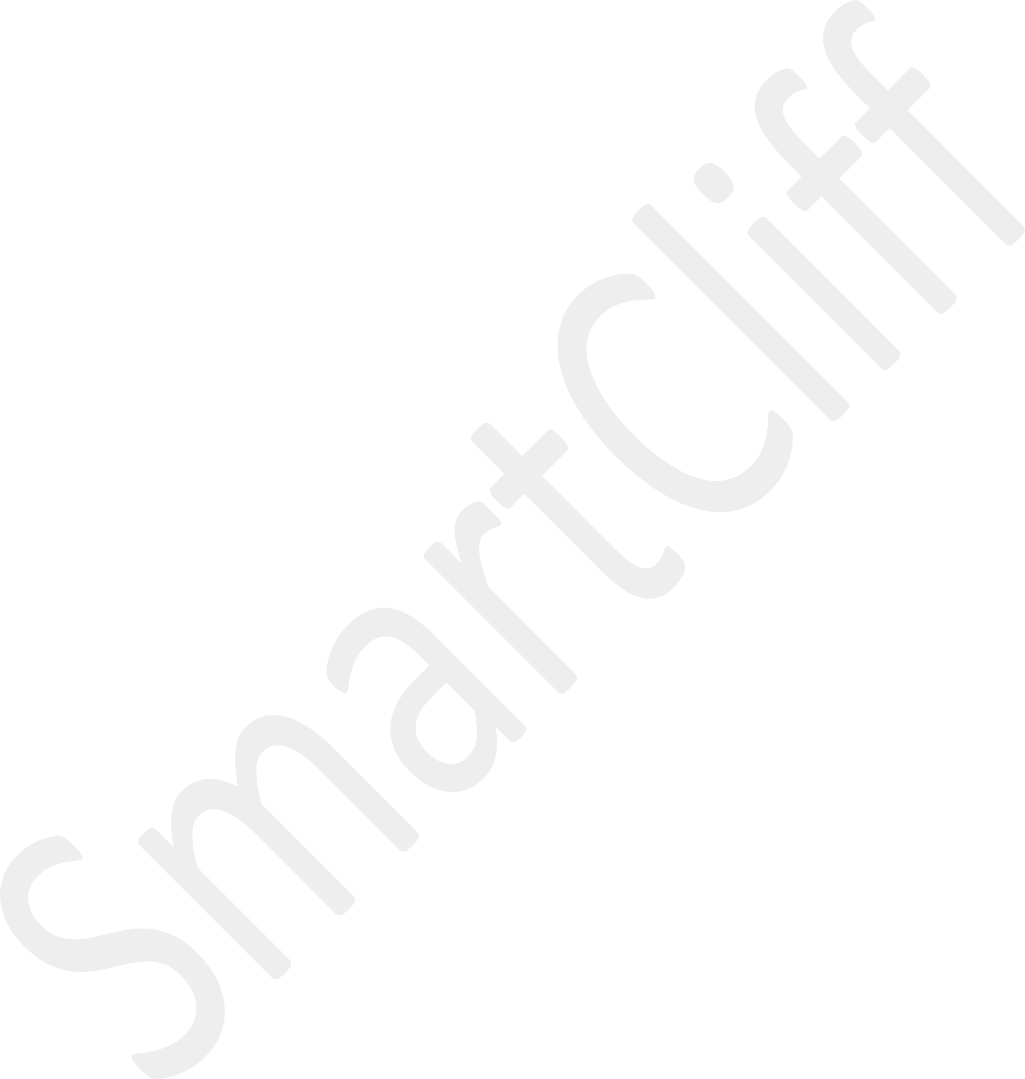
Practice No. 10

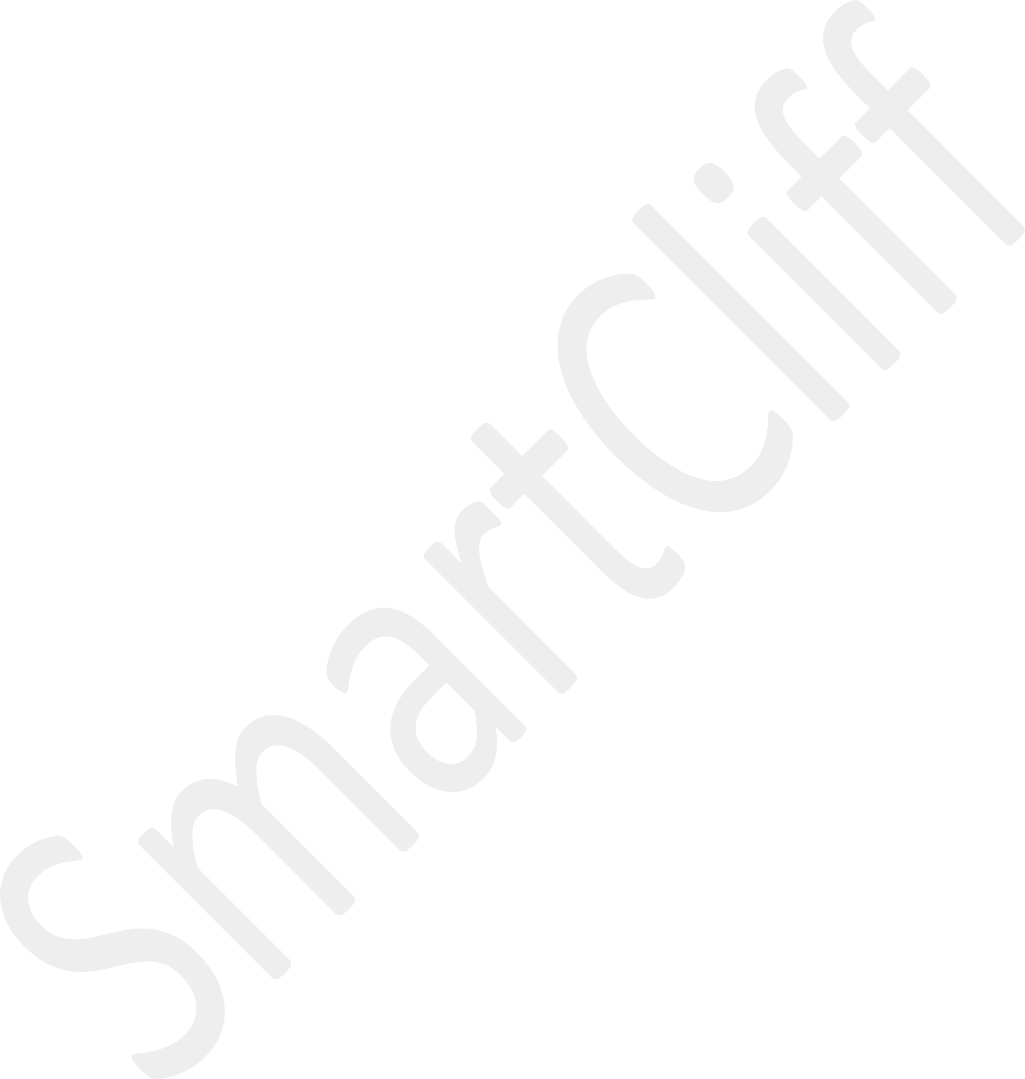
Topic : Linked List

Date : 10-07-2024

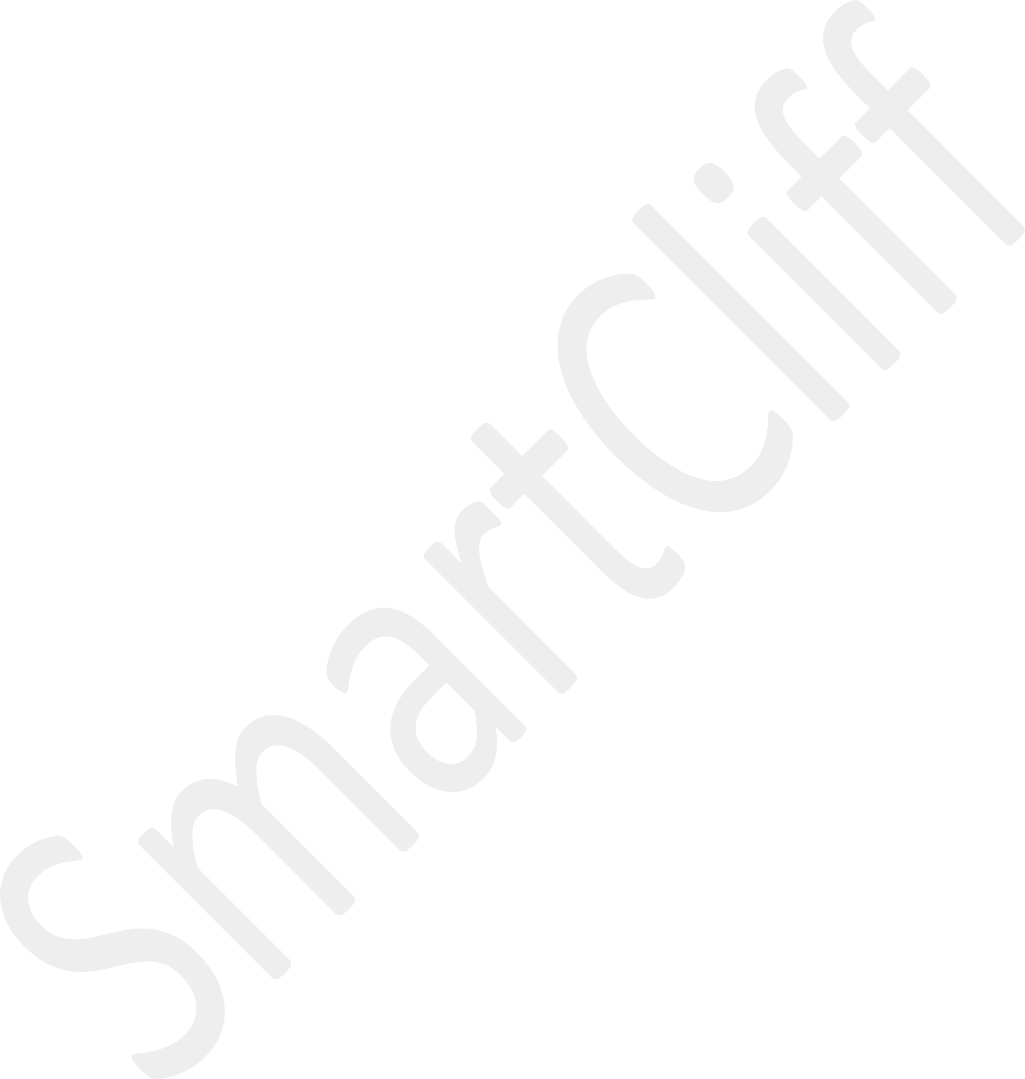
Solve the following problems

|  |  |  |
| --- | --- | --- |
| **Q.**  **No.** | **Question Detail** | **Level** |
| **1** | **Find the** [**N-th Node From The End**](https://www.naukri.com/code360/problems/nth-node-from-end_920751?interviewProblemRedirection=true) | Easy |
|  | **Problem statement** |  |
|  | You are given a Singly Linked List of integers. You have to find the N-th node from |  |
|  | end. |  |
|  | **For Example** |  |
|  | If the given list is (1 -> -2 -> 0 -> 4) and N=2: |  |
|  |  |  |
|  | Then the 2nd node from the end is 0. |  |
|  | **Input 1** |  |
|  | 2 |  |
|  | 1 -2 0 4 -1 |  |
|  | 2 |  |
|  | 9 9 -1 |  |
|  | 1 |  |
|  | **Output 1** |  |
|  | 0 |  |
|  | 9 |  |
|  | **Explanation for Input 1** |  |
|  | In the 1st test case, the 2nd node from the end is 0. |  |
|  | In the 2nd test case, the 1st node from the end is 9. |  |
|  | **Input 2** |  |
|  | 2 |  |

