Leetcode(python版) 树中等题

94二叉树的中序遍历（做过）

95不同的二叉搜索树II

题目：

给定一个整数 *n*，生成所有由 1 ... *n* 为节点所组成的**二叉搜索树**。

**示例:**

**输入:** 3

**输出:**

[

  [1,null,3,2],

  [3,2,null,1],

  [3,1,null,null,2],

  [2,1,3],

  [1,null,2,null,3]

]

**解释:**

以上的输出对应以下 5 种不同结构的二叉搜索树：

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

思路：对于求数量的题目一般考虑DP，而枚举所有符合条件的情况，一般用DFS。每次选取其中一个为根结点，然后求左右子树。当输入n时，二叉查找树由【1,2，···i,···n】构成，根据二叉查找树的构成性质：当选择i为根结点时，【1,2，···i-1】构成其左子树，【i+1,i+2,···，n】构成其右子树。b,e为开始和结束数字，当b > e 时，返回空(注意不是null)；

当b <= e时，用leftTree 和rightTree来分别接收左右子树。

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** generateTrees(self, n):
10. """
11. :type n: int
12. :rtype: List[TreeNode]
13. """
14. **if** n==0:
15. **return** []
16. **return** self.dfs(1,n)
17. **def** dfs(self,b,e):
18. **if** b>e:
19. **return** [None]
20. res=[]
21. **for** rootval **in** range(b,e+1):
22. leftTree=self.dfs(b,rootval-1)
23. rightTree=self.dfs(rootval+1,e)
24. **for** i **in** leftTree:
25. **for** j **in** rightTree:
26. root=TreeNode(rootval)
27. root.left=i
28. root.right=j
29. res.append(root)
30. **return** res

96不同的二叉搜索树

题目：

给定一个整数 *n*，求以 1 ... *n* 为节点组成的二叉搜索树有多少种？

**示例:**

**输入:** 3

**输出:** 5

**解释:**

给定 *n* = 3, 一共有 5 种不同结构的二叉搜索树:

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

思路：

代码：

1. **class** Solution(object):
2. **def** numTrees(self, n):
3. """
4. :type n: int
5. :rtype: int
6. """
7. num = [[0] \* (n + 1) **for** i **in** range(n + 1)]
8. num[0][0] = 1
9. **for** i **in** range(1, n + 1):
10. **for** j **in** range(1, n + 1):
11. num[i][j] = sum(num[i - j]) \* sum(num[j - 1])
12. **return** sum(num[n])

98验证二叉搜索树

题目：给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

* 节点的左子树只包含**小于**当前节点的数。
* 节点的右子树只包含**大于**当前节点的数。
* 所有左子树和右子树自身必须也是二叉搜索树。

**示例 1:**

**输入:**

2

/ \

1 3

**输出:** true

**示例 2:**

**输入:**

5

/ \

1 4

  / \

  3 6

**输出:** false

**解释:** 输入为: [5,1,4,null,null,3,6]。

  根节点的值为 5 ，但是其右子节点值为 4 。

思路一：可以通过递归来看左右子树是否符合要求

思路二：将二叉树中序遍历，如果不是递增的序列，则是假的

代码二：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** isValidBST(self, root):
10. """
11. :type root: TreeNode
12. :rtype: bool
13. """
14. **def** inorderTraversal(root):
15. **if** root==None:
16. **return** []
17. res=[]
18. res+=inorderTraversal(root.left)
19. res.append(root.val)
20. res+=inorderTraversal(root.right)
21. **return** res
22. res=inorderTraversal(root)
23. **if** res!=sorted(list(set(res))):
24. **return** False
25. **return** True

102二叉树的层次遍历

题目：给定一个二叉树，返回其按层次遍历的节点值。 （即逐层地，从左到右访问所有节点）。

例如:  
给定二叉树: [3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

返回其层次遍历结果：

[

[3],

[9,20],

[15,7]

]

思路：使用队列，BFS即可

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** levelOrder(self, root):
10. """
11. :type root: TreeNode
12. :rtype: List[List[int]]
13. """
14. **if** root==None:
15. **return** []
16. queue=[root]
17. res=[]
18. **while** queue:
19. templist=[]
20. l=len(queue)
21. **for** i **in** range(l):
22. temp=queue.pop(0)
23. templist.append(temp.val)
24. **if** temp.left:
25. queue.append(temp.left)
26. **if** temp.right:
27. queue.append(temp.right)
28. res.append(templist)
29. **return** res

103二叉树的锯齿形层次遍历（做过）

105从前序与中序遍历序列构造二叉树

题目：

根据一棵树的前序遍历与中序遍历构造二叉树。

**注意:**  
你可以假设树中没有重复的元素。

例如，给出

前序遍历 preorder = [3,9,20,15,7]

中序遍历 inorder = [9,3,15,20,7]

返回如下的二叉树：

3

/ \

9 20

/ \

15 7

思路：由前序可以得到根节点，然后根据根节点在中序遍历中的位置得到左右子树，依次递归操作即可

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** buildTree(self, preorder, inorder):
10. """
11. :type preorder: List[int]
12. :type inorder: List[int]
13. :rtype: TreeNode
14. """
15. **def** dfs(pbegin,pend,ibegin,iend):
16. **if** pbegin >= pend:
17. **return** None
18. **if** pbegin + 1 ==  pend:
19. **return** TreeNode(preorder[pbegin])
20. i = inorder.index(preorder[pbegin])
21. i -= ibegin
22. ans = TreeNode(preorder[pbegin])
23. ans.left = dfs(pbegin + 1,pbegin + i + 1,ibegin,ibegin+i)
24. ans.right = dfs(pbegin + i + 1,pend,ibegin + 1 + i,iend)
25. **return** ans
26. **return** dfs(0,len(preorder),0,len(inorder))

106从中序与后序遍历序列构造二叉树

题目：

根据一棵树的中序遍历与后序遍历构造二叉树。

**注意:**  
你可以假设树中没有重复的元素。

例如，给出

中序遍历 inorder = [9,3,15,20,7]

后序遍历 postorder = [9,15,7,20,3]

返回如下的二叉树：

3

/ \

9 20

/ \

15 7

思路：同105

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** buildTree(self, inorder, postorder):
10. """
11. :type inorder: List[int]
12. :type postorder: List[int]
13. :rtype: TreeNode
14. """
15. **if** **not** postorder:
16. **return** None
17. root = TreeNode(postorder[-1])
18. n = inorder.index(root.val)
19. root.left = self.buildTree(inorder[:n],postorder[:n])
20. root.right = self.buildTree(inorder[n+1:],postorder[n:-1])
21. **return** root

113路径综合II

题目：

给定一个二叉树和一个目标和，找到所有从根节点到叶子节点路径总和等于给定目标和的路径。

**说明:** 叶子节点是指没有子节点的节点。

**示例:**  
给定如下二叉树，以及目标和 sum = 22，

**5**

/ \

**4** **8**

/ / \

**11** 13 **4**

/ \ / \

7 **2** **5** 1

返回:

[

[5,4,11,2],

[5,8,4,5]

]

思路：最好是使用dfs

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** pathSum(self, root, sum):
10. """
11. :type root: TreeNode
12. :type sum: int
13. :rtype: List[List[int]]
14. """
15. **if** root== None:
16. **return** []
17. ans=[]
18. **if** root.left==None **and** root.right==None:
19. **if** root.val==sum:
20. **return** [[sum]]
21. **else**:
22. **return** []
23. l,r=self.pathSum(root.left,sum-root.val),self.pathSum(root.right,sum-root.val)
24. **for** i **in** l:
25. ans.append([root.val]+i)
26. **for** i **in** r:
27. ans.append([root.val]+i)
28. **return** ans

114二叉树展开为链表

题目：

给定一个二叉树，[原地](https://baike.baidu.com/item/%E5%8E%9F%E5%9C%B0%E7%AE%97%E6%B3%95/8010757" \t "_blank)将它展开为链表。

例如，给定二叉树

1

/ \

2 5

/ \ \

3 4 6

将其展开为：

1

\

2

\

3

\

4

\

5

\

6

思路一：使用递归，先连接左子树再连右子树

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** flatten(self, root):
10. """
11. :type root: TreeNode
12. :rtype: void Do not return anything, modify root in-place instead.
13. """
14. **if** root == None:
15. **return**
16. **if** root.left == None **and** root.right == None:
17. **return**
18. self.flatten(root.left);self.flatten(root.right)
19. tmp = root.right
20. root.right = root.left
21. root.left = None
22. **while** root.right:
23. root = root.right
24. root.right = tmp

思路二：用列表保存该二叉树的前序遍历的结点，最后循环列表重新赋左右子树

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** flatten(self, root):
10. """
11. :type root: TreeNode
12. :rtype: void Do not return anything, modify root in-place instead.
13. """
14. ans = []
15. **def** preOrderTravel(root):
16. **if** **not** root:
17. **return**
18. ans.append(root)
19. preOrderTravel(root.left)
20. preOrderTravel(root.right)
21. preOrderTravel(root)
22. copy = root
23. **for** i **in** range(1, len(ans)):
24. copy.left = None
25. copy.right = ans[i]
26. copy = copy.right

ans[i]存放的就是结点

116填充同一层的兄弟节点

题目：

给定一个二叉树

struct TreeLinkNode {

TreeLinkNode \*left;

TreeLinkNode \*right;

TreeLinkNode \*next;

}

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 NULL。

初始状态下，所有 next 指针都被设置为 NULL。

**说明:**

* 你只能使用额外常数空间。
* 使用递归解题也符合要求，本题中递归程序占用的栈空间不算做额外的空间复杂度。
* 你可以假设它是一个完美二叉树（即所有叶子节点都在同一层，每个父节点都有两个子节点）。

思路：借鉴层次遍历的思想，单独获得每一层节点的list然后对每一个list的next进行赋值。   
空间复杂度O(k),k表示最后一层的叶节点数量，时间复杂度O（n）

代码：

1. # Definition for binary tree with next pointer.
2. # class TreeLinkNode:
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
7. #         self.next = None
9. **class** Solution:
10. # @param root, a tree link node
11. # @return nothing
12. **def** connect(self, root):
13. **if** root!=None:
14. temp\_list=[root]
15. new\_list=[root]
16. **while** True:
17. new\_list=[]
18. **while** temp\_list!=[]:
19. temp\_node=temp\_list.pop(0)
20. new\_list.append(temp\_node.left)
21. new\_list.append(temp\_node.right)
22. **if** new\_list[0]==None:
23. **break**
24. **for** i **in** range(len(new\_list)-1):
25. new\_list[i].next=new\_list[i+1]
26. temp\_list=new\_list

思路二：直接在树节点上面进行修改，两层循环嵌套：第一层是以层次为一层，每次结束后，令node表示为node.left进入下一层，第二层循环是层次内部的节点的遍历，令node表示为node.next进入下一个节点。

代码：

1. # Definition for binary tree with next pointer.
2. # class TreeLinkNode:
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
7. #         self.next = None
9. **class** Solution:
10. # @param root, a tree link node
11. # @return nothing
12. **def** connect(self, root):
13. **while** root:
14. temp=root
15. **while** root **and** root.left:
16. root.left.next=root.right
17. **if** root.next:
18. root.right.next=root.next.left
19. root=root.next

117填充同一层的兄弟节点II

题目：

给定一个二叉树

struct TreeLinkNode {

TreeLinkNode \*left;

TreeLinkNode \*right;

TreeLinkNode \*next;

}

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 NULL。

初始状态下，所有 next 指针都被设置为 NULL。

**说明:**

* 你只能使用额外常数空间。
* 使用递归解题也符合要求，本题中递归程序占用的栈空间不算做额外的空间复杂度。

思路：用两个指针来记录下一个指针和下一层第一个指针。

代码：

1. # Definition for binary tree with next pointer.
2. # class TreeLinkNode:
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
7. #         self.next = None
9. **class** Solution:
10. # @param root, a tree link node
11. # @return nothing
12. **def** connect(self, root):
13. **if** root:
14. tmp,tmp1,tmp2 = root,None,None
15. **while** tmp:
16. **if** tmp.left:
17. **if** tmp1:
18. tmp1.next = tmp.left
19. tmp1 = tmp.left
20. **if** **not** tmp2:
21. tmp2 = tmp1
22. **if** tmp.right:
23. **if** tmp1:
24. tmp1.next = tmp.right
25. tmp1 = tmp.right
26. **if** **not** tmp2:
27. tmp2 = tmp1
28. tmp = tmp.next
29. self.connect(tmp2)

129求跟到叶子节点数字之和

题目：

给定一个二叉树，它的每个结点都存放一个 0-9 的数字，每条从根到叶子节点的路径都代表一个数字。

例如，从根到叶子节点路径 1->2->3 代表数字 123。

计算从根到叶子节点生成的所有数字之和。

**说明:** 叶子节点是指没有子节点的节点。

**示例 1:**

**输入:** [1,2,3]

1

/ \

2 3

**输出:** 25

**解释:**

从根到叶子节点路径 1->2 代表数字 12.

从根到叶子节点路径 1->3 代表数字 13.

因此，数字总和 = 12 + 13 = 25.

**示例 2:**

**输入:** [4,9,0,5,1]

4

/ \

9 0

 / \

5 1

**输出:** 1026

**解释:**

从根到叶子节点路径 4->9->5 代表数字 495.

从根到叶子节点路径 4->9->1 代表数字 491.

从根到叶子节点路径 4->0 代表数字 40.

因此，数字总和 = 495 + 491 + 40 = 1026.

思路：这题主要是怎么计算叶子的数值。leaf = node.sum \* 10 + leaf.val。记录所有叶子的数值，然后相加即可

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** sumNumbers(self, root):
10. """
11. :type root: TreeNode
12. :rtype: int
13. """
14. ans=[]
15. **def** solve(sum,root):
16. **if** root:
17. sum+=root.val
18. **if** root.right==None **and** root.left==None:
19. ans.append(sum)
20. **if** root.right:
21. solve(sum\*10,root.right)
22. **if** root.left:
23. solve(sum\*10,root.left)
24. solve(0,root)
25. res=0
26. **for** i **in** ans:
27. res+=i
28. **return** res

144二叉树的前序遍历（做过）

173二叉搜索树迭代器（做过）

199二叉树的右视图

题目：Given a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

For example:

Given the following binary tree,

1 <---

/ \

2 3 <---

\ \

5 4 <---

You should return [1, 3, 4].

思路：层次遍历打印每一层的最后一个结点的值就可以，

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. def rightSideView(self, root):
10. """
11. :type root: TreeNode
12. :rtype: List[**int**]
13. """
14. res = []
15. self.levelOrder(root, 0, res)
16. **return** [level[-1] **for** level in res]
18. def levelOrder(self, root, level, res):
19. **if** not root: **return**
20. **if** len(res) == level: res.append([])
21. res[level].append(root.val)
22. **if** root.left: self.levelOrder(root.left, level + 1, res)
23. **if** root.right: self.levelOrder(root.right, level + 1, res)

思路二：非递归解法，使用队列。这个解题的技巧在于，queue其实也可以用[-1]直接找到这个层的最后一个元素。每次进行while循环，都是开始了新的一层，for循环的巧妙在于，直接遍历队列中已有的元素，也就是上层的元素。这样的话就直接把上层的遍历完了，新的层也加入了队列

代码二：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** rightSideView(self, root):
10. """
11. :type root: TreeNode
12. :rtype: List[int]
13. """
14. res = []
15. **if** **not** root: **return** res
16. queue = collections.deque()
17. queue.append(root)
18. **while** queue:
19. res.append(queue[-1].val)
20. **for** i **in** range(len(queue)):
21. node = queue.popleft()
22. **if** node.left:
23. queue.append(node.left)
24. **if** node.right:
25. queue.append(node.right)
26. **return** res

222Count Complete Tree Nodes

题目：

Given a **complete** binary tree, count the number of nodes.

**Note:**

Definition of a complete binary tree from [Wikipedia](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees):  
In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes inclusive at the last level h.

