TREES

Trees are a fundamental data structure in computer science, commonly used to represent hierarchical relationships. This unit explores the characteristics of trees, specifically **binary trees**, traversal methods, representation, and applications, including **binary search trees** and **height-balanced trees** (AVL trees).

1. Introduction to Trees

A **tree** is a non-linear data structure consisting of nodes connected by edges. It has the following key characteristics:

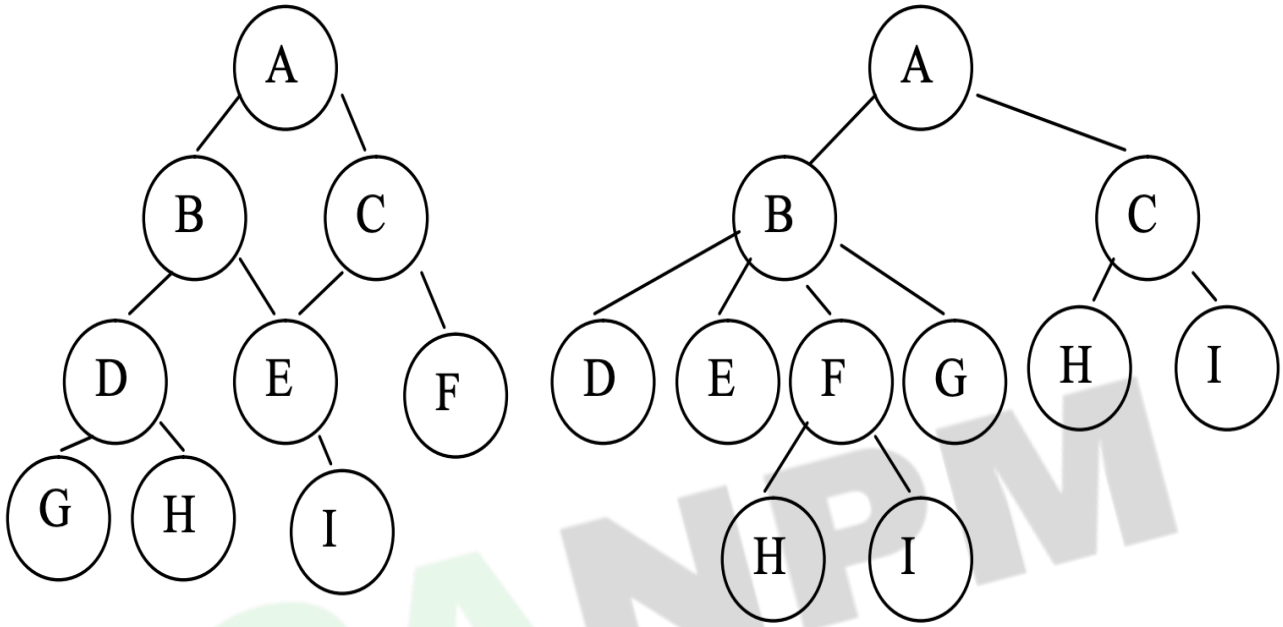
- **Root**: The topmost node of the tree.
- **Nodes**: Elements of the tree that contain data.
- **Edges**: Connections between nodes.
- **Leaf Nodes**: Nodes with no children.
- **Subtree**: Any node and its descendants form a subtree.

2. Binary Trees

A **binary tree** is a type of tree in which each node has at most two children, referred to as the left child and the right child. The properties of binary trees include:

- Each node has a maximum of two children.

