Name: Nawal Nadim

Unit Name: Data Structures and Algorithms

Program Leader: Ms Benita

Due Date: 31 March 2025

Assignment Name: ADTs for Autonomous Vehicle Management System

**Introduction**:

This assignment was about building an Autonomous Vehicle Management System (AVMS) using your data structures and algorithms. The goal was to manage a fleet of self-driving vehicles by keeping track of their status and location, creating versions of key components like graphs for the road network, hash tables for storing vehicle details, and sorting algorithms to organize vehicles based on different criteria.

**Purpose**:

The purpose is to show how you can design and implement data structures and algorithms such as graphs, hash tables, and sorting algorithms.

**Objectives**:

**Graph for Road Network:**  A Graph where each node represents a location and each edge represents a road with details like distance or travel time.

**Vehicle Hash Table:** Creates a hash table to store vehicle information using a custom hash function and collision resolution.

**Vehicle Class:** Develop a Vehicle class that includes attributes like vehicle ID, current location, destination, distance to destination, and battery level, with methods to get and set these values.

**Sorting and Recommendations:**  Uses heapsort to sort vehicles by distance to destination in ascending order and find the closest vehicle.

Use quicksort to sort vehicles by battery level in descending order and identify the one with the highest battery.

**Interactive Menu:** Implements a user-friendly menu to access all the features and ensure the program handles errors gracefully...

**Graph Class Overview**

**Purpose:**The Graph class is designed to represent a road network for the Autonomous Vehicle Management System. Each location (or vertex) is a key in a map, and its value is a list of roads (edges) connecting it to other locations. The class supports adding locations, connecting them with roads, retrieving neighbors, displaying the whole network, and checking if there’s a path between two locations using Breadth-First Search (BFS).

### **Key Components and Methods**

**Data Structures Used:**

* **Map (HashMap):**Stores the graph where each key is a location (a String) and each value is a list of Edge objects. This helps in quickly finding all connected roads from any given location.
* **List (ArrayList):**Used to store a list of edges for each location. Each edge represents a connection (or road) to another location.
* **Queue (LinkedList):**Used in the isPath method to perform BFS, which helps in exploring the network level by level.
* **Set (HashSet):**Keeps track of visited locations during the BFS to prevent processing the same location multiple times

**1.Inner Class: Edge**

**Purpose:**Represents a road between two locations.

**Key Attributes:**

* destination: The neighboring location.
* distance: The distance or travel time to that neighboring location.

1. **toString Method:**it provides a readable format to display the edge (e.g., "LocationB (10 km)").
2. **AddVertex(String location):**

**Purpose:**the purpose is it adds a new location to the graph.

**Implementation:**

It checks if the location name is valid (not null or empty).

And adds the location to the adjacencyList if it isn’t already present.

**Error Handling:** it throws and catches an exception if the location is invalid.

**3.addEdge(String source, String destination, double distance):**

**Purpose:**the purpose is it connects two locations by adding a road between them.

**Implementation:**

It checks if the source and destination exist in the graph and adds an edge from the source to the destination and, assuming the graph is undirected, an edge from the destination to the source

**Error Handling:** it validates the input and prints an error message if the source or destination does not exist or if the distance is not positive.

**4) getNeighbors(String location):**

**Purpose:**the purpose is it retrieves a list of roads (edges) that connect from the specified location.

**Implementation:**

It validates the location name and returns the list of edges (neighbors) for the given location.

**Error Handling:**If the location is invalid or does not exist, it prints an error message and returns an empty list.

**5) displayGraph():**

**Purpose:**the purpose is it prints the entire road network

**Implementation:**

It iterates through each location in the adjacencyList and for each location, it prints the connected neighbors and their road details.

**User Interface:**it heelps visualize the road network, showing which locations are connected.

**6) isPath(String source, String destination):**

**Purpose:**the purpose is it checks if there is a path between two locations using Breadth-First Search (BFS).

**Implementation:**

It Validates the input locations, If the source and destination are the same, it returns true immediately, Uses a Queue to explore the graph level by level, Uses a Set to keep track of visited locations, Returns true if the destination is found during the search; otherwise, returns false.

**7. Interactive Menu (main method):**

**Purpose:**the purpose is it Provides a simple console-based user interface for testing and interacting with the graph.

**Features:** Adding locations and adding roads , displaying neighbors for a given locations, displaying the entire road network, checking if a path exists between two locations.

**Input Handling:**it includes error handling to manage invalid inputs, ensuring the system remains robust.

### **VehicleHashTable Class**

**Purpose:**This class implements a hash table to store and manage vehicles and it uses an array of linked lists (separate chaining) to handle collisions when multiple vehicles hash to the same index.

