**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #4**

Due: October 22nd, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab, we were asked to compare the running times of a binary search tree (BST) and a B-tree, two data structures both designed to hold a list of objects in ascending order. The objects in question for this lab are those of the WordEmbedding class, in which a word and its vector description, or embedding, is stored. To implement these data structures, we will read a set of 400,000 words and their embeddings from a file and construct instances of these trees using the classes and methods defined previously in class and written in previous assignments. The main objective of this lab is to recognize how the nature of each data structure influences its respective running time, as well as to ensure an understanding of how to access any particular element of these data structures.

**Proposed Solution Design and Implementation**

**Item #1 (User Prompt):**

For this item, we simply place the initial prompt and respective initial construction of the chosen tree in a try-except statement such that if neither 1 for a BST or 2 for a B-tree are entered in response to the prompt, an exception will be raised and the user will be informed of their invalid entry before being given the prompt again. Once either 1 or 2 is selected, the user will be informed that the desired tree is being constructed. Also, if 2 is selected, then the user will also be asked for the maximum number of items to be stored in a node of the B-tree.

**Item #2 (Tree Construction):**

As for said construction, both involve simply reading the file of 400,000 words and their embeddings line by line, constructing a WordEmbedding object by splitting a given line and using the resulting list’s first element as the word value and the remaining list as the embedding value, and then inserting said objects into an initially empty tree; however, in the case of the BST, I found that the tree first had to have a node before further nodes could be inserted, so I simply had the program read the first line and make its resulting WordEmbedding object the root before continuing with the same for loop used for the entirety of the B-tree construction, as this did not lead to the first line being read twice.

Something else to note is that the BST and B-Tree classes previously established in class were not initially suited to handle WordEmbedding objects. Thus, the changes to be made involved simply changing any reference to the data of a given to the reference as said data’s word value, such that comparisons could properly be made during sorting and inserting. Also, on the topic of inserting, I found that the given *sort()* method used in the *InsertLeaf()* method of the B-Tree class failed to operate given the type of object. To fix this, I implemented the *quicksort()* method from Lab 2, changing the references in the *partition()* method to references of the list’s elements’ word values, such that the vital sorting process of the method could still take place.

Once the construction was complete, I had the program report the number of nodes, the height, and the amount of time taken to make the resulting tree. For the method that calculated the number of nodes in particular, I simply took the method that computed the number of items in a tree previously written for the exercise on B-trees, and altered it such that rather than adding up each and every item of a given node, it simply added up the nodes detected.

**Item #3 (Two-Word Similarity Computation):**

For this item, a second .txt file had to be written to simulate the user input for this program’s similarity computation utility. This second file consisted of two words per line, and initially contained the pair of words as depicted in the appendix of this lab’s handout such that I could be assured that my coding produced the correct results. From there, I simply copied those same 15 pairs from the appendix 10 times over, since the point is not to check that every answer is correct (as we can assume they will be given the initial testing), but to see how the running times vary with larger amounts of comparing, which in this case will be 150 elements over 15.

As for how my coding was constructed, it was about as simple as writing down the given formula using the *numpy* methods. What first came as a struggle was reading the file properly. To do this, I first created a list of the lines by splitting the whole file with the *splitlines()* method, such that the dangling line breaks were not included. From there, the program goes line by line, splitting each line into a list of two words, and then uses either the *Find()* method of the BST class or the *Search()* method of the B-tree class (determined by the previous tree choice integer from the user prompt) to access the embeddings of said two words as stored in the given tree. Then, those embedding values could be computed in the given formula with the *numpy* methods to find the similarity of the two words, or the cosine distance of their embeddings. Before the for loop would continue to the next line and its comparison of two words, the similarity of the current two words would be displayed such that each resulting value did not have to be stored in a list.

**Item #4 (Displaying the Running Times of Tree Construction and Similarity Computation):**

As mentioned previously in Item #2, the running time required to build a given tree was displayed after its construction. This time was calculated by taking the internal time right before the file was read and the internal time after the last word was inserted into the given tree and simply displaying the difference between the two.

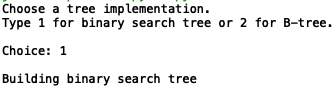
As for the running time of the similarity computation, a variable *total* was used to track the cumulative differences between the start and end internal times for each iteration of the for loop that would find the similarity between the two words of a given line. After all the similarity values are printed out, and thus the second file finished being read, this cumulative total is then displayed at the very end, noting which of the two data structures was being used by using one of two nearly identical print statements, again determined by the initial tree choice input.

**Experimental Results**

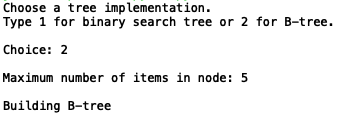
**Item #1:**

For this item, I tested inputting 1, 2, and 3 at the beginning of the program, and as expected, the program began either constructing the appropriate tree or looping back to the initial prompt if an invalid entry was given. Also, for input 2, the follow-up prompt requesting the maximum number of items to be held within a node of the B-tree appeared and functioned properly.

