**1. Define a doubly linked list**.

A doubly linked list is a type of linked list where each node contains not only a reference to the next node in the sequence but also a reference to the previous node. This means that each node in the list has three components: the data it holds, a pointer to the next node, and a pointer to the previous node.

In a doubly linked list, the first node is called the head, and the last node is called the tail. The head's previous pointer and the tail's next pointer point to null, indicating the boundaries of the list.

Here's a simple Python-like pseudo code representation of a doubly linked list node:

class Node:

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

self.data = data

self.prev = None

self.next = None

In this representation, self.data represents the value stored in the node, self.prev points to the previous node in the list (or None if it's the head), and self.next points to the next node in the list (or None if it's the tail).

**2. Write a function to reverse 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\_node = None

current\_node = head

while current\_node is not None:

next\_node = current\_node.next # Store the next node

current\_node.next = prev\_node # Reverse the link

# Move pointers one position ahead

prev\_node = current\_node

current\_node = next\_node

# After the loop, prev\_node will be the new head of the reversed list

return prev\_node

**3. Detect cycle in a linked list.**

class ListNode:

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

self.val = val

self.next = next

def has\_cycle(head):

if head is None:

return False

slow\_pointer = head

fast\_pointer = head

while fast\_pointer is not None and fast\_pointer.next is not None:

slow\_pointer = slow\_pointer.next

fast\_pointer = fast\_pointer.next.next

# If the pointers meet, there's a cycle

if slow\_pointer == fast\_pointer:

return True

return False

**4. Merge two sorted linked list into one**.

**1->3->5->6-> null and 2->4->6->8-> null should be merged to make 1->2->3->4->5->6->7->8**

class ListNode:

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

self.val = val

self.next = next

def mergeTwoLists(l1,l2):

dummy = ListNode(0)

current = dummy

while l1 and l2:

if l1.val < l2.val:

current.next = l1

l1 = l1.next

else:

current.next = l2

l2 = l2.next

current = current.next

#Attach remaining nodes from either list, if any

if l1:

current.next = l1

elif l2:

current.next = l2

return dummy.next

# Example usage

#Create the first linked list : 1->3->5->6->None

l1 = ListNode(1)

l1.next = ListNode(3)

l1.next.next = ListNode(5)

l1.next.next.next = ListNode(6)

#create the second linked list : 2->4->6->8->None

l2 = ListNode(2)

l2.next = ListNode(4)

l2.next.next = ListNode(6)

l2.next.next.next = ListNode(8)

merged\_list = mergeTwoLists(l1,l2)

#print the merged list

while merged\_list:

print(merged\_list.val,end='->' if merged\_list.next else '->None')

merged\_list = merged\_list.next

**output : 1->2->3->4->5->6->6->8->None**

**5. Write a function to remove nth node from the end in a linked list**

**1->2->3->4->5->6 should be changed to 1->2->3->4->6**

class ListNode:

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

self.val = val

self.next = next

def removeNthFromEnd(head,n):

#Create a dummy node to handle edge cases

dummy = ListNode(0)

dummy.next = head

fast = slow = dummy

#move the fast pointer n steps ahead

for \_ in range(n):

fast = fast.next

#move both pointers until the fast pointer reaches the end

while fast.next:

fast = fast.next

slow = slow.next

#Remove the nth node from the end

slow.next = slow.next.next

return dummy.next

#helper function to print the linked list

def printLinkedList(head,n):

while head:

print(head.val,end='->' if head.next else '->None')

head = head.next

#Example usage

#Create the linked list : 1->2->3->3->4->4->4->5->None

head = ListNode(1)

head.next = ListNode(2)

head.next.next = ListNode(3)

head.next.next.next= ListNode(4)

head.next.next.next.next = ListNode(5)

head.next.next.next.next.next = ListNode(6)

n = 2

head = removeNthFromEnd(head,n)

#Print the modified linked list

printLinkedList(head,n)

**output : 1->2->3->4->6->None**

**6. Remove duplicates from a sorted linked list**

**1->2->3->3->4->4->4->5 should be changed to 1->2->3->4->5**

class ListNode:

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

self.val = val

self.next = next

def deleteDuplicates(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

#Function to print the linked list

def printLinkedList(head):

current = head

while current:

print(current.val,end='->')

current = current.next

print()

#Example usage:

#Create the linked list

head = ListNode(1)

head.next = ListNode(2)

head.next.next = ListNode(3)

head.next.next.next = ListNode(3)

head.next.next.next.next = ListNode(4)

head.next.next.next.next.next = ListNode(4)

head.next.next.next.next.next.next = ListNode(4)

head.next.next.next.next.next.next.next = ListNode(5)

print('Original linked list:')

printLinkedList(head,)

head = deleteDuplicates(head)

print("linked list after removing duplicates:")

printLinkedList(head)

**output: Original linked list:**

**1->2->3->3->4->4->4->5->**

**linked list after removing duplicates:**

**1->2->3->4->5->**

**7. Find the intersection of the two linked lists**

**1->2->3->4->8->6->9->5->1->6->7, intersection 1->6**

class ListNode:

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

self.val = val

self.next = next

def getIntersectionNode(headA, headB):

if not headA or not headB:

return None

# Function to get the length of the linked list

def getLength(head):

length = 0

while head:

length += 1

head = head.next

return length

# Get the lengths and tails of both linked lists

lenA = getLength(headA)

lenB = getLength(headB)

tailA = headA

tailB = headB

while tailA.next:

tailA = tailA.next

while tailB.next:

tailB = tailB.next

# If tails are not the same, no intersection

if tailA != tailB:

return None

# Reset the pointers to the heads of the linked lists

currA = headA

currB = headB

# Advance the pointer of the longer linked list by the difference in lengths

if lenA > lenB:

for \_ in range(lenA - lenB):

currA = currA.next

elif lenB > lenA:

for \_ in range(lenB - lenA):

currB = currB.next

# Traverse both linked lists simultaneously until intersection is found

while currA != currB:

currA = currA.next

currB = currB.next

return currA

# Function to print the linked list

def printLinkedList(head):

current = head

while current:

print(current.val, end=" ")

current = current.next

print()

# Example usage:

# Create the first linked list

headA = ListNode(1)

headA.next = ListNode(2)

headA.next.next = ListNode(3)

headA.next.next.next = ListNode(4)

headA.next.next.next.next = ListNode(8)

headA.next.next.next.next.next = ListNode(6)

headA.next.next.next.next.next.next = ListNode(9)

headA.next.next.next.next.next.next.next = ListNode(5)

headA.next.next.next.next.next.next.next.next = ListNode(1)

headA.next.next.next.next.next.next.next.next.next = ListNode(6)

headA.next.next.next.next.next.next.next.next.next.next = ListNode(7)

# Create the second linked list

headB = ListNode(1)

headB.next = ListNode(6)

# Set intersection node

intersection = headA.next.next.next.next # Intersection at value 8

# Connect the intersection node

headB.next.next = intersection

print("First linked list:")

printLinkedList(headA)

print("Second linked list:")

printLinkedList(headB)

intersection\_node = getIntersectionNode(headA, headB)

if intersection\_node:

print("Intersection node value:", intersection\_node.val)

else:

print("No intersection")

**output: First linked list:**

**1 2 3 4 8 6 9 5 1 6 7**

**Second linked list:**

**1 6 8 6 9 5 1 6 7**

**Intersection node value: 8**

**8. Rotate a linked list by k positions to the right**

**1->2->3->4->8->6->9 , after rotating for 2 times becomes,3->4->8->6->9->1->2**

class ListNode:

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

self.val = val

self.next = next

def rotateRight(head, k):

if not head or k == 0:

return head

# Find the length of the linked list

length = 1

tail = head

while tail.next:

tail = tail.next

length += 1

# Calculate the effective rotation count

k %= length

if k == 0:

return head

# Find the new head position

new\_head\_index = length - k

new\_tail\_index = new\_head\_index - 1

# Traverse to the new tail

new\_tail = head

for \_ in range(new\_tail\_index):

new\_tail = new\_tail.next

# Perform rotation

new\_head = new\_tail.next

new\_tail.next = None

tail.next = head

return new\_head

# Function to print the linked list

def printLinkedList(head):

current = head

while current:

print(current.val, end=" ")

current = current.next

print()

