**Exercise 1: Inventory Management System**

Q:Why data structures and algorithms are essential in handling large inventories?And types of data structures suitable for this problem.

A: Dealing with frequent activities like lookups, updates, and removals is a part of managing huge inventories. Poor data structure selection might result in scalability problems and slow performance. In a list of more than 10,000 entries, for example, linear search is far from effective. For these processes, algorithms and data structures assist guarantee the ideal time and space complexity.   
Inappropriate Data Structures:

-ArrayList Perfect for iteration and organized collections. Inadequate for frequent ID lookups.   
-With productId as the key, HashMap is ideal for constant-time access.   
- TreeMap: Although access time is O(log n), this is useful if you need to sort by productId.

-Since we frequently search for items by their ID, HashMap is the best option in this case.

InventoryManagementSystem project:

**Product.java:**

**package** Inventory;

**public** **class** Product {

**private** String productId;

**private** String name;

**private** **int** quantity;

**private** **double** price;

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

**this**.productId = productId;

**this**.name = name;

**this**.quantity = quantity;

**this**.price = price;

}

**public** String getProductId() {

**return** productId;

}

**public** String getName() {

**return** name;

}

**public** **int** getQuantity() {

**return** quantity;

}

**public** **double** getPrice() {

**return** price;

}

// Override toString for readable output

@Override

**public** String toString() {

**return** "Product ID: " + productId +

", Name: " + name +

", Quantity: " + quantity +

", Price: ₹" + price;

}

}

**Inventory.java:**

**package** Inventory;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Inventory {

**private** Map<String, Product> products;

**public** Inventory() {

products = **new** HashMap<>();

}

// This method adds a new product to the inventory

**public** **void** addProduct(Product product) {

String productId = product.getProductId();

**if** (products.containsKey(productId)) {

System.***out***.println("Product already exists with ID: " + productId);

**return**;

}

products.put(productId, product);

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

}

// This method updates an existing product

**public** **void** updateProduct(String productId, Product updatedProduct) {

**if** (!products.containsKey(productId)) {

System.***out***.println("Cannot update: No product found with ID " + productId);

**return**;

}

products.put(productId, updatedProduct);

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

}

// This method deletes a product from the inventory

**public** **void** deleteProduct(String productId) {

**if** (products.remove(productId) != **null**) {

System.***out***.println("Product removed: " + productId);

} **else** {

System.***out***.println("Product not found: " + productId);

}

}

// This method retrieves a product by ID

**public** Product getProduct(String productId) {

**if** (!products.containsKey(productId)) {

System.***out***.println("Product not found: " + productId);

**return** **null**;

}

**return** products.get(productId);

}

// This method displays all products in the inventory

**public** **void** listAllProducts() {

**if** (products.isEmpty()) {

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

**return**;

}

**for** (Product p : products.values()) {

System.***out***.println(p);

}

}

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

Inventory inventory = **new** Inventory();

Product p1 = **new** Product("P001", "Keyboard", 10, 799.00);

Product p2 = **new** Product("P002", "Mouse", 20, 499.00);

Product p3 = **new** Product("P003", "Monitor", 5, 9999.00);

inventory.addProduct(p1);

inventory.addProduct(p2);

inventory.addProduct(p3);

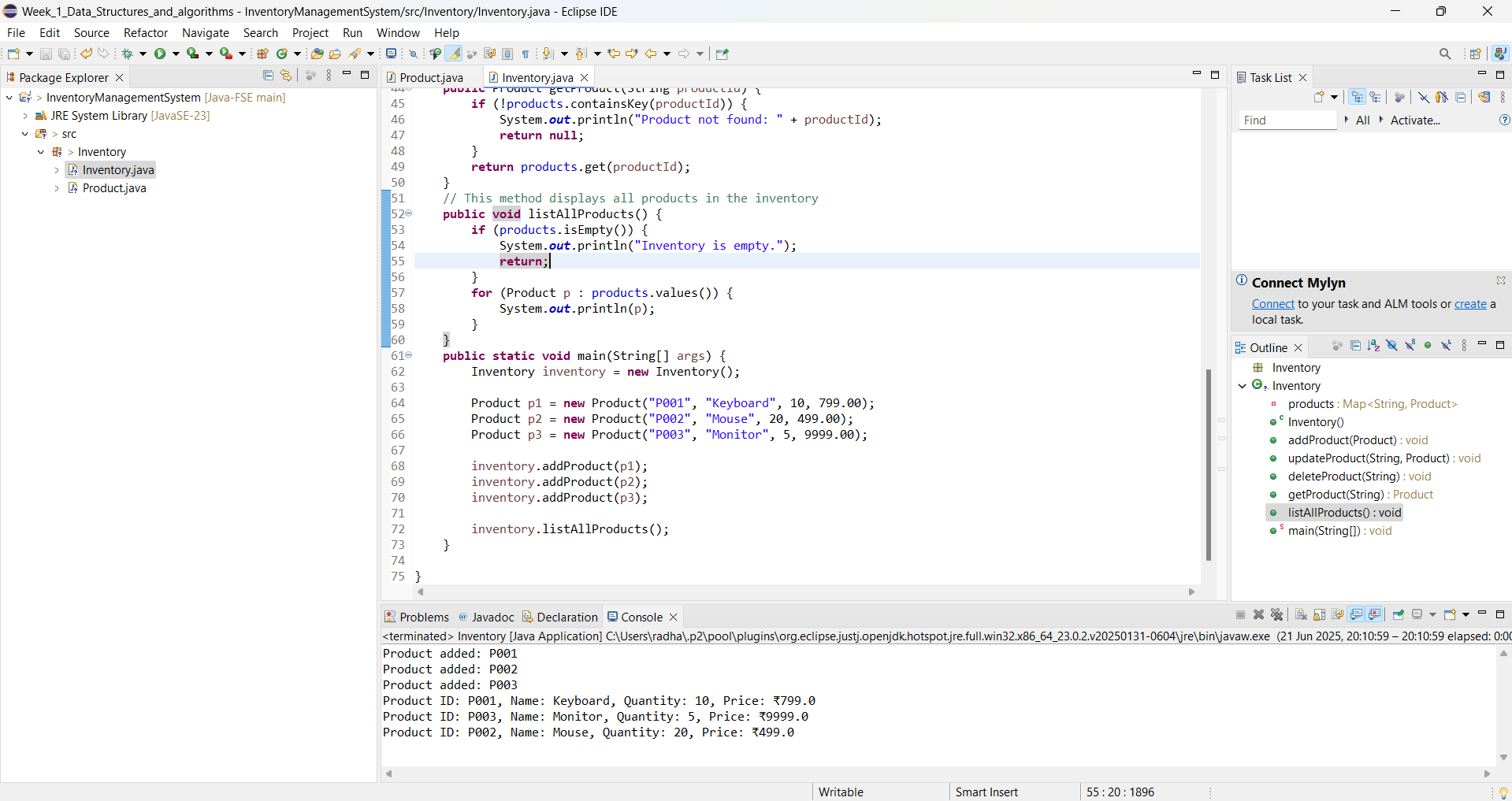
inventory.listAllProducts();

