CMPSCI 187 (Fall 2018) Lab 07: Deque

This lab reviews some topics on the midterm. To work on the assignment:

- Go to ***File -> Make a Copy*** to make an editable copy of this Google Doc for your account
- Follow the instructions to complete the assignment
- When you are done, go to ***File -> Download As -> PDF Document***
- Log in to Gradescope and submit your PDF

Background

A queue is a collection that supports adding at the rear and removing from the front (i.e. enqueue(), and dequeue()). Here we define a related data structure called a deque for "double-ended queue." In a deque we can add, remove, or look at elements at either end (front and back). Its methods are:

```
public interface Deque<T> {
        public void addToFront(T element);
        public T removeFront() throws NoSuchElementException;
        public T first() throws NoSuchElementException;
        public void addToRear(T element);
        public T removeRear() throws NoSuchElementException;
        public T last()  throws NoSuchElementException;
        public boolean isEmpty();
}
```

Like queues, deques are typically implemented with a linked list. Notice that in order to efficiently operate on both the front and the back of the queue, we need to keep references to both the head and the tail of our linked list. However, we need to be able to do more than simply add to the front and the back; we must also be able to remove. Note that for the back we need to be able to get the second to last node in order to disconnect the last node from the list. In order to do this, we'll be using doubly-linked lists, so that the list can be explored in either direction.

Doubly-linked lists are pretty similar to singly-linked lists. However, they have a link to both the next node and the previous node. So when adding and removing from the list, you must make sure to update both of the links.

Section A: Debugging

The methods dealing with the front of the deque have been given to you. Most of these methods are correct, but one of them has a bug. Run the tests, use the debugger, and look at the code to find this bug.

Which method (out of addToFront, removeFront, first) had the bug in it? [1pts]

removeFront

Briefly explain what the bug was. [1pts]

head is not set to head.next after removing the element.

Section B: Methods

Now fill in the methods referring to the end of the deque (addToRear, removeRear, and last). [2 pts]

```
public void addToRear(T element) {
   tail = new DLNode<T>(element, null, tail);
   if (tail.prev == null)
           head = tail;
   else
           tail.prev.next = tail;
}
```

(End of page 2)

```
public T removeRear() throws NoSuchElementException { // [3 pts]
   T data = last();
   if (tail.prev != null)
           tail.prev.next = null;
   else
           head = null;
   tail = tail.prev;
   return data;
}

public T last() throws NoSuchElementException { // [2 pts]
   if (isEmpty())
           throw new NoSuchElementException();
   return tail.data;
}
```

Section C: Tests
Add new JUnit test to cover these newly added methods. You don't need to create a completely exhaustive test for this lab, just add a simple test that uses these methods and makes sure you get the expected result.

```
@Test (expected = NoSuchElementException.class)
public void testRearMethods() { // [2 pts]
   assertTrue(deque.isEmpty());
   deque.addToRear("Bob");
   assertFalse(deque.isEmpty());
   assertEquals("Bob", deque.last());
   deque.addToRear("VIP Alice");
   assertEquals("VIP Alice", deque.last());
   assertEquals("VIP Alice", deque.removeRear());
   assertFalse(deque.isEmpty());
   assertEquals("Bob", deque.last());
   deque.removeRear();
   assertTrue(deque.isEmpty());
   deque.removeRear();

   /*
    * Obviously, this is nearly identical to testFrontMethods().
    * The nice thing about supporting both front and rear operations
    * is that you can just treat rear operations like front operations
    * with 'next' and 'prev' swapped and 'front' and 'rear' swapped.
    */
}
```