# 1469. Find All The Lonely Nodes

Easy | Tree | Depth-First Search | Breadth-First Search | Binary Tree

Leetcode Link

## Problem Description

In this problem, you are given a binary tree and need to find all the "lonely" nodes within it. A lonely node is defined as a node that is the only child of its parent, meaning it has no sibling. The node, in this context, could be either a left child with no right sibling or a right child with no left sibling. It is important to note that the root of the tree is not considered lonely since it does not have a parent. The goal is to return a list of the values of these lonely nodes. The order of values in the output list is not important.

## Intuition

The solution to this problem uses a classic tree traversal approach using a Depth-First Search (DFS) algorithm. The idea is to traverse the tree starting from the root and check at each node if it has any lonely children. If a node has exactly one child (either left or right), that child is a lonely node, and its value is added to the answer list.

Here's the intuition behind the DFS approach for this problem:

- Start DFS from the root of the tree.
- Upon visiting each node, check if the node has only one child.
- If the node only has a left child (meaning the right child is `None`), record the value of the left child.
- If the node only has a right child (meaning the left child is `None`), record the value of the right child.
- Recur for both the left and right children of the current node.
- Once DFS is complete, the answer list will contain the values of all the lonely nodes.
- Return the answer list.

This method ensures that every node is visited, and no lonely nodes are missed. It uses the recursive nature of DFS to backtrack and traverse all paths within the tree efficiently.

## Solution Approach

The provided Python solution implements a recursive Depth-First Search (DFS) strategy on the binary tree, which is a common approach for solving tree traversal problems.

Here's a step-by-step explanation of how the code works:

1. A helper function named `dfs` is defined, which will be invoked on the root node and recursively on each child. The `root` parameter of this function refers to the current node being visited.

2. The base case of our recursive function checks for two conditions:

   - If the `root` is `None`, meaning we have reached a leaf node's child (which doesn't exist), in which case the function returns immediately without doing anything further.
   - If both the left and right children of the current node are `None`, meaning the current node is a leaf, there's no need to proceed further as leaf nodes cannot have lonely nodes.

3. For each non-null node visited by `dfs`, the function checks if the node has a single child. This is verified by checking if `root.left` is `None` when a right child exists, or `root.right` is `None` when a left child exists.

4. If the node has only one child, the value of that lonely child node (`root.right.val` or `root.left.val`) is added to the `ans` list.

5. After checking for loneliness, `dfs` is called recursively on both the left and right children of the current node, if they exist. This allows the function to traverse the whole tree thoroughly.

6. Before calling the `dfs` function, an empty list named `ans` is created. This will be used to collect the values of the lonely nodes that are encountered during the recursion.

7. The `dfs` function is called with the `root` of the binary tree as its argument, starting the traversal.

8. Once the full tree has been traversed, the `ans` list is complete, and it is returned as the final result.

The underlying concepts used in this solution include recursive functions, tree traversal, and DFS, which is a fundamental pattern for exploring all nodes of a tree systematically. This specific problem doesn't require maintaining any additional data structures aside from the `ans` list that accumulates the result. The simplicity and elegance of recursion make the solution concise and highly readable.

### Example Walkthrough

Let's consider a binary tree with the following structure:

```
     1
    / \
   2   3
  /   / \
 4   5   null
    /
   6
```

In this tree:

- Node 1 is the root and has two children (2 and 3), so it's not lonely.
- Node 2 has one child (4), but no right sibling, so it's lonely.
- Node 3 has one child missing (right child), so its left child (5) is lonely.
- Node 4 is the only child of node 2, and it's also lonely.
- Node 5 has a left child (6), but no right sibling, so node 5 is also lonely.

We want to find all the lonely nodes, which in this case are 2, 4, 5, and 6.

Now, following the solution approach:

1. We define `dfs(root)`, which we'll call on the root node (1 in our example).
2. We check if `root` is `None`. Since 1 has children, it's not `None`, we proceed. It's also not a leaf, so no base case conditions are met.
3. We check if node 1 has a single child. It has two, so no nodes are added to the `ans` list.
4. We call `dfs(2)` and `dfs(3)` recursively.

For node 2:

- It's not `None`, and it's not a leaf.
- Check if node 2 has a single child: Node 2 has only one child (4), so we add 4 to `ans`.
- We call `dfs(4)` recursively.

For node 4:

- It is `None` when checking for its children, so no further actions are taken (leaf node).

For node 3:

- It's not `None`, and it's not a leaf.
- Node 3 has one child missing (right child is `None`), so we add 5 (its left child) to `ans`.
- We call `dfs(5)`.

For node 5:

- It's not `None`, and it's not a leaf.
- Node 5 has no right sibling, so we add 6 (its left child) to `ans`.
- We call `dfs(6)`.

For node 6:

- All children are `None`. It's a leaf node, so no further action is taken.

