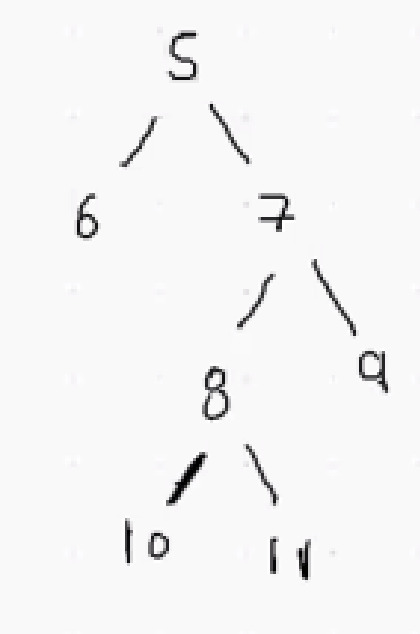
Tree – 3

Given a Binary Tree, count the number of nodes in it.



Frist Approach would be based on global variable and second approach would be Best approach. (how to think post ordered traversal?)

Basic Solution using count (global variable count).

```
def solve(root):
    if root is none:
        global count
        return
    count += 1
    solve (root.left)
    solve (root.right)
```

Size of Binary Tree (geeksforgeeks)

```
count = 0
def solve(node):
    global count
    if node is None:
        return
    count += 1
    solve(node.left)
    solve(node.right)
```

```
def getSize(node):
    global count
    count = 0
    solve(node)
    return count
```

Second Approach:

What is the **size of the sub-tree below 7,** '3' ➔ (7 8 9)?

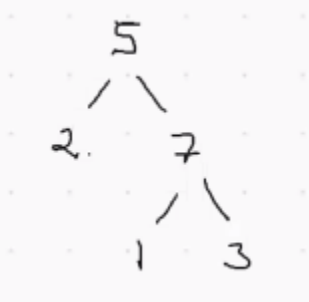
Each node will return its size below it. If 10 and 11 show that their size is 1 (each), then 9 will return 3.

Now, 8 will return 1 and 9 will return 3 to 7. So, the value at 7 would b 1 + 3 + 1 = **5**.

At 5, 6 will return 1 and 7 is returning 5. So, the return at 5 would be **7**.

The size of a leaf node is 1. SO, the size of 6, 8, 10, 11 will return size as 1. We can say that all the leaf nodes, return 1 and for other nodes we can say return

```
def solve (root):
    if root is none:
        return 0
    if root.left is none and root.right is none:
        return 1
    lsize = solve(root.left)
    rsize = solve(root.right)
    return = lsize + rsize + 1
```



We'll make a stack and our root = 5.

So in the stack, lsize = root.left, which is solve of 2. The program goes back to line one and will check if root is None.

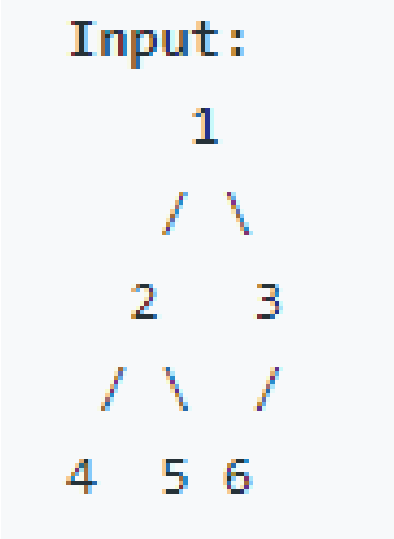
The code will go to line no. 4, rsize = solve(7). For solve (7), since it is neither none or leaf node, the program will go to line 3 and will now call solve(1).

Since 1 is a leaf node, it will return 1. At line 4, we have rsize at root.right. So, solve(3) will return 1 as it is a leaf node. The code will return to 7 and at 7, the rsize will become 1 + 1 + 1 = 3.

222. Count Complete Tree Nodes

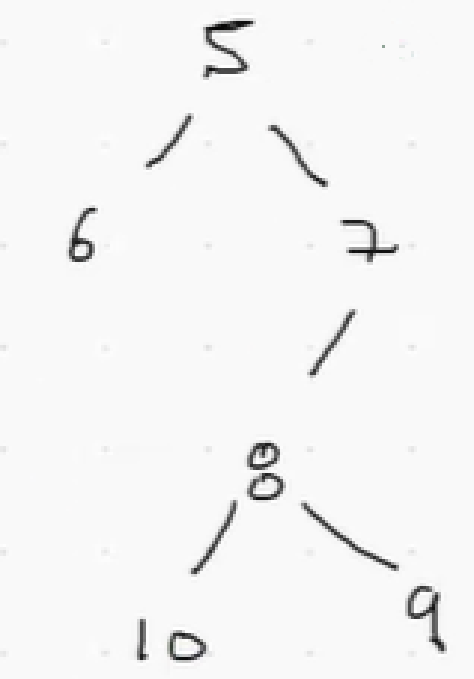
```
class Solution:
    def countNodes(self, root: TreeNode) -> int:
        if root is None:
            return 0
        if root.left is None and root.right is None:
            return 1

        lans = self.countNodes(root.left)
        rans = self.countNodes(root.right)

        return lans + rans + 1
```

We have a better approach for this solution. (H.W: do it using binary search).



The maximum number of nodes, this tree has is 2^3 -1 = 7 and min number of nodes will be 1. So, our answer will be between 1 to 7. The mid for this would be 3. ***Try-n-do.***

Given a binary tree, find the height of it.



The height of the tree is 4.

Each node should give me its left and right height values. Once it knows the height, it should take the max value between its left and right plus its own height ‘1’.

At 8, with two leaf nodes on left (10) and right (9), it will return 2 i.e., the 2 leaf nodes will return the value 1 & 1 and its own height which is also one.

At 7, the height at its right is zero as there is no value on the right and on the left with 8, it will return the value 2. Since 2 > 0, height at 7 would be 2 + 1 (1 being its own height) = 3.

At 6 as it has no left or right values, it will return its own height as 1. Now, between 6 & 7 the greater value is at 7 (=3). So, the height value at 5 would be 3 + 1 = 4.

H.W ➔ write the code.

If we put a mirror in front of the binary tree, we can see its mirror image.

Given a tree transform into its mirror. We can do by swapping the tree (nodes).

```
def solve(root):
    if root is none:
        return
    root.left, root.right = root.right, root.left
    solve(root.left)
    solve(root.right)
```

Recursively do the swapping for left and for right. Another optimization would be if it is a leaf node as well, then return.

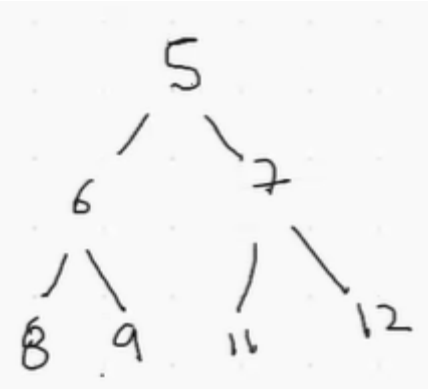
CODE 1

```
class Solution:

    def solve(self, root):
        if root is None:
            return
        root.left, root.right = root.right, root. left

        self.solve(root.left)
        self.solve(root.right)

    def invertTree(self, root: TreeNode) -> TreeNode:
        self.solve(root)
        return root
```

CODE 2:

```
class Solution:

    def invertTree(self, root: TreeNode) -> TreeNode:
        if root is None:
            return
        root.left, root.right = root.right, root. left

        self.invertTree(root.left)
        self.invertTree(root.right)

        return root
```

Given a tree, print it in a level order fashion



The height of this tree is 4.

If a function ‘fn(i)’ will print the level at that height.

```
for height in range(4):
    print (height at ‘ht’ level)
```

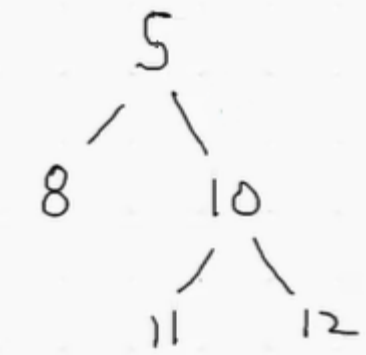
The problem is broken down to a problem which will print nodes at a particular height (h).

