1. **Arrays:**

Arrays in memory are represented using contiguous allocation, meaning they are stored in consecutive memory locations. This allows for direct access to elements via their indices.

**Memory Representation:**

**Contiguous Allocation:** Arrays are stored in contiguous memory locations, allowing direct access to elements via their indices.

**Indexing:** Elements are accessed using the formula: Address(A[i]) = Base Address + (i × Element Size)

**Advantages:**

- Direct Access: O(1) time complexity for element access.

- Cache-Friendly: Benefits from spatial locality, improving cache performance.

- Ease of Iteration: Elements stored consecutively simplify iteration.

- Predictable Memory Usage: Fixed size upon creation.

- Efficient Memory Use: No overhead for pointers or structures.

**Time Complexity:**

**Adding**

- At the End: O(1) if there's space.  
- At a Specific Index: O(n) because elements need to be shifted.

**Searching**

- Unsorted Array: O(n) for linear search.

- Sorted Array: O(log n) for binary search.

**Traversing**

Time Complexity: O(n) as each element is accessed once.

**Deleting**

- At a Specific Index: O(n) because elements need to be shifted.

- At the End: O(1).

**Limitations of Arrays:**

**Fixed Size**

Arrays have a fixed size defined at creation, making them inflexible for dynamic datasets.

**Inefficient insertion/deletion**

Inserting or deleting elements, especially not at the end, is inefficient (O(n)) due to shifting elements.

**Contiguous memory requirement**

Requires contiguous memory, which may be difficult to allocate for large arrays.

**When to use arrays**

**Predictable memory usage needed**

Arrays provide predictable and efficient memory allocation.

**Fast access and retrieval**

Ideal for scenarios requiring fast access to elements via indices (O(1) time complexity).

**When data size is fixed**

Suitable for datasets with a known and fixed size.