# Department of Computing

# School of Electrical Engineering and Computer Science

**CS-250: Data Structure and Algorithms**

**Class: BSCS 13D  
  
Submitted by: M. Hamid Sarfraz  
CMS ID: 453812**

# 

# Lab 4: Implementation of Stacks & Queue in different problems

# Course Instructor: Dr. Syed Imran Ali

# Lab Instructor: Ms Areeba Rameen

# 

**SmartRide: Project Design (Revised)**

**1. Core Modules:**

* **User Management (user\_management.py)**
  + Handles user registration, login, profile management, and authentication.
  + Stores user data (ID, name, contact, location, etc.) in dictionaries.
* **Driver Management (driver\_management.py)**
  + Handles driver registration, login, availability management, and driver profiles.
  + Stores driver data (ID, name, vehicle, location, availability, ratings) in dictionaries.
* **Ride Request Management (ride\_request.py)**
  + Processes ride requests (including ride type: normal, emergency, rideshare).
  + Matches users with available drivers.
  + Manages ride queues.
  + Utilizes Priority Queues (Min-Heaps) for driver matching and emergency requests.
* **Location Service (location\_service.py)**
  + Handles location data, map representation (graph), and pathfinding algorithms.
  + Stores location coordinates and distances in a separate file (map\_data.py).
  + Implements Dijkstra's algorithm for shortest path calculation.
* **Pricing & Billing (pricing.py)**
  + Calculates ride fares based on distance, time, and demand.
  + Handles fare splitting for rideshare requests.
* **Rating System (rating\_system.py)**
  + Allows users to rate drivers.
  + Stores and displays driver ratings.
  + Potentially uses a balanced tree for efficient retrieval of top-rated drivers.
* **Ride History (ride\_history.py)**
  + Stores and retrieves user ride history.
  + Uses a Doubly Linked List for efficient navigation through past rides.
* **Emergency Handler (emergency\_handler.py)**
  + Handles emergency ride requests with high priority.
  + Utilizes a separate, high-priority queue for emergency requests.
  + Dispatches the nearest available emergency vehicle.
* **Social Ridesharing (social\_rideshare.py)**
  + Handles user connections (friend requests, connections).
  + Implements a graph data structure to represent user connections.
  + Matches users for rideshares based on route similarity and availability.
  + Facilitates communication between rideshare partners.

**2. Data Flow:**

1. **User/Driver Registration:** New users and drivers register, and their data is stored in the respective modules.
2. **Ride Request:**
   * User requests a ride, specifying type (normal, emergency, rideshare).
   * ride\_request.py determines the appropriate handling based on the request type.
3. **Driver Matching:**
   * For normal requests, the system matches the user with the nearest available driver using the min-heap and location data from location\_service.py.
   * For emergency requests, the emergency\_handler module prioritizes and dispatches the nearest emergency vehicle.
   * For rideshare requests, the social\_rideshare module finds potential matches based on user connections and route similarity.
4. **Route Calculation:**
   * The location\_service module calculates the shortest path using Dijkstra's algorithm.
5. **Ride Completion:**
   * Upon ride completion, the user rates the driver.
   * The rating\_system module updates the driver's rating.
   * Ride details are stored in the user's ride history.
6. **Rideshare Coordination:**
   * If a rideshare match is found, the social\_rideshare module facilitates communication between users and helps coordinate the ride.

**3. Data Storage:**

* **Primary:** Use Python's built-in data structures (dictionaries, lists) for primary data storage within each module.
* **Map Data:** Store location coordinates and distances in a separate file (map\_data.py).
* **Optional: Data Persistence:** Consider using a simple database (like SQLite) for long-term storage of user data, ride history, and system logs.

**4. User Interface (Optional):**

* Implement a basic terminal-based interface for user and driver interactions.
* Explore creating a simple GUI using Tkinter or PyQt.

**5. Testing:**

* Unit tests for individual modules (e.g., user registration, driver matching, pathfinding).
* Integration tests to verify the interaction between different modules.
* Test cases for various scenarios (e.g., successful ride requests, emergency requests, rideshare matching, edge cases).

**6. Visualization (Optional):**

* Visualize the city map using a library like NetworkX.
* Consider basic visualizations for driver locations and ride requests.

This revised design incorporates more specific details regarding the emergency\_handler.py and social\_rideshare.py modules, further enhancing the clarity and structure of the SmartRide project.