94. Binary Tree Inorder Traversal

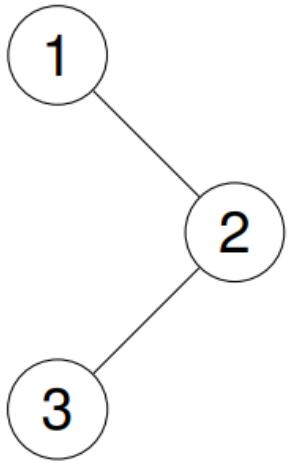
Given the `root` of a binary tree, return *the inorder traversal of its nodes' values*.

Example 1:

Input: root = [1,null,2,3]

Output: [1,3,2]

Explanation:

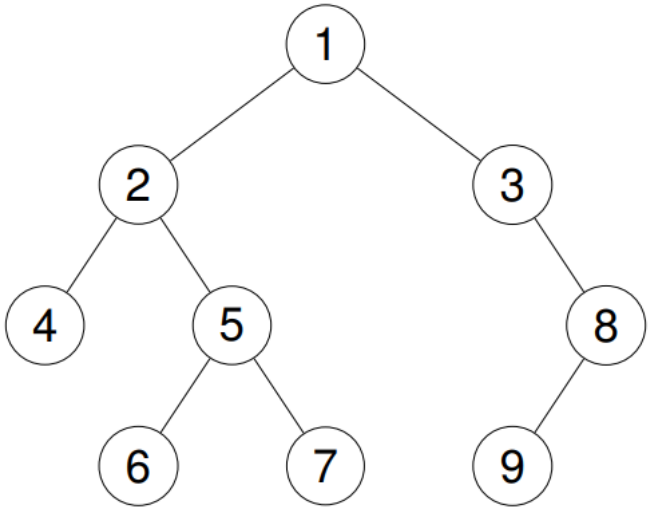


Example 2:

Input: root = [1,2,3,4,5,null,8,null,null,6,7,9]

Output: [4,2,6,5,7,1,3,9,8]

Explanation:



Example 3:

Input: root = []

Output: []

Example 4:

Input: root = [1]

Output: [1]

Constraints:

- The number of nodes in the tree is in the range `[0, 100]`.
- `-100 <= Node.val <= 100`

Follow up: Recursive solution is trivial, could you do it iteratively?

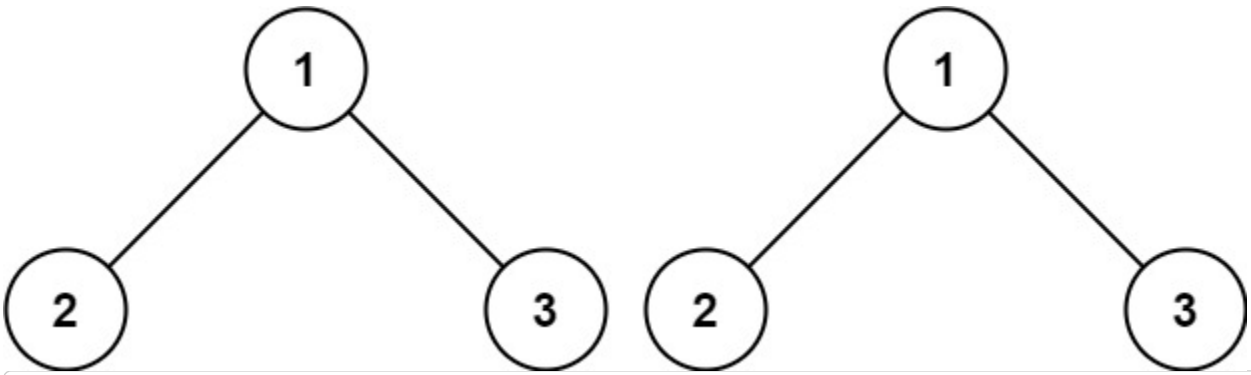
```
class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if not root:return []
        ans = []
        cur = root
        stack = []

        while cur or stack:
            while cur:
                stack.append(cur)
                cur = cur.left
            cur = stack.pop()
            ans.append(cur.val)

            cur = cur.right
        return ans
```

100. Same Tree

Given the roots of two binary trees `p` and `q`, write a function to check if they are the same or not.

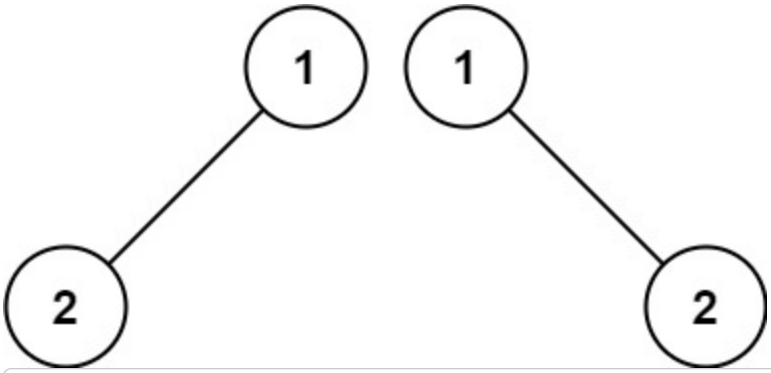
Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

Example 1:



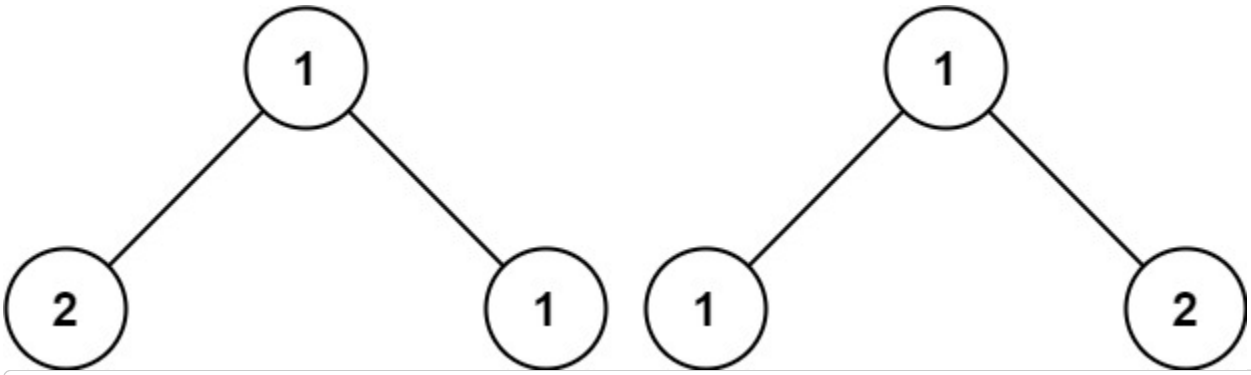

```
Input: p = [1,2,3], q = [1,2,3]
Output: true
```

Example 2:




```
Input: p = [1,2], q = [1,null,2]
Output: false
```

Example 3:




```
Input: p = [1,2,1], q = [1,1,2]
Output: false
```

Constraints:

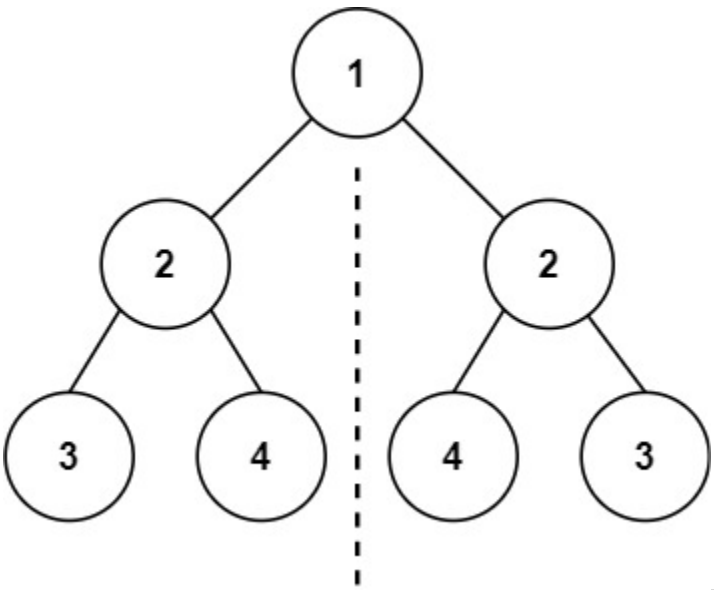
- The number of nodes in both trees is in the range `[0, 100]`.
- `-10^4 <= Node.val <= 10^4`

```
class Solution:
    def isSameTree(self, p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
        def dfs(p,q):
            if not p and not q:
                return True
            if not p and q or p and not q:
                return False
            if p.val != q.val: return False
            if not dfs(p.left,q.left):return False
            if not dfs(p.right,q.right):return False
            return True
        return dfs(p,q)
```

