**DSA – ASSIGNMENT 13**

💡 **Question 1** Given two linked list of the same size, the task is to create a new linked list using those linked lists. The condition is that the greater node among both linked list will be added to the new linked list.

**Examples:**

Input: list1 = 5->2->3->8

list2 = 1->7->4->5

Output: New list = 5->7->4->8

Input:list1 = 2->8->9->3

list2 = 5->3->6->4

Output: New list = 5->8->9->4

**Solution. :-**

* Initialize a pointer, current1, to traverse the first linked list (list1).
* Initialize a pointer, current2, to traverse the second linked list (list2).
* Initialize a new linked list (newList) and a pointer, newCurrent, to keep track of the current node in the new list.
* Traverse both list1 and list2 simultaneously.
* For each pair of nodes, compare the values of the nodes.
* If the value of the node in list1 is greater, add it to the new list and move the current1 pointer to the next node in list1.
* If the value of the node in list2 is greater or equal, add it to the new list and move the current2 pointer to the next node in list2.
* Continue this process until the end of either list1 or list2 is reached.
* If there are any remaining nodes in list1 or list2, add them to the new list.
* Return the new list.

**class Node:**

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

**self.data = data**

**self.next = None**

**def mergeGreaterNodes(list1, list2):**

**if not list1: # If list1 is empty, return list2**

**return list2**

**if not list2: # If list2 is empty, return list1**

**return list1**

**current1 = list1**

**current2 = list2**

**newList = None**

**newCurrent = None**

**while current1 and current2:**

**if current1.data >= current2.data:**

**if not newList:**

**newList = Node(current1.data)**

**newCurrent = newList**

**else:**

**newCurrent.next = Node(current1.data)**

**newCurrent = newCurrent.next**

**current1 = current1.next**

**else:**

**if not newList:**

**newList = Node(current2.data)**

**newCurrent = newList**

**else:**

**newCurrent.next = Node(current2.data)**

**newCurrent = newCurrent.next**

**current2 = current2.next**

**# Add any remaining nodes from list1**

**while current1:**

**newCurrent.next = Node(current1.data)**

**newCurrent = newCurrent.next**

**current1 = current1.next**

**# Add any remaining nodes from list2**

**while current2:**

**newCurrent.next = Node(current2.data)**

**newCurrent = newCurrent.next**

**current2 = current2.nex**

**return newList**

**# Create the first linked list**

**list1 = Node(5)**

**list1.next = Node(2)**

**list1.next.next = Node(3)**

**list1.next.next.next = Node(8)**

**# Create the second linked list**

**list2 = Node(1)**

**list2.next = Node(7)**

**list2.next.next = Node(4)**

**list2.next.next.next = Node(5)**

**# Merge the greater nodes from list1 and list2 into a new linked list**

**newList = mergeGreaterNodes(list1, list2)**

**# Print the new linked list**

**current = newList**

**while current:**

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

**current = current.next**

💡 **Question 2** Write a function that takes a list sorted in non-decreasing order and deletes any duplicate nodes from the list. The list should only be traversed once.

For example if the linked list is 11->11->11->21->43->43->60 then removeDuplicates() should convert the list to 11->21->43->60.

**Example 1:**

Input:

LinkedList:

11->11->11->21->43->43->60

Output:

11->21->43->60

**Example 2:**

Input:

LinkedList:

10->12->12->25->25->25->34

Output:

10->12->25->34

**Solution. :-**

* Initialize a pointer, current, to traverse the linked list.
* While the current node and its next node are not None:
  + If the data of the current node is equal to the data of its next node, skip the next node by updating the current node's next pointer to the node after the next node.
  + If the data of the current node is not equal to the data of its next node, move the current node to its next node.
* Return the modified linked list.

**class Node:**

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

**self.data = data**

**self.next = None**

**def removeDuplicates(head):**

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

**return head**

**current = head**

**while current and current.next:**

**if current.data == current.next.data:**

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

**else:**

**current = current.next**

**return head**

**# Create the linked list**

**head = Node(11)**

**node2 = Node(11)**

**node3 = Node(11)**

**node4 = Node(21)**

**node5 = Node(43)**

**node6 = Node(43)**

**node7 = Node(60)**

**head.next = node2**

**node2.next = node3**

**node3.next = node4**

**node4.next = node5**

**node5.next = node6**

**node6.next = node7**

**# Remove duplicate nodes from the linked list**

**head = removeDuplicates(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**. The task is to reverse every **k** nodes (where k is an input to the function) in the linked list. If the number of nodes is not a multiple of *k* then left-out nodes, in the end, should be considered as a group and must be reversed (See Example 2 for clarification).