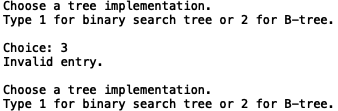
Input 1:



Input 2:



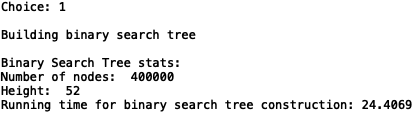
Input 3:



**Item #2:**

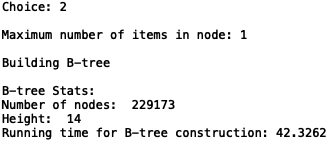
For this item, as expected, the BST stats reported 400,000 nodes being created, one for each WordEmbedding object. The minimum height of a BST is floor(log2*n*) and the maximum height is *n* - 1, where *n* is the number of nodes. Thus, for 400,000 nodes, the minimum height is 18 and the maximum is 399,999. In this case, the height is 52, thus much closer to the minimum height.

BST:

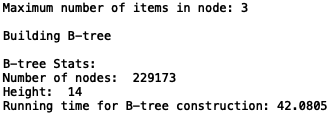


As for the B-tree stats, I sought to compare the number of nodes and resulting heights of several B-trees based on the same set of 400,000 words, but given different maxItems values. Given the constructor method for the BTree class, maxItems = 1, 3 created the same B-tree. Using maxItems = 5 made for the most efficient construction time, despite maxItems = 7, 9 making smaller trees with fewer nodes and lesser heights. This is presumably because trying to store more elements within a single node leads to longer sorting processes as more elements are inserted.

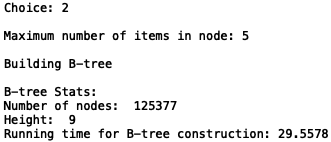
B-Tree, maxItems = 1:



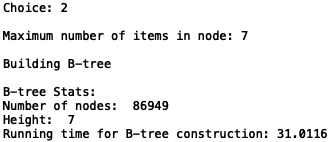
B-Tree, maxItems = 3:



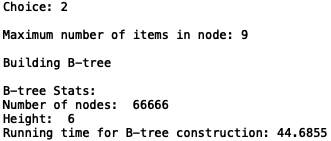
B-Tree, maxItems = 5:



B-Tree, maxItems = 7:



B-Tree, maxItems = 9:

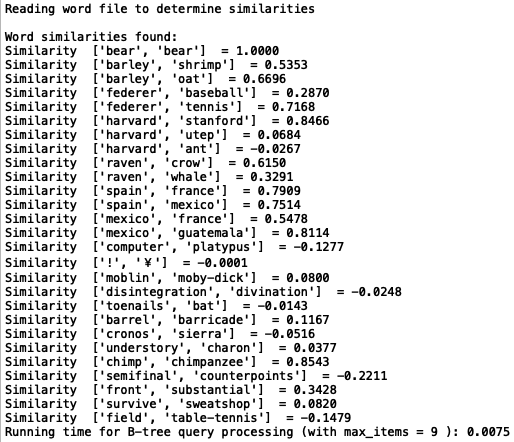


**Item #3:**

For this item, I took the total running times of each block of similarity comparisons per BST or B-Tree of a given maxItems value. Of note is that the word similarities are the same for each and every tree, meaning the process of finding the nodes with those words was always possible.

Also, in regards to the running times, the BST appears to have the fastest time making 150 comparisons for similarity, whereas the B-Tree appears to have faster and faster times the more items there are within a single node.

Results of Similarity Comparison for All Trees:



BST:



B-Tree, maxItems = 1:



B-Tree, maxItems = 3:



B-Tree, maxItems = 5:



B-Tree, maxItems = 7:



B-Tree, maxItems = 9:



**Item #4:**

For this item, the running times are all depicted in the previous items.

**Tables:**

**Running Times (ms)**

|  |  |  |
| --- | --- | --- |
| **Tree** | **Construction** | **Similarity Comparison** |
| **BST** | 24.4069 | 0.0053 |
| **B-Tree(maxItems = 1)** | 42.3262 | 0.0224 |
| **B-Tree(maxItems = 3)** | 42.0805 | 0.0192 |
| **B-Tree(maxItems = 5)** | 29.5578 | 0.0133 |
| **B-Tree(maxItems = 7)** | 31.0116 | 0.0128 |
| **B-Tree(maxItems = 9)** | 44.6855 | 0.0124 |

As the results indicate, constructing a BST and a B-Tree with maxItems = 5 (as it was originally intended) take the least amount of time, whereas constructing a B-Tree with maxItems != 5 results in longer running times. Regarding the B-Tree, this is likely because in the case of smaller maxItems values, splitting of nodes has to be performed more often, and in the case of larger maxItems values, more sorting has to be done for each insertion into a node. Also of note is that the running times for constructing a B-Tree with maxItems = 1 and a B-Tree with maxItems = 3 are practically identical, since, again, the constructor method of the BTree class makes any maxItems value smaller than 3 equal to 3, thus making the same tree.

