Zadanie 1:

public class OneWayLinkedList<E> {  
  
 // węzeł:  
  
 public static class Element<E> {  
 E value;  
 Element<E> next;  
  
 public Element(E value) {  
 this.value = value;  
 }  
  
 @Override  
 public String toString() {  
 return value.toString();  
 }  
 }  
  
  
  
 // pierwszy  
 private Element<E> first;  
 private int size;  
  
 public OneWayLinkedList() {  
 first = null;  
 size = 0;  
 }  
  
 public boolean isEmpty() {  
 return first == null;  
 }  
  
 // operacja o złożoności O(1)  
 public int Size() {  
 return size;  
 }  
  
 public E get(int pos) {  
 if (isEmpty() || pos < 0) {  
 throw new IndexOutOfBoundsException("index: " + pos);  
 }  
  
 int currentPos = pos;  
 Element<E> actElem = first;  
  
 while (currentPos > 0) {  
 if (actElem == null) {  
 throw new IndexOutOfBoundsException("index: " + pos);  
 }  
  
 currentPos--;  
 actElem = actElem.next;  
 }  
  
 return actElem.value;  
 }  
  
  
 public E set(int pos, E e) {  
 if (isEmpty() || pos < 0) {  
 throw new IndexOutOfBoundsException("index: " + pos);  
 }  
  
 int currentPos = pos;  
 Element<E> actElem = first;  
  
 while(currentPos > 0) {  
 if (actElem == null) {  
 throw new IndexOutOfBoundsException("index: " + pos);  
 }  
  
 currentPos--;  
 actElem = actElem.next;  
 }  
  
 E retValue = actElem.value;  
 actElem.value = e;  
 return retValue;  
 }  
  
  
  
 // metody pomocnicze:  
 private static class ElementPair<E> {  
 private final Element<E> previous;  
 private final Element<E> current;  
  
 public ElementPair(Element<E> previous, Element<E> current) {  
 this.previous = previous;  
 this.current = current;  
 }  
 }  
  
 // biorę dwa elementy, ten na indeksie i ten przed nim  
 private ElementPair<E> getElementWithPrevious(int index) {  
 if (isEmpty() || index < 0) {  
 throw new IndexOutOfBoundsException("index: " + index);  
 }  
  
 Element<E> previousElement = null;  
 Element<E> currentElement = first;  
 int currentIndex = index;  
  
 while(currentIndex > 0) {  
 if (currentElement == null) {  
 throw new IndexOutOfBoundsException("index: " + index);  
 }  
  
 previousElement = currentElement;  
 currentElement = currentElement.next;  
 currentIndex--;  
 }  
 return new ElementPair<>(previousElement, currentElement);  
 }  
  
  
  
 // reszta metod:  
  
 public void insert(int pos, E e) {  
  
 // dodanie do pustej:  
 if (pos == 0 && first == null) {  
 first = new Element<>(e);  
 size++;  
 return;  
 }  
  
 ElementPair<E> pair = getElementWithPrevious(pos);  
 Element<E> previousElement = pair.previous;  
 Element<E> elementAtIndex = pair.current;  
  
 // dodanie na początek:  
 if (previousElement == null) {  
 first = new Element<>(e);  
 first.next = elementAtIndex;  
 size++;  
 return;  
 }  
  
 // w pozostałym miejscu:  
 Element<E> newELem = new Element<>(e);  
 newELem.next = elementAtIndex;  
 previousElement.next = newELem;  
 size++;  
 }  
  
 public void addEnd(E e) {  
 Element<E> actElem = first;  
  
 if (Size() == 0) {  
 first = new Element<>(e);  
 size++;  
 return;  
 }  
  
 if (Size() == 1) {  
 first.next = new Element<>(e);  
 size++;  
 return;  
 }  
  
 int counter = 0;  
  
 while (counter < size - 1) {  
 actElem = actElem.next;  
 counter++;  
 }  
  
 size++;  
 actElem.next = new Element<>(e);  
 }  
  
  
 boolean contains(E e) {  
 Element<E> actElem = first;  
  
 while (actElem != null) {  
 if (actElem.value == e)  
 break;  
 actElem = actElem.next;  
 }  
 return actElem != null;  
 }  
  
 public void clear() {  
 first = null;  
 size = 0;  
 }  
  
 public E deletePos(int pos) {  
 if (isEmpty() || pos < 0) {  
 throw new IndexOutOfBoundsException("index: " + pos);  
 }  
  
