**要求代码和实验报告规范，在算法思想中：对实验涉及的数据结构进行有效设计和分析；对算法进行分析并给出时间、空间复杂度的结论；清晰表达实验思路、出现的问题及解决方法。**

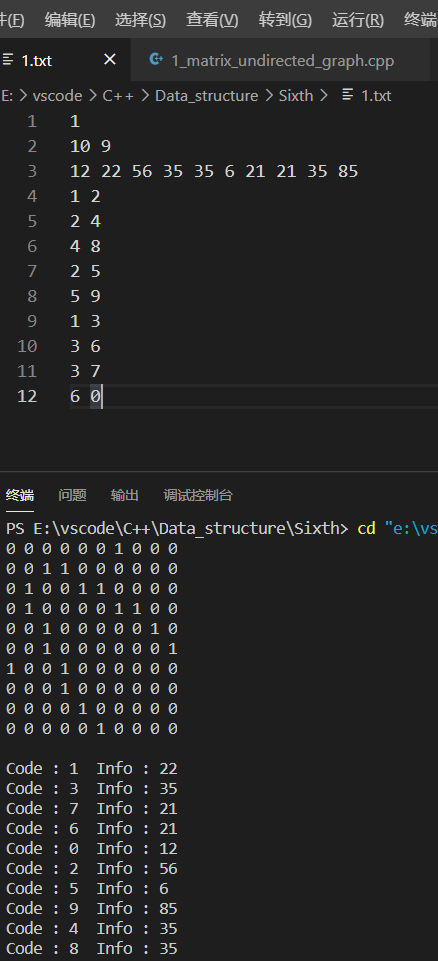
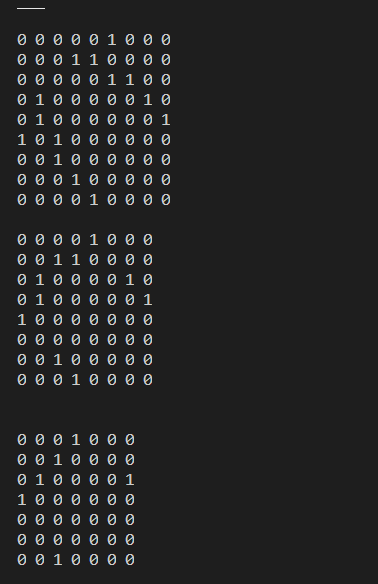
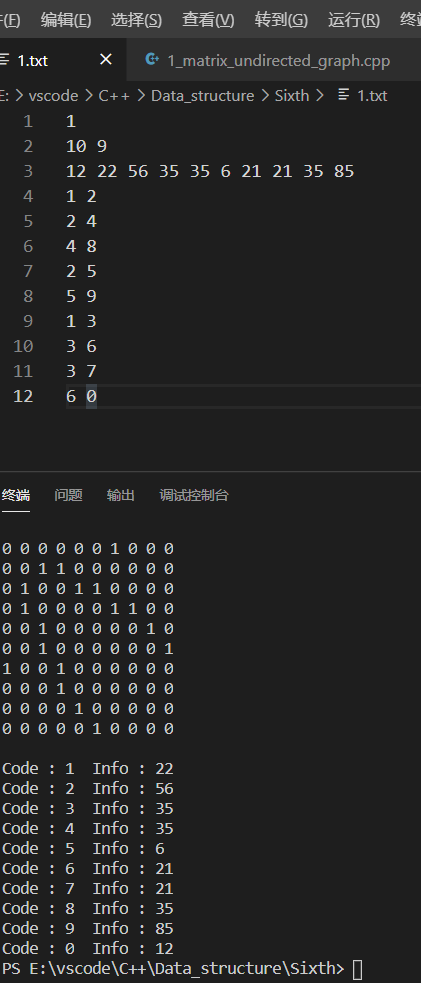
一、调试成功程序及说明

1、

题目：利用邻接矩阵储存方式实现图的基本操作

算法思想：作出基本操作

运行结果：

结果分析：运行结果均正确

附源程序。

#include <iostream>

#include <string>

#include <fstream>

#include <stack>

#include <queue>

using namespace std;

#define ElemType int

#define INFINITY INT\_MAX //定义正无穷

#define MAX\_V 20 //最大定点数目

#define GraphKind int //声明图的标志 1无向图 2有向图 3有向网 4无向网

typedef struct

{

int code; //结点编号0~G.vernum

ElemType info; //结点其他信息，这里是点权

}VertexType;

typedef struct

{

int arcs[MAX\_V][MAX\_V]; //邻接矩阵

int vexnum, arcnum; //顶点个数和边的个数

VertexType vexs[MAX\_V]; //存放定点信息

GraphKind type;

}MGraph; //图的邻接矩阵类型

bool CreateGraph(MGraph &G, string FileName)

{

fstream file(FileName.c\_str(),ios::in);

if(!file)

{

cout << "Floating TIP File " << FileName << " can't be opened !"<<endl;

return false;

}

file >> G.type; //输入图的类型

file >> G.vexnum >> G.arcnum ;

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

{

G.vexs[i].code = i;

file >> G.vexs[i].info;

}

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

{

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

{

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

}

}

int x = 0, y = 0;

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

{

file >> x >> y;

G.arcs[x][y] = 1;

G.arcs[y][x] = 1;

}

return true;

}

//建立无向图

void DestroyGraph(MGraph &G)

{

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

{

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

{

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

}

}

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

{

G.vexs[i].code = 0;

G.vexs[i].info = 0;

}

G.arcnum = 0;

