**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #4**

Due: October 4, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

Lab 4: Binary Search Trees (BST) and B-Trees focuses on developing a program that will store word embeddings in a BST or B-Tree based on the user’s preference. After storing the data, the chosen data structure, the similarities between a pair of words is compared. These pairs of words are located in a file which contain two words per line, separated by commas. In order to identify the similarities between the words, their vectors must be compared. Thus, the program will search the BST or B-Tree to find the two words and retrieve the emb attribute from the WordEmbedding object. In order to compare the words, cosine distance is used. In order to find the cosine distance, the dot product of the first embedding and second embedding is divided by the magnitudes of the first embedding and second embedding. The efficiency of constructing the B-Tree and BST is then compared by analyzing the runtimes for building the tree and computing the similarities. Since a major theme of this lab was to learn how data structures are used in the “real-world,” the importance of word embedding is further highlighted below. By learning how to implement BST and B-Trees to find the word embeddings, a new tool is utilized.

**Word Embeddings**

Word Embeddings, in general, are words represented through a vector of number. These vectors allow words with a similar context or meaning to be correlated. Thus, the word “apple” and “banana” would have more similar vectors than that of “apple” and “piano”. The vector for each word is crucial to capturing the words meaning. One way to represent these words is through one-hot encoding. One hot coding using a table of 1s and 0s where a 1 marks the position of the word. However, this approach can be very lengthy since a majority of zeros are used and the table will grow as more words are added. Thus, custom word embeddings are used to limit the amount of zeros by using custom dimensions that best fit the word. So, for “apple,” “banana,” “red,” “yellow,” and “fruit” may be further used to demonstrate the relationship and differences between the words. Additionally, if “tomato” was another word, then the position of “apple” and “banana” would be closer to each other on the table then “banana” and “tomato”. Thus, measuring the “Euclidean distance” is useful in these situations (Lynn, n.d.). Word embeddings are further defined by the context. In theory, a word will usually be embedded among similar words that usually reappear with the word. Some common algorithms used to find the contexts include Distributed Sematic Models and Neural Network Models (Lynn, n.d.). Word Embeddings, is a fairly recent way for representing words from a vector of numbers allowing modern day technologies to respond to input by extracting corresponding meaning.

**Proposed Solution Design and Implementation**

**Overall Design**

This lab consists of four files: BSTClass.py, BTreeClass.py, WordEmbedding.py, and NicholeMaldonado\_Lab4.py. BSTClass.py, BTreeClass.py, and WordEmbedding.py contain the class as well as any specific methods that are related to the class. For example, the tree searches and insertions are located with their corresponding class. The main part of the program lies in the NicholeMaldonado\_Lab4.py. This portion of program signals the actual creation of the tree and interprets the users input. It is essentially the backbone of the lab. I decided to use four files to keep all of my classes separated and organized. Initially, I had two files: one with the class definitions and all the other methods in another file. However, I realized that the code lacked a logical layout so I used separate files to solve this problem.

**Part 1**

Main Design

Based on the described output, the main functionality builds a BST or B-Tree, based on the user’s preference, and searches for pairs provided by a second file. The similarities of each word pair are then displayed. During the program, the runtime for the tree construction and tree search is also exhibited. Since two files are needed, I decided to first have the program prompt the user to enter the file path for each file. After receiving the file paths, each path is first checked to ensure that it is a .txt file. I thus, used the same method from lab 1 in order to ensure that the program would not attempt to open some other miscellaneous file, like a .jpg for example. The user is then given the option to use a BST or B-Tree to store the word embeddings and retrieve the designated word embeddings. In order to keep ordered, I created two separate functions bst\_analysis and btree\_analysis, which would both perform the same tasks but for the different data structures. I enclosed each method call around a try block, in the event that an error occurred while trying to access any of the two files. For the btree\_analysis and bst\_analysis, both file paths were accepted as parameters. Both files would then call their corresponding tree “setup” method in which the tree would be constructed. After the tree was created, it would be returned to the method. The word pairs from the second file would then be evaluated by calling the corresponding “find\_similarities” function. This function would find the similarities and return the runtime. This runtime would then be printed, and the program would terminate. In order to make the comparison between the B-Tree and BST fair, I made the overall design structure as cohesive as possible. Both data structures have the same operations performed on them and similar functions reporting the same type of information.

**Part 2**

BST Construction

The main construction method for the BST is file\_to\_bst. This method takes the file path of the glove.6B.50d.txt file as a parameter. The file is then opened and each line of the file is read one at a time. As each line is read, it is processed and turned into a word\_embedding object. The line is first striped of any newlines. A check also ensures that the word only contains letters. I thought that this was an important design decision since I only wanted to compare words rather than symbols or punctuations. For example, word embeddings that included “(” or “ ’s” were not included. I wanted to purely focus on words since “apple” and “banana” would be more closely related then “apple” or “(”. Thus, I am able to avoid comparing punctation with words and focus purely on word comparison. After the line is checked, it is split by spaces into a list. The word of the WordEmbedding object is then assigned to the value at index one and the emb attribute is assigned to the rest of the list. With the WordEmbedding object created, it is then inserted into the BST.

The insert function for the BST is fairly straightforward. It takes the current BST and the word embedding to be added. If the BST is None, then a new BST object with the word embedding as the data is returned. Otherwise the tree will traverse according to the value of the word embedding’s word, which is compared with the current BST node’s word. The left or right pointer of the current node is then reassigned to the node returned from the next recursive call. In the end, this recursive method will return the BST with a new node added. It is also important to note that I added these nodes alphabetically to the BST. For example, if the root had “apple” as the head then “banana” would belong to the right of the node and “ape” would go to the left.

After all the lines were read and all the word embedding objects were added to the tree, I returned the tree and the performance time. The performance time was calculated by subtracting the time the file was initially iterated through from the end of the iterations.

As for the efficiency of the function, m word\_embeddings will be added to the BST. Thus, the insertion method will be called m times. The insertion method will traverse through a specific path of the tree and then insert the node where it belongs at a leaf. As the height of the tree continues to grow, more nodes will be compared. If the tree was perfectly balanced, it would have logn levels, where n is the number of nodes in the tree, making the search function faster. Initially, I was going to sort a list and create a perfectly balanced tree. After implementing this, I realized that the tree construction increased by 7 seconds whereas the search would decrement by 0.03 seconds. I realized that this tradeoff was very minimal and decided to keep the current algorithm.

B-Tree Construction

The main construction method for the B-Tree is file\_to\_btree. This method takes the file path of the glove.6B.50d.txt file and the max\_items, determined by the user, as a parameter. The file is then opened and each line of the file is read one at a time. Similar to the BST implementation, as each line is read, it is processed and turned into a word\_embedding object. The line is striped of any newlines and a check ensures that the word only contains letters. The word\_embedding object and the B-Tree are then passed as a parameter to the BSTInsert method.

