# Question

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

# Solution

#### **How to traverse the tree**

There are two general strategies to traverse a tree:

* Depth First Search (DFS)

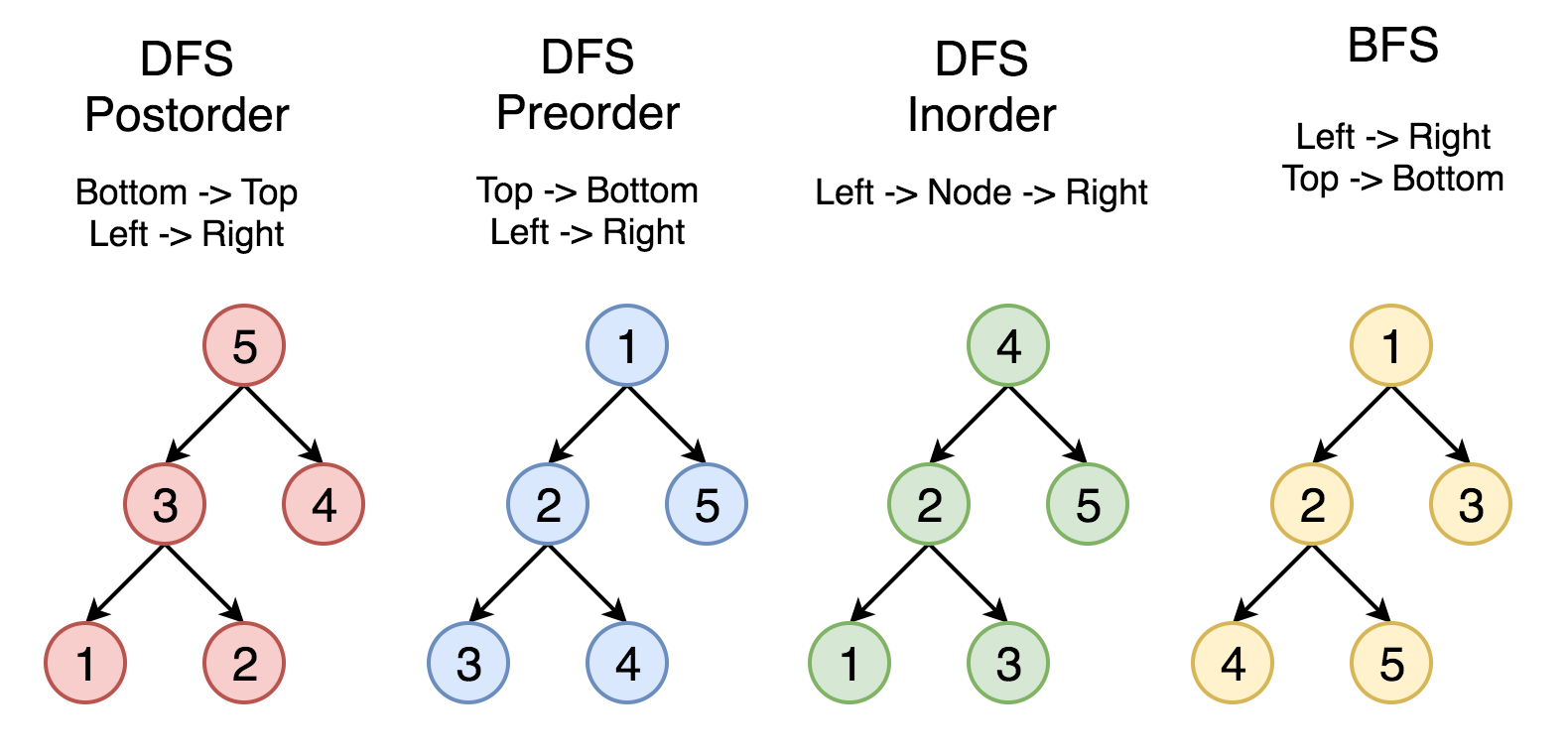
In this strategy, we adopt the depth as the priority, so that one would start from a root and reach all the way down to certain leaf, and then back to root to reach another branch.

