Assignment 12  
  
💡 \*\*Answer 1\*\*

Here's the approach we can follow:

class Node:

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

self.data = data

self.next = None

def deleteMiddle(head):

if head is None or head.next is None:

return None

slowPtr = head

fastPtr = head

prevPtr = None

while fastPtr is not None and fastPtr.next is not None:

fastPtr = fastPtr.next.next

prevPtr = slowPtr

slowPtr = slowPtr.next

prevPtr.next = slowPtr.next

return head

# Example usage

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)

newHead = deleteMiddle(head)

# Print the modified linked list

temp = newHead

while temp:

print(temp.data, end=" ")

temp = temp.next

The output for the given example would be:

```

1 2 4 5

```

💡 \*\*Answer 2\*\*

Here's the approach we can follow:

class Node:

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

self.data = data

self.next = None

def hasLoop(head):

slowPtr = head

fastPtr = head

while fastPtr is not None and fastPtr.next is not None:

slowPtr = slowPtr.next

fastPtr = fastPtr.next.next

if slowPtr == fastPtr:

return True

return False

# Example usage

head = Node(1)

head.next = Node(2)

head.next.next = Node(3)

head.next.next.next = Node(4)

head.next.next.next.next = head.next # Creating a loop

print(hasLoop(head))

The output for the given example would be:

```

True

```

💡 \*\*Answer 3\*\*

Here's the approach we can follow:

class Node:

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

self.data = data

self.next = None

def getNthFromLast(head, N):

mainPtr = head

refPtr = head

# Move the refPtr N nodes ahead

for \_ in range(N):

if refPtr is None:

return -1

refPtr = refPtr.next

# Move both pointers until refPtr reaches the end

while refPtr is not None:

mainPtr = mainPtr.next

refPtr = refPtr.next

return mainPtr.data

# Example usage

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(getNthFromLast(head, 2))

The output for the given example would be:

```

4

```

💡 \*\*Answer 4\*\*

Here's the approach we can follow:

class Node:

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

self.data = data

self.next = None

def reverseList(head):

prevPtr = None

currPtr = head

while currPtr is not None:

nextPtr = currPtr.next

currPtr.next = prevPtr

prevPtr = currPtr

currPtr = nextPtr

return prevPtr

def isPalindrome(head):

slowPtr = head

fastPtr = head

# Find the middle node

while fastPtr is not None and fastPtr.next is not None:

slowPtr = slowPtr.next

fastPtr = fastPtr.next.next

# Reverse the second half of the list

secondHalf = reverseList(slowPtr.next)

# Compare the first half with the reversed second half

firstHalf = head

while secondHalf is not None:

if firstHalf.data != secondHalf.data:

return False

firstHalf = firstHalf.next

secondHalf = secondHalf.next

return True

# Example usage

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(isPalindrome(head))

The output for the given example would be:

```

True

```

💡 \*\*Answer 5\*\*

Here's the approach we can follow:

class Node:

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

self.data = data

self.next = None

def detectAndRemoveLoop(head):

# Step 1: Detect the loop using Floyd's Cycle Detection algorithm

slowPtr = head

fastPtr = head

while fastPtr is not None and fastPtr.next is not None:

slowPtr = slowPtr.next

fastPtr = fastPtr.next.next

if slowPtr == fastPtr:

break

# If there is no loop, return the linked list as is

if slowPtr != fastPtr:

return head

# Step 2: Move ptr1 to the head and keep ptr2 at the meeting point

ptr1 = head

while ptr1.next != fastPtr.next:

ptr1 = ptr1.next

fastPtr = fastPtr.next

# Step 3: Break the loop by setting the next pointer of the last node to None

fastPtr.next = None

return head

# Example usage

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 # Creating a loop

newHead = detectAndRemoveLoop(head)

# Print the modified linked list

temp = newHead

while temp:

print(temp.data, end=" ")

temp = temp.next

The output for the given example would be:

```

1 2 3 4 5

```

💡 \*\*Answer 6\*\*

Here's the approach we can follow:

class Node:

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

self.data = data

self.next = None

def deleteMNodesAfterN(head, M, N):

currPtr = head

prevPtr = None

while currPtr is not None:

# Traverse M nodes

for \_ in range(M):

if currPtr is None:

return head

currPtr = currPtr.next

# Traverse N nodes and delete them

for \_ in range(N):

if currPtr is None:

break

nextPtr = currPtr.next

currPtr = nextPtr

# Update the links after deleting N nodes

if prevPtr is not None:

prevPtr.next = currPtr

return head

# Example usage

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)

head.next.next.next.next.next.next.next = Node(8)

head.next.next.next.next.next.next.next.next = Node(9)

head.next.next.next.next.next.next.next.next.next = Node(10)

newHead = deleteMNodesAfterN(head, 3, 2)

# Print the modified linked list

temp = newHead

while temp:

print(temp.data, end=" ")

temp = temp.next

The output for the given example would be:

```

1 2 3 6 7 8

```

💡 \*\*Answer 7\*\*

Here's the approach we can follow:

class Node:

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

self.data = data

self.next = None

def insertAtAlternatePositions(firstHead, secondHead):

if firstHead is None:

return secondHead

if secondHead is None:

return firstHead

firstPtr = firstHead

secondPtr = secondHead

while firstPtr is not None and secondPtr is not None:

nextFirstPtr = firstPtr.next

nextSecondPtr = secondPtr.next

firstPtr.next = secondPtr

secondPtr.next = nextFirstPtr

firstPtr = nextFirstPtr

secondPtr = nextSecondPtr

if secondPtr is not None:

firstPtr.next = secondPtr

secondHead = None

return firstHead

# Example usage

firstHead = Node(5)

firstHead.next = Node(7)

firstHead.next.next = Node(17)

firstHead.next.next.next = Node(13)

firstHead.next.next.next.next = Node(11)

secondHead = Node(12)

secondHead.next = Node(10)

secondHead.next.next = Node(2)

secondHead.next.next.next = Node(4)

secondHead.next.next.next.next = Node(6)

newFirstHead = insertAtAlternatePositions(firstHead, secondHead)

# Print the modified first list

temp = newFirstHead

while temp:

print(temp.data, end=" ")

temp = temp.next

# Print the modified second list

print()

temp = secondHead

while temp:

print(temp.data, end=" ")

temp = temp.next

The output for the given example would be:

```

5 12 7 10 17 2 13 4 11 6

```

💡 \*\*Answer 8\*\*

Here's the approach we can follow:

class Node:

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

self.data = data

self.next = None

def isCircular(head):

if head is None:

return False

slowPtr = head

fastPtr = head

while fastPtr is not None and fastPtr.next is not None:

slowPtr = slowPtr.next

fastPtr = fastPtr.next.next

if slowPtr == fastPtr:

return True

return False

# Example usage

head = Node(1)

head.next = Node(2)

head.next.next = Node(3)

head.next.next.next = Node(4)

head.next.next.next.next = head # Creating a loop

print(isCircular(head))

The output for the given example would be:

```

True

```