TREES AND HIERARCHICAL STRUCTURES

Binary Trees and Binary Search Trees

Binary Tree

It allows data to be organized hierarchically, making it valuable in scenarios where data relationships exist. A binary tree can have up to two (2) child nodes per parent, providing flexibility in structuring data.

Every node acts either like a parent node or a child node. The topmost node of this tree is the **root node**. Every **parent node** may have only two (2) children. They are commonly called as left child/**child node** and the right child/**child node**.

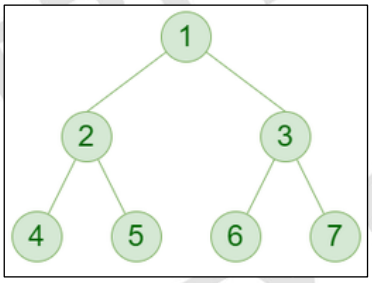
A node in the binary tree may have three (3) fields:

- **Data Element** – functions as a storage for the data value that the node needs to store.
- **Pointer to the Left Child** – holds the address (or the reference to) the left child node.
- **Pointer to the Right Child** – holds the address (or the reference to) the right child node.

The **end nodes** or **leaf nodes** have no children, and a NULL pointer points to the left and right children of a leaf node.

In context, binary trees can be used in databases, such as organizing files into folders. The node, in this case the folder, represents a record, and this setup makes it easier to find specific records. It is like having a well-organized filing system for the information.

For example:



In the example, each node has either 0, 1, or 2 children. Therefore, this tree is a binary tree.

Tree Traversal

It is the process of visiting every node in a tree exactly once in a particular order to access or process data. It allows searching, printing, and manipulating tree data, and is essential in algorithms such as expression evaluation, file systems, and decision-making.

Linear data structures such as arrays, stacks, queues, and linked lists have only one way to read data, but a hierarchical data structure like a tree can be traversed in different ways.

Tree traversal methods are generally categorized into two (2) primary types: **Breadth-First Traversal (BFT)** and **Depth-First Traversal (DFT)**. Each type represents a systematic way to visit all the nodes in a tree structure.

In **Breadth-First Traversal**, also known as **Level Order Traversal**, visits the nodes level by level from the root downwards, moving horizontally across each level before descending to the next. This is implemented using a queue and is especially useful in applications that require processing of nodes based on their proximity to the root, such as in scheduling systems, finding the shortest path in graphs, or representing data hierarchies.

In **Depth-First Traversal**, the traversal proceeds by exploring as deeply as possible along each branch before backtracking.

- **Inorder Traversal**: The nodes are visited in the order of left subtree first, then the root node, and finally the right subtree. This method is particularly useful in binary search trees because it retrieves values in a naturally sorted order.
- **Preorder Traversal**: This visits the root node first, followed by the left subtree and then the right subtree. This order is commonly applied in tasks such as creating a duplicate of the tree or evaluating prefix expressions.
- **Postorder Traversal**: This visits the left subtree first, then the right subtree, and finally the root node. Postorder traversal is useful when a task requires processing child nodes before their parent node, such as safely deleting a tree or evaluating postfix expressions. It is useful for deleting nodes or postfix expression evaluation.

Binary Tree	Traversal Type	Order of Visit
	**In order** (L-Root-R)	2, 5, 7, 10, 12, 15
	Preorder (Root-L-R)	10, 5, 2, 7, 15, 12
	Postorder (L-R-Root)	2, 7, 5, 12, 15, 10
	Level Order	10, 5, 15, 2, 7, 12

Binary Search Tree (BST)

It is used for retrieving, sorting, and searching data. It has its nodes arranged in a specific order and thus, is also called the **Ordered Binary Tree**. It has the following properties:

- Both the right and left subtrees must also be binary search trees

  In a BST, not only must the parent node follow the BST rule, but its **left** and **right subtrees** must also individually follow BST properties. This means the structure is **recursively valid** at every node.

	The subtree rooted at 10 (5, 12) is a BST The subtree rooted at 25 (20, 30) is also a BST Therefore, the entire tree is a valid BST

- A node's right subtree only contains nodes with keys that are greater than the node's key.

  All nodes in the right subtree of any given node must have keys **strictly greater** than the key of the node itself.

Correct: 	The right child 30 is greater than 20
Incorrect: 	15 is in the right subtree, but **less than** 20

- A node's left subtree only contains nodes with keys that are less than the node's key.

  All nodes in the left subtree of any given node must have keys **strictly less** than the node's key.

