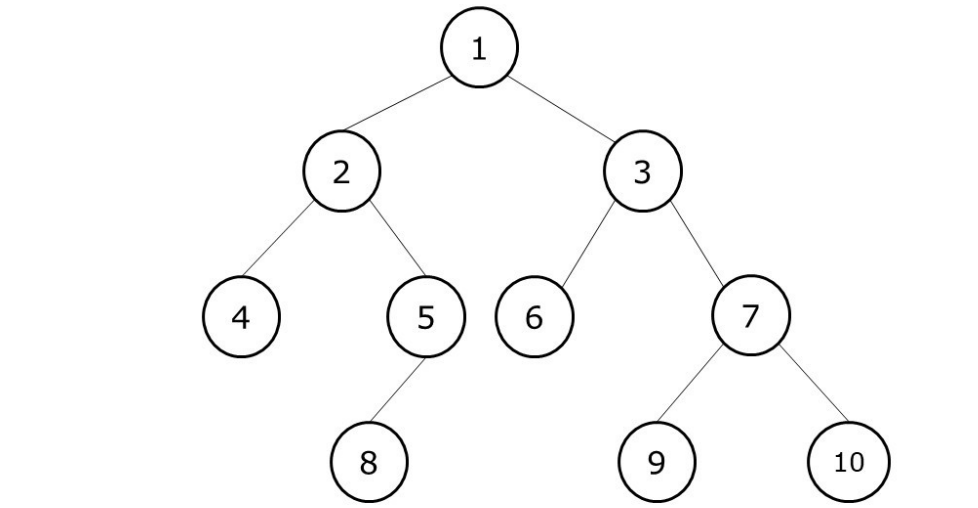
Post-Order Traversal Of Binary Tree
Problem Statement: Postorder Traversal of a binary tree. Write a program for the postorder traversal of a binary tree.

Example:



Postorder Traversal:
[left, right, root]

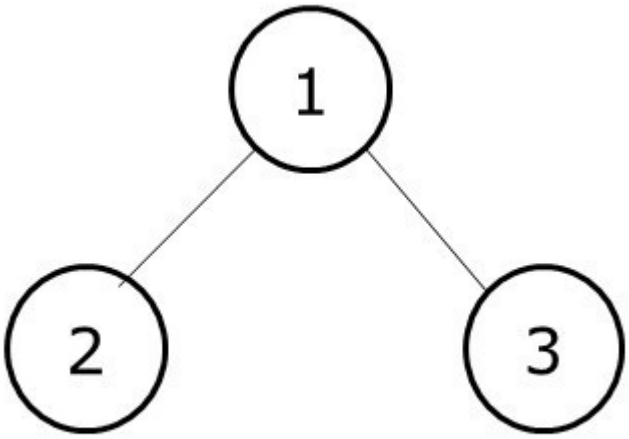
4	8	5	2	6	9	10	7	3	1

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution [Recursive]:

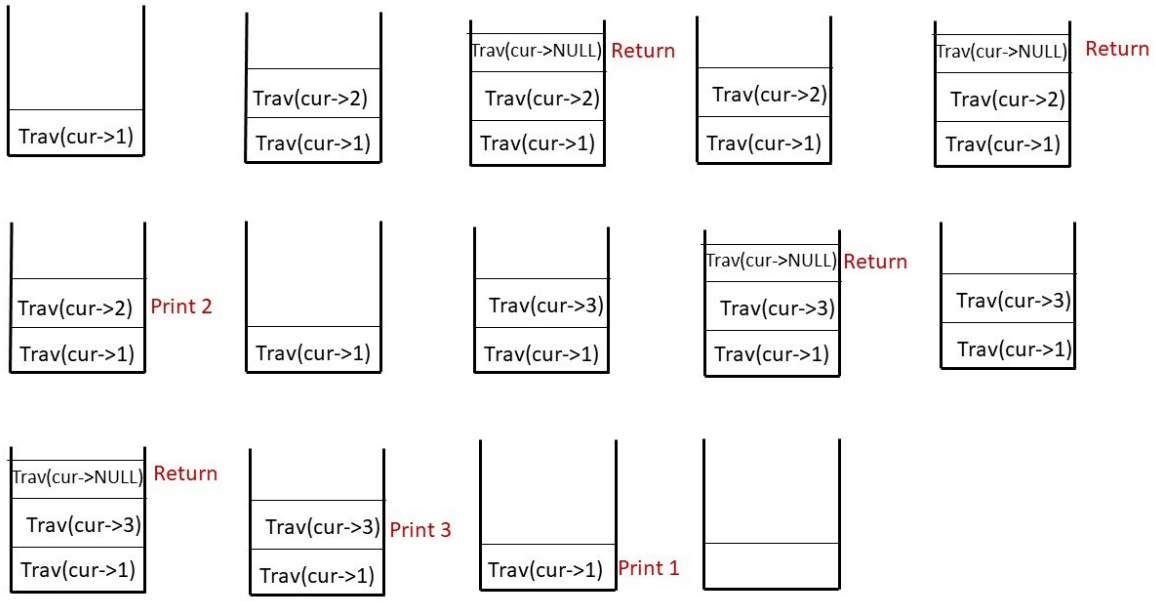
Approach: In postorder traversal, the tree is traversed in this way: **left, right**, **root**.
The algorithm approach can be stated as:
- We first recursively visit the left child and go on left till we find a node pointing to NULL.
- Then we return to its parent.
- Then we recursively visit the right child.
- After we have returned from the right child as well, only then will we print the current node value.