**Key Components and Methods:**

**Data Structure:**

**Array of LinkedLists:**The hash table is implemented as an array (table) with a fixed size (10). Each element of the array is a linked list that stores vehicles with the same hash index.

**Hash Function (hashFunction):**

**Purpose:**the purpose is it computes an index based on the vehicle ID.

**Implementation**It uses the built-in hashCode() method of the String class, takes the absolute value, and then calculates the remainder when divided by the table size. This ensures that the index is within the bounds of the array.

**Insert (insert):**

**Purpose:**the purpose is it adds a new vehicle to the hash table.

**Implementation**

First, it computes the hash index for the vehicle using its ID. Then, it checks the linked list at that index to see if a vehicle with the same ID already exists. If it does, it prints an error message, if the vehicle is unique, it adds the vehicle to the linked list and prints a confirmation message (for debugging).

**Search (search):**

**Purpose:**the purpose is it Finds a vehicle by its ID.

**Implementation**

It calculates the hash index for the given ID, it then iterates through the linked list at that index, if it finds a matching vehicle, it returns that vehicle; otherwise, it returns null.

**Delete (delete):**

**Purpose:**the purpose is it removes a vehicle from the hash table by its ID.

**Implementation:** It finds the correct linked list using the hash function, it searches the list for the vehicle with the specified ID and, if found, removes it and prints a success message, if the vehicle is not found, it prints an error message and returns false.

**Display All (displayAll):**

**Purpose:**the purpose is it prints all vehicles stored in the hash table.

**Implementation:** It iterates through the array. For each index, it prints the index number and then the list of vehicles stored in that linked list. If the list is empty, it prints "Empty."

**Interactive Menu (main method):**

**Purpose:**the purpose is it provides a user-friendly interface for testing the hash table.

**Implementation:**The menu offers options to add a vehicle, search for a vehicle, delete a vehicle, or display all vehicles, it handles invalid input by checking if the user entered a valid number, the program continues to run until the user chooses to exit.

## **VehicleManagementSystem Class**

**Purpose:**The VehicleManagementSystem class manages a fleet of vehicles. It allows the user to add, edit, display, and search for vehicles using a simple console-based interactive menu.

**Key Methods and Implementation Details:**

**Data Structure: ArrayList**

Vehicles are stored in an ArrayList<Vehicle>, which allows dynamic resizing and easy iteration.

**Helper Methods:**

**getValidInt(String prompt, Scanner scanner):**

**Purpose:**the purpose is it ensures the user enters a valid integer.

**Implementation:**It uses a loop with exception handling (InputMismatchException) to repeatedly prompt until valid input is provided.

**getValidDouble(String prompt, Scanner scanner):**

**Purpose:**the purpose is it ensures the user enters a valid non-negative double value.

**Implementation:**it validates input and ensures values are non-negative.

* **addVehicle(Scanner scanner):**

**Purpose:**the purpose is it collects data from the user to create and add a new vehicle.

**Implementation:**it prompts the user for vehicle ID, location, destination, distance to destination, and battery level. It then creates a new Vehicle object and adds it to the list.

* **editVehicle(Scanner scanner):**

**Purpose:**the purpose is it updates the details of an existing vehicle.

**Implementation:**it searches for a vehicle by its ID, then prompts the user for new data to update the vehicle’s location, destination, distance, and battery level.

* **displayAllVehicles():**

**Purpose:**the purpose is it prints the details of all vehicles in the system.

**Implementation:**it iterates through the ArrayList and prints each vehicle’s details. If the list is empty, a message is displayed.

* **searchVehicle(int id):**

**Purpose:**the purpose is it looks up a vehicle by its ID.

**Implementation:**it iterates through the ArrayList and returns the vehicle that matches the provided ID or returns null if not found.

* **run(Scanner scanner):**

**Purpose:**the purpose is it implements the interactive menu for user input.

**Implementation:**it displays options (add, edit, display, exit) and uses the helper methods to validate user input, then calls the corresponding functions based on the user’s choice.

* **main(String[] args):**

**Purpose:**the purpose is it acts as the entry point for the program.

**Implementation:**it creates a VehicleManagementSystem object and starts the interactive menu.

### **VehicleSorter Class**

**Purpose:**The VehicleSorter class provides sorting functions for a list of vehicles based on two criteria:

* **Distance to Destination (ascending order):** Using a modified Heapsort.
* **Battery Level (descending order):** Using Quicksort.

The class includes methods to find the nearest vehicle and the one with the highest battery, along with a method to print the list of vehicles.

### **Key Methods and Their Implementations**

**Heapsort for Distance to Destination  
Method: heapSort(List<Vehicle> vehicles)**.

**Algorithm:**

**Building a Max-Heap:**The method starts by turning the list into a max-heap,this implementation is used as part of the extraction process to eventually achieve an ascending order.

