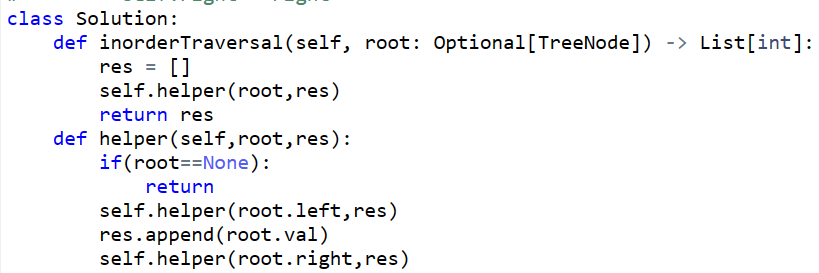
**TREE**

**Tree traversal: BFS (LEVEL ORDER)**

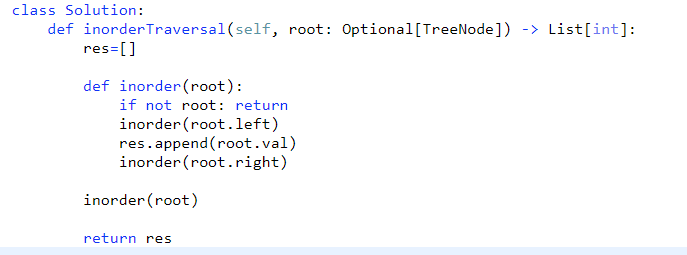
**DFS (PREORDER, POSTORDER, INORDER)**

**RECURSIVE SOLUTION:**

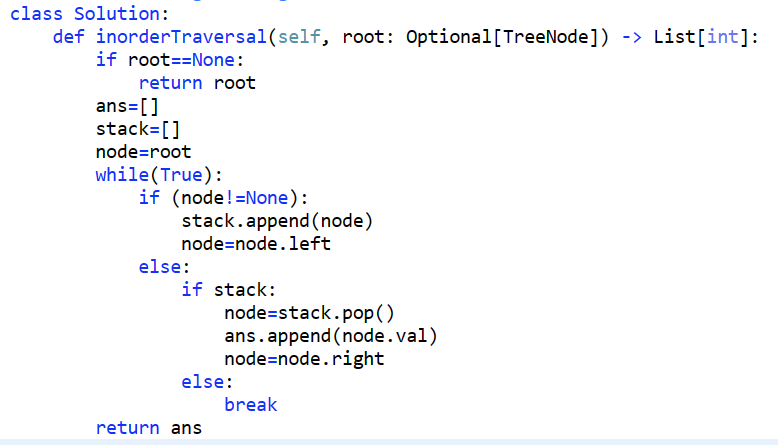
* **INORDER: LEFT ROOT RIGHT**

****

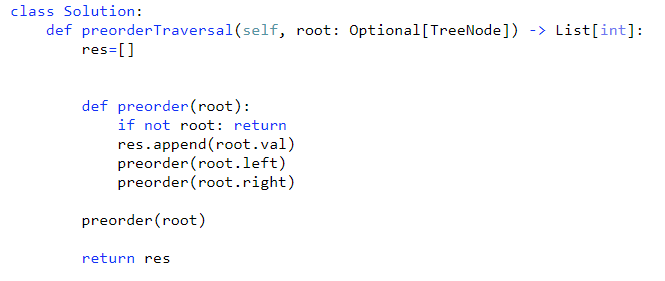
**Another Soln :**

****

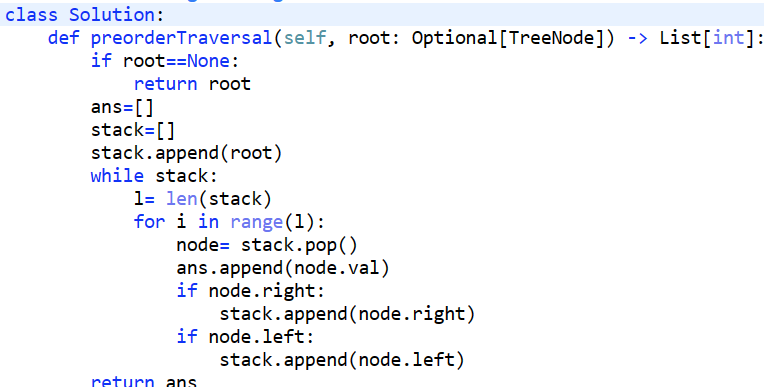
**ITERATIVE INORDER SOLUTION:**

****

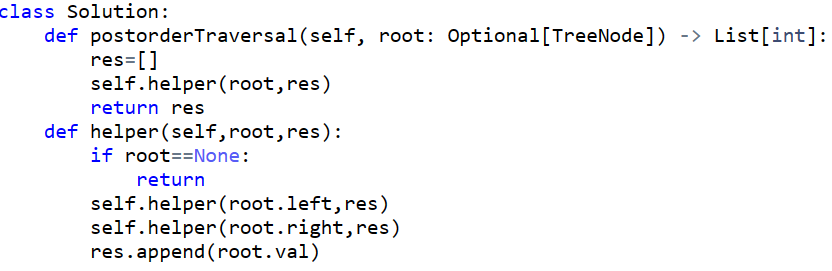
* **PREORDER : ROOT LEFT RIGHT**

****

**ITERATIVE SOLUTION:PREORDER**

****

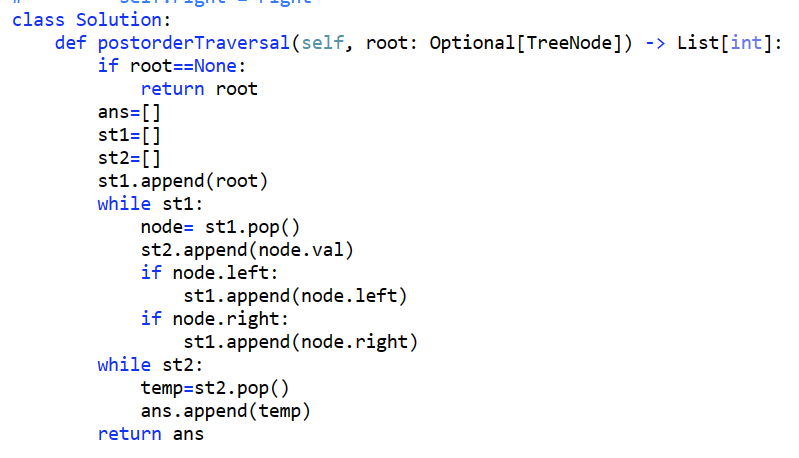
* **POSTORDER: LEFT RIGHT ROOT**

****

**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:**

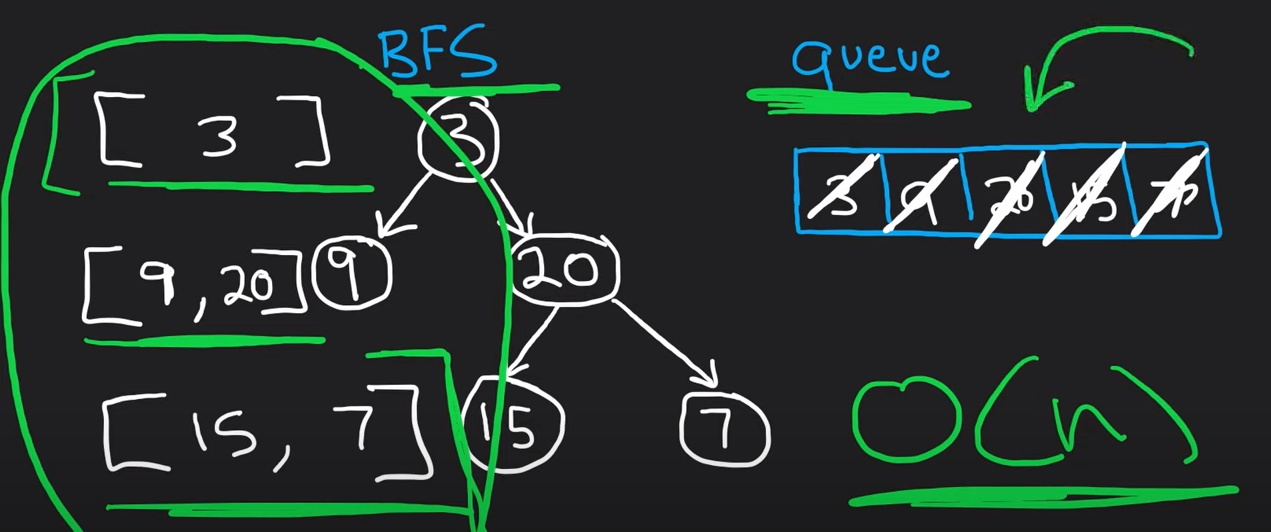
* **Using 2 stacks:**

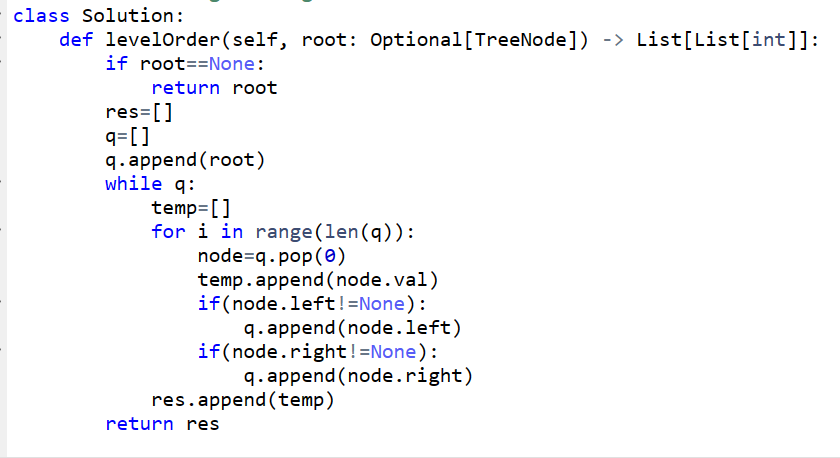


* **Using 1 stack:**

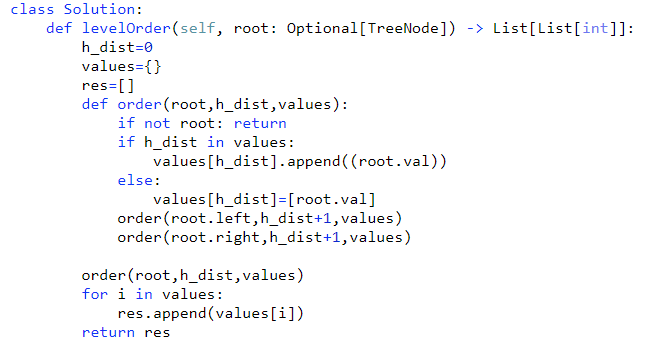
**LEVEL ORDER(BFS) TRAVERSAL:**

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

****

****

**Recursive approach:**

****

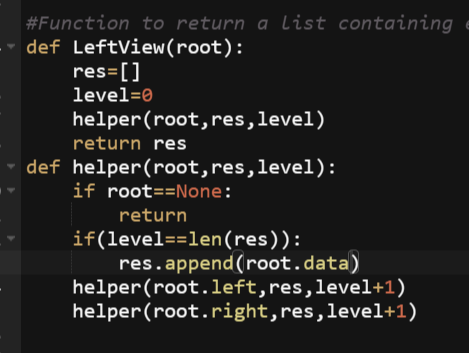
