**Exercise 1: Inventory Management System**

//Product.java  
  
public class Product {

    private String product\_id;

    private String product\_name;

    private int quantity;

    private double price;

    public Product(String product\_id, String product\_name, int quantity, double price) {

        this.product\_id = product\_id;

        this.product\_name = product\_name;

        this.quantity = quantity;

        this.price = price;

    }

    public String getProduct\_Id() {

        return product\_id;

    }

    public void setProductId(String product\_id) {

        this.product\_id = product\_id;

    }

    public String getProductName() {

        return product\_name;

    }

    public void setProduct\_Name(String product\_name) {

        this.product\_name = product\_name;

    }

    public int getQuantity() {

        return quantity;

    }

    public void setQuantity(int quantity) {

        this.quantity = quantity;

    }

    public double getPrice() {

        return price;

    }

    public void setPrice(double price) {

        this.price = price;

    }

    @Override

    public String toString() {

        return "Product [product\_id=" + product\_id + ", product\_name=" + product\_name + ", quantity=" + quantity + ", price=" + price + "]";

    }

}

//InventoryManagementSystem.java  
  
import java.util.HashMap;

import java.util.Map;

public class InventoryManagementSystem {

    private Map<String, Product> inventory;

    public InventoryManagementSystem() {

        this.inventory = new HashMap<>();

    }

    // Add your products here

    public void add\_Product(Product product) {

        inventory.put(product.getProduct\_Id(), product);

    }

    // Update your product here

    public void update\_Product(String product\_id, String new\_name, int new\_quantity, double new\_price) {

        Product product = inventory.get(product\_id);

        if (product != null) {

            product.setProduct\_Name(new\_name);

            product.setQuantity(new\_quantity);

            product.setPrice(new\_price);

        }

    }

    // Delete your product here

    public void delete\_Product(String product\_id) {

        inventory.remove(product\_id);

    }

    // Get your product here

    public Product get\_Product(String product\_id) {

        return inventory.get(product\_id);

    }

    // Print here

    public void print\_Inventory() {

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

            System.out.println(product);

        }

    }

    public static void main(String[] args) {

        InventoryManagementSystem ims = new InventoryManagementSystem();

        Product product1 = new Product("101", "CLRS Book", 15, 1524.0);

        Product product2 = new Product("102", "Java Book", 25, 750.0);

        ims.add\_Product(product1);

        ims.add\_Product(product2);

        System.out.println("\nInventory after adding products:");

        ims.print\_Inventory();

        ims.update\_Product("101", "Updated CLRS Book", 20, 1500.0);

        System.out.println("\nInventory after updating product 101:");

        ims.print\_Inventory();

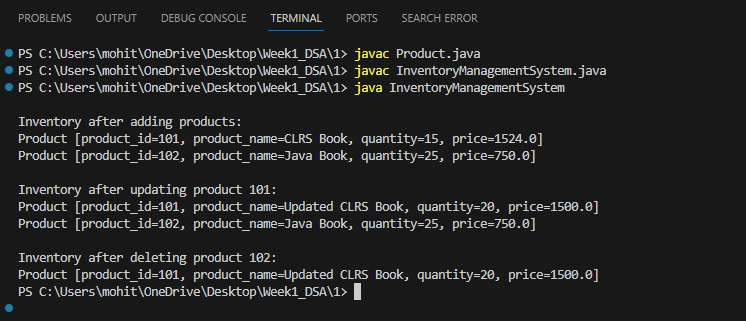
        ims.delete\_Product("102");

        System.out.println("\nInventory after deleting product 102:");

        ims.print\_Inventory();

    }

}

Output  


1. Understand Search Algorithms

**Explain linear search and binary search algorithms?**

*=>Linear Search*

Description: Linear search is a straightforward algorithm that checks each element in a list sequentially until the desired element is found or the list ends.

*Time Complexity*:

Best Case: O(1) (if the element is the first one)

Average Case: O(n)

Worst Case: O(n)

*Binary Search*

Description: Binary search is a more efficient algorithm that works on sorted lists. It repeatedly divides the search interval in half, comparing the target value to the middle element of the interval and adjusting the search range based on the comparison.

*Time Complexity:*

Best Case: O(1) (if the element is the middle one)

Average Case: O(log n)

Worst Case: O(log n)

**.Compare the time complexity of linear and binary search ?**

*=>Linear Search:*

Best Case: O(1) (if the element is the first one)

Average Case: O(n)

Worst Case: O(n)

*Binary Search:*

Best Case: O(1) (if the element is the middle one)

Average Case: O(log n)

Worst Case: O(log n)

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

*=>Linear Search:*

1.Suitable for small or unsorted datasets.