}

}

Note:Here the listAllProducts() is used to jus confirm whether they are added or not.

**Output:**

****

**Analysis:**

Time Complexity (Using HashMap)

* **Add method**: O(1)
* **Update method**: O(1) — same as put() if the key already exists
* **Delete method**: O(1)
* **Get methid**: O(1)

Optimization Tips

* Validate input: Ensure productId is unique when adding.
* Batch updates: If applying multiple updates, group them to avoid repeated rehashing.
* Use LinkedHashMap if insertion order matters.
* For large inventories, consider serialization or database integration for persistent storage.

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

Q:Explain Big O notation and how it helps in analyzing algorithms.

Q:Describe the best, average, and worst-case scenarios for search operations.

Big O notation explains how an algorithm's runtime increases as the size of the input increases. By giving us a worst-case upper bound, it enables us to compare the effectiveness of algorithms without regard to implementation or hardware.   
Best, Average, and Worst Situation   
Regarding search activities:   
In the best scenario, the item appears in the initial comparisons.   
The item falls somewhere in the middle of the average case.   
In the worst scenario, the item is either missing or at the end.

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

**E-CommercePlatformSearch project:**

**Product.java:**

**package** ecommerce;

**public** **class** Product {

**private** String productId;

**private** String productName;

**private** String category;

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

**this**.productId = productId;

**this**.productName = productName;

**this**.category = category;

}

**public** String getProductId() {

**return** productId;

}

**public** String getProductName() {

**return** productName;

}

**public** String getCategory() {

**return** category;

}

@Override

**public** String toString() {

**return** productId + " - " + productName + " [" + category + "]";

}

}

**SearchingOperations.java:**

**package** ecommerce;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** SearchingOperations {

**public** **static** Product linearSearch(Product[] products, String targetName) {

**for** (Product product : products) {

**if** (product.getProductName().equalsIgnoreCase(targetName)) {

**return** product;

}

}

**return** **null**;

}

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

Arrays.*sort*(products, Comparator.*comparing*(Product::getProductName));

**int** low = 0, high = products.length - 1;

**while** (low <= high) {

**int** mid = (low + high) / 2;

**int** cmp = products[mid].getProductName().compareToIgnoreCase(targetName);

**if** (cmp == 0) {

**return** products[mid];

} **else** **if** (cmp < 0) {

low = mid + 1;

} **else** {

high = mid - 1;

}

}

**return** **null**;

}

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

Product[] products = {

**new** Product("P001", "Keyboard", "Electronics"),

**new** Product("P002", "Shoes", "Fashion"),

**new** Product("P003", "Mouse", "Electronics"),

**new** Product("P004", "Shirt", "Fashion"),

};

// Linear search

Product found1 = SearchingOperations.*linearSearch*(products, "Mouse");

System.***out***.println("Linear Search Result: " + (found1 != **null** ? found1 : "Not Found"));

// Binary search

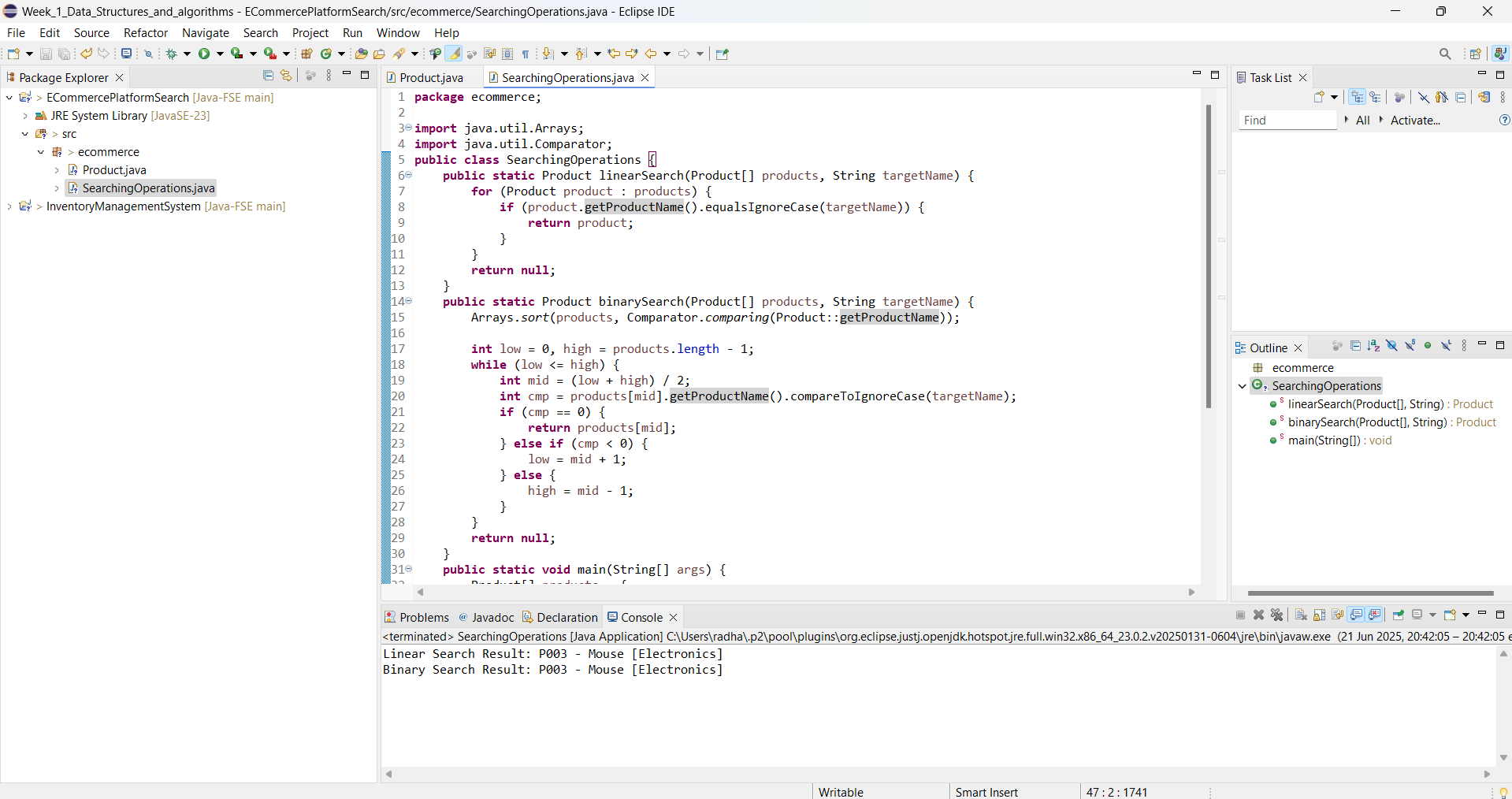
Product found2 = SearchingOperations.*binarySearch*(products, "Mouse");

