**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。**

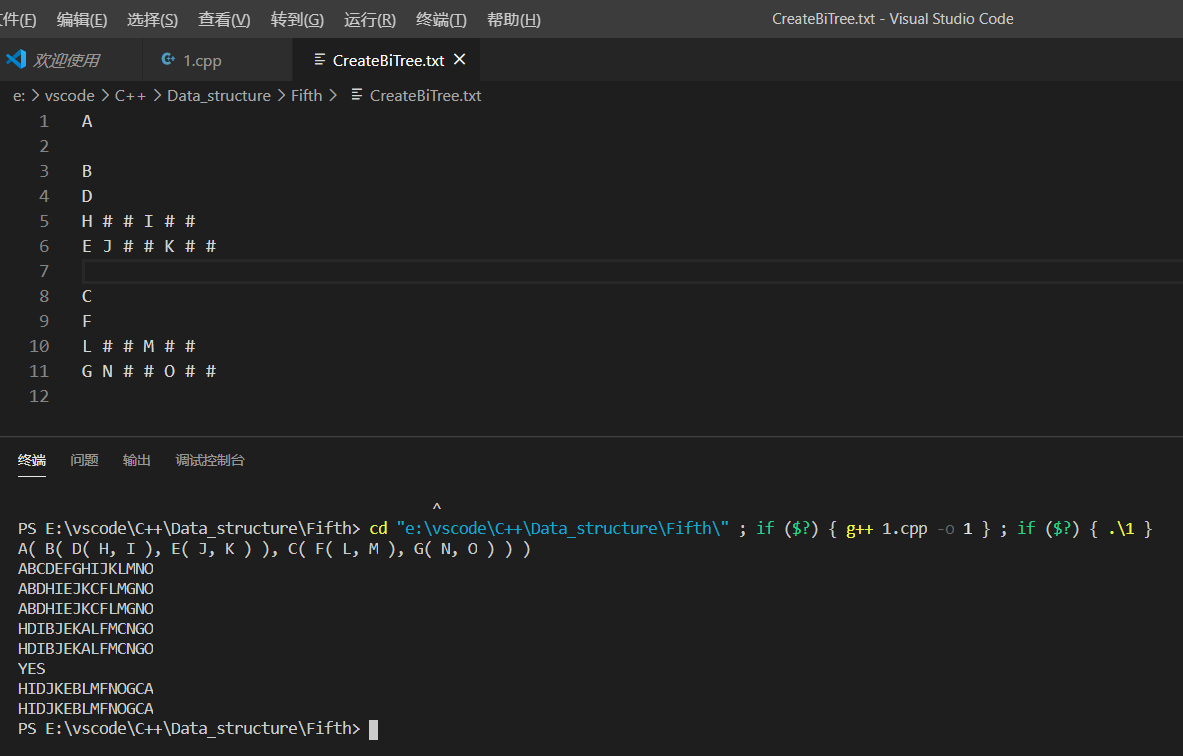
一、调试成功程序及说明

1、

题目:编程实现书P75 ADT BinaryTree 基本操作20个，用二叉链表结构实现；

算法思想：进行基本的操作

运行结果：



结果分析：运行结果均正确

附源程序。

#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)

{

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

int main()

{

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;

LevelOrderTraverse(BT);

cout << endl;

PreOrderTraverse(BT);

cout << endl;

PreOrderTraverseWithoutRecursion(BT);

cout << endl;

InOrderTraverse(BT);

cout << endl;

InOrderTraverseWithoutRecursion(BT);

cout << endl;

cout << "YES" << endl;

PostOrderTraverse(BT);

cout << endl;

PostOrderTraverseWithoutRecursion(BT);

cout << endl;

/\*DestroyBiTree(BT);

LevelOrderTraverse(BT);

int depth = 0;

depth=BiTreeDepth(BT);

cout << depth << endl;

TElemtype e = '0';

Root(BT, e);

cout << e << endl;

BiTNode\* p = BT->lchild->rchild->rchild;

Value(BT, p, e);

cout << e << endl;

Assign(BT, p, '9');

LevelOrderTraverse(BT);

cout << endl;

DeleteChild(BT, p, 1);

LevelOrderTraverse(BT);

cout << endl;\*/

/\* LeftChild(BT, p, q);

cout << "p is " << p->data << " and it's leftChild is " << q->data << endl;\*/

/\* BiTNode\* q;

Parent(BT, p, q);

cout <<"p is "<< p->data<< " and it's parent is " << q->data << endl;

LeftBrother(BT, p, q);

cout << "p is " << p->data << " and it's leftBrother is " <<q->data<< endl;

Assign(BT, BT->rchild, 'D');

LevelOrderTraverse(BT);

cout << endl;

Pruning(BT,'D',BT);

LevelOrderTraverse(BT);

Pruning(BT, 'D', BT);

LevelOrderTraverse(BT);

cout << endl;

