COSC-211: DATA STRUCTURES
HW5: BINARY SEARCH TREES

Due Thursday, April 1, 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 Getting Started

Create a new directory/project for yourself, and download the following ZIP archive:

`http://bit.ly/cosc-211-21s-hw5`

Extract/copy the files in this archive into your new directory/project. You will find the following files:

- `Dictionary.java`
  The interface that matches our *Dictionary* ADT.
- `sampletext.txt`
  A text file, for use in the second part of the assignment.

You'll notice that we've given you much less starter code than usual, for this assignment. That's because we want to give you practice implementing a binary search tree from start to finish. You'll find lots of information below about how to name your classes and methods; it's very important to follow these naming rules exactly!

2 The Assignment

2.1 Binary Search Tree Implementation

In this assignment, you will write a binary search tree from scratch. Your tree be generic, and store keys of type `K` and values of type `V`, where `K` must extend `Comparable` (thus allowing you to use the `compareTo` method to determine which of two keys is larger).

Your tree should be in a class called `BinarySearchTree.java` (**exactly** like that, capitalization and all). The class declaration should look like:
`public class BinarySearchTree<K extends Comparable<K>,V>`.
In your `BinarySearchTree` class, you should include the following methods:

1. A method called `add` that takes a `K key` and an `V 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, and return the old value.

If `key` was not already in tree, this method should return `null`. (Notice that this is a little different from the version of `add` that we discussed in class; our in-class version refused to add a `<key,value>` pair if the key was already present, whereas this version should replace the old value with the new value.)

2. A method called `remove` that takes a `K 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 `K 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 will want to write a `Node` 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.

As usual, you should test your `BinarySearchTree` thoroughly before submitting it. Write yourself some test code, modeled after the tester from HW3. For your reference implementation, any class that implements Java's `Map` interface is a good choice (for example, `TreeMap`). You can write yourself a wrapper class for `TreeMap`, as we've done in the past for other reference implementations.

2.2 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. You should convert each word to lowercase (so that, for example, "Hello" and "hello" are counted as the same word). The `toLowerCase()` method in the `String` class can help you with this. You should also remove any non-alphabetical characters from the word, so that punctuation is not included as part of the word. If your string is stored in variable `word`, you can do the punctuation removal with the line:

```
word = word.replaceAll("[^a-zA-Z]","");
```

This says to take all characters that are not in the range `a-z` or `A-Z` and replace them with the empty string, `""`.

Once you've removed non-alphabetical characters and converted the word to lowercase, add it to the dictionary. 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.

The HW downloads include a text file that 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).

3 Submit your work

Submit your work via Gradescope. You should submit your `BinarySearchTree.java`, `WordCount.java`, and any other supporting classes you've written that are needed for your `BinarySearchTree` and `WordCount` to run (e.g., `Node.java`). Do not submit your test code.

A note about the autograder: the autograder will assume that you've submitted both a `BinarySearchTree.java` and a `WordCount.java`. However, we *highly* recommend that you run your `BinarySearchTree` through the autograder to check if it passes our (extremely basic) compile-and-run test before you move on to writing your `WordCount`. If you submit your complete `BinarySearchTree.java` along with a `WordCount.java` that just contains an empty main method, this should pass our autograder tests.

This assignment is due on Thursday, April 1, 11:59pm.