CmpSci 187 Discussion #7: Using Deques
Individual Handout

Marc Liberatore and John Ridgway

9 March 2015

Last Thursday we introduced the queue abstract data type. A queue is a collection that supports adding at the rear and removing from the front – the DJW interfaces have methods `enqueue()`, and `dequeue()`, along with `isEmpty()` and `isFull()` (the latter for bounded queues only).

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:

```
1  public interface Deque<T> {
2      public void addToFront (T element);
3      public T removeFront () throws DequeUnderflowException;
4      public T first () throws DequeUnderflowException;
5      public void addToRear (T element);
6      public T removeRear () throws DequeUnderflowException;
7      public T last () throws DequeUnderflowException;
8      public boolean isEmpty ();
9      public boolean isFull ();
10 }
```

We'll deal with implementing deques in our next discussion, and on Tuesday we will show how to implement queues with either circular arrays or linked data structures. Today we will be just getting familiar with deques and their operations.

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`. Draw the contents of a deque through the following sequence of operations, starting with an empty deque:

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

Question 2: Write complete code for two new classes: `DequeStack<T>` and `DequeQueue<T>`. Each `extends` a class `DequeClass<T>`. You should assume `DequeClass<T> implements Deque<T>`.

The class `DequeStack<T>` should have the methods `push()`, `pop()`, and `peek()`, and the class `DequeQueue<T>` should have the methods `enqueue()` and `dequeue()`. Implement these methods by calling methods of `DequeClass<T>`. You may leave out the constructors (do you see why?).

Question 3: Using the `Deque<T>` methods, write a method of `DequeClass<T>` named `switchLastTwo()`. This method should do nothing if the calling collection has zero or one element. If the calling collection has more than one element, the method should exchange the positions of the last two elements (those closest to the rear of the deque).

Question 4: Write a `switchLastTwo()` method for a class implementing `StackInterface<T>`. The last two elements are the two most recently added to the stack. You may use only stack operations (`push()`, `pop()`, `size()`, etc.) but *not* methods from `Deque<T>`. Can you do it in O(1) operations?

Question 5: Write a `switchLastTwo()` method for a class implementing `QueueInterface<T>`. The last two elements are the two most recently added to the queue. You may use only queue operations (`enequeue()`, `dequeue()`, `size()`, etc.) but *not* methods from `Deque<T>`. Can you do it in O(1) operations?