**Q1: Left view of tree:**

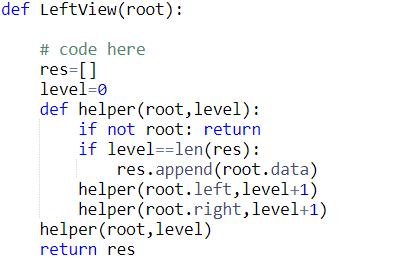
**Intuition:** Traverse like pre order traversal, **ROOT LEFT RIGHT**

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



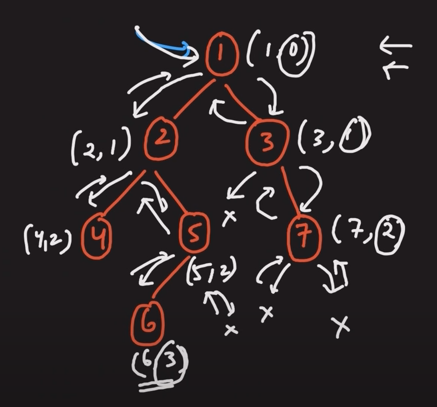


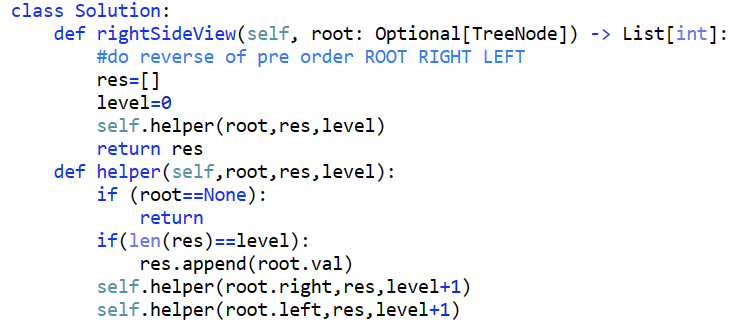
**Q2: Right view of tree:**

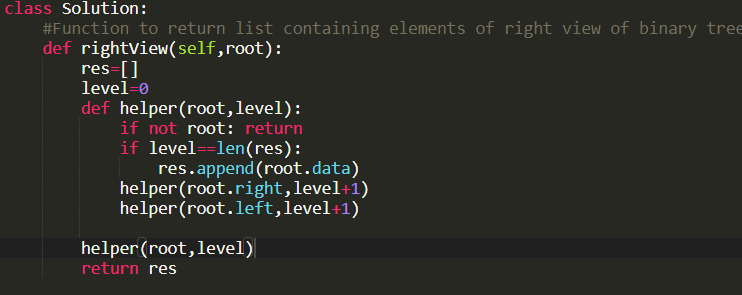
Intuition: its now reverse of pre-order traversal, **ROOT RIGHT LEFT**

TC:O(N)

SC:O(H) H=height of tree

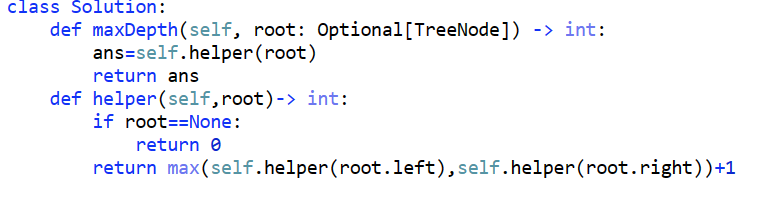




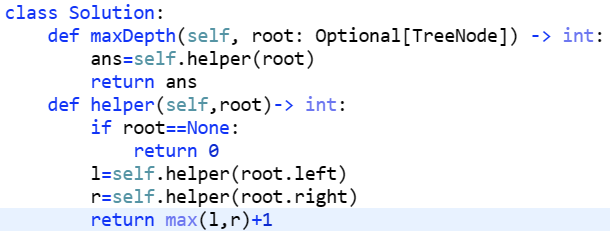
88

**Q3:Finding depth of tree:**

**Using recursion:**

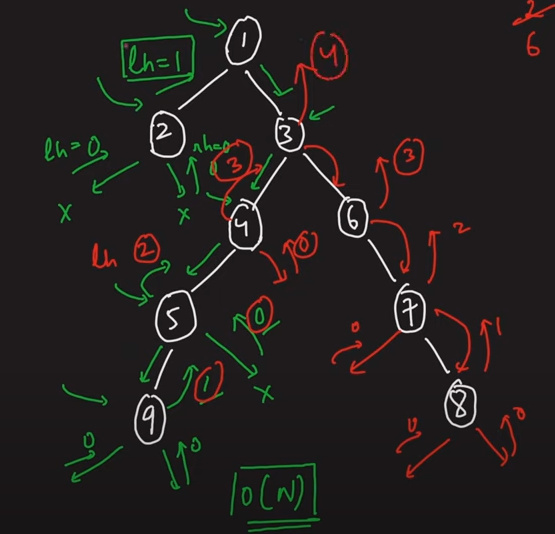


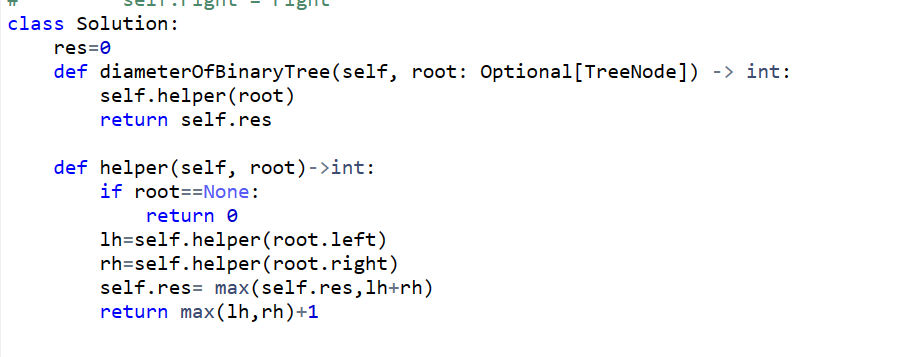
Just another variation in code to understand the logic, nothing different from above code:

****

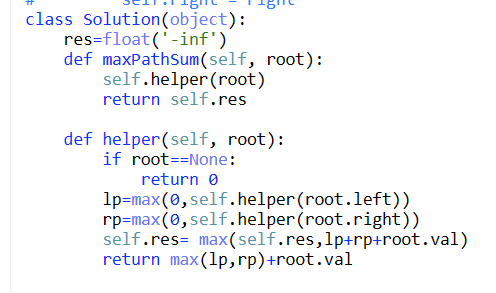
**Q4: Diameter of Tree:**

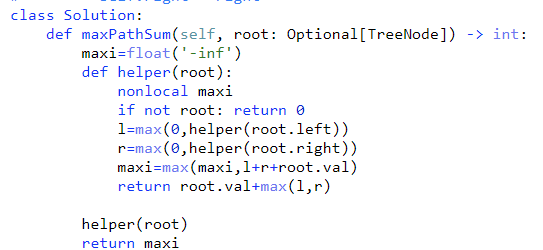
**Diameter:** Longest path between two nodes which need not to be passed through the root.



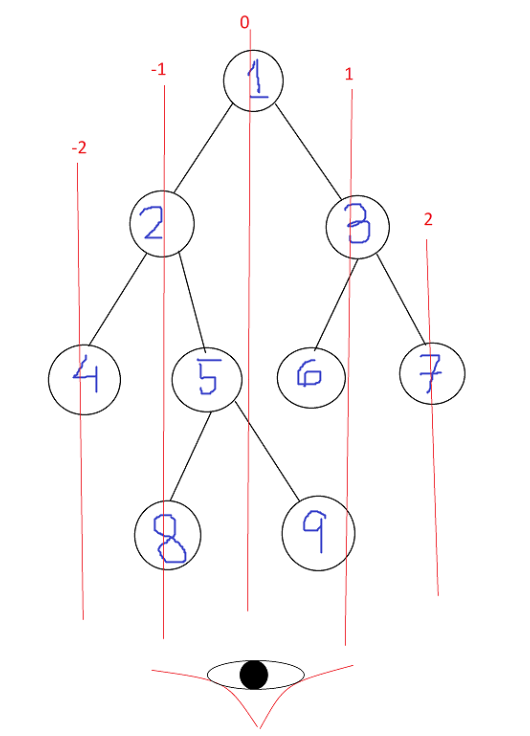


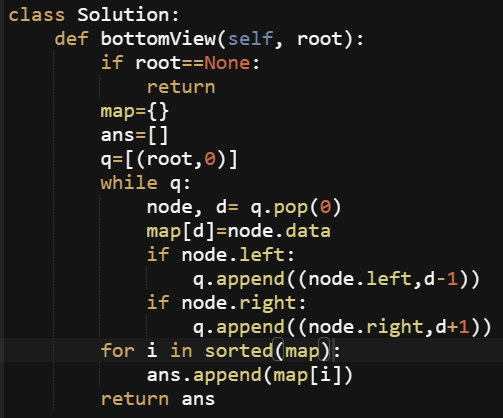
**Q: MAX PATH SUM:**



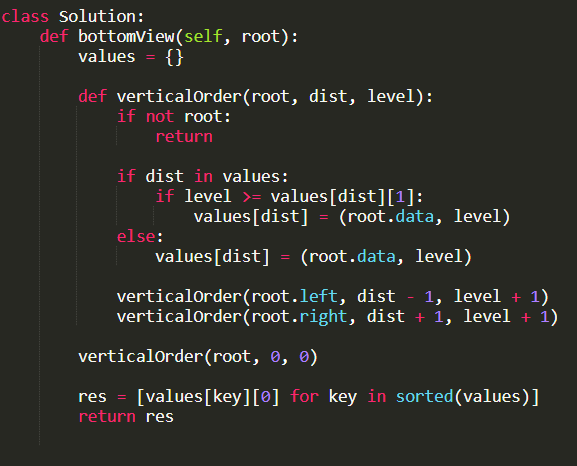


**Q5: Bottom view of tree:**

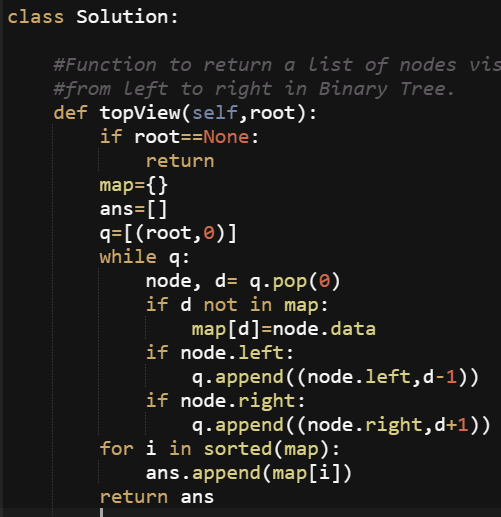




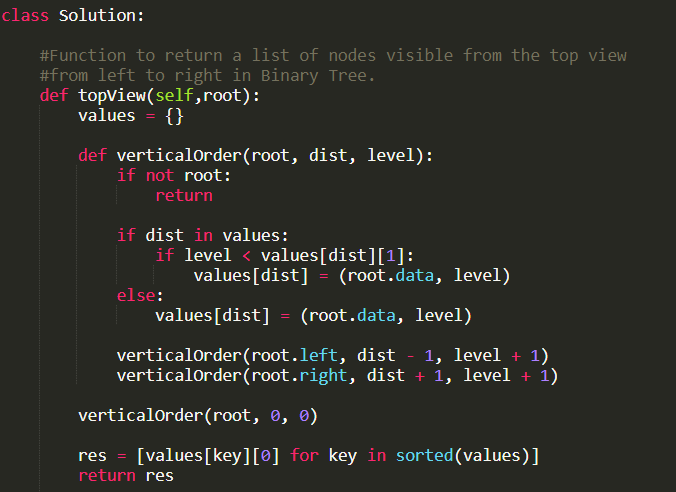
**Recursive approach:**



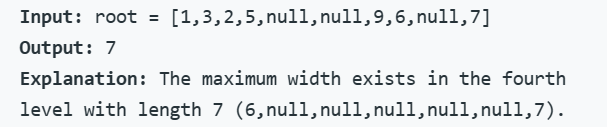
Q6: **Top view of tree:**

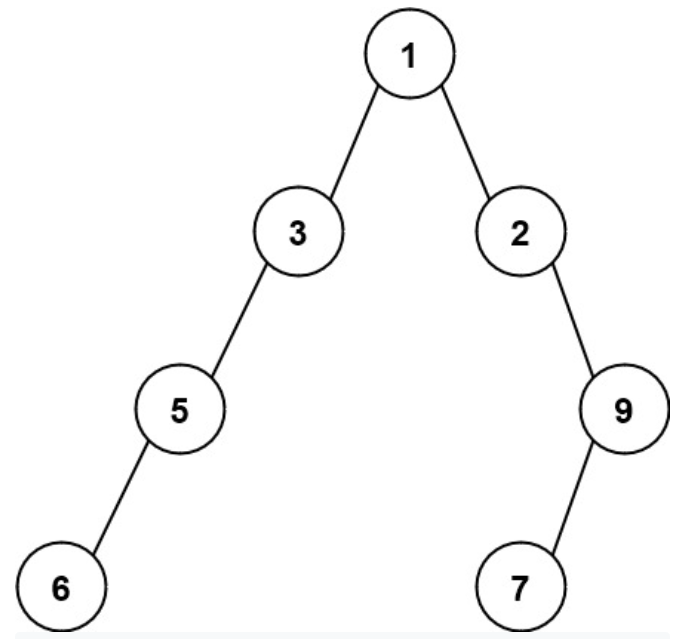


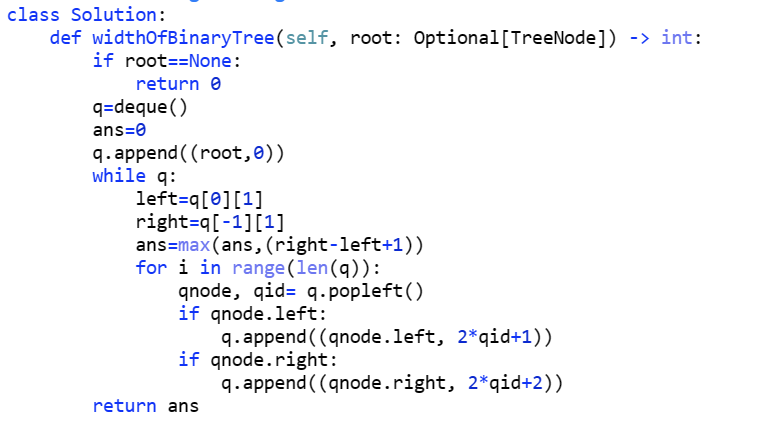
Recursive Approach:



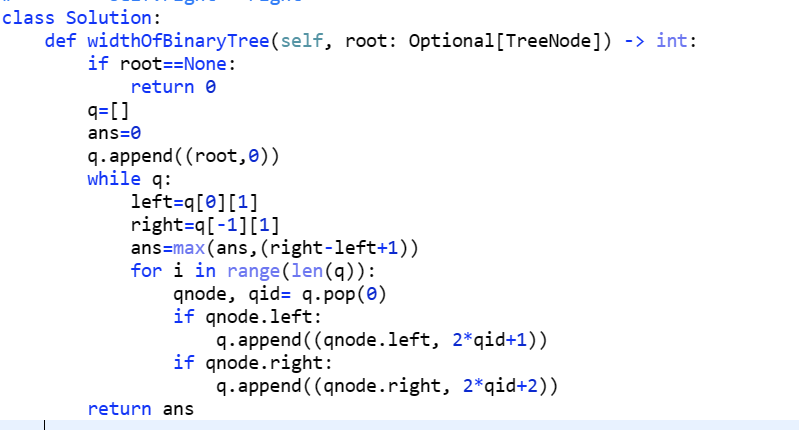
**Q7: Width of binary tree:**







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.