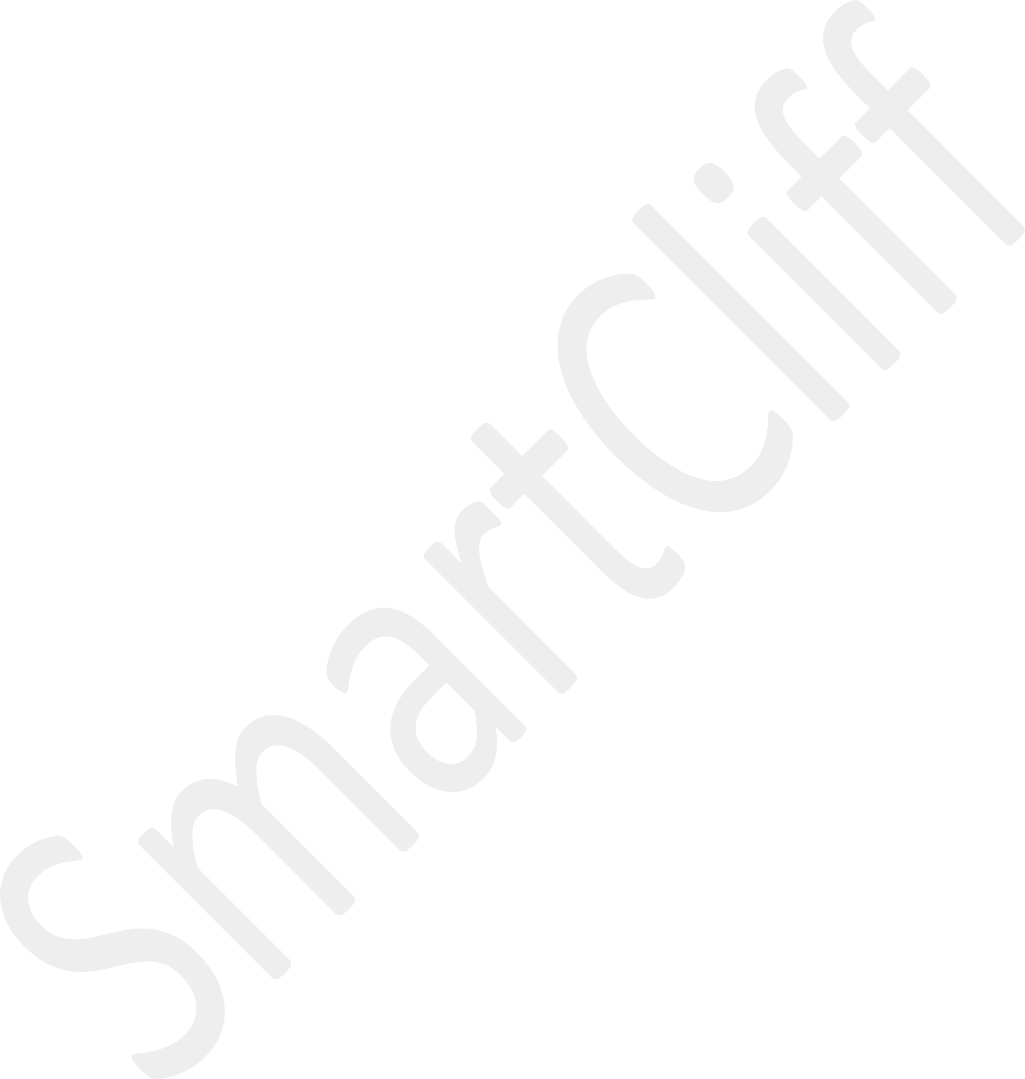
|  |  |  |
| --- | --- | --- |
|  | 1 1 1 2 5 -1  4  6 -3 -1  2  **Output 2**  1  6  **Constraints**  1 <= T <= 10  1 <= L <= 10^4  -10^9 <= data <= 10^9 1<= N <= L  data != -1  Where 'L' is the number of nodes in the linked list, ‘data’ represents the value of the nodes of the list.  Program:  package com.SelfPractice;  import java.util.Scanner;  class Node{  int data;  Node next;  Node prev;  Node(int data){  this.data = data;  this.next = null;  this.prev = null;  }  }  public class FindtheNthNodeFromTheEnd {  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list : ");  int n = sc.nextInt();  System.***out***.print("Enter the elements : ");  Node head = new Node(sc.nextInt());  int count = 0;  Node curr = head;  while (count < n-1) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  count++;  }  System.***out***.print("Enter the nth node : ");  int nth = sc.nextInt();  curr = head;  int size = 1;  while (curr.next != null) {  curr = curr.next;  size++;  }  int delpos = size - nth+1;  curr = head;  count = 1;  if(delpos==1)  head = head.next;  else {  Node prev = null;  while (curr.next != null && count < delpos) {  prev = curr;  curr = curr.next;  count++;  }  prev.next = curr.next;  }  System.***out***.print("After deletion : ");  curr = head;  while (curr != null) {  System.***out***.print(curr.data + " ");  curr = curr.next;  }  }  }  Output:  Enter the size of the list : 5  Enter the elements : 1 2 3 4 5  Enter the nth node : 1  After deletion : 1 2 3 4 |  |
| **2** | **Delete without head pointer**  **Problem statement :** You are given a node **del\_node** of a Singly Linked List where you have to **delete** a **value** of the given node from the linked list but you are not given the **head** of the list.  By deleting the node value, we do not mean removing it from memory. We mean:   * The value of the given node should not exist in the linked list. * The number of nodes in the linked list should decrease by one. * All the values before & after the **del\_node** node should be in the same order.   **Note:**   1. Multiple nodes can have the same **values** as the **del\_node**, but you must only remove the node whose pointer **del\_node** is given to you. 2. It is guaranteed that there exists a node with a value equal to the del\_node **value** and it will not be the last node of the linked list.   **Example 1: Input:**  Linked List = 1 -> 2  del\_node = 1  **Output:**  2 | Easy |



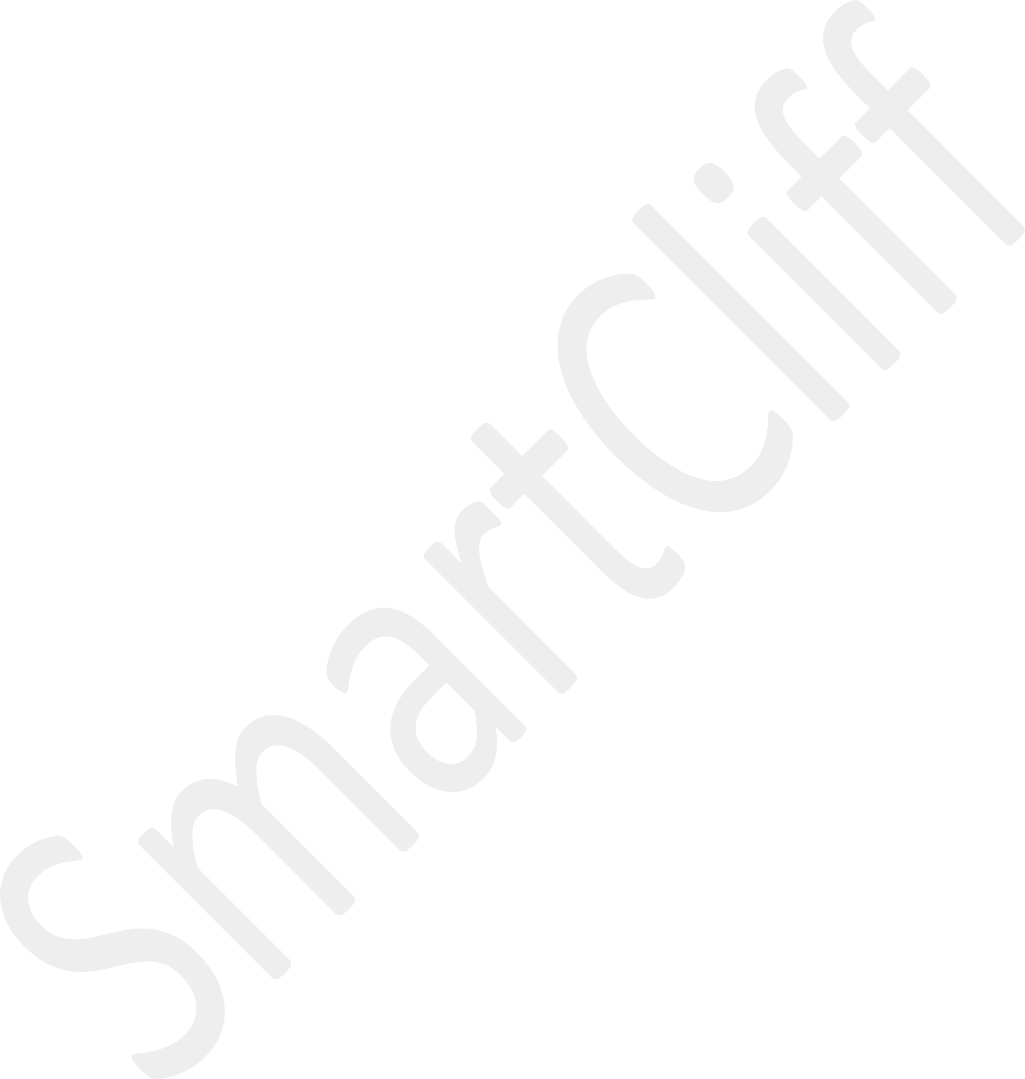
|  |  |  |
| --- | --- | --- |
|  | **Explanation:**  After deleting 1 from the linked list, we have remaining nodes as 2.  **Example 2:**  **Input:**  Linked List = 10 -> 20 -> 4 -> 30  del\_node = 20  **Output:**  10 4 30  **Explanation:**  After deleting 20 from the linked list, we have remaining nodes as 10, 4, 30.  **Constraints:**  2 <= n <= 103  1 <= elements of the linked list <= 109  Program:  package com.SelfPractice;  class ListNode {  int val;  ListNode next;  ListNode(int x) { val = x; }  }  public class DeleteNodeWithoutHeadPointer {  public static void deleteNode(ListNode del\_node) {  if (del\_node == null || del\_node.next == null) {  return; // As per the problem, this case will not happen  }  // Copy the value from the next node  del\_node.val = del\_node.next.val;  // Skip the next node  del\_node.next = del\_node.next.next;  }  public static void main(String[] args) {  // Create a sample linked list: 1 -> 2 -> 3 -> 4  ListNode head = new ListNode(1);  head.next = new ListNode(2);  head.next.next = new ListNode(3);  head.next.next.next = new ListNode(4);  // Node to be deleted (for example, node with value 3)  ListNode del\_node = head.next.next;  // Print the linked list before deletion  System.***out***.print("Linked list before deletion: ");  *printList*(head);  // Call the deleteNode function  *deleteNode*(del\_node);  // Print the linked list after deletion  System.***out***.print("Linked list after deletion: ");  *printList*(head);  }  // Helper function to print the linked list  public static void printList(ListNode head) {  ListNode current = head;  while (current != null) {  System.***out***.print(current.val + " ");  current = current.next;  }  System.***out***.println();  }  }  Output:  Linked list before deletion: 1 2 3 4  Linked list after deletion: 1 2 4 |  |
| **3** | **Sort Linked List of 0s, 1s, and 2s**  **Problem Statement:** Given a linked list of N nodes where nodes can contain values 0s, 1s, and 2s only. The task is to segregate 0s, 1s, and 2s linked list such that all zeros segregate to head side, 2s at the end of the linked list, and 1s in the mid of 0s and 2s.  **Example 1:**  Input:  N = 8  value[] = {1,2,2,1,2,0,2,2}  Output: 0 1 1 2 2 2 2 2  Explanation: All the 0s are segregated to the left end of the linked list,  2s to the right end of the list, and 1s in between.  **Example 2:**  Input:  N = 4  value[] = {2,2,0,1}  Output: 0 1 2 2 | Easy |