 ElementPair<E> pair = getElementWithPrevious(pos);  
 Element<E> previousElement = pair.previous;  
 Element<E> elementToDelete = pair.current;  
 E removedELement = elementToDelete.value;  
  
 // wyrzucenie pierwszego:  
 if (previousElement == null) {  
 first = elementToDelete.next;  
 size--;  
 return removedELement;  
 }  
  
 previousElement.next = elementToDelete.next;  
 size--;  
 return removedELement;  
 }  
  
 public boolean delete(E e) {  
 if (Size() == 0) {  
 return false;  
 }  
  
 if (Size() == 1) {  
 first = null;  
 size--;  
 return true;  
 }  
  
 Element<E> actElem = first;  
 int counter = 0;  
  
 while (counter < size-1) {  
 if (actElem.next.value == e) {  
 size--;  
 actElem.next = actElem.next.next;  
 return true;  
 }  
  
 actElem = actElem.next;  
 counter++;  
 }  
  
 if (actElem.value == e) {  
 size--;  
 return true;  
 }  
  
 return false;  
 }  
  
 public E deleteElem(Element<E> e) {  
 if (Size() == 0) {  
 throw new IndexOutOfBoundsException();  
 }  
  
 if (Size() == 1) {  
 E retVal = first.value;  
 first = null;  
 size--;  
 return retVal;  
 }  
  
 Element<E> actElem = first;  
 if (e.value == actElem.value) {  
 size--;  
 E retVal = actElem.value;  
 first = actElem.next;  
 return retVal;  
 }  
  
 int counter = 0;  
  
 while (counter < size-1) {  
 if (actElem.next.value == e.value) {  
 size--;  
 E retVal = actElem.next.value;  
 actElem.next = actElem.next.next;  
 return retVal;  
 }  
  
 actElem = actElem.next;  
 counter++;  
 }  
  
 if (actElem.value == e.value) {  
 size--;  
 return actElem.value;  
 }  
  
 return null;  
 }  
  
 public void wyswietlListe() {  
 if (size == 0) {  
 System.*out*.println("lista jest pusta");  
 }  
  
 Element<E> actElem = first;  
 int counter = 0;  
  
 while (counter < size) {  
 System.*out*.println(actElem.toString());  
 actElem = actElem.next;  
 counter++;  
 }  
 }  
}

demonstracja działania (klasa testowa w której wszystkie testy przeszły):

package Za1;  
  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class OneWayLinkedListTest {  
 OneWayLinkedList <Integer> list;  
  
 @BeforeEach  
 public void setUp() {  
 list = new OneWayLinkedList<>();  
 }  
  
 @Test  
 public void isThrowingExceptionWhenGettingFromEmptyList() {  
 *assertThrows*(  
 IndexOutOfBoundsException.class,  
 () -> list.get(0)  
 );  
  
 *assertThrows*(  
 IndexOutOfBoundsException.class,  
 () -> list.get(10)  
 );  
 }  
  
 @Test  
 public void isinsertingAndRemovingWorking() {  
 list.insert(0, 1);  
  
 *assertEquals*(1, list.get(0));  
  
 list.insert(1, 2);  
 list.insert(2, 3);  
 list.insert(3, 4);  
  
 int a = list.deletePos(3);  
 *assertEquals*(a, 4);  
  
 list.deletePos(2);  
 list.deletePos(1);  
 list.deletePos(0);  
  
 *assertThrows*(  
 IndexOutOfBoundsException.class,  
 () -> list.deletePos(0)  
 );  
  
 list.addEnd(1);  
 list.addEnd(2);  
  
 *assertEquals*(list.get(0), 1);  
 *assertEquals*(list.deletePos(1), 2);  
 list.clear();  
 }  
  
