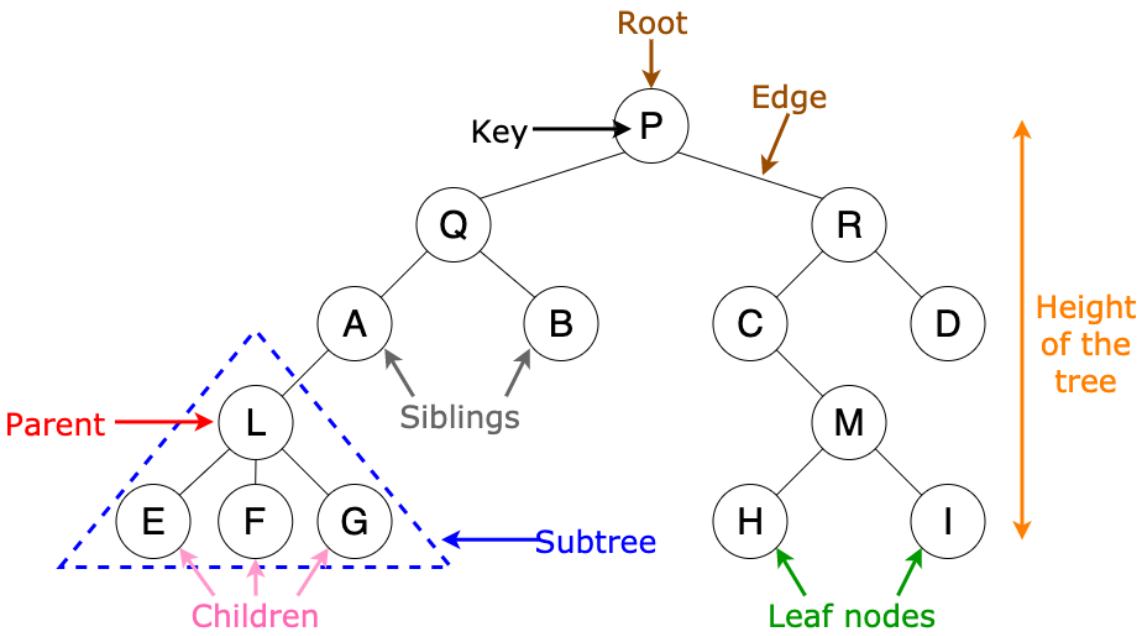
🌳 Tree Data Structures: A Comprehensive Tutorial

🌱 What is a Tree?

A Tree is a non-linear data structure where elements, called nodes, are connected like a family tree. Each node is linked to others through connections called edges, and there's always one and only one way to travel between any two nodes.

In a tree:

- The **topmost node** is called the **root**.
- Each node contains a **value/data**.
- Nodes may have **child nodes** connected via edges.
- A node with no children is called a **leaf node**.
- There is **exactly one path** between any two nodes.



🌳 Terminology

- **Root**: The topmost node in the tree.
- **Parent**: A node that has one or more children.
- **Child**: A node that descends from another node.
- **Leaf**: A node with no children.
- **Edge**: A connection between two nodes.
- **Path**: A sequence of nodes connected by edges.

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

🌲 Types of Trees

- **General Tree:** No limit on children per node.
- **Binary Tree:** A binary tree is a tree in which each node can have at most two children, referred to as the left child and the right child.
- **Binary Search Tree (BST):** A binary search tree is a binary tree in which the left child of a node contains a value smaller than the node's value, and the right child contains a value greater than the node's value. This property allows for efficient searching, insertion, and deletion operations.
- **AVL Tree:** An AVL tree is a self-balancing binary search tree. It maintains a balance factor for each node to ensure that the tree remains balanced, which improves the overall performance of operations.
- **B-tree:** A B-tree is a self-balancing search tree that is commonly used in databases and file systems. It is optimized for systems that read and write large blocks of data, making it efficient for disk-based storage.
- **Red-Black Tree:** A red-black tree is another type of self-balancing binary search tree. It ensures that the tree remains balanced by applying specific rules related to node coloring.
- **N-ary Tree:** Each node has at most N children.
- **Balanced Tree:** Height difference between left and right subtree is minimal.
- **Degenerate Tree:** Each parent has only one child (like a linked list).