**Data Structure: it us**es a List<Vehicle> as the underlying structure.

**Key Step:**The loop runs from the last non-leaf node to the root and calls the helper method trickleDown to ensure each subtree satisfies the heap property.

**Extracting Elements:**After the max-heap is built, the method repeatedly swaps the root (largest element) with the last element in the heap, then reduces the heap size and restores the heap property. This process results in the list being sorted in ascending order of distance.

**Helper Method: trickleDown(List<Vehicle> vehicles, int numItems, int i)**

**Implementation: it c**ompares the current element with its left and right children, swaps with the largest child if necessary, recursively applies the process to the affected subtree.

**Data Structures:**Works with indices in the List<Vehicle>, treating it like an array-based heap.

* **Quicksort for Battery Level  
  Method: quickSort(List<Vehicle> vehicles, int low, int high)**

**Algorithm: it u**ses the Quicksort algorithm, which is based on partitioning the list around a pivot; recursively sorts the sublists before and after the pivot.

**Key Data Structures:**it operates directly on the List<Vehicle> without needing extra storage.

**Helper Method: partition(List<Vehicle> vehicles, int low, int high)**

**Implementation: t**he pivot is chosen as the element at index high; the method then iterates through the sublist, swapping elements as needed to place higher battery levels to the left; returns the correct position of the pivot after partitioning.

* **Finding the Best Vehicles**

**Method: findNearestVehicle(List<Vehicle> vehicles)**

**Implementation: it it**erates over all vehicles and keeps track of the one with the minimum distance.

**Method: findVehicleWithHighestBattery(List<Vehicle> vehicles)**

**Implementation:it i**terates over all vehicles and tracks the one with the maximum battery level.

**Method: printVehicles(List<Vehicle> vehicles)**

**Purpose:**the purpose is it print the details of each vehicle in the list.

**Implementation:**it simply iterates over the list and prints the output of each vehicle’s toString() meth

### **AutonomousVehicleManagementSystem (Main Class)**

* **Purpose**: This is the entry point of the program, providing the user interface to interact with the system; presents a menu to the user and allows them to perform operations related to road networks, vehicle details, and vehicle sorting/searching.
* **Key Methods**:
  + **main(String[] args)**:

This method is responsible for displaying the main menu, capturing user input, and directing the user to the appropriate functionality; the menu offers options to interact with the road network (Graph), manage vehicle details (VehicleHashTable), and sort/search vehicles (VehicleManagementSystem); It repeatedly prompts the user for a choice and calls the respective methods based on user input.

**Data Structures**: This class relies on the use of Scanner for user input and manages interaction flow without involving complex data structures.

**graphMenu()**:

Provides an interface for managing the road network. The user can add locations, roads, check path existence, and display the road network.

**Data Structures**: Uses the Graph class to store and manage locations and roads. Internally, a graph typically uses **adjacency lists** or **adjacency matrices** to represent connections between locations.

**vehicleHashTableMenu()**:

Provides an interface to manage vehicles using a hash table; users can add, delete, search for vehicles, and display all vehicles.

**Data Structures**: it uses the VehicleHashTable, which likely uses a **hash map** or **hash table** to store vehicle records; this allows for constant-time complexity (O(1)) for insertions, deletions, and searches

.

