*“You can either experience the pain of discipline or the pain of regret. The choice is yours.”*

Dear reader, I welcome you to a problem named ‘**’**. I must say, it is a simpler problem and you should try to deduce an algorithm after reading the problem statement on your own.

***Problem Statement***

* You are given a partially written LinkedList class. You are required to complete the body of ***removeDuplicates*** function. (Input and Output is managed for you.)
* You are provided with a ***sorted*** linked list. The function must remove all duplicates from the list in linear time and constant space.

Note: We can ***modify the original linked list*** to achieve constant space solution.

***Example:***

For the linked list: *1 -> 2 -> 2 -> 3 -> 3 -> 3 -> 5 -> 6 -> 6 -> null*, the resultant linked list with no duplicates will be: *1 -> 2 -> 3 -> 5 -> 6 -> null*.

I recommend you to watch the [question video](https://www.youtube.com/watch?v=T4Od7rqE2BM&list=TLGGXghYSXUeCoYyOTA2MjAyMQ), where the problem statement is explained in detail.

***Solution***

Algorithm is very simple, since the original linked list given is already sorted. Hence, all the duplicates in the linked list will occur ***consecutively***.

Thus, we need to take only the first occurrence of every duplicate element in our resultant linked list.

We will create an empty linked list res which will store the resultant linked list (without duplicates).

Now, we will keep on removing the first node of the original linked list using the ***removeFirst()*** function and check whether to add it in the resultant linked list or not.

If the resultant linked list is empty or the tail node is not equal to the current element, then we will add this current element in the resultant list using ***addLast()*** function, as it is the ***first*** (may or may not be the only) ***occurrence*** of the data element.

Else, if the tail node’s data is equal to the current node’s data, then this node is not the first occurrence, hence we will not add it in the resultant linked list.

Finally, we will get all the unique elements in the res linked list and the original linked list will become empty (as we did not traverse through it but kept on removing nodes).

We will make ***this.head point to res.head*** and ***this.tail point to res.tail***, and also make ***this.size equal to res.size***. By doing so, we are making the original linked list point to the resultant linked list.

Please refer to the [solution video](https://www.youtube.com/watch?v=ErSDF5IM1fo&list=TLGGh6zB1a35988yOTA2MjAyMQ) if you find difficulty in understanding the algorithm completely.

***Pseudo Code/ Algorithm***

1. Create empty linked list *res*.
2. While size of original linked list is greater than 0:
   1. Take the value of the first node’s data in a variable *val* and remove the first node of this linked list using ***removeFirst()*** function.
   2. If the resultant linked list is empty (head = null), or tail node’s value is not equal to val, then add a node in the last of the resultant linked list with data = val using ***addLast()*** function.
   3. Else, ignore the duplicate node and continue with the next iteration.
3. Update the original linked list by:
   1. this.head = res.head
   2. this.tail = res.tail
   3. this.size = res.size

***Implementation (Java)***

How about first trying by yourself without reading the code we provide?

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;

}

// removes duplicates from a sorted linked list

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 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);

}

l1.display();

l1.removeDuplicates();

