题描述]

　　消除类游戏是深受大众欢迎的一种游戏，游戏在一个包含有n行m列的游戏棋盘上进行，棋盘的每一行每一列的方格上放着一个有颜色的棋子，当一行或一列上有连续三个或更多的相同颜色的棋子时，这些棋子都被消除。当有多处可以被消除时，这些地方的棋子将同时被消除。

　　现在给你一个n行m列的棋盘，棋盘中的每一个方格上有一个棋子，请给出经过一次消除后的棋盘。

　　请注意：一个棋子可能在某一行和某一列同时被消除。

输入格式

　　输入第一行包含两个整数n, m，用空格分隔，分别表示棋盘的行数和列数。

　　接下来n行，每行m个整数，用空格分隔，分别表示每一个方格中的棋子的颜色。颜色使用1至9编号。

输出格式

　　输出n行，每行m个整数，相邻的整数之间使用一个空格分隔，表示经过一次消除后的棋盘。如果一个方格中的棋子被消除，则对应的方格输出0，否则输出棋子的颜色编号。

样例输入

4 5

2 2 3 1 2

3 4 5 1 4

2 3 2 1 3

2 2 2 4 4

样例输出

2 2 3 0 2

3 4 5 0 4

2 3 2 0 3

0 0 0 4 4

样例说明

　　棋盘中第4列的1和第4行的2可以被消除，其他的方格中的棋子均保留。

样例输入

4 5

2 2 3 1 2

3 1 1 1 1

2 3 2 1 3

2 2 3 3 3

样例输出

2 2 3 0 2

3 0 0 0 0

2 3 2 0 3

2 2 0 0 0

样例说明

　　棋盘中所有的1以及最后一行的3可以被同时消除，其他的方格中的棋子均保留。

问题分析：这个问题与树无关，可以使用二维数组来存储，通过一遍遍历对符合条件的格子进行标记，然后第二遍遍历时消除符合条件的格子。

1. 数据结构：

二维数组

1. 算法设计思想：

先判断每一行是否有可以消除的，若有则设为负数(绝对值不变)，再判断每一列是否有可以消除的，若有则设为负数，最后把这些负数全部设为0，即表示消除完成。

1. 源程序：

#include <iostream>

#include <cmath>

using namespace std;

const int N = 30;

int grid[N][N], n, m;

int main()

{

// 初始化数据

cin >> n >> m;

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

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

{

cin >> grid[i][j];

}

//行标记（可以消除则置为负）

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

for(int j=0; j<m-2; j++)

{

if(abs(grid[i][j]) == abs(grid[i][j+1]) && abs(grid[i][j+1]) == abs(grid[i][j+2]))

{

if(grid[i][j] > 0)

grid[i][j] = - grid[i][j];

if(grid[i][j+1] > 0)

grid[i][j+1] = - grid[i][j+1];

if(grid[i][j+2] > 0)

grid[i][j+2] = - grid[i][j+2];

}

}

//列标记（可以消除则置为负）

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

for(int i=0; i<n-2; i++)

{

if(abs(grid[i][j]) == abs(grid[i+1][j]) && abs(grid[i+1][j]) == abs(grid[i+2][j]))

{

if(grid[i][j] > 0)

grid[i][j] = - grid[i][j];

if(grid[i+1][j] > 0)

grid[i+1][j] = - grid[i+1][j];

if(grid[i+2][j] > 0)

grid[i+2][j] = - grid[i+2][j];

}

}

// 进行清除（消除则置为0）并且输出结果

cout << "结果：" << endl;

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

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

{

if(grid[i][j] < 0)

grid[i][j] = 0;

if(j != 0)

cout << " ";

cout << grid[i][j];

}

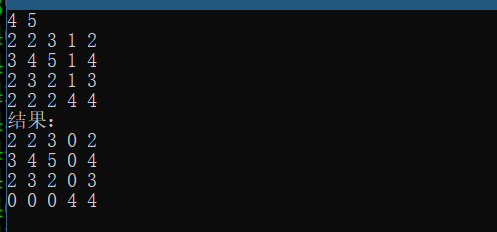
cout << endl;

}

return 0;

}

1. 测试数据和结果：



1. 时间复杂度分析：

需要遍历三次二维数组，时间复杂度为O(n^2)

# 公共钥匙盒

[问题描述]

　　有一个学校的老师共用N个教室，按照规定，所有的钥匙都必须放在公共钥匙盒里，老师不能带钥匙回家。每次老师上课前，都从公共钥匙盒里找到自己上课的教室的钥匙去开门，上完课后，再将钥匙放回到钥匙盒中。

　　钥匙盒一共有N个挂钩，从左到右排成一排，用来挂N个教室的钥匙。一串钥匙没有固定的悬挂位置，但钥匙上有标识，所以老师们不会弄混钥匙。

　　每次取钥匙的时候，老师们都会找到自己所需要的钥匙将其取走，而不会移动其他钥匙。每次还钥匙的时候，还钥匙的老师会找到最左边的空的挂钩，将钥匙挂在这个挂钩上。如果有多位老师还钥匙，则他们按钥匙编号从小到大的顺序还。如果同一时刻既有老师还钥匙又有老师取钥匙，则老师们会先将钥匙全还回去再取出。

　　今天开始的时候钥匙是按编号从小到大的顺序放在钥匙盒里的。有K位老师要上课，给出每位老师所需要的钥匙、开始上课的时间和上课的时长，假设下课时间就是还钥匙时间，请问最终钥匙盒里面钥匙的顺序是怎样的？

[基本要求]

输入格式

　　输入的第一行包含两个整数N, K。

　　接下来K行，每行三个整数w, s, c，分别表示一位老师要使用的钥匙编号、开始上课的时间和上课的时长。可能有多位老师使用同一把钥匙，但是老师使用钥匙的时间不会重叠。

　　保证输入数据满足输入格式，你不用检查数据合法性。

输出格式

　　输出一行，包含N个整数，相邻整数间用一个空格分隔，依次表示每个挂钩上挂的钥匙编号。

样例输入

5 2

4 3 3

2 2 7

样例输出

1 4 3 2 5

样例说明

　　第一位老师从时刻3开始使用4号教室的钥匙，使用3单位时间，所以在时刻6还钥匙。第二位老师从时刻2开始使用钥匙，使用7单位时间，所以在时刻9还钥匙。

　　每个关键时刻后的钥匙状态如下（X表示空）：

　　时刻2后为1X345；

　　时刻3后为1X3X5；

　　时刻6后为143X5；

　　时刻9后为14325。

课程设计要求：

（1）要求从文本文件中输入；

（2）根据时间进程，将取走钥匙和归还钥匙分别视为事件，放入队列中，然后通过每个事件的先后发生对钥匙盒的状态进行变更；

（3）严格按照要求的输入输出格式进行数据的输入、输出（训练CSP考试中的格式化输入输出的正确性）；

（4）选做：通过图形界面来显示钥匙盒的即时状态，以及事件队列的状态。

1. 数据结构：

链表，队列，数组

1. 算法设计思想：

先将数据排序，一组根据借走的时间排序，另一组根据归还的时间排序，然后创建两个链式队列依次入队，即一个队列按顺序存放借钥匙的信息，另一个队列按顺序存放还钥匙的信息。然后再进行借和还的操作，最后若还有为归还的钥匙，继续进行归还的操作直至钥匙全部归还。

1. 源程序：

#include<iostream>

#include<stdlib.h>

#include<stdio.h>

#include<algorithm>//使用sort函数

using namespace std;

struct key//钥匙以及借还信息

{

int identity;

int starttime;

int endtime;

};

struct keyinfo//钥匙是否归还

{

int identity;

bool flag;

};

typedef struct QueueNode//钥匙队列

{

key keymessage;

struct QueueNode\* next;

}QueueNode,\*Queueptr;

typedef struct//队列信息

{

Queueptr front;

Queueptr rear;

int length;

}LinkQueue;

key elem[256];

keyinfo kp[256];

int n, time, m;

int InitQueue(LinkQueue& Q)//初始化队列

{

Q.front = Q.rear = (QueueNode\*)malloc(sizeof(struct QueueNode));

if (!Q.front || !Q.rear)

{

cout << "申请空间失败！" << endl;

return 0;

}

Q.front->next = NULL;

Q.length = 0;

return 1;

}

int EnQueue(LinkQueue& Q, key e)//入队

{

QueueNode\* p = (QueueNode\*)malloc(sizeof(struct QueueNode));

if (!p)

{

cout << "空间申请失败" << endl;

return 0;

}

p->keymessage = e;

p->next = NULL;

Q.rear->next = p;

Q.rear = p;

Q.length++;

return 1;

}

int DeQueue(LinkQueue& Q, key& e)//出队

{

if (Q.rear == Q.front)

{

cout << "队列中不存在元素!" << endl;

return 0;

}

QueueNode\* p = (QueueNode\*)malloc(sizeof(struct QueueNode));

if (!p)

{

cout << "空间申请失败" << endl;

return 0;

}

p = Q.front->next;

e = p->keymessage;

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

if (Q.rear == p)

{

Q.rear = Q.front;

}

free(p);

Q.length--;

return 1;

}

int QueueTraverse(LinkQueue Q)//遍历队列

{

if (Q.rear == Q.front)

{

cout << "队列中无元素！" << endl;

return 0;

}

QueueNode\* p = (QueueNode\*)malloc(sizeof(struct QueueNode));

if (!p)

{

cout << "空间申请失败" << endl;

return 0;

}

p = Q.front->next;

while (p != NULL)

{

cout << p->keymessage.identity << " " << p->keymessage.starttime << " " << p->keymessage.endtime << endl;

p = p->next;

}

return 1;

}

bool QueueEmpty(LinkQueue Q)//判空

{

if (Q.front == Q.rear)

return true;

return false;

}

bool cmp\_starttime(const key &a,const key &b)//判断两把钥匙借走的时间

{

if (a.starttime == b.starttime)

return false;

return a.starttime < b.starttime;

}

bool cmp\_endtime(const key &a,const key &b)//判断两把钥匙归还的时间

{

if (a.endtime == b.endtime)

{

if (a.identity < b.identity)

return true;

return false;

}

return a.endtime < b.endtime;

}

void Initkeyinfo()//初始化归还信息，初始化钥匙全部都在

{

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

{

kp[i].flag = true;

kp[i].identity = i;

}

}

void PutKey(key a)//归还钥匙

{

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

{

if (kp[i].flag == false)//该处可以还钥匙

{

kp[i].identity = a.identity;

kp[i].flag = true;

i = n;

}

}

}

void TakeKey(key a)//借走钥匙

{

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

{

if (kp[i].identity == a.identity&&kp[i].flag==true)//该处可以借

{

kp[i].flag = false;

kp[i].identity = -1;//该位置为空

break;

}

}

}

void Traversekey()//输出钥匙顺序

{

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

{

cout << kp[i].identity << " ";

}

cout << kp[n].identity;

cout << endl;

}

void keymanage(LinkQueue& take, LinkQueue& back)//钥匙管理

{

int temp = 0;

key tk, bk;

DeQueue(take, tk);

DeQueue(back, bk);

//借和还

while ((!QueueEmpty(take) && !QueueEmpty(back))||temp==1)

{

if (tk.starttime < bk.endtime)

{

TakeKey(tk);

if (take.length == 0)

break;

DeQueue(take, tk);

if (take.length == 0)

temp = 1;//全部借出

}

else

{

PutKey(bk);

DeQueue(back, bk);

}

}

temp = 0;

//还

while (!QueueEmpty(back)||temp==1)

{

PutKey(bk);

if (temp == 1)

break;

DeQueue(back, bk);

if (back.length == 0)

temp = 1;//全部归还

}

}

int main()

{

//初始化

LinkQueue take,back;//拿钥匙队列和还钥匙队列

InitQueue(take);

InitQueue(back);

FILE \*fp = fopen("10.txt", "r");

if(!fp) exit(0);

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

Initkeyinfo();//初始化钥匙盒

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

{

fscanf(fp, "%d %d %d", &elem[i].identity, &elem[i].starttime, &time);

elem[i].endtime = elem[i].starttime + time;

}

sort(elem+1, elem + m+1, cmp\_starttime);//根据取走的时间，对钥匙进行排序然后依次入队

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

EnQueue(take, elem[i]);

sort(elem+1, elem + m+1, cmp\_endtime);//根据归还的时间，对钥匙进行排序然后入队

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

EnQueue(back, elem[i]);

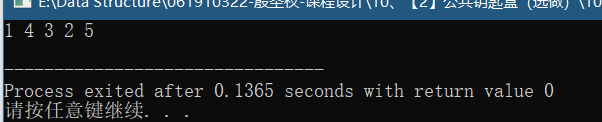
keymanage(take, back);//钥匙管理

Traversekey();//输出钥匙顺序

return 0;

}

1. 测试数据和结果：



五、时间复杂度分析：

Sort()排序算法时间复杂度为O(nlogn)，进行钥匙管理的时间复杂度为O(n)

# 行车路线

[问题描述]

　　小明和小芳出去乡村玩，小明负责开车，小芳来导航。

　　小芳将可能的道路分为大道和小道。大道比较好走，每走1公里小明会增加1的疲劳度。小道不好走，如果连续走小道，小明的疲劳值会快速增加，连续走s公里小明会增加s2的疲劳度。

　　例如：有5个路口，1号路口到2号路口为小道，2号路口到3号路口为小道，3号路口到4号路口为大道，4号路口到5号路口为小道，相邻路口之间的距离都是2公里。如果小明从1号路口到5号路口，则总疲劳值为(2+2)2+2+22=16+2+4=22。

现在小芳拿到了地图，请帮助她规划一个开车的路线，使得按这个路线开车小明的疲劳度最小。

[基本要求]

输入格式：

　　输入的第一行包含两个整数n, m，分别表示路口的数量和道路的数量。路口由1至n编号，小明需要开车从1号路口到n号路口。

接下来m行描述道路，每行包含四个整数t, a, b, c，表示一条类型为t，连接a与b两个路口，长度为c公里的双向道路。其中t为0表示大道，t为1表示小道。保证1号路口和n号路口是连通的。

输出格式

输出一个整数，表示最优路线下小明的疲劳度。

样例输入

6 7

1 1 2 3

1 2 3 2

0 1 3 30

0 3 4 20

0 4 5 30

1 3 5 6

1 5 6 1

样例输出

76

样例说明

　　从1走小道到2，再走小道到3，疲劳度为52=25；然后从3走大道经过4到达5，疲劳度为20+30=50；最后从5走小道到6，疲劳度为1。总共为76。

课程设计要求：

（1）要求从文本文件中输入；

（2）采用适当的数据结构存储由输入数据中的道路所形成的图结构；

（3）编写尽可能优的算法，处理好连续走小道造成的疲劳值的指数增长（提示：基于迪杰斯特拉算法进行改进即可完成本题）；

（4）除严格按题目要求进行输出以外，还要求输出最优路线的路径，以及从出发点到各个点的最小疲劳值。

1. 数据结构：

图、数组

1. 算法设计思想：

改变迪杰斯特拉算法，将两站之间的距离转化为疲劳度。记录上一条路径是大道还是小道，若为小道还需累加疲劳值。

1. 源程序：

#include<iostream>

#include<vector>

#include<math.h>

using namespace std;

struct tow

{

int name;

int type;

int distance;

};

struct Edge//边集

{

vector<tow> to;

int name;

};

struct Graph//图的邻接表结构

{

vector<Edge> e;

int vexnum, arcnum;

};

struct DjNode//迪杰斯特拉结点结构

{

int name;

int distance;//距离源点

int pre;//上一个点

int flag;//是否选中

int type;//上一条路的类型

int sroad;//累计疲劳值

};

DjNode\* DJ = new DjNode[5000];

void Dj(int start, Graph G)//迪杰斯特拉算法

{

//初始化Dj数组

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

{

DJ[i].distance = INT\_MAX - 1;

DJ[i].flag = false;

DJ[i].name = G.e[i].name;

DJ[i].pre = start;

DJ[i].type = 0;

DJ[i].sroad = 0;

}

DJ[start - 1].pre = start;

DJ[start - 1].flag = true;

DJ[start - 1].distance = 0;

for (int i = 0; i < G.e[start - 1].to.size(); i++)

{

if (G.e[start - 1].to[i].type == 0)//若为大道

{

if (DJ[start - 1].distance + G.e[start-1].to[i].distance < DJ[G.e[start - 1].to[i].name - 1].distance)

{

DJ[G.e[start - 1].to[i].name - 1].distance = DJ[start - 1].distance + G.e[start - 1].to[i].distance;

DJ[G.e[start - 1].to[i].name - 1].pre = start;

DJ[G.e[start - 1].to[i].name - 1].type = 0;

DJ[G.e[start - 1].to[i].name - 1].sroad = 0;

}

}

else//若为小道

{

if (DJ[start - 1].distance + G.e[start - 1].to[i].distance\*G.e[start-1].to[i].distance < DJ[G.e[start - 1].to[i].name - 1].distance)

{

DJ[G.e[start - 1].to[i].name - 1].distance = DJ[start - 1].distance + G.e[start - 1].to[i].distance \* G.e[start - 1].to[i].distance;

DJ[G.e[start - 1].to[i].name - 1].pre = start;

DJ[G.e[start - 1].to[i].name - 1].type = 1;

DJ[G.e[start - 1].to[i].name - 1].sroad = G.e[start - 1].to[i].distance \* G.e[start - 1].to[i].distance;

}

}

}

while (1)//循环选中剩余顶点

{

int minnposition = 0, minn = INT\_MAX;

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

{

if (DJ[i].flag == true)

continue;

if (DJ[i].distance < minn)//找出最小疲劳值

{

minn = DJ[i].distance;

minnposition = i;

}

}

DJ[minnposition].flag = true;//选中

//更新Dj数组

for (int i = 0; i < G.e[minnposition].to.size(); i++)

{

if (G.e[minnposition].to[i].type == 0)//当前为大道

{

if (DJ[minnposition].distance + G.e[minnposition].to[i].distance < DJ[G.e[minnposition].to[i].name - 1].distance)

{

//若当前距离小则赋值

DJ[G.e[minnposition].to[i].name - 1].distance = DJ[minnposition].distance + G.e[minnposition].to[i].distance;

//更新上一点

DJ[G.e[minnposition].to[i].name - 1].pre = minnposition+1;

//累计疲劳值清零

DJ[G.e[minnposition].to[i].name - 1].sroad = 0;

//标记下一条路的上一条路为大道

DJ[G.e[minnposition].to[i].name - 1].type = 0;

}

}

else if(G.e[minnposition].to[i].type==1&&DJ[minnposition].type==0)//当前为小道，上一次为大道

{

if (DJ[minnposition].distance + G.e[minnposition].to[i].distance \* G.e[minnposition].to[i].distance < DJ[G.e[minnposition].to[i].name - 1].distance)

{

DJ[G.e[minnposition].to[i].name - 1].distance = DJ[minnposition].distance + G.e[minnposition].to[i].distance \* G.e[minnposition].to[i].distance;

DJ[G.e[minnposition].to[i].name - 1].pre = minnposition+1;

//增加疲劳值

DJ[G.e[minnposition].to[i].name - 1].sroad = G.e[minnposition].to[i].distance \* G.e[minnposition].to[i].distance;

DJ[G.e[minnposition].to[i].name - 1].type = 1;

}

}

else if (G.e[minnposition].to[i].type == 1 && DJ[minnposition].type == 1)//当前和上一次均为小道

{

if (DJ[minnposition].distance-DJ[minnposition].sroad + (sqrt(DJ[minnposition].sroad) + G.e[minnposition].to[i].distance) \* (sqrt(DJ[minnposition].sroad) + G.e[minnposition].to[i].distance) < DJ[G.e[minnposition].to[i].name - 1].distance)

{

DJ[G.e[minnposition].to[i].name - 1].distance = DJ[minnposition].distance-DJ[minnposition].sroad + (sqrt(DJ[minnposition].sroad) + G.e[minnposition].to[i].distance) \* (sqrt(DJ[minnposition].sroad) + G.e[minnposition].to[i].distance);

DJ[G.e[minnposition].to[i].name - 1].pre = minnposition + 1;

//疲劳值累加

DJ[G.e[minnposition].to[i].name - 1].sroad = (sqrt(DJ[minnposition].sroad) + G.e[minnposition].to[i].distance) \* (sqrt(DJ[minnposition].sroad) + G.e[minnposition].to[i].distance);

DJ[G.e[minnposition].to[i].name - 1].type = 1;

}

}

}

//判断是否全部选中

int sum = 0;

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

{

if (DJ[i].flag == true)

{

sum++;

}

}

if (sum == G.vexnum)

break;

}

}

int main()

{

int n, m;

Graph G;

vector<tow> nope;

FILE \*fp = fopen("11.txt", "r");

if(!fp) exit(0);

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

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

{

G.e.push\_back({ nope,i + 1 });

}

G.vexnum = n;

G.arcnum = m;

while (m--)

{

int start, end,type,distance;

fscanf(fp, "%d %d %d %d", &type, &start, &end, &distance);

G.e[start - 1].to.push\_back({end,type,distance});

G.e[end - 1].to.push\_back({start,type,distance});

}

Dj(1,G);

cout << DJ[G.vexnum - 1].distance << endl;

vector<int> sta;

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

{

int temp = DJ[i].name;

while (DJ[temp - 1].pre != temp)

{

temp = DJ[temp - 1].pre;

sta.push\_back(temp);

}

//输出到达其他点的最小疲劳值

for (int i = sta.size() - 1; i >= 0; i--)

{

cout << sta[i] << "->";

}

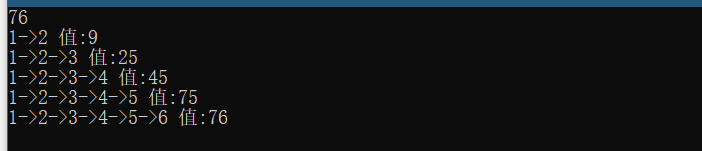
cout << DJ[i].name << " 值:" << DJ[i].distance << endl;

sta.clear();

}

}

1. 测试数据和结果：



1. 时间复杂度分析：

因为是迪杰斯特拉的改版，所以时间复杂度为O(n^2)

# 算术表达式求值

[问题描述]

　　一个算术表达式是由操作数(operand)、运算符(operator)和界限符(delimiter)组成的。假设操作数是正实数，运算符只含加减乘除等四种运算符，界限符有左右括号和表达式起始、结束符“#”，如：#6+15\*（21-8/4）#。引入表达式起始、结束符是为了方便。编程利用“运算符优先法”求算术表达式的值。

[基本要求]

（1） 从键盘或文件读入一个合法的算术表达式，输出正确的结果。

（2） 显示输入序列和栈的变化过程。

（3） 考虑算法的健壮性，当表达式错误时，要给出错误原因的提示。

（4） 实现非整数的处理（\*）。

1. 数据结构：

栈结构

1. 算法设计思想：

先把表达式转化为后缀表达式，即遇到数字，则直接放入后缀表达式字符串，遇到符号如果没有数字在栈中，则先入栈，然后遇到右括号或者#结束标志一并输入到后缀表达式字符串中。后缀表达式求解即先数字进栈，如果碰到符号则两数字出栈运算后在进栈。另外注意空格的存放。

1. 源程序：

#include<iostream>

#include<stack>

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

#define stacksize 100

#define add 10

using namespace std;

typedef struct SNode

{

char\* base;

char\* top;

int Stacksize;

int length;

}SNode, \* SqStack;

stack<double> nu;//储存计算结果

char input[1000000], output[1000000];//储存初始表达式和后缀表达式

int InitStack(SqStack& S)

{

S = (SNode\*)malloc(sizeof(SNode));

S->Stacksize = stacksize;

S->base = (char\*)malloc(sizeof(char) \* stacksize);

S->top = S->base;

S->length = 0;

return 1;

}

int Push(SqStack& S, char e)

{

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

{

S->base = (char\*)realloc(S->base, sizeof(char) \* (stacksize + add));

S->Stacksize += add;

}

\*(S->top) = e;

S->top++;

S->length++;

return 1;

}

int Pop(SqStack& S, char& e)

{

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

return 0;

e = \*(S->top - 1);

S->top--;

S->length--;

return 1;

}

int shift(SqStack& op)//转化为后缀表达式存在output中

{

int flag = 0;

int i = 0, j = 0, black\_white = 1;

char c;

while (input[i] != '\0')

{

if (input[i] >= '0' && input[i] <= '9' || input[i] == '.')//若是数字

{

if (input[i] == '.')

flag = 1;

output[j] = input[i];

i++;

j++;

if (input[i] == '.' && flag == 1)

{

cout << "小数点重复出现！格式错误！" << endl;

return -1;

}

if ((input[i]<'0' || input[i]>'9') && input[i] != '.')

{

flag = 0;

output[j] = ' ';

j++;

}

}

else if (input[i] == '+')

{

if (op->length == 0)//没有数字在栈中

{

Push(op, input[i]);

}

else//否则

{

do

{

Pop(op, c);

if (c == '(')

{

Push(op, c);

continue;

}

output[j++] = c;

output[j++] = ' ';

} while (op->length != 0 && c != '(');

Push(op, input[i]);

}

i++;

}

else if (input[i] == '-' && ((input[i - 1] >= '0' && input[i - 1] <= '9') || input[i - 1] == ')'))

{

if (op->length == 0)

{

Push(op, input[i]);

}

else

{

do

{

Pop(op, c);

if (c == '(')

{

Push(op, c);

continue;

}

output[j++] = c;

output[j++] = ' ';

} while (op->length != 0 && c != '(');

Push(op, input[i]);

}

i++;

}

else if ((input[i] == '-' && i == 0) || (input[i] == '-' && (input[i - 1] == '(' || input[i - 1] == '+' || input[i - 1] == '-' || input[i - 1] == '\*' || input[i - 1] == '/')))

{

output[j] = input[i];

j++;

i++;

}

else if (input[i] == '\*' || input[i] == '/')

{

if (op->length == 0)

{

Push(op, input[i]);

}

else

{

do

{

Pop(op, c);

if (c == '(')

{

Push(op, c);

continue;

}

if (c == '+' || c == '-')

{

Push(op, c);

break;

}

else if (c == '\*' || c == '/')

{

Push(op, c);

break;

}

} while (op->length != 0 && c != '(');

Push(op, input[i]);

}

i++;

}

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

{

Push(op, input[i]);

i++;

}

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

{

Pop(op, c);

while (c != '(')

{

output[j] = c;

j++;

output[j] = ' ';

j++;

Pop(op, c);

}

i++;

}

else if (input[i] == '#')

{

if (op->length == 0)

break;

else

{

Pop(op, c);

output[j++] = c;

output[j++] = ' ';

}

}

}

}

double calculate()//根据后缀表达式进行按顺序计算

{

int i = 0, j = 0;

double t = 1;

if (output[0] == '-')

{

t = -1;

i++;

}

char num[100];

double cal = 0;

while (output[i + 1] != '\0')

{

if (output[i] >= '0' && output[i] <= '9' || output[i] == '.')

{

num[j++] = output[i];

i++;

if (output[i] == ' ')

{

if (i >= 100)

{

cout << "数字过大无法计算！" << endl;

return -1;

}

double a = atof(num);

nu.push(a \* t);

memset(num, '\0', 100);

j = 0;

}

}

else if (output[i] == ' ' && output[i + 1] >= '0' && output[i + 1] <= '9')

{

i++;

t = 1;

continue;

}

else if (output[i] == ' ' && output[i + 1] == '-' && output[i + 2] != ' ')

{

i++;

i++;

t = -1;

continue;

}

else if (output[i] == ' ' && ((output[i + 1] == '-' && output[i + 2] == ' ') || output[i + 1] == '+' || output[i + 1] == '\*' || output[i + 1] == '/' || (output[i + 1] >= '0' && output[i + 1] <= '9')))

{

i++;

continue;

}

else if (output[i] == '+')

{

double a, b;

if (nu.empty())

{

cout << "输入格式错误！数字缺失！" << endl;

return -1;

}

a = nu.top();

nu.pop();

if (nu.empty())

{

cout << "输入格式错误！数字缺失！" << endl;

return -1;

}

b = nu.top();

nu.pop();

nu.push(a + b);

i++;

cout << a << "+" << b << "=" << a + b << endl;

}

else if (output[i] == '/')

{

double a, b;

if (nu.empty())

{

cout << "输入格式错误！数字缺失！" << endl;

return -1;

}

a = nu.top();

nu.pop();

if (nu.empty())

{

cout << "输入格式错误！数字缺失！" << endl;

return -1;

}

b = nu.top();

nu.pop();

if (a == 0)

{

cout << "除数不能为0" << endl;

return -1;

}

nu.push(b / a);

i++;

cout << b << "/" << a << "=" << b / a << endl;

}

else if (output[i] == '-')

{

double a, b;

if (nu.empty())

{

cout << "输入格式错误！数字缺失！" << endl;

return -1;

}

a = nu.top();

nu.pop();

if (nu.empty())

{

cout << "输入格式错误！数字缺失！" << endl;

return -1;

}

b = nu.top();

nu.pop();

nu.push(b - a);

i++;

cout << b << "-" << a << "=" << b - a << endl;

}

else if (output[i] == '\*')

{

double a, b;

if (nu.empty())

{

cout << "输入格式错误！数字缺失！" << endl;

return -1;

}

a = nu.top();

nu.pop();

if (nu.empty())

{

cout << "输入格式错误！数字缺失！" << endl;

return -1;

}

b = nu.top();

nu.pop();

nu.push(a \* b);

i++;

cout << b << "\*" << a << "=" << a \* b << endl;

}

}

return nu.top();

}

int main()

{

SqStack op;

InitStack(op);

cin >> input;

shift(op);

cout << output << endl;

double cal = calculate();

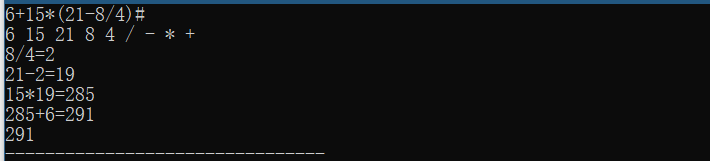
cout << cal;

delete op;

return 0;

}

1. 测试数据和结果：



1. 时间复杂度分析：

时间复杂度为O(n)，n为字符串长度。

# 树的应用

[问题描述]

　　JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式，可以用来描述半结构化的数据。JSON 格式中的基本单元是值 (value)，出于简化的目的本题只涉及 2 种类型的值：

　　\* 字符串 (string)：字符串是由双引号 " 括起来的一组字符（可以为空）。如果字符串的内容中出现双引号 "，在双引号前面加反斜杠，也就是用 \" 表示；如果出现反斜杠 \，则用两个反斜杠 \\ 表示。反斜杠后面不能出现 " 和 \ 以外的字符。例如：""、"hello"、"\"\\"。

　　\* 对象 (object)：对象是一组键值对的无序集合（可以为空）。键值对表示对象的属性，键是属性名，值是属性的内容。对象以左花括号 { 开始，右花括号 } 结束，键值对之间以逗号 , 分隔。一个键值对的键和值之间以冒号 : 分隔。键必须是字符串，同一个对象所有键值对的键必须两两都不相同；值可以是字符串，也可以是另一个对象。例如：{}、{"foo": "bar"}、{"Mon": "weekday", "Tue": "weekday", "Sun": "weekend"}。

　　除了字符串内部的位置，其他位置都可以插入一个或多个空格使得 JSON 的呈现更加美观，也可以在一些地方换行，不会影响所表示的数据内容。例如，上面举例的最后一个 JSON 数据也可以写成如下形式。

　　{

　　"Mon": "weekday",

　　"Tue": "weekday",

　　"Sun": "weekend"

　　}

　　给出一个 JSON 格式描述的数据，以及若干查询，编程返回这些查询的结果。

输入格式

　　第一行是两个正整数 n 和 m，分别表示 JSON 数据的行数和查询的个数。

　　接下来 n 行，描述一个 JSON 数据，保证输入是一个合法的 JSON 对象。

　　接下来 m 行，每行描述一个查询。给出要查询的属性名，要求返回对应属性的内容。需要支持多层查询，各层的属性名之间用小数点 . 连接。保证查询的格式都是合法的。

[基本要求]

输出格式

　　对于输入的每一个查询，按顺序输出查询结果，每个结果占一行。

　　如果查询结果是一个字符串，则输出 STRING <string>，其中 <string> 是字符串的值，中间用一个空格分隔。

　　如果查询结果是一个对象，则输出 OBJECT，不需要输出对象的内容。

　　如果查询结果不存在，则输出 NOTEXIST。

样例输入

10 5

{

"firstName": "John",

"lastName": "Smith",

"address": {

"streetAddress": "2ndStreet",

"city": "NewYork",

"state": "NY"

},

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

}

firstName

address

address.city

address.postal

esc\aped

样例输出

STRING John

OBJECT

STRING NewYork

NOTEXIST

STRING "hello"

[基本要求]

（1）要求从文本文件中输入；

（2）本题目其实就是一棵普通的树（即每个结点的孩子数不固定，不能单纯采用n叉树来解决），可以考虑使用孩子兄弟表示法等进行表示和存储；

（3）严格按照要求的输入输出格式进行数据的输入、输出（训练CSP考试中的格式化输入输出的正确性）；

（4）选做：使用图形界面（或字符格式化摆成的树形结构，参考Linux下的tree命令），以树状形式显示输入的JSON格式数据。

1. 数据结构：

字符串和数组

1. 算法设计思想：

使用了map关联容器，当读取到某个字符串的时候对应输出所要求的内容。对文本的标点进行操作，截取文本中的属性以及对应的值存在map中。

1. 源程序：

#include <iostream>

#include <stdio.h>

#include <string>

#include <string.h>

#include <map>

using namespace std;

int n, m;

string s, key;

char \*s1 = new char[1024], \*key1;

map<string, string> json;

int state;

void handle(string &s)

{

for(int i = 0; s[i]; i++)

{

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

{

if(state == 0)//第一行

{

key = "";

}

else//否则为对象

{

json[key] = "OBJECT";

}

state = 1;

}

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

{

key = "";

}

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

{

state = 2;

}

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

{

state = 1;