**Example:**

**Input:**

1

/ \

2 3

/ \ /

4 5 6

**Output:** 6

思路：

我知道这个肯定有简单算法的，否则可以直接遍历去求，那样没意义。

求完全二叉树的节点数目。注意完全二叉树和满二叉树Full Binary Tree的唯一区别是，完全二叉树最后一层的节点不满，而且假设最后一层有节点，都是从左边开始。 这样我们可以利用这个性质得到下面的结论：

假如左子树高度等于右子树高度，则右子树为完全二叉树，左子树为满二叉树。

假如高度不等，则左字数为完全二叉树，右子树为满二叉树。

求高度的时候只往左子树来找。

可以使用上面的结论得出本题的解法：

先构造一个getHeight方法， 用来求出二叉树的高度。这里我们只用求从根节点到最左端节点的长度。

求出根节点左子树高度leftHeight和根节点右子树高度rightHeight

假如两者相等，那么说明左子树是满二叉树，而右子树可能是完全二叉树。

– 我们可以返回 2 ^ leftHeight - 1 + 1 + countNodes(root.right)

– 这里+1是因为把根节点也算进去，化简一下就是 1 << leftHeight + countNodes(root.right)，返回结果

否则两者不等，说明左子树是完全二叉树，右子树是满二叉树

– 我们可以返回 2^ rightHeight - 1 + 1 + countNodeS(root.left)

– 化简以后得到 1 << rightHeight + countNodes(root.left)，返回结果

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** countNodes(self, root):
10. """
11. :type root: TreeNode
12. :rtype: int
13. """
14. **if** **not** root:
15. **return** 0
16. nodes=0
17. left\_height=self.getHeight(root.left)
18. right\_height=self.getHeight(root.right)
19. **if** left\_height==right\_height:
20. nodes=2\*\*left\_height+self.countNodes(root.right)
21. **else**:
22. nodes=2\*\*right\_height+self.countNodes(root.left)
23. **return** nodes
24. **def** getHeight(self,root):
25. height=0
26. **while** root:
27. height+=1
28. root=root.left
29. **return** height

230Kth Smallest Element in a BST

题目：

Given a binary search tree, write a function kthSmallest to find the **k**th smallest element in it.

**Note:**   
You may assume k is always valid, 1 ≤ k ≤ BST's total elements.

**Example 1:**

**Input:** root = [3,1,4,null,2], k = 1

3

/ \

1 4

\

  2

**Output:** 1

**Example 2:**

**Input:** root = [5,3,6,2,4,null,null,1], k = 3

5

/ \

3 6

/ \

2 4

/

1

**Output:** 3

**Follow up:**  
What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

思路：对二叉树进行中序遍历，遍历一个节点就进行计数，计数达到k的时候就结束

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** kthSmallest(self, root, k):
10. """
11. :type root: TreeNode
12. :type k: int
13. :rtype: int
14. """
15. self.count=k
16. self.res=0
17. **def** core(root):
18. **if** root:
19. core(root.left)
20. self.count=self.count-1
21. **if** self.count==0:
22. self.res=root.val
23. core(root.right)
24. core(root)
25. **return** self.res

236 Lowest Common Ancestor of a Binary Tree

题目：

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

Given the following binary tree:  root = [3,5,1,6,2,0,8,null,null,7,4]

\_\_\_\_\_\_\_3\_\_\_\_\_\_

/ \

\_\_\_5\_\_ \_\_\_1\_\_

/ \ / \

6 \_2 0 8

/ \

7 4

**Example 1:**

**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

**Output:** 3

**Explanation:** The LCA of nodes 5 and 1 is 3.

**Example 2:**

**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

**Output:** 5

**Explanation:** The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself

according to the LCA definition.

**Note:**

* All of the nodes' values will be unique.
* p and q are different and both values will exist in the binary tree.

思路：递归搜索左右子树，如果左子树和右子树都不为空，说明最近父节点一定在根节点。

反之，如果左子树为空，说明两个节点一定在右子树；

同理如果右子树为空，说明两个节点一定在左子树

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** lowestCommonAncestor(self, root, p, q):
10. """
11. :type root: TreeNode
12. :type p: TreeNode
13. :type q: TreeNode
14. :rtype: TreeNode
15. """
16. **if** root==None:
17. **return** root
18. **if** root==p **or** root==q:
19. **return** root
20. left\_node=self.lowestCommonAncestor(root.left,p,q)
21. right\_node=self.lowestCommonAncestor(root.right,p,q)
23. **if** left\_node!=None **and** right\_node !=None:
24. **return** root
25. **if** left\_node==None:
26. **return** right\_node
27. **if** right\_node==None:
28. **return** left\_node

337打家劫舍III

题目：

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。 除了“根”之外，每栋房子有且只有一个“父“房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。 如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

**示例 1:**

**输入:** [3,2,3,null,3,null,1]

**3**

/ \

2 3

\ \

**3** **1**

**输出:** 7

**解释:** 小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = **7**.

**示例 2:**

**输入:** [3,4,5,1,3,null,1]

  3

/ \

**4** **5**

/ \ \

1 3 1

**输出:** 9

**解释:** 小偷一晚能够盗取的最高金额 = **4** + **5** = **9**.

思路：本题的做法，就是求本节点+孙子更深节点vs儿子节点+重孙更深的节点的比较。

道理能想明白，代码有点难写。用了dfs函数，虽然递归是自顶向下的，但是因为是不断的return，所以真正求值是从底向上的。用到了一个有两个元素的列表，分别保存了之前层的，不取节点和取节点的情况。然后遍历左右子树，求出当前节点取和不取能得到的值，再返回给上一层。注意这个里面的robcurr是当前节点能达到的最大值，所以最后返回结果的时候试试返回的root节点robcurr的值

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** rob(self, root):
10. """
11. :type root: TreeNode
12. :rtype: int
13. """
14. **def** dfs(root):
15. # from bottom to top
16. **if** **not** root: **return** [0, 0] # before layer, no robcurr, robcurr
17. robleft = dfs(root.left)
18. robright = dfs(root.right)
19. norobcurr = robleft[1] + robright[1]
20. robcurr = max(root.val + robleft[0] + robright[0], norobcurr)
21. **return** [norobcurr, robcurr]
22. **return** dfs(root)[1]

449序列化和反序列化二叉搜索树

题目：

序列化是将数据结构或对象转换为一系列位的过程，以便它可以存储在文件或内存缓冲区中，或通过网络连接链路传输，以便稍后在同一个或另一个计算机环境中重建。

设计一个算法来序列化和反序列化**二叉搜索树**。 对序列化/反序列化算法的工作方式没有限制。 您只需确保二叉搜索树可以序列化为字符串，并且可以将该字符串反序列化为最初的二叉搜索树。

**编码的字符串应尽可能紧凑。**

**注意**：不要使用类成员/全局/静态变量来存储状态。 你的序列化和反序列化算法应该是无状态的。

思路：看到BST，就一定想到了一个性质： BST的中序遍历是有序的。但我们又知道，只知道树的一种遍历方式，是没法确定这个树的，BST也不例外。

因此，这个题采用前序遍历的方式，这样，遍历得到的第一个数组就是BST的根节点，数组后面的这些数中比根节点的值小的是根节点的左子树，比根节点值大的是根节点的右子树（BST的最重要性质）。

因此，重要结论：BST的前序遍历能唯一的确定一颗BST

解编码过程是通过一个队列进行操作。其实也可以是list，不过队列的效率更高

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Codec:
10. **def** serialize(self, root):
11. """Encodes a tree to a single string.
13. :type root: TreeNode
14. :rtype: str
15. """
16. vals=[]
17. **def** preOrder(root):
18. **if** root:
19. vals.append(root.val)
20. preOrder(root.left)
21. preOrder(root.right)
22. preOrder(root)
23. **return** ' '.join(map(str,vals))
25. **def** deserialize(self, data):
26. """Decodes your encoded data to tree.
28. :type data: str
29. :rtype: TreeNode
30. """
31. vals=collections.deque(int(val) **for** val **in** data.split())
32. **def** build(minVal,maxVal):
33. **if** vals **and** minVal<vals[0]<maxVal:
34. val=vals.popleft()
35. root=TreeNode(val)
36. root.left=build(minVal,val)
37. root.right=build(val,maxVal)
38. **return** root
39. **return** build(float('-inf'),float('inf'))

42. # Your Codec object will be instantiated and called as such:
43. # codec = Codec()
44. # codec.deserialize(codec.serialize(root))

450删除二叉搜索树中的节点

题目：

给定一个二叉搜索树的根节点 **root** 和一个值 **key**，删除二叉搜索树中的 **key**对应的节点，并保证二叉搜索树的性质不变。返回二叉搜索树（有可能被更新）的根节点的引用。

