# 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

# 

**LAB: SmartRide: A Ride-Hailing Service Project**

**Objective:**

This project aims to design and implement a simplified ride-hailing service, focusing on the application of core data structures and algorithms. You will simulate key features of a ride-hailing platform, including user and driver management, ride requests, matching, and more.

**Project Requirements:**

* **Programming Language:** Python
* **Data Structures:**
  + Hash Tables (Dictionaries)
  + Priority Queues (Min-Heaps)
  + Graphs (Adjacency List)
  + Queues (FIFO)
  + Doubly Linked Lists
  + (Optional) Balanced Binary Search Trees (e.g., AVL Trees)
* **Algorithms:**
  + Hashing
  + Heap operations (insertion, deletion, heapify)
  + Graph traversal (Dijkstra's algorithm)
  + Queue operations (enqueue, dequeue)
  + Linked list operations (insertion, deletion, traversal)
  + (Optional) Tree rotations (for balanced trees)
* **Features:**

Here are brief descriptions of the key features of the SmartRide project (the proposed data structures are suggestions, you are encouraged to use what you seem best):

**1. User and Driver Management**

* **Functionality:**
  + Registration and login for both users and drivers.
  + Storing user and driver profiles (name, contact, location, vehicle details, etc.).
* **Data Structures:**
  + **Dictionaries:** To efficiently store and retrieve user and driver information.

**2. Ride Request Matching**

* **Functionality:**
  + Matching user ride requests with available drivers.
  + Prioritizing drivers based on proximity to the user.
* **Data Structures:**
  + **Priority Queue (Min-Heap):** To efficiently prioritize drivers based on distance.

**3. Real-Time Location Tracking & Shortest Path**

* **Functionality:**
  + Tracking user and driver locations (simulated in this project).
  + Calculating the shortest path between the user's location and the driver's location.
* **Data Structures:**
  + **Graph:** To represent the city map with nodes (locations) and edges (roads).
  + **Algorithms:** Dijkstra's algorithm for finding the shortest path.

**4. Ride Request Queue**

* **Functionality:**
  + Storing ride requests when no drivers are immediately available.
  + Processing requests in a first-come, first-served order.
* **Data Structures:**
  + **Queue (FIFO)**

**5. Driver Rating System**

* **Functionality:**
  + Allowing users to rate drivers after each ride.
  + Storing and displaying driver ratings.
* **Data Structures:**
  + **Dictionaries** to store driver ratings.
  + **Potentially a balanced tree for efficient retrieval of top-rated drivers.**

**6. User Ride History**

* **Functionality:**
  + Storing and displaying a user's past ride history (date, time, driver, rating).
* **Data Structures:**
  + **Doubly Linked List** to allow easy navigation through past rides.

**7. Emergency Ride Requests**

* **Functionality:**
  + Prioritizing emergency ride requests (medical, fire, police).
  + Matching emergencies with the nearest available emergency vehicles.
* **Data Structures:**
  + **Priority Queue** with higher priority for emergency requests.

**8. Social Ridesharing**

* **Functionality:**
  + Enabling users to connect with friends or colleagues.
  + Matching users with compatible rideshare routes.
* **Data Structures:**
  + **Graph** to represent user connections.
  + **Algorithms** to find users with similar routes.

These are the core features of the SmartRide project, along with their key functionalities and relevant data structures. Remember that this is a simplified overview, and the actual implementation may involve more complexities.

**Project Structure:**

* **Modular Approach:** Organize your code into separate Python files for better maintainability:
  + user\_management.py
  + driver\_management.py
  + ride\_request.py
  + location\_service.py
  + pricing.py
  + rating\_system.py
  + ride\_history.py
  + emergency\_handler.py
  + social\_rideshare.py
  + map\_data.py
  + data\_structures.py
  + main.py
* **Data Structures File (data\_structures.py)**
  + Implement core data structures from scratch (hash tables, heaps, graphs, linked lists) in this file.
  + This promotes code reusability and better understanding of data structure implementations.

