1) Duoc Nguyen, Patrick Leung

2) Approximately less than one week.

3) We expected the Hash table to perform the fastest because its access time is O(1).

4)

Total time duration to insert all words from Hamlet in BST: 71 Millis

Total time duration to insert all words from Hamlet in AVLTree: 489 Millis

Total time duration to insert all words from Hamlet in HashTable: 33 Millis

Total time duration to get all word counts from Hamlet in BST: 2 Millis

Total time duration to get all word counts from Hamlet in AVLTree: 2 Millis

Total time duration to get all word counts from Hamlet in HashTable: 1 Millis

Total time duration to search a random phrase 1000000 times in BST: 148 Millis

Total time duration to search a random phrase 1000000 times in AVLTree: 76 Millis

Total time duration to search a random phrase 1000000 times in HashTable: 23 Millis

Total time duration to search all the existing words in the HashTable 2 Millis

Total time duration to search all the existing words in the BST 8 Millis

Total time duration to search all the existing words in the AVLTree 11 Millis

Above are the time duration for different back end structure to run it’s function. We made test cases in order to really understand which back end structure works the fastest. Base on our data above, HashTable data structure beat BST and AVLTree in insertions and searches tremendously. Thus, our prediction was correct when we predicted Hash Table would perform the fastest.

5)

In general, HashTable data structure is the best structure out of the 3(AVLTree, BST, HashTable) for the following reasons: coding HashTable takes less time and less lines of code than BST and AVLTree. Although HashTable can be a little complicated when debugging HashTable’s performance of constant time outweighs the debugging complication. Our team ran into a rehashing problem when we implement HashTable and it took us awhile to trace what was the problem. How we debugged this problem we made our text file relatively small inorder to clearly see the difference when rehashing. Debugging trees data structure is a little more simplier. In term of memory efficiency, table seems to waste more memory than tree structure. The way we deal with collisions was separate chaining so we added a pointer to our DataCount class. This increases the memory size to hold a pointer. In addition, to keep the access run time O(1), we need to make sure that our load factor is low(0.5 or less), we rehash our table whenever size <= tableSize/2. Thus, we waste half of the table keeping it empty. Tree data structure is better at memory usage because it only creates a new node and adds a pointer for every element inserted into the tree. However, this does not implies that tree data structure is better than hash table because disk access and run time average input of hash table is better than tree. In term of disk access pattern….. When comparing the average run time of input, hash table has a O(1) time complexity for each insertion. Whereas, for tree it is O(log n ) time complexity because it has to traverse the height of the tree. Thus, comparing the advantages and disadvantages of tree and table, table is the “better” data structure.

6)

Total time duration to complete correlator with hash table for hamlet.txt and the-new-atlantis.txt : 100 Millis

Total time duration to complete correlator with BST for hamlet.txt and the-new-atlantis.txt : 125 Millis

Total time duration to complete correlator with AVLTree for hamlet.txt and the-new-atlantis.txt : 684 Millis

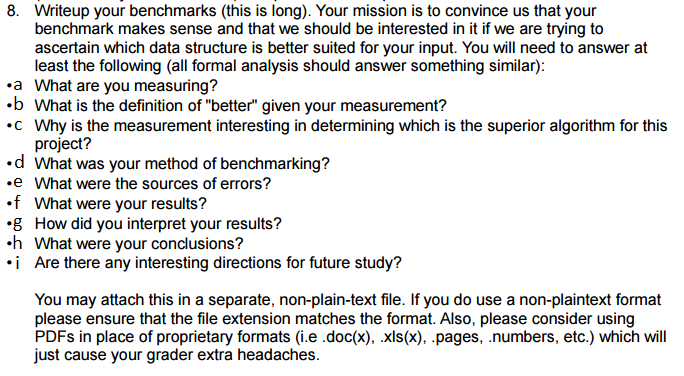
Referring to our BenchMark for more data analysis:

HashTable seems to give the fastest time duration and BST can keep up with the performance. However, AVLTree significantly performs poorly on the Correlator. The reason being is AVLTree has to call balance on each insertion and if tree is unbalanced, rotations takes more time than BST and HashTable.

