**Exercise 1: Inventory Management System**

**Q. Explain why data structures and algorithms are essential in handling large inventories.**

Data structures and algorithms are important because they help store and manage large

amounts of inventory efficiently. With the right structures, we can quickly find, update, or remove a product without slowing down the system. Algorithms improve how we search or organize data, making the system faster and scalable as the inventory grows.

1. **Discuss the types of data structures suitable for this problem**

**-ArrayList**: Useful for ordered data but slow for searching, updating, or deleting (O(n) time).

**-HashMap**: Best choice for fast access using productId as the key. Allows add, update, and delete operations in O(1) average time.

**-TreeMap**: Maintains sorted order of products. Operations take O(log n) time, useful if sorted display is needed.

**InventoryManagementSystem.project**

**Product.java**

public class Product {

int productId;

String productName;

int quantity;

double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

@Override

public String toString() {

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

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

}

}

**Inventory.java**

import java.util.HashMap;

public class Inventory {

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

public void addProduct(Product product) {

inventory.put(product.productId, product);

}

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

if (inventory.containsKey(productId)) {

Product p = inventory.get(productId);

p.quantity = newQuantity;

p.price = newPrice;

} 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 displayInventory() {

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

System.out.println(p);

}

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Inventory inv = new Inventory();

Product p1 = new Product(1, "Monitor", 10, 12000.0);

Product p2 = new Product(2, "Keyboard", 20, 1500.0);

inv.addProduct(p1);

inv.addProduct(p2);

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

inv.displayInventory();

inv.updateProduct(1, 8, 11500.0);

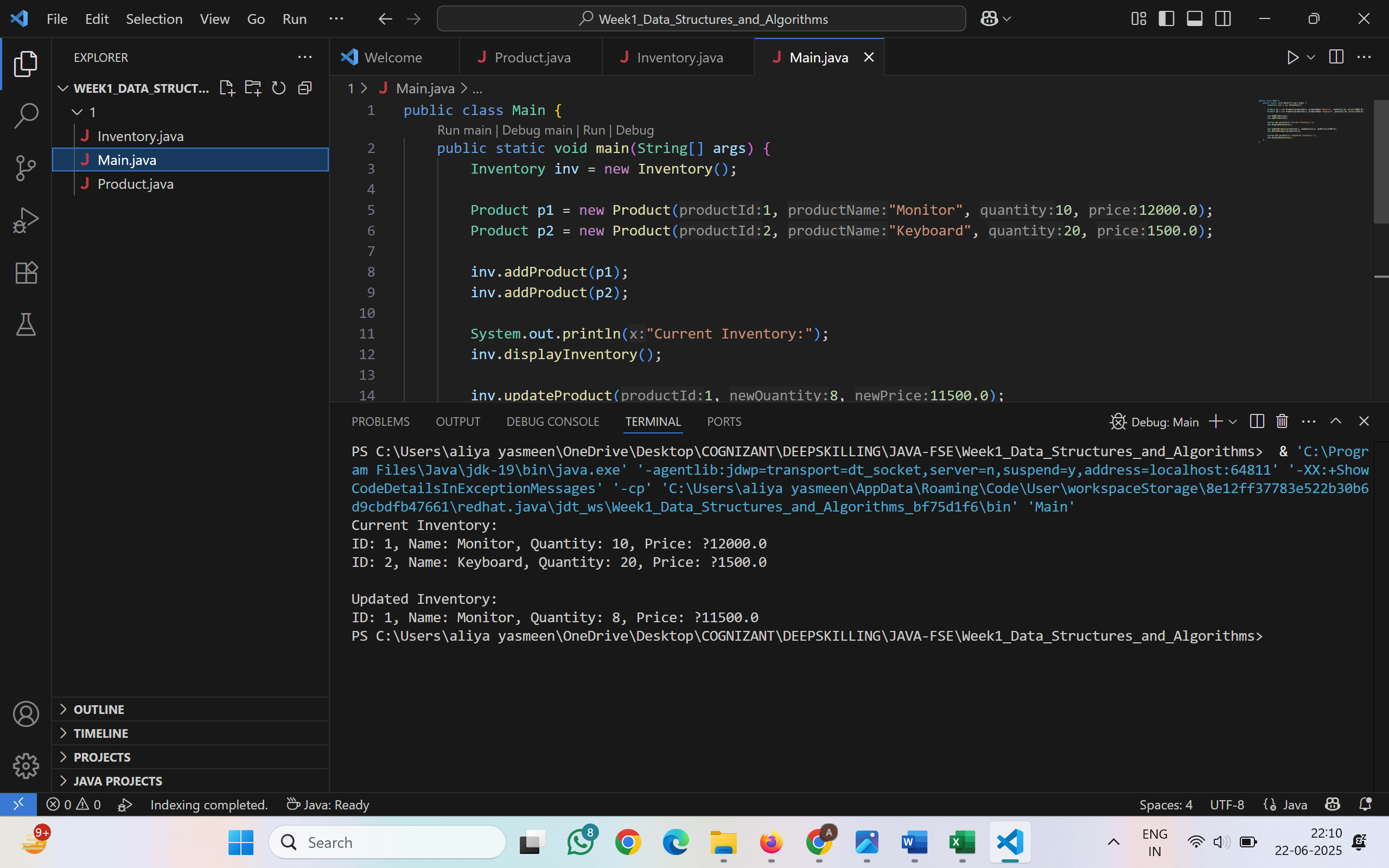
inv.deleteProduct(2);

