Session - 13

Tree

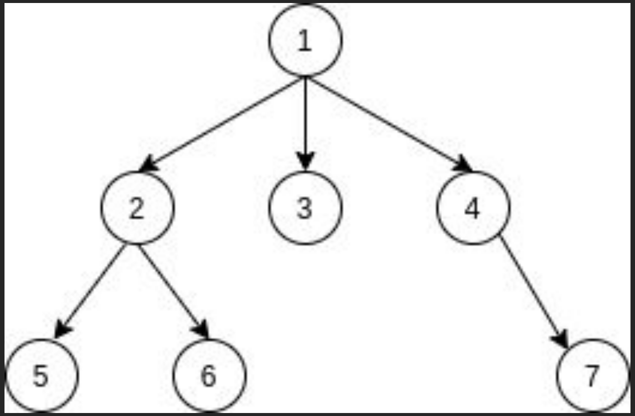
CONTENTS

01

Introduction to Tree

Introduction to Tree

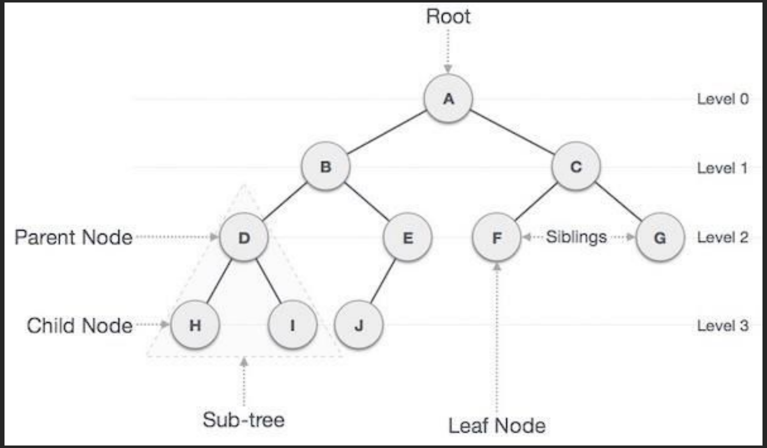
- A **tree** is a special type of **linked list**
- Each node points to its **children**
- Can store any data (int, string, etc.)
- The **structure is recursive**
- No cycles allowed
- Nodes with no children = **Leaf Nodes**



Introduction to Tree

Tree Basics

- **Root Node**: Starting point of the tree
- **Child Nodes**: Connected to a parent
- **Recursive Definition**: Tree made of subtrees
- Usually, **no link to parent** (but can be added if needed)



02

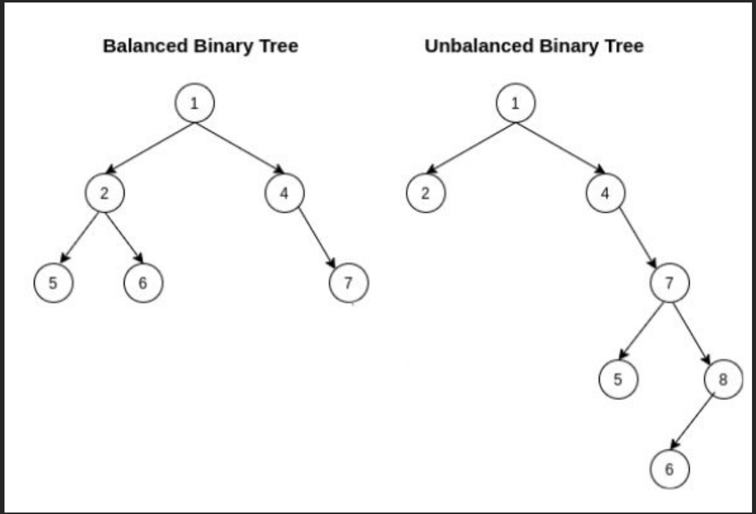
Binary Tree

Binary Tree

- A **Binary Tree** has at most **2 children**
- Left Child and Right Child

Types:

1. **Balanced**: Leaf nodes are close in depth
2. **Unbalanced**: Some branches deeper than others



Balanced Binary Tree

- ➢ Height difference between left & right subtree ≤ 1
- ➢ Ensures faster operations: **O(log n)**
- ➢ Helps with performance for:
  - **Insert**
  - **Search**
  - **Update**
  - **Delete**

Applications of Binary Tree

- **Binary Search Tree (BST)** – fast lookup
- **Binary Heap** – for priority queues
- **Hash Trees** – cryptographic use
- **Abstract Syntax Trees** – compilers
- **Huffman Tree** – compression
- **Routing Trees** – networks

03

Binary Search Tree

Binary Search Tree

- A type of Binary Tree where:
  - Left child ≤ Parent
  - Right child > Parent
- **All BSTs are Binary Trees**, but not vice versa
- Search time:
  - **O(log n)** average (if balanced)
  - **O(n)** worst (if unbalanced)

Balanced BST

- To keep search time fast, **BST must be balanced**
- Balancing methods:
  - **Left Rotation**
  - **Right Rotation**
- Examples of **Self-Balancing BSTs**:
  - AVL Tree
  - Red-Black Tree

BST – Pros & Cons

Advantages:

- Fast insertion, deletion, and lookup: **O(h)**
- Ideal for large datasets
- Used in:
  - **Priority Queues**
  - **Range queries**
  - **Database indexing**

Disadvantages:

- Can **degenerate** to linked list if not balanced
- **Balancing takes time**, but it's worth it

04

Tree Traversal

Tree Traversals

- **In-order**: Left → Root → Right
- **Pre-order**: Root → Left → Right
- **Post-order**: Left → Right → Root
- Can be done using **stack (iterative)** or **recursion**

Binary Tree Views

- **Left View**: First node at each level

  Example :- [Left View]

- **Right View**: Last node at each level

- **Top View**: First node from top for each vertical line

  Example :- [Top View]

- **Bottom View**: Last node from bottom for each vertical line

  ➡ Use **Level Order** or **Vertical Order Traversal**

Level Order Traversal

- Use **Queue**
- Push root into queue
- While queue is not empty
  - Pop node
  - Push its children
- Use nested loop to process nodes **level by level**
- Optional: Use **marker (e.g., -1)** to separate levels

Vertical Order Traversal

- ➢ Use **horizontal distance (hd)**
- ➢ Root = hd 0
  Left child: hd - 1
  Right child: hd + 1
- ➢ Store nodes in a **Map<hd, List<Node>>**
- ➢ All nodes with same hd → Same vertical line

05

Activity

Activity Tree

- Invert Binary Tree
- Same Tree
- Symmetric Tree
- Subtree of Another Tree
- Maximum Depth of Another Tree
- Balanced Binary Tree
- Binary Tree Level Order Traversal
- Binary Tree Zigzag Level Order Traversal

Activity Tree

- </> [Lowest Common Ancestor of a Binary Tree]
- </> [Lowest Common Ancestor of a Binary Search]
- </> [Kth Smallest Element in a bst]
- </> [Validate Binary Search Tree]
- </> [Serialize and Deserialize Binary Tree]
- </> [Convert Sorted array to Binary Search Tree]
- </> [Binary Tree Right Side View]
- </> [Construct Binary tree from preorder and inorder traversal]

Activity Tree

- Binary Tree Maximum Path Sum
- Maximum Width of Binary Tree
- All nodes distance K in Binary Tree