CSE 143: Computer Programming II — Autumn 2016

Final Exam

Name:	**Sample Solutions**		
E-mail:	**bovik**	`@washington.edu`	
TA:	**The Best**	**Section:**	**A9**

INSTRUCTIONS:

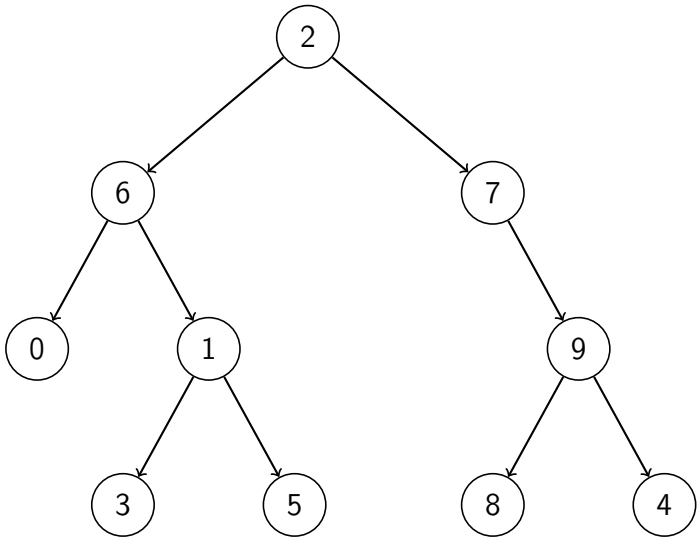
- You have **110 minutes** to complete the exam.
- You *will* receive a deduction if you keep working after the instructor calls for papers.
- This exam is closed-book and closed-notes. You may not use any computing devices including calculators.
- Code will be graded on proper behavior/output and not 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 extra scratch paper on this exam. Use the provided spaces for extra work.
- If you write work you want graded on a strange page, clearly label it.
- 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	15	
2	4		6	15	
3	6		7	20	
4	14		8	20	
			EC	1	
			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. Leaving The Order [6 points]
Write the elements of the tree in the order they would be seen by pre-order, in-order, and post-order traversals.

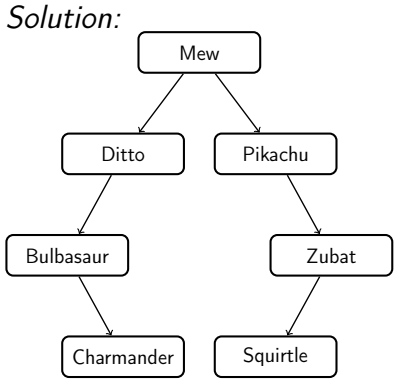


Pre-order:	**2, 6, 0, 1, 3, 5, 7, 9, 8, 4**
In-order:	**0, 6, 3, 1, 5, 2, 7, 8, 9, 4**
Post-order:	**0, 3, 5, 1, 6, 8, 4, 9, 7, 2**

2. Gotta Insert 'Em All! [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) Mew
(2) Pikachu
(3) Zubat
(4) Ditto
(5) Squirtle
(6) Bulbasaur
(7) Charmander

Solution:



3. `Collections` **Mystery** [6 points]

```
1  public static void mystery(List<Integer> list) {
2     Map<Integer, Integer> result = new TreeMap<Integer, Integer>();
3
4     while (!list.isEmpty()) {
5        Iterator<Integer> it = list.iterator();
6
7        while (it.hasNext()) {
8           int i = it.next();
9           if (i % 2 == 0) {
10             if (!result.containsKey(i)) {
11                result.put(i, 0);
12             }
13             result.put(i, result.get(i) + 1);
14          }
15          it.remove();
16       }
17    }
18
19    System.out.println(result);
20 }
```

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

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

Question #4: Scratch Work

4. Polymorphism Mystery [14 points]

Consider the following classes:

```
1  public class Punk extends Jazz {
2    public void method2() {
3      System.out.println("Punk 2");
4    }
5
6    public void method3() {
7      System.out.println("Punk 3");
8    }
9  }
10
11 public class Rap extends Blues {
12   public void method2() {
13     System.out.println("Rap 2");
14   }
15
16   public void method3() {
17     System.out.println("Rap 3");
18   }
19 }
20
21 public class Blues {
22   public void method1() {
23     System.out.println("Blues 1");
24     this.method3();
25   }
26
27   public void method3() {
28     System.out.println("Blues 3");
29   }
30 }
31
32 public class Jazz extends Blues {
33   public void method3() {
34     System.out.println("Jazz 3");
35     super.method3();
36   }
37 }
```

Consider the following variables:

```
1 Blues var1 = new Jazz();
2 Punk var2 = new Punk();
3 Blues var3 = new Rap();
4 Blues var4 = new Punk();
5 Object var5 = new Jazz();
```

What does each of these lines output?

