```java
package proj2;  // Gradescope needs this.
/**
 * This is the sequence class it stores strings
 * @author Matthew Caulfield
 * @version 9/29/17
 * I affirm that I have carried out the attached academic endeavors with full
 * academic honesty, in accordance with the Union College Honor Code and the course
 * syllabus.
 */
public class Sequence
{
	private final int DEFAULT_CAPACITY = 10;

	private String [] data;
	private int numItems;
	private int currentIndex;
	private int capacity;
    /**
     * Creates a new sequence with initial capacity 10.
     */
    public Sequence() {
	data = new String[DEFAULT_CAPACITY];
	for(int i = 0; i < DEFAULT_CAPACITY; i++) {
		data[i] = "";
	}
	numItems = 0;
	currentIndex = numItems;
	capacity = DEFAULT_CAPACITY;

    }


    /**
     * Creates a new sequence.
     *
     * @param initialCapacity the initial capacity of the sequence.
     */
    public Sequence(int initialCapacity){
	data = new String[initialCapacity];
	for(int i = 0; i < initialCapacity; i++) {
		data[i] = "";
	}
	numItems = 0;
	currentIndex = numItems;
	capacity = initialCapacity;
    }


    /**
     * Adds a string to the sequence in the location before the
     * current element. If the sequence has no current element, the
     * string is added to the beginning of the sequence.
     *
     * The added element becomes the current element.
     *
```

```
 * If the sequences's capacity has been reached, the sequence will
 * expand to twice its current capacity plus 1.
 *
 * @param value the string to add.
 */
public void addBefore(String value)
{
   Sequence copy = this.clone();
   if (this.numItems + 1 > this.capacity) {
          this.doubleCapacity();
   }
   if(this.currentIndex == this.numItems) {
          this.currentIndex = 0;
   }
   for(int i = 0; i < currentIndex; i++){
          this.data[i] = copy.data[i];
   }
   this.data[this.currentIndex] = value;
   this.numItems++;
   for(int i = this.currentIndex + 1; i < this.numItems; i++) {
          this.data[i] = copy.data[i-1];
   }

}


/**
 * Adds a string to the sequence in the location after the current
 * element. If the sequence has no current element, the string is
 * added to the end of the sequence.
 *
 * The added element becomes the current element.
 *
 * If the sequences's capacity has been reached, the sequence will
 * expand to twice its current capacity plus 1.
 *
 * @param value the string to add.
 */
public void addAfter(String value)
{
   Sequence copy = this.clone();
   if (this.numItems + 1 > this.capacity) {
          this.doubleCapacity();
   }
   if(this.currentIndex == this.numItems) {
          this.currentIndex = this.numItems - 1;
   }
   for(int i = 0; i <= currentIndex; i++){
          this.data[i] = copy.data[i];
   }
   this.currentIndex++;
   this.data[this.currentIndex] = value;
   this.numItems++;
   for(int i = this.currentIndex + 1; i < this.numItems; i++) {
          this.data[i] = copy.data[i-1];
```

```
   }
}

/**
 * copies the data array
 * @param start index of array to start copy
 * @param end 1 + index of array to end copy
 * @return copied string[]
 */
private String[] cloneData(int start, int end) {
   String[] clone = new String[capacity];
   for(int i = start; i < end; i++) {
          clone[i] = data[i];
   }
   return clone;
}

/**
 * doubles and adds one to the capacity of a sequence
 *
 */
private void doubleCapacity() {
   capacity = 2*capacity+1;
          data = new String [capacity];
          for(int i = 0; i < capacity; i++) {
          data[i] = "";
   }
}

/**
 * @return true if and only if the sequence has a current element.
 */
public boolean isCurrent()
{
   return(currentIndex < numItems);
}


/**
 * @return the capacity of the sequence.
 */
public int getCapacity()
{
   return capacity;
}


/**
 * @return the element at the current location in the sequence, or
 * null if there is no current element.