101. Symmetric Tree

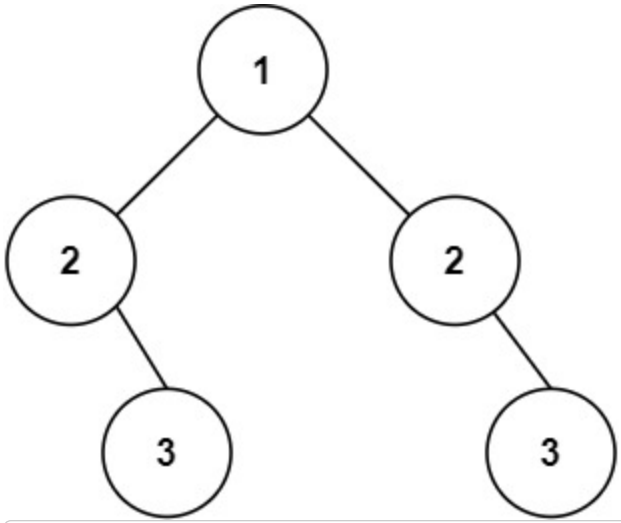
Given the `root` of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

Example 1:




```
Input: root = [1,2,2,3,4,4,3]
Output: true
```

Example 2:




```
Input: root = [1,2,2,null,3,null,3]
Output: false
```

Constraints:

- The number of nodes in the tree is in the range `[1, 1000]` .
- `-100 <= Node.val <= 100`

Follow up: Could you solve it both recursively and iteratively?

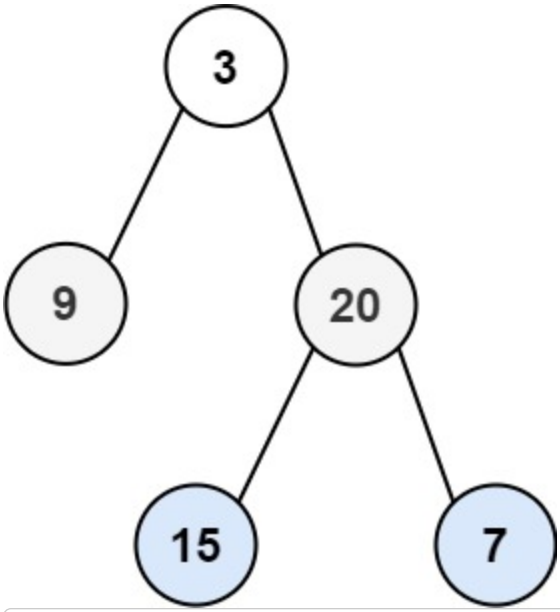
```
class Solution:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        def dfs(p,q):
            if not p and not q:return True
            if not p or not q : return False
            if p.val != q.val :return False
            return  dfs(p.left,q.right) and dfs(p.right,q.left)

        return dfs(root.left,root.right)
```

---

102. Binary Tree Level Order Traversal

Given the `root` of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

Example 1:




```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]
```

Example 2:

```
Input: root = [1]
Output: [[1]]
```

Example 3:

```
Input: root = []
Output: []
```

Constraints:

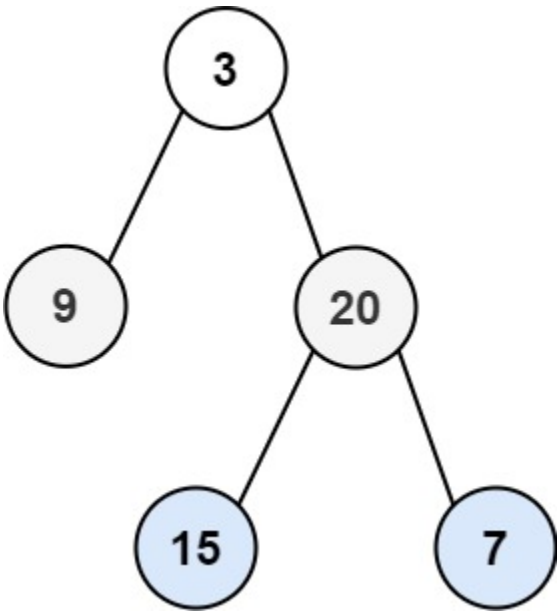
- The number of nodes in the tree is in the range `[0, 2000]`.
- `-1000 <= Node.val <= 1000`

```
from collections import deque
class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        if not root:return []
        result = []
        queue = deque([root])
        while queue:
            level = []
            for _ in range(len(queue)):
                node = queue.popleft()
                level.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            result.append(level)
        return result
```

103. Binary Tree Zigzag Level Order Traversal

Given the `root` of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

Example 1:



```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[20,9],[15,7]]
```

Example 2:

```
Input: root = [1]
Output: [[1]]
```

Example 3:

```
Input: root = []
Output: []
```

Constraints:

- The number of nodes in the tree is in the range `[0, 2000]` .
- `-100 <= Node.val <= 100`

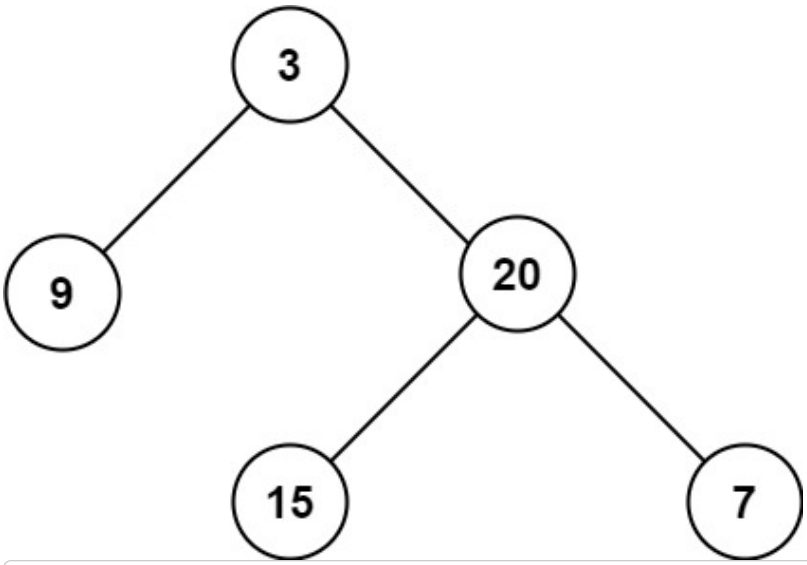
```
from collections import deque
class Solution:
    def zigzagLevelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        if not root:
            return []
        que = deque([root])
        res = []
        left_to_right = True
        while que:
            level = deque()
            for _ in range(len(que)):
                node = que.popleft()
                level.append(node.val)
                if left_to_right:
                    level.append(node.val)
                else:
                    level.appendleft(node.val)

                if node.left:
                    que.append(node.left)
                if node.right:
                    que.append(node.right)
            res.append(level)
            left_to_right = not left_to_right
        return res
```

104. Maximum Depth of Binary Tree

Given the `root` of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

Example 1:




```
Input: root = [3,9,20,null,null,15,7]
Output: 3
```

Example 2:

```
Input: root = [1,null,2]
Output: 2
```

Constraints:

- The number of nodes in the tree is in the range `[0, 10^4]`.
- `-100 <= Node.val <= 100`

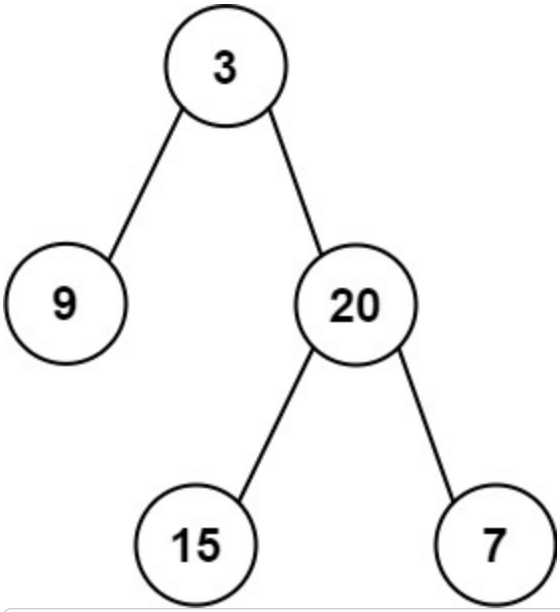
```
class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        if not root: return 0
        left = self.maxDepth(root.left)
        right = self.maxDepth(root.right)
        return 1+ max(left,right)
```

---

105. Construct Binary Tree from Preorder and Inorder Traversal

Given two integer arrays `preorder` and `inorder` where `preorder` is the preorder traversal of a binary tree and `inorder` is the inorder traversal of the same tree, construct and return *the binary tree*.

