## Exercise 1:Inventory Management System

### 1. Understanding the Problem

In an inventory management system that handles a large number of products, data structures and algorithms play a critical role in ensuring performance, accuracy, and efficiency.

### Why Are Data Structures Important?

**Efficient Searching:**
When managing thousands of products, searching for a product by its ID or name using a basic list can be slow — it requires going through each item one by one (O(n) time). Instead, using a HashMap can reduce this time to O(1), allowing immediate access based on a product's ID.

**Faster Updates and Deletions:**
Inventory data changes frequently — for instance, prices or quantities are often updated. If we store data in a simple list, we need to search through it to make updates, which takes time. With a HashMap, we can directly access a product using its ID and update or delete it instantly.

**Organized Storage:**
Different structures serve different access needs:

- Arrays or Lists are good for sequential access
- HashMaps offer direct access using keys like product ID
- Trees are useful when we want data sorted by key

**Memory Efficiency:**
Choosing the right structure helps manage memory effectively, which is vital when storing large volumes of data.

**Scalability and Performance:**
As the inventory grows — from hundreds to thousands of products — the choice of structure determines whether the system remains fast or becomes sluggish. A well-designed system using efficient data structures ensures performance even as the data scales.

## 2. Suitable Data Structures for Inventory Management

For a system where you need to:

- Add new products
- Update details like quantity or price
- Delete items
- Search by product ID
- Display all items

Certain data structures are more appropriate than others:

**HashMap (Best Option):**
Ideal for fast, key-based access. Each product can be stored with its ID as the key. Common operations like adding, updating, deleting, or searching are done in constant time — O(1).

```
HashMap<Integer, Product> inventory = new HashMap<>();
```

**ArrayList (Not Preferred):**
While it's simple to use and fine for small data, searching or updating by ID is slow (O(n) time), since you must loop through the list.

**TreeMap (Optional - For Sorted Order):**
Useful if you want products automatically sorted by their IDs. Slightly slower than HashMap (O(log n)) but gives sorted output without extra steps.

**LinkedList (Not Suitable):**
Does not support key-based access. Searching by ID is inefficient (O(n)).

## 3. Time Complexity Analysis (Using HashMap)

HashMap<Integer, Product> inventory;

Where each productId maps to a Product object.

**Adding a Product:**

inventory.put(productId, product);

Time Complexity: **O(1)**
Insert the product at the hash index — no need to loop or search.

**Updating a Product:**

Product p = inventory.get(productId);

p.quantity = newQty;

p.price = newPrice;

Time Complexity: **O(1)**
Access the product by its ID and directly update the required fields.

**Deleting a Product:**

inventory.remove(productId);

Time Complexity: **O(1)**
Direct deletion using the product ID — no scanning or shifting needed.

## 4. Optimizations Used

To make the system even faster and more reliable, I implemented several best practices:

1. **Using Product ID as the Key:**
   This allows direct access without scanning the entire data.

For example:

```
inventory.get(101);
```

gives us the product instantly.

**2. Preventing Duplicate Entries Before Adding:**

Before adding a new product, I check if it already exists to avoid overwriting:

```
if (!inventory.containsKey(101)) {
   inventory.put(101, newProduct);
}
```

**3. Safe Deletion with a Check:**

Before deleting, I ensure the product is actually present:

```
if (inventory.containsKey(101)) {
   inventory.remove(101);
}
```

**4. Planning for Scalability:**

If I know the inventory may contain thousands of products, I define an initial capacity:

```
HashMap<Integer, Product> inventory = new HashMap<>(1000);
```

This minimizes the need for rehashing and improves performance when handling large datasets.