94. Binary Tree Inorder Traversal

Easy Stack Tree Depth-First Search Binary Tree

Problem Description

The problem asks us to perform an inorder traversal on a binary tree and return the sequence of values from the nodes. In binary tree traversal, there are three types of depth-first searches - inorder, preorder, and postorder. Specifically, the inorder traversal follows a defined sequence to visit the nodes:

- Visit the left subtree
- Visit the root node
- Visit the right subtree

This pattern is recursive and is applied to each subtree within the tree. The result of performing an inorder traversal is that the nodes of the tree are visited in ascending order if the binary tree is a binary search tree. For this problem, we are required to collect the values of the nodes in the order they are visited and return them as a list.

Intuition

The proposed solution uses the Morris Traversal approach, which is an optimized way to do tree traversal without recursion and without using extra space for a stack. The basic idea of Morris Traversal is to link the rightmost node of a node's left subtree back to the node itself, which helps us to get back to the root node after we are done traversing the left subtree.

Here's the process how we arrive at the Morris Traversal solution approach:

- Start with the `root` node.
- If the `root` has no left child, it means this node can be visited now, so we add its value to the result list and move to its right child.
- If the `root` has a left child, find the rightmost node in the left subtree (the inorder predecessor of `root`).
  - If the rightmost node has no right child (is not already linked back to the `root`), we create a temporary link from it to the `root` and move the `root` to its left child.
  - If the rightmost node's right child is the `root` (already linked back), it means we have visited the left subtree already, so we add the `root`'s value to the result list, unlink (restore the tree structure), and move to the right child of the `root`.

This approach allows us to use the tree structure itself as a way to navigate through the tree without additional memory usage for the call stack or an auxiliary stack, thus giving us the inorder sequence of node values in O(n) time with O(1) space complexity.

Solution Approach

The solution provided implements the Morris Traversal as the algorithm for inorder traversal of the binary tree. Here is a walkthrough of the implementation:

- The function `inorderTraversal` begins with an empty list `ans`, which will contain the sequence of node values in inorder.

- The main loop runs as long as there is a `root` to process. The steps in the loop correspond to the ideas discussed in the intuition:

  - If `root.left` is `None`, it implies there's no left subtree, and the `root` node can be visited. So, `root.val` is added to the `ans` list and `root` is updated to be `root.right`, moving on to the next node in the inorder sequence.

  - If `root.left` exists, this means we have a left subtree that needs to be processed first. We then find the inorder predecessor of the `root` by traversing rightwards (`prev = prev.right`) until we find a node that either has no right child or whose right child is the current `root`. This predecessor will act as a temporary bridge back to the `root` after we've finished with the left subtree.

    - If the predecessor's (`prev`) right child is `None`, this means we haven't processed the left subtree yet. We, therefore, make a temporary link from the predecessor's right child to the current `root` (`prev.right = root`). This allows us to come back to the `root` after we're done with the left subtree. Then we move `root` to its left child and continue the loop.

    - If the predecessor's right child is the current `root`, this indicates we've returned from traversing the left subtree, and it's now time to visit the `root`. We, therefore, add `root.val` to the `ans` list, remove the temporary link to restore the tree's structure (`prev.right = None`), and proceed with `root.right`.

The loop continues until every node has been visited in the inorder sequence. Since we're altering the tree during traversal, the Morris Traversal uses no additional space for data structures like stacks or the system call stack, making it a very space-efficient approach.

The result is a list of node values `ans` that have been collected in inorder. This list is then returned, providing the solution to the problem.

This method achieves an `O(n)` time complexity for traversing through `n` nodes and `O(1)` space complexity, as it does not utilize recursion or an explicit stack to maintain the state during the traversal.

Example Walkthrough

Let's consider a simple binary tree for our example:

```
    2
   / \
  1   3
```

In this tree, we have three nodes, where 2 is the root node, 1 is to its left, and 3 is to its right. We want to perform an inorder traversal, which would visit the nodes in the order `1, 2, 3`, as 1 comes first in the left subtree, followed by 2, the root, and finally 3 in the right subtree.

Using the Morris Traversal approach, we would proceed as follows:

1. We start at the root, which is `2`. The root has a left child, so we find the inorder predecessor which is the rightmost node in the left subtree of `2` (which happens to be the node itself since it has no right child in its subtree). Since node `1` has no right child, we make a temporary link from node `1` to node `2` and then move the root to its left child (`1`).

2. Now the current `root` is `1`, which has no left child. Since there's no left subtree to process, we add `1` to the `ans` list. There is also no right child, so we would follow the temporary link back to `2`. After visiting node `1`, we have `ans = [1]`.

3. We arrive back at `2` because of the temporary link. We remove that temporary link and add `2` to the `ans` list, as we now are to visit this root node. Then we move to the right child of `2`, which is node `3`. Now the `ans` list is `[1, 2]`.

4. At node `3`, since there is no left child to process, we visit this node and add its value to the `ans` list. Now `ans = [1, 2, 3]`.

5. As there are no more unvisited nodes left, and the right child of `3` is `None`, the traversal is complete.

The final `ans` list is `[1, 2, 3]`, which is indeed the inorder traversal of the given binary tree.