7)

It is finally time to settle the age old debate of whether or not Sir Francis Bacon wrote Shakespeare's plays. From our data, it is clear to see that our two beloved authors are big fans of their particles and prepositions. In particular, the words “the” and “to” are used so many times, 5% of Shakespeare’s Hamlet consists of those two words alone. Before we draw any conclusions, it is important to note that our data, supported by the two texts, *“The Tragedy of Hamlet, Prince of Denmark”* by Shakespeare and *“New Atlantis”* by Francis Bacon are different in that one is a play while the other is a novel which was never finished. The first thing that was interesting to note in our data was that the words “I”, “you”, “my”, and “your” in Shakespeare’s work showed up much more often than they did in Bacon’s novel. This is partly due to the fact that Hamlet is a play and purely based on dialogue, but can also be an indication of a “signature” which contrasts Bacon’s novel in which he uses words like “we” and “us” much more than Shakespeare.

Taking a look at other works by the two authors, in particular, *“Romeo and Juliet”* by Shakespeare and *“The Essays Or Counsels, Civil And Moral, Of Francis Ld. Verulam Viscount St. Albans”* by Bacon, we can see similarities in the author’s “signature”, or writing style. Again, we see that certain words mentioned earlier come much more frequently in one author’s work compared to the other. In conclusion, the work of Shakespeare shares some similarities to that of Sir Francis Bacon, but the differences between the “signatures” of the two authors are very visible. However, comparing a play to a novel is like comparing a spoon to a fork. It is safe to say that Francis Bacon did not write Shakespeare’s plays, but there are no sure answers. As far as computer science goes, the works from these two authors are just more toys that we coders can play around with.



a) With these benchmarks below, we are putting the AVL tree and Hash table against each other to see which performs the fastest with the binary search tree as the reference.

b) In terms of run time, “better” means whichever algorithm can perform its task in the faster time. We consider the performance at small and large. We also consider the data structure’s insertion and search performance at small and large.

c) The measurements for run time in algorithms is important because they show how well an algorithm can perform under a certain circumstance and teach us which algorithm is best given the context in which the algorithm is intended to be used.

d) For the benchmark method, we took the time duration for each data structure to complete the WordCount class which uses BST or AVLTree or HashTable data structure has its backend structure. Then WordCount can either analyze the text files’ word frequencies or number of unique words. We analyze 13 books in total with “King James Bible” the largest text file. The WordCount class’s runtime depends on the data structure that it uses. One of the noticeable difference is it’s sorting algorithm. If data structure is a tree, then it uses a insertion sort algorithm(O(n^2)). Whereas if it uses a table data strucutre, then it uses quicksort algorithm(O(nlogn)). Another method is timing how long it takes to complete the Correlator class. This class’s run time depends on the search method of the data structure that it uses. If a tree data structure is used, then it search time complexity is O(logn) whereas table’s search time complexity is O(1). We also made some test for the data structure’s insertion method, getCounts method, search method in small and large n. We time each function using 13 different books. For the search method, small n is the number of words in the file. Large n number of word is 1,000,000 because it was a reasonable large number to analyze. We had all 3 data structure search (fail) a random phrase such as “asdfasdf” for 1,000,000. The reason why we wanted to search(fail) a random phrase is we wanted the tree data structure to traverse through every element as it’s worst case. Whereas, the worst case of search method for a table would be O(1) because we made the load factor <= 0.5 to avoid long linked lists.

e) One possible source of error could come from the speed of our computers. Personally, I (Patrick) am not one that is able to focus for too long on one thing. More often than not, I have many other applications running in the background of my old laptop along with eclipse. I am not sure how much that can affect run time, but I’m sure it plays some part in the performance of a program. Another source of error may come from our implementations of the DataCounter interface. Looking at the not so surprisingly terrible insertion results for the AVL Tree, I (Patrick) keep thinking that I may have messed up my implementation, but each time I look at the code, it seems fine.

f) From the data collected, it was clear (and expected) that the AVL Tree would lack in performance when inserting items compared to the BST and Hash Table implementations. Many of our tests included insertions, but the few tests that accessed from the data structure showed that the AVL Tree was able to have good performance, sometimes even beating the BST. The Hash Table on the other hand, was a very reliable implementation. While its insertion time was good in comparison to the BST, its access time was miles ahead. Had we used a text file that was far larger than the novels and plays that we tested, the Hash Table would have by far been the winner of this performance competition.