G.vexnum = 0;

G.type = -1;

return;

}

//销毁无向图

VertexType GetVex(MGraph G, int v) //v是编号，返回元素值

{

VertexType e;

if(v < 0 || v >= G.vexnum)

{

e.code = -1;

e.info = -1;

return e;

}

else

{

e = G.vexs[v];

return e;

}

}

//返回下标为v的元素值

int FirstAdjVex(MGraph G, int v)

{

if(v < 0 || v >= G.vexnum)

{

return -1;

}

bool flag = false;

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

{

if(G.arcs[v][j] == 1)

{

flag = true;

return j;

}

}

if(!flag)

{

return -1; //找不到邻接点

}

}

int NextAdjVex(MGraph G, int v, int w)

{

if(v < 0 || v >= G.vexnum)

{

return -1;

}

bool flag = false;

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

{

if(G.arcs[v][j] == 1)

{

flag = true;

return j;

}

}

if(!flag)

{

return -1;

}

}

void VisitVertex(VertexType Ver)

{

cout << "Code : "<<Ver.code << " Info : " << Ver.info << " "<<endl;

return;

}

//访问的函数

void DFSTraverse(MGraph G, int v) //v记录当前遍历到的结点

{

bool flag[G.vexnum];

stack <int> S;

S.push(v); //从v开始进行遍历

int now;

while(!S.empty())

{

now = S.top();

VisitVertex(G.vexs[now]);

flag[now] = true;

S.pop();

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

{

if(G.arcs[now][j] == 1 && !flag[j]) //找到一个连接的点，并且没有访问过

{

S.push(j);

}

}

}

return ;

}

//进行深度优先遍历

void BFSTraverse(MGraph G, int v)

{

bool flag[G.vexnum];

queue <int> Q;

int now;

VisitVertex(G.vexs[v]);

flag[v] = true;

Q.push(v);

while(!Q.empty())

{

now = Q.front();

Q.pop();

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

{

if(G.arcs[now][j] == 1 && !flag[j])

{

VisitVertex(G.vexs[j]);

flag[j] = true;

Q.push(j);

}

}

}

return;

}

//广度优先遍历

void InsertVex(MGraph &G, VertexType ver)

{

if(G.vexnum == MAX\_V)

{

cout << "Memory is overflowed, vertex can't insert !"<<endl;

}

ver.code = G.vexnum;

G.vexs[G.vexnum] = ver; //储存顶点信息

G.vexnum++;

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

{

G.arcs[ver.code][i] = 0;

G.arcs[i][ver.code] = 0;

}

return;

}

//插入一个顶点

void InsertArc(MGraph &G, int v, int w)

{

if(v < 0 || v >= G.vexnum || w < 0 || w >= G.vexnum)

{

cout << "can't add edge"<<endl;

return ;

}

G.arcs[v][w] = 1;

G.arcs[w][v] = 1;

G.arcnum ++;

return;

}

//插入一条边

void DeleteVex(MGraph &G, int v)

{

if(v < 0 || v >= G.vexnum)

{

cout << v <<" vertex is not exist !"<<endl;

return ;

}

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

{

if(G.arcs[i][v] == 1 || G.arcs[v][i] == 1)

{

G.arcnum--;

}

G.arcs[v][i] = 0;

G.arcs[i][v] = 0;

}

//直接全扫一遍

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

{

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

{

G.arcs[i][j] = G.arcs[i+1][j];

}

}

for(int i = 0; i < G.vexnum - 1; i++)

{

for(int j = v; j < G.vexnum - 1; j++)

{

G.arcs[i][j] = G.arcs[i][j+1];

}

}

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

{

G.vexs[i] = G.vexs[i+1];

G.vexs[i].code--;

}

//对点进行移位

G.vexnum--;

return;

}

void DeleteArc(MGraph &G, int v, int w)

{

if(v < 0 || v >= G.vexnum || w < 0 || w >= G.vexnum)

{

cout << "Can't delete edge"<<endl;

return ;

}

if(G.arcs[v][w] == 1 || G.arcs[w][v] == 1)

{

G.arcs[v][w] = 0;

G.arcs[w][v] = 0;

cout << "Floating TIP Delete the arc betewwn " << v << " and "<< w << " successfully ! "<<endl;

}

else

{

cout << "Floating TIP The arc between "<< v << " and "<< w<< " is now exist."<<endl;

}

return;

}

void DisplayMGraph(MGraph G)

{

if(G.vexnum == 0 && G.vexnum ==0)

{

cout << "NULL Graph"<<endl;

}

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

{

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

{

cout << G.arcs[i][j]<< " ";

}

cout << endl;

}

}

int main()

{

string f = "1.txt";

MGraph G;

VertexType V;

V.code =100;

V.info = 99;

CreateGraph(G,f);

DisplayMGraph(G);

cout << endl;

DFSTraverse(G,1);

cout << endl;

DisplayMGraph(G);

cout << endl;

BFSTraverse(G,1);

//DestroyGraph(G);

//VisitVertex(V);

//V = GetVex(G,1);

//VisitVertex(V);

//DisplayMGraph(G);

// DisplayMGraph(G);

// cout << FirstAdjVex(G,2)<<endl;;

// cout << NextAdjVex(G,2,1);

DeleteVex(G,1);

DisplayMGraph(G);

cout <<endl;

DeleteVex(G,2);

DisplayMGraph(G);

cout <<endl;

cout <<endl;

DeleteVex(G,2);

DisplayMGraph(G);

cout <<endl;

return 0;