 */
public String getCurrent()
{
   if(currentIndex != numItems) {
          return(data[currentIndex]);
```

```
    }
    else {
          return null;
    }


}


/**
 * Increase the sequence's capacity to be
 * at least minCapacity.  Does nothing
 * if current capacity is already >= minCapacity.
 *
 * @param minCapacity the minimum capacity that the sequence
 * should now have.
 */
public void ensureCapacity(int minCapacity)
{
    if(capacity < minCapacity) {
          capacity = minCapacity;
          Sequence copy = this.clone();
          data = new String[capacity];
          for(int i = 0; i<numItems; i++) {
                data[i] = copy.data[i];
          }
    }
}


/**
 * Places the contents of another sequence at the end of this sequence.
 *
 * If adding all elements of the other sequence would exceed the
 * capacity of this sequence, the capacity is changed to make (just enough) room
 * for all of the elements to be added.
 *
 * Postcondition: NO SIDE EFFECTS!  the other sequence should be left
 * unchanged.  The current element of both sequences should remain
 * where they are. (When this method ends, the current element
 * should refer to the same element that it did at the time this method
 * started.)
 *
 * @param another the sequence whose contents should be added.
 */
public void addAll(Sequence another)
{
    int totalItems = this.numItems + another.numItems;
    this.ensureCapacity(totalItems);
    Sequence anotherCopy =  another.clone();
    for(int i = this.numItems; i < totalItems; i++) {
          this.data[i] = anotherCopy.data[i-this.numItems];
    }
    if (this.currentIndex == this.numItems) {
          this.currentIndex = totalItems;
```

```
    }
    this.numItems = totalItems;

}


/**
 * Move forward in the sequence so that the current element is now
 * the next element in the sequence.
 *
 * If the current element was already the end of the sequence,
 * then advancing causes there to be no current element.
 *
 * If there is no current element to begin with, do nothing.
 */
public void advance()
{
    if(isCurrent()) {
          currentIndex++;
    }
}


/**
 * Make a copy of this sequence.  Subsequence changes to the copy
 * do not affect the current sequence, and vice versa.
 *
 * Postcondition: NO SIDE EFFECTS!  This sequence's current
 * element should remain unchanged.  The clone's current
 * element will correspond to the same place as in the original.
 *
 * @return the copy of this sequence.
 */
public Sequence clone()
{
    Sequence duplicate = new Sequence(this.getCapacity());
    duplicate.numItems = this.numItems;
    duplicate.currentIndex = this.currentIndex;
    duplicate.data = this.cloneData(0,  this.numItems);
    return(duplicate);
}


/**
 * Remove the current element from this sequence.  The following
 * element, if there was one, becomes the current element.  If
 * there was no following element (current was at the end of the
 * sequence), the sequence now has no current element.
 *
 * If there is no current element, does nothing.
 */
public void removeCurrent()
{
    Sequence copy = this.clone();
    for(int i = 0; i<this.currentIndex; i++) {
```

```
            this.data[i] = copy.data[i];
        }
        if(this.currentIndex < this.numItems) {
            this.numItems = this.numItems-1;
        }
        for(int i = this.currentIndex; i<this.numItems;  i++) {
            this.data[i] = copy.data[i+1];
        }

    }


    /**
     * @return the number of elements stored in the sequence.
     */
    public int size()
    {
        return numItems;
    }


    /**
     * Sets the current element to the start of the sequence.  If the
     * sequence is empty, the sequence has no current element.
     */
    public void start()
    {
        currentIndex = 0;
    }


    /**
     * Reduce the current capacity to its actual size, so that it has
     * capacity to store only the elements currently stored.
