package com.bayuedekui.test;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.LinkedList;

import java.util.Queue;

import java.util.Stack;

import java.awt.List;

import java.lang.Math;

## //判断数组中的内容是不是二叉搜索树的后续遍历

public boolean VerifySquenceOfBST(int [] sequence) {

if(sequence.length==0){

return false;

}

if(sequence.length==1){

return true;

}

return isBinSearchHouXu(sequence, 0, sequence.length-1);

}

public boolean isBinSearchHouXu(int[] sequence,int start,int end){

int i=end;//根节点的下标

if(start>=end){

return true;//当起始坐标大于等于末尾坐标,大于不可能,只有可能等于,表明就一个元素

}

while(i>start&&sequence[i-1]>sequence[end]){

i--;//一直找到左子树的部分

}

for(int j=start;j<i;j++){

if(sequence[j]>sequence[end]){//通过右子树找左子树,然后每一个都比根小才满足二叉搜索树条件

return false;

}

}

//递归到左右子树

return isBinSearchHouXu(sequence, start, i-1)&&isBinSearchHouXu(sequence, i, end-1);

}

//二叉树的定义

public class TreeNode {

int val = 0;

TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) {

this.val = val;

}

}

## //将二叉搜索树抓转化为双向列表

TreeNode leftHead=null;

TreeNode rightHead=null;

public TreeNode Convert(TreeNode pRootOfTree){

if(pRootOfTree==null){

return pRootOfTree;

}

Convert(pRootOfTree.left);

if(rightHead==null){//一直遍历到最后

leftHead=rightHead=pRootOfTree;

}else{

//把根节点插到链表的右边

rightHead.right=pRootOfTree;

pRootOfTree.left=rightHead;

rightHead=pRootOfTree;

}

//递归右子树

Convert(pRootOfTree.right);

return leftHead;

}

## //数组中的逆序对,我采用的方法不对,时间复杂度太高了

@Test

public int InversePairs(int [] array) {

int count=0;

for(int i=0;i<array.length;i++){

for(int j=i+1;j<array.length;j++){

if(array[i]>array[j]){

count++;

}

}

}

return count%1000000007;

}

## //两个链表的第一个公共节点

public class ListNode{

int val=0;

ListNode next=null;

public ListNode(int val){

this.val=val;

}

}

public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {

HashMap<ListNode, Integer> hashMap=new HashMap<>();

while(pHead1!=null){

hashMap.put(pHead1, 1);

pHead1=pHead1.next;

}

while(pHead2!=null){

if(hashMap.containsKey(pHead2)){

return pHead2;

}

pHead2=pHead2.next;

}

return null;

}

## //求二叉树的深度(递归法和非递归法)

//1递归法

public int TreeDepth(TreeNode root) {

if(root==null){

return 0;

}

int left=TreeDepth(root.left);

int right=TreeDepth(root.right);

return Math.max(left, right)+1;

}

//非递归(层次)

public int TreeDepth1(TreeNode root){

if(root==null){

return 0;

}

Queue<TreeNode> queue=new LinkedList<>();

queue.add(root);

int depth=0,count=0,nextcount=1;

while(queue.size()!=0){

TreeNode node=queue.poll();

count++;

if(node.left!=null){

queue.add(node.left);

}

if(node.right!=null){

queue.add(node.right);

}

if(count==nextcount){

nextcount=queue.size();//记录一轮存进去多少个节点

count=0;

depth++;

}

}

return depth;

}

//二叉树镜像(即所有的左右子树交换)

/\* public class TreeNode{

int val=0;

TreeNode left=null;

TreeNode right=null;

public TreeNode(int val){

this.val=val;

}

}\*/

public void Mirror(TreeNode root){

if(root==null){

return;

}

if(root.left==null&&root.right==null){

return;

}

TreeNode temp=root.left;

root.left=root.right;

root.right=temp;//实现左右子树的交换

if(root.left!=null){

Mirror(root.left);

}

if(root.right!=null){

Mirror(root.right);

}

}

//判断一个数是否是另一个树的子结构

/\*首先是二叉树的定义同上定义\*/

public boolean HasSubTree(TreeNode root1,TreeNode root2){

if(root2==null){

return false;//题目规定树为空时,不是子树

}

if(root1==null&&root2!=null){

return false;

}

boolean flag=false;

if(root1.val==root2.val){

flag=isSubTree(root1,root2);//首先看看整棵树是不是

}

if(!flag){

flag=isSubTree(root1.left, root2);//看root的左子树包不包含root2

if(!flag){

flag=isSubTree(root1.right, root2);

}

}

return flag;

