[Click on the logo to find **Problem Statement**]

Intuition

Dummy Node for Edge Cases: Adding a dummy node at the beginning simplifies the logic. It ensures that the head of the linked list never needs special treatment. The dummy node acts as a placeholder and helps handle edge cases where the first node needs to be removed.

Two-Pointer Technique: The algorithm uses two pointers, prev and current, to traverse the linked list. prev always points to the node before current. This setup allows us to modify the pointers of nodes in the linked list effectively.

Iterative Traversal: The algorithm iterates through the linked list using the current pointer. It checks whether the value of the current node matches the given val.

Removing Nodes with Given Value: If the current node's value matches the given value, the prev.next pointer is adjusted to skip the current node. This effectively removes the current node from the linked list. If the current node's value does not match, the prev pointer is updated to prev.next, maintaining the connection in the linked list.

Disconnecting Tail Node: After the loop, the algorithm ensures that the last node's next pointer is set to None. This step handles the case where the last node in the original list has the given value and needs to be removed.

Returning Modified Linked List: Finally, the modified linked list starts from dummy.next. Since the dummy node was never linked to a node with the given value, it acts as the new head of the modified linked list.

Approach

Create a dummy node and set its next pointer to the head of the given linked list. This dummy node helps handle edge cases where the first node needs to be removed.

Initialize two pointers: prev and current. Start with prev pointing to the dummy node and current pointing to the head of the linked list. Traverse the linked list using the current pointer.

Check if the val of the current node is equal to the given value. If it is equal, skip the current node by updating the prev.next pointer to point directly to the node after the current node, effectively

removing the current node. If the val is not equal, update prev to point to the current node.

Move the current pointer to the next node in the linked list. Repeat steps 3 and 4 until the end of the linked list is reached.

After the loop, ensure that the last node (tail) is properly disconnected by setting prev.next to None.

The modified linked list starts from the dummy.next node. Return dummy.next.

Complexity

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

Code

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
  def removeElements(self, head: ListNode, val: int) -> ListNode:
    dummy = ListNode(0, head)
    prev = dummy

    while head:
      if head.val != val:
        prev.next = head
        prev = prev.next
      head = head.next
    prev.next = None

    return dummy.next
```