107. Binary Tree Level Order Traversal II (/problems/binary-tree-level-order-traversal-ii/)

May 30, 2020 | 5.6K views

Average Rating: 4.89 (9 votes)

Given a binary tree, return the *bottom-up level order* traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

For example:
Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its bottom-up level order traversal as:

```
[
  [15,7],
  [9,20],
  [3]
]
```

Solution

How to traverse the tree

There are two general strategies to traverse a tree:

- *Depth First Search* (`DFS`)

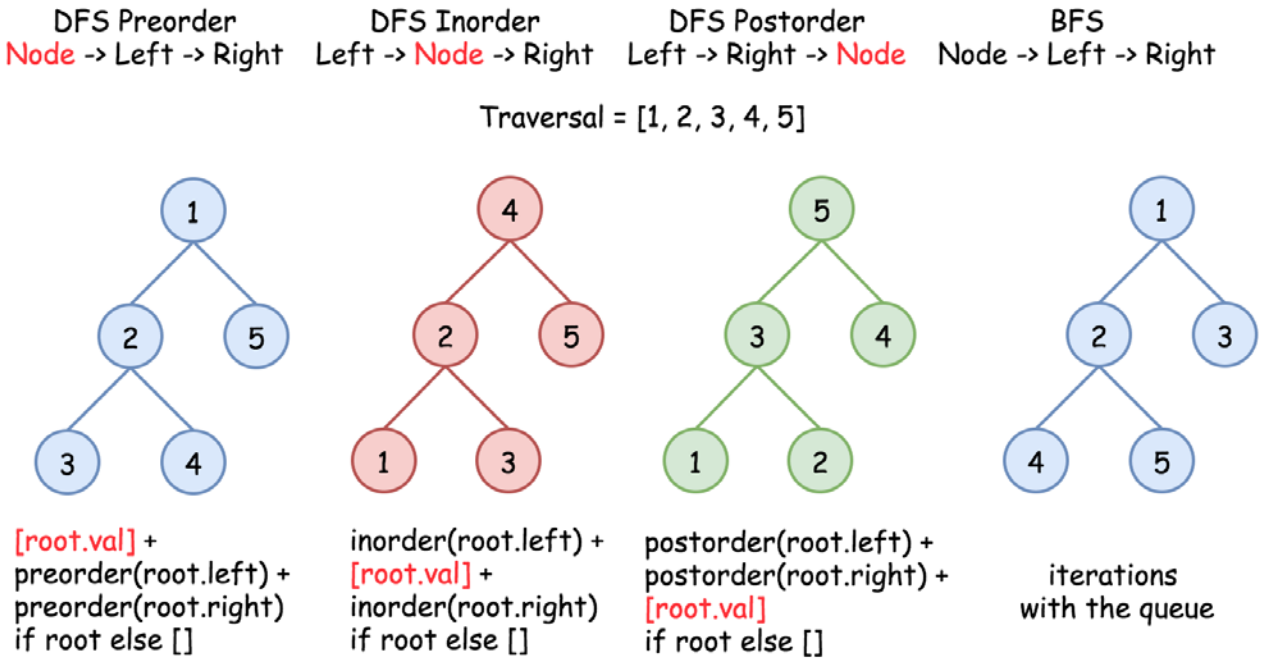
  In this strategy, we adopt the `depth` as the priority, so that one would start from a root and reach all the way down to a certain leaf, and then back to root to reach another branch.

  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 a higher level would be visited before the ones on lower levels.

In the following figure the nodes are enumerated 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 : `[[4, 5], [2, 3], [1]]` . That means we could use one of the `Node->Left->Right` techniques: BFS or DFS Preorder.

We already discussed three different ways (https://leetcode.com/articles/binary-tree-right-side-view/) to implement iterative BFS traversal with the queue, and compared iterative BFS vs. iterative DFS (https://leetcode.com/problems/deepest-leaves-sum/solution/). Let's use this article to discuss the two most simple and fast techniques:

- Recursive DFS.
- Iterative BFS with two queues.

> Note, that both approaches are root-to-bottom traversals, and we're asked to provide bottom-up output. To achieve that, the final result should be reversed.

