# Exercise 4: Employee Management System

### 

### **Understand Array Representation**

**Array Representation in Memory:**

* **Memory Layout:** Arrays are stored in contiguous memory locations. This means that elements are stored sequentially, one after another, which allows for efficient indexing and access.
* **Advantages:**
  + **Constant-Time Access:** Accessing an element by its index is very fast and takes constant time, O(1)O(1)O(1).
  + **Cache Friendly:** Due to their contiguous memory allocation, arrays have good locality of reference, which improves cache performance.

### **Step 2: Setup**

**Employee Class:**

class Employee {

int employeeId;

String name;

String position;

double salary;

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

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

// Getters

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 [employeeId=" + employeeId + ", name=" + name + ", position=" + position + ", salary=" + salary + "]";

}

}

### **Step 3: Implementation**

**Employee Management System using Array:**

public class EmployeeManagementSystem {

private Employee[] employees;

private int size;

public EmployeeManagementSystem(int capacity) {

employees = new Employee[capacity];

size = 0;

}

// Method to add an employee

public void addEmployee(Employee employee) {

if (size < employees.length) {

employees[size++] = employee;

} else {

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

}

}

// Method to search for an employee by employeeId

public Employee searchEmployee(int employeeId) {

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

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

return employees[i];

}

}

return null;

}

// Method to traverse all employees

public void traverseEmployees() {

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

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

}

}

// Method to delete an employee by employeeId

public void deleteEmployee(int employeeId) {

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

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

employees[i] = employees[size - 1]; // Replace with last employee

employees[size - 1] = null; // Nullify the last slot

size--;

return;

}

}

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

}

}

### **Step 4: Analysis**

**Time Complexity:**

* **Add Operation:**
  + Best Case: O(1)O(1)O(1) (when there is space in the array)
  + Average Case: O(1)O(1)O(1)
  + Worst Case: O(1)O(1)O(1)
* **Search Operation:**
  + Best Case: O(1)O(1)O(1) (if the employee is the first element)
  + Average Case: O(n)O(n)O(n) (linear search)
  + Worst Case: O(n)O(n)O(n) (if the employee is the last element or not found)
* **Traverse Operation:**
  + Best Case: O(n)O(n)O(n)
  + Average Case: O(n)O(n)O(n)
  + Worst Case: O(n)O(n)O(n)
* **Delete Operation:**
  + Best Case: O(1)O(1)O(1) (if the employee to be deleted is the last element)
  + Average Case: O(n)O(n)O(n) (linear search to find the employee)
  + Worst Case: O(n)O(n)O(n) (if the employee is the first element or not found)

**Limitations of Arrays:**

* **Fixed Size:** Arrays have a fixed size, which means you must know the maximum number of elements in advance.
* **Inefficient Deletion and Insertion:** Deleting or inserting elements (except at the end) requires shifting elements, which can be slow.
* **Wasted Space:** If the array is not fully used, it wastes memory.

**When to Use Arrays:**

* **Small to Medium Data:** When the data set is relatively small or medium-sized.
* **Read-Intensive Operations:** When you need fast access to elements via indices.
* **Static Data:** When the number of elements does not change frequently.

In cases where dynamic resizing, efficient insertions and deletions, and varying sizes are needed, other data structures like linked lists, dynamic arrays (ArrayLists in Java), or more advanced data structures (like trees or hash tables) might be more suitable.