CIS 120 Final Exam 7 May 2015

Name (printed): _____

Pennkey (login id): _____

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	/9
2	/17
3	/10
4	/10
5	/10
6a	/5
6b	/12
6c	/12
7	/15
Total	/100

- Do not begin the exam until you are asked to do so.
- You have 120 minutes to complete the exam.
- There are 100 total points.
- There are 14 pages in this exam, plus an Appendix. Do not write any answers in the Appendix.

1. Tree recursion (9 points)

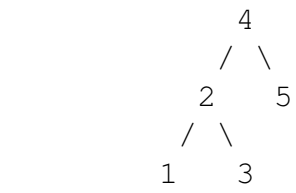
Recall a type of integer carrying binary search trees in OCaml.

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

An example of such a tree is given by:

```
let t : tree =
  Node(Node(Node(Empty, 1, Empty), 2, Node(Empty, 3, Empty)),
       4,
       Node(Empty, 5, Empty))
```

We draw it like this (omitting the `Empty` values):

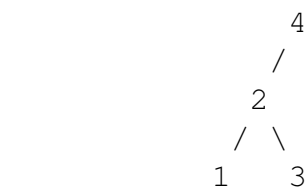


Each code snippet below is a definition of an `insert` function for binary search trees. Some or all of these definitions are *incorrect*. Circle the result of calling each of these functions with the tree above.

a. Circle the result of `insert t 6` with the following definition.

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

(a)



(b)

(c)

6

(d)

(Empty)

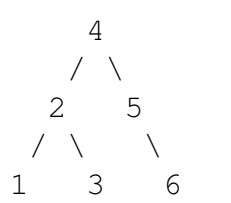
(e)

Infinite Loop
(stack overflow)

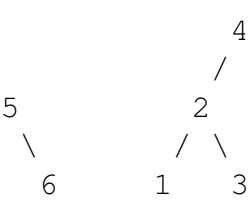
b. Circle the result of `insert t 6` with the following definition.

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

(a)



(b)



(c) 6

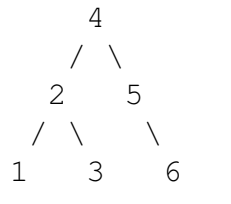
(d) (Empty)

(e) Infinite Loop (stack overflow)

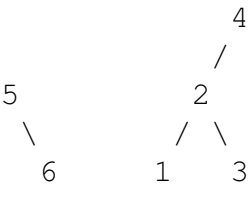
c. Circle the result of `insert t 6` with the following definition.

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

(a)



(b)



(c) 6

(d) (Empty)

(e) Infinite Loop (stack overflow)

2. Mutable data structures in OCaml (17 points total)

This problem concerns an implementation of *mutable* binary search trees, as shown in Appendix A.

a. (1 point) What is the result of evaluating the expression

```
leaf 3 == leaf 3
```

Circle one: **`true`** **`false`** *infinite loop*

b. (1 point) What is the result of evaluating the expression

```
leaf 3 = leaf 3
```

Circle one: **`true`** **`false`** *infinite loop*

c. (1 point) What is the result of evaluating the expression

```
let n = leaf 3 in
  Some n == Some n
```

Circle one: **`true`** **`false`** *infinite loop*

d. (2 points) Suppose the following code were placed on the workspace for the OCaml ASM.

```
let t1 : int tree =
 let no2 = Some (leaf 1) in
 let no3 = Some {left = None; v=2; right = no2} in
 { root = no3 }
```

Which picture (a) - (i) in Appendix B depicts the heap at the end of this execution?

Write the corresponding letter here: _____.

e. (2 points) Now suppose the following code were placed on the workspace for the OCaml ASM.

```
let t2 : int tree =
 let no2 = Some (leaf 1) in
 let no3 = Some {left = no2; v=2; right = no2} in
 { root = no3 }
```

Which picture (a) - (i) in Appendix B depicts the heap at the end of this execution?

Write the corresponding letter here: _____.

f. (2 points) Now suppose the following code were placed on the workspace for the OCaml ASM.

```
let t3 : int tree =
  let no2 = Some (leaf 1) in
  let n = {left = no2; v=2; right = no2} in
  let no3 = Some n in
  n.left <- no3;
  { root = no3 }
```

Which picture (a) - (i) in Appendix B depicts the heap at the end of this execution?

