Employee Management System Using Arrays

1. Array Representation in Memory

Memory Representation:

Arrays are stored as contiguous blocks of memory where:

* Each element occupies the same amount of space
* Elements are accessed via index using the formula: memory\_address = base\_address + (index \* element\_size)
* The array variable holds a reference to the first element's memory location

Advantages of Arrays:

1. **Fast Random Access**: O(1) access time for any element by index
2. **Memory Efficiency**: No overhead for pointers/links (unlike linked lists)
3. **Cache Friendliness**: Contiguous memory improves cache performance
4. **Simplicity**: Easy to implement and understand

2. Implementation

public class EmployeeManagementSystem {

private Employee[] employees;

private int count; // Tracks number of employees currently stored

public EmployeeManagementSystem(int initialCapacity) {

employees = new Employee[initialCapacity];

count = 0;

}

// Employee class

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 String getName() { return name; }

public String getPosition() { return position; }

public double getSalary() { return salary; }

@Override

public String toString() {

return String.format("ID: %d, Name: %s, Position: %s, Salary: $%.2f",

employeeId, name, position, salary);

}

}

public void addEmployee(Employee emp) {

if (count == employees.length) {

resizeArray();

}

employees[count++] = emp;

}

public Employee searchById(int employeeId) {

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

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

return employees[i];

}

}

return null;

}

public void traverseEmployees() {

System.out.println("\nCurrent Employees:");

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

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

}

}

public boolean deleteEmployee(int employeeId) {

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

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

// Shift all elements after i one position left

System.arraycopy(employees, i + 1, employees, i, count - i - 1);

count--;

employees[count] = null; // Clear last reference

return true;

}

}

return false;

}

private void resizeArray() {

Employee[] newArray = new Employee[employees.length \* 2];

System.arraycopy(employees, 0, newArray, 0, employees.length);

employees = newArray;

}

public static void main(String[] args) {

EmployeeManagementSystem ems = new EmployeeManagementSystem(5);

// Add employees

ems.addEmployee(ems.new Employee(101, "John Doe", "Developer", 75000));

ems.addEmployee(ems.new Employee(102, "Jane Smith", "Manager", 90000));

ems.addEmployee(ems.new Employee(103, "Bob Johnson", "Designer", 65000));

// Traverse and display

ems.traverseEmployees();

// Search for an employee

Employee emp = ems.searchById(102);

System.out.println("\nFound employee: " + emp);

// Delete an employee

System.out.println("\nDeleting employee 101...");

ems.deleteEmployee(101);

ems.traverseEmployees();

// Add more employees to test resizing

ems.addEmployee(ems.new Employee(104, "Alice Brown", "QA Engineer", 70000));

ems.addEmployee(ems.new Employee(105, "Mike Davis", "DevOps", 80000));

ems.addEmployee(ems.new Employee(106, "Sarah Wilson", "Product Manager", 95000));

ems.traverseEmployees();

}

}

3. Time Complexity Analysis

| **Operation** | **Time Complexity** | **Notes** |
| --- | --- | --- |
| Add | O(1) amortized | O(n) when resizing needed |
| Search by ID | O(n) | Linear search through array |
| Traverse | O(n) | Must visit each element |
| Delete | O(n) | Search + shift elements |

4. Limitations of Arrays and When to Use Them

Limitations:

1. **Fixed Size**: Need to resize manually when capacity is exceeded
2. **Insertion/Deletion**: Expensive operations requiring element shifting
3. **Memory Waste**: May allocate more memory than needed
4. **Homogeneous**: Can only store elements of the same type

When to Use Arrays:

1. **Known Size**: When maximum number of elements is known in advance
2. **Frequent Access**: When random access by index is the primary operation
3. **Memory Constraints**: When memory efficiency is critical
4. **Simple Data**: For basic collections without complex relationships

When to Consider Alternatives:

1. **Frequent Insertions/Deletions**: Linked lists may be better
2. **Dynamic Sizing**: ArrayLists (Java) or Vectors (C++) handle resizing automatically
3. **Key-Value Pairs**: HashMaps/Dictionaries for O(1) access by key
4. **Frequent Searching**: Binary search trees for O(log n) searches

Optimization Recommendations

1. **For Small Systems** (<100 employees): This array implementation is sufficient
2. **For Medium Systems** (100-10,000 employees): Use Java's ArrayList
3. **For Large Systems** (>10,000 employees): Consider database with indexing
4. **For High Search Needs**: Maintain a parallel HashMap for O(1) lookups by ID

Output:

