**ALGORITHMS AND DATA STRUCTURES**

**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.
   * Discuss the types of data structures suitable for this problem.
2. Setup:
   * Create a new project for the inventory management system.
3. 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.
4. Analysis:
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
   * Discuss how you can optimize these operations.

**SOLUTION:**

import java.util.HashMap;

import java.util.Scanner;

class Product {

private String productId;

private String productName;

private int quantity;

private double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public String getProductId() {

return productId;

}

public void setProductName(String name) {

this.productName = name;

}

public void setQuantity(int qty) {

this.quantity = qty;

}

public void setPrice(double price) {

this.price = price;

}

public void display() {

System.out.println("ID: " + productId + ", Name: " + productName +

", Qty: " + quantity + ", Price: ₹" + price);

}

}

class InventoryManager {

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

public void addProduct(Product product) {

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

}

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

Product p = inventory.get(productId);

if (p != null) {

p.setProductName(name);

p.setQuantity(qty);

p.setPrice(price);

} else {

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

}

}

public void deleteProduct(String productId) {

if (inventory.remove(productId) == null) {

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

}

}

public void displayAll() {

if (inventory.isEmpty()) {

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

} else {

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

p.display();

}

}

}

}

public class InventoryManagementSystem {

public static void main(String[] args) {

InventoryManager manager = new InventoryManager();

manager.addProduct(new Product("P001", "Laptop", 10, 75000));

manager.addProduct(new Product("P002", "Keyboard", 50, 1200));

manager.displayAll();

manager.updateProduct("P002", "Mechanical Keyboard", 40, 1500);

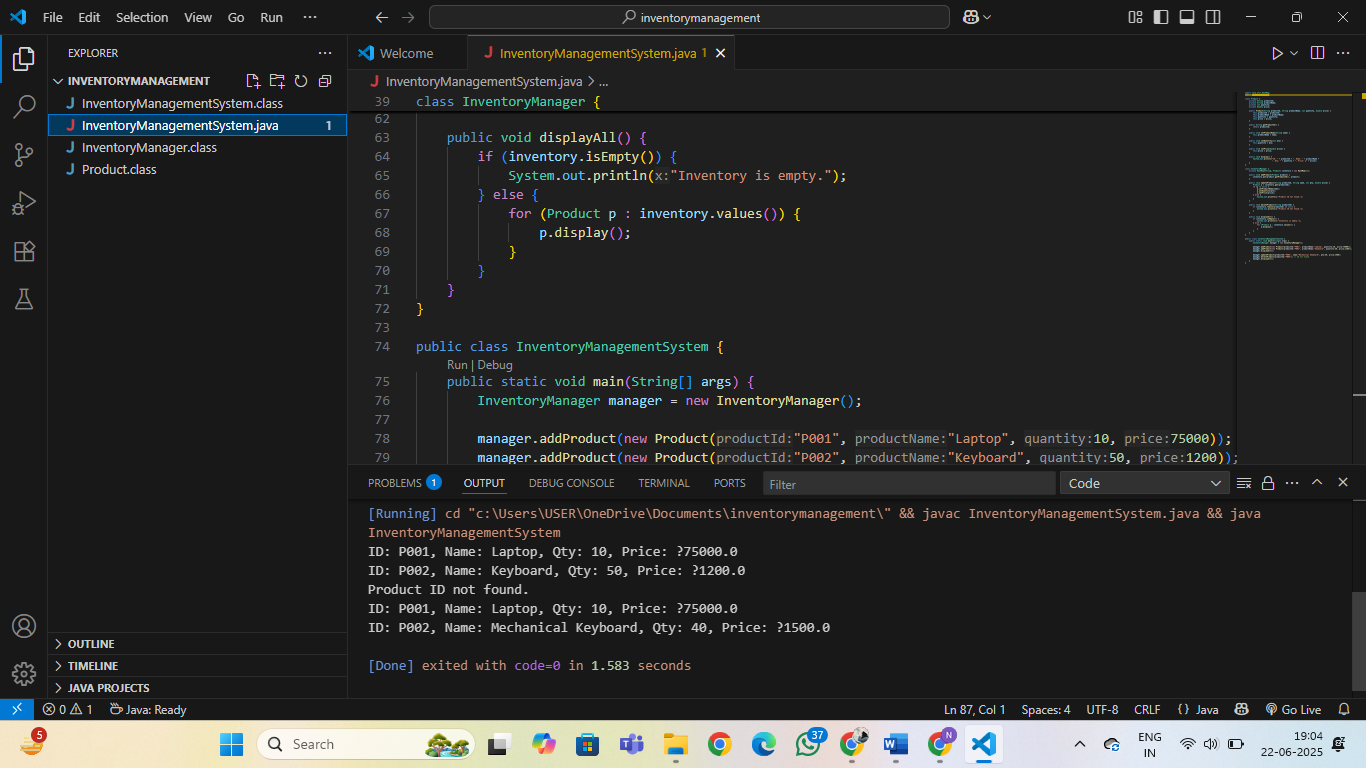
manager.deleteProduct("P003"); // ID not found