2.Simple to implement and does not require the data to be sorted.

3.Useful when the cost of sorting the data outweighs the benefits of faster searches.

*Binary Search:*

1.Suitable for large, sorted datasets.

2.Requires initial sorting of the data, which can be time-consuming but allows for much faster search times afterwards.

3.Ideal when multiple searches are needed on a static dataset, as the overhead of sorting is amortized over many search operations.  
  
  
**Exercise 2: E-commerce Platform Search Function  
  
  
//BinarySearch**

package EcommercePlatform;

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

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

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

        int left = 0;

        int right = products.length - 1;

        while (left <= right) {

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

            int cmp = products[mid].getProductId().compareTo(targetProductId);

            if (cmp == 0) {

                return products[mid];

            } else if (cmp < 0) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return null;

    }

}

**//LinearSearch**

package EcommercePlatform;

public class LinearSearch {

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

        for (Product product : products) {

            if (product.getProductId().equals(targetProductId)) {

                return product;

            }

        }

        return null;

    }

}

**//Product.java**

package EcommercePlatform;

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 "Product [productId=" + productId + ", productName=" + productName + ", category=" + category + "]";

    }

}

**//EcommerceSearch**

package EcommercePlatform;

public class ECommerceSearch {

    public static void main(String[] args) {

        Product[] products = {

            new Product("001", "iPhone 13", "Electronics"),

            new Product("002", "MacBook Pro", "Electronics"),

            new Product("003", "Samsung Galaxy S21", "Electronics"),

            new Product("004", "Sony WH-1000XM4", "Accessories"),

            new Product("005", "Kindle Paperwhite", "Books"),

            new Product("006", "Nike Air Max", "Fashion"),

            new Product("007", "Adidas Ultraboost", "Fashion"),

            new Product("008", "Apple Watch Series 6", "Accessories"),

            new Product("009", "Dell XPS 13", "Electronics"),

            new Product("010", "Amazon Echo", "Smart Home"),

            new Product("011", "Google Nest Hub", "Smart Home"),

            new Product("012", "Harry Potter Box Set", "Books"),

            new Product("013", "Logitech MX Master 3", "Accessories"),

            new Product("014", "Bose QuietComfort 35 II", "Accessories"),

            new Product("015", "The Lean Startup", "Books")

        };

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

        Product result1 = LinearSearch.linearSearch(products, "015");

        System.out.println(result1 != null ? "Found: " + result1 : "Not Found");

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

        Product result2 = BinarySearch.binarySearch(products, "015");

        System.out.println(result2 != null ? "Found: " + result2 : "Not Found");

    }

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

=>Big O Notation: Big O notation describes the upper bound of the time complexity of an algorithm, providing an estimate of the worst-case scenario in terms of input size (n). It helps in understanding how an algorithm's performance scales with increasing input sizes.

Why It's Important: Analyzing algorithms with Big O notation helps developers choose the most efficient algorithm, ensuring optimal performance, especially for large datasets.

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

=>Best, Average, and Worst-Case Scenarios for Search Operations:

1.Best Case: The minimum time an algorithm takes to complete, usually when the desired element is found at the first position. For example, O(1) for linear search if the element is the first item.

2.Average Case: The expected time an algorithm takes to complete over all possible inputs. For example, O(n/2) for linear search, as it may need to check half of the elements on average.

3.Worst Case: The maximum time an algorithm takes to complete, usually when the desired element is not found, or is the last element. For example, O(n) for linear search when the element is not in the array.

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

=>Linear Search: O(n)

Best Case: O(1) - When the element is the first item.

Average Case: O(n/2) ≈ O(n) - When the element is somewhere in the middle.

Worst Case: O(n) - When the element is the last item or not present.

Binary Search: O(log n)

Best Case: O(1) - When the element is the middle item.

Average Case: O(log n) - Each step reduces the search interval by half.

Worst Case: O(log n) - When the element is not present, or requires maximum steps to be found.

**Discuss which algorithm is more suitable for your platform and why?**

=> Binary Search is more suitable for the e-commerce platform due to its logarithmic time complexity, making it significantly faster for large datasets. However, it requires the array to be sorted. If the product data changes frequently, maintaining a sorted array might add overhead, but the performance gain in search operations justifies it for platforms with a large number of products.