Correct: 25 / \ 10 40	10 is less than 25
Incorrect: 25 / \ 30 40	30 is in the left subtree but **greater than** 25

- A Binary Search Tree does not allow duplicate nodes
  BSTs **do not allow duplicate keys** because the rule of left being less and right being greater would be broken. Each key must appear **only once**.

Correct: 8 / \ 3 10	No duplicates
Incorrect: 8 / \ 3 8	8 is duplicated on the right

- Every node has a unique key.
  Every node's key is **unique** in the entire tree, ensuring efficient **search, insert, and delete operations** with a time complexity of O(log n) (on balanced BSTs).

50 / \ 30 70 / \ / \ 20 40 60 80	All keys {20, 30, 40, 50, 60, 70, 80} are unique. If any key (e.g., another 30) is added, the BST property is violated.

Here is how Binary Tree, AVL Tree, and Binary Search Tree differ:

Aspect	Binary Tree	Binary Search Tree (BST)	AVL Tree
Ordering Rule	No ordering rule	Strict ordering rule: Left subtree < node < Right subtree.	Strict ordering rule **plus self-balancing** after every insertion or deletion to maintain height balance.
Duplicates	May allow duplicates depending on the use case	No duplicates (keys are unique).	No duplicates (keys are unique); balancing requires distinct keys for correct tree adjustments.
Usage	Used for general hierarchical data, like expression trees, heap trees.	Used for sorted data for quick search, insert, and delete.	Used for sorted data where **guaranteed balanced height** ensures consistent **O(log n)** search, insert, and delete performance.

Sorting Property	No sorting of elements.	Sorted structure by key value.	Sorted structure by key value **with height balancing for optimal efficiency**

AVL Trees

An Adelson-Velskii Landis (**AVL**) tree is a self-balancing BST in which each node maintains extra information called a **balance factor** whose value is either -1, 0, or +1.

In an AVL tree, the balance factor of a node is the difference between the height of the left subtree and the height of the right subtree.

Balance Factor = Height (Left Subtree) - Height (Right Subtree)

A balance factor of:

- +1 means the left subtree is taller by 1
- 0 means the left and right subtrees are of equal height
- -1 means the right subtree is taller by 1

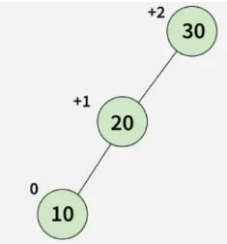
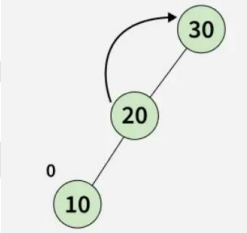
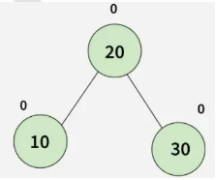
If the balance factor goes beyond -1 or +1, the tree self-balances using rotations.

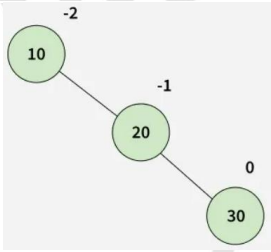
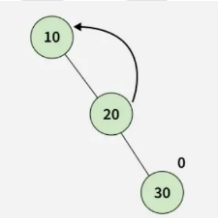
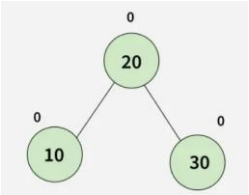
Balanced 30 (0) / \ 20(0) 40(0)	**Node 30**: • Left subtree height = 1 (20), Right subtree height = 1 (40) • Balance Factor = 1 - 1 = **0** **Node 20, 40**: • Both have no children, so the height = 0. • Balance Factor = 0 - 0 = **0** **AVL property satisfied** — balance factors are in {-1, 0, +1}.
Unbalanced 30 (+2) / 20 (+1) / 10 (0)	**Node 30:** • Left height = 2 (20 → 10), Right height = 0 • Balance Factor = 2 - 0 = +2 (Unbalanced) **Node 20:** • Left height = 1 (10), Right height = 0 • Balance Factor = 1 - 0 = +1 **Node 10:** • No children → height = 0 → balance factor = 0.

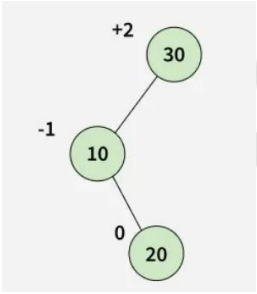
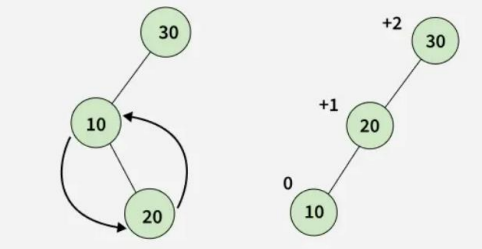
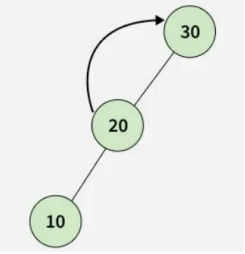
Rotations

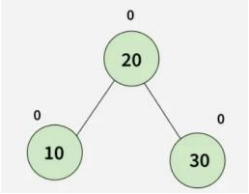
In **AVL Trees**, whenever an insertion or deletion operation causes a node's **balance factor** to go beyond -1 or +1, **rotations** are applied to restore balance.

Examples:

<table>
<tr>
<td>Left-Left (LL) Rotation
Insertion happens in the LEFT subtree of the LEFT child.
 
</td>
<td>Reason for Imbalance:
○ Balance Factor of 30 = +2 (left-heavy).
○ Left child 20 is also left-heavy.
○ Single Right Rotation solves this.
 
</td>
<td>Result: Right Rotation at 30
 

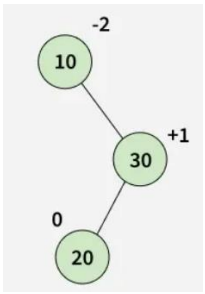
○ 20 becomes the new root of the subtree.
○ Left subtree reduces height → balanced.</td>
</tr>
<tr>
<td>Right-Right (RR) Rotation
Insertion happens in the RIGHT subtree of the RIGHT child.
 
</td>
<td>Reason for Imbalance:
○ Balance Factor of 10 = -2 (right-heavy).
○ Right child 20 is also right-heavy.
○ Single Left Rotation restores balance.
 
</td>
<td>Result: Left Rotation at 10
 

○ 20 becomes the new root of the subtree.
○ Right subtree reduces height → balanced.</td>
</tr>
<tr>
<td>Left-Right (LR) Rotation
Insertion happens in the RIGHT subtree of the LEFT child (zig-zag).
 
</td>
<td>Reason for Imbalance:
○ Balance Factor of 30 = +2 (left-heavy).
○ Left child 10 = -1 (right-heavy).
○ Double Rotation is needed.
(Applying Left Rotation) (After Left Rotation)
 
</td>
<td>Result: Applying Right Rotation on nodes 20 and 30
 

Result: After Right Rotation
 
</td>
</tr>
</table>

Right-Left (RL) Rotation

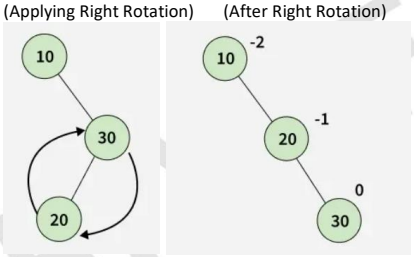
Insertion happens in the LEFT subtree of the RIGHT child (zig-zag).



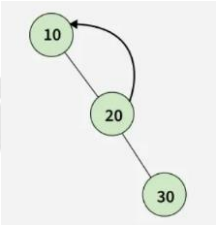
Reason for Imbalance:

- Balance Factor of 10 = -2 (right-heavy).
- Right child 30 = +1 (left-heavy).
- **Double Rotation** required.

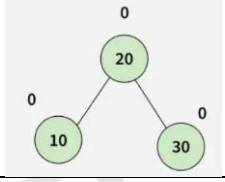
(Applying Right Rotation) (After Right Rotation)



