**Exercise 1: Inventory Management System**

**1. Understanding the Problem**

An inventory management system helps a warehouse efficiently store and retrieve information about products. As the number of products grows into hundreds or thousands, it becomes crucial to use appropriate **data structures and algorithms** to keep operations fast and scalable.

Without efficient algorithms, even a simple task like finding a product to update could take a long time, leading to poor system performance and user experience. The right choice of data structures allows us to:

* Access and modify products quickly.
* Scale the system as the inventory grows.
* Keep the memory usage optimized.
* Maintain readable and maintainable code.

**Choosing the Right Data Structure**

In this project, we are dealing with operations like **add**, **update**, **delete**, and **display** products. The choice of data structure directly impacts the efficiency of these operations.

Some options include:

| **Data Structure** | **Advantages** | **Drawbacks** |
| --- | --- | --- |
| ArrayList | Simple to use, maintains order, dynamic size | Slow updates/deletes for large lists |
| HashMap | Very fast search, add, update, delete (O(1) average) | No order maintained |
| TreeMap | Maintains sorted order of keys | Slightly higher overhead (O(log n)) |

For this exercise, I chose **ArrayList<Product>** because:

* It is simple and straightforward to implement.
* It allows us to maintain the order of product entry.
* For a small to medium warehouse, performance is acceptable.

**2. Project Setup**

I created a new Java project called **Inventory Management System**. The project consists of two main classes:

1. **Product** – represents each product in the inventory.
2. **Inventory** – manages a list of products and implements add, update, delete, and display operations.

**3. Implementation**

**Product Class**

The Product class has the following attributes:

* productId: unique identifier for the product.
* productName: name of the product.
* quantity: available stock quantity.
* price: price per item.

It includes:

* A constructor to initialize products.
* Getters and setters to access and modify product details.
* A toString() method to display product information neatly.

**Inventory Class**

The Inventory class uses an ArrayList<Product> to store products. The following methods are implemented:

* addProduct(Product product): Adds a new product to the inventory.
* updateProduct(int productId, String productName, int quantity, double price): Updates the product’s details based on productId.
* deleteProduct(int productId): Removes the product from the inventory.
* displayProducts(): Displays all products currently in inventory.

**4. Analysis**

**Time Complexity of Operations**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Add Product | O(1) (amortized) | ArrayList append operation |
| Update Product | O(n) | Need to scan the list to find matching productId |
| Delete Product | O(n) | Need to scan the list and shift elements after removal |
| Display Products | O(n) | Iterates over all products |

As shown above, **adding a product is very fast**, but updating and deleting require scanning the list — which takes linear time. This is acceptable for a small inventory but becomes inefficient for very large datasets.

**How to Optimize Further**

If the warehouse were to manage **thousands of products**, using a **HashMap<Integer, Product>** would be a better choice. Here's how it would compare:

| **Operation** | **ArrayList** | **HashMap** |
| --- | --- | --- |
| Add | O(1) | O(1) |
| Update | O(n) | O(1) |
| Delete | O(n) | O(1) |
| Display All | O(n) | O(n) |

HashMap would allow **constant-time lookup**, update, and delete, which is perfect for large-scale applications — though it does not maintain the order of products.

**5. Conclusion**

In this exercise, I implemented a simple but functional Inventory Management System using **ArrayList**. For small to medium warehouses, this is an easy-to-maintain solution with reasonable performance.

For future scalability:

* If we need faster updates or deletions, using **HashMap** would provide major performance benefits.
* If sorted display of products is also required, **TreeMap** could be an alternative.

This project demonstrates the importance of **choosing the right data structure** based on the size and requirements of the system. A good understanding of **algorithms and time complexity** ensures that the system will remain efficient as the business grows.