**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.

**Theory:-**

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

- In real-world systems (e.g., Amazon warehouses), millions of products need to be handled efficiently. If you choose the wrong data structure:

* Search becomes slow
* Updates lag
* Deletion takes longer
* Overall performance drops

**Data structures help us:**

* Organize data meaningfully
* Perform operations in **minimum time**
* Scale to large input sizes

Discuss the types of data structures suitable for this problem.

- Let’s compare:

| **Structure** | **Search** | **Insert** | **Delete** | **Suitable When** |
| --- | --- | --- | --- | --- |
| **ArrayList** | O(n) | O(1) | O(n) | When order matters |
| **HashMap** | O(1)\* | O(1)\* | O(1)\* | When fast lookup needed |

We chose **HashMap** for this exercise because:

* It maps a **key (productId)** to a **value (Product object)**
* Supports **O(1)** average time complexity for add, update, delete, and get operations
* Ensures each productId is **unique**, avoiding duplicates

**Source Code:-**

***Product.java:***

public class Product {

    private int productId;

    private String productName;

    private int quantity;

    private double price;

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

        this.productId = productId;

        this.productName = productName;

        this.quantity = quantity;

        this.price = price;

    }

    // Getters

    public int getProductId() { return productId; }

    public String getProductName() { return productName; }

    public int getQuantity() { return quantity; }

    public double getPrice() { return price; }

    // Setters

    public void setProductName(String name) { this.productName = name; }

    public void setQuantity(int quantity) { this.quantity = quantity; }

    public void setPrice(double price) { this.price = price; }

    @Override

    public String toString() {

        return "Product[ID=" + productId + ", Name=" + productName +

               ", Quantity=" + quantity + ", Price=" + price + "]";

    }

}

***InventoryManagement.java(Logic):***

import java.util.HashMap;

public class InventoryManager {

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

    // Add product

    public void addProduct(Product product) {

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

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

    }

    // Update product

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

        Product product = inventory.get(productId);

        if (product != null) {

            product.setProductName(name);

            product.setQuantity(quantity);

            product.setPrice(price);

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

        } else {

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

        }

    }

    // Delete product

    public void deleteProduct(int productId) {

        Product removed = inventory.remove(productId);

        if (removed != null) {

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

        } else {

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

        }

    }

    // Display all products

    public void displayInventory() {

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

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

            System.out.println(product);

        }

    }

}

***TestingClass.java:***

public class TestingClass {

    public static void main(String[] args) {

        InventoryManager manager = new InventoryManager();

        // Create some products

        Product p1 = new Product(101, "HP Laptop", 10, 75000.0);

        Product p2 = new Product(102, "Lenovo Mouse", 50, 500.0);

        Product p3 = new Product(103, "Acer Keyboard", 30, 1500.0);

        // Add products

        manager.addProduct(p1);

        manager.addProduct(p2);

        manager.addProduct(p3);

        // Display current inventory

        manager.displayInventory();

        // Update a product

        manager.updateProduct(102, "Wireless Mouse", 40, 650.0);

        // Delete a product

        manager.deleteProduct(103);

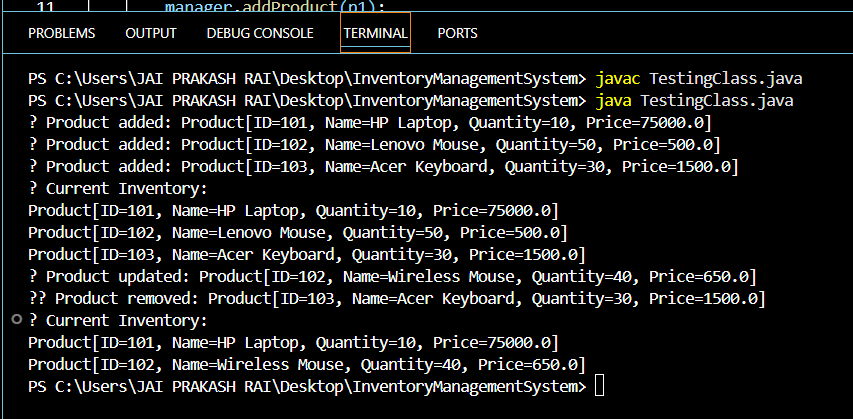
        // Final inventory state

        manager.displayInventory();

    }

}

**Output:-**



**Analysis:-**

We're using HashMap<Integer, Product> for storing products.

| **Operation** | **Time Complexity** | **Why it's efficient?** |
| --- | --- | --- |
| Add Product | O(1) average | Insert via unique key (productId) |
| Update | O(1) average | Direct access by key, then modify |
| Delete | O(1) average | Direct remove by key |
| Display All | O(n) | Need to iterate over all products |

So, this is **highly optimized for large inventories**.

