**DATASTRUCTURES AND ALGORITHMS:**

**Exercise-4:Employee Management System:**

**Array Representation**

Arrays are a basic data structure that stores elements in **contiguous memory locations**. This means the elements are placed one after the other in memory. Each element is accessed using an index that starts from 0.

**Advantages of arrays:**

* **Fast access:** We can directly access any element using its index in constant time O(1).
* **Simple structure:** Easy to declare and use in programs.
* **Efficient memory allocation:** Since memory is continuous, it reduces overhead compared to some other structures.

**Employee Class**

We create a class Employee with the following attributes:

* employeeId (int)
* name (String)
* position (String)
* salary (double)

**Employee.java**

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;

}

void display() {

System.out.println("ID: " + employeeId + ", Name: " + name +

", Position: " + position + ", Salary: " + salary);

}

}

**EmployeeManagement.java**

public class EmployeeManagement {

Employee[] employees;

int count;

public EmployeeManagement(int size) {

employees = new Employee[size];

count = 0;

}

public void addEmployee(Employee emp) {

if (count < employees.length) {

employees[count++] = emp;

} else {

System.out.println("No space to add more employees.");

}

}

// Search employee by ID

public Employee searchEmployee(int id) {

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

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

return employees[i];

}

}

return null;

}

// Display all employees

public void displayEmployees() {

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

employees[i].display();

}

}

// Delete employee by ID

public void deleteEmployee(int id) {

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

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

// Shift elements to the left

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

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

}

employees[--count] = null;

System.out.println("Employee deleted.");

return;

}

}

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

}

public static void main(String[] args) {

EmployeeManagement system = new EmployeeManagement(5);

system.addEmployee(new Employee(1, "Alice", "Developer", 60000));

system.addEmployee(new Employee(2, "Bob", "Manager", 80000));

system.addEmployee(new Employee(3, "Charlie", "Analyst", 50000));

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

system.displayEmployees();

System.out.println("\nSearching for Employee with ID 2:");

Employee e = system.searchEmployee(2);

if (e != null) {

e.display();

} else {

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

}

System.out.println("\nDeleting Employee with ID 1:");

system.deleteEmployee(1);

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

system.displayEmployees();

}

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**Limitations of Arrays:**

* **Fixed size:** We cannot change the size once it's defined.
* **Slow deletions/insertions:** Deleting or inserting in the middle requires shifting elements.
* **Wasted memory:** If the array is too large and not fully used

**Time complexity of operations:**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Add | O(1) | Add at the end using count index |
| Search | O(n) | May have to check all elements in worst case |
| Traverse | O(n) | Must visit each employee |
| Delete | O(n) | Find the element and shift remaining elements |
|  |  |  |
|  |  |  |

**When to Use Arrays:**

* When we **know the number of employees** in advance.
* When we **need fast access by index**.
* When operations like add/delete are **infrequent** or mostly done at the end.