COSC-211: DATA STRUCTURES
HW5: BINARY SEARCH TREES

Due Friday, March 29, 11:59pm

Reminder regarding intellectual responsibility: This is an individual assignment, and the work you submit should be your own. Do not look at anyone else's code, and do not show anyone your code (except for me and the course TAs).

1 The Assignment

1.1 Binary Search Tree Implementation

In this assignment, you will write a binary search tree from scratch. Your tree should hold keys of type `String` and values of type `Integer` (I encourage you to make your BST generic, but you are not required to do so for this assignment). You can use the `compareTo` method in the `String` class to determine which of two keys is larger.

Your tree should be in a class called `BinarySearchTree.java` (**exactly** like that, capitalization and all). In this file, you should include the following methods:

1. A method called `add` that takes a `String key` and an `Integer value` as input and inserts the `<key, value>` pair into the appropriate position in your tree. If `key` already appears in the tree, the method should replace the existing value with the new value.

2. A method called `remove` that takes a `String key` as input and removes the associated `<key, value>` pair from the tree. The method should return the `value` if it successfully removed `key` from the tree, and it should return `null` if it did not remove `key` from the tree (i.e., if `key` was not in the tree).

3. A method called `lookup` that takes a `String key` as input and determines whether `key` appears in the tree. The method should return the associated `value` if the tree contains `key` and `null` otherwise.

4. A method called `inOrderTraverse` that, when called on the root, prints all `<key, value>` pairs in the tree in increasing order, with each pair appearing on a new line in the format `(key, value)`. This method should have no input parameters and should not return anything.

Your methods must have **exactly the names, input parameters, and return types** specified above.

You might decide that you want to write additional methods, include fields, write additional classes (for example, you probably will want to write a `Node` class as we discussed in class), or any other number of things. This is all fine, provided that the four methods specified above do what they are supposed to do. Any methods or fields that you add to the `BinarySearchTree` class should be private.

You likely will want to write some code to test your binary search tree before you submit it. This code should be in a *separate Java file*; do *not* include test code in your `BinarySearchTree.java`. Some things to think about when you are testing your code: What happens when you try to call one of your methods on an empty tree? Does `add` work properly when you try to insert a key that is already in the tree? Does removing the root work properly? This is not a comprehensive list of cases to check; you are responsible for making sure that you tree always works as described above.

1.2 Word Counts

An interesting use for a dictionary is to count the number of occurrences of each word in a text document. The idea is that a key represents a word, and its associated value is the number of times the word appears in the document.

Your goal in this section is to write a program, called `WordCount.java`, that will perform this task. Specifically, you will read through a text document, one word at a time (see Section 1.2.1 below for tips on how to do this). You should convert each word to lowercase (so that, for example, "Hello" and "hello" are counted as the same word). If the word is not yet in your dictionary, add a new `<key, value>` pair. If the word already appears in your dictionary, update its associated word count `value` accordingly.

Your program should print out a list of all words that appear in the text file you read, in alphabetical order, with their word count.

You can find a sample text file at:
https://kgardner.people.amherst.edu/courses/s19/cosc211/assignments/hw5/sample.txt.
This particular file contains the entire text of the novel "Pride and Prejudice," by Jane Austen. You can find lots of other text files to play with at Project Gutenberg (https://www.gutenberg.org/) (be sure to use the ASCII text format, not the UTF-8 text format).

1.2.1 Reading from a file

One of the easier ways to read from a file involves using a `Scanner`. You can use the line:

```
Scanner sc = new Scanner(new File("myfile.txt"));
```

This will create a `Scanner` object that reads test out of the file called `myfile.txt`. The command `sc.hasNext()` checks whether there is anything else to read from the file, and the command `sc.next()` reads and returns a `String` containing the next word in the file.

In order to use the `Scanner` and `File` classes, you will need to import them at the top of your program:

```
import java.util.Scanner;
import java.io.File;
```

You also will need to handle the possibility that an error could occur when reading from the file. The easiest way to do this involves putting your code inside a `try/catch` block. This looks something like the following:

```
try {
    // whatever code uses the File class
}
catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

This piece of code says that we want to "try" to do whatever code is contained in the `try` block. If, in the course of running this code, a `FileNotFoundException` occurs, we will move to the `catch` block and execute the code in the `catch` block. In order to do this, we also have to import the error that we're catching:

```
import java.io.FileNotFoundException;
```

1.2.2 Working with Scanners

By default, `Scanners` are delimited by whitespace, meaning that a "word" is a unit of characters separated by a space, tab, or newline. You may want to change the delimiters of the `Scanner`, for example to remove punctuation. The command

```
sc.useDelimiter("([^\\p{IsAlphabetic}^\\p{IsDigit}]*\\s+
                     [^\\p{IsAlphabetic}^\\p{IsDigit}]*)|--");
```

will accomplish this. This command says to use as a delimiter any sequence of characters that fits the following pattern:

- `[^\\p{IsAlphabetic}^\\p{IsDigit}]*`: zero or more (that's the `*`) characters that are not alphabetic (that's the `^\\p{IsAlphabetic}`; the `^` is a negation and the `\\p{IsAlphabetic}` identifies whether the character is a letter) and not a digit,
- `\\s+`: Followed by one or more whitespace characters (the `\\s` indicates whitespace, and the + indicates one or more),
- `[^\\p{IsAlphabetic}^\\p{IsDigit}]*`: followed by zero or more characters that are not alphabetic and not a digit.

(You can search the phrase "regular expressions" if you want to learn more about how to construct and interpret this kind of pattern.)

Update 3/26/19: A few of you have noticed that the above delimiter doesn't work properly for you; instead of getting all words and no punctuation, you got all punctuation and no words. Obviously that isn't what we want! If you're having this issue, please feel free to skip the delimiter entirely and run the Scanner with its default delimiter, which separates words on white space. If you do this, you'll see some words with punctuation attached, so, for example, "hello" and "hello," (with a comma at the end) will be counted as different words.

Conrad suggested the following alternative delimiter, which worked properly for him:

```
sc.useDelimiter("(\\W*\\s+\\W*)|--");
```

Feel free to use this, any other variation that works for you, or no special delimiter at all.

2 Submit your work

Make sure you **test your code thoroughly** before submitting it. Code that does not compile will not receive credit.

Submit all of your Java files using either the submission web site or (from `remus/romulus`) the command line:

```
$ cssubmit *.java
```

This assignment is due on Friday, March 29, 11:59pm.