**1. Understand Array Representation**

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

**Memory Representation:** Arrays are stored in contiguous memory locations, meaning that each element is placed next to the previous one. This allows for efficient indexing through simple arithmetic operations. For example, if an array starts at a memory address called base\_address, the element at index i is located at base\_address + i \* element\_size.

**Advantages:**

1. **Constant-Time Access:** Accessing any element by its index is O(1), making arrays highly efficient for retrieval operations.
2. **Cache Friendliness:** The contiguous memory layout of arrays results in good cache locality, which enhances access speed.
3. **Predictable Memory Usage:** Arrays have a fixed memory usage that is determined at creation, which can simplify memory management in certain scenarios.

**4. Analysis**

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

**Time Complexity for Each Operation:**

1. **Add Employee:** O(1) - Adding an employee is a constant-time operation as long as there is available space in the array.
2. **Search Employee:** O(n) - Searching through the array requires a linear scan, making the time complexity proportional to the number of elements.
3. **Traverse Employees:** O(n) - Traversing the array to access or print each employee requires linear time.
4. **Delete Employee:** O(n) - Deleting an element involves shifting elements, resulting in linear time complexity.

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

**Limitations of Arrays:**

1. **Fixed Size:** Arrays have a predefined size, which means you must allocate sufficient space at creation. This can lead to wasted memory if the array is underutilized or require resizing if it becomes full.
2. **Inefficient Deletion:** Removing an element from an array involves shifting elements to fill the gap, which can be inefficient for large arrays.
3. **Poor Insertion Performance:** Inserting an element at any position other than the end requires shifting elements, which can be slow.

**When to Use Arrays:**

1. **Known Fixed Size:** Arrays are suitable when the number of elements is known in advance and remains relatively constant.
2. **Need for Fast Access:** Arrays are ideal for situations where quick access to elements by index is essential.
3. **Memory Contiguity:** Arrays are advantageous when memory efficiency and good cache performance are needed.