Array Representation in Memory:

- **Contiguous Memory Allocation**: Arrays are stored in contiguous memory locations. This means that all elements of the array are stored next to each other in memory, with no gaps.
- **Indexing**: Each element in an array can be accessed using its index, which represents the element's position in the array. The first element is at index 0, the second at index 1, and so on.
- **Fixed Size**: The size of an array is defined at the time of creation and cannot be changed. This makes arrays static in terms of size.

Advantages of Arrays:

- **Constant Time Access**: Due to contiguous memory allocation and indexing, accessing any element in an array has a constant time complexity, O(1). This makes arrays extremely efficient for lookups when the index is known.
- **Memory Efficiency**: Arrays do not require additional memory for storing pointers or other structures, unlike linked lists or other dynamic data structures.
- **Cache-Friendly**: Because arrays are stored in contiguous memory locations, they tend to be more cache-friendly, leading to potential performance benefits due to better cache utilization.

Analysis :

Time Complexity of Operations

- **Add**:
  - **Best Case**: O(1) (constant time, as we add to the end of the array if there is space)
  - **Worst Case**: O(n) (if we need to resize the array, though in this implementation, we assume the array has a fixed size and no resizing)
- **Search**:
  - **Best Case**: O(1) (if the desired employee is at the beginning of the array)
  - **Worst Case**: O(n) (if the employee is at the end or not present in the array)
- **Traverse**: O(n) (we need to visit each element in the array)
- **Delete**:
  - **Best Case**: O(1) (if the employee to be deleted is the last one)
  - **Worst Case**: O(n) (if the employee is at the beginning, and we need to shift all subsequent elements)

Limitations of Arrays

- **Fixed Size**: The size of the array is fixed upon creation. If the array is full, adding new elements requires creating a new, larger array and copying the existing elements, which is costly.
- **Inefficient Insertions and Deletions**: While accessing elements is fast, insertions and deletions (except at the end) require shifting elements, leading to O(n) time complexity.
- **Wasted Space or Overflow**: Allocating a larger array than needed wastes memory, while too small an array requires resizing operations, which are expensive.