System.out.println("\nUpdated Inventory:");

inv.displayInventory();

}

}

**Output:** 

**Analysis**

Time Complexity of Operations (Using HashMap)

* **Add Product**
  + Time Complexity: *O(1)* (average case)
  + Products are inserted using productId as a key, allowing constant-time insertion.
* **Update Product**
  + Time Complexity: *O(1)* (average case)
  + Products are updated directly by accessing their productId.
* **Delete Product**
  + Time Complexity: *O(1)* (average case)
  + Deletion is quick because items are removed using the key.
* **Display Inventory**
  + Time Complexity: *O(n)*
  + All product entries are iterated through for display.

**Optimization Suggestions**

* Use a TreeMap if sorted display of products (by productId or productName) is needed.
* Use ConcurrentHashMap if the inventory system is accessed by multiple threads (e.g., in multi-user applications).
* For large-scale inventory (e.g., 10,000+ products), integrate a database system like MySQL or MongoDB.

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

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

Big O notation describes how the performance (time or space) of an algorithm grows with the size of input data. It helps us understand how fast or slow an algorithm is, especially when the dataset becomes very large**.**

|  |  |  |  |
| --- | --- | --- | --- |
| 1. **Best Case**:   * The item is found immediately (e.g., first element). * Very fast performance. * Example:   + Linear Search → **O(1)**   + Binary Search → **O(1)** (item is in the middle)   **2.Average Case**:   * The item is located somewhere in the middle of the dataset. * Represents the expected time in typical conditions. * Example:   + Linear Search → **O(n/2)**   + Binary Search → **O(log n)**   **3.Worst Case**:   * The item is at the end of the list or not found at all. * Algorithm must check all possible positions. * Example:   + Linear Search → **O(n)**   + Binary Search → **O(log n)**   **E-CommercePlatformSearch project:**  **Product.java:**  public class Product {      int productId;      String productName;      int quantity;      double price;      public Product(int productId, String productName, int quantity, double price) {          this.productId = productId;          this.productName = productName;  this.quantity = quantity;          this.price = price;      }      @Override      public String toString() {          return "ID: " + productId +                 ", Name: " + productName +                 ", Quantity: " + quantity +                 ", Price: ₹" + price;      }  }  **SearchFunctions.java:**  public class SearchFunctions {      // Linear Search      public static Product linearSearch(Product[] products, String name) {          for (Product p : products) {              if (p.productName.equalsIgnoreCase(name)) {                  return p;              }          }          return null;      }      // Binary Search (requires sorted array)      public static Product binarySearch(Product[] products, String name) {          int left = 0;          int right = products.length - 1;          while (left <= right) {              int mid = (left + right) / 2;              int comparison = products[mid].productName.compareToIgnoreCase(name);              if (comparison == 0) {                  return products[mid];              } else if (comparison < 0) {                  left = mid + 1;              } else {                  right = mid - 1;              }          }          return null;      }  }  **Main.java:**  import java.util.Arrays;  import java.util.Comparator;  public class Main {      public static void main(String[] args) {          Product[] products = {              new Product(1, "Laptop", 5, 75000.0),              new Product(2, "Shoes", 20, 2999.99),              new Product(3, "Book", 50, 499.50),              new Product(4, "T-shirt", 15, 899.0)          };          System.out.println(" Linear Search Result:");          Product result1 = SearchFunctions.linearSearch(products, "Laptop");          System.out.println(result1 != null ? result1 : "Product not found");          Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));          System.out.println("\n Binary Search Result:");          Product result2 = SearchFunctions.binarySearch(products, "Laptop");          System.out.println(result2 != null ? result2 : "Product not found");      }  }  **Output:** |  |  |  |
|  |  |  |  |

**Analysis**

* + **Time Complexity Comparison**

1. **Linear Search**

Time complexity: **O(n)**

Suitable for: **Small or unsorted datasets**

Slower as the number of products increases.

**2.Binary Search**

Time complexity: **O(log n)**

Suitable for: **Large, sorted datasets**

Much faster due to halving the search space.

* + **Which Algorithm is Better and Why?**
* **Binary Search** is far more efficient for e-commerce platforms because:
  + Product lists are usually large (thousands to millions).
* **Linear Search** is only useful:
  + When data is small or dynamic.
  + When sorting is not possible.
* **In real systems**, even better structures are used:
  + Tries, HashMaps, or Elasticsearch for real-time search
  + Autocomplete suggestions, filtering, and ranking also use efficient algorithms.

