1. **Arrays in Memory**

**Representation**: Arrays are stored in contiguous memory locations. This allows for efficient access to elements using an index.

**Advantages**:

* **Fast Access**: O(1) time complexity for accessing elements by index.
* **Simple Structure**: Easy to use and understand.
* **Efficient Space Usage**: No overhead beyond the array size itself.

4a.) **Time Complexity**

1. **Add**:
   * **Time Complexity**: O(1) (amortized) - Adding an employee is constant time unless the array is full, in which case additional logic may be needed.
   * **Space Complexity**: O(n) where n is the capacity of the array.
2. **Search**:
   * **Time Complexity**: O(n) - Linear search is used to find an employee.
   * **Space Complexity**: O(1) - No extra space beyond the input array.
3. **Traverse**:
   * **Time Complexity**: O(n) - Traversing all employees requires linear time.
   * **Space Complexity**: O(1) - No extra space beyond the input array.
4. **Delete**:
   * **Time Complexity**: O(n) - Linear search is used to find the employee, and shifting elements takes linear time.
   * **Space Complexity**: O(1) - No extra space beyond the input array.

4b.) **Limitations of Arrays**

1. **Fixed Size**:
   * **Limitation**: Arrays have a fixed size, which can be a constraint if the number of employees exceeds the initial capacity.
   * **Solution**: Consider using dynamic data structures like ArrayList if the number of employees can vary significantly.
2. **Insertion and Deletion**:
   * **Limitation**: Insertion and deletion require shifting elements, which can be inefficient.
   * **Solution**: For better performance in insertion and deletion, consider using data structures like linked lists or balanced trees.
3. **Scalability**:
   * **Limitation**: Arrays may not be suitable for very large datasets due to the need to handle resizing and memory management.
   * **Solution**: Advanced data structures like HashMap or TreeMap can offer better performance for large-scale applications.