```java
package proj2;  // Gradescope needs this.
/**
 *
 * @author Emma Vu
 * @version 4/19/2020
 * 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
 *
 *CLASS INVARIANTS (From the bottom page 150 in textbook):
 *
 * Our suggested design for the sequence ADT has three private instance
variables.
 * The first variable, data, is an array that stores the elements of the
sequence. Just
 * like the bag, data is a partially filled array, and a second instance
variable,
 * called manyItems, keeps track of how much of the data array is currently
 * being used. Therefore, the used part of the array extends from data[0] to
 * data[manyItems-1]. The third instance variable, currentIndex, gives the
 * index of the current element in the array (if there is one). Sometimes a
sequence
 * has no current element, in which case currentIndex will be set to the same
 * number as manyItems (since this is larger than any valid index). The
complete
 * invariant of our ADT is stated as three rules:
 *
 * 1. The number of elements in the sequence is stored in the instance
variable
 * manyItems.
 *
 * 2. For an empty sequence (with no elements), we do not care what is stored
 * in any of data; for a nonempty sequence, the elements of the sequence
 * are stored from the front to the end in data[0] to data[manyItems-1],
 * and we don’t care what is stored in the rest of data.
 *
 * 3. If there is a current element, then it lies in data[currentIndex]; if
there
 * is no current element, then currentIndex equals manyItems.
 *
 * if currentIndex < 0 or currentIndex >= manyItems, then the sequence doesn'
t have current element.
 * By default, we indicate that currentIndex = manyItems means there is no
current element
 *
 */
public class Sequence
{
    /**
     * Creates a new sequence with initial capacity 10.
     */
    private final int INITIAL_CAPACITY = 10;
```

```
private String[] data;
private int manyItems;
private int currentIndex;

//default constructor
public Sequence() {
    // capacity is reflected in the length of the
    // internal array
    data = new String[INITIAL_CAPACITY];
    manyItems = 0;
    currentIndex = 0;


}


/**
 * Creates a new sequence.
 *
 * @param initialCapacity the initial capacity of the sequence.
 */

//non-default constructor
public Sequence(int initialCapacity){
    // capacity is reflected in the length of the
    // internal

    //Make sure initialCapacity is valid (non-negative integer)
    if (initialCapacity < 0){
        initialCapacity = INITIAL_CAPACITY;
    }
    data = new String[initialCapacity];
    manyItems = 0;
    currentIndex = 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)
{

    expandNotEnoughCapTwicePlusOne();
```

```
        if(isCurrent()){

            addToIndex(value,currentIndex);
        }
        else{
            addToIndex(value,0);
            currentIndex = 0;
        }


    }


    /**
     * 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)

    {

        expandNotEnoughCapTwicePlusOne();
        if(isCurrent()){
            currentIndex += 1;
            addToIndex(value, currentIndex);

        }
        else{


            addToIndex(value,size());

            //after the addToIndex method, the size of sequence + 1
            // so the currentIndex = the old size before adding value which
is at the end of the sequence

            currentIndex = size() - 1;
        }


    }


    /**
     * @return true if and only if the sequence has a current element.
```

```
 */
public boolean isCurrent()
{
    if(currentIndex >= size()){
        return false;
    }
    else{
        return true;
    }
}


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

}


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

    }
}


/**
 * 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(getCapacity() < minCapacity){
        String[] increaseCapacity = new String[minCapacity];
        copyArray(data,increaseCapacity,0,getCapacity(),0);

        data = increaseCapacity;
```

```
        }
    }


    /**
     * 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 bothSize = this.size() + another.size();
        if(!isCurrent()){
            currentIndex = bothSize;
        }
        if(bothSize > this.getCapacity()){
            ensureCapacity(bothSize);
        }
        copyArray(another.data,data,0,another.size(),size());


        this.manyItems = bothSize;

    }


    /**
     * 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 += 1;
        }
```

```
}


/**
 * 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 cloneSq = new Sequence();

    cloneSq.currentIndex = this.currentIndex;


    cloneSq.manyItems = size();
    String[] cloneArray = new String[getCapacity()];
    copyArray(data,cloneArray,0,getCapacity(),0);

    cloneSq.data = cloneArray;
    return cloneSq;



}


/**
 * 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(isCurrent()) {
    String[] remove = new String[getCapacity()];
    copyArray(data, remove, 0, currentIndex, 0);
    copyArray(data, remove, currentIndex + 1, size(), currentIndex);


    data = remove;
    manyItems -= 1;


}
}
```

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


/**
 * 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()
{
    if(getCapacity() > size()){
        String[] trim = new String[size()];
        copyArray(data,trim,0,size(),0);

        data = trim;
    }
}


/**
 * 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 printToString = "{} (capacity = " + getCapacity() + ")";

    if(!isEmpty()){
        printToString = toStringOfNonEmpty();
    }

    return printToString;
}

/**
 * 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)
{
    return this.size() == other.size() &&
            this.currentIndex == other.currentIndex &&
            compareTo(other);

}


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

        return false;

    }
}


/**
 *  empty the sequence.  There should be no current element.
 */
public void clear()
```

