863. All Nodes Distance K in Binary Tree

Medium | Tree | Depth-First Search | Breadth-First Search | Binary Tree | Leetcode Link

Problem Description

The problem presents a binary tree and asks us to find all nodes that are a certain distance **k** from a given target node. It's important to note that distance here refers to the number of edges we must traverse to reach one node from another. This problem doesn't ask to find nodes only in the subtree of the target node but from the entire tree, so we must consider not only the children of the target node but also its ancestors and siblings.

Intuition

To solve this problem, we need to find all nodes that are at the specified distance **k** from the target node, regardless of whether they're ancestors, descendants, or neither (siblings or cousins). The approach to finding this solution is divided into several parts:

1. **Mapping Parents:** We first need a way to traverse the tree not only downwards from the root to the leaves (as is normally the case with tree traversals) but also upwards, from any node to its parent. This is not usually possible in a binary tree because nodes don't have a reference to their parent. Therefore, we create a map that keeps track of each node's parent.
2. **Performing the Search:** Once we have the ability to move both up and down the tree, we perform a depth-first search (DFS) starting from the target node. As we explore the tree, we keep track of the current distance from the target. When this distance equals **k**, we add the current node's value to our answer.
3. **Avoiding Revisits:** To ensure we don't count any nodes twice or enter a loop, we keep a set of visited nodes. Whenever we visit a node, we add it to the set. If we encounter a node that's already in our set, we skip it.

With this approach, we can explore all possible paths - down to the children, up to the parent, and across to siblings and other relatives - to find all nodes that are exactly **k** distance away from the target.

Solution Approach

The solution is implemented using a depth-first search (DFS) algorithm along with additional data structures to track parent nodes and visited nodes. Here's a step-by-step breakdown of how the code accomplishes the task:

1. **Parent Mapping**: The first function, `parents`, is a recursive function that populates a dictionary `p` where keys are nodes of the tree and values are their respective parent nodes. It goes through all the nodes in a pre-order traversal (parent, then left child, then right child) and assigns the current node's parent to it in the dictionary.
2. **Depth-First Search (DFS)**: The second function, `dfs`, is the crucial part of the implementation where we perform a depth-first search, starting from the target node. The function takes a node and the distance `k` as arguments.
3. **Visited Set**: Inside the DFS function, we first ensure that the current node is not `None` and hasn't been visited yet, as indicated by checking the `vis` set. If we have seen the node already, we return early to avoid revisiting it.
4. **Base Case for Distance**: If the distance `k` is 0, it means we have found a node at the exact distance from the target we are looking for, so we append the value of the current node to the `ans` list, which will eventually be our output.
5. **Recursive Case for Traversal**: If `k` is not 0, we recursively call DFS on the left child, the right child, and the parent of the current node, each time with `k - 1` to account for the decrease in distance as we move one step away from the target.
6. **Execution**: Finally, we initialize our data structures - `p` for parents, `ans` for answers, and `vis` for visited nodes. We then populate `p` using the `parents` function and execute `dfs` starting with the target node and the initial distance `k`.
7. **Result**: After DFS completes the traversal of the tree, `ans` will contain all the node values that are `k` distance away from the target node. At the end of the function, we simply return `ans`.

In summary, the algorithm traverses the entire binary tree to build a parent-child relationship map, then it uses DFS to search all possible paths from the target node to find nodes that are exactly `k` steps away while avoiding loops and repetitions.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach described above. Suppose we have the following binary tree and we want to find all the nodes that are a distance `k = 2` away from the target node with value `5`.

```
    1
   / \
  2   3
 / \ / \
4  5 6  7
```

Step 1: Parent Mapping First, we construct a map of parent pointers:

- The parent of node 2 is 1.
- The parent of node 3 is 1.
- The parent of node 4 is 2.
- The parent of node 5 is 2, and so on.

Step 2: Performing the Search We then perform a DFS starting from the target node 5. We are looking for nodes that are at a distance `k = 2`.

Step 3: Avoiding Revisits We start with an empty `vis` set to keep track of visited nodes.

