Linked List

Mustafa Muhammad

12th November 2021

1 Traversals

Pre, In and Post order traversals

```
class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        res = []
        def inorder(root):
            if root == None:
                return
            inorder(root.left)
            res.append(root.val)
            inorder(root.right)

    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        res = []
        def pre(root):
            if root == None:
                return
            res.append(root.val)
            pre(root.left)
            pre(root.right)
        pre(root)
        return res

    def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        res = []
        def pos(root):
            if root == None:
                return
            pos(root.left)
            pos(root.right)
            res.append(root.val)
        pos(root)
        return res
```

2 Level order Traversal (BFS)

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).

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

Output: [[3],[9,20],[15,7]]

```
class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        queue = []
        if root == None:
            return []
        queue.append(root)
        temp = []
        while len(queue) != 0:
            res = []
            for i in range(0, len(queue)):
                popped = queue.pop(0)
                if popped.left != None:
                    queue.append(popped.left)

                if popped.right != None:
                    queue.append(popped.right)

                res.append(popped.val)
            temp.append(res)
        return temp
```

3 Kth Smallest Element in a BST

```
class Solution:
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        num = [0]
        res = []
        def traverse(root):
            if root == None:
                return
            traverse(root.left)
            num[0]+=1
            if num[0] == k:
                res.append(root.val)
            traverse(root.right)
        traverse(root)
        return res[0]
```

4 Max Depth of BT

```
class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        if root == None:
            return 0

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

        return 1+max(left, right)
```

5 Invert Flip BT

```
class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if root == None:
            return None
        left = self.invertTree(root.left)
        right = self.invertTree(root.right)
        root.right = left
        root.left = right
        return root
```

6 Same Tree

```
class Solution:
    def isSameTree(self, p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
        if p != None and q == None:
            return False
        if p == None and q != None:
            return False
        if p == None and q == None:
            return True
        if p.val != q.val:
            return False

        l = self.isSameTree(p.left, q.left)
        r = self.isSameTree(p.right, q.right)

        return l and r
```

7 Subtree of another tree

Two functions: isSame and isSubtree. Recurse in isSame if current node values are matching else simply return false.

```
class Solution:
    def isSubtree(self, root: Optional[TreeNode], subRoot: Optional[TreeNode]) -> bool:
        def same_tree(s, t):
            if s == None and t == None:
                return True

            if s != None and t != None and s.val == t.val:
                return same_tree(s.left, t.left) and same_tree(s.right, t.right)

            return False

        if root == None:
            return False
        if subRoot == None:
            return True

        same = same_tree(root, subRoot)

        l = self.isSubtree(root.left, subRoot)
        r = self.isSubtree(root.right, subRoot)

        return same or l or r
```

8 Construct Binary Tree from Inorder and PreOrder Traversal

Note: Take special care about the mid points in the array splicing

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

Output: [3,9,20,null,null,15,7]

```
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        if len(preorder) == 0 or len(inorder) == 0:
            return None

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

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

        return root
```

9 Lowest Common Ancestor

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.

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> '
        cur = root

        while cur != None:
            if p.val < cur.val and q.val < cur.val:
                cur = cur.left
            elif p.val > cur.val and q.val > cur.val:
                cur = cur.right
            else:
                return cur

        return None
```

10 Validate BST

Very simple and elegant code

```
class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        def validate(root, left, right):
            if root == None:
                return True

            if not (left < root.val and root.val < right):
                return False

            l = validate(root.left, left, root.val)
            r = validate(root.right, root.val, right)

            return l and r
        return validate(root, float('-inf'), float('inf'))
```