```
    {


        manyItems = 0;

        currentIndex = size();


    }

    /**
     * Take in two arrays, copy a certain part of old array to the new one
given the certain range from the old array
     * to copy and the starting index in the new array to put the copied
elements consecutively to the new array
     *
     * PRECONDITION: The size of the copied elements from the old array
should not exceed the capacity left in the new
     * array starting from the index in the new array that wants to copy to
     * POSTCONDITION: The old array should not be changed, now the new array
has a certain part of the old array
     *
     * @param oldArrayCopyFrom the array that is copied from
     * @param newArrayCopyTo the new array that is copied to
     * @param startIndexOld the starting index in the old array to copy from
     * @param endIndexOld the ending index in the old array to copy from
     * @param startIndexNew the starting index in the new array that want to
put the copied element in
     */


    private static void copyArray(String[] oldArrayCopyFrom, String[]
newArrayCopyTo,
                                  int startIndexOld, int endIndexOld, int
startIndexNew ){
        for (int i = 0; i < endIndexOld - startIndexOld; i++){
            newArrayCopyTo[startIndexNew + i] = oldArrayCopyFrom[
startIndexOld + i];
        }

    }

    /**
     * compare if two sequences have the same elements and order
     * by going through each element in each sequence correspondingly
     * @param other other sequence to compare to
     * @return true if they both have the same elements and order; else false
     */


    private boolean compareTo(Sequence other){
        boolean compare = true;
        for(int i = 0; i < size(); i++){
```

```
            if(!this.data[i].equals(other.data[i])){
                compare = false;
            }
        }
        return compare;
    }

    /**
     * add a certain value to a specific index in the data array.
     * First create a new array and then copy the first part from the data
array
     * (before the certain index) to the new array
     *
     * Then assign the value to the certain index in the new array
     * After that copy the rest (after the certain index till the end of the
data array) to the new array
     * Then assign data array have the same contents as the new array (one
more element added at a certain index)
     * @param value the value to add to
     * @param index the index that has the given value added
     */

    private void addToIndex(String value, int index){
        if(index >= 0) {
            String[] toAdd = new String[getCapacity()];

            copyArray(data, toAdd, 0, index, 0);
            toAdd[index] = value;
            copyArray(data, toAdd, index, size(), index+1);

            data = toAdd;
            manyItems += 1;

        }

    }

    /**
     * a private helper function used for addBefore and addAfter when the
original capacity is not enough
     * so after adding one element the capacity expands twice its original
capacity plus one
     */



    private void expandNotEnoughCapTwicePlusOne(){
        if(size() + 1 > getCapacity()){
            int newCapacity = 2*size() + 1;
            ensureCapacity(newCapacity);
        }
    }
```

```
    /**
     * Private helper method to print toString of a non empty sequence
     * 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.
     *
     * @return
     */

    private String toStringOfNonEmpty(){
        String nonEmptyString = "{";
        for(int i = 0; i < size(); i++){
            if(i == 0) {
                if (i == currentIndex) {
                    nonEmptyString = nonEmptyString + ">" + getCurrent();
                }
                else {
                    nonEmptyString = nonEmptyString + data[i];
                }
            }
            else{
                if(i == currentIndex){
                    nonEmptyString = nonEmptyString + ", >" + getCurrent();

                }
                else{
                    nonEmptyString = nonEmptyString + ", " + data[i];
                }
            }

        }
        nonEmptyString = nonEmptyString + "} (capacity = " + getCapacity() +
")";
        return nonEmptyString;
    }




}
```

```java
/**
 * JUnit test class.  Use these tests as models for your own.
 */
import org.junit.*;


import org.junit.rules.Timeout;

import static org.junit.Assert.*;

import proj2.Sequence;




public class SequenceTest {

    @Rule
    // a test will fail if it takes longer than 1/10 of a second to run
    public Timeout timeout = Timeout.millis(100);

    /**
     * customize the sequence from the string array and be able to set the
capacity and current index
     * @param newArray the string array given so that convert to sequence
     * @param capacity the capacity wanted in the sequence
     * @param currentIndex the current index for the element in the sequence
     * @return the customized sequence
     */

    private Sequence createSequence(String[] newArray, int capacity, int
currentIndex){
        Sequence sq = new Sequence(capacity);
        for(String element:newArray){
            sq.addAfter(element);
        }
        sq.start();
        for(int i = 0; i < currentIndex; i++){
            sq.advance();
        }
        return sq;

    }


    @Test
    //Test  default constructor
    public void test_DefaultConstructor_EmptySequence(){
        Sequence sq = new Sequence();
        String toStringExpected = "{} (capacity = 10)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals(10,sq.getCapacity());
        assertNull(sq.getCurrent());
```

```
        assertEquals(0,sq.size());

    }


    @Test
    //Test non-default constructor with positive initial capacity
    public void test_NonDefaultConstructor_EmptySequence_PositiveCap(){
        Sequence sq = new Sequence(22);
        String toStringExpected = "{} (capacity = 22)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals(22,sq.getCapacity());
        assertNull(sq.getCurrent());
        assertEquals(0,sq.size());