The insertion method for the B-Tree is the same code provided in class, however it has adjustments. The method still starts of by determining if the root of the B-Tree needs to be split or if the data can proceed to being inserted internally. The insertInternal method takes the B-Tree and the word\_embedding object as a parameter. Since data can only be added to leafs, the current B-Tree node is a leaf, then the method InsertLeaf is called. Unlike the example provided in class, the word\_embedding object cannot simply be appended to the list. Therefore I implemented a BinaryInsertion method and LinearInsertion method, to place the word\_embedding object in it’s correct alphabetical order. Binary Insertion takes a list, the word\_embedding object, and the start and end index as parameters. The function operates similar to binary search, updating the start and end index based on where the word\_embedding should be located in the list. Once the start is greater than or equal to end, the word\_embedding object is inserted into the list. The Linear insertion only takes the list of the B-Tree’s data and the word\_embedding as a parameter. The algorithm will through all the elements in the list until the word\_embedding correct alphabetical position is found. The word\_embedding object will then be inserted in the list. The reason why my program uses both LinearInsertion and BinaryInsertion to insert word\_embeddings in the current node’s list is because I found that the linear insertion was actually faster for lists of approximately 115. Larger lists are therefore handled by the BinaryInsertion. Apart from inserting a word\_word\_embedding object into a leaf, the InsertInternal method also uses the FindChild method which takes the current node and the object as parameters and returns the index of the next child to be searched. The InsertInternal method also splits any full node that it encounters by calling the Split method which takes the current full node. A left and right child for this node are then created and returned as well as the node’s new value.

After all the word\_embedding objects are inserted, the B-Tree and the performance time are returned. The performance time was also calculated by subtracting the time when the file was first iterated through from the end of all the iterations.

As for the efficiency of the function, m word\_embeddings will be added to the B-Tree. Thus, the insertion method will be called m times. Since all keys are added to a leaf, the height of the tree will be traversed. Once the leaf is reached, the BinaryInsertion or LinearInsertion will be used. For the BinaryInsertion the list of data is halved for each recursive call resulting iwhich yield O(logd), where d is the amount of data in that leaf. The LinearInsertion, however, in the worst case will iterate through the entire list resulting in d elements seen, yielding O(d). It may be wondered why the BinaryInsertion was not chosen for all data insertions since O(logd) is better than O(d). When I was constructing the tree, I found that the running times using LinearInsertion were actually faster for 115 elements than using the BinaryInsertion method. For example, using the BinarySearch with a max\_data of 5, the running time was approximately 23 seconds. When LinearInsertion was used, the running time dramatically decreased to 13 seconds. Thus, if the current node’s data has 115 elements or less, Linear Insertion is used. If the node’s data has more than 115 elements then BinaryInsertion is used.

**Part 3**

Compute Similarities

In order to compare the similarities, the compute\_similarity function is called. This function takes the two vectors of the two word\_embedding objects as parameters. The dot product, obtained by using np.dot() is then divided by the magnitude and returned. The magnitude is obtained by multiplying the result from the np.linalg.norm() function from the vectors.

BST Search

The main method for reading a file containing the pairs of words and finding the similarity of the words is the find\_similarities\_bst function. This function takes the BST and the file path for the word pairs as a parameter. The file is opened and as each line is iterated through the current pair of word\_embeddings are found in the BST. The vectors of the embeddings are then compared using the function described above. Additionally, as each line is read, the words and their similarities are added to a list. When all the pairs have been evaluated, the similarities are then displayed to the screen. The reason why I added the pairs and similarities to a list rather than display them was because I wanted the runtime for the function to be as exact as possible. As demonstrated in Lab 3, printing information to the screen can dramatically increase the runtime. The performance time calculated from the start of the file iteration to the end was then returned.

In order to find the word\_embedding in the BST, the method FindWordBST was used. This method took the BST and the word as a parameter and returned the emb attribute of the word\_embedding found. If the word was not found, then None was returned. The FindWordBST is also makes recursive calls to the left or right node, based on the words relation to the other word\_embedding’s word. For example, if the word was “horse” and the current node had a word of “apple”, then a recursive call would be made to the node’s right node.

As for efficiency of the function, the search function for the BST will in the worse case iterate through the entire height of the tree until finding the word. This results in a runtime complexity of O(h). If the BST is perfectly balanced, the runtime complexity could ideally be O(logn). In the worst case, the height of the tree will be n, resulting in O(n). Since m pairs are read from the file, two calls two the search method will be made resulting in 2h. However, this constant is negligible. Thus, in total, the runtime complexity will be O(h \* m) where m is the number of pairs read from the file.

B-Tree Search

The main method for reading a file containing the pairs of words and finding the similarity of the words is the find\_similarities\_btree function. This function takes the B-Tree and the file path for the word pairs as a parameter. The file is opened and as each line is iterated through the current pair of word\_embeddings are found in the B-Tree. The vectors of the embeddings are then compared using the function described above. Additionally, as each line is read, the words and their similarities are added to a list. When all the pairs have been evaluated, the similarities are then displayed to the screen. The performance time is also calculated from the start of file read until the end, since all pairs will have been evaluated at that time. The similarities are then displayed to the screen and the performance time is returned.

In order to find the word\_embedding in the BTree, the method FindWordBTree was used. This method took the B-Tree and the word as a parameter and returned the emb attribute of the word\_embedding found. If the word was not found, then None was returned. In order to identify if the word was in the current B-Tree node’s data, Binary Search was performed. The BinarySearch function took the data list, the word, the start index, and end index as parameters. The middle element was evaluated, if the middle element matched the word, then middle was returned. Otherwise, the left or right sublist would be evaluated accordingly. If the start index ever became larger or equal to the end index, then two options could occur. Start was returned if the key was less than or equal to the word at start. Otherwise, start plus one was returned since the word belongs to the right of the value. After receiving this index, the FindWordBTree method would then return the emb attribute of word\_embedding object if a match was found, or it would make a recursive call to the child at the index returned. Initially, I used a linear search to identify if the word was in the current Node’s data. However, I realized that since the data was sorted, I could easily perform binary search in O(logn) rather than O(n).

As for runtime complexity, the search function for the B-Tree will, in the worse case iterate through the entire height of the tree until finding the word. However, with each level accessed, Binary Search is performed on the current data list to identify if the key exists resulting in O(logd) where d is the amount of data in the node visited. Additionally, 2 calls are made for every m pairs in the file. Thus, the BST may lag in terms of efficiency since it not only has to go through a specific path of the tree, but it must also traverse through the current node’s data to find which child should be visited next. This hypothesis will be supported or disproved in the tests.

**Part 4**

Running Time Display

In order to compare the running time of the construction and similarity computation using the BST and B-Tree, I displayed each performance time after the operation occurred. These running times were usually returned to a method to be printed and displayed along with the other stats.

BST Stats

In order to display the stats of the BST, I created a HeightBST and NumberOfNodesBST function to compute the attributes. The HeightBST returns -1 if the BST is empty. Otherwise a recursive call is made to the left and right node and the max height of the two calls is returned. One is added to the height since the current node exists and is then returned. As for the runtime complexity of this operation, the whole tree will need to be traversed resulting in O(n) where n is the number of nodes in the tree. However, since this is a utility function and not used for construction, its performance time was not calculated. For the NumberOfNodesBST, 1 is added for every non-None node seen. Thus, the whose tree has to be reversed resulting in T(n) = O(n) where n is the number of nodes in the tree. Similar to the HeightBST, since this was a utility function, its performance time was not added to the construction runtime.

B-Tree Stats

In order to display the stats of the B-Tree, I created a HeightBTree and NumberOfNodesBTree function to compute the attributes. The HeightBTree returns 0 if the current node is released. Otherwise, a recursive call is then made to each subsequent rightmost child. For each level seen, 1 is added. As for the runtime complexity of this operation, the h levels will be visited where h is the height of the tree resulting in O(h). However, since this is a utility function and not used for construction, its performance time was not calculated. For the NumberOfNodesBST, 0 is returned if the current B-Tree node is a root. Otherwise a recursive call is made to each of the current node’s children and a sum of the nodes seen is incremented and returned. Thus, the whose tree has to be reversed resulting in T(n) = O(n) in all cases where n is the number of nodes in the tree. Similar to HeightBTree, since this was a utility function, its performance time was not added to the construction runtime.

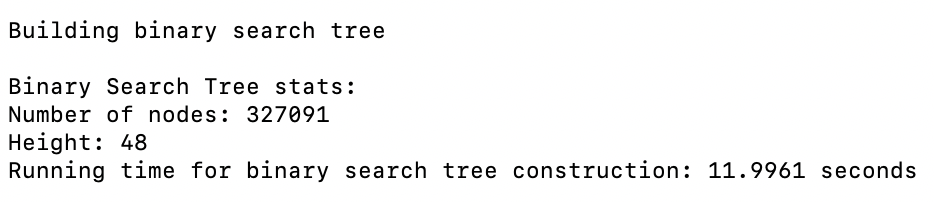
**Experimental Results**

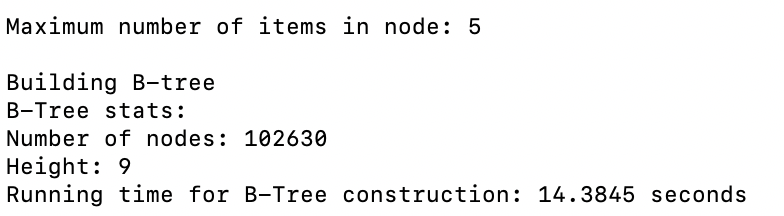
NOTE: The tests below summarize results obtained from the automated tests.

**Part 1 - Tree Construction:**

For the Tree Construction, a one test run for the BST and seven B-Trees with max\_data of 3, 15, 49, 99, 499, and 4999. In order to identify how the parameters of the B-Tree would affect the construction time, I choose a wide variety of max\_data values. I also wanted to demonstrate for the B-Tree how effective the LinearInsertion is for smaller max\_data values and how effective the Binary Insertion is for values like 4,999. I also chose 3 since it was as small to a BST that could be attained with a BST.

Sample Output:





Case 1 (“construction”):

BST Number of Nodes: 327, 091, Height = 48

B-Tree, max\_data = 3, Number of Nodes: 187,240, Height = 14

B-Tree, max\_data = 15, Number of Nodes: 32,066, Height = 5

B-Tree, max\_data = 49, Number of Nodes: 9,666, Height = 3

B-Tree, max\_data = 99, Number of Nodes: 4,751, Height = 2

B-Tree, max\_data = 499, Number of Nodes: 951, Height = 2

B-Tree, max\_data = 999, Number of Nodes: 461, Height = 1

B-Tree, max\_data = 4999, Number of Nodes: 95, Height = 1





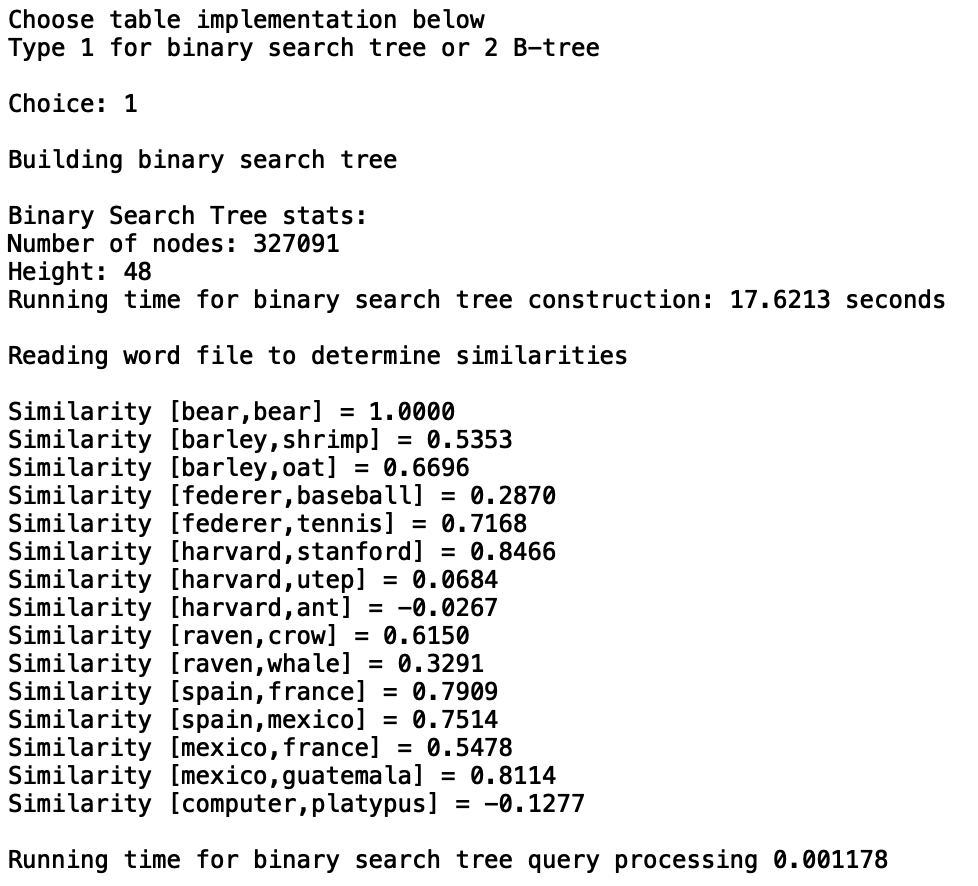
From the bar chart, it is evident that the B-Tree had the fastest and slowest runtime for constructing the tree. However, it is also crucial to note that the B-Tree’s with the lowest runtimes were approximately a second shorter than the BST’s average runtime. The reason the B-Tree has a higher runtime with a max\_value of 3 rather than 999, is because the effects of the Binary Search for the FindChild method is not noticeable. By constantly splitting a large value, like 999, in half less data is seen. As for a max\_value of 3, in the best case 1 value will be seen and in the worst case two elements will be seen. Furthermore, the reason why the BST is able to construct the tree so quickly is because for each word\_embedding, a maximum of h + 1 nodes need to compared, where h is the height of tree. Since the height of the BST is 48, the worst-case scenario for inserting the at element at level 48, is 49 comparisons. Additionally, for the BST, it has significantly lower runtimes then B-Trees with a smaller max\_data since it does not have to worry about finding the next child to visit or performing a split. The BST simply needs to look at the data in the current node and move to the left or right subtree accordingly. For the B-Tree, h + 1 nodes need to be compared and the data of the visited node needs to be iterated through to identify where the new word\_embedding can be placed or find the next child. In some instances, like when the max\_data is 4,999, the height of the tree is only 1 resulting in a maximum of only 2 nodes visited. Although the B-Tree does have a shorter height then BST, the runtime can still be high due to these checks, as seen with the max\_data of 3 and max\_data of 99. Additionally, since the B-Trees require split operations even more comparisons are necessary, and nodes constantly need to be adjusted if they are full. As can be seen for the B-Tree with max data of 99, a total of 4,751 nodes which were the result of splitting full nodes. As for the B-Tree with max\_data of 4,999, its runtime is less since it can hold more data before having to split into a separate node. Lastly the increase and then decrease in performance time at max\_data of 99 is probably due to the algorithms switch to use binary insertion instead of linear insertion. The linear insertion was able to insert an element into the leaf’s data quickly for smaller sizes of data. However, as the data grew the inefficiency of using linear insertion is noted and thus binary insertion is used. The true effects of constantly splitting the data is not noticeable until larger sizes of max\_data are employed.

