Q: Linked List

1. Problem Understanding

- A Linked List is a linear data structure where elements (called nodes) are stored in separate memory locations.
- Each node is connected using a reference (pointer) to the next node.
- Unlike arrays, elements in a linked list are not stored in contiguous memory.
- It solves the problem of fragmented memory by dynamically allocating nodes.
- **Intuition Behind the Concept**
  - Imagine a chain of people: each person holds their own data and the hand of the next person.
  - This allows flexibility â€” people (nodes) can stand anywhere, yet still stay connected.
  - The chain starts at the head â€” the only way to access everyone else.
- **Node Structure**

```
class Node {
    int data;      // stores data
    Node next;     // stores address of next node
}
```

  - Each node = data + reference to next node.
  - The linked list = collection of connected nodes.
  - Size of linked list = number of nodes.
- **Core Terms**
  - Head â†’ First node of the linked list. Access point for the entire list.
  - Tail â†’ Last node, whose next is null (or head, in circular lists).
  - Current â†’ A temporary pointer used to traverse the list safely.
  - Size â†’ Total number of nodes in the list.
  - Always remember:
    - We never move the head directly.
    - Instead, create a pointer like Node current = head; and move it.
- **Types of Linked Lists**
  - Singly Linked List (SLL)
    - Each node points to the next node only.
    - Can move in one direction (forward).
    - The last nodeâ€™s next pointer is null.
  - Doubly Linked List (DLL)

        - Each node contains both next and previous references.
        - Can move in both directions (forward and backward).
        - Headâ€™s previous is null; tailâ€™s next is null.
    - Circular Linked List (CLL)
        - The last node points back to the head.
        - No node has a null reference.
        - Traversal loops continuously.

- **Why Use Linked Lists**

    - Supports dynamic memory allocation.
    - Insertion and deletion are efficient (no shifting like arrays).
    - Useful when frequent insertions/deletions are needed.
    - Can grow or shrink at runtime.

- **Strategy to Solve Linked List Questions**

    - Always draw a general diagram of 4â€“5 nodes.
    - Dry run pointer movement step by step.
    - Always handle two edge cases:
        - Empty list â†’ if (head == null)
        - Single node â†’ if (head.next == null)
    - Use separate pointers (current, prev, next) for clarity.
    - Preserve head to maintain access to the list.

---

2. Edge Cases

- If list is empty â†’ head == null.
- If list has one node â†’ head.next == null.
- Always check for null before accessing next or prev.
- Do not move the head; always use a separate pointer (current).

---

3. Approaches

Approach 1: Traversal

Java Code:

```
Singly Linked List

Node current = head;
while (current != null) {
    System.out.print(current.data + " ");
    current = current.next;
}


Doubly Linked List
```

```
Node current = head;
while (current != null) {
    System.out.print(current.data + " ");
    current = current.next;
}


Circular Linked List

Node current = head;
if (head != null) {
    do {
        System.out.print(current.data + " ");
        current = current.next;
    } while (current != head);
}
```

Approach 2: Insertion

Java Code:

```
Singly Linked List

Insert at beginning:

Node newNode = new Node(data);
newNode.next = head;
head = newNode;


Insert at end:

Node newNode = new Node(data);
if (head == null) { head = newNode; return; }
Node current = head;
while (current.next != null)
    current = current.next;
current.next = newNode;


Insert at specific position:

Node current = head;
for (int i = 1; i < pos - 1 && current != null; i++)
    current = current.next;
newNode.next = current.next;
current.next = newNode;


Doubly Linked List
```

```
Insert at beginning:

Node newNode = new Node(data);
newNode.next = head;
if (head != null) head.prev = newNode;
head = newNode;


Insert at end:

Node newNode = new Node(data);
Node current = head;
while (current.next != null)
    current = current.next;
current.next = newNode;
newNode.prev = current;


Insert at specific position:

Node current = head;
for (int i = 1; i < pos - 1; i++)
    current = current.next;
newNode.next = current.next;
if (current.next != null) current.next.prev = newNode;
current.next = newNode;
newNode.prev = current;


Circular Linked List

Insert at beginning:

Node newNode = new Node(data);
if (head == null) {
    head = newNode;
    head.next = head;
} else {
    Node temp = head;
    while (temp.next != head)
        temp = temp.next;
    temp.next = newNode;
    newNode.next = head;
    head = newNode;
}


Insert at end:

Node newNode = new Node(data);
if (head == null) {
    head = newNode;
    head.next = head;
```

```
} else {
    Node temp = head;
    while (temp.next != head)
        temp = temp.next;
    temp.next = newNode;
    newNode.next = head;
}
```

Approach 3: Deletion

Java Code:

```
Singly Linked List

Delete at beginning:

if (head == null) return;
head = head.next;


Delete at end:

if (head == null || head.next == null) { head = null; return; }
Node current = head;
while (current.next.next != null)
    current = current.next;
current.next = null;


Delete by value:

Node current = head, prev = null;
if (current != null && current.data == key) { head = current.next; return; }
while (current != null && current.data != key) {
    prev = current;
    current = current.next;
}
if (current == null) return;
prev.next = current.next;


Doubly Linked List

Delete at beginning:

if (head == null) return;
head = head.next;
if (head != null) head.prev = null;


Delete at end:
```

```
if (head == null) return;
Node current = head;
while (current.next != null)
    current = current.next;
if (current.prev != null)
    current.prev.next = null;
else head = null;


Delete by value:

Node current = head;
while (current != null && current.data != key)
    current = current.next;
if (current == null) return;
if (current.prev != null) current.prev.next = current.next;
else head = current.next;
if (current.next != null) current.next.prev = current.prev;


Circular Linked List

Delete head node:

if (head == null) return;
if (head.next == head) { head = null; return; }
Node temp = head;
while (temp.next != head)
    temp = temp.next;
temp.next = head.next;
head = head.next;
```

Approach 4: Interview Patterns on Linked List

Java Code:

```
1ï¸âƒ£ Reverse a Linked List

Iterative Approach

Node prev = null, current = head, next = null;
while (current != null) {
    next = current.next;
    current.next = prev;
    prev = current;
    current = next;
}
head = prev;
```

```
Intuition: Reverse the link direction step-by-step.
Complexity: Time O(N), Space O(1).

2ï¸âƒ£ Find Middle Element

Two-pointer approach

Node slow = head, fast = head;
while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next.next;
}
System.out.println("Middle Node: " + slow.data);


Intuition: Fast moves twice as quickly as slow; when fast reaches the end, slow is
at the middle.
Complexity: Time O(N), Space O(1).

3ï¸âƒ£ Detect Loop in Linked List (Floydâ€™s Cycle Detection)
Node slow = head, fast = head;
while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next.next;
    if (slow == fast) {
        System.out.println("Loop detected");
        return;
    }
}
System.out.println("No loop");


Intuition: If a loop exists, slow and fast will eventually meet.
Complexity: Time O(N), Space O(1).

4ï¸âƒ£ Remove Loop in Linked List
Node slow = head, fast = head;
boolean loop = false;
while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next.next;
    if (slow == fast) { loop = true; break; }
}
if (loop) {
    slow = head;
    while (slow.next != fast.next) {
        slow = slow.next;
        fast = fast.next;
    }
    fast.next = null;
}


Intuition: Reset one pointer to head; move both one step until they meet at loop
```

```
start, then break the loop.

5ï¸âƒ£ Remove Duplicates (from Sorted Linked List)
Node current = head;
while (current != null && current.next != null) {
    if (current.data == current.next.data)
        current.next = current.next.next;
    else
        current = current.next;
}


Intuition: Skip nodes that have the same data consecutively.
Complexity: Time O(N), Space O(1).

6ï¸âƒ£ Find Nth Node from End

Two-pointer approach

Node first = head, second = head;
for (int i = 0; i < n; i++) {
    if (first == null) return;
    first = first.next;
}
while (first != null) {
    first = first.next;
    second = second.next;
}
System.out.println("Nth node from end: " + second.data);


Intuition: Maintain a gap of n between two pointers.
Complexity: Time O(N), Space O(1).
```

Complexity (Time & Space):

- â±ï¸ Time Complexity Summary
  - Traversal â†’ O(N)
  - Insertion (head) â†’ O(1)
  - Insertion (end) â†’ O(N)
  - Deletion (head) â†’ O(1)
  - Deletion (end) â†’ O(N)
  - Search â†’ O(N)
  - Reversal â†’ O(N)
  - Loop detection â†’ O(N)
  - Middle element â†’ O(N)
- ðŸ'¾ Space Complexity
  - Most operations use only pointer variables â†’ O(1)
  - The list itself uses O(N) memory for N nodes.

---

4. Tips & Observations

- Visualize every pointer change.
- Always protect head.
- Watch out for NullPointerException.
- Handle empty and single-node lists.
- Use the two-pointer technique for efficient traversal-based questions.
- Draw, dry run, and code â€” in that order.
- **Summary**
  - Linked List = dynamic, flexible, pointer-based structure.
  - Node = data + next (and sometimes prev).
  - Head = entry point to entire list.
  - Use current pointer for traversal.
  - Always check for nulls and edge cases.
  - Know the main types:
    - Singly Linked List â†’ forward only
    - Doubly Linked List â†’ both directions
    - Circular Linked List â†’ continuous loop
  - Master key patterns:
    - Reversal
    - Middle element
    - Loop detection & removal
    - Remove duplicates
    - Nth node from end

Q102: Delete Kth Element of a Doubly Linked List

1. Problem Understanding

- You are given the head of a doubly linked list (DLL) and an integer k.
- Your task is to delete the node at the kth position (1-based index) and return the head of the modified DLL.

---

2. Constraints

- 1 <= n <= 100 (where n = number of nodes)
- 0 <= Node.val <= 100
- 1 <= k <= n

---

3. Edge Cases

- k = 1 â†' delete the head node (head must be updated).
- k = n â†' delete the last node (tail must be updated).
- List has only one node â†' after deletion, list becomes empty (head = null).
- Middle node deletion â†' need to adjust both prev and next links.

---

4. Examples

```
Example 1

Input:
head -> 2 <-> 5 <-> 7 <-> 9, k = 2
Output:
head -> 2 <-> 7 <-> 9
Explanation: Node with value 5 (2nd node) is deleted.

Example 2

Input:
head -> 2 <-> 5 <-> 7, k = 1
Output:
head -> 5 <-> 7
Explanation: Node with value 2 is deleted (head updated).

Example 3

Input:
head -> 2 <-> 5 <-> 7, k = 3
Output:
head -> 2 <-> 5
Explanation: Node with value 7 (tail) is deleted.
```

---

5. Approaches

Approach 1: Iterative Traversal

Idea:

- Traverse from the head to reach the kth node.
- Once found:
  - If it's the head â†' update head = head.next.
  - If it has a previous node â†' update prev.next.
  - If it has a next node â†' update next.prev.
- Return the new head.

