C212 Practice Final Exam (150 points)
Dec 11/13, 2023

Please read these directions before starting your exam.

This is a closed-note exam aside from your one page of notes, double-sided. You may not use any electronic devices to complete this exam, nor can you communicate with anyone besides the proctors and professor. *If you are caught cheating, you will receive an F in the course.*

For any question, unless specified otherwise, you may use any class without a corresponding `import`. E.g., if you want to use `HashMap`, you do not need to also import `java.util.HashMap`.

Unless otherwise stated, you do not need to spell out the "full design recipe", i.e., write the signature, documentation comments, and tests. Of course, doing so may aid you in your solution.

If you find a mistake, please raise your hand and let one of the proctors know; we will determine whether or not this is the case.

When you are finished, turn in your exam and notes sheet if you have one, then quietly exit.

You have 120 minutes to complete the exam.

Good luck!

Question	Points	Score
1	60	
2	30	
3	20	
4	20	
5	20	
Total:	150	

Name: ______________________________

IU Email: ______________________________

1. (60 points) Files on a computer are organized into *directories*, which are just locations for files to exist. Namely, directories can be nested inside of other directories. Therefore, we can categorize a directory as a data definition.

```
A Directory is one of:
 - File
 - new List<Directory>
```

We must then define a `File` as well, which contains two values: a name and a size (in bytes).

```
A File is a new File(String, Integer)
```

A directory that contains no files will have an empty list of subdirectories. We need a way of labeling that `File` and `Directory` are related, so we will define the `IContent` interface, which contains two methods to denote whether something is a file or a directory.

```
interface IContent {

  /**
   * Determines whether or not the implementing class is a File.
   */
  boolean isFile();

  /**
   * Determines whether or not the implementing class is a Directory.
   */
  boolean isDirectory();
}
```

(a) (10 points) Design the `Directory` class, which implements `IContent`, and stores, as an instance variable, a `List<IContent>`. Then, design the `File` class, which also implements `IContent` and stores the two relevant instance variables as described above. Of course, this means you will need to override the `isFile` and `isDirectory` methods respectively. **Write your code on the next page.**

```
class Directory                                      {










}

class File                                                  {










}
```

(b) (4 points) Design the `void add(IContent c)` method inside `Directory`, which receive a `File`/`Directory` and adds it to the list of content.

(c) (6 points) Implement the `Comparable` interface for `File` that returns a comparison of the file names. Remember that a class can implement multiple interfaces!

(d) (8 points) Design the `boolean isPresent(File f)` method inside `Directory`, which determines whether or not a file f exists inside the directory instance.

(e) (8 points) Design the `int countFiles()` method inside `Directory`, which returns the number of files that exist in that directory. It might make sense to write a recursive helper method to solve this problem.

(f) (8 points) Design the `int countDirectories()` method inside `Directory`, which returns the number of directories that exist in that directory. Do not include the directory itself in this total.

(g) (6 points) Design the `boolean isEmpty()` method inside `Directory`, which returns whether or not the directory contains any content.

(h) (10 points) Write coherent tests for your `Directory` and `File` classes. In particular, you should test the following methods: `isPresent`, `countFiles`, `countDirectories`, and `isEmpty`. It might make sense to create a couple of directories outside each test method, then test them inside those methods.

```
import static Assertions.assertAll;
import static Assertions.assertEquals;

class DirectoryTester {






  @Test
  void dirIsPresentTest() {





  }

  @Test
  void dirCountFilesTest() {





  }

  @Test
  void dirCountDirectoriesTest() {





  }

  @Test
  void dirIsEmptyTest() {





  }
}
```

2. (30 points) This question has five parts.

   (a) *(6 points)* First, write the `boolean isVowel(char ch)` method, which returns whether or not the character *ch* is a vowel.

```
isVowel('A') => true
isVowel('a') => true
isVowel('X') => false
isVowel('?') => false
```

   (b) *(6 points)* Next, write the `char swapVowelCasing(char ch)` method, which receives a character and, if it is a vowel, we swap its casing. That is, if it is uppercase, it becomes lowercase, and vice versa. Leave non-vowels alone. The `Character.toUpperCase`, `Character.toLowerCase`, `isUpperCase`, and `isLowerCase` methods will be helpful.

```
swapVowelCasing('a') => 'A'
swapVowelCasing('A') => 'a'
swapVowelCasing('b') => 'b'
swapVowelCasing('?') => '?'
```

(c) *(6 points)* Design the *standard recursive* `String swapVowelCasingString(String s)` method, which receives a string and swaps the casing of the vowels thereof.

