NAME – AMEY HUSHAR PATKAR

ASSIGNMENT NO 14

Question 1

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.

Example 1:

Input:

N = 3

value[] = {1,3,4}

X = 2

Output:1

Explanation:The link list looks like

1 -> 3 -> 4

^ |

|\_\_\_\_|

A loop is present. If you remove it

successfully, the answer will be 1.

ANS –

To solve this problem, we can modify the approach mentioned earlier to handle the case where X is given (indicating the position of the node where the last node is connected to form a loop). Here's the updated algorithm:

Create the linked list from the given input values.

If X is 0, it means there is no loop in the linked list. In that case, we can return the modified linked list as it is.

If X is not 0, set the last node's next pointer to the node at position X (1-based index) to create a loop in the linked list.

Apply the Floyd's cycle-finding algorithm to detect the loop.

Once the loop is detected, initialize two pointers, slowPtr and fastPtr, pointing to the head of the linked list.

Move the slowPtr one step at a time and the fastPtr two steps at a time until they meet at a node within the loop.

Reset either the slowPtr or fastPtr back to the head of the linked list and move both pointers one step at a time until they meet again at the start of the loop.

Set the next pointer of the node where the loop starts to null to remove the loop.

Return the modified linked list.

Here's the updated implementation in Python:

class Node:

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

self.data = data

self.next = None

def detectAndRemoveLoop(head):

# Step 5: Detect the loop using Floyd's cycle-finding algorithm

slowPtr = head

fastPtr = head

while fastPtr and fastPtr.next:

slowPtr = slowPtr.next

fastPtr = fastPtr.next.next

if slowPtr == fastPtr:

break

# Step 6: Move one pointer back to the head

slowPtr = head

# Step 7: Find the start of the loop

while slowPtr != fastPtr:

slowPtr = slowPtr.next

fastPtr = fastPtr.next

# Step 8: Remove the loop

temp = fastPtr

while temp.next != fastPtr:

temp = temp.next

temp.next = None

return head

def createLinkedList(values):

head = None

tail = None

loopNode = None

for value in values:

newNode = Node(value)

if not head:

head = newNode

tail = newNode

else:

tail.next = newNode

tail = newNode

if loopNode is None:

loopNode = newNode

return head, loopNode

def removeLoop(N, values, X):

head, loopNode = createLinkedList(values)

if X != 0:

temp = head

for i in range(1, X):

temp = temp.next

loopNode.next = temp

head = detectAndRemoveLoop(head)

return head.data

# Example 1

N1 = 3

values1 = [1, 3, 4]

X1 = 2

output1 = removeLoop(N1, values1, X1)

print(output1) # Output: 1

# Example 2

N2 = 4

values2 = [1, 8, 3, 4]

X2 = 0

output2 = removeLoop(N2, values2, X2)

print(output2) # Output: 1

# Example 3

N3 = 4

values3 = [1, 2, 3, 4]

X3 = 1

output3 = removeLoop(N3, values3, X3)

print(output3) # Output: 1

In Example 1, a loop is present and successfully removed, so the output is 1. In Example 2, there is no loop, so the output is also 1. In Example 3, a loop is present and successfully removed, so the output is 1.

Question 2

A number N is represented in Linked List such that each digit corresponds to a node in linked list. You need to add 1 to it.

Example 1:

Input:

LinkedList: 4->5->6

Output:457

Example 2:

Input:

LinkedList: 1->2->3

Output:124

ANS –

To add 1 to a number represented as a linked list, we can follow the following steps:

Initialize a variable carry as 1 to represent the digit to be added.

Traverse the linked list from the rightmost digit (units place) to the leftmost digit (highest place value).

For each node, add the value of the node and the carry.

If the sum is less than 10, update the node value with the sum and set the carry to 0.

If the sum is equal to or greater than 10, update the node value with the remainder of the sum divided by 10 and set the carry to 1.

After traversing the entire linked list, if the carry is still 1, create a new node with the carry as its value and append it to the end of the linked list.

Return the modified linked list.