}

if(s[i] == '"')// 双引号

{

string tmp;

for(i++; s[i]; i++)

{

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

tmp += s[++i];

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

break;

else

tmp += s[i];

}

if(state == 1)

{

if(key == "")

key = tmp;

else//对象内元素

key += '.' + tmp;

state = 2;

}

else if(state == 2)

{

json[key] = "STRING " + tmp;

int j;

for(j = (int)key.size() - 1; j >= 0; j--)

{

if(key[j] == '.')

break;

}

if(j >= 0)

key = key.substr(0, j);

else

key = "";

state = 1;

}

}

}

}

int main()

{

FILE \*fp = fopen("13.txt", "r");

if(!fp)

{

cout << "文件打开失败！" << endl;

exit(0);

}

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

fgets(s1, 1024, fp);

int i = strlen(s1);

s1[strlen(s1) - 1] = '\0';

s = s1;

state = 0;//初始化为0

while(n--)

{

fgets(s1, 1024, fp);

int i = strlen(s1);

s1[strlen(s1) - 1] = '\0';

s = s1;

handle(s);

}

while(m--)

{

fgets(s1, 1024, fp);

if(m == 0)

{

s = s1;

cout << (json[s] == "" ? "NOTEXIST" : json[s]) << endl;

continue;

}

int i = strlen(s1);

s1[strlen(s1) - 1] = '\0';

s = s1;

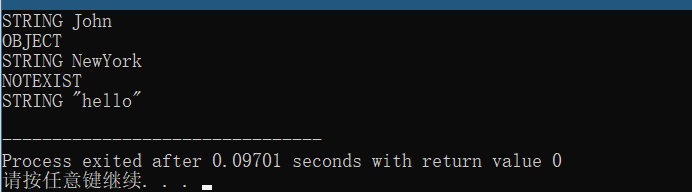
cout << (json[s] == "" ? "NOTEXIST" : json[s]) << endl;

}

return 0;

}

1. 测试数据和结果：



1. 时间复杂度分析：

时间复杂度为O(n)

# 最小生成树

[问题描述]

利用普利姆算法和克鲁斯卡尔算法实现最小生成树问题。

[基本要求]

（1）自行建立图的数据文件，第一行是顶点个数，然后依次是顶点名，接下来是边，用float表示边的权值；

（2）以邻接表或者邻接矩阵表示图皆可；

（3）分别利用Prim和Kruskal算法实现最小生成树；

（4）输出最小生成树的权值之和，及所用的边。

1. 数据结构：

图的邻接矩阵结构

1. 算法设计思想：

Prim：每次选择所有连接在树上和未连接在树上的点之间的边权值最小者，当所有点都在树上时，算法结束。

Kruskal：将边按权值大小升序排序，并依次选择一条边，若选中该边后未构成回路，则选边成功，当所有点都在树上时，算法结束。注意要设置一个并查集，标记每个在树上的点，以此来判断是否会构成回路。

1. 源程序：

1、

#include <iostream>

#include <cstdlib>

#include <stdio.h>

using namespace std;

#define MAX\_V 256

#define INFINITY 9999

typedef struct mGraph

{

int vexName[MAX\_V];

int adjMatrix[MAX\_V][MAX\_V];

int vexNum, eNum;

}mGraph;

typedef struct closest//辅助结构

{

int vexcode;

int lowcost;

}closest;

void create\_mGraph(mGraph &G)

{

FILE \*fp = fopen("14.txt", "r");

if(!fp)

{

cout << "文件打开失败！\n";

exit(0);

}

fscanf(fp, "%d %d", &G.vexNum, &G.eNum);

int i, j, weight, k;

for(i=1; i<=G.vexNum; i++)

{

for(j=1; j<=G.vexNum; j++)

{

if(i != j)

G.adjMatrix[i][j] = INFINITY;

else

G.adjMatrix[i][j] = 0;

}

}

for(k=1; k<=G.eNum; k++)

{

G.vexName[k] = k;

fscanf(fp, "%d %d %d", &i, &j, &weight);

G.adjMatrix[i][j] = weight;

G.adjMatrix[j][i] = weight;

}

fclose(fp);

}

void minSpanningTree\_prim(mGraph G, int v)//利用普利姆算法，从顶点v出发求最小生成树

{

printf("Prim：\n");

int i, j, k, min, min\_all = 0;

closest clo[MAX\_V];//初始化clo数组

for(j=1; j<=G.vexNum; j++)

{

clo[j].vexcode = v;

clo[j].lowcost = G.adjMatrix[v][j];

}

clo[v].lowcost = 0;//标记v已经被选中

for(i=2; i<=G.vexNum; i++)//将剩余的顶点加入最小生成树

{

min = INFINITY;

for(j=1; j<=G.vexNum; j++)//在clo数组中寻找lowcost非零的最小值

{

if(clo[j].lowcost && clo[j].lowcost < min)

{

min = clo[j].lowcost;

k = j;

}

}

printf("%d->%d %d\n",k,clo[k].vexcode,G.adjMatrix[k][clo[k].vexcode]);

min\_all += min;

clo[k].lowcost = 0;//选中本轮中lowcost最小的顶点

for(j=1; j<=G.vexNum; j++)//更新clo数组

{

if(G.adjMatrix[k][j] < clo[j].lowcost)//不在树上的顶点到顶点k的距离比原来的lowcost小

{

clo[j].lowcost = G.adjMatrix[k][j];

clo[j].vexcode = k;

}

}

}

printf("权值之和为：%d\n", min\_all);

}

int main()

{

mGraph G;

create\_mGraph(G);

minSpanningTree\_prim(G, 1);

return 0;

}

2、

#include <iostream>

#include <cstdlib>

#include <stdio.h>

using namespace std;

#define MAX\_V 256

typedef struct Edge//v1,v2存顶点，weight存权重。

{

int v1;

int v2;

int weight;

