**Digital NUTURE 4.0 – DEEP SKILLING STAGE**

Data Structure And Algorithm

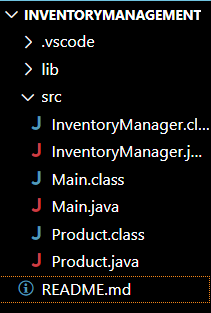
**Exercise 1: Inventory Management System**

**Scenario:**

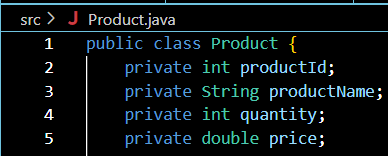
You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

**Steps:**

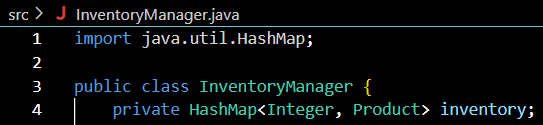
1. **Understand the Problem:**
   * Explain why data structures and algorithms are essential in handling large inventories.
     + **Efficiency**: Large inventories can have thousands of products; efficient algorithms ensure quick add/update/delete/search.
     + **Scalability**: Proper data structures handle growing datasets without degrading performance.
     + **Organization**: Structures like HashMap allow for quick lookups based on keys (e.g., productId).
   * Discuss the types of data structures suitable for this problem.
     + **ArrayList**: Good for ordered lists but slow for search/delete by ID (O(n) time).
     + **HashMap**: Ideal for key-based retrieval using productId as the key (O(1) average time for add, delete, and update).
     + **Best Choice: HashMap<String, Product>** where String is the productId.
2. **Setup:**
   * Create a new project for the inventory management system.



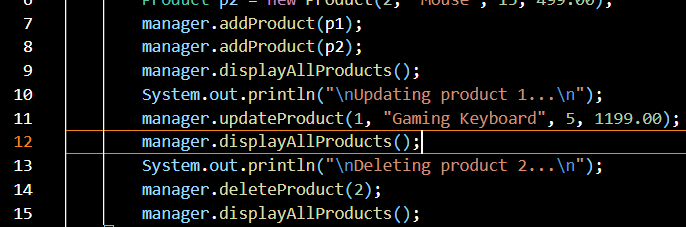
1. **Implementation:**
   * Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.



* + Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).



* + Implement methods to add, update, and delete products from the inventory.



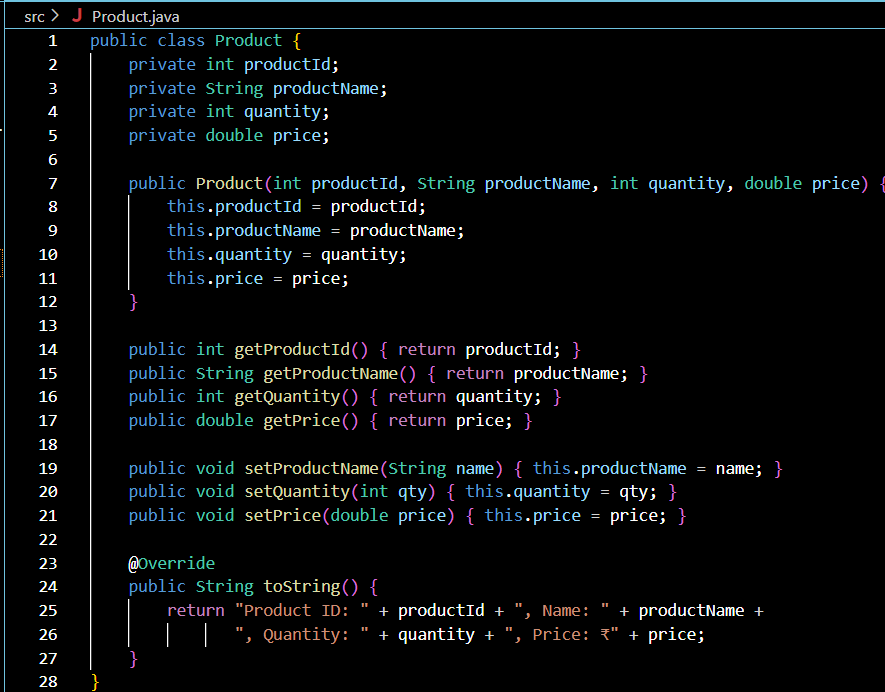
1. **Analysis:**
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.

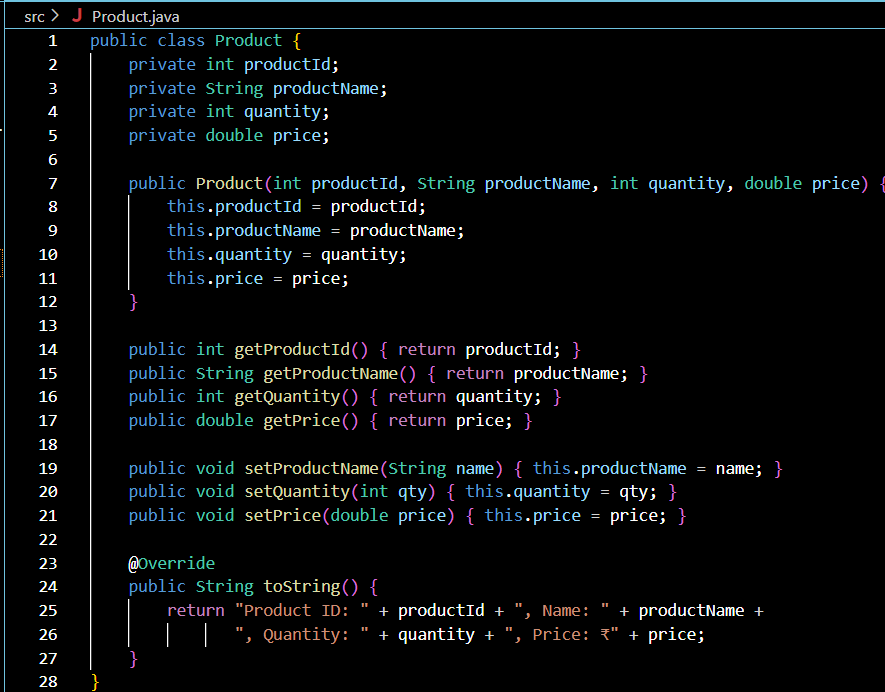
| **Operation** | **Complexity** | **Explanation** |
| --- | --- | --- |
| Add | O(1) average | Hash-based insertion |
| Update | O(1) average | Direct access by key |
| Delete | O(1) average | Remove by key |
| Search | O(1) average | Direct access by key |

* + Discuss how you can optimize these operations.
    - **Avoid key collisions**: Use unique, well-distributed productIds.
    - **Indexing**: If searching by product name or price is needed, consider additional indexing structures (e.g., TreeMap for sorted access).
    - **Persistence**: Store data in a file or database for durability.