🌳 Binary Tree and Its Types

Types:

- **Full Binary Tree**: Every node has 0 or 2 children.
- **Complete Binary Tree**: All levels filled except possibly the last.
- **Perfect Binary Tree**: All internal nodes have 2 children, all leaves at the same level.
- **Skewed Tree**: All nodes have only left or right child.

Tree Representation in Code (Java Example)

```java
class Node {
    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
}

public class BinaryTree {
    Node root;

    BinaryTree() {
        root = null;
    }
}
```

🌿 Traversal Techniques

Depth First Search (DFS):

- **Inorder (Left → Root → Right)**

```java
void inorder(Node node) {
    if (node == null)
        return;
    inorder(node.left);
    System.out.print(node.data + " ");
    inorder(node.right);
}
```

- **Preorder Traversal (Root → Left → Right)**

```
void preorder(Node node) {
    if (node == null) return;
    System.out.print(node.data + " ");
    preorder(node.left);
    preorder(node.right);
}
```

- **Postorder Traversal (Left → Right → Root)**

```
void postorder(Node node) {
    if (node == null) return;
    postorder(node.left);
    postorder(node.right);
    System.out.print(node.data + " ");
}
```

Breadth First Search (BFS): Level Order Traversal

```
void levelOrder(Node root) {
    Queue<Node> queue = new LinkedList<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        Node temp = queue.poll();
        System.out.print(temp.data + " ");
        if (temp.left != null)
            queue.add(temp.left);
        if (temp.right != null)
            queue.add(temp.right);
    }
}
```

🔍 Binary Search Tree (BST)

A BST is a type of binary tree where:

- Left subtree nodes < Root
- Right subtree nodes > Root

Insertion in BST

```
Node insert(Node root, int key) {
    if (root == null)
        return new Node(key);
    if (key < root.data)
        root.left = insert(root.left, key);
    else if (key > root.data)
        root.right = insert(root.right, key);
    return root;
}
```

🔍 Searching in BST

```
boolean search(Node root, int key) {
    if (root == null) return false;
    if (root.data == key) return true;
    return key < root.data ?
      search(root.left, key) : search(root.right, key);
}
```

Deletion:

```
public int findMin(Node root){
    int minVal = Integer.MAX_VALUE;
    Node temp=root;
    while(temp!=null){
        minVal=temp.data;
        temp=temp.left;
    }
    return minVal;
}
```

```java
public Node deleteKey(Node root, int key){
    if(root==null) return null;
    else{
        if(key<root.data) root.left=deleteKey(root.left, key);
        else if(key>root.data) root.right=deleteKey(root.right, key);
        else{ //when I found the key, now delete the key
            if(root.left==null) return root.right; // Right Child Exist
            else if(root.right==null) return root.left; // Left Child Exist
            else{//key contains both left and right children
                int minValue = findMin(root.right);
                root.data=minValue;
                root.right=deleteKey(root.right, root.data);
            }
        }
    }
    return root;
}
```

⚖️ Balanced Trees

1. AVL Tree

- Self-balancing BST
- Balancing factor: height(left) - height(right)
- Rotation needed after insertion/deletion

2. Red-Black Tree

- Self-balancing BST with coloring rules
- At most O(log n) height

🛠️ Heap (Min & Max)

- Complete binary tree
- **Min-Heap**: Parent < children
- **Max-Heap**: Parent > children

Insert in Heap:

- Add at end, bubble up.

Insert in Heap (Bubble Up):

```
void insert(List<Integer> heap, int val) {
    heap.add(val);
    int i = heap.size() - 1;
    while (i > 0 && heap.get(i) < heap.get((i - 1) / 2)) {
        Collections.swap(heap, i, (i - 1) / 2);
        i = (i - 1) / 2;
    }
}
```

Delete from Heap:

- Replace root with last, heapify down.

