

CSE 143, Winter 2009
Final Exam
Thursday, March 19, 2009

Personal Information:

Name: _____

Section: _____ TA: _____

Student ID #: _____

- You have 110 minutes to complete this exam.
You may receive a deduction if you keep working after the instructor calls for papers.
- This exam is open-book/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.
 - 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.

Good luck!

Score summary: (for grader only)

Problem	Description	Earned	Max
1	Inheritance/Polymorphism		15
2	Inheritance/Comparable Programming		15
3	Linked List Programming		15
4	Searching and Sorting		15
5	Binary Search Trees		15
6	Binary Tree Programming		10
7	Binary Tree Programming		15
X	Extra Credit		+1
TOTAL	Total Points		100

1. Inheritance and Polymorphism

Consider the following classes
(System.out.println has been abbreviated as S.o.pln):

```
public class Leo extends Don {
    public void method1() {
        S.o.pln("Leo 1");
    }

    public void method3() {
        S.o.pln("Leo 3");
        method1();
    }
}

public class Mike extends Leo {
    public void method2() {
        S.o.pln("Mike 2");
        super.method2();
    }
}

public class Raph {
    public void method1() {
        S.o.pln("Raph 1");
    }
}

public class Don extends Raph {
    public void method2() {
        method1();
        S.o.pln("Don 2");
    }
}
```

The following variables are defined:

```
Raph var1 = new Don();
Leo var2 = new Mike();
Object var3 = new Raph();
Don var4 = new Leo();
```

In the table below, indicate in the right-hand column the output produced by the statement in the left-hand column. If the statement produces more than one line of output, indicate the line breaks with slashes as in "a / b / c" to indicate three lines of output with "a" followed by "b" followed by "c". If the statement causes an error, fill in the right-hand column with the phrase "error" to indicate this.

<u>Statement</u>	<u>Output</u>
var1.method1();	_____
var1.method2();	_____
var1.method3();	_____
var2.method1();	_____
var2.method2();	_____
var2.method3();	_____
var3.method1();	_____
var4.method1();	_____
var4.method2();	_____
var4.method3();	_____
((Don) var1).method2();	_____
((Mike) var2).method2();	_____
((Raph) var3).method1();	_____
((Don) var3).method2();	_____
((Leo) var4).method3();	_____

2. Inheritance and Comparable Programming

You have been asked to extend a pre-existing class `Person` that represents a person used as part of a Gale-Shapley stable marriage program (as seen in your HW3). The `Person` class includes the following constructors and methods:

Constructor/Method	Description
<code>public Person(String name)</code>	constructs a person with the given name
<code>public String getName()</code>	returns the person's name
<code>public void engageTo(Person partner)</code>	sets the person to be engaged to the given partner
<code>public Person getFiancee()</code>	returns the person's fiancée, or <code>null</code> if none
<code>public boolean isSingle()</code>	returns <code>true</code> if the person has no fiancée
<code>public Queue<String> getPreferences()</code>	returns a queue of preferred partners
<code>public Map<String, Integer> getRankings()</code>	returns a map of rankings
<code>public String toString()</code>	returns a string representation of the person

You are to **define a new class called `Playa` that extends this class through inheritance**. A `Playa` should behave like a `Person` except that it makes mischief by allowing itself to be engaged to multiple persons at the same time, keeping track of a collection of all such fiancées. You should provide the same methods as the superclass, as well as the following new behavior.

Constructor/Method	Description
<code>public Playa(String name)</code>	constructs a <code>playa</code> with the given name
<code>public int countFiancees()</code>	returns the number of fiancées to which this <code>playa</code> is currently engaged

The behaviors related to preferences and rankings of potential partners are unaffected by this subclass.

Some of the existing behaviors from `Person` should behave differently on `Playa` objects:

- When the `engageTo` method is called, the `Playa` should still retain the existing `engageTo` behavior (because it maintains important internal state), but it should also keep track of a collection of all engagement partners seen so far. Each partner passed to `engageTo` should become part of this collection. It should not be possible for the same person to appear twice in this collection. If `null` is passed to `engageTo`, your `Playa` should instead clear its collection of partners to become single again. (A `Playa` can become engaged to any person(s), not just other `Playas`.)
- The `getFiancee` method should return the partner to which the `Playa` most recently became engaged (not counting `null`). This will occur automatically if the original `engageTo` behavior from `Person` is retained.
- The `isSingle` method should return `true` only if the `Playa` has no partners in its engagement collection.

