WEEK 1: Algorithms\_Data Structures

**Exercise 1: E-commerce Platform Search Function**

**1)**

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

Efficient data structures and algorithms are essential in inventory management because they enhance speed, increase memory efficiency and make code scalability. Warehouses often manage thousands of products and using optimized algorithms ensures that operations like adding, searching, and updating inventory records are performed quickly. Additionally, data structures help minimize memory usage, making the system more resource-efficient.

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

An ArrayList is useful for maintaining simple lists of products of small size and maintains order. However, it performs poorly when searching, updating, or deleting items by their Id, as these operations require linear time O(N). In contrast, a HashMap is more effective when quick access to products using a unique key like Id is needed. It allows for fast lookups, insertions, and deletions with an average time complexity of O(1), making it ideal for dynamic and large-scale inventory systems.

**2) CODE**

**Main.java**

public class Main {

public static void main(String[] args) {

Manager m = new Manager();

Product p1 = new Product("M3001", "Keypad", 10, 1500.0);

Product p2 = new Product("P2002", "Mouse", 50, 25.0);

m.addP(p1);

m.addP(p2);

m.listP();

m.updateP("M3001", "Monitor", 40, 30.0);

m.deleteP("P2002");

m.deleteP("Q3002");

m.listP();

}

}

**Manager.java**

import java.util.HashMap;

public class Manager {

private HashMap<String, Product> inventory;

public Manager() {

inventory = new HashMap<>();

}

public void addP(Product product) {

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

System.out.println("Product added: " + product);

}

public void updateP(String productId, String name, int quantity, double price) {

Product product = inventory.get(productId);

if (product != null) {

product.setProductName(name);

product.setQuantity(quantity);

product.setPrice(price);

System.out.println("Product updated: " + product);

} else {

System.out.println("Not found.");

}

}

public void deleteP(String productId) {

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

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

} else {

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

}

}

public void listP() {

System.out.println("\nInventory List");

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

System.out.println(product);

}

System.out.println("\n");

}

}

**Product.java**

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;

}

public String getProductId() { return productId; }

public String getProductName() { return productName; }

public int getQuantity() { return quantity; }

public double getPrice() { return price; }

public void setProductName(String productName) { this.productName = productName; }

public void setQuantity(int quantity) { this.quantity = quantity; }

public void setPrice(double price) { this.price = price; }

@Override

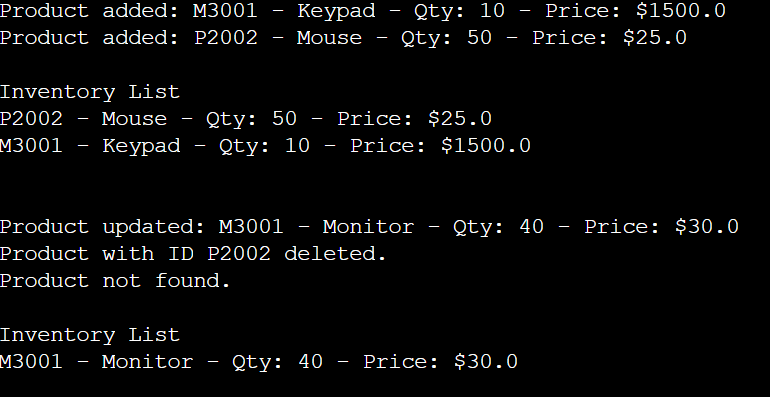
public String toString() {

return productId + " - " + productName + " - Qty: " + quantity + " - Price: $" + price;

}

}

**Output**

****

**3) Analysis**

**Time complexity**

Add product - O(1)

Update product - O(1)

Delete product - O(1)

List all products - O(n)

**Exercise 7: Financial Forecasting**

**1) Explain the concept of recursion and how it can simplify certain problems.**

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. Each recursive call works on a simpler or smaller version of the original problem until it reaches a **base case**, which stops the recursion. Recursion can be used when a problem can be broken down into subproblems of the same type.

**2) CODE**

**Main.java**

public class Main

{ public static double pred(double principal, double rate, int years) {

if (years == 0) {

return principal;

}

return pred(principal, rate, years - 1) \* (1 + rate);

}

public static void main(String[] args) {

double principal = 1000.0;

double rate = 0.05;

int years = 10;

double result = pred(principal, rate, years);

System.out.printf("Presicted value for the future is "+ Math.round(result));

}

}

**Output**



**3) Analysis**

Time complexity - O(n) (n=No of years)

Optimization Techniques

* Memoization (Top-Down) - Storing intermediate results in an array to avoid recomputation
* Iterative (Bottom-up) - Avoids the call stack overhead