Stephanie Galvan

Date: November 1, 2019

Lab #4 – Option B Anagrams

CS 2302 – Datat Structures MW 1:30PM – 2:50PM

Instructor: Diego Aguirre

TA: Gerardo Barraza

Fall 2019

Lab 4 – Option B Anagrams

Introduction

Lab 4 is a continuation of Lab 3 which dealt with binary-search self-balancing trees. In addition to the implementation of AVL and Red-Black trees, Lab 4 also introduces B-Trees as an additional data structure for comparison. Again, the trees will contain 'valid English words' which will be used to find the valid anagrams of a given word. The program will automatically run all trees and showcase differences in performance given a text file and a word. It will also allow the user to choose between testing all trees or testing how changing the number of keys affect the performance of B-trees.

Proposed Solution Design and Implementation

Since Lab4 is a continuation of Lab3, most of the original code remained the same; however, the main method was modified to allow the user to test and experiment with B-trees. The only method from Lab3 that will be used is counting anagrams since that will allow the comparison between trees. The only additional and new method for Lab4 is *degrees* which will be discussed below (Explanation for counting anagrams is also provided although this was already explained in lab 3).

Degrees:

This method takes a file, a maximum number of keys, and a word as parameters. The method would populate a B-tree given the file and maximum number of keys and then it will call the counting anagrams method given the word. The user will be able to see the running time of populating the tree and counting the anagrams of the word.

Counting Anagrams **(Refer to Lab 3 findings)**:

The operation to count anagrams was divided into two different functions (one which was defined as the helper function). First, the method would prompt the user to write the word they wanted to find the valid anagrams for. Once the input was entered, the helper function was called. The helper function was called _count_anagrams and it takes four parameters: English_words (represented as a previously chosen tree), the word the user input, an initial empty list, and a prefix with an empty default value. The function is recursive and it takes at least O(n^2) to compute all possible anagrams. At every call, the function will check if an anagram is part of English_words by calling the contains method; if it is true, it would append the anagram to the list. The function will return the length of the list which represents the number of anagrams of the given word. A possible downfall for the implementation is the use of the list to count anagram instead of using a int variable; a list was implement initially in order to print the list for debugging; however, a better design would use an int instead of a list to save space memory.

Run-time (Refer to Lab 3 findings)

Counting Anagrams:

T(n) = T(n^2) + n + 6 (in which n^2 refers to finding the anagrams of word n, and the extra work refers to n (length of word) and 6 (3 is related to the base case, and the other 3 refer to the else statement

Experimental Results

In order to compare the performance of the trees, files of more than 1k lines were used to determine the running time for populating each of the different types of trees and for counting anagrams in each tree. The word *"spear"* (or any of its combinations) was used for testing since it has at least ten anagrams.

The testing was divided in three parts. The first part will compare populating the trees, next searching for anagrams, and lastly testing how changing the number of keys affect the performance of B-trees. When comparing all trees, the default B-trees had the default 5 as maximum number of keys. Each result is separated by figures 1 A-B, 2 A-B, and 3 A-B.

Figure 1.A: Table for Populating Trees

AVL TREE	RED-BLACK TREE	B-TREE
6.1995 s	0.0875 s	0.0254 s
4.3549 s	0.05103 s	0.01099 s
5.3469 s	0.05399 s	0.01399 s
4.0889 s	0.042029 s	0.0130021 s
3.73603 s	0.03999876 s	0.01099801 s
3.7089886 s	0.036998987 s	0.010991811 s
3.71702 s	0.036035 s	0.0109953 s