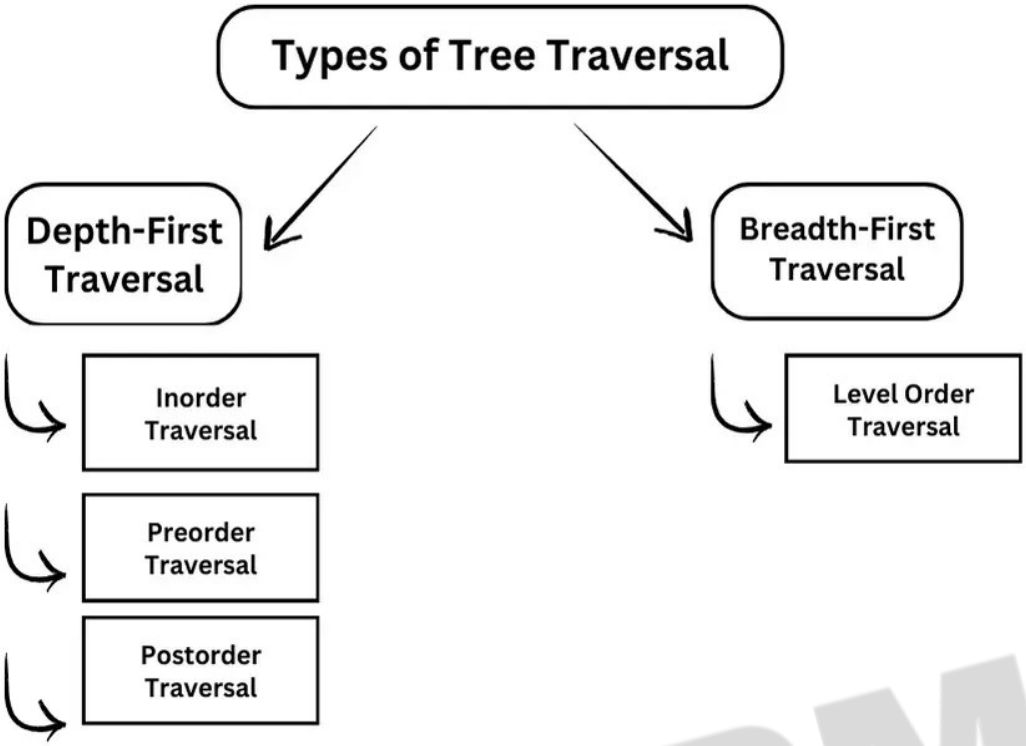
- The left child of a node must contain values less than or equal to its parent node, and the right child must contain values greater than its parent (for binary search trees).



3. Traversal Methods

Traversal is the process of visiting all the nodes in a tree.

There are several methods to traverse a binary tree, each serving different purposes. The primary traversal methods include:



a) Pre-order Traversal

In **pre-order traversal**, the nodes are recursively visited in the following order:

1. Visit the root node.
2. Traverse the left subtree.
3. Traverse the right subtree.

This method is useful for creating a copy of the tree or getting a prefix expression in expression trees.

b) In-order Traversal

In **in-order traversal**, the nodes are recursively visited in the following order:

1. Traverse the left subtree.

2. Visit the root node.

3. Traverse the right subtree.

This method is particularly important for binary search trees as it visits nodes in non-decreasing order, allowing easy retrieval of sorted values.

c) Post-order Traversal

In **post-order traversal**, the nodes are recursively visited in the following order:

1. Traverse the left subtree.

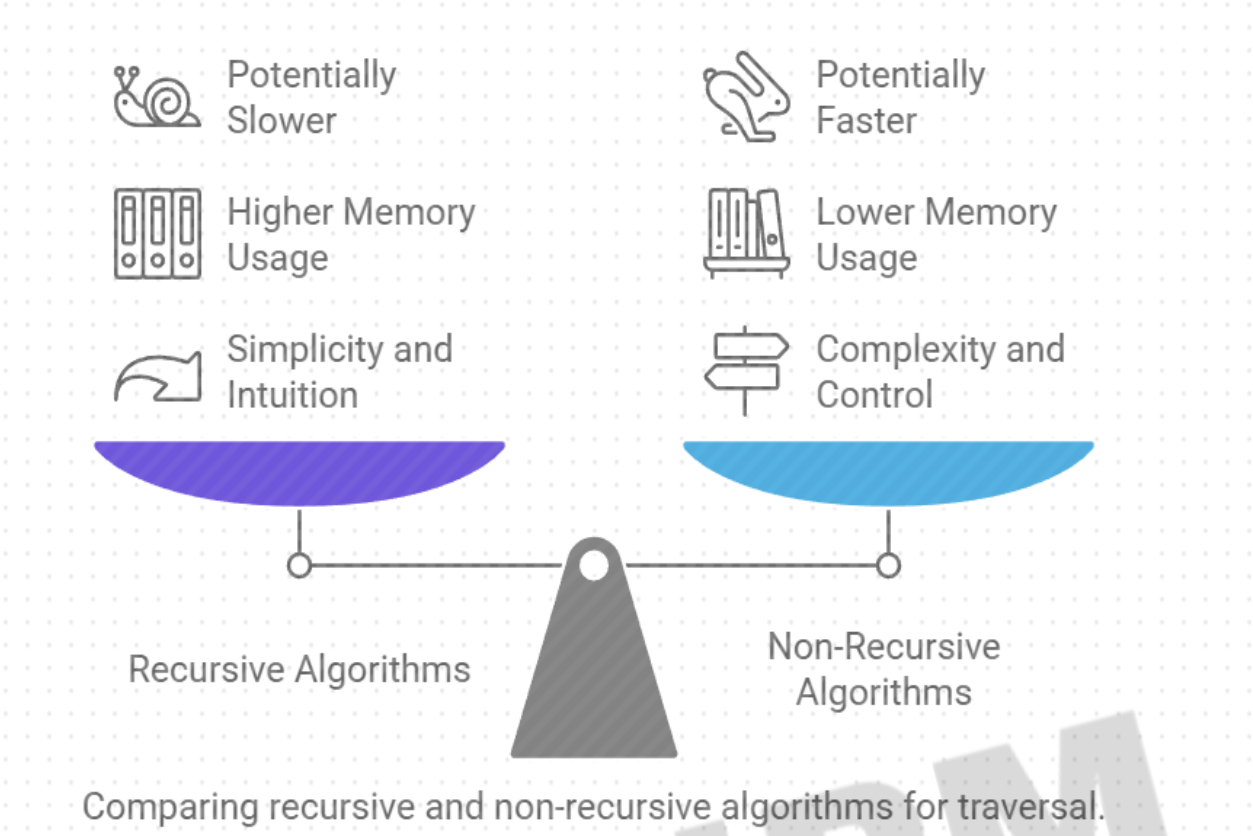
2. Traverse the right subtree.

3. Visit the root node.

This method is used in applications like deleting a tree or evaluating postfix expressions.

4. Recursive and Non-Recursive Algorithms

Traversal can be implemented using both recursive and non-recursive methods. Below is a discussion on both approaches for each traversal method.



Comparing recursive and non-recursive algorithms for traversal.

a) Recursive Algorithm

Recursive algorithms for tree traversal rely on function calls to visit nodes. For example:

1. **Pre-order Traversal (Recursive)**:
    - Visit the node and then call the function recursively for the left and right children.
2. **In-order Traversal (Recursive)**:
    - Recursively call the function for the left child, visit the node, and then call it for the right child.
3. **Post-order Traversal (Recursive)**:
    - Recursively call the function for the left and right children before visiting the node.

b) Non-Recursive Algorithm

Non-recursive algorithms typically use a stack to simulate the behavior of recursion. The main advantage is that they avoid the overhead associated with recursive function calls.

1. **Pre-order Traversal (Non-Recursive)**:
    - Use a stack to keep track of nodes to visit. Push the root, then repeatedly pop and visit nodes, pushing their right and left children onto the stack.
2. **In-order Traversal (Non-Recursive)**:
    - Use a stack to traverse left as far as possible, visiting nodes as you backtrack (pop) to the right.
3. **Post-order Traversal (Non-Recursive)**:
    - Post-order traversal is slightly more complex non-recursively and can be achieved using two stacks or a single stack with an auxiliary marker to distinguish between node visits.

