Data Structure & Object-Oriented Programming LAB

COMPLEX ENGINEERING ACTIVITY

CAB Booking System

<table>
<tr><td colspan="2" align="center">Name: Waleed Asif
Syed Anis Shah
Jibran khan
Adnan Ali
Roll no: 2420532,242073, 242039, 242108
Class: BEMTS-F-24-A
Submitted to: Engr. Sidrish Ehsan</td></tr>
<tr><td colspan="2" align="center">SESSION:
Fall 25</td></tr>
<tr><td align="center">DATE OF REPORT SUBMITTED:
25 December 2025</td><td align="center">GRADE/POINTS:</td></tr>
</table>

Table of Contents

CAB BOOKING SYSTEM

1. Introduction:

This project implements a Cab Management System using C++ and fundamental data structures. The system simulates how ride-hailing platforms manage drivers, customers, and city maps. It uses a graph-based city map to calculate shortest paths, a binary search tree (BST) to store drivers, a queue to manage pending ride requests, and a linked list to maintain ride history.

The system assigns the nearest available driver to a ride request using Dijkstra's shortest path algorithm, calculates fare based on distance, and supports ride completion, cancellation, and history tracking through a menu-driven interface.

The system enables users to book rides via a computer or smartphone, automatically assigning a nearby available driver and calculating the route and fare. In this project, the system models a city map of intersections (nodes) and roads (weighted edges), and handles user requests for rides, driver management, and ride history. The motivation is to simulate core functions of a ride-sharing service: adding/removing drivers, requesting and completing rides, and tracking history. This involves efficient data management and algorithms to ensure quick driver lookup, shortest-path computation, and orderly handling of pending requests. A GeeksforGeeks description highlights such a system's features – users enter pickup/drop-off locations, the system finds an available driver, shows estimated fare, and maintains trip history.

2. Project Working

The cab management system integrates multiple fundamental data structures and algorithms to simulate real-world ride-hailing operations. Its core functionalities are described below.

1. Driver Management using Binary Search Tree (BST)

Driver information is maintained using a **Binary Search Tree (BST)**, where each node is uniquely identified by a **driverID**. This structure enables efficient insertion, deletion, searching, and ordered traversal of driver records. Each driver node stores essential attributes, including driver ID, name, current location (currentNode), availability status, and total earnings.

The **CabSystem::addDriver** function inserts a new driver into the BST by invoking **DriverBST::insert**, while driver removal is handled through **DriverBST::remove**. To display all registered drivers, the system performs an **in-order traversal** of the BST, which outputs drivers in ascending order of driver ID. This ordered structure ensures organized and readable presentation of driver data.

2. Ride Requests and Ride History Management using Linked List

All ride-related records—whether ongoing, completed, or cancelled—are stored in a **singly linked list** implemented by the RideHistory class. The linked list structure allows dynamic growth of ride records and supports constant-time insertion.

Whenever a new ride is created, a RideRecord node containing the ride ID, driver ID, source node, destination node, travel distance, fare, and ride status is created and inserted at the head of the linked list using the **RideHistory::add** method. This approach ensures **O(1)** insertion complexity and maintains the ride history in reverse chronological order, with the most recent ride displayed first.

Users can view all ride records through the "Show Ride History" menu option, which traverses the linked list sequentially and displays each stored ride. The linked list is well-suited for this use case because the number of rides is not fixed and efficient insertion is required.

3. Shortest Path Computation using Graph and Dijkstra's Algorithm

The city map is modelled as a **weighted undirected graph**, implemented using an **adjacency matrix**. Nodes represent locations, and weighted edges represent roads with associated distances.

When a ride request is made from a source node to a destination node, the system computes the shortest distances from the source to all other nodes using **Dijkstra's algorithm**. These distances are used to determine the nearest available driver by comparing the shortest-path distance between the source node and each available driver's current location.

Once a driver is selected, the shortest-path distance from the source to the destination is retrieved and used to calculate the fare based on a fixed per-kilometer rate. Dijkstra's algorithm is appropriate for this scenario because all edge weights are non-negative and single-source shortest paths are required.

The Graph class encapsulates the adjacency matrix representation and provides a dijkstra() function to compute shortest-path distances efficiently.

4. Pending Ride Management using Queue (FIFO)

If no suitable driver is available or reachable at the time of a ride request, the request is temporarily stored in a **First-In-First-Out (FIFO) queue**. The system uses two parallel queues to store the source and destination nodes of pending ride requests.

When drivers become available, the "Process Pending Requests" option dequeues each request in the order it was received and attempts to assign a driver again. The FIFO queue ensures fairness by servicing earlier requests before newer ones and provides a simple and effective mechanism for managing delayed ride assignments.

