中缀式转二叉树表达式

题意：

输入说明 :

第一行：中缀表达式字符串

输出说明 :

第一行：表达式二叉树前序遍历结果（前缀式）

第二行：表达式二叉树中序遍历结果（中缀式，未带加括号处理）

第三行：表达式二叉树后序遍历结果（后缀式）

#include<iostream>

#include<cstring>

#include<sstream>

#include<queue>

#include<stack>

#include<utility>

#include<cstdio>

#include<iterator>

using namespace std;

//二叉树ADT原型参考如下：

/\* 二叉表的结点定义 \*/

template<class ElemType>

struct BinaryTreeNode

{

ElemType data;

BinaryTreeNode<ElemType>\* LChild, \* RChild;

BinaryTreeNode() : LChild(NULL), RChild(NULL) {} //构造函数1，用于构造根结点

BinaryTreeNode(const ElemType& item, BinaryTreeNode<ElemType>\* Lptr = NULL, BinaryTreeNode<ElemType>\* Rptr = NULL) //构造函数2，用于构造其他结点

//函数参数表中的形参允许有默认值，但是带默认值的参数需要放后面

{

LChild = Lptr;

RChild = Rptr;

data = item;

}

ElemType getData() { return data; } //取得结点中的数据

void SetLChild(BinaryTreeNode<ElemType>\* link) { LChild = link; } //修改结点的左孩子域

void SetRChild(BinaryTreeNode<ElemType>\* link) { RChild = link; } //修改结点的右孩子域

void SetData(ElemType value) { data = value; } //修改结点的data域

BinaryTreeNode<ElemType>\* GetLChild() const { return LChild; } //获取左孩子结点

BinaryTreeNode<ElemType>\* GetRChild() const { return RChild; } //获取左孩子结点

};

//用于辅助完全二叉树的一些相关操作

template<class ElemType>

struct elem {

BinaryTreeNode<ElemType>\* node;//node用于指示二叉树中的某个结点

int num;//二叉树中的某个结点按层序遍历时的标号

elem<ElemType>() :node(NULL), num(0) {}

};

//二叉树

template<class ElemType>

class BinaryTree {

private:

BinaryTreeNode<ElemType>\* root; // 头指针

void BinaryTreeDestroy\_Cursive(BinaryTreeNode<ElemType>\* T); //销毁树（递归准备，private）

public:

//一、构造类函数：

//1.无参数的构造函数

BinaryTree<ElemType>() : root(NULL) {}

//2.带参数的构造函数

BinaryTree<ElemType>(const ElemType& item) { root = new BinaryTreeNode<ElemType>(item); }

//3.拷贝构造函数（外壳部分）

BinaryTree(const BinaryTree<ElemType>& B);

//复制二叉树（实际递归部分）

BinaryTreeNode<ElemType>\* Copy\_BinaryTree(const BinaryTree<ElemType>& B, BinaryTreeNode<ElemType>\* Tb, BinaryTreeNode<ElemType>\* Ta);

//\*运算符重载函数：

//4.拷贝赋值运算符：

BinaryTree<ElemType>& operator=(const BinaryTree<ElemType>& B);

//5.重载运算符==（递归外壳，建议用户调用的部分）

bool operator==(const BinaryTree<ElemType>& B)const;

//运算符==重载函数

//实际的递归部分

bool checkEqual(const BinaryTreeNode<ElemType>\* root1,const BinaryTreeNode<ElemType>\* root2)const;

//二、生成树类相关操作：

//生成树

void makeBinaryTree(const ElemType& item, BinaryTreeNode<ElemType>\*& left, BinaryTreeNode<ElemType>\* &right);

//1.建立二叉树的存储结构

BinaryTreeNode<ElemType>\* createBinaryTree(ElemType x[], const ElemType& empty, int& n);

//三、销毁删除类函数：

//1.析构函数

~BinaryTree() { BinaryTreeDestroy(); /\*cout << "调用了析构函数" << endl;\*/ }

//2.销毁树

bool BinaryTreeDestroy();

//3.销毁以值等于x的结点为根结点的子树的外壳函数（建议用户调用的函数）

bool delete\_x\_child(const ElemType& x, const int& flag);

//销毁以T为根结点的子树（作为上一个函数的递归函数）

bool delete\_root\_tree(BinaryTreeNode<ElemType>\* T);

//四、信息获取类操作：

//a、与高度相关的函数：

//返回以元素值x为根结点的（子）二叉树的高度（实际递归做事的部分）

unsigned getBinaryTreeHeight\_Cursive(BinaryTreeNode<ElemType>\* T, const ElemType& x) const;

//1.计算以与x相等的值为根结点的子树的高度（建议用户直接调用的函数）

unsigned getBinaryTreeHeight(const ElemType& x) const;

//计算以与x相等的值为根结点的子树的高度（实际的递归函数）

unsigned getHeight(BinaryTreeNode<ElemType>\* T)const;

//b、结点个数相关函数：

//1.统计结点个数

int BinaryTreeSize(BinaryTreeNode<ElemType>\* T) const;

//2.统计度为2的结点个数

unsigned CountDegreeTwo(BinaryTreeNode<ElemType>\* T) const;

//3.统计叶子结点个数（度为0的结点个数）

unsigned CountLeaves()const;

//c、判断类函数：

//1.判断二叉树是否为空

bool isEmpty() const { return root == NULL; }

//2.判断是否是完全二叉树

bool isCompleteTree() const;

//d、根结点信息获取类函数：

//1.获取根结点

BinaryTreeNode<ElemType>\* getRoot() const { return root; }

//2.获取根结点元素值

ElemType getRootData() const { return root->data; }

//五、查找类函数：

//1.按前序遍历查找值为x的结点的位置，返回的值的类型是一个指针，若二叉树中有多个结点的值为x，则只返回按前序遍历先遇到的那个的位置

//因此使用时要注意

BinaryTreeNode<ElemType>\* get\_x\_location(const ElemType& x)const;

