94. Binary Tree Inorder Traversal

Total Accepted: **100218** Total Submissions: **264514** Difficulty: **Medium**

> Given a binary tree, return the *inorder* traversal of its nodes' values.
>
> For example:
>
> Given binary tree `{1,#,2,3}`,
>
> ```
>    1
>     \
>      2
>     /
>    3
> ```
>
> return `[1,3,2]`.

Binary Tree review:
A binary tree is an ordered tree with the following properties:
1. Every node has at most two children.
2. Each child node is labeled as being either a left child or a right child.
3. A left child precedes a right child in the order of children of a node.

The subtree rooted at a left or right child of an internal node v is called a left subtree or right subtree, respectively, of v. A binary tree is proper if each node has either zero or two children. Some people also refer to such trees as being full binary trees. Thus, in a proper binary tree, every internal node has exactly two children. A binary tree that is not proper is improper.

Inorder traversal of Binary Tree
Visit a position between the recursive traversals of its left and right subtrees.
The inorder traversal of a binary tree T can be informally viewed as visiting the nodes of T “from left to right.”
Indeed, for every position p, the inorder traversal visits p after all the positions in the left subtree of p and before all the positions in the right subtree of p.
Pseudo-code

```
inorder(p):
        if p has left child lc
                inorder(lc)
        perform the visit action for position p
        if p has a right child rc then
                inorder(rc)
```

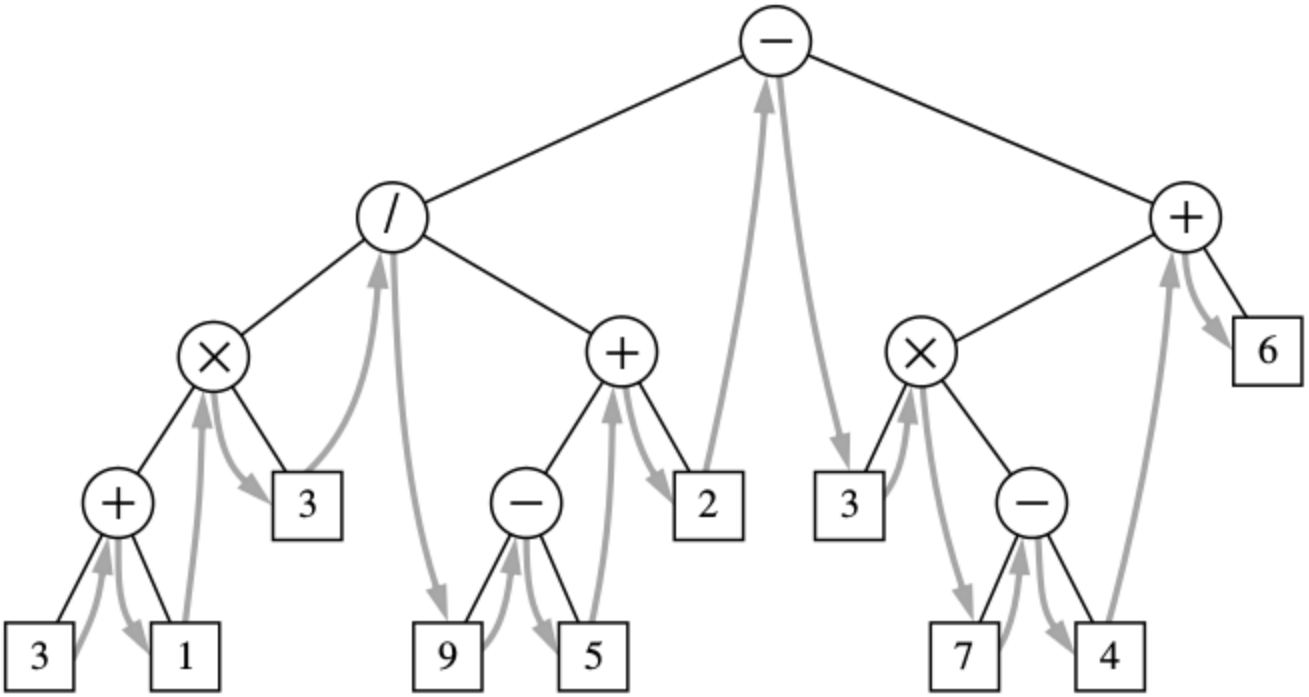

Figure 8.18: Inorder traversal of a binary tree.

```python
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    # @param {TreeNode} root
    # @return {integer[]}
    def __init__(self):
        self.result = []

    def inorderTraversal(self, root):
        if root:
            self.inorderTraversal(root.left)
            self.result.append(root.val)
            self.inorderTraversal(root.right)

        return self.result
```

Solution 2: Using a stack to keep track on every node that has been visited.

First, the current pointer is initialized to the root. Keep traversing to its left child while pushing visited nodes onto the stack.

When we reach the Null node, we pop off an element from the stack and set it to current. Now is the time to add the node to result. Meaning that this one has been visited.

Then the current is set to its right child and repeat the process again. When the stack is empty, we are done.

```
class Solution(object):
    # @param {tree node} root
    # @return {integer[]}

    def inorderTraversal(self, root):

        stack = []
        result = []
        current = root
        done = False

        while done is False:      #done is false
            if current is not None:
                stack.append(current)
                current = current.left
            else:
                if len(stack)==0:
                    done = True
                else:
                    current = stack.pop()
                    result.append(current.val)
                    current = current.right

        return result
```