    }

    @Test
    //Test non-default constructor with negative initial capacity. Then the
capacity set to 10
    public void test_NonDefaultConstructor_EmptySequence_NegativeCap(){
        Sequence sq = new Sequence(-3);
        String toStringExpected = "{} (capacity = 10)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals(10,sq.getCapacity());
        assertNull(sq.getCurrent());
        assertEquals(0,sq.size());

    }


    @Test
    //Test addBefore to empty sequence.
    // Then the element added is the first element in the sequence and also
the current element
    public void test_AddBefore_EmptySequence(){
        Sequence sq = new Sequence();


        sq.addBefore("A");

        String toStringExpected = "{>A} (capacity = 10)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals("A",sq.getCurrent());
        assertEquals(10,sq.getCapacity());
        assertEquals(1,sq.size());

    }

    @Test
    //Test addBefore to a full sequence.
    //Then the sequence's current capacity will expand twice plus 1
    //The size should +1, the added element become the current element
```

```
    public void test_AddBefore_FullSequence(){
        Sequence sq = createSequence(new String[]{"A","B","C"},3,3);
        sq.addBefore("D");
        assertEquals(4,sq.size());
        assertEquals(7,sq.getCapacity());
        assertEquals("D",sq.getCurrent());
        String toStringExpected = "{>D, A, B, C} (capacity = 7)";
        assertEquals(toStringExpected,sq.toString());
    }

    @Test
    //Test addBefore to a sequence that has room so capacity not change. The
added element becomes the current
    public void test_AddBefore_NonEmptySequence_ThatHasRoom(){
        Sequence sq = createSequence(new String[]{"A","B","C"},8,1);
        sq.addBefore("D");
        assertEquals(4,sq.size());
        assertEquals(8,sq.getCapacity());
        assertEquals("D",sq.getCurrent());
        String toStringExpected = "{A, >D, B, C} (capacity = 8)";
        assertEquals(toStringExpected,sq.toString());

    }

    @Test

    //add an element to an almost full sequence. The size should increase by
1, capacity stay the same

    public void test_AddBefore_AlmostFull(){
        Sequence sq = createSequence(new String[]{"A", "B", "C","D","E"},6,4
);
        sq.addBefore("F");
        assertEquals(6,sq.size());
        assertEquals(6,sq.getCapacity());
        assertEquals("F",sq.getCurrent());
        String toStringExpected = "{A, B, C, D, >F, E} (capacity = 6)";
        assertEquals(toStringExpected,sq.toString());

    }

    @Test
    //test add before to a sequence with no current. Then the added element
become the current at the first
    public void test_AddBefore_NoCurrent(){

        Sequence sq = createSequence(new String[]{"A","B","C"},5,3);
        sq.addBefore("D");
        assertEquals(4,sq.size());
        assertEquals(5,sq.getCapacity());
        assertEquals("D",sq.getCurrent());
        String toStringExpected = "{>D, A, B, C} (capacity = 5)";
        assertEquals(toStringExpected,sq.toString());
```

```
    }


    @Test
    //Test addAfter to an empty sequence
    //Then the element added is the end of the sequence and also the current
element
    public void test_AddAfter_EmptySequence(){
        Sequence sq = new Sequence();
        sq.addAfter("A");
        String toStringExpected = "{>A} (capacity = 10)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals("A",sq.getCurrent());
        assertEquals(10,sq.getCapacity());
        assertEquals(1,sq.size());
    }

    @Test
    //Test addAfter to a full sequence
    //Then the sequence's current capacity is expanded twice plus 1
    public void test_AddAfter_FullSequence(){
        Sequence sq = createSequence(new String[]{"A", "B", "C", "D"},4,4);
        sq.addAfter("F");

        assertEquals("F",sq.getCurrent());
        assertEquals(9,sq.getCapacity());
        assertEquals(5,sq.size());

        String toStringExpected = "{A, B, C, D, >F} (capacity = 9)";
        assertEquals(toStringExpected, sq.toString());
    }

    @Test

    //Test add after to non empty sequence that has room. Capacity should not
 change, size should + 1

    public void test_AddAfter_NonEmptySequence_ThatHasRoom(){

        Sequence sq = createSequence(new String[]{"A", "B", "C", "D"},6,1);
        sq.addAfter("F");

        assertEquals("F",sq.getCurrent());
        assertEquals(6,sq.getCapacity());
        assertEquals(5,sq.size());

        String toStringExpected = "{A, B, >F, C, D} (capacity = 6)";
        assertEquals(toStringExpected, sq.toString());

    }

    @Test
```

```
    //Should be added at the end. The added is the current
    public void test_AddAfter_NoCurrent(){

        Sequence sq = createSequence(new String[]{"A", "B", "C", "D"},6,4);
        sq.addAfter("F");

        assertEquals("F",sq.getCurrent());
        assertEquals(6,sq.getCapacity());
        assertEquals(5,sq.size());

        String toStringExpected = "{A, B, C, D, >F} (capacity = 6)";
        assertEquals(toStringExpected, sq.toString());