**Optimization Ideas**

You can make this system even better by:

* Adding a **search by name** using filters or TreeMap.
* Adding **file storage** to save/load inventory between sessions.
* Introducing a **category system** using nested HashMaps or a ProductCategory class.
* Adding validation (e.g., no duplicate IDs).

**Exercise 7: Financial Forecasting(Mandatory)**

**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.

**Theory:-**

**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 instances of a problem. Instead of using loops, recursion breaks a complex problem into simpler sub-problems until it reaches a base case. This approach is especially useful when the problem naturally follows a repetitive or hierarchical structure, such as tree traversals, factorial calculations, or in this case, financial forecasting where each year’s prediction is based on the previous year.

**Source Code:-**

*FinancialForecast.java:*

public class FinancialForecast {

    // Recursive method to calculate future value

    public static double forecastValueRecursive(double initialAmount, double growthRate, int years) {

        if (years == 0) {

            return initialAmount; // Base case

        } else {

            // Recursive case

            return forecastValueRecursive(initialAmount, growthRate, years - 1) \* (1 + growthRate);

        }

    }

    // Optimized version using iteration to avoid deep recursion

    public static double forecastValueIterative(double initialAmount, double growthRate, int years) {

        double amount = initialAmount;

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

            amount \*= (1 + growthRate);

        }

        return amount;

    }

    public static void main(String[] args) {

        double initialAmount = 10000;      // Starting capital

        double growthRate = 0.08;          // 8% annual growth

        int years = 5;                     // Forecast period

        double resultRecursive = forecastValueRecursive(initialAmount, growthRate, years);

        System.out.println("Future Value (Recursive): " + resultRecursive);

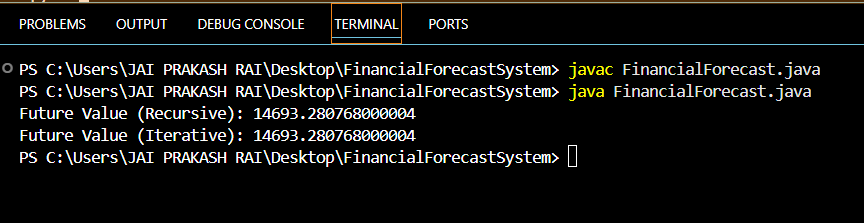
        double resultIterative = forecastValueIterative(initialAmount, growthRate, years);

        System.out.println("Future Value (Iterative): " + resultIterative);

    }

}

**Output:-**



**Analysis:-**

**Time Complexity:**

* **Recursive Version**:  
  Time Complexity = O(n)  
  Space Complexity = O(n) due to recursive call stack
* **Iterative Version**:  
  Time Complexity = O(n)  
  Space Complexity = O(1)

**When to Optimize Recursion**

* **Recursive solutions can lead to stack overflow** for large values of n.
* Repeated computations (as in Fibonacci) may need **memoization** or an **iterative alternative**.
* In this example, recursion is fine for small periods (years), but iteration is safer for long-term forecasting.

**Exercise 2: E-commerce Platform Search Function(Mandatory)**

**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.

**Theory:-**

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

- Big O notation expresses the **time or space complexity** of an algorithm in terms of input size n. It helps estimate:

* How well your algorithm scales
* Whether it will perform well with large data (like thousands of products)

**Why it matters in Search?**

Search is a **core operation** in any e-commerce platform:

* Users search by **name**, **ID**, or **category**
* You must retrieve results **quickly**

**Time complexity** of your search algorithm decides:

* How many products can be searched **in acceptable time**
* How it performs under **traffic spikes**

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

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

-

**Source Code:-**

***Product.java:***

public class Product {

private int productId;

private String productName;

private String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() { return productId; }

public String getProductName() { return productName; }

public String getCategory() { return category; }

@Override

public String toString() {

return "Product[ID=" + productId + ", Name=" + productName + ", Category=" + category + "]";

}

}

***SearchService.java:***

import java.util.Arrays;

import java.util.Comparator;

public class SearchService {

// Linear Search (by Product Name, case-insensitive)

