**Author:** Julian Gonzalez

**Assignment:** Lab 5 Report

**Course:** CS 2302 - Data Structures 10:30-11:50

**Instructor:** Fuentes, Olac

**T.A.:** Nath, Anindita

***Introduction***

Our given task was to highlight how similar two words are given a text file that contained words and vectors that show how similar a word is to another essentially a big matrix an put it into a Hash Table or a Binary search tree.

***Solutions***

The first task was to prompt the user which type of data structure they wished to use either hash table or binary search tree. This is easily done with the input function and some if statements for each of the corresponding date structures.

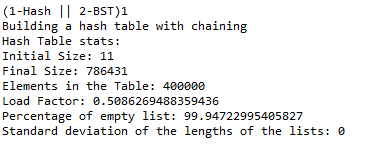
Our second task was to read the given txt file and then store it in the user choice of data structure. Reading the file to hash table and BST was done essentially the same the only difference between the two was how the data was store, both of which had their own functions that would do this. The insert function for the hash table was fundamentally the same from the given code but with the one change of checking the load factor if it was below 1. The insert function for the BST was also very similar to the given code that we have had used previously

The third task was relatively simple we where asked to show specific stats after creating each data structure. Hash table stats was initial size, final size, load factor, percentage of empty list, and the standard deviation of the length of lists. The ones that require more thought into them was the load factor and percentage of empty lists. For the load factor I called a function that would return the number of items in the hash table / the length of the hash table. For percentage I used a function again to count the number of empty lists in the hash and then divided it but the number of items of the entire list multiplied by 100 to obtain a percentage. For BST we had to display the Number of nodes in the Tree which was easily done with a previous function that we had used for another lab with some modification. We also had to display the height of the tree again easily done with code that was created from exercises done in class.

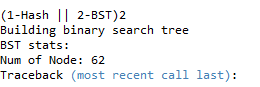
Now after reading and store the contents of the file into each respective data structure, the last major thing that had to be done was to read user provided text file and compare words similarities using our store data structure data. To do this both the hash and BST read a text file which would read line by line storing the words then I would use those two words to compare to each other. The comparing was done by a separate function for BST and hash. In both of them the code was essentially similar with the only difference being the way we found the specific word and its embedding in either the hash or BST. A for loop was used that would obtain the dot product of the word and its two magnitudes. Finally, the dot product was divided by the two magnitudes multiplied together both of which squared beforehand. Then the similarity value was returned to be printed. The last thing asked of use was to find the running times of each data structure which was easily done with the time.time() function and printed at the end of each call

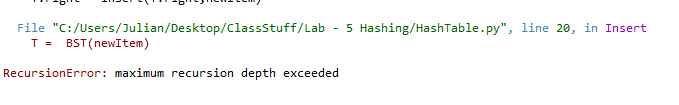
***Experimental***

A screenshot of a cell phone

Description automatically generated For my initial runs a I used a file containing 5 comparisons which would allow for the program to run quickly but eventually I did use a bigger file to find the time complexity of each Data structure. Here are the outputs with 5 comparisons happening. For the runtimes I found something very interesting the time for comparing 5 words 1150 lines of words compared to each other the almost the same runtime. The runtime for comparisons seemed to be constant so O(1) but the one thing is the reading of the file which would take N amount of time because it depends on the size of the file. This really confused me as the runtime of the file with 5 comparisons was 10.4485 and the file with 1150 comparisons was 10.9825. Overall the runtime seems to be O(n^2) as the for loop to read the initial file and the case of reaching the load factor and iterating the size to double are dependent on each other.

For BST I had many problems when it came to reading the provided file into a BST.If I used a significantly smaller file for creating the BST the program was able to create the BST instead of throwing a “maximum recursion depth exceeded” error, but afterwards the program would fail to search through the tree to find the words and their embedding I believe this may have to do with the words now not existing in the BST to be found, this could be solved with a case to handle this in the BST search function

Here’s an example of the output when reading the large provided file “glove.6B.50d.txt”

Here is the Code working when the file to embed into the BST is significantly smaller but then I run into the problem stated above which does not allow me to do comparisons.

***Conclusion***

This lab was tough, but I really enjoyed working with the Hash Table and figuring out how to implement of what was asked of us. The BST part of the lab was not too bad, but I had problems when it came to having a functioning code in certain scenarios.

Appendix:

1. # -\*- coding: utf-8 -\*-
2. """
3. Created on Sun Mar 31 19:54:48 2019
4. @author: Julian
5. Assignment: Lab 5
6. Instructor: Fuentes, Olac
7. T.A: Nath, Anidita
8. """
9. **import** time
10. **import** math
11. **import** numpy as np
13. **class** BST(object):
14. # Constructor
15. **def** \_\_init\_\_(self, item, left=None, right=None):
16. self.item = item
17. self.left = left
18. self.right = right
20. **def** Insert(T,newItem):
21. **if** T **is** None:
22. T =  BST(newItem)
23. **elif** T.item[0] > newItem[0]:
24. T.left = Insert(T.left,newItem)
25. **else**:
26. T.right = Insert(T.right,newItem)
27. **return** T
29. **def** IterSearch(T,k):
30. **while** T **is** **not** None:
31. **if** T.item[0] == k:## if the current node of the tree is k return current node T
32. **return** T.item[1]
33. **elif** T.item[0] > k:#checks if k is greater than current node item if it is continues to the right of the tree
34. T = T.left
35. **else**:#if all other checks fail it will defualt to looking in the left tree
36. T = T.right
37. **print**('Could not find k inside the list, inserting k into list')
38. T = Insert(T,k)# doesnt actually insert into the main list
39. **return** T
41. **def** height(T):
42. **if** T **is** **not** None: #base case
43. **return** 1+max([(height(T.left)),height(T.right)])