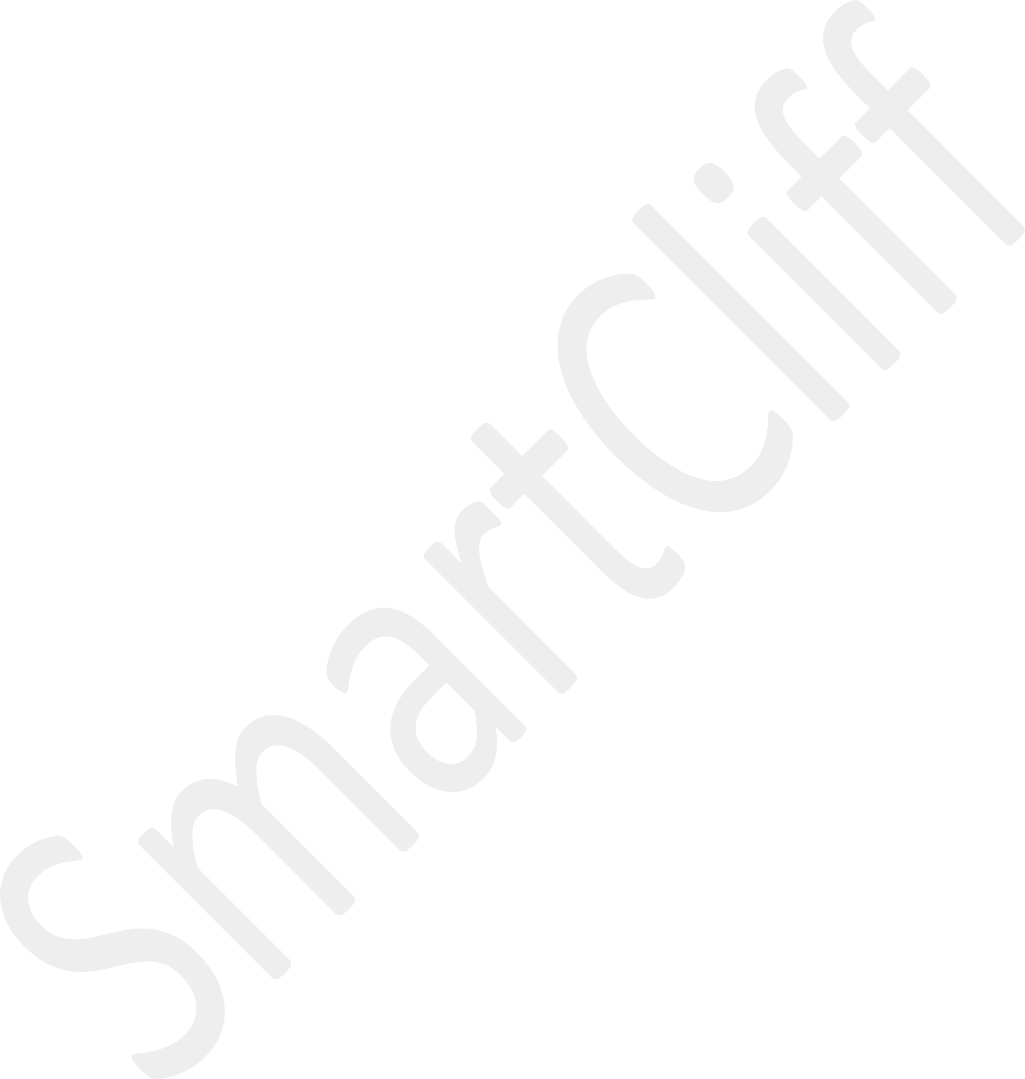
|  |  |  |
| --- | --- | --- |
|  | Explanation: After arranging all the 0s,1s and 2s in the given format, the output will be 0 1 2 2.  **Constraints**:  1 <= N <= 10^6  Program:  package com.SelfPractice;  import java.util.Scanner;  //class Node {  // int data;  // Node next;  // Node(int data) {  // this.data = data;  // this.next = null;  // }  //}  public class Sort0s1s2s {  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list : ");  int n = sc.nextInt();  System.***out***.print("Enter the elements : ");  Node head = new Node(sc.nextInt());  int count = 1;  Node curr = head;  while (count < n) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  count++;  }  Node zerohead = new Node(-1);  Node onehead = new Node(-1);  Node twohead = new Node(-1);  Node zero = zerohead;  Node one = onehead;  Node two = twohead;  curr = head;  while (curr != null) {  if(curr.data == 0) {  zero.next = curr;  zero = zero.next;  } else if (curr.data == 1) {  one.next = curr;  one = one.next;  } else if (curr.data == 2) {  two.next = curr;  two = two.next;  }  curr = curr.next;  }  zero.next = (onehead.next != null) ? onehead.next : twohead.next;  one.next = twohead.next;  two.next = null;  Node newHead = (zerohead.next != null) ? zerohead.next :  (onehead.next != null) ? onehead.next : twohead.next;  curr = newHead;  while (curr != null) {  System.***out***.print(curr.data + " ");  curr = curr.next;  }  }  }  Output:  Enter the size of the list : 5  Enter the elements : 1 1 2 0 0  0 0 1 1 2 |  |
| **4** | **Modular node**  **Problem statement :** Given a singly linked list and a number **k**, you are required to complete the function **modularNode()** which returns the modular node of the linked list.  A modular node is the last node of the linked list whose **Index** is divisible by the number **k**, i.e. **i%k==0**.  **Note:** If no such node is available, return **-1**. We are following 1 indexing.  **Example 1:**  Input**:** LinkedList: 1->2->3->4->5->6->7  k = 3  Output**:** 6  Explanation**:** Indexing of the linked list is same as the values of the Node. The maximum  index is which is divisible by 3 is 6 and the node value of the 6th index is 6.  **Example 2:**  Input**:** LinkedList: 19->28->37->46->55  k = 2  Output**:** 46  Explanation**:** Index start from 1 and end at 5.The maximum Index which is divisble by 2 is 4  and 4th Node in the list is 46.  **Constraints:**  1 <= T <= 100  1 <= N <= 500  Program:  package com.SelfPractice;  import java.util.Scanner;  public class ModularNode {  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list : ");  int n = sc.nextInt();  System.***out***.print("Enter the elements : ");  Node head = new Node(sc.nextInt());  int count = 0;  Node curr = head;  while (count < n-1) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  count++;  }  System.***out***.print("Enter the value of k : ");  int k = sc.nextInt();  curr = head;  int val = -1;  while (curr.next != null) {  if(curr.data%k == 0)  {  if(curr.data>val)  val = curr.data;  }  curr = curr.next;  }  System.***out***.println(val);  }  }  Output:  Enter the size of the list : 7  Enter the elements : 1 2 3 4 5 6 7  Enter the value of k : 3  6 | Easy |
| **5** | **Deletion in Circular Linked list Problem statement:**  You are given a Circular Linked List of integers, and an integer, 'key'.  You have to write a function that finds the given key in the list and deletes it. If no such key is present, then the list remains unchanged. | Easy |



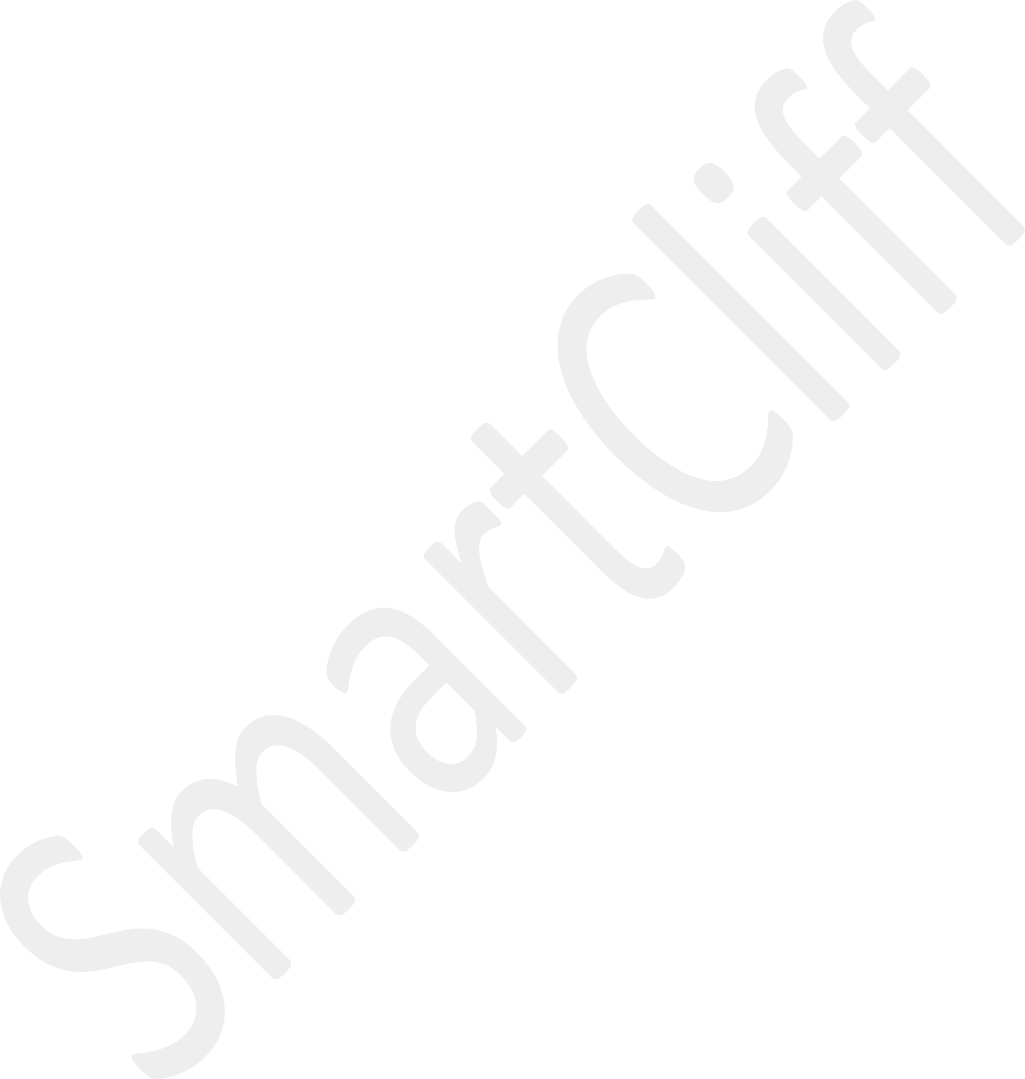
|  |  |  |
| --- | --- | --- |
|  | For Example :  This is a visualization of the Circular Linked List, represented by: 1 2 3 4 5 -1    Note :  The Circular Linked List before/after deletion may happen to be empty. In that case, only print -1.All integers in the list are unique.  **Example 1:**  Sample Input 1 :  1 2 3 4 5 -1  3  Sample Output 1 :  1 2 4 5 -1  Explanation for Sample 1 :  The given linked list, before deletion:    After deletion :    **Example 2:** Sample Input 2 : 1 2 3 4 5 -1  1  Sample Output 2 : |  |