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

for (Product product : products) {

// Compare ignoring case

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

return product; // found

}

}

return null; // not found

}

// Binary Search (by Product Name, case-insensitive)

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

// Sort first by product name (case-insensitive)

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

String midName = products[mid].getProductName().toLowerCase();

String target = targetName.toLowerCase();

if (midName.equals(target)) {

return products[mid]; // match found

} else if (midName.compareTo(target) < 0) {

left = mid + 1; // search right half

} else {

right = mid - 1; // search left half

}

}

return null; // not found

}

}

***TestingClass.java:***

public class TestingClass {

public static void main(String[] args) {

// Sample Product Array

Product[] products = {

new Product(101, "Laptop", "Electronics"),

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

new Product(103, "Shoes", "Fashion"),

new Product(104, "Book", "Education"),

new Product(105, "Smartphone", "Electronics")

};

String searchTarget = "Shoes";

// --- Linear Search -----

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

Product foundLinear = SearchService.linearSearch(products, searchTarget);

if (foundLinear != null) {

System.out.println("Found using Linear Search: " + foundLinear);

} else {

System.out.println(" Not found using Linear Search.");

}

// --- Binary Search -----

System.out.println("\nPerforming Binary Search...");

Product foundBinary = SearchService.binarySearch(products, searchTarget);

if (foundBinary != null) {

System.out.println("Found using Binary Search: " + foundBinary);

} else {

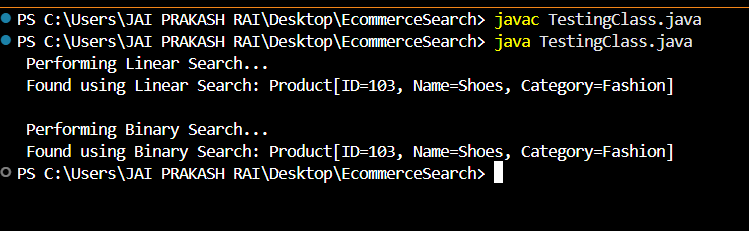
System.out.println(" Not found using Binary Search.");

}

}

}

**Output:-**



**Analysis:-**

| **Search Type** | **Time Complexity** | **Conditions** | **Performance** |
| --- | --- | --- | --- |
| **Linear Search** | O(n) | Unsorted array | Slower for large n |
| **Binary Search** | O(log n) + O(n log n) for sort | Needs sorted data | Faster after sorting |

So binary search is more suitable because it is **faster** once data is sorted that is ideal for large product catalogs.

**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.

**Theory:-**

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

-Sorting algorithms help organize data to:

* **Make searching faster**
* **Help visualize priority** (e.g., high-value orders)
* **Improve user experience** (sorted results)

Let’s look at the most common ones:

**Bubble Sort**

* Repeatedly **swaps adjacent elements** if they're in the wrong order.
* Very simple but very **slow** for large datasets.

**Time Complexities:**

* Best Case: O(n) (already sorted)
* Average Case: O(n²)
* Worst Case: O(n²)

**Insertion Sort**

* Builds the sorted array one item at a time.
* Fast for **small** or **nearly sorted** data.

**Time Complexities:**

* Best: O(n)
* Average/Worst: O(n²)

**Quick Sort**

* Uses **divide and conquer**:
  + Picks a **pivot**
  + Partitions array around pivot
  + Recursively sorts left and right parts

**Time Complexities:**

* Best/Average: O(n log n)
* Worst: O(n²) (rare, happens if bad pivot always picked)

-**Preferred in real-world** due to speed & low memory use.

**Merge Sort**

* Also divide and conquer, but always divides evenly
* Needs **extra memory** for merging

**Time Complexities:**

* Always: O(n log n) (best, average, worst)
* Space: O(n)

-Great for **stable sorting** and large datasets.