 @Test  
 public void isSizeWorking() {  
 list.insert(0, 1);  
 list.insert(1, 2);  
 list.insert(2, 3);  
 list.insert(3, 4);  
 *assertEquals*(4, list.Size());  
  
 list.deletePos(0);  
 list.deletePos(1);  
 *assertEquals*(2, list.Size());  
  
 list.clear();  
 *assertEquals*(0, list.Size());  
 }  
  
 @Test  
 public void isContainsWorking() {  
 list.insert(0, 1);  
 list.insert(1, 2);  
 list.insert(2, 3);  
 list.insert(3, 4);  
  
 *assertTrue*(list.contains(2));  
 *assertTrue*(list.contains(4));  
 *assertTrue*(list.contains(1));  
 *assertFalse*(list.contains(5));  
 }  
  
 @Test  
 public void isSettingAndDeletingWorking() {  
  
 // boolean delete(E e)  
 list.addEnd(1);  
 list.addEnd(2);  
 list.addEnd(3);  
 list.addEnd(4);  
  
 list.set(1,10);  
 *assertEquals*(list.get(1), 10);  
  
 list.delete(10);  
 *assertEquals*(list.get(1), 3);  
  
 list.delete(4);  
 *assertEquals*(list.get(list.Size()-1), 3);  
 list.clear();  
  
  
 //E delete(Element e)  
 list.addEnd(1);  
 list.addEnd(2);  
 list.addEnd(3);  
 list.addEnd(4);  
  
 int a = list.deleteElem(new OneWayLinkedList.Element(4));  
 *assertEquals*(a, 4);  
 *assertEquals*(list.Size(),3 );  
  
 int b = list.deleteElem(new OneWayLinkedList.Element<>(2));  
 *assertEquals*(b, 2);  
 }  
  
 @Test  
 public void Show() {  
 list.addEnd(1);  
 list.wyswietlListe();  
 System.*out*.println();  
  
 list.addEnd(2);  
 list.addEnd(3);  
 list.addEnd(4);  
 list.addEnd(5);  
 list.wyswietlListe();  
 System.*out*.println();  
  
 list.delete(3);  
 list.wyswietlListe();  
 System.*out*.println();  
 }  
  
  
}

Zadanie 2

import Za1.OneWayLinkedList;  
  
public class CircleGame {  
 private final int n;  
 private final int k;  
 OneWayLinkedList<Integer> list;  
  
 public CircleGame(int n, int k) {  
 this.n = n;  
 this.k = k;  
 list = new OneWayLinkedList<>();  
 }  
  
 private void FillTheList() {  
 for (int i = 1; i <= n; i++) {  
 list.addEnd(i);  
 }  
 }  
  
 public int LastSurvived() {  
 FillTheList();  
 int i = k;  
 int loopCounter = 0;  
 int killed;  
  
 if (k == n) {  
 return 0;  
 }  
  
 while (list.Size() != 1) {  
 killed = list.deleteElem(new OneWayLinkedList.Element<>(i));  
 System.*out*.println("Killed in " + ++loopCounter + " loop: " + killed);  
 i += k;  
 i %= n;  
 }  
  
 return list.get(0);  
 }  
  
  
 public static void main(String[] args) {  
 CircleGame circleGame = new CircleGame(7, 3);  
 int a = circleGame.LastSurvived();  
 System.*out*.println("Last man standing: " + a);  
 }  
  
}

Zadanie 3:

Podpunkt a kolejka:

Błąd:

public class EmptyQueueException extends Exception{  
}

kolejka:

public class MyQueue<T> {  
  
 private final OneWayLinkedList<T> oneWayLinkedList;  
  
 public MyQueue() {  
 oneWayLinkedList = new OneWayLinkedList<>();  
 }  
  
 public boolean isEmpty() {  
 return oneWayLinkedList.isEmpty();  
 }  
  
 public T dequeue() throws EmptyQueueException {  
 if (isEmpty())  
 throw new EmptyQueueException();  
  
 T retVal = oneWayLinkedList.deletePos(0);  
 return retVal;  
 }  
  
 public void enqueue(T elem) {  
 oneWayLinkedList.insert(oneWayLinkedList.Size(), elem);  
 }  
  
 public int Size() {  
 return oneWayLinkedList.Size();  
 }  
  
 public T first() throws EmptyQueueException {  
 if (isEmpty())  
 throw new EmptyQueueException();  
 return oneWayLinkedList.get(0);  
 }  
}

demonstracja działania:

class MyQueueTest {  
  
 MyQueue <Integer> myQueue;  
  