The DFS strategy can further be distinguished as preorder, inorder, and postorder depending on the relative order among the root node, left node and right node.

* Breadth First Search (BFS)

We scan through the tree level by level, following the order of height, from top to bottom. The nodes on higher level would be visited before the ones with lower levels.

On the following figure the nodes are numerated in the order you visit them, please follow 1-2-3-4-5 to compare different strategies.



Here the problem is to implement split-level BFS traversal : [[1], [2, 3], [4, 5]].

#### **Approach 1: Recursion**

**Algorithm**

The simplest way to solve the problem is to use a recursion. Let's first ensure that the tree is not empty, and then call recursively the function helper(node, level), which takes the current node and its level as the arguments.

This function does the following :

* The output list here is called levels, and hence the current level is just a length of this list len(levels). Compare the number of a current level len(levels) with a node level level. If you're still on the previous level - add the new one by adding a new list into levels.
* Append the node value to the last list in levels.
* Process recursively child nodes if they are not None : helper(node.left / node.right, level + 1).

**Implementation**

1 / 15

|  |
| --- |
| **class Solution {**  **List<List<Integer>> levels = new ArrayList<List<Integer>>();**  **public void helper(TreeNode node, int level) {**  **// start the current level**  **if (levels.size() == level)**  **levels.add(new ArrayList<Integer>());**  **// fulfil the current level**  **levels.get(level).add(node.val);**  **// process child nodes for the next level**  **if (node.left != null)**  **helper(node.left, level + 1);**  **if (node.right != null)**  **helper(node.right, level + 1);**  **}**    **public List<List<Integer>> levelOrder(TreeNode root) {**  **if (root == null) return levels;**  **helper(root, 0);**  **return levels;**  **}**  **}** |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) since each node is processed exactly once.
* Space complexity : \mathcal{O}(N)O(*N*) to keep the output structure which contains N node values.

#### **Approach 2: Iteration**

**Algorithm**

The recursion above could be rewritten in the iteration form.

Let's keep nodes of each tree level in the queue structure, which typically orders elements in a FIFO (first-in-first-out) manner. In Java one could use [LinkedList implementation of the Queue interface](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html). In Python using [Queue structure](https://docs.python.org/3/library/queue.html) would be an overkill since it's designed for a safe exchange between multiple threads and hence requires locking which leads to a performance loose. In Python the queue implementation with a fast atomic append() and popleft() is [deque](https://docs.python.org/3/library/collections.html#collections.deque).

The zero level contains only one node root. The algorithm is simple :

* Initiate queue with a root and start from the level number 0 : level = 0.
* While queue is not empty :
  + Start the current level by adding an empty list into output structure levels.
  + Compute how many elements should be on the current level : it's a queue length.
  + Pop out all these elements from the queue and add them into the current level.
  + Push their child nodes into the queue for the next level.
  + Go to the next level level++.

**Implementation**

|  |
| --- |
| class Solution {  public List<List<Integer>> levelOrder(TreeNode root) {  List<List<Integer>> levels = new ArrayList<List<Integer>>();  if (root == null) return levels;  Queue<TreeNode> queue = new LinkedList<TreeNode>();  queue.add(root);  int level = 0;  while ( !queue.isEmpty() ) {  // start the current level  levels.add(new ArrayList<Integer>());  // number of elements in the current level  int level\_length = queue.size();  for(int i = 0; i < level\_length; ++i) {  TreeNode node = queue.remove();  // fulfill the current level  levels.get(level).add(node.val);  // add child nodes of the current level  // in the queue for the next level  if (node.left != null) queue.add(node.left);  if (node.right != null) queue.add(node.right);  }  // go to next level  level++;  }  return levels;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) since each node is processed exactly once.
* Space complexity : \mathcal{O}(N)O(*N*) to keep the output structure which contains N node values.