manager.displayAll();

}

}

**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.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**SOLUTION:**

**Product Class:**

public class Product {

    int productId;

    String productName;

    String category;

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

        this.productId = productId;

        this.productName = productName;

        this.category = category;

    }

    public String toString() {

        return productId + " - " + productName + " (" + category + ")";

    }

}

**Linear Search Class:**

public class LinearSearch {

    public static Product search(Product[] products, String name) {

        for (Product product : products) {

            if (product.productName.equalsIgnoreCase(name)) {

                return product;

            }

        }

        return null;

    }

}

**Binary Search Class:**

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

    public static Product search(Product[] products, String name) {

        // Binary search assumes the array is already sorted

        int left = 0;

        int right = products.length - 1;

        while (left <= right) {

            int mid = (left + right) / 2;

            int cmp = products[mid].productName.compareToIgnoreCase(name);

            if (cmp == 0) {

                return products[mid];

            } else if (cmp < 0) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return null;

    }

}

**Main Class:**

import java.util.Comparator;

import java.util.Arrays;

public class Main {

    public static void main(String[] args) {

        Product[] products = {

            new Product(101, "Shoes", "Fashion"),

            new Product(102, "Laptop", "Electronics"),

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

            new Product(104, "Watch", "Accessories"),

            new Product(105, "Bag", "Fashion")

        };    // LINEAR SEARCH

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

        Product result1 = LinearSearch.search(products, "Mobile");