Steps:

- If head == null, return null.
- Traverse the list until count == k.
- When you find the node:
  - If it's head â†' move head to head.next.
  - If it has a prev â†' link prev.next = curr.next.
  - If it has a next â†' link next.prev = curr.prev.
- Return the updated head.

Java Code:

```
Node deleteNode(Node head, int k) {
    if (head == null) return null;

    Node curr = head;
    int count = 1;

    // Traverse to kth node
    while (curr != null && count < k) {
        curr = curr.next;
        count++;
    }

    // If node to delete doesn't exist
    if (curr == null) return head;

    // If deleting head node
    if (curr.prev == null) {
        head = curr.next;
        if (head != null) head.prev = null;
    }
    // If deleting middle or last node
    else {
        curr.prev.next = curr.next;
        if (curr.next != null) {
            curr.next.prev = curr.prev;
        }
    }

    return head;
}

ðŸŒ² Recursion Tree (Conceptual)

This problem is primarily iterative, but conceptually:
If you imagined recursion, it would traverse nodes until k == 1, then delete and
backtrack.
For head -> 2 <-> 5 <-> 7, k = 2:

delete(2, k=2)
 â””â”€â”€ delete(5, k=1) â†’ removes node 5
Backtrack â†’ fix links
Result: 2 <-> 7
```

ðŸ' Intuition Behind the Approach:

- In a doubly linked list, every node maintains pointers to both its previous and next nodes.
- To delete the kth node:
- Traverse to the kth node.
- Update its neighborsâ€™ pointers:
    - prev.next = next
    - next.prev = prev
- Handle special cases when the node is head or tail.

Complexity (Time & Space):

- â±ï¸ Time Complexity
    - Traversal to kth node â†' O(k)
    - Pointer updates â†' O(1)
    - âž¡ï¸ Total: O(k)
- ðŸ'¾ Space Complexity
    - No extra memory used â†' O(1)

---

6. Justification / Proof of Optimality

- Deletion in DLL is efficient since both directions are accessible.
- No extra data structures are needed.
- Updates are O(1) after reaching the node.

---

7. Variants / Follow-Ups

- Delete all occurrences of a given value.
- Delete last node or middle node dynamically.
- Delete node when pointer to the node (not head) is given.

---

8. Tips & Observations

- Always handle head and tail cases separately.
- Donâ€™t forget to update both prev and next pointers.
- In interviews, emphasize constant space and clean pointer handling.

Q103: Insert Before Given Node in a Doubly Linked List

---

1. Problem Understanding

- You are given a reference to a node in a Doubly Linked List (DLL) and an integer X.
- You need to insert a new node with value X before the given node, while maintaining the DLLâ€™s bidirectional linkage integrity.
- You are not given the head of the list, and it is guaranteed that the given node is not the head of the list.

---

2. Constraints

- 2 <= n <= 100 (where n = number of nodes in the DLL)
- 0 <= ListNode.val <= 100
- 0 <= X <= 100
- The given node will always exist in the DLL.
- The given node will never be the head (so node.prev != null).

---

3. Edge Cases

- The given node is the tail â†’ still valid, should insert before it.
- DLL has only two nodes â†’ ensure correct re-linking of both sides.
- Multiple nodes have the same value â†’ insertion depends on the node reference, not the value.

---

4. Examples

```
Example 1

Input:
head -> 1 <-> 2 <-> 6, node = 2, X = 7
Output:
head -> 1 <-> 7 <-> 2 <-> 6
Explanation:
The node 7 is inserted before node 2.

Example 2

Input:
head -> 7 <-> 5 <-> 5, node = 5 (last one), X = 10
Output:
head -> 7 <-> 5 <-> 10 <-> 5
Explanation:
The second 5 (the last node) is referenced; new node 10 is inserted before it.

Example 3

Input:
head -> 7 <-> 6 <-> 5, node = 5, X = 10
Output:
head -> 7 <-> 6 <-> 10 <-> 5
Explanation:
Node 10 inserted before node 5.
```

5. Approaches

Approach 1: Explicit Pointer Linking

Idea:

- We already have a reference to the node.
- To insert before it, we just need to connect the new node between node.prev and node.
- This requires adjusting four pointers.

Steps:

- Store prevNode = node.prev.
- Create the new node.
- Set the links:
  - newNode.prev = prevNode
  - newNode.next = node
- Adjust the existing neighbors:
  - prevNode.next = newNode
  - node.prev = newNode

Java Code:

```
class Solution {
    public void insertBeforeGivenNode(ListNode node, int X) {
        if (node == null || node.prev == null) return; // safety check

        ListNode newNode = new ListNode(X);
        ListNode prevNode = node.prev;

        newNode.prev = prevNode;
        newNode.next = node;
        prevNode.next = newNode;
        node.prev = newNode;
    }
}
```

ðŸ' Intuition Behind the Approach:

- In a DLL, each node maintains pointers in both directions.
- To insert before a node, we only need to reconnect four links around the insertion point â€" no traversal or head access is required.
- This makes the operation constant time (O(1)) and very efficient.

Complexity (Time & Space):

- Time Complexity: O(1)

- Space Complexity: O(1)

Approach 2: Using Constructor for Cleaner Code

Idea:

- If the ListNode class provides a constructor with (int val, ListNode next, ListNode prev),
- we can simplify the code by assigning links directly during node creation.

Steps:

- Store prevNode = node.prev.
- Create a new node using the parameterized constructor:
  - ListNode newNode = new ListNode(X, node, prevNode);
- Update only the neighboring pointers:
  - prevNode.next = newNode
  - node.prev = newNode

Java Code:

```
class Solution {
    public void insertBeforeGivenNode(ListNode node, int X) {
        ListNode prev = node.prev;
        ListNode newNode = new ListNode(X, node, prev);
        prev.next = newNode;
        node.prev = newNode;
    }
}
```

ðŸ' Intuition Behind the Approach:

- This is a more compact version of the explicit pointer approach.
- The constructor sets up both links immediately, reducing boilerplate while maintaining correctness.
- Itâ€™s ideal for clean, production-level code when the constructor is available.

Complexity (Time & Space):

- Time Complexity: O(1)
- Space Complexity: O(1)

---

6. Justification / Proof of Optimality

- Both approaches perform only local pointer updates â€” no traversal or data shifting.
- The correctness follows from maintaining DLL bidirectional linkage (prev and next) consistently for all involved nodes.

---

7. Variants / Follow-Ups

- Insert After Given Node in DLL â†’ Similar logic but swap pointer direction.
- Insert Before Head (with head reference) â†’ Special handling since node.prev == null.
- Insert at Given Position (1-based index) â†’ Requires traversal to that position first.

---

8. Tips & Observations

- Always ensure both prev and next links are updated symmetrically.
- Avoid accessing node.prev without checking null â€" unless guaranteed not to be head.
- Prefer constructor-based shorthand when supported, but show pointer logic explicitly in interviews.
- All insertions in DLL can be done in constant time when node reference is available.

Q104: Add Two Numbers in a Linked List

1. Problem Understanding

- We are given two non-empty singly linked lists l1 and l2, where each node represents a single digit of a number.
- The digits are stored in reverse order â€" meaning the 1â€™s digit is at the head of the list.
- We must add the two numbers and return the sum as a new linked list, also in reverse order.
- Example:
- l1 = [5 â†’ 4] represents 45
- l2 = [4] represents 4
- Sum = 49 â†’ Output: [9 â†’ 4]

---

2. Constraints

- 1 <= Number of nodes in each list <= 100
- 0 <= value of each node <= 9
- Lists have no leading zeros (except when the number itself is 0).

---

3. Edge Cases

- One list is longer than the other.
- Carry remains after the final addition.
- One list could represent zero (e.g., [0]).
- Both lists contain only one node.

---

4. Examples

```
Example 1:
Input:
l1 = [5 â†’ 4], l2 = [4]
```

```
Output:
[9 â†’ 4]
Explanation: 45 + 4 = 49.

Example 2:
Input:
l1 = [4 â†’ 5 â†’ 6], l2 = [1 â†’ 2 â†’ 3]
Output:
[5 â†’ 7 â†’ 9]
Explanation: 654 + 321 = 975.

Example 3:
Input:
l1 = [1], l2 = [8 â†’ 7]
Output:
[9 â†’ 7]
Explanation: 1 + 78 = 79.
```

---

5. Approaches

Approach 1: Iterative Addition with Carry (Optimal)

Idea:

- Perform addition like manual digit-by-digit addition:
  - Traverse both linked lists simultaneously.
  - Sum digits + carry.
  - Create a new node for each resulting digit.
  - Continue until both lists are processed and carry is zero.

Steps:

- Initialize a dummy node and a carry = 0.
- Traverse both lists until both are null:
  - Compute sum = (l1.val if exists) + (l2.val if exists) + carry.
  - Create a new node with sum % 10.
  - Update carry = sum / 10.
- If carry > 0 after traversal, create one more node.
- Return dummy.next.

Java Code:

```java
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(-1);
        ListNode curr = dummy;
        int carry = 0;

        while (l1 != null || l2 != null) {
```

```
            int sum = carry;
            if (l1 != null) {
                sum += l1.val;
                l1 = l1.next;
            }
            if (l2 != null) {
                sum += l2.val;
                l2 = l2.next;
            }

            carry = sum / 10;
            curr.next = new ListNode(sum % 10);
            curr = curr.next;
        }

        if (carry > 0) {
            curr.next = new ListNode(carry);
        }

        return dummy.next;
    }
}
```

ðŸ' Intuition Behind the Approach:

- The process mirrors normal addition from right to left â€" except the digits are already stored in reverse.
- Each step adds the corresponding digits and carries over the overflow (like adding column-wise).

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(max(N, M))
  - (where N and M are lengths of l1 and l2)
- ðŸ'¾ Space Complexity
  - O(max(N, M)) (for the new linked list)

Approach 2: Recursive Approach

Idea:

- Use recursion to simulate the digit-by-digit addition from the start of both lists.
- Each recursive call handles one pair of digits and the carry.

Steps:

- Base case: If both lists are null and carry is 0, return null.
- Sum up current nodesâ€™ values + carry.
- Create a node with sum % 10.
- Recurse to process the next nodes and pass the carry (sum / 10).
- Link the newly created node to the result of recursion.

