PROJECT 3: BENCHMARK

By: Riadiani Marcelita and Bendik Svalastog

CS 146: Anna Shaverdian

For our third project in our CS 146 class, we had to determine if Sir Francis Bacon wrote William Shakespeare's plays. We did so by taking in a piece of work by the two authors, counting the words in each document, then correlating them to find out if they were roughly similar to each other. We did so by containing the words in three kinds of data structures: BST, AVL tree, and hash tables. After finishing the project and experimenting with it, we came to several conclusions about our project.

For benchmarking this particular project, we focused on three main components: the project's overall runtime, the length of document being analyzed, and the memory usage of each data structure.

The first method of benchmarking we analyzed is overall runtime. When we ran Hamlet and The New Atlantis with our three data structures, we received different runtime results. For Hamlet, for instance, the runtime that we received are:

1) BST: 0.11309341 seconds
2) AVL: 7.739418316 seconds
3) Hashtable: 0.043845764 seconds

We did the same runtime analysis for Correlator for two documents, Hamlet and The New Atlantis, and received somewhat similar results:

1) BST: 4,736,701 seconds.
2) AVL: 3,743,884 seconds.
3) Hash table: 3,286,360 seconds.

From our analysis, we conclude that hash table is a more efficient data structure than BST and AVL tree. We think this is because hash table has a runtime of O(1) (constant time) for creation and implementation, whereas BST, in its average case, has a runtime of O(log(N)), and a worst case of O(N). AVL tree also has a constant runtime of O(log(N)), be it average or worst case. O(log(N)) is the runtime for single insertions, deletions, or searches in AVL trees. However, the time it takes to insert values into and build the tree is O(Nlog(N)). This explains why hash tables

have a faster runtime than BST and AVL tree. This benchmarking method is the runtime of the overall program (full program), and it does not include time for user input because in our code, we start analyzing the time when the user supposedly already finishes inputting the information needed to run the program, and we end the time when the entire program is operated. Thus, our runtime results do not include user input, and is therefore a more accurate representation of how fast the entire program operates.

The second method we used to measure our program is the length of document that we analyze. We first analyzed Hamlet and The New Atlantis, two relatively long documents with many words. The runtime that we received were mentioned above. We also ran the program using two simple and short documents, named Tester and Tester 2. They contained no longer than 8 words each, and the runtime we received when we ran the two short documents in WordCount were:

1) BST: 1,177,563 seconds.
2) AVL: 1,243,883 seconds.
3) Hash table: 1,939,051 seconds.

Based on our analysis, if we ran the program/code using smaller sized documents, AVL and BST would give us a faster runtime. We reckon this is because with shorter documents, the BST and AVL trees do not have to insert and balance many words, thus taking lesser time to do the entire operation than they do with larger/longer documents. However, we believe that the reason our hash table is faster than our BST and AVL trees for larger documents is because for hash tables, the data is already stored in an array, and it is easy to get through the indexes of the array and getting each element. Whereas for BST and AVL trees, we have to put them in nodes, compare the data with each node to decide which subtree it should go to, put them into an array in traversed order, then correlate it. That might be why they take a longer time than the hash table.

The third benchmarking method we measured is memory usage of each data structure. With Hamlet, the memory usage of each data structure is:

1) BST: 6,134,328 bytes.
2) AVL: 6,134,440 bytes.
3) Hash table: 7,497,656 bytes.

From our analysis, hash table takes up more memory than BST and AVL tree. This is because BST and AVL tree only takes up new memory if we increment the count of the existing words in

the tree or when we make a new node in the tree. Whereas for hash table, we have to create an array with a relative size, then if it is filled, we have to "rehash" or resize it. This caused our array to have a relatively larger size, and since not all of the array's indexes have to be filled, there might be array slots/indexes that are empty. These empty array indexes take up extra space, causing our hash table implementation to take up higher memory usage than our BST and AVL tree.

Benchmarking with these three methods give relative results because they all depend on many factors, such as the complexity of our code, our hash table's initial size, our rehash method/formula, the style of probing or collision handling we did, and many others. If our benchmarking methods were to be tested using another student's code, for example, it would probably give different results than ours, so these results are not definite or precise. However, these benchmarking methods are efficient and logical to use because they are methods that programmers have to consider and manage whenever they make programs. Programmers have to think about how effective their program is in relation to their runtime, length of data analyzed, and memory usage. So it is reasonable for us to use these three methods to benchmark our program, since they make up quite important aspects of program-making/designing.

There are also several minor sources of error in our program, mainly our implementation of hash code. We realized that when our data is printed in WordCount, it is not printed in alphabetical order. This partially is because we did not find a way to sort the elements of our array in our hash table implementation. However, this source of error is trivial in our opinion, since our results are the same. Our results are still reliable, and it does not affect the information our program provides significantly. Additionally, when we ran Correlator using hash table, we also received a frequency that is only off by very few digits with our frequency using BST and AVL, 4.564351156875463E-4 for the frequency by BST and AVL, and 8.847547372260941E-5 for the frequency by our hash table. Since this difference is quite minor, we do not think that this source of error affects the efficiency of our program too much, since it still indicates that the two documents being correlated have a relatively low frequency and thus are similar in style, letting us come to the conclusion that Sir Francis Bacon did, in fact, wrote Shakespeare's plays for him.