Q1. Remove middle element

class Solution:

def deleteMiddle(self, head: Optional[ListNode]) -> Optional[ListNode]:

if not head.next:

return None

slow, fast = head, head

mid = None

while fast and fast.next:

mid = slow

slow = slow.next

fast = fast.next.next

mid.next = mid.next.next

return head

Q2. Is cycle

class Solution:

def hasCycle(self, head: Optional[ListNode]) -> bool:

if not head:

return False

slow,fast=head, head

while(fast and fast.next):

slow=slow.next

fast=fast.next.next

if fast==slow:

return True

return False

Q3. Nth node from last

**def** printNthFromLast(self, n):

        temp **=** self.head

        length **=** 0

**while** temp **is** **not** None:

            temp **=** temp.next

            length **+=** 1

        # Print count

**if** n > length:

**print**('Location is greater than the' **+**

                  ' length of LinkedList')

**return**

        temp **=** self.head

**for** i **in** range(0, length **-** n):

            temp **=** temp.next

**print**(temp.data)

Q4. Given a singly linked list of characters, write a function that returns true if the given list is a palindrome, else false.

class Solution:

  def isPalindrome(self, head: ListNode) -> bool:

    def reverseList(head: ListNode) -> ListNode:

      prev = None

      curr = head

      while curr:

        next = curr.next

        curr.next = prev

        prev = curr

        curr = next

      return prev

    slow = head

    fast = head

    while fast and fast.next:

      slow = slow.next

      fast = fast.next.next

    if fast:

      slow = slow.next

    slow = reverseList(slow)

    while slow:

      if slow.val != head.val:

        return False

      slow = slow.next

      head = head.next

    return True

Q5. 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.

**class** LinkedList:

    # Function to initialize head

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

        self.head **=** None

    # Function to insert a new node at the beginning

**def** push(self, new\_data):

        new\_node **=** Node(new\_data)

        new\_node.next **=** self.head

        self.head **=** new\_node

**def** detectAndRemoveLoop(self):

**if** self.head **is** None:

**return**

**if** self.head.next **is** None:

**return**

      slow\_p **=** self.head

      fast\_p **=** self.head

**while**(slow\_p **and** fast\_p **and** fast\_p.next):

          slow\_p **=** slow\_p.next

          fast\_p **=** fast\_p.next.next

          # If slow\_p and fast\_p meet at some point then

          # there is a loop

**if** slow\_p **==** fast\_p:

            slow\_p **=** self.head

              # Finding the beginning of the loop

**while** (slow\_p.next !**=** fast\_p.next):

              slow\_p **=** slow\_p.next

              fast\_p **=** fast\_p.next

                # Sinc fast.next is the looping point

            fast\_p.next **=** None  # Remove loop

    # Utility function to print the LinkedList

**def** printList(self):

        temp **=** self.head

**while**(temp):

**print**(temp.data, end **=** ' ')

            temp **=** temp.next

Q6. 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

**class** LinkedList:

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

        self.head **=** None

**def** push(self, new\_data):

        new\_node **=** Node(new\_data)

        new\_node.next **=** self.head

        self.head **=** new\_node

**def** printList(self):

        temp **=** self.head

**while**(temp):

**print** (temp.data,end**=**" ")

            temp **=** temp.next

**def** skipMdeleteN(self, M, N):

        curr **=** self.head

**while**(curr):

            # Skip M nodes

**for** count **in** range(1, M):

**if** curr **is** None:

**return**

                curr **=** curr.next

**if** curr **is** None :

**return**

            t **=** curr.next

**for** count **in** range(1, N**+**1):

**if** t **is** None:

**break**

                t **=** t.next

            curr.next **=** t

            curr **=** t

Q7. 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.

**class** LinkedList(object):

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

        self.head **=** None

**def** push(self, new\_data:int):

        new\_node **=** Node(new\_data)

        new\_node.next **=** self.head

        self.head **=** new\_node

**def** printList(self):

        temp **=** self.head

**while** temp !**=** None:

**print**(temp.data)

            temp **=** temp.next

**def** merge(self, p, q):

        p\_curr **=** p.head

        q\_curr **=** q.head

**while** p\_curr !**=** None **and** q\_curr !**=** None:

            p\_next **=** p\_curr.next

            q\_next **=** q\_curr.next

            q\_curr.next **=** p\_next

            p\_curr.next **=** q\_curr

            p\_curr **=** p\_next

            q\_curr **=** q\_next

            q.head **=** q\_curr

Q8. Given a singly linked list, find if the linked list is [circular](https://www.geeksforgeeks.org/circular-linked-list/amp/) or not.

**class** LinkedList:

    # Function to initialize head

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

        self.head **=** None

**def** Circular(head):

**if** head **==** None:

**return** True

    # Next of head

    node **=** head.next

    i **=** 0

**while**((node **is** **not** None) **and** (node **is** **not** head)):

        i **=** i **+** 1

        node **=** node.next

**return**(node **==** head)