     */
    public void trimToSize()
    {
        Sequence copy = this.clone();
        this.capacity = this.numItems;
        this.data = new String[this.capacity];
        this.data = copy.cloneData(0, this.capacity);
    }


    /**
     * Produce a string representation of this sequence.  The current
     * location is indicated by a >.  For example, a sequence with "A"
     * followed by "B", where "B" is the current element, and the
     * capacity is 5, would print as:
     *
     *    {A, >B} (capacity = 5)
     *
     * The string you create should be formatted like the above example,
     * with a comma following each element, no comma following the
     * last element, and all on a single line.  An empty sequence
```

```
 * should give back "{}" followed by its capacity.
 *
 * @return a string representation of this sequence.
 */
public String toString()
{
   String output = "{";
   for(int i = 0; i<numItems; i++) {
          if(i>0) {
                output += ", ";
          }
          if(i == currentIndex) {
                output += ">";
          }
          output += data[i];
   }
   output += "} (capacity = " + Integer.toString(capacity)+")";
   return output;
}

/**
 * Checks whether another sequence is equal to this one.  To be
 * considered equal, the other sequence must have the same size
 * as this sequence, have the same elements, in the same
 * order, and with the same element marked
 * current.  The capacity can differ.
 *
 * Postcondition: NO SIDE EFFECTS!  this sequence and the
 * other sequence should remain unchanged, including the
 * current element.
 *
 * @param other the other Sequence with which to compare
 * @return true iff the other sequence is equal to this one.
 */
public boolean equals(Sequence other)
{
   if(this.numItems != other.numItems) {
          return false;
   }
   else if(this.currentIndex != other.currentIndex) {
          return false;
   }
   else {
          for(int i = 0; i < this.numItems; i++) {
                if (this.data[i] != other.data[i]) {
                      return false;
                }
          }
          return true;
   }

}


/**
```

```
     *
     * @return true if Sequence empty, else false
     */
    public boolean isEmpty()
    {
       return(numItems == 0);
    }


    /**
     *  empty the sequence.  There should be no current element.
     */
    public void clear()
    {
       for(int i = 0; i<numItems; i++) {
              data[i] = null;
       }
       numItems = 0;
       currentIndex = 0;
    }
}
```

```
import junit.framework.TestCase;
import proj2.Sequence;
import org.junit.*;
import static org.junit.Assert.*;
/**
 * JUnit tests for sequence
 */
public class JUnitSequenceTest extends TestCase {

       /**
        * makes a sequence with strings from a given array of strings
        * @param sList is an array of strings to be stored in the sequence
        * @param index is desired pointer position
        * @return the sequence
        */
       private Sequence makeSequence(String[] sList, int index) {
              Sequence s = new Sequence();
              for(int i = 0; i<sList.length; i++) {
                     s.addAfter(sList[i]);
              }
              s.start();
              for(int i=0; i<index; i++) {
                     s.advance();
              }
              return s;
       }

       /**
        * makes a sequence with strings from a given array of strings
        * with a non default capacity
        * @param sList is an array of strings to be stored in the sequence
        * @param index is desired pointer position
        * @param cap is the desired capacity of the sequence
```

```
	 * @return the sequence
	 */
