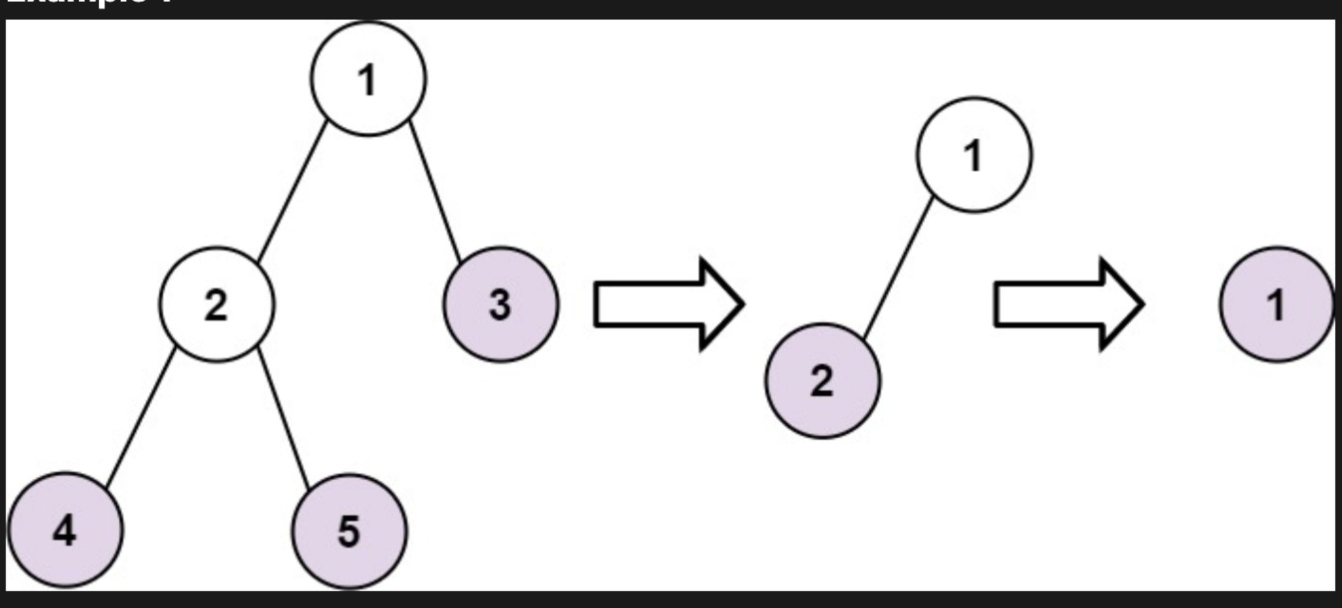
# 366. Find Leaves of Binary Tree

Given the `root` of a binary tree, collect a tree's nodes as if you were doing this:

- Collect all the leaf nodes.
- Remove all the leaf nodes.
- Repeat until the tree is empty.

**Example 1:**



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

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

**Explanation:**

`[[3,5,4],[2],[1]]` and `[[3,4,5],[2],[1]]` are also considered correct answers since per each level it does not matter the order on which elements are returned.

**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`

## Solution

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.

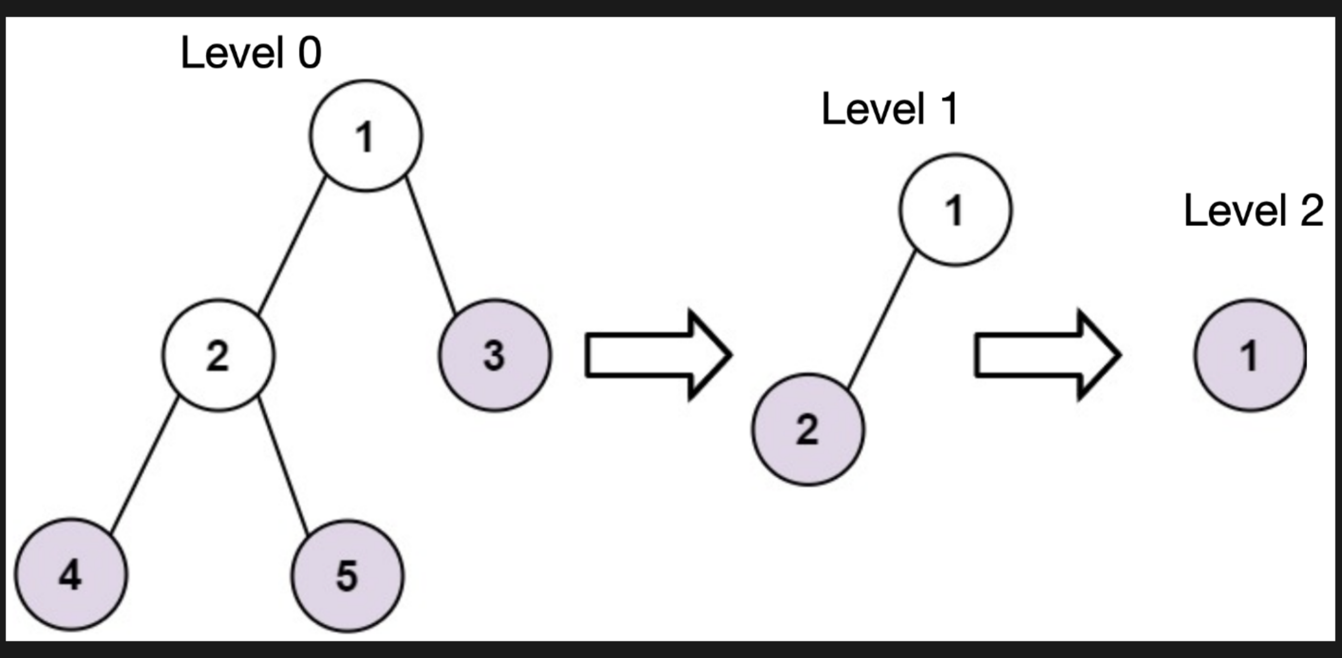
Let $N$ denote the number of nodes in the tree.

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.

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 $O(N)$.

**Time Complexity:** $O(N)$

### Space Complexity

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

**Space Complexity:** $O(N)$

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
 * right(right)
 * {}
 * };
 */
class Solution {
    vector<vector<int>> ans;  // ans[i] stores all nodes with a level of i
   public:
    int dfs(TreeNode* u) {    // dfs function returns the level of current node
        if (u == nullptr) {
            return -1;
        }
        int leftLevel = dfs(u->left);
        int rightLevel = dfs(u->right);
        int currentLevel =
            max(leftLevel, rightLevel) + 1;  // calculate level of current node
        while (ans.size() <=
               currentLevel) {  // create more space in ans if necessary
            ans.push_back({});
        }
        ans[currentLevel].push_back(u->val);
        return currentLevel;
    }
    vector<vector<int>> findLeaves(TreeNode* root) {
        dfs(root);
        return ans;
    }
};
```

```java
/**
 * 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;
    }
}
```

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def findLeaves(self, root: Optional[TreeNode]) -> List[List[int]]:
        ans = [[]]  # ans[i] stores all nodes with a level of i

        def dfs(u):
            if u == None:
                return -1
            leftLevel = dfs(u.left)
            rightLevel = dfs(u.right)
            currentLevel = (
                max(leftLevel, rightLevel) + 1
            )  # calculate level of current node
            while len(ans) <= level:  # create more space in ans if necessary
                ans.append([])
            ans[level].append(u.val)
            return level

        dfs(root)
        return ans
```