Linked List

Core Patterns Identified in This Chunk:

1. **Two Pointers** (with dummy node or fast/slow)
2. **Pointer Rewiring** (reversing, reordering, splitting linked lists)
3. **Fast/Slow Pointers** (cycle detection, middle finding)
4. **Dummy Node Technique** (clean traversal and manipulation)
5. **Hash Map + Doubly Linked List** (for LRU Cache — advanced but critical)

Pattern 1: Two Pointers (with Dummy Node)

How to Recognize

- You're working with a **linked list** and need to traverse it efficiently.
- Common use cases:
  - Merging two sorted lists
  - Removing nodes from end (e.g., Nth from end)
  - Swapping adjacent nodes
- Look for phrases like:
  - "Remove the nth node from the end"
  - "Merge two sorted lists"
  - "Swap every two adjacent nodes"

Step-by-Step Thinking Process (Recipe)

1. Use a **dummy head** to avoid edge case handling (e.g., removing the first node).
2. Initialize two pointers: `left` and `right`.
3. Position them appropriately (e.g., `right` starts at `head`, `left` at dummy).
4. Move one pointer ahead by N steps (if needed).
5. Move both pointers until `right` reaches the end.

6. Now `left.next` is the node to remove/edit.
7. Perform the required operation (update `next`, reverse links, etc.).

Pitfalls & Edge Cases

- Forgetting to return `dummy.next` instead of `head`.
- Not handling empty list (`head == None`).
- Off-by-one errors when counting from end.
- Not updating `prev` correctly during rewiring.

Pattern 2: Fast/Slow Pointers (Floyd's Cycle Detection)

How to Recognize

- Problem asks about:
  - Finding the **middle** of a linked list.
  - Detecting a **cycle**.
  - Determining if a list has a loop.
- Key phrase: "find the middle", "detect cycle", "loop".

Step-by-Step Thinking Process (Recipe)

1. Initialize two pointers: `slow = head`, `fast = head`.
2. Move `fast` two steps per iteration, `slow` one step.
3. When `fast` hits the end (`fast == None` or `fast.next == None`), `slow` is at the middle.
4. For cycle detection:
   - If `fast` meets `slow` again → cycle exists.
   - Otherwise, no cycle.

Pitfalls & Edge Cases

- `fast` might be `None` before `fast.next`, so check `fast and fast.next`.
- Don't forget to reset pointers after detecting cycle.
- In some variants (like reorder), you must reverse the second half properly.

Pattern 3: Pointer Rewiring (Manual Link Manipulation)

How to Recognize

- You're asked to:
  - Reverse a sublist.
  - Swap pairs.
  - Reorder nodes (odd/even split).
  - Split and merge lists.
- The solution requires manually changing `.next` pointers.

Step-by-Step Thinking Process (Recipe)

1. Use temporary variables to store references (`prev`, `curr`, `nxt`).
2. Traverse while saving next node before modifying current.
3. Update `current.next = previous`.
4. Move `previous` and `current` forward.
5. Be careful not to lose the chain.

Pitfalls & Edge Cases

- Losing reference to the rest of the list.
- Not returning the new head (especially after reversal).
- Misplacing `prev` or `head` after loops.

Pattern 4: Dummy Node Technique

How to Recognize

- You're doing operations that may affect the **head** of the list.
- Examples: insertion, deletion, merging.
- Avoids writing special logic for head changes.

Step-by-Step Thinking Process (Recipe)

1. Create a dummy node: `dummy = ListNode(0)`.
2. Set `dummy.next = head`.
3. Use `cur = dummy` as the working pointer.
4. After all operations, return `dummy.next`.

Pitfalls & Edge Cases

- Forgetting to return `dummy.next`.
- Using `dummy` directly instead of `dummy.next`.

Pattern 5: Hash Map + Doubly Linked List (LRU Cache)

How to Recognize

- You're implementing an **LRU (Least Recently Used)** cache.
- Need to support `get(key)` and `put(key, value)` in O(1).
- Must maintain order of usage.