}

2、

题目：实现霍夫曼树

算法思想：按照权重进行树的合并

运行结果：

结果分析：

附源程序。/\*4、Huffman编码与解码 (必做)（Huffman编码、二叉树）

[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。

\*/

/\*实现对ASCII码 32~126 这95个字符的编码，他们的权重由在源文件中出现的频率决定，首先读取文件，计算每个ASCII码对应的权重，\*/

#include <iostream>

#include <fstream>

#include <malloc.h>

#include <string>

using namespace std;

#define MAX\_VALUE 2222222

#define MaxBit 10

typedef struct HTNode

{

int weight;

int parent, lchild, rchild;

bool flag; //对树合并时进行标记

}HTNode,\*HufTree;

typedef struct Code //存放哈夫曼编码的数据元素结构

{

char data; //存入字符数据

int bit[MaxBit];//数组

int start;//编码的起始下标

int weight;//字符的权值

}HufCode;

void CaculateWeightAndSaveFile(int \*weight, string inName, string outName)

{

//对ASCII码在32~126的可显示码进行统计

fstream inFile(inName.c\_str(), ios::in);

fstream outFile(outName.c\_str(), ios::out);

if(!inFile)

{

throw "Floating TIP In the function void CaculateWeightAndSaveFile() File " + inName + " can't be opened , please check it. ";

}

for(int i = 0; i < 95; i++)

{

weight[i] = 0;

}

char ch = 'A';

while (!inFile.eof())

{

ch = inFile.get();

if(ch >= 32 && ch <= 126)

{

weight[ch - 32]++;

}

}

for(int i = 0; i < 95; i++)

{

outFile << (char)(i + 32) << " " << weight[i]<<endl;

//cout << (char)i << " " << weight[i]<<endl;

}

inFile.close();

outFile.close();

return;

}

void readFrequency(int &n, HufCode code[], string frequencyFileName) //个数 编码数组 文件名

{

fstream file(frequencyFileName.c\_str(),ios::in);

if(!file)

{

throw "Floating TIP In the function void readFrequency() File " + frequencyFileName + " can't be opened , please check it. ";

}

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

{

file >> code[i].data >> code[i].weight;

}

file.close();

return;

}

void initHufTree(HufTree &HT,int n,int \*w)

{

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

{

if(i <= n)

{

HT[i].weight = w[i];

}

else

{

HT[i].weight = 0;

}

HT[i].parent = 0;

HT[i].flag = false;

HT[i].lchild = -1;

HT[i].rchild = -1;

}

}

void CreateHufTree(HufTree &HT, int n, int \*w)

{

if(n <= 0)

{

return ;

}

HT = (HufTree)malloc(sizeof(HTNode)\*2\*n + 10);

int x1,x2;

int w1,w2;

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

{

x1 = 1;

x2 = 1;

w1 = MAX\_VALUE;

w2 = MAX\_VALUE;

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

{

if(HT[j].weight< w1 && !HT[j].flag)

{

w2 = w1;

x2 = x1;

x1 = j;

w1 = HT[j].weight;

}

else if(HT[j].weight < w2 && !HT[j].flag)

{

x2 = j;

w2 = HT[j].weight;

}

}

HT[n + i].lchild = x1;

HT[n + i].rchild = x2;

HT[n + i].weight = HT[x1].weight + HT[x2].weight;

HT[x1].parent = n + i;

HT[x2].parent = n + i;

HT[x1].flag = true;

HT[x2].flag = true;

}

return;

}

//建立霍夫曼树

void CaculateHufCoding(HufTree HT, HufCode HufTable[], int n) //Tree ,霍夫曼编码表 n个节点

{

HufCode \*cd = (HufCode\*)malloc(sizeof(HufCode));

int child, parent; //临时变量

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

{

cd->start = 0;

cd->weight = HT[i].weight;

child = i;

parent = HT[child].parent;

while(parent != 0) //向上寻找,直到找到根节点

{

if(HT[parent].lchild == child)

{

cd->bit[cd->start] = 0; //左边是0

}

if(HT[parent].rchild == child)

{

cd->bit[cd->start] = 1; //右边是1

}

cd->start ++;

child = parent;

parent = HT[parent].parent;

}

}

}

void ShowHufTree(HufTree &BT)

{

}

int main()

{

int N;

int weight[256];

string inFileName = "E://vscode//C++//Data\_structure//Cours\_Design//Sourse//source.txt";

string outFileName = "E://vscode//C++//Data\_structure//Course\_Design//Sourse//Huffman.txt";

try

{

CaculateWeightAndSaveFile(weight, inFileName, outFileName);

}catch (string ss)

{

cout << ss <<endl;

}

//HufTree HT;

//read(N,weight);

//CreateHufTree(HT, N ,weight);

return 0;

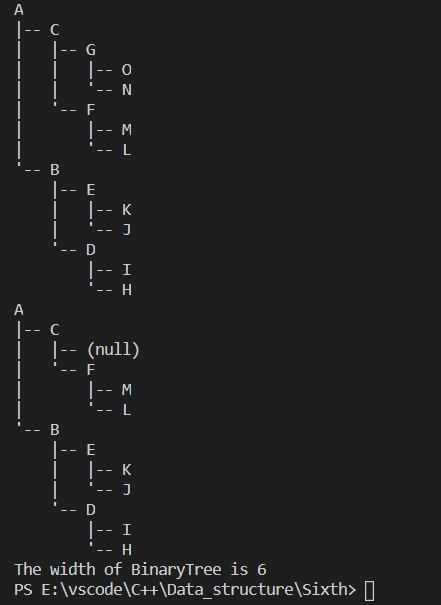
}

3、

题目：编写函数，对二叉链表结构的二叉树，求宽度。（书P94 4）

算法思想：方法类似于层序遍历，记录每一层的宽度即可。

运行结果：



结果分析：运行结果正确，可以正确显示宽度。

附源程序（为了方便运行检查，这里附上二叉树的程序，运行test函数即可）。

{

BiTNode \*p = BT;

queue <BiTNode\*> Q;

