Today's topic is **Linked Lists**. Before we jump into the coding side, let's first understand why we even need them, what they look like, and what kind of operations we can perform.

Basic knowledge

Why Do We Need Linked Lists?

So far, everything we've learned — whether it's a **hash table**, a **stack**, or a **queue** — is actually built on top of one main structure: the **array**.

Arrays let us access any element in **O(1)** time using an index, and they're stored **contiguously in memory**, which makes them really efficient for caching.

But arrays also come with two big limitations:

- **Fixed length** — once created, you can't resize an array easily.
- **Inefficient insertions and deletions** — if you insert or remove something in the middle, all the elements after that have to move, which takes **O(n)** time.

So, computer scientists came up with a more flexible structure — the **Linked List**.

The main characteristics of linked lists are:

- The nodes don't need to be stored next to each other in memory.
- You can insert or delete nodes in O(1) time just by adjusting pointers.

So, when our data changes frequently or we don't know how big dataset we will get, a linked list is often a better choice.

Basic Structure

A linked list is basically a sequence of **nodes** connected by **pointers**. The first node is called the **head**, and the last node points to `None`, which marks the end of the list.

Each node has two parts:

1. **Value (or data)** — the actual data you want to store.
2. **Pointer (or next)** — a reference to the next node.

Here's what a simple node looks like in Python:

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
```

So a linked list is a collection of these nodes chained together by their `next` pointers. Unlike arrays that rely on **indexes**, linked lists rely entirely on **references** to connect the data.

Types

Depending on the structure and pointer direction, linked lists come in three main types:

Singly Linked List

Singly Linked List — each node has one pointer: `next`

This is the most common form you'll see in interviews and LeetCode problems.

- You can only traverse it in one direction, from head to tail.
- To delete a node, you usually need to keep track of the previous node.

```
# Each node contains only one pointer next, which points to the next node.
head
 ↓
[1] → [2] → [3] → [4] → None
```

Doubly Linked List

Doubly Linked List — each node has two pointers: `prev` and `next`.

When we need to move both forward and backward, a doubly linked list is much more convenient — think about LRU caches or browser history.

- Each node has both `prev` and `next` pointers.
- Deleting a node is easier — no need to store the previous one separately. This structure costs a bit more memory, but it gives you extra flexibility.

```
# Each node contains two pointers: prev (predecessor) and next (successor).
None ← [1] ⇄ [2] ⇄ [3] ⇄ [4] → None
```

Circular Linked List

Circular Linked List — the last node points back to the head

- The last node doesn't point to `None`; instead, it loops back to the head — forming a ring.

```
# The next pointer of the tail node points to the head node, forming a ring.
      cur
       ↓
[1] → [2] → [3] → [4]
 ↑                  |
 └──────────────────┘
```

This is great for problems like implementing a **circular queue**. Because there's no real "end," traversing requires extra care — you'll often rely on stopping conditions or counters instead.

Common Operations

Most linked list operations come down to **rearranging pointers**. Let's look at the most common ones and their time complexities:

Operation	Description	Time Complexity
Traverse	Visit each node from `head`	O(n)
Search	Find a node with a specific value	O(n)
Insert	Insert a new node after a given one	O(1) if `prev` known
Delete	Remove a specific node	O(1) if `prev` known
Reverse	Reverse the entire list	O(n)

Traversal

This is the foundation for everything. We start from the head and move node by node until we hit `None`.

```
cur = head
while cur:
    cur = cur.next
```

Insertion

To insert a node, you just need to adjust two pointers — no shifting elements.

```
# Insert new_node after cur
new_node.next = cur.next
cur.next = new_node
```

Deletion

Deletion is skip over the node we want to remove.

```
# Delete the node after cur
cur.next = cur.next.next
```

Reversing

This is the classic interview problem. The idea is to flip the direction of all pointers so that every node points to its previous one.

The key is that **the order of pointer updates matters**. You must save `next_node` before breaking the link, otherwise you'll lose the rest of the list.