Java Code:

```
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        return addHelper(l1, l2, 0);
    }

    private ListNode addHelper(ListNode l1, ListNode l2, int carry) {
        if (l1 == null && l2 == null && carry == 0) return null;

        int sum = carry;
        if (l1 != null) sum += l1.val;
        if (l2 != null) sum += l2.val;

        ListNode node = new ListNode(sum % 10);
        node.next = addHelper(
            (l1 != null ? l1.next : null),
            (l2 != null ? l2.next : null),
            sum / 10
        );
        return node;
    }
}
```

ðŸ’ Intuition Behind the Approach:

- Recursion naturally handles the propagation of carry and traversal of both lists.
- Each call represents the computation of a single digit and delegates the remaining work to the next call.
- Itâ€™s a clean, functional approach â€” just like stack-based addition.

Complexity (Time & Space):

- â±ï¸ Time Complexity
- O(max(N, M))
- ðŸ’¾ Space Complexity
- O(max(N, M)) (recursive call stack + new list nodes)

---

6. Justification / Proof of Optimality

- Both approaches perform exactly one traversal over each list, ensuring linear time and minimal space.
- The iterative version is slightly more memory-efficient, but the recursive version is more elegant.

---

7. Variants / Follow-Ups

- Digits stored in forward order: Use stacks or reverse the lists first.
- Addition with large numbers in strings: Convert strings to digits and simulate addition.
- Add K numbers represented as linked lists: Use repeated pairwise addition or a priority queue.

---

8. Tips & Observations

- Always use a dummy node to simplify result list creation.
- Keep careful track of carry propagation.
- Recursive solutions are elegant but can cause stack overflow if lists are extremely long.
- Iterative is preferred for production; recursion is good for conceptual clarity.

Q105: Segregate Odd and Even Nodes in Linked List

1. Problem Understanding

- You are given the head of a singly linked list.
- You need to group all nodes at odd indices first, followed by all nodes at even indices â€” maintaining their original relative order.
- Note:
  - The index starts from 1 (i.e., the head is the 1st node).
- You must rearrange nodes in place and return the new head of the reordered list.

---

2. Constraints

- 1 <= Number of nodes <= 100
- Node values can be any integer.
- The relative order within odd and even groups must remain the same.
- Must use O(1) extra space (rearrange in-place).

---

3. Edge Cases

- Empty list (head == null)
- List with only one node
- List with only two nodes
- All odd or all even length lists (no structural issue should occur)

---

4. Examples

```
Example 1:
Input:
head -> 1 -> 2 -> 3 -> 4 -> 5
Output:
head -> 1 -> 3 -> 5 -> 2 -> 4
Explanation: Odd index nodes are [1, 3, 5], even index nodes are [2, 4].
```

```
Example 2:
Input:
head -> 4 -> 3 -> 2 -> 1
Output:
head -> 4 -> 2 -> 3 -> 1
Explanation: Odd index nodes â†' [4, 2], even index nodes â†' [3, 1].
```

---

5. Approaches

Approach 1: Two Pointer (In-place Rearrangement â€" Optimal)

Idea:

- Use two pointers â€" one for odd nodes and one for even nodes.
- We can separate the linked list into two sublists:
  - One containing odd-indexed nodes.
  - One containing even-indexed nodes.
- Then, simply link the end of the odd list to the head of the even list.

Steps:

- Handle base cases: if head == null or head.next == null, return head.
- Initialize:
  - odd = head
  - even = head.next
  - evenHead = even (store head of even list)
- While both odd.next and even.next exist:
  - Connect odd.next = even.next
  - Move odd = odd.next
  - Connect even.next = odd.next
  - Move even = even.next
- After loop ends, link odd.next = evenHead.
- Return the modified head.

Java Code:

```
class Solution {
    public ListNode oddEvenList(ListNode head) {
        if (head == null || head.next == null) return head;

        ListNode odd = head;
        ListNode even = head.next;
        ListNode evenHead = even;

        while (even != null && even.next != null) {
            odd.next = even.next;
            odd = odd.next;
```

```
            even.next = odd.next;
            even = even.next;
        }

        odd.next = evenHead;
        return head;
    }
}
```

ðŸ' Intuition Behind the Approach:

- Think of this as unzipping the linked list into two separate sequences:
  - All nodes in odd positions
  - All nodes in even positions
- We keep track of both sequences while traversing once and reconnect them at the end.
- This maintains order and uses no extra space â€" just pointer manipulation.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(N) â€" single traversal of the linked list
- ðŸ'¾ Space Complexity
  - O(1) â€" no extra space used, in-place rearrangement

Approach 2: Construct New Lists (Using Two Dummy Nodes)

Idea:

- Instead of rearranging in place, we can create two new lists:
  - One for odd nodes
  - One for even nodes
- Then connect the two lists at the end and return the new head.

Steps:

- Create two dummy nodes â€" oddDummy and evenDummy.
- Traverse the original list, keeping a position counter i.
  - If i is odd â†' add node to odd list.
  - If i is even â†' add node to even list.
- After traversal, link the end of the odd list to the start of the even list.
- Set the end of even listâ€™s next = null.
- Return oddDummy.next

Java Code:

```
class Solution {
    public ListNode oddEvenList(ListNode head) {
        if (head == null) return null;

        ListNode oddDummy = new ListNode(-1);
```

```
        ListNode evenDummy = new ListNode(-1);
        ListNode odd = oddDummy, even = evenDummy;
        int index = 1;

        while (head != null) {
            if (index % 2 != 0) {
                odd.next = head;
                odd = odd.next;
            } else {
                even.next = head;
                even = even.next;
            }
            head = head.next;
            index++;
        }

        even.next = null;     // terminate even list
        odd.next = evenDummy.next; // connect lists

        return oddDummy.next;
    }
}
```

ðŸ' Intuition Behind the Approach:

- We explicitly build two separate linked lists based on node indices, then merge them.
- Itâ€™s simpler to understand but uses extra references for dummy nodes (still O(1) auxiliary space if pointer reuse is allowed).

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(N) â€" each node visited once
- ðŸ'¾ Space Complexity
  - O(1) â€" constant extra space (if we reuse original nodes)

---

6. Justification / Proof of Optimality

- Both approaches achieve linear time and constant space.
- However, the two-pointer in-place version (Approach 1) is preferred â€" it is more efficient and elegant without needing dummy nodes.

---

7. Variants / Follow-Ups

- Segregate even and odd values (instead of indices).
- Group nodes by multiple conditions (e.g., multiples of 3).
- Rearrange alternatingly (odd, even, odd, even).

---

8. Tips & Observations

- Donâ€™t confuse index parity with node value parity.
- Always store the head of the even list before rearranging â€” otherwise, it gets lost.
- Maintain relative ordering within groups.
- Avoid creating new nodes unnecessarily; re-link existing ones.

Q106: Merge Two Sorted Linked Lists

1. Problem Understanding

- You are given two sorted singly linked lists.
- Your task is to merge them into one sorted linked list by rearranging pointers (not creating new nodes).
- Return the head of the merged list.
- Important:
  - Both lists are sorted in non-decreasing order.
  - Output must also be sorted.
  - Lists may be empty.

---

2. Constraints

- Number of nodes: 0 â‰¤ n, m â‰¤ 50
- Node value: -100 â‰¤ val â‰¤ 100
- Both input lists are sorted.
- Linked lists are singly linked.

---

3. Edge Cases

- One list empty â†’ return the other list
- Both empty â†’ return null
- Duplicate values (e.g., 1â†’2â†’4 and 1â†’1â†’3)
- All elements of one list smaller than the other
- Lists of different lengths
- Negative values

---

4. Examples

```
Example 1

Input:

1 2 4
1 3 4
```

```
Output:

1 1 2 3 4 4

Example 2

Input:

1 5 9
1 3 4


Output:

1 1 3 4 5 9
```

---

5. Approaches

Approach 1: Iterative Merge (Optimal, In-Place)

Idea:

- Use a dummy node.
- Keep pointers cx and cy on both lists.
- Attach the smaller node to the result and move the pointer.
- No new nodes are created.

Steps:

- Create a dummy node to simplify linking.
- Maintain tail pointer â€” end of merged list.
- Compare cx.data and cy.data:
- Attach the smaller one to tail.next
- Move that pointer
- When one list is exhausted, attach the remaining list.
- Return dummy.next.

Java Code:

```java
static Node merge(Node x, Node y) {
    Node dummy = new Node(-1);
    Node tail = dummy;

    while (x != null && y != null) {
        if (x.data <= y.data) {
            tail.next = x;
            x = x.next;
```

```
            } else {
                tail.next = y;
                y = y.next;
            }
            tail = tail.next;
        }

        if (x != null) tail.next = x;
        if (y != null) tail.next = y;

        return dummy.next;
    }
```

ðŸ′ Intuition Behind the Approach:

- We treat merging like merging two sorted arrays:
- always pick the smaller front element.
- Linked lists allow us to do this without copying values, we only move pointers.
- This ensures:
    - Correct ordering
    - O(1) extra space
    - Minimal pointer operations

Complexity (Time & Space):

- Time: O(n + m)
- Space: O(1) (in-place merge)

Approach 2: Recursive Merge (Elegant but Uses O(n+m) Stack Space)

Idea:

- Pick the smaller head between the two lists.
- That node becomes the head of the merged list.
- Recursively merge the rest.

Steps:

- Base cases: if one list is null â†′ return the other
- Compare heads
- The smaller node becomes head of merged list
- Recursively merge remaining part

Java Code:

```
static Node merge(Node a, Node b) {
    if (a == null) return b;
    if (b == null) return a;

    if (a.data <= b.data) {
```

```
            a.next = merge(a.next, b);
            return a;
        } else {
            b.next = merge(a, b.next);
            return b;
        }
    }
```

ðŸ' Intuition Behind the Approach:

- The smaller head must always come first.
- Recursively apply the same logic on the remaining lists.
- Result naturally builds up as recursion unwinds.

Complexity (Time & Space):

- Time: O(n + m)
- Space: O(n + m) due to recursion stack

---

6. Justification / Proof of Optimality

- The in-place iterative merge is the best solution because:
- It processes each node exactly once â†' O(n+m)
- No extra nodes â†' O(1) space
- Avoids recursion stack overhead
- Preserves the original nodes
- This matches the optimal behavior expected for merging sorted linked lists.

