### **PROJECT DESCRIPTION**

**Technologies Used:**

The application is built using C++, demonstrating the use of object-oriented programming, dynamic memory allocation, and advanced data structures like graphs, vectors, priority queue, stack and various sorting algorithms.

### **Feature Description**

#### **Admin Portal**

**Admin Login:** Admin can login with the set admins in the program.

For example, you can use

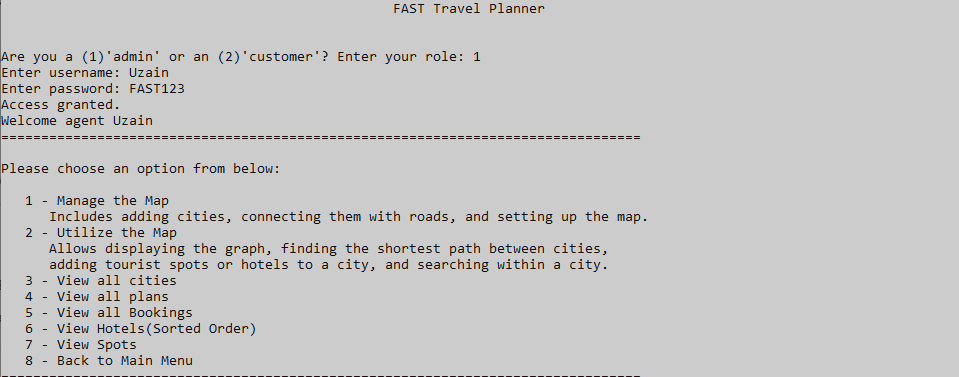
**username: Bilal**

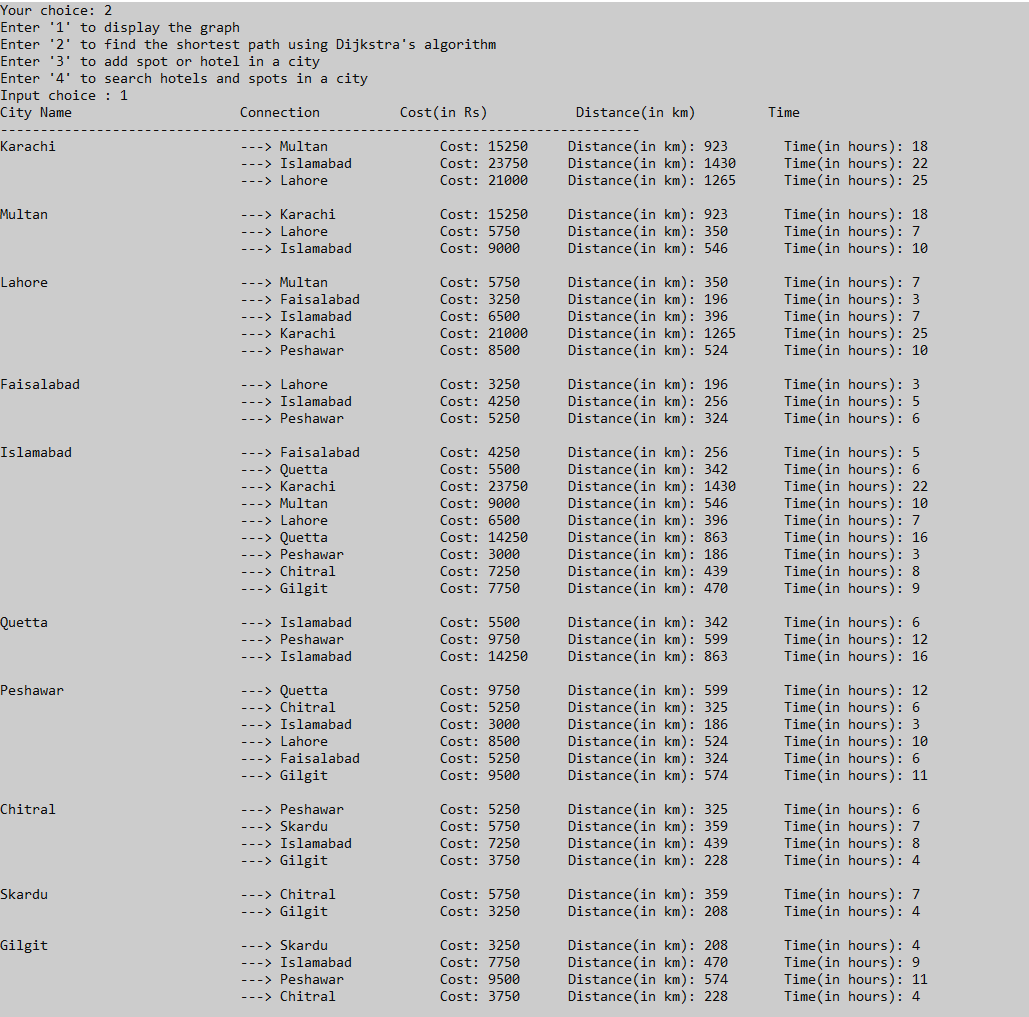
**password: FAST123**

**Manage Cities and Routes:** Admins can dynamically add cities and define