//查找值为x的结点的位置（递归）

void Location\_Cursive(BinaryTreeNode<ElemType>\* T, const ElemType& x, BinaryTreeNode<ElemType>\*& location)const;

//2.查找值为x的结点的孩子结点的指针（flag=0，左孩子；flag=1，右孩子）

BinaryTreeNode<ElemType>\* location\_Child(BinaryTreeNode<ElemType>\* parent, const ElemType& x, int& flag) const;

//3.获取值为x的结点的父亲结点的外壳函数，用户一般通过这个调用，书写形式较为简单

BinaryTreeNode<ElemType>\* getParent(const ElemType& x)const;

//获取值为x的结点的父亲结点（此时对应的结点不为parent如果为parent，则使得到的结点为NULL）

//实际用于递归的部分，外面还有一个外壳函数是用户实际调用的函数

void GetParent\_Cursive(BinaryTreeNode<ElemType>\* parent, const ElemType& x, BinaryTreeNode<ElemType>\*& result) const;

//返回指向值为x的结点的兄弟结点的指针（flag=0，左兄弟；flag=1，右兄弟）

BinaryTreeNode<ElemType>\* getSibling(const ElemType& x, const int& flag)const;

//5.查找从根结点到某一个等于x的结点的路径（x是按照前序遍历找出来的）

//并将最终的结果路径放到一个栈中（为了最后的路径输出）（注意：该栈可能为空）

void findPath(const ElemType& x, stack<BinaryTreeNode<ElemType>\* >& s)const;

//六、插入类函数

//1.在以元素值x为根结点的位置插入子树（外壳，flag=0，左子树；flag=1，右子树）

bool Insert\_ChildTree(BinaryTreeNode<ElemType>\* T, const ElemType& x, BinaryTreeNode<ElemType>\* child, const int& p\_flag, const int& c\_flag);

//2.在以元素值为x的结点的位置插入子树（flag=0，左子树；flag=1，右子树） （实际递归的函数）

BinaryTreeNode<ElemType>\* insert\_ChildTree(BinaryTreeNode<ElemType>\* child);

//这是相关操作的外壳函数（建议用户直接调用的部分）

//如果没找到x或者待插入的子树不符合要求（即有右子树），则返回false

bool insert\_child\_at\_x(const BinaryTree<ElemType>& C, const ElemType& x, const int& flag);

//3.交换二叉树中所有结点的左右子树 （外壳）

void BinaryTree\_Revolute();

//交换二叉树中所有结点的左右子树 （递归部分）

void BinaryTree\_Revolute\_Cursive(BinaryTreeNode<ElemType>\* T);

//七、关于结点改变的函数

//1.设置根结点

void setRoot(BinaryTreeNode<ElemType>\* p) { root = p; }

//2.给元素值为x的结点重新赋值

void Assign\_NodeData(BinaryTreeNode<ElemType>\* T, const ElemType& x, const ElemType& value);

//八、遍历类函数：

//1.前序遍历

bool preOrderTraverse(BinaryTreeNode<ElemType>\* T/\*, bool (\*visit)(BinaryTreeNode<ElemType>\* T)\*/) const; //前序遍历（递归）

//2.中序遍历

bool inOrderTraverse(BinaryTreeNode<ElemType>\* T/\*, bool (\*visit)(BinaryTreeNode<ElemType>\* T)\*/) const; //中序遍历（递归）

//3.后序遍历

bool postOrderTraverse(BinaryTreeNode<ElemType>\* T/\*, bool (\*visit)(BinaryTreeNode<ElemType>\* T)\*/) const; //后序遍历（递归）

//4.按树状打印出二叉树的形状（递归};

void Print\_BinaryTree(BinaryTreeNode<ElemType>\* root, int i); //按树状打印出二叉树的形状（递归）

//5.二叉树的层次遍历（先左后右）

bool LayerOrderTraverse(/\*bool (\*visit)(BinaryTreeNode<ElemType>\* T)\*/) const;

//非递归实现前、中、后遍历一个二叉树

void preOrderTraverse()const;

void inOrderTraverse()const;

void postOrderTraverse()const;

};

//一、构造类：

//拷贝构造函数（外壳部分）

template<class ElemType>

BinaryTree<ElemType>::BinaryTree(const BinaryTree<ElemType>& B) {

BinaryTreeNode<ElemType>\* Tb = B.getRoot();

if (Tb == NULL) {

return;

}

else {

root = new BinaryTreeNode<ElemType>;//先分配好空间

BinaryTreeNode<ElemType>\* Ta = root;//Ta、Tb最初分别指向待创建的二叉树的根结点和B的根结点

Ta->data = Tb->data;

if (Tb->LChild == NULL && Tb->RChild == NULL) {

return;

}

else {

Ta->LChild = this->Copy\_BinaryTree(B, Tb->LChild, Ta->LChild);

Ta->RChild = this->Copy\_BinaryTree(B, Tb->RChild, Ta->RChild);

}

}

}

//复制函数（将一个二叉树对象拷贝到另一个二叉树对象中）

template<class ElemType>

BinaryTreeNode<ElemType>\* BinaryTree<ElemType>::Copy\_BinaryTree(const BinaryTree<ElemType>& B, BinaryTreeNode<ElemType>\* Tb, BinaryTreeNode<ElemType>\* Ta) {

if (Tb != NULL) {

BinaryTreeNode<ElemType>\* tmp = new BinaryTreeNode<ElemType>;

tmp->data = Tb->data;

