CSC 364 Assignment #1
Total points: 50 points

Based on Exercise 24.3 (*Implement a doubly-linked list*):

Download the following files from the assignment page on Canvas -

MyDoublyLinkedList.java *(this is the file that you will modify to include the additional methods)*
MyList.java
MyAbstractList.java
MyAbstractSequentialList.java
TestMyDoublyLinkedList.java

Your task is to create a public, concrete class named **MyDoublyLinkedList** that extends **MyAbstractSequentialList** and implements **Cloneable**:

```
public class MyDoublyLinkedList<E> extends MyAbstractSequentialList<E>
implements Cloneable {
```

The class must override the clone and equals methods that are inherited from the Object class. All supported methods should work just like those in **java.util.LinkedList**.

Tip1: It is not necessary to override the equals method in MyDoublyLinkedList in order to write the contains, indexOf, and lastIndexOf methods. The equals method that you will call in those methods is the one provided by the elements of the list, not the equals method provided by the list class.

Tip 2: You need to make sure that you understand how the ListIterator is expected to work for implementing **MyAbstractSequentialList.** The following is description of ListIterator from the Java Docs. Specifically, you have to note how the remove() and set() methods work. Therefore,

- If you have invoked next(), and then call the remove(), it will delete the node that is the previous of current.
- If you have invoked previous(), and then call remove(), it will delete the node that is current.
- If you have invoked next(), and then call set(), it will set the node that is the previous of current.
- If you have invoked previous(), and then call set(), it will set the node that is the is current.
- By default, you cannot call remove() or set() before invoking next() or previous().
- After you have called add() or remove() once, you cannot call remove() or set() before invoking next() or previous() again.

```
public interface ListIterator<E>
extends Iterator<E>
```

An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list. A `ListIterator` has no current element; its *cursor position* always lies between the element that would be returned by a call to `previous()` and the element that would be returned by a call to `next()`. An iterator for a list of length `n` has `n+1` possible cursor positions, as illustrated by the carets (^) below:

```
                      Element(0)   Element(1)   Element(2)   ... Element(n-1)
 cursor positions:  ^            ^            ^            ^                  ^
```

Note that the `remove()` and `set(Object)` methods are *not* defined in terms of the cursor position; they are defined to operate on the last element returned by a call to `next()` or `previous()`.

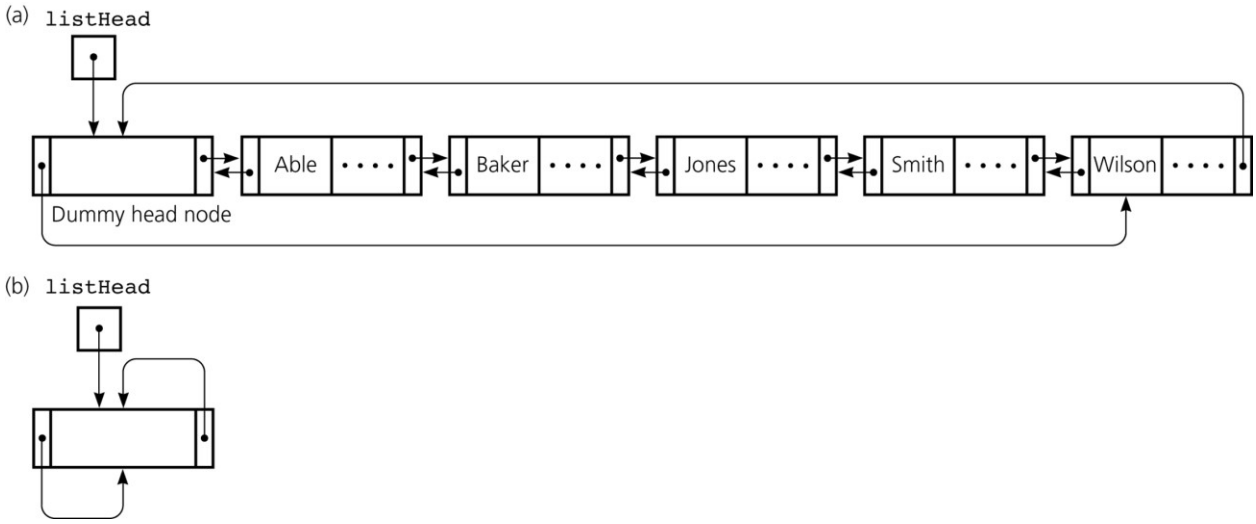
Tip 3: In the iterator's remove() method you will probably switch based on the value of iterState., which indicates if a node can be deleted, cannot be deleted, and if the previous node can be deleted or not.

- If the value of iterState is CAN_REMOVE_PREV, consider whether you need to update indexOfNext. (Hint: The index of the next element will be changed by the deletion.)
- If the value of iterState is CAN_REMOVE_CURRENT, consider whether you need to update current. (Hint: You don't want current to point to a node that is no longer in the list.)

