Documentation for Binary Tree Inorder Traversal

Problem Description

Given the root of a binary tree, return the inorder traversal of its nodes' values.

Example 1:

Input: root = [1, null, 2, 3]

Output: [1, 3, 2]

Example 2:

Input: root = []

Output: []

Example 3:

Input: root = [1]

Output: [1]

Constraints

- The number of nodes in the tree is in the range [0, 100].
- -100 <= Node.val <= 100

Follow-up

- Recursive solution is trivial, could you do it iteratively?

Explanation

1. **TreeNode Class:** This class defines a node in the binary tree. Each node contains a value (val), a reference to the left child (left), and a reference to the right child (right).

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

2. **Solution Class:** This class contains the method inorderTraversal that performs the inorder traversal of a binary tree.

```
class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
```

3. **Inorder Traversal:**

- Initialization: An empty list result is used to store the values of nodes in inorder. An empty list stack is used to simulate the recursion stack. current is initialized to root.

```
result = []

stack = []

current = root
```

- Traversal Loop: The outer while loop continues as long as there is a node to process (current) or the stack is not empty.

```
while current or stack:
```

- Left Subtree Traversal: The inner while loop goes as deep as possible down the left subtree, pushing each node onto the stack.

```
while current:
    stack.append(current)
    current = current.left
```

- Node Processing: Once there are no more left nodes, the node at the top of the stack is popped, its value is added to the result, and current is moved to its right child.

```
current = stack.pop()
result.append(current.val)
current = current.right
```

4. **Return Result:** After the traversal is complete, the result list contains the inorder traversal of the binary tree.

```
return result
```

Iterative Approach

This solution uses an iterative approach with an explicit stack to simulate the system's call stack that would be used in a recursive approach. This method avoids the pitfalls of recursion such as stack overflow for very deep trees.

By following this method, we ensure that we visit each node in the correct inorder sequence: left subtree, root, right subtree. This iterative approach is efficient and well-suited for larger trees where recursion might not be feasible due to depth limitations.