Note that during the entire process, no extra space was used for stack or recursion, and the tree's original structure was restored after it had been temporarily altered to maintain the traversal state.

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 inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        # Initialize the output list to store the inorder traversal
        result = []

        # Continue traversing until there are no more nodes to process
        while root:
            # If there is no left child, add the current node's value to the result
            # and move to the right child
            if root.left is None:
                result.append(root.val)
                root = root.right
            else:
                # Find the rightmost node in the left subtree or the left child itself
                # if it does not have a right child. This node will be our "predecessor"
                predecessor = root.left
                while predecessor.right and predecessor.right != root:
                    predecessor = predecessor.right

                # If the predecessor's right child is not set to the current node,
                # set it to the current node and move to the left child of the current node
                if predecessor.right is None:
                    predecessor.right = root
                    root = root.left
                else:
                    # If the predecessor's right child is set to the current node,
                    # it means we have processed the left subtree, so add the current
                    # node's value to the result and sever the temporary link to restore
                    # the tree structure. Then, move to the right child.
                    result.append(root.val)
                    predecessor.right = None
                    root = root.right

        # Return the result of the inorder traversal
        return result

```

Java Solution

```java
/**
 * Definition for a binary tree node.
 * public 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 {
    public List<Integer> inorderTraversal(TreeNode root) {
        // Initialize an empty list to store the inorder traversal result
        List<Integer> result = new ArrayList<>();

        // Continue the process until all nodes are visited
        while (root != null) {
            // If there is no left child, visit the current node and go to the right child
            if (root.left == null) {
                result.add(root.val);
                root = root.right;
            } else {
                // Find the inorder predecessor of the current node
                TreeNode predecessor = root.left;
                // Move to the rightmost node of the left subtree or
                // the right child of the predecessor if it's already set
                while (predecessor.right != null && predecessor.right != root) {
                    predecessor = predecessor.right;
                }
                // If the right child of the predecessor is not set,
                // this means this is our first time visit this node, thus,
                // set the right child to the current node and move to the left child
                if (predecessor.right == null) {
                    predecessor.right = root;
                    root = root.left;
                } else {
                    // If the right child is already set to the current node,
                    // it means we are visiting the node the second time.
                    // Thus, we should visit the current node and remove the link.
                    result.add(root.val);
                    predecessor.right = null;
                    root = root.right;
                }
            }
        }
        // Return the completed list of nodes in inorder
        return result;
    }
}

```

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:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;      // This vector will store the inorder traversal result.

        // Loop through all nodes of the tree using Morris Traversal technique.
        while (root != nullptr) {
            // If there is no left subtree, print the root and move to the right subtree.
            if (!root->left) {
                result.push_back(root->val); // Add the current node value.
                root = root->right;          // Move to the right subtree.
            } else {
                // Find the inorder predecessor of the current root.
                TreeNode* predecessor = root->left;

                // Navigate to the rightmost node of the left subtree or to the current root
                // if the link is already established.
                while (predecessor->right != nullptr && predecessor->right != root) {
                    predecessor = predecessor->right;
                }

                // Establish a link from the predecessor to the current root, if it does not exist.
                if (!predecessor->right) {
                    predecessor->right = root; // Link predecessor to the root.
                    root = root->left;         // Move root to its left child.
                } else {
                    // A link from the predecessor to the current root already exists,
                    // which means we have finished processing the left subtree.
                    result.push_back(root->val); // Add the value of the current node.
                    predecessor->right = nullptr; // Remove the link to restore tree structure.
                    root = root->right;           // Move to the right subtree.
                }
            }
        }
        return result; // Return the result vector containing the inorder traversal.
    }
};

```

Typescript Solution

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

/**
 * Performs an inorder traversal of a binary tree.
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @returns {number[]} - An array of node values in the inorder sequence.
 */
function inorderTraversal(root: TreeNode | null): number[] {
    // Base case: if the current root is null, return an empty array.
    if (root === null) {
        return [];
    }

    // Recursive case:
    // 1. Traverse the left subtree and collect the values.
    // 2. Include the value of the current node.
    // 3. Traverse the right subtree and collect the values.
    // Then concatenate them in inorder sequence.
    return [
        ...inorderTraversal(root.left), // Left subtree values
        root.val,                       // Current node value
        ...inorderTraversal(root.right) // Right subtree values
    ];
}

```

Time and Space Complexity

The code implements the Morris In-order Traversal algorithm for a binary tree. Let's analyze both the time and space complexity:

Time Complexity

The time complexity of the algorithm is `O(n)`, where `n` is the number of nodes in the binary tree. Each node gets visited exactly twice in the worst case, once to establish the temporary link to its in-order predecessor and once to remove it and visit the node itself. The otherwise `O(n)` traversal is not affected by this temporary linking as it only adds a constant amount of work for each node.

Space Complexity

The space complexity of the algorithm is `O(1)`. Unlike traditional in-order traversal using recursion (which could lead to `O(h)` space complexity where `h` is the height of the tree due to the call stack), the Morris Traversal does not use any additional space for auxiliary data structures such as stacks or recursion. It uses the given tree's null right pointers to temporarily store the successors of nodes, thereby using the tree itself to guide the traversal.