Starting from node 5, the search goes as follows:

- Visit node 5 (distance `k = 2`), mark it as visited.
- Move to node 8 (distance `k = 1`), mark it as visited.
- Move to node 4 (distance `k = 0`), since the distance is 0 and node 4 hasn't been visited, add it to the answer.
- Similarly, move to node 9 (distance `k = 1`), mark it as visited.
- Since node 9 has no children, return to node 5 and move up to its parent node 2 (distance `k = 1`), mark it as visited.
- Since node 2 is visited for the first time with distance `k = 1`, move to node 4 (already visited) and node 3 (distance `k = 0` via parent 1). Again, as the distance is 0, add node 3 to the answer.

Step 4: Summary of the Results At the end of the DFS, we find that the nodes at distance 2 from the target node 5 are nodes 3 and 4. Thus, the answer is `[3, 4]`.

By following the steps in the provided solution approach, we have successfully found all nodes at a specific distance from a given target in a binary tree.

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 distanceK(self, root: TreeNode, target: TreeNode, k: int) -> List[int]:
        # Helper function to store parent pointers for each node
        def map_parents(node, parent):
            if node:
                parent_map[node] = parent
                map_parents(node.left, node)
                map_parents(node.right, node)

        # Helper function to perform Depth-First Search (DFS) and find nodes at distance k
        def find_nodes_at_distance_k(node, remaining_distance):
            if not node or node.val in visited:
                return
            visited.add(node.val)
            if remaining_distance == 0:
                result.append(node.val)
            else:
                # Visit children and parent
                find_nodes_at_distance_k(node.left, remaining_distance - 1)
                find_nodes_at_distance_k(node.right, remaining_distance - 1)
                find_nodes_at_distance_k(parent_map[node], remaining_distance - 1)

        # Initialize the parent map and result list
        parent_map = {}
        result = []
        visited = set()

        # Map parents for all nodes in the tree
        map_parents(root, None)

        # Start DFS from the target node
        find_nodes_at_distance_k(target, k)

        return result
```

Java Solution

```java
import java.util.*;

// Definition for a binary tree node.
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

class Solution {
    private Map<TreeNode, TreeNode> parentMap;
    private Set<Integer> visited;
    private List<Integer> answer;

    // This method finds all nodes at distance K from the target node.
    public List<Integer> distanceK(TreeNode root, TreeNode target, int K) {
        parentMap = new HashMap<>(); // Used to store parent-child relationships.
        visited = new HashSet<>(); // Tracks visited nodes to prevent cycles.
        answer = new ArrayList<>(); // Stores the results.

        // Associates each node with its parent.
        mapParents(root, null);

        // Performs DFS to find nodes at distance K.
        findNodesKDistanceK(target, K);

        return answer;
    }

    // Maps each node to its parent recursively.
    private void mapParents(TreeNode node, TreeNode parent) {
        if (node == null) {
            return;
        }

        parentMap.put(node, parent);
        mapParents(node.left, node); // Parent for left child is 'node'
        mapParents(node.right, node); // Parent for right child is 'node'
    }

    // DFS to find and add all nodes that are K distance from the given node.
    private void findNodesKDistanceK(TreeNode node, int k) {
        if (node == null || visited.contains(node.val)) {
            // Base case: if the node is null or already visited, stop the search.
            return;
        }

        visited.add(node.val); // Mark this node as visited.

        if (k == 0) {
            // If the distance is zero, the current node is K distance from the target.
            answer.add(node.val);
            return; // Node added to the answer, and this path here.
        }

        // Continue searching in the left subtree.
        findNodesKDistanceK(node.left, k - 1);

        // Continue searching in the right subtree.
        findNodesKDistanceK(node.right, k - 1);

        // Also search the sub-tree above using the parent references.
        findNodesKDistanceK(parentMap.get(node), k - 1);
    }
}
```

C++ Solution

```cpp
/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Maps each node to its parent
    unordered_map<TreeNode*, TreeNode*> parentMap;

    // Keeps track of visited nodes
    unordered_set<int> visited;

    // Stores the results
    vector<int> result;

    // Function that returns all nodes at distance K from the target node
    vector<int> distanceK(TreeNode* root, TreeNode* target, int K) {
        // Populate the parent pointers for all nodes
        storeParents(root, nullptr);

