**CS 143 Assignment 5**
**Improving DoubleLinkedList (DLList)**
**DUE DATE: See Canvas**

Your assignment is to improve the DoubleLinkedList class we wrote in class. **Please start with the version uploaded to Canvas called `DLList` (abbreviated version shown on the back of this page *FOR REFERENCE ONLY*)** to make sure there is a common starting point, but it should be equivalent to what we write in class. **(This version includes generics and an iterator.)**

**descendingIterator method and BackwardConductor (20 points)**
Implement a new inner `BackwardConductor` class that **`implements`** `Iterator<T>`. In addition, implement the method **`public`** `Iterator<T> descendingIterator()` in DLList (a sibling to the `iterator()` method) which returns a **`new`** `BackwardConductor` object that visits the nodes from back to front. (The first call to `next()` on the returned iterator object would return the data in the last node, the second call to `next()` would return the data in the node before the last, etc.)

**size method (20 points)**
Implement a method **`public int`** `size()` that returns the number of nodes in the list in O(1) constant time. You will need to add a member field to DLList to support it. Other methods will have to change the member field as operations are performed to add or remove elements from the list.

**get method speedup (10 points)**
Improve the speed of **`public`** `T get(`**`int`** `i)` by having it work backward from the end of the list if you attempt to get an element which is closer to the end than the start. (If you get an element near the middle, the direction you choose isn't important.) This improvement will be tested through timing tests on large lists. The method should still throw an `IndexOutOfBoundsException` if `i` is not a valid index.

**reverse method (25 points)**
Implement a method **`public void`** `reverse()` that reverses the list. The list A-B-C-D would become D-C-B-A. The old start (A in the example) should be the new end and the old end should be the start. The next element after the start should be what was the previous element before the end, and so forth. The method should work for a list of any length, including the empty list and should execute in O(n) time. **This method should not create new Nodes (no use of the new keyword) or other objects; the tester will check for this. References to existing nodes, like the walker variable in the get method, are good!**

**add with index method (25 points)**
Implement an O(i) method **`public boolean`** `add(`**`int`** `i, T data)` that adds (inserts) a node containing `data` of type T at index `i`. (Note that this uses Java's overload facility since we already have an add method with just a String argument.) Return false if i is not a valid index in the list, otherwise add the element and return true. (Note that this doesn't let you add a new element to the very end but the existing add method accomplishes this.) The elements before index i should be unchanged. Your new element will be placed at index i. The elements which were already at index i and after should all be in the same order but moved down (at one index value greater than they were before the call to your add method). As with remove, your links in **both** directions must be correctly updated!

**Extra Credit** (2 points): Similar to the get method speedup, speed up add with index by working backward from the end of the list if you attempt to add an element which is closer to the end than the start, resulting in O(min(i, n-i)). This improvement will be tested through timing tests on large lists.

```java
import java.util.Iterator;
public class DLList<T> implements Iterable<T>{
  private static class Node<T> {
    public Node<T> before, after;
    public T data;
    public Node(Node<T> before, T data,
                Node<T> after) {
      this.before = before;
      this.after = after;
      this.data = data;
    }
  }
  private Node<T> first, last; // Both ends

  /**
   * Forward iterator class (conductor).
   */
  private static class Conductor<T>
              implements Iterator<T> {
    public Node<T> car; // Next node to visit
    public Conductor(DLList<T> list) {
      car = list.first; // Begin at first
    }

    public boolean hasNext() {
      return car != null; // No more to visit
    }

    public T next() {
      T data = car.data; // Remember current
      car = car.after; // Advance to next car
      return data; // Return old car data
    }
  }

  public DLList() {
    first = last = null; // Empty list
  }

  /**
   * Add data to the end (last) of the list.
   */
  public void add(T data) {
    if (last == null) {
      // One node is first and last
      first = new Node<>(null,data,null);
      last = first;
    } else { // Put new node after last
      last.after = new Node<>(last,data,null);
      last = last.after; //New one is now last
    }
  }
```

```java
  /**       IOOBE is IndexOutOfBoundsException
   * Retrieve an element from middle of list.
   * @param i Zero-based index of element
   * @return element i (throws if invalid i)
   */
  public T get(int i) {
    if (i < 0) throw new IOOBE();
    Node<T> walker = first;
    for (int j=0; walker!=null && j<i; j++) {
      // Count our way up to desired element
      walker = walker.after;
    }
    if (walker == null) throw new IOOBE();
    return walker.data;
  }

  /**
   * Get and remove element i from the list.
   * @param i Zero-based index of element
   * @return element i (throws if invalid i)
   */
  public T remove(int i) {
    if (i < 0) throw new IOOBE();
    Node<T> walker = first;
    for (int j=0; walker!=null && j<i; j++) {
      // Count our way up to desired element
      walker = walker.after;
    }
    if (walker == null) throw new IOOBE();
    if (walker.before != null)
      // Link before's after to new after
      walker.before.after = walker.after;
    else first = first.after;
    if (walker.after != null)
      // Link after's before to new before
      walker.after.before = walker.before;
    else last = last.before;
    return walker.data;
  }

  /**
   * Create a forward iterator for this list.
   */
  public Iterator<T> iterator() {
    // The Conductor object can walk this list
    // forward, front to back. Each time
    // .next() is called, the Conductor
    // produces one more piece of data,
    // starting at first and ending with last
    return new Conductor<T>(this);
  }
}
```