CSE 331 20wi Midterm Exam 2/11/20 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 cannot happen and there are no fractional parts to numbers) and integer division is truncating division as in Java, i.e., 5/3 => 1.

Question 1. (5 points) (Forward reasoning) Starting with the given assertion, insert appropriate assertions in each blank line. You should simplify your final answers if possible, but do not weaken your final assertion in the process.

```
{ x >= 5 }
y = 3 * x;
{  x >= 5 ∧ y = 3*x  }
x = x + 2;
{  x >= 7 ∧ y = 3*(x-2)  }
y = y + 1;
{  x >= 7 ∧ y-1 = 3*(x-2)  } => { x>=7 ∧ y=3*x-5}
```

Question 2. (8 points) (Backward reasoning). Find the weakest precondition for the sequence of statements below to establish the given postcondition. Write appropriate assertions in each line and simplify your final answer if possible.

```
{ (x>0∧y>0) ∨ (x<=0∧x>1) } => {(x>0∧y>0) ∨ false}
                           => {(x>0 ∧ y>0)}
if (x > 0) {
   { y+1>0 ∧ y>0 } => {y>-1 ∧ y>0} => {y>0}
   x = y + 1;
   { x>0 ∧ y>0 }
} else {
   { x>0 ∧ x-1>0 } => {x>0 ∧ x>1} => {x>1}
   y = x - 1;
   { x>0 ∧ y>0  }
}
{ x > 0 && y > 0 }
```

The next several questions concern classes `Point` and `FinitePointBag`. The code for these classes is included on separate pages at the end of the exam. You should **remove those pages** from the exam and use them while answering these questions.

Question 3. (11 points) Let's look first at class `Point`. First, we need to complete the `equals` method. Here are several possible `return` statements that could appear in `equals` in place of the TODO comment.

```
E1:   return this.x == p.x;
E2:   return this.x == p.x && this.y == p.y;
E3:   return this.x + this.y == p.x + p.y;
E4:   return true;
```

Now here are several possibilities for completing method `hashCode`:

```
H1:   return 31;
H2:   return this.x;
H3:   return this.x + this.y;
H4:   return 31*this.x + this.y;
```

(a) (8 points) In the following table, put an X in the space if the given hash function from the above list is *consistent with* (i.e., satisfies the requirements for) the given equality relation from the first list. Your answer should ignore whether or not the `equals` relation does, in fact, define a correct equals method that satisfies the required properties for equality. Just mark an X where the `hashCode` is consistent with that particular definition of `equals`.

	E1	E2	E3	E4
H1	**X**	**X**	**X**	**X**
H2	**X**	**X**		
H3		**X**	**X**	
H4		**X**		

(b) (3 points) Given the above possibilities for `equals` (E1-E4) and `hashCode` (H1-H4), what choice for `equals` and for `hashCode` would be best for a typical 2-D Point class like this one? Pick one of H1-H4 and one of E1-E4 and write your choice in the blank spaces. You do not need to justify your answer.

`equals` (E1-E4): **E2**

`hashCode` (H1-H4): **H4**

Note: H4 is somewhat better than H3 since it is more likely to give different values for different `Points`: for instance, (x=17,y=42) and (x=42,y=17). See *Effective Java* Item 11 for more.

Question 4. (10 points) ADT/RI/AF. Now on to the `FinitePointBag` class. The first thing we need to do is be sure we understand the ADT that this class represents. To do that we need to provide an abstract description of the class, a rep invariant, and an abstraction function. Your answers should be consistent with the given instance variables and code in `FinitePointBag`.

(a) (3 points) Give a suitable description of the `FinitePointBag` ADT and its abstract value(s), as would normally appear in the JavaDoc comment right above the class definition. (Hint: the answer might be quite short.)

A `FinitePointBag` is a mutable, unordered collection of points {p1, p2, …, pn} possibly containing duplicates, with a finite capacity.

(b) (4 points) Give a suitable representation invariant (RI) for `FinitePointBag`.

