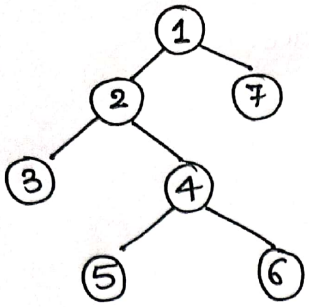
Iterative Pre-order Traversal In Binary Tree

→ Root - Left → Right.

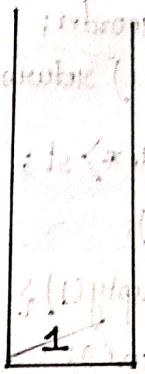
1 2 3 4 5 6 7 → "Pre-Order Traversal."



how to solve w/o Recursion → take stack

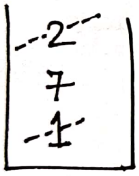
Whatever is at root of the tree put it into the stack.

→ Now Important Point is First Take Right and then Left and put it into the Stack.



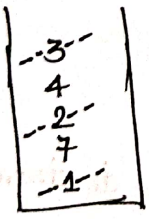
Stack is a last in first out as a Recursion D.S.

* Whatever is in the stack print it.



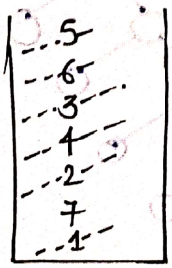
* So Now remove LIFO element → Now 1, 2. is in the Queue.

So on next Traversal



→ So the output 1, 2, 3, after this there is nothing on the left side and right side of the node.

* So on next traversal you find 4 so 1, 2, 3, 4 was print



→ 1, 2, 3, 4, 5, 5 doesn't have left or right. so take 5,

Now take 6, 6 doesn't have left or rights, so take 6.

1, 2, 3, 4, 5, 6

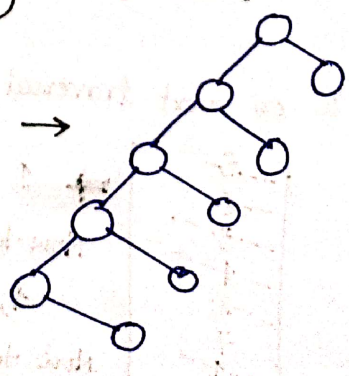
Now take 7, 1 2 3 4 5 6 7.

C++ Code:-

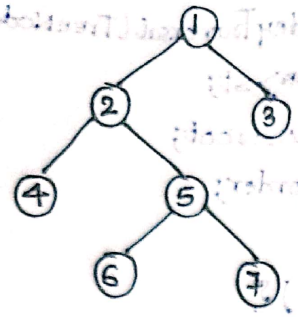
```
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root)   // Root of the Tree.
    {
        vector<int> preorder;
        if(root == NULL) return preorder;   // means tree is empty.

        stack<TreeNode*> st;   // take stack
        st.push(root);   // and initially you put root.
        while(!st.empty()){   // keep iterating of the stack, if not empty.
            root = st.top();   // you get the top most element. -> in every iteration take top most of the stack, that is your preorder.
            st.pop();
            preorder.push_back(root->val);
            if(root->right != NULL){   // check if there exists a right.
                st.push(root->right);
            }
            if(root->left != NULL){   // check if there exists a left.
                st.push(root->left);
            }
        }   // Once the entire stack becomes empty;
        }   // Return the Pre-order.
        return preorder;
    }
};
```

TIME COMPLEXITY = O(N) → because you have to travel for every node.

S.C = at the WORST CASE is O(N). →



ITERATIVE INORDER TRAVERSAL IN BINARY TREE

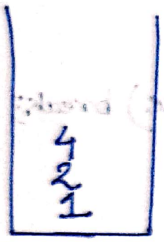


→ Inorder Traversal of the Tree :→

∴ 4 2 6 5 7 1 3

LEFT → ROOT → RIGHT

node = 1, remember at every iteration take whatever is present at the node. put it in the stack.



Stack - D.S

LIFO D.S

node = 1

2

4

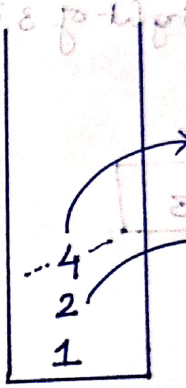
null.

↳ If you found null there is NO NEED TO FOUND ANY NODE, ANY MORE.

No 4's right is null

So there is one more null.

take element and print it.



4

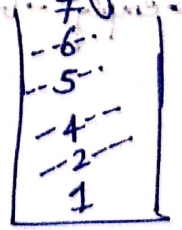
2

↳ null. :(now remove 2 from the stack) =

now (till here it is 2's ~~right~~ left

now moves to 4 was right).

↳ 5 →



4265.7

↳ 6 →

↳ left of 6 is null

↳ so it will take top of the stack.

↳ right of 6 is null again. → so now take 5.

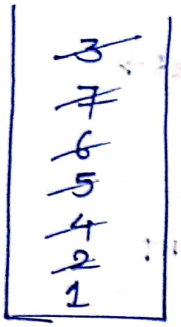
~~7~~ →

↳ ~~null~~

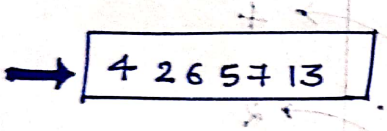
↳ null.

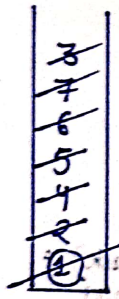
C++ code.

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



→ 3
↳ left is null.
Print 3
↳ right of 3 is null.



