**Exercise 4: Employee Management System**

**Scenario:**

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

**1. Understanding Array Representation**

**Array Representation in Memory**

* **Structure:** Arrays are a collection of elements stored in contiguous memory locations. All elements are of the same type and accessed via an index.

**Advantages**

* **Constant Time Access:** Accessing an element by its index is O(1), making it very fast.
* **Simple Structure:** Easy to implement and manage, particularly for fixed-size collections.
* **Cache Friendly:** Contiguous storage of elements makes array access patterns efficient for the CPU cache, enhancing performance.

**2. Setup:**

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;

}

@Override

public String toString() {

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

}

}

**3. Implementation:**

public class EmployeeManagementSystem {

private Employee[] employees;

private int size;

public EmployeeManagementSystem(int capacity) {

employees = new Employee[capacity];

size = 0;

}

// Add an employee

public void addEmployee(Employee employee) {

if (size < employees.length) {

employees[size++] = employee;

} else {

System.out.println("Array is full. Cannot add more employees.");

}

}

// Search for an employee by ID

public Employee searchEmployee(int employeeId) {

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

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

return employees[i];

}

}

return null; // Employee not found

}

// Traverse and print all employees

public void traverseEmployees() {

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

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

}

}

// Delete an employee by ID

public void deleteEmployee(int employeeId) {

int index = -1;

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

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

index = i;

break;

}

}

if (index != -1) {

for (int i = index; i < size - 1; i++) {

employees[i] = employees[i + 1];

}

employees[--size] = null;

} else {

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

}

}

public static void main(String[] args) {

EmployeeManagementSystem ems = new EmployeeManagementSystem(5);

ems.addEmployee(new Employee(1, "Alice", "Manager", 75000));

ems.addEmployee(new Employee(2, "Bob", "Developer", 65000));

ems.addEmployee(new Employee(3, "Charlie", "Analyst", 55000));

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

ems.traverseEmployees();

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

Employee emp = ems.searchEmployee(2);

System.out.println(emp != null ? emp : "Employee not found");

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

ems.deleteEmployee(2);

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

ems.traverseEmployees();

}

}

**4. Analysis**

**Time Complexity**:

* **Add**: O(1) (when there is space in the array; otherwise, it can be O(n) if resizing is needed).
* **Search**: O(n) (requires scanning through the array).
* **Traverse**: O(n) (needs to visit each element).
* **Delete**: O(n) (requires shifting elements to fill the gap left by the deleted employee).

**Limitations of Arrays**:

* **Fixed Size**: Arrays have a fixed size, which can be limiting if the number of records changes frequently.
* **Insertion and Deletion**: Adding or removing elements involves shifting elements and may not be efficient for large datasets.

**When to Use Arrays**:

* When the number of records is known and fixed or changes infrequently.
* For applications requiring fast random access and simple structure.