routes (edges) between them. Each route is characterized by its distance and estimated travel time, enabling a graph representation of the travel network.

**Manage Hotels and Bookings:** The admin can add hotel information, including hotel names, distance from the city center, ratings, and cost. Additionally, the admin can view and manage booking records.

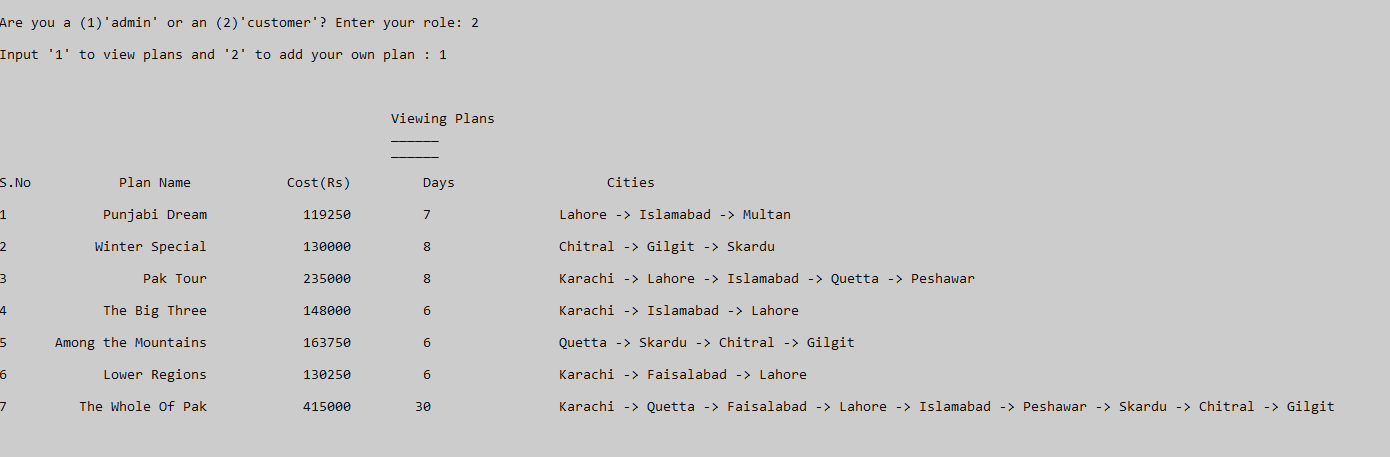
**View and Manage Bookings:** The admin can access the priority queue of bookings to view all planned trips in chronological order. This feature allows for better oversight of upcoming travel plans and resource allocation.

