Basic knowledge

Why Do We Need Linked Lists?

Array

Advantages:

- O(1) random access by index
- Contiguous memory, good cache locality

Limitations:

- Fixed length
- O(n) insertion/deletion due to element shifting

Linked Lists

Advantages

- Dynamic size (nodes added/removed anytime). Nodes stored non-contiguously (flexible memory use)
- O(1) insertion/deletion (if pointer known)

Use When: Data changes frequently, or the total size is unpredictable.

Basic Structure

A **Linked List** = sequence of **nodes** connected by **pointers**.

Each node has:

1. `val` → actual value
2. `next` → reference to the next node

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

Key Idea: Unlike arrays (indexed), linked lists depend on **pointer references**.

Types

Singly Linked List

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

- Traversal is one-way (head → tail)
- Deletion needs the `prev` pointer

```
# 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`.

- Supports bidirectional traversal
- Deletion becomes easier (no need to track `prev`). Slightly higher memory cost

```
# 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

- No `None` terminator — forms a loop. Traversal needs stopping condition to avoid infinite loop.

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

Common Operations

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

Move step by step until reaching `None`:

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

Insertion

Modify pointers only — no shifting required:

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

Deletion

Skip the node to remove:

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

Reversing

Change all `next` pointers to point backward:

Key point: Save `next_node` *before* breaking the link, or the rest of the list will be lost.

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

Leetcode techniques

Two Pointers

Core Idea: Use one traversal to achieve two goals.

- By using two pointers that move at different speeds or keep a fixed distance, we can collect key information efficiently during a single pass.

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

1. 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.

Typical Use Cases:

- 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

The **Dummy Head** (or *sentinel node*) is a "fake" node placed before the actual head.

Without it, we often need special handling for the head node during:

- Insertion at the beginning
- Deletion of the first node
- Empty list checks

With a dummy node, every node is treated the same.

```
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
```

- Don't manipulate `head` directly
- Always return `dummy.next`
- Simplifies all insertion/deletion logic

Recursion

Linked lists are naturally recursive structures. Each node connects to a smaller “sub-list.” Recursion lets us process them elegantly without manual loops.

Here’s a classic recursive reversal:

- Reverse the list by re-linking during the backtracking phase.

```
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:

- Always pick the smaller head, then recursively merge the rest. The recursion automatically links nodes in sorted order.

```
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
```

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]

430. Flatten a Multilevel Doubly Linked List