Q.push(BT);

Q.push(NULL);

int max\_cnt = -1;

int cnt = 1;

bool flag = false;

while(!Q.empty())

{

p = Q.front();

Q.pop();

if(p == NULL)

{

if(flag)

{

break;

}

max\_cnt = max(max\_cnt,cnt);

cnt = 0;

Q.push(NULL);

flag = true;

continue;

}

flag = false;

if(p->lchild != NULL)

{

cnt ++;

Q.push(p->lchild);

}

if(p->rchild != NULL)

{

cnt ++;

Q.push(p->rchild);

}

}

return max\_cnt;

}

#define TElemtype char

#include <iostream>

#include <fstream>

#include <malloc.h>

#include <cmath>

#include <queue>

#include <stack>

using namespace std;

typedef struct BiTNode

{

TElemtype data;

struct BiTNode\* lchild, \* rchild;

}BiTNode, \* BiTree;

void print(TElemtype e)

{

cout << e;

}

bool InitBitree(BiTree& T)

{

T = NULL;

return true;

}

//构造一颗空的二叉树，空即没有结点

void DestroyBiTree(BiTree& T)

{

if (T == NULL)

{

return ;

}

if (T->lchild != NULL)

{

DestroyBiTree(T->lchild);

}

if (T->rchild != NULL)

{

DestroyBiTree(T->rchild);

}

free(T);

T = NULL;

return;

}

//销毁一颗树

BiTree CreateBiTree(BiTree T, fstream &file)

{

if (file.eof())

{

return NULL;

}

TElemtype data; //试验阶段数据类型为char型

file >> data;

if (data == '#')

{

return NULL;

}

else

{

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

T->data = data;

T->lchild = CreateBiTree(T->lchild, file);

//cout << "LeftChild over i is " << i <<" data is "<< data <<endl;

T->rchild = CreateBiTree(T->rchild, file);

//cout << "RightChild over i is " << i << " data is " << data << endl;

}

return T;

}

//建树,利用递归的方法，先建立左子树，再建立又子树

bool BiTreeEmpty(BiTree T)

{

if (T == NULL)

{

return true;

}

else

{

return false;

}

}

//判断树是否为空树

int BiTreeDepth(BiTree T)

{

if (T == NULL)

{

return 0;

}

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

{

return 1;

}

return max(BiTreeDepth(T->lchild), BiTreeDepth(T->rchild))+1;

}

//返回树的深度

bool Root(BiTree T, TElemtype& e)

{

if (T == NULL)

{

return false;

}

else

{

e = T->data;

return true;

}

}

//二叉树存在，返回二叉树跟的元素

bool Value(BiTree T, BiTNode\* cur\_p, TElemtype& e)

{

if (T == NULL)

{

return false;

}

else

{

e = cur\_p->data;

return true;

}

}

// 返回cur\_p指向的结点的值e

void FindFather(BiTree T, BiTNode\* cur\_p, BiTree& parent) //parent为指针的引用

{

if (T == NULL)

{

return;

}

if (T->lchild == cur\_p)

{

parent = T;

return;

}

if (T->rchild == cur\_p)

{

parent = T;

return;

}

FindFather(T->lchild, cur\_p, parent);

FindFather(T->rchild, cur\_p, parent);

return;

}

//递归寻找双亲的辅助函数

bool Parent(BiTree T, BiTNode\* cur\_p, BiTree& parent)

{

if (T == NULL)

{

parent = NULL;

return false;

}

if (T == cur\_p)

{

//当传入指针为根节点时候，他没有双亲

parent = NULL;

return false;

}

parent = NULL;

//上边情况都不是的话，那么递归查找他的双亲

FindFather(T, cur\_p, parent);

if (parent == NULL)

{

//没有找到双亲的情况

return false;

}

else

{

return true;

}

}

//通过引用返回双亲指针

bool LeftChild(BiTree T, BiTNode\* cur\_p, BiTree &leftChild)

{

if (T == NULL)

{

return false;

}

if (cur\_p->lchild != NULL)

{

leftChild = cur\_p->lchild;

return true;

}

else

{

leftChild = NULL;

return false;

}

}

//返回左儿子的指针

bool RightChild(BiTree T, BiTNode\* cur\_p, BiTree &rightChild)

{

if (T == NULL)

{

rightChild = NULL;

return false;

}

if (cur\_p->rchild != NULL)

{

rightChild = cur\_p->rchild;

return true;

}

else

{

rightChild = NULL;

return false;

}

}

//返回右儿子的指针

bool LeftBrother(BiTree T, BiTNode\* cur\_p, BiTree& leftBrother)

{

BiTNode \*parent = NULL;

Parent(T, cur\_p, parent);

cout << "yes" << endl;

cout << "parent is " << parent << endl;

if (parent == NULL)

{

leftBrother = NULL;

return false;

}

if (parent->lchild != NULL && parent->lchild != cur\_p)

{

leftBrother = parent->lchild;

return true;

}

else

{

leftBrother = NULL;

return false;

