<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.

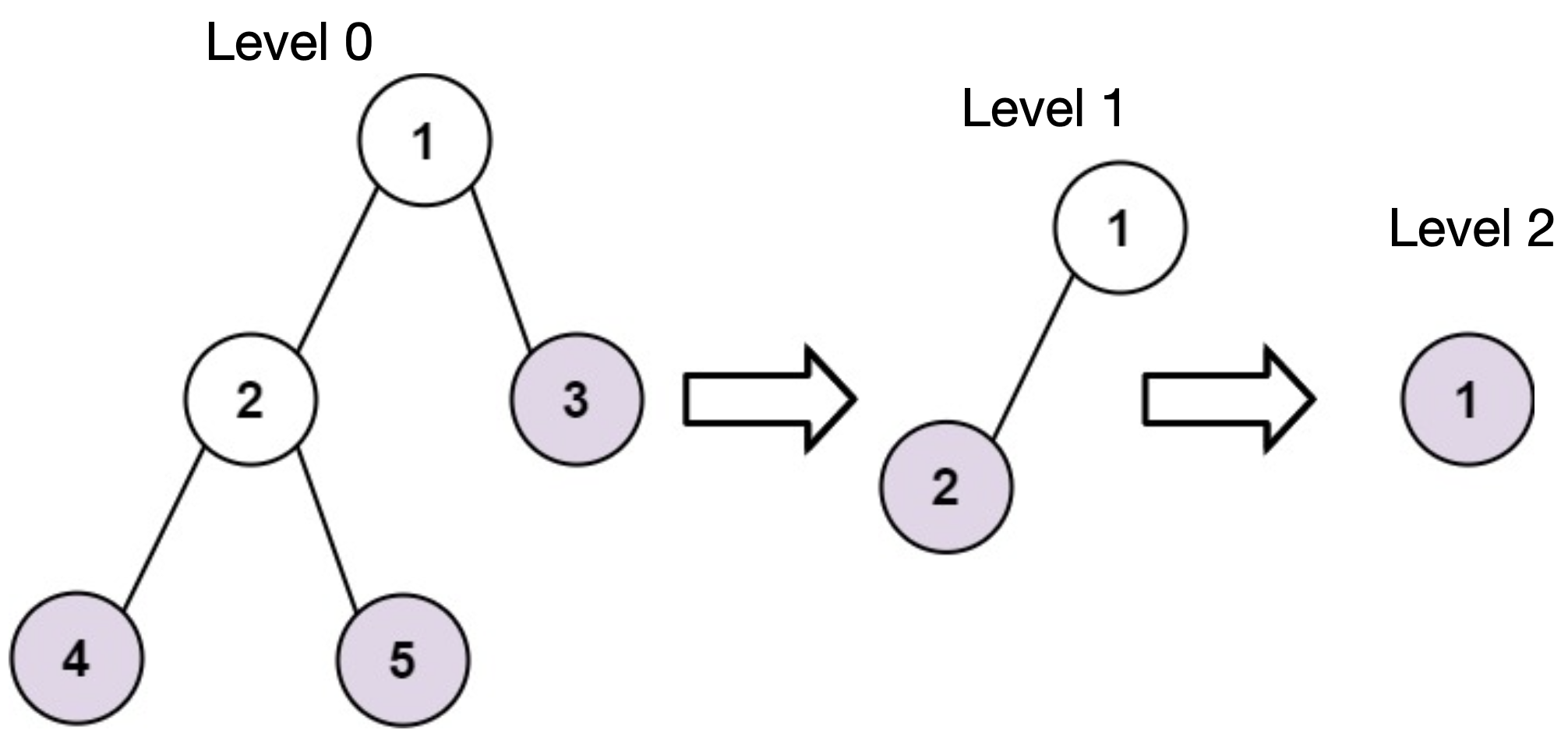
In the worst scenario (line graph), we will repeat this process O(N) times and obtain a time complexity of O(N^2).