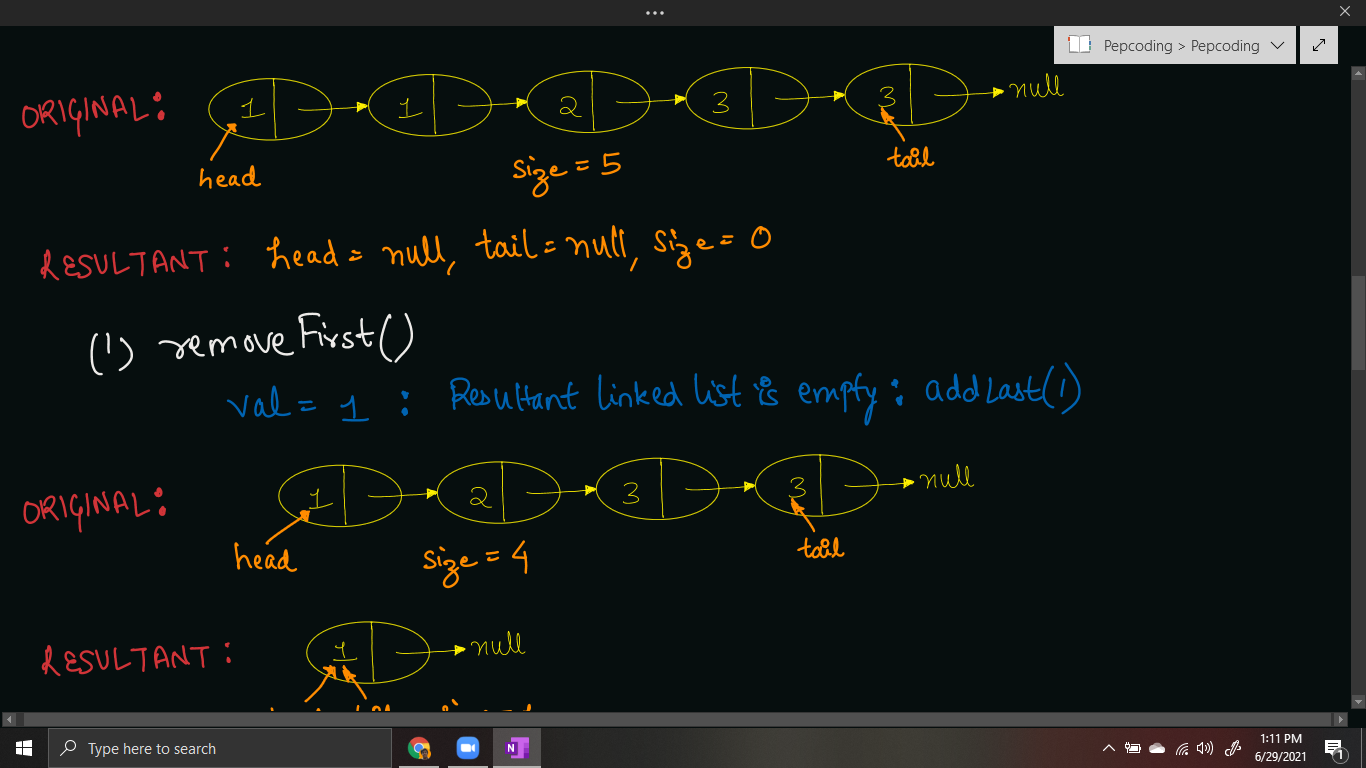
l1.display();

