**Arrays in Memory**

**Representation:**

* Arrays are stored in contiguous memory blocks.
* Each element occupies the same amount of space.

**Advantages**

1. **Direct Access:**
   * **Time Complexity:** O(1) for accessing elements by index.
2. **Cache-Friendly:**
   * Sequential access improves CPU cache usage.
3. **Memory Efficiency:**
   * Less overhead compared to data structures like linked lists.
4. **Simple Implementation:**
   * Easy to declare, initialize, and iterate.
5. **Ease of Traversal:**
   * Simple and efficient iteration with loops.

**Example**

For an integer array:

int[] arr = {1, 2, 3, 4, 5};

* **Access:** arr[2] is at base\_address + (2 \* size\_of\_element).

**Analysis:**

### Time Complexity Analysis

Here is the time complexity analysis of each operation in my code:

#### **1. Traverse (traverseEmployee Method)**

* **Time Complexity:** O(n)
* **Explanation:** The method iterates through the employeeDetails array once, where n is the length of the array. Each element is checked and printed, making the time complexity linear with respect to the number of elements in the array.

#### **2. Add (addEmployee Method)**

* **Time Complexity:** O(1)
* **Explanation:** This method creates a new Employee object with constant time operations: reading input and returning the new employee. The complexity is constant, irrespective of the size of the input.

#### **3. Search (searchEmployee Method)**

* **Time Complexity:** O(log n)
* **Explanation:** The method performs a binary search on the employee array. Binary search has a logarithmic time complexity with respect to the number of elements, assuming the array is sorted. If the array is not sorted, this method will not work correctly without sorting.

#### **4. Delete (deleteEmployee Method)**

* **Time Complexity:** O(n)
* **Explanation:** The method iterates through the employees array to find and delete the employee with the specified ID. The contains method in the code is incorrect because it is not applicable to integers directly; the correct check should be a simple comparison. Even with a correct implementation, the time complexity remains linear, as the method needs to check each element in the array.

**Limitations of Arrays**

1. **Fixed Size:**
   * Arrays have a fixed size determined at initialization. Resizing requires creating a new array and copying elements, which is costly.
2. **Insertion and Deletion:**
   * Adding or removing elements from the middle or start is inefficient because it involves shifting elements.
3. **Memory Usage:**
   * Arrays can waste memory if allocated size is larger than the number of elements used.
4. **Flexibility:**
   * Arrays are less flexible compared to dynamic data structures like linked lists, which can grow and shrink dynamically.

**When to Use Arrays:**

1. **When Size is Known and Fixed:**
   * Ideal for situations where the number of elements is known in advance and does not change often.
2. **For Fast Access:**
   * Suitable for applications needing constant-time access to elements (e.g., lookups by index).
3. **When Memory Efficiency is Critical:**
   * Efficient when low memory overhead is required, as there is no additional space for pointers.
4. **For Simple Traversal:**
   * Useful when simple, sequential access or traversal is needed, and operations like sorting or searching are not frequent.