**Example 1:**

Input:

LinkedList: 1->2->2->4->5->6->7->8

K = 4

Output:4 2 2 1 8 7 6 5

Explanation:

The first 4 elements 1,2,2,4 are reversed first and then the next 4 elements 5,6,7,8. Hence, the resultant linked list is 4->2->2->1->8->7->6->5.

**Example 2:**

Input:

LinkedList: 1->2->3->4->5

K = 3

Output:3 2 1 5 4

Explanation:

The first 3 elements are 1,2,3 are reversed first and then elements 4,5 are reversed.Hence, the resultant linked list is 3->2->1->5->4.

**Solution. :-**

* Initialize three pointers: current, prev, and nextNode.
* Traverse the linked list with the current pointer, reversing every k nodes.
* For each group of k nodes:
  + Initialize prev as None.
  + Reverse the current group of k nodes by iterating k times:
    - Assign the nextNode pointer as the next node of the current node.
    - Update the next node of the current node to be prev.
    - Move prev to the current node.
    - Move the current node to the nextNode.
* If there are more nodes remaining in the list (less than k), reverse them as a separate group.
* Update the head of the linked list to point to the prev node.
* Return the modified linked list.

**class Node:**

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

**self.data = data**

**self.next = None**

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

**if not head or not head.next or k == 1:**

**return head**

**dummy = Node(0)**

**dummy.next = head**

**prev = dummy**

**current = head**

**count = 0**

**while current:**

**count += 1**

**if count % k == 0:**

**prev = reverseGroup(prev, current.next)**

**current = prev.next**

**else:**

**current = current.next**

**return dummy.next**

**def reverseGroup(prev, nextNode):**

**lastNode = prev.next**

**current = lastNode.next**

**while current != nextNode:**

**lastNode.next = current.next**

**current.next = prev.next**

**prev.next = current**

**current = lastNode.next**

**return lastNode**

**# Create the linked list**

**head = Node(1)**

**node2 = Node(2)**

**node3 = Node(2)**

**node4 = Node(4)**

**node5 = Node(5)**

**node6 = Node(6)**

**node7 = Node(7)**

**node8 = Node(8)**

**head.next = node2**

**node2.next = node3**

**node3.next = node4**

**node4.next = node5**

**node5.next = node6**

**node6.next = node7**

**node7.next = node8**

**# Reverse every k nodes in the linked list**

**k = 4**

**head = reverseKNodes(head, k)**

**# Print the modified linked list**

**current = head**

**while current:**

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

**current = current.next**

💡 **Question 4** Given a linked list, write a function to reverse every alternate k nodes (where k is an input to the function) in an efficient way. Give the complexity of your algorithm.

**Example:**

Inputs: 1->2->3->4->5->6->7->8->9->NULL and k = 3

Output: 3->2->1->4->5->6->9->8->7->NULL.

**Solution. :-**

* Initialize three pointers: current, prev, and nextNode.
* Traverse the linked list with the current pointer, reversing every alternate k nodes.
* For each group of k nodes:
  + Initialize prev as None.
  + Reverse the current group of k nodes by iterating k times:
  + Assign the nextNode pointer as the next node of the current node.
  + Update the next node of the current node to be prev.
  + Move prev to the current node.
  + Move the current node to the nextNode.
* If there are more nodes remaining in the list (less than k), do not reverse them.
* After reversing the alternate k nodes, connect the last node of the reversed group to the next node of the current group.
* Return the modified linked list.