	private Sequence makeSequenceWithCapacity(String[] sList, int index, int cap)
{
		Sequence s = new Sequence(cap);
		for(int i = 0; i<sList.length; i++) {
			s.addAfter(sList[i]);
		}
		s.start();
		for(int i=0; i<index; i++) {
			s.advance();
		}
		return s;
	}


	//@test tests construction of the sequence and the to string of sequence
	public void testConstruction() {
		String [] list = {"A", "B", "C"};
		Sequence seq = makeSequence( list , 0);
		assertEquals("{>A, B, C} (capacity = 10)", seq.toString());
		Sequence seq2 = makeSequenceWithCapacity( list, 0, 5);
		assertEquals("{>A, B, C} (capacity = 5)", seq2.toString());
		String [] list2 = {};
		Sequence seq3 = makeSequence( list2 , 0);
		assertEquals("{} (capacity = 10)", seq3.toString());
		Sequence seq4 = makeSequenceWithCapacity( list2, 0, 5);
		assertEquals("{} (capacity = 5)", seq4.toString());
		String [] list3 = {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J"};
		Sequence seq5 = makeSequence( list3 , 0);
		assertEquals("{>A, B, C, D, E, F, G, H, I, J} (capacity = 10)",
seq5.toString());
	}

	//@test tests addBefore method
	public void testAddBefore() {
		String [] list = {};
		Sequence seq = makeSequenceWithCapacity( list , 0, 4);
		seq.addBefore("D");
		assertEquals("{>D} (capacity = 4)", seq.toString());
		seq.addBefore("C");
		assertEquals("{>C, D} (capacity = 4)", seq.toString());
		seq.addBefore("B");
		assertEquals("{>B, C, D} (capacity = 4)", seq.toString());
		seq.addBefore("A");
		assertEquals("{>A, B, C, D} (capacity = 4)", seq.toString());
		seq.addBefore("Z");
		assertEquals("{>Z, A, B, C, D} (capacity = 9)", seq.toString());
	}

	//@test tests addAfter method
	public void testAddAfter() {
		String [] list = {};
		Sequence seq = makeSequenceWithCapacity( list , 0, 4);
		seq.addAfter("A");
```

```
        assertEquals("{>A} (capacity = 4)", seq.toString());
        seq.addAfter("B");
        assertEquals("{A, >B} (capacity = 4)", seq.toString());
        seq.addAfter("C");
        assertEquals("{A, B, >C} (capacity = 4)", seq.toString());
        seq.addAfter("D");
        assertEquals("{A, B, C, >D} (capacity = 4)", seq.toString());
        seq.addAfter("E");
        assertEquals("{A, B, C, D, >E} (capacity = 9)", seq.toString());
}

//@test tests isCurrent
public void testIsCurrent() {
        String [] list = {"A", "B", "C"};
        Sequence seq = makeSequence( list , 0);
        assertEquals(true, seq.isCurrent());
        for(int i = 0; i<3; i++) {
                seq.advance();
        }
        assertEquals(false, seq.isCurrent());
}

//@test tests getCapacity
public void testGetCapacity() {
        String [] list = {"A", "B", "C"};
        Sequence seq = makeSequence( list , 0);
        assertEquals(10, seq.getCapacity());
}

//@test tests getCurrent
public void testGetCurrent() {
        String [] list = {"A", "B", "C"};
        Sequence seq = makeSequence( list , 0);
        assertEquals("A", seq.getCurrent());
        seq.advance();
        assertEquals("B", seq.getCurrent());
        seq.advance();
        assertEquals("C", seq.getCurrent());
        seq.advance();
        assertEquals(null, seq.getCurrent());
}

//@test tests ensureCapacity
public void testEnsuresCapacity() {
        String [] list = {"A", "B", "C"};
        Sequence seq = makeSequence( list , 0);
        seq.ensureCapacity(11);
        assertEquals(11, seq.getCapacity());
        seq.ensureCapacity(9);
        assertEquals(11, seq.getCapacity());
        assertEquals("{>A, B, C} (capacity = 11)", seq.toString());
}

//@test tests addAll
public void testAddAll() {
```

```
        String [] listE = {};
        Sequence seqE1 = makeSequence( listE, 0);
        Sequence seqE2 = makeSequence( listE, 0);
        seqE1.addAll(seqE2);