5. Representation of Trees

Trees can be represented in various ways, depending on the application and specific requirements:

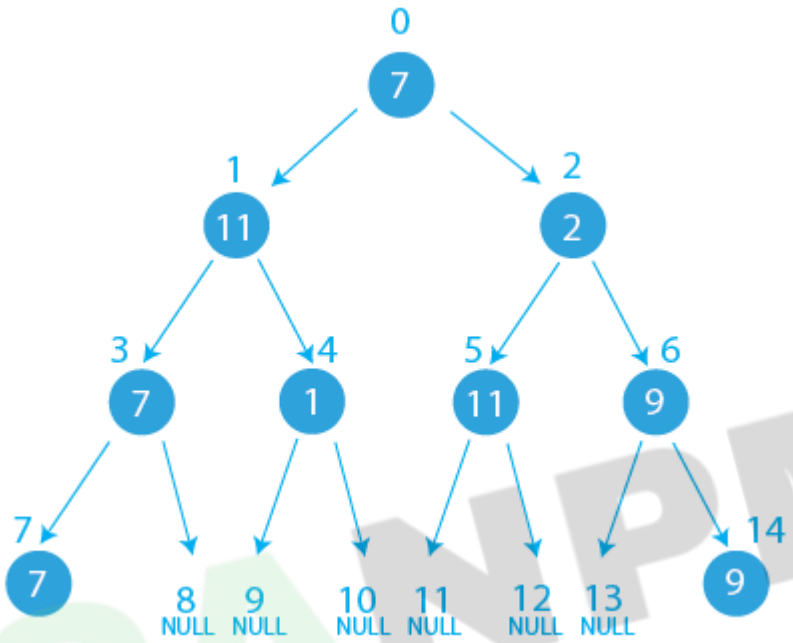
a) Array Representation

In an array representation of a binary tree, each node is stored in an array where:

- The root is at index 0.

- The left child of a node at index i is at index 2 i + 1.
- The right child is at index 2 i + 2.

This method is efficient for complete or nearly complete binary trees but can waste space for sparse trees.

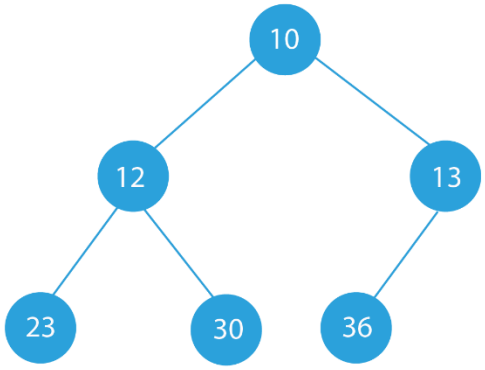


b) Linked Representation

In a linked representation, each node is a structure containing:

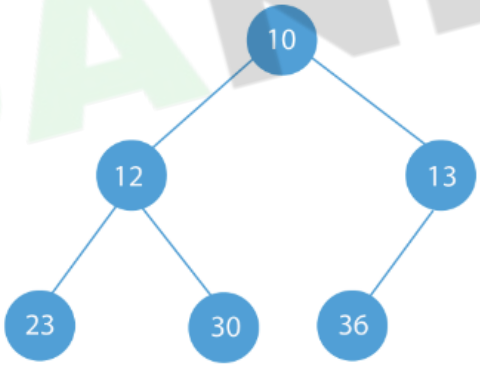
- Data.
- A pointer/reference to the left child.
- A pointer/reference to the right child.

This representation is more flexible and can efficiently handle sparse trees, as memory is allocated dynamically.