一般来说，删除节点可分为两个步骤：

1. 首先找到需要删除的节点；
2. 如果找到了，删除它。

**说明：** 要求算法时间复杂度为 O(h)，h 为树的高度。

**示例:**

root = [5,3,6,2,4,null,7]

key = 3

5

/ \

3 6

/ \ \

2 4 7

给定需要删除的节点值是 3，所以我们首先找到 3 这个节点，然后删除它。

一个正确的答案是 [5,4,6,2,null,null,7], 如下图所示。

5

/ \

4 6

/ \

2 7

另一个正确答案是 [5,2,6,null,4,null,7]。

5

/ \

2 6

\ \

4 7

思路：这个题的解法并不是固定的，删除之后的二叉树也不止一种。比如可以有下面两种主要的方法：

节点没有左子树：返回其右子树

节点没有右子树：返回其左子树

既有左子树，又有右子树：

1）查找到其右子树的最小值的节点，替换掉被删除的节点，并删除找到的最小节点

2）查找到其左子树的最大值的节点，替换掉被删除的节点，并删除找到的最大节点

我用的是查找右子树的最小值节点的方法，最小节点就是左子树的最靠左边的节点。代码使用的递归，最核心的是找到该节点之后的操作，特别是把值进行交换一步很重要，因为我们并没有删除了该最小值节点，所以把最小值的节点赋值成要查找的节点，然后在之后的操作中将会把它删除

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** deleteNode(self, root, key):
10. """
11. :type root: TreeNode
12. :type key: int
13. :rtype: TreeNode
14. """
15. **if** **not** root: **return** None
16. **if** root.val == key:
17. **if** **not** root.right:
18. left = root.left
19. **return** left
20. **else**:
21. right = root.right
22. **while** right.left:
23. right = right.left
24. root.val, right.val = right.val, root.val
25. root.left = self.deleteNode(root.left, key)
26. root.right = self.deleteNode(root.right, key)
27. **return** root

508出现次数最多的子树元素和

题目：给出二叉树的根，找出出现次数最多的子树元素和。一个结点的子树元素和定义为以该结点为根的二叉树上所有结点的元素之和（包括结点本身）。然后求出出现次数最多的子树元素和。如果有多个元素出现的次数相同，返回所有出现次数最多的元素（不限顺序）。

**示例 1**  
输入:

5

/ \

2 -3

返回 [2, -3, 4]，所有的值均只出现一次，以任意顺序返回所有值。

**示例 2**  
输入:

5

/ \

2 -5

返回 [2]，只有 2 出现两次，-5 只出现 1 次。

**提示：** 假设任意子树元素和均可以用 32 位有符号整数表示。

思路：题目中的数据结构是二叉树，那么自然想到用递归。题目要求输出子树元素和出现次数最多的元素和，那么一个简单的想法就是递归的找寻每一个节点的子树元素和，并用一个哈希表存下每一个元素和的出现次数，然后在排序输出次数最大的元素和即可，注意可能有多个子树元素和出现次数相同。

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** findFrequentTreeSum(self, root):
10. """
11. :type root: TreeNode
12. :rtype: List[int]
13. """
14. **if** **not** root:
15. **return** []
16. **if** root.left **is** None **and** root.right **is** None:
17. **return** [root.val]
18. temp\_dict=dict()
19. **def** recur\_find\_sum(root):
20. **if** **not** root:
21. **return** 0
22. temp=root.val
23. temp+=recur\_find\_sum(root.left)
24. temp+=recur\_find\_sum(root.right)
25. **if** **not** temp\_dict.get(temp):
26. temp\_dict[temp]=1
27. **else**:
28. temp\_dict[temp]+=1
29. **return** temp
30. recur\_find\_sum(root)
31. tt=sorted(temp\_dict.iteritems(),key=**lambda** x:x[1],reverse=True)
32. **return** [tt[i][0] **for** i **in** xrange(len(tt)) **if** tt[i][1]==tt[0][1]]

513找树左下角的值

题目：

给定一个二叉树，在树的最后一行找到最左边的值。

**示例 1:**

**输入:**

2

/ \

1 3

**输出:**

1

**示例 2:**

**输入:**

1

/ \

2 3

/ / \

4 5 6

/

7

**输出:**

7

思路：

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** findBottomLeftValue(self, root):
10. """
11. :type root: TreeNode
12. :rtype: int
13. """
14. queue = [root]
15. **for** node **in** queue:
16. queue += filter(None, (node.right, node.left))
18. **return** node.val

515在每个树行中找最大值

题目：

您需要在二叉树的每一行中找到最大的值。

**示例：**

**输入:**

1

/ \

3 2

/ \ \

5 3 9

**输出:** [1, 3, 9]

思路：使用队列和一个max函数

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** largestValues(self, root):
10. """
11. :type root: TreeNode
12. :rtype: List[int]
13. """
14. ans=[]
15. **if** root ==None:
16. **return** ans
17. node=[root]
18. **while** node:
19. ans.append(max(i.val **for** i **in** node))
20. curr=[]
21. **for** each **in** node:
22. **if** each.left:
23. curr.append(each.left)
24. **if** each.right:
25. curr.append(each.right)
26. node=curr
27. **return** ans

623在二叉树中增加一行

题目：

给定一个二叉树，根节点为第1层，深度为 1。在其第 d 层追加一行值为 v 的节点。

添加规则：给定一个深度值 d （正整数），针对深度为 d-1 层的每一**非空**节点 N，为 N 创建两个值为 v 的左子树和右子树。

将 N 原先的左子树，连接为新节点 v 的左子树；将 N 原先的右子树，连接为新节点 v 的右子树。

如果 d 的值为 1，深度 d - 1 不存在，则创建一个新的根节点 v，原先的整棵树将作为 v 的左子树。

**示例 1:**

**输入:**

二叉树如下所示:

4

/ \

2 6

/ \ /

3 1 5

**v = 1**

**d = 2**

**输出:**

4

/ \

1 1

/ \

2 6

/ \ /

3 1 5

**示例 2:**

**输入:**

二叉树如下所示:

4

/

2

/ \

3 1

**v = 1**

**d = 3**

**输出:**

4

/

2

/ \

1 1

/ \

3 1

**注意:**

1. 输入的深度值 d 的范围是：[1，二叉树最大深度 + 1]。
2. 输入的二叉树至少有一个节点。

思路：采用的是递归的解法。其实仔细读题是有很大的作用的，题目中说了很清楚了，如果d是1的话，新建节点，原来的root设成该节点的左边节点。如果d是2的话，就给root节点新建出左右节点，把原来的root子节点设置成新子节点的子节点即可～

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** addOneRow(self, root, v, d):
10. """
11. :type root: TreeNode
12. :type v: int
13. :type d: int
14. :rtype: TreeNode
15. """
16. **if** **not** root:
17. **return** root
18. **if** d==1:
19. left=TreeNode(v)
20. left.left=root
21. root=left
22. **elif** d==2:
23. left=TreeNode(v)
24. right=TreeNode(v)
25. left.left=root.left
26. right.right=root.right
27. root.left=left
28. root.right=right
29. **else**:
30. self.addOneRow(root.left,v,d-1)
31. self.addOneRow(root.right,v,d-1)
32. **return** root

652寻找重复的子树

题目：给定一棵二叉树，返回所有重复的子树。对于同一类的重复子树，你只需要返回其中任意**一棵**的根结点即可。

两棵树重复是指它们具有相同的结构以及相同的结点值。

**示例 1：**

1

/ \

2 3

/ / \

4 2 4

/

4

下面是两个重复的子树：

2

/

4

和

4

因此，你需要以列表的形式返回上述重复子树的根结点。

思路：我们要找到一个重复的子树，可以把树和hash结合起来啊！

每一棵子树，都能把它的结构使用先序遍历或者后序遍历保存下来，然后把这个结构保存在hash表格里面，这样当我们下次再遇到这个树的结构的时候，就能很容易查表得知，然后放到结果res中

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** findDuplicateSubtrees(self, root):
10. """
11. :type root: TreeNode
12. :rtype: List[TreeNode]
13. """
15. res = []
16. m = collections.defaultdict(int)
17. self.helper(root, m, res)
18. **return** res
20. **def** helper(self, root, m, res):
21. **if** **not** root:
22. **return** '#'
23. path = str(root.val) + ',' + self.helper(root.left, m, res) + ',' + self.helper(root.right, m, res)
24. **if** m[path] == 1:
25. res.append(root)
26. m[path] += 1
27. **return** path