`points != null && 0 <= size <= points.length` && all entries in `points[0..size-1]` are not equal to `null`.

(c) (3 points) Give a suitable abstraction function (AF) for `FinitePointBag`.

`points.length` is the capacity of `this`; `size` is the number of `Points` in `this`, and `points[0..size-1]` contains references to the `Point` objects in `this`.

Question 5. (12 points) As usual, whoever writes these exams doesn't provide proper specifications for things. Below, fill in a correct CSE 331-style specification for the `contains` and the `removeAll` methods of class `FinitePointBag`. Your answer should be consistent with the given code and what it does when executed. For CSE 331-specific custom tags, you can write `@spec.xyz` or just `@xyz` – whichever you prefer. Hint: remember that `assert` statements are for debugging, and are not necessarily executed in production code. Your specifications should be consistent with that knowledge.

```
/** Return true if p is contained in this, otherwise false
 *
 * @param p the Point to search for in this
 *
 * @requires p != null
 *
 * @return true if p is an element of this, false otherwise
 *
 *
 *
 *
 *
 *
 *
 *
 */
public boolean contains(Point p) { ... }

/** Remove all copies of a point from this FinitePointBag
 *
 * @param p the Point to remove from this
 *
 * @requires p != null
 *
 * @modifies this
 *
 * @effects all items that are equal to p are removed from this
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */
public void removeAll(Point p) { ... }
```

Question 6. (16 points) Proving code (the loop proof question – you knew it would be somewhere in here!). We think method `removeAll` in `FinitePointBag` is correct, but we'd like to prove that it actually removes all of the objects equal to `p` from the array. To keep the problem simple, you do not need to keep track of history variables or other things that would be needed to prove that the values copied are the original values in the array, just show that the `FinitePointBag` does not contain any copies of `p` after the method executes. Also, you can write `{inv}` in your proof to reference the loop invariant without having to re-write all the details each time.

```
// remove all copies of p from this FinitePointBag
public void removeAll(Point p) {
  assert p != null;
  { p != null and RI - rep invariant - here }
  int i, k;
  i = 0;
  k = 0;
  {inv: points[0..i-1] are points not equal to p
        && points[k..size-1] are elements not yet examined* }
  while (k != size) {
    { inv && k != size }
    if (!points[k].equals(p)) {
      { points[0..i-1] != p && points[k] != p
              && points[k+1..size-1] not examined }
      points[i] = points[k];
      { points[0..i] != p && points[k+1..size-1] not examined }
      i = i + 1;
      { points[0..i-1]!=p && points[k+1..size-1] not examined }
    } else { // no code - place for any needed assertions
      { points[0..i-1]!=p && points[k+1..size-1] not examined }
    } // end if
    { points[0..i-1]!=p && points[k+1..size-1] not examined }
    k = k + 1;
    { inv }
  } // end while
  {inv && k = size}=>{points[0..i-1]!=p && all points examined}
  size = i;
  { points[0..size-1] != p }   (which reestablishes part of the rep invariant)
}
```

***Grading note: we deliberately simplified the problem so that the proof only needed to show that elements not equal to `p` remain in the `FinitePointBag` when this method terminates, and not also prove that all the other values that were in the array originally remain at the end. It's possible to prove what was asked without referring to the unexamined part of the array (`points[k..size-1]`) in the proof, and solutions that ignored that part of the array received credit. It is included in the above invariant to help show the operation of the algorithm.**

Question 7. (12 points, 4 each) Testing. Although we now have good confidence that `removeAll` works as expected because of our proof, we also need to test it. Describe three distinct black-box tests that could be used to verify that the `removeAll` method works properly. Each test description should describe the test input and expected output. For full credit each test should be different in some significant way from the other tests (think about boundary conditions and subdomains, etc.). You **should not** provide JUnit or other code, just a clear, precise description of each test, and your descriptions should be a few lines each, at most.

