94. Binary Tree Inorder Traversal

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

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

```
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

```
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

```
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

```
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

```
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
```