Step-by-Step Thinking Process (Recipe)

1. Use a **hash map** to store `{key: node}` for O(1) access.
2. Use a **doubly linked list** to maintain order:
    - Most recently used at front.
    - Least recently used at back.
3. On `get`:
    - If key exists → move node to front.
    - Return value.
4. On `put`:
    - If key exists → update and move to front.
    - Else add new node to front.
    - If size > capacity → remove tail node.
5. Maintain helper methods: `add_to_front(node)`, `remove_node(node)`, `pop_tail()`.

Pitfalls & Edge Cases

- Forgetting to remove old node before adding new one.
- Not updating hash map on removal.
- Handling empty cache.
- Double-checking `self.capacity` vs actual size.

1. Merge Two Sorted Lists

Summary

Merge two sorted linked lists into a single sorted list.

Pattern(s)

- Two Pointers (with Dummy Node)

Solution with Inline Comments

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

def mergeTwoLists(list1, list2):
    # Create a dummy node to simplify pointer management
    dummy = ListNode(0)
    # 'tail' points to the last node in merged list
    tail = dummy

    # While both lists are non-empty
    while list1 and list2:
        # Compare values; attach smaller one
        if list1.val < list2.val:
            tail.next = list1
            list1 = list1.next
        else:
            tail.next = list2
            list2 = list2.next
        # Move tail forward
        tail = tail.next

    # Attach remaining nodes (one list might be non-empty)
    if list1:
        tail.next = list1
    elif list2:
        tail.next = list2

    # Return the merged list (skip dummy)
```

```python
    return dummy.next


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: list1 = [1,2,4], list2 = [1,3,4]
    l1 = ListNode(1, ListNode(2, ListNode(4)))
    l2 = ListNode(1, ListNode(3, ListNode(4)))

    # Call function
    merged = mergeTwoLists(l1, l2)

    # Output: [1,1,2,3,4,4]
    result = []
    while merged:
        result.append(merged.val)
        merged = merged.next
    print("Output:", result)  # Output: [1, 1, 2, 3, 4, 4]
```

Walkthrough (Example)

- Initial: `list1 = [1,2,4]`, `list2 = [1,3,4]`, `dummy -> None`
- Step 1: `1 <= 1` → attach `list1` (val=1), now `list1 = [2,4]`
- Step 2: `2 > 1` → attach `list2` (val=1), now `list2 = [3,4]`
- Step 3: `2 < 3` → attach `list1` (val=2)
- Step 4: `3 < 4` → attach `list2` (val=3)
- Step 5: `4 == 4` → attach `list1` (val=4), `list1` becomes empty
- Final: Attach remaining `list2` → `[4]`
- Result: `[1,1,2,3,4,4]`

Complexity

- **Time:** O(m + n), where m, n are lengths of lists
- **Space:** O(1), only pointers used (excluding output)

2. Linked List Cycle

Summary

Determine if a linked list has a cycle. Return `True` if yes, `False` otherwise.

Pattern(s)

- Fast/Slow Pointers

Solution with Inline Comments

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

def hasCycle(head):
    # Handle empty list
    if not head or not head.next:
        return False

    # Initialize slow and fast pointers
    slow = head
    fast = head

    # Move slow by 1 step, fast by 2 steps
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

        # If they meet, there's a cycle
        if slow == fast:
            return True

    # If fast reaches end, no cycle
    return False


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: head = [3,2,0,-4], pos = 1 (cycle to node with value 2)
    # Create nodes
    node0 = ListNode(3)
    node1 = ListNode(2)
    node2 = ListNode(0)
    node3 = ListNode(-4)
```

```python
    # Link them
    node0.next = node1
    node1.next = node2
    node2.next = node3
    node3.next = node1  # creates cycle back to node1 (pos=1)

    # Check for cycle
    print("Has Cycle:", hasCycle(node0))  # Output: True
```

Walkthrough (Example)

- `slow = 3`, `fast = 3`
- Step 1: `slow = 2`, `fast = 0`
- Step 2: `slow = 0`, `fast = -4`
- Step 3: `slow = -4`, `fast = 2`
- Step 4: `slow = 2`, `fast = 0`
- Step 5: `slow = 0`, `fast = -4`
- Step 6: `slow = -4`, `fast = 2`
- Step 7: `slow = 2`, `fast = 0`
- Eventually, `slow == fast` → cycle detected.