---

7. Variants / Follow-Ups

- Merge k sorted linked lists â†' use a min-heap (O(N log k))
- Merge arrays instead of linked lists
- Merge two descending sorted lists
- Merge lists where duplicates should be removed
- Merge circular linked lists
- Merge based on custom comparator (e.g., absolute value)

---

8. Tips & Observations

- Always use a dummy node to simplify code.
- Avoid creating new nodes unless required by the question.
- Recursion is elegant but not memory-efficient.
- When merging k lists, use:
  - Min-heap, or
  - Divide & Conquer (merge sort style)

Q107: Print in Reverse

1. Problem Understanding

- Given the head of a singly linked list, reverse the list and return the new head.
- Example:
- Input: 1 â†' 2 â†' 3 â†' 4 â†' 5
- Output: 5 â†' 4 â†' 3 â†' 2 â†' 1
- Important:
  - The list must be modified in-place
  - No new nodes should be created
  - Return the new head of the reversed list

---

2. Constraints

- Number of nodes: 1 â‰¤ n â‰¤ 5000
- Node values are integers
- The list is singly linked

---

3. Edge Cases

- Single node list â†' return same node
- Empty list â†' return null
- Already reversed pattern doesnâ€™t matter
- Large list (iterative preferred over recursion due to stack depth)

---

4. Examples

```
Example 1

Input:
2 6 8 10 1
Output:
1 10 8 6 2

Example 2

Input:
1 2 3 4 5 6
Output:
6 5 4 3 2 1
```

---

5. Approaches

Approach 1: Iterative Reversal (MOST OPTIMAL)

Idea:

- Use 3 pointers: prev, curr, next.
- Reverse each link one by one.

Steps:

- Initialize:
  - prev = null
  - curr = head
- For each node:
  - Store next â†' next = curr.next
  - Reverse pointer â†' curr.next = prev
  - Move prev â†' curr
  - Move curr â†' next
- When loop finishes, prev is the new head.

Java Code:

```
static Node reverse(Node head) {
    Node prev = null;
    Node curr = head;

    while (curr != null) {
        Node next = curr.next;   // store next
        curr.next = prev;        // reverse pointer
        prev = curr;             // move prev
        curr = next;             // move curr
    }

    return prev; // new head
}
```

ðŸ' Intuition Behind the Approach:

- We walk through the list once.
- Each link is reversed so that direction becomes backward.
- At the end, the last node becomes the new head.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(n)
  - Why?
  - Each of the n nodes is visited exactly once.
- ðŸ'¾ Space Complexity
  - O(1)

- Why?
- Only uses constant extra pointers (prev, curr, next).

Approach 2: Recursion (Elegant but Not Safe for Large n)

Idea:

- Break the list into two parts:
  - Reverse the rest of the list
  - Attach the first node at the end

Steps:

- Base case:
  - If head is null or head.next == null â†' return head
- Recursively reverse the remaining list
- Point head.next.next = head
- Set head.next = null
- Return new head from recursion

Java Code:

```
static Node reverse(Node head) {
    if (head == null || head.next == null)
        return head;

    Node newHead = reverse(head.next);

    head.next.next = head; // reverse link
    head.next = null;

    return newHead;
}
```

ðŸ' Intuition Behind the Approach:

- Recursion unwinds from the last node upward.
- Each call flips the direction of a single link.
- The last node naturally becomes the head.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(n)
  - Why?
  - Each node participates in one recursive call.
- ðŸ'¾ Space Complexity
  - O(n)
  - Why?

    - Recursion depth = number of nodes (n)
    - Each call uses stack space.

---

6. Justification / Proof of Optimality

- The iterative pointer reversal approach is the optimal solution because:
- It visits each node once
- It requires constant memory (O(1))
- It modifies the list in-place
- It avoids recursion depth issues
- This is the version expected in interviews and coding rounds.

---

7. Variants / Follow-Ups

- Reverse a list in groups of k
- Reverse a sub-list between indices [m, n]
- Reverse even/odd-position nodes
- Reverse using recursion in tail-call optimized languages
- Reverse doubly linked list (constant time per node)

---

8. Tips & Observations

- Always maintain next pointer before modifying curr.next
- Iterative solution is the fastest and safest
- Recursion works beautifully but stack overflow is a risk
- Do not use extra data structures unless required

Q108: Remove Nth Node From End of List,

1. Problem Understanding

- You are given a singly linked list and an integer n.
- Your task: Remove the nth node from the end and return the new head of the list.
- Key Details:
    - The list has at least 1 node.
    - Removing the nth from end can also mean removing the head (if n == length).
    - Problem expects pointer manipulation, not converting to array.

---

2. Constraints

- 1 â‰¤ k â‰¤ 30 (length of list)

- 1 â‰¤ Node.val < 100
- 1 â‰¤ n â‰¤ k
- List is singly linked

---

3. Edge Cases

- Remove the first node (if n == k)
- Remove the last node (n == 1)
- Single node list (k = 1)
- Removing a middle node
- Removing when duplicates exist

---

4. Examples

```
Example 1

Input:

1 2 3 4 5 6
n = 2


Output:

1 2 3 4 6


Explanation: 2nd from end = value 5 â†' remove it.

Example 2

Input:

7 6 5 4 3
n = 4


Output:

7 5 4 3


Explanation: 4th from end = value 6 â†' remove it.
```

---

5. Approaches

Approach 1: Two Pointer Technique (Optimal Approach)

Idea:

- Use two pointers:
- Move fast pointer n steps ahead
- Then move slow and fast together until fast reaches end
- slow will be just before the node to delete
- This avoids counting the total length.

Steps:

- Create a dummy node before head
- Move fast pointer n + 1 steps
- Move both slow and fast until fast == null
- Now slow.next is the node to remove
- Do slow.next = slow.next.next

Java Code:

```
static Node removeNthFromEnd(Node head, int n) {
    Node dummy = new Node(-1);
    dummy.next = head;

    Node fast = dummy;
    Node slow = dummy;

    // Move fast n+1 steps
    for (int i = 0; i <= n; i++) {
        fast = fast.next;
    }

    // Move both until fast reaches end
    while (fast != null) {
        fast = fast.next;
        slow = slow.next;
    }

    // Delete node
    slow.next = slow.next.next;

    return dummy.next;
}
```

ðŸ' Intuition Behind the Approach:

- If fast is n nodes ahead of slow,
- when fast reaches the end,
- slow will be exactly at the node before the one to remove.
- This eliminates the need for an explicit length calculation.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(k)
  - Why?
    - Both pointers traverse the list once.
    - No extra nested loops.
- ðŸ'¾ Space Complexity
  - O(1)
  - Why?
    - Only a few pointers are used regardless of list size.

Approach 2: Length Counting (Simpler but 2-pass)

Idea:

- Traverse the list to count total length
- Compute position from front: pos = length - n
- Traverse again and delete that node

Steps:

- First pass: count nodes
- If removing head â†' return head.next
- Second pass: stop at pos-1
- Remove using pointer skip

Java Code:

```
static Node removeNthFromEnd(Node head, int n) {
    int length = 0;
    Node curr = head;

    while (curr != null) {
        length++;
        curr = curr.next;
    }

    // Remove head case
    if (n == length) return head.next;

    curr = head;
    int steps = length - n - 1;

    for (int i = 0; i < steps; i++) {
        curr = curr.next;
    }

    curr.next = curr.next.next;
    return head;
}
```

ðŸ' Intuition Behind the Approach:

- This method directly calculates the index to remove.
- Itâ€™s straightforward but requires two complete passes.

Complexity (Time & Space):

- â±ï¸ Time Complexity
    - O(k) for counting
    - O(k) for deleting
    - Total: O(k)
    - Why?
        - Two linear passes â†' still linear.
- ðŸ'¾ Space Complexity
    - O(1)
        - Why?
            - Only counters and pointers, no extra structures.

---

6. Justification / Proof of Optimality

- The two-pointer technique is the optimal solution because:
    - It completes in one pass
    - Uses constant memory
    - Works for all cases including removing the head
    - Cleaner and preferred in interviews

---

7. Variants / Follow-Ups

- Remove nodes with a given value
- Remove duplicates in sorted list
- Remove middle node
- Remove nth node from start
- Remove kth node using recursion
- Remove last occurrence of a value
- Delete node in circular linked list

---

8. Tips & Observations

- Always use a dummy node to avoid special cases
- Two-pointer approach is the standard interview solution
- Avoid stack/recursion due to unnecessary memory
- After deleting, check if list becomes empty

Q109: Middle Node of a Linked List

---

1. Problem Understanding

- You are given the head of a singly linked list.
- Your task: Return the middle node.
- Rules:
  - If n is odd â†' return the exact middle
  - If n is even â†' return the second middle
  - (Example: [5, 4, 3, 2] â†' two middles: 4, 3 â†' return 3)
- Driver code will print the linked list starting from the returned middle node.

---

2. Constraints

- 1 â‰¤ n â‰¤ 10^5
- Node values are integers
- Linked list is singly linked
- Function should run in linear time

---

3. Edge Cases

- Single node â†' return head
- Two nodes â†' return second node
- Even sized list â†' return n/2 + 1 node
- Long list (up to 1e5 nodes) â†' recursion should be avoided
- List containing duplicates

---

4. Examples

```
Example 1

Input:

5 4 3 2


Output:

3 2

Example 2

Input:

5 7 1


Output:

```

```
7 1
```

---

5. Approaches

Approach 1: Slow & Fast Pointers (Optimal)

Idea:

- Use two pointers:
  - slow moves one step
  - fast moves two steps
- When fast reaches the end:
- â†' slow is at the middle.
- This automatically returns the second middle for even-length lists.

Steps:

- Initialize:
- slow = head
- fast = head
- While fast != null and fast.next != null:
- slow = slow.next
- fast = fast.next.next
- When loop ends â†' slow is middle
- Return slow

Java Code:

```
static Node midpointOfLinkedList(Node head) {
    Node slow = head;
    Node fast = head;

    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }

    return slow;
}
```

ðŸ' Intuition Behind the Approach:

- Fast moves twice as fast.
- So when fast finishes the list:
  - If odd length â†' slow is exactly middle
  - If even length â†' slow is second middle

- This matches the required behavior perfectly.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(n)
  - Why?
    - Every iteration advances fast by 2 and slow by 1.
    - Together, they traverse at most n steps.
- ðŸ'¾ Space Complexity
  - O(1)
  - Why?
    - Only two pointers used, regardless of list size.

Approach 2: Count Length + Traverse (Two-pass Method)

Idea:

- Traverse list to calculate length L
- Middle index = L/2 (integer division, gives second middle)
- Traverse again to reach middle

Steps:

- Count nodes
- Stop at index L/2
- Return that node

Java Code:

```java
static Node midpointOfLinkedList(Node head) {
    int length = 0;
    Node curr = head;

    while (curr != null) {
        length++;
        curr = curr.next;
    }

    int midIndex = length / 2;

    curr = head;
    for (int i = 0; i < midIndex; i++) {
        curr = curr.next;
    }

    return curr;
}
```

ðŸ' Intuition Behind the Approach:

- Straightforward method:
- Just get total length and go to L/2 index.
- Simple but requires two complete passes.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
    - O(n) for counting length
    - O(n) for reaching middle
    - Total: O(n)
    - Why?
        - Two linear passes over list.
- ðŸ’¾ Space Complexity
    - O(1)
    - Why?
        - Only counters and pointers used.