**Map Data Management**

* **Dedicated File:** Create a separate Python file (e.g., map\_data.py) to store all location and distance information related to the Islamabad city map.
* **Benefits:**
  + **Improved Organization:** Enhances code readability and maintainability.
  + **Reusability:** Allows for easy reuse of map data within the project and potentially in other projects.
  + **Flexibility:** Facilitates easier modification of map data without affecting other parts of the code.
* **File Structure:**
  + The map\_data.py file should contain:
    - A dictionary to store location coordinates (e.g., locations = {'F-6 Markaz': (0, 0), ...}).
    - A dictionary or other suitable data structure to store distances between locations (e.g., distances = {('F-6 Markaz', 'F-7 Markaz'): 5, ...}).
    - Optional helper functions to access and manipulate map data (e.g., get\_distance(location1, location2)).

**Defining Locations for the Islamabad City Map**

**Include the following locations in your graph:**

* **Sectors:**
  + F-6 Markaz
  + F-7 Markaz
  + G-9 Markaz
  + I-8 Markaz
  + E-11 Markaz
  + F-10 Markaz
  + G-11 Markaz
  + F-17
  + E-11
  + G-10
  + D-12
  + F-11
  + G-6
* **Universities:**
  + Quaid-i-Azam University (QAU)
  + International Islamic University Islamabad (IIUI)
  + Air University
  + FAST University
  + COMSATS University Islamabad
  + **National University of Sciences and Technology (NUST)**
* **Landmarks:**
  + Faisal Mosque
  + Shakarparian
  + Pakistan Monument
  + Daman-e-Koh
  + Lok Virsa Museum
  + Centaurus Mall
  + Serena Hotel

**Note:**

* This list provides a foundation. You can expand it based on the specific requirements and complexity of your project.

**Representing Distances:**

* **Approximate Distances:**
  + Use a combination of real-world distances and estimations to assign distances between locations.
  + You can use online mapping services (like Google Maps) to get approximate distances.
* **Graph Representation:**
  + Represent the map as a graph where:
    - Nodes represent the locations (sectors, universities, landmarks).
    - Edges represent the connections between locations (roads).
    - Edge weights represent the distances between connected locations.

**Example (Conceptual):**

* **Graph:**
  + graph = { 'F-6 Markaz': {'F-7 Markaz': 5, 'G-9 Markaz': 10}, 'F-7 Markaz': {'F-6 Markaz': 5, 'G-9 Markaz': 8, 'QAU': 3}, 'NUST': {'E-11': 2, 'I-8 Markaz': 5}, # ... other connections }
* This example shows a simplified representation where:
  + F-6 Markaz is connected to F-7 Markaz with a distance of 5 units.
  + NUST is connected to E-11 with a distance of 2 units.

**Remember:**

* **Focus on Core Concepts:** The primary goal is to demonstrate your understanding of graph data structures and pathfinding algorithms.
* **Keep it Manageable:** Start with a manageable number of locations and gradually increase complexity as needed.
* **Adapt to Your Needs:** Adjust the locations and distances based on the specific requirements and scope of your project.

By following these guidelines, you can effectively define locations and distances for your Islamabad city map and use them to implement your ride-hailing service project.

**Lab Manual Guideline:**

**Ridesharing Feature**

* **User Connections:**
  + Implement a graph data structure (e.g., adjacency list) to represent user connections within the system.
  + Allow users to establish connections with other users.
  + Consider features like friend requests and connection approvals.
* **Rideshare Matching:**
  + Develop an algorithm to match users with compatible rideshare requests based on route similarity, availability, and user preferences.
  + Utilize the user connection graph to identify potential rideshare partners.
  + Consider factors like origin, destination, and desired number of passengers.
* **Rideshare Coordination:**
  + Implement basic communication mechanisms (e.g., in-app messaging) to facilitate coordination between potential rideshare partners.
  + Explore mechanisms for fare splitting among rideshare participants.
* **Route Optimization:**
  + If a rideshare match is successful, recalculate the optimal route considering the multiple pick-up and drop-off points.
  + Modify existing pathfinding algorithms (e.g., Dijkstra's algorithm) to accommodate multiple destinations.
* **Implementation Considerations:**
  + Focus on efficient graph traversal and matching algorithms.
  + Design the user interface to support user connections, rideshare requests, and communication.
  + Consider potential limitations and challenges in implementing a robust rideshare system.