Ta = tmp;

Ta->LChild = this->Copy\_BinaryTree(B, Tb->LChild, Ta->LChild);

Ta->RChild = this->Copy\_BinaryTree(B, Tb->RChild, Ta->RChild);

return Ta;

}

else {

return NULL;

}

}//暂时经过测试认为没有问题

//拷贝赋值运算符：

template<class ElemType>

BinaryTree<ElemType>& BinaryTree<ElemType>::operator=(const BinaryTree<ElemType>& B) {

BinaryTreeNode<ElemType>\* Tb = B.getRoot();

if (Tb == NULL) {

this->root = NULL;

return \*this;

}

else {

root = new BinaryTreeNode<ElemType>;

BinaryTreeNode<ElemType>\* Ta = root;//Ta、Tb最初分别指向待创建的二叉树的根结点和B的根结点

Ta->data = Tb->data;

if (Tb->LChild || Tb->RChild) {

Ta->LChild = this->Copy\_BinaryTree(B, Tb->LChild, Ta->LChild);

Ta->RChild = this->Copy\_BinaryTree(B, Tb->RChild, Ta->RChild);

}

}

/\*cout << "调用了赋值运算符重载函数" << endl;\*/

return \*this;

}

//运算符==重载函数（递归部分）

template<class ElemType>

bool BinaryTree<ElemType>::checkEqual(const BinaryTreeNode<ElemType>\* root1,

const BinaryTreeNode<ElemType>\* root2)const {

if (root1 && root2) {

if (root1->data != root2->data) {

return false;

}

else {

bool fg = this->checkEqual(root1->LChild, root2->LChild);

if (fg) {

fg = this->checkEqual(root1->RChild, root2->RChild);

return fg;

}

else {

return false;

}

}

}

else if (root1 == NULL && root2 == NULL) {

return true;

}

else {

return false;

}

}

//重载运算符==（外壳部分）（实际调用的）

template<class ElemType>

bool BinaryTree<ElemType>::operator==(const BinaryTree<ElemType>& B)const {

BinaryTreeNode<ElemType>\* pa = this->getRoot();

BinaryTreeNode<ElemType>\* pb = B.getRoot();

if (pa && pb) {

if (pa->data != pb->data) {

return false;

}

else {

bool fg = this->checkEqual(pa->LChild, pb->LChild);

if (fg) {

fg = this->checkEqual(pa->RChild, pb->RChild);

return fg;

}

else {

return false;

}

}

}

else if (pa == NULL && pb == NULL) {

return true;

}

else {

return false;

}

}

//二、生成二叉树类：

//2、生成树

/\*template<class ElemType>//暂时放下不太会写

void BinaryTree<ElemType>::makeBinaryTree(const ElemType& item, BinaryTreeNode<ElemType>\* &left,

BinaryTreeNode<ElemType>\* &right) {//item为这个结点的值，left为指向左边子树的根结点的指针，right为指向右边子树的根结点的指针

root = new BinaryTreeNode<ElemType>;

root->data = item;

root->LChild = left;

root->RChild = right;

}\*/

//2、建立二叉树的存储结构 (递归部分，成员函数）

//注意，这里的创建规则不是按我们所习惯的一层一层所创建二叉树的，而是采用了前序创建的规则，先创建出一个根结点

//接着再创建根结点的左边部分，左边部分整体都创建完以后再整体创建根结点的右边部分，所以采用这种方式所创建出来

//的二叉树的顺序和前序遍历的结果一致，这点必须特别注意：只要左边的接收到的标志不是结束标志，就一直创建左边的

//只有当左边接受到了结束标志，才创建右边的

template<class ElemType>

BinaryTreeNode<ElemType>\* BinaryTree<ElemType>::createBinaryTree(ElemType x[], const ElemType& empty, int& n) {

ElemType element = x[n];

n++;

if (element == empty) {

return NULL;

}

else {

BinaryTreeNode<ElemType>\* Node = new BinaryTreeNode<ElemType>;

Node->data = element;

Node->LChild = createBinaryTree(x, empty, n);

Node->RChild = createBinaryTree(x, empty, n);

return Node;

}

}

//建立二叉树的存储结构 （外壳部分，用户函数）

//注意这个不是类内的函数

template<class ElemType>

void CreateTree(BinaryTree<ElemType>& T, const string& str, ElemType& empty) {

ElemType tmp, t[100];

int num = 0;

istringstream input\_T(str);

while (input\_T >> tmp) {//这里是把str字符串中的内容以空格分开并转化为tmp类型依次读到tmp中

t[num] = tmp;

num++;

}

BinaryTreeNode<ElemType>\* root;

num = 0;//num

root = T.createBinaryTree(t, empty, num);

T.setRoot(root);

}

//三、销毁删除类：

//1.销毁树

//按照后续遍历的顺序销毁树

template<class ElemType>

bool BinaryTree<ElemType>::BinaryTreeDestroy() {

BinaryTreeDestroy\_Cursive(this->root);

return true;

}

//实际执行销毁操作的private函数：

//相当于上面1的递归部分

template<class ElemType>

void BinaryTree<ElemType>::BinaryTreeDestroy\_Cursive(BinaryTreeNode<ElemType>\* T) { //销毁树（递归准备，private）

if (T != NULL) {

BinaryTreeDestroy\_Cursive(T->LChild);

BinaryTreeDestroy\_Cursive(T->RChild);

}

delete T;//执行到这一步表明某个结点的左右子树均已销毁完，是时候对这个结点自身下手了

}//经测试暂时认为没有问题

//2.销毁以值等于x的结点为根结点的子树的外壳函数（建议用户调用的函数）

template<class ElemType>

bool BinaryTree<ElemType>::delete\_x\_child(const ElemType& x, const int& flag) {