Complexity

- **Time:** O(n), worst case: fast goes around cycle once
- **Space:** O(1)

3. Reverse Linked List

Summary

Reverse a singly linked list.

Pattern(s)

- Pointer Rewiring

Solution with Inline Comments

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

def reverseList(head):
    prev = None
    curr = head

    # Traverse the list
    while curr:
        # Store next node before breaking link
        nxt = curr.next

        # Reverse the link: curr → prev
        curr.next = prev

        # Move prev and curr forward
        prev = curr
        curr = nxt

    # prev now points to the new head
    return prev


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: head = [1,2,3,4,5]
    head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))

    # Call function
    new_head = reverseList(head)

    # Output: [5,4,3,2,1]
    result = []
    while new_head:
        result.append(new_head.val)
        new_head = new_head.next
    print("Output:", result)  # Output: [5, 4, 3, 2, 1]
```

Walkthrough (Example)

- Start: `prev=None`, `curr=1`
- Iteration 1: `nxt=2`, `1.next=None`, `prev=1`, `curr=2`
- Iteration 2: `nxt=3`, `2.next=1`, `prev=2`, `curr=3`
- … continues until `curr=None`
- Final: `prev=5`, which is the new head.

Complexity

- **Time:** O(n)
- **Space:** O(1)

4. Middle of the Linked List

Summary

Find the middle node of a linked list. If even length, return the second middle.

Pattern(s)

- Fast/Slow Pointers

Solution with Inline Comments

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

def middleNode(head):
    # Both pointers start at head
    slow = head
    fast = head

    # Fast moves twice as fast
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # When fast reaches end, slow is at middle
    return slow
```

```python
# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: head = [1,2,3,4,5]
    head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))

    # Find middle
    mid = middleNode(head)
    print("Middle Value:", mid.val)  # Output: 3

    # Example Input: head = [1,2,3,4,5,6]
    head2 = ListNode(
    1,
    ListNode(
        2,
        ListNode(
            3,
            ListNode(
                4,
                ListNode(
                    5,
                    ListNode(6))))))
    mid2 = middleNode(head2)
    print("Middle Value:", mid2.val)  # Output: 4
```

Walkthrough (Example 1)

- `slow=1`, `fast=1`
- Step 1: `slow=2`, `fast=3`
- Step 2: `slow=3`, `fast=5`
- Step 3: `fast.next = None` → stop
- Return `slow=3`

Complexity

- **Time:** O(n)
- **Space:** O(1)

5. LRU Cache

Summary

Implement an LRU cache with `get(key)` and `put(key, value)` in O(1).

Pattern(s)