There are many possibilities. Here are a few. For all tests below, assume we create ahead of time two distinct points (using @Before if this were a JUnit test):

Point p1 = new Point(1,2);
Point p2 = new Point(3,4);

1. **Input/setup: create a new empty FinitePointBag g with capacity 5.**
   Execute g.remove(p1);
   Expected output: g.size() = 0, g.capacity = 5, g.contains(p1) = false,
   g.contains(p2) = false

2. **Input/setup: create a new empty FinitePointBag g with capacity 5.**
   Execute g.add(p1); g.removeAll(p1);
   Expected output: g.size() = 0, g.capacity() = 5, g.contains(p1) = false,
   g.contains(p2) = false

3. **Input/setup: create a new empty FinitePointBag g with capacity 5.**
   Execute g.add(p1); g.removeAll(p2);
   Expected output: g.size() = 1, g.capacity() = 5, g.contains(p1) = true,
   g.contains(p2) = false

4. **Input/setup: create a new empty FinitePointBag g with capacity 5.**
   Execute g.add(p1); g.add(p2); g.add(p1); g.removeAll(p2);
   Expected output: g.size() = 2, g.capacity() = 5, g.contains(p1) = true,
   g.contains(p2) = false

5. **Input/setup: create a new empty FinitePointBag g with capacity 5.**
   Execute g.add(p1); g.add(p2); g.add(p1); g.removeAll(p1);
   Expected output: g.size() = 1, g.capacity() = 5, g.contains(p1) = false,
   g.contains(p2) = true

etc., etc., etc…

Question 8. (14 points, 2 each) Overloading, overriding, and `equals`. This question refers to code that uses the `Thing` and `Blob` classes printed on the last separate code page at the end of the exam. Detach that page and use it to answer this question.

The following main program uses the `Thing` and `Blob` classes. One line of code is omitted at the end, and supplied below. All of the code compiles and runs without errors.

```
  public static void main(String[] args) {
    Thing ta = new Thing(1);
    Thing tb = new Thing(1);
    Blob ba = new Blob(3,7);
    Blob bb = new Blob(2,4);
    Object ota = ta;
    Object otb = tb;
    Object oba = ba;
    Object obb = bb;
    _____________________ ;  // insert code from below here
  }
}
```

For each line of code below, indicate what happens if it is inserted by itself at the end of the `main` method above and then the program is executed. For each one, circle the correct answers to indicate which method is called during execution (`Object.equals`, `Thing.equals`, or `Blob.equals`) and whether the method call returns `true` or `false`. Circle only the class of the first `equals` method called, even if that method calls another one.

(a) `ta.equals(tb);`

`equals` method executed: `Object` (`Thing`) `Blob` Result: (`true`) `false`

(b) `ta.equals(bb);`

`equals` method executed: `Object` (`Thing`) `Blob` Result: `true` (`false`)

(c) `ota.equals(otb);`

`equals` method executed: (`Object`) `Thing` `Blob` Result: `true` (`false`)

(d) `ba.equals(bb);`

`equals` method executed: `Object` `Thing` (`Blob`) Result: `true` (`false`)

(e) `ba.equals(tb);`

`equals` method executed: `Object` `Thing` (`Blob`) Result: `true` (`false`)

(f) `ota.equals(tb);`

`equals` method executed: (`Object`) `Thing` `Blob` Result: `true` (`false`)

(g) `oba.equals(ba);`

`equals` method executed: (`Object`) `Thing` `Blob` Result: (`true`) `false`

Question 9. (10 points) Comparing specifications. Here are four possible specifications for a method that allocates an `int` array whose length is given by the parameter *n*, and returns a reference to the newly allocated array.

A.
```
@param n
@return a new int array with n elements if n > 0,
     otherwise return null
```

B.
```
@param n
@requires n > 0
@return a new int array with n elements
```

C.
```
@param n
@requires n >= 0
@return a new int array with n elements if n > 0,
     otherwise return null
```

D.
```
@param n
@return a new int array with n elements
@throws IllegalArgumentException if n <= 0
```

(a) List all of the specification that are stronger than A (do not include A): **none**

(b) List all of the specification that are stronger than B (do not include B): **A, C, D**

(c) List all of the specification that are stronger than C (do not include C): **A**

(d) List all of the specification that are stronger than D (do not include D): **none**

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

(f) Is it possible for a single implementation to satisfy both B and C? (yes or no) **yes**

Question 10. (2 free points) (All reasonable answers receive the points. All answers are reasonable as long as there is an answer. ☺)

(a) (1 point) What question were you expecting to appear on this exam that wasn't included?

