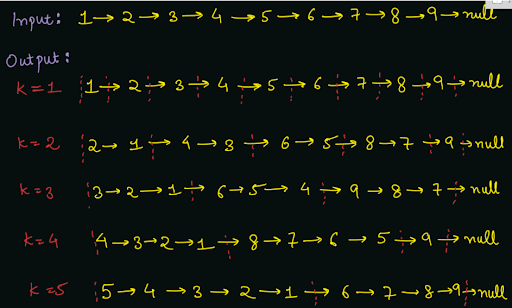
**1. Problem Discussion :**

You are given a partially written LinkedList class. (Input and Output is managed for you.) You are required to complete the body of the kReverse function. The function is expected to tweak the list such that all groups of k elements in the list get reversed and linked. If the last set has less than k elements, leave it as it is (don't reverse).

To understand the problem statement if you are not able to get it, I recommend you to watch the question video associated with this problem.

Example

****

**2. Approach :**

Deducing Algorithm

Before solving this problem, I request you to solve some easier problems based on this concept. We have covered problems: 'Odd Even Linked List' and 'Remove Duplicates from Sorted Linked List', and this problem follows a similar approach to those discussed there.

We will maintain two linked lists: previous and current which will be empty initially. Now, we will keep on removing the first element of the original linked list one by one using removeFirst() method. Let the current element removed be equal to val. If the size of the current linked list is less than k, then we can add a node in the front of the current linked list (with value = val), using the addFirst() method. By doing addFirst() one by one, we are adding the elements in reverse order, hence the segment of nodes in the current linked list will be reversed from the order in the original linked list.

Note: addFirst() will manage the corner cases if the current linked list is empty or not. Do not worry about these functions.

Else, the current linked list has size equal to k. Hence we got a segment of k nodes, which are in reverse order in the current linked list. We will place this whole segment at the last of the previous linked list. By placing the whole segment in last, I mean that, we will make the next pointer of the tail node of the previous linked list point to the head of the current linked list. Also, update the tail of the previous linked list as the tail of the current linked list, and increase previous' size by current's size. We will have to handle a corner case in this case, when the previous linked list is empty. In this case, tail will point to null, hence updating the next pointer of null node will give a run-time error. Thus, before appending the current linked list to the previous list, we will check if the previous list is empty or not (using previous' head != null). If it is empty, then we will simply make previous = current. Also, we will make the current list empty by making head and tail point to null, so that we can process the remaining original linked list.

Q) Have we handled all the corner cases? What about the condition when the last segment has nodes less than k?

R) So, it seems that the question was not that easy! Before removing the element from the original linked list, we will have to check if the size of the remaining list is less than k, then we need not reverse the linked list, hence we can directly append the remaining nodes to the previous list by removingFirst() in original linked list and addLast() in previous linked list.

After completing the above steps, we will be left with an empty original list, and the previous list will contain all the nodes in k-reversed form. Hence, we will update this.head = prev.head, this.tail = prev.tail and this.size = prev.size.

Please refer to the solution video if you find difficulty in understanding the algorithm completely.

Please I request you to give it a try before reading the code!

**3. CODE:**

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

public static class Node {

int data;

Node next;

}

public static class LinkedList {

Node head;

Node tail;

int size;

void addLast(int val) {

Node temp = new Node();

temp.data = val;

temp.next = null;

if (size == 0) {

head = tail = temp;

} else {

tail.next = temp;

tail = temp;

}

size++;

}

public int size() {

return size;

}

public void display() {

for (Node temp = head; temp != null; temp = temp.next) {

System.out.print(temp.data + " ");

}

System.out.println();

}

public void removeFirst() {

if (size == 0) {

System.out.println("List is empty");

} else if (size == 1) {

head = tail = null;

size = 0;

} else {

head = head.next;

size--;

}

}

public int getFirst() {

if (size == 0) {

System.out.println("List is empty");

return -1;

} else {

return head.data;

}

}

public int getLast() {

if (size == 0) {

System.out.println("List is empty");

return -1;

} else {

return tail.data;

}

}

public int getAt(int idx) {

if (size == 0) {

System.out.println("List is empty");

return -1;

} else if (idx < 0 || idx >= size) {

System.out.println("Invalid arguments");

return -1;

} else {

Node temp = head;

for (int i = 0; i < idx; i++) {

temp = temp.next;

}

return temp.data;

}

}

public void addFirst(int val) {

Node temp = new Node();

temp.data = val;

temp.next = head;

head = temp;

if (size == 0) {

tail = temp;

}

size++;

}

public void addAt(int idx, int val) {

if (idx < 0 || idx > size) {

System.out.println("Invalid arguments");

} else if (idx == 0) {

addFirst(val);

} else if (idx == size) {

addLast(val);

} else {

Node node = new Node();

node.data = val;

Node temp = head;

for (int i = 0; i < idx - 1; i++) {

temp = temp.next;

}

node.next = temp.next;

temp.next = node;

size++;

}

}

public void removeLast() {

if (size == 0) {

System.out.println("List is empty");

} else if (size == 1) {

head = tail = null;

size = 0;

} else {

Node temp = head;

for (int i = 0; i < size - 2; i++) {

temp = temp.next;

}

tail = temp;

tail.next = null;

size--;

}

}

public void removeAt(int idx) {

if (idx < 0 || idx >= size) {

System.out.println("Invalid arguments");

} else if (idx == 0) {

removeFirst();

} else if (idx == size - 1) {

removeLast();

