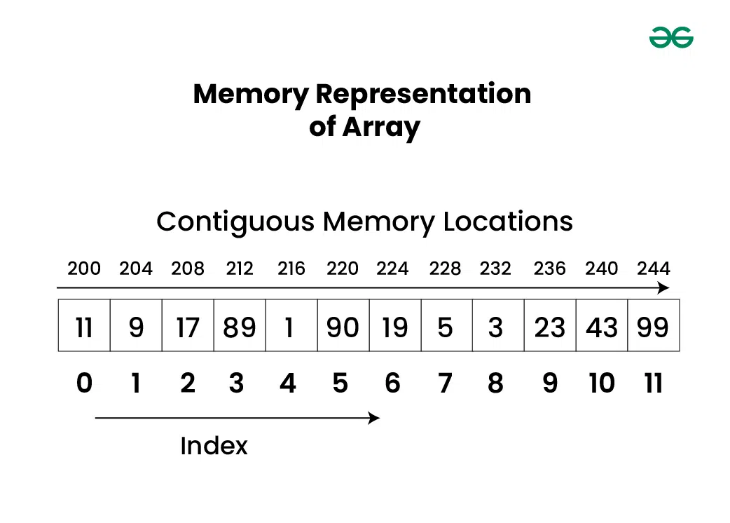
**Exercise 4: Employee Management System**

1. **Understand Array Representation:**

* *Explain how arrays are represented in memory and their advantages.*

Ans. In an array, all the elements are stored in contiguous memory locations. So, if we initialize an array, the elements will be allocated sequentially in memory.

This allows for efficient access and manipulation of elements.



1. **Setup:**

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

1. **Implementation:**

* Use an array to store employee records.
* Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.

Code:

package Algorithm\_DataStructures.Exercise4.Employee\_Management\_System;

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;

    }

    public int getEmployeeId() {

        return employeeId;

    }

    public String getName() {

        return name;

    }

    public String getPosition() {

        return position;

    }

    public double getSalary() {

        return salary;

    }

    @Override

    public String toString() {

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

    }

}

public class EMS {

    private Employee[] employees;

    private int size;

    public EMS(int capacity) {

        employees = new Employee[capacity];

        size = 0;

    }

    public void addEmployee(Employee employee) {

        if (size < employees.length) {

            employees[size++] = employee;

            System.out.println("Employee added successfully.");

        } else {

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

        }

    }

    public Employee searchEmployee(int employeeId) {

        for (Employee employee : employees) {

            if (employee != null && employee.getEmployeeId() == employeeId) {

                return employee;

            }

        }

        return null;

    }

    public void traverseEmployees() {

        for (Employee employee : employees) {

            if (employee != null) {

                System.out.println(employee);

            }

        }

    }

    public void deleteEmployee(int employeeId) {

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

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

                System.out.println("Employee deleted: " + employees[i]);

                employees[i] = null;

                // Move the last employee record to the deleted position

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

                employees[size - 1] = null;

                size--;

                return;

            }

        }

        System.out.println("Employee with ID " + employeeId + " not found.");

    }

    public static void main(String[] args) {

       EMS empManagement = new EMS(5);

        empManagement.addEmployee(new Employee(101, "Alice", "Manager", 50000.0));

        empManagement.addEmployee(new Employee(202, "Bob", "Developer", 40000.0));

        empManagement.addEmployee(new Employee(303, "Charlie", "Analyst", 35000.0));

        System.out.println("\nTraversing all employees:");

        empManagement.traverseEmployees();

        System.out.println("\nSearch for employee with ID 202:");

        Employee foundEmployee = empManagement.searchEmployee(202);

        if (foundEmployee != null) {

            System.out.println("Employee found: " + foundEmployee);

        } else {

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

        }

        System.out.println("\nDeleting employee with ID 202:");

        empManagement.deleteEmployee(202);

        System.out.println("\nTraversing all employees after deletion:");

        empManagement.traverseEmployees();

    }

}

1. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).

Ans. **Time Complexity Analysis:**

1. **Add Operation:**
   * Time Complexity: O(1) (Amortized)
   * Adding an employee to the array has a constant time complexity of O(1) in the average case. However, if the array needs to be resized due to reaching its capacity, the time complexity becomes O(n) where n is the number of elements in the array.
2. **Search Operation:**
   * Time Complexity: O(n)
   * Searching for an employee in the array requires iterating over all elements (in the worst case) until the target employee is found. This results in a time complexity of O(n) where n is the number of elements in the array.
3. **Traverse Operation:**
   * Time Complexity: O(n)
   * Traversing all employees in the array involves visiting each element once, resulting in a time complexity of O(n) where n is the number of elements in the array.
4. **Delete Operation:**
   * Time Complexity: O(n)
   * Deleting an employee involves searching for the employee in the array (O(n) time complexity) and then potentially shifting elements in the array to fill the gap left by the deleted employee (O(n) time complexity in the worst case).

* Discuss the limitations of arrays and when to use them.

Ans. **Limitations of Arrays:**

1. **Fixed Size:**
   * Arrays have a fixed size, which means you need to know the maximum number of elements that will be stored in advance. Resizing an array can be expensive as it involves creating a new array and copying existing elements.
2. **Contiguous Memory Allocation:**
   * Arrays require contiguous memory allocation, which can lead to memory fragmentation issues, especially when dealing with large arrays.
3. **Limited Operations:**
   * Arrays are not dynamic data structures and do not support efficient insertion or deletion of elements at arbitrary positions without shifting elements.

**When to Use Arrays:**

* + Arrays are suitable when the size of the data is known in advance and doesn't change frequently.
  + Use arrays when fast random access to elements by index is required.
  + Arrays are efficient for simple data storage and retrieval operations, especially when the number of elements is small and doesn't change often.