System.***out***.println("Binary Search Result: " + (found2 != **null** ? found2 : "Not Found"));

}

}

**Output:**

****

Q: Compare the time complexity of linear and binary search algorithms. Discuss which algorithm is more suitable for your platform and why.

Comparison of Time Complexity   
- Linear Search: Search is O(n), but no sorting is necessary.   
-Binary search is O(log n), but it first requires sorting (O(n log n)).   
-For applications that require a lot of reading, such as e-commerce platforms:   
-Better performance is achieved using binary search (on a sorted array or with Collections.binarySearch).   
-If you search frequently and speed in real time is important, choose a HashMap or Trie for lookups that take almost constant amounts of time.

**Exercise 3: Sorting Customer Orders**

Q:Different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort)?

A:

* **Bubble Sort**: Repeatedly compares and swaps adjacent elements if they're in the wrong order. Very intuitive, but inefficient.
  + Time Complexity: Worst/Average - **O(n²)**, Best - **O(n)** (when array is already sorted)
* **Insertion Sort**: Builds the sorted list one element at a time by inserting each new element into its correct position.
  + Time Complexity: Worst/Average - **O(n²)**, Best - **O(n)**
* **Quick Sort**: A divide-and-conquer algorithm. Picks a pivot, partitions the array, and recursively sorts subarrays.
  + Time Complexity: Average - **O(n log n)**, Worst - **O(n²)** (rare, with bad pivot choice)
* **Merge Sort**: Also divide-and-conquer. Always splits the array in half, sorts and merges.
* Time Complexity: All cases - **O(n log n)**, but extra space required.
* **Quick Sort** is faster in practice than Bubble Sort and Insertion Sort due to its divide-and-conquer strategy.

**Order.java:**

**package** Sorting;

**public** **class** Order {

String orderId;

String customerName;

**double** totalPrice;

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

**this**.orderId = orderId;

**this**.customerName = customerName;

**this**.totalPrice = totalPrice;

}

**public** String toString() {

**return** orderId + " - " + customerName + " : ₹" + totalPrice;

} }

**BubbleSorter.java:**

**package** Sorting;

**public** **class** BubbleSorter {

**public** **static** **void** bubbleSort(Order[] orders) {

**int** n = orders.length;

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

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

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

// swap

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

}

**QuickSorter.java:**

**package** Sorting;

**public** **class** QuickSorter {

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

**if** (low < high) {

**int** pivotIndex = *partition*(orders, low, high);

*quickSort*(orders, low, pivotIndex - 1);

*quickSort*(orders, pivotIndex + 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++;

// swap

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

// swap pivot to correct place

Order temp = orders[i + 1];

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

orders[high] = temp;

**return** i + 1;

}

}

**Main.java:**

**package** Sorting;

**public** **class** Main {

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

Order[] orders = {

**new** Order("O101", "Ali", 2499.99),

**new** Order("O102", "Bhanu", 999.50),

**new** Order("O103", "Chetan", 1500.00),

**new** Order("O104", "Deepa", 2999.00)

};

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

*printOrders*(orders);

// Bubble Sort

BubbleSorter.*bubbleSort*(orders);

System.***out***.println("\nSorted by Bubble Sort (Low to High):");

*printOrders*(orders);

// Shuffle and re-sort using Quick Sort

Order[] orders2 = {

**new** Order("O101", "Ali", 2499.99),

**new** Order("O102", "Bhanu", 999.50),

**new** Order("O103", "Chetan", 1500.00),

**new** Order("O104", "Deepa", 2999.00)

};

QuickSorter.*quickSort*(orders2, 0, orders2.length - 1);

System.***out***.println("\nSorted by Quick Sort (Low to High):");

*printOrders*(orders2);