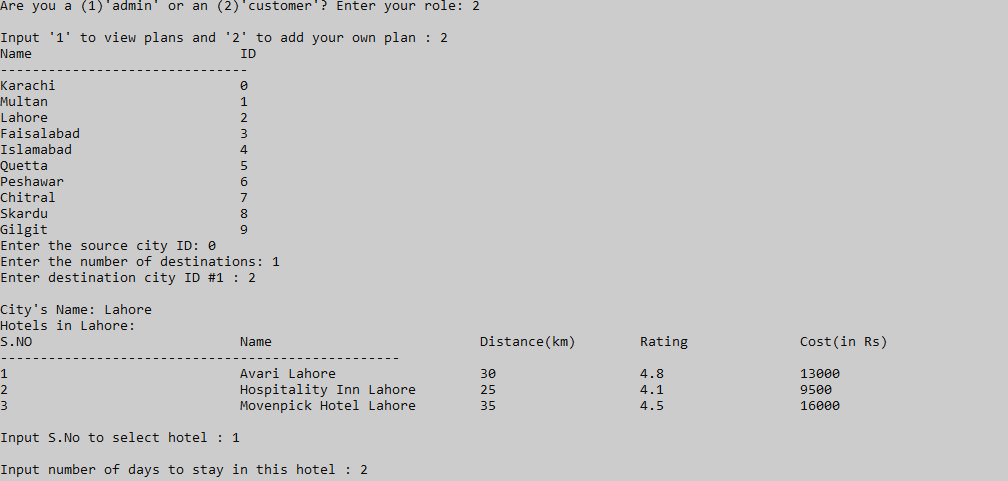


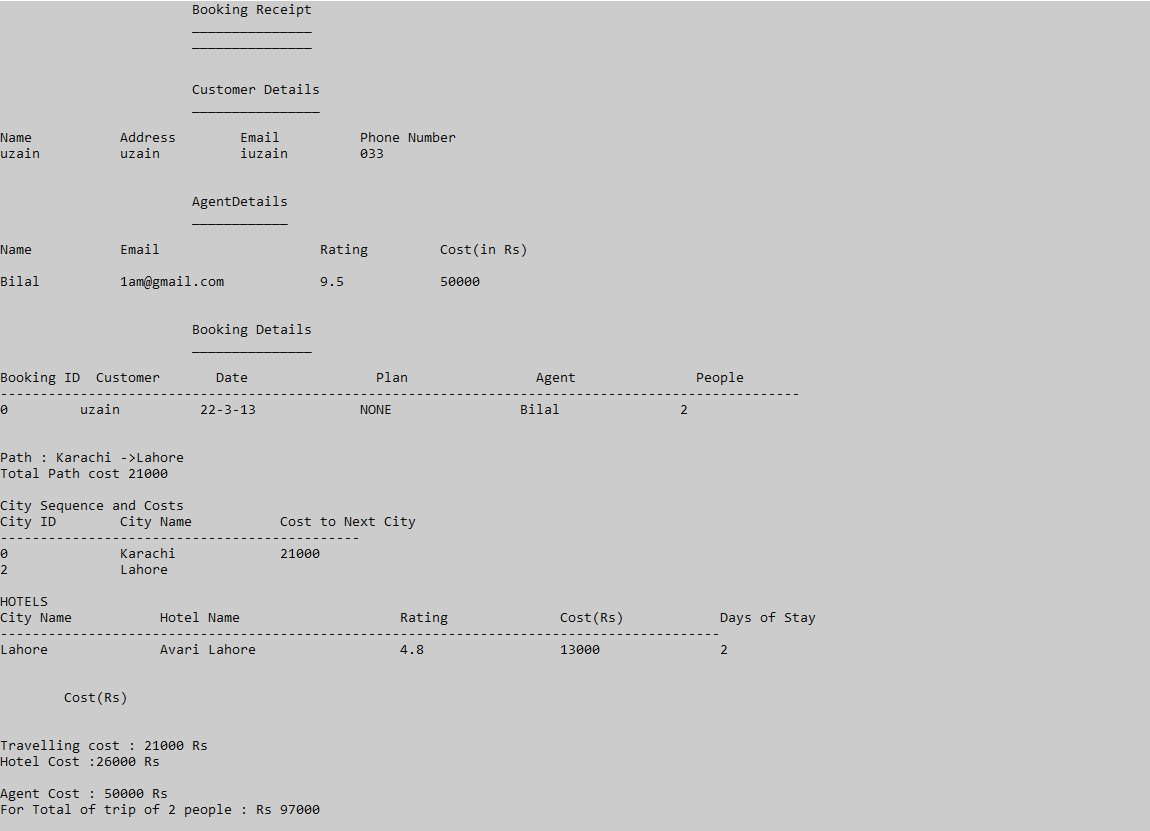


#### **Customer Portal**

* **Pre Plan or Custom Plan Options:** Customers have the option to select a pre-planned travel itinerary or create a custom plan. The pre-plan offers a list of destinations with predetermined routes, while the custom plan allows for greater flexibility.
* **Pre-Planned Trips:** For pre-planned trips, the itinerary, including hotels and travel routes, is predefined. Customers have the convenience of selecting their source city to start the journey and the number of people traveling. The cost of the trip is adjusted based on the number of people, source city, and agent ensuring that the planning process is straightforward yet flexible.
* **Custom Plans:** For custom plans, customers have the flexibility to select their preferred cities, hotels, and agents. The system intelligently checks for the availability of hotels and agents on the selected dates, allowing for a personalized travel experience. This option caters to those who prefer to tailor their travel plans according to specific preferences and needs.