---

6. Justification / Proof of Optimality

- The fast & slow pointer approach is the optimal method because:
- It finishes in a single pass
- It uses constant memory
- It naturally returns the second middle
- Works for large inputs (n = 100000) without stack risk
- This is the approach expected in interviews.

---

7. Variants / Follow-Ups

- Return the first middle for even length
- Find the node before middle
- Delete the middle node
- Find middle in circular linked list
- Get the middle element at each insertion (useful for practice problems)

---

8. Tips & Observations

- Always prefer fast & slow pointers for mid-related linked list problems
- Avoid recursion for deep lists
- For even length lists, common definition is to return second middle
- Two-pointer technique is also used in:
    - Cycle detection
    - Palindrome check
    - Finding kth element from end

Q110: Intersection of Two Linked Lists

1. Problem Understanding

- You are given two singly linked lists which eventually merge into a common tail.
- Your task: Find the intersection point (node value).
- Points to note:
    - Intersection means same node by reference, not same value.
    - After intersection, both lists share the same nodes.
    - Lists can have different lengths.
    - Must return the value of the intersection node.

---

2. Constraints

- 1 â‰¤ T â‰¤ 10
- 1 â‰¤ N, M â‰¤ 10^4
- Node values up to 10^5
- Lists are non-empty
- Intersection is guaranteed (based on input structure)

---

3. Edge Cases

- Intersection at head
- Intersection at very last node
- One list is significantly longer
- Large lists (10k nodes) â†’ avoid recursion
- No intersection (not given in this problem, but generally possible)

---

4. Examples

```
Example 1

Input:

5 1 3
3 6 9 15 30
10


Output:

15

Example 2
```

```
Input:

5 1 3
1 2 3 4 5
3


Output:

4
```

---

5. Approaches

Approach 1: Length Difference Method (Optimal & Standard)

Idea:

- Compute lengths of both lists.
- Advance the pointer of the longer list by |len1 - len2| steps.
- Now both pointers are the same distance from intersection.
- Move both one step at a time until they meet â†' intersection node.

Steps:

- Count length of L1 â†' len1
- Count length of L2 â†' len2
- Set ptr1 = head1, ptr2 = head2
- If one list is longer, advance its pointer
- Move both until ptr1 == ptr2
- Return ptr1 (intersection)

Java Code:

```java
static Node intersection(Node head1, Node head2) {
    int len1 = 0, len2 = 0;

    Node temp1 = head1, temp2 = head2;

    while (temp1 != null) {
        len1++;
        temp1 = temp1.next;
    }

    while (temp2 != null) {
        len2++;
        temp2 = temp2.next;
    }

    temp1 = head1;
```

```
        temp2 = head2;

        int diff = Math.abs(len1 - len2);

        if (len1 > len2) {
            while (diff-- > 0) temp1 = temp1.next;
        } else {
            while (diff-- > 0) temp2 = temp2.next;
        }

        while (temp1 != temp2) {
            temp1 = temp1.next;
            temp2 = temp2.next;
        }

        return temp1;
    }
```

ðŸ' Intuition Behind the Approach:

- If you skip extra nodes from the longer list,
- both pointers will reach the intersection at the same time.
- Now they walk together until they meet.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(N + M)
  - Why?
    - One pass for length counting + one pass for alignment + one pass to find intersection.
- ðŸ'¾ Space Complexity
  - O(1)
  - Why?
    - Only pointers and counters used.

Approach 2: Two Pointer Magic (Elegant, No Length Calculation)

Idea:

- This is the most elegant and interview-famous solution.
- Let pointers p1 and p2 traverse both lists.
- When a pointer reaches the end, redirect it to the other list's head.
- Why does this work?
- Because both pointers travel exactly:
- len1 + len2 distance â†' guarantees meeting point.

Steps:

- Init p1 = head1, p2 = head2
- Move both pointers forward
- When pointer reaches null â†' reset it to other listâ€™s head

- Eventually both pointers meet at intersection

Java Code:

```
static Node intersection(Node head1, Node head2) {
    Node p1 = head1;
    Node p2 = head2;

    while (p1 != p2) {
        p1 = (p1 == null) ? head2 : p1.next;
        p2 = (p2 == null) ? head1 : p2.next;
    }

    return p1; // or p2
}
```

ðŸ' Intuition Behind the Approach:

- After switching heads:
  - Both pointers travel the same total distance
  - So they meet exactly at intersection
  - If no intersection â†' both reach null together
  - (not required in this problem, but useful truth)

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(N + M)
  - Why?
    - Each pointer traverses both lists exactly once.
- ðŸ'¾ Space Complexity
  - O(1)
  - Why?
    - Only two pointers used.

---

6. Justification / Proof of Optimality

- The Two Pointer Magic method is the most elegant, one-pass, and constant-space solution.
- The Length Difference method is more intuitive but equally optimal.
- Both satisfy the constraints perfectly.

---

7. Variants / Follow-Ups

- Detect intersection without guarantee
- Find intersection of two circular linked lists
- Merge two lists and find intersection
- Find intersection of more than 2 lists

- Intersection where values equal but references differ (not considered here)

---

8. Tips & Observations

- Intersection means same node, not just same value
- Two-pointer method simplifies logic
- Good for interviewer discussion
- Hashing method is simple but not optimal
- Avoid brute-force for large lists

Q111: Remove Duplicates From Sorted Linked List

---

1. Problem Understanding

- You are given the head of a sorted singly linked list.
- Your task is to remove all duplicates so that each element appears only once.
- Return the final head.
- Since the list is sorted, duplicates always appear next to each other.

---

2. Constraints

- 1 â‰¤ n â‰¤ 300
- Node values range from -100 to 100
- List is sorted in non-decreasing order
- Only deletion of duplicates required, not rearrangement

---

3. Edge Cases

- Single node list â†’ no change
- All nodes identical (e.g., 1 1 1 1)
- No duplicates at all
- Duplicates only at the start
- Duplicates only at the end
- Negative values
- Mixed duplicates

---

4. Examples

```
Example 1
```

```
Input:

1 1 2


Output:

1 2

Example 2

Input:

1 1 2 3 3


Output:

1 2 3
```

---

5. Approaches

Approach 1: One-Pass Iterative (Optimal)

Idea:

- Because the list is sorted, duplicates appear consecutively.
- We just check if next node has the same value, and skip it.

Steps:

- Use a pointer curr starting at head
- If curr.data == curr.next.data:
- Skip the next node â†' curr.next = curr.next.next
- Else move curr = curr.next
- Continue until list ends
- Return head

Java Code:

```
static Node deleteDuplicates(Node head) {
    if (head == null) return null;

    Node curr = head;

    while (curr != null && curr.next != null) {
        if (curr.data == curr.next.data) {
            curr.next = curr.next.next;
        } else {
            curr = curr.next;
```

```
            }
        }

        return head;
    }
```

ðŸ' Intuition Behind the Approach:

- Since the list is sorted, duplicates will always be together.
- We donâ€™t need extra memoryâ€”just skip equal consecutive nodes.
- Each duplicate is simply removed by updating pointers.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(n)
  - Why?
    - We traverse the list once, checking each pair only once.
- ðŸ'¾ Space Complexity
  - O(1)
  - Why?
    - We only use a few pointers, no additional data structures.

Approach 2: Recursive Removal (Elegant but Costly)

Idea:

- Recursively remove duplicates from the next nodes,
- then compare head with the next returned node.

Steps:

- If list empty or 1 node â†' return head
- Recursively call for head.next
- If head.data == head.next.data, skip next
- Else keep as it is

Java Code:

```
static Node deleteDuplicates(Node head) {
    if (head == null || head.next == null) return head;

    head.next = deleteDuplicates(head.next);

    if (head.data == head.next.data) {
        return head.next;
    } else {
        return head;
    }
}
```

ðŸ' Intuition Behind the Approach:

- Work on the tail first, then fix head comparison.
- Cleaner approach but uses recursion stack.

Complexity (Time & Space):

- â±ï¸ Time Complexity
- O(n)
- ðŸ'¾ Space Complexity
    - O(n) due to recursion stack
    - Why?
        - Worst-case recursion depth = list length.

---

6. Justification / Proof of Optimality

- The one-pass iterative method is the best because:
    - Works in a single traversal
    - Uses zero extra memory
    - Avoids recursion
    - Guaranteed correct due to sorted property
- This is the method expected in interviews.

---

7. Variants / Follow-Ups

- Remove duplicates from unsorted list â†' use Set or sorting
- Remove duplicates so elements appear at most twice
- Keep only nodes that appear exactly once
- Remove duplicates from doubly linked list
- Remove duplicates from a circular list

---

8. Tips & Observations

- Sorted property is key â†' simplifies logic
- Always check curr != null && curr.next != null
- Avoid moving curr forward on removing duplicates
- Perfect warm-up for pointer manipulation problems
- Beware of infinite loops when skipping nodes

Q112: Unfold the Linked List

---

1. Problem Understanding

- You are given a folded linked list.
- Folding pattern:
  - Original: L0 â†' L1 â†' L2 â†' L3 â†' L4 â†' L5 â†' ...
  - Folded: L0 â†' Ln â†' L1 â†' Ln-1 â†' L2 â†' Ln-2 â†' ...
- Your task is to unfold the list and reconstruct the original order.