Explanation: It is very important to understand how recursion works behind the scenes to traverse the tree. For it we will see a simple case:



Postorder traversal of this tree: 2,3,1
Initially, we pass the root node pointing to 1 to our traversal function. The algorithm steps are as follows:
- As we are doing a postorder traversal, the first thing we will do is to recursively visit the left child. We continue till the time we find a node pointing to NULL
- As we can't move further left, we need to return back to node 2 but how do we do so? Remember that our nodes only have pointers to the children and not to the parent, therefore we can move only from parent to child and not from a child to the parent.
- The answer to this question is **recursion**. We recursively called the same function with the current node pointing to the NULL node.This second function was pushed to our call stack. We do our execution which is to return from that node(as the node is pointing to NULL).
- As the execution stops, we need to come back to the parent, we simply return to it as is present at the top of the recursion call stack.
- Similar execution is performed at all nodes.

The call stack diagram will help to understand the recursion better.



Dry Run: In case you want to watch the dry run for this approach, please watch the video attached below.
Code:

- C++ Code
- Java Code

```cpp
#include <bits/stdc++.h>

using namespace std;

struct node {
  int data;
  struct node * left, * right;
};

void postOrderTrav(node * curr, vector < int > & postOrder) {
  if (curr == NULL)
    return;

  postOrderTrav(curr -> left, postOrder);
  postOrderTrav(curr -> right, postOrder);
  postOrder.push_back(curr -> data);
}

struct node * newNode(int data) {
  struct node * node = (struct node * ) malloc(sizeof(struct node));
  node -> data = data;
  node -> left = NULL;
  node -> right = NULL;

  return (node);
}

int main() {

  struct node * root = newNode(1);
  root -> left = newNode(2);
  root -> right = newNode(3);
  root -> left -> left = newNode(4);
  root -> left -> right = newNode(5);
  root -> left -> right -> left = newNode(8);
  root -> right -> left = newNode(6);
```

```
    root -> right -> right = newNode(7);
    root -> right -> right -> left = newNode(9);
    root -> right -> right -> right = newNode(10);

    vector < int > postOrder;
    postOrderTrav(root, postOrder);

    cout << "The postOrder Traversal is : ";
    for (int i = 0; i < postOrder.size(); i++) {
        cout << postOrder[i] << " ";
    }
    return 0;
}
```

Output:

The postOrder Traversal is : 4 8 5 2 6 9 10 7 3 1

Time Complexity: O(N).