5. Integrated System Operation

These components operate cohesively to manage the complete ride lifecycle. When a ride is requested, the system computes shortest paths using the graph, identifies available drivers through an in-order traversal of the BST, and selects the nearest driver. The ride is then logged in the linked list, and the driver's availability is updated accordingly.

Upon ride completion or cancellation, the ride status is updated in the linked list, and the corresponding driver's availability and earnings are adjusted in the BST. Through the combined use of a BST, linked list, graph, and queue, the system efficiently supports all major operations of a cab management platform.

3. Implementation Details

The C++ implementation of the Cab Management System is organized around several key classes and data structures, each responsible for a specific subsystem of the application.

1. SimpleQueue

The SimpleQueue class implements a **fixed-size, array-based queue** to manage pending ride requests. It provides standard queue operations such as enqueue(int), dequeue(), empty(), and full(), and follows **First-In-First-Out (FIFO)** behavior. Two parallel instances of this queue are used to store the source and destination nodes of pending ride requests. This design ensures that ride requests are processed in the order they are received.

2. RideHistory and RideRecord (Linked List)

The RideHistory class implements a **singly linked list** composed of RideRecord nodes. Each RideRecord stores complete ride information, including ride ID, driver ID, source node, destination node, travel distance, fare, ride status, and a pointer to the next record.

New ride records are added using the RideHistory::add method, which inserts each new node at the head of the list. This approach provides **constant-time insertion** and maintains the history in reverse chronological order. The printAll method traverses the list from the head to display all stored rides, while the findByID(int) method performs a linear search to locate a specific ride for completion or cancellation. The linked list structure is well-suited for this use case due to its dynamic size and efficient insertion characteristics.

3. Driver and DriverBST

The Driver class represents an individual driver as a node in a **Binary Search Tree (BST)**. Each driver node contains the driver ID, name, current location, availability status, total earnings, and pointers to left and right child nodes.

The DriverBST class manages all driver-related operations, including insertion, searching, deletion, and traversal. Drivers are inserted into the BST based on their unique driver ID, preserving the BST ordering property. The find(int) and remove(int) methods use recursive search logic to efficiently locate or delete drivers.

To display drivers, the inorderPrint() method performs an **in-order traversal**, outputting drivers sorted by driver ID. Additionally, the collectAvailable() method traverses the BST and collects pointers to available drivers into an array, which is later used to determine the nearest driver during ride assignment.

4. Graph and Shortest Path Computation

The Graph class represents the city road network using a **weighted adjacency matrix**. The matrix is initialized with a large sentinel value to indicate the absence of direct edges between nodes. Roads are added using the addEdge(u, v, w) method, which creates an undirected connection between two nodes with a specified weight.

The graph size is managed dynamically using the resize() function. The print() method outputs the adjacency matrix in a readable format, displaying all connected edges and their weights.

Shortest-path computation is handled by the dijkstra(int source, double dist[]) method. This method implements the classic **$O(V^2)$** version of Dijkstra's algorithm, which is suitable for the relatively small graph size used in this project. The algorithm computes the minimum distance from the source node to all other nodes and stores the results in the dist array.

5. CabSystem (Main Controller)

The CabSystem class serves as the central controller that integrates all system components. It aggregates instances of DriverBST, RideHistory, Graph, and two SimpleQueue objects, along with configuration parameters such as nextRideID and ratePerKm.

The class provides high-level methods corresponding to user menu operations, including:

initGraph(int nodes): Initializes the city map with a given number of nodes.

addRoad(int u, int v, double w): Adds a road between two nodes in the graph.

setRate(double r): Sets the fare rate per unit distance.

addDriver(int id, string name, int node): Inserts a new driver into the BST.

removeDriver(int id): Removes a driver from the BST.

updateDriverLocation(int id, int node): Updates a driver's current location.

printDrivers(): Displays all drivers using in-order BST traversal.

printMap(): Displays the city road network.

requestRide(int src, int dest): Validates input, computes shortest paths, selects the nearest available driver, updates driver status, calculates fare, records the ride in the linked list, and handles pending requests if necessary.

processPendingRequests(): Dequeues and reprocesses pending ride requests.

completeRide(int rideID): Marks a ride as completed and updates driver location and earnings.

cancelRide(int rideID): Cancels an ongoing ride and restores driver availability.

