**Assignment 14 - Linked List**

**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.

**Example 2:**

Input:

N = 4

value[] = {1,8,3,4}

X = 0

Output:1

Explanation:The Linked list does not

contains any loop.

**Example 3**

Input:

N = 4

value[] = {1,2,3,4}

X = 1

Output:1

Explanation:The link list looks like

1 -> 2 -> 3 -> 4

^ |

|\_\_\_\_\_\_\_\_\_\_\_\_\_\_|

A loop is present.

If you remove it successfully,

the answer will be 1.

**Sol.**

def removeLoop(head):

# Step 1: Detect the loop

slow = fast = head

loop\_exists = False

while fast and fast.next:

slow = slow.next

fast = fast.next.next

if slow == fast:

loop\_exists = True

break

if not loop\_exists:

return head

# Step 2: Find the start of the loop

slow = head

while slow.next != fast.next:

slow = slow.next

fast = fast.next

# Step 3: Remove the loop

fast.next = None

return head

# Creating the linked list with a loop

head = Node(1)

node2 = Node(3)

node3 = Node(4)

head.next = node2

node2.next = node3

node3.next = node2

# Calling the function to remove the loop

head = removeLoop(head)

# Creating the linked list without a loop

head = Node(1)

node2 = Node(8)

node3 = Node(3)

node4 = Node(4)

head.next = node2

node2.next = node3

node3.next = node4

# Calling the function to remove the loop

head = removeLoop(head)

# Creating the linked list with a loop

head = Node(1)

node2 = Node(2)

node3 = Node(3)

node4 = Node(4)

head.next = node2

node2.next = node3

node3.next = node4

node4.next = node2

# Calling the function to remove the loop

head = removeLoop(head)

**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

**Sol:**

class Node:

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

self.data = data

self.next = None

def addOne(head):

# Base case: If the linked list is empty, return 1 as a new node

if not head:

return Node(1)

# Recursive call to the next node

carry = addOne(head.next)

# Add the carry to the current node

sum = head.data + carry.data

# Update the value of the current node

head.data = sum % 10

# Return the carry to be propagated to the previous node

return Node(sum // 10)

def convertToList(head):

# Convert the linked list to an integer

num = 0

while head:

num = num \* 10 + head.data

head = head.next

return num

def createLinkedList(num):

# Create a linked list from an integer

head = None

prev = None

for digit in str(num):

node = Node(int(digit))

if not head:

head = node

if prev:

prev.next = node

prev = node

return head

def addOneToLinkedList(head):

carry = addOne(head)

# If there is a carry, create a new node at the beginning

if carry.data > 0:

new\_head = Node(carry.data)

new\_head.next = head

return new\_head

return head

# Example 1:

num1 = 456

head1 = createLinkedList(num1)

result1 = addOneToLinkedList(head1)

print(convertToList(result1)) # Output: 457

# Example 2:

num2 = 123

head2 = createLinkedList(num2)

result2 = addOneToLinkedList(head2)

print(convertToList(result2)) # Output: 124

**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.)

**Sol:**

class Node:

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

self.data = data

self.next = None

self.bottom = None

def mergeLists(head1, head2):

# Base cases: If either list is empty, return the other list

if not head1:

return head2

if not head2:

return head1

# Merge the two lists in sorted order

merged = None

if head1.data < head2.data:

merged = head1

merged.bottom = mergeLists(head1.bottom, head2)

else:

merged = head2

merged.bottom = mergeLists(head1, head2.bottom)

return merged

def flattenLinkedList(head):

# Base case: If the list is empty or has only one node, return the list

if not head or not head.next:

return head

# Recursively flatten the next node

head.next = flattenLinkedList(head.next)

# Merge the current sub-linked list with the flattened sub-linked list of the next node

head = mergeLists(head, head.next)

return head

def printList(head):

# Print the flattened linked list using the bottom pointer

current = head

while current:

print(current.data, end=" ")

current = current.bottom

print()