    }

    @Test

    //The capacity should not change. Size + 1

    public void test_AddAfter_AlmostFull(){

        Sequence sq = createSequence(new String[]{"A", "B", "C", "D"},5,4);
        sq.addAfter("F");

        assertEquals("F",sq.getCurrent());
        assertEquals(5,sq.getCapacity());
        assertEquals(5,sq.size());

        String toStringExpected = "{A, B, C, D, >F} (capacity = 5)";
        assertEquals(toStringExpected, sq.toString());

    }

    @Test
    //return false to an empty default sequence. Return true after adding
    public void test_IsCurrent_EmptyDefault(){
        Sequence sq = new Sequence();
        assertFalse(sq.isCurrent());
        sq.addAfter("A");
        assertTrue(sq.isCurrent());


    }


    @Test
    //Test isCurrent to a sequence after advancing the last current to no
current.Return false
    public void test_IsCurrent_Advancing_LastCurrent(){
        Sequence sq = createSequence(new String[]{"A","B","C"},5,2);
        assertTrue(sq.isCurrent());
        sq.advance();
```

```
        assertFalse(sq.isCurrent());

    }



    @Test
    //Test getCapacity to a default sequence. Return 10
    public void test_GetCapacity_DefaultSequence(){
        Sequence sq = new Sequence();
        assertEquals(10,sq.getCapacity());

    }

    @Test
    //Test getCapacity to a non-default sequence. Return the capacity
    public void test_GetCapacity_NonDefaultSequence(){
        Sequence sq = new Sequence(12);
        assertEquals(12,sq.getCapacity());
    }

    @Test
    //return null if sequence is empty
    public void test_GetCurrent_Empty(){
        Sequence sq = new Sequence();
        assertNull(sq.getCurrent());

    }

    @Test
    //return null if the non empty sequence has no current
    public void test_GetCurrent_NonEmpty_NoCurrent(){
        Sequence sq = createSequence(new String[]{"A","B","C"},4,3);
        assertNull(sq.getCurrent());


    }

    @Test
    //return the element of the non empty sequence
    public void test_GetCurrent_NonEmpty_Current(){
        Sequence sq = createSequence(new String[]{"A","B","C"},4,1);
        assertEquals("B",sq.getCurrent());
    }

    @Test
    //Test ensureCapacity of a sequence that has capacity smaller than
minCapacity.
    //Increase the capacity equal to minCapacity
    //Contents don't change
    public void test_EnsureCapacity_Smaller(){
        Sequence sq = createSequence(new String[]{"A","B","C"},10,1);
        sq.ensureCapacity(1000);
```

```
        assertEquals(1000,sq.getCapacity());
        String toStringExpected = "{A, >B, C} (capacity = 1000)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals(3,sq.size());
        assertEquals("B",sq.getCurrent());

    }

    @Test
    //Test ensureCapacity of a sequence that has capacity greater than
minCapacity
    //Do nothing to the capacity
    //Contents don't change
    public void test_EnsureCapacity_Greater(){

        Sequence sq = createSequence(new String[]{"A","B","C"},1000,1);
        sq.ensureCapacity(10);
        assertEquals(1000,sq.getCapacity());
        String toStringExpected = "{A, >B, C} (capacity = 1000)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals(3,sq.size());
        assertEquals("B",sq.getCurrent());

    }

    @Test
    //Test ensureCapacity of a sequence that has capacity equal to
minCapacity
    //Do nothing to the capacity
    //Contents don't change
    public void test_EnsureCapacity_Equal(){

        Sequence sq = createSequence(new String[]{"A","B","C"},1000,1);
        sq.ensureCapacity(1000);
        assertEquals(1000,sq.getCapacity());
        String toStringExpected = "{A, >B, C} (capacity = 1000)";
        assertEquals(toStringExpected,sq.toString());
        assertEquals(3,sq.size());
        assertEquals("B",sq.getCurrent());

    }

    @Test
    //Test addAll of an empty sequence to non empty sequence.
    //If this.sequence is empty and the other is not, then return this.
sequence with contents of the other sequence
    //The size of this.sequence must include the size of the other sequence.
Capacity not change
    //BUT when call isCurrent of this.sequence it's still false, getCurrent
other sequence it's still the same element


    public void test_AddAll_Empty_To_NonEmpty_Current(){
```

```
        Sequence sq = new Sequence();
        Sequence toAddAllSq = createSequence(new String[]{"A","B"},3,0);
        sq.addAll(toAddAllSq);
        assertEquals(2,sq.size());

        assertEquals("A",toAddAllSq.getCurrent());
        assertFalse(sq.isCurrent());
        String toStringExpected = "{A, B} (capacity = 10)";
        assertEquals(toStringExpected, sq.toString());

    }

    @Test

    //Test addAll of an empty sequence to non empty sequence without current.
    //If this.sequence is empty and the other is not, then return this.
sequence with contents of the other sequence
    //The size of this.sequence must include the size of the other sequence.
Capacity not change
    //BUT when call isCurrent of this.sequence it's still false, of the other
 sequence it's still false


    public void test_AddAll_Empty_To_NonEmpty_NoCurrent(){

        Sequence sq = new Sequence();
        Sequence toAddAllSq = createSequence(new String[]{"A","B"},3,2);
        sq.addAll(toAddAllSq);
        assertEquals(2,sq.size());

        assertFalse(toAddAllSq.isCurrent());
        assertFalse(sq.isCurrent());
        String toStringExpected = "{A, B} (capacity = 10)";
        assertEquals(toStringExpected, sq.toString());


