**🔷 THEORY**

**📌 AVL Tree**

An **AVL Tree** is a **self-balancing Binary Search Tree (BST)** where the **balance factor** (height difference between left and right subtree) of every node is between -1 and 1.

This balance is maintained by performing **rotations** after insertion or deletion:

* **Left-Left (LL)** → Right Rotation
* **Right-Right (RR)** → Left Rotation
* **Left-Right (LR)** → Left-Right Rotation
* **Right-Left (RL)** → Right-Left Rotation

**📌 Dictionary Using AVL Tree**

* Each **node** stores a **keyword** and its **meaning**.
* Operations include:
  + **Insert**: Add or update a keyword.
  + **Delete**: Remove a keyword.
  + **Search**: Find a keyword and return its meaning with comparison count.
  + **Display**: Show keywords in **ascending or descending** order.

**📌 Advantages**

* **Fast search, insert, and delete** operations with guaranteed time complexity of **O(log n)**.
* Ensures the tree remains balanced after updates.

**🔷 ALGORITHM**

**🔧 Insert (Add or Update)**

text

CopyEdit

Input: root, keyword, meaning

Output: Updated root with balanced AVL subtree

1. If root is NULL → create new Node with keyword

2. If keyword < root.keyword → insert in left subtree

3. If keyword > root.keyword → insert in right subtree

4. If keyword == root.keyword → update meaning

5. Update height of root:

height = 1 + max(height(left), height(right))

6. Calculate balance = height(left) - height(right)

7. Perform rotations if unbalanced:

a. Left-Left → Right Rotation

b. Right-Right → Left Rotation

c. Left-Right → Left Rotate left child, then Right Rotate

d. Right-Left → Right Rotate right child, then Left Rotate

**🔧 Delete**

text

CopyEdit

Input: root, keyword

Output: Updated and balanced AVL tree after deletion

1. Perform standard BST delete

2. If node has two children:

- Replace with inorder successor (minimum value in right subtree)

3. Update height

4. Calculate balance

5. Perform rotations as needed (same as insert)

**🔧 Search**

text

CopyEdit

Input: root, keyword

Output: Keyword found (or not), and number of comparisons

1. If root is NULL → not found

2. If root.keyword == keyword → found

3. If keyword < root.keyword → search left

4. Else → search right

5. Track comparisons in each step

**🔧 Display Ascending (Inorder)**

text

CopyEdit

1. Traverse left subtree

2. Print keyword and meaning

3. Traverse right subtree

**🔧 Display Descending (Reverse Inorder)**

text

CopyEdit

1. Traverse right subtree

2. Print keyword and meaning

3. Traverse left subtree