Binary Tree Inorder Traversal

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]`.

Note: Recursive solution is trivial, could you do it iteratively?

confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.

OJ's Binary Tree Serialization:
The serialization of a binary tree follows a level order traversal, where '#' signifies a path terminator where no node exists below.

Here's an example:

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

The above binary tree is serialized as `"{1,2,3,#,#,4,#,#,5}"`.

Solution 1

```
public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> list = new ArrayList<Integer>();

    Stack<TreeNode> stack = new Stack<TreeNode>();
    TreeNode cur = root;

    while(cur!=null || !stack.empty()){
        while(cur!=null){
            stack.add(cur);
            cur = cur.left;
        }
        cur = stack.pop();
        list.add(cur.val);
        cur = cur.right;
    }

    return list;
}
```

written by lvlolitte original link here

Solution 2

Method 1: Using one stack and the binary tree node will be changed. Easy ,not Practical

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode *root) {
        vector<int> vector;
        if(!root)
        return vector;
        stack<TreeNode *> stack;
        stack.push(root);
        while(!stack.empty())
        {
            TreeNode *pNode = stack.top();
            if(pNode->left)
            {
                stack.push(pNode->left);
                pNode->left = NULL;
            }
            else
            {
                vector.push_back(pNode->val);
                stack.pop();
                if(pNode->right)
                stack.push(pNode->right);
            }
        }
        return vector;
    }
};
```

Method 2: Using one stack and one unordered_map, this will not changed the node. Better

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode *root) {
        vector<int> vector;
        if(!root)
        return vector;
        unordered_map<TreeNode *, bool> map;//left child has been visited:true.
        stack<TreeNode *> stack;
        stack.push(root);
        while(!stack.empty())
        {
            TreeNode *pNode = stack.top();
            if(pNode->left && !map[pNode])
            {
                stack.push(pNode->left);
                map[pNode] = true;
            }
            else
            {
                vector.push_back(pNode->val);
                stack.pop();
                if(pNode->right)
                stack.push(pNode->right);
            }
        }
        return vector;
    }
};
```

Method 3: Using one stack and will not changed the node. Best(at least in this three solutions)

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode *root) {
        vector<int> vector;
        stack<TreeNode *> stack;
        TreeNode *pCurrent = root;

        while(!stack.empty() || pCurrent)
        {
            if(pCurrent)
            {
                stack.push(pCurrent);
                pCurrent = pCurrent->left;
            }
            else
            {
                TreeNode *pNode = stack.top();
                vector.push_back(pNode->val);
                stack.pop();
                pCurrent = pNode->right;
            }
        }
        return vector;
    }
};
```

written by KaiChen original link here

Solution 3

Hi, this is a fundamental and yet classic problem. I share my three solutions here:

1. Iterative solution using stack --- `O(n)` time and `O(n)` space;
2. Recursive solution --- `O(n)` time and `O(n)` space (considering the spaces of function call stack);
3. **Morris traversal --- `O(n)` time and `O(1)` space!!!**

Iterative solution using stack:

```
vector<int> inorderTraversal(TreeNode* root) {
    vector<int> nodes;
    stack<TreeNode*> toVisit;
    TreeNode* curNode = root;
    while (curNode || !toVisit.empty()) {
        if (curNode) {
            toVisit.push(curNode);
            curNode = curNode -> left;
        }
        else {
            curNode = toVisit.top();
            toVisit.pop();
            nodes.push_back(curNode -> val);
            curNode = curNode -> right;
        }
    }
    return nodes;
}
```

Recursive solution:

```
void inorder(TreeNode* root, vector<int>& nodes) {
    if (!root) return;
    inorder(root -> left, nodes);
    nodes.push_back(root -> val);
    inorder(root -> right, nodes);
}
vector<int> inorderTraversal(TreeNode* root) {
    vector<int> nodes;
    inorder(root, nodes);
    return nodes;
}
```

Morris traversal:

```
vector<int> inorderTraversal(TreeNode* root) {
    TreeNode* curNode = root;
    vector<int> nodes;
    while (curNode) {
        if (curNode -> left) {
            TreeNode* predecessor = curNode -> left;
            while (predecessor -> right && predecessor -> right != curNode)
                predecessor = predecessor -> right;
            if (!(predecessor -> right)) {
                predecessor -> right = curNode;
                curNode = curNode -> left;
            }
            else {
                predecessor -> right = NULL;
                nodes.push_back(curNode -> val);
                curNode = curNode -> right;
            }
        }
        else {
            nodes.push_back(curNode -> val);
            curNode = curNode -> right;
        }
    }
    return nodes;
}
```

written by jianchao.li.fighter original link here