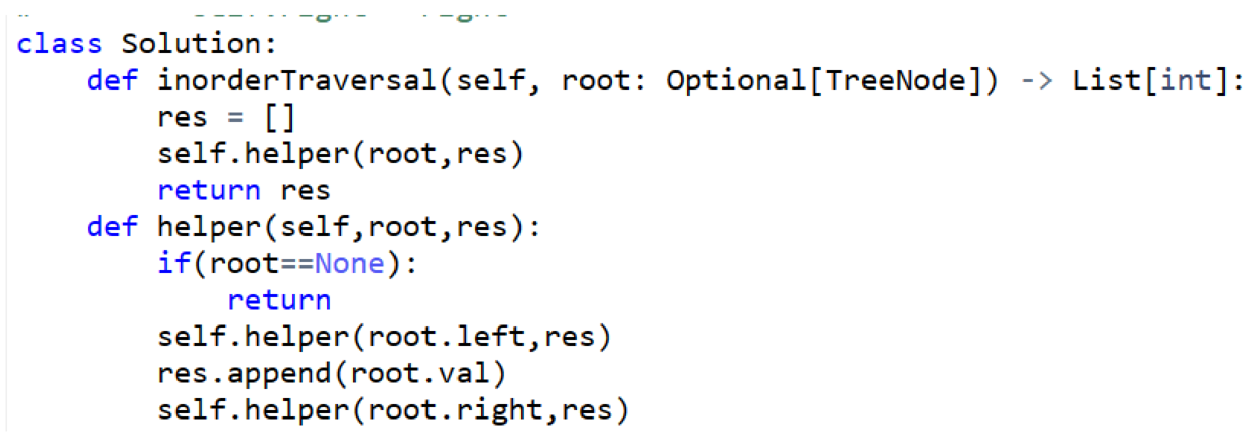
TREE

Tree traversal: BFS (LEVEL ORDER)
DFS (PREORDER, POSTORDER, INORDER)

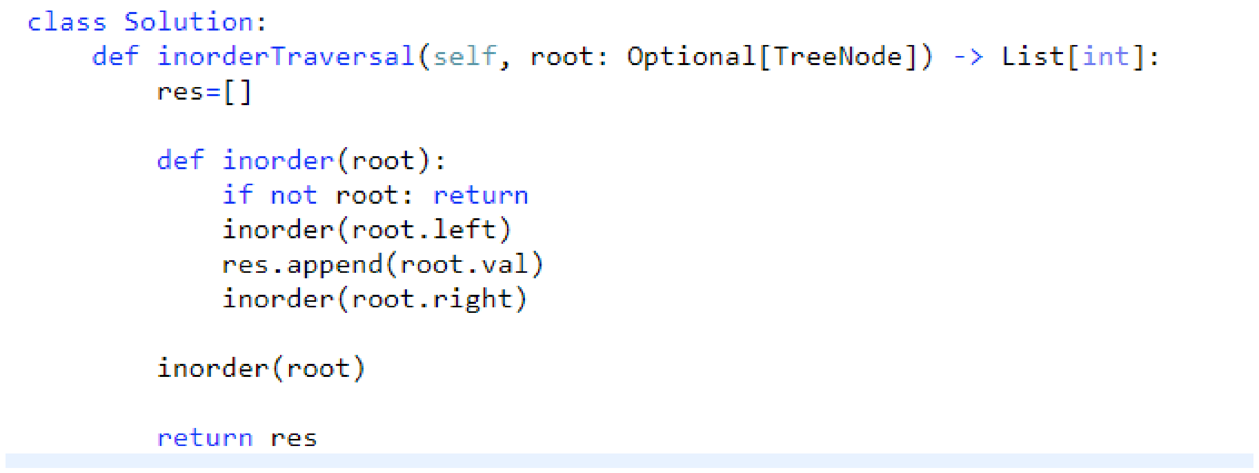
RECURSIVE SOLUTION:

- **INORDER: LEFT ROOT RIGHT**



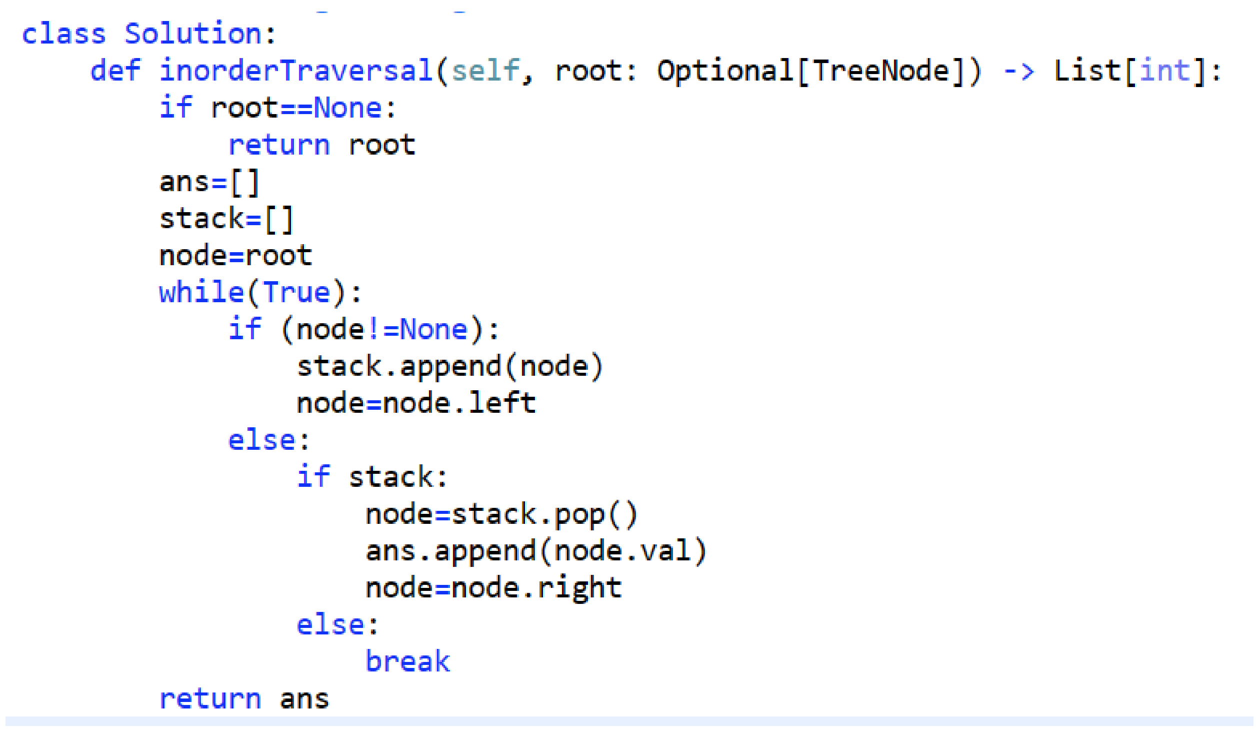
```python
class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        res = []
        self.helper(root,res)
        return res
    def helper(self,root,res):
        if(root==None):
            return
        self.helper(root.left,res)
        res.append(root.val)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

Another Soln :



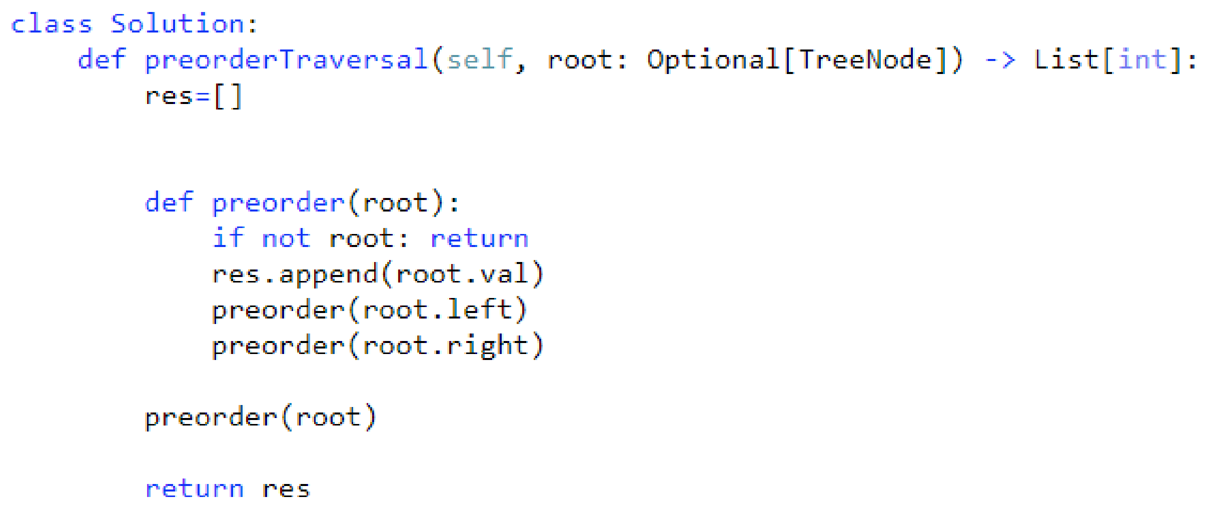
```python
class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        res=[]

        def inorder(root):
            if not root: return
            inorder(root.left)
            res.append(root.val)
            inorder(root.right)

        inorder(root)

        return res
```

ITERATIVE INORDER SOLUTION:



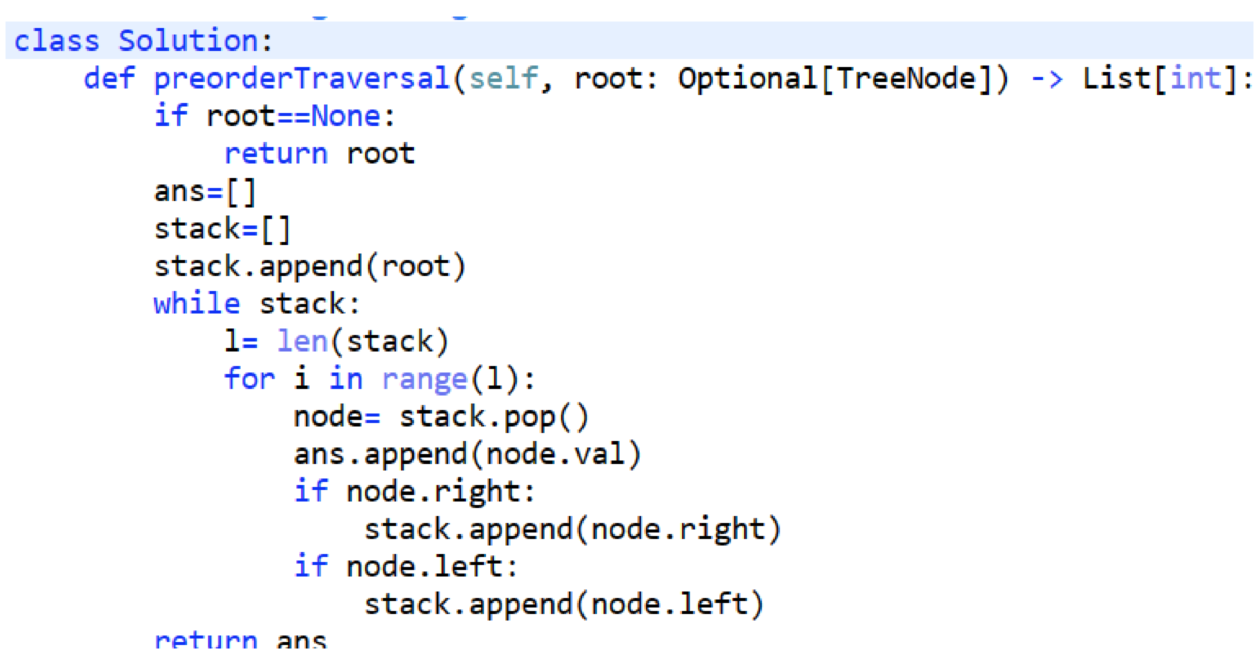
```
class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if root==None:
            return root
        ans=[]
        stack=[]
        node=root
        while(True):
            if (node!=None):
                stack.append(node)
                node=node.left
            else:
                if stack:
                    node=stack.pop()
                    ans.append(node.val)
                    node=node.right
                else:
                    break
        return ans
```

- **PREORDER : ROOT LEFT RIGHT**



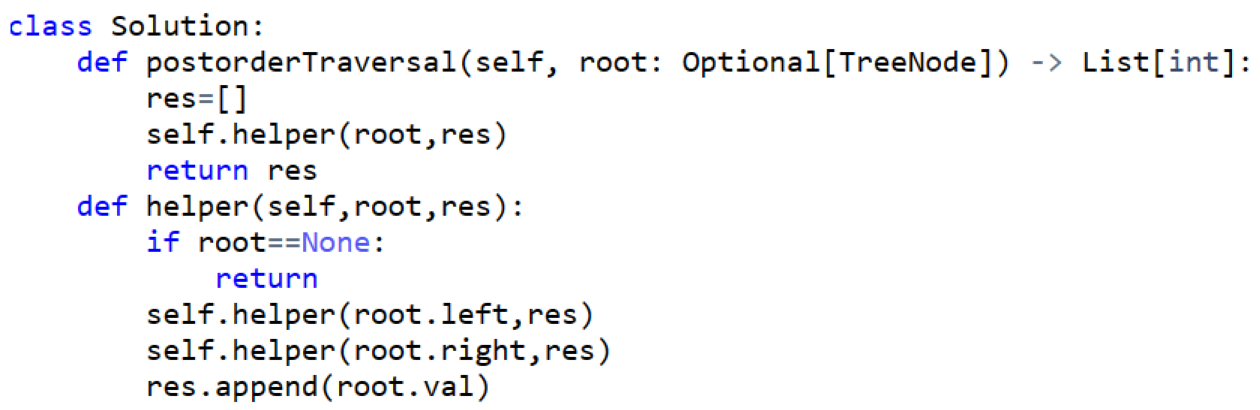
```
class Solution:
    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        res=[]


        def preorder(root):
            if not root: return
            res.append(root.val)
            preorder(root.left)
            preorder(root.right)

        preorder(root)

        return res
```

ITERATIVE SOLUTION:PREORDER



```
class Solution:
    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if root==None:
            return root
        ans=[]
        stack=[]
        stack.append(root)
        while stack:
            l= len(stack)
            for i in range(l):
                node= stack.pop()
                ans.append(node.val)
                if node.right:
                    stack.append(node.right)
                if node.left:
                    stack.append(node.left)
        return ans
```

- **POSTORDER: LEFT RIGHT ROOT**



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

NOTE: List is passed by reference so if we append any value it will reflect every where, but if we simply take any variable then we have to make it global and we can access it only with help of self.variable_name

POSTORDER ITERATIVE SOLUTION:

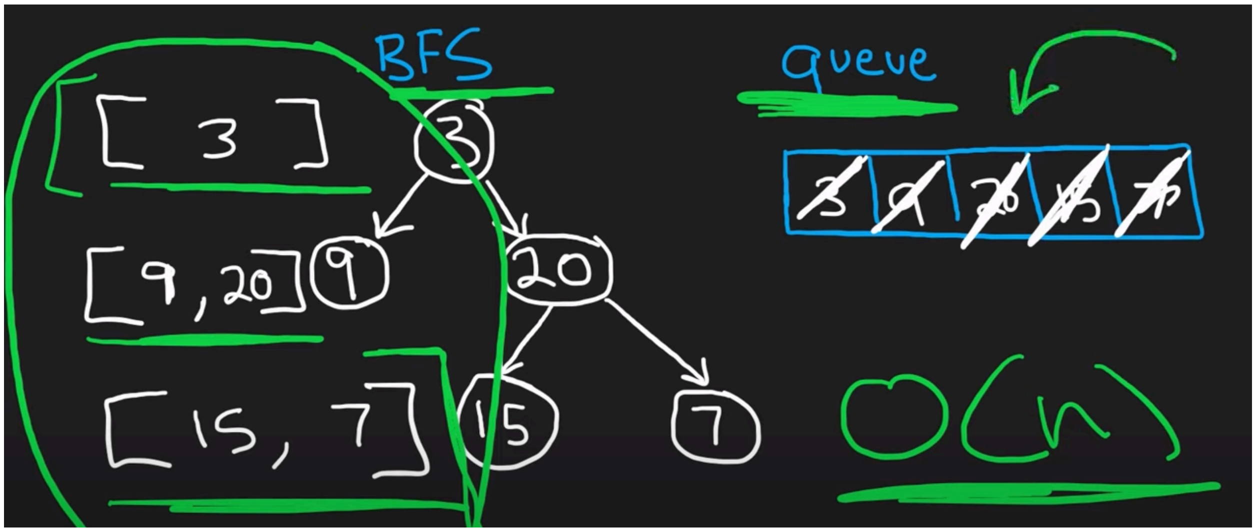
- **Using 2 stacks:**

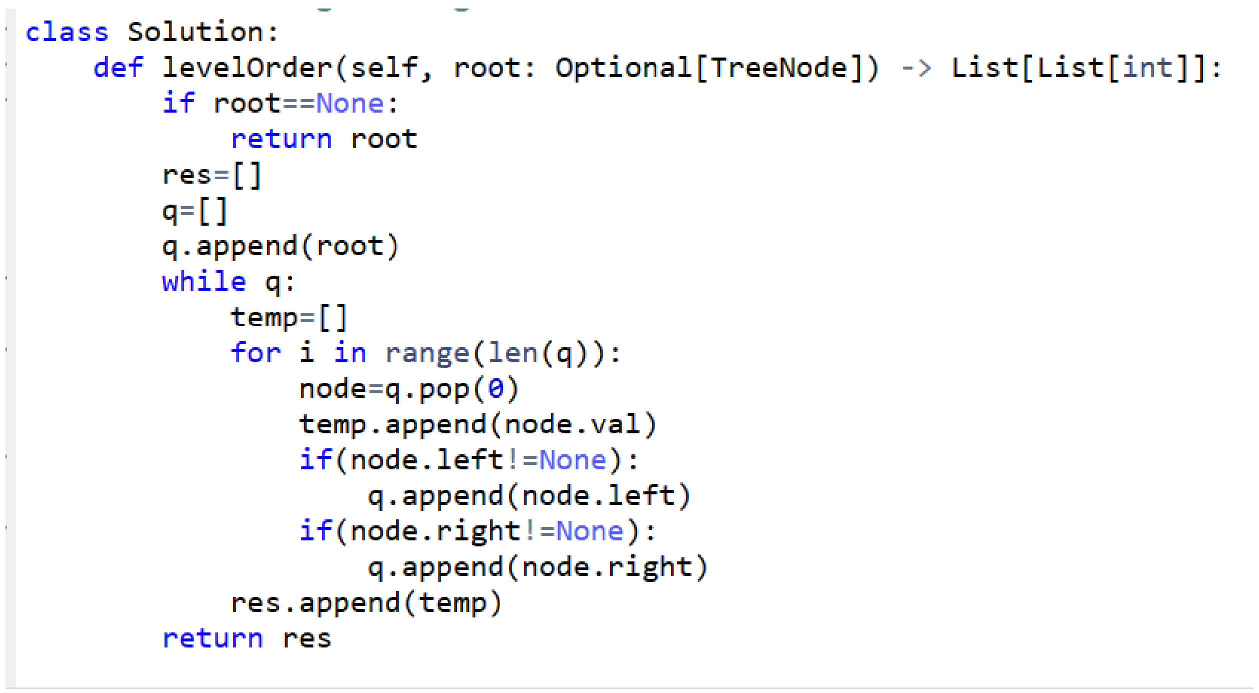
```python
class Solution:
    def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if root==None:
            return root
        ans=[]
        st1=[]
        st2=[]
        st1.append(root)
        while st1:
            node= st1.pop()
            st2.append(node.val)
            if node.left:
                st1.append(node.left)
            if node.right:
                st1.append(node.right)
        while st2:
            temp=st2.pop()
            ans.append(temp)
        return ans
```

- **Using 1 stack:**

LEVEL ORDER(BFS) TRAVERSAL:

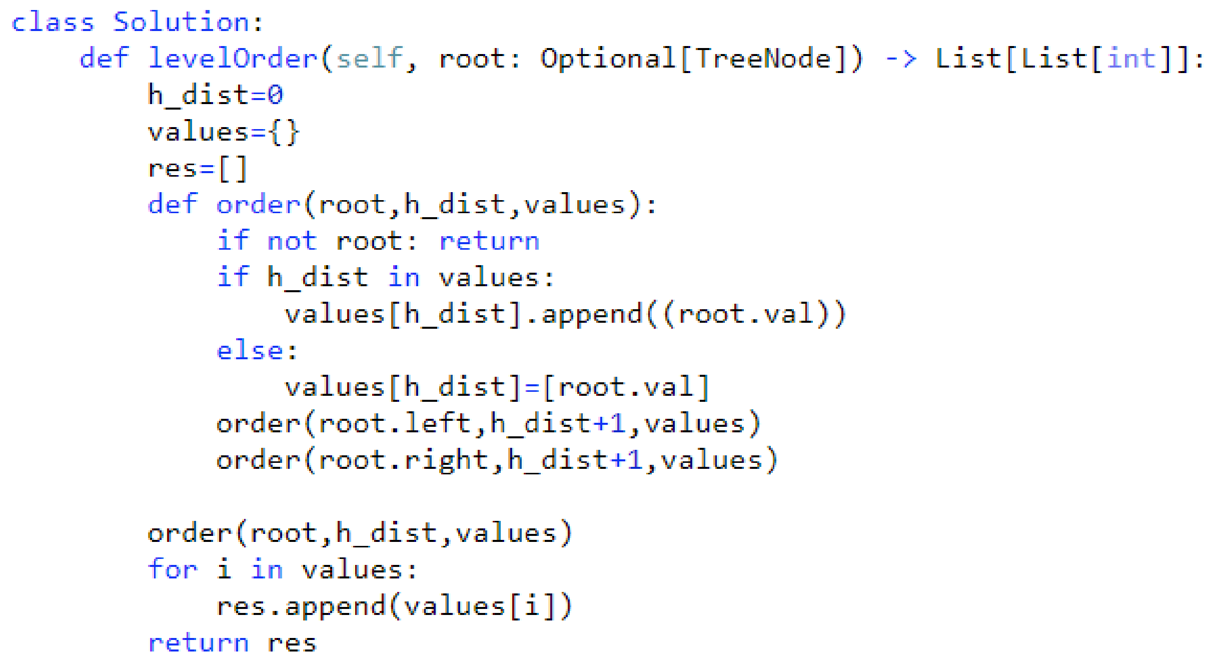
Reference: https://www.youtube.com/watch?v=6ZnyEApgFYg





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

Recursive approach:

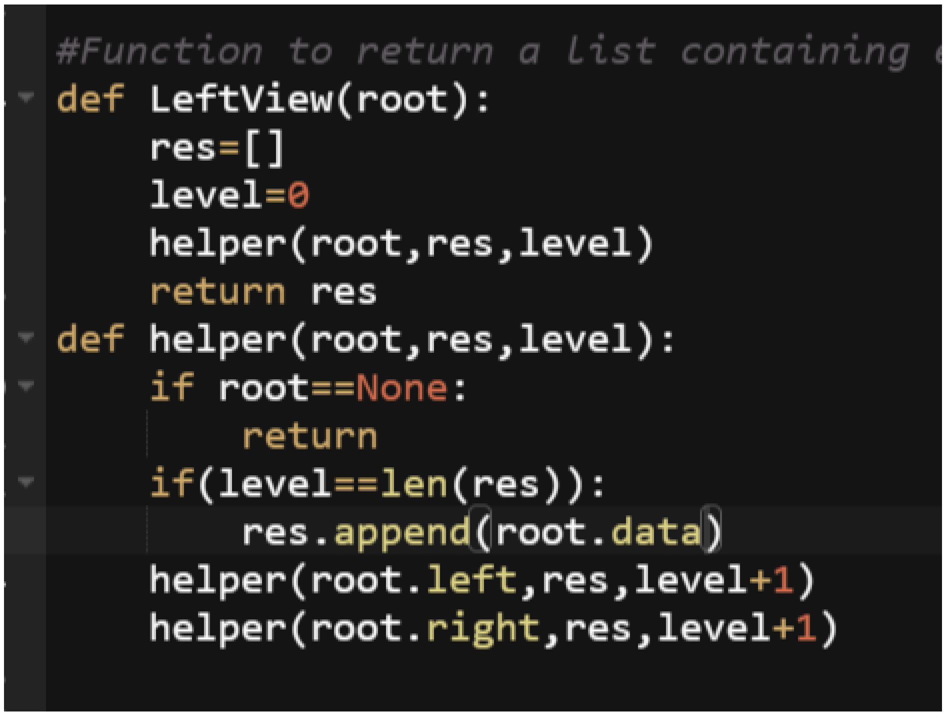


