CIS 211 Final Exam, Winter 2011
3:15pm - 5:15pm, March 16, 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 your half-sheet of handwritten notes. This exam consists of 7 questions, worth a total of 100 points.

HINT: 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.

Below, we provide the Iterator and (partial) List interfaces, should you need them for reference:

interface Iterator<E>:

// Returns true if the iteration has more elements.
boolean hasNext();

// Returns the next element in the iteration.
E next();

// Removes from the underlying collection the last element returned by the iterator.
void remove();

interface List<E>:

boolean add(E e); // add to the end of the list
void add(int index, E element); // insert at specified index
void clear(); // remove all elements
boolean contains(Object o); // true if list contains the object
E get(int index); // get element at specified index
int indexOf(Object o); // index of first occurrence (or -1)
boolean isEmpty(); // list is empty (size() == 0)
Iterator<E> iterator(); // return iterator through this list
int lastIndexOf(Object o); // index of last occurrence (or -1)
E remove(int index); // remove at specified index
boolean remove(Object o); // remove first occurrence of object
E set(int index, E element); // set item at index to be given element
int size(); // number of elements in list
Below are the definitions of the Stack and Queue interfaces introduced during lecture. You are free to use these classes in place of the standard Java ones if you wish.

Creating a `Stack<E>`:

```java
Stack<Integer> s = new ArrayStack<Integer>;
```

**Stack<E> interface:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>push(value)</code></td>
<td>places given value on top of stack</td>
</tr>
<tr>
<td><code>pop()</code></td>
<td>removes top value from stack and returns it</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>returns number of elements in stack</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>returns true if stack has no elements</td>
</tr>
</tbody>
</table>

Creating a `Queue<E>`:

```java
Queue<String> s = new LinkedQueue<String>;
```

**Queue<E> interface:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>enqueue(value)</code></td>
<td>places given value on top of stack</td>
</tr>
<tr>
<td><code>dequeue()</code></td>
<td>removes value from front of queue and returns it; throws a <code>IllegalStateException</code> if queue is empty.</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>returns number of elements in queue</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>returns true if queue has no elements</td>
</tr>
</tbody>
</table>

The `IntNode`, `IntTree`, and `SearchTree` classes:

```java
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 subtree constructors
    // ...constructors...
}
```

```java
public class IntTree {
    private IntTreeNode overallRoot;
    // ...methods...
}
```

```java
public class SearchTree {
    private IntTreeNode overallRoot;
    // ...methods...
}
```

You will not need any constructors or other methods in `IntNode` or `IntTree`. `SearchTree` is identical to `IntTree`, except that all nodes are ordered, making it a binary search tree instead of just a binary tree.
1. (10) **Binary Search Trees.** Show the binary search tree that would result from inserting the following items in order: “Have”, “A”, “Happy”, “Saint”, “Patrick’s”, “Day”. Assume an alphabetical sort order (‘A’ < ‘B’ < … < ‘Z’).

**ANSWER:**

```
    Have
   /   \
  A    Saint
 /    /  \
Happy Patrick's
/    /    \
    Day
```

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

![Binary Tree Diagram]

**ANSWER:**

```
  pre-order  _5 2 1 4 3 7 6____________________________

  in order   _1 2 3 4 5 6 7____________________________

  post-order _1 3 4 2 6 7 5____________________________
```
3. (20) **Recursive tracing.** What does the following code print out for each input?

```java
public static void mystery(int x)
{
    if (x > 0) {
        mystery(x / 5);
        System.out.print(x % 5);
    }
}
```

2
5
25
36
4. (20) **Programming with Inheritance and Comparable.** An elf is a mythical, magical being. For the purposes of this problem, every elf has a name and a favorite color. The Elf class has the following public methods:

- **Elf(String name, String favoriteColor)**
  Constructs an Elf with the specified name and favorite color.

- **String name()**
  Returns the elf’s name.

- **String favoriteColor()**
  Returns the elf’s favorite color.