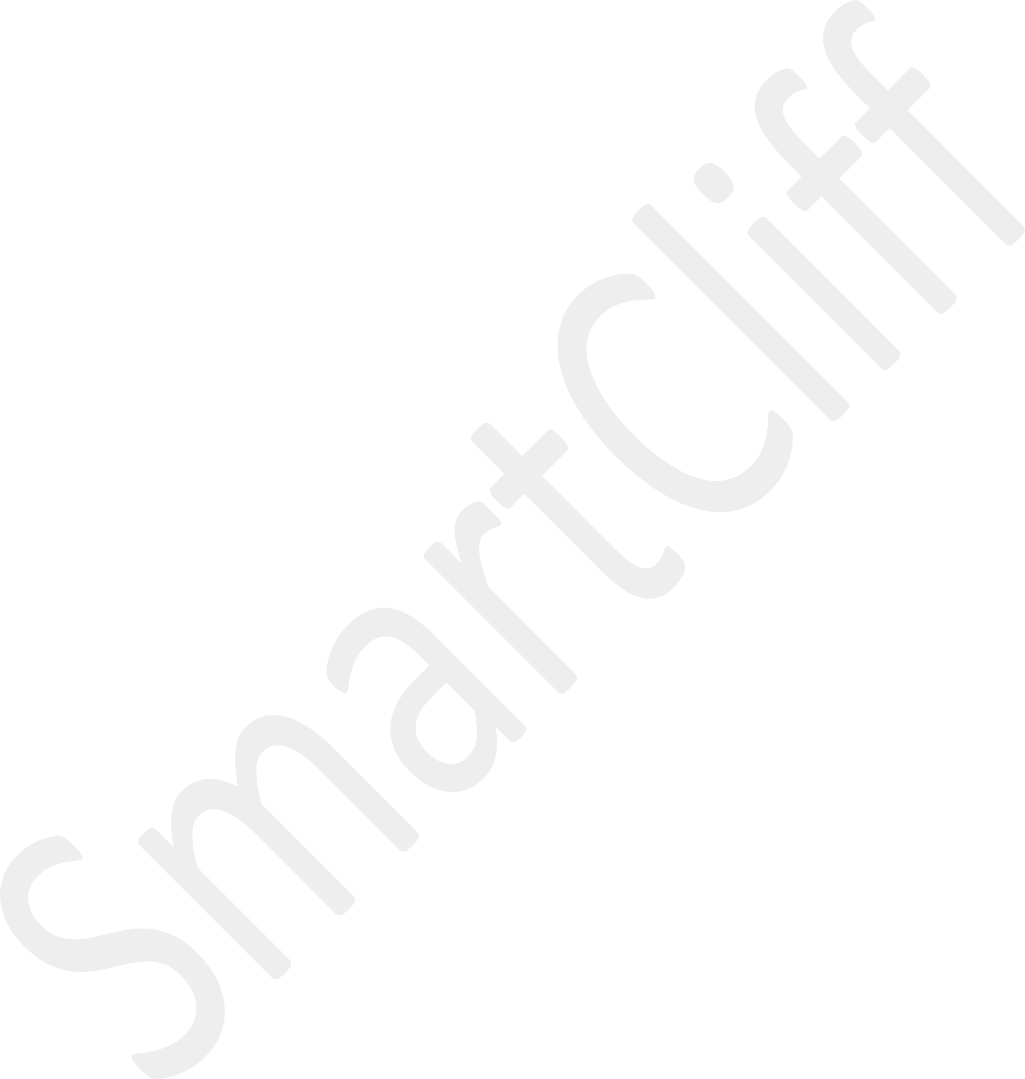
|  |  |  |
| --- | --- | --- |
|  | 2 3 4 5 -1  **Constraints :**  0 <= N <= 10 ^ 5  1 <= key <= 10 ^ 5  Where 'N' is the length of the Circular Linked List.  Program:  package com.SelfPractice;  import java.util.Scanner;  //class Node {  // int data;  // Node next;  //  // Node(int data) {  // this.data = data;  // next = null;  // }  //}  public class DeletioninCircularLinkedList {  static Node *head*;  static int *n* = 0;  static void insertAtBeg(int data) {  Node newNode = new Node(data);  if (*head* == null) {  *head* = newNode;  newNode.next = *head*;  } else {  Node curr = *head*;  while (curr.next != *head*) {  curr = curr.next;  }  curr.next = newNode;  newNode.next = *head*;  *head* = newNode;  }  *n*++;  }  static void insertAtEnd(int data) {  Node newNode = new Node(data);  if (*head* == null) {  *head* = newNode;  newNode.next = *head*;  } else {  Node curr = *head*;  while (curr.next != *head*) {  curr = curr.next;  }  curr.next = newNode;  newNode.next = *head*;  }  *n*++;  }  static void insertAtAny(int data, int pos) {  Node newNode = new Node(data);  if (pos == 1) {  *insertAtBeg*(data);  } else if (pos == *n* + 1) {  *insertAtEnd*(data);  } else {  Node curr = *head*;  int count = 1;  while (count < pos - 1 && curr.next != *head*) {  curr = curr.next;  count++;  }  newNode.next = curr.next;  curr.next = newNode;  *n*++;  }  }  static void deleteAtBeg() {  Node curr = *head*;  while (curr.next != *head*) {  curr = curr.next;  }  *head* = *head*.next;  curr.next = *head*;  *n*--;  }  static void deleteAtEnd() {  Node curr = *head*;  Node prev = null;  if (*head* == null)  System.***out***.println("Empty list ");  else {  while (curr.next != *head*) {  prev = curr;  curr = curr.next;  }  }  prev.next = *head*;  curr.next = null;  *n*--;  }  static void deleteAtAny(int pos) {  if (*head* == null)  System.***out***.println("List is empty");  else if (pos == 1) {  *deleteAtBeg*();  } else if (pos == *n*) {  *deleteAtEnd*();  } else {  Node curr = *head*;  int count = 1;  while (count < pos - 1 && curr.next != *head*) {  curr = curr.next;  count++;  }  curr.next = curr.next.next;  *n*--;  }  }  static void display() {  if (*head* == null) {  System.***out***.println("List is empty");  } else {  Node curr = *head*;  do {  System.***out***.print(curr.data + " ");  curr = curr.next;  } while (curr != *head*);  System.***out***.println();  }  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.***in***);  int choice;  do {  System.***out***.println("\nCircular Linked List Operations:");  System.***out***.println("1. Insert at beginning");  System.***out***.println("2. Insert at end");  System.***out***.println("3. Insert at any position");  System.***out***.println("4. Delete from beginning");  System.***out***.println("5. Delete from end");  System.***out***.println("6. Delete from any position");  System.***out***.println("7. Display");  System.***out***.println("8. Exit");  System.***out***.print("Enter your choice (1-8): ");  choice = scanner.nextInt();  switch (choice) {  case 1:  System.***out***.print("Enter data to insert: ");  int data = scanner.nextInt();  *insertAtBeg*(data);  break;  case 2:  System.***out***.print("Enter data to insert: ");  data = scanner.nextInt();  *insertAtEnd*(data);  break;  case 3:  System.***out***.print("Enter data to insert: ");  data = scanner.nextInt();  System.***out***.print("Enter position: ");  int pos = scanner.nextInt();  *insertAtAny*(data, pos);  break;  case 4:  *deleteAtBeg*();  break;  case 5:  *deleteAtEnd*();  break;  case 6:  System.***out***.print("Enter position: ");  pos = scanner.nextInt();  *deleteAtAny*(pos);  break;  case 7:  *display*();  break;  case 8:  System.***out***.println("Exiting...");  break;  default:  System.***out***.println("Invalid choice. Please try again.");  }  } while (choice != 8);  scanner.close();  }  }  Output:  Circular Linked List Operations:  1. Insert at beginning  2. Insert at end  3. Insert at any position  4. Delete from beginning  5. Delete from end  6. Delete from any position  7. Display  8. Exit  Enter your choice (1-8): 1  Enter data to insert: 1  Circular Linked List Operations:  1. Insert at beginning  2. Insert at end  3. Insert at any position  4. Delete from beginning  5. Delete from end  6. Delete from any position  7. Display  8. Exit  Enter your choice (1-8): 1  Enter data to insert: 5  Circular Linked List Operations:  1. Insert at beginning  2. Insert at end  3. Insert at any position  4. Delete from beginning  5. Delete from end  6. Delete from any position  7. Display  8. Exit  Enter your choice (1-8): 1  Enter data to insert: 5  Circular Linked List Operations:  1. Insert at beginning  2. Insert at end  3. Insert at any position  4. Delete from beginning  5. Delete from end  6. Delete from any position  7. Display  8. Exit  Enter your choice (1-8): 34  Invalid choice. Please try again.  Circular Linked List Operations:  1. Insert at beginning  2. Insert at end  3. Insert at any position  4. Delete from beginning  5. Delete from end  6. Delete from any position  7. Display  8. Exit  Enter your choice (1-8): 1  Enter data to insert: 243  Circular Linked List Operations:  1. Insert at beginning  2. Insert at end  3. Insert at any position  4. Delete from beginning  5. Delete from end  6. Delete from any position  7. Display  8. Exit  Enter your choice (1-8): 1  Enter data to insert: 32  Circular Linked List Operations:  1. Insert at beginning  2. Insert at end  3. Insert at any position  4. Delete from beginning  5. Delete from end  6. Delete from any position  7. Display  8. Exit  Enter your choice (1-8): 1  Enter data to insert: 34  Circular Linked List Operations:  1. Insert at beginning  2. Insert at end  3. Insert at any position  4. Delete from beginning  5. Delete from end  6. Delete from any position  7. Display  8. Exit  Enter your choice (1-8): 7  34 32 243 5 5 1  Circular Linked List Operations:  1. Insert at beginning  2. Insert at end  3. Insert at any position  4. Delete from beginning  5. Delete from end  6. Delete from any position  7. Display  8. Exit  Enter your choice (1-8): 5  Circular Linked List Operations:  1. Insert at beginning  2. Insert at end  3. Insert at any position  4. Delete from beginning  5. Delete from end  6. Delete from any position  7. Display  8. Exit  Enter your choice (1-8): 7  34 32 243 5 5  Circular Linked List Operations:  1. Insert at beginning  2. Insert at end  3. Insert at any position  4. Delete from beginning  5. Delete from end  6. Delete from any position  7. Display  8. Exit  Enter your choice (1-8): |  |
| **6** | **Reverse a Doubly Linked List Problem Statement**  Given a doubly linked list of n elements. Your task is to reverse the doubly linked list in-place.  **Example 1:**  Input:  LinkedList: 3 <--> 4 <--> 5  Output: 5 4 3  Example 2:  Input:  LinkedList: 75 <--> 122 <--> 59 <--> 196  Output: 196 59 122 75  **Constraints**:  1 <= number of nodes <= 10^4 0 <= value of nodes <= 10^4  Program:  package com.SelfPractice;  import java.util.Scanner;  public class ReverseDLL {  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list : ");  int n = sc.nextInt();  System.***out***.print("Enter the elements : ");  Node head = new Node(sc.nextInt());  int count = 0;  Node curr = head;  while (count < n-1) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  count++;  }  Node prev = null;  curr = head;  while (curr != null) {  Node next = curr.next;  curr.next = prev;  prev = curr;  curr = next;  }  curr = prev;  while (curr != null) {  System.***out***.print(curr.data + " ");  curr = curr.next;  }  }  }  Output:  Enter the size of the list : 5  Enter the elements : 1 2 3 4 5  5 4 3 2 1 | Easy |
| **7** | **Insert in Sorted way in a Sorted DLL Problem Statement**  Given a sorted doubly linked list and an element X, your need to insert the element X into correct position in the sorted DLL.  Note: The DLL is sorted in ascending order  **Example:**  Input:  LinkedList: | Easy |



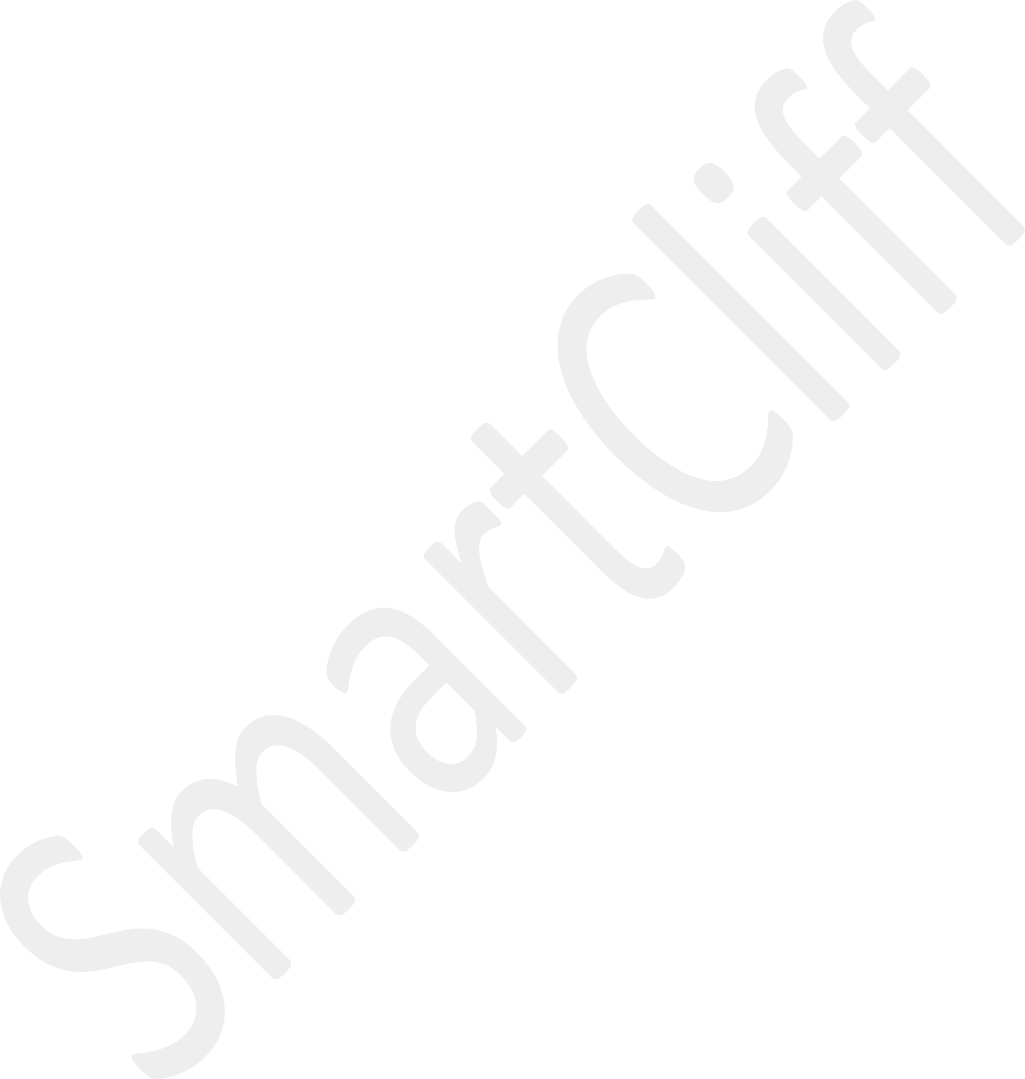
|  |  |  |
| --- | --- | --- |
|  | X = 9  Output:    **Constraints**:  1 <= Number of nodes <= 10^3  Program:  package com.SelfPractice;  import java.util.Scanner;  public class InsertSortedDLL {  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list : ");  int n = sc.nextInt();  System.***out***.print("Enter the elements : ");  Node head = new Node(sc.nextInt());  Node curr = head;  for (int i = 1; i < n; i++) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  }  System.***out***.print("Enter the element to insert : ");  int ele = sc.nextInt();  Node inserted = new Node(ele);  curr = head;  Node prev = null;  if (head.data >= ele) {  inserted.next = head;  head = inserted;  } else {  while (curr != null && curr.data < ele) {  prev = curr;  curr = curr.next;  }  prev.next = inserted;  inserted.next = curr;  }  curr = head;  while (curr != null) {  System.***out***.print(curr.data + " ");  curr = curr.next;  }  }  }  Output:  Enter the size of the list : 3  Enter the elements : 10 93 100  Enter the element to insert : 9  9 10 93 100 |  |
| **8** | **Add one to a number represented as Linked List**  **Problem statement :**You're given a positive integer represented in the form of a singly linked-list of digits. The length of the number is **'n'.**Add 1 to the number, i.e., increment the given number by one.  The digits are stored such that the most significant digit is at the head of the linked list and the least significant digit is at the tail of the linked list.  **Input**: Initial Linked List: 1 -> 5 -> 2  **Output**: Modified Linked List: 1 -> 5 -> 3  **Explanation**:  Initially the number is 152. After incrementing it by 1, the number becomes 153  **Constraints:**   * 1 <= 'n' <= 10^5 * 0 <= Node in linked list <= 9 There are no leading zeroes in the number.   Program:  package com.SelfPractice;  import java.util.Scanner;  public class Addoneincrement {  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list : ");  int n = sc.nextInt();  System.***out***.print("Enter the elements : ");  Node head = new Node(sc.nextInt());  int count = 0;  Node curr = head;  while (count < n-1) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  count++;  }  curr = head;  Node prev = null;  while (curr.next != null)  {  prev = curr;  curr = curr.next;  }  if(curr.data<8)  curr.data += 1;  else  {  prev.data += 1;  curr.data = 0;  }  curr = head;  System.***out***.print("Modified list : ");  while (curr != null)  {  System.***out***.print(curr.data + " ");  curr = curr.next;  }  }  }  Output:  Enter the size of the list : 3  Enter the elements : 1 2 3  Modified list : 1 2 4 | Medium |
| **9** | **Make maximum number**  **Problem statement :** Given a linked list such that each node represents a digit. Construct the maximum number possible from the given digits.  You just need to print the maximum Integer that can be formed  **Input 1 :**  1 2 2 0 9 -1  **Output 1 :**  92210  **Explanation For Input1:** | Medium |



