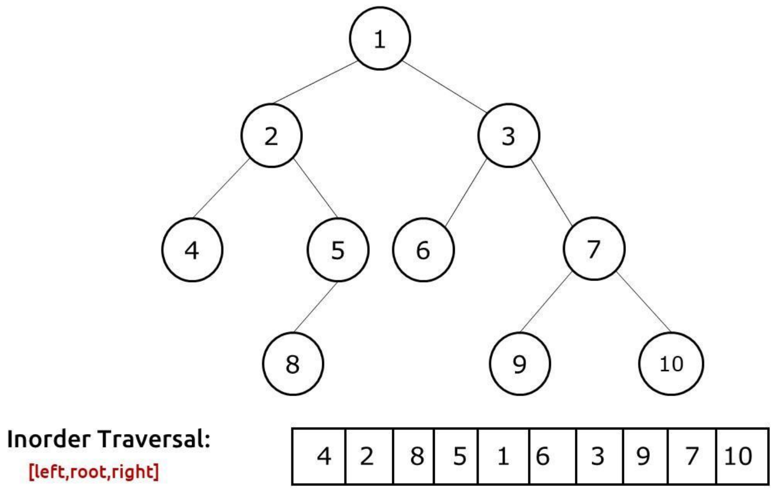
Inorder Traversal of Binary Tree

Problem Statement: Given a Binary Tree. Find and print the inorder traversal of Binary Tree.
Examples:
`Input:`



`Output:` `The inOrder Traversal is : 4 2 8 5 1 6 3 9 7 10`

Solution [Iterative]:

Intuition: In inorder traversal, the tree is traversed in this way: **root,** left, right. We first visit the left child, after returning from it we print the current node value, then we visit the right child. The fundamental problem we face in this scenario is that there is no way that we can move from a child to a 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.
Approach:
The algorithm approach can be stated as:

- We first take an explicit stack data structure and set an infinite loop.
- In every iteration we check whether our current node is pointing to NULL or not.
- If it is not pointing to null, we simply push the current value to the stack and move the current node to its left child.
- If it is pointing to NULL, we first check whether the stack is empty or not. If the stack is empty, it means that we have traversed the tree and we break out of the loop.
- If the stack is not empty, we pop the top value of the stack, print it and move the current node to its right child.

Stack is a Last-In-First-Out (LIFO) data structure, therefore when we encounter a node, we simply push it to the stack and try to find nodes on its left. When the current node points to NULL, it means that there is nothing left to traverse and we should move to the parent.
This parent is always placed at the top of the stack. If the stack is empty, then we had already traversed the whole tree and should stop the execution.

Dry Run: In case you want to watch the dry run for this approach, please watch the video attached below.
Code:
```
#include <bits/stdc++.h>

using namespace std;

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

vector < int > inOrderTrav(node * curr) {
  vector < int > inOrder;
  stack < node * > s;
  while (true) {
    if (curr != NULL) {
      s.push(curr);
      curr = curr -> left;
    } else {
      if (s.empty()) break;
      curr = s.top();
      inOrder.push_back(curr -> data);
      s.pop();
      curr = curr -> right;
    }
  }
  return inOrder;
}
```

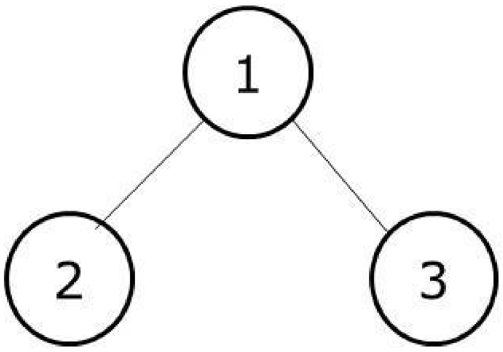
Output:
The inOrder Traversal is : 4 2 8 5 1 6 3 9 7 10
Time Complexity: O(N).
Reason: We are traversing N nodes and every node is visited exactly once.
Space Complexity: O(N)
Reason: In the worst case (a tree with just left children), the space complexity will be O(N).

Solution 2 [Recursive]:

Approach: In inorder traversal, the tree is traversed in this way: **left,** root, **right**.
The algorithm approach can be stated as:

- We first recursively visit the left child and go on till we find a leaf node.
- Then we print that node value.
- Then we recursively visit the right child.
- If we encounter a node pointing to NULL, we simply return to its parent.

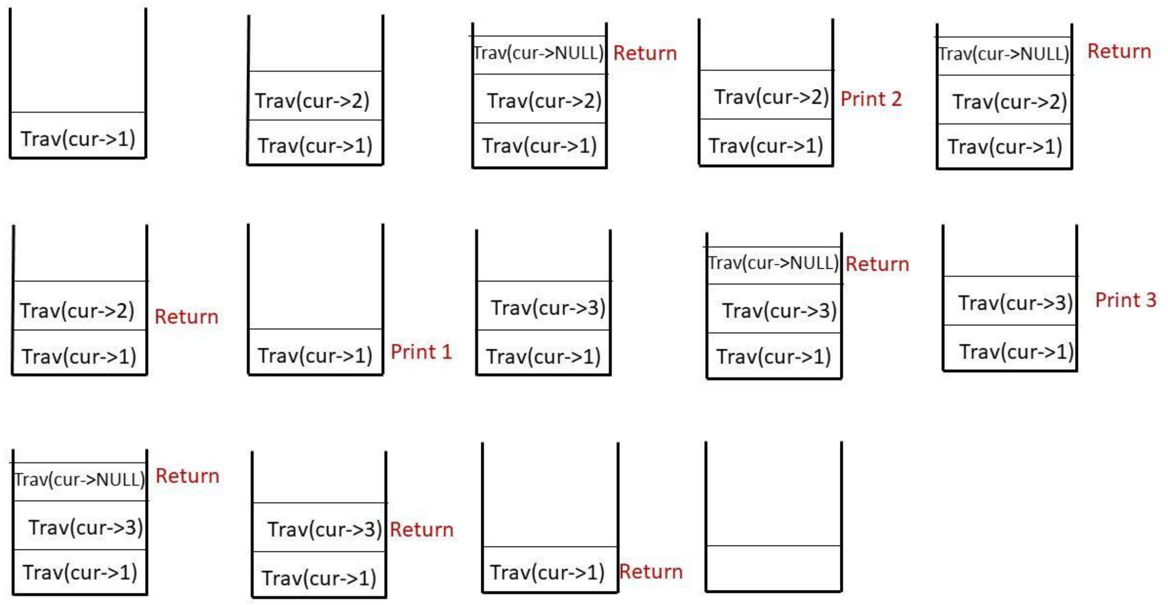
Explanation: It is very important to understand how recursion works behind the scenes to traverse the tree. For it we will see a simple case:



Inorder traversal of this tree: 2,1,3
Initially, we pass the root node pointing to 1 to our traversal function. The algorithm steps are as follows:

- As we are doing an inorder traversal, the first thing we will do is to recursively visit the left child. We continue till the time we find a leaf node. At node 2, as there is no more left child, we print its value.
- Then we need to move back to node 1 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**. When we move to node 2, we call it recursively. This second function is then pushed to our call stack. We do our execution which is to visit the left child of node 2, return from it as it is NULL, print current node value and then again recursively call its right child.
- This second function will then be removed from our call stack and we will return to the first function. Then we again recursively call the function for the right child and do the execution, i.e print 1 and then visit its right child.

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:

```
#include <bits/stdc++.h>
using namespace std;
struct node {
  int data;
  struct node * left, * right;
};
void inOrderTrav(node * curr, vector < int > & inOrder) {
  if (curr == NULL)
    return;

  inOrderTrav(curr -> left, inOrder);
  inOrder.push_back(curr -> data);
  inOrderTrav(curr -> right, inOrder);
}
```

Output:

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

Time Complexity: O(N).

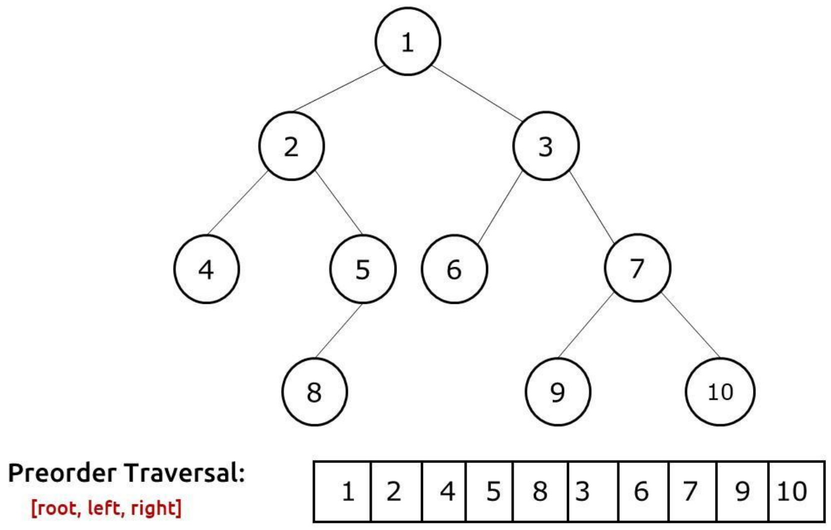
Reason: We are traversing N nodes and every node is visited exactly once.

Space Complexity: O(N)

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

Preorder Traversal of Binary Tree

Problem Statement: Given a binary tree print the preorder traversal of the binary tree.
Example:



Solution:
Solution 1: Iterative
Intuition: In preorder traversal, the tree is traversed in this way: **root,** left, right. When we visit a node, we print its value, and then we want to visit the left child followed by the right child. The fundamental problem we face in this scenario is that there is no way that we can move from a child to a 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.
Approach:
The algorithm approach can be stated as:

- We first take an explicit stack data structure and push the root node to it.(if the root node is not NULL).
- Then we use a while loop to iterate over the stack till the stack remains non-empty.
- In every iteration we first pop the stack's top and print it.
- Then we first push the right child of this popped node and then push the left child, if they are not NULL. We do so because stack is a last-in-first-out(LIFO) data structure. We need to access the left child first, so we need to push it at the last.
- The execution continues and will stop when the stack becomes empty. In this process, we will get the preorder traversal of the tree.

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

Code:

```
#include <bits/stdc++.h>
using namespace std;
struct node {
  int data;
  struct node * left, * right;
};
vector < int > preOrderTrav(node * curr) {
  vector < int > preOrder;
  if (curr == NULL)
    return preOrder;

  stack < node * > s;
  s.push(curr);

  while (!s.empty()) {
    node * topNode = s.top();
    preOrder.push_back(topNode -> data);
    s.pop();
    if (topNode -> right != NULL)
      s.push(topNode -> right);
    if (topNode -> left != NULL)
      s.push(topNode -> left);
  }
  return preOrder;

}
```

Output:

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

Time Complexity: O(N).

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

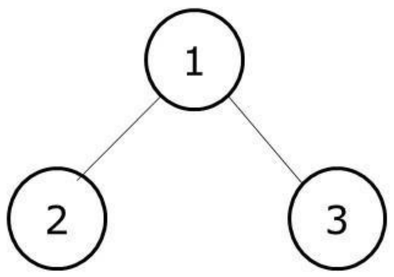
Space Complexity: O(N)

Reason: In the worst case, (a tree with every node having a single right child and left-subtree, follow the video attached below to see the illustration), the space complexity can be considered as O(N).

Solution 2: Recursive
Intuition: In preorder traversal, the tree is traversed in this way: **root,** left, right. When we visit a node, we print its value, and then we want to visit the left child followed by the right child. The fundamental problem we face in this scenario is that there is no way that we can move from a child to a parent. To solve this problem, we use recursion and the recursive call stack to locate ourselves back to the parent node when execution at a child node is completed.
Approach: In preorder traversal, the tree is traversed in this way: **root, left, right**.
The algorithm approach can be stated as:

- We first visit the root node and before visiting its children we print its value.
- After this, we recursively visit its left child.
- Then we recursively visit the right child.
- If we encounter a node pointing to NULL, we simply return to its parent.

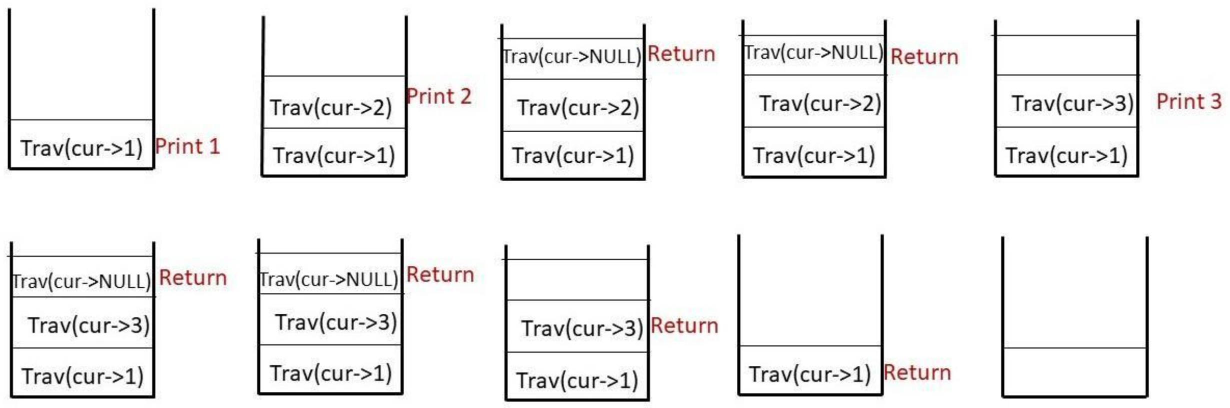
Code:
```
#include <bits/stdc++.h>
using namespace std;
struct node {
  int data;
  struct node * left, * right;
};
void preOrderTrav(node * curr, vector < int > & preOrder) {
  if (curr == NULL)
    return;
  preOrder.push_back(curr -> data);
  preOrderTrav(curr -> left, preOrder);
  preOrderTrav(curr -> right, preOrder);
}
```
Output:
```
The preOrder Traversal is : 1 2 4 5 8 3 6 7 9 10
```
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).
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:



Preorder traversal of this tree: 1,2,3.
Initially, we pass the root node pointing to 1 to our traversal function. The algorithm steps are as follows:

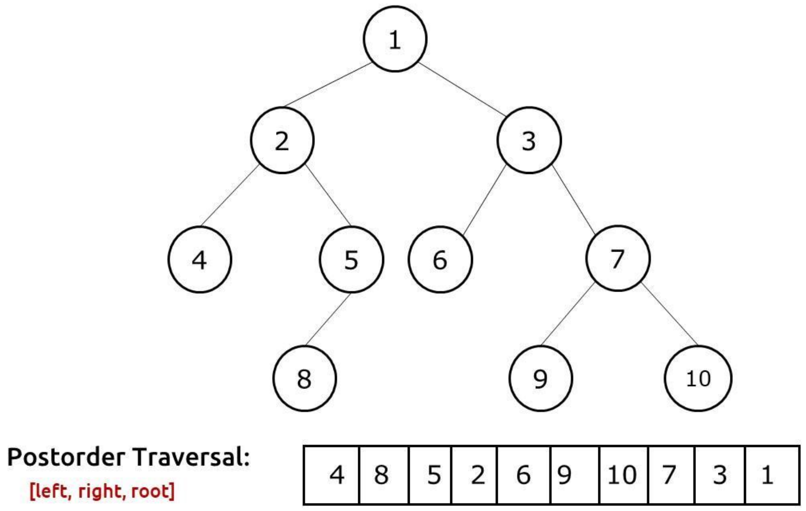
- As we are doing a preorder traversal, the first thing we will do is to print the current node value, i.e 1.
- Then we need to move left but how do we do so? Remember that our nodes only have pointers to the children and not to the parent, therefore we can move only from parent to child and not from a child to the parent.
- The answer to this question is **recursion**. We call the same function with the current node pointing to the left child, i.e 2. This second function is then pushed to our call stack. We do our execution which is to visit node 2 print it and then again recursively call its both children. As both of them point to NULL, we will return from them and execution of the second function stops.
- This second function will then be removed from our call stack and we will return to the first function. Then we again recursively call the function for the right child and do the execution.

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



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:`

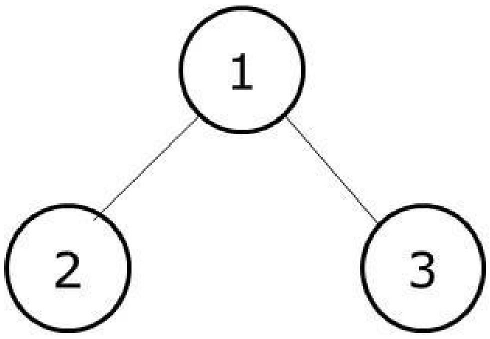


Solution [Recursive]:

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.