**Exercise 3: Sorting Customer Orders**

**Q. Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).**

**1. Bubble Sort**:

* Compares adjacent elements and swaps them if they are in the wrong order.
* Time Complexity: **O(n²)** (inefficient for large data)

**2.Insertion Sort**:

* Builds the sorted array one element at a time by comparing and inserting.
* Time Complexity: **O(n²)** (good for small/partially sorted data)

**3.Quick Sort**:

* Divide-and-conquer method; selects a pivot and partitions the array.
* Time Complexity: **O(n log n)** average, **O(n²)** worst case

**4.Merge Sort**:

* Recursively splits the array and merges sorted halves.
* Time Complexity: **O(n log n)**

**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 ID: " + orderId +

               ", Customer: " + customerName +

               ", Total Price: ₹" + totalPrice;

    }

}

**OrderSorting.java**

public class OrderSorting {

    // Bubble Sort

    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) {

                    Order temp = orders[j];

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

                    orders[j+1] = temp;

                }

            }

        }

    }

    // Quick Sort

    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;

    }

}

**Customer.java:**

public class Customer {

    public static void main(String[] args) {

        Order[] orders = {

            new Order(1, "Aliya", 2500.00),

            new Order(2, "Sara", 4999.99),

            new Order(3, "John", 1299.00),

            new Order(4, "Neha", 3200.75)

        };

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

        for (Order o : orders) {

            System.out.println(o);

        }

        // Sort using Bubble Sort

        OrderSorting.bubbleSort(orders);

        System.out.println("\nSorted by Bubble Sort:");

        for (Order o : orders) {

            System.out.println(o);

        }

        // Re-initialize and sort using Quick Sort

        orders = new Order[] {

            new Order(1, "Aliya", 2500.00),

            new Order(2, "Sara", 4999.99),

            new Order(3, "John", 1299.00),

            new Order(4, "Neha", 3200.75)

        };

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

        System.out.println("\nSorted by Quick Sort:");

        for (Order o : orders) {

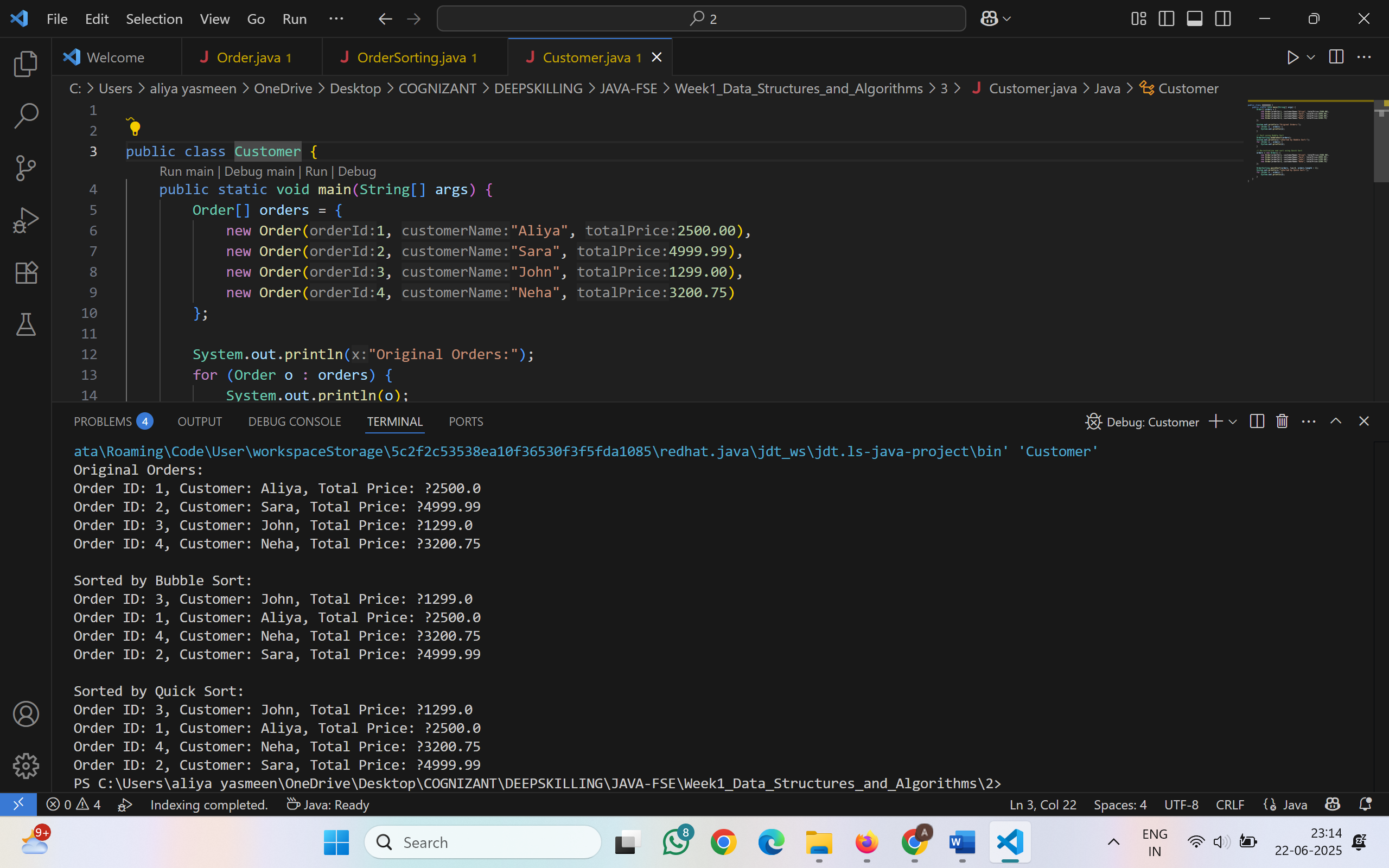
            System.out.println(o);

