**ALGORITHMS DATASTRUCTURES**

**Exercise 4: Employee Management System**

**Scenario:**

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

**Steps:**

1. **Understand Array Representation:**
   * Explain how arrays are represented in memory and their advantages.
2. **Setup:**
   * Create a class Employee with attributes like employeeId, name, position, and salary.
3. **Implementation:**
   * Use an array to store employee records.
   * Implement methods to add, search, traverse, and delete employees in the array.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).
   * Discuss the limitations of arrays and when to use them.

**ARRAY REPRESENTATION AND THEIRE ADVANTAGES:**

* **Contiguous Memory:** Arrays are stored in a block of contiguous memory locations.
* **Index-Based Access:** Each element is accessed directly using an index, making retrieval fast.
* **Advantages:**
  + Fast access (O(1)) to any element.
  + Simple and efficient for fixed-size collections.
  + Easy to implement basic operations.

**PROGRAM:**

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 String toString() {

return "ID: " + employeeId + ", Name: " + name + ", Position: " + position + ", Salary: ₹" + salary;

}

}

public class EmployeeManagementSystem {

static final int MAX = 100;

static Employee[] employees = new Employee[MAX];

static int count = 0;

public static void main(String[] args) {

addEmployee(new Employee(101, "Keerthi", "Developer", 45000));

addEmployee(new Employee(102, "Riya", "Manager", 60000));

addEmployee(new Employee(103, "Ajay", "Tester", 40000));

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

traverseEmployees();

System.out.println("\nSearch for ID 102:");

searchEmployee(102);

System.out.println("\nDeleting Employee ID 101:");

deleteEmployee(101);

traverseEmployees();

}

static void addEmployee(Employee e) {

if (count < MAX) {

employees[count++] = e;

}

}

static void traverseEmployees() {

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

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

}

}

static void searchEmployee(int id) {

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

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

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

return;

}

}

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

}

static 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;

}

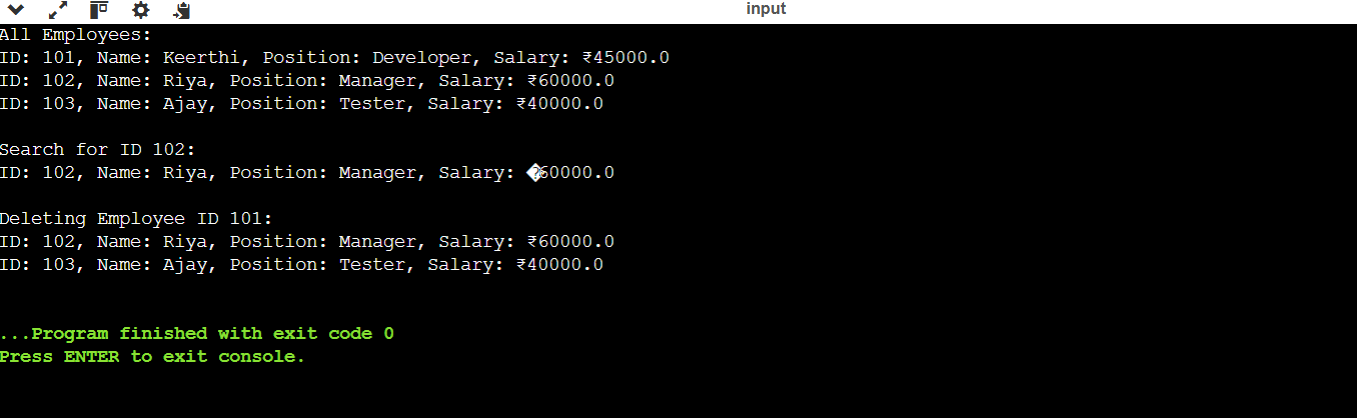
}

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

}

}

**OUTPUT:**

****

**ANALYSIS:**

**Time Complexity**

* **Add Employee: O(1)** Insert at the end using index count**.**
* **Search Employee: O(n)** Linear search through the array for matching employeeId.
* **Traverse Employees: O(n)** Visit each employee to display.
* **Delete Employee: O(n)**

**Space Complexity**

* O(n)  
  Space grows with the number of employees (n entries).
* Fixed Allocation  
  Memory is pre-allocated which can lead to unused space.

**LIMITATIONS OF ARRAY:**

* **Fixed Size**:
  + You must define the size upfront; cannot grow or shrink dynamically.
* **Costly Insertions/Deletions**:
  + Inserting or deleting elements (especially in the middle) requires shifting, which takes **O(n)** time.
* **Wasted Memory**:
  + If the array is not fully filled, unused space leads to memory wastage.
* **Limited Flexibility**:
  + Difficult to implement advanced operations like sorting or filtering without extra logic or structures.

**WHEN TO USE ARRAYS:**

* **Known Fixed Size**:
  + Best when the number of elements is predetermined and constant.
* **Fast Element Access**:
  + Use when frequent **index-based access** (O(1)) is required.
* **Memory-Efficient for Simple Tasks**:
  + Suitable for lightweight, low-overhead data storage in small applications.
* **When Using Primitive Types**:
  + Arrays work well with int, char, etc., as they avoid object overhead.