Name: _____

Pennkey (eg, sweirich): _____

CIS 120 Final Exam May 8, 2012

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

Signature: _____ Date: _____

1	/12
$2(a, b, c)$	/8
$2(d, e)$	/10
$2(f)$	/10
3	/5
4	/20
5	/18
6	/10
$7(a)$	/12
$7(b, c)$	/15
Total	/120

- Do not begin the exam until you are told to do so.
- You have 120 minutes to complete the exam.
- There are 120 total points.
- There are 17 pages in this exam.
- Make sure your name and Pennkey (a.k.a. username) are on the top of this page.

1. True or False (12 points)

a. [T] F Once an array is created, its size may never change.

b. [T] F Every variable declared in a Java program must have a type.

c. T [F] In a variable declaration `A x = new B()`, the type `A` must be a subtype of `B`.

d. T [F] In Java, variables of primitive type (like ints) must be initialized with the **`new`** command.

e. [T] F Static methods can be invoked directly using the name of a class, e.g. `Math.cos(80)`.

f. [T] F In Java, the special value **`null`** is used in situations where an OCaml programmer would use an `option` type.

g. T [F] The value **`null`** can be assigned to a variable of any type in Java.

h. [T] F Java `String` objects are immutable. Once created, their size and contents may never change.

i. T [F] Suppose `s` is a variable of type `String`. The Java expression `s.equals(s)` always returns **`true`**.

j. [T] F Suppose `s` is a variable of type `String`. The Java expression `s == s` always returns **`true`**.

k. T [F] Suppose we have the declaration `String s = "CIS 120";` and there are no assignments to the variable `s`. The Java expression `s == "CIS 120"` always returns **`true`**.

l. [T] F Suppose we have the declaration `String s = "CIS 120";` and there are no assignments to the variable `s`. The expression `s.equals("CIS 120")` always returns **`true`**.

2. Program Design (28 points total)

In this problem you will use the test-driven design methodology to implement a Java class that implements finite maps from `int` keys to `int` values. Pages 13 and 14 of the reference handout include the `IntMap` interface and part of the `ArrayIntMap` implementation. Note that the implementation is for *bounded* maps, with `int` keys in the range 0 to `bound-1`.

(0 points) **a.** The first step of the program design process is to understand the problem. Your experience with finite maps from homework 3 and 6 will help you here. Although there is nothing to write for this step, make sure you **read over the code in the reference appendix now.**

(4 points) **b.** The next step is to *define the interface*. The interface has already been defined for you in this problem. However, the interface to the `get` method is *incomplete*. It doesn't say anything about exceptions! Based on the *implementation*, list *all* of the exceptions that this method throws and the situations that would trigger those exceptions.

The method throws an `ArrayIndexOutOfBoundsException` *when given an index less than 0 or greater than the length of elt. The method throws a* `NullPointerException` *when there is no mapping for the element.*

(4 points) **c.** However, the interface that you described above isn't consistent with the treatment of exceptions by the other methods in the class. Rewrite the `get` method so that it *only* ever throws an `IllegalArgumentException`.

```
public int get(int k) {
     checkBound(k);
     if (elts[k] != null)
       return elts[k].intValue();
     else {
       throw new IllegalArgumentException("Key not found.");
     }
   }
```

(4 points) **d.** The implementor of the `ArrayIntMap` class intends the code to enforce the following invariant about the instance variables of the class:

`size` = number of nonnull elements of `elts`

Briefly explain, using one or two sentences or a code example, why the code *cannot* rely on this invariant if the instance variable `elts` were declared **`public`**.

