**ADVANCED PROGRAMMING: AVL TREES**

1. **Describe the data structure**

**Data structures** are specialized formats for organizing, managing, and storing data in a way that makes it efficient to access and modify.

An **AVL tree** (Adelson-Velsky and Landis tree) is a type of **self-balancing binary search tree**. It ensures that the height of the tree remains logarithmic, which guarantees efficient operations like insertion, deletion, and search.

**Key Properties of an AVL Tree:**

1. **Binary Search Tree Property**:
   * The left subtree of a node contains only nodes with values less than the node.
   * The right subtree contains only nodes with values greater than the node.
2. **Balance Factor**:
   * For every node, the difference in height between the left and right subtrees (called the balance factor) must be **-1, 0, or 1**.
   * Balance factor = height(left subtree) - height(right subtree).
3. **Self-Balancing**:
   * If an operation (insertion or deletion) causes the balance factor to go out of range (-1 to 1), the tree performs rotations to restore balance.

**Rotations in AVL Trees:**

Rotations are used to maintain the AVL property after an insertion or deletion. We have the single rotations(LL Imbalance and RR Imbalance) and the double rotations(the LR and the RL Imbalance).

**Operations in an AVL Tree:**

1. **Insertion**:Similar to a binary search tree insertion but may require rotations to maintain balance.
2. **Deletion**:Remove a node like in a binary search tree, but after deletion, balance factors are checked, and rotations are performed if necessary.
3. **Search**:Follow the binary search tree search logic, which operates in O(log⁡n)O(\log n)O(logn) time due to the balanced nature of AVL trees.
4. **Full ADT specification**

**Domain**

The domain of the AVL tree includes:

* A collection of unique elements from a totally ordered set (e.g., integers, strings).
* Each node contains:
  + A key (value).
  + References to left and right children.
  + A height value (used to compute balance factors).

**Operations**

**1. Create**

* **Purpose**: Initialize an empty AVL tree.
* **Signature**: Create() → AVLTree
* **Precondition**: None.
* **Postcondition**: Returns an empty AVL tree with no nodes.

**2. Insert**

* **Purpose**: Insert a new key into the AVL tree while maintaining balance.
* **Signature**: Insert(tree: AVLTree, key: K) → AVLTree
* **Precondition**: key is not already in the tree.
* **Postcondition**: The tree includes the new key, and the AVL property is preserved.

**3. Delete**

* **Purpose**: Remove a key from the AVL tree while maintaining balance.
* **Signature**: Delete(tree: AVLTree, key: K) → AVLTree
* **Precondition**: key must exist in the tree.
* **Postcondition**: The key is removed from the tree, and the AVL property is preserved.

**4. Search**

* **Purpose**: Determine if a key exists in the tree.
* **Signature**: Search(tree: AVLTree, key: K) → Boolean
* **Precondition**: None.
* **Postcondition**: Returns true if the key is found, otherwise false.

**5. FindMin**

* **Purpose**: Retrieve the smallest key in the AVL tree.
* **Signature**: FindMin(tree: AVLTree) → K
* **Precondition**: The tree is not empty.
* **Postcondition**: Returns the smallest key in the tree.

**6. FindMax**

* **Purpose**: Retrieve the largest key in the AVL tree.
* **Signature**: FindMax(tree: AVLTree) → K
* **Precondition**: The tree is not empty.
* **Postcondition**: Returns the largest key in the tree.

**7. Height**

* **Purpose**: Return the height of the AVL tree.
* **Signature**: Height(tree: AVLTree) → Integer
* **Precondition**: None.
* **Postcondition**: Returns the height of the tree.

**8. Traverse**

* **Purpose**: Visit all nodes in the tree in a specified order (e.g., in-order, pre-order, post-order).
* **Signature**: Traverse(tree: AVLTree, order: String) → List[K]
* **Precondition**: The tree is not empty.
* **Postcondition**: Returns a list of keys in the specified traversal order.

**9. RotateLeft**

* **Purpose**: Perform a left rotation to restore balance.
* **Signature**: RotateLeft(tree: AVLTree, node: Node) → AVLTree
* **Precondition**: The balance factor of the node is -2, and the right subtree is heavier.
* **Postcondition**: The tree is rotated left, and the AVL property is preserved.