This addition to the lab manual will guide students on the key aspects of implementing the Ridesharing feature, emphasizing the use of graph data structures, algorithms, and user interaction considerations.

**Data Handling:**

* **For this project, you can use Python's built-in data structures (dictionaries, lists) for initial data storage.**
* **Simulate Real-Time Data:**
  + For location tracking and ride requests, you will need to simulate real-time data.
  + Create sample user locations, driver locations, and ride requests.
* **Data Storage (Very Optional):**
  + For a more advanced implementation, you can explore using a simple database (like SQLite) to store user data, ride history, and system logs.

**Terminal Interface (Optional):**

* **Basic Interaction:** Implement a simple terminal-based interface for user and driver interactions.
* **Key Features:**
  + User/Driver Login/Registration
  + Request Ride
  + View Ride History
  + Driver Dashboard (View/Accept/Decline Requests, Update Availability)
  + Emergency Request Handling (Simplified)
* **Keep it Simple:** Focus on core functionality and data structure implementations.

**Project Report:**

* **Documentation:**
  + Document your code thoroughly with comments and docstrings.
  + Include a separate README file summarizing the project, its features, and how to run the code.
* **Data Structures and Algorithms:**
  + Explain the choice and implementation of each data structure used in your project.
  + Discuss the time and space complexity of your algorithms.
  + Include diagrams and visualizations to illustrate the data structures and their operations.
* **Project Design:**
  + Describe the overall design of your system, including the modular structure, data flow, and how different components interact.
* **Testing:**
  + Describe your testing approach (unit tests, integration tests).
  + Include test cases and their results.
* **Conclusion:**
  + Summarize your findings and discuss potential improvements or extensions to the project.

**Evaluation Criteria:**

* **Functionality:** Correct implementation of all required features.
* **Data Structures & Algorithms:** Correct and efficient implementation of data structures and algorithms.
* **Code Quality:** Code readability, modularity, and maintainability.
* **Documentation:** Clarity and completeness of documentation.
* **Project Report:** Quality and depth of the project report.
* **Teamwork & Communication:** (If applicable) Effective collaboration within your group.

**Additional Considerations:**

* **Emergency Requests:** Prioritize emergency requests and simulate dispatching appropriate emergency vehicles.
* **Social Ridesharing:** Implement a basic matching algorithm for users with similar routes.
* **User Interface (Optional):** Explore creating a simple GUI using a framework like Tkinter or PyQt.
* **Visualization:** Consider visualizing data (e.g., driver locations on a map) to enhance user experience.

This comprehensive guideline should provide you with a solid foundation for your SmartRide project. Remember to focus on understanding and implementing the core concepts of data structures and algorithms. Good luck!

**Defining Test Cases for the SmartRide Project (these are suggestions, its better to modify them according to your own project)**

**1. User Management**

* **Test Case 1:**
  + **Test Description:** Verify successful user registration.
  + **Test Steps:**
    1. Enter valid user details (name, phone, email, password).
    2. Click "Register."
  + **Expected Result:**
    1. Registration successful.
    2. User account is created successfully.
* **Test Case 2:**
  + **Test Description:** Verify unsuccessful user registration (invalid email).
  + **Test Steps:**
    1. Enter user details with an invalid email address.
    2. Click "Register."
  + **Expected Result:**
    1. Registration fails.
    2. Error message is displayed (e.g., "Invalid email address").
* **Test Case 3:**
  + **Test Description:** Verify user login.
  + **Test Steps:**
    1. Enter valid user ID and password.
    2. Click "Login."
  + **Expected Result:**
    1. Login successful.
    2. User is redirected to the main user interface.
* **Test Case 4:**
  + **Test Description:** Verify unsuccessful user login (invalid credentials).
  + **Test Steps:**
    1. Enter incorrect user ID or password.
    2. Click "Login."
  + **Expected Result:**
    1. Login fails.
    2. Error message is displayed (e.g., "Invalid credentials").