        }

    }

}

**Output:**



|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

**Analysis**

* **Bubble Sort**:
  + Time Complexity: O(n²)
  + Not efficient for large datasets
  + Easy to implement, best for educational use
* **Quick Sort**:
  + Time Complexity: O(n log n) average, O(n²) worst-case
  + Much faster for large datasets
  + Commonly used in real systems (or improved versions like Dual Pivot QuickSort)

**Exercise 4: Employee Management System**

**Q. Explain how arrays are represented in memory and their advantages.**

* Arrays are stored in contiguous memory locations, meaning each element is placed next to the previous one in memory.
* **Advantages of arrays:**
  + Fast access using index (O(1) time).
  + Easy to traverse.
  + Efficient for fixed-size collections.

**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;

    }

    @Override

    public String toString() {

        return "ID: " + employeeId +

               ", Name: " + name +

               ", Position: " + position +

               ", Salary: ₹" + salary;

    }

}

**EmployeeManager.java:**

public class EmployeeManager {

    Employee[] employees;

    int count;

    public EmployeeManager(int size) {

        employees = new Employee[size];

        count = 0;

    }

    // Add employee

    public void addEmployee(Employee emp) {

        if (count < employees.length) {

            employees[count++] = emp;

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

        } else {

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

        }

    }

    // Search employee by ID

    public Employee searchEmployee(int empId) {

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

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

                return employees[i];

            }

        }

        return null;

    }

    // Traverse all employees

    public void displayAllEmployees() {

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

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

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

        }

    }

    // Delete employee by ID

    public void deleteEmployee(int empId) {

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

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

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

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

                }

                employees[--count] = null;

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

                return;

            }

        }

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

    }

}

**Employees.java:**

public class Employees{

public static void main(String[] args) {

EmployeeManager manager = new EmployeeManager(5);

// Adding employees

manager.addEmployee(new Employee(101, "Aliya", "Developer", 60000));

manager.addEmployee(new Employee(102, "Ravi", "Tester", 45000));

manager.addEmployee(new Employee(103, "Sneha", "Manager", 75000));

// Display all employees

manager.displayAllEmployees();

// Search

System.out.println("\n Searching for employee ID 102:");

Employee emp = manager.searchEmployee(102);

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

// Delete

System.out.println("\n Deleting employee ID 101...");

manager.deleteEmployee(101);

// Display after deletion

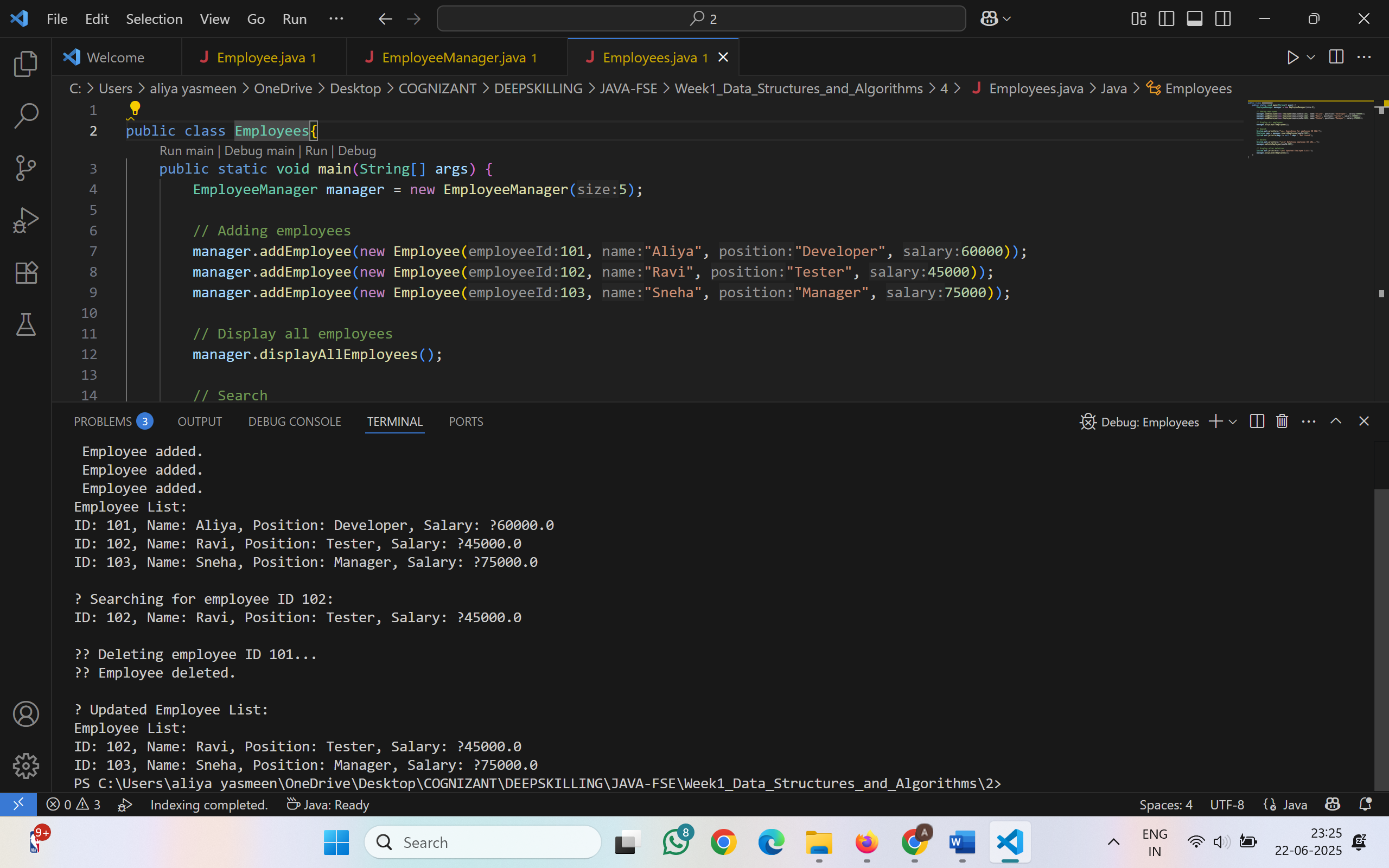
System.out.println("\n Updated Employee List:");