}

**static** **void** printOrders(Order[] orders) {

**for** (Order o : orders) {

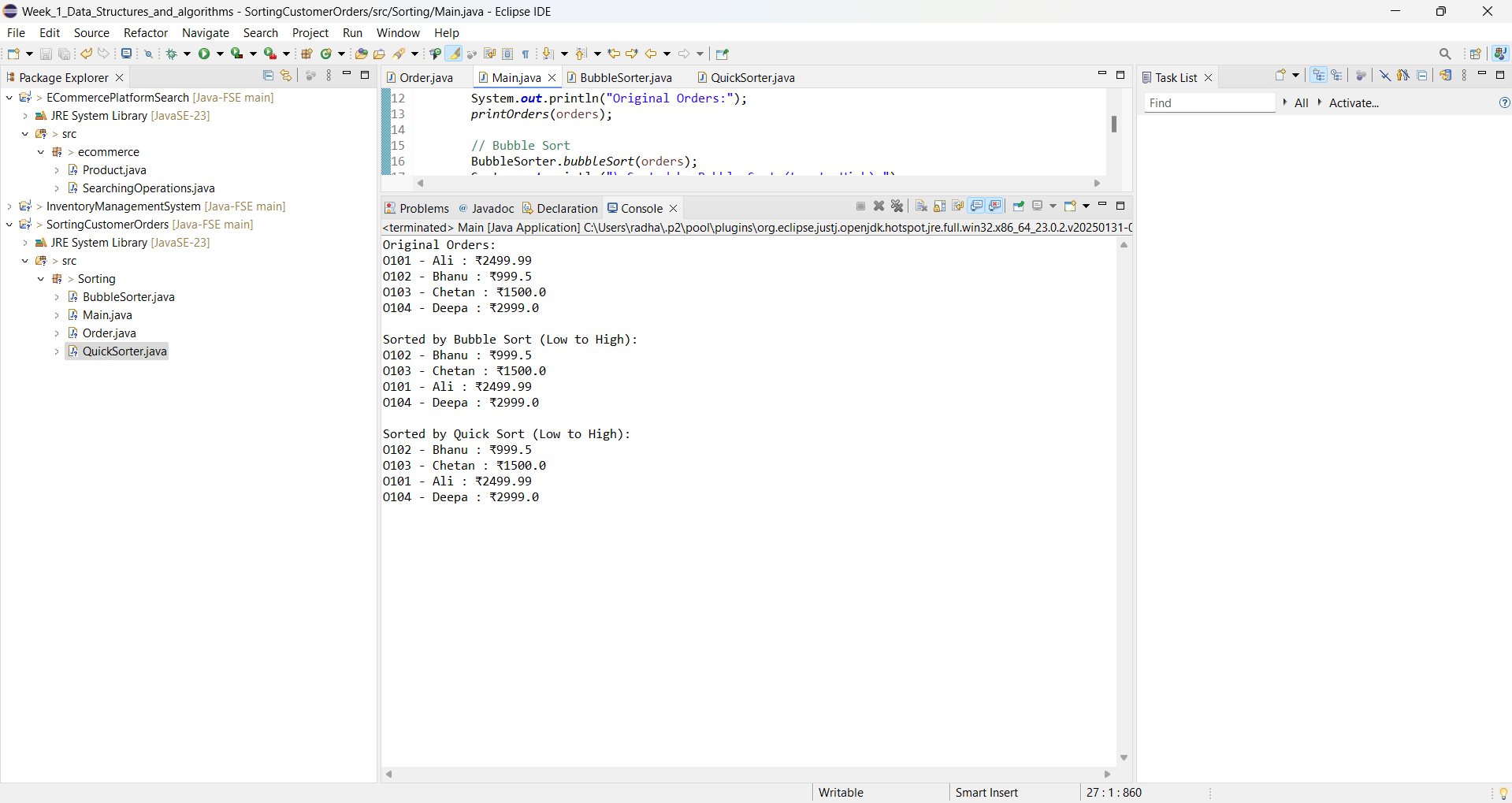
System.***out***.println(o);

}

}

}

**Output:**

****

Q:Compare the performance (time complexity) of Bubble Sort and Quick Sort. Discuss why Quick Sort is generally preferred over Bubble Sort.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity (Avg)** | **Time Complexity (Worst)** | **Space** | **Is Stable?** |
| Bubble Sort | O(n²) | O(n²) | O(1) | Yes |
| Quick Sort | O(n log n) | O(n²) | O(log n) | No |

Why Quick Sort ?

* **Efficiency**: Performs very well in practice with large datasets.
* **In-place sorting**: Saves memory compared to merge sort.
* **Divide-and-conquer strategy**: Smart partitioning makes it fast even without knowing the final sorted shape

**Exercise 4: Employee Management System**

Q:How arrays are represented in memory and their advantages?

A: Arrays are **contiguous blocks of memory** where elements are stored one after another.

Advantages:

-Fast access with index.

-Easy to implement and use.

-Suitable when the number of elements is known and fixed.

**Employee.java:**

**package** EmployeeManagement;

**public** **class** Employee {

String employeeId;

String name;

String position;

**double** salary;

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

**this**.employeeId = employeeId;

**this**.name = name;

**this**.position = position;

**this**.salary = salary;

}

**public** String toString() {

**return** employeeId + " - " + name + " (" + position + ") ₹" + salary;

}

}

**EmployeeManagementSystem.java:**

**package** EmployeeManagement;

**public** **class** EmployeeManagementSystem {

**private** Employee[] employees;

**private** **int** count;

**public** EmployeeManagementSystem(**int** capacity) {

employees = **new** Employee[capacity];

count = 0;

}

// Adds an employee

**public** **void** addEmployee(Employee e) {

**if** (count < employees.length) {

employees[count] = e;

count++;

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

} **else** {

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

}

}

// Search employee by ID

**public** Employee searchEmployee(String empId) {

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

**if** (employees[i].employeeId.equalsIgnoreCase(empId)) {

**return** employees[i];

}

}

**return** **null**;

}

// Traverse all employees

**public** **void** listEmployees() {

**if** (count == 0) {

System.***out***.println("No employees to display.");

**return**;

}

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

System.***out***.println(employees[i]);

}

}

// Delete employee by ID

**public** **void** deleteEmployee(String empId) {

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

**if** (employees[i].employeeId.equalsIgnoreCase(empId)) {

// Shift elements left

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

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

}

employees[count - 1] = **null**;

count--;

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

**return**;

}

}

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

}

}

**Main.java:**

**package** EmployeeManagement;

**public** **class** Main {

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

EmployeeManagementSystem manager = **new** EmployeeManagementSystem(5);

manager.addEmployee(**new** Employee("E001", "Ali", "Manager", 50000));

manager.addEmployee(**new** Employee("E002", "Bhanu", "Developer", 35000));

manager.addEmployee(**new** Employee("E003", "Chaaru", "Designer", 30000));

System.***out***.println("\nAll Employees:");

manager.listEmployees();

System.***out***.println("\nSearching for E002:");

Employee e = manager.searchEmployee("E002");

System.***out***.println(e != **null** ? e : "Not found");

