AVL Tree

Created by	j jubink
Created time	@February 26, 2025 10:19 PM
Tags	

Implementation

Step 1: Define the Node Class

Each node in the AVL tree contains:

- `key` → Stores the value.
- `left` → Pointer to the left child.
- `right` → Pointer to the right child.
- `height` → Height of the node (used for balancing).

```
python
CopyEdit
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1  # Every node starts with height 1
```

Step 2: Define the AVLTree Class

The `AVLTree` class will handle:

✅ **Insertion** - Adds a node while maintaining balance.

✅ **Rotation** - Performs `left_rotate()` and `right_rotate()` if needed.

✅ **Balance Factor Calculation** - Determines when to rotate.

✅ **Deletion** - Removes a node and re-balances the tree.

✅ **Traversals** - Inorder traversal to display tree contents.

```
python
CopyEdit
class AVLTree:
    def __init__(self):
        self.root = None  # Root of the AVL tree
```

Step 3: Insert a Node into AVL Tree

How does AVL insertion work?

1. **Insert the node like in a normal BST**
2. **Update the height** of the current node.
3. **Check the balance factor**:
   - If **balance > 1** → Left-heavy (Right Rotation needed).
   - If **balance < -1** → Right-heavy (Left Rotation needed).
4. **Perform rotations** if unbalanced:
   - **Left-Left (LL) Case** → Single **Right Rotation**.
   - **Right-Right (RR) Case** → Single **Left Rotation**.
   - **Left-Right (LR) Case** → **Left Rotation**, then **Right Rotation**.
   - **Right-Left (RL) Case** → **Right Rotation**, then **Left Rotation**.

```
python
CopyEdit
    def insert(self, root, key):
```

```
        """Insert a new node and maintain AVL balance."""
        if not root:
            return Node(key)  # Create a new node if root is None

        # Standard BST insertion
        if key < root.key:
            root.left = self.insert(root.left, key)
        elif key > root.key:
            root.right = self.insert(root.right, key)
        else:
            return root  # Duplicate keys are ignored

        # Step 1: Update height of the current node
        root.height = 1 + max(self.get_height(root.left), self.get_height(root.right))

        # Step 2: Get the balance factor
        balance = self.get_balance(root)

        # Step 3: Perform rotations if unbalanced
        # Left Heavy (Right Rotation)
        if balance > 1 and key < root.left.key:
            return self.right_rotate(root)

        # Right Heavy (Left Rotation)
        if balance < -1 and key > root.right.key:
            return self.left_rotate(root)

        # Left-Right Case (Left Rotate + Right Rotate)
        if balance > 1 and key > root.left.key:
            root.left = self.left_rotate(root.left)
            return self.right_rotate(root)

        # Right-Left Case (Right Rotate + Left Rotate)
        if balance < -1 and key < root.right.key:
            root.right = self.right_rotate(root.right)
            return self.left_rotate(root)
```

```
        return root
```

Step 4: Perform Rotations

Right Rotation (For Left-Heavy Trees)

```
python
CopyEdit
    def right_rotate(self, z):
        y = z.left
        T3 = y.right

        # Perform rotation
        y.right = z
        z.left = T3

        # Update heights
        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 (For Right-Heavy Trees)

```
python
CopyEdit
    def left_rotate(self, z):
        y = z.right
        T2 = y.left

        # Perform rotation
```

```
        y.left = z
        z.right = T2

        # Update heights
        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
```

Step 5: Get Height and Balance Factor

```
python
CopyEdit
    def get_height(self, node):
        """Returns the height of the node."""
        return node.height if node else 0

    def get_balance(self, node):
        """Returns the balance factor of the node."""
        return self.get_height(node.left) - self.get_height(node.right) if node else
0
```

Step 6: Inorder Traversal

```
python
CopyEdit
    def inorder_traversal(self, root):
        """Inorder traversal (Left → Root → Right)."""
        if root:
            self.inorder_traversal(root.left)
            print(root.key, end=" ")
```

```
            self.inorder_traversal(root.right)
```

Step 7: Wrapper Methods to Handle Root

```
python
CopyEdit
    def insert_key(self, key):
        """Wrapper function to insert into the tree."""
        self.root = self.insert(self.root, key)

    def display(self):
        """Displays the inorder traversal of the AVL tree."""
        print("Inorder Traversal of AVL Tree:")
        self.inorder_traversal(self.root)
        print()
```

Step 8: Testing the AVL Tree

```
python
CopyEdit
# Create AVL Tree
avl = AVLTree()