**vehicleSorterMenu()**:

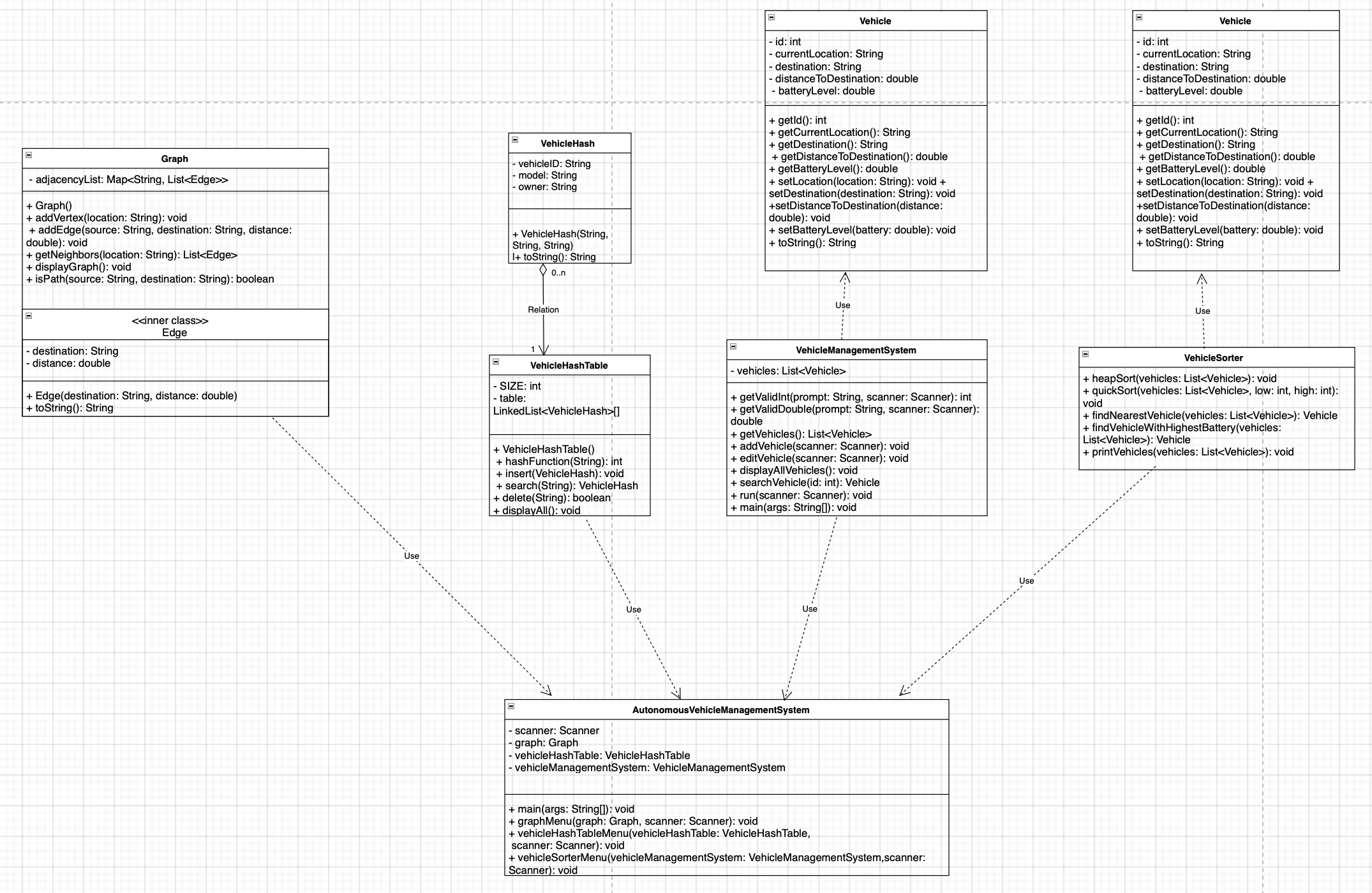
This interface allows the user to perform sorting and searching on a list of vehicles based on various criteria like distance and battery level.

**Data Structures**: it uses List<Vehicle> to store the vehicles and utilizes sorting algorithms like **Quick Sort** and **Heap Sort** to reorder the list based on different attributes.

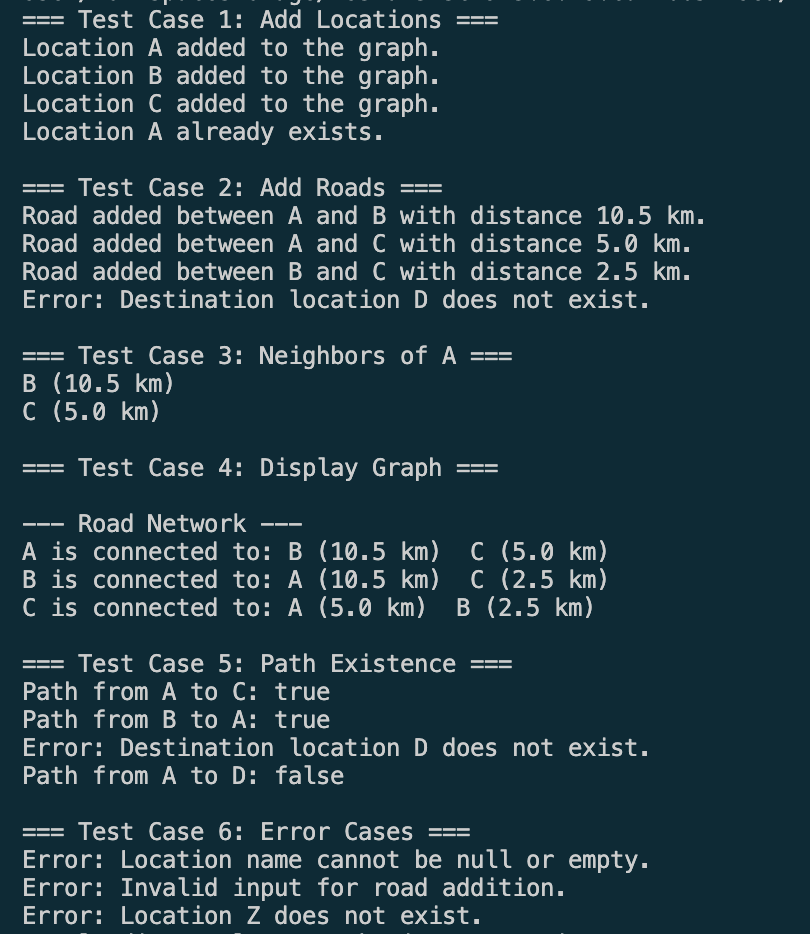
**vehicleManagementSystemMenu()**:

This method provides an interface for vehicle management tasks and allows the user to manage vehicles, sort or search vehicles based on distance and battery level, etc. The user will be directed to the sorting or searching operations through this menu.