Figure 1.B: Graph for Populating Trees

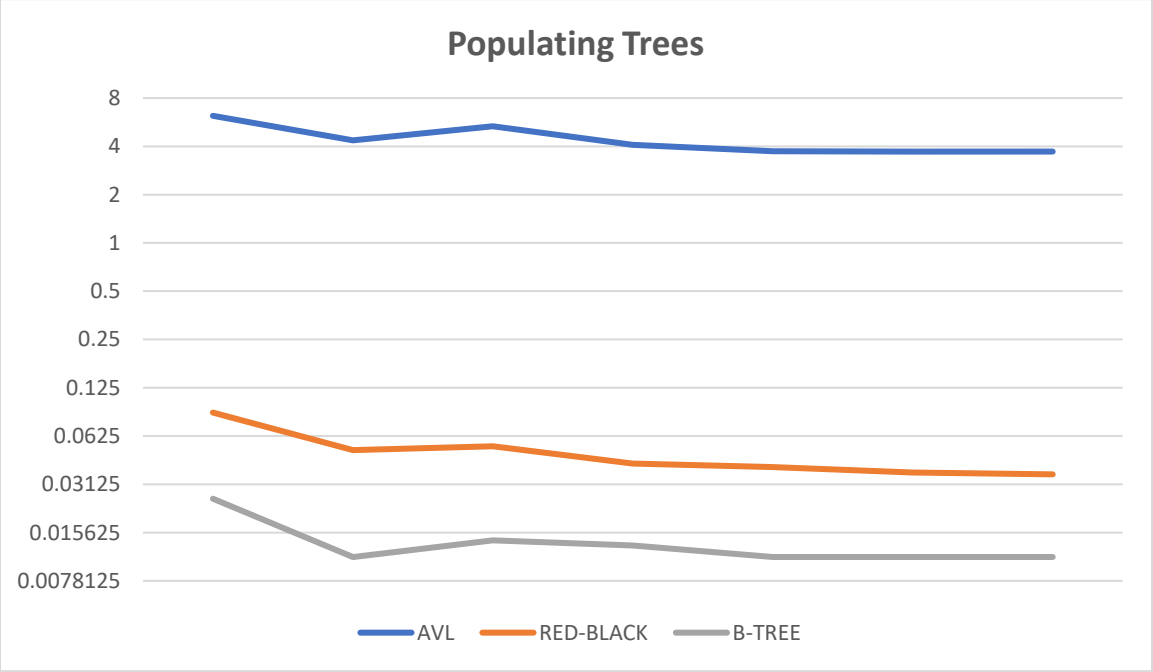


Figure 2.A: Table for Counting Anagrams

AVL TREE	RED-BLACK TREE	B-TREE
0.0010027 s	0.0049989 s	0.001000165 s
0.00096154 s	0.00400257 s	0.000988245 s
0.0010139 s	0.00399947 s	0.000979651 s
0.001000642 s	0.00397062 s	0.0010001659 s
0.001002311 s	0.0040020942 s	0.0010011196 s
0.00100708 s	0.0030009746 s	0.0010006427 s
0.00098831 s	0.00300209961 s	0.001001358 s