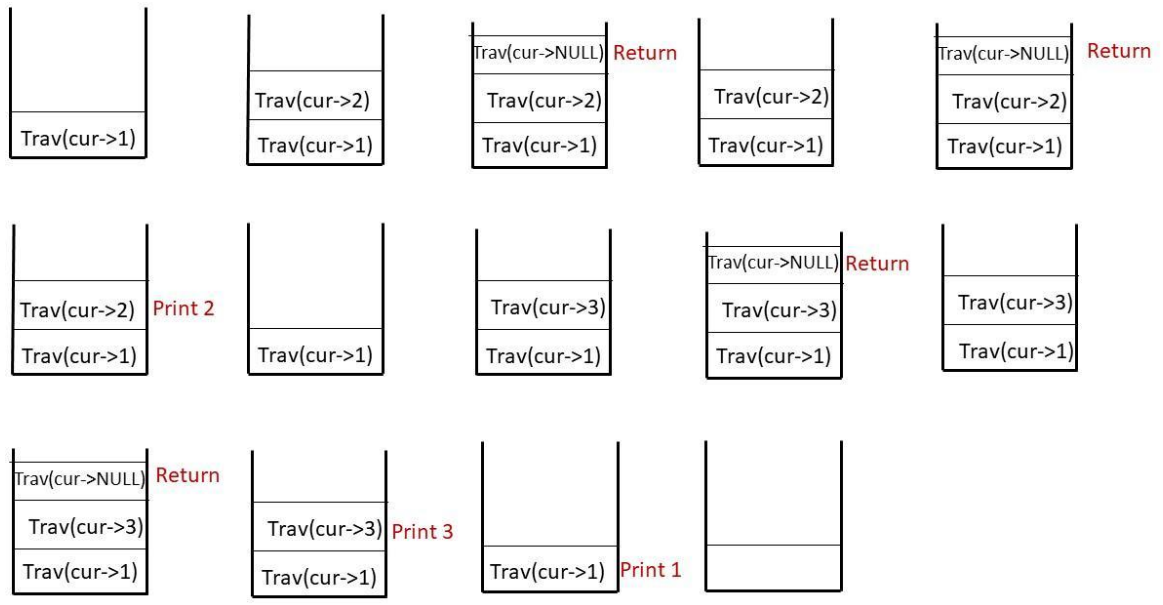
Explanation: It is very important to understand how recursion works behind the scenes to traverse the tree. For it we will see a simple case:



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:

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

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

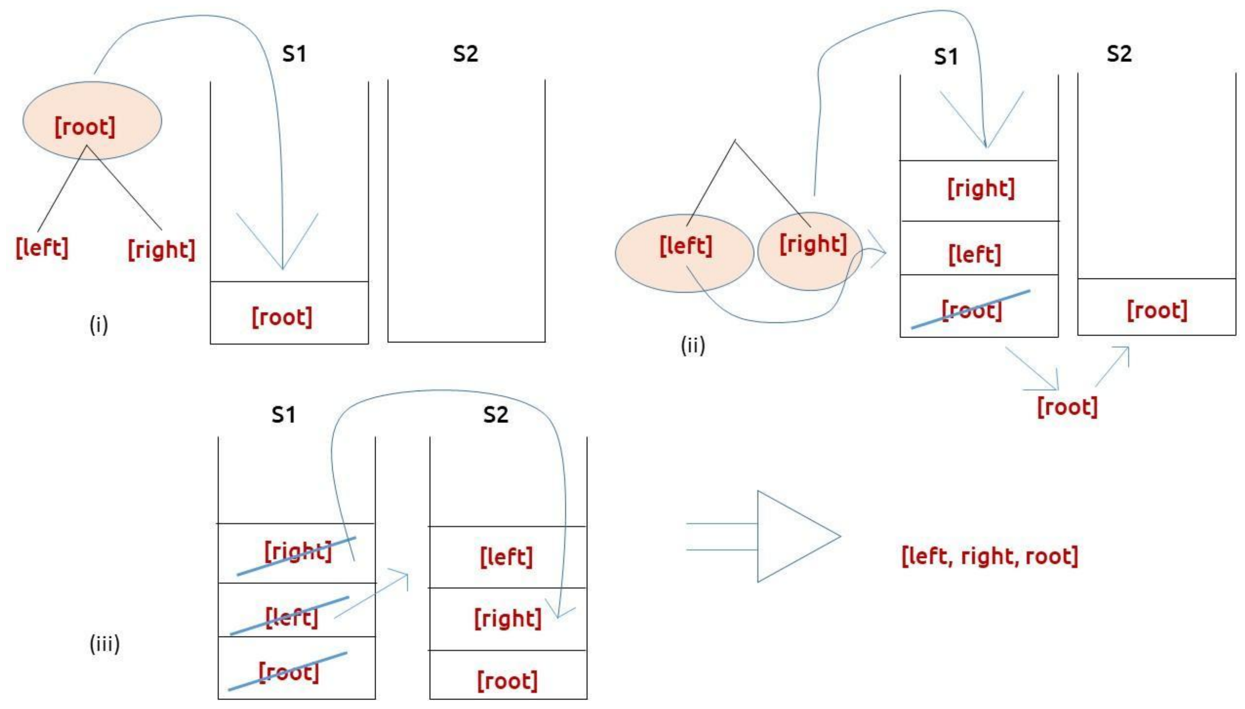
```
void postOrderTrav(node * curr, vector < int > & postOrder) {
  if (curr == NULL)
    return;
  postOrderTrav(curr -> left, postOrder);
  postOrderTrav(curr -> right, postOrder);
  postOrder.push_back(curr -> data);
}
```

