# Exercise 1: Inventory Management System

#### 1.Explain why data structures and algorithms are essential in handling large inventories.

* **Efficient Storage**: Data structures help in organizing and storing inventory data efficiently. This ensures that space is utilized optimally.
* **Fast Retrieval**: Algorithms enable quick search, retrieval, and update operations, which are crucial for real-time inventory management.
* **Scalability**: Proper data structures and algorithms ensure that the system can handle a growing number of products without significant performance degradation.
* **Complex Operations**: Advanced algorithms can be used for inventory forecasting, restocking alerts, and optimization, which are important for warehouse management.

#### 2. Discuss the types of data structures suitable for this problem.

* **ArrayList**: Good for storing a dynamic list of products where order matters and random access is needed. It provides average O(1) time complexity for accessing elements but O(n) for insertions and deletions in the middle.
* **HashMap**: Ideal for situations where fast lookups, insertions, and deletions are needed. It offers average O(1) time complexity for these operations. It maps product IDs to product details.
* **LinkedList**: Useful if frequent insertions and deletions are required, but less efficient for random access as it provides O(n) time complexity for accessing elements.
* **TreeMap**: Useful if you need a sorted map of products. It provides O(log n) time complexity for insertions, deletions, and lookups.

#### 

#### 3.Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price.**

#### public class Product {

private 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;

}

// Getters and Setters

public String getProductId() {

return productId;

}

public void setProductId(String productId) {

this.productId = productId;

}

public String getProductName() {

return productName;

}

public void setProductName(String productName) {

this.productName = productName;

}

public int getQuantity() {

return quantity;

}

public void setQuantity(int quantity) {

this.quantity = quantity;

}

public double getPrice() {

return price;

}

public void setPrice(double price) {

this.price = price;

}

}

#### 4.Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).

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

import java.util.HashMap;

public class Inventory {

private HashMap<String, Product> productMap;

public Inventory() {

productMap = new HashMap<>();

}

// Method to add a product

public void addProduct(Product product) {

productMap.put(product.getProductId(), product);

}

// Method to update a product

public void updateProduct(Product product) {

if (productMap.containsKey(product.getProductId())) {

productMap.put(product.getProductId(), product);

} else {

System.out.println("Product not found!");

}

}

// Method to delete a product

public void deleteProduct(String productId) {

if (productMap.containsKey(productId)) {

productMap.remove(productId);

} else {

System.out.println("Product not found!");

}

}

// Method to display all products

public void displayProducts() {

for (Product product : productMap.values()) {

System.out.println("ID: " + product.getProductId() + ", Name: " + product.getProductName() +

", Quantity: " + product.getQuantity() + ", Price: " + product.getPrice());

}

}

}

#### 6.Analyze the time complexity of each operation (add, update, delete) .

#### **Time Complexity Analysis:**

* **Add Product**: The time complexity is O(1) on average since HashMap allows for constant-time insertion.
* **Update Product**: The time complexity is O(1) on average since updating an existing entry in a HashMap is also a constant-time operation.
* **Delete Product**: The time complexity is O(1) on average because deleting an entry from a HashMap is a constant-time operation.
* **Display Products**: The time complexity is O(n), where n is the number of products in the inventory, as we need to iterate over all entries to display them.

#### 

#### 7.Discuss how you can optimize these operations.

#### **Optimization:**

* **Load Factor and Rehashing**: Ensuring that the HashMap does not exceed a certain load factor (default is 0.75) helps maintain O(1) operations. When the load factor is exceeded, rehashing occurs, which can be optimized by initializing the HashMap with a proper capacity.
* **Efficient Use of Memory**: Ensure that the HashMap capacity is managed efficiently to avoid excessive memory usage and rehashing.
* **Concurrency**: If the inventory system is used in a multi-threaded environment, use ConcurrentHashMap instead of HashMap to handle concurrent updates and accesses.

Sample Test :

public class InventoryTest {

public static void main(String[] args) {

Inventory inventory = new Inventory();

Product product1 = new Product("P001", "Laptop", 10, 1000.00);

Product product2 = new Product("P002", "Smartphone", 20, 500.00);

Product product3 = new Product("P003", "Tablet", 15, 300.00);

// Adding products

inventory.addProduct(product1);

inventory.addProduct(product2);

inventory.addProduct(product3);

// Displaying products

inventory.displayProducts();

// Updating a product

product2.setQuantity(25);

inventory.updateProduct(product2);

// Displaying products after update

System.out.println("\nAfter updating quantity:");

inventory.displayProducts();

// Deleting a product

inventory.deleteProduct("P001");

// Displaying products after deletion

System.out.println("\nAfter deleting a product:");

inventory.displayProducts();

}

basic inventory management system using a HashMap to store and manage products, demonstrating efficient data storage and retrieval with appropriate methods for adding, updating, and deleting products.