        assertEquals("{} (capacity = 10)", seqE1.toString());
        assertEquals(0, seqE1.size());
        String [] list = {"A", "B", "C"};
        Sequence seq = makeSequence( list , 0);
        seq.addAll(seqE1);
        assertEquals("{>A, B, C} (capacity = 10)", seq.toString());
        assertEquals(3, seq.size());
        seqE1.addAll(seq);
        assertEquals("{>A, B, C} (capacity = 10)", seq.toString());
        String [] list2 = {"A", "B"};
        Sequence seq2 = makeSequence( list2 , 0);
        seq.addAll(seq2);
        assertEquals("{>A, B, C, A, B} (capacity = 10)", seq.toString());
        assertEquals(5, seq.size());
        Sequence seq3 = makeSequenceWithCapacity(list, 0, 3);
        seq3.addAll(seq2);
        assertEquals("{>A, B, C, A, B} (capacity = 5)", seq3.toString());
}

//@test test clone
public void testClone() {
        String [] list = {"A", "B", "C"};
        Sequence seq = makeSequence( list , 0);
        Sequence seqClone = seq.clone();
        assertEquals(seq.toString(), seqClone.toString());
        seqClone.removeCurrent();
        assertEquals("{>A, B, C} (capacity = 10)", seq.toString());
        seq.removeCurrent();
        seq.removeCurrent();
        assertEquals("{>B, C} (capacity = 10)", seqClone.toString());
}

//@test test removeCurrent
public void testRemoveCurrent() {
        String [] list = {};
        Sequence seq = makeSequence(list , 0);
        seq.removeCurrent();
        assertEquals("{} (capacity = 10)", seq.toString());
        String [] list2 = {"A", "B", "C"};
        Sequence seq2 = makeSequence(list2, 3);
        seq2.removeCurrent();
        assertEquals("{A, B, C} (capacity = 10)", seq2.toString());
        seq2.addAfter("D");
        seq2.start();
        seq2.removeCurrent();
        assertEquals("{>B, C, D} (capacity = 10)", seq2.toString());
        seq2.advance();
        seq2.removeCurrent();
        assertEquals("{B, >D} (capacity = 10)", seq2.toString());
        seq2.removeCurrent();
        assertEquals("{B} (capacity = 10)", seq2.toString());
```

```
}

//@test test trimToSize
public void testTrimToSize() {
	String [] list = {"A", "B", "C"};
	Sequence seq = makeSequence(list , 0);
	seq.trimToSize();
	assertEquals("{>A, B, C} (capacity = 3)", seq.toString());
	seq.trimToSize();
	assertEquals("{>A, B, C} (capacity = 3)", seq.toString());
	String [] listE = {};
	Sequence seqE = makeSequence(listE, 0);
	seqE.trimToSize();
	assertEquals("{} (capacity = 0)", seqE.toString());
}

//@test test equals
public void testEquals() {
	String [] list = {"A", "B", "C"};
	Sequence seq = makeSequence(list , 0);
	Sequence seq2 = makeSequenceWithCapacity(list , 0, 4);
	assertEquals(true, seq.getCapacity()!=seq2.getCapacity());
	assertEquals(true, seq.equals(seq2));
	assertEquals(true, seq2.equals(seq));
	String [] listE = {};
	Sequence seqE1 = makeSequence( listE, 0);
	Sequence seqE2 = makeSequence( listE, 0);
	assertEquals(true, seqE1.equals(seqE2));
	assertEquals(false, seq.equals(seqE2));
	assertEquals(true, seq.equals(seq));
	seq2.advance();
	assertEquals(false, seq.equals(seq2));
	seq.advance();
	seq.advance();
	seq.advance();
	seq2.advance();
	seq2.advance();
	assertEquals(true, seq.equals(seq2));
}

//@test test isEmpty
public void testIsEmpty() {
	String [] list = {"A", "B", "C"};
	Sequence seq = makeSequence(list , 0);
	assertEquals(false, seq.isEmpty());
	String [] listE = {};
	Sequence seqE = makeSequence(listE, 0);
	assertEquals(true, seqE.isEmpty());
}

//@test clear
public void testClear() {
	String [] list = {"A", "B", "C"};
	Sequence seq = makeSequence(list , 0);
	seq.clear();
```

```
            assertEquals(true, seq.isEmpty());
            assertEquals(false, seq.isCurrent());
        }

    }
```