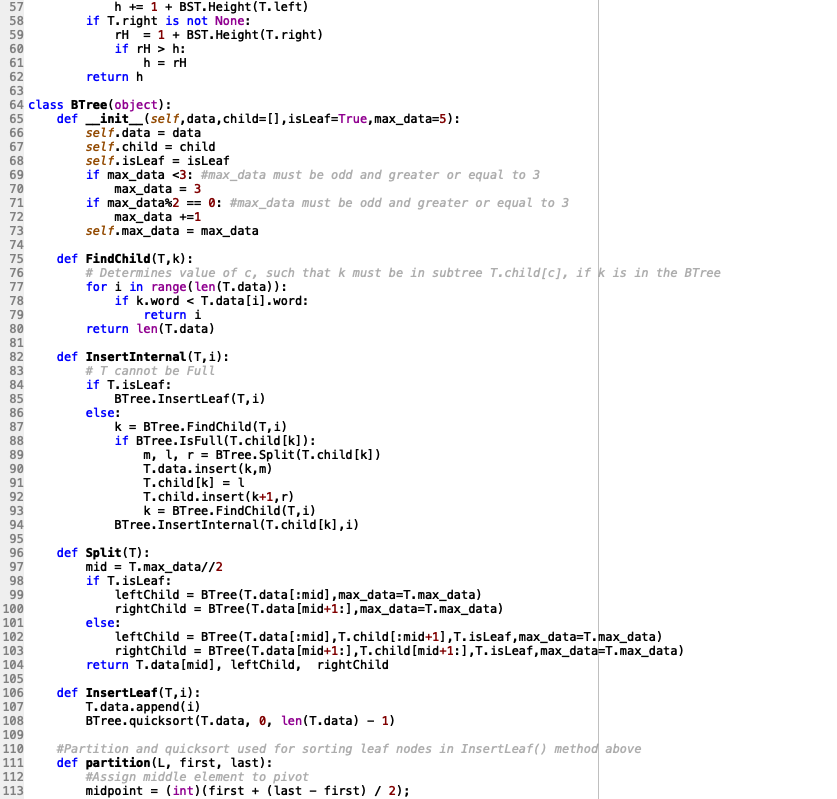
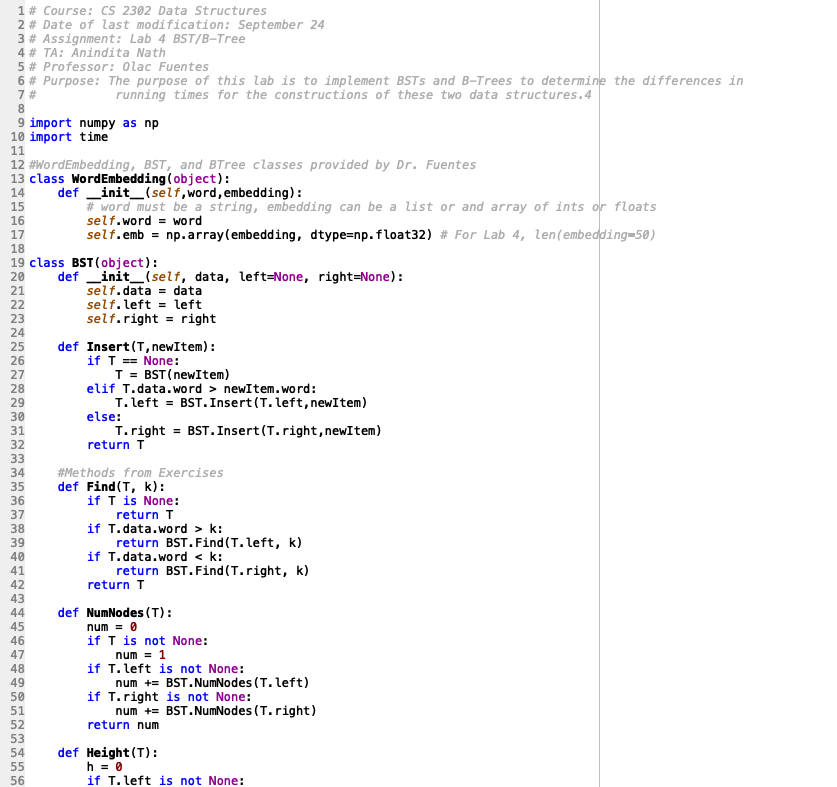
As for the running times of the similarity comparisons, the BST again takes the cake for fastest running time, whereas the B-Tree takes less and less time only as its nodes are able to hold more and more items. What this most likely implies is that if we were to reconfigure the \_\_init\_\_ method of the BTree class such that its max\_data value is expected to be a much greater number, then we could much more efficiently construct and traverse such a B-Tree, potentially to the point of outpacing the BST.

