**Exercise 4: Employee Management System**

**Step 1: Understand Array Representation**

**How Arrays Are Represented in Memory**

* **Contiguous Memory Allocation:** Arrays are stored in contiguous memory locations, which means that each element is placed next to the previous one in memory.
* **Index-Based Access:** Each element in an array can be accessed using its index. This allows for O(1) time complexity for access operations.
* **Fixed Size:** The size of the array is defined at the time of its creation and cannot be changed.

**Advantages of Arrays**

* **Fast Access:** O(1) time complexity for accessing elements by index.
* **Predictable Performance:** Consistent time complexity for access and traversal operations.
* **Simple Data Structure:** Easy to implement and understand.

**Setup**

Create a class Employee with attributes like employeeId, name, position, and salary.

public 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;

}

// Getters and setters

public String getEmployeeId() {

return employeeId;

}

public void setEmployeeId(String employeeId) {

this.employeeId = employeeId;}

public String getName() {

return name;}

public void setName(String name) {

this.name = name;}

public String getPosition() {

return position;}

public void setPosition(String position) {

this.position = position;}

public double getSalary() {

return salary;}

public void setSalary(double salary) {

this.salary = salary;}}

**Implementation**

public class EmployeeManagement {

private Employee[] employees;

private int count;

public EmployeeManagement(int capacity) {

employees = new Employee[capacity];

count = 0;}

public void addEmployee(Employee employee) {

if (count < employees.length) {

employees[count] = employee;

count++;

} else {

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

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() {

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

System.out.println(employees[i].getName() + " - " + employees[i].getPosition() + " - $" + employees[i].getSalary());}}

public void deleteEmployee(String employeeId) {

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

if (employees[i].getEmployeeId().equals(employeeId)) {

employees[i] = employees[count - 1];

employees[count - 1] = null;

count--;

return;

}

}

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

}

public static void main(String[] args) {

EmployeeManagement management = new EmployeeManagement(5);

// Adding employees

Employee emp1 = new Employee("001", "Alice", "Manager", 80000);

Employee emp2 = new Employee("002", "Bob", "Engineer", 60000);

Employee emp3 = new Employee("003", "Charlie", "Technician", 40000);

management.addEmployee(emp1);

management.addEmployee(emp2);

management.addEmployee(emp3);

// Traversing employees

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

management.traverseEmployees();

// Searching for an employee

System.out.println("\nSearching for Employee with ID '002':");

Employee result = management.searchEmployee("002");

System.out.println(result != null ? result.getName() + " - " + result.getPosition() + " - $" + result.getSalary() : "Employee not found");

// Deleting an employee

System.out.println("\nDeleting Employee with ID '002':");

management.deleteEmployee("002");

// Traversing employees after deletion

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

management.traverseEmployees();}

}

**Analysis**

**Time Complexity of Operations**

* **Add Employee:** O(1)
  + Adding an employee is a constant time operation as it involves inserting the employee at the next available index.
* **Search Employee:** O(n)
  + In the worst case, the search operation requires checking each element in the array.
* **Traverse Employees:** O(n)
  + Traversing all employees requires visiting each element in the array once.
* **Delete Employee:** O(n)
  + Deletion involves searching for the employee and then removing it, which can take up to O(n) time.

**Limitations of Arrays**

* **Fixed Size:** Once an array is created, its size cannot be changed. This limits the flexibility to accommodate dynamic changes in the number of employees.
* **Inefficient Deletions:** Deleting an element requires shifting elements to fill the gap, which can be inefficient for large arrays.
* **Memory Wastage:** If the array is not fully utilized, it leads to memory wastage.

**When to Use Arrays**

* **Small Dataset:** Arrays are suitable for managing a small and fixed number of records where the overhead of dynamic data structures is not justified.
* **Fast Access:** Arrays are ideal when fast access to elements by index is required.
* **Memory Contiguity:** When memory contiguity is important, arrays provide this advantage.