**Linked List:**

Linked List is a linear data structure, in which elements are not stored at a contiguous location, rather they are linked using pointers.

In a linked list, nodes are linked together using pointers or references. Each node contains two parts: the data and a reference to the next node in the sequence. This sequence of nodes creates a chain-like structure.

1. \*\*Node\*\*: Each element in a linked list is represented by a node. A node typically contains two fields: a data field to store the actual value or payload, and a reference field (often called a pointer or link) that points to the next node in the sequence.

2. \*\*Head\*\*: The head of the linked list is the first node in the sequence. It serves as the entry point for accessing the elements in the list.

3. \*\*Tail\*\*: The tail of the linked list is the last node in the sequence. Its next reference typically points to null, indicating the end of the list.

**Operations:**

* Traversal - To access each element of the linked list.
* Insertion - To add/insert a new node to the list.
* Deletion - To remove ane existing node from the list.
* Search - To find a node in the list.
* Sort - To sort the nodes.

**TYPES:**

- Singly Linked List: In a singly linked list, each node has a reference to the next node in the sequence, forming a unidirectional chain. Traversal of a singly linked list is only possible in one direction, starting from the head and moving towards the tail.

// A Single linked list node

**class** Node {

**public**:

**int** data;

    Node\* next;

};

- Doubly Linked List: In a doubly linked list, each node has references to both the next and the previous nodes in the sequence, forming a bidirectional chain. This allows for traversal in both forward and backward directions.

/\* Node of a doubly linked list \*/

**class** Node {

**public**:

**int** data;

    Node\* next; // Pointer to next node in DLL

    Node\* prev; // Pointer to previous node in DLL

};

- Circular Linked List: In a circular linked list, the last node's next reference points back to the head of the list, forming a loop. This creates a circular structure where traversal can start from any node in the list.

Circular linked lists are similar to single Linked Lists with the exception of connecting the last node to the first node.

**struct** Node {

**int** data;

**struct** Node \*next;

};

**Applications:**

1. Dynamic Memory Allocation: Efficient memory management in languages like C.

2. Stacks and Queues: Implementing fundamental data structures.

3. Sparse Matrix Representation: Efficiently storing matrices with mostly zero values.

4. Graphs: Representing graphs in adjacency list representation.

5. Memory Management: Maintaining free memory blocks in operating systems.

6. Text Editors: Implementing undo functionality in text editors.

7. Symbol Tables: Storing symbols and their associated information in compilers.

8. Polynomial Manipulation: Representing and manipulating polynomials efficiently.

**Advantages:**

* **Dynamic Data structure:**The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.
* **Ease of Insertion/Deletion:**The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
* **Efficient Memory Utilization:**As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
* **Implementation:**Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

**Disadvantages:**

* **Random Access:**Unlike arrays, linked lists do not allow direct access to elements by index. Traversal is required to reach a specific node.
* **Extra Memory:** Linked lists require additional memory for storing the pointers, compared to arrays.