EE205 Fall 2018 Project 3: Yimacs
Search and Replace in a Large Text File

Wan Ju Kang and Hoyong Choi

November 2, 2018

1 Introduction

Starting with the most primitive line editors, all text editors have had a mechanism that allows the user to search for a particular string and replace all occurrences of it with some other string. This is usually accomplished by reading the text sequentially, finding the matching strings, and replacing them. This assignment explores a different approach motivated by a different situation. Suppose that the only strings you are allowed to replace are words as defined below:

- A *word* in a text file is a sequence of characters preceded an succeeded by whitespace.

This situation may take place when wanting to rename variables in a program. Another feature of this situation is that the file on which the search and replace operation is to be done could be large while the number of distinct words could be small. In this case, doing a large number of search and replace operations in a single batch would be a long list of updates to a large piece of software.

You are to implement a text editor that is capable of the said search and replace feature. We will call it Yimacs, short for Yi Yung + Emacs. In particular, you will be asked to **demonstrate your understanding of AVL trees and Red-Black trees in this search-and-replace application**.

2 Requirements

The text editor will read an input text file, perform a series of replacements, and output the result of these replacements in the following manner:

```
./Editor input_file file_after_replacements
```

Upon entering the line above, the user must be prompted to input a sequence of commands at the standard input (the console) specifying the word to be searched and the replacement word in the following form:

```
R search_word replacement_word
```

where R executes the replacement operation.

The program responds by writing to the console, not the output file, a list of the lines in which any replacement took place. The pre-replacement version and the post-replacement version must each be preceded by < and >, respectively. The angled bracket must be followed by a whitespace, and then the line. For example, upon entering the following line,

```
R all Who
```

the program must respond by,

```
< But I think that the most likely reason of all
> But I think that the most likely reason of Who
```

where the first line is the "before the replacement" version of the line, and the second line is the "after the replacement" version of the line. Each line needs to be printed only once, even if that line contains multiple replacements. For examples,

```
R Who What
```

```
< But the Grinch, Who lived just north of Who-ville, Did NOT!
> But the Grinch What lived just north of Who-ville, Did NOT!

< But I think that the most likely reason of Who Who
> But I think that the most likely reason of What What
```

Entering the character Q should terminate the program. Upon terminating the program, the final version (after all the replacements) of the text file should be written to an output file.

```
Q
```

Side Note. Instead of handling multiple commands from the console one by one, the program may take as an input file a sequence of pre-manufactured commands. That is,

```
./Editor input_file file_after_replacements < command_file > /dev/null
```

where command_file is a file containing a sequence of instructions such as:

```
R Who What
R all Who
```

You do NOT need to implement this single-line command for Project 3. Just implementing the console-entered commands is enough.

3 Example

Suppose that the input is

```
EveryWho Down in Who-ville Liked Christmas a lot...
But the Grinch, Who lived just north of Who-ville, Did NOT!
The Grinch hated Christmas! The whole Christmas season!
Now, please don't ask why. No one quite knows the reason.
It could be his head wasn't screwed on just right.
It could be, perhaps, that his shoes were too tight.
But I think that the most likely reason of all
May have been that his heart was two sizes too small.
```

An example sequence of commands and the program's correct responses to them are shown below, with commands in bold and responses in italics.

R all Who
< But I think that the most likely reason of all
> But I think that the most likely reason of Who
R Who What
< But the Grinch, Who lived just north of Who-ville, Did NOT!
> But the Grinch, What lived just north of Who-ville, Did NOT!

< But I think that the most likely reason of who
> But I think that the most likely reason of What

Notice several important things:

- A single word may undergo multiple replacements. (all, Who, What)
- Who-ville is considered a single word because it is surrounded by whitespace.
- There is no need to consider capitalization issues. "Who", "who", "wHO", *etc.* are considered the same word.

The output file, which must be created upon entering Q and terminating the program, should look like this.

```
EveryWho Down in Who-ville Liked Christmas a lot...
But the Grinch, What lived just north of Who-ville, Did NOT!
The Grinch hated Christmas! The whole Christmas season!
Now, please don't ask why. No one quite knows the reason.
It could be his head wasn't screwed on just right.
It could be, perhaps, that his shoes were too tight.
But I think that the most likely reason of What
May have been that his heart was two sizes too small.
```

The bold font is just for easier display. No need to implement it in the actual output text file.

4 Hints

Your implementation of this program must run in a *reasonable* amount of time, even on large, malicious inputs. This means that you will have to use some form of a balanced binary search tree. Some of our test inputs will exhibit worst-case behavior for dictionaries implemented as linked lists or as ordinary binary search trees.