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

  // Sort for binary search

        Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

    // BINARY SEARCH

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

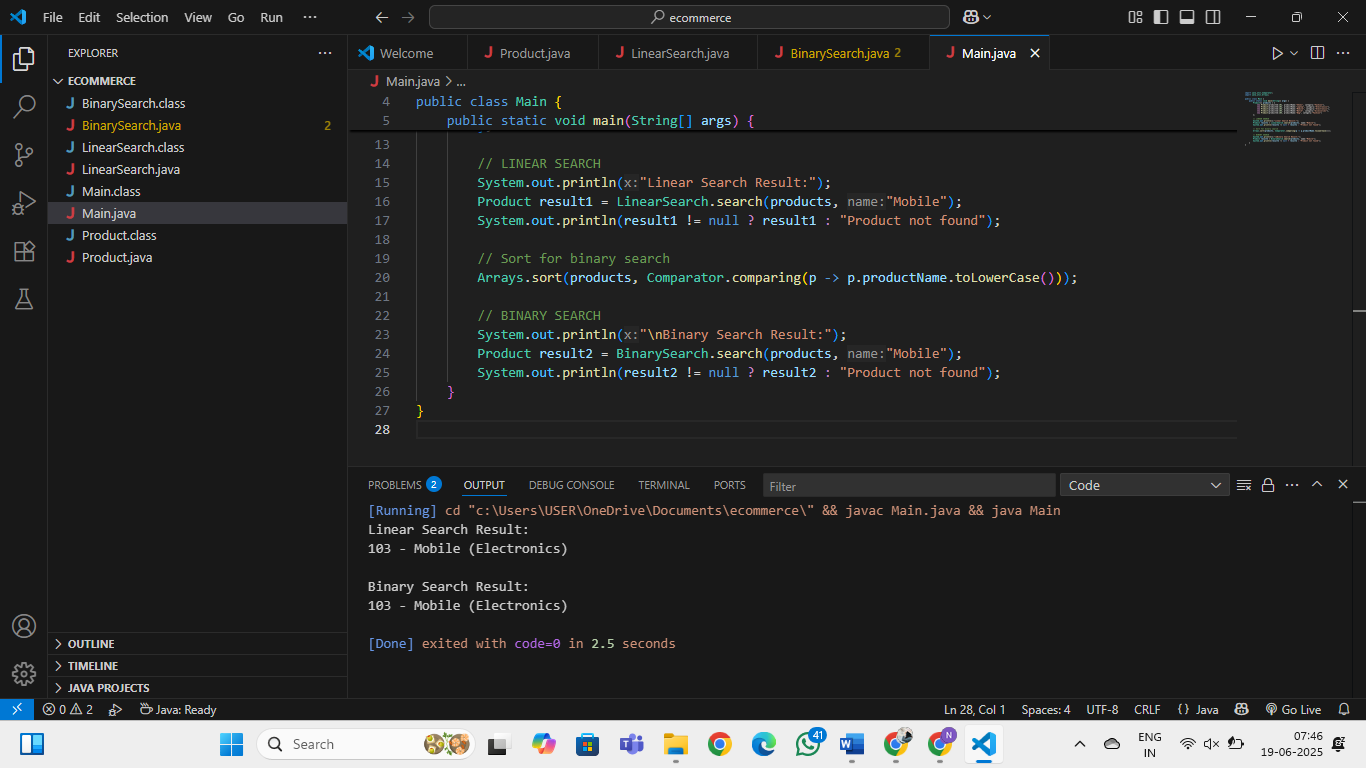
        Product result2 = BinarySearch.search(products, "Mobile");

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

  }

}

**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).
2. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   * Discuss why Quick Sort is generally preferred over Bubble Sort.

**SOLUTION:**

**Order Class:**

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 void display() {

        System.out.println(orderId + " | " + customerName + " | ₹" + totalPrice);

    }

}

**BubbleSort Class:**

class BubbleSort {

    public static void sort(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;

                }

            }

        }

    }

}

**QuickSort Class:**

class QuickSort {

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

        if (low < high) {

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

            sort(orders, low, pi - 1);

            sort(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;

    }

}

**Main Class:**

public class SortingCustomerOrders {

    public static void main(String[] args) {

        Order[] orders = {

            new Order("O101", "Alice", 5200),

            new Order("O102", "Bob", 1500),

            new Order("O103", "Charlie", 9800),

            new Order("O104", "David", 3200),

            new Order("O105", "Eva", 15000)

        };

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

        for (Order o : orders) o.display();

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

        BubbleSort.sort(orders);

        for (Order o : orders) o.display();

        Order[] orders2 = {

            new Order("O101", "Alice", 5200),

            new Order("O102", "Bob", 1500),

            new Order("O103", "Charlie", 9800),

            new Order("O104", "David", 3200),

            new Order("O105", "Eva", 15000)

        };

