**Exercise 1: Inventory Management System**

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

Data structures and algorithms are essential in handling large inventories because they enable:

* Efficient data management
* Fast data retrieval
* Automated inventory tracking and optimization
* Scalability and improved performance

This leads to increased efficiency, accuracy, and customer satisfaction, while reducing costs.

**Suitable Data Structures:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Structure** | **Use Case** | **Pros** | **Cons** |
| ArrayList | If order matters, few deletions | Fast iteration | Slow lookup by ID |
| HashMap | When fast lookup by productId is needed | O(1) operations | Slightly more memory usage |
| TreeMap | If sorted productId list is needed | Sorted access | O(log n) operations |

**Code:**

**//Product.java**

public 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 setProductId(int productId) {

this.productId = productId;

}

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 "Product{" +

"productId=" + productId +

", productName='" + productName + '\'' +

", quantity=" + quantity +

", price=" + price +

'}';

}

}

**//InventoryManager.java**

import java.util.\*;

public class InventoryManager {

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

public void addProduct(Product product) {

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

}

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

Product product = inventory.get(productId);

if (product != null) {

product.setProductName(name);

product.setQuantity(quantity);

product.setPrice(price);

}

else {

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

}

}

public void deleteProduct(int productId) {

if (inventory.containsKey(productId)) {

inventory.remove(productId);

}

else {

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

}

}

public void viewAllProducts() {

if (inventory.isEmpty()) {

System.out.println("No products in inventory.");

} else {

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

System.out.println(product);

}

}

}

}

**//App.java**

public class App {

public static void main(String[] args) throws Exception {

InventoryManager manager = new InventoryManager();

Product p1 = new Product(101, "Laptop", 50, 70000);

Product p2 = new Product(102, "Mouse", 200, 500);

manager.addProduct(p1);

manager.addProduct(p2);

manager.viewAllProducts();

manager.updateProduct(101, "Gaming Laptop", 40, 85000);

manager.viewAllProducts();

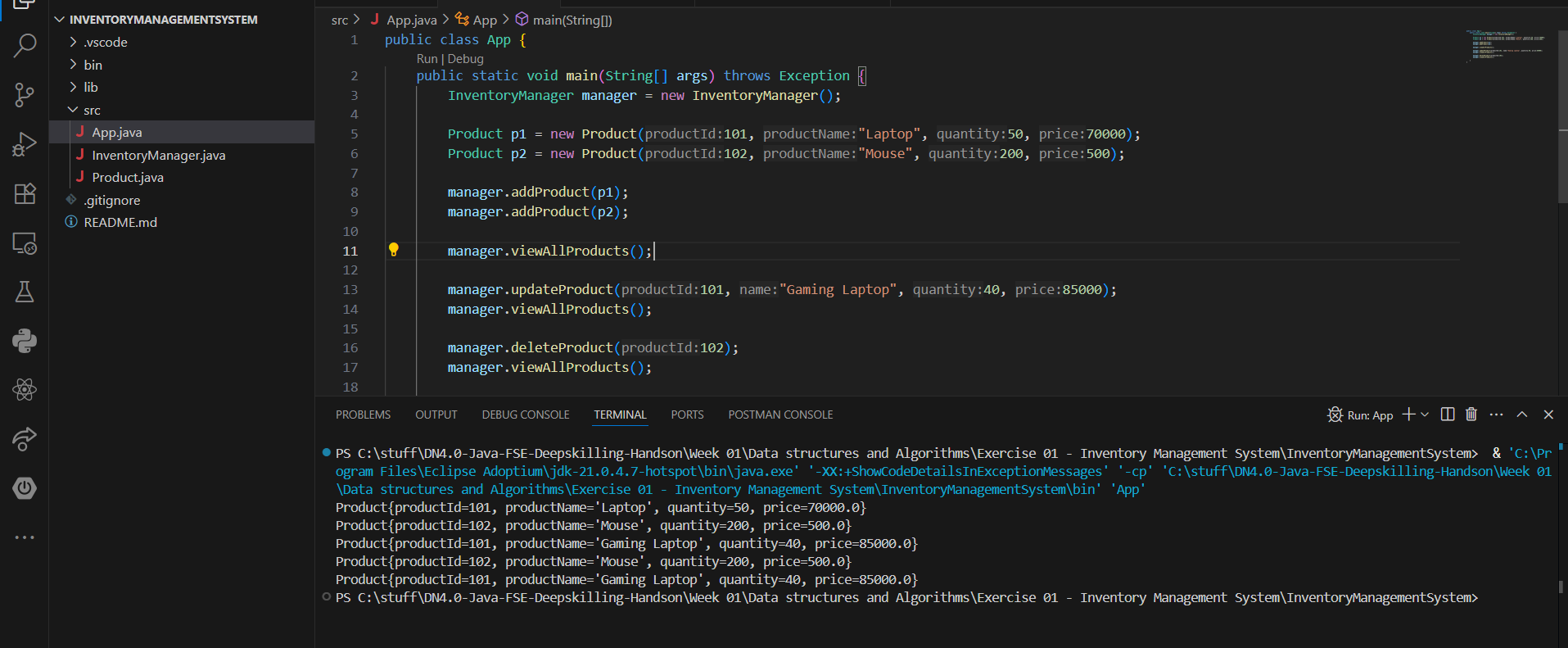
manager.deleteProduct(102);

manager.viewAllProducts();