Linear Search is simpler and does not require a sorted array, but its linear time complexity makes it less efficient for large datasets.

-----------------------------------------------------------------------------------------------------------------------------

**Exercise 3: Sorting Customer Orders**

**//BubbleSort**

package SortingCustomerOrder;

public class BubbleSort {

    public static void bubbleSort(Order[] orders) {

        int n = orders.length;

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

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

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

                    // Swapping

                    Order temp = orders[j];

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

                    orders[j + 1] = temp;

                }

            }

        }

    }

}

**//QuickSort**

package SortingCustomerOrder;

public class QuickSort {

    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].getTotalPrice();

        int i = (low - 1);

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

            if (orders[j].getTotalPrice() < pivot) {

                i++;

                //swapping

                Order temp = orders[i];

                orders[i] = orders[j];

                orders[j] = temp;

            }

        }

        //swapping

        Order temp = orders[i + 1];

        orders[i + 1] = orders[high];

        orders[high] = temp;

        return i + 1;

    }

}

//Order.java

package SortingCustomerOrder;

public class Order {

    private String orderId;

    private String customerName;

    private double totalPrice;

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

        this.orderId = orderId;

        this.customerName = customerName;

        this.totalPrice = totalPrice;

    }

    // Getters

    public String getOrderId() {

        return orderId;

    }

    public String getCustomerName() {

        return customerName;

    }

    public double getTotalPrice() {

        return totalPrice;

    }

    @Override

    public String toString() {

        return "Order [orderId=" + orderId + ", customerName=" + customerName + ", totalPrice=" + totalPrice + "]";

    }

}

//Main.java

package SortingCustomerOrder;

import java.util.Arrays;

public class Main {

    public static void main(String[] args) {

        Order[] orders = {

                new Order("O001", "Alice", 250.50),

                new Order("O002", "Bob", 150.75),

                new Order("O003", "Charlie", 300.00),

                new Order("O004", "David", 100.25),

                new Order("O005", "Eve", 200.10)

        };

        // Bubble Sort

        Order[] bubbleSortedOrders = Arrays.copyOf(orders, orders.length);

        BubbleSort.bubbleSort(bubbleSortedOrders);

        System.out.println("Bubble Sorted Orders:");

        for (Order order : bubbleSortedOrders) {

            System.out.println(order);

        }

        // Quick Sort

        Order[] quickSortedOrders = Arrays.copyOf(orders, orders.length);

        QuickSort.quickSort(quickSortedOrders, 0, quickSortedOrders.length - 1);

        System.out.println("Quick Sorted Orders:");

        for (Order order : quickSortedOrders) {

            System.out.println(order);

        }

    }

}

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

**=>Bubble Sort**

Description: Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

Time Complexity:

Best Case: O(n)

Average Case: O(n^2)

Worst Case: O(n^2)

**Quick Sort**

Description: Quick Sort is a divide-and-conquer algorithm that picks a pivot element, partitions the array into elements less than and greater than the pivot, and recursively sorts the sub-arrays.

Time Complexity:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n^2)

**Compare the performance (time complexity) of Bubble Sort and Quick Sort?**

**=>Bubble Sort:**

Best Case: O(n)

Average Case: O(n^2)

Worst Case: O(n^2)

Quick Sort:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n^2) (but rare with good pivot selection)

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

=>1.Efficiency: Quick Sort's average and best-case time complexity of O(n log n) makes it much faster for large datasets compared to Bubble Sort’s O(n^2).

2.Scalability: Quick Sort can handle large arrays more efficiently and is well-suited for in-place sorting.

3.Practical Performance: In practice, Quick Sort is faster than Bubble Sort for most inputs due to lower constants and better cache performance.

[------------------------------------------------------------------------------------------------------------------------------------]  
  
**Exercise 4: Employee Management System**

//Employee.java

package EmployeeManagementSystem;

public class Employee {

    private String employeeId;

    private String name;

    private String position;

    private double salary;

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

        this.employeeId = employeeId;

        this.name = name;

        this.position = position;

        this.salary = salary;

    }

    // Getters and Setters

    public String getEmployeeId() {

        return employeeId;

    }

    public void setEmployeeId(String employeeId) {

        this.employeeId = employeeId;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public String getPosition() {

        return position;

    }

    public void setPosition(String position) {

        this.position = position;

    }

    public double getSalary() {

        return salary;

