Tree Traversal Implementation

Inorder Traversal: Write a function to perform inorder traversal of a binary tree.

In **inorder traversal**, the nodes of a binary tree are visited in the following order:

1. **Left subtree**
2. **Root**
3. **Right subtree**

Java Implementation (Without Recursion)

Explanation:

- Instead of using recursion, we use an explicit **stack** to track the nodes as we traverse the tree.
- The key idea is:
    1. Go as far left as possible, pushing each node onto the stack.
    2. When you hit a null left child, you pop the top of the stack (which is the current node), visit it, and then move to its right child.
    3. Repeat this process until both the stack is empty and you've visited all nodes.

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

Step-by-Step Inorder Traversal (Without Recursion):

1. **Initialization**:
    - We start at the root node (1).
    - Create an empty stack to keep track of the nodes.
    - Initialize current as the root node (current = root = 1).
2. **First Iteration**:

    - current = 1, it's not null, so push it onto the stack. Now, move to the left child (current = current.left = 2).
    - Stack: [1]
3. **Second Iteration**:
    - current = 2, push it onto the stack, and move to its left child (current = 4).
    - Stack: [1, 2]
4. **Third Iteration**:
    - current = 4, push it onto the stack, and move to its left child, which is null.
    - Stack: [1, 2, 4]
    - Since current = null, we pop from the stack.
5. **Visiting Node 4**:
    - Pop 4 from the stack. Visit 4 (print 4), and move to its right child, which is null.
    - Stack: [1, 2]
    - Output so far: 4
6. **Visiting Node 2**:
    - Since current = null, pop 2 from the stack. Visit 2 (print 2), and move to its right child (current = 5).
    - Stack: [1]
    - Output so far: 4 2
7. **Fourth Iteration**:
    - current = 5, push it onto the stack, and move to its left child, which is null.
    - Stack: [1, 5]
8. **Visiting Node 5**:

    - Since current = null, pop 5 from the stack. Visit 5 (print 5), and move to its right child, which is null.
    - Stack: [1]
    - Output so far: 4 2 5
9. **Visiting Node 1**:
    - Since current = null, pop 1 from the stack. Visit 1 (print 1), and move to its right child (current = 3).
    - Stack: []
    - Output so far: 4 2 5 1
10. **Fifth Iteration**:
    - current = 3, push it onto the stack, and move to its left child, which is null.
    - Stack: [3]
11. **Visiting Node 3**:
    - Since current = null, pop 3 from the stack. Visit 3 (print 3), and move to its right child, which is null.
    - Stack: []
    - Output so far: 4 2 5 1 3
12. **End**:
    - Both the stack is empty, and current = null. The traversal is complete.

Final Output: 4 2 5 1 3

Summary of Steps:

- Move as far left as possible, pushing each node onto the stack.
- When null is reached, pop from the stack, print the node, and move to its right child.
- Continue this process until the entire tree is traversed.

The output follows **inorder traversal**: Left subtree -> Root -> Right subtree.

Java Code:

```
import java.util.Stack;
class Node {
   int data;
   Node left, right;
   public Node(int item) {
      data = item;
      left = right = null;
   }
}
public class BinaryTree {
   Node root;

   // Function for inorder traversal without recursion
   void inorderTraversal() {
      if (root == null) {
         return;
      }

      Stack<Node> stack = new Stack<>();
      Node current = root;

      // Traverse the tree
      while (current != null || !stack.isEmpty()) {
         // Reach the leftmost node of the current node
         while (current != null) {
            stack.push(current);
            current = current.left;
         }

         // Current must be null at this point, so pop the stack
         current = stack.pop();

         // Visit the node
         System.out.print(current.data + " ");
```

```
            // Now visit the right subtree
            current = current.right;
        }
    }

    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();

        // Creating the binary tree
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        System.out.println("Inorder Traversal of the binary tree is: ");
        tree.inorderTraversal();
    }
}
```

Java implementation (with recursion)

```
class Node {
    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
}

public class BinaryTree {
    Node root;

    // Function for inorder traversal
```

```
    void inorderTraversal(Node node) {
        if (node == null)
            return;

        // Traverse the left subtree
        inorderTraversal(node.left);

        // Visit the root node
        System.out.print(node.data + " ");

        // Traverse the right subtree
        inorderTraversal(node.right);
    }

    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();

        // Creating the binary tree
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        System.out.println("Inorder Traversal of the binary tree is: ");
        tree.inorderTraversal(tree.root);
    }
}
```

Preorder Traversal: Write a function to perform preorder traversal of a binary tree.

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

In **preorder traversal**, the order of visiting nodes is:

1. **Root**
2. **Left subtree**
3. **Right subtree**

Code to Perform Preorder Traversal (Without Recursion)

Here's how to write a **preorder traversal** without recursion in Java, using a stack:

Java Code (Preorder Traversal Without Recursion):

Explanation:

- In preorder traversal, the root node is visited first, then the left subtree, and finally the right subtree.
- Instead of recursion, we use a stack:
    1. Push the root node onto the stack.
    2. Pop the stack, visit the node, then push its right child (if it exists) and then its left child (if it exists).
    3. Continue until the stack is empty.

```
import java.util.Stack;

class Node {
   int data;
   Node left, right;

   public Node(int item) {
      data = item;
      left = right = null;
   }
}

public class BinaryTree {
```

```java
    Node root;

    // Function for preorder traversal without recursion
    void preorderTraversal() {
        if (root == null) {
            return;
        }

        Stack<Node> stack = new Stack<>();
        stack.push(root);

        while (!stack.isEmpty()) {
            // Pop the current node from the stack and print it
            Node current = stack.pop();
            System.out.print(current.data + " ");

            // Push right child first (so that the left child is processed first)
            if (current.right != null) {
                stack.push(current.right);
            }

            // Push left child to stack
            if (current.left != null) {
                stack.push(current.left);
            }
        }
    }

    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();

        // Creating the binary tree
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        System.out.println("Preorder Traversal of the binary tree is: ");
        tree.preorderTraversal();
    }
}
```

Output:
Preorder Traversal of the binary tree is: 1 2 4 5 3

Java Implementation (with recursion)

```
class Node {
   int data;
   Node left, right;

   public Node(int item) {
      data = item;
      left = right = null;
   }
}

public class BinaryTree {
   Node root;

   // Function for preorder traversal with recursion
   void preorderTraversal(Node node) {
      if (node == null) {
         return;
      }

      // Visit the root node
      System.out.print(node.data + " ");

      // Recursively traverse the left subtree
      preorderTraversal(node.left);

      // Recursively traverse the right subtree
      preorderTraversal(node.right);
   }

   public static void main(String[] args) {
      BinaryTree tree = new BinaryTree();

      // Creating the binary tree
      tree.root = new Node(1);
      tree.root.left = new Node(2);
```

```
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        System.out.println("Preorder Traversal of the binary tree is: ");
        tree.preorderTraversal(tree.root);
    }
}
```

Output:
Preorder Traversal of the binary tree is: 1 2 4 5 3

Postorder Traversal: Write a function to perform postorder traversal of a binary tree.

In **postorder traversal**, the order of visiting nodes is:

1. **Left subtree**
2. **Right subtree**
3. **Root**

Step-by-Step Postorder Traversal (Without Recursion):

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

Initialization:

- Start at the root node (1).
- Create two stacks, stack1 and stack2.
- Push the root node onto stack1.
- Initial state:
    - Stack1: [1]
    - Stack2: []

First Iteration:

- Pop 1 from stack1 and push it onto stack2. Push its left child (2) and right child (3) onto stack1.
- Stack1: [2, 3]
- Stack2: [1]

Second Iteration:

- Pop 3 from stack1 and push it onto stack2. 3 has no children, so nothing is pushed onto stack1.
- Stack1: [2]
- Stack2: [1, 3]

Third Iteration:

- Pop 2 from stack1 and push it onto stack2. Push its left child (4) and right child (5) onto stack1.
- Stack1: [4, 5]
- Stack2: [1, 3, 2]

Fourth Iteration:

- Pop 5 from stack1 and push it onto stack2. 5 has no children, so nothing is pushed onto stack1.
- Stack1: [4]
- Stack2: [1, 3, 2, 5]

Fifth Iteration:

- Pop 4 from stack1 and push it onto stack2. 4 has no children, so nothing is pushed onto stack1.
- Stack1: []
- Stack2: [1, 3, 2, 5, 4]

End:

- stack1 is empty, and all nodes are in stack2 in reverse postorder. We pop nodes from stack2 and print them.
- Output: 4 5 2 3 1

Java Code (Postorder Traversal Without Recursion - Using Two Stacks):

```
import java.util.Stack;

class Node {
   int data;
   Node left, right;
```

```java
    public Node(int item) {
        data = item;
        left = right = null;
    }
}

public class BinaryTree {
    Node root;

    // Function for postorder traversal without recursion (using two stacks)
    void postorderTraversal() {
        if (root == null) {
            return;
        }

        Stack<Node> stack1 = new Stack<>();
        Stack<Node> stack2 = new Stack<>();

        // Push the root node to stack1
        stack1.push(root);

        // Run while stack1 is not empty
        while (!stack1.isEmpty()) {
            // Pop from stack1 and push the node to stack2
            Node node = stack1.pop();
            stack2.push(node);

            // Push the left and right children of the popped node to stack1
            if (node.left != null) {
                stack1.push(node.left);
            }
            if (node.right != null) {
                stack1.push(node.right);
            }
        }
```

```
        // Now, stack2 will contain nodes in postorder traversal, so pop from stack2
        while (!stack2.isEmpty()) {
            Node node = stack2.pop();
            System.out.print(node.data + " ");
        }
    }

    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();

        // Creating the binary tree
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        System.out.println("Postorder Traversal of the binary tree is: ");
        tree.postorderTraversal();
    }
}
```

Output:
Postorder Traversal of the binary tree is: 4 5 2 3 1

Java Code (Postorder Traversal with Recursion):

```
class Node {
    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
```

```
}

public class BinaryTree {
    Node root;

    // Function for postorder traversal with recursion
    void postorderTraversal(Node node) {
        if (node == null) {
            return;
        }

        // Recursively traverse the left subtree
        postorderTraversal(node.left);

        // Recursively traverse the right subtree
        postorderTraversal(node.right);

        // Visit the root node (after left and right subtrees)
        System.out.print(node.data + " ");
    }

    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();

        // Creating the binary tree
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        System.out.println("Postorder Traversal of the binary tree is: ");
        tree.postorderTraversal(tree.root);
    }
}
```

Output:

Postorder Traversal of the binary tree is: 4 5 2 3 1