Write the corresponding letter here: _____.

g. (2 points) Now consider the `inorder` function, shown in Appendix A, which converts the tree to a list using an inorder traversal.

What is the result of `inorder t1`, where `t1` is the tree defined in part (d)? Note that `t1` may not be a valid binary search tree.

Circle the correct answer.

i. `[]`

ii. `[1;2]`

iii. `[2;1;2]`

iv. `[2;1]`

v. `[1;2;1]`

vi. *infinite loop*

h. (2 points) What is the result of `inorder t2`, where `t2` is the tree defined in part (e)? Note that `t2` may not be a valid binary search tree.

Circle the correct answer.

i. `[]`

ii. `[1;2]`

iii. `[2;1;2]`

iv. `[2;1]`

v. `[1;2;1]`

vi. *infinite loop*

i. (2 points) What is the result of `inorder t3`, where `t3` is the tree defined in part (f)? Note that `t3` may not be a valid binary search tree.

Circle the correct answer.

i. `[]`

ii. `[1;2]`

iii. `[2;1;2]`

iv. `[2;1]`

v. `[1;2;1]`

vi. *infinite loop*

j. (2 points) Finally consider the `insert` function, shown in Appendix A, which inserts a new value into the tree.

What is the result of `inorder (insert 3 t2)`, where `t2` is the tree defined above? Note that `t2` may not be a valid binary search tree.

Circle the correct answer.

i. `[]`

ii. `[1;2;3]`

iii. `[1;2;1;3]`

iv. `[2;1;3]`

v. `[1;3;2;1;3]`

vi. Stack overflow

3. Java Dynamic Dispatch (10 points)

Consider the class definitions shown in Appendix C. For each of the unit tests below, circle whether the test passes, fails with an assertion error, produces a null pointer exception (NPE) or produces a stack overflow because of an infinite loop.

a. Circle the result of this unit test below:

```
@Test public void test1 () {
    A b = new B();
    assertEquals(4, b.mappr());
}
```

Test Passes Assertion Error NPE Stack Overflow

b. Circle the result of this unit test below:

```
@Test public void test2 () {
    A a = new A();
    assertEquals(3, a.mappr());
}
```

Test Passes Assertion Error NPE Stack Overflow

c. Circle the result of this unit test below:

```
@Test public void test3 () {
    A b = new B();
    assertEquals(6, b.knowit());
}
```

Test Passes Assertion Error NPE Stack Overflow

d. Circle the result of this unit test below:

```
@Test public void test4 () {
    A b = new B();
    assertEquals(7, b.pressify());
}
```

Test Passes Assertion Error NPE Stack Overflow

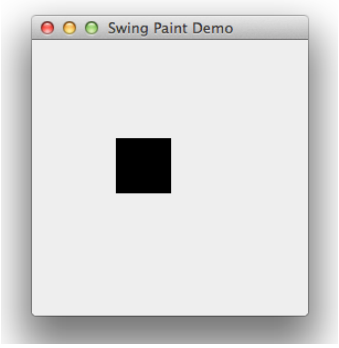
e. Circle the result of this unit test below:

```
@Test public void test5 () {
    B b = new B();
    assertEquals(8, b.scattery());
}
```

Test Passes Assertion Error NPE Stack Overflow

4. Java Swing Programming (10 points)

The code in Appendix D implements a simple Java GUI program in which a 50x50 black box follows the mouse cursor around the window. It looks like this (the mouse cursor is not shown):



The following true/false questions concern this application and Java Swing programming in general.

a. T F The type `MyPanel` is a subtype of `Object`.

b. T F The new object created on line 27 is an instance of class `MyPanel`.

c. T F The instance variables `x` and `y`, declared on lines 23 and 24, can only be modified by the methods of the `MyPanel` class or the methods of any inner classes of `MyPanel`.

d. T F The dynamic class of `f` (declared on line 13) is `JFrame`.

e. T F The `mouseMoved` method on line 28 is called by the Swing event loop in reaction to the user moving the mouse in the main window of the application.

f. T F The `paintComponent` method on line 42 is only invoked once, at the start of the application.

g. T F A `JPanel` cannot be added to another `JPanel` using the `add` method.

h. T F If the user replaced the code on line 31 (i.e. the call to `repaint()`) with `paintComponent(new Graphics())`, then the behavior of the application would not change.

i. T F The anonymous class defined on line 27 implements or inherits all members of the `MouseMotionListener` interface.

j. T F The method `createAndShowGUI` cannot be invoked without an instance of class `GUI`.