Approach 1: Recursion: DFS Preorder Traversal

Intuition

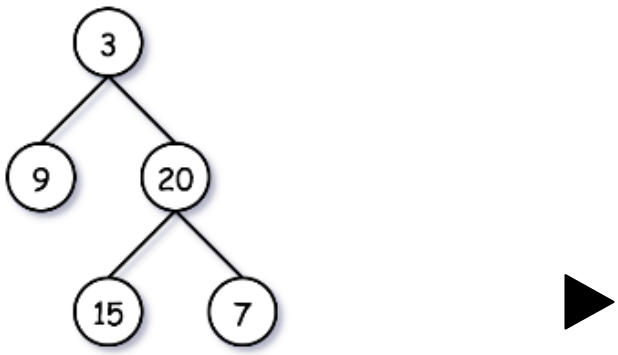
The first step is to ensure that the tree is not empty. The second step is to implement the recursive function `helper(node, level)`, which takes the current node and its level as the arguments.

Algorithm for the Recursive Function

Here is its implementation:

- Initialize the output list `levels`. The length of this list determines which level is currently updated. You should compare this level `len(levels)` with a node level `level`, to ensure that you add the node on the correct level. If you're still on the previous level - add the new level by adding a new list into `levels`.
- Append the node value to the last level in `levels`.
- Process recursively child nodes if they are not `None`: `helper(node.left / node.right, level + 1)`.

Implementation



levels = []

1 / 1

Java Python Copy

```java
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>());

         // append the current node value
         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>> levelOrderBottom(TreeNode root) {
        if (root == null) return levels;
        helper(root, 0);
        Collections.reverse(levels);
        return levels;
    }
}
```

Complexity Analysis

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

Approach 2: Iteration: BFS Traversal

Algorithm

The recursion above could be rewritten in the iteration form.

Let's keep each tree level in the *queue* structure, which typically orders elements in a FIFO (first-in-first-out) manner. In Java one could use `ArrayDeque` implementation of the `Queue` interface (https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.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 downgrade. In Python the queue implementation with a fast atomic `append()` and `popleft()` is `deque` (https://docs.python.org/3/library/collections.html#collections.deque).

Algorithm

- Initialize two queues: one for the current level, and one for the next. Add root into `nextLevel` queue.
- While `nextLevel` queue is not empty:
  - Initialize the current level `currLevel = nextLevel`, and empty the next level `nextLevel`.
  - Iterate over the current level queue:
    - Append the node value to the last level in `levels`.
    - Add first *left* and then *right* child node into `nextLevel` queue.
- Return reversed `levels`.

Implementation

Java Python Copy

```java
class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        List<List<Integer>> levels = new ArrayList<List<Integer>>();
        if (root == null) return levels;

        ArrayDeque<TreeNode> nextLevel = new ArrayDeque() {{ offer(root); }};
        ArrayDeque<TreeNode> currLevel = new ArrayDeque();

        while (!nextLevel.isEmpty()) {
            currLevel = nextLevel.clone();
            nextLevel.clear();
            levels.add(new ArrayList<Integer>());

            for (TreeNode node : currLevel) {
                // append the current node value
                levels.get(levels.size() - 1).add(node.val);

                // process child nodes for the next level
                if (node.left != null)
                    nextLevel.offer(node.left);
                if (node.right != null)
                    nextLevel.offer(node.right);
            }
        }

        Collections.reverse(levels);
        return levels;
    }
}
```

Complexity Analysis

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

Rate this article:

Previous (/articles/binary-tree-right-side-view/) Next (/articles/word-break-ii/)

Comments: 4

Sort By

Type comment here... (Markdown is supported)

Preview Post

(/jyshi302)

jyshi302 (/jyshi302) ★ 0 an hour ago

BFS, use Linked List to store the result

```
class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        LinkedList<List<Integer>> result = new LinkedList<>();
```

Read More

0 Share Reply

(/super_saiyan_2)

super_saiyan_2 (/super_saiyan_2) ★ 27 2 hours ago

Any opinions on my solution? Just a classic BFS, where we utilize a linked list to be able to add elements in the beginning fast.

```
class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
```

Read More

0 Share Reply

(/leetcoding_master)

LeetCoding_Master (/leetcoding_master) ★ 112 3 hours ago

```
class Solution
{
    public List<List<Integer>> levelOrderBottom(TreeNode root)
    {
```

Read More

0 Share Reply

(/gdkou90)

gdkou90 (/gdkou90) ★ 50 5 hours ago

This problem is exactly the one that I was asked during an interview, which was followed by problem 102.

0 Share Reply

SHOW 2 REPLIES