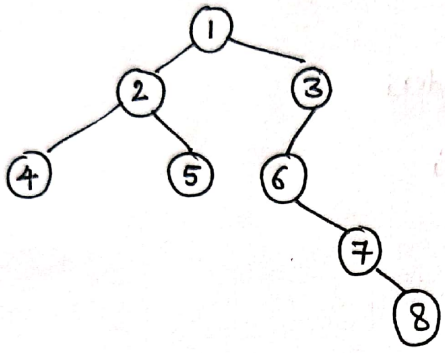


st → LIFO
↳ empty - break.

Data Structure and Algorithms
By
Malay Tripathi

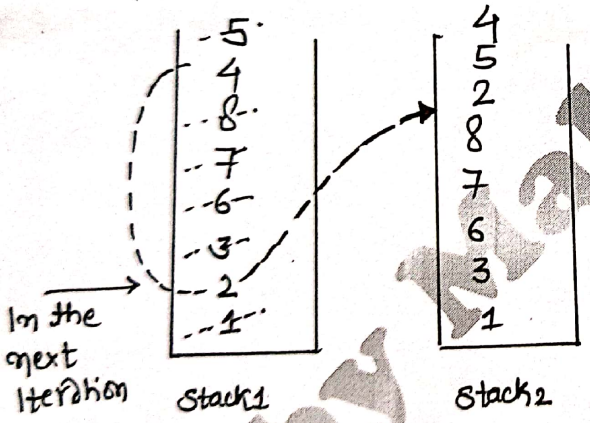
ITERATIVE POST-ORDER TREE TRAVERSAL

→ It is using a stack.



4 5 2 8 7 6 3 1

Pre-Order Traversal → LEFT RIGHT ROOT.



4 5 2 8 7 6 3 1.

- Every value when removed from stack1 than every element of its left and right put into the stack1. Then when ever top element removed (same steps are followed. and ultimately whole stack2 will be poped out. and we get post order.

Data Structure and Algorithms
By
Malay Tripathi

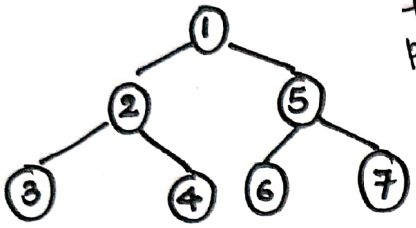
```
class Solution {
public:
 vector<int> postorderTraversal(TreeNode* root) {
  vector<int> postorder;
  if(root == NULL) return postorder;
  stack<TreeNode*> st1, st2;
  st1.push(root);
  while(!st1.empty()) {
    root = st1.top();
    st1.pop();
    st2.push(root);
    if(root->left != NULL) {
      st1.push(root->left);
    }
    if(root->right != NULL) {
      st1.push(root->right);
    }
  }
  while(!st2.empty()) {
    postorder.push_back(st2.top()->val);
    st2.pop();
  }
  return postorder;
 }
};
```

∴ TIME COMPLEXITY - O(N) ∴ because Traversing all the Node.

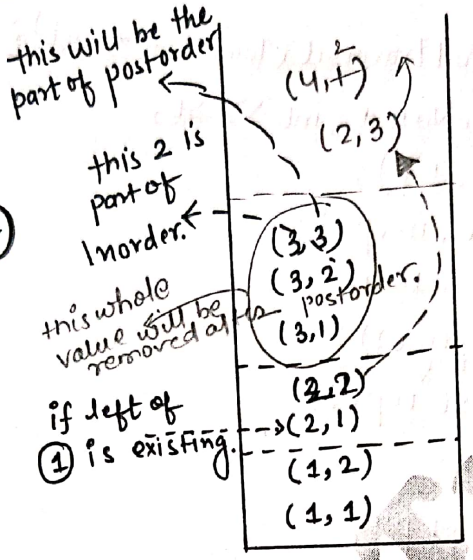
∴ SPACE COMPLEXITY - O(2N) ∴ becz. 2 STACKS.

Data Structure and Algorithms
By
Malay Tripathi

PREORDER / INORDER / POSTORDER TRAVERSALS IN ONE TRAVERSAL



Pre → 1 2 3 4 5 ~~6~~ 6 7
In → 3 2 4 1 6 5 7
Post → 3 4 2 6 7 5 1



if num = 1
preorder
++
left

if num = 2
inorder
++
right

if num = 3
postorder.

Stack - (mode, num) - LIFO

TIME COMPLEXITY → O(3×N)
→ Because we are iterating for thrice.

SPACE COMPLEXITY → ~~O(2N)~~. O(4N).

C++ CODE

Data Structure and Algorithms
By
Malay Tripathi

```
class Solution {
public:
  vector<int> preInPostTraversal(TreeNode* root) {
    stack<pair<TreeNode*, int>> st;
    st.push({root, 1});
    vector<int> pre, in, post;
    if(root == NULL) return;
    while(!st.empty()) {
        auto it = st.top();
        st.pop();
    // this is part of pre
    // increment 1 to 2
    // push the leftSide of the tree
    if(it.second == 1) {
        pre.push_back(it.first->val);
        it.second++;
        st.push(it);
        if(it.first->left != NULL) {
            st.push({it.first->left, 1});
        }
    }
    //this is a part of in
    // increment 2 to 3
    // push right
    else if(it.second == 2) {
        in.push_back(it.first->val);
        it.second++;
        st.push(it);
        if(it.first->right != NULL) {
            st.push({it.first->right, 1});
        }
    }
    // don't push it back again
    else {
        post.push_back(it.first->val);
    }
  }
};
```