Week 1: Algorithms & Data Structures

**Exercise 1: Inventory System Management**

# Scenario:

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

# Understanding the problem

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

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

### 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.

### 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

### 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).

### 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.

# Setup

* Create a new Java project for the inventory management system using any IDE such as Visual Studio Code, Eclipse, or a simple text editor with terminal.
* Inside the project directory, create two Java classes: **Inventory.java** and **Product.java**.
* Compile both Java files and run the Inventory class which contains the main method to interact with the system.

# Implementation

## Inventory Class Definition:

## import java.util.HashMap;

## import java.util.Map;

## import java.util.Scanner;

## public class Inventory {

## private Map<Integer, Product> productMap;

## private Scanner sc;

## public Inventory() {

## productMap = new HashMap<>();

## sc = new Scanner(System.in);

## }

## public void addProduct() {

## char choice = 'y';

## while (choice == 'y' || choice == 'Y') {

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

## int id = sc.nextInt();

## sc.nextLine();

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

## String name = sc.nextLine();

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

## int qty = sc.nextInt();

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

## double price = sc.nextDouble();

## Product p = new Product(id, name, qty, price);

## productMap.put(id, p);

## System.out.println("Product added.");

## System.out.print("Add another product? (y/n): ");

## choice = sc.next().charAt(0);

## sc.nextLine();

## }

## }

## public void updateProduct() {

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

## int id = sc.nextInt();

## sc.nextLine();

## Product p = productMap.get(id);

## if (p != null) {

## System.out.print("Enter New Name: ");

## String name = sc.nextLine();

## System.out.print("Enter New Quantity: ");

## int qty = sc.nextInt();

## System.out.print("Enter New Price: ");

## double price = sc.nextDouble();

## p.setProductName(name);

## p.setQuantity(qty);

## p.setPrice(price);

## System.out.println("Product updated.");

## } else {

## System.out.println("No product found with ID: " + id);

## }

## }

## public void deleteProduct() {

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

## int id = sc.nextInt();

## if (productMap.remove(id) != null) {

## System.out.println("Product deleted.");

## } else {

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

## }

## }

## public void displayProducts() {

## if (productMap.isEmpty()) {

## System.out.println("Inventory is empty.");

## } else {

## System.out.println("Inventory:");

## for (Product p : productMap.values()) {

## System.out.println(p);

## }

## }

## }

## public void start() {

## boolean run = true;

## while (run) {

## System.out.print(

## "\n--- Menu ---\n1. Add Product  |  2. Update  |  3. Delete  |  4. View  |  5. Exit\nChoose (1-5): "

## );

## int opt = sc.nextInt();

## switch (opt) {

## case 1 -> addProduct();

## case 2 -> updateProduct();

## case 3 -> deleteProduct();

## case 4 -> displayProducts();

## case 5 -> {

## run = false;

## System.out.println("Exiting...");

## }

## default -> System.out.println("Invalid input. Try 1-5.");

## }

## }

## }

## public static void main(String[] args) {

## Inventory i = new Inventory();

## i.start();

## }

## }

## Product Class Definition:

## public class Product {

## private int id;

## private String name;

## private int quantity;

## private double price;

## public Product(int id, String name, int quantity, double price) {

## this.id = id;

## this.name = name;

## this.quantity = quantity;

## this.price = price;

## }

## public int getProductId() {

## return id;

## }

## public String getProductName() {

## return name;

## }

## public int getQuantity() {

## return quantity;

## }

## public double getPrice() {

## return price;

## }

## public void setProductName(String name) {

## this.name = name;

## }

## public void setQuantity(int quantity) {

## this.quantity = quantity;

## }

## public void setPrice(double price) {

## this.price = price;

## }

## @Override

## public String toString() {

## return String.format("ID: %d | Name: %s | Qty: %d | Price: %.2f", id, name, quantity, price);

## }

## }

## Data Structures Used:

To manage the inventory, a **HashMap<Integer, Product>** is used. The productId is used as the key, and the Product object is the value. This allows:

* Fast lookups for updating or deleting specific products
* Unique identification of each product
* Efficient storage and access using keys

## Core Functionalities Implemented:

### Add Product: The user enters details for a new product. A Product object is created and added to the HashMap with its ID as the key. After each entry, the user is prompted whether they want to add another product.