}

}

**Output:**

****

**Analysis:**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Complexity** | **Reason** |
| addProduct() | O(1) | HashMap insertion |
| updateProduct() | O(1) | HashMap access and update |
| deleteProduct() | O(1) | HashMap deletion |
| viewAllProducts() | O(n) | Iterating over all products |

**Optimization:**

* Use TreeMap if **sorted order of productId** is important.
* For **frequent search by product name**, use an additional HashMap<String, List<Product>>

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

**Big O Notation:**

Big O notation describes the upper bound of an algorithm’s running time as the input size grows. It helps you understand how well your code will perform as your dataset scales.

* O(1) – Constant time
* O(log n) – Logarithmic time
* O(n) – Linear time
* O(n log n) – Linearithmic time
* O(n²) – Quadratic time

**Best, Average, and Worst-Case:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| **Linear Search** | O(1) (found early) | O(n/2) ≈ O(n) | O(n) |
| **Binary Search** | O(1) (found at mid) | O(log n) | O(log n) |

Binary search is faster but requires a sorted array.

**Code:**

**//Product.java**

public class Product {

private int productId;

private String productName;

private String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

}

**//** **SearchUtility.java**

public class SearchUtility {

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

for (Product product : products) {

if (product.getProductId() == productId) {

return product;

}

}

return null;

}

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

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

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

**//App.java**

import java.util.Arrays;

import java.util.Comparator;

public class App {

public static void main(String[] args) throws Exception {

Product[] products = {

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

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

new Product(103, "Phone", "Electronics"),

new Product(104, "Shoes", "Footwear"),

new Product(105, "Watch", "Accessories" )

};

Product foundProduct = SearchUtility.linearSearch(products, "Shoes");

if (foundProduct != null) {

System.out.println("Linear Search Found: " + foundProduct.getProductName() + " in " + foundProduct.getCategory());

}

else {

System.out.println("Product not found using Linear Search.");

}

foundProduct = SearchUtility.linearSearch(products, "Laptop");

if (foundProduct != null) {

System.out.println("Linear Search Found: " + foundProduct.getProductName() + " in " + foundProduct.getCategory());

}

else {

System.out.println("Product not found using Linear Search.");

}

Arrays.sort(products, Comparator.comparing(Product::getProductName, String.CASE\_INSENSITIVE\_ORDER));

foundProduct = SearchUtility.binarySearch(products, "Shirt");

if (foundProduct != null) {

System.out.println("Binary Search Found: " + foundProduct.getProductName() + " in " + foundProduct.getCategory());

}

else {

System.out.println("Product not found using Binary Search.");

}

foundProduct = SearchUtility.binarySearch(products, "Headphones");

if (foundProduct != null) {

System.out.println("Binary Search Found: " + foundProduct.getProductName() + " in " + foundProduct.getCategory());

