958. Check Completeness of a Binary Tree

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

Problem Description

In this problem, we are provided with the root node of a binary tree. Our objective is to determine whether the binary tree is a "complete binary tree" or not. A complete binary tree has a few defining characteristics:

- All levels are completely filled, except possibly the last level.
- On the last level, if it's not completely filled, all the nodes are as far left as possible.

This definition means that in a complete binary tree, if we traverse the nodes level by level from left to right, we will encounter all the non-null nodes first before any null node. If any null nodes are found, there should not be any more non-null nodes after them in this level order traversal.

To sum it up, the task is to check if the given binary tree meets the complete binary tree criteria.

Intuition

To determine if a binary tree is complete, we can perform a level order traversal on the tree. In an ordinary level order traversal (also known as breadth-first search), we enqueue all children of each node and process each node from the left to the right of each level.

The solution is based on an observation that in a complete binary tree, once we encounter the first null node (representing a missing child in the last level or an indication of no more nodes on the last level), all subsequent nodes encountered during the level order traversal must also be null. If we find a non-null node after encountering a null, the tree cannot be considered complete.

Here's the step-by-step reasoning for arriving at the solution:

1. Initiate a queue to perform level order traversal. Start by enqueuing the root node.
2. Begin the level order traversal, dequeuing a node from the front of the queue at each step.
3. Check the current node:
   - If it's not null, enqueue its left and right children (regardless of whether they're null or not) for later processing.
   - If the node is null, this indicates the end of all the non-null nodes encountered so far for the complete binary tree. Any subsequent nodes should be null to ensure the tree is complete.
4. After encountering a null node, check the rest of the queue:
   - If all remaining nodes in the queue are null, the tree is complete.
   - If any non-null node is found, then it's not a complete binary tree.

The algorithm will employ a deque (short for double-ended queue) as it allows efficient insertions and deletions from both ends, which is perfect for our level order traversal needs.

Solution Approach

The solution approach implements a level order traversal of the binary tree using a queue data structure. The standard queue is replaced with a `deque` from Python's collections module for efficient manipulation of the queue's endpoints.

Here's a step-by-step overview of the implementation:

1. Initialize a `deque` and add the root node of the binary tree to it.

```
q = deque([root])
```

2. Begin iterating through the queue:
   - For each iteration, dequeue (`popleft`) an element from the front of the queue.
   - This element represents the current node being processed.

```
while q:
    node = q.popleft()
```

3. The key condition: If the current node is `None`, it signals that we've encountered a gap in our level order traversal. Break out of the loop as we should not find any more non-null nodes after this point.

```
if node is None:
    break
```

4. If the current node is not `None`, enqueue its children (left and right) to the back of the queue. These operations will include `None` values if the current node doesn't have a corresponding child.

```
q.append(node.left)
q.append(node.right)
```

5. After breaking out of the loop, we must verify that all the remaining nodes in the queue are `None`. This would confirm that we did not encounter any non-null nodes after the first `None` node, which is a requirement for a complete binary tree.

```
return all(node is None for node in q)
```

This method relies heavily on the properties of a complete binary tree observed during a level order traversal. By utilizing these properties, the algorithm effectively determines the completeness of the binary tree with an efficient traversal and a simple while loop.

The pattern used in this method is a common Breadth-First Search (BFS) pattern, adapted to check the continuity of node presence at each level of the tree. The presence of a `None` value in the queue is used as an indicator to signal potential discontinuity, which would imply the binary tree isn't complete. After the first `None` is seen, the result of the `all` function confirms whether the rest of the elements left in the queue adhere to the complete binary tree property.

Implementing this approach provides an intuitive and direct method to verify that a binary tree is a complete binary tree. The use of a queue to keep track of nodes for processing and the break condition for encountering `None` values combine to form an elegant solution for this problem.

Example Walkthrough

Let's assume we have a binary tree structured like this:

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

This binary tree is a complete binary tree because all levels except possibly the last one are fully filled, and all nodes in the last level are as far left as possible.

Now let's walk through the solution approach with this binary tree as an example:

1. We initialize a `deque` and add the root node of the binary tree (the node with value 1) to it.

```
q = deque([1])
```

2. We begin iterating through the queue.
   - We dequeue an element from the front of the queue (`popleft()`), which is the root node with value 1.
   - Since it is not `None`, we enqueue its children nodes with values 2 and 3.

```
while q:
    node = q.popleft()  # node with value 1
    q.append(node.left)  # node with value 2
    q.append(node.right) # node with value 3
```

3. Since node with value 1 is not `None`, we continue with the loop:
   - We again `popleft()` on the deque, this time getting the node with value 2.
   - We enqueue its children nodes with values 4 and 5.

```
node = q.popleft()  # node with value 2
q.append(node.left)  # node with value 4
q.append(node.right) # node with value 5
```

4. We repeat the above step for node with value 3, enqueuing its child with value 6. Since node 3 has no right child, we insert `None`.

```
node = q.popleft()  # node with value 3
q.append(node.left)  # node with value 6
q.append(node.right) # None
```

5. Next, we process nodes 4, 5, and 6. Since they are at the last level and have no children, they would add `None` values to the queue.
   - Eventually, we encounter the first `None` value after all nodes on the last level.

```
node = q.popleft()  # node with value 4, adds None values to the queue
node = q.popleft()  # node with value 5, adds None values to the queue
node = q.popleft()  # node with value 6, adds None values to the queue
node = q.popleft()  # None, indicates end of non-null nodes
```

6. Since we encountered a `None`, we break out of the loop.
7. Now we check that the rest of the queue only contains `None` elements. If it does, we confirm that we have a complete binary tree. In this case, all remaining elements are `None`, so the tree is complete.

```
return all(node is None for node in q)  # returns True, confirming the tree is complete
```

By following this level order traversal and checking for the first occurrence of `None`, we have verified the binary tree is complete using our example.

