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

Intuition

Imagine two runners running on a track, one is fast (moves two steps at a time) and the other is slow (moves one step at a time). If the track is a circular one, the fast runner will eventually meet the slow runner.

In the context of a linked list:

Cycle Existence: If there is a cycle in the linked list, the fast pointer and slow pointer will eventually meet inside the cycle.

Speed Difference: The fast pointer moves at twice the speed of the slow pointer. If there is a cycle, the fast pointer will "lap" the slow pointer at least once during its traversal.

Meeting Point: Once inside the cycle, the fast and slow pointers are guaranteed to meet. This is because the fast pointer is always closing in on the slow pointer by one node at each step (due to the speed difference). Eventually, they will meet.

No Cycle Case: If there is no cycle, the fast pointer will reach the end of the linked list and the algorithm will terminate. There will be no meeting point of fast and slow pointers in this case.

Approach

Initialize Dummy Node: 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.

Iterate Through the Linked List: 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.

Remove Nodes with the Given Value: 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.

Continue Iterating: 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.

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

Return Modified Linked List: 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
```