**class Node:**

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

**self.data = data**

**self.next = None**

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

**if not head or not head.next or k == 1:**

**return head**

**dummy = Node(0)**

**dummy.next = head**

**prev = dummy**

**current = head**

**count = 0**

**shouldReverse = False**

**while current:**

**count += 1**

**if count % k == 0:**

**prev, current = reverseGroup(prev, current.next, shouldReverse)**

**shouldReverse = not shouldReverse**

**else:**

**current = current.next**

**return dummy.next**

**def reverseGroup(prev, nextNode, shouldReverse):**

**lastNode = prev.next**

**current = lastNode.next**

**prev.next = lastNode**

**if shouldReverse:**

**while current != nextNode:**

**lastNode.next = current.next**

**current.next = prev.next**

**prev.next = current**

**current = lastNode.next**

**else:**

**prev.next = lastNode.next**

**return lastNode, prev**

**# Create the linked list**

**head = Node(1)**

**node2 = Node(2)**

**node3 = Node(3)**

**node4 = Node(4)**

**node5 = Node(5)**

**node6 = Node(6)**

**node7 = Node(7)**

**node8 = Node(8)**

**node9 = Node(9)**

**head.next = node2**

**node2.next = node3**

**node3.next = node4**

**node4.next = node5**

**node5.next = node6**

**node6.next = node7**

**node7.next = node8**

**node8.next = node9**

**# Reverse every alternate k nodes in the linked list**

**k = 3**

**head = reverseAlternateKNodes(head, k)**

**# Print the modified linked list**

**current = head**

**while current:**

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

**current = current.next**

💡 **Question 5** Given a linked list and a key to be deleted. Delete last occurrence of key from linked. The list may have duplicates.

**Examples**:

Input: 1->2->3->5->2->10, key = 2

Output: 1->2->3->5->10

**Solution. :-**

* Initialize three pointers: prevToDelete, toDelete, and lastKeyNode.
* Traverse the linked list and keep track of the last occurrence of the key:
  + If the current node's data is equal to the key, update the lastKeyNode pointer to the current node.
  + Update the prevToDelete pointer to the previous node.
* If the lastKeyNode is None, it means the key was not found in the list. In this case, return the original linked list.
* If lastKeyNode is the head of the linked list, update the head to the next node.
* Otherwise, update the next pointer of the previous node (prevToDelete) to skip the lastKeyNode.
* Return the modified linked list.

**class Node:**

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

**self.data = data**

**self.next = None**

**def deleteLastOccurrence(head, key):**

**if not head:**

**return head**

**prevToDelete = None**

**toDelete = None**

**lastKeyNode = None**

**current = head**

**while current:**

**if current.data == key:**

**lastKeyNode = current**

**current = current.next**

**if not lastKeyNode:**

**return head**

**if lastKeyNode == head:**

**head = head.next**

**else:**

**current = head**

**while current.next != lastKeyNode:**

**current = current.next**

**current.next = lastKeyNode.next**

**return head**

**# Create the linked list**

**head = Node(1)**

**node2 = Node(2)**

**node3 = Node(3)**

**node4 = Node(5)**

**node5 = Node(2)**

**node6 = Node(10)**

**head.next = node2**

**node2.next = node3**

**node3.next = node4**

**node4.next = node5**

**node5.next = node6**

**# Delete the last occurrence of the key in the linked list**

**key = 2**

**head = deleteLastOccurrence(head, key)**

**# Print the modified linked list**

**current = head**

**while current:**

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

**current = current.next**

💡 **Question 6** Given two sorted linked lists consisting of **N** and **M** nodes respectively. The task is to merge both of the lists (in place) and return the head of the merged list.

**Examples:**

Input: a: 5->10->15, b: 2->3->20

Output: 2->3->5->10->15->20

Input: a: 1->1, b: 2->4

Output: 1->1->2->4

**Solution. :-**

* Create a dummy node as the head of the merged list.
* Initialize three pointers: current to track the current node in the merged list, a to track the current node in the first list, and b to track the current node in the second list.
* Compare the values of nodes in lists a and b, and assign the smaller value to the current node in the merged list.
* Move the current pointer in the merged list and the pointer in the corresponding list with the smaller value.
* Repeat steps 3-4 until one of the lists is exhausted.
* Append the remaining nodes of the non-empty list to the merged list.
* Return the head of the merged list (dummy.next).

**class Node:**

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

**self.data = data**

**self.next = None**

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

**dummy = Node(0)**

**current = dummy**

**while a and b:**

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

**current.next = a**

**a = a.next**

**else:**

**current.next = b**

**b = b.next**

**current = current.next**

**if a:**

**current.next = a**

**else:**

**current.next = b**

**return dummy.next**

**# Create the first sorted linked list**

**a = Node(5)**

**a.next = Node(10)**

**a.next.next = Node(15)**

**# Create the second sorted linked list**

**b = Node(2)**

**b.next = Node(3)**

**b.next.next = Node(20)**

**# Merge the two sorted linked lists**

**merged = mergeSortedLists(a, b)**

**# Print the merged linked list**

**current = merged**

**while current:**

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

**current = current.next**

💡 **Question 7** Given a **Doubly Linked List**, the task is to reverse the given Doubly Linked List.

