Practical No 8

Aim : Write A Program To Implement Balanced Trees & Priority Queues

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

    def right_rotate(self, y):
        x = y.left
        T2 = x.right

        # Perform rotation
        x.right = y
        y.left = T2

        # Update heights
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        x.height = 1 + max(self.get_height(x.left), self.get_height(x.right))
        return x

    def left_rotate(self, x):
        y = x.right
        T2 = y.left

        # Perform rotation
        y.left = x
        x.right = T2

        # Update heights
        x.height = 1 + max(self.get_height(x.left), self.get_height(x.right))
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        return y

    def insert(self, node, key):
        # 1. Normal BST insert
        if not node:
            return AVLNode(key)
        elif key < node.key:
            node.left = self.insert(node.left, key)
        else:
```

```
            node.right = self.insert(node.right, key)

        # 2. Update height
        node.height = 1 + max(self.get_height(node.left), self.get_height(node.right))

        # 3. Balance factor
        balance = self.get_balance(node)

        # 4. Rebalance cases
        # Left Left
        if balance > 1 and key < node.left.key:
            return self.right_rotate(node)
        # Right Right
        if balance < -1 and key > node.right.key:
            return self.left_rotate(node)
        # Left Right
        if balance > 1 and key > node.left.key:
            node.left = self.left_rotate(node.left)
            return self.right_rotate(node)
        # Right Left
        if balance < -1 and key < node.right.key:
            node.right = self.right_rotate(node.right)
            return self.left_rotate(node)

        return node

    def inorder(self, root):
        return self.inorder(root.left) + [root.key] + self.inorder(root.right) if root else []

# ---------- Priority Queue (Min-Heap) ----------
import heapq

class PriorityQueue:
    def __init__(self):
        self.queue = []

    def push(self, priority, task):
        """Insert into priority queue (min-heap)."""
        heapq.heappush(self.queue, (priority, task))

    def pop(self):
        """Remove and return highest priority task (lowest priority number)."""
        if self.queue:
            return heapq.heappop(self.queue)
        return None

    def show(self):
        return sorted(self.queue)

# ---------- Example Usage ----------
if __name__ == "__main__":
    print("\n--- AVL Tree Example ---")
    avl = AVLTree()
```

```
root = None
values = [10, 20, 30, 40, 50, 25]

for v in values:
    root = avl.insert(root, v)
    print(f"Inserted {v}, In-order Traversal: {avl.inorder(root)}")

print("\nFinal Balanced AVL Tree (In-order):", avl.inorder(root))

print("\n--- Priority Queue Example ---")
pq = PriorityQueue()
pq.push(2, "Job A")
pq.push(1, "Emergency Patient")
pq.push(3, "Routine Checkup")
pq.push(5, "Background Task")
pq.push(4, "Job B")

print("Current Queue:", pq.show())
while pq.queue:
    priority, task = pq.pop()
    print(f"Processing Task: {task} (Priority {priority})")
```

```
--- AVL Tree Example ---
Inserted 10, In-order Traversal: [10]
Inserted 20, In-order Traversal: [10, 20]
Inserted 30, In-order Traversal: [10, 20, 30]
Inserted 40, In-order Traversal: [10, 20, 30, 40]
Inserted 50, In-order Traversal: [10, 20, 30, 40, 50]
Inserted 25, In-order Traversal: [10, 20, 25, 30, 40, 50]

Final Balanced AVL Tree (In-order): [10, 20, 25, 30, 40, 50]

--- Priority Queue Example ---
Current Queue: [(1, 'Emergency Patient'), (2, 'Job A'), (3, 'Routine Checkup'), (4, 'Job
B'), (5, 'Background Task')]
Processing Task: Emergency Patient (Priority 1)
Processing Task: Job A (Priority 2)
Processing Task: Routine Checkup (Priority 3)
Processing Task: Job B (Priority 4)
Processing Task: Background Task (Priority 5)
```