Data Structures in Java: Final Exam Capstone Project
Word Replacer

1. **Objective**

   Your goal is to use several data structures and a graph algorithm you developed this semester to develop a program that performs a global search and replace in a text file. The program reads in the input text file and a text file containing word/replacement pairs. These pairs must be stored in a map, and we developed three data structures, namely the BSTreeMap, RBTreeMap, and MyHashMap, that implement the MyMap interface and work well with this task. With proper object-oriented programming techniques, specifically the use of polymorphism, you can create any one of these data structures and refer to it from a MyMap reference. Though all three classes implement the same interface, their methods are implemented very differently, leading to different execution times on the computer. You will time your program's execution to see how each of the data structures performs.

2. **Problem**

   You will develop a command line Java application that performs a global search and replace on a text file. Suppose you have the simple word replacement rules defined below:

```
Alice -> Bob
supermarket -> diner
```

   Now suppose you have the following text file:

```
Alice will meet Jennifer at the supermarket.
```

   Upon completion of the program, the new contents of the text file are displayed on the screen.

```
Bob will meet Jennifer at the diner.
```

   For simplicity, the program will work on whole words only. So, you can consider a word a sequence of characters that contains only capital and/or lowercase letters. Substrings of words are not considered separate words: e.g. the rule "`where -> body`" should **not** change "`nowhere`" to "`nobody`". No rules containing strings with multiple words, such as "gas station", will be tested, and the solution need not be implemented. There will be no contradictory rules such as "`Alice -> Bob`" and "`Alice -> Ben`", and no duplicate rules will be present.

The name of the public class **must be WordReplacer**, and it must reside in a file named **WordReplacer.java**.

Parsing Command Line Arguments
The program will take in three command line arguments. If the number of arguments supplied is incorrect, the program will print the usage message

```
"Usage: java WordReplacer <input text file> <word replacements file> <bst|rbt|hash>"
```

and a newline character to stderr and exit in failure.

If the number of command line arguments is correct, the program will then verify that the input text file exists. If it does not, the program will print the message

```
"Error: Cannot open file '" + inputTextFile + "' for input."
```

and a newline character to stderr, where `inputTextFile` contains the name of the text file the user is attempting to open, and exit in failure.

The program will then verify that the word replacements file exists. If it does not, the program will print the message

```
"Error: Cannot open file '" + wordReplacementsFile + "' for input."
```

and a newline character to stderr, where `wordReplacementsFile` contains the name of the text file the user is attempting to open, and exit in failure.

The program will then try to parse the command line for which data structure the user wishes to use. Valid strings are “bst”, “rbt”, and “hash”. Any other string including those with different capitalization is considered invalid, causing the program to print the message

```
"Error: Invalid data structure '" + dataStructure + "' received."
```

and a newline character to stderr, where `dataStructure` contains the data structure string name, and exit in failure.

Creating the Map
After validating the command line arguments, you’ll need to instantiate either a BSTreeMap, RBTreeMap, or MyHashMap. You **must** create a reference to the interface rather than the class itself, as in:

```
MyMap<String, String> map = new BSTreeMap<>();
```

No credit will be given if your map variable is not of type MyMap. The key-to-value mapping is String-to-String. The key will be the original word, and the value will be the replacement word.

Parsing the Input Files

Each line in the word replacements file will contain a word followed by a space, ->, space, and a word. Each word is a sequence of uppercase and/or lowercase letters only, as shown below. The rules are case-sensitive.

```
Alice -> Bob
Bob -> Carly
```

As you read each line, store the key-value mapping in the data structure the user wishes to instantiate.

While each line of the input file is guaranteed to be syntactically correct, it is possible, though, that the data can exhibit transitive dependencies. Upon processing the data above in your map, both `Alice` and `Bob` should be replaced with `Carly`.

Furthermore, the data can have cycles. Consider the following word replacements file:

```
Alice -> Bob
Bob -> Carly
Carly -> Alice
```

In the file above, we see the cycle `Alice -> Bob -> Carly -> Alice`. A replacement of a word with itself like `Alice -> Alice` immediately creates a cycle. Because of the potential presence of transitive dependencies and cycles, you should model the rules as a graph and employ relevant algorithms while parsing the word replacements file. If a cycle occurs, your program must detect it, print an error message to standard error, and exit in failure. The error message must be:

```
"Error: Cycle detected when trying to add replacement rule: " + a + " -> " + b
```

where `a` is the word before the `" -> "` and `b` is the word after it. Rules should be added in the order in which they appear in the word replacements file. The first rule which creates a cycle is the one that must be printed in this error message. In the example above, the output must be

```
Error: Cycle detected when trying to add replacement rule: Carly -> Alice
```

Note, transitive dependencies might be extremely long, so optimize your algorithm accordingly.

If an IO error occurs, while reading the word replacements file, your program must print the error message

```
"Error: An I/O error occurred reading '" + wordReplacementsFile + "'."
```

to standard error and exit in failure. `wordReplacementsFile` is, of course, the actual file name of the word replacements file. This scenario is difficult to test in practice, so just ensure your program can handle it, if the condition does arise.

After the map has been populated, your program must read the input text file, line-by-line, word-by-word. Again, our definition of a word is a sequence of characters that contains only capital and/or lowercase letters. As you read the file, when you come across a letter, keep building up the word until you come across the first non-alphabetic character. Ignore non-alphabetic characters and start parsing the next word when you reach a letter again. Use `Character.isLetter()` to test each character.