 @BeforeEach  
 public void setUp() {  
 myQueue = new MyQueue<>();  
 }  
  
 @Test  
 public void isThrowingExceptions() {  
 *assertThrows*(  
 EmptyQueueException.class,  
 () -> myQueue.dequeue()  
 );  
  
  
 *assertThrows*(  
 EmptyQueueException.class,  
 () -> myQueue.first()  
 );  
 }  
  
 @Test  
 public void isEnqueueAndDequeueWorking() throws EmptyQueueException{  
 myQueue.enqueue(1);  
 myQueue.enqueue(2);  
 myQueue.enqueue(3);  
  
 *assertEquals*(myQueue.first(), 1);  
 *assertEquals*(myQueue.dequeue(), 1);  
 *assertEquals*(myQueue.dequeue(), 2);  
 *assertEquals*(myQueue.dequeue(), 3);  
 }  
  
 @Test  
 public void isSizeWorking() throws EmptyQueueException {  
  
 myQueue.enqueue(1);  
 myQueue.enqueue(2);  
 myQueue.enqueue(3);  
 *assertEquals*(3, myQueue.Size());  
  
 myQueue.dequeue();  
 myQueue.dequeue();  
 *assertEquals*(1, myQueue.Size());  
 }  
}

podpunkt b stos:

Błąd:

public class EmptyStackException extends Exception {  
}

Stos:

public class MyStack<T> {  
  
 private final OneWayLinkedList<T> oneWayLinkedList;  
  
 public MyStack() {  
 oneWayLinkedList = new OneWayLinkedList<>();  
 }  
  
 public boolean isEmpty() {  
 return oneWayLinkedList.isEmpty();  
 }  
  
 public void push(T elem) {  
 oneWayLinkedList.insert(oneWayLinkedList.Size(), elem);  
 }  
  
 public T pop() throws EmptyStackException {  
 if (isEmpty())  
 throw new EmptyStackException();  
 T retVal = oneWayLinkedList.deletePos(oneWayLinkedList.Size()-1);  
 return retVal;  
 }  
  
 public int Size() {  
 return oneWayLinkedList.Size();  
 }  
  
 public T top() throws EmptyStackException {  
 if (isEmpty())  
 throw new EmptyStackException();  
  
 T retVal = oneWayLinkedList.get(oneWayLinkedList.Size()-1);  
 return retVal;  
 }  
}

pokazanie że działa:

class MyStackTest {  
  
 MyStack<Integer> myStack;  
  
 @BeforeEach  
 public void setUp() {  
 myStack = new MyStack<>();  
 }  
  
 @Test  
 public void isThrowingExceptions() {  
 *assertThrows*(  
 EmptyStackException.class,  
 () -> myStack.pop()  
 );  
  
 *assertThrows*(  
 EmptyStackException.class,  
 () -> myStack.top()  
 );  
 }  
  
 @Test  
 public void isPushingAndPopingWorking() throws EmptyStackException {  
 myStack.push(1);  
 myStack.push(2);  
 myStack.push(3);  
  
 *assertEquals*(myStack.top(), 3);  
 *assertEquals*(myStack.pop(), 3);  
 *assertEquals*(myStack.pop(), 2);  
 *assertEquals*(myStack.pop(), 1);  
 }  
  
 @Test  
 public void isSizeWorking() throws EmptyStackException {  
 *assertEquals*(0, myStack.Size());  
  
 myStack.push(1);  
 myStack.push(2);  
 myStack.push(3);  
 *assertEquals*(3, myStack.Size());  
  
 myStack.pop();  
 myStack.pop();  
 *assertEquals*(1, myStack.Size());  
  
 }  
  
}

Zadanie 4

public class TwoWayLinkedListWithSentinel<T> {  
  
 // węzeł  
 private static class Node<T> {  
 T Value;  
 Node<T> next;  
 Node<T> prev;  
  
 public Node(T value) {  
 this.Value = value;  
 }  
  
 public void insertAfter(Node<T> node) {  
 node.next = this.next;  
 node.prev = this;  
 this.next = node;  
 this.next.prev = node;  
 }  
  
 public void insertBefore(Node <T> node) {  
 node.next = this;  
 node.prev = this.prev;  
 this.prev = node;  
 this.prev.next = node;  
 }  
  
 public void remove() {  
 this.prev.next = this.next;  
 this.next.prev = this.prev;  
 }  
 }  
  