```
class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        h_dist=0
        values={}
        res=[]
        def order(root,h_dist,values):
            if not root: return
            if h_dist in values:
                values[h_dist].append((root.val))
            else:
                values[h_dist]=[root.val]
            order(root.left,h_dist+1,values)
            order(root.right,h_dist+1,values)

        order(root,h_dist,values)
        for i in values:
            res.append(values[i])
        return res
```

Q1: Left view of tree:

Intuition: Traverse like pre order traversal, **ROOT LEFT RIGHT**

And take the list where store the result of each level which is seen from left, so whenever the first time reaches to any level then store it to ans and then proceed.
TC:O(N)
SC:O(H) H=height of tree



```python
#Function to return a list containing 
def LeftView(root):
    res=[]
    level=0
    helper(root,res,level)
    return res
def helper(root,res,level):
    if root==None:
        return
    if(level==len(res)):
        res.append(root.data)
    helper(root.left,res,level+1)
    helper(root.right,res,level+1)
```

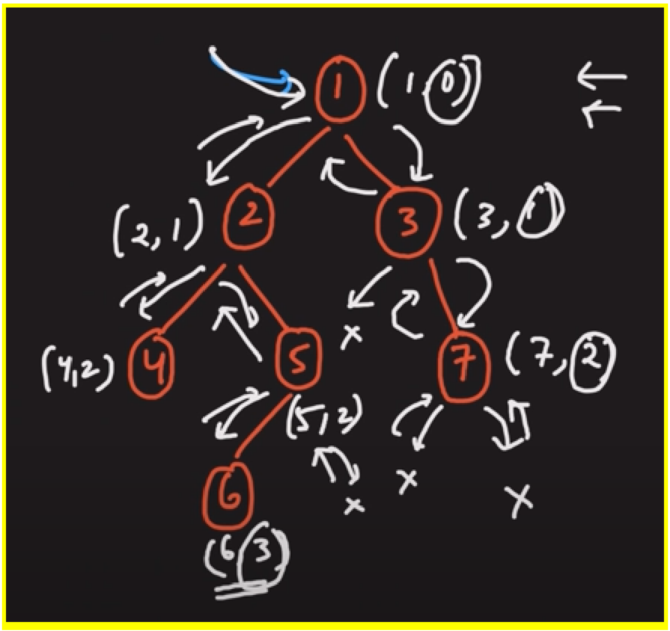
```python
def LeftView(root):

    # code here
    res=[]
    level=0
    def helper(root,level):
        if not root: return
        if level==len(res):
            res.append(root.data)
        helper(root.left,level+1)
        helper(root.right,level+1)
    helper(root,level)
    return res
```

Q2: Right view of tree:

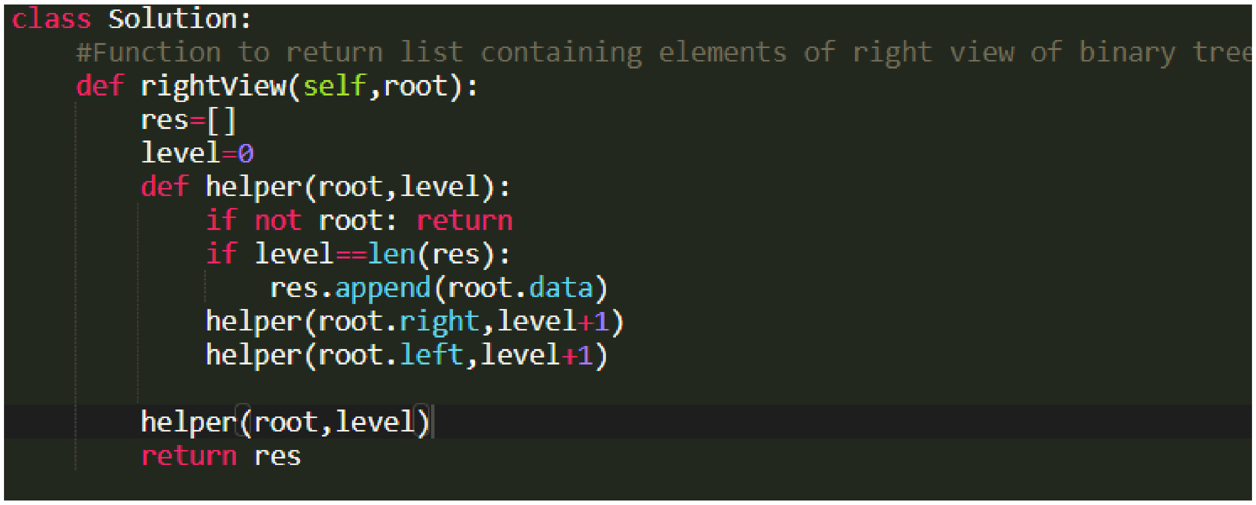
Intuition: its now reverse of pre-order traversal, **ROOT RIGHT LEFT**

TC:O(N)

SC:O(H) H=height of tree

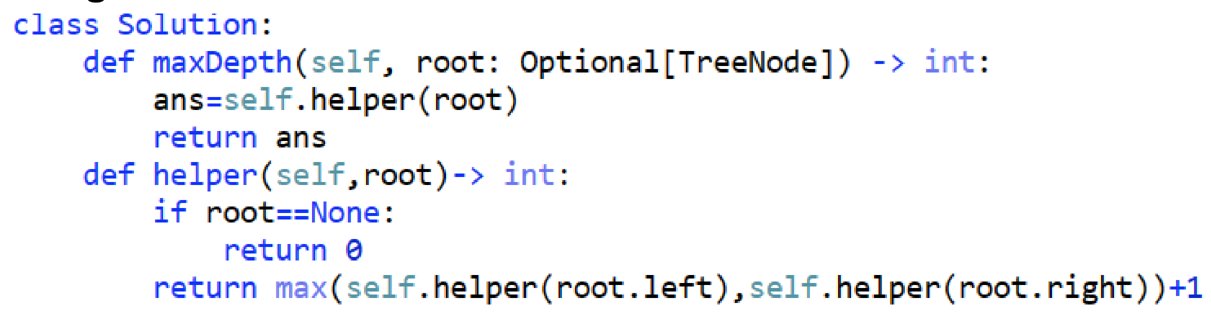


```
class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        #do reverse of pre order ROOT RIGHT LEFT
        res=[]
        level=0
        self.helper(root,res,level)
        return res
    def helper(self,root,res,level):
        if (root==None):
            return
        if(len(res)==level):
            res.append(root.val)
        self.helper(root.right,res,level+1)
        self.helper(root.left,res,level+1)
```

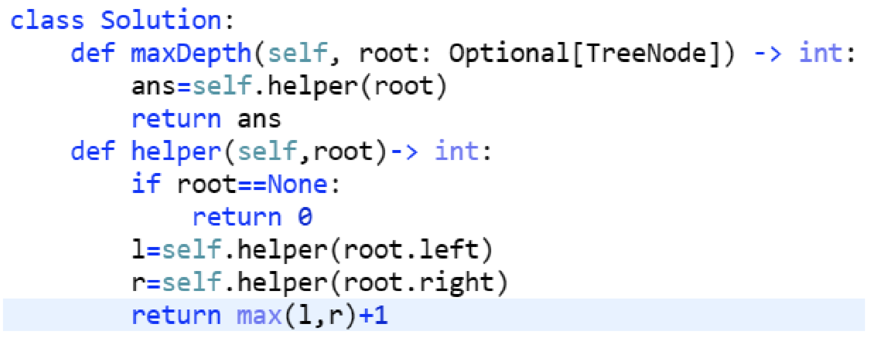

```python
class Solution:
    #Function to return list containing elements of right view of binary tree
    def rightView(self,root):
        res=[]
        level=0
        def helper(root,level):
            if not root: return
            if level==len(res):
                res.append(root.data)
            helper(root.right,level+1)
            helper(root.left,level+1)

        helper(root,level)
        return res
```
88

Q3:Finding depth of tree:

Using recursion:

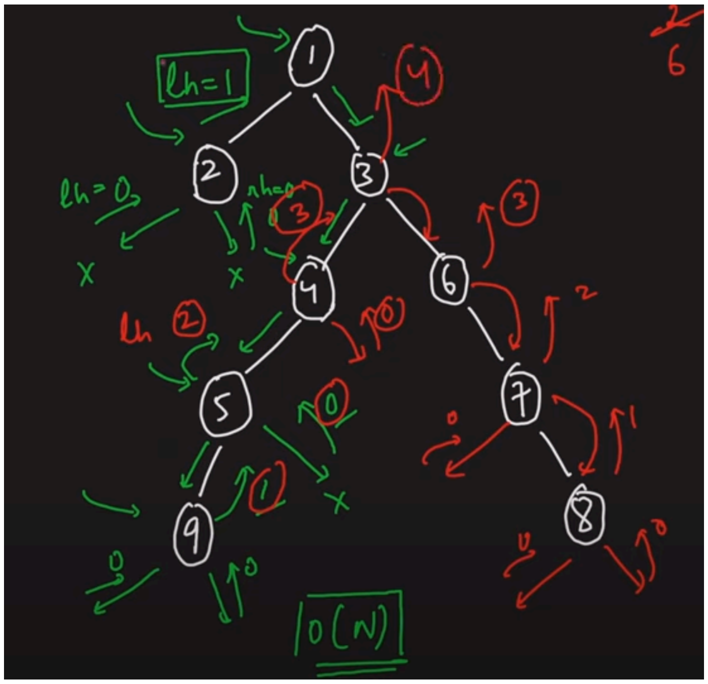

```python
class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        ans=self.helper(root)
        return ans
    def helper(self,root)-> int:
        if root==None:
            return 0
        return max(self.helper(root.left),self.helper(root.right))+1
```

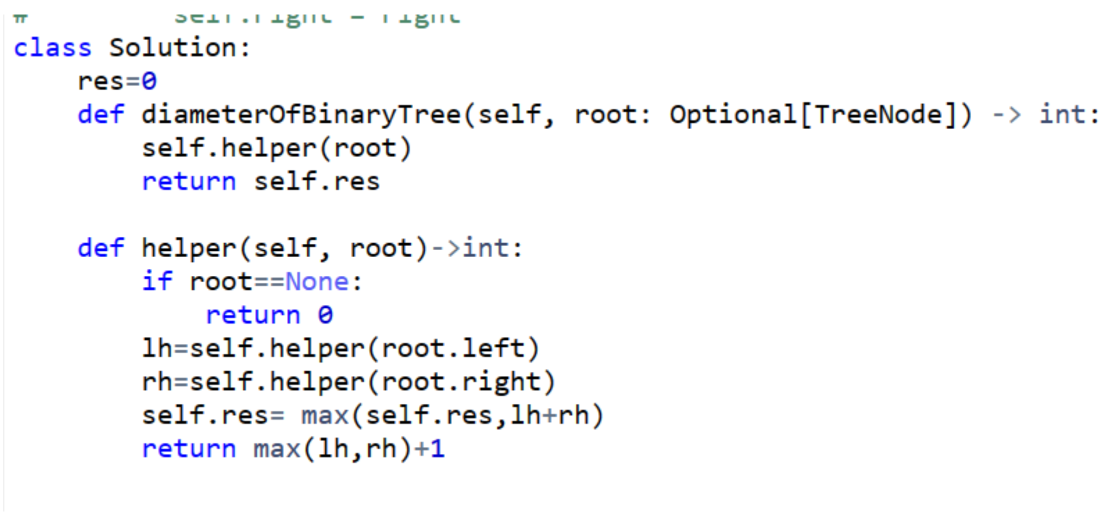
Just another variation in code to understand the logic, nothing different from above code:


```python
class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        ans=self.helper(root)
        return ans
    def helper(self,root)-> int:
        if root==None:
            return 0
        l=self.helper(root.left)
        r=self.helper(root.right)
        return max(l,r)+1
```

Q4: Diameter of Tree:

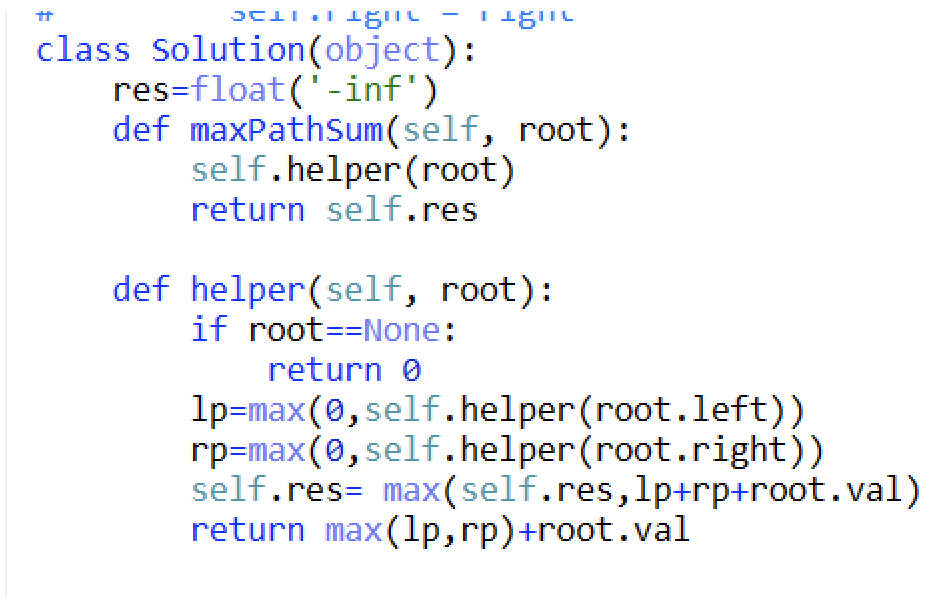
Diameter: Longest path between two nodes which need not to be passed through the root.



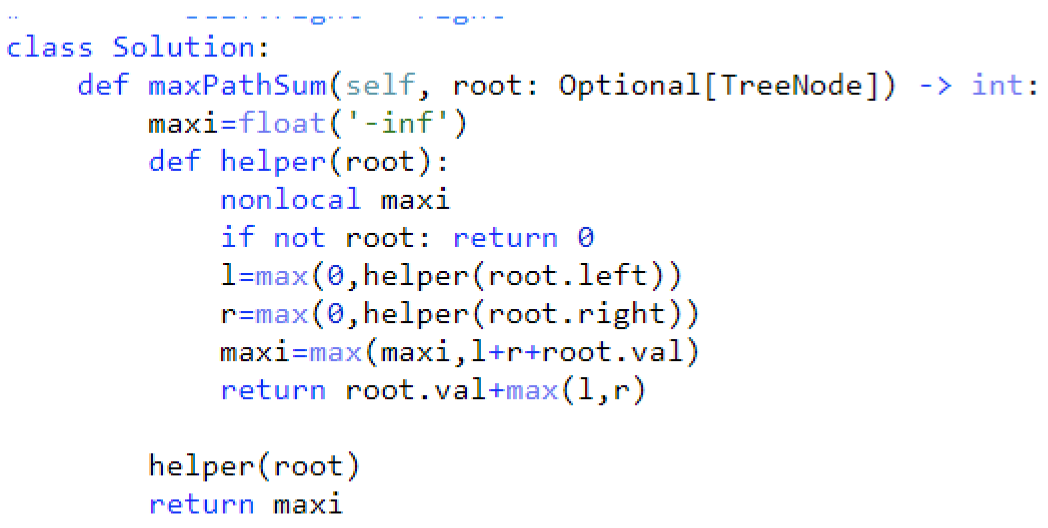