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;

}

}

**Compare with Leetcode 1325 solution as "process subtree nodes first, then process current node", we find the second solution (set leaf nodes level by level as null) in Leetcode 366 is "process current node first, then process subtree nodes", what's the difference between ?**

**Refer to chatGPT as "one thing leads to another" Q & A**

**Q1: Compare to Leetcode 1325, the Leetcode 366 has similar requirement as record then remove each leaf node level by level, can you solve it in two ways:**

**1. first process current node itself, then process current node's left and right subtree nodes**

**2. first process current node's left and right subtree nodes, then process current node itself**

LeetCode 366 ("Find Leaves of Binary Tree") can be solved in both ways you mentioned:

First process the current node itself, then process its left and right subtrees.

First process the current node's left and right subtrees, then process the current node itself.

**1. Process Current Node First, Then Subtrees**

In this approach, you first check if the current node is a leaf. If it is, you record it, remove it by returning null, and then move to its subtrees.

import java.util.ArrayList;

import java.util.List;

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

}

}

public class Solution {

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

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

while (root != null) {

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

root = removeLeaves(root, leaves);

result.add(leaves);

}

return result;

}

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

if (node == null) {

return null;

}

// If it's a leaf node, add to the list and return null

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

leaves.add(node.val);

return null;

}

// Otherwise, process its left and right subtrees

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

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

return node;

}

}

**Explanation:**

Remove Current Node if It's a Leaf: If the node has no children (i.e., it's a leaf), it's recorded and then removed by returning null.

Process Subtrees: The left and right subtrees are processed after the current node, which allows the removal of leaves from the bottom up.

Repeat Until Root is Null: This process is repeated until the root becomes null, indicating all nodes have been removed.

**2. Process Subtrees First, Then Current Node**

In this approach, you first process the left and right subtrees to identify and remove leaves, and then check if the current node has become a leaf after its children were processed.

import java.util.ArrayList;

import java.util.List;

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

}

}

public class Solution {

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

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

collectLeaves(root, result);

return result;

}

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

if (node == null) {

return -1;

}

// Process the left and right subtrees first

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

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

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

// Ensure the result list has enough lists for each level

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

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

}

// Add the current node's value to the list corresponding to its height

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

// Return the height of this node, effectively removing it as a leaf

return currHeight;

}

}

**Explanation:**

Process Subtrees First: The left and right subtrees are processed first to determine the "height" or "level" of each node.

Record Node as a Leaf if Appropriate: After processing its children, the current node’s "height" is determined. If it's a leaf (i.e., at the lowest level), it gets added to the corresponding list in result.

Collect Leaves by Level: The currHeight is used to track and categorize leaves by their level, ensuring that all nodes on the same level are grouped together.

**Key Differences:**

**Node-First Approach:** The first approach prioritizes checking and removing the current node if it's a leaf before its children are processed. This ensures that leaves are identified and removed immediately.

**Subtrees-First Approach:** The second approach processes the subtrees first, allowing for a full pass through the tree before evaluating and removing the current node. This approach naturally groups leaves by their level and can simplify the logic in some scenarios.

**Example Execution:**

For both solutions with input:

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

1

/ \

2 3

/ \

4 5

**Expected Output**:

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

Both approaches will yield the correct output, but they differ in how they traverse and process the tree. The choice between them depends on how you prefer to structure your recursion and where you want to focus the logic—on the node itself or on its subtrees.

**Q2: In above Leetcode 366 Solution 2. Process Subtrees First, Then Current Node, the given solution is not removing leaf nodes by setting it to NULL, can we also implement "Solution 2. Process Subtrees First, Then Current Node" as setting leaf nodes to NULL way ?**

Yes, the "Process Subtrees First, Then Current Node" approach in LeetCode 366 can also be implemented by setting leaf nodes to null. The idea is to recursively process the left and right subtrees first, and then decide whether the current node should be removed (i.e., set to null) if it has become a leaf after its children are processed.

