CSE 12 Summer Session 2 Final Examination....................................2 September 2011

Name: ___________________________ Student ID: ___________________________

This exam is open-book -- you may refer to any book or any notes you have brought with you during the exam. However, you may **not** use a computer of any kind (including cell-phones) during the exam.

Score:

Problem 1: ______/6

Problem 2: ______/6

Problem 3: ______/3

Problem 4: ______/4

Problem 5: ______/4

Problem 6: ______/4

Problem 7: ______/3

Total: ______/30

Problem 1: Short-answers -- 6 points

1. Why would it not make sense to try to instantiate an object of an *interface* type? (Your answer cannot be “because it won’t compile” -- explain *why* the compiler prevents you from doing this.)

2. Both Mergesort and Quicksort divide the input array into two parts. Describe how the “partitioning” differs between these two algorithms.

3. Describe one similarity and one difference between interfaces and abstract classes.

4. When implementing a `DoublyLinkedList` class that implements `Iterable`, it makes sense to define a `DLLIterator` class as a **non-static inner class** of `DoublyLinkedList` -- why should this inner class be defined as **non-static**?

5. In the space below, declare a class called `MyClass` that takes a generic type parameter that can be any class that is a subclass of `String`. (The class doesn’t have to contain any variables or methods.)

6. If a class `C` implements the `Comparable` interface, then which one method must `C` implement? What does this method do, and what does it return?

Problem 2: Queues -- 6 points

Consider the following (incomplete) implementation of a **queue** that is implemented using a linked list of **nodes** instead of a ring buffer. Recall that a queue is a first-in-first-out (FIFO) data structure.

```
class Queue<T> {
      static class Node<T> {
            T _data;
            Node<T> _next;
      }

      private Node<T> _head, _tail;

      Queue () {
            _head = new Node<T>();
            _tail = _head;
      }

      // Adds the specified object to the TAIL of the queue.
      public void enqueue (T o) {
            ...  // See the code snippets below
      }

      // Removes and returns the object at the HEAD of queue.
      public T dequeue () throws NoSuchElementException {
            if (_head == _tail) {
                  throw new NoSuchElementException("Queue is empty!");
            }

            _head = _head._next;
            final T _data = _head._data;
            _head._data = null;
            return _data;
      }
}
```

Now, for each of the 6 possible implementations of `enqueue(o)` below, say whether it is correct or incorrect **given the code shown above, without any modifications or additions.** It is possible that some, all, or none of the implementations are correct. You will receive 1 point for every right answer, and you will *lose* one point for every wrong answer; hence, guessing is discouraged. The minimum score you can receive on this problem is 0.

```
public void enqueue (T o) {
      Node<T> node = new Node<T>();
      node._data = o;
      node._next = _head;
      _tail = node;
}
```

Correct or incorrect? __________

```
public void enqueue (T o) {
        Node<T> node = new Node<T>();
        node._data = o;
        _tail._next = node;
        _tail = node;
}
```

Correct or incorrect? __________

```
public void enqueue (T o) {
        Node<T> node = new Node<T>();
        node._data = o;
        _head._next = node;
        _head = node;
}
```

Correct or incorrect? __________

```
public void enqueue (T o) {
        Node<T> node = new Node<T>();
        node._data = o;
        node._next = _head;
        _head = node;
}
```

Correct or incorrect? __________

```
public void enqueue (T o) {
        Node<T> node = new Node<T>();
        _tail._data = o;
        _tail = _tail._next;
        _tail = node;
}
```

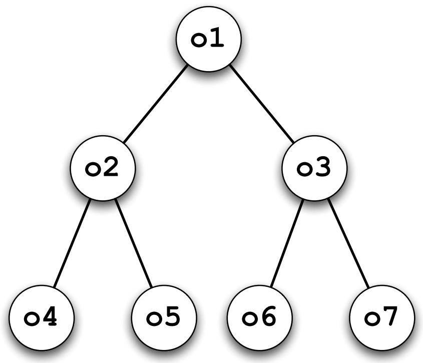
Correct or incorrect? __________

```
public void enqueue (T o) {
        Node<T> node = new Node<T>();
        _tail._next = node;
        _tail = node;
        node._data = o;
}
```

Correct or incorrect? __________

Problem 3: Binary search trees (BSTs) -- 3 points

Consider the **binary search tree (BST)** shown below:



3.1 -- 2 points

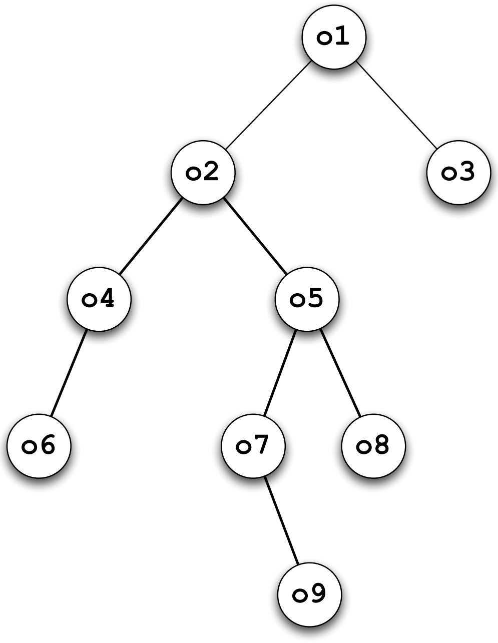
Assume that the BST above contains no duplicates. Based on the node structure shown above, order the objects `o1`, `o2`, ..., `o7` so that they are in **ascending order** (i.e., smallest element first, largest element last). For instance, if you think `o1` is the smallest, `o4` is the second-smallest, `o5` is the third-smallest, etc., then you should respond "`o1 < o4 < o5 <`...". If you are only certain about the relative order of a few pairs of elements, then list only those, e.g., "`o1 < o5,    o3 < o4`". (This helps us to assign partial credit.)

3.2 -- 1 point

Draw the BST that arises if node `o1` is removed.

Problem 4: Trees -- 4 points

The figure below shows a **tree**. Consider each node and its *set of ancestors*. We define the *set of ancestors* of a node *n* to consist of *n* itself, *n*'s parent, *n*'s parent's parent, and so on. For instance, the set of ancestors of node **o5** contains **o5**, **o2**, and **o1**. The *lowest common ancestor x* between two nodes *n* and *m* is the node *x* that is an ancestor of both *n* and *m*, such that no child of *x* is also an ancestor of both *n* and *m*. For instance, the lowest common ancestor between **o6** and **o7** is **o2**. Notice that, although **o1** is also a common ancestor of **o6** and **o7**, it is not the *lowest* common ancestor. For another example, the lowest common ancestor between **o3** and **o3** is just **o3** itself (since the set of ancestors of *n* includes *n* itself).



Now, examine the (incomplete) implementation of the **Tree** class shown on the next page. **Tree** contains a static inner-class **Node**. (For simplicity, we assume the tree is a binary tree; hence, each node contains a **_leftChild** and **_rightChild** pointer.) **Tree** also contains an instance variable **_root** that specifies the root of the tree. Write a method **findLowestCommonAncestor(n1, n2)** that returns the lowest common ancestor of nodes **n1** and **n2**, which are both non-**null** but are not necessarily distinct. If you find it useful, you are allowed to instantiate and use any of the collection classes from P5, e.g., a **Heap**, **BinarySearchTree**, **HashTable**, or **LinkedList**. Recall that their public interface contains methods **add(o)**, **contains(o)**, **size()**, **clear()**, and **remove(o)**.

```
class Tree {
      static class Node {
            Node _parent;
            Node _leftChild, _rightChild;
            ...
      }

      Node _root;
      ...

      Node findLowestCommonAncestor (Node n1, Node n2) {









      }
}
```

Problem 5: Heaps -- 4 points

As you know, heaps are **complete trees** in which every node contains one element that the user stored in the heap. Heaps are usually implemented using an array as the underlying storage, but in theory they could also be implemented using node objects that contain parent and child pointers. Consider the (incomplete) implementation of the node-based `DAryHeap` class shown below. `DAryHeap` contains a static inner-class `Node`. Each `Node` contains a `_parent` pointer, a `_data` pointer (to store an object the user adds to the heap), an array `Node[] _children` to store up to *d* children (for a *d*-ary heap), and finally a `_numChildren` variable to keep track of how many children the node actually has.

In the space allotted, implement the `trickleDown(node)` method for a max-heap. `trickleDown(node)` is analogous to the `trickleDown(index)` method discussed during lecture: The data stored in the specified `node` may possibly be "less than" the data in one (or more) of its children. It is `trickleDown`'s responsibility to "restore" the heap condition by recursively swapping `node._data` down through the subtree rooted at `node`.

```
class DAryHeap<T extends Comparable<? super T>> {
      ...

      static class Node<T extends Comparable<? super T>> {
            Node<T> _parent;
            Node<T>[] _children;
            T _data;
            int _numChildren;
      }

      void trickleDown (Node<T> node) {




      }
}
```

Problem 6: Time cost analysis -- 4 points

6.1 -- 1 point
In the method below, assume that the `BinarySearchTree` is **self-balancing** (e.g., using AVL rotations).

```
long someMethod (int N) {
      final BinarySearchTree<Integer> bst = new BinarySearchTree<Integer>();
      for (int i = 0; i < N; i++) {
            bst.add(i);
      }

      final long start = ArtificialClock.getNumTicks();
      for (int i = 0; i < N; i++) {
            bst.remove(i);
      }
      final long end = ArtificialClock.getNumTicks();
      return end - start;
}
```