---

2. Constraints

- 1 â‰¤ n â‰¤ 1000
- Linked list is singly linked
- Values are integers
- Folding is guaranteed valid

---

3. Edge Cases

- Single node â†' same list
- Two nodes â†' already unfolded
- Odd vs even number of nodes
- Repeated values
- Must preserve original order, not sorted order

---

4. Examples

```
Example 1

Input:
1 6 2 5 3 4
Output:
1 2 3 4 5 6

Example 2

Input:
1 5 2 4 3
Output:
1 2 3 4 5
```

---

5. Approaches

Approach 1: Using Two Lists (Optimal & Common Approach)

Idea:

- The folded list has two interleaved lists:
  - odd positions â†' original left half

- even positions â†' reversed right half
- Steps to restore original:
- Split into two lists:
  - left = L0 â†' L1 â†' L2 â†' ...
  - right = Ln â†' Ln-1 â†' ... (in reverse order)
- Reverse the right list
- Attach reversed right list to end of left list

Steps:

- Create two dummy lists: odd and even
- Traverse original list:
  - nodes at index 0,2,4â€¦ â†' odd
  - nodes at index 1,3,5â€¦ â†' even
- Reverse the even list
- Concatenate odd list + reversed even list
- Return head of odd list

Java Code:

```
static Node unfold(Node head) {
    if (head == null || head.next == null)
        return head;

    Node oddHead = head;
    Node evenHead = head.next;

    Node odd = oddHead;
    Node even = evenHead;

    // Separate odd and even positioned nodes
    while (even != null && even.next != null) {
        odd.next = even.next;
        odd = odd.next;

        even.next = odd.next;
        even = even.next;
    }

    odd.next = null; // end odd list

    // Reverse even list
    Node revEven = reverse(evenHead);

    // Attach reversed even list
    odd.next = revEven;

    return oddHead;
}

// Standard reverse function
```

```
static Node reverse(Node head) {
    Node prev = null;
    Node curr = head;

    while (curr != null) {
        Node next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

ðŸ' Intuition Behind the Approach:

- Folding mixes the list like:
  - L0, Ln, L1, Ln-1, L2, Ln-2 ...
- So:
  - Odd positioned elements produce the left-to-right order (partially correct)
  - Even positioned elements are in reverse order of the right half
- Thus:
  - Separate odd/even
  - Reverse even list
  - Attach
- This reconstructs the original order.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(n)
  - Why?
  - Single traversal to split + single traversal to reverse + constant-time attach.
- ðŸ'¾ Space Complexity
  - O(1)
  - Why?
  - Only pointers used; no extra lists except reconstructed pointer links.

Approach 2: Using an Array (Simpler but Not Optimal)

Idea:

- Convert nodes into array, rearrange them into original order, rebuild list.

Steps:

- Put all nodes into array
- First half = original left half
- Second half = reversed right half
- Re-link nodes in order

Java Code:

```
static Node unfold(Node head) {
    ArrayList<Node> arr = new ArrayList<>();
    Node curr = head;
    while (curr != null) {
        arr.add(curr);
        curr = curr.next;
    }

    int i = 0, j = arr.size() - 1;
    Node dummy = new Node(-1);
    Node tail = dummy;

    while (i <= j) {
        tail.next = arr.get(i++);
        tail = tail.next;
        if (i <= j) {
            tail.next = arr.get(i);
            i++;
        }
        if (i <= j) {
            tail.next = arr.get(j--);
            tail = tail.next;
        }
    }

    tail.next = null;
    return dummy.next;
}
```

ðŸ' Intuition Behind the Approach:

- Using array, we can directly reorder nodes since we have random access.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
- O(n)
- ðŸ'¾ Space Complexity
    - O(n)
    - Why?
        - Array stores all nodes.

---

6. Justification / Proof of Optimality

- The odd-even separation + reverse method is optimal because:
- Uses only pointer manipulation
- No additional memory

- Linear time
- Matches the exact fold/unfold logic
- This is the approach used in interviews and competitive programming.

---

7. Variants / Follow-Ups

- Fold a list
- Check if a list is folded
- Fold/unfold a doubly linked list
- Zig-zag rearrangement
- Rearrange list into L0 â†’ Ln â†’ L1 â†’ Ln-1 â€¦
- Merge two halves alternatively

---

8. Tips & Observations

- Always check for null before accessing next
- Use a dummy node for clarity in some variations
- Reversing even list is essential
- Odd-even logic ONLY works because fold is deterministic
- Avoid array unless memory is not a concern

Q113: Segregating 0s, 1s, and 2s in a Linked List

---

1. Problem Understanding

- You are given a linked list with nodes containing only 0, 1, and 2.
- You must rearrange the linked list so that:
  - All 0s come first
  - then 1s
  - then 2s
- Order among equal elements does not need to be preserved strictly (but in-place methods preserve relative order).

---

2. Constraints

- 1 â‰¤ n â‰¤ 1000
- Node values: 0, 1, or 2
- List is singly linked

---

3. Edge Cases

- List of only 0s, only 1s, or only 2s
- Empty list
- Single element
- Already sorted
- Reverse sorted
- All duplicates clustered

---

4. Examples

```
Example 1

Input:
0 1 1 1 2 0 1 0 1 1

Output:
0 0 0 1 1 1 1 1 1 2

Example 2

Input:
0 1 2 0 1

Output:
0 0 1 1 2
```

---

5. Approaches

Approach 1: Counting Method (Optimal, Simplest logic)

Idea:

- Count number of 0, 1, 2
- Then traverse list again and overwrite values.

Steps:

- Traverse list and count c0, c1, c2
- Traverse list again:
    - First fill c0 times 0
    - Then c1 times 1
    - Finally c2 times 2
- Return head

Java Code:

```
static Node segregate(Node head) {
    int c0=0, c1=0, c2=0;
```

```
    Node curr = head;
    while (curr != null) {
        if (curr.data == 0) c0++;
        else if (curr.data == 1) c1++;
        else c2++;
        curr = curr.next;
    }

    curr = head;
    while (c0-- > 0) { curr.data = 0; curr = curr.next; }
    while (c1-- > 0) { curr.data = 1; curr = curr.next; }
    while (c2-- > 0) { curr.data = 2; curr = curr.next; }

    return head;
}
```

ðŸ' Intuition Behind the Approach:

- This is the linked-list version of Dutch National Flag.
- Weâ€™re not changing nodes, just updating values.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(n)
  - Why?
    - One pass to count + one pass to rewrite values.
- ðŸ'¾ Space Complexity
  - O(1)
  - Why?
    - Only integer counters used.

Approach 2: Three Dummy Lists

Idea:

- Create 3 separate lists:
  - One for 0s
  - One for 1s
  - One for 2s
- Use original nodes (no new nodes).
- Then connect:
  - 0-list â†' 1-list â†' 2-list

Steps:

- Create 3 dummy heads: zero, one, two
- Traverse input list:
  - If data is 0 â†' move node to zero list

- If 1 â†’ move to one list
- If 2 â†’ move to two list
- Connect:
  - zero â†’ one â†’ two
  - Return zero.next

Java Code:

```
static Node segregate(Node head) {
    Node zeroD = new Node(-1), oneD = new Node(-1), twoD = new Node(-1);
    Node zero = zeroD, one = oneD, two = twoD;

    Node curr = head;

    while (curr != null) {
        if (curr.data == 0) {
            zero.next = curr;
            zero = zero.next;
        }
        else if (curr.data == 1) {
            one.next = curr;
            one = one.next;
        }
        else {
            two.next = curr;
            two = two.next;
        }
        curr = curr.next;
    }

    // Connect lists
    zero.next = oneD.next != null ? oneD.next : twoD.next;
    one.next = twoD.next;
    two.next = null;

    return zeroD.next;
}
```

ðŸ' Intuition Behind the Approach:

- Instead of creating new nodes (your mistake),
- we reuse the original nodes and rearrange pointers.
- This is fully in-place.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(n)
  - Why?
    - One single pass through list.

- ðŸ'¾ Space Complexity
    - O(1)
    - Why?
        - Only uses dummy heads (constant nodes).

Approach 3: Dutch National Flag with Pointer Swaps

Idea:

- Instead of counting or splitting the list into multiple lists,
- we perform a 3-way partition directly on the linked list â€"
- just like the Dutch National Flag algorithm.
- We maintain 3 regions while traversing:
    - Region of 0s
    - Region of 1s
    - Region of 2s
- But since it's a singly linked list, we can't use array-like indices.
- Instead, we use three pointers to maintain boundaries and swap data values.
- Important:
- This method only swaps node.data, not links.

Steps:

- Use three pointers:
    - ptr0 â†' last node where data = 0
    - ptr1 â†' last node where data = 1
    - ptr2 â†' current scanning node
- Traverse list using ptr2:
- If ptr2.data == 0
    - Swap ptr2.data with ptr0.next.data
    - Move both ptr0 and ptr2
- If ptr2.data == 1
    - Swap with ptr1.next.data
    - Move ptr1 and ptr2
- If ptr2.data == 2
    - Just move ptr2
- Goal:
    - [0 region] [1 region] [Unprocessed region]
    - This builds 0s first, then 1s, then 2s.

Java Code:

```
static Node segregate(Node head) {
    if (head == null || head.next == null)
        return head;

    // ptr0 will mark end of 0s region
    Node ptr0 = new Node(-1);
```

```
        ptr0.next = head;
        ptr0 = ptr0.next;

        // ptr1 will mark end of 1s region
        Node ptr1 = ptr0;

        // ptr2 scans list
        Node ptr2 = ptr0.next;

        while (ptr2 != null) {
            if (ptr2.data == 0) {
                // expand 0-region -> swap ptr2 with ptr0
                int temp = ptr2.data;
                ptr2.data = ptr0.data;
                ptr0.data = temp;

                ptr0 = ptr0.next;
                if (ptr1 == ptr0) ptr1 = ptr1.next;

                ptr2 = ptr2.next;
            }
            else if (ptr2.data == 1) {
                // expand 1-region -> swap ptr2 with ptr1
                int temp = ptr2.data;
                ptr2.data = ptr1.data;
                ptr1.data = temp;

                ptr1 = ptr1.next;
                ptr2 = ptr2.next;
            }
            else {
                // data == 2 â†’ unprocessed region
                ptr2 = ptr2.next;
            }
        }

        return head;
    }
```

ðŸ' Intuition Behind the Approach:

- This uses the exact logic of Dutch National Flag:
- Move 0s to front
- Move 1s to the middle
- Leave 2s at the back
- But since linked lists canâ€™t jump back and forth,
- we simulate the 3-way partition by maintaining pointers to the boundaries
- and swapping values into correct regions.
- We simulate the 3 bins (0,1,2) in-place, rearranging each item into its correct region by swapping data fields.
- This avoids:

- Creating new nodes
- Managing multiple lists
- Using counting passes
- Instead, it dynamically rearranges data during one traversal.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(n)
  - WHY?
    - Each of the nodes is visited exactly once by ptr2.
- ðŸ’¾ Space Complexity
  - O(1)
  - WHY?
    - Only a few pointers (ptr0, ptr1, ptr2) and temp swaps are used.
- When to Use / Avoid
  - Use when:
  - You want in-place partitioning
  - No extra memory allowed
  - Youâ€™re allowed to modify node data
  - Avoid when:
    - Data swap not allowed
    - Nodes must not be modified except via linking
    - List contains large objects (not primitive values)

---

6. Justification / Proof of Optimality

- The optimal solutions are:
- Approach 1 (Counting) â†’ simplest, fastest
- Approach 2 (3-lists in-place) â†’ best pointer-based solution
- Your old solution was close to Approach 2, but created new nodes, making it O(n) space.
- Correct approach uses no new nodes, just pointer attachments.

---

7. Variants / Follow-Ups

- Sort linked list of 0â€“1 only
- Sort linked list with arbitrary integers (merge sort)
- Sort linked list based on custom 3-way criteria
- Sort linked list but preserve stability

---

8. Tips & Observations

- For values limited to 0, 1, 2 â†’ counting is easiest
- For general linked list sorting â†’ merge sort is required
- Avoid creating new nodesâ€”pointer manipulation is faster & memory-efficient

---

Q114: Rearrange Evenâ€“Odd Nodes

1. Problem Understanding

- Given the head of a singly linked list, rearrange the list so that:
  - All even-valued nodes come before all odd-valued nodes
  - Relative order inside even group must remain preserved
  - Relative order inside odd group must remain preserved
  - Modify the list in-place (no creation of new actual nodes)

---

2. Constraints

- 1 â‰¤ n â‰¤ 1000
- Node values are integers
- Must preserve original ordering within each group
- Must not create new nodes representing list values

---

3. Edge Cases

- All values even â†’ no change
- All values odd â†’ no change
- Single element
- Mixed order
- Already partitioned
- Even values appear after some odd values

---

4. Examples

