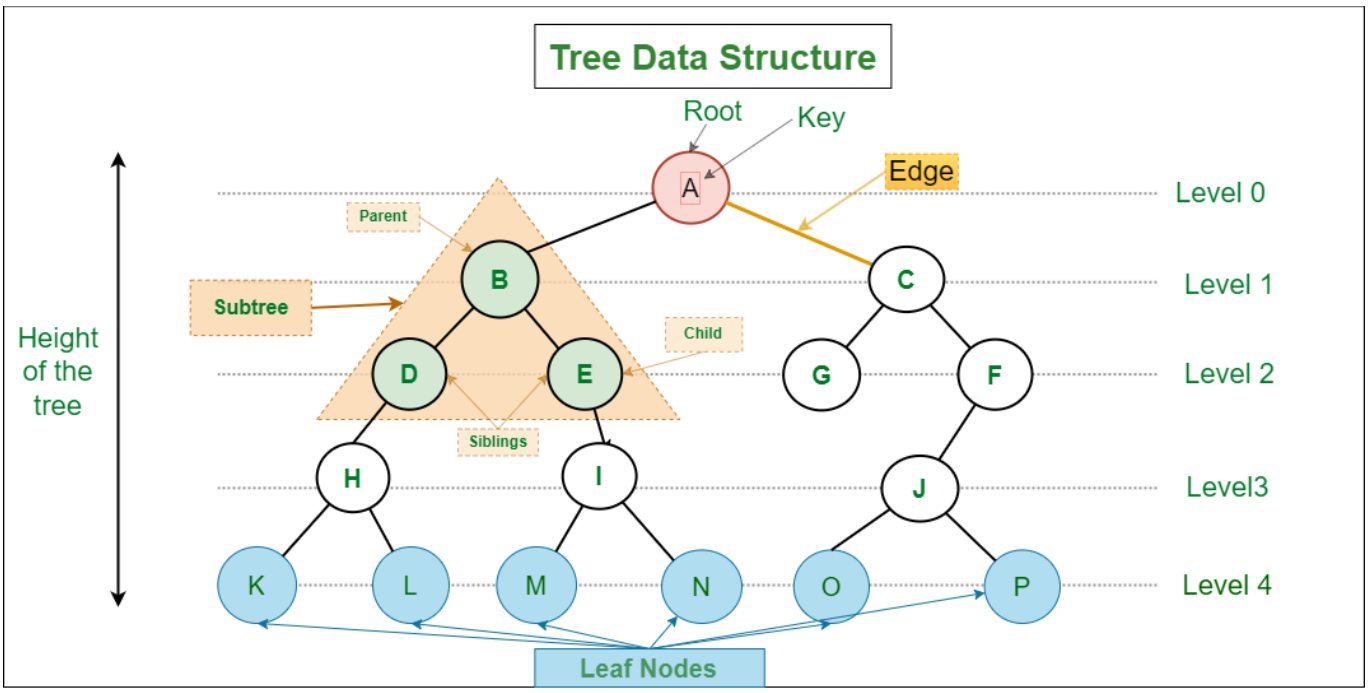
Session 7 & 8 - Trees

In computer science, a tree is a hierarchical data structure that consists of nodes connected by edges. It is called a "tree" because it often resembles a real-world tree turned upside down, with the root at the top and the leaves at the bottom.

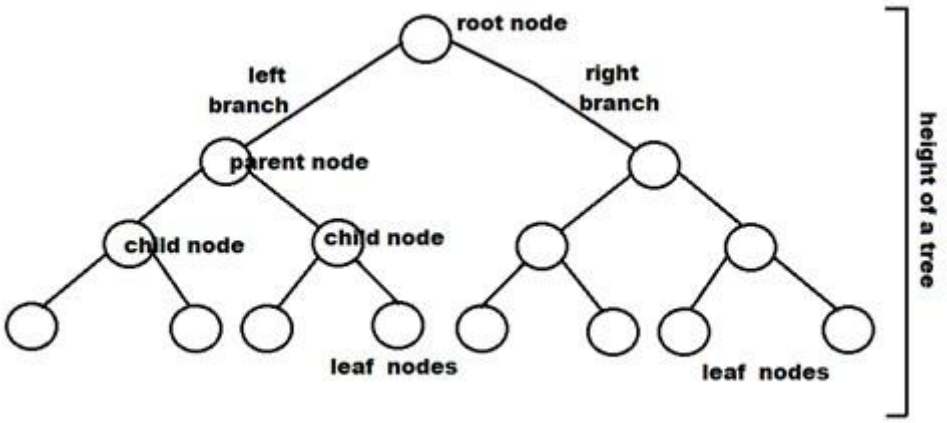


Key Terminology:

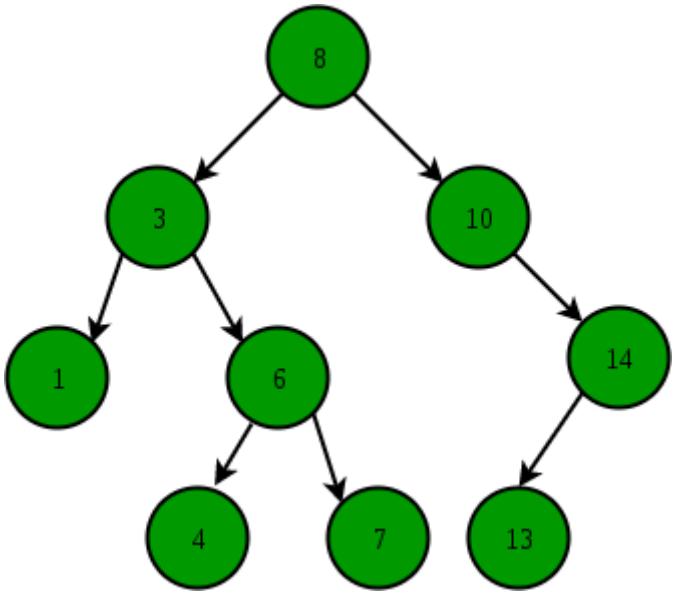
- Node: Each element in a tree is called a node. Each node contains a value (or data) and may have links (or references) to other nodes.
- Root: The topmost node of the tree, which has no parent. It serves as the starting point to access all other nodes in the tree.
- Parent: A node that has one or more child nodes connected to it.
- Child: Nodes directly connected to a parent node via edges.
- Siblings: Nodes that share the same parent.
- Leaf: Nodes that have no children. They are the end nodes of the tree.
- Depth: The distance between the root and a specific node. The depth of the root is 0.
- Height: The length of the longest path from a node to a leaf. The height of a tree is the height of the root node.

Types of Trees:

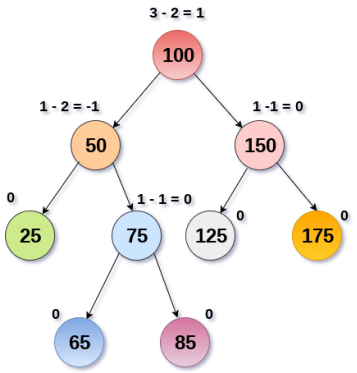
- Binary Tree: A tree in which each node can have at most two children, often referred to as the "left child" and the "right child."



- Binary Search Tree (BST): A binary tree with the following properties: the left child of a node contains a value less than or equal to the node's value, and the right child contains a value greater than the node's value. This property enables efficient searching, insertion, and deletion operations.

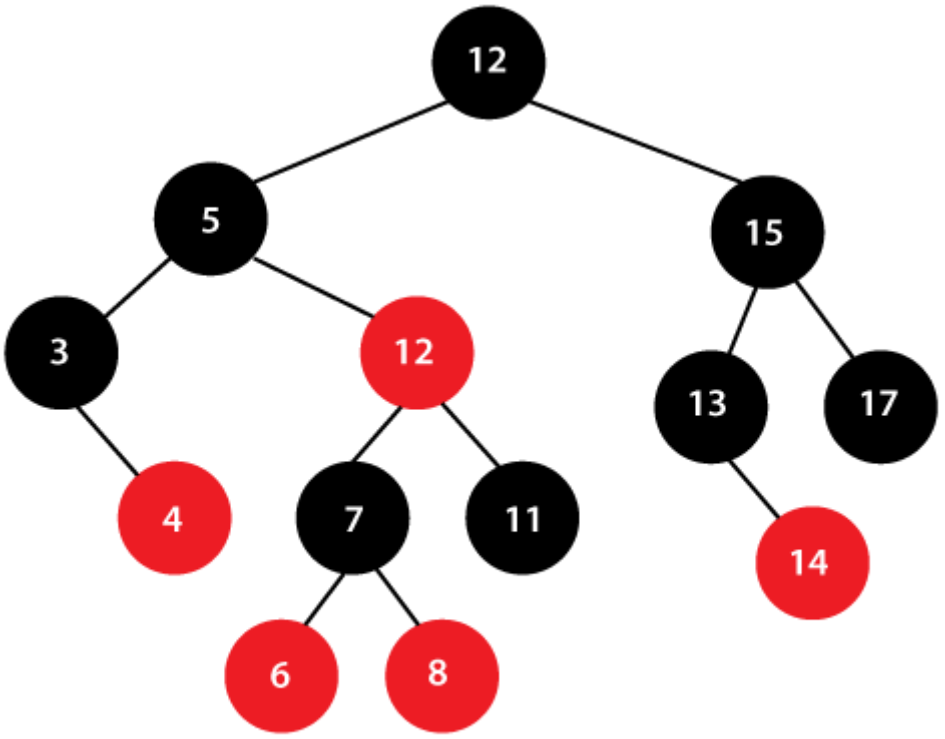


- AVL Tree: A self-balancing binary search tree in which the heights of the two child subtrees of every node differ by at most one. This ensures a balanced tree and guarantees O(log n) time complexity for various operations.