Python Solution

```python
from collections import deque  # Import deque, as it's used for the queue.

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

class Solution:
    def isCompleteTree(self, root: TreeNode) -> bool:
        # Initialize a queue with the root node to check level by level.
        nodes_queue = deque([root])

        # Iterate over the nodes until you find the first None, which signals incomplete level.
        while nodes_queue:
            current_node = nodes_queue.popleft()

            # When a None is encountered, the check for completeness starts.
            if current_node is None:
                break

            # Append left and right children to the queue for further level checking.
            nodes_queue.append(current_node.left)
            nodes_queue.append(current_node.right)

        # At this point, all remaining nodes in the queue should be None for a complete tree.
        # This loop checks if there is any node that is not None after the first None was found.
        for node in nodes_queue:
            if node is not None:
                return False  # The tree is not complete as there is a node after a None.

        return True  # The tree is complete as all nodes after the first None are None.

# Example usage:
# tree = TreeNode(1, TreeNode(2), TreeNode(3))
# solution = Solution()
# print(solution.isCompleteTree(tree))  # Output: True
```

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 {
    /**
     * Determine if a binary tree is complete.
     * A binary tree is complete if all levels are completely filled except possibly the last,
     * which is filled from left to right.
     *
     * @param root The root node of the binary tree.
     * @return true if the tree is complete, false otherwise.
     */
    public boolean isCompleteTree(TreeNode root) {
        // Queue to hold tree nodes for level order traversal
        Deque<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

        // Traverse the tree using breadth-first search.
        // Keep adding children nodes until a null is encountered.
        while (queue.peek() != null) {
            TreeNode currentNode = queue.poll();
            queue.offer(currentNode.left);
            queue.offer(currentNode.right);
        }

        // Remove any trailing nulls from the queue.
        // If any non-null nodes are found afterwards it means that the tree is not complete.
        while (!queue.isEmpty() && queue.peek() == null) {
            queue.poll();
        }

        // If the queue is empty, then all the levels of the tree are fully filled.
        // Otherwise, the tree is not complete.
        return queue.isEmpty();
    }
}
```

C++ Solution

```cpp
#include <queue>

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    // Constructor initializes the node with a value, and the left and right pointers as null.
    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:
    // Method to check if a binary tree is complete.
    bool isCompleteTree(TreeNode* root) {
        // We use a queue to do a level order traversal of the tree.
        std::queue<TreeNode*> nodeQueue;
        nodeQueue.push(root);

        // Continue level order traversal until we find a null pointer.
        while (nodeQueue.front() != nullptr) {
            TreeNode* currentNode = nodeQueue.front();
            nodeQueue.pop(); // Remove front of the queue after saving the node.

            // Add child nodes to the queue for further processing.
            nodeQueue.push(currentNode->left);
            nodeQueue.push(currentNode->right);
        }

        // After finding the first null pointer, all subsequent nodes must also be null.
        // Pop all null pointers from the queue.
        while (!nodeQueue.empty() && nodeQueue.front() == nullptr) {
            nodeQueue.pop();
        }

        // If the queue is empty, it means there were no non-null nodes after the first null
        // which indicates the tree is complete. Otherwise, the tree is not complete.
        return nodeQueue.empty();
    }
};
```

Typescript Solution

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

    // Constructor initializes the node with a value, and the left and right pointers as null.
    constructor(val: number, left?: TreeNode | null, right?: TreeNode | null) {
        this.val = val;
        this.left = left === undefined ? null : left;
        this.right = right === undefined ? null : right;
    }
}

// Method to check if a binary tree is complete.
function isCompleteTree(root: TreeNode | null): boolean {
    if (!root) {
        // If the root is null, the tree is trivially complete.
        return true;
    }

    // We use a queue to do a level order traversal of the tree.
    const nodeQueue: (TreeNode | null)[] = [root];

    let foundNull = false;

    // Continue level order traversal until the queue is empty.
    while (nodeQueue.length > 0) {
        const currentNode = nodeQueue.shift();

        if (!currentNode) {
            // We encountered a null node; from this point forward, all nodes must be null.
            foundNull = true;
        } else {
            if (foundNull) {
                // If we previously encountered a null node and now we see a non-null node,
                // then the tree is not complete.
                return false;
            }
            // Add child nodes to the queue for further processing.
            nodeQueue.push(currentNode.left);
            nodeQueue.push(currentNode.right);
        }
    }

    // If we finished processing all nodes without finding a non-null
    // node after a null node, the tree is complete.
    return true;
}
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the number of nodes in the binary tree. This is because each node is processed exactly once when it is dequeued for examination.

The space complexity of the code is also `O(n)`. In the worst-case scenario, the queue could contain all the nodes at the last level of a complete binary tree, just before the loop terminates upon encountering a `None` node. This means that the number of elements in the queue could approach the number of leaves in the full binary tree, which is approximately `n/2`. Considering the constant factor is omitted in Big O notation, the space complexity remains `O(n)`.