**Final Report for Railway Network Support System (Version I)**

**Authors:**

* Name 1 (Student ID)
* Name 2 (Student ID)

**1. Overview**

This report outlines the development of the Railway Network Support System implemented in Java 21 for **Version I**, as specified in the handout. The project does not utilize the java.util package except for the Scanner class, ensuring a design that relies on custom data structures and implementations.

The system is designed to manage a railway network, providing operations to:

* Insert, remove, and query railway lines.
* Add, remove, and query train schedules.
* Find the best route based on user criteria.
* Persist data across program executions.

**2. Data Structure Choices**

**RailwaySystem**

* **Structure:** Custom dictionary implemented as a hash table.
* **Purpose:** Maps line names (case-insensitive) to Line objects.
* **Justification:**
  + Fast lookups for operations like retrieving a line or checking existence.
  + Minimal overhead for insertions and deletions.

**Line**

* **Structure:**
  + **Stations:** Custom dynamic array.
  + **Schedules:** Linked list.
* **Purpose:** Maintains a list of stations and associated train schedules.
* **Justification:**
  + Dynamic array ensures efficient ordered storage of stations.
  + Linked list allows efficient insertion and deletion of schedules while preserving order.

**Schedule**

* **Structure:** Array-based structure storing (station, time) pairs.
* **Purpose:** Represents a train's timetable for a specific line.
* **Justification:**
  + Simplicity and direct indexing for sequential operations.
  + Efficient traversal and validation of constraints (e.g., increasing times).

**3. Implementation Details**

**Phase 1 Operations**

1. **Insert Line (IL)**
   * **Description:** Adds a new line with a sequence of stations.
   * **Validation:** Checks if the line name already exists.
   * **Edge Cases:** Handles case-insensitivity and empty station lists.
   * **Data Structures Used:**
     + Custom hash table for line storage.
     + Dynamic array for station names.
2. **Remove Line (RL)**
   * **Description:** Removes a line, deleting stations not shared with other lines.
   * **Validation:** Ensures the line exists before deletion.
   * **Data Structures Used:**
     + Iterates through the hash table.
3. **List Stations of a Line (CL)**
   * **Description:** Lists stations of a specific line in insertion order.
   * **Validation:** Checks if the line exists.
   * **Data Structures Used:**
     + Traverses the dynamic array in Line.
4. **Insert Schedule (IH)**
   * **Description:** Adds a train timetable to a line.
   * **Validation:**
     + Ensures the first station is a terminal station.
     + Validates strictly increasing times and no train overtaking.
   * **Data Structures Used:**
     + Linked list for schedules.

**Phase 2 Operations**

1. **List Lines by Station (CE)**
   * **Description:** Lists all lines passing through a station in lexicographic order.
   * **Validation:** Ensures the station exists.
   * **Data Structures Used:**
     + Traverses the hash table and checks each line’s dynamic array.
2. **List Trains by Station (LC)**
   * **Description:** Lists all trains passing through a station in increasing order of departure time.
   * **Validation:** Ensures the station exists.
   * **Data Structures Used:**
     + Aggregates schedules from all lines using the linked lists.
3. **Find Best Timetable (MH)**
   * **Description:** Finds the train route that arrives closest to the expected time without being late.
   * **Validation:**
     + Ensures the line, departure station, and destination station exist.
     + Checks for feasibility of the route.
   * **Data Structures Used:**
     + Sequentially scans the linked list of schedules for the given line.

**4. Complexity Analysis**

**Time Complexity**

1. **Insert Line (IL)**:
   * Best/Worst Case: O(1)O(1) (hash table insertion).
2. **Remove Line (RL)**:
   * Best Case: O(1)O(1) (line not found).
   * Worst Case: O(L+S)O(L + S), where LL is the number of lines, SS is the average number of stations.
3. **List Stations of a Line (CL)**:
   * Best/Worst Case: O(S)O(S), where SS is the number of stations in the line.
4. **List Lines by Station (CE)**:
   * Best Case: O(1)O(1) (station not found in the first line).
   * Worst Case: O(L×S)O(L \times S), where LL is the number of lines.
5. **List Trains by Station (LC)**:
   * Best/Worst Case: O(L×T)O(L \times T), where TT is the average number of schedules per line.
6. **Find Best Timetable (MH)**:
   * Best Case: O(1)O(1) (first schedule matches).
   * Worst Case: O(T×S)O(T \times S), where TT is the number of schedules, SS is the number of stops per schedule.

**Space Complexity**

* **RailwaySystem**:
  + O(L+S+T)O(L + S + T), where LL = number of lines, SS = total stations, TT = total schedules.
* **Line**:
  + O(S+T)O(S + T), where SS = number of stations, TT = schedules.
* **Schedule**:
  + O(S)O(S), where SS = number of stops.

**5. Challenges and Solutions**

1. **No Java.util Classes:**
   * Created custom hash table, dynamic arrays, and linked lists to manage data.
   * Implemented basic sorting and lexicographic comparisons manually.
2. **Efficient Train Scheduling:**
   * Ensured schedules are validated without violating constraints (e.g., no overtaking).

**6. Conclusion**

This project successfully implements a railway network management system in Java, adhering strictly to **Version I** constraints. All functionalities were implemented using custom data structures, ensuring compliance and efficiency.