**Cost Calculation:**

* **Path Cost Calculation:** The cost of traveling between cities is computed based on the distance. Assuming a vehicle can travel 15 km per liter of petrol, and the cost of petrol is 250 per liter, the cost is calculated as (Distance / 15) \* 250. This formula applies to the calculated shortest path between cities using Dijkstra's algorithm.
* **Hotel Cost Calculation:** The cost of hotels is calculated with an adjustment based on the number of people. For hotel costs, the application uses the formula ceil(peopleCount / 2) to determine the effective number of hotel units required. This approach is based on the assumption that two people can share a single hotel unit, hence rounding up to the nearest whole number to accommodate odd-numbered groups.
* **Agent Fees:** The agent fee is considered a fixed cost and is applied per booking, irrespective of the number of people. This cost is not multiplied or altered based on the number of travelers.

### **Technical Implementation: Path Cost Calculation**

The application employs a combination of Dijkstra's algorithm and permutation logic to calculate the most cost-effective path for a sequence of cities in a travel plan. This approach is especially crucial for custom travel plans where the customer can choose multiple cities to visit in any order.

#### **Algorithmic Workflow**

* **Permutation Generation:**
  + The function generate\_city\_permutations generates all possible permutations of the given set of destination cities.
  + Each permutation represents a possible sequence in which the cities can be visited.
* **Calculating Path Costs for Each Permutation (perm Function):**
  + For each city sequence permutation, the perm function is invoked.
  + The function starts by clearing previous paths and resetting total costs.
  + Dijkstra’s algorithm (dijkstra) is then used to calculate the shortest path cost from the source city to the first city in the permutation.
  + The process is repeated for each subsequent city pair in the permutation. The path cost from city v[i] to v[i + 1] is calculated and added to the total path cost.
  + If at any point, a path is not found (pathCost == INT\_MAX), the permutation is discarded (flagged).
* **Evaluating the Optimal Path:**
  + After calculating the total cost for a permutation, the system checks if this total cost is less than the current minimum.
  + If it is, this permutation is considered the most cost-effective path so far, and its details (path sequence and costs) are stored.
  + The process repeats for all permutations, ultimately finding the sequence of cities with the minimum total travel cost.
* **Storing and Retrieving the Best Path:**
  + The application stores the most cost-effective path sequence (best\_city\_paths) and its corresponding costs (best\_city\_path\_costs).
  + These details can be used for presenting the optimal travel plan to the customer.

#### **Implementation Considerations**

* **Efficiency:** This approach ensures that the customer is provided with the most cost-effective travel plan, considering all possible city sequences.
* **Flexibility:** It allows customers to explore multiple cities in an order that is both personally appealing and cost-effective.
* **User Experience:** By automating the calculation of the most efficient path, the application enhances user experience, making complex travel planning simple and accessible.

**Additional Components:**

* **Last Booking Stack:** A stack data structure is implemented to save the most recent booking. This allows for quick access and retrieval of the last booking's details for receipt generation.
* **Booking Priority Queue:** A priority queue is utilized to store all bookings in ascending order based on the booking date. This structure aids the admin in efficiently managing and viewing upcoming bookings.
* **Exception Handling for Input Validation:** The application includes robust exception handling mechanisms to manage incorrect or invalid user inputs. For example, if a user selects a city ID that does not exist (like selecting city ID 10 when there are only 9 cities), the system will prompt the user to re-enter a valid choice. This ensures a more reliable and user-friendly interface, preventing crashes or unexpected behavior due to incorrect inputs.
* **Basic Graphical User Interface (GUI) Enhancements:** Utilizing the iomanip library, the application features a basic but enhanced text-based user interface. This includes formatting tools to set widths and alignments, improving the readability and overall presentation of information on the screen. These GUI enhancements contribute to a more engaging and accessible user experience

.