**2. Driver Management**

* **Test Case 1:**
  + **Test Description:** Verify successful driver registration.
  + **Test Steps:**
    1. Enter valid driver details (name, phone, email, vehicle type, license number, password).
    2. Click "Register."
  + **Expected Result:**
    1. Registration successful.
    2. Driver account is created successfully.
* **Test Case 2:**
  + **Test Description:** Verify unsuccessful driver registration (missing required fields).
  + **Test Steps:**
    1. Attempt to register a driver without entering a required field (e.g., license number).
    2. Click "Register."
  + **Expected Result:**
    1. Registration fails.
    2. Error message is displayed (e.g., "Please fill in all required fields").
* **Test Case 3:**
  + **Test Description:** Verify driver login.
  + **Test Steps:**
    1. Enter valid driver ID and password.
    2. Click "Login."
  + **Expected Result:**
    1. Login successful.
    2. Driver is redirected to the driver dashboard.

**3. Ride Request Matching**

* **Test Case 1:**
  + **Test Description:** Verify successful ride request matching.
  + **Test Steps:**
    1. User requests a ride.
    2. System matches the user with the nearest available driver.
  + **Expected Result:**
    1. Driver is notified of the ride request.
    2. User receives confirmation of ride request and estimated arrival time.
* **Test Case 2:**
  + **Test Description:** Verify ride request queueing when no drivers are available.
  + **Test Steps:**
    1. User requests a ride when no drivers are available.
  + **Expected Result:**
    1. Ride request is added to the queue.
    2. User is notified that no drivers are currently available and their request is in the queue.
* **Test Case 3:**
  + **Test Description:** Verify ride request assignment to available driver.
  + **Test Steps:**
    1. User requests a ride.
    2. A driver becomes available.
  + **Expected Result:**
    1. The queued ride request is assigned to the available driver.
    2. User and driver are notified.

**4. Real-Time Location Tracking & Shortest Path**

* **Test Case 1:**
  + **Test Description:** Verify shortest path calculation.
  + **Test Steps:**
    1. Simulate a ride request with a specific pickup and drop-off location.
  + **Expected Result:**
    1. The system calculates the shortest path between the pickup and drop-off locations.
    2. The calculated path is displayed (or used for navigation).
* **Test Case 2:**
  + **Test Description:** Verify path recalculation in case of traffic changes.
  + **Test Steps:**
    1. Simulate a change in traffic conditions.
  + **Expected Result:**
    1. The system recalculates the shortest path to avoid congested areas.

**5. Driver Rating System**

* **Test Case 1:**
  + **Test Description:** Verify user can rate a driver after a ride.
  + **Test Steps:**
    1. User completes a ride.
    2. User provides a rating for the driver.
  + **Expected Result:**
    1. Driver's rating is updated successfully.

**6. User Ride History**

* **Test Case 1:**
  + **Test Description:** Verify user can view their ride history.
  + **Test Steps:**
    1. User accesses their ride history.
  + **Expected Result:**
    1. A list of past rides is displayed (date, time, driver, rating).

**7. Emergency Ride Requests**

* **Test Case 1:**
  + **Test Description:** Verify prioritization of emergency requests.
  + **Test Steps:**
    1. Submit an emergency ride request.
  + **Expected Result:**
    1. Emergency request is prioritized over other requests.
    2. The system dispatches the nearest available emergency vehicle.

**8. Social Ridesharing**

* **Test Case 1:**
  + **Test Description:** Verify user can connect with other users for potential rideshares.
  + **Test Steps:**
    1. User searches for potential rideshare partners.
  + **Expected Result:**
    1. The system displays potential matches based on route