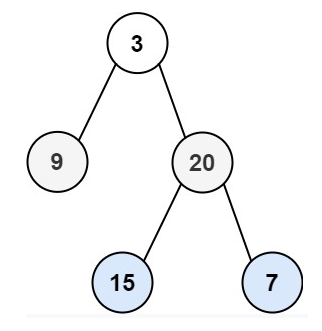
<https://leetcode.com/problems/binary-tree-level-order-traversal/>

Given the root of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

**Example 1:**



Input: root = [3,9,20,null,null,15,7]

Output: [[3],[9,20],[15,7]]

**Example 2:**

Input: root = [1]

Output: [[1]]

**Example 3:**

Input: root = []

Output: []

**Constraints:**

* The number of nodes in the tree is in the range [0, 2000].
* -1000 <= Node.val <= 1000

**Attempt 1: 2022-11-02**

**Solution 1: Queue with queue size calculation(10 min)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> result = new ArrayList<List<Integer>>();

if(root == null) {

return result;

}

Queue<TreeNode> q = new LinkedList<TreeNode>();

q.offer(root);

while(!q.isEmpty()) {

List<Integer> list = new ArrayList<Integer>();

int size = q.size();

for(int i = 0; i < size; i++) {

TreeNode node = q.poll();

list.add(node.val);

if(node.left != null) {

q.offer(node.left);

}

if(node.right != null) {

q.offer(node.right);

}

}

result.add(list);

}

return result;

}

}

Time Complexity: O(N), where N is number of nodes in the Binary Tree

Space Complexity: O(N)

**Solution 2: Recursive traversal (60 min)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> result = new ArrayList<List<Integer>>();

if(root == null) {

return result;

}

helper(root, result, 0);

return result;

}

private void helper(TreeNode root, List<List<Integer>> result, int depth) {

if(root == null) {

return;

}

if(result.size() == depth) {

result.add(new ArrayList<Integer>());

}

result.get(depth).add(root.val);

helper(root.left, result, depth + 1);

helper(root.right, result, depth + 1);

}

}

Time Complexity: O(N), where N is number of nodes in the Binary Tree

Space Complexity: O(N)

**How level order traversal implement by preorder traversal (DFS) works ?**

The below statement is critical and only triggered by hit each node on very left path goes from root node to leaf node since DFS based on preorder traversal

if(result.size() == depth) {

result.add(new ArrayList<Integer>());

}

For example, on below tree the creation of new ArrayList only happens on when very left path goes from root node 5 to left node 2 (5 -> 3 -> 2), and result size will fix at 3, 1 more than depth as 2,  so in later recursion no matter how preorder traversal goes back and forth, the "result.size() == depth" logic will never hit again.

e.g

5 5 (depth=0), in first recursion of helper method, result={}, size=0

/ \ / then create first array and add 5 into it, result={{5}}

3 6 very left path 3 (depth=1), in second recursion of helper method, result={{5}}, size=1

/ \ \ ====> / then create second array and add 3 into it, result={{5},{3}}

2 4 7 2 (depth=2), in third recursion of helper method, result={{5},{3}}, size=2

then create third array and add 2 into it, result={{5},{3},{2}}

-------------------------------------------------------------------------------------------------------------

Now all creation of new ArrayList() has been done for this binary tree, result size fixed at 3, and depth between 0 to 2, will never hit 3.

-------------------------------------------------------------------------------------------------------------

Take a look at what will happen to node 4, after add 2 into third array, the preorder traversal will goes back to 3 and forth to 4, now, result.size()=3 but depth=2, the new array won't create again and directly flow into logic as

result.get(depth).add(root.val) => reult.get(2).add(4), so result={{5},{3},{2,4}}

**Refer to**

<https://leetcode.com/problems/binary-tree-level-order-traversal/discuss/33445/Java-Solution-using-DFS>

The flaw of this solution is the naming of variable "height" suppose to be "depth", because depth from top to down is increasing and start from 0, height is from bottom to up, which not the case for "result.size() == depth" logic here.

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> res = new ArrayList<List<Integer>>();

levelHelper(res, root, 0);

return res;

}

public void levelHelper(List<List<Integer>> res, TreeNode root, int height) {

if (root == null) return;

if (height >= res.size()) {

res.add(new LinkedList<Integer>());

}

res.get(height).add(root.val);

levelHelper(res, root.left, height+1);

levelHelper(res, root.right, height+1);

}