INVENTORY MANAGEMENT SYSTEM – EX-1

1. Understanding the Problem

Importance of Data Structures and Algorithms in Handling Large Inventories

Data structures and algorithms are crucial for managing large inventories because they enable efficient data storage, retrieval, and manipulation. Efficient data structures help minimize memory usage, while well-designed algorithms optimize the time complexity of operations such as adding, updating, and deleting products. This is especially important in scenarios where there are thousands or millions of products, as poor efficiency can lead to slow response times and degraded system performance.

Suitable Data Structures for Inventory Management

For an inventory management system, the following data structures are commonly used:

HashMap (Dictionary in Python, HashMap in Java): This data structure is ideal for storing products with a unique identifier (like productId) as the key. It offers average time complexity of O(1) for insertions, deletions, and lookups, making it efficient for managing a dynamic inventory.

ArrayList (List in Python, ArrayList in Java): Useful for maintaining an ordered list of products. However, it may not be as efficient as HashMap for lookups and updates since these operations have O(n) complexity in the worst case.

TreeMap: This data structure maintains the keys in sorted order and allows for efficient range queries. However, it has a higher time complexity of O(log n) for insertions and deletions compared to HashMap.

In this scenario, a HashMap is chosen for its efficiency in handling common operations like adding, updating, and deleting products.

3. Implementation

The implementation provided includes:

Class Product: Defines the properties of a product, including productId, productName, quantity, and price.

Class InventoryManagementSystem: Manages the inventory using a HashMap<String, Product>. This class provides methods to add, update, delete, and display products.

Code Explanation

Product Class:

Encapsulates the details of a product.

Includes getter and setter methods for accessing and modifying product attributes.

InventoryManagementSystem Class:

Manages a HashMap of products.

Methods:

addProduct(Product product): Adds a product to the inventory.

updateProduct(Product product): Updates the details of an existing product.

deleteProduct(String productId): Removes a product from the inventory.

getProduct(String productId): Retrieves a product based on its ID.

displayInventory(): Displays all products in the inventory.

code:

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 */

package com.mycompany.inventorymanagementsystem;

/**
 *
 * @author nikhi
 */

import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

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;
    }

    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;
    }
}

public class InventoryManagementSystem {
    private Map<String, Product> inventory;

    public InventoryManagementSystem() {
        this.inventory = new HashMap<>();
    }
```

```
public void addProduct(Product product) {
    inventory.put(product.getProductId(), product);
    // tc: O(1)
}


public void updateProduct(Product product) {
    if (inventory.containsKey(product.getProductId())) {
        inventory.put(product.getProductId(), product);
    } else {
        System.out.println("Product not found in the inventory.");
    }
    // tc: O(1)
}

public void deleteProduct(String productId) {
    inventory.remove(productId);
    // tc: O(1)
}


public Product getProduct(String productId) {
    return inventory.get(productId);
    // tc: O(1)
}

public void displayInventory() {
    for (Product product : inventory.values()) {
        System.out.println("ID: " + product.getProductId() +
                ", Name: " + product.getProductName() +
                ", Quantity: " + product.getQuantity() +
                ", Price: " + product.getPrice());
    }
}


public static void main(String[] args) {
    InventoryManagementSystem ims = new InventoryManagementSystem();
    Scanner scanner = new Scanner(System.in);
    boolean exit = false;

    while (!exit) {
        System.out.println("Inventory Management System:");
        System.out.println("1. Add Product");
        System.out.println("2. Update Product");
        System.out.println("3. Delete Product");
        System.out.println("4. Display Inventory");
        System.out.println("5. Exit");
        System.out.print("Choose an option: ");
        int choice = scanner.nextInt();
        scanner.nextLine();

        switch (choice) {
            case 1:
                System.out.print("Enter product ID: ");
                String addId = scanner.nextLine();
                System.out.print("Enter product name: ");
                String addName = scanner.nextLine();
                System.out.print("Enter quantity: ");
                int addQuantity = scanner.nextInt();
                System.out.print("Enter price: ");
                double addPrice = scanner.nextDouble();
                scanner.nextLine();
                Product newProduct = new Product(addId, addName, addQuantity, addPrice);
                ims.addProduct(newProduct);
                System.out.println("Product added successfully.");
                break;
            case 2:
                System.out.print("Enter product ID to update: ");
                String updateId = scanner.nextLine();
                System.out.print("Enter new product name: ");
                String updateName = scanner.nextLine();
                System.out.print("Enter new quantity: ");
                int updateQuantity = scanner.nextInt();
                System.out.print("Enter new price: ");
                double updatePrice = scanner.nextDouble();
                scanner.nextLine();
                Product updatedProduct = new Product(updateId, updateName, updateQuantity,
updatePrice);
                ims.updateProduct(updatedProduct);
                System.out.println("Product updated successfully.");
                break;
            case 3:
                System.out.print("Enter product ID to delete: ");
                String deleteId = scanner.nextLine();
                ims.deleteProduct(deleteId);
                System.out.println("Product deleted successfully.");
                break;
            case 4:
                System.out.println("Displaying all products:");
                ims.displayInventory();
                break;
            case 5:
                exit = true;
                System.out.println("Exiting Inventory Management System.");
                break;
            default:
                System.out.println("Invalid option. Please try again.");
          }
        }

    }
}
```

4. Analysis

Time Complexity Analysis

Add Product (addProduct): O(1) - Insertion in a HashMap is average O(1) due to hashing.

Update Product (updateProduct): O(1) - Updating a value in a HashMap is average O(1).

Delete Product (deleteProduct): O(1) - Deletion in a HashMap is average O(1).

Get Product (getProduct): O(1) - Lookup in a HashMap is average O(1).

Optimization Discussion

To optimize the operations:

Handling Collisions in HashMap: Ensure that the hash function used provides a good distribution of hash codes to minimize collisions.

Resizing Strategy: Implement a strategy for resizing the HashMap when the load factor exceeds a certain threshold, to maintain efficient operation.

Concurrency: If the system needs to handle concurrent access, consider using a concurrent variant of HashMap (like ConcurrentHashMap in Java) to prevent race conditions.