Q.1] Given a singly linked list, delete **middle**of the linked list. For example, if given linked list is 1->2->**3**->4->5 then linked list should be modified to 1->2->4->5.If there are **even** nodes, then there would be **two middle**nodes, we need to delete the second middle element. For example, if given linked list is 1->2->3->4->5->6 then it should be modified to 1->2->3->5->6.If the input linked list is NULL or has 1 node, then it should return NULL

Solution :

class Node:

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

self.data = data

self.next = None

def delete\_middle(head):

if head is None or head.next is None:

return None

slow\_ptr = head

fast\_ptr = head

prev\_ptr = None

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

fast\_ptr = fast\_ptr.next.next

prev\_ptr = slow\_ptr

slow\_ptr = slow\_ptr.next

prev\_ptr.next = slow\_ptr.next

slow\_ptr = None

return head

# Helper function to print the linked list

def print\_list(head):

current = head

while current is not None:

print(current.data, end=" ")

current = current.next

print()

# Example usage

# 1->2->3->4->5

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)

print("Original linked list:")

print\_list(head)

head = delete\_middle(head)

print("Linked list after deleting middle node:")

print\_list(head)

Q.2] Given a linked list of **N** nodes. The task is to check if the linked list has a loop. Linked list can contain self loop.

Solution :

class Node:

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

self.data = data

self.next = None

def detect\_loop(head):

slow\_ptr = head

fast\_ptr = head

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

slow\_ptr = slow\_ptr.next

fast\_ptr = fast\_ptr.next.next

if slow\_ptr == fast\_ptr:

return True

return False

# Example usage

# Create a linked list with a loop: 1->2->3->4->5->3

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)

head.next.next.next.next.next = head.next.next

has\_loop = detect\_loop(head)

if has\_loop:

print("The linked list has a loop")

else:

print("The linked list does not have a loop")

Q.3] Given a linked list consisting of **L** nodes and given a number **N**. The task is to find the **N**th node from the end of the linked list.

Solution :

class Node:

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

self.data = data

self.next = None

def get\_nth\_node\_from\_end(head, n):

first\_ptr = head

second\_ptr = head

# Move first\_ptr n positions ahead

for \_ in range(n):

if first\_ptr is None:

# The linked list has fewer than n nodes

return None

first\_ptr = first\_ptr.next

# Move both pointers until first\_ptr reaches the end

while first\_ptr is not None:

first\_ptr = first\_ptr.next

second\_ptr = second\_ptr.next

return second\_ptr

# Helper function to print the linked list

def print\_list(head):

current = head

while current is not None:

print(current.data, end=" ")

current = current.next

print()

# Example usage

# Create a linked list: 1->2->3->4->5->6->7

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)

head.next.next.next.next.next = Node(6)

head.next.next.next.next.next.next = Node(7)

print("Original linked list:")

print\_list(head)

n = 3

nth\_node = get\_nth\_node\_from\_end(head, n)

if nth\_node is not None:

print(f"The {n}th node from the end is: {nth\_node.data}")

else:

print(f"The linked list has fewer than {n} nodes")

Q.4] Given a singly linked list of characters, write a function that returns true if the given list is a palindrome, else false.

Solution :

class Node:

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

self.data = data

self.next = None

def is\_palindrome(head):

if head is None or head.next is None:

return True

stack = []

slow\_ptr = head

fast\_ptr = head

length = 0

# Traverse the linked list to find its length and store characters in a stack

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

stack.append(slow\_ptr.data)

slow\_ptr = slow\_ptr.next

fast\_ptr = fast\_ptr.next.next

length += 2

# Handle odd length linked list

if fast\_ptr is not None:

length += 1

# Compare characters with the popped elements from the stack

while slow\_ptr is not None:

if slow\_ptr.data != stack.pop():

return False

slow\_ptr = slow\_ptr.next

return True

# Helper function to print the linked list

def print\_list(head):

current = head

while current is not None:

print(current.data, end=" ")

current = current.next

print()

# Example usage

# Create a linked list: 'r' -> 'a' -> 'd' -> 'a' -> 'r'

head = Node('r')

head.next = Node('a')

head.next.next = Node('d')

head.next.next.next = Node('a')

head.next.next.next.next = Node('r')

print("Original linked list:")

print\_list(head)

if is\_palindrome(head):

print("The linked list is a palindrome")

else:

print("The linked list is not a palindrome")

Q.5] 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 :

class Node:

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

self.data = data

self.next = None

def detect\_loop(head):

slow\_ptr = head

fast\_ptr = head

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

slow\_ptr = slow\_ptr.next

fast\_ptr = fast\_ptr.next.next

if slow\_ptr == fast\_ptr:

return slow\_ptr # Return the meeting point

return None # No loop found

def remove\_loop(head):

meeting\_point = detect\_loop(head)

if meeting\_point is None:

return head # No loop, so return the original list

# Move one pointer to the head and the other pointer from the meeting point,

# both pointers move one step at a time until they meet again

ptr1 = head

ptr2 = meeting\_point

while ptr1.next != ptr2.next:

ptr1 = ptr1.next

ptr2 = ptr2.next

# Break the loop by setting the next pointer of the last node in the loop to None

ptr2.next = None

return head

# Helper function to create a loop in the linked list

def create\_loop(head, pos):

if pos == 0:

return head

count = 1

current = head

while count < pos and current.next is not None:

current = current.next

count += 1

if current.next is None:

return head

last\_node = current

while last\_node.next is not None:

last\_node = last\_node.next

last\_node.next = current

return head

# Helper function to print the linked list

def print\_list(head):

current = head

while current is not None:

print(current.data, end=" ")

current = current.next

print()

# Example usage

# Create a linked list: 1->2->3->4->5->6->7 with a loop at position 4 (connecting last node to node at position 4)

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)

head.next.next.next.next.next = Node(6)

head.next.next.next.next.next.next = Node(7)

Q.6] Given a linked list and two integers M and N. Traverse the linked list such that you retain M nodes then delete next N nodes, continue the same till end of the linked list.

Difficulty Level: Rookie

Solution:

class ListNode:

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

self.val = val

self.next = next

def traverse\_and\_delete(head, M, N):

if not head or M <= 0 or N <= 0:

return head

# Helper pointers to traverse and delete nodes

current\_node = head

previous\_node = None

# Traverse the linked list

while current\_node:

# Retain M nodes

for \_ in range(M):

if not current\_node:

return head

previous\_node = current\_node

current\_node = current\_node.next

# Delete N nodes

for \_ in range(N):

if not current\_node:

break

current\_node = current\_node.next

# Adjust the previous node's next pointer

previous\_node.next = current\_node

return head

# Example usage

# Create a linked list: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 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)

M = 2

N = 1

new\_head = traverse\_and\_delete(head, M, N)

# Print the resulting linked list: 1 -> 2 -> 4 -> 6 -> None

current\_node = new\_head

while current\_node:

print(current\_node.val, end=" -> ")

current\_node = current\_node.next

print("None")

Q.7] Given two linked lists, insert nodes of second list into first list at alternate positions of first list. For example, if first list is 5->7->17->13->11 and second is 12->10->2->4->6, the first list should become 5->12->7->10->17->2->13->4->11->6 and second list should become empty. The nodes of second list should only be inserted when there are positions available. For example, if the first list is 1->2->3 and second list is 4->5->6->7->8, then first list should become 1->4->2->5->3->6 and second list to 7->8.

Use of extra space is not allowed (Not allowed to create additional nodes), i.e., insertion must be done in-place. Expected time complexity is O(n) where n is number of nodes in first list.

Solution:

class ListNode:

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

self.val = val

self.next = next

def insert\_alternate(head1, head2):

if not head2:

return head1

current1 = head1

current2 = head2

while current1 and current2:

next1 = current1.next

next2 = current2.next

current1.next = current2

current2.next = next1

current1 = next1

current2 = next2

return head1

# Example usage

# First linked list: 5 -> 7 -> 17 -> 13 -> 11

head1 = ListNode(5)

head1.next = ListNode(7)

head1.next.next = ListNode(17)

head1.next.next.next = ListNode(13)

head1.next.next.next.next = ListNode(11)

# Second linked list: 12 -> 10 -> 2 -> 4 -> 6

head2 = ListNode(12)

head2.next = ListNode(10)

head2.next.next = ListNode(2)

head2.next.next.next = ListNode(4)

head2.next.next.next.next = ListNode(6)

new\_head1 = insert\_alternate(head1, head2)

# Print the resulting first linked list: 5 -> 12 -> 7 -> 10 -> 17 -> 2 -> 13 -> 4 -> 11 -> 6

current\_node = new\_head1

while current\_node:

print(current\_node.val, end=" -> ")

current\_node = current\_node.next

print("None")

# Print the resulting second linked list: None

new\_head2 = head2

current\_node = new\_head2

while current\_node:

print(current\_node.val, end=" -> ")

current\_node = current\_node.next

print("None")

Q.8] Given a singly linked list, find if the linked list is [circular](https://www.geeksforgeeks.org/circular-linked-list/amp/) or not.

A linked list is called circular if it is not NULL-terminated and all nodes are connected in the form of a cycle. Below is an example of a circular linked list.

Solution :

class ListNode:

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

self.val = val

self.next = next

def is\_circular(head):

if not head:

return False

tortoise = head

hare = head.next

while hare and hare.next:

if tortoise == hare:

return True

tortoise = tortoise.next

hare = hare.next.next

return False

# Example usage

# Create a circular linked list: 1 -> 2 -> 3 -> 4 -> 5 -> 2 (points back to the second node)

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 = head.next

is\_circular\_list = is\_circular(head)

print(is\_circular\_list) # True