# Example usage:

# Create the linked list

head = ListNode(1)

head.next = ListNode(2)

head.next.next = ListNode(3)

head.next.next.next = ListNode(4)

head.next.next.next.next = ListNode(8)

head.next.next.next.next.next = ListNode(6)

head.next.next.next.next.next.next = ListNode(9)

print("Original linked list:")

printLinkedList(head)

k = 2

head = rotateRight(head, k)

print(f"Linked list after rotating {k} times to the right:")

printLinkedList(head)

**output: Original linked list:**

**1 2 3 4 8 6 9**

**Linked list after rotating 2 times to the right:**

**6 9 1 2 3 4 8**

**9. Add two numbers represented by linkedlists.**

**Given two non-empty linked lists representing two non-negative integers, where the digits are stored in reverse order, add the two numbers and return it as a linked list.**

class ListNode:

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

self.val = val

self.next = next

def addTwoNumbers(l1, l2):

dummy = ListNode(0)

current = dummy

carry = 0

while l1 or l2:

x = l1.val if l1 else 0

y = l2.val if l2 else 0

sum\_val = x + y + carry

carry = sum\_val // 10

current.next = ListNode(sum\_val % 10)

current = current.next

if l1:

l1 = l1.next

if l2:

l2 = l2.next

if carry > 0:

current.next = ListNode(carry)

return dummy.next

# Example usage:

# Create linked lists

l1 = ListNode(2)

l1.next = ListNode(4)

l1.next.next = ListNode(3)

l2 = ListNode(5)

l2.next = ListNode(6)

l2.next.next = ListNode(4)

# Call function to add two numbers represented by linked lists

result = addTwoNumbers(l1, l2)

# Print the result

while result:

print(result.val, end=" ")

result = result.next

**output: 7 0 8**

**10. Clone a linked list with next and random pointer**.

Given a linked list of size N where each node has two links: one pointer points to the next node and the second pointer points to any node in the list. The task is to create a clone of this linked list in O(N) time.

Note: the pointer pointing to the next node is 'next' pointer and the one pointing to an arbitrary node is called 'arbit' pointer as it can point to any arbitrary node in the linked list.

class Node:

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

self.data = data

self.next = None

self.arbitrary = None

def cloneLinkedList(head):

if not head:

return None

current = head

# Step 1: Insert a new node after each original node

while current:

new\_node = Node(current.data)

new\_node.next = current.next

current.next = new\_node

current = new\_node.next

# Step 2: Set arbitrary pointers for new nodes

current = head

while current:

if current.arbitrary:

current.next.arbitrary = current.arbitrary.next

current = current.next.next

# Step 3: Extract the cloned linked list

cloned\_head = head.next

cloned\_current = cloned\_head

current = head

while current:

current.next = cloned\_current.next

if cloned\_current.next:

cloned\_current.next = cloned\_current.next.next

current = current.next

cloned\_current = cloned\_current.next

return cloned\_head

# Example usage:

# Create the original linked list

a = Node(1)

b = Node(2)

c = Node(3)

d = Node(4)

a.next = b

b.next = c

c.next = d

a.arbitrary = c # Arbitrary pointer for node 'a'

b.arbitrary = d # Arbitrary pointer for node 'b'

c.arbitrary = b # Arbitrary pointer for node 'c'

d.arbitrary = a # Arbitrary pointer for node 'd'

# Clone the linked list

cloned\_head = cloneLinkedList(a)

# Print the cloned linked list

current = cloned\_head

while current:

print("Data:", current.data)

print("Arbitrary Data:", current.arbitrary.data if current.arbitrary else None)

print("---")

current = current.next

**output: Data: 1**

**Arbitrary Data: 3**

**---**

**Data: 2**

**Arbitrary Data: 4**

**---**

**Data: 3**

**Arbitrary Data: 2**

**---**

**Data: 4**

**Arbitrary Data: 1**

**---**