Module IV:
Tree Terminologies

Trees are fundamental data structures in computer science, used to represent hierarchical relationships between elements. Unlike linear data structures like arrays or linked lists, where elements are arranged sequentially, trees organize data in a parent-child fashion. Understanding the basic terminologies is crucial for comprehending tree operations and their applications.

At the very top of a tree is the **root**. The root is the starting point of the tree and does not have any parent. Every other element in the tree can be traced back to the root. For example, in a file system, the root directory is the starting point from which all other files and folders branch out.

Each element in a tree is called a **node**. Nodes store the actual data. A node can have connections to other nodes. The connections between nodes are called **edges**. Edges represent the relationships between the elements.

A node that has one or more child nodes is called a **parent**. Conversely, a node that has a parent is called a **child**. For instance, if node A connects to node B and node C, then A is the parent of B and C, and B and C are the children of A. Nodes that share the same parent are called **siblings**.

Nodes that do not have any children are called **leaves** or **external nodes**. These are the terminal nodes of the tree. In contrast, nodes that have at least one child are called **internal nodes**.

The **height** of a tree is the length of the longest path from the root to a leaf node. It's often defined as the maximum number of edges on the path from the root to any leaf. The height of a single-node tree (just the root) is usually considered 0. The **depth** of a node, on the other hand, is the length of the path from the root to that specific node. The root node has a depth of 0.

A **subtree** is a portion of a tree that can itself be considered a tree. Any node in a tree, along with all its descendants, forms a subtree. For example, if a node has two children, each child and its subsequent descendants form a subtree.

The **degree** of a node is the number of children it has. The degree of a tree is the maximum degree of any node in the tree. The **level** of a node refers to its distance from the root. The root is at level 0, its children are at level 1, and so on.

Understanding these terminologies provides a foundational vocabulary for discussing and analyzing various types of trees and their associated algorithms. These concepts are essential for grasping more complex tree structures like binary trees, binary search trees, and AVL trees, which have specific rules regarding the number of children a node can have and how data is organized within the tree. Trees are widely used in various applications, including organizing data in databases, implementing file systems, parsing expressions, and routing algorithms, making these terminologies indispensable for anyone working with data structures.

Binary Trees

A binary tree is a special type of tree data structure where each node has at most two children, referred to as the left child and the right child. This constraint on the number of children makes binary trees simpler to manage and analyze compared to general trees, leading to their widespread use in various

computer science applications. The order of children matters in a binary tree; a left child is distinct from a right child, even if they hold the same value.

The structure of a binary tree is inherently recursive. Each node within a binary tree can be considered the root of its own subtree, with the same properties applying. This recursive nature is often leveraged when designing algorithms for binary trees, such as traversal methods.

Binary trees can be further classified based on their structural properties:

- **Full Binary Tree:** A binary tree is full if every node has either 0 or 2 children. This means there are no nodes with only one child. All internal nodes have two children.
- **Complete Binary Tree:** A binary tree is complete if all levels are completely filled, except possibly the last level, and the last level has all its nodes as far left as possible. A perfect binary tree is always a complete binary tree.
- **Perfect Binary Tree:** A binary tree is perfect if all internal nodes have two children and all leaves are at the same level. This results in a perfectly symmetrical structure.
- **Skewed Binary Tree:** A binary tree is skewed if all nodes have only one child, forming a linear list. It can be left-skewed (all nodes are left children) or right-skewed (all nodes are right children).

The simplicity of the two-child constraint makes binary trees efficient for many operations. For example, in a balanced binary tree, searching for an element can be done in logarithmic time, O(logn), where n is the number of nodes. This efficiency is a major reason for their popularity.

Applications of binary trees are diverse. They are used to implement expression trees, where internal nodes represent operators and leaf nodes represent operands. Binary trees are also the underlying structure for binary heaps, which are crucial for priority queues and sorting algorithms like heapsort. Furthermore, they serve as the foundation for more specialized tree structures such as Binary Search Trees (BSTs) and AVL trees, which impose additional rules on the ordering of elements to facilitate efficient searching and retrieval. The concept of a binary tree is fundamental to understanding these advanced data structures and their applications in areas like database indexing, file systems, and algorithm design. Their hierarchical nature and the constraint on the number of children provide a powerful framework for organizing and manipulating data efficiently.