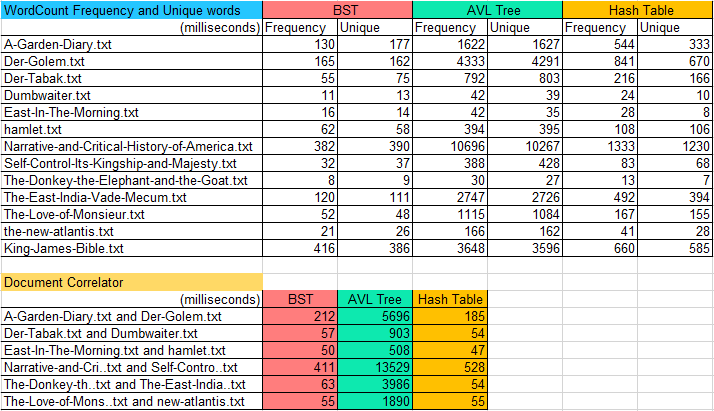
g) We looked the times duration of each method that we used for benchmarking and compare their values. If the difference is noticeable, in multiple methods, then we can conclude that one data structure is better than the other. For example, BST’s time duration for frequency and unique words for the book “King James Bible” is (416, 386) whereas HashTable’s time duration is (660, 586) both in milliseconds. At this point, we cannot conclude that BST is better than HashTable. If we look at time duration of Correlator, insertion alone, and search method in large and small, we would consider HashTable a better data structure. For example, searching 1,000,000 times for a random phrase in “the new atlantis” , BST(122) AVLTree(105) and HashTable(5), HashTable data structure does a significantly better job than both trees structure. Thus, if one data structure show significant advantage in multiple benchmarking methods, we would interpret its performance as “better” than the others.

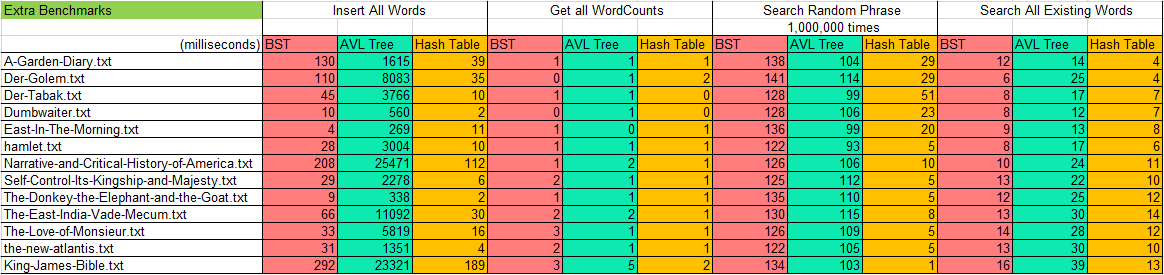
h) Our conclusion from this project is that Hash Tables have insanely fast insertion and access time, but only beat out the other two data structures when there is a large number of items to be stored. The BST had beaten the Hash Table in the word frequency tests but when larger texts, and with them, larger amounts of items to be stored were in the equation, the Hash Table was relatively able to maintain its speed. Take note that the first test for word frequency showed the BST beating the Hash Table by a fair amount of time but by the last test containing the largest file size, the Hash Table’s insertion time went up steadily while the BST’s time jumped. The other tests we had showed the Hash Table performing consistently well while the BST and AVL Tree performances were as expected. Our ending conclusions were exactly what we thought would happen from the start of the project. The BST would perform as it does, the AVL Tree has terrible insertion time but decent access time due to its balanced structure, and the Hash Table does consistently well with its insertions while its access time is the opposite of an AVL Tree’s insertion time.

i) The use of a word counter has plenty potential to be useful in the future. One possible idea is to use it to find these writers signatures within papers turned in by students to catch cheating. This could also be a helpful resource put on the internet, where there are probably many others, to get an overall word count as well as word statistics throughout a given amount of text. In the context of analyzing smaller documents a BST may be better simply because of its faster run time with shorter amounts of text.

