Linked List Ultimate Cheat Sheet: From Basics to Advanced Patterns

1. Introduction

- **Linked List:** A dynamic linear data structure where each element (node) points to the next.
- Unlike arrays, linked lists enable efficient insertions/deletions without shifting.

Types of Linked Lists

Type	Description
Singly Linked List	Each node points to next only
Doubly Linked List	Nodes point to both next and previous
Circular Linked List	Last node points back to head

2. Node and Basic List Structure in Python

Singly Linked List Node

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Singly Linked List Class (Basic Append and Print)

```
class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        cur = self.head
        while cur.next:
            cur = cur.next
        cur.next = new_node

    def print_list(self):
        curr = self.head
        while curr:
            print(curr.data, end=" -> ")
            curr = curr.next
        print("None")
```

3. Core Operations

Operation	Purpose	Example Code Snippet
Insert at beginning	Add a new head node	`insert_at_beginning()` (see next block)
Insert at tail	Append node	`append(data)` as above
Insert by position	Add node at a specific index	Custom method
Delete by value	Remove first match	`delete_node(data)` (see next block)
Delete by position	Remove at index	Custom method
Search	Find node or index	Custom method
Reverse	Reverse the list	See reverse examples below
Length	Number of nodes	Traverse and count

Insert at Head Example

```
def insert_at_beginning(self, data):
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node
```

Delete Node by Value Example

```
def delete_node(self, key):
    cur = self.head
    prev = None
    while cur and cur.data != key:
        prev = cur
        cur = cur.next
    if not cur:
        return
    if prev is None:
        self.head = cur.next
    else:
        prev.next = cur.next
    cur = None
```

4. Reversing a Linked List

Iterative

```
def reverse(self):
    prev = None
    cur = self.head
    while cur:
        nxt = cur.next
        cur.next = prev
        prev = cur
        cur = nxt
    self.head = prev
```

Recursive

```
def reverse_recursive(self, node):
    if node is None or node.next is None:
        self.head = node
        return node
    rest = self.reverse_recursive(node.next)
    node.next.next = node
    node.next = None
    return rest
```

5. Doubly Linked List (Basics)

```
class DoublyNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = DoublyNode(data)
        if not self.head:
            self.head = new_node
            return
        cur = self.head
        while cur.next:
            cur = cur.next
        cur.next = new_node
        new_node.prev = cur
```

6. Advanced Linked List Patterns with Full Python Solutions

Pattern 1: Detect Cycle (Floyd’s Cycle Detection)

```
def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False
```

Pattern 2: Find Middle of Linked List

```
def middle_node(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow  # second middle if even length
```

Pattern 3: Reverse a Linked List (Iterative)

```
def reverse_list(head):
    prev = None
    curr = head
    while curr:
        nxt = curr.next
        curr.next = prev
        prev = curr
        curr = nxt
    return prev
```

Pattern 4: Merge Two Sorted Linked Lists

```
def merge_sorted(l1, l2):
    dummy = Node(0)
    tail = dummy
    while l1 and l2:
        if l1.data < l2.data:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next
    tail.next = l1 or l2
    return dummy.next
```

Pattern 5: Remove Duplicates from Sorted List

```
def remove_duplicates(head):
    curr = head
    while curr and curr.next:
        if curr.data == curr.next.data:
            curr.next = curr.next.next
        else:
            curr = curr.next
    return head
```

Pattern 6: Reverse Nodes in k-Group

```
def reverse_k_group(head, k):
    count = 0
    curr = head
    while curr and count < k:
        curr = curr.next
        count += 1
    if count == k:
        prev = reverse_k_group(curr, k)
        while count > 0:
            nxt = head.next
            head.next = prev
            prev = head
            head = nxt
            count -= 1
        return prev
    return head
```

Pattern 7: Remove Nth Node from End of List

```
def remove_nth_from_end(head, n):
    dummy = Node(0)
    dummy.next = head
    fast = slow = dummy
    for _ in range(n):
        fast = fast.next
    while fast.next:
        fast = fast.next
        slow = slow.next
    slow.next = slow.next.next
    return dummy.next
```

Pattern 8: Find Intersection Node of Two Linked Lists

```
def get_intersection_node(headA, headB):
    a, b = headA, headB
    while a != b:
        a = a.next if a else headB
        b = b.next if b else headA
    return a
```

Pattern 9: Palindrome Linked List Check

```
def is_palindrome(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    prev = None
    curr = slow
    while curr:
        nxt = curr.next
        curr.next = prev
        prev = curr
        curr = nxt

    left, right = head, prev
    while right:
        if left.data != right.data:
            return False
        left = left.next
        right = right.next
    return True
```

Pattern 10: Flatten Multilevel Linked List (Doubly)

```
def flatten(head):
    if not head:
        return head
    curr = head
    while curr:
        if getattr(curr, 'child', None):
            child = flatten(curr.child)
            nxt = curr.next
            curr.next = child
            child.prev = curr

            tail = child
            while tail.next:
                tail = tail.next
            tail.next = nxt
            if nxt:
                nxt.prev = tail
            curr.child = None
        curr = curr.next
    return head
```

7. Complexity Summary

Operation	Time Complexity	Space Complexity
Traverse/Print	O(n)	O(1)
Insert at Head	O(1)	O(1)
Insert at Tail	O(n) (O(1) with tail ptr)	O(1)
Search	O(n)	O(1)
Delete (by value)	O(n)	O(1)
Reverse	O(n)	O(1)
Detect Cycle	O(n)	O(1)
Merge Sorted Lists	O(n)	O(1)

8. Tips & Best Practices

- Use dummy nodes to simplify edge cases.
- Always check for `None` before accessing `.next`.
- Recursive methods are elegant but watch stack limits.
- Maintain tail pointer if many tail inserts.
- Visualize list changes during debugging for clarity.