CSE 331 Midterm Exam 5/13/16 Sample Solution

Remember: For all of the questions involving proofs, assertions, invariants, and so forth, you should assume that all numeric quantities are unbounded integers (i.e., overflow can not happen) and that integer division is truncating division as in Java, i.e., 5/3 => 1.

Question 1. (10 points) (Forward reasoning) Using forward reasoning, write an assertion in each blank space indicating what is known about the program state at that point, given the precondition and the previously executed statements. Your final answers should be simplified. Be as specific as possible, but be sure to retain all relevant information.

(a)
```
{ x > 7 }

x = x + 4;

{ x > 11 }

y = x - 2;

{ x > 11 && y > 9 }

x = 2 * x;

{ x > 22 && y > 9 }
```

(b)
```
{ |x| > 5 }

if (x > 0)

   { |x| > 5 && x > 0 } => { x > 5 }

   x = 3 - x;

   { x < -2 }

else

   { |x| > 5 && x <= 0 } => { x < -5 }

   x = x - 1;

   { x < -6 }


{ (x < -2) || (x < -6) } => { x < -2 }
```

Question 2. (12 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 final answers if possible.

(a) (5 points)

```
{ |y-2 + 2| = 10 } => { |y| = 10 }
x = y - 2;

{ |x+2| = 10 }
w = x + 2;

{|w| = 10}
```

(b) (7 points)

```
{ (x > y && |x-y| < y) || (x <= y && |x+y| < y) }
if (x > y) {
   { |x-y| < y }
   x = x - y;
   { |x| < y }
} else {
   { |x+y| < y }
   x = x + y;
   { |x| < y }
}
{ |x| < y }
```

Note: This one had a precondition that was more complicated than we intended, and there isn't any easy way to simplify it. We gave full credit for preconditions that were equivalent to this one.

Question 3. (17 points) Loops. The code on the next page is alleged to sort an array of integers using the algorithm "pancake sort". The only operation that modifies the array is `flip(a,k)`, which reverses the elements in the array section `a[k..a.length-1]`. For example, if `a` = [5, 2, 6, 3, 2, 4] then `flip(a,2)` would reverse the section starting at `a[2]` and change `a` to [5, 2, *4, 2, 3, 6*] (the reversed elements are shown in *italics*). You do not need to implement this operation or prove that it works – assume that it is implemented correctly elsewhere.

Prove that `pancakeSort` successfully sorts its array argument by adding appropriate assertions and invariants to the code below. You may assume that the array parameter `a` is not `null` and has at least one element.

To save writing, you may write "a[i..j] is sorted" to mean a[i] ≤ a[i+1] ≤ … ≤ a[j]. You may also use notations like "x = max(a[i..j])" to mean "x is the largest value in a[i..j]" and use similar notations to describe minimum values in sections of the array.

Use the space below for scratch work, then provide your proof on the next page, which should have enough room for your work.

You may remove this page from the exam if you wish.

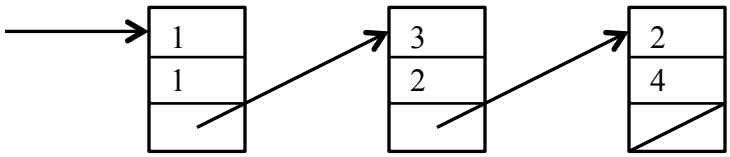
We announced during the exam that to save time it was not necessary to provide a complete correctness proof of the inner loop as long as an appropriate postcondition for that loop was given. In the sample solution we've included an invariant for the inner loop for reference, and the remaining proof steps should be easy to fill in.

We also edited the problem a little in the posted version. In the original version of the problem, the inner loop was initialized with "int j = i; int minLoc = j;", which doesn't quite establish a clean invariant for the loop. The problem is edited here to make the posted version of the exam more useful for studying and practice in the future.