}

public boolean isSubTree(TreeNode root1,TreeNode root2){//判断是否为子树

if(root2==null){

return true;//一直遍历root2,遍历到最后都符合,那就返回true

}

if(root1==null&&root2!=null){//当root1为空,而root2不为空的.root2肯定就不是root1的子树了

return false;

}

if(root1.val==root1.val){

return isSubTree(root1.left, root2.left)&&isSubTree(root1.right, root2.right);

}else{

return false;

}

}

## //二叉树的深度

//1递归写法

public int treeDepth(TreeNode root){

if(root==null){

return 0;

}

int left=treeDepth(root.left);

int right=treeDepth(root.right);

return Math.max(left, right)+1;

}

//非递归写法

public int treeDepth1(TreeNode root){

if(root==null){

return 0;

}

if(root.left==null&&root.right==null){

return 1;

}

Queue<TreeNode> q=new LinkedList<TreeNode>();//用队列去实现二叉树的层次遍历,顺便记录树的深度

q.add(root);

int depth=0,count=0,nextcount=1;

while(q.size()!=0){

TreeNode temp=q.poll();

count++;

if(temp.left!=null){

q.add(temp.left);

}

if(temp.right!=null){

q.add(temp.right);

}

if(count==nextcount){

nextcount=q.size();

count=0;

depth++;

}

}

return depth;

}

## //判断是否是平衡二叉树

public boolean isBalanceTree(TreeNode root) {

if(root==null){

return true;

}

if(Math.abs(getHeight(root.left)-getHeight(root.right))>1){//根不为空,最外层大子树是否平衡

return false;

}

//具体的子树情况

return isBalanceTree(root.left)&&isBalanceTree(root.right);

}

public int getHeight(TreeNode root){

if(root==null){

return 0;

}

return Math.max(getHeight(root.left),getHeight(root.right))+1;

}

//一个链表中包含环，请找出该链表的环的入口结点

public ListNode EntryNodeOfLoop(ListNode pHead){

if(pHead==null||pHead.next==null){

return null;

}

ListNode p1=pHead;

ListNode p2=pHead;

while(p2!=null&&p2.next!=null){

p1=p1.next;//p1走一步

p2=p2.next.next;//p2走两步

if(p1==p2){//第一次相遇时

p2=pHead;

while(p1!=p2){

p1=p1.next;

p2=p2.next;

}

if(p1==p2){

return p1;

}

}

}

return null;

}

## //在一个排序的链表中，存在重复的结点，请删除该链表中重复的结点，重复的结点不保留，

//返回链表头指针。 例如，链表1->2->3->3->4->4->5 处理后为 1->2->5

//1非递归

public ListNode deleteDuplication(ListNode pHead){

if(pHead==null){

return null;

}

if(pHead!=null&&pHead.next==null){

return pHead;

}

//设置一个不包含第一个节点的头结点

ListNode first=new ListNode(-1);

first.next=pHead;

ListNode p=pHead;

ListNode last=first;//用于记录新的链表

while(p!=null&&p.next!=null){

if(p.val==p.next.val){

//p=p.next;//这样只能向后挪一个

int val=p.val;

while(p!=null&&p.val==val){

p=p.next;

}

last.next=p;//这边用于将last连起来,只记录需要记录的

}else{

last=p;//不重复时,用last来记录p,主要是通过last来连接的,因为last.next=pHead

p=p.next;

}

}

return first.next;

}

//2非递归

public ListNode deleteDuplication1(ListNode pHead){

if(pHead==null||pHead.next==null){//只有零个或一个节点是,返回本身

return pHead;

}

if(pHead.val==pHead.next.val){

ListNode temp=pHead.next;//设置一个临时节点,然后对其进行操作

while(temp!=null&&temp.val==pHead.val){//跳过相同的全部节点

temp=temp.next;

}

return deleteDuplication(temp);//因为这边有temp=pHead.next,所以返回temp的处理结果就可以,

}else{//当前节点不是重复节点

pHead.next=deleteDuplication(pHead.next);

return pHead;//返回的还是pHead,pHead中有传递的消息

}

}

## //二叉树的下一个节点

(给定一个二叉树和其中的一个结点，请找出中序遍历顺序的下一个结点并且返回。注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针)

public class TreeLinkNode {

int val;

TreeLinkNode left = null;

TreeLinkNode right = null;

TreeLinkNode next = null;

TreeLinkNode(int val) {

this.val = val;

}

}

public TreeLinkNode GetNext(TreeLinkNode node){

if(node==null){

return null;

}

if(node.right!=null){//如果有右子树,那么写一个节点为右子树的最左节点

node=node.right;

while(node.left!=null){

node=node.left;

}

return node;

}else{//如果没有右子树,那么寻找本节点(是他父节点的左孩子)的父节点

while(node.next!=null){

if(node.next.left==node){

return node.next;