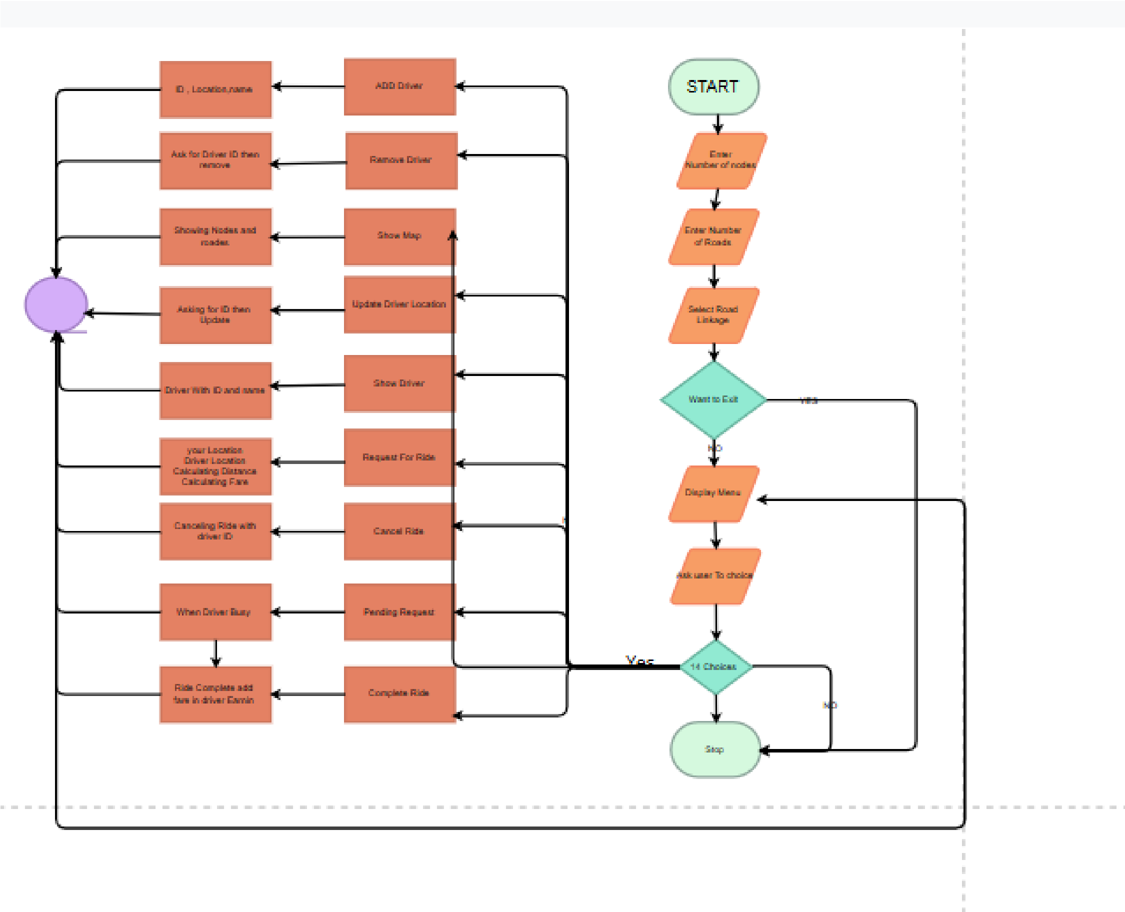
showRideHistory(): Displays all recorded rides.

6. Class Interaction and System Design

The system follows **object-oriented design principles**, where each class encapsulates a specific data structure or subsystem. The CabSystem class coordinates interactions between the graph (route computation), the driver BST (driver selection and management), the linked list (ride history), and the queue (pending rides).

For example, when a ride is requested, the CabSystem invokes the graph to compute shortest paths, queries the BST to identify available drivers, records the ride in the linked list, and updates the relevant driver's state. This modular design improves clarity, maintainability, and scalability of the system.

4. Flow Chart



5. Code:

```cpp
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

// ------------------- Simple Queue -------------------
class SimpleQueue
{
private:
    static const int MAXN = 100;
    int arr[MAXN];
    int frontIdx;
    int backIdx;
    int sz;

public:
    SimpleQueue()
    {
        frontIdx = 0;
        backIdx = -1;
        sz = 0;
    }

    bool empty()
    {
        return sz
    }

    bool full()
    {
        return sz == MAXN;
    }

    void enqueue(int value)
    {
        if (full())
        {
            cout << "Queue is full!" << endl;
```