}

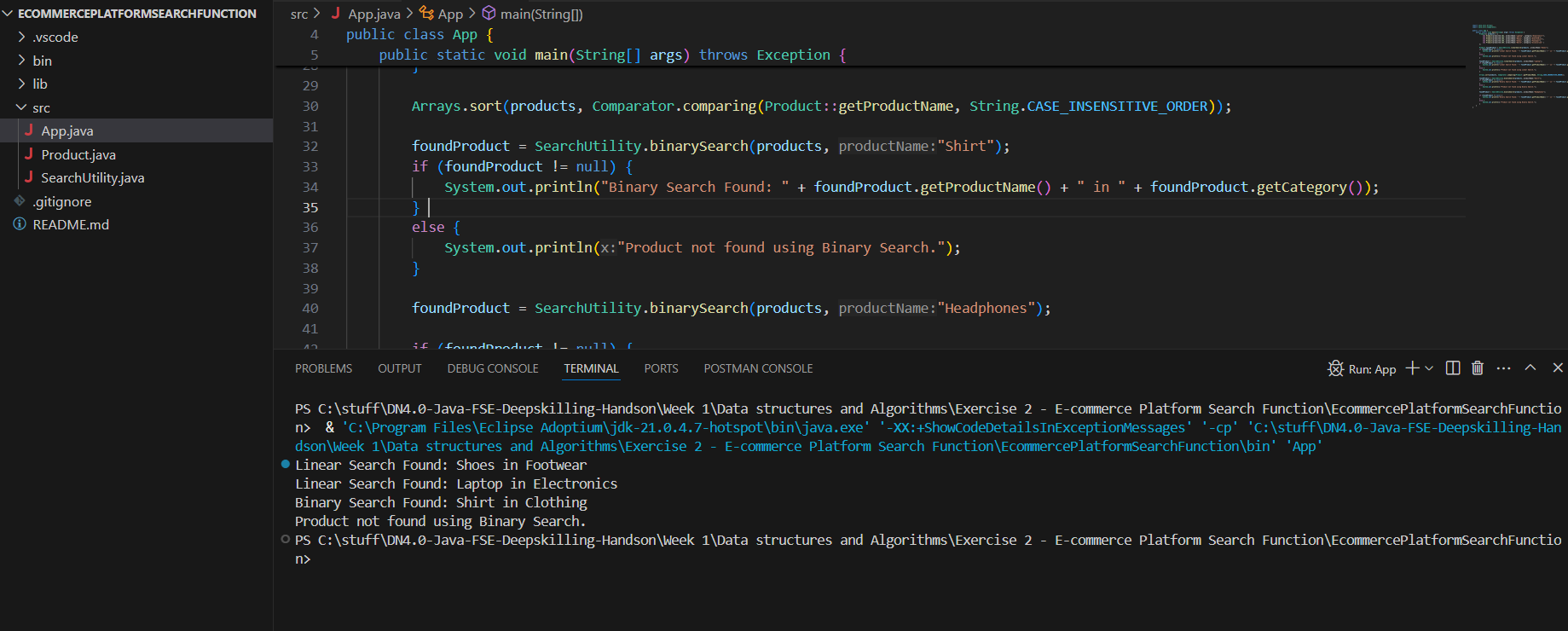
else {

System.out.println("Product not found using Binary Search.");

}

}

}

**Output:**

**Analysis:**

**Time Complexity:**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Data Requirement** |
| Linear Search | O(n) | Unsorted Array |
| Binary Search | O(log n) | Sorted Array |

**When to Use What?**

* Use linear search when:
  + The array is small.
  + Products are not sorted.
  + Real-time updates (insert/delete) are frequent.
* Use binary search when:
  + Data is large and relatively static.
  + You can afford to keep the array sorted.
  + You prioritize search speed.

**Exercise 3: Sorting Customer Orders**

**Sorting Algorithms:**

**Bubble Sort**

* Compares adjacent elements and swaps them if they are in the wrong order.
* Repeats until the list is sorted.
* Time Complexity:
* Best: O(n)
* Average/Worst: O(n²)
* Space Complexity: O(1) (in-place)
* Stable Sort: Yes

**Insertion Sort**

* Builds the sorted list one item at a time by comparing each new element to those already sorted.
* Time Complexity:
* Best: O(n)
* Average/Worst: O(n²)
* Space Complexity: O(1)
* Stable Sort: Yes

**Quick Sort**

* Divides the array using a pivot and sorts the subarrays recursively.
* Time Complexity:
* Best/Average: O(n log n)
* Worst: O(n²) (when pivot is poorly chosen)
* Space Complexity: O(log n) (for recursion stack)
* Stable Sort: No (but very efficient)

**Merge Sort**

* Divides the array into halves, sorts them, then merges the results.
* Time Complexity: Always O(n log n)
* Space Complexity: O(n)
* Stable Sort: Yes

**Code:**

**//Order.java**

public class Order {

int orderId;

String customerName;

double totalPrice;

public Order(int orderId, String customerName, double totalPrice) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

@Override

public String toString() {

return "Order{" +

"orderId=" + orderId +

", customerName='" + customerName + '\'' +

", totalPrice=" + totalPrice +

'}';

}

public static void bubbleSort(Order[] orders) {

int n = orders.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - 1 - i; j++) {

if (orders[j].totalPrice > orders[j + 1].totalPrice) {

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

}

}

}

}

public static void quickSort(Order[] orders, int low, int high) {

if (low < high) {

int pi = partition(orders, low, high);

quickSort(orders, low, pi - 1);

quickSort(orders, pi + 1, high);

}

}

private static int partition(Order[] orders, int low, int high) {

double pivot = orders[high].totalPrice;

int i = low - 1;

for (int j = low; j < high; j++) {

if (orders[j].totalPrice <= pivot) {

i++;

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

Order temp = orders[i + 1];

orders[i + 1] = orders[high];

orders[high] = temp;

return i + 1;