**Similarities – Provided Output:**

For the Similarity test, two tests ere run. These tests focused finding the similarities of the pairs provided in the lab. I thought that this case was very important to identify if the compute\_similarity function was producing the correct results for both the BST and B-Tree.

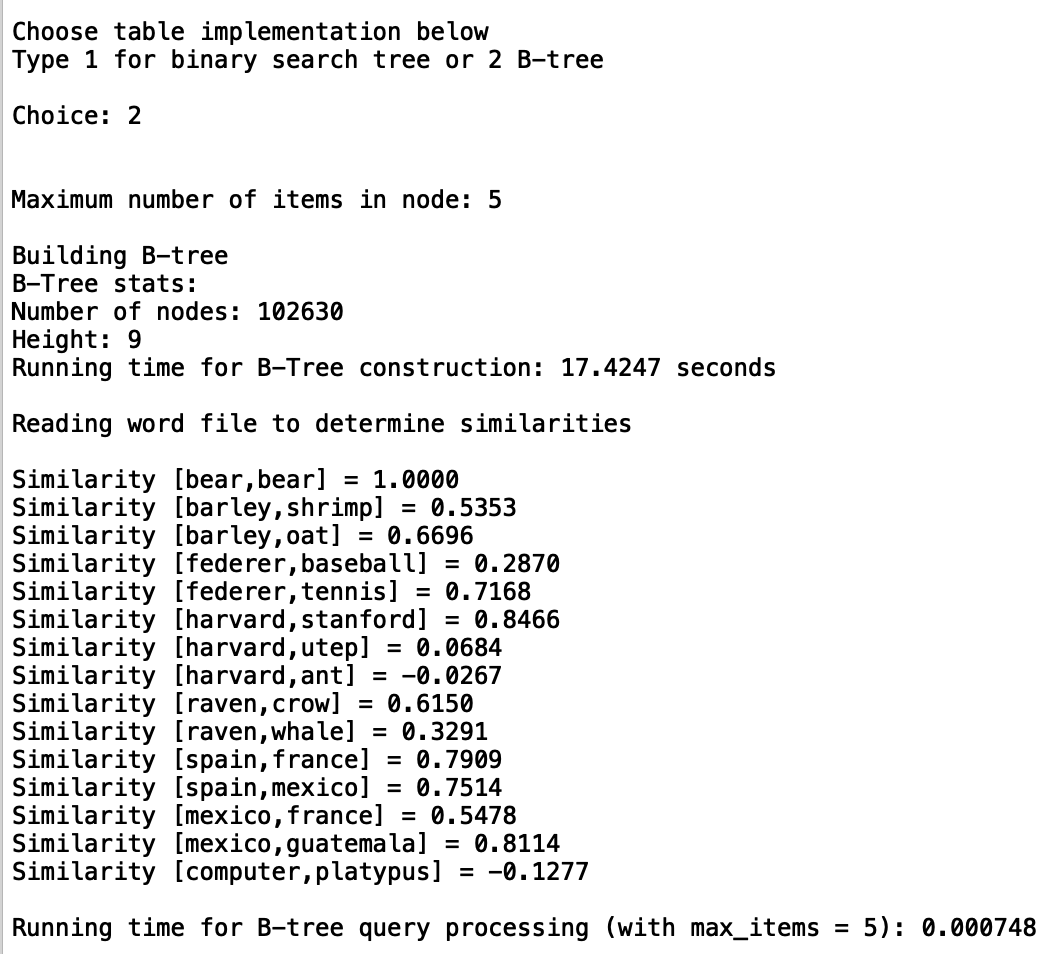
Case 1 (“BST”):

BST Number of Nodes: 327, 091, Height = 48



Case 1 (“B-Tree”):

B-Tree, max\_data = 5, Number of Nodes: 102,630, Height = 9



From the outputs, I was able to identify that the correct results for the similarities between the words was being computed. In doing so, I was able to focus on the running times without having to worry if the correct similarities are being computed. This test case also demonstrated that the running times for the B-Trees and BST are almost negligible since the amount of pairs is so small.

**Similarities – Small and Average Number of Pairs:**

For the Similarity test, five tests were run. This test focused gathering the runtime complexities for files with 100, 500, 1000, 2000, and 5000 pairs. Each test had three different trials with the .txt files corresponding to the trial number. Each .txt file had randomly selected pairs. I thought that this test was very important to identify how the BST and B-Trees would respond to average size and smaller data. Having a smaller range of pairs is also crucial to evaluate how the efficiency of each data structure will change as more pairs are searched for and compared.

