CSE 143: Computer Programming II — Spring 2015

Final Exam Solutions

Name:	**Sample Solutions**		
ID #:	**1234567**		
TA:	**The Best**	**Section:**	**A9**

INSTRUCTIONS:

- You have **110 minutes** to complete the exam.
- You *will* receive a deduction if you keep working after the instructor calls for papers.
- This exam is closed-book and closed-notes. You may not use any computing devices including calculators.
- Code will be graded on proper behavior/output and not on style, unless otherwise indicated.
- Do not abbreviate code, such as "ditto" marks or dot-dot-dot ("...") marks. The only abbreviations that are allowed for this exam are: `S.o.p` for `System.out.print` and `S.o.pln` for `System.out.println`.
- You do not need to write import statements in your code.
- You may not use scratch paper on this exam. If you need extra space, use the back of a page.
- If you enter the room, you must turn in an exam before leaving the room.
- You must show your Student ID to a TA or instructor for your exam to be accepted.
- If you get stuck on a problem, move on and come back to it later.

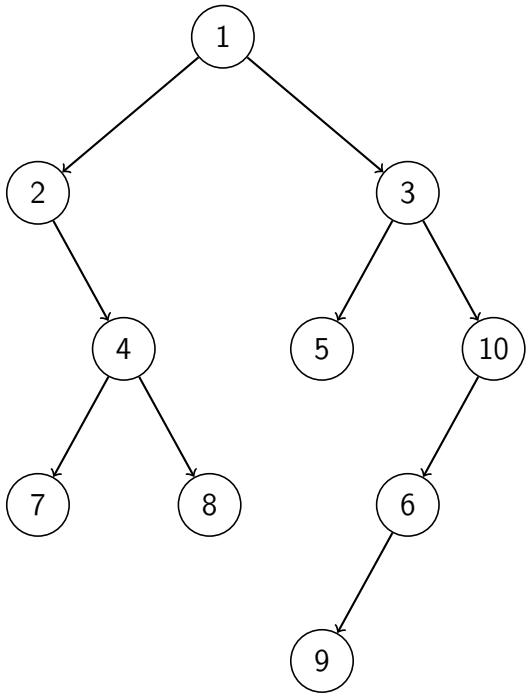
Problem	Points	Score	Problem	Points	Score
1	6		5	13	
2	4		6	15	
3	8		7	20	
4	14		8	20	
			Σ	100	

Mechanical Missions.

This section tests whether you are able to trace through code of various types in the same way a computer would.

1. I'm Rooting For You! [6 points]

Write the elements of the tree in the order they would be seen by pre-order, in-order, and post-order traversals.

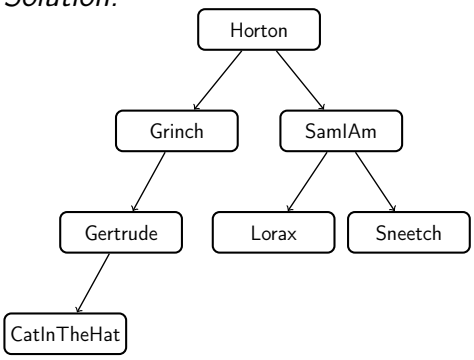


Pre-order:	**1, 2, 4, 7, 8, 3, 5, 10, 6, 9**
In-order:	**2, 7, 4, 8, 1, 5, 3, 9, 6, 10**
Post-order:	**7, 8, 4, 2, 5, 9, 6, 10, 3, 1**

2. I Do Not Like Them [4 points]
Draw a picture of the binary search tree that would result from inserting the following words into an empty binary search tree in the following order:

(1) Horton
(2) SamIAm
(3) Grinch
(4) Lorax
(5) Gertrude
(6) CatInTheHat
(7) Sneetch

Solution:



3. `Collections` **Mystery** [8 points]

```
1  public static void mystery(List<Integer> list) {
2     Map<Integer, Integer> result = new TreeMap<Integer, Integer>();
3     Iterator<Integer> it = list.iterator();
4
5        while (it.hasNext()) {
6           int j = it.next();
7           if (it.hasNext()) {
8              int k = it.next();
9              result.put(k, j);
10             result.put(j, k);
11          }
12       }
13    System.out.println(result);
14 }
```

For each of the following, fill in the *output* printed when `mystery` is called on the given `Collection`.