```
#         self.right = right
class Solution:
    res=0
    def diameterOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        self.helper(root)
        return self.res

    def helper(self, root)->int:
        if root==None:
            return 0
        lh=self.helper(root.left)
        rh=self.helper(root.right)
        self.res= max(self.res,lh+rh)
        return max(lh,rh)+1
```

Q: MAX PATH SUM:

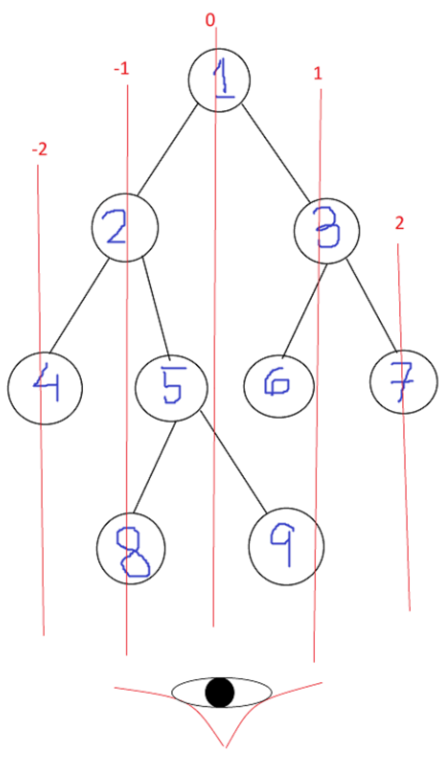


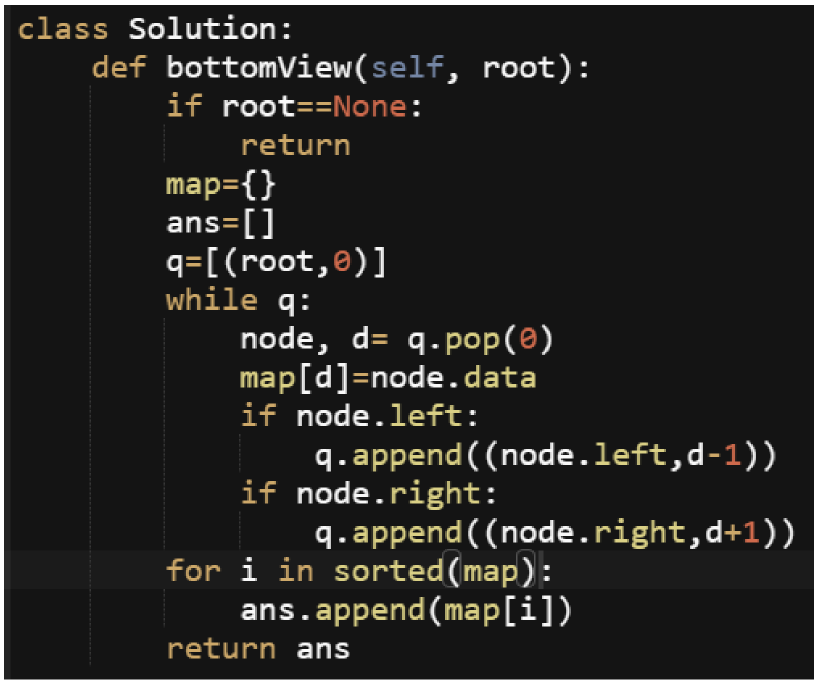
```
#         self.right = right
class Solution(object):
    res=float('-inf')
    def maxPathSum(self, root):
        self.helper(root)
        return self.res

    def helper(self, root):
        if root==None:
            return 0
        lp=max(0,self.helper(root.left))
        rp=max(0,self.helper(root.right))
        self.res= max(self.res,lp+rp+root.val)
        return max(lp,rp)+root.val
```



```
#         self.right = right
class Solution:
    def maxPathSum(self, root: Optional[TreeNode]) -> int:
        maxi=float('-inf')
        def helper(root):
            nonlocal maxi
            if not root: return 0
            l=max(0,helper(root.left))
            r=max(0,helper(root.right))
            maxi=max(maxi,l+r+root.val)
            return root.val+max(l,r)

        helper(root)
        return maxi
```

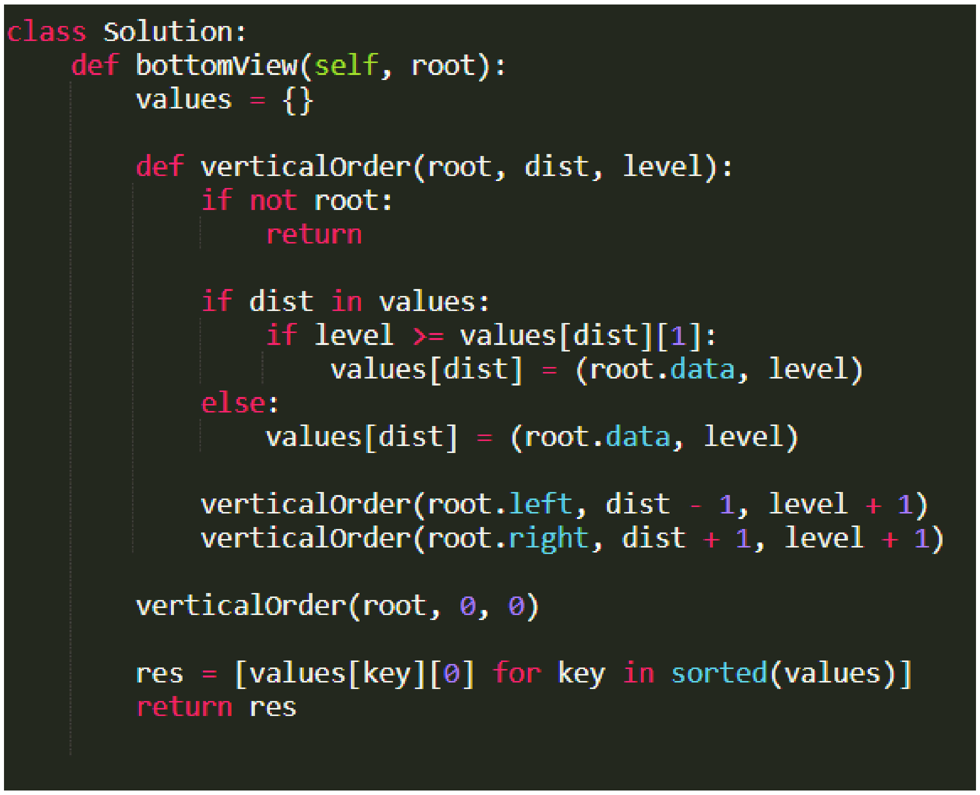
Q5: Bottom view of tree:





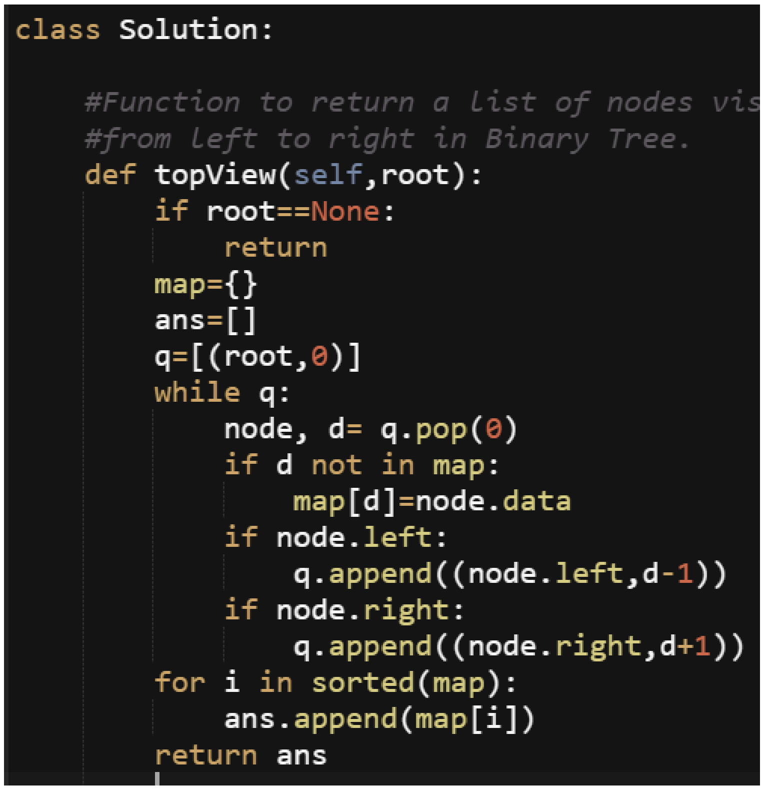
```
class Solution:
    def bottomView(self, root):
        if root==None:
            return
        map={}
        ans=[]
        q=[(root,0)]
        while q:
            node, d= q.pop(0)
            map[d]=node.data
            if node.left:
                q.append((node.left,d-1))
            if node.right:
                q.append((node.right,d+1))
        for i in sorted(map):
            ans.append(map[i])
        return ans
```

Recursive approach:



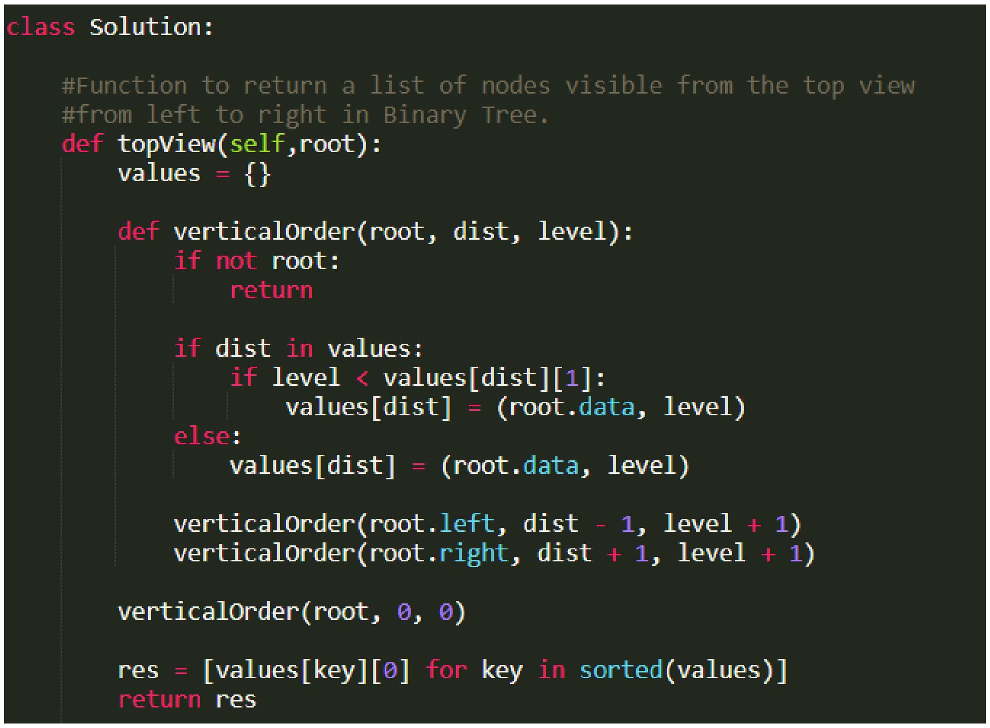
```
class Solution:
    def bottomView(self, root):
        values = {}

        def verticalOrder(root, dist, level):
            if not root:
                return

            if dist in values:
                if level >= values[dist][1]:
                    values[dist] = (root.data, level)
            else:
                values[dist] = (root.data, level)

            verticalOrder(root.left, dist - 1, level + 1)
            verticalOrder(root.right, dist + 1, level + 1)

        verticalOrder(root, 0, 0)

        res = [values[key][0] for key in sorted(values)]
        return res
```

Q6: **Top view of tree:**



```
class Solution:

    #Function to return a list of nodes vis
    #from left to right in Binary Tree.
    def topView(self,root):
        if root==None:
            return
        map={}
        ans=[]
        q=[(root,0)]
        while q:
            node, d= q.pop(0)
            if d not in map:
                map[d]=node.data
            if node.left:
                q.append((node.left,d-1))
            if node.right:
                q.append((node.right,d+1))
        for i in sorted(map):
            ans.append(map[i])
        return ans
```

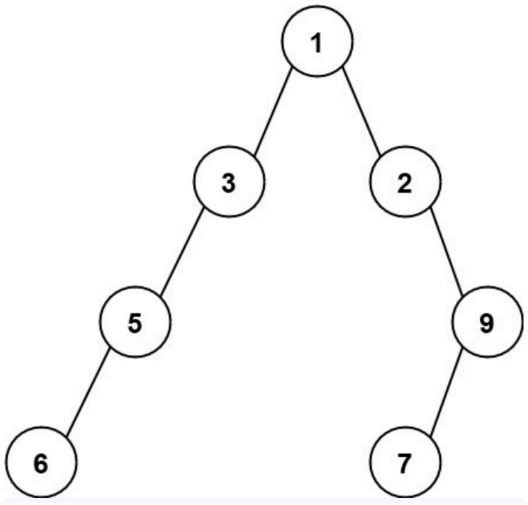
Recursive Approach:

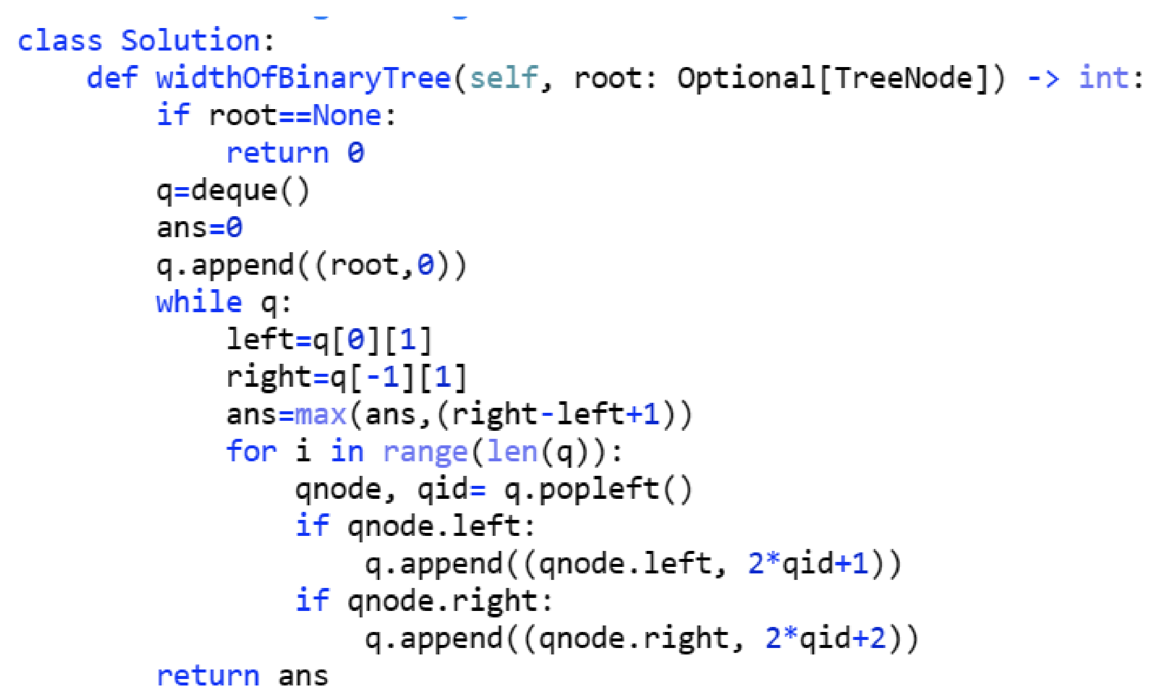


```python
class Solution:

    #Function to return a list of nodes visible from the top view
    #from left to right in Binary Tree.
    def topView(self,root):
        values = {}

        def verticalOrder(root, dist, level):
            if not root:
                return

            if dist in values:
                if level < values[dist][1]:
                    values[dist] = (root.data, level)
            else:
                values[dist] = (root.data, level)

            verticalOrder(root.left, dist - 1, level + 1)
            verticalOrder(root.right, dist + 1, level + 1)

        verticalOrder(root, 0, 0)

        res = [values[key][0] for key in sorted(values)]
        return res
```

Q7: Width of binary tree:

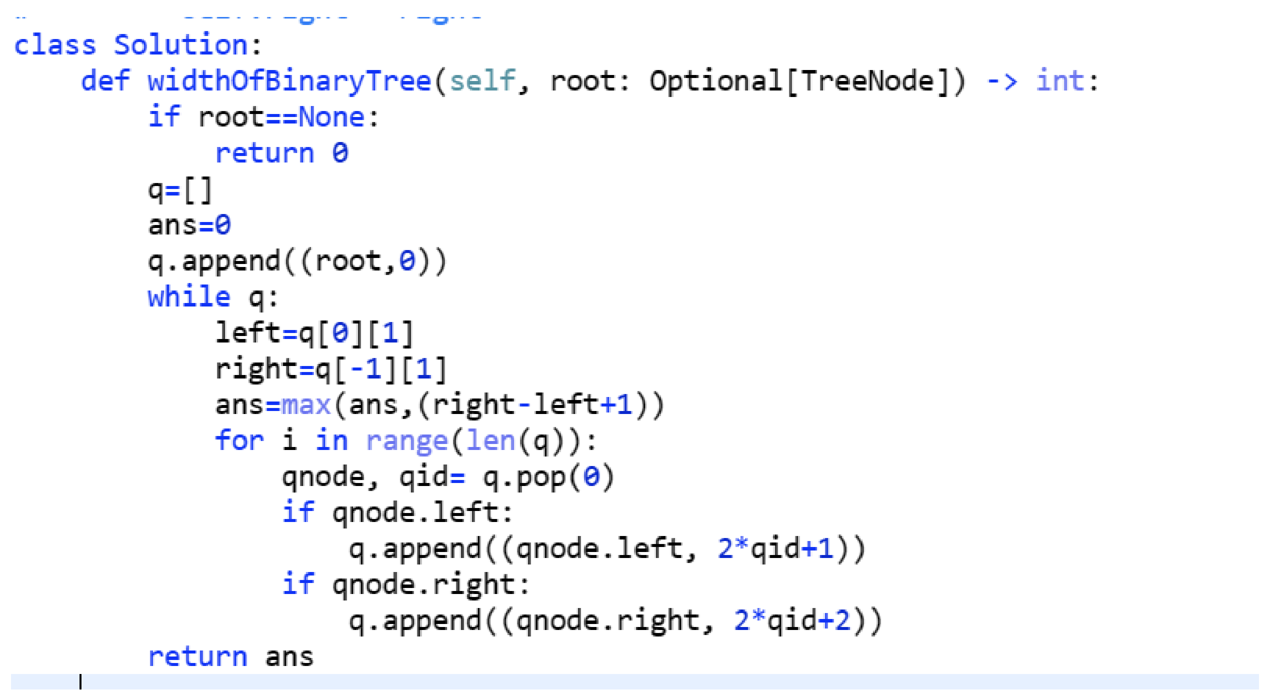
```
Input: root = [1,3,2,5,null,null,9,6,null,7]
Output: 7
Explanation: The maximum width exists in the fourth
level with length 7 (6,null,null,null,null,null,7).
```





```python
class Solution:
    def widthOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        if root==None:
            return 0
        q=deque()
        ans=0
        q.append((root,0))
        while q:
            left=q[0][1]
            right=q[-1][1]
            ans=max(ans,(right-left+1))
            for i in range(len(q)):
                qnode, qid= q.popleft()
                if qnode.left:
                    q.append((qnode.left, 2*qid+1))
                if qnode.right:
                    q.append((qnode.right, 2*qid+2))
        return ans
```

Deque is similar as list as we can insert and delete element from both the end. I am writing this code now with list with very minute variation . It is as follows.



