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

Problem Summary

You are given two **sorted linked lists**, and you need to merge them into a single sorted linked list.

- Input: Two `ListNode` heads (`list1`, `list2`)
- Output: A new sorted linked list containing all nodes from both input lists.

  Example:
  `list1 = [1,2,4]`, `list2 = [1,3,4]` → `merged = [1,1,2,3,4,4]`

Pattern(s)

- Two Pointers (with Dummy Node)

Solution with Inline Comments

```
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
```

```python
        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)
    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]
```

Step-by-Step Code Walkthrough

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

What is `ListNode`?

A simple node in a singly linked list: - `val`: stores the value. - `next`: points to the next node (or `None` if it's the last).

Function Definition

```
def mergeTwoLists(list1, list2):
```

This function takes two head nodes of sorted linked lists and returns the head of the merged sorted list.

---

1. Create a Dummy Node

```
dummy = ListNode(0)
tail = dummy
```

- **Why?** To avoid handling edge cases like empty lists or inserting the first node.
- `dummy` is a **fake head** that helps us build the result without worrying about where the real head goes.
- `tail` keeps track of the last node we added — so we can append new nodes easily.

  Think of `dummy` as a "placeholder" at the front. The actual result starts at `dummy.next`.

---

2. While Both Lists Are Non-Empty

```
while list1 and list2:
```

We compare the current values of both lists.

Compare & Attach Smaller Value

```
if list1.val < list2.val:
    tail.next = list1
    list1 = list1.next
else:
    tail.next = list2
    list2 = list2.next
```

- If `list1.val < list2.val`, attach `list1` to `tail.next`.
- Then move `list1` forward (`list1 = list1.next`).
- Otherwise, do the same for `list2`.

> This ensures the merged list stays sorted.

Move `tail` Forward

```
tail = tail.next
```

After attaching a node, update `tail` to point to the newly added node.

> Now `tail` will be ready to receive the next node.

---

3. Handle Remaining Nodes

```
if list1:
    tail.next = list1
elif list2:
    tail.next = list2
```

At this point, one of the lists is exhausted. But the other may still have remaining nodes.

Since both lists were already sorted, we can just **append the rest** of the non-empty list directly.

> No need to compare anymore!

---

4. Return Merged List

```
return dummy.next
```

- `dummy.next` is the **first real node** in our merged list.
- We skip the dummy node because it was only used for convenience.

Example Execution: `[1,2,4]` and `[1,3,4]`

Let's trace this manually:

Initial State:

```
list1: 1 → 2 → 4 → None
list2: 1 → 3 → 4 → None
dummy: 0 → ?
tail → points to dummy
```

Iteration 1: `list1.val == 1`, `list2.val == 1`

- Equal → go to `else`: attach `list2`
- `tail.next = list2` → now `tail` points to first `1` from `list2`
- Move `list2` → `list2` now points to `3`
- Move `tail` → now `tail` points to the first `1` (from list2)

  Current merged: `0 → 1` (from list2), then `1 → 3 → 4`

Iteration 2: `list1.val == 1`, `list2.val == 3`

- `1 < 3` → attach `list1`
- `tail.next = list1` → add second `1`
- Move `list1` → now `list1` points to `2`
- Move `tail` → now `tail` points to this `1`

  Merged: `0 → 1 → 1 → 2 → ...`

Iteration 3: `list1.val == 2`, `list2.val == 3`

- `2 < 3` → attach `list1`
- Add `2`, move `list1` → now `list1` points to `4`
- Move `tail`

  Merged: `0 → 1 → 1 → 2 → 3 → ...`

Iteration 4: `list1.val == 4`, `list2.val == 3`

- `4 > 3` → attach `list2`
- Add `3`, move `list2` → now `list2` points to `4`
- Move `tail`

  Merged: `... → 3 → 4`

Iteration 5: `list1.val == 4`, `list2.val == 4`

- Equal → attach `list2` (arbitrary choice due to `else`)
- Add `4`, move `list2` → now `list2` is `None`
- Move `tail`

Now `list2` is done.

Exit Loop: `list1` **still has** `4`**,** `list2` **is** `None`

So we run:

```
if list1:
    tail.next = list1
```

→ Append remaining `4` from `list1`.

Final merged list:

```
0 → 1 → 1 → 2 → 3 → 4 → 4 → None
```

Return `dummy.next` → which is the first `1`.

Final Output

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

Matches expected output.

Key Insights

Feature	Purpose
`dummy` node	Avoids special case handling for the first insertion
`tail` pointer	Keeps track of where to append next node
Linear traversal	O(m + n) time complexity, optimal
Stable merge	Equal elements handled correctly (e.g., `1 == 1`)

Time & Space Complexity

- **Time:** O(m + n) – each node visited once.
- **Space:** O(1) – only using pointers, not extra data structures (excluding output).

  Note: The returned list uses new nodes (not modifying inputs), so it's **not** in-place.

Summary

The algorithm elegantly merges two sorted linked lists using: - A **dummy head** to simplify logic, - A **two-pointer technique**, - And direct appending of leftovers.

Pro Tip: Always consider using a `dummy` node when building linked lists dynamically — it reduces code complexity and avoids edge-case bugs!

2. Linked List Cycle

Problem Overview

We are given a singly linked list and need to determine if it contains a **cycle** — meaning some node points back to a previous node, forming a loop.

- If there is a cycle → return `True`
- If no cycle → return `False`

The algorithm uses **two pointers**: - `slow`: moves 1 step at a time - `fast`: moves 2 steps at a time

If there is a cycle, the fast pointer will eventually "lap" the slow pointer inside the loop.

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 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)

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

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

Step-by-Step Code Walkthrough

1. Define the ListNode Class

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

This defines a node with a value (`val`) and a reference to the next node (`next`).

---

2. The `hasCycle(head)` Function

Handle Edge Cases

```python
if not head or not head.next:
    return False
```

- If the list is empty (`head is None`) or has only one node, it **cannot** form a cycle.
- So we return `False`.

  This avoids unnecessary processing.

Initialize Pointers

```python
slow = head
fast = head
```

Both pointers start at the beginning of the list.

Loop: Move Pointers

```
while fast and fast.next:
    slow = slow.next
    fast = fast.next.next
```

- `slow` moves one step forward.
- `fast` moves two steps forward.
- We check `fast and fast.next` to ensure `fast.next` exists before accessing `fast.next.next`.

  Why check `fast.next`? Because `fast.next.next` would cause an error if `fast.next` is `None`.

Check for Meeting Point

```
if slow == fast:
    return True
```

- If the two pointers meet at any point, that means there is a **cycle**.
- This is because the fast pointer is moving faster and can only catch up to the slow pointer if they're both in a loop.

  Key Insight: In a cycle, the fast pointer will eventually "lap" the slow pointer.

No Cycle Detected

```
return False
```

- If `fast` reaches the end (`fast is None` or `fast.next is None`), then there's no cycle.

Example Walkthrough

Given Input:

```
head = [3, 2, 0, -4], pos = 1
```

That means: - Node 0: `3` → points to node 1 - Node 1: `2` → points to node 2 - Node 2: `0` → points to node 3 - Node 3: `-4` → points back to **node 1** (i.e., cycle starts at index 1)

So the structure looks like:

```
3 → 2 → 0 → -4
    ↑       |
```

Now let's simulate the algorithm.

Simulation Step-by-Step

Step	`slow`	`fast`	Notes
Start	`node0 (3)`	`node0 (3)`	Both start at head
1	`node1 (2)`	`node2 (0)`	slow: +1, fast: +2
2	`node2 (0)`	`node1 (2)`	slow: +1, fast: +2 (from node2 → node3 → node1)
3	`node3 (-4)`	`node2 (0)`	slow: +1, fast: +2 (node1 → node2 → node3)
4	`node1 (2)`	`node1 (2)`	slow: +1 (node3 → node1), fast: +2 (node2 → node3 → node1)

At **step 4**, `slow == fast` → both are at `node1`.

They meet! → Return `True`

Final Output

```
Has Cycle: True
```

Which matches expected behavior.

Why Does This Work?

- In a **non-cyclic** list: `fast` will reach the end first → loop exits → return `False`.
- In a **cyclic** list: `fast` runs in a loop and will eventually catch up to `slow` (since it's faster).
- The meeting point doesn't matter — just the fact that they meet proves a cycle exists.

  Time Complexity: O(n) – worst case, fast goes around the loop a few times.

  Space Complexity: O(1) – only two pointers used.

Summary

Feature	Explanation
Algorithm	Floyd's Cycle Detection (Tortoise & Hare)
Logic	Fast pointer moves twice as fast; if they meet, cycle exists
Efficiency	Optimal: O(n) time, O(1) space
Use Case	Detecting cycles in linked lists (LeetCode #141)

---

3. Reverse Linked List

1. Summary

Reverse a singly linked list by changing the direction of the `next` pointers so that the last node becomes the first, and the original head becomes the tail.

---

2. Pattern(s)

- **Pointer Rewiring** (Primary)
- **Iterative Traversal with State Tracking**

  This problem teaches how to safely reverse links without losing access to the rest of the list.

---

3. Solution with Inline Comments

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

def reverseList(head):
    """
    Reverses a singly linked list iteratively.
    Time: O(n), Space: O(1)
    Returns new head of the reversed list.
    """
    prev = None   # Will become the new head at the end
    curr = head   # Current node we are processing

    while curr:  # Traverse until we reach the end
        nxt = curr.next # 1:Save the next node (don't lose the rest!)
        curr.next = prev # 2:Reverse the link (point backward)
        prev = curr # 3:Move prev forward (curr becomes part of reversed part)
        curr = nxt  # 4:Move curr to the next node

    return prev  # At the end, prev points to the last node → new head
```

---

4. Example Code Walkthrough (Easy & Visual)

Let’s reverse: `1 → 2 → 3 → null`

We’ll go **step by step**, showing what each pointer does.

Initial State:

```
prev = None
curr = 1 → 2 → 3 → null
```

Step 1: Process Node 1

```
nxt = curr.next    → nxt points to 2
curr.next = prev   → 1 now points to None (reverse!)
prev = curr        → prev moves to 1
curr = nxt         → curr moves to 2
```

After Step 1:

```
prev → 1 → None
curr → 2 → 3 → null
List so far: 1 is reversed; 2 and 3 still forward
```

Step 2: Process Node 2

```
nxt = curr.next    → nxt points to 3
curr.next = prev   → 2 now points to 1
prev = curr        → prev moves to 2
curr = nxt         → curr moves to 3
```

After Step 2:

```
prev → 2 → 1 → None
curr → 3 → null
List so far: 2→1 is reversed; 3 still forward
```

Step 3: Process Node 3

```
nxt = curr.next    → nxt = None (end of list)
curr.next = prev   → 3 now points to 2
prev = curr        → prev moves to 3
curr = nxt         → curr = None → loop ends
```

After Step 3:

```
prev → 3 → 2 → 1 → null
curr = None → exit loop
```

Final Result:

```
return prev  → returns 3 (the new head)
 List is reversed: 3 → 2 → 1 → null
```

---

5. Time and Space Complexity

Metric	Value	Explanation
Time Complexity	O(n)	We visit each node exactly once
Space Complexity	O(1)	Only use 3 variables (`prev`, `curr`, `nxt`) — no extra data structures

This is optimal — can't do better than O(n) time or O(1) space.

Bonus: Interview Tips

- **Say this aloud**:
  "I'll reverse the list by rewiring each node to point to its previous node, using three pointers to do it safely."
- **Clarify edge cases**:
  - If `head is None` or `head.next is None`, return `head` — but our code already handles this naturally.
- **Visualize as you code**:
  Draw `prev → curr → nxt` on paper or whiteboard.
- **Mention the recursive version briefly** (if asked):

```
def reverseListRecursive(head):
    if not head or not head.next:
        return head
    new_head = reverseListRecursive(head.next)
    head.next.next = head
    head.next = None
    return new_head
```

  - Time: O(n), Space: O(n) due to call stack.
- **Practice until muscle memory**:
  You should be able to write this in under 2 minutes — it's a warm-up question!

---

4. Middle of the Linked List

Problem Statement: Middle of the Linked List

Given the `head` of a singly-linked list, return **the middle node** of the linked list.

If there are two middle nodes (even number of nodes), return **the second middle node**.

Sample Input & Output

Example 1:

```
Input:  [1, 2, 3, 4, 5]
Output: Node with value 3
Explanation: The middle node is 3.
```

Example 2:

```
Input:  [1, 2, 3, 4, 5, 6]
Output: Node with value 4
Explanation: There are two middle nodes: 3 and 4. Return the second one → 4.
```

Approach: Two Pointers (Slow and Fast)

We use the **"Tortoise and Hare"** technique: - `slow` pointer moves **1 step** at a time. - `fast` pointer moves **2 steps** at a time. - When `fast` reaches the end, `slow` will be at the middle.

> Why does this work?
> By the time the fast pointer has gone the full length (`n`), the slow pointer has gone half (`n/2`).

Clean Code with Detailed Comments

```python
# Definition for a singly-linked list node
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def middleNode(head):
    """
    Find the middle node of a linked list using slow and fast pointers.
```

```
    Args:
        head: ListNode - The head of the linked list

    Returns:
        ListNode - The middle node of the list
    """
    # Initialize both pointers at the head
    slow = head
    fast = head

    # Traverse: move slow by 1 step, fast by 2 steps
    while fast and fast.next:
        slow = slow.next           # 1 step forward
        fast = fast.next.next      # 2 steps forward

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

Example Walkthrough

Example 1: `[1 → 2 → 3 → 4 → 5]`

Step	slow (1 step)	fast (2 steps)	fast.next check
0	1	1	Continue
1	2	3	Continue
2	3	5	fast.next = None → Stop

Loop ends → `slow` is at **3** → Correct middle.

