CSE 331 Summer 2016 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, closed notes, and closed electronics.

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

Score _________________ / 66

1. ______ / 12

2. ______ / 12

3. ______ / 9

4. ______ / 18

5. ______ / 15

(Bonus: ______ / 15)

CSE 331 Summer 2016 Midterm Exam

Question 1 (Concepts). Answer each of the following questions. For a and c, give a short answer (1 or 2 sentences).

a. Why are specifications necessary for checking correctness?

The specification provides the precondition and postcondition. We can't argue correctness without those.

b. Suppose that we change the specification of a method from S_1 to S_2. This change *will not break any existing client code*[1] if (circle one)

S_2 is weaker than S_1

[✓ S_2 is stronger than S_1]

c. Why is it difficult to verify that our representation invariant always holds if there is *representation exposure*?

With representation exposure, clients could break our representation invariant, so we'd have to check all of the client code as well.

d. Suppose that `a`, `b`, and `c` are objects. Suppose further that `a.equals(b)` and `b.hashCode() == c.hashCode()` are both true. Circle those of the following that must also be true:

`a.equals(c)`

[✓ `a.hashCode() == c.hashCode()`]

[1] Assume that all client code is correct.

Question 2 (Reasoning). **Complete** the proof of correctness on the next page for this method that you wrote in HW4. It computes the product of two polynomials.

This method uses two others you may remember from HW4:

- `scaleCoeff(list, num)`: replaces each term in `list` with a new term whose exponent is unchanged and whose coefficient is multiplied by `num`.
- `incremExpt(list, deg)`: replaces each term in `list` with a new term whose coefficient is unchanged and whose exponent is increased by `deg`.

You may assume (as the code does) that `other` is not `null`.

```
public RatPoly mul(RatPoly other) {
  {{ }}
  RatPoly prod = new RatPoly();  // zero
  {{ prod = 0 }}
  int i = 0;
  {{ prod = 0 and i = 0 }}
      –  if i = 0, then Inv says prod = 0, which shows that Inv is true initially.
  {{ Inv: prod = this * (other.terms[0] + other.terms[1] + … + other.terms[i-1]) }}
  while (i != other.terms.size()) {
    {{ Inv }}
    RatTerm t = other.terms.get(i);
    {{ Inv and t = other.terms[i] }}
    ArrayList newTerms = new ArrayList(terms); // make a copy
    {{ Inv and t = other.terms[i] and
      newTerms = [terms[0], …, terms[n-1]] where n = terms.size() }}
    scaleCoeff(newTerms, t.getCoeff());
    {{ Inv and t = other.terms[i] and
      newTerms = [t.getCoeff() * terms[0], …, t.getCoeff() * terms[n-1]] }}
    incremExpt(newTerms, t.getExpt());
    {{ Inv and t = other.terms[i] and newTerms = [t * terms[0], …, t * terms[n-1]] }}
    RatPoly s = new RatPoly(newTerms);
    {{ Inv and t = other.terms[i] and s = this * t }}
    prod = prod.add(s);
    {{ prod = this * (other.terms[0] + other.terms[1] + … + other.terms[i]) }}
    i = i + 1;
    {{ prod = this * (other.terms[0] + other.terms[1] + … + other.terms[i-1] }}
      –  This is precisely Inv.
  }
      –  if i = other.terms.size(), then Inv says prod = this * other
  {{ prod = this * other }}
  return prod;
}
```

Question 3 (More Reasoning). Below are three different snippets of code that attempt to compute the sum of an array `A` of `BigInteger` objects. (A `BigInteger` stores an integer that can grow arbitrarily large.)

Each of these snippets is **incorrect**. For each one, indicate whether the proof of correctness fails because (1) the invariant does not hold initially, (2) the invariant does not hold after the loop body is executed, or (3) the invariant does not imply the post condition upon termination of the loop. (No explanation is necessary.)

The variable `n` holds the length of `A`. You may assume that `n` is at least 2.

-
  ```
  BigInteger v = A[n-1];
  int i = n - 2;
  {{ Inv: v = A[i+1] + A[i+2] + … + A[n-1] }}
  while (i != 0) {
    v = A[i].add(v);
    i = i - 1;
  }
  {{ v = A[0] + A[1] + … + A[n-1] }}
  ```

  Upon **termination**, we have v = A[1] + [2] + … + A[n-1].

-
  ```
  BigInteger v = A[n-1];
  int i = n - 2;
  {{ Inv: v = A[i] + A[i+1] + … + A[n-1] }}
  while (i != 0) {
    i = i - 1;
    v = A[i].add(v);
  }
  {{ v = A[0] + A[1] + … + A[n-1] }}
  ```

  Initially, we have i = n – 2, which means Inv says v = A[n-2] + A[n-1], whereas the actual value of v initially is A[n-1].

-
  ```
  BigInteger v = A[n-2].add(A[n-1]);
  int i = n - 2;
  {{ Inv: v = A[i] + A[i+1] + … + A[n-1] }}
  while (i != 0) {
    v = A[i].add(v);
    i = i - 1;
  }
  {{ v = A[0] + A[1] + … + A[n-1] }}
  ```

  After the loop body, we have v = A[i+1] + A[i+1] + … A[n-1] (not Inv).

