Linked List – Deletion in Linked List

1. Introduction

Deletion in a Linked List is the operation of **removing a node** from the list.

Unlike arrays, deletion in a linked list **does not require shifting elements**.

Instead, deletion is performed by **changing the links (pointers)** between nodes.

---

2. Why Deletion is Efficient in Linked List?

In arrays:

- Elements must be shifted after deletion
- Time complexity increases

In linked lists:

- Only pointers are updated
- No shifting of data
- More memory-efficient

---

3. Types of Deletion in Linked List

Deletion in a singly linked list can be performed in three main ways:

1. **Deletion at the Beginning**
2. **Deletion at the End**
3. **Deletion of a Specific Node (Middle)**

---

4. Deletion at the Beginning

What Happens?

The **first node (head)** of the linked list is removed.

Logic (Plain English)

1. Check if the list is empty
2. Store the current head node in a temporary variable
3. Move the head pointer to the next node
4. Delete the old head node

Example

Before:

```
Head → 10 → 20 → 30 → NULL
```

After deletion:

```
Head → 20 → 30 → NULL
```

---

5. Deletion at the End

What Happens?

The **last node** of the linked list is removed.

Logic (Plain English)

1. Check if the list is empty
2. Traverse the list to the second-last node
3. Set the second-last node’s next to NULL
4. Delete the last node

Example

Before:

```
Head → 10 → 20 → 30 → NULL
```

After deletion:

```
Head → 10 → 20 → NULL
```

---

6. Deletion of a Specific Node (Middle)

What Happens?

A node with a **specific value or position** is removed.

Logic (Plain English)

1. Check if the list is empty
2. Traverse the list to find the node before the target node
3. Change the next pointer to skip the target node
4. Delete the target node

Example

Delete node 20:

```
Head → 10 → 20 → 30 → NULL
```

After deletion:

```
Head → 10 → 30 → NULL
```

---

7. Pointer Adjustment Visualization

```
Before:
10 → 20 → 30
```

```
After:
10 → 30
```

Only **links change**, not data movement.

---

8. Time Complexity of Deletion

Type of Deletion	Time Complexity
At Beginning	O(1)
At End	O(n)
At Specific Position	O(n)

---

9. Edge Cases

- Deleting from an empty list
- Deleting the only node
- Deleting the head node
- Deleting a non-existing element

---

10. Advantages of Linked List Deletion

- No shifting of elements
- Efficient memory usage
- Faster than array deletion
- Dynamic structure

---

11. Limitations

- Traversal required for middle/end deletion
- No backward traversal in singly linked list
- Pointer handling must be careful

---

12. Real-World Applications

- Removing tasks from task lists
- Deleting songs from playlists
- Memory management
- Undo/Redo operations
- Managing dynamic data collections

---

13. Summary

- Deletion removes a node from the linked list
- Can be done at beginning, end, or middle
- Achieved by updating pointers
- Efficient compared to arrays
- Time complexity depends on position