    }

    @Test
    //Test addAll of non empty sequence to empty sequence.
    //Return this.sequence with contents of the other sequence. In this case
, contents don't change
    //The size of this.sequence must include the size of the other sequence.
Capacity not change
    //BUT when call isCurrent of other.sequence it's still false, getCurrent
this.sequence it's still the same element



    public void test_AddAll_NonEmpty_Current_To_Empty(){
```

```
        Sequence toAddAllSq = new Sequence();
        Sequence sq = createSequence(new String[]{"A","B"},3,1);
        sq.addAll(toAddAllSq);
        assertEquals(2,sq.size());


        assertEquals("B",sq.getCurrent());

        String toStringExpected = "{A, >B} (capacity = 3)";
        assertEquals(toStringExpected, sq.toString());

    }

    @Test
    //Test addAll of non empty sequence to empty sequence without current.
    //Return this.sequence with contents of the other sequence. In this case
, contents don't change
    //The size of this.sequence must include the size of the other sequence.
Capacity not change
    //BUT when call isCurrent of other.sequence it's still false, of this.
sequence it's still false


    public void test_AddAll_NonEmpty_NoCurrent_To_Empty(){

        Sequence toAddAllSq = new Sequence();
        Sequence sq = createSequence(new String[]{"A","B"},3,2);
        sq.addAll(toAddAllSq);
        assertEquals(2,sq.size());


        assertFalse(sq.isCurrent());

        String toStringExpected = "{A, B} (capacity = 3)";
        assertEquals(toStringExpected, sq.toString());
    }

    @Test
    //Test addAll of both empty sequences.
    // Then print out empty contents and both sequences when call isCurrent
still false

    public void test_AddAll_BothEmpty(){
        Sequence sq = new Sequence(3);
        Sequence toAddAllSq = new Sequence();
        sq.addAll(toAddAllSq);
        assertFalse(sq.isCurrent());

        assertEquals(0,sq.size());

        assertEquals(3,sq.getCapacity());
        String expected = "{} (capacity = 3)";
        String expectedToAdd = "{} (capacity = 10)";
```

```
        assertEquals(expected,sq.toString());
        assertEquals(expectedToAdd,toAddAllSq.toString());

    }

    @Test
    //Test addAll of two filled sequences but adding all the elements of
other.sequence will result in this.sequence
    // expanding just enough room (the capacity is changed). Then the
contents are both the elements from this and
    // other sequence and the current elements are the same

    public void test_AddAll_NotEnoughCapacity(){
        Sequence sq = createSequence(new String[]{"A","B"},2,1);
        Sequence toAddAllSq = createSequence(new String[]{"C","D","E"},3,0);
        sq.addAll(toAddAllSq);
        String expected = "{A, >B, C, D, E} (capacity = 5)";
        assertEquals(expected, sq.toString());
        assertEquals(5,sq.size());
        assertEquals(5,sq.getCapacity());
        assertEquals("C",toAddAllSq.getCurrent());
        assertEquals(3,toAddAllSq.size());
        assertEquals(3,toAddAllSq.getCapacity());

    }


    @Test
    //other.sequence has repeated contents to this.sequence. But still add
all to this.sequence anyway.
    // Current elements don't change
    public void test_AddAll_Repeat_One(){
        Sequence sq = createSequence(new String[]{"A"},5,1);
        Sequence toAddAllSq = createSequence(new String[]{"A","B","C","D"},4,
3);
        sq.addAll(toAddAllSq);
        assertEquals(5,sq.size());
        assertEquals("D",toAddAllSq.getCurrent());
        String expected = "{A, A, B, C, D} (capacity = 5)";
        assertEquals(expected,sq.toString());


    }


    @Test
    //other.sequence has repeated contents to this.sequence. But still add
all to this.sequence anyway.
    // Current elements don't change
    public void test_AddAll_Repeat_Two(){
        Sequence sq = createSequence(new String[]{"A","B","C"},5,1);
        Sequence toAddAllSq = createSequence(new String[]{"A","B"},3,2);
        sq.addAll(toAddAllSq);
```

```
        assertEquals(5,sq.size());
        String expected = "{A, >B, C, A, B} (capacity = 5)";
        assertEquals(expected,sq.toString());
        assertFalse(toAddAllSq.isCurrent());

    }


    @Test
    //other.sequence has same contents to this.sequence. But still add all to
 this.sequence anyway.
    // Current elements don't change. They are different objects
    public void test_AddAll_Repeat_All(){
        Sequence sq = createSequence(new String[]{"A","B"},5,1);
        Sequence toAddAllSq = createSequence(new String[]{"A","B"},5,1);
        sq.addAll(toAddAllSq);
        assertEquals(4,sq.size());
        String expected = "{A, >B, A, B} (capacity = 5)";
        assertEquals(expected,sq.toString());
        assertEquals("B",toAddAllSq.getCurrent());
        toAddAllSq.start();
        assertNotEquals(sq.getCurrent(),toAddAllSq.getCurrent());
        assertNotSame(sq,toAddAllSq);

