**Exercise 1: Inventory Management System**

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

Data structures and algorithms are crucial in handling large inventories efficiently due to the following reasons -

1. Efficiency –
   1. Time Complexity: Choosing optimal structures and algorithms makes operations like searching, inserting, updating, and deleting faster—even with growing inventory sizes. For example, linear search takes O(n) time in an unsorted list, but using binary search on a sorted array or leveraging balanced trees can reduce that to O(log n).
   2. Space Complexity: Thoughtful selection of data structures also helps conserve memory, which becomes crucial when storing vast amounts of inventory data.
2. Scalability - Well-designed algorithms and structures ensure that a system can handle more data without a proportional rise in processing time. This scalability is vital to maintain smooth performance as inventory size increases.

**Discuss the types of data structures suitable for this problem.**

The choice of data structure for an inventory system depends on its specific needs.

1. **ArrayList**

* *Pros:* Easy to use, supports dynamic resizing, and is well-suited for accessing items in order.
* *Cons:* Operations like searching and deleting can be slow in the worst case, taking O(n) time.

1. **HashMap**

* *Pros:* Offers fast average-case performance (O(1)) for inserting, deleting, and retrieving elements, making it ideal for quick searches.
* *Cons:* Doesn’t preserve the order of items, and if many hash collisions occur, performance can degrade to O(n).

**Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.**

1. **Add Operation**

public void addProduct(Product product) {

products.add(product);

}

Time Complexity:

Appending an element to the end of an ArrayList usually takes **O(1)** time, since the element is simply inserted into the next available position. However, if the internal array is full, the ArrayList must **resize** by allocating a larger array and copying over all existing elements, which is an **O(n)** operation. Since these resizing events occur infrequently, the **amortized time complexity** for insertions at the end remains **O(1)** overall.

1. **Update Operation**

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

boolean found = false;

for (Product product : products) {

if (product.getProductId() == productId) {

product.setProductName(productName);

product.setQuantity(quantity);

product.setPrice(price);

System.out.println("Product with ID : " + productId + " Updated");

found = true;

break;

}

}

if (!found) {

System.out.println("Invalid Product ID : " + productId);

}

}

Time Complexity: O(n)

Looking up a product by its productId in an ArrayList involves scanning each element one by one. If the desired item is at the end—or not present at all—the algorithm examines every element, resulting in a worst-case time complexity of O(n).

1. **Delete Operation**

public boolean deleteProduct(int productId) {

for (Product product : products) {

if (product.getProductId() == productId) {

products.remove(product);

System.out.println("Product with ID : " + productId + " Removed");

return true;

}

}

System.out.println("Invalid Product ID : " + productId);

return false;

}

Time Complexity: O(n)

Locating a product using its productId in an ArrayList involves checking elements sequentially, which leads to a linear search. Additionally, when an item is removed, all elements that follow must be shifted one position to the left. As a result, both operations—searching and removal—can take O(n) time in the worst-case scenario.

**Discuss how you can optimize these operations.**

To make these operations more efficient, a HashMap can be used with the productId as the key and the Product object as the value. This approach significantly improves performance by reducing the time complexity of key operations like search, insert, and delete.

* + 1. **Add Operation**

public void addProduct(Product product) {

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

}

Time Complexity: O(1) Inserting an element into a `HashMap` is O(1) on average.

**2. Update Operation**

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

Product product = productsMap.get(productId);

if (product != null) {

product.setProductName(productName);

product.setQuantity(quantity);

product.setPrice(price);

System.out.println("Product with ID : " + productId + " Updated");

} else {

System.out.println("Invalid Product ID : " + productId);

}

}

Time Complexity: O(1) Retrieving an element from a `HashMap` is O(1) on average.

**3. Delete Operation**

public boolean deleteProduct(int productId) {

if (productsMap.remove(productId) != null) {

System.out.println("Product with ID : " + productId + " Removed");

return true;

} else {

System.out.println("Invalid Product ID : " + productId);

return false;

}

}

Time Complexity: O(1) Removing an element from a `HashMap` is O(1) on average.

Implementation using HashMap –

import java.\*;

public class Inventory {

private Map<Integer, Product> productsMap;

public Inventory() {

this.productsMap = new HashMap<>();

}

// Method to add a product

public void addProduct(Product product) {

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

}

// Method to update a product

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

Product product = productsMap.get(productId);

if (product != null) {

product.setProductName(productName);

product.setQuantity(quantity);

product.setPrice(price);

System.out.println("Product with ID : " + productId + " Updated");

} else {

System.out.println("Invalid Product ID : " + productId);

}

}

// Method to delete a product

public boolean deleteProduct(int productId) {

if (productsMap.remove(productId) != null) {

System.out.println("Product with ID : " + productId + " Removed");

return true;

} else {

System.out.println("Invalid Product ID : " + productId);

return false;

}

}

// Method to display all products

public void displayProducts() {

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

System.out.println(product);

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

Inventory inventory = new Inventory();

int productId;

String productName;

int quantity;

double price;

// Adding Product

char makeEntry = 'y';

while (makeEntry == 'y') {

System.out.println("Enter the Product Details");

System.out.print("Product ID: ");

productId = sc.nextInt();

sc.nextLine();

System.out.print("Product Name: ");

productName = sc.nextLine();

System.out.print("Quantity: ");

quantity = sc.nextInt();

System.out.print("Price: ");

price = sc.nextDouble();

Product p = new Product(productId, productName, quantity, price);

inventory.addProduct(p);

System.out.print("Want to add another product [y/n]: ");

makeEntry = sc.next().charAt(0);

System.out.println();

}

// Displaying products

System.out.println("Products in inventory:");

inventory.displayProducts();

// Updating a product

System.out.print("Enter Product ID to update: ");

productId = sc.nextInt();

inventory.updateProduct(productId, "Gaming Laptop", 5, 95000.00);

// Displaying products after update

System.out.println("Products in inventory after update:");

inventory.displayProducts();

// Deleting a product

System.out.print("Enter Product ID to delete: ");

productId = sc.nextInt();

inventory.deleteProduct(productId);

// Displaying products after deletion

System.out.println("Products in inventory after deletion:");

inventory.displayProducts();

sc.close();

}

}

Using HashMap instead of ArrayList enhances the performance of update and delete operations by reducing their time complexity from O(n) to O(1) on average. This change significantly boosts system efficiency, particularly when managing large-scale inventories.