**EMPLOYEE MANAGEMENT SYSTEM**

**Understanding Array Representation**

**1. Memory Representation:**

* Contiguous Memory Allocation: Arrays are stored in contiguous blocks of memory, meaning all elements are stored one after another in adjacent memory locations.
* The base address of the array (i.e., the memory location of the first element) is used along with the element’s index to access any element directly using the formula:  
  address = base\_address + (index × size\_of\_each\_element)

**2. Advantages of Arrays:**

* **Direct Access (Random Access)**: Elements can be accessed in constant time using the index (O(1) time complexity).
* **Efficient Memory Usage**: Arrays do not have extra overhead like linked data structures (no pointers).
* **Easy Traversal**: Iterating over an array is straightforward with loops.
* **Cache-Friendly**: Due to contiguous allocation, arrays are more cache-efficient compared to other data structures.

**Time Complexity of Array Operations**

1. **Add (Insert at End)**
   * **Time Complexity:** O(1) *(*if space is available*)*
   * **Explanation:** Adding a new element at the next available index is a constant-time operation.
   * If the array is full, you need to create a new array and copy elements , this takes **O(n)** time.
2. **Search (By Value or ID)**
   * **Time Complexity:** O(n)
   * **Explanation:** You must iterate through the array to find a matching element in the worst case.
3. **Traverse**
   * **Time Complexity:** O(n)
   * **Explanation:** Visiting every element once requires linear time.
4. **Delete (By Value or ID)**
   * **Time Complexity:** O(n)
   * **Explanation:** You must first find the element, and then shift all subsequent elements one position left to fill the gap.

**Limitations of Arrays**

* **Fixed Size:** Once declared, the size of the array cannot be changed. Dynamic resizing requires creating a new array.
* **Costly Insertions and Deletions:** Inserting or deleting elements from the middle requires shifting elements, which is inefficient (O(n)).
* **Wasted Memory:** If the array is sparsely filled, it leads to unused memory allocation.
* **No Built-in Flexibility:** Unlike ArrayList or other dynamic structures, arrays lack methods for easy management (e.g., automatic resizing, searching).

**When to Use Arrays**

* When the number of elements is known in advance and unlikely to change.
* When you need **fast, indexed access** to elements.
* For **simple storage** where insertions/deletions are rare.
* When working in **low-level programming** (like embedded systems) where memory efficiency is crucial.