Task 1: You are designing a **network connectivity verifier** for a distributed system. The system consists of Node objects that belong to different groups. These groups are managed using **Disjoint Set Union (DSU)** (Union-Find).
Your task is to:

1. Implement a base class Node and a derived class ServerNode.
2. Implement a Network class using **Disjoint Set Union (DSU)**.
3. Throw appropriate exceptions when invalid operations are performed.

Output:

ServerNode 1 connected to Node 2
A and B are NOT connected.
Exception: Cannot connect to a null node!
Exception: Node not found in the network!

Task 2: A modern transportation company needs to simulate its fleet, which includes electric, gasoline, and hybrid vehicles. Hybrid vehicles combine both electric and gas engines while sharing common vehicle attributes. Additionally, every vehicle must maintain a log of its service history (maintenance events and repairs) to ensure reliable operation.

- **Vehicles:** Have general attributes such as make, model, year, and mileage.
- **Engines:**
  - *Electric vehicles* are equipped with a battery system (battery level in percentages).
  - *Gasoline vehicles* have a fuel system (fuel level in liters).
  - *Hybrid vehicles* can operate in either electric or gas mode, with the ability to switch between them.
- **Operations:** Vehicles must be able to start and stop their engines with behavior specific to their power source.
- **Service History:** Each vehicle internally records service events.
- **Usability:** Vehicles should be comparable based on mileage (using an overloaded < operator) and display detailed information (using an overloaded << operator), including their service records.

Using the case study above, design and implement an object-oriented solution in C++ that includes the following ideas—while you decide on the class names, attributes, and methods based on the narrative:

- **Abstract Base Class:** Create a base vehicle class that encapsulates common properties (e.g., make, model, year, mileage) and declares pure virtual functions for starting and stopping the engine. Overload the < operator to compare vehicles based on mileage.
- **Derived Classes:**
  - Develop a class for electric vehicles with an added battery-level property and a method to charge the battery.
  - Develop a class for gasoline vehicles with an added fuel-level property and a method to refuel.
- **Hybrid Vehicle:**
  - Create a hybrid vehicle class that inherits from both electric and gasoline vehicle classes, avoiding duplication of the common vehicle attributes.
  - Implement functionality to toggle between electric and gas modes.
  - Overload the << operator to display detailed information about the vehicle, including its service history.
- **Service History:**
  - Ensure each vehicle can manage its own log of maintenance events. Decide how to encapsulate and maintain this internal record.
- **Main Function:**
  - Demonstrate your design by creating at least one hybrid vehicle object.
  - Simulate operations such as charging, refueling, toggling modes, starting/stopping the engine, adding service records, comparing vehicles, and printing vehicle details.
  - Attempt (and comment out) direct access to a private attribute (e.g., the vehicle's make) to illustrate why encapsulation is important.
- **Conceptual Analysis:**
  - **(a)** Briefly explain how virtual inheritance can solve the diamond problem in this scenario, and what might happen if it were omitted.
  - **(b)** Discuss the benefits and drawbacks of overloading operators (like < and <<) in your class hierarchy, especially regarding access to internal state.
  - **(c)** Reflect on the decision to internally manage a vehicle's service history. How does this approach help organize functionality and maintain data integrity?

Task1: You are tasked with creating a simulation for different types of vehicles. The system must handle electric, gasoline, and hybrid vehicles. Hybrid vehicles combine electric and gas engines and must inherit from both types without duplicating the common base (i.e., the general Vehicle attributes).

1. **Abstract Base Class – Vehicle:**
   - Create an **abstract** class named Vehicle that includes:
     - **Private data members**: std::string make, std::string model, and int year.
     - **Protected data member**: double mileage (in kilometers).
     - **Public getter and setter functions** for all private members.
     - A **pure virtual function** startEngine() and stopEngine() that derived classes must implement.
     - An **overloaded operator <** to compare two vehicles based on mileage (i.e., lower mileage means "less").
2. **Derived Classes with Virtual Inheritance:**
   - **ElectricVehicle:**
     - Virtually inherit from Vehicle.
     - Add a **private** member: double batteryLevel (percentage, from 0.0 to 100.0).
     - Provide a method charge(double amount) to increase the battery level, ensuring it does not exceed 100%.
     - Override startEngine() and stopEngine() to simulate electric engine behavior (e.g., print messages like "Electric engine starting...").
   - **GasVehicle:**
     - Virtually inherit from Vehicle.
     - Add a **private** member: double fuelLevel (in liters).
     - Provide a method refuel(double amount) to increase the fuel level.
     - Override startEngine() and stopEngine() to simulate gas engine behavior (e.g., print messages like "Gas engine starting...").
3. **Hybrid Vehicle – HybridVehicle:**
   - Inherit **publicly** from both ElectricVehicle and GasVehicle.
   - Add a **private** data member to track the current operating mode (e.g., an enum with ELECTRIC and GAS).
   - Override startEngine() and stopEngine() in such a way that they:
     - Check the current mode (electric or gas) and then call the corresponding base class's implementation.
     - Print an additional message indicating the active mode.
   - Provide a method toggleMode() to switch between electric and gas modes.
   - Overload the << operator to print detailed information about the hybrid vehicle, including make, model, year, mileage, battery level, fuel level, and current mode.
4. **Main Function Demonstration:**
   - Instantiate at least one HybridVehicle object.
   - Simulate realistic operations:
     - Charge the battery and refuel the gas tank.
     - Toggle the operating mode and call startEngine() and stopEngine() in both modes.
     - Compare two vehicles using the overloaded < operator based on their mileage.
5. **Conceptual Analysis:**
   - **(a)** Explain in a short paragraph how virtual inheritance solves the diamond problem in this scenario, and what issues might arise if it were omitted.
   - **(b)** Analyze the benefits and drawbacks of overloading operators (such as < and <<) in a class hierarchy, especially in relation to the object's internal state and encapsulation. Also, why friend functions or direct access should be avoided.

Task 2: At a logistics center, parcels are sorted and stored in a stack. Each parcel has a unique identifier and is destined for a specific delivery zone. Parcels are grouped by destination within the stack according to the following rules:

- **INSERT x zone:**
  When a parcel with identifier **x** from a given **zone** is inserted:
  - If the stack is empty or the destination zone of the parcel at the top differs from **zone**, start a new group for this zone and push the parcel.
  - If the top group is for the same zone, simply add the parcel to that group.
- **REMOVE:**
  Remove and output the identifier of the parcel from the top of the stack (i.e., the most recently inserted parcel in the top group). If a group becomes empty after a removal, remove that group from the stack.
- The operations end with a line containing the word **END**.

Input Format:

- Each line represents an operation
  - INSERT x zone — where **x** is the parcel identifier (e.g., P101) and **zone** is an integer representing its delivery zone.

Input	Output
INSERT P101 1 INSERT P202 2 INSERT P102 1 REMOVE INSERT P203 2 REMOVE REMOVE REMOVE END	P102 P203 P202 P101