5. Subtyping and Collections (10 points)

Consider the Java classes and interfaces excerpted from the Collections Framework, as shown in Appendix E. 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.
```
Set<Integer> s = new TreeSet<Integer>();
s.add(1);
s.add(1);
System.out.println(s.size());
```

0 1 2 Ill typed

b.
```
Collection<Integer> lst = new LinkedList<Integer>();
lst.add(1);
lst.add(1);
System.out.println(lst.size());
```

0 1 2 Ill typed

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

0 1 2 Ill typed

d.
```
List<Integer> c = new LinkedList<Integer>();
Set<Integer> s = new TreeSet<Integer>();
c.add(1);
System.out.println(c.equals(s));
```

false **true** Ill typed

e.
```
List<Collection<Integer>> c = new LinkedList<Collection<Integer>>();
c.add(new LinkedList<Integer>());
LinkedList<Integer> lst = c.get(0);
System.out.println(lst.size());
```

0 1 2 Ill typed

6. Java Data Structures Programming

Appendix G shows (part of) the implementation of a new Java `Collection` class called `StringArrayList`, which implements the `List` interface. The code provided stores the list elements in an array, which is expanded as necessary to contain all elements. Note that this class is *not* generic, it can only store (potentially null) strings.

Your task in this problem will be to implement some missing functionality of the `StringArrayList` class.

a. (5 points) The `StringArrayList` class includes two private instance variables, `data` and `size`.

Which invariants about these instance variables does the current implementation maintain? Circle *all* that apply.

i. `0 < size`

ii. `0 <= size`

iii. `size <= data.length`

iv. `size == data.length`

v. `size > data.length`

vi. `data` is never `null`

vii. For all `i < size`, `data[i]` is not `null`

viii. If `data.length == 0` then `size` is `null`

ix. None of the above

b. (12 points) Now, implement the `contains` method for the `StringArrayList` class. The JavaDoc for this method is given below:

> **`boolean contains(Object o)`**
> Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element `e` such that either `o` and `e` are both `null` or `o` is non-`null` and `o.equals(e)`.

```
public boolean contains(Object o) {




}
```

c. (12 points) Next, implement a remove method for this data structure. The JavaDoc for this method is again given below.

> **`String remove(int index)`**
> Removes the element at the specified position in this list. Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the list.
> **Parameters:** index - the index of the element to be removed
> **Returns:** the element previously at the specified position
> **Throws:** `IndexOutOfBoundsException` - if the index is out of range `index < 0` or `index >= size()`

Complete the `remove` method to match the description given above.

```
public String remove(int index) {









}
```

7. Programming with Collections

Suppose you are given a recitation roster (i.e. a mapping from students to their recitations) and you would like to *invert* that roster. In other words, you would like to create a data structure that records the collection of students that are enrolled in each recitation section.

For example, if we know that Alice is in section 201, Bob is in 202 and Charlotte is in 201, then we should be able to calculate that the students in 201 are Alice and Charlotte and the only student in 202 is Bob.

a. (3 points) Design the interface of the static method `invert`, which you will need to implement below.

You can assume that there are already classes to represent students (like Alice) and recitation sections (like 201). These two classes implement the comparable interface and also have overridden their respective equals methods with appropriate implementations.

```
class Student implements Comparable<Student> { ... }
class Section implements Comparable<Section> { ... }
```

The `invert` method should take a map from students to sections and should return an appropriate data structure for accessing the students in each recitation section. But what should that data structure be?

```
public static _________ invert (Map <Student,Section> roster)
```

Circle the most appropriate result type for `invert`:

i. `Map<Student,Section>`

ii. `Map<Student,Set<Section>>`

iii. `Map<Section,Student>`

iv. `Map<Section,Set<Student>>`

v. `Set<Map<Student,Section>>`

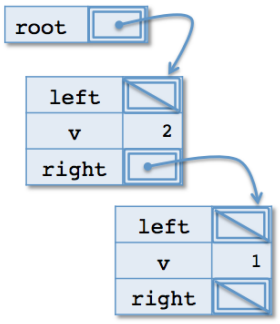
vi. `List<Section>`

b. (12 points) Now, implement the `invert` method. For reference, documentation about maps from the Java Collections framework appears in Appendix F.