**Data Structures**: it uses the VehicleManagementSystem to display the relevant operations for managing and sorting vehicles the method calls the run() method from **VehicleManagementSystem**, which executes vehicle-related tasks and displays its own menu to the user for further interaction.

UML diagram:

Test cases: Graph.java

Output:

 **Functionality Testing**:

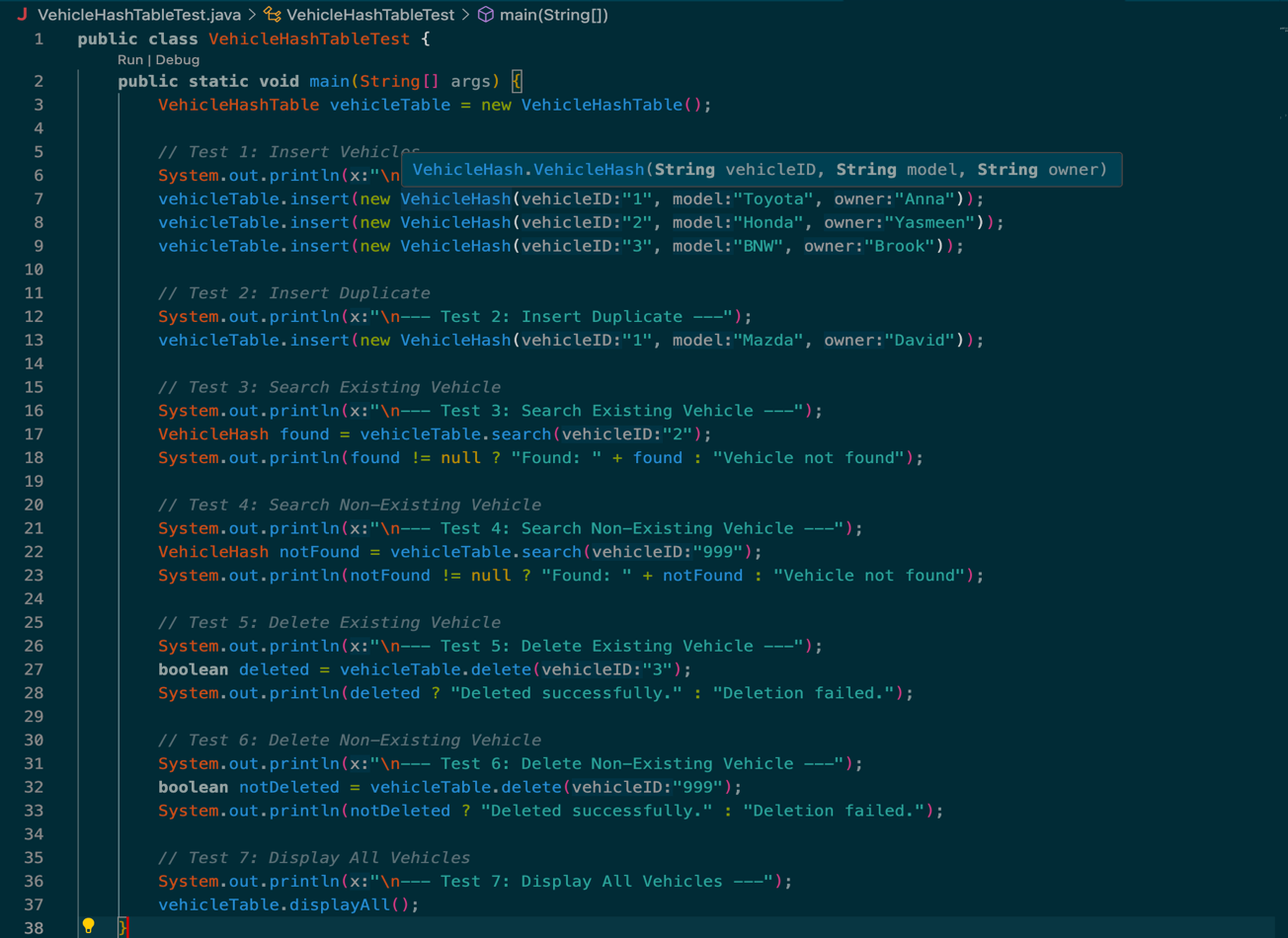
* **What was tested**: The core functionality of the graph, including adding/removing nodes and edges.
* **How it was tested**: Manually inserted nodes and edges, and then verified the correct structure by checking for the existence of nodes and edges.

