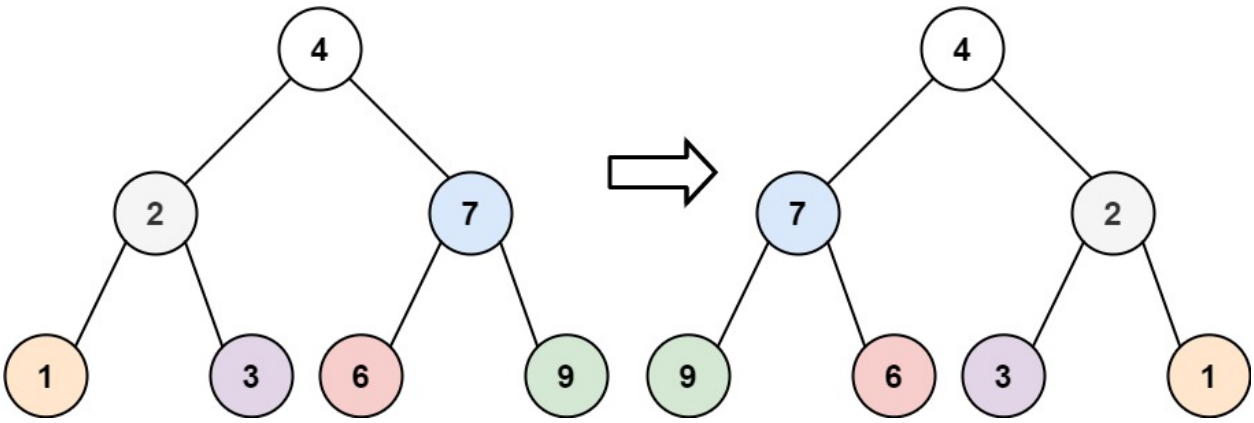
226. Invert Binary Tree

Created	@September 1, 2022 10:29 PM
Difficulty	Easy
LC Url	https://leetcode.com/problems/invert-binary-tree/
Importance	
Tag	DFS NEET Recursion Tree
Video	

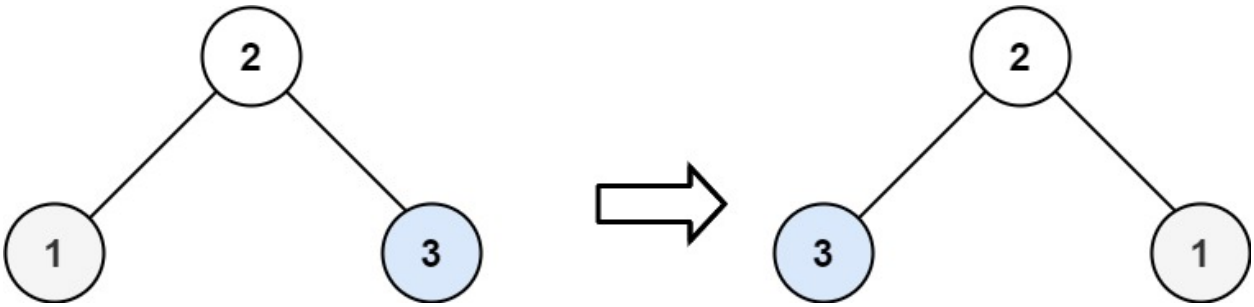
Given the `root` of a binary tree, invert the tree, and return *its root*.

Example 1:




```
Input: root = [4,2,7,1,3,6,9]
Output: [4,7,2,9,6,3,1]
```

Example 2:




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

Example 3:

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

Constraints:

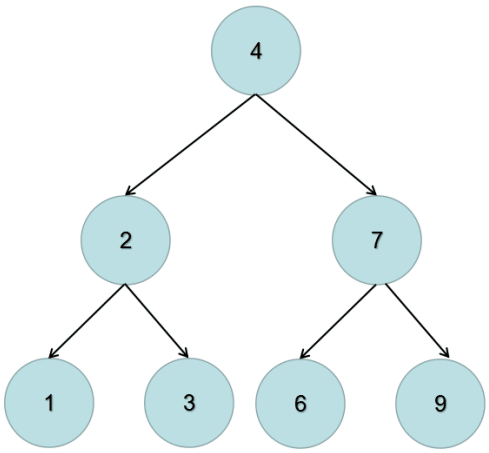
- The number of nodes in the tree is in the range `[0, 100]`.
- `100 <= Node.val <= 100`

Solution

递归-DFS

- 终止条件：当前节点为 `null` 时返回
- 交换当前节点的左右节点，再递归的交换当前节点的左节点，递归的交换当前节点的右节点

时间复杂度：每个元素都必须访问一次，所以是 $O(n)$
空间复杂度：最坏的情况下，需要存放 $O(h)$ 个函数调用(h是树的高度)，所以是 $O(h)$
代码实现如下：




```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root:
            return None

        root.left, root.right = root.right, root.left
        self.invertTree(root.left)
        self.invertTree(root.right)
        return root
```

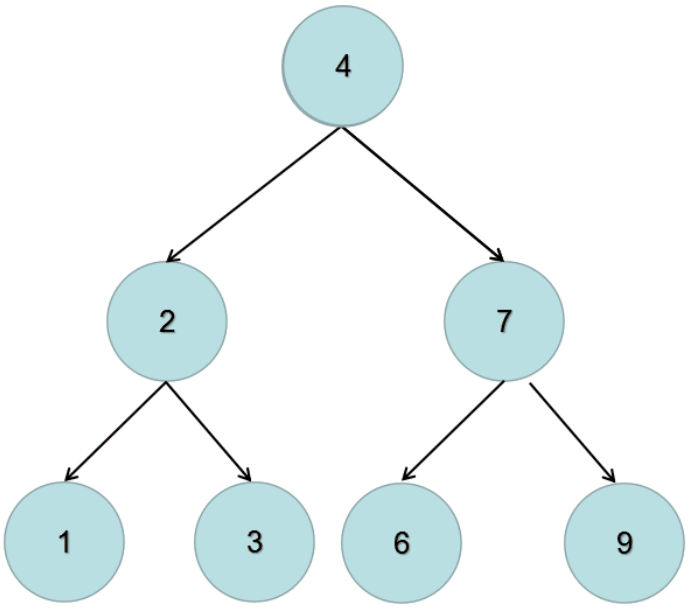
层序遍历-BFS

递归实现也就是深度优先遍历的方式，那么对应的就是广度优先遍历。
广度优先遍历需要额外的数据结构--队列，来存放临时遍历到的元素。
深度优先遍历的特点是一竿子插到底，不行了再退回来继续；而广度优先遍历的特点是层层扫荡。
所以，我们需要先将根节点放入到队列中，然后不断的迭代队列中的元素。
对当前元素调换其左右子树的位置，然后：

- 判断其左子树是否为空，不为空就放入队列中
- 判断其右子树是否为空，不为空就放入队列中

时间复杂度：同样每个节点都需要入队列/出队列一次，所以是 $O(n)$

空间复杂度：最坏的情况下会包含所有的叶子节点，完全二叉树叶子节点是 `n/2` 个，所以时间复杂度是 $0(n)$




```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root:
            return None

        queue = [root]
        while queue:
            tmp = queue.pop(0)
            tmp.left, tmp.right = tmp.right, tmp.left
            if tmp.left:
                queue.append(tmp.left)
            if tmp.right:
```

```
                queue.append(tmp.right)
        return root
```

力扣

https://leetcode.cn/problems/invert-binary-tree/solution/dong-hua-yan-shi-liang-chong-shi-xian-226-fan-zhua/

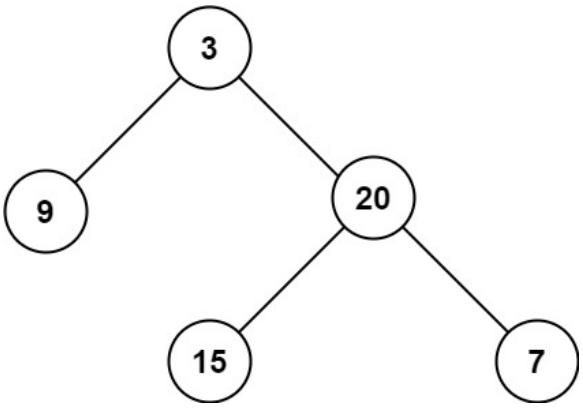
104. Maximum Depth of Binary Tree

Created	@September 10, 2022 7:52 PM
Difficulty	Easy
LC Url	https://leetcode.com/problems/maximum-depth-of-binary-tree/
Importance	
Tag	DFS NEET Recursion
Video	https://maxming0.github.io/2020/12/01/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`

Solution

递归：深度为左右子树最大深度+1

```
class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        return max(self.maxDepth(root.left), self.maxDepth(root.right)) + 1 if root else 0
```

BFS

非递归：从根开始bfs，每做一层，结果+1

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0

        queue = [root]
        depth = 0

        while queue:
            queue_temp = []
            for node in queue:
                if node.left:
                    queue_temp.append(node.left)
                if node.right:
                    queue_temp.append(node.right)
```

```
            queue = queue_temp
            depth += 1
        return depth
```

543. Diameter of Binary Tree

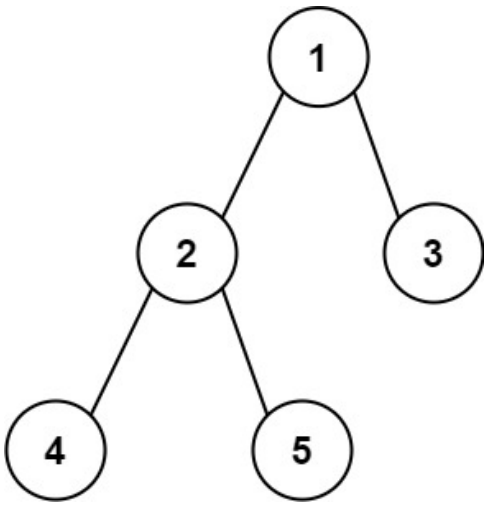
Created	@August 28, 2022 9:25 PM
Difficulty	Easy
LC Url	https://leetcode.com/problems/diameter-of-binary-tree/
Importance	
Tag	DFS NEET Tree
Video	【HOT 100】9.二叉树的直径 Python3 递归也需要看清题意 - 二叉树的直径 - 力扣（LeetCode）

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`

Solution

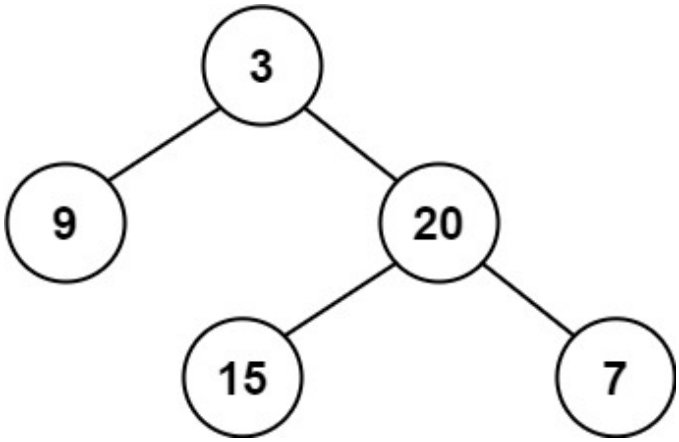
```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    res = 0

    def diameterOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0

        self.dfs(root)

        return self.res

    def dfs(self, root):
        if not root:
            return 0
        left_depth = self.dfs(root.left)
        right_depth = self.dfs(root.right)
        # 将每个节点最大直径(左子树深度+右子树深度)当前最大值比较并取大者
        self.res = max(self.res, left_depth + right_depth)
        # 返回该节点为根的子树的深度
        return max(left_depth, right_depth) + 1
```

110. Balanced Binary Tree

Created	@November 26, 2022 10:52 AM
Difficulty	Easy
LC Url	https://leetcode.com/problems/balanced-binary-tree/
Importance	
Tag	NEET Tree
Video	

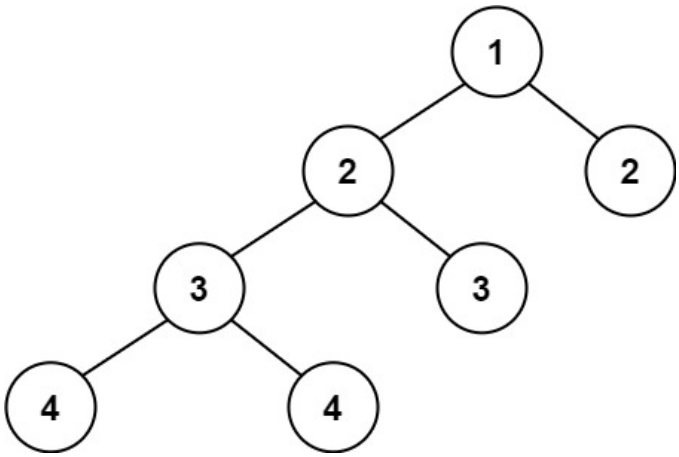
Given a binary tree, determine if it is **height-balanced**.

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`

Solution

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    is_balanced = True

    def isBalanced(self, root: Optional[TreeNode]) -> bool:
        self.maxDepth(root)
        return self.is_balanced
```

```
    def maxDepth(self, root):
        if not root:
            return 0

        left_depth = self.maxDepth(root.left)
        right_depth = self.maxDepth(root.right)

        if abs(right_depth - left_depth) > 1:
            self.is_balanced = False

        return max(left_depth, right_depth) + 1
```

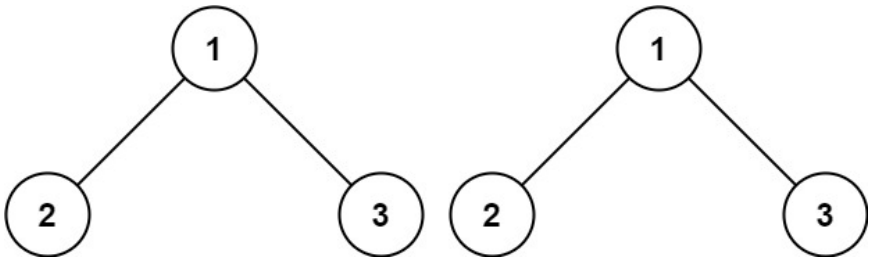
100. Same Tree

Created	@November 26, 2022 11:13 AM
Difficulty	Easy
LC Url	https://leetcode.com/problems/same-tree/
Importance	
Tag	DFS NEET Tree
Video	