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

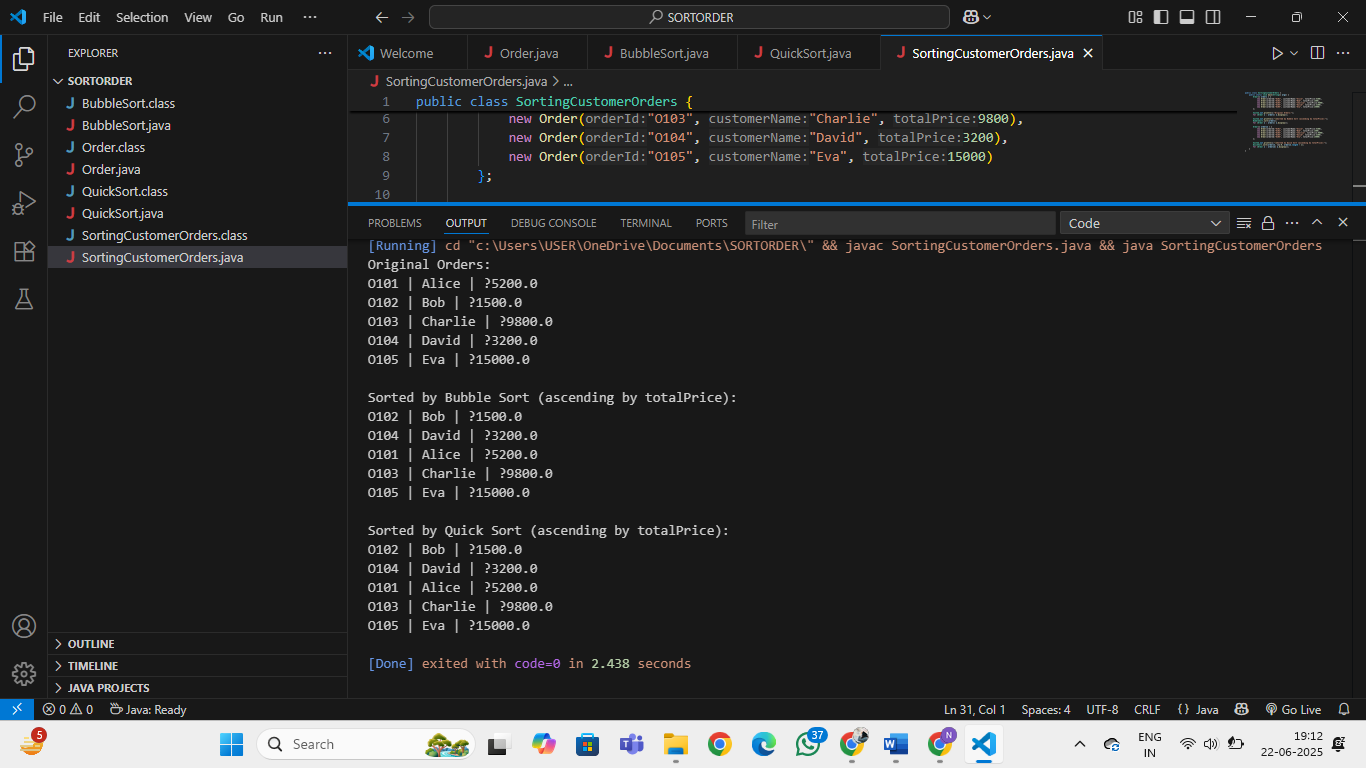
        QuickSort.sort(orders2, 0, orders2.length - 1);

        for (Order o : orders2) o.display();

    }

}

**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.
2. **Setup:**
   * Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
3. **Implementation:**
   * Use an array to store employee records.
   * Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).
   * Discuss the limitations of arrays and when to use them.

**SOLUTION:**

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 void display() {

System.out.println(employeeId + " | " + name + " | " + position + " | ₹" + salary);

}

}

class EmployeeManager {

private Employee[] employees;

private int size;

public EmployeeManager(int capacity) {

employees = new Employee[capacity];

size = 0;

}

public void addEmployee(Employee emp) {

if (size < employees.length) {

employees[size++] = emp;

} else {

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

}

}

public void searchEmployee(String empId) {

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

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

employees[i].display();

return;

}

}

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

}

public void traverseEmployees() {

if (size == 0) {

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

} else {

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

employees[i].display();

}

}

}

public void deleteEmployee(String empId) {

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

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

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

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

}

employees[--size] = null;

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

return;

}

}

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

}

}

public class EmployeeManagementSystem {

public static void main(String[] args) {

EmployeeManager manager = new EmployeeManager(5);

manager.addEmployee(new Employee("E001", "Ravi", "Manager", 75000));

manager.addEmployee(new Employee("E002", "Divya", "Developer", 55000));

manager.addEmployee(new Employee("E003", "Arjun", "Tester", 40000));