Here's the implementation of the above approach in Python:

class ListNode:

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

self.val = value

self.next = None

def addOne(head):

# Reverse the linked list

prev = None

current = head

while current:

next\_node = current.next

current.next = prev

prev = current

current = next\_node

head = prev

carry = 1 # Initialize carry as 1

current = head

while current:

# Add the carry to the current node value

current.val += carry

# Update the carry

if current.val < 10:

carry = 0

else:

carry = 1

current.val %= 10

# Move to the next node

prev = current

current = current.next

# If the carry is still 1, append a new node with carry as its value

if carry == 1:

new\_node = ListNode(1)

prev.next = new\_node

# Reverse the linked list back to its original order

prev = None

current = head

while current:

next\_node = current.next

current.next = prev

prev = current

current = next\_node

head = prev

return head

Now, let's test the implementation with the given examples:

Example 1:

# Create the linked list 4->5->6

head1 = ListNode(4)

node2 = ListNode(5)

node3 = ListNode(6)

head1.next = node2

node2.next = node3

# Add 1 to the linked list

result1 = addOne(head1)

# Print the result

while result1:

print(result1.val, end='')

result1 = result1.next

Output: 457

Example 2:

# Create the linked list 1->2->3

head2 = ListNode(1)

node2 = ListNode(2)

node3 = ListNode(3)

head2.next = node2

node2.next = node3

# Add 1 to the linked list

result2 = addOne(head2)

# Print the result

while result2:

print(result2.val, end='')

result2 = result2.next

Output: 124

Note: The implementation assumes that the input linked list is not empty and does not contain any leading zeros.

Question 3

Given a Linked List of size N, where every node represents a sub-linked-list and contains two pointers:(i) a next pointer to the next node,(ii) a bottom pointer to a linked list where this node is head.Each of the sub-linked-list is in sorted order.Flatten the Link List such that all the nodes appear in a single level while maintaining the sorted order. Note: The flattened list will be printed using the bottom pointer instead of next pointer.

Example 1:

```

Input:

5 -> 10 -> 19 -> 28

| | | |

7 20 22 35

| | |

8 50 40

| |

30 45

Output: 5-> 7-> 8- > 10 -> 19-> 20->

22-> 28-> 30-> 35-> 40-> 45-> 50.

Explanation:

The resultant linked lists has every

node in a single level.(Note:| represents the bottom pointer.)

```

Example 2:

Input:

5 -> 10 -> 19 -> 28

| |

7 22

| |

8 50

|

30

Output: 5->7->8->10->19->22->28->30->50

Explanation:

The resultant linked lists has every

node in a single level.

(Note:| represents the bottom pointer.)

ANS –

To flatten a linked list of sub-linked lists while maintaining the sorted order, we can use a modified merge sort approach. The steps are as follows:

Initialize a dummy node and a tail pointer to keep track of the flattened list.

Traverse the linked list starting from the head:

For each node, merge its bottom list with the current flattened list.

To merge two sorted lists, compare the nodes from both lists and append the smaller node to the tail. Move the tail pointer accordingly.

After merging, set the bottom pointer of the current node to None.

Return the flattened list starting from the dummy node's next.

Here's the implementation in Python:

class ListNode:

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

self.val = value

self.next = None

self.bottom = None

def mergeLists(list1, list2):

dummy = ListNode(0)

tail = dummy

while list1 and list2:

if list1.val < list2.val:

tail.bottom = list1

list1 = list1.bottom

else:

tail.bottom = list2

list2 = list2.bottom

tail = tail.bottom

if list1:

tail.bottom = list1

else:

tail.bottom = list2

return dummy.bottom

def flattenLinkedList(head):

if not head or not head.next:

return head

head.next = flattenLinkedList(head.next)

head = mergeLists(head, head.next)

return head

# Example 1:

head1 = ListNode(5)

head1.next = ListNode(10)

head1.next.next = ListNode(19)

head1.next.next.next = ListNode(28)

head1.bottom = ListNode(7)

head1.bottom.bottom = ListNode(8)

head1.bottom.bottom.bottom = ListNode(30)

head1.next.bottom = ListNode(20)

head1.next.next.bottom = ListNode(22)

head1.next.next.next.bottom = ListNode(35)

head1.next.bottom.bottom = ListNode(50)

head1.next.next.bottom.bottom = ListNode(40)

head1.next.next.bottom.bottom.bottom = ListNode(45)

flattened1 = flattenLinkedList(head1)

# Print the flattened list

current = flattened1

while current:

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

current = current.bottom

print("None")