manager.displayAllEmployees();

}

}

**Output:**



**Analysis**

* Add employee → O(1)
* Search employee → O(n)
* Traverse all → O(n)
* Delete employee → O(n) (due to shifting)

**Limitations of Arrays**

* Fixed size — cannot grow dynamically.
* Insert/delete in middle = shifting elements → O(n)
* Better alternatives for dynamic data:
  + ArrayList (Java)
  + LinkedList (for frequent insert/delete)

**Exercise 5: Task Management System**

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

**1.** **Singly Linked List**:

* Each node points to the next node.
* Efficient for insertion/deletion at the beginning.
* One-directional traversal.

1. **Doubly Linked List**:

* Each node has two references: next and previous.
* Allows two-way traversal.
* Slightly more memory usage due to extra pointer.

**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 "Task ID: " + taskId +

", Name: " + taskName +

", Status: " + status;

}

}

class TaskNode {

    Task task;

    TaskNode next;

    public TaskNode(Task task) {

        this.task = task;

        this.next = null;

    }

}

**TaskManager.java:**

public class TaskManager {

    private TaskNode head;

    // Add task at end

    public void addTask(Task task) {

        TaskNode newNode = new TaskNode(task);

        if (head == null) {

            head = newNode;

        } else {

            TaskNode temp = head;

            while (temp.next != null) {

                temp = temp.next;

            }

            temp.next = newNode;

        }

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

    }// Search task by ID

    public Task searchTask(int taskId) {

        TaskNode temp = head;

        while (temp != null) {

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

                return temp.task;

            }

            temp = temp.next;

        }

        return null;

    }

    // Display all tasks

    public void displayTasks() {

        TaskNode temp = head;

        if (temp == null) {

            System.out.println(" No tasks to display.");

            return;

        }

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

        while (temp != null) {

            System.out.println(temp.task);

            temp = temp.next;

        }

    }

    // Delete task by ID

    public void deleteTask(int taskId) {

        if (head == null) {

            System.out.println(" Task list is empty.");

            return;

        }

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

            head = head.next;

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

            return;

        }

        TaskNode prev = head;

        TaskNode current = head.next;

        while (current != null) {

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

                prev.next = current.next;

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

                return;

            }

            prev = current;

            current = current.next;

        }

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

    }

}

**Manager.java:**

public class manager {

    public static void main(String[] args) {

        TaskManager manager = new TaskManager();

        // Add tasks

        manager.addTask(new Task(201, "Design Homepage", "Pending"));

        manager.addTask(new Task(202, "Write API", "In Progress"));

        manager.addTask(new Task(203, "Test Module", "Pending"));

        // Display all tasks

        manager.displayTasks();

        // Search for a task

        System.out.println("\n Searching for task ID 202:");

        Task found = manager.searchTask(202);

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

        // Delete a task

        System.out.println("\n Deleting task ID 201...");

        manager.deleteTask(201);