}

public static void main(String[] args) {

Order[] orders = {

new Order(1, "Alice", 350.75),

new Order(2, "Bob", 120.50),

new Order(3, "Charlie", 780.00),

new Order(4, "Diana", 450.20)

};

System.out.println("Original Orders:");

for (Order o : orders)

System.out.println(o);

bubbleSort(orders);

System.out.println("\nBubble Sorted Orders:");

for (Order o : orders)

System.out.println(o);

orders = new Order[]{

new Order(1, "Alice", 350.75),

new Order(2, "Bob", 120.50),

new Order(3, "Charlie", 780.00),

new Order(4, "Diana", 450.20)

};

quickSort(orders, 0, orders.length - 1);

System.out.println("\nQuick Sorted Orders:");

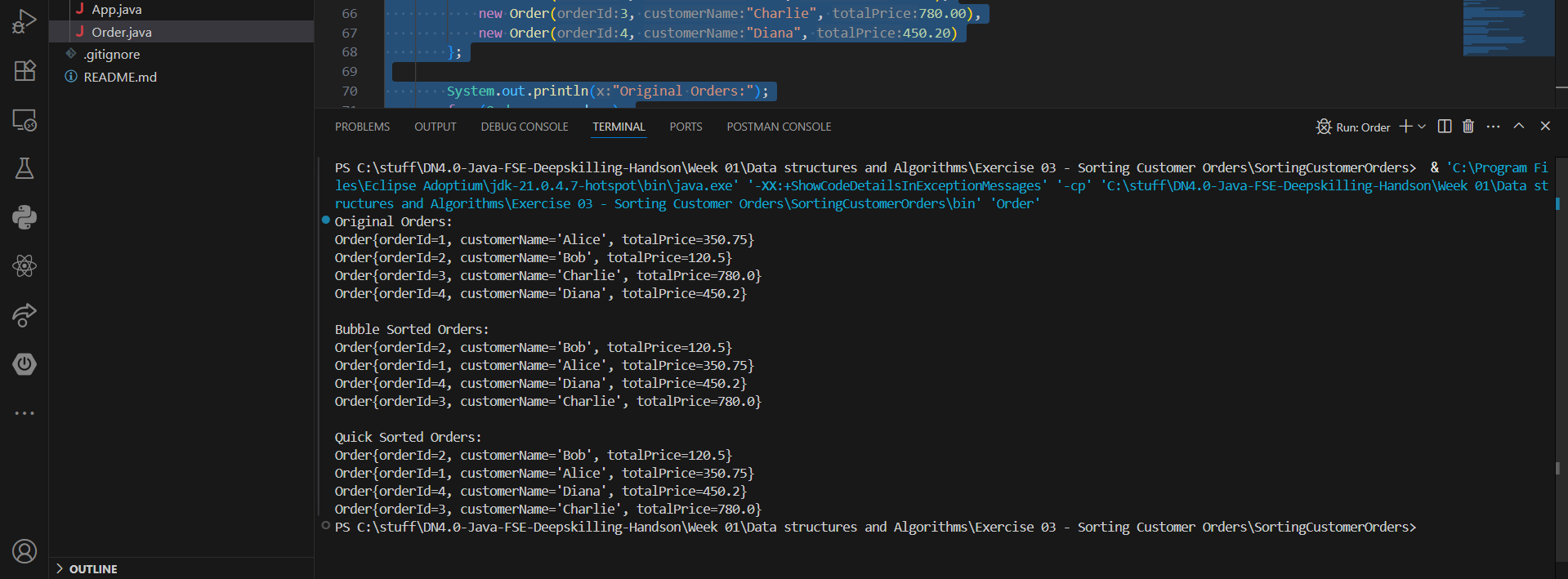
for (Order o : orders)

System.out.println(o);

}

}

**Output:**



**Analysis:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Best** | **Average** | **Worst** |
| Bubble Sort | O(n) | O(n²) | O(n²) |
| Quick Sort | O(n log n) | O(n log n) | O(n²) |

**Why Quick Sort is preferred:**

* Quick Sort is significantly faster on average due to better partitioning logic and less swapping.
* Though its worst-case is O(n²), with random pivoting or hybrid approaches, it's rarely encountered.
* Bubble Sort is simple but highly inefficient for large datasets.

**Exercise 4: Employee Management System**

**Array Representation:**

Arrays are contiguous memory locations where each element is stored sequentially.

Each element is accessible via an index, and the index is used to compute the memory address of any element using the formula: address = base\_address + (index × size\_of\_element)

**Advantages:**

* Fast access using indices: O(1) time for accessing any element.
* Cache-friendly: Due to contiguous memory storage.
* Simplicity: Easy to implement and use.

**Code**

**//Employee.java**

public class Employee {

int employeeId;

String name;

String position;

double salary;

public Employee(int employeeId, String name, String position, double salary) {

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

public void displayInfo() {

System.out.println("ID: " + employeeId + ", Name: " + name + ", Position: " + position + ", Salary: " + salary);

}

}

**//EmployeeManagementSystem.java**

public class EmployeeManagementSystem {

private Employee[] employees;

private int count;

public EmployeeManagementSystem(int capacity) {

employees = new Employee[capacity];

count = 0;

}

public void addEmployee(Employee emp) {

if (count < employees.length) {

employees[count++] = emp;

System.out.println("Employee added successfully.");

} else {

System.out.println("Employee list is full.");

}

}

public Employee searchEmployee(int id) {

for (int i = 0; i < count; i++) {

if (employees[i].employeeId == id) {

return employees[i];

}

}

return null;

}

public void traverseEmployees() {

if (count == 0) {

System.out.println("No employees found.");

}

for (int i = 0; i < count; i++) {

employees[i].displayInfo();

}

}

public void deleteEmployee(int id) {

boolean found = false;

for (int i = 0; i < count; i++) {

if (employees[i].employeeId == id) {

found = true;

// Shift remaining elements to the left

for (int j = i; j < count - 1; j++) {

employees[j] = employees[j + 1];

}

employees[--count] = null; // Remove last duplicate

System.out.println("Employee deleted successfully.");

break;

}

}

if (!found) {

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

}

}

}

**//App.java**

public class App {

public static void main(String[] args) throws Exception {

EmployeeManagementSystem ems = new EmployeeManagementSystem(5);

Employee emp1 = new Employee(101, "Alice", "Manager", 75000);

Employee emp2 = new Employee(102, "Bob", "Developer", 60000);

Employee emp3 = new Employee(103, "Charlie", "Designer", 55000);

ems.addEmployee(emp1);

ems.addEmployee(emp2);

ems.addEmployee(emp3);

System.out.println("\n--- All Employees ---");

ems.traverseEmployees();

System.out.println("\n--- Search Employee with ID 102 ---");

Employee result = ems.searchEmployee(102);

if (result != null) {

result.displayInfo();

} else {

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

}

System.out.println("\n--- Delete Employee with ID 102 ---");

ems.deleteEmployee(102);

System.out.println("\n--- Updated Employee List ---");

ems.traverseEmployees();

ems.addEmployee(new Employee(104, "David", "QA", 50000));

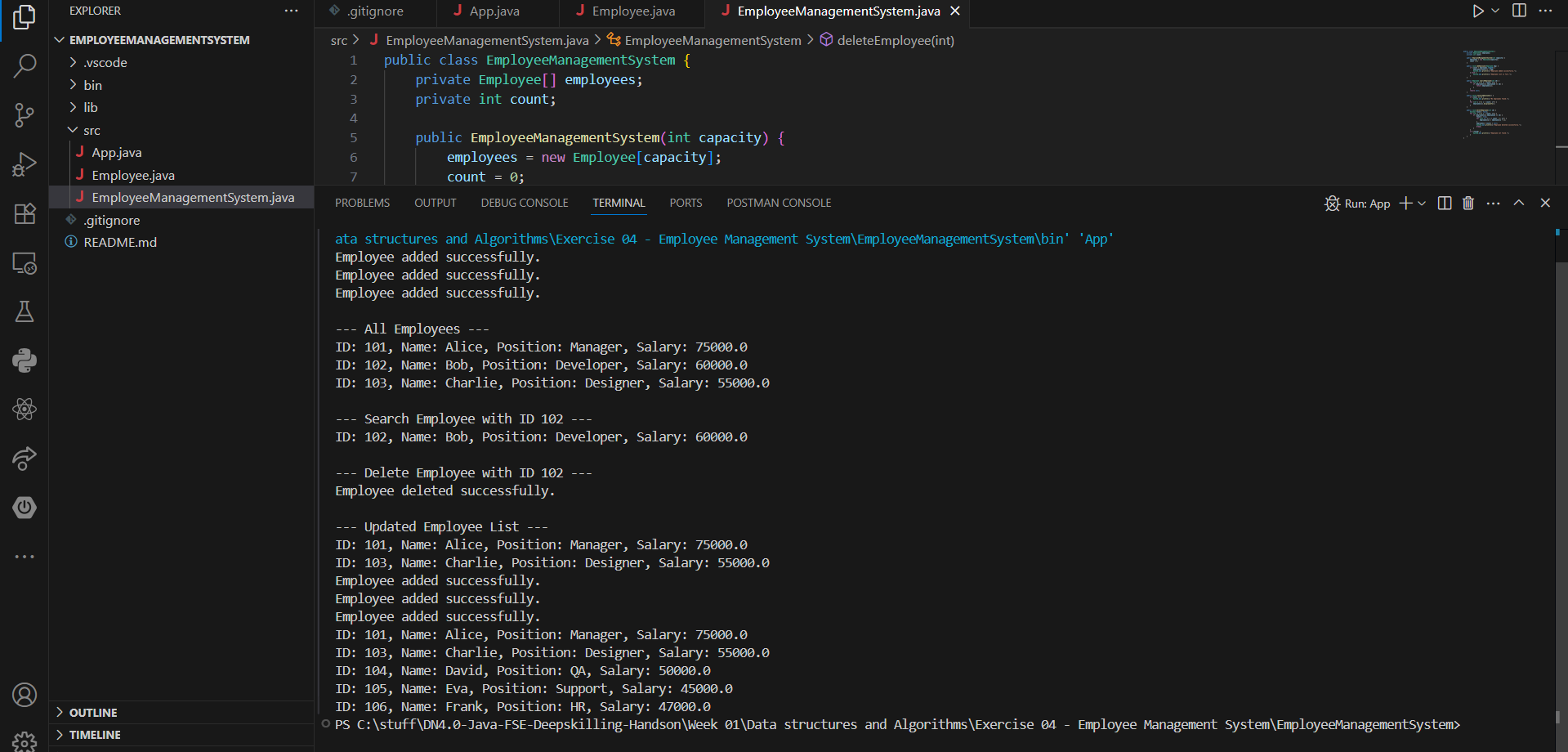
ems.addEmployee(new Employee(105, "Eva", "Support", 45000));

ems.addEmployee(new Employee(106, "Frank", "HR", 47000));

}

}

**Output:**

****

**Analysis:**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Time Complexity** | **Explanation** |
| Add | O(1) | Add at the end if not full |
| Search | O(n) | Linear search through the array |
| Traverse | O(n) | Go through each employee |
| Delete | O(n) | Shift elements after deletion |

**Limitations of Arrays:**

* Fixed Size: Must know the size in advance; can't dynamically grow or shrink.
* Costly Deletions: Shifting elements takes time (O(n)).
* Inefficient Insertions: Inserting in the middle requires shifting.

**When to use arrays:**

* When the number of elements is fixed or changes rarely.
* When fast random access is needed.
* When memory locality and performance are a priority.

**Exercise 5: Task Management System**

**Linked Lists:**

**Singly Linked List:**

* Each node points to the next node.
* Allows forward traversal only.
* Efficient in memory (less overhead than doubly).

**Doubly Linked List:**

* Each node has a pointer to both next and previous nodes.
* Allows bidirectional traversal.
* Slightly more memory usage due to extra pointer.

**Code:**

**//Task.java**

public class Task {

int taskId;

String taskName;

String status;

public Task(int taskId, String taskName, String status) {

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

@Override

public String toString() {

return "TaskID: " + taskId + ", Name: " + taskName + ", Status: " + status;

}

}

**//TaskNode.java**

class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

**//TaskManager.java**

public class TaskManager {

private TaskNode head;

public void addTask(Task task) {

TaskNode newNode = new TaskNode(task);

if (head == null) {

head = newNode;

} else {

TaskNode current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

System.out.println("Task added: " + task.taskName);

}

public void traverseTasks() {

TaskNode current = head;

if (current == null) {

System.out.println("No tasks found.");

return;

}

System.out.println("Task List:");

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

public Task searchTask(int taskId) {

TaskNode current = head;

while (current != null) {

if (current.task.taskId == taskId) {

return current.task;

}

current = current.next;

}

return null;

}

public void deleteTask(int taskId) {

if (head == null) return;

if (head.task.taskId == taskId) {

head = head.next;

System.out.println("Task with ID " + taskId + " deleted.");

return;

}

TaskNode current = head;

while (current.next != null && current.next.task.taskId != taskId) {

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

System.out.println("Task with ID " + taskId + " deleted.");

} else {

System.out.println("Task with ID " + taskId + " not found.");

}

}

}

**//App.java**

public class App {

public static void main(String[] args) {

TaskManager tm = new TaskManager();

tm.addTask(new Task(1, "Design UI", "Pending"));

tm.addTask(new Task(2, "Develop Backend", "In Progress"));

tm.addTask(new Task(3, "Test Features", "Pending"));

tm.traverseTasks();

System.out.println("\nSearching for Task ID 2:");

Task task = tm.searchTask(2);

System.out.println(task != null ? task : "Task not found");

System.out.println("\nDeleting Task ID 2:");