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

manager.traverseEmployees();

System.out.println("\nSearch Employee E002:");

manager.searchEmployee("E002");

System.out.println("\nDelete Employee E001:");

manager.deleteEmployee("E001");

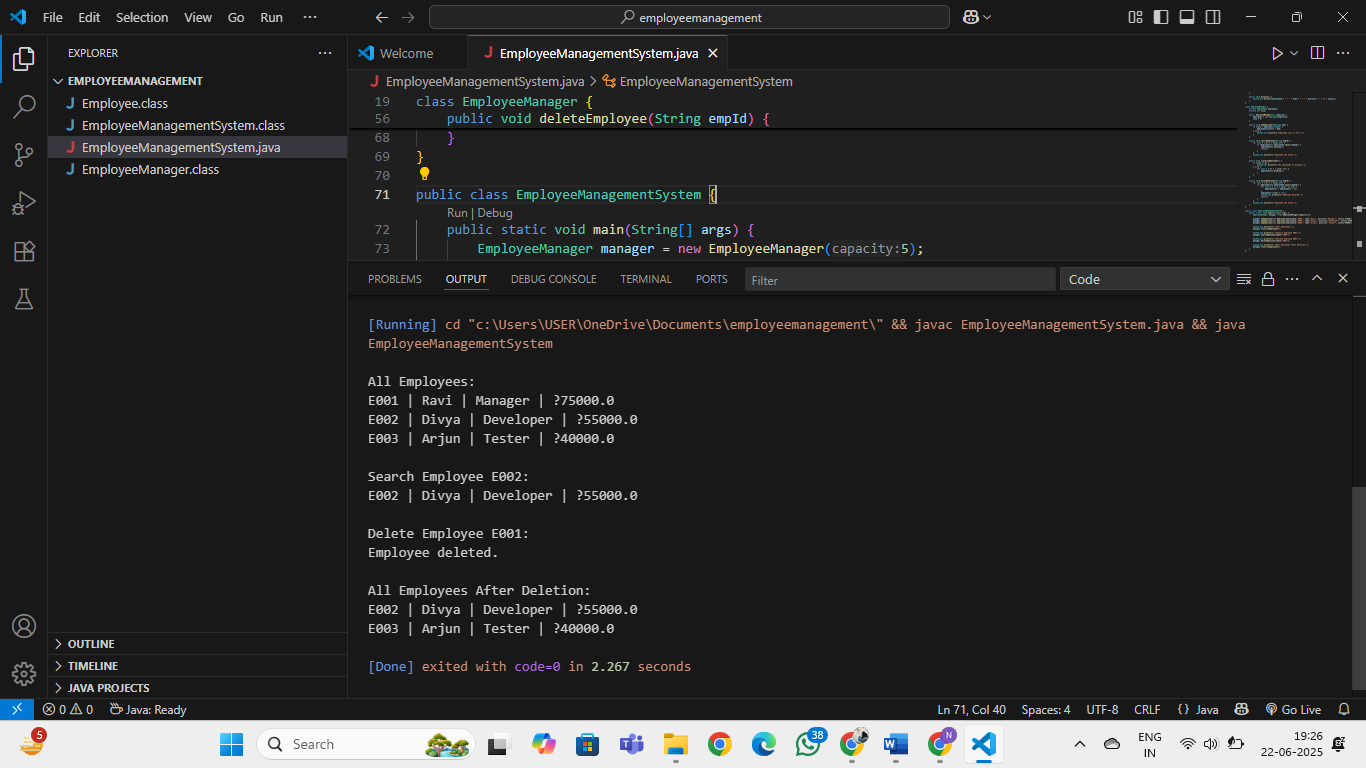
System.out.println("\nAll Employees After Deletion:");

manager.traverseEmployees();

}

}

OUTPUT:



**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).
2. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
3. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.
4. **Analysis:**
   * Analyze the time complexity of each operation.
   * Discuss the advantages of linked lists over arrays for dynamic data.

