CSE 331 Midterm Exam **Sample Solution** 11/5/12

Question 1. (16 points) (assertions) Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your answers if possible.

(a)

{ **(z+1) + 2*z = 4** } = **{ 3*z+1 = 4 }** = **{ z = 1}**

```
x = z + 1;
```

{ **x + 2*z = 4** }

```
y = 2 * z;

{ x + y = 4 }
```

(b)

{ **| 2 * (x-1) | > 2** } = **{ | 2*x – 2 | > 2 }** = **{ 2*x-2 > 2 or 2*x-2 < -2 }**
= **{ x > 2 or x < 0 }**

```
x = x - 1;
```

{ **| 2*x | > 2** }

```
y = 2 * x;

{ |y| > 2 }
```

Question 2. (12 points) Specification. Here are four possible specifications for the function `double sqrt(double x)`, a method that returns the square root of its argument.

A @requires x >= 0
@return y such that |y*y - x| < 0.001

B @requires x >= 0
@return y such that |y*y - x| < 0.001
@throws IllegalArgumentException if x < 0

C @return y such that |y*y - x| < 0.001 if x >= 0, and 0.0 if x < 0

D @requires x >= 0
@return y such that |y*y - x| < 0.00001

For each of the following pairs of specifications, circle the stronger specification or circle "neither" if the two specifications are either equivalent or incomparable.

(i)	A	B	neither
(ii)	A	C	neither
(iii)	A	D	neither
(iv)	B	C	neither
(v)	B	D	neither
(vi)	C	D	neither

After the exam we discovered a bug in this question and wound up discarding it and giving everyone full credit. The bug is that specification B contains an error: it both specifies a precondition and also specifies an exception that is thrown if the argument is negative. Those clauses are incompatible – either the precondition or the @throws clause needs to be deleted.

The next several questions concern the following class that stores a finite, sorted list of strings, possibly containing duplicates. Feel free to remove this page from the exam for reference while you're working.

To save writing, we refer to the class name FiniteSortedStringList as FSSL in parts of this question. We also use the term "capacity" to refer to the maximum number of items that can be stored in the list and "size" to refer to the number of items currently in the list.

```
public class FiniteSortedStringList {

  // items[0..size-1] contains Strings sorted in order
  // given by compareTo (i.e., items[0]<=items[1]....
  // Entries are not null.  There may be multiple copies
  // of the same string in the FSSL.

  private String[] items;
  private int size;

  /** Construct new FSSL with given capacity. */
  public FiniteSortedStringList(int capacity) {
    this.items = new String[capacity];
    this.size  = 0;
  }

  /** Return capacity of this FSSL */
  public int capacity() { return items.length; }

  /** Return current size of this FSSL */
  public int size() { return size; }

  /** Return item at position i of this FSSL */
  public String get(int i) {
    return items[i];
  }

  /** Return whether s is located in this FSSL */
  public boolean contains(String s) {
    if (size == 0) return false;
    return Arrays.binarySearch(items, 0, size, s) >= 0;
  }

  // additional methods to be added or discussed below...

}
```

Question 3. (10 points) The get method shown above is not specified carefully. In particular, it can fail if used improperly.

Below, give a complete, appropriate specification for method `get`. If some part of the specification would be empty or "none", say so explicitly.

```
/**
 * Return item at position i of this FSSL */
 *

 * @requires 0 <= i < size()

 * @modifies  nothing

 * @effects  none

 * @return  String located at position i of this

 * @throws  nothing

public String get(int i) {
  return items[i];
}
```

Question 4. (24 points) Loop development. We would like to add a method to this class to insert new strings. The informal method specification is:

```
/**
 * Add new string to this FSSL in an appropriate location
 * @param s String to add
 * @requires this.size() < this.capacity() and s not null
 */
public void add(String s) { ... }
```

For this problem, develop an implementation of this method and prove your implementation is correct. Your code will need to locate the correct place in the array to insert the new string, and shift over existing strings to make room. If this string is a duplicate of one or more strings already in the array, your code may insert the new copy in any appropriate place.

For full credit, you must invent an appropriate loop invariant and show it is established by any initialization code, is maintained as the loop executes, and that termination of the loop and any additional code executed after that stores the new string in the proper location. Your proof does not need to be completely formal, but needs to be sufficiently careful to convince the reader that the code and proof are correct (i.e., you can skip over tedious, obvious logic steps and simplifications – provided they really are obvious).

You may not call any other methods. This method should be self-contained.

When writing assertions and reasoning about the code, feel free to use notations like `s>t` and `s<=t` to discuss String values. However in the implementation you should use the appropriate Java `compareTo` method to actually compare Strings. (Recall that `s.compareTo(t)` returns a negative number if "s < t", zero if "s equals t", and positive if "s > t".)

You may handle precondition (@requires) violations in any way you wish, including ignoring them and blaming the user or launching the missiles ☺ (e.g., you don't need to worry about precondition violations for this problem).