# Insert values
values = [10, 20, 30, 40, 50, 25]
for val in values:
    avl.insert_key(val)

# Display tree (Inorder Traversal)
```

```
avl.display()
```

Step 9: Understanding the Output

```
objectivec
CopyEdit
Inorder Traversal of AVL Tree:
10 20 25 30 40 50
```

The AVL Tree remains **balanced** automatically after every insertion!

Summary of Steps

✅ **Step 1** - Create a `Node` class with `key`, `left`, `right`, and `height`.

✅ **Step 2** - Create an `AVLTree` class to manage the tree.

✅ **Step 3** - Implement `insert()` with balance checking.

✅ **Step 4** - Implement `left_rotate()` and `right_rotate()`.

✅ **Step 5** - Implement `get_height()` and `get_balance()`.

✅ **Step 6** - Implement `inorder_traversal()` to display the tree.

✅ **Step 7** - Implement wrapper methods for easy usage.

✅ **Step 8** - Insert values and verify the balanced tree.

🌳 AVL Tree Practice Questions with Solutions

AVL trees are **self-balancing** binary search trees where the **height difference** (balance factor) between the left and right subtrees of any node is at most **±1**.

Each problem includes:

✅ **Step-by-step algorithm**

✅ **Python implementation**

🔷 Beginner Level (AVL Tree Fundamentals)

1. Insert Nodes into an AVL Tree

📌 **Concept**: AVL tree follows BST insertion, then performs rotations if the tree becomes unbalanced.

Algorithm

1. Perform **BST insertion**.
2. Update the **height** of the node.
3. Compute the **balance factor**:
   - If balance > 1 → **Left-heavy (LL or LR case)**
   - If balance < -1 → **Right-heavy (RR or RL case)**
4. Perform the necessary rotation to balance the tree.

Python Code

```
python
CopyEdit
class AVLNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:
    def get_height(self, node):
        return node.height if node else 0
```

```
    def get_balance_factor(self, node):
        return self.get_height(node.left) - self.get_height(node.right) if node else
0

    def rotate_right(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = max(self.get_height(y.left), self.get_height(y.right)) + 1
        x.height = max(self.get_height(x.left), self.get_height(x.right)) + 1
        return x

    def rotate_left(self, x):
        y = x.right
        T2 = y.left
        y.left = x
        x.right = T2
        x.height = max(self.get_height(x.left), self.get_height(x.right)) + 1
        y.height = max(self.get_height(y.left), self.get_height(y.right)) + 1
        return y

    def insert(self, root, key):
        if not root:
            return AVLNode(key)
        if key < root.key:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)

        root.height = max(self.get_height(root.left), self.get_height(root.right)) + 1
        balance = self.get_balance_factor(root)

        # Balance the tree
        if balance > 1 and key < root.left.key:  # Left-Left Case
```

```
            return self.rotate_right(root)
        if balance < -1 and key > root.right.key:  # Right-Right Case
            return self.rotate_left(root)
        if balance > 1 and key > root.left.key:  # Left-Right Case
            root.left = self.rotate_left(root.left)
            return self.rotate_right(root)
        if balance < -1 and key < root.right.key:  # Right-Left Case
            root.right = self.rotate_right(root.right)
            return self.rotate_left(root)

        return root

# Usage
avl = AVLTree()
root = None
values = [10, 20, 30, 40, 50, 25]
for v in values:
    root = avl.insert(root, v)
```

---

2. Delete a Node from an AVL Tree

📌 **Concept**: Deletion in AVL tree follows BST deletion, then rebalances if needed.

Algorithm

1. Perform **BST deletion**.
2. Update the **height** of the current node.
3. Calculate the **balance factor** and perform rotations if necessary.

Python Code

```
python
CopyEdit
def delete(root, key):
```

```
    if not root:
        return root
    if key < root.key:
        root.left = delete(root.left, key)
    elif key > root.key:
        root.right = delete(root.right, key)
    else:
        if not root.left:
            return root.right
        elif not root.right:
            return root.left
        temp = find_min(root.right)
        root.key = temp.key
        root.right = delete(root.right, temp.key)

    if not root:
        return root

    root.height = max(get_height(root.left), get_height(root.right)) + 1
    balance = get_balance_factor(root)

