Tyler Salas

1:30-2:50 Class

Dr. Fuentes

80596221

Lab 5 Report

Lab 5, the lab assigned this week, was based around the idea of searching for words in different data structures and how efficient these structures were. The two structures we had to work with were a Binary-Search Tree and a Hash Table. Each node in each of the structures was required to contain a word, and an embedding that represented the words as points position in a fifty-dimensional space. With this, two words could be compared and their likeness would be returned as a float point from negative one to one (The cosine distance of the angle between the two points).

The base idea was to read the file given by the link provided by Dr. Fuentes, from the words and embeddings given by the file, the selected data structure would then be created containing all data needed to receive comparisons.

**Reading The File**

The program reads two different files, one that contains all the words that are to be compared in the comparisons area and the other that contains the words and embeddings. The files go through a process of three different methods that prepare it for insertion into the chosen data structure, both being the same methods.

In the first step, the method takes the file and stores each different line into a list after reading it. For the array with words comparisons, each index of the list contains only two words, for the other list each index contains the word and its embeddings.

In the second step, the lists with their data are split by the spaces in their lines. What this does is create a list of lists that contain all the elements of each index (Meaning the word and its embeddings). This is where the list containing two words stops, as there is no need for the next step.

From here the embeddings are separated from the words and put in a separate list in the [1] index each having their own index as they are changed from strings to floats in order to be used in computations.

**Binary Search Tree**

**Example Output**

What type of structure would you like?:

1. Binary Tree

2. Hash Table

Choice: 1

Building Binary Search Tree...

BST Stats:

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

Number of Nodes: 327091

Height Of Binary Tree: 19

Time Taken To Build BST: 30.162792444229126 Seconds

Reading Word File To Determine Similarities...

Word Similarities Found:

1 : ['keyboard', 'keyboard'] = 1.0000

2 : ['keyboard', 'computer'] = 0.5768

3 : ['keyboard', 'music'] = 0.5730

4 : ['remote', 'television'] = 0.3916

5 : ['remote', 'isolated'] = 0.7149

6 : ['remote', 'sofa'] = 0.1483

7 : ['ant', 'shopping'] = 0.1597

8 : ['ant', 'beetle'] = 0.6550

9 : ['ant', 'alarm'] = 0.1792

10 : ['computer', 'data'] = 0.7605

11 : ['computer', 'strap'] = 0.1247

12 : ['computer', 'platypus'] = -0.1277

13 : ['communism', 'stalin'] = 0.7117

14 : ['communism', 'reagