<https://leetcode.ca/all/366.html>

Given a binary tree, collect a tree's nodes as if you were doing this: Collect and remove all leaves, repeat until the tree is empty.

**Example 1:**

Input: [1,2,3,4,5]

1

/ \

2 3

/ \

4 5

Output: [[4,5,3],[2],[1]]

**Explanation:**

Removing the leaves [4,5,3] would result in this tree:

1

/

2

Now removing the leaf [2] would result in this tree:

1

Now removing the leaf [1] would result in the empty tree:

[]

**Example 2:**

Input: root = [1] Output: [[1]]

**Constraints:**

The number of nodes in the tree is in the range [1, 100].

-100 <= Node.val <= 100

**Attempt 1: 2024-07-31**

**Solution 1: DFS + Divide and Conquer (30 min)**

**Style 1:**

/\*\*

 \* 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>> findLeaves(TreeNode root) {

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

        getHeight(root, result);

        return result;

    }

    /\*\*

     \* The most important theory is building relation between

     \* tree's 'height' and result list's 'index', few tips:

     \* (1) Leaf node 'height' = 0 -> mapping to result list's

     \* 'index' 0 position's sublist, all leaf nodes should

     \* stored at sublist which at index = 0 of result list

     \* (2) All nodes 'height' = 1 -> mapping to result list's

     \* 'index' = 1 sublist, 'height' = 2 -> mapping to

     \* result list's 'index' = 2 sublist... etc.

     \* (3) Current node's 'height' depends on Math.max(left

     \* child node's height, right child node's height) + 1

     \* (4) Empty node 'height' = -1 -> which means leaf node's

     \* left and right child node as null, since leaf node itself

     \* has 'height' = 0, its child null node has 'height' = -1

     \*/

    private int getHeight(TreeNode node, List<List<Integer>> result) {

        // Base case: null nodes have height -1

        if (node == null) {

            return -1;

        }

        int leftHeight = getHeight(node.left, result);

        int rightHeight = getHeight(node.right, result);

        int currentHeight = Math.max(leftHeight, rightHeight) + 1;

        // Ensure the result list has enough lists to accommodate the current height

        // Since we traverse from root to leaf nodes, and since its DFS

        // traversal and we use Divide and Conquer tech which means we will

        // traverse to leaf node level first and then start build result list,

        // First we try to add all leaf nodes to result list, leaf node's

        // height = 0, at this moment result list is empty(size = 0), it has

        // no sublist, so we have to create one sublist and put in result list's

        // 'index' = 0 position, use this sublist to hold all leaf nodes.

        // When DFS travese from bottom height = 0 leaf nodes back to one level

        // higher, these round of nodes have height = 1, and at this moment result

        // list only have one sublist which hold all leaf nodes earlier, but now

        // we have to record one level nodes above leaf nodes, so we need another

        // sublist, we create this list and put at 'index' = 1 position, the 'index'

        // again equal to current node's height... etc.

        if (result.size() == currentHeight) {

            result.add(new ArrayList<>());

        }

        result.get(currentHeight).add(node.val);

        // After processing the current node, return its height

        return currentHeight;

    }

}

Time Complexity: Our algorithm is a DFS which runs in O(N)

Space Complexity: Since we return O(N), integers, our space complexity is O(N)

**Refer to chatGPT**

To solve Leetcode 366, "Find Leaves of Binary Tree," you can use a recursive depth-first search (DFS) approach. The key idea is to repeatedly remove leaves from the tree and collect them until the tree is empty.

Here’s a step-by-step solution:

Define a Helper Function: The helper function will find the height of each node. The height is defined as the number of edges from the node to the farthest leaf node. Leaf nodes have a height of 0.

Collect Leaves by Height: Use a list of lists to collect nodes with the same height together.

Recursive Traversal: Traverse the tree recursively to determine the height of each node and add it to the corresponding list.

Edge Cases: Handle edge cases such as an empty tree.

Here’s the complete code for the solution:

/\*\*

\* 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>> findLeaves(TreeNode root) {

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

getHeight(root, result);

return result;

}

private int getHeight(TreeNode node, List<List<Integer>> result) {

if (node == null) {

return -1; // Base case: null nodes have height -1

}

int leftHeight = getHeight(node.left, result);

int rightHeight = getHeight(node.right, result);

int currentHeight = Math.max(leftHeight, rightHeight) + 1;

// Ensure the result list has enough lists to accommodate the current height

if (result.size() == currentHeight) {

result.add(new ArrayList<>());

}

result.get(currentHeight).add(node.val);

// After processing the current node, return its height

return currentHeight;

}

}

**Explanation of the Code**

findLeaves **Method**:

This is the main method that initializes the result list and calls the getHeight method.

getHeight **Method**:

This is a recursive method that calculates the height of each node.

For a null node, the height is defined as -1.

For non-null nodes, it recursively calculates the height of the left and right children.

The height of the current node is Math.max(leftHeight, rightHeight) + 1.

The method ensures that the result list has enough sublists to accommodate nodes of the current height. If not, it adds a new list.

It adds the current node's value to the corresponding list in the result.

Finally, it returns the height of the current node.

**How the Solution Works**

The tree is recursively traversed to determine the height of each node.

Nodes with the same height are collected together.

The result list is built such that each sublist contains nodes with the same height, effectively grouping leaves together and then their parent nodes in subsequent iterations.

This approach ensures that all leaves are collected in one pass, and the tree structure is maintained and processed efficiently.

**Refer to**

<https://www.cnblogs.com/grandyang/p/5616158.html>

**这道题给了我们一个二叉树，让我们返回其每层的叶节点，就像剥洋葱一样，将这个二叉树一层一层剥掉，最后一个剥掉根节点。那么题目中提示说要用DFS来做，思路是这样的，每一个节点从左子节点和右子节点分开走可以得到两个深度，由于成为叶节点的条件是左右子节点都为空，所以我们取左右子节点中较大值加1为当前节点的深度值，知道了深度值就可以将节点值加入到结果res中的正确位置了，求深度的方法我们可以参见**[**Maximum Depth of Binary Tree**](http://www.cnblogs.com/grandyang/p/4051348.html)**中求最大深度的方法，参见代码如下：**

/\*\*

\* 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>> findLeaves(TreeNode root) {

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

helper(root, res);

return res;

}

private int helper(TreeNode root, List<List<Integer>> res) {

if (root == null) return -1;

int depth = 1 + Math.max(helper(root.left, res), helper(root.right, res));

if (depth >= res.size()) res.add(new ArrayList<>());

res.get(depth).add(root.val);

return depth;

}

}

**下面这种DFS方法没有用计算深度的方法，而是使用了一层层剥离的方法，思路是遍历二叉树，找到叶节点，将其赋值为NULL，然后加入leaves数组中，这样一层层剥洋葱般的就可以得到最终结果了：**

/\*\*

\* 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>> findLeaves(TreeNode root) {

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

while (root != null) {

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

root = remove(root, leaves);

res.add(leaves);

}

return res;

}

private TreeNode remove(TreeNode node, List<Integer> leaves) {

if (node == null) return null;

// If the current node is a leaf node (both left and right

// children are null), it adds the node's value to the leaves

// list and returns null (indicating that this node is removed).

if (node.left == null && node.right == null) {

leaves.add(node.val);

return null;

}

node.left = remove(node.left, leaves);

node.right = remove(node.right, leaves);

return node;

}

}

**Explanation of the Code**

findLeaves **Method**:

This is the main method that initializes the result list (res).

It repeatedly calls the remove method to collect leaves until the tree is empty.

For each iteration, it collects the leaves of the current tree and adds them to the result list.

remove **Method**:

This is a recursive method that removes leaves from the tree and collects them.

If the current node is null, it returns null.

**If the current node is a leaf node (both left and right children are null), it adds the node's value to the leaves list and returns null (indicating that this node is removed).**

If the current node is not a leaf node, it recursively calls remove on its left and right children.

It updates the left and right children of the current node with the results of the recursive calls.

Finally, it returns the current node (which may have its children removed).

**How the Solution Works**

The findLeaves method uses a loop to remove all leaves from the tree in each iteration and collect them.

The remove method performs a post-order traversal of the tree, collecting leaf nodes and removing them.

This process is repeated until the tree becomes empty, and all leaves at each level are collected and added to the result list.

This approach effectively removes and collects leaves layer by layer

**Refer to**

<https://algo.monster/liteproblems/366>

**Brute Force**

We can simply implement a solution that does what the problem asks one step at a time.

First, we will run a DFS to find all leaf nodes. Then, we'll remove them from the tree. We'll keep repeating that process until the tree is empty.

Let N denote the number of nodes in the tree.

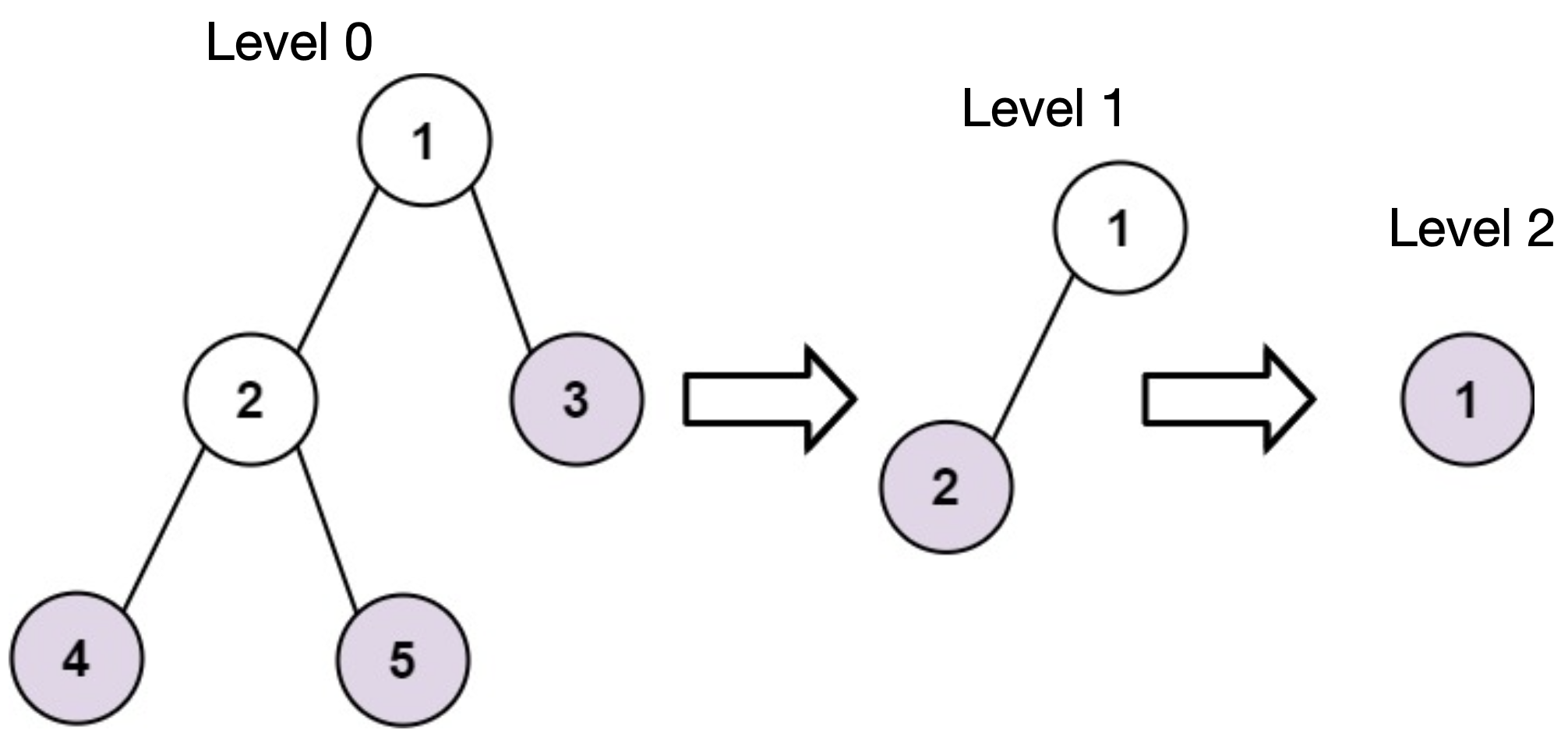
In the worst scenario (line graph), we will repeat this process O(N) times and obtain a time complexity of O(N^2).

However, a more efficient solution exists.

**Full Solution**

Let's denote the level of a node u as the step u will be removed as a leaf node. For convenience, we will start counting steps from 0.

**Example**



Here, nodes 3, 4, 5 have a level of 0. Node 2 has a level of 1 and node 1 has a level of 2.

**How do we find the level of a node?**

One observation we can make is that for a node to be removed as a leaf in some step, all the children of that node have to be removed one step earlier. Obviously, if a node is a leaf node in the initial tree, it will be removed on step 0.

If a node u has one child v, u will be removed one step after v (i.e. level[u] = level[v] + 1).

**However, if a node u has two children v and w, u is removed one step after both v and w get removed. Thus, we obtain level[u] = max(level[v], level[w]) + 1.**

For the algorithm, we will run a DFS and calculate the level of all the nodes in postorder with the method mentioned above. An article about postorder traversal can be found here. For this solution, we need to visit the children of a node before that node itself as the level of a node is calculated from the level of its children. Postorder traversal is suitable for our solution because it does exactly that.

**Time Complexity**

Our algorithm is a DFS which runs in O(N)

**Space Complexity**

Since we return O(N), integers, our space complexity is O(N)

/\*\*

\* 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 {

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

// ans[i] stores all nodes with a level of i

public int dfs(TreeNode u) {

if (u == null) {

return -1;

}

int leftLevel = dfs(u.left);

int rightLevel = dfs(u.right);

int currentLevel = Math.max(leftLevel, rightLevel)

+ 1; // calculate level of current node

while (ans.size()

<= currentLevel) { // create more space in ans if necessary

ans.add(new ArrayList<Integer>());

}

ans.get(currentLevel).add(u.val);

return currentLevel;

}

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

dfs(root);

return ans;

}

}

**Refer to**

[What is the difference between tree depth and height](note://BDC7B166ABD743F29E3B4BC31D6FA0B1)

[L104.Maximum Depth of Binary Tree (Ref.L222,L366)](note://DFEAE5072F224B7F85934A11F3381C70)