    # Rotations to balance the tree
    if balance > 1 and get_balance_factor(root.left) >= 0:
        return rotate_right(root)
    if balance > 1 and get_balance_factor(root.left) < 0:
        root.left = rotate_left(root.left)
        return rotate_right(root)
    if balance < -1 and get_balance_factor(root.right) <= 0:
        return rotate_left(root)
    if balance < -1 and get_balance_factor(root.right) > 0:
        root.right = rotate_right(root.right)
        return rotate_left(root)

    return root
```

3. Find the Height of an AVL Tree

📌 **Concept**: The height of a tree is the length of the longest path from the root to a leaf node.

Algorithm

1. If the tree is **empty**, return **0**.
2. Recursively find the **height** of left and right subtrees.
3. Return **1 + max(left height, right height)**.

Python Code

```
python
CopyEdit
def get_height(root):
    if not root:
        return 0
    return max(get_height(root.left), get_height(root.right)) + 1

# Usage
print("Height of AVL Tree:", get_height(root))
```

4. Find the Balance Factor of Each Node in an AVL Tree

📌 **Concept**: The balance factor is `height(left subtree) - height(right subtree)`.

Python Code

```
python
CopyEdit
def get_balance_factor(node):
```

```
    return get_height(node.left) - get_height(node.right) if node else 0
```

5. Find the Minimum and Maximum Value in an AVL Tree

📌 **Concept**:

- **Minimum value**: Traverse **left** until a leaf node is reached.
- **Maximum value**: Traverse **right** until a leaf node is reached.

Python Code

```
python
CopyEdit
def find_min(root):
    while root.left:
        root = root.left
    return root.key

def find_max(root):
    while root.right:
        root = root.right
    return root.key

# Usage
print("Minimum:", find_min(root))
print("Maximum:", find_max(root))
```

6. Find the Successor and Predecessor of a Given Node

📌 **Concept**:

- **Successor**: Smallest node in the **right subtree**.
- **Predecessor**: Largest node in the **left subtree**.

Python Code

```
python
CopyEdit
def find_successor(root):
    return find_min(root.right)

def find_predecessor(root):
    return find_max(root.left)
```

---

7. Check if a Given Tree is an AVL Tree

📌 **Concept**:

- The tree must follow **BST properties**.
- Each node must have a **balance factor** of `1, 0, or 1`.

Python Code

```
python
CopyEdit
def is_avl(root):
    if not root:
        return True
    balance = get_balance_factor(root)
    if abs(balance) > 1:
        return False
    return is_avl(root.left) and is_avl(root.right)

print("Is AVL:", is_avl(root))
```

🌿 AVL Tree - Intermediate Level Questions with Solutions

Intermediate-level AVL tree problems focus on **rotations, balancing operations, and advanced queries**.

Each problem includes:

✅ **Step-by-step algorithm**

✅ **Python implementation**

🔷 Intermediate Level (Rotations & Balancing Operations)

1. Implement Left Rotation, Right Rotation, Left-Right Rotation, and Right-Left Rotation in an AVL Tree

📌 **Concept**: Rotations are used to maintain balance in AVL trees.

- **Left Rotation (LL Case)**: Right-heavy tree → Rotate **left**.
- **Right Rotation (RR Case)**: Left-heavy tree → Rotate **right**.
- **Left-Right Rotation (LR Case)**: Left-heavy subtree → **Left Rotate, then Right Rotate**.
- **Right-Left Rotation (RL Case)**: Right-heavy subtree → **Right Rotate, then Left Rotate**.

Python Code

```
python
CopyEdit
class AVLTree:
    def rotate_right(self, y):
        x = y.left
        T2 = x.right
```

```
        x.right = y
        y.left = T2
        y.height = max(self.get_height(y.left), self.get_height(y.right)) + 1
        x.height = max(self.get_height(x.left), self.get_height(x.right)) + 1
        return x

    def rotate_left(self, x):
        y = x.right
        T2 = y.left
        y.left = x
        x.right = T2
        x.height = max(self.get_height(x.left), self.get_height(x.right)) + 1
        y.height = max(self.get_height(y.left), self.get_height(y.right)) + 1
        return y
```

---

2. Find the kth Smallest/Largest Element in an AVL Tree

📌 **Concept**:

- **Inorder traversal** gives **sorted order** of elements in a BST/AVL tree.
- **kth smallest** → Traverse **inorder** and find the **kth node**.
- **kth largest** → Traverse **reverse inorder** (right → root → left).

Python Code

```
python
CopyEdit
def inorder_traversal(root, result):
    if root:
        inorder_traversal(root.left, result)
        result.append(root.key)
        inorder_traversal(root.right, result)
```