[*] main.cpp

```cpp

    bool full()
    {
        return sz == MAXN;
    }

    void enqueue(int value)
    {
        if (full())
        {
            cout << "Queue is full!" << endl;
            return;
        }
        backIdx = (backIdx + 1) % MAXN;
        arr[backIdx] = value;
        sz++;
    }

    int dequeue()
    {
        if (empty())
        {
            cout << "Queue is empty!" << endl;
            return -1;
        }
        int value = arr[frontIdx];
        frontIdx = (frontIdx + 1) % MAXN;
        sz--;
        return value;
    }

    int front()
    {
        if (empty())
        {
            cout << "Queue is empty!" << endl;
            return -1;
        }
        return arr[frontIdx];
    }

    int size()
    {
        return sz;
    }
};

// ------------------- Ride History -------------------
class RideRecord
{
public:
    int rideID;
    int driverID;
    int srcNode;
    int destNode;
    double distance;
    double fare;
    string status;
    RideRecord* next;

```

[*] main.cpp

```cpp
    string status;
    RideRecord* next;

    RideRecord(int r, int d, int s, int de, double dist, double f, string st)
    {
        rideID = r;
        driverID = d;
        srcNode = s;
        destNode = de;
        distance = dist;
        fare = f;
        status = st;
        next = nullptr;
    }
};

class RideHistory
{
private:
    RideRecord* head;

public:
    RideHistory()
    {
        head = nullptr;
    }

    void add(int rideID, int driverID, int src, int dest, double dist, double fare, string status)
    {
        RideRecord* node = new RideRecord(rideID, driverID, src, dest, dist, fare, status);
        node->next = head;
        head = node;
    }

    void printAll()
    {
        cout << "Ride History (most recent first):" << endl;
        RideRecord* cur = head;
        if (cur == NULL)
        {
            cout << "  No rides yet." << endl;
            return;
        }
        while (cur != NULL)
        {
            cout << "  Ride " << cur->rideID
                << " | Driver " << cur->driverID
                << " | " << cur->srcNode << " -> " << cur->destNode
                << " | dist: " << cur->distance
                << " | fare: " << cur->fare
                << " | " << cur->status << endl;
            cur = cur->next;
        }
    }

    RideRecord* findByID(int rideID)
    {
        RideRecord* cur = head;
        while (cur)
        {
```

```cpp
        RideRecord* cur = head;
        while (cur)
        {
            if (cur->rideID == rideID)
                return cur;
            cur = cur->next;
        }
        return nullptr;
    }
};

// ------------------- Driver BST -------------------
class Driver
{
public:
    int driverID;
    string name;
    int currentNode;
    bool available;
    double earnings;
    Driver* left;
    Driver* right;

    Driver(int id = 0, string n = "", int node = 0, bool avail = true)
    {
        driverID = id;
        name = n;
        currentNode = node;
        available = avail;
        earnings = 0;
        left = nullptr;
        right = nullptr;
    }
};

class DriverBST
{
private:
    Driver* root;

    Driver* insertRec(Driver* node, Driver* toIns)
    {
        if (!node)
            return toIns;
        if (toIns->driverID < node->driverID)
            node->left = insertRec(node->left, toIns);
        else if (toIns->driverID > node->driverID)
            node->right = insertRec(node->right, toIns);
        else
        {
            node->name = toIns->name;
            node->currentNode = toIns->currentNode;
            node->available = toIns->available;
            node->earnings = toIns->earnings;
        }
        return node;
    }

    Driver* findRec(Driver* node, int id)
    {
```

[*] main.cpp

```cpp

    Driver* findRec(Driver* node, int id)
    {
        if (!node)
            return nullptr;
        if (id == node->driverID)
            return node;
        if (id < node->driverID)
            return findRec(node->left, id);
        return findRec(node->right, id);
    }

    Driver* deleteRec(Driver* node, int id)
    {
        if (!node)
            return nullptr;
        if (id < node->driverID)
            node->left = deleteRec(node->left, id);
        else if (id > node->driverID)
            node->right = deleteRec(node->right, id);
        else
        {
            if (!node->left)
            {
                Driver* r = node->right;
                delete node;
                return r;
            }
            if (!node->right)
            {
                Driver* l = node->left;
                delete node;
                return l;
            }

            Driver* succParent = node;
            Driver* succ = node->right;
            while (succ->left)
            {
                succParent = succ;
                succ = succ->left;
            }

            node->driverID = succ->driverID;
            node->name = succ->name;
            node->currentNode = succ->currentNode;
            node->available = succ->available;
            node->earnings = succ->earnings;

            if (succParent->left == succ)
                succParent->left = succ->right;
            else
                succParent->right = succ->right;

            delete succ;
        }
        return node;
    }

    void inorderRec(Driver* node)
```


