```
# Defining a Double Linked List.
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node

    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node

    def print_forward(self):
        current = self.head
        while current:
            print(current.data)
            current = current.next

    def print_backward(self):
        current = self.tail
        while current:
            print(current.data)
            current = current.prev

dll = DoublyLinkedList()
dll.append(1)
dll.append(2)
dll.append(3)
dll.prepend(0)
print("Forward traversal:")
dll.print_forward()
print("\nBackward traversal:")
dll.print_backward()
```

```
Forward traversal:
0
1
2
3

Backward traversal:
3
2
1
0
```

```
# Write a function to reverse a linked list in-place
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node

    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node

    def delete(self, data):
        current = self.head
        while current:
            if current.data == data:
                if current.prev:
                    current.prev.next = current.next
                if current.next:
                    current.next.prev = current.prev
                if current == self.head:
                    self.head = current.next
                if current == self.tail:
                    self.tail = current.prev
                return
            current = current.next

    def traverse_forward(self):
        current = self.head
        while current:
            print(current.data, end=" ")
            current = current.next
        print("None")

    def traverse_backward(self):
        current = self.tail
        while current:
            print(current.data, end=" ")
            current = current.prev
        print("None")

    def reverse(self):
        current = self.head
        temp = None
        while current:
            temp = current.prev
            current.prev = current.next
            current.next = temp
            current = current.prev

        if temp is not None:
            self.head = temp.prev

dll = DoublyLinkedList()
dll.append(1)
dll.append(2)
dll.append(3)
dll.append(4)
print("Original list:")
dll.traverse_forward()
dll.reverse()
print("Reversed list:")
dll.traverse_forward()
```

```
Original list:
1 2 3 4 None
4 3 2 1 None
Reversed list:
4 3 2 1 None
```

```
# Create a cycle in a linked list.
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setNext(self, next_node):
        self.next = next_node

def IsCyclic(head):
    fast = head
    while fast and fast.getNext():
        fast = fast.getNext().getNext()
        if fast and fast.getData() == fast:
            return True
    return False

head = Node(1)
node2 = Node(2)
node3 = Node(3)
node4 = Node(4)
head.setNext(node2)
node2.setNext(node3)
node3.setNext(node4)
node4.setNext(node2)
print("Is Cyclic:", IsCyclic(head))
```

```
True
```

```
# Reverse the sorted linked list into one
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setNext(self, next_node):
        self.next = next_node

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.getNext():
            last = last.getNext()
        last.setNext(new_node)

    def print_list(self):
        current = self.head
        while current.getData() is not None:
            current = current.getNext()
        print("None")

def merge_sorted_lists(list1, list2):
    dummy = Node(0)
    tail = dummy

    l1 = list1.head
    l2 = list2.head

    while l1 and l2:
        if l1.getData() <= l2.getData():
            tail.setNext(l1)
            l1 = l1.getNext()
        else:
            tail.setNext(l2)
            l2 = l2.getNext()
        tail = tail.getNext()

    if l1:
        tail.setNext(l1)
    if l2:
        tail.setNext(l2)

    merged_list = LinkedList()
    merged_list.head = dummy.getNext()

    return merged_list

# Creating first sorted linked list
list1 = LinkedList()
list1.append(1)
list1.append(3)
list1.append(5)
print("First sorted list:")
list1.print_list()
# Creating second sorted linked list
list2 = LinkedList()
list2.append(2)
list2.append(4)
list2.append(6)
print("Second sorted list:")
list2.print_list()
# Merging the lists
merged_list = merge_sorted_lists(list1, list2)
print("Merged sorted list:")
merged_list.print_list()
```

```
First sorted list:
1 3 5 None
Second sorted list:
2 4 6 None
Merged sorted list:
1 2 3 4 5 6 None
```

```
# Write a function to remove nth node from the end in a linked list
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setNext(self, next_node):
        self.next = next_node

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.getNext():
            last = last.getNext()
        last.setNext(new_node)

    def print_list(self):
        current = self.head
        while current:
            print(current.getData(), end=" ")
            current = current.getNext()
        print("None")

def remove_nth_from_end(head, n):
    dummy = Node(0)
    dummy.setNext(head)
    first = dummy
    second = dummy

    for _ in range(n + 1):
        first = first.getNext()
    while first:
        first = first.getNext()
        second = second.getNext()

    second.setNext(second.getNext().getNext())
    return dummy.getNext()

# Testing the function
ll = LinkedList()
ll.append(1)
ll.append(2)
ll.append(3)
ll.append(4)
ll.append(5)
print("Original list:")
ll.traverse()
n = 2
head = remove_nth_from_end(ll.head, n)
print(f"List after removing {n}th node from the end:")
ll.traverse()
```

```
Original list:
1 2 3 4 5 None
List after removing 2th node from the end:
1 2 3 5 None
```

```
# Remove duplicates from a sorted linked list.
def remove_duplicates(head):
    current = head
    while current and current.getNext():
        if current.getData() == current.getNext().getData():
            current.setNext(current.getNext().getNext())
        else:
            current = current.getNext()
    return head

# Testing the function
ll = LinkedList()
ll.append(1)
ll.append(1)
ll.append(2)
ll.append(3)
ll.append(3)
ll.append(4)
ll.append(5)
ll.append(5)
print("Original list with duplicates:")
ll.traverse()
ll.head = remove_duplicates(ll.head)
print("List after removing duplicates:")
ll.traverse()
```

```
Original list with duplicates:
1 1 2 3 3 4 5 5 None
List after removing duplicates:
1 2 3 4 5 None
```