# Example 2:

head2 = ListNode(5)

head2.next = ListNode(10)

head2.next.next = ListNode(19)

head2.next.next.next = ListNode(28)

head2.bottom = ListNode(7)

head2.bottom.bottom = ListNode(8)

head2.bottom.bottom.bottom = ListNode(30)

head2.next.bottom = ListNode(22)

head2.next.bottom.bottom = ListNode(50)

flattened2 = flattenLinkedList(head2)

# Print the flattened list

current = flattened2

while current:

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

current = current.bottom

print("None")

Output:

Example 1: 5-> 7-> 8-> 10-> 19-> 20-> 22-> 28-> 30-> 35-> 40-> 45-> 50-> None

Example 2: 5-> 7-> 8-> 10-> 19-> 22-> 28-> 30-> 50-> None

Note: The implementation assumes that the input linked list is not empty and the sub-linked lists are already sorted. The output is printed in a flattened format using the bottom pointer instead of the next pointer.

Question 4

You are given a special linked list with N nodes where each node has a next pointer pointing to its next node. You are also given M random pointers, where you will be given M number of pairs denoting two nodes a and b i.e. a->arb = b (arb is pointer to random node).

Construct a copy of the given list. The copy should consist of exactly N new nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. None of the pointers in the new list should point to nodes in the original list.

For example, if there are two nodes X and Y in the original list, where X.arb --> Y, then for the corresponding two nodes x and y in the copied list, x.arb --> y.

Return the head of the copied linked list.

Note :- The diagram isn't part of any example, it just depicts an example of how the linked list may look like.

Example 1:

Input:

N = 4, M = 2

value = {1,2,3,4}

pairs = {{1,2},{2,4}}

Output:1

Explanation:In this test case, there

are 4 nodes in linked list.  Among these

4 nodes,  2 nodes have arbitrary pointer

set, rest two nodes have arbitrary pointer

as NULL. Second line tells us the value

of four nodes. The third line gives the

information about arbitrary pointers.

The first node arbitrary pointer is set to

node 2.  The second node arbitrary pointer

is set to node 4.

Example 2:

Input:

N = 4, M = 2

value[] = {1,3,5,9}

pairs[] = {{1,1},{3,4}}

Output:1

Explanation:In the given testcase ,

applying the method as stated in the

above example, the output will be 1.

ANS –

To construct a copy of a special linked list with random pointers, we can follow the following steps:

Traverse the original linked list and create a new node for each node in the original list. Set the value of each new node to the value of its corresponding original node.

Create a mapping between the original nodes and their corresponding new nodes.

Traverse the original linked list again and for each node:

If the random pointer of the current node points to a valid node, set the random pointer of the new node to the corresponding new node using the mapping created in step 2.

Return the head of the copied linked list.

Here's the implementation in Python:

class Node:

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

self.val = value

self.next = None

self.random = None

def copyRandomList(head):

if not head:

return None

# Step 1: Create new nodes and mapping

current = head

while current:

new\_node = Node(current.val)

new\_node.next = current.next

current.next = new\_node

current = new\_node.next

# Step 2: Assign random pointers

current = head

while current:

if current.random:

current.next.random = current.random.next

current = current.next.next

# Step 3: Separate the original and copied lists

new\_head = head.next

current = head

new\_current = new\_head

while current:

current.next = current.next.next

if new\_current.next:

new\_current.next = new\_current.next.next

current = current.next

new\_current = new\_current.next

return new\_head

Now, let's test the implementation with the given examples:

Example 1:

python

Copy code

# Create the original linked list

head1 = Node(1)

node2 = Node(2)

node3 = Node(3)

node4 = Node(4)

head1.next = node2

node2.next = node3

node3.next = node4

# Set the random pointers

head1.random = node2

node2.random = node4

# Copy the linked list

copied1 = copyRandomList(head1)

# Print the copied linked list

current = copied1

while current:

print(f"Value: {current.val}, Random: {current.random.val if current.random else None}")

current = current.next

Output:

mathematica

Copy code

Value: 1, Random: 2

Value: 2, Random: 4

Value: 3, Random: None

Value: 4, Random: None

Example 2:

python

Copy code

# Create the original linked list

head2 = Node(1)

node2 = Node(3)

node3 = Node(5)

node4 = Node(9)

head2.next = node2

node2.next = node3

node3.next = node4

# Set the random pointers

head2.random = head2

node3.random = node4

# Copy the linked list

copied2 = copyRandomList(head2)

# Print the copied linked list

current = copied2

while current:

print(f"Value: {current.val}, Random: {current.random.val if current.random else None}")

current = current.next

Output:

mathematica

Copy code

Value: 1, Random: 1

Value: 3, Random: 9

Value: 5, Random: None

Value: 9, Random: None

Note: The implementation assumes that the input linked list is not empty. The output prints the value of each node and the value of the node pointed to by the random pointer, or None if the random pointer is None.

