**Name : Madhavakumar D**

**Registered Email ID : madhavakumar.d@gmail.com**

**Superset ID : 5018705**

**Exercise 1: Inventory Management System**

**Scenario:**

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

**Steps:**

1. **Understand the Problem:**
   * Explain why data structures and algorithms are essential in handling large inventories.
   * Discuss the types of data structures suitable for this problem.
2. **Setup:**
   * Create a new project for the inventory management system.
3. **Implementation:**
   * Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.
   * Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
   * Implement methods to add, update, and delete products from the inventory.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
   * Discuss how you can optimize these operations.

**Solution :**

1. **Understanding :**

* **Efficient Storage and Retrieval**: Proper data structures enable efficient storage and quick retrieval of product information, which is crucial for operations like searching for a product, updating product details, and managing stock levels.
* **Scalability**: As the size of the inventory grows, the system must scale efficiently. Suitable data structures and algorithms ensure that performance remains optimal even with large data sets.
* **Data Integrity**: Ensuring data consistency and integrity is vital in inventory management. Proper data structures help in maintaining accurate records and avoiding data corruption.
* **Operations Efficiency**: Common operations such as adding, updating, and deleting products should be performed in minimal time to ensure smooth operation of the warehouse.

#### Suitable Data Structures:

* **ArrayList**: Useful for maintaining a dynamic list of products. It provides quick access by index but has slower search times for large datasets.
* **HashMap**: Ideal for storing products with a unique product ID as the key. It allows for fast access, insertion, and deletion operations.
* **TreeMap**: Useful if you need to maintain a sorted order of products based on keys

1. **Setup and 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;

}

}

import java.util.HashMap;

import java.util.Map;

public class Inventory {

private Map<String, Product> products;

public Inventory() {

this.products = new HashMap<>();

}

public void addProduct(Product product) {

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

}

public void updateProduct(Product product) {

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

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

} else {

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

}

}

public void deleteProduct(String productId) {

if (products.containsKey(productId)) {

products.remove(productId);

} else {

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

}

}

public Product getProduct(String productId) {

return products.get(productId);

}

public void displayAllProducts() {

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

System.out.println("Product ID: " + product.getProductId() +

", Name: " + product.getProductName() +

", Quantity: " + product.getQuantity() +

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

}

}

public static void main(String[] args) {

Inventory inventory = new Inventory();

Product p1 = new Product("1", "Product A", 100, 10.991);

Product p2 = new Product("2", "Product B", 150, 12.991);

inventory.addProduct(p1);

inventory.addProduct(p2);

inventory.displayAllProducts();

p1.setPrice(11.99);

inventory.updateProduct(p1);

inventory.displayAllProducts();

inventory.deleteProduct("2");

inventory.displayAllProducts();

}

}

1. **Analysis:**

#### Time Complexity Analysis

* **Add Product**: **Time Complexity**: O(1) on average due to the efficient hash-based access provided by HashMap.
* **Update Product**: **Time Complexity**: O(1) on average since updating a value in a HashMap is also O(1).
* **Delete Product**: **Time Complexity**: O(1) on average as deletion in a HashMap is O(1).
* **Retrieve Product**: **Time Complexity**: O(1) on average due to hash-based access.

#### Optimization Discussion

* **Load Factor and Rehashing**: Ensure that the HashMap does not become too full to avoid excessive collisions, which can degrade performance. The default load factor of 0.75 is a good balance between time and space complexity.
* **Concurrent Access**: For a multi-threaded environment, consider using Concurrent HashMap instead of HashMap to handle concurrent modifications efficiently.
* **Indexing**: If product retrieval by attributes other than productId becomes necessary, consider additional indexing or using a more complex data structure like a combination of HashMap and TreeMap.