**Source Code:-**

***Order.java:***

public class Order {

private int orderId;

private String customerName;

private double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public int getOrderId() { return orderId; }

public String getCustomerName() { return customerName; }

public double getTotalPrice() { return totalPrice; }

@Override

public String toString() {

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

}

}

***SortService.java:***

public class SortService {

// ---------- Bubble Sort ----------

public static void bubbleSort(Order[] orders) {

int n = orders.length;

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

boolean swapped = false;

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

// Compare totalPrice

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

// Swap

Order temp = orders[j];

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

orders[j + 1] = temp;

swapped = true;

}

}

if (!swapped) break; // Optimization: stop if already sorted

}

}

// ---------- Quick Sort ----------

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

if (low < high) {

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

// Recursively sort left and right subarrays

quickSort(orders, low, pivotIndex - 1);

quickSort(orders, pivotIndex + 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++;

// Swap

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

// Swap pivot into correct place

Order temp = orders[i + 1];

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

orders[high] = temp;

return i + 1;

}

// Utility function to print orders

public static void printOrders(Order[] orders) {

for (Order order : orders) {

System.out.println(order);

}

}

}

***TestingClass.java:***

public class Main {

public static void main(String[] args) {

// Sample data

Order[] orders = {

new Order(101, "Alice", 1500.0),

new Order(102, "Bob", 200.0),

new Order(103, "Charlie", 850.0),

new Order(104, "Diana", 4000.0),

new Order(105, "Eve", 1200.0)

};

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

SortService.printOrders(orders);

// ---- Bubble Sort ----

Order[] bubbleSorted = orders.clone(); // Copy original

SortService.bubbleSort(bubbleSorted);

System.out.println("\nOrders Sorted by Bubble Sort (Ascending Total Price):");

SortService.printOrders(bubbleSorted);

// ---- Quick Sort ----

Order[] quickSorted = orders.clone(); // Copy original

SortService.quickSort(quickSorted, 0, quickSorted.length - 1);

System.out.println("\nOrders Sorted by Quick Sort (Ascending Total Price):");

SortService.printOrders(quickSorted);

}

}

**Output:**



**Analysis:-**

**Bubble Sort:**

* **Time Complexity**:
  + Best: O(n) (if already sorted)
  + Worst: O(n²)
* **Drawbacks**:
  + Very slow for large arrays
  + Too many unnecessary comparisons/swaps

**Quick Sort:**

* **Time Complexity**:
  + Average: O(n log n)
  + Worst: O(n²) (if poor pivot)
* **Advantages**:
  + In-place (low memory)
  + Much faster in practice
  + Preferred in real-world systems

Thus, **Quick Sort** is preferred over Bubble Sort!!

**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.

**Theory:-**

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

- An array is a **contiguous block of memory** where elements are stored **sequentially**. Each element can be accessed directly using an **index**.

Arrays offer several key advantages that make them a fundamental data structure in programming, especially for applications like an employee management system:

* **Indexed Access**: Arrays allow constant-time access (O(1)) to any element using its index. This means retrieving an employee record is very fast and efficient.
* **Memory Efficiency**: Elements are stored in a **contiguous block of memory**, making arrays highly memory-efficient and **cache-friendly**, which improves performance on a low-level (hardware) scale.
* **Simplicity**: Arrays are simple to declare, initialize, and use. This makes them ideal for beginner-level systems or small-scale applications where the number of items is known beforehand.
* **Performance**: Due to their predictable structure and direct memory allocation, arrays offer faster access and manipulation compared to some complex data structures.
* **Ease of Debugging**: The fixed-size and structured nature of arrays make debugging and memory tracking easier, especially when managing a limited number of records.

**Source Code:-**

***Employee.java:***

public class Employee {

    private int employeeId;

    private String name;

    private String position;

    private double salary;

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

        this.employeeId = employeeId;

        this.name = name;

        this.position = position;

        this.salary = salary;

    }

    public int getEmployeeId() { return employeeId; }

    public String getName() { return name; }

    public String getPosition() { return position; }

    public double getSalary() { return salary; }

    @Override

    public String toString() {

        return "Employee[ID=" + employeeId + ", Name=" + name + ", Position=" + position + ", Salary=" + salary + "]";

    }

}

***EmployeeManager.java:***

public class EmployeeManager {

    private Employee[] employees;

    private int count;

    public EmployeeManager(int size) {

        employees = new Employee[size];

        count = 0;

    }

    // Add a new employee

    public void addEmployee(Employee emp) {

        if (count < employees.length) {

            employees[count++] = emp;

            System.out.println("Added: " + emp);

        } else {

            System.out.println("Cannot add employee. Array is full.");

        }

    }

    // Search employee by ID (Linear search)

    public Employee searchEmployee(int id) {

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

            if (employees[i].getEmployeeId() == id) {

                return employees[i];

            }

        }

        return null;

    }

    // Traverse and print 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 id) {

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

            if (employees[i].getEmployeeId() == id) {

                // Shift left from i

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

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

                }

                employees[--count] = null;