Note that our AVL tree and RB tree classes are a dictionary, which has a (key, value) pair. In Yimacs, a key would be a word, and the corresponding value would be a linked list storing the line numbers where the word occurs, as shown in Figure 1. Here, each node is a word in the text file, and the linked list for each node represents the line numbers where that word is found. For example, the word "not" was found in lines 3, 12, and 24. The exact type of the tree in Figure 1 may be an AVL tree or an RB tree, depending on the implementation you are working on.

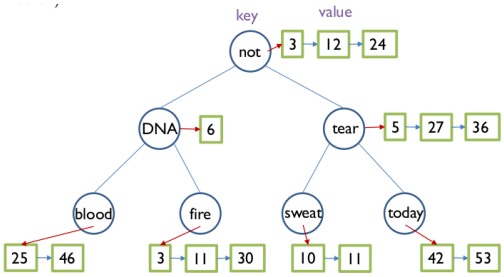

Figure 1: A balanced binary search tree, whose nodes point to a linked list

General flow of Yimacs. First, store the text as a list of strings, such that each line would correspond to a string. Then, the words can be stored in an AVL tree with a list of references to the lines in which they occur. The line numbers can be stored as an ordinary linked list without concerns about efficiency because

- during the input phase, line numbers occur in an increasing sequence
- when a word is replaced, the replacement will occur in some additional lines; the list of additional lines will be sorted in an ascending order and can be merged with the current list of the replacement in linear time

Recall that the number of distinct words will be much smaller than the length of the text, so search for a word in a tree will be much more efficient than search for it in the text itself.

At the input phase, read one line from the text, and fill the tree accordingly. Repeat this for all the lines of the text file.

5 Deliverables

- Implement the AVLTree class. You need to implement all the functions in avl_tree.h.
- Implement the RBTree class. You need to implement all the functions in rb_tree.h.
- Implement Yimacs using AVL tree-based dictionary.
- Implement Yimacs using RB tree-based dictionary, and compare it with the AVL tree-based dictionary in terms of their running time.

Your submission should include:

- avl_tree.cpp and avl_tree.h
- rb_tree.cpp and rb_tree.h
- your linked list implementation (unless you used standard library)
- yimacs_avl.cpp and yimacs_rb.cpp
- a report explaining your implementation and the comparison between AVL tree-based Yimacs and RB tree-based Yimacs.

Submit a file with the name EE205_project3_20XXYYYY.tar.gz, containing every source code of your implementation and the report containing the results and explanations about your code. Make sure that your .tar.gz file does not contain subdirectories, *i.e.*, no hierarchies.

6 Grading

The following components make up Project 3, with a total score of 10.

- 10% for compiling and executing without any error
- 35% for correctly implementing and explaining the AVL tree-based Yimacs
- 35% for correctly implementing and explaining the RB tree-based Yimacs
- 20% for the analysis of the comparison between the AVL tree-based Yimacs and the RB tree-based Yimacs

You are strongly advised to test your program on the servers for compiler standard, library dependencies, kernel version, and other potential issues.

This project is due on November 18 (Sun.), 2018 at 11:59 p.m. However, there will be a "soft" deadline on November 10 (Sat.), 2018 just for the AVL tree-based implementation of Yimacs. Consider this soft deadline as your personal timeline for doing Project 3. You will not incur any delay penalty even if you miss the soft deadline. Nevertheless, we strongly recommend that you submit the AVL tree part of the project by the soft deadline (on Nov. 10, there will be a submission folder on KLMS just for the AVL tree-based Yimacs.) Please do not spend too much time on AVL tree and then rush the RB tree implementation. Both implementations require an adequate amount of time. 25% penalty applies on each day after the hard deadline (Nov. 18), resulting in 0 points for all submissions after 11:59 p.m. of November 21, 2018.

A comment on the report. Please be sincere with writing the report. It is more than just a collection of program execution results. However, try not to cherry-pick good-looking results. Going back to coding just for "nicer" results will not produce a desirable report. When reporting on the performance between AVL tree and RB tree, your execution results may not show significant difference. *This is OK* as long as you show in the report what factors were taken into account. Some (imaginary) well-written examples with "not-so-good" results:

- Maybe, the text file was not long enough, so a longer text was given as an input, which eventually produced a non-negligible gap between AVL tree and RB tree.
- Maybe, the text file was long enough, but it had many different words, thereby breaking our assumption that $n_{distinct\_words} << length_{text}$. This probably means that both trees were too deep, to the extent that they lose their advantage over non-tree methods. So, I gave a different text file (which satisfies our assumption) and measured the depths of our BSTs, which turned out to be shallower.
- Maybe, the text file was long enough and contained very few distinct words, but the words were arranged in such a way that it removes some of the burden on the AVL tree with fewer rotation operations. So, I tried the same text file but randomized the order of the words in it. Then, I counted the number of rotation operations, and the results were more persuasive.

Whatever the result is, please try to justify your observations with scientifically convincing evidence, instead of going back to modifying the BST code and hoping for prettier results.