**Exercise 4: Employee Management System**

**Scenario:**

You are developing an employee management system for a company. Efficiently managing employee records is crucial.

**Understanding Array Representation**

Arrays are one of the most fundamental data structures used in software development. They are contiguous blocks of memory used to store elements of the same data type. Each element in an array is indexed, allowing for constant-time access O(1) to any element if its index is known. This characteristic makes arrays extremely efficient for traversing and accessing data.

In the context of an employee management system, arrays can be a simple and effective choice for storing a fixed number of employee records. Their simplicity and speed are major advantages when working with datasets of predictable or moderate size. However, one must be cautious, as arrays have a fixed size and lack flexibility for dynamic insertion and deletion — which is often required in real-world systems.

**Setup**

To begin with, we define a core data model: the Employee class. This class will serve as a blueprint for creating employee objects. It contains the following attributes:

* employeeId – A unique integer identifier for the employee.
* name – A string representing the employee's full name.
* position – The role or designation of the employee in the organization.
* salary – The monthly or annual salary of the employee.

This object-oriented design ensures encapsulation and provides a structured way to represent employee data within the system.

**Employee.java :-**

public class Employee {

int employeeId;

String name;

String position;

double salary;

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

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

public void display() {

System.out.println("ID: " + employeeId + ", Name: " + name + ", Position: " + position + ", Salary: " + salary);

}

}

**Implementation**

In this step, we manage the employee records using a simple array of Employee objects. Since arrays in Java are of fixed size, we maintain a counter to keep track of the number of employees currently stored.

**Key Operations:**

* **Add**: Inserts a new employee at the next available index.
* **Search**: Iterates through the array to find an employee by their employeeId.
* **Traverse**: Iterates over all active records and prints them.
* **Delete**: Removes an employee record by shifting the remaining elements left.

**Main.java :-**

public class Main {

public static void main(String[] args) {

// Create EmployeeManager with capacity for 5 employees

EmployeeManager manager = new EmployeeManager(5);

// Adding employees

manager.addEmployee(new Employee(101, "Alice", "Developer", 60000));

manager.addEmployee(new Employee(102, "Bob", "Designer", 55000));

manager.addEmployee(new Employee(103, "Charlie", "Manager", 75000));

System.out.println("All Employees:");

manager.displayAllEmployees();

// Searching for an employee

System.out.println("\nSearching for employee with ID 102:");

Employee e = manager.searchEmployee(102);

if (e != null) {

e.display();

} else {

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

}

// Deleting an employee

System.out.println("\nDeleting employee with ID 102...");

manager.deleteEmployee(102);

System.out.println("\nEmployees after deletion:");

manager.displayAllEmployees();

}

}

**EmployeeManager.java :-**

public class EmployeeManager {

private Employee[] employees;

private int count;

public EmployeeManager(int size) {

employees = new Employee[size];

count = 0;

}

public void addEmployee(Employee e) {

if (count < employees.length) {

employees[count++] = e;

} else {

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

}

}

public Employee searchEmployee(int id) {

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

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

return employees[i];

}

}

return null;

}

public void deleteEmployee(int id) {

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

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

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

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

}

employees[--count] = null;

return;

}

}

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

}

public void displayAllEmployees() {

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

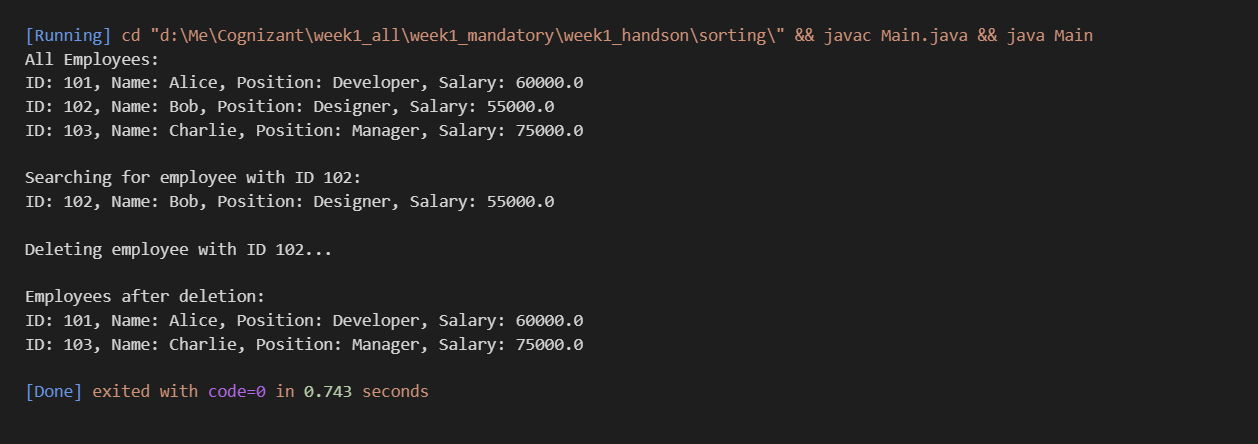
employees[i].display();

}

}

}

**Output :-**



**Analysis :-**

**Time Complexity Analysis:**

| **Operation** | **Time Complexity** | **Description** |
| --- | --- | --- |
| Add | O(1) | Direct insert at the next available index |
| Search | O(n) | Linear search through the array |
| Traverse | O(n) | Iterate through all elements |
| Delete | O(n) | Shift all elements after the deleted index |

**Limitations of Arrays:**

While arrays provide constant-time access and are efficient for fixed-size datasets, they suffer from several limitations:

* **Fixed Size**: Once declared, the size cannot be changed. This leads to inefficient memory usage if the size is overestimated or overflow issues if underestimated.
* **Costly Deletions/Insertions**: Removing or inserting elements in the middle of the array requires shifting elements, resulting in O(n) complexity.
* **No Built-in Dynamic Behaviour**: Arrays do not expand or contract based on the data volume.

**When to Use Arrays:**

* When the maximum number of elements is known in advance.
* When memory usage needs to be minimal and performance on indexed access is critical.
* When insertions and deletions are rare or occur only at the end of the structure.

In more dynamic scenarios, developers often prefer using ArrayList or other dynamic data structures from the Java Collections Framework which handle resizing and provide more flexibility.

**Conclusion**

The Employee Management System implemented using arrays provides a simple and efficient way to manage fixed-size records. While suitable for controlled environments, modern systems may benefit from dynamic structures when scalability and flexibility are required. The understanding of underlying time complexities and trade-offs helps developers make informed decisions regarding data structures in real-world applications.