                System.out.println("Deleted employee with ID: " + id);

                return;

            }

        }

        System.out.println("Employee with ID " + id + " not found.");

    }

}

***TestingClass.java:***

public class TestingClass {

     public static void main(String[] args) {

        EmployeeManager manager = new EmployeeManager(5);

        // Add Employees

        manager.addEmployee(new Employee(101, "Alice", "Manager", 75000));

        manager.addEmployee(new Employee(102, "Bob", "Developer", 50000));

        manager.addEmployee(new Employee(103, "Charlie", "Designer", 40000));

        // Display All

        manager.displayAllEmployees();

        // Search Employee

        int searchId = 102;

        Employee found = manager.searchEmployee(searchId);

        if (found != null) {

            System.out.println(" Found: " + found);

        } else {

            System.out.println("Employee with ID " + searchId + " not found.");

        }

        // Delete Employee

        manager.deleteEmployee(102);

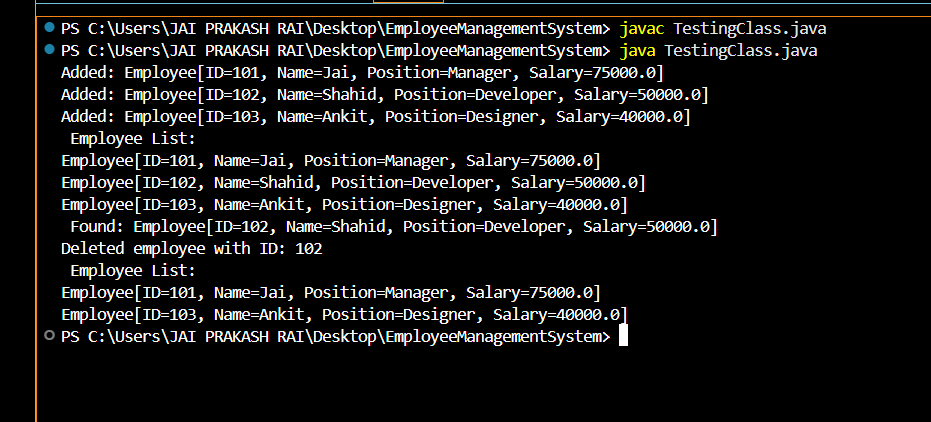
        // Display After Deletion

        manager.displayAllEmployees();

    }

}

**Output:-**



**Analysis:-**

| **Operation** | **Method Used** | **Time Complexity** |
| --- | --- | --- |
| Add | Insert at end | O(1) (if space) |
| Search | Linear Search | O(n) |
| Traverse | For-loop | O(n) |
| Delete | Shift elements left | O(n) |

While arrays are simple and efficient for certain use cases, they also come with several limitations that can impact scalability and flexibility in an employee management system:

* **Fixed Size**: Once an array is created, its size cannot be changed. This means if the number of employees exceeds the initial capacity, the array cannot accommodate new entries without creating a new, larger array and copying all existing data.
* **Costly Deletion and Insertion**: Removing or inserting an element at a specific position (other than at the end) requires shifting all subsequent elements, which leads to a time complexity of O(n)—inefficient for large datasets.
* **Wasted Memory or Overflow**: If you overestimate the size, unused array space leads to memory wastage. If you underestimate, the array may overflow, requiring manual resizing and copying.
* **Lack of Flexibility**: Arrays do not provide built-in methods for dynamic operations like resizing, filtering, or advanced searching. These features must be implemented manually, increasing development effort.
* **No Associative Access**: Arrays do not support key-based lookup like HashMap. So, searching by employee ID or name requires linear search, which can be slow as the data grows.

**When Should You Use Arrays?**

Use arrays when:

* Size is known and fixed
* Memory usage must be minimal
* Fast, indexed access is needed

**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.

**Theory:-**

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

- A **linked list** is a linear data structure where elements, called **nodes**, are stored in non-contiguous memory locations and are linked together using pointers. Each node contains two parts: the **data** (e.g., a task or value) and a **reference to the next node** in the sequence. Unlike arrays, linked lists do not require a fixed size and can grow or shrink dynamically at runtime, making them efficient for applications that involve frequent insertion and deletion of elements.