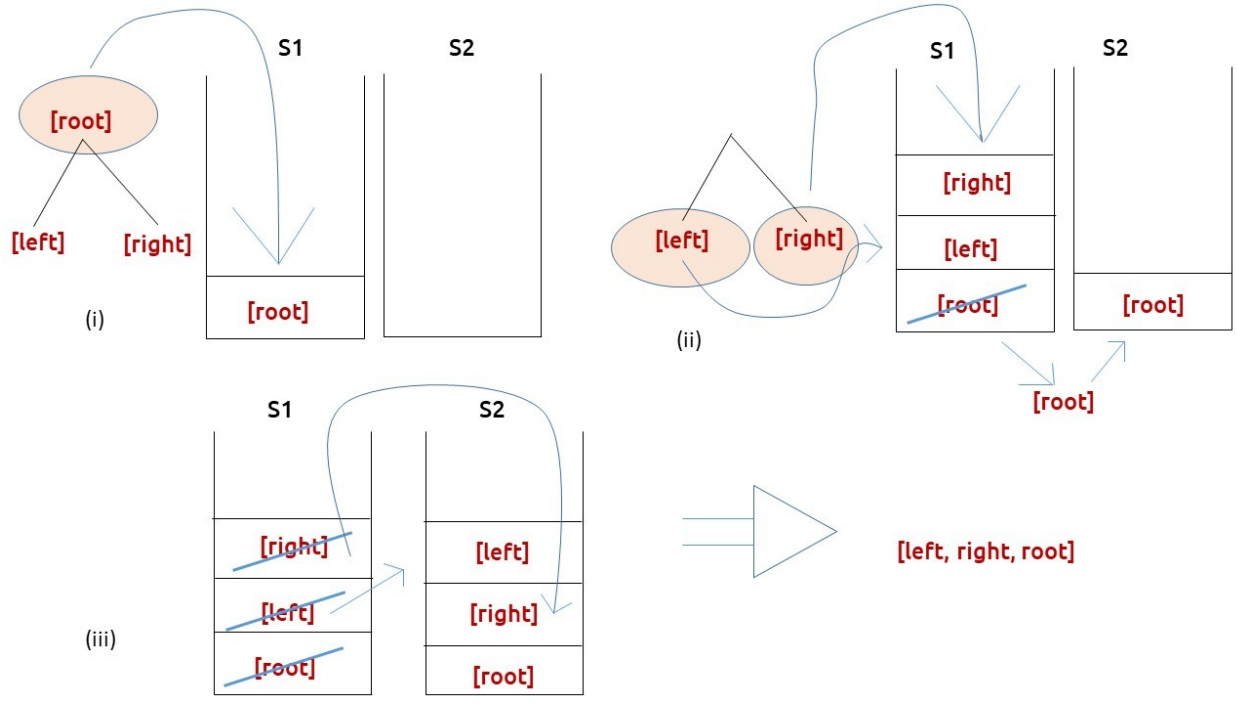
Reason: We are traversing N nodes and every node is visited exactly once.

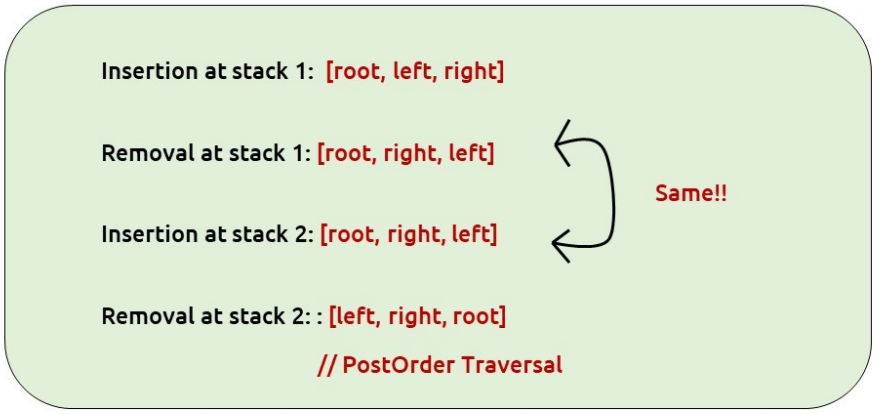
Space Complexity: O(N)

Reason: Space is needed for the recursion stack. In the worst case (skewed tree), space complexity can be O(N).

Iterative Solution
Prerequisite: Recursive solution to preorder traversal.
Intuition: In postorder traversal, the tree is traversed in this way: **left,** right, root. We first visit the left child, after returning from it we visit the right child, and after returning from both of them, we print the value of the current node. The fundamental problem we face in this scenario is that there is no way that we can move from a child to the parent using as our node points to only children and not to the parent. To solve this problem, we use an explicit stack data structure. While traversing we can insert node values to the stack in such a way that we always get the next node value at the top of the stack.
Solution 1: Using two stacks
Approach:
The algorithm approach can be stated as:
- We take two explicit stacks S1 and S2.
- We insert our node to S1(if it's not pointing to NULL).
- We will set up a while loop to run till S1 is non-empty.
- In every iteration, we pop out the top of S1 and then push this popped node to S2. Moreover we will push the left child and right child of this popped node to S1.(If they are not pointing to NULL).
- Then we start the next iteration with the next node as top of S1.
- We stop the iteration when S1 becomes empty.
- At last we start popping at the top of S2 and print the node values, we will get the postorder traversal.

Stack is a Last-In-First-Out (LIFO) data structure. To understand the two-stack approach, we need to understand how we insert and remove nodes in both stacks.