```python
class Solution:
    def widthOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        if root==None:
            return 0
        q=[]
        ans=0
        q.append((root,0))
        while q:
            left=q[0][1]
            right=q[-1][1]
            ans=max(ans,(right-left+1))
            for i in range(len(q)):
                qnode, qid= q.pop(0)
                if qnode.left:
                    q.append((qnode.left, 2*qid+1))
                if qnode.right:
                    q.append((qnode.right, 2*qid+2))
        return ans
```

DEQUE:

A deque is a generalization of a queue where elements can be added or removed from both ends. In other words, it supports operations such as inserting and removing elements from both the front and the back of the queue.

For most cases, using a deque as a simple queue is sufficient, as it provides the necessary methods for queue operations like append() to add elements to the back of the queue and popleft() to remove elements from the front of the queue efficiently.

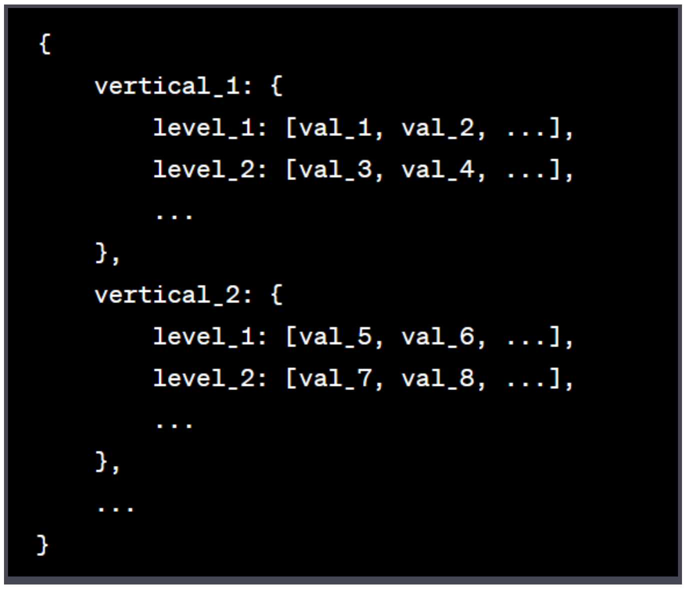
- Time Complexity:
- Appending and popping from both ends (append, appendleft, pop, popleft): O(1)
- Accessing elements by index (deque[i]): O(1)
- Removing an arbitrary element (remove): O(n)
- Insertion or deletion in the middle: O(n)
- Searching for an element: O(n)
- Space Complexity: O(n) - the size of the deque grows linearly with the number of elements store

QUEUE:

If you only need a simple queue functionality, using a deque is a suitable choice. However, if you require additional features like constant-time random access or indexing of elements, you may consider using other specialized data structures like queue.Queue or collections.deque based on your specific requirements

- The queue module in Python provides multiple queue implementations, including Queue, LifoQueue, and PriorityQueue. For simplicity, we'll focus on Queue.
- Time Complexity:
- Enqueue (put): O(1)
- Dequeue (get): O(1)
- Accessing elements by index is not supported in Queue.
- Space Complexity: O(n) - the space used by a Queue object is proportional to the number of elements stored.

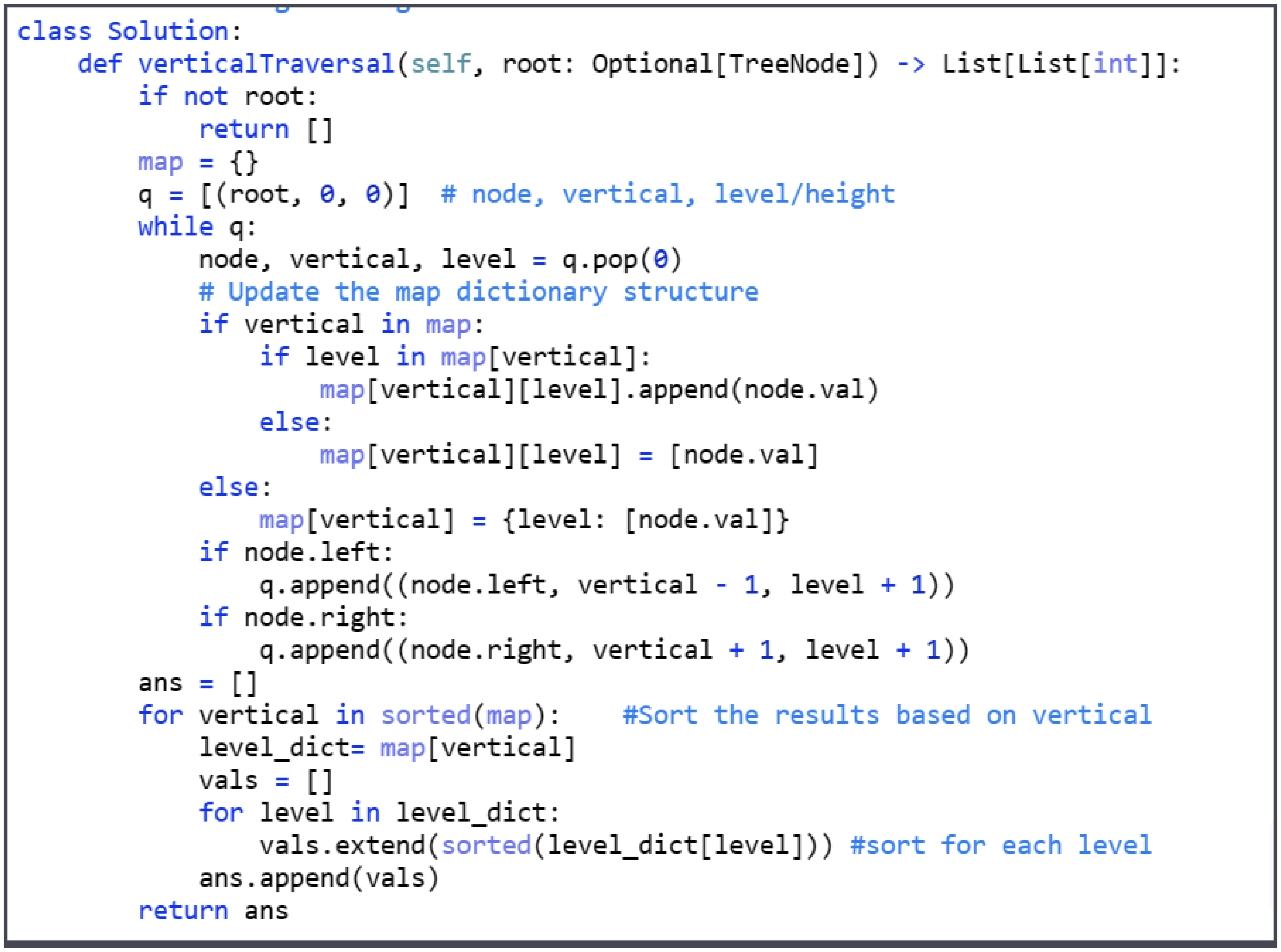
Q8: Vertical Order Traversal of a Binary Tree



```
{
    vertical_1: {
        level_1: [val_1, val_2, ...],
        level_2: [val_3, val_4, ...],
        ...
    },
    vertical_2: {
        level_1: [val_5, val_6, ...],
        level_2: [val_7, val_8, ...],
        ...
    },
    ...
}
```

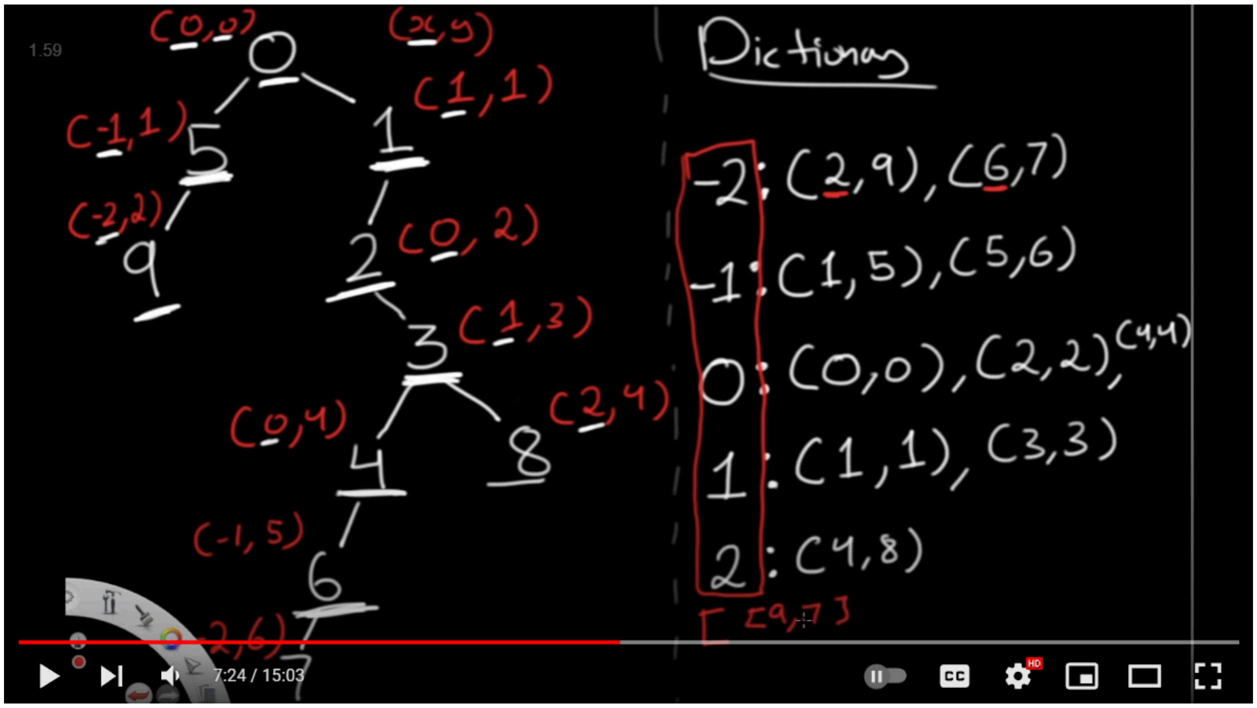
Intuition: We take a queue and filled it with (nodeval, vertical, level)
Initially , it is (root, 0, 0)
We make a dictionary-like structure shown above in image. We need to append based on verticals , like -2, -1,0,1,2 so we need it in sorted order.

pop(): pop the element from end of the list
pop(0): when we need to pop the element from front
del list_name[-1]: delete from end of the list

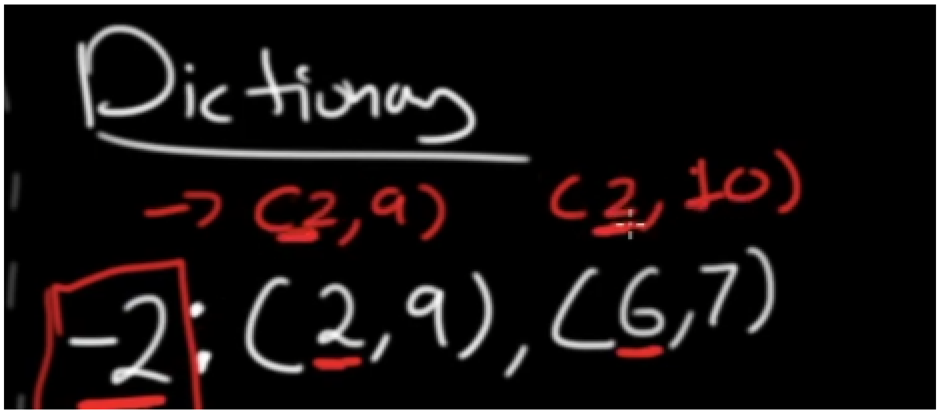


```
class Solution:
    def verticalTraversal(self, root: Optional[TreeNode]) -> List[List[int]]:
        if not root:
            return []
        map = {}
        q = [(root, 0, 0)]  # node, vertical, level/height
        while q:
            node, vertical, level = q.pop(0)
            # Update the map dictionary structure
            if vertical in map:
                if level in map[vertical]:
                    map[vertical][level].append(node.val)
                else:
                    map[vertical][level] = [node.val]
            else:
                map[vertical] = {level: [node.val]}
            if node.left:
                q.append((node.left, vertical - 1, level + 1))
            if node.right:
                q.append((node.right, vertical + 1, level + 1))
        ans = []
        for vertical in sorted(map):    #Sort the results based on vertical
            level_dict= map[vertical]
            vals = []
            for level in level_dict:
                vals.extend(sorted(level_dict[level])) #sort for each level
            ans.append(vals)
        return ans
```

Another soln:
Reference: https://www.youtube.com/watch?v=xs_deEJXflw
In this soln you have to be clear about one thing that if at same horizontal distance two values are coming then we do not have to sort them in increasing order **but when at same horizontal distance and at same vertical distance then we have to sort in ascending order**
Eg:
In this example as value 9 and 7 do not come at same vertical distance but only at same horizontal distance so we will save (9,7) not (7,9)



In this case 9 and 10 both comes at same horizontal and vertical level so now we have to save it in ascending order.

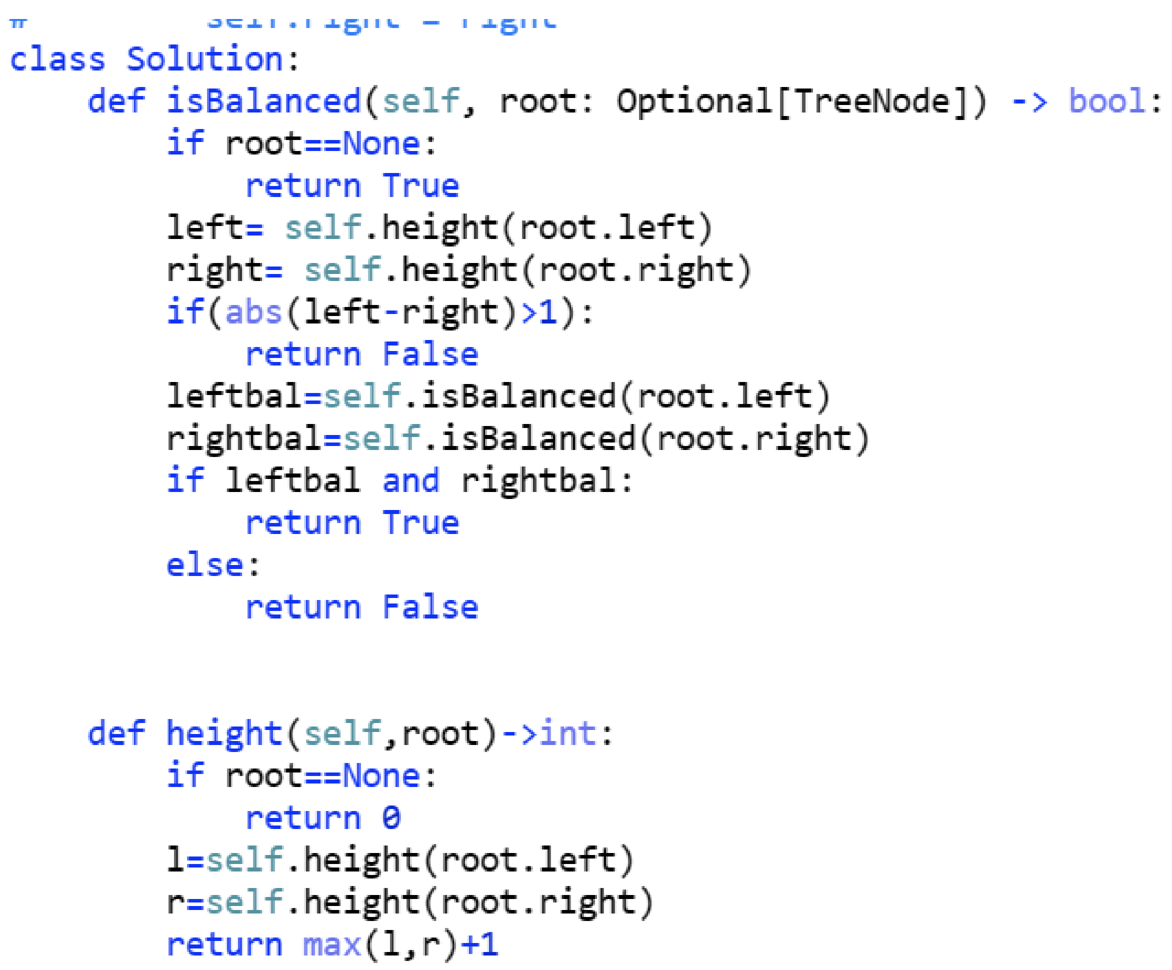


```python
    def verticalTraversal(self, root: Optional[TreeNode]) -> List[List[int]]:
        v_dist=0
        h_dist=0
        values={}
        res=[]


        def verticalOrder(root,h_dist,v_dist,values):
            if not root: return

            if h_dist in values:
                values[h_dist].append((v_dist,root.val))
            else:
                values[h_dist]=[(v_dist,root.val)]

            verticalOrder(root.left,h_dist-1,v_dist+1,values)
            verticalOrder(root.right,h_dist+1,v_dist+1,values)

        verticalOrder(root,h_dist,v_dist,values)
        for x in sorted(values.keys()):
            column=[i[1] for i in sorted(values[x])]
            res.append(column)
        return res
```

110. Balanced Binary Tree

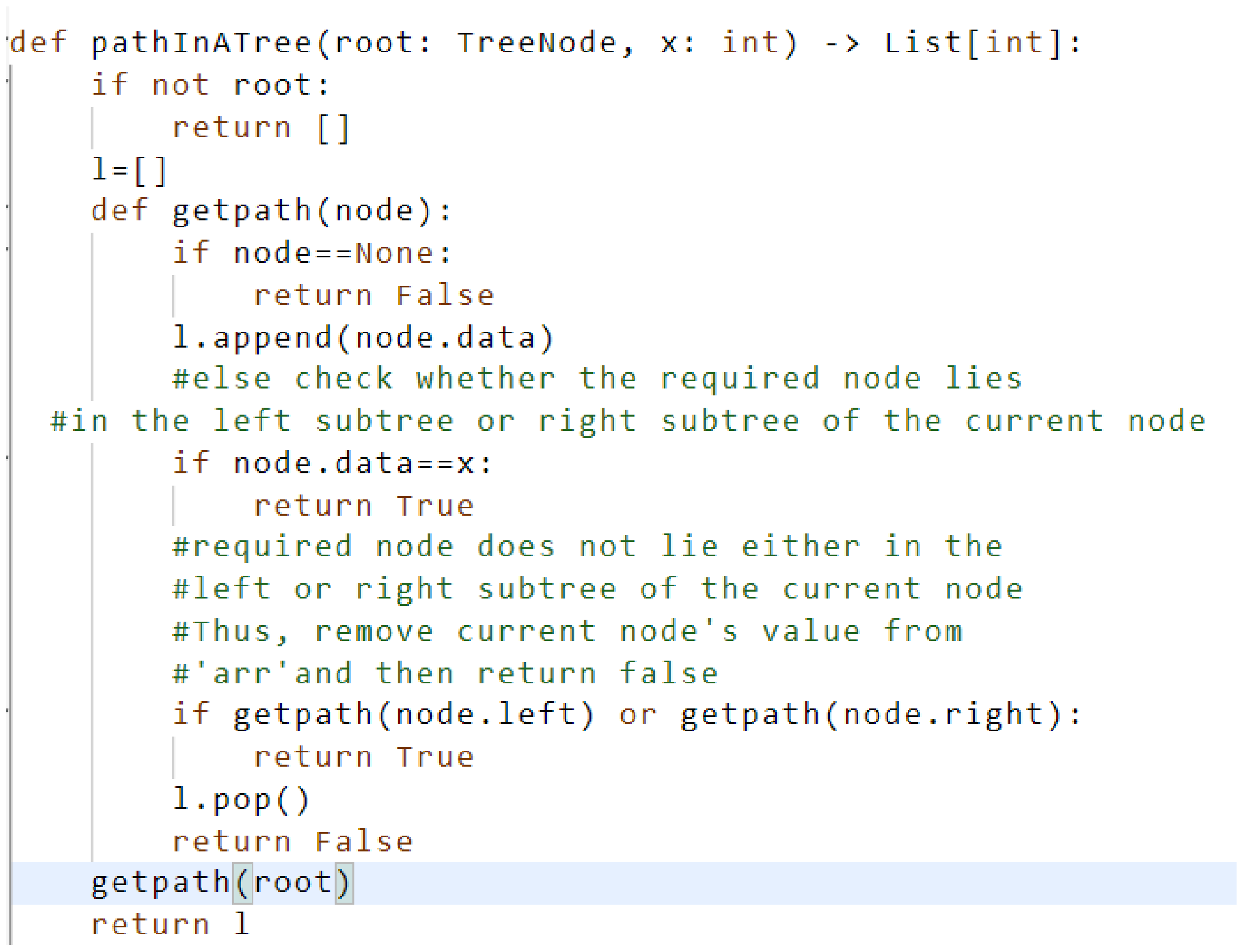
```python
class Solution:
    def isBalanced(self, root: Optional[TreeNode]) -> bool:
        if not root: return True
        def height(root):
            if not root: return 0
            l=height(root.left)
            r=height(root.right)
            return 1+max(l,r)