[*] main.cpp

```cpp
    }

    void inorderRec(Driver* node)
```


```cpp
    {
        if (!node)
            return;
        inorderRec(node->left);
        cout << "  ID: " << node->driverID
             << " | Name: " << node->name
             << " | Node: " << node->currentNode
             << " | " << (node->available ? "Available" : "Busy")
             << " | Earnings: " << node->earnings << endl;
        inorderRec(node->right);
    }

    void collectAvailRec(Driver* node, Driver** arr, int& idx)
    {
        if (!node)
            return;
        collectAvailRec(node->left, arr, idx);
        if (node->available)
            arr[idx++] = node;
        collectAvailRec(node->right, arr, idx);
    }

public:
    DriverBST()
    {
        root = nullptr;
    }

    void insert(int id, string name, int node, bool avail = true)
    {
        Driver* d = new Driver(id, name, node, avail);
        root = insertRec(root, d);
    }

    Driver* find(int id)
    {
        return findRec(root, id);
    }

    void remove(int id)
    {
        root = deleteRec(root, id);
    }

    void inorderPrint()
    {
        if (!root)
        {
            cout << "No drivers." << endl;
            return;
        }
        cout << "Drivers (in-order by ID):" << endl;
        inorderRec(root);
    }

    void collectAvailable(Driver** arr, int& count)
    {
```

[*] main.cpp

```cpp

    void collectAvailable(Driver** arr, int& count)
    {
        count = 0;
        collectAvailRec(root, arr, count);
    }
};

// ------------------- Graph & Dijkstra -------------------
class Graph
{
private:
    int n;
    double adj[100][100];

public:
    Graph(int nodes = 0)
    {
        n = nodes;
        for (int i = 0; i < 100; i++)
            for (int j = 0; j < 100; j++)
                adj[i][j] = 1e18;
    }

    void resize(int nodes)
    {
        n = nodes;
        for (int i = 0; i < 100; i++)
            for (int j = 0; j < 100; j++)
                adj[i][j] = 1e18;
    }

    void addEdge(int u, int v, double w)
    {
        if (u < 1 || v < 1 || u > n || v > n)
        {
            cout << "Invalid node index" << endl;
            return;
        }
        adj[u][v] = w;
        adj[v][u] = w;
    }

    int size()
    {
        return n;
    }

    void dijkstra(int source, double dist[100])
    {
        bool visited[100] = { false };
        for (int i = 1; i <= n; i++)
            dist[i] = 1e18;
        dist[source] = 0;

        for (int count = 1; count <= n; count++)
        {
            int u = -1;
            for (int i = 1; i <= n; i++)
                if (!visited[i] && (u == -1 || dist[i] < dist[u]))
```

[*] main.cpp

```cpp
            int u = -1;
            for (int i = 1; i <= n; i++)
                if (!visited[i] && (u == -1 || dist[i] < dist[u]))
                    u = i;

            if (dist[u] == 1e18)
                break;
            visited[u] = true;

            for (int v = 1; v <= n; v++)
                if (adj[u][v] < 1e17)
                    if (dist[v] > dist[u] + adj[u][v])
                        dist[v] = dist[u] + adj[u][v];
        }
    }

    void print()
    {
        cout << "City map adjacency matrix:" << endl;
        for (int i = 1; i <= n; i++)
        {
            cout << i << ": ";
            for (int j = 1; j <= n; j++)
                if (adj[i][j] < 1e17)
                    cout << j << "(" << adj[i][j] << ") ";
            cout << endl;
        }
    }
};

// ------------------- Cab Management -------------------
class CabSystem
{
private:
    Graph graph;
    DriverBST drivers;
    RideHistory rideHistory;
    int nextRideID;
    double ratePerKm;
    SimpleQueue pendingSrc;
    SimpleQueue pendingDest;

public:
    CabSystem()
    {
        graph.resize(0);
        nextRideID = 1;
        ratePerKm = 10.0;
    }

    void initGraph(int nodes)
    {
        graph.resize(nodes);
    }

    void addRoad(int u, int v, double w)
    {
        graph.addEdge(u, v, w);
    }