Test your code using **TestMyDoublyLinkedList**. The sample output is shown as follows:

```
Test 1 successful
Test 2 successful
Test 3 successful
Test 4 successful
Test 5 successful
Test 6 successful
Test 7 successful
Test 8 successful
Test 9 successful
Test 10 successful
Test 11 successful
Test 12 successful
Test 13 successful
Test 14 successful
Test 15 successful
Test 16 successful
Test 17 successful
Test 18 successful
Test 19 successful
Test 20 successful
Test 21 successful
Test 22 successful
Test 23 successful
Test 24 successful
Test 25 successful
Test 26 successful
Testing clone method:
Test 27 successful
Test 28 successful
Test 29 successful
Test 30 successful
Testing equals method:
Test 31 successful
Test 32 successful
Test 33 successful
Test 34 successful
Test 35 successful
Test 36 successful
Test 37 successful
Test 38 successful
Test 39 successful
Test 40 successful
```

You are required to implement your class by using a circular doubly-linked list with a dummy head node. The following diagrams from *Data Abstraction & Problem Solving with Java* by Frank M. Carrano and Janet J. Prichard, 1st edition show this data structure:



Each node object should have three data fields: one for the element, one for the previous node, and one for the next node. Where the diagrams above use the name **`listHead`**, I will use the name `head`. Note that the first element of a non-empty list is **`head.next.element`**. The last element is **`head.prev.element`**. In your list class you will not need a separate data field that points to the tail. If you find yourself using a data field called `tail` then you are not implementing the data structure properly.

The methods **`contains`**, **`indexOf`**, and **`lastIndexOf`** should compare elements to e by using the **`equals`** method. You may need to handle a null value as a special case because the call **`e.equals(…)`** will throw a **NullPointerException** if **`e`** is null. The following description (from http://docs.oracle.com/javase/7/docs/api/java/util/List.html) of the contains method shows a good way to handle this:

boolean contains(Object e)
Returns `true` if this list contains the specified element. More formally, returns `true` if and only if this list contains at least one element `o` such that `(e == null ? o == null : e.equals(o))`.

`remove` and `set` should throw an IndexOutOfBoundsException if index < 0 or index >= size(). When set does not throw an exception, it should return the element that was previously at the given index. `add` should throw an IndexOutOfBoundsException if index < 0 or index > size().

Iterators:
You will need to write an inner class that implements the ListIterator interface. Sections 24.3-4 showed some examples of this, although those iterators did not implement the full ListIterator interface – they only implemented `hasNext` and `next`. Carefully read the following Java documentation on the ListIterator `add`, `remove`, and `set` ListIterator methods. Note that the `remove` and `set` methods need to throw an IllegalStateException in certain circumstances.

```
    /**
* Inserts the specified element into the list. The element is inserted
```

```
* immediately before the element that would be returned by next(), if
* any, and after the element that would be returned by previous(), if
* any. (If the list contains no elements, the new element becomes the
* sole element on the list.) The new element is inserted before the
* implicit cursor: a subsequent call to next would be unaffected, and a
* subsequent call to previous would return the new element. (This call
* increases by one the value that would be returned by a call to
* nextIndex or previousIndex.)
     */
    void add(E e);


    /**
* Removes from the list the last element that was returned by next()
* or previous(). This call can only be made once per call to next or
* previous. It can be made only if add(E) has not been called after the
* last call to next or previous.
* throws IllegalStateException if neither next nor previous have been
* called, or remove or add have been called after the last call to next
* or previous.
     */
    void remove();


    /**
* Replaces the last element returned by next() or previous() with the
* specified element. This call can be made only if neither remove() nor
* add(E) have been called after the last call to next or previous.
* throws IllegalStateException if neither next nor previous have been
* called, or remove or add have been called after the last call to next
* or previous.
     */
    void set(E e);
```

Note also that an iterator's `next` method should throw a NoSuchElementException if there is no next element. Likewise, a list iterator's `previous` method should throw a NoSuchElementException if there is no previous element.

clone() method:
Here is the Java documentation for the clone() method –
`public Object clone()`
Returns a shallow copy of this `LinkedList`. (The elements themselves are not cloned.)

The textbook discusses cloning in Section 13.7. The second clone method on page 516 (i.e. the one that does not have a throws declaration in the header) may be helpful in getting you started: You also do not want a throws declaration in the header of your clone method. Following the format of that example, here is one good way to accomplish your task: Inside your try block, after you have called super.clone, allocate a new Node for the dummy head. Then make the next and previous from the clone's head point back to the clone's head. Set the `size` data field of the clone to 0. Then use an iterator and a loop to iterate through this list and add every element to the clone. Finally, return the clone. In the catch block of your clone method you can just throw a RuntimeException. That catch block should never be executed. Here is the above algorithm expressed as pseudocode:

```
clone():   try
    Create the clone by calling super.clone.
    Make the clone's head point to a new dummy head node.
    Make the next and previous in the clone's dummy head     node point
back to the clone's dummy head node.
    Initially set size of the clone to 0.
    Iterate through this linked list, adding each element to     the
clone by calling the clone's add method.
    Return the clone.
  catch CloneNotSupportedException
    If you've done things correctly, this catch block      should
never execute.     But for good style throw some sort of
RuntimeException.
```

equals(Object other) method:
The equals method should return true if and only if `other` is an instance of MyList with the same size as this list and with the corresponding elements equal to the elements of this list. Here is some pseudocode for a good way to accomplish this:

```
equals(Object other):   if
this == other     return
true
  else if other is not an instance of MyList     return
false
  if other's size != this's size
return false   else
    get an iterator to this
get an iterator to other
    if corresponding elements are not equal*
return false
    return true
```

In the above algorithm, once you get past the check that ensures that other is an instance of MyList, you can typecast other to type (MyList<?>). This will enable you to call methods such as size() and iterator().

*Note that lists are allowed to have null elements. Your equals method should not throw a NullPointerException in that case. Deal with this issue like you did in the contains, indexOf, and lastIndexOf methods. I.e., you should use the == operator to compare a null reference, but use the equals method to compare actual objects.

What to turn in:
Via Canvas, submit your **MyDoublyLinkedList.java** file.