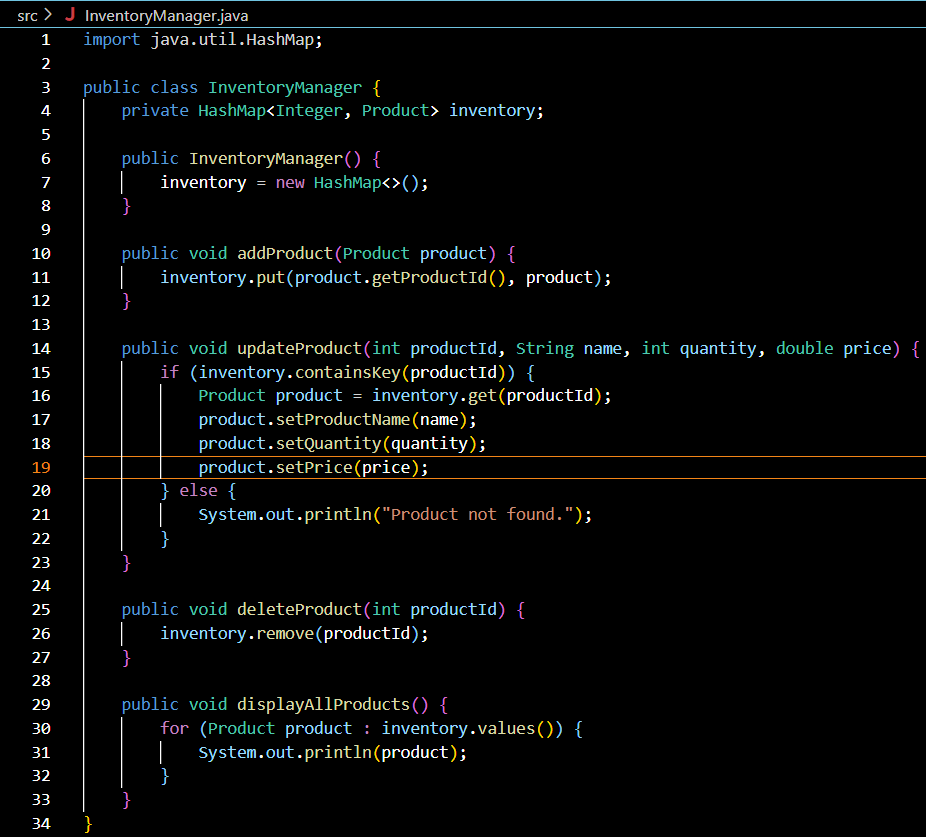
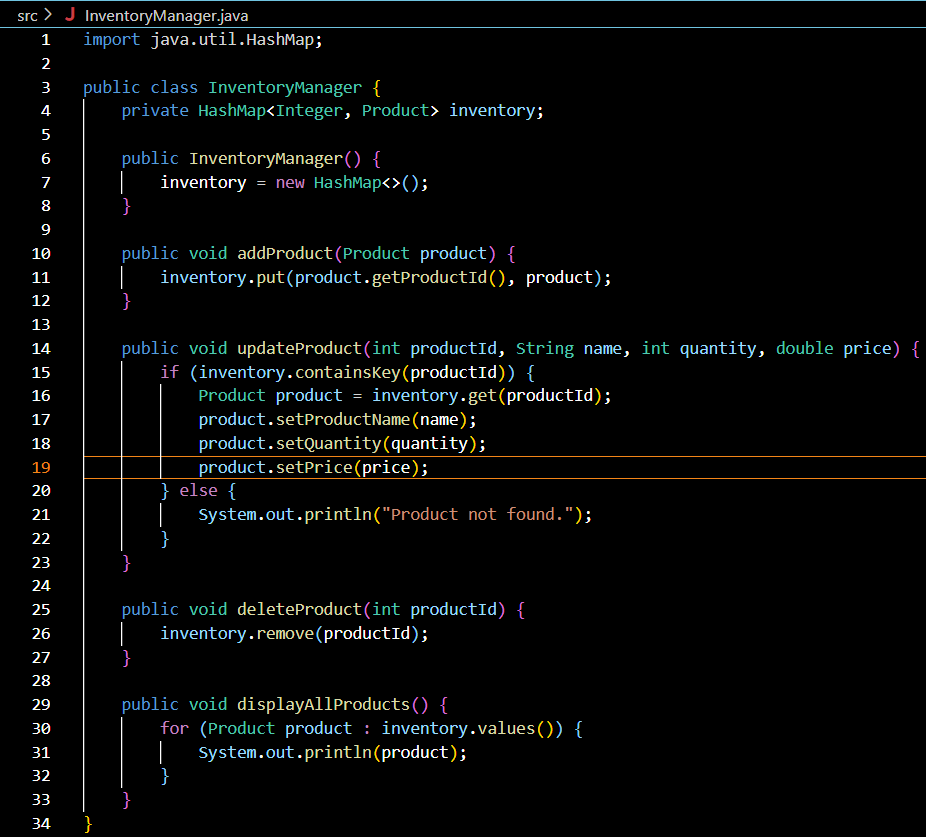
**FINAL CODE->**

**Product.java:**

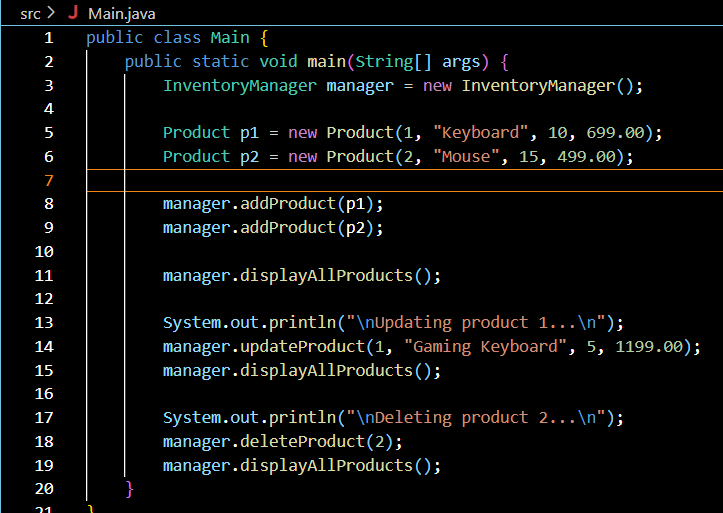


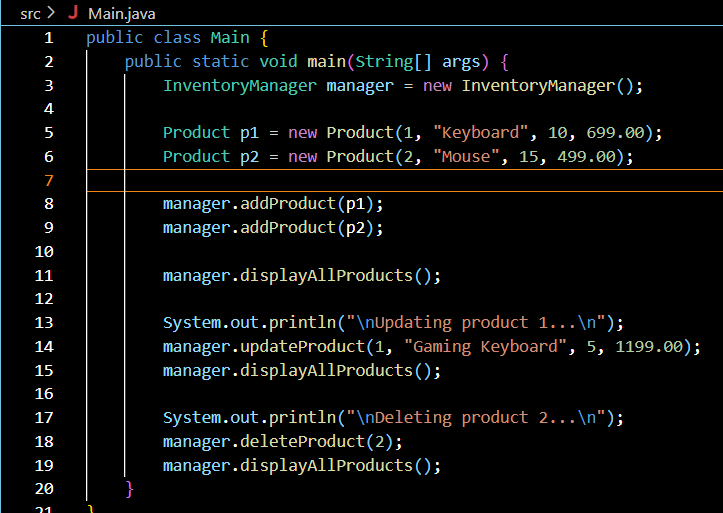


**InventoryManager.java**

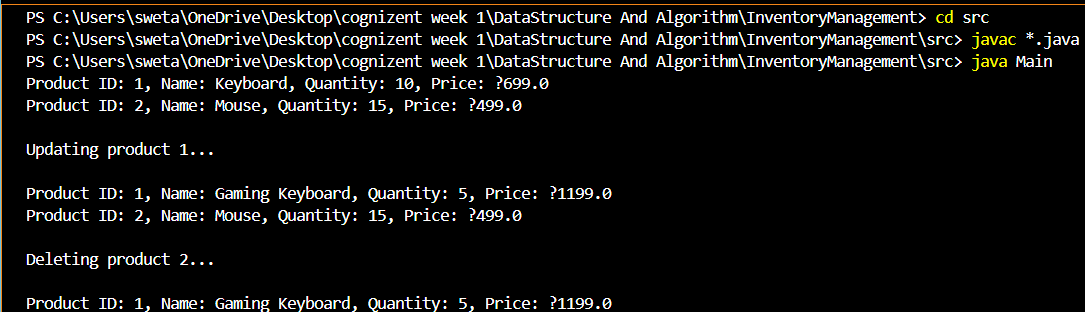


**Main.java:**





**Output:**



**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.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.

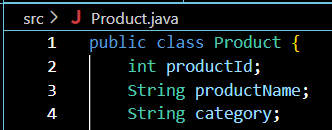
Big O Notation is used to describe the **time complexity** of an algorithm — how the runtime grows with the size of the input n.

| **Notation** | **Meaning** | **Example Algorithm** |
| --- | --- | --- |
| O(1) | Constant time | Accessing an array element |
| O(n) | Linear time | Linear search |
| O(log n) | Logarithmic time | Binary search |
| O(n²) | Quadratic time | Nested loops |

* + Describe the best, average, and worst-case scenarios for search operations.

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Linear Search | O(1) (first match) | O(n/2) ≈ O(n) | O(n) |
|  |  |  |  |
| Binary Search | O(1) (middle) | O(log n) | O(log n) |

1. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.



1. **Implementation:**
   * Implement linear search and binary search algorithms.



* + Store products in an array for linear search and a sorted array for binary search.



1. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.

| **Algorithm** | **Time Complexity** | **Requires Sorting?** | **Suitable For** |
| --- | --- | --- | --- |
| Linear Search | O(n) | no | Small datasets |
| Binary Search | O(log n) | Sorted array | Large datasets |

* + Discuss which algorithm is more suitable for your platform and why.
    - Use **Linear Search** when:

Data is small.

Data is unsorted.

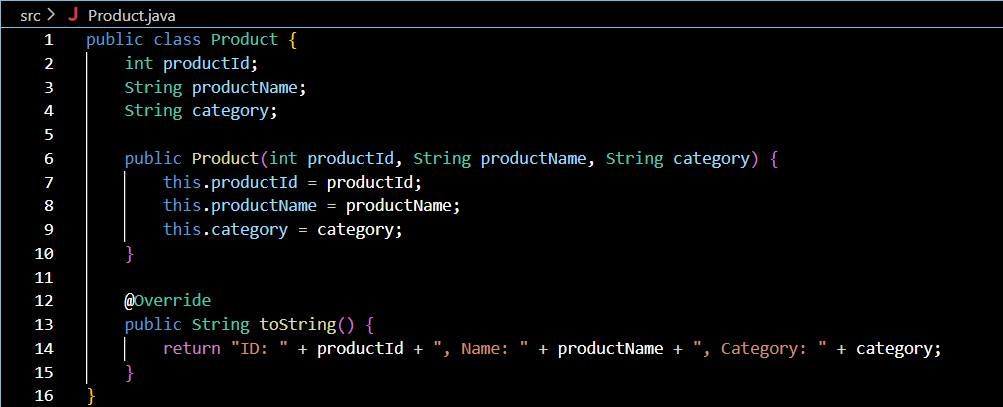
* Use **Binary Search** when:

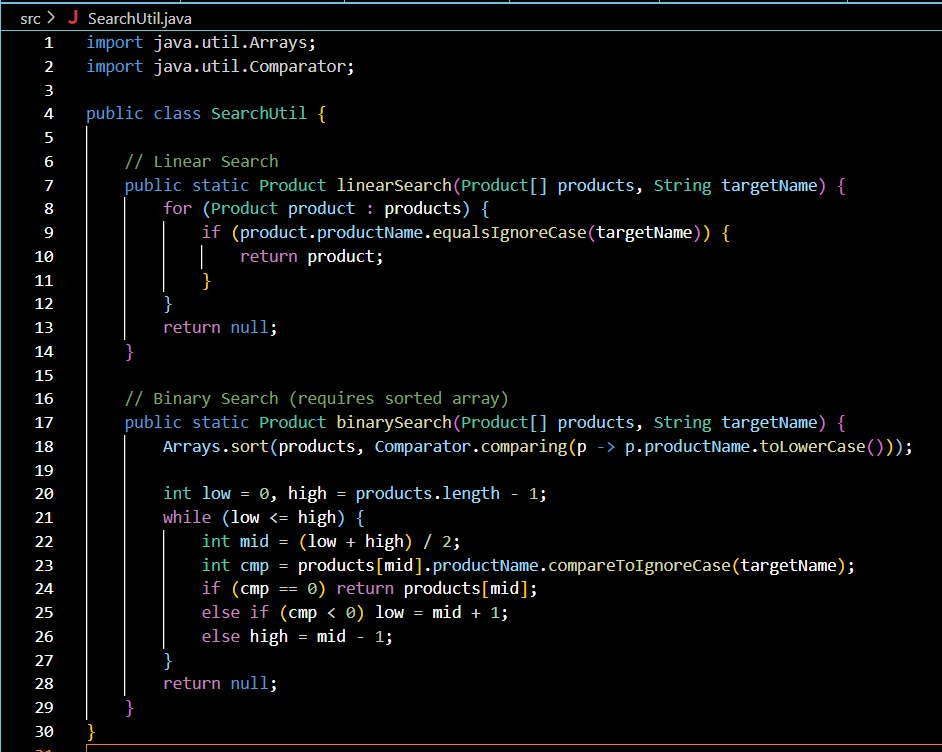
Data is large.

Data is sorted and static (not changing frequently).

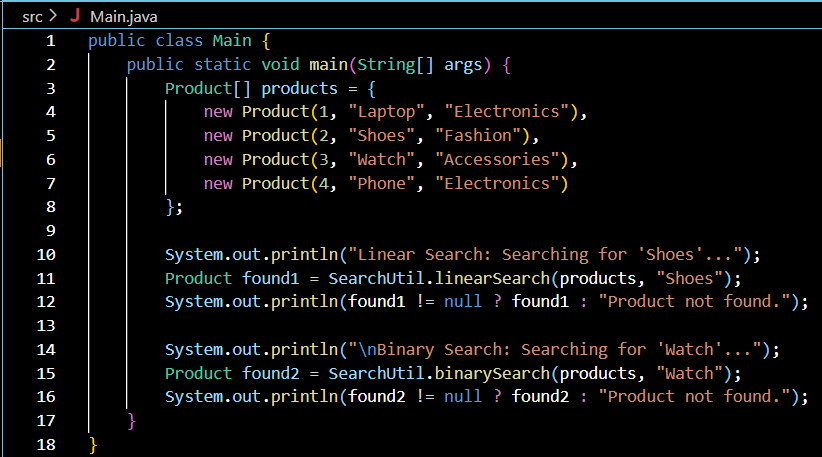
**FINAL CODE->**

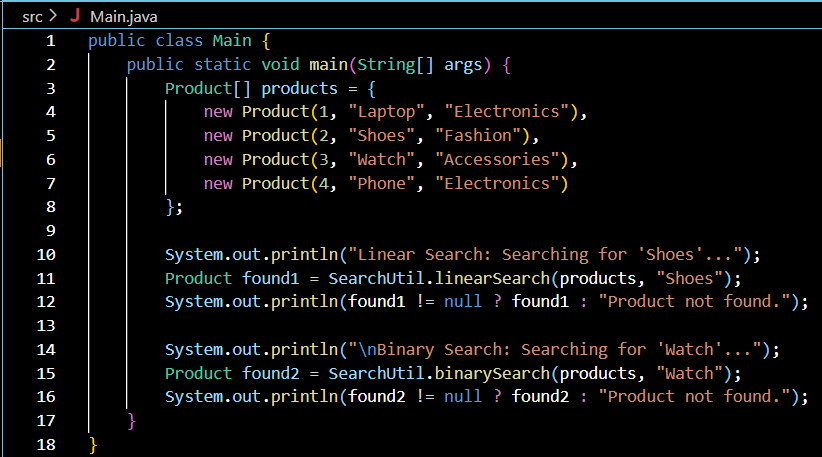
**Product.java:**



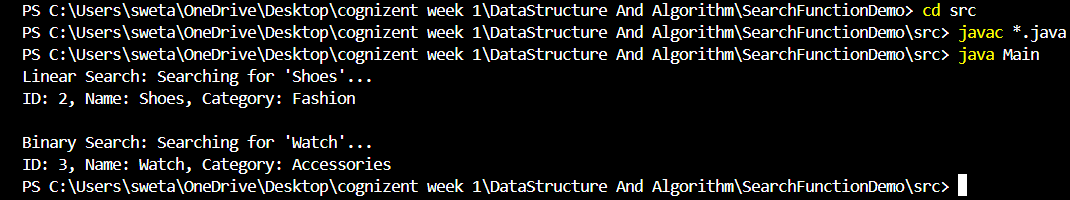
**SearchUtil.java:** 

**Main.java:**





**Output:**



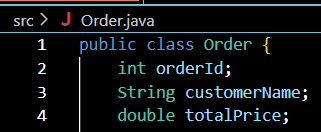
**Exercise 3: Sorting Customer Orders**

**Scenario:**

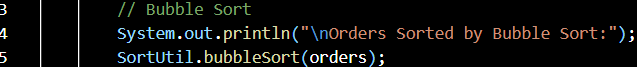
You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

**Steps:**

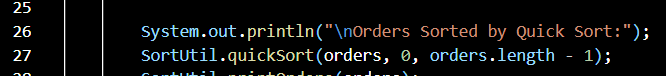
1. **Understand Sorting Algorithms:**
   * Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
   * **Bubble Sort**
     + Repeatedly swaps adjacent elements if they are in the wrong order.
     + **Time Complexity:**
     + Best: O(n) (already sorted)
     + Average/Worst: O(n²)
     + **Use Case:** Educational or very small datasets.
     + **Insertion Sort**
     + Builds the final sorted array one item at a time.
     + **Time Complexity:**
     + Best: O(n)
     + Average/Worst: O(n²)
   * **Quick Sort**
     + Uses divide and conquer; selects a pivot, partitions the array.
     + **Time Complexity:**
     + Best/Average: O(n log n)
     + Worst: O(n²) (rare, but happens when poorly chosen pivots)
   * **Merge Sort**
     + Divides array into halves, recursively sorts, then merges.
     + **Time Complexity: O(n log n)** in all cases.
     + Needs extra space, unlike Quick Sort.
2. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.



1. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.



* + Implement **Quick Sort** to sort orders by **totalPrice**.



1. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Space Complexity** |
| --- | --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) | O(1) |
| Quick Sort | O(n log n) | O(n log n) | O(n²)\* | O(log n) |

* + Discuss why Quick Sort is generally preferred over Bubble Sort.

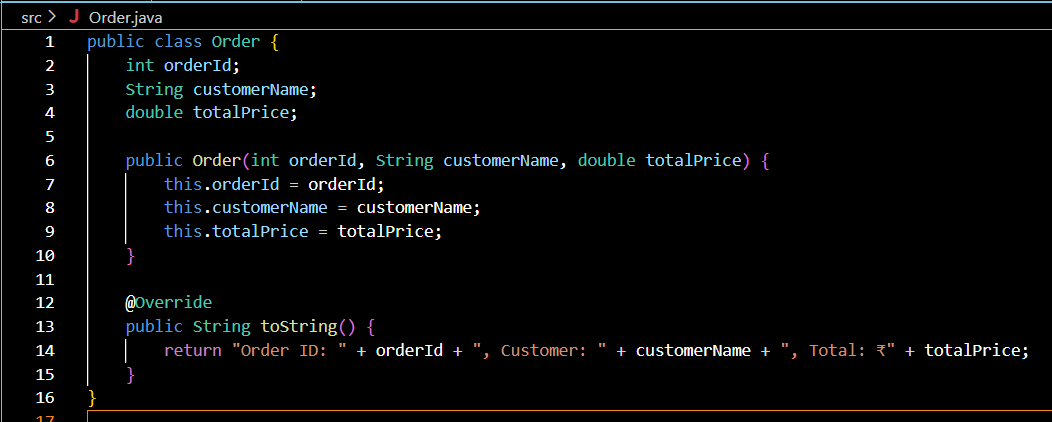
Much faster on average than Bubble Sort.

More efficient for large datasets due to O(n log n) complexity.

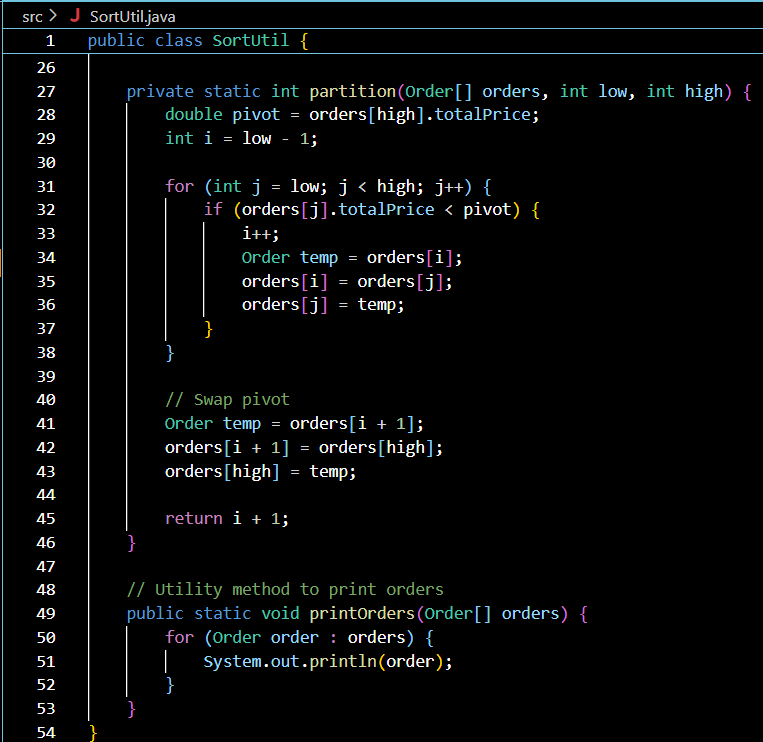
Bubble Sort is simple but inefficient — mainly used for teaching.