    }

    public void setSalary(double salary) {

        this.salary = salary;

    }

    @Override

    public String toString() {

        return "Employee [employeeId=" + employeeId + ", name=" + name + ", position=" + position + ", salary=" + salary

                + "]";

    }

}

//EmployeeManagementSystem.java

package EmployeeManagementSystem;

public class EmployeeManagementSystem {

    private Employee[] employees;

    private int size;

    public EmployeeManagementSystem(int capacity) {

        employees = new Employee[capacity];

        size = 0;

    }

    public void addEmployee(Employee employee) {

        if (size >= employees.length) {

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

            return;

        }

        employees[size++] = employee;

    }

    public Employee searchEmployee(String employeeId) {

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

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

                return employees[i];

            }

        }

        return null;

    }

    public void traverseEmployees() {

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

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

        }

    }

    public void deleteEmployee(String employeeId) {

        int index = -1;

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

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

                index = i;

                break;

            }

        }

        if (index != -1) {

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

                employees[i] = employees[i + 1];

            }

            employees[size - 1] = null;

            size--;

        } else {

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

        }

    }

}

//Main.java

package EmployeeManagementSystem;

public class Main {

    public static void main(String[] args) {

        EmployeeManagementSystem system = new EmployeeManagementSystem(5);

        // Adding employees

        system.addEmployee(new Employee("E001", "Alice", "Manager", 75000));

        system.addEmployee(new Employee("E002", "Bob", "Developer", 60000));

        system.addEmployee(new Employee("E003", "Charlie", "Designer", 50000));

        system.addEmployee(new Employee("E004", "David", "Tester", 55000));

        system.addEmployee(new Employee("E005", "Eve", "HR", 45000));

        // Traversing employees

        System.out.println("All Employees:");

        system.traverseEmployees();

        // Searching for an employee

        Employee searchResult = system.searchEmployee("E003");

        System.out.println("\nSearch Result:");

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

        // Deleting an employee

        system.deleteEmployee("E004");

        System.out.println("\nEmployees after deletion:");

        system.traverseEmployees();

    }

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Explain how Arrays Are Represented in Memory and Their Advantages?**

=>Memory Representation: Arrays are stored in contiguous memory locations, which means that each element of the array is stored right next to the other. This allows for efficient indexing using simple arithmetic.

For example, if an array starts at memory location base\_address, the element at index i is located at base\_address + i \* element\_size.

Advantages:

1.Constant-Time Access: Accessing any element by index is O(1), making arrays very efficient for retrieval.

2.Cache Friendliness: Due to their contiguous memory allocation, arrays exhibit good cache locality, leading to faster access times.

3.Predictable Memory Usage: The memory usage of arrays is fixed at creation time,

**Analyze the time complexity of each operation (add, search, traverse, delete)?**

=>Time Complexity of Each Operation are:

1.Add Employee: O(1) - Adding an employee is a constant-time operation as long as there is space in the array.

2.Search Employee: O(n) - Linear search through the array is required, making the time complexity proportional to the number of employees.

3.Traverse Employees: O(n) - Traversing the array to print each employee takes linear time.

4.Delete Employee: O(n) - Deleting an employee requires shifting elements in the array, resulting in linear time complexity.

**Discuss the limitations of arrays and when to use them?**

=>Limitations of Arrays and When to Use Them are:

1.Fixed Size: Arrays have a fixed size, which means you must allocate enough space at creation time. This can lead to wasted memory if the array is not fully utilized or the need for resizing operations if the array becomes full.

2.Inefficient Deletion: Deleting an element from an array requires shifting elements, which is inefficient for large arrays.

3.Poor Insertion Performance: Inserting an element at any position other than the end requires shifting elements, which can be inefficient.  
------------------------------------------------------------------------------------------------------------------------------  
  
**Exercise 5: Task Management System**

//SinglyLinkedList.java

package TaskManagementSystem;

public class SinglyLinkedList {

    private class Node {

        Task task;

        Node next;

        Node(Task task) {

            this.task = task;

            this.next = null;

        }

    }

    private Node head;

    public SinglyLinkedList() {

        head = null;

    }

    // Add

    public void addTask(Task task) {

        Node newNode = new Node(task);

        if (head == null) {

            head = newNode;

        } else {

            Node current = head;

            while (current.next != null) {

                current = current.next;

            }

            current.next = newNode;