If the `elts` *instance variable were public, then the object cannot encapsulate this state. Any users of an object of this class could modify the array without using the methods of the object. This means that they could change a value in the array to null without modifying the size.*

(6 points) **e.** The third step is to *write test cases*. Complete three *distinct* test cases for the `remove` method (whose implementation was omitted). Each test case should test a different aspect of `remove`'s functionality. **Add a comment to each case saying what aspect is tested.** You may insert additional uses of `assertEquals` or `assertTrue`.

```
//
@Test
public void testRemove0() {
   ArrayIntMap s = new ArrayIntMap(3);
   assertFalse(s.remove(0));
   assertEquals(0, s.size());
}

//
@Test
public void testRemove1() {
   ArrayIntMap s = new ArrayIntMap(3);
   s.put(0,3);
   assertTrue(s.remove(0));
   assertEquals(0, s.size());
}

@Test
public void testRemove2() {
   ArrayIntMap s = new ArrayIntMap(3);
   s.put(0,3);
   s.put(1,3);
   assertTrue(s.remove(0));
   assertEquals(1, s.size());
}

// remove nonexisting element from
// nonempty map
@Test
public void testRemove3() {
   ArrayIntMap s = new ArrayIntMap(3);
   s.put(0,1);
   assertFalse(s.remove(1));
   assertEquals(1, s.size());
}
```

(10 points) **f.** Complete the implementation of the `remove` method. It should never throw an exception other than the one listed below, and it should preserve the `ArrayIntMap` invariant. NOTE: you *do not* need to use **`try...catch`**.

```
// Removes the mapping for a key from this map if it is present.
// Returns whether the map originally had a mapping for that key.
// Throws IllegalArgumentException if the key is not in bounds.
public boolean remove(int k) {
  checkBound(k);
  if (elts [k] == null) return false;
  elts[k] = null;
  size--;
  return true;
}
```

3. Object Encoding (5 points)

Translate the following OCaml definitions to a Java class definition.

```
type counter_state = { mutable cnt : int }

type counter = { incr : unit -> unit; get : unit -> int }

let make_counter () : counter =
  let ths : counter_state = { cnt = 0 } in
    { incr = (fun () -> ths.cnt <- ths.cnt + 1);
      get = (fun () -> ths.cnt) }
```

Answer:

```
class Counter {
  private int cnt = 0;
  public void incr() { cnt++; }
  public int get() { return cnt; }
}
```

4. Tree recursion (20 points)

Recall our type of integer carrying binary search trees.

```
type tree =
  | Empty
  | Node of tree * int * tree
```

Recall also the *correct* definition of the insertion function for binary search trees.

```
let rec insert (t:tree) (n:int) : tree =
 begin match t with
   | Empty -> Node (Empty, n, Empty)
   | Node(lt, x, rt) ->
       if n < x then
          Node(insert lt n, x, rt)
       else if n > x then
          Node(lt, x, insert rt n)
       else t
 end
```

We know that the insert code is correct because we *tested* it. In particular, we wrote the following test cases (before implementing the insert method)...

```
(* Some trees to work with *)
let t1 : tree = Node (Empty, 2, Empty)
let t2 : tree = Node (Node (Empty, 1, Empty), 2, Empty)
let t3 : tree = Node (Empty, 2, Node (Empty, 3, Empty))

(* Some test cases *)
let test1 () : bool = (insert Empty 2 = t1)
let test2 () : bool = (insert t1 1 = t2)
let test3 () : bool = (insert t1 3 = t3)

(* Actually running the tests *)
;; run_test "insert empty" test1
;; run_test "insert smaller" test2
;; run_test "insert larger" test3
```

...and they all passed.

```
Running: insert empty ... Test passed!
Running: insert smaller ... Test passed!
Running: insert larger ... Test passed!
```

However, often things don't always work out so smoothly, as you are well aware. Page 15 of the reference appendix gives **five** *incorrect* variants of the insertion function. For each of the following testing outputs below, circle the variant (or variants) that produced that result. There should be exactly five circles on this page.

a.
```
Running: insert empty ... Test passed!
Running: insert smaller ...
Test error: `insert smaller` reported `Stack overflow`
Running: insert larger ... Test passed!
```

1 2 3 4 *5*

b.
```
Running: insert empty ... Test passed!
Running: insert smaller ...
Test failed: insert smaller
Running: insert larger ... Test passed!
```

1 *2* 3 4 5

c.
```
Running: insert empty ... Test passed!
Running: insert smaller ...
Test failed: insert smaller
Running: insert larger ...
Test failed: insert larger
```