654最大二叉树

题目：给定一个不含重复元素的整数数组。一个以此数组构建的最大二叉树定义如下：

1. 二叉树的根是数组中的最大元素。
2. 左子树是通过数组中最大值左边部分构造出的最大二叉树。
3. 右子树是通过数组中最大值右边部分构造出的最大二叉树。

通过给定的数组构建最大二叉树，并且输出这个树的根节点。

**Example 1:**

**输入:** [3,2,1,6,0,5]

**输入:** 返回下面这棵树的根节点：

6

/ \

3 5

\ /

2 0

\

1

**注意:**

1. 给定的数组的大小在 [1, 1000] 之间

思路：使用递归操作，先找最大值，然后左右两边子树递归

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** constructMaximumBinaryTree(self, nums):
10. """
11. :type nums: List[int]
12. :rtype: TreeNode
13. """
14. **if** nums==[]:
15. **return** None
16. maxNode=max(nums)
17. maxIndex=nums.index(maxNode)
18. root=TreeNode(maxNode)
19. root.left=self.constructMaximumBinaryTree(nums[:maxIndex])
20. root.right=self.constructMaximumBinaryTree(nums[maxIndex+1:])
21. **return** root

655输出二叉树

题目：

在一个 m\*n 的二维字符串数组中输出二叉树，并遵守以下规则：

1. 行数 m 应当等于给定二叉树的高度。
2. 列数 n 应当总是奇数。
3. 根节点的值（以字符串格式给出）应当放在可放置的第一行正中间。根节点所在的行与列会将剩余空间划分为两部分（**左下部分和右下部分**）。你应该将左子树输出在左下部分，右子树输出在右下部分。左下和右下部分应当有相同的大小。即使一个子树为空而另一个非空，你不需要为空的子树输出任何东西，但仍需要为另一个子树留出足够的空间。然而，如果两个子树都为空则不需要为它们留出任何空间。
4. 每个未使用的空间应包含一个空的字符串""。
5. 使用相同的规则输出子树。

**示例 1:**

**输入:**

1

/

2

**输出:**

[["", "1", ""],

["2", "", ""]]

**示例 2:**

**输入:**

1

/ \

2 3

\

4

**输出:**

[["", "", "", "1", "", "", ""],

["", "2", "", "", "", "3", ""],

["", "", "4", "", "", "", ""]]

**示例 3:**

**输入:**

1

/ \

2 5

/

3

/

4

**输出:**

[["", "", "", "", "", "", "", "1", "", "", "", "", "", "", ""]

["", "", "", "2", "", "", "", "", "", "", "", "5", "", "", ""]

["", "3", "", "", "", "", "", "", "", "", "", "", "", "", ""]

["4", "", "", "", "", "", "", "", "", "", "", "", "", "", ""]]

**注意:** 二叉树的高度在范围 [1, 10] 中。

思路：最初认为，给空节点留下位置加大了题目难度。其实真正理解题目要考察的内容之后，发现这个条件让我们可以使用完全二叉树的数学公式，所以使题目变得简单了。

这个题首先要求出树的高度，然后求出完全二叉树的宽度。根据高度和宽度构建出二维数组，再利用递归求出每个层次的每个节点对应的二维数组的位置，设为节点的值即可。

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** printTree(self, root):
10. """
11. :type root: TreeNode
12. :rtype: List[List[str]]
13. """
14. **if** **not** root:
15. **return** [""]
16. **def** getDepth(root):
17. **if** **not** root:
18. **return** 0
19. **return** 1+max(getDepth(root.left),getDepth(root.right))
20. d=getDepth(root)
21. cols=2\*\*d-1
22. self.res=[["" **for** i **in** range(cols)] **for** j **in** range(d)]
23. **def** helper(root,d,pos):
24. self.res[-d-1][pos]=str(root.val)
25. **if** root.left:helper(root.left,d-1,pos-2\*\*(d-1))
26. **if** root.right:helper(root.right,d-1,pos+2\*\*(d-1))
27. helper(root,d-1,2\*\*(d-1)-1)

**return** self.res

662二叉树最大宽度

题目：

思路：

代码：

684冗余连接

题目：在本问题中, 树指的是一个连通且无环的**无向**图。

输入一个图，该图由一个有着N个节点 (节点值不重复1, 2, ..., N) 的树及一条附加的边构成。附加的边的两个顶点包含在1到N中间，这条附加的边不属于树中已存在的边。

结果图是一个以边组成的二维数组。每一个边的元素是一对[u, v] ，满足 u < v，表示连接顶点u 和v的**无向**图的边。

返回一条可以删去的边，使得结果图是一个有着N个节点的树。如果有多个答案，则返回二维数组中最后出现的边。答案边 [u, v] 应满足相同的格式 u < v。

**示例 1：**

**输入:** [[1,2], [1,3], [2,3]]

**输出:** [2,3]

**解释:** 给定的无向图为:

1

/ \

2 - 3

**示例 2：**

**输入:** [[1,2], [2,3], [3,4], [1,4], [1,5]]

**输出:** [1,4]

**解释:** 给定的无向图为:

5 - 1 - 2

| |

4 - 3

**注意:**

* 输入的二维数组大小在 3 到 1000。
* 二维数组中的整数在1到N之间，其中N是输入数组的大小。

思路：采用并查集的方法；实现并查集查找根节点的代码，并且做了路径压缩，防止树太高导致查找根节点缓慢。

具体到这个题，虽然说是返回最后一个边，但我们知道只需要去除一条边就够了，之前的边不会构成环，直至多余的那条边出现。

另外要注意，当一条边的左右节点的根节点不同时，要把他们设置相同，这样等下次判断某条边的左右节点相同的情况时，说明是多余的那条边了。

代码：

1. **class** Solution(object):
2. **def** findRedundantConnection(self, edges):
3. """
4. :type edges: List[List[int]]
5. :rtype: List[int]
6. """
7. tree=[-1]\*(len(edges)+1)
8. **for** edge **in** edges:
9. a=self.findRoot(edge[0],tree)
10. b=self.findRoot(edge[1],tree)
11. **if** a!=b:
12. tree[a]=b
13. **else**:
14. **return** edge
15. **def** findRoot(self,x,tree):
16. **if** tree[x]==-1:
17. **return** x
18. **else**:
19. root=self.findRoot(tree[x],tree)
20. tree[x]=root
21. **return** root

701二叉搜索树中的插入操作

题目:

给定二叉搜索树（BST）的根节点和要插入树中的值，将值插入二叉搜索树。 返回插入后二叉搜索树的根节点。 保证原始二叉搜索树中不存在新值。

注意，可能存在多种有效的插入方式，只要树在插入后仍保持为二叉搜索树即可。 你可以返回任意有效的结果。

例如,

给定二叉搜索树:

4

/ \

2 7

/ \

1 3

和 插入的值: 5

你可以返回这个二叉搜索树:

4

/ \

2 7

/ \ /

1 3 5

或者这个树也是有效的:

5

/ \

2 7

/ \

1 3

\

4

思路：使用递归方法，从根节点开始，依次判断，得到对应的位置，新建立节点并插入就好。

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** insertIntoBST(self, root, val):
10. """
11. :type root: TreeNode
12. :type val: int
13. :rtype: TreeNode
14. """
15. **if** **not** root:
16. **return** TreeNode(val)
17. **if** val>root.val:
18. root.right=self.insertIntoBST(root.right,val)
19. **if** val<root.val:
20. root.left=self.insertIntoBST(root.left,val)
21. **return** root

814二叉树剪枝

题目：

给定二叉树根结点 root ，此外树的每个结点的值要么是 0，要么是 1。

返回移除了所有不包含 1 的子树的原二叉树。

( 节点 X 的子树为 X 本身，以及所有 X 的后代。)

**示例1:**

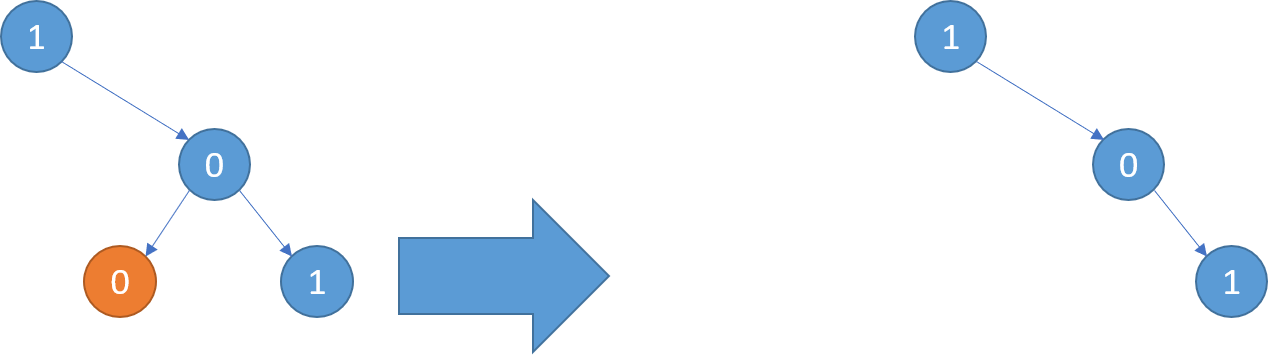
**输入:** [1,null,0,0,1]