} else {

Node temp = head;

for (int i = 0; i < idx - 1; i++) {

temp = temp.next;

}

temp.next = temp.next.next;

size--;

}

}

private Node getNodeAt(int idx) {

Node temp = head;

for (int i = 0; i < idx; i++) {

temp = temp.next;

}

return temp;

}

public void reverseDI() {

int li = 0;

int ri = size - 1;

while (li < ri) {

Node left = getNodeAt(li);

Node right = getNodeAt(ri);

int temp = left.data;

left.data = right.data;

right.data = temp;

li++;

ri--;

}

}

public void reversePI() {

if (size <= 1) {

return;

}

Node prev = null;

Node curr = head;

while (curr != null) {

Node next = curr.next;

curr.next = prev;

prev = curr;

curr = next;

}

Node temp = head;

head = tail;

tail = temp;

}

public int kthFromLast(int k) {

Node slow = head;

Node fast = head;

for (int i = 0; i < k; i++) {

fast = fast.next;

}

while (fast != tail) {

slow = slow.next;

fast = fast.next;

}

return slow.data;

}

public int mid() {

Node f = head;

Node s = head;

while (f.next != null && f.next.next != null) {

f = f.next.next;

s = s.next;

}

return s.data;

}

public static LinkedList mergeTwoSortedLists(LinkedList l1, LinkedList l2) {

LinkedList ml = new LinkedList();

Node one = l1.head;

Node two = l2.head;

while (one != null && two != null) {

if (one.data < two.data) {

ml.addLast(one.data);

one = one.next;

} else {

ml.addLast(two.data);

two = two.next;

}

}

while (one != null) {

ml.addLast(one.data);

one = one.next;

}

while (two != null) {

ml.addLast(two.data);

two = two.next;

}

return ml;

}

public static Node midNode(Node head, Node tail) {

Node f = head;

Node s = head;

while (f != tail && f.next != tail) {

f = f.next.next;

s = s.next;

}

return s;

}

public static LinkedList mergeSort(Node head, Node tail) {

if (head == tail) {

LinkedList br = new LinkedList();

br.addLast(head.data);

return br;

}

Node mid = midNode(head, tail);

LinkedList fsh = mergeSort(head, mid);

LinkedList ssh = mergeSort(mid.next, tail);

LinkedList sl = mergeTwoSortedLists(fsh, ssh);

return sl;

}

public void removeDuplicates() {

LinkedList res = new LinkedList();

while (this.size() > 0) {

int val = this.getFirst();

this.removeFirst();

if (res.size() == 0 || val != res.tail.data) {

res.addLast(val);

}

}

this.head = res.head;

this.tail = res.tail;

this.size = res.size;

}

public void oddEven() {

LinkedList odd = new LinkedList();

LinkedList even = new LinkedList();

while (this.size > 0) {

int val = this.getFirst();

this.removeFirst();

if (val % 2 == 0) {

even.addLast(val);

} else {

odd.addLast(val);

}

}

if (odd.size > 0 && even.size > 0) {

odd.tail.next = even.head;

this.head = odd.head;

this.tail = even.tail;

this.size = odd.size + even.size;

} else if (odd.size > 0) {

this.head = odd.head;

this.tail = odd.tail;

this.size = odd.size;

} else if (even.size > 0) {

this.head = even.head;

this.tail = even.tail;

this.size = even.size;

}

}

public void kReverse(int k) {

LinkedList prev = null;

while (this.size > 0) {

LinkedList curr = new LinkedList();

if (this.size >= k) {

for (int i = 0; i < k; i++) {

int val = this.getFirst();

this.removeFirst();

curr.addFirst(val);

}

} else {

int sz = this.size;

for (int i = 0; i < sz; i++) {

int val = this.getFirst();

this.removeFirst();

curr.addLast(val);

}

}

if (prev == null) {

prev = curr;

} else {

prev.tail.next = curr.head;

prev.tail = curr.tail;

prev.size += curr.size;

}

}

this.head = prev.head;

this.tail = prev.tail;

this.size = prev.size;

}

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int n1 = Integer.parseInt(br.readLine());

LinkedList l1 = new LinkedList();

String[] values1 = br.readLine().split(" ");

for (int i = 0; i < n1; i++) {

int d = Integer.parseInt(values1[i]);

l1.addLast(d);

}

int k = Integer.parseInt(br.readLine());

int a = Integer.parseInt(br.readLine());

int b = Integer.parseInt(br.readLine());

l1.display();

l1.kReverse(k);

l1.display();

l1.addFirst(a);

l1.addLast(b);

l1.display();

}

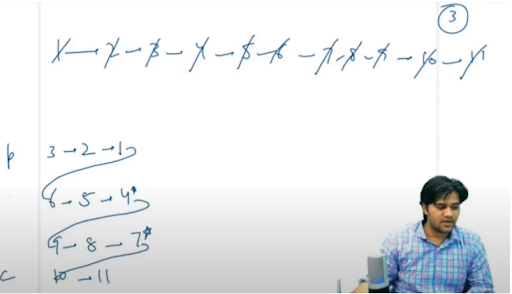
}

This code is written and explained by our team in this video. Please refer to it if you are stuck somewhere.

For more clarity of the question, watch the question video

Play Video

You should perform a dry run of the algorithm on some examples, to get a better understanding. It is also explained in the same video.

****

**4. Analysis:**

Time Complexity:

We are processing each node from the original linked list, and adding it in the current linked list (or directly to the previous linked list if size < k). Hence, time complexity is equal to O(n) where n = number of nodes in linked list.