        // Perform search to find all nodes at distance K
        depthFirstSearch(target, K);

        return result;
    }

    // Helper method to relate each node to its parent in the tree
    void storeParents(TreeNode* node, TreeNode* parent) {
        if (!node) return;

        parentMap[node] = parent;

        storeParents(node->left, node);
        storeParents(node->right, node);
    }

    // The DFS method that traverses the tree to find nodes at a distance k
    void depthFirstSearch(TreeNode* node, int k) {
        // If the node is null or already visited, stop the search
        if (!node || visited.count(node->val)) return;

        // Mark the current node as visited
        visited.insert(node->val);

        // If the desired distance is reached, add the node's value to the results
        if (k == 0) {
            result.push_back(node->val);
            return;
        }

        // Continue the search in left and right subtree
        depthFirstSearch(node->left, k - 1);
        depthFirstSearch(node->right, k - 1);

        // Also consider the parent node in the search
        depthFirstSearch(parentMap[node], k - 1);
    }
};
```

Typescript Solution

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

  constructor(val: number) {
    this.val = val;
    this.left = null;
    this.right = null;
  }
}

// Maps each node to its parent
const parentMap: Map<TreeNode, TreeNode | null> = new Map();

// Keeps track of visited nodes
const visited: Set<number> = new Set();

// Stores the results
const result: number[] = [];

// Function that returns all nodes at distance K from the target node
function distanceK(root: TreeNode | null, target: TreeNode | null, k: number): number[] {
  // Populate the parent pointers for all nodes
  storeParents(root, null);

  // Perform search to find all nodes at distance k
  depthFirstSearch(target, k);

  return result;
}

// Helper function to relate each node to its parent in the tree
function storeParents(node: TreeNode | null, parent: TreeNode | null) {
  if (!node) return;

  parentMap.set(node, parent);

  storeParents(node.left, node);
  storeParents(node.right, node);
}

// The DFS method that traverses the tree to find nodes at a distance k
function depthFirstSearch(node: TreeNode | null, k: number) {
  // If the node is null or already visited, stop the search
  if (!node || visited.has(node.val)) return;

  // Mark the current node as visited
  visited.add(node.val);

  // If the desired distance is reached, add the node's value to the results
  if (k === 0) {
    result.push(node.val);
    return;
  }

  // Continue the search in the left and right subtree
  depthFirstSearch(node.left, k - 1);
  depthFirstSearch(node.right, k - 1);

  // Also consider the parent node in the search
  const parentNode = parentMap.get(node) || null;
  depthFirstSearch(parentNode, k - 1);
}
```

Time and Space Complexity

The provided Python code defines a function `distanceK` that finds all nodes at distance `k` from a target node in a binary tree.

Time Complexity:

To analyze the time complexity, let's break down the procedure into its main steps:

1. The function `parents` traverses the entire tree to build a dictionary `p` mapping each node to its parent. This is a depth-first search (DFS) with a single visit to each node, hence its complexity is $O(N)$, where `N` is the number of nodes in the tree.
2. The function `dfs` is a recursive depth-first search. In the worst case, it can visit each node once when `k` equals the height of the tree. Additionally, it might visit each node's parent, resulting in $O(2N)$ operations, which simplifies to $O(N)$.

Given these steps, the overall time complexity is $O(N)$ since both `parents` and `dfs` functions operate linearly with respect to the number of nodes in the tree.

Space Complexity:

Now, let's consider the space complexity:

1. The `p` dictionary holds a pair for each node in the tree, hence $O(N)$ space.
2. The `ans` list could hold up to `N` nodes (in the case that the tree is a straight line and `k` equals `N-1`), so its space is $O(N)$.
3. The call stack for the DFS function can go up to the height of the tree in the case of a skewed tree, which in the worst case is $O(N)$.
4. The `vis` set contains nodes that are already visited, which in the worst case can be $O(N)$ as well when every node is visited.

The space complexity is determined by adding the space required for the `p` dictionary, the `ans` list, the call stack, and the `vis` set, which all contribute linearly to $O(N)$ space.

Overall, taking into account both the worst-case scenarios for the recursive calls stack and the space needed for auxiliary data structures, the space complexity is $O(N)$.