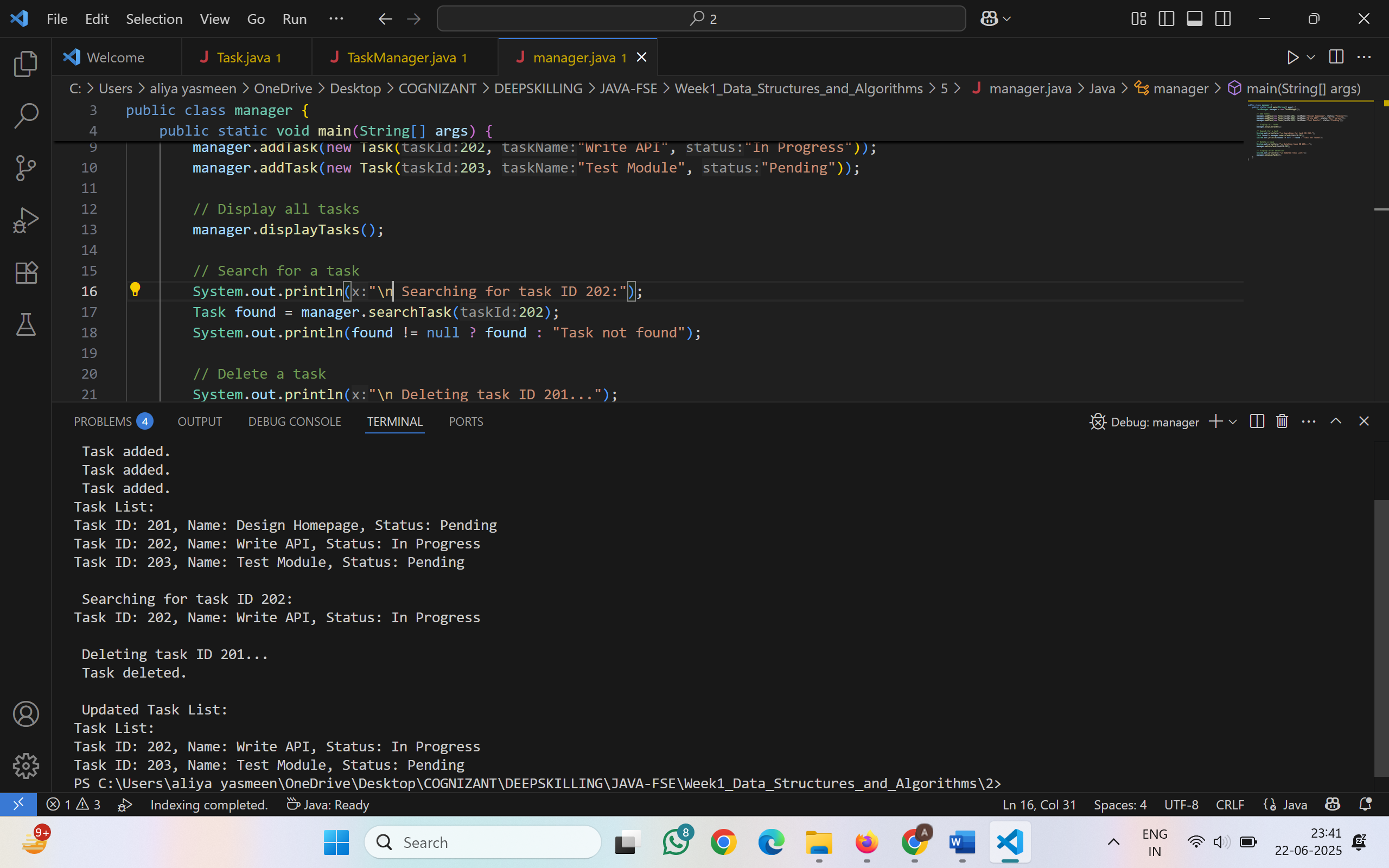
        System.out.println("\n Updated Task List:");

        manager.displayTasks();

    }

}

**Output:**



**Analysis of Operations**

* Add Task → O(n) (at end), O(1) if at head
* Search Task → O(n)
* Traverse Tasks → O(n)
* Delete Task → O(n)

**Advantages of Linked Lists over Arrays**

* Dynamic size — no need to define max size.
* Efficient for insertion and deletion (no shifting).
* Memory allocated as needed.

**Exercise 6: Library Management System**

**Q.Explain linear search and binary search algorithms.**

1. **Linear Search**:

* Checks each element one by one.
* Time Complexity: **O(n)**
* Suitable for **unsorted or small datasets**.

2. **Binary Search**:

* Divides the array into halves to find the target.
* Time Complexity: **O(log n)**
* Requires the array to be **sorted**.
* Much faster for large datasets.

**Book.java:**

public class Book {

int bookId;

String title;

String author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

@Override

public String toString() {

return "Book ID: " + bookId + ", Title: " + title + ", Author: " + author;

}

}

**Library.java:**

public class Library {

    public static void main(String[] args) {

        Book[] books = {

            new Book(1, "Java Programming", "James Gosling"),

            new Book(2, "Data Structures", "Mark Allen"),

            new Book(3, "Operating Systems", "Silberschatz"),

            new Book(4, "Computer Networks", "Tanenbaum")

        };

        System.out.println(" Linear Search for 'Data Structures':");

        Book found1 = LibrarySearch.linearSearch(books, "Data Structures");

        System.out.println(found1 != null ? found1 : "Not found");

        System.out.println("\n Binary Search for 'Operating Systems':");

        Book found2 = LibrarySearch.binarySearch(books, "Operating Systems");

        System.out.println(found2 != null ? found2 : "Not found");

       System.out.println("\n Binary Search for 'Python Basics':");