Example 1:




```
Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]
```

Example 2:

```
Input: preorder = [-1], inorder = [-1]
Output: [-1]
```

Constraints:

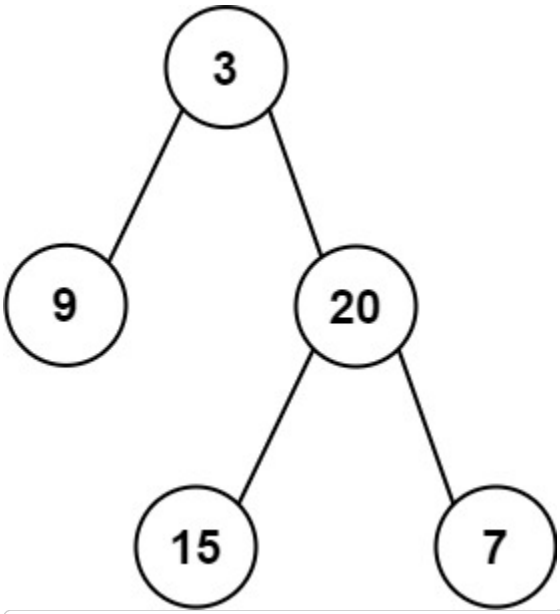
- `1 <= preorder.length <= 3000`
- `inorder.length == preorder.length`
- `-3000 <= preorder[i], inorder[i] <= 3000`
- `preorder` and `inorder` consist of **unique** values.
- Each value of `inorder` also appears in `preorder`.
- `preorder` is **guaranteed** to be the preorder traversal of the tree.
- `inorder` is **guaranteed** to be the inorder traversal of the tree.

```
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNo
de]:
        mapp = {num:i for i,num in enumerate(inorder)}
        def dfs(ind,in_start,in_end):
            if in_start > in_end:
                return None
            root = TreeNode(preorder[ind[0]])
            in_index = mapp[root.val]
            ind[0]+=1
            root.left = dfs(ind,in_start,in_index-1)
            root.right = dfs(ind,in_index+1,in_end)
            return root
        return dfs([0],0,len(inorder)-1)
```

106. Construct Binary Tree from Inorder and Postorder Traversal

Given two integer arrays `inorder` and `postorder` where `inorder` is the inorder traversal of a binary tree and `postorder` is the postorder traversal of the same tree, construct and return *the binary tree*.

Example 1:




```
Input: inorder = [9,3,15,20,7], postorder = [9,15,7,20,3]
Output: [3,9,20,null,null,15,7]
```

Example 2:

```
Input: inorder = [-1], postorder = [-1]
Output: [-1]
```

Constraints:

- `1 <= inorder.length <= 3000`
- `postorder.length == inorder.length`
- `-3000 <= inorder[i], postorder[i] <= 3000`
- `inorder` and `postorder` consist of **unique** values.
- Each value of `postorder` also appears in `inorder`.
- `inorder` is **guaranteed** to be the inorder traversal of the tree.
- `postorder` is **guaranteed** to be the postorder traversal of the tree.

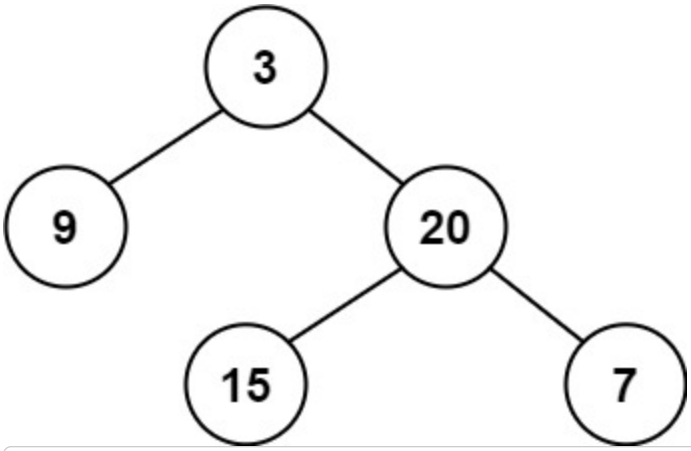
```
class Solution:
    def buildTree(self, inorder: List[int], postorder: List[int]) -> Optional[TreeN
ode]:
        mapp = {num:i for i,num in enumerate(inorder)}
        postorder = postorder[::-1]
        def dfs(ind,in_start,in_end):
            if in_start > in_end:
                return None
            root = TreeNode(postorder[ind[0]])
            in_index = mapp[root.val]
            ind[0]+=1
            root.right = dfs(ind,in_index+1,in_end)
            root.left = dfs(ind,in_start,in_index-1)
            return root

        return dfs([0],0,len(inorder)-1)
```

---

110. Balanced Binary Tree

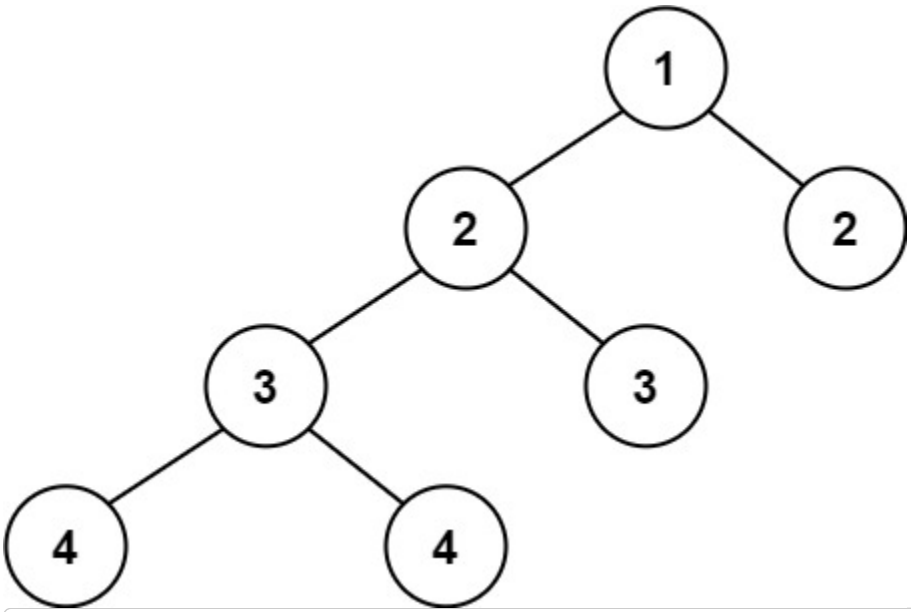
Given a binary tree, determine if it is **height-balanced**.

Example 1:




```
Input: root = [3,9,20,null,null,15,7]
Output: true
```

Example 2:




```
Input: root = [1,2,2,3,3,null,null,4,4]
Output: false
```

Example 3:

```
Input: root = []
Output: true
```

Constraints:

- The number of nodes in the tree is in the range `[0, 5000]`.
- `-10^4 <= Node.val <= 10^4`

```
class Solution:
    def isBalanced(self, root: Optional[TreeNode]) -> bool:
        def check(root):
            if not root:
                return 0
            left = check(root.left)
            if left == -1:
                return -1
            right = check(root.right)
            if right == -1:
                return -1
            if abs(left - right) > 1:
                return -1
            return 1 + max(left, right)

        return check(root) != -1
```

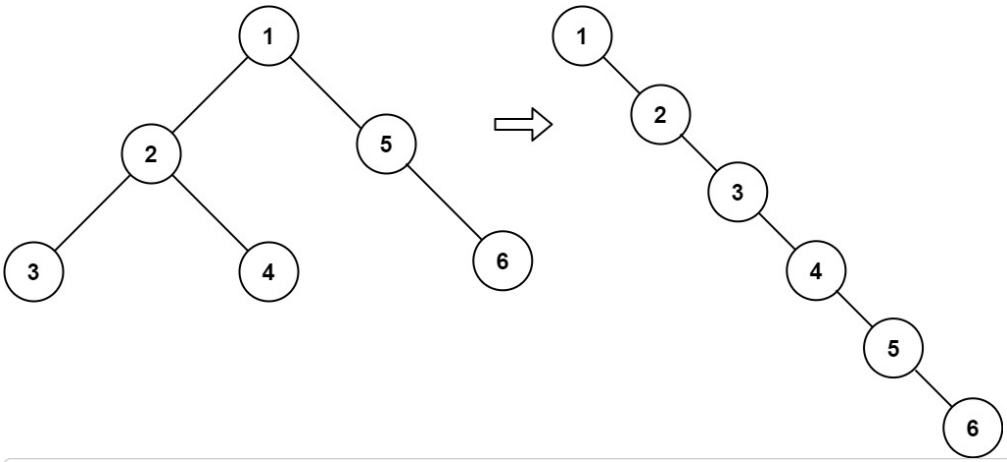
---

114. Flatten Binary Tree to Linked List

Given the `root` of a binary tree, flatten the tree into a "linked list":

- The "linked list" should use the same `TreeNode` class where the `right` child pointer points to the next node in the list and the `left` child pointer is always `null`.
- The "linked list" should be in the same order as a **pre-order traversal** (https://en.wikipedia.org/wiki/Tree_traversal#Pre-order,_NLR) of the binary tree.

Example 1:




```
Input: root = [1,2,5,3,4,null,6]
Output: [1,null,2,null,3,null,4,null,5,null,6]
```

Example 2:

```
Input: root = []
Output: []
```

Example 3:

```
Input: root = [0]
Output: [0]
```

Constraints:

- The number of nodes in the tree is in the range `[0, 2000]`.
- `-100 <= Node.val <= 100`

Follow up: Can you flatten the tree in-place (with `O(1)` extra space)?

```
class Solution:
    def flatten(self, root: Optional[TreeNode]) -> None:
        prev = None
        def dfs(node):
            nonlocal prev
            if not node:
                return None

            dfs(node.right)
            dfs(node.left)

            node.right = prev
            node.left = None
            prev = node
        dfs(root)
```

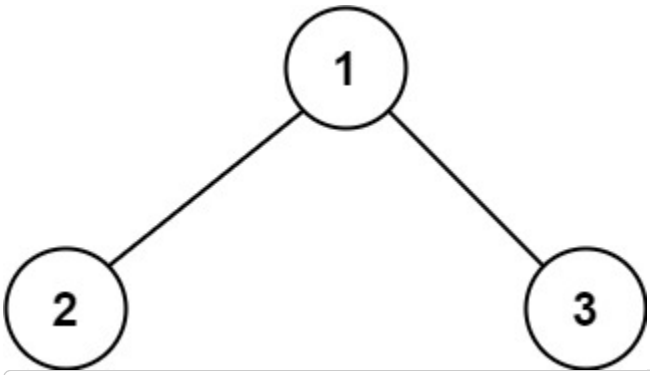
124. Binary Tree Maximum Path Sum

A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

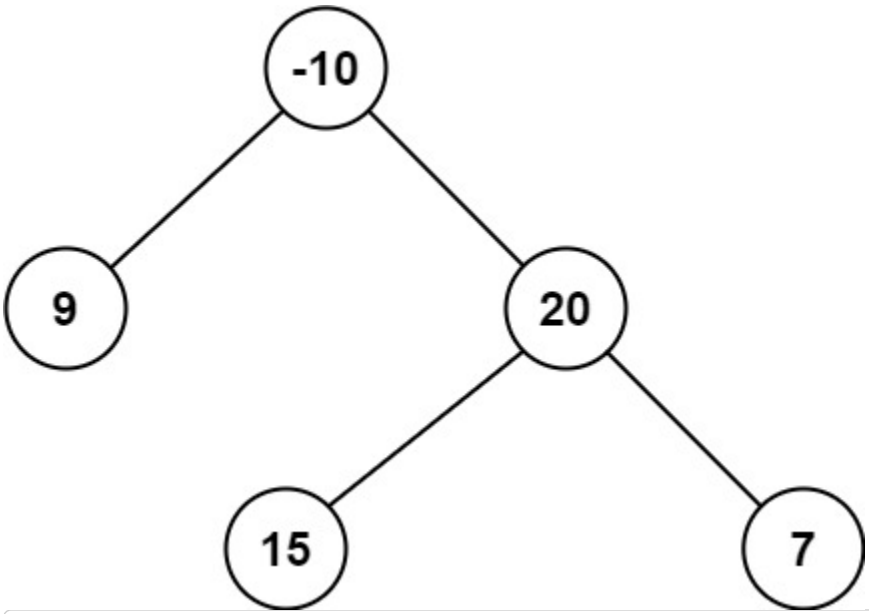
Given the `root` of a binary tree, return *the maximum **path sum** of any **non-empty** path*.

Example 1:




```
Input: root = [1,2,3]
Output: 6
Explanation: The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.
```

Example 2:




```
Input: root = [-10,9,20,null,null,15,7]
Output: 42
Explanation: The optimal path is 15 -> 20 -> 7 with a path sum of 15 + 20 + 7 = 42.
```

Constraints:

- The number of nodes in the tree is in the range `[1, 3 * 10^4]`.

- `-1000 <= Node.val <= 1000`

```
class Solution:
    def maxPathSum(self, root: Optional[TreeNode]) -> int:
        maxx_path = root.val

        def dfs(node):
            nonlocal maxx_path
            if not node:return 0

            left = max(0,dfs(node.left))
            right = max(0,dfs(node.right))

            maxx_path = max(maxx_path,left+right+node.val)

            return node.val + max(left,right)
        dfs(root)
        return maxx_path
```

144. Binary Tree Preorder Traversal

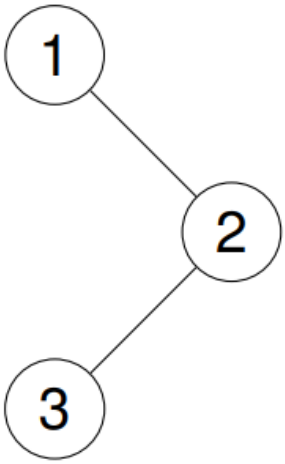
Given the `root` of a binary tree, return *the preorder traversal of its nodes' values*.

Example 1:

Input: root = [1,null,2,3]

Output: [1,2,3]

Explanation:

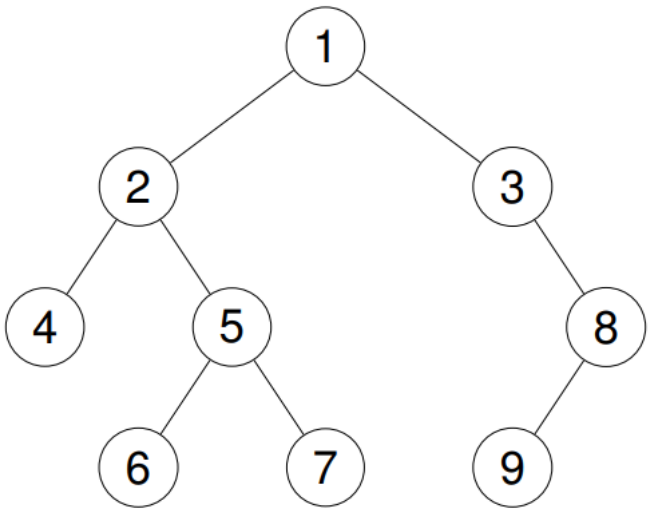


Example 2:

Input: root = [1,2,3,4,5,null,8,null,null,6,7,9]

Output: [1,2,4,5,6,7,3,8,9]

Explanation:



Example 3:

Input: root = []

Output: []

Example 4:

Input: root = [1]

Output: [1]

Constraints:

- The number of nodes in the tree is in the range `[0, 100]`.
- `-100 <= Node.val <= 100`

Follow up: Recursive solution is trivial, could you do it iteratively?

```
class Solution:
    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return []
        que = [root]
        ans = []
        while que:
            node = que.pop()
            ans.append(node.val)
            if node.right:
                que.append(node.right)
            if node.left:
                que.append(node.left)
        return ans
```

145. Binary Tree Postorder Traversal

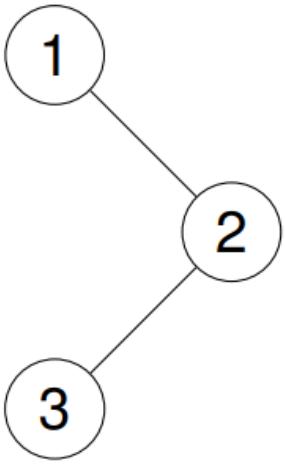
Given the `root` of a binary tree, return *the postorder traversal of its nodes' values*.

Example 1:

Input: root = [1,null,2,3]

Output: [3,2,1]

Explanation:

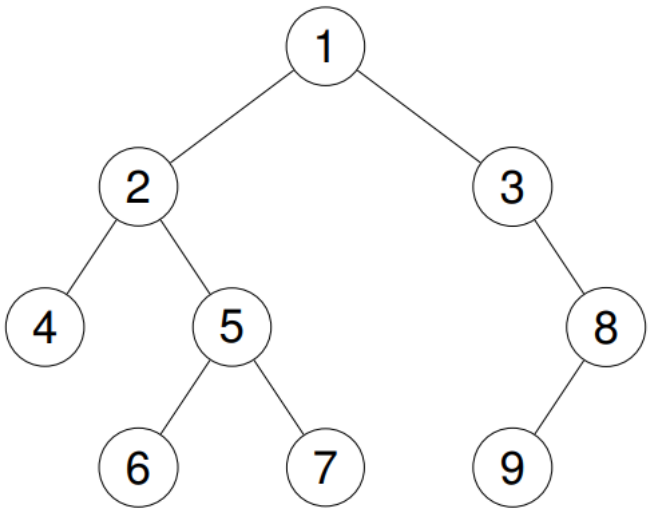


Example 2:

Input: root = [1,2,3,4,5,null,8,null,null,6,7,9]

Output: [4,6,7,5,2,9,8,3,1]

Explanation:



Example 3:

Input: root = []

Output: []

Example 4:

Input: root = [1]

Output: [1]

Constraints:

- The number of the nodes in the tree is in the range `[0, 100]`.
- `-100 <= Node.val <= 100`

Follow up: Recursive solution is trivial, could you do it iteratively?

```
class Solution:
    def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if not root:return []
        result = []
        stack = [root]
        while stack:
            current = stack.pop()
            result.append(current.val)
            if current.left:
                stack.append(current.left)
            if current.right:
                stack.append(current.right)
        return result[::-1]
```

199. Binary Tree Right Side View

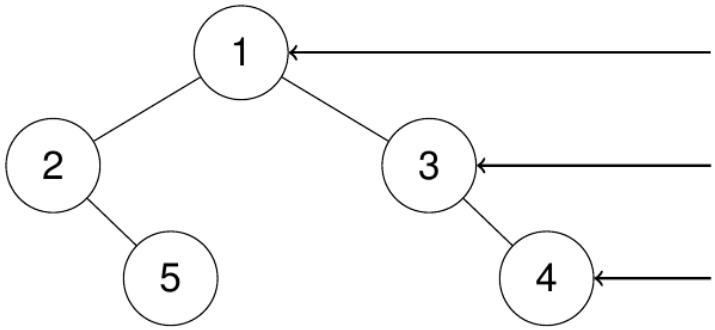
Given the `root` of a binary tree, imagine yourself standing on the **right side** of it, return *the values of the nodes you can see ordered from top to bottom*.

Example 1:

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

Output: [1,3,4]

Explanation:

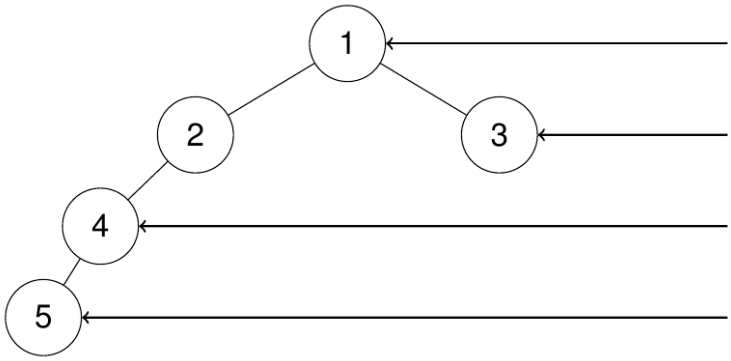


Example 2:

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

Output: [1,3,4,5]

Explanation:



Example 3:

Input: root = [1,null,3]

Output: [1,3]

Example 4:

Input: root = []

Output: []

Constraints:

- The number of nodes in the tree is in the range `[0, 100]`.
- `-100 <= Node.val <= 100`

```
from collections import deque
class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        if not root:return []
        level = 0
        res= []
        que = deque([[root,0]])
        while que:
            node,level = que.popleft()
            if level == len(res):
                res.append(node.val)
            if node.right:
                que.append([node.right,level+1])
            if node.left:
                que.append([node.left,level+1])
        return res
```

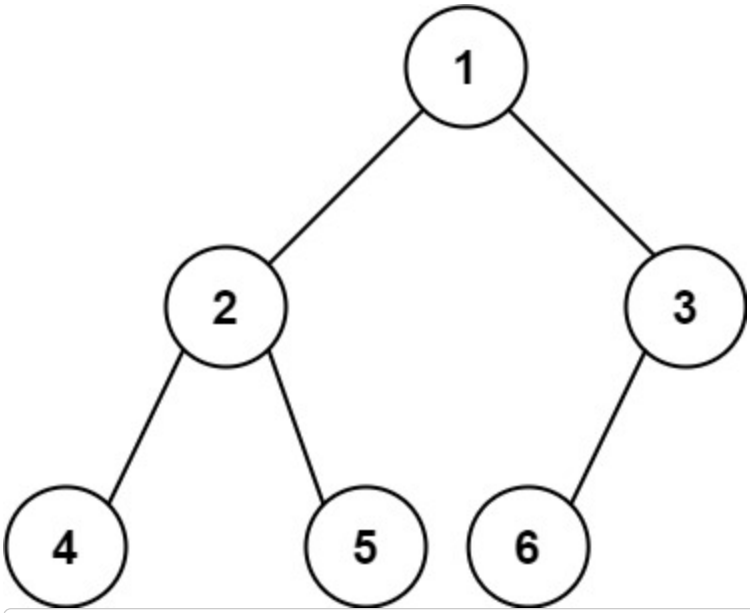
222. Count Complete Tree Nodes

Given the `root` of a **complete** binary tree, return the number of the nodes in the tree.

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

Design an algorithm that runs in less than `O(n)` time complexity.

Example 1:




```
Input: root = [1,2,3,4,5,6]
Output: 6
```

Example 2:

```
Input: root = []
Output: 0
```

Example 3:

```
Input: root = [1]
Output: 1
```

Constraints:

- The number of nodes in the tree is in the range `[0, 5 * 10^4]`.
- `0 <= Node.val <= 5 * 10^4`
- The tree is guaranteed to be **complete**.

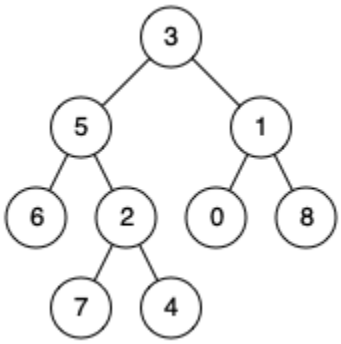
```
class Solution:
    def countNodes(self, root: Optional[TreeNode]) -> int:
        cur = root
        level = 0
        while cur:
            level+=1
            cur = cur.left
        cur = root
        last = 0
        while cur:
            cur = cur.right
            last +=1
        if last == level:
            return (2**level) -1
        return 1+ self.countNodes(root.left)+ self.countNodes(root.right)
```

236. Lowest Common Ancestor of a Binary Tree

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

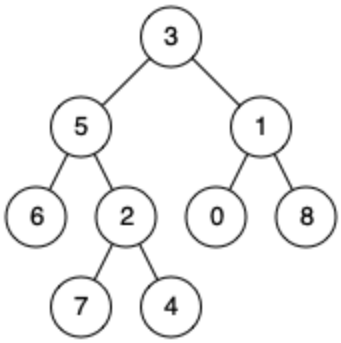
According to the definition of LCA on Wikipedia (https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes `p` and `q` as the lowest node in `T` that has both `p` and `q` as descendants (where we allow **a node to be a descendant of itself**).”