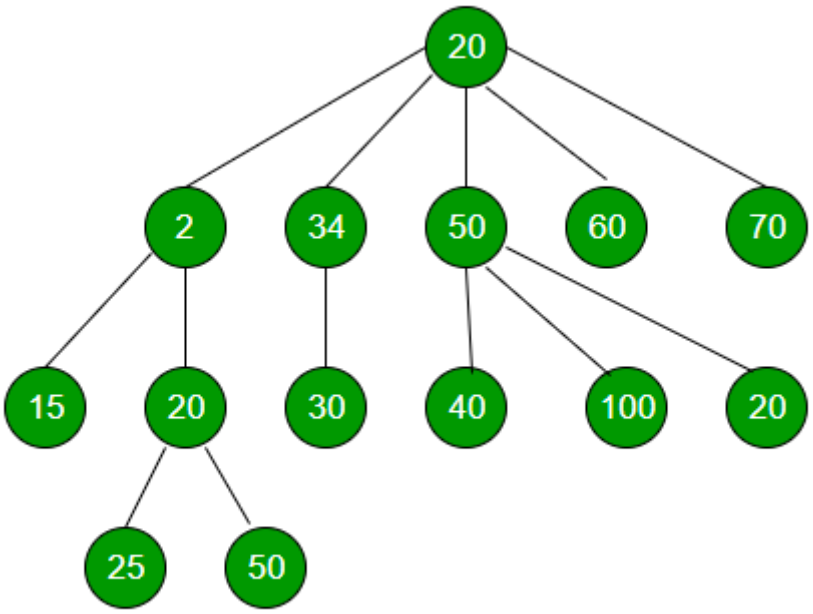
AVL Tree

- Red-Black Tree: Another self-balancing binary search tree that uses color-coding to maintain balance. It ensures O(log n) time complexity for basic operations like search, insertion, and deletion.



Red Black Tree

- N-ary Tree: A tree in which each node can have more than two children (e.g., a ternary tree, quaternary tree, etc.)



Applications of Trees:

Trees have numerous applications in computer science and beyond, including:

- Hierarchical data representation (e.g., file systems, organization charts).
- Search and retrieval operations (e.g., binary search trees).
- Expression evaluation (e.g., expression trees).
- Decision-making (e.g., decision trees, game trees).
- Network routing algorithms.
- Syntax analysis in compilers (e.g., parse trees).

Trees are fundamental data structures that play a crucial role in various algorithms and applications. Understanding trees and their properties is essential for designing efficient and optimised solutions to various computational problems.

Tree traversals

Tree traversal is a fundamental operation in computer science used to visit and process all the nodes of a tree data structure systematically. In tree traversal, you start from the root node (or any specific node in the tree) and visit all the nodes in a specific order, ensuring that you cover all the nodes exactly once. Different traversal algorithms determine the order in which the nodes are visited.

The three most common tree traversal algorithms are:

- **In-Order Traversal:** In an in-order traversal, you visit the left subtree, then the current node, and finally the right subtree. For a binary search tree, this traversal visits the nodes in ascending order. In general, this is the recommended traversal for binary search trees when you want to process the nodes in sorted order.

- Algorithm:

In-Order Traversal (Left-Root-Right):

- Start at the root node of the binary tree.
- If the current node is not null, recursively perform the following steps:
    - Traverse the left subtree:
        - Call the In-Order Traversal function with the left child of the current node as the argument.
    - Visit the current node (process the node in some way):
        - Print the value of the current node or perform any operation you want on it.
    - Traverse the right subtree:
        - Call the In-Order Traversal function with the right child of the current node as the argument.
- If the current node is null, return from the function.

- **Pre-Order Traversal:** In a pre-order traversal, you visit the current node first, then the left subtree, and finally the right subtree. This traversal is often used to make a copy of the tree or flatten the tree into an array.

- Algorithm:

Pre-Order Traversal (Root-Left-Right):

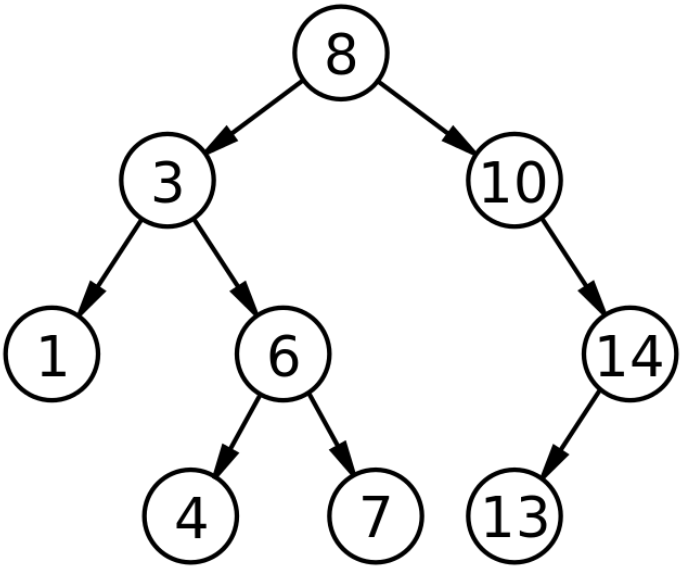
- Start at the root node of the binary tree.
- If the current node is not null, perform the following steps:
    - Visit the current node (process the node in some way):
        - Print the value of the current node or perform any operation you want on it.
    - Traverse the left subtree:
        - Call the Pre-Order Traversal function with the left child of the current node as the argument.
    - Traverse the right subtree:
        - Call the Pre-Order Traversal function with the right child of the current node as the argument.
- If the current node is null, return from the function.