}

}

//返回左兄弟的指针

bool RightBrother(BiTree &T, BiTNode\* cur\_p, BiTree& rightBrother)

{

BiTNode\* parent = NULL;

Parent(T, cur\_p, parent);

if (parent == NULL)

{

rightBrother = NULL;

return false;

}

if (parent->rchild != NULL && parent->rchild != cur\_p)

{

rightBrother = parent->rchild;

return true;

}

else

{

rightBrother = NULL;

return false;

}

}

//返回右兄弟的指针

void visit(BiTNode\* cur\_p)

{

if (cur\_p == NULL)

{

return;

}

else

{

print(cur\_p->data);

return;

}

}

//visit 访问结点的函数

void PreOrderTraverse(BiTree T)

{

visit(T);

if (T == NULL)

{

return;

}

PreOrderTraverse(T->lchild);

PreOrderTraverse(T->rchild);

return;

}

//进行先序遍历

void InOrderTraverse(BiTree T)

{

if (T == NULL)

{

visit(T);

return;

}

InOrderTraverse(T->lchild);

visit(T);

InOrderTraverse(T->rchild);

return;

}

//中序遍历

void PostOrderTraverse(BiTree T)

{

if (T == NULL)

{

visit(T);

return;

}

PostOrderTraverse(T->lchild);

PostOrderTraverse(T->rchild);

visit(T);

return;

}

//实现后序遍历

void LevelOrderTraverse(BiTree T)

{

if (T == NULL)

{

cout << "At least you still have an empty set " << endl;

return;

}

queue <BiTNode\*> Q;

BiTNode\* p, \* q;

p = T;

Q.push(p);

while (!Q.empty())

{

q = Q.front();

visit(q);

Q.pop();

if (q->lchild != NULL)

{

Q.push(q->lchild);

}

if (q->rchild != NULL)

{

Q.push(q->rchild);

}

}

return;

}

//实现层次遍历

bool Assign(BiTree T, BiTree& cur\_p, TElemtype value)

{

if (T == NULL)

{

return false;

}

cur\_p->data = value;

return true;

}

//将cur\_p指向结点的data值赋值为value

bool InsertChild(BiTree T, BiTNode\* cur\_p, bool LR, BiTree c)

{

if (T == NULL)

{

return false;

}

BiTNode\* temp = NULL;

if (!LR)

{

temp = cur\_p->lchild;

cur\_p->lchild = c;

c->rchild = temp;

}

else

{

temp = cur\_p->rchild;

cur\_p->rchild = c;

c->rchild = temp;

}

return true;

}

//插入c为p的左或者右子树，c的右子树为p原来被替换的子树

bool DeleteChild(BiTree T, BiTNode\* cur\_p, bool LR)

{

if (T == NULL)

{

return false;

}

if (!LR)

{

DestroyBiTree(cur\_p->lchild);

}

else

{

DestroyBiTree(cur\_p->rchild);

}

return true;

}

void ShowBiTree(BiTree T)

{

if (T == NULL)

{

return;

}

print(T->data);

if (T->lchild != NULL || T->rchild != NULL)

{

cout << "( ";

ShowBiTree(T->lchild);

cout << ", ";

ShowBiTree(T->rchild);

cout << " )";

}

return;

}

void Pruning(BiTree& T, TElemtype x,BiTree oldTree)

{

if (T == NULL)

{

return;

}

if (T->data == x)

{

if(T == oldTree)

{

DestroyBiTree(T);

return;

}

BiTNode\* parent;

Parent(oldTree, T, parent);

if (parent->lchild == T)

{

parent->lchild = NULL;

}

if (parent->rchild == T)

{

parent->rchild = NULL;

}

DestroyBiTree(T);

return;

}

Pruning(T->lchild, x, oldTree);

Pruning(T->rchild, x, oldTree);

return;

}

bool IsCompleteBiTree(BiTree T)

{

BiTNode\* p = NULL;

queue<BiTNode\*> Q;

int flag = 0;

Q.push(T);

while (!Q.empty())

{

p = Q.front();

Q.pop();

if (p->lchild == NULL && p->rchild == NULL) //没儿子

{

flag = 1;

}

else if (p->lchild == NULL && p->rchild != NULL) //只有右儿子

{

return false;

}

else if (p->lchild != NULL && p->rchild == NULL) //只有左儿子

{

if (flag != 0)

{

return false; //前边已经有度为1，或者叶子结点了，你却有了一个儿子

}

else

{

flag = 1;

}

}

else

{

if (flag > 0)

{

return false; //此前已经有度不大于1的结点被访问过

}

Q.push(p->lchild);

Q.push(p->rchild);

}

}

return true;

}

void PreOrderTraverseWithoutRecursion(BiTree T)

{

if (T == NULL)

{

return;

}

stack <BiTNode\*> S;

BiTNode\* p, \* q;

p = T;

S.push(p);

while (!S.empty())

{

q = S.top();

S.pop();

visit(q);

if (q->rchild != NULL)

{

S.push(q->rchild);

}

if (q->lchild != NULL)

{

S.push(q->lchild);

}

}

}

//非递归方法实现先序遍历

void InOrderTraverseWithoutRecursion(BiTree T)

{

if (T == NULL)

{

return;

}

stack <BiTNode\*> S;

BiTNode \*p;

p = T;

while (!S.empty() || p != NULL)

{

while (p != NULL)

{

S.push(p);

p = p->lchild;

}

if (!S.empty()) //栈顶元素没有左儿子

{

p = S.top();

visit(p);

S.pop();

p = p->rchild;

}

}

return;