- Hash Map + Doubly Linked List

Solution with Inline Comments

```
class DListNode:
    def __init__(self, key=0, val=0):
        self.key = key
        self.val = val
        self.prev = None
        self.next = None

class LRUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.cache = {}  # maps key -> DListNode
        self.head = DListNode()  # dummy head
        self.tail = DListNode()  # dummy tail
        self.head.next = self.tail
        self.tail.prev = self.head

    def _add_node(self, node):
        """Insert node right after head (most recent)"""
        node.prev = self.head
        node.next = self.head.next
        self.head.next.prev = node
        self.head.next = node

    def _remove_node(self, node):
        """Remove node from list"""
        node.prev.next = node.next
        node.next.prev = node.prev

    def _move_to_head(self, node):
        """Move existing node to head (most recent)"""
        self._remove_node(node)
        self._add_node(node)

    def _pop_tail(self):
```

```python
        """Remove and return tail node (least recent)"""
        node = self.tail.prev
        self._remove_node(node)
        return node

    def get(self, key: int) -> int:
        if key not in self.cache:
            return -1
        node = self.cache[key]
        self._move_to_head(node)
        return node.val

    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            # Update existing node
            node = self.cache[key]
            node.val = value
            self._move_to_head(node)
        else:
            # New node
            node = DListNode(key, value)
            self.cache[key] = node
            self._add_node(node)

            # If over capacity, remove least recent
            if len(self.cache) > self.capacity:
                removed = self._pop_tail()
                del self.cache[removed.key]


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Initialize cache with capacity 2
    lru = LRUCache(2)

    # Operations
    lru.put(1, 1)
    lru.put(2, 2)
    print("Get 1:", lru.get(1))  # Output: 1
    lru.put(3, 3)  # Removes key 2
    print("Get 2:", lru.get(2))  # Output: -1
    lru.put(4, 4)  # Removes key 1
```

```python
    print("Get 1:", lru.get(1))  # Output: -1
    print("Get 3:", lru.get(3))  # Output: 3
    print("Get 4:", lru.get(4))  # Output: 4
```

Walkthrough (Example)

- Put(1,1): cache={1:node1}, list: head <-> 1 <-> tail
- Put(2,2): cache={1:node1,2:node2}, list: head <-> 2 <-> 1 <-> tail
- Get(1): move 1 to front → list: head <-> 1 <-> 2 <-> tail
- Put(3,3): capacity full → remove 2 → list: head <-> 3 <-> 1 <-> tail
- Get(2): returns -1

Complexity

- **Time:** O(1) for both `get` and `put`
- **Space:** O(capacity)

6. Remove Nth Node From End of List

Summary

Remove the nth node from the end of a linked list.

Pattern(s)

- Two Pointers + Dummy Node

Solution with Inline Comments

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

def removeNthFromEnd(head, n):
    # Dummy node helps handle edge case: removing head
    dummy = ListNode(0)
    dummy.next = head

    # Left and right pointers
```

```python
    left = dummy
    right = head

    # Move right n steps ahead
    for _ in range(n):
        right = right.next

    # Move both until right reaches end
    while right:
        left = left.next
        right = right.next

    # Now left.next is the node to remove
    left.next = left.next.next

    # Return new head
    return dummy.next


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: head = [1,2,3,4,5], n = 2
    head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))

    # Remove 2nd from end → remove 4
    new_head = removeNthFromEnd(head, 2)

    # Output: [1,2,3,5]
    result = []
    while new_head:
        result.append(new_head.val)
        new_head = new_head.next
    print("Output:", result)  # Output: [1, 2, 3, 5]
```

Walkthrough (Example)

- `left=dummy`, `right=head=1`
- Move `right` 2 steps: `right=3`
- Then move both until `right=None`:
  - `left=1`, `right=4`
  - `left=2`, `right=5`

- `left=3`, `right=None`

- Remove `left.next` (node 4)
- Result: `[1,2,3,5]`

Complexity

- **Time:** O(n)
- **Space:** O(1)

7. Swap Nodes in Pairs

Summary

Swap every two adjacent nodes in a linked list.

Pattern(s)

- Pointer Rewiring + Dummy Node

Solution with Inline Comments

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

def swapPairs(head):
    # Dummy node to simplify handling
    dummy = ListNode(0)
    dummy.next = head
    prev = dummy

    # Traverse in pairs
    while prev.next and prev.next.next:
        # Nodes to swap
        first = prev.next
        second = first.next

        # Swap: prev → second → first → rest
        prev.next = second
```

```python
        first.next = second.next
        second.next = first

        # Move prev two steps forward
        prev = first

    return dummy.next


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: head = [1,2,3,4]
    head = ListNode(1, ListNode(2, ListNode(3, ListNode(4))))

    # Swap pairs
    swapped = swapPairs(head)

    # Output: [2,1,4,3]
    result = []
    while swapped:
        result.append(swapped.val)
        swapped = swapped.next
    print("Output:", result)  # Output: [2, 1, 4, 3]
```

Walkthrough (Example)

- `prev=dummy`, `first=1`, `second=2`
- Swap: `dummy → 2 → 1 → 3 → 4`
- Move `prev=1`
- Next pair: `first=3`, `second=4`
- Swap: `1 → 4 → 3 → None`
- Final: `2 → 1 → 4 → 3`

Complexity

- **Time:** O(n)
- **Space:** O(1)

8. Odd Even Linked List

Summary

Reorder a linked list so that all odd-positioned nodes come before even-positioned ones, preserving relative order.

