

# Assignment #5: "树"算：概念、表示、解析、遍历

---

Updated 2124 GMT+8 March 17, 2024

2024 spring, Compiled by 天幕 化学与分子工程学院

## 说明：

1) The complete process to learn DSA from scratch can be broken into 4 parts:

Learn about Time complexities, learn the basics of individual Data Structures, learn the basics of Algorithms, and practice Problems.

2) 请把每个题目解题思路（可选），源码Python, 或者C++（已经在Codeforces/Openjudge上AC），截图（包含Accepted），填写到下面作业模版中（推荐使用 typora <https://typoraio.cn>，或者用 word）。AC 或者没有AC，都请标上每个题目大致花费时间。

3) 提交时候先提交pdf文件，再把md或者doc文件上传到右侧“作业评论”。Canvas需要有同学清晰头像、提交文件有pdf、“作业评论”区有上传的md或者doc附件。

4) 如果不能在截止前提交作业，请写明原因。

## 编程环境

操作系统：Windows 11 23H2

Python编程环境：Visual Studio Code 1.86.2230.

## 1. 题目

---

### 27638: 求二叉树的高度和叶子数目

<http://cs101.openjudge.cn/practice/27638/>

思路：使用init()完成树的构建，一次性得到深度和叶数。感觉有点臃肿。

代码

```
1 class treeNode(object):
2     leftSubNodeID:int
3     rightSubNodeID:int
4     nodeID:int
5     level:int = 0
6     isLeafNode:bool = False
7     def __init__(self, id, left, right):
```

```

8         self.leftSubNodeID = left
9         self.rightSubNodeID = right
10        self.nodeID = id
11        if left == -1 == right:
12            self.isLeafNode = True
13
14        def getLeftSubNodeID(self):
15            return self.leftSubNodeID
16        def getRightSubNodeID(self):
17            return self.rightSubNodeID
18        def getLevel(self):
19            return self.level
20        def setLevel(self, int):
21            self.level = int
22
23    class biTree(object):
24        nodeDic:dict
25        depth:int = 0
26        leaf:int = 0
27        def __init__(self):
28            self.nodeDic = dict()
29        def add(self, key, node):
30            self.nodeDic[key] = node
31        def get(self, key):
32            if key == -1:
33                return False
34            else:
35                return self.nodeDic[key]
36        def init(self):
37            for treeNodeID, treeNode in self.nodeDic.items():
38                if not treeNode.isLeafNode:
39                    self.depth = max(self.depth, self.getNodeLevel(treeNodeID))
40                else:
41                    self.leaf += 1
42        def getNodeLevel(self, nodeID):
43            currentNode = self.get(nodeID)
44            if currentNode:
45                if currentNode.isLeafNode:
46                    return 0
47                else:
48                    if currentNode.level == 0:
49                        currentNode.setLevel(1 +
max(self.getNodeLevel(currentNode.getLeftSubNodeID()),
self.getNodeLevel(currentNode.getRightSubNodeID())))
50                    return currentNode.level
51            return -1
52
53    n = int(input())
54    myBiTree = biTree()
55    for i in range(n):
56        l, r = map(int, input().split())
57        myBiTree.add(i, treeNode(i, l, r))
58    myBiTree.init()
59    print(myBiTree.depth, myBiTree.leaf)

```

代码运行截图



状态: Accepted

源代码

```
class treeNode(object):
    leftSubNodeID:int
    rightSubNodeID:int
    nodeID:int
    level:int = 0
    isLeafNode:bool = False
    def __init__(self, id, left, right):
        self.leftSubNodeID = left
        self.rightSubNodeID = right
        self.nodeID = id
        if left == -1 == right:
            self.isLeafNode = True

    def getLeftSubNodeID(self):
        return self.leftSubNodeID
    def getRightSubNodeID(self):
        return self.rightSubNodeID
    def getLevel(self):
        return self.level
    def setLevel(self, int):
        self.level = int

class biTree(object):
    nodeDic:dict
    depth:int = 0
    leaf:int = 0
    def __init__(self):
        self.nodeDic = dict()
    def add(self, key, node):
        self.nodeDic[key] = node
    def get(self, key):
        if key == -1:
            return False
        else:
            return self.nodeDic[key]
    def init(self):
        for treeNodeID, treeNode in self.nodeDic.items():
            if not treeNode.isLeafNode:
                self.depth = max(self.depth, self.getNodeLevel(treeNodeID))
            else:
                self.leaf += 1
    def getNodeLevel(self, nodeID):
        currentNode = self.get(nodeID)
        if currentNode:
            if currentNode.isLeafNode:
                return 0
            else:
                if currentNode.level == 0:
                    currentNode.setLevel(1 + max(self.getNodeLevel(currentNode.leftSubNodeID), self.getNodeLevel(currentNode.rightSubNodeID)))
                return currentNode.level
        return -1

n = int(input())
myBiTree = biTree()
for i in range(n):
```

```

    l, r = map(int, input().split())
    myBiTree.add(i, treeNode(i, l, r))
myBiTree.init()
print(myBiTree.depth, myBiTree.leaf)

```

## 24729: 括号嵌套树

<http://cs101.openjudge.cn/practice/24729/>

思路：种树用栈，输出用递归。唉，数算。

思路和之前的栈或树相比其实差不多，就是写起来比较费时间。

代码

```

1  from collections import deque
2
3
4  class Node(object):
5      NodeID:str
6      pNodeID:str
7      sub:list
8      def __init__(self, NodeID, l, pNodeID= ''):
9          self.NodeID = NodeID
10         self.pNodeID = pNodeID
11         self.sub = []
12         for node in l:
13             self.sub.append(node)
14         def __str__(self): #Debug
15             return "🌿 {} p: {}, s: {}".format(self.NodeID, self.pNodeID, self.sub)
16         def info(self):
17             return (self.NodeID, self.sub)
18
19  class Tree(object):
20      tree:dict
21      root:Node
22      def __init__(self):
23          self.tree = dict()
24          self.root = None
25
26      def add(self, node:Node):
27          cNodeID, cSubNodes = node.info()
28
29          self.tree[cNodeID] = node    #加入树
30
31          if not self.root:    #尝试转移根节点
32              self.root = node
33          elif self.root.NodeID in cSubNodes:
34              self.root = node

```

```

35
36         for nodes in self.tree.values(): #尝试添加父节点
37             aNodeID, aSubNodes = nodes.info()
38             if cNodeID in aSubNodes: #是子节点
39                 node.pNodeID = aNodeID
40     def get(self, nodeID):
41         if nodeID == "":
42             return False
43         else:
44             return self.tree[nodeID]
45     def getDepth(self, node:Node):
46         cSubNodes = node.sub
47         if cSubNodes:
48             return(1 + max([self.getDepth(self.get(subNode)) for subNode in
cSubNodes]))
49         return 1
50     def getTreeDep(self):
51         return self.getDepth(self.root)
52
53     def preorderFrom(self, node:Node): # 先序遍历
54         return node.NodeID + "".join([self.preorderFrom(self.get(x)) for x
in node.sub])
55     def preorder(self):
56         return self.preorderFrom(self.root)
57     def postorderFrom(self, node:Node): # 后序遍历
58         return "".join([self.postorderFrom(self.get(x)) for x in node.sub])
+ node.NodeID
59     def postorder(self):
60         return self.postorderFrom(self.root)
61
62
63
64     def getTreeFromString(stree:str):
65         outTree = Tree()
66         if len(stree) == 1:
67             node = Node(stree, [])
68             outTree.add(node)
69             outTree.root = node
70             return outTree
71         stack = []
72         for char in stree:
73             if char == ')':
74                 ichar = stack[-1]
75                 l = []
76                 while ichar != '(':
77                     l.append(ichar)
78                     stack.pop()
79                     ichar = stack[-1]
80                 stack.pop()
81                 for x in l:
82                     if not x in outTree.tree.keys():
83                         outTree.tree[x] = Node(x, [])
84                 node = Node(stack[-1], reversed(l))
85                 outTree.add(node)
86             elif char == ',':
87                 continue

```

```
88         else:
89             stack.append(char)
90         return outTree
91
92
93 myTree = getTreeFromString(input())
94 print(myTree.preorder() + "\n" + myTree.postorder())
```

代码运行截图

## 状态: Accepted

### 源代码

```
from collections import deque

class Node(object):
    NodeID:str
    pNodeID:str
    sub:list
    def __init__(self, NodeID, l, pNodeID= ''):
        self.NodeID = NodeID
        self.pNodeID = pNodeID
        self.sub = []
        for node in l:
            self.sub.append(node)
    def __str__(self): #Debug
        return "🌿 {} p: {}, s: {}".format(self.NodeID, self.pNodeID, self.sub)
    def info(self):
        return (self.NodeID, self.sub)

class Tree(object):
    tree:dict
    root:Node
    def __init__(self):
        self.tree = dict()
        self.root = None

    def add(self, node:Node):
        cNodeID, cSubNodes = node.info()

        self.tree[cNodeID] = node #加入树

        if not self.root: #尝试转移根节点
            self.root = node
        elif self.root.NodeID in cSubNodes:
            self.root = node

        for nodes in self.tree.values(): #尝试添加父节点
            aNodeID, aSubNodes = nodes.info()
            if cNodeID in aSubNodes: #是子节点
                node.pNodeID = aNodeID
    def get(self, nodeID):
        if nodeID == "":
            return False
        else:
            return self.tree[nodeID]
    def getDepth(self, node:Node):
        cSubNodes = node.sub
        if cSubNodes:
            return 1 + max([self.getDepth(self.get(subNode)) for subNode in cSubNodes])
        return 1
    def getTreeDep(self):
        return self.getDepth(self.root)

    def preorderFrom(self, node:Node): # 先序遍历
        return node.NodeID + " ".join([self.preorderFrom(self.get(x)) for x in node.sub])
    def preorder(self):
        return self.preorderFrom(self.root)
```



```

        return self.preorderFrom(self.root)
    def postorderFrom(self, node:Node): # 后序遍历
        return "".join([self.postorderFrom(self.get(x)) for x in node.subtree])
    def postorder(self):
        return self.postorderFrom(self.root)

def getTreeFromString(stree:str):
    outTree = Tree()
    if len(stree) == 1:
        node = Node(stree, [])
        outTree.add(node)
        outTree.root = node
        return outTree
    stack = []
    for char in stree:
        if char == ')':
            ichar = stack[-1]
            l = []
            while ichar != '(':
                l.append(ichar)
                stack.pop()
                ichar = stack[-1]
            stack.pop()
            for x in l:
                if not x in outTree.tree.keys():
                    outTree.tree[x] = Node(x, [])
            node = Node(stack[-1], reversed(l))
            outTree.add(node)
        elif char == ',':
            continue
        else:
            stack.append(char)
    return outTree

myTree = getTreeFromString(input())
print(myTree.preorder() + "\n" + myTree.postorder())

```

©2002-2022 POJ 京ICP备20010980号-1

## 02775: 文件结构“图”

<http://cs101.openjudge.cn/practice/02775/>

思路：写了半天发现名字可以重复……看来确实要好好审题。递归+树+栈，还是一个套路。

代码

```

1 indent = "|      "
2 enter = "\n"
3
4 def getPos(node):
5     name = node.name
6     if name.startswith('d'):
7         return str(node.NodeID)
8     else:
9         return name
10
11 class Node(object):
12     _ID = 0
13     NodeID:int
14     pNodeID:int
15     name:str
16     sub:list    #List<Node>
17     depth:int
18     def __init__(self, name, l, pNodeID= -1, depth:int=0):
19         self.NodeID = self._ID
20         self.__class__._ID += 1
21         self.pNodeID = pNodeID
22         self.sub = []
23         self.name = name
24         self.depth = depth
25         for node in l:
26             self.sub.append(node)
27     def info(self):
28         return (self.NodeID, self.sub)
29     def toString(self, depth:int=-1):
30         if self.name.startswith('d') or self.name == "ROOT":
31             self.sub.sort(key=lambda x: getPos(x))
32             return (depth + 1) * indent + self.name + enter +
33             "".join([x.toString(depth + 1) for x in self.sub])
34         else:
35             return depth * indent + self.name + enter
36
37 class Tree(object):
38     tree:dict
39     root:Node
40     def __init__(self):
41         self.tree = dict()
42         self.root = None
43
44     def add(self, node:Node):
45         cNodeID, cSubNodes = node.info()
46
47         self.tree[cNodeID] = node    #加入树
48
49         if not self.root:    #尝试转移根节点
50             self.root = node
51         elif self.get(self.root.NodeID) in cSubNodes:
52             self.root = node
53
54         for nodes in self.tree.values():    #尝试添加父节点
55             aNodeID, aSubNodes = nodes.info()
56             if self.get(cNodeID) in aSubNodes:    #是子节点

```

```

56         node.pNodeID = aNodeID
57     def get(self, nodeID):
58         if nodeID == -1:
59             return False
60         else:
61             return self.tree[nodeID]
62     def getDepth(self, node:Node):
63         cSubNodes = node.sub
64         if cSubNodes:
65             if node.depth == 0:
66                 self.depth = 1 + max([self.getDepth(subNode) for subNode in
cSubNodes])
67             return node.depth
68         return 0
69     def getTreeDep(self): # This can also init the tree
70         return self.getDepth(self.root)
71
72     def toString(self):
73         return self.root.toString()
74
75 def printTree(i):
76     myTree = Tree()
77     stack = [Node("ROOT", []), '[']
78     file:str = input()
79     if file == '#':
80         raise EOFError
81     while True:
82         if file == ']' or file == '*':
83             ifile = stack[-1]
84             l = []
85             while ifile != '[':
86                 l.append(ifile)
87                 stack.pop()
88                 ifile = stack[-1]
89             stack.pop()
90             node = stack[-1]
91             for x in reversed(l):
92                 try:
93                     node.sub.append(x)
94                 except AttributeError:
95                     pass # WTF how does this even work
96             myTree.add(node)
97             if file == '*':
98                 break
99             elif file.startswith('d'):
100                 stack.append(Node(file, []))
101                 stack.append('[')
102             elif file.startswith('f'):
103                 stack.append(Node(file, []))
104             file:str = input()
105     print("DATA SET {}: ".format(i) + enter + myTree.toString())
106
107 i = 1
108 while True:
109     try:
110         printTree(i)

```

```
111     except EOFError:
112         break
113     i += 1
```

代码运行截图

## 状态: Accepted

### 源代码

```
indent = "  "
enter = "\n"

def getPos(node):
    name = node.name
    if name.startswith('d'):
        return str(node.NodeID)
    else:
        return name

class Node(object):
    _ID = 0
    NodeID:int
    pNodeID:int
    name:str
    sub:list      #List<Node>
    depth:int
    def __init__(self, name, l, pNodeID= -1, depth:int=0):
        self.NodeID = self._ID
        self.__class__._ID += 1
        self.pNodeID = pNodeID
        self.sub = []
        self.name = name
        self.depth = depth
        for node in l:
            self.sub.append(node)
    def info(self):
        return (self.NodeID, self.sub)
    def toString(self, depth:int=-1):
        if self.name.startswith('d') or self.name == "ROOT":
            self.sub.sort(key=lambda x: getPos(x))
            return (depth + 1) * indent + self.name + enter + "".join([x
        else:
            return depth * indent + self.name + enter

class Tree(object):
    tree:dict
    root:Node
    def __init__(self):
        self.tree = dict()
        self.root = None

    def add(self, node:Node):
        cNodeID, cSubNodes = node.info()

        self.tree[cNodeID] = node      #加入树

        if not self.root:      #尝试转移根节点
            self.root = node
        elif self.get(self.root.NodeID) in cSubNodes:
            self.root = node

        for nodes in self.tree.values():      #尝试添加父节点
            aNodeID, aSubNodes = nodes.info()
            if self.get(cNodeID) in aSubNodes:      #是子节点
                node.pNodeID = aNodeID
```

```

def get(self, nodeID):
    if nodeID == -1:
        return False
    else:
        return self.tree[nodeID]
def getDepth(self, node:Node):
    cSubNodes = node.sub
    if cSubNodes:
        if node.depth == 0:
            self.depth = 1 + max([self.getDepth(subNode) for subNode
            return node.depth
        return 0
def getTreeDep(self): # This can also init the tree
    return self.getDepth(self.root)

def toString(self):
    return self.root.toString()

def printTree(i):
    myTree = Tree()
    stack = [Node("ROOT", []), '[']
    file:str = input()
    if file == '#':
        raise EOFError
    while True:
        if file == ']' or file == '*':
            ifile = stack[-1]
            l = []
            while ifile != '[':
                l.append(ifile)
                stack.pop()
                ifile = stack[-1]
            stack.pop()
            node = stack[-1]
            for x in reversed(l):
                try:
                    node.sub.append(x)
                except AttributeError:
                    if node == '[':
                        raise IndexError
                    pass # WTF how does this even work
            myTree.add(node)
            if file == '*':
                break
            elif file.startswith('d'):
                stack.append(Node(file, []))
                stack.append('[')
            elif file.startswith('f'):
                stack.append(Node(file, []))
            file:str = input()
    print("DATA SET {}: ".format(i) + enter + myTree.toString())

i = 1
while True:
    try:
        printTree(i)
    except EOFError:
        break
    i += 1

```

## 25140: 根据后序表达式建立队列表达式

<http://cs101.openjudge.cn/practice/25140/>

思路：和上面基本没变化的栈+树。代码复用率极高。

代码

```
1 class Node(object):
2     _ID = 0
3     NodeID:int
4     pNodeID:int
5     name:str
6     sub:list    #List<Node>
7     depth:int
8     def __init__(self, name, l, pNodeID= -1, depth:int=0):
9         self.NodeID = self._ID
10        self.__class__._ID += 1
11        self.pNodeID = pNodeID
12        self.sub = []
13        self.name = name
14        self.depth = depth
15        for node in l:
16            self.sub.append(node)
17    def info(self):
18        return (self.NodeID, self.sub)
19
20 class Tree(object):
21     tree:dict
22     root:Node
23     def __init__(self):
24         self.tree = dict()
25         self.root = None
26
27     def add(self, node:Node):
28         cNodeID, cSubNodes = node.info()
29
30         self.tree[cNodeID] = node    #加入树
31
32         if not self.root:    #尝试转移根节点
33             self.root = node
34         elif self.get(self.root.NodeID) in cSubNodes:
35             self.root = node
36
37         for nodes in self.tree.values():    #尝试添加父节点
38             aNodeID, aSubNodes = nodes.info()
39             if self.get(cNodeID) in aSubNodes:    #是子节点
40                 node.pNodeID = aNodeID
41     def get(self, nodeID):
```

```

42         if nodeID == -1:
43             return False
44         else:
45             return self.tree[nodeID]
46     def getDepth(self, node:Node):
47         cSubNodes = node.sub
48         if cSubNodes:
49             if node.depth == 0:
50                 self.depth = 1 + max([self.getDepth(subNode) for subNode in
cSubNodes])
51             return node.depth
52         return 0
53     def getTreeDep(self): # This can also init the tree
54         return self.getDepth(self.root)
55
56     def levelOrderFrom(self, node:Node):
57         if not node: return []
58
59         res, queue = [], [node]
60         while queue:
61             level_node = []
62
63             for _ in range(len(queue)):
64                 node = queue.pop(0)
65                 level_node.append(node.name)
66
67                 for x in node.sub:
68                     if x:
69                         queue.append(x)
70             res.append(level_node)
71
72         return "".join(["".join(x) for x in res])
73
74     def levelOrder(self):
75         return self.levelOrderFrom(self.root)
76
77     def toTree(stree:str):
78         myTree = Tree()
79         stack = []
80         for char in stree:
81             if char.isupper():
82                 node = Node(char, stack[-2:])
83                 myTree.add(node)
84                 stack.pop()
85                 stack.pop()
86                 stack.append(node)
87             else:
88                 stack.append(Node(char, []))
89         return(myTree)
90
91     for _ in range(int(input())):
92         print(toTree(input()).levelOrder()[::-1])

```



## 状态: Accepted

### 源代码

```
class Node(object):
    _ID = 0
    NodeID:int
    pNodeID:int
    name:str
    sub:list      #List<Node>
    depth:int
    def __init__(self, name, l, pNodeID= -1, depth:int=0):
        self.NodeID = self._ID
        self.__class__._ID += 1
        self.pNodeID = pNodeID
        self.sub = []
        self.name = name
        self.depth = depth
        for node in l:
            self.sub.append(node)
    def info(self):
        return (self.NodeID, self.sub)

class Tree(object):
    tree:dict
    root:Node
    def __init__(self):
        self.tree = dict()
        self.root = None

    def add(self, node:Node):
        cNodeID, cSubNodes = node.info()

        self.tree[cNodeID] = node      #加入树

        if not self.root:      #尝试转移根节点
            self.root = node
        elif self.get(self.root.NodeID) in cSubNodes:
            self.root = node

        for nodes in self.tree.values():      #尝试添加父节点
            aNodeID, aSubNodes = nodes.info()
            if self.get(cNodeID) in aSubNodes:      #是子节点
                node.pNodeID = aNodeID
    def get(self, nodeID):
        if nodeID == -1:
            return False
        else:
            return self.tree[nodeID]
    def getDepth(self, node:Node):
        cSubNodes = node.sub
        if cSubNodes:
            if node.depth == 0:
                self.depth = 1 + max([self.getDepth(subNode) for subNode in cSubNodes])
            return node.depth
        return 0
    def getTreeDep(self):      # This can also init the tree
        return self.getDepth(self.root)

    def levelOrderFrom(self, node:Node):
```

```

        if not node: return []

    res, queue = [], [node]
    while queue:
        level_node = []

        for _ in range(len(queue)):
            node = queue.pop(0)
            level_node.append(node.name)

            for x in node.sub:
                if x:
                    queue.append(x)
        res.append(level_node)

    return "".join(["".join(x) for x in res])

def levelOrder(self):
    return self.levelOrderFrom(self.root)

def toTree(stree:str):
    myTree = Tree()
    stack = []
    for char in stree:
        if char.isupper():
            node = Node(char, stack[-2:])
            myTree.add(node)
            stack.pop()
            stack.pop()
            stack.append(node)
        else:
            stack.append(Node(char, []))
    return myTree

for _ in range(int(input())):
    print(toTree(input()).levelOrder()[::-1])

```

©2002-2022 POJ 京ICP备20010980号-1

## 24750: 根据二叉树中后序序列建树

<http://cs101.openjudge.cn/practice/24750/>

思路：同样通过递归把问题分解，感觉递归最重要的是终止条件。

代码

```

1 class Node(object):
2     _ID = 0
3     NodeID:int
4     pNodeID:int
5     name:str

```

```

6     sub:list      #List<Node>
7     depth:int
8     def __init__(self, name, l, pNodeID= -1, depth:int=0):
9         self.NodeID = self._ID
10        self.__class__._ID += 1
11        self.pNodeID = pNodeID
12        self.sub = []
13        self.name = name
14        self.depth = depth
15        for node in l:
16            self.sub.append(node)
17    def info(self):
18        return (self.NodeID, self.sub)
19
20    class Tree(object):
21        tree:dict
22        root:Node
23        def __init__(self):
24            self.tree = dict()
25            self.root = None
26
27        def add(self, node:Node):
28            cNodeID, cSubNodes = node.info()
29
30            self.tree[cNodeID] = node    #加入树
31
32            if not self.root:    #尝试转移根节点
33                self.root = node
34            elif self.get(self.root.NodeID) in cSubNodes:
35                self.root = node
36
37            for nodes in self.tree.values():    #尝试添加父节点
38                aNodeID, aSubNodes = nodes.info()
39                if self.get(cNodeID) in aSubNodes:    #是子节点
40                    node.pNodeID = aNodeID
41        def get(self, nodeID):
42            if nodeID == -1:
43                return False
44            else:
45                return self.tree[nodeID]
46        def getDepth(self, node:Node):
47            cSubNodes = node.sub
48            if cSubNodes:
49                if node.depth == 0:
50                    self.depth = 1 + max([self.getDepth(subNode) for subNode in
cSubNodes])
51                return node.depth
52            return 0
53        def getTreeDep(self):    # This can also init the tree
54            return self.getDepth(self.root)
55
56        def levelOrderFrom(self, node:Node):
57            if not node: return []
58
59            res, queue = [], [node]
60            while queue:

```

```

61         level_node = []
62
63         for _ in range(len(queue)):
64             node = queue.pop(0)
65             level_node.append(node.name)
66
67             for x in node.sub:
68                 if x:
69                     queue.append(x)
70         res.append(level_node)
71
72         return "".join(["".join(x) for x in res])
73
74     def levelOrder(self):
75         return self.levelOrderFrom(self.root)
76     def preOrderFrom(self, node:Node): # 先序遍历
77         if not node: return ''
78         return node.name + "".join([self.preOrderFrom(x) for x in
node.sub])
79     def preOrder(self):
80         return self.preOrderFrom(self.root)
81     def postOrderFrom(self, node:Node): # 后序遍历
82         if not node: return ''
83         return "".join([self.postOrderFrom(x) for x in node.sub]) +
node.name
84     def postOrder(self):
85         return self.postOrderFrom(self.root)
86
87     def toTree(middle: str, preOrPost: str, index: int) -> Tree:
88         def toNode(tree: Tree, middle: str, preOrPost: str, index: int):
89             try:
90                 rootName = preOrPost[index]
91                 rootIndex = middle.find(rootName)
92                 info = middle[:rootIndex], middle[rootIndex + 1:],
preOrPost[:rootIndex], preOrPost[rootIndex:-1]
93             except IndexError:
94                 return False
95             if info == ('', '', '', ''):
96                 node = Node(rootName, [])
97                 tree.add(node)
98                 return(node)
99             lSubTreeMiddle, rSubTreeMiddle, lSubTreePreOrPost,
rSubTreePreOrPost = info
100             node = Node(rootName, [toNode(tree, lSubTreeMiddle,
lSubTreePreOrPost, index), toNode(tree, rSubTreeMiddle, rSubTreePreOrPost,
index)])
101             tree.add(node)
102             return(node)
103         myTree = Tree()
104         toNode(myTree, middle, preOrPost, index)
105         return myTree
106
107     print(toTree(input(), input(), -1).preOrder())

```

代码运行截图



## 状态: Accepted

### 源代码

```
class Node(object):
    _ID = 0
    NodeID:int
    pNodeID:int
    name:str
    sub:list    #List<Node>
    depth:int
    def __init__(self, name, l, pNodeID= -1, depth:int=0):
        self.NodeID = self._ID
        self.__class__._ID += 1
        self.pNodeID = pNodeID
        self.sub = []
        self.name = name
        self.depth = depth
        for node in l:
            self.sub.append(node)
    def info(self):
        return (self.NodeID, self.sub)

class Tree(object):
    tree:dict
    root:Node
    def __init__(self):
        self.tree = dict()
        self.root = None

    def add(self, node:Node):
        cNodeID, cSubNodes = node.info()

        self.tree[cNodeID] = node    #加入树

        if not self.root:    #尝试转移根节点
            self.root = node
        elif self.get(self.root.NodeID) in cSubNodes:
            self.root = node

        for nodes in self.tree.values():    #尝试添加父节点
            aNodeID, aSubNodes = nodes.info()
            if self.get(cNodeID) in aSubNodes:    #是子节点
                node.pNodeID = aNodeID
    def get(self, nodeID):
        if nodeID == -1:
            return False
        else:
            return self.tree[nodeID]
    def getDepth(self, node:Node):
        cSubNodes = node.sub
        if cSubNodes:
            if node.depth == 0:
                self.depth = 1 + max([self.getDepth(subNode) for subNode in cSubNodes])
            return node.depth
        return 0
    def getTreeDep(self):    # This can also init the tree
        return self.getDepth(self.root)

    def levelOrderFrom(self, node:Node):
        if not node:
            return []
        queue = [node]
        result = []
        while queue:
            node = queue.pop(0)
            result.append(node.name)
            if node.sub:
                queue.extend(node.sub)
```

```

    if not node: return []

    res, queue = [], [node]
    while queue:
        level_node = []

        for _ in range(len(queue)):
            node = queue.pop(0)
            level_node.append(node.name)

            for x in node.sub:
                if x:
                    queue.append(x)
        res.append(level_node)

    return "".join(["".join(x) for x in res])

def levelOrder(self):
    return self.levelOrderFrom(self.root)
def preOrderFrom(self, node:Node): # 先序遍历
    if not node: return ''
    return node.name + "".join([self.preOrderFrom(x) for x in node.sub])
def preOrder(self):
    return self.preOrderFrom(self.root)
def postOrderFrom(self, node:Node): # 后序遍历
    if not node: return ''
    return "".join([self.postOrderFrom(x) for x in node.sub]) + node.name
def postOrder(self):
    return self.postOrderFrom(self.root)

def toTree(middle: str, preOrPost: str, index: int) -> Tree:
    def toNode(tree: Tree, middle: str, preOrPost: str, index: int):
        try:
            rootName = preOrPost[index]
            rootIndex = middle.find(rootName)
            info = middle[:rootIndex], middle[rootIndex + 1:], preOrPost[index + 1:]
        except IndexError:
            return False
        if info == ('', '', '', ''):
            node = Node(rootName, [])
            tree.add(node)
            return (node)
        lSubTreeMiddle, rSubTreeMiddle, lSubTreePreOrPost, rSubTreePreOrPost = info
        node = Node(rootName, [toNode(tree, lSubTreeMiddle, lSubTreePreOrPost, 0),
                               toNode(tree, rSubTreeMiddle, rSubTreePreOrPost, 0)])
        tree.add(node)
        return (node)
    myTree = Tree()
    toNode(myTree, middle, preOrPost, index)
    return myTree

print(toTree(input(), input(), -1).preOrder())

```

## 22158: 根据二叉树前中序序列建树

<http://cs101.openjudge.cn/practice/22158/>

思路：和上题基本一致的代码。本来可以做到完全一致 但是懒得改了。

代码

```
1 class Node(object):
2     _ID = 0
3     NodeID:int
4     pNodeID:int
5     name:str
6     sub:list    #List<Node>
7     depth:int
8     def __init__(self, name, l, pNodeID= -1, depth:int=0):
9         self.NodeID = self._ID
10        self.__class__._ID += 1
11        self.pNodeID = pNodeID
12        self.sub = []
13        self.name = name
14        self.depth = depth
15        for node in l:
16            self.sub.append(node)
17    def info(self):
18        return (self.NodeID, self.sub)
19
20 class Tree(object):
21     tree:dict
22     root:Node
23     def __init__(self):
24         self.tree = dict()
25         self.root = None
26
27     def add(self, node:Node):
28         cNodeID, cSubNodes = node.info()
29
30         self.tree[cNodeID] = node    #加入树
31
32         if not self.root:    #尝试转移根节点
33             self.root = node
34         elif self.get(self.root.NodeID) in cSubNodes:
35             self.root = node
36
37         for nodes in self.tree.values():    #尝试添加父节点
38             aNodeID, aSubNodes = nodes.info()
39             if self.get(cNodeID) in aSubNodes:    #是子节点
40                 node.pNodeID = aNodeID
41     def get(self, nodeID):
42         if nodeID == -1:
43             return False
44         else:
```



```

45         return self.tree[nodeID]
46     def getDepth(self, node:Node):
47         cSubNodes = node.sub
48         if cSubNodes:
49             if node.depth == 0:
50                 self.depth = 1 + max([self.getDepth(subNode) for subNode in
cSubNodes])
51             return node.depth
52         return 0
53     def getTreeDep(self): # This can also init the tree
54         return self.getDepth(self.root)
55
56     def levelOrderFrom(self, node:Node):
57         if not node: return []
58
59         res, queue = [], [node]
60         while queue:
61             level_node = []
62
63             for _ in range(len(queue)):
64                 node = queue.pop(0)
65                 level_node.append(node.name)
66
67                 for x in node.sub:
68                     if x:
69                         queue.append(x)
70             res.append(level_node)
71
72         return "".join(["".join(x) for x in res])
73
74     def levelOrder(self):
75         return self.levelOrderFrom(self.root)
76     def preOrderFrom(self, node:Node): # 先序遍历
77         if not node: return ''
78         return node.name + "".join([self.preOrderFrom(x) for x in
node.sub])
79     def preOrder(self):
80         return self.preOrderFrom(self.root)
81     def postOrderFrom(self, node:Node): # 后序遍历
82         if not node: return ''
83         return "".join([self.postOrderFrom(x) for x in node.sub]) +
node.name
84     def postOrder(self):
85         return self.postOrderFrom(self.root)
86
87     def toTree(preOrPost: str, middle: str, index: int) -> Tree:
88         def toNode(tree: Tree, middle: str, preOrPost: str, index: int):
89             try:
90                 rootName = preOrPost[index]
91                 rootIndex = middle.find(rootName)
92                 info = middle[:rootIndex], middle[rootIndex + 1:],
preOrPost[1:rootIndex + 1], preOrPost[rootIndex + 1:]
93             except IndexError:
94                 return False
95             if info == ('', '', '', ''):
96                 node = Node(rootName, [])

```

```

97         tree.add(node)
98         return(node)
99         lSubTreeMiddle, rSubTreeMiddle, lSubTreePreOrPost,
rSubTreePreOrPost = info
100         node = Node(rootName, [toNode(tree, lSubTreeMiddle,
lSubTreePreOrPost, index), toNode(tree, rSubTreeMiddle, rSubTreePreOrPost,
index)])
101         tree.add(node)
102         return(node)
103     myTree = Tree()
104     toNode(myTree, middle, preOrPost, index)
105     return myTree
106
107 while True:
108     try:
109         print(toTree(input(), input(), 0).postOrder())
110     except EOFError:
111         break
```

代码运行截图

## 状态: Accepted

源代码

```
class Node(object):
    _ID = 0
    NodeID:int
    pNodeID:int
    name:str
    sub:list      #List<Node>
    depth:int
    def __init__(self, name, l, pNodeID= -1, depth:int=0):
        self.NodeID = self._ID
        self.__class__._ID += 1
        self.pNodeID = pNodeID
        self.sub = []
        self.name = name
        self.depth = depth
        for node in l:
            self.sub.append(node)
    def info(self):
        return (self.NodeID, self.sub)

class Tree(object):
    tree:dict
    root:Node
    def __init__(self):
        self.tree = dict()
        self.root = None

    def add(self, node:Node):
        cNodeID, cSubNodes = node.info()

        self.tree[cNodeID] = node      #加入树

        if not self.root:      #尝试转移根节点
            self.root = node
        elif self.get(self.root.NodeID) in cSubNodes:
            self.root = node

        for nodes in self.tree.values():      #尝试添加父节点
            aNodeID, aSubNodes = nodes.info()
            if self.get(cNodeID) in aSubNodes:      #是子节点
                node.pNodeID = aNodeID
    def get(self, nodeID):
        if nodeID == -1:
            return False
        else:
            return self.tree[nodeID]
    def getDepth(self, node:Node):
        cSubNodes = node.sub
        if cSubNodes:
            if node.depth == 0:
                self.depth = 1 + max([self.getDepth(subNode) for subNode in cSubNodes])
            return node.depth
        return 0
    def getTreeDep(self):      # This can also init the tree
        return self.getDepth(self.root)

    def levelOrderFrom(self, node:Node):
```

```

    if not node: return []

    res, queue = [], [node]
    while queue:
        level_node = []

        for _ in range(len(queue)):
            node = queue.pop(0)
            level_node.append(node.name)

            for x in node.sub:
                if x:
                    queue.append(x)
        res.append(level_node)

    return "".join(["".join(x) for x in res])

def levelOrder(self):
    return self.levelOrderFrom(self.root)
def preOrderFrom(self, node:Node): # 先序遍历
    if not node: return ''
    return node.name + "".join([self.preOrderFrom(x) for x in node.sub])
def preOrder(self):
    return self.preOrderFrom(self.root)
def postOrderFrom(self, node:Node): # 后序遍历
    if not node: return ''
    return "".join([self.postOrderFrom(x) for x in node.sub]) + node.name
def postOrder(self):
    return self.postOrderFrom(self.root)

def toTree(preOrPost: str, middle: str, index: int) -> Tree:
def toNode(tree: Tree, middle: str, preOrPost: str, index: int):
    try:
        rootName = preOrPost[index]
        rootIndex = middle.find(rootName)
        info = middle[:rootIndex], middle[rootIndex + 1:], preOrPost[index + 1:]
    except IndexError:
        return False
    if info == ('', '', '', ''):
        node = Node(rootName, [])
        tree.add(node)
        return (node)
    lSubTreeMiddle, rSubTreeMiddle, lSubTreePreOrPost, rSubTreePreOrPost = info
    node = Node(rootName, [toNode(tree, lSubTreeMiddle, lSubTreePreOrPost, 0),
                           toNode(tree, rSubTreeMiddle, rSubTreePreOrPost, 0)])
    tree.add(node)
    return (node)
myTree = Tree()
toNode(myTree, middle, preOrPost, index)
return myTree

while True:
    try:
        print(toTree(input(), input(), 0).postOrder())
    except EOFError:
        break

```

## 2. 学习总结和收获

---

这周的题花了很多时间精力在做，主要是学习一些新的概念以及与各种递归越界或者溢出斗智斗勇，当然还有一些奇怪的bug。感觉写树和递归越来越熟练了。