}

//非递归方法实现中序遍历

void PostOrderTraverseWithoutRecursion(BiTree T)

{

if (T == NULL)

{

return;

}

stack <BiTNode\*> S;

BiTNode\* cur = NULL, \*pre = NULL;

cur = T;

S.push(cur);

while (!S.empty())

{

cur = S.top();

if ((cur->lchild == NULL && cur->rchild == NULL) || (pre != NULL && (pre == cur->lchild || pre == cur->rchild)))

{

visit(cur);

S.pop();

pre = cur;

}

else

{

if (cur->rchild != NULL)

{

S.push(cur->rchild);

}

if (cur->lchild != NULL)

{

S.push(cur->lchild);

}

}

}

return;

}

//非递归方法实现后序遍历

// 一个数组，数组长度不低于二叉树的高度，为了简便起见，这里假设是一百,

// 自己使用得时候可以改成动态数组(或链表)

int vec\_left[100] = {0};

// 显示二叉树的函数，只要调用Display(root, 0)即可

void DisplayBinaryTree(BiTree &root, int ident)

{

if(ident > 0)

{

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

{

printf(vec\_left[i] ? "| " : " ");

}

printf(vec\_left[ident-1] ? "|-- " : "'-- ");

}

if(! root)

{

cout << "(null)" <<endl;

return;

}

cout << root->data << endl;

if(!root->lchild && !root->rchild)

{

return;

}

vec\_left[ident] = 1;

DisplayBinaryTree(root->rchild, ident + 1);

vec\_left[ident] = 0;

DisplayBinaryTree(root->lchild, ident + 1);

}

int CaculateWidthOfBinaryTree(BiTree &BT)

{

BiTNode \*p = BT;

queue <BiTNode\*> Q;

Q.push(BT);

Q.push(NULL);

int max\_cnt = -1;

int cnt = 1;

bool flag = false;

while(!Q.empty())

{

p = Q.front();

Q.pop();

if(p == NULL)

{

if(flag)

{

break;

}

max\_cnt = max(max\_cnt,cnt);

cnt = 0;

Q.push(NULL);

flag = true;

continue;

}

flag = false;

if(p->lchild != NULL)

{

cnt ++;

Q.push(p->lchild);

}

if(p->rchild != NULL)

{

cnt ++;

Q.push(p->rchild);

}

}

return max\_cnt;

}

void test()

{

fstream file("E://vscode//C++//Data\_structure//Fifth//CreateBiTree.txt", ios::in);

if (!file)

{

cout << "file can't be opened ！" << endl;

exit(0);

}

BiTree BT;

InitBitree(BT);

BT = CreateBiTree(BT, file);

//ShowBiTree(BT);

cout << endl;

DisplayBinaryTree(BT,0);

//Pruning(BT,'A',BT);

// LevelOrderTraverse(BT);

Pruning(BT, 'G', BT);

// cout << endl;

// cout << IsCompleteBiTree(BT) << endl;

int width = 0;

width = CaculateWidthOfBinaryTree(BT);

DisplayBinaryTree(BT,0);

cout << "The width of BinaryTree is "<< width << endl;

return ;

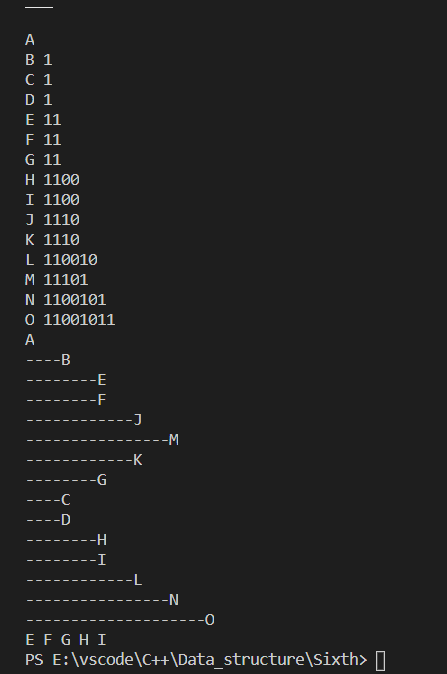
}

4、

题目：编写函数，对一棵以孩子-兄弟链表表示的树，输出第i层的所有元素。

算法思想：这个函数实现起来思想比较简单，类似于层序遍历的思想，遍历这一层的元素，将这个元素的所有孩子存入队列中，直到要找的那一层。由于使用的是孩子兄弟表示法，所以遍历到每个元素时，要先找到第一个孩子，然后根据第一个孩子进行兄弟的寻找。当找到最终层数时，输出最终结果即可。

运行结果：



结果分析：结果运行正确，实现了建树，对树进行显示，输出某一层的所有元素。

附源程序。

/\*4. 编写函数，对一棵以孩子-兄弟链表表示的树，输出第i层的所有元素。\*/

#include <iostream>

#include <string>

#include <queue>

#include <malloc.h>

#include <fstream>

using namespace std;

#define TElemType char

typedef struct TreeNode

{

TElemType data;

struct TreeNode \*firstchild, \*nextBrother;

}CBTreeNode, \*CBTree;

void CreateCBTree(CBTree &T)

{

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

T->data = 0;

T->firstchild = NULL;

T->nextBrother = NULL;

cout << "YES"<<endl;

return;

}

//创建一课孩子兄弟表示法表示的树

//pos 为一个01串，可以表示寻找结点的过程，0表示向右，1表示向下

bool InsertNode(CBTree &T, char data, string pos) //pos is a series of 01

{

if(T == NULL)

