

## Binary Tree Level Order Traversal (easy)

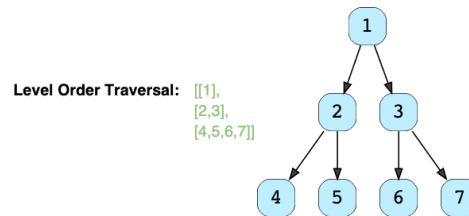
### We'll cover the following ^

- Problem Statement
- Try it yourself
- Solution
- Code
  - Time complexity
  - Space complexity

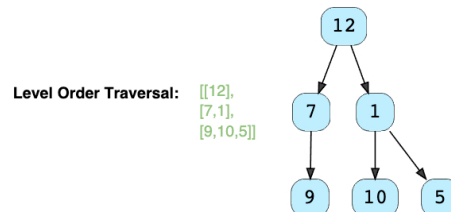
### Problem Statement #

Given a binary tree, populate an array to represent its level-by-level traversal. You should populate the values of all **nodes of each level from left to right** in separate sub-arrays.

Example 1:



Example 2:



### Try it yourself #

Try solving this question here:

Java Python3 JS C++

```
1 import java.util.*;
2
3 class TreeNode {
4     int val;
5     TreeNode left;
6     TreeNode right;
7
8     TreeNode(int x) {
9         val = x;
10    }
11 }
12
13 class LevelOrderTraversal {
14     public static List<List<Integer>> traverse(TreeNode root) {
15         List<List<Integer>> result = new ArrayList<List<Integer>>();
16         // TODO: Write your code here
17         return result;
18     }
19 }
```

```
20 public static void main(String[] args) {
21     TreeNode root = new TreeNode(12);
22     root.left = new TreeNode(7);
23     root.right = new TreeNode(1);
24     root.left.left = new TreeNode(9);
25     root.right.left = new TreeNode(10);
26     root.right.right = new TreeNode(5);
27     List<List<Integer>> result = LevelOrderTraversal.traverse(root);
28     System.out.println("Level order traversal: " + result);

```

Run

Save

Reset

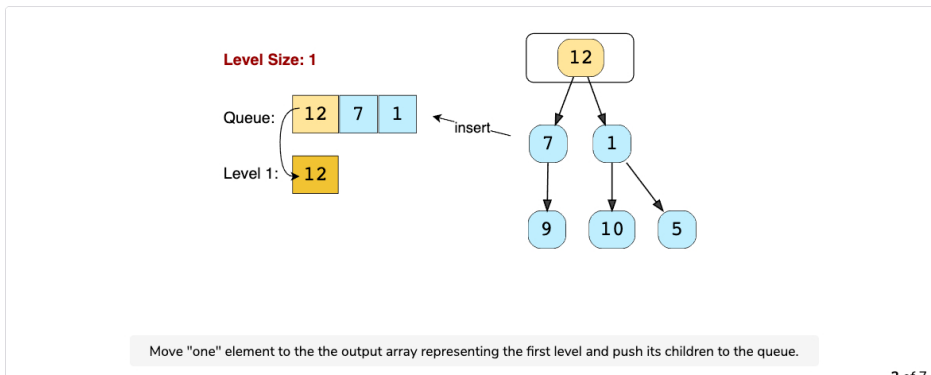
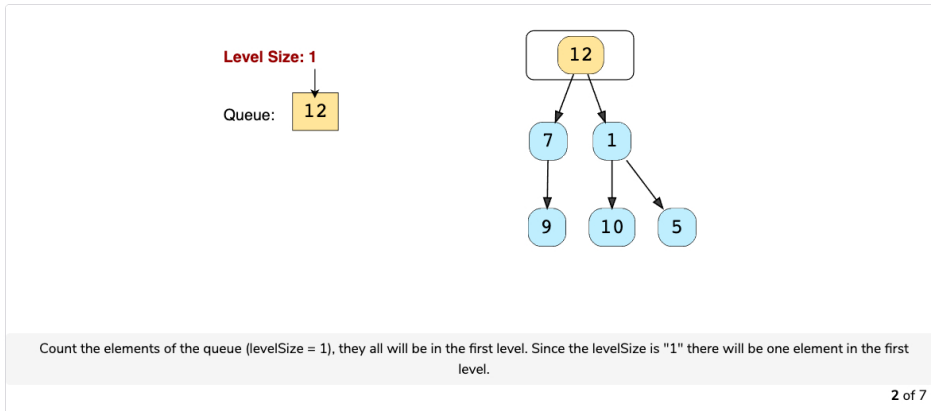
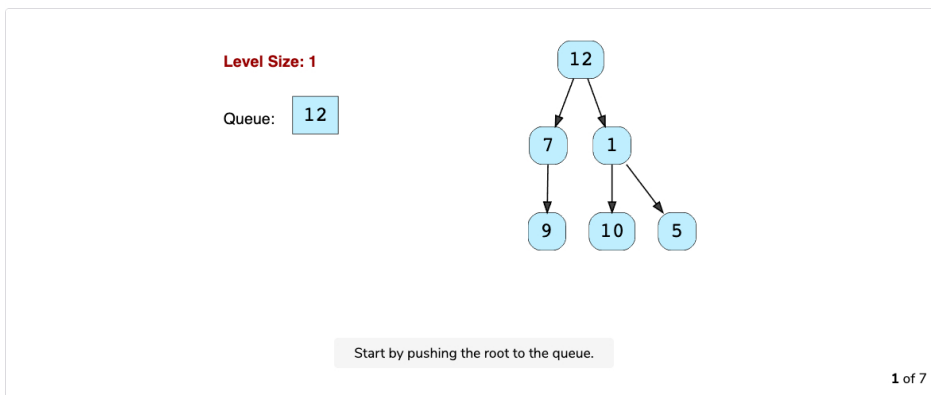
## Solution #

