ASSIGNMENT 14

1. Given a linked list of **N** nodes such that it may contain a loop.

A loop here means that the last node of the link list is connected to the node at position X(1-based index). If the link list does not have any loop, X=0.

Remove the loop from the linked list, if it is present, i.e. unlink the last node which is forming the loop.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


def detect_and_remove_loop(head):
    if not head or not head.next:
        return

    slow = head
    fast = head

    # Move the slow and fast pointers until they meet
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

        if slow == fast:
            break

    # If there is no loop in the linked list
    if slow != fast:
        return

    # Move the slow pointer back to the head and move both pointers at the same pace
    slow = head

    while slow.next != fast.next:
        slow = slow.next
        fast = fast.next

    # Unlink the last node to remove the loop
    fast.next = None
```

2. A number **N** is represented in Linked List such that each digit corresponds to a node in linked list. You need to add 1 to it.

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


def add_one(head):
    if not head:
        return ListNode(1)

    dummy = ListNode(0)  # Dummy node to handle the case when the most significant digit becomes 0
    dummy.next = head

    current = head
    last_non_nine = dummy

    # Find the last non-nine digit
    while current:
        if current.val != 9:
            last_non_nine = current
        current = current.next

    # Add 1 to the last non-nine digit
    last_non_nine.val += 1

    # Set all digits after the last non-nine digit to 0
    current = last_non_nine.next
    while current:
        current.val = 0
        current = current.next

    # If the most significant digit becomes 0, insert a new node with value 1 at the beginning
    if dummy.val == 0:
        return dummy.next

    return dummy


# Create the linked list representing the number 123
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)

# Add 1 to the number
new_head = add_one(head)

# Print the resulting linked list
current = new_head
while current:
    print(current.val, end=" -> ")
    current = current.next
```

3. Given a Linked List of size N, where every node represents a sub-linked-list and contains two pointers:(i) a **next** pointer to the next node,(ii) a **bottom** pointer to a linked list where this node is head.Each of the sub-linked-list is in sorted order.Flatten the Link List such that all the nodes appear in a single level while maintaining the sorted order. **Note:** The flattened list will be printed using the bottom pointer instead of next pointer.

```
class Node:
    def __init__(self, data=None, next=None, bottom=None):
        self.data = data
        self.next = next
        self.bottom = bottom


def merge_lists(head1, head2):
    if not head1:
        return head2
    if not head2:
        return head1

    result = None

    if head1.data <= head2.data:
        result = head1
        result.bottom = merge_lists(head1.bottom, head2)
    else:
        result = head2
        result.bottom = merge_lists(head1, head2.bottom)

    return result


def flatten_linked_list(head):
    if not head or not head.next:
        return head

    head.next = flatten_linked_list(head.next)

    head = merge_lists(head, head.next)

    return head
```

4. You are given a special linked list with **N** nodes where each node has a next pointer pointing to its next node. You are also given **M** random pointers, where you will be given **M** number of pairs denoting two nodes **a** and **b i.e. a->arb = b** (arb is pointer to random node)**.**

Construct a copy of the given list. The copy should consist of exactly **N** new nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. None of the pointers in the new list should point to nodes in the original list.