BinaryTreeNode<ElemType>\* p = this->get\_x\_location(x);

bool result;

if (p == NULL) return false;

else {

if (flag == 0) {

result = delete\_root\_tree(p->LChild);

if (result) {

p->LChild = NULL;

}

}

else {

result = delete\_root\_tree(p->RChild);

if (result) {

p->RChild = NULL;

}

}

return result;

}

}

//销毁以T为根结点的指针

//上一个函数的递归调用部分

template<class ElemType>

bool BinaryTree<ElemType>::delete\_root\_tree(BinaryTreeNode<ElemType>\* T) {

if (T != NULL) {

delete\_root\_tree(T->LChild);

delete\_root\_tree(T->RChild);

delete T;

return true;

}

else {//表明一开始没有找到对应的要删除的结点

return false;

}

}

//四、信息获取类：

//a、与高度相关：

//1、返回以元素值x为根结点的（子）二叉树的高度（递归）

//注意：返回的层数是从1开始的，当树为空或者要查找的元素不在树中时，则返回0

template<class ElemType>

unsigned BinaryTree<ElemType>::getBinaryTreeHeight\_Cursive(BinaryTreeNode<ElemType>\* T, const ElemType& x) const {

unsigned count = 0;//无符号类型的计数器

if (T != NULL) {

if (T->data == x) {

++count;

if (T != this->root)

++count;

return count;

}

else {

int tmp = count;

count += getBinaryTreeHeight\_Cursive(T->LChild, x);

if (count == tmp) {

count += getBinaryTreeHeight\_Cursive(T->RChild, x);

if (count > tmp && T != this->root) {

++count;

}

return count;

}

else {

if (T != this->root)

++count;

return count;

}

}

}

else {

return count;

}

}//暂时没有测出什么问题，不过这个函数极有可能会出问题，要注意

//2、计算以与x相等的值为根结点的子树的高度(外壳函数)，建议用户到时候直接调用这个

template<class ElemType>

unsigned BinaryTree<ElemType>::getBinaryTreeHeight(const ElemType& x) const {

BinaryTreeNode<ElemType>\* p = this->get\_x\_location(x);//p指向所求子树的根结点

return this->getHeight(p);

}

//计算以与x相等的值为根结点的子树的高度（实际的递归函数）

template<class ElemType>

unsigned BinaryTree<ElemType>::getHeight(BinaryTreeNode<ElemType>\* T)const {

if (T == NULL) {

return 0;

}

else {

unsigned count;

count = max(this->getHeight(T->LChild), this->getHeight(T->RChild));

//递归思想，以该结点为根结点的高度等于以其左右孩子结点为根结点的子树的高度的最大值再加一，抓住这个层层递归求得最终的高度

count += 1;

return count;

}//注意：本题 还可以用队列来解，并且队列解不需要递归，队列简直就是解二叉树关于层的问题的标配

}

//b、结点个数相关：

//1、统计结点个数

template<class ElemType>

int BinaryTree<ElemType>::BinaryTreeSize(BinaryTreeNode<ElemType>\* T) const {//采用前序遍历法统计

int count = 0;

if (T != NULL) {

++count;

count += this->BinaryTreeSize(T->LChild);

count += this->BinaryTreeSize(T->RChild);

return count;

}

else {

return count;

}

}

//2、统计度为2的结点个数

template<class ElemType>

unsigned BinaryTree<ElemType>::CountDegreeTwo(BinaryTreeNode<ElemType>\* T) const {

unsigned count = 0;

if (T != NULL) {

count += CountDegreeTwo(T->LChild);

count += CountDegreeTwo(T->RChild);

if (T->LChild != NULL && T->RChild != NULL) {

++count;

return count;

}

else return count;

}

else {

return count;

}

}//目前认为没有问题，但是测试数据过少，有待于进一步测试

//3.统计叶子结点个数（度为0的结点个数）

template<class ElemType>

unsigned BinaryTree<ElemType>::CountLeaves()const {

return this->CountDegreeTwo(this->root)+1;//运用二叉树的性质n0=n2+1；

}

//c、判断类：

//判断二叉树是否为完全二叉树

//判断思想：先运用层序遍历给这个二叉树的每一个结点从1开始进行一个编号，同时顺便统计结点的个数，只有当结点的

//个数每次与编号相等时，并且这个条件满足持续到最后一个结点，那么这个二叉树就是一个完全二叉树，否则就不是

//我该如何判断是终点

template<class ElemType>

bool BinaryTree<ElemType>::isCompleteTree() const {

if (this->root == NULL) {

return true;//空树也算作完全二叉树的一种

}

else {

unsigned count = 0;//用于随时记录遍历到的结点个数

queue<BinaryTreeNode<ElemType>\* >q;

q.push(this->root);

elem<ElemType> e;

int i = 0;

while (!q.empty()) {

BinaryTreeNode<ElemType>\* tmp = q.front();

q.pop();

e.node = tmp;

++e.num;

if (tmp != NULL) {

++count;

q.push(tmp->LChild);

q.push(tmp->RChild);

}

else {//一旦碰到了NULL就立即跳出，碰到NULL无非有两种情况，如果这确实是一个完全二叉树的话，那么碰到的

//NULL就应该是最后一层靠右侧的某个结点开始为NULL，或者是是一棵满二叉树，其最后一层的每一个结点

//的左右子树都为NULL，我们在上述操作把这些NULL也放入了队列，这个时候就要对马上停止继续放的操作

//转而检查队列，如果队列中放的全部都是NULL，那么必为完全二叉树

break;