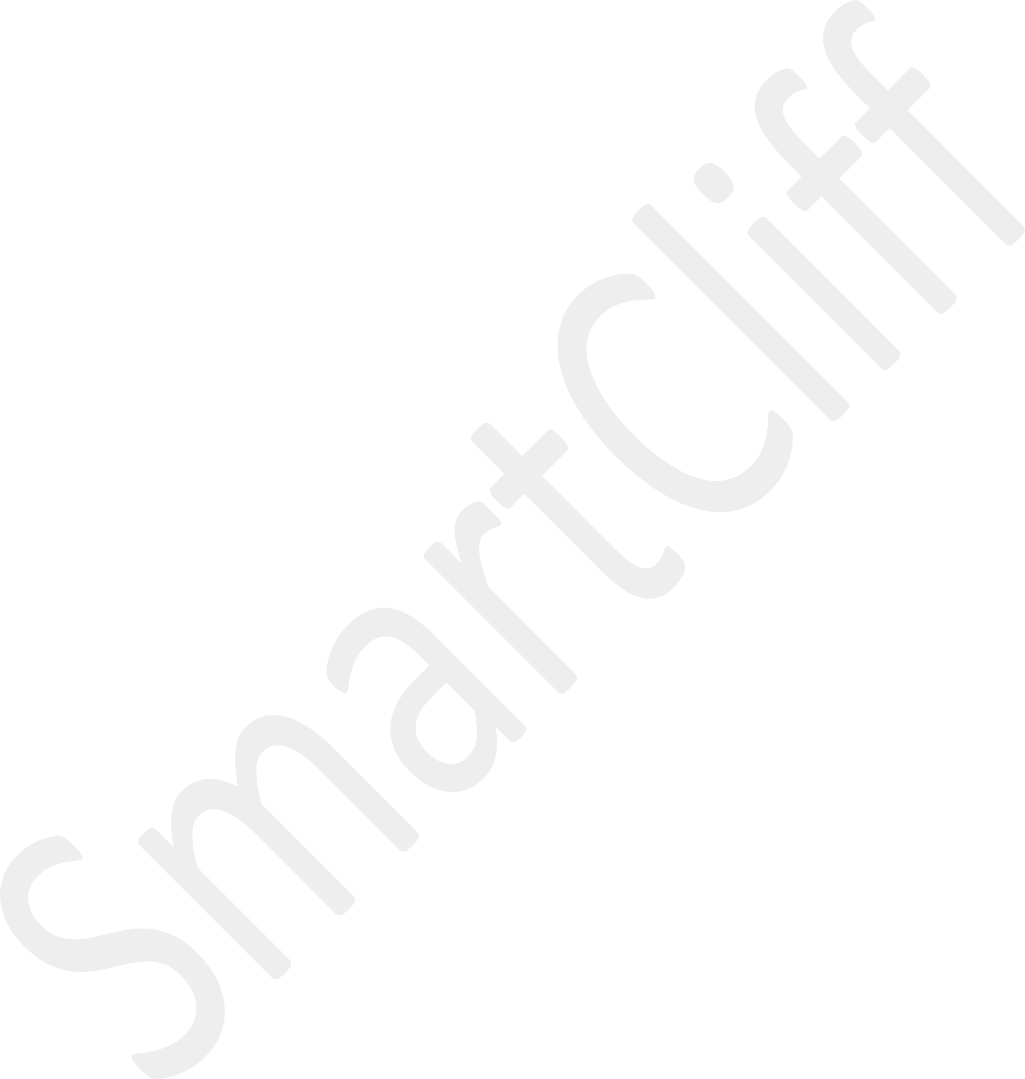
|  |  |  |
| --- | --- | --- |
|  | The digits that are present in the linked list are : (1, 2, 2, 0, 9 )  So the answer is the maximum of all the numbers that are formed using these digits is 92210.  **Input 2 :**  1 0 0 0 3 -1  **Output 2 :**  31000  **Constraints:**  1 <= N <= 10^6  0 <= data <= 9  Program:  package com.SelfPractice;  import java.util.Scanner;  public class MakeaNumberMax {  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list : ");  int n = sc.nextInt();  System.***out***.print("Enter the elements : ");  Node head = new Node(sc.nextInt());  int count = 0;  Node curr = head;  while (count < n-1) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  count++;  }  curr = head;  for(int i = 0; i <= n; i++) {  curr = head;  while (curr.next != null) {  if (curr.data < curr.next.data) {  int temp = curr.data;  curr.data = curr.next.data;  curr.next.data = temp;  }  curr = curr.next;  }  }  curr = head;  int max = 0;  while (curr != null) {  max = (max\*10) + curr.data;  curr = curr.next;  }  System.***out***.println(max);  }  }  Output:  Enter the size of the list : 5  Enter the elements : 1 0 0 0 3  31000 |  |
| **10** | **Next greater node :** 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 :**    **Input:** head = [2,1,5]  **Output:** [5,5,0]  **Example 2:**    **Input:** head = [2,7,4,3,5]  **Output:** [7,0,5,5,0]  **Constraints:**   * The number of nodes in the list is n. | Medium |



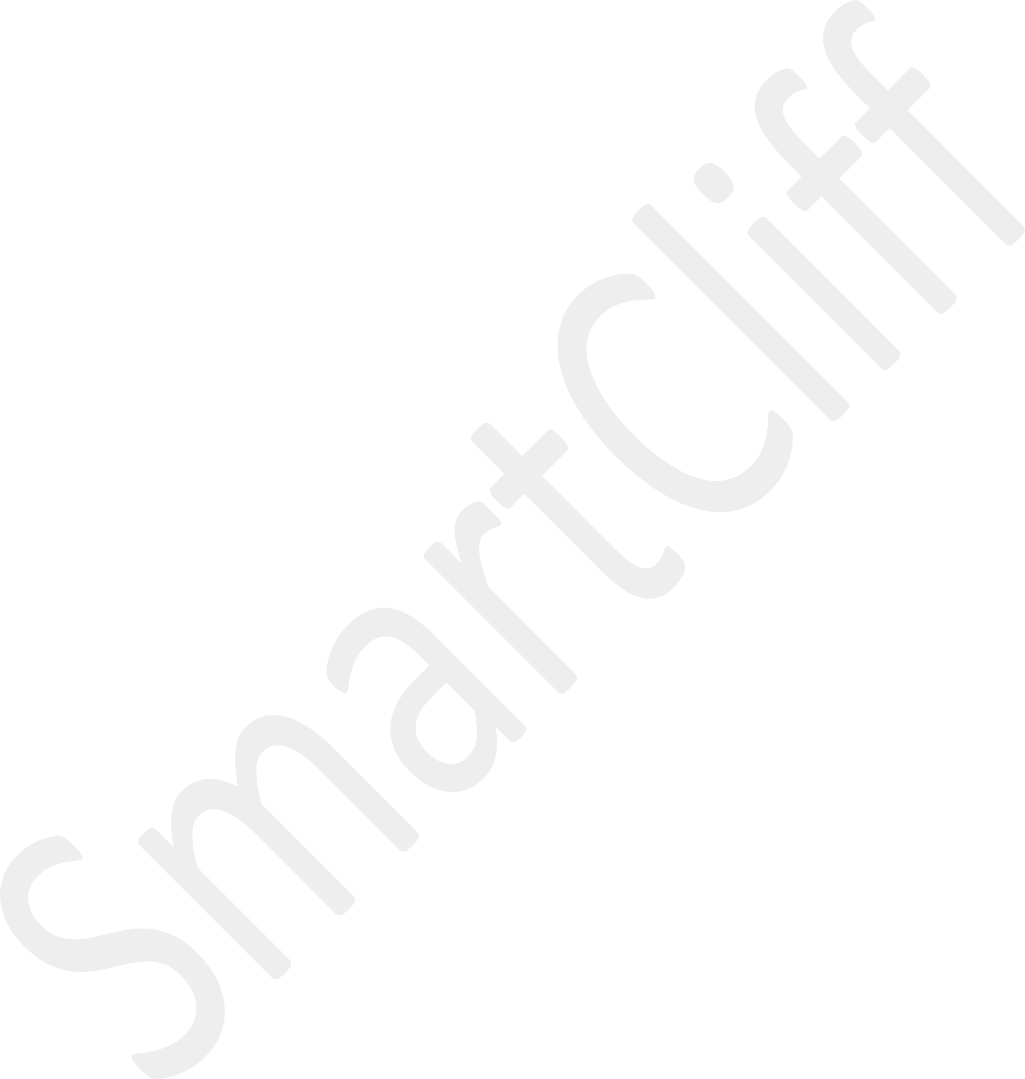
|  |  |  |
| --- | --- | --- |
|  | * 1 <= n <= 104 * 1 <= Node.val <= 109   Program:  package com.SelfPractice;  import java.util.Scanner;  public class NextGreater {  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list : ");  int n = sc.nextInt();  System.***out***.print("Enter the elements : ");  Node head = new Node(sc.nextInt());  Node curr = head;  for (int i = 1; i < n; i++) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  }  curr = head;  while (curr != null) {  int nextGreater = 0;  Node temp = curr.next;  while (temp != null) {  if (temp.data > curr.data) {  nextGreater = temp.data;  break;  }  temp = temp.next;  }  System.***out***.print(nextGreater + " ");  curr = curr.next;  }  }  }  Output:  Enter the size of the list : 5  Enter the elements : 1 3 7 4 9  3 7 9 9 0 |  |
| **11** | **Linked List Components**  You are given the head of a linked list containing unique integer values and an integer array nums that is a subset of the linked list values.  Return the number of connected components in nums where two values are connected if they appear consecutively in the linked list.  **Example 1:**    Input: head = [0,1,2,3], nums = [0,1,3] Output: 2  Explanation: 0 and 1 are connected, so [0, 1] and [3] are the two connected components.  **Example 2:**    Input: head = [0,1,2,3,4], nums = [0,3,1,4] Output: 2  Explanation: 0 and 1 are connected, 3 and 4 are connected, so [0, 1] and [3, 4] are the two connected components.  **Constraints:**   * The number of nodes in the linked list is n. * 1 <= n <= 10^4 * 0 <= Node.val < n * All the values Node.val are unique. * 1 <= nums.length <= n * 0 <= nums[i] < n * All the values of nums are unique.   Program:  package com.SelfPractice;  import java.util.HashSet;  import java.util.Scanner;  public class LinkedListComponents {  static int numComponents(Node head, int[] nums) {  HashSet<Integer> set = new HashSet<>();  for (int num : nums) {  set.add(num);  }  int count = 0;  boolean inComponent = false;  Node curr = head;  while (curr != null) {  if (set.contains(curr.data)) {  if (!inComponent) {  count++;  inComponent = true;  }  } else {  inComponent = false;  }  curr = curr.next;  }  return count;  }  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list: ");  int n = sc.nextInt();  System.***out***.print("Enter the list elements: ");  Node head = new Node(sc.nextInt());  Node curr = head;  for (int i = 1; i < n; i++) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  }  System.***out***.print("Enter the size of the array: ");  n = sc.nextInt();  int[] arr = new int[n];  for (int i = 0; i < n; i++) {  arr[i] = sc.nextInt();  }  int result = *numComponents*(head, arr);  System.***out***.println(result);  }  Output:  Enter the size of the list: 5  Enter the list elements: 0 1 2 3 4  Enter the size of the array: 4  0 3 1 4  2 | Medium |