**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.

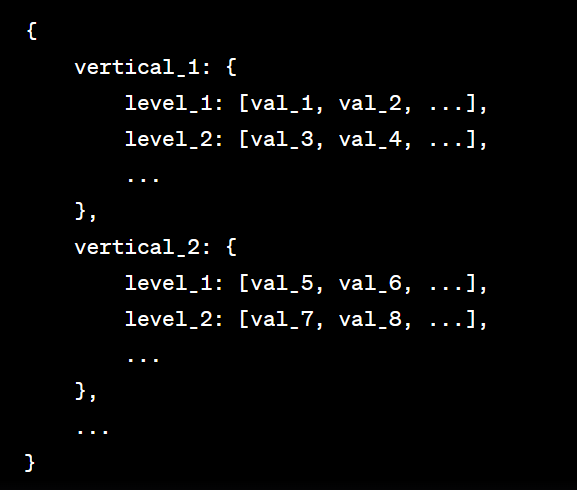
* Time Complexity:
* Appending and popping from both ends (append, appendleft, pop, popleft): O(1)
* Accessing elements by index (deque[i]): O(1)
* Removing an arbitrary element (remove): O(n)
* Insertion or deletion in the middle: O(n)
* Searching for an element: O(n)
* Space Complexity: O(n) - the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



**Intuition:** We take a queue and filled it with (nodeval, vertical, level)

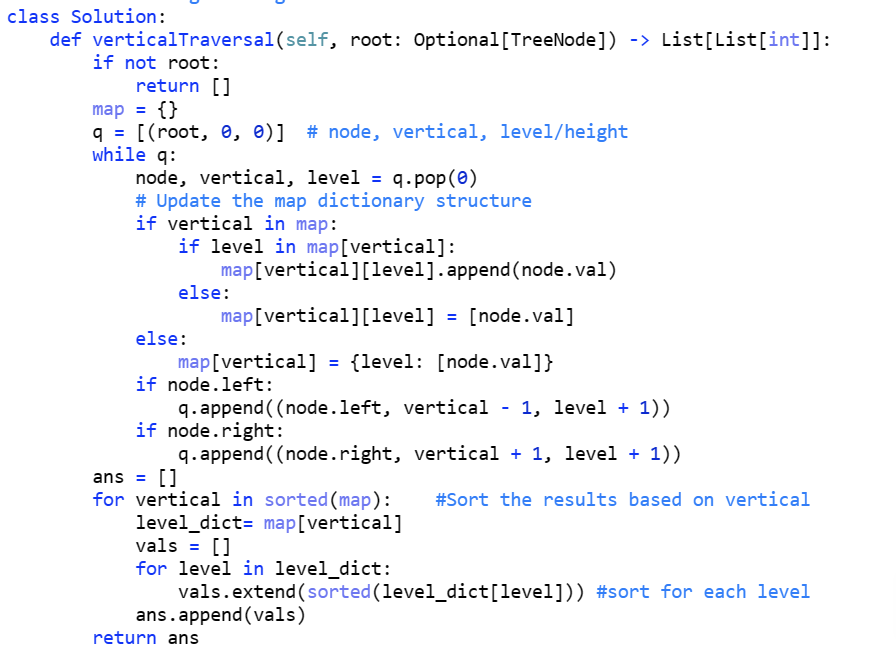
Initially , it is (root, 0, 0)

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



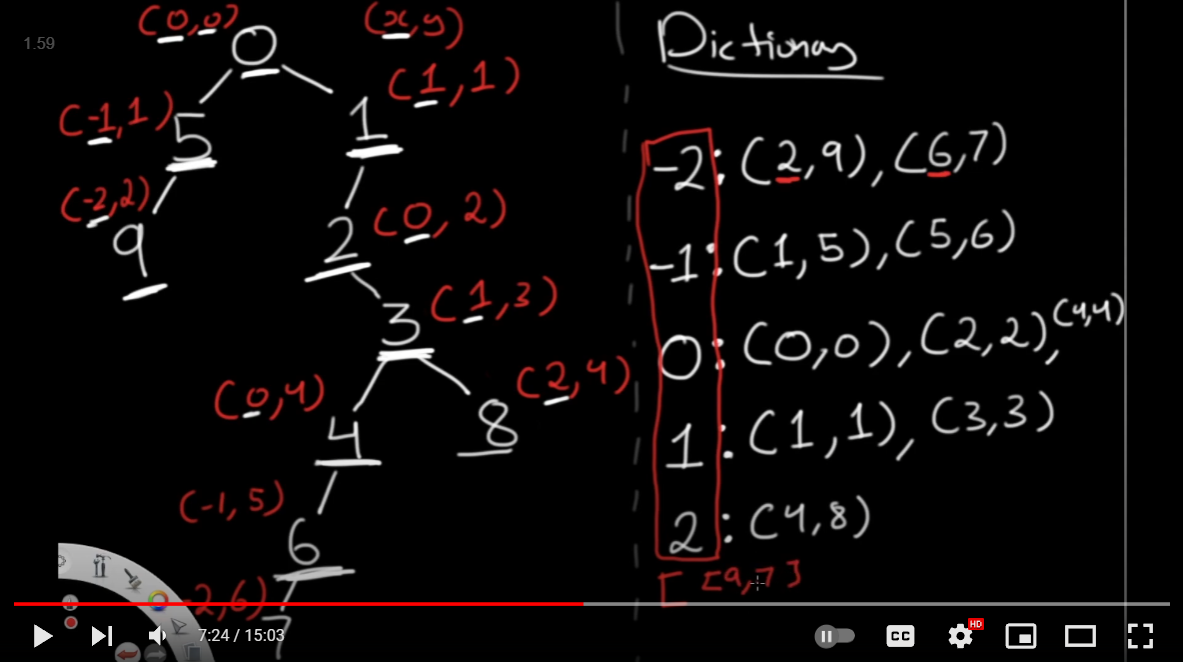
**Another soln:**

**Reference: https://www.youtube.com/watch?v=xs\_deEJXflw**

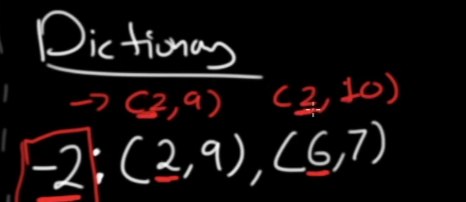
In this soln you have to be clear about one thing thatif at same horizontal distance two values are coming then we do not have to sort them in increasing order **but when at same horizontal distance and at same vertical distance then we have to sort in ascending order**