**输出:** [1,null,0,null,1]

**解释:**

只有红色节点满足条件“所有不包含 1 的子树”。

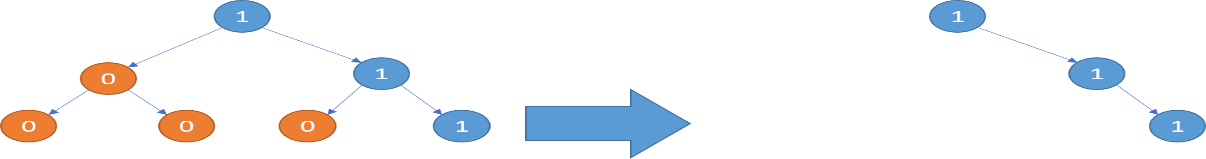
右图为返回的答案。



**示例2:**

**输入:** [1,0,1,0,0,0,1]

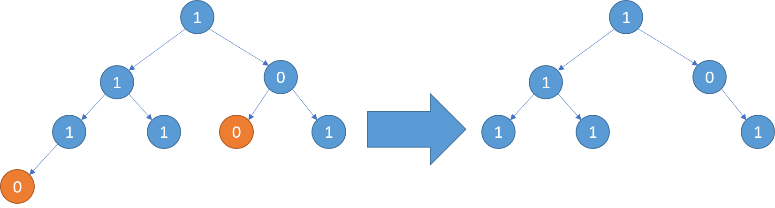
**输出:** [1,null,1,null,1]



**示例3:**

**输入:** [1,1,0,1,1,0,1,0]

**输出:** [1,1,0,1,1,null,1]



**说明:**

* 给定的二叉树最多有 100 个节点。
* 每个节点的值只会为 0 或 1 。

思路：

我们直接把节点的左孩子和右孩子重新设置就好了。这个题是后序遍历！

一定要注意的是，我们判断这个节点是叶子节点并且节点值是1的这个步骤要放在左右子树处理之后。可以从Example2中看出来，如果0节点的左右子节点都是0，那么把左右节点都减去了之后，还要判断自身是不是0，然后把自己也剪了。也就是说这一步相当于后序遍历，把孩子都处理结束之后，然后再处理自身。

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** pruneTree(self, root):
10. """
11. :type root: TreeNode
12. :rtype: TreeNode
13. """
14. **if** **not** root:
15. **return**
16. root.left=self.pruneTree(root.left)
17. root.right=self.pruneTree(root.right)
18. **if** **not** root.left **and** **not** root.right **and** root.val==0:
19. **return** None
20. **return** root

863二叉树中所有距离为K的结点

题目：

给定一个二叉树（具有根结点 root）， 一个目标结点 target ，和一个整数值 K 。

返回到目标结点 target 距离为 K 的所有结点的值的列表。 答案可以以任何顺序返回。

**示例 1：**

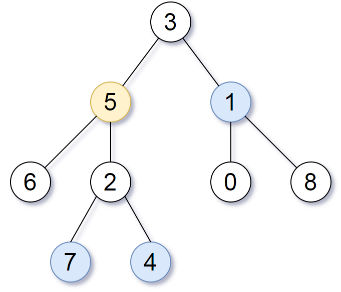
**输入：**root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2

**输出：**[7,4,1]

**解释：**

所求结点为与目标结点（值为 5）距离为 2 的结点，

值分别为 7，4，以及 1



注意，输入的 "root" 和 "target" 实际上是树上的结点。

上面的输入仅仅是对这些对象进行了序列化描述。

**提示：**

1. 给定的树是非空的，且最多有 K 个结点。
2. 树上的每个结点都具有唯一的值 0 <= node.val <= 500 。
3. 目标结点 target 是树上的结点。
4. 0 <= K <= 1000.

思路：这个题的做法就是通过DFS建立一个邻接矩阵，然后在这个邻接矩阵上使用BFS

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** distanceK(self, root, target, K):
10. """
11. :type root: TreeNode
12. :type target: TreeNode
13. :type K: int
14. :rtype: List[int]
15. """
16. #DFS
17. conn=collections.defaultdict(list)
18. **def** connect(parent,child):
19. **if** parent **and** child:
20. conn[parent.val].append(child.val)
21. conn[child.val].append(parent.val)
22. **if** child.left:
23. connect(child,child.left)
24. **if** child.right:
25. connect(child,child.right)
26. connect(None,root)
27. #BFS
28. que=collections.deque()
29. que.append(target.val)
30. visited=set([target.val])
31. **for** k **in** range(K):
32. size=len(que)
33. **for** i **in** range(size):
34. node=que.popleft()
35. **for** j **in** conn[node]:
36. **if** j **not** **in** visited:
37. que.append(j)
38. visited.add(j)
39. **return** list(que)

865具有所有最深结点的最小子树

题目：

给定一个根为 root 的二叉树，每个结点的*深度*是它到根的最短距离。

如果一个结点在**整个树**的任意结点之间具有最大的深度，则该结点是*最深的*。

一个结点的子树是该结点加上它的所有后代的集合。

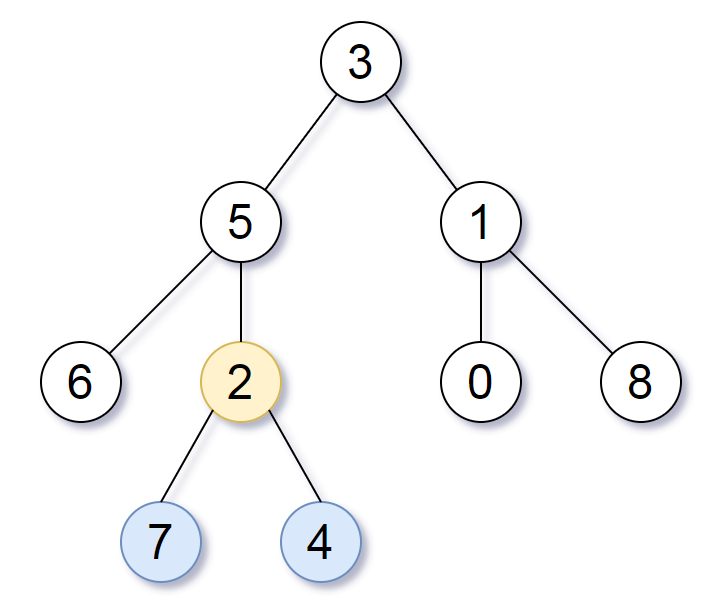
返回能满足“以该结点为根的子树中包含所有最深的结点”这一条件的具有最大深度的结点。

**示例：**

**输入：**[3,5,1,6,2,0,8,null,null,7,4]

**输出：**[2,7,4]

**解释：**



我们返回值为 2 的结点，在图中用黄色标记。

在图中用蓝色标记的是树的最深的结点。

输入 "[3, 5, 1, 6, 2, 0, 8, null, null, 7, 4]" 是对给定的树的序列化表述。

输出 "[2, 7, 4]" 是对根结点的值为 2 的子树的序列化表述。

输入和输出都具有 TreeNode 类型。

**提示：**

* 树中结点的数量介于 1 和 500 之间。
* 每个结点的值都是独一无二的。

思路：

这个题的模型其实比较左右子树的高度，如果左右子树的高度相等，说明当前节点就是要求的。这个解释是这样的：必须包含所有的最大高度的叶子，左右叶子高度相等，所以必须包含当前节点。

