102-二叉树的层序遍历

leetcode #102-二叉树的层序遍历 (中等)

> 给你一个二叉树根节点，请你返回其按 **层序遍历** 得到的节点值。 （即逐层地，从左到右访问所有节点）。
>
> 示例：
> 二叉树：[3,9,20,null,null,15,7], 返回其层序遍历结果：[[3], [9, 20], [15, 7]]。

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) {
        this.val = val;
    }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}
```

解法一：递归

算法描述

从root开始遍历，由于节点只有左右链的信息，通过这些信息只能在纵向上移动。考虑添加一个level信息，如果遍历向下层推进时使得level加1，又使得level的数值与输出到结果二维列表中用于保存该层节点val信息的一维列表levelList有确定关系，那么就可以在对节点x递归调用遍历方法时传入level+1来向levelList中输入x.val。请结合代码注释理解本解法。

时空复杂度

每个节点都会且只会遍历一次(遍历方法的调用)，n个节点的树的时间复杂度为O(n)。

空间复杂度取决于栈深，而栈深与该二叉树的形状有关，如果为链状，达到最大空间复杂度O(n)，如果为完全二叉树(complete binary tree)，栈深为O(logn)。

代码

```java
private List<List<Integer>> res = new ArrayList<List<Integer>>();

public List<List<Integer>> levelOrderTraversalCur(TreeNode root) {
    if(root != null) {
        levelOrderTraversalCur(root, 0);
    }
    return res;
}

/**
 * 传入层号，使得递归调用levelOrderTraversalCur时，传入的同一层节点中，
 * 只有第一个传入的节点会在res中创建一个用于存储本层节点val的list。
 */
private void levelOrderTraversalCur(TreeNode node, int level) {
    // 如下第一个if使得递归调用levelOrderTraversalCur时，传入的同一层节点中，
    // 只有第一个节点会导致res中添加用于保存该层节点值的list。
    // 例如程序开始时，level = 0，此时res中尚无list，所以0 = 0，向res中
    // add一个用于保存0层节点val的list。假设根节点有左孩子l和右孩子r，当程序进行到
    // 后面两个if中递归调用了levelOrderTraversalCur方法，那么当遍历左孩子l
    // 时level = res.size() = 1，于是向res中add一个用于保存1层节点val的list。
    // 之后当右孩子r也进行if(level == res.size())判断时，res.size()比level大1，
    // 不会再向res中添加list。
    if(level == res.size()) {
        res.add(new ArrayList<Integer>());
    }
    // 同一层从左到右的节点，都会先后执行此条，向保存当前层元素的level列表中add自己的val
    res.get(level).add(node.val);
    // node如果有左孩子，递归调用层序遍历方法，传入的level加1
    if(node.left != null) {
        // 注意level+1不可写成level++
        levelOrderTraversalCur(node.left, level+1);
    }
    // node如果有右孩子，递归调用层序遍历方法，传入的level加1
    if(node.right != null) {
        levelOrderTraversalCur(node.right, level+1);
    }
}
```

解法二：队列+迭代

算法描述

观察到递归解法属于尾递归，故有迭代方式的写法。实际上树的层序遍历就是广度优先搜索BFS执行搜索到最后一个节点的过程。本题要求不仅要实现“层序”，输出结果中还要分层，即每层用一个一维列表保存结果。用BFS实现本题要求的层序遍历，通过如下方式实现”分层“效果。

首先root入队，然后在一个考察队列是否为空的while中先取得当前队列的大小size，并在一个循环次数为size的for中出队并加入到本层结果列表中，接着将该出队节点的左右孩子(如果有的话)入队。通过次数为size的for保证了for结束后该层所有节点都依序输出了val，并且其下一层所有节点也均入队。

时空复杂度

每个节点都会入队一次，出队一次，时间复杂度为O(n)。

空间复杂度取决于队列长度，显然不大于n(树只有根节点时等于n)，空间复杂度为O(n)。

代码

```java
public List<List<Integer>> levelOrderTraversalLoop(TreeNode root) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    if(root == null) {
        return res;
    }
    Queue<TreeNode> queue = new LinkedList<>();

    // root不为null时入队
    queue.offer(root);
    while(!queue.isEmpty()) {
        // 从左到右保存当前层的节点的val
        List<Integer> level = new ArrayList<>();
        // 当前队列的大小，实际上就是当前层节点个数，用于作为下一条for语句的边界条件，
        // 使得for遍历后正好使得level保存了当前层所有节点的val
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            // 当前层的节点出队
            TreeNode top = queue.poll();
            // 加入到本层结果list中
            level.add(top.val);
            // 如果top有左孩子，作为下一层的节点入队，由于有size边界控制，
            // 它不会在本轮for循环中输出到level，而是在下一轮for循环中输出
            if(top.left != null) {
                queue.offer(top.left);
            }
            // 如果top有右孩子，作为下一层的节点入队
            // 它不会在本轮for循环中输出到level，而是在下一轮for循环中输出
            if(top.right != null) {
```

```
                queue.offer(top.right);
            }
        }
        // 完成本层元素遍历，加入到结果中
        res.add(level);
    }
    return res;
}
```