**Exercise 4: Employee Management System**

**Scenario:**

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

1. Understand Array Representation:

* Explain how arrays are represented in memory and their advantages.
* **Array representation in memory:**
* **Contiguous Memory Allocation:** Arrays are stored in contiguous memory locations. This means that all elements of an array are stored in a block of memory that is next to each other.
* When an array is declared, a single block of memory is allocated to hold all its elements.
* The starting address of the array is stored in a variable (array reference), and each element can be accessed by its offset from this base address.
* For an array of type int with 5 elements, the memory is allocated as a single contiguous block where each integer occupies a fixed number of bytes (typically 4 bytes on most systems).
* **Index-Based Access**
* **Access Formula**: Accessing an element in the array is done using the formula: Address of element[i]=base address+(i×size of each element) where i is the index of the element and size of each element is the size of the data type (e.g., 4 bytes for int).

* **Advantages**:
* Direct Indexing: Constant time complexity for access.
* Efficient Memory Usage: Better memory and cache utilization.
* Ease of Implementation: Simple and straightforward to use.
* Predictable Size: Fixed size makes allocation straightforward.
* Batch Operations: Facilitates operations over collections of elements.
* Memory Access Patterns: Optimized by CPU caching mechanisms.
* These characteristics make arrays a versatile and powerful data structure for a wide range of programming tasks.

2. Setup:

* + **Class Creation**: Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.

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

}

}

* **class Employee:** Defines a class named Employee.

**Instance variables (or attributes) of the Employee class:**

* **int employeeId:** An integer attribute to store the unique identifier for each employee.
* **String name:** A string attribute to store the name of the employee.
* **String position:** A string attribute to store the position where the employee works.
* **double salary:** A double attribute to store the salary of the employee.

**Constructor:**

* **public Employee(int employeeId, String name, String position, double salary):** A constructor that initializes a Employee object with the provided employeeId, name, position, and salary.
* **this.employeeId = employeeId**;: Assigns the value of the parameter employeeId to the instance variable employeeId.
* **this.name = name:** Assigns the value of the parameter name to the instance variable name.
* **this.position = position;:** Assigns the value of the parameter position to the instance variable position.
* **this.salary = salary;:** Assigns the value of the parameter salary to the instance variable salary.

3. Implementation:

* + Use an array to store employee records.

public class EmployeeManagementSystem {

private static Employee[] employees = new Employee[100];

private static int employeeCount = 0;

* **employees**: An array to store Employee objects.
* **employeeCount**: A counter to track the number of employees in the array.
  + Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array:

public static void addEmployee(Scanner scanner) {

if (employeeCount >= employees.length) {

System.out.println("Employee array is full.");

return;

}

System.out.println("Enter Employee ID:");

int id = scanner.nextInt();

scanner.nextLine();

System.out.println("Enter Name:");

String name = scanner.nextLine();

System.out.println("Enter Position:");

String position = scanner.nextLine();

System.out.println("Enter Salary:");

double salary = scanner.nextDouble();

employees[employeeCount++] = new Employee(id, name, position, salary);

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

}

public static void searchEmployee(Scanner scanner) {

System.out.println("Enter Employee ID to search:");

int id = scanner.nextInt();

boolean found = false;

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

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

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

found = true;

break;

}

}

if (!found) {

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

}

}

public static void traverseEmployees() {

if (employeeCount == 0) {

System.out.println("No employees to display.");

return;

}

System.out.println("Employee Records:");

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

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

}

}

public static void deleteEmployee(Scanner scanner) {

System.out.println("Enter Employee ID to delete:");

int id = scanner.nextInt();

boolean found = false;

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

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

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

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

}

employees[--employeeCount] = null;

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

found = true;

break;

}

}

* **addEmployee**: Adds a new employee to the array.
* **searchEmployee**: Searches for an employee by employeeId.
* **traverseEmployees**: Displays all employees in the array.
* **deleteEmployee**: Deletes an employee based on employeeId and shifts the remaining elements to fill the gap

Here is the github repo link –

4. Analysis:

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

1. **Check Array Fullness:**
   * if (employeeCount >= employees.length) checks if the array is full.
   * This comparison operation is O(1) (constant time).
2. **Reading User Input:**
   * scanner.nextInt() reads an integer.
   * scanner.nextLine() reads a string.
   * scanner.nextDouble() reads a double.
   * These input operations are generally O(1) each, assuming average cases and standard input handling.
3. **Array Operation:**
   * employees[employeeCount++] = new Employee(id, name, position, salary):
     + TaskNode current = head;
     + Creating a new Employee object involves creating an object and assigning its attributes, which is also O(1) for each operation.
     + The assignment operation (employees[employeeCount++]) is an array write operation that occurs in constant time O(1).