当左子树高度>右子树高度的时候，要求的节点在左边；反之，在右边。

所以，递归思路 + 一个pair。这个pair的思路是，保存了当前节点的深度和当前节点的最深子树节点。

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** subtreeWithAllDeepest(self, root):
10. """
11. :type root: TreeNode
12. :rtype: TreeNode
13. """
14. **return** self.depth(root)[1]
15. **def** depth(self,root):
16. **if** **not** root:
17. **return** 0,None
18. l,r=self.depth(root.left),self.depth(root.right)
19. **if** l[0]>r[0]:
20. **return** l[0]+1,l[1]
21. **elif** l[0]<r[0]:
22. **return** r[0]+1,r[1]
23. **else**:
24. **return** l[0]+1,root

889根据前序和后序遍历构造二叉树

题目：

返回与给定的前序和后序遍历匹配的任何二叉树。

 pre 和 post 遍历中的值是不同的正整数。

**示例：**

**输入：**pre = [1,2,4,5,3,6,7], post = [4,5,2,6,7,3,1]

**输出：**[1,2,3,4,5,6,7]

**提示：**

* 1 <= pre.length == post.length <= 30
* pre[] 和 post[] 都是 1, 2, ..., pre.length 的排列
* 每个输入保证至少有一个答案。如果有多个答案，可以返回其中一个。

思路：前序和后序的遍历并没有打乱整棵树的关系，一棵树的节点在两种遍历方式所得到的还都是在一块的。

所以pre[0]是根节点，也就是post[-1];

post[-2]时候右子树的根节点，因此在前序遍历中找到post[-2]的位置idx就能分开两棵子树

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** constructFromPrePost(self, pre, post):
10. """
11. :type pre: List[int]
12. :type post: List[int]
13. :rtype: TreeNode
14. """
15. **if** **not** pre **or** **not** post:
16. **return** None
17. root=TreeNode(pre[0])
18. **if** len(pre)==1:
19. **return** root
20. idx=pre.index(post[-2])
21. root.left=self.constructFromPrePost(pre[1:idx],post[:idx-1])
22. root.right=self.constructFromPrePost(pre[idx:],post[idx-1:-1])
23. **return** root

894所有可能的满二叉树

题目：

*满二叉树*是一类二叉树，其中每个结点恰好有 0 或 2 个子结点。

返回包含 N 个结点的所有可能满二叉树的列表。 答案的每个元素都是一个可能树的根结点。

答案中每个树的每个结点都**必须**有 node.val=0。

你可以按任何顺序返回树的最终列表

**输入：**7

**输出：**

[[0,0,0,null,null,0,0,null,null,0,0],

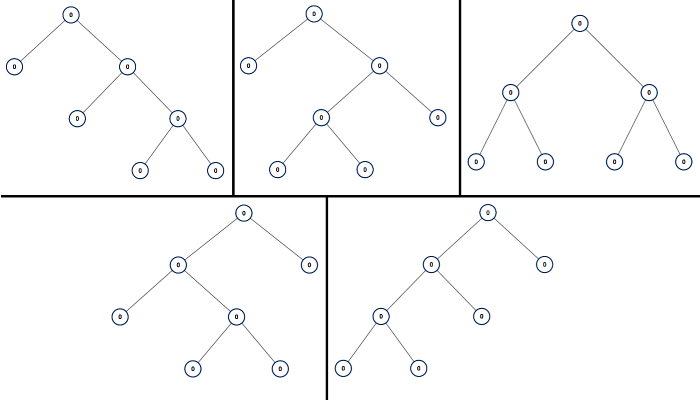
[0,0,0,null,null,0,0,0,0],

[0,0,0,0,0,0,0],

[0,0,0,0,0,null,null,null,null,0,0],

[0,0,0,0,0,null,null,0,0]]

**解释：**



思路：所有能构成的树，并且返回的不是数目，而是真正的树。所以一定会把所有的节点都求出来。一般就使用了递归。

这个题中，重点是返回一个列表，也就是说每个能够成的树的根节点都要放到这个列表里。而且当左子树、右子树的节点个数固定的时候，也会出现排列组合的情况，所以使用了两重for循环来完成所有的左右子树的组合。

另外的一个技巧就是，左右子树的个数一定是奇数个。

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** allPossibleFBT(self, N):
10. """
11. :type N: int
12. :rtype: List[TreeNode]
13. """
14. N -=1
15. **if** N==0:
16. **return** [TreeNode(0)]
17. res=[]
18. **for** l **in** range(1,N,2):
19. **for** left **in** self.allPossibleFBT(l):
20. **for** right **in** self.allPossibleFBT(N-l):
21. node=TreeNode(0)
22. node.left=left
23. node.right=right
24. res.append(node)
25. **return** res

919完成二叉树插入器

题目：

完全二叉树是每一层（除最后一层外）都是完全填充（即，结点数达到最大）的，并且所有的结点都尽可能地集中在左侧。

设计一个用完全二叉树初始化的数据结构 CBTInserter，它支持以下几种操作：

* CBTInserter(TreeNode root) 使用头结点为 root 的给定树初始化该数据结构；
* CBTInserter.insert(int v) 将 TreeNode 插入到存在值为 node.val = v  的树中以使其保持完全二叉树的状态，**并返回插入的 TreeNode 的父结点的值**；
* CBTInserter.get\_root() 将返回树的头结点。

**示例 1：**

**输入：**inputs = ["CBTInserter","insert","get\_root"], inputs = [[[1]],[2],[]]

**输出：**[null,1,[1,2]]

**示例 2：**

**输入：**inputs = ["CBTInserter","insert","insert","get\_root"], inputs = [[[1,2,3,4,5,6]],[7],[8],[]]

**输出：**[null,3,4,[1,2,3,4,5,6,7,8]]

**提示：**

1. 最初给定的树是完全二叉树，且包含 1 到 1000 个结点。
2. 每个测试用例最多调用 CBTInserter.insert  操作 10000 次。
3. 给定结点或插入结点的每个值都在 0 到 5000 之间。

思路：

完全二叉树是每一层都满的，因此找出要插入节点的父亲节点是很简单的。如果用数组tree保存着所有节点的层次遍历，那么新节点的父亲节点就是tree[(N -1)/2]，N是未插入该节点前的树的元素个数。

构建树的时候使用层次遍历，也就是BFS把所有的节点放入到tree里。插入的时候直接计算出新节点的父亲节点。获取root就是数组中的第0个节点。

时间复杂度是O(N)，空间复杂度是O(N)。

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** CBTInserter(object):
10. **def** \_\_init\_\_(self, root):
11. """
12. :type root: TreeNode
13. """
14. self.tree=list()
15. queue=collections.deque()
16. queue.append(root)
17. **while** queue:
18. node=queue.popleft()
19. self.tree.append(node)
20. **if** node.left:
21. queue.append(node.left)
22. **if** node.right:
23. queue.append(node.right)

26. **def** insert(self, v):
27. """
28. :type v: int
29. :rtype: int
30. """
31. \_len=len(self.tree)
32. father=self.tree[(\_len-1)/2]
33. node=TreeNode(v)
34. **if** **not** father.left:
35. father.left=node
36. **else**:
37. father.right=node
38. self.tree.append(node)
39. **return** father.val


43. **def** get\_root(self):
44. """
45. :rtype: TreeNode
46. """
47. **return** self.tree[0]

50. # Your CBTInserter object will be instantiated and called as such:
51. # obj = CBTInserter(root)
52. # param\_1 = obj.insert(v)
53. # param\_2 = obj.get\_root()