        return abs(height(root.left)-height(root.right))<=1 and self.isBalanced(root.left) and
self.isBalanced(root.right)
```

Another solution you can look at:

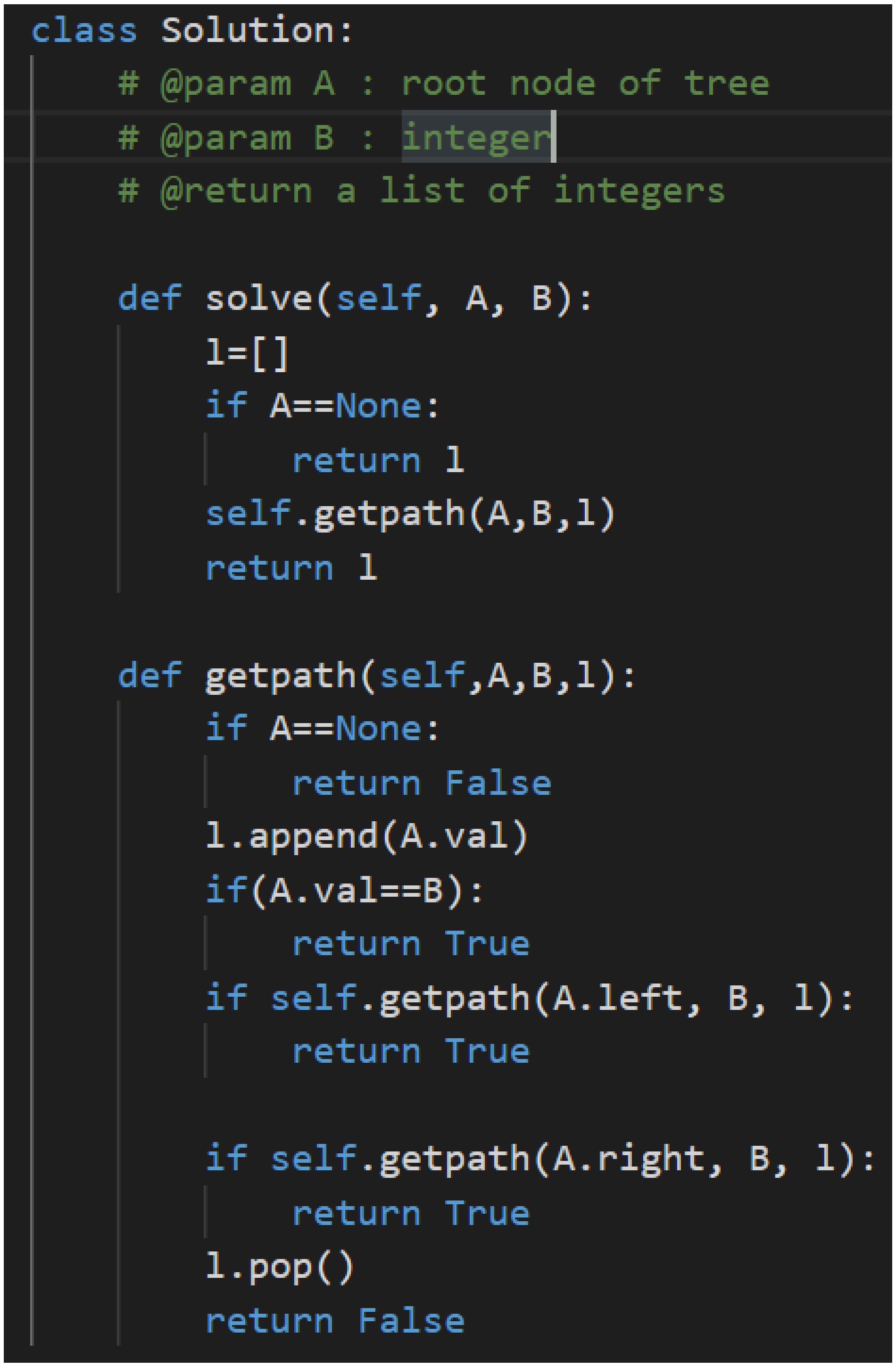


```
#         self.right = right
class Solution:
    def isBalanced(self, root: Optional[TreeNode]) -> bool:
        if root==None:
            return True
        left= self.height(root.left)
        right= self.height(root.right)
        if(abs(left-right)>1):
            return False
        leftbal=self.isBalanced(root.left)
        rightbal=self.isBalanced(root.right)
        if leftbal and rightbal:
            return True
        else:
            return False


    def height(self,root)->int:
        if root==None:
            return 0
        l=self.height(root.left)
        r=self.height(root.right)
        return max(l,r)+1
```

Q: Path to Given Node: Given a Binary Tree A containing N nodes.You need to find the path from Root to a given node B.



```
def pathInATree(root: TreeNode, x: int) -> List[int]:
    if not root:
        return []
    l=[]
    def getpath(node):
        if node==None:
            return False
        l.append(node.data)
        #else check whether the required node lies
  #in the left subtree or right subtree of the current node
        if node.data==x:
            return True
        #required node does not lie either in the
        #left or right subtree of the current node
        #Thus, remove current node's value from
        #'arr'and then return false
        if getpath(node.left) or getpath(node.right):
            return True
        l.pop()
        return False
    getpath(root)
    return l
```



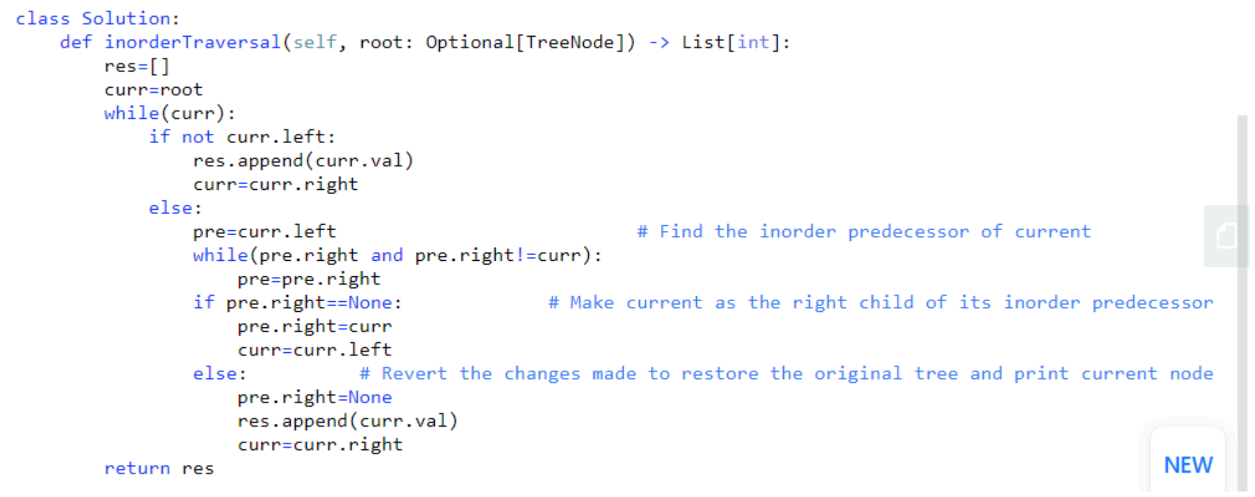
```python
class Solution:
    # @param A : root node of tree
    # @param B : integer
    # @return a list of integers

    def solve(self, A, B):
        l=[]
        if A==None:
            return l
        self.getpath(A,B,l)
        return l

    def getpath(self,A,B,l):
        if A==None:
            return False
        l.append(A.val)
        if(A.val==B):
            return True
        if self.getpath(A.left, B, l):
            return True

        if self.getpath(A.right, B, l):
            return True
        l.pop()
        return False
```

TC: O(N) **SC:O(N)**

Q; Morris Inorder Traversal (Space Complexity - O(1))

```
1. Initialize current as root
```

```
2. While current is not NULL
   If the current does not have left child
      a) Print current's data
      b) Go to the right, i.e., current = current->right
   Else
      a) Find rightmost node in current left subtree OR
              node whose right child == current.
         If we found right child == current
             a) Update the right child as NULL of that node whose right child is current
             b) Print current's data
             c) Go to the right, i.e. current = current->right
         Else
             a) Make current as the right child of that rightmost
                node we found; and
             b) Go to this left child, i.e., current = current->left
```

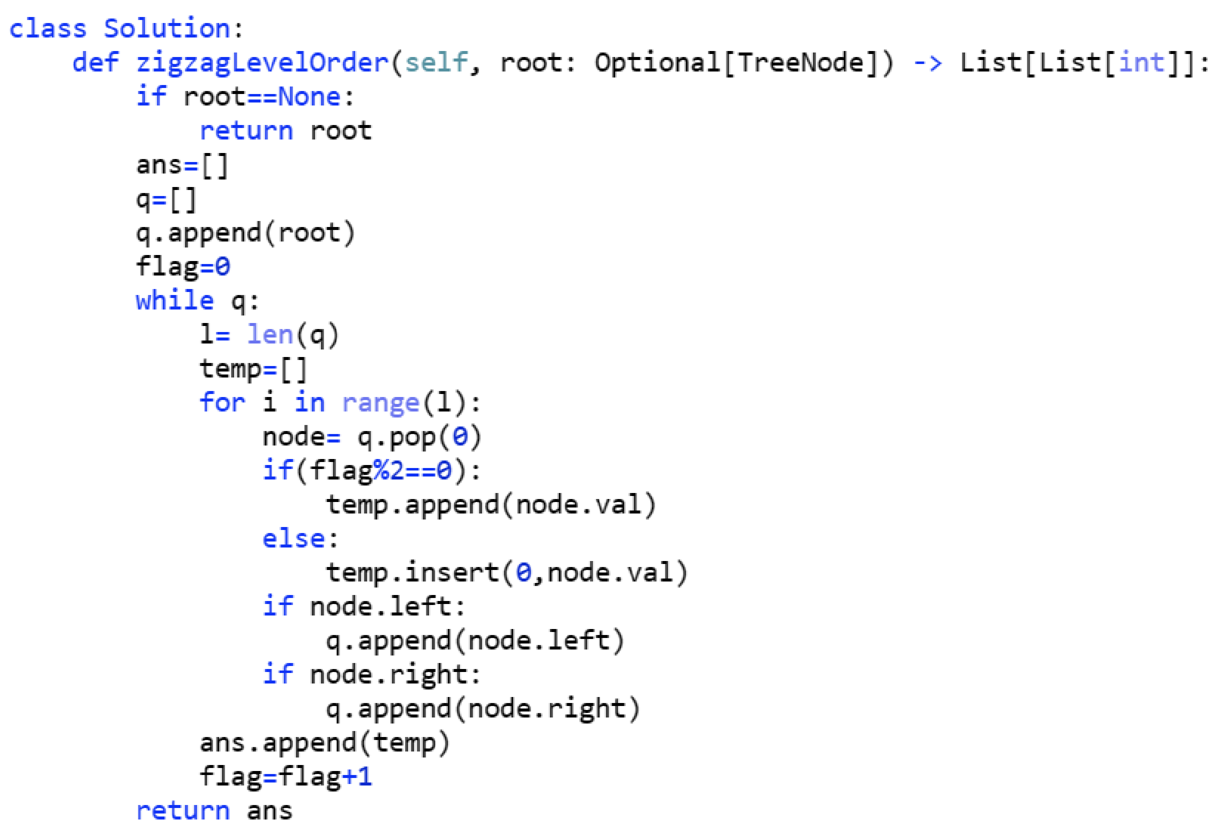


```python
class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        res=[]
        curr=root
        while(curr):
            if not curr.left:
                res.append(curr.val)
                curr=curr.right
            else:
                pre=curr.left                    # Find the inorder predecessor of current
                while(pre.right and pre.right!=curr):
                    pre=pre.right
                if pre.right==None:              # Make current as the right child of its inorder predecessor
                    pre.right=curr
                    curr=curr.left
                else:          # Revert the changes made to restore the original tree and print current node
                    pre.right=None
                    res.append(curr.val)
                    curr=curr.right
        return res
```

Time complexity : O(n) **Space complexity: O(1)**

Q: Zig-Zag traversal:

Intuition: just like level order traversal , only introduce the flag to keep track of each alternative level.

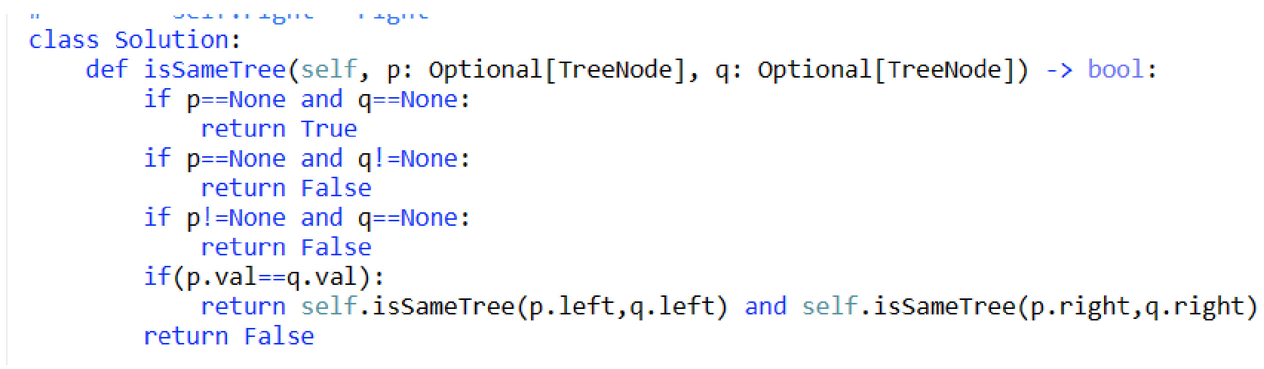


```
class Solution:
    def zigzagLevelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        if root==None:
            return root
        ans=[]
        q=[]
        q.append(root)
        flag=0
        while q:
            l= len(q)
            temp=[]
            for i in range(l):
                node= q.pop(0)
                if(flag%2==0):
                    temp.append(node.val)
                else:
                    temp.insert(0,node.val)
                if node.left:
                    q.append(node.left)
                if node.right:
                    q.append(node.right)
            ans.append(temp)
            flag=flag+1
        return ans
```

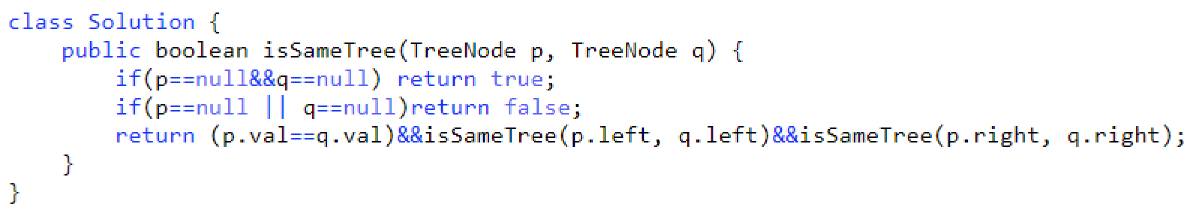
Note: To insert any value in list , we use insert(index,value_to_insert)

To insert at back of list, we use list_name.append(value)

Q: Same Tree:



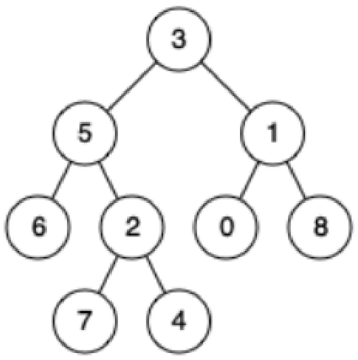
```
class Solution:
    def isSameTree(self, p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
        if p==None and q==None:
            return True
        if p==None and q!=None:
            return False
        if p!=None and q==None:
            return False
        if(p.val==q.val):
    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



```
class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if(p==null&&q==null) return true;
        if(p==null || q==null)return false;
        return (p.val==q.val)&&isSameTree(p.left, q.left)&&isSameTree(p.right, q.right);
    }
}
```

Q: Lowest Common Ancestor of a Binary Tree

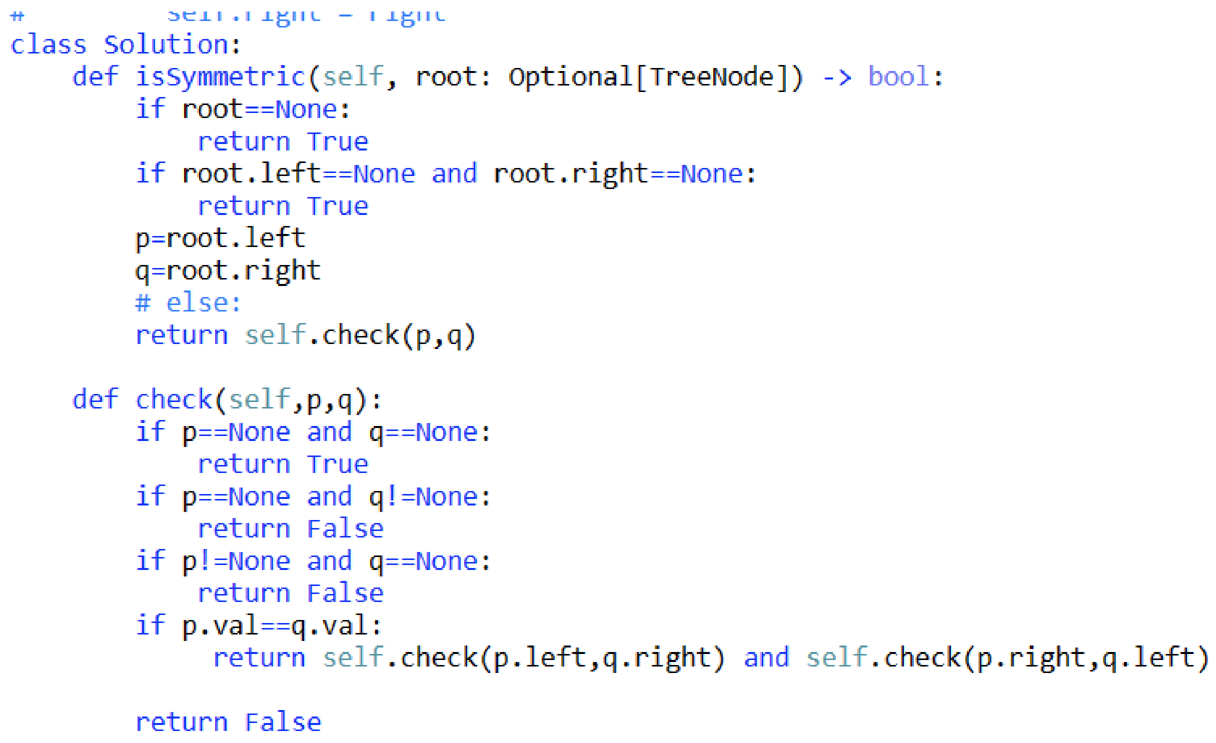
:xample 2:



```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q =
4
Output: 5
Explanation: The LCA of nodes 5 and 4 is 5, since a node
can be a descendant of itself according to the LCA
definition.
```

