**Tree Data Structure**

**Overview**

**A Tree is a hierarchical data structure consisting of nodes, where:**

* **The topmost node is called the root.**
* **Each node contains data and may have child nodes.**
* **Nodes without children are called leaves.**
* **A parent node points to its children.**
* **Trees do not contain cycles (i.e., they are connected and acyclic).**

**Properties of Trees**

1. **A tree with N nodes has N-1 edges.**
2. **Exactly one path exists between any two nodes.**
3. **Common types of trees:** 
   * **Binary Tree**
   * **Binary Search Tree (BST)**
   * **AVL Tree (Self-balancing BST)**
   * **Red-Black Tree**
   * **Trie**
   * **N-ary Tree**

**Terminologies**

* **Depth: Length of the path from the root to a node.**
* **Height: Length of the path from a node to the deepest leaf.**
* **Level: Nodes at the same depth.**
* **Degree: Number of children of a node.**

**Types of Trees**

**1. Binary Tree**

**Each node has at most two children:**

* **Full Binary Tree: All nodes have 0 or 2 children.**
* **Complete Binary Tree: All levels are fully filled except possibly the last level.**
* **Perfect Binary Tree: All levels are fully filled.**
* **Skewed Tree: All nodes have only one child (left or right).**

**Code Structure in Java:**

**class TreeNode {**

**int val;**

**TreeNode left, right;**

**TreeNode(int val) {**

**this.val = val;**

**this.left = null;**

**this.right = null;**

**}**

**}**

**2. Binary Search Tree (BST)**

* **Left child < Parent < Right child.**
* **Inorder traversal yields sorted order of elements.**

**Code Structure in Java:**

**class BSTNode {**

**int val;**

**BSTNode left, right;**

**BSTNode(int val) {**

**this.val = val;**

**this.left = null;**

**this.right = null;**

**}**

**void insert(int key) {**

**if (key < val) {**

**if (left == null) left = new BSTNode(key);**

**else left.insert(key);**

**} else {**

**if (right == null) right = new BSTNode(key);**

**else right.insert(key);**

**}**

**}**

**}**

**3. AVL Tree**

* **Self-balancing binary search tree.**
* **Balance factor: Difference between heights of left and right subtrees is at most 1.**

**Code Structure in Java:**

**class AVLNode {**

**int val, height;**

**AVLNode left, right;**

**AVLNode(int val) {**

**this.val = val;**

**this.height = 1;**

**this.left = null;**

**this.right = null;**

**}**

**}**

**class AVLTree {**

**AVLNode root;**

**int height(AVLNode node) {**

**return node == null ? 0 : node.height;**

**}**

**int getBalance(AVLNode node) {**

**return node == null ? 0 : height(node.left) - height(node.right);**

**}**

**// Add rotation and insertion methods here...**

**}**

**4. Red-Black Tree**

* **A self-balancing binary search tree with properties like red and black coloring for balancing.**

**Code Structure in Java:**

**class RBNode {**

**int val;**

**RBNode left, right, parent;**

**boolean color; // true for red, false for black**

**RBNode(int val) {**

**this.val = val;**

**this.color = true; // New nodes are always red initially**

**}**

**}**

**class RedBlackTree {**

**RBNode root;**

**// Add insertion, deletion, and balancing methods...**

**}**

**5. Segment Tree**

* **A binary tree used for storing intervals or segments.**
* **Each node represents an interval, and the root represents the entire range.**

**Code Structure in Java:**

**class SegmentTree {**

**int[] tree;**

**int n;**

**SegmentTree(int[] arr) {**

**n = arr.length;**

**tree = new int[4 \* n];**

**build(arr, 0, 0, n - 1);**

**}**

**void build(int[] arr, int node, int start, int end) {**

**if (start == end) {**

**tree[node] = arr[start];**

**} else {**

**int mid = (start + end) / 2;**

**build(arr, 2 \* node + 1, start, mid);**

**build(arr, 2 \* node + 2, mid + 1, end);**

**tree[node] = tree[2 \* node + 1] + tree[2 \* node + 2];**

**}**

**}**

**int query(int node, int start, int end, int l, int r) {**

**if (r < start || l > end) return 0; // Out of range**

**if (l <= start && end <= r) return tree[node];**

**int mid = (start + end) / 2;**

**return query(2 \* node + 1, start, mid, l, r) + query(2 \* node + 2, mid + 1, end, l, r);**

**}**

**}**

**6. Trie**

* **Tree used for efficient string operations (prefix matching).**
* **Nodes represent characters of strings.**

