EE205 Fall 2018 Project 3: Yimacs

20170504 Juan Lee

November 17, 2018

1 Introduction

The Project 3 asks me to implement AVL Tree, Red-Black Tree, and text editors implemented based on those two trees. For the details of the project, please refer to the instruction PDF. I wrote this report with those contents:

- The implementation of AVL Tree
- The implementation of Yimacs-avl
- The implementation of RB Tree
- The implementation of Yimacs-rb
- Analysis of the comparison between two Yimacs

2 AVL Tree Implementation

AVL Tree was written on the skeleton code, given by TA. I was asked to fill six functions, inorder_print, preorder_print, search, rotate_left, balance, and remove. Among six, printing simply follows the order and codes them recursively, and rotate_left is written easily due to symmetry with rotate_right.

2.1 search

Search function finds node with given key and returns the node if exists, or, return external node. search was described to work recursively in Lecture Node, however, I wrote this iteratively since the skeleton code has a very difficult form to implement recursive function. Nevertheless, I followed the way to implement search, i.e., the result is exactly same with that on Lecture Note.

2.2 balance

Balance function balances AVL tree. It only works when balance_factor is 2 or -2, which is the difference of heights of two children. I handled four different cases, which combines rotate_left and rotate_right to solve balancing problem.

Note that balance functioning can break the balance itself, so I call the function recursively from the inserted node to the root.

2.3 remove

Since balance function is already written, remove was easy to implement. I basically divide into two cases(however, the code divides it into three cases but two of them are very similar). One for the node has two internal nodes, and the other for other case.

For the first case, I found a successor to be exchanged with the node. Then I exchanged the key and value, and removed the successor. The right child of successor should be maintained. For the other cases, it is enough to skip the node, which means connecting its child to its parent directly without itself.

Of course after both cases, balance should be called.

3 AVL-Based Yimacs

The new text editor, Yimacs, has two special commands. R for replace the word, which is defined as the sequence of characters surrounding by white spaces, without case sensitivity. Q for quit the program. Here, Yimacs is written based on AVL Tree.

3.1 Way to Replace

Since the word must handle all the white spaces, and is also replaced without case sensitivity, it was necessary to change the given replace_all function. Given function only handled space bar ' ', and it was case sensitive.

For solving case sensitivity, I added std::regex_constants::icase flag to std::regex. This is the way C++ supports. (1) Then iterate them for all the white spaces.

3.2 Way to get input

At the very beginning of the program, the program loads all the words and puts them into AVL Tree. This is because AVL Tree has relatively faster insert and remove speed, so it is efficient for text editor which required many operation for insertions and removals. Note that all the words are stored as lowercase for the case insensitivity.

As I mentioned, the word is stored into AVL Tree as a key of specific node. Each node, then, holds a linked list – double linked list in my case – which stores the line numbers where word exists in increasing order.

3.3 Way to handle user command

Then, program gets commands from users, as a form of "R from_word to_word" or "Q". Q halts the program and saves the result into output file, and R replaces from_word into to_word without case sensitivity.

I just converts from_word into lowercase, finds it from AVL Tree, and replace all the lines containing from_word.

4 RB Tree Implementation

Unlike to AVL Tree, no skeleton code was given for RB Tree. So, I changes the skeleton code of AVL tree slightly for RB Tree. RB Tree has similar operations, insert, search, and remove, but different re-organizing ways like restruct, recolor, double_red_remedy, or double_black_remedy.

4.1 insertion

When we put new node into RB Tree, the inserted node is always red except it is the very first node since red node does not break the depth property. However, when we insert red node as a child of red node, it violates red rule, so we have to solve it.

There are two ways to solve such a double red problem. They are restructuring and recoloring. The conditions and ways to code is explained in Lecture Note, and I strictly followed it.

Most difficult part of insertion was that recoloring may cause new double red problem. Since recoloring changes the parent node as red, if grandparent is red, this is also double red. I implemented recolor function as recursive function to solve it.

