Employee Management System - EX 4

1. Understanding Array Representation

Arrays in Memory:

- An array is a contiguous block of memory where each element is stored at a fixed offset from the previous one. The elements can be accessed directly using their index because the starting address of the array and the size of each element are known.
- **Advantages of Arrays**:
    - **Fast Access**: Arrays provide O(1) time complexity for accessing elements by index due to their contiguous memory allocation.
    - **Ease of Use**: They are simple to use and understand, making them a straightforward data structure for managing collections of elements.

2. Setup

Class Definition: Employee

The Employee class includes attributes like employeeId, name, position, and salary.

```
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;
   }

   public String 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;
    }
}
```

3. Implementation

The system uses an array to store employee records and provides methods for adding, searching, traversing, and deleting employees.

Array-Based Employee Management

```
public class EmployeeManagementSystem {
    private Employee[] employees;
    private int count;

    public EmployeeManagementSystem(int capacity) {
        employees = new Employee[capacity];
```

```
    count = 0;
}

public void addEmployee(Employee employee) {
    if (count < employees.length) {
        employees[count] = employee;
        count++;
        System.out.println("Employee added successfully.");
    } else {
        System.out.println("Employee list is full.");
    }
}

public Employee searchEmployee(String employeeId) {
    for (int i = 0; i < count; i++) {
        if (employees[i].getEmployeeId().equals(employeeId)) {
            return employees[i];
        }
    }
    return null;
}

public void traverseEmployees() {
    if (count == 0) {
        System.out.println("No employees in the list.");
    } else {
        for (int i = 0; i < count; i++) {
            System.out.println(employees[i]);
        }
```

```
    }
}

public void deleteEmployee(String employeeId) {
    int index = -1;
    for (int i = 0; i < count; i++) {
        if (employees[i].getEmployeeId().equals(employeeId)) {
            index = i;
            break;
        }
    }

    if (index != -1) {
        for (int i = index; i < count - 1; i++) {
            employees[i] = employees[i + 1];
        }
        employees[count - 1] = null;
        count--;
        System.out.println("Employee deleted successfully.");
    } else {
        System.out.println("Employee not found.");
    }
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    EmployeeManagementSystem ems = new EmployeeManagementSystem(5);

    while (true) {
```

```
System.out.println("\nEmployee Management System:");
System.out.println("1. Add Employee");
System.out.println("2. Search Employee");
System.out.println("3. Traverse Employees");
System.out.println("4. Delete Employee");
System.out.println("5. Exit");
System.out.print("Enter your choice: ");
int choice = scanner.nextInt();
scanner.nextLine();

switch (choice) {
    case 1:
        System.out.print("Enter Employee ID: ");
        String id = scanner.nextLine();
        System.out.print("Enter Employee Name: ");
        String name = scanner.nextLine();
        System.out.print("Enter Employee Position: ");
        String position = scanner.nextLine();
        System.out.print("Enter Employee Salary: ");
        double salary = scanner.nextDouble();
        scanner.nextLine();

        Employee newEmployee = new Employee(id, name, position, salary);
        ems.addEmployee(newEmployee);
        break;

    case 2:
        System.out.print("Enter Employee ID to search: ");
        String searchId = scanner.nextLine();
```

```
            Employee employee = ems.searchEmployee(searchId);
            if (employee != null) {
                System.out.println("Employee found: " + employee);
            } else {
                System.out.println("Employee not found.");
            }
            break;

        case 3:
            System.out.println("Employee List:");
            ems.traverseEmployees();
            break;

        case 4:
            System.out.print("Enter Employee ID to delete: ");
            String deleteId = scanner.nextLine();
            ems.deleteEmployee(deleteId);
            break;

        case 5:
            System.out.println("Exiting the system. Goodbye!");
            scanner.close();
            return;

        default:
            System.out.println("Invalid choice. Please try again.");
      }
    }
}
```

}

4. Analysis

Time Complexity Analysis:

- **Add**: O(1) if the array has space; O(n) if the array needs to be resized (which typically involves copying elements to a new array).
- **Search**: O(n) in the worst case since it requires a linear scan through the array.
- **Traverse**: O(n) since it involves visiting each element in the array.
- **Delete**: O(n) in the worst case, as elements may need to be shifted to fill the gap left by the deleted element.

Limitations of Arrays:

1. **Fixed Size**: The size of the array must be defined at creation, limiting flexibility. If more employees need to be added than the initial capacity, a new, larger array must be created, and existing data must be copied.
2. **Inefficient Insertions and Deletions**: Inserting or deleting elements from the middle of the array requires shifting other elements, leading to O(n) time complexity.
3. **Memory Utilization**: If the array is underutilized, it wastes memory; if it is overfilled, it requires resizing, which is costly in terms of performance.

When to Use Arrays:

- Arrays are suitable for scenarios where the number of elements is known and fixed or changes infrequently.
- They are beneficial when fast access to elements by index is required.
- For dynamic scenarios with frequent insertions and deletions, other data structures like linked lists, ArrayLists, or more complex structures like hash tables or trees may be more appropriate.