# Trees

A **Tree** is a **non-linear data structure** that represents elements in a hierarchical relationship.  
It’s a **collection of nodes**, where each node can have **0 or more child nodes**.

Key property:  
There’s exactly **one path** between the **root node** and any other node in the tree.

## Terminology (must know)

A screenshot of a computer

AI-generated content may be incorrect.

| **Term** | **Meaning** | **Example** |
| --- | --- | --- |
| **Node** | Each element in the tree | A, B, C, D, E, F |
| **Root** | Top-most node (no parent) | A |
| **Parent** | Node that has children | A, B, C |
| **Child** | Node that has a parent | B, C are children of A |
| **Leaf** | Node with no children | D, E, F |
| **Edge** | Connection between two nodes | A–B, A–C |
| **Siblings** | Nodes with same parent | B & C, D & E |
| **Path** | Sequence of nodes to reach a target | A→B→E |
| **Height** | Longest path from root to a leaf | Height(A)=2 (A→B→D) |
| **Depth** | Distance from root to that node | Depth(D)=2 |
| **Degree** | Number of children a node has | Degree(A)=2 |

## Why use Trees?

| **Feature** | **Advantage** |
| --- | --- |
| **Hierarchical Data Representation** | Perfect for file systems, XML/JSON, org charts |
| **Fast Search/Insertion (BST)** | O(log n) in balanced trees |
| **Efficient Hierarchical Queries** | Used in databases, AI decision trees, compiler parsing |

## Types of Trees (Basics)

1. **General Tree** – any node can have any number of children
2. **Binary Tree** – each node can have at most **two children** (left and right)
3. **Binary Search Tree (BST)** – Binary tree with rule: left < root < right
4. **Full Binary Tree** – every node has 0 or 2 children
5. **Complete Binary Tree** – all levels filled except last, filled left to right
6. **Perfect Binary Tree** – all internal nodes have 2 children, and all leaves at same level

## Binary Tree Representations

## A diagram of a tree AI-generated content may be incorrect.

There are **two main ways** to represent a tree in memory:

| **Representation** | **Used In** | **Example** |
| --- | --- | --- |
| **Linked Representation** | General trees, Binary trees | Uses Node objects (like your Node { val, left, right }) |
| **Array Representation** | Complete / Full Binary Trees | Used in **Heaps** or perfect binary trees |

## Array Representation — The Core Idea

If you store nodes **level by level (top to bottom, left to right)** in an array,  
then there’s a simple **mathematical relationship** between parent and child indices.

Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Value: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

## Note on 0-based Indexing (Used in Java Arrays)

If your array starts at **index 0**, formulas change slightly:

| **Node Relationship** | **Formula**  **(1-based indexing** | **Formula**  **(0-based indexing)** | **Example** |
| --- | --- | --- | --- |
| **Left Child** | 2 \* i | 2 \* i + 1 | Node at index 0 → left = 1 |
| **Right Child** | 2 \* i + 1 | 2 \* i + 2 | Node at index 0 → right = 2 |
| **Parent** | (i) / 2 | (i - 1) / 2 | Node at index 4 → parent = 1 |



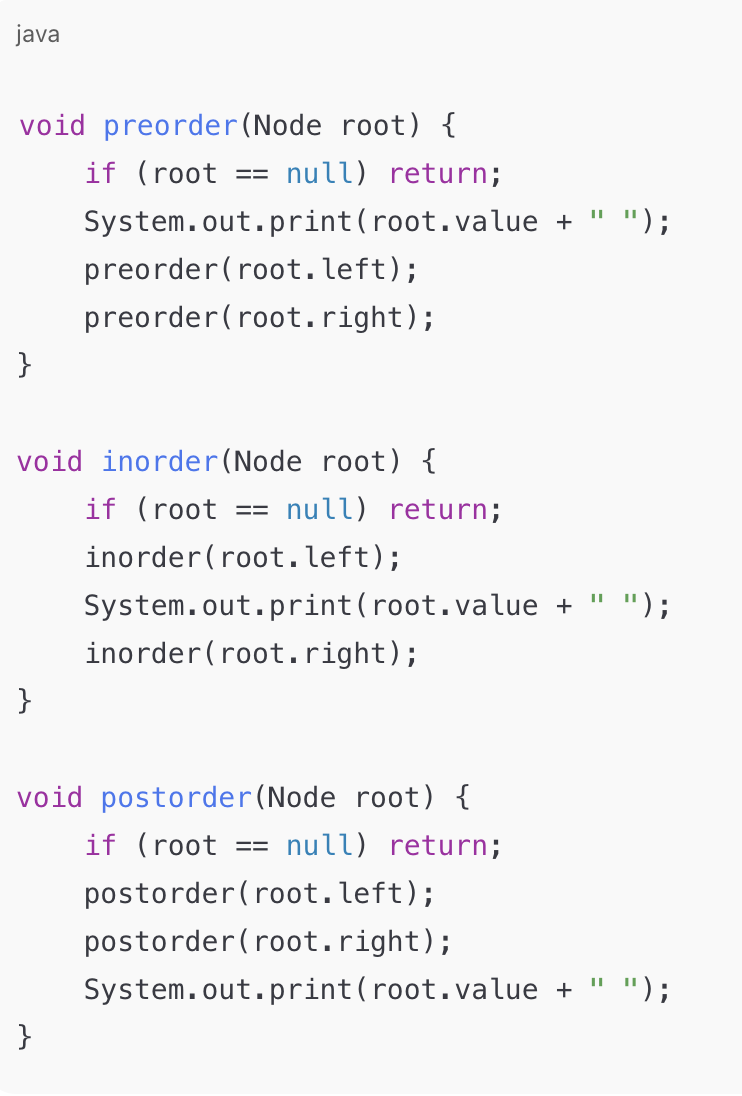
## Example 4: Tree Traversals (Recursive) (TreeTraversal.java)

