Name : Saurav Suman

Reg No : 12210691

1. Introduction to Tree Data Structures

A **tree** is a widely used hierarchical data structure that mimics a tree-like model with a root and branching children. Each node holds a value and may point to one or more child nodes. Unlike linear data structures such as arrays or linked lists, trees naturally represent relationships and hierarchies.

Trees are utilized in various applications:

- Hierarchical file systems
- Decision trees in AI
- Databases (e.g., B-trees in indexing)
- Network routing protocols

Trees offer logarithmic search and update operations in many cases, making them ideal for dynamic data structures.

---

2. Tree Terminology

Understanding the terminology of trees is essential for implementing and working with them:

- **Root**: The first node in the tree.
- **Parent**: A node that has branches to one or more child nodes.
- **Child**: A node that descends from a parent node.
- **Leaf**: A node that does not have any children.

- **Edge**: A connection between two nodes.
- **Height**: The number of edges on the longest downward path between the root and a leaf.
- **Depth**: The number of edges from the root to a specific node.
- **Subtree**: A portion of the tree consisting of a node and its descendants.

Understanding these helps in building traversal and insertion logic.

---

3. Types of Trees

- **Binary Tree**: Every node has at most two children—left and right.
- **Full Binary Tree**: Every node has 0 or 2 children.
- **Complete Binary Tree**: All levels are filled except possibly the last, filled from left to right.
- **Perfect Binary Tree**: All internal nodes have two children and all leaves are at the same level.
- **Binary Search Tree (BST)**: Left child < root < right child; used for efficient search.
- **AVL Tree**: Self-balancing binary search tree with rotation mechanisms.
- **Heap**: Complete binary tree satisfying the heap property (min or max).
- **Trie**: Tree used for storing words in a character-wise manner.
- **Segment Tree**: Stores intervals or segments; useful in range query problems.
- **N-ary Tree**: A tree where nodes can have more than two children.

---

4. Binary Tree Implementation

Binary trees are fundamental structures for tree operations. A basic implementation includes:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

To create a tree:

```
root = TreeNode(10)
root.left = TreeNode(5)
root.right = TreeNode(15)
```

We use recursive or iterative methods to insert, delete, or search values.

Use Cases:

- Representation of arithmetic expressions
- Parsing trees in compilers

---

5. Binary Search Tree (BST)

A BST is a binary tree that maintains sorted order, enabling fast search and insertion.

Key Rules:

- Left subtree has values less than the node.
- Right subtree has values greater than the node.

Insertion:

```
def insert(root, val):
    if not root:
        return TreeNode(val)
    if val < root.val:
        root.left = insert(root.left, val)
    else:
        root.right = insert(root.right, val)
    return root
```

Example **Problem**:
Insert 10, 5, 15, 3, 7 → Inorder traversal gives [3, 5, 7, 10, 15]

Time Complexity: O(log n) on average, O(n) worst-case (unbalanced)

---

6. Tree Traversals

Tree traversal is the process of visiting each node in a tree.

Depth-First Traversal:

- **Inorder (Left, Root, Right)**: Produces sorted order in BST.
- **Preorder (Root, Left, Right)**: Used for tree cloning.
- **Postorder (Left, Right, Root)**: Used for deletion.

Breadth-First Traversal:

- **Level Order**: Visits nodes level-by-level (uses a queue).

Inorder Example:

```
def inorder(root):
    if root:
        inorder(root.left)
        print(root.val)
        inorder(root.right)
```

Use Case: Traversals are key in evaluating expressions, serialization, and search operations.

---

7. Balanced Trees: AVL Tree

AVL trees maintain a balance factor to ensure optimal performance:

Balance Factor:

BF = height(left subtree) - height(right subtree)

Allowed values: -1, 0, 1

Rotations:

- **Right Rotation (RR)**
- **Left Rotation (LL)**
- **Left-Right Rotation (LR)**
- **Right-Left Rotation (RL)**

Insertion with Rebalancing:

```
if balance > 1 and val < node.left.val:
    return rightRotate(node)
```

Time Complexity: O(log n)

Use Case: Ensures worst-case logarithmic operations.

