**Technical Design Document: Baggage Flow System**

**1. Introduction**

This document outlines the design of a high-performance Baggage Flow System. The system's objective is to robustly manage two critical airport operations: the individual tracking of all luggage and the prioritized loading of baggage onto aircraft. To achieve guaranteed efficiency, the architecture employs a self-balancing **AVL Tree** for cataloging and a **Min Heap** (as a Priority Queue) for operational prioritization. This combination ensures the system remains performant even under non-ideal, real-world data patterns.

**2. Data Structure Selection & Trade-offs**

**2.1. AVL Tree**

* **Usage**: Serves as the master catalog for all baggage, indexed by a unique baggage\_id.
* **Justification**: A production-grade system requires guaranteed performance. While a standard BST offers good average-case speed, it can degrade if data is inserted in a sorted or predictable manner. An AVL tree is chosen to mitigate this risk entirely. Its self-balancing nature ensures that all catalog operations—insertion, searching, and deletion—are always executed in logarithmic time.
* **Trade-offs**:
  + **Pro**: Guaranteed O(logn) time complexity for all major operations, eliminating the worst-case O(n) scenario of a standard BST. This makes the system resilient and predictable.
  + **Con**: The implementation is more complex due to the need for height tracking and rebalancing logic (rotations). This adds a small, constant overhead to insertion and deletion operations. However, this is a necessary trade-off for achieving a truly robust system.

**2.2. Min Heap (Priority Queue)**

* **Usage**: Manages the loading queue, determining which bag should be loaded onto the plane next based on a "Last-In, First-Out" (LIFO) priority scheme.
* **Justification**: A Min Heap is the most efficient data structure for priority queue operations. It provides constant time (O(1)) access to the highest-priority item and logarithmic time (O(logn)) for updates.
* **Trade-offs**:
  + **Pro**: Optimal performance for a priority-based workflow.
  + **Con**: Not designed for searching arbitrary elements (O(n)). This is an acceptable trade-off as all search operations are handled by the AVL tree.

**3. Algorithm Justification**

**3.1. AVL Tree vs. Standard BST**

The core algorithm for managing our baggage catalog is the **AVL tree's self-balancing insertion**.

* **Chosen Algorithm**: AVL Insertion/Search.
* **Alternative**: Standard Binary Search Tree (BST) Insertion/Search.
* **Justification**: A standard BST is simple, but its performance is unreliable. If baggage is processed with sequential IDs, a BST becomes an unbalanced chain, making search times linear (O(n)). The **AVL tree algorithm**, with its built-in rotation logic, **guarantees** that the tree remains balanced. This ensures that search and insert operations are always fast and predictable (O(logn)), which is critical for a time-sensitive, production-grade system. We trade a small amount of insertion complexity for rock-solid performance reliability.

**3.2. Min Heap vs. Sorted List**

To manage the loading queue, we use a **priority queue** implemented with a Min Heap.

* **Chosen Algorithm**: heapq.heappush and heapq.heappop.
* **Alternative**: Adding to a list and re-sorting it.
* **Justification**: The heap algorithm is fundamentally more efficient for this task. Every time we add a bag, heappush integrates it into the queue in O(logn) time. The alternative, adding the bag to a list and re-sorting the entire list, would take O(nlogn) time. For an airport processing thousands of bags, the heap's logarithmic performance is vastly superior to the nearly linear time of the sorting approach, preventing the loading queue from becoming a system bottleneck.

**4. System Architecture and Flowcharts**

**4.1. High-Level Diagram**

+-----------------+

[New Baggage] ---------------->| add\_baggage() |

| function |

+-------+---------+

|

+-------------------+-------------------+

| |

v v

+-------------------------+ +--------------------------+

| AVL Tree | | Min Heap |

| (Catalog by baggage\_id) | | (Priority Queue) |

+-------------------------+ +--------------------------+

^ |

| v

+------------------+------+ +--------------------------+

| find\_baggage\_by\_id() | | load\_next\_bag() |

+-------------------------+ +--------------------------+

**5. Core Functions & Complexity Analysis**

**5.1. BaggageFlowSystem.add\_baggage()**

* **Time Complexity**: O(logn)
* **Space Complexity**: O(1)

**5.2. AVLTree.insert()**

* **Time Complexity**: O(logn)
* **Space Complexity**: O(logn)

**5.3. AVLTree.search()**

* **Time Complexity**: O(logn)
* **Space Complexity**: O(logn)

**5.4. BaggageFlowSystem.load\_next\_bag()**

* **Time Complexity**: O(logn)
* **Space Complexity**: O(1)

**6. Pseudocode**

**6.1. AVL Tree Insertion**

Code snippet

PROCEDURE insert(node, baggage):

// 1. Perform standard BST insertion

IF node IS NULL:

RETURN CREATE\_NODE(baggage)

IF baggage.id < node.key:

node.left = insert(node.left, baggage)

ELSE:

node.right = insert(node.right, baggage)

// 2. Update height and get balance factor

node.height = 1 + MAX(getHeight(node.left), getHeight(node.right))

balance = getBalance(node)

// 3. If unbalanced, perform rotations

IF balance > 1 AND baggage.id < node.left.key: // Left-Left Case

RETURN rightRotate(node)

IF balance < -1 AND baggage.id > node.right.key: // Right-Right Case

RETURN leftRotate(node)

// ... Handle other two-step rotation cases ...

RETURN node

END PROCEDURE

**7. Benchmarks (Qualitative)**

The AVL tree architecture provides superior, guaranteed performance compared to both a naive list-based approach and a standard BST.

| Operation | Chosen Architecture (AVL + Heap) | Naive Approach (Single List) | Justification |
| --- | --- | --- | --- |
| **Find a Bag** | **Guaranteed** O(logn) | O(n) | The AVL tree's balanced structure ensures fast lookups, unlike a linear scan. |
| **Load Next Bag** | O(logn) | O(n) | The heap provides logarithmic performance versus a linear scan to find the correct priority item in a list. |

**Conclusion**: The use of an AVL tree makes the system's catalog robust and reliable, preventing the performance bottlenecks that a standard BST could face with non-random data. This architecture is suitable for a production environment where predictable performance is a requirement.