Question 3. (cont.) Write your proof that the pancake sort method is correct by adding assertions and invariants below. You might not need to put an assertion between every two lines of code, but do not omit any essential details or steps.

```
{ pre: a != null && a.length >= 1 }
public static void pancakeSort(int[] a) {
  int i = 0;
  { inv: a[0..i-1] sorted and all a[0..i-1] <= a[i..a.length-1] }
  while (i != a.length) {
    { inv && i != a.length }
    int minloc = i;
    int j = i + 1;
    { inv2: a[minloc] = min(a[i..j-1]) }  -- not required
    while (j != a.length) {
      if (a[j] < a[minLoc]) {
        minLoc = j;
      }
      j = j + 1;
    }
    { a[minLoc] = min(a[i..a.length-1]) && inv }
    flip(a, minLoc);
    { a[a.length-1] = min(a[i..a.length-1]) && inv }
    flip(a, i);
    { a[0..i] sorted and all a[0..i] <= a[i+1..a.length-1] }
    i = i + 1;
    { inv }
  } // end of while loop
  { inv && i = a.length } => { a[0..a.length-1] sorted }
  { post: a[0..a.length-1] is sorted }
}
```

A *bag* or *multiset* is a set that allows duplicate elements. The next several questions concern a class called `IntBag` that implements a bag of integer values. The representation of data in this bag is an unordered linked list, where each node in the list contains an integer value and the number of times (> 0) that the integer value occurs in the bag. For instance, one possible representation of the multiset { 2, 3, 3, 2, 2, 1, 2 } is the following linked list:



Here is the beginning of the code for this class, showing the instance variable declarations (the rep) and a zero-argument constructor that initializes an empty `IntBag`. Also included are a pair of observer functions to return information about the contents of an `IntBag`.

```
public class IntBag {

  // Rep: an IntBag is stored as an unordered linked list of
  // nodes, each with a unique value and a positive count of
  // how many times that value occurs in the IntBag.

  private static class Link {
    int val;       // data in this node
    int ncopies;   // number of copies > 0 of val in this IntBag
    Link next;     // next node in the list or null if none
  }

  // List of values stored in this IntBag
  private Link elts;

  /** Construct a new empty IntBag. */
  public IntBag() {
    elts = null;
  }

  /** Return the size of this IntBag.
   *  (Example: for { 1 2 2 2 3 }, the value returned is 5.) */
  public int size() {
     // implementation omitted
  }

  /** Return number of copies of n in this IntBag or 0 if n
   *  is not contained in this IntBag. */
  public int contains(int n) {
      // implementation omitted
  }
```

Answer questions about this ADT on the next few pages. You may remove this page from the test for convenience if you wish.

Question 4. (12 points) (a) (3 points) Give a suitable abstract description of the class as would be written in the JavaDoc comment above the `IntBag` class heading.

An IntBag is a multiset of integers that is unordered and allows duplicate values. A typical IntBag would be {e1, …, en}. Examples are: { } (empty), { 1, -2, 3 }, and { 1, 3, -4, 3, 3, 2, 1 } which is the same as { 1, 1, 2, 3, 3, 3, -4 }.

(b) (5 points) Give a suitable Representation Invariant (RI) for this class.

elts == null, or
if elts != null:

- **elts references a single-linked list of Links with no cycles**
- **In the last node in the list p, p.next == null**
- **For every node p in the list, p.ncopies > 0**
- **If p and q are two nodes in the list and p != q, then p.val != q.val.**

(c) (4 points) Give a suitable Abstraction Function (AF) for this class relating the RI to the abstract value of a `IntBag`.

If elts == null, this IntBag represents an empty multiset { }
Otherwise elts references a linked list of nodes p_1, …, p_n.

- **Each node p_i represents the IntBag b_i = { v, v, v, … v } where each v = p_i.val and the number of elements (copies of p_i.val) in the IntBag is p_i.ncopies.**
- **The abstract value of the complete IntBag is the union of the individual node sets b_i, i.e., $b_1 \cup b_2 \cup \ldots \cup b_n$.**

Question 5. (10 points) Specification. One function we need in our `IntBag` ADT is a method to add an integer value to an `IntBag`, which might, of course, simply increase the number of copies of that value in the `IntBag`. For consistency with other Java collection add methods, this method should return true if it alters the collection, which will always happen for a multiset like our `IntBag`. We'll get to the implementation on the next page, but we first need to properly specify this method below.

Complete the JavaDoc comments for the `IntBag` `add` method to provide the most suitable specification. Leave any unneeded parts blank. There is space at the beginning (before @param) where you should write a summary description of `add` as is done at the beginning of every JavaDoc method specification. Hint: the answer probably won't need nearly this much space.

```
/**
 * Add an integer value to this
 *
 *
 * @param n value to be added
 *
 *
 * @requires
 *
 *
 * @modifies this
 *
 *
 * @effects this_post = this_pre U { n }
 *
 *
 * @throws
 *
 *
 * @returns true
 *
 *
 */
public boolean add(int n) {
   // Implementation omitted
}
```

Notes: Set notation was not required in the `@effects` clause as long as the description was clear. But the mathematical notation allows this to be done very concisely.