 **Traversal Testing**:

* **What was tested**: Traversal methods like Depth-First Search (DFS) and Breadth-First Search (BFS).
* **How it was tested**: After adding a set of nodes and edges, you tested both DFS and BFS to ensure they visit the correct nodes in the expected order.

**Edge Case Testing**:

* **What was tested**: Edge cases such as an empty graph, graphs with a single node, or disconnected graphs.
* **How it was tested**: Ran tests on these scenarios to ensure the graph handles them properly without errors.

VehicleHashTableTest

Output:  


 **Basic Functionality Testing**:

**What was tested**: The ability to add, retrieve, and remove vehicles in the hash table.

**How it was tested**: Vehicles were added to the hash table with unique IDs. Retrieval was performed using these IDs, and the vehicles were checked to ensure the correct data was returned. Vehicles were also removed, and their absence was confirmed.

 **Collision Handling Testing**:

**What was tested**: The hash table’s ability to handle collisions (when two or more vehicles have the same hash).

**How it was tested**: Multiple vehicles with intentionally chosen IDs that lead to hash collisions were added to the table. After adding, retrieval of these vehicles was tested to ensure no data loss or errors occurred.

VehicleManagementSystemTest:



Output:

 **Basic Functionality Testing**:

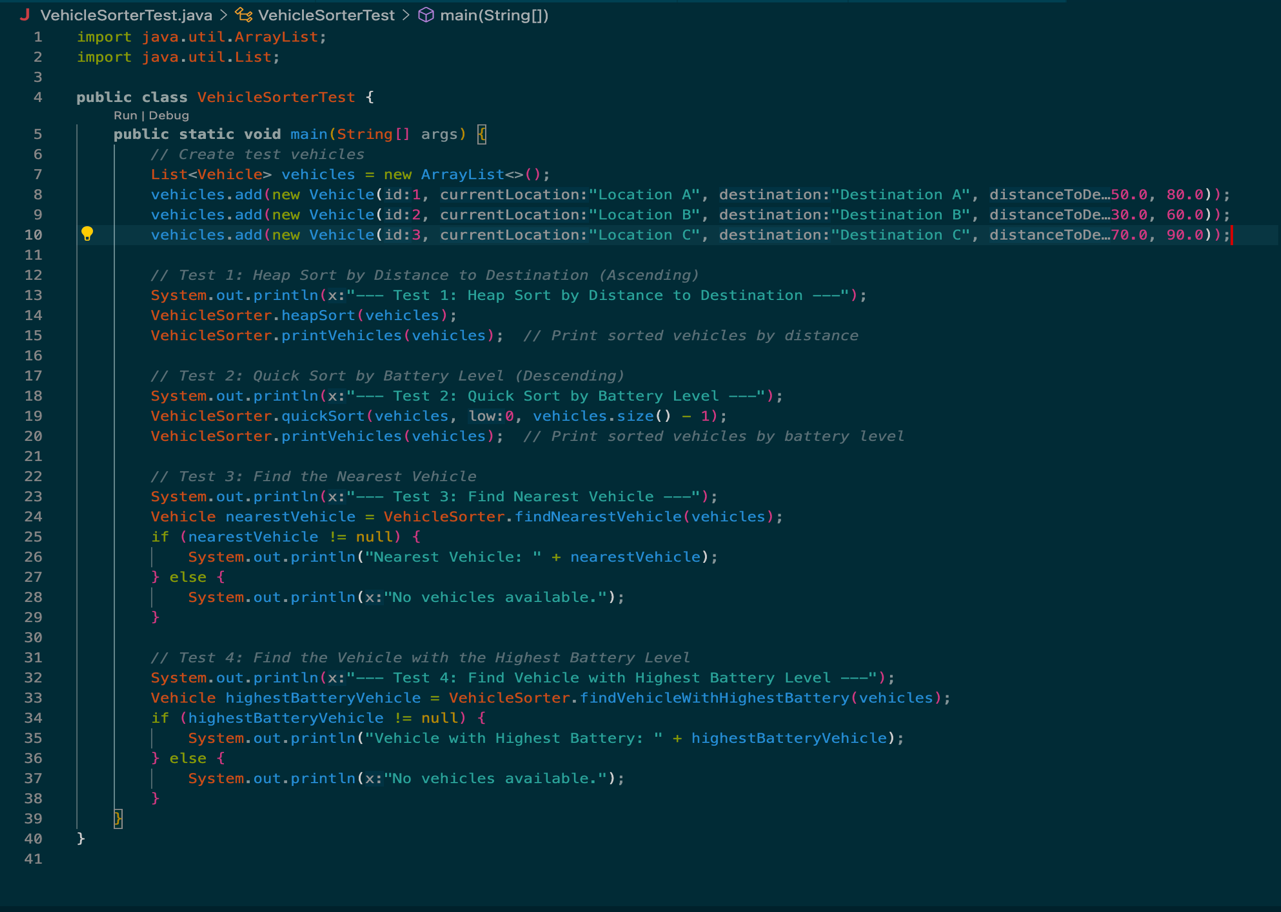
* **What was tested**: The ability to add, edit, and display vehicles in the system.
* **How it was tested**: Vehicles were added to the system with valid data (ID, location, destination, distance, battery level). Then, the displayAllVehicles() method was called to ensure that the added vehicles appeared correctly. Vehicles were also edited with new data, and their updated information was displayed to verify changes.

