CMSC 132A, Midterm 2

Spring 2018

NAME:_____

UID: _____

Question	Points
1	15
2	15
3	15
4	15
Total:	60

This test is open-book, open-notes, but you may not use any computing device other than your brain and may not communicate with anyone. You have 50 minutes to complete the test.

The phrase "design a program" means follow the steps of the design recipe. Unless specifically asked for, you do not need to provide intermediate products like templates or stubs, though they may be useful to help you construct correct solutions.

You may use any of the data definitions given to you within this exam and do not need to repeat their definitions.

Unless specifically instructed otherwise, you may use any Java language features we have seen in class.

You do not need to write visibility modifiers such as `public` and `private`.

When writing tests, you may use a shorthand for writing check-expects by drawing an arrow between two expressions to mean you expect the first to evaluate to same result as the second. For example, you may write `"foo".length()` $\rightarrow$ `3` instead of `t.checkExpect("foo".length(), 3)`. You do not have to place tests inside a test class.

Problem 1 (15 points). Assume for the moment that a person's name consists of a first name (also known as a given name) and a last name (also known as a surname or family name). This is a Euro-centric perspective on names, but we adopt it for the purposes of this problem.

Given the following representation of names, implement the `Comparable<T>` interface so that names are ordered alphabetically by last name and then by first name if the last names are the same. You may assume that the `compareTo` method in `String` orders strings alphabetically.

```
class Name implements Comparable<Name> {
  String first;
  String last;

  Name(String first, String last) {
    this.first = first;
    this.last = last;
  }
}
```

Problem 2 (15 points). A `ListIterator<X>` is an object that implements the `Iterator<X>` interface by delivering elements of a list given to the constructor. This is useful for computing something over the elements of a list, but sometimes it's necessary to know both the element and it's position in the list.

Assuming a working `ListIterator<X>` implementation (which you do not need to write), design a `ListPosIterator<X>` class (abbreviated `LPI`) that implements `Iterator<Pairof<X,Integer>>` that delivers pairs of elements and their position (starting at 0).

You may start with the following code, just indicate which section of code you are writing by adding a label of 1, 2, 3, or 4 corresponding to the number on the ellipses.

```
class LPI<X> implements Iterator<Pairof<X,Integer>> {
  ...1
  ListPosIterator(Listof<X> elems) { ...2 }
  bool hasNext() { ...3 }
  Pairof<X,Integer> next() { ...4 }
}
```

Problem 3 (15 points). Here is a class for representing text with a cursor located somewhere within the text. The `left` field holds the content to the left of the cursor and the `right` field holds content to the right.

```
class Text {
  String left;
  String right;
  Text(String left, String right) {
    this.left = left;
    this.right = right;
  }
}
```

Suppose that we want to consider text the same if it has the same contents, regardless of where the cursor is placed. For example `new Text("a", "bc")` should be considered the same as `new Text("ab", "c")`.

Override the `equals` and `hashCode` methods to use this notion of equality. Be sure to obey the fundemantal law of `hashCode` and make sure you `hashCode` method can distinguish at least some `Text` objects.

[Space for problem 3.]

Problem 4 (15 points). Here is a parameterized definition for binary trees of elements of type `T`, with the visitor pattern implemented:

```
interface BT<T> {
  <R> R accept(BTVisitor<T,R> v);
}

interface BTVisitor<T,R> {
  R visitLeaf(Leaf<T> l);
  R visitNode(Node<T> n);
}

class Leaf<T> implements BT<T> {
  <R> accept(BTVisitor<T,R> v) {
    return v.visitLeaf(this);
  }
}

class Node<T> implements BT<T> {
  T elem;
  BT<T> left;
  BT<T> right;
  Node(T elem, BT<T> left, BT<T> right) {
    this.elem = elem;
    this.left = left;
    this.right = right;
  }
  <R> accept(BTVisitor<T,R> v) {
    return v.visitNode(this);
  }
}
```

Design a visitor called `Mirror` that computes the mirror image of a binary tree, i.e. the mirror image of a non-empty tree has the mirror image of the left subtree as its right subtree and the mirror image of its left subtree as its right subtree.

You may not add anything to the `BT`, `Leaf`, `Node`, or `BTVisitor` definitions.

[Space for problem 4.]