Why?

(b) (1 point) Should we include that question on the final exam? (circle or fill in)

Yes

No

Heck No!!

$!@$^*% No !!!!!

No opinion / don't care

None of the above. My answer is **Why not?**

All answers received full credit (i.e., the 2 free points).

Code for classes `Point` and `FinitePointBag`. Remove these pages from the exam and return them for recycling when you are done. This code is used in several questions in the exam. Some parts of the code are incomplete or missing, and the questions address those issues. Except for the missing pieces, all of the code here does compile and work as intended.

Class **`Point`**: an immutable 2-D point on the plane with rectangular coordinates.

```
public class Point {
  // instance variables -- coordinates
  private final int x, y;

  //constructor
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }

  // observer methods
  public int getX() { return this.x; }
  public int getY() { return this.y; }

  @Override
  public boolean equals(Object o) {
    if (! (o instanceof Point)) {
      return false;
    }
    Point p = (Point) o;
    return /* TODO: figure out what to put here */;
  }

  @Override
  public int hashCode() {
    return /* TODO: figure out what to put here also */;
  }
}  // end class Point
```

Class **`FinitePointBag`**: a mutable unordered collection of `Point` objects, possibly containing duplicates, and with a finite capacity.

```
public class FinitePointBag {
  // instance variables: uses an array and size instead of a List
  private Point[] points;
  private int size;

  // construct a new FinitePointBag with given capacity, which
  // cannot be negative
  public FinitePointBag(int maxItems) {
    points = new Point[maxItems];
    size = 0;
  }
```

Class **FinitePointBag**: (continued)

```
  // = capacity of this
  public int capacity() { return points.length; }

  // = current number of items in this
  public int size() { return size; }

  // add a new point if space available; return true if succeeded
  public boolean add(Point p) {
    assert p != null : "attempt to add null to a FinitePointBag";
    if (size < capacity()) {
      points[size] = p;
      size++;
      return true;
    } else {
      return false;
    }
  }

  // return true if p is contained in this, otherwise false
  public boolean contains(Point p) {
    assert p != null;
    for (int k = 0; k < size(); k++) {
      if (points[k].equals(p)) {
        return true;
      }
    }
    return false;
  }

  // remove all copies of p from this
  public void removeAll(Point p) {
    assert p != null;
    int i, k;
    i = 0;
    k = 0;
    while (k != size) {
      if (!points[k].equals(p)) {
        points[i] = points[k];
        i = i + 1;
      }
      k = k + 1;
    }
    size = i;
  }

}  // end of FinitePointBag
```

Code for `Thing`/`Blob` classes used in `equals` overloading/overriding question. Remove this page from the exam and return it for recycling when you are done.

Notice that the parameters to the `equals` methods have unusual types (possibly not what they should be, but the question is about what happens given this code as written).

```
/** A Thing object holds an integer value. */
class Thing {
  private int t;

  public Thing(int t) {
    this.t = t;
  }

  public boolean equals(Thing o) {
    if (! (o instanceof Thing)) {
      return false;
    }
    Thing t = (Thing) o;
    return this.t == t.t;
  }
}

/** A Blob is a Thing with an additional integer value. */
class Blob extends Thing {
  private int b;

  public Blob(int t, int b) {
    super(t);
    this.b = b;
  }

  public boolean equals(Thing o) {
    if (! (o instanceof Thing)) {
      return false;
    }
    if (! (o instanceof Blob)) {
      return super.equals(o);
    }
    Blob b = (Blob) o;
    return super.equals(b) && this.b == b.b;
  }
}
```