**Example:**

Original Linked list 10 8 4 2

Reversed Linked list 2 4 8 10

**Solution. :-**

* Initialize three pointers: current to track the current node, prev to track the previous node, and nextNode to track the next node.
* Traverse the linked list and for each node:
  + Set nextNode to the next node of the current node.
  + Update the next and previous pointers of the current node to reverse the links.
  + Move prev to the current node.
  + Move current to the nextNode.
* After the traversal, update the head of the doubly linked list to prev, which will now be pointing to the last node.
* Return the updated head.

**class Node:**

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

**self.data = data**

**self.prev = None**

**self.next = None**

**def reverseDoublyLinkedList(head):**

**current = head**

**prev = None**

**while current:**

**nextNode = current.next**

**current.next = prev**

**current.prev = nextNode**

**prev = current**

**current = nextNode**

**head = prev**

**return head**

**# Create the doubly linked list**

**head = Node(10)**

**node1 = Node(8)**

**node2 = Node(4)**

**node3 = Node(2)**

**head.next = node1**

**node1.prev = head**

**node1.next = node2**

**node2.prev = node1**

**node2.next = node3**

**node3.prev = node2**

**# Reverse the doubly linked list**

**head = reverseDoublyLinkedList(head)**

**# Print the reversed doubly linked list**

**current = head**

**while current:**

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

**current = current.next**

💡 **Question 8** Given a doubly linked list and a position. The task is to delete a node from given position in a doubly linked list.

**Example 1:**

Input:

LinkedList = 1 <--> 3 <--> 4

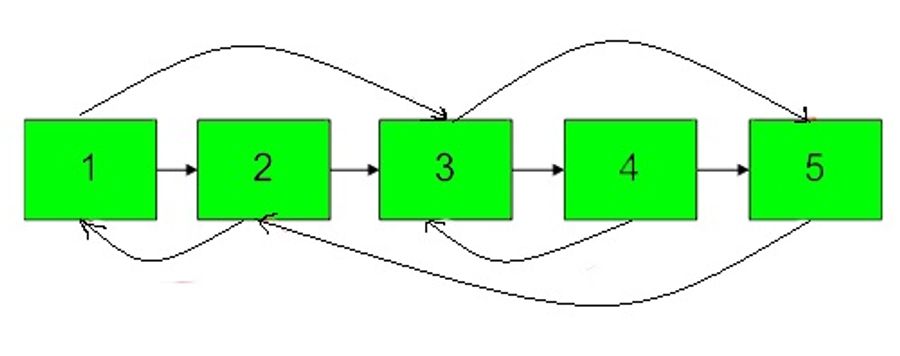
x = 3

Output:1 3

Explanation:

After deleting the node at position 3 (position starts from 1), the linked list will be now as 1->3.

**Example 2:**



**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.

**Solution. :-**

* If the doubly linked list is empty, return the list as it is.
* If the position is 1, it means the head node needs to be deleted. In this case:
  + Set the next node as the new head by updating its prev pointer to None.
  + Return the new head.
* Traverse the doubly linked list to find the node at the given position:
  + Initialize a counter variable to track the current position.
  + Start from the head and move to the next node until reaching the desired position or the end of the list.
* If the desired position is beyond the length of the list, no deletion is possible. Return the list as it is.
* If the node at the desired position is the last node, update the prev pointer of its previous node to None.
* Otherwise, update the next and prev pointers of the adjacent nodes to skip the node at the desired position.
* Return the head of the modified doubly linked list.

**class Node:**

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

**self.data = data**

**self.prev = None**

**self.next = None**

**def deleteNode(head, position):**

**if not head:**

**return head**

**if position == 1:**

**newHead = head.next**

**if newHead:**

**newHead.prev = None**

**return newHead**

**current = head**

**counter = 1**

**while current and counter < position:**

**current = current.next**

**counter += 1**

**if not current:**

**return head**

**if current.next:**

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

**if current.prev:**

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

**return head**

**# Create the doubly linked list**

**head = Node(1)**

**node1 = Node(3)**

**node2 = Node(4)**

**head.next = node1**

**node1.prev = head**

**node1.next = node2**

**node2.prev = node1**

**# Delete a node from a given position**

**position = 3**

**head = deleteNode(head, position)**

**# Print the modified doubly linked list**

**current = head**

**while current:**

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

**current = current.next**