**Exercise 4: Employee Management System**

**1. Understand Array Representation**

**Array Representation in Memory:**

**Contiguous Block: Arrays are stored in contiguous memory locations, which means all elements are placed next to each other.**

**Index-Based Access: Arrays provide constant-time access to elements via indexing (i.e., accessing an element by its index), which is O(1).**

**Fixed Size: The size of an array is fixed upon creation, which means it cannot be resized dynamically.**

**Advantages:**

**Fast Access: Direct access to elements by index is very efficient.**

**Cache Friendliness: Due to contiguous memory allocation, arrays benefit from spatial locality, making them cache-friendly.**

**Simple Implementation: Arrays are straightforward to implement and use, which simplifies the code for basic operations.**

**2. Setup**

**// Employee.java**

**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 +**

**'}';**

**}**

**}**

**3. Implementation**

**// EmployeeManagementSystem.java**

**public class EmployeeManagementSystem {**

**private Employee[] employees;**

**private int size;**

**public EmployeeManagementSystem(int capacity) {**

**employees = new Employee[capacity];**

**size = 0;**

**}**

**// Add employee**

**public void addEmployee(Employee employee) {**

**if (size < employees.length) {**

**employees[size++] = employee;**

**} else {**

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

**}**

**}**

**// Search employee by ID**

**public Employee searchEmployeeById(String employeeId) {**

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

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

**return employees[i];**

**}**

**}**

**return null; // Employee not found**

**}**

**// Traverse all employees**

**public void traverseEmployees() {**

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

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

**}**

**}**

**// Delete employee by ID**

**public void deleteEmployeeById(String employeeId) {**

**int indexToRemove = -1;**

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

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

**indexToRemove = i;**

**break;**

**}**

**}**

**if (indexToRemove != -1) {**

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

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

**}**

**employees[size - 1] = null; // Avoid memory leak**

**size--;**

**} else {**

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

**}**

**}**

**}**

**4. Analysis**

**Time Complexity of Each Operation:**

**Add: O(1) - Adding an employee to the end of the array is constant time, assuming there is space.**

**Search: O(n) - Searching for an employee involves a linear scan of the array.**

**Traverse: O(n) - Traversing all employees requires a scan of the entire array.**

**Delete: O(n) - Deleting an employee involves finding the element (linear time) and shifting elements to fill the gap (linear time).**

**Limitations of Arrays:**

**Fixed Size: Arrays have a fixed size, which can lead to inefficiency if the number of employees grows or shrinks frequently.**

**Inefficient Insertions/Deletions: Inserting or deleting elements (except at the end) requires shifting elements, which can be inefficient.**

**Lack of Dynamic Resizing: Arrays cannot dynamically resize themselves, which means you might need to manage resizing manually or use more complex data structures for dynamic data.**

**When to Use Arrays:**

**Small to Medium Data: Arrays are suitable for smaller datasets or when you have a rough estimate of the size.**

**Fixed Size Data: When the size of the dataset is known and does not change frequently.**

**Simple Data Operations: For simple applications where advanced operations and dynamic resizing are not required.**