#include <iostream>

#include <string>

#include <queue>

#include <stack>

using namespace std;

#define MAX 100 //前中序输入的最大数目

static string outputString = "";

template<class T>

class BinaryTreeNode

{

public:

BinaryTreeNode() { LeftChild = RightChild = 0; }

BinaryTreeNode(const T&e) { data = e; LeftChild = RightChild = 0; }

BinaryTreeNode(const T&e, BinaryTreeNode \*l, BinaryTreeNode \*r) { data = e; LeftChild = l; RightChild = r; }

T data;

BinaryTreeNode<T>\*LeftChild; //左子树

BinaryTreeNode<T>\*RightChild; //右子树

};

template<class T>

class BinaryTree

{

public:

BinaryTree() { root = 0; }

~BinaryTree() {};

bool IsEmpty()const { return ((root) ? false : true); } //如果root存在则为false，也就是不为空。

bool Root(T&x)const;

void MakeTree(const T&element, BinaryTree<T>&left, BinaryTree<T>&right);

void MakeTree(BinaryTreeNode<T>\*r) { root = r; }

void BreakTree(T&element, BinaryTree<T>&left, BinaryTree<T>&right);

void PreOrder(void(\*Visit)(BinaryTreeNode<T>\*u)) { PreOrder(Visit, root); }

void InOrder(void(\*Visit)(BinaryTreeNode<T>\*u)) { InOrder(Visit, root); }

void PostOrder(void(\*Visit)(BinaryTreeNode<T>\*u)) { PostOrder(Visit, root); }

void LevelOrder(void(\*Visit)(BinaryTreeNode<T>\*u)) { LevelOrder(Visit, root); };

//二叉树类的扩充

void PreOutput();

void InOutput();

void PostOutput();

void LevelOutput();

void Delete(); //删除二叉树并释放其节点

void AddNode(const T&u); //删除二叉树并释放其节点

int Height(); //返回树的高度

int Size(); //返回树中的节点数

int getSize(BinaryTreeNode<T> \*p);

int TreeHeight(BinaryTreeNode<T> \*root);

BinaryTreeNode<T>\*root;

void PreOrder(void(\*Visit)(BinaryTreeNode<T>\*u), BinaryTreeNode<T>\*t);

void InOrder(void(\*Visit)(BinaryTreeNode<T>\*u), BinaryTreeNode<T>\*t);

void PostOrder(void(\*Visit)(BinaryTreeNode<T>\*u), BinaryTreeNode<T>\*t);

void LevelOrder(void(\*Visit)(BinaryTreeNode<T>\*u), BinaryTreeNode<T>\*t);

};

//取根节点的data域

//如果没有根节点则返回false

template<class T>

bool BinaryTree<T>::Root(T&x)const

{

if (root)

{

x = root->data;

return true;

}

else

{

return false;

}

}

//将left，right，element合并成一颗新树

//left、right和this必须是不同的树

template<class T>

void BinaryTree<T>::MakeTree(const T&element, BinaryTree<T>&left, BinaryTree<T>&right)

{

root = new BinaryTreeNode<T>(element, left.root, right.root);

left.root = right.root = 0; //禁止通过其他途径访问left和right

}

void BadInput() {

cout << "Bad Input!" << endl;

}

//将this拆分成left、right和element

//left、right和this必须是不同的树

template<class T>

void BinaryTree<T>::BreakTree(T&element, BinaryTree<T>&left, BinaryTree<T>&right)

{

if (root)

throw BadInput();//空树

//分解树

element = root->data;

left.root = root->LeftChild;

right.root = root->RightChild;

delete root;

root = 0;

}

//静态成员函数Output输出树

template<class T>

static void Output(BinaryTreeNode<T>\*t)

{

outputString += t->data;

outputString += ",";

}

template<class T>

void BinaryTree<T>::PreOutput()

{

outputString = "";

PreOrder(Output, root);

outputString.erase(outputString.end() - 1);

cout << outputString << endl;

}

template<class T>

void BinaryTree<T>::InOutput()

{

outputString = "";

InOrder(Output, root);

outputString.erase(outputString.end() - 1);

cout << outputString << endl;

}

template<class T>

void BinaryTree<T>::PostOutput()

{

outputString = "";

PostOrder(Output, root);

outputString.erase(outputString.end() - 1);

cout << outputString << endl;

}

template<class T>

void BinaryTree<T>::LevelOutput()

{

outputString = "";

LevelOrder(Output, root);

outputString.erase(outputString.end() - 1);

cout << outputString << endl;

}

template<class T>

void BinaryTree<T>::PreOrder(void(\*Visit)(BinaryTreeNode<T>\*u), BinaryTreeNode<T>\*t)

{

if (t)

{

Visit(t);

