Linked List – Reverse Linked List

1. Introduction

Reversing a Linked List means changing the direction of links between nodes so that the list is reversed.

After reversal, the **last node becomes the first node**, and the **head pointer is updated** accordingly.

This is one of the **most important and frequently asked** linked list problems in exams and interviews.

---

2. What Does Reversing a Linked List Mean?

Original list:

```
Head → 10 → 20 → 30 → 40 → NULL
```

Reversed list:

```
Head → 40 → 30 → 20 → 10 → NULL
```

Only **links are reversed**, not the data.

---

3. Why Reverse a Linked List?

Reversing a linked list is useful for:

- Understanding pointer manipulation
- Solving complex linked list problems
- Implementing algorithms like palindrome checking
- Improving logical thinking with pointers

---

4. Key Idea Behind Reversing a Linked List

To reverse a linked list:

- Traverse the list once
- Change the next pointer of each node
- Make each node point to its **previous node**
- Update the head at the end

---

5. Variables Used (Conceptual)

To reverse a singly linked list, we use:

- **prev** → points to previous node
- **current** → points to current node
- **next** → stores next node temporarily

---

6. Logic for Reversing Linked List (Plain English)

1. Initialize prev as NULL
2. Set current to the head node
3. While current is not NULL:
    - Store current.next in next
    - Change current.next to prev
    - Move prev to current
    - Move current to next
4. After loop ends, set head = prev

---

7. Step-by-Step Example

Initial:

```
prev = NULL
```

```
current = 10
```

Iteration 1:

```
10 → NULL
prev = 10
current = 20
```

Iteration 2:

```
20 → 10 → NULL
prev = 20
current = 30
```

Iteration 3:

```
30 → 20 → 10 → NULL
prev = 30
current = 40
```

Iteration 4:

```
40 → 30 → 20 → 10 → NULL
```

Update head → 40

---

8. Visualization of Pointer Changes

```
Before:
10 → 20 → 30 → 40 → NULL

After:
NULL ← 10 ← 20 ← 30 ← 40
Head = 40
```

---

9. Time and Space Complexity

Aspect	Complexity
Time Complexity	O(n)
Space Complexity	O(1)

- Reversal is done **in-place**
- No extra memory is used

---

10. Edge Cases

- Empty linked list → remains empty
- Single node → no change
- Two nodes → links swapped
- Very large list → still efficient

---

11. Advantages of Iterative Reversal

- Fast and efficient
- Uses constant extra space
- Easy to understand once logic is clear
- Preferred in exams and interviews

---

12. Limitations

- Pointer handling is error-prone
- Difficult for beginners initially
- Requires careful step-by-step thinking

---

13. Applications of Reversing Linked List

- Palindrome checking

- Undo operations
- Data processing
- Algorithmic problem solving
- Interview questions

---

14. Summary

- Reversal changes the direction of links
- Uses three pointers: prev, current, next
- Head is updated at the end
- Time complexity is O(n)
- Space complexity is O(1)
