**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

**Ans.**

**class** Node:

    # Function to initialise the node object

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

        self.data **=** data

        self.next **=** None

# Function to insert node in a linked list

**def** insert(root, item):

    temp **=** Node(0)

    temp.data **=** item

    temp.next **=** None

**if** (root **==** None):

        root **=** temp

**else** :

        ptr **=** root

**while** (ptr.next !**=** None):

            ptr **=** ptr.next

        ptr.next **=** temp

**return** root

# Function which returns new linked list

**def** newList(root1, root2):

    ptr1 **=** root1

    ptr2 **=** root2

    root **=** None

**while** (ptr1 !**=** None) :

        temp **=** Node(0)

        temp.next **=** None

        # Compare for greater node

**if** (ptr1.data < ptr2.data):

            temp.data **=** ptr2.data

**else**:

            temp.data **=** ptr1.data

**if** (root **==** None):

            root **=** temp

**else** :

            ptr **=** root

**while** (ptr.next !**=** None):

                ptr **=** ptr.next

            ptr.next **=** temp

        ptr1 **=** ptr1.next

        ptr2 **=** ptr2.next

**return** root

**def** display(root):

**while** (root !**=** None) :

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

        root **=** root.next

    print(" ");

# Driver Code

**if** \_\_name\_\_**==**'\_\_main\_\_':

    root1 **=** None

    root2 **=** None

    root **=** None

    # First linked list

    root1 **=** insert(root1, 5)

    root1 **=** insert(root1, 2)

    root1 **=** insert(root1, 3)

    root1 **=** insert(root1, 8)

**print**("First List: ", end **=** " ")

    display(root1)

    # Second linked list

    root2 **=** insert(root2, 1)

    root2 **=** insert(root2, 7)

    root2 **=** insert(root2, 4)

    root2 **=** insert(root2, 5)

**print**("Second List: ", end **=** " ")

    display(root2)

    root **=** newList(root1, root2)

**print**("New List: ", end **=** " ")

    display(root)

**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:

1. >21->43->60

**Ans.**

**class** Node:

    # Constructor to initialize

    # the node object

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

        self.data **=** data

        self.next **=** None

**class** LinkedList:

    # Function to initialize head

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

        self.head **=** None

    # Function to insert a new node

    # at the beginning

**def** push(self, new\_data):

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

        new\_node.next **=** self.head

        self.head **=** new\_node

    # Given a reference to the head of a

    # list and a key, delete the first

    # occurrence of key in linked list

**def** deleteNode(self, key):

        # Store head node

        temp **=** self.head

        # If head node itself holds the

        # key to be deleted

**if** (temp **is** **not** None):

**if** (temp.data **==** key):

                self.head **=** temp.next

                temp **=** None

**return**

        # Search for the key to be deleted,

        # keep track of the previous node as

        # we need to change 'prev.next'

**while**(temp **is** **not** None):

**if** temp.data **==** key:

**break**

            prev **=** temp

            temp **=** temp.next

        # if key was not present in

        # linked list

**if**(temp **==** None):

**return**

        # Unlink the node from linked list

        prev.next **=** temp.next

        temp **=** None

    # Utility function to print the

    # linked LinkedList

**def** printList(self):

        temp **=** self.head

**while**(temp):

            print(temp.data, end**=**' ')

            temp **=** temp.next

    # This function removes duplicates

    # from a sorted list

**def** removeDuplicates(self):

        temp **=** self.head

**if** temp **is** None:

**return**

**while** temp.next **is** **not** None:

**if** temp.data **==** temp.next.data:

                new **=** temp.next.next

                temp.next **=** None

                temp.next **=** new

**else**:

                temp **=** temp.next

**return** self.head

# Driver Code

llist **=** LinkedList()

llist.push(20)

llist.push(13)

llist.push(13)

llist.push(11)

llist.push(11)

llist.push(11)

print("Created Linked List: ")

llist.printList()

print()

print("Linked List after removing",

      "duplicate elements:")

llist.removeDuplicates()

llist.printList()

**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.

**Ans.**

class Node:

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

self.data = data

self.next = None

def reverse\_k\_nodes(head, k):

if head is None or k <= 1:

return head

current = head

next\_node = None

prev = None

count = 0

# Reverse the first k nodes

while current is not None and count < k:

next\_node = current.next

current.next = prev

prev = current

current = next\_node

count += 1

# Recursively reverse the remaining nodes

if next\_node is not None:

head.next = reverse\_k\_nodes(next\_node, k)

return prev

**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.

**Ans.**

**import** math

# Link list node

**class** Node:

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

        self.data **=** data

        self.next **=** None

# Reverses alternate k nodes and

#returns the pointer to the new head node

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

    current **=** head

    next **=** None

    prev **=** None

    count **=** 0

    #1) reverse first k nodes of the linked list

**while** (current !**=** None **and** count < k) :

        next **=** current.next

        current.next **=** prev

        prev **=** current

        current **=** next

        count **=** count **+** 1;

    # 2) Now head pos to the kth node.

    # So change next of head to (k+1)th node

**if**(head !**=** None):

        head.next **=** current

    # 3) We do not want to reverse next k

    # nodes. So move the current

    # pointer to skip next k nodes

    count **=** 0

**while**(count < k **-** 1 **and** current !**=** None ):

        current **=** current.next

        count **=** count **+** 1

    # 4) Recursively call for the list

    # starting from current.next. And make

    # rest of the list as next of first node

**if**(current !**=** None):

        current.next **=** kAltReverse(current.next, k)

    # 5) prev is new head of the input list

**return** prev

# UTILITY FUNCTIONS

# Function to push a node

**def** push(head\_ref, new\_data):

    # allocate node

    new\_node **=** Node(new\_data)

    # put in the data

    # new\_node.data = new\_data

    # link the old list of the new node

    new\_node.next **=** head\_ref

    # move the head to po to the new node

    head\_ref **=** new\_node

**return** head\_ref

# Function to print linked list

**def** printList(node):

    count **=** 0

**while**(node !**=** None):

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

        node **=** node.next

        count **=** count **+** 1

**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

**Ans.**

class Node:

def \_\_init\_\_(self, new\_data):

self.data = new\_data

self.next = None

def deleteLast(head, x):

temp = head

ptr = None

while (temp != None):

if (temp.data == x):

ptr = temp

temp = temp.next

if (ptr != None and ptr.next == None):

temp = head

while (temp.next != ptr):

temp = temp.next

temp.next = None

if (ptr != None and ptr.next != None):

ptr.data = ptr.next.data

temp = ptr.next

ptr.next = ptr.next.next

return head

def newNode(x):

node = Node(0)

node.data = x

node.next = None

return node

def display(head):

temp = head

if (head == None):

print("NULL\n")

return

while (temp != None):

print( temp.data, end = " ")

temp = temp.next

print("NULL")

head = newNode(1)

head.next = newNode(2)

head.next.next = newNode(7)

head.next.next.next = newNode(5)

head.next.next.next.next = newNode(2)

head.next.next.next.next.next = newNode(10)

print("Created Linked list: ", end = '')

display(head)

# Pass the address of the head pointer

head = deleteLast(head, 2)

print("List after deletion of 4: ", end = '')

display(head)

**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

**Ans.**

**class** Node:

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

        self.data **=** data

        self.next **=** None

# Constructor to initialize the node object

**class** LinkedList:

    # Function to initialize head

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

        self.head **=** None

    # Method to print linked list

**def** printList(self):

        temp **=** self.head

**while** temp:

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

            temp **=** temp.next

    # Function to add of node at the end.

**def** append(self, new\_data):

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

**if** self.head **is** None:

            self.head **=** new\_node

**return**

        last **=** self.head

**while** last.next:

            last **=** last.next

        last.next **=** new\_node

# Function to merge two sorted linked list.

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

    # create a temp node NULL

    temp **=** None

    # List1 is empty then return List2

**if** head1 **is** None:

**return** head2

    # if List2 is empty then return List1

**if** head2 **is** None:

**return** head1

    # If List1's data is smaller or

    # equal to List2's data

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

        # assign temp to List1's data

        temp **=** head1

        # Again check List1's data is smaller or equal List2's

        # data and call mergeLists function.

        temp.next **=** mergeLists(head1.next, head2)

