CMSC 132A, Midterm 2
SOLUTION

Spring 2019

NAME:____

UID: ____

Question	Points
1	15
2	15
3	15
4	10
5	5
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). Here is the usual definition for binary trees:

```
interface BT<X> {}

class Leaf<X> implements BT<X> {}
class Node<X> implements BT<X> {
  X val;
  BT<X> left;
  BT<X> right;
  Node(X val, BT<X> left, BT<X> right) {
    this.val = val;
    this.left = left;
    this.right = right;
  }
}
```

Design the following method for binary trees:

```
// Produce an element satisfying p in this tree (if there is one)
Optional<X> findPred(Predicate<X> p);
```

[Space for problem 1.]

```
SOLUTION:

lf = new Leaf<>();
tr1 = new Node<>(4, lf, lf);
tr2 = new Node<>(6, lf, lf);
tr3 = new Node<>(5, tr1, tr2);

lf.findPred(n -> true) --> Optional.empty
t3.findPred(n -> (n % 2) == 0) --> Optioanl.of(4)
t3.findPred(n -> n > 5) --> Optioanl.of(6)
t3.findPred(n -> n > 8) --> Optioanl.empty

// In Leaf

Optional<X> findPred(Predicate<X> p) {
  return Optional.empty;
}

// In Node

Optional<X> findPred(Predicate<X> p) {
  if (p.test(this.val)) {
     return Optional.of(this.first);
  } else {
    Optional<X> r = this.left.findPred(p);
    return r.isPresent() ?
      r :
      this.right.findPred(p);
    }
  }
}
```

Problem 2 (15 points). Consider the following representation for dictionaries: collections of zero or more terms and their definitions (both of which are strings):

```
interface Dict {}

class Mt implements Dict {}

class ConsDict implements Dict {
  String term;
  String defn;
  Dict rest;
  ConsDict(String term, String defn, Dict rest) {
    this.term = term;
    this.defn = defn;
    this.rest = rest;
  }
}
```

Implement the visitor pattern for dictionaries. You must define a `DictVisitor` interface and add the appropriate methods to the `Dict` interface and its implementing classes.

Design a visitor that computes the total length of all the terms in a dictionary.

[Space for problem 2.]

SOLUTION:

```
interface DictVisitor<R> {
  R visitMt();
  R visitConsDict(ConsDict c);
}

// in Dict

// Visit this dictionary
<R> accept(DictVisitor<R> v);

// in Mt

// Visit this empty dictionary
<R> accept(DictVisitor<R> v) { return v.visitMt(); }

// in ConsDict

// Visit this non-empty dictionary
<R> accept(DictVisitor<R> v) {
  return v.visitCons(this);
}

// Visitor for computing the length of a dictionary
class LengthVisitor implements DictVisitor<Integer> {
  Integer visitMt() { return 0; }
  Integer visitCons(ConsDict c) {
    return 1 + c.rest.accept(this);
  }
}
```

Problem 3 (15 points). Here is a definition of pairs we've seen in class:

```
class Pair<X,Y> {
    X left;
    Y right;

    Pair(X left, Y right) {
        this.left = left;
        this.right = right;
    }
}
```

Override `equals` to implement structural equality (you only need to handle the case of the argument being a `Pair<X,Y>`). Override `hashCode` to (1) obey the fundamental law of `hashCode` consistent with your `equals` methods and (2) be able to distinguish at least some pairs.

Problem 4 (10 points). Design a class called `StrIter` that implements `Iterable<String>` and `Iterator<String>` interfaces. The idea is that a `StrIter` is used to iterate through singleton strings (strings of length 1) in a given string. For example:

```
for (String s : new StrIter("abcd")) System.out.println(s);
```

will print:

```
a
b
c
d
```

Here is a start:

```
class StrIter implements Iterable<String>, Iterator<String> {
  String str;
  StrIter(String str) {
    this.str = str;
  }

  public Iterator<String> iterator() { return this; }
  public boolean hasNext() { ... }
  public String next() { ... }
}
```

Complete the class definition. There are many ways to solve this problem. If you need to add fields, you may. For each piece of code you write, label where it should go in the class definition: fields, constructor, `next`, or `hasNext`.

Here are a few method signatures from the `String` class which may be helpful:

```
// Compute the length of this string
// "smiles".length() --> 6
int length()

// Compute substring of this starting at beginIndex going to endIndex - 1
// "smiles".substring(1, 5) --> "mile"
String substring(int beginIndex, int endIndex)
```

[Space for problem 4.]

```
SOLUTION:

// Soln 1: mutate the string field

// hasNext()
return this.str.length() > 0;

// next()
String s = this.str.substring(0,1);
this.str = this.str.substring(1, this.str.length());
return s;


// Soln 2: mutate an index

// fields
Integer i;

// constructor
this.i = 0;

// hasNext()
return this.str.length() > i;

// next()
String s = this.str.substring(i,i+1);
this.i = this.i + 1;
return s;


// Tests

StrIter si = new StrIter("ab");
si.hasNext() --> true
si.next() --> "a"
si.hasNext() --> true
si.next() --> "b"
si.hasNext() --> false
```

Problem 5 (5 points). The "Hamming distance" between two strings of equal length is the number of positions at which the corresponding letters are different. For example `"roons"` and `"pools"` have a Hamming distance of 2 because you can substitute just two positions in one string to obtain the other (`"r"` for `"p"` at position 0, and `"n"` for `"l"` at position 3). Hamming distance is used in a wide variety of domains including telecommunications, genetics, linguistics, and more.

In order to compute the distance, you've decided to use your `StrIter` class from problem 4. Assuming it works, complete the following code to compute the Hamming distance of `s1` and `s2`.

```
class Hamming {
  // Compute the Hamming distance between s1 and s2
  // ASSUME: s1 and s2 have the same length
  static Integer dist(String s1, String s2) {
    Iterator<String> i1 = new StrIter(s1);
    Iterator<String> i2 = new StrIter(s2);
    Integer d = 0;
    ...
    return d;
  }
}

Hamming.dist("karolin", "kathrin") --> 3
Hamming.dist("bob", "rob") --> 1


SOLUTION:

// In dist

while (i1.hasNext()) {
  if (!i1.next().equals(i2.next())) d++;
}
```