Exercise 4: Employee Management System

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

**Array Representation in Memory:**

* **Memory Layout:** Arrays are stored in contiguous memory locations. This means that elements are stored sequentially, one after another, which allows for efficient indexing and access.
* **Advantages:**
  + **Constant-Time Access:** Accessing an element by its index is very fast and takes constant time, O(1).
  + **Cache Friendly:** Due to their contiguous memory allocation, arrays have good locality of reference, which improves cache performance.

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

public class Employee {

private int id;

private String name;

private String position;

private double salary;

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

this.name = name;

this.id = id;

this.position = position;

this.salary = salary;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

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;

}

public String toString(){

return this.id + " " + this.name + " " + this.salary;

}

}

1. Employee Management System using Array:

public class EmployeeManager {

Employee[] employees;

public int capacity;

public int size;

EmployeeManager(int capacity){

this.capacity = capacity;

employees = new Employee[capacity];

size = 0;

}

public void add(Employee employee){

if(size == capacity - 1 ){

System.out.println("Array is full !");

return;

}

employees[size] = employee;

size++;

}

public Employee search(int id){

for(Employee employee : this.employees){

if(employee.getId() == id)

return employee;

}

return null;

}

public void traverse(){

System.out.printf("%-20s %-20s %-20s %s\n","Employee Id","Employee Name","Position","Salary");

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

Employee employee = employees[i];

System.out.printf("%-20s %-20s %-20s %s\n",employee.getId(),employee.getName(),employee.getPosition(),employee.getSalary());

}

}

public void delete(int id){

int j = 0;

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

int currId = employees[i].getId();

if(currId != id)

employees[j++] = employees[i];

}

while(j < capacity)

employees[j++] = null;

size--;

}

}

1. Analysis

**Time Complexity:**

* **Add Operation:**
  + Best Case: O(1) (when there is space in the array)
  + Average Case: O(1)
  + Worst Case: O(1)
* **Search Operation:**
  + Best Case: O(1) (if the employee is the first element)
  + Average Case: O(n) (linear search)
  + Worst Case: O(n) (if the employee is the last element or not found)
* **Traverse Operation:**
  + Best Case: O(n)
  + Average Case: O(n)
  + Worst Case: O(n)
* **Delete Operation:**
  + Best Case: O(1) (if the employee to be deleted is the last element)
  + Average Case: O(n) (linear search to find the employee)
  + Worst Case: O(n) (if the employee is the first element or not found)

**Limitations of Arrays:**

* **Fixed Size:** Arrays have a fixed size, which means you must know the maximum number of elements in advance.
* **Inefficient Deletion and Insertion:** Deleting or inserting elements (except at the end) requires shifting elements, which can be slow.
* **Wasted Space:** If the array is not fully used, it wastes memory.

**When to Use Arrays:**

* **Small to Medium Data:** When the data set is relatively small or medium-sized.
* **Read-Intensive Operations:** When you need fast access to elements via indices.
* **Static Data:** When the number of elements does not change frequently.