```

ources Compile Log Debug Find Results Console

```cpp

    void addRoad(int u, int v, double w)
    {
        graph.addEdge(u, v, w);
    }

    void setRate(double r)
    {
        ratePerKm = r;
    }

    void addDriver(int id, string name, int node)
    {
        drivers.insert(id, name, node, true);
    }

    void removeDriver(int id)
    {
        drivers.remove(id);
    }

    void updateDriverLocation(int id, int node)
    {
        Driver* d = drivers.find(id);
        if (!d)
        {
            cout << "Driver not found." << endl;
            return;
        }
        d->currentNode = node;
    }

    void printDrivers()
    {
        drivers.inorderPrint();
    }

    void printMap()
    {
        graph.print();
    }

    void requestRide(int src, int dest)
    {
        if (src < 1 || dest < 1 || src > graph.size() || dest > graph.size())
        {
            cout << "Invalid source/destination nodes." << endl;
            return;
        }

        cout << "Processing ride from " << src << " to " << dest << "..." << endl;
        double dist[100];
        graph.dijkstra(src, dist);

        Driver* availArr[100];
        int availCount;
        drivers.collectAvailable(availArr, availCount);

        if (availCount == 0)
        {
```

```cpp

        if (availCount == 0)
        {
            cout << "No available drivers. Added to pending queue." << endl;
            pendingSrc.enqueue(src);
            pendingDest.enqueue(dest);
            return;
        }

        Driver* chosen = nullptr;
        double minDist = 1e18;

        for (int i = 0; i < availCount; i++)
            if (dist[availArr[i]->currentNode] < minDist)
            {
                minDist = dist[availArr[i]->currentNode];
                chosen = availArr[i];
            }

        if (!chosen || minDist > 1e17)
        {
            cout << "No reachable available drivers. Added to pending queue." << endl;
            pendingSrc.enqueue(src);
            pendingDest.enqueue(dest);
            return;
        }

        chosen->available = false;
        graph.dijkstra(src, dist);
        double tripDist = dist[dest];

        if (tripDist > 1e17)
        {
            cout << "Destination unreachable. Cancelling." << endl;
            chosen->available = true;
            return;
        }

        double fare = tripDist * ratePerKm;
        int rideID = nextRideID++;

        // Ride starts as ongoing
        rideHistory.add(rideID, chosen->driverID, src, dest, tripDist, fare, "Ongoing");

        cout << "Assigned Driver " << chosen->driverID << " (" << chosen->name << ")." << endl;
        cout << "Estimated distance: " << tripDist << " km, fare: " << fare << endl;
    }

    void completeRide(int rideID)
    {
        RideRecord* ride = rideHistory.findByID(rideID);
        if (!ride)
        {
            cout << "Ride not found." << endl;
            return;
        }
        if (ride->status != "Ongoing")
        {
            cout << "Ride not ongoing. Current status: " << ride->status << endl;
            return;
```

[*] main.cpp

```cpp
        }

        ride->status = "Completed";

        Driver* d = drivers.find(ride->driverID);
        if (d)
        {
            d->currentNode = ride->destNode;
            d->earnings += ride->fare;
            d->available = true;
        }

        cout << "Ride " << rideID << " completed." << endl;
    }

    void cancelRide(int rideID)
    {
        RideRecord* ride = rideHistory.findByID(rideID);
        if (!ride)
        {
            cout << "Ride ID " << rideID << " not found." << endl;
            return;
        }
        if (ride->status != "Ongoing")
        {
            cout << "Cannot cancel ride. Status: " << ride->status << endl;
            return;
        }

        ride->status = "Cancelled";

        Driver* d = drivers.find(ride->driverID);
        if (d)
            d->available = true;

        cout << "Ride " << rideID << " has been cancelled." << endl;
    }

    void processPendingRequests()
    {
        int qsize = pendingSrc.size();
        for (int i = 0; i < qsize; i++)
        {
            int src = pendingSrc.dequeue();
            int dest = pendingDest.dequeue();
            cout << "Trying pending request " << src << " -> " << dest << "..." << endl;
            requestRide(src, dest);
        }
    }

    void showRideHistory()
    {
        rideHistory.printAll();
    }
};

// ------------------- Menu -------------------


// ------------------- Main -------------------
```