The `@returns` clause needs to say that the function always returns `true`, since it always modifies the `IntBag`.

Question 6. (15 points) Implementation. Give an implementation of method `add` for `IntBag` below using the linked-list representation described on previous pages. Your implementation should satisfy the specification of the method given in your answer to the previous question, and should be consistent with the rep invariant and abstraction functions you gave earlier.

```
public boolean add(int n) {

    // search for existing node containing n and
    // update its ncopies field if found
    Link curr = elts;
    while (curr != null) {
      if (curr.val == n) {
        // n found - increase number of copies & return
        curr.ncopies++;
        return true;
      } else {
        // advance to next node
        curr = curr.next;
      }
    }
    // n not found - add it to front of the list
    Link p = new Link();
    p.val = n;
    p.ncopies = 1;
    p.next = elts;
    elts = p;
    return true;
  }

}
```

Question 7. (9 points) Describe three separate, distinct "black box" tests for your `IntBag add` method. For each test give the input values and expected result(s). You do not need to write JUnit tests or other Java code. Reminder: there are `size()` and `contains()` observer methods defined for this class that might be useful.

There are obviously many, many good answers. Answers were judged on whether they contained good descriptions of inputs and expected outputs and whether they tested different things. Here are several possibilities:

Input: create an empty IntBag b.
Output: verify that b.size() == 0; verify b.contains(1) == 0.

Input: create IntBag b; b.add(1)
Output: verify b.size() == 1, b.contains(1) = 1, b.contains(0) == 0.

Input: create IntBag b; b.add(-1)
Output: verify b.size() == 1, b.contains(-1) = 1, b.contains(1) == 0.

Input: create IntBag b; b.add(1); b.add(2)
Output: verify b.size() == 2, b.contains(1) == 1, b.contains(2) == 1, b.contains(0)==0

Input: create IntBag b; b.add(1); b.add(1)
Output: verify b.size() == 2, b.contains(1) == 2, b.contains(2) == 0

Input: create IntBag b; b.add(1); b.add(2); b.add(2)
Output: verify b.size() == 3, b.contains(1) == 1, b.contains(2) == 2, b.contains(3)==0

Question 8. (15 points) Equality. Give an implementation of a proper `equals` method for class `IntBag`. You need to include the method heading, but do not need to write Javadoc comments. Two `IntBags` `a` and `b` are equal if they contain exactly the same elements – i.e., every integer that is found in one `IntBag` occurs exactly the same number of times in the other `IntBag`. Note: you are only being asked to implement `equals`. You are not being asked to provide a `hashCode` function.

Hint: the solution does not need to be all that long, particularly if you take advantage of some of the other methods in the class – in particular, the sample solution didn't require a nested loop. However, you may not assume that there are additional methods available in `IntBag` besides the ones already given previously or implemented in earlier problems, or that would be inherited by `IntBag` from class `Object`.

```
@Override
public boolean equals(Object other) {
  if (! (other instanceof IntBag ))
    return false;
  IntBag b = (IntBag) other;
  // if number of elements do not match then the
  // IntBags can't be equal
  if (b.size() != this.size())
    return false;
  // if sizes match, then each integer in this IntBag
  // must occur the same number of times in the other
  // one for the IntBags to be equal.
  Link p = elts;
  while (p != null) {
    if (p.ncopies != b.contains(p.val))
      return false;
    p = p.next;
  }
  return true;
}
```

Question 9. (10 points) Comparing specifications. This (final) question does not concern the `IntBag` ADT from the previous questions.

Here are parts of four possible specifications for a method that returns a prime number. (Recall that prime numbers have no integer divisors other than themselves and 1. The first few primes are 2, 3, 5, 7, 11, 13, … .)

A. @param x
@requires x is prime
@return the next prime number greater than x

B. @param x
@requires x > 0
@return the next prime number greater than x

C. @param x
@throws IllegalArgumentException if x < 3 or x is not prime
@return the next prime number greater than x

D. @param x
@requires x is prime
@return the next integer greater than x (i.e., x+1)

(a) List all of the specification that are stronger than A. __**B**__

(b) List all of the specification that are stronger than B. __**none**__

(c) List all of the specification that are stronger than C. __**none**__

(d) List all of the specification that are stronger than D. __**none**__

(e) Is it possible for a single method to satisfy A and B? (yes or no) __**yes**__

(f) Is it possible for a single method to satisfy A and C? (yes or no) __**no**__