Introduction

You will compare the performance of simple binary search trees versus red-black trees by reading in a corpus of text, storing the words therein into a search tree, and then performing manipulations on the resulting tree. You will implement the following functions for both kinds of trees:

- **insert** which inserts a value into the tree (or updates a frequency count)
- **delete** which removes a value from the tree
- **find** which reports the frequency of a value in the tree

These operations should maintain *bst* ordering. You will also implement an interpreter which processes requests to manipulate the trees. Your interpreter should handle the following commands:

- `i W` insert word *W* into the tree
- `d W` delete word *W* from the tree
- `f W` report the frequency of word *W*
- `s` show the tree
- `r` report statistics

Legal tree values are simply words from the character set [a-z]. When reading a word, you should read a whitespace delimited token and then delete from it all non-letter characters. You should then convert all upper-case letters to lower case. For example, "Girl’s." would be rendered as "girls". Do not insert empty strings into the tree.

Inserting a word already in the tree would increase its frequency count by one. When deleting a word from the tree, you should reduce its frequency count by one. If the frequency count goes to zero, you should remove its corresponding node completely from the tree.

When showing the tree, display the nodes with a breadth-first (left-first) traversal. All nodes at a given level should be on the same line of output. The level number should precede the display of the nodes at that level. Display each node according to the following format:

- an optional equals sign (if the node is a leaf), followed by
- the node value, followed by
- an optional asterisk (if the node is colored red), followed by
- a parenthesized display of the parent’s value and color, followed by
- the frequency count, followed
- an X if the node is the root, an L if the node is a left child, and an R otherwise

Note that the parent of the root is itself. For the simple *bst*, all nodes are considered black. Here is an example of a *red-black* tree:

1: beta(beta)1X
2: =alpha*(beta)1L =gamma*(beta)1R

If the colors were reversed, the tree would display as:

1: beta*(beta*)1X
2: =alpha(beta*)1L =gamma(beta*)1R

The corpus to generate such a tree might look like:

```
beta alpha
  gamma
```

The words should be ordered within a tree in case-insensitive lexicographic ordering. Suppose the corpus was:

```
The quick brown fox
    jumped over the girl
        and her lazy, lazy dog.
```
found in a file named data. Suppose the file rbcommands holds the command:

```

```

Then a display of a bst generated by this corpus would look something like:

```
$ java -classpath classfiles main.Trees -1 data rbcommands
1: the(the)2X
2: quick(the)1L
3: brown(quick)1L
4: =and(brown)1L fox(brown)1R
5: =dog(fox)1L jumped(fox)1R
6: girl(jumped)1L over(jumped)1R
7: =her(girl)1R =lazy(over)2L
```

When inserted into a red-black tree, the resulting structure might look like:

```
$ java -classpath classfiles main.Trees -2 data rbcommands
1: jumped(jumped)1X
2: fox*(jumped)1L quick*(jumped)1R
3: brown(fox*)1L girl(fox*)1R over(quick*)1L =the(quick*)2R
4: =and*(brown)1L =dog*(brown)1R =her*(girl)1R =lazy*(over)2L
```

The statistics to be reported are:
- the number of nodes in the tree
- the minimum depth of the tree (the root is at depth 1)
- the maximum depth of the tree (the root is at depth 1)

The commands will be read from a free format file; individual tokens may be separated by arbitrary amounts of whitespace. For example, these three file contents are all legal and equivalent:

```
i spongebob
f Patrick
s
```

or

```
i spongebob f Patrick s
```

or

```
i spongebob f

Patrick

s
```

**Error handling**

You should ignore, but report via stderr, an attempt to delete a word that does not exist in the tree. Thus you ought to be able to randomly generate a large number commands and have your interpreter run without failing.

**Program invocation**

Your program will process a free-format corpus of text and a free-format file containing an arbitrary number of commands. The name of the corpus and the file of commands will be passed to the interpreter as a command line arguments. Switching between the two tree implementations is to be accomplished by providing the command line options -1 (simple bst) and -2 (red-black tree). Here is an example call to your interpreter:

```
java -classpath classfiles main.Trees -1 corpus commands
```

where corpus is a file of text and commands is the name of a file which contains a sequence of commands. In executing this call, you would read the words found in corpus, store them into a simple binary search tree, and the process the sequence of commands found in commands. The commands file may be empty.
Program output
All output should go to the console (stdout). When processing commands, only the result should be echoed to the console with a terminating newline. The command should not be echoed.

The insert and delete commands do not have a printable result and therefore should be processed silently.

Documentation
All code you hand in should be attributed to the author. Comment sparingly but well. Do explain the purpose of your program. Do not explain obvious code. If code is not obvious, consider rewriting the code rather than explaining what is going on through comments.

Project Organization
In your working directory, create the subdirectories src and classfiles. Place your source code files in the src directory and your class files in the classfiles directory. Your working directory will hold your makefile. The top-level rule in your makefile should look like:

```
top :
  javac -d classfiles -sourcepath src main/Trees.java
```

You should also have a test rule that tests your implementation on a relatively small set of commands:

```
test :
  @echo testing simple BST
  java -classpath classfiles main.Trees -1 mytestcorpus mytestcommands
  @echo testing red-black tree
  java -classpath classfiles main.Trees -2 mytestcorpus mytestcommands
```

Your commands file should feature all of the interpreter commands. You can name your test files anything you want. Finally, you should have a clean rule which deletes all your class files.

You must implement your interpreter in Java 7. Your program must compile cleanly and with no warnings or errors. You may develop on any system you wish but your program will be compiled and tested on a Linux system. Only the most foolish students would neglect to thoroughly test their implementation on a Linux system before submission.

Grading
Your implementation is worth 100 points. Serious points will be deducted for unreasonable amounts of duplicated code. Points will also be deducted for sloppy formatting, insufficient or overly verbose documentation, compiler warnings, run-time crashes, and other such transgressions. Moreover, your implementation needs to be relatively efficient. I will be testing your implementation on very large data sets; should your implementation take an unreasonably long time, I will regard that as a failure.

Note: you are required to use the standard read pattern for reading in your data:
read a value
while not at end of file
{
    process the value that was read
    read a value
}

Failure to do so will result in a 60 point deduction.

Handing in results
When you are ready to mail me the material, delete all intermediate files and executables. Then send me all the files in your directory by running the command

    submit cs201 lusth assign2

Again, your implementation may be developed on other hardware and operating systems, but it must also compile and run cleanly and correctly on a Linux system.