



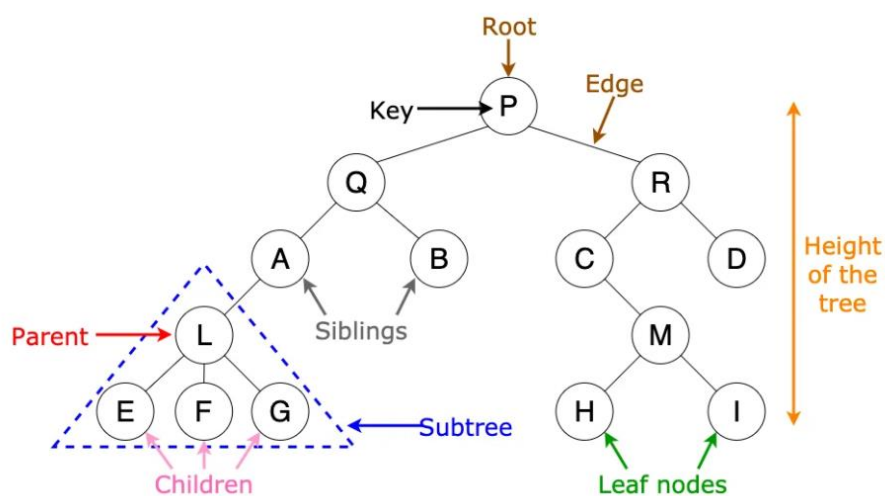
Islamic University of Gaza
Faculty of Engineering
Computer Engineering
Department



ECOM 3001

Data Structures and Algorithms Lab [Fall 2024]

Lab "07" Topic: Trees



Prepared by:
Eng. Waseem Alfarram

Supervised by:
Dr. Ahmad Mahdi

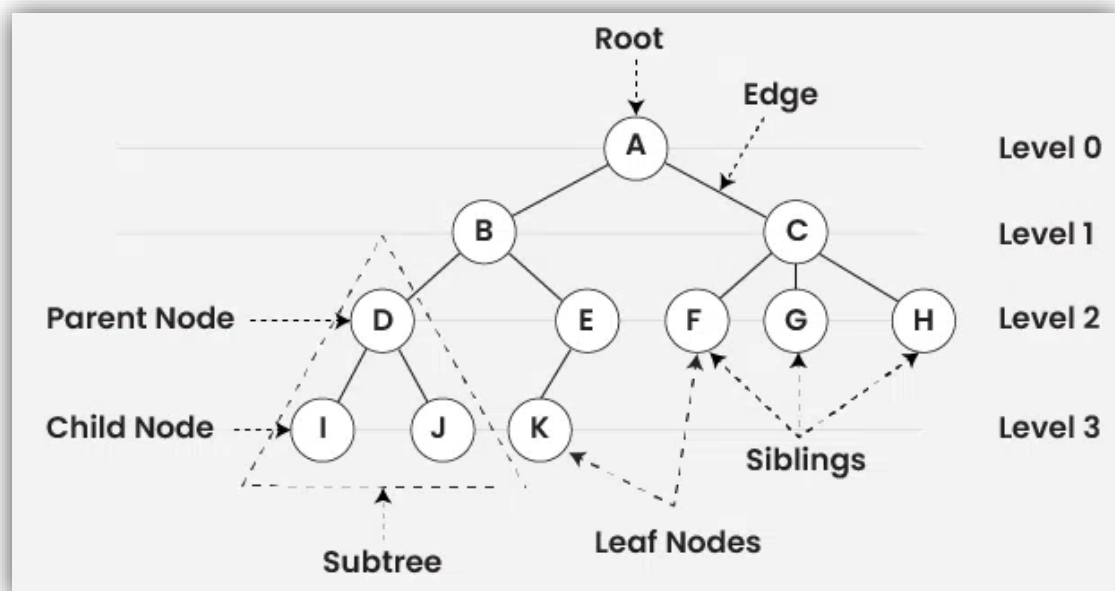
Lab Objectives:

- To define **Tree** data structure and Recognize some **Terminologies** and **Properties** about it.
- To distinguish the **Types of Tree** data structure (**based on number of children**): Binary Tree , Ternary Tree and N-ary Tree.
- To go deeper in the **Types of the Binary Tree** based on: the **Number of Children** (Full , Degenerate and Skewed) and based on the **Completion of Levels** (Complete , Perfect and Balanced).
- To get familiar with some **Important Special Types of Binary Trees** like Binary Search Tree, AVL Tree and Red-Black Tree.
- To **implement the Binary Tree** using Java.
- To distinguish between different **Binary Tree Traversal methods** like **DFS** (preorder , inorder and postorder) and **BFS**.
- To **implement the Binary Tree Traversal methods** using Java.
- To get familiar more with the **Binary Search Tree**, its **Applications** and its **Implementation** in Java.
- To recognize some **Advantages and Disadvantages of the Tree** data structure.
- To recognize some **Applications of the Tree** data structure in real-world scenarios.
- To **solve some Exercises about Tree** data structure to deepen your understanding for this lab topics.

What is Tree Data Structure?

A **Tree** data structure is a hierarchal (**non-linear**) data structure in which a collection of elements known as nodes are connected to each other via edges such that there exists exactly one path between any two nodes. The data in the Tree are not stored in a sequential manner i.e., they are not stored linearly. Instead, they are arranged on multiple levels or we can say it is a hierarchical structure. For this reason, the tree is considered to be a non-linear data structure.

The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.



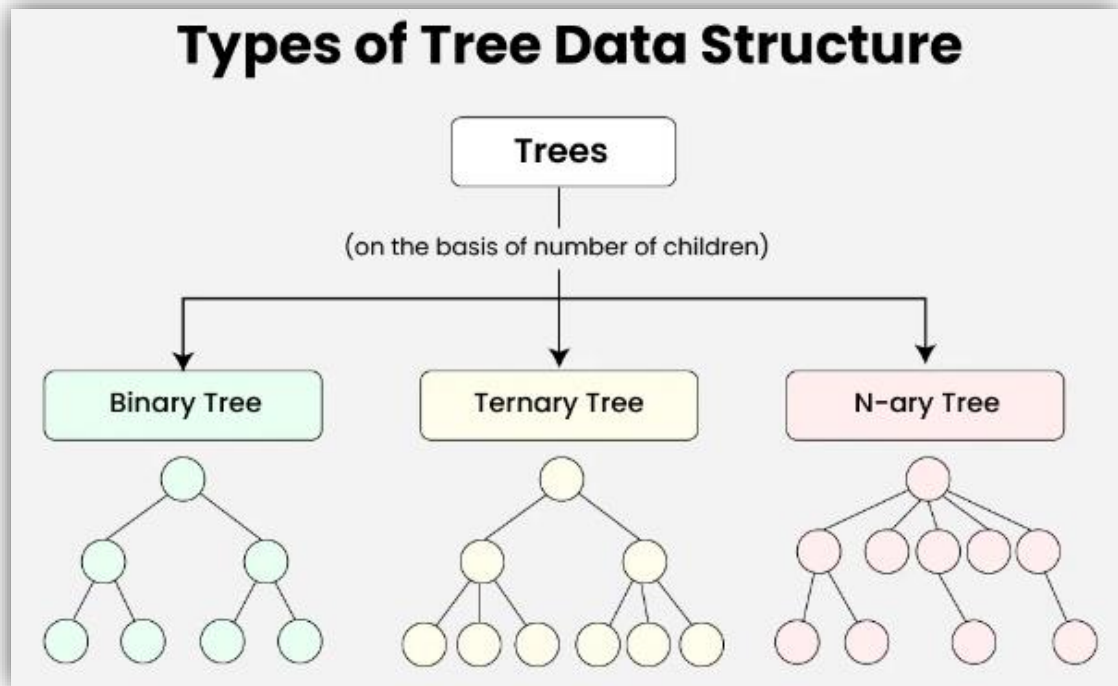
➤ Terminologies in Tree Data Structure:

1. **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {B} is the parent node of {D, E}.
2. **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {D, E} are the child nodes of {B}.
3. **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
4. **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {I, J, K, F, G, H} are the leaf nodes of the tree.
5. **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {A, B} are the ancestor nodes of the node {E}.
6. **Descendant:** A node x is a descendant of another node y if and only if y is an ancestor of x.
7. **Sibling:** Children of the same parent node are called siblings. {D, E} are called siblings.
8. **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.
9. **Internal node:** A node with at least one child is called Internal Node.
10. **Neighbor of a Node:** Parent or child nodes of that node are called neighbors of that node.
11. **Subtree:** Any node of the tree along with its descendant.

➤ Properties of Tree Data Structure:

- **Number of edges:** An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have $(N-1)$ edges. There is only one path from each node to any other node of the tree.
- **Depth of a node:** The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.
- **Height of a node:** The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.
- **Height of the Tree:** The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.
- **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be 0. The degree of a tree is the maximum degree of a node among all the nodes in the tree.

➤ Types of Tree Data Structure:



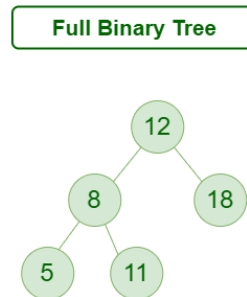
1. **Binary Tree:** In a binary tree, each node can have a maximum of two children linked to it. Some common types of binary trees include **full binary trees**, **complete binary trees**, **balanced binary trees**, and **degenerate** or **pathological binary trees**.
2. **Ternary Tree:** A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.
3. **N-ary Tree or Generic Tree:** Generic trees are a collection of nodes where each node is a data structure that consists of records (data) and a list of references to its children (duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

➤ Types of Binary Tree:

- **Based on the Number of Children:**

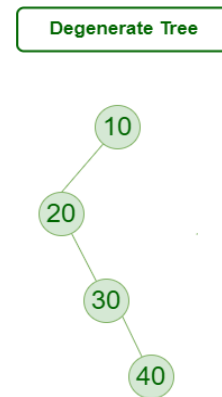
1. **Full (Proper) Binary Tree:**

A Binary Tree is a full binary tree if every node has 0 or 2 children. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.



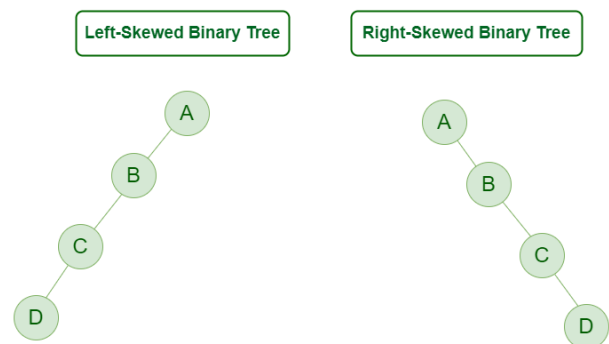
2. **Degenerate (Pathological) Binary Tree:**

It is a type of binary tree in which every internal node has one child. Such trees are performance-wise same as linked list. A degenerate or pathological tree is a tree having a single child either left or right.



3. **Skewed Binary Trees:**

It is a type of binary tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.

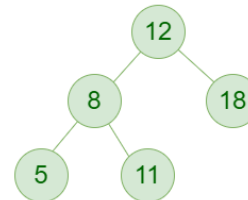


- **Based on the Completion of Levels:**

1. **Complete Binary Tree:**

It is a type of binary tree in which all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.