* Prints Root , Then Left subtree , Right subtree

| **Type** | **Order** | **Description** |
| --- | --- | --- |
| **Preorder** | Root → Left → Right | Used to copy tree |
| **Inorder** | Left → Root → Right | Gives sorted order in BST |
| **Postorder** | Left → Right → Root | Used to delete tree |

| **Aspect** | **DFS (Depth-First Search)** | **BFS (Breadth-First-Search)** |
| --- | --- | --- |
| **Traversal order** | Go **deep** into a branch before exploring siblings | Explore **level by level** |
| **Data structure used** | **Stack** (or recursion) | **Queue** |
| **Use case** | When depth/path matters (like backtracking, tree diameter, etc.) | When shortest path or level matters |
| **Traversal Example (for Binary Tree)** | Preorder / Inorder / Postorder are DFS variants | Level-order traversal is BFS |

A screenshot of a computer

AI-generated content may be incorrect.

# Binary Seach Tree

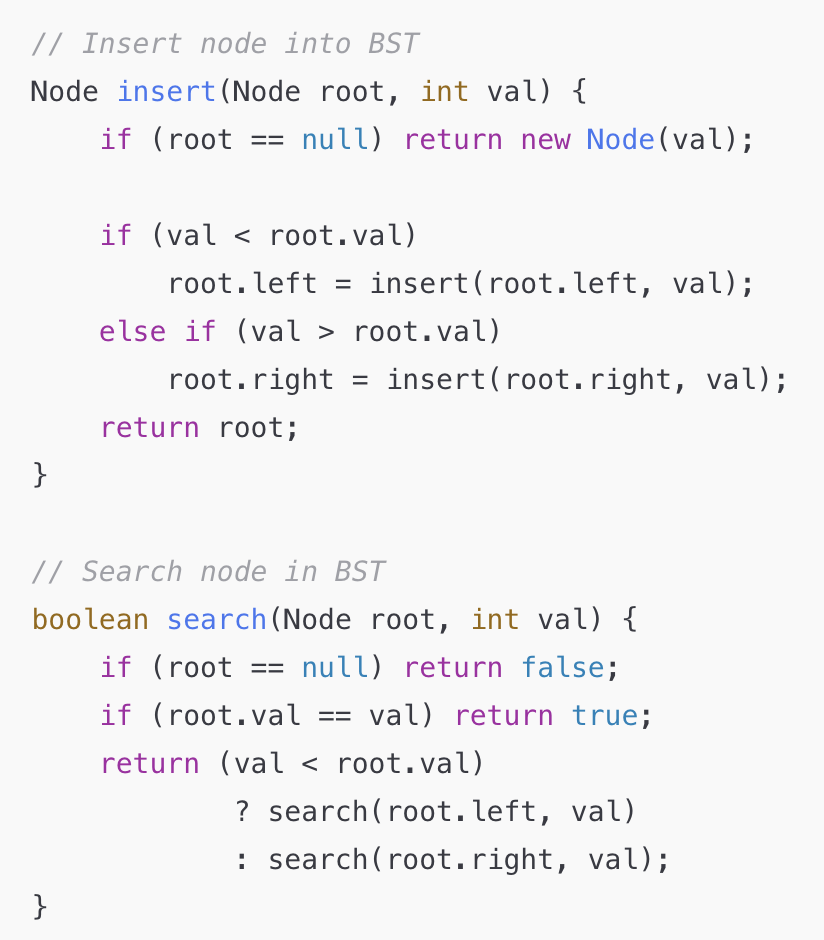
A **Binary Search Tree** is a **binary tree** where:

* For every **node N**:
  + All **values in its left subtree** are **less than N.val**.
  + All **values in its right subtree** are **greater than N.val**.
* **No duplicate values** are allowed (in the standard definition).

## Properties of BST

| **Property** | **Explanation** |
| --- | --- |
| **Ordering** | Left < Root < Right |
| **Inorder Traversal** | Always gives **sorted order** |
| **Search Complexity** | O(h) → h = tree height (best O(log n), worst O(n)) |
| **Insert / Delete** | Also O(h) |
| **Balanced BST** | Keeps h ≈ log₂(n) ⇒ efficient (e.g., AVL, Red-Black Trees) |

A triangle with numbers and letters

AI-generated content may be incorrect.

## Real-Time Use Cases of BSTs

| **Application** | **Why BST?** |
| --- | --- |
| **Databases (Indexing)** | Search faster by ordered keys |
| **Symbol Tables (Compilers)** | Fast insert/search of variables |
| **Auto-complete / Dictionary lookups** | Predictive searching |
| **Range Search / Ranking Systems** | Efficient min–max queries |
| **Sets & Maps in Java (TreeSet, TreeMap)** | Internally Red-Black BSTs |