# CMSC 132A, Midterm 1

## Spring 2019

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).** A simple representation of a date is three numbers: a year, a month, and a day. For example, today's date can be represented with 2019, 3, and 8. Design a class representation for dates and design a method called `before` that determines whether a date comes (strictly) before a given date.

**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;
  }
}
```

A `OptString` is used to represent the result of looking a term up in a dictionary. There are two variants of an `OptString`: `None` indicates there is no result, while `Some` indicates there is a result and contains a string:

```
interface OptString {}

class None implements OptString {}

class Some implements OptString {
  String val;
  Some(String val) {
    this.val = val;
  }
}
```

Design a method for dictionaries called `lookup` that given a term, looks up the corrsponding definition and produces an `OptString`. If the term is not defined, it should produce `None`; if it is defined, it should produce a `Some` containing the definition.

You do not need to rewrite the interface and class definitions. Instead, for each piece of code, label which interface or class it belongs to.

[Space for problem 2.]

**Problem 3 (15 points).** Using the double-dispatch approach, design a method called `same` for `OptString` (defined in problem 2) that determines if an `OptString` is the same as a given one.

[Space for problem 3.]

**Problem 4 (15 points).** Here is a parameterized definition for binary trees of elements of type `T`:

```
interface BT<T> {}

class Leaf<T> implements BT<T> {}

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;
  }
}
```

Design a method called `map` that consumes a function (`Function<T,R>`) and produces a `BT<R>` that results from applying the function to each element of the tree. As a reminder, here is the definition of `Function<T,R>`:

```
interface Function<T,R> {
  // Apply this function to t
  R apply(T t);
}
```

Here are some tests (you don't have to write additional tests):

```
BT<Integer> l1 = new Leaf<>();
BT<Integer> b1 =
  new Node<>(1, new Node<>(2, l1, l1), new Node<>(3, l1, l1));
BT<Integer> b2 =
  new Node<>(1, new Node<>(4, l1, l1), new Node<>(9, l1, l1));

BT<String> l2 = new Leaf<>();
BT<String> b3 =
  new Node<>("A", new Node<>("BC", l2, l2), new Node<>("DEF", l2, l2));

l1.map(i -> i * i) ---> l1
b1.map(i -> i * i) ---> b2
l2.map(s -> s.length()) ---> l1
b3.map(s -> s.length()) ---> b1
```

You do not need to rewrite the interface and class definitions. Instead, for each piece of code, label which interface or class it belongs to.

[Space for problem 4.]