* **Enhanced Travel Planning: Dynamic Agent Availability Display**

In our travel planner functionality, users can seamlessly input their desired travel date, and our system takes care of the rest. This feature not only checks the availability of agents but also intelligently displays only those agents who are free on the specified travel date.

**How it works:**

**1. *User-Defined Travel Date:*** Users enter the date for their travel plans, indicating when they require assistance or coordination from an agent.

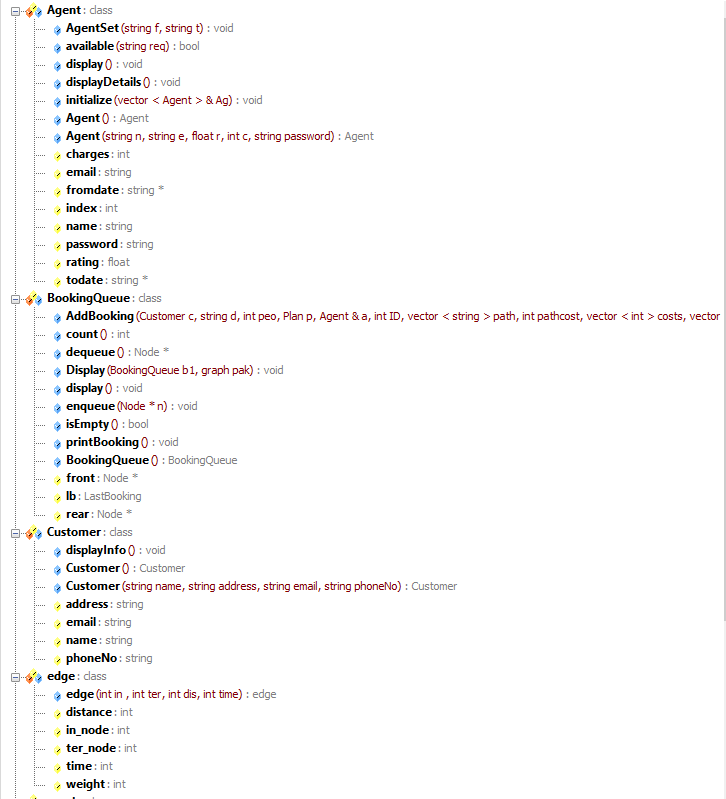
**2. *Real-time Availability Check****:* Our system conducts a real-time availability check for all agents to identify those who are free on the specified travel date using checkDate() function.

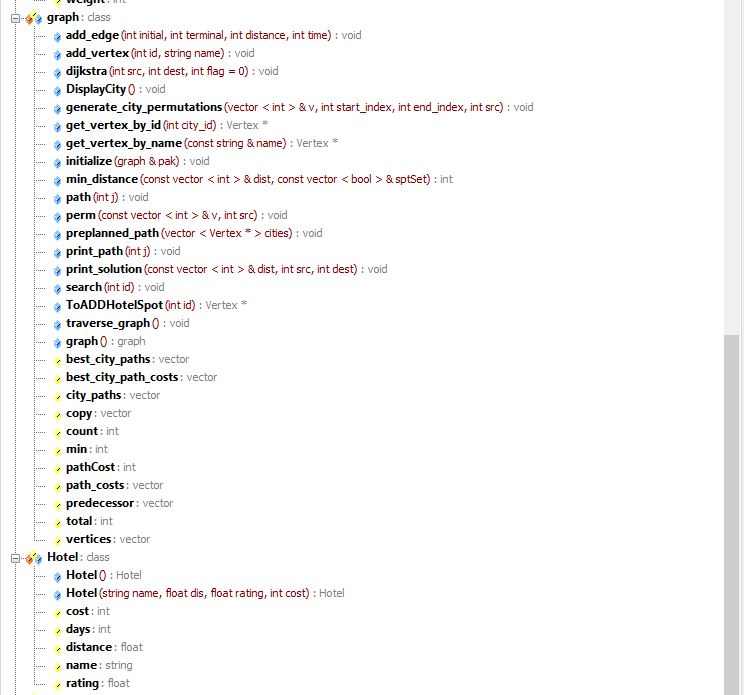
**3. *Filtered Agent Display****:* The user interface dynamically updates to display only the agents who are available on the selected travel date. This focused display ensures that users are presented with options that align with their specific travel needs.