PreOrder(Visit, t->LeftChild);

PreOrder(Visit, t->RightChild);

}

}

template<class T>

void BinaryTree<T>::InOrder(void(\*Visit)(BinaryTreeNode<T>\*u), BinaryTreeNode<T>\*t)

{

if(t)

{

InOrder(Visit,t->LeftChild);

Visit(t);

InOrder(Visit,t->RightChild);

}

}

template<class T>

void BinaryTree<T>::PostOrder(void(\*Visit)(BinaryTreeNode<T>\*u), BinaryTreeNode<T>\*t)

{

if (t)

{

PostOrder(Visit, t->LeftChild);

PostOrder(Visit, t->RightChild);

Visit(t);

}

}

template<class T>

void BinaryTree<T>::LevelOrder(void(\*Visit)(BinaryTreeNode<T>\*u), BinaryTreeNode<T>\*t)

{

queue<BinaryTreeNode<T>\*>myQueue;

while (t)

{

Visit(t);

if (t->LeftChild)

myQueue.push(t->LeftChild);

if (t->RightChild)

myQueue.push(t->RightChild);

if (!myQueue.empty())

{

t = myQueue.front();

myQueue.pop();

}

else

{

break;

}

}

}

template<class T>

void BinaryTree<T>::AddNode(const T&u) {

if (!root)

{

root = new BinaryTreeNode<T>;

root->data = u;

root->LeftChild = 0;

root->RightChild = 0;

return;

}

BinaryTreeNode<T> \*newNode = new BinaryTreeNode<T>;

newNode->data = u;

queue<BinaryTreeNode<T>\*>myQueue;

BinaryTreeNode<T>\*t;

t = root;

while (t)

{

if (t->LeftChild) {

myQueue.push(t->LeftChild);

}

if (t->RightChild) {

myQueue.push(t->RightChild);

}

if (!t->LeftChild)

{

t->LeftChild = newNode;

return;

}

else {

}

if (!t->RightChild)

{

t->RightChild = newNode;

return;

}

else {

}

if (!myQueue.empty())

{

t = myQueue.front();

myQueue.pop(); //队列中删除一个节点并且将其赋值给t

}

}

}

template<class T>

static void Free(BinaryTreeNode<T>\*t)

{

delete t;

}

template<class T>

void BinaryTree<T>::Delete()

{

PostOrder(Free, root);

root = 0;

}

template<class T>

int BinaryTree<T>::Height() {

return TreeHeight(root);

}

template<class T>

int BinaryTree<T>::TreeHeight(BinaryTreeNode<T> \*root) {

if (root == NULL

)

return 0;

else

return 1+ max(TreeHeight(root->LeftChild), TreeHeight(root->RightChild));

}

template<class T>

int BinaryTree<T>::Size()

{

return getSize(root);

}

template<class T>

int BinaryTree<T>::getSize(BinaryTreeNode<T> \*p)

{

int con = 0;

if (p != NULL)

{

con++;

con += getSize(p->LeftChild);

con += getSize(p->RightChild);

}

return con;

}

BinaryTree<char> a, x, y, z;

template<class T>

void countTree(BinaryTreeNode<T>\*t)

{

int count = 0;

count++;

}

BinaryTreeNode<char>\* creat(char \*pre, char \*in, int len)

{

int k;

if (len <= 0)

return NULL;

BinaryTreeNode<char> \*head = new BinaryTreeNode<char>;

head->data = \*pre;

char \*p;

for (p = in; \*p != NULL; p++) //应该是\*p!= NULL, 而不是p!=NULL

if (\*p == \*pre)

break;

//判断失败

if (\*p == NULL)

{

printf("NO ANSWER.\n");

return NULL;

}

k = p - in;

head->LeftChild = creat(pre + 1, in, k);

head->RightChild = creat(pre + k + 1, p + 1, len - k - 1);

return head;

}

int main()

{

cout << "Input1" << endl;

string myInput = " ";

cin >> myInput;

int which = 0;

while (myInput[which])

{

char temp = myInput[which];

y.AddNode(temp);

which++;

}

cout << "Output1" << endl;

y.PreOutput();

y.InOutput();

y.PostOutput();

cout << y.Size() << endl;

cout << y.Height() << endl;

cout << "Input2" << endl;

string input1;

cin >> input1;

string input2;

cin >> input2;

char\*preOrder = (char\*)input1.c\_str();

char\*inOrder = (char\*)input2.c\_str();

BinaryTreeNode<char>\*myRoot = creat(preOrder, inOrder, input2.length());

BinaryTree<char>newTree;

newTree.MakeTree(myRoot);

cout << "Output2" << endl;

newTree.PostOutput();

newTree.LevelOutput();

cout << "End" << endl;

return 0;

}