**DSA – ASSIGNMENT 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.

**Solution. :-**

* Initialize two pointers, slow and fast, pointing to the head of the linked list.
* Move slow one step at a time and fast two steps at a time.
* If the fast pointer encounters a null node or reaches the end of the list, it means there is no loop. Return the linked list as it is.
* If the slow and fast pointers meet at some point, it confirms the presence of a loop.
* To find the loop node, reset either the slow or fast pointer to the head of the linked list.
* Move both pointers one step at a time until they meet again. The meeting point is the loop node.
* Once the loop node is found, set the next pointer of the node preceding the loop node to null to break the loop.

**class Node:**

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

**self.data = data**

**self.next = None**

**def removeLoop(head):**

**slow = head**

**fast = head**

**# Check if loop exists**

**while fast and fast.next:**

**slow = slow.next**

**fast = fast.next.next**

**if slow == fast:**

**break**

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

**if slow != fast:**

**return head**

**# Find the loop node**

**slow = head**

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

**slow = slow.next**

**fast = fast.next**

**# Remove the loop**

**fast.next = None**

**return head**

**# Create the linked list with a loop**

**head = Node(1)**

**node1 = Node(3)**

**node2 = Node(4)**

**head.next = node1**

**node1.next = node2**

**node2.next = node1 # Loop**

**# Remove the loop from the linked list**

**head = removeLoop(head)**

**# Print the modified linked list**

**current = head**

**while current:**

**print(current.data, end=" ")**

**current = current.next**

💡 **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

**Solution. :-**

* Initialize a carry variable to 1, as we need to add 1 to the number.
* Traverse the linked list in reverse order (from the last node to the first node):
  + Get the sum of the current node's value and the carry.
  + Update the current node's value with the remainder of the sum divided by 10.
  + Update the carry by dividing the sum by 10.
* After traversing the entire linked list, if the carry is still 1, create a new node with the value 1 and append it at the end of the linked list.
* Reverse the linked list to restore the original order.
* Return the head of the modified linked list.

**class Node:**

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

**self.data = data**

**self.next = None**

**def addOne(head):**

**if not head:**

**return None**

**# Reverse the linked list**

**prev = None**

**current = head**

**while current:**

**next\_node = current.next**

**current.next = prev**

**prev = current**

**current = next\_node**

**# Add 1 to the number**

**carry = 1**

**current = prev**

**while current:**

**sum = current.data + carry**

**current.data = sum % 10**

**carry = sum // 10**

**if carry == 0:**

**break**

**current = current.next**

**# If carry is still 1, append a new node with value 1**

**if carry == 1:**

**new\_node = Node(1)**

**current.next = new\_node**

**# Reverse the linked list again**

**new\_head = None**

**while prev:**

**next\_node = prev.next**

**prev.next = new\_head**

**new\_head = prev**

**prev = next\_node**

**return new\_head**

**# Create the linked list**

**head = Node(4)**

**node1 = Node(5)**

**node2 = Node(6)**

**head.next = node1**

**node1.next = node2**

**# Add 1 to the linked list**

**head = addOne(head)**

**# Print the modified linked list**

**current = head**

**while current:**

**print(current.data, end="")**

**current = current.next**

💡 **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.)

**Solution. :-**

* Initialize a new head pointer as None and a tail pointer as None.
* Traverse the main linked list:
  + For each node in the main linked list, traverse its bottom linked list:
  + If the new list is empty, set the head and tail pointers to the first node of the bottom list.
  + Otherwise, append each node of the bottom list to the tail of the new list and update the tail pointer.
* Return the head of the new flattened list.

