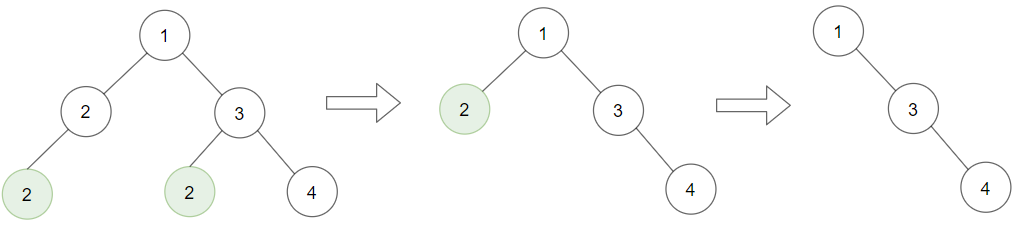
<https://leetcode.com/problems/delete-leaves-with-a-given-value/description/>

Given a binary tree root and an integer target, delete all the **leaf nodes** with value target.

Note that once you delete a leaf node with value target**,**if its parent node becomes a leaf node and has the value target, it should also be deleted (you need to continue doing that until you cannot).

**Example 1:**

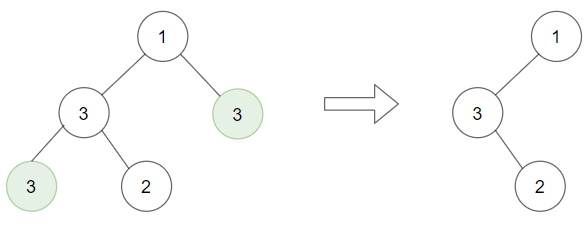


**Input:** root = [1,2,3,2,null,2,4], target = 2

**Output:** [1,null,3,null,4]

**Explanation:** Leaf nodes in green with value (target = 2) are removed (Picture in left). After removing, new nodes become leaf nodes with value (target = 2) (Picture in center).

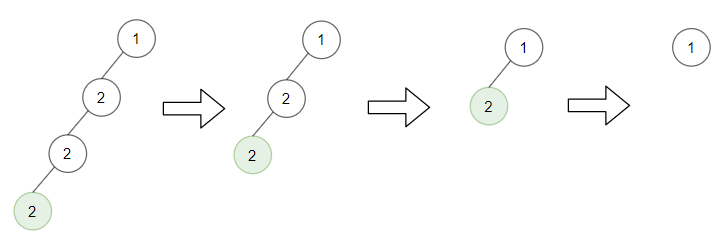
**Example 2:**



**Input:** root = [1,3,3,3,2], target = 3

**Output:** [1,3,null,null,2]

**Example 3:**



**Input:** root = [1,2,null,2,null,2], target = 2

**Output:** [1]

**Explanation:** Leaf nodes in green with value (target = 2) are removed at each step.

**Constraints:**

The number of nodes in the tree is in the range [1,3000].

1 <= Node.val, target <= 1000

**Attempt 1: 2024-08-07**

**Solution 1: DFS + Divide and Conquer (120 min)**

**Wrong Solution**

**The wrong solution use the same style as L366, which process current node first then move on to its left and right subtree**

Input

root = [1,2,3,2,null,2,4]

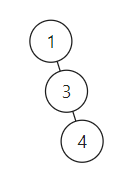
target = 2

Output

[1,2,3,null,null,null,4]

Expected

[1,null,3,null,4]



**The wrong solution fails for the given test case because it checks if the current node is a leaf and has the target value before processing its children. Therefore, it misses the scenario where a node becomes a leaf after its children are removed.**

**To fix this, we should first recursively remove the target leaves from the left and right subtrees, and then check if the current node itself has become a leaf with the target value.**

**In this corrected solution:**

**Process Subtrees First: The left and right subtrees are processed first. This ensures that all target leaves are removed from the subtrees before we check the current node.**

**Check Current Node: After processing the subtrees, we check if the current node has become a leaf and if its value matches the target. If both conditions are met, the current node is removed by returning null.**

/\*\*

 \* 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 TreeNode removeLeafNodes(TreeNode root, int target) {

        return helper(root, target);

    }

    private TreeNode helper(TreeNode root, int target) {

        if(root == null) {

            return null;

        }

// In wrong solution we process current node first and lose the ability

// to check if current node becomes leaf node after processing its left and

// right subtree later, since the processing of its left and right subtree

// only happen after processing current node, the processing order

// chronologically block the possibility to come back and process current

// node even it turn into leaf node

// e.g root = [1,2,3,2,null,2,4], target = 2

// In wrong solution, the first 2([1,{2},3,2,null,2,4]) ideally should become

// leaf node after we remove second 2([1,2,3,{2},null,2,4]) as leaf node match

// target value 2, but since we only check first 2 once and identify its not

// a leaf node at that moment before processing second 2, even after we remove

// second 2 as leaf node, chronologically, since the statement of check first 2

// statement("if(root.left == null && root.right == null && root.val == target)")