        Book found3 = LibrarySearch.binarySearch(books, "Python Basics");

        System.out.println(found3 != null ? found3 : "Not found");

    }

}

**LibrarySearch.java:**

import java.util.Arrays;

import java.util.Comparator;

public class LibrarySearch {

    // Linear Search

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

        for (Book b : books) {

            if (b.title.equalsIgnoreCase(title)) {

                return b;

            }

        }

        return null;

    }

    // Binary Search (requires sorted array by title)

    public static Book binarySearch(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;

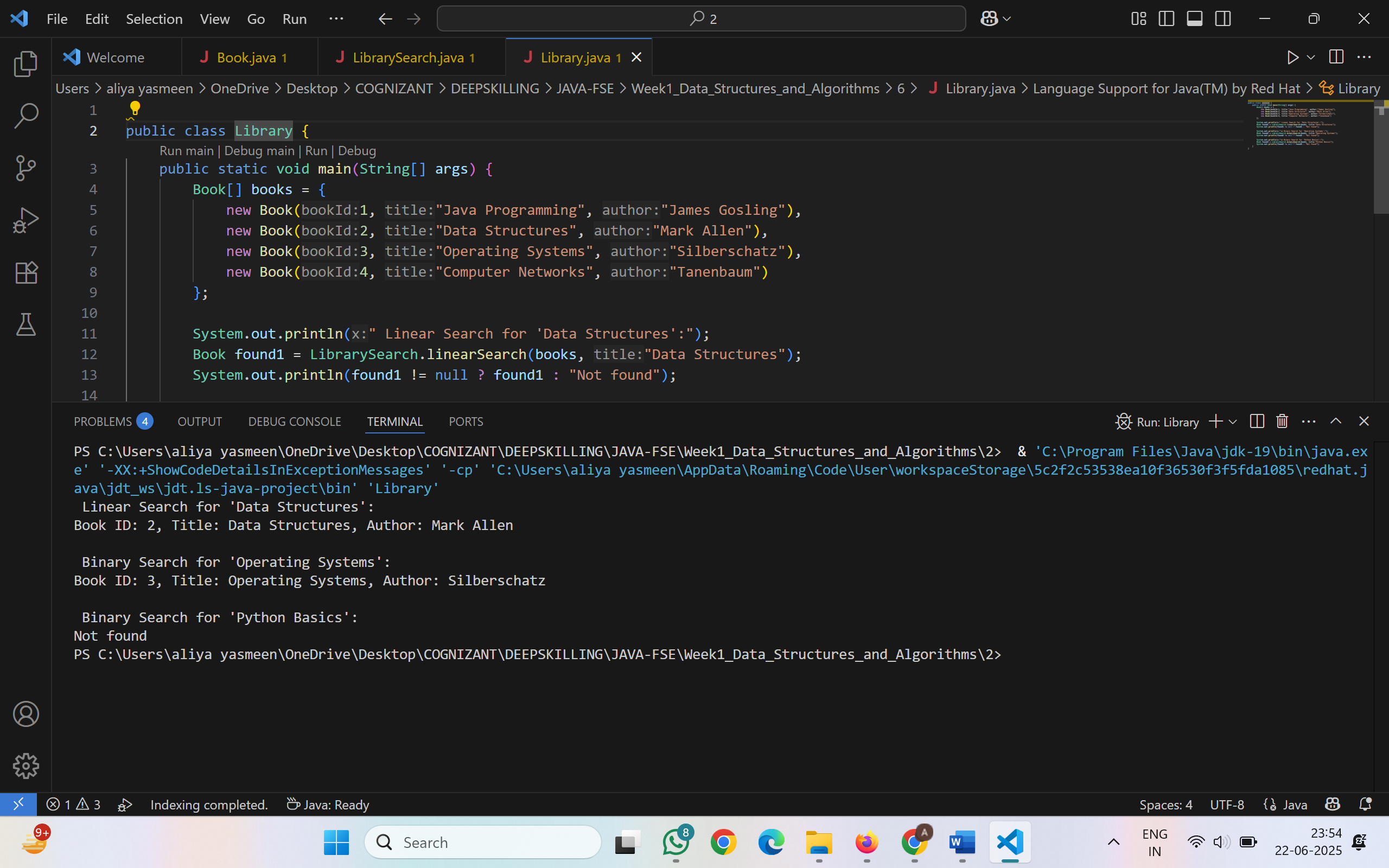
        }

        return null;

    }

}

**Output:**



**Analysis**

* **Linear Search**
  + Time Complexity: O(n)
  + Best for small or unsorted lists
  + No preprocessing needed
* **Binary Search**
  + Time Complexity: O(log n)
  + Best for large, sorted data
  + Sorting step required (O(n log n))

**When to Use Which?**

* Use **linear search** if:
  + Book list is small
  + New books are frequently added (unsorted)
* Use **binary search** if:
  + Book list is large
  + Sorted order is maintained (e.g., by title or author)