Delete Root from Heap (Heapify Down):

```
void deleteRoot(List<Integer> heap) {
    int n = heap.size();
    if (n == 0) return;
    heap.set(0, heap.get(n - 1));
    heap.remove(n - 1);
    int i = 0;
    while (true) {
        int left = 2 * i + 1, right = 2 * i + 2, smallest = i;
        if (left < heap.size() &&
            heap.get(left) < heap.get(smallest))
            smallest = left;
        if (right < heap.size() &&
            heap.get(right) < heap.get(smallest))
            smallest = right;
        if (smallest == i) break;
        Collections.swap(heap, i, smallest);
        i = smallest;
    }
}
```

🧠 Advanced Trees

1. Segment Tree – The Range Query Pro

What it is:
A Segment Tree is like a supercharged binary tree that helps you answer questions about a range of data — like “What’s the sum of elements between index 2 and 7?” — very quickly.

Why use it:
When you have to do multiple **range queries** and **updates** (like in competitive programming or games with dynamic data), it reduces the time complexity from O(n) to O(log n).

How it works:

- The array is divided recursively into halves (like a tournament tree).
- Each node stores information about a segment (like the sum or min of a range).
- Updates and queries only need to travel down log(n) levels.

Example Use Case:
You’re building a weather app and want to track the maximum temperature over the past 7 days — and the values change daily. Segment Tree handles this efficiently.

2. Trie (Prefix Tree) – The Dictionary Genius

What it is:
A Trie is a tree that stores strings character by character. It’s like how a phonebook works — narrowing down names as you type.

Why use it:
It makes searching for prefixes (like all words starting with "pre") super-fast. Unlike a normal tree or HashMap, it doesn’t check the whole word every time.

How it works:

- Each node represents a single character.
- A path from the root to a node represents a prefix or full word.
- Insert and search operations take **O(L)** time where **L is the length of the word**.

Example Use Case:
Autocomplete in Google Search — as you type "progr...", it instantly shows "programming", "progress", etc.

3. B+ Tree – The Database Workhorse

What it is:
A B+ Tree is like a balanced, multi-way tree that's optimized for **disk storage**. It stores **all actual data in the leaf nodes**, and internal nodes are used just for navigating.

Why use it:
It's perfect for huge datasets (millions of records) where loading everything into memory is not practical.

How it works:

- Unlike binary trees, nodes can have more than 2 children.
- Internal nodes guide the search, while **leaf nodes contain all the data** (and are linked together for fast range traversal).
- Designed to reduce disk I/O.

Example Use Case:
Used in databases like MySQL, PostgreSQL, and file systems to index large records efficiently.

📘 Important Tree Problems:

Practice Problems:

Q1. Lowest Common Ancestor of a Binary Tree

Q2. Lowest Common Ancestor of a Binary Search Tree

Q3. Unique Binary Search Trees

Q4. kth Smallest Element in a BST

Q5. Maximum Sum BST in a Binary Tree

Q6. Balance a Binary Search Tree

📌 *Note: Read about rotations in AVL Trees*

Q7. Validate Binary Search Tree

Q8. Symmetric Binary Trees

Q9. Binary Tree Zig-Zag Level Order Traversal

Q10. Binary Tree Level Order Traversal - II

Q11. Inorder Successor & Predecessor in a BST

- 🔐 *Leetcode Premium*: Inorder Successor in BST
- 📚 TakeUForward Explanation

Q12. Flatten Binary Tree to a Linked List

Q13. Binary Tree Paths

Q14. Merge BSTs to create a new BST

🌲 Advanced Tree Practice Problems

1. Range Sum Query - Mutable
2. Queue Reconstruction by Height
3. My Calendar I
4. My Calendar II
5. My Calendar III
6. Count Number of Teams
7. The Skyline Problem
8. Longest Increasing Subsequence II

📁 GitHub DSA Repository (by Harsh Raj)

- 🔗 **Full DSA Repository**
- 🌳 **Binary Tree Implementation (trees.java)**
- 📘 **Binary Search Tree Implementation (binarySearchTree.java)**