9) The enjoyable parts of this assignment were how we collectively worked together as a team to implement the back end data structures. Of course, taking time and really think about how the data structure works is fun. Also, it was fun figuring out the run time complexity of each function to get the best run time. However, debugging is not always fun and it may be frustrated. Debugging a rehash function of HashTable took 4-5 hours. The way we finally solve it was cutting down the test size and trace to see if hash table rehash correctly. The bug was we did not copy over the counts of each word when we rehash. This project would be a lot more enjoyable if there was specific requirement on run time of each class. For example, Correlator class must be O(nlogn) for tree structure or O(n) for HashTable.

**Data Analysis From Benchmarks**





**Benchmarks**

**BST data structure measuring frequency time on:**

A-Garden-Diary.txt 130 millis

Der-Golem.txt 165 millis

Der-Tabak.txt 55 millis

Dumbwaiter.txt 11 millis

East-In-The-Morning.txt 16 millis

hamlet.txt 62 millis

Narrative-and-Critical-History-of-America.txt 382 millis

Self-Control-Its-Kingship-and-Majesty.txt 32 millis

The-Donkey-the-Elephant-and-the-Goat.txt 8 millis

The-East-India-Vade-Mecum.txt 120 millis

The-Love-of-Monsieur.txt 52 millis

the-new-atlantis.txt 21 millis

King-James-Bible.txt 416 millis

**BST data structure measuring number of unique word time on:**

A-Garden-Diary.txt 177 millis

Der-Golem.txt 162 millis

Der-Tabak.txt 75 millis

Dumbwaiter.txt 13 millis

East-In-The-Morning.txt 14 millis

hamlet.txt 58 millis

Narrative-and-Critical-History-of-America.txt 390 millis

Self-Control-Its-Kingship-and-Majesty.txt 37 millis

The-Donkey-the-Elephant-and-the-Goat.txt 9 millis

The-East-India-Vade-Mecum.txt 111 millis

The-Love-of-Monsieur.txt 48 millis

the-new-atlantis.txt 26 millis

King-James-Bible.txt 386 millis

**AVLTree data structure measuring frequency time on:**

A-Garden-Diary.txt 1622 millis

Der-Golem.txt 4333 millis

Der-Tabak.txt 792 millis

Dumbwaiter.txt 42 millis

East-In-The-Morning.txt 42 millis

hamlet.txt 394 millis

Narrative-and-Critical-History-of-America.txt 10696 millis

Self-Control-Its-Kingship-and-Majesty.txt 388 millis

The-Donkey-the-Elephant-and-the-Goat.txt 30 millis

The-East-India-Vade-Mecum.txt 2747 millis

The-Love-of-Monsieur.txt 1115 millis

the-new-atlantis.txt 166 millis

King-James-Bible.txt 3648 millis

**AVLTree data structure measuring number of unique word time on:**

A-Garden-Diary.txt 1627 millis

Der-Golem.txt 4291 millis

Der-Tabak.txt 803 millis

Dumbwaiter.txt 39 millis

East-In-The-Morning.txt 35 millis

