Efficient Baggage Management for Airports

Using Data Structures and Algorithms

Baggage Routing and Tracking Platform

- Secure Baggage Handling:

A secure system is crucial for tracking and safeguarding luggage to prevent theft, loss, and tampering.

- Scalability and Performance:

The system must handle high volumes of luggage, especially during peak times, to ensure smooth operations.

- Efficient Baggage Allocation:

Effective tracking and routing of baggage based on flight details and terminal locations optimize the handling process.

Data Structure Selection: Considerations

01

Data Types and Operations

Manage data such as baggage ID, passenger details, flight information, terminal numbers, and routing status.

02

Performance Requirements

Utilize data structures that allow for efficient insertion, search, and update operations, as baggage data needs constant tracking.

03

Memory Management

Choose data structures that optimize memory usage for handling thousands of baggage records without performance lags.

Baggage Check-In and Sorting Process

Unique Baggage ID Allocation:

Use a hash set to manage unique baggage IDs, ensuring no duplicate entries in the system.

Queue Management for Sorting:

Utilize a queue to manage baggage handling in the order of check-in, ensuring a systematic flow.

Baggage Lookup by ID:

Quickly locate baggage details using hash tables for efficient and secure tracking.

Data Structures to Use

- HashSet: For unique baggage ID verification to avoid duplicates.
- Queue: To maintain order during the sorting and handling of baggage.
- Hash Table: To store and retrieve baggage details, enabling fast lookups.

Sample Code for Baggage Check-In:

```python
class BaggageCheckIn:
    def __init__(self):
        self.baggage_ids = set()  # HashSet for unique baggage IDs
        self.queue = []           # Queue for baggage sorting
        self.baggage_data = {}    # Hash table for storing baggage details

    def check_in_baggage(self, baggage_id, passenger_name, flight_no):
        if baggage_id in self.baggage_ids:
            print("Baggage already checked in.")
            return
        self.baggage_ids.add(baggage_id)
        self.queue.append(baggage_id)
        self.baggage_data[baggage_id] = {
            "passenger_name": passenger_name,
            "flight_no": flight_no
        }
        print(f"Baggage {baggage_id} checked in successfully!")
```

Linked Lists for Conveyor Belt Tracking

Baggage Movement

Represent each baggage item as a node in a linked list, tracking its position as it moves along the conveyor belt.

Efficiency

Linked lists allow for easy addition and removal of baggage nodes, ideal for handling dynamic belt operations.

Traversal

Traverse the linked list to track the current position of each baggage item on the belt.

Sample Code for Conveyor Belt Linked List:

```
class BaggageNode:
    def __init__(self, baggage_id):
        self.baggage_id = baggage_id
        self.next = None

class ConveyorBelt:
    def __init__(self):
        self.head = None
        self.tail = None

    def add_baggage(self, baggage_id):
        new_baggage = BaggageNode(baggage_id)
        if not self.head:
            self.head = new_baggage
            self.tail = new_baggage
        else:
            self.tail.next = new_baggage
            self.tail = new_baggage
```

Hash Tables for Passenger-Baggage Lookup

Insertion

Add baggage details to the hash table.

Lookup

Retrieve baggage details by baggage ID or passenger ID.

Update/Deletion

Update or remove baggage data as needed.

Sample Code for Baggage Lookup:

```
class BaggageLookup:
    def __init__(self):
        self.baggage_records = {}  # Dictionary as a hash table for
baggage

    def add_baggage(self, baggage_id, details):
        self.baggage_records[baggage_id] = details

    def get_baggage_details(self, baggage_id):
        return self.baggage_records.get(baggage_id, "Baggage not
found.")

    def remove_baggage(self, baggage_id):
        if baggage_id in self.baggage_records:
            del self.baggage_records[baggage_id]
            return f"Baggage {baggage_id} removed."
        else:
            return "Baggage ID not found."
```

Graphs for Routing and Path Optimization

Optimized Path for Routing

Use graph algorithms like Dijkstra's to find the shortest and most efficient path for baggage routing within the airport.

Real-Time Updates for Baggage Routing

Incorporate real-time updates to reroute baggage if terminals or conveyors are congested or closed.

Routing Flexibility

Use graphs to reroute baggage in case of flight changes or terminal adjustments.

Algorithm for Optimized Baggage Routing

Add Terminal:

- Input: terminal_id
- Procedure: Create a node in the graph representing each terminal and connecting routes.

Find Shortest Path for Baggage Routing:

- Input: start_terminal, destination_terminal
- Procedure: Use Dijkstra's algorithm to find the shortest path for routing baggage from check-in to loading.

Update Routes Dynamically:

- Adjust paths as needed based on real-time conditions to optimize baggage flow.

Real-Time Tracking and Queue Management

- Real-Time Data Updates:
  Update baggage status, flight delays, and gate changes in real time for accurate tracking.

- Queue Management for Baggage Loading:
  Manage the order of baggage for loading onto planes, optimizing loading based on flight departure times.

- Alerts and Notifications:
  Send alerts for critical updates, such as flight delays or baggage rerouting, to ensure timely handling.

Conclusion

- Using DSA in baggage handling systems optimizes baggage tracking, sorting, and routing for efficient airport operations.
- Real-time updates minimize baggage delays and ensure timely loading for on-time departures.
- Efficient data structures improve security, reduce baggage loss, and create a seamless experience for passengers and staff.

THANK
YOU

Namita Nahata

RA2311026010677

AA-2