For example, if there are two nodes **X** and **Y** in the original list, where **X.arb --> Y**, then for the corresponding two nodes **x** and **y** in the copied list, **x.arb --> y.**

```
class Node:
    def __init__(self, data=None, next=None, random=None):
        self.data = data
        self.next = next
        self.random = random


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

    # Create a hash map to store the mapping of original nodes to copied nodes
    node_map = {}

    # Create a new head node and current pointer for the copied list
    new_head = Node(head.data)
    new_current = new_head

    # Store the mapping of original head to copied head in the hash map
    node_map[head] = new_head

    # Traverse the original list and create copies of each node
    current = head.next
    while current:
        new_node = Node(current.data)
        new_current.next = new_node
        new_current = new_current.next

        # Store the mapping of original node to copied node in the hash map
        node_map[current] = new_node

        current = current.next

    # Update the next and random pointers of the copied nodes
    current = head
    new_current = new_head
    while current:
        new_current.random = node_map.get(current.random)
        current = current.next
        new_current = new_current.next

    return new_head


# Create the special linked list
head = Node(1)
head.next = Node(2)
head.next.next = Node(3)
head.next.next.next = Node(4)
head.next.next.next.next = Node(5)

# Set the random pointers
head.random = head.next.next
```

```
head.next.random = head
head.next.next.random = head.next.next.next.next
head.next.next.next.random = head.next.next
head.next.next.next.next.random = head

# Copy the special linked list
new_head = copy_special_linked_list(head)

# Print the copied list
current = new_head
while current:
    print(f"Data: {current.data}, Random: {current.random.data if current.random else None}")
    current = current.next
```

5.Given the `head` of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return *the reordered list*.

The **first** node is considered **odd**, and the **second** node is **even**, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

You must solve the problem in `O(1)` extra space complexity and `O(n)` time complexity.

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


def odd_even_list(head):
    if not head or not head.next:
        return head

    odd_head = odd_tail = head
    even_head = even_tail = head.next

    current = head.next.next
    is_odd = True

    while current:
        if is_odd:
            odd_tail.next = current
            odd_tail = odd_tail.next
        else:
            even_tail.next = current
            even_tail = even_tail.next
            current = current.next
            is_odd = not is_odd

        odd_tail.next = even_head
       even_tail.next = None
  return odd_head
```

6. Given a singly linked list of size **N**. The task is to **left-shift** the linked list by **k** nodes, where **k** is a given positive integer smaller than or equal to length of the linked list.

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


def left_shift_linked_list(head, k):
    if not head or k == 0:
        return head

    current = head
    length = 0

    # Traverse to find the length of the linked list
    while current.next:
        current = current.next
        length += 1

    current.next = head  # Make the list circular

    shift_count = k % (length + 1)  # Effective number of shifts

    # Traverse to find the new head of the shifted list
    current = head
    for _ in range(shift_count):
        current = current.next

    new_head = current.next
    current.next = None  # Set the end of the shifted list

    return new_head
```

7. You are given the `head` of a linked list with `n` nodes.

For each node in the list, find the value of the **next greater node**. That is, for each node, find the value of the first node that is next to it and has a **strictly larger** value than it.

Return an integer array `answer` where `answer[i]` is the value of the next greater node of the `ith` node (**1-indexed**). If the `ith` node does not have a next greater node, set `answer[i] = 0`.

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


def next_greater_node(head):
    # Convert the linked list to a list
    values = []
    current = head
```

```
    while current:
        values.append(current.val)
        current = current.next

    stack = []  # Stack to store the indices of potential next greater nodes
    result = [0] * len(values)

    for i in range(len(values) - 1, -1, -1):
        while stack and values[stack[-1]] <= values[i]:
            stack.pop()

        if stack:
            result[i] = values[stack[-1]]

        stack.append(i)

    return result
```

8. Given the `head` of a linked list, we repeatedly delete consecutive sequences of nodes that sum to `0` until there are no such sequences.

After doing so, return the head of the final linked list. You may return any such answer.

(Note that in the examples below, all sequences are serializations of `ListNode` objects.)

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


def remove_zero_sum_sublists(head):
    dummy = ListNode(0)
    dummy.next = head

    prefix_sum = 0
    prefix_sums = {}  # Dictionary to store prefix sums and their corresponding nodes

    current = dummy

    while current:
        prefix_sum += current.val

        if prefix_sum in prefix_sums:
            # Remove nodes between prefix sum occurrences
            node = prefix_sums[prefix_sum].next
            while node != current.next:
                prefix_sum += node.val
                del prefix_sums[prefix_sum]
                node = node.next

            # Update previous node's next pointer
```

```
        prefix_sums[prefix_sum].next = current.next

    else:
        prefix_sums[prefix_sum] = current

    current = current.next

return dummy.next
```