Q.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.

Solution :

Define Node:

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.next = None

Define LnkedList Class:

class LinkedList:

def \_\_init\_\_(self):

self.head = None

def add\_node(self, value):

new\_node = Node(value)

if self.head is None:

self.head = new\_node

else:

current = self.head

while current.next is not None:

current = current.next

current.next = new\_node

@staticmethod

def create\_new\_linked\_list(list1, list2):

if list1 is None or list2 is None:

return None

result = LinkedList()

current1 = list1.head

current2 = list2.head

while current1 is not None and current2 is not None:

if current1.value >= current2.value:

result.add\_node(current1.value)

current1 = current1.next

else:

result.add\_node(current2.value)

current2 = current2.next

# Append remaining nodes from list1 or list2, if any

while current1 is not None:

result.add\_node(current1.value)

current1 = current1.next

while current2 is not None:

result.add\_node(current2.value)

current2 = current2.next

return result

Q.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.

Solution :

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.next = None

def remove\_duplicates(head):

if head is None:

return head

current = head

while current.next is not None:

if current.value == current.next.value:

current.next = current.next.next

else:

current = current.next

return head

Q.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).

Solution ;

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.next = None

def reverse\_k\_nodes(head, k):

if head is None or k == 0 or k == 1:

return head

def reverse\_sublist(head, k):

prev = None

current = head

count = 0

while current is not None and count < k:

next\_node = current.next

current.next = prev

prev = current

current = next\_node

count += 1

if next\_node is not None:

head.next = reverse\_sublist(next\_node, k)

return prev

new\_head = reverse\_sublist(head, k)

return new\_head

Q.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.

Solution :

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.next = None

def reverse\_alternate\_k\_nodes(head, k):

if head is None or k <= 1:

return head

def reverse\_group(node, k, reverse):

if node is None:

return None

current = node

prev = None

count = 0

# Reverse k nodes

while current is not None and count < k:

next\_node = current.next

if reverse:

current.next = prev

prev = current

current = next\_node

count += 1

# Connect the reversed group back to the main linked list

if reverse:

node.next = reverse\_group(current, k, not reverse)

return prev

else:

prev.next = reverse\_group(current, k, not reverse)

return node

return reverse\_group(head, k, True)

Q.5] <aside> 💡 **Question 5**

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

</aside>

Solution:

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.next = None

def delete\_last\_occurrence(head, key):

if head is None:

return head

# Find the last occurrence of the key

last\_occurrence = None

prev = None

current = head

while current is not None:

if current.value == key:

last\_occurrence = current

prev = current

current = current.next

if last\_occurrence is None:

return head # Key not found

if last\_occurrence == head:

return head.next # Key is the head node

prev.next = last\_occurrence.next

return head

Q.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.

Solution :

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.next = None

def merge\_sorted\_lists(head1, head2):

if head1 is None:

return head2

if head2 is None:

return head1

dummy = Node(None)

current = dummy

while head1 is not None and head2 is not None:

if head1.value <= head2.value:

current.next = head1

head1 = head1.next

else:

current.next = head2

head2 = head2.next

current = current.next

if head1 is not None:

current.next = head1

if head2 is not None:

current.next = head2

return dummy.next

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

Solution :

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.prev = None

self.next = None

def reverse\_doubly\_linked\_list(head):

if head is None or head.next is None:

return head

current = head

while current is not None:

# Swap prev and next pointers for the current node

temp = current.prev

current.prev = current.next

current.next = temp

# Move to the next node

current = current.prev

# Update the head and tail pointers

if temp is not None:

head = temp.prev

return head

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

Solution :

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.prev = None

self.next = None

def delete\_node\_at\_position(head, position):

if head is None:

return head

if position == 1:

new\_head = head.next

if new\_head is not None:

new\_head.prev = None

return new\_head

current = head

count = 1

while current is not None and count < position:

current = current.next

count += 1

if current is None:

return head

prev\_node = current.prev

prev\_node.next = current.next

if current.next is not None:

current.next.prev = prev\_node

del current

return head