1602. Find Nearest Right Node in Binary Tree

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

Problem Description

Given a binary tree (a tree where each node has at most two children), we need to find the nearest node to the right of a given node `u` on the same level. If there is no node to the right of `u` on the same level (meaning `u` is the rightmost node), we should return `null`. The problem requires us to consider the level of the nodes in the tree, which is the depth at which they are found, with the root node being at level one.

To understand the problem, let's consider an example:

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

If `u` is node 2, the nearest right node on the same level is 3. If `u` is node 4, the nearest right node is 5. If `u` is node 5, since there are no further nodes to its right, the answer would be `null`.

To get the solution, we need an approach that allows us to identify the nodes at each level and their order from left to right.

Intuition

The intuitive approach to solving this problem is to use Breadth-First Search (BFS). BFS is a graph traversal method that explores nodes level by level. For a binary tree, BFS starts at the root node, then explores all the children nodes, and so on down the levels of the tree. It is often implemented with a queue data structure.

In this context, BFS allows us to traverse the tree level by level, which aligns perfectly with the requirement to find nodes on the same level. By keeping track of nodes as we traverse each level, we can identify each node's immediate right neighbor on the same level.

The implementation of the solution initializes a queue with the root node. Then, it enters a loop that continues as long as there are nodes left to visit. Within this loop, we iterate over the number of elements that the queue has at the beginning of each iteration, which corresponds to all the nodes at the current level of the tree. We dequeue each of these nodes, checking if it matches the node `u`. If we find the node `u`, we return the next node in the queue, which will be `u`'s nearest right neighbor on the same level. If `u` happens to be the last node on its level, the queue would be empty after it, so we return `null`. If the current node has left or right children, these children are added to the queue to explore the next level on subsequent iterations.

By the end of the BFS process, we will have either found the nearest right neighbor of `u` or determined that no such node exists, returning the correct result in either case.

Solution Approach

The given solution uses Breadth-First Search (BFS), a classic algorithm for traversing or searching tree or graph data structures. Here's how it's applied in this case:

1. **Initialization**: First, a queue named `q` is initialized with the root node of the tree. This queue will be used to keep track of the nodes to visit next, starting from the root. In BFS, the queue is crucial for managing the order of exploration.
2. **Traversal**: Then we enter a while loop which continues as long as there are nodes in the queue.
3. **Level-wise Iteration**: Inside the while loop, we have a for loop that runs as many times as there are elements in the queue at the start of the while loop iteration. This is to ensure that we only process nodes that are at the same level in each iteration.
   - For each iteration, a node is popped from the left of the queue using `q.popleft()`. This operation ensures that we are visiting nodes from left to right at the current level—just like reading a text.
4. **Checking the Target Node**:
   - Within this for loop, we check if the popped node is the node `u` that we are trying to find the nearest right node for.
   - If the current node is `u`, we then return the next right neighbor if it exists. This is done by checking whether the for loop index `i` is zero. If `i` is not zero, then it means there's another node at this level to the right of `u`, so we return `q[0]`, the next node. If `i` is zero, `u` is the last node on its level, so we return `null`.
5. **Adding Child Nodes**:
   - If the current node is not `u`, we add its child nodes to the queue if they exist. This prepares the next level for exploration. First, the left child is added using `q.append(root.left)` and then the right child using `q.append(root.right)`.
6. **Loop Continuation**: The for loop ensures that all nodes on the same level are processed, after which the while loop moves on to the next level.
7. **Completing the Search**: If we never find the target node `u` (which shouldn't happen given the problem constraints), or there's no right node, the while loop will eventually end when there are no more nodes to process in the queue.

By implementing BFS, we ensure that we visit all nodes at each level from left to right, which is essential for finding the nearest right node to a given node `u` on the same level.

Example Walkthrough

Consider a binary tree as follows:

```
     A
    / \
   B   C
  /   / \
 D   E   F
```

We want to find the nearest right node of node `B`.

Following the solution approach:

1. **Initialization**: We initialize a queue `q` and enqueue the root node `A`.
2. **Traversal**: We enter the while loop since our queue is not empty.
3. **Level-wise Iteration**: We have one element (the root `A`) in our queue, so the for loop inside the while loop will run once.
   - We dequeue node `A` from `q` and since `A` is not our target node `B`, we move forward.
4. **Adding Child Nodes**:
   - We add `A`'s child nodes `B` and `C` to the queue. Now our queue looks like: `[B, C]`.
5. **Loop Continuation**: The for loop ends, and we go back to the start of the while loop. Now, the queue has two elements at this level, `B` and `C`.
6. **Level-wise Iteration**: The for loop inside the while loop will now run twice since there are two elements.
   - We dequeue node `B`. This is our target node.
7. **Checking the Target Node**:
   - Since `B` is the target, we check if there's a right neighbor. Since we're not at the end of the level (the for loop has not finished all its iterations), we peek and see that the next element in the queue is node `C`.
   - We return `C` as the nearest right node to `B`.

If our target was node `C`, following similar steps, we would dequeue `C`, and because there would be no next element in the queue at this level, we would return `null`, indicating there is no right neighbor at the same level.

This walkthrough illustrates how Breadth-First Search can be used to navigate a binary tree level by level to return the nearest right node on the same level as the target node. If the target node is rightmost on its level, we correctly return `null`.