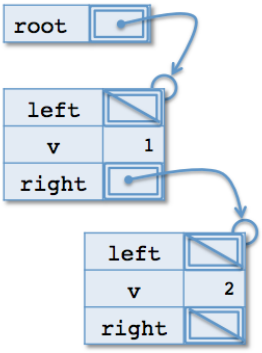
A Mutable binary search trees

```
(* nodes in the tree *)
type 'a node =
  { mutable left : 'a node option;
    v : 'a;
    mutable right: 'a node option }

(* the root of a mutable tree *)
type 'a tree = { mutable root: 'a node option }


(* ---- helper to create a leaf node ---- *)
let leaf (y : 'a) : 'a node =
    { left = None; v = y; right = None }

(* --- Inorder traversal of a mutable tree --- *)
let rec inorder (x : 'a tree) : 'a list =
  let rec loop (no : 'a node option) : 'a list =
    begin match no with
    | None -> []
    | Some n ->
      loop n.left @ [n.v] @ loop n.right
    end in
  loop x.root


(* ---- BST insertion into mutable trees ---- *)
let rec insert (y : 'a) (x: 'a tree) : unit =
  let rec loop (n : 'a node) : unit =
     if y < n.v then
       begin match n.left with
       | Some n -> loop n
       | None -> n.left <- Some (leaf y)
       end
     else if y > n.v then
       begin match n.right with
       | Some n -> loop n
       | None -> n.right <- Some (leaf y)
       end
     else () in
  begin match x.root with
  | Some n -> loop n
  | None -> x.root <- Some (leaf y)
  end
```

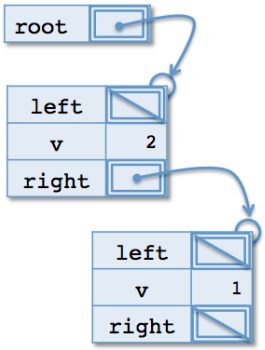
B Mutable trees in the heap



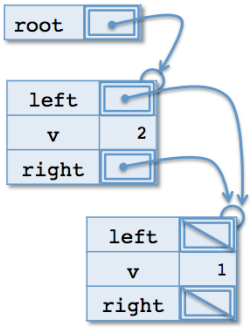
(a)



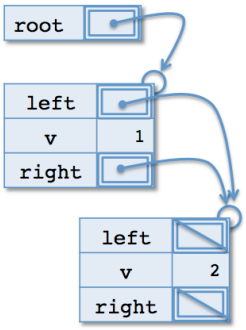
(b)



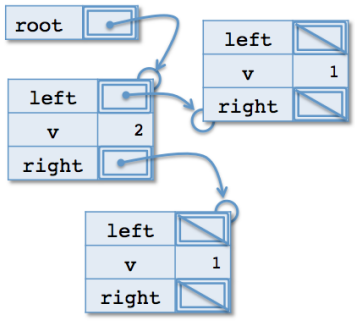
(c)



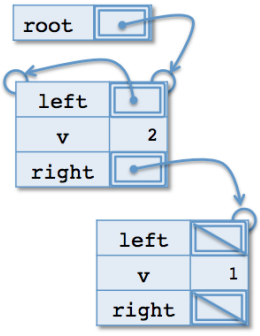
(d)



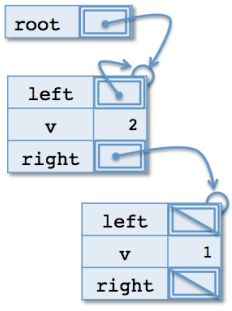
(e)



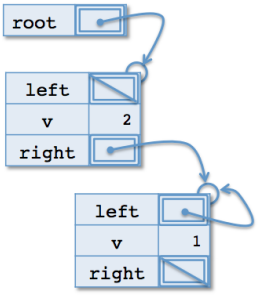
(f)



(g)



(h)



(i)

C Class Definitions for Problem 3

```
class A {
  public int x;
  public B b;

  public A() {
    x = 1;
  }

  public int mappr() {
    return x + B.flattn();
  }
  public int knowit() {
    return mappr() + x + 1;
  }
  public int pressify() {
    return b.pressify();
  }
  public int scattery() {
    b = new B();
    return b.scattery();
  }
}

class B extends A {
  public int y;
  public A a;

  public B(){
    super();
    y = 2;
    a = new A();
  }

  public static int flattn() {
    return 4;
  }

  @Override
  public int mappr() {
    return 2 + y;
  }

}
```

