# Exercise 4: Employee Management System

# 1. Understand Array Representation

**Array Representation in Memory:**

- Memory Layout: Arrays are stored in contiguous memory locations, allowing for efficient access to elements by index.

**- Advantages:**

- Constant Time Access: Accessing elements by index is O(1) due to contiguous memory.

- Efficient Space Usage: No overhead for pointers or additional structures.

- Simplicity: Easy to implement and understand with predictable performance.

**- Drawbacks:**

- Fixed Size: Arrays require a known size at creation, and resizing is needed for dynamic changes.

- Inefficient Insertions/Deletions: Adding or removing elements can be costly due to the need to shift elements, resulting in O(n) time complexity.

## 2. Setup

**Create a Class `Employee`:**

- Purpose: Represents an employee with attributes such as employee ID, name, position, and salary.

**- Methods:**

- Constructor: Initializes an `Employee` instance with the given attributes.

- Getters and Setters: Provide access to and modification of `employeeId`, `name`, `position`, and `salary`.

## 3. Implementation

**Use an Array to Store Employee Records:**

**1. Method: `addEmployee(Employee employee)`**

- Purpose: Adds a new employee to the array.

- Functionality: Checks if there is enough space in the array. If not, it resizes the array to accommodate more elements before adding the new employee.

**2. Method: `searchEmployee(String employeeId)`**

- Purpose: Searches for an employee by their ID.

- Functionality: Iterates through the array to find an employee with the matching ID. Returns the employee if found, or `null` if not found.

**3. Method: `traverseEmployees()`**

- Purpose: Displays all employees' details.

- Functionality: Iterates through the array and prints the information of each employee.

**4. Method: `deleteEmployee(String employeeId)`**

- Purpose: Deletes an employee by their ID.

- Functionality: Finds the index of the employee with the matching ID, then shifts the remaining elements to fill the gap left by the removed employee. Clears the last element and adjusts the size of the array.

**5. Method: `resize()`**

- Purpose: Resizes the array to accommodate more elements when it is full.

- Functionality: Creates a new, larger array, copies the existing elements to this new array, and updates the reference to the new array.

## 4. Analysis

**Time Complexity Analysis:**

- Add Operation:

- Average Case: O(1) when there is space available in the array.

- Worst Case: O(n) when resizing is necessary, as it involves copying all elements to a new array.

**- Search Operation:**

- Time Complexity: O(n) because searching through the array may require inspecting every element in the worst case.

**- Traverse Operation:**

- Time Complexity: O(n) as it involves visiting each element in the array once.

**- Delete Operation:**

- Average Case: O(n) because it includes finding the element (O(n)) and shifting elements (O(n)) to remove it.

**Limitations of Arrays:**

- Fixed Size: Requires a predefined size or resizing when full.

- Inefficient Insertions/Deletions: Both operations involve shifting elements, making them less efficient compared to more dynamic data structures like linked lists.

**When to Use Arrays:**

- Predictable Size: When the number of elements is known ahead of time or does not change frequently.

- Fast Access: When you need efficient, constant-time access to elements by index.

- Simple Data: Suitable for small or simple datasets where dynamic resizing or frequent insertions/deletions are not needed.

.