**Exercise 7: Financial Forecasting**

**Q.Explain the concept of recursion and how it can simplify certain problems.**

* Recursion is a programming technique where a method calls itself to solve smaller subproblems.
* It simplifies problems that can be broken into similar subproblems.
* Common use cases: factorial, Fibonacci, tree traversals, forecasting, etc.
* **Real Example: Financial Forecasting**
* **Formula**:  
  FutureValue(years) = CurrentValue × (1 + growthRate)^years

**Forecast.java:**

public class Forecast {

    // Recursive function to calculate future value

    public static double predictValue(double currentValue, double growthRate, int years) {

        if (years == 0) {

            return currentValue;

        } else {

            return predictValue(currentValue \* (1 + growthRate), growthRate, years - 1);

        }

    }

}

**FinancialForecast.java:**

public class FinancialForecast{

    public static void main(String[] args) {

        double currentValue = 10000;

        double growthRate = 0.1;

        int years = 5;

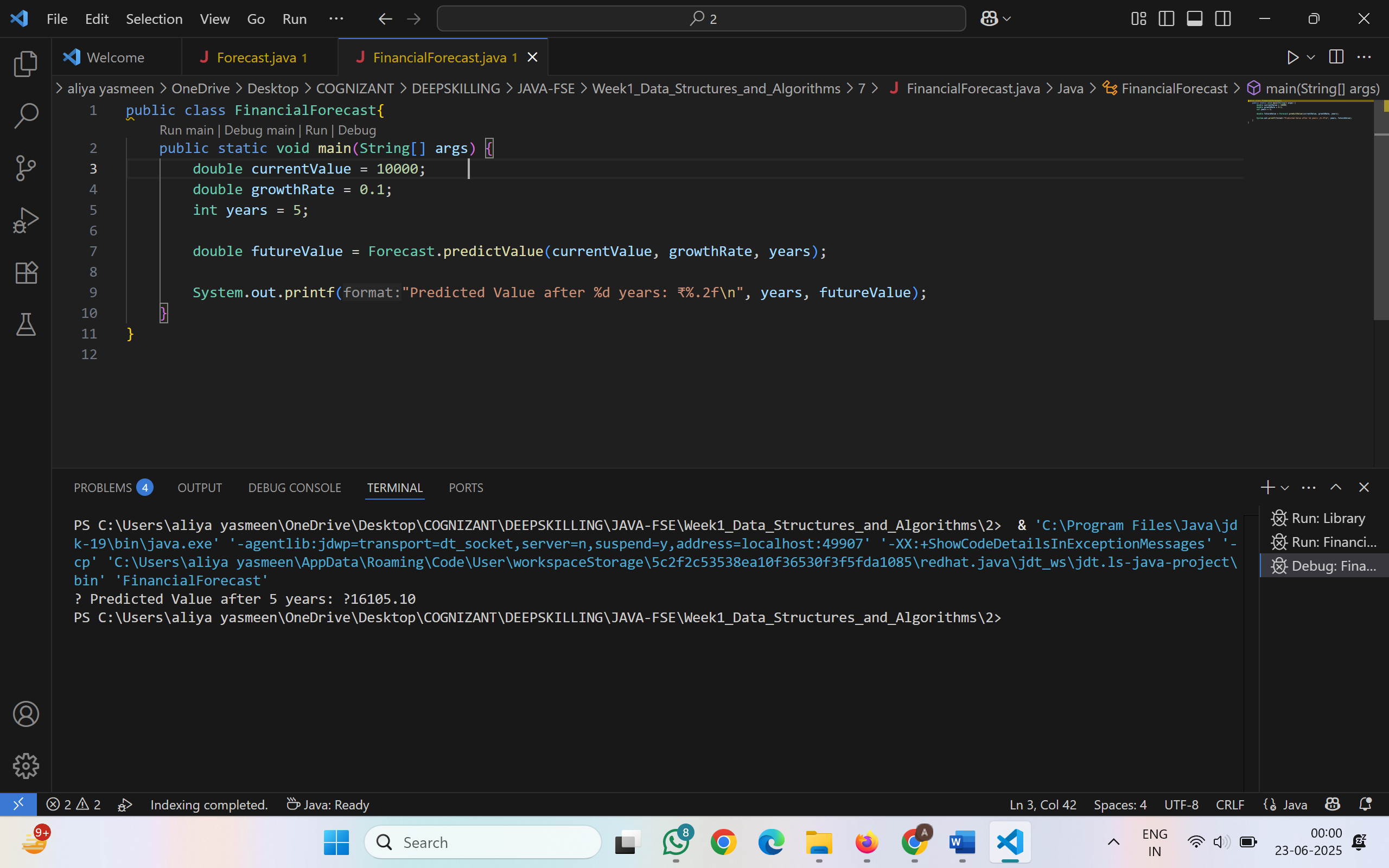
        double futureValue = Forecast.predictValue(currentValue, growthRate, years);

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

    }

}

**Output:**



**Analysis**

* **Time Complexity**: Recursive method: O(n) → one call for each year
* **Space Complexity**: O(n) due to function call stack (can lead to stack overflow for large n)

**Optimization Techniques**

* **Use Iteration** instead of recursion for large datasets.
* **Memorization**: Store results of previous calls to avoid re-computation.
* **Tail Recursion** (where supported): Compiler can optimize recursion better.