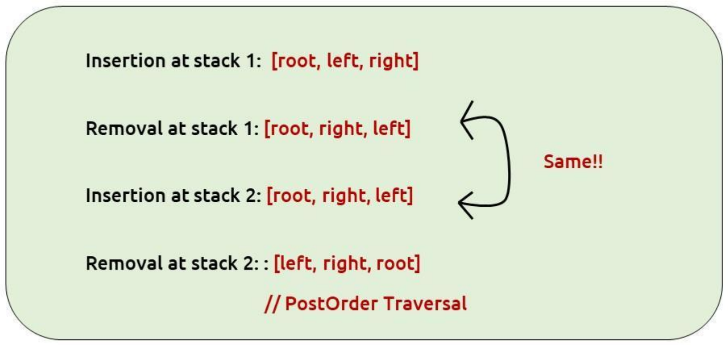
Output:
The postOrder Traversal is : 4 8 5 2 6 9 10 7 3 1
Time Complexity: O(N).
Reason: We are traversing N nodes and every node is visited exactly once.
Space Complexity: O(N)
Reason: Space is needed for the recursion stack. In the worst case (skewed tree), space complexity can be O(N).

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.




Insertion at stack 1: [root, left, right]

Removal at stack 1: [root, right, left]

Insertion at stack 2: [root, right, left]

Removal at stack 2: : [left, right, root]

// PostOrder Traversal

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

Code:

```
#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;
}
```

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.

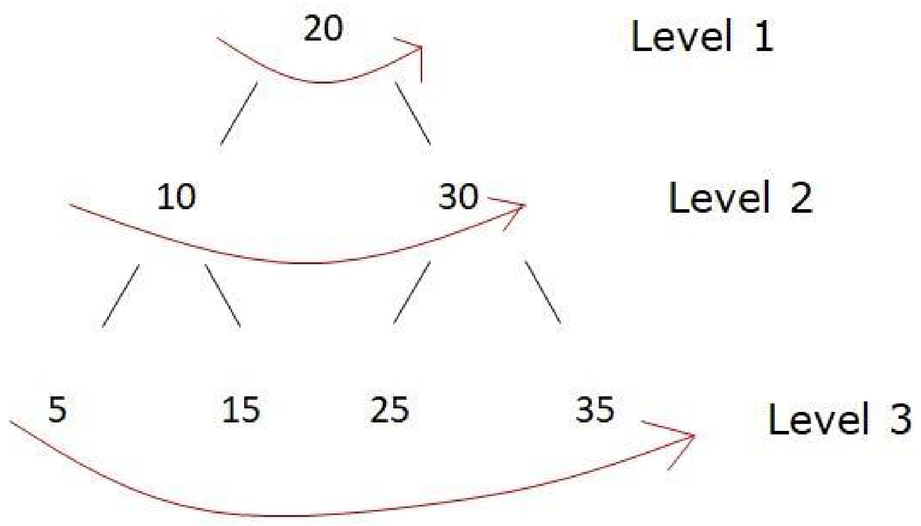
Code:

```
#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;

}
```

Output:
The postOrder Traversal is : 4 8 5 2 6 9 10 7 3 1
Time Complexity: O(N).
Space Complexity: O(N)

Level Order Traversal of a Binary Tree

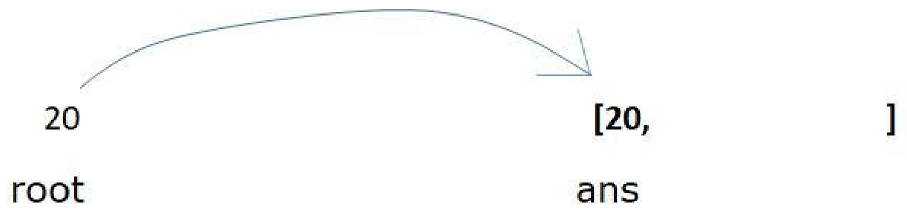
Problem Statement: Level order traversal of a binary tree. Given the root node of the tree and you have to print the value of the level of the node by level.
Example 1:



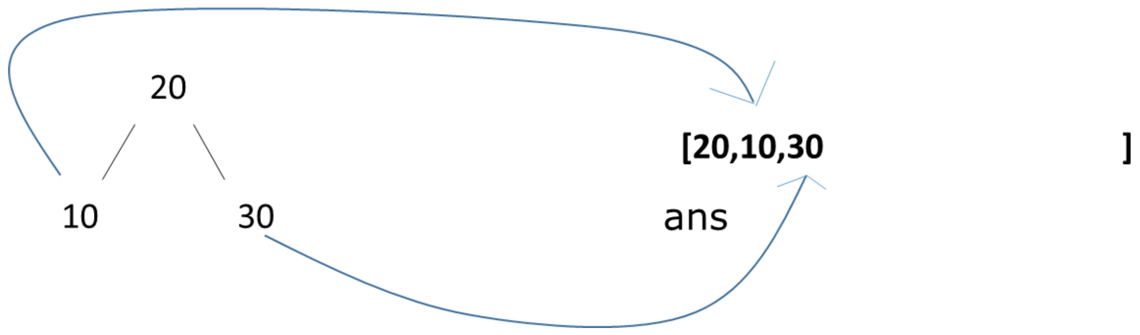
Output:
```
20 10 30 5 15 25 35
```
We will print the nodes of the first level (20), then we will print nodes of second level(10,30) and at last we will print nodes of the last level(5,15,25,35)

Solution:

Let us consider the example 1 given above.
We need to print a level by level traversal. Let us say, we store all the levels in a data structure named 'ans'. We can use a vector<int> or List<int> to store our answer.
Now we need to store the first level to 'ans'. In any given binary tree, the first level will always be the root node, so we can easily store it.