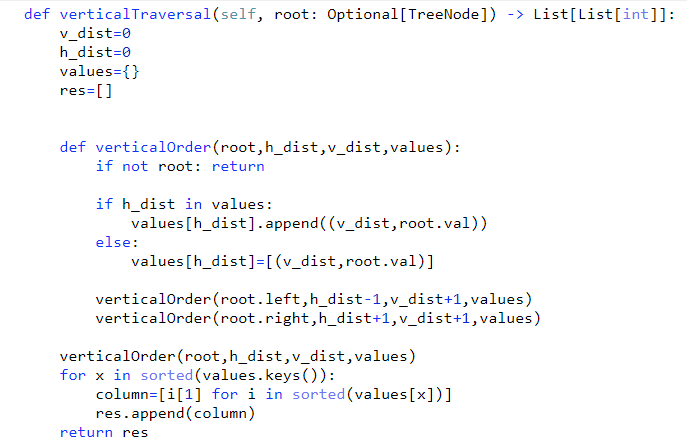
**Eg:**

**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)**

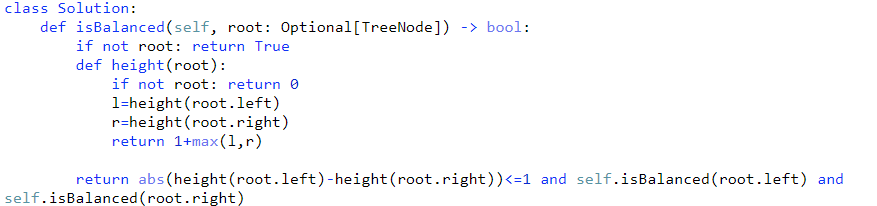
****

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

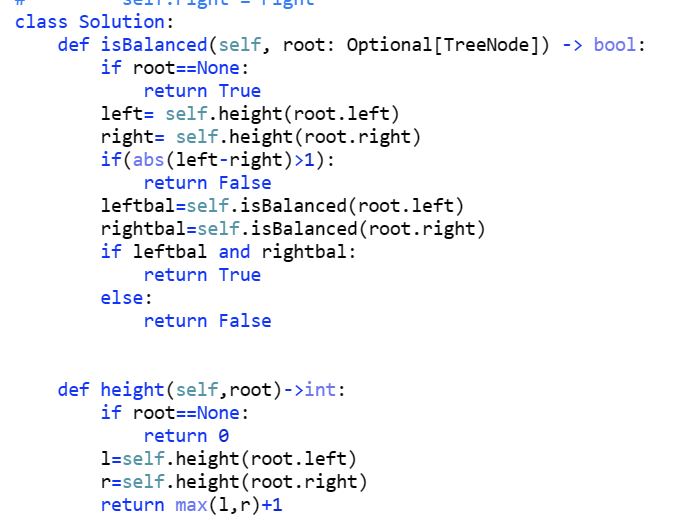
****

****

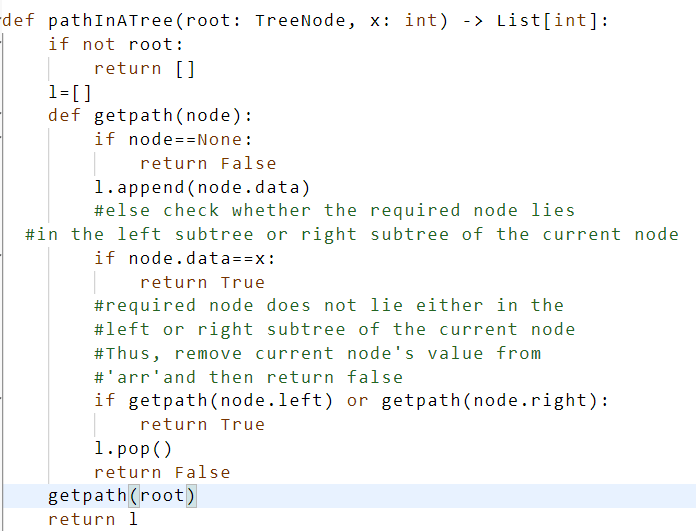
**110. Balanced Binary Tree**

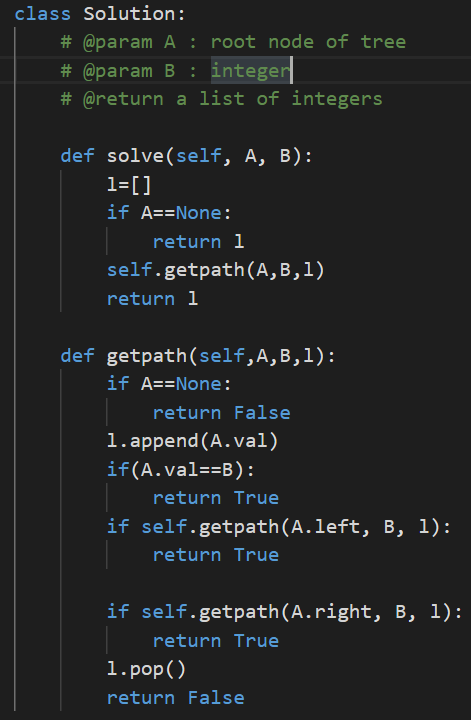


Another solution you can look at:



**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.





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

**Q;** [**Morris Inorder Traversal**](https://takeuforward.org/data-structure/morris-inorder-traversal-of-a-binary-tree/) **(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**

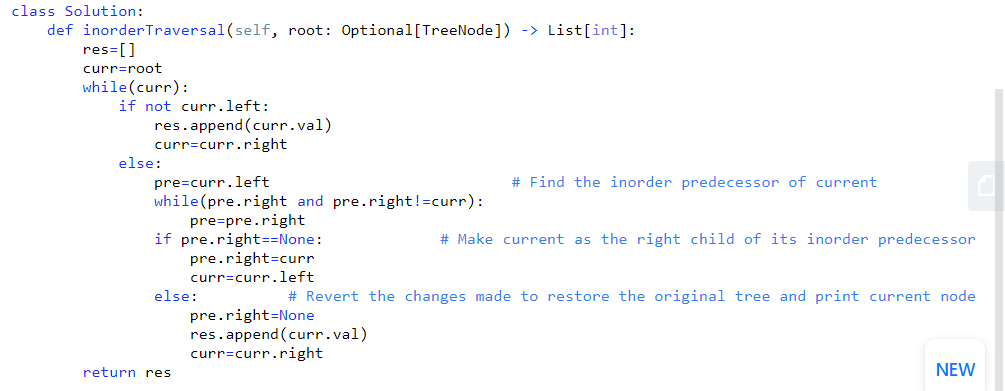
**c) Go to the right, i.e. current = current->right**

**Else**

**a) Make current as the right child of that rightmost**

**node we found; and**

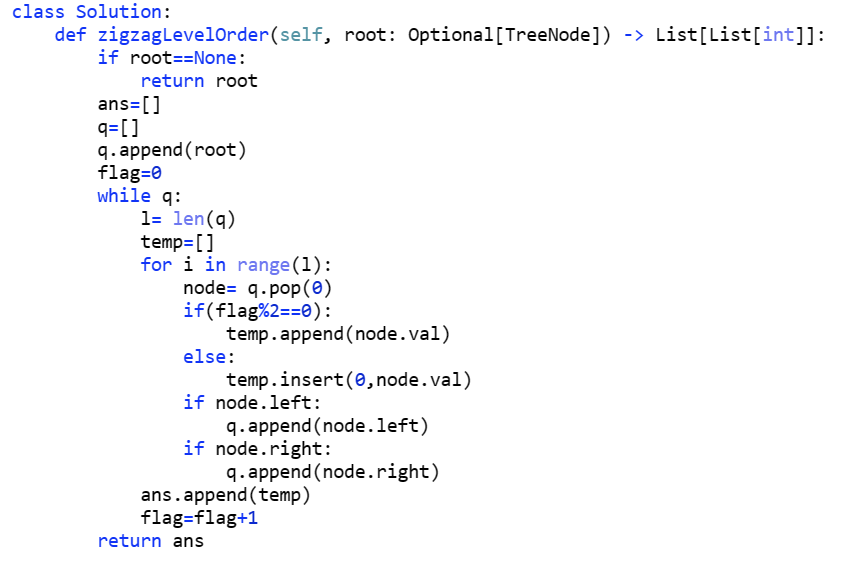
**b) Go to this left child, i.e., current = current->left**

****

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

**Q: Zig-Zag traversal:**

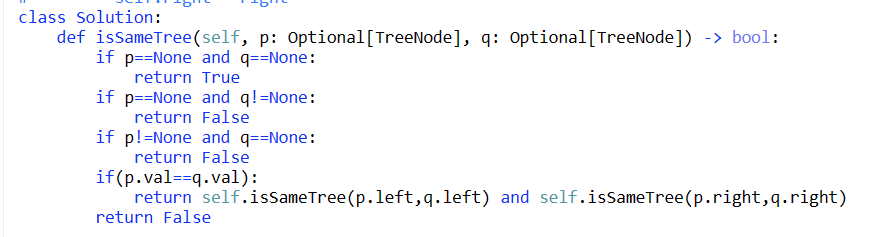
**Intuition: just like level order traversal , only introduce the flag to keep track of each alternative level.**

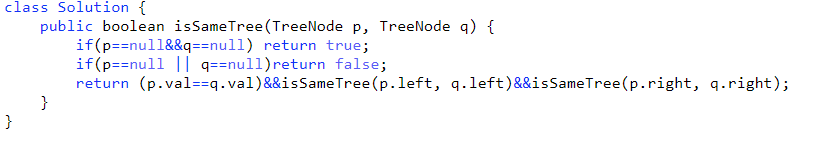
****

**Note:** To insert any value in list , we use insert(index,value\_to\_insert)

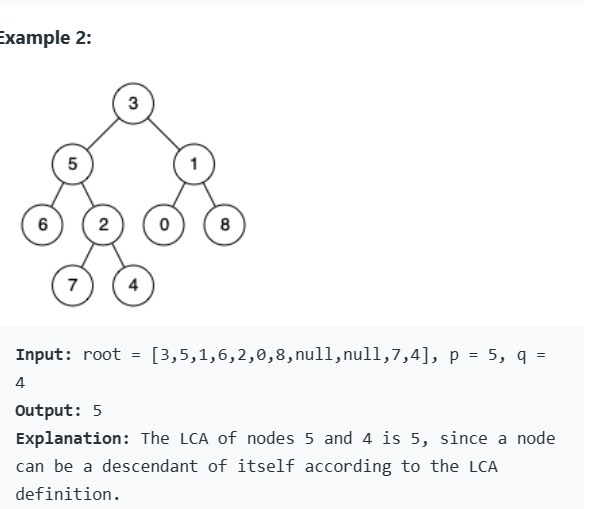
To insert at back of list, we use list\_name.append(value)

