**DSA – ASSIGNMENT 23**

💡 1. Given preorder of a binary tree, calculate its [**depth(or height)**](https://www.geeksforgeeks.org/write-a-c-program-to-find-the-maximum-depth-or-height-of-a-tree/) [starting from depth 0]. The preorder is given as a string with two possible characters.

1. ‘l’ denotes the leaf
2. ‘n’ denotes internal node

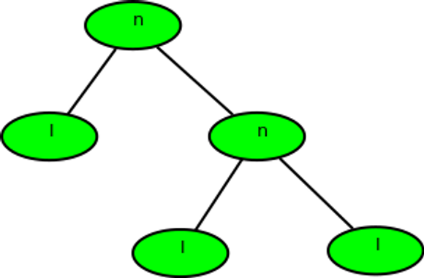
The given tree can be seen as a full binary tree where every node has 0 or two children. The two children of a node can ‘n’ or ‘l’ or mix of both.

**Examples :**

Input : nlnll

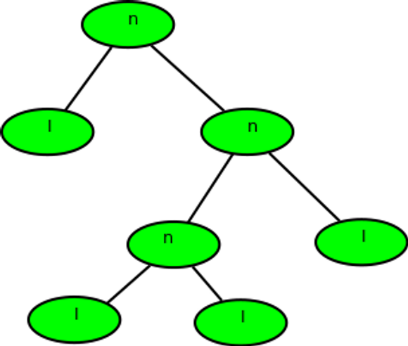
Output : 2

Explanation :



Input : nlnnlll

Output : 3



**Solution. :-**

* Define a function calculateDepth that takes the preorder traversal string as an argument.
* If the given preorder traversal string is empty or None, return -1 to indicate an empty tree (depth of -1).
* Initialize a variable depth to 0.
* While the preorder traversal string is not empty, do the following:
  + - Extract the first character from the preorder traversal string.
  + If the extracted character is 'n' (internal node), increment the depth variable by 1.
  + If the extracted character is 'l' (leaf node), return the depth value (as the tree is assumed to be a full binary tree, a leaf node indicates the end of a subtree).
  + Recursively call the calculateDepth function with the remaining part of the preorder traversal string.
  + Update the preorder traversal string to exclude the extracted character and its corresponding subtree.
* If the loop finishes without returning a value, return the depth variable as the overall depth of the binary tree.

**def calculateDepth(preorder):**

**if not preorder:**

**return -1**

**depth = 0**

**while preorder:**

**char = preorder[0]**

**preorder = preorder[1:]**

**if char == 'n':**

**depth += 1**

**elif char == 'l':**

**return depth**

**return depth**

**# Example usage**

**preorder = "nlnll"**

**depth = calculateDepth(preorder)**

**print(depth)**

💡 2. Given a Binary tree, the task is to print the **left view** of the Binary Tree. The left view of a Binary Tree is a set of leftmost nodes for every level.

**Examples:**

***Input:***

4

/   \\

5     2

/   \\

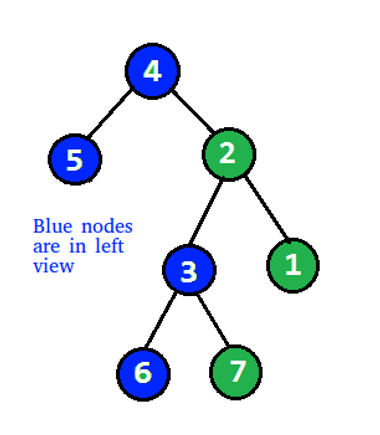
3     1

/  \\

6    7

***Output:****4 5 3 6*

**Explanation:**



***Input:***

1

/   \\

2       3

\\

4

\\

5

\\

6

**Output:** 1 2 4 5 6

**Solution. :-**

* Define a class Node to represent each node in the binary tree. Each node will have a value, left pointer, and right pointer.
* Define a function printLeftView that takes the root node of the binary tree as an argument.
* Define a helper function printLeftViewUtil that takes the current node, the current level, and a list maxLevel to track the maximum level reached so far.
* Inside the printLeftViewUtil function, do the following:
  + If the current node is None, return.
  + If the current level is greater than maxLevel[0], print the value of the current node and update maxLevel[0] to the current level.
  + Recursively call printLeftViewUtil for the left child of the current node with the current level incremented by 1.
  + Recursively call printLeftViewUtil for the right child of the current node with the current level incremented by 1.
* Call the printLeftViewUtil function with the root node, initial level 1, and the list maxLevel containing a single element initialized to 0.
* The output will be the left view of the binary tree.

**class Node:**

**def \_\_init\_\_(self, value):**

**self.data = value**

**self.left = None**

**self.right = None**

**def printLeftView(root):**

**if root is None:**

**return**

**maxLevel = [0] # list to track the maximum level reached so far**

**def printLeftViewUtil(node, level, maxLevel):**

**if node is None:**

**return**

**if level > maxLevel[0]:**

**print(node.data, end=" ")**

**maxLevel[0] = level**

**printLeftViewUtil(node.left, level + 1, maxLevel)**

**printLeftViewUtil(node.right, level + 1, maxLevel)**

**printLeftViewUtil(root, 1, maxLevel)**

**# Example usage**

**root = Node(4)**

**root.left = Node(5)**

**root.right = Node(2)**

**root.right.left = Node(3)**

**root.right.right = Node(1)**

**root.right.left.left = Node(6)**

**root.right.left.right = Node(7)**

**print("Left View:")**

**printLeftView(root)**

💡 3. Given a Binary Tree, print the Right view of it.

