

Reverse Level Order Traversal (easy)

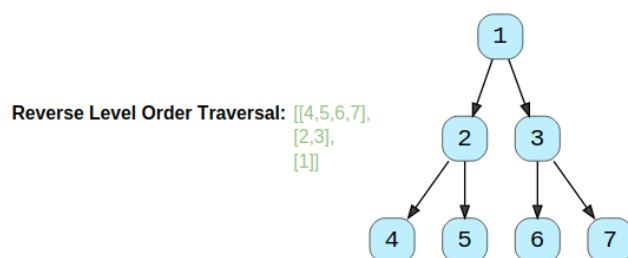
We'll cover the following ^

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

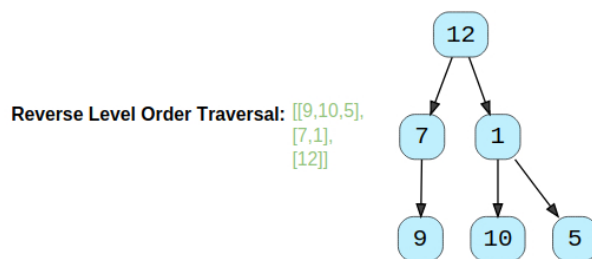
Problem Statement

Given a binary tree, populate an array to represent its level-by-level traversal in reverse order, i.e., the **lowest level comes first**. You should populate the values of all nodes in 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++

```

4   def __init__(self, val):
5       self.val = val
6       self.left, self.right = None, None
7
8   def traverse(root):
9       result = deque()
10      # TODO: Write your code here
11      return result
12
13  def main():
14      root = TreeNode(12)
15      root.left = TreeNode(7)
16      root.right = TreeNode(1)
17      root.left.left = TreeNode(9)
18      root.right.left = TreeNode(10)
19      root.right.right = TreeNode(5)
20      print("Reverse level order traversal: " + str(traverse(root)))
21
22
23  main()
24

```

Run

Save

Reset



Solution

This problem follows the [Binary Tree Level Order Traversal](#) pattern. We can follow the same **BFS** approach. The only difference will be that instead of appending the current level at the end, we will append the current level at the beginning of the result list.

Code

Here is what our algorithm will look like; only the highlighted lines have changed. Please note that, for **Java**, we will use a [LinkedList](#) instead of an [ArrayList](#) for our result list. As in the case of [ArrayList](#), appending an element at the beginning means shifting all the existing elements. Since we need to append the level array at the beginning of the result list, a [LinkedList](#) will be better, as this shifting of elements is not required in a [LinkedList](#). Similarly, we will use a double-ended queue (deque) for **Python**, **C++**, and **JavaScript**.

[Java](#)
[Python3](#)
[C++](#)
[JS](#)

```

1  from collections import deque
2
3
4  class TreeNode:
5      def __init__(self, val):
6          self.val = val
7          self.left, self.right = None, None
8
9
10 def traverse(root):
11     result = deque()
12     if root is None:
13         return result
14
15     queue = deque()
16     queue.append(root)
17     while queue:
18         levelSize = len(queue)
19         currentLevel = []
20         for _ in range(levelSize):
21             currentNode = queue.popleft()
22             # add the node to the current level
23             currentLevel.append(currentNode.val)
24             # insert the children of current node in the queue
25             if currentNode.left:
26                 queue.append(currentNode.left)
27             if currentNode.right:
28                 queue.append(currentNode.right)
29
30     result.appendleft(currentLevel)
31
32     return result
33
34

```

```

35 def main():
36     root = TreeNode(12)
37     root.left = TreeNode(7)
38     root.right = TreeNode(1)
39     root.left.left = TreeNode(9)
40     root.right.left = TreeNode(10)
41     root.right.right = TreeNode(5)
42     print("Reverse level order traversal: " + str(traverse(root)))
43
44
45 main()
46

```

Run

Save

Reset



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

Binary Tree Level Order Traversal (easy)

Next →

Zigzag Traversal (medium)

✓ Completed

⚠ Report an Issue 🤔 Ask a Question