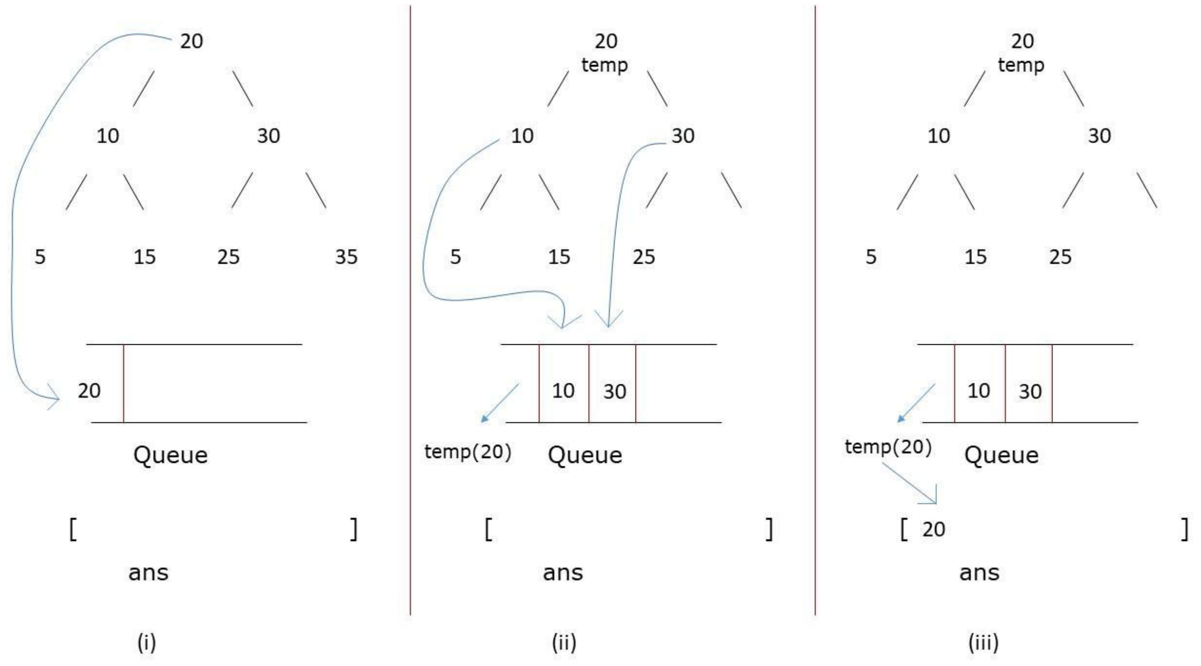
Next, we want to store the second level, this can be done by pushing the left child of root and then the right child of the root to our 'ans' data structure.



Now what about the next level? We just can't keep writing root→left→left / root→left→right and so on to reach further levels. We need an additional data structure to store all the nodes of a level. When we are at level 1, we want its left child to be stored first, followed by the right child as the next level and store it in our data structure. Similarly at level 2, we want to access first the left child (which was added first to the data structure), to store the nodes of the next level.
As there is a first-in first-out (FIFO) operation, we will use the **queue** data structure.
Approach:
The algorithm steps are stated as:

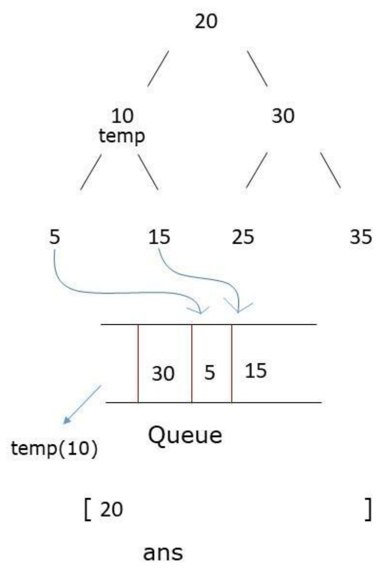
- Take a queue data structure and push the root node to the queue.
- Set a while loop which will run till our queue is non-empty.
- In every iteration, pop out from the front of the queue and assign it to a variable (say temp).
- If temp has a left child, push it to the queue.
- If temp has a right child, push it to the queue.
- At last push the value of the temp node to our "ans" data structure.

Dry Run:
We will discuss example 1.

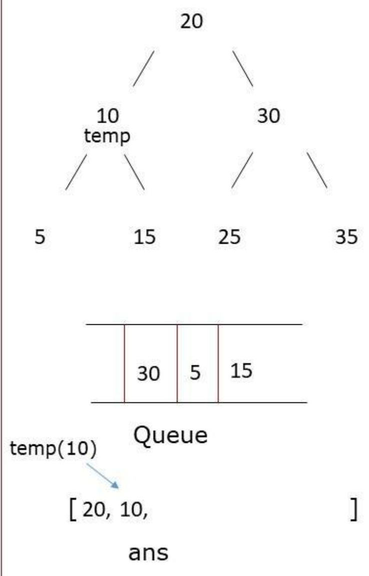


1. First we insert the root node to the queue.
2. We set our while loop and pop its front element as temp Then we inserted the left child followed by the right child of temp to the queue.
3. Then we push the temp value to our ans list.

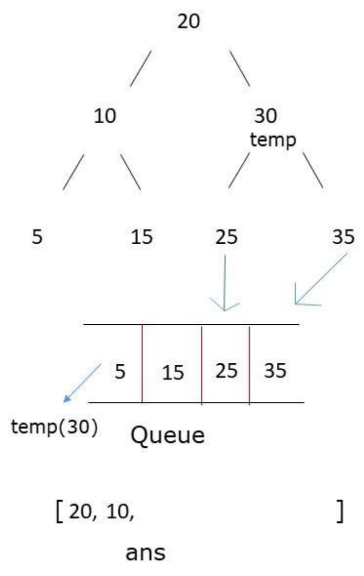
Then we follow these steps for all nodes till the time our queue is non-empty, as shown in the figures below.