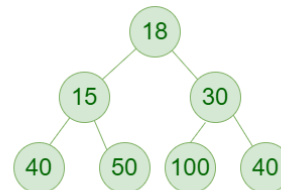
Complete Binary Tree



2. **Perfect Binary Tree:**

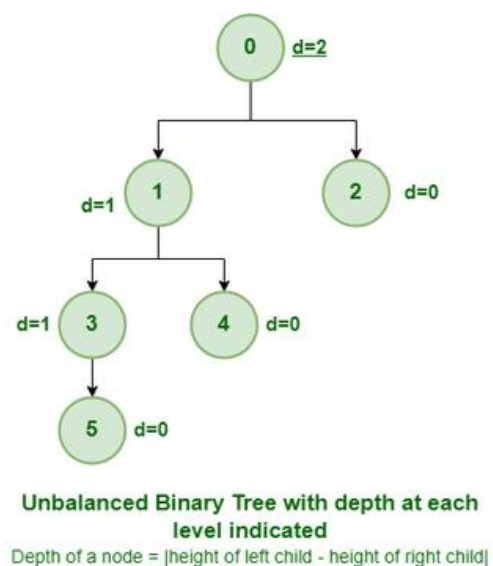
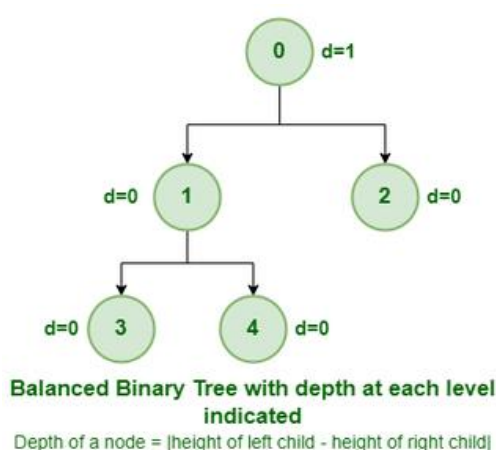
It is a type of binary tree in which all the internal nodes have two children and all leaf nodes are at the same level.

Perfect Binary Tree

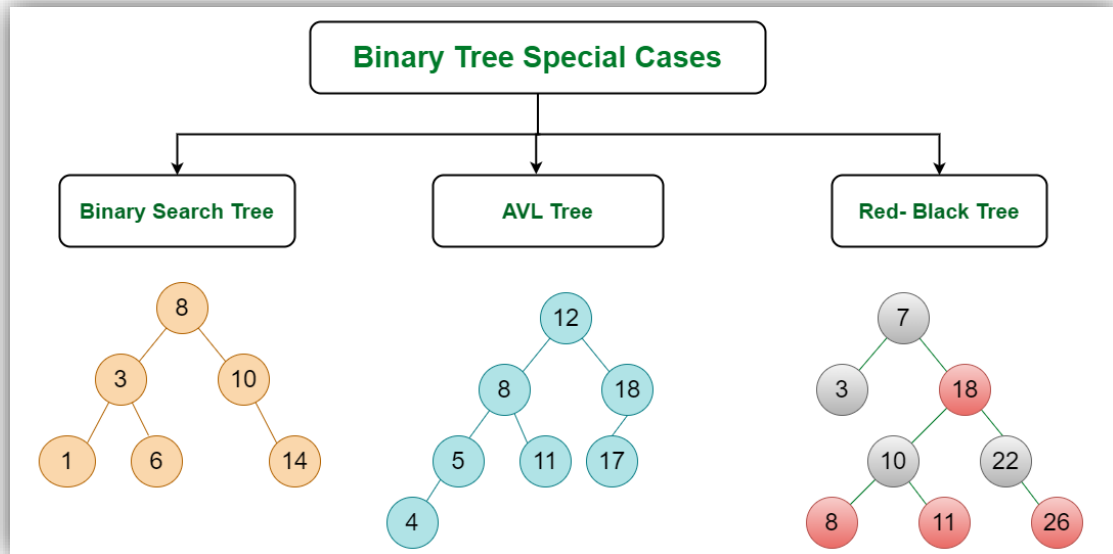


3. **Balanced Binary Tree:**

It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.