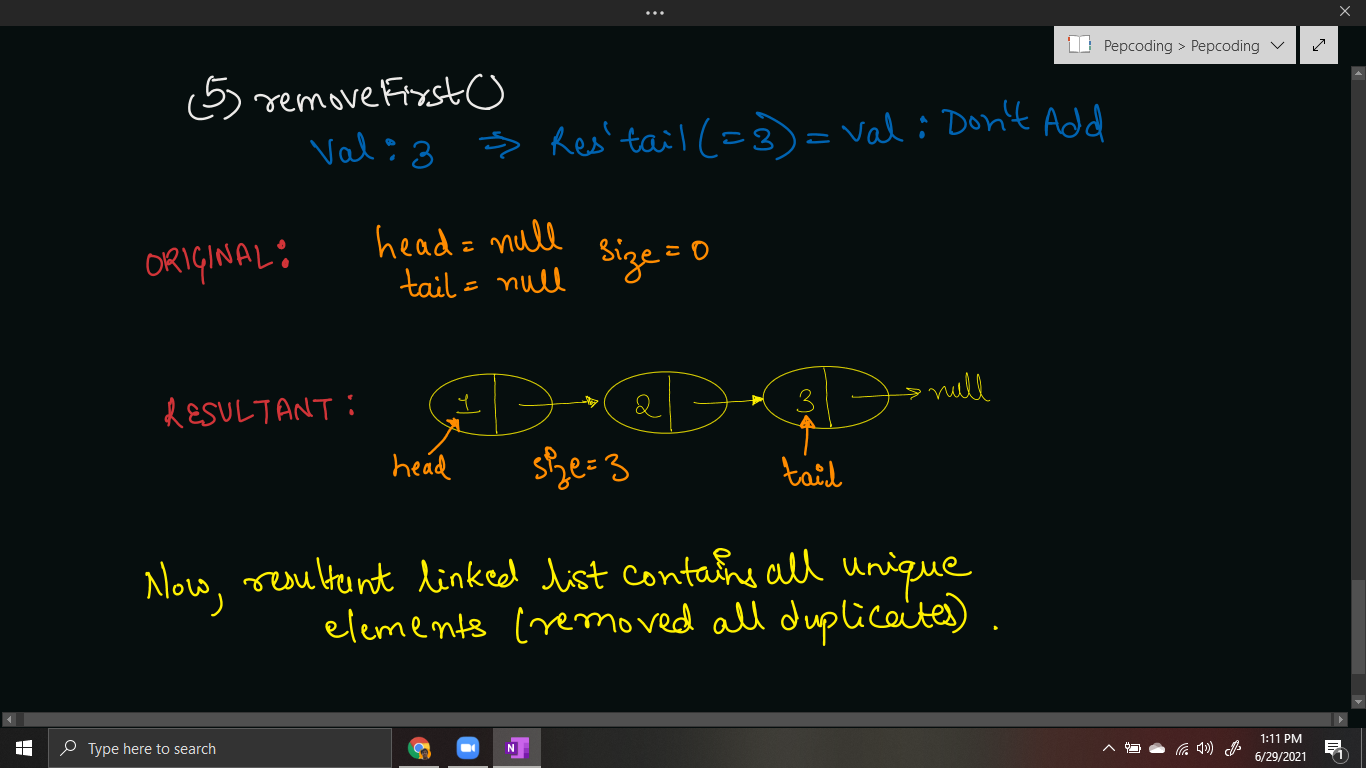
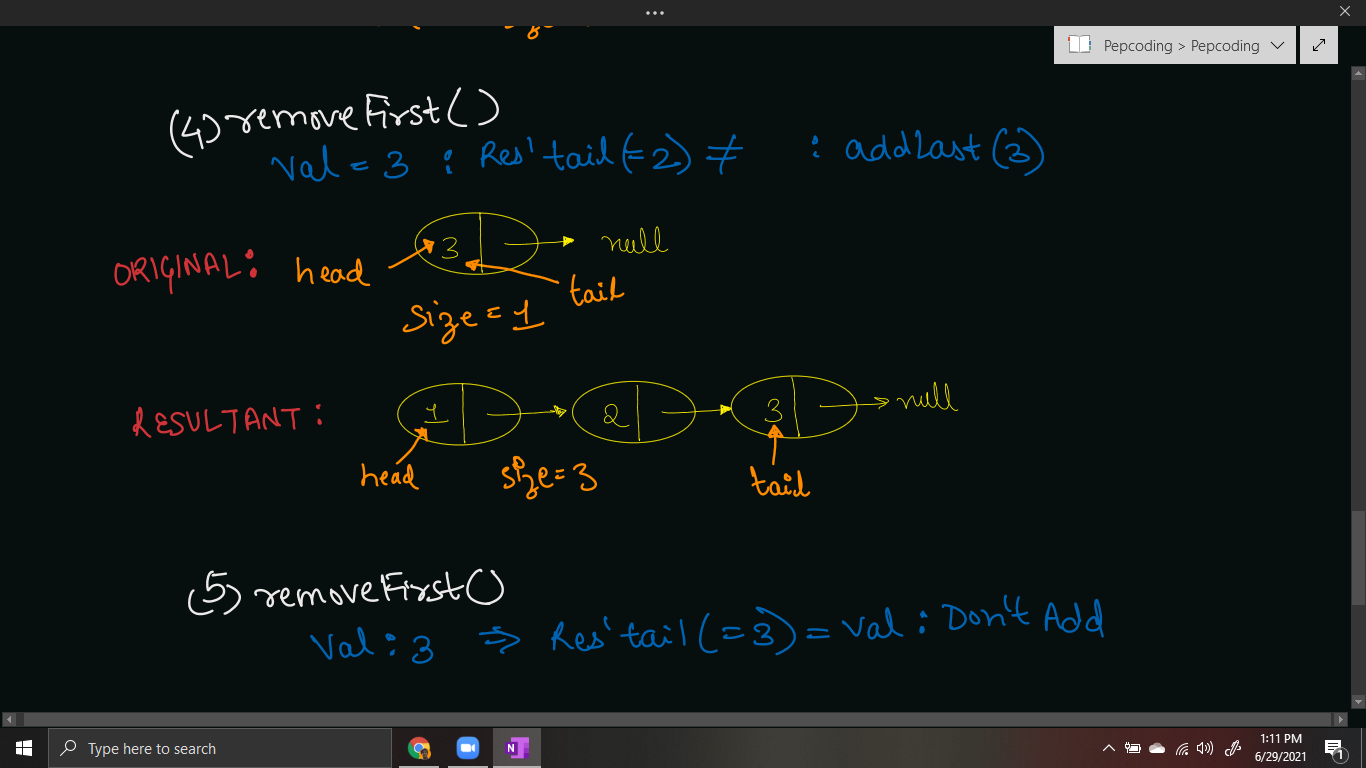