**SOLUTION:**

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 void display() {

System.out.println(taskId + " | " + taskName + " | " + status);

}

}

class Node {

Task task;

Node next;

public Node(Task task) {

this.task = task;

this.next = null;

}

}

class TaskManager {

private Node head;

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;

}

}

public void traverseTasks() {

if (head == null) {

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

return;

}

Node temp = head;

while (temp != null) {

temp.task.display();

temp = temp.next;

}

}

public void searchTask(String taskId) {

Node temp = head;

while (temp != null) {

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

temp.task.display();

return;

}

temp = temp.next;

}

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

}

public void deleteTask(String taskId) {

if (head == null) {

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

return;

}

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

head = head.next;

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

return;

}

Node prev = head;

Node curr = head.next;

while (curr != null) {

if (curr.task.taskId.equals(taskId)) {

prev.next = curr.next;

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

return;

}

prev = curr;

curr = curr.next;

}

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

}

}

public class TaskManagementSystem {

public static void main(String[] args) {

TaskManager manager = new TaskManager();

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

manager.addTask(new Task("T002", "Fix Login Bug", "In Progress"));

manager.addTask(new Task("T003", "Update Database", "Completed"));

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

manager.traverseTasks();

System.out.println("\nSearch Task T002:");

manager.searchTask("T002");

System.out.println("\nDelete Task T001:");

manager.deleteTask("T001");

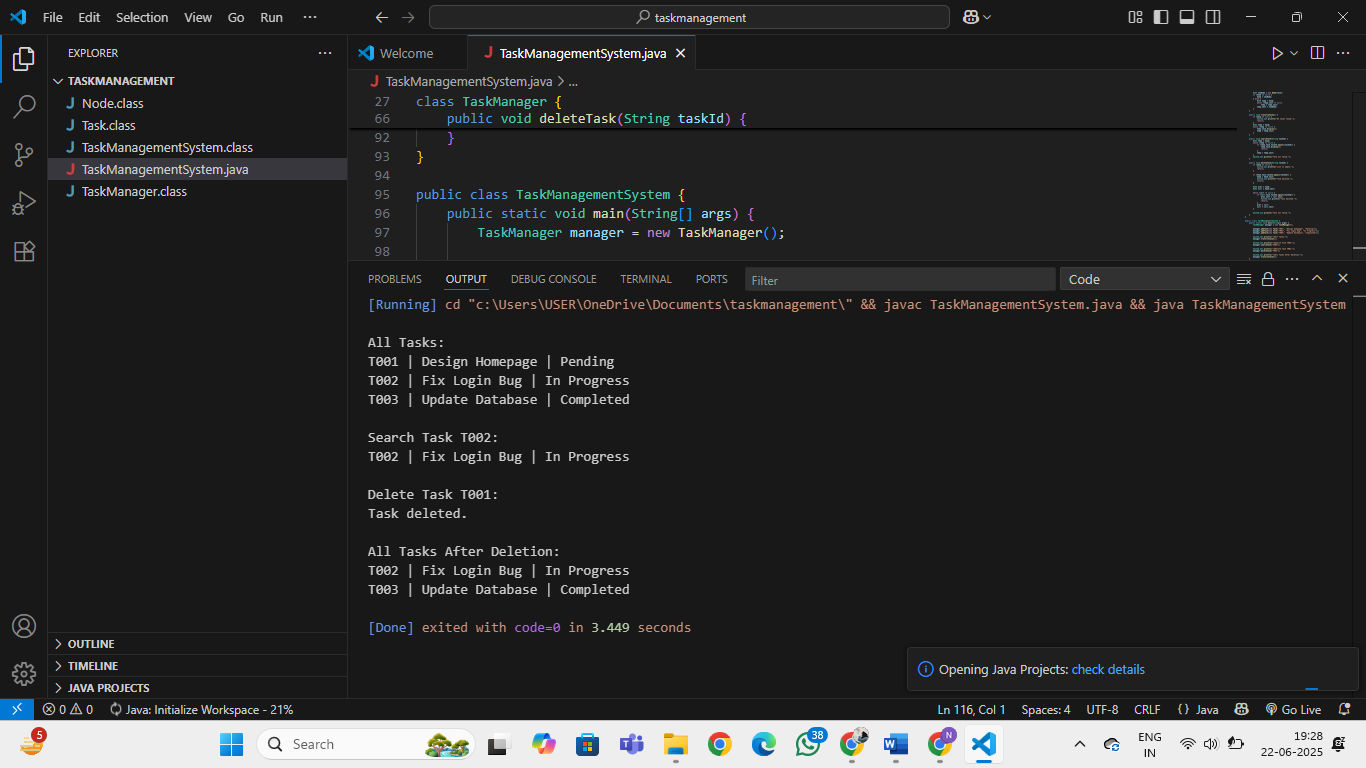
System.out.println("\nAll Tasks After Deletion:");