```
def kth_smallest(root, k):
    result = []
    inorder_traversal(root, result)
    return result[k-1] if k <= len(result) else None

def kth_largest(root, k):
    result = []
    inorder_traversal(root, result)
    return result[-k] if k <= len(result) else None

# Usage
print("3rd Smallest:", kth_smallest(root, 3))
print("2nd Largest:", kth_largest(root, 2))
```

---

3. Find the Distance Between Two Nodes in an AVL Tree

📌 **Concept**:

1. Find the **Lowest Common Ancestor (LCA)** of the two nodes.
2. Compute the **distance** from LCA to both nodes.
3. Return `distance(node1, LCA) + distance(node2, LCA)`.

Python Code

```
python
CopyEdit
def find_lca(root, n1, n2):
    if not root:
        return None
    if root.key > n1 and root.key > n2:
        return find_lca(root.left, n1, n2)
    elif root.key < n1 and root.key < n2:
        return find_lca(root.right, n1, n2)
```

```
    return root

def find_distance(root, key, dist=0):
    if root is None:
        return -1
    if root.key == key:
        return dist
    if key < root.key:
        return find_distance(root.left, key, dist+1)
    return find_distance(root.right, key, dist+1)

def distance_between_nodes(root, n1, n2):
    lca = find_lca(root, n1, n2)
    if not lca:
        return -1
    d1 = find_distance(lca, n1)
    d2 = find_distance(lca, n2)
    return d1 + d2

# Usage
print("Distance between 10 and 30:", distance_between_nodes(root, 10, 30))
```

---

4. Convert a Sorted Array into a Balanced AVL Tree

📌 **Concept**:

- **Recursively** pick the **middle element** as the root.
- Recursively construct **left subtree** from the left half.
- Recursively construct **right subtree** from the right half.

Python Code

```
python
CopyEdit
```

```
def sorted_array_to_avl(arr, start, end):
    if start > end:
        return None
    mid = (start + end) // 2
    root = AVLNode(arr[mid])
    root.left = sorted_array_to_avl(arr, start, mid - 1)
    root.right = sorted_array_to_avl(arr, mid + 1, end)
    root.height = max(get_height(root.left), get_height(root.right)) + 1
    return root

arr = [10, 20, 30, 40, 50, 60, 70]
avl_root = sorted_array_to_avl(arr, 0, len(arr) - 1)
```

---

5. Find the Number of Rotations Performed While Inserting Nodes in an AVL Tree

📌 **Concept**:

- Count rotations performed **during insertion** when balancing the tree.

Python Code

```
python
CopyEdit
class AVLTreeWithRotationCount(AVLTree):
    def __init__(self):
        super().__init__()
        self.rotation_count = 0

    def rotate_right(self, y):
        self.rotation_count += 1
        return super().rotate_right(y)

    def rotate_left(self, x):
```

```
        self.rotation_count += 1
        return super().rotate_left(x)

    def insert(self, root, key):
        root = super().insert(root, key)
        return root

# Usage
avl = AVLTreeWithRotationCount()
root = None
values = [10, 20, 30, 40, 50, 25]
for v in values:
    root = avl.insert(root, v)

print("Total Rotations:", avl.rotation_count)
```

---

6. Find the Maximum Width of an AVL Tree

📌 **Concept**:

- Use **level-order traversal** to track **max nodes at any level**.

Python Code

```
python
CopyEdit
from collections import deque

def max_width(root):
    if not root:
        return 0
    q = deque([root])
    max_width = 0
    while q:
        max_width = max(max_width, len(q))
```

```
        for _ in range(len(q)):
            node = q.popleft()
            if node.left:
                q.append(node.left)
            if node.right:
                q.append(node.right)
    return max_width

print("Max Width:", max_width(root))
```

🌿 AVL Tree - Advanced Level Questions with Solutions

Advanced AVL tree problems focus on **applications, conversions, and performance optimizations**.

Each problem includes:

✅ **Step-by-step algorithm**

✅ **Python implementation**

🔷 Advanced Level (AVL Tree Applications & Performance Analysis)

1. Convert an AVL Tree into a BST by Removing Balance Enforcement

📌 **Concept**:

- **An AVL tree is a BST with balance enforcement.**
- **Remove balancing operations** while keeping BST insertion logic.
- **Inorder traversal of AVL tree** gives a **sorted array** → **Reconstruct BST**.

Python Code