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

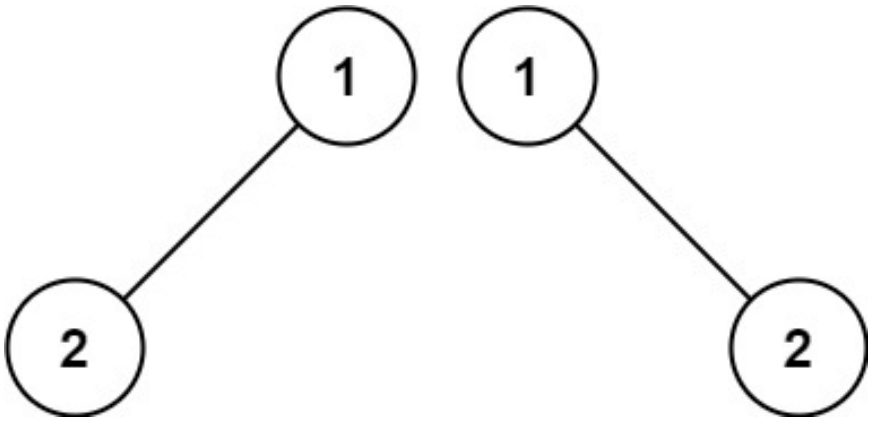
Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

Example 1:



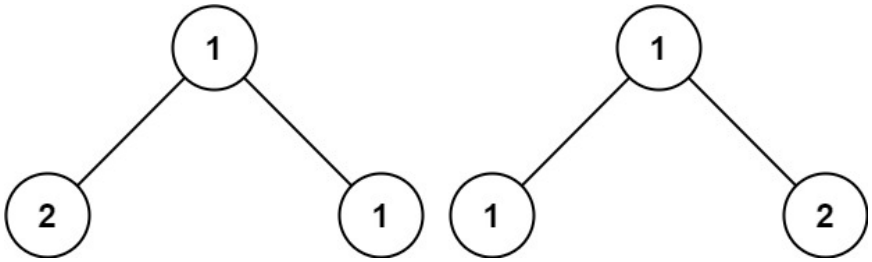

```
Input: p = [1,2,3], q = [1,2,3]
Output: true
```

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`

Solution

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
```

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

写的非常好：写树算法的套路框架 - 相同的树 - 力扣（LeetCode）

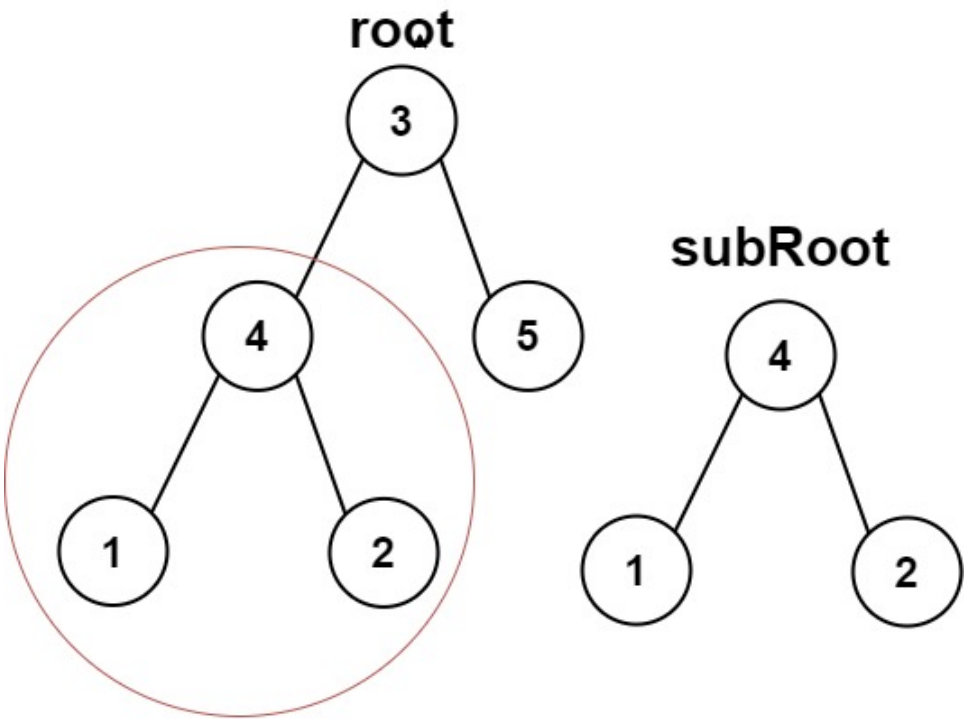
572. Subtree of Another Tree

Created	@November 26, 2022 1:22 PM
Difficulty	Easy
LC Url	https://leetcode.com/problems/subtree-of-another-tree/
Importance	
Tag	NEET Tree
Video	

Given the roots of two binary trees `root` and `subRoot`, return `true` if there is a subtree of `root` with the same structure and node values of `subRoot` and `false` otherwise.

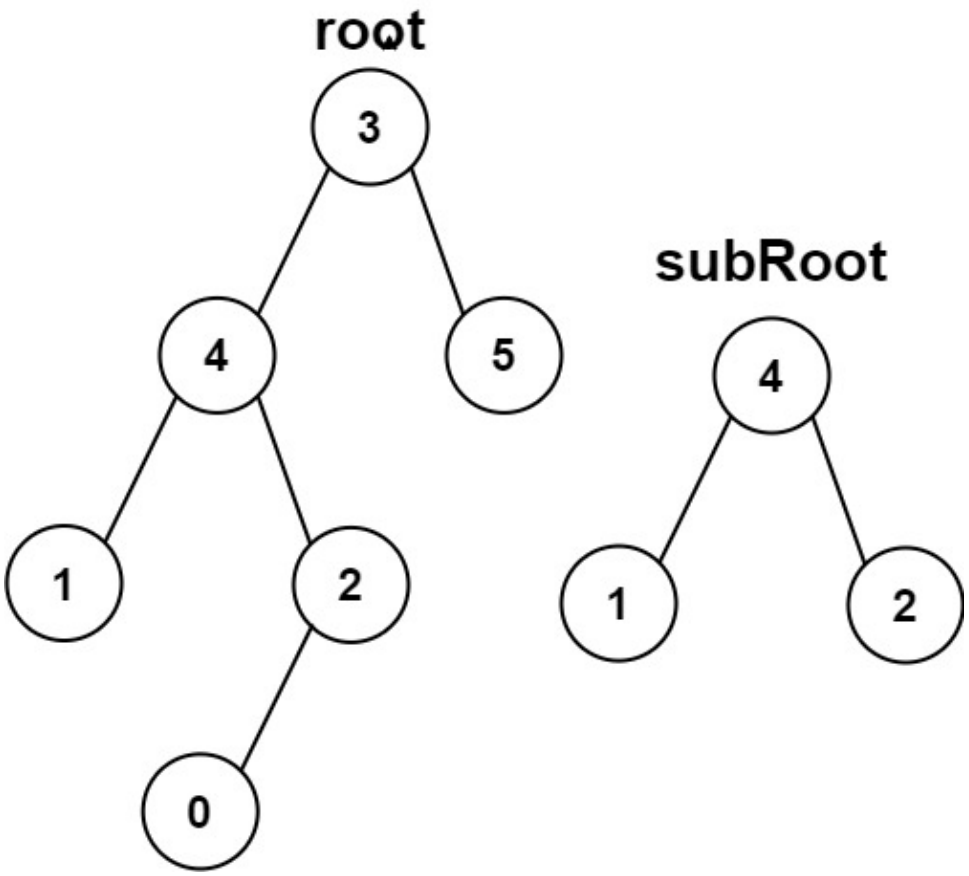
A subtree of a binary tree `tree` is a tree that consists of a node in `tree` and all of this node's descendants. The tree `tree` could also be considered as a subtree of itself.

Example 1:



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

Example 2:




```
Input: root = [3,4,5,1,2,null,null,null,null,0], subRoot = [4,1,2]
Output: false
```

Constraints:

- The number of nodes in the `root` tree is in the range `[1, 2000]`.
- The number of nodes in the `subRoot` tree is in the range `[1, 1000]`.
- `10 4 <= root.val <= 10 4`
- `10 4 <= subRoot.val <= 10 4`

Solution

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「遍历」的思维。

遍历以 `root` 为根的这棵二叉树的所有节点，用 100. 相同的树 中的 `isSame` 函数判断以该节点为根的子树是否和以 `subRoot` 为根的那棵树相同。

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def isSubtree(self, root: Optional[TreeNode], subRoot: Optional[TreeNode]) -> bool:
        if not root:
            return subRoot == None

        # 判断以root为根的二叉树是否与subRoot相同
        if self.isSameTree(root, subRoot):
            return True

        # 去左右子树中判断是否有和subRoot相同的子树
        return self.isSubtree(root.left, subRoot) or self.isSubtree(root.right, subRoot)

    def isSameTree(self, 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 self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)
```

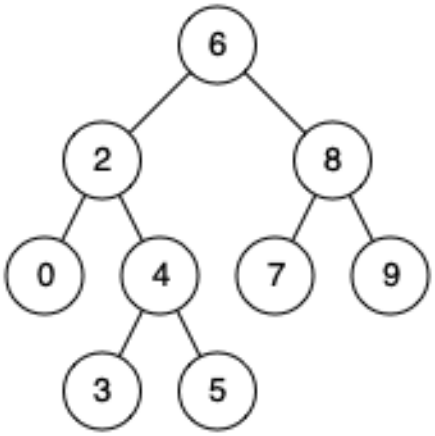
235. Lowest Common Ancestor of a Binary Search Tree

Created	@September 9, 2022 1:15 PM
Difficulty	Medium
LC Url	https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-search-tree/
Importance	****
Tag	NEET Recursion Tree
Video	

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

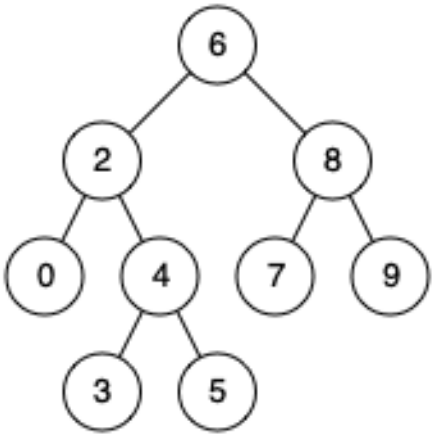
According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes `p` and `q` as the lowest node in `T` that has both `p` and `q` as descendants (where we allow **a node to be a descendant of itself**).”

Example 1:




```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
Explanation: The LCA of nodes 2 and 8 is 6.
```

Example 2:




```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
Output: 2
Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.
```

Example 3:

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

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 BST.

Solution

如果在 BST 中寻找最近公共祖先，反而容易很多，主要利用 BST 左小右大（左子树所有节点都比当前节点小，右子树所有节点都比当前节点大）的特点即可。

1、如果 `p` 和 `q` 都比当前节点小，那么显然 `p` 和 `q` 都在左子树，那么 LCA 在左子树。

2、如果 `p` 和 `q` 都比当前节点大，那么显然 `p` 和 `q` 都在右子树，那么 LCA 在右子树。

3、一旦发现 `p` 和 `q` 在当前节点的两侧，说明当前节点就是 LCA。

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        if not root:
            return None

        # 保证p.val < q.val
        if p.val > q.val:
```

```
            return self.lowestCommonAncestor(root, q, p)

        # p和q分别在root的左右子树，那么root就是LCA
        if p.val <= root.val <= q.val:
            return root

        if root.val > q.val:
            return self.lowestCommonAncestor(root.left, p, q)
        else:
            return self.lowestCommonAncestor(root.right, p, q)
```

方法一：两次遍历

注意到题目中给出的是一棵「二叉搜索树」，因此我们可以快速地找出树中的某个节点以及从根节点到该节点的路径，例如我们需要找到节点 p：

- 我们从根节点开始遍历；
- 如果当前节点就是 p，那么成功地找到了节点；
- 如果当前节点的值大于 p 的值，说明 p 应该在当前节点的左子树，因此将当前节点移动到它的左子节点；
- 如果当前节点的值小于 p 的值，说明 p 应该在当前节点的右子树，因此将当前节点移动到它的右子节点。

对于节点 q 同理。在寻找节点的过程中，我们可以顺便记录经过的节点，这样就得到了从根节点到被寻找节点的路径。

当我们分别得到了从根节点到 p 和 q 的路径之后，我们就可以很方便地找到它们的最近公共祖先了。显然，p 和 q 的最近公共祖先就是从根节点到它们路径上的「分岔点」，也就是最后一个相同的节点。因此，如果我们设从根节点到 p 的路径为数组 $path\_p$，从根节点到 q 的路径为数组 $path\_q$，那么只要找出最大的编号 i，其满足

$$path\_p[i] = path\_q[i]$$

那么对应的节点就是「分岔点」，即 p 和 q 的最近公共祖先就是 $path\_p[i]$（或 $path\_q[i]$）。

代码

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        def getPath(root, target):
            path = []
            node = root
            while node != target:
                path.append(node)
                if target.val < node.val:
                    node = node.left
                else:
                    node = node.right
            path.append(node)
            return path

        path_p = getPath(root, p)
        path_q = getPath(root, q)
```

```
        ancestor = None
        for u, v in zip(path_p, path_q):
            if u == v:
                ancestor = u
            else:
                break
        return ancestor

        # 链接：https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-search-tree/solution/er-cha-sou-suo-shu-de-zui-jin-gong-gong
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是给定的二叉搜索树中的节点个数。上述代码需要的时间与节点 p 和 q 在树中的深度线性相关，而在最坏的情况下，树呈现链式结构，p 和 q 一个是树的唯一叶子结点，一个是该叶子结点的父节点，此时时间复杂度为 $\Theta(n)$。
- 空间复杂度：$O(n)$，我们需要存储根节点到 p 和 q 的路径。和上面的分析方法相同，在最坏的情况下，路径的长度为 $\Theta(n)$，因此需要 $\Theta(n)$ 的空间。

方法二：一次遍历

在方法一中，我们对从根节点开始，通过遍历找出到达节点 p 和 q 的路径，一共需要两次遍历。我们也可以考虑将这两个节点放在一起遍历。

整体的遍历过程与方法一中的类似：

- 我们从根节点开始遍历；
- 如果当前节点的值大于 p 和 q 的值，说明 p 和 q 应该在当前节点的左子树，因此将当前节点移动到它的左子节点；
- 如果当前节点的值小于 p 和 q 的值，说明 p 和 q 应该在当前节点的右子树，因此将当前节点移动到它的右子节点；
- 如果当前节点的值不满足上述两条要求，那么说明当前节点就是「分岔点」。此时，p 和 q 要么在当前节点的不同的子树中，要么其中一个就是当前节点。

可以发现，如果我们将这两个节点放在一起遍历，我们就省去了存储路径需要的空间。

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        ancestor = root
        while True:
            if p.val < ancestor.val and q.val < ancestor.val:
                ancestor = ancestor.left
            elif p.val > ancestor.val and q.val > ancestor.val:
                ancestor = ancestor.right
            else:
                break
        return ancestor
```

复杂度分析

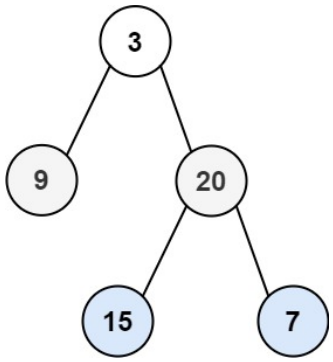
- 时间复杂度：$O(n)$，其中 n 是给定的二叉搜索树中的节点个数。分析思路与方法一相同。
- 空间复杂度：$O(1)$。