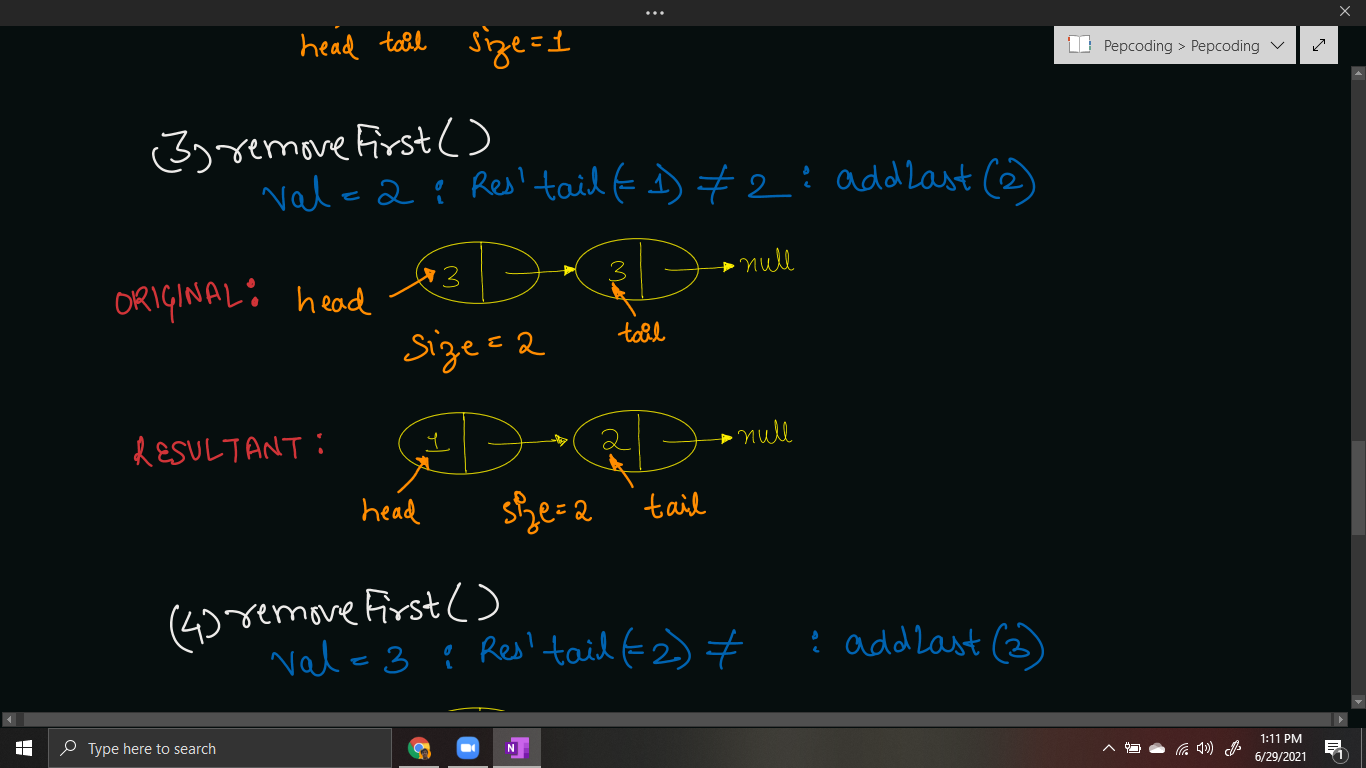
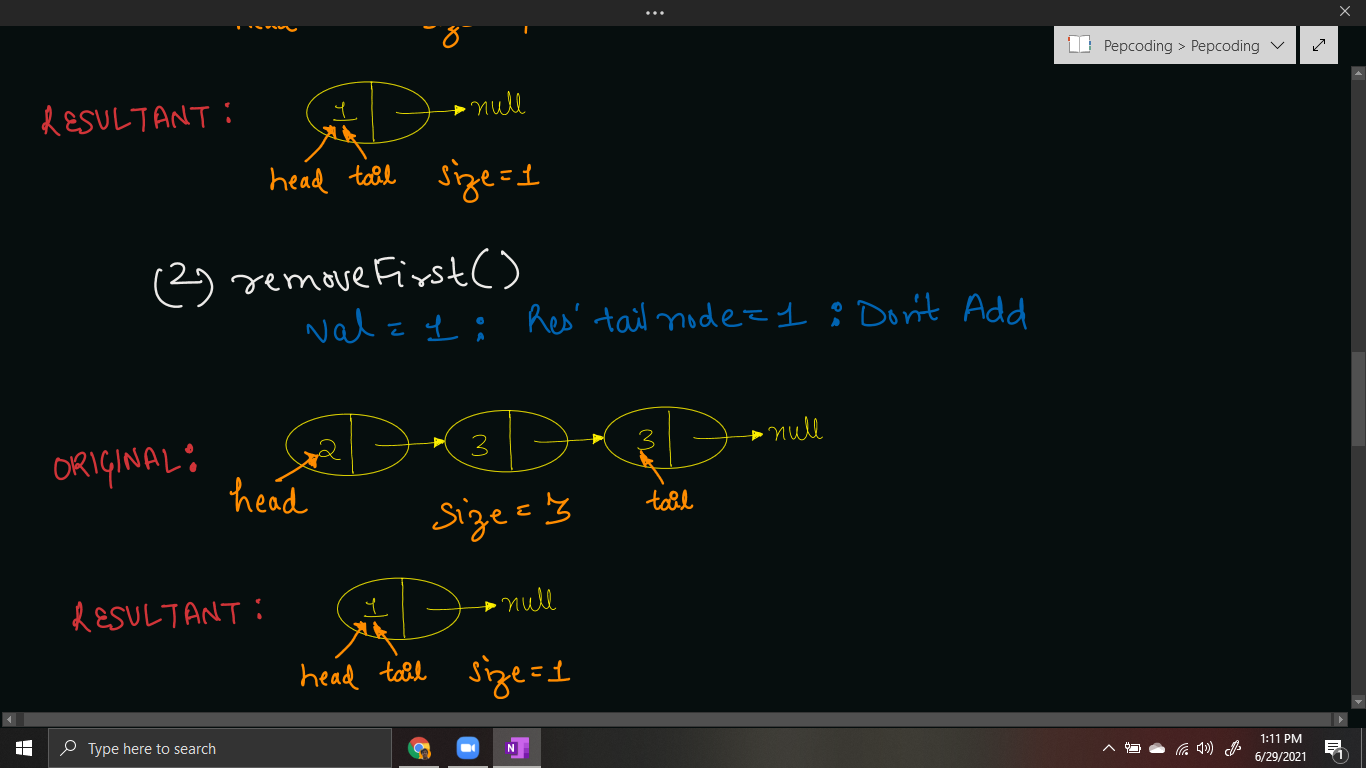
}