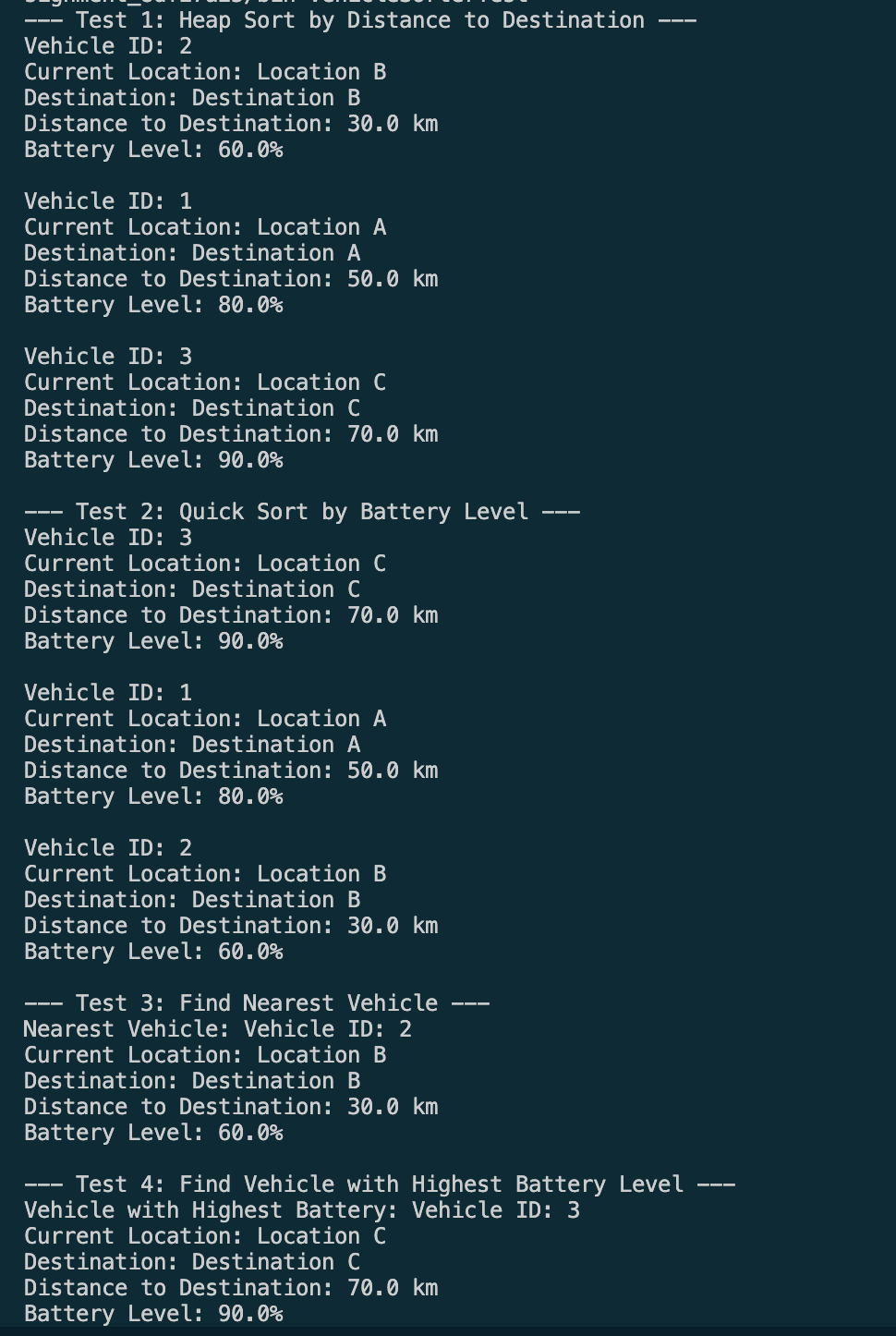
 **Input Validation Testing**:

* **What was tested**: The system's handling of invalid input during vehicle addition and editing.
* **How it was tested**: Invalid inputs, such as non-numeric values for vehicle ID or distance, were provided to test how the system handles them. The input validation ensured that the program prompts the user again if the input was invalid, preventing crashes or incorrect data.