Hint: The code and proof are simpler (or at least shorter) if you shift array elements at the same time as you search for an appropriate place to add the new string.

Write

Your

Answer

On the

Next page …

Question 4. (cont.) Write your code and correctness proof below.

```
/** Add s to this list in the appropriate location */
public void add(String s) {

   // Strategy: shift items from right to left until
   // all items > s have been moved over

   { pre: s != null &&size() < capacity() &&
         items[0..size-1] sorted (RI) }

   int k = size;

   { inv: items[0..k-1] <= items[k+1..size] &&
          items[k+1..size] > s }

   while (k > 0 && items[k-1].compareTo(s) > 0) {

      { inv && k > 0 && items[k-1] > s }

      items[k] = items[k-1];

      { items[k..size] > s }

      k = k-1;

      { items[k+1..size] > s => inv }

   }

   { inv && (k = 0 || items[k-1] <= s  =>

      items[0..k-1] <= s && items[k+1..size] > s }

   items[k] = s;

   { items[0..size] are sorted }

   size = size + 1;

   [ items[0..size-1] are sorted => RI }


}
```

In this and the following questions, some answers in this sample solution are longer or more descriptive than would be needed on the test.

Question 5. (12 points) Testing. As part of implementing your `add` method, you should also create tests for it. Below, describe four good **black box** tests for this method. For full credit, each test should cover a different revealing subdomain. You do not need to give JUnit or other code, but for full credit you do need to describe a specific test setup and the expected results for each. Be brief and to the point.

For full credit, each of the four tests needed to specifically describe the test setup and observed results. Simply saying things like "insert two different strings into the list" would not be adequate since it doesn't say how to verify the results.

Here are some possibilities:

Create an empty list and add "x". Verify that get(0) returns "x", contains("x") returns true, and size()==1.

Create an empty list, add strings "pqr", "abc", and "xyz" in that order. Verify that size()==3 and use get() to verify that the strings are stored in sorted order.

Create a list with capacity == 4. Add "def", "abc", "xyz", and "pqr". Verify that size()==4 and use get() to check that the 4 strings are contained in the list in the correct locations.

Create an empty list, add "def" and "pqr". Verify that size()==2. Add "def", verify that size()==3, and use get() to check that the list is sorted and contains the three strings in the expected order.

Question 6. (7 points) Are there any potential representation exposure problems with the FSSL (FiniteSortedStringList) class from the last several questions? (That includes the `add` method you implemented above.) Why or why not? (But be brief!)

No. All of the instance variables are private and the only data that is shared by the list and client code are references to immutable String objects, which the client cannot change.

Question 7. (7 points) The company intern, A. Hacker, points out that this FSSL class does not have a `checkRep` method, and, being a diligent CSE 331 student, suggests including the following method and inserting calls to it at the beginning and end of each public method in the class:

```
private void checkRep() throws RuntimeException {
  if (size < 0 || size > items.length)
    throw new RuntimeException("size out of bounds");
  for (int k = 0; k < size; k++)
    if (items[k] == null)
      throw new RuntimeException("null item");
  for (int k = 0; k < size-1; k++)
    if (items[k].compareTo(items[k+1]) > 0)
      throw new RuntimeException("items out of order");
}
```

Is this a reasonable `checkRep` method to include and use in this class? All of the time, part of the time, or never? Why or why not? (Again, be brief?)

This would be useful during debugging, but not in production code. The cost of this `checkRep` method is $O(n)$, while most of the operations in the class are constant or $O(\log n)$ time. Using this `checkRep` for those operations in production would not be reasonable since it would dominate their actual cost.

A possible exception is method `add`, where an extensive check would guard against most potential RI problems. Since `add` already costs $O(n)$ time, the asymptotic time, at least, would be no worse with this `checkRep`, although it would increase the cost of `add` by a significant constant factor.

Question 8. (12 points) Suppose we have the following code for a class that represents circles in a 2-D graphics system.

```
public class Circle {
  private int x;          // x and y center coordinates
  private int y;
  private int radius;     // radius

  // two Circles are considered to be equal if they
  // occupy the same place on the plane
  public boolean equals(Object other) {
    if (! (other instanceof Circle)) return false;
    Circle c = (Circle) other;
    return this.x == c.x && this.y == c.y;
  }
```

Here are four possible hashCode functions for this class:

```
  public int hashCode() {    // #1
    return x;
  }

  public int hashCode() {    // #2
    return x*x + y*y;
  }

  public int hashCode() {    // #3
    return x+y+radius;
  }

  public int hashCode() {    // #4
    return 42;
  }
```

(a) Which of the four `hashCode` functions above meet the requirements for a correct implementation? You do not need to give reasons, just list the ones that are correct.

Correct: #1, #2, #4 (#3 is incorrect since two Circles that are equal (c1.equals(c2) true) might have different radius values).

(b) Which of the four functions above is likely to be the best quality `hashCode` function and (briefly) why?

#2 since it uses more information about the state of the Circle and is more likely to produce different values for circles with different coordinates.