Example 2: `[1 → 2 → 3 → 4 → 5 → 6]`

Step	slow	fast	fast.next check
0	1	1	yes

Step	slow	fast	fast.next check
1	2	3	yes
2	3	5	yes
3	4	None	Loop stops

`slow` ends at **4** → Second middle (as required) → Correct!

Time and Space Complexity

Metric	Value	Explanation
Time Complexity	O(n)	We traverse the list once with the fast pointer.
Space Complexity	O(1)	Only two pointers used — constant extra space.

> Optimal: No need to store values or traverse twice.

Key Takeaways for Interviews

1. **Two-pointer technique** is essential for linked list problems.
2. This method avoids:
   - Counting total nodes and looping again.
   - Using extra memory (like storing nodes in an array).
3. Handles **odd and even lengths** naturally.
4. Always check `fast and fast.next` to avoid `None.next` error.
5. Great pattern to know for:
   - Finding cycles (Floyd's Algorithm)
   - Reversing second half of list
   - Palindrome check in linked list

Bonus: How to Test It

```
if __name__ == "__main__":
    # Test Case 1: [1,2,3,4,5] → Output: 3
    head1 = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
    mid1 = middleNode(head1)
    print("Middle Value:", mid1.val)  # Output: 3

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

---

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:
```

```python
    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):
        """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]
```

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

Problem Overview: LRU Cache

- **LRU (Least Recently Used)**: When cache is full, remove the *least recently used* item.
- Supports:
  - `get(key)` → returns value if key exists, else `-1`
  - `put(key, value)` → inserts or updates key-value pair
- Must maintain order of usage (most recent at front, least recent at end)

We use a **doubly linked list + hash map (dictionary)** for O(1) operations.

Data Structures Used

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

Each node stores `key`, `value`, and pointers to previous/next nodes.

```
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
```

- `cache`: fast lookup via key → node
- `head` and `tail`: dummy nodes to simplify insertion/removal
- List structure: `head  node1  node2  ...  tail`

Key Helper Methods

1. `_add_node(node)`

Inserts a new node **right after head** (i.e., most recently used).

```
def _add_node(self, node):
    node.prev = self.head
    node.next = self.head.next
    self.head.next.prev = node
    self.head.next = node
```

Example:
Before: `head → A → B → tail`
After adding `X`: `head → X → A → B → tail`

Places the node at the front (most recent)

---

2. `_remove_node(node)`

Removes a node from the list.

```
def _remove_node(self, node):
    node.prev.next = node.next
    node.next.prev = node.prev
```

Breaks the links around `node`. Useful when moving or deleting.

---

3. `_move_to_head(node)`

Moves an existing node to the front (most recent).

```
def _move_to_head(self, node):
    self._remove_node(node)
    self._add_node(node)
```

Reuse helper functions to efficiently update position.

---

4. `_pop_tail()`

Removes the **least recently used** node (just before `tail`).

```
def _pop_tail(self):
    node = self.tail.prev
    self._remove_node(node)
    return node
```

> Returns the node so we can delete it from `cache`.

Core Operations

`get(key)`

```
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
```

- If not found → return `-1`
- If found → move to front (mark as recently used), return value

---

`put(key, value)`

```
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]
```

- If key exists: update value, move to front
- Else: create new node, add to front
- If size exceeds capacity: remove tail (least recent), delete from `cache`

Example Walkthrough

```
lru = LRUCache(2)
```

Initial state:

```
head   tail
cache = {}
```

Step 1: `lru.put(1, 1)`

- New key `1`, value `1`
- Create node: `DListNode(1, 1)`
- Add to front: `head → [1] → tail`
- `cache = {1: node_1}`

List: `head   1   tail`
Cache: `{1: node_1}`

Step 2: `lru.put(2, 2)`

- New key `2`, value `2`
- Create node: `DListNode(2, 2)`
- Add to front: `head → [2] → [1] → tail`
- `cache = {1: node_1, 2: node_2}`

List: `head   2   1   tail`
Cache: `{1: node_1, 2: node_2}`

Step 3: `print("Get 1:", lru.get(1))`

- Key `1` exists → get node
- Move `1` to front: remove `1`, insert after `head`
- Now: `head   1   2   tail`
- Return `1`

Output: `1`

Step 4: `lru.put(3, 3)`

- Key `3` not in cache → new entry
- Create `DListNode(3, 3)`
- Add to front: `head → 3 → 1 → 2 → tail`
- Now cache has 3 entries → over capacity (max 2)
- Remove **least recent**: tail's prev = `2`
- Remove `2` from list and cache

Final state: - List: `head   3   1   tail` - Cache: `{3: node_3, 1: node_1}`

`2` is evicted!

Output: `Get 2:` → `-1` (because `2` was removed)

Step 5: `lru.put(4, 4)`

- Key 4 not in cache → new entry
- Add 4 to front: `head → 4 → 3 → 1 → tail`
- Size now 3 → must remove least recent (1)
- Remove `1` from list and cache

Final state: - List: `head 4 3 tail` - Cache: `{4: node_4, 3: node_3}`

> `1` is gone!

Final Queries:

```python
print("Get 1:", lru.get(1))  # -1 (not in cache)
print("Get 3:", lru.get(3))  # 3 (was recently accessed)
print("Get 4:", lru.get(4))  # 4
```

Output:

```
Get 1: 1
Get 2: -1
Get 1: -1
Get 3: 3
Get 4: 4
```

Wait — there's a small discrepancy in expected output.

Let's correct the **actual expected outputs** based on logic:

Operation	Output
`get(1)` after `put(1,1)` and `put(2,2)`	`1`
`get(2)` after `put(3,3)`	`-1` (evicted)
`get(1)` after `put(4,4)`	`-1` (evicted)
`get(3)`	`3`
`get(4)`	`4`

So final printout should be:

```
Get 1: 1
Get 2: -1
Get 1: -1
Get 3: 3
Get 4: 4
```

Matches expected behavior.