    }

    @Test
    //Add all two non empty sequences without current element. The size
should be the sum of two sequences
    //After add two sequences still have no current elements
    public void test_AddAll_NonEmpty_NoCurrent(){

        Sequence sq = createSequence(new String[]{"A","B"},5,2);
        Sequence toAddAllSq = createSequence(new String[]{"C","D"},3,2);
        sq.addAll(toAddAllSq);
        assertEquals(4,sq.size());
        String expected = "{A, B, C, D} (capacity = 5)";
        assertEquals(expected,sq.toString());
        assertFalse(toAddAllSq.isCurrent());
        assertFalse(sq.isCurrent());


    }

    @Test

    //Add all two non empty sequences without current element. The size
should be the sum of two sequences
    //After add two sequences still have their original current elements

    public void test_AddAll_NonEmpty_Current(){
        Sequence sq = createSequence(new String[]{"A","B"},5,0);
        Sequence toAddAllSq = createSequence(new String[]{"C","D"},3,1);
```

```
        sq.addAll(toAddAllSq);
        assertEquals(4,sq.size());
        String expected = "{>A, B, C, D} (capacity = 5)";
        assertEquals(expected,sq.toString());
        assertEquals("D",toAddAllSq.getCurrent());
        assertEquals("A",sq.getCurrent());

    }


    @Test
    //test Advance empty sequence. Nothing change
    public void test_Advance_Empty(){
        Sequence sq = new Sequence();
        sq.advance();
        assertEquals(10,sq.getCapacity());
        assertEquals(0,sq.size());
        assertFalse(sq.isCurrent());
        String toStringExpected = "{} (capacity = 10)";
        assertEquals(toStringExpected,sq.toString());

    }

    @Test
    //Test advance a non empty sequence with current. After advancing the
current element change to the next one
    public void test_Advance_Current(){
        Sequence sq = createSequence(new String[]{"A","B","C","D"},4,0);
        sq.advance();
        assertEquals("B",sq.getCurrent());
        String toStringExpected = "{A, >B, C, D} (capacity = 4)";

        assertEquals(toStringExpected,sq.toString());

        sq.advance();
        assertEquals("C",sq.getCurrent());
        String expected = "{A, B, >C, D} (capacity = 4)";
        assertEquals(expected,sq.toString());

        sq.advance();
        assertEquals("D",sq.getCurrent());
        String answer = "{A, B, C, >D} (capacity = 4)";
        assertEquals(answer,sq.toString());


    }

    @Test
    //Test advance with a non empty sequence without current element. Call
isCurrent return false
    public void test_Advance_NoCurrent(){
        Sequence sq = createSequence(new String[]{"A","B","C","D"},4,4);
        sq.advance();
```

```
        assertFalse(sq.isCurrent());
        String toStringExpected = "{A, B, C, D} (capacity = 4)";
        assertEquals(toStringExpected,sq.toString());

    }



    @Test
    //Test advance with a sequence whose current element is at the end. Then
advancing causes no current element (null)
    public void test_Advance_Last(){
        Sequence sq = createSequence(new String[]{"A","B","C","D"},4,3);
        sq.advance();
        assertFalse(sq.isCurrent());
        String toStringExpected = "{A, B, C, D} (capacity = 4)";
        assertEquals(toStringExpected,sq.toString());


    }



    @Test
    //Test clone with an empty sequence. Then the clone has the same contents
 as the original
    public void test_Clone_EmptySequence(){
        Sequence sq = new Sequence();
        Sequence toCloneSq = sq.clone();
        assertEquals(10,toCloneSq.getCapacity());
        assertEquals(0,toCloneSq.size());
        assertFalse(toCloneSq.isCurrent());
        String expected = "{} (capacity = 10)";
        assertEquals(expected,toCloneSq.toString());

    }

    @Test
    //Test clone of a non empty sequence. Then the current elements of both
sequences are the same position
    //When make changes to clone or original, the other one doesn't change

    public void test_Clone_Sequence_DifferentObjects(){
        Sequence sq = createSequence(new String[]{"A","B","C","D","E"},10,3);
        Sequence toCloneSq = sq.clone();
        assertNotSame(sq,toCloneSq);
        sq.addAfter("F");
        assertNotEquals(sq.toString(),toCloneSq.toString());

        toCloneSq = sq.clone();
        toCloneSq.addAfter("F");
        assertNotSame(sq,toCloneSq);
```

```
    }

    @Test
    //Cloning should produce identical contents: same size, capacity, current
 element
    public void test_Clone_Identical(){
        Sequence sq = createSequence(new String[]{"A","B","C","D","E"},10,3);
        Sequence toCloneSq = sq.clone();
        assertTrue(sq.equals(toCloneSq));

    }

    @Test
    //Test removeCurrent of non empty sequence with no current element.
    //When getCurrent it is null
    public void test_Remove_NoCurrent(){
        Sequence sq = createSequence(new String[]{"A","B"},3,2);
        sq.removeCurrent();
        assertNull(sq.getCurrent());
    }

