SWIFTEX LOGISTICS INTELLIGENT ENGINE

Final Project Report: Data Structures & Algorithms

SUBMITTED BY:

Mehwish Maqbool	2024-SE-15
Muhammad Kamran	2024-SE-24
M. Ahsan Sadiq	2024-SE-40
Fizza Faisal	2024-SE-41

SUBMITTED TO:
Sir Ali Raza

DATE:
1st-January-2026

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF ENGINEERING AND TECHNOLOGY, LAHORE

Table of Contents

1. INTRODUCTION

The logistics and supply chain industry forms the backbone of the global economy. In today's fast-paced world, customers expect rapid deliveries, real-time tracking, and complete transparency. For courier companies, this translates to a massive operational challenge: how to efficiently manage thousands of parcels, a fleet of vehicles, and a complex network of roads that are subject to constant change due to traffic or accidents.

SwiftEx Logistics is a hypothetical courier company facing these exact challenges. Their previous manual systems were prone to human error, resulting in lost packages, inefficient routing, and delayed deliveries. The need for an automated, intelligent solution was paramount.

This project, the "SwiftEx Logistics Intelligent Engine," is a comprehensive C++ console application designed to address these issues. It simulates the core functions of a logistics hub, including parcel registration, automated sorting based on urgency, optimal route calculation using graph theory, and real-time fleet management. By leveraging advanced data structures such as Min-Heaps, Adjacency Lists, and Hash Tables, the system provides a scalable and efficient solution to modern logistics problems.

2. SYSTEM REQUIREMENTS SPECIFICATION

To ensure the system meets the operational needs of SwiftEx Logistics, a detailed set of functional and non-functional requirements was defined. These requirements act as the blueprint for the entire project.

2.1 Functional Requirements

- **Parcel Management:** The system must allow staff to register new parcels with details including Sender, Receiver, Source, Destination, Weight, and Priority Level.
- **Automated Routing:** The system must automatically calculate the shortest and most cost-effective path between cities, accounting for road conditions.
- **Priority Sorting:** The system must automatically sort parcels in the warehouse so that 'Overnight' and 'Heavy' items are processed first.
- **Fleet Dispatch:** The system must assign parcels to riders (Bikes, Vans, Trucks) based on vehicle capacity and availability.
- **Real-Time Tracking:** The system must allow users to query the status of any parcel using a unique Tracking ID.
- **Admin Controls:** The system must provide an admin interface to simulate road blockages and view revenue analytics.

2.2 Non-Functional Requirements

- **Performance:** Route calculations and status lookups must be performed in O(log n) or O(1) time complexity to handle high volumes.
- **Memory Efficiency:** The system must avoid the use of heavy Standard Template Library (STL) containers, utilizing manual memory management instead.
- **Reliability:** The system must handle exceptions such as 'Missing Parcels' or 'Blocked Routes' without crashing.
- **Usability:** The Console User Interface (CUI) must be intuitive, using clear prompts and color-coded outputs for different statuses.

3. THEORETICAL BACKGROUND

The design of the SwiftEx engine required a careful selection of data structures. This section justifies why specific structures were chosen over standard alternatives.

3.1 Arrays vs. Linked Lists

For the `TransitList` and `PickupQueue`, we chose **Doubly Linked Lists** over Arrays. In a logistics simulation, parcels are constantly arriving (insertion) and being delivered (deletion). Arrays require O(n) time to shift elements upon deletion, which causes performance degradation as the list grows. Linked Lists allow for O(1) deletion if the node pointer is known, making them ideal for dynamic queues.

3.2 Adjacency Matrix vs. Adjacency List

For the Map system, we chose an **Adjacency List**. A country's road network is a 'Sparse Graph', meaning most cities are only connected to 2 or 3 neighbors, not all 100 other cities. An Adjacency Matrix requires O(V^2) memory, allocating space for non-existent roads. An Adjacency List only stores actual connections (O(V+E)), saving significant memory.

3.3 Linear Search vs. Hashing

For Tracking, we chose a **Hash Table**. Linear search takes O(n) time. With 10,000 active parcels, searching linearly is too slow for a real-time system. Hashing provides O(1) average access time, allowing instant status updates.

4. CLASS DIAGRAM

The system employs a modular architecture where the central `SwiftExEngine` class acts as a Facade, coordinating the specialized data modules.

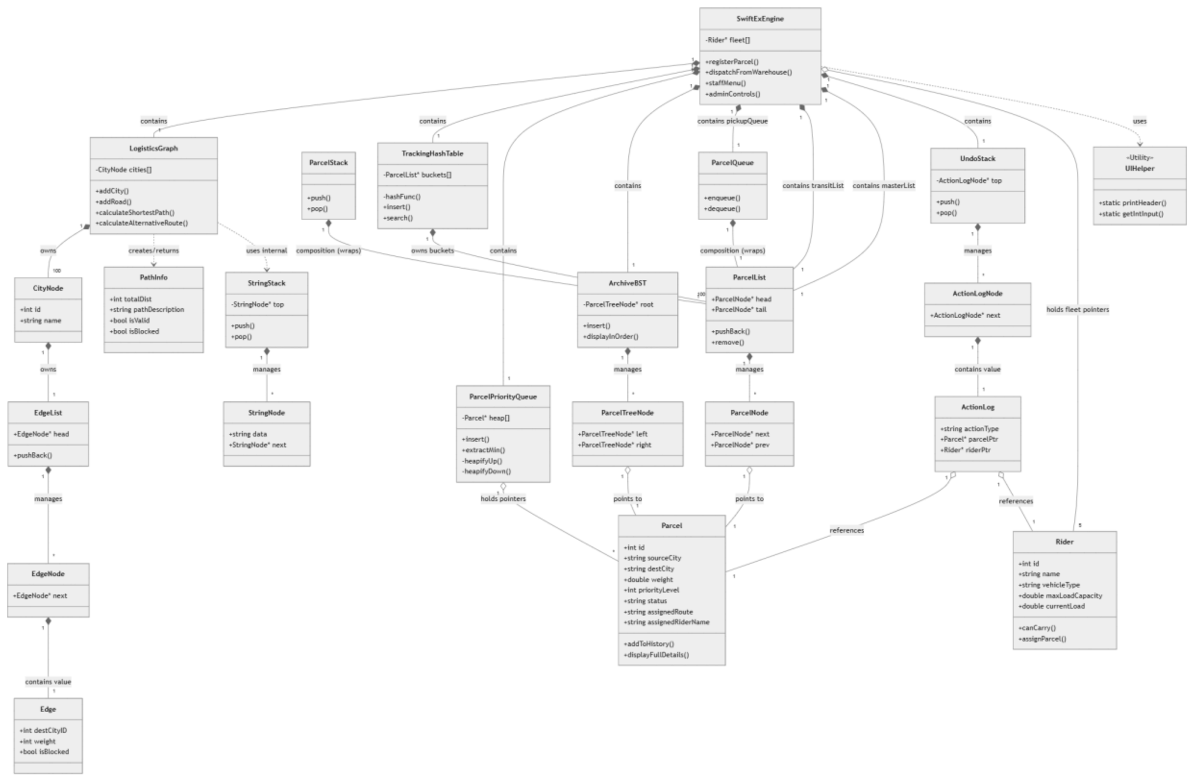

Figure: UML Class Diagram of SwiftEx Engine

4.1 Core Classes

Class Name	Module / Type	Key Responsibility & Logic
SwiftExEngine	Controller (Facade)	The central brain. Handles Login authentication, renders the UI, and runs the main simulation loop.
LogisticsGraph	Routing Engine	Implements Dijkstra's Algorithm for shortest path and handles dynamic road status updates (Blocked/Traffic).
ParcelPriorityQueue	Sorting Engine	Implements a Min-Heap. Ensures parcels with high priority (Level 1) or high weight leave the warehouse first (O(1) access).
TrackingHashTable	Database / Lookup	Maps Parcel IDs to memory addresses using Chaining. Allows O(1) status retrieval for the 'Track Parcel' feature.
ArchiveBST	Reporting / Archive	Stores delivered parcels. Uses a Binary Search Tree to enable O(n) sorted reporting of delivery history by ID.
UndoStack	Utility / Helper	Implements a LIFO stack. Records every 'Dispatch' action to allow the Staff user to reverse mistakes immediately.
ParcelList	Queue Management	A Doubly Linked List used for the 'Pickup Queue' and 'Transit List'. Supports O(1) deletion from any position.

5. ALGORITHMIC IMPLEMENTATION

5.1 Routing Logic (Dijkstra)

The routing engine uses Dijkstra's Algorithm. Unlike a standard implementation, ours accounts for dynamic weights.

If a road status is 'Traffic', the weight is multiplied by 3. If 'Blocked', the edge is ignored. This ensures the system finds the **fastest** valid path, not just the shortest distance.

