```java
import java.lang.reflect.Array;

/**
 *  Sequence is an abstract data type that acts as a disk storing strings.
 *  You can advance position and add strings to before and after the current
 *  position.
 *
 *  Current will never be less than 0 unless it does not exist, in which case it
will be -1
 *
 *  There will never be empty spaces between elements in the Sequence.
 *
 *  @author xavier
 */
public class Sequence
{

      private int current;

      private String[] seq;


    public Sequence() {
      seq = new String[10];
      current=0;
    }


    /**
     * Creates a new sequence.
     *
     * @param initialCapacity the initial capacity of the sequence.
     */
    public Sequence(int initialCapacity){
      seq = new String[initialCapacity];
      current=0;

    }


    /**
     * 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)
    {

      if(current>=0) {
            moveFromLeft();
      }
      else {
```

```
        current=0;
  }
  seq[current]=value;
}


/**
 * 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)
{

  if(current==0 && seq[current]==null) {
        seq[current]=value;
  }
  else if (current>=0){
        moveFromRight();
  }
  else {
        current=(this.size());
  }
  seq[current]=value;

}


/**
 * @return true if and only if the sequence has a current element.
 */
public boolean isCurrent()
{
  if(current !=-1 && seq[current]!=null) {
        return true;
  }
  return false;
}


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


/**
 * @return the element at the current location in the sequence, or
 * null if there is no current element.
```

```
 */
public String getCurrent()
{
  if(this.isCurrent()) {
        return seq[current];
  }
  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(this.getCapacity()<minCapacity) {
        scaleTo(minCapacity);
  }

}


/**
 * 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 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)
{
  Sequence tmpSeq = another.clone();
  int maxSize=(another.size()+this.size());

  //If too small
  if(this.getCapacity()<(another.size()+this.size())) {
        scaleTo((another.size()+this.size()));
  }

  tmpSeq.start();


  for(int i=this.size();i<maxSize;i++) {

        seq[i]=tmpSeq.getCurrent();
```

```
            tmpSeq.advance();
      }
    }


    /**
     * 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(current+1==this.size() || current==-1) { //So I am not sure if by the end
of  the sequence you mean
                                                  //end of the values or end of the
capacity, or we decide.
            current=-1;                     //So I have decided that as part of my
invariant current can never be on a null

      }
      else {
            current++;
      }

    }


    /**
     * 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*/
    {
      Sequence newSeq=new Sequence(this.getCapacity());

      for(int i=0;i<this.size();i++) {
            newSeq.addAfter(seq[i]);
      }

      return newSeq;
    }


    /**
     * 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()
{

  if(this.isCurrent()) {
        for(int i = current;i<this.size();i++) {
              seq[i]=seq[i+1];
        }
  }
  if(this.getCurrent()==null) {
        current=-1;
  }
}


/**
 * @return the number of elements stored in the sequence.
 */
public int size()
{
  int size=0;
  while(size<this.getCapacity() && seq[size]!=null) {
        size++;
  }
  return size;
}


/**
 * Sets the current element to the start of the sequence.  If the
 * sequence is empty, the sequence has no current element.
 */
public void start()
{
  if(seq[0]==null) {
        current=-1;
  }
  else {
        current=0;
  }
}


/**
 * Reduce the current capacity to its actual size, so that it has
 * capacity to store only the elements currently stored.
 */
public void trimToSize()
{
  scaleTo(this.size());
}


/**
 * 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 toReturn = "{";
            int tester=0;

            while (tester<this.getCapacity()-1 && seq[tester] != null) {
                  toReturn = toReturn + seq[tester];
                  if (tester+1<this.getCapacity()-1 && seq[tester+1] != null) {
                        toReturn+=", ";
                  }
                  tester++;
            }
            toReturn+="} (capacity = " + this.getCapacity() + ")";
            return toReturn;

    }

    /**
     * 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.toString().substring(0, this.toString().length()-
3).equals(other.toString().substring(0, other.toString().length()-3))) {
            return true;
      }
      return false;
    }


    /**
     *
     * @return true if Sequence empty, else false
     */
    public boolean isEmpty()
    {
      if(this.size()==0) {
```

```
            return true;
        }
        return false;
    }


    /**
     *  empty the sequence.  There should be no current element.