// before move on to its left and right subtree statement("helper(root.left, target),

// helper(root.right, target)"), we won't go back to first 2 and check first 2's

// updated new status.

        if(root.left == null && root.right == null && root.val == target) {

            return null;

        }

// Then move on to its left and right subtree

        root.left = helper(root.left, target);

        root.right = helper(root.right, target);

        return root;

    }

}

**Correct 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 TreeNode removeLeafNodes(TreeNode root, int target) {

        return helper(root, target);

    }

    private TreeNode helper(TreeNode root, int target) {

        if(root == null) {

            return null;

        }

// In correct solution we process current node's left and right subtree first

        root.left = helper(root.left, target);

        root.right = helper(root.right, target);

// Then move on to current node itself

        if(root.left == null && root.right == null && root.val == target) {

            return null;

        }

        return root;

    }

}

**Explain why wrong solution failed**

**简单说对于 Leetcode 1325 如果按照错误的思路先处理当前 node 再处理它的左右子树上的 node，由于模版采用的是标准 DFS 模版 (1.base -> 2.处理当前层 -> 3.递归到小问题)，在处理完步骤2的当前层以后，如果判断当前 node 不是 leaf node，那么即使处理完步骤3，也就是处理完递归后的当前 node 的左右子树之后也不会再反过来做一遍步骤2，也就是说不会再判断一次当前 node 是不是 leaf node，哪怕因为处理完步骤3后把当前 node 已经变成了 leaf node 也不会再做一次判断，所以永远失去了去掉变更为 leaf node 后的当前 node 的能力**

**而 Leetcode 1325 正确的解法中，即先处理当前node的左右子树上的 node 再处理当前 node，由于模版采用的是标准 Divide and Conquer 模版 (1.base -> 2.递归到小问题 -> 3.处理当前层)，在处理完步骤3的当前层的返回过程中 (利用递归天然的自动返回上一层属性)，我们会对到达当前 node 的递归途中之前的 parent node 的状态重新审视，也就是 re-evaluate，也就有了去掉变为 leaf node 的当前 node 的能力**

**这是两个模版能力上的差异，DFS模版不具备 re-evaluate 的 parent node 的能力，但是 Divide and Conquer 模版具备**

**以下是参考chatGPT的正确和错误的思路的推导过程**

**Refer to chatGPT**

**Question:**

For Leetcode 1325 i have below correct solution as:

class Solution {

public TreeNode removeLeafNodes(TreeNode root, int target) {

return helper(root, target);

}

private TreeNode helper(TreeNode root, int target) {

if(root == null) {

return null;

}

root.left = helper(root.left, target);

root.right = helper(root.right, target);

if(root.left == null && root.right == null && root.val == target) {

return null;

}

return root;

}

}

and wrong solution as:

class Solution {

public TreeNode removeLeafNodes(TreeNode root, int target) {

return helper(root, target);

}

private TreeNode helper(TreeNode root, int target) {

if(root == null) {

return null;

}

if(root.left == null && root.right == null && root.val == target) {

return null;

}

root.left = helper(root.left, target);

root.right = helper(root.right, target);

return root;

}

}

**The only difference between correct and wrong solution is the order to proceed current node first or proceed current node's left and right subtree first.**

**Can you use below example to elaborate what's the deep reason in wrong solution which proceed current node first failed to remove first node 2 in input, the node initially as non-leaf node but turns out to leaf node after we remove second node 2 in input as leaf node ?**

Input: root = [1,2,3,2,null,2,4], target = 2

Correct solution output: [1,null,3,null,4]

wrong solution output: [1,2,3,null,null,null,4]

**chatGPT Answer:**

The core issue with the wrong solution in the provided example is that it checks the current node's status (whether it's a leaf with the target value) before recursively processing its children. This approach fails to account for situations where the removal of child nodes transforms the current node into a leaf.

Let’s walk through the given input to illustrate why this happens:

Input:

root = [1,2,3,2,null,2,4], target = 2

Tree structure:

1

/ \

2 3

/ / \

2 2 4

**Correct Solution Walkthrough:**

private TreeNode helper(TreeNode root, int target) {

if(root == null) {

return null;

}

root.left = helper(root.left, target); // Process the left subtree first

root.right = helper(root.right, target); // Process the right subtree next

if(root.left == null && root.right == null && root.val == target) {

return null; // Now check if the current node is a leaf and has the target value

}

return root;

}

**Steps:**

Starting at root (1):

Recurse on the left subtree (2).

At node (2):

Recurse on the left subtree (2).

At node (2) [left leaf]:

No children, return null since it's a leaf with the target value 2.

The parent node (2) now has no left child.

**Back at parent node (2):**

**Recurse on the right subtree (which is null).**

**Now, the parent node (2) itself is a leaf and has the target value 2, so it is removed.**

Back at root (1):

Recurse on the right subtree (3).

At node (3):

Recurse on the left subtree (2).

At node (2) [right leaf]:

No children, return null since it's a leaf with the target value 2.

The parent node (3) now has no left child.