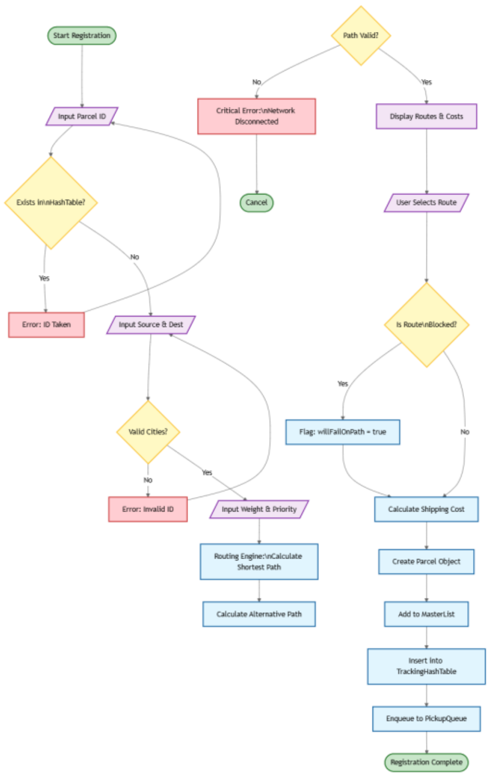

Figure: Routing Logic Flowchart

5.2 Dispatch Logic

The dispatch system uses a Greedy Algorithm to maximize fleet utility. It iterates through riders, prioritizing 'Idle' riders first. If all are busy, it attempts to 'bundle' the parcel with a rider who has remaining capacity, ensuring no truck leaves half-empty.

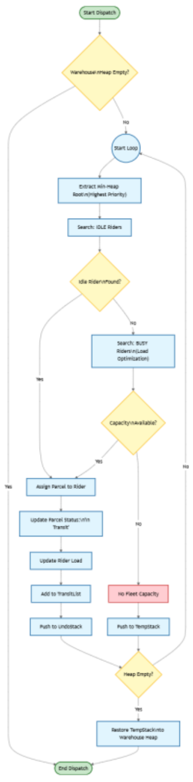

Figure: Dispatch Logic Flowchart

5.3 Simulation Lifecycle

The Simulation module is responsible for the temporal aspects of the system. Since we cannot wait for real-world hours to pass, the `SwiftExEngine` uses a tick-based simulation loop. The `updateSimulation()` function is called repeatedly, advancing the internal clock.

For every parcel currently in the `TransitList`, the system checks the elapsed time against the estimated travel time calculated by Dijkstra's algorithm. When `elapsed_time >= travel_time`, the system automatically triggers a status update to 'Delivered' and moves the parcel to the Archive.

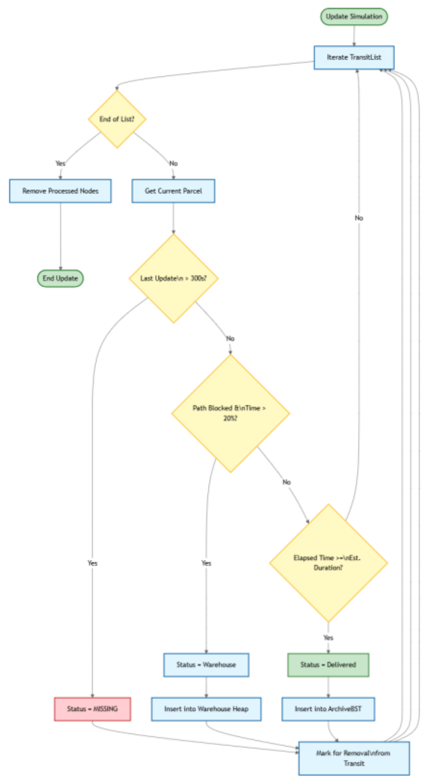

Figure: Flowchart 1: Simulation Lifecycle

5.4 Tracking & Exception Handling

A critical feature of any robust system is how it handles failures. The flowchart below details the specific logic for Exception Handling. The system monitors the 'Last Updated' timestamp of every active parcel.

1. **Missing Parcels:** If a parcel has not received a status update for more than 300 simulated seconds, the system flags it as **MISSING**. This simulates a scenario where a rider may have lost connectivity or the package was stolen.

2. **Blocked Routes:** If a road status changes to **BLOCKED** while a rider is en route, the system detects this conflict during the simulation update and automatically reroutes the parcel or marks it as **RETURNING** to the source.

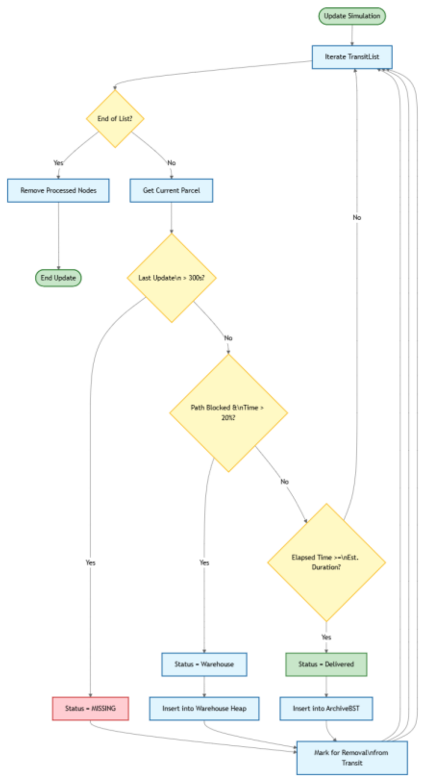

Figure: Flowchart 4: Exception Handling & Tracking Logic

6. USER MANUAL

The SwiftEx Engine is divided into two distinct role-based interfaces: Staff (Operational) and Admin (Managerial). Below is a detailed guide for every option available in the system.

