Binary Tree Level Order Traversal II

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,#,#,15,7}`,

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

return its bottom-up level order traversal as:

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

confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.

OJ's Binary Tree Serialization:
The serialization of a binary tree follows a level order traversal, where '#' signifies a path terminator where no node exists below.

Here's an example:

```
   1
  / \
 2   3
    /
   4
    \
     5
```

The above binary tree is serialized as `"{1,2,3,#,#,4,#,#,5}"`.

Solution 1

The way I see this problem is that it is EXACTLY the same as "Level-Order Traversal I" except that we need to reverse the final container for output, which is trivial. Is there a better idea that fits this problem specifically?

The attached is my current recursive solution. In each function call, we pass in the current node and its level. If this level does not yet exist in the output container, then we should add a new empty level. Then, we add the current node to the end of the current level, and recursively call the function passing the two children of the current node at the next level. This algorithm is really a DFS, but it saves the level information for each node and produces the same result as BFS would.

```
vector<vector<int> > res;

void DFS(TreeNode* root, int level)
{
    if (root == NULL) return;
    if (level == res.size()) // The level does not exist in output
    {
        res.push_back(vector<int>()); // Create a new level
    }

    res[level].push_back(root->val); // Add the current value to its level
    DFS(root->left, level+1); // Go to the next level
    DFS(root->right,level+1);
}

vector<vector<int> > levelOrderBottom(TreeNode *root) {
    DFS(root, 0);
    return vector<vector<int> > (res.rbegin(), res.rend());
}
```

written by stellari original link here

Solution 2

DFS solution:

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

        if(root == null) return wrapList;

        queue.offer(root);
        while(!queue.isEmpty()){
            int levelNum = queue.size();
            List<Integer> subList = new LinkedList<Integer>();
            for(int i=0; i<levelNum; i++) {
                if(queue.peek().left != null) queue.offer(queue.peek().left);
                if(queue.peek().right != null) queue.offer(queue.peek().right);
                subList.add(queue.poll().val);
            }
            wrapList.add(0, subList);
        }
        return wrapList;
    }
}
```

BFS solution:

```
public class Solution {
        public List<List<Integer>> levelOrderBottom(TreeNode root) {
            List<List<Integer>> wrapList = new LinkedList<List<Integer>>();
            levelMaker(wrapList, root, 0);
            return wrapList;
        }

        public void levelMaker(List<List<Integer>> list, TreeNode root, int level
) {
            if(root == null) return;
            if(level >= list.size()) {
                list.add(0, new LinkedList<Integer>());
            }
            levelMaker(list, root.left, level+1);
            levelMaker(list, root.right, level+1);
            list.get(list.size()-level-1).add(root.val);
        }
    }
```

written by SOY original link here

Solution 3

The addFirst() method of LinkedLinked save us from reverse final result.

```
public List<List<Integer>> levelOrderBottom(TreeNode root) {
    LinkedList<List<Integer>> list = new LinkedList<List<Integer>>();
    addLevel(list, 0, root);
    return list;
}

private void addLevel(LinkedList<List<Integer>> list, int level, TreeNode node) {
    if (node == null) return;
    if (list.size()-1 < level) list.addFirst(new LinkedList<Integer>());
    list.get(list.size()-1-level).add(node.val);
    addLevel(list, level+1, node.left);
    addLevel(list, level+1, node.right);
}
```

written by pavel-shlyk original link here