}Edge;

typedef struct mGraph

{

int vexNum, eNum;

Edge \*p;//边集数组

int vexName[MAX\_V];

}mGraph;

void create\_mGraph(mGraph &G)

{

FILE \*fp = fopen("14.txt", "r");

if(!fp)

{

cout << "文件打开失败！\n";

exit(0);

}

fscanf(fp, "%d %d", &G.vexNum, &G.eNum);

int i,j;

G.p = (Edge \*)malloc(sizeof(Edge)\*(G.eNum+1));

for(i=0; i<G.eNum; i++)

{

G.vexName[i] = i+1;

fscanf(fp, "%d %d %d", &G.p[i].v1, &G.p[i].v2, &G.p[i].weight);

}

for(i=0; i<G.eNum; i++)//冒泡排序法，权重从小到大存在边集数组中

{

for(j=G.eNum-1; j>i; j--)

{

if(G.p[i].weight > G.p[j].weight)

{

G.p[G.eNum] = G.p[i];

G.p[i] = G.p[j];

G.p[j] = G.p[G.eNum];

}

}

}

}

int Find(int \*parent, int g) //返回该点对应的标记，以此判断是否有回路

{

while(parent[g] != 0)

{

g = parent[g];

}

return g;

}

int Finish(mGraph G,int \*parent)//判断生成树是否完成，完成的标志是生成树的边等于顶点的数量减1

{

int i,n = 0;

for(i=0; i<G.vexNum; i++)

{

if(parent[i])

{

n++;

}

}

if(n == G.vexNum-1)

{

return 1;

}

return 0;

}

void MinTree\_Kruskal(mGraph G)

{

printf("Kruskal：\n");

int i, a, b;

int min\_all = 0;

int parent[G.vexNum];//对每个点标记为父结点的编号

for(i=0; i<G.vexNum; i++)//初始化parent[]

{

parent[i] = 0;

}

for(i=0; i<G.eNum; i++)

{

a = Find(parent,G.p[i].v1);

b = Find(parent,G.p[i].v2);

if(a != b)//如果a与b相等则代表a和b在同一颗生成树上，如果a和b连接则为生成环，不符合生成树

{

parent[a] = b;//标记

printf("%d->%d %d\n", G.p[i].v1, G.p[i].v2, G.p[i].weight);

min\_all += G.p[i].weight;

}

if(Finish(G,parent))//完成后返回

{

printf("权值之和为：%d\n", min\_all);

return ;

}

}

}

int main()

{

mGraph G;

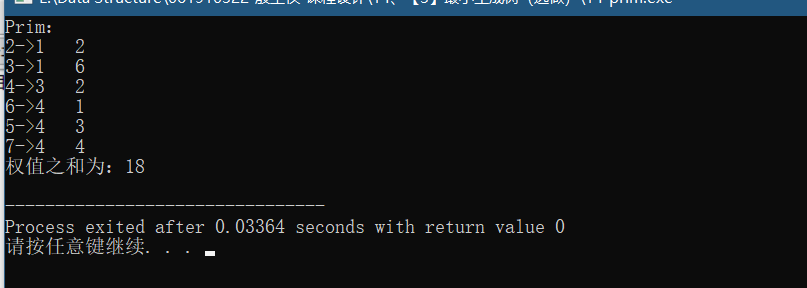
create\_mGraph(G);

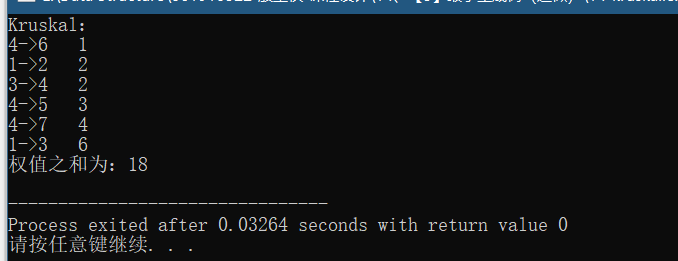
MinTree\_Kruskal(G);

return 0;

}

1. 测试数据和结果：





1. 时间复杂度分析：

Prim算法内的核心部分有个双重循环，因此它的时间复杂度为O(n^2)；Kruscal算法的时间复杂度为O(nlogn)，n为图中的边的数量。

# 心得体会

本次数据结构课程设计，真的是极大的磨练了我的程序设计能力以及对我C/C++的各种知识的掌握起到了非常大的作用。一开始，看到课程设计题的时候，我无从下手，毫无头绪，然后呢我也是硬着头皮开始着手设计第一题的数据结构，第一题是系统进程统计，涉及调用系统进程的相关知识，这一上来就给我来了个下马威，我花了很长的时间在网上寻找调用系统进程的相关操作，但是结果不尽人意，后来经过同学的帮助，我才学会了如何便捷地调用系统内存和时间，做完了第一题我就感觉自己对C/C++的很多语法都十分模糊，不管是vector还是map，甚至是文件读取操作和对字符串的操作我都不太熟悉，然后我又花了一天的时间来熟悉这些操作。然后我就开始了长达一周没日没夜的代码生活。

经过这一周，我不仅收获了大大的黑眼圈，也收获了宝贵的对程序设计的理解和编程技术的提升。现在，我已经不是一周前的我了，不再畏惧没有学过的知识，现在拿到一道程序设计题我都会先想用什么数据结构，有什么算法能够使用，什么算法的时间复杂度更优，然后才开始写代码，然后我觉得对编程语法的基础的掌握也是非常重要的，这也可以说是我总结出来的赤裸裸的教训，比如没有很好的掌握C语言指针的相关知识，那么更不用谈什么数据结构了。

总之，这一周的数据结构课程设计我受益匪浅，这为我将来的编程之路奠定了基础！最后感谢孙涵老师以及林昱涵学长一个学期以来的耐心指导！我今后一定继续努力，保证不会懈怠！