This exercise is intended to ensure that you grasp the concepts taught in Lecture 14, last Monday. It should take you no more than an hour or two to complete. The questions assume that you have at hand the code corresponding to the lecture, which includes the rep invariant and abstraction function of the ListSet class.

1. Here's the code for the checkRep method of ListSet, the list-based implementation of sets that we discussed in class. Why is the last assertion likely to have the greatest execution cost, and how will the cost vary with the size of the set?

   ```java
   private static final boolean DEEP_REPCHECK = true;
   void checkRep () {
       assert elements != null : "ListSet, Rep invariant: elements non-null";
       List<E> l = elements;
       if (!DEEP_REPCHECK) return;
       while (l.size() > 0) {
           E first = l.first();
           List<E> rest = l.rest();
           assert first != null : "ListSet, Rep invariant: no null elements";
           assert !rest.contains(first) : "ListSet, Rep invariant: no duplicates";
           l = rest;
       }
   }
   ```

2. Examine the code of ListSet in the repository. Why is checkRep called only at the end of the constructors, and not for producers, which also generate new values of the rep?

3. Consider this code for the add method. (a) Why the assertion? (b) Give a brief argument that the method is correct (that is, returns a rep value that satisfies the rep invariant, whose corresponding abstract value meets the specification).

   ```java
   /**
    * @requires e != null
    * @return this \ {e}
    */
   public Set<E> add (E e) {
       assert e != null : "ListSet.add(null)";
       if (elements.contains (e)) return this;
       return new ListSet<E> (elements.add (e));
   }
   ```

   Here is an annotated version of the remove method that includes intermediate invariants that explains (in part) why the code meets the specification:

   ```java
   /**
    * @requires e != null
    * @return this \ {e}
    */
   public Set<E> remove (E e) {
       if (isEmpty()) return this;
       E first = elements.first();
       ListSet<E> rest = new ListSet<E> (elements.rest());
       // A(this) = A(rest) U {first} and first not in rest
       if (first.equals(e))
           // first = e => A(this) \ {e} = A(rest)
           return rest;
       else
           // first != e => A(this) \ {e} = (A(rest) \ {e}) U {first}
           return rest.remove(e).add(first);
   }
   ```
4. Complete the argument, by explaining why there are no (a) null pointer dereferences, and (b) calls that violate preconditions.

5. For the precondition, and each part of the rep invariant, say where it is used (if at all).

Now consider the removeAll method:

```java
/**
 * @requires s != null
 * @return this \ s
 */
public Set<E> removeAll (Set<E> s) {
    if (isEmpty() || s.isEmpty()) return this;
    else {
        E e = elements.first();
        Set<E> s2 = new ListSet<E>(elements.rest()).removeAll(s);
        if (s.contains(e))
            return s2;
        else
            return s2.add(e);
    }
}
```

6. Construct a similar argument by adding annotations to this method in the style of the annotations for remove.

7. Modify the code of the method so that it exploits the rep invariant more fully.