# -\*- coding:utf-8 -\*-

import sys

from collections import deque

"""

二叉树的相关算法、

二叉树相关概念

完全二叉树：设二叉树的高度为h,除却第h层外，其他各层的节点数都达到最大值，第h层有叶子节点

并且叶子节点都是从左到右排布，这就是完全二叉树

满二叉树：除了叶子节点外每一个节点都有左右子树且叶子节点都处在最底层的二叉树

平衡二叉树：：是空树，或者左右两个子树的高度差绝对值不超过1，并且左右两个子树都是平衡二叉树

"""

class TreeNode(object):

def \_\_init\_\_(self,x):

self.val = x

self.left = None

self.right = None

"""

算法1：depth函数，二叉树深度

算法2：pre\_order函数，先序遍历

算法3：mid\_order函数，中序遍历

算法4：post\_order函数，后序遍历

算法5：width函数，二叉树的宽度

算法6：level函数，二叉树层次遍历

算法7：count函数，二叉树所有节点数

算法8：level\_count函数，二叉树第k层节点数：第k层节点个数等于左子树第k-1层节点数加上右子树第k-1层节点数

算法9：leaf\_count函数，二叉树叶子节点个数：等于左子树叶子节点个数加上右子树叶子节点个数

算法10：max\_count函数，二叉树的最大直径：即二叉树所有节点之间距离的最大值

有三种情况:第一种：两个节点分别在左子树和右子树中，第二种：都在左子树中，第三种：都在右子树中

算法11：node\_path函数，二叉树中某个节点到根节点的路径

算法12：public\_node函数，寻找二叉树中两个节点最近的公共父节点

算法12：two\_node\_path函数，二叉树中两个节点之间的路径

算法12：reverse\_recur函数，使用递归的方法对二叉树进行反转

算法13：reverse\_nonrecur函数，使用递归的方法对二叉树进行反转

算法14：two\_node\_path函数，node1到node2之间的路径以及距离

算法15：judge\_complete\_binary\_tree函数，判断二叉树是否是完全二叉树

"""

class binary(object):

def main(self,root):

# 二叉树深度

self.depth(root)

node\_list = []

# 先序遍历

a = self.pre\_order(root,node\_list)

print a

# 中序遍历

node\_list = []

a = self.mid\_order(root,node\_list)

print a

# 后续遍历

node\_list = []

a = self.post\_order(root,node\_list)

print a

# 二叉树的宽度，某一层节点数量最大值

a = self.width(root)

print a

# 二叉树层次遍历

node\_list = []

a =self.level(root,node\_list)

print a

# 二叉树所有节点数：左子树和右子树节点数加1

a = self.count(root)

print a

# 某一层的节点数：左子树的第k-1层节点数和右子树的第k-1层节点数

a = self.level\_count(root,3)

print a

# 叶子节点的数量：左子树的叶子节点加上右子树的叶子节点数量

a = self.leaf\_count(root)

print a

# 二叉树的最大直径：即二叉树所有节点之间距离的最大值

a = self.max\_count\_first(root)

print "first: ",a

a = self.max\_count\_second(root)

print a[1]

# 寻找某个节点是否存在二叉树中，若存在，返回路径

path = []

a = self.node\_path(root, path, 17)

print a,path

# 寻找两个节点的所有公共节点，

node1 = 8

node2 = 17

path = []

k = self.public\_node(root, node1, node2)

print k.val

# 二叉树反转，递归方法

import copy

proot = copy.copy(root)

self.reverse\_recur(proot)

a = proot

# 二叉树反转，非递归方法

proot = copy.copy(root)

self.reverse\_nonrecur(proot)

a = proot

# node1到node2之间的路径

path = self.two\_node\_path(root,node1,node2)

print path

def depth(self,root):

if root is None:

return 0

left,right = self.depth(root.left),self.depth(root.right)

return max(left,right)+1

def pre\_order(self,root, node\_list):

if root is None:

return node\_list

node\_list.append(root.val)

self.pre\_order(root.left,node\_list)

self.pre\_order(root.right,node\_list)

return node\_list

def mid\_order(self,root,node\_list):

if root is None:

return node\_list

self.mid\_order(root.left,node\_list)

node\_list.append(root.val)

self.mid\_order(root.right, node\_list)

return node\_list

def post\_order(self,root,node\_list):

if root is None:

return node\_list

self.mid\_order(root.left,node\_list)

self.mid\_order(root.right, node\_list)

node\_list.append(root.val)

return node\_list

def width(self, root):

from collections import deque

a = deque()

a.append(root)

# 层的最大宽度

width\_max = 1

while len(a)>0:

for i in range(len(a)):

node = a.popleft()

if node.left:

a.append(node.left)

if node.right:

a.append(node.right)

width\_max = max(width\_max,len(a))

return width\_max

def level(self,root,node\_list):

from collections import deque

a = deque()

a.append(root)

while len(a)>0:

node = a.popleft()

node\_list.append(node.val)

if node.left:

a.append(node.left)

if node.right:

a.append(node.right)

return node\_list

def count(self,root):

if not root:

return 0

return self.count(root.left) + self.count(root.right) + 1

def level\_count(self, root, k):

if not root or k==0:

return 0

if k==1:

return 1

return self.level\_count(root.left,k-1) + self.level\_count(root.right,k-1)

def leaf\_count(self,root):

if not root:

return 0

if not root.left and not root.right:

return 1

return self.leaf\_count(root.left) + self.leaf\_count(root.right)

def max\_count\_first(self,root):

if not root:

return 0

# 第一种情况：等于左子树深度加上右子树深度

value1 = self.depth(root.left) + self.depth(root.right)