4.2 removal

Removal was very difficult due to so many conditions. Removal has many conditions because removal of black node violates the path rule, so we have to solve it.

For the remove operation, starts with removing it from normal binary search tree. If this does not violates the rule, we are done. This is case 0.

However, it breaks the rule whenever we are trying to remove black node and it does not have any red node to make it black instead of removed node. In this case, we have to solve double black problem. Double black problem can be solved by dividing it into three cases, and this is explained in lecture note.

5 RB-Based Yimacs

What I only changed for implementing RB-Based Yimacs from AVL-Based Yimacs is re-typing of AVL into RB. Since AVL Tree and RB Tree share same operations used in Yimacs, and same working result, this works very well if I would implement RB Tree well.

6 Analysis

We made two Yimacs based with AVL Tree and RB Tree. As I mentioned, those two Yimacs has very similar structure, but different base. In this section, I compare and contrast those two implementations.

6.1 Running Time Comparison

First of all, I compare the running time without any deep-thoughts. Both implementations are tested on my laptop computer, with generated lorem ipsum file with size of about 25MB. (2) I recorded the time elapsed with C++ chrono utilities. (3) Also, for control the keyboard input time, I made command file and pass it with stdin redirection. Thus, command file '.command' is "R lorem DSTest \n Q" and I run the command like "./yimacs_rb lorem output <.command". It works.

	RB Base	AVL Base
try 1	559.498s	558.533s
try 2	548.385s	546.419s
try 3	557.961S	557.083s
average	555.281s	554.012

Above table is the result of the running. I tried same command for same file three times, and calculate the average for the results. Both programs were running on same computing environment.

6.2 About the Result

Although we can see AVL-Based implementation is very slightly faster than RB-Based implementation, I cannot conclude which one is better since there is no meaningful difference between them in terms of running time.

6.3 Theoretical Analysis

AVL Tree and RB Tree have a significant difference at the way to reorganize the tree. AVL Tree uses balance, which changes the entire tree structure for balancing the heights of the children, so that it can maintain all the height differences less than 2. However, for balancing all the subtrees, it should be operated for all nodes from the leaf to the root. This may affect the running of insertion, or removal which requires re-balancing of the tree.

However, RB Tree does not strictly re-balance all the structures, but it just solves double-red or double-black locally. Thus, RB Tree may have more flexible structure than AVL Tree, which allows it to insert and remove faster. However, since RB Tree does not balance the children, it has relatively slow speed for the searching.

Not only for the time complexity, but also for the space complexity, they have slight difference. AVL Tree requires to maintain heights of all the nodes, so it should hold O(N) extra spaces for them. However, RB Tree requires only one bit to maintain its color, so it could keep more data than AVL Tree.

6.4 Regarding the Project

Regarding the project and test case that I made, there were many duplicated words in my file. (Since I copied and pasted same Lorem Ipsum file for making it 25MB) So, there were more search calls than insertion or removals. Also, insertion and removals use search inside its implementation. I guess this is why AVL-Based is slightly faster than RB-Based since there are much more search calls rather than insert or remove calls. (Also, I only tested it with single replace command)

7 Conclusion

From this project, I learned what AVL and RB are, what the properties are, and how to implement them. Although the result was different to what I expected, but I thought the test case was the reason of unexpected result.

I compiled and tested manually first, and I could not find any other errors or bugs that I did not fix. Also, I wrote some code for testing all the cases that I can expect, and test my programs automatically. Those automation code includes 26 tests for AVL Tree, 35 tests for RB Tree, and 16 tests for Yimacs implementations. All the examples in Lecture Note were included.

References

[1] https://stackoverflow.com/questions/24744819/does-stdregex-support-i-for-case-insensitivity, Stack Overflow

[2] https://loremipsum.io/generator/?n=999t=p

[3] C++ Chrono, https://en.cppreference.com/w/cpp/chrono, CppReference.com