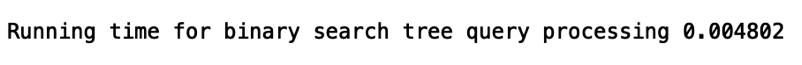
Sample Output:

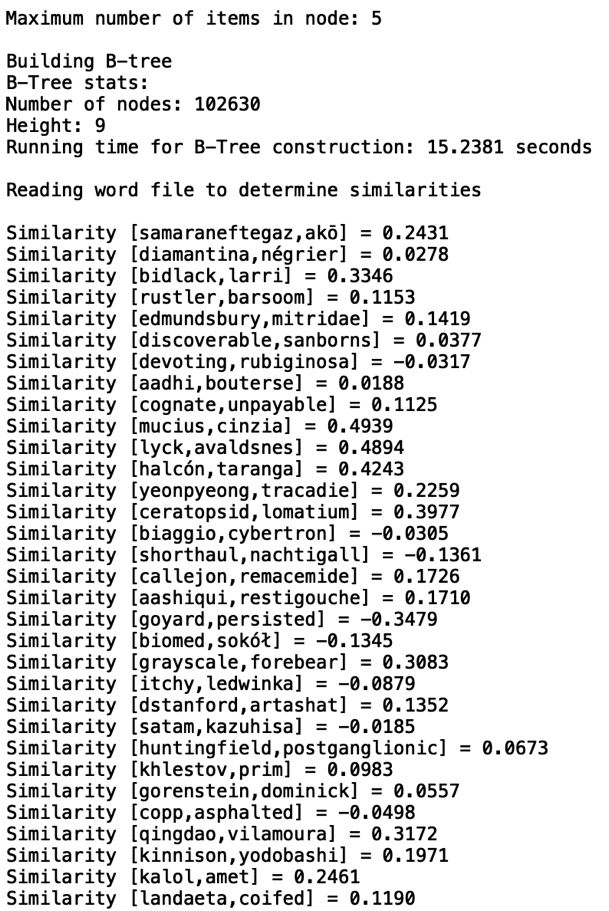


.

.

.

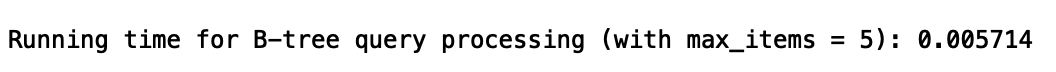




.

.

.



Case 1 (100):

Trial1\_100.txt

Trial2\_100.txt

Trial3\_100.txt

BST Number of Nodes: 327, 091, Height = 48

B-Tree, max\_data = 5, Number of Nodes: 102,630, Height = 9

B-Tree, max\_data = 15, Number of Nodes: 32,066, Height = 5

B-Tree, max\_data = 99, Number of Nodes: 4,751, Height = 2

B-Tree, max\_data = 499, Number of Nodes: 951, Height = 2



Case 2 (500):

Trial1\_500.txt

Trial2\_500.txt

Trial3\_500.txt

BST Number of Nodes: 327, 091, Height = 48

B-Tree, max\_data = 5, Number of Nodes: 102,630, Height = 9

B-Tree, max\_data = 15, Number of Nodes: 32,066, Height = 5

B-Tree, max\_data = 99, Number of Nodes: 4,751, Height = 2

B-Tree, max\_data = 499, Number of Nodes: 951, Height = 2



Case 3 (1000):

Trial1\_1000.txt

Trial2\_1000.txt

Trial3\_1000.txt

BST Number of Nodes: 327, 091, Height = 48

B-Tree, max\_data = 5, Number of Nodes: 102,630, Height = 9

B-Tree, max\_data = 15, Number of Nodes: 32,066, Height = 5

B-Tree, max\_data = 99, Number of Nodes: 4,751, Height = 2

B-Tree, max\_data = 499, Number of Nodes: 951, Height = 2



Case 4 (2000):

Trial1\_2000.txt

Trial2\_2000.txt

Trial3\_2000.txt

BST Number of Nodes: 327, 091, Height = 48

B-Tree, max\_data = 5, Number of Nodes: 102,630, Height = 9

B-Tree, max\_data = 15, Number of Nodes: 32,066, Height = 5

B-Tree, max\_data = 99, Number of Nodes: 4,751, Height = 2

B-Tree, max\_data = 499, Number of Nodes: 951, Height = 2



Case 5 (5000):