Python Solution

```python
from collections import deque
from typing import Optional

# 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 findNearestRightNode(self, root: TreeNode, target_node: TreeNode) -> Optional[TreeNode]:
        # Initialize the queue with the root node
        queue = deque([root])

        # Perform level order traversal using the queue
        while queue:
            # Iterate through the current level
            for i in range(len(queue) - 1, -1, -1):
                current_node = queue.popleft()

                # Check if the currentNode is the targetNode
                if current_node == target_node:
                    # Return the next node in the queue if it's not the last node in the level
                    return queue[0] if i else None

                # Enqueue the child nodes of the current node
                if current_node.left:
                    queue.append(current_node.left)
                if current_node.right:
                    queue.append(current_node.right)
        # If no right node is found, return None
        return None
```

Java Solution

```java
class Solution {

    /**
     * Finds the nearest right node to node u in the same level.
     *
     * @param root The root of the binary tree.
     * @param u    The target node to find its nearest right node.
     * @return     The nearest right neighbor in the same level as u,
     *             or null if u is the rightmost node.
     */
    public TreeNode findNearestRightNode(TreeNode root, TreeNode u) {
        // Create a queue to perform level order traversal of the tree.
        Deque<TreeNode> queue = new ArrayDeque<>();

        // Start from the root of the tree.
        queue.offer(root);

        // Perform level order traversal.
        while (!queue.isEmpty()) {
            // Process each level of the tree.
            for (int i = queue.size(); i > 0; --i) {
                // Dequeue node from the queue.
                TreeNode currentNode = queue.pollFirst();

                // Check if the current node is the target node u.
                if (currentNode == u) {
                    // If not the last node in its level, return the next node, otherwise return null.
                    return i > 1 ? queue.peekFirst() : null;
                }

                // If current node has a left child, enqueue it.
                if (currentNode.left != null) {
                    queue.offer(currentNode.left);
                }

                // If current node has a right child, enqueue it.
                if (currentNode.right != null) {
                    queue.offer(currentNode.right);
                }
            }
        }

        // If the target node u is not found or does not have a right neighbor, return null.
        return null;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Finds the nearest node to the right of the given node 'u' at the same level in a binary tree
    TreeNode* findNearestRightNode(TreeNode* root, TreeNode* u) {
        // Initialize a queue to conduct a level-order traversal
        queue<TreeNode*> nodeQueue;
        nodeQueue.push(root);

        // Execute the level-order traversal until the queue is empty
        while (!nodeQueue.empty()) {
            // Determine the number of nodes at the current level
            int levelSize = nodeQueue.size();

            // Iterate through all nodes at the current level
            for (int i = 0; i < levelSize; ++i) {
                // Access the front node in the queue
                TreeNode* currentNode = nodeQueue.front();
                nodeQueue.pop();

                // Check if the current node is the target node 'u'
                if (currentNode == u) {
                    // If the target node is not at the end of the level, return the next node
                    // Otherwise, return nullptr because there is no node to the right at the same level
                    return i < levelSize - 1 ? nodeQueue.front() : nullptr;
                }

                // Enqueue the left child if it exists
                if (currentNode->left) {
                    nodeQueue.push(currentNode->left);
                }

                // Enqueue the right child if it exists
                if (currentNode->right) {
                    nodeQueue.push(currentNode->right);
                }
            }
        }

        // If the target node 'u' is not found or does not have a right neighbor, return nullptr
        return nullptr;
    }
};
```

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;
    }
}

/**
 * Find the nearest right node to the given node `u` in the same level of a binary tree.
 * @param {TreeNode} root - The root of the binary tree.
 * @param {TreeNode} u - The target node to find the nearest right node for.
 * @return {TreeNode | null} - The nearest right node or null if there's no such node.
 */
var findNearestRightNode = function (root: TreeNode, u: TreeNode): TreeNode | null {
    // Queue to implement level-order traversal
    const queue: TreeNode[] = [root];

    // Execute a level-order traversal using a queue
    while (queue.length) {
        // Process all nodes on the current level
        for (let i = queue.length; i > 0; --i) {
            // Retrieve the front node from the queue
            let currentNode: TreeNode = queue.shift()!;

            // If the current node is the target node `u`
            if (currentNode === u) {
                // If this is not the last node of the level, return the next node
                // Otherwise, return null
                return i > 1 ? queue[0] : null;
            }

            // If the current node has a left child, enqueue it
            if (currentNode.left) {
                queue.push(currentNode.left);
            }

            // If the current node has a right child, enqueue it
            if (currentNode.right) {
                queue.push(currentNode.right);
            }
        }
    }

    // If the target node `u` was not found, return null
    return null;
};
```

Time and Space Complexity

The given code performs a level order traversal on a binary tree to find the nearest right node of a given node `u`.

Time Complexity

The time complexity of the code is `O(N)` where `N` is the number of nodes in the binary tree. This is because every node in the tree is visited exactly once during the level order traversal.

Space Complexity

The space complexity of the code is also `O(N)`. In the worst-case scenario (when the tree is a perfect binary tree), the maximum number of nodes at the last level of the tree will be around `N/2`. Hence, the queue `q` can potentially hold up to `N/2` nodes, which simplifies to `O(N)` when represented in Big O notation.