}

This code is written and explained by our team in [this video](https://www.youtube.com/watch?v=ErSDF5IM1fo&list=TLGGh6zB1a35988yOTA2MjAyMQ) from *[1:30, 4:55]*. Please refer to it if you are stuck somewhere.

***Example:*** Let us dry run the algorithm using a sample test case.





***Time & Space Complexity Analysis***

**Time Complexity** -

As we are calling removeFirst() method on original linked list for n times (n = size of linked list), thus time taken will be n \* O(1) = O(n).

For some nodes, we are calling addLast() method on the resultant linked list. In worst case, when all nodes are distinct/unique, we will call addLast() for n times, thus time taken will be n \* O(1) = O(n).

Hence, the total time taken will be O(n) + O(n) = ***O(n)*** only.

**Space Complexity** -

This is very interesting! We were asked to remove duplicates in constant space.

You can argue that we are creating a new linked list of size O(n) (in the worst case), thus we are not performing the removeDuplicates() operation in constant space.

But, my friend, you forgot that we are also **REMOVING** the nodes from the original linked list. By removeFirst() operation, we are removing the node and *deallocating the space taken by this node*. Only after removal, we are either creating a node (with same value) in the resultant linked list.

Hence, if the resultant linked list is of O(n) space, then the original linked list will be reduced to an empty list with constant space also.

We are only allocating extra space of 3 integer variables: ***head, tail*** *and* ***size*** for the new linked list. Hence the space taken is 4 \* 3 bytes = 12 bytes = O(1) constant space only, and this space (of 12 bytes) is constant irrespective of the number of nodes present in the linked list.

***Suggestion***: Even if you are not able to solve such easy tagged problems, please do not worry. Atleast, you are trying hard and giving your best. Whether the problem was completely solved or not does not matter, what matters is whether you gave an honest attempt or not.

I hope you enjoyed this problem of linked lists. Next problem to tackle is ‘[Odd Even Linked List](https://www.pepcoding.com/resources/online-java-foundation/linked-lists/odd-even-linked-list-official/ojquestion)’, which is a very interesting and medium difficulty problem. Good Bye!

Contributor : [Archit Aggarwal](https://www.linkedin.com/in/archit-aggarwal-6a7716189/)