Q.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.

Solution :

def remove\_loop(head):

if not head or not head.next:

return head

slow = head

fast = head

# Detect loop

while fast and fast.next:

slow = slow.next

fast = fast.next.next

if slow == fast:

break

# No loop found

if slow != fast:

return head

# Move slow to head

slow = head

# Find the node where the loop starts

while slow.next != fast.next:

slow = slow.next

fast = fast.next

# Remove the loop

fast.next = None

return head

Q.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.

Solution :

class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

def addOne(head):

# Reverse the linked list

prev = None

current = head

while current:

next\_node = current.next

current.next = prev

prev = current

current = next\_node

head = prev

# Add 1 to the linked list

carry = 1

current = head

while current:

sum = current.val + carry

current.val = sum % 10

carry = sum // 10

if carry == 0:

break

current = current.next

# If there is still a carry, add a new node

if carry == 1:

new\_node = ListNode(1)

current.next = new\_node

# Reverse the linked list back to its original order

prev = None

current = head

while current:

next\_node = current.next

current.next = prev

prev = current

current = next\_node

head = prev

return head

Q.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.

Solution :

class Node:

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

self.data = data

self.next = None

self.bottom = None

def merge(left, right):

if not left:

return right

if not right:

return left

result = None

if left.data <= right.data:

result = left

result.bottom = merge(left.bottom, right)

else:

result = right

result.bottom = merge(left, right.bottom)

return result

def flatten(head):

if not head or not head.next:

return head

# Recursively flatten the rest of the list

head.next = flatten(head.next)

# Merge the flattened list with the current list

head = merge(head, head.next)

return head

Q.4] <aside> 💡 **Question 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.**

Return the head of the copied linked list.

</aside>

Solution:

class Node:

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

self.data = data

self.next = None

self.random = None

def copyRandomList(head):

if not head:

return None

# Create new nodes and insert them between original nodes

current = head

while current:

new\_node = Node(current.data)

new\_node.next = current.next

current.next = new\_node

current = new\_node.next

# Update random pointers of the new nodes

current = head

while current:

if current.random:

current.next.random = current.random.next

current = current.next.next

# Separate the modified list into two separate lists

new\_head = head.next

current = head

while current:

new\_node = current.next

current.next = new\_node.next

if new\_node.next:

new\_node.next = new\_node.next.next

current = current.next

return new\_head

Q.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.**

Return the head of the copied linked list.

Solution :

class Node:

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

self.data = data

self.next = None

self.random = None

def copyRandomList(head):

if not head:

return None

node\_map = {} # HashMap to map original nodes to copied nodes

# Create new nodes and map them to the original nodes

current = head

while current:

new\_node = Node(current.data)

node\_map[current] = new\_node

current = current.next

# Set the next and random pointers of the copied nodes

current = head

while current:

copied\_node = node\_map[current]

copied\_node.next = node\_map.get(current.next)

copied\_node.random = node\_map.get(current.random)

current = current.next

return node\_map[head]

Q.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.

Solution :

class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

def oddEvenList(head):

if not head or not head.next:

return head

odd\_head = head

even\_head = head.next

odd\_current = odd\_head

even\_current = even\_head

current = even\_head.next

index = 3

while current:

if index % 2 == 1:

odd\_current.next = current

odd\_current = odd\_current.next

else:

even\_current.next = current

even\_current = even\_current.next

current = current.next

index += 1

odd\_current.next = even\_head

even\_current.next = None

return odd\_head

Q.6] <aside> 💡 **Question 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.

</aside>

Solution :

class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

def leftShiftLinkedList(head, k):

if not head or k == 0:

return head

length = 0

current = head

while current:

length += 1

current = current.next

k = k % length

if k == 0:

return head

current = head

for \_ in range(k - 1):

current = current.next

new\_head = current.next

current.next = None

current = new\_head

while current.next:

current = current.next

current.next = head

return new\_head

Q.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.

Solution :

class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

def nextLargerNodes(head):

stack = []

result = []

index = 0

# Traverse the linked list in reverse order

while head:

# Pop nodes with smaller values from the stack

while stack and stack[-1][1] <= head.val:

stack.pop()

# Set the next greater value for the current node

if stack:

result.append(stack[-1][1])

else:

result.append(0)

# Push the current node onto the stack

stack.append((index, head.val))

# Move to the next node

head = head.next

index += 1

# Reverse the result list

result.reverse()

return result

Q.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.)

Solution :

class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

def removeZeroSumSublists(head):

dummy = ListNode(0)

dummy.next = head

prefix\_sum = 0

prefix\_sum\_map = {}

current = dummy

while current:

prefix\_sum += current.val

if prefix\_sum in prefix\_sum\_map:

# Remove nodes between previous occurrence and current node

prev = prefix\_sum\_map[prefix\_sum].next

node\_sum = prefix\_sum + prev.val

while prev != current:

node\_sum += prev.val

del prefix\_sum\_map[node\_sum]

prev = prev.next

# Update the next pointer of the previous occurrence

prefix\_sum\_map[prefix\_sum].next = current.next

else:

prefix\_sum\_map[prefix\_sum] = current

current = current.next

return dummy.next