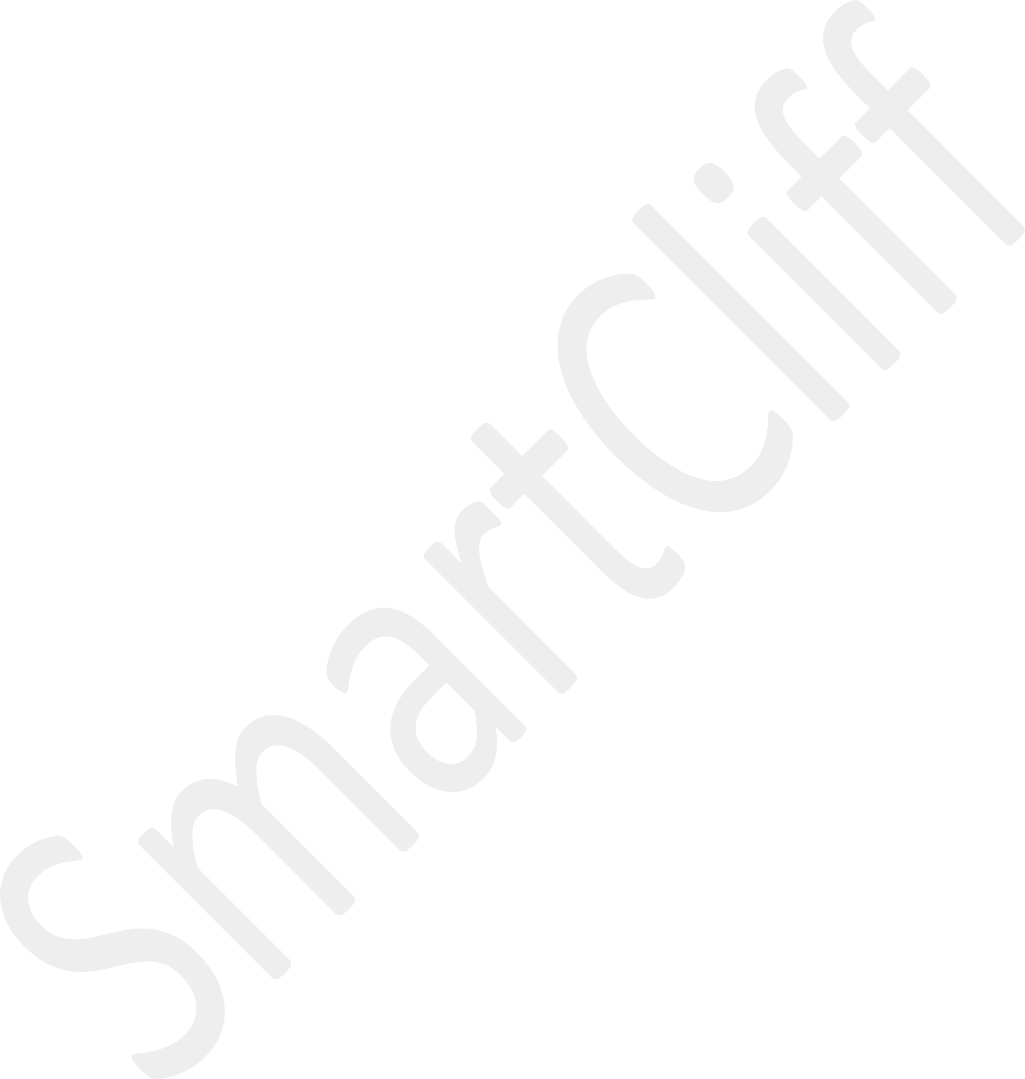
|  |  |  |
| --- | --- | --- |
| **12** | **Multiply Linked Lists** | Medium |
|  | **Problem statement** |  |
|  | Given two numbers represented by linked lists. Your task is to find the multiplied |  |
|  | list and return the head of the multiplied list. |  |
|  | The multiplied list is a linked list representation of the multiplication of two |  |
|  | numbers. |  |
|  | **Input 1:** |  |
|  | 1 |  |
|  | 5 6 3 -1 |  |
|  | 8 4 2 -1 |  |
|  | **Output 1:** |  |
|  | 4 7 4 0 4 6 -1 |  |
|  | **Explanation of Input 1:** |  |
|  | 563 \* 842 = 474046 |  |
|  | **Input 2:** |  |
|  | 2 |  |
|  | 7 5 9 4 6 -1 |  |
|  | 0 -1 |  |
|  | 0 2 3 4 0 - 1 |  |
|  | 0 0 1 -1 |  |
|  | **Output 2:** |  |
|  | 0 -1 |  |
|  | 2 3 4 0 -1 |  |
|  | **Explanation of Input 2:** |  |
|  | 75946 \* 0 = 0 |  |
|  | 02340 \* 001 = 2340 |  |
|  | **Constraints :** |  |
|  | 1 <= T <= 10 |  |
|  | 1 <= N, M <= 100 |  |
|  | 0 <= data <= 9 |  |
|  | Where N and M are the number of nodes in the two linked lists.  Program:  package com.SelfPractice;  import java.util.Scanner;  public class MultiplyLL {  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list1 : ");  int n = sc.nextInt();  System.***out***.print("Enter the list1 elements : ");  Node head = new Node(sc.nextInt());  int count = 0;  Node curr = head;  while (count < n-1) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  count++;  }  System.***out***.print("Enter the size of list2 : ");  int n2 = sc.nextInt();  System.***out***.print("Enter the list2 elements : ");  Node head2 = new Node(sc.nextInt());  int count2 = 0;  Node curr2 = head2;  while (count2 < n2-1) {  curr2.next = new Node(sc.nextInt());  curr2 = curr2.next;  count2++;  }  int a = head.data;  int b = head2.data;  curr = head.next;  while (curr != null) {  a = (a\*10)+curr.data;  curr = curr.next;  }  curr2 = head2.next;  while (curr2 != null) {  b = (b\*10)+curr2.data;  curr2 = curr2.next;  }  int val = a\*b;  int temp = val;  int rev = 0;  while (temp != 0) {  int rem = temp % 10;  rev = rev\*10+rem;  temp = temp/10;  }  val = rev;  Node head3 = new Node(val%10);  curr = head3;  val /= 10;  while (val != 0) {  curr.next = new Node(val%10);  curr = curr.next;  val /= 10;  }  System.***out***.print("Multiplication of the two lists is : ");  curr = head3;  while (curr != null) {  System.***out***.print(curr.data+" ");  curr = curr.next;  }  }  }  Output:  Enter the size of the list1 : 3  Enter the list1 elements : 5 3 5  Enter the size of list2 : 3  Enter the list2 elements : 8 4 2  Multiplication of the two lists is : 4 5 0 4 7 |  |
| **13** | **Remove Nodes From Linked List**  You are given the head of a linked list. | Medium |



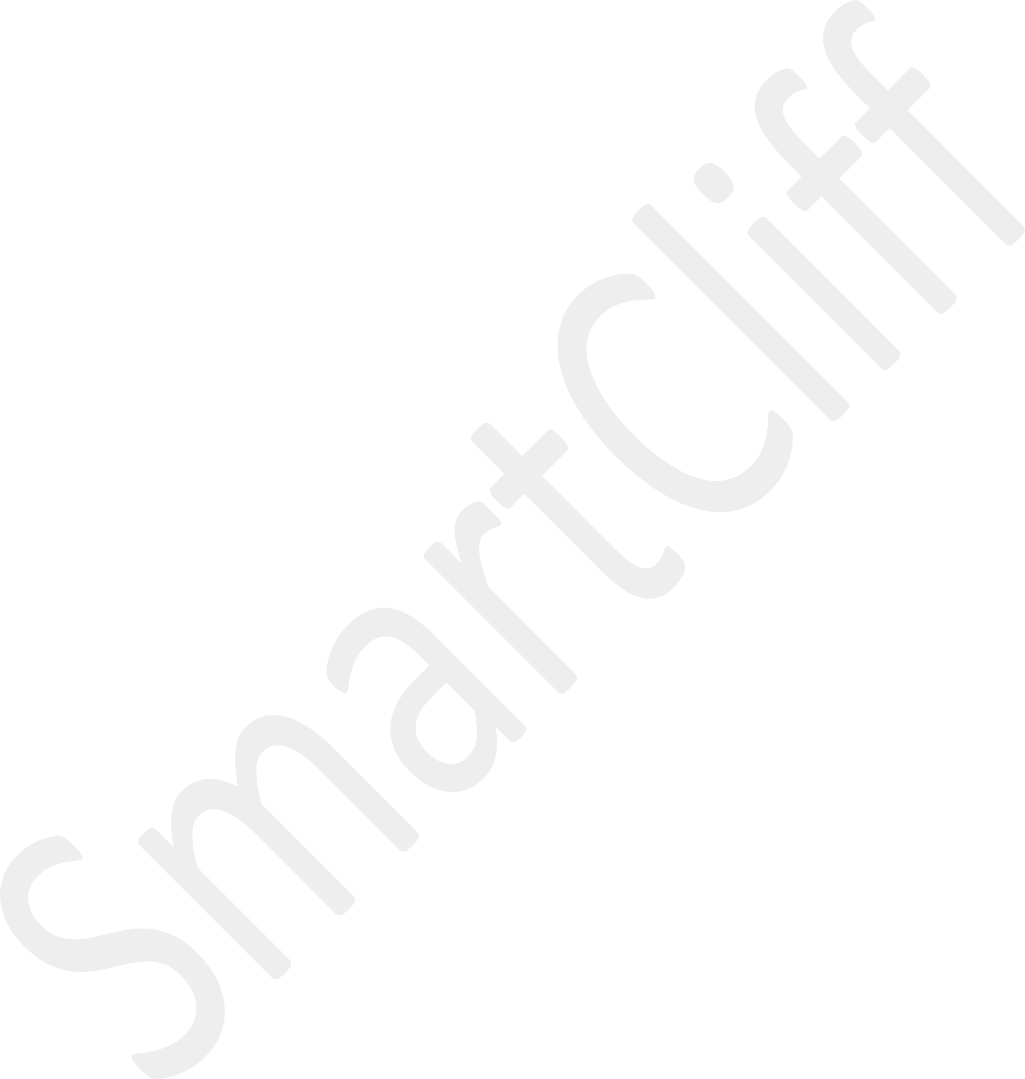
|  |  |  |
| --- | --- | --- |
|  | Remove every node which has a node with a greater value anywhere to the right side of it.  Return *the* head *of the modified linked list.*  **Example 1:**    Input: head = [5,2,13,3,8] Output: [13,8]  Explanation: The nodes that should be removed are 5, 2 and 3.   * Node 13 is to the right of node 5. * Node 13 is to the right of node 2. * Node 8 is to the right of node 3.   **Example 2:**  Input: head = [1,1,1,1] Output: [1,1,1,1]  Explanation: Every node has value 1, so no nodes are removed.  **Constraints**:   * The number of the nodes in the given list is in the range [1, 105]. * 1 <= Node.val <= 105   Program:  package com.SelfPractice;  import java.util.ArrayList;  import java.util.Collections;  import java.util.Scanner;  public class RemoveNodesFromLL {  static public Node removeNodes(Node head) {  ArrayList<Integer> ls = new ArrayList<>();  ArrayList<Integer> res = new ArrayList<>();  Node temp = head;  while(temp != null)  {  ls.add(temp.data);  temp = temp.next;  }  int n = ls.size();  int max = ls.get(n-1);  res.add(max);  for(int i=n-2;i>=0;i--)  {  int val = ls.get(i);  if(val>=max)  {  res.add(val);  max = val;  }  }  Collections.*reverse*(res);  Node new\_head = new Node(res.get(0));  Node temp2 = new\_head;  for(int i=1;i<res.size();i++)  {  temp2.next = new Node(res.get(i));  temp2 = temp2.next;  }  return new\_head;  }  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list1 : ");  int n = sc.nextInt();  System.***out***.print("Enter the list1 elements : ");  Node head = new Node(sc.nextInt());  int count = 0;  Node curr = head;  while (count < n-1) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  count++;  }  curr = *removeNodes*(head);  System.***out***.print("After deletion : ");  while (curr != null) {  System.***out***.print(curr.data + " ");  curr = curr.next;  }  }  }  Output:  Enter the size of the list1 : 5  Enter the list1 elements : 5 2 13 3 8  After deletion : 13 8 |  |
| **14** | **Delete the middle node**  **Problem Statement:** You are given the head of a linked list. Delete the middle node, and return *the* head *of the modified linked list*.  The middle node of a linked list of size n is the ⌊n / 2⌋th node from the start using 0- based indexing, where ⌊x⌋ denotes the largest integer less than or equal to x.   * For n = 1, 2, 3, 4, and 5, the middle nodes are 0, 1, 1, 2, and 2, respectively.   **Note**: K will always be between 1 and N, where N is the length of the LL.  Example 1:    **Input**: head = [1,3,4,7,1,2,6]  **Output**: [1,3,4,1,2,6]  **Explanation**: | Medium |



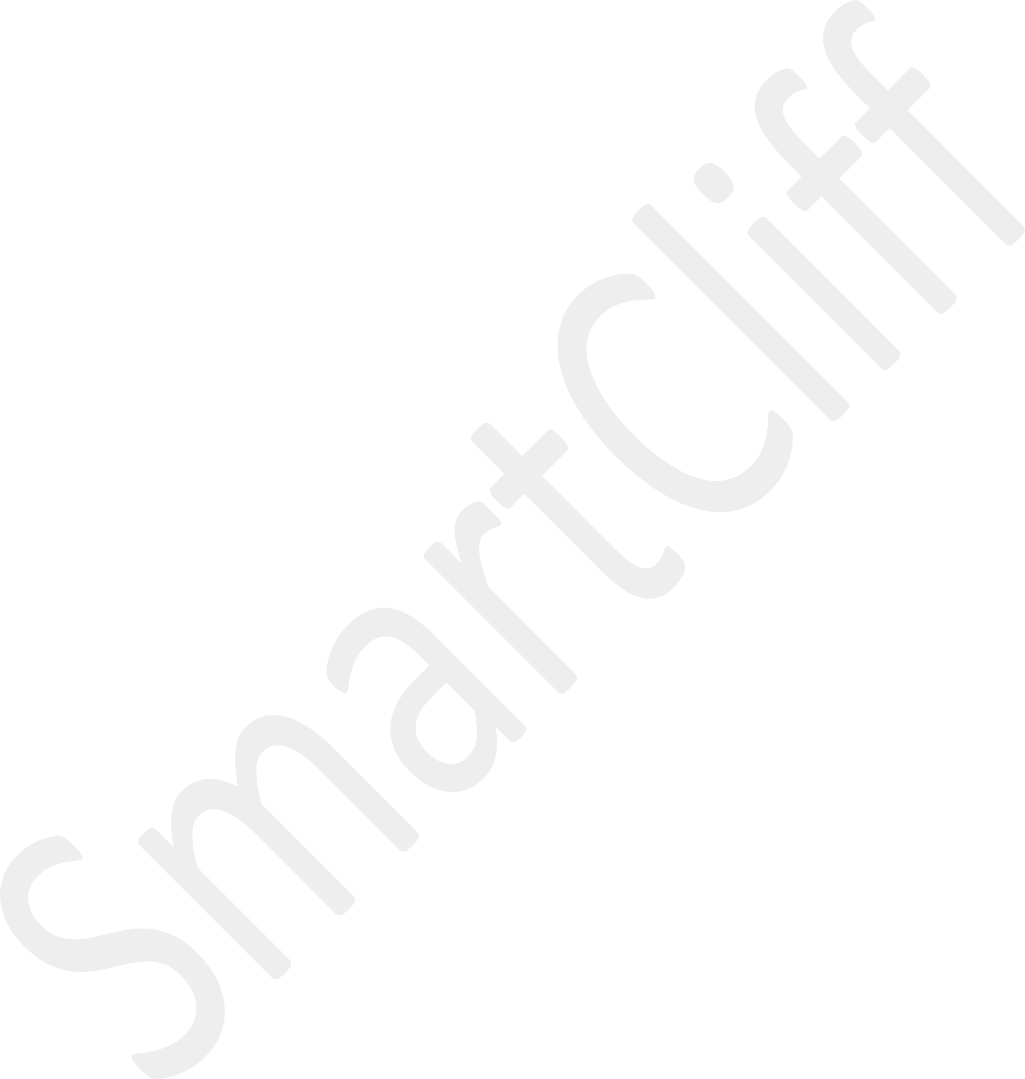
|  |  |  |
| --- | --- | --- |
|  | The above figure represents the given linked list. The indices of the nodes are written below.  Since n = 7, node 3 with value 7 is the middle node, which is marked in red. We return the new list after removing this node.  **Example 2:**  **Input**: head = [1,2,3,4] **Output**: [1,2,4] **Explanation**:  The above figure represents the given linked list.  For n = 4, node 2 with value 3 is the middle node, which is marked in red.  **Constraints :**   * The number of nodes in the list is in the range [1, 105]. * 1 <= Node.val <= 105   Program:  package com.SelfPractice;  import java.util.Scanner;  public class DeleteMiddleNode {  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list1 : ");  int n = sc.nextInt();  System.***out***.print("Enter the list1 elements : ");  Node head = new Node(sc.nextInt());  int count = 0;  Node curr = head;  while (count < n-1) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  count++;  }  Node fast = head;  Node slow = head;  while (fast.next != null && fast.next.next != null) {  fast = fast.next.next;  slow = slow.next;  }  curr = head;  while (curr.next != slow && curr.next != null) {  curr = curr.next;  }  curr.next = slow.next;  curr = head;  System.***out***.print("After deleting middle : ");  while(curr != null)  {  System.***out***.print(curr.data + " ");  curr = curr.next;  }  }  }  Output:  Enter the size of the list1 : 5  Enter the list1 elements : 1 2 3 4 5  After deleting middle : 1 2 4 5 |  |
| **15** | **Delete N nodes after M nodes :** You are given the head of a linked list and two integers m and n.  Traverse the linked list and remove some nodes in the following way:   * Start with the head as the current node. * Keep the first m nodes starting with the current node. * Remove the next n nodes * Keep repeating steps 2 and 3 until you reach the end of the list. Return the head of the modified list after removing the mentioned nodes. **Example 1:**     Input: head = [1,2,3,4,5,6,7,8,9,10,11,12,13], m = 2, n = 3 Output: [1,2,6,7,11,12]  Explanation: Keep the first (m = 2) nodes starting from the head of the linked List (1 ->2) show in black nodes.  Delete the next (n = 3) nodes (3 -> 4 -> 5) show in read nodes.  Continue with the same procedure until reaching the tail of the Linked List. Head of the linked list after removing nodes is returned.  **Example 2:** | Medium |