# Example 1:

head1 = Node(5)

head1.next = Node(10)

head1.next.next = Node(19)

head1.next.next.next = Node(28)

head1.bottom = Node(7)

head1.bottom.bottom = Node(8)

head1.bottom.bottom.bottom = Node(30)

head1.next.bottom = Node(20)

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

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

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

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

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

head1 = flattenLinkedList(head1)

printList(head1) # Output: 5 7 8 10 19 20 22 28 30 35 40 45 50

# Example 2:

head2 = Node(5)

head2.next = Node(10)

head2.next.next = Node(19)

head2.next.next.next = Node(28)

head2.bottom = Node(7)

head2.bottom.bottom = Node(8)

head2.bottom.bottom.bottom = Node(30)

head2.next.bottom = Node(22)

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

head2 = flattenLinkedList(head2)

printList(head2) # Output: 5 7 8 10 19 22 28 30 50

**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.

**Sol:**

class Node:

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

self.data = data

self.next = None

self.random = None

def copyRandomList(head):

if not head:

return None

# Step 1: Create a copy of each node and map original nodes to their copies

mappings = {}

current = head

while current:

mappings[current] = Node(current.data)

current = current.next

# Step 2: Update the next and random pointers of the copied nodes

current = head

while current:

copy\_node = mappings[current]

copy\_node.next = mappings.get(current.next)

copy\_node.random = mappings.get(current.random)

current = current.next

return mappings[head]

def printList(head):

# Print the linked list

current = head

while current:

random\_data = current.random.data if current.random else "None"

print(f"Value: {current.data}, Random: {random\_data}")

current = current.next

# Example 1:

head1 = Node(1)

node2 = Node(2)

node3 = Node(3)

node4 = Node(4)

head1.next = node2

head1.random = node2

node2.next = node3

node2.random = node4

node3.next = node4

node3.random = None

node4.random = node4

copied\_head1 = copyRandomList(head1)

printList(copied\_head1)

"""

Output:

Value: 1, Random: 2

Value: 2, Random: 4

Value: 3, Random: None

Value: 4, Random: 4

"""

# Example 2:

head2 = Node(1)

node3 = Node(3)

node5 = Node(5)

node9 = Node(9)

head2.next = node3

head2.random = head2

node3.next = node5

node3.random = node9

node5.next = node9

node5.random = None

node9.random = None

copied\_head2 = copyRandomList(head2)

printList(copied\_head2)

"""

Output:

Value: 1, Random: 1

Value: 3, Random: 9

Value: 5, Random: None

Value: 9, Random: 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]

**Sol:**

class ListNode:

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

self.val = val

self.next = next

def oddEvenList(head):

if not head or not head.next:

return head

oddHead = oddCurrent = head

evenHead = evenCurrent = head.next

current = head.next.next

isOdd = True

while current:

if isOdd:

oddCurrent.next = current

oddCurrent = oddCurrent.next

else:

evenCurrent.next = current

evenCurrent = evenCurrent.next

current = current.next

isOdd = not isOdd

oddCurrent.next = evenHead

evenCurrent.next = None

return oddHead

def printList(head):

# Print the linked list

current = head

while current:

print(current.val, end=" ")

current = current.next

print()

# Example 1:

head1 = ListNode(1)

head1.next = ListNode(2)

head1.next.next = ListNode(3)

head1.next.next.next = ListNode(4)

head1.next.next.next.next = ListNode(5)

reordered1 = oddEvenList(head1)

printList(reordered1) # Output: 1 3 5 2 4

# Example 2:

head2 = ListNode(2)

head2.next = ListNode(1)

head2.next.next = ListNode(3)

head2.next.next.next = ListNode(5)

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

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

head2.next.next.next.next.next.next = ListNode(7)

reordered2 = oddEvenList(head2)

printList(reordered2) # Output: 2 3 6 7 1 5 4

**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

**Sol:**

class ListNode:

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

self.val = val

self.next = next

def leftShift(head, k):

if not head or not head.next:

return head

# Find the (k+1)-th node from the start of the linked list

current = head

for \_ in range(k):

current = current.next

# Update the head of the linked list and disconnect the node before it

new\_head = current.next

current.next = None

# Traverse to the end of the linked list and connect the last node to the original head

current = new\_head

while current.next:

current = current.next

current.next = head

return new\_head

def printList(head):

# Print the linked list

current = head

while current:

print(current.val, end=" ")

current = current.next

print()

# Example 1:

head1 = ListNode(2)

head1.next = ListNode(4)

head1.next.next = ListNode(7)

head1.next.next.next = ListNode(8)

head1.next.next.next.next = ListNode(9)

shifted1 = leftShift(head1, 3)

printList(shifted1) # Output: 8 9 2 4 7

# Example 2:

head2 = ListNode(1)

head2.next = ListNode(2)

head2.next.next = ListNode(3)

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

head2.next.next.next.next = ListNode(5)

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

head2.next.next.next.next.next.next = ListNode(7)

head2.next.next.next.next.next.next.next = ListNode(8)

shifted2 = leftShift(head2, 4)

printList(shifted2) # Output: 5 6 7 8 1 2 3 4

**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]

**Sol:**

class ListNode:

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

self.val = val

self.next = next

def nextLargerNodes(head):

# Step 1: Convert the linked list to a list

values = []

current = head

while current:

values.append(current.val)

current = current.next

# Step 2: Traverse the list in reverse order and find the next greater node for each node

stack = []

result = [0] \* len(values)

for i in range(len(values) - 1, -1, -1):

while stack and values[i] >= stack[-1][0]:

stack.pop()

if stack:

result[i] = stack[-1][0]

stack.append((values[i], i))

return result

# Example 1:

head1 = ListNode(2)

head1.next = ListNode(1)

head1.next.next = ListNode(5)

result1 = nextLargerNodes(head1)

print(result1) # Output: [5, 5, 0]

# Example 2:

head2 = ListNode(2)

head2.next = ListNode(7)

head2.next.next = ListNode(4)

head2.next.next.next = ListNode(3)

head2.next.next.next.next = ListNode(5)

result2 = nextLargerNodes(head2)

print(result2) # Output: [7, 0, 5, 5, 0]

**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]

**Sol:**

class ListNode:

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

self.val = val

self.next = next

def removeZeroSumSublists(head):

# Step 1: Create a dummy node and connect it to the head of the linked list

dummy = ListNode(0)

dummy.next = head

# Step 2: Traverse the linked list to compute the running sum

prefix\_sum = 0

prefix\_sums = {}

current = dummy

while current:

prefix\_sum += current.val

if prefix\_sum in prefix\_sums:

# Remove the subsequence with a sum of 0

prev = prefix\_sums[prefix\_sum]

prev.next = current.next

prefix\_sum -= current.val

current = prev

else:

prefix\_sums[prefix\_sum] = current

current = current.next

return dummy.next

def printList(head):

# Print the linked list

current = head

while current:

print(current.val, end=" ")

current = current.next

print()

# Example 1:

head1 = ListNode(1)

head1.next = ListNode(2)

head1.next.next = ListNode(-3)

head1.next.next.next = ListNode(3)

head1.next.next.next.next = ListNode(1)

result1 = removeZeroSumSublists(head1)

printList(result1) # Output: 3 1

# Example 2:

head2 = ListNode(1)

head2.next = ListNode(2)

head2.next.next = ListNode(3)

head2.next.next.next = ListNode(-3)

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

result2 = removeZeroSumSublists(head2)

printList(result2) # Output: 1 2 4

# Example 3:

head3 = ListNode(1)

head3.next = ListNode(2)

head3.next.next = ListNode(3)

head3.next.next.next = ListNode(-3)

head3.next.next.next.next = ListNode(-2)

result3 = removeZeroSumSublists(head3)

printList(result3) # Output: 1