```
prev = None
cur = head
while cur:
    next_node = cur.next
    cur.next = prev
    prev = cur
    cur = next_node
# prev becomes the new head
```

Now, let's do some summary. We started with **why** we need linked lists, understood their **structure and types**,and practiced some **core operations** like traversal, insertion, deletion, and reversal.

Now that we've covered the basics, let's move on to some techniques — how to actually apply these operations to solve LeetCode problems efficiently.

Leetcode techniques

When we are solving linked list problems on LeetCode, we notice that most of them fall into a few common technique patterns.

Two Pointers

Let's start with the most powerful idea — **the two-pointer technique**.

The core idea is simple: "Use one traversal to achieve two goals."

- By controlling two pointers that move at different **speeds** or maintain a certain **distance**, we can capture key structural information in a single pass through the list.

The most famous version is the **fast and slow pointer. Here's how it works:

1. You create two pointers, `fast` and `slow`.
2. `fast` moves two steps at a time, while `slow` moves one step.
3. When `fast` reaches the end, `slow` will be halfway — that's the midpoint.

Because of this speed difference, you can solve all kinds of problems like:

- Finding the middle node
- Detecting a cycle
- Finding the start of the cycle
- Removing the N-th node from the end

Here's the typical pattern:

```
fast, slow = head, head
while fast and fast.next:
    fast = fast.next.next
    slow = slow.next
# slow now points to the middle node
```

Dummy Head

Next, let's talk about Dummy Head.

We can think of the dummy head node as a "fake head" that doesn't store a real value. It's just there to make your logic cleaner and avoid edge cases.

Because whenever you have to insert or delete nodes, handling the head node separately can get messy. With a dummy node, every node is treated the same way, so our code becomes much simpler.

```
Dummy → [1] → [2] → [3] → None
```

Here's an example — removing all nodes with a certain value:

```
dummy = ListNode(0)
dummy.next = head
cur = dummy
while cur.next:
    if cur.next.val == target:
        cur.next = cur.next.next
    else:
        cur = cur.next
return dummy.next
```

Notice how we never touch `head` directly. We just return `dummy.next` at the end, and everything works cleanly.

Recursion

And finally, let's talk about **recursion**, a very natural way to handle linked lists because the structure itself is recursive.

In linked list problems, recursion is mostly used for **reversing** and **merging**. The key idea is: "Break the problem into smaller sublists, and rebuild the links when backtracking."

Here's a classic recursive reversal:

```
def reverseList(head):
    # Base case — reach the last node
    if not head or not head.next:
        return head

    new_head = reverseList(head.next)

    # Reverse the current connection
    head.next.next = head
    head.next = None
    return new_head
```

And here's the recursive version of merging two sorted lists:

```python
def mergeTwoLists(l1, l2):
    # base case
    if not l1 or not l2:
        return l1 or l2

    # Each time select a smaller node as the current head, and then recursively process the
rest.
    if l1.val < l2.val:
        l1.next = mergeTwoLists(l1.next, l2)
        return l1
    else:
        l2.next = mergeTwoLists(l1, l2.next)
        return l2
```

Notice how clean it looks, we don't manage any `while` loops or `pointers` manually. We just define the relationship between nodes, and recursion handles the rest.

Once we get comfortable with these, we will see how every linked list problem is just a variation of one of these three.

Question classification

Basic Operations

Node atomic operations such as traversal, counting, insertion, deletion, and reversal. All more complex problems are built on these operations.

203. Remove Linked List Elements

206. Reverse Linked List

876. Middle of the Linked List

Double pointers

"Relative position control" is achieved through two pointers with different speeds or distances.

141. Linked List Cycle

142. Linked List Cycle II

19. Remove Nth Node From End of List

21. Merge Two Sorted Lists

160. Intersection of Two Linked Lists

Changing substructure

The focus is on the local reorganization capability of the linked list, which is essentially "sub-interval reversal and splicing, and boundary maintenance."

25. Reverse Nodes in k-Group

92. Reverse Linked List II

143. Reorder List

725. Split Linked List in Parts

Complex linked lists

Nodes no longer have a single `next` pointer, but instead have additional pointers like `random` and `child`. Essentially, this examines "how to maintain logical consistency as the number of pointers increases."

138. Copy List with Random Pointer

430. Flatten a Multilevel Doubly Linked List