CIS 211 Final Exam, Spring 2011
10:15am - 12:15pm, June 7, 2011

Instructions: Use the space provided for each answer. If necessary, use the back of the page or scratch paper (available at the front of the classroom). You must complete this exam independently, with no outside resources of any kind, except for the provided API reference and your sheet of handwritten notes. This exam consists of 8 questions, worth a total of 100 points.

HINT 1: Look over all problems before you start and manage your time well. This is not an easy test. Do what you can in the time you have.

HINT 2: Don’t forget to consult the provided API reference as much as necessary.
1. (10) **Binary Search Trees.** Show the binary search tree that would result from inserting the following items in this order: “twas”, “brillig”, “and”, “the”, “slithy”, “toves”. Assume an alphabetical sort order for the binary search tree (’a’ < ’b’ < ... < ’z’). (Note that “the” < “toves” < “twas”.)

2. (10) **Traversals.** Show the node orders resulting from each traversal in the following binary tree:

```
pre-order ________________________________________________

in order ________________________________________________

post-order ________________________________________________
```
3. (5) **GUI Layouts.** Circle the GUI that best matches the following code snippet:

```java
(JFrame frame = new JFrame("Hello");
frame.setLayout(new GridLayout(4,4));
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        JPanel p = new JPanel();
        if ((i + j) % 2 == 0) {
            p.setBackground(Color.WHITE);
        } else {
            p.setBackground(Color.BLACK);
        }
        frame.add(p);
    }
}
frame.setSize(400, 400);
frame.setVisible(true);
```
4. (15) **Recursive tracing.** What does `mystery(x)` print out for each input?

```java
private static int helper(int x)
{
    if (x > 0) {
        return helper(x / 2) + 1;
    } else {
        return 0;
    }
}

class Mystery{
    public static void mystery(int x)
    {
        System.out.println(helper(x));
    }
}
```

1
-----------------------------
5
-----------------------------
-5
-----------------------------
16
-----------------------------
5. (15) **Recursive programming.** Write a method `isPowerOfTwo(int x)` that returns true if (and only if) x is a power of two. **Your solution must use recursion.**

HINT 1: Every power of two is even, except for 1 ($2^0 = 1$).

HINT 2: If x is a power of two and x > 1, then x/2 must also be a power of two.

HINT 3: Remember that integer division truncates, so that $3/2 = 1$.

**Examples:**

<table>
<thead>
<tr>
<th>Method call</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>powerOfTwo(-2)</code></td>
<td>false</td>
</tr>
<tr>
<td><code>powerOfTwo(0)</code></td>
<td>false</td>
</tr>
<tr>
<td><code>powerOfTwo(1)</code></td>
<td>true</td>
</tr>
<tr>
<td><code>powerOfTwo(2)</code></td>
<td>true</td>
</tr>
<tr>
<td><code>powerOfTwo(3)</code></td>
<td>false</td>
</tr>
<tr>
<td><code>powerOfTwo(16)</code></td>
<td>true</td>
</tr>
<tr>
<td><code>powerOfTwo(2048)</code></td>
<td>true</td>
</tr>
</tbody>
</table>

```java
/**
 * Returns true if x is an integer power of two.
 * @return true if x is a power of two.
 */
public static boolean isPowerOfTwo(int x)
```
6. (15) Write a method `String mostCommonGrade(Map<String, String> grades)` that accepts one argument, a map from student names to grades, and returns the most common grade in the Map. In the case of a tie, any of the most common grades may be returned.

HINT: To solve this problem, you will need to construct an additional Map of your own. Feel free to refer to the Map interface in the API reference.

Example: Suppose the Map `grades` contains the following entries:

```
{"Twilight Sparkle":"Excellent", "Applejack":"Good",
 "Rainbow Dash":"Fair", "Pinkie Pie":"Good",
 "Fluttershy":"Very Good", "Rarity":"Very Good",
 "Spike":"Good"}
```

`mostCommonGrade(grades)` should return “Good”, since it occurs the most times.

```java
/**
 * Returns the most common grade value among all students.
 * @param grades map from student name (key) to student grade (value)
 * @return most common grade
 */
public String mostCommonGrade(Map<String, String> grades)
```
7. (15) **Background:** Priority queues are often implemented using binary trees that satisfy a “heap ordering” property: the value at each node is smaller than any of its descendants. Therefore, the smallest value is always at the root of the tree, making it easy to find and remove. (NOTE: the heap ordering property has nothing to do with binary search trees, which have a completely different ordering property.)

**Instructions:** Add the method `isHeapOrdered` to the IntTree class discussed in lecture. `isHeapOrdered` returns true if and only if the value at each node is smaller than all of its descendants. Your solution must not create any new objects of any type. (In other words, you may not use the “new” keyword.) You may use additional private methods that you write yourself, but should call no other IntTree methods.

**Examples:** The left tree is heap ordered. The right one is not, since 3 is below 4 and 7 is below 9.

```
/**
 * Returns true if and only if the tree satisfies the heap ordering property. (That is, the value at each node is smaller than the
 * values of its descendents.)
 * @return true if the tree satisfies the heap ordering property.
 */
public bool isHeapOrdered()
```
8. (15) Add the method \texttt{pruneToDepth(int d)} to the \texttt{IntTree} class discussed in lecture. \texttt{pruneToLevel} removes all nodes below depth \(d\), while leaving all other nodes where they were. If \(d\) is negative, \texttt{pruneToDepth} throws an \texttt{IllegalArgumentException}.

Your solution must not create any new objects of any type. (In other words, you may not use the “new” keyword.) You may use additional private methods that you write yourself, but should call no other \texttt{IntTree} methods.

**Example:** Suppose the \texttt{IntTree} initially looks like this:

```
        3
       /|
      2 7
     /  \
    1  5
   /  \
   9  6
```

The following table shows what the tree would look like after different calls to \texttt{pruneToDepth}.

<table>
<thead>
<tr>
<th>Method call</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{pruneToDepth(3)}</td>
<td>![Tree after pruneToDepth(3)]</td>
</tr>
<tr>
<td>\texttt{pruneToDepth(2)}</td>
<td>![Tree after pruneToDepth(2)]</td>
</tr>
<tr>
<td>\texttt{pruneToDepth(0)}</td>
<td>(empty tree)</td>
</tr>
<tr>
<td>\texttt{pruneToDepth(-5)}</td>
<td>\texttt{IllegalArgumentException} thrown</td>
</tr>
</tbody>
</table>

There is space to write your solution on the next page.
/**
 * Prune tree by removing all nodes that exceed a maximum depth.
 * @param d maximum depth of tree after pruning
 */
public void pruneToDepth(int d)