    @Test
    //Test removeCurrent of an empty sequence which means there is no current
 element.
    //When getCurrent it is null
    public void test_Remove_EmptySequence(){
        Sequence sq = new Sequence();
        sq.removeCurrent();
        assertNull(sq.getCurrent());

    }

    @Test
    //Test removeCurrent of a non empty sequence whose current element is at
the end.
    //Then after remove call getCurrent it is null
    public void test_Remove_Last(){
        Sequence sq = createSequence(new String[]{"A","B","C"},4,2);
        sq.removeCurrent();
        assertNull(sq.getCurrent());

    }

    @Test
    //Test removeCurrent of a non empty sequence whose current element is NOT
 at the end.
    //Then the next element is the current element
    public void test_Remove_First(){
        Sequence sq = createSequence(new String[]{"A","B","C"},4,0);
        sq.removeCurrent();
        assertEquals("B",sq.getCurrent());

    }
```

```
    @Test
    //Test removeCurrent of a non empty sequence whose current element is NOT
 at the end.
    //Then the next (last) element is the current element
    public void test_Remove_Middle(){
        Sequence sq = createSequence(new String[]{"A","B","C"},4,1);
        sq.removeCurrent();
        assertEquals("C",sq.getCurrent());
    }

    @Test
    //Test the number of element in an empty default sequence. Return 0

    public void test_Size_Default(){
        Sequence sq = new Sequence();
        assertEquals(0,sq.size());

    }

    @Test
    //Test the number of element in an empty non default sequence. Return 0

    public void test_Size_NonDefault(){
        Sequence sq = new Sequence();
        assertEquals(0,sq.size());

    }

    @Test
    //Test the number of element in a non empty sequence. Return that number

    public void test_Size_NonEmpty(){
        Sequence sq = createSequence(new String[]{"A","B"},3,1);
        assertEquals(2,sq.size());

    }

    @Test
    //Test start of an empty sequence. Then do nothing because it has no
current element.
    //When call isCurrent return false
    public void test_Start_EmptySequence(){

        Sequence sq = new Sequence();
        sq.start();
        assertFalse(sq.isCurrent());


    }

    @Test
    //Test start of a filled sequence. Then the first element of the sequence
```

```
is the current element
    public void test_Start_NonEmpty_Current(){

        Sequence sq = createSequence(new String[]{"A","B","C"},5,1);
        sq.start();
        assertEquals("A",sq.getCurrent());
        String toStringExpected = "{>A, B, C} (capacity = 5)";
        assertEquals(toStringExpected,sq.toString());
    }

    @Test
    //Test start of a filled sequence. Then the first element of the sequence
is the current element

    public void test_Start_NonEmpty_NoCurrent(){

        Sequence sq = createSequence(new String[]{"A","B","C"},5,3);
        sq.start();
        assertEquals("A",sq.getCurrent());
        String toStringExpected = "{>A, B, C} (capacity = 5)";
        assertEquals(toStringExpected,sq.toString());

    }

    @Test
    //Test trimToSize of an empty sequence. Then capacity = 0
    public void test_TrimToSize_EmptySequence(){
        Sequence sq = new Sequence();
        sq.trimToSize();
        assertEquals(0,sq.getCapacity());

    }

    @Test
    //Test trimToSize of a filled sequence. Reduce the capacity to its actual
size

    public void test_TrimToSize_NonEmpty(){

        Sequence sq = createSequence(new String[]{"A","B"},10,1);
        sq.trimToSize();
        assertEquals(2,sq.getCapacity());

    }

    @Test
    //Test toString of an empty sequence. Return {} followed by its capacity
    public void test_ToString_Empty(){
        Sequence sq = new Sequence();
        String expected = "{} (capacity = 10)";
        assertEquals(expected,sq.toString());

    }
```

```
    @Test
    //Test toString of a sequence with no current element
    public void test_ToString_NoCurrent(){
        Sequence sq = createSequence(new String[]{"A","D"},3,2);
        String expected = "{A, D} (capacity = 3)";
        assertEquals(expected,sq.toString());

    }

    @Test
    //Test toString of a sequence with current element.

    public void test_ToString_Current(){
        Sequence sq = createSequence(new String[]{"A","D"},3,1);
        String expected = "{A, >D} (capacity = 3)";
        assertEquals(expected,sq.toString());

    }

    @Test
    //should be equal if they both have no current, same cap, same size same
order
    public void test_Equals_NoCurrent(){
        Sequence sq = createSequence(new String[]{"A","B","D"},3,3);
        Sequence anotherSq = createSequence(new String[]{"A","B","D"},3,3);
        assertTrue(sq.equals(anotherSq));


    }

    @Test
    //Test equals of two empty sequences. Return true
    public void test_Equals_TwoEmpty_Sequences(){
        Sequence sq = new Sequence();
        Sequence anotherSq = new Sequence(12);
        assertTrue(sq.equals(anotherSq));