{

cout << "You can't insert anything to NULL Tree"<<endl;

return false;

}

int count = pos.length();

TreeNode \*node = (TreeNode\*)malloc(sizeof(TreeNode));

node->data = data;

node->firstchild = NULL;

node->nextBrother = NULL;

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

p->firstchild = T;

int i=0;

char temp;

while(count > 0 && p != NULL)

{

temp = pos[i];

if(temp == '0')

{

p = p->nextBrother;

}

else if(temp == '1')

{

p = p->firstchild;

}

else

{

cout << "Your pos is invalid, the string pos is "<< pos<<endl;

return false;

}

count --;

i++;

}

if(p == NULL)

{

cout << "Insert position is invalid, the string pos is "<< pos <<" and the count is "<< count <<endl;

return false;

}

CBTreeNode \*q = p->firstchild;

if(q == NULL)

{

p->firstchild = node;

}

else

{

while(q->nextBrother != NULL)

{

q = q->nextBrother ;

}

q->nextBrother = node;

}

return true;

}

//对树进行结点的插入

void readTree(CBTree &T, string FileName)

{

int N=0;

char data;

string pos;

fstream file(FileName.c\_str(), ios::in);

if(!file)

{

cout << "file can't be opened !"<<endl;

exit(0);

}

file >> N; //多少个结点，包括根结点

file.ignore();

file >> T->data;

cout << T->data<<endl;

char a[2];

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

{

file.ignore();

file.read((char\*)&a,1);

data = a[0];

file >> pos;

InsertNode(T, data, pos);

cout << data << " " << pos<<endl;

}

return;

}

//从文件中读入树的数据

void r\_display(CBTreeNode \*node, int gap)

{

if(node == NULL)

{

return;

}

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

{

cout << "-";

}

cout << node->data << endl;

CBTreeNode \*child = node->firstchild;

r\_display(child, gap+4);

CBTreeNode \*brother = node->nextBrother;

r\_display(brother, gap);

}

//递归打印树

void Display(CBTree &T)

{

if(T == NULL)

{

cout << "null" <<endl;

}

else

{

r\_display(T,0);

}

}

void printNode(CBTreeNode \*node)

{

cout << node->data << " ";

}

void PrintTH(CBTree &T,int i)

{

CBTreeNode \*p = T, \*q;

queue <CBTreeNode\*> Q; //搞一个队列

bool flag = false;

int cnt = 0;

i --;

Q.push(p);

Q.push(NULL);

while(!Q.empty())

{

if(cnt == i)

{

while(!Q.empty())

{

q = Q.front();

Q.pop();

if(q != NULL)

{

printNode(q);

}

}

break;

}

p = Q.front();

Q.pop();

if(p == NULL)

{

if(flag)

{

cout << "Floating TIP WARNING OVERFLOW" <<endl;

break;

}

cnt ++;

Q.push(NULL);

flag = true;

continue;

}

flag = false;

if(p->firstchild != NULL)

{

q = p->firstchild;

do

{

Q.push(q);

q = q->nextBrother;

}while(q!=NULL);

}

}

}

//打印第i层

int main()

{

CBTree T;

CreateCBTree(T);

readTree(T,"4.txt");

Display(T);

PrintTH(T,3);

// char p;

// fstream file("4.txt",ios::in);

// while (!file.eof())

// {

// file >> p;

// cout << p <<endl;

// }

return 0;

}

5、

题目：

CSP题目

问题描述：俄罗斯方块是俄罗斯人阿列克谢·帕基特诺夫发明的一款休闲游戏。

　　游戏在一个15行10列的方格图上进行，方格图上的每一个格子可能已经放置了方块，或者没有放置方块。每一轮，都会有一个新的由4个小方块组成的板块从方格图的上方落下，玩家可以操作板块左右移动放到合适的位置，当板块中某一个方块的下边缘与方格图上的方块上边缘重合或者达到下边界时，板块不再移动，如果此时方格图的某一行全放满了方块，则该行被消除并得分。

　　在这个问题中，你需要写一个程序来模拟板块下落，你不需要处理玩家的操作，也不需要处理消行和得分。

　　具体的，给定一个初始的方格图，以及一个板块的形状和它下落的初始位置，你要给出最终的方格图。

输入格式：

　　输入的前15行包含初始的方格图，每行包含10个数字，相邻的数字用空格分隔。如果一个数字是0，表示对应的方格中没有方块，如果数字是1，则表示初始的时候有方块。输入保证前4行中的数字都是0。

　　输入的第16至第19行包含新加入的板块的形状，每行包含4个数字，组成了板块图案，同样0表示没方块，1表示有方块。输入保证板块的图案中正好包含4个方块，且4个方块是连在一起的（准确的说，4个方块是四连通的，即给定的板块是俄罗斯方块的标准板块）。

　　第20行包含一个1到7之间的整数，表示板块图案最左边开始的时候是在方格图的哪一列中。注意，这里的板块图案指的是16至19行所输入的板块图案，如果板块图案的最左边一列全是0，则它的左边和实际所表示的板块的左边是不一致的（见样例）