```
Example 1

Input:

1 2 3 4 5


Output:

2 4 1 3 5

Example 2

Input:

2 4 6 8 10 1 3 5 7 9
```

```
Output:

2 4 6 8 10 1 3 5 7 9
```

---

5. Approaches

Approach 1: In-Place Stable Partition Using Dummy Nodes (Optimal)

Idea:

- Use two pointer chains:
  - even list to gather all nodes with value % 2 == 0
  - odd list to gather all nodes with value % 2 == 1
- We reuse the same nodes, we simply attach them into two separate chains, then connect:
- even-list â†' odd-list

Steps:

- Create two dummy nodes evenD, oddD
- Traverse original list:
  - If node.val is even â†' append to even chain
  - Else â†' append to odd chain
- Connect even chain tail â†' odd chain head
- end odd list with null
- Return evenD.next

Java Code:

```java
public Node rearrangeList(Node head) {
    if (head == null || head.next == null) return head;

    Node evenD = new Node(-1), oddD = new Node(-1);
    Node even = evenD, odd = oddD;

    Node curr = head;

    while (curr != null) {
        if (curr.val % 2 == 0) {
            even.next = curr;
            even = even.next;
        } else {
            odd.next = curr;
            odd = odd.next;
        }
        curr = curr.next;
    }

    odd.next = null;              // end odd list
```

```
        even.next = oddD.next;          // connect even + odd

        return evenD.next;              // head of rearranged list
    }
```

ðŸ’ Intuition Behind the Approach:

- Partitioning the original list into two linked chains maintains the order.
- Even list grows first, preserving all even nodes as they appear.
- Odd list grows next, preserving all odd nodes as they appear.
- Finally attaching the two lists produces the required output.
- This is exactly like stable partition in arrays but done with pointer adjustments.
- Dummy nodes are in-place because they do not store real data nodes, do not become part of the final list, and use only constant memory. The final list contains only original nodes.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(n)
  - Why?
    - Single traversal collecting even and odd chains.
- ðŸ'¾ Space Complexity
  - O(1)
  - Why?
    - Dummy nodes are constant-memory and do not scale with input size.

---

6. Variants / Follow-Ups

- Partition around any pivot value
- Stable partition without extra memory
- Partition doubly linked list
- Partition around multiple conditions (e.g., negative/zero/positive)

---

7. Tips & Observations

- Dummy nodes make merging logic easier
- Always end the final list with null
- Avoid creating new nodes with original values
- Use pointer manipulation for true in-place modification

Q115: Quick Sort on Linked List

---

1. Problem Understanding

- You are given the head of a linked list.
- You must sort the list using Quick Sort, but without converting it to an array.
- Quick Sort for linked list works differently from arrays:
- No random access
- We use partition by swapping data, not nodes
- We recursively sort left and right partitions

---

2. Constraints

- 1 <= n <= 20000
- Linked list nodes contain integers
- Must use Quick Sort, not Merge Sort
- Must maintain linked list structure

---

3. Edge Cases

- Single-node list â†' already sorted
- Already sorted list
- Reverse sorted list
- Duplicate values
- Very large list (ensure tail-next=null)
- Pivot at end may cause skew â†' still valid

---

4. Examples

```
Example 1

Input:
3
1 6 2

Output:
1 2 6

Example 2

Input:
4
1 9 3 8

Output:
1 3 8 9
```

---

5. Approaches

Approach 1: QuickSort Using Last Node as Pivot

Idea:

- Choose last node as pivot (end).
- Partition list so:
  - Nodes with value < pivot â†' left side
  - Nodes with value â‰¥ pivot â†' right side
- Swap values, not nodes (much easier).
- Recursively sort left and right sublists.

Steps:

- Find the tail of the list.
- Call quickSort(head, tail).
- Partition around pivot (the tail).
- Get pivotâ€™s correct position.
- QuickSort left part.
- QuickSort right part.

Java Code:

```java
// Partition the list around the end node as pivot
static Node partition(Node head, Node end, Node[] newHead, Node[] newEnd) {
    Node pivot = end;
    Node prev = null, curr = head, tail = pivot;

    // Initially new head and new end will be assigned later
    while (curr != pivot) {
        if (curr.data < pivot.data) {
            if (newHead[0] == null)
                newHead[0] = curr;
            prev = curr;
            curr = curr.next;
        } else {
            // Move nodes >= pivot to end
            if (prev != null)
                prev.next = curr.next;

            Node temp = curr.next;
            curr.next = null;
            tail.next = curr;
            tail = curr;
            curr = temp;
        }
    }

    if (newHead[0] == null)
        newHead[0] = pivot;

    newEnd[0] = tail;
```

```
        return pivot;
    }


    // Recursively apply quick sort
    static Node quickSortRecur(Node head, Node end) {
        if (head == null || head == end)
            return head;

        Node[] newHead = new Node[1];
        Node[] newEnd = new Node[1];

        Node pivot = partition(head, end, newHead, newEnd);

        // If pivot is not the smallest element
        if (newHead[0] != pivot) {
            Node temp = newHead[0];

            while (temp.next != pivot)
                temp = temp.next;

            temp.next = null;

            newHead[0] = quickSortRecur(newHead[0], temp);

            temp = getTail(newHead[0]);
            temp.next = pivot;
        }

        pivot.next = quickSortRecur(pivot.next, newEnd[0]);

        return newHead[0];
    }


    // Returns the tail of a linked list
    static Node getTail(Node head) {
        while (head != null && head.next != null)
            head = head.next;
        return head;
    }


    // Main QuickSort function
    static Node quickSort(Node head) {
        Node tail = getTail(head);
        return quickSortRecur(head, tail);
    }
```

ðŸ' Intuition Behind the Approach:

- QuickSort works best when we split partitions.

- Since linked lists donâ€™t allow random access, swapping nodes is hard.
- Instead, we swap data, making partition simpler.
- Pivot is chosen as last node for consistency.
- Recursion builds sorted sublists automatically.
- Final linking maintains correct order.

Complexity (Time & Space):

- Time Complexity
  - Best/Average: O(N log N)
  - Worst (already sorted / reverse): O(N^2)
- Space Complexity
  - Recursion depth: O(log N) (best)
  - Worst-case recursion: O(N)

Approach 2: QuickSort on Values Only

Idea:

- Copy all values to an array â†’ apply QuickSort â†’ rebuild list.
- â Œ Not allowed here (problem requires QuickSort on list itself).

---

6. Justification / Proof of Optimality

- QuickSort is chosen because:
- Partitioning is straightforward using last node as pivot.
- Works in-place; no extra arrays.
- Allowed by problem even though MergeSort is normally preferred.

---

7. Variants / Follow-Ups

- QuickSort with random pivot
- QuickSort by node swapping instead of data swapping
- 3-way partition QuickSort (useful with many duplicates)

---

8. Tips & Observations

- Always use last node as pivot for simplicity.
- Use data swapping, not node swapping.
- Ensure to properly cut and reattach sublists.
- Linked list QuickSort is trickier than array QuickSort because of no random access.
- For interviews, MergeSort is usually preferred â€” but QuickSort is asked for concept testing.

Q116: Merge Sort for Linked List

---

1. Problem Understanding

- You are given the head of a linked list with n nodes.
- Your task is to sort the linked list using Merge Sort.
- Merge Sort is ideal for linked lists because:
- It does not need random access
- It works in O(N log N) time
- Merging can be done efficiently using pointers
- This is the standard, most optimal sorting method for linked lists.

---

2. Constraints

- 1 <= n <= 100000
- 1 <= node.data <= 1000
- Must use Merge Sort (not arrays, not quick sort)
- Large input size â†' must be O(N log N) or better

---

3. Edge Cases

- Only 1 node â†' already sorted
- 2 nodes swapped â†' should sort correctly
- Repeated values
- Negative values? (Not required here, but logic supports it)
- Very long list (ensure no stack overflow)

---

4. Examples

```
Example 1

Input:
5
3 5 2 4 1

Output:
1 2 3 4 5

Example 2

Input:
6
3 5 2 4 1 6

Output:
1 2 3 4 5 6
```

---

5. Approaches

Approach 1: Classic Merge Sort on Linked List (Optimal)

Idea:

- Find mid using slow & fast pointers
- Split list into two halves
- Recursively sort each half
- Merge both halves using a sorted merge function

Steps:

- Base case: if head is null or head.next is null â†' return head
- Find mid: using slowâ€"fast pointers
- Split list: left = head, right = mid.next
- Sort both halves:
  - left = mergeSort(left)
  - right = mergeSort(right)
- Merge: return merge(left, right)

Java Code:

```
Find Middle Node
static Node getMid(Node head) {
    Node slow = head, fast = head.next;

    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
}

Merge Two Sorted Lists
static Node merge(Node a, Node b) {
    if (a == null) return b;
    if (b == null) return a;

    Node dummy = new Node(-1);
    Node temp = dummy;

    while (a != null && b != null) {
        if (a.data <= b.data) {
            temp.next = a;
            a = a.next;
        } else {
            temp.next = b;
            b = b.next;
        }
        temp = temp.next;
```

```
    }

    if (a != null) temp.next = a;
    else temp.next = b;

    return dummy.next;
}

Merge Sort on Linked List
static Node mergeSort(Node head) {
    if (head == null || head.next == null)
        return head;

    Node mid = getMid(head);
    Node rightHead = mid.next;
    mid.next = null;