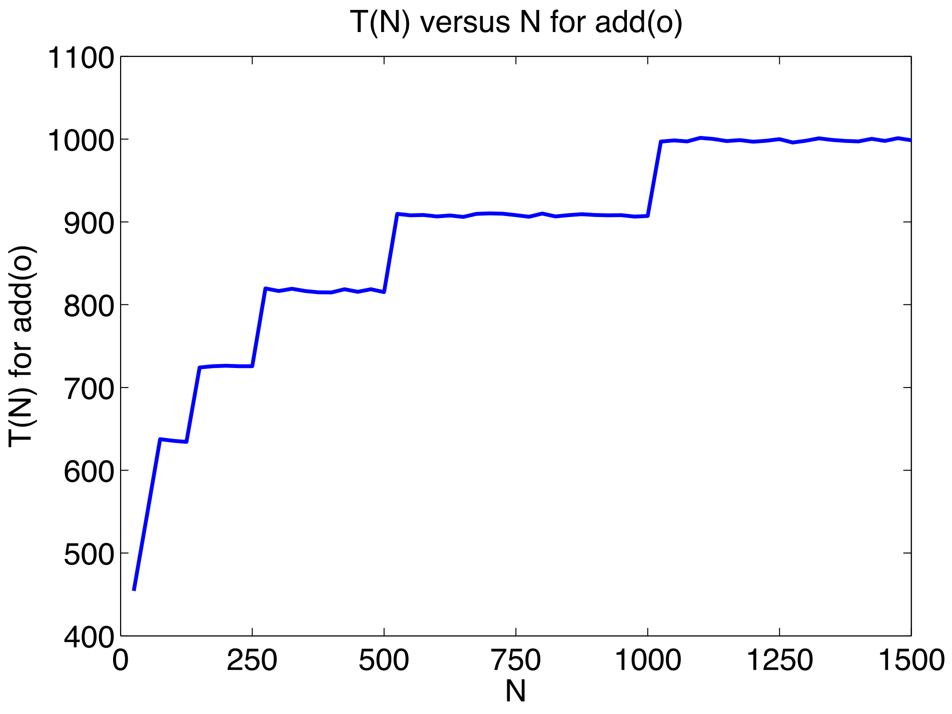
If you plot a curve of the return value of `someMethod(N)` versus `N` itself, what would be its asymptotic behavior? Justify your response.

6.2 -- 1 point
In the code below, assume the `HashTable` has the same interface as it did in P5 -- it offers an `add(o)` method which uses `o.hashCode()` to determine the "slot" in which to store `o`.

```
class A {
      static class B {
            int _num;
            B (int num) { _num = num; }
            int hashCode () {
                  return 17;  // Hash code is always the same!
            }
      }
      long someOtherMethod (int N) {
            final HashTable<B> hashTable = new HashTable<B>();
            final B[] lotsaB = new B[N];
            for (int i = 0; i < lotsaB.length; i++) {
                  lotsaB[i] = new B(i);
            }
            final long start = ArtificialClock.getNumTicks();
            for (int i = 0; i < N; i++) {
                  hashTable.add(lotsaB[i]);
            }
            final long end = ArtificialClock.getNumTicks();
            return end - start;
      }
}
```

If you plot a curve of the return value of `A.someOtherMethod(N)` versus `N` itself, what would be its asymptotic behavior? Justify your response.

6.3 -- 2 points
The graph below shows the worst-case time cost T(N) for adding a single element to a **binary heap** that already contains N - 1 elements. T(N) is approximately logarithmic; however, it has a "staircase" appearance to it -- for certain values of N, T(N) increases sharply, then it plateaus, and then at some later value of N it increases again.



Explain why the curve above has this "staircase" property. (The answer is not "noise".)

Problem 7: Fantasy data structure -- 3 points

7.1 -- 2 points

Suppose there existed a class called `FantasyDataStructure` that allowed the storage of `Comparable` data, as shown below, and that **all of its methods operated in time *O*(1).**

```
class FantasyDataStructure<T extends Comparable<? super T>> {
      // Adds the specified object to the FantasyDataStructure in O(1) time.
      void add (T o) {
            ...
      }
      // Removes the specified object from the FantasyDataStructure in O(1) time.
      void remove (T o) {
            ...
      }
      // Returns the value of the largest data element contained in the
      // FantasyDataStructure in O(1) time.
      T peekLargest () {
      }
}
```

In the space below, write code to use the `FantasyDataStructure` described above to sort a list of integers **in *O*(*n*) time** in the **worst case**.

```
// numbers should be in ascending order by the time sort(numbers) finishes.
void sort (int[] numbers) {












}
```

7.2 -- 1 point

Based on the lower-bound of the time complexity of the worst-case performance of any comparison-based sorting algorithm (given during lecture), explain why the `FantasyDataStructure` described above cannot possibly exist.