

# CIS 120 Lab 8: Midterm II Review

October 28, 2009

1. Extend the `LinkedListSimpleCollection` class with a method that “rotates” the first element to the end of the list. For example, if the list contains

```
"one", "two", "three", "four" ,
```

then calling `rotate()` on this list would yield the list

```
"two", "three", "four", "one".
```

```
public void rotate() {
```

```
}
```

2. Add a method that reverses the list. For example, if the list contains

```
"one", "two", "three", "four",
```

then calling `reverse()` on this list would yield

```
"four", "three", "two", "one".
```

```
public void reverse() {
```

```
}
```

3. Add a method that removes *all* occurrences of a given element from the list. For example, if the list was

```
"one", "two", "two", "three", "two", "four",
```

then calling `removeAll("two")` on this list would result in the list

```
"one", "three", "four".
```

*Note:* You should *not* use `LinkedListSimpleIterator`'s `remove()` method to accomplish this task. Instead, work directly with the list's nodes.

```
public void removeAll(E toRemove) {
```

```
}
```

4. Consider the following implementations of the `numberOfOccurrences` method from the `TextProcessor` lab.

A correct implementation:

```
public static String numberOfOccurrences
    (String input, String word) {
    int pos = input.indexOf(word);
    int num = 1;

    while (pos != -1) {
        int splitPos = pos + word.length();
        input = input.substring(0, splitPos) + num +
            input.substring(splitPos);
        num++;
        pos = input.indexOf(word, pos+1);
    }
    return input;
}
```

An incorrect implementation:

```
public static String numberOfOccurrencesBad
    (String input, String word) {
    String result = "";
    int count = 0;
    String[] parts = input.split(" ");

    for(int i = 0; i < parts.length; i++) {
        if(parts[i].equals(word)) {
            count++;
            result += parts[i] + count + " ";
        } else {
            result += parts[i] + " ";
        }
    }
    return result.substring(0, result.length() - 1);
}
```

Briefly describe a situation where the bad implementation will return an incorrect string:

Now write a JUnit test capturing this situation—*i.e.*, fill in the body of the methods below so that `testgood()` will succeed and `testbad()` will fail. Both methods should include a single call to `assertEquals`. `testgood()` and `testbad()` should be identical except that `testgood()` calls `numberOccurrences` and `testbad()` calls `numberOccurrencesBad`.

```
public class TestNumberOccurrences extends TestCase {

    public void testgood () {

        //Fill in here

    }

    public void testbad () {

        //Fill in here

    }
}
```

## Reference

```
public interface SimpleCollection<E> {
    boolean add(E element);
    boolean contains(Object o);
    SimpleIterator<E> iterator();
}
```

```
public interface SimpleIterator<E> {
    boolean hasNext();
    E next();
}
```

```
public class Node<E> {
    public E element;
    public Node<E> next;

    public Node (E element, Node<E> next) {
        this.element = element;
        this.next = next;
    }
}
```

```
public class LinkedSimpleCollection<E>
    implements SimpleCollection<E> {
    Node<E> first = null;

    public boolean add(E element) {
        Node<E> newnode = new Node<E>(element, first);
        first = newnode;
        return true;
    }

    public boolean contains(Object o) {

        for ( Node<E> current = first
            ; current != null
            ; current = current.next) {

            if ( (current.element == null && o == null)
                || current.element.equals(o)) {
                return true;
            }
        }
        return false;
    }

    public SimpleIterator<E> iterator() {
        return new LinkedSimpleIterator<E>(this);
    }
}
```