[*] main.cpp

```cpp


// ------------------- Main -------------------
int main() {



        CabSystem sys;
        int nodes;
        cout << "Initialize city map (number of nodes): ";
        cin >> nodes;
        sys.initGraph(nodes);

        int m;
        cout << "Create roads (enter m edges): ";
        cin >> m;

        for (int i = 0; i < m; i++)
        {
            int u, v;
            double w;
            cout << "Edge " << i + 1 << " u v w: ";
            cin >> u >> v >> w;
            sys.addRoad(u, v, w);
        }

        sys.setRate(10.0);

        while (true)
        {
            cout << "\n--- Cab Management Menu ---" << endl;
            cout << "1. Add Driver" << endl;
            cout << "2. Remove Driver" << endl;
            cout << "3. Update Driver Location" << endl;
            cout << "4. Show Drivers" << endl;
            cout << "5. Show Map" << endl;
            cout << "6. Request Ride" << endl;
            cout << "7. Process Pending Requests" << endl;
            cout << "8. Show Ride History" << endl;
            cout << "9. Cancel Ride" << endl;
            cout << "10. Complete Ride" << endl;
            cout << "11. Exit" << endl;
            cout << "Choose: ";

            int ch;
            cin >> ch;

            if (ch == 1)
            {
                int id, node;
                string name;
                cout << "Driver ID: ";
                cin >> id;
                cout << "Name: ";
                cin.ignore();
                getline(cin, name);
                cout << "Node: ";
                cin >> node;
                sys.addDriver(id, name, node);
            }
```

```cpp
                cout << "Name: ";
                cin.ignore();
                getline(cin, name);
                cout << "Node: ";
                cin >> node;
                sys.addDriver(id, name, node);
            }
            else if (ch == 2)
            {
                int id;
                cout << "Driver ID to remove: ";
                cin >> id;
                sys.removeDriver(id);
            }
            else if (ch == 3)
            {
                int id, node;
                cout << "Driver ID: ";
                cin >> id;
                cout << "New node: ";
                cin >> node;
                sys.updateDriverLocation(id, node);
            }
            else if (ch == 4)
                sys.printDrivers();
            else if (ch == 5)
                sys.printMap();
            else if (ch == 6)
            {
                int s, d;
                cout << "Source node: ";
                cin >> s;
                cout << "Destination node: ";
                cin >> d;
                sys.requestRide(s, d);
            }
            else if (ch == 7)
                sys.processPendingRequests();
            else if (ch == 8)
                sys.showRideHistory();
            else if (ch == 9)
            {
                int rideID;
                cout << "Ride ID to cancel: ";
                cin >> rideID;
                sys.cancelRide(rideID);
            }
            else if (ch == 10)
            {
                int rideID;
                cout << "Ride ID to complete: ";
                cin >> rideID;
                sys.completeRide(rideID);
            }
            else
                break;
        }
    }
```

6. Output:

```
C:\Users\hp\Desktop\3

Initialize city map (number of nodes): 4
Create roads (enter m edges): 3
Edge 1 u v w: 1 2 5
Edge 2 u v w: 2 3 2
Edge 3 u v w: 3 4 3

--- Cab Management Menu ---
1. Add Driver
2. Remove Driver
3. Update Driver Location
4. Show Drivers
5. Show Map
6. Request Ride
7. Process Pending Requests
8. Show Ride History
9. Cancel Ride
10. Complete Ride
11. Exit
Choose: 1
Driver ID: 123
Name: Anis
Node: 1

--- Cab Management Menu ---
1. Add Driver
2. Remove Driver
3. Update Driver Location
4. Show Drivers
5. Show Map
6. Request Ride
7. Process Pending Requests
8. Show Ride History
9. Cancel Ride
10. Complete Ride
11. Exit
Choose: 1
Driver ID: 232
Name: Waleed
Node: 3

--- Cab Management Menu ---
1. Add Driver
```

C:\Users\hp\Desktop\3

```
--- Cab Management Menu ---
1. Add Driver
2. Remove Driver
3. Update Driver Location
4. Show Drivers
5. Show Map
6. Request Ride
7. Process Pending Requests
8. Show Ride History
9. Cancel Ride
10. Complete Ride
11. Exit
Choose: 2
Driver ID to remove: 123

--- Cab Management Menu ---
1. Add Driver
2. Remove Driver
3. Update Driver Location
4. Show Drivers
5. Show Map
6. Request Ride
7. Process Pending Requests
8. Show Ride History
9. Cancel Ride
10. Complete Ride
11. Exit
Choose: 3
Driver ID: 232
New node: 1

--- Cab Management Menu ---
1. Add Driver
2. Remove Driver
3. Update Driver Location
4. Show Drivers
5. Show Map
6. Request Ride
7. Process Pending Requests
8. Show Ride History
9. Cancel Ride
```

C:\Users\hp\Desktop\3

```
8. Show Ride History
9. Cancel Ride
10. Complete Ride
11. Exit
Choose: 4
Drivers (in-order by ID):
  ID: 232 | Name: Waleed | Node: 1 | Available | Earnings: 0

--- Cab Management Menu ---
1. Add Driver
2. Remove Driver
3. Update Driver Location
4. Show Drivers
5. Show Map
6. Request Ride
7. Process Pending Requests
8. Show Ride History
9. Cancel Ride
10. Complete Ride
11. Exit
Choose: 5
City map adjacency matrix:
1: 2(5)
2: 1(5) 3(2)
3: 2(2) 4(3)
4: 3(3)

--- Cab Management Menu ---
1. Add Driver
2. Remove Driver
3. Update Driver Location
4. Show Drivers
5. Show Map
6. Request Ride
7. Process Pending Requests
8. Show Ride History
9. Cancel Ride
10. Complete Ride
11. Exit
Choose: 6
Source node: 1
Destination node: 3
```