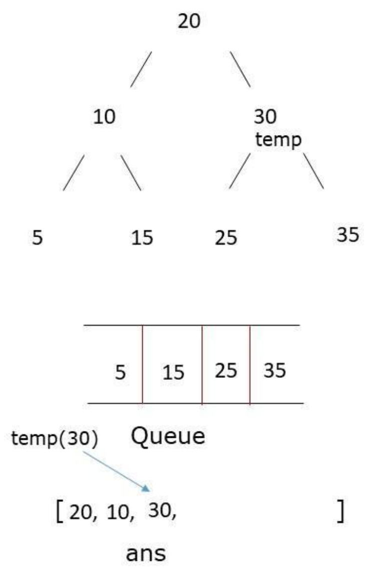
(iv)



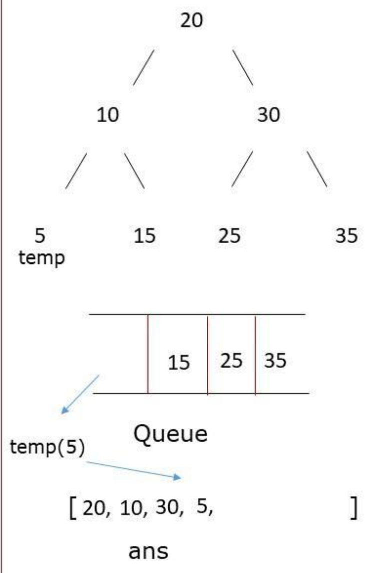
(v)



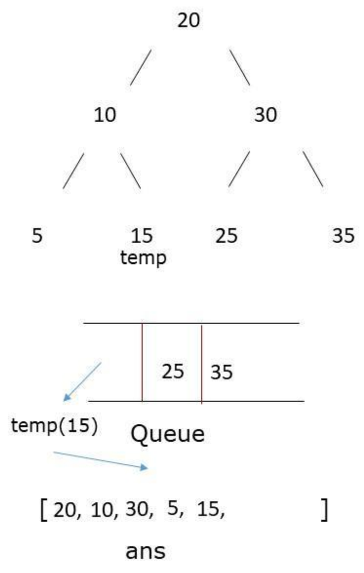
(vi)



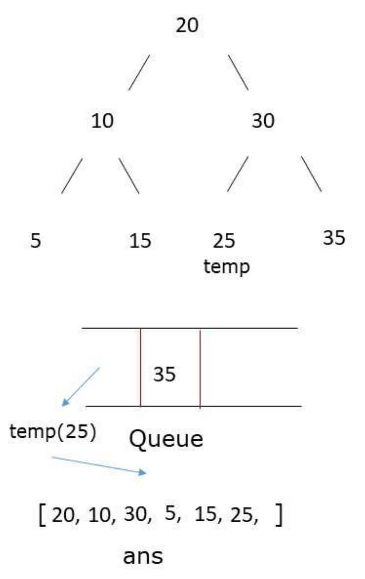
(vii)



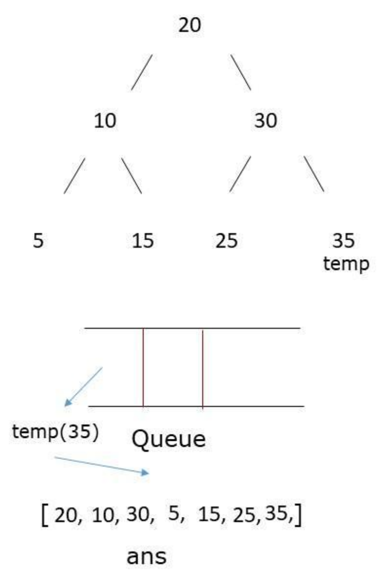
(viii)



(ix)



(x)



(xi)

Code:
```
class Solution {
public:
    vector<int> levelOrder(TreeNode* root) {
        vector<int> ans;
        if(root == NULL)
            return ans;
        queue<TreeNode*> q;
        q.push(root);
        while(!q.empty()) {

            TreeNode *temp = q.front();
            q.pop();

            if(temp->left != NULL)
                q.push(temp->left);
            if(temp->right != NULL)
                q.push(temp->right);
            ans.push_back(temp->val);
        }
        return ans;
    }
};
```
Time Complexity: O(N)
Space Complexity: O(N)

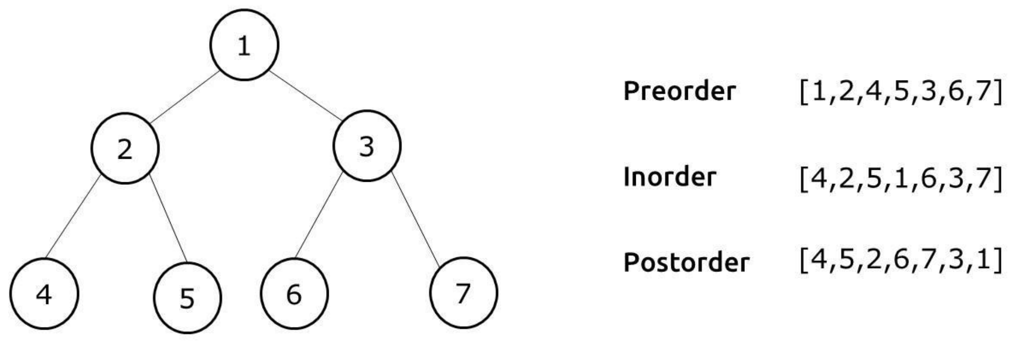
What if we have to print the level numbers as well?
In the above approach we print the nodes level-wise but we can’t differentiate from our ans that whether two nodes are from the same level or not.
To store the level-order traversal along with individual levels stored together ([[20],[10,30],[5,15,25,35]]), we need to make the following changes:

- First we need to declare a 2d array to store our answer(vector<vector<int>> in C++ and List<List<int>> in Java).
- Inside the while loop, first we declare a list to store nodes of a level (say level), then we need to set another for loop, which iterates for the size of the queue and inside this for loop we need to write the logic which we had discussed in the first approach
- The for loop ensures that all the nodes of a particular level are inserted together and when the iteration of the for loop ends, the queue contains the elements of only one level at a time.
- Inside the for loop we push the value of temp to ‘level’.
- After the for loop ends, we push ‘level’ to the answer.

Note: The inner for loop runs just for the size of the queue, for which the while loop would also have to run, therefore introducing the for loop doesn’t increase the time complexity of the program.

Preorder Inorder Postorder Traversals in One Traversal

Problem Statement: Preorder Inorder Postorder Traversals in One Traversal. Write a program to print Preorder, Inorder, and Postorder traversal of the tree in a single traversal.
Example:

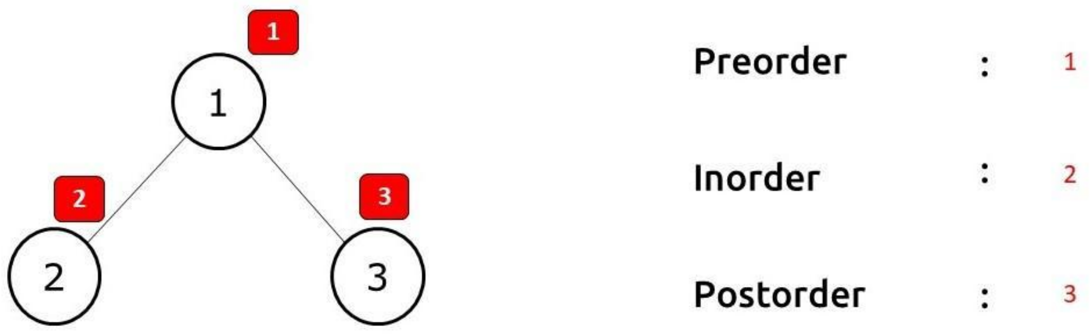


Problem Description:
We need to write a function allTraversals() which returns us three lists of a preorder, inorder and postorder traversals of the tree at the same time. We are not allowed to write separate codes for each traversal. We want all traversals in a single piece of code, in a single instance.

***Pre-req**: Traversals Video, Stack Data Structure*

Solution :

Intuition: If we study the preorder, inorder and postorder traversals, we will observe a pattern. To understand this pattern, we take a simple example:



The number inside the red boxes is the visit when we print the node. In preorder traversal, we print a node at the first visit itself. Whereas, in inorder traversal at the first visit to a node, we simply traverse to the left child. It is only when we return from the left child and visit that node the second time, that we print it. Similarly, in postorder traversal, we print a node in its third visit after visiting both its children.

Approach:
The algorithm steps can be described as follows:
We take a stack data structure and push a pair<val, num> to it. Here Val is the value of the root node and num the visit to the node. Initially, the num is 1 as we are visiting the root node for the first time. We also create three separate lists for preorder, inorder and postorder traversals.
Then we set an iterative loop to run till the time our stack is non-empty.
In every iteration, we pop the top of the stack (say, T). Then we check the second value(num) of T. Three cases can arise:

- **Case 1 : When num==1**

This means that we are visiting the node for the very first time, therefore we push the node value to our preorder list. Then we push the same node with num=2(for Case 2). After this, we want to visit the left child. Therefore we make a new pair Y(<val, num>) and push it to the stack (if there exists a left child). The val of Y is equal to the left child's node value and num is equal to 1.

- **Case 2 : When num==2**

This means that we are visiting the node for the second time, therefore on our second visit to the node, we push the node value to our inorder list. Then we push the same node with num=3(for Case 3). After this, we want to visit the right child. Therefore as in the first case, we check if there exists a right child or not. If there is, we push the right child and num value=1 as a pair to our stack.

- **Case 3 When num==3**

This means that we are visiting the node for the third time. Therefore we will push that node's value to our postorder list. Next, we simply want to return to the parent so we will not push anything else to the stack.
Dry Run: In case you want to watch the dry run for this approach, please watch the video attached below.
Code:
```
#include <bits/stdc++.h>
using namespace std;
struct node {
  int data;
  struct node * left, * right;
};

void allTraversal(node * root, vector < int > & pre, vector < int > &
in , vector < int > & post) {
  stack < pair < node * , int >> st;
  st.push({
    root,
    1
  });
  if (root == NULL) return;

  while (!st.empty()) {
    auto it = st.top();
    st.pop();

    // this is part of pre
    // increment 1 to 2
    // push the left side of the tree
    if (it.second == 1) {
      pre.push_back(it.first -> data);
      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 -> data);
        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 -> data);
      }
    }
  }
```

Output:
The preorder Traversal is: 1 2 4 5 3 6 7
The inorder Traversal is: 4 2 5 1 6 3 7
The postorder Traversal is: 4 5 2 6 7 3 1
Time Complexity: O(N)
Reason: We are visiting every node thrice therefore time complexity will be O(3*N), which can be assumed as linear time complexity.
Space Complexity: O(N)
Reason: We are using three lists and a stack to store the nodes. The time complexity will be about O(4*N), which can be assumed as linear time complexity.