**Types of Linked Lists (Bulleted)**

* **Singly Linked List**
  + Each node stores data and a pointer to the **next node** only.
  + Allows **unidirectional traversal** (from head to tail).
  + Simpler and requires less memory.
* **Doubly Linked List**
  + Each node contains data, a pointer to the **next node**, and a pointer to the **previous node**.
  + Allows **bidirectional traversal** (forward and backward).
  + Slightly more complex but offers more flexibility.
* **Circular Linked List**
  + The last node points back to the **first node**, forming a circle.
  + Can be singly or doubly linked.
  + Useful for implementing cyclic processes like round-robin scheduling.

**Source Code:-**

***Task.java:***

public class Task {

    private int taskId;

    private String taskName;

    private String status;

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

        this.taskId = taskId;

        this.taskName = taskName;

        this.status = status;

    }

    public int getTaskId() { return taskId; }

    public String getTaskName() { return taskName; }

    public String getStatus() { return status; }

    @Override

    public String toString() {

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

    }

}

***TaskManager.java***

public class TaskManager {

    private TaskNode head;

    public TaskManager() {

        head = null;

    }

    // Add task at the end

    public void addTask(Task task) {

        TaskNode newNode = new TaskNode(task);

        if (head == null) {

            head = newNode;

        } else {

            TaskNode current = head;

            while (current.next != null) {

                current = current.next;

            }

            current.next = newNode;

        }

        System.out.println("Task added: " + task);

    }

    // Traverse all tasks

    public void displayTasks() {

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

        TaskNode current = head;

        while (current != null) {

            System.out.println(current.task);

            current = current.next;

        }

    }

    // Search task by ID

    public Task searchTask(int taskId) {

        TaskNode current = head;

        while (current != null) {

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

                return current.task;

            }

            current = current.next;

        }

        return null;

    }

    // Delete task by ID

    public void deleteTask(int taskId) {

        if (head == null) {

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

            return;

        }

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

            head = head.next;

            System.out.println("Deleted task with ID: " + taskId);

            return;

        }

        TaskNode current = head;

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

            current = current.next;

        }

        if (current.next == null) {

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

        } else {

            current.next = current.next.next;

            System.out.println("Deleted task with ID: " + taskId);

        }

    }

}

***TaskNode.java:***

public class TaskNode {

    Task task;

    TaskNode next;

    public TaskNode(Task task) {

        this.task = task;

        this.next = null;

    }

}

***TestingClass.java:***

public class TestingClass {

    public static void main(String[] args) {

        TaskManager manager = new TaskManager();

        // Add tasks

        manager.addTask(new Task(1, "Design UI", "Pending"));

        manager.addTask(new Task(2, "Develop Backend", "In Progress"));

        manager.addTask(new Task(3, "Write Tests", "Not Started"));

        // Display tasks

        manager.displayTasks();

        // Search task

        Task t = manager.searchTask(2);

        if (t != null) {

            System.out.println("Task Found: " + t);

        } else {

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

        }

        // Delete task

        manager.deleteTask(2);

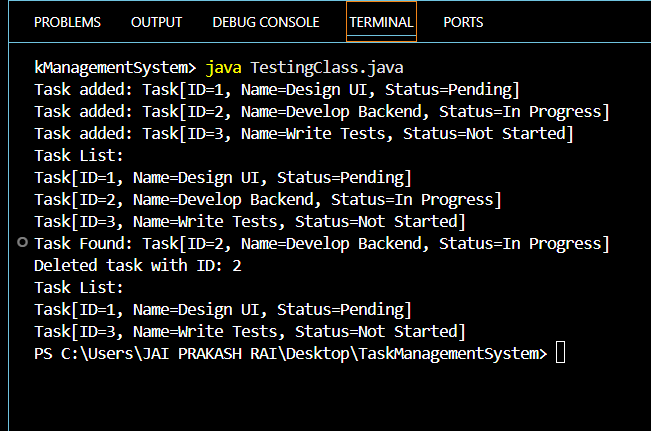
        // Display again

        manager.displayTasks();

    }

}

**Output:-**



**Analysis:-**

| **Operation** | **Time Complexity** |
| --- | --- |
| Add (at end) | O(n) |
| Traverse | O(n) |
| Search by ID | O(n) |
| Delete by ID | O(n) |

**Why O(n)?**

Because we must **iterate** through the list to reach the desired position or value.

