```java
/**
 * Linked List is a collection of data nodes. All methods here relate to how one
 * can manipulate those nodes.
 *
 * @author Xavier
 * @version 02.09.17
 *
 *          * 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 LinkedList<E extends Comparable<E>> {
      private int length; // number of nodes
      private ListNode<E> firstNode; // pointer to first node

      public LinkedList() {
            length = 0;
            firstNode = null;
      }


      /**
       * Inserts at specified location
       * @param The index at which to insert, starting at 0
       * @param The thing to insert
       */
      public void insertAt(int place, E toInsert) {
            ListNode<E> newNode = new ListNode<E>(toInsert);

            if (getLength() == 0) {
                  firstNode = newNode; //If theres nothing in the list, insert at
start
            }

            else {
                  ListNode<E> n;
                  n = firstNode;
                  int index = 0;
                  while (index != (this.getLength()-1) && index < (place - 1)) {
//goes until right before place or end
                        n = n.next;
                        index++;
                  }

                  if (index == this.getLength()) { //adds to end if place is above
end
                        n.next = newNode;
                  }
                  else if(place<=0) { //if place is 0 or less, insert at start
                        newNode.next=firstNode; //
                        firstNode=newNode;
                  }

                  else { //n.next is 'place', adds before place.
                        newNode.next = n.next;
                        n.next = newNode;
                  }
            }
            length++;
      }

      /**
       * Removes the data from the given location and returns it
       * @param The location to remove, starting at 0
       * @return The removed information
       */

      public E removeAt(int place) {
            if(place<0 || place>=this.getLength())
                  return null;

            ListNode<E> n;
            n = firstNode;
            E toReturn;

            if(place==0) { //If its the first node
                  toReturn=n.data;
                  firstNode=n.next;
                  length--;
                  return toReturn;
            }

            int index = 0;
            while (index != (this.getLength()-1) && index < (place - 1)) { //Goes
until before the location
                  n = n.next;
                  index++;
            }

            toReturn = (E)(n.next.data); //returns the data at the place
            n.next=n.next.next;


            length--;
            return toReturn;
      }


      /**
       * Turn entire chain into a string
       *
       * @return return linked list as printable string of format
       *         (string, string, string)
       */
      public String toString() {
            String toReturn = "(";
            ListNode<E> n;
            n = firstNode;
            while (n != null) {
                  toReturn = toReturn + n.toString();
                  n = n.next;
                  if (n != null) {
                        toReturn = toReturn + ", ";
                  }
            }
            toReturn = toReturn + ")";
            return toReturn;
      }
      /**
       * getter for number of nodes in the linked list
       *
       * @return length of LL
       */
      public int getLength() {
            return length;
      }

      /**
       * Gets the information from the specified location
       * @param The index of what to get, starting at 0
       * @return the information
       */
      public E getData(int place) {
            if(place<0 || place>=this.getLength())
                  return null;

            ListNode<E> n;
            n = firstNode;
            E toReturn;
            if(place==0) {
                  toReturn=n.data;
                  return toReturn;
            }

            int index = 0;
            while (index != (this.getLength()-1) && index < (place - 1)) {
                  n = n.next;
                  index++;
            }
            toReturn = (E)(n.next.data);
            return toReturn;
      }

      /**
       * Clears the linked list of everything
       */
      public void clear() {
            firstNode=null;
            length=0;
      }



}
```

```java
/**
 * ListNode is a building block for a linked list of data items
 *
 * This is the only class where I'll let you use public instance variables.
 * It's so we can reference information in the nodes using cascading dot
 * notation, like
 *          N.next.data instead of
 *          N.getNext().getData()
 *
 * @author C. Fernandes and G. Marten
 * @version 2/6/2012
 */
public class ListNode<E extends Comparable<E>>
{
    public E data;      // a "reservation" of the conference room
    public ListNode next;   // pointer to next node

    /** Non-default constructor
     *
     * @param String a reservation you want stored in this node
     */
    public ListNode(E String)
    {
        this.data = String;
        this.next = null;
    }

    // if you say "System.out.println(N)" where N is a ListNode, the
    // compiler will call this method automatically to print the contents
    // of the node.  It's the same as saying "System.out.println(N.toString())"
    public String toString()
    {
      if(this.data!=null)
            return data.toString();
        return null;  // call the toString() method in String class
    }

      /**
       * Compares two objects that.
       * @param other The other object to compare to
       * @return -1 if less than, 0 if equal, 1 if greater than.
       */
      public int compare(E other) {
            return this.compare((E)other);
      }
}
```

```java
/**
 * 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. If it does not exist, it will be a value
 * greater than the number of items in the sequence
 *
 * There will never be empty spaces between elements in the Sequence, and the
spaces after the elements will
 * be null.
 *
 * @author xavier
 */
public class Sequence {

      private int current;
      private int capacity;
      private LinkedList<String> seq;

      public Sequence() {
            seq = new LinkedList<String>();
            capacity = 10;
            current = this.size() + 1;
      }

      /**
       * Creates a new sequence.
       *
       * @param initialCapacity
       *            the initial capacity of the sequence.
       */
      public Sequence(int initialCapacity) {
            seq = new LinkedList<String>();
            capacity = initialCapacity;
            current = this.size() + 1;
      }

      /**
       * 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) {

            this.sizeCheck();

            if (current > this.size()) {
                  seq.insertAt(0, value);
                  current=0;
            } else {
```

```
            seq.insertAt(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) {
        this.sizeCheck();
        if (current > this.size()) {
            seq.insertAt(this.size()+1, value);
            current=0;
        } else {

            seq.insertAt(current + 1, value);
            current++;
        }

    }

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

    /**
     * @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 (this.isCurrent()) {
            return seq.getData(current);
        }
        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 (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(current>this.size()) {
            current=(this.size()+another.size()+1);
        }

        // 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.insertAt(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.getData(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() {
      seq.removeAt(current);


}
```

