**Step 1: Understanding Array Representation**

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

An **array** is a **fixed-size collection of elements** stored **contiguously in memory**, meaning one after the other.

* Every element in an array is stored at a specific **memory address**.
* The **index** helps access any element directly using:  
  address of element = base address + (index × size of data type)

For example:  
If arr is an int[] with values {10, 20, 30} and base address is 1000,  
then:

* arr[0] → address = 1000
* arr[1] → address = 1004 (assuming int = 4 bytes)
* arr[2] → address = 1008

### Advantages of Arrays:

* **Fast access** using indices (O(1))
* **Efficient memory** allocation for known-sized data
* **Simple and lightweight** data structure
* Good for **static lists** where size doesn't change often

**Step 4: Analysis**

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

| Operation | Description | Time Complexity |
| --- | --- | --- |
| **Add** | Insert at next available index | O(1) |
| **Search** | Linear search by ID | O(n) |
| **Traverse** | Print all employee records | O(n) |
| **Delete** | Find by ID, then shift remaining items | O(n) |

II. Discuss the limitations of arrays and when to use them.

### Limitations of Arrays

* **Fixed size**: One must define the size in advance (e.g., 100 employees). He/She cannot exceed it.
* **Inefficient deletions**: Deleting an element requires shifting others — slow for large lists.
* **Wasted memory**: If a person only store 5 employees in a 100-sized array, 95 spaces are unused.
* **Not dynamic**: Cannot grow or shrink as needed.

Use arrays when:

* One know the **exact number of elements** ahead of time.
* One need **fast random access** (e.g., access by index).
* One want **simple and fast insertions at the end**.
* Memory layout (contiguity) matters for performance-sensitive apps.