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 itsel
```

Example 3:

```
Input: root = [1,2], p = 1, q = 2
Output: 1
```

Constraints:

- The number of nodes in the tree is in the range `[2, 10^5]`.
- `-10^9 <= Node.val <= 10^9`
- All `Node.val` are **unique**.
- `p != q`
- `p` and `q` will exist in the tree.

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode')
-> 'TreeNode':
        def dfs(root,x,y):
            if not root or root == p or root == q:return root
            left = dfs(root.left,x,y)
            right = dfs(root.right,x,y)
            if left and right:
                return root
            return left if left else right
        return dfs(root,p,q)
```

---

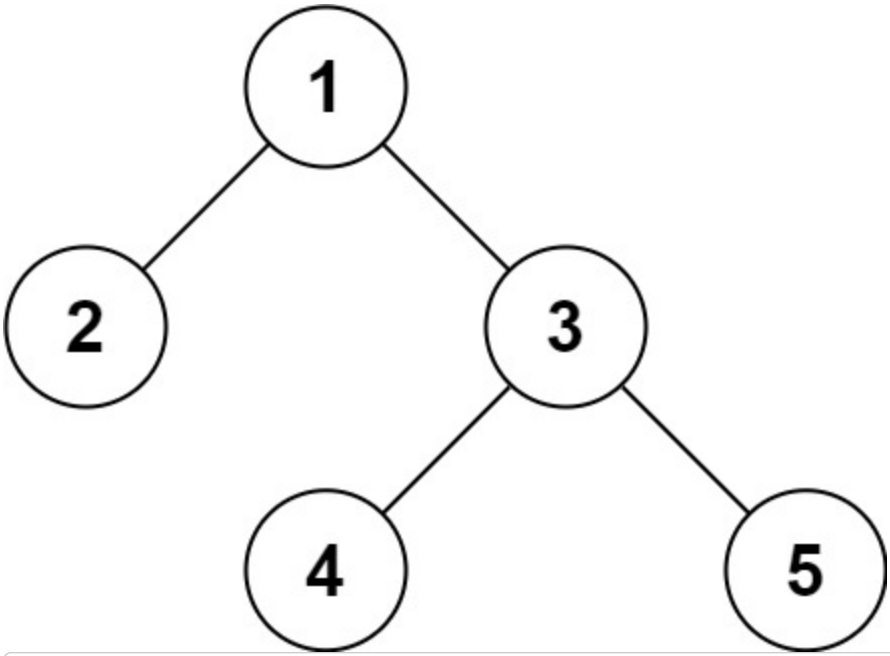
297. Serialize and Deserialize Binary Tree

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

Clarification: The input/output format is the same as how LeetCode serializes a binary tree (https://support.leetcode.com/hc/en-us/articles/32442719377939-How-to-create-test-cases-on-LeetCode#h_01J5EGREAW3NAEJ14XC07GRW1A). You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

Example 1:




```
Input: root = [1,2,3,null,null,4,5]
Output: [1,2,3,null,null,4,5]
```

Example 2:

```
Input: root = []
Output: []
```

Constraints:

- The number of nodes in the tree is in the range `[0, 10^4]`.
- `-1000 <= Node.val <= 1000`

```
from collections import deque
class Codec:
    def serialize(self, root):
        if not root:return "#"
        res = []
        que = deque([root])
        while que:
            node = que.popleft()
            if node:
                res.append(str(node.val))
                que.append(node.left)
                que.append(node.right)
            else:
                res.append("#")
        return ",".join(res)

    def deserialize(self, data):
        if not data or data[0] == "#":
            return None
        data = data.split(",")
        root = TreeNode(int(data[0]))
        que = deque([root])
        i=1
        while que and i < len(data):
            node = que.popleft()
            if data[i] != "#":
                node.left = TreeNode(int(data[i]))
                que.append(node.left)
            i+=1

            if data[i] != "#":
                node.right = TreeNode(int(data[i]))
                que.append(node.right)
            i+=1
        return root
```

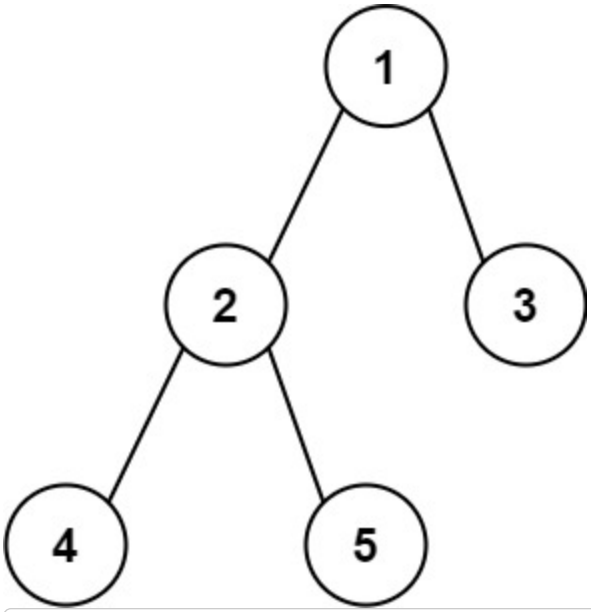
543. Diameter of Binary Tree

Given the `root` of a binary tree, return *the length of the **diameter** of the tree*.

The **diameter** of a binary tree is the **length** of the longest path between any two nodes in a tree. This path may or may not pass through the `root`.

The **length** of a path between two nodes is represented by the number of edges between them.

Example 1:




```
Input: root = [1,2,3,4,5]
Output: 3
Explanation: 3 is the length of the path [4,2,1,3] or [5,2,1,3].
```

Example 2:

```
Input: root = [1,2]
Output: 1
```

Constraints:

- The number of nodes in the tree is in the range $[1, 10^4]$.
- `-100 <= Node.val <= 100`

```
class Solution:
    def diameterOfBinaryTree(self, node: Optional[TreeNode]) -> int:
        maxx = 0
        def dfs(root):
            nonlocal maxx
            if not root :return 0
            left = dfs(root.left)
            right = dfs(root.right)
            maxx = max(maxx,left+right)
            return 1 + max(left,right)
        dfs(node)
        return maxx
```

662. Maximum Width of Binary Tree

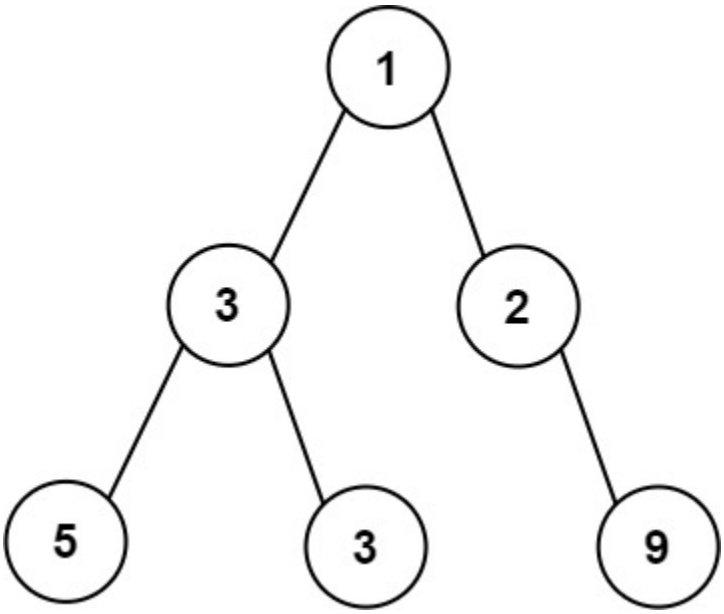
Given the `root` of a binary tree, return *the **maximum width** of the given tree*.

The **maximum width** of a tree is the maximum **width** among all levels.

The **width** of one level is defined as the length between the end-nodes (the leftmost and rightmost non-null nodes), where the null nodes between the end-nodes that would be present in a complete binary tree extending down to that level are also counted into the length calculation.