If the word has a replacement word in the map, it should be replaced in the output, accounting for transitive dependencies. Otherwise, if the word does not have a replacement, it must be left as is. Furthermore, all other characters that are not part of a word must also remain unchanged. You must use a StringBuilder to build up the text that will be printed on the screen. Use of the `+=` operator will result in abysmal performance.

If an IO error occurs, while reading the input text file, your program must print the error message

```
"Error: An I/O error occurred reading '" + inputTextFile + "'."
```

to standard error and exit in failure. `inputTextFile` is, of course, the actual file name of the input text file. Once again, this scenario is difficult to test in practice.

Displaying the Output

The output is simply the modified text displayed on standard out. If no words were replaced, the output must match the contents of the input text file, with a new line character appended at the end. Even if the input file does not have a trailing new line character, your program must print one anyway, so that the prompt starts at the beginning of the line in the terminal window. A single `System.out.printf()` is needed, assuming you have used a StringBuilder to create the new String. Use the format string `"%s\n"`, since otherwise Windows users will end up with "\r\n" at the end of the output.

There must **never be any whitespace characters**, such as extra spaces or tabs, at the end of a line of output. After the last visible character on a line, the cursor must be placed on the next line.

3. Sample Input / Output

Consider the following text inside the input file `input1.txt`.

```
Jake went on a nice walk with Alice, and later Bob joined.
Meanwhile, Justin was having lunch with Carl.
Peter sat down under a tree to read the newspaper.
```

The replacement file `replacements1.txt` contains the following data:

```
Alice -> Bob
Carl -> Bob
Justin -> Alex
Peter -> Alice
```

Assuming `input1.txt` and `replacements1.txt` are found in the same folder as the compiled class files, the program is run as follows:

```
java WordReplacer input1.txt replacements1.txt bst
```

The expected output for this particular test case is shown below.

```
Jake went on a nice walk with Bob, and later Bob joined.
Meanwhile, Alex was having lunch with Bob.
Bob sat down under a tree to read the newspaper.
```

Each line of the output ends with a newline character.

We have provided you with one additional sample test.
The output of your program when run on `sample_input.txt` and `sample_replacements.txt` should match `sample_output.txt` exactly.

4. Requirements

You must use the supplied implementations of the BSTreeMap, RBTreeMap, and MyHashMap. You **must not modify** these files in any way. You must write all your code in WordReplacer.java. We will copy the source for the data structures into the same folder as your source code when grading your submission. If you wish to write additional classes to help you encapsulate the data you process, put the non-public class inside WordReplacer.java.

You are permitted to import only from the following packages:

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
```

You **must not import** any other packages from the Java API. They will be stripped away before grading, and your code will not compile.
You are **expected to comment your code**. You must use Javadoc for every method. Additionally, inline comments are recommended.

You may use the Internet to help with various parts of this assignment, including reading from a file, working with exceptions, etc. Be sure to cite the source in an inline comment above the lines of code you reference. You may not consult with another human. You may not use openai.com or any other AI tools. Since the project is open-ended, you can solve the problem in many different ways. We will use MOSS to check your code against every other student's code in the class. **If your code is too similar to someone else's in the class, you will fail the course.** Please do everything possible to avoid this scenario!

5. Timing Your Work and Evaluating the Results

After you are confident you are getting the correct output for smaller files, try processing the input file `warandpeace.txt` and the word replacements file `warandpeace_replacements.txt` supplied in EdStem. Upon completion, the program should output the capitalized version of Leo Tolstoy's "War and Peace", with the 562,492 words in the file replaced with their uppercase equivalents. Time your program from the command line as follows:

```
time java WordReplacer warandpeace.txt warandpeace_replacements.txt bst > /dev/null
time java WordReplacer warandpeace.txt warandpeace_replacements.txt rbt > /dev/null
time java WordReplacer warandpeace.txt warandpeace_replacements.txt hash > /dev/null
```

Note that redirecting standard output to `/dev/null` is required to more accurately time the execution of your algorithm and data structure, as printing to the screen often takes significantly longer than performing operations requiring no visible output. When you run the command with `> /dev/null`, no output will appear on the screen.

You will see something like the following on Cygwin/Linux:

```
real    0mD.DDDs ← use 'real' value, and take the average of 5 executions
user    0mD.DDDs
sys     0mD.DDDs
```

You will see something like the following on a Mac terminal:

```
D.DDs user D.DDs system DD% cpu D.DDD total ← use 'total' value, and take
the average of 5 executions
```

You should time your program 5 times for each data structure used, and take the average execution time, since it will likely vary every time you run the program. The real value that appears tells you how much time it takes your program to fully complete. Each D represents a digit, so you might see something like

```
real    0m0.287s
```

which means the program took 287 ms to complete on that particular run.

Answer the following questions in the provided readme.txt file template.

1. How long does your program take to produce the answer when using the bst, rbt, and hash flags? Copy the results into this readme file.
2. What data structure do you expect to have the best (fastest) performance? Which one do you expect to be the slowest? Do the results of timing your program's execution match your expectations? If so, briefly explain the correlation. If not, state what run times deviated and briefly explain why you think this is the case.

6. Submission

Create a zip file, called WordReplacer.zip, containing only WordReplacer.java and readme.txt (ensure the file names have the exact capitalization), and upload it to Canvas.