Introduction

First, I want to talk about my program. There are 2 classes and 1 interface in this program. The concept of interface occupies an important place in Java. Methods defined on an interface are bodyless. The class that implements the interface has to use the methods in the interface. This program has an interface called HW2Interface. There are methods to be written in this interface. There is a class called Double Link Node. In this class element, right and left are defined. Another class is the LinkedList class. The LinkedList class implements the interface, and the methods are written here, and double link list operations are performed.

The program gives a warning message in case of errors in the insertion and deletion methods. For example, when trying to add an element to a nonexistent index or delete an element from a nonexistent index, the program gives an error message and then continues from where it left off.

After understanding how the program should work, the algorithm was rationalized and started to be written. When the writing process is finished, the data requested in the assignment has been tested and the output has been obtained correctly.

Algorithm

Head and tail are created for the double link list. Abstract method is written, and initial values are assigned as null.

```
public class LinkedList implements HW2Interface {

    DoubleLinkNode head;
    DoubleLinkNode tail;

    public LinkedList() {
        head = null;
        tail = null;
    }
```

public void Insert(int newElement, int pos)

The value and position information of the element to be added in integer type are given in the insertion method. Created two DoubleLinkNode type variables (isaretci1, isaretci2). One of them isaretci2 indicates head and the other one indicates null. Adding an element consists of 4 main topics. If there is no element in the list, add the first element, prepend, append to a middle index, or append it to the end.

- If there are no elements in the list, the first element displays both head and tail.

```
if (head == null) {
    // liste boş
    head = yenieleman;
    tail = yenieleman;
```

- If the position to be added is the 0th index, a new node named n is created and the right of this node shows the old head element. New head, new element is made.

```
} else if (pos == 0) { // basa ekleme
    DoubleLinkNode n = new DoubleLinkNode( eleman: newElement);
    n.right = head;
    if (head != null) {
        head.left = n;
    }
    head = n;
```

- In order to add to the middle and the end, the isaretci1 and isaretci2 that we created at the beginning are hovered in the list. If isaretci2 shows null, it means that tail will be added and the new element is made tail. If it is not null, it is added to the middle.

```
        } else {
            for (int i = 0; i < pos; i++) {
                isaretci1 = isaretci2;
                isaretci2 = isaretci2.right;
            }
            if (isaretci2 == null) {
                isaretci1.right = yenieleman;
                tail = yenieleman;
            } else {
                yenieleman.right = isaretci2;
                isaretci2.left = yenieleman;
                isaretci1.right = yenieleman;
                yenieleman.left = isaretci1;

            }
        }
```

public void Delete(int pos)

In the delete method, the position information of the element to be singled out is given. Created two DoubleLinkNode type variables (isaretci1, isaretci2). One of them isaretci2 indicates head and the other one indicates null. Element deletion consists of 3 main headings. Deleting the leading element, deleting the middle element, and deleting the last element.

- For deletion from the head, we set the left to null. We equate the right of the head to the head.

```
if (pos == 0) {
    // bastan silme
    head = isaretci2.right;
    isaretci2.left = null;
```

- For end and middle deletion operations, as in addition, pointers are created to traverse the list, and when isaretci2.right is null, it performs end deletion.

```
} else {
    for (int i = 0; i < pos; i++) {
        isaretci1 = isaretci2;
        isaretci2 = isaretci2.right;
    }
    if (isaretci2.right == null) {
        // sondan sil
        isaretci1.right = null;
        tail = isaretci1;
    } else { //aradan sil
        isaretci1.right = isaretci2.right;
        isaretci1.right.left = isaretci2.left;
    }
}
```

public void LinkReverse()

Created 3 DoubleLinkNode variables named temp, prev and q. The list was traversed until q became null and the arrows reversed their direction. So, the list has changed direction.

```
@Override
public void LinkReverse() {
    DoubleLinkNode temp;
    DoubleLinkNode prev = null;
    DoubleLinkNode q = head;
    while (q != null) {
        temp = q.right;
        q.right = prev;
        prev = q;
        q = temp;
    }
    head = prev;
}
```

public void Output():

