**WEEK 1 ASSIGNMENT**

**Algorithms and Data Structures**

**- Aritra Das**

**Superset ID: 6362294**

**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:**

* **Why Data Structures and Algorithms Are Essential for Handling Large Inventories:**

In a warehouse environment, managing a large volume of inventory efficiently is critical. Efficient **data structures** and **algorithms** ensure that operations like adding new products, updating stock levels, or deleting outdated items are performed quickly, even as the size of the inventory grows.

Without proper structures, searching through thousands of items linearly would become slow and inefficient. Instead, using optimized structures (like hash tables) allows constant-time operations regardless of inventory size.

* **Suitable Data Structures for This Problem:**
  + **HashMap** (used in the code): Ideal for scenarios where quick access by a unique identifier (like productId) is required.
    - **Advantages**:
      * Fast lookup, insertion, and deletion (O(1) average time complexity).
      * Allows mapping product IDs to product objects directly.
* **Alternative options** (not used, but viable in other contexts):
  + **ArrayList**: Useful for ordered lists but slower for searching/updating by ID (O(n)).
  + **TreeMap**: Useful if sorted order is needed by product ID, with O(log n) performance.

1. **Code:**

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

InventoryService inventory = new InventoryService();

Scanner scanner = new Scanner(System.in);

// Sample Products

inventory.addProduct(new Product(1, "Laptop", 18, 1350.00));

inventory.addProduct(new Product(2, "Phone", 12, 900.00));

System.out.println("Inventory Management System");

System.out.println("---------------------------");

while (true) {

System.out.println("\nChoose an option:\n1. Add Product\n2. Update Product\n3. Delete Product\n4. Display Inventory\n5. Exit");

int choice = scanner.nextInt();

scanner.nextLine();

switch (choice) {

case 1:

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

int id = scanner.nextInt();

scanner.nextLine();

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

String name = scanner.nextLine();

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

int qty = scanner.nextInt();

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

double price = scanner.nextDouble();

inventory.addProduct(new Product(id, name, qty, price));

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

break;

case 2:

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

id = scanner.nextInt();

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

qty = scanner.nextInt();

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

price = scanner.nextDouble();

if (inventory.updateProduct(id, qty, price)) {

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

} else {

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

}

break;

case 3:

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

id = scanner.nextInt();

if (inventory.deleteProduct(id)) {

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

} else {

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

}

break;

case 4:

inventory.displayInventory();

break;

case 5:

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

scanner.close();

return;

default:

System.out.println("Invalid choice.");

}

}

}

}

// Product class

class Product {

private int productId;

private String productName;

private int quantity;

private double price;

public Product(int productId, String productName, int quantity, double price) {

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public int getProductId() { return productId; }

public String getProductName() { return productName; }

public int getQuantity() { return quantity; }

public double getPrice() { return price; }

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

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

@Override

public String toString() {

return "Product ID: " + productId + ", Name: " + productName +

", Quantity: " + quantity + ", Price: $" + price;

}

}

// InventoryService class

class InventoryService {

private Map<Integer, Product> inventory = new HashMap<>();

public void addProduct(Product product) {

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

}

public boolean updateProduct(int productId, int newQuantity, double newPrice) {

Product product = inventory.get(productId);

if (product != null) {

product.setQuantity(newQuantity);

product.setPrice(newPrice);

return true;

}

return false;

}

public boolean deleteProduct(int productId) {

return inventory.remove(productId) != null;

}

public void displayInventory() {

if (inventory.isEmpty()) {

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

return;