The right view of a Binary Tree is a set of nodes visible when the tree is visited from the Right side.

**Examples:**

**Input:**

1

/     \\

2        3

/   \       /  \

4     5   6    7

\\

8

**Output**:

Right view of the tree is 1 3 7 8

**Input:**

1

/

8

/

7

**Output**:

Right view of the tree is 1 8 7

**Solution. :-**

* Define a class Node to represent each node in the binary tree. Each node will have a value, left pointer, and right pointer.
* Define a function printRightView that takes the root node of the binary tree as an argument.
* Define a helper function printRightViewUtil that takes the current node, the current level, and a list maxLevel to track the maximum level reached so far.
* Inside the printRightViewUtil function, do the following:
  + If the current node is None, return.
  + If the current level is greater than maxLevel[0], print the value of the current node and update maxLevel[0] to the current level.
  + Recursively call printRightViewUtil for the right child of the current node with the current level incremented by 1.
  + Recursively call printRightViewUtil for the left child of the current node with the current level incremented by 1.
* Call the printRightViewUtil function with the root node, initial level 1, and the list maxLevel containing a single element initialized to 0.
* The output will be the right view of the binary tree.

**class Node:**

**def \_\_init\_\_(self, value):**

**self.data = value**

**self.left = None**

**self.right = None**

**def printRightView(root):**

**if root is None:**

**return**

**maxLevel = [0] # list to track the maximum level reached so far**

**def printRightViewUtil(node, level, maxLevel):**

**if node is None:**

**return**

**if level > maxLevel[0]:**

**print(node.data, end=" ")**

**maxLevel[0] = level**

**printRightViewUtil(node.right, level + 1, maxLevel)**

**printRightViewUtil(node.left, level + 1, maxLevel)**

**printRightViewUtil(root, 1, maxLevel)**

**# Example usage**

**root = Node(1)**

**root.left = Node(2)**

**root.right = Node(3)**

**root.left.left = Node(4)**

**root.left.right = Node(5)**

**root.right.left = Node(6)**

**root.right.right = Node(7)**

**root.right.right.right = Node(8)**

**print("Right View:")**

**printRightView(root)**

💡4. Given a Binary Tree, The task is to print the **bottom view** from left to right. A node **x** is there in output if x is the bottommost node at its horizontal distance. The horizontal distance of the left child of a node x is equal to a horizontal distance of x minus 1, and that of a right child is the horizontal distance of x plus 1.

**Examples:**

**Input:**

20

/     \\

8         22

/      \\         \\

5         3        25

/    \\

10       14

**Output:** 5, 10, 3, 14, 25.

**Input:**

20

/     \\

8         22

/      \\      /   \\

5         3   4     25

/    \\

10       14

**Output:**

5 10 4 14 25.

**Explanation:**

If there are multiple bottom-most nodes for a horizontal distance from the root, then print the later one in the level traversal.

**3 and 4** are both the bottom-most nodes at a horizontal distance of 0, we need to print 4.

**Solution. :-**

* Define a class Node to represent each node in the binary tree. Each node will have a value, left pointer, and right pointer.
* Define a function printBottomView that takes the root node of the binary tree as an argument.
* Initialize an empty dictionary horizontalDistance to store the horizontal distance of each node.
* Initialize a queue and enqueue the root node along with its horizontal distance, which is initially 0.
* While the queue is not empty, do the following:
  + Dequeue a node and its horizontal distance from the queue.
  + Update the horizontalDistance dictionary with the horizontal distance of the node. Since we want to print the bottom view, we will always update the horizontal distance with the latest value encountered for each horizontal distance.
  + Enqueue the left child of the dequeued node with its horizontal distance decreased by 1.
  + Enqueue the right child of the dequeued node with its horizontal distance increased by 1.
* Sort the horizontalDistance dictionary by keys to get the nodes in horizontal distance order.
* Print the values of the sorted horizontalDistance dictionary, which represent the bottom view of the binary tree.

**from collections import deque**

**class Node:**

**def \_\_init\_\_(self, value):**

**self.data = value**

**self.left = None**

**self.right = None**

**def printBottomView(root):**

**if root is None:**

**return**

**horizontalDistance = {} # dictionary to store horizontal distance of nodes**

**queue = deque()**

**queue.append((root, 0)) # enqueue root node with horizontal distance 0**

**while queue:**

**node, hd = queue.popleft()**

**# Update the horizontal distance value**

**horizontalDistance[hd] = node.data**

**# Enqueue the left child with horizontal distance decreased by 1**

**if node.left:**

**queue.append((node.left, hd - 1))**

**# Enqueue the right child with horizontal distance increased by 1**

**if node.right:**

**queue.append((node.right, hd + 1))**

**# Sort the horizontal distance dictionary by keys**

**sorted\_hd = sorted(horizontalDistance.items(), key=lambda x: x[0])**

**# Print the bottom view values**

**print("Bottom View:")**

**for hd, value in sorted\_hd:**

**print(value, end=" ")**

**print()**

**# Example usage**

**root = Node(20)**

**root.left = Node(8)**

**root.right = Node(22)**

**root.left.left = Node(5)**

**root.left.right = Node(3)**

**root.right.right = Node(25)**

**root.left.right.left = Node(10)**

**root.left.right.right = Node(14)**

**printBottomView(root)**