Pattern(s)

- Pointer Rewiring (splitting and merging)
- Two Pointers (odd/even heads)

Solution with Inline Comments

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

def oddEvenList(head):
    # Handle empty or single node
    if not head or not head.next:
        return head

    # Create two dummy heads for odd and even lists
    odd_head = ListNode(0)
    even_head = ListNode(0)

    odd_curr = odd_head
    even_curr = even_head
    curr = head
    is_odd = True  # Start with odd position (1st node)

    # Traverse and split
    while curr:
        if is_odd:
            odd_curr.next = curr
            odd_curr = curr
        else:
            even_curr.next = curr
            even_curr = curr
        # Toggle for next node
        is_odd = not is_odd
        curr = curr.next

    # Terminate both lists
```

```python
    odd_curr.next = None
    even_curr.next = None

    # Merge: odd list → even list
    odd_curr.next = even_head.next

    # Return new head (first node of odd list)
    return odd_head.next


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: head = [1,2,3,4,5]
    head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))

    # Reorder
    result_head = oddEvenList(head)

    # Output: [1,3,5,2,4]
    result = []
    while result_head:
        result.append(result_head.val)
        result_head = result_head.next
    print("Output:", result)  # Output: [1, 3, 5, 2, 4]
```

Walkthrough (Example)

- `curr=1` (odd): attach to `odd_curr`, `odd_curr=1`
- `curr=2` (even): attach to `even_curr`, `even_curr=2`
- `curr=3` (odd): `odd_curr=3`
- `curr=4` (even): `even_curr=4`
- `curr=5` (odd): `odd_curr=5`
- Now: `odd_list = 1→3→5`, `even_list = 2→4`
- Link: `5.next = 2`
- Final: `1→3→5→2→4`

Complexity

- **Time:** O(n)
- **Space:** O(1)

9. Add Two Numbers

Summary

Add two numbers represented as reverse-linked lists (each digit in a node). Return sum as a similar list.

Pattern(s)

- Linked List + Arithmetic (carry logic)

Solution with Inline Comments

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

def addTwoNumbers(l1, l2):
    # Dummy head for result
    dummy = ListNode(0)
    curr = dummy
    carry = 0

    # Process both lists and carry
    while l1 or l2 or carry:
        # Get values (0 if list exhausted)
        val1 = l1.val if l1 else 0
        val2 = l2.val if l2 else 0

        # Sum including carry
        total = val1 + val2 + carry

        # New digit and carry
        carry = total // 10
        digit = total % 10

        # Add digit to result
        curr.next = ListNode(digit)
        curr = curr.next

        # Move forward
```

```python
        l1 = l1.next if l1 else None
        l2 = l2.next if l2 else None

    return dummy.next


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: l1 = [2,4,3], l2 = [5,6,4]
    # Represents 342 + 465 = 807
    l1 = ListNode(2, ListNode(4, ListNode(3)))
    l2 = ListNode(5, ListNode(6, ListNode(4)))

    # Add
    result = addTwoNumbers(l1, l2)

    # Output: [7,0,8]
    output = []
    while result:
        output.append(result.val)
        result = result.next
    print("Output:", output)  # Output: [7, 0, 8]
```

Walkthrough (Example)

- Step 1: `2+5=7`, carry=0 → node 7
- Step 2: `4+6=10`, carry=1 → digit=0, node 0
- Step 3: `3+4+1=8`, carry=0 → node 8
- Done → `7→0→8`

Complexity

- **Time:** O(max(m,n))
- **Space:** O(max(m,n)) for result

10. Sort List

Summary

Sort a linked list in ascending order using **merge sort (divide & conquer)**.