    Node left = mergeSort(head);
    Node right = mergeSort(rightHead);

    return merge(left, right);
}
```

ðŸ’ Intuition Behind the Approach:

- Merge Sort is naturally suited for linked lists.
- Splitting using slowâ€“fast traversal is efficient (O(N)).
- Merging is pointer-based and does not require shifting values.
- Unlike arrays, QuickSort is inefficient on linked lists due to no random access.
- Merge Sort ensures stable, predictable O(N log N) behavior.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(N log N)
  - (N for each level merge, log N levels of recursion)
- ðŸ’¾ Space Complexity
  - O(log N) recursion stack
  - No extra list or array created

Approach 2: (Not Recommended): Convert to Array and Sort

Idea:

- Copy values into an array
- Sort array
- Rebuild list
- âŒ Violates problem requirement
- âŒ Uses extra O(N) space

---

6. Justification / Proof of Optimality

- Merge Sort is the most optimal and preferred method for sorting linked lists because:
- Linked lists allow cheap merging
- Do not support random access for QuickSort
- Merge Sort guarantees good performance in worst case
- Stable sorting method

---

7. Variants / Follow-Ups

- Merge sort on Doubly Linked List
- Bottom-up Merge Sort (iterative)
- Sorting K-sorted linked list using merge logic
- Merging K sorted lists (Heap + Merge logic)

---

8. Tips & Observations

- Always break the list into exact halves to avoid infinite recursion.
- Use slowâ€"fast technique to find the middle reliably.
- Merge function should not create new nodes â€" re-link existing ones.
- For very large lists, iterative bottom-up merge sort avoids recursion depth issues.
- After merge, ensure next pointers are not accidentally left pointing to old references.

Q117: Clone a Linked List with Next and Random Pointer

---

1. Problem Understanding

- We are given a linked list where each node has:
  - next pointer â†' points to next node
  - random pointer â†' points to ANY node in the list (or NULL)
- We must create a deep copy:
  - Create N new nodes
  - Each new node:
    - Has the same value
    - next and random pointers must point to new nodes only
  - No pointer of the cloned list should reference original nodes.
- This is a classic Random Pointer Linked List Cloning problem.

---

2. Constraints

- 1 <= N <= 100

- 1 <= M <= N
- Random pointer pairs given as:
  - a b meaning â†' node a has random pointer to node b
- Random pointer can be missing â†' treated as NULL
- Must return head of cloned list

---

3. Edge Cases

- Only 1 node
- No random pointers
- All random pointers NULL
- Random pointer pointing to itself
- Multiple nodes pointing to same random target
- Random pointers forming cycles
- Last node having random pointer
- Random pointers out of order (given as values not linked structure)

---

4. Examples

```
Example 1
1 -> 2 -> 3 -> 4
1->random = 2
2->random = 4


Output: 1 (means clone is valid)

Example 2
1 -> 3 -> 5 -> 9
1->random = 1
3->random = 4


Output: 1
```

---

5. Approaches

Approach 1: O(1) Extra Space â€" Interleaving Nodes (Optimal)

Idea:

- Clone each node and insert it next to original node:
  - 1 -> 1' -> 2 -> 2' -> 3 -> 3' ...
- Set all cloned nodes' random pointers by using:
  - clone.random = original.random.next
- Detach cloned list from original list.

Steps:

- Step 1 â€" Insert clone nodes after each original
  - For each node A, create A':
  - A -> A' -> B -> B' -> C -> C' ...
- Step 2 â€" Set random pointer of cloned nodes
  - If A.random = R
  - then
    - A'.random = R.next
  - Because R.next â†' Râ€™s clone.
- Step 3 â€" Detach cloned list
  - Extract all A' nodes into a separate cloned list.

Java Code:

```
static Node cloneLinkedList(Node head) {
    if (head == null) return null;

    Node curr = head;

    // Step 1: Insert cloned nodes
    while (curr != null) {
        Node next = curr.next;
        Node copy = new Node(curr.data);
        curr.next = copy;
        copy.next = next;
        curr = next;
    }

    // Step 2: Set random pointers
    curr = head;
    while (curr != null) {
        if (curr.random != null)
            curr.next.random = curr.random.next;
        curr = curr.next.next;
    }

    // Step 3: Separate cloned list
    curr = head;
    Node copyHead = head.next;
    Node copy = copyHead;

    while (curr != null) {
        curr.next = curr.next.next;
        if (copy.next != null)
            copy.next = copy.next.next;

        curr = curr.next;
        copy = copy.next;
    }
```

```
    return copyHead;
}
```

ðŸ’ Intuition Behind the Approach:

- We avoid extra space like HashMap by creating a twin node right next to original.
- This interleaving gives us direct access to cloned node of any node using curr.next.
- Since random pointers can point anywhere, interleaving ensures we can assign clone-randoms in O(1) time.
- Finally, we untangle both lists to restore the original and produce the clone.
- This is efficient and elegant.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(N)
  - (Every node visited a constant number of times)
- ðŸ’¾ Space Complexity
  - O(1) extra space
  - (No hashing, no extra arrays)

Approach 2: HashMap Method (Simpler but uses O(N) space)

Idea:

- Create all new nodes in first pass
- Store mapping: oldNode â†’ newNode
- Second pass: assign next & random using map

Java Code:

```
HashMap<Node, Node> map = new HashMap<>();

Node curr = head;
while (curr != null) {
    map.put(curr, new Node(curr.data));
    curr = curr.next;
}

curr = head;
while (curr != null) {
    map.get(curr).next = map.get(curr.next);
    map.get(curr).random = map.get(curr.random);
    curr = curr.next;
}

return map.get(head);
```

6. Justification / Proof of Optimality

- Interleaving method:
  - Uses constant space
  - Solves random pointer linking efficiently
  - Avoids extra memory overhead
  - Is the most optimal approach taught in interviews

---

7. Variants / Follow-Ups

- Clone a doubly linked list with random pointers
- Clone a tree with random pointers
- Clone graph using BFS/DFS
- Create deep copy of complex structures

---

8. Tips & Observations

- Interleaving list ensures random pointers can be assigned without storing mappings
- Always detach the lists at the end, or you will corrupt original list
- Use .next.next carefully during traversal
- Random pointer may be NULL â†' handle that condition
- This question appears frequently in FAANG interviews

Q118: Flattening a Linked List

---

1. Problem Understanding

- You are given a linked list where:
  - Each node has 2 pointers:
    - right â†' points to next list head
    - down â†' points to a sorted sub-linked-list
- Every sub-list is sorted
- We must flatten all lists into a single sorted list, using only the down pointer.
- After flattening:
  - All nodes appear in a single down chain
  - Output is printed using the down pointer only
  - The right pointer should not be used in final list
- This is similar to merge K sorted linked lists.

---

2. Constraints

- Number of lists n â‰¤ 50
- Size of each sublist k â‰¤ 20

- Total nodes â‰¤ 1000
- Values â‰¤ 1000
- Must maintain sorted order

---

3. Edge Cases

- Empty main list (n = 0)
- Only 1 list
- Each list has only 1 element
- Some lists have size 1, others long
- All values identical
- Highly skewed down chains

---

4. Examples

```
Example 1:

Flatten:

5 â†’ 10 â†’ 19 â†’ 28
â†“    â†“     â†“     â†“
7    20    22    35
â†“           â†“     â†“
8           50    40
â†“                 â†“
30                45


Output:

5 7 8 10 19 20 22 28 30 35 40 45 50

Example 2:

Output:

5 7 8 10 19 22 28 30 50
```

---

5. Approaches

Approach 1: Merge down lists one by one (Optimal & Simple)

Idea:

- Treat each right sub-list as a sorted linked list.
- We repeatedly:
  - Flatten the right sublist

    - Merge the current list with the flattened right list
    - Return merged result
- This reduces to merge of K sorted lists using recursion.

Steps:

- Base case â†' if head == null or head.right == null, return head
- Recursively flatten the list starting from head.right
- Merge head and flatten(head.right)
- Ensure result uses only down pointers
- Set right = null to fully flatten structure

Java Code:

```
Merge two sorted â€œdownâ€ linked lists
static Node merge(Node a, Node b) {
    if (a == null) return b;
    if (b == null) return a;

    Node result;

    if (a.data < b.data) {
        result = a;
        result.down = merge(a.down, b);
    } else {
        result = b;
        result.down = merge(a, b.down);
    }

    result.right = null; // ensure right pointers removed
    return result;
}

Flatten function
static Node flatten(Node root) {
    if (root == null || root.right == null)
        return root;

    // Flatten the right side first
    root.right = flatten(root.right);

    // Merge current list with flattened right list
    root = merge(root, root.right);

    return root;
}
```

ðŸ' Intuition Behind the Approach:

- Each nodeâ€™s â€œdownâ€ is a sorted list.
- The main list is: L1 -> L2 -> L3 -> ... -> Ln

- Flattening means merging these lists:
- merge(L1, merge(L2, merge(L3, ...)))
- Merging two sorted lists is O(N)
- Using recursion ensures gradually flattening from right â†’ left
- The right pointer is ignored in final flattened list
- This is efficient and clean.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
    - Let total nodes = N
        - Each merge operation = O(N)
        - Number of merges = n (number of lists â‰¤ 50)
    - Overall:
        - Time: O(N * log n) (if optimized)
        - Simple recursive merge gives O(N * n) (still fine for constraints)
- ðŸ’¾ Space Complexity
    - Recursion depth = O(n) â‰¤ 50
    - No extra linked lists created
    - Uses same nodes â†’ in-place flattening

---

6. Justification / Proof of Optimality

- The problem requires flattening multiple sorted linked lists into one sorted list.
- The optimal way to combine sorted lists is through merge operations, just like merging in Merge Sort.
- Since each nodeâ€™s down list is already sorted, merging two lists using the down pointer preserves sorted order without extra space.
- Recursively flattening the right side first ensures we gradually combine all lists from right to left, simplifying structure.
- Eliminating right pointers and relying only on down ensures the final flattened list is a single-level sorted list, exactly as required.
- This approach avoids creating new nodes, ensuring in-place, memory-efficient flattening.
- The merge-based method maintains optimal time complexity and is the standard recommended solution for this problem.

---

7. Variants / Follow-Ups

- Flatten a multilevel doubly linked list
- Merge K sorted linked lists (priority queue)
- Flatten a tree-like structure

---

8. Tips & Observations

- Always merge using down pointer only
- Merge like merging 2 sorted linked lists
- Set right = null to avoid mixing structures

- The flatten process works right â†' left
- Base case simplifies recursion
- Print using down pointer always
- Do not create new nodes (in-place merge required)