Trial1\_5000.txt

Trial2\_5000.txt

Trial3\_5000.txt

BST Number of Nodes: 327, 091, Height = 48

B-Tree, max\_data = 5, Number of Nodes: 102,630, Height = 9

B-Tree, max\_data = 15, Number of Nodes: 32,066, Height = 5

B-Tree, max\_data = 99, Number of Nodes: 4,751, Height = 2

B-Tree, max\_data = 499, Number of Nodes: 951, Height = 2



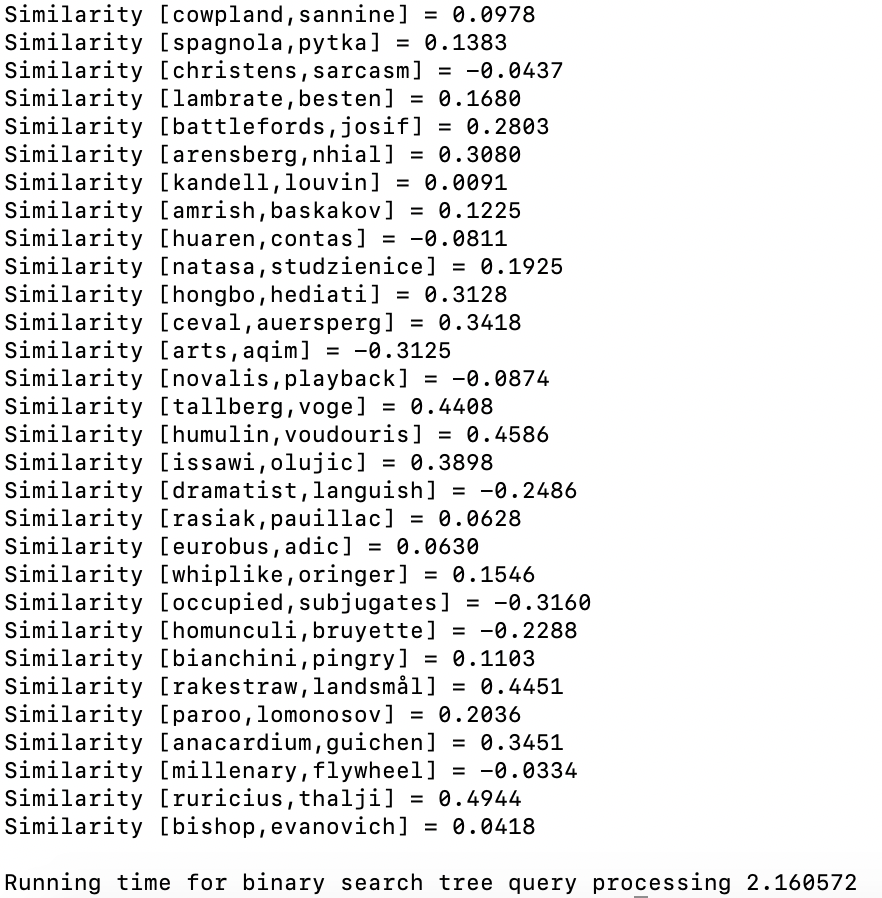


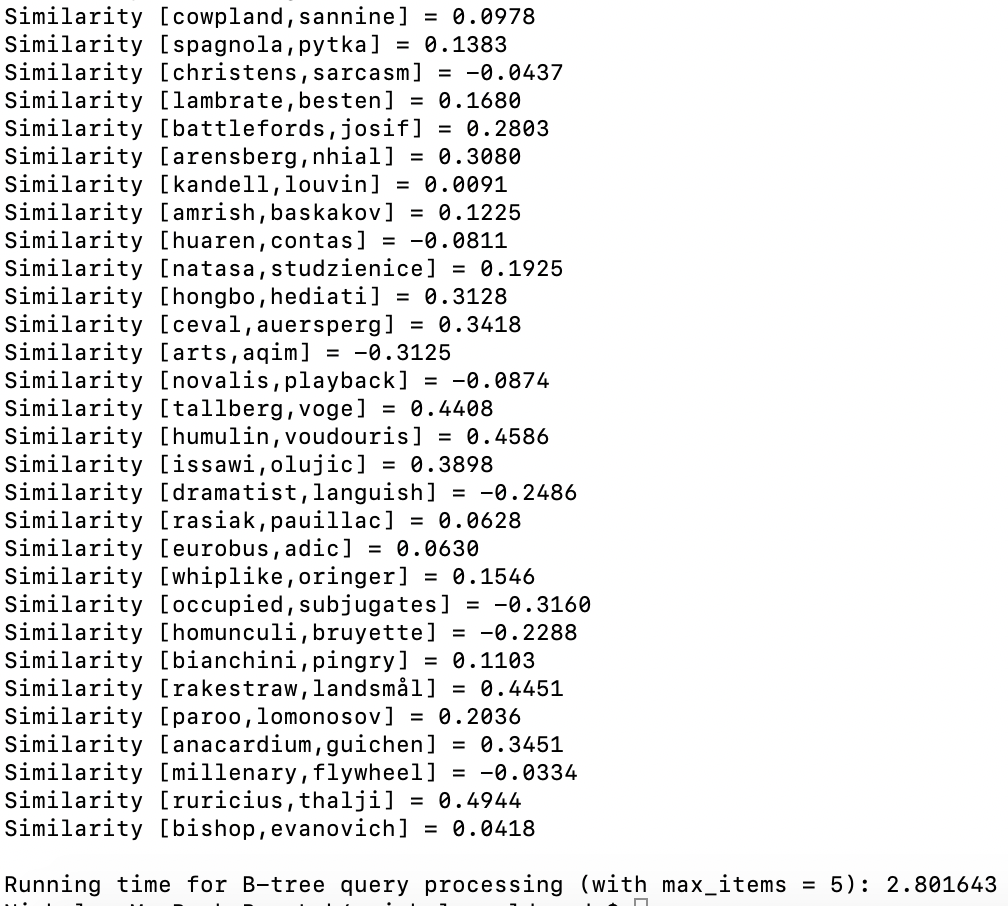
The graph above summarizing the tables clearly demonstrate that in all cases the B-Tree with max\_data of 5 had the worst runtime in all cases. Since max\_data holds a very limited number of nodes, the Binary Search performed on each node visited will probably compare at least 2 to 3 values. Thus, the effects of constantly dividing the data is not evident, resulting in a longer runtime. However, as the max\_data increases, the effects of using Binary Search to find the index of the embedded word or the next child to visit, becomes more evident. Since the B-Tree has larger lists of data, this results in smaller proportion of data being visited. Additionally, since the Binary Search is halving the data first from 499 to 249 to 124 … This is more significant then halving data from 5 to 2 to 1. Additionally, the height of the BTree plays an integral role in the runtime complexity. For max\_data of 499, its average running time was approximately 0.006 seconds less than the running time for the BST. Even the B-Tree with a max\_data of 99, was 0.001 seconds faster than the BST. The reason why is probably attributed to its shorter height. Whereas the BST has a height of 48, both B-Trees have a height of only 2. Then, with the binary search essentially halving the data seen in each two nodes, the superior runtime for the B-Tree with extremely large max\_data is evident. The B-Tree with a max\_data, however, must still traverse through multiple node’s and their data resulting in a longer running time than the BST. This can also be attributed to the fact that the BST requires less “code” comparisons of if branches, whereas the code for the B-Tree has more if statements for edge cases.

**Similarities – Medium Files:**

For the Similarity test, two tests were run. This test focused gathering the runtime complexities for files with 50,000 and 100,000 pairs. Each test had three different trials with the .txt files corresponding to the trial number. Each .txt file had randomly selected pairs. I thought that this test was very important to identify how the BST and B-Trees would respond to average medium sized data. I specifically chose these file sizes since 100,000 pairs is approximately a third of the number of words in the file. Additionally, the B-Trees tested had max\_data of 5, 15, 499, and 19999. I chose these values for the max\_data since they represent a wide range of values. I also included 19999 to see if there would be a significant change in the runtime complexity from 499 to 19999.

Sample Output:





Case 1 (50,000):

Trial1\_50000.txt

Trial2\_50000.txt

Trial3\_50000.txt

BST Number of Nodes: 327, 091, Height = 48

B-Tree, max\_data = 5, Number of Nodes: 102,630, Height = 9

B-Tree, max\_data = 15, Number of Nodes: 32,066, Height = 5

B-Tree, max\_data = 499, Number of Nodes: 951, Height = 2

B-Tree, max\_data = 19,999, Number of Nodes: 25, Height = 1



Case 2 (100,000):