Tree Traversal

Tree traversal refers to the process of visiting each node in a tree data structure exactly once, in a systematic manner. Unlike linear data structures where traversal is straightforward (e.g., iterating through an array), trees require specific algorithms to ensure all nodes are visited and no node is visited multiple times. The order in which nodes are visited depends on the specific traversal method used. For binary trees, there are three primary traversal methods: Inorder, Preorder, and Postorder. Each method offers a unique sequence of visiting nodes and is suitable for different applications.

1. Inorder Traversal (Left-Root-Right): Inorder traversal visits the left subtree first, then the root node, and finally the right subtree. This traversal method is particularly useful for binary search trees (BSTs) because it visits the nodes in non-decreasing (sorted) order of their values. The steps are:

1. Traverse the left subtree recursively.
2. Visit the root node (process its data).

3. Traverse the right subtree recursively. For example, if a BST contains nodes (4, 2, 5, 1, 3), an inorder traversal would output 1, 2, 3, 4, 5. This property makes inorder traversal ideal for tasks requiring sorted output, such as printing the elements of a BST in ascending order.

2. Preorder Traversal (Root-Left-Right): Preorder traversal visits the root node first, then the left subtree, and finally the right subtree. This method is often used to create a copy of a tree or to represent the tree's structure. It's also useful for parsing expression trees (e.g., to get a prefix expression). The steps are:

1. Visit the root node (process its data).
2. Traverse the left subtree recursively.
3. Traverse the right subtree recursively. For the same example (4, 2, 5, 1, 3), a preorder traversal might output 4, 2, 1, 3, 5. This sequence effectively shows the "order of operations" or the hierarchy from the root downwards.

3. Postorder Traversal (Left-Right-Root): Postorder traversal visits the left subtree first, then the right subtree, and finally the root node. This traversal is commonly used to delete a tree (deallocating memory from leaf nodes upwards) or to evaluate expression trees (e.g., to get a postfix expression). The steps are:

1. Traverse the left subtree recursively.
2. Traverse the right subtree recursively.
3. Visit the root node (process its data). For the example (4, 2, 5, 1, 3), a postorder traversal might output 1, 3, 2, 5, 4. This order ensures that child nodes are processed before their parents, which is crucial for tasks like deallocation to avoid dereferencing freed memory.

Besides these depth-first traversals, there is also **Level-Order Traversal** (Breadth-First Traversal), which visits nodes level by level, starting from the root and moving across each level from left to right. This is typically implemented using a queue.

Choosing the appropriate traversal method depends entirely on the problem at hand. Understanding these fundamental traversal techniques is essential for working with tree data structures, as they form the basis for many tree-based algorithms and applications.

Binary Search Trees (BSTs)

A Binary Search Tree (BST) is a specialized type of binary tree that maintains a specific ordering property among its nodes, which greatly facilitates efficient searching, insertion, and deletion operations. This property dictates that for any given node:

1. All values in its **left subtree** must be less than the value of the node itself.
2. All values in its **right subtree** must be greater than the value of the node itself.
3. Both the left and right subtrees must themselves be Binary Search Trees.

This ordering principle ensures that searching for an element becomes highly efficient. Starting from the root, if the target value is less than the current node's value, the search proceeds to the left child; if it's greater, the search proceeds to the right child. If the target value is equal to the current node's value, the element is found. This process eliminates half of the remaining tree with each comparison, leading to a logarithmic time complexity, O(logn), in the average case (for a balanced BST), where n is the number of nodes. In the worst-case scenario (e.g., a skewed tree where elements are inserted in

strictly ascending or descending order), a BST can degenerate into a linked list, and search time can become O(n).

Insertion into a BST: To insert a new node, we traverse the tree starting from the root, following the same comparison logic as searching. If the new value is less than the current node, we go left; otherwise, we go right. We continue this until we reach a null pointer (an empty spot), where the new node is then inserted. Duplicates are usually handled by either storing them in the right subtree, storing a count in the node, or disallowing them.

Deletion from a BST: Deleting a node from a BST is more complex as it needs to maintain the BST property. There are three main cases:

1. **Node is a leaf:** Simply remove the node.
2. **Node has one child:** Replace the node with its only child.
3. **Node has two children:** This is the most complex case. We replace the node's value with either its inorder predecessor (largest value in the left subtree) or its inorder successor (smallest value in the right subtree). After replacement, the predecessor/successor node is then deleted from its original position (which will fall into one of the simpler deletion cases).

BSTs are widely used in applications requiring quick lookups, insertions, and deletions, such as dictionaries, sets, and database indexing. However, their performance heavily relies on the tree remaining relatively balanced. If the tree becomes unbalanced, operations can degrade to linear time. This limitation leads to the development of self-balancing BSTs, such as AVL trees and Red-Black trees, which automatically adjust their structure to maintain balance after insertions and deletions, guaranteeing O(logn) performance in the worst case. Despite this, understanding basic BSTs is fundamental to grasping more advanced tree structures.

AVL Trees

AVL trees are a type of self-balancing Binary Search Tree (BST) named after their inventors, Adelson-Velsky and Landis. The key characteristic of an AVL tree is that for every node, the difference between the height of its left subtree and the height of its right subtree (known as the **balance factor**) must be at most 1. The balance factor for a node is calculated as height(left_subtree)−height(right_subtree). If this balance factor becomes greater than 1 or less than -1, the tree is considered unbalanced, and a rebalancing operation (rotation) is performed to restore the AVL property. This strict balance condition ensures that an AVL tree remains relatively balanced, guaranteeing O(logn) time complexity for search, insertion, and deletion operations, even in the worst case, unlike a standard BST which can degrade to O(n).

Balance Factor: For each node, the balance factor can be -1, 0, or 1.

- 0: The left and right subtrees have the same height.
- 1: The left subtree is one level taller than the right subtree.
- -1: The right subtree is one level taller than the left subtree.

Rotations: When an insertion or deletion causes a node's balance factor to become 2 or -2, a rotation is performed to restore balance. There are four types of rotations:

1. **Left Rotation (LL Imbalance):** Occurs when a node becomes unbalanced due to an insertion in its left child's left subtree. This involves a single right rotation. The parent node shifts to become the right child of its left child.

2. **Right Rotation (RR Imbalance):** Occurs when a node becomes unbalanced due to an insertion in its right child's right subtree. This involves a single left rotation. The parent node shifts to become the left child of its right child.
3. **Left-Right Rotation (LR Imbalance):** Occurs when a node becomes unbalanced due to an insertion in its left child's right subtree. This is a double rotation: first, a left rotation on the left child, then a right rotation on the current node.
4. **Right-Left Rotation (RL Imbalance):** Occurs when a node becomes unbalanced due to an insertion in its right child's left subtree. This is also a double rotation: first, a right rotation on the right child, then a left rotation on the current node.

These rotations are relatively simple operations that involve re-arranging a few pointers, ensuring that the tree remains balanced and the BST property is preserved.

Insertion into an AVL Tree: Insertion in an AVL tree is similar to a standard BST. The new node is inserted as a leaf. After insertion, we trace back up the path from the inserted node to the root, checking the balance factor of each ancestor. If any node's balance factor goes beyond the allowed range (2 or -2), the appropriate rotation(s) are performed to rebalance that subtree.

Deletion from an AVL Tree: Deletion is also similar to a BST. After deleting a node, the balance of the affected ancestors (from the deleted node's parent up to the root) needs to be checked and rotations performed if necessary. This process can be more complex than insertion as multiple rotations might be required up the path to the root.

AVL trees offer a good balance between the performance benefits of a BST and the guaranteed logarithmic time complexity. They are used in scenarios where frequent insertions and deletions are expected and consistent performance is critical, such as in database systems, symbol tables, and network routing tables. While slightly more complex to implement than basic BSTs due to the balancing logic, the performance guarantees often justify the added complexity.