C:\Users\hp\Desktop\3

```
Destination node: 3
Processing ride from 1 to 3...
Assigned Driver 232 (Waleed).
Estimated distance: 7 km, fare: 70

--- Cab Management Menu ---
1. Add Driver
2. Remove Driver
3. Update Driver Location
4. Show Drivers
5. Show Map
6. Request Ride
7. Process Pending Requests
8. Show Ride History
9. Cancel Ride
10. Complete Ride
11. Exit
Choose: 7

--- Cab Management Menu ---
1. Add Driver
2. Remove Driver
3. Update Driver Location
4. Show Drivers
5. Show Map
6. Request Ride
7. Process Pending Requests
8. Show Ride History
9. Cancel Ride
10. Complete Ride
11. Exit
Choose: 8
Ride History (most recent first):
  Ride 1 | Driver 232 | 1 -> 3 | dist: 7 | fare: 70 | Ongoing

--- Cab Management Menu ---
1. Add Driver
2. Remove Driver
3. Update Driver Location
4. Show Drivers
5. Show Map
6. Request Ride
```

C:\Users\hp\Desktop\3

```
7. Process Pending Requests
8. Show Ride History
9. Cancel Ride
10. Complete Ride
11. Exit
Choose: 8
Ride History (most recent first):
  Ride 1 | Driver 232 | 1 -> 3 | dist: 7 | fare: 70 | Ongoing

--- Cab Management Menu ---
1. Add Driver
2. Remove Driver
3. Update Driver Location
4. Show Drivers
5. Show Map
6. Request Ride
7. Process Pending Requests
8. Show Ride History
9. Cancel Ride
10. Complete Ride
11. Exit
Choose: 10
Ride ID to complete: 1
Ride 1 completed.

--- Cab Management Menu ---
1. Add Driver
2. Remove Driver
3. Update Driver Location
4. Show Drivers
5. Show Map
6. Request Ride
7. Process Pending Requests
8. Show Ride History
9. Cancel Ride
10. Complete Ride
11. Exit
Choose: 11

--------------------------------
Process exited after 178.6 seconds with return value 0
Press any key to continue . . .
```

7. Learning Outcomes

After working on this project, I gain experience in:

- **Object-Oriented Design:** Defining classes (CabSystem, DriverBST, RideHistory, Graph) and their interactions. Designing methods for specific operations (adding drivers, handling rides).
- **Data Structures:** Implementing a BST for driver management, a linked list for ride history, a queue for pending requests, and a graph (adjacency matrix) for the city map. Understanding when and why each structure is appropriate (e.g. BST for sorted access linked list for dynamic records, queue for FIFO order).
- **Algorithms:** Applying Dijkstra's algorithm to find shortest paths in the graph. This reinforces knowledge of graph algorithms and their implementation in C++.
- **C++ Programming:** Using pointers, dynamic memory, and class encapsulation. (The code builds custom data structures rather than relying on STL, reinforcing how these structures work under the hood.)
- **Problem Solving:** Integrating multiple components to fulfill complex requirements (driver matching, request queuing, route finding). Developing logic to handle edge cases (no available drivers, unreachable destinations, ride cancellation).
- **Debugging and Testing:** Running sample scenarios, checking menu driven I/O, and verifying outputs (as shown in the example).

Overall, the Cab Booking System project provides practical experience with key computer science concepts (data structures, algorithms) and C++ programming techniques in a realistic application setting.

8. Conclusion

The Cab Booking System project successfully demonstrates the integration of object-oriented programming and fundamental data structures in solving a real-world problem. By modeling a city map using graphs, managing drivers with a binary search tree, handling ride requests through queues, and recording trip details via linked lists, the project reflects a comprehensive understanding of data structures and algorithmic logic in C++.

Through this project, we not only built a functional ride-booking simulation but also gained practical experience in designing interconnected systems, implementing efficient algorithms like Dijkstra's for shortest path calculation, and applying concepts of encapsulation, modularity, and dynamic data handling. This system can serve as a foundational framework for more advanced transport or logistics applications, potentially expandable to include file storage, real-time driver tracking, or user accounts.

Overall, the project highlights the power of C++ for structured problem-solving and lays a strong foundation for further exploration into complex system development.