hamlet.txt 395 millis

Narrative-and-Critical-History-of-America.txt 10267 millis

Self-Control-Its-Kingship-and-Majesty.txt 428 millis

The-Donkey-the-Elephant-and-the-Goat.txt 27 millis

The-East-India-Vade-Mecum.txt 2726 millis

The-Love-of-Monsieur.txt 1084 millis

the-new-atlantis.txt 162 millis

King-James-Bible.txt 3596 millis

**HashTable data structure measuring frequency time on:**

A-Garden-Diary.txt 544 millis

Der-Golem.txt 841 millis

Der-Tabak.txt 216 millis

Dumbwaiter.txt 24 millis

East-In-The-Morning.txt 28 millis

hamlet.txt 108 millis

Narrative-and-Critical-History-of-America.txt 1333 millis

Self-Control-Its-Kingship-and-Majesty.txt 83 millis

The-Donkey-the-Elephant-and-the-Goat.txt 13 millis

The-East-India-Vade-Mecum.txt 492 millis

The-Love-of-Monsieur.txt 167 millis

the-new-atlantis.txt 41 millis

King-James-Bible.txt 660 millis

**HashTable data structure measuring number of unique word time on:**

A-Garden-Diary.txt 333 millis

Der-Golem.txt 670 millis

Der-Tabak.txt 166 millis

Dumbwaiter.txt 10 millis

East-In-The-Morning.txt 8 millis

hamlet.txt 106 millis

Narrative-and-Critical-History-of-America.txt 1230 millis

Self-Control-Its-Kingship-and-Majesty.txt 68 millis

The-Donkey-the-Elephant-and-the-Goat.txt 7 millis

The-East-India-Vade-Mecum.txt 394 millis

The-Love-of-Monsieur.txt 155 millis

the-new-atlantis.txt 28 millis

King-James-Bible.txt 585 millis

***THE ABOVE ARE DATA FROM WORDCOUNT.JAVA***

**AVLTree correlator:**

A-Garden-Diary.txt and Der-Golem.txt : 5696 Millis

Der-Tabak.txt and Dumbwaiter.txt : 903 Millis

East-In-The-Morning.txt and hamlet.txt : 508 Millis

Narrative-and-Critical-History-of-America.txt and Self-Control-Its-Kingship-and-Majesty.txt : 13529 Millis

The-Donkey-the-Elephant-and-the-Goat.txt and The-East-India-Vade-Mecum.txt : 3986 Millis

The-Love-of-Monsieur.txt and the-new-atlantis.txt : 1890 Millis

**BST correlator:**

A-Garden-Diary.txt and Der-Golem.txt : 212 Millis

Der-Tabak.txt and Dumbwaiter.txt : 57 Millis

East-In-The-Morning.txt and hamlet.txt : 50 Millis

Narrative-and-Critical-History-of-America.txt and Self-Control-Its-Kingship-and-Majesty.txt : 411 Millis

The-Donkey-the-Elephant-and-the-Goat.txt and The-East-India-Vade-Mecum.txt : 63 Millis

The-Love-of-Monsieur.txt and the-new-atlantis.txt : 55 Millis

**HashTable correlator:**

A-Garden-Diary.txt and Der-Golem.txt : 185 Millis

Der-Tabak.txt and Dumbwaiter.txt : 54 Millis

East-In-The-Morning.txt and hamlet.txt : 47 Millis

Narrative-and-Critical-History-of-America.txt and Self-Control-Its-Kingship-and-Majesty.txt : 528 Millis

The-Donkey-the-Elephant-and-the-Goat.txt and The-East-India-Vade-Mecum.txt : 54 Millis