//而第二种情况就是不是完全二叉树，有某些不该缺的结点有缺失，但是我们在上述操作中把那些缺了的结点

//（以NULL代替也放入了队列），这个时候队列里绝对还有其他的不是NULL的结点，那就立刻停止上述操作，转而

//检查队列，如果发现了不是NULL的结点，这就一定不是一个完全二叉树

}

}

while (!q.empty()) {

BinaryTreeNode<ElemType>\* tmp = q.front();

if (tmp != NULL) {

return false;

}

else {

q.pop();

}

}

return true;

}

}

//d、根结点信息获取类：（略，类中已实现）

//五、查找类：

//查找值为x的结点的位置（递归）

//注意：如果树中有多个值等于x，那么我们只返回按照前序遍历找到的等于x的位置的值

//如果树为空或找不到等于x的值，就让location等于NULL

//还要注意的是：location是指针的引用

//1.查找值为x的结点的函数的递归外壳部分，仅是因为形式比较简单，方便用户调用

template<class ElemType>

BinaryTreeNode<ElemType>\* BinaryTree<ElemType>::get\_x\_location(const ElemType& x)const {

BinaryTreeNode<ElemType>\* location = NULL;

this->Location\_Cursive(this->root, x, location);

return location;

}

//上一个函数的递归部分

template<class ElemType>

void BinaryTree<ElemType>::Location\_Cursive(BinaryTreeNode<ElemType>\* T,

const ElemType& x, BinaryTreeNode<ElemType>\*& location)const {

if (T != NULL) {

if (T->data == x) {

location = T;

return;

}

else {

Location\_Cursive(T->LChild, x, location);

if (location == NULL)

Location\_Cursive(T->RChild, x, location);

}

}

else {

location = NULL;

return;

}

}

//2、查找值为x的结点的孩子结点的指针（flag=0，左孩子；flag=1，右孩子）

template<class ElemType>

BinaryTreeNode<ElemType>\* BinaryTree<ElemType>::location\_Child(BinaryTreeNode<ElemType>\* parent,

const ElemType& x, int& flag) const {

BinaryTreeNode<ElemType>\* location = parent;

if (parent != NULL) {

if (parent->data == x) {

location = parent;

if (flag == 0) {//要的是左孩子

flag = 2;

return location->LChild;

}

else {//要的是右孩子

flag = 2;

return location->RChild;

}

}

else {

location = location\_Child(parent->LChild, x, flag);

if (flag != 2) {

location = location\_Child(parent->RChild, x, flag);

}

return location;

}

}

else {

return NULL;

}

}//经测试基本没有什么问题

//3、获取父结点（外壳部分，建议用户调用的函数，形式简单）

//获取父结点（外壳）

template<class ElemType>

BinaryTreeNode<ElemType>\* BinaryTree<ElemType>::getParent(const ElemType& x)const {

BinaryTreeNode<ElemType>\* result = NULL;

this->GetParent\_Cursive(this->root, x, result);

return result;

}//经测试暂时没问题

//获取父结点（递归部分，成员函数）

//获取父结点（递归）

template<class ElemType>

void BinaryTree<ElemType>::GetParent\_Cursive(BinaryTreeNode<ElemType>\* parent,

const ElemType& x, BinaryTreeNode<ElemType>\*& result) const {

if (parent != NULL) {

if (parent == this->root) {

if (parent->data == x) {

return;

}

else {

if (parent->LChild != NULL) {

if (parent->LChild->data == x) {

result = parent;

return;

}

}

if (parent->RChild != NULL) {

if (parent->RChild->data == x) {

result = parent;

return;

}

}

//执行到这一步，说明上面两步并没有找到：

GetParent\_Cursive(parent->LChild, x, result);

if (result == NULL) {

GetParent\_Cursive(parent->RChild, x, result);

}

}

}

else {//如果parent不是根结点，要现在就把其左右孩子都找一遍，这样保证在找下一个的时候，传入的parent已经找过

if (parent->LChild != NULL) {

if (parent->LChild->data == x) {

result = parent;

}

}

if (parent->RChild != NULL) {

if (parent->RChild->data == x) {

result = parent;

return;

}

}

//执行到这一步，说明上面两步并没有找到：

GetParent\_Cursive(parent->LChild, x, result);

if (result == NULL) {

GetParent\_Cursive(parent->RChild, x, result);

}

}

}

else {//如果传进来的parent结点为NULL，那么直接退出寻找

return;

}

}

//返回指向值为x的结点的兄弟结点的指针（flag=0，左兄弟；flag=1，右兄弟）

template<class ElemType>

BinaryTreeNode<ElemType>\* BinaryTree<ElemType>::getSibling(const ElemType& x, const int& flag)const {

BinaryTreeNode<ElemType>\* parent = this->getParent(x);

if (parent == NULL) {

return NULL;//没有找到值等于x的结点的双亲结点或者是值等于x的结点就是这个子树的根结点，那么其兄弟结点必没有

}

else {//表明双亲结点存在

if (flag == 0) {//看看有没有左兄弟结点

if (parent->LChild->data == x) {

return NULL;

}

else {

return parent->LChild;//此时返回的依然有可能是NULL

}

}

else if (flag == 1) {//看看有没有右兄弟结点

if (parent->LChild->data == x) {

return parent->RChild;

}

else {

return NULL;

}

}

}

}

//5.查找从根结点到某一个等于x的结点的路径（x是按照前序遍历找出来的）

//并将最终的结果路径放到一个栈中（为了最后的路径输出）（注意：该栈可能为空）

template<class ElemType>

void BinaryTree<ElemType>::findPath(const ElemType& x, stack<BinaryTreeNode<ElemType>\* >& s)const {

BinaryTreeNode<ElemType>\* p = this->get\_x\_location(x);

if (p == NULL) { return; }

else if (s.empty()) {

s.push(p);

while (p != this->root) {

p = this->getParent(p->data);

if (p != NULL) {

s.push(p);