- **Post-Order Traversal:** In a post-order traversal, you visit the left subtree first, then the right subtree, and finally the current node. This traversal is often used to delete the entire tree, as it ensures that child nodes are deleted before their parents.

- Algorithm:

Post-Order Traversal (Left-Right-Root):

- Start at the root node of the binary tree.
- If the current node is not null, perform the following steps:
    - Traverse the left subtree:
        - Call the Post-Order Traversal function with the left child of the current node as the argument.
    - Traverse the right subtree:
        - Call the Post-Order Traversal function with the right child of the current node as the argument.
    - Visit the current node (process the node in some way):
        - Print the value of the current node or perform any operation you want on it.
- If the current node is null, return from the function.



Binary Trees:

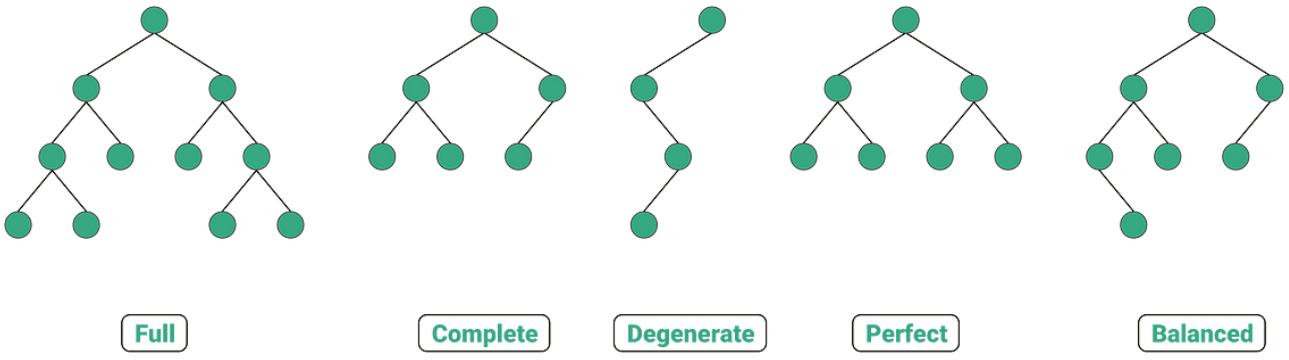
A binary tree is a hierarchical data structure composed of nodes, where each node has at most two children, referred to as the left child and the right child. The topmost node in a binary tree is called the root, and nodes with no children are called leaves. Binary trees can be used to represent various hierarchical structures, such as mathematical expressions, file systems, or decision trees.

Here are some key concepts related to binary trees:

- **Node**: Each element in a binary tree is called a node. Each node contains a value and may have references (pointers) to its left and right children.
- **Root**: The topmost node of the binary tree, from which all other nodes are accessible. A binary tree has only one root.
- **Child**: A node that has a parent is called a child node. In a binary tree, a node can have at most two children, the left child and the right child.
- **Parent**: The node which has a reference to a specific node is called the parent of that node.
- **Leaf**: A node with no children is called a leaf node (also known as external node).
- **Internal Node**: A node that has at least one child is called an internal node.
- **Subtree**: A binary tree itself can be considered as a subtree of another binary tree. A subtree is a tree formed by a node and all its descendants.

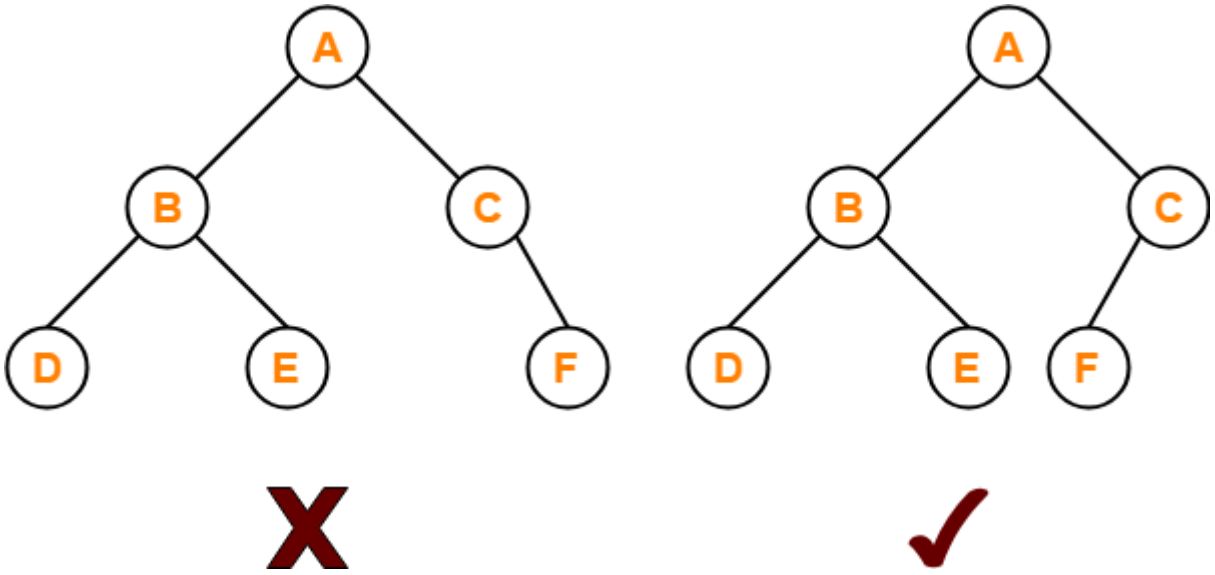
Binary trees are widely used in various algorithms and data structures. Some common types of binary trees include:

- **Binary Search Tree (BST):** A binary tree in which the left child of a node contains a value less than or equal to 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.
- **Full Binary Tree:** A binary tree in which every node has either zero or two children. There are no nodes with only one child.
- **Complete Binary Tree:** A binary tree in which all levels are filled except, possibly, the last level, and the last level's nodes are as left as possible.
- **Perfect Binary Tree:** A binary tree in which all internal nodes have two children, and all leaves are at the same level.



Binary trees provide a foundation for various tree-based algorithms, like tree traversals (In-Order, Pre-Order, Post-Order), searching, sorting, and balancing techniques, among others. They play a crucial role in computer science and are used in various applications to efficiently organize and manage data.

Complete binary trees / Almost complete binary tree (ACBT)



A Complete Binary Tree (CBT) is a special type of binary tree where all levels of the tree are fully filled, except possibly the last level, which is filled from left to right. In a complete binary tree, all nodes have either zero or two children (no node has only one child).

Key properties of a Complete Binary Tree:

- The last level of the tree may not be completely filled, but if there are any nodes in the last level, they must appear from left to right without any gaps.
- The height of a complete binary tree is always the minimum possible for the number of nodes it contains.
- If a node at depth 'd' (0-based index) is missing in a complete binary tree, then all nodes at depths greater than 'd' must also be missing.

Almost Complete Binary Tree (ACBT) is a more relaxed variant of the complete binary tree. In an Almost Complete Binary Tree, all levels are fully filled except possibly the last level, which may have some missing nodes. However, the missing nodes should only be in the last level, and they must appear from left to right without any gaps.

Both Complete Binary Trees and Almost Complete Binary Trees have applications in various data structures and algorithms. They are used in heap data structures, priority queues, and binary tree implementations where space efficiency is desired. Additionally, the properties of these trees are leveraged to perform efficient insertions and deletions while maintaining the structure.

Array implementation of ACBT java

```
import java.util.Scanner;

public class AlmostCompleteBinaryTree {
    private int[] treeArray;
    private int size;
    private int capacity;

    public AlmostCompleteBinaryTree(int capacity) {
        this.treeArray = new int[capacity];
```

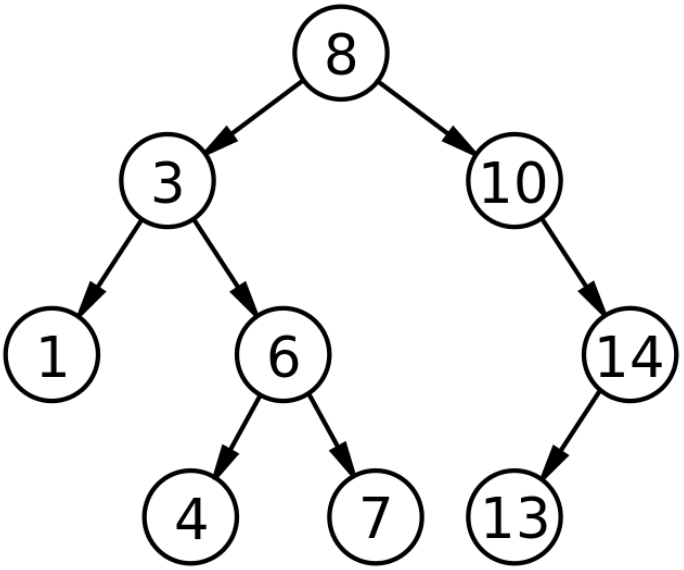
```
        this.size = 0;
        this.capacity = capacity;
    }

    public boolean isFull() {
        return size == capacity;
    }

    public void insert(int value) {
        if (isFull()) {
            System.out.println("Tree is full. Cannot insert more elements.");
            return;
        }

        treeArray[size] = value;
        size++;
    }

    public void printLevelOrder() {
        System.out.println("Level-Order Traversal:");
        for (int i = 0; i < size; i++) {
            System.out.print(treeArray[i] + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the capacity of the Almost Complete Binary Tree: ");
        int capacity = scanner.nextInt();

        AlmostCompleteBinaryTree acbt = new AlmostCompleteBinaryTree(capacity);

        System.out.println("Enter " + capacity + " elements to insert into the tree:");
        for (int i = 0; i < capacity; i++) {
            int value = scanner.nextInt();
            acbt.insert(value);
        }

        acbt.printLevelOrder();

        scanner.close();
    }
}
```

Binary Search Trees:

A Binary Search Tree (BST) is a binary tree data structure with the following properties:

- **Node Structure:** Each node in the BST has three parts:
  - A key (value) that represents the node's data.
  - A reference (pointer) to the left child node, which has a value less than the node's key.
  - A reference (pointer) to the right child node, which has a value greater than the node's key.
- **Ordering Property:** For any node in the BST, all nodes in its left subtree have keys less than the node's key, and all nodes in its right subtree have keys greater than the node's key.
- **Unique Keys:** Each key in the BST must be unique; no two nodes can have the same key.