102. Binary Tree Level Order Traversal

Created	@October 28, 2021 9:54 PM
Difficulty	Medium
LC Url	https://leetcode.com/problems/binary-tree-level-order-traversal/
Importance	****
Tag	BFS NEET Queue Tree
Video	https://www.youtube.com/watch?v=MBZ-gBkjdMc

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:

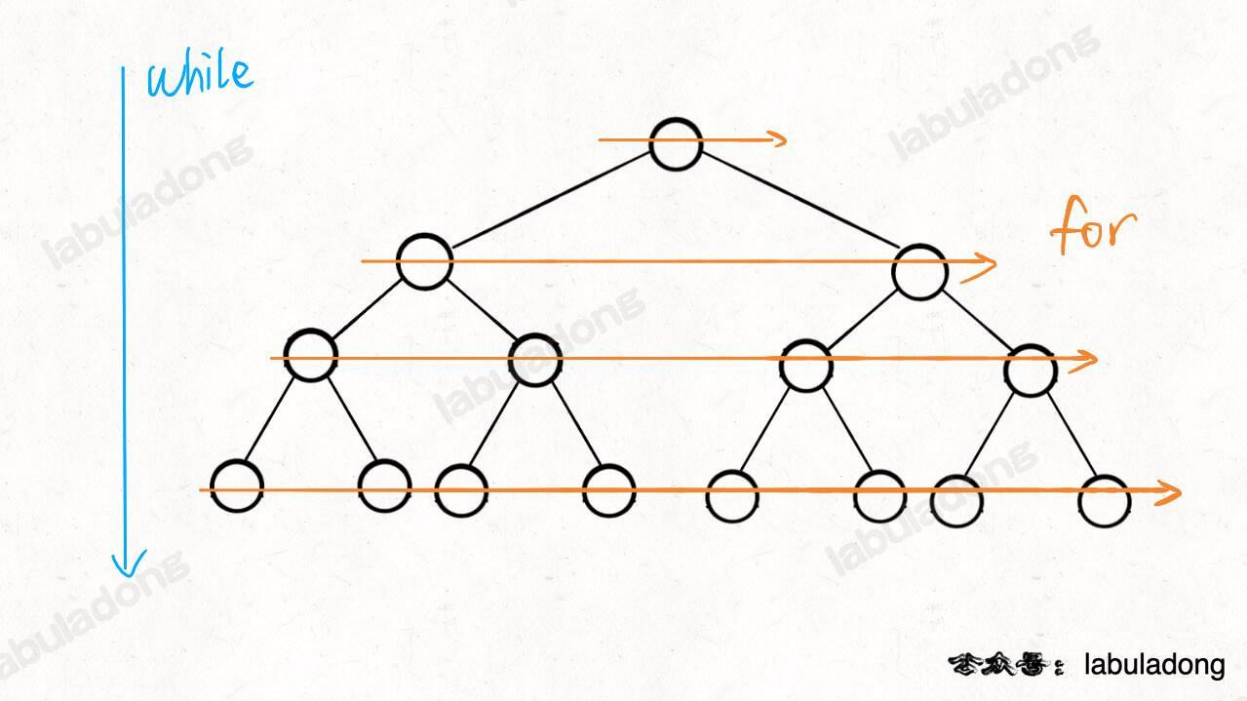
```
Input: root = []
Output: []
```

Constraints:

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

Solution

BFS




```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        # 作者：fuxuemingzhu
        # 链接：https://leetcode.cn/problems/binary-tree-level-order-traversal/solution/tao-mo-ban-bfs-he-dfs-du-ke-yi-jie-jue-by-fuxuemin/
        queue = collections.deque()
        queue.append(root)
        res = []
        while queue:
            size = len(queue)
            level = []
            for _ in range(size):
                cur = queue.popleft()
                if not cur:
                    continue
                level.append(cur.val)
                queue.append(cur.left)
                queue.append(cur.right)
            if level:
                res.append(level)
        return res
```

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        if root is None:
            return []

        queue = [root]
        next_queue = []
        level = []
        result = []
```

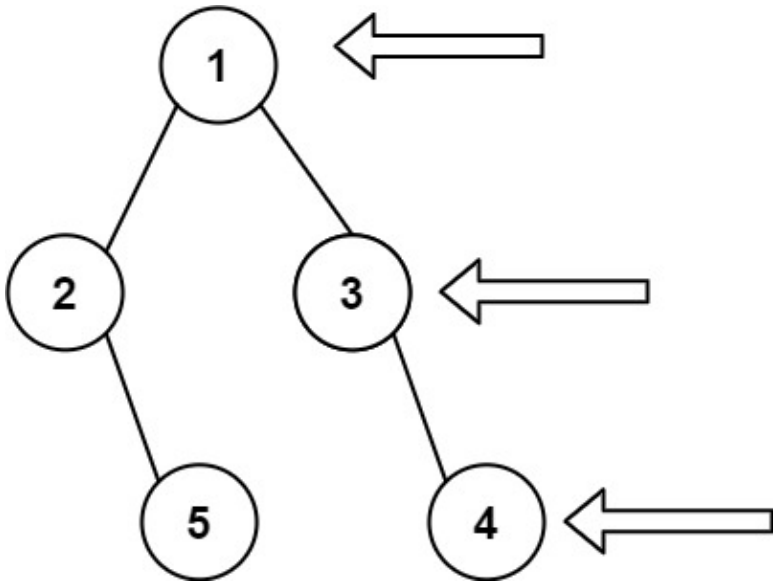
```
        while queue != []:
            for root in queue:
                level.append(root.val)
                if root.left is not None:
                    next_queue.append(root.left)
                if root.right is not None:
                    next_queue.append(root.right)
            result.append(level)
            level = []
            queue = next_queue
            next_queue = []
        return result
```

199. Binary Tree Right Side View

Created	@November 26, 2022 3:59 PM
Difficulty	Medium
LC Url	https://leetcode.com/problems/binary-tree-right-side-view/
Importance	
Tag	NEET Tree
Video	https://leetcode.cn/problems/binary-tree-right-side-view/solution/python3-gui-na-liao-san-dao-ti-yi-ci-gao-jx9u/

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]
```

Example 2:

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

Example 3:

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

Constraints:

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

Solution

BFS

相当于记录每一层的最后一个节点，可用队列实现层序遍历，在内层循环时判断是否为当前层最后一个节点，若是则记录该节点值到res，遍历完成后返回res即可。

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return []

        res = []
        queue = collections.deque([root])
        while queue:
            sz = len(queue)
            for i in range(sz):
                cur = queue.popleft()
                if i == sz - 1:
                    res.append(cur.val)
```

```
                if cur.left:
                    queue.append(cur.left)
                if cur.right:
                    queue.append(cur.right)
        return res
```

右视图即每层最右边的值，因此只需要做一个非常简单的小变化，即返回的列表只需要对每一层的列表的最后一个元素入结果列表就可以了。

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        if root is None:
            return []
        res = []
        queue = []
        queue.append(root)

        while queue:
            sz = len(queue)
            res.append(queue[-1].val)
            level = []
            for i in range(sz):
                cur = queue[i]
                if cur.left:
                    level.append(cur.left)
                if cur.right:
                    level.append(cur.right)
            queue = level
        return res
```

DFS

若使用深度优先遍历，先遍历右子树，则每次遍历到的第一个节点就是最右边的节点。使用深度depth来判断当前节点是否是该深度首次访问到的节点（depth==len(res)），每次递归更新depth，若是首次则加到结果res中，最后返回res。

