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 )
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?

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?

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
    int i = 0;

    while (i != other.terms.size()) {
        RatTerm t = other.terms.get(i);
        ArrayList newTerms = new ArrayList(terms);  // make a copy
        scaleCoeff(newTerms, t.getCoeff());
        incremExpt(newTerms, t.getExpt());
        RatPoly s = new RatPoly(newTerms);
        prod = prod.add(s);
        i = i + 1;
    }

    prod = this * other
    return prod;
}
CSE 331 Summer 2016 Midterm Exam

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 $i = n - 2$;
  while ($i != 0$) {
    $v = A[i].add(v);$ 
    $i = i - 1;$ 
  } 

• BigInteger $v = A[n-1]$;
  int $i = n - 2$;
  while ($i != 0$) {
    $i = i - 1;$ 
    $v = A[i].add(v);$ 
  } 

• BigInteger $v = A[n-2].add(A[n-1])$;
  int $i = n - 2$;
  while ($i != 0$) {
    $v = A[i].add(v);$ 
    $i = i - 1;$ 
  } 
The next two questions make use of the following class `SymbolTable`.

```java
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 = { ________________________________ },
   table.getId(______) should ________________________________.

2. 

3. (Boundary case)

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?

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

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.

5.
Question 5 (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.

```java
public int getId(String symbol) {
```
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.