     */
    public void clear()
    {
        for(int i=0;i<=this.size();i++) {
            seq[i]=null;
        }
        current=-1;
    }

    //If too small, resizes to 2x+1
    private void sizeCheck() {

        if(this.getCapacity()<=this.size()+1) {
            scaleTo((Array.getLength(seq)*2)+1);
        }

    }

    private void moveFromLeft() {

        sizeCheck();

        for(int i = this.size();i>current-1;i--) {
            seq[i+1]=seq[i];

        }

    }

    private void moveFromRight() {

        sizeCheck();

        for(int i = this.size();i>current;i--) {
            seq[i+1]=seq[i];
        }
        current++;
    }


    private void scaleTo(int newSize) {
        String[] newSeq = new String[newSize];
            int runner=0;
            while(seq[runner]!=null) {
                newSeq[runner]=seq[runner];
                runner++;
            }
            seq=newSeq;
    }

}
```

```java
/**
 *
 * This is made to test all possible cases of the Sequence class by calling all
methods in different situations.
 *  @author xavier
 *
 *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 SequenceTests {

    public static void main(String[] args)
    {
      Testing.setVerbose(true); // use false for less testing output
            Testing.startTests();

      testCreate();
            testAdding();
            testIsCurrent();
            testGetCurrent();
            testEnsureCapacity();
            testAddAll_Clone();
            testRemoveCurrent();
            testTrimToSize();
            testEquals();
            testIsEmpty();
            testClear();

      // add calls to more test methods here.
      // each of the test methods should be
      // a private static method that tests
      // one method in Sequence.

      Testing.finishTests();

    }

      private static void testCreate()
      {
            Testing.testSection("Creation tests and toString of empty sequence");

            Sequence s1 = new Sequence();
            Testing.assertEquals("Default constructor", "{} (capacity = 10)",
s1.toString());
            Testing.assertEquals("Default constructor, initial size", 0,
s1.size());

            Sequence s2 = new Sequence(20);
            Testing.assertEquals("Non-default constructor", "{} (capacity = 20)",
s2.toString());
            Testing.assertEquals("Non-default constructor, initial size", 0,
s2.size());
      }


      private static void testAdding() {
            Testing.testSection("Tests addBefore");
```

```
            Sequence s1 = new Sequence();
            Sequence s2 = new Sequence();
            s1.addBefore("one");
            Testing.assertEquals("Tests if added before works", "{one} (capacity =
10)", s1.toString());

            s1.addAfter("two");
            Testing.assertEquals("Tests if added after works", "{one, two}
(capacity = 10)", s1.toString());
            Testing.assertEquals("Tests if added keeps current", "two",
s1.getCurrent());

            s1.addBefore("three");
            Testing.assertEquals("Tests if added after works", "{one, three, two}
(capacity = 10)", s1.toString());
            Testing.assertEquals("Tests if added keeps current", "three",
s1.getCurrent());

            s1.start();
            s1.addAfter("four");
            Testing.assertEquals("Tests if added after works with other parts",
"{one, four, three, two} (capacity = 10)", s1.toString());
            Testing.assertEquals("Tests if added keeps current", "four",
s1.getCurrent());

            Testing.assertEquals("Tests if size is correct", 4, s1.size());


            s2.addAfter("test");
            Testing.assertEquals("Tests if added after works", "{test} (capacity =
10)", s2.toString());


      }


      private static void testIsCurrent() {
            Testing.testSection("Tests isCurrent");

            Sequence s1 = new Sequence();
            Testing.assertEquals("Tests if current exists, doesn't", false,
s1.isCurrent());

            s1.addAfter("tmp");
            Testing.assertEquals("Check if current exists, does", true,
s1.isCurrent());


      }

      private static void testGetCurrent() {