}

}

}

}

//六、插入类：

//1、在以元素值x为根结点的位置插入子树（外壳，flag=0，左子树；flag=1，右子树）

//child为传入的指向子树的根结点的指针，p\_flag为指示该位于双亲结点的哪边（0左1右）

//c\_flag为指示原有的双亲结点若有孩子结点，那个以孩子结点为根的子树该插在新插进来子树的哪边（0左1右）

//注意：因为本人水平的限制，所插入的这个子树最多只能有一个结点，即这个child子树只可以有一个根结点，其左右子树的指针均为NULL

template<class ElemType>

bool BinaryTree<ElemType>::Insert\_ChildTree(BinaryTreeNode<ElemType>\* T, const ElemType& x,

BinaryTreeNode<ElemType>\* child, const int& p\_flag, const int& c\_flag) {

//首先要找到那个x的结点，不过如果树中有多个值等于x，那么我们只在按前序遍历所找出来的第一个等于x的结点

//通常靠近左边

if (T != NULL) {

if (T->data == x) {//找到了要插入的子树的位置

BinaryTreeNode<ElemType>\* tmp = NULL;

BinaryTreeNode<ElemType>\* p = new BinaryTreeNode<ElemType>;

if (p\_flag == 0) {//要把child子树插在该树的左子树上

tmp = T->LChild;

p->data = child->data;

T->LChild = p;//注意指针里存储地址的特殊性，这里如果用“T->LChild==child”

//那就会使得程序结束时调用析构函数的时候，先把第一个销毁后再销毁待

//插入的子树时，会发现这个子树已经被销毁，因为这个插入操作使得待插入

//的已经变成了被插入者的一部分，从而发生了试图销毁一个已经被销毁的对象从而引发断点

T = T->LChild;//让插完后的这个指针指向新插进来的子树的根结点

}

else {//要把child子树插在该树的右子树上

tmp = T->RChild;

p->data = child->data;

T->RChild = p;

T = T->RChild;//让插完后的这个指针指向新插进来的子树的根结点

}//即使这个tmp等于NULL，即双亲结点的那一侧没有孩子结点，也不影响接下来按照一般情况直接做

//插完新的子树后，开始把原有的子树插到新插入的子树后，不过

if (c\_flag == 0) {//要插在左子树上

p->LChild = tmp;

}

else {//要插在右子树上

p->RChild = tmp;

return true;

}

}

else {

bool fg = this->Insert\_ChildTree(T->LChild, x, child, p\_flag, c\_flag);

if (fg == false) {

fg = this->Insert\_ChildTree(T->RChild, x, child, p\_flag, c\_flag);

return fg;

}

else {

return true;

}

}

}

else {

return false;

}

}//经测试暂时没有发现什么问题

//2、在以元素值为x的结点的位置插入一棵子树（这棵子树的右子树必须为空，这样原有的插入位置若有子树，就把原来的子树

//插到这棵新插进来子树的右面）

//这是相关操作的外壳函数（建议用户直接调用的部分）

//如果没找到x或者待插入的子树不符合要求（即有右子树），则返回false

template<class ElemType>

bool BinaryTree<ElemType>::insert\_child\_at\_x(const BinaryTree<ElemType>& C, const ElemType& x, const int& flag) {

BinaryTreeNode<ElemType>\* pc = C.getRoot();

if (pc->RChild != NULL) {

return false;

}

else {

BinaryTreeNode<ElemType>\* pb = get\_x\_location(x);

if (pb == NULL) {

return false;

}

else {

if (flag == 0) {

BinaryTreeNode<ElemType>\* tmp = pb->LChild;

pb->LChild = insert\_ChildTree(pc);

pb->LChild->RChild = tmp;

}

else if (flag == 1) {

BinaryTreeNode<ElemType>\* tmp = pb->RChild;

pb->RChild = insert\_ChildTree(pc);

pb->RChild->RChild = tmp;

}

}

}

}

//在以元素值为x的结点的位置插入子树（flag=0，左子树；flag=1，右子树） （实际递归的函数）

template<class ElemType>

BinaryTreeNode<ElemType>\* BinaryTree<ElemType>::insert\_ChildTree(BinaryTreeNode<ElemType>\* child) {//传进来的parent指向待插入的根结点

if (child) {

BinaryTreeNode<ElemType>\* tmp = new BinaryTreeNode<ElemType>;

tmp->data = child->data;

tmp->LChild = this->insert\_ChildTree(child->LChild);

tmp->RChild = this->insert\_ChildTree(child->RChild);

return tmp;

}

else {

return NULL;

}

}

//3.交换二叉树中所有结点的左右子树 （外壳）

template<class ElemType>

void BinaryTree<ElemType>::BinaryTree\_Revolute() {

if (this->root == NULL) {//空树不做处理

return;

}

else if (this->root->LChild == NULL && root->RChild == NULL) {//左右子树皆空没有必要处理

return;

}

else {

BinaryTree\_Revolute\_Cursive(root);//调用递归部分

}

}

//交换二叉树中所有结点的左右子树 （递归部分）

template<class ElemType>

void BinaryTree<ElemType>::BinaryTree\_Revolute\_Cursive(BinaryTreeNode<ElemType>\* T) {//T初次传进来的一定是

//这个二叉树的根结点

if (T->LChild)

BinaryTree\_Revolute\_Cursive(T->LChild);

if (T->RChild)

BinaryTree\_Revolute\_Cursive(T->RChild);

if (T->LChild || T->RChild) {

BinaryTreeNode<ElemType>\* tmp = T->LChild;

T->LChild = T->RChild;

T->RChild = tmp;