**10. RotateRight**

* **Purpose**: Perform a right rotation to restore balance.
* **Signature**: RotateRight(tree: AVLTree, node: Node) → AVLTree
* **Precondition**: The balance factor of the node is +2, and the left subtree is heavier.
* **Postcondition**: The tree is rotated right, and the AVL property is preserved.

**11. RotateLeftRight**

* **Purpose**: Perform a left rotation followed by a right rotation to restore balance.
* **Signature**: RotateLeftRight(tree: AVLTree, node: Node) → AVLTree
* **Precondition**: The balance factor of the node is +2, and the left subtree's right child is heavier.
* **Postcondition**: The tree is balanced using the left-right rotation.

**12. RotateRightLeft**

* **Purpose**: Perform a right rotation followed by a left rotation to restore balance.
* **Signature**: RotateRightLeft(tree: AVLTree, node: Node) → AVLTree
* **Precondition**: The balance factor of the node is -2, and the right subtree's left child is heavier.
* **Postcondition**: The tree is balanced using the right-left rotation.

**Axioms**

**1. AVL Property:**

* For every node in the tree, |Height(left subtree) - Height(right subtree)| ≤ 1.

**2. Insertion:**

* After inserting a key, the balance factor of all nodes remains within the range [-1, 1].

**3. Deletion:**

* After deleting a key, the balance factor of all nodes remains within the range [-1, 1].

**4. Search:**

* If Search(tree, key) returns true, then the key is in the tree.

**5. Traversal:**

* The Traverse(tree, "in-order") operation produces a sorted list of all keys in the tree.

1. **Choose an Application Domain**

**Database Indexing**

* + AVL trees ensure O(log⁡n)O(\log n)O(logn) complexity for search, insertion, and deletion, making them ideal for maintaining indexes in databases.
  + They keep the data balanced, ensuring quick lookups for queries.
* **Use Cases**:
  + Range queries (e.g., finding all records between two dates).
  + Managing primary or secondary indices in database systems.
* **Example**:
  + A database for managing e-commerce orders or student academic records.

1. **Implement an application**

**Use Case for Database Indexing**

* Suppose you have a database of **student records**, and you want to:

1. Index the records by **student ID**.
2. Perform efficient searches for a specific student ID.
3. Insert and delete student records while maintaining the balance.

**Design**

**Data Structure:**

* Each node in the AVL tree will represent a **student record**, containing:
  + student\_id (key for indexing)
  + name
  + age
  + grade

**Operations:**

1. **Insert**: Add a new student record.
2. **Delete**: Remove a student record by ID.
3. **Search**: Find a student record by ID.
4. **Display**: Traverse and print the tree in sorted order of IDs (in-order traversal).

**Code Implementation**

`python`

class AVLNode:

def \_\_init\_\_(self, student\_id, name, age, grade):

self.student\_id = student\_id

self.name = name

self.age = age

self.grade = grade

self.height = 1 # Height of this node

self.left = None # Left child

self.right = None # Right child

class AVLTree:

def \_\_init\_\_(self):

self.root = None

# Utility function to get the height of a node

def get\_height(self, node):

return node.height if node else 0

# Utility function to calculate balance factor

def get\_balance(self, node):

return self.get\_height(node.left) - self.get\_height(node.right) if node else 0

# Right rotation

def rotate\_right(self, z):

y = z.left

T3 = y.right

y.right = z

z.left = T3

z.height = 1 + max(self.get\_height(z.left), self.get\_height(z.right))

y.height = 1 + max(self.get\_height(y.left), self.get\_height(y.right))

return y

# Left rotation

def rotate\_left(self, z):

y = z.right

T2 = y.left

y.left = z

z.right = T2

z.height = 1 + max(self.get\_height(z.left), self.get\_height(z.right))

y.height = 1 + max(self.get\_height(y.left), self.get\_height(y.right))

return y

# Insert a new record into the AVL tree

def insert(self, node, student\_id, name, age, grade):

if not node:

return AVLNode(student\_id, name, age, grade)