    }

    @Test
    //empty sequence != non-empty sequence
    public void test_Equals_EmptyVsNonEmpty(){
        Sequence sq = new Sequence();
        Sequence anotherSq = createSequence(new String[]{"A","B"},3,1);
        assertFalse(sq.equals(anotherSq));

    }

    @Test
    //non-empty sequence != empty sequence
```

```
    public void test_Equals_NonEmptyVsEmpty(){
        Sequence anotherSq = new Sequence();
        Sequence sq = createSequence(new String[]{"A","B"},3,1);
        assertFalse(sq.equals(anotherSq));

    }

    @Test
    //Test equals of two non-empty sequences
    //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.
    public void test_Equals_TwoDifferentCapacity(){
        Sequence sq = createSequence(new String[]{"A"},2,0);
        Sequence anotherSq = createSequence(new String[]{"A"},3,0);
        assertTrue(sq.equals(anotherSq));

    }

    @Test
    //not equal when two equals  have different size even though they have
same capacity and same current element
    public void test_Equals_DifferentSize(){
        Sequence sq = createSequence(new String[]{"A","B","C"},3,1);
        Sequence anotherSq = createSequence(new String[]{"A","B"},3,1);
        assertFalse(sq.equals(anotherSq));

    }

    @Test
    //Two sequences with different elements should not be equal even if they
have same capacity, size, current element

    public void test_Equals_DifferentElements(){
        Sequence sq = createSequence(new String[]{"A","B","D"},3,1);
        Sequence anotherSq = createSequence(new String[]{"A","B","G"},3,1);
        assertFalse(sq.equals(anotherSq));

    }

    @Test
    //Should not be equal even though they are identical in capacity, size,
elements, order

    public void test_EqualsIdentical_ButOneHasCurrentOneNot(){
        Sequence sq = createSequence(new String[]{"A","B","D"},3,1);
        Sequence anotherSq = createSequence(new String[]{"A","B","D"},3,3);
        assertFalse(sq.equals(anotherSq));
```

```
    }

    @Test
    //Should not be equal even though they are identical in element, capacity
, size, current element
    public void test_Equals_DifferentOrder(){
        Sequence sq = createSequence(new String[]{"A","B","D"},3,1);
        Sequence anotherSq = createSequence(new String[]{"D","B","A"},3,1);
        assertFalse(sq.equals(anotherSq));



    }

    @Test
    //Should not be equal even though they are identical in element, capacity
, size, order
     public void test_Equals_DifferentCurrent(){
        Sequence sq = createSequence(new String[]{"A","B","D"},3,1);
        Sequence anotherSq = createSequence(new String[]{"A","B","D"},3,0);
        assertFalse(sq.equals(anotherSq));

    }

    @Test
    //equal to itself
    public void test_Equals_Reflexivity(){
        Sequence sq = createSequence(new String[]{"A","B","C"},3,1);
        assertTrue(sq.equals(sq));

    }

    @Test
    //clone equal to original
    public void test_Equals_Clone(){
        Sequence sq = createSequence(new String[]{"A","B","D"},3,1);
        Sequence anotherSq = sq.clone();
        assertTrue(sq.equals(anotherSq));


    }

    @Test
    //if A = B then B = A

    public void test_Equals_Symmetry(){
        Sequence sq = createSequence(new String[]{"A","B","C"},3,1);
        Sequence anotherSq = createSequence(new String[]{"A","B","C"},7,1);
        assertTrue(sq.equals(anotherSq));
        assertTrue(anotherSq.equals(sq));


    }
```

```
    @Test
    //Test isEmpty of an empty sequence. Return true

    public void test_IsEmpty_EmptySequence(){
        Sequence sq = new Sequence();
        assertTrue(sq.isEmpty());

    }

    @Test
    //Test isEmpty of a non empty sequence with current. Return false
    public void test_IsEmpty_NonEmptySequence_Current(){
        Sequence sq = createSequence(new String[]{"A","B"},3,0);
        assertFalse(sq.isEmpty());
    }

    @Test
    //Test isEmpty of a non empty sequence without current. Return false
    public void test_IsEmpty_NonEmptySequence_NoCurrent(){
        Sequence sq = createSequence(new String[]{"A","B"},3,2);
        assertFalse(sq.isEmpty());
    }

    @Test
    //Test clear an empty sequence. Capacity not change. Still the same
contents
    public void test_Clear_EmptySequence(){
        Sequence sq = new Sequence();
        sq.clear();
        assertEquals(10,sq.getCapacity());
        assertFalse(sq.isCurrent());
        assertEquals(0,sq.size());
        String toStringExpected = "{} (capacity = 10)";
        assertEquals(toStringExpected,sq.toString());


    }

    @Test
    //Test clear a non empty sequence. After clearing the sequence should be
empty and there is no current element
    //Capacity don't change. Size after clearing = 0
    public void test_Clear_NonEmpty(){
        Sequence sq = createSequence(new String[]{"A","B"},3,0);
        sq.clear();
        assertEquals(0,sq.size());
        assertEquals(3,sq.getCapacity());
        assertFalse(sq.isCurrent());
        String toStringExpected = "{} (capacity = 3)";
        assertEquals(toStringExpected,sq.toString());
```

```
        }



}
```