The BST's ordering property enables efficient searching, insertion, and deletion operations. Searching in a BST can be done in O(log n) time complexity on average, making it a useful data structure for various applications.

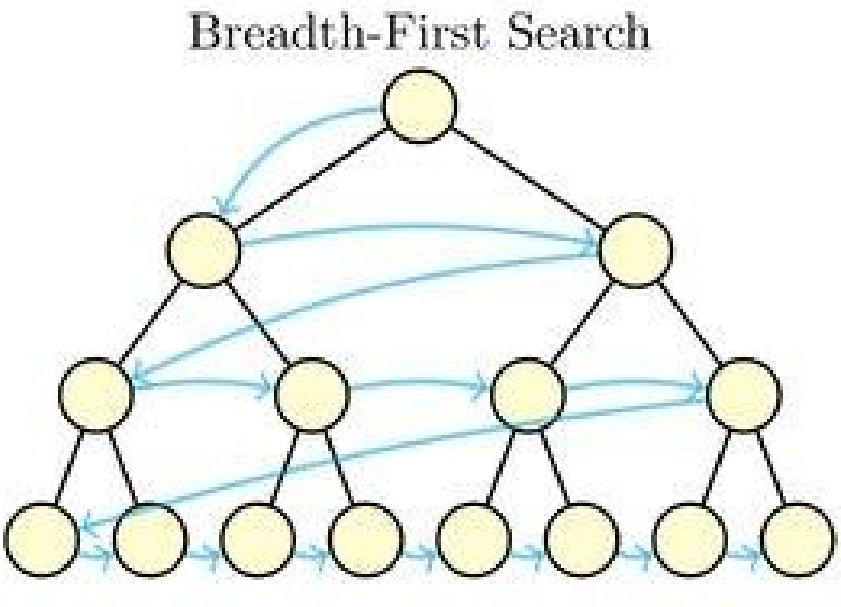


Breadth First Search:

Breadth-First Search (BFS) is a tree traversal algorithm used to explore all the nodes of a graph in breadthward motion, i.e., it visits all the nodes at the current depth level before moving on to nodes at the next depth level. BFS is often implemented using a queue data structure. It guarantees that it will visit all nodes at the same level before moving to the next level.

Let's break down the steps of the BFS algorithm:

- Initialize a queue (Q) and a set to keep track of visited nodes.
- Enqueue the starting node into the queue and mark it as visited.
- While the queue is not empty:
  - Dequeue a node from the front of the queue (the current node).
  - Process the current node (e.g., print its value or perform any desired operations).
  - Enqueue all unvisited neighbor nodes of the current node into the queue and mark them as visited.



By following these steps, BFS ensures that all nodes in the tree are visited in breadth-first order, level by level.

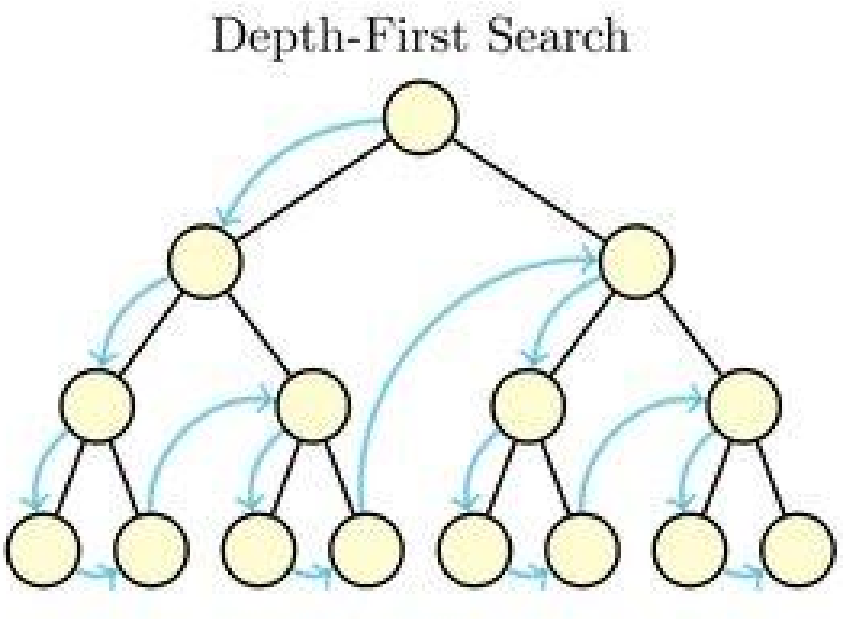
Depth First Search - DFS

A DFS tree (Depth-First Search tree) is a tree traversal technique. The DFS tree represents the exploration of the tree in depth-first order, starting from a chosen source node.

