🔗 Linked List Interview Quick-Check Pattern

Master pointer-based thinking with this cheat sheet for solving linked list problems under pressure.

---

1. 🧠 Understand the Nature of the Problem

Ask yourself:

- ❓ Do I need to **traverse**, **reverse**, or **detect a cycle**?
- ❓ Am I deleting or inserting **nodes** (not just values)?
- ❓ Is it a **singly** or **doubly** linked list?
- ❓ Do I need **constant space** (O(1))?
- ❓ Do I need to return the **head** or **middle**?

> 💡 Think in **nodes** and **pointers**, not indexes.

---

2. 🔍 Most Common Linked List Patterns

Problem Type	Technique/Pattern
Traverse Entire List	While loop + `current = current.next`
Find Middle Node	Slow and Fast Pointers
Detect Cycle	Floyd’s Cycle Detection (slow/fast)
Reverse a List	Iterative pointer swap
Merge Two Lists	Dummy head + pointer
Delete Node(s)	Prev tracking + pointer shift
Add Numbers (as list)	Simulate carry using recursion or loop

Palindrome List	Find middle + reverse second half

---

3. 🔩 Core Templates

✅ Traverse List

```
function traverse(head) {
  let curr = head;
  while (curr) {
    console.log(curr.val);
    curr = curr.next;
  }
}
```

✅ Reverse List

```
function reverseList(head) {
  let prev = null, curr = head;
  while (curr) {
    let next = curr.next;
    curr.next = prev;
    prev = curr;
    curr = next;
  }
  return prev;
}
```

✅ Find Middle Node

```
function findMiddle(head) {
  let slow = head, fast = head;
  while (fast && fast.next) {
    slow = slow.next;
    fast = fast.next.next;
  }
  return slow;
}
```

✅ Detect Cycle

```
function hasCycle(head) {
  let slow = head, fast = head;
  while (fast && fast.next) {
    slow = slow.next;
    fast = fast.next.next;
    if (slow === fast) return true;
  }
  return false;
}
```

✅ Merge Two Sorted Lists

```
function mergeTwoLists(l1, l2) {
  let dummy = new ListNode(-1), curr = dummy;
  while (l1 && l2) {
    if (l1.val < l2.val) {
      curr.next = l1;
      l1 = l1.next;
    } else {
      curr.next = l2;
      l2 = l2.next;
    }
    curr = curr.next;
  }
  curr.next = l1 || l2;
  return dummy.next;
}
```

✅ Remove N-th Node from End

```
function removeNthFromEnd(head, n) {
  let dummy = new ListNode(0, head);
  let fast = dummy, slow = dummy;

  while (n--) fast = fast.next;
  while (fast.next) {
    fast = fast.next;
    slow = slow.next;
  }

  slow.next = slow.next.next;
  return dummy.next;
}
```

---

4. 🧱 Edge Cases to Always Think About

- Empty list (`null`)
- Single-node list
- Removing the **head** node
- Removing the **last** node
- Fast pointer reaches end (`fast.next.next`)
- `next = null` edge (during reversal)
- List with a **cycle** — infinite loop risk!
- Recursive depth (e.g. very long list with recursion)

> 🧠 Always draw a 3-node visual to catch pointer errors.

---

5. 🧠 Mental Model for Linked List Problems

Think in "Pointer Movement", not index math.

Question Type	Pattern Used
"Loop through the list"	While loop, `curr = curr.next`
"Modify list in-place"	Prev/curr/next manipulation
"Go N steps ahead"	Fast pointer
"From end of list"	Fast & Slow (N steps gap)
"Undo/Backtrack"	Recursion, stack (for doubly list)
"Is it a palindrome?"	Reverse 2nd half, compare

🔁 Problem Solving Loop

1. 🔍 What's the input? (head, node, N-th position?)
2. 🧷 Will I use a dummy node to simplify logic?
3. 🧪 Do I need a fast/slow pointer combo?
4. 🧼 Can I reverse, copy, or detach parts of the list?
5. 🚨 Did I forget to handle `head`, `null`, or tail edge?

✅ Final Interview Checklist

- Did I use a dummy node where appropriate?
- Are my `next` pointers being reassigned correctly?
- Is `head` affected — do I return `head` or a new head?
- Any infinite loop risk? (cycles, missed end condition)
- Have I handled edge cases like 0, 1, or N nodes?