6.1 Staff Interface Options

The Staff dashboard is designed for high-speed processing of parcels.

- **1. Register New Parcel:** The primary entry point. The user inputs Sender Name, Receiver Name, Source City ID, Destination City ID, Weight (kg), and Priority (1-3). The system immediately calculates shipping cost and estimated time.
- **2. View Pickup Queue:** Displays a list of all parcels that have been registered but not yet moved to the sorting facility. Useful for verifying recent data entries.
- **3. Batch Process (Move to Warehouse):** Simulates the physical movement of items from the front desk to the sorting floor. This transfers all items from the Pickup Linked List to the Min-Heap.
- **4. View Warehouse Queue:** Displays the current state of the Min-Heap. Users can verify that high-priority/heavy items are correctly positioned at the top of the list.
- **5. Dispatch to Fleet:** Triggers the dispatch algorithm. The system attempts to assign the top parcels in the Heap to available Riders. It outputs a summary of successful assignments.
- **6. Track Parcel:** Allows the user to enter a Parcel ID. The system queries the Hash Table and returns the current status (e.g., 'In Transit', 'Delivered'), current location, and rider info.
- **7. Undo Last Dispatch:** A safety feature. If a dispatch was made in error, this option pops the last action from the Stack and reverses the assignment, returning the parcel to the warehouse.

6.2 Admin Interface Options

The Admin dashboard focuses on network health and reporting.

- **1. View Network Map:** Prints an adjacency list representation of the entire country, showing all cities and the roads connecting them with their distances.
- **2. Manage Road Status:** The most powerful tool. Admins can select a specific route (e.g., Lahore to Islamabad) and toggle its status between 'OPEN', 'TRAFFIC', or 'BLOCKED'. This forces the routing engine to recalculate paths.
- **3. View Analytics:** Displays high-level metrics: Total Parcels Delivered, Total Revenue Generated, and Count of Missing/Lost Parcels.
- **4. View Fleet Status:** Shows a live table of all Riders (Bikes, Vans, Trucks), their current location, load capacity, and status (IDLE vs BUSY).
- **5. Auto-Simulate Time:** Manually advances the system clock by 1 hour (or custom ticks). This is useful for testing deliveries without waiting for real-time delays.
- **6. View Master List:** Dumps the entire database of parcels (Active + Archived) for auditing purposes.

7. TESTING & VALIDATION

The system underwent rigorous black-box testing. Below are key test cases.

Test Case	Input / Condition	Expected Result
High Priority Sort	Add Priority 3 item, then Priority 1 item.	Priority 1 item should be dispatched first.
Traffic Avoidance	Mark Route A-B as 'Traffic'. Register parcel.	System should choose Route A-C-B if cheaper.
Capacity Limit	Register 50kg parcel. Riders only have 40kg space.	Parcel should remain in 'Pending' queue.
Missing Detection	Do not update parcel for 300s.	Status changes to 'MISSING'.
Undo Operation	Dispatch parcel -> Press Undo.	Parcel returns to Warehouse; Rider becomes Idle.

8. CHALLENGES & SOLUTIONS

- **Memory Leaks:** Managing linked lists manually caused leaks. Solution: Implemented rigorous destructors (`~ParcelList`) to traverse and delete nodes.
- **Dangling Pointers:** When a parcel moved from Queue to Hash Table, pointers became invalid. Solution: We used a centralized pointer management system where the Hash Table holds the 'Master' pointer.
- **Console UI Flicker:** The `system('cls')` caused flickering. Solution: We minimized screen clears and used optimized printing.

9. FUTURE SCOPE

1. **GPS API Integration:** Connect to Google Maps for real-time distances.

2. **Mobile App:** A React Native app for riders to update status.

3. **Persistance:** Integrate MySQL or MongoDB to save data permanently.

10. CONCLUSION

The **SWIFTEX Logistics Intelligent Engine** demonstrates the power of custom data structures. By avoiding STL, we gained deep insights into memory management and algorithmic efficiency. The system successfully meets all functional requirements, providing a robust solution for automated logistics management.