**Advantages of Linked Lists over Arrays**

* **Dynamic Size**: Unlike arrays, linked lists do not require pre-defined size. You can add as many tasks as needed at runtime.
* **Efficient Insert/Delete**: Inserting or deleting elements (especially at the beginning or middle) is faster (O(1) once position is known), without shifting elements.
* **Memory Allocation**: Uses memory more flexibly — only allocates space when needed.
* **No Wastage**: No risk of over-allocating like in arrays where unused space is wasted.

**When to Prefer Linked Lists**

Use linked lists when:

* You need **frequent insertion/deletion**
* Data size is **unknown or dynamic**
* Sequential traversal is sufficient
* You can tolerate **linear time access**

**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.

**Theory:-**

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

- Search algorithms are used to find specific elements within a collection of data. Two fundamental search techniques are linear search and binary search. Linear search is a simple technique that sequentially checks each element in the list until a match is found or the list ends. It works on both sorted and unsorted data but can be inefficient for large datasets. On the other hand, binary search is a more efficient method but requires the data to be sorted. It repeatedly divides the search range in half, comparing the target with the middle element until the desired value is found or the search range is exhausted.

**Source Code :-**

***Book.java:***

public class Book {

    private int bookId;

    private String title;

    private String author;

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

        this.bookId = bookId;

        this.title = title;

        this.author = author;

    }

    public int getBookId() { return bookId; }

    public String getTitle() { return title; }

    public String getAuthor() { return author; }

    @Override

    public String toString() {

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

    }

}

***LibraryManager.java:***

import java.util.Arrays;

import java.util.Comparator;

public class LibraryManager {

    private Book[] books;

    public LibraryManager(Book[] books) {

        this.books = books;

    }

    // Linear Search by title

    public Book linearSearchByTitle(String title) {

        for (Book book : books) {

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

                return book;

            }

        }

        return null;

    }

    // Binary Search by title (assumes array is sorted)

    public Book binarySearchByTitle(String title) {

        int low = 0;

        int high = books.length - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

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

            if (comparison == 0) {

                return books[mid];

            } else if (comparison < 0) {

                low = mid + 1;

            } else {

                high = mid - 1;

            }

        }

        return null;

    }

    // Sort books by title

    public void sortBooksByTitle() {

        Arrays.sort(books, Comparator.comparing(Book::getTitle, String.CASE\_INSENSITIVE\_ORDER));

    }

    // Display all books

    public void displayBooks() {

        for (Book book : books) {

            System.out.println(book);

        }

    }

}

***TestingClass.java:***

public class TestingCLass {

    public static void main(String[] args) {

        Book[] books = {

            new Book(1, "Data Structures", "Mahasaya1"),

            new Book(2, "Operating Systems", "Mahasaya2"),

            new Book(3, "Java Programming", "Mahasaya3"),

            new Book(4, "Database Systems", "Mahasaya33"),

            new Book(5, "Computer Networks", "Mahasaya4")

        };

        LibraryManager library = new LibraryManager(books);

        // Linear Search (works without sorting)

        Book result1 = library.linearSearchByTitle("Java Programming");

        if (result1 != null) {

            System.out.println("Linear Search Result: " + result1);

        } else {

            System.out.println("Book not found using linear search.");

        }

        // Sort and use Binary Search

        library.sortBooksByTitle();

        Book result2 = library.binarySearchByTitle("Java Programming");

        if (result2 != null) {

            System.out.println("Binary Search Result: " + result2);

        } else {

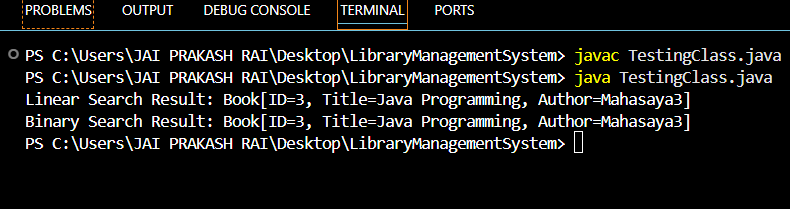
            System.out.println("Book not found using binary search.");

        }

    }

}

**Output:-**



**Analysis:-**

| **Operation** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| Best Case | O(1) | O(1) |
| Average Case | O(n) | O(log n) |
| Worst Case | O(n) | O(log n) |
| Requires Sorting | No | Yes |

**When to Use Each**

* Use **linear search** when:
  + The dataset is **small**
  + The list is **unsorted**
  + Simplicity is preferred
* Use **binary search** when:
  + The dataset is **large**
  + The list is **sorted**
  + You need faster search performance.