CSE 101
Introduction to Data Structures and Algorithms
Programming Assignment 8

In this project you will re-create the Dictionary ADT from pa7, but now based on a Red-Black Tree. Red black trees are covered in Chapter 13 of the text, and will be discussed at length in lecture. All relevant algorithms for RBTs (and BSTs) are posted on the webpage under Examples/Pseudo-code. Aside from having a RBT as its underlying data structure, your Dictionary ADT will have only slight changes to its interface. The recommended approach for this project is to just copy Dictionary.cpp from pa7 and make the necessary changes, but you can start from scratch if you feel it is necessary. The header file Dictionary.h is posted in Examples/pa8. It's most significant difference from the header file for pa7 is a new field in the `Node` class of type `int` called `color`. Other than that, the only difference is a new section for RBT helper functions. Although these functions are listed as optional, and you may make changes as you like, you should consider them as absolutely necessary for this project. Feel free to add other helper functions if you find it useful.

You will create two top-level clients in this assignment. The first will be called Order.c, with the same specifications as in pa7. No changes should be necessary from that project. Again, several pairs of input-output files are given in Example/pa8, along with a random input file generator. Note that the input files are identical to those of pa7, but the paired outputs are different. In particular, the output file sections giving all keys in a pre-order traversal are different, since the trees are now balanced by the RBT algorithms. The output is also different in that function `preOrderString()` indicates which keys are stored in red Nodes.

The second top-level client will be called WordFrequency.cpp. It will read in each line of a file, parse the individual words on each line, convert each word to all lower case characters, then place it (as `key`) in a Dictionary. Individual words in the input file may be repeated however. The number of times a given word is encountered (its frequency) will also be stored (as `value`) in the Dictionary. Thus, as your program is reading in words, it should first check to see if the word (`key`) is already present, using `contains()`. If it is a new word, add it using `setValue()`. If it already exists, increment the corresponding value by calling `getValue()`. Recall that the `getValue()` function returns a reference to a value, which can then be used to alter that value. Use the example FileIO.cpp posted in /Examples/C++/FileIO as the starting point for WordFrequency.cpp, since much of what you need is already there. The program FileIO.cpp contains a string variable called `delim`, which is initialized to be a single space.

```
string delim = " ";
```

This is the *delimiter* used by the string functions `find_first_of()` and `find_first_not_of()` to determine which characters belong to tokens, and which do not. Thus FileIO.cpp tokenizes the file around spaces. Your program WordFrequency.cpp will instead tokenize around a larger set of characters. The words in our file will be substrings that contain only alphabetic characters. To accomplish this, you can reset `delim` as follows.

```
string delim = " \t\\\"\',<.>/?;:[{]}|`~!@#$%^&*()-_=+0123456789";
```

So, to parse the input file, remove all whitespace, punctuation and special characters. What's left are the words to be placed in the Dictionary, along with their frequencies.

Once all the words from an input file are placed in the Dictionary, along with their frequences, your program WordFrequency.cpp will print the Dictionary to the output file. You will find two very large text files here called Shakespeare.txt (containing the complete works of William Shakespeare) and Gutenberg.txt

(containing several English language texts provided by Project Gutenberg). You will also find the corresponding output files called Shakespeare-out.txt and Gutenberg-out.txt. Test WordFrequency.cpp on these files.

Also, as before, a test client called DictionaryClient.cpp is posted in Examples/pa8. This program is similar to the pa7 version, but it has different output, which you can find in DictionaryClient-out. You should still consider it a weak test of the Dictionary ADT, and as always, design your own tests.

Altogether this should be a straightforward assignment, especially if pa7 went well for you. Submit the following 7 files in the usual manner before the end of the grace period.

README.md	Written by you, a catalog of submitted files and any notes to the grader
Makefile	Provided, alter as you see fit
Dictionary.h	Provided, you may alter the "helper functions" sections, but nothing else
Dictionary.cpp	Written by you, the majority of work for this project
DictionaryTest.cpp	Written by you, your test client of the Dictionary ADT
Order.cpp	Written by you, a client for this project, unchanged from pa7
WordFrequency.cpp	Written by you, a client for this project

As usual, do not turn in any executable files, binaries, large text files, test input and output files, or anything not listed above. Good luck.