**AVL TREES**

**1. Binary Search Tree (BST) and its Time Complexity**

A **Binary Search Tree (BST)** is a hierarchical data structure in which:

* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree contains only nodes with keys greater than the node’s key.
* The left and right subtrees are also BSTs.

**Time Complexity of BST Operations**

| **Operation** | **Best Case (Balanced BST)** | **Worst Case (Skewed BST)** |
| --- | --- | --- |
| **Search** | O(log n) | O(n) |
| **Insertion** | O(log n) | O(n) |
| **Deletion** | O(log n) | O(n) |

A **balanced BST** ensures that the height remains O(log n), but an **unbalanced BST** can degrade to O(n), making operations inefficient.

**2. Need for Self-Balancing Trees**

BST operations become inefficient when the tree becomes skewed. Self-balancing trees like **AVL Trees** and **Red-Black Trees** maintain the tree’s height close to O(log n), ensuring optimal performance for search, insert, and delete operations.

**Advantages of Self-Balancing Trees:**

* Prevents worst-case O(n) time complexity.
* Ensures efficient searching, insertion, and deletion.
* Useful in database indexing, memory management, and caches.

**3. AVL Trees: Introduction and Detection of Imbalance**

An **AVL Tree** (named after Adelson-Velsky and Landis) is a self-balancing BST where the **height difference (balance factor) between left and right subtrees of any node is at most 1**.

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

* **BF = {-1, 0, 1}** → The tree is balanced.
* **BF < -1 or BF > 1** → The tree is unbalanced and requires rotation to restore balance.

**4. Sub-Cases of Imbalance in AVL Trees**

There are **four types of imbalance cases** based on insertion:

1. **Left-Left (LL) Case** → Inserted node is in the left subtree of the left child.
2. **Left-Right (LR) Case** → Inserted node is in the right subtree of the left child.
3. **Right-Right (RR) Case** → Inserted node is in the right subtree of the right child.
4. **Right-Left (RL) Case** → Inserted node is in the left subtree of the right child.

**5. How to Balance Each of These Cases**

Balancing AVL trees requires **rotations**:

1. **LL Case (Right Rotation - Single Rotation)**
   * Perform a **Right Rotation** on the unbalanced node.
2. **LR Case (Left-Right Rotation - Double Rotation)**
   * Perform a **Left Rotation** on the left child.
   * Then perform a **Right Rotation** on the unbalanced node.
3. **RR Case (Left Rotation - Single Rotation)**
   * Perform a **Left Rotation** on the unbalanced node.
4. **RL Case (Right-Left Rotation - Double Rotation)**
   * Perform a **Right Rotation** on the right child.
   * Then perform a **Left Rotation** on the unbalanced node.

**6. AVL Trees vs. Red-Black Trees: When to Use Which?**

Both AVL and Red-Black trees maintain balance, but they have different use cases.

| **Feature** | **AVL Tree** | **Red-Black Tree** |
| --- | --- | --- |
| **Balancing** | More strictly balanced | Less strictly balanced |
| **Search** | O(log n), faster than RBT | O(log n), slightly slower |
| **Insertion** | O(log n), requires more rotations | O(log n), fewer rotations |
| **Deletion** | O(log n), complex | O(log n), simpler |
| **Best Use Case** | Search-intensive apps (e.g., DB indexing) | Frequent insertions (e.g., OS scheduling, memory management) |

**Use AVL Trees when:**

* Faster search is required.
* Insertions and deletions are relatively infrequent.

**Use Red-Black Trees when:**

* Insertion and deletion operations are frequent (e.g., dynamic memory allocation).

**7. UML Diagram for Web UI Project Showing AVL Tree Visualization**

The UML diagram will include:

* **User Interface (UI)** → Input Box, Buttons (Insert, Delete, Reset).
* **AVL Tree Controller** → Handles user actions and manages the AVL Tree.
* **AVL Tree Model** → Represents tree structure and operations (rotations, insertions).
* **AVL Tree View** → Renders tree visualization using HTML, CSS, and JavaScript.