**else**:

        # If List2's data is greater than or equal List1's

        # data assign temp to head2

        temp **=** head2

        # Again check List2's data is greater or equal List's

        # data and call mergeLists function.

        temp.next **=** mergeLists(head1, head2.next)

    # return the temp list.

**return** temp

**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

**Ans.**

**class** Node:

    # Constructor to create a new node

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

        self.data **=** data

        self.next **=** None

        self.prev **=** None

**class** DoublyLinkedList:

     # Constructor for empty Doubly Linked List

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

        self.head **=** None

    # Function reverse a Doubly Linked List

**def** reverse(self):

        temp **=** None

        current **=** self.head

        # Swap next and prev for all nodes of

        # doubly linked list

**while** current **is** **not** None:

            temp **=** current.prev

            current.prev **=** current.next

            current.next **=** temp

            current **=** current.prev

        # Before changing head, check for the cases like

        # empty list and list with only one node

**if** temp **is** **not** None:

            self.head **=** temp.prev

    # Given a reference to the head of a list and an

    # integer,inserts a new node on the front of list

**def** push(self, new\_data):

        # 1. Allocates node

        # 2. Put the data in it

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

        # 3. Make next of new node as head and

        # previous as None (already None)

        new\_node.next **=** self.head

        # 4. change prev of head node to new\_node

**if** self.head **is** **not** None:

            self.head.prev **=** new\_node

        # 5. move the head to point to the new node

        self.head **=** new\_node

**def** printList(self, node):

**while**(node **is** **not** None):

            print(node.data, end**=**' ')

            node **=** node.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.

**Ans.**

**class** Node:

    # Constructor to create a new node

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

        self.data **=** data

        self.next **=** None

        self.prev **=** None

# Function to delete a node in a Doubly Linked List.

# head\_ref -. pointer to head node pointer.

# del -. pointer to node to be deleted.

**def** deleteNode(head\_ref, del\_):

    # base case

**if** (head\_ref **==** None **or** del\_ **==** None):

**return**

    # If node to be deleted is head node

**if** (head\_ref **==** del\_):

        head\_ref **=** del\_.next

    # Change next only if node to be deleted is NOT

    # the last node

**if** (del\_.next !**=** None):

        del\_.next.prev **=** del\_.prev

    # Change prev only if node to be deleted is NOT

    # the first node

**if** (del\_.prev !**=** None):

        del\_.prev.next **=** del\_.next

**return** head\_ref

# Function to delete the node at the given position

# in the doubly linked list

**def** deleteNodeAtGivenPos(head\_ref,n):

    # if list in None or invalid position is given

**if** (head\_ref **==** None **or** n <**=** 0):

**return**

    current **=** head\_ref

    i **=** 1

    # traverse up to the node at position 'n' from

    # the beginning

**while** ( current !**=** None **and** i < n ):

        current **=** current.next

        i **=** i **+** 1

    # if 'n' is greater than the number of nodes

    # in the doubly linked list

**if** (current **==** None):

**return**

    # delete the node pointed to by 'current'

    deleteNode(head\_ref, current)

**return** head\_ref

# Function to insert a node at the beginning

# of the Doubly Linked List

**def** push(head\_ref, new\_data):

    # allocate node

    new\_node **=** Node(0)

    # put in the data

    new\_node.data **=** new\_data

    # since we are adding at the beginning,

    #prev is always None

    new\_node.prev **=** None

    # link the old list of the new node

    new\_node.next **=** (head\_ref)

    # change prev of head node to new node

**if** ((head\_ref) !**=** None):

        (head\_ref).prev **=** new\_node

    # move the head to point to the new node

    (head\_ref) **=** new\_node

**return** head\_ref

# Function to print nodes in a given doubly

# linked list

**def** printList(head):

**while** (head !**=** None) :

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

        head **=** head.next