**Exercise 1: Inventory Management System**

**Why are Data Structures and Algorithms Important in Inventory Systems?**

An inventory system needs to handle **large volumes of product data** efficiently. This includes adding, updating, deleting, and retrieving product information. Poor performance in these areas can lead to:

* **Slower response times**
* **Data inconsistency**
* **Difficulty in scaling**

Using the **right data structures and algorithms** ensures:

* **Faster search and update operations**
* **Efficient memory usage**
* **Scalability as data grows**

**Suitable Data Structures for Inventory Management**

**🔹 1. HashMap**

* **Best for**: Fast lookup, insert, and delete by productId
* **Time Complexity**: Average-case **O(1)** for add, update, and delete
* **Use case**: Primary choice when productId is the key

**🔹 2. ArrayList**

* **Best for**: Iteration and ordered data
* **Time Complexity**: O(n) for searching and removing by value
* **Use case**: Not ideal for large datasets if fast access is needed

**🔹 3. TreeMap**

* **Best for**: Sorted data access
* **Time Complexity**: O(log n) for operations
* **Use case**: When you need to maintain products in sorted order

**✅ Chosen Structure:**HashMap<String, Product>

This allows constant-time operations using productId as the key.

**Source Code:**

**Product.java**

import java.util.Objects;

public class Product {

private final String productId;

private String productName;

private int quantity;

private double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public String getProductId() { return productId; }

public String getProductName() { return productName; }

public int getQuantity() { return quantity; }

public double getPrice() { return price; }

public void setProductName(String productName) { this.productName = productName; }

public void setQuantity(int quantity) { this.quantity = quantity; }

public void setPrice(double price) { this.price = price; }

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (!(o instanceof Product)) return false;

Product p = (Product) o;

return productId.equals(p.productId);

}

@Override

public int hashCode() {

return Objects.hash(productId);

}

@Override

public String toString() {

return productId + " | " + productName + " | Qty: " + quantity + " | ₹" + price;

}

}

**InventoryManager.java**

import java.util.Collection;

import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

public class InventoryManager {

private final Map<String, Product> catalog = new HashMap<>();

public boolean add(Product p) {

return catalog.putIfAbsent(p.getProductId(), p) == null;

}

public boolean update(Product p) {

return catalog.replace(p.getProductId(), p) != null;

}

public boolean delete(String productId) {

return catalog.remove(productId) != null;

}

public Product find(String productId) {

return catalog.get(productId);

}

public Collection<Product> listAll() {

return Collections.unmodifiableCollection(catalog.values());

}

}

**Demo.java**

public class Demo {

public static void main(String[] args) {

InventoryManager inv = new InventoryManager();

inv.add(new Product("SKU-101", "Widget-A", 150, 79.99));

inv.add(new Product("SKU-102", "Widget-B", 90, 59.50));

inv.update(new Product("SKU-102", "Widget-B-New", 120, 64.00));

inv.delete("SKU-101");

inv.listAll().forEach(System.out::println);

}

}

**Output:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Time Complexity Analysis**

| **Operation** | **Data Structure Used** | **Time Complexity** | **Explanation** |
| --- | --- | --- | --- |
| Add Product | HashMap | O(1) average | Uses hashCode and putIfAbsent |
| Update Product | HashMap | O(1) average | Uses key lookup and replace |
| Delete Product | HashMap | O(1) average | Uses key lookup and remove |
| Find Product | HashMap | O(1) average | Uses key lookup |
| List All | HashMap | O(n) | Iterates over all entries |

**Note:**

Worst-case time complexity may rise to **O(n)** during hash collisions or rehashing.

**Optimizing Operations**

**🔹 1. Load Factor and Initial Capacity**

* Adjust initial capacity to avoid rehashing
* Set load factor to balance performance vs memory

**🔹 2. Use of ConcurrentHashMap**

* If using in a **multi-threaded environment**, switch to ConcurrentHashMap for thread safety

**🔹 3. Secondary Indexes**

* Create additional maps like:
  + Map<Double, List<Product>> to quickly find products by price
  + Map<String, List<Product>> for category-wise retrieval

**🔹 4. Range Queries**

* Use TreeMap if you want to perform **range-based lookups** (e.g., price between ₹100 and ₹500)

**🔹 5. Caching Frequently Accessed Products**

* Use an **LRU Cache** (e.g., LinkedHashMap with eviction policy) to cache hot products