# Perform standard BST insertion

if student\_id < node.student\_id:

node.left = self.insert(node.left, student\_id, name, age, grade)

elif student\_id > node.student\_id:

node.right = self.insert(node.right, student\_id, name, age, grade)

else:

return node # Duplicates are not allowed

# Update height of the current node

node.height = 1 + max(self.get\_height(node.left), self.get\_height(node.right))

# Check balance and perform rotations if needed

balance = self.get\_balance(node)

# Left Left Case

if balance > 1 and student\_id < node.left.student\_id:

return self.rotate\_right(node)

# Right Right Case

if balance < -1 and student\_id > node.right.student\_id:

return self.rotate\_left(node)

# Left Right Case

if balance > 1 and student\_id > node.left.student\_id:

node.left = self.rotate\_left(node.left)

return self.rotate\_right(node)

# Right Left Case

if balance < -1 and student\_id < node.right.student\_id:

node.right = self.rotate\_right(node.right)

return self.rotate\_left(node)

return node

# Delete a record from the AVL tree

def delete(self, node, student\_id):

if not node:

return node

# Perform standard BST deletion

if student\_id < node.student\_id:

node.left = self.delete(node.left, student\_id)

elif student\_id > node.student\_id:

node.right = self.delete(node.right, student\_id)

else:

# Node with one or no child

if not node.left:

return node.right

elif not node.right:

return node.left

# Node with two children: Get the inorder successor

temp = self.get\_min\_value\_node(node.right)

node.student\_id = temp.student\_id

node.name = temp.name

node.age = temp.age

node.grade = temp.grade

node.right = self.delete(node.right, temp.student\_id)

# Update height and check balance

node.height = 1 + max(self.get\_height(node.left), self.get\_height(node.right))

balance = self.get\_balance(node)

# Balance the tree

if balance > 1 and self.get\_balance(node.left) >= 0:

return self.rotate\_right(node)

if balance > 1 and self.get\_balance(node.left) < 0:

node.left = self.rotate\_left(node.left)

return self.rotate\_right(node)

if balance < -1 and self.get\_balance(node.right) <= 0:

return self.rotate\_left(node)

if balance < -1 and self.get\_balance(node.right) > 0:

node.right = self.rotate\_right(node.right)

return self.rotate\_left(node)

return node

# Utility to find the node with the smallest key

def get\_min\_value\_node(self, node):

current = node

while current.left:

current = current.left

return current

# Search for a student record by ID

def search(self, node, student\_id):

if not node or node.student\_id == student\_id:

return node

if student\_id < node.student\_id:

return self.search(node.left, student\_id)

return self.search(node.right, student\_id)

# In-order traversal (sorted order)

def in\_order(self, node):

if node:

self.in\_order(node.left)

print(f"ID: {node.student\_id}, Name: {node.name}, Age: {node.age}, Grade: {node.grade}")

self.in\_order(node.right)

# Example Usage

if \_\_name\_\_ == "\_\_main\_\_":

tree = AVLTree()

# Insert records

tree.root = tree.insert(tree.root, 101, "Alice", 20, "A")

tree.root = tree.insert(tree.root, 102, "Bob", 22, "B")

tree.root = tree.insert(tree.root, 103, "Charlie", 21, "A-")

# Display all records

print("In-order traversal:")

tree.in\_order(tree.root)

# Search for a record

print("\nSearching for ID 102:")

result = tree.search(tree.root, 102)

if result:

print(f"Found: {result.student\_id}, {result.name}, {result.age}, {result.grade}")

else:

print("Record not found.")

# Delete a record

print("\nDeleting ID 102...")

tree.root = tree.delete(tree.root, 102)

# Display all records after deletion

print("\nIn-order traversal after deletion:")

tree.in\_order(tree.root)

**Features of the Code**

1. **Insertion**: Adds a student record while maintaining balance.
2. **Search**: Finds a record by student ID.
3. **Deletion**: Removes a record and ensures the AVL tree remains balanced.
4. **In-order Traversal**: Prints all records in sorted order by student ID.