System.***out***.println("\nDeleting E002:");

manager.deleteEmployee("E002");

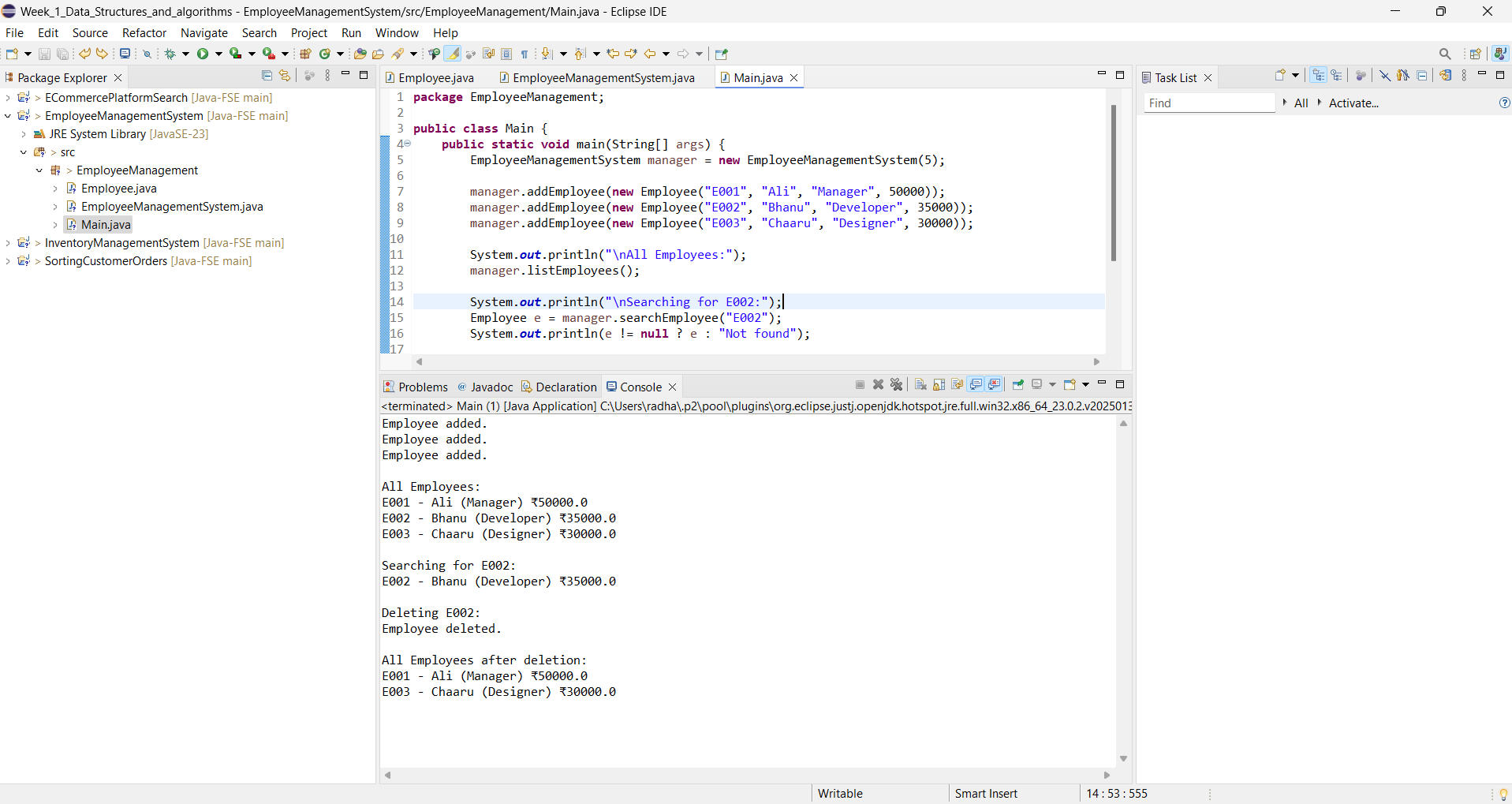
System.***out***.println("\nAll Employees after deletion:");

manager.listEmployees();

}

}

**Output:**



Q: Analyze the time complexity of each operation (add, search, traverse, delete). o Discuss the limitations of arrays and when to use them.

|  |  |  |
| --- | --- | --- |
| **Operation** | **Time Complexity** | **Notes** |
| Add | O(1) | At end of array |
| Search | O(n) | Linear search |
| Traverse | O(n) | One pass through array |
| Delete | O(n) | Due to shifting |

**Limitations of Arrays**

* **Fixed size**: Cannot grow dynamically.
* **Wasted space**: If declared too large and unused.
* **Costly delete**: Shifting elements is inefficient.
* Better to use **ArrayList** or **HashMap** for large, dynamic datasets.

**Exercise 5: Task Management System**

Q: Explain the different types of linked lists (Singly Linked List, Doubly Linked List).

A: Singly Linked List

* Each node contains **data** and a **pointer to the next** node.
* Simple, memory-efficient structure.
* One-directional traversal only.

Doubly Linked List

* Each node contains **data**, a **next pointer**, and a **previous pointer**.
* Allows **bidirectional traversal**.
* Heavier in memory due to the extra pointer but more flexible for operations like reverse traversal and deletions.

**Task.java:**

package TaskManagement;

public class Task {

String taskId;

String taskName;

String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

public String toString() {

return taskId + " - " + taskName + " [" + status + "]";

}

}

**Node.java:**

package TaskManagement;

class Node {

Task task;

Node next;

public Node(Task task) {

this.task = task;

this.next = null;

}

}

**TaskManagement.java:**

**package** TaskManagement;

**public** **class** TaskManagement {

**private** Node head;

// Add task to the end

**public** **void** addTask(Task task) {

Node newNode = **new** Node(task);

**if** (head == **null**) {

head = newNode;

} **else** {

Node temp = head;

**while** (temp.next != **null**) {

temp = temp.next;

}

temp.next = newNode;

}

System.***out***.println("Task added.");

}

// Search task by ID

**public** Task searchTask(String taskId) {

Node temp = head;

**while** (temp != **null**) {

**if** (temp.task.taskId.equalsIgnoreCase(taskId)) {

**return** temp.task;

}

temp = temp.next;