Figure 2.B: Graph for Counting Anagrams

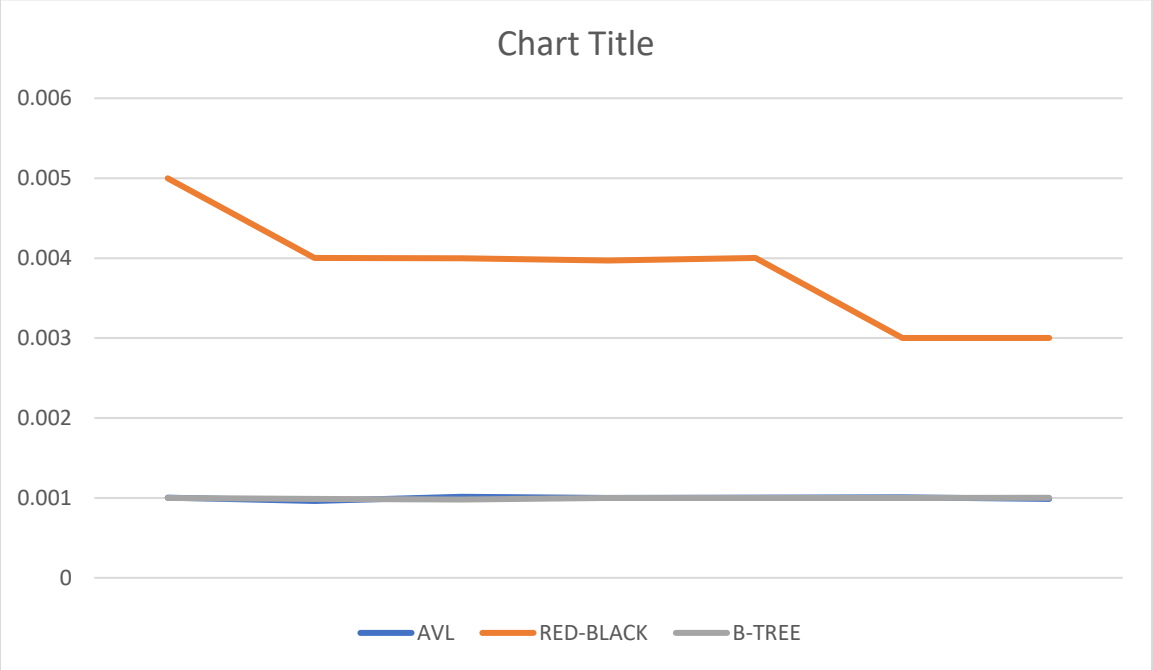
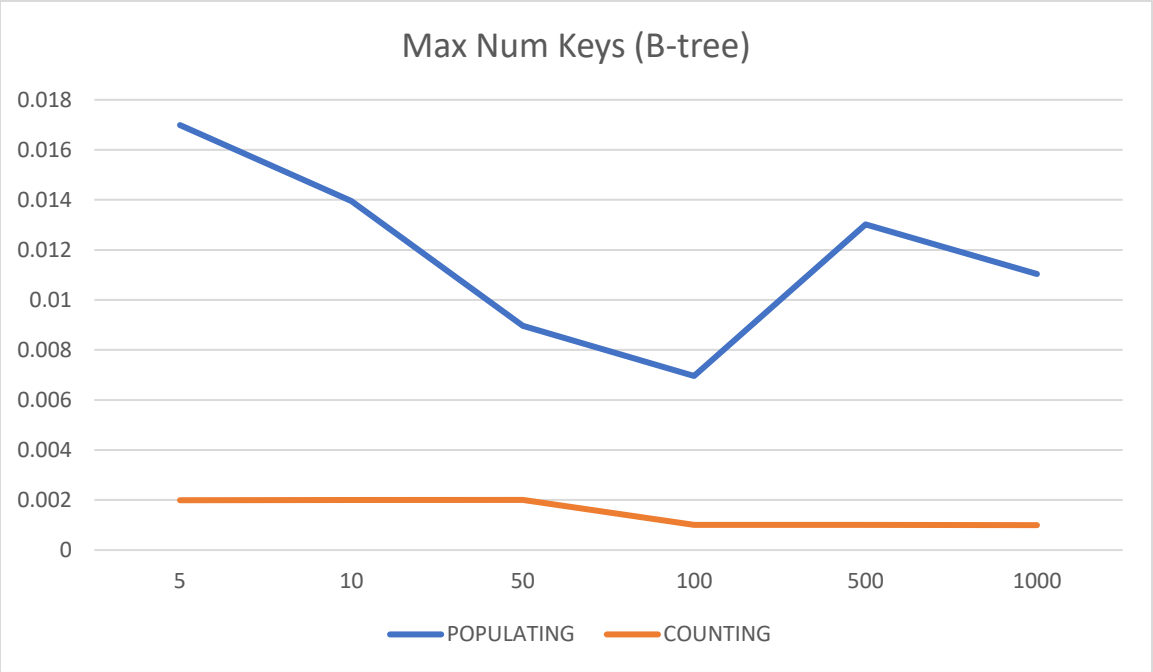


Figure 3.A: Table for Degrees (B-tree)

MAX NUM KEYS	POPULATING B-TREE	COUNTING ANAGRAMS
5	0.01699 s	0.00199 s
10	0.01396 s	0.002002 s
50	0.00896 s	0.00200057 s
100	0.00696 s	0.00100111 s
500	0.01302 s	0.001001358 s
1000	0.011032 s	0.000997 s

Figure 3.B: Graph for Degrees (B-tree)



Conclusion

Overall, B-tree is faster than both AVL and Red-Black for populating a tree and for searching for the number of anagrams of a given word. As it was demonstrated on the previous lab, AVL trees are faster at searching than Red-Black trees while Red-Black trees were faster at populating its data than AVL trees. However, as shown in Figure 2.B, AVL and B-trees have an incredibly similar running time for searching, but their slight differences are more noticeable by comparing the data collected from table on Figure 2.A.

Next, B-trees performance was also analyzed by the maximum number of keys allowed. As the maximum number of keys increased, B-trees generally became faster at both populating and searching anagrams. However, counting anagrams remained relatively more stable than populating the B-tree. As demonstrated on figure 3.B, there is a steep decline in time when populating the tree, but then the time spikes again when 500 maximum numbers of keys are used. However, this unusual trend might account to an edge occurrence (this could have been affected due to the search of a specific word and due to its location, it could change the time of the algorithm); thus, one can conclude that generally B-trees perform faster when the maximum numbers of keys increase. As number of keys increase, the number of levels decrease which might explain the reduction of time as more keys were added.

Ultimately, B-trees are the great combination of both fast populating the data or searching the data. However, depending the task, an AVL or a Red-Black tree might be a more suitable choice if a problem requires only one task (populating data or searching data).

Appendix

Source code found at https://github.com/stephypy/CS2302_Lab4

Code originally written in py charm.

Honesty Certification:

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.

INITIALS: **SG** FULL NAME: **Stephanie Galvan** DATE: **October 23, 2019**