993. Cousins in Binary Tree

Easy · Tree · Depth-First Search · Breadth-First Search · Binary Tree — Leetcode Link

Problem Description

The problem requires determining if two nodes within a binary tree are cousins. In binary tree terminology, cousins are defined as nodes that are on the same level (or depth) but do not share the same parent.

The inputs to the function are:

- A binary tree `root`, where each node contains a unique value.
- Two integer values `x` and `y` that correspond to the values of two nodes within the binary tree.

The output is:

- A boolean value `true` or `false` indicating whether the two nodes with values `x` and `y` are indeed cousins.

It is important to clarify that the depth of the root node is considered to be `0`, and each successive level in the tree increments the depth by `1`.

Intuition

The solution to this problem requires a way to traverse the tree and determine the depth and parent of each node we are interested in (`x` and `y`). The approach chosen here uses Depth-First Search (DFS) to explore the tree.

The DFS allows us to traverse down each branch of the tree until we hit a leaf, keeping track of the depth and parent node at each step. Each time we make a move to a child node, the depth increases by `1`.

During the DFS, whenever we come across either of the values `x` or `y`, we store the parent node and depth in a tuple. Since `x` and `y` are distinct and unique, when one is found, its corresponding information is stored in an array `t`, at index `0` for `x`, and index `1` for `y`.

Lastly, after the traversal, we compare the stored depths and parents of nodes `x` and `y`. To be cousins, the following conditions must both hold:

1. The depths of nodes `x` and `y` must be the same, which ensures they are at the same level of the tree.
2. The parents of nodes `x` and `y` must be different, which ensures they do not share the same parent.

If both conditions are satisfied, we return `true`, confirming that the nodes are cousins; otherwise, we return `false`.

Solution Approach

The provided solution uses Depth-First Search (DFS), a classic tree traversal algorithm that explores as far as possible down each branch before backtracking. DFS is well-suited for this problem because it allows us to track the depth and parent of each node as we traverse the tree.

Here's a step-by-step implementation of the DFS algorithm for this problem:

1. Create a helper function `dfs` that takes the current node being visited (`root`), its parent (`fa`), and the current depth (`d`) as arguments. The helper function will also need access to an array `t` that stores the parent and depth information for nodes `x` and `y`.

2. If the current node `root` is `None` (meaning we've reached a leaf or the tree is empty), the function simply returns as there is nothing further to explore.

3. The function checks if the current node's value is equal to `x` or `y`. If it is, the corresponding tuple of `(parent, depth)` is stored in the array `t`. Specifically, `t[0]` is used for `x` and `t[1]` for `y`. This is how the function keeps track of the required information for determining if the nodes are cousins.

4. Continue the DFS on the left and right children of the current node, increasing the depth by `1` and passing the current node as the new parent.

5. After initiating the DFS from the root node with a `None` parent and depth `0`, the solution checks whether the stored parents are different and the stored depths are the same for `x` and `y`. It uses the statement `return t[0][0] != t[1][0] and t[0][1] == t[1][1]` which essentially says, return `true` if the parents are not the same and the depths are the same; otherwise, return `false`.

This solution is efficient, as DFS ensures that each node is visited exactly once, resulting in a time complexity of O(n), where n is the number of nodes in the tree. No additional space is used besides the recursive stack and the array `t`, leading to a space complexity of O(n) due to the height of the recursive call stack.

By leveraging DFS and efficiently keeping track of the parent and depth of each node, this solution effectively determines whether two nodes are cousins in a binary tree.

Example Walkthrough

Let's consider a simple binary tree and follow the steps of the solution approach to determine if two nodes are cousins. Here's our example binary tree:

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

Let's say we want to check if node 4 (with value `x=4`) and node 5 (with value `y=5`) are cousins.

1. We start DFS traversal from the root node (value `1`). The `dfs` function is called with the arguments `root` (the current node), `fa` (the parent node), and `d` (the depth). So initially, `dfs(1, None, 0)` is called since `1` is the root with no parent and is at depth `0`.

2. The recursion now explores the left child of the root node > `dfs(2, 1, 1)`.

   - At node `2`, neither `x` nor `y` is found, so we proceed to its left child with a depth increased by 1 > `dfs(4, 2, 2)`.
3. At node `4`, we have found `x=4`. We store the parent and depth information in `t[0] = (2, 2)` and return to the previous call.

4. Since node `2` has no right child, the recursion ends here, and we go back to the root node to explore its right child > `dfs(3, 1, 1)`.

5. At node `3`, neither `x` nor `y` is found, so we proceed to its right child with increased depth > `dfs(5, 3, 2)`.

6. At node `5`, we have found `y=5`. We store the parent and depth information in `t[1] = (3, 2)` and return to the previous call.

7. All nodes have been visited and the recursion concludes.

After traversing the tree, we examine the values in `t`. We have:

- `t[0] = (2, 2)` for node `4`, and
- `t[1] = (3, 2)` for node `5`.

We compare the parents, `2` and `3`, and find they are different. We also compare the depths, both `2`, and find they are the same.

According to our two conditions for the nodes to be cousins:

- The depths are the same (true).
- The parents are different (true).

Therefore, the function returns `true` signifying that node `4` and node `5` are indeed cousins in the binary tree.

Python Solution

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def isCousins(self, root: TreeNode, x: int, y: int) -> bool:
        # Helper function to perform depth-first search (DFS)
        def dfs(node, parent, depth):
            if node is None:
                return
            if node.val == x:
                # Record the parent and depth for x
                found_nodes[0] = (parent, depth)
            if node.val == y:
                # Record the parent and depth for y
                found_nodes[1] = (parent, depth)

            # Recursion for left and right children
            dfs(node.left, node, depth + 1)
            dfs(node.right, node, depth + 1)

        # Initialize nodes as a list to hold the pair (parent, depth) for x and y
        found_nodes = [None, None]

        # Call DFS starting from the root, without a parent and at depth 0
        dfs(root, None, 0)

        # Check if x and y have different parents and the same depth
        return found_nodes[0][0] != found_nodes[1][0] and found_nodes[0][1] == found_nodes[1][1]

# Example of using the class:
# Create a binary tree with TreeNode instances, then
# solution = Solution()
# result = solution.isCousins(root, x, y)

```

Java Solution

```java
/**
 * Definition for 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 {
    private int targetValueX, targetValueY;
    private TreeNode parentX, parentY;
    private int depthX, depthY;

    /**
     * Determines if two nodes are cousins in a binary tree.
     * Nodes are considered cousins if they are on the same level of the tree,
     * but have different parents.
     *
     * @param root The root node of the binary tree.
     * @param x The value of the first node.
     * @param y The value of the second node.
     * @return true if the nodes with values x and y are cousins, false otherwise.
     */
    public boolean isCousins(TreeNode root, int x, int y) {
        this.targetValueX = x;
        this.targetValueY = y;
        // Start the depth-first search from the root, with null parent and depth 0
        dfs(root, null, 0);
        // Nodes are cousins if they have the same depth but different parents
        return parentX != parentY && depthX == depthY;
    }

    /**
     * Helper method to perform a depth-first search on the binary tree.
     *
     * @param node The current node to process.
     * @param parent The parent of the current node.
     * @param depth The current depth in the tree.
     */
    private void dfs(TreeNode node, TreeNode parent, int depth) {
        if (node == null) {
            return;
        }
        if (node.val == targetValueX) {
            parentX = parent;
            depthX = depth;
        }
        if (node.val == targetValueY) {
            parentY = parent;
            depthY = depth;
        }
        // Recursively process left and right subtrees, increasing the depth
        dfs(node.left, node, depth + 1);
        dfs(node.right, node, depth + 1);
    }
}

```

C++ Solution

```cpp
#include <functional> // Include the functional header for std::function

// 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:
    // Determines if two nodes of a binary tree are cousins (same depth, but different parents)
    bool isCousins(TreeNode* root, int x, int y) {
        TreeNode *parentX, *parentY; // Nodes to keep track of each target node's parent
        int depthX, depthY; // Variables to keep track of each target node's depth

        // Depth-first search lambda function to find parent and depth of target nodes
        std::function<void(TreeNode*, TreeNode*, int)> dfs = [&](TreeNode* node, TreeNode* parent, int depth) {
            if (!node) {
                return; // Base case: if the node is null, do nothing
            }
            if (node->val == x) {
                // If the current node value is x, store the parent and depth
                parentX = parent;
                depthX = depth;
            }
            if (node->val == y) {
                // If the current node value is y, store the parent and depth
                parentY = parent;
                depthY = depth;
            }
            // Recursive calls to search in the left and right subtrees, increasing depth by 1
            dfs(node->left, node, depth + 1);
            dfs(node->right, node, depth + 1);
        };

        // Initialize the search on the root with null parent and depth 0
        dfs(root, nullptr, 0);

        // Two nodes are cousins if they have different parents but the same depth
        return parentX != parentY && depthX == depthY;
    }
};

```

Typescript Solution

```typescript
// Binary tree node structure
interface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

// Function to create a new TreeNode given a value, left and right nodes
function createTreeNode(val: number, left?: TreeNode, right?: TreeNode): TreeNode {
  return {
    val: val,
    left: left || null,
    right: right || null
  };
}

// Variable to keep track of a target node's parent
let parentNodeX: TreeNode | null;
let parentNodeY: TreeNode | null;

// Variables to keep track of each target node's depth
let depthNodeX: number;
let depthNodeY: number;

// Depth-first search function to find parent and depth of target nodes
function depthFirstSearch(node: TreeNode | null, parent: TreeNode | null, depth: number) {
  if (!node) {
    return; // Base case: if the node is null, do nothing
  }
  if (node.val === x) {
    // If the current node value is x, store the parent and depth
    parentNodeX = parent;
    depthNodeX = depth;
  }
  if (node.val === y) {
    // If the current node value is y, store the parent and depth
    parentNodeY = parent;
    depthNodeY = depth;
  }

  // Recursive calls to search in the left and right subtrees, increasing depth by 1
  depthFirstSearch(node.left, node, depth + 1);
  depthFirstSearch(node.right, node, depth + 1);
}

// Determines if two nodes of a binary tree are cousins (same depth, but different parents)
function isCousins(root: TreeNode, x: number, y: number): boolean {
  // Initialize the search on the root with null parent and depth 0
  depthFirstSearch(root, null, 0);

  // Two nodes are cousins if they have different parents but the same depth
  return parentNodeX !== parentNodeY && depthNodeX === depthNodeY;
}

// Example usage:
// let root = createTreeNode(1, createTreeNode(2), createTreeNode(3));
// let result = isCousins(root, 2, 3); // Should be false since 2 and 3 are siblings, not cousins

```

Time and Space Complexity

Time Complexity

The given code traverses the binary tree to find the levels and parents of the nodes with values `x` and `y`. It uses a Depth First Search (DFS) approach that visits every node exactly once. Therefore, the time complexity of the code is `O(n)`, where `n` is the number of nodes in the binary tree. No matter what, the algorithm must visit every node to ensure that it finds the nodes `x` and `y`.

Space Complexity

The space complexity of the code is mainly determined by the maximum depth of the recursion stack, which depends on the height of the tree. In the worst-case scenario for a skewed tree, the height of the tree can be `O(n)`, leading to a space complexity of `O(n)`. However, in a balanced tree, the height would be `O(log n)`, resulting in a space complexity of `O(log n)`. The auxiliary space used to store the parent and depth (`t[0]` and `t[1]`) is constant, hence it doesn't significantly affect the space complexity. So, the overall space complexity is `O(n)` in the worst case for a skewed tree or `O(log n)` for a balanced tree.