**Code Structure in Java:**

**class TrieNode {**

**TrieNode[] children;**

**boolean isEndOfWord;**

**TrieNode() {**

**children = new TrieNode[26];**

**isEndOfWord = false;**

**}**

**}**

**class Trie {**

**TrieNode root;**

**Trie() {**

**root = new TrieNode();**

**}**

**void insert(String word) {**

**TrieNode node = root;**

**for (char c : word.toCharArray()) {**

**int index = c - 'a';**

**if (node.children[index] == null) node.children[index] = new TrieNode();**

**node = node.children[index];**

**}**

**node.isEndOfWord = true;**

**}**

**}**

**7. N-ary Tree**

* **A tree where each node can have at most N children.**

**Code Structure in Java:**

**class NaryNode {**

**int val;**

**List<NaryNode> children;**

**NaryNode(int val) {**

**this.val = val;**

**this.children = new ArrayList<>();**

**}**

**}**

**Traversal Techniques**

**1. Depth-First Search (DFS)**

* **Preorder Traversal: Root -> Left -> Right.**
* **Inorder Traversal: Left -> Root -> Right.**
* **Postorder Traversal: Left -> Right -> Root.**

**2. Breadth-First Search (BFS)**

* **Level order traversal.**
* **Use a queue to visit nodes level by level.**

**Common Tree Problems**

**1. Tree Traversals**

* **Problem: Print nodes in preorder, inorder, and postorder.**
* **Techniques:** 
  + **Recursive solution.**
  + **Iterative solution using stacks (for DFS).**

**2. Find Height of a Tree**

* **Problem: Compute the height of a tree.**
* **Techniques:** 
  + **Recursively find the max height of left and right subtrees.**

**3. Check if Binary Tree is Balanced**

* **Problem: Check if the height difference between left and right subtrees is ≤ 1 for all nodes.**
* **Techniques:** 
  + **Use a recursive function that returns both height and balance status.**

**4. Lowest Common Ancestor (LCA)**

* **Problem: Find the lowest common ancestor of two nodes.**
* **Techniques:** 
  + **Recursive approach.**
  + **For BSTs, leverage the sorted property.**

**5. Diameter of a Tree**

* **Problem: Longest path between any two nodes.**
* **Techniques:** 
  + **Recursive function that calculates both height and diameter.**

**6. Serialize and Deserialize a Tree**

* **Problem: Convert a tree to a string and vice versa.**
* **Techniques:** 
  + **BFS or DFS-based serialization.**

**7. Symmetric Tree**

* **Problem: Check if a tree is symmetric (mirror image of itself).**
* **Techniques:** 
  + **Recursive comparison of left and right subtrees.**

**8. Path Sum Problems**

* **Problem: Find paths with a given sum.**
* **Techniques:** 
  + **DFS with backtracking to find all paths.**
  + **Use prefix sum for optimization.**

**9. Subtree of Another Tree**

* **Problem: Check if one tree is a subtree of another.**
* **Techniques:** 
  + **Preorder traversal comparison.**
  + **String matching using serialization.**

**10. Construct Tree from Traversals**

* **Problem: Construct a tree from preorder, inorder, or postorder traversal arrays.**
* **Techniques:** 
  + **Use hashmap to optimize construction process.**

**Problem-Solving Patterns**

**Pattern 1: Divide and Conquer**

* **Split the problem into subproblems and combine results.**
* **Examples: Height of a tree, LCA, diameter of a tree.**

**Pattern 2: Backtracking**

* **Explore all paths and backtrack when needed.**
* **Examples: Path sum problems.**

**Pattern 3: Dynamic Programming on Trees**

* **Use postorder traversal for bottom-up computation.**
* **Examples: Diameter of a tree, max path sum.**

**Pattern 4: Two Pointers / Iterative Traversals**

* **Use a stack or queue to iteratively traverse the tree.**
* **Examples: Iterative inorder traversal.**

**Optimization Techniques**

1. **Memoization: Cache results for overlapping subproblems.**
2. **Space Optimization: Use Morris traversal for inorder traversal without recursion or stack.**
3. **Efficient Search: Leverage BST properties for optimized search.**

**Applications of Trees**

1. **Expression Trees: Used in compilers to evaluate expressions.**
2. **File Systems: Directory structures are represented as trees.**
3. **Network Routing: Hierarchical organization of routers.**
4. **Game Trees: Used in AI for decision-making.**
5. **Tries: Used in autocomplete and dictionary implementations.**