Here's how the DFS algorithm works and how it constructs the DFS tree:

1. Start the DFS from a chosen source node (sometimes referred to as the root of the DFS tree).
2. Mark the source node as visited.
3. Explore the adjacent unvisited nodes of the current node recursively by performing a depth-first search.
4. While exploring the adjacent nodes, for each unvisited neighbour node, repeat steps 1 to 3.
5. Once all possible paths from the source node have been explored, the DFS tree is constructed.

The DFS tree can be represented using a parent-pointer representation or an adjacency list representation.



AVL tree

An AVL tree is a self-balancing binary search tree (BST) named after its inventors, Adelson-Velsky and Landis. The primary goal of an AVL tree is to maintain its height balance, ensuring that the height difference between the left and right subtrees of any node (called the balance factor) is at most 1.

In an AVL tree, the balance factor of any node is defined as follows:
Balance Factor = Height of left subtree - Height of right subtree

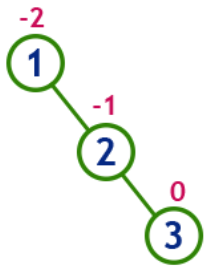
The balance factor of a node can be -1, 0, or 1 for the tree to be balanced. If the balance factor becomes greater than 1 or less than -1, the tree becomes unbalanced and requires rebalancing.

To maintain the balance, AVL trees perform rotations during insertion and deletion operations to keep the height balanced and guarantee efficient search, insertion, and deletion operations, all in O(log n) time complexity.

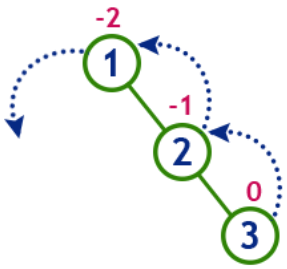
There are four types of rotations used in AVL trees:

- Left Rotation: Used to balance the tree when the left subtree is taller than the right subtree by 2.

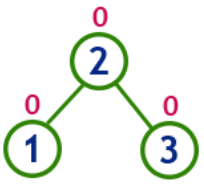
insert 1, 2 and 3



Tree is imbalanced



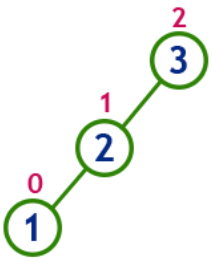
To make balanced we use LL Rotation which moves nodes one position to left



After LL Rotation Tree is Balanced

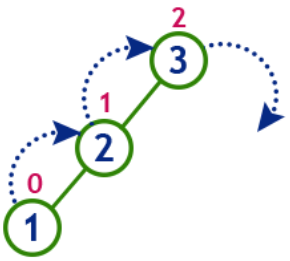
- Right Rotation: Used to balance the tree when the right subtree is taller than the left subtree by 2.

insert 3, 2 and 1

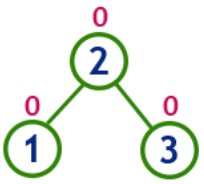


Tree is imbalanced

because node 3 has balance factor 2



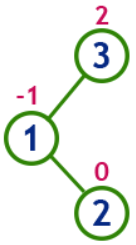
To make balanced we use RR Rotation which moves nodes one position to right



After RR Rotation Tree is Balanced

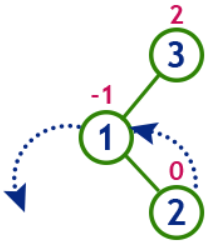
- Left-Right Rotation: A combination of left rotation followed by right rotation.

