**Step 1: Understand Array Representation**

**Array Representation in Memory:**

* **Definition:** An array is a data structure that stores a fixed-size sequential collection of elements of the same type.
* **Memory Allocation:**
  + Arrays are stored in contiguous memory locations. Each element in an array is accessed using its index, with the base address and the size of each element used to calculate the address.
  + **Advantages:**
    - **Fast Access:** Direct access to elements using index, resulting in O(1) time complexity for retrieval.
    - **Simple Implementation:** Easy to use and understand, with a straightforward indexing mechanism.
    - **Low Overhead:** Minimal memory overhead compared to more complex data structures.

**Disadvantages:**

* **Fixed Size:** Once created, the size of the array cannot be changed. This can lead to wasted space or a need to resize (which is not straightforward).
* **Inefficient Insertions/Deletions:** Adding or removing elements involves shifting other elements, which can be time-consuming (O(n) in the worst case).

**Step 4: Analysis**

**Time Complexity Analysis:**

1. **Add Employee:**
   * **Best Case:** O(1) – When there is space in the array.
   * **Worst Case:** O(n) – When the array needs to be resized (due to copying elements to a new array).
2. **Search Employee by ID:**
   * **Best Case:** O(1) – When the employee is at the beginning of the array.
   * **Worst Case:** O(n) – When the employee is at the end of the array or not present.
3. **Traverse Employees:**
   * **Time Complexity:** O(n) – You have to visit each element in the array.
4. **Delete Employee by ID:**
   * **Best Case:** O(1) – When the employee is at the end of the array.
   * **Worst Case:** O(n) – When the employee is at the beginning of the array or when shifting elements.

**Limitations of Arrays:**

* **Fixed Size:** Once an array is created, its size cannot be changed without creating a new array.
* **Resizing Overhead:** Resizing involves copying all elements to a new array, which can be inefficient.
* **Inefficient Insertions/Deletions:** Adding or removing elements involves shifting elements, which can be time-consuming for large arrays.

**When to Use Arrays:**

* **Small to Medium Datasets:** When the size is known or does not change frequently.
* **Simple Use Cases:** For scenarios where fixed-size and simple access patterns are sufficient.

**Alternatives:**

* **Dynamic Arrays (e.g., ArrayList in Java):** Can grow in size dynamically and handle insertions and deletions more efficiently.
* **Linked Lists:** For scenarios with frequent insertions and deletions, as they allow for efficient element additions and removals.