---

8. Heaps (Min Heap and Max Heap)

Heaps are used in priority queues and sorting:

- **Min Heap**: Root is minimum.
- **Max Heap**: Root is maximum.

Operations:

- **Insert**: Add at end, heapify-up
- **Remove**: Replace root with last element, heapify-down

Implementation:
Use arrays. For index i:

- Left child: 2i + 1
- Right child: 2i + 2
- Parent: (i - 1) // 2

Example: Insert 5, 3, 8, 1 → Min Heap becomes [1, 3, 8, 5]

---

9. Tries (Prefix Trees)

Tries store sets of strings efficiently:

```
class TrieNode:
    def __init__(self):
        self.children = {}
```

```
        self.end_of_word = False
```

Operations:

- **Insert**
- **Search**
- **Prefix Matching**

Use Cases:

- Auto-complete
- Spell checking
- IP routing

Tries offer O(L) operations where L is word length.

---

10. Segment Tree

Segment Trees solve range query problems:

- **Build**: Recursively divide array and build tree.
- **Query**: Sum/Min/Max in a range
- **Update**: Modify value and update parent nodes

Build Example:

```
def build(arr, node, start, end):
    if start == end:
        tree[node] = arr[start]
    else:
        mid = (start + end) // 2
        build(arr, 2*node, start, mid)
```

```
    build(arr, 2*node+1, mid+1, end)
    tree[node] = tree[2*node] + tree[2*node+1]
```

Time Complexity:

- Build: O(n)
- Query/Update: O(log n)

---

11. Advanced Tree Concepts

Red-Black Tree:

- Self-balancing BST
- Each node is red or black
- Ensures height balance via color properties

Fenwick Tree (Binary Indexed Tree):

- Efficient for cumulative frequency
- Update/query in O(log n)

Suffix Tree:

- Compressed trie of all suffixes
- Used in string matching, bioinformatics

These structures provide optimized performance for specialized tasks.

---

12. Sample Problems with Solutions

1. Count Leaf Nodes

```
def count_leaves(root):
    if not root:
        return 0
    if not root.left and not root.right:
        return 1
    return count_leaves(root.left) + count_leaves(root.right)
```

2. Check if BST

```
def isBST(root, min_val=float('-inf'), max_val=float('inf')):
    if not root:
        return True
    if not (min_val < root.val < max_val):
        return False
    return isBST(root.left, min_val, root.val) and isBST(root.right, root.val, max_val)
```

3. Lowest Common Ancestor

```
def LCA(root, p, q):
    if not root or root == p or root == q:
        return root
    left = LCA(root.left, p, q)
    right = LCA(root.right, p, q)
    return root if left and right else left or right
```

4. Convert Sorted Array to BST

```
def sortedArrayToBST(nums):
    if not nums:
        return None
    mid = len(nums) // 2
    root = TreeNode(nums[mid])
```

```
    root.left = sortedArrayToBST(nums[:mid])
    root.right = sortedArrayToBST(nums[mid+1:])
    return root
```

5. Serialize and Deserialize Tree

```
def serialize(root):
    vals = []
    def dfs(node):
        if node:
            vals.append(str(node.val))
            dfs(node.left)
            dfs(node.right)
        else:
            vals.append('#')
    dfs(root)
    return ' '.join(vals)
```

---

13. Summary

Trees are a cornerstone of computer science. They allow efficient hierarchical data management, fast searching, and advanced computation via specialized forms like BSTs, AVL trees, and Tries. Mastering them equips developers for roles in data processing, algorithm design, and system development.

Key Takeaways:

- Trees model hierarchy and recursion
- Balanced trees provide fast operations
- Traversals enable computation and structure
- Tries, Heaps, and Segment Trees solve real-world problems

Practice on platforms like LeetCode, GeeksforGeeks, and HackerRank is recommended for mastery.