CSU2510H Exam 2 – Spring 2012

Name: ______________________________

Student Id (last 4 digits): ______________________________

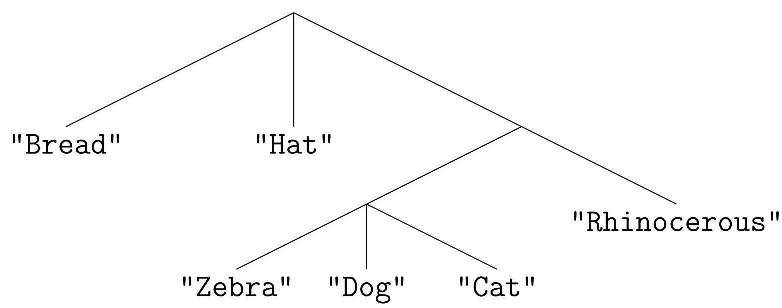
- All problems must be done in Java. You may use any Java we have used in class and in lab; anything else must be defined.
- You may write $c \rightarrow e$ as shorthand for writing `Tester.checkExpect`; the `Examples` class and `test` method around the tests are not required.
- To add a method to an existing class definition, you may write just the method and indicate the appropriate class name rather than re-write the entire class definition.
- We expect data *and* interface definitions.
- If an interface is given to you, you do not need to repeat the contract and purpose statements in your implementations. Likewise, you do not need to repeat any test cases given to you, but you should add tests wherever appropriate.
- Unless specifically requested, templates and super classes are *not* required.
- Some basic test taking advice: Before you start answering any problems, read *every* problem, so your brain can think about the harder problems in the background while you knock off the easy ones.

Problem	Points	/out of
1		/ 15
2		/ 22
3		/ 15
4		/ 15
5		/ 20
Total		/ 87

Good luck!

Problem 1 While trying to manage the grade database, Asumu hit upon a new idea for a data structure: the 2/3-tree. A 2/3-tree is either a leaf (which holds a `String` value), a 2-node, or a 3-node. A 2-node has two children, and a 3-node has three children. The twist is that a 2-node can't have any 2-nodes as children, and a 3-node can't have any 3-nodes as children. 15 POINTS

Here's an example of a 2/3-tree:



1. Design data, class, and interface definitions, in Java, to represent 2/3-trees. Construct the above example using your definitions.

2. Design the `height` method, which computes the height of the tree (i.e., the maximum distance from the root to a leaf). The example tree has a height of 4. You may find the function `Math.max` useful; it takes two numbers and produces the largest.

[Here is some more space for the previous problem.]

[Here is some more space for the previous problem.]

Problem 2 22 POINTS

1. Design a `TreeVisitor` interface for 2/3-trees, and implement the `accept` method in all of your classes from the previous problem. Make sure you design the interfaces and methods so that a visitor can return any type of data.

[Here is some more space for the previous problem.]

2. Design the `CountVisitor` class, which counts the number of leaves in a 2/3-tree (there are 6 in the given example.)

3. Design the `LongestVisitor` class, which produces the longest `String` in a 2/3-tree. The longest string in the example is `"Rhinocerous"`.

Problem 3 In mathematics, an **unordered** pair is a pair of values in which the order of the elements does not matter. So for example, if $\{3,4\}$ is an unordered pair, then it should be considered "the same" as the unordered pair $\{4,3\}$. Here is an implementation of `UnPair`, which represents unordered pairs. Notice that both `equals` and `hashCode` have been overridden.

15 POINTS

```
class UnPair<X> {
  X left;
  X right;
  UnPair(X left, X right) {
    this.left = left;
    this.right = right;
  }

  // Compute the hash of this unordered pair.
  public int hashCode() {
    return left.hashCode();
  }

  // Is this unordered pair the same as the given unordered pair?
  public boolean same(UnPair<X> p) { ... }

  // Is this unordered pair the same as the given object?
  public boolean equals(Object that) {
    return (that instanceof UnPair)
        && this.same((UnPair<X>)that);
  }
}
```

1. Implement the omitted `same` method that compares this unordered pair to a given unordered pair. This method should work regardless of the order of the pair, so for example `new UnPair<Integer>(3,4)` is the same (according to `same`) as `new UnPair<Integer>(4,3)`.

2. Assuming `same` and `equals` work as expected, is the given `hashCode` method valid? That is, does it satisfy or violate the law of `hashCode`? If it is valid, give an argument for *why*. If it is not, give a counter-example to the law of `hashCode`.

Problem 4 Lists are nice, but sometimes it would be better to have a list without an end. For example, we might want to model the days of the week as a list that wraps around at the end so that Monday follows Sunday. To do so, we need lists that are *circular*, and in order to make circular lists, we need to use mutation. Here's an idea for a list method that makes is possible to create cyclic lists: 15 POINTS

```
interface List<X> {
   // EFFECT: set the rest of the last cons in this list
   // to the given list.
   void setTail(List<X> ls);
}
```

It works as follows:

```
List<Integer> waltz =
  new Cons<Integer>(1,
    new Cons<Integer>(2,
      new Cons<Integer>(3,
        new MT<Integer>())));
```

At this point, `waltz` is just a list of three elements `1, 2, 3`. To make the list circular we can call `setTail` giving `waltz` as the new tail:

```
waltz.setTail(waltz);
```

Now `waltz` is a list of elements: `1, 2, 3, 1, 2, 3, 1, 2, 3, ...`, and so on, *ad infinitum*.

Of course, it's also possible to use `setTail` to update a list in a non-circular way. Imagine instead we had just done:

```
waltz.setTail(new Cons<Integer>(4, new MT<Integer>()));
```

In this case, `waltz` is now just `1, 2, 3, 4`.

Design an implementation of `setTail` that works as described. Revise the `List<X>` interface with any methods you need to add to make `setTail` work. (You'll notice that the effect statement for `setTail` implies that this list cannot be empty; if you try to `setTail` on an empty list, you may raise an exception.)

[Here is some more space for the previous problem.]

Problem 5 Iterators are an extremely useful concept. In particular, they aren't limited to iterating over the contents of data structures such as `ArrayLists`. We can also construct iterators directly for sequences we are interested in.

20 POINTS

1. Design the class `RangeIterator`, which takes two `Integers` as constructor arguments, and iterates over all of the `Integers` between them, including *both* end points. `RangeIterator` must implement the `Iterator` interface, given below.

```
interface Iterator<T> {
  Boolean hasNext();
  T next();
  // Java requires an Iterator to define remove,
  // but you do *not* need to implement remove.
}
```

[Here is some more space for the previous problem.]

2. Recall our definition of function objects:

   ```
   interface Function<T,U> {
     U call(T x);
   }
   ```

   Define the `Square` class, which is a `Function` that squares `Integers`.

3. Using this interface, design and implement the `MapIterator` class, which takes a `Function` and an `Iterator` as arguments, and iterates over the values produced by applying the function to each result produced by the iterator.

4. Now design the method `sumSquares`, which takes two `Integers`, and uses the classes you've just defined, along with a `for` or `while` loop, to compute the sum of the squares of all the numbers between the given `Integers`, inclusive.

[Here is some more space for the previous problem.]