Summary: How It Works

Feature	Implementation
Fast lookup	Hashmap (`cache`)
Order tracking	Doubly linked list
Most recent	At front (`head.next`)
Least recent	Just before `tail`
Insertion	At front
Removal	From tail when full
Update	Move existing node to front

Time Complexity

- `get()` → O(1)
- `put()` → O(1)
- All operations are constant time due to hash map + double linked list.

Why This Design?

- **Hash map**: O(1) access to any node
- **Doubly linked list**: O(1) insertion/deletion anywhere
- Dummy nodes eliminate edge cases (e.g., empty list)

This is the **standard optimal solution** for LRU Cache on LeetCode.

Pro Tips

- Always use **dummy head/tail** to avoid null checks
- Keep `cache` mapping key → node (so you don't need to search the list)
- When updating a key, **move it to head** to mark as recently used
- When inserting a new key, **add to head**, then **evict tail** if needed

---

6. Remove Nth Node From End of List

Problem Overview

Given a linked list and an integer `n`, remove the **n-th node from the end** of the list.

For example: - Input: `[1,2,3,4,5]`, `n = 2` - The 2nd node from the end is `4` - Output: `[1,2,3,5]`

Pattern(s)

- Two Pointers + Dummy Node

Solution with Inline Comments

```
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
    left = dummy
    right = head

    # Move right n steps ahead
```

```python
    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]
```

Key Idea: Two Pointers (Fast & Slow)

We use two pointers: - `left` starts at a dummy node. - `right` starts at the head.

We move `right` ahead by `n` steps first. Then we move both pointers together until `right` reaches the end (`None`). At that point, `left` will be just before the node to delete.

This way, we can easily bypass the target node using `left.next = left.next.next`.

Why Use a Dummy Node?

To handle edge cases like removing the **head** of the list.

Without a dummy: - If you try to remove the first node (i.e., when `n == length`), there's no "previous" node to update.

With a dummy: - The dummy acts as a fake head. - You always have a valid `left` pointer pointing to the node before the one to remove. - Even if you're removing the actual head, `dummy.next` will correctly point to the new head.

Step-by-Step Execution with Example

Input:

```
head = [1,2,3,4,5], n = 2
```

So the linked list looks like:

```
1 → 2 → 3 → 4 → 5 → None
```

Step 1: Create Dummy Node

```
dummy = ListNode(0)
dummy.next = head  # dummy → 1 → 2 → 3 → 4 → 5 → None
```

Now: - `left = dummy` (at 0) - `right = head` (at 1)

Step 2: Move `right` ahead by `n = 2` steps

Loop: `for _ in range(2)` - After 1st iteration: `right = 2` - After 2nd iteration: `right = 3`

Now: - `left = dummy` (still at 0) - `right = 3` (node with value 3)

Step 3: Move both pointers until `right` hits `None`

While `right` is not `None`: - Move both `left` and `right` one step forward.

Iteration 1:

- `left = 1`, `right = 4`

Iteration 2:

- `left = 2`, `right = 5`

Iteration 3:

- `left = 3`, `right = None` → loop ends

Now: - `left` is at node `3` - `right` is `None`

> So `left.next` is the node to remove — which is `4`.

Step 4: Remove the Node

```
left.next = left.next.next
```

That means:

```
# left.next was 4 → 5
# left.next.next is 5
# So now: left.next becomes 5
```

So the list becomes:

```
dummy → 1 → 2 → 3 → 5 → None
```

Step 5: Return `dummy.next`

Return `dummy.next`, which is `1`.

So final result: `[1, 2, 3, 5]`

Final Output

```
print("Output:", result)  # Output: [1, 2, 3, 5]
```

Correct!

Time & Space Complexity

Metric	Complexity
Time	O(L), where L is the length of the list (we traverse once)
Space	O(1), only using constant extra space

Very efficient!

Edge Case Check: Removing Head

Try `n = 5` (remove 5th from end → first node):

- `right` moves 5 steps: goes from 1 → 2 → 3 → 4 → 5 → `None`
- Now `right` is `None`, so the while loop doesn't run.
- `left` still points to `dummy`
- Then: `left.next = left.next.next` → removes `1`
- Return `dummy.next` → which is `2`

Works perfectly!

Summary

- **Algorithm**: Two-pointer technique with a dummy node.
- **Why it works**: Gap of `n` between `left` and `right` ensures `left` stops just before the target node.
- **Strengths**: Single pass, handles all edge cases cleanly.
- **Use Case**: Ideal for "remove Nth from end" problems.

Pro Tip

You can generalize this idea to other similar problems: - Find the middle of a linked list (fast/slow pointers). - Detect cycles (Floyd's algorithm). - Check if a linked list is a palindrome (reverse half + compare).

Complexity

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

7. Swap Nodes in Pairs

Problem Recap:

We are given a singly linked list. We need to **swap every two adjacent nodes**, i.e., swap (1st and 2nd), (3rd and 4th), etc., and return the new head.

For input:
`1 → 2 → 3 → 4`
Expected output:
`2 → 1 → 4 → 3`

We must **swap nodes**, not just values.

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
        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
```

```python
    print("Output:", result)  # Output: [2, 1, 4, 3]
```

Code Overview

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

This defines a node in a singly linked list.

```python
def swapPairs(head):
    dummy = ListNode(0)
    dummy.next = head
    prev = dummy
```

- We create a **dummy node** pointing to the head. This simplifies edge cases (like swapping the first pair).
- `prev` is a pointer that always points to the node **before** the current pair we're about to swap.

Main Loop: Swapping in Pairs

```python
while prev.next and prev.next.next:
```

We can only swap if there are **at least two nodes** after `prev`.

Inside the loop:

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

We identify the two nodes to swap: - `first`: first node of the pair - `second`: second node of the pair

Now perform the swap:

```
prev.next = second  # prev now points to second
# first skips second and points to whatever comes after
first.next = second.next
second.next = first # second now points to first → completes the swap
```

After this:

```
Before:   prev → first → second → X
After:    prev → second → first → X
```

So the pair is swapped.

Then:

```
prev = first  # Move prev forward by two nodes
```

Why `first`? Because after swapping: - `first` is now the **second** node in the new order. - So `prev = first` sets us up to process the next pair after this one.

Step-by-Step Example: `1→2→3→4`

Initial state:

```
dummy → 1 → 2 → 3 → 4
prev
```

First Iteration: Swap 1 and 2

- `first = 1`, `second = 2`
- `prev.next = second` → dummy → 2 …
- `first.next = second.next` → 1 → 3
- `second.next = first` → 2 → 1 Now:

```
dummy → 2 → 1 → 3 → 4
```

Then: `prev = first` → `prev = 1`

```
dummy → 2 → 1 → 3 → 4
             prev
```

Second Iteration: Swap 3 and 4

- `first = 3`, `second = 4`
- `prev.next = second` → 1 → 4
- `first.next = second.next` → 3 → None (since 4.next is None)
- `second.next = first` → 4 → 3

Now:

```
dummy → 2 → 1 → 4 → 3
```

Then: `prev = first = 3`

Now `prev.next` is `None`, so loop ends.

Return Result

```python
return dummy.next
```

Return the new head: `2 → 1 → 4 → 3`

Output Code

```python
result = []
while swapped:
    result.append(swapped.val)
    swapped = swapped.next
print("Output:", result)
```

This traverses the list and collects values: - `swapped` starts at `2` - Then `1`, `4`, `3` - Output: `[2, 1, 4, 3]`

Key Insights

Concept	Why It Matters
Dummy Node	Handles edge case when head is swapped (no need for special logic)
prev pointer	Always points to node before the pair; makes re-linking easy
Swapping nodes	Not just values — actual pointers are changed
Move prev = first	After swap, `first` is now the second node in the pair, so next pair starts after it

---

Edge Cases Handled

- Empty list (`head = None`) → returns `None`
- Single node → no swap, returns same
- Odd-length list → last node remains unswapped

Example: `1→2→3` becomes `2→1→3`

Final Thoughts

This algorithm runs in: - **Time:** O(n) — each node visited once - **Space:** O(1) — only a few pointers used

It's an elegant and efficient solution commonly used in LeetCode and interviews.

Tip for Interviews

You can briefly explain: > "I use a dummy node to simplify pointer manipulation. Then, for every pair, I relink the previous node to the second node, then rewire the two nodes in reverse, and move forward by two nodes."

---

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

```python
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
```

```python
        is_odd = not is_odd
        curr = curr.next

    # Terminate both lists
    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]
```

Let's walk through the provided Python code step by step, focusing on how the `swapPairs` function works to **swap every two adjacent nodes** in a linked list.

Problem Recap:

We are given a singly linked list. We need to **swap every two adjacent nodes**, i.e., swap (1st and 2nd), (3rd and 4th), etc., and return the new head.

For input:
`1 → 2 → 3 → 4`

Expected output:
`2 → 1 → 4 → 3`

We must **swap nodes**, not just values.

Code Overview

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

This defines a node in a singly linked list.

```
def swapPairs(head):
    dummy = ListNode(0)
    dummy.next = head
    prev = dummy
```

- We create a **dummy node** pointing to the head. This simplifies edge cases (like swapping the first pair).
- `prev` is a pointer that always points to the node **before** the current pair we're about to swap.

Main Loop: Swapping in Pairs

```
while prev.next and prev.next.next:
```

We can only swap if there are **at least two nodes** after `prev`.

Inside the loop:

```
first = prev.next
second = first.next
```

We identify the two nodes to swap: - `first`: first node of the pair - `second`: second node of the pair

Now perform the swap:

```
prev.next = second           # prev now points to second
first.next = second.next     # first skips second and points to whatever comes after
second.next = first          # second now points to first → completes the swap
```

After this:

```
Before:   prev → first → second → X
After:    prev → second → first → X
```

So the pair is swapped.

Then:

```
prev = first  # Move prev forward by two nodes
```

Why `first`? Because after swapping: - `first` is now the **second** node in the new order. - So `prev = first` sets us up to process the next pair after this one.

Step-by-Step Example: `1→2→3→4`

Initial state:

```
dummy → 1 → 2 → 3 → 4
prev
```

First Iteration: Swap 1 and 2

- `first = 1`, `second = 2`
- `prev.next = second` → dummy → 2 …
- `first.next = second.next` → 1 → 3
- `second.next = first` → 2 → 1 Now:

```
dummy → 2 → 1 → 3 → 4
```

Then: `prev = first` → `prev = 1`

```
dummy → 2 → 1 → 3 → 4
             prev
```

Second Iteration: Swap 3 and 4

- `first = 3`, `second = 4`
- `prev.next = second` → 1 → 4
- `first.next = second.next` → 3 → None (since 4.next is None)
- `second.next = first` → 4 → 3

Now:

```
dummy → 2 → 1 → 4 → 3
```

Then: `prev = first = 3`

Now `prev.next` is `None`, so loop ends.

Return Result

```python
return dummy.next
```

Return the new head: `2 → 1 → 4 → 3`

Output Code

```python
result = []
while swapped:
    result.append(swapped.val)
    swapped = swapped.next
print("Output:", result)
```

This traverses the list and collects values: - `swapped` starts at `2` - Then `1`, `4`, `3` - Output: `[2, 1, 4, 3]`

Key Insights

Concept	Why It Matters
Dummy Node	Handles edge case when head is swapped (no need for special logic)
prev pointer	Always points to node before the pair; makes re-linking easy
Swapping nodes	Not just values — actual pointers are changed
Move prev = first	After swap, `first` is now the second node in the pair, so next pair starts after it

---

Edge Cases Handled

- Empty list (`head = None`) → returns `None`
- Single node → no swap, returns same
- Odd-length list → last node remains unswapped

Example: `1→2→3` becomes `2→1→3`

Final Thoughts

This algorithm runs in: - **Time:** O(n) — each node visited once - **Space:** O(1) — only a few pointers used

It's an elegant and efficient solution commonly used in LeetCode and interviews.

---

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

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

    return dummy.next


# ---- Official LeetCode Example ----
```

```python
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

```
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
```

```python
    # 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

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

    return True
```

```python
# ---- 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

```python
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)

Let's walk through the provided `reorderList` function step-by-step with a detailed explanation, using the example input `[1,2,3,4]`.

Goal

Reorder a linked list such that: - The first node stays at the front, - Then the last node, - Then the second node, - Then the second-to-last node, - And so on.

For input: `[1,2,3,4]` → Output: `[1,4,2,3]`

This is known as **reordering a linked list in zigzag (interleaved) order**.

Step 1: Define the ListNode Class

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

Simple singly linked list node definition.

Step 2: Input Setup

```
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4))))
```

Creates the linked list:

```
1 -> 2 -> 3 -> 4
```

Step 3: `reorderList(head)` Function

Part A: Base Case

```
if not head or not head.next:
    return
```

If there's only one node or none, no reordering needed. We skip.

Our list has 4 nodes → continue.

Part B: Find the Middle Using Slow & Fast Pointers