Here’s how you can implement this:

**Java Code Implementation**

import java.util.ArrayList;

import java.util.List;

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

}

}

public class Solution {

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

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

removeLeaves(root, result);

return result;

}

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

if (node == null) {

return null;

}

// Process the left and right subtrees first

node.left = removeLeaves(node.left, result);

node.right = removeLeaves(node.right, result);

// If the current node is now a leaf, add it to the result and remove it

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

if (result.isEmpty() || result.size() == 0) {

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

}

result.get(result.size() - 1).add(node.val);

return null;

}

return node;

}

}

**Explanation:**

Recursive Subtree Processing:

The function removeLeaves is called recursively to process the left and right subtrees before the current node.

Check if Current Node is a Leaf:

After processing the subtrees, if both node.left and node.right are null, the current node is a leaf.

Remove the Leaf Node:

If the current node is a leaf, it is added to the list of leaves in result.

The node is then removed from the tree by returning null.

Return the Updated Tree:

If the node is not a leaf, the original node is returned, keeping it in the tree.

**Example Execution:**

For the input:

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

1

/ \

2 3

/ \

4 5

**Expected Output**:

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

**Key Points:**

This approach correctly removes nodes by setting them to null after their children have been processed, ensuring that leaves are removed in a bottom-up manner.

It groups the leaves by level, just like in the first approach, but processes the subtrees before making decisions about the current node.

This method is consistent with the requirement to first process the subtrees and then decide what to do with the current node.

This approach is a flexible way to handle problems that involve modifying the tree structure while traversing it recursively.

**Q3: Actually above new solution ("Solution 2. Process Subtrees First, Then Current Node" as setting leaf nodes to NULL way) is wrong, if test with below input:**

class Solution {

public List<List<Integer>> findLeaves1(TreeNode root) {

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

removeLeaves(root, result);

return result;

}

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

if (node == null) {

return null;

}

// Process the left and right subtrees first

node.left = removeLeaves(node.left, result);

node.right = removeLeaves(node.right, result);

// If the current node is now a leaf, add it to the result and remove it

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

if (result.isEmpty() || result.size() == 0) {

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

}

result.get(result.size() - 1).add(node.val);

return null;

}

return node;

}

public static void main(String[] args) {

/\*\*

\* 1

\* / \

\* 2 3

\* / / \

\* 5 6 4

\*/

TreeSolution so = new TreeSolution();

TreeNode one = so.new TreeNode(1);

TreeNode two = so.new TreeNode(2);

TreeNode three = so.new TreeNode(3);

TreeNode five = so.new TreeNode(5);

TreeNode six = so.new TreeNode(6);

TreeNode four = so.new TreeNode(4);

one.left = two;

one.right = three;

two.left = five;

three.left = six;

three.right = four;

List<List<Integer>> result = so.findLeaves1(one);

System.out.println(result);

}

}

**the output is wrong as [[5, 2, 6, 4, 3, 1]], not the expect result as [[5, 6, 4], [2, 3], [1]], analyze why its wrong and how to fix ?**

**The reason the provided solution produces the incorrect output is that it does not properly group nodes based on their "height" in the tree when they are removed as leaves. In your solution, when a node becomes a leaf, it is immediately added to the current list in the result, regardless of whether it's at the same level as previously removed leaves.**

**Issue Breakdown:**

**Incorrect Grouping by Height:**

**The issue arises because the current implementation does not differentiate between nodes at different levels (or heights) of the tree. All leaves, no matter their level, are added to the same list.**

**Accumulation of Leaves in a Single List:**

**As a result, when a leaf node is removed, it is added to the last list in the result, which causes all nodes to be grouped together, resulting in [[5, 2, 6, 4, 3, 1]].**

**How to Fix:**

**To fix this issue, we need to ensure that nodes at the same height in the tree are grouped together. The height of a node is defined by the maximum height of its children plus one.**

**Corrected Solution:**

We can modify the function so that it calculates the height of each node and uses this height to determine the correct list to place the node in the result.