insert 3, 1 and 2

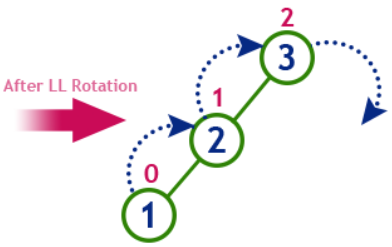


Tree is imbalanced

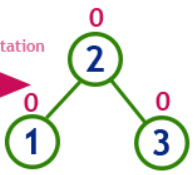
because node 3 has balance factor 2



LL Rotation

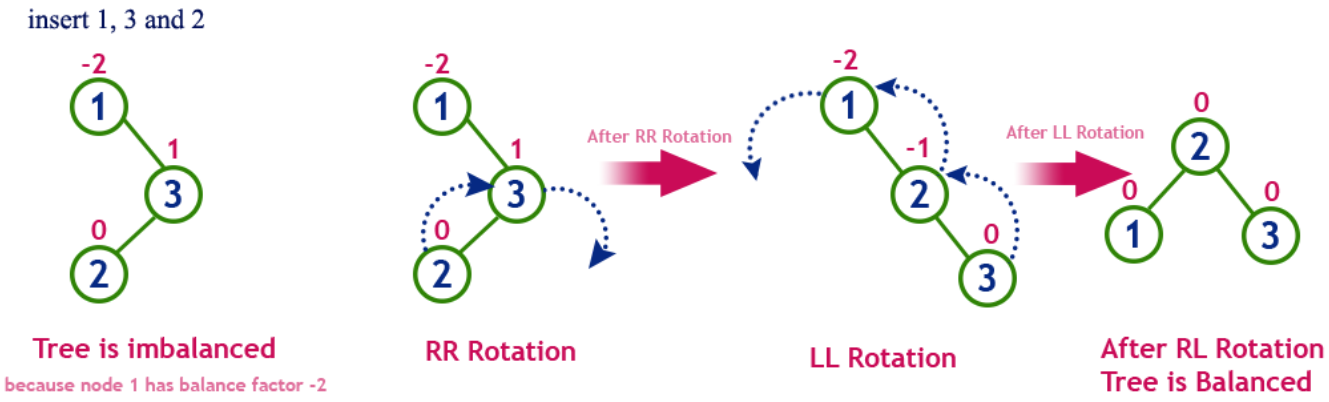


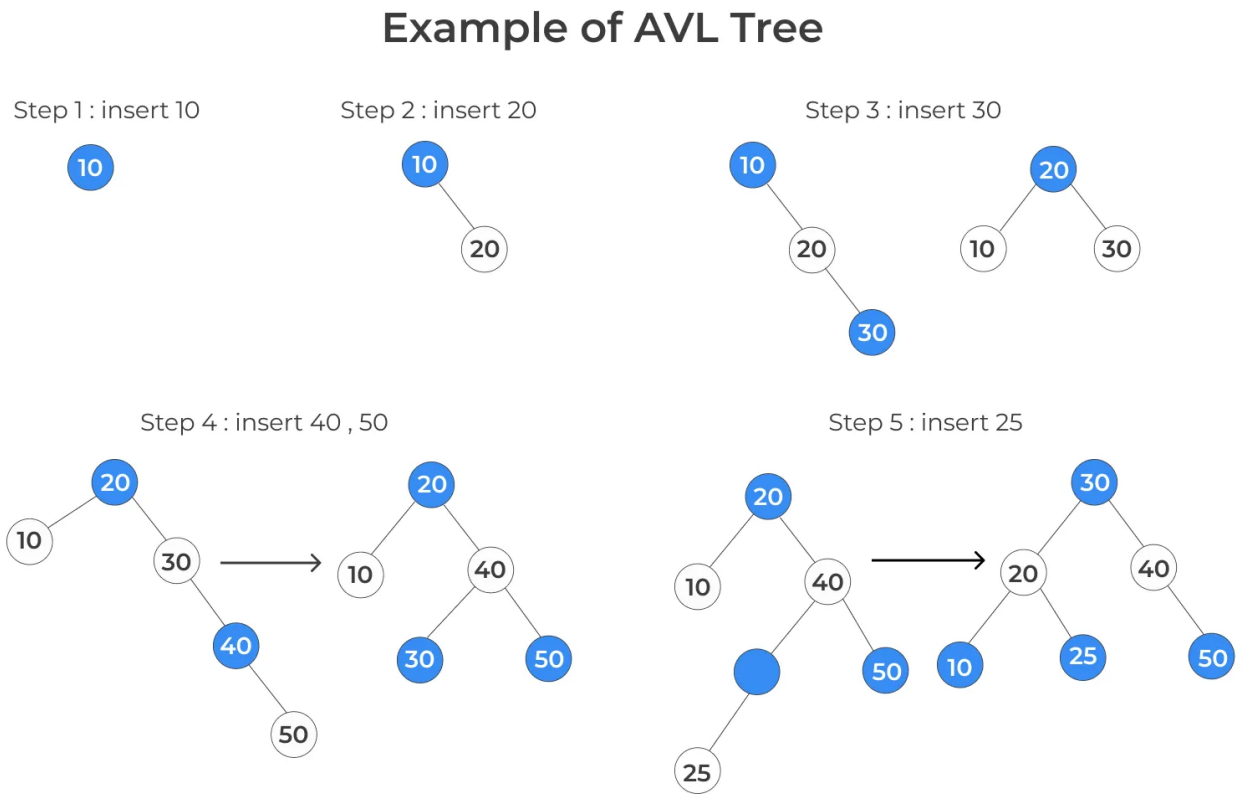
RR Rotation



After LR Rotation Tree is Balanced

- Right-Left Rotation: A combination of right rotation followed by left rotation.





The AVL tree algorithm ensures that after each insertion or deletion, the balance factors of all nodes along the path to the root are checked and rotations are performed as needed to maintain the AVL tree properties.

The balancing property of AVL trees guarantees that the height of the tree remains logarithmic with respect to the number of nodes, ensuring efficient search, insertion, and deletion operations.

However, the additional balancing operations in AVL trees come with increased overhead during insertion and deletion compared to regular binary search trees like a standard BST. In cases where there are more frequent read operations and fewer writes or updates, AVL trees are a good choice due to their self-balancing property and faster search times.

Some of the most common applications of AVL trees include database indexing, maintaining ordered datasets, and in cases where a self-balancing binary search tree is required to guarantee efficient performance in the worst-case scenario.