Input:

Output:



6. Applications of Trees

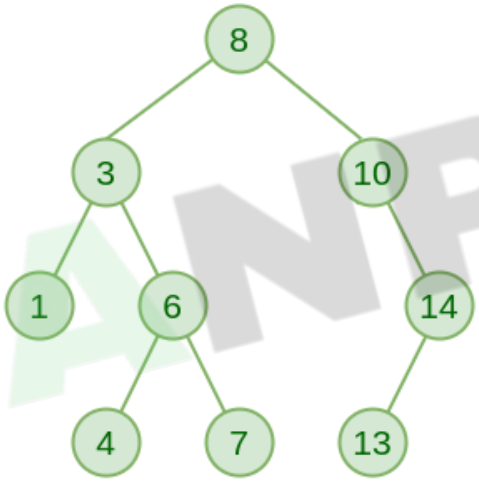
Trees have numerous applications in computer science, including:

a) Binary Search Trees (BST)

A **binary search tree** is a binary tree with the following properties:

- The left subtree of a node contains only nodes with values less than or equal to the node's value.
- The right subtree contains only nodes with values greater than the node's value.

This structure allows for efficient searching, insertion, and deletion operations, typically performed in $O(\log n)$ time for balanced trees.



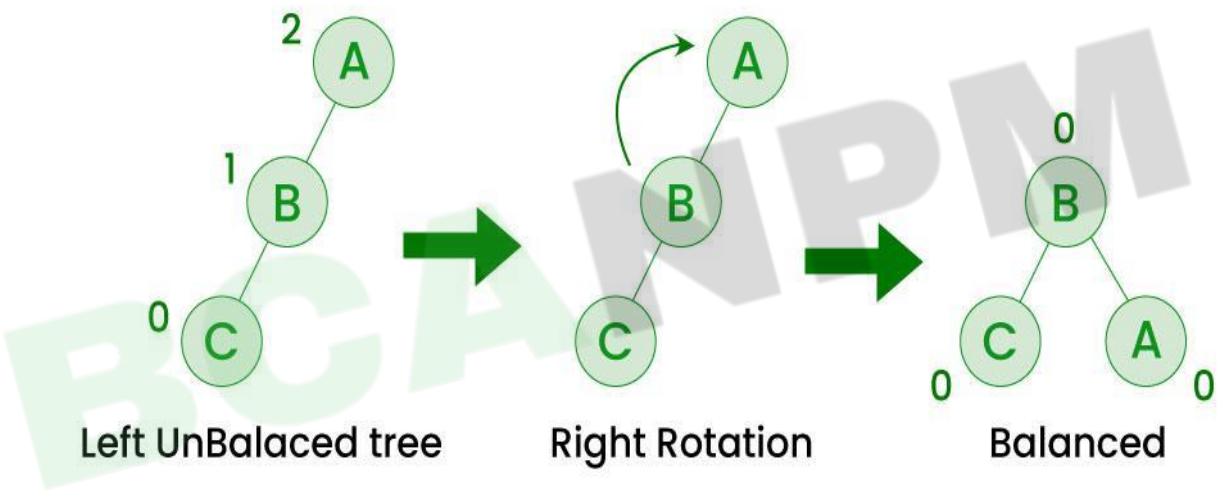
Binary Search Tree

b) Height-Balanced Trees (AVL Trees)

An **AVL tree** is a type of self-balancing binary search tree where the difference in height between the left and right subtrees of any node (called the balance factor) is at most 1. This ensures that the tree remains balanced, providing efficient operations.

Key Features of AVL Trees:

- **Height-Balancing**: Maintains the balance factor during insertions and deletions through rotations.
- **Rotations**: Single and double rotations are used to rebalance the tree when the balance factor exceeds the allowed range.
- **Efficiency**: Search, insertion, and deletion operations are performed in O(log n) time due to the balanced nature of the tree.



Conclusion

Trees, particularly binary trees, are vital data structures that provide efficient ways to store and manage hierarchical data. Understanding traversal methods, representation techniques, and specialized forms like binary search trees and AVL trees equips students with the necessary tools for efficient algorithm development and problem-solving in computer science. The applications of trees extend across various domains, including databases, file systems, and

artificial intelligence, making them an essential topic in a BCA curriculum.