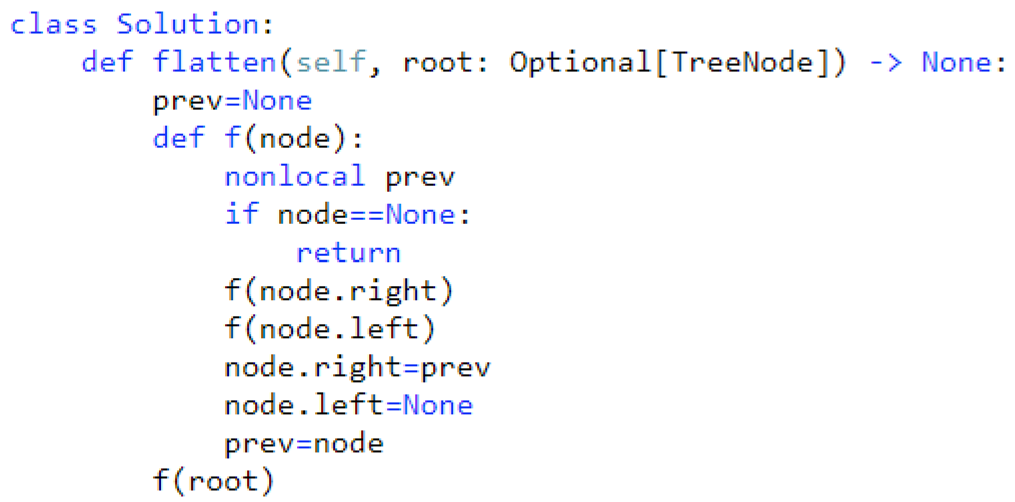
```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') ->
'TreeNode':
        if root==None or root==p or root==q:
            return root
        left= self.lowestCommonAncestor(root.left,p,q)
        right= self.lowestCommonAncestor(root.right,p,q)
        if left and right:    #both the value existes
            return root
        else:
            return left or right    #it return None when both are None otherwise return the
node.
```

Q: Symmetric Tree:

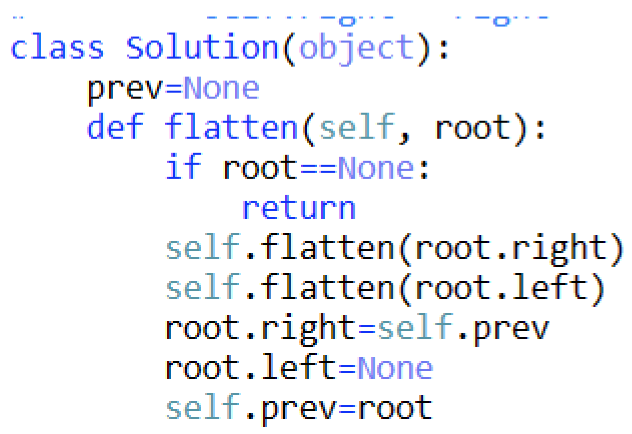


```
#         self.right = right
class Solution:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        if root==None:
            return True
        if root.left==None and root.right==None:
            return True
        p=root.left
        q=root.right
        # else:
        return self.check(p,q)

    def check(self,p,q):
        if p==None and q==None:
            return True
        if p==None and q!=None:
            return False
        if p!=None and q==None:
            return False
        if p.val==q.val:
             return self.check(p.left,q.right) and self.check(p.right,q.left)

        return False
```

Q. Flatten Binary Tree to Linked List

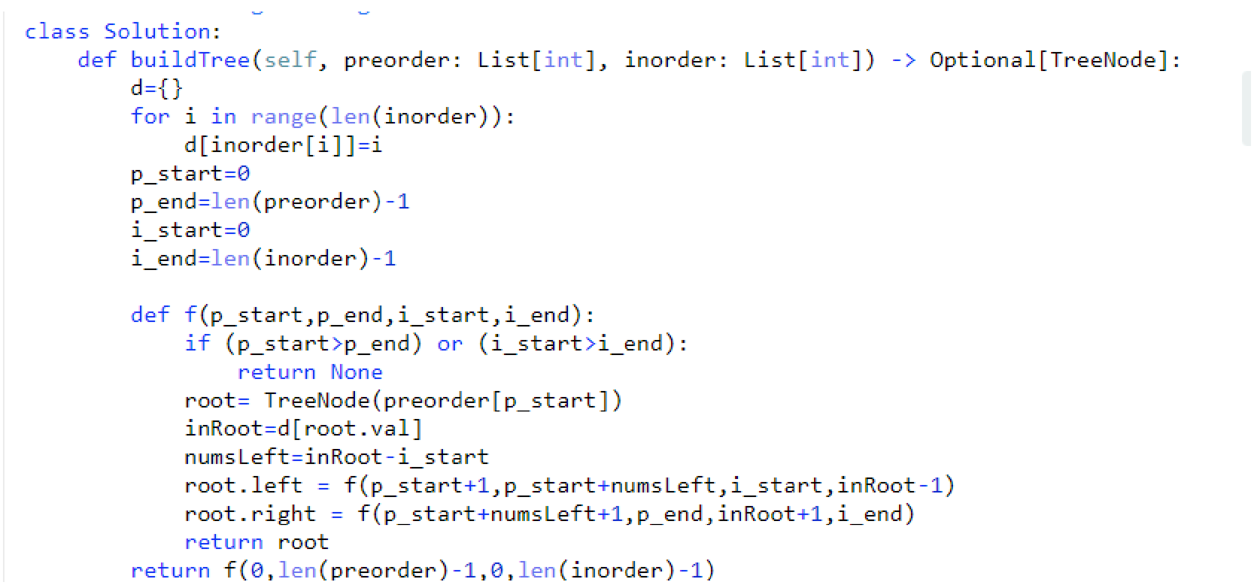


```
class Solution:
    def flatten(self, root: Optional[TreeNode]) -> None:
        prev=None
        def f(node):
            nonlocal prev
            if node==None:
                return
            f(node.right)
            f(node.left)
            node.right=prev
            node.left=None
            prev=node
        f(root)
```



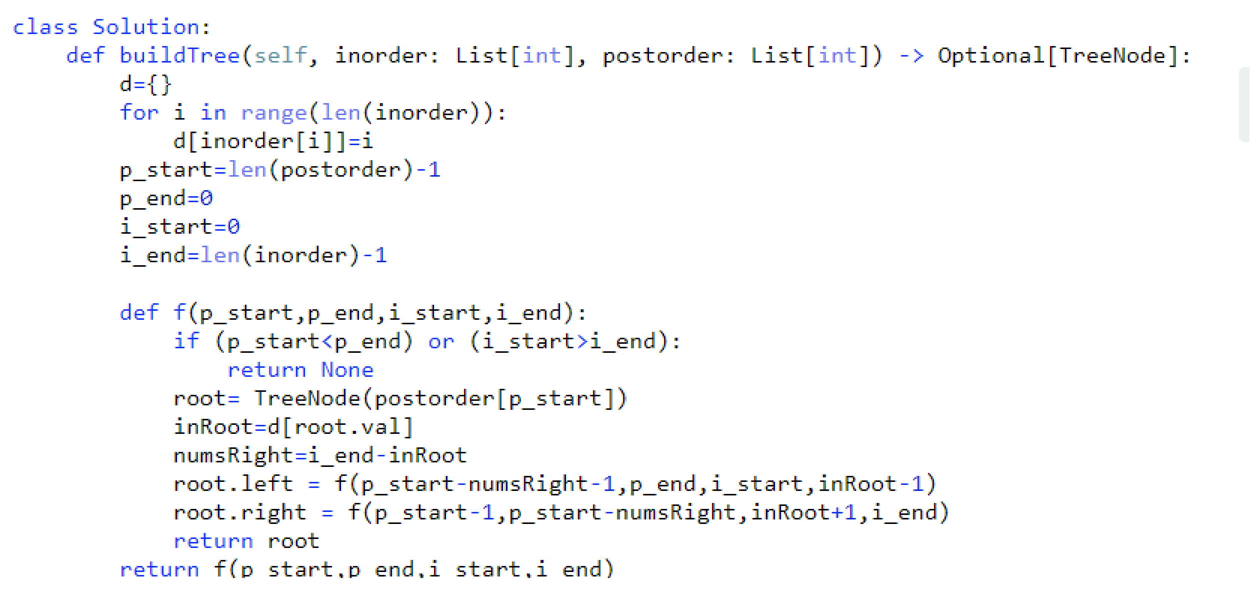
```python
class Solution(object):
    prev=None
    def flatten(self, root):
        if root==None:
            return
        self.flatten(root.right)
        self.flatten(root.left)
        root.right=self.prev
        root.left=None
        self.prev=root
```

Q Construct Binary Tree from Preorder and Inorder Traversal



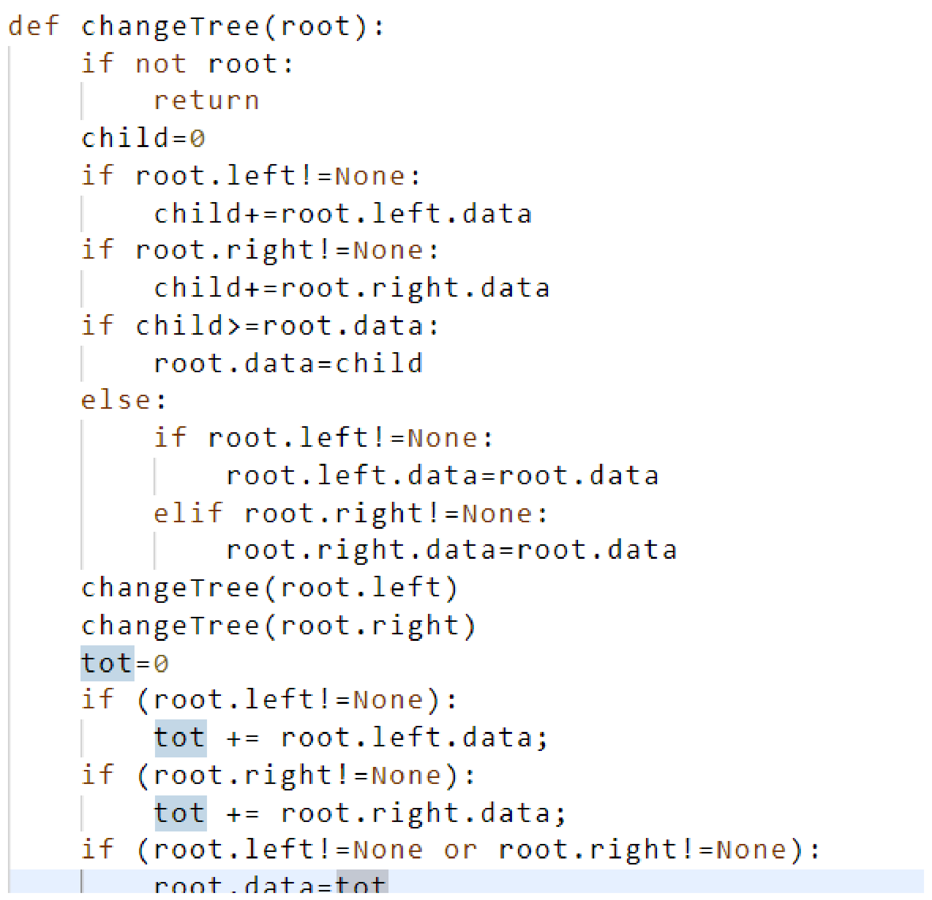
```python
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        d={}
        for i in range(len(inorder)):
            d[inorder[i]]=i
        p_start=0
        p_end=len(preorder)-1
        i_start=0
        i_end=len(inorder)-1

        def f(p_start,p_end,i_start,i_end):
            if (p_start>p_end) or (i_start>i_end):
                return None
            root= TreeNode(preorder[p_start])
            inRoot=d[root.val]
            numsLeft=inRoot-i_start
            root.left = f(p_start+1,p_start+numsLeft,i_start,inRoot-1)
            root.right = f(p_start+numsLeft+1,p_end,inRoot+1,i_end)
            return root
        return f(0,len(preorder)-1,0,len(inorder)-1)
```

Q Construct Binary Tree from Inorder and Postorder Traversal



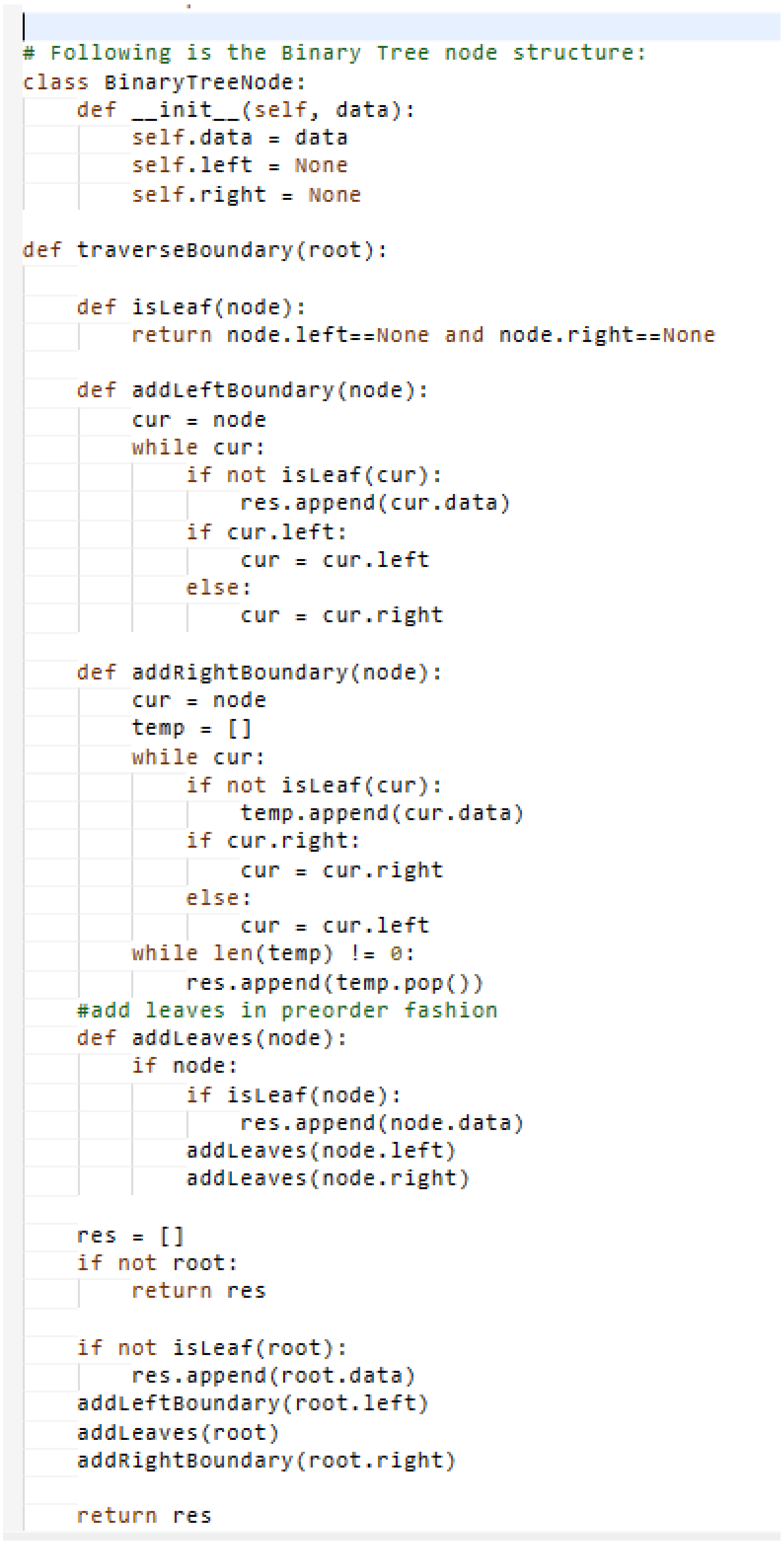
```python
class Solution:
    def buildTree(self, inorder: List[int], postorder: List[int]) -> Optional[TreeNode]:
        d={}
        for i in range(len(inorder)):
            d[inorder[i]]=i
        p_start=len(postorder)-1
        p_end=0
        i_start=0
        i_end=len(inorder)-1

        def f(p_start,p_end,i_start,i_end):
            if (p_start<p_end) or (i_start>i_end):
                return None
            root= TreeNode(postorder[p_start])
            inRoot=d[root.val]
            numsRight=i_end-inRoot
            root.left = f(p_start-numsRight-1,p_end,i_start,inRoot-1)
            root.right = f(p_start-1,p_start-numsRight,inRoot+1,i_end)
            return root
        return f(p_start,p_end,i_start,i_end)
```

Q: Children Sum Property



```python
def changeTree(root):
    if not root:
        return
    child=0
    if root.left!=None:
        child+=root.left.data
    if root.right!=None:
        child+=root.right.data
    if child>=root.data:
        root.data=child
    else:
        if root.left!=None:
            root.left.data=root.data
        elif root.right!=None:
            root.right.data=root.data
    changeTree(root.left)
    changeTree(root.right)
    tot=0
    if (root.left!=None):
        tot += root.left.data;
    if (root.right!=None):
        tot += root.right.data;
    if (root.left!=None or root.right!=None):
        root.data=tot
```

Q: Boundary traversal:



```python
# Following is the Binary Tree node structure:
class BinaryTreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def traverseBoundary(root):

    def isLeaf(node):
        return node.left==None and node.right==None

    def addLeftBoundary(node):
        cur = node
        while cur:
            if not isLeaf(cur):
                res.append(cur.data)
            if cur.left:
                cur = cur.left
            else:
                cur = cur.right

    def addRightBoundary(node):
        cur = node
        temp = []
        while cur:
            if not isLeaf(cur):
                temp.append(cur.data)
            if cur.right:
                cur = cur.right
            else:
                cur = cur.left
        while len(temp) != 0:
            res.append(temp.pop())
    #add leaves in preorder fashion
    def addLeaves(node):
        if node:
            if isLeaf(node):
                res.append(node.data)
            addLeaves(node.left)
            addLeaves(node.right)

    res = []
    if not root:
        return res

    if not isLeaf(root):
        res.append(root.data)
    addLeftBoundary(root.left)
    addLeaves(root)
    addRightBoundary(root.right)

    return res
```

POPULATE NEXT POINTER:

Example 1:

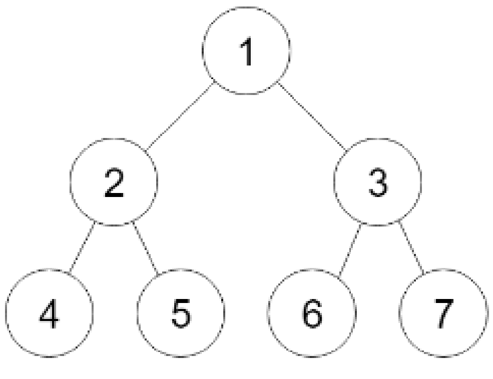


Figure A

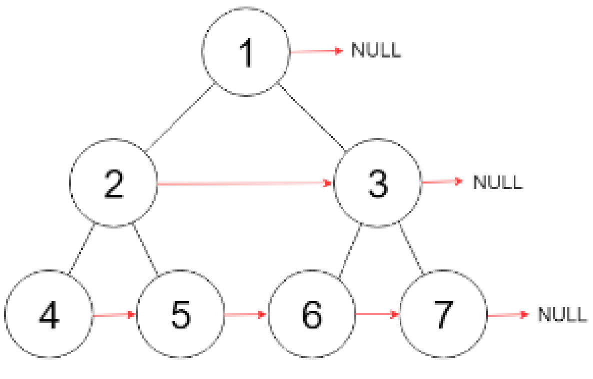
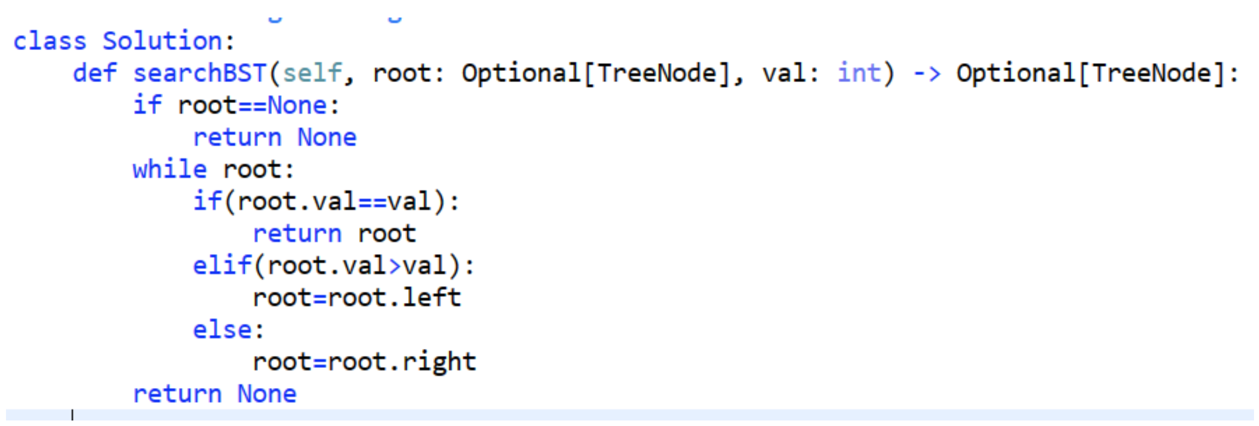


Figure B

```python
def connect(self, root: 'Node') -> 'Node':
    if not root:
        return root

    # Start with the root node
    queue = [root]

    while queue:
        # Get the number of nodes in the current level
        size = len(queue)

        # Traverse through the nodes in the current level
        for i in range(size):
            # Get the first node from the queue
            node = queue.pop(0)

            # If it's not the last node in the level, set its next to the next node in the queue
            if i < size - 1:
                node.next = queue[0]

            # Add the node's children to the queue if they exist
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

    # Return the root node
    return root
```

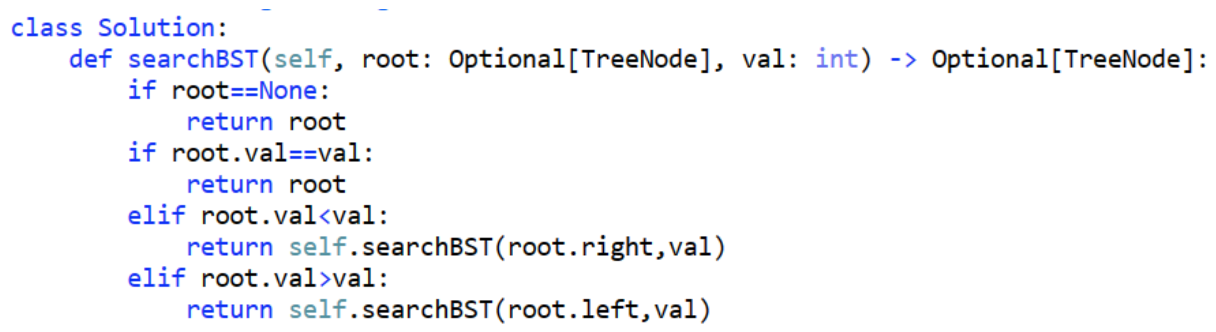
BST:

Q.Search in BST:



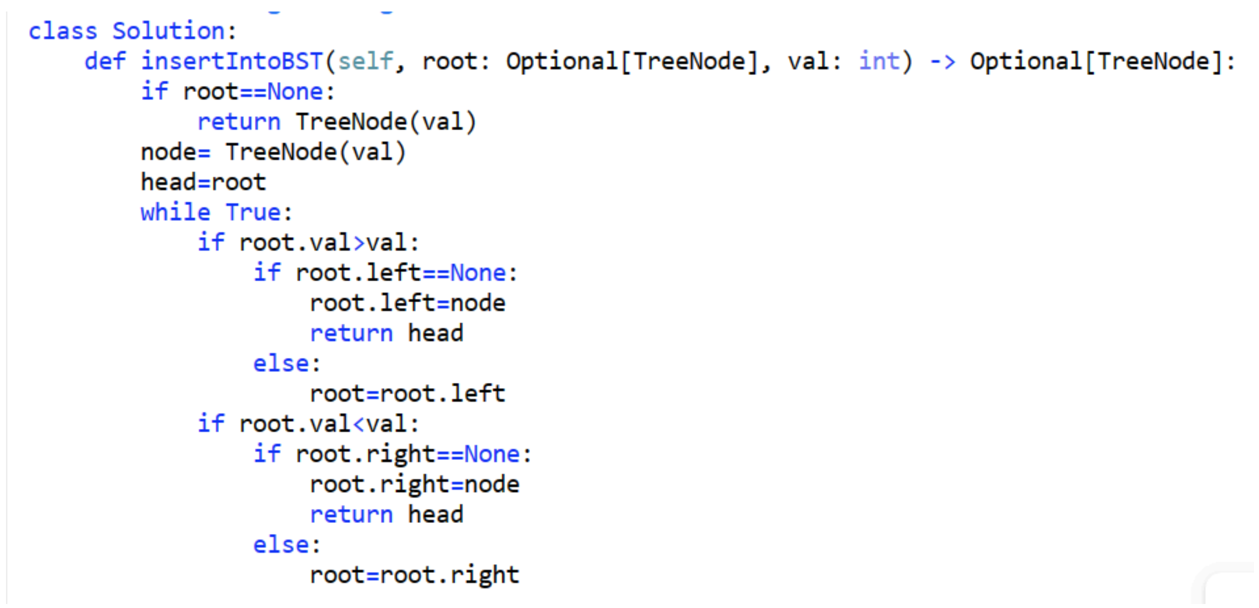
```python
class Solution:
    def searchBST(self, root: Optional[TreeNode], val: int) -> Optional[TreeNode]:
        if root==None:
            return None
        while root:
            if(root.val==val):
                return root
            elif(root.val>val):
                root=root.left
            else:
                root=root.right
        return None
```

Using recursion:



```python
class Solution:
    def searchBST(self, root: Optional[TreeNode], val: int) -> Optional[TreeNode]:
        if root==None:
            return root
        if root.val==val:
            return root
        elif root.val<val:
            return self.searchBST(root.right,val)
        elif root.val>val:
            return self.searchBST(root.left,val)
```

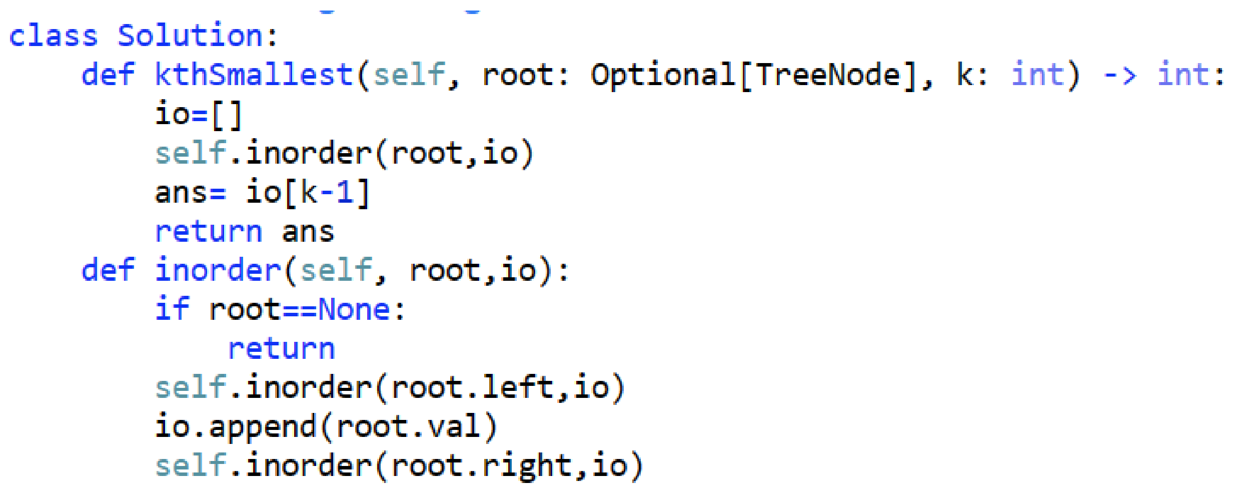
Q: Insert into a BST:



```
class Solution:
    def insertIntoBST(self, root: Optional[TreeNode], val: int) -> Optional[TreeNode]:
        if root==None:
            return TreeNode(val)
        node= TreeNode(val)
        head=root
        while True:
            if root.val>val:
                if root.left==None:
                    root.left=node
                    return head
                else:
                    root=root.left
            if root.val<val:
                if root.right==None:
                    root.right=node
                    return head
                else:
                    root=root.right
```

Q: Kth smallest in BST:

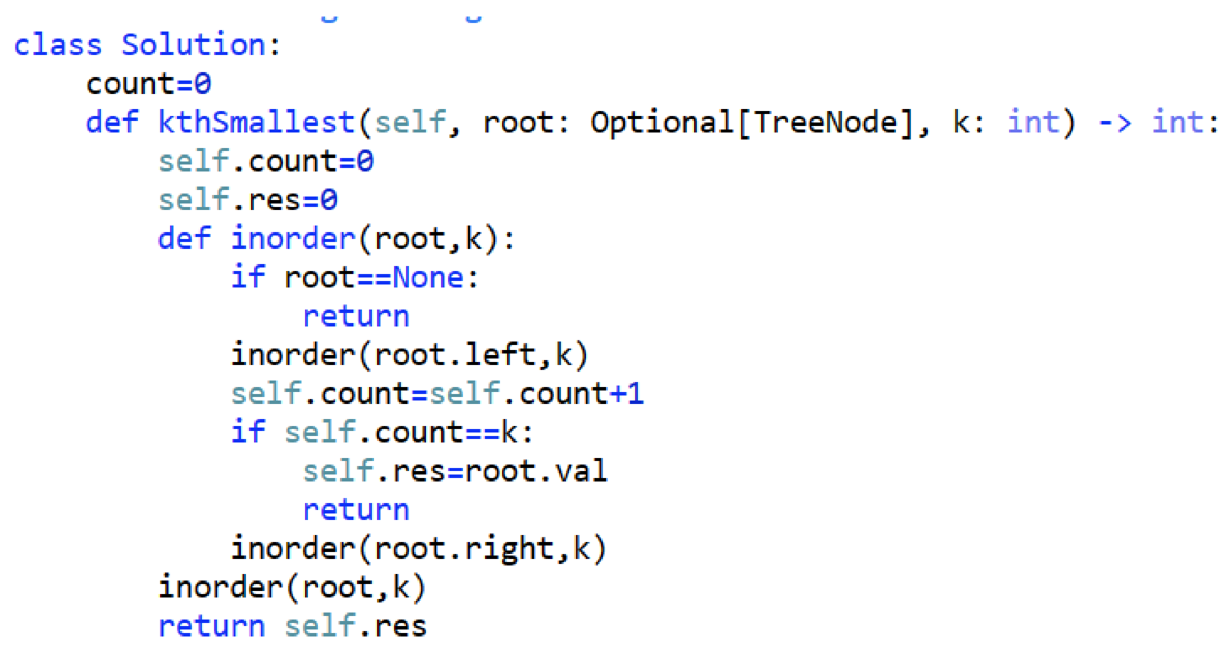
Intuition: Inorder traversal of BST gives the values in sorted form.



```
class Solution:
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        io=[]
        self.inorder(root,io)
        ans= io[k-1]
        return ans
    def inorder(self, root,io):
        if root==None:
            return
        self.inorder(root.left,io)
        io.append(root.val)
        self.inorder(root.right,io)
```

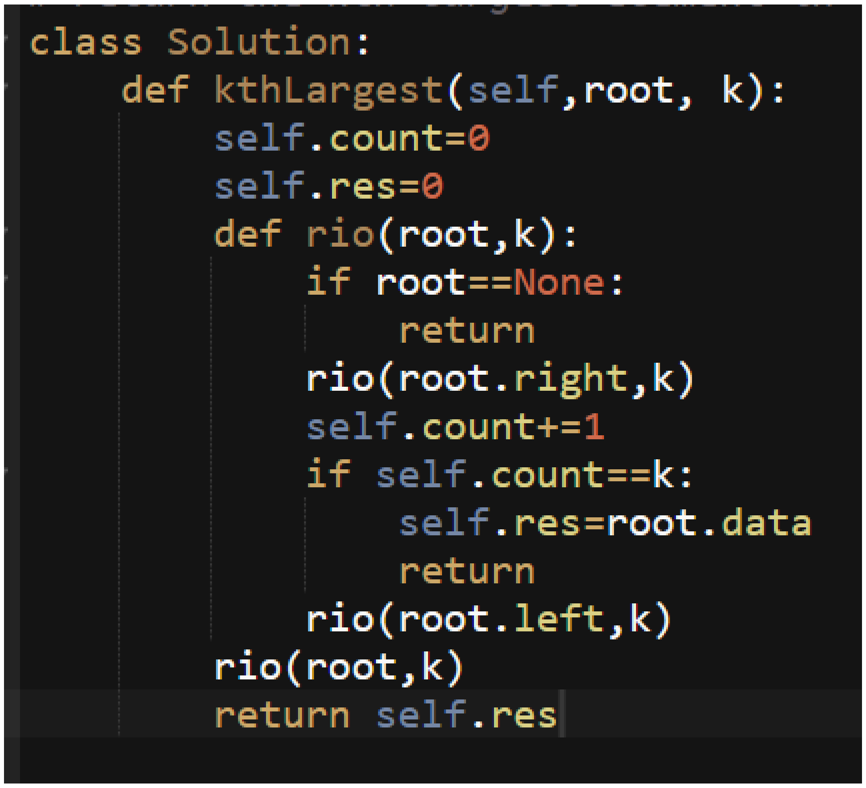
Here TC: O(N)   SC:O(N)

NOW lets see solution which take constant space: As



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

Q: kth largest in BST: Reverse of Inorder:

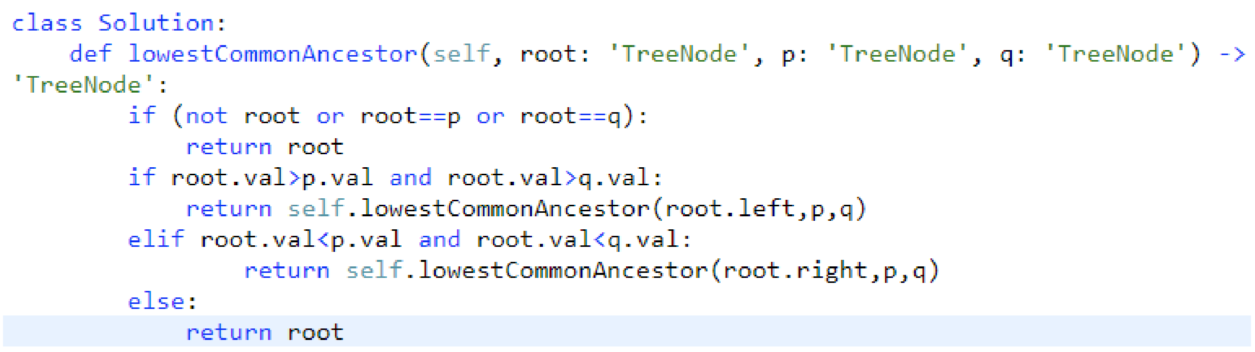


```python
class Solution:
    def kthLargest(self,root, k):
        self.count=0
        self.res=0
        def rio(root,k):
            if root==None:
                return
            rio(root.right,k)
            self.count+=1
            if self.count==k:
                self.res=root.data
                return
            rio(root.left,k)
        rio(root,k)
        return self.res
```

Q: Lowest common ancestor in BST:

Intuition: Utilize the power of BST to solve this problem . search if p and q both are less than root then ancestor must be in left subtree , if p and q both are greater than root, it must be right subtree and if both lie in two different subtree then root itself is the lowest common ancestor.

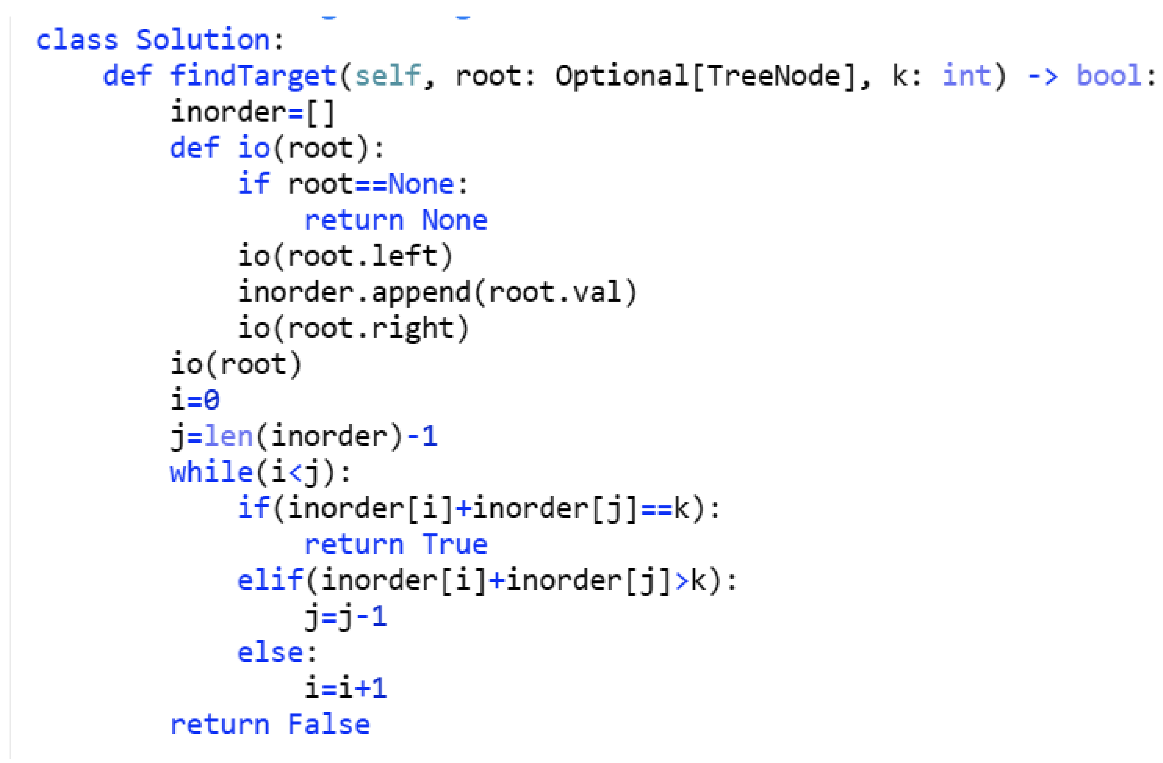
Time Complexity: O(h) Space Complexity : O(h)



```python
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') ->
'TreeNode':
        if (not root or root==p or root==q):
            return root
        if root.val>p.val and root.val>q.val:
            return self.lowestCommonAncestor(root.left,p,q)
        elif root.val<p.val and root.val<q.val:
                return self.lowestCommonAncestor(root.right,p,q)
        else:
            return root
```

Q: TWO SUM in BST:

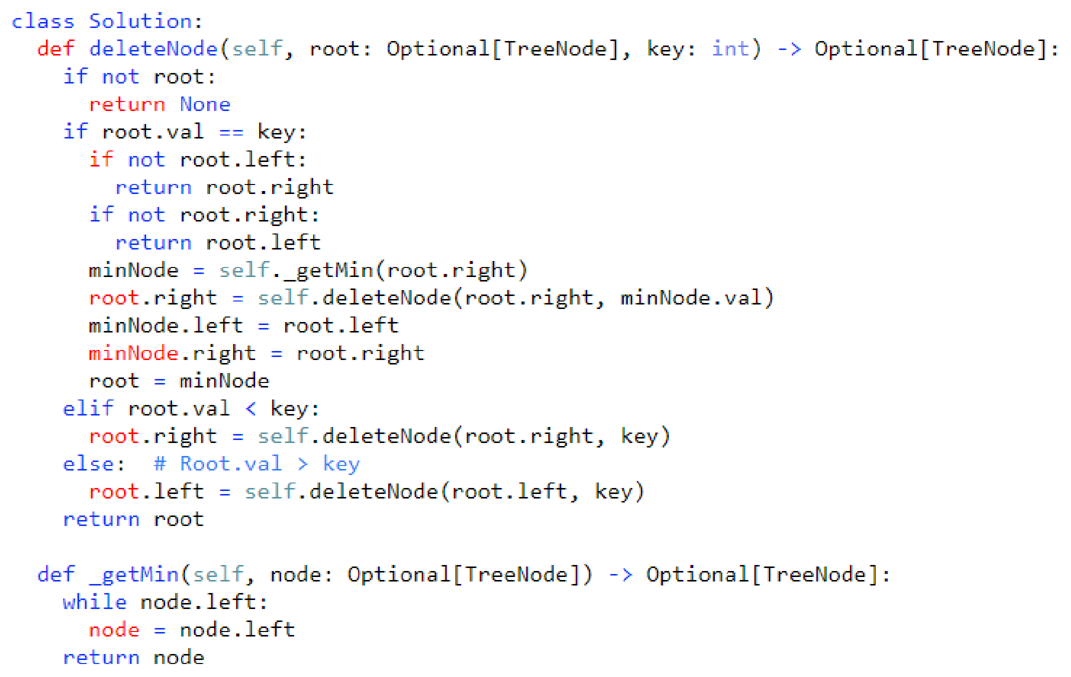
This takes TC: O(N) and SC: O(N)



```python
class Solution:
    def findTarget(self, root: Optional[TreeNode], k: int) -> bool:
        inorder=[]
        def io(root):
            if root==None:
                return None
            io(root.left)
            inorder.append(root.val)
            io(root.right)
        io(root)
        i=0
        j=len(inorder)-1
        while(i<j):
            if(inorder[i]+inorder[j]==k):
                return True
            elif(inorder[i]+inorder[j]>k):
                j=j-1
            else:
                i=i+1
        return False
```