1 2 *3* *4* 5

d.
```
Running: insert empty ...
Test failed: insert empty
Running: insert smaller ...
Test failed: insert smaller
Running: insert larger ...
Test failed: insert larger
```

1 2 3 4 5

5. Subtyping and Collections (18 points)

Consider the the Java classes and interfaces excerpted from the Collections Framework, as shown in the reference Appendix on page 16. For each code snippet below, indicate what result will get printed to the console, or mark "Ill typed" if the snippet has a type error.

a.
```
List<Integer> lst = new List<Integer>();
lst.add(1);
lst.add(1);
System.out.println(lst.size());
```

0 1 2 *Ill typed*

b.
```
Set<Integer> s = new TreeSet<Integer>();
s.add(1);
s.add(1);
System.out.println(s.size());
```

0 1 2 Ill typed

c.
```
Set<List<Integer>> s = new TreeSet<LinkedList<Integer>>();
List<Integer> lst = new LinkedList<Integer>();
s.add(lst);
System.out.println(s.size());
```

0 1 2 *Ill typed*

d.
```
Collection<Integer> c = new TreeSet<Integer>();
c.add(1);
c.add(1);
System.out.println(c.size());
```

0 1 2 Ill typed

e.
```
Collection<Integer> c = new LinkedList<Integer>();
c.add(1);
c = new TreeSet<Integer>();
c.add(2);
System.out.println(c.size());
```

0 1 2 Ill typed

f.
```
Collection<Integer> c = new LinkedList<Integer>();
c.add(1);
c.add(1,2);
System.out.println(c.size());
```

0 1 2 *Ill typed*

6. GUI and OO Program Style (10 points)

Consider the following code snippets from the drawing example presented in class. The interface `Shape` describes something that can be drawn on the canvas, the field `shapes` (a member of the class `DrawingExample`) stores a list of shapes to draw, and the method `paintComponent` (a member the class `DrawingExampleCanvas` an inner class of `DrawingExample`) actually draws the shapes in the window.

```
interface Shape { public void draw(Graphics gc); }
private List<Shape> shapes = ...;
public void paintComponent(Graphics gc) {
  for (Shape s : shapes) {
    s.draw(gc);
  }
}
```

The two variants below, although they behave correctly, are written with *worse style* than the version above. Why? Explain the problem using one or two sentences.

a.
```
interface Shape { public void draw(Graphics gc); }
private LinkedList<Shape> shapes = ...;
public void paintComponent(Graphics gc) {
  for (int i=0; i<shapes.size(); i++) {
    shapes.get(i).draw(gc);
  }
}
```

The biggest problem is the use of get with a linked list. This dramatically decreases the running time. Can also comment that for-each loops are better style because they don't have indexing errors. Can also comment that using a class (LinkedList) as the type of shapes is less flexible than using the interface (List).

b.
```
enum Shape { CIRCLE; SQUARE }
private List<Shape> shapes = ...;
public void paintComponent(Graphics gc) {
  for (Shape s : shapes) {
    if (s == Shape.CIRCLE) {
      gc.drawCircle(...);
    } else if (s == Shape.SQUARE) {
      gc.drawSquare(...);
    }
  }
}
```

This code is less extensible than the previous version. The paint component method must be modified when each new shape is added to the program. In contrast, the version above uses dynamic dispatch to determine how shapes should be drawn.