manager.traverseTasks();

}

}

**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.
2. **Setup:**
   * Create a class **Book** with attributes like **bookId**, **title**, and **author**.
3. **Implementation:**
   * Implement linear search to find books by title.
   * Implement binary search to find books by title (assuming the list is sorted).
4. **Analysis:**
   * Compare the time complexity of linear and binary search.
   * Discuss when to use each algorithm based on the data set size and order.

**SOLUTION:**

import java.util.Arrays;

import java.util.Comparator;

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 void display() {

System.out.println(bookId + " | " + title + " | " + author);

}

}

class Library {

Book[] books;

int size;

public Library(int capacity) {

books = new Book[capacity];

size = 0;

}

public void addBook(Book book) {

if (size < books.length) {

books[size++] = book;

}

}

public void linearSearch(String title) {

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

boolean found = false;

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

if (books[i].title.equalsIgnoreCase(title)) {

books[i].display();

found = true;

}

}

if (!found) {

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

}

}

public void binarySearch(String title) {

System.out.println("Binary Search Result:");

Arrays.sort(books, 0, size, Comparator.comparing(b -> b.title.toLowerCase()));

int left = 0, right = size - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0) {

books[mid].display();

return;

} else if (cmp < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

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

}

public void displayAllBooks() {

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

books[i].display();

}

}

}

public class LibraryManagementSystem {

public static void main(String[] args) {

Library library = new Library(10);

library.addBook(new Book("B001", "Java Programming", "James Gosling"));

library.addBook(new Book("B002", "Python Basics", "Guido van Rossum"));

library.addBook(new Book("B003", "C++ Guide", "Bjarne Stroustrup"));

library.addBook(new Book("B004", "Data Structures", "Mark Allen Weiss"));

library.addBook(new Book("B005", "Python Basics", "John Doe"));

System.out.println("All Books in Library:");

library.displayAllBooks();

System.out.println("\nSearching for 'Python Basics' using Linear Search:");

library.linearSearch("Python Basics");

System.out.println("\nSearching for 'Java Programming' using Binary Search:");

library.binarySearch("Java Programming");

System.out.println("\nSearching for 'Artificial Intelligence' using Linear Search:");

library.linearSearch("Artificial Intelligence");