        }

    }

    // Search

    public Task searchTask(String taskId) {

        Node current = head;

        while (current != null) {

            if (current.task.getTaskId().equals(taskId)) {

                return current.task;

            }

            current = current.next;

        }

        return null;

    }

    // Traverse

    public void traverseTasks() {

        Node current = head;

        while (current != null) {

            System.out.println(current.task);

            current = current.next;

        }

    }

    // Delete

    public void deleteTask(String taskId) {

        if (head == null)

            return;

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

            head = head.next;

            return;

        }

        Node current = head;

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

            current = current.next;

        }

        if (current.next != null) {

            current.next = current.next.next;

        }

    }

}

//Task.java

package TaskManagementSystem;

public class Task {

    private String taskId;

    private String taskName;

    private String status;

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

        this.taskId = taskId;

        this.taskName = taskName;

        this.status = status;

    }

    // Getters and Setters

    public String getTaskId() {

        return taskId;

    }

    public void setTaskId(String taskId) {

        this.taskId = taskId;

    }

    public String getTaskName() {

        return taskName;

    }

    public void setTaskName(String taskName) {

        this.taskName = taskName;

    }

    public String getStatus() {

        return status;

    }

    public void setStatus(String status) {

        this.status = status;

    }

    @Override

    public String toString() {

        return "Task [taskId=" + taskId + ", taskName=" + taskName + ", status=" + status + "]";

    }

}

//Main.java

package TaskManagementSystem;

public class Main {

    public static void main(String[] args) {

        SinglyLinkedList taskList = new SinglyLinkedList();

      //Add

        taskList.addTask(new Task("T001", "Task 1", "Pending"));

        taskList.addTask(new Task("T002", "Task 2", "Completed"));

        taskList.addTask(new Task("T003", "Task 3", "In Progress"));

        // Traverse

        System.out.println("All Tasks:");

        taskList.traverseTasks();

        // Search

        Task searchResult = taskList.searchTask("T002");

        System.out.println("\nSearch Result:");

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

        // Delete

        taskList.deleteTask("T002");

        System.out.println("\nTasks after deletion:");

        taskList.traverseTasks();

    }

}

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

---Understanding Linked Lists

A linked list is a linear data structure where elements are not stored at contiguous memory locations.

Instead, each element (node) contains data and a reference (link) to the next node in the sequence.

Types of Linked Lists:

Singly Linked List: Each node points to the next node.

Doubly Linked List: Each node points to both the next and previous nodes.

**Analyze the time complexity of each operation.**

---Time Complexity Analysis

Add: O(n) in the worst case (adding at the end), O(1) if adding at the beginning

Search: O(n)

Traverse: O(n)

Delete: O(n) in the worst case (deleting the last element)

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

--Advantages of Linked Lists over Arrays

Dynamic size: Linked lists can grow or shrink as needed, unlike arrays which have a fixed size.

Efficient insertions and deletions: Inserting or deleting elements in a linked list is generally faster than in an array, especially in the middle of the list.

Flexibility: Linked lists can be used to implement various data structures like stacks, queues, and graphs.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Exercise 6: Library Management System**

**//Book.java**

package LibraryManagementSystem;

public class Book {

    private String bookId;

    private String title;

    private String author;

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

        this.bookId = bookId;

        this.title = title;

        this.author = author;

    }

    // Getters and Setters

    public String getBookId() {

        return bookId;

    }

    public void setBookId(String bookId) {

        this.bookId = bookId;

    }

    public String getTitle() {

        return title;

    }

    public void setTitle(String title) {

        this.title = title;

    }

    public String getAuthor() {

        return author;

    }

    public void setAuthor(String author) {

        this.author = author;

    }

    @Override

    public String toString() {

        return "Book [bookId=" + bookId + ", title=" + title + ", author=" + author + "]";

    }

}

**//Library.java**

package LibraryManagementSystem;

import java.util.Arrays;

import java.util.Comparator;

public class Library {

private Book[] books;

private int size;

public Library(int capacity) {

books = new Book[capacity];

size = 0;

}

public void addBook(Book book) {

if (size >= books.length) {

System.out.println("Library is full");

return;

}

books[size++] = book;

}

public Book linearSearchByTitle(String title) {

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

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

return books[i];

}

}

return null;

}

public Book binarySearchByTitle(String title) {

Arrays.sort(books, 0, size, Comparator.comparing(Book::getTitle));

int left = 0;

int right = size - 1;

while (left <= right) {

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

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

if (compareResult == 0) {

return books[mid];

