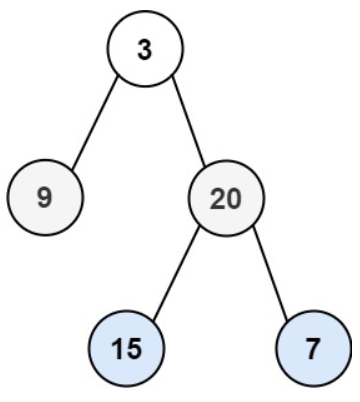
<https://leetcode.com/problems/binary-tree-zigzag-level-order-traversal/>

Given the root of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

**Example 1:**



Input: root = [3,9,20,null,null,15,7]

Output: [[3],[20,9],[15,7]]

**Example 2:**

Input: root = [1]

Output: [[1]]

**Example 3:**

Input: root = []

Output: []

**Constraints:**

* The number of nodes in the tree is in the range [0, 2000].
* -100 <= Node.val <= 100

**Attempt 1: 2023-09-10**

**Solution 1:  BFS + Level order traversal (10 min)**

/\*\*

\* Definition for a binary tree node.

\* public 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;

\* }

\* }

\*/

class Solution {

public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

if(root == null) {

return result;

}

int depth = 0;

Queue<TreeNode> q = new LinkedList<>();

q.offer(root);

while(!q.isEmpty()) {

int size = q.size();

List<Integer> list = new ArrayList<>();

for(int i = 0; i < size; i++) {

TreeNode node = q.poll();

if(depth % 2 == 0) {

list.add(node.val);

} else {

list.add(0, node.val);

}

if(node.left != null) {

q.offer(node.left);

}

if(node.right != null) {

q.offer(node.right);

}

}

result.add(list);

depth++;

}

return result;

}

}

**Solution 2: DFS + Pre-order traversal (30 min)**

/\*\*

\* Definition for a binary tree node.

\* public 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;

\* }

\* }

\*/

class Solution {

public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

helper(root, result, 0);

return result;

}

private void helper(TreeNode root, List<List<Integer>> result, int depth) {

if(root == null) {

return;

}

// 难点：

// 我们也可以使用递归的方法来解，这里实际上用的是先序遍历，递归函数需要

// 一个变量 depth 来记录当前的深度，由于 depth 是从0开始的，假如结果

// res 的大小等于 depth，就需要在结果 res 中新加一个空集，这样可以保证

// res[depth] 不会越界。取出 res[depth] 之后，判断 depth 的奇偶，

// 若其为偶数，则将 node->val 加入 oneLevel 的末尾，若为奇数，则加在

// oneLevel 的开头。然后分别对 node 的左右子结点调用递归函数，此时要传入

// level+1 即可

if(result.size() <= depth) {

List<Integer> newList = new ArrayList<>();

result.add(newList);

}

List<Integer> list = result.get(depth);

if(depth % 2 == 0) {

list.add(root.val);

} else {

list.add(0, root.val);

}

helper(root.left, result, depth + 1);

helper(root.right, result, depth + 1);

}

}

**Refer to**

<https://grandyang.com/leetcode/103/>

这道二叉树的之字形层序遍历是之前那道 [Binary Tree Level Order Traversal](http://www.cnblogs.com/grandyang/p/4051321.html) 的变形，不同之处在于一行是从左到右遍历，下一行是从右往左遍历，交叉往返的之字形的层序遍历。最简单直接的方法就是利用层序遍历，并使用一个变量 cnt 来统计当前的层数（从0开始），将所有的奇数层的结点值进行翻转一下即可，参见代码如下：

解法一：

class Solution {

public:

vector<vector<int>> zigzagLevelOrder(TreeNode\* root) {

if (!root) return {};

vector<vector<int>> res;

queue<TreeNode\*> q{{root}};

int cnt = 0;

while (!q.empty()) {

vector<int> oneLevel;

for (int i = q.size(); i > 0; --i) {

TreeNode \*t = q.front(); q.pop();

oneLevel.push\_back(t->val);

if (t->left) q.push(t->left);

if (t->right) q.push(t->right);

}

if (cnt % 2 == 1) reverse(oneLevel.begin(), oneLevel.end());

res.push\_back(oneLevel);

++cnt;

}

return res;

}

};

我们可以将上面的解法进行优化一下，翻转数组虽然可行，但是比较耗时，假如能够直接计算出每个结点值在数组中的坐标，就可以直接进行更新了。由于每层的结点数是知道的，就是队列的元素个数，所以可以直接初始化数组的大小。此时使用一个变量 leftToRight 来标记顺序，初始时是 true，当此变量为 true 的时候，每次加入数组的位置就是i本身，若变量为 false 了，则加入到 size-1-i 位置上，这样就直接相当于翻转了数组。每层遍历完了之后，需要翻转 leftToRight 变量，同时不要忘了将 oneLevel 加入结果 res，参见代码如下：

解法二：

class Solution {

public:

vector<vector<int>> zigzagLevelOrder(TreeNode\* root) {

if (!root) return {};

vector<vector<int>> res;

queue<TreeNode\*> q{{root}};

bool leftToRight = true;

while (!q.empty()) {

int size = q.size();

vector<int> oneLevel(size);

for (int i = 0; i < size; ++i) {

TreeNode \*t = q.front(); q.pop();

int idx = leftToRight ? i : (size - 1 - i);

oneLevel[idx] = t->val;

if (t->left) q.push(t->left);

if (t->right) q.push(t->right);

}

leftToRight = !leftToRight;

res.push\_back(oneLevel);

}

return res;

}

};

我们也可以使用递归的方法来解，这里实际上用的是先序遍历，递归函数需要一个变量 level 来记录当前的深度，由于 level 是从0开始的，假如结果 res 的大小等于 level，就需要在结果 res 中新加一个空集，这样可以保证 res[level] 不会越界。取出 res[level] 之后，判断 level 的奇偶，若其为偶数，则将 node->val 加入 oneLevel 的末尾，若为奇数，则加在 oneLevel 的开头。然后分别对 node 的左右子结点调用递归函数，此时要传入 level+1 即可，参见代码如下：

解法三：

class Solution {

public:

vector<vector<int>> zigzagLevelOrder(TreeNode\* root) {

vector<vector<int>> res;

helper(root, 0, res);

return res;

}

void helper(TreeNode\* node, int level, vector<vector<int>>& res) {

if (!node) return;

if (res.size() <= level) {

res.push\_back({});

}

vector<int> &oneLevel = res[level];

if (level % 2 == 0) oneLevel.push\_back(node->val);

else oneLevel.insert(oneLevel.begin(), node->val);

helper(node->left, level + 1, res);

helper(node->right, level + 1, res);

}

};