Implementing a Tree using Linked List in Java and C++

To implement a tree using a linked list, the most common type of tree is a **binary tree** where each node has at most two children: left and right.

Steps:

1. **Define the Node Structure**:
   - A node will typically consist of:
     - A data field (to store the value)
     - A reference/pointer to the left child
     - A reference/pointer to the right child
2. **Create the Tree Structure**:
   - The tree class typically contains a reference/pointer to the root node and functions to manipulate the tree (like inserting, traversing, etc.).
3. **Insertion/Traversal**:
   - Functions to insert nodes, traverse the tree (e.g., in-order, pre-order, post-order), and other operations.

Java Implementation

Step 1: Define the Node Class

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

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

Step 2: Define the Binary Tree Class

```
class BinaryTree {
    Node root;
    public BinaryTree() {
        root = null;
    }
    // Insert a node in a binary tree (basic example)
    public void insert(int data) {
        root = insertRec(root, data);
    }
    private Node insertRec(Node root, int data) {
        if (root == null) {
            root = new Node(data);
            return root;
        }

        if (data < root.data) {
            root.left = insertRec(root.left, data);
        } else if (data > root.data) {
            root.right = insertRec(root.right, data);
        }

        return root;
    }
    // In-order Traversal (Left, Root, Right)
    public void inorderTraversal(Node node) {
        if (node != null) {
            inorderTraversal(node.left);
            System.out.print(node.data + " ");
            inorderTraversal(node.right);
        }
    }
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
```

```java
        tree.insert(50);
        tree.insert(30);
        tree.insert(70);
        tree.insert(20);
        tree.insert(40);
        tree.insert(60);
        tree.insert(80);
        System.out.println("In-order Traversal of the Tree:");
        tree.inorderTraversal(tree.root);
    }
}
```

C++ Implementation:

Step 1: Define the Node Structure

```cpp
#include <iostream>
using namespace std;

class Node {
public:
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = right = nullptr;
    }
};
```

Step 2: Define the Binary Tree Class

```cpp
class BinaryTree {
public:
    Node* root;

    BinaryTree() {
```

```
        root = nullptr;
    }
    // Insert a node in a binary tree (basic example)
    Node* insert(Node* node, int data) {
        if (node == nullptr) {
            return new Node(data);
        }

        if (data < node->data) {
            node->left = insert(node->left, data);
        } else if (data > node->data) {
            node->right = insert(node->right, data);
        }

        return node;
    }
    // In-order Traversal (Left, Root, Right)
    void inorderTraversal(Node* node) {
        if (node != nullptr) {
            inorderTraversal(node->left);
            cout << node->data << " ";
            inorderTraversal(node->right);
        }
    }
};

int main() {
    BinaryTree tree;
    tree.root = tree.insert(tree.root, 50);
    tree.insert(tree.root, 30);
    tree.insert(tree.root, 70);
    tree.insert(tree.root, 20);
    tree.insert(tree.root, 40);
    tree.insert(tree.root, 60);
    tree.insert(tree.root, 80);
    cout << "In-order Traversal of the Tree:" << endl;
    tree.inorderTraversal(tree.root);
    return 0;
}
```

Implementing a Binary Search Tree (BST) in Java and C++

A **Binary Search Tree (BST)** is a binary tree in which for each node:

- The value of all nodes in the left subtree is less than the node's value.
- The value of all nodes in the right subtree is greater than the node's value.

This structure allows for efficient searching, insertion, and deletion operations.

Steps:

1. **Define the Node Structure**:
    - Each node consists of:

        - A data field (to store the value)
        - A reference/pointer to the left child
        - A reference/pointer to the right child
2. **Define the BST Operations**:
    - Insertion: Add a new node while maintaining the BST property.
    - Search: Find a node with a given value.
    - Traversal: In-order traversal is most commonly used to print nodes in ascending order.
    - Deletion (optional but important): Remove a node from the tree while maintaining the BST property.

Java Implementation

Step 1: Define the Node Class

```
class Node {
  int data;
  Node left, right;
  public Node(int data) {
    this.data = data;
    left = right = null;
  }
}
```

Step 2: Define the Binary Search Tree (BST) Class

```
class BinarySearchTree {
  Node root;

  public BinarySearchTree() {
    root = null;
  }

  // Insert a node into the BST
  public void insert(int data) {
    root = insertRec(root, data);
  }

  // Recursive function to insert a new node
  private Node insertRec(Node root, int data) {
    if (root == null) {
      root = new Node(data);
      return root;
    }

    if (data < root.data) {
      root.left = insertRec(root.left, data);
    } else if (data > root.data) {
      root.right = insertRec(root.right, data);
    }
```

```
        return root;
    }

    // Search for a node in the BST
    public boolean search(int data) {
        return searchRec(root, data);
    }

    private boolean searchRec(Node root, int data) {
        if (root == null) {
            return false;
        }
        if (root.data == data) {
            return true;
        }
        if (data < root.data) {
            return searchRec(root.left, data);
        }
        return searchRec(root.right, data);
    }

    // In-order Traversal (Left, Root, Right)
    public void inorderTraversal(Node node) {
        if (node != null) {
            inorderTraversal(node.left);
            System.out.print(node.data + " ");
            inorderTraversal(node.right);
        }
    }

    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();

        bst.insert(50);
        bst.insert(30);
        bst.insert(70);
        bst.insert(20);
        bst.insert(40);
        bst.insert(60);
        bst.insert(80);

        System.out.println("In-order Traversal of the Tree:");
        bst.inorderTraversal(bst.root);

        System.out.println("\n\nSearch for 60: " + bst.search(60)); // Output: true
        System.out.println("Search for 90: " + bst.search(90)); // Output: false
    }
}
```