**FINAL CODE->**

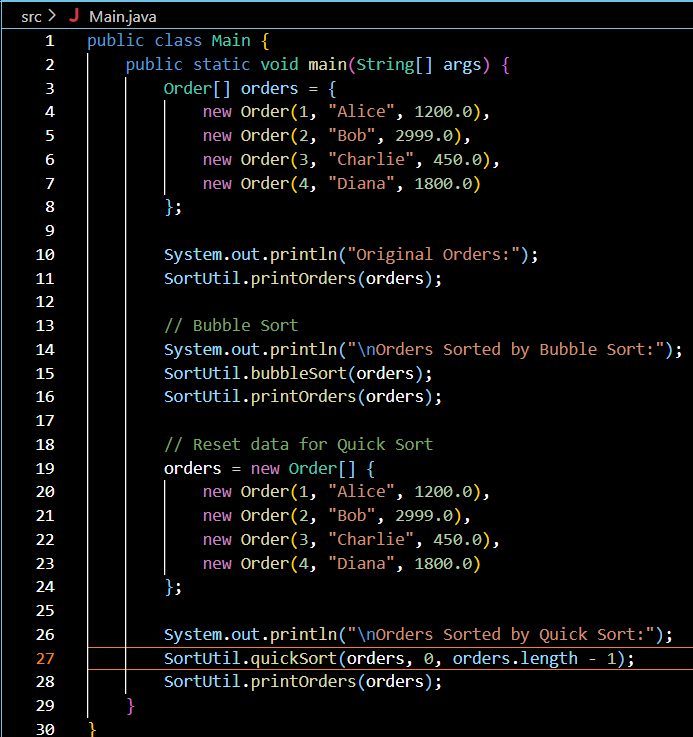
**Order.java:**

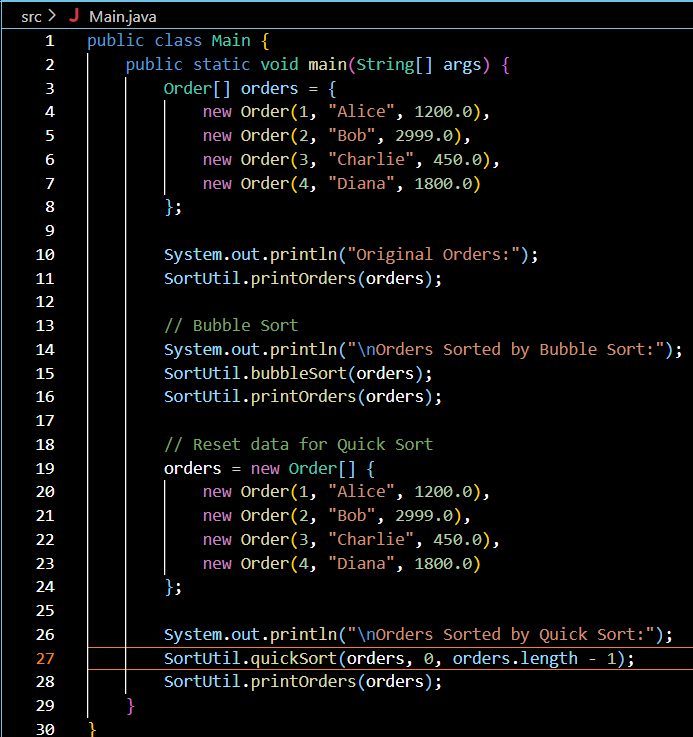


**SortUtill.java**

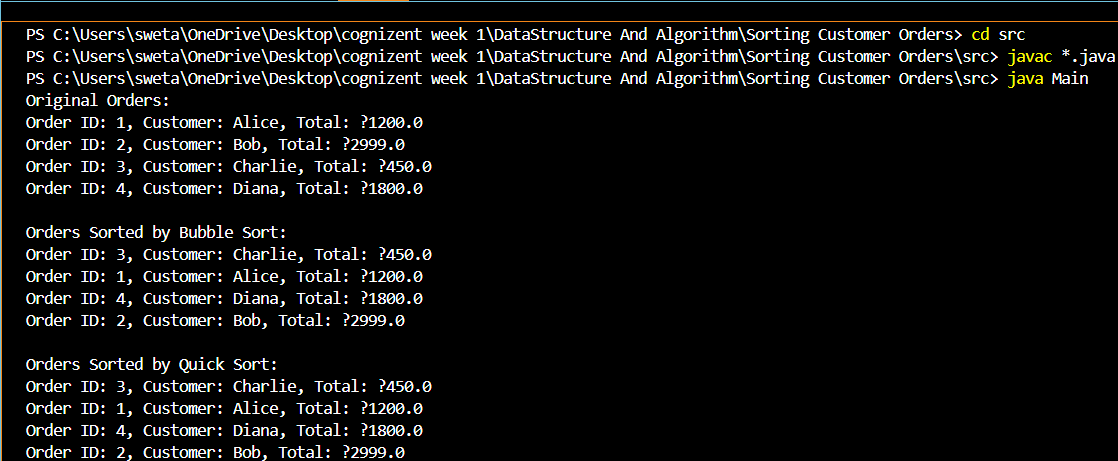


**Main.java**





**Output:**



**Exercise 4: Employee Management System**

**Scenario:**

You are developing an employee management system for a company. Efficiently managing employee records is crucial.

**Steps:**

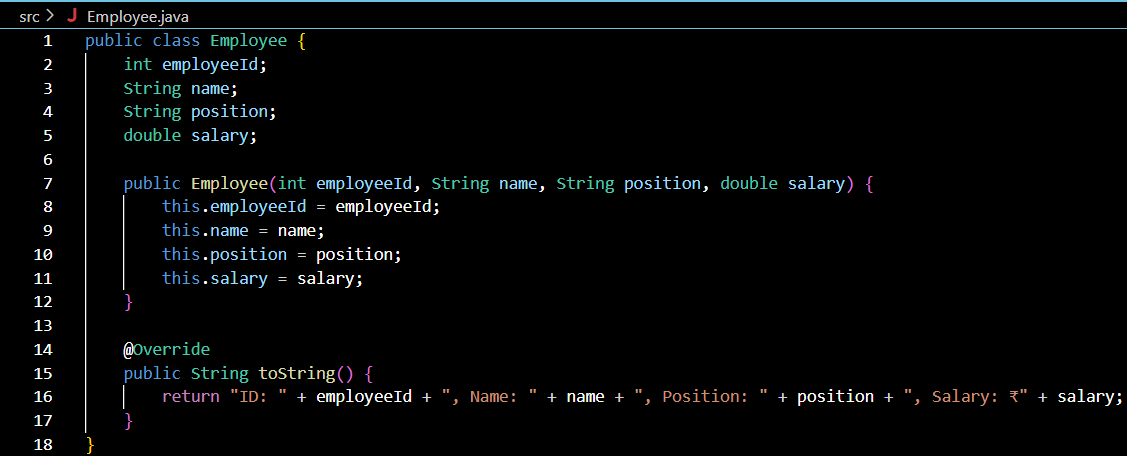
1. **Understand Array Representation:**
   * Explain how arrays are represented in memory and their advantages.

**How Arrays Are Represented in Memory**

* Arrays are **contiguous blocks** of memory.
* Each element is accessed by **indexing**: base\_address + (index × size of data type).
* Efficient for **direct access** using index (O(1) time).

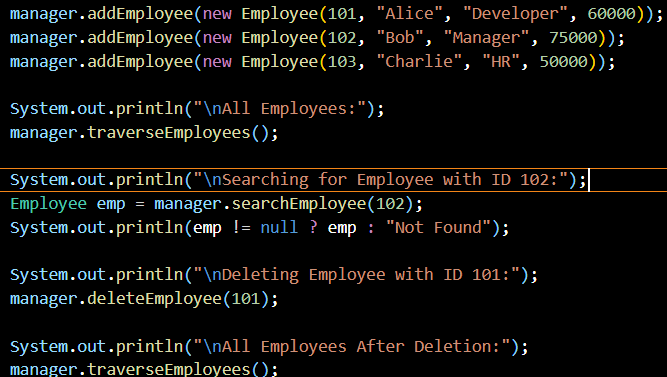
**Advantages of Arrays**

* Fast random access to elements.
* Easy to implement and use.
* Memory efficient for fixed-size data.

1. **Setup:**
   * Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
2. **Implementation:**
   * Use an array to store employee records.



* + Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.



1. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| **Add** | O(1) | Direct insert at next index |
| **Search** | O(n) | Linear scan required |
| **Traverse** | O(n) | One pass over all elements |
| **Delete** | O(n) | Find + shift elements to fill the gap |

* + Discuss the limitations of arrays and when to use them.

**Limitations of Arrays**

Fixed size — hard to scale dynamically.

Insert/delete operations (not at end) are expensive (O(n)).

Not ideal when data size changes frequently.

**When to Use Arrays**

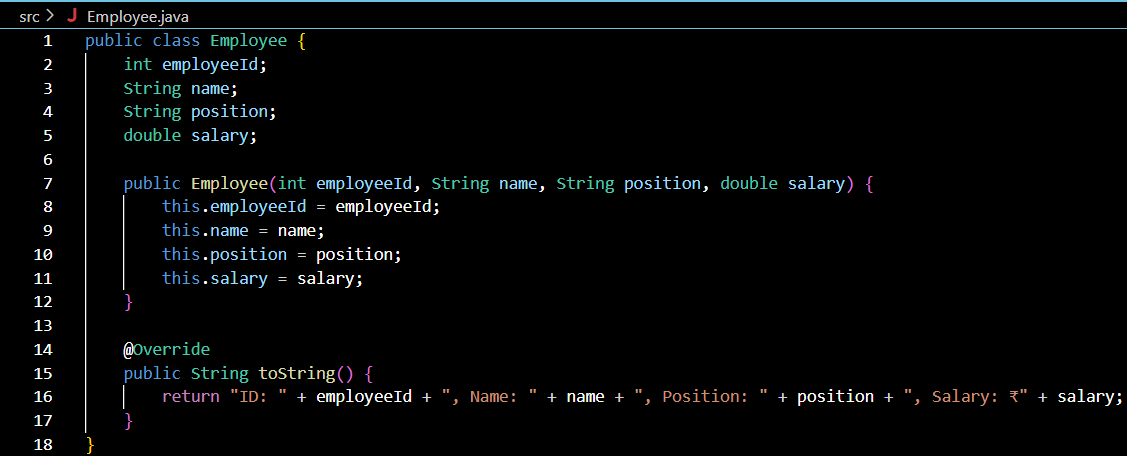
When size is known ahead of time.

When random access (via index) is needed.

For static datasets or performance-critical lookups.