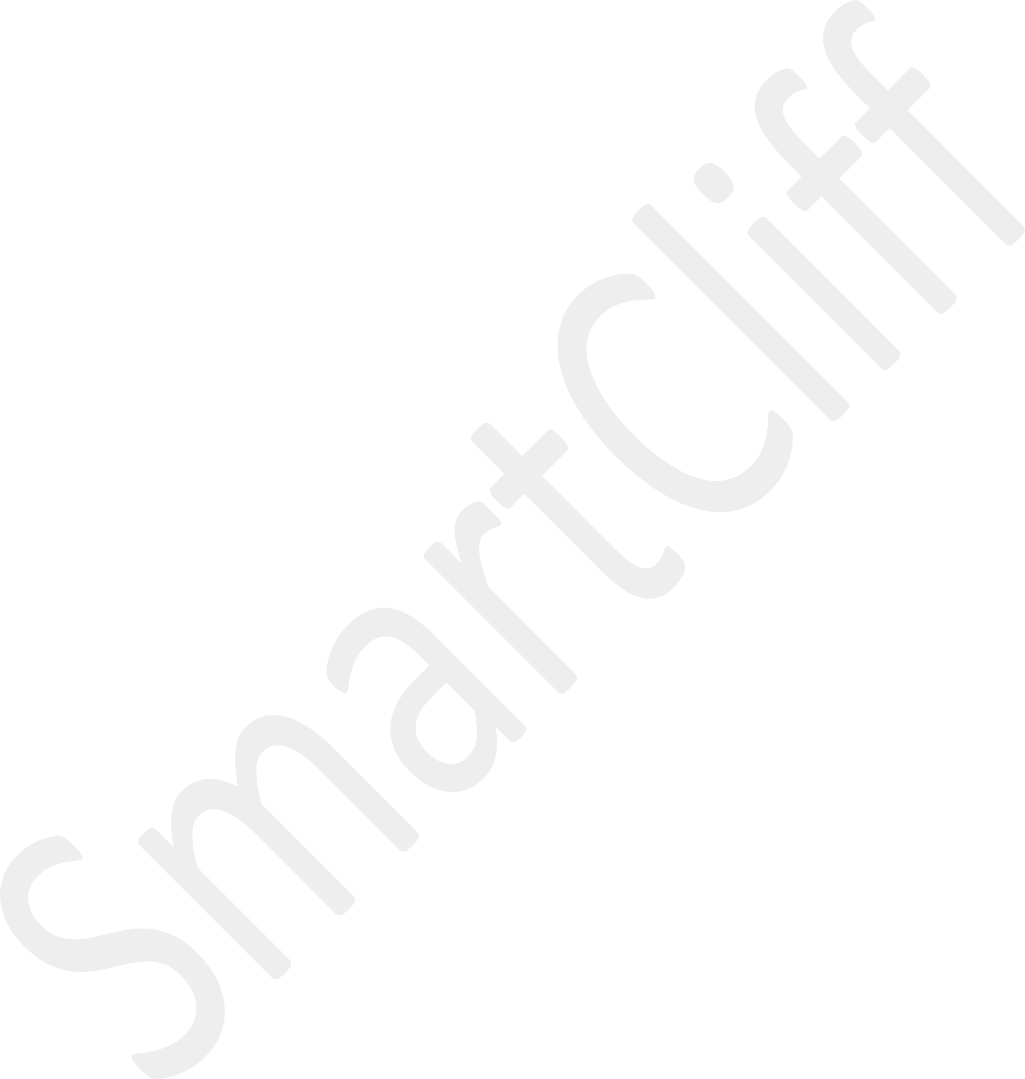
|  |  |  |
| --- | --- | --- |
|  | Input: head = [1,2,3,4,5,6,7,8,9,10,11], m = 1, n = 3  Output: [1,5,9]  Explanation: Head of linked list after removing nodes is returned.  **Constraints**:  The number of nodes in the list is in the range [1, 10^4]. 1 <= Node.val <= 10^6  1 <= m, n <= 1000  Program:  package com.SelfPractice;  import java.util.Scanner;  public class DeleteNnodeafterMnodes {  static void deleteafterM( Node head, int M, int N)  {  Node curr = head, t;  int count;  while (curr!=null)  {  for (count = 1; count < M && curr != null; count++)  curr = curr.next;  if (curr == null)  return;  t = curr.next;  for (count = 1; count <= N && t != null; count++)  {  Node temp = t;  t = t.next;  }  curr.next = t;  curr = t;  }  }  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list1 : ");  int n = sc.nextInt();  System.***out***.print("Enter the list1 elements : ");  Node head = new Node(sc.nextInt());  int count = 0;  Node curr = head;  while (count < n-1) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  count++;  }  System.***out***.print("Enter m nodes : ");  int m = sc.nextInt();  System.***out***.print("Enter n nodes : ");  n = sc.nextInt();  *deleteafterM*(head, m, n);  curr = head;  while (curr != null) {  System.***out***.print(curr.data + " ");  curr = curr.next;  }  }  }  Output:  Enter the size of the list1 : 13  Enter the list1 elements : 1 2 3 4 5 6 7 8 9 10 11 12 13  Enter m nodes : 2  Enter n nodes : 3  1 2 6 7 11 12 |  |
| **16** | **Retain last appearance**  **Problem statement :** You are given an unsorted Singly Linked List with 'N' nodes which may contain duplicate elements. You are supposed to remove all duplicate elements from the linked list and keep only the last appearance of elements.  **Input 1:**  7 11 13 23 7 11 13 -1  **Output 1:**  23 7 11 13 -1  **Explanation of Input 1:**  7, 11, and 13 have appeared two times, so we remove the first appearance of 7, 11, and 13 and keep their last appearance. Since 7, 11 and 13 are removed, so now the head of the modified, linked list becomes 23.  **Input 2:**  13 1 19 3 9 -1  **Output 2:**  13 1 19 3 9 -1  **Explanation of Input 2:**  There are no duplicate elements in the given linked list, so there will be no elements deleted. And the modified linked list will be the same as the given  linked list. | Hard |



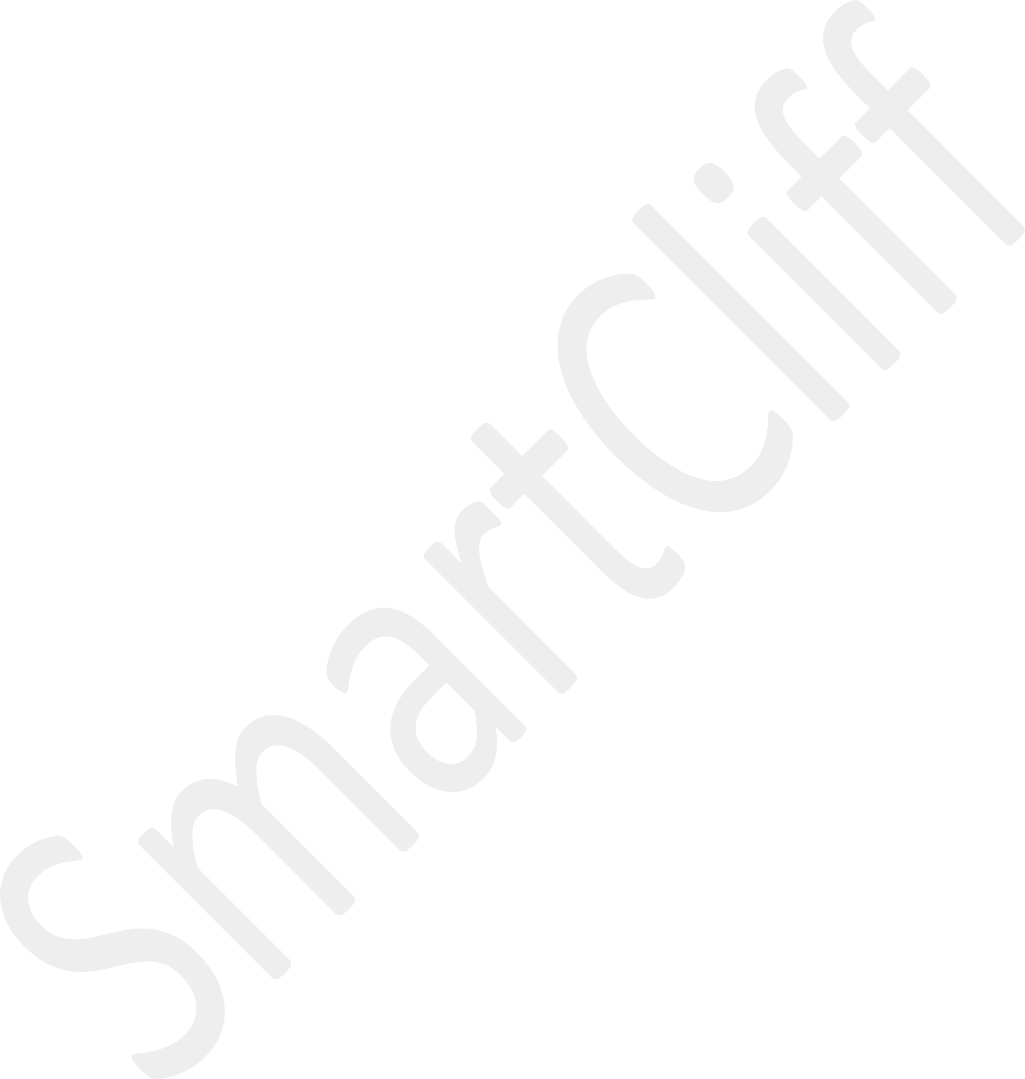
|  |  |  |
| --- | --- | --- |
|  | **Constraints :**  0 <= N <= 5 \* 10^5  -10^9 <= data <= 10^9 and data != -1  Where 'data' is the value of elements that exist in the linked list.  Program:  package com.SelfPractice;  import java.util.Arrays;  import java.util.Scanner;  public class RetainLastAppearance {  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list1 : ");  int n = sc.nextInt();  System.***out***.print("Enter the list1 elements : ");  Node head = new Node(sc.nextInt());  int count = 0;  Node curr = head;  int[] arr = new int[n];  while (count < n-1) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  count++;  }  curr = head;  int i = 0;  while (curr != null) {  arr[i] = curr.data;  curr = curr.next;  i++;  }  for (i=0;i<n;i++)  {  for (int j = i+1; j < n; j++) {  if (arr[i] == arr[j]) {  arr[i] = -1;  }  }  }  Node newHead = new Node(-1);  Node ptr = newHead;  for (int j = 0; j < n; j++) {  if (arr[j] != -1) {  ptr.next = new Node(arr[j]);  ptr = ptr.next;  }  }  newHead = newHead.next;  curr = newHead;  System.***out***.print("After retaining last appearance : ");  while (curr != null) {  System.***out***.print(curr.data + " ");  curr = curr.next;  }  }  }  Program:  Enter the size of the list1 : 7  Enter the list1 elements : 7 11 13 23 7 11 13  After retaining last appearance : 23 7 11 13 |  |
| **17** | **Divide a Linked list into Two** | Hard |
|  | **Problem statement :** You have been given a singly Linked List of integers. Your |  |
|  | task is to divide this list into two smaller singly-linked lists in which the nodes |  |
|  | appear in alternating fashion from the original list. |  |
|  | **For example:** |  |
|  | If the given linked list is 1 -> 2 -> 3 -> 4 -> 5 -> NULL |  |
|  | The first sub-list is 1 -> 3 -> 5 -> NULL. |  |
|  | The second sub-list is 2 -> 4 -> NULL. |  |
|  | If it is impossible to make any of the two smaller sub-lists, return an empty list |  |
|  | i.e. NULL. |  |
|  | **Input 1 :** |  |
|  | 2 |  |
|  | 1 2 3 4 5 -1 |  |
|  | 1 2 -1 |  |
|  | **Output 1 :** |  |
|  | 1 3 5 -1 |  |
|  | 2 4 -1 |  |
|  | 1 -1 |  |
|  | 2 -1 |  |
|  | **Explanation For Input 1 :** |  |
|  | For the first test case, we have 1, 3, 5 in the first and 2, 4 in the second linked |  |
|  | list. |  |
|  | For the second test case, we have 1 in the first and 2 in the second linked list. |  |
|  | **Input 2 :** |  |
|  | 2 |  |
|  | 1 2 3 -1 |  |
|  | 1 -1 |  |
|  | **Output 2 :** |  |
|  | 1 3 -1 |  |