D An Example Java GUI Program

```
1  // library imports omitted to save space (this code compiles)
2
3  public class GUI {
4    public static void main(String[] args) {
5      SwingUtilities.invokeLater(new Runnable() {
6        public void run() {
7          createAndShowGUI();
8        }
9      });
10   }
11
12   static void createAndShowGUI() {
13     JFrame f = new JFrame("Swing Paint Demo");
14     f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15     f.add(new MyPanel());
16     f.pack();
17     f.setVisible(true);
18   }
19 }
20
21 @SuppressWarnings("serial")
22 class MyPanel extends JPanel {
23   private int x;
24   private int y;
25
26   public MyPanel() {
27     addMouseMotionListener(new MouseAdapter() {
28       public void mouseMoved(MouseEvent e) {
29         x = e.getX();
30         y = e.getY();
31         repaint();
32       }
33     });
34   }
35
36   @Override
37   public Dimension getPreferredSize() {
38     return new Dimension(250, 250);
39   }
40
41   @Override
42   public void paintComponent(Graphics g) {
43     super.paintComponent(g);
44     g.setColor(Color.BLACK);
45     g.fillRect(x, y, 50, 50);
46   }
47 }
```

E Excerpt from the Collections Framework (Lists and Sets)

```
interface Collection<E> extends Iterable<E> {
  public boolean add(E o);
  // Ensures that this collection contains the specified element
  // Returns true if this collection changed as a result of the call.
  // (Returns false if this collection does not permit duplicates
  // and already contains the specified element.)

  public boolean contains(Object o);
  // Returns true if this collection contains the specified element.

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

interface List<E> extends Collection<E> {
  public E get(int i);
  // Returns the element at the specified position in this list
  public E remove(int i);
  // Removes the element at the specified position in the list
}


class LinkedList<E> implements List<E> {
   public LinkedList();
   // Constructs an empty list

   // ... methods specified by interface
}

interface Set<E> extends Collection<E> {
   // no new methods
}


class TreeSet<E> implements Set<E> {
   public TreeSet();
   // Constructs an empty set

   // ... methods specified by interface
}
```

F Excerpt from the Collections Framework (Maps)

```
interface Map<K,V> {


  public V get(Object key)
  // Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
  // More formally, if this map contains a mapping from a key k to a value v
  // such that (key==null ? k==null : key.equals(k)), then this method returns
  // v; otherwise it returns null. (There can be at most one such mapping.)

  public V put(K key, V value)
  // Associates the specified value with the specified key in this map

  public int size()
  // Returns the number of key-value mappings in this map.

  public boolean isEmpty()
  // Returns true if this map contains no key-value mappings.

  public Set<K> keySet()
  // Returns a Set view of the keys contained in this map. The set is backed
  // by the map, so changes to the map are reflected in the set, and
  // vice-versa.

  public boolean containsKey(Object key)
  // Returns true if this map contains a mapping for the specified key.

  public boolean containsValue(Object value)
  // Returns true if this map maps one or more keys to the specified value.


}


class TreeMap<K,V> implements Map<K,V> {

  public TreeMap()
  // constructor
  // Constructs a new, empty tree map, using the natural ordering of its keys.

  // ... methods specified by interface
}
```

G StringArrayList implementation

```
public class StringArrayList implements List<String> {
  // instance variables
  private String[] data = new String[0];
  private int size = 0;

  // default constructor
  public StringArrayList() {}

  private void ensureCapacity() {
    if (size == data.length) {
      String[] newdata = new String[Math.max(data.length * 2, size+1)];
      for (int i=0; i<data.length; i++) {
        newdata[i] = data[i];
      }
      data = newdata;
    }
  }

  // Appends the specified element ( potentially null ) to the end of this list
  public boolean add(String x) {
    ensureCapacity();
    data[size] = x;
    size++;
    return true;
  }

  // Returns the element at the specified position in this list
  public String get(int index) {
    if (index < 0 || index >= size) {
      throw new IndexOutOfBoundsException();
    }
    return data[index];
  }

  public int size() {
    return size;
  }

  public boolean isEmpty() {
    return (size == 0);
  }

  // additional methods not shown
}
```