Write the output of each method call in the corresponding box. If the code produces an error, write "error". Use some consistent notation to indicate new lines.

	Statement	Output
(a)	`var1.method1();`	Blues 1 / Jazz 3 / Blues 3
(b)	`var4.method1();`	Blues 1 / Punk 3
(c)	`var5.method1();`	error
(d)	`var2.method2();`	Punk 2
(e)	`var3.method2();`	error
(f)	`var1.method3();`	Jazz 3 / Blues 3
(g)	`var2.method3();`	Punk 3
(h)	`((Punk) var5).method1();`	error
(i)	`((Jazz) var3).method2();`	error
(j)	`((Punk) var4).method2();`	Punk 2
(k)	`((Blues) var2).method2();`	error
(l)	`((Rap) var3).method2();`	Rap 2
(m)	`((Jazz) var4).method2();`	error
(n)	`((Jazz) var4).method3();`	Punk 3

Programming Pursuits.

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

5. Buying a Laptop [15 points]

Define a class `Laptop` that represents a laptop.

Your class should have the following *public* methods:

`Laptop(`**price, amountOfRAM, processorSpeed, brand, hasSSD**`)`	Constructs a `Laptop` based on its specs. The price should be given in dollars; the amount of RAM is given in GBs; the processor speed is given in GHz (e.g., 1.5GHz); the brand is given as a string. If any of `price`, `amountOfRAM`, or `processorSpeed` is negative or if `brand` is empty or null, the constructor should throw an `IllegalArgumentException`.
`addRAM(`**howMuch**`)`	This method increases the amount of RAM in this laptop by `howMuch`.
`changePrice(`**howMuch**`)`	This method alters the price in this laptop by `howMuch` *cents*.
`toString()`	This method returns a String representation of the laptop that looks like the following: "`<brand> Laptop ($<price>): <processorSpeed>GHz with <amountOfRAM>G of RAM`". If the laptop has an SSD, "`and an SSD`" should be appended to the end of the string representation. The price should be included with two decimal points, and the processorSpeed should be included with one decimal point. You may assume you have access to a method **round(num, precision)** which returns a String version of **num** with **precision** many decimal places.

Below are some examples of `Laptops`:

```
1 Laptop m1 = new Laptop(2000.99, 4, 3.5, "Dell", false);
2 Laptop m2 = new Laptop(9000.99, 8, 5.5, "Apple", true);
```

Example Output

Method Call	Return Value
`m1.addRAM(5)`	
`m1.toString()`	`Dell Laptop ($2000.99): 3.5GHz with 9G of RAM`
`m2.changePrice(10)`	
`m2.toString()`	`Apple Laptop ($9001.09): 5.5GHz with 8G of RAM and an SSD`

We would like to compare laptops to decide the best one to buy. The better fit the laptop is to buy, the *lesser* the laptop should be considered.

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

- Lower price is better.
- Otherwise, more RAM is preferable.
- Otherwise, higher processor speed is preferable.