  
 // lista  
 private Node<T> sentinel;  
  
 public TwoWayLinkedListWithSentinel() {  
 sentinel = new Node<>(null);  
 sentinel.prev = sentinel;  
 sentinel.next = sentinel;  
 }  
  
 private Node<T> getNode(int index) {  
 Node <T> node = sentinel.next;  
 int counter = 0;  
  
 while (node != sentinel && counter < index) {  
 node = node.next;  
 counter++;  
 }  
  
 if (node == sentinel) {  
 throw new IndexOutOfBoundsException("Index: " + index);  
 }  
  
 return node;  
 }  
  
 private Node<T> getNode(T value) {  
 Node<T> actNode = sentinel.next;  
 int counter = 0;  
  
 while (actNode != sentinel && !value.equals(actNode.Value)) {  
 counter++;  
 actNode = actNode.next;  
 }  
  
 if (actNode == sentinel)  
 return null;  
 return actNode;  
 }  
  
  
  
 // metody użytkownika:  
 public boolean isEmpty() {  
 return sentinel.next == sentinel;  
 }  
  
 public void clear() {  
 sentinel.next = sentinel;  
 sentinel.prev = sentinel;  
 }  
  
 public int indexOf(T value) {  
 Node<T> node = sentinel.next;  
 int counter = 0;  
  
 while(node != sentinel && !node.Value.equals(value)) {  
 counter ++;  
 node = node.next;  
 }  
  
 if (node == sentinel)  
 return -1;  
 return size()-counter;  
 }  
  
 public boolean contains(T value) {  
 return indexOf(value) != -1;  
 }  
  
 public T get(int index) {  
 Node<T> node = getNode(index);  
 return node.Value;  
 }  
  
 public T set(int index, T value) {  
 Node<T> node = getNode(index);  
 T retVal = node.Value;  
 node.Value = value;  
 return retVal;  
 }  
  
  
 public boolean add(T value) {  
 Node<T> newNode = new Node<>(value);  
 sentinel.insertAfter(newNode);  
 return true;  
 }  
  
 public boolean add(int index, T value) {  
 Node<T> newNode = new Node<>(value);  
  
 if (index == 0) {  
 sentinel.insertAfter(newNode);  
 }  
 else {  
 Node<T> node = getNode(index);  
 node.insertAfter(newNode);  
 }  
 return true;  
 }  
  
  
  
 public T remove(int index) {  
 Node<T> toRemove = getNode(index);  
 toRemove.remove();  
 return toRemove.Value;  
 }  
  
 public boolean remove(T value) {  
 Node<T> toRemove = getNode(value);  
 if (toRemove == null)  
 return false;  
 toRemove.remove();  
 return true;  
 }  
  
 public int size() {  
 Node<T> node = sentinel.next;  
 int counter = 0;  
  
 while(node != sentinel) {  
 counter++;  
 node = node.next;  
 }  
  
 return counter;  
 }  
  
  
 // 4a  
 public boolean addNewListOnTheEnd(TwoWayLinkedListWithSentinel<T> newList) {  
 Node<T> lastListI = getNode(size());  
 Node<T> firstListII = newList.getNode(0);  
  
 lastListI.next = firstListII;  
 firstListII.prev = lastListI;  
  
  
 Node<T> firstListI = getNode(0);  
  
 firstListI.prev = newList.sentinel;  
  
 return true;  
 }  
  
  
 // 4b  
 public boolean addNewListOnTheIndex(int index, TwoWayLinkedListWithSentinel<T> newList) {  
 Node<T> first = newList.getNode(0);  
 Node<T> last = newList.getNode(newList.size());  
  
 Node<T> thisBeforeYouWantToInsert = getNode(index);  
 thisBeforeYouWantToInsert.insertBefore(last);  
  
 Node<T> thisAfterYouWantToInsert = getNode(index-1);  
 thisAfterYouWantToInsert.insertAfter(first);  
  
 return true;  
 }  
  
  
  
}