}

}

//七、关于结点改变的操作：

//给元素值为x的结点重新赋值为value

//如果不止一个结点的值为x，则将其全部赋值为value

//采用先序遍历遍历整个树

template<class ElemType>

void BinaryTree<ElemType>::Assign\_NodeData(BinaryTreeNode<ElemType>\* T, const ElemType& x, const ElemType& value) {

if (T != NULL) {

if (T->data == x) {

T->data = value;

}

Assign\_NodeData(T->LChild, x, value);

Assign\_NodeData(T->RChild, x, value);

}

else {

return;

}

}//暂时认为没有问题

//八、遍历类：

//1、前序遍历

template<class ElemType>

bool BinaryTree<ElemType>::preOrderTraverse(BinaryTreeNode<ElemType>\* T) const { //前序遍历（递归）问：这个visit函数指针是用来干什么的

if (T != NULL) {

cout << T->data << ' ';

preOrderTraverse(T->LChild);

preOrderTraverse(T->RChild);

return true;

}

else if (T == NULL) {

return false;

}

}

//2、中序遍历

template<class ElemType>

bool BinaryTree<ElemType>::inOrderTraverse(BinaryTreeNode<ElemType>\* T) const { //中序遍历（递归）

if (T != NULL) {

inOrderTraverse(T->LChild);

cout << T->data << ' ';

inOrderTraverse(T->RChild);

return true;

}

else {

return false;

}

}

//3、后序遍历：

template<class ElemType>

bool BinaryTree<ElemType>::postOrderTraverse(BinaryTreeNode<ElemType>\* T) const { //后序遍历（递归）

if (T != NULL) {

postOrderTraverse(T->LChild);

postOrderTraverse(T->RChild);

cout << T->data << ' ';

return true;

}

else {

return false;

}

}

//4、以下这个按树的形状打印输出二叉树是直接套用的老师提供的代码，这个代码写的很好，应当多多揣摩

//按树的形状打印二叉树（实际递归部分）

template<class ElemType>

void BinaryTree<ElemType>::Print\_BinaryTree(BinaryTreeNode<ElemType>\* root, int i) {

if (!root) return;

if (root->RChild) Print\_BinaryTree(root->RChild, i + 1);

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

cout << root->data << " " << endl;

if (root->LChild) Print\_BinaryTree(root->LChild, i + 1);

}

//按树的形状打印二叉树的外壳函数（实际用户调用部分）

template<class ElemType>

bool Print\_BinaryTree(BinaryTree<ElemType>& T) {

BinaryTreeNode<ElemType>\* root;

int layer = 0;

root = T.getRoot();

if (root == NULL) return false;

else T.Print\_BinaryTree(root, layer);

}

//5、层次遍历，成员函数

template<class ElemType>

bool BinaryTree<ElemType>::LayerOrderTraverse(/\*bool (\*visit)(BinaryTreeNode<ElemType>\* T)\*/) const {

if (root != NULL) {

queue<BinaryTreeNode<ElemType>\* >q;//用于存放每一个结点的地址值

q.push(this->root);//根结点入队

while (!q.empty()) {//队列不空

BinaryTreeNode<ElemType>\* tmp = q.front();

q.pop();

cout << tmp->data << ' ';

if (tmp->LChild != NULL) {

q.push(tmp->LChild);

}

if (tmp->RChild != NULL) {

q.push(tmp->RChild);

}

}

return true;

}

else {

return false;

}

}

///前、中、后序遍历的非递归版本

//前序遍历非递归版本（空链表将没有输出）

template<class ElemType>

void BinaryTree<ElemType>::preOrderTraverse()const {

if (!this->isEmpty()) {//该二叉树不空

stack<BinaryTreeNode<ElemType>\* >s;

s.push(this->root);

while (!s.empty()) {

BinaryTreeNode<ElemType>\* tmp = s.top();

s.pop();

cout << tmp->data << ' ';

if (tmp->RChild) {//这里一定要注意入栈的顺序，结合栈后进先出的特点和遍历先左后右的要求应该先放

//右孩子再放左孩子

s.push(tmp->RChild);

}

if (tmp->LChild) {

s.push(tmp->LChild);

}

}

}

else {

return;

}

}

//中序遍历的非递归版本

template<class ElemType>

void BinaryTree<ElemType>::inOrderTraverse()const{

if (!this->isEmpty()) {

stack<BinaryTreeNode<ElemType>\* >s;

s.push(root);

while (!s.empty()) {

BinaryTreeNode<ElemType>\* tmp = s.top();

if (tmp->LChild) {

s.push(tmp->LChild);

}

else {

cout << tmp->data << ' ';

s.pop();

while (!s.empty()&&!tmp->RChild) {

tmp = s.top();

cout << tmp->data << ' ';

s.pop();

}

if(tmp!=NULL&&tmp->RChild!=NULL)

s.push(tmp->RChild);

}

}

}

else {

return;

}

}

//后序遍历的非递归版本

template<class ElemType>

void BinaryTree<ElemType>::postOrderTraverse()const {

if (!this->isEmpty()) {

vector<ElemType>v;

stack<pair<BinaryTreeNode<ElemType>\*, bool > >s;

s.push(make\_pair(this->root, false));

bool visited;

while (!s.empty()) {

BinaryTreeNode<ElemType>\* tmp = s.top().first;

visited = s.top().second;

s.pop();

if (tmp == NULL) {

continue;

}

if(visited) {//表明已经访问过，可以输出

v.push\_back(tmp->data);

}

else {

s.push(make\_pair(tmp, true));

s.push(make\_pair(tmp->RChild, false));

s.push(make\_pair(tmp->LChild, false));

}

}

typename vector<ElemType>::const\_iterator it = v.begin();

for (; it != v.end(); ++it) {

cout << \*it << ' ';