Design a new class called **Leprechaun**, which is a special kind of Elf. Leprechauns are similar to elves, except that every leprechaun has some number of gold coins. Every Leprechaun’s favorite color is “green”. When a leprechaun has 1000 coins or more, he appends the title “the Wealthy” to his name. This means that if a Leprechaun is constructed with the name “Sam”, then name() should return “Sam the Wealthy” when Sam has 1000 coins or more, and “Sam” otherwise.

In addition to the methods inherited from Elf, your class must include the following new methods:

- **Leprechaun(String name, int numCoins)**
  Constructs a Leprechaun with the specified name and initial number of coins.

- **void setNumCoins(int numCoins)**
  Sets the number of coins to the given value.

- **int numCoins()**
  Returns the number of coins the Leprechaun currently has.

Your Leprechaun class must also implement the **Comparable<Leprechaun>** interface, so that Leprechauns are ordered by the number of coins they own.

```java
public class Leprechaun extends Elf implements Comparable<Leprechaun> {
    private int numCoins;

    public Leprechaun(String name, int numCoins) {
        super(name, "green");
        this.numCoins = numCoins;
    }

    public String name() {
        if (numCoins >= 1000) {
            return super.name() + " the Wealthy";
        } else {
            return super.name();
        }
    }
}
```
public void setNumCoins(int numCoins)
{
    this.numCoins = numCoins;
}

public int numCoins()
{
    return numCoins;
}

public int compareTo(Leprechaun other)
{
    return numCoins - other.numCoins;
}
5. (15) Write a method void removeDuplicates(Queue<String> q) that removes all duplicates entries from a queue. When removing duplicate elements, the first occurrence of any given element should remain and the later ones should be removed. Elements must retain their respective orders.

Example: Suppose a queue contains the following elements before removeDuplicates:
front [1, 3, 2, 8, 2, 4, 3, 3, 1, 17] back
After calling removeDuplicates, it should contain:
front [1, 3, 2, 8, 4, 17] back
You may create AT MOST TWO auxiliary data structures (such as a Queue, Stack, TreeMap, or LinkedList). However, this problem can easily be solved with just one. For full credit, your solution must run in \( O(n) \) time where \( n \) is the initial length of the queue.

```java
/**
 * Removes all duplicate elements from the specified queue, keeping
 * only the first occurrences.
 */
public void removeDuplicates(Queue<String> q) {

    Map<String, Integer> m = new HashMap<String, Integer>();
    int queueSize = q.size();
    for (int i = 0; i < queueSize; i++) {
        String s = q.remove();
        if (!m.containsKey(s)) {
            m.add(s, 1);
            q.add(s);
        }
    }
}
```
6. (15) Add the method `isLucky` to the `IntTree` class discussed in lecture. `isLucky` returns true if and only if the `IntTree` has exactly four leaves. 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 lucky, since it has 4 leaves. The right one is not, since it has 5 leaves.

```java
/**
 * @return true if the tree has exactly four leaves.
 */
public boolean isLucky()
{
    return numLeaves(overallRoot) == 4;
}

private int numLeaves(IntNode root)
{
    if (root == null) {
        return 0;
    } else if (root.left == null && root.right == null) {
        return 1;
    } else {
        return numLeaves(root.left) + numLeaves(root.right);
    }
}
```
7. (10) Recall that the SearchTree class is identical to IntTree, except that the nodes are always ordered as a binary search tree. In other words, the value at any node is greater than all nodes in its left subtree and less than all nodes in its right subtree.

Add the method void removeNegatives() to the SearchTree class, which removes all nodes with negative values from the tree. All nodes with non-negative values must remain in the tree, and the tree must remain in sorted order. For full credit, your solution must run in \( O(d) \) time where \( d \) is the depth of the tree.

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.

**Example:** Calling removeNegatives() on the SearchTree to the left results in the SearchTree on the right.

```java
/**
 * Removes all negative values from the SearchTree, while maintaining
 * sorted order.
 */
public void removeNegatives()

public void removeNegatives()
{
    overallRoot = removeNegatives(overallRoot);
}

public IntNode removeNegatives(IntNode root)
{
    if (root == null) {
        return null;
    } else if (root.data < 0) {
        return removeNegatives(root.right);
    } else {
        root.left = removeNegatives(root.left);
        return root;
    }
}
```