Result: Applying left rotation on nodes 10 and 20



Result: After Left Rotation

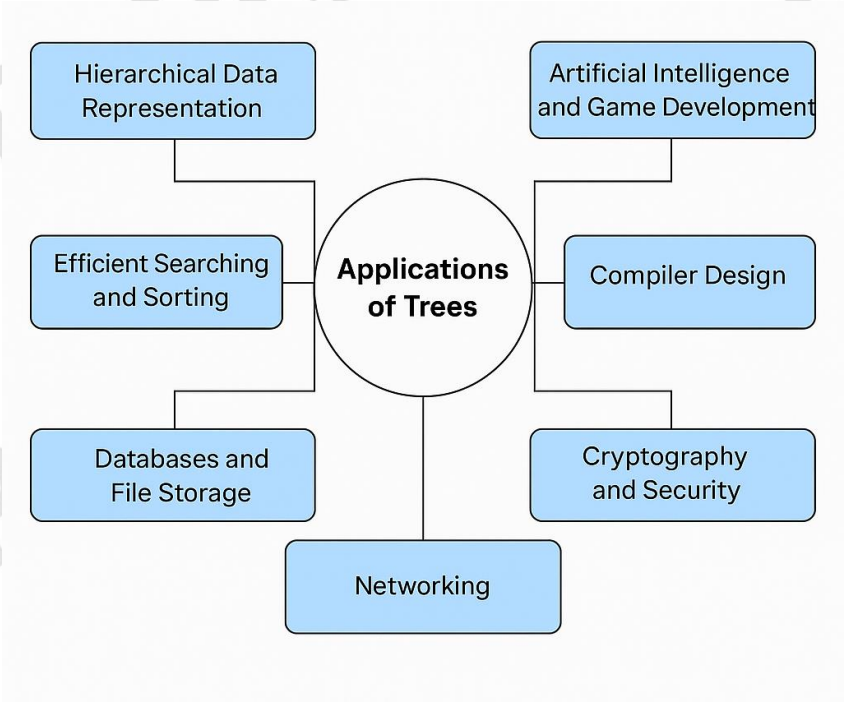


Here is a simple summary:

Before	Rotation(s)	After
A → B → C (LL)	Right rotate A	B root
A ← B ← C (RR)	Left rotate A	B root
A → B ← C (LR)	Left rotate B, right rotate A	C root
A ← B → C (RL)	Right rotate B, left rotate A	C root

Application of Trees

Trees are widely used in computer science due to their efficient representation of hierarchical relationships and their ability to optimize data operations.



Hierarchical Data Representation
In **hierarchical data structures**, trees are foundational, commonly used to represent file systems where directories contain subdirectories and files in a tree-like format. Similarly, organizational charts and the Document Object Model (DOM) in HTML or XML documents use tree structures to illustrate nested relationships clearly.

Efficient Searching and Sorting
In the area of **efficient searching and sorting**, trees like Binary Search Trees (BST), AVL Trees, and Red-Black Trees provide fast access, insertion, and deletion operations. Balanced trees ensure that these operations remain efficient, especially in scenarios with frequent data updates.

Databases and File Storage Systems
This also relies heavily on tree structures, particularly B-Trees and B+ Trees, which are used to manage large volumes of data and support efficient indexing and range queries in relational databases like MySQL and Oracle. Additionally, specialized trees like Tries (prefix trees) are used for fast retrieval in applications such as autocomplete systems, dictionary lookups, and IP routing in networks.

Artificial Intelligence (AI) and Game Development
In the field of AI and game development, trees play a central role. Decision Trees are commonly used for classification problems in machine learning, while Minimax Trees guide decision-making in competitive games like chess, often optimized using alpha-beta pruning. Behavior Trees are another example, widely used in modern video games to control non-player character (NPC) behaviors in a structured and modular way.

Compiler Design
Compilers rely on Abstract Syntax Trees (ASTs) to represent the syntactic structure of programming languages, helping to parse and analyze code efficiently. Parse Trees also help visualize the grammatical structure of code inputs.

Cryptography and Security
Merkle Trees are a prominent example, especially in blockchain technologies like Bitcoin and Ethereum, where they facilitate fast and secure verification of transactions. Authentication systems also use tree structures to organize and verify credentials efficiently.

Networking
In **computer networking**, trees are essential for building routing tables and performing fast IP lookups using trie structures. Algorithms like Minimum Spanning Tree (MST), employed through Kruskal's or Prim's algorithm, are used to optimize network designs by minimizing connection costs.

Trees are indispensable across various fields in computing. They ensure efficient data organization, quick retrieval and modification of data, and structured management of complex systems. Their adaptability makes them crucial in operating systems, databases, networking, AI, game development, compilers, security, and graphics, providing both theoretical robustness and practical efficiency in diverse applications.

References:

Geeks for Geeks. (2025). *AVL tree data structure.* Retrieved on July 16, 2025, from https://www.geeksforgeeks.org/dsa/introduction-to-avl-tree/

Naukiri Code360. (2024). *Difference between binary tree and binary search tree.* Retrieved on July 15, 2025, from https://www.naukri.com/code360/library/difference-between-binary-tree-and-binary-search-tree