**Data Structure Used:**

The data structure used in this code is a **binary tree**. Specifically, each Node represents a tree node, which contains:

* An integer value (value) or an operation (operation for arithmetic operations like +, -, \*, /).
* Two references (leftChild and rightChild) to its left and right child nodes, respectively, forming the binary tree structure.

The tree represents arithmetic expressions, where each internal node represents an operator and each leaf node represents an operand (either a value or another expression).

**Algorithms Used:**

The following algorithms are implemented for various operations on the binary tree:

1. **Counting Elements (Nodes):**
   * **Algorithm**: A recursive depth-first traversal of the tree. Starting from the root, count the current node and recursively count the nodes in the left and right subtrees.
   * **Time Complexity**: **O(n)**, where n is the number of nodes in the tree. Each node is visited once.
2. **Computing Height of the Tree:**
   * **Algorithm**: A recursive algorithm that calculates the height of the tree. The height of a node is defined as 1 + the maximum height of its left and right subtrees.
   * **Time Complexity**: **O(n)**, where n is the number of nodes. Each node is visited once.
3. **Counting Leaves (Leaf Nodes):**
   * **Algorithm**: A recursive depth-first traversal that counts nodes that have no children (leaf nodes).
   * **Time Complexity**: **O(n)**, where n is the number of nodes. Each node is checked once to determine if it's a leaf.
4. **Checking if the Tree is Fully Balanced:**
   * **Algorithm**: A recursive algorithm that checks if the tree is balanced by ensuring that for every node, the heights of its left and right subtrees differ by no more than 1.
   * **Time Complexity**: **O(n)**, where n is the number of nodes. Each node is visited once, and the height of each subtree is calculated.
5. **Checking if Two Trees are Identical:**
   * **Algorithm**: A recursive comparison of two trees. For each node, check if both the value and operation (if applicable) are the same for both trees and then recursively check the left and right subtrees.
   * **Time Complexity**: **O(n)**, where n is the number of nodes in the smaller tree. In the worst case, it compares all nodes in both trees.
6. **Prefix, Infix, and Postfix Traversals:**
   * **Algorithm**: These are standard tree traversal algorithms:
     + **Prefix**: Visit the node first, then recursively visit the left and right children.
     + **Infix**: Recursively visit the left child, then the node, and then the right child.
     + **Postfix**: Recursively visit the left and right children first, then the node.
   * **Time Complexity**: **O(n)**, where n is the number of nodes in the tree. Each node is visited once.
7. **Evaluating the Expression:**
   * **Algorithm**: A recursive evaluation where for each node, if it's an operator, the algorithm evaluates its left and right subtrees and applies the operation. If it's a leaf, it returns the node’s value.
   * **Time Complexity**: **O(n)**, where n is the number of nodes. Each node is visited once.
8. **Displaying the Tree:**
   * **Algorithm**: A recursive depth-first traversal with indentation to show the tree structure visually. The tree is printed top-down, with indentation representing the depth of each node.
   * **Time Complexity**: **O(n)**, where n is the number of nodes. Each node is visited once for printing.

**Time Complexity Summary:**

* **Count Elements**: O(n)
* **Compute Height**: O(n)
* **Count Leaves**: O(n)
* **Check Balance**: O(n)
* **Is Identical**: O(n)
* **Prefix, Infix, Postfix Traversals**: O(n)
* **Evaluate Expression**: O(n)
* **Show Tree**: O(n)