➤ Important Special Types of Binary Trees:



1. Binary Search Tree:

It is a node-based binary tree data structure that has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

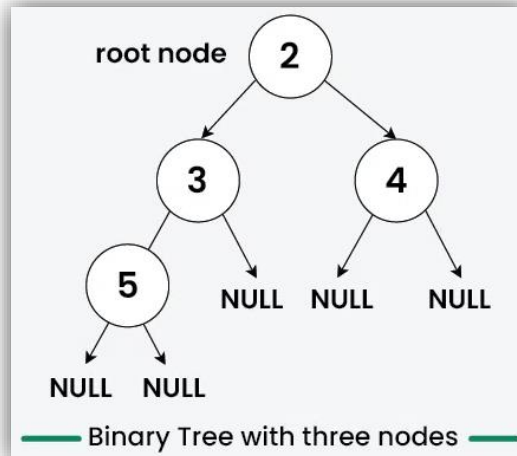
2. AVL Tree (Balanced Binary Search Tree):

It is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

3. Red-Black Tree:

It is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions.

➤ Binary Tree Implementation in Java:



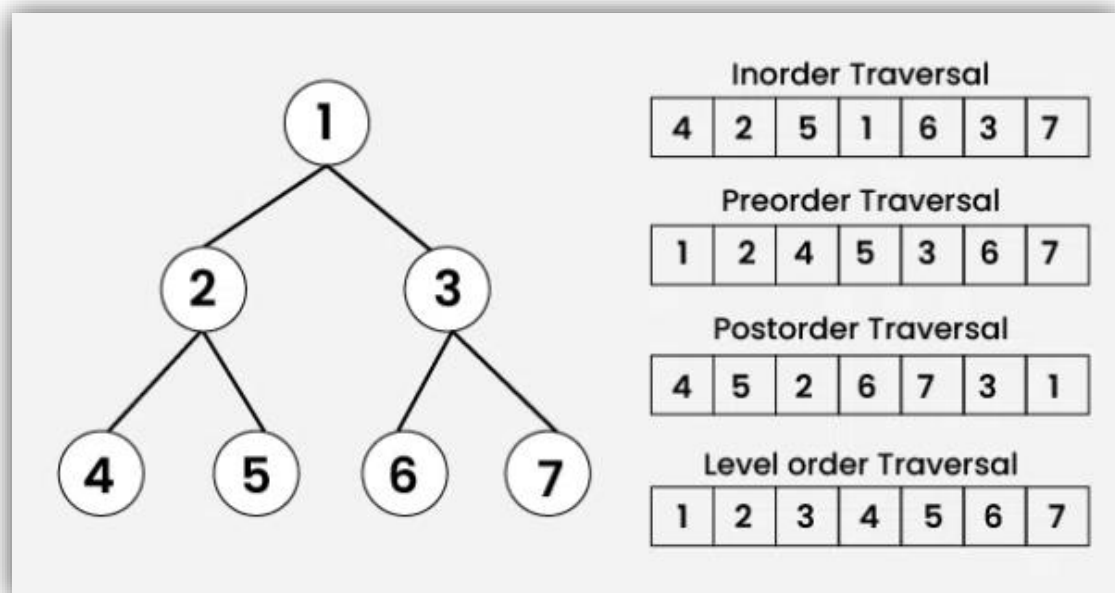
```
class Node {
    int data;
    Node left, right;
    Node(int d) {
        data = d;
        left = null;
        right = null;
    }
}

class BinaryTree {
    public static void main(String[] args) {
        // Initialize and allocate memory for tree nodes
        Node root = new Node(2);
        Node firstNode = new Node(3);
        Node secondNode = new Node(4);
        Node thirdNode = new Node(5);

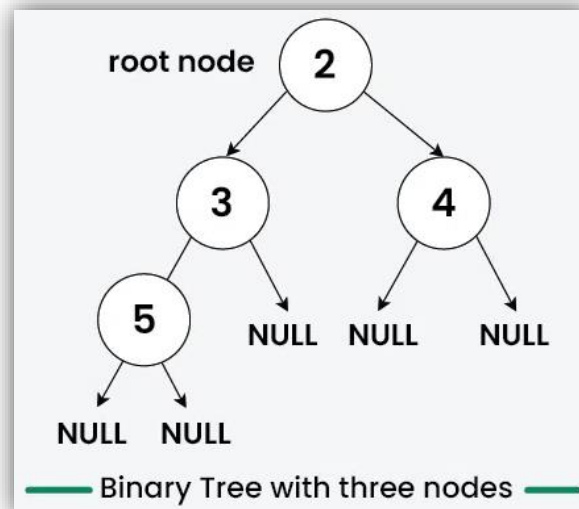
        // Connect binary tree nodes
        root.left = firstNode;
        root.right = secondNode;
        firstNode.left = thirdNode;
    }
}
```