输出格式：

输出15行，每行10个数字，相邻的数字之间用一个空格分隔，表示板块下落后的方格图。注意，你不需要处理最终的消行。

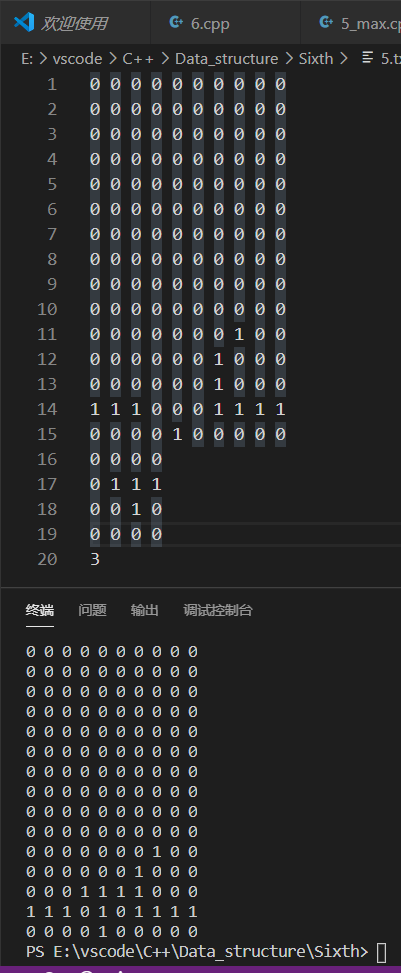
算法思想：

这道题目我尝试了三种方法。第一种是首先计算碰撞点的坐标，然后把图案复制过去，第二种是进行连通性判断，利用深度优先搜索，找到连通部分的数值，如果是4的话，那么物块下落。第三种是最简单有效的方法，判断每一列物块下边是不是为空，是的话就将物块下落。为了达到上述目的，我们为了方便搜索，将下落物块进行染色处理，其标记为2，原有物块标记为1，空的部分标记为0。根据题意，前四行数据全为零，即可以将4\*4的物块复制到原图形上来，顺便将其染色。

首先我们根据需求，实现物块下落的功能，对于每一列，按照行数，从最后一行向上搜索，搜索到标记为2的块，就将其下边的一个变为2，当前格子变为其上方的数据。

实现判断物块是否可以继续下落的函数，如果所有列的下方都为零，那么物块可以下落。（这里我进行计数，如果物体占有的列数，等于下方为零的数目，那么认为物块可以继续下落）

运行结果：



结果分析：运行结果正确，通过CSP测试。



附源程序。

#include <iostream>

#include <fstream>

#include <string.h>

using namespace std;

int map[20][20];

int object[10][10];

int position = 0;

int max\_col[20];

int min\_col[10];

void print()

{

for (int i = 1; i <= 15; i++)

{

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

{

if(map[i][j] != 0)

{

cout << "1";

}

else

{

cout << "0";

}

cout <<" ";

}

cout << endl;

}

return ;

}

void test\_print()

{

for (int i = 1; i <= 15; i++)

{

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

{

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

}

cout << endl;

}

return ;

}

int read()

{

fstream file("5.txt",ios::in);

if(!file)

{

cout << "file can't be opened !"<<endl;

exit(0);

}

for (int i = 1; i <= 15; i++)

{

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

{

file >> map[i][j];

}

}

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

{

map[16][j] = 1;

}

for (int i = 1; i <= 4; i++)

{

for(int j = 1; j <= 4; j ++)

{

file >> object[i][j];

}

}

file >> position ;

}

void start()

{

for(int i = 1; i <= 4; i++)

{

for(int j = 1; j <= 4; j++)

{

if(object[i][j] != 0)

{

map[i][position+j-1] = 2;

}

}

}

return ;

}

void drop( )

{

for (int j = 0; j < 4; j++)

{

for(int i = 15; i >= 1; i--)

{

if(map[i][j+position] == 2)

{

map[i+1][j+position] = map[i][j+position];

map[i][j+position] = map[i-1][j+position];

}

}

}

return;

}

int cnt = 0;

void dfs(int x, int y)

{

if(map[x][y] == 0)

{

cout<<"yes"<<endl;

return;

}

else

{

cnt++;

map[x][y] = 0;

if(x + 1 < 16 && map[x+1][y] != 0)

{

dfs(x+1,y);

}

if(x - 1 > 0 && map[x-1][y] != 0)

{

dfs(x-1,y);

}

if(y + 1 < 11 && map[x][y+1] != 0)

{

dfs(x,y+1);

}

if(y - 1 > 0 && map[x][y-1] != 0)

{

dfs(x,y-1);

}

map[x][y] = 2;

}

}

void FindFirst(int &x, int &y)

{

for(int i = 1; i <= 15; i++)

{

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

{

if(map[i][j] == 2)

{

x = i;

y = j;

return ;

}

}

}

return;

}

bool judge()

{

int sum = 0;

int cnt\_1 = 0;

bool flag = false;

for (int j = 0; j < 4; j++)

{

flag = false;

for(int i = 15; i >= 1; i--)

{

if(map[i][j+position] == 2)

{

flag = true;

if(map[i+1][j+position] == 0)

{

cnt\_1++;

}

}

}

if(flag)

{

sum ++;

}

}

if(sum == cnt\_1)

{

return true;

}

else

{

return false;

}

}

int main()

{

int x = 1, y =1;

read();

start();

do

{

if(cnt == 4)

{

drop();

}

cnt = 0;

FindFirst(x,y);

cout << "x = "<<x<<" y = "<<y << endl;

dfs(x,y);

}while(cnt == 4);

while(judge())

{

drop();

}

cout << endl;

print();

return 0;

}

6、

题目：

算法思想：

运行结果：

结果分析：

附源程序。

......

二、未调试成功程序及说明

1、

题目：

算法思想：

错误原因：

附源程序。

2、

题目：

算法思想：

错误原因：

附源程序。

......

三、代码行数及小结