Back at node (3):

Recurse on the right subtree (4).

Node (3) is not a leaf and doesn't need removal, so it remains.

Final tree structure:

1

\

3

\

4

**Wrong Solution Walkthrough:**

private TreeNode helper(TreeNode root, int target) {

if(root == null) {

return null;

}

if(root.left == null && root.right == null && root.val == target) {

return null; // Process current node first

}

root.left = helper(root.left, target); // Process the left subtree afterward

root.right = helper(root.right, target); // Process the right subtree afterward

return root;

}

**Steps:**

Starting at root (1):

Immediately check if it's a leaf and has the target value. It doesn't, so proceed.

Recurse on the left subtree (2).

At node (2):

Check if it's a leaf and has the target value. It doesn’t (has a left child), so proceed.

Recurse on the left subtree (2).

At node (2) [left leaf]:

No children, return null since it's a leaf with the target value 2.

The parent node (2) now has no left child.

**Back at parent node (2):**

**Recurse on the right subtree (which is null).**

**Since the parent node (2) was not re-evaluated after its children were processed, it’s not removed, even though it has now become a leaf with the target value 2.**

Back at root (1):

Recurse on the right subtree (3).

At node (3):

Check if it's a leaf and has the target value. It doesn’t, so proceed.

Recurse on the left subtree (2).

At node (2) [right leaf]:

No children, return null since it's a leaf with the target value 2.

The parent node (3) now has no left child.

Back at node (3):

Recurse on the right subtree (4).

Node (3) is not a leaf and doesn't need removal, so it remains.

Final tree structure:

1

/ \

2 3

\

4

**Key Takeaway:**

Correct Solution: By processing the children first (root.left = helper(root.left, target); root.right = helper(root.right, target);) and then checking the current node, the algorithm accurately removes nodes that become leaves after their children are removed.

Wrong Solution: By checking the current node first (if(root.left == null && root.right == null && root.val == target) { return null; }) before processing its children, the algorithm misses removing nodes that become leaves after their children are removed.

**In your specific case, the second node 2 (the one at the left of the root) becomes a leaf after its left child is removed, but since the wrong solution already checked this node before processing its children, it doesn’t get removed when it becomes a leaf.**

**Why in Leetcode 366 we check current node status before recursively check on its left and right subtrees, but in Leetcode 1325 we recursively check current node's left and right subtree leaf nodes status before check current node ?**

**Refer to chatGPT**

The difference in checking the current node's status before or after recursively processing its children in LeetCode problems 366 and 1325 arises from the specific requirements and goals of each problem.

**LeetCode 366: Find Leaves of Binary Tree**

In this problem, the goal is to collect all leaves at each level and then remove them from the tree. The process is repeated until the tree is empty.

Remove Leaves Approach:

First, we check if the current node is a leaf (both left and right children are null). If it is, we add it to the list of leaves and return null to indicate that the node should be removed.

Then, we recursively process the left and right subtrees to find and remove other leaves.

By checking the current node first, we immediately identify and handle leaf nodes at each level. This ensures that we correctly remove all leaves at the current level before moving deeper into the tree.

**LeetCode 1325: Delete Leaves With a Given Value**

In this problem, the goal is to remove leaves with a specific value and to repeat the process if removing a leaf causes its parent to become a new leaf with the same value.

Remove Specific Leaves Approach:

First, we recursively process the left and right subtrees to ensure that any leaves with the target value are removed.

Then, we check if the current node itself has become a leaf with the target value. If it has, we remove it by returning null.

By processing the children first, we ensure that we have correctly removed all target leaves from the subtrees before we check and potentially remove the current node. This is crucial because removing a leaf might turn its parent into a new leaf, which also needs to be checked for removal.

**Summary**

LeetCode 366:

Collect and remove all leaves layer by layer.

Check if the current node is a leaf first, then process children.

Immediate identification of leaves is necessary for accurate layer-wise removal.

LeetCode 1325:

Remove leaves with a specific value and check if parents become new leaves.

Process children first, then check the current node.

Ensures that any newly formed leaves (after child removal) are correctly identified and removed.

Here's the side-by-side comparison for clarity:

**LeetCode 366 Solution**

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;

}

// Check if the current node is a leaf

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

leaves.add(node.val);

return null;

}

// Process the left and right subtrees

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

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

return node;

}

}

**LeetCode 1325 Solution**

public class Solution {

public TreeNode removeLeafNodes(TreeNode root, int target) {

if (root == null) {

return null;

}

// Process the left and right subtrees first

root.left = removeLeafNodes(root.left, target);

root.right = removeLeafNodes(root.right, target);

// Check if the current node is a leaf with the target value

if (root.left == null && root.right == null && root.val == target) {

return null;

}

return root;

}

}

In summary, the order of checking the current node versus its children depends on the specific requirements of the problem and what needs to be achieved at each step of the recursion.

**Refer to**

[L366.Find Leaves of Binary Tree (Ref.L104,L872)](note://WEBaec386e979a0193801a395a9a33ba86d)