Pattern(s)

- Divide & Conquer (Merge Sort)
- Fast/Slow Pointers (to split list)

Solution with Inline Comments

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

def sortList(head):
    # Base case: empty or single node
    if not head or not head.next:
        return head

    # Find middle using fast/slow pointers
    slow = head
    fast = head.next
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # Split: right half starts at slow.next
    mid = slow.next
    slow.next = None  # Break link

    # Recursively sort both halves
    left = sortList(head)
    right = sortList(mid)

    # Merge sorted halves
    return merge(left, right)

def merge(l1, l2):
    dummy = ListNode(0)
    curr = dummy

    while l1 and l2:
        if l1.val < l2.val:
            curr.next = l1
```

```
            l1 = l1.next
        else:
            curr.next = l2
            l2 = l2.next
        curr = curr.next

    # Attach remaining
    if l1:
        curr.next = l1
    if l2:
        curr.next = l2

    return dummy.next


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: head = [4,2,1,3]
    head = ListNode(4, ListNode(2, ListNode(1, ListNode(3))))

    # Sort
    sorted_head = sortList(head)

    # Output: [1,2,3,4]
    result = []
    while sorted_head:
        result.append(sorted_head.val)
        sorted_head = sorted_head.next
    print("Output:", result)  # Output: [1, 2, 3, 4]
```

Walkthrough (Example)

- Split: 4→2→1→3 → left: 4→2, right: 1→3
- Recurse: `sort([4,2])` → split → 4 and 2 → merge → 2→4
- Recurse: `sort([1,3])` → merge → 1→3
- Merge 2→4 and 1→3: compare → 1, then 2, 3, 4 → 1→2→3→4

Complexity

- **Time:** O(n log n)
- **Space:** O(log n) due to recursion stack

11. Palindrome Linked List

Summary

Check if a linked list reads the same forwards and backwards.

Pattern(s)

- Fast/Slow Pointers (find middle)
- Reverse Second Half
- Compare

Solution with Inline Comments

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

def isPalindrome(head):
    # Find middle using fast/slow
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # Reverse second half
    prev = None
    curr = slow
    while curr:
        nxt = curr.next
        curr.next = prev
        prev = curr
        curr = nxt

    # Compare first half and reversed second half
    left = head
    right = prev  # now points to start of reversed second half

    while right:
        if left.val != right.val:
            return False
```

```python
        left = left.next
        right = right.next

    return True


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: head = [1,2,2,1]
    head = ListNode(1, ListNode(2, ListNode(2, ListNode(1))))

    # Check palindrome
    print("Is Palindrome:", isPalindrome(head))  # Output: True

    # Example Input: head = [1,2]
    head2 = ListNode(1, ListNode(2))
    print("Is Palindrome:", isPalindrome(head2))  # Output: False
```

Walkthrough (Example 1)

- `slow` reaches node 2 (middle)
- Reverse second half: `2→1` becomes `1→2`
- Compare: `1==1`, `2==2` → true

Complexity

- **Time:** O(n)
- **Space:** O(1)

12. Reorder List

Summary

Reorder a list: L →L →...→L →L → L →L →L →L →...

Pattern(s)

- Fast/Slow (find mid)
- Reverse Second Half
- Merge Alternating

Solution with Inline Comments

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

def reorderList(head):
    if not head or not head.next:
        return

    # Find middle
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # Reverse second half
    prev = None
    curr = slow
    while curr:
        nxt = curr.next
        curr.next = prev
        prev = curr
        curr = nxt

    # Now prev is head of reversed second half
    # Merge first half and reversed second half
    first = head
    second = prev

    while second.next:
        # Save next nodes
        tmp1 = first.next
        tmp2 = second.next

