# **Analysis:**

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

The analysis of the time complexity for each operation in the EmployeeManagementSystem:

1. Add Employee (addEmployee):
   * Time Complexity: O(1)
   * Explanation: Adding an employee involves placing it at the next available index. This operation takes constant time as it only requires assigning a reference.
2. Search Employee (searchEmployee):
   * Time Complexity: O(n)
   * Explanation: In the worst case, the method may need to traverse all n employees to find the one with the specified employeeId. Thus, it has linear time complexity.
3. Traverse Employees (traverseEmployees):
   * Time Complexity: O(n)
   * Explanation: This operation involves visiting each employee in the array once to print their details, leading to linear time complexity.
4. Delete Employee (deleteEmployee):
   * Time Complexity: O(n)
   * Explanation: Similar to the search operation, the worst case requires finding the employee (O(n)) and then shifting subsequent employees to fill the gap (O(n)). Therefore, the total complexity remains linear.

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

Limitations of Arrays:

1. Fixed Size:
   * Arrays have a fixed size that must be defined at the time of creation. This can lead to inefficient memory usage if the array is too large or cause overflow if it is too small.
2. Inefficient Insertion/Deletion:
   * Inserting or deleting elements requires shifting other elements, resulting in O(n) time complexity, making arrays less efficient for dynamic datasets.
3. Homogeneous Data:
   * Arrays can typically store only one data type (e.g., all integers or all strings), limiting flexibility in handling heterogeneous data.
4. Memory Allocation:
   * Since arrays require contiguous memory allocation, they may lead to fragmentation in memory management, especially in long-running applications.
5. Complexity in Multi-Dimensional Arrays:
   * Managing multi-dimensional arrays can become complex and cumbersome, especially with variable-sized sub-arrays.

When to Use Arrays:

1. Static Datasets:
   * Use arrays when the size of the dataset is known beforehand and does not change frequently.
2. Fast Access:
   * When fast access to elements is crucial (O(1) time complexity), arrays are suitable due to their direct indexing capability.
3. Simple Structures:
   * Arrays are ideal for implementing simple data structures like stacks, queues, and matrices where performance and memory efficiency are prioritized.
4. Low Overhead:
   * Use arrays when low overhead is necessary, as they have minimal memory requirements compared to more complex data structures.
5. Performance-Critical Applications:
   * In performance-critical applications, where memory locality and cache efficiency are important, arrays are often preferred.