1. Understand Array Representation

(i) Explain how arrays are represented in memory and their advantages.

Arrays in Memory:

Arrays are represented in memory as a contiguous block of memory locations. The key characteristics of arrays in memory include:

- Contiguous Memory Allocation: All elements of an array are stored in consecutive memory locations. This allows the array to have a fixed starting address, and each element can be accessed directly using its index.

- Index Calculation: The address of an element is computed using a formula based on its index. For an array of elements `arr` where each element is of size `S`, the address of the element at index `i` is `base_address + i * S`. This formula provides O(1) time complexity for accessing elements, known as direct or random access.

Advantages of Arrays:

- Fast Access: Arrays provide constant time O(1) access to elements due to direct indexing.

- Memory Efficiency: Arrays use memory efficiently when the size is known in advance, as they avoid the overhead associated with dynamic data structures.

- Simplicity: Arrays are straightforward to implement and use, making them ideal for simple data storage where the size is fixed or changes infrequently.

2.Analysis

(i) Analyze the time complexity of each operation (add, search, traverse, delete).

Operations on Arrays:

- addEmployee(): O(1)

  - Reason: Adding an employee to the end of an array involves assigning a value to the next available index. If there is space in the array, this operation is constant time. However, if the array is full and resizing is required, the time complexity would include the cost of copying elements to a new array.

- searchEmployee(): O(n)

  - Reason: Searching for an employee requires scanning through each element of the array until the target is found. In the worst case, the employee might be at the last position or not present at all, leading to linear time complexity.

-traverseEmployees(): O(n)

  - Reason: Traversing the array involves visiting each element once to perform an action, such as printing details. This results in linear time complexity.

- deleteEmployee(): O(n)

  - Reason: Deleting an employee requires finding the element (O(n)) and then shifting all subsequent elements to fill the gap, which also takes linear time.

(ii) Discuss the limitations of arrays and when to use them.

Limitations of Arrays:

- Fixed Size: Once an array is created, its size cannot be changed. This can lead to wasted memory if the allocated size is larger than needed or insufficient space if the array becomes full.

- No Dynamic Resizing: Arrays do not support dynamic resizing. To accommodate more elements, a new array with a larger size must be created, and existing elements must be copied over. This resizing operation can be costly in terms of performance.

- Inefficient Insertions and Deletions: Inserting or deleting elements in an array, especially at positions other than the end, requires shifting elements. This can be inefficient, particularly for large arrays.

When to Use Arrays:

- Fixed Size Data: Arrays are ideal when the number of elements is known in advance and does not change frequently.

- Simple Data Structures:Arrays are suitable for simple data structures where dynamic resizing and complex operations are not required.

- Performance-Critical Applications: Arrays are useful when fast access to elements and minimal overhead are critical, such as in systems with real-time requirements.

In summary, while arrays offer efficient access and straightforward implementation, their limitations in flexibility and dynamic resizing make them less suitable for scenarios requiring frequent modifications or where the size of the dataset varies significantly.