**1.Understand Linked Lists:**

**Explain the different types of linked lists (Singly Linked List, Doubly Linked List):**

**Linked List:**

A **linked list** is a **linear data structure** where elements (nodes) are stored in **non-contiguous** memory locations. Each node contains two (or more) parts:

* **Data** – stores the actual value.
* **Pointer** – stores the address of the next (or previous) node.

Unlike arrays, linked lists do not have a fixed size. This makes them highly suitable for dynamic memory allocation.

**Types of Linked Lists**

**1. Singly Linked List (SLL)**

A **singly linked list** is a linear data structure in which each element (called a **node**) contains:

* **Data** (value stored in the node)
* **Pointer to the next node**

Each node points only to the **next** node in the sequence, and the **last node points to null**, indicating the end of the list.

**Structure:**

[Data | Next] -> [Data | Next] -> [Data | null]

**Characteristics:**

* **Traversal** is possible only in the **forward** direction.
* Operations like **insertion and deletion** are efficient, especially at the beginning of the list.
* Requires **less memory** since only one pointer is stored per node.

**Common Operations:**

* Insert at head
* Insert at end
* Search
* Delete

**Applications:**

* Stack (Last In First Out)
* Adjacency lists in graph representations
* Simple memory-efficient data structure

**Doubly Linked List (DLL)**

A **doubly linked list** is a type of linked list where each node contains:

* **Data**
* A **pointer to the next node**
* A **pointer to the previous node**

This allows traversal in **both forward and backward** directions.

**Structure:**

null <- [Prev | Data | Next] <-> [Prev | Data | Next] <-> [Prev | Data | Next] -> null

**Characteristics:**

* More **flexible** than singly linked lists.
* Enables **bidirectional traversal**.
* Each node uses **more memory** due to the additional pointer.

**Common Operations:**

* Insert at head/tail
* Delete a node
* Search
* Traverse forward and backward

**Applications:**

* Browser history navigation (forward/back)
* Music/media playlists
* Undo/Redo functionality in editors
* MRU/LRU cache implementations

**4.Analysis:**

**Analyze the time complexity of each operation:**

| **Operation** | **Singly Linked List (SLL)** | **Doubly Linked List (DLL)** | **Explanation** |
| --- | --- | --- | --- |
| **Insertion at Head** | O(1) | O(1) | Directly point new node to current head. No traversal needed. |
| **Insertion at Tail** | O(n) | O(n) (or O(1)\*) | Requires traversal to end in SLL. DLL can be O(1) if tail pointer is maintained. |
| **Insertion in Middle** | O(n) | O(n) | Need to traverse to specific position to insert. |
| **Deletion at Head** | O(1) | O(1) | Just move the head pointer to the next node. |
| **Deletion at Tail** | O(n) | O(n) (or O(1)\*) | In SLL, you must traverse. DLL allows direct access if tail and prev pointers are available. |
| **Deletion by Value** | O(n) | O(n) | Must search through list to find node. |
| **Search** | O(n) | O(n) | May need to look through every node until the match is found. |
| **Traversal** | O(n) | O(n) | Visit each node one by one. DLL can traverse backward as well. |

**Discuss the advantages of linked lists over arrays for dynamic data:**

**Dynamic Size Allocation**

* **Linked List**: Size can grow or shrink dynamically at runtime without memory reallocation.
* **Array**: Requires a fixed size at declaration or resizing using costly operations (like copying to a new array).

In dynamic scenarios (e.g., adding/removing unknown number of tasks), linked lists are more memory-efficient and flexible.

**Efficient Insertion and Deletion**

* **Linked List**: Insertion and deletion can be done in **O(1)** time at the head (and tail in DLL with tail pointer).
* **Array**: Insertion and deletion require **shifting elements**, which takes **O(n)** time.

Example: Removing the first task in a task manager is faster in a linked list.

**No Wasted Memory (No Pre-allocation Needed)**

* **Linked List**: Allocates memory **only when needed**.
* **Array**: You may allocate more memory than needed “just in case,” which wastes space.

In memory-limited systems, this is a big advantage

**No Need for Resizing**

* **Array**: Once full, arrays must be resized — which involves **creating a new array and copying all data**.
* **Linked List**: Just add a new node; no reallocation required.

In high-frequency insert/delete operations, linked lists scale better.

**Faster Insertions in Middle**

* **Linked List**: Inserting between nodes is efficient if you have the pointer (just adjust next and prev).
* **Array**: Inserting in the middle requires shifting all elements after the insert point.