}

**return** **null**;

}

// Delete task by ID

**public** **void** deleteTask(String taskId) {

**if** (head == **null**) {

System.***out***.println("List is empty.");

**return**;

}

**if** (head.task.taskId.equalsIgnoreCase(taskId)) {

head = head.next;

System.***out***.println("Task deleted.");

**return**;

}

Node prev = head;

Node curr = head.next;

**while** (curr != **null**) {

**if** (curr.task.taskId.equalsIgnoreCase(taskId)) {

prev.next = curr.next;

System.***out***.println("Task deleted.");

**return**;

}

prev = curr;

curr = curr.next;

}

System.***out***.println("Task not found.");

}

// Traverse and display tasks

**public** **void** listTasks() {

**if** (head == **null**) {

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

**return**;

}

Node temp = head;

**while** (temp != **null**) {

System.***out***.println(temp.task);

temp = temp.next;

}

}

}

**Main.java:**

**package** TaskManagement;

**public** **class** Main {

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

TaskManagement manager = **new** TaskManagement();

manager.addTask(**new** Task("T001", "Design UI", "Pending"));

manager.addTask(**new** Task("T002", "Develop Backend", "In Progress"));

manager.addTask(**new** Task("T003", "Testing", "Pending"));

System.***out***.println("\nAll Tasks:");

manager.listTasks();

System.***out***.println("\nSearching for T002:");

Task found = manager.searchTask("T002");

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

System.***out***.println("\nDeleting T002:");

manager.deleteTask("T002");

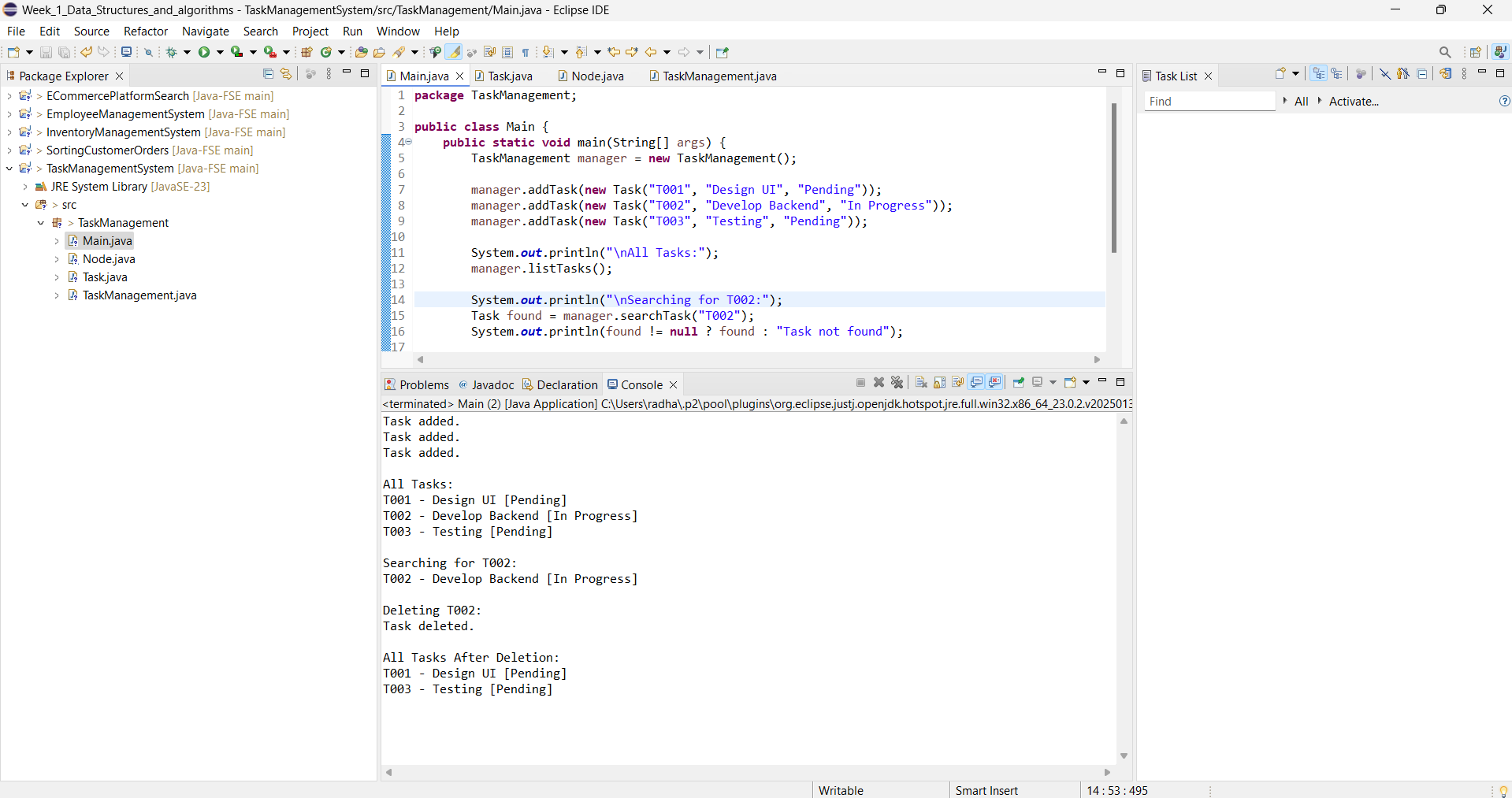
System.***out***.println("\nAll Tasks After Deletion:");

manager.listTasks();

}

}

**Output**:



Analysis

|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Notes |
| Add | O(n) | Traverses to the end to insert |
| Search | O(n) | Linear search by taskId |
| Traverse | O(n) | Visits each task once |
| Delete | O(n) | May need to find and shift links |

Advantages of Linked Lists Over Arrays

* Dynamic resizing: No fixed capacity—it can grow or shrink as needed.
* Efficient insertions or deletions: No need to shift elements like in arrays.
* Better for frequent adds or removes, especially at the start or middle.

**Exercise 6: Library Management System**

Q: Explain linear search and binary search algorithms.