BST Number of Nodes: 327, 091, Height = 48

B-Tree, max\_data = 5, Number of Nodes: 102,630, Height = 9

B-Tree, max\_data = 15, Number of Nodes: 32,066, Height = 5

B-Tree, max\_data = 499, Number of Nodes: 951, Height = 2

B-Tree, max\_data = 19,999, Number of Nodes: 25, Height = 1



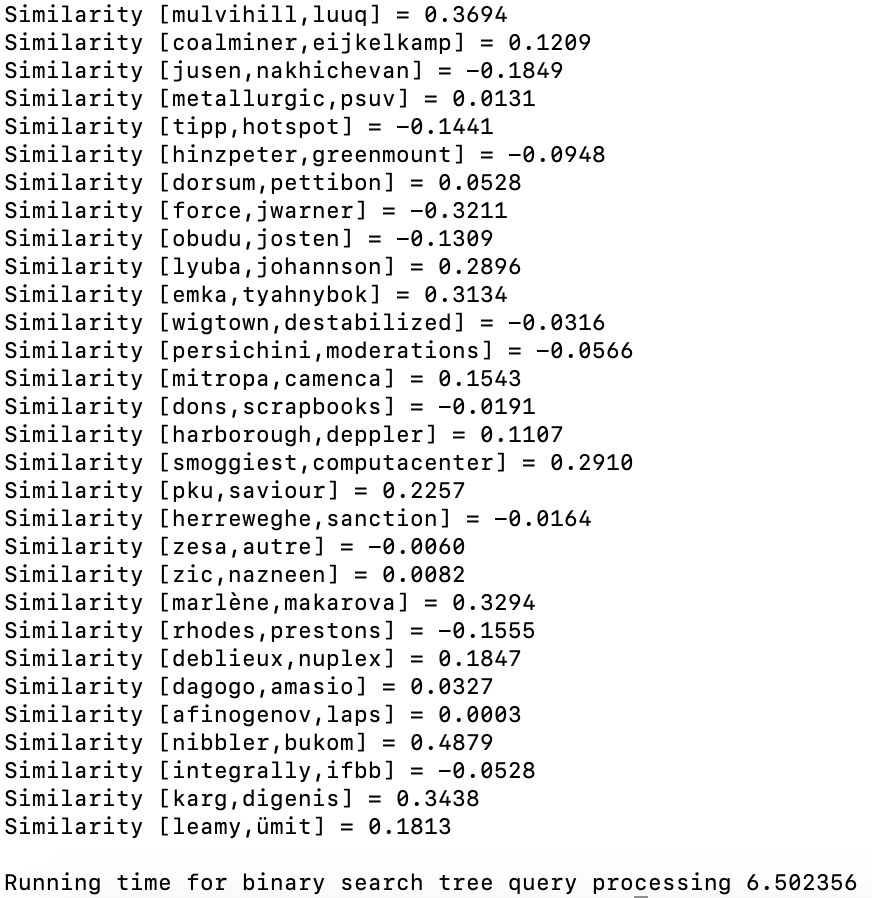


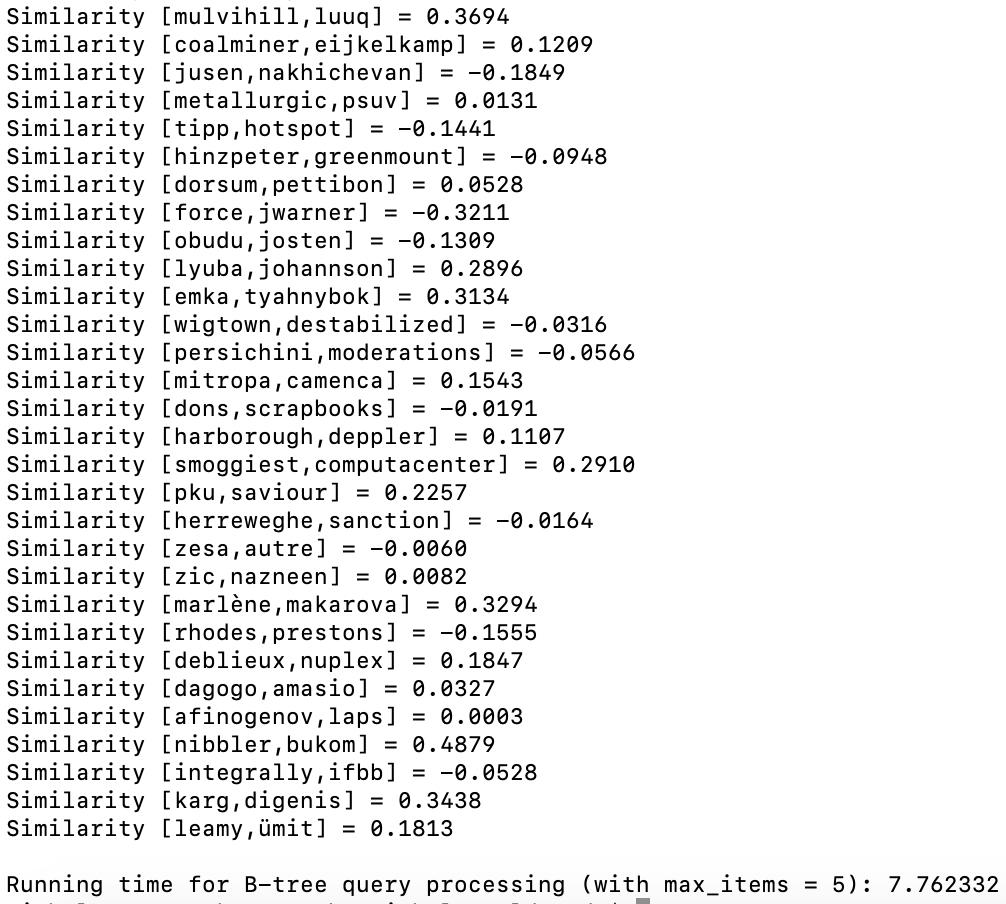
The graph above summarizing Case 1 and Case 2, demonstrates, the runtimes for the BST and B-Trees with max\_data of 5, 15, 499, and 19999. For the most part, the runtimes for all the data was closely related. As with any logarithmic function, after the initial curve in data, the runtimes start to settle out. This is clearly demonstrated in the graph and demonstrates that the algorithm is consistent. For the first case, the BTree with max\_data of 5 had a slightly higher performance time than the BST but the roles were reversed for the second case. This demonstrates that for either the BST or B-Tree with max\_data of 5, it is possible for either to perform better than the other. This again is attributed to where the data searched for is located in the tree. Since the B-Tree with max\_data of 5 has a height of 9, it is possible that the value searched for could be at the very bottom of the tree and as far left in the list as possible. The same data could be located at the very top of the BST resulting in a faster performance time. This yet again demonstrates how different words location in the trees affect the runtime complexity, exhibiting how crucial it is to have different randomized pairs to search from. The relationship between the B-Tree with max\_data of 499 and with max\_data of 19,999 is also evident. Since the height of the tree with max\_data of 19,999 is 1 level less than that of the tree with max\_data of 499, in the worst case, 1 less node needs to be visited. This results in less data that needs to be visited resulting in a faster running time. However, as the graph shows, this difference is very minor. Lastly, the graph clearly demonstrates that the B-Trees with the smallest max\_data have the highest runtimes when compared to the BST. Although the runtime for the BST is larger than the B-Tree with max\_data of 5 at 100,000 pairs, the B-Tree with max\_data of 15 is still larger. Yet again, more levels, result in more data searches, which results in a longer running time.