	Input `Collection`	**Output**
(a)	`[1, 1]`	`{1=1}`
(b)	`[200, 200, 300, 400]`	`{200=200, 300=400, 400=300}`
(c)	`[1, 2, 1, 2]`	`{1=2, 2=1}`
(d)	`[9, 8, 7, 6, 5, 4]`	`{4=5, 5=4, 6=7, 7=6, 8=9, 9=8}`

4. Polymorphism Mystery [14 points]

Consider the following classes:

```
1  public class Ninja extends Robot {
2     public void method1() {
3        System.out.println("Ninja 1");
4     }
5
6     public void method3() {
7        System.out.println("Ninja 3");
8     }
9  }
10
11 public class Pirate {
12    public void method1() {
13       System.out.println("Pirate 1");
14    }
15
16    public void method2() {
17       System.out.println("Pirate 2");
18       this.method1();
19    }
20 }
21
22 public class Robot extends Pirate {
23    public void method1() {
24       System.out.println("Robot 1");
25       super.method1();
26    }
27 }
28
29 public class Alien extends Pirate {
30    public void method3() {
31       System.out.println("Alien 3");
32    }
33 }
```

Consider the following variables:

```
1 Pirate var1 = new Ninja();
2 Pirate var2 = new Robot();
3 Pirate var3 = new Pirate();
4 Object var4 = new Robot();
5 Robot var5 = new Ninja();
6 Object var6 = new Alien();
```

What does each of these lines output?

	Statement	Output
(a)	`var1.method1();`	Ninja 1
(b)	`var2.method1();`	Robot 1 / Pirate 1
(c)	`var3.method1();`	Pirate 1
(d)	`var4.method1();`	error
(e)	`var5.method1();`	Ninja 1
(f)	`var2.method2();`	Pirate 2 / Robot 1 / Pirate 1
(g)	`var3.method2();`	Pirate 2 / Pirate 1
(h)	`var6.method2();`	error
(i)	`((Robot)var1).method3();`	error
(j)	`((Alien)var6).method3();`	Alien 3
(k)	`((Ninja)var4).method1();`	error
(l)	`((Pirate)var6).method2();`	Pirate 2 / Pirate 1
(m)	`((Pirate)var4).method1();`	Robot 1 / Pirate 1
(n)	`((Ninja)var3).method3();`	error

Programming Pursuits.

This section tests whether you synthesized various topics well enough to write novel programs using those topics.

5. Daisy, Daisy... [13 points]

Define a class `Bicycle` that represents a bicycle.

Your class should have the following *public* methods:

`Bicycle(`**fixedWheel**`, `**gears**`)`	Constructs a `Bicycle` based on if it is a fixed-wheel bicycle or not and how many gears it has. All bicycles should store the current speed which should be initialized to 1.0mph. This constructor should throw an `IllegalArgumentException` if `fixedWheel` is true and `gears` is not equal to one. It should also throw an `IllegalArgumentException` if `gears` is less than 1.
`isFixedWheel()`	This method returns true if the bicycle is a fixed-wheel bicycle and false otherwise.
`getSpeed()`	This method returns the current speed of the bicycle.
`changeSpeed(`**howMuch**`)`	This method takes in a *multiplier* for how much more or less the bicyclist is pedaling. If `s` is the old speed, then the new speed should be `howMuch * s`.
`toString()`	If the bicycle is a *fixed-wheel*, prepend "`Fixie`". Otherwise, prepend "`<gears>-gear bicycle`". Then, the remainder of the result should be "`(speed: <speed>mph)`".

Below are some examples of `Bicycles`:

```
1 Bicycle fixie = new Bicycle(true, 1);
2 Bicycle bike = new Bicycle(false, 8);
```

Example Output

Method Call	Return Value
`fixie.isFixedWheel()`	`true`
`fixie.getSpeed()`	`1.0`
`fixie.changeSpeed(1.55444)`	
`fixie.getSpeed()`	`1.55444`
`fixie.toString()`	`Fixie (speed: 1.55444mph)`
`bike.changeSpeed()`	`0.155`
`bike.toString()`	`8-gear bicycle (speed: 0.155mph)`

We would like to compare bikes by "coolness factor". So, your `Bicycle` class should implement the `Comparable` interface. **Cooler bicycles should be considered *greater*.**

Implement the `Comparable<E>` interface by considering aspects in the following order:

- Fixed-wheel bicycles are the coolest.
- Otherwise, faster bicycles are cooler.
- Otherwise, bicycles with more gears are cooler.

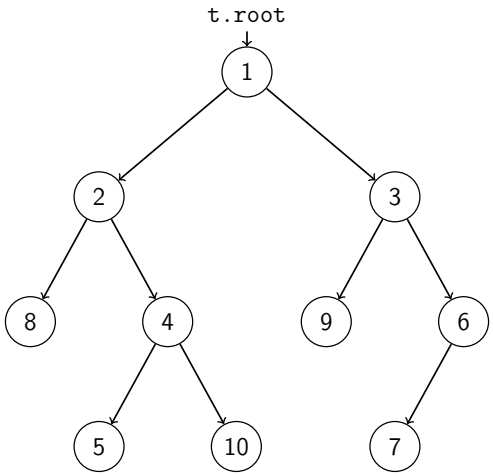
Solution:

```
1  public class Bicycle implements Comparable<Bicycle> {
2     private boolean fixedWheel;
3     private int gears;
4     private double speed;
5
6     public Bicycle(boolean fixedWheel, int gears) {
7        if (fixedWheel && gears != 1 || gears < 1) {
8           throw new IllegalArgumentException();
9        }
10       this.fixedWheel = fixedWheel;
11       this.gears = gears;
12       this.speed = 1.0;
13    }
14
15    public boolean isFixedWheel() {
16       return this.fixedWheel;
17    }
18
19    public double getSpeed() {
20       return this.speed;
21    }
22
23    public void changeSpeed(double howMuch) {
24       this.speed *= howMuch;
25    }
26
27    public String toString() {
28       if (this.fixedWheel) {
29          return "Fixie (speed: " + this.speed + " mph)";
30       }
31       else {
32          return this.gears + "-gear bicycle (speed: " + this.speed + " mph)";
33       }
34    }
35
36    public int compareTo(Bicycle other) {
37       if (this.fixedWheel && !other.fixedWheel) {
38          return 1;
39       }
40       else if (!this.fixedWheel && other.fixedWheel) {
41          return -1;
42       }
43
44       int result = ((Double)this.speed).compareTo(other.speed);
45       if (result == 0) {
46          result = ((Integer)this.gears).compareTo(other.gears);
47       }
48
49       return result;
50    }
51 }
```