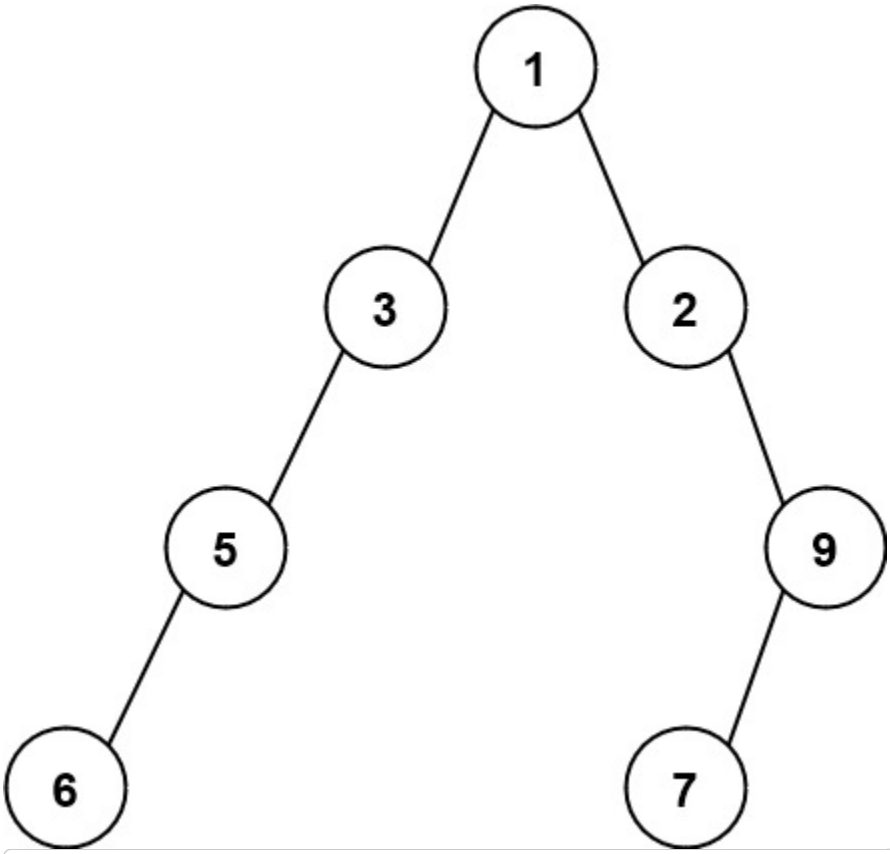
It is **guaranteed** that the answer will in the range of a **32-bit** signed integer.

Example 1:



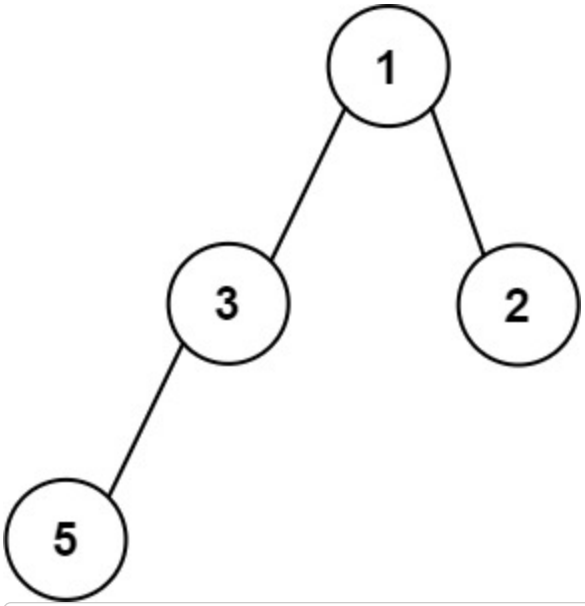
```
Input: root = [1,3,2,5,3,null,9]
Output: 4
Explanation: The maximum width exists in the third level with length 4 (5,3,null,9).
```

Example 2:




```
Input: root = [1,3,2,5,null,null,9,6,null,7]
Output: 7
Explanation: The maximum width exists in the fourth level with length 7 (6,null,null,
```

Example 3:




```
Input: root = [1,3,2,5]
Output: 2
Explanation: The maximum width exists in the second level with length 2 (3,2).
```

Constraints:

- The number of nodes in the tree is in the range `[1, 3000]`.
- `-100 <= Node.val <= 100`

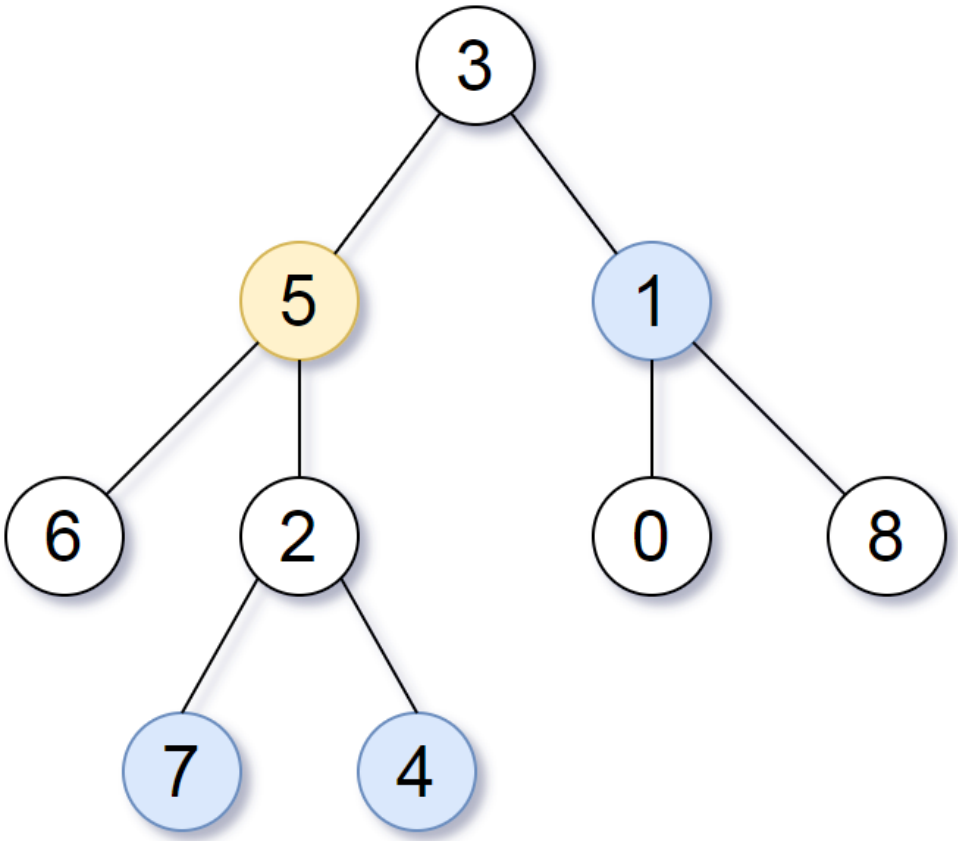
```
from collections import deque
class Solution:
    def widthOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        if not root:return 0
        que = deque([[root,1]])
        max_width = 1
        while que:
            for _ in range(len(que)):
                node,ind  = que.popleft()
                if node.left:
                    que.append([node.left,2*ind])
                if node.right:
                    que.append([node.right,2*ind+1])
            if len(que) >1:
                first = que[0][1]
                last = que[-1][1]
                max_width = max(max_width,last-first+1)
        return max_width
```

863. All Nodes Distance K in Binary Tree

Given the `root` of a binary tree, the value of a target node `target`, and an integer `k`, return *an array of the values of all nodes that have a distance* `k` *from the target node.*

You can return the answer in **any order**.

Example 1:




```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, k = 2
Output: [7,4,1]
Explanation: The nodes that are a distance 2 from the target node (with value 5) have
```

Example 2:

```
Input: root = [1], target = 1, k = 3
Output: []
```

Constraints:

- The number of nodes in the tree is in the range `[1, 500]`.