import java.util.ArrayList;

import java.util.List;

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

}

}

public class Solution {

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

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

findLeavesHelper(root, result);

return result;

}

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

if (node == null) {

return -1;

}

// Process the left and right subtrees first

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

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

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

// Ensure the result list has enough lists for each level

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

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

}

// Add the current node's value to the list corresponding to its height

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

// Return the height of this node, effectively removing it as a leaf

return currHeight;

}

public static void main(String[] args) {

/\*\*

\* 1

\* / \

\* 2 3

\* / / \

\* 5 6 4

\*/

Solution so = new Solution();

TreeNode one = new TreeNode(1);

TreeNode two = new TreeNode(2);

TreeNode three = new TreeNode(3);

TreeNode five = new TreeNode(5);

TreeNode six = new TreeNode(6);

TreeNode four = new TreeNode(4);

one.left = two;

one.right = three;

two.left = five;

three.left = six;

three.right = four;

List<List<Integer>> result = so.findLeaves(one);

System.out.println(result);

}

}

**Explanation of the Fix:**

**Calculate Height Recursively:** The findLeavesHelper method calculates the height of each node by recursively calculating the heights of its children. The height of a node is defined as the maximum of its left and right subtree heights plus one.

**Group Nodes by Height:** Nodes at the same height are grouped together in the result list. This is done by ensuring the result list has enough sublists to accommodate all possible heights.

**Correct Grouping:** With this approach, nodes are grouped correctly based on their height, leading to the correct output.

**Example Output:**

For the given input:

1

/ \

2 3

/ / \

5 6 4

The correct output will be:

[[5, 6, 4], [2, 3], [1]]

**Summary:**

The problem with the initial solution was that it didn't properly group nodes by their height.

By calculating and grouping nodes by their height, the issue is resolved, and the solution now correctly outputs leaves level by level.

**Q4: Above "Process Subtrees First, Then Current Node" solution encounter "height" issue, then why in below solution, "which process current node first then process subtree nodes" we don't encounter "height" issue like above solution ?**

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;

}

public static void main(String[] args) {

/\*\*

\* 1

\* / \

\* 2 3

\* / / \

\* 5 6 4

\*/

TreeSolution so = new TreeSolution();

TreeNode one = so.new TreeNode(1);

TreeNode two = so.new TreeNode(2);

TreeNode three = so.new TreeNode(3);

TreeNode five = so.new TreeNode(5);

TreeNode six = so.new TreeNode(6);

TreeNode four = so.new TreeNode(4);

one.left = two;

one.right = three;

two.left = five;

three.left = six;

three.right = four;

List<List<Integer>> result = so.findLeaves(one);

System.out.println(result);

}

}

The reason why the "process current node first then process subtree nodes" approach doesn't encounter the same "height" issue as the "process subtree nodes first then process current node" approach lies in the order and logic of how nodes are removed.

**Breakdown of the Two Approaches**

**1. Process Current Node First, Then Subtrees (Your Approach)**

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.

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

leaves.add(node.val);

return null;

}

// Recursively process the left and right subtrees

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

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

return node;

}

**Immediate Removal of Leaves: In this approach, when the remove function encounters a leaf node (both left and right are null), it immediately adds the node’s value to the leaves list and returns null, effectively removing the node from the tree.**

**Top-Down Processing: Since each node is checked whether it's a leaf before processing its children, the leaves of the current tree structure are removed first in each iteration. After removing all the current leaves, the next iteration will remove the leaves of the remaining tree structure, which were not leaves in the previous iteration but became leaves after their children were removed.**

**Natural Grouping: Each time remove is called on the entire tree, it removes one "level" of leaves. These leaves are naturally at the same "height" (from the bottom of the tree), and they are grouped together in the leaves list during that iteration. This is why the solution correctly groups leaves without explicitly needing to calculate the height.**

**2. Process Subtrees First, Then Current Node (Other Approach)**

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

if (node == null) {

return -1;

}

// Process the left and right subtrees first

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

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

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

// Ensure the result list has enough lists for each level

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

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

}

// Add the current node's value to the list corresponding to its height

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

// Return the height of this node, effectively removing it as a leaf

return currHeight;

}

**Height-Based Grouping:** Here, the findLeavesHelper function processes the left and right subtrees first and then determines the height of the current node based on the heights of its children. Nodes are grouped by their calculated height.

