Project Description: Ticket to Ride

Objective:

The objective of this project is to develop a route planner that allows users to travel from one destination to another, using multiple routes. The students will utilize a Java **Linked List** data structure to represent various routes between destinations, calculate distances between them, and determine the quickest path based on the total distance or time.

Project Overview:

In this project, students will create a **Linked List**-based system where each node represents a **destination** and the edges between the nodes represent **routes** (paths with a distance or time). The system will allow users to:

1. **Define multiple routes** between a set of destinations.
2. **Store these routes** in a linked list, where each node holds information about a destination and its connected destinations (edges).
3. **Find the shortest route** between two destinations, based on the criteria of either distance or time.

Requirements:

1. **Classes and Data Structures:**
    - **DestinationNode**: Represents a destination in the system. Each node contains:
        - The **destination name** (e.g., "Paris", "Berlin", etc.).
        - A list of **connected destinations** (i.e., the routes from this destination to others).
        - The **distance** or **time** associated with the route.
    - **LinkedList**: A custom implementation of a linked list to store **DestinationNodes**. You can leverage the ones we've already made, you just need to add some extra methods for finding the shortest route.
    - **RoutePlanner**: A class that manages the linked list and performs operations such as adding destinations, adding routes between destinations, and finding the shortest path.
2. **Operations:**
    - **Adding Destinations and Routes**:
        - Allow the user to add destinations and define routes between them. Each route will include a distance or travel time.
    - **Finding the Shortest Path**:

        - Implement a method to calculate the shortest path between two destinations based on **total distance** or **total time** (depending on user choice).
        - The algorithm used for pathfinding could be a simple version of **Dijkstra's Algorithm** or **Breadth-First Search (BFS)** for simplicity.
    - **Display All Routes**:
        - Provide a method to display all available routes between destinations.
    - **Input and Output**:
        - The user should input starting and ending destinations, and the system should output the shortest route with the total distance or time required.
3. **Optional Advanced Features:**

    - **Multiple Criteria**: Let the user choose whether they want to find the fastest route (based on time) or the shortest route (based on distance).
    - **Graph Representation**: Represent the entire set of routes as a graph where nodes are destinations and edges are routes with weights (distances/times).
    - **Error Handling**: Implement error handling for cases like invalid destinations, non-existent routes, or no path available.

Example Scenario:

Let's say you are building a route planner for a travel agency. The destinations are cities, and the routes are flights or roads connecting them.

1. **Add Destinations**: You start by adding several cities like "Paris," "Berlin," "Madrid," and "Rome."
2. **Add Routes**: You then define the routes between these cities:
    - Paris -> Berlin (distance: 300 km, time: 3 hours)
    - Berlin -> Madrid (distance: 1500 km, time: 12 hours)
    - Madrid -> Rome (distance: 1400 km, time: 14 hours)
    - Paris -> Madrid (distance: 1000 km, time: 10 hours)
    - Paris -> Rome (distance: 1200 km, time: 11 hours)
3. **Find the Shortest Route**: A user queries the system to find the quickest route from Paris to Rome. The system will calculate the total distance and/or time for all possible routes and return the one that is the fastest or shortest, depending on the user's preference.

Example Output:

```
Unset
Starting destination: Paris
Ending destination: Rome
Shortest path:
```

```
Paris -> Rome (Distance: 1200 km, Time: 11 hours)
```

Grading Criteria:

- **Correctness of the Linked List Implementation**: The system should correctly implement a linked list to manage the destinations and routes.
- **Pathfinding Algorithm**: The algorithm used to find the shortest route should be correct and efficient.
- **User Interface**: The input and output should be clear and intuitive, allowing the user to easily interact with the system.

Submission Requirements:

- **Java Code**: Submit the full implementation of your project (Java source code files).
- **Documentation**: Include a brief report explaining your design choices, the classes you implemented, and how the program works. Give extra focus to your path finding method.
- **Test Cases**: Provide a few sample inputs and outputs, demonstrating that the system works as expected.