CMPSCI 187 Discussion #7: Double-Ended Queues
Individual Handout

David Mix Barrington
24 October 2012

Last week in lecture we introduced queues, both as presented in DJW with its three interfaces and as they exist in `java.util` with the `Queue` interface. A queue is a linear data structure that supports adding at the rear and removing from the front – the `QueueInterface` interface in DJW has methods `dequeue`, and `isEmpty`, and its extensions `BoundedQueueInterface` and `UnboundedQueueInterface` have an `enqueue` method, with or without an `isFull` method respectively.

Here we define a related linear data structure called a **deque** for "double-ended queue" – the word is pronounced like "deck" and should not be confused with the "dequeue" operation which is pronounced like the letters "DQ". In a deque we can add, remove, or look at elements at either end – its methods are:

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

Only the last two questions of this pencil-and-paper discussion deal with implementation of deques – today we will mostly just be getting familiar with deques.

Question 1: Once again assume that we have the `Dog` class defined and that we have declared and created `Dog` objects named `ace`, `biscuit`, `cardie`, and `duncan`. Trace a deque (describing a queue starting from the front) through the following sequence of operations, starting with an empty `Deque<Dog>` object:

```
addToRear(cardie); addToFront(duncan); addToFront(biscuit); removeFront( ); addToRear(ace);
removeFront( ); addToRear(cardie); addToRear(duncan); removeFront( ); removeRear( );
addToFront(biscuit);
```

Question 2: Write a method `switchLastTwo` to go into a class `DequeClass` that implements `Deque`, using `Deque` methods to duplicate the functionality of Discussion #5's method with the same name in the `DogTeam` class. (It switches the last two elements if there are at least two, and otherwise does nothing.)

Question 2: Write complete code for two new generic classes: `DequeStack<T>` and `DequeQueue<T>`, each of which will extend a class `DequeClass<T>` that implements `Deque<T>`. The class `DequeStack<T>` should have the methods `push`, `pop` (as in DJW), and `top`, and the class `DequeQueue<T>` should have the methods `enqueue` and `dequeue`. Implement these methods by calling on those of `DequeClass<T>`. You may leave out the constructors (why?).

Question 4: (only if time) Do you think it will be easy to implement a deque with a circular array like the one for `ArrayBoundedQueue` and `ArrayUnboundedQueue`? Why or why not?

Question 5: (only if time) Do you think it will be easy to implement a deque with linear nodes like those we used for `LinkedStack` and `DogTeam`? Why or why not?