Step 1: Understand Array Representation

Arrays are a fundamental data structure in programming, and understanding how they are represented in memory is essential.

Array Representation in Memory:

In memory, an array is a contiguous block of memory locations, each of which can store a value of a specific data type. The array is stored in a single block of memory, and each element is accessed using an index or subscript.

Advantages of Arrays:

Efficient Memory Usage: Arrays store elements in contiguous memory locations, making them memory-efficient.

Fast Access: Arrays provide fast access to elements using their index.

Simple Implementation: Arrays are easy to implement and understand.

Step 2: Setup

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

public class Employee {

private int employeeId;

private String name;

private String position;

private double salary;

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

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

// Getters and setters

public int getEmployeeId() {

return employeeId;

}

public void setEmployeeId(int 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;

}

}

Step 3: Implementation

public class EmployeeManagementSystem {

private Employee[] employees;

private int size;

public EmployeeManagementSystem(int capacity) {

employees = new Employee[capacity];

size = 0;

}

public void addEmployee(Employee employee) {

if (size < employees.length) {

employees[size] = employee;

size++;

} else {

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

}

}

public Employee searchEmployee(int employeeId) {

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

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

return employees[i];

}

}

return null;

}

public void traverseEmployees() {

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

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

}

}

public void deleteEmployee(int employeeId) {

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

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

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

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

}

size--;

return;

}

}

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

}

}

Step 4: Analysis

Time Complexity Analysis:

Add Employee: O(1) - Adding an employee takes constant time.

Search Employee: O(n) - Searching for an employee takes linear time, where n is the number of employees.

Traverse Employees: O(n) - Traversing employees takes linear time, where n is the number of employees.

Delete Employee: O(n) - Deleting an employee takes linear time, where n is the number of employees.

Limitations of Arrays:

Fixed Size: Arrays have a fixed size, which can lead to memory waste or insufficient space.

Slow Search and Delete: Searching and deleting elements in an array can be slow for large datasets.

No Dynamic Resizing: Arrays cannot be dynamically resized, which can lead to memory issues.

When to Use Arrays:

Small, Fixed-Size Datasets: Arrays are suitable for small, fixed-size datasets where memory efficiency is crucial.

Cache-Friendly Access: Arrays provide fast access to elements using their index, making them suitable for cache-friendly access patterns.

Simple Implementation: Arrays are easy to implement and understand, making them a good choice for simple applications.

In conclusion, arrays are a fundamental data structure in programming, and understanding their representation in memory, advantages, and limitations is essential for efficient employee management system development.