**Module 2 - Data Structures and Algorithms**

**Exercise 1: Inventory Management System**

**1. Understand the Problem**

**Why Data Structures and Algorithms are Essential:**

* Efficient data handling ensures fast access, minimal memory usage, and scalability.
* Inventory operations (add, search, update, delete) must be optimized to handle thousands of products without lag.

**Suitable Data Structures:**

| **Data Structure** | **Use Case** |
| --- | --- |
| **ArrayList** | Useful for simple lists and small to medium datasets; good for iteration. |
| **HashMap** | Ideal for fast lookup, insertion, and deletion using productId as the key. |
| **TreeMap** | Maintains sorted order based on key (e.g., sorted productId). |

For fast access by productId, we’ll use HashMap<Integer, Product>.

**2,3).Setup and Implementation:**

**Code:**

import java.util.HashMap;

class Product {

    int productId;

    String productName;

    int quantity;

    double price;

    public Product(int productId, String productName, int quantity, double price) {

        this.productId = productId;

        this.productName = productName;

        this.quantity = quantity;

        this.price = price;

    }

    @Override

    public String toString() {

        return "Product ID: " + productId + ", Name: " + productName +

               ", Quantity: " + quantity + ", Price: " + price;

    }

}

class InventoryManager {

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

    public void addProduct(Product product) {

        inventory.put(product.productId, product);

    }

    public boolean updateProduct(int id, int quantity, double price) {

        if (inventory.containsKey(id)) {

            Product p = inventory.get(id);

            p.quantity = quantity;

            p.price = price;

            return true;

        }

        return false;

    }

    public boolean deleteProduct(int id) {

        return inventory.remove(id) != null;

    }

    public void displayInventory() {

        if (inventory.isEmpty()) {

            System.out.println("Inventory is empty.");

        } else {

            for (Product p : inventory.values()) {

                System.out.println(p);

            }

        }

    }

}

public class InventorySystem {

    public static void main(String[] args) {

        InventoryManager manager = new InventoryManager();

        manager.addProduct(new Product(101, "Keyboard", 50, 799.99));

        manager.addProduct(new Product(102, "Mouse", 100, 499.49));

        manager.addProduct(new Product(103, "Monitor", 30, 6999.00));

        System.out.println("Initial Inventory:");

        manager.displayInventory();

        System.out.println("\nUpdating product 102...");

        boolean updated = manager.updateProduct(102, 80, 479.49);

        System.out.println(updated ? "Update successful." : "Product not found.");

        System.out.println("\nDeleting product 101...");

        boolean deleted = manager.deleteProduct(101);

        System.out.println(deleted ? "Deletion successful." : "Product not found.");

        System.out.println("\nFinal Inventory:");

        manager.displayInventory();

    }

}

**Output:**

****

**4. Analysis**

**Time Complexity:**

* **Add, Update, Delete** → O(1) using HashMap (fast key access)
* **Display All** → O(n), since we loop through all products

**Optimization Tips:**

* HashMap gives fast performance for operations using productId.
* Use TreeMap if sorted order (by name/price) is needed.
* For large inventories, consider indexing by multiple fields (like category/supplier).

**Optional Features:**

* Save/load inventory using file or serialization.
* Add search by product name (using string match).
* Track sold/deleted items in another list or map.