}

node=node.next;

}

}

return null;//全部遍历了还没找到,就为空了

}

## //判断一颗二叉树是不是对称二叉树

(二叉树与二叉树的镜像是同样的)

//我的思路:首先求这棵树的镜像,然后随便什么序遍历原来的和镜像过后的,相等就是

public TreeNode getMirror(TreeNode preNode){

TreeNode mirNode=preNode;

if(mirNode==null){

return null;

}

if(mirNode.left==null&&mirNode.right==null){

return mirNode;

}

TreeNode temp =mirNode.left;

mirNode.left=mirNode.right;

mirNode.right=temp;//进行左右子树的交换

//递归应用于左右子树

if(mirNode.left!=null){

getMirror(mirNode.left);

}

if(mirNode.right!=null){

getMirror(mirNode.right);

}

return mirNode;

}

boolean isSymmetrical(TreeNode pRoot){

TreeNode mirNode=getMirror(pRoot);

return pRoot==mirNode;//这边有疑问,应该采用什么方式比较,才能证明两个二叉树相等

}

//他人思路,直接针对本题,未采用getMirror方法

boolean isSymmetrical1(TreeNode pRoot){

if(pRoot==null){

return true;

}

return isDuiCheng(pRoot.left,pRoot.right);

}

boolean isDuiCheng(TreeNode tree1,TreeNode tree2){

if(tree1==null&&tree2==null){

return true;

}

if(tree1!=null&&tree2!=null){

return tree1.val==tree2.val&&isDuiCheng(tree1.left, tree2.right)&&isDuiCheng(tree1.right, tree2.left);

}

return false;

}

## //请实现一个函数按照之字形打印二叉树

，即第一行按照从左到右的顺序打印，第二层按照从右至左的顺序打印，第三行按照从左到右的顺序打印，其他行以此类推。

//思路:借用一个栈,奇数层右子树先进去,左子树后进去,然后存入Arraylist,偶数层相反

public ArrayList<ArrayList<Integer> > Print(TreeNode pRoot) {

ArrayList<ArrayList<Integer>> finList=new ArrayList<ArrayList<Integer>>();

if(pRoot==null){

return finList;

}

//建两个栈,用于存正序或者逆序

Stack<TreeNode> s1=new Stack<TreeNode>();//用于基数,存正序右子树先进

s1.add(pRoot);

Stack<TreeNode> s2=new Stack<TreeNode>();//用于偶数,存逆序左子树先进

while(!s1.isEmpty()||!s2.isEmpty()){//保证s1终究还是可以变成空的

if(!s1.isEmpty()){//存基数不为空时

ArrayList<Integer> finList2=new ArrayList<Integer>();

while(!s1.isEmpty()){

TreeNode temp=s1.pop();

finList2.add(temp.val);//将根节点存入,所以下一个节点肯定是偶行

if(temp.left!=null){

s2.add(temp.left);//偶行先存入左,右,出来就是优作

}

if(temp.right!=null){

s2.add(temp.right);

}

}

finList.add(finList2);

}else{

ArrayList<Integer> finList2=new ArrayList<Integer>();

while(!s2.isEmpty()){

TreeNode temp=s2.pop();

if(temp.right!=null){//奇数行先入右左,然后出左右

s1.add(temp.right);

}

if(temp.left!=null){

s1.add(temp.left);

}

finList2.add(temp.val);

}

finList.add(finList2);

}

}

return finList;

}

## //从上到下按层打印二叉树

，同一层结点从左至右输出。每一层输出一行。

//思路,类似,层次遍历,借助一个队列,可以用LinkedList,因为他实现了Deque(queue)接口,也实现list接口

public ArrayList<ArrayList<Integer>> Print1(TreeNode pRoot) {

ArrayList<ArrayList<Integer>> res=new ArrayList<ArrayList<Integer>>();

if(pRoot==null){

return res;

}

Queue<TreeNode> layer=new LinkedList<TreeNode>();

ArrayList<Integer> layerList=new ArrayList<Integer>();

layer.add(pRoot);

int start=0,end=1;

while(!layer.isEmpty()){

TreeNode temp=layer.poll();//用的是队列的方法,去除前一个并删除,也可用list的方法,删除第一个

layerList.add(temp.val);//将第一个节点加入ArrayList中

start++;

if(temp.left!=null){//左子树进入队列

layer.add(temp.left);

}

if(temp.right!=null){//右子树进入队列

layer.add(temp.right);

}

if(start==end){//已经搞完,着手下一层

end=layer.size();

start=0;

res.add(layerList);//一层节点进入最终结果arraylist

layerList.clear();//清空上一轮存入的一轮的数据,以便进行下一轮的数据进入

}

}

return res;

