```python
# DSA
# Basic linked list operations


# Reversee a linked list in-place

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

def reverse_linked_list(head):
    prev = None
    current = head
    while current:
        next_node = current.next
        current.next = prev
        prev = current
        current = next_node
    return prev  # New head of the reversed list



# Detect a cycle in a linked list

def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True  # Cycle detected
    return False



# Merge two sorted linked lists

def merge_two_sorted_lists(l1, l2):
    dummy = ListNode()
    tail = dummy

    while l1 and l2:
        if l1.val < l2.val:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next
    tail.next = l1 or l2
    return dummy.next



# Remove the nth node from the end

def remove_nth_from_end(head, n):
    dummy = ListNode(0, head)
    first = second = dummy

    for _ in range(n + 1):
        first = first.next

    while first:
        first = first.next
        second = second.next

    second.next = second.next.next
    return dummy.next

def remove_duplicates(head):
    current = head
    while current and current.next:
        if current.val == current.next.val:
            current.next = current.next.next
        else:
            current = current.next
    return head



# Remove duplicates from a sorted linked list

def remove_duplicates(head):
    current = head
    while current and current.next:
        if current.val == current.next.val:
            current.next = current.next.next
        else:
            current = current.next
    return head



# Find the intersection of two linked lists
def get_intersection_node(headA, headB):
    if not headA or not headB:
        return None
    a, b = headA, headB
    while a != b:
        a = a.next if a else headB
        b = b.next if b else headA
    return a




# Assignment Quesition


# Rotate a linked list by k positions

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

    # Find length and last node
    length, tail = 1, head
    while tail.next:
        tail = tail.next
        length += 1

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

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

    new_head = new_tail.next
    new_tail.next = None
    tail.next = head

    return new_head



# Add two numbers represented by linked lists

def add_two_numbers(l1, l2):
    dummy = ListNode()
    current, carry = dummy, 0

    while l1 or l2 or carry:
        sum_val = (l1.val if l1 else 0) + (l2.val if l2 else 0) + carry
        carry, val = divmod(sum_val, 10)

        current.next = ListNode(val)
```

```python
        current = current.next

        if l1:
            l1 = l1.next
        if l2:
            l2 = l2.next

    return dummy.next



# Clone a linked list with a random pointer

class RandomListNode:
    def __init__(self, val=0, next=None, random=None):
        self.val = val
        self.next = next
        self.random = random

def clone_linked_list(head):
    if not head:
        return None

    # Step 1: Insert new nodes
    current = head
    while current:
        new_node = RandomListNode(current.val, current.next)
        current.next = new_node
        current = new_node.next

    # Step 2: Copy random pointers
    current = head
    while current:
        if current.random:
            current.next.random = current.random.next
        current = current.next.next

    # Step 3: Separate original and cloned list
    current, cloned_head = head, head.next
    cloned_current = cloned_head
    while current:
        current.next = cloned_current.next
        current = current.next
        if cloned_current.next:
            cloned_current.next = cloned_current.next.next
            cloned_current = cloned_current.next

    return cloned_head
```