```
# Find the intersection of the two linked list.
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
def find_intersection(head1, head2):
    if head1 is None or head2 is None:
        return None

    len1 = 0
    len2 = 0
    temp1 = head1
    temp2 = head2
    while temp1:
        len1 += 1
        temp1 = temp1.next
    while temp2:
        len2 += 1
        temp2 = temp2.next
    diff = abs(len1 - len2)
    if len1 > len2:
        for _ in range(diff):
            head1 = head1.next
    else:
        for _ in range(diff):
            head2 = head2.next
    while head1 and head2:
        if head1.data == head2.data:
            return head1.data
        head1 = head1.next
        head2 = head2.next
    return None
head1 = Node(1)
head1.next = Node(2)
head1.next.next = Node(3)
head1.next.next.next = Node(4)
head1.next.next.next.next = Node(5)
head2 = Node(6)
head2.next = Node(7)
head2.next.next = Node(3)
head2.next.next.next = Node(4)
intersection_node = find_intersection(head1, head2)
if intersection_node:
    print("Intersection node:", intersection_node.data)
else:
    print("No intersection found")
```

```
Intersection node: 3
```

```
# Rotate a linked list by k positions to the right
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
    def rotate_right(self, k):
        if not self.head or not self.head.next:
            return
        length = 1
        tail = self.head
        while tail.next:
            tail = tail.next
            length += 1
        k = k % length
        if k == 0:
            return
        new_tail = length - k - 1
        new_head = self.head
        for _ in range(new_tail):
            new_tail = new_tail.next
        tail.next = self.head
        self.head = new_head
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = None
    def traverse(self):
        current = self.head
        while current:
            print(current.data, end=" ")
            current = current.next
        print("None")
ll = LinkedList()
ll.append(1)
ll.append(2)
ll.append(3)
ll.append(4)
ll.append(5)
print("Original linked list:")
ll.traverse()
ll.rotate_right(2)
print("Linked list after rotating (k) times to the right:")
ll.traverse()
```

```
Original linked list:
1 2 3 4 5 None
Linked list after rotating (k) times to the right:
4 5 1 2 3 None
```

```
# Add two numbers represented by linked lists.Add two numbers (stored like linked lists representing two non-negative integers, where the digits are stored in reverse order) and return the sum 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()
    current = dummy
    carry = 0
    while l1 or l2 or carry:
        val1 = l1.val if l1 else 0
        val2 = l2.val if l2 else 0
        total = val1 + val2 + carry
        carry = total // 10
        current.next = ListNode(total % 10)
        current = current.next
        l1 = l1.next if l1 else None
        l2 = l2.next if l2 else None
    return dummy.next
def printLinkedList(ll):
    current = ll
    while current:
        print(current.val, end=" -> ")
        current = current.next
    print("None")
l1 = ListNode(2)
l1.next = ListNode(4)
l1.next.next = ListNode(3)
l2 = ListNode(5)
l2.next = ListNode(6)
l2.next.next = ListNode(4)
result = addTwoNumbers(l1, l2)
printLinkedList(result)
```

```
7 -> 0 -> 8 -> None
```

```
# Create a linked list with head and Random Pointer.
import random
class Node:
    def __init__(self, val, next=None, random=None):
        self.val = val
        self.next = next
        self.random = random
def clone(llist):
    if not llist:
        return None
    current = head
    while current:
        new_node = Node(current.val)
        current.next = new_node
        current = new_node.next
    current = head
    while current:
        if current.next:
            current.next.next = node_map[current.next]
        if current.random:
            node_map[current].random = node_map[current.random]
        current = current.next
    return node_map[head]
def printLinkedList(llist):
    current = llist
    while current is not None:
        print(f"Node {current.val}, Random: {current.random.val if current.random else None}")
        current = current.next
head = Node(1)
head.next = Node(2)
head.next.next = Node(3)
head.next.random = head
head.next.next.random = head.next
cloned_head = clone(head)
print("Original Linked List:")
printLinkedList(head)
print("\nCloned Linked List:")
printLinkedList(cloned_head)
```

```
Original Linked List:
Node: 1, Random: 3
Node: 2, Random: 1
Node: 3, Random: 2

Cloned Linked List:
Node: 1, Random: 3
Node: 2, Random: 1
Node: 3, Random: 2
```

```
Traceback (most recent call last):
  File "main.py", line 1
    if current.random:
IndentationError: expected an indented block after 'if' statement on line 1
```

```
SyntaxError: unexpected character after line continuation character
```

```
In [ ]:
```