### Update Product: The product is searched using its ID. If found, the product name, quantity, and price can be updated using user input.

### Delete Product: The product is removed from the map using its ID. If the ID does not exist, a message is displayed.

# Analysis

## Time Complexity Analysis:

### Add Operation: Inserting a new product into the HashMap using put() takes O(1) time on average. This is because hashing allows direct access to the storage location based on the key (product ID).

### Update Operation: Updating a product involves retrieving it using get() and modifying its attributes. The retrieval is O(1) on average, so the total time for the update is also O(1).

### Delete Operation: Removing a product using remove() takes O(1) time on average, as it directly accesses the product using the key and deletes it.

## Optimization Discussion:

Using a HashMap significantly improves the efficiency compared to an ArrayList. In an ArrayList, operations like update or delete require a linear search, resulting in **O(n)** time complexity. By switching to a HashMap, all major operations add, update, delete are optimized to **O(1)** on average.

A screenshot of a computer

AI-generated content may be incorrect.**Output**

Week 1: Algorithms & Data Structures

**Exercise 2: E-commerce Platform Search Function**

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

# Understanding the Asymptotic Notations

* **Big O Notation**: Describes the upper bound of an algorithm’s running time or space complexity, helping us understand how the algorithm scales with input size.
* **Best-case**: Minimum time required (example: In linear search, the item found at first position).

**Average-case**: Expected time over all possible scenarios. (example: In linear search, the item is found in the middle of the array or near to that position)

**Worst-case**: Maximum time required (example: In linear Search, the item not found at all, or in the last position).

# Setup

# Create a new Java project using any IDE or a basic text editor with terminal. Inside the project directory, create two java classes: Product.java and SearchDemo.java. In Product.java, define attributes such as:

# productId (int): A unique identifier

# productName (String): The name of the product

# category (String): The category the product belongs to

# Include a constructor, getter/setter methods if required, and a toString() method for displaying product details.

**Implementation**

public class ProductSearch {

public static Product linearSearch(Product[] products, int id) {

for (Product p : products) {

if (p.id == id)

return p;

}

return null;

}

public static Product binarySearch(Product[] products, int id) {

int left = 0, right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

if (products[mid].id == id)

return products[mid];

else if (products[mid].id < id)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

class Product {

    int id;

    String name;

    String category;

    public Product(int id, String name, String category) {

        this.id = id;

        this.name = name;

        this.category = category;