        # Interleave
        first.next = second
        second.next = tmp1

        # Move forward
        first = tmp1
        second = tmp2
```

```python
# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: head = [1,2,3,4]
    head = ListNode(1, ListNode(2, ListNode(3, ListNode(4))))

    reorderList(head)

    # Output: [1,4,2,3]
    result = []
    while head:
        result.append(head.val)
        head = head.next
    print("Output:", result)  # Output: [1, 4, 2, 3]
```

Walkthrough (Example)

- `slow` at `3`, `fast` at `4`
- Reverse second half: `3→4` → `4→3`
- Merge: `1→4→2→3`
- `second.next` is `None` after `4→3`, so loop stops.

Complexity

- **Time:** O(n)
- **Space:** O(1)

13. Rotate List

Summary

Rotate a linked list to the right by k places.

Pattern(s)

- Two Pointers + Modular Arithmetic

Solution with Inline Comments

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

def rotateRight(head, k):
    if not head or not head.next:
        return head

    # Step 1: Get length and find tail
    length = 1
    tail = head
    while tail.next:
        tail = tail.next
        length += 1

    # Step 2: Normalize k
    k %= length
    if k == 0:
        return head  # no rotation needed

    # Step 3: Find new tail (k steps from end)
    # So we want to stop at length - k - 1
    new_tail = head
    for _ in range(length - k - 1):
        new_tail = new_tail.next

    # Step 4: New head is next of new_tail
    new_head = new_tail.next

    # Step 5: Break and reconnect
    new_tail.next = None
    tail.next = head  # connect old tail to old head

    return new_head


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: head = [1,2,3,4,5], k = 2
    head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
```

```python
# Rotate right by 2
rotated = rotateRight(head, 2)

# Output: [4,5,1,2,3]
result = []
while rotated:
    result.append(rotated.val)
    rotated = rotated.next
print("Output:", result)  # Output: [4, 5, 1, 2, 3]
```

Walkthrough (Example)

- Length = 5, `k=2`, so effective `k=2`
- New tail at position `5-2-1 = 2` → node `3`
- New head = `4`
- Break: `3.next = None`
- Connect `tail(5)` to `head(1)`
- Result: `4→5→1→2→3`

Complexity

- **Time:** O(n)
- **Space:** O(1)

14. Reverse Nodes in k-Group

Summary

Reverse every k nodes in groups. If fewer than k remain, leave them unchanged.

Pattern(s)

- Pointer Rewiring + Dummy Node + Reversing Sublist

Solution with Inline Comments

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

def reverseKGroup(head, k):
    # Dummy node to simplify
    dummy = ListNode(0)
    dummy.next = head
    prev_group_end = dummy

    while True:
        # Find kth node from prev_group_end
        kth = prev_group_end
        for _ in range(k):
            kth = kth.next
            if not kth:
                return dummy.next  # Less than k nodes left

        # Next group's start
        next_group_start = kth.next

        # Reverse the k nodes between prev_group_end and kth
        # We'll reverse from prev_group_end.next to kth
        current = prev_group_end.next
        prev = None

        while current != next_group_start:
            temp = current.next
            current.next = prev
            prev = current
            current = temp

        # Link reversed group
        # Old head now points to next group
        prev_group_end.next.next = next_group_start
        # New head is kth
        prev_group_end.next = kth

        # Move to next group
        # Jump to start of next group
        prev_group_end = prev_group_end.next.next
```

```python
# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: head = [1,2,3,4,5], k = 2
    head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))

    # Reverse in groups of 2
    result = reverseKGroup(head, 2)

    # Output: [2,1,4,3,5]
    output = []
    while result:
        output.append(result.val)
        result = result.next
    print("Output:", output)  # Output: [2, 1, 4, 3, 5]
```

Walkthrough (Example)

- Group 1: `1→2` → reverse → `2→1`
- `prev_group_end.next = 2`, `1.next = 4`
- Group 2: `3→4` → reverse → `4→3`
- `3.next = 5`
- Remaining `5` → untouched
- Final: `2→1→4→3→5`

Complexity

- **Time:** O(n)
- **Space:** O(1)