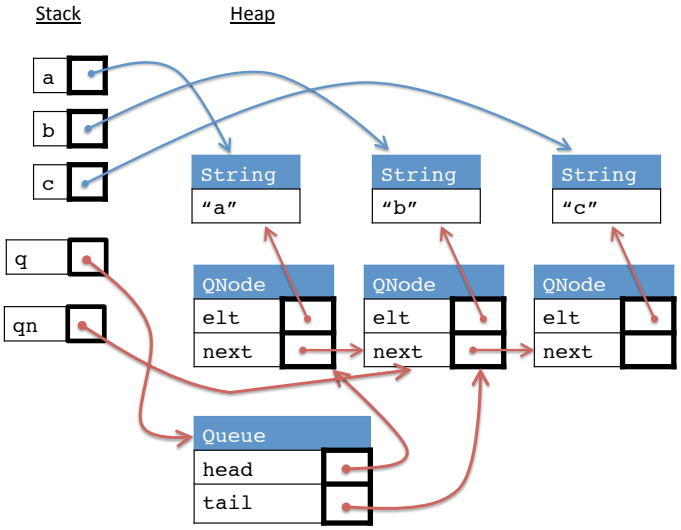
7. Java ASM and Queue (27 points total)

This problem concerns an implementation of a singly-linked `Queue`, as shown on page 17.

(12 points) **a.** Draw the state of the Java ASM at the point marked "HERE" after running the code below. Assume this code is defined in the `Queue` class, so that it has access to the private members.

The state of the ASM at the point marked "START" has been given to you. Your answer should extend this drawing. Do *not* show the class table or the workspace, only show the stack and heap. Be sure to label the *dynamic class* of each object in the heap.

```
String a = "a"; String b = "b"; String c = "c";
// START
Queue<String> q = new Queue<String>();
q.enq(a);
q.enq(b);
q.enq(c);
QNode<String> qn = q.head;
qn = qn.next;
q.tail = qn;
// HERE
```



(3 points) **b.** Does the queue `q` still satisfy the queue invariants after this code has executed? Write YES or NO. *NO!*

(12 points) **c.** Add the method `rotate(int i)` to the `Queue` class. This method moves all elements after position `i` to the front of the queue, preserving their order. For example, if the `Queue q` contains the elements `1, 2, 3, 4` in order from head to tail, then `q.rotate(1)` will update `q` to contain the elements `3, 4, 1, 2`. Similarly, if the `Queue q` contains the elements `1, 2, 3, 4` in order from head to tail, then `q.rotate(0)` will update `q` to contain the elements `2, 3, 4, 1`. Calling rotate with the index of the last element of the queue should not change the queue.

Your method should reuse the `QNode` objects from the original queue; you cannot use the `deq` and `enq` operations—manipulate the pointers directly. Be sure to preserve the queue invariants.

You may assume that the position `i` is a valid index into the queue.

```
// moves all elements after position i to the front of the queue, preserving thei
// assumes that i is less than the length of the queue
public void rotate(int i) {
  QNode<E> curr = head;
  // find the correct queue node
  for (int j=0; j<i; j++) {
    curr = curr.next;
  }
  // move everything after curr to the front
  if (curr != tail) {
    tail.next = head;
    tail = curr;
    head = curr.next;
    curr.next = null;
  }
}
```

Reference Appendix

Make sure all of your answers are written in your exam booklet. These pages are provided for your reference, we will not grade any answers written in this section.

IntMap interface for Problem 2

```
interface IntMap {

  public void put(int k, int v);
  // Associates the specified value with the specified key in this map
  // Throws IllegalArgumentException if the key is not in bounds

  public int  get(int k);
  // Returns the value to which the specified key is mapped

  public int  bound();
  // Returns one more than the largest potential key that
  // may be stored in the map.

  public int  size();
  // Returns the number of key−value mappings

  public boolean containsKey(int k);
  // Returns true if this map contains a mapping for the specified key
  // Throws IllegalArgumentException if the key is not in bounds

  public boolean remove(int k);
  // Removes the mapping for a key from this map if it is present
  // Returns whether the map originally had a mapping for that key
  // Throws IllegalArgumentException if the key is not in bounds
}
```