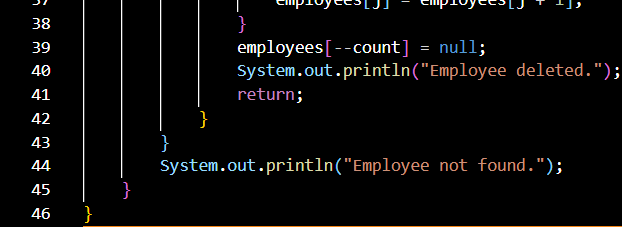
**FINAL CODE->**

**Employee.java**

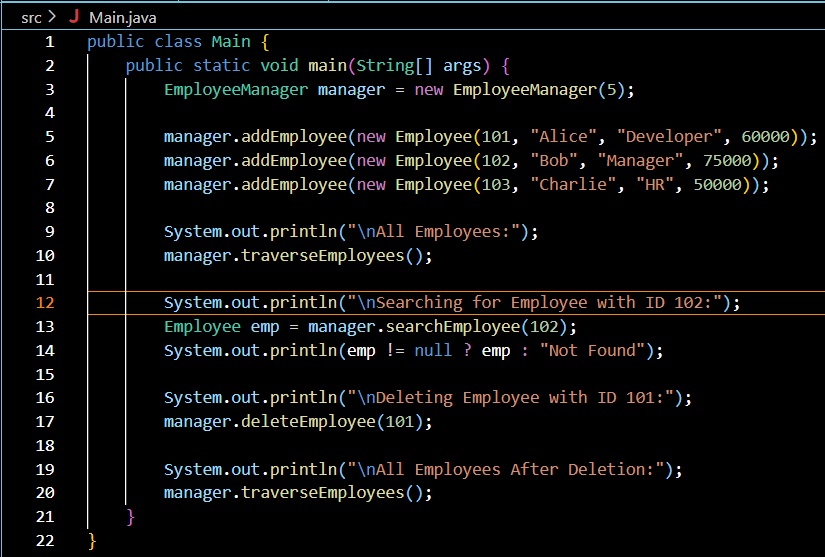
****

**EmployeeManager.java:**

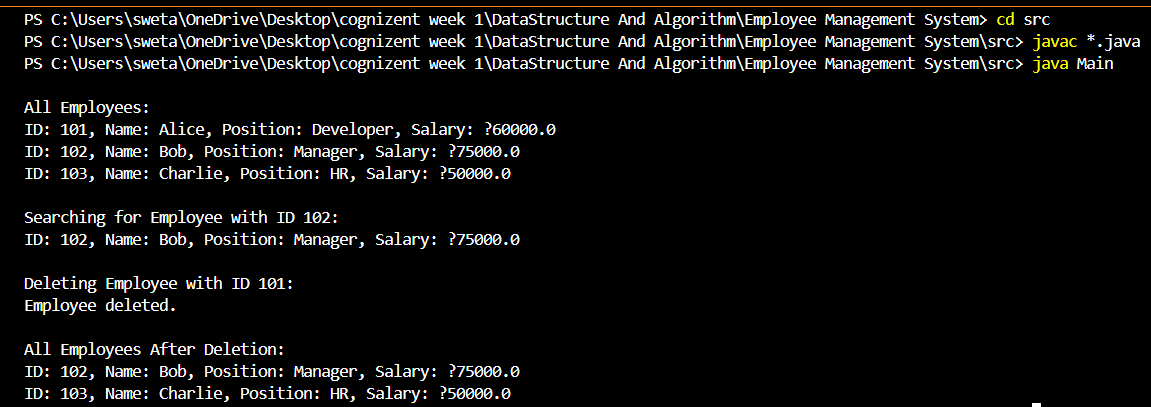
****

****

**Main.java:**

****

**Output.java:**

****

**Exercise 5: Task Management System**

**Scenario:**

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

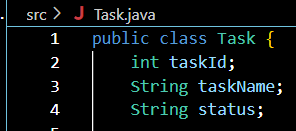
**Steps:**

1. **Understand Linked Lists:**
   * Explain the different types of linked lists (Singly Linked List, Doubly Linked List).

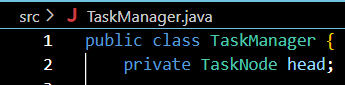
**Types of Linked Lists:**

* **Singly Linked List:**
  + Each node points to the next node.
  + Simple, efficient for forward traversal.
* **Doubly Linked List:**
  + Each node has both next and prev pointers.
  + Allows traversal in both directions.
  + Requires extra memory per node.

1. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.



1. **Implementation:**
   * Implement a singly linked list to manage tasks.



* + Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.







1. **Analysis:**
   * Analyze the time complexity of each operation.

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Add | O(n) | Must traverse to the end of the list |
| Search | O(n) | Linear search through the list |
| Traverse | O(n) | Visit each node once |
| Delete | O(n) | Search + relinking pointers |

* + Discuss the advantages of linked lists over arrays for dynamic data.

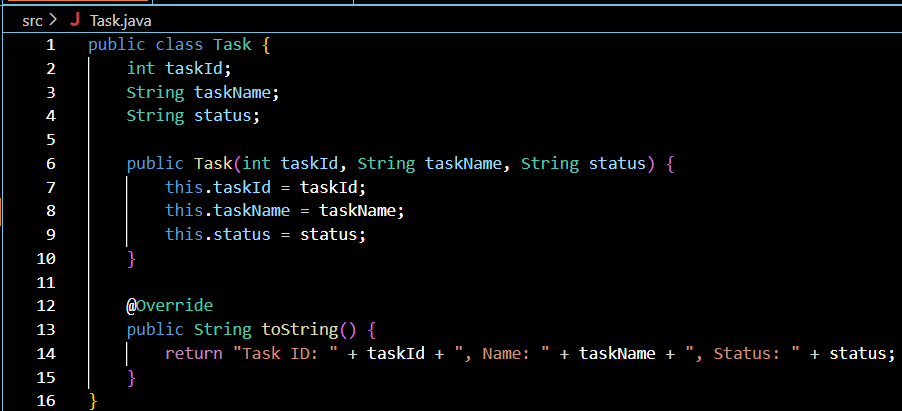
**Dynamic Size**: Easily grows/shrinks without reallocating memory.

**Efficient Deletion/Insertion**: Especially in the middle or start.

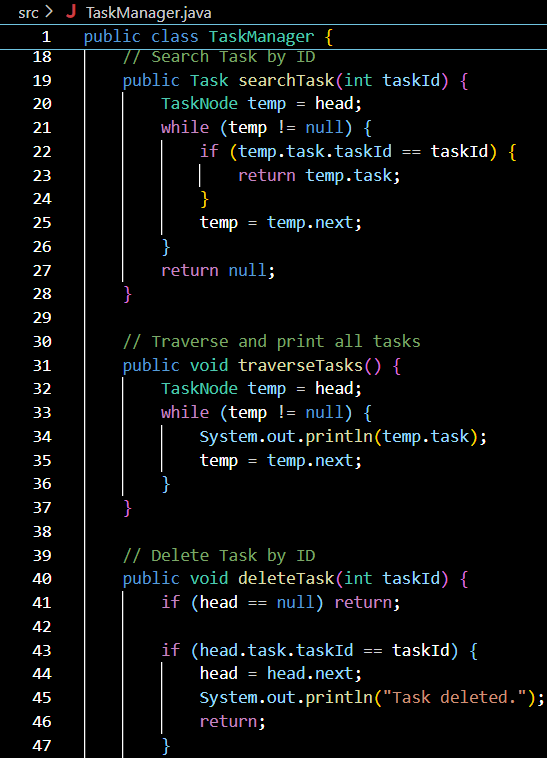
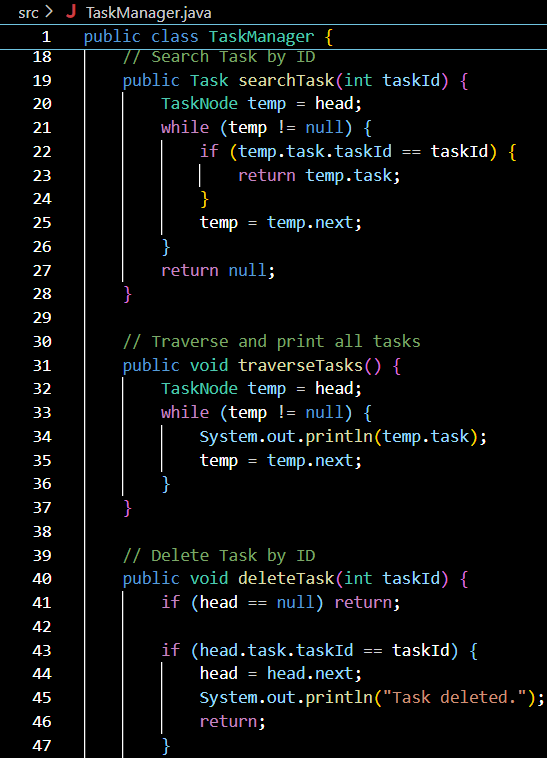
**No Wasted Space**: Arrays may reserve unused slots.