**class Node:**

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

**self.data = data**

**self.next = None**

**self.bottom = None**

**def mergeLists(head1, head2):**

**if not head1:**

**return head2**

**if not head2:**

**return head1**

**merged\_head = None**

**if head1.data < head2.data:**

**merged\_head = head1**

**merged\_head.bottom = mergeLists(head1.bottom, head2)**

**else:**

**merged\_head = head2**

**merged\_head.bottom = mergeLists(head1, head2.bottom)**

**return merged\_head**

**def flattenLinkedList(head):**

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

**return head**

**# Recursively flatten the remaining linked lists**

**head.next = flattenLinkedList(head.next)**

**# Merge the current list with the flattened list**

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

**return head**

**def printLinkedList(head):**

**current = head**

**while current:**

**print(current.data, end="->")**

**current = current.bottom**

**print("None")**

**# Create the linked list**

**head = Node(5)**

**node1 = Node(10)**

**node2 = Node(19)**

**node3 = Node(28)**

**head.next = node1**

**node1.next = node2**

**node2.next = node3**

**head.bottom = Node(7)**

**node1.bottom = Node(20)**

**node2.bottom = Node(22)**

**node3.bottom = Node(35)**

**head.bottom.bottom = Node(8)**

**node2.bottom.bottom = Node(50)**

**node3.bottom.bottom = Node(40)**

**head.bottom.bottom.bottom = Node(30)**

**node3.bottom.bottom.bottom = Node(45)**

**# Flatten the linked list**

**head = flattenLinkedList(head)**

**# Print the flattened linked list**

**printLinkedList(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.

**Solution. :-**

* 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 the corresponding original node.
* While traversing the original list, create a mapping between each original node and its corresponding new node.
* Traverse the original list again and set the next and random pointers of each new node based on the mapping created in step 2.
* Return the head of the new list.

**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 new nodes and map original nodes to new nodes**

**node\_map = {}**

**current = head**

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

**new\_current = new\_head**

**node\_map[current] = new\_current**

**while current.next:**

**current = current.next**

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

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

**new\_current = new\_node**

**node\_map[current] = new\_current**

**# Step 2: Set the next and random pointers of new nodes**

**current = head**

**new\_current = new\_head**

**while current:**

**new\_current.random = node\_map.get(current.random)**

**current = current.next**

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

**return new\_head**

**# Create the original linked list**

**head = Node(1)**

**node2 = Node(2)**

**node3 = Node(3)**

**node4 = Node(4)**

**node5 = Node(5)**

**head.next = node2**

**head.random = node3**

**node2.next = node3**

**node2.random = node1**

**node3.next = node4**

**node3.random = node5**

**node4.next = node5**

**node4.random = node2**

**node5.random = node4**

**# Create a copy of the linked list**

**new\_head = copyRandomList(head)**

**# Print the original and copied linked lists**

**print("Original Linked List:")**

**current = head**

**while current:**

**print("Node:", current.data, "Next:", current.next.data if current.next else "None", "Random:", current.random.data if current.random else "None")**

**current = current.next**

**print("\nCopied Linked List:")**

**current = new\_head**

**while current:**

**print("Node:", current.data, "Next:", current.next.data if current.next else "None", "Random:", current.random.data if current.random else "None")**

**current = current.next**

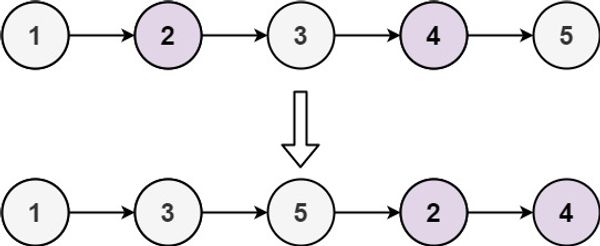
💡 **Question 5** Given the head of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return the reordered list.

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

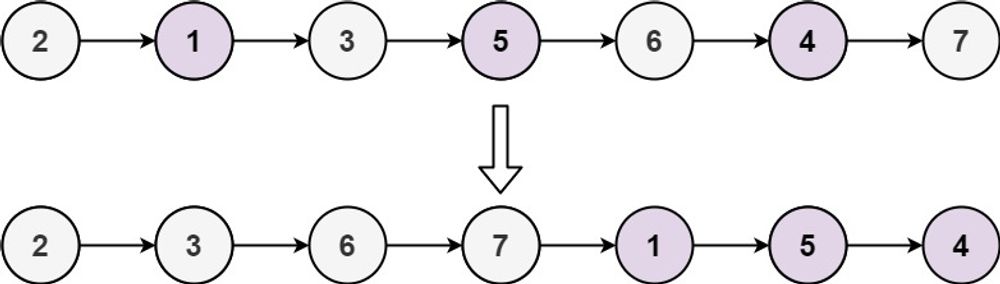
Note that the relative order inside both the even and odd groups should remain as it was in the input.

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]

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

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

**Solution. :-**

* Create two separate linked lists, one for odd-indexed nodes and one for even-indexed nodes.
* Traverse the original linked list and iterate over each node, keeping track of the current node and the next node.
* For each node, append it to the corresponding odd or even list based on its index.
* After traversing the original list, append the even list to the end of the odd list.
* Finally, set the next pointer of the last node in the even list to None to terminate the merged list.

**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**

**# Separate odd and even nodes**

**odd\_head = head**

**even\_head = head.next**

**odd\_tail = odd\_head**

**even\_tail = even\_head**

**while even\_tail and even\_tail.next:**

**odd\_tail.next = even\_tail.next**

**odd\_tail = odd\_tail.next**

**even\_tail.next = odd\_tail.next**

**even\_tail = even\_tail.next**

**# Merge odd and even lists**

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

**return odd\_head**

**# Create the original linked list**

**head = ListNode(1)**

**node2 = ListNode(2)**

**node3 = ListNode(3)**

**node4 = ListNode(4)**

**node5 = ListNode(5)**

**head.next = node2**

**node2.next = node3**

**node3.next = node4**

**node4.next = node5**

**# Reorder the linked list**

**reordered\_head = oddEvenList(head)**

**# Print the reordered linked list**

**current = reordered\_head**

**while current:**

**print(current.val, end=" ")**

**current = current.next**

💡 **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

**Solution. :-**

* Find the kth node from the beginning of the linked list. This 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.
* Update the next pointer of the kth node to None, as it will be the new tail of the shifted list.

**class ListNode:**

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

**self.val = val**

**self.next = next**

**def leftShiftLinkedList(head, k):**

**if not head or not head.next or k == 0:**

**return head**

**# Find the kth node from the beginning**

**current = head**

**count = 1**

**while count < k and current:**

**current = current.next**

**count += 1**

**if not current:**

**return head**

**new\_head = current**

**# Traverse to the end of the linked list**

**while current.next:**

**current = current.next**

**# Connect the last node to the original head**

**current.next = head**

**# Update the next pointer of the kth node to None**

**head = new\_head.next**

**new\_head.next = None**

**return head**

**# Create the linked list**

**head = ListNode(2)**

**node2 = ListNode(4)**

**node3 = ListNode(7)**

**node4 = ListNode(8)**

**node5 = ListNode(9)**

**head.next = node2**

**node2.next = node3**

**node3.next = node4**

**node4.next = node5**

**# Shift the linked list by 3 nodes**

**k = 3**

**shifted\_head = leftShiftLinkedList(head, k)**

**# Print the shifted linked list**

**current = shifted\_head**

**while current:**

**print(current.val, end=" ")**

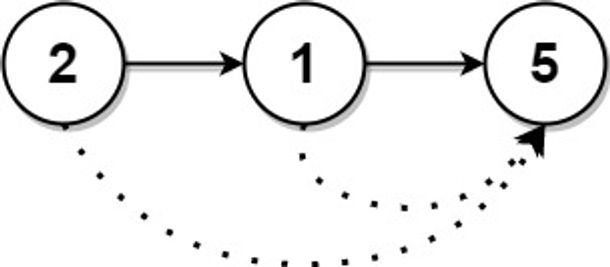
**current = current.next**

💡 **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.

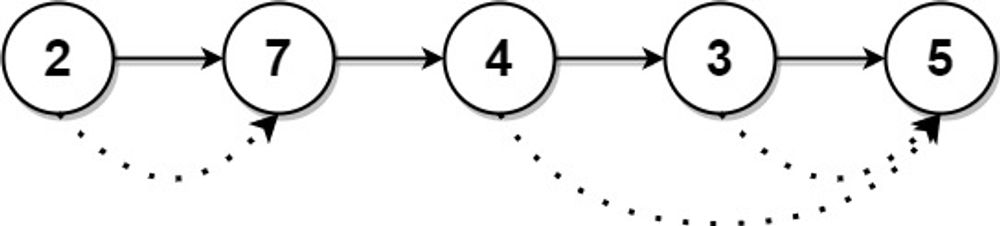
Return an integer array answer where answer[i] is the value of the next greater node of the ith node (**1-indexed**). If the ith node does not have a next greater node, set answer[i] = 0.