**Similarities – Large Files:**

For the Similarity test, three tests were run. These tests focused gathering the runtime complexities for files with 150,000, 200,000, and 300,000 pairs. Each test had three different trials with the .txt files corresponding to the trial number. Each .txt file had randomly selected pairs. I thought that this test was very important to identify how the BST and B-Trees would respond to average medium sized data. I specifically chose these file sizes since 300,000 pairs is is close to the 327, 091 words in the file. I wanted to test that how each algorithm would respond if a search for nearly every word. Additionally, the B-Trees tested had max\_data of 5, 15, 499, and 19999.

Sample Output:





Case 1 (150,000):

Trial1\_150000.txt

Trial2\_150000.txt

Trial3\_150000.txt

BST Number of Nodes: 327, 091, Height = 48

B-Tree, max\_data = 5, Number of Nodes: 102,630, Height = 9

B-Tree, max\_data = 15, Number of Nodes: 32,066, Height = 5

B-Tree, max\_data = 499, Number of Nodes: 951, Height = 2

B-Tree, max\_data = 19,999, Number of Nodes: 25, Height = 1



Case 2 (200,000):

Trial1\_200000.txt

Trial2\_200000.txt

Trial3\_200000.txt

BST Number of Nodes: 327, 091, Height = 48

B-Tree, max\_data = 5, Number of Nodes: 102,630, Height = 9

B-Tree, max\_data = 15, Number of Nodes: 32,066, Height = 5

B-Tree, max\_data = 499, Number of Nodes: 951, Height = 2

B-Tree, max\_data = 19,999, Number of Nodes: 25, Height = 1



Case 3 (300,000):

Trial1\_300000.txt

Trial2\_300000.txt

Trial3\_300000.txt

BST Number of Nodes: 327, 091, Height = 48

B-Tree, max\_data = 5, Number of Nodes: 102,630, Height = 9

B-Tree, max\_data = 15, Number of Nodes: 32,066, Height = 5

B-Tree, max\_data = 499, Number of Nodes: 951, Height = 2

B-Tree, max\_data = 19,999, Number of Nodes: 25, Height = 1





The graph above summarizing Case 1 to Case 3, demonstrates the runtimes for the BST and B-Tree with max\_data of 5, 15, 499, and 19,999. Yet again, the B-Tree with max\_data 5 had the highest performance time. This is because it has a larger height than the other B-Trees. Additionally, the halving of data through the binary search is less evident since only a maximum of 5 values can be stored per node. The B-Tree with max\_data of 19,999 continued to have the lowest runtimes for each case. Since the B-Tree only has 2 levels of nodes, one could almost compare it to a binary search on a linked list, which can be achieved in O(logn) where n is the number of nodes in the list, and consequently the amount of words stored in the entire B-Tree. Thus, by compressing the tree through a larger max\_data, a better runtime is achieved. As for the BST, its running time was still lower than the B-Tree with max\_data of 5 and max\_data of 15. This is attributed to the fact that the BST only needs to look at the current node’s data whereas the B-Tree has to look at a list of data at each row. As for the B-Tree’s with larger max\_datas the BST has a slower running time. Since the B-Tree is more compressed it is able to compensate with the fact that it has a list of data in each node thanks to the binary tree and minuet height.

**Worst Case Performance Time Comparison:**

The graphs below compares the running times for BSTs and B-Trees with max\_data of 5, 15, 499, and 19,999. The first graph shows the growth in runtimes for 2,000 pairs to 100,000 pairs. The second graph shows the growth in runtimes for 50,000 to 300,000. I thought that this comparison was extremely important since each category of pair sizes were tested individually. The best way to evaluate how the growth rates changed as the amount of pairs to be compared grew was by graphing the overall change.







By viewing the graphs above, it is very clear that the B-Tree with max\_data of 5 had the longest runtimes. This is because the tree has a larger height than the other B-Trees. Additionally, since the length of each node’s data is small, the effects of the binary search implemented on the list is neglible. The B-Tree with max\_data of 499 and 19,999 had the overall lowest runtime complexities. By having a smaller height and larger data lengths, the tree becomes compressed. Thus, when binary search is performed, its ability to continue to halve the array is noticeable. As for the binary search tree, its runtimes were the third lowest. Since the tree was somewhat balanced, its runtime complexity was closer to O(logn) where n is the number of nodes in the tree. This can be seen from the logarithm growth of the graphs. From the graphs it can also be concluded that for larger searches, an increasing the max\_data of the B-Tree will result in faster runtimes. Thus, the importance of the max\_data parameters of the B-Tree is clearly evident. As for the BST, it had a shorter runtime than half of the B-Trees tested. Thus, if one simply wanted to store a single piece of data in a node, using a BST is still a good choice. However, the optimal choice for larger sets of data would be to use a B-Tree with a larger max\_data.

**Conclusion**

In all, I learned about a real world approach for B-Trees and Binary Search Trees. In most computer science classes, different computer science theories and data structures are taught. Often the application of these data structures and algorithms is neglected. Thus, I thought that this lab was very interesting since it introduced a new topic – word embeddings. I had often wondered how Natural Language Processing applications like Siri and Alexa worked. I did not know the context of these words were compared by a list of vectors. This lab also gave me a better insight to the role of max\_data. Before completing this lab, I always thought that a B-Tree with a minimal max\_data would be the most effective. However, after completing this lab, I learned that for larger amounts of data a larger max\_data is more effective. I also learned that a BST is still a useful data structure, but I can clearly see B-Trees are used for data bases. Since databases read and write larger groups of data, a B-Tree would clearly be more efficient than a BST. This lab was also a great refresher for object – oriented programming. Since the word embeddings are essentially objects incasing data inside of another object, it was difficult at first to conceptualize, especially for the B-Tree. However, as I continued with my program I became more comfortable with comparing the data inside the WordEmbedding objects. Another interesting topic that I learned about in this lab was how to write files in python. When creating the .txt files for the pairs of words, I realized that I need randomized pairs to attain the correct results. At first, I thought of creating the files by hand, but realized that I wanted to test very large files. I thus created a program that would read the file with the word embeddings and store it in a list. Then two random indices were selected for each word pair and the corresponding element was written to the list. In all, this lab allowed me to compare B-Tree and Binary Search Tree in a real-world setting. Prior to the lab, my knowledge of the benefits and drawbacks of using B-Trees and Binary Search Trees was limited. However, I now know that when given large groups of data B-Trees would be the most efficient to search the data. Due to the multiple insertion methods, the construction of the B-Tree takes longer than that of a BST. Thus, if one wanted to quickly construct a tree from a group of data, then they should select a BST. As for the max\_data, a larger max\_data is better for searches. One must also be careful since a larger max\_data can sometimes result in larger tree building times.

**References**

Lynn, Shane. (n.d.). *Get Busy with Word Embeddings – An Introduction.* Retrieved from <https://www.shanelynn.ie/get-busy-with-word-embeddings-introduction/>

**Appendix A**

**NicholeMaldonado\_Lab4.py**

**SortedList.py**

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.

X



-----------------------------------