The list is printed by traversing all the elements of the list. A pointer was used for this.

public void ReverseOutput():

A new list is created, and the existing list is thrown there, and then reverse printing is performed.

public int GetNth(int index):

A variable of type DoubleLinkNode named current was created and displayed as the head. Then a counter with an initial value of zero was created. When count is equal to the entered index, the element there is returned, count and current are updated. Thanks to this method, the element in the desired index is brought to us and we can use it.

```
public int GetNth(int index) {
    DoubleLinkNode current = head;
    int count = 0;
    while (current != null) {
        if (count == index) {
            return current.Element;
        }
        count++;
        current = current.right;
    }

    assert (false);
    return 0;
}
```

static int findSize(DoubleLinkNode node):

The list is navigated by moving the variable named Node to the right, and the size of the list is found by increasing the res in each scroll.

```
static int findSize(DoubleLinkNode node) {
    int res = 0;
    while (node != null) {
        // System.out.println(node.Element);
        res++;
        node = node.right;
    }

    return res;
}
```

public void SacuraL():

In the Sacral method, a list is created and the elements of our list are transferred to this list in order. A variable named PreviousData is created, and this variable returns the 0th index of the list with the getNth method. Creating the count variable with an initial value of 0. The for loop navigates to the last index of the list and currentData i. syncs to the variable. Then, if previousData is equal to currentData, count is increased by 1 and added to the list with count and Insert method as an element. If not, the process continues by adding it to our list named resultData. Finally, the head is updated to resultData.head.

```
}
list.tail = head;
int listsize = findSize( node: list.head);
// list.Output();

LinkedList resultData = new LinkedList();

Integer previousData = list.GetNth( index: 0);
Integer currentData;
int count = 0;
int j = 0;
for (int i = 0; i < findSize( node: list.head); i++) {
    currentData = list.GetNth( index: i);

    if (previousData.equals( obj: currentData)) {
        count += 1;

    } else {
        resultData.Insert( newElement: previousData,  pos: j);
        resultData.Insert( newElement: count, j + 1);
        count = 1;
        j = j + 2;

    }
    previousData = currentData;
}

resultData.Insert( newElement: previousData,  pos: j);
resultData.Insert( newElement: count, j + 1);

head = resultData.head;
```

public void OpacuraL()

First, a list named lis1 and newlist is created. Elements are placed in list1. A boolean variable named isNew and isFinish is created. These are displayed as true and false. It is traversing list1 in the for loop. When i is an even number and isNew is true , the variable number returns the

number getNth(i). In short, the number of the element entered here is determined. Then, if the (i+1)nth index in the list is not equal to count, count is incremented and number is added to the 0th index of the newList. i is decremented by one and isNew is false. If count is equal, isNew is true, count is 0, and isFinish is false. Thanks to this method, the desired element is written as many as the desired number. Then with linkreverse the list is reversed, and tail and head are updated.

```
DoubleLinkNode node = head;
LinkedList list1 = new LinkedList();
LinkedList yeniListe = new LinkedList();

int deney = 0;
while (node != null) {
    list1.Insert( newElement: node.Element,  pos: deney);
    deney++;
    node = node.right;
}
// list1.tail=head;
boolean isNew = true;
boolean isFinish = false;
int number = 0;
int count = 0;
for (int i = 0; i < findSize( node: list1.head) - 1; i++) {
    if (i % 2 == 0 && isNew) {
        number = list1.GetNth( index: i);
        isFinish = true;
    }
    if (isFinish) {
        if (count != list1.GetNth(i + 1)) {
            count++;
            yeniListe.Insert( newElement: number,  pos: 0);
            i--;
            isNew = false;
        } else {
            isNew = true;
            count = 0;
            isFinish = false;
        }
    }
```