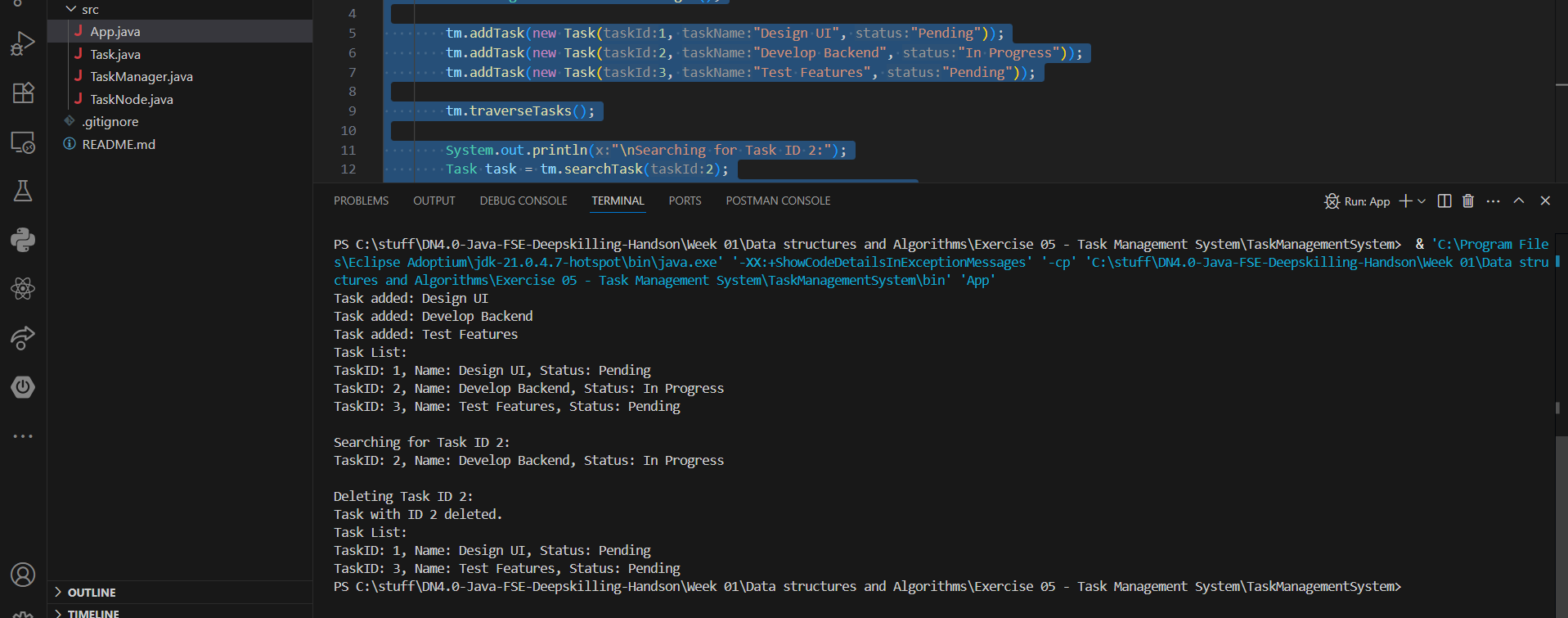
tm.deleteTask(2);

tm.traverseTasks();

}

}

**Output:**

****

**Analysis:**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Time Complexity** | **Reason** |
| Add Task | O(n) | Traverse to the end of list |
| Traverse | O(n) | Visit each node |
| Search Task | O(n) | Linear search required in singly linked list |
| Delete Task | O(n) | Need to find the node before the target node |

**Advantages of Linked Lists over Arrays for Dynamic Data:**

* Dynamic size adjustment: They can grow or shrink as elements are added or removed.
* Efficient insertion/deletion: Elements can be inserted or deleted without shifting all elements.
* Flexible memory use: Memory is allocated only for the elements in use.

**Exercise 7: Financial Forecasting**

**Recursion Concept:**

Recursion is a fundamental concept in programming where a function calls itself repeatedly until it reaches a base case that stops the recursion.

Key elements of recursion:

1. Base case: A condition that stops the recursion.
2. Recursive call: The function calls itself**.**

How recursion works:

1. The function is called with an initial input.
2. The function checks the base case. If true, it returns a value.
3. If not, the function calls itself with a modified input.
4. This process repeats until the base case is reached.

In Financial Forecasting:

We can recursively predict the **future value** using a fixed growth rate:

FVn​ = FVn−1 ​× (1+r)

Where:

* FVn = future value at year n
* r = annual growth rate
* Base case: FV0 = initial value

**Code:**

**//FinancialForecast.java**

import java.util.HashMap;

public class FinancialForecast {

public static double calculateFutureValue(double presentValue, double interestRate, int years) {

if(years==0) {

return presentValue;

}

return calculateFutureValue(presentValue, interestRate, years-1) \* (1 + interestRate);

}

public static double memoizedCalculateFutureValue(double presentValue, double interestRate, int years, HashMap<Integer, Double> memo) {

if(years == 0) {

return presentValue;

}

if (memo.containsKey(years)) {

return memo.get(years);

}

double futureValue = memoizedCalculateFutureValue(presentValue, interestRate, years-1, memo)\*(1+ interestRate);

memo.put(years, futureValue);

return futureValue;

}

}

**// App.java**

import java.util.HashMap;

public class App {

public static void main(String[] args) {

double presentValue = 6700.0;

double interestRate = 0.17;

int years = 10;

double futureValue = FinancialForecast.calculateFutureValue(presentValue, interestRate, years);

System.out.println("Future Value Calculation using basic Recursion:");

System.out.printf("Future Value after %d years: %.2f%n", years, futureValue);

HashMap <Integer, Double> memo = new HashMap<>();

futureValue = FinancialForecast.memoizedCalculateFutureValue(presentValue, interestRate, years, memo);

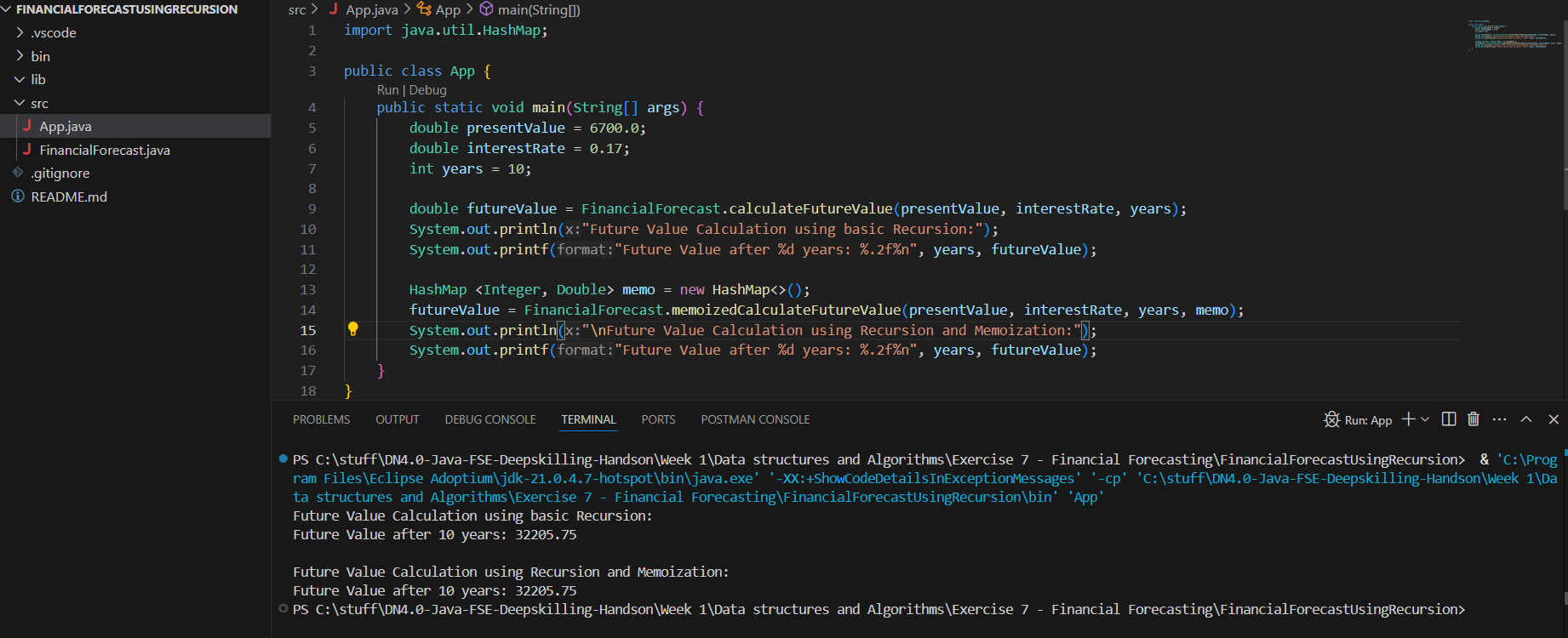
System.out.println("\nFuture Value Calculation using Recursion and Memoization:");

System.out.printf("Future Value after %d years: %.2f%n", years, futureValue);

}

}

**Output:**



**Analysis:**

**Time Complexity:**

Time Complexity: O(n) — since it makes a recursive call for each year.

Space Complexity: O(n) — due to function call stack.

**Optimization:**

Use Memoization (only useful for variable rates or overlapping subproblems): If growth rate changes per year, we can memoize the computed results.