The-Love-of-Monsieur.txt and the-new-atlantis.txt : 55 Millis

**USING: A-Garden-Diary.txt**

insert all words from A-Garden-Diary.txt in BST: 130 Millis

insert all words from A-Garden-Diary.txt in AVLTree: 1615 Millis

insert all words in HashTable: 39 Millis

get all word counts in BST: 1 Millis

get all word counts in AVLTree: 1 Millis

get all word counts in HashTable: 1 Millis

search a random phrase 1000000 times in BST: 138 Millis

search a random phrase 1000000 times in AVLTree: 104 Millis

search a random phrase 1000000 times in HashTable: 29 Millis

search all the existing words in the HashTable 4 Millis

search all the existing words in the BST 12 Millis

search all the existing words in the AVLTree 14 Millis

**USING: Der-Golem.txt**

insert all words from Der-Golem.txt in BST: 110 Millis

insert all words from Der-Golem.txt in AVLTree: 8083 Millis

insert all words in HashTable: 35 Millis

get all word counts in BST: 0 Millis

get all word counts in AVLTree: 1 Millis

get all word counts in HashTable: 2 Millis

search a random phrase 1000000 times in BST: 141 Millis

search a random phrase 1000000 times in AVLTree: 114 Millis

search a random phrase 1000000 times in HashTable: 29 Millis

search all the existing words in the HashTable 4 Millis

search all the existing words in the BST 6 Millis

search all the existing words in the AVLTree 25 Millis

**USING: Der-Tabak.txt**

insert all words from Der-Tabak.txt in BST: 45 Millis

insert all words from Der-Tabak.txt in AVLTree: 3766 Millis

insert all words in HashTable: 10 Millis

get all word counts in BST: 1 Millis

get all word counts in AVLTree: 1 Millis

get all word counts in HashTable: 0 Millis

search a random phrase 1000000 times in BST: 128 Millis

search a random phrase 1000000 times in AVLTree: 99 Millis

search a random phrase 1000000 times in HashTable: 51 Millis

search all the existing words in the HashTable 7 Millis

search all the existing words in the BST 8 Millis

search all the existing words in the AVLTree 17 Millis

**USING: Dumbwaiter.txt**

insert all words from Dumbwaiter.txt in BST: 10 Millis

insert all words from Dumbwaiter.txt in AVLTree: 560 Millis

insert all words in HashTable: 2 Millis

get all word counts in BST: 0 Millis

get all word counts in AVLTree: 1 Millis

get all word counts in HashTable: 0 Millis

search a random phrase 1000000 times in BST: 128 Millis

search a random phrase 1000000 times in AVLTree: 106 Millis

search a random phrase 1000000 times in HashTable: 23 Millis

search all the existing words in the HashTable 7 Millis

search all the existing words in the BST 8 Millis

search all the existing words in the AVLTree 12 Millis

**USING: East-In-The-Morning.txt**

insert all words from East-In-The-Morning.txt in BST: 4 Millis

insert all words from East-In-The-Morning.txt in AVLTree: 469 Millis

insert all words in HashTable: 11 Millis

get all word counts in BST: 1 Millis

get all word counts in AVLTree: 0 Millis

get all word counts in HashTable: 1 Millis

search a random phrase 1000000 times in BST: 136 Millis

search a random phrase 1000000 times in AVLTree: 99 Millis

search a random phrase 1000000 times in HashTable: 20 Millis

search all the existing words in the HashTable 8 Millis

search all the existing words in the BST 9 Millis

search all the existing words in the AVLTree 13 Millis

**USING: hamlet.txt**

insert all words from hamlet.txt in BST: 28 Millis

insert all words from hamlet.txt in AVLTree: 3004 Millis

insert all words in HashTable: 10 Millis

get all word counts in BST: 1 Millis

get all word counts in AVLTree: 1 Millis

get all word counts in HashTable: 1 Millis

search a random phrase 1000000 times in BST: 122 Millis

search a random phrase 1000000 times in AVLTree: 93 Millis

search a random phrase 1000000 times in HashTable: 5 Millis

search all the existing words in the HashTable 6 Millis

search all the existing words in the BST 8 Millis

search all the existing words in the AVLTree 17 Millis

**USING: Narrative-and-Critical-History-of-America.txt**

insert all words from Narrative-and-Critical-History-of-America.txt in BST: 208 Millis

insert all words from Narrative-and-Critical-History-of-America.txt in AVLTree: 25471 Millis

insert all words in HashTable: 112 Millis

get all word counts in BST: 1 Millis

get all word counts in AVLTree: 2 Millis

get all word counts in HashTable: 1 Millis

search a random phrase 1000000 times in BST: 126 Millis

search a random phrase 1000000 times in AVLTree: 106 Millis

search a random phrase 1000000 times in HashTable: 10 Millis

search all the existing words in the HashTable 11 Millis

search all the existing words in the BST 10 Millis

search all the existing words in the AVLTree 24 Millis

**USING: Self-Control-Its-Kingship-and-Majesty.txt**

insert all words from Self-Control-Its-Kingship-and-Majesty.txt in BST: 29 Millis

insert all words from Self-Control-Its-Kingship-and-Majesty.txt in AVLTree: 2278 Millis