```
python
CopyEdit
def store_inorder(root, arr):
    if root:
        store_inorder(root.left, arr)
        arr.append(root.key)
        store_inorder(root.right, arr)

def sorted_array_to_bst(arr, start, end):
    if start > end:
        return None
    mid = (start + end) // 2
    root = BSTNode(arr[mid])  # Create a BST node (not AVL)
    root.left = sorted_array_to_bst(arr, start, mid - 1)
    root.right = sorted_array_to_bst(arr, mid + 1, end)
    return root

def avl_to_bst(root):
    arr = []
    store_inorder(root, arr)
    return sorted_array_to_bst(arr, 0, len(arr) - 1)

# Usage
bst_root = avl_to_bst(avl_root)
```

2. Convert a BST into a Height-Balanced AVL Tree

📌 **Concept**:

- A BST **may not be balanced** → Convert it to an AVL tree.
- **Inorder traversal** to get a sorted array.
- **Rebuild AVL tree** from sorted array using **recursion**.

Python Code

```
python
CopyEdit
def bst_to_avl(root):
    arr = []
    store_inorder(root, arr)  # Store BST nodes in sorted order
    return sorted_array_to_avl(arr, 0, len(arr) - 1)

avl_root = bst_to_avl(bst_root)
```

---

3. Implement an AVL Tree Map (Key-Value Storage using AVL Tree)

📌 **Concept**:

- Store **(key, value) pairs** instead of just keys.
- **Modify AVL tree nodes** to include a **value field**.
- Perform **balanced insert, delete, and search** operations.

Python Code

```
python
CopyEdit
class AVLNode:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.left = None
        self.right = None
        self.height = 1

class AVLTreeMap:
    def insert(self, root, key, value):
```

```
        if not root:
            return AVLNode(key, value)
        if key < root.key:
            root.left = self.insert(root.left, key, value)
        elif key > root.key:
            root.right = self.insert(root.right, key, value)
        else:
            root.value = value  # Update value for existing key
        return self.balance(root)

    def search(self, root, key):
        if not root or root.key == key:
            return root.value if root else None
        return self.search(root.left, key) if key < root.key else self.search(root.rig
ht, key)

# Usage
avl_map = AVLTreeMap()
root = None
root = avl_map.insert(root, 1, "A")
root = avl_map.insert(root, 2, "B")
print("Value for key 2:", avl_map.search(root, 2))
```

---

4. Merge Two AVL Trees into a Single Balanced AVL Tree

📌 **Concept**:

- **Perform inorder traversal** of both AVL trees to get sorted arrays.
- **Merge the two sorted arrays** into a single sorted list.
- **Rebuild AVL tree** from the merged array.

Python Code

```
python
CopyEdit
def merge_sorted_arrays(arr1, arr2):
    return sorted(arr1 + arr2)

def merge_avl_trees(root1, root2):
    arr1, arr2 = [], []
    store_inorder(root1, arr1)
    store_inorder(root2, arr2)
    merged_arr = merge_sorted_arrays(arr1, arr2)
    return sorted_array_to_avl(merged_arr, 0, len(merged_arr) - 1)

merged_avl_root = merge_avl_trees(avl_root1, avl_root2)
```

---

5. Implement a Self-Balancing Interval Tree using AVL Trees

📌 **Concept**:

- Interval trees store **[low, high] intervals** and allow **efficient range queries**.
- Use **AVL tree balancing** for optimal performance.

Python Code

```
python
CopyEdit
class IntervalNode:
    def __init__(self, low, high):
        self.low = low
        self.high = high
        self.max_high = high
        self.left = None
        self.right = None
```

```
        self.height = 1

class IntervalAVL:
    def insert(self, root, low, high):
        if not root:
            return IntervalNode(low, high)
        if low < root.low:
            root.left = self.insert(root.left, low, high)
        else:
            root.right = self.insert(root.right, low, high)
        root.max_high = max(root.high, self.get_max_high(root.left), self.get_max
_high(root.right))
        return self.balance(root)

    def search_overlap(self, root, low, high):
        if not root:
            return None
        if root.low <= high and root.high >= low:
            return root.low, root.high
        if root.left and root.left.max_high >= low:
            return self.search_overlap(root.left, low, high)
        return self.search_overlap(root.right, low, high)

# Usage
interval_avl = IntervalAVL()
interval_root = None
interval_root = interval_avl.insert(interval_root, 10, 20)
interval_root = interval_avl.insert(interval_root, 15, 25)
print("Overlapping Interval for (18,22):", interval_avl.search_overlap(interval_ro
ot, 18, 22))
```