}

}

else {

return;

}

}

//visit函数（具体功能根据实际需要，样例仅仅输出data域的信息）

//目前还是不知道用这个函数有什么用，主要是不太会用函数指针

/\*template<class ElemType>

bool visit(BinaryTreeNode<ElemType>\* root) {

if (!root) return false;

else {

cout << root->data << " ";

return true;

}

}\*/

bool isOperator(const char& ch) {

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

return true;

}

else {

return false;

}

}

//运算符级别比较函数，若ope1更高级，返回1，若ope2更高级，则返回-1，若二者同级（不考虑先后顺序），则返回0

//且不考虑括号运算符

inline int operator\_cmp(const char& ope1, const char& ope2) {

if (ope1 == '+' || ope1 == '-') {

if (ope2 == '+' || ope2 == '-')return 0;

else return -1;

}

else if (ope1 == '\*' || ope1 == '/') {

if (ope2 == '+' || ope2 == '-')return 1;

else return 0;

}

}

//将中缀表达式转换成表达式二叉树 （用户函数）

BinaryTreeNode<string>\* Inffix\_BianryTree(const string& inffix) {

string::const\_iterator it;

string num = "";

stack<BinaryTreeNode<string>\* >OPND;//结点栈

stack<char>OPTR;//运算符栈

for (it = inffix.begin(); it != inffix.end(); ++it) {//这个是从前往后扫

if (\*it == ' ') continue;

else if (\*it >= '0' && \*it <= '9') {//表明it扫描到的是数

num += \*it;

if (it!=inffix.end()-1) {//表明读完了这个数

if (\*(it + 1) == ' ' || isOperator(\*(it + 1))) {

BinaryTreeNode<string>\* tmp = new BinaryTreeNode<string>;

tmp->data = num;

OPND.push(tmp);

num = "";

}

}

else if(it == inffix.end() - 1) {

BinaryTreeNode<string>\* tmp = new BinaryTreeNode<string>;

tmp->data = num;

OPND.push(tmp);

num = "";

}

}

else {//表明读到的应该是运算符

if (OPTR.empty()) {//运算符栈为空，直接压入栈

OPTR.push(\*it);

}

else {//运算符栈不为空，那么久要进行一些比较操作

char e = OPTR.top();

if (e == '(') {

OPTR.push(\*it);

}

else {//表明OPTR栈顶的不是左括号，需要进行运算符比较

if (\*it == '(') {

OPTR.push(\*it);

}

else if (\*it == ')') {//\*it为‘)’，我们需要对OPTR栈中的内容进行弹出并创建二叉树结点，直到遇到‘（’

while (e != '(') {

if(!OPTR.empty())

OPTR.pop();

if (!OPND.empty()) {

BinaryTreeNode<string>\* r = new BinaryTreeNode<string>;

BinaryTreeNode<string>\* R = OPND.top();

if(!OPND.empty())

OPND.pop();

BinaryTreeNode<string>\* L = OPND.top();

if(!OPND.empty())

OPND.pop();

r->RChild = R;

r->LChild = L;

r->data = e;

OPND.push(r);

}

if(!OPTR.empty())

e = OPTR.top();

}

if(e=='(')

OPTR.pop();//对右括号进行弹出

}

else if (operator\_cmp(\*it, e) == 1) {//表明\*it不是括号型运算符，且\*it运算级别更高，把\*itpush入OPTR栈中

OPTR.push(\*it);

}

else {//表明\*it运算符的优先级低于e

while (!OPTR.empty()&&operator\_cmp(\*it, e)!=1&&e!='(') {//如果e的优先级一直不如\*it低，那么一直就进行弹出和创建结点的操作

if (e == '-' || (e == '\*' && \*it != '\*') || e == '/') {

//只有在以上的运算符的情况下才有必要进行OPTR栈的pop操作（为了保证计算结果的正确性）

//其余的运算符情况均只需要往OPTR栈里push而无需改变顺序，并且这样也不影响结果，同时符合OJ输出规则

if (!OPTR.empty())

OPTR.pop();

if (!OPND.empty()) {

BinaryTreeNode<string>\* r = new BinaryTreeNode<string>;

BinaryTreeNode<string>\* R = OPND.top();

if (!OPND.empty())

OPND.pop();

BinaryTreeNode<string>\* L = OPND.top();

if (!OPND.empty())

OPND.pop();

r->RChild = R;

r->LChild = L;

r->data = e;

OPND.push(r);

}

if (!OPTR.empty())

e = OPTR.top();

}

else break;

}

OPTR.push(\*it);

}

}

}

}

}

while (!OPTR.empty()) {//操作符栈不空

char e = OPTR.top();

OPTR.pop();

BinaryTreeNode<string>\* r = new BinaryTreeNode<string>;

BinaryTreeNode<string>\* R = OPND.top();

OPND.pop();

BinaryTreeNode<string>\* L = OPND.top();

OPND.pop();

r->RChild = R;

r->LChild = L;

r->data = e;

OPND.push(r);

}

return OPND.top();

}

int main(){

string inffix;

getline(cin, inffix);

BinaryTree<string> result;

result.setRoot(Inffix\_BianryTree(inffix));

result.preOrderTraverse(result.getRoot());

cout << endl;

result.inOrderTraverse(result.getRoot());

cout << endl;

result.postOrderTraverse(result.getRoot());

return 0;

}

//(12 + 14) \* 3 - 400 / (30 + 10 \* 5) + 62

/\*+ - \* + 12 14 3 / 400 + 30 \* 10 5 62

12 + 14 \* 3 - 400 / 30 + 10 \* 5 + 62

12 14 + 3 \* 400 30 10 5 \* + / - 62 + \*/