1. **Singly Linked List**

A singly linked list is a type of data structure that consists of a sequence of elements, each of which points to the next element in the sequence. The key characteristics are:

**Node Structure**: Each node contains two parts:

**Data**: The value or information that the node holds.

**Next Pointer**: A reference to the next node in the list.

**Head Node**: The first node in the list is called the head. The head is the starting point for any traversal or operations on the list.

**Traversal**: You can only traverse the list in one direction—from the head to the end of the list. You cannot move backward from any node.

**Insertion and Deletion**: Operations like insertion and deletion are efficient, especially at the beginning of the list (inserting at the head) or at a known position.

**Doubly Linked List**

A doubly linked list is an extension of a singly linked list, with an additional pointer in each node that points to the previous node. This allows for traversal in both directions.

**Node Structure**: Each node contains three parts:

* 1. **Data**: The value or information that the node holds.
  2. **Next Pointer**: A reference to the next node in the list.
  3. **Previous Pointer**: A reference to the previous node in the list.

**Head and Tail Nodes**: The first node is called the head, and the last node is called the tail. The head’s previous pointer and the tail’s next pointer are null.

**Traversal**: You can traverse the list in both directions—from the head to the tail and from the tail to the head.

**Insertion and Deletion**: Operations are more flexible because you can easily access and manipulate nodes from both ends. However, it requires more memory due to the extra pointer for the previous node.

**Summary of Time Complexities:**

**Adding a Task**:

**To the End**: O(n), where ‘n’ is the number of tasks in the list. You need to traverse the list to find the last node.

**To the Beginning**: O(1), if you insert the new task at the head.

**Searching for a Task**:

**By ID**: O(n), where ‘n’ is the number of tasks in the list. You may need to traverse the entire list to find the task.

**Traversing the Tasks**:

**Traverse All**: O(n), where ‘n’ is the number of tasks in the list. Each node is visited once.

**Deleting a Task**:

**By ID**: O(n), where ‘n’ is the number of tasks in the list. You need to find the node to be deleted, which may require traversing the list, and then adjust pointers.

**Advantages of Linked Lists over Arrays for dynamic data:**

**Advantages of Linked Lists over Arrays for Dynamic Data**

1. **Dynamic Size:**
   * Linked Lists: Can easily grow and shrink in size by adding or removing nodes. There is no need to define an initial size, making them more flexible for dynamic data.
   * Arrays: Have a fixed size once initialized. Changing the size of an array requires creating a new array and copying the data, which can be inefficient.
2. **Efficient Insertions and Deletions:**
   * Linked Lists: Inserting or deleting an element in a linked list can be done in constant time O(1) if the pointer to the node is known. This is particularly advantageous for operations at the beginning or middle of the list.
   * Arrays: Inserting or deleting an element requires shifting all subsequent elements, which takes linear time O(n).
3. **Memory Utilization:**
   * Linked Lists: Allocate memory for each element dynamically. This means that memory usage is more efficient, as it only uses what is needed for the elements stored.
   * Arrays: Allocate a fixed amount of memory upfront. If the array is not fully utilized, there can be wasted memory.
4. **No Wasted Space:**
   * Linked Lists: Only use memory for the actual elements plus pointers. There is no need to allocate extra space.
   * Arrays: Often have unused allocated space if the array is not fully occupied or if it needs to be resized frequently.
5. **Better for Unknown or Changing Sizes:**
   * Linked Lists: Ideal when the number of elements is unknown in advance or is expected to change frequently. They handle size changes gracefully.
   * Arrays: Require a good estimate of the number of elements to allocate memory efficiently. Poor estimates can lead to frequent resizing operations.