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
```

```
#         self.left = left
#         self.right = right
class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        res = []

        def dfs(root, depth):
            if root is None:
                return

            if depth == len(res):
                res.append(root.val)
            depth += 1
            dfs(root.right, depth)
            dfs(root.left, depth)

        dfs(root, 0)
        return res
```

链接：https://leetcode.cn/problems/binary-tree-right-side-view/solution/er-cha-shu-de-you-shi-tu-bfs-dfs-by-huan-b6nh/

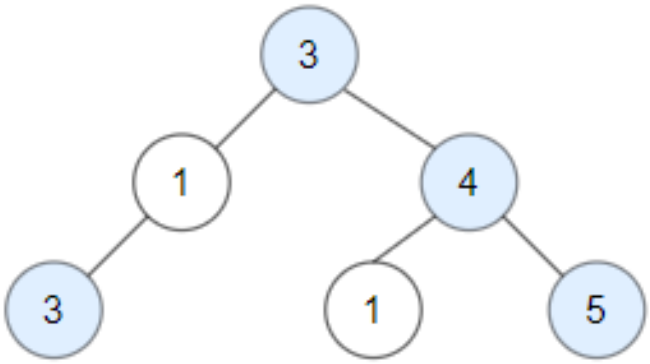
1448. Count Good Nodes in Binary Tree

Created	@November 27, 2022 9:26 AM
Difficulty	Medium
LC Url	https://leetcode.com/problems/count-good-nodes-in-binary-tree/
Importance	
Tag	NEET Tree
Video	

Given a binary tree `root`, a node *X* in the tree is named **good** if in the path from root to *X* there are no nodes with a value *greater than* X.

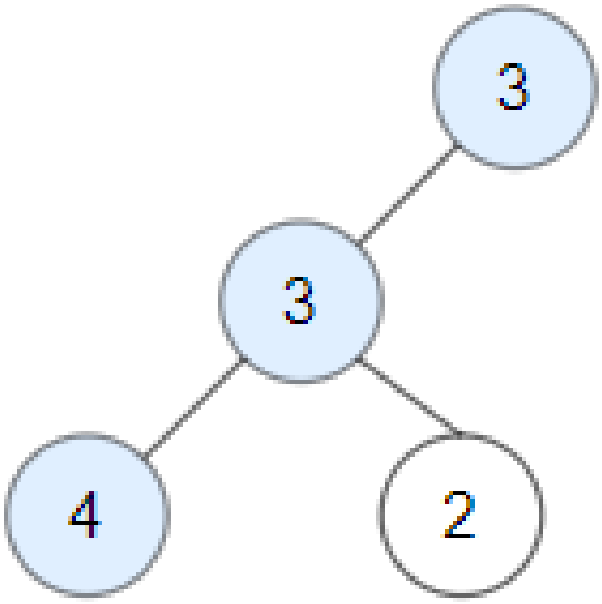
Return the number of **good** nodes in the binary tree.

Example 1:




```
Input: root = [3,1,4,3,null,1,5]
Output: 4
Explanation: Nodes in blue aregood.
Root Node (3) is always a good node.
Node 4 -> (3,4) is the maximum value in the path starting from the root.
Node 5 -> (3,4,5) is the maximum value in the path
Node 3 -> (3,1,3) is the maximum value in the path.
```

Example 2:




```
Input: root = [3,3,null,4,2]
Output: 3
Explanation: Node 2 -> (3, 3, 2) is not good, because "3" is higher than it.
```

Example 3:

```
Input: root = [1]
Output: 1
Explanation: Root is considered asgood.
```

Constraints:

- The number of nodes in the binary tree is in the range `[1, 10^5]`.
- Each node's value is between `[-10^4, 10^4]`.

Solution

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「遍历」的思维，利用函数参数给子树传递信息。

函数参数 `pathMax` 记录从根节点到当前节点路径中的最大值，通过比较 `root.val` 和 `pathMax` 比较就可判断 `root` 节点是不是「好节点」。

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    cnt = 0

    def goodNodes(self, root: TreeNode) -> int:
        self.traverse(root, root.val)
        return self.cnt

    def traverse(self, root, path_max):
        """
        二叉树遍历函数，pathMax 参数记录从根节点到当前节点路径中的最大值
        """
        if not root:
            return

        if path_max <= root.val:
            # 找到一个「好节点」
            self.cnt += 1

        # 更新路径上的最大值
        path_max = max(path_max, root.val)

        self.traverse(root.left, path_max)
        self.traverse(root.right, path_max)
```

Ref: labuladong

98. Validate Binary Search Tree

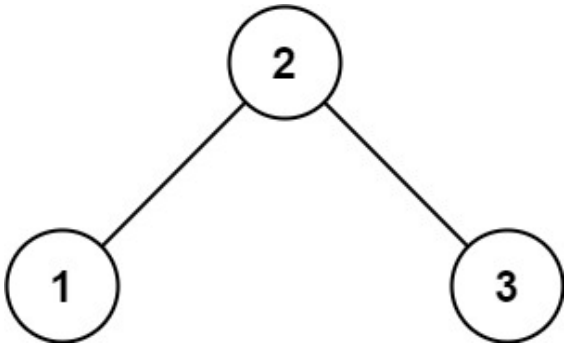
Created	@October 28, 2021 9:38 PM
Difficulty	Medium
LC Url	https://leetcode.com/problems/validate-binary-search-tree/
Importance	
Tag	BFS NEET Tree
Video	https://www.youtube.com/watch?v=ewrpOMA6LrY

Given the `root` of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

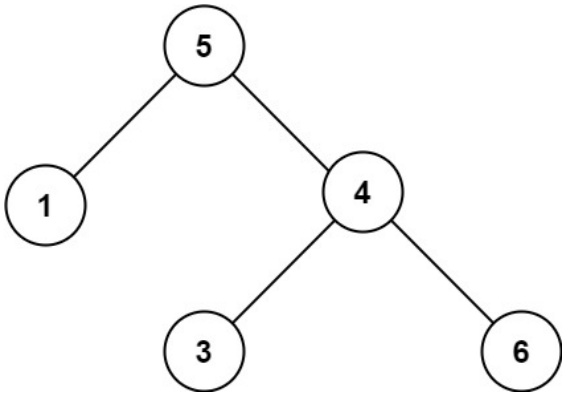
- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:




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

Example 2:




```
Input: root = [5,1,4,null,null,3,6]
Output: false
Explanation: The root node's value is 5 but its right child's value is 4.
```

Constraints:

- The number of nodes in the tree is in the range `[1, 10 4 ]`.
- `2 31  <= Node.val <= 2 31  - 1`

Solution

初学者做这题很容易有误区：BST 不是左小右大么，那我只要检查 `root.val > root.left.val` 且 `root.val < root.right.val` 不就行了？

这样是不对的，因为 BST 左小右大的特性是指 `root.val` 要比左子树的所有节点都更大，要比右子树的所有节点都小，你只检查左右两个子节点当然是不够的。

正确解法是通过使用辅助函数，增加函数参数列表，在参数中携带额外信息，将这种约束传递给子树的所有节点，这也是二叉搜索树算法的一个小技巧吧。

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        return self.check(root, None, None)