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

            return seq.getLength();
      }

      /**
       * 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 (this.isEmpty()) {
                  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 (seq.getData(tester) != null && tester < this.getCapacity()) {
//While you don't run out of nodes

                  if(tester==current) { //Add > for the current string
                        toReturn=toReturn + ">";
                  }

                  toReturn = toReturn + seq.getData(tester); //Adds the info

                  if (tester  < this.getCapacity() - 1
                              && seq.getData(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 if the other sequence is equal to this one.
       */
      public boolean equals(Sequence other) {
            if(this.size()!=other.size()) {
                  return false;
            }
            
            Sequence cloneOne=this.clone();
            Sequence cloneTwo=other.clone();
            cloneOne.start();
            cloneTwo.start();
            
            for(int i=0;i<this.size();i++) {
                  if(cloneOne.getCurrent()==null || cloneTwo.getCurrent()==null ||
cloneOne.getCurrent().compareTo(cloneTwo.getCurrent())!=0) {
                        return false;
                  }
                  cloneOne.advance();
                  cloneTwo.advance();
            }
            
            if(this.getCurrent()!=null && other.getCurrent()!=null &&
this.getCurrent().compareTo(other.getCurrent())!=0) {
                  return false;
            }
            
            return true;
      }

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

```java
    /**
     * empty the sequence. There should be no current element.
     */
    public void clear() {
        seq.clear();
        current = -1;
    }

    /**
     * If adding to the sequence will overfill it, make it 2x the size plus one
     */
    private void sizeCheck() {
        if (this.getCapacity() < this.size() + 1) {
            scaleTo((seq.getLength() * 2) + 1);
        }

    }

    /**
     * Resizes the capacity to the given size
     * @param newSize the size it will resize to
     */
    private void scaleTo(int newSize) {
        capacity = newSize;
    }

}
```

```
/**
 * Testing suite for BetterBag
 *
 * @author Xavier Qunn, Chris Fernandes, and Matt Anderson
 * *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 LinkedListTester {

    public static final boolean VERBOSE = true;

    /* Runs a bunch of tests for the BetterBag class.
     * @param args is ignored
     */
    public static void main(String[] args)
    {

    Testing.setVerbose(true);
      Testing.startTests();

      testInserts();
      testRemove();


      Testing.finishTests();

    }






    private static void testInserts() {
      Testing.testSection("Tests insertAtHead, insertAtTail, and toString");
      LinkedList<String> list=new LinkedList<String>();

      LinkedList<String> list2=new LinkedList<String>();

      LinkedList<Integer> intList=new LinkedList<Integer>();



      list.insertAt(0, "One");
      Testing.assertEquals("Tests addition in empty list at start", "(One)",
list.toString());
      Testing.assertEquals("Tests addition in empty list capacity", 1,
list.getLength());

      list.insertAt(5, "Two");
      Testing.assertEquals("Tests addition at location longer than length", "(One,
Two)", list.toString());
      Testing.assertEquals("Tests addition in empty list capacity", 2,
list.getLength());

      list.insertAt(1, "Three");
      Testing.assertEquals("Tests addition between nodes", "(One, Three, Two)",
list.toString());
```

```
      Testing.assertEquals("Tests addition in empty list capacity", 3,
list.getLength());

      list.insertAt(0, "Four");
      Testing.assertEquals("Tests addition at start", "(Four, One, Three, Two)",
list.toString());
      Testing.assertEquals("Tests addition in empty list capacity", 4,
list.getLength());

      list.insertAt(-6, "Five");
      Testing.assertEquals("Tests addition at negative index", "(Five, Four, One,
Three, Two)", list.toString());

      Testing.assertEquals("Tests addition in empty list capacity", 5,
list.getLength());
      list.insertAt(6, "Six");
      Testing.assertEquals("Tests addition at end", "(Five, Four, One, Three, Two,
Six)", list.toString());
      Testing.assertEquals("Tests addition in empty list capacity", 6,
list.getLength());

      list2.insertAt(0, "a");
      list2.insertAt(1, null);
      list2.insertAt(2, "b");


      Testing.assertEquals("Tests addition between nodes", "(a, null, b)",
list2.toString());


      intList.insertAt(0, 1);
      Testing.assertEquals("Tests addition in empty list at start", "(1)",
intList.toString());
      Testing.assertEquals("Tests addition in empty list capacity", 1,
intList.getLength());

    }


    private static void testRemove() {
      Testing.testSection("Tests insertAtHead, insertAtTail, and toString");
      LinkedList<String> list=new LinkedList<String>();
      list.insertAt(10, "One");
      list.insertAt(10, "Two");
      list.insertAt(10, "Three");
      list.insertAt(10, "Four");
      Testing.assertEquals("Just checking", "(One, Two, Three, Four)",
list.toString());
      Testing.assertEquals("Tests addition in empty list capacity", 4,
list.getLength());

      Testing.assertEquals("Test removal of last", "Four", list.removeAt(3));
      Testing.assertEquals("Test removal of last", "(One, Two, Three)",
list.toString());
      Testing.assertEquals("Tests capacity after removal", 3, list.getLength());


      Testing.assertEquals("Test removal of first", "One", list.removeAt(0));
      Testing.assertEquals("Test removal of first", "(Two, Three)",
list.toString());
      Testing.assertEquals("Tests capacity after removal", 2, list.getLength());

      Testing.assertEquals("Test removal of first", null, list.removeAt(-5));
      Testing.assertEquals("Test removal of first", "(Two, Three)",
list.toString());
      Testing.assertEquals("Tests capacity after removal", 2, list.getLength());


      Testing.assertEquals("Test removal of first", null, list.removeAt(5));
      Testing.assertEquals("Test removal of first", "(Two, Three)",
list.toString());
      Testing.assertEquals("Tests capacity after removal", 2, list.getLength());
    }


}
```

```
/**
 * 
 * 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 before 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();
            Testing.assertEquals("Tests clone", true, s1.equals(s2));

            s1.addBefore("four");

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

            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();
            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");


      }
}
```