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    def findHeight(root):
        if root is None:
            return 0

        if root.left == None and root.right == None:
            return 1

        lh = findHeight(root.left)
        rh = findHeight(root.right)
        return max(lh, rh) + 1

    def printNodeAtHeight(root, ht):
        if root is None:
            return

        if ht == 1:
```

```
            print(root.data)
            return

        printNodeAtHeight(root.left, ht-1)
        printNodeAtHeight(root.right, ht-1)

    def printLevelWise(root):
        if root is None:
            return

        ht = findHeight(root)

        for h in range(1, ht + 1):
            printNodeAtHeight(root, h)



if __name__ == "__main__":
    root = Node(5)
    root.left = Node(10)
    root.right = Node(15)

    root.left.right = Node(100)
    root.left.left = Node(50)

    printLevelWise(root)
```

The maximum height of a binary tree would be n. So, the time complexity would be in O (n^2)

Level Order Traversal



Queue is a best data structure while solving level order traversal of tree. In Queue, the elements are inserted from the back and elements are removed from the front. So, we will put 5 in the queue.

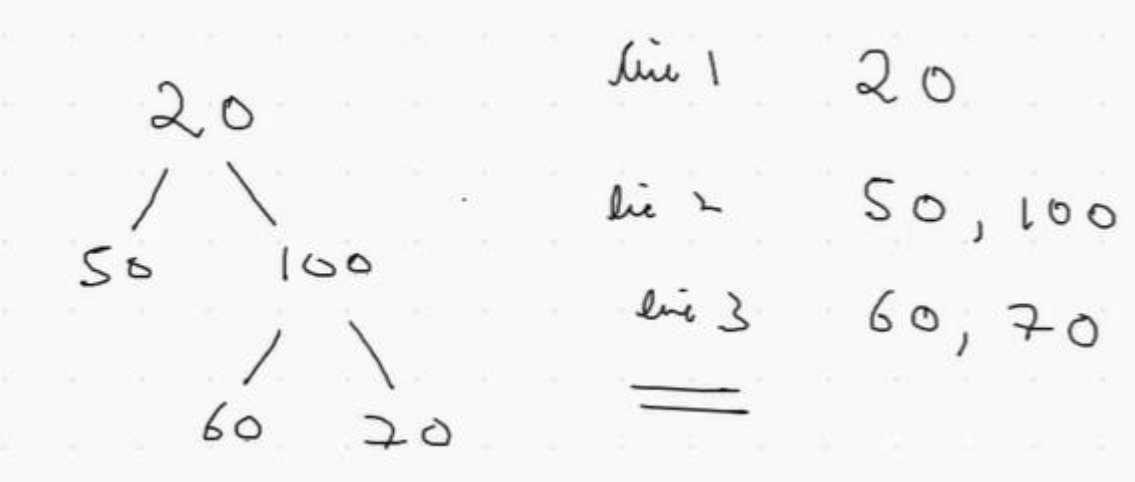
Now, while queue is not empty, we can pop one item from queue and print it and then enqueue children of 5 (8, 10). Is the queue empty? NO. So we will dequeue from the queue and print 8 and check for children of 8. Since 8 is a leaf node, we wont enqueue any.

We will dequeue 10 and check for its children. 10 has two children (11, 12) which will be enqueued into the queue.

11 will be dequeued and check for its children. Since there are not children nothing will be enqueued. 12 also will be dequeued and since it is a leaf node, nothing will be enqueued.

```
def printLevelOrderQueue(root):
    if root is None:
        return

    queue = list()
    queue.append(root)

    while len(queue) > 0:
        x = queue.pop(0)
        print(x.data)

        if x.left is None:
            queue.append(x.left)

        if x.right is None:
            queue.append(x.right)
```

If you want an output as:



We will use -1 in the queue and every time we enqueue an element into the queue.

First we will enqueue 20 to the queue and enqueue -1 as well.

20 – 1

Then, we will dequeue 20 and check if it has children and enqueue its children into the queue.

~~20~~ -1 50 100

Once -1 is popped as well, the print function will go to the new line and at the same time will enqueue -1 at the end of the queue.

50 100 -1

Since 50 is a leaf node, it has no children and so we will not enqueue any elements into the queue.

100 -1

100 is not a leaf node and hence once you dequeue 100, you will enqueue its children 60 & 70.

-1 60 70

The next element is -1 and once you dequeue it, the print function will go to next line.

As 60 and 70 are leaf codes nothing will be enqueue when 60 and 70 are dequeued.