**Explicit Height Calculation:** This approach explicitly calculates and uses the height of each node to group nodes correctly. Without this explicit grouping by height, nodes at different heights could end up in the same list, as seen in the original problem.

**Why the Current Node First Approach Works Without Height Calculation:**

**Implicit Level Grouping:** **By removing leaves first (before processing subtrees), the algorithm naturally divides the tree into levels. Each iteration of the remove function processes and removes the current leaves, effectively lowering the "height" of the remaining nodes. This means that nodes are grouped by their "level" in successive iterations, which aligns with their height without needing explicit height tracking.**

**Order of Operations: The order of operations in this approach ensures that once a node is identified as a leaf and removed, it’s not revisited in subsequent iterations. The remaining nodes gradually become the next level of leaves, which are then removed in the following iteration.**

**Conclusion:**

The "process current node first" method works correctly without height calculation because it leverages the natural structure of the tree and removes leaves iteratively, ensuring that nodes are removed level by level. In contrast, the "process subtrees first" method requires explicit height calculations to ensure proper grouping because it processes nodes in a different order, which doesn't inherently group nodes by their level in the tree.

**Q5: If not using iteration way but using pure recursion way and keep "process current node first then process subtree nodes" style, how the solution looks like ?**

If you want to use a pure recursive approach while maintaining the "process current node first, then process subtree nodes" style, the solution can be structured similarly to how the iterative approach works, but fully within a recursive framework. In this method, we would still remove leaves level by level, but the logic would be embedded entirely within recursive calls.

Here's how the solution would look:

**Java Code Implementation**

import java.util.ArrayList;

import java.util.List;

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

}

}

public class Solution {

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

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

// We can call a helper function that modifies the result list

// directly while it recursively processes nodes.

removeLeaves(root, res);

return res;

}

// The helper function will return the "height" of the node, while

// adding leaves to the appropriate list in `res`.

private int removeLeaves(TreeNode node, List<List<Integer>> res) {

if (node == null) return -1; // Return -1 to indicate null, making leaf node height as 0

// First, remove the current node if it's a leaf.

int leftHeight = removeLeaves(node.left, res);

int rightHeight = removeLeaves(node.right, res);

// The current node's height is determined by the max height of its children.

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

// Ensure the `res` list is large enough to accommodate the current height.

if (currHeight >= res.size()) {

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

}

// Add the current node's value to the list corresponding to its height.

res.get(currHeight).add(node.val);

// If the node is a leaf (both children are null), we return null to remove it.

return currHeight;

}

public static void main(String[] args) {

/\*\*

\* 1

\* / \

\* 2 3

\* / / \

\* 5 6 4

\*/

Solution so = new Solution();

TreeNode one = new TreeNode(1);

TreeNode two = new TreeNode(2);

TreeNode three = new TreeNode(3);

TreeNode five = new TreeNode(5);

TreeNode six = new TreeNode(6);

TreeNode four = new TreeNode(4);

one.left = two;

one.right = three;

two.left = five;

three.left = six;

three.right = four;

List<List<Integer>> result = so.findLeaves(one);

System.out.println(result);

}

}

**Explanation:**

Recursive Tree Traversal:

The removeLeaves function recursively traverses the tree, first checking the current node before traversing its left and right children.

Determine Node Height:

Each node's height is determined by the maximum height of its left and right children, plus one.

Grouping Nodes by Height:

The current node's height dictates which list in the result res it should be added to. The result list is dynamically expanded to accommodate the height of the node if necessary.