➤ Binary Tree Traversal:

1. **Depth-First Search (DFS) Algorithms:** DFS explores as far down a branch as possible before backtracking. It is implemented using recursion. The main traversal methods in DFS for binary trees are:
 - ❖ **Preorder Traversal (current-left-right):**
Visits the node first, then left subtree, then right subtree.
 - ❖ **Inorder Traversal (left-current-right):**
Visits left subtree, then the node, then the right subtree.
 - ❖ **Postorder Traversal (left-right-current):**
Visits left subtree, then right subtree, then the node.
2. **Breadth-First Search (BFS) Algorithm:** BFS explores all nodes at the present depth before moving on to nodes at the next depth level. It is typically implemented using a queue. BFS in a binary tree is commonly referred to as **Level Order Traversal**.



➤ Binary Tree Traversal Implementations in Java:



```
import java.util.LinkedList;
import java.util.Queue;

class Node {
    int data;
    Node left, right;

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

// BinaryTree class
class BinaryTree {
    // In-order DFS: Left, Root, Right
    static void inOrderDFS(Node node) {
        if (node == null)
            return;
        inOrderDFS(node.left);
        System.out.print(node.data + " ");
        inOrderDFS(node.right);
    }

    // Pre-order DFS: Root, Left, Right
    static void preOrderDFS(Node node) {
        if (node == null)
            return;
        System.out.print(node.data + " ");
        preOrderDFS(node.left);
        preOrderDFS(node.right);
    }
}
```

```

// Post-order DFS: Left, Right, Root
static void postOrderDFS(Node node) {
    if (node == null)
        return;
    postOrderDFS(node.left);
    postOrderDFS(node.right);
    System.out.print(node.data + " ");
}

// BFS: Level order traversal
static void BFS(Node root) {
    if (root == null)
        return;
    Queue<Node> queue = new LinkedList<>();
    queue.add(root);
    while(!queue.isEmpty()) {
        Node node = queue.poll();
        System.out.print(node.data + " ");
        if (node.left != null)
            queue.add(node.left);
        if (node.right != null)
            queue.add(node.right);
    }
}

public static void main(String[] args) {
    // Creating the tree
    Node root = new Node(2);
    root.left = new Node(3);
    root.right = new Node(4);
    root.left.left = new Node(5);

    System.out.print("In-order DFS: ");
    inOrderDFS(root);

    System.out.print("\nPre-order DFS: ");
    preOrderDFS(root);

    System.out.print("\nPost-order DFS: ");
    postOrderDFS(root);

    System.out.print("\nLevel order (BFS): ");
    BFS(root);
}

```

Output:

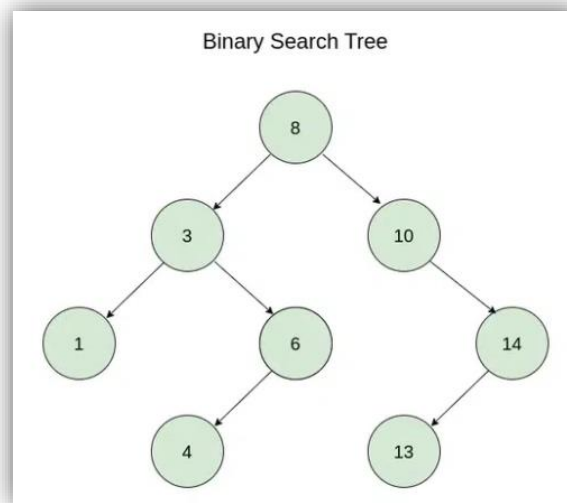
In-order DFS: 5 3 2 4

Pre-order DFS: 2 3 5 4

Post-order DFS: 5 3 4 2

Level order (BFS): 2 3 4 5

➤ Binary Search Tree:

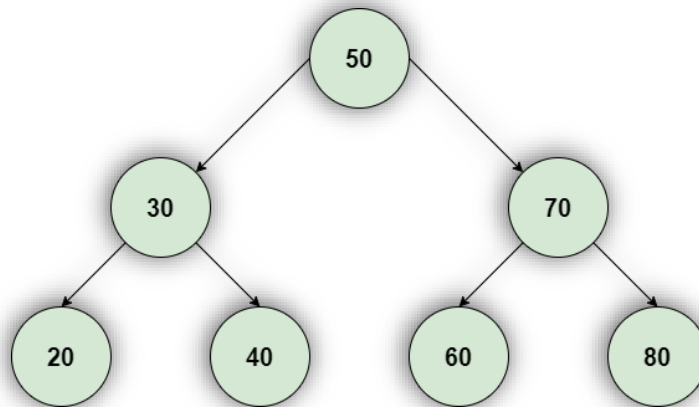


A **Binary Search Tree (BST)** is organized as a hierarchical structure where each node contains the **key value** and **two pointers** to the **left** and **right** children. The left child contains keys **less than the parent node's key** and the right child key contains keys **greater than the parent node's key**. This hierarchical organization allowed for efficient operations such as **searching, insertion, and deletion** operations in an average-case time complexity of **$O(\log n)$** .

➤ Applications of the Binary Search Tree (BST):

- Symbol Tables and Databases.
- Auto-Complete and Spell Checking.
- Binary Search.
- File System Organization.
- Dynamic Sets and Ordered Data.
- Range Queries and Interval Trees.
- Priority Queues.
- Optimization Algorithms.

➤ Binary Search Tree (BST) Implementation in Java:



```
import java.util.LinkedList;
import java.util.Queue;

// Java Program to Implement BST
class Node {
    int key;
    Node left, right;
    public Node(int item) {
        key = item;
        left = right = null;
    }
}

class BinarySearchTree {
    Node root;
    public BinarySearchTree() {
        root = null;
    }

    // Insertion operation
    void insert(int key) {
        root = insertHelper(root, key);
    }

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

    // Deletion operation
    void delete(int key) {
```

```

        root = deleteHelper(root, key);
    }

    Node deleteHelper(Node root, int key) {
        if (root == null)
            return root;
        if (key < root.key)
            root.left = deleteHelper(root.left, key);
        else if (key > root.key)
            root.right = deleteHelper(root.right, key);
        else {
            if (root.left == null)
                return root.right;
            else if (root.right == null)
                return root.left;
            root.key = minValue(root.right);
            root.right = deleteHelper(root.right, root.key);
        }
        return root;
    }

    int minValue(Node root) {
        int minVal = root.key;
        while (root.left != null) {
            minVal = root.left.key;
            root = root.left;
        }
        return minVal;
    }

    // Search operation
    boolean search(int key) {
        return searchHelper(root, key);
    }

    boolean searchHelper(Node root, int key) {
        if (root == null)
            return false;
        if (root.key == key)
            return true;
        if (root.key < key)
            return searchHelper(root.right, key);
        return searchHelper(root.left, key);
    }

    // Inorder traversal
    void inOrder() {
        inOrderHelper(root);
        System.out.println("\n");
    }

    void inOrderHelper(Node root) {
        if (root != null) {
            inOrderHelper(root.left);
            System.out.print(root.key + " ");
            inOrderHelper(root.right);
        }
    }