5. After all recursions complete, we've added all the values of the lonely nodes - 4, 5, and 6 - to the `ans` list.

We would have just walked through the algorithm to correctly identify nodes 4, 5, and 6 as the lonely nodes in the given tree based on the DFS strategy laid out in the problem's solution approach.

## Python Solution

```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 get_lonely_nodes(self, root: Optional[TreeNode]) -> List[int]:
        """
        Perform a depth-first search to find all nodes that have only one child (lonely nodes)
        """

        # Helper function to perform DFS
        def dfs(node):
            # If the node is None or is a leaf node (no children), there's nothing to do
            if node is None or (node.left is None and node.right is None):
                return

            # If the node has a right child but no left child, add the right child's value
            if node.left is None:
                lonely_nodes.append(node.right.val)

            # If the node has a left child but no right child, add the left child's value
            if node.right is None:
                lonely_nodes.append(node.left.val)

            # Recursively apply DFS to the left and right children
            dfs(node.left)
            dfs(node.right)

        # Initialize an empty list to store lonely node values
        lonely_nodes = []
        # Trigger DFS from the root of the tree
        dfs(root)
        # Return the list of lonely node values
        return lonely_nodes

# Note: The 'Optional' type and 'List' need to be imported from 'typing' module.
# Example:
# from typing import List, Optional

```

## Java Solution

```java
// Class to define a binary tree node
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 to store the values of lonely nodes
    private List<Integer> lonelyNodes = new ArrayList<>();

    // Public method to find all lonely nodes
    public List<Integer> getLonelyNodes(TreeNode root) {
        // Start depth-first search traversal from root to find lonely nodes
        dfs(root);
        return lonelyNodes;
    }

    // Private helper method to perform depth-first search
    private void dfs(TreeNode node) {
        // Base case: if the node is null or it's a leaf (no children)
        if (node == null || (node.left == null && node.right == null)) {
            return;
        }
        // If the node has no left child, the right child is a lonely node
        if (node.left == null) {
            lonelyNodes.add(node.right.val);
        }
        // If the node has no right child, the left child is a lonely node
        if (node.right == null) {
            lonelyNodes.add(node.left.val);
        }
        // Recursively apply DFS to the left subtree
        dfs(node.left);
        // Recursively apply DFS to the right subtree
        dfs(node.right);
    }
}

```

## C++ Solution

```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 {
public:
    // Function to collect all lonely nodes in a binary tree
    vector<int> getLonelyNodes(TreeNode* root) {
        vector<int> lonelyNodes; // This will hold the lonely nodes, which are nodes that have only one child

        // Define the DFS function to traverse the tree
        function<void(TreeNode*)> dfs = [&](TreeNode* node) {
            // Base case: If the node is null or it is a leaf node (no children), return
            if (!node || (!node->left && !node->right)) return;

            // If the node only has a right child
            if (!node->left) {
                lonelyNodes.push_back(node->right->val); // Add the value of the right child to our answer
            }

            // If the node only has a left child
            if (!node->right) {
                lonelyNodes.push_back(node->left->val); // Add the value of the left child to our answer
            }

            // Recursively call the DFS on the left and right children
            dfs(node->left);
            dfs(node->right);
        };

        // Start DFS traversal from the root
        dfs(root);
        // Return the list of lonely nodes found
        return lonelyNodes;
    }
};

```

## Typescript Solution

```typescript
// Definition for a binary tree node
class TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;

    constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

// This array will hold the lonely nodes, which are nodes that have only one child
const lonelyNodes: number[] = [];

// The DFS function to traverse the tree
const dfs = (node: TreeNode | null): void => {
    // Base case: If the node is null or it is a leaf node (no children), return
    if (!node || (!node.left && !node.right)) return;

    // If the node only has a right child
    if (!node.left) {
        // Add the value of the right child to our answer
        lonelyNodes.push(node.right.val);
    }

    // If the node only has a left child
    if (!node.right) {
        // Add the value of the left child to our answer
        lonelyNodes.push(node.left.val);
    }

    // Recursively call the DFS on the left and right children
    dfs(node.left);
    dfs(node.right);
};

// Function to collect all lonely nodes in a binary tree
const getLonelyNodes = (root: TreeNode): number[] => {
    // Start DFS traversal from the root
    dfs(root);
    // Return the list of lonely nodes found
    return lonelyNodes;
};

```

## Time and Space Complexity

### Time Complexity

The time complexity of the code is `O(n)`, where `n` is the number of nodes in the given binary tree. This complexity arises because the algorithm needs to visit each node exactly once to determine if it has a lonely node (a node with only one child).

### Space Complexity

The space complexity of the code is `O(h)`, where `h` is the height of the binary tree. This is because the depth of the recursive call stack will go as deep as the height of the tree in the worst case. Note that this doesn't include the space taken by the output list `ans` which, in the worst case, can have up to `n - 1` elements if all nodes have only one child. If including the output, the space complexity would be `O(n)`.