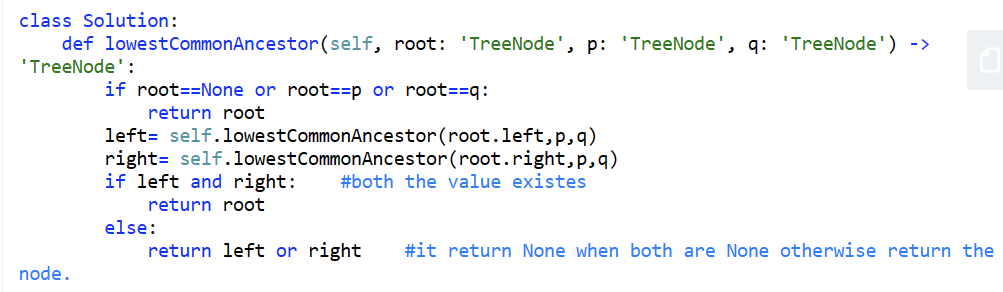
**Q: Same Tree:**



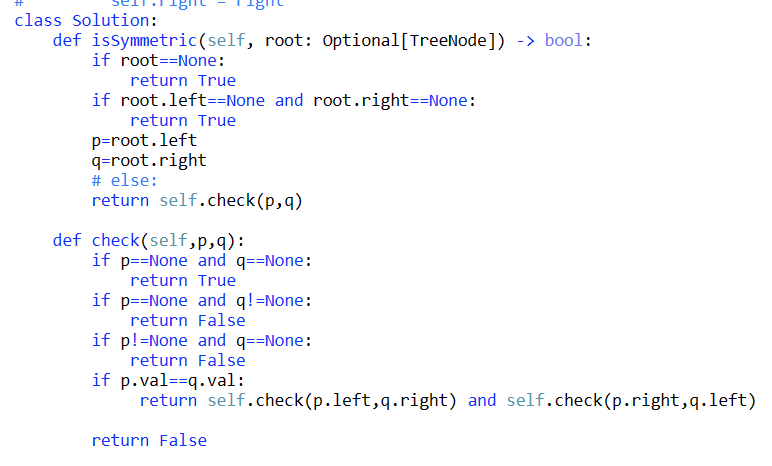


**Q: Lowest Common Ancestor of a Binary Tree**

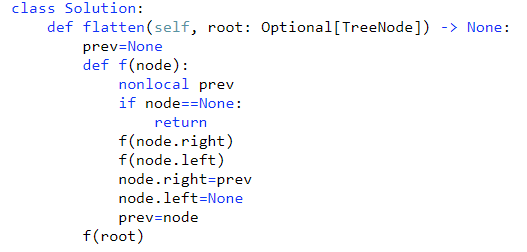
****

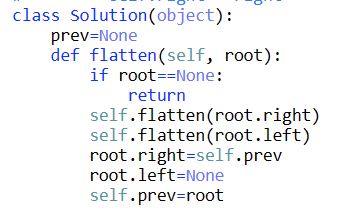


**Q: Symmetric Tree:**

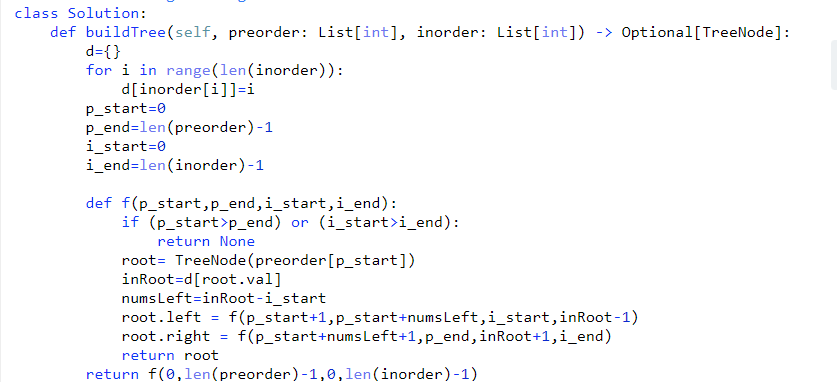


**Q. Flatten Binary Tree to Linked List**

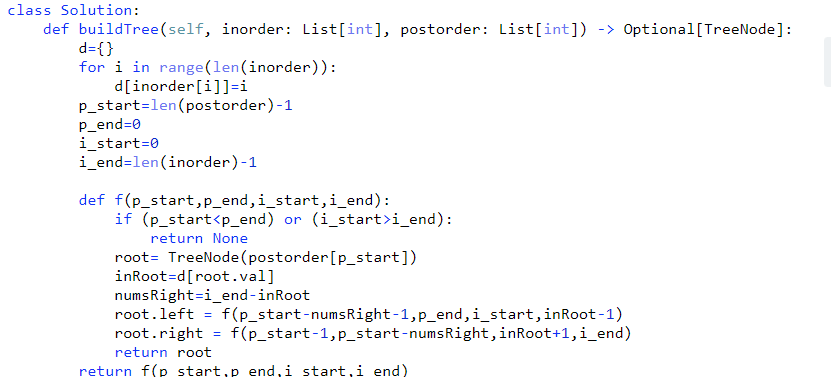
****

****

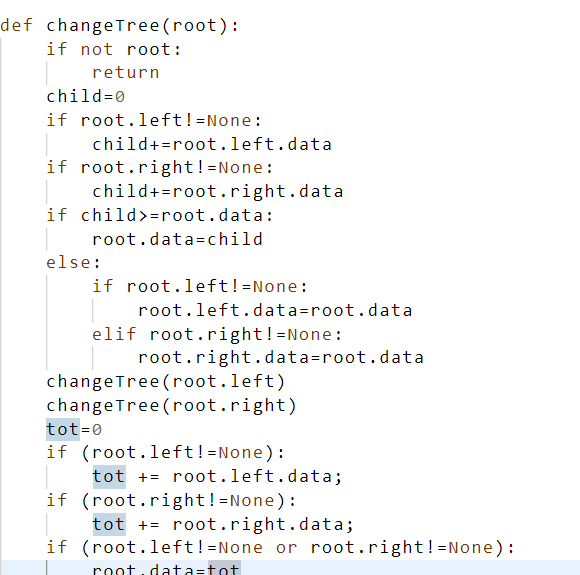
**Q Construct Binary Tree from Preorder and Inorder Traversal**

****

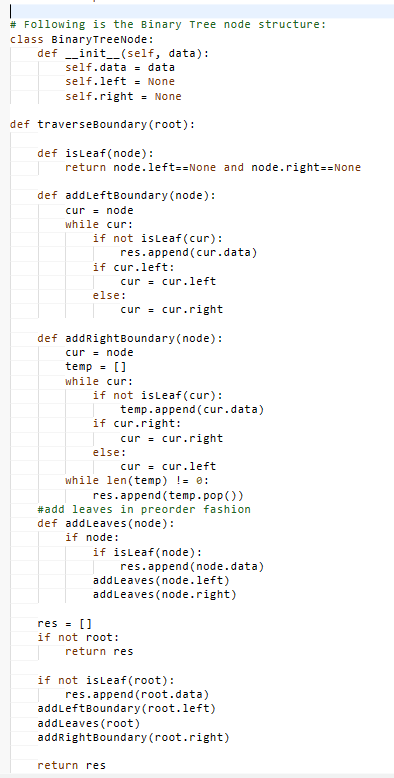
**Q**  **Construct Binary Tree from Inorder and Postorder Traversal**



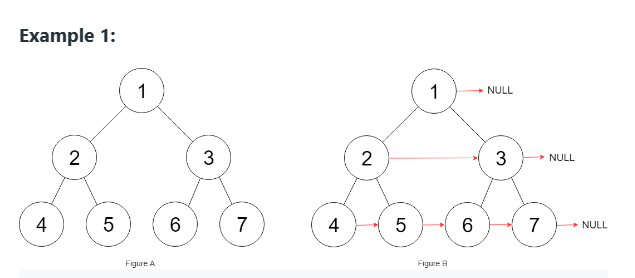
**Q: Children Sum Property**

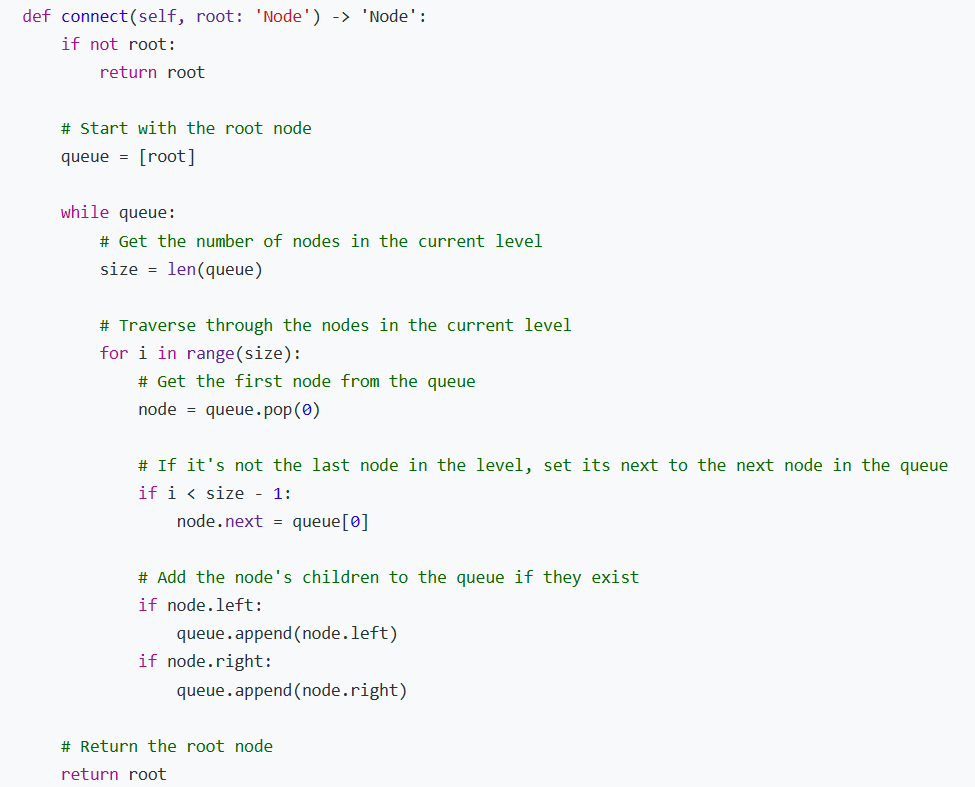


**Q: Boundary traversal:**



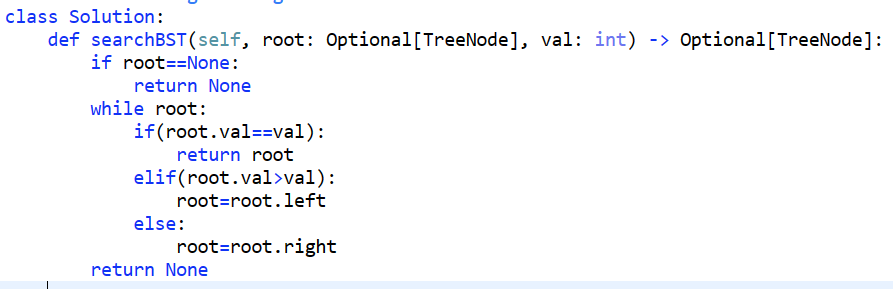
POPULATE NEXT POINTER:



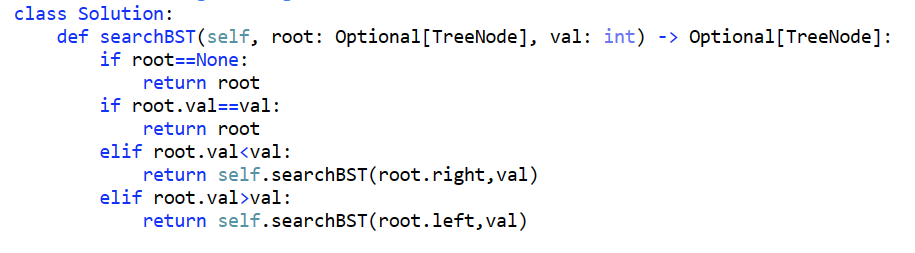


**BST:**

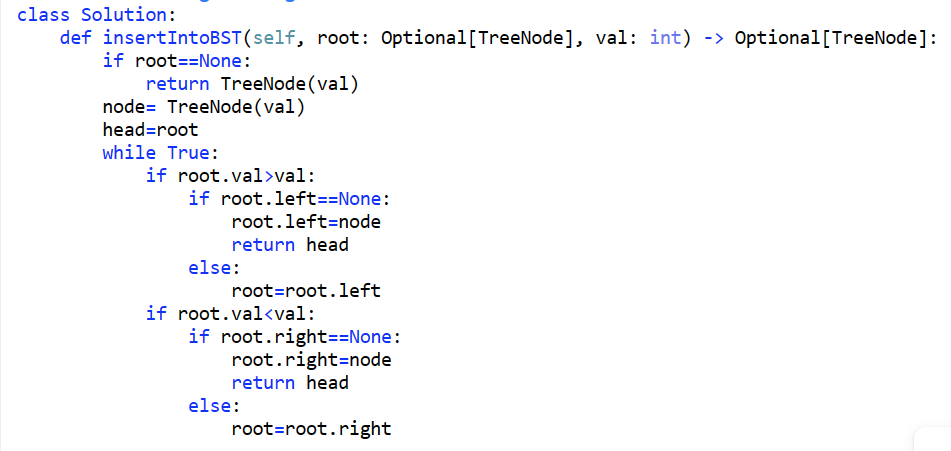
**Q.Search in BST:**

****

**Using recursion:**

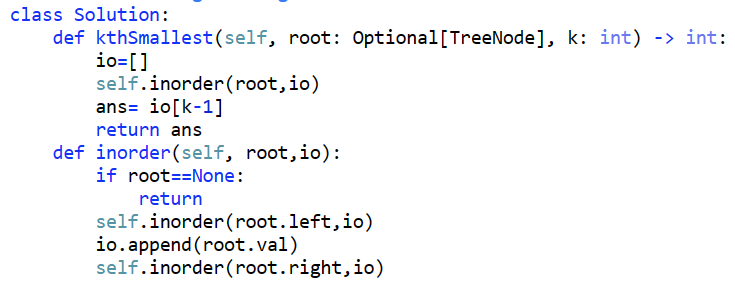
****

**Q: Insert into a BST:**

****

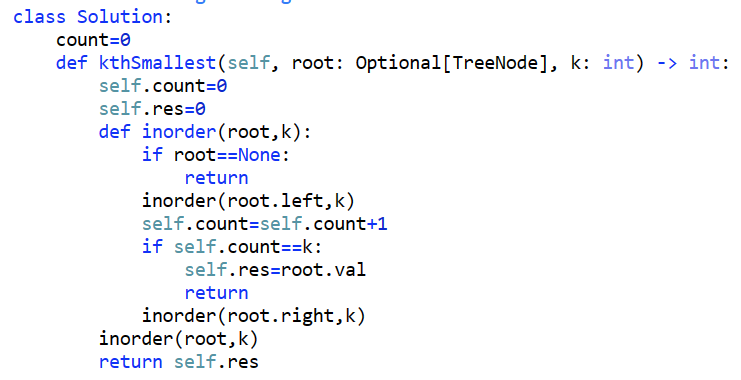
**Q: Kth smallest in BST:**

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

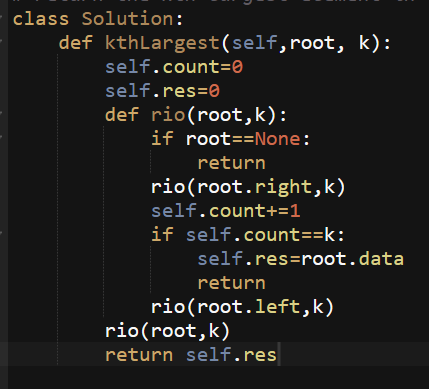
****

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

**NOW lets see solution which take constant space: As**

****

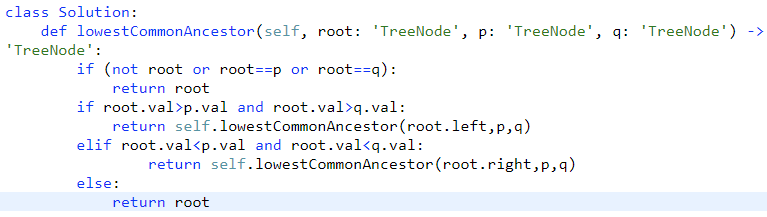
**Q: kth largest in BST: Reverse of Inorder:**

****

**Q: Lowest common ancestor in BST:**

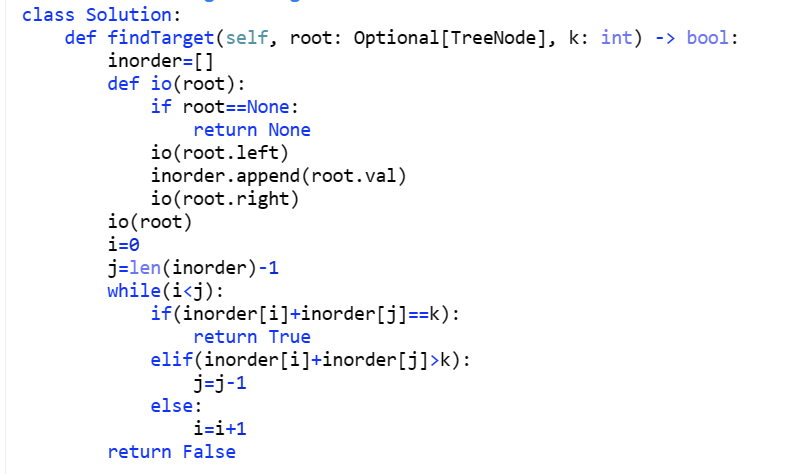
**Intuition:** Utilize the power of BST to solve this problem . search if p and q both are less than root then ancestor must be in left subtree , if p and q both are greater than root, it must be right subtree and if both lie in two different subtree then root itself is the lowest common ancestor.