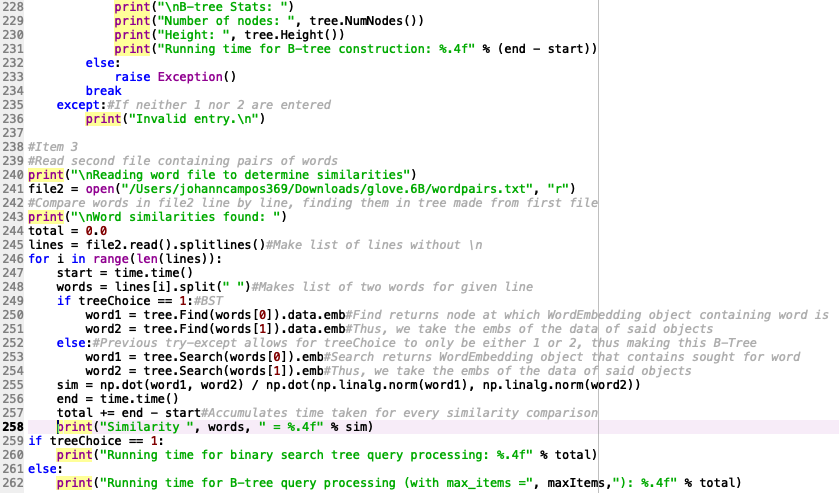
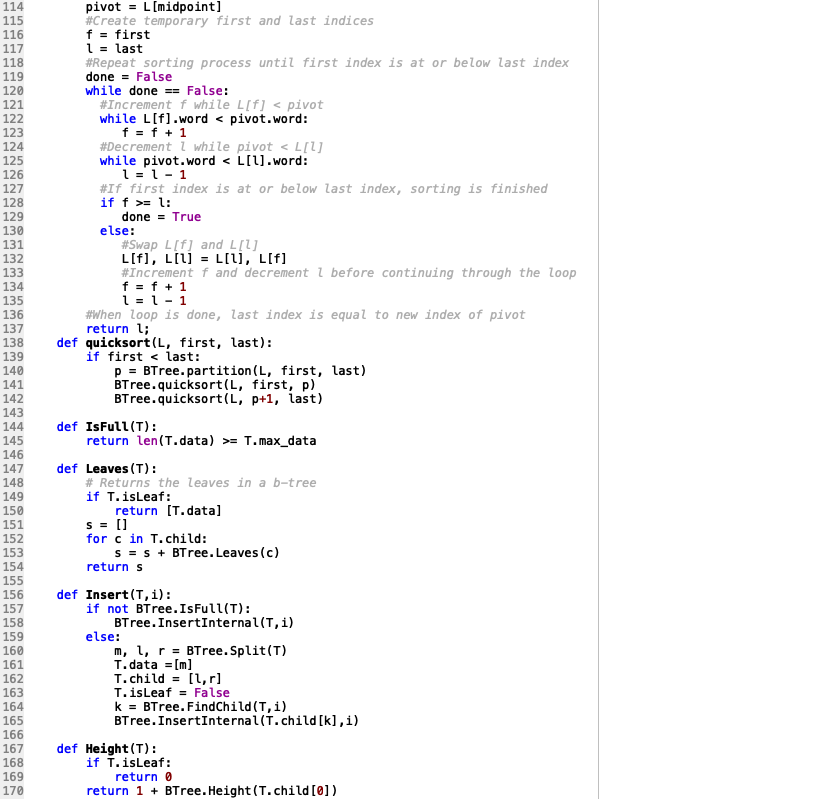
**Conclusion**

From this lab, I learned how to implement binary search trees and b-trees as methods of organizing large quantities of data. That said, I fear I might have gone astray when replacing the given sort() method in the *InsertLeaf()* method of the BTree class with the *quicksort()* method from Lab 2, as that may have hampered the BTree class altogether, but regardless, I did at least come to see the connection between the increase of items per node positively correlating with the time needed to construct a BTree and negatively correlating with the amount of time needed to find a specific element.

I also gained experience handling a more unorthodox object (at least in a classroom setting), in this case, the WordEmbedding object. When it comes to representing words, it does seem like a beneficial way of expressing written language numerically such that something like finding how similar two words are can be done. It might not be immediately apparent how these embeddings relate to their assigned words, but the conclusion that the words “bear” and “bear” are so similar as to be exactly the same is intuitive enough.

**Appendix**





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.