You must also **make `Playa` objects comparable to each other using the `Comparable` interface**. `Playas` are compared by their number of fiancées, breaking ties by name. In other words, a `Playa` object with fewer fiancées in its partner collection is considered to be "less than" one with more fiancées in its collection. If two `Playa` objects have the same number of fiancées, the one whose name comes first in alphabetical order is considered "less." If the two objects have the same number of fiancées and the same name, they are considered to be "equal."

The majority of your grade comes from implementing the correct behavior. Part of your grade also comes from appropriately utilizing the behavior you have inherited from the superclass and not re-implementing behavior that already works properly in the superclass. (Write your answer on the next page.)

2. Inheritance and Comparable Programming (writing space)

3. Linked List Programming

Write a method `expand` that could be added to the `LinkedList` class from lecture and section. The method accepts an integer f as a parameter and replaces every element value i with f copies of the value (i/f) . Suppose a `LinkedList` variable `list` stores the following values:

```
[21, 8, 15, 0, -3, 32]
```

The call `list.expand(3);` would change the list to store the following elements:

```
[7, 7, 7, 2, 2, 2, 5, 5, 5, 0, 0, 0, -1, -1, -1, 10, 10, 10]
```

If an element of the original list is not evenly divisible by f , as with 8 and 32 above, the resulting list should truncate any fractional component (as is done naturally by integer division). If the parameter value passed is 1, the list is unchanged. If it is 0 or negative, the list should become empty.

For full credit, your solution must run in $O(N)$ time where N is the length of the list. You may not call any methods of the linked list class to solve this problem, and you may not use any auxiliary data structure to solve this problem (such as an array, `ArrayList`, `Queue`, `String`, etc).

Assume that you are using the `LinkedList` and `ListNode` class as defined in lecture and section:

```
public class LinkedList {
    private ListNode front;

    methods
}

public class ListNode {
    public int data; // data stored in this node
    public ListNode next; // a link to the next node in the list

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

4. Searching and Sorting

(a) Suppose we are performing a **binary search** on a sorted array called `numbers` initialized as follows:

```
// index      0  1  2  3  4  5  6  7  8  9 10 11 12 13
int[] numbers = {-2, 0, 1, 7, 9, 16, 19, 28, 31, 40, 52, 68, 85, 99};
```

```
// search for the value 5
int index = binarySearch(numbers, 5);
```

Write the indexes of the elements that would be examined by the binary search (the `mid` values in our algorithm's code) and write the value that would be returned from the search. Assume that we are using the binary search algorithm shown in lecture and section.

- Indexes examined: _____
- Value Returned: _____

(b) Write the elements of the array below after each of the first 3 passes of the outermost loop of a **selection sort**.

```
int[] numbers = {63, 9, 45, 72, 27, 18, 54, 36};
selectionSort(numbers);
```

(c) Trace the complete execution of the **merge sort** algorithm when called on the array below, similarly to the example trace of merge sort shown in the lecture slides. Show the sub-arrays that are created by the algorithm and show the merging of sub-arrays into larger sorted arrays.