(d) *(6 points)* Design the `String swapVowelCasingStringTR(String s)` as well as the `String swapVowelCasingStringTRHelper(...)` methods. The former acts as the driver to the latter; the latter solves the same problem that `swapVowelCasingString` does, but it instead uses tail recursion. Remember to include the relevant access modifiers!

(e) *(6 points)* Design the `String swapVowelCasingStringLoop(String s)` method, which solves the problem using either a `while` or `for` loop.

3. (20 points) We consider a *key string* to be the string obtained after alphabetizing the letters of a string. For example, the string `"deloop"` is a key string of the strings `"poodle"` and `"looped"`. Write the `static HashMap<String, List<String>> keyStringGroups(List<String> ls)` method, which maps all key strings to the strings in *ls* using the above criteria. We provide an example below. You **cannot** use this example in your tests.

```
ls = ["ant", "introduces", "poodle", "tan", "looped", "discounter", "nastier",
      "polled", "retains", "retinas", "reductions"]

keyStringGroups(ls) => {{"ant", ["tan", "ant"]},
                        {"deloop", ["poodle", "looped"]},
                        {"dellop", ["polled"]},
                        {"cdeinorsu", ["discounter", "introduces", "reductions"]},
                        {"aeinrst", ["retains", "retinas", "nastier"]}}

import static Assertions.assertAll;
import static Assertions.assertEquals;

class KeyStringGroupTester {

  @Test
  void testKeyStringGroup() {










  }
}
```

```
import java.util.*; // Import all necessary collections.

class KeyStringGroup {

  /**
   *
   *
   *
   */
  static HashMap<String, List<String>> keyStringGroups(List<String> ls) {











  }
}
```

4. (20 points) Two strings s_1 and s_2 are isomorphic if we can create a mapping from s_1 from s_2. For example, the strings `"DCBA"` and `"ZYXW"` are isomorphic because we can map D to Z, C to Y, and so forth. Another example is `"ABACAB"` and `"XYXZXY"` for similar reasons. A non-example is `"PROXY"` and `"ALPHA"`, because once we map `"A"` to `"P"`, we cannot create a map between `"A"` and `"Y"`. Write the `isIsomorphic` method, which determines whether or not two strings are isomorphic. Follow the design recipe from class. That is, write the purpose statement, followed by a sequence of examples, then the definition. **The skeleton code is on the next page.**

```
import static Assertions.assertAll;
import static Assertions.assertEquals;

class IsomorphicStringTester {

  @Test
  void isomorphicStringTest() {







  }
}
```

```
import java.util.*; // Import all necessary collections.

class IsomorphicString {

  /**
   *
   *
   *
   */
  static boolean isIsomorphic(String s1, String s2) {

















  }
}
```

5. (20 points) Oh no! Joshua's cat, Nebraska, has scratched part of this exam away and we need you to fix the missing code. Fill in the missing code for this merge sort implementation. Note that this is a *in-place* implementation of the merge sort, meaning that we modify the list in-place, rather than returning a new one. Only the `mergeSort` and `merge` methods should be filled in.

```
import java.util.List;

interface IMergeSort<____________________> {

  List<___> mergeSort(List<___> ls);
}


class InPlaceMergeSort<___ extends Comparable<___>> implements IMergeSort<___> {

  private void mergeSortHelper(List<___> ls, int low, int high) {
    if (low < high) {
      int mid = ______________________;
      mergeSortHelper(ls, ________, ________);
      mergeSortHelper(ls, ________, ________);
      merge(ls, low, mid, high);
    }
  }

  private void merge(List<___> ls, int low, int mid, int high) {
    List<___> left = new ArrayList<>();
    List<___> right = new ArrayList<>();

    for (int i = low; i <= mid; i++) { ______________________________; }
    for (int j = mid + 1; j <= high; j++) { ______________________________; }

    int mergedIdx = _________;
    int i = 0;
    int j = 0;

    while (______________________________________________) {
      if (left.get(i).compareTo(right.get(j)) < 0) {
        ls.set(mergedIdx++, ____________________);
      } else {
        ls.set(mergedIdx++, ____________________); }
    }

    while (_______________________) { ls.set(mergedIdx++, left.get(i++)); }
    while (_______________________) { ls.set(mergedIdx++, right.get(j++)); }
  }
}
```

Scratch work

Scratch work