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

****

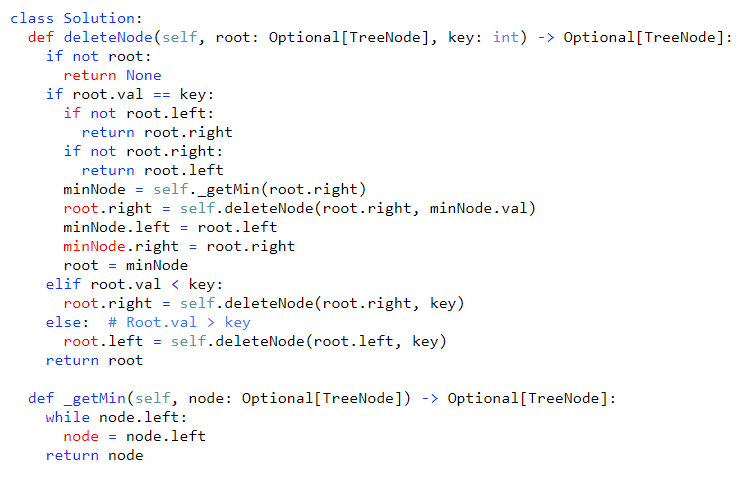
**Q: TWO SUM in BST:**

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



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

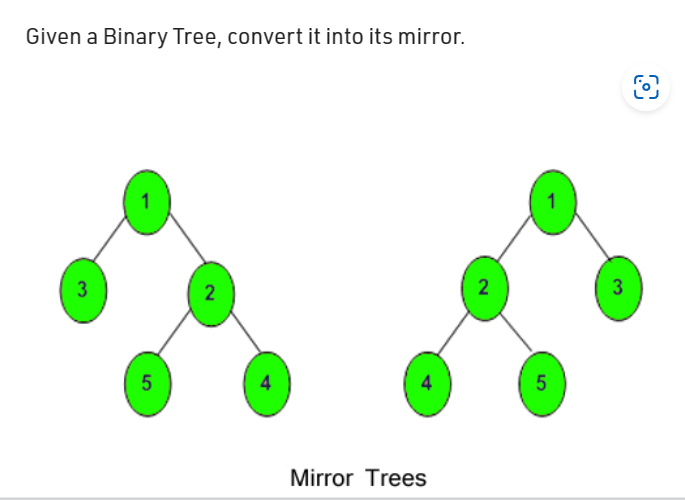
**Delete a node from BST**

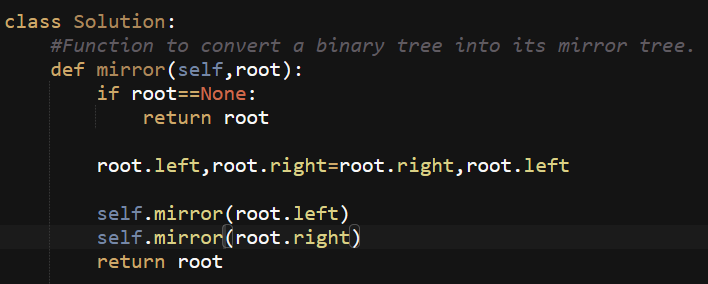
****

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

**To be done**

**Q:**

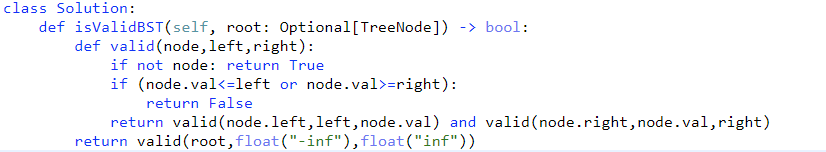
****

****

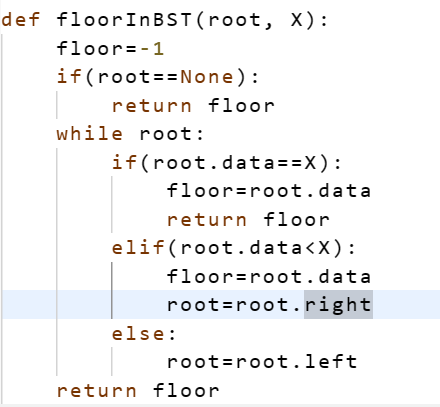
**Validate Binary Search Tree**

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

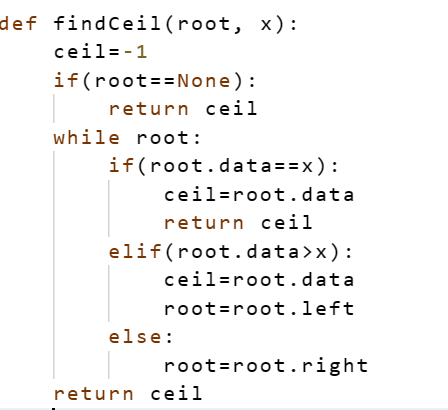
**Note :** here while comparing node value with left and right boundary = is also imp as tree can be 2,2,2 and it is valid BST

****

**Q: Floor of BST:**

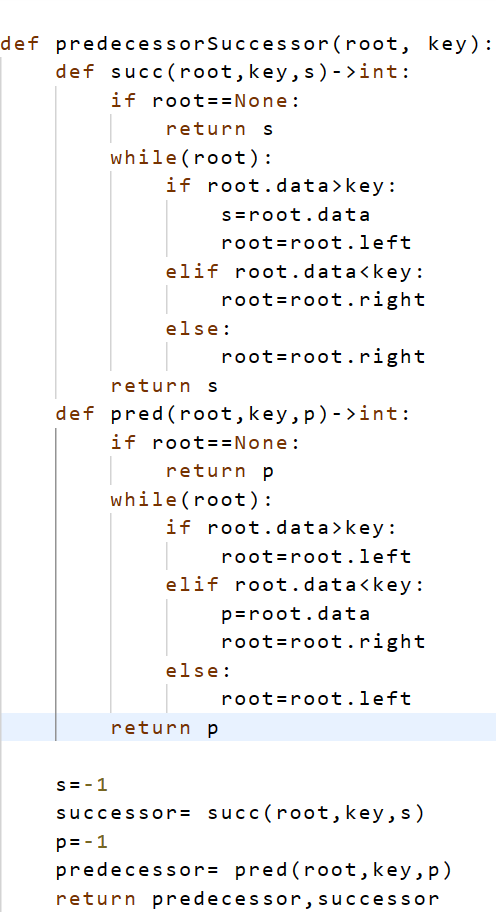
****

**Q: Ceil of BST:**

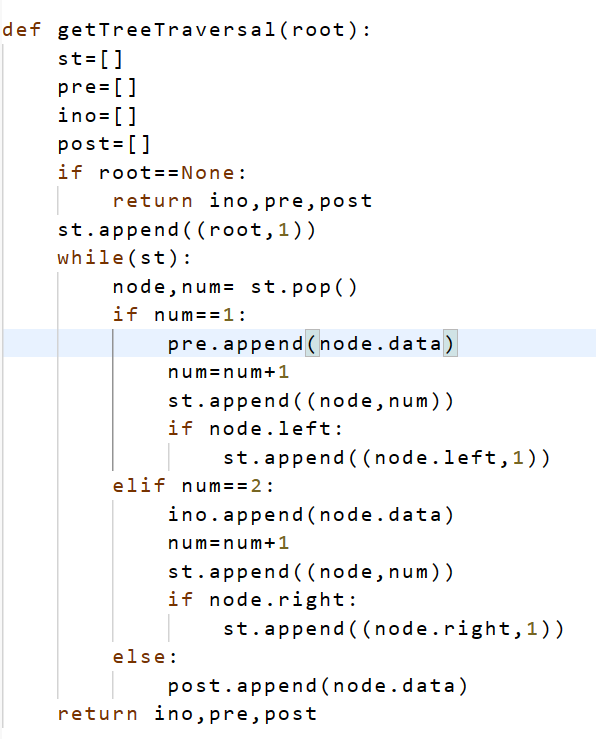
****

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

**SC=O(1)**

****

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

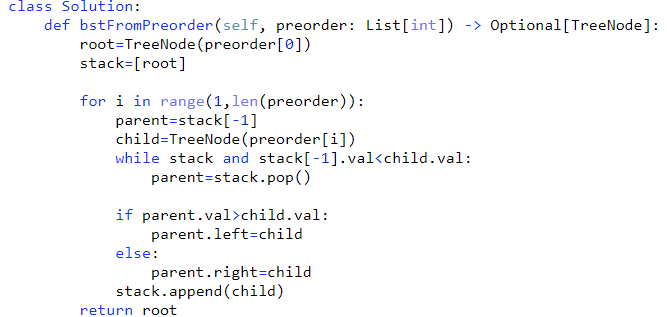
****

**Intuition:** Using stack

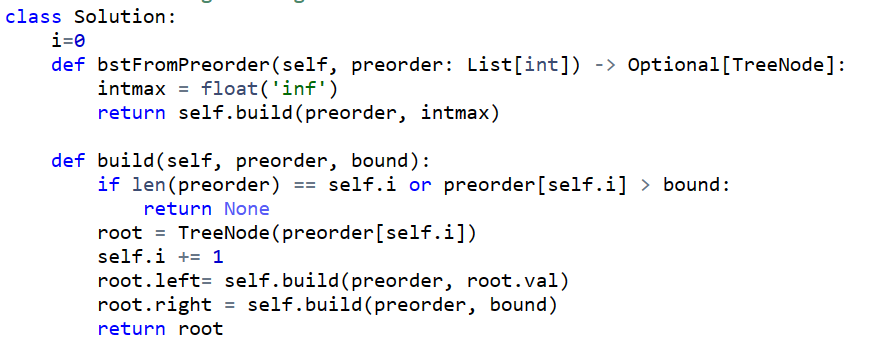
**Q. Construct Binary Search Tree from Preorder Traversal**

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.

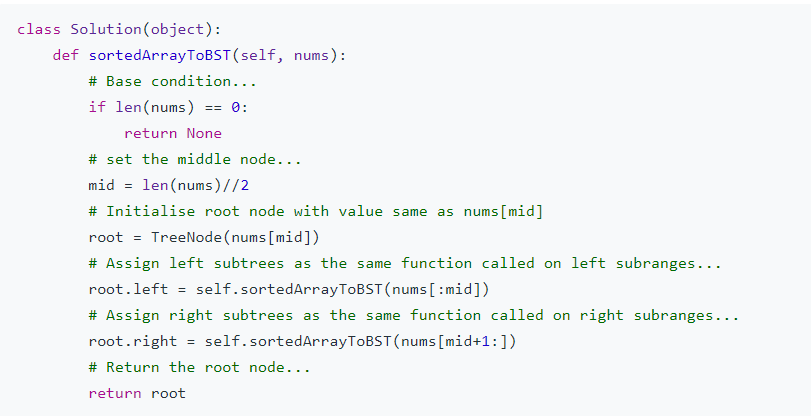
Time Complexity: O(n) and Space complexity: O(h)

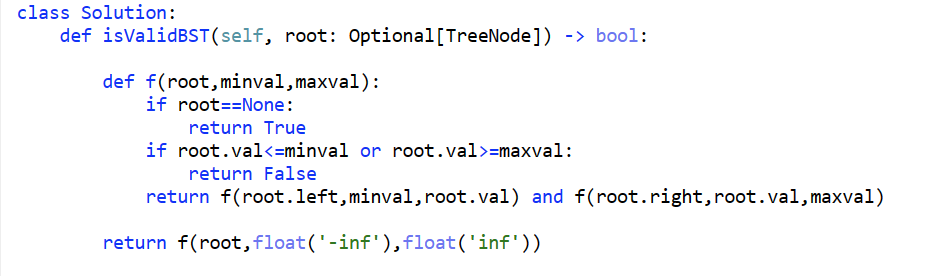


Another approach using recursion:



**Q : Convert Sorted Array to Binary Search Tree**

****

**Q: Check whether given tree is BST or not:**