# ASSIGNMENT 13

1.Given two linked list of the same size, the task is to create a new linked list using those linked lists. The condition is that the greater node among both linked list will be added to the new linked list.

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


def create_new_linked_list(list1, list2):
    if not list1 or not list2:
        return None

    dummy = ListNode()  # Dummy node to keep track of the new linked list
    current = dummy

    while list1 and list2:
        if list1.val >= list2.val:
            current.next = ListNode(list1.val)
            list1 = list1.next
        else:
            current.next = ListNode(list2.val)
            list2 = list2.next
        current = current.next

    # Add any remaining nodes from list1 or list2
    if list1:
        current.next = list1
    elif list2:
        current.next = list2

    return dummy.next
```

2. Write a function that takes a list sorted in non-decreasing order and deletes any duplicate nodes from the list. The list should only be traversed once.

For example if the linked list is 11->11->11->21->43->43->60 then removeDuplicates() should convert the list to 11->21->43->60.

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


def remove_duplicates(head):
    if not head:
        return head
```

```
    current = head

    while current.next:
        if current.val == current.next.val:
            current.next = current.next.next
        else:
            current = current.next

    return head


# Create the linked list: 11 -> 11 -> 11 -> 21 -> 43 -> 43 -> 60
head = ListNode(11)
head.next = ListNode(11)
head.next.next = ListNode(11)
head.next.next.next = ListNode(21)
head.next.next.next.next = ListNode(43)
head.next.next.next.next.next = ListNode(43)
head.next.next.next.next.next.next = ListNode(60)

# Remove duplicate nodes from the linked list
new_head = remove_duplicates(head)

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

# Output: 11 -> 21 -> 43 -> 60 ->
```

3. Given a linked list of size **N**. The task is to reverse every **k** nodes (where k is an input to the function) in the linked list. If the number of nodes is not a multiple of *k* then left-out nodes, in the end, should be considered as a group and must be reversed (See Example 2 for clarification).

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


def reverse_k_nodes(head, k):
    if not head or k <= 1:
        return head

    # Helper function to reverse a linked list
    def reverse_list(head):
        prev = None
        current = head

        while current:
            next_node = current.next
            current.next = prev
```

```
            prev = current
            current = next_node

        return prev

    dummy = ListNode()  # Dummy node to keep track of the new head of the reversed linked list
    dummy.next = head
    prev_group_end = dummy
    current = head
    count = 0

    while current:
        count += 1

        if count % k == 0:
            next_group_start = current.next  # Start of the next group
            current.next = None  # Disconnect the current group

            # Reverse the current group and connect it to the previous group
            prev_group_end.next = reverse_list(head)
            prev_group_end = head

            head = next_group_start  # Update the head for the next group
            current = next_group_start
        else:
            current = current.next

    # Reverse the left-out nodes (if any)
    prev_group_end.next = reverse_list(head)

    return dummy.next
```

4. Given a linked list, write a function to reverse every alternate k nodes (where k is an input to the function) in an efficient way. Give the complexity of your algorithm.

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


def reverse_alternate_k_nodes(head, k):
    if not head or k <= 1:
        return head

    # Helper function to reverse a linked list
    def reverse_list(head):
        prev = None
        current = head

        while current:
            next_node = current.next
```

```
            current.next = prev
            prev = current
            current = next_node

        return prev

    dummy = ListNode()  # Dummy node to keep track of the new head of the reversed linked list
    dummy.next = head
    prev_group_end = dummy
    current = head
    count = 1
    reverse = True  # Flag to indicate if the current group should be reversed

    while current:
        next_node = current.next

        if count % k == 0:
            if reverse:
                next_group_start = current.next  # Start of the next group
                current.next = None  # Disconnect the current group

                # Reverse the current group and connect it to the previous group
                prev_group_end.next = reverse_list(head)
                prev_group_end = head

                head = next_group_start  # Update the head for the next group
                current = next_group_start
            else:
                prev_group_end.next = current  # Connect the current group to the previous group
                prev_group_end = current

                current = next_node
        else:
            current = next_node

        count += 1
        reverse = not reverse  # Toggle the reverse flag for the next group

    prev_group_end.next = head  # Connect the remaining nodes (if any) to the previous group

    return dummy.next
```

5. Given a linked list and a key to be deleted. Delete last occurrence of key from linked. The list may have duplicates.

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


def delete_last_occurrence(head, key):
    if not head:
        return head
```

```
    dummy = ListNode()  # Dummy node to handle the case of deleting the head node
    dummy.next = head

    prev_to_delete = None
    current = head
    last_occurrence = None

    while current:
        if current.val == key:
            last_occurrence = current
        current = current.next

    # If the last occurrence is found, delete the node
    if last_occurrence:
        current = dummy.next
        while current.next != last_occurrence:
            current = current.next
        current.next = last_occurrence.next

    return dummy.next
```

6. Given two sorted linked lists consisting of **N** and **M** nodes respectively. The task is to merge both of the lists (in place) and return the head of the merged list

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


def merge_lists(head1, head2):
    dummy = ListNode()  # Dummy node to keep track of the merged list
    current = dummy

    while head1 and head2:
        if head1.val <= head2.val:
            current.next = head1
            head1 = head1.next
        else:
            current.next = head2
            head2 = head2.next

        current = current.next

    # Append the remaining nodes of list1 or list2 (if any)
    if head1:
        current.next = head1
    if head2:
        current.next = head2

    return dummy.next
```

7. Given a **Doubly Linked List**, the task is to reverse the given Doubly Linked List.

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


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

    current = head
    new_head = None

    while current:
        # Swap the prev and next pointers of the current node
        temp = current.next
        current.next = current.prev
        current.prev = temp

        # Move to the next node
        new_head = current
        current = temp

    return new_head
```

8. Given a doubly linked list and a position. The task is to delete a node from given position in a doubly linked list.

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


def delete_node(head, position):
    if not head:
        return head

    if position == 1:
        if head.next:
            head.next.prev = None
        return head.next

    current = head
    count = 1

    while current and count < position:
        current = current.next
        count += 1

    if not current:
        return head
```

```
if current.next:
    current.next.prev = current.prev
current.prev.next = current.next

return head
```