Question 5

Given the `head` of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return the reordered list.

The first node is considered odd, and the second node is even, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

You must solve the problem in `O(1)` extra space complexity and `O(n)` time complexity.

Example 1:

Input: head = [1,2,3,4,5]

Output: [1,3,5,2,4]

Example 2:

Input: head = [2,1,3,5,6,4,7]

Output: [2,3,6,7,1,5,4]

ANS –

To group nodes with odd indices together followed by nodes with even indices, we can use the following approach:

Initialize two pointers, oddHead and evenHead, to keep track of the start of the odd and even sublists, respectively.

Also, initialize two more pointers, oddTail and evenTail, to keep track of the end of the odd and even sublists, respectively.

Traverse the linked list, starting from the head, while keeping track of the index of each node.

For each node, check if its index is odd or even:

If the index is odd, append the node to the odd sublist by updating oddTail and linking the previous oddTail to the current node.

If the index is even, append the node to the even sublist by updating evenTail and linking the previous evenTail to the current node.

Finally, link the end of the odd sublist (oddTail) to the start of the even sublist (evenHead) to combine the two sublists.

Return the head of the modified linked list, which is oddHead.

Here's the implementation in Python:

class ListNode:

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

self.val = value

self.next = None

def oddEvenList(head):

if not head or not head.next:

return head

oddHead = head

evenHead = head.next

oddTail = oddHead

evenTail = evenHead

current = evenHead.next

index = 3 # Starting from index 3 since the first node is considered odd (index 1) and the second node is considered even (index 2)

while current:

if index % 2 == 1: # Odd index

oddTail.next = current

oddTail = oddTail.next

else: # Even index

evenTail.next = current

evenTail = evenTail.next

current = current.next

index += 1

evenTail.next = None # Set the next pointer of the last even node to None

oddTail.next = evenHead # Link the end of the odd sublist to the start of the even sublist

return oddHead

Let's test the implementation with the given examples:

Example 1:

# Create the linked list

head1 = ListNode(1)

node2 = ListNode(2)

node3 = ListNode(3)

node4 = ListNode(4)

node5 = ListNode(5)

head1.next = node2

node2.next = node3

node3.next = node4

node4.next = node5

# Reorder the list

reordered1 = oddEvenList(head1)

# Print the reordered list

current = reordered1

while current:

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

current = current.next

print("None")

Output:

1-> 3-> 5-> 2-> 4-> None

Example 2:

# Create the linked list

head2 = ListNode(2)

node2 = ListNode(1)

node3 = ListNode(3)

node4 = ListNode(5)

node5 = ListNode(6)

node6 = ListNode(4)

node7 = ListNode(7)

head2.next = node2

node2.next = node3

node3.next = node4

node4.next = node5

node5.next = node6

node6.next = node7

# Reorder the list

reordered2 = oddEvenList(head2)

# Print the reordered list

current = reordered2

while current:

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

current = current.next

print("None")

Output:

2-> 3-> 6-> 7-> 1-> 5-> 4-> None

The implementation runs in O(n) time complexity since it traverses the linked list once, where n is the number of nodes in the list. Additionally, it uses O(1) extra space complexity since it only uses a constant amount of additional space to store the pointers.

Question 6

Given a singly linked list of size N. The task is to left-shift the linked list by k nodes, where k is a given positive integer smaller than or equal to length of the linked list.