```
slow = fast = head
while fast and fast.next:
    slow = slow.next
    fast = fast.next.next
```

Let's trace this:

Iteration	`fast`	`slow`
Start	`1`	`1`
1st	`3` (next of 2)	`2` (next of 1)
2nd	`4` (next of 3)	`3` (next of 2)
3rd	`None`	`4` (next of 3)

Wait! Let's correct:

Actually:

- Initially: `fast = 1`, `slow = 1`
- First loop:
  - `fast = fast.next.next = 3` → `fast.next = 4`, so `fast = 3`

  - `slow = slow.next = 2`
- Second loop:
  - `fast = fast.next.next = 4.next = None` → `fast = 4.next = None` → stops loop
  - So `fast` becomes `None`, exit loop.

So after loop: - `slow` is at **node 3** - `fast` is `None`

So middle is **node 3**, which splits the list into: - First half: `1 -> 2` - Second half: `3 -> 4`

> Note: Since length is even (4), we want to split after 2nd node. This method gives us the **start of second half** at node 3 — correct.

Part C: Reverse the Second Half

We reverse the part starting from `slow` (node 3):

```
prev = None
curr = slow  # curr starts at node 3
```

Now reverse the second half (`3 -> 4`) → becomes `4 -> 3`

Trace:

Step	`curr`	`nxt`	`curr.next = prev`	`prev = curr`	`curr = nxt`
1	3	4	3.next = None	prev = 3	curr = 4
2	4	None	4.next = 3	prev = 4	curr = None

After this loop: - `prev = 4` → new head of reversed second half - The list now looks like: - First half: `1 -> 2` - Reversed second half: `4 -> 3` (but still connected via original links)

But note: the original link `3->4` is broken during reversal.

So now: - `head = 1 -> 2` - `prev = 4 -> 3`

Part D: Merge First Half and Reversed Second Half

```
first = head      # first = 1
second = prev     # second = 4
```

Now merge by interleaving:

```
while second.next:
    tmp1 = first.next   # save first.next (which is 2)
    tmp2 = second.next  # save second.next (which is 3)

    # Interleave
    first.next = second        # 1 -> 4
    second.next = tmp1         # 4 -> 2

    # Move forward
    first = tmp1               # first = 2
    second = tmp2              # second = 3
```

Let's go step by step:

First iteration:

- `tmp1 = first.next = 2`
- `tmp2 = second.next = 3`
- Set:
  - `first.next = second` → `1.next = 4` → `1 -> 4`
  - `second.next = tmp1` → `4.next = 2` → `4 -> 2`
- Now:
  - `first = 2`
  - `second = 3`

Current state:

```
1 -> 4 -> 2 -> 3
```

And `3.next` is still `None`? Wait – what about `3`?

Originally, after reversal, `4 -> 3`, then `3.next` was set to `None` during reversal. But here we didn't change it.

Yes: `3.next` is `None`. So `second.next` is `None`.

So `while second.next:` → condition fails!

So we break out of the loop **after just one iteration**.

Problem Detected!

The loop condition is:

```python
while second.next:
```

That means we stop when `second.next` is `None`.

But in our case: - After first iteration, `second = 3`, and `3.next = None` → loop ends. - So we never connect `2` to `3`.

Wait — but currently: `- 1 -> 4 -> 2 - 2` points to `3`? No — `2.next` was saved as `tmp1`, but we did not update `2.next`.

Hold on — let's reconstruct the actual structure after the first iteration:

Before:

```
1 -> 2 -> 3 -> 4
```

After reversing second half:

```
1 -> 2
4 -> 3 (and 3.next = None)
```

Then: - `first = 1`, `second = 4` - `tmp1 = 2`, `tmp2 = 3` - `first.next = second` → `1.next = 4` - `second.next = tmp1` → `4.next = 2` - Then `first = 2`, `second = 3`

Now the chain is:

```
1 -> 4 -> 2
     ↑
     3 (but 3 is not connected to anything)
```

And `2.next` is still pointing to `3` originally? No — wait!

When we did `tmp1 = first.next`, we saved `2` as `tmp1`, but `2.next` is still `3`.

So after: - `4.next = 2` - `2.next` is still `3` → so `2 -> 3`

So now full list is:

```
1 -> 4 -> 2 -> 3
```

That's exactly what we want!

But why did the loop stop? Because `second = 3`, and `3.next = None`, so `second.next` is `None`, so `while second.next:` fails.

But that's okay! We don't need more iterations because: - We've already merged the two parts. - The remaining nodes are already connected correctly.

Wait — is `2 -> 3` valid?

Yes! After merging: - `1 -> 4` - `4 -> 2` - `2 -> 3` (original connection, untouched)

So final result: `1 -> 4 -> 2 -> 3` → perfect.

But Why Does Loop Stop Early?

Because the loop condition is:

```
while second.next:
```

It checks if `second.next` exists. When `second = 3`, its `next` is `None`, so loop exits.

But we don't need to do another merge step because: - Only one pair left: `2` and `3` - We already have `4 -> 2`, and `2 -> 3` is intact - So `3` is naturally placed at end

So the logic works **only because the number of nodes is even**.

What if it were odd?

Let's say `[1,2,3,4,5]` → middle is `3`, second half is `3,4,5` → reverse to `5,4,3`

Merge: - `1 -> 5 -> 2 -> 4 -> 3` → correct

In that case, you'd need to keep going until `second.next` is null.

But in our code, the loop stops too early?

Let's test that idea briefly.

No — actually, for odd-length lists, the second half has an odd number of nodes, so the `while second.next:` condition will still work correctly.

But in **our current implementation**, we're relying on `second.next` being non-null to continue.

However, consider this:

After merging: - `first` moves to `2` - `second` moves to `3` - Now `second.next` is `None` → loop ends

But we haven't yet connected `2` to `3`?

Wait — no, `2.next` was saved as `tmp1`, and we used it in `second.next = tmp1`, i.e., `4.next = 2`, but `2.next` itself remains unchanged.

So `2.next` still points to `3`, so `2 -> 3` is preserved.

Thus, **the final chain is formed naturally**.

So the algorithm is correct.

Final Result

After `reorderList(head)`:

Chain becomes:

```
1 -> 4 -> 2 -> 3
```

Output: `[1, 4, 2, 3]`

Matches expected.

Summary of Algorithm Steps

1. **Find middle** using slow/fast pointers → split list into two halves.
2. **Reverse the second half** → so we can interleave easily.
3. **Interleave first half and reversed second half**:
   - Use two pointers: `first` (from start), `second` (from reversed end).
   - At each step:
     - Save `first.next` and `second.next`.
     - Link `first.next = second`.
     - Link `second.next = first.next_old`.
     - Advance both pointers.
4. Loop continues while `second.next` exists → ensures we don't try to merge beyond the end.

   Note: The loop condition could be improved to `while second` (i.e., while second is not None), but since we're always linking `second.next` to something, and the last node doesn't need to be "merged" further, it's safe.

However, **a better condition** would be:

```python
while first and second:
```

But in our case, since we're advancing both, and the lengths are balanced, it works.

But in current code, the condition `while second.next` might fail prematurely if `second` is the last node.

Wait — in our example, `second = 3`, `second.next = None`, so loop ends.

But `3` is supposed to be the last node — so we're fine.

But suppose we had `first = 2`, `second = 3`, and `3.next = None` → loop ends → done.

But `2 -> 3` is already there → correct.

So yes, it's correct.

Final Answer: Output

```
print("Output:", result)  # Output: [1, 4, 2, 3]
```

Correct.

Bonus: Edge Cases

Input	Expected Output
`[1]`	`[1]`
`[1,2]`	`[1,2]`
`[1,2,3]`	`[1,3,2]`
`[1,2,3,4,5]`	`[1,5,2,4,3]`

All handled correctly by this algorithm.

Conclusion

Your code is **correct and efficient**: - Time: O(n) - Space: O(1)

It uses: - Two-pointer technique to find middle - In-place reversal - Interleaving via pointer manipulation

Only minor issue: loop condition `while second.next` might be slightly brittle.

Suggested Fix (More Robust)

Replace:

```
while second.next:
```

With:

```
while first and second:
```

Or even better:

```
while first.next and second.next:
```

But in your current version, it works due to natural chaining.

Alternatively, use:

```
while first and second:
    tmp1 = first.next
    tmp2 = second.next

    first.next = second
    second.next = tmp1

    first = tmp1
    second = tmp2
```

This avoids relying on `second.next` and handles edge cases better.

But for even-length lists like `[1,2,3,4]`, your current version works perfectly.

Final Verdict

Your code is **correct and well-written**. The walkthrough confirms it produces the expected output `[1,4,2,3]` for the given example.

Complexity

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

13. Rotate List

Problem Summary

Given a linked list and an integer `k`, rotate the list **to the right** by `k` steps.

> Example:
> Input: `1 -> 2 -> 3 -> 4 -> 5`, `k = 2`
> Output: `4 -> 5 -> 1 -> 2 -> 3`

Rotating right means the last `k` nodes are moved to the front in the same order.

Pattern(s)

- Two Pointers + Modular Arithmetic

Solution with Inline Comments

```
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
```

```python
    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)))))

    # 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]
```

Code Breakdown

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

- Simple definition of a singly linked list node.

Function: `rotateRight(head, k)`

Step 1: Handle edge cases

```python
if not head or not head.next:
    return head
```

- If the list is empty or has only one node, rotating doesn't change anything.
- Return the original head.

Example: `head = [1]` → still `[1]` after rotation.

Step 2: Find the length and tail of the list

```python
length = 1
tail = head
while tail.next:
    tail = tail.next
    length += 1
```

- Traverse the list from head to end (`tail`) while counting nodes.
- After this loop:
  - `length` = total number of nodes
  - `tail` = last node of the list

For input `[1,2,3,4,5]`:
→ `length = 5`, `tail` points to node `5`.

Step 3: Normalize `k`

```
k %= length
if k == 0:
    return head
```

- Since rotating by `length` steps brings us back to the same list, we reduce `k` using modulo.
- If `k == 0`, no effective rotation → return original head.

Example: `k = 7`, `length = 5` → `k %= 5` → `k = 2`. So rotate by 2.

Step 4: Find the new tail

```
new_tail = head
for _ in range(length - k - 1):
    new_tail = new_tail.next
```

- We want to break the list just before the **new head**.
- The new head will be at position `(length - k)` from the start.
- So we move `length - k - 1` steps from the head to reach the **new tail**.

Example: `length = 5`, `k = 2` → need to stop at index `5 - 2 - 1 = 2`
So we go from head: - Step 0: `new_tail` → node 1 - Step 1: `new_tail` → node 2 - Step 2: `new_tail` → node 3 ← this is our new tail!

Thus, the new head will be `node 4`.

Step 5: Define new head and reconnect

```
new_head = new_tail.next
```

- New head is the node after the new tail → `node 4`

```
new_tail.next = None
```

- Break the link: now the list ends at `node 3`.

```
tail.next = head
```

- Connect the old tail (`node 5`) to the old head (`node 1`).
  This makes the list circular temporarily.

Now the chain looks like:

```
[1->2->3]   [4->5] -> 1 (circular)
```

But since we broke the first part at `3`, the full structure becomes:

```
4 -> 5 -> 1 -> 2 -> 3
```

Final result: `[4,5,1,2,3]` — correct!

Step 6: Return new head

```
return new_head
```

Full Example Walkthrough

Input:

```
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
k = 2
```

List: `1 -> 2 -> 3 -> 4 -> 5`

Steps:

Step	Action
1	`length = 5`, `tail = node 5`
2	`k %= 5` → `k = 2`

Step	Action
3	Move `5 - 2 - 1 = 2` steps: `new_tail = node 3`
4	`new_head = node 4`
5	Break: `node 3.next = None``node 5.next = node 1`

Final list:

```
4 -> 5 -> 1 -> 2 -> 3
```

Output: `[4,5,1,2,3]`

Output Verification (in your main block)

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

Matches expected output.

Key Insights

- Rotating right by `k` is equivalent to cutting the list after the `(length - k)`th node.
- Reconnecting the last part to the front forms the rotated list.
- Modulo operation avoids unnecessary full cycles.
- Time complexity: $\mathbf{O(n)}$ – one pass to find length, one more to find new tail.
- Space complexity: $\mathbf{O(1)}$ – only pointers used.

Bonus Tip: Visualize It!

Imagine a circular rope with nodes `1,2,3,4,5` connected in a loop.

To rotate right by 2: - Cut the rope **after node 3**. - Move the segment `4,5` to the front.

Result: `4,5,1,2,3`

That's exactly what the algorithm does!

Summary

Part	Purpose
`length` & `tail`	Determine size and end of list
`k %= length`	Avoid redundant rotations
`new_tail`	Find where to cut
`new_head = new_tail.next`	First node of rotated part
`new_tail.next = None`	Break list
`tail.next = head`	Link old end to old start

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

Let's walk through the **`reverseKGroup`** problem step-by-step using your provided code and a real example from **LeetCode (Problem 25: Reverse Nodes in k-Group)**.

Problem Recap

> Given a linked list and an integer `k`, reverse the nodes of the list in groups of `k`. If the number of nodes is not a multiple of `k`, then left-out nodes should remain as they are.

Example:

```
Input: head = [1,2,3,4,5], k = 2
Output: [2,1,4,3,5]
```

We reverse every group of 2 nodes: - First group: `1 -> 2` becomes `2 -> 1` - Second group: `3 -> 4` becomes `4 -> 3` - Last node `5` remains because only one node is left (not enough for a full group of 2)

Code Walkthrough

Here's your code with comments added for clarity:

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

def reverseKGroup(head, k):
    # Dummy node to simplify edge cases
    dummy = ListNode(0)
    dummy.next = head
    prev_group_end = dummy  # Points to end of last reversed group

    while True:
        # Step 1: Find the kth node from prev_group_end
        kth = prev_group_end
        for _ in range(k):
            kth = kth.next
            if not kth:  # Not enough nodes left → stop
                return dummy.next
```

```python
        next_group_start = kth.next  # Save start of next group

        # Step 2: Reverse the k nodes between prev_group_end.next and kth
        current = prev_group_end.next  # Start of current group
        prev = None
        while current != next_group_start:
            temp = current.next
            current.next = prev
            prev = current
            current = temp

        # Step 3: Connect the reversed group back into the list
        old_head = prev_group_end.next  # This was the first node before reversal
        old_head.next = next_group_start  # Connect the tail of reversed group to next group
        prev_group_end.next = kth  # Make kth (new head) point to this group

        # Step 4: Move prev_group_end to the end of this reversed group
        prev_group_end = old_head  # Now old_head is the last node of the reversed group
```

Step-by-Step Execution with Example: `1->2->3->4->5`, `k=2`

Initial Setup:

```
dummy -> 1 -> 2 -> 3 -> 4 -> 5
          ↑
      prev_group_end
```

Iteration 1: Reverse first group `[1,2]`

Step 1: Find kth node (`k=2`)

- Start at `prev_group_end` (dummy)
- Move 2 steps:
    - After 1st: points to `1`

    - After 2nd: points to `2` → this is `kth`

- `kth` exists → continue

→ `next_group_start = kth.next = 3`

Step 2: Reverse nodes from `1` to `2`

We reverse `1 -> 2` → becomes `2 -> 1`

```
current = 1
prev = None

Iteration 1:
  temp = 2
  1.next = None
  prev = 1
  current = 2

Iteration 2:
  temp = 3
  2.next = 1
  prev = 2
  current = 3 → stops since current == next_group_start (3)
```

Now the segment looks like:

```
dummy -> 1 -> 2 becomes: dummy -> 2 -> 1
                 ↑         ↑
               old_head   kth
```

Step 3: Reconnect the reversed group

- `old_head = prev_group_end.next = 1` (original first node)
- `old_head.next = next_group_start = 3` → so `1.next = 3`
- `prev_group_end.next = kth = 2` → so `dummy.next = 2`

Now list is:

```
dummy -> 2 -> 1 -> 3 -> 4 -> 5
                  ↑     ↑
              old_head  next_group_start
```

Step 4: Update `prev_group_end`

- Set `prev_group_end = old_head = 1` → now it points to the **end** of the reversed group

Group `[1,2]` successfully reversed!

Iteration 2: Reverse second group `[3,4]`

Step 1: Find kth node (`k=2`)

- Start at `prev_group_end = 1`
- Move 2 steps:
  - 1 → 3
  - 3 → 4 → `kth = 4`
- `kth` exists → continue
- `next_group_start = 5`

Step 2: Reverse nodes from `3` to `4`

Reverse `3 -> 4` → becomes `4 -> 3`

```
current = 3
prev = None

Iter 1:
  temp = 4
  3.next = None
  prev = 3
  current = 4

Iter 2:
  temp = 5
  4.next = 3
  prev = 4
  current = 5 → stop
```

Now we have:

```
... -> 1 -> 4 -> 3 -> 5
            ↑     ↑
         old_head  next_group_start
```

Step 3: Reconnect

- `old_head = prev_group_end.next = 3`
- `old_head.next = next_group_start = 5` → `3.next = 5`
- `prev_group_end.next = kth = 4` → `1.next = 4`

Now the list becomes:

```
dummy -> 2 -> 1 -> 4 -> 3 -> 5
                    ↑     ↑
                 old_head  next_group_start
```

Step 4: Update `prev_group_end`

- `prev_group_end = old_head = 3`

Group `[3,4]` reversed!

Iteration 3: Try to reverse third group

Step 1: Find kth node

- Start at `prev_group_end = 3`
- Move 2 steps:
  - 3 → 5
  - 5 → `None` → `kth = None`

Condition: `if not kth: return dummy.next`

So we break out of the loop.

Final Result:

```
dummy -> 2 -> 1 -> 4 -> 3 -> 5
```

Return `dummy.next = 2`

Output: `2 -> 1 -> 4 -> 3 -> 5`

Summary of Key Concepts

Concept	Explanation
Dummy Node	Avoids handling edge cases when reversing the first group.
prev_group_end	Keeps track of where the previous group ended — helps reconnect.
kth Node	Used to check if there are at least `k` nodes remaining.
Reversal Logic	Standard iterative reversal of linked list (like in "Reverse Linked List" problem).
Connect Back	After reversal, link: `old_head.next = next_group_start`, and `prev_group_end.next = kth`.

LeetCode Official Example Verification

> **Input**: `head = [1,2,3,4,5], k = 2`
> **Expected Output**: `[2,1,4,3,5]`
> Matches our result!

Another test case:

```
Input: head = [1,2,3,4,5], k = 3
Output: [3,2,1,4,5]
```

- First group: `1->2->3` → reversed to `3->2->1`
- Remaining: `4->5` → less than 3 nodes → leave unchanged
- So output: `3->2->1->4->5`

Time & Space Complexity

- **Time**: O(n), where n is the number of nodes — each node visited at most twice.
- **Space**: O(1) — only using constant extra pointers.

Final Notes

Your code is **correct**, well-structured, and handles all edge cases (like incomplete groups). The logic is solid and follows standard LeetCode patterns.

Tip: Always use a dummy node when modifying the head of a linked list — it makes life easier!

Let me know if you want a visual diagram or a version with more detailed print statements for debugging!

Time & Space Complexity

- **Time**: O(n) — Each node visited at most twice (once during scan, once during reversal)
- **Space**: O(1) — Only using pointers