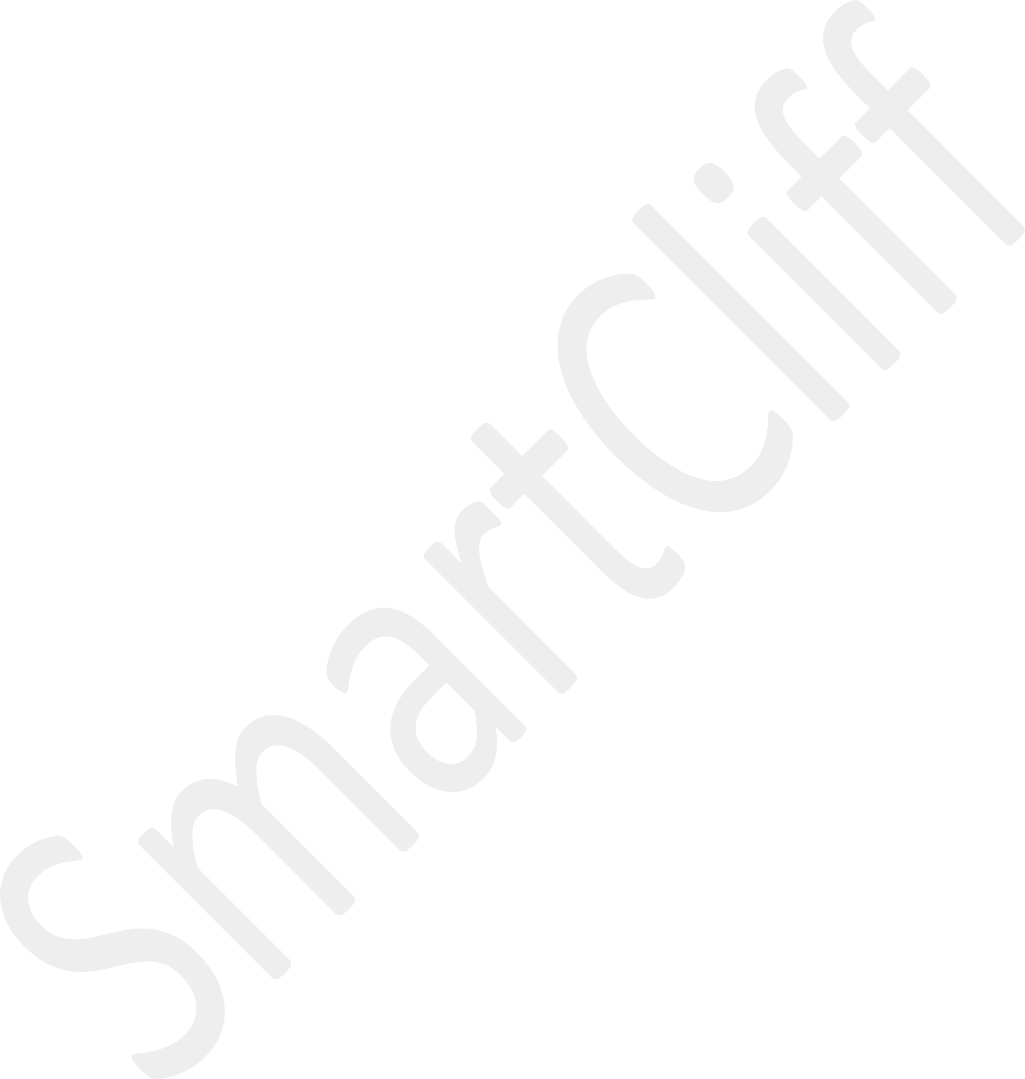
|  |  |  |
| --- | --- | --- |
|  | 2 -1  1 -1  -1  **Constraints :**  1 <= T <= 10^2  0 <= LEN <= 5\*10^3  1 <= data <= 10^9  Where LEN is the number of nodes in the Linked List.  Program:  package com.SelfPractice;  import java.util.Scanner;  public class DivideLinkedListintoTwo {  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list : ");  int n = sc.nextInt();  System.***out***.print("Enter the elements : ");  Node head = new Node(sc.nextInt());  int count = 0;  Node curr = head;  while (count < n-1) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  count++;  }  if (head.next == null) {  System.***out***.println("Divided list 1: " + head.data);  System.***out***.println("Divided list 2: ");  } else {  Node odd = head;  Node even = head.next;  Node oddHead = odd;  Node evenHead = even;  while (even != null && even.next != null) {  odd.next = even.next;  odd = odd.next;  even.next = odd.next;  even = even.next;  }  System.***out***.print("Divided list 1: ");  while (oddHead != null) {  System.***out***.print(oddHead.data + " ");  oddHead = oddHead.next;  }  System.***out***.println();  System.***out***.print("Divided list 2: ");  while (evenHead != null) {  System.***out***.print(evenHead.data + " ");  evenHead = evenHead.next;  }  }  }  }  Output:  Enter the size of the list : 5  Enter the elements : 1 2 3 4 5  Divided list 1: 1 3 5  Divided list 2: 2 4 |  |
| **18** | **Reverse List In K-Groups** | Hard |
|  | **Problem Statement:** You are given a linked list of 'n' nodes and an integer 'k', |  |
|  | where 'k' is less than or equal to 'n'. |  |
|  | Your task is to reverse the order of each group of 'k' consecutive nodes, if 'n' is |  |
|  | not divisible by 'k', then the last group of nodes should remain unchanged. |  |
|  | Implement a method that performs this reversal, and returns the head of the |  |
|  | modified linked list. |  |
|  | **Example :** |  |
|  | **Input** : 1->2->3->4->5 and k = 3 |  |
|  | **Output** : 3->2->1->4->5. |  |
|  | **Explanation** : We have to reverse the first three elements, and leave the last two |  |
|  | elements unchanged. Thus, the final linked list being 3->2->1->4->5. |  |
|  | **Input 1:** |  |
|  | 6 |  |
|  | 5 4 3 7 9 2 |  |
|  | 4 |  |
|  | **Output 1:** |  |
|  | 7 3 4 5 9 2 |  |
|  | **Explanation of the Input 1:** |  |
|  | For the given test case, we reverse the nodes in groups of four. But for the last 2 |  |
|  | elements, we cannot form a group of four, so leave them as they are. The linked |  |
|  | list becomes 7->3->4->5->9->2. Hence the output is 7 3 4 5 9 2 |  |
|  | **Input 2:** |  |
|  | 4 |  |
|  | 4 3 2 8 |  |
|  | 4 |  |
|  | **Output 2:** |  |
|  | 8 2 3 4 |  |

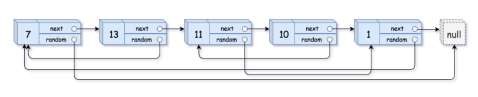


|  |  |  |
| --- | --- | --- |
|  | **Constraints:**   * 1 <= n <= 10^4 * 1 <= k <= n   Program:  package com.SelfPractice;  import java.util.Scanner;  public class RotateListinKgroup {  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list : ");  int n = sc.nextInt();  System.***out***.print("Enter the elements : ");  Node head = new Node(sc.nextInt());  int count = 0;  Node curr = head;  while (count < n-1) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  count++;  }  System.***out***.print("Enter k : ");  int k = sc.nextInt();  curr = head;  count = 0;  while (count < k-1 && curr.next != null) {  curr = curr.next;  count++;  }  Node curr2 = curr.next;  curr.next = null;  Node tra = head;  Node prev = null;  while (tra != null) {  Node next = tra.next;  tra.next = prev;  prev = tra;  tra = next;  }  head = prev;  while (prev.next != null) {  prev = prev.next;  }  prev.next = curr2;  curr = head;  while (curr != null) {  System.***out***.print(curr.data + " ");  curr = curr.next;  }  }  }  Output:  Enter the size of the list : 4  Enter the elements : 4 3 2 8  Enter k : 4  8 2 3 4 |  |
| **19** | **Rotate by k places**  **Problem Statement :** Given the head of a linked list, rotate the list to the right by k places  **Example** :    **Input**: head = [1,2,3,4,5], k = 2  **Output**: [4,5,1,2,3]  **Example 2:**    **Input:** head = [0,1,2], k = 4  **Output:** [2,0,1]  **Constraints:**   * The number of nodes in the list is in the range [0, 500]. | Hard |



|  |  |  |
| --- | --- | --- |
|  | * -100 <= Node.val <= 100 * 0 <= k <= 2 \* 109   Program:  package com.SelfPractice;  import java.util.Scanner;  public class RotatebyKplaces {  static Node rotateRight(Node head, int k) {  if (head == null || head.next == null) {  return head;  }  int length = 1;  Node curr = head;  while (curr.next != null) {  length++;  curr = curr.next;  }  k = k % length;  curr = head;  for (int i = 0; i < length - k - 1; i++) {  curr = curr.next;  }  Node newHead = curr.next;  curr.next = null;  curr = newHead;  while (curr.next != null) {  curr = curr.next;  }  curr.next = head;  return newHead;  }  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list: ");  int n = sc.nextInt();  System.***out***.print("Enter the elements: ");  Node head = new Node(sc.nextInt());  Node curr = head;  for (int i = 1; i < n; i++) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  }  System.***out***.print("Enter the k value: ");  int k = sc.nextInt();  head = *rotateRight*(head, k);  curr = head;  while (curr != null) {  System.***out***.print(curr.data + " ");  curr = curr.next;  }  System.***out***.println();  }  }  Program:  Enter the size of the list: 5  Enter the elements: 1 2 3 4 5  Enter the k value: 2  4 5 1 2 3 |  |
| **20** | **Copy List with Random Pointer**  **Problem Statement:** A linked list of length n is given such that each node contains an additional random pointer, which could point to any node in the list, or null.  Construct a deep copy of the list. The deep copy should consist of exactly n brand 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.random -  -> Y, then for the corresponding two nodes x and y in the copied list, x.random -  -> y.  Return the head of the copied linked list.  The linked list is represented in the input/output as a list of n nodes. Each node is represented as a pair of [val, random\_index] where:  val: an integer representing Node.val  random\_index: the index of the node (range from 0 to n-1) that the random pointer points to, or null if it does not point to any node.  **Example :**  **Input**: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]  **Output**: [[7,null],[13,0],[11,4],[10,2],[1,0]]  **Example 2:**    **Input:** head = [[1,1],[2,1]]  **Output:** [[1,1],[2,1]] | Hard |





|  |  |  |
| --- | --- | --- |
|  | **Example 3:**    **Input:** head = [[3,null],[3,0],[3,null]]  **Output:** [[3,null],[3,0],[3,null]]  **Constraints:**   * 0 <= n <= 1000 * -104 <= Node.val <= 104 * Node.random is null or is pointing to some node in the linked list.   Program:  package com.SelfPractice;  import java.util.Scanner;  class Node{  int data;  Node next;  Node prev;  public Object random;  Node(int data){  this.data = data;  this.next = null;  this.prev = null;  }  }  public class FindtheNthNodeFromTheEnd {  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  System.***out***.print("Enter the size of the list : ");  int n = sc.nextInt();  System.***out***.print("Enter the elements : ");  Node head = new Node(sc.nextInt());  int count = 0;  Node curr = head;  while (count < n-1) {  curr.next = new Node(sc.nextInt());  curr = curr.next;  count++;  }  System.***out***.print("Enter the nth node : ");  int nth = sc.nextInt();  curr = head;  int size = 1;  while (curr.next != null) {  curr = curr.next;  size++;  }  int delpos = size - nth+1;  curr = head;  count = 1;  if(delpos==1)  head = head.next;  else {  Node prev = null;  while (curr.next != null && count < delpos) {  prev = curr;  curr = curr.next;  count++;  }  prev.next = curr.next;  }  System.***out***.print("After deletion : ");  curr = head;  while (curr != null) {  System.***out***.print(curr.data + " ");  curr = curr.next;  }  }  }  Output:  Enter the size of the list : 5  Enter the elements : 1 2 3 4 5  Enter the nth node : 0 1 2 3 4  After deletion : 1 2 3 4 |  |