**Overall Time Complexity**

* Checking if the array is full: O(1)
* Reading user input: O(1)
* Creating and adding a new Employee object: O(1)
* Search:

1. **Reading User Input**:
   * scanner.nextInt() reads an integer.
   * This operation is O(1) (constant time).
2. **Searching the Array**:
   * The for loop iterates over the array employees from 0 to employeeCount - 1.
   * In the worst case, the loop may need to examine every element in the array to find the employee with the specified id, or to determine that the employee is not present.
   * The comparison employees[i].employeeId == id inside the loop is O(1) for each element.
   * Thus, the time complexity for searching through the array is O(n), where n is the number of employees in the array (employeeCount).
3. **Early Exit**:
   * If the employee is found before the end of the array, the break statement will exit the loop early.
   * The worst-case scenario still assumes that the loop could run through all elements before finding the employee or confirming their absence.

**Overall Time Complexity**

* Reading user input: O(1)
* Searching the array: O(n), where n is employeeCount
* Traverse:

1. **Checking Array Emptiness**:
   * if (employeeCount == 0): This is a constant-time operation, O(1), as it only involves a comparison.
2. **Traversing the Array**:
   * The for loop iterates over the array employees from 0 to employeeCount - 1.
   * During each iteration, System.out.println(employees[i]) prints the details of each employee.
   * The operation inside the loop (System.out.println(employees[i])) is generally considered to be O(1) for each employee, assuming that printing an object is constant time relative to the object’s size.
   * The loop executes employeeCount times, so the overall time complexity for this part is O(n), where n is employeeCount.

**Overall Time Complexity**

* Checking array emptiness: O(1)
* Traversing the array and printing records: O(n), where n is employeeCount.
* Delete:

1. **Reading User Input:**
   * scanner.nextInt(): This operation is O(1) (constant time).
2. **Finding the Employee**:
   * The for loop iterates over the employees array to find the employee with the specified id.
   * In the worst case, it may need to examine all elements in the array until it finds the employee or determines that the employee is not present.
   * The time complexity of this operation is O(n), where n is employeeCount.
3. **Shifting Elements**:
   * Once the employee is found, the second for loop starts at the position i and shifts all subsequent elements one position to the left to fill the gap.
   * This loop runs from i to employeeCount - 1, so in the worst case, it iterates over approximately n - i elements.
   * If the employee is found at the beginning of the array, this operation involves shifting almost all remaining elements, so the worst-case time complexity for this step is O(n).
4. **Updating Array Size**:
   * The operation employees[--employeeCount] = null is a constant-time operation, O(1).

**Overall Time Complexity**

* Finding the employee: O(n)
* Shifting elements: O(n) (in the worst case)
* Updating array size: O(1)
* Discuss the limitations of arrays and when to use them.

 **Limitations of Arrays**

* 1. **Fixed Size:**

Description: Arrays have a fixed size determined at the time of creation. Once an array is initialized, its size cannot be changed.

Implication: This limitation means that if the array is too small, we will need to create a new, larger array and copy the data over, which can be inefficient. Conversely, if the array is too large, we may waste memory.

* 1. **Inefficient Insertions and Deletions:**

Description: Inserting or deleting elements in the middle of an array requires shifting elements to accommodate the change.

Implication: These operations can be inefficient, especially for large arrays, as they involve moving multiple elements. This makes arrays less suitable for applications requiring frequent insertions or deletions.

* 1. **Memory Allocation:**

Description: Arrays allocate memory contiguously.

Implication: This can be inefficient if the array is very large or if the array's size is not known in advance. Fragmentation or insufficient contiguous memory space can lead to problems.

* 1. **Lack of Built-in Methods:**

Description: Arrays do not come with built-in methods for common operations like resizing, searching, or sorting (beyond basic sorting algorithms).

Implication: we have to implement or use additional libraries for such operations, which can increase development time and complexity.

* 1. **No Bounds Checking:**

Description: Arrays in many languages do not automatically check for index bounds.

Implication: Accessing an index outside the valid range can lead to runtime errors or undefined behavior.

 **When to use Arrays**

* 1. **When Size is Known and Fixed:**

Use Case: When we know the exact number of elements we need and it doesn’t change, arrays are a good choice. For example, storing the days of the week or the months of the year.

* 1. **When Performance is Critical:**

Use Case: Arrays provide constant-time access to elements by index, which can be beneficial in performance-critical applications where fast retrieval is needed.

* 1. **For Simple Data Structures:**

Use Case: Arrays are ideal for simple data structures where complex operations (insertions, deletions) are not required. For example, in cases where we need to implement a fixed-size buffer or a simple list of items.

* 1. **When Working with Primitive Data Types:**

Use Case: Arrays can be used efficiently for storing and manipulating primitive data types (e.g., int, char, float) as they are directly managed in memory without overhead.

* 1. **When Memory Efficiency is a Concern:**

Use Case: Arrays can be more memory-efficient than some other data structures because they store data in a contiguous block of memory. For example, arrays are preferred in situations where memory layout and performance are critical, like in embedded systems or low-level programming.

Output:





