**Types of Linked Lists**

**1. Singly Linked List**

- **Structure**: Each node consists of two parts:

- Data: The value stored in the node.

- Next: A pointer/reference to the next node in the sequence. The last node points to null, indicating the end of the list.

- **Advantages**:

- Dynamic Size: Can grow and shrink as needed without pre-allocating memory.

- Efficient Insertions/Deletions: Adding or removing nodes at the beginning or middle is efficient if you have a reference to the previous node.

- **Disadvantages**:

- Unidirectional Traversal: You can only traverse the list from the head to the end, making reverse traversal difficult.

- Memory Overhead: Requires extra memory for storing the next pointer.

**2. Doubly Linked List**

- **Structure**: Each node consists of three parts:

- Data: The value stored in the node.

- Next: A pointer/reference to the next node.

- Prev: A pointer/reference to the previous node. The first node’s prev points to null, and the last node’s next points to null.

- **Advantages**:

- Bidirectional Traversal: Allows traversal in both directions (forward and backward), making certain operations easier.

- Flexible Insertions/Deletions: Insertion and deletion are more straightforward, especially when dealing with nodes near the end of the list.

- **Disadvantages**:

- Increased Memory Usage: Requires additional memory for the prev pointer.

- Complexity: More complex to implement and manage due to the additional prev pointers.

**Time Complexity of Linked List Operations**

**1. Singly Linked List**

- **Insertion**:

- At Beginning: O(1) – Directly add the new node at the head.

- At End: O(n) – Requires traversing the list to find the end.

- At Middle: O(n) – Requires finding the insertion point by traversing from the head.

- **Deletion**:

- At Beginning: O(1) – Directly remove the head node.

- At End: O(n) – Requires traversing the list to find the end and update the second-to-last node’s next pointer.

- At Middle: O(n) – Requires finding the deletion point by traversing from the head.

- **Search**: O(n) – Must traverse the list from the head to find the target element.

- **Traversal**: O(n) – Each node must be visited once.

**2. Doubly Linked List**

- **Insertion**:

- At Beginning: O(1) – Directly add the new node at the head.

- At End: O(1) – Directly add the new node if you have a reference to the tail.

- At Middle: O(n) – Requires finding the insertion point, which involves traversing the list.

- **Deletion**:

- At Beginning: O(1) – Directly remove the head node.

- At End: O(1) – Directly remove the tail node if you have a reference to it.

- At Middle: O(1) – Remove the node by updating the next and prev pointers of adjacent nodes, if you have a reference to the node.

- **Search**: O(n) – Must traverse the list from either end to find the target element.

- **Traversal**: O(n) – Can be done in both directions (forward and backward).

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

**1. Dynamic Size:**

- Linked Lists: Can easily grow or shrink in size by adding or removing nodes without the need for resizing or reallocating memory.

- Arrays: Fixed size once created, requiring resizing and copying if the number of elements changes.

**2. Efficient Insertions/Deletions:**

- Linked Lists: Insertion and deletion operations are efficient (O(1)) at the beginning of the list and can be efficient in the middle if you have a reference to the node.

- Arrays: Insertion and deletion operations can be costly (O(n)) as elements may need to be shifted.