**Exercise 1: Inventory Management System**

**Steps: Understand the Problem**

**Importance of Data Structures and Algorithms in Handling Large Inventories**

Data structures and algorithms are crucial in handling large inventories because they:

* **Enhance Efficiency:** Efficient data structures reduce the time complexity of operations such as searching, adding, updating, and deleting products.
* **Optimize Storage:** Appropriate data structures minimize memory usage, which is important when dealing with large amounts of data.
* **Improve Scalability:** Good algorithms ensure that the system can handle increasing amounts of data without significant performance degradation.

**Suitable Data Structures for Inventory Management**

* **ArrayList:** Useful for maintaining an ordered list of products, but has a higher time complexity for search, insert, and delete operations due to the need for shifting elements.
* **HashMap:** Provides average constant-time complexity for search, insert, and delete operations, making it efficient for large datasets. It allows fast retrieval using keys (e.g., productId).
* **TreeMap:** Maintains a sorted order of keys and allows efficient range queries, but has a higher time complexity (O(log n)) for basic operations compared to HashMap.

**Setup**

* We'll create a new project for the inventory management system. For demonstration, the implementation will be in Java.

**Implementation:**

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;

}

}

For this problem, **HashMap** is a suitable choice due to its average constant-time complexity for the core operations.

import java.util.HashMap;

public class Inventory {

private HashMap<String, Product> products;

public Inventory() {

products = new HashMap<>();

}

public void addProduct(Product product) {

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

}

public void updateProduct(String productId, Product newProduct) {

if (products.containsKey(productId)) {

products.put(productId, newProduct);

} else {

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

public void deleteProduct(String productId) {

products.remove(productId);}

public Product getProduct(String productId) {

return products.get(productId);}

public static void main(String[] args) {

Inventory inventory = new Inventory();

// Adding products

Product product1 = new Product("101", "Laptop", 10, 999.99);

Product product2 = new Product("102", "Smartphone", 20, 599.99);

inventory.addProduct(product1);

inventory.addProduct(product2);

// Updating a product

Product updatedProduct = new Product("101", "Laptop", 8, 949.99);

inventory.updateProduct("101", updatedProduct);

// Deleting a product

inventory.deleteProduct("102");

// Retrieving a product

Product product = inventory.getProduct("101");

System.out.println(product.getProductName() + " - " + product.getQuantity() + " - $" + product.getPrice()); }}

**Input Operations:**

Add Product 1:

ID: "101"

Name: "Laptop"

Quantity: 10

Price: $999.99

Add Product 2:

ID: "102"

Name: "Smartphone"

Quantity: 20

Price: $599.99

Update Product 1:

ID: "101"

New Quantity: 8

New Price: $949.99

Delete Product 2:

ID: "102"

Retrieve Product 1:

ID: "101"

**Output:**

Laptop - 8 - $949.99

**Analysis**

**Time Complexity of Operations**

* **Add Product:** O(1) on average. HashMap allows for constant-time insertion.
* **Update Product:** O(1) on average. Updating a value in HashMap is done in constant time.
* **Delete Product:** O(1) on average. Deletion in HashMap is also a constant-time operation.
* **Retrieve Product:** O(1) on average. Retrieving a product by key is a constant-time operation.

**Optimization**

* **Handling Collisions:** Use a good hash function to minimize collisions in HashMap.
* **Load Factor Management:** Maintain an optimal load factor (typically 0.75) to balance time and space complexity.
* **Concurrent Access:** For multi-threaded environments, use ConcurrentHashMap to avoid synchronization issues.