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

return 0;

}

2、

题目：实现二叉树的先序、中序、后序遍历，用递归和非递归方法；实现层次遍历。

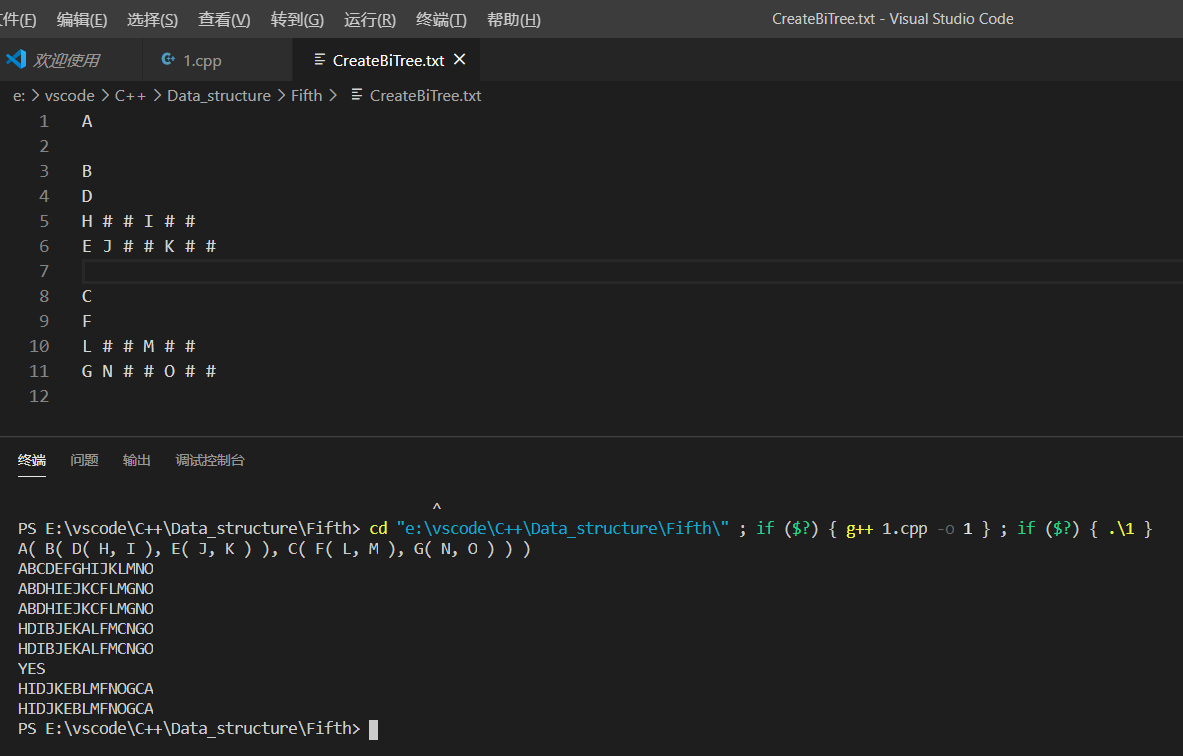
算法思想：用栈实现递归的操作。

①先序遍历:访问根结点，右儿子压栈，左儿子压栈，弹栈访问，直至栈空。

②中序遍历:（1）如果此结点有左儿子，将这个结点压栈，并对这个儿子进行上述判断。（2）压栈后访问栈顶，如果栈顶元素有右儿子，那么右儿子压栈（3）直至循环结束

③（1）将根节点压栈（2）访问栈顶元素，如果这个元素没有孩子，或者上一个访问的元素是他的孩子，那么访问这个孩子。（3）不满足（2）的条件就将当期结点的右儿子和左儿子先后压栈。

运行结果：



结果分析：运行结果正确

附源程序。

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;

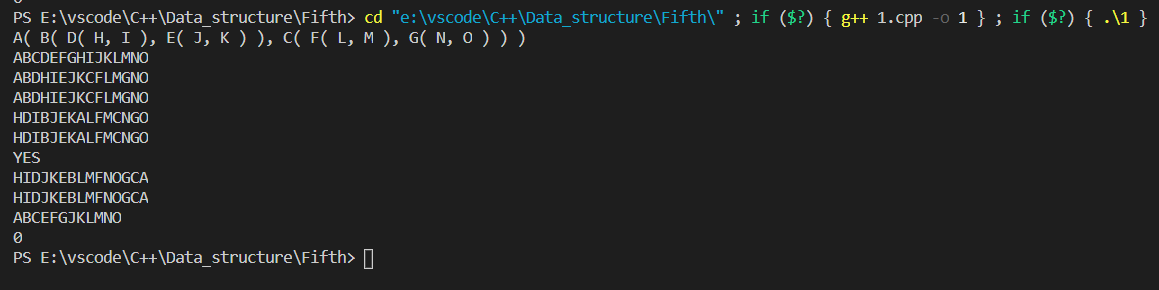
}

3、

题目：设二叉树采用二叉链表存储，编写函数，对二叉树中每个元素值为x的结点，删除以它为根的子树，并释放相应空间。（习题集6.45）

算法思想：这个函数我们可以调用已有的销毁子树的函数，我们进行先序递归遍历，如果这个结点等于要查找的值，那么我们销毁这棵子树，否则继续进行递归遍历。

运行结果：



结果分析：运行结果正确

附源程序。

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

{

if (T == NULL)