Optimal approach : TC: O(N) and SC: O(1)

Delete a node from BST



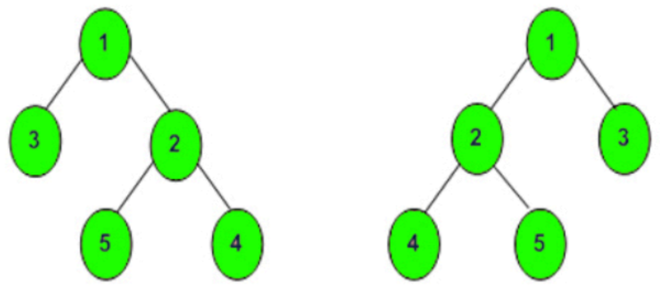
```python
class Solution:
  def deleteNode(self, root: Optional[TreeNode], key: int) -> Optional[TreeNode]:
    if not root:
      return None
    if root.val == key:
      if not root.left:
        return root.right
      if not root.right:
        return root.left
      minNode = self._getMin(root.right)
      root.right = self.deleteNode(root.right, minNode.val)
      minNode.left = root.left
      minNode.right = root.right
      root = minNode
    elif root.val < key:
      root.right = self.deleteNode(root.right, key)
    else:  # Root.val > key
      root.left = self.deleteNode(root.left, key)
    return root

  def _getMin(self, node: Optional[TreeNode]) -> Optional[TreeNode]:
    while node.left:
      node = node.left
    return node
```

TC: O(N) and SC: O(1)

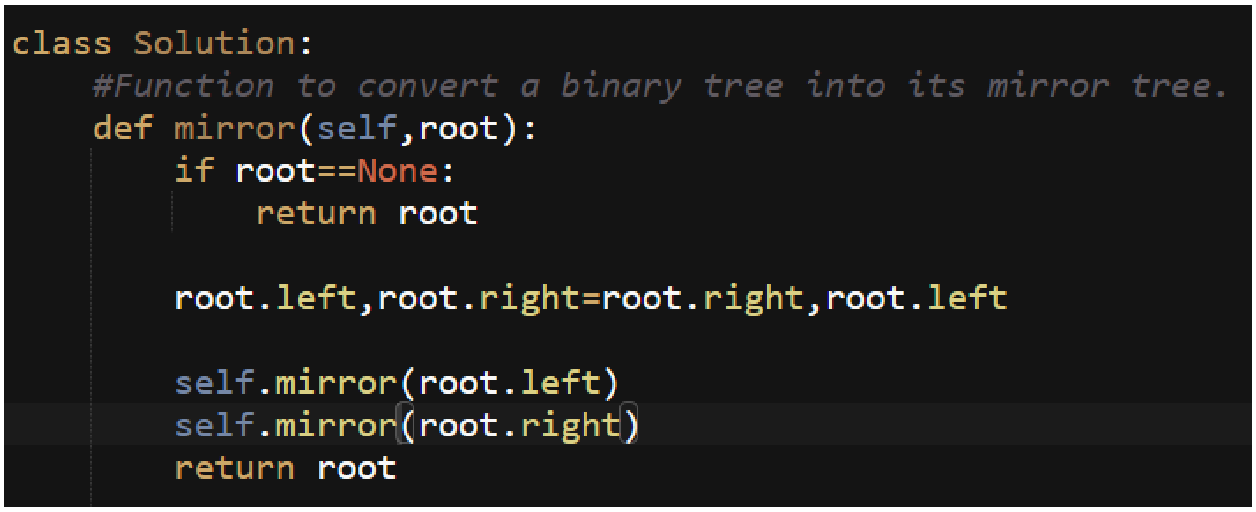
To be done

Q:

Given a Binary Tree, convert it into its mirror.



Mirror Trees

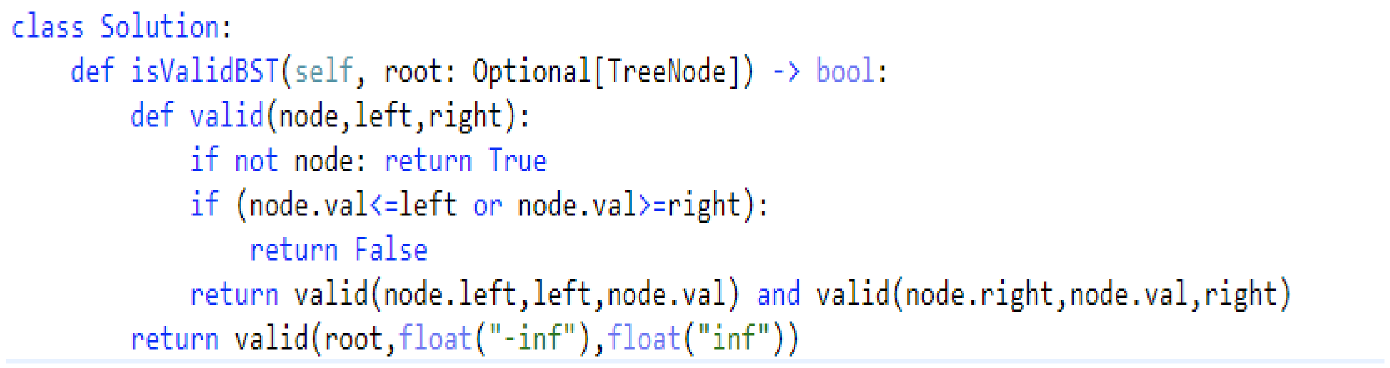


```
class Solution:
    #Function to convert a binary tree into its mirror tree.
    def mirror(self,root):
        if root==None:
            return root

        root.left,root.right=root.right,root.left

        self.mirror(root.left)
        self.mirror(root.right)
        return root
```

Validate Binary Search Tree

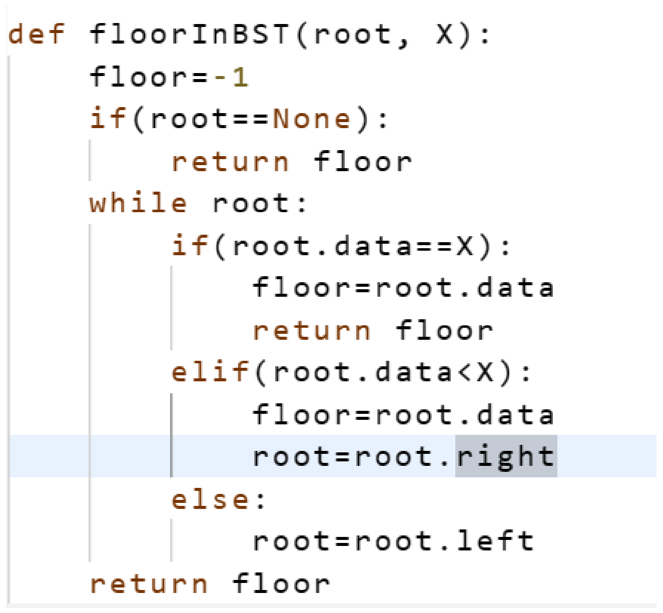
Time Complexity: O(n)   Space Complexity : O(h)

Note : here while comparing node value with left and right boundary = is also imp as tree can be 2,2,2 and it is valid BST



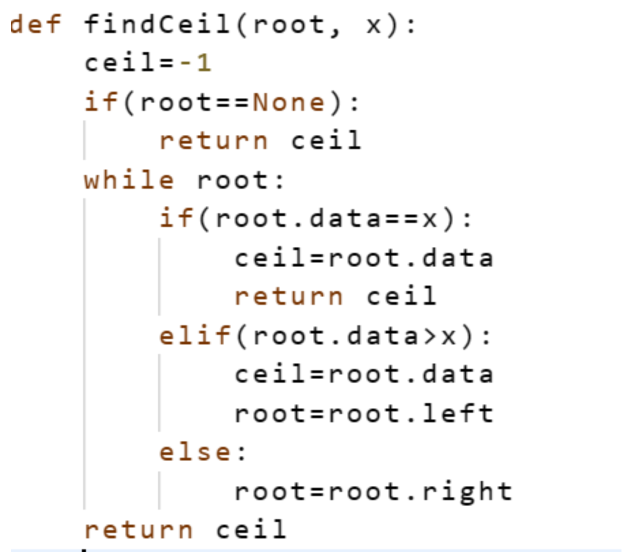
```python
class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        def valid(node,left,right):
            if not node: return True
            if (node.val<=left or node.val>=right):
                return False
            return valid(node.left,left,node.val) and valid(node.right,node.val,right)
        return valid(root,float("-inf"),float("inf"))
```

Q: Floor of BST:



```python
def floorInBST(root, X):
    floor=-1
    if(root==None):
        return floor
    while root:
        if(root.data==X):
            floor=root.data
            return floor
        elif(root.data<X):
            floor=root.data
            root=root.right
        else:
            root=root.left
    return floor
```

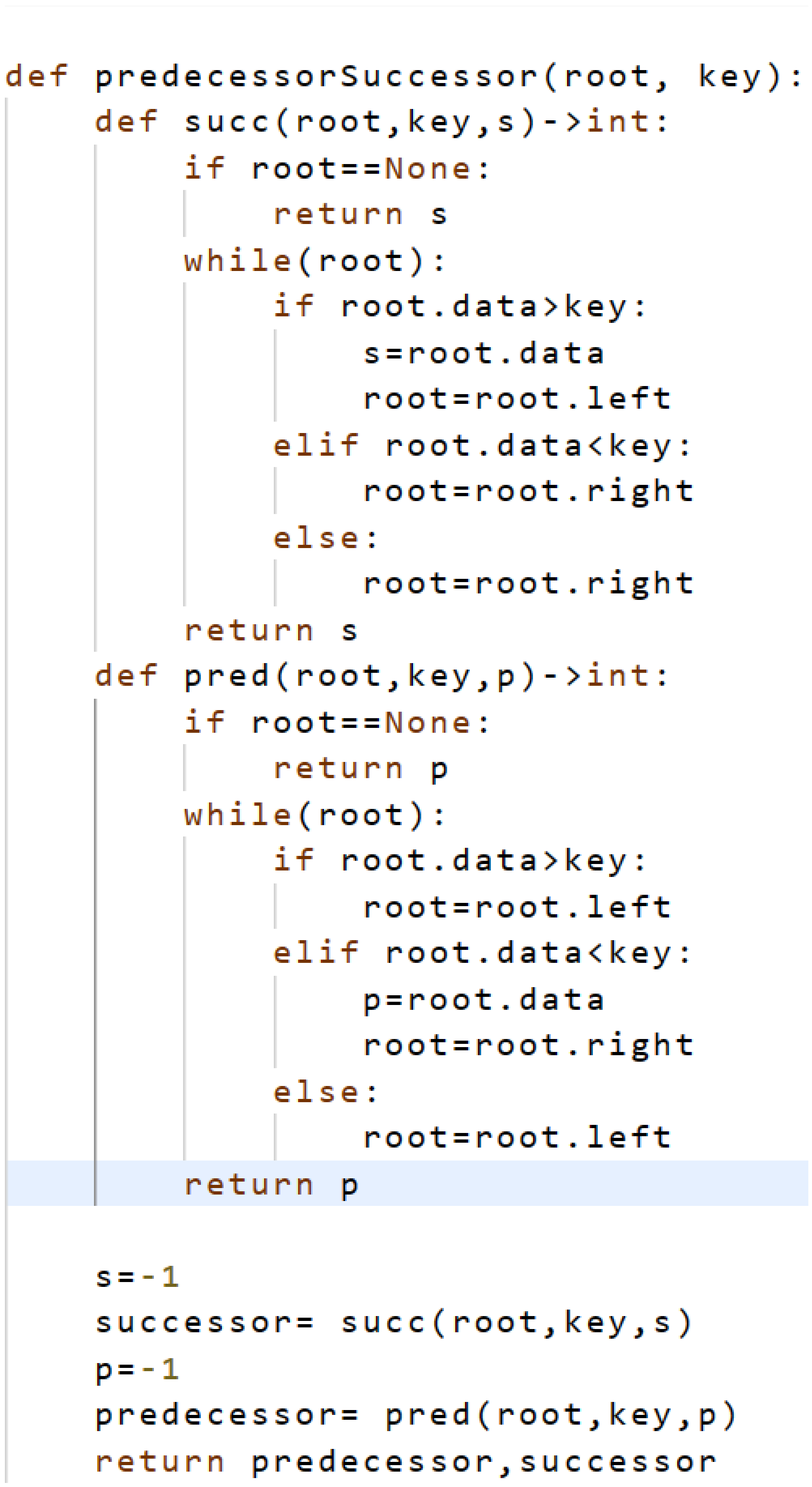
Q: Ceil of BST:



```python
def findCeil(root, x):
    ceil=-1
    if(root==None):
        return ceil
    while root:
        if(root.data==x):
            ceil=root.data
            return ceil
        elif(root.data>x):
            ceil=root.data
            root=root.left
        else:
            root=root.right
    return ceil
```

Q: Sucessor/Predecessor of BST: TC: O(H)

SC=O(1)



```python
def predecessorSuccessor(root, key):
    def succ(root,key,s)->int:
        if root==None:
            return s
        while(root):
            if root.data>key:
                s=root.data
                root=root.left
            elif root.data<key:
                root=root.right
            else:
                root=root.right
        return s
    def pred(root,key,p)->int:
        if root==None:
            return p
        while(root):
            if root.data>key:
                root=root.left
            elif root.data<key:
                p=root.data
                root=root.right
            else:
                root=root.left
        return p

    s=-1
    successor= succ(root,key,s)
    p=-1
    predecessor= pred(root,key,p)
    return predecessor,successor
```

Q: tree traversal : all in one , inorder, preorder, postorder:

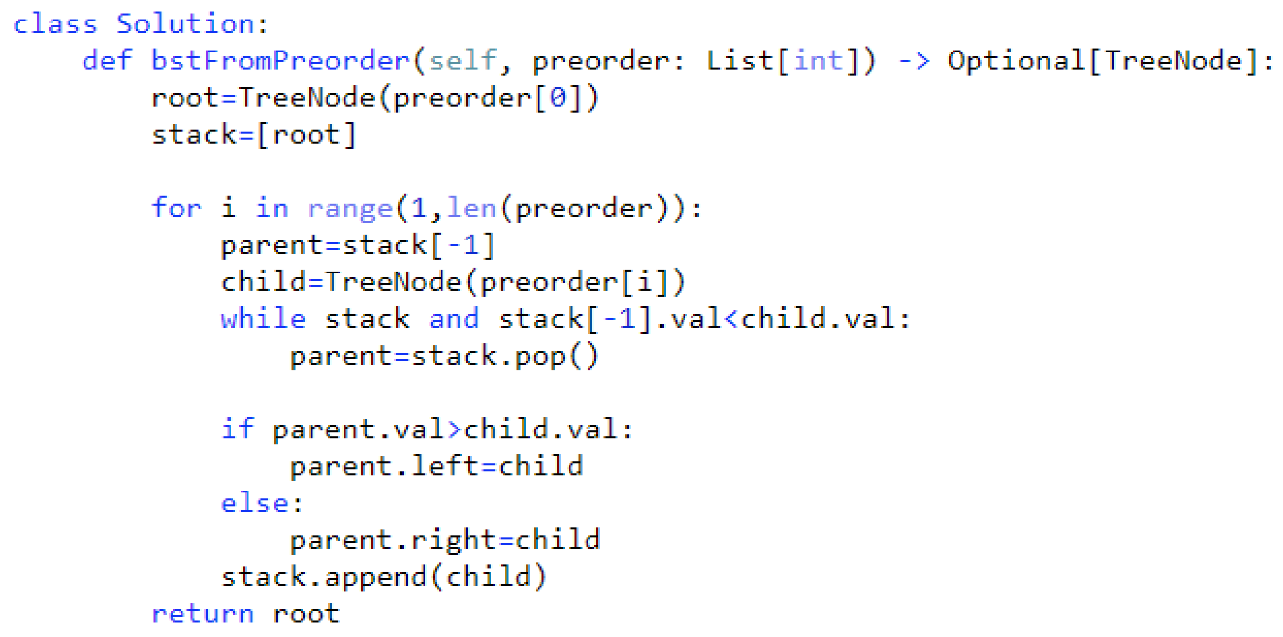
```
def getTreeTraversal(root):
    st=[]
    pre=[]
    ino=[]
    post=[]
    if root==None:
        return ino,pre,post
    st.append((root,1))
    while(st):
        node,num= st.pop()
        if num==1:
            pre.append(node.data)
            num=num+1
            st.append((node,num))
            if node.left:
                st.append((node.left,1))
        elif num==2:
            ino.append(node.data)
            num=num+1
            st.append((node,num))
            if node.right:
                st.append((node.right,1))
        else:
            post.append(node.data)
    return ino,pre,post
```

Intuition: Using stack

Q. Construct Binary Search Tree from Preorder Traversal

In this code that while loop is main key in which we are dealing the case when child value > parent value for that we are popping the element till we get some parent which is greater than child or clear stack if we didn’t get some parent which is greater than child.

Time Complexity: O(n) and Space complexity: O(h)



```python
class Solution:
    def bstFromPreorder(self, preorder: List[int]) -> Optional[TreeNode]:
        root=TreeNode(preorder[0])
        stack=[root]

        for i in range(1,len(preorder)):
            parent=stack[-1]
            child=TreeNode(preorder[i])
            while stack and stack[-1].val<child.val:
                parent=stack.pop()

            if parent.val>child.val:
                parent.left=child
            else:
                parent.right=child
            stack.append(child)
        return root
```

Another approach using recursion:

```
class Solution:
    i=0
    def bstFromPreorder(self, preorder: List[int]) -> Optional[TreeNode]:
        intmax = float('inf')
        return self.build(preorder, intmax)

    def build(self, preorder, bound):
        if len(preorder) == self.i or preorder[self.i] > bound:
            return None
        root = TreeNode(preorder[self.i])
        self.i += 1
        root.left= self.build(preorder, root.val)
        root.right = self.build(preorder, bound)
        return root
```

Q : Convert Sorted Array to Binary Search Tree

```
class Solution(object):
    def sortedArrayToBST(self, nums):
        # Base condition...
        if len(nums) == 0:
            return None
        # set the middle node...
        mid = len(nums)//2
        # Initialise root node with value same as nums[mid]
        root = TreeNode(nums[mid])
        # Assign left subtrees as the same function called on left subranges...
        root.left = self.sortedArrayToBST(nums[:mid])
        # Assign right subtrees as the same function called on right subranges...
        root.right = self.sortedArrayToBST(nums[mid+1:])
        # Return the root node...
        return root
```

Q: Check whether given tree is BST or not:

```python
class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:

        def f(root,minval,maxval):
            if root==None:
                return True
            if root.val<=minval or root.val>=maxval:
                return False
            return f(root.left,minval,root.val) and f(root.right,root.val,maxval)

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