insert all words in HashTable: 6 Millis

get all word counts in BST: 2 Millis

get all word counts in AVLTree: 1 Millis

get all word counts in HashTable: 1 Millis

search a random phrase 1000000 times in BST: 125 Millis

search a random phrase 1000000 times in AVLTree: 112 Millis

search a random phrase 1000000 times in HashTable: 5 Millis

search all the existing words in the HashTable 10 Millis

search all the existing words in the BST 13 Millis

search all the existing words in the AVLTree 22 Millis

**USING: The-Donkey-the-Elephant-and-the-Goat.txt**

insert all words from The-Donkey-the-Elephant-and-the-Goat.txt in BST: 9 Millis

insert all words from The-Donkey-the-Elephant-and-the-Goat.txt in AVLTree: 338 Millis

insert all words in HashTable: 2 Millis

get all word counts in BST: 1 Millis

get all word counts in AVLTree: 1 Millis

get all word counts in HashTable: 1 Millis

search a random phrase 1000000 times in BST: 135 Millis

search a random phrase 1000000 times in AVLTree: 110 Millis

search a random phrase 1000000 times in HashTable: 5 Millis

search all the existing words in the HashTable 12 Millis

search all the existing words in the BST 12 Millis

search all the existing words in the AVLTree 25 Millis

**USING: The-East-India-Vade-Mecum.txt**

insert all words from The-East-India-Vade-Mecum.txt in BST: 66 Millis

insert all words from The-East-India-Vade-Mecum.txt in AVLTree: 11092 Millis

insert all words in HashTable: 30 Millis

get all word counts in BST: 2 Millis

get all word counts in AVLTree: 2 Millis

get all word counts in HashTable: 1 Millis

search a random phrase 1000000 times in BST: 130 Millis

search a random phrase 1000000 times in AVLTree: 115 Millis

search a random phrase 1000000 times in HashTable: 8 Millis

search all the existing words in the HashTable 14 Millis

search all the existing words in the BST 13 Millis

search all the existing words in the AVLTree 30 Millis

**USING: The-Love-of-Monsieur.txt**

insert all words from The-Love-of-Monsieur.txt in BST: 33 Millis

insert all words from The-Love-of-Monsieur.txt in AVLTree: 5819 Millis

insert all words in HashTable: 16 Millis

get all word counts in BST: 3 Millis

get all word counts in AVLTree: 1 Millis

get all word counts in HashTable: 1 Millis

search a random phrase 1000000 times in BST: 126 Millis

search a random phrase 1000000 times in AVLTree: 109 Millis

search a random phrase 1000000 times in HashTable: 5 Millis

search all the existing words in the HashTable 12 Millis

search all the existing words in the BST 14 Millis

search all the existing words in the AVLTree 28 Millis

**USING: the-new-atlantis.txt**

insert all words from the-new-atlantis.txt in BST: 31 Millis

insert all words from the-new-atlantis.txt in AVLTree: 1351 Millis

insert all words in HashTable: 4 Millis

get all word counts in BST: 2 Millis

get all word counts in AVLTree: 1 Millis

get all word counts in HashTable: 1 Millis

search a random phrase 1000000 times in BST: 122 Millis

search a random phrase 1000000 times in AVLTree: 105 Millis

search a random phrase 1000000 times in HashTable: 5 Millis

search all the existing words in the HashTable 10 Millis

search all the existing words in the BST 13 Millis

search all the existing words in the AVLTree 30 Millis

**USING: King-James-Bible.txt**

insert all words from King-James-Bible.txt in BST: 292 Millis

insert all words from King-James-Bible.txt in AVLTree: 23321 Millis

insert all words in HashTable: 189 Millis

get all word counts in BST: 3 Millis

get all word counts in AVLTree: 5 Millis

get all word counts in HashTable: 2 Millis

search a random phrase 1000000 times in BST: 134 Millis

search a random phrase 1000000 times in AVLTree: 103 Millis

search a random phrase 1000000 times in HashTable: 1 Millis

search all the existing words in the HashTable 13 Millis

search all the existing words in the BST 16 Millis

search all the existing words in the AVLTree 39 Millis