- `0 <= Node.val <= 500`
- All the values `Node.val` are **unique**.
- `target` is the value of one of the nodes in the tree.
- `0 <= k <= 1000`

```
from collections import defaultdict
class Solution:
    def distanceK(self, node: TreeNode, target: TreeNode, k: int) -> List[int]:
        adj = defaultdict(list)
        def dfs(root):
            if not root:return
            if root.left:
                adj[root].append(root.left)
                adj[root.left].append(root)
                dfs(root.left)
            if root.right:
                adj[root].append(root.right)
                adj[root.right].append(root)
                dfs(root.right)
        dfs(node)

        que = deque([[target,0]])
        vis = set()
        res = []
        while que:
            node,level = que.popleft()
            vis.add(node)
            if level == k:
                res.append(node.val)
                continue
            for neigh in adj[node]:
                if neigh not in vis:
                    que.append([neigh,level+1])
        return res
```

987. Vertical Order Traversal of a Binary Tree

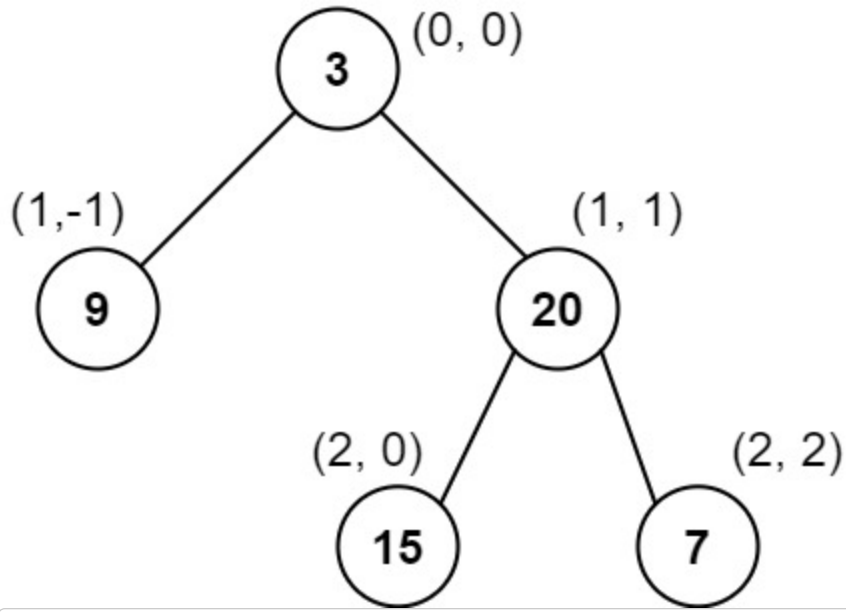
Given the `root` of a binary tree, calculate the **vertical order traversal** of the binary tree.

For each node at position `(row, col)`, its left and right children will be at positions `(row + 1, col - 1)` and `(row + 1, col + 1)` respectively. The root of the tree is at `(0, 0)`.

The **vertical order traversal** of a binary tree is a list of top-to-bottom orderings for each column index starting from the leftmost column and ending on the rightmost column. There may be multiple nodes in the same row and same column. In such a case, sort these nodes by their values.

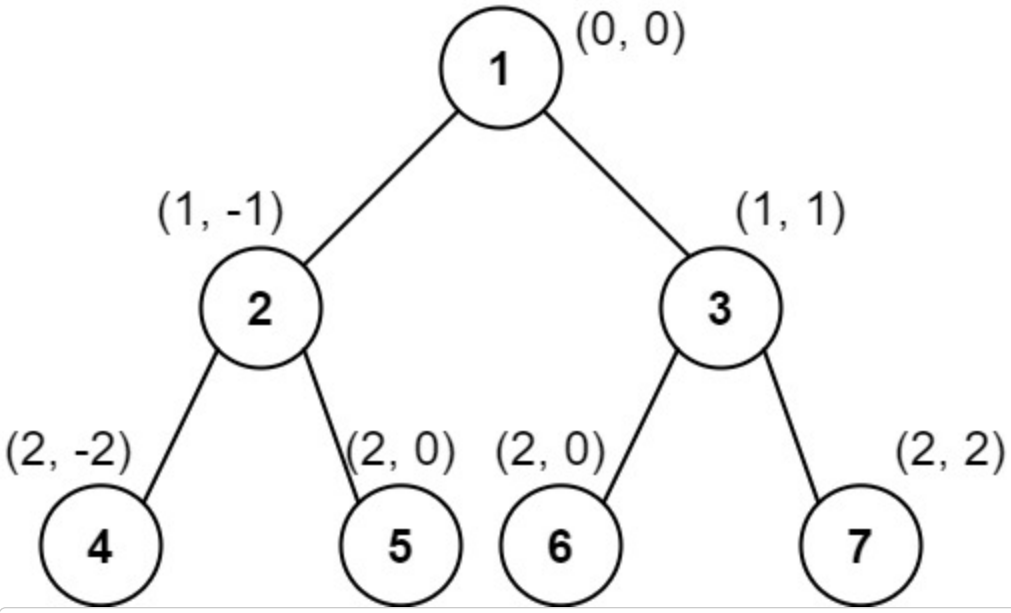
Return *the **vertical order traversal** of the binary tree*.

Example 1:



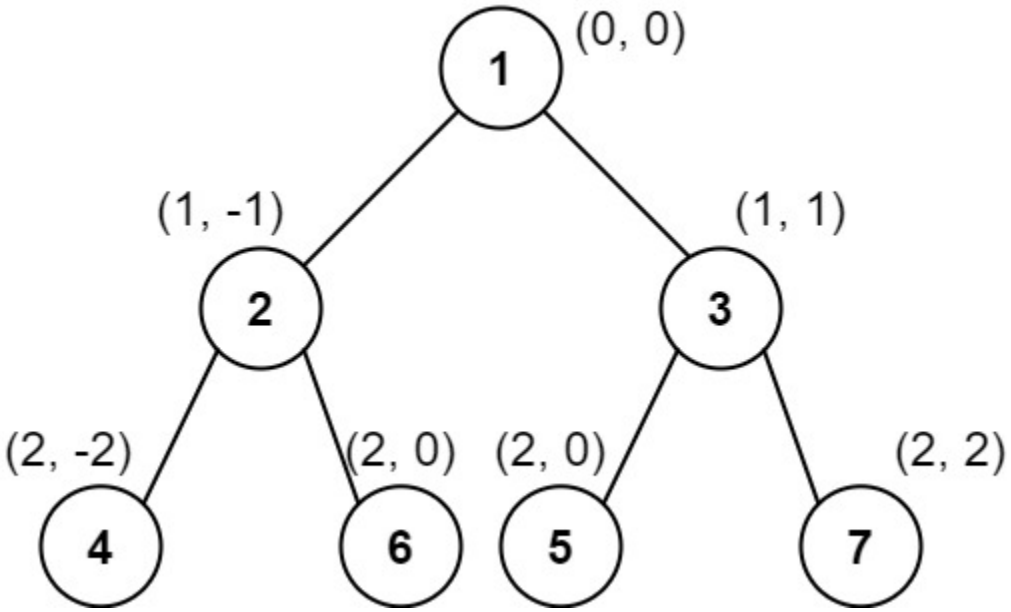

```
Input: root = [3,9,20,null,null,15,7]
Output: [[9],[3,15],[20],[7]]
Explanation:
Column -1: Only node 9 is in this column.
Column 0: Nodes 3 and 15 are in this column in that order from top to bottom.
Column 1: Only node 20 is in this column.
Column 2: Only node 7 is in this column.
```

Example 2:




```
Input: root = [1,2,3,4,5,6,7]
Output: [[4],[2],[1,5,6],[3],[7]]
Explanation:
Column -2: Only node 4 is in this column.
Column -1: Only node 2 is in this column.
Column 0: Nodes 1, 5, and 6 are in this column.
          1 is at the top, so it comes first.
          5 and 6 are at the same position (2, 0), so we order them by their value, 5
Column 1: Only node 3 is in this column.
Column 2: Only node 7 is in this column.
```

Example 3:



```
Input: root = [1,2,3,4,6,5,7]
Output: [[4],[2],[1,5,6],[3],[7]]
Explanation:
This case is the exact same as example 2, but with nodes 5 and 6 swapped.
Note that the solution remains the same since 5 and 6 are in the same location and sh
```

Constraints:

- The number of nodes in the tree is in the range `[1, 1000]` .
- `0 <= Node.val <= 1000`

```
from collections import defaultdict,deque
class Solution:
    def verticalTraversal(self, root: Optional[TreeNode]) -> List[List[int]]:
        column_table = defaultdict(list)
        que = deque([(root,0,0)])
        res= []
        while que:
            node,col,row = que.popleft()
            column_table[col].append([row,node.val])
            if node.left:
                que.append([node.left,col-1,row+1])
            if node.right:
                que.append([node.right,col+1,row+1])

        for col in sorted(column_table.keys()):
            row_node = sorted(column_table[col],key = lambda x:(x[0],x[1]))
            temp_ans = [node_val for _,node_val in row_node]
            res.append(temp_ans)
        return res
```