}

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

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

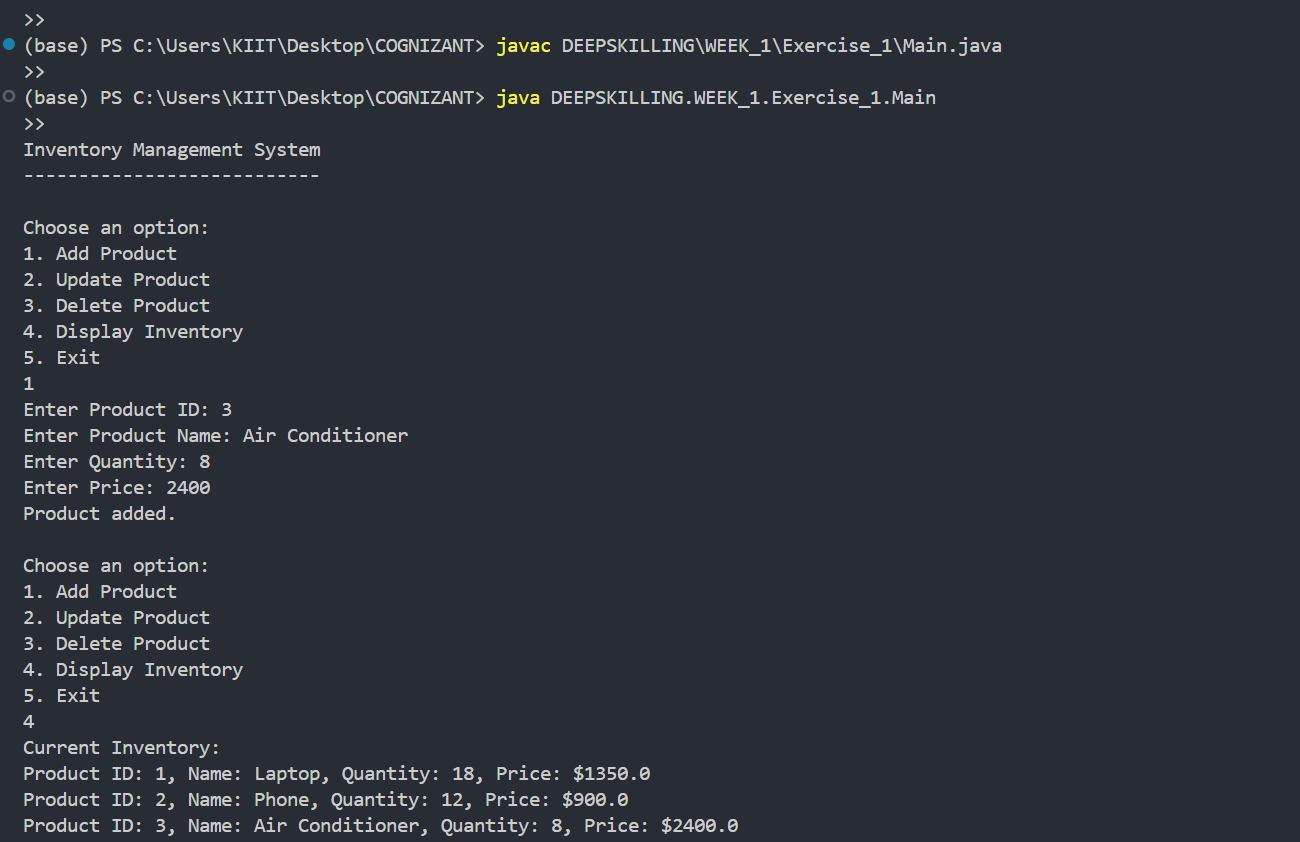
System.out.println(product);

}

}

}

1. **Output:**

****

1. **Analysis:**

**Time Complexity of Operations in HashMap:**

* **Add Product (addProduct)**:
  + Uses put() method of HashMap.
  + **Time Complexity:** O(1) on average (amortized), due to hashing.
* **Update Product (updateProduct)**:
  + Uses get() to retrieve the product and set methods to modify fields.
  + **Time Complexity:** O(1) for get() + constant time updates = O(1) average.
* **Delete Product (deleteProduct)**:
  + Uses remove() method of HashMap.
  + **Time Complexity:** O(1) average.
* **Display Inventory (displayInventory)**:
  + Iterates over values() of the HashMap.
  + **Time Complexity:** O(n) where n is the number of products.

**Optimizations:**

* The current use of HashMap is already optimal for basic CRUD operations.
* Further optimizations could include:
  + **Input validation** to prevent duplicate productIds during addition..  
    **Indexing or secondary maps** if search by name or price range is needed.

**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.

**1. Understand Asymptotic Notation**

* **What is Big O Notation?**  
  Big O notation is a mathematical representation used to describe the upper bound of an algorithm’s runtime or space complexity in terms of input size (**n**). It helps evaluate how the performance of an algorithm scales as the size of the data increases.
* **Why it's useful:**  
  • Provides a way to compare algorithms independently of hardware.  
  • Helps identify bottlenecks in code.  
  • Categorizes algorithms by efficiency (e.g., O(1), O(log n), O(n), O(n log n), O(n²)).
* **Best, Average, and Worst-Case Scenarios for Search Operations:**
  + **Linear Search:**
* Best Case: O(1) — target is the first item.
* Average Case: O(n/2) → O(n).
* Worst Case: O(n) — target is at the end or not present.
* **Binary Search (on sorted data):**
* Best Case: O(1) — target is the middle item.
* Average Case: O(log n).
* Worst Case: O(log n) — repeatedly divide the array until the target is found or confirmed absent.

**2. Code**

package DEEPSKILLING.WEEK\_1.Exercise\_2;

import java.util.Arrays;

import java.util.Scanner;

*// Product class*

class Product {

    int productId;

    String productName;

    String category;

    public Product(int productId, String productName, String category) {

        this.productId = productId;

        this.productName = productName;

        this.category = category;

    }

    @Override

    public String toString() {

        return "ID: " + productId + ", Name: " + productName + ", Category: " + category;

    }

}

public class SearchDemo {

    public static void main(String[] args) {

        Product[] products = {

            new Product(101, "Laptop", "Electronics"),

            new Product(102, "Shirt", "Clothing"),

            new Product(103, "Book", "Stationery"),

            new Product(104, "Phone", "Electronics"),

            new Product(105, "Shoes", "Footwear")

        };

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter product name to search: ");

        String target = scanner.nextLine();

*// Linear Search*

        System.out.println("\nUsing Linear Search:");

        linearSearch(products, target);

*// Binary Search (requires sorted array by productName)*

        Arrays.sort(products, (a, b) -> a.productName.compareToIgnoreCase(b.productName));

        System.out.println("\nUsing Binary Search:");

        binarySearch(products, target);

        scanner.close();

    }

*// Linear Search*

    public static void linearSearch(Product[] products, String target) {

        for (Product p : products) {

            if (p.productName.equalsIgnoreCase(target)) {

                System.out.println("Found: " + p);

                return;

            }

        }

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

    }

*// Binary Search*

    public static void binarySearch(Product[] products, String target) {

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

        while (left <= right) {

            int mid = (left + right) / 2;

            int comparison = products[mid].productName.compareToIgnoreCase(target);

            if (comparison == 0) {

                System.out.println("Found: " + products[mid]);

                return;

            } else if (comparison < 0) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

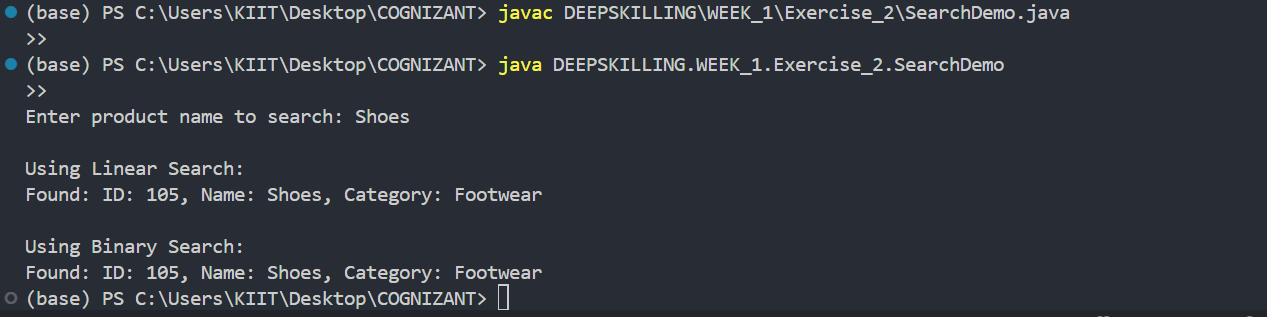
        }

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

    }

}

**3. Output**

****

**4. Analysis**

* **Time Complexity Comparison:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Linear Search** | **Binary Search (Sorted)** |  |
| **Best Case** | O(1) | O(1) |  |
| **Average Case** | O(n) | O(log n) |  |
| **Worst Case** | O(n) | O(log n) |  |

* **Which Algorithm Is More Suitable and Why?**
* **Binary Search** is significantly more efficient than Linear Search for large datasets — but only if the data is **sorted**.
* For unsorted data or real-time dynamic lists, **Linear Search** may be more appropriate without the overhead of sorting.
* **Conclusion:**
* For a live e-commerce platform where products are sorted alphabetically or by ID, **Binary Search** is optimal.
* If the dataset is small or unsorted, **Linear Search** may suffice temporarily.

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**1. Understand Recursive Algorithms**

* **What is Recursion?**
* **Recursion** is a programming technique where a method calls itself to solve smaller instances of the same problem.
* It is especially useful in problems that exhibit a repetitive or self-similar pattern, such as mathematical computations, tree traversal, or financial predictions.
* **Benefits of Recursion:  
  •** Simplifies code by breaking a problem into sub-problems. **•** Natural fit for divide-and-conquer strategies. **•** Makes complex formulas or logic easier to understand.

**2. Code**

package DEEPSKILLING.WEEK\_1.Exercise\_7;

public class FInancialForecast {

    public static double predictFutureValue(double initialAmount, double growthRate, int years) {

        if (years == 0) {

            return initialAmount;

        }

        return (1 + growthRate) \* predictFutureValue(initialAmount, growthRate, years - 1);

    }

    public static void main(String[] args) {

        double initialAmount = 16000.0;

        double annualGrowthRate = 0.98;

        int years = 6;

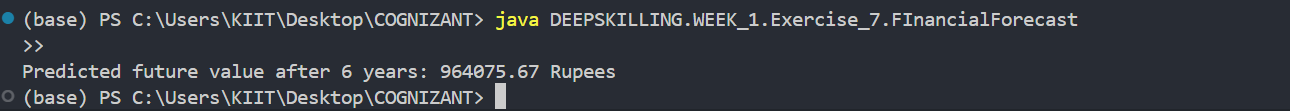
        double futureValue = predictFutureValue(initialAmount, annualGrowthRate, years);

        System.out.printf("Predicted future value after %d years: %.2f Rupees\n", years, futureValue);

    }

}

**3. Output**

****

**4. Analysis**

* **Time Complexity of the Recursive Algorithm:**
* **Time Complexity: O(n)**
  + The recursive method **predictFutureValue()** calls itself once per year.
  + For **n** years, it performs **n** recursive calls ⇒ **linear time**.
* **Space Complexity: O(n)**
  + Due to recursive call stack: each call waits for the next → stack grows to **n** depth.
* **Optimizations to Improve Efficiency:**

**1. Memoization (not essential here, but useful for expensive operations):**

* In more complex recursive financial models, storing intermediate results can avoid redundant calculations.

**2. Iterative Approach Alternative:**

* Use a loop to calculate compound interest:

**double result = initialAmount;**

**for (int i = 0; i < years; i++) {**

**result \*= (1 + growthRate);**

**}**

* This is more memory-efficient (O(1) space) and avoids deep recursion for large **n**.

**3. Use Mathematical Formula (Optimal):**

* Future Value = P × (1 + r)^n

**double futureValue = initialAmount \* Math.pow(1 + growthRate, years);**

* This is **O(1)** in both time and space — ideal when performance is critical.