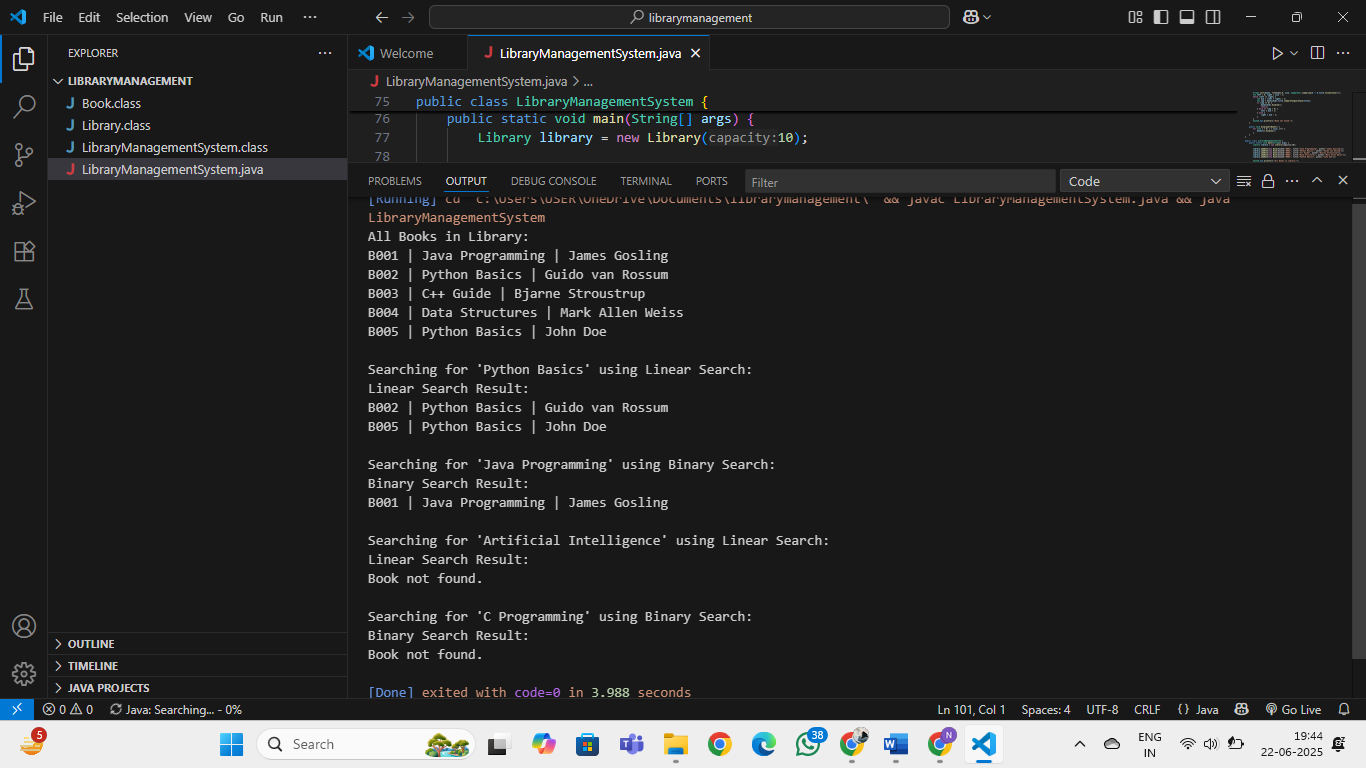
System.out.println("\nSearching for 'C Programming' using Binary Search:");

library.binarySearch("C Programming");

}

}

**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.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**SOLUTION:**

**Forecast Class:**

public class Forecast {

    // Recursive method to calculate future value

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

        if (years == 0) {

            return presentValue;

        }

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

    }

    // Optimized version using memoization (optional)

    public static double calculateFutureValueMemo(double presentValue, double growthRate, int years, double[] memo) {

        if (years == 0) {

            return presentValue;

        }

        if (memo[years] != 0) {

            return memo[years];

        }

        memo[years] = calculateFutureValueMemo(presentValue, growthRate, years - 1, memo) \* (1 + growthRate);

        return memo[years];

    }

}

**Main Class:**

public class Main {

    public static void main(String[] args) {

        double presentValue = 1000.0;      // Starting investment

        double growthRate = 0.10;          // 10% growth

        int years = 5;

        double futureValue = Forecast.calculateFutureValue(presentValue, growthRate, years);

        System.out.printf("Future Value (recursive): %.2f\n", futureValue);

        double[] memo = new double[years + 1];

        double optimizedValue = Forecast.calculateFutureValueMemo(presentValue, growthRate, years, memo);

        System.out.printf("Future Value (memoized): %.2f\n", optimizedValue);

    }

}

**OUTPUT:**

