## Exercise 4: Employee Management System

### Step 1: Understand Array Representation

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

#### Array Representation in Memory

* **Memory Allocation**: Arrays are contiguous blocks of memory. Each element in an array is stored at successive memory locations, allowing for efficient indexing.
* **Indexing**: Elements in an array can be accessed directly using their index, making retrieval operations very fast (O(1) time complexity).
* **Fixed Size**: Arrays have a fixed size, determined at the time of their creation. This fixed size can be both an advantage and a limitation.

#### Advantages of Arrays

* **Fast Access**: Direct access to elements using index positions.
* **Efficient Traversal**: Since arrays are contiguous blocks of memory, iterating through elements is efficient.
* **Memory Efficiency**: No overhead associated with node pointers, as in linked lists.

### Step 2: Setup

#### Define the Employee Class

// Java implementation

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

public String getEmployeeId() {

return employeeId;

}

public String getName() {

return name;

}

public String getPosition() {

return position;

}

public double getSalary() {

return salary;

}

}

### Step 3: Implementation

#### Employee Management System using Array

// Java implementation

import java.util.Arrays;

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) {

// Resize the array if needed

employees = Arrays.copyOf(employees, employees.length \* 2);

}

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].getName() + " - " + employees[i].getPosition());

}

}

public boolean deleteEmployee(String employeeId) {

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

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

// Shift elements to the left

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

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

}

employees[size - 1] = null; // Clear the last element

size--;

return true;

}

}

return false;

}

}

### Step 4: Analysis

#### Time Complexity Analysis

* **Add Employee**:
  + Best Case: O(1) (if there is space in the array)
  + Average Case: O(1) (with occasional resizing)
  + Worst Case: O(n) (if resizing the array, where n is the number of elements)
* **Search Employee**: O(n) (linear search)
* **Traverse Employees**: O(n) (iterating through all elements)
* **Delete Employee**: O(n) (finding the employee and shifting elements)

#### Limitations of Arrays

* **Fixed Size**: Initial size must be defined, and resizing (if implemented) can be costly in terms of time and memory.
* **Inefficient Insertions/Deletions**: Insertions and deletions are inefficient as they may require shifting elements.
* **Linear Search**: Searching is inefficient for large datasets (O(n) time complexity).

#### When to Use Arrays

* **Known Size**: When the number of elements is known and fixed.
* **Fast Access**: When quick access to elements via index is required.

**Memory Efficiency**: When minimal memory overhead is necessary.