ArrayIntMap implementation for Problem 2

```
class ArrayIntMap implements IntMap {

  private Integer[] elts;
  private int   size;

  public ArrayIntMap(int bound) {
    elts = new Integer[bound];
    size = 0;
  }

  // A helper function that throws an exception when a given
  // key is not in bounds
  private void checkBound(int k) {
    if (k < 0 || k >= elts.length) {
      throw new IllegalArgumentException("Key out of range.");
    }
  }

  public void put(int k, int v) {
    // ensure that the key is in bounds
    checkBound(k);
    // if there isn't a mapping for the key k, increment the size
    if (elts[k] == null) {
      size++;
    }
    elts[k] = new Integer(v);
  }

  public int get(int k) {
    // recall that the intValue method converts an Integer to an int
    return elts[k].intValue();
  }

  public int size() { return size; }

  public int bound() { return elts.length; }

  public boolean containsKey(int k) {
    checkBound(k);
    return elts[k] != null;
  }

  public boolean remove(int k) {
    ...
  }
```

BST insert variants for Problem 4

1. ```
   let rec insert (t:tree) (n:int) : tree =
     begin match t with
       | Empty -> Empty
       | Node(lt, x, rt) ->
           if n < x then
              Node(insert lt n, x, rt)
           else if n > x then
              Node(lt, x, insert rt n)
           else t
     end
   ```
2. ```
   let rec insert (t:tree) (n:int) : tree =
     begin match t with
       | Empty -> Node (Empty, n, Empty)
       | Node(lt, x, rt) ->
           if n < x then
              insert lt n
           else if n > x then
              Node(lt, x, insert rt n)
           else t
     end
   ```
3. ```
   let rec insert (t:tree) (n:int) : tree =
     begin match t with
       | Empty -> Node (Empty, n, Empty)
       | Node(lt, x, rt) ->
           if n < x then
              Node(insert lt x,n, rt)
           else if n > x then
              Node(lt, n, insert rt x)
           else t
     end
   ```
4. ```
   let rec insert (t:tree) (n:int) : tree =
     begin match t with
       | Empty -> Node (Empty, n, Empty)
       | Node(lt, x, rt) ->
           if n > x then
              Node(insert lt n, x, rt)
           else if n > x then
              Node(lt, x, insert rt n)
           else t
     end
   ```
5. ```
   let rec insert (t:tree) (n:int) : tree =
     begin match t with
       | Empty -> Node (Empty, n, Empty)
       | Node(lt, x, rt) ->
           if n < x then
              Node(insert (Node (lt,x,rt)) n, x, rt)
           else if n > x then
              Node(lt, x, insert rt n)
           else t
     end
   ```
   ```

Excerpt from the Collections Framework for Problem 5

```
interface Collection<E> {

  public int size();
  // Returns the number of elements in this collection .

  public void add (E o);
  // Ensures that this collection contains the specified element

  ...

}

interface List<E> extends Collection<E> {

  public void add(int index, E o);
  // Inserts the specified element at the specified position in this list

  public E   get(int index);
  // Returns the element at the specified position in this list

  ...

}

interface Set<E> extends Collection<E> { }

class LinkedList<E> implements List<E> {
   ...
   // constructor
   public LinkedList() { ... }
   // methods
   public int size() { ... }
   public void add(E o) { ... }
   public void add(int index, E element) { ... }
   public E get(int index) { ... }
   ...
}
class TreeSet<E> implements Set<E> {
   ...
   // constructor
   public TreeSet() { ... }
   // methods
   public int size() { ... }
   public void add(E o) { ... }
   ...
}
```

Queue implementation for Problem 7

```
import java.util.NoSuchElementException;

class QNode<E> {
   public E v;
   public QNode<E> next;
   QNode(E elt, QNode<E> n) {
      v = elt;
      next = n;
   }
}

class Queue<E> {
   private QNode<E> head;
   private QNode<E> tail;

   public Queue() {
      head = null;
      tail = null;
   }

   public boolean isEmpty() {
      return head == null;
   }

   public void enq(E elt) {
      QNode<E> newNode = new QNode<E>(elt, null);
      if (tail == null) {
         head = newNode;
         tail = newNode;
      } else {
         tail.next = newNode;
         tail = newNode;
      }
   }

   public E deq() {
      if (head == null) {
         throw new NoSuchElementException();
      }
      E x = head.v;
      head = head.next;
      if (head == null) {
         tail = null;
      }
      return x;
   }
}
```