Understand Linked Lists

Types of Linked Lists:

- **Singly Linked List**:
  - **Structure**: Each node contains a data element and a reference (or pointer) to the next node in the sequence.
  - **Operations**: Can efficiently perform insertions and deletions, especially at the beginning or end of the list. However, traversing backward is not possible.
  - **Memory Usage**: Each node requires additional memory for storing the reference to the next node.
- **Doubly Linked List**:
  - **Structure**: Each node contains a data element, a reference to the next node, and a reference to the previous node.
  - **Operations**: Supports efficient insertions and deletions from both ends and any position in the list. Traversal can be done in both directions (forward and backward).
  - **Memory Usage**: Each node requires more memory than a singly linked list because it stores two references (next and previous).

Analysis :

Time Complexity of Operations

- **Add**:
  - **Best Case**: O(1) (if adding to the front)
  - **Worst Case**: O(n) (if adding to the end requires traversal)
- **Search**:
  - **Best Case**: O(1) (if the task is at the head)
  - **Worst Case**: O(n) (if the task is at the end or not present)
- **Traverse**: O(n) (requires visiting each node in the list)
- **Delete**:
  - **Best Case**: O(1) (if deleting the head node)
  - **Worst Case**: O(n) (if the task is at the end or not present)

Advantages of Linked Lists Over Arrays

- **Dynamic Size**: Linked lists can grow and shrink in size dynamically, unlike arrays which have a fixed size once created.
- **Efficient Insertions/Deletions**: Inserting or deleting elements, especially at the beginning or middle, is more efficient compared to arrays where shifting elements is necessary.
- **No Wasted Space**: Linked lists do not require pre allocating space, making them more memory efficient for unpredictable or variable-sized datasets.

Limitations of Linked Lists

- **Memory Overhead**: Each node requires extra memory for storing the reference(s) in addition to the data.
- **Sequential Access**: Accessing elements requires traversal from the head node, which is less efficient compared to direct indexing in arrays.