6. Oddly Even [15 points]

Write an `IntTree` method `countTolerantParents` that returns the number of nodes that have exactly one even child and exactly one odd child.

For example, if a variable `t` stores a reference to the following tree:



A call to `t.countTolerantParents()` should return 3, because `1, 4, 3` all have one even child and one odd child.

```
public class IntTree {
   private class IntTreeNode {
      public final int data;    // data stored in this node
      public IntTreeNode left;  // reference to left subtree
      public IntTreeNode right; // reference to right subtree

      public IntTreeNode(int data) { ... }
      public IntTreeNode(int data, IntTreeNode left) { ... }
      public IntTreeNode(int data, IntTreeNode left, IntTreeNode right) { ... }
   }

   private IntTreeNode root;

   // Write Your Solution Here
```

Solution:

```
1  public int countTolerantParents() {
2     return countTolerantParents(this.root);
3  }
4
5  private int countTolerantParents(IntTreeNode current) {
6     if (current == null) {
7        return 0;
8     }
9     else if (current.left != null && current.right != null &&
10          (current.left.data % 2 != current.right.data % 2)) {
11       return 1 + countTolerantParents(current.left)
12             + countTolerantParents(current.right);
13    }
14    else {
15       return countTolerantParents(current.left) +
16            countTolerantParents(current.right);
17    }
18 }
```

```
}
```

7. Can You Do The Limbo? [20 points]

Write an `IntTree` method `compressTree` that compresses a tree by removing all parents that **have exactly one child** from the tree.

```
public class IntTree {
   private class IntTreeNode {
      public final int data;    // data stored in this node
      public IntTreeNode left;  // reference to left subtree
      public IntTreeNode right; // reference to right subtree

      public IntTreeNode(int data) { ... }
      public IntTreeNode(int data, IntTreeNode left) { ... }
      public IntTreeNode(int data, IntTreeNode left, IntTreeNode right) { ... }
   }

   private IntTreeNode root;

   // Write Your Solution Here

}
```

Solution:

```
1  public void compressTree() {
2     this.root = compressTree(this.root);
3  }
4
5  private IntTreeNode compressTree(IntTreeNode current) {
6     if (current != null) {
7        if (current.left != null && current.right == null) {
8           current = compressTree(current.left);
9        }
10       else if (current.left == null && current.right != null) {
11          current = compressTree(current.right);
12       }
13       else {
14          current.left = compressTree(current.left);
15          current.right = compressTree(current.right);
16       }
17    }
18    return current;
19 }
```

8. Wibble Wobble [20 points]

Write a `LinkedIntList` method `wobble` that rearranges a list by moving all the values in *even-numbered* positions to the end of the list (and otherwise preserving list order). If the list is empty or only has one element, the list should remain unchanged.

Example Output

Before `list.wobble()`	After `list.wobble()`
`[1, 2, 3, 4]`	`[2, 4, 1, 3]`
`[2, 4, 6, 8]`	`[4, 8, 2, 6]`
`[3, 5, 7, 9, 11]`	`[5, 9, 3, 7, 11]`

```
public class LinkedIntList {
   private class ListNode {
      public final int data; // data stored in this node
      public ListNode next;  // reference to the next node

      public ListNode(int data) { ... }
      public ListNode(int data, ListNode next) { ... }
   }

   private ListNode front;

   // Write Your Solution Here
```

Solution:

```
1  public void wobble() {
2     if (front != null && front.next != null) {
3        ListNode otherFront = front;
4        ListNode otherBack = front;
5        front = front.next;
6        ListNode current = front;
7        while (current.next != null && current.next.next != null) {
8           otherBack.next = current.next;
9           otherBack = current.next;
10          current.next = current.next.next;
11          current = current.next;
12       }
13       if (current.next != null) {
14          otherBack.next = current.next;
15          otherBack = current.next;
16       }
17       current.next = otherFront;
18       otherBack.next = null;
19    }
20 }
```

```
}
```