The next two questions make use of the following class `SymbolTable`.

```
/**
 * Stores a map from strings (symbols) to integer IDs. A
 * typical instance is {"foo" -> 1, "bar" -> 2, "baz" -> 3}
 *
 * Abstract Invariant: For any two symbols x and y in this,
 * we have this[x] != this[y]. (I.e., this is of the form
 * {x -> A, y -> B, ...} where A != B.)
 */
public class SymbolTable {

  private final ArrayList<String> symbols;
  private final ArrayList<Integer> ids;

  // Representation Invariant: symbols.size() == ids.size()
  // and neither list contains any nulls.
  //
  // Abstraction Function: AF(r) =
  //   {r.symbols[i] -> r.ids[i] for i = 0 .. ids.size()}

  /** @effects Makes an empty map {} */
  public SymbolTable() {
    this.symbols = new ArrayList<String>();
    this.ids = new ArrayList<Integer>();
  }

  /**
   * Returns the ID of the given symbol if it is already in
   * this and otherwise adds it to this and returns the new
   * unique ID that it was assigned.
   *
   * @param symbol The symbol to look up in this.
   * @requires symbol != null
   * @modifies this
   * @effects adds {symbol -> N} if symbol is not in this,
   *   where N is an ID not used for any other symbol.
   * @returns this[symbol] (possibly after adding to this)
   */
  public int getId(String symbol) {
    // (implementation omitted)
  }

}
```

Question 4 (Testing). Describe tests for the `getId` method from question 4.

Start by writing three black-box, specification tests. For each, describe the state that the table should be in before the call to `getId`, what inputs you will pass to `getId`, and what conditions (*if any*) its return value should satisfy. The third test should be a special, boundary case.

Black Box, Specification Tests

1. With table = { "foo" -> 1, "bar" -> 2 },

   `table.getId("foo")` should return 1.

2. With table = { "foo" -> 1 },

   `table.getId("bar")` should return an ID other than 1.

3. (Boundary case)

   With table = { }, `table.getId("foo")` should return (anything).

After you have finished question 5, return to this problem and complete the clear-box tests as described on the next page….

Clear Box, Implementation Tests

What additional information about the state of the table, not provided in the specification, would be **useful to have access to** in order to test that the implementation is working as intended?

It would be useful to know the symbol that is last in the list.

Choose a name for a new method that will retrieve this information (any is fine):

Let's call this getLast().

Suppose that the above method has been implemented.

Write two more tests, in the same format as above but now including conditions about the state of `table`, using the method named above, both **before** and **after** the call to `table.getId`, that will help verify that the implementation is working as intended.

4. With table = { "foo" -> 1, "bar" -> 2 } and table.last = "foo",

   `table.getId("foo")` should return 1. Afterward, table.last = "foo".

5. With table = { "foo" -> 1, "bar" -> 2 } and table.last = "bar",

   `table.getId("foo")` should return 1. Afterward, table.last = "foo".

Question 6 (ADT Implementation). Implement the `getId` method below.

In addition to satisfying the specification, your implementation must also make the following **performance optimizations**:

- Every time `getId` returns, the symbol on which it was called should now be the last element in the `symbols` and `ids` lists.
- When searching through the symbols list to see whether the symbol is already present, you must start your search at the end of the list. (That way, if `getId` is called on the same symbol repeatedly, each call after the first will take constant time.)

You do not need to write down a proof of correctness for your code — though you should, of course, complete one as you will be graded on its correctness. However, you must document the invariant of any loop in your code to help readers understand how your code works. You can use any combination of English prose and mathematical notation as long as it is clear.

You can also assume an operation `swap(ArrayList list, int i, int j)` that swaps the elements at indices `i` and `j` in the list has been provided.

```
public int getId(String symbol) {
  int i = symbols.size() - 1;
  int maxID = -1;

  {{ Inv: symbol not in [symbols[i+1], ..., symbols[n-1], where n = symbols.size(),
          and maxID = max(ids[i+1], ..., ids[n-1]) }}
  while ((i >= 0) && !symbols.get(i).equals(symbol)) {
    maxID = Math.max(maxID, ids.get(i));
    i = i - 1;
  }

  if (i < 0) {
    symbols.add(symbol);
    ids.add(maxID + 1);
  } else {
    swap(symbols, i, symbols.size() - 1);
    swap(ids, i, symbols.size() - 1);
  }

  return ids.get(ids.size() - 1);
}
```

[Alternative: add to representation invariant that the max value in ids is always ids.size(). Then, we know to choose the ID for a new symbol to be ids.size() + 1.]

Bonus Question. In this course, we have learned several techniques of *defensive programming* that are effective at preventing subtle bugs that would be hard to track down. These include

- Making copies of inputs and outputs of methods.
- Using immutable classes.
- Not defining multiple methods in one class with the same name and number of arguments.

For one of these techniques of your choice, describe an example of a bug that (1) would be very hard to track down and (2) would be prevented by that technique. You must not only describe the bug in detail but also why the bug would be hard to detect via tools, inspection, and testing.

I don't need nightmares. Skipping this problem…