 **Search Functionality Testing**:

* **What was tested**: The ability to search for a vehicle by its ID.
* **How it was tested**: The searchVehicle() method was tested by searching for vehicles that existed in the system. Results were checked to ensure the correct vehicle was returned. A test was also conducted for a non-existing vehicle ID to confirm that the system correctly returns null when the vehicle is not found

VehicleSorterTest:

Output

**Sorting by Distance (Heapsort)**:

**What was tested**: The sorting of vehicles by their distance to the destination using the heapsort algorithm.

**How it was tested**: A list of vehicles was provided, each with a specific distance to its destination; After calling the heapsort() method, the vehicles were checked to ensure they were sorted in ascending order based on the distance. The list was visually inspected to confirm that the smallest distance was at the top and the largest was at the bottom.

**Sorting by Battery Level (Quicksort)**:

**What was tested**: The sorting of vehicles by their battery level in descending order using the quicksort algorithm. **How it was tested**: A list of vehicles with different battery levels was created. After calling the quicksort() method, the vehicles were verified to be sorted from the highest to the lowest battery level. The list was checked for correctness by inspecting the order of the vehicles based on their battery levels.

**Challenges and Solutions**

1. **Algorithm Implementation**
   * **Challenge** The challenge I faced was applying algorithms like HeapSort and QuickSort on custom objects (e.g., Vehicle) which was complex at first.
   * **Solution**: so then I broke the algorithms down step-by-step, referred to algorithm resources, and tested with small data sets to understand their behavior.
2. **Designing and Managing Multiple Classes**
   * **Challenge**: I was having difficulty in coordinating interactions between classes like Vehicle, VehicleManager, VehicleHashTable, and VehicleSorter which was initially confusing.
   * **Solution**: so I used clear method names and focused on separating responsibilities for each class and followed OOP principles like encapsulation and modularity.
3. **Debugging and Testing**
   * **Challenge**: I had difficulty in Identifying bugs or incorrect logic in sorting, searching, and data structure operations.
   * **Solution**: Then I wrote test cases for each function and printed outputs to verify results. And used step-by-step debugging to trace issues.

**Efficiency Analysis and Potential Improvements**

**1. HeapSort (VehicleSorter - sort by distance):**

* **Time Complexity:**
  + Building the heap: **O(n)**
  + Sorting: **O(n log n)**
* **Space Complexity:**
  + **O(1)** (in-place sorting)
* **Efficiency:**  
  It is efficient for in-place sorting without additional memory use, but can be harder to understand due to its tree-like logic.
* **Improvement:**  
  Can consider using **MergeSort** for better clarity (also O(n log n)), especially if recursive logic is easier to follow.

**2. QuickSort (VehicleSorter - sort by battery level):**

* **Time Complexity:**
  + Average: **O(n log n)**
  + Worst case: **O(n²)** (when pivot is poorly chosen)
* **Space Complexity:**
  + **O(log n)** due to recursion stack
* **Efficiency:**  
  It is fast in practice, but can degrade on nearly sorted or reverse-sorted lists.
* **Improvement:**  
  It is randomized pivot or hybrid approach (e.g., QuickSort for large arrays + InsertionSort for small ones) can improve worst-case performance.

**3. Searching for nearest or highest battery vehicle:**

* **Time Complexity:** **O(n)**
* **Efficiency:**  
  It is straightforward linear search, efficient for small lists.
* **Improvement:**  
  If frequent searches are needed, maintain a **min-heap or max-heap** structure to reduce search time to **O(1)** for nearest or highest.

**4. HashTable (VehicleHashTable):**

* **Time Complexity:**
  + Insert/Search/Delete: **O(1)** average, **O(n)** worst-case (with collisions)
* **Space Complexity:** **O(n)**
* **Efficiency:**  
  Efficient for quick lookups by ID.

**Conclusion**

This assignment helped me learn how to use data structures and algorithms, like HashTables, Graphs, and sorting methods (HeapSort and QuickSort) and how they are structured and are very helpful in creating and I also understood how to organize code better using object-oriented programming and how to check if everything works correctly by writing test cases.

It also showed me how important it is to write code that is clean and easy to fix. I improved my problem-solving skills and became more confident in writing and testing Data Algorithms. Overall, this assignment was a good learning experience and helped me grow as a beginner developer.

.