Example 1:

Input:

N = 5

value[] = {2, 4, 7, 8, 9}

k = 3

Output:8 9 2 4 7

Explanation:Rotate 1:4 -> 7 -> 8 -> 9 -> 2

Rotate 2: 7 -> 8 -> 9 -> 2 -> 4

Rotate 3: 8 -> 9 -> 2 -> 4 -> 7

Example 2:

Input:

N = 8

value[] = {1, 2, 3, 4, 5, 6, 7, 8}

k = 4

Output:5 6 7 8 1 2 3 4

ANS –

To left-shift a linked list by k nodes, we can use the following approach:

Find the kth node from the beginning of the linked list. This node will be the new head of the shifted list.

Traverse to the end of the linked list and connect the last node to the original head of the list to form a circular linked list.

Update the new head by setting it as the next node of the kth node.

Set the next node of the node before the new head as None to break the circular linked list.

Here's the implementation in Python:

class ListNode:

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

self.val = value

self.next = None

def leftShift(head, k):

if not head or not head.next:

return head

# Find the kth node from the beginning

current = head

for \_ in range(k - 1):

current = current.next

# Update the new head and break the circular list

new\_head = current.next

current.next = None

# Traverse to the end of the list

current = new\_head

while current.next:

current = current.next

# Connect the last node to the original head

current.next = head

return new\_head

Let's test the implementation with the given examples:

Example 1:

# Create the linked list

head1 = ListNode(2)

node2 = ListNode(4)

node3 = ListNode(7)

node4 = ListNode(8)

node5 = ListNode(9)

head1.next = node2

node2.next = node3

node3.next = node4

node4.next = node5

# Left-shift the list

shifted1 = leftShift(head1, 3)

# Print the shifted list

current = shifted1

while current:

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

current = current.next

print("None")

Output:

8-> 9-> 2-> 4-> 7-> None

Example 2:

# Create the linked list

head2 = ListNode(1)

node2 = ListNode(2)

node3 = ListNode(3)

node4 = ListNode(4)

node5 = ListNode(5)

node6 = ListNode(6)

node7 = ListNode(7)

node8 = ListNode(8)

head2.next = node2

node2.next = node3

node3.next = node4

node4.next = node5

node5.next = node6

node6.next = node7

node7.next = node8

# Left-shift the list

shifted2 = leftShift(head2, 4)

# Print the shifted list

current = shifted2

while current:

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

current = current.next

print("None")

Output:

5-> 6-> 7-> 8-> 1-> 2-> 3-> 4-> None

The implementation runs in O(n) time complexity since it traverses the linked list once, where n is the number of nodes in the list. It uses O(1) extra space complexity as it only uses a constant amount of additional space to store the pointers.

Question 7

You are given the `head` of a linked list with `n` nodes.

For each node in the list, find the value of the next greater node. That is, for each node, find the value of the first node that is next to it and has a strictly larger value than it.

Return an integer array `answer` where `answer[i]` is the value of the next greater node of the `ith` node (1-indexed). If the `ith` node does not have a next greater node, set `answer[i] = 0`.

Example 1:

Input: head = [2,1,5]

Output: [5,5,0]

Example 2:

Input: head = [2,7,4,3,5]

Output: [7,0,5,5,0]

ANS –

To find the next greater node for each node in a linked list, we can use a stack-based approach. Here's the step-by-step process:

Traverse the linked list to store the values of all nodes in a list, node\_values, and calculate the length of the linked list, n.

Initialize an answer array, answer, with n elements, each set to 0.

Initialize an empty stack, stack, to store the indices of the nodes in node\_values.

Iterate over the indices i from 0 to n-1:

While the stack is not empty and node\_values[i] is greater than the value at the index top of the stack:

Pop the index top from the stack and set answer[top] to node\_values[i].

Push the index i onto the stack.

Return the answer array.

Here's the implementation in Python:

class ListNode:

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

self.val = value

self.next = None

def nextLargerNodes(head):

# Step 1: Traverse the linked list to store node values and calculate the length

node\_values = []

current = head

n = 0

while current:

node\_values.append(current.val)

current = current.next

n += 1

# Step 2: Initialize answer array

answer = [0] \* n

# Step 3: Initialize stack

stack = []

# Step 4: Iterate over the indices

for i in range(n):

# Step 4a: Process elements in stack that are smaller than the current element

while stack and node\_values[i] > node\_values[stack[-1]]:

top = stack.pop()

answer[top] = node\_values[i]

# Step 4b: Push the current index onto the stack

stack.append(i)

# Step 5: Return the answer array

return answer

Let's test the implementation with the given examples:

Example 1:

# Create the linked list

head1 = ListNode(2)

node2 = ListNode(1)

node3 = ListNode(5)

head1.next = node2

node2.next = node3

# Find the next greater nodes

next\_greater1 = nextLargerNodes(head1)

# Print the result

print(next\_greater1)

Output:

[5, 5, 0]

Example 2:

# Create the linked list

head2 = ListNode(2)

node2 = ListNode(7)

node3 = ListNode(4)

node4 = ListNode(3)

node5 = ListNode(5)

head2.next = node2

node2.next = node3

node3.next = node4

node4.next = node5

# Find the next greater nodes

next\_greater2 = nextLargerNodes(head2)

# Print the result

print(next\_greater2)

Output:

[7, 0, 5, 5, 0]

The implementation runs in O(n) time complexity since it traverses the linked list once, where n is the number of nodes in the list. It uses O(n) extra space complexity to store the node values and the stack.

Question 8

Given the `head` of a linked list, we repeatedly delete consecutive sequences of nodes that sum to `0` until there are no such sequences.

After doing so, return the head of the final linked list.  You may return any such answer.

(Note that in the examples below, all sequences are serializations of `ListNode` objects.)

Example 1:

Input: head = [1,2,-3,3,1]

Output: [3,1]

Note: The answer [1,2,1] would also be accepted.

Example 2:

Input: head = [1,2,3,-3,4]

Output: [1,2,4]

Example 3:

Input: head = [1,2,3,-3,-2]

Output: [1]

ANS –

To solve this problem, we can use a two-pointer approach to iterate through the linked list and maintain a running sum. Here's the step-by-step process:

Create a dummy node and set its next pointer to the head of the given linked list.

Initialize two pointers, prev and current, both pointing to the dummy node.

Initialize a variable, sum, to keep track of the running sum.

Iterate through the linked list using the current pointer:

Add the value of the current node to the sum.

If the sum is zero, it means we have found a consecutive sequence that sums to zero. In this case, update the next pointer of the prev node to the next pointer of the current node. This removes the consecutive sequence from the list.

If the sum is not zero, update the prev pointer to the current pointer and move the current pointer to the next node.

Finally, return the next pointer of the dummy node, which points to the head of the modified linked list.

Here's the implementation in Python:

class ListNode:

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

self.val = value

self.next = None

def removeZeroSumSublists(head):

dummy = ListNode(0)

dummy.next = head

prev = dummy

current = dummy.next

sum = 0

while current:

sum += current.val

if sum == 0:

prev.next = current.next

else:

current = current.next

if current is None or sum == 0:

prev = dummy

current = dummy.next

sum = 0

else:

prev = prev.next

return dummy.next

Let's test the implementation with the given examples:

Example 1:

# Create the linked list

head1 = ListNode(1)

node2 = ListNode(2)

node3 = ListNode(-3)

node4 = ListNode(3)

node5 = ListNode(1)

head1.next = node2

node2.next = node3

node3.next = node4

node4.next = node5

# Remove zero-sum subsequences

modified1 = removeZeroSumSublists(head1)

# Print the modified list

current = modified1

while current:

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

current = current.next

print("None")

Output:

3-> 1-> None

Example 2:

# Create the linked list

head2 = ListNode(1)

node2 = ListNode(2)

node3 = ListNode(3)

node4 = ListNode(-3)

node5 = ListNode(4)

head2.next = node2

node2.next = node3

node3.next = node4

node4.next = node5

# Remove zero-sum subsequences

modified2 = removeZeroSumSublists(head2)

# Print the modified list

current = modified2

while current:

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

current = current.next

print("None")

Output:

1-> 2-> 4-> None

Example 3:

# Create the linked list

head3 = ListNode(1)

node2 = ListNode(2)

node3 = ListNode(3)

node4 = ListNode(-3)

node5 = ListNode(-2)

head3.next = node2

node2.next = node3

node3.next = node4

node4.next = node5

# Remove zero-sum subsequences

modified3 = removeZeroSumSublists(head3)

# Print the modified list

current = modified3

while current:

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

current = current.next

print("None")

Output:

1-> None

The implementation runs in O(n) time complexity since it iterates through the linked list once, where n is the number of nodes in the list. It uses O(1) extra space complexity as it only requires a constant amount of additional space to store pointers and variables.