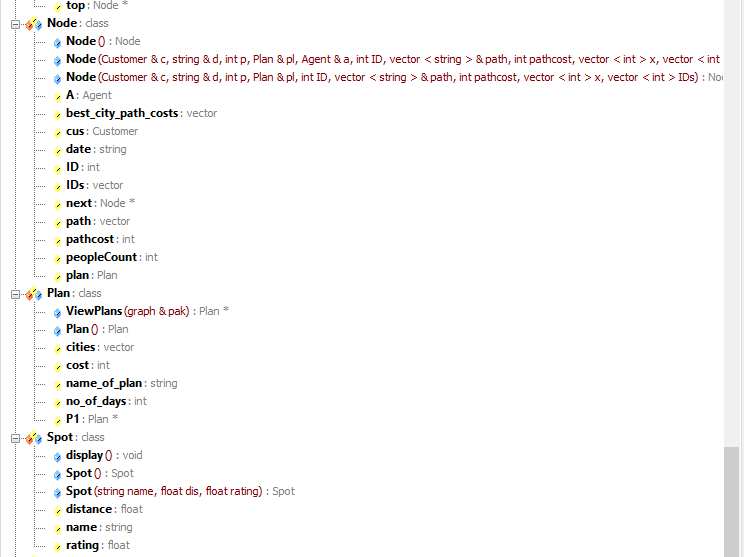
### **Code Walkthrough**

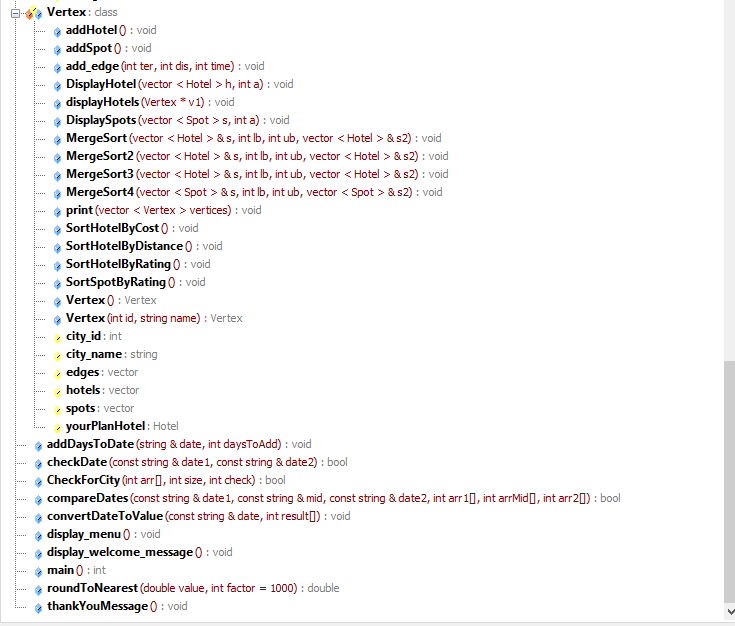
* **Classes Description:**
* edge: Represents a route with properties like initiating and terminating nodes, weight, distance, and time.
* Hotel: Stores hotel-related data including name, distance, rating, cost, and days of stay.
* Spot: Represents tourist spots with attributes like name, distance, and rating.
* Vertex: Represents a city in the graph, containing a list of edges (routes to other cities), hotels, and spots.
* Agent: Contains agent information including availability dates, name, email, rating, and charges.
* Customer: Stores customer details such as name, address, email, and phone number.
* graphs: Manages a network of vertices and edges, facilitating operations like adding, finding paths, and displaying graph data.
* LastBooking: Acts as a stack to manage booking records, providing functionality to add, and retrieve the most recent booking.
* BookingQueue: Implements a priority queue for managing bookings, allowing operations such as enqueueing, dequeueing, and displaying bookings based on a set priority.

**Project Outline:**

****

****

****

****

**Note:**

1. **CheckForCity: checks if user not selecting a city twice**

**2) Dates functions are interconnected and are being used to check the availability of agents on trip dates and comparing with agents’ current trip dates if any.**

**It is recommended to fully run the program and examine the code to get full idea of the project complexities and features. Happy Learning!**

**The code is attached in cpp and txt format.**