Dry Run: In case you want to watch the dry run for this approach, please watch the video attached below.
Code:

- C++ Code
- Java Code

```cpp
#include <bits/stdc++.h>

using namespace std;

struct node {
  int data;
  struct node * left, * right;
};

vector < int > postOrderTrav(node * curr) {

  vector < int > postOrder;
  if (curr == NULL) return postOrder;

  stack < node * > s1;
  stack < node * > s2;
  s1.push(curr);
  while (!s1.empty()) {
    curr = s1.top();
    s1.pop();
    s2.push(curr);
    if (curr -> left != NULL)
      s1.push(curr -> left);
    if (curr -> right != NULL)
      s1.push(curr -> right);
  }
  while (!s2.empty()) {
    postOrder.push_back(s2.top() -> data);
    s2.pop();
  }
  return postOrder;
}

struct node * newNode(int data) {
  struct node * node = (struct node * ) malloc(sizeof(struct node));
  node -> data = data;
  node -> left = NULL;
```

```
    node -> right = NULL;

    return (node);
}

int main() {

    struct node * root = newNode(1);
    root -> left = newNode(2);
    root -> right = newNode(3);
    root -> left -> left = newNode(4);
    root -> left -> right = newNode(5);
    root -> left -> right -> left = newNode(8);
    root -> right -> left = newNode(6);
    root -> right -> right = newNode(7);
    root -> right -> right -> left = newNode(9);
    root -> right -> right -> right = newNode(10);

    vector < int > postOrder;
    postOrder = postOrderTrav(root);

    cout << "The postOrder Traversal is : ";
    for (int i = 0; i < postOrder.size(); i++) {
        cout << postOrder[i] << " ";
    }
    return 0;
}
```

Output

The postOrder Traversal is : 4 8 5 2 6 9 10 7 3 1

Time Complexity: O(N).

Reason: We are traversing N nodes and every node is visited exactly once.

Space Complexity: O(N+N)

Solution 2: Using a single stack:
Intuition: First we need to understand what we do in a postorder traversal. We first explore the left side of the root node and keep on moving left until we encounter a node pointing to NULL. As now there is nothing more to traverse on the left, we move to the immediate parent(say node P) of that NULL node. Now we again start our left exploration from the right child of that node P. We will print a node's value only when we have returned from its both children.
Approach:
The algorithm steps can be stated as:
●We take an explicit data structure and a cur pointer pointing to the root of the tree.
●We run a while loop till the time the cur is not pointing to NULL or the stack is non-empty.
●If cur is not pointing to NULL, it means then we simply push that value to the stack and move the cur pointer to its left child because we want to explore the left child before the root or the right child.
●If the cur is pointing to NULL, it means we can't go further left, then we take a variable temp and set it to cur's parent's right child (as we have visited the left child, now we want to visit the right child). We have node cur's parent at the top of the stack.
●If node temp is not pointing to NULL, we simply initialise cur as node temp so that we can again start looking at the left of node temp from the next iteration.
●If node temp is pointing to NULL, then first of all we are sure that we have visited both children of temp's parent, so it's time to print it. Therefore we set temp to its parent(present at the top of stack), pop the stack and then print temp's value. Additionally, this new temp(parent of NULL-pointing node) can be the right child of the node present at the top of stack after popping.In that case the node at top of the stack is parent of temp and the next node to be printed. Therefore we run an additional while loop to check if that is the case, if true then again move temp to its parent and print its value.

Dry Run: In case you want to watch the dry run for this approach, please watch the video attached below.
Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

struct node {
    int data;
    struct node * left, * right;
};
```

```
vector < int > postOrderTrav(node * cur) {

   vector < int > postOrder;
   if (cur == NULL) return postOrder;

   stack < node * > st;
   while (cur != NULL || !st.empty()) {

      if (cur != NULL) {
         st.push(cur);
         cur = cur -> left;
      } else {
         node * temp = st.top() -> right;
         if (temp == NULL) {
            temp = st.top();
            st.pop();
            postOrder.push_back(temp -> data);
            while (!st.empty() && temp == st.top() -> right) {
               temp = st.top();
               st.pop();
               postOrder.push_back(temp -> data);
            }
         } else cur = temp;
      }
   }
   return postOrder;

}

struct node * newNode(int data) {
   struct node * node = (struct node * ) malloc(sizeof(struct node));
   node -> data = data;
   node -> left = NULL;
   node -> right = NULL;

   return (node);
}

int main() {

   struct node * root = newNode(1);
   root -> left = newNode(2);
   root -> right = newNode(3);
   root -> left -> left = newNode(4);
   root -> left -> right = newNode(5);
   root -> left -> right -> left = newNode(8);
   root -> right -> left = newNode(6);
   root -> right -> right = newNode(7);
   root -> right -> right -> left = newNode(9);
   root -> right -> right -> right = newNode(10);

   vector < int > postOrder;
   postOrder = postOrderTrav(root);

   cout << "The postOrder Traversal is : ";
   for (int i = 0; i < postOrder.size(); i++) {
      cout << postOrder[i] << " ";
   }
   return 0;
}
```

Output:

The postOrder Traversal is : 4 8 5 2 6 9 10 7 3 1

Time Complexity: O(N).

Space Complexity: O(N)