```

```

    }
}

// Preorder traversal
void preOrder() {
    preOrderHelper(root);
    System.out.println("\n");
}

void preOrderHelper(Node root) {
    if (root != null) {
        System.out.print(root.key + " ");
        preOrderHelper(root.left);
        preOrderHelper(root.right);
    }
}

// Postorder traversal
void postOrder() {
    postOrderHelper(root);
    System.out.println("\n");
}

void postOrderHelper(Node root) {
    if (root != null) {
        postOrderHelper(root.left);
        postOrderHelper(root.right);
        System.out.print(root.key + " ");
    }
}

// Level order traversal
void levelOrder() {
    levelOrderHelper(root);
    System.out.println("\n");
}

void levelOrderHelper(Node root) {
    if (root == null)
        return;
    Queue<Node> queue = new LinkedList<>();
    queue.add(root);
    while(!queue.isEmpty()) {
        Node node = queue.poll();
        System.out.print(node.key + " ");
        if (node.left != null)
            queue.add(node.left);
        if (node.right != null)
            queue.add(node.right);
    }
}
}

// Driver Class
public class test {

```

```

// Main Function
public static void main(String[] args) {
    BinarySearchTree tree = new BinarySearchTree();

    // Inserting elements
    tree.insert(50);
    tree.insert(30);
    tree.insert(20);
    tree.insert(40);
    tree.insert(70);
    tree.insert(60);
    tree.insert(80);

    System.out.println("Inorder traversal:");
    tree.inOrder();

    // Deleting elements
    tree.delete(20);
    tree.delete(30);
    System.out.println("Inorder traversal after deletion:");
    tree.inOrder();

    // Searching for an element
    int searchKey = 70;
    System.out.println("Is " + searchKey + " present in the tree?
" + tree.search(searchKey) + "\n");

    // Traversals
    System.out.println("Preorder traversal:");
    tree.preOrder();
    System.out.println("Postorder traversal:");
    tree.postOrder();
    System.out.println("Level order traversal:");
    tree.levelOrder();
}
}

```

Output:

Inorder traversal:

20 30 40 50 60 70 80

Inorder traversal after deletion:

40 50 60 70 80

Is 70 present in the tree? True

Preorder traversal:

50 40 70 60 80

Postorder traversal:

40 60 80 70 50

Level order traversal:

50 40 70 60 80

➤ Advantages of Tree Data Structure:

- Tree offers **Efficient Searching** Depending on the type of tree, with average search times of **$O(\log n)$** for balanced trees like AVL.
- Trees provide a hierarchical representation of data, making it **easy to organize and navigate** large amounts of information.
- The recursive nature of trees makes them **easy to traverse and manipulate** using recursive algorithms.

➤ Disadvantages of Tree Data Structure:

- Unbalanced Trees, meaning that the height of the tree is skewed towards one side, which can lead to **inefficient search times**.
- Trees demand **more memory space requirements** than some other data structures like arrays and linked lists, especially if the tree is very large.
- The implementation and **manipulation of trees can be complex** and require a good understanding of the algorithms.

➤ Applications of Tree Data Structure:

- **File System**: This allows for efficient navigation and organization of files.
- **Data Compression**: **Huffman coding** is a popular technique for data compression that involves constructing a binary tree where the leaves represent characters and their frequency of occurrence. The resulting tree is used to encode the data in a way that minimizes the amount of storage required.
- **Compiler Design**: In compiler design, a syntax tree is used to represent the structure of a program.
- **Database Indexing**: B-trees and other tree structures are used in database indexing to efficiently search for and retrieve data.

Leetcode Exercises:

- <https://leetcode.com/problems/same-tree/>
- <https://leetcode.com/problems/sum-of-left-leaves/>
- <https://leetcode.com/problems/balanced-binary-tree/>
- <https://leetcode.com/problems/binary-tree-zigzag-level-order-traversal/>
- <https://leetcode.com/problems/validate-binary-search-tree/>

End of Lab 😊