} else if (compareResult < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

**//Main.java**

package LibraryManagementSystem;

public class Main {

    public static void main(String[] args) {

        Library library = new Library(5);

        // Adding books

        library.addBook(new Book("B001", "The Great Gatsby", "F. Scott Fitzgerald"));

        library.addBook(new Book("B002", "Moby Dick", "Herman Melville"));

        library.addBook(new Book("B003", "1984", "George Orwell"));

        library.addBook(new Book("B004", "To Kill a Mockingbird", "Harper Lee"));

        library.addBook(new Book("B005", "The Catcher in the Rye", "J.D. Salinger"));

        // Linear Search

        System.out.println("Linear Search for '1984':");

        Book linearSearchResult = library.linearSearchByTitle("1984");

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

        // Binary Search

        System.out.println("\nBinary Search for '1984':");

        Book binarySearchResult = library.binarySearchByTitle("1984");

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

    }

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Explain linear search and binary search algorithms?**

=>Linear Search

Description: Linear search is a straightforward algorithm that checks each element in a list sequentially until the desired element is found or the list ends.

Time Complexity:

Best Case: O(1) (if the element is the first one)

Average Case: O(n)

Worst Case: O(n)

Binary Search

Description: Binary search is a more efficient algorithm that works on sorted lists. It repeatedly divides the search interval in half, comparing the target value to the middle element of the interval and adjusting the search range based on the comparison.

Time Complexity:

Best Case: O(1) (if the element is the middle one)

Average Case: O(log n)

Worst Case: O(log n)

**Compare the time complexity of linear and binary search ?**

=>Linear Search:

Best Case: O(1) (if the element is the first one)

Average Case: O(n)

Worst Case: O(n)

Binary Search:

Best Case: O(1) (if the element is the middle one)

Average Case: O(log n)

Worst Case: O(log n)

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

=>Linear Search:

1.Suitable for small or unsorted datasets.

2.Simple to implement and does not require the data to be sorted.

3.Useful when the cost of sorting the data outweighs the benefits of faster searches.

Binary Search:

1.Suitable for large, sorted datasets.

2.Requires initial sorting of the data, which can be time-consuming but allows for much faster search times afterwards.

3.Ideal when multiple searches are needed on a static dataset, as the overhead of sorting is amortized over many search operations.

**Exercise 7: Financial Forecasting**

**//** **FinancialForecasting.java**

package FinancialForecasting;

import java.util.HashMap;

import java.util.Map;

public class FinancialForecasting {

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

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

        if (years == 0) {

            return initialValue;

        }

        if (memo.containsKey(years)) {

            return memo.get(years);

        }

        double result = calculateFutureValue(initialValue, growthRate, years - 1) \* (1 + growthRate);

        memo.put(years, result);

        return result;

    }

    public static void main(String[] args) {

        double initialValue = 1500;

        double growthRate = 0.06;

        int years = 9;

        double futureValue = calculateFutureValue(initialValue, growthRate, years);

        System.out.println("The predicted future value after " + years + " years is: " + futureValue);

    }

}

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

=>Recursion: Recursion is a programming technique where a function calls itself directly or indirectly to solve a problem. Each recursive call works on a smaller instance of the same problem, and the solution to the overall problem is built up from the solutions to these smaller instances.

Base Case and Recursive Case: A recursive algorithm must have at least one base case that does not make a recursive call, to prevent infinite recursion. The recursive case reduces the problem size and makes the recursive call.

Advantages of Recursion:

1.Simplicity: Recursive solutions can be more intuitive and easier to implement for problems that have a natural recursive structure, such as tree traversal, factorial computation, and Fibonacci sequence.

2.Divide-and-Conquer: Many algorithms (e.g., quicksort, mergesort) benefit from the divide-and-conquer strategy, which is naturally implemented using recursion.

Q.Discuss the time complexity of your recursive algorithm?

=>The time complexity of the recursive algorithm is O(n), where n is the number of periods. This is because the function makes a recursive call for each period, resulting in n recursive calls in total.

**Explain how to optimize the recursive solution to avoid excessive computation?**

=>Optimizing the Recursive Solution:

1.Memoization: To avoid redundant computations in recursive algorithms, memoization can be used. Memoization involves storing the results of expensive function calls and reusing the stored results when the same inputs occur again.

2.Iterative Approach: Converting the recursive solution to an iterative one can improve performance and avoid the overhead of recursive calls.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_