A: **Linear search** is the most basic and straightforward searching algorithm. It works by checking each element in a list or array one by one until the desired value is found or the end of the list is reached. This method does not require the data to be sorted, making it suitable for small or unsorted datasets. However, it is not efficient for large datasets because it may have to examine every element. In the best case, the item is found at the beginning of the list, requiring only one comparison. In the worst case, the item is either at the end or not present at all, resulting in a time complexity of O(n), where *n* is the number of elements.

**Binary search**, on the other hand, is a much faster and more efficient searching technique but it requires the data to be sorted. The algorithm works by repeatedly dividing the search interval in half. It starts by comparing the middle element of the array with the target value. If they match, the search ends successfully. If the target value is smaller, the search continues on the left half; if it is larger, the search proceeds on the right half. This divide-and-conquer approach drastically reduces the number of comparisons, leading to a time complexity of O(log n). Binary search is ideal for large datasets that are already sorted, as it significantly outperforms linear search in such scenarios.

**Book.java:**

**package** Library;

**public** **class** Book {

String bookId;

String title;

String author;

**public** Book(String bookId, String title, String author) {

**this**.bookId = bookId;

**this**.title = title;

**this**.author = author;

}

**public** String toString() {

**return** bookId + " - \"" + title + "\" by " + author;

}

}

**LinearSearch.java:**

**package** Library;

**public** **class** LinearSearch {

**public** **static** Book searchByTitle(Book[] books, String title) {

**for** (Book book : books) {

**if** (book.title.equalsIgnoreCase(title)) {

**return** book;

}

}

**return** **null**;

}

}

**BinarySearch.java:**

**package** Library;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** BinarySearch {

**public** **static** Book searchByTitle(Book[] books, String title) {

Arrays.*sort*(books, Comparator.*comparing*(b -> b.title.toLowerCase()));

**int** low = 0, high = books.length - 1;

**while** (low <= high) {

**int** mid = (low + high) / 2;

**int** cmp = books[mid].title.compareToIgnoreCase(title);

**if** (cmp == 0) **return** books[mid];

**else** **if** (cmp < 0) low = mid + 1;

**else** high = mid - 1;

}

**return** **null**;

}

}

**Main.java:**

**package** Library;

**public** **class** Main {

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

Book[] books = {

**new** Book("B001", "The Alchemist", "Paulo Coelho"),

**new** Book("B002", "Harry Potter", "J.K. Rowling"),

**new** Book("B003", "The horror-terror", "Harper Lee"),

**new** Book("B004", "1984", "George Orwell")

};

// Linear search

System.***out***.println("Linear Search:");

Book result1 = LinearSearch.*searchByTitle*(books, "Harry Potter");

System.***out***.println(result1 != **null** ? result1 : "Book not found");

// Binary search

System.***out***.println("\nBinary Search:");

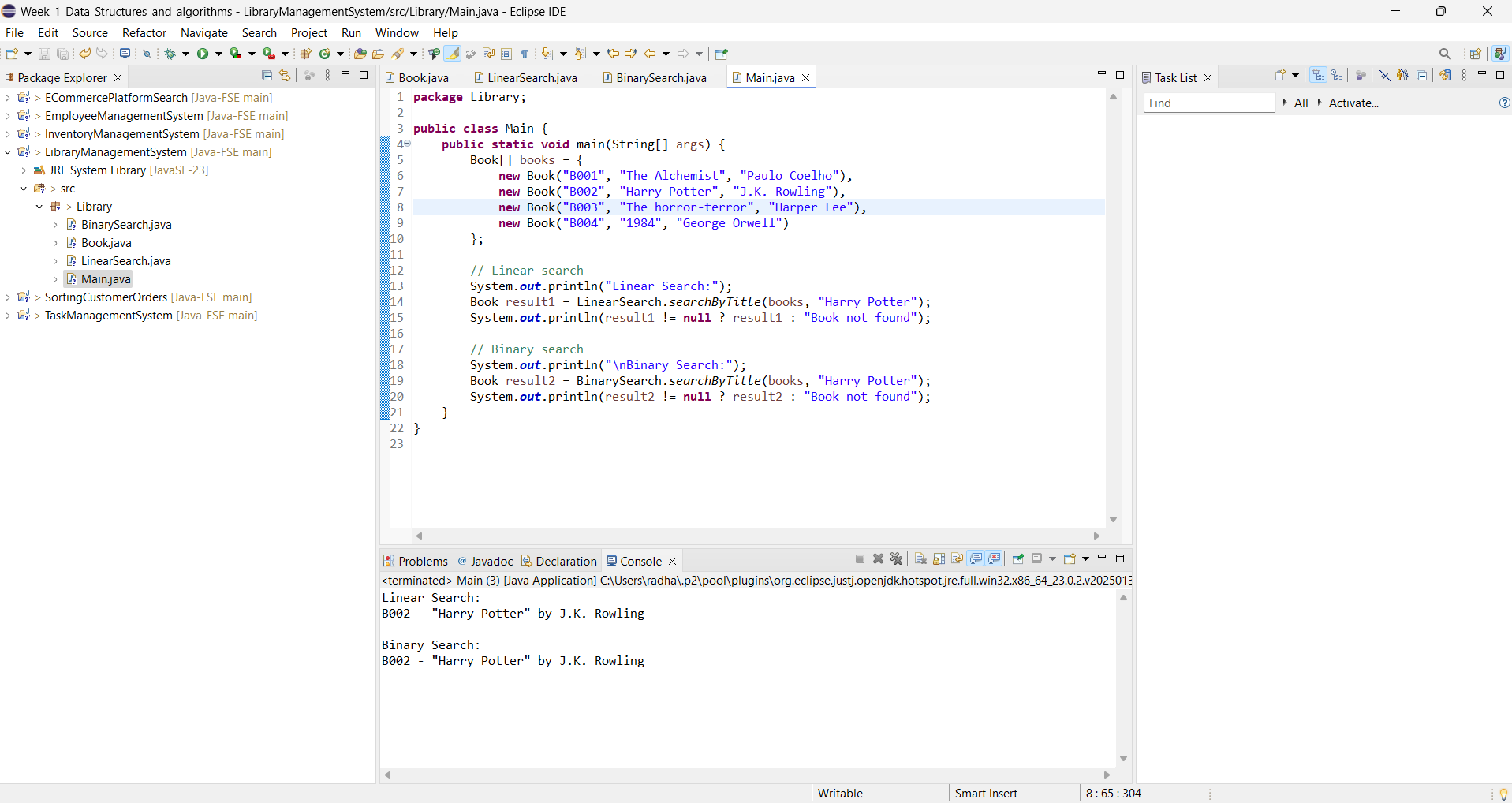
Book result2 = BinarySearch.*searchByTitle*(books, "Harry Potter");

System.***out***.println(result2 != **null** ? result2 : "Book not found");

}

}

**Output:**

****

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

When to Use Each

* **Linear Search**: Small or unsorted collections; simplicity and speed.
* **Binary Search**: Large datasets with sorted entries; optimal for performance.