```
int[] numbers = {63, 9, 45, 72, 27, 18, 54, 36};
mergeSort(numbers);
```

5. Binary Search Trees

(a) Write the binary search tree that would result if these elements were added to an empty tree in this order:

- Meg, Stewie, Peter, Joe, Lois, Brian, Quagmire, Cleveland

(b) Write the elements of your above tree in the order they would be visited by each kind of traversal:

- Pre-order: _____
- In-order: _____
- Post-order: _____

6. Binary Tree Programming

Write a method `nodesAtLevels` that could be added to the `IntTree` class from lecture and section. The method accepts minimum and maximum integers as parameters and returns a count of how many elements exist in the tree at those levels, inclusive. Recall that the root of a tree is at level 1, its children are at level 2, their children at level 3, and so on. The table below shows the results of several calls on an `IntTree` variable `tree`:

Levels	tree	Call	Value Returned
1		<code>tree.nodesAtLevels(2, 4)</code>	8
2		<code>tree.nodesAtLevels(4, 5)</code>	6
3		<code>tree.nodesAtLevels(1, 2)</code>	3
4		<code>tree.nodesAtLevels(3, 3)</code>	3
5		<code>tree.nodesAtLevels(7, 9)</code>	0
		<code>tree.nodesAtLevels(5, 9)</code>	3
		<code>tree.nodesAtLevels(1, 1)</code>	1

For example, `tree.nodesAtLevels(4, 5)` returns 6 because -8, 80, -2, 57, 1, and 27 are in that range of levels.

Your method should throw an `IllegalArgumentException` if the minimum passed is less than 1 or is greater than the maximum passed. It is legal for the minimum and/or maximum to be larger than the height of the tree; a tree has 0 nodes at any levels that exceed its height. An empty tree contains 0 nodes at any level.

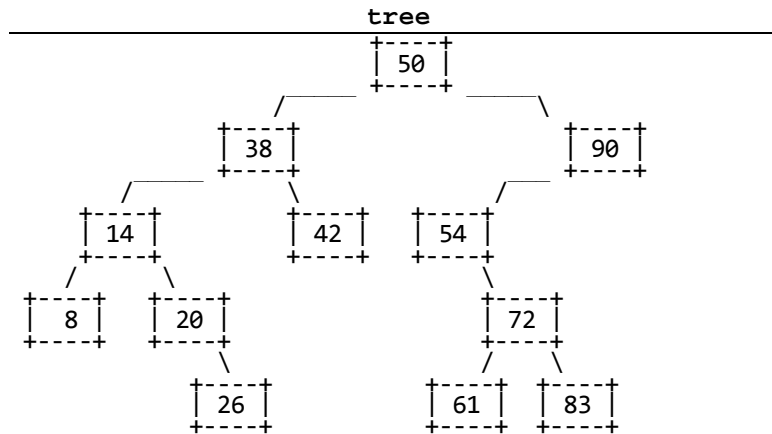
You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the class nor create any data structures such as arrays, lists, etc. Your method should not change the structure or contents of the tree being examined.

(The contents of the `IntTree` and `IntTreeNode` classes are shown on the next page for reference.)

7. Binary Tree Programming

Write a method `trim` that could be added to the `IntTree` class from lecture and section. The method accepts minimum and maximum integers as parameters and removes from the tree any elements that are not within that range, inclusive. For this method you should assume that your tree is a binary search tree (BST) and that its elements are in valid BST order. Your method should maintain the BST ordering property of the tree.

For example, suppose a variable of type `IntTree` called `tree` stores the following elements:



The table below shows what the state of the tree would be if various `trim` calls were made. The calls shown are separate; it's not a chain of calls in a row. You may assume that the minimum is less than or equal to the maximum.

<code>tree.trim(25, 72);</code>	<code>tree.trim(54, 80);</code>	<code>tree.trim(18, 42);</code>	<code>tree.trim(-3, 6);</code>
<pre> graph TD 50 --- 38 50 --- 54 38 --- 26 38 --- 42 54 --- 72 72 --- 61 </pre>	<pre> graph TD 54 --- 72 72 --- 61 </pre>	<pre> graph TD 38 --- 20 38 --- 42 20 --- 26 </pre>	

Hint: The BST ordering property is important for solving this problem. If a node's data value is too large or too small to fit within the range, this may also tell you something about whether that node's left or right subtree elements can be within the range. Taking advantage of such information makes it more feasible to remove the correct nodes.

You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the class nor create any data structures such as arrays, lists, etc. (Write your answer on the next page.)

Recall the `IntTree` and `IntTreeNode` classes as shown in lecture and section:

```

public class IntTreeNode {
    public 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, IntTreeNode right) {...}
}

public class IntTree {
    private IntTreeNode overallRoot;

    methods
}
  
```

7. Binary Tree Programming (writing space)

X. Extra Credit

Draw a picture of your TA using a data structure in his/her everyday life.

For example, you could draw a picture of the TA eating a `Stack` of pancakes.

Make sure to put the TA's name somewhere in the picture so we know who it is.

(This is not an art class, so any drawing that appears to reflect more than a few moments' effort will get the point.)
