**Assignment Questions 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.

**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.

**CODE:**

**class** Node:

**def** \_\_init\_\_(self, data**=None**):

self**.**data **=** data

self**.**next **=** **None**

**def** detectAndRemoveLoop(head):

slow **=** head

fast **=** head

*# Move slow and fast pointers until they meet or reach the end of the list*

**while** fast **and** fast**.**next:

slow **=** slow**.**next

fast **=** fast**.**next**.**next

**if** slow **==** fast:

**break**

*# If fast pointer reached the end, there is no loop in the list*

**if** fast **is** **None** **or** fast**.**next **is** **None**:

**return** head

*# Reset slow pointer to the head and move both pointers by one step until they meet again*

slow **=** head

**while** slow**.**next **!=** fast**.**next:

slow **=** slow**.**next

fast **=** fast**.**next

*# Break the loop by setting the next pointer of the node where slow and fast pointers meet to None*

fast**.**next **=** **None**

**return** 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

Code:

class Node:

**def** \_\_init\_\_(self, data**=None**):

self**.**data **=** data

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

*# Add 1 to the first digit and propagate carry if needed*

current **=** head

carry **=** 1

**while** current **and** carry **>** 0:

sum **=** current**.**data **+** carry

current**.**data **=** sum **%** 10

carry **=** sum **//** 10

current **=** current**.**next

*# Reverse the linked list again*

prev **=** **None**

current **=** head

**while** current:

next\_node **=** current**.**next

current**.**next **=** prev

prev **=** current

current **=** next\_node

head **=** prev

**return** head

**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.)

**CODE:**

**def** merge(a, b):

**if** **not** a:

**return** b

**if** **not** b:

**return** a

result **=** **None**

**if** a**.**data **<=** b**.**data:

result **=** a

result**.**bottom **=** merge(a**.**bottom, b)

**else**:

result **=** b

result**.**bottom **=** merge(a, b**.**bottom)

result**.**next **=** **None**

**return** result

**def** flatten(head):

**if** **not** head **or** **not** head**.**next:

**return** head

head**.**next **=** flatten(head**.**next)

head **=** merge(head, head**.**next)

**return** head

**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.

!<https://contribute.geeksforgeeks.org/wp-content/uploads/clone.jpg>

**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.

**CODE:**

class Node:

**def** \_\_init\_\_(self, data**=None**):

self**.**data **=** data

self**.**next **=** **None**

self**.**random **=** **None**

**def** copyRandomList(head):

**if** **not** head:

**return** **None**

*# Step 1: Create new nodes with the same values*

current **=** head

**while** current:

new\_node **=** Node(current**.**data)

new\_node**.**next **=** current**.**next

current**.**next **=** new\_node

current **=** new\_node**.**next

*# Step 2: Set random pointers of new nodes*

current **=** head

**while** current:

**if** current**.**random:

current**.**next**.**random **=** current**.**random**.**next

current **=** current**.**next**.**next

*# Step 3: Separate original and copied lists*

current **=** head

new\_head **=** head**.**next

**while** current:

new\_node **=** current**.**next

current**.**next **=** new\_node**.**next

**if** new\_node**.**next:

new\_node**.**next **=** new\_node**.**next**.**next

current **=** current**.**next

**return** new\_head

**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:**

!<https://assets.leetcode.com/uploads/2021/03/10/oddeven-linked-list.jpg>

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

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

**Example 2:**

!<https://assets.leetcode.com/uploads/2021/03/10/oddeven2-linked-list.jpg>

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

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

**CODE:**

**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

odd\_head **=** head

even\_head **=** head**.**next

odd **=** odd\_head

even **=** even\_head

**while** even **and** even**.**next:

odd**.**next **=** even**.**next

odd **=** odd**.**next

even**.**next **=** odd**.**next

even **=** even**.**next

odd**.**next **=** even\_head

**return** odd\_head

**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

**CODE:**

**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 **or** k **==** 0:

**return** head

length **=** 0

current **=** head

**while** current:

length **+=** 1

current **=** current**.**next

**if** k **==** length **or** k **%** length **==** 0:

**return** head

k **=** k **%** length

prev **=** head

curr **=** head

**for** \_ **in** range(k):

curr **=** curr**.**next

new\_head **=** curr

**while** curr**.**next:

curr **=** curr**.**next

curr**.**next **=** head

prev**.**next **=** **None**

**return** new\_head

**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.

**CODE:**

**class** ListNode:

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

self**.**val **=** val

self**.**next **=** next

**def** nextLargerNodes(head):

*# Convert the linked list into a list*

values **=** []

current **=** head

**while** current:

values**.**append(current**.**val)

current **=** current**.**next

stack **=** []

result **=** []

*# Iterate over the list in reverse order*

**for** val **in** reversed(values):

**while** stack **and** stack[**-**1] **<=** val:

stack**.**pop()

**if** **not** stack:

result**.**append(0)

**else**:

result**.**append(stack[**-**1])

stack**.**append(val)

*# Reverse the result list*

result**.**reverse()

**return** result

**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]

**CODE:**

**class** ListNode:

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

self**.**val **=** val

self**.**val **=** val

self**.**next **=** next

**def** removeZeroSumSublists(head):

*# Create a dummy node as the new head*

dummy **=** ListNode(0)

dummy**.**next **=** head

*# Initialize prefix sum variables*

prefix\_sum **=** 0

prefix\_sum\_map **=** {0: dummy}

current **=** head

*# Traverse the linked list*

**while** current:

prefix\_sum **+=** current**.**val

*# Check if the prefix sum is already in the map*

**if** prefix\_sum **in** prefix\_sum\_map:

*# Get the node pointer from the map*

node\_before\_sum **=** prefix\_sum\_map[prefix\_sum]

*# Update the next pointer to skip the nodes in the sequence*

node **=** node\_before\_sum**.**next

temp\_sum **=** prefix\_sum **+** node**.**val

**while** node **!=** current:

prefix\_sum\_map**.**pop(temp\_sum)

node **=** node**.**next

temp\_sum **+=** node**.**val

node\_before\_sum**.**next **=** current**.**next

*# Add the prefix sum and node pointer to the map*

prefix\_sum\_map[prefix\_sum] **=** current

current **=** current**.**next

**return** dummy**.**next