951反转等价二叉树

题目：

我们可以为二叉树 T 定义一个翻转操作，如下所示：选择任意节点，然后交换它的左子树和右子树。

只要经过一定次数的翻转操作后，能使 X 等于 Y，我们就称二叉树 X *翻转等价*于二叉树 Y。

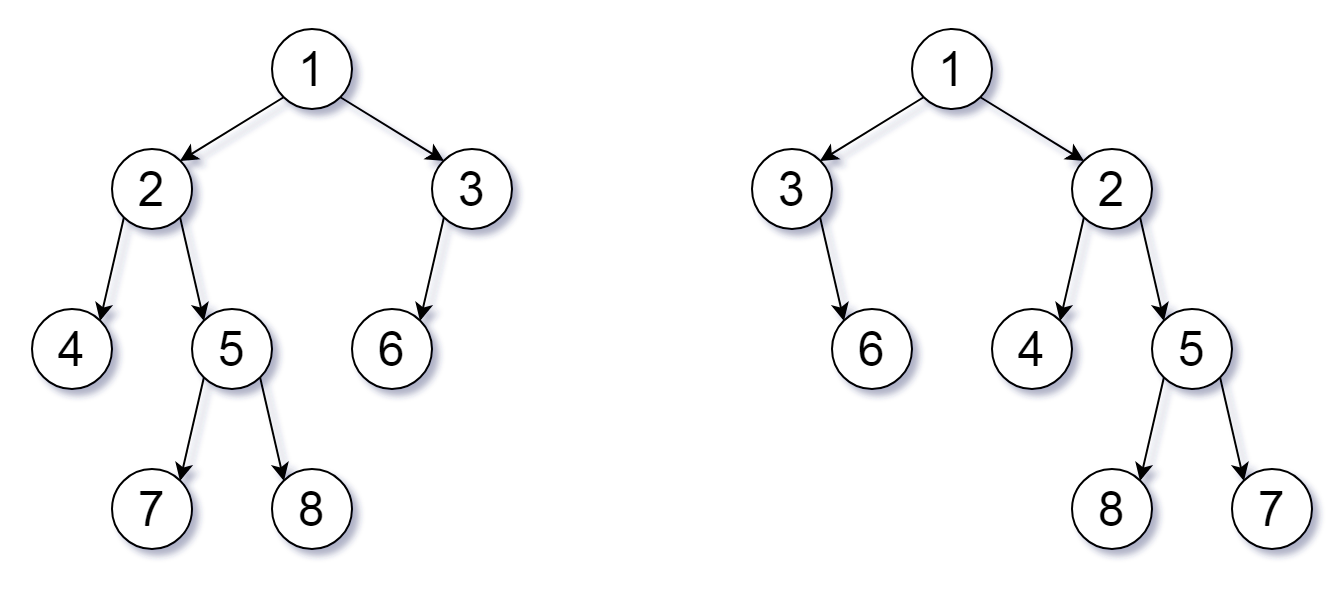
编写一个判断两个二叉树是否是*翻转等价*的函数。这些树由根节点 root1 和 root2 给出。

**示例：**

**输入：**root1 = [1,2,3,4,5,6,null,null,null,7,8], root2 = [1,3,2,null,6,4,5,null,null,null,null,8,7]

**输出：**true

**解释：**We flipped at nodes with values 1, 3, and 5.



**提示：**

1. 每棵树最多有 100 个节点。
2. 每棵树中的每个值都是唯一的、在 [0, 99] 范围内的整数。

思路：题目中我们不确定翻转了哪些节点，首先我们可以知道如果两个树都是空树，那么可以互相得到。如果两个数有一个是空，另一个不空，那么一定不能互相得到。如果两个树的节点的值不等，也不能互相得到。

重点来了：我们现在已经确定了两个树都不空，企鹅值相等，如何确定它们的子树们是否翻转相等呢？首先，我们来回顾一下flipEquiv函数的含义：判断两个树在进行翻转/不进行翻转的情况下，能不能相等。所以，我们不确定两个子树的状况，只需要对两个子树进行翻转/不翻转两种状态判断即可。如果进行翻转，那么root1的左子树可以通过root2的右子树得到，同时root1的左子树通过root2的右子树得到；如果不进行翻转，那么root1和root2的对应左右子树通过操作应该也能互相得到。

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** flipEquiv(self, root1, root2):
10. """
11. :type root1: TreeNode
12. :type root2: TreeNode
13. :rtype: bool
14. """
15. **if** **not** root1 **and** **not** root2:
16. **return** True
17. **if** **not** root1 **and** root2:
18. **return** False
19. **if** root1 **and** **not** root2:
20. **return** False
21. **if** root1.val!=root2.val:
22. **return** False
23. **return** (self.flipEquiv(root1.left,root2.right) **and** self.flipEquiv(root1.right,root2.left))
24. **or** (self.flipEquiv(root1.left,root2.left) **and** self.flipEquiv(root1.right,root2.right))

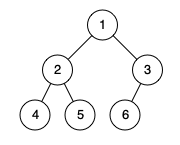
958二叉树的完全性检查

题目：

给定一个二叉树，确定它是否是一个*完全二叉树*。

[百度百科](https://baike.baidu.com/item/完全二叉树/7773232?fr=aladdin)中对完全二叉树的定义如下：

若设二叉树的深度为 h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。（注：第 h 层可能包含 1~ 2h 个节点。）



**输入：**[1,2,3,4,5,6]

**输出：**true

**解释：**最后一层前的每一层都是满的（即，结点值为 {1} 和 {2,3} 的两层），且最后一层中的所有结点（{4,5,6}）都尽可能地向左。

思路：使用BFS的话层次遍历比较简单，因为我们从每层的从左到右进行遍历，如果某一层已经出现None之后，后面还有非空叶子节点的话，那么就不是完全二叉树

代码：

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** isCompleteTree(self, root):
10. """
11. :type root: TreeNode
12. :rtype: bool
13. """
14. **if** **not** root:
15. **return** True
16. res=[]
17. que=collections.deque()
18. que.append(root)
19. hasNone=False
20. **while** que:
21. size=len(que)
22. **for** i **in** range(size):
23. node=que.popleft()
24. **if** **not** node:
25. hasNone=True
26. **continue**
27. **if** hasNone:
28. **return** False
29. que.append(node.left)
30. que.append(node.right)
31. **return** True

971翻转二叉树以匹配先序遍历

题目：

给定一个有 N 个节点的二叉树，每个节点都有一个不同于其他节点且处于 {1, ..., N} 中的值。

通过交换节点的左子节点和右子节点，可以翻转该二叉树中的节点。

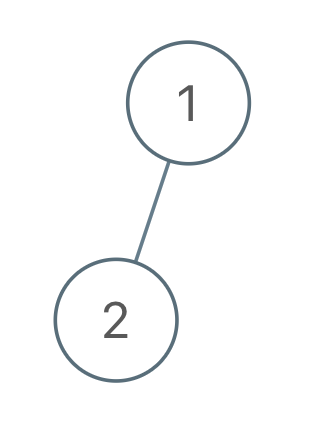
考虑从根节点开始的先序遍历报告的 N 值序列。将这一 N 值序列称为树的行程。

（回想一下，节点的先序遍历意味着我们报告当前节点的值，然后先序遍历左子节点，再先序遍历右子节点。）

我们的目标是翻转**最少的**树中节点，以便树的行程与给定的行程 voyage 相匹配。

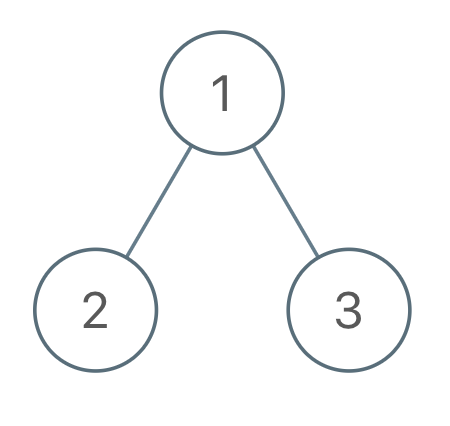
如果可以，则返回翻转的所有节点的值的列表。你可以按任何顺序返回答案。

如果不能，则返回列表 [-1]。



**输入：**root = [1,2], voyage = [2,1]

**输出：**[-1]



**输入：**root = [1,2,3], voyage = [1,3,2]

**输出：**[1]

思路：这个问题非常简单，我们只要遍历root，然后遍历voyage，比较两者的val是不是一样，不一样的话，自然就不对，如果一样的话，继续遍历下去即可。我们发现此时的root.val和voyage[0]一致，所以我们比较root.left和root.right。我们发现此时root.left和voyage[1]不一样，所以我们需要将root.left和root.right交换，然后继续判断root.left和voyage[1]，并且判断voyage[2]和root.right。

代码：代码有一些问题

1. # Definition for a binary tree node.
2. # class TreeNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
8. **class** Solution(object):
9. **def** flipMatchVoyage(self, root, voyage):
10. """
11. :type root: TreeNode
12. :type voyage: List[int]
13. :rtype: List[int]
14. """
16. res = list()
17. i = 0
18. **def** dfs(root):
19. nonlocal i
20. **if** **not** root:
21. **return** True
23. **if** root.val != voyage[i]:
24. **return** False
26. i += 1
27. **if** root.left **and** root.left.val != voyage[i]:
28. res.append(root.val)
29. root.left, root.right = root.right, root.left
31. **return** dfs(root.left) **and** dfs(root.right)
33. **return** res **if** dfs(root) **else** [-1]