CSE 331 Summer 2017 Midterm Exam

Name _____

The exam has 5 regular problems and 1 bonus problem. Only the regular problems will count toward your midterm score. The bonus problem will be included in your bonus score, which may be used to break ties if you end up on the border between different grades.

The exam is closed book and closed electronics. One page of notes is allowed.

Please **wait to turn the page** until everyone is told to begin.

Score _____ / 70

1. _____ / 14

2. _____ / 16

3. _____ / 16

4. _____ / 14

5. _____ / 10

Bonus: _____ / 10

CSE 331 Summer 2017 Midterm Exam

Problem 1 (Concepts). Answer each of the following questions. For parts c – f, give a short answer (1 or 2 sentences) for each question.

a. A failure encountered by users is usually most expensive to fix when the cause is (circle one)

a defect in the implementation

a defect in the design

a defect in the requirements

The next two questions refer to the following method:

```
// @requires x > 0
// @returns the x-th Fibonacci number
public static Number fibonacciNumber(Integer x) { … }
```

b. Which of the following would strengthen the specification of `fibonacciNumber` (circle all that apply)?

change `@requires` to x >= 0

change the type of `x` to `Number`

change the type of the return value to `Integer`

change the type of the return value to `Object`

c. Give an example of client code that would be broken by making one of the described changes above.

```
Number x = fibonacciNumber(new Integer(1));
```

Kevin has written the following class:

```
// Represents a queue of integers.
public class IntQueue {

  // AF(this) = values (first item is first out etc.)
  private List<Integer> values;  // RI: values != null

  public IntQueue() {
    this.values = new LinkedList<Integer>();
  }

  // Returns list of the values in the queue right now.
  public List<Integer> getValues() { return values; }

  ...
}
```

Now suppose that Bob writes the following code, using Kevin's class:

```
IntQueue Q = new IntQueue();
...

// I checked and getValues always returns a LinkedList,
// so it's safe to use LinkedList.getFirst here!
System.out.println(
    ((LinkedList<Integer>) Q.getValues()).getFirst());
```

Later, Kevin changes his `IntQueue` class to use an `ArrayList` instead of a `LinkedList`. Although all his tests pass, he discovers the program now crashes, with a stack trace indicating an exception on the final line of Bob's code.

d. Although Bob will soon be fired, this situation is partly Kevin's fault. What was his mistake, and what defensive programming technique would have prevented it?

Kevin should have copied the values into a new list in `getValues()`.

Suppose that Kevin implements `equals` and `hashCode` as follows:

```
public boolean equals(Object o) { return o == this; }

public int hashCode() {
  // Want hash codes to be pretty random for good
  // performance in hash tables, so this will be perfect!
  return (int)(100000 * Math.random());
}
```

e. What required property of the hashCode contract does this violate? (If you don't remember the name, just describe the property.)

Consistency

Kevin's friend took CSE 331 and showed him `IntQueue2` from HW5, which stores the elements in a Java array. He loves it and decides to use it himself:

```
// AF(this) = [entries[front], entries[(front+1) % n], ...,
//     entries[(front+size-1) % n]] with n = entries.length
int[] entries;   // RI: not null, all unused entries zeroed
int front, size; // RI: non-negative, ...
```

He also changes `equals` so that it checks if the two queues have the same abstract value (i.e., they would produce the same elements in the same order), and he implements `hashCode` as follows:

```
public int hashCode() {
  int c = 0;
  for (int i = 0; i < entries.length; i++)
    c = 3 * c + entries[i];  // unused entry = 0 so ignored
  return c;
}
```

f. What required property of the hashCode contract does this violate? (If you don't remember the name, just describe the property.)

Consistency with equals

The next few problems refer to the following `TextBuffer` class, which is partially implemented for you:

```
/**
 * Represents a sequence of characters with a movable "cursor" at some
 * position in the sequence. The cursor can be moved anywhere in the sequence,
 * but only the characters at the cursor position can be changed, either by
 * deleting or inserting new ones.
 *
 * A typical buffer looks like [abcd^efg], where the sequence contains the same
 * characters as the string "abcdefg" and the cursor is just before the "e".
 * Calling removeChar() at that point will remove 'e', leaving the buffer in
 * the state [abcd^fg], with the cursor just before the 'f'. Calling
 * insertChar('e') after that will insert an 'e' before the 'f', putting the
 * buffer back into its original state [abcd^efg].
 */
public class TextBuffer {

  // AF(this) = sequence of characters
  //   prefixChars[0..prefixLen-1] + reverse(suffixChars[0..suffixLen-1])
  //   with cursor at index cursorPos
  // RI: prefixChars != null and suffixChars != null and
  //     0 <= prefixLen <= prefixChars.length and
  //     0 <= suffixLen <= suffixChars.length and
  //     0 <= cursorPos <= prefixLen + suffixLen
  private char[] prefixChars, suffixChars;
  private int prefixLen = 0, suffixLen = 0;
  private int cursorPos = 0;

  /** Creates an empty sequence with the cursor at the beginning. */
  public TextBuffer() {
    this.prefixChars = new char[4];  // make room for a few chars
    this.suffixChars = new char[0];
  }

  /** @returns the length of the sequence */
  public int getLength() { return prefixLen + suffixLen; }

  /** @returns the sequence of characters as a string */
  public String getText() {
    StringBuilder buf = new StringBuilder();
    buf.append(prefixChars, 0, prefixLen);
    for (int i = suffixLen-1; i >= 0; i--)
      buf.append(suffixChars[i]);
    return buf.toString();
  }

  /** @returns the index where the cursor is currently at */
  public int getCursorPos() { return cursorPos; }

  @Override
  public String toString() {
    StringBuilder buf = new StringBuilder("[");
    buf.append(getText().substring(0, cursorPos));
    buf.append("^");
    buf.append(getText().substring(cursorPos));
    buf.append("]");
    return buf.toString();
  }
```

(this page left intentionally blank)

Problem 2 (ADTs).

a. Give a specification and implementation for a method `moveCursorForward()` that will move the cursor one character closer to the end. (Note that there is more than one reasonable answer to this question.)

```
/**
 * Moves the cursor one index closer to the end of the sequence.
 * @modifies this
 * @effects leaves the characters unchanged but moves the cursor one index
 *   further unless it is at the end already (then, it stays there).
 */
public void moveCursorForward() {
  if (cursorPos < getLength())
    cursorPos += 1;

  checkRep();
}
```

b. Give a specification and implementation for the method `removeChar()` that removes the character under the cursor. (Note that there is more than one reasonable answer to this question.)

You may **assume** the existence of a helper method `moveSplitTo(int pos)` that rearranges `prefixChars` and `suffixChars` so that exactly `pos` characters are in `prefixChars` with the rest in `suffixChars`.

```
/**
 * Removes the character at the cursor position.
 * @modifies this
 * @effects changes state from [A^xB] to [A^B], where A and B are
 *   subsequences (possibly empty) and x is a character.
 * @throws IllegalStateException if the cursor is at the end
 */
public void removeChar() {
  moveSplitTo(cursorPos);
  if (suffixLen > 0) {
    suffixLen -= 1;
  } else {
    throw new IllegalStateException("no character to remove");
  }

  checkRep();
}
```

Problem 3 (Testing).

a. Describe one test for `moveCursorForward()`. Give the initial/final states using the notation from the overview of `TextBuffer` (and returned by `toString`).

In state _____[^abc]_____,
calling `moveCursorForward()`
should produce state _____[a^bc]_____.

Does this test give 100% statement coverage?	Yes	No
Does this test give 100% branch coverage?	Yes	No
Does this test give 100% path coverage?	Yes	No

b. Describe two tests for `removeChar()` using the format above. If possible, you should test two distinct behaviors of the method.

a. In state _____[^abc]_____,
calling `removeChar()`
should produce state _____[^bc]_____.

b. In state _____[abc^]_____,
calling `removeChar()`
should _____throw an `IllegalStateException`_____.

Do these tests give 100% statement coverage?	Yes	No
Do these tests give 100% branch coverage?	Yes	No
Do these tests give 100% path coverage?	Yes	No

c. Are these specification or implementation tests?

All are specification tests

Problem 4 (Reasoning I). Give a **complete** proof of correctness for the following method that evaluates a polynomial at a given point (similar to `RatPoly.eval` in HW4):

```
// @requires coeffs != null
// @returns the value of the polynomial with the given
//    coefficients at the point x. The value coeffs[i] is
//    the coefficient of x^i. For example, if coeffs is
//    [2, 3, 1], it represents x^2 + 3x + 2, evaluating it
//    at x = 5 gives 5^2 + 3*5 + 2 = 42.
public static double eval(double[] coeffs, double x) {
  double val = 0;
  int i = coeffs.length;
```

{{ _____ val = 0 and i = n _____ }}

When i = n, Inv says val = 0, so this holds initially.

{{ Inv: val = coeffs[i] + coeffs[i+1] x + … + coeffs[n-1] x^{n-1-i}, where n = coeffs.length }}

```
  while (i != 0) {
```

x * val + coeffs[i-1], with val as in Inv, gives the right side below.

{{ x * val + coeffs[i-1] = coeffs[i-1] + coeffs[i] x + … + coeffs[n-1] x^{n-1-i+1} }}

```
    i--;
```

{{ x * val + coeffs[i] = coeffs[i] + coeffs[i+1] x + … + coeffs[n-1] x^{n-1-i} }}

```
    val = x * val + coeffs[i];
```

{{ val = coeffs[i] + coeffs[i+1] x + … + coeffs[n-1] x^{n-1-i} }}

```
  }
```

Inv says the following when i = 0, and i = 0 upon termination.

{{ val = coeffs[0] + coeffs[1] x + … + coeffs[n-1] x^{n-1} }}

```
  return val;
}
```

Problem 5 (Reasoning II). The following method, from the first version of HW2, takes an array `A` as input and is supposed to reverse its first `n` elements. As many students noticed, it contains a bug, which we fixed by changing the loop condition to "`i+1 < j`".

(In the assertions below, A[i] refers to the current value in A[i], while $A[i]_1$ (subscript "1") refers to the original value in A[i] when the method was called.)

```
{{ 0 < n <= A.length }}
public void reverse(int[] A, int n) {
  int i = -1;
  int j = n;

  {{ Inv: A[0] = A[n-1]₁, .., A[i] = A[n-1-i]₁ and A[j] = A[n-1-j]₁, …, A[n-1] = A[0]₁
        and A[i+1], … A[j-1] are unchanged and j = n-1-i }}
  while (i < j) {
    i = i + 1;
    j = j - 1;
    swap A[i], A[j];
  }

  {{ A[0] = A[n-1]₁, …, A[n-1] = A[0]₁ }}
}
```

Since the code is incorrect (without the fix), it's proof of correctness does not go through. Explain precisely why the proof of correctness fails. (That is, when going through the proof of correctness as in the previous question, where do you run into a problem?)

The swap A[i], A[j] line only produces the result $A[i] = A[j]_1$ and $A[j] = A[i]_1$ (required for Inv to hold) if A[i] and A[j] were unchanged before that line. If i + 1 = j, then both A[i] and A[j] at that line have already been changed, so the Inv does not necessarily hold after.

Explain why that part of the proof now goes through when the loop condition is changed to instead say "`i+1 < j`".

If i+1 < j, then A[i+1] holds its original value. The same fact tells us that i < j-1, so A[j-1] also holds its original value. Thus, the body leaves A[i+1] and A[j-1] holding the values that were originally in A[n-1-(i+1)] and A[n-1-(j-1)] respectively.

(Note, in this case, that it's no longer obvious that the post-condition holds upon termination. It does hold, but it takes more thought to see why.)

Bonus Problem (Reasoning III). The following code merges

```
// @requires A and B are sorted, n < A.length, m < B.length
//     and C.length >= n+m
// @modifies C
// @effects C stores A[0..n-1] + B[0..m-1] and is sorted
public void merge(int[] A, int n, int[] B, int m, int[] C){
  int i = 0, j = 0, k = 0;

  {{ Inv: C[0..k-1] holds A[0..i-1] + B[0..j-1] and is sorted, A & B are sorted }}
  while (i < n && j < m) {
    if (A[i] < B[j])
      C[k++] = A[i++]
    else
      C[k++] = B[j++]
  }

  {{ Inv: C[0..k-1] holds A[0..i-1] + B[0..j-1] and is sorted, A & B are sorted }}
  while (i != n)
    C[k++] = A[i++]

  {{ Inv: C[0..k-1] holds A[0..n-1] + B[0..j-1] and is sorted, A & B are sorted }}
  while (j != m)
    C[k++] = B[j++]

  {{ C[0..k-1] holds A[0..n-1] + B[0..m-1] and is sorted, A & B are sorted }}
}
```

In this case, the code is **correct**. However, once again the proof of correctness does not go through. Where does the proof of correctness fail?

C is not necessarily sorted at the end of the body, depending on what was in A[i] or B[j].

To make the proof work, the three loop invariants must be strengthened. What condition can you add to the **first** loop invariant to make the correctness proof work up to the end of the first loop? (You'd need similar changes to the other loops also.)

Add that C[0], …, C[k-1] < A[i] (if i < n) and C[0], …, C[k-1] < B[j] (if j < m). Then adding the smaller element at the end of C preserves sorted order.