    }

    @Override

    public String toString() {

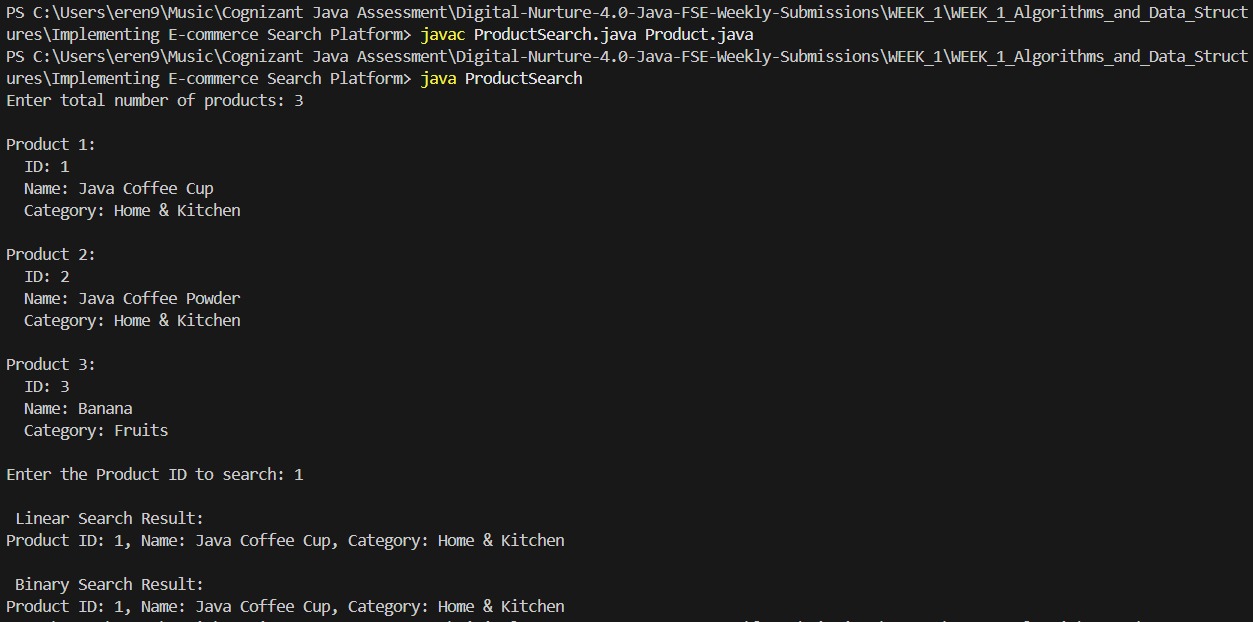
        return "Product ID: " + id + ", Name: " + name + ", Category: " + category;

    }

}

**Analysis**

* **Linear Search** checks each element in the list one by one until it finds the target or reaches the end. It works on both sorted and unsorted data. Its time complexity is **O(n)**, meaning it can be slow for large datasets. However, it's simple to implement and useful when data isn't sorted.
* **Binary Search**, on the other hand, is much faster with a time complexity of **O(log n)**, but it only works on sorted data. It works by dividing the array into halves and narrowing the search range with each step. This makes it efficient for large datasets, but it requires extra steps to ensure the data is sorted before searching.
* In summary, linear search is better for small or unsorted datasets due to its simplicity, while binary search is more suitable for large and sorted datasets because of its speed.

**Output**

Week 1: Algorithms & Data Structures

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

# Understanding Recursive Algorithms

# Recursion is a method where a function calls itself to solve smaller instances of a problem. It's especially useful for problems that can be broken down into smaller, similar subproblems. In financial forecasting, recursion can simplify the calculation of future values by applying the same growth logic repeatedly.

# Setup

# We define a method that uses recursion to compute the future value of an investment based on:

# The initial amount

# The annual growth rate

# The number of years

**Implementation**

import java.util.Scanner;

**//recursive method to calculate future value**

public class FinForecast {

public static double futureValue(double principal, double rate, int years) { //using recursion to calcute future value

if (years == 0) {

return principal;

}

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

**//using the same method to calculate the future value**

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the initial amount: ");

double principal = scanner.nextDouble();

System.out.print("Enter the annual growth rate (as a decimal): ");

double rate = scanner.nextDouble();

System.out.print("Enter the number of years: ");

int years = scanner.nextInt();

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

System.out.printf("Future Value : %.2f\n", result);

}

}

**Analysis**

**Time Complexity:**

* The time complexity of the recursive futureValue() method is O(n), where n is the number of years. For each year, a recursive call is made until the base case is reached.

**Optimization:**

* In this simple recursion, the number of calls grows linearly with years. However, if you had overlapping subproblems (like in Fibonacci), it could lead to exponential time complexity. In such cases, memoization or dynamic programming would help avoid repeated computation.
* For this problem, since each year's value depends directly and only on the previous year's, the recursion is efficient and doesn’t need further optimization.

A black screen with white text

AI-generated content may be incorrect.**Output**