    # 限定以 root 为根的子树节点必须满足 max.val > root.val > min.val
    def check(self, root, min_node, max_node):
```

```
        if root is None:
            return True
        # 若 root.val 不符合 max 和 min 的限制，说明不是合法 BST
        if min_node and root.val <= min_node.val:
            return False
        if max_node and root.val >= max_node.val:
            return False
        # 限定左子树的最大值是 root.val，右子树的最小值是 root.val
        return self.check(root.left, min_node, root) and self.check(root.right, root, max_node)
```

使用DFS遍历树，遍历的时候向下传上下界，根据上下界判断是否符合要求

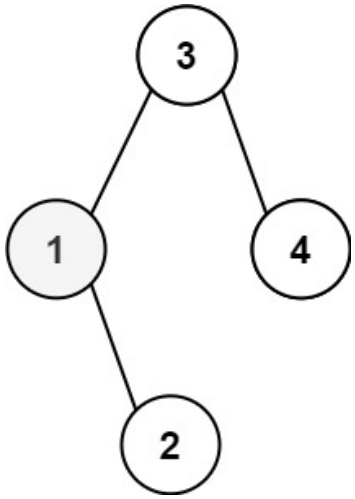
```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        def check(node, left, right):
            if not node:
                return True

            val = node.val

            if val <= left or val >= right:
                return False

            return check(node.left, left, val) and check(node.right, val, right)

        return check(root, float('-inf'), float('inf'))
```

230. Kth Smallest Element in a BST

Created	@September 18, 2022 3:34 PM
Difficulty	Medium
LC Url	https://leetcode.com/problems/kth-smallest-element-in-a-bst/
Importance	
Tag	BST NEET Recursion Tree
Video	https://www.youtube.com/watch?v=5LUXSvjmGCw

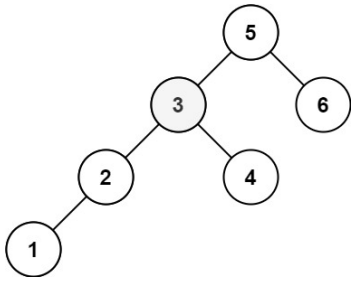
Given the `root` of a binary search tree, and an integer `k`, return *the* `k th` *smallest value (**1-indexed**) of all the values of the nodes in the tree*.

Example 1:




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

Example 2:




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

Constraints:

- The number of nodes in the tree is `n`.
- `1 <= k <= n <= 10 4`
- `0 <= Node.val <= 10 4`

Follow up: If the BST is modified often (i.e., we can do insert and delete operations) and you need to find the kth smallest frequently, how would you optimize?

Solution

迭代法

- 迭代法遍历二叉树，其实跟dfs的遍历是相通的
- 遍历过程中，先找到最小的值，此时pop出来，$k-=1$
- 当 $k=0$ 即可返回结果

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        stack = []
        while root or stack:
            while root:
                stack.append(root)
                root = root.left
            root = stack.pop()
            k -= 1
            if k == 0:
                return root.val
            root = root.right

        # 链接：https://leetcode.cn/problems/kth-smallest-element-in-a-bst/solution/er-cha-sou-suo-shu-zhong-di-kxiao-de-yua-8o07/
```

https://leetcode.cn/problems/kth-smallest-element-in-a-bst/solution/chi-xiao-dou-nojie-ti-python-dfszhong-xu-m5ql/

中序遍历

- 利用二叉搜索树的中序遍历为有序数组的特点
- 深度优先遍历，获得所有节点的值保存到数组中，此时数组是有序的，从小到大排列

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:

        def dfs(node, res):

            if node is None:
                return
```

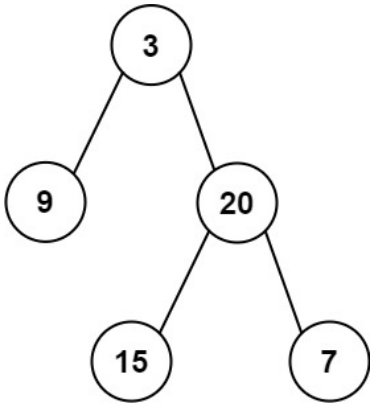
```
            dfs(node.left, res)
            res.append(node.val)
            dfs(node.right, res)

        res = []
        dfs(root, res)
        return res[k-1]
```

105. Construct Binary Tree from Preorder and Inorder Traversal

Created	@November 2, 2021 9:02 AM
Difficulty	Medium
LC Url	https://leetcode.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/
Importance	
Tag	BST NEET Tree
Video	https://www.youtube.com/watch?v=Qc0XLGFnchU, https://www.youtube.com/watch?v=S1wNG5hx-30

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.

Solution

递归

思路

对于任意一颗树而言，前序遍历的形式总是

```
[ 根节点, [左子树的前序遍历结果], [右子树的前序遍历结果] ]
```

即根节点总是前序遍历中的第一个节点。而中序遍历的形式总是

```
[ [左子树的中序遍历结果], 根节点, [右子树的中序遍历结果] ]
```

只要我们在中序遍历中**定位**到根节点，那么我们就可以分别知道左子树和右子树中的节点数目。由于同一颗子树的前序遍历和中序遍历的长度显然是相同的，因此我们就可以对应到前序遍历的结果中，对上述形式中的所有**左右括号**进行定位。

这样以来，我们就知道了左子树的前序遍历和中序遍历结果，以及右子树的前序遍历和中序遍历结果，我们就可以递归地对构造出左子树和右子树，再将这两颗子树接到根节点的左右位置。

细节

在中序遍历中对根节点进行定位时，一种简单的方法是直接扫描整个中序遍历的结果并找出根节点，但这样做的时间复杂度较高。我们可以考虑使用哈希表来帮助我们快速地定位根节点。对于哈希映射中的每个键值对，键表示一个元素（节点的值），值表示其在中序遍历中的出现位置。在构造二叉树的过程之前，我们可以对中序遍历的列表进行一遍扫描，就可以构造出这个哈希映射。在此后构造二叉树的过程中，我们就只需要 $O(1)$ 的时间对根节点进行定位了。

下面的代码给出了详细的注释。

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        if not preorder or not inorder:
            return None

        root_value = preorder[0]
        root = TreeNode(root_value)
        inorder_index = inorder.index(root_value)

        root.left = self.buildTree(preorder[1:inorder_index+1], inorder[:inorder_index])
```

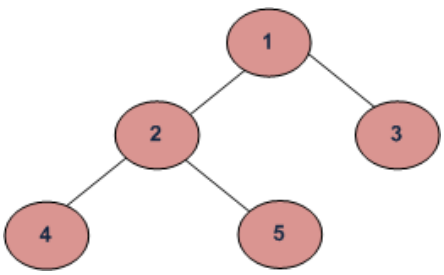
```
        root.right = self.buildTree(preorder[inorder_index+1:], inorder[inorder_index+1:])

        return root
```

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        if (preorder == null || inorder == null || preorder.length == 0 || preorder.length != inorder.length) return null;
        return buildTreeHelper(preorder, inorder, 0, 0, preorder.length-1);
    }

    public TreeNode buildTreeHelper(int[] preorder, int[] inorder, int pre_st, int in_st, int in_end) {
        if (pre_st > preorder.length || in_st > in_end) return null;
        TreeNode current = new TreeNode(preorder[pre_st]);
        int idx = in_st;
        while (idx < in_end) {
            if (inorder[idx] == preorder[pre_st]) break;
            idx++;
        }
        current.left = buildTreeHelper(preorder, inorder, pre_st+1, in_st, idx-1);
        current.right = buildTreeHelper(preorder, inorder, pre_st+(idx-in_st)+1, idx+1, in_end);