**Example 1:**



Input: head = [2,1,5]

Output: [5,5,0]



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

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

**Solution. :-**

* Convert the linked list into an array for easier manipulation.
* Initialize an array answer of the same length as the input array, with all elements set to 0.
* Initialize an empty stack to store the indices of the array elements.
* Iterate through the array from right to left:
  + While the stack is not empty and the current element is greater than the element at the index stored at the top of the stack, update the answer at that index with the current element.
  + Push the index of the current element into the stack.
* Return the answer array.

**class ListNode:**

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

**self.val = val**

**self.next = next**

**def nextGreaterNodes(head):**

**# Convert the linked list to an array**

**arr = []**

**current = head**

**while current:**

**arr.append(current.val)**

**current = current.next**

**n = len(arr)**

**answer = [0] \* n**

**stack = []**

**# Iterate through the array from right to left**

**for i in range(n - 1, -1, -1):**

**# Update the answer for elements in the stack that are smaller than the current element**

**while stack and arr[i] >= arr[stack[-1]]:**

**stack.pop()**

**# If there is a greater element in the stack, update the answer**

**if stack:**

**answer[i] = arr[stack[-1]]**

**# Push the current index into the stack**

**stack.append(i)**

**return answer**

**# Create the linked list**

**head = ListNode(2)**

**node2 = ListNode(1)**

**node3 = ListNode(5)**

**head.next = node2**

**node2.next = node3**

**# Find the next greater nodes**

**result = nextGreaterNodes(head)**

**# Print the result**

**print(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]

**Example 3:**

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

Output: [1]

**Solution. :-**

* Create a dummy node and set its next pointer to the head of the linked list.
* Initialize a variable prefix\_sum to keep track of the cumulative sum of nodes encountered so far. Also, initialize a hash map called sum\_map to store the cumulative sums as keys and their corresponding node references as values.
* Iterate through the linked list while updating the prefix\_sum and sum\_map:
  + Calculate the cumulative sum by adding the current node's value to the prefix\_sum.
  + Check if the prefix\_sum exists in the sum\_map:
    - If it exists, it means that there is a sequence of nodes from a previous occurrence of the same prefix\_sum to the current node that sums up to zero.
    - In this case, we need to remove the nodes between the previous occurrence and the current node from the linked list. Update the next pointer of the previous node to skip the nodes in between.
    - Remove the entries from the sum\_map for the cumulative sums between the previous occurrence and the current node.
  + Add the current prefix\_sum and its corresponding node reference to the sum\_map.
  + Move to the next node.
* Return the dummy.next as the head of the final linked list.

**Class ListNode:**

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

**self.val = val**

**self.next = next**

**def removeZeroSumSublists(head):**

**# Create a dummy node and set its next pointer to the head**

**dummy = ListNode(0)**

**dummy.next = head**

**# Initialize prefix\_sum and sum\_map**

**prefix\_sum = 0**

**sum\_map = {0: dummy}**

**# Iterate through the linked list**

**current = dummy.next**

**while current:**

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

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

**# Remove nodes between previous occurrence and current node**

**prev = sum\_map[prefix\_sum]**

**prev.next = current.next**

**# Remove entries from sum\_map between previous occurrence and current node**

**node = prev.next**

**while node != current:**

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

**del sum\_map[prefix\_sum]**

**node = node.next**

**# Continue to the next iteration**

**current = prev.next**

**else:**

**# Add the current prefix\_sum and its node reference to sum\_map**

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

**# Move to the next node**

**current = current.next**

**return dummy.next**

**# Create the linked list**

**head = ListNode(1)**

**node2 = ListNode(2)**

**node3 = ListNode(-3)**

**node4 = ListNode(3)**

**node5 = ListNode(1)**

**head.next = node2**

**node2.next = node3**

**node3.next = node4**

**node4.next = node5**

**# Remove zero sum sublists**

**result = removeZeroSumSublists(head)**

**# Print the result**

**while result:**

**print(result.val, end=' ')**

**result = result.next**