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

Data structures and algorithms are crucial in handling large inventories for several reasons:

1. **Efficiency**:
   * **Time Complexity**: Efficient data structures and algorithms ensure that operations such as search, insert, update, and delete can be performed quickly, even as the inventory grows. For instance, a linear search in an unsorted list has a time complexity of O(n), whereas a binary search in a sorted list or operations in a balanced tree can be O(log n).
   * **Space Complexity**: Proper data structures can help manage memory usage more efficiently, which is essential for large inventories.
2. **Scalability**:
   * Efficient algorithms and data structures allow systems to scale and handle increased loads without a proportional increase in processing time. This is critical for maintaining performance as the number of items grows.

### **Suitable Data Structures for Inventory Management**

Different data structures can be used depending on the specific requirements of the inventory system:

1. **ArrayList**:
   * **Advantages**: Simple to implement, provides dynamic resizing, and is efficient for sequential access.
   * **Disadvantages**: Searching and deleting elements can be slow (O(n) in worst case).
2. **HashMap**:
   * **Advantages**: Provides average O(1) time complexity for insert, delete, and search operations. Suitable for fast lookups.
   * **Disadvantages**: Does not maintain any order of elements, and the worst-case time complexity can degrade to O(n) if there are many hash collisions.

**Time Complexity Analysis**

**Using ArrayList**

**1. Add Operation**

public void addProduct(Product product) {

products.add(product);

}

Time Complexity: O(1)

- Adding an element to the end of an `ArrayList` is generally O(1) because it involves placing the element in the next available slot.

- Occasionally, the `ArrayList` needs to resize itself when it runs out of space, which involves copying all elements to a new array. This operation is O(n), but it happens infrequently, so the amortized time complexity remains O(1).

2.**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)

- Searching for a product by its `productId` requires a linear search through the `ArrayList`. In the worst case, this operation has a time complexity of O(n).

**3. 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)

- Finding the product by `productId` requires a linear search, and removing an element involves shifting subsequent elements, which is also O(n) in the worst case.

**Optimizations**

To optimize these operations, we can use a `HashMap` where the key is `productId` and the value is the `Product` object. This will reduce the time complexity of most operations.

**Using HashMap**

**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 Example with `HashMap**`

Here's an optimized version of the inventory management system using a `HashMap`:

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

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();

}

}

Therefore,

- Using `ArrayList`:

- Add: O(1)

- Update: O(n)

- Delete: O(n)

- Using `HashMap`:

- Add: O(1)

- Update: O(1)

- Delete: O(1)

Switching from `ArrayList` to `HashMap` optimizes the time complexity of update and delete operations from O(n) to O(1), making the system more efficient, especially for large inventories.