#第二种情况

value2 = self.max\_count\_first(root.left)

# 第三种情况

value3 = self.max\_count\_first(root.right)

return max(value1,max(value2,value3))

def max\_count\_second(self,root):

"""

第二种方法，因为在计算深度和最大距离时存在重复遍历，造成效率很低，将计算深度时和最大距离放在同一遍历中

返回的数组中，第一个值为深度，第二个值为最大距离

:param root:

:return:

"""

if not root:

return [0,0]

left = self.max\_count\_second(root.left)

right = self.max\_count\_second(root.right)

a = [0,0]

a[0] = max(left[0],right[0]) + 1

a[1] = max(max(left[1],right[1]),left[0]+right[0])

return a

def node\_path(self, root, path, node):

"""

寻找节点是否在二叉树中，若是返回二叉树根节点到该节点的路径

思想：先将该节点压入堆栈中，分别寻找该节点的左子树和右子树

若在左子树或右子树中找到该节点，则返回True，并将该节点压入堆栈，

若在左子树和右子树中都没有找到，则退出堆栈中数据，并返回False

:param root:

:param path:

:param node:

:return:

"""

if not root or not node:

return False

if root.val == node:

path.append(root.val)

return True

path.append(root.val)

bool\_find1 = self.node\_path(root.left,path,node)

bool\_find2 = self.node\_path(root.right,path,node)

if not bool\_find1 and not bool\_find2:

path.pop()

return False

return True

def public\_node(self,root,node1,node2):

"""

两个节点的第一个公共节点肯定是根节点，

第一种方法：分别寻找出两个节点的路径，找出路径中最近的公共节点

第二种方法：分别在左子树中和右子树中寻找，若在左子树中和右子树中分别找到了两个节点

第二种方法：当左子树和左子树中都找到该节点，则返回True，path不做处理

当左子树和右子树中只有一个找到该节点，则将该节点肯定不是公共节点，将该节点推出堆栈，并返回True

当左子树和右子树中都没有找到该节点，则将该节点肯定不是公共节点，将该节点推出堆栈，并返回False

第二种想法错误：只考虑到两个节点再根节点的不同子树中，未考虑到其他情况

第三种方法：如果该节点为空，则返回空None,如果该节点等于其中一个节点，则返回该节点

对左子树和右子树的查找结果而言，如果两者都不为空，则该节点即为父节点

如果根节点的左子树不为空，则该根节点的左节点即为两者的公共节点，则返回该左节点，一直返回该左节点

如果根节点的右子树不为空，理由同上

第四种方法：使用LCA算法Tarjan求解（并查集+深度搜索）

:param root:

:param node1:

:param node2:

:return:

"""

# 第二种方法

# if not root:

# return False

# if root.val==node1:

# return True

# if root.val==node2:

# return True

# path.append(root.val)

# find1 = self.public\_node(root.left,node1,node2,path)

# find2 = self.public\_node(root.right,node1,node2,path)

# if not find1 and not find2:

# path.pop()

# return False

# elif find1 and find2:

# return True

# else:

# path.pop()

# return True

# 第三种方法

if root==None or root.val==node1 or root.val==node2:

return root

left = self.public\_node(root.left, node1, node2)

right = self.public\_node(root.right,node1,node2)

if left and right:

return root

return left if left else right

def reverse\_recur(self,root):

"""

使用递归的方法对链表进行反转

:param root:

:return:

"""

if not root:

return root

if root.left or root.right:

root.left, root.right = root.right, root.left

self.reverse\_recur(root.left)

self.reverse\_recur(root.right)

def reverse\_nonrecur(self,root):

a = deque()

a.append(root)

while len(a)>0:

node = a.popleft()

if node.left or node.right:

node.left,node.right = node.right,node.left

if node.left:

a.append(node.left)

if node.right:

a.append(node.right)

def two\_node\_path(self,root,node1,node2):

"""

查找二叉树中两个节点之间的路径

:param root:

:param node1:

:param node2:

:return:

"""

# 查找两个节点的最近公共节点

common\_node = self.public\_node(root,node1,node2)

path1 = []

self.node\_path(common\_node, path1, node1)

path2 = []

self.node\_path(common\_node, path2, node2)

path1.reverse()

path = path1[:-1] + path2

return path

def judge\_complete\_binary\_tree(self,root):

"""

:param root:

:return:

"""

a = deque()

a.append(root)

flag = False

while len(a)>0:

node = a.pop()

if flag is False:

if node.right:

if node.left:

a.append(node.left)

a.append(node.right)

else:

return False

else:

flag = True

else:

if node.left or node.right:

return False

return True

def judge\_full\_binary\_tree(self,root):

if not root:

return False

depth = self.depth(root)

leaf\_node\_count = self.leaf\_count(root)

if leaf\_node\_count==pow(2,depth-1):

return True

else:

return False

a1 = TreeNode(1)

a2 = TreeNode(2)

a3 = TreeNode(3)

a4 = TreeNode(7)

a5 = TreeNode(8)

a6 = TreeNode(9)

a7 = TreeNode(10)

a8 = TreeNode(13)

a9 = TreeNode(17)

a1.left = a2

a1.right = a3

a2.left = a4

a2.right = a5

a3.left = a6

a3.right = a7

a6.left = a8

a7.right = a9

test =binary().main(a1)