**1. Introduction and System Goals**

Airlines handle millions of bags daily, making efficient tracking critical. When a bag is reported missing, the system must perform two key operations with high efficiency: retrieve the bag's metadata and last known location instantly, and trace its entire journey through various checkpoints.

This document outlines the design of a **Lost Baggage Tracker** system that achieves these goals by combining a **Hash Table** and a **Doubly Linked List**. The system is designed to support dynamic insertion and deletion of baggage records while ensuring constant-time lookups for the most recent data.

**2. Data Structure Selection and Justification**

The core of this system is a hybrid data structure model that leverages the individual strengths of hash tables and doubly linked lists to meet the demanding performance requirements.

**2.1 Hash Table (Python dict)**

* **Purpose**: To provide a direct mapping from a unique baggage\_id to its most recent scan record (BaggageNode). This enables instant access to any bag's current status.
* **Justification vs. Alternatives**:
  + **Array / List**: A simple list of bags would require a linear scan to find a specific bag, resulting in an O(n) search time. This is unacceptable for a system requiring real-time lookups.
  + **Balanced Binary Search Tree (BST)**: A BST offers efficient O(logn) search times. However, a hash table provides a superior average-case time complexity of O(1), which is ideal for our primary goal of instant lookups. The overhead of maintaining tree balance is unnecessary as we do not require sorted traversal of baggage IDs.
* **Trade-offs**: The primary trade-off is memory usage, as hash tables can have a higher memory footprint than a simple list. Additionally, in the rare case of extensive key collisions, performance can degrade to O(n) in the worst case. Modern hashing algorithms make this highly unlikely.

**2.2 Doubly Linked List (DLL)**

* **Purpose**: To maintain a chronological, ordered history of scan events for each individual bag. Each bag's journey is its own DLL.
* **Justification vs. Alternatives**:
  + **Dynamic Array / List**: Storing a bag's history in a dynamic array would make insertions inefficient. Adding a new scan would require appending to the array, and if an erroneous scan needed to be removed from the middle, it would be an O(m) operation (where m is the number of scans), as subsequent elements must be shifted. A DLL provides O(1) insertion at the head or tail.
  + **Singly Linked List (SLL)**: An SLL is a strong candidate, but a DLL was chosen for its **bidirectional traversal**. Our primary "trace" operation involves traversing backwards from the latest scan using the prev pointer. The inclusion of a next pointer provides crucial flexibility for future enhancements (e.g., traversing forward from a specific checkpoint) and simplifies node deletion from the middle of the list.
* **Trade-offs**: A DLL has a higher memory overhead compared to an SLL, as each node must store an additional prev pointer. The insertion and deletion logic is also slightly more complex due to the need to manage both prev and next pointers.

**3. System Architecture and Flow**

The hash table (baggage\_map) acts as the central index. Each key (baggage\_id) points to the **tail node** of a doubly linked list that represents that specific bag's journey.

**3.1 System Diagram**

Hash Table (baggage\_map) Doubly Linked Lists (One per bag)

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

| "BAG-UA-123" | --+

+------------------+ | (prev) (prev) (prev)

| "BAG-DL-456" | --+ +------+ +------+ +------+

+------------------+ | | null | <----- |Scan 1| <----- |Scan 2| (latest scan / Tail)

| +------+ +------+ +------+

+----->|Scan 1| |Scan 2| -----> | null |

+------+ +------+ (next)

+------------------+ (prev) (prev)

| "BAG-AA-789" | --+ +------+ +------+

+------------------+ | | null | <----- |Scan A| <----- |Scan B| (latest scan / Tail)

| +------+ +------+ +------+

+----->|Scan A| |Scan B| -----> | null |

+------+ +------+ (next)

**3.2 add\_scan Operation Flowchart**

[Start] -> [Receive bag\_id, checkpoint, metadata]

|

V

[Create a new BaggageNode]

|

V

<Does bag\_id exist in baggage\_map?>

|

/ \

/ \

YES NO

| |

| V

| [New node's prev and next are null]

| |

| |

| V

| [Link new node to end of existing list]

| [last\_node.next = new\_node]

| [new\_node.prev = last\_node]

| |

\ /

\ /

V

[Update baggage\_map[bag\_id] to point to the new\_node]

|

V

[End]

**4. Core Function Analysis**

**4.1 add\_scan(baggage\_id, checkpoint, metadata)**

* **Purpose**: Adds a new checkpoint scan to a bag's history.
* **Time Complexity**: **O(1)**
  + Accessing the hash table to find the bag's last node is an average-case O(1) operation. All subsequent pointer manipulations (next, prev) are also constant-time operations.
* **Space Complexity**: **O(1)**
  + The function creates exactly one new BaggageNode object, consuming a constant amount of additional space.
* **Pseudocode**:
* FUNCTION add\_scan(bag\_id, checkpoint, metadata):
* new\_node = CREATE\_NODE(bag\_id, checkpoint, metadata)
* IF bag\_id is a key in baggage\_map:
* last\_node = GET baggage\_map[bag\_id]
* last\_node.next = new\_node
* new\_node.prev = last\_node
* SET baggage\_map[bag\_id] = new\_node

**4.2 get\_last\_known\_location(baggage\_id)**

* **Purpose**: Retrieves the most recent scan record for a given bag.
* **Time Complexity**: **O(1)**
  + This operation is a direct lookup in the hash table, which is an average-case O(1) operation.
* **Space Complexity**: **O(1)**
  + No new space is allocated; a reference to an existing object is returned.
* **Pseudocode**:
* FUNCTION get\_last\_known\_location(bag\_id):
* RETURN GET baggage\_map[bag\_id] OR null if not found

**4.3 trace\_baggage\_history(baggage\_id)**

* **Purpose**: Returns the complete, ordered journey of a specific bag.
* **Time Complexity**: **O(M)**
  + Where **M is the number of checkpoints for that specific bag**. The function finds the entry point in O(1) and then traverses backwards only along that bag's linked list.
* **Space Complexity**: **O(M)**
  + A new list is created to store the history, so the space required is proportional to the length of that bag's journey.
* **Pseudocode**:
* FUNCTION trace\_baggage\_history(bag\_id):
* last\_node = get\_last\_known\_location(bag\_id)
* IF last\_node is null:
* RETURN EMPTY\_LIST
* history = CREATE\_LIST()
* current\_node = last\_node
* WHILE current\_node is not null:
* ADD current\_node to history
* current\_node = current\_node.prev
* REVERSE history
* RETURN history

**5. Benchmarks (Theoretical Analysis)**

This table compares the time complexity of our proposed solution against a naive approach using a single list for all scan events.

Let N be the total number of unique bags in the system.

Let S be the total number of scan events across all bags.

Let M be the average number of scans for a single bag (SapproxNtimesM).

| Operation | Naive Approach (Single List of all Scans) | Proposed Approach (Hash Table + DLL) | Justification |
| --- | --- | --- | --- |
| **Add Scan** | O(1) | **O(1)** | Both approaches can add a new scan in constant time (append to list). |
| **Get Last Known Location** | O(S) | **O(1)** | The naive approach must scan the entire list to find all events for a bag and determine the latest. Our approach is a direct hash map lookup. |
| **Trace Baggage History** | O(S) | **O(M)** | The naive approach must filter all S scans to find the ones for a specific bag. Our approach traverses only the M relevant nodes. |

**Conclusion**: The proposed architecture provides an exponential performance improvement for the critical operations of lookup and tracing, making it far superior for this application.