        return current;
    }
}
```

Tree Traversals (Inorder, Preorder and Postorder)

- Difficulty Level : **Easy**
- Last Updated : 21 Oct, 2021

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Attention reader! Don’t stop learning now. Get hold of all the important DSA concepts with the **DSA Self Paced Course** at a student-friendly price and become industry ready. To complete your preparation from learning a language to DS Algo and many more, please refer **Complete Interview Preparation Course.**

In case you wish to attend **live classes** with experts, please refer **DSA Live Classes for Working Professionals** and **Competitive Programming Live for Students**.

Depth First Traversals:

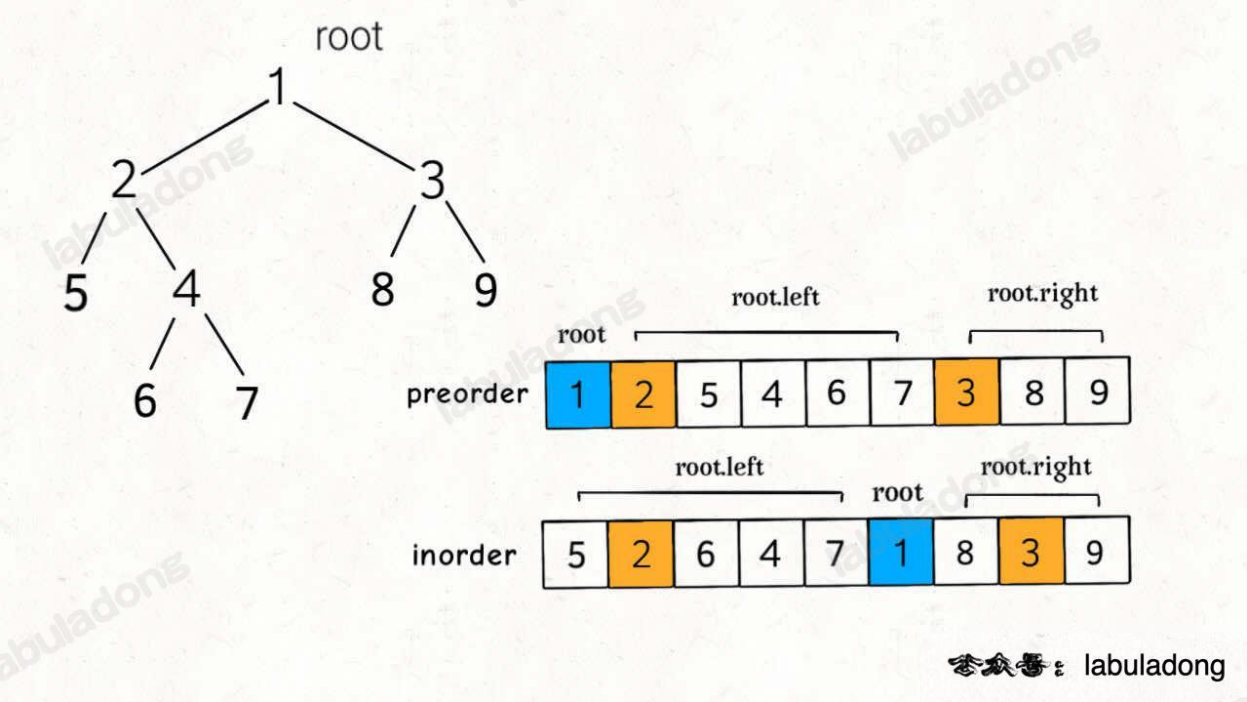
(a) Inorder (Left, Root, Right) : 4 2 5 1 3

(b) Preorder (Root, Left, Right) : 1 2 4 5 3

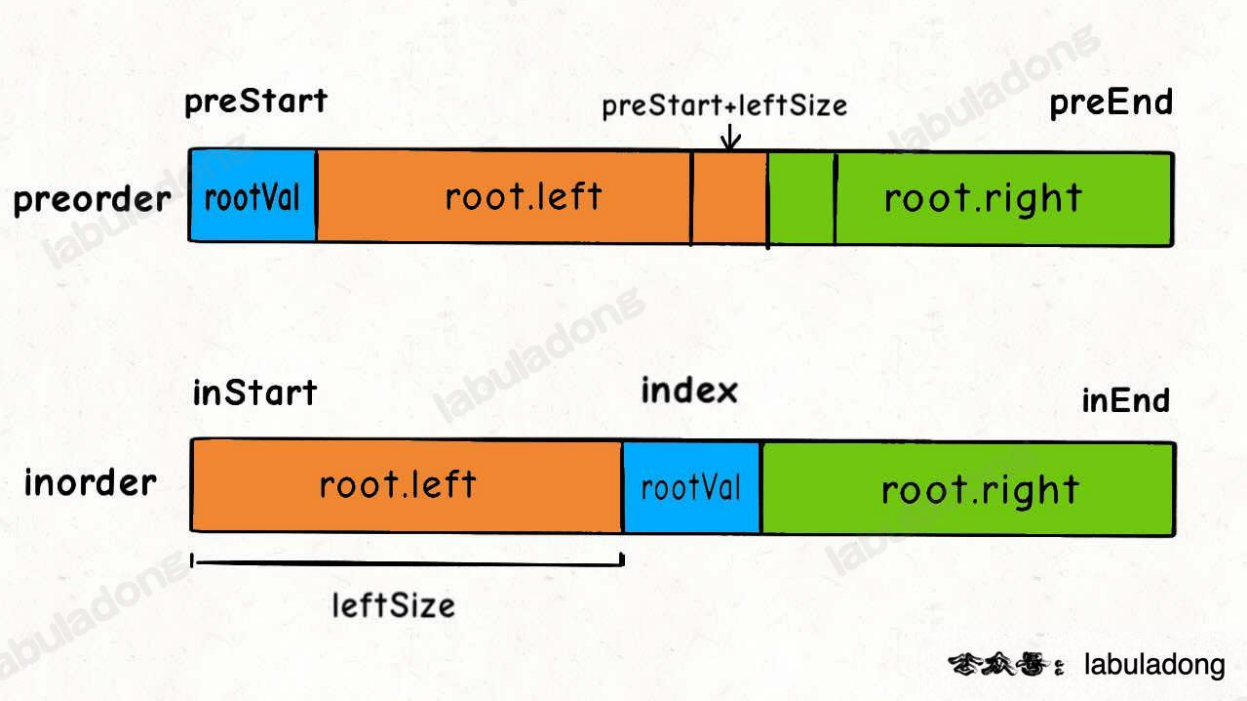
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

(d) Breadth-First or Level Order Traversal: 1 2 3 4 5

二叉树的前序和中序遍历结果的特点如下：



前序遍历结果第一个就是根节点的值，然后再根据中序遍历结果确定左右子树的节点。




```
class Solution {
    // 存储 inorder 中值到索引的映射
    HashMap<Integer, Integer> valToIndex = new HashMap<>();

    public TreeNode buildTree(int[] preorder, int[] inorder) {
        for (int i = 0; i < inorder.length; i++) {
            valToIndex.put(inorder[i], i);
        }
        return build(preorder, 0, preorder.length - 1,
                     inorder, 0, inorder.length - 1);
    }

    /*
       定义：前序遍历数组为 preorder[preStart..preEnd]，
       中序遍历数组为 inorder[inStart..inEnd]，
       构造这个二叉树并返回该二叉树的根节点
    */
    TreeNode build(int[] preorder, int preStart, int preEnd,
                   int[] inorder, int inStart, int inEnd) {
        if (preStart > preEnd) {
            return null;
        }

        // root 节点对应的值就是前序遍历数组的第一个元素
        int rootVal = preorder[preStart];
        // rootVal 在中序遍历数组中的索引
        int index = valToIndex.get(rootVal);

        int leftSize = index - inStart;

        // 先构造出当前根节点
        TreeNode root = new TreeNode(rootVal);
        // 递归构造左右子树
        root.left = build(preorder, preStart + 1, preStart + leftSize,
                inorder, inStart, index - 1);

        root.right = build(preorder, preStart + leftSize + 1, preEnd,
                inorder, index + 1, inEnd);
        return root;
```

```
    }
}
// 详细解析参见：
// https://labuladong.github.io/article/?qno=105
```