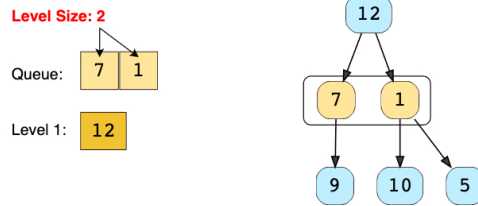
Since we need to traverse all nodes of each level before moving onto the next level, we can use the **Breadth First Search (BFS)** technique to solve this problem.

We can use a Queue to efficiently traverse in BFS fashion. Here are the steps of our algorithm:

1. Start by pushing the `root` node to the queue.
2. Keep iterating until the queue is empty.
3. In each iteration, first count the elements in the queue (let's call it `levelSize`). We will have these many nodes in the current level.
4. Next, remove `levelSize` nodes from the queue and push their `value` in an array to represent the current level.
5. After removing each node from the queue, insert both of its children into the queue.
6. If the queue is not empty, repeat from step 3 for the next level.

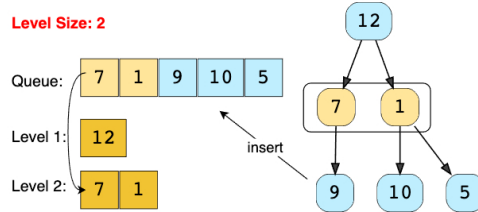
Let's take the example-2 mentioned above to visually represent our algorithm:





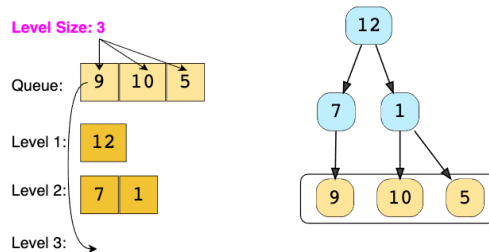
Count the elements of the queue (levelSize = 2), they all will be in the second level. Since the levelSize is "2" there will be two elements in the second level.

4 of 7



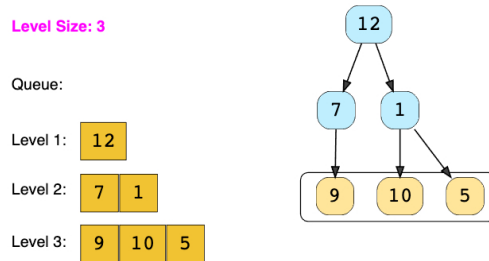
Move "two" elements to the output array representing the second level and push their children to the queue in the same order.

5 of 7



Count the elements of the queue (levelSize = 3), they all will be in the third level. Since the levelSize is "3" there will be three elements in the third level.

6 of 7



Move "three" elements to the output array representing third level.

7 of 7



## Code #

Here is what our algorithm will look like:

Java Python3 C++ JS

```

1 import java.util.*;
2
3 class TreeNode {
4     int val;
  
```



```
5   TreeNode left;
6   TreeNode right;
7
8   TreeNode(int x) {
9       val = x;
10  }
11 };
12
13 class LevelOrderTraversal {
14     public static List<List<Integer>> traverse(TreeNode root) {
15         List<List<Integer>> result = new ArrayList<List<Integer>>();
16         if (root == null)
17             return result;
18
19         Queue<TreeNode> queue = new LinkedList<>();
20         queue.offer(root);
21         while (!queue.isEmpty()) {
22             int levelSize = queue.size();
23             List<Integer> currentLevel = new ArrayList<>(levelSize);
24             for (int i = 0; i < levelSize; i++) {
25                 TreeNode currentNode = queue.poll();
26                 // add the node to the current level
27                 currentLevel.add(currentNode.val);
28                 // insert the children of current node in the queue
            }
        }
    }
}
```

[Run](#) [Save](#) [Reset](#) [Copy](#)

### Time complexity #

The time complexity of the above algorithm is  $O(N)$ , where 'N' is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

### Space complexity #

The space complexity of the above algorithm will be  $O(N)$  as we need to return a list containing the level order traversal. We will also need  $O(N)$  space for the queue. Since we can have a maximum of  $N/2$  nodes at any level (this could happen only at the lowest level), therefore we will need  $O(N)$  space to store them in the queue.

[← Back](#)[Introduction](#)[Next →](#)[Reverse Level Order Traversal \(easy\)](#)[✔ Mark as Completed](#)[! Report an Issue](#) [? Ask a Question](#)