We are going to develop and implement a doubly linked list class with sentinels. In order to avoid some special cases when operating near the boundaries of a doubly linked list, it helps to add special nodes at both ends of the list: a header node at the beginning of the list, and a trailer node at the end of the list. These "dummy" nodes are known as sentinels (or guards), and they do not store elements of the primary sequence. A doubly linked list with such sentinels is shown in Figure 1.

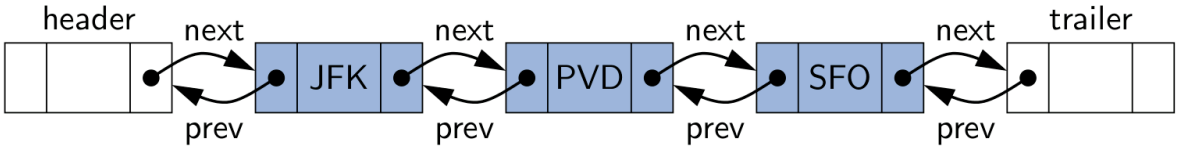

Figure 1 - A doubly linked list representing the sequence { JFK, PVD, SFO }, using sentinels header and trailer to demarcate the ends of the list.

Initiate by downloading and extracting the project PL2_DoublyLinkedList_Initial from moodle to your Projects folder. The contents of the project are:

a. One generic source outer class **DoublyLinkedList<E>**, which has one static nested class **Node<E>** (already complete) and one inner class **DoublyLinkedListIterator**.

b. Two test classes **DoublyLinkedListTest** and **DoublyLinkedListIteratorTest**.

The interface for the class **DoublyLinkedList** (which we will be developing) is the following:

```
                                    Public:
size(): Returns the number of elements in the list.
isEmpty(): Returns true if the list is empty, and false otherwise.
first(): Returns (but does not remove) the first element in the list.
last(): Returns (but does not remove) the last element in the list.
addFirst(e): Adds a new element to the front of the list.
addLast(e): Adds a new element to the end of the list.
removeFirst(): Removes and returns the first element of the list.
removeLast(): Removes and returns the last element of the list.
                                    Private:
addBetween(E e, Node<E> predecessor, Node<E> successor): Adds the element e
between nodes predecessor and successor.
E remove(Node<E> node): removes node from the linked list.
```

Laboratory Class 2

Doubly Linked Lists

Estruturas de Informação

Part 1: Development of the generic class **DoublyLinkedList<E>:**

1. Test the project and verify that all the tests fail due to non-existing implementation.
2. Implement the simple methods ***size*** and ***isEmpty***.
3. Implement both methods ***first*** and ***last***, which return the first and last element of the list respectively.
4. Implement the private method ***addBetween***, which wraps an element inside a Node and adds it between two existing nodes. Notice that since this is a list with sentinels, all additions will be between two existing nodes. Don't forget to update all the relevant variables.
5. Implement both methods **addFirst** and **addLast**, using the addBetween method previously developed.
6. Test the project. At this point two tests must have passed. If they do not, check your design/implementation.
7. Implement the private method **remove**, which discards a node from the list by updating its previous and successor nodes. Notice that since this is a list with sentinels, all removals will be between two existing nodes. Don't forget to update all the relevant variables.
8. Implement both methods ***removeFirst*** and ***removeLast***, using the previously developed *remove* method.
9. Test the project. At this point eight tests of the DoublyLinkedListTest class must have passed. If they do not, check your design/implementation.

End of Part 1.

Up until now we have developed a linked list with the ability to add/remove elements in both extremes. Nevertheless, some algorithms may require more advanced manipulation of the structure, which can be done in its most generic form using iterators. There are two generic iterator interfaces: *Iterator* and *ListIterator*. *ListIterator* extends Iterator and has the ability to add, remove and reset elements in its underlying collection. We will implement the interface *ListIterator* in our doubly linked list class. Our implementation will use an inner class called **DoublyLinkedListIterator** which implements the generic interface ListIterator (the interface is available online).

The inner class **DoublyLinkedListIterator** will use two nodes to indicate which one will be returned by a call to the next or previous nodes. It will also use an additional node to keep which (if any) was previously returned. There is an integer to save which is the expected modification count so that the iterator is invalidated in the presence of an addition or removal that did not use its interface (concurrent modification). An iterator is valid while there are no external modifications to the iterated structure. The initial project already has a method to verify the iterator's validity and it is already called in the appropriate places. Do not change the calls to *checkForComodification* in the inner class.

Part 2: Development of the generic class **DoublyLinkedListIterator:**

1. Test the project and verify that all the iterator tests fail.
2. Implement the ***hasNext*** and ***hasPrevious*** methods.
3. Implement the **next** method. It should return an element and update all the nodes. In case hasNext fails, an exception NoSuchElementException("End of list reached.") should be thrown.
4. Test the project. At this point eleven tests must have passed. If they do not, check your design/implementation.
5. Implement the **previous** method. It should return an element and update all the nodes. In case hasPrevious fails, an exception NoSuchElementException("Beginning of list reached.") should be thrown.
6. Test the project. At this point thirteen tests must have passed. If they do not, check your design/implementation.
7. Implement the **add** method using the outer class addBetween method. Notice that for an inner class to call an outer class method it should call: DoublyLinkedList.this.addBetween(…). Do not forget to update all the necessary variables.
8. At this point only one test must not have passed. Check if it is so and if not make the necessary adjustments.
9. Implement the **remove** method. Start by checking which of the nodes will be removed so that the next and previous references match after the removal. Perform the actual removal using the outer class private remove method.
10. Now all the iterator tests have passed.
11. In the iterator tests, concurrent modification has not been tested. Add the necessary code to implement iterator's validity tests.

End of Part 2

Now we can develop methods, which require the handling of a set of elements. In this part we are going to override the implementation of *equals* and transform it from a shallow comparison (only attributes) to a deep comparison (entire structure). We will also implement the Cloneable interface, performing a deep copy of the structure.

There are two ways of implementing deep structure algorithms from within the structure's class: using the internal structure or using iterators. This part was left for after the iterator had been developed so that it could be implemented in any of the two ways. Note that iteration can be used outside of the class but the usage of the internal structure cannot.

Part 3: Development of the deep methods of the generic class **DoublyLinkedList <E>**:

1. Implement the methods ***equals***. If the classes are not the same, equality fails. Make sure to perform a deep comparison of the two lists. The algorithm for equality should be easily implemented using either *Iterator* or directly *Node*. Since the algorithm is the same, implementations should be easily interchangeable.
2. Test the project. At this point the test about the *equals* method should have passed. If it does not, check your design/implementation.
3. Implement the methods ***clone*** from the Cloneable interface. This method should be public so that it can be called from the outside. Make sure to perform a deep copy of the list.
4. Test the project. At this point the test about the *clone* method should have passed. If it does not, check your design/implementation.
5. That's it. Now all the tests have passed and your data structure is finished.

End of Part 3.

Part 4: Additional enhancements to the project

Now that the project is implemented let us check if there is something, which could be improved. Suggest eventual tests, which could improve the debugging of the data structure. Discuss and implement your solutions to be able to increase the range of tests performed on the structure.