            Testing.testSection("Tests getCurrent");

            Sequence s1 = new Sequence();
            Testing.assertEquals("Gets current value when it doesn't exist", null,
s1.getCurrent());

            s1.addAfter("tmp");
```

```
            Testing.assertEquals("Gets current value", "tmp", s1.getCurrent());

      }

      private static void testEnsureCapacity() {
            Testing.testSection("Tests ensureCapacity");

            Sequence s1 = new Sequence();
            s1.ensureCapacity(5);
            Testing.assertEquals("Tests ensureCapacity when value is less than
current", 10, s1.getCapacity());

            s1.ensureCapacity(15);
            Testing.assertEquals("Tests ensureCapacity when value is less than
current", 15, s1.getCapacity());


      }

      private static void testAddAll_Clone() {
            Testing.testSection("Tests addAll and Clone");

            Sequence s1 = new Sequence(3);
            Sequence s2 = new Sequence(3);

            s1.addBefore("one");
            s1.addBefore("two");
            s1.addBefore("three");
            s2=s1.clone();


            s2=s1.clone();
            System.out.println("s2" + s2.toString());
            Testing.assertEquals("Tests clone", s2.toString(), s1.toString());

            s1.addBefore("four");

            Testing.assertEquals("Tests cloned sequence after one has been
changed", false, s1.toString().equals(s2.toString()));

            s1.addAll(s2);
            Testing.assertEquals("Tests addAll", "{four, three, two, one, three,
two} (capacity = 7)", s1.toString());

      }


      private static void testRemoveCurrent() {
            Testing.testSection("Tests removeCurrent");

            Sequence s1 = new Sequence();

            s1.removeCurrent();
            Testing.assertEquals("Tests removeCurrent with empty sequence", "{}
(capacity = 10)", s1.toString());

            s1.addAfter("one");
            s1.addAfter("two");
```

```
            s1.removeCurrent();
            Testing.assertEquals("Tests removeCurrent at end of sequence", "{one}
(capacity = 10)", s1.toString());
            Testing.assertEquals("Tests checks for current value (doesnt exist)",
null, s1.getCurrent());
            s1.removeCurrent();
            Testing.assertEquals("Tests removeCurrent", null, s1.getCurrent());

            s1.addAfter("three");
            s1.addAfter("four");
            s1.start();
            System.out.println(s1.toString());
            s1.removeCurrent();
            Testing.assertEquals("Tests removeCurrent with values after it",
"{three, four} (capacity = 10)", s1.toString());
            Testing.assertEquals("Tests checks for current value", "three",
s1.getCurrent());

      }


      private static void testTrimToSize() {
            Testing.testSection("Tests trimToSize");

            Sequence s1 = new Sequence();

            s1.addAfter("one");
            s1.addBefore("two");
            s1.trimToSize();
            Testing.assertEquals("Tests trim to size of 2", 2, s1.getCapacity());


      }

      private static void testEquals() {
            Testing.testSection("Tests equals");

            Sequence s1 = new Sequence();
            Sequence s2 = new Sequence();

            s1.addAfter("tmp");
            s1.addBefore("first");
            s2 = s1.clone();
            Testing.assertEquals("Tests equals, should be true", true,
s1.equals(s2));

            s1.addAfter("fred");
            Testing.assertEquals("Tests equals, should be false", false,
s1.equals(s2));




      }

      private static void testIsEmpty() {
            Testing.testSection("Tests isEmpty");

            Sequence s1 = new Sequence();
```

```
            Testing.assertEquals("Tests if empty Sequence is empty", true,
s1.isEmpty());

            s1.addAfter("tmp");
            Testing.assertEquals("Tests if non-empty Sequence is empty", false,
s1.isEmpty());

      }

      private static void testClear() {
            Testing.testSection("Tests clear");

            Sequence s1 = new Sequence();
            s1.addAfter("tmp");


      }
}
```