{

return;

}

if (T->data == x)

{

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;

}

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;

}

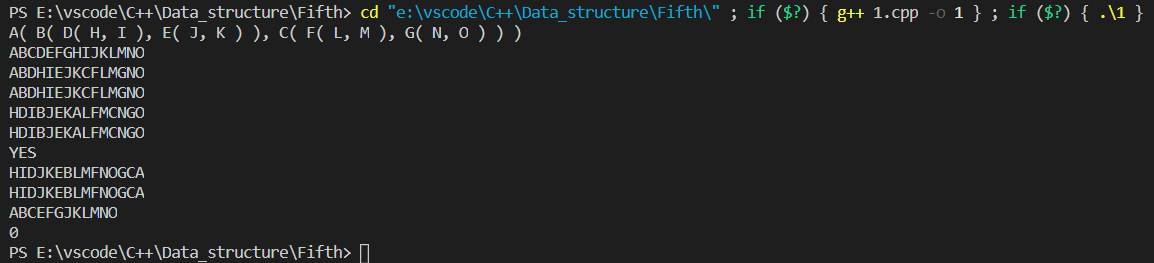
//销毁一颗树

4、

题目：编写函数，判断给定的二叉树是否是完全二叉树。（习题集6.49）

算法思想：对于每一个结点①如果他没有左儿子，只有右儿子，那么这棵树不是完全二叉树。②度为一的结点最有可以有一个，再多的话就不是完全二叉树③如果度为1的结点不是最后一个被访问的，那么这棵树不是完全二叉树。

运行结果：



结果分析：运行结果正确

附源程序。

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;

}

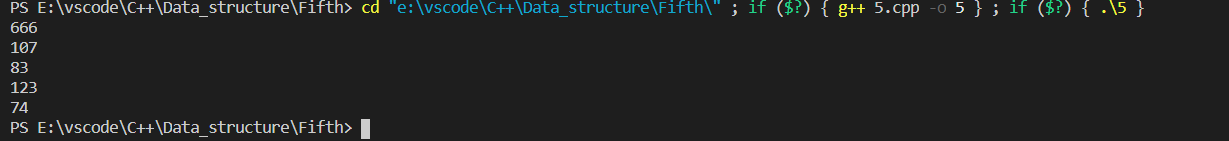
5、

题目：题目描述：甲乙丙丁决定玩一个报数的游戏来打发时间。游戏规则为四个人从1开始轮流进行报数，但如果需要报出的数是7的倍数或含有数字7则直接跳过。

此外大家约定，在总共报出了n个数后（不计入被跳过的数）游戏结束。现在需要你来帮忙统计，游戏过程中每个人各自跳过了几次。

算法思想：开一个数组，下标表示报数的数字，内容标记这个数字是不是应该被跳过去。然后后遍历这个数组，如果这个数字不应该跳过，那么所报数字加一，如果这个数字要跳过，那么所对应的人被跳过的次数加一。

运行结果：



结果分析：运行结果正确，通过csp模拟测试。

附源程序。

#include <iostream>

using namespace std;

bool map[2005];

int main()

{

int p = 0,q = 0;

for (int i = 1; i <= 2000; i++)

{

p = i / 10;

q = i / 100;

if(i % 7 == 0)

{

map[i] = true;

}

if((i % 10 == 7) ||(p % 10 == 7)||(q == 7))

{

map[i] = true;

}

if(i % 100 == 7)

{

map[i] = true;

}

}

int n;

int cnt1 = 0;

int cnt2 = 0;

int cnt3 = 0;

int cnt4 = 0;

int cnt = 0;

cin >> n;

for (int i = 1; i <= 2000; i++)

{

if(cnt == n)

{

break;

}

if(!map[i])

{

cnt++;

}

if(map[i])

{

if(i % 4 == 0)

{

cnt4++;

}

if(i % 4 == 1)

{

cnt1++;

}

if(i % 4 == 2)

{

cnt2++;

}

if(i % 4 == 3)

{

cnt3++;

}

}

}

cout << cnt1<< endl;

cout << cnt2<< endl;

cout << cnt3<< endl;

cout << cnt4;

return 0;

}

6、

题目：

算法思想：

运行结果：

结果分析：

附源程序。

7、

题目：

算法思想：

运行结果：

结果分析：

附源程序。

8、

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