Remove Nodes:

If a node is a leaf (both left and right children are null), it is added to the list corresponding to its height and the node itself is effectively "removed" by the recursive return process (the node's reference is not maintained, resulting in it being excluded from the tree).

Recursive Removal:

As the function unwinds, each node is evaluated, grouped by height, and removed if it has become a leaf after its children are processed.

**Output:**

For the input:

1

/ \

2 3

/ / \

5 6 4

The output will be:

[[5, 6, 4], [2, 3], [1]]

**Summary:**

This solution maintains the "process current node first, then process subtree nodes" style.

The key difference is that it relies on recursive height calculation and grouping, without needing iterative loops.

It effectively removes nodes level by level, similar to the iterative solution, but the recursive structure allows it to be more elegant and pure in its approach.

**Q6: Your above solution is pure recursion but you didn't follow the requirement as "process current node first, it go into process left and right subtree nodes first", can you write a pure recursion solution but "process current node first then move on to subtree nodes" ?**

Certainly! Below is a pure recursion solution for Leetcode 366 that follows the requirement to "process the current node first, then move on to the left and right subtree nodes."

**Java Code Implementation**

import java.util.ArrayList;

import java.util.List;

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

}

}

public class Solution {

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

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

removeLeaves(root, res);

return res;

}

private boolean removeLeaves(TreeNode node, List<List<Integer>> res) {

if (node == null) return false;

// If current node is a leaf, add it to the current leaves list.

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

if (res.isEmpty() || res.get(res.size() - 1).isEmpty()) {

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

}

res.get(res.size() - 1).add(node.val);

return true; // return true indicating this node is a leaf and should be removed

}

// Process the left subtree

if (removeLeaves(node.left, res)) {

node.left = null; // Remove the left leaf

}

// Process the right subtree

if (removeLeaves(node.right, res)) {

node.right = null; // Remove the right leaf

}

// Return false because this node is not a leaf after processing its children

return false;

}

public static void main(String[] args) {

/\*\*

\* 1

\* / \

\* 2 3

\* / / \

\* 5 6 4

\*/

Solution so = new Solution();

TreeNode one = new TreeNode(1);

TreeNode two = new TreeNode(2);

TreeNode three = new TreeNode(3);

TreeNode five = new TreeNode(5);

TreeNode six = new TreeNode(6);

TreeNode four = new TreeNode(4);

one.left = two;

one.right = three;

two.left = five;

three.left = six;

three.right = four;

List<List<Integer>> result = so.findLeaves(one);

System.out.println(result);

}

}

**Explanation:**

Process Current Node First:

The method removeLeaves processes the current node first by checking if it's a leaf node (i.e., both its left and right children are null).

**If it is a leaf, it adds the node's value to the current leaves list in res and returns true to indicate that the node should be removed.**

Recursive Removal of Leaves:

After processing the current node, the method recursively processes the left and right children.

If the left child is a leaf and is removed, the left pointer is set to null. The same is done for the right child.

Return Value:

The return value of removeLeaves is true if the current node is a leaf, prompting its parent to remove it.

If the current node is not a leaf, it returns false, ensuring that the parent node remains in the tree.

Result List Handling:

The result list res keeps track of the nodes removed at each level. A new list is added to res each time the method finds a new set of leaves.

**Output:**

For the input:

1

/ \

2 3

/ / \

5 6 4

The output will be:

[[5, 6, 4], [2, 3], [1]]

**Summary:**

This solution strictly adheres to the requirement of "processing the current node first, then processing the left and right subtree nodes."

It removes the leaves in a pure recursive manner without using any iterative constructs, while ensuring that nodes are grouped by their removal order from the bottom up.

**Refer to**

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

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

[L872.Leaf-Similar Trees (Ref.L366)](note://WEBf218f3479bd7b283fc01e5af3f2f9878)

[L1325.Delete Leaves With a Given Value (Ref.L366)](note://WEB1392ba1762b507a4c5fa65955a6fbd7c)