**Exercise 7: Financial Forecasting**

Recursion is a programming technique where a method calls itself to solve smaller instances of a problem until it reaches a base case. It is useful for problems that can be broken down into simpler, repeating sub-problems like financial projections, factorials, or Fibonacci numbers.

A basic financial forecasting formula using compound growth is:

FutureValue(years) = PresentValue × (1 + growthRate)^years

**FinancialForecast.java:**

**package** Finance;

**public** **class** FinancialForecast {

// Recursive method to calculate future value

**public** **static** **double** forecast(**double** presentValue, **double** growthRate, **int** years) {

**if** (years == 0) {

**return** presentValue;

}

**return** (1 + growthRate) \* *forecast*(presentValue, growthRate, years - 1);

}

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

**double** presentValue = 10000;

**double** growthRate = 0.08;

**int** years = 5;

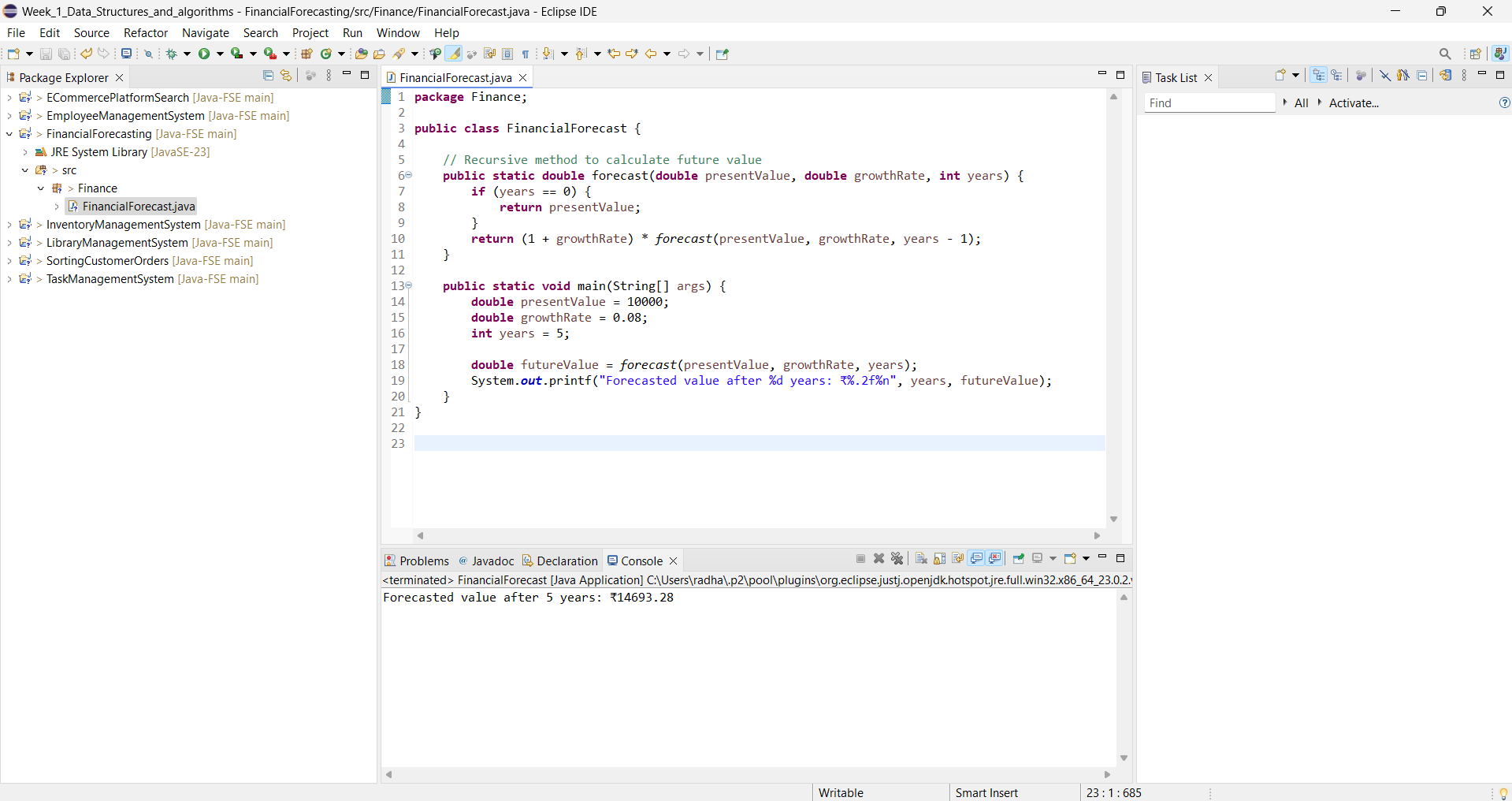
**double** futureValue = *forecast*(presentValue, growthRate, years);

System.***out***.printf("Forecasted value after %d years: ₹%.2f%n", years, futureValue);

}

}

**Output:**



Time Complexity

* The time complexity is O(n), where n is the number of years.
* Each recursive call processes one year until the base case (years == 0) is reached.

Optimization can be done:

Recursion can be elegant but also inefficient if recomputation happens. In this case, recursion is safe because:

* There's only one recursive path (no overlapping subproblems).
* The function is tail-recursive and memory usage is minimal.

For more complex forecasting (e.g., using Fibonacci-like patterns), memoization or dynamic programming can be used to store intermediate results and avoid redundant computation.

Alternatively, for efficiency, use the non-recursive formul**a**:

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