}

## //请实现两个函数，分别用来序列化和反序列化二叉树

public String Serialize(TreeNode root) {

StringBuffer sb1=new StringBuffer();

if(root==null){

sb1.append("#,");

return sb1.toString();

}

sb1.append(root.val+",");

sb1.append(Serialize(root.left));

sb1.append(Serialize(root.right));

return sb1.toString();

}

int index=-1;

public TreeNode Deserialize(String str) {//采用递归,所以外部要定义个index,每次递归++,那后去除字符串

index++;

int len=str.length();

if(index>=len){

return null;//str为空,返回空树

}

String[] strArr=str.split(",");

TreeNode node=null;//初始化节点,因为要反序列化,构造树

if(!strArr[index].equals("#")){

node=new TreeNode(Integer.valueOf(strArr[index]));

node.left=Deserialize(str);

node.right=Deserialize(str);//还是采用str,因为递归是有index来控制取str转化的数组下标

}

return node ;

}

## //给定一颗二叉搜索树，请找出其中的第k大的结点。

例如， 5 / \ 3 7 /\ /\ 2 4 6 8 中，按结点数值大小顺序第三个结点的值为4。

//思路:就是中序遍历,然后取第几个

int index1=0;

public TreeNode KthNode(TreeNode pRoot, int k){

if(pRoot==null){

return null;

}

if(pRoot != null){ //中序遍历寻找第k个

TreeNode node = KthNode(pRoot.left,k);

if(node != null){

return node;

}

index1 ++;

if(index == k){

return pRoot;

}

node = KthNode(pRoot.right,k);

if(node != null){

return node;

}

}

return null;

## //从尾到头打印链表

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {

Stack<Integer> stack=new Stack<Integer>();

while(listNode!=null){

stack.push(listNode.val);

listNode=listNode.next;

}

ArrayList<Integer> arrayList=new ArrayList<Integer>();

while(!stack.isEmpty()){

arrayList.add(stack.pop());

}

return arrayList;

}

## //重建二叉树(根据前序和中序)

public TreeNode reConstructBinaryTree(int [] pre,int [] in) {

if(pre.length==0||in.length==0){

return null;

}

TreeNode node=new TreeNode(pre[0]);

for(int i=0;i<in.length;i++){

if(pre[0]==in[i]){

node.left=reConstructBinaryTree(Arrays.copyOfRange(pre, 1, i+1), Arrays.copyOfRange(in, 0, i));

node.right=reConstructBinaryTree(Arrays.copyOfRange(pre, i+1, pre.length), Arrays.copyOfRange(in, i+1, in.length));

}

}

return node;

## //链表中倒数第k个节点

public ListNode FindKthToTail(ListNode head,int k) {

//采用栈,只过50

//一下采用正序

if(head==null){

return head;

}

ListNode node=head;

int count=0;

while(node!=null){

count++;//统计链表的节点数

node=node.next;

}

if(count<k) return null;

ListNode temp=head;

for(int i=0;i<count-k;i++){

temp=temp.next;

}

return temp;

}

## //反转链表

public ListNode ReverseList(ListNode head) {

//要的是反转链表,而不是用集合反转数

if(head==null) return head;

ListNode pre=null;

ListNode next=null;

while(head!=null){//1->2->3-----1<-2<-3

next=head.next;

head.next=pre;

pre=head;

head=next;

}

return pre;//pre就是每一个节点

}

## //合并两个排序链表

public ListNode Merge(ListNode list1,ListNode list2) {

if(list1==null){

return list2;

}

if(list2==null){

return list1;

}

ListNode pMergeHead=null;

if(list1.val<list2.val){

pMergeHead=list1;

pMergeHead.next=Merge(list1.next, list2);

}else{

pMergeHead=list2;

pMergeHead.next=Merge(list2.next,list1);

}

return pMergeHead;

}

## //树的子结构

public boolean HasSubtree(TreeNode root1,TreeNode root2) {

//root1先序遍历,root2先序遍历,root1包涵root2

if(root2==null){

return false;

}

if(root1==null&&root2!=null){

return false;

}

boolean flag=false;

if(root1.val==root2.val){

flag=isSubTree(root1, root2);

}

if(!flag){

flag=HasSubtree(root1.left, root2);

if(!flag){

flag=HasSubtree(root1.right, root2);

}

}

return flag;

}

public boolean isSubTree(TreeNode tree1,TreeNode tree2){//判断是否为子树

if(tree2==null){return true;}//递归一直走到最后tree2为空

if(tree1==null&&tree2!=null){return false;}

if(tree1.val==tree2.val){

return isSubTree(tree1.left,tree2.left)&&isSubTree(tree1.right, tree2.right);

}else{

return false;

}

}