**EXERCISE 1**

**Inventory Management System**

**1. Understanding the Problem**

**Importance of Data Structures and Algorithms**

Efficient data structures and algorithms are crucial in managing large inventories for several reasons:

* **Performance**: They ensure that operations like searching, adding, updating, and deleting products are fast, which is essential for maintaining efficient warehouse operations.
* **Scalability**: As the inventory grows, the system should handle the increased data efficiently without significant performance degradation.
* **Memory Management**: Efficient data structures help in optimizing memory usage, preventing wastage and ensuring the system runs smoothly.

**Suitable Data Structures**

Various data structures can be used to handle inventory data:

* **ArrayList**: Good for dynamic arrays where the size can grow as needed. It provides fast access to elements but can be slow for insertions and deletions.
* **HashMap**: Excellent for key-value pair storage, offering fast retrieval, insertion, and deletion based on the product ID.
* **LinkedList**: Useful when frequent insertions and deletions are needed, as these operations are generally more efficient than with arrays.
* **TreeMap**: Keeps elements in a sorted order, useful if you need to frequently traverse the inventory in a specific order.

For this scenario, a HashMap is a suitable choice due to its efficient handling of insertions, deletions, and lookups.

**2. Setup**

Create a new project for the inventory management system.

**3. Implementation**

**Defining the Product Class**

Define a Product class with attributes like productId, productName, quantity, and price. Include appropriate constructors, getters, setters, and a toString method for displaying product information.

**Choosing a Data Structure and Implementing Methods**

Choose a HashMap to store the products, using the productId as the key for fast retrieval.

Implement methods to add, update, and delete products from the inventory.

* **Add Product**: Adds a new product to the inventory.
* **Update Product**: Updates an existing product in the inventory.
* **Delete Product**: Deletes a product from the inventory.
* **Retrieve Product**: Retrieves a product by its productId.
* **Display All Products**: Displays all products in the inventory.

**4. Analysis**

**Time Complexity**

* **Add Product**:
  + Average Case: O(1) – Adding an element to a HashMap is generally a constant time operation.
  + Worst Case: O(n) – In case of hash collisions, it could degrade to linear time.
* **Update Product**:
  + Average Case: O(1) – Updating an existing element is a constant time operation.
  + Worst Case: O(n) – Similar to adding, in case of hash collisions.
* **Delete Product**:
  + Average Case: O(1) – Removing an element is generally a constant time operation.
  + Worst Case: O(n) – In case of hash collisions.
* **Retrieve Product**:
  + Average Case: O(1) – Retrieving an element by key is generally a constant time operation.
  + Worst Case: O(n) – In case of hash collisions.

**Optimization**

* **Load Factor Management**: By maintaining an optimal load factor, rehashing operations can be minimized, ensuring the HashMap remains efficient.
* **Hash Function**: Using a good hash function reduces the chances of collisions, ensuring that operations remain efficient.
* **Thread Safety**: In a multi-threaded environment, consider using ConcurrentHashMap to handle concurrent access.

This approach ensures that the inventory management system is efficient, scalable, and maintains optimal performance as the inventory size grows.