46. **def** numNodes(T):
47. **if** T **is** **not** None:
48. **return** 1 + numNodes(T.left)+numNodes(T.right)
49. **return** 0
51. **def** similarBST(T,w1,w2):
52. ww1 = IterSearch(H,w1)#finds the word in the bst
53. ww2 = IterSearch(H,w2)
54. top = 0#dot product
55. bottom\_a = 0#mag 1
56. bottom\_b = 0#mag 2
57. **for** i **in** range(len(ww1)):#does the formula to find magnitude
58. top += float(ww1[i]) \* float(ww2[i])
59. bottom\_a += float(ww1[i]) \*\* 2
60. bottom\_b += float(ww2[i]) \*\* 2
62. bottom\_a = math.sqrt(bottom\_a)
63. bottom\_b = math.sqrt(bottom\_b)
64. ###########################################################
65. **class** HashTableC(object):
66. **def** \_\_init\_\_(self,size):
67. self.item = []
68. self.numItems = 0#Tracks bumber of items in the entire hash
69. **for** i **in** range(size):
70. self.item.append([])
72. **def** CalFactor(H):#calculates load factor
73. **return** H.numItems / len(H.item)
75. **def** InsertC(H,k,l):
76. # Inserts k in appropriate bucket (list)
77. # Does nothing if k is already in the table
78. b = h(k,len(H.item))
79. **if** CalFactor(H) >= 1.0: ## checks the load factor if greater than increase the size of the hash
80. **for** i **in** range(len(H.item)+1):
81. H.item.append([])
82. b = h(k,len(H.item))
84. H.item[b].append([k,l])  ##MIGH NOT NEED L
85. H.numItems = H.numItems + 1#increases to keep track
87. **def** FindC(H,k):
88. # Returns bucket (b) and index (i)
89. # If k is not in table, i == -1
90. b = h(k,len(H.item))
91. **for** i **in** range(len(H.item[b])):
92. **if** H.item[b][i][0] == k:
93. **return** H.item[b][i][1]
94. **return** -math.inf
96. **def** percentage(H):
97. c = 0
98. **for** i **in** range(len(H.item)):
99. **if** len(H.item[i])==0:
100. c = c +1
101. #print(c)
102. **return** c / len(H.item)\*100
104. **def** similar(H,w1,w2):
105. ww1 = FindC(H,w1)#finds the word in the heap with its embedding
106. ww2 = FindC(H,w2)
107. top = 0#dot product
108. bottom\_a = 0#mag 1
109. bottom\_b = 0#mag 2
110. **for** i **in** range(len(ww1)):#does the formula to find magnitude
111. top += float(ww1[i]) \* float(ww2[i])
112. bottom\_a += float(ww1[i]) \*\* 2
113. bottom\_b += float(ww2[i]) \*\* 2
115. bottom\_a = math.sqrt(bottom\_a)
116. bottom\_b = math.sqrt(bottom\_b)
118. **return** top / (bottom\_a \* bottom\_b)
120. **def** h(s,n):
121. r = 0
122. **for** c **in** s:
123. r = (r\*n + ord(c))% n
124. **return** r
126. **def** reCompute(H):
127. temp = HashTableC(len(H.item))
128. **for** i **in** range(len(H.item)):
129. **for** j **in** range(len(H.item[i])):
130. InsertC(temp,H.item[i][j][0],H.item[i][j][1])
131. **return** temp
132. #file = "smaller\_file.txt"
133. file = "glove.6B.50d.txt"
134. size = 10
135. H = HashTableC(size)
136. **print**('Choose implementation:')
137. c = int(input("(1-Hash || 2-BST)"))
139. **if** c == 1:
140. **print**("Building a hash table with chaining")
141. start = time.time()
142. with open (file,encoding='utf8') as file:## makes the hashtable from a file
143. **for** l **in** file:
144. string = l.split(' ')
145. word = l.split(' ')[0]#gets position 0 of L (like a list)
146. emb = string[1:-1]#embed from beginning+1 to end of file
147. InsertC(H,word,emb)
148. **print**("Hash Table stats:")
149. **print**("Initial Size:",size)
150. **print**("Final Size:",len(H.item))
151. **print**("Elements in the Table:",H.numItems)
152. **print**("Load Factor:",CalFactor(H))
153. **print**("Percentage of empty list:",percentage(H))
154. **print**("Standard deviation of the lengths of the lists:",0)###DO THIS
155. Hre = reCompute(H)
156. **print**("\n Reading word file to determine similarities\n")
157. with open("similarities.txt",) as compares:
158. **for** line **in** compares:
159. string2 = line.split()
160. **print**(string2[0] + " " + string2[1]+ " ")
161. **print**(str(similar(Hre, string2[0], string2[1])))
163. end = time.time()
164. **print**("Running time",end-start )
166. **if** c == 2:
167. T = None
168. **print**("Building binary search tree")
169. start2 = time.time()
170. with open (file,encoding='utf8') as file:## makes the hashtable from a file
171. **for** l **in** file:
172. string = l.split(' ')
173. word = l.split(' ')[0]#gets position 0 of L (like a list)
174. emb = string[1:-1]#embed from beginning+1 to end of file
175. T= Insert(T,word)
176. **print**("BST stats:")
177. **print**("Num of Node:",numNodes(T))
178. **print**("Height:",height(T))
179. end2 = time.time()
180. **print**("Running time of building tree",end2-start2)
181. with open("similarities.txt",) as compares:
182. **for** line **in** compares:
183. string2 = line.split()
184. **print**(string2[0] + " " + string2[1]+ " ")
185. **print**(str(similarBST(T, string2[0], string2[1])))

188. **else**:
189. **print**("not a valid input")

I Julian Gonzalez 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.

* Julian Gonzalez