Question #5: Solution Space

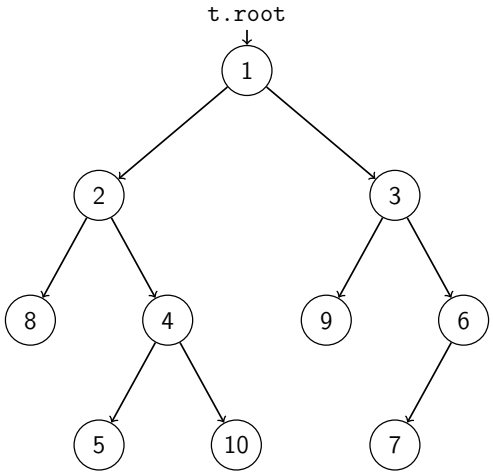
Solution:

```
1  public class Laptop implements Comparable<Laptop> {
2     private double price;
3     private int amountOfRAM;
4     private double processorSpeed;
5     private String brand;
6     private boolean hasSSD;
7
8     public Laptop(double price, int amountOfRAM, double speed, String brand, boolean hasSSD) {
9        if (price < 0 || amountOfRAM < 0 || processorSpeed < 0 || brand == null || brand.isEmpty
             ()) {
10          throw new IllegalArgumentException();
11       }
12       this.price = price;
13       this.amountOfRAM = amountOfRAM;
14       this.processorSpeed = speed;
15       this.brand = brand;
16       this.hasSSD = hasSSD;
17    }
18
19    public void addRAM(int howMuch) {
20       this.amountOfRAM += howMuch;
21    }
22
23    public void changePrice(int howMuch) {
24       this.price += howMuch/100.0;
25    }
26
27    public String toString() {
28       String result = this.brand + " Laptop ($" + round(this.price, 2) + "): " +
29                    round(this.processorSpeed, 1) + "GHz with " + this.amountOfRAM + "G of RAM";
30       if (this.hasSSD) {
31          result += " and an SSD";
32       }
33       return result;
34    }
35
36    public int compareTo(Laptop other) {
37       int result = Double.compare(this.price, other.price);
38       if (result != 0) {
39          return result;
40       }
41
42       result = Integer.compare(this.amountOfRAM, other.amountOfRAM);
43       if (result != 0) {
44          return -result;
45       }
46
47       return -Double.compare(this.processorSpeed, other.processorSpeed);
48    }
49 }
```

6. The Path to Victory! [15 points]

Write an `IntTree` method `countPathsOfLength` that takes one argument n, and returns the number of *paths* that have length exactly n. Recall that a path is a list of nodes starting at the root and ending at a leaf, and the length of a path is the number of nodes it has. You may assume that n is not zero.

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



A call to `t.countPathsOfLength(6)` should return `0`; however, a call to `t.countPathsOfLength(3)` should return 2, because 1,2,8 and 1,3,9 are the only paths of length 3 in the tree.

Implementation Restrictions

- You may not call any other methods on the `IntTree` object (e.g., `add`, `remove`)
- You *may not* construct new `IntTreeNode` objects.
- You may not use any other data structures such as arrays, lists, queues, etc.
- Your solution should run in $\mathcal{O}(n)$ time, where n is the number of elements in the tree.

Use This Box For Scratch Work **DO NOT WRITE YOUR SOLUTION HERE**

Question #6: Solution Space

```
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:

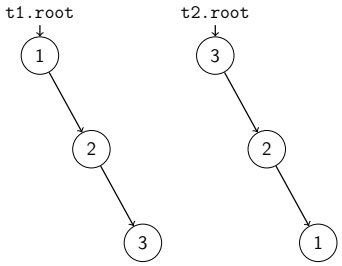
```
1  public int countPathsOfLength(int length) {
2     return countPathsOfLength(this.root, length);
3  }
4
5  private int countPathsOfLength(IntTreeNode current, int length) {
6     if (current == null && length != 0) {
7        return 0;
8     }
9     else if (current == null && length == 0) {
10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

7. Branching Out... [20 points]

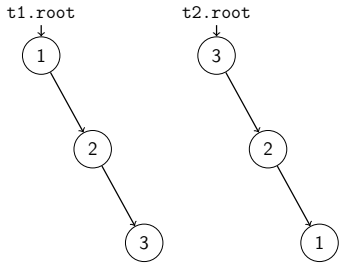
Write an `IntTree` method `indicateMatching` which compares the nodes of this `IntTree` to the nodes of a second `IntTree`. For each node, change the tree in the following way:

- If a node exists in this tree but not the other tree, replace it with -1.
- If a node exists in the other tree but not this tree, replace it with -2.
- Otherwise, leave the node unchanged.

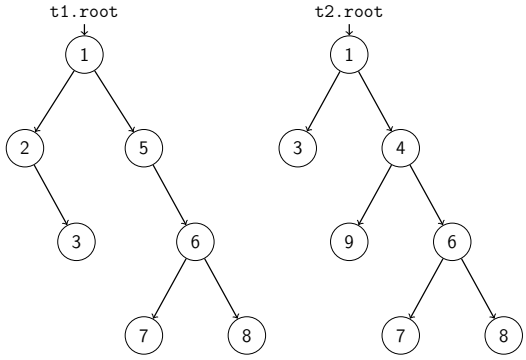
Before `t1.indicateMatching(t2)`



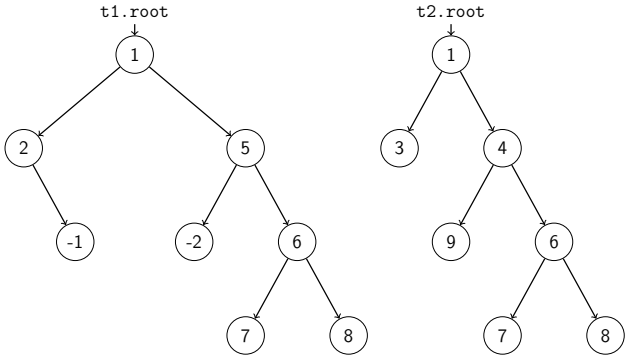
After `t1.indicateMatching(t2)`



Before `t1.indicateMatching(t2)`



After `t1.indicateMatching(t2)`



Implementation Restrictions

- You must make sure to create **as few new** `IntTreeNodes` **as possible**.
- You may not call any other methods on the `IntTree` object (e.g., `add`, `remove`)
- You may not use any other data structures such as arrays, lists, queues, etc.
- Your solution should run in $\mathcal{O}(n)$ time, where n is the number of elements in the tree.

Use This Box For Scratch Work **DO NOT WRITE YOUR SOLUTION HERE**

Question #7: Solution Space

```
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 indicateMatching(IntTree other) {
2     this.root = indicateMatching(this.root, other.root);
3  }
4
5  private IntTreeNode indicateMatching(IntTreeNode me, IntTreeNode them) {
6     if (me == null && them == null) {
7        return null;
8     }
9     else if (me == null) {
10       IntTreeNode newMe = new IntTreeNode(-2);
11       newMe.left = indicateMatching(null, them.left);
12       newMe.right = indicateMatching(null, them.right);
13       return newMe;
14    }
15    else if (them == null) {
16       IntTreeNode newMe = new IntTreeNode(-1);
17       newMe.left = indicateMatching(me.left, null);
18       newMe.right = indicateMatching(them.left, null);
19       return newMe;
20    }
21    else {
22       me.left = indicateMatching(me.left, them.left);
23       me.right = indicateMatching(me.right, them.right);
24       return me;
25    }
26 }
```

8. Don't Be Negative! [20 points]

Write a `LinkedIntList` method `splitBySign` that splits this list into two parts. The non-negative values should remain in this list unchanged. The negative values should be removed and returned as a new `LinkedIntList` in the same order they originally appeared in.

Example Output

Before `list.splitBySign()`	`list` after `list.splitBySign()`	Return Value
`[0, -2, 3, -4]`	`[0, 3]`	`[-2, -4]`
`[-1, -1, -2, -3, 3, 0, 1, 2, 3]`	`[3, 0, 1, 2, 3]`	`[-1, -1, -2, -3]`

Implementation Restrictions

- You may not call any other methods on the `LinkedIntList` object (e.g., `add`, `remove`)
- You *may not* construct new `ListNode` objects.
- You may not use any other data structures such as Java arrays, lists, queues, etc.
- Your solution should run in $\mathcal{O}(n)$ time, where n is the number of elements in the list.

Use This Box For Scratch Work

DO NOT WRITE YOUR SOLUTION HERE

Question #8: Solution Space

```
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) { ... }
   }

   public LinkedIntList() { ... }

   private ListNode front;

   // Write Your Solution Here
```

Solution:

```
1  public LinkedIntList splitBySign() {
2     ListNode myFront = null;
3     ListNode negativesFront = null;
4
5     ListNode myCurrent = null;
6     ListNode negativesCurrent = null;
7
8     while (this.front != null) {
9        if (this.front.data < 0) {
10          if (negativesFront == null) {
11             negativesFront = this.front;
12             negativesCurrent = negativesFront;
13          }
14          else {
15             negativesCurrent.next = this.front;
16             negativesCurrent = negativesCurrent.next;
17          }
18       }
19       else {
20          if (myFront == null) {
21             myFront = this.front;
22             myCurrent = myFront;
23          }
24          else {
25             myCurrent.next = this.front;
26             myCurrent = myCurrent.next;
27          }
28       }
29       ListNode temp = this.front.next;
30       this.front.next = null;
31       this.front = temp;
32    }
33
34    this.front = myFront;
35    LinkedIntList negatives = new LinkedIntList();
36    negatives.front = negativesFront;
37
38    return negatives;
39 }
```