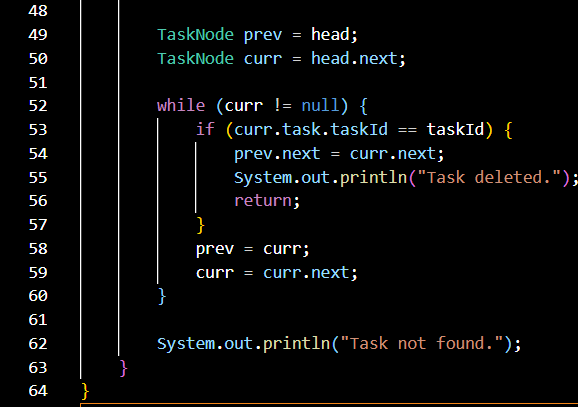
**FINAL CODE->**

**Task.java:**

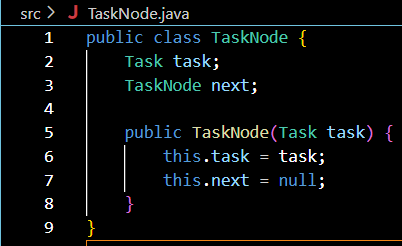


**TaskManager.java:**

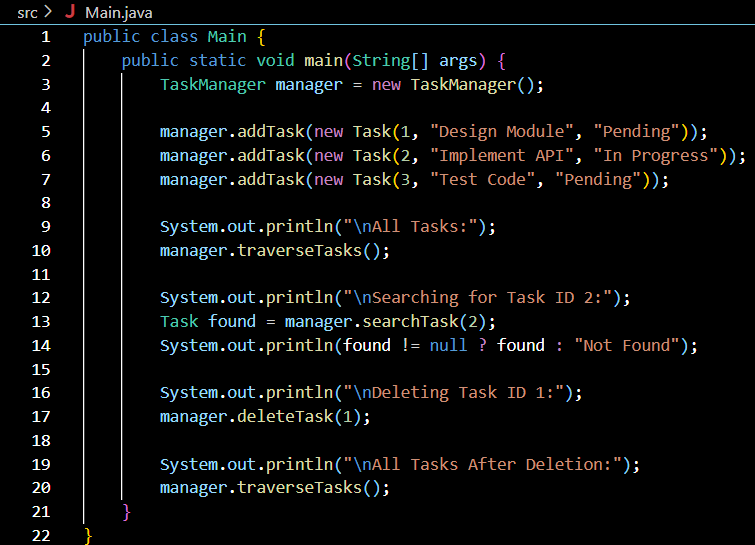
** **

****

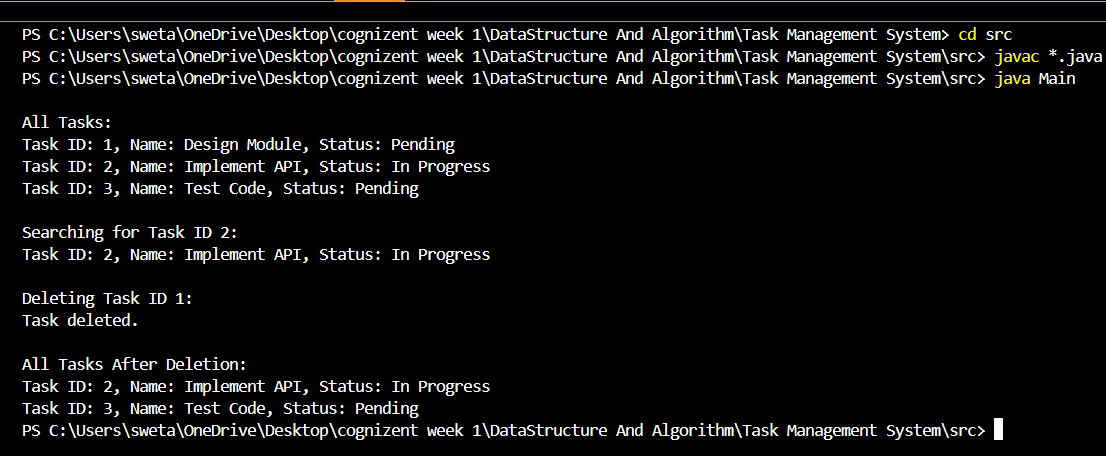
**TaskNode.java:**

****

**Main.java:**

****

**Output:**

****

**Exercise 6: Library Management System**

**Scenario:**

You are developing a library management system where users can search for books by title or author.

**Steps:**

1. **Understand Search Algorithms:**
   * Explain linear search and binary search algorithms.

**Linear Search:**

Scans each element in sequence until a match is found.

**Time Complexity:**

Best: O(1)

Average/Worst: O(n)

Works on **unsorted** data.

**Binary Search:**

Requires a **sorted** array or list.

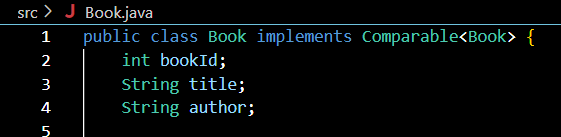
Repeatedly divides the array in half to locate the item.

**Time Complexity:**

Best: O(1)

Average/Worst: O(log n)

**Faster** but only usable on sorted data.

1. **Setup:**
   * Create a class **Book** with attributes like **bookId**, **title**, and **author**. 
2. **Implementation:**
   * Implement linear search to find books by title.



* + Implement binary search to find books by title (assuming the list is sorted).



1. **Analysis:**
   * Compare the time complexity of linear and binary search.

| **Operation** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Time Complexity** | O(n) | O(log n) |
| **Requires Sorting?** | No | Yes |
| **Best Case** | O(1) (first match) | O(1) (middle match) |
| **Usage** | Unsorted data | Sorted data |
| **When to Use** | Small or dynamic data sets | Large, static, sorted data sets |

* + Discuss when to use each algorithm based on the data set size and order.

**Linear Search**

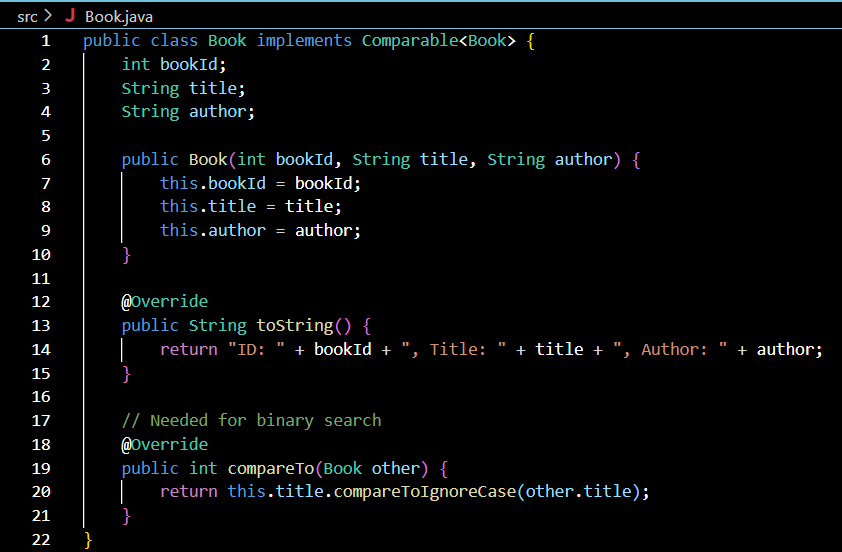
* **When to Use:**
  + **Small data sets**: The overhead of sorting or maintaining order isn't worth it for small collections.
  + **Unsorted or dynamic data**: If the data changes frequently (inserts, deletions), sorting it before every search is inefficient.
  + **Searching in real-time input**: For example, user-entered values on-the-fly.
* **Why it's appropriate:**
  + Does **not require sorted data**.
  + Simple and works on **any data structure** (lists, arrays, etc.).
  + Ideal for **one-time or occasional searches** in unsorted data.

**Binary Search**

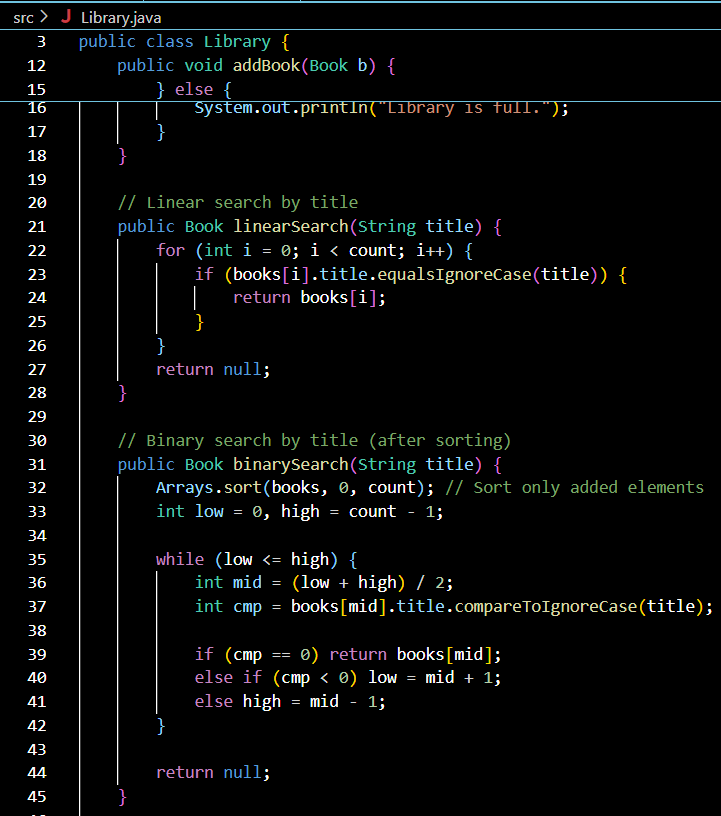
* **When to Use:**
  + **Large, static, and sorted data sets**: Excellent for high-performance searches on large datasets where data doesn’t change often.
  + **Frequent searches**: If many searches are required on the same data, sorting once and using binary search saves time in the long run.
* **Why it's effective:**
  + Much **faster** than linear search on large datasets—**O(log n)** vs **O(n)**.
  + Reduces the number of comparisons drastically as it **halves the search space** each time.
  + Works best in **read-heavy scenarios**.

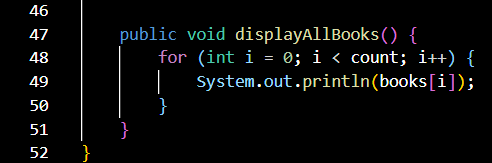
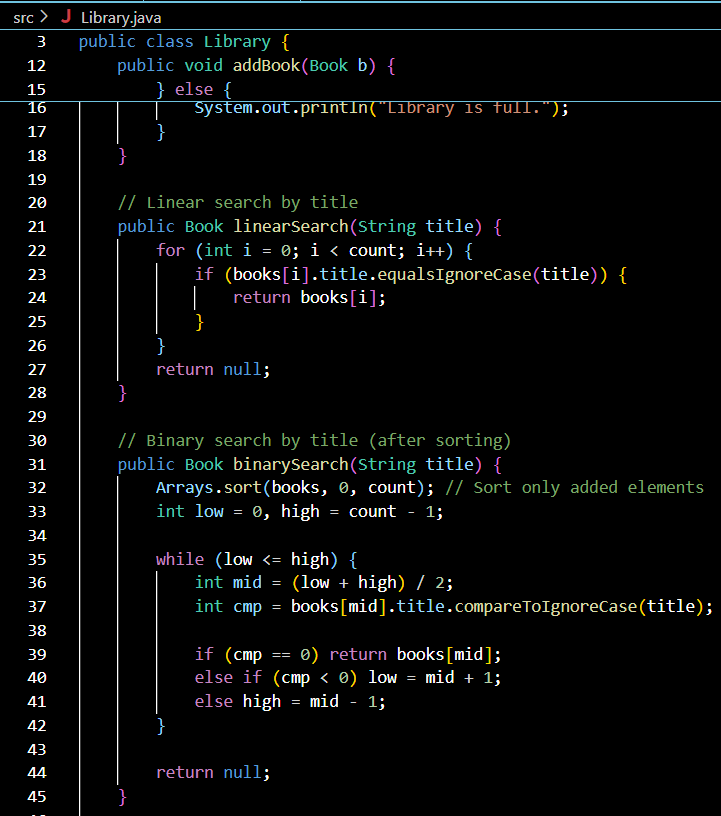
**FINAL CODE->**

**Book.java:**

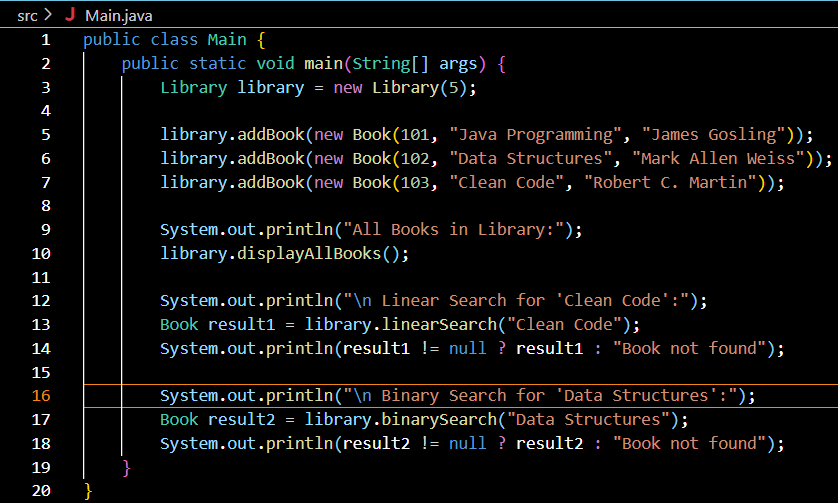
****

**Library.java:**

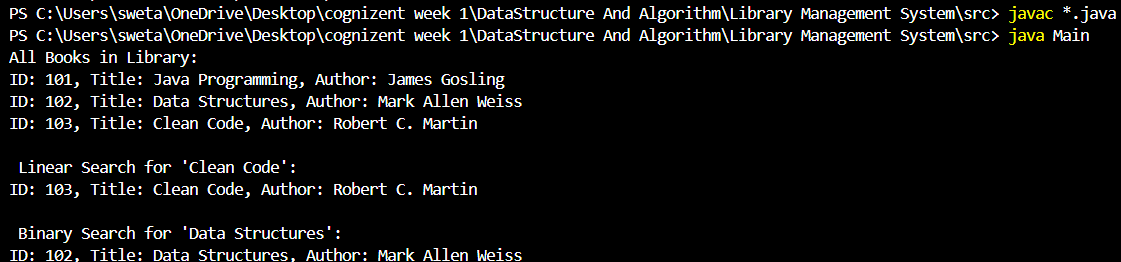
****

****

**Main.java:**

****

**Output:**

****

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.

**Recursion** is a technique where a method calls itself to solve smaller instances of a problem.

**Key Concepts**:

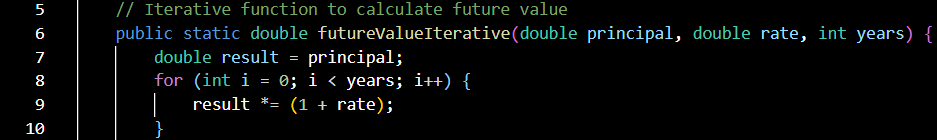
* **Base case**: When recursion stops.
* **Recursive case**: When the function continues to call itself.

Example Use Case in Finance:  
Predicting the **future value (FV)** of an investment:

FV = PresentValue × (1 + rate)^n  
This can be solved recursively as:  
FV(n) = FV(n - 1) × (1 + rate)

**2. Setup:**

* + Create a method to calculate the future value using a recursive approach.



1. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.



1. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.

**Time Complexity**

**Time complexity** of futureValue() is **O(n)**, where n is the number of years.

**Space complexity** is also **O(n)** due to recursive call stack.

* + Explain how to optimize the recursive solution to avoid excessive computation.

**Optimization Strategies**

**Memoization**: Store previously calculated results.

**Iterative Approach**: Convert to loop to avoid stack overflow for large n.

1. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.

The time complexity of a recursive algorithm depends on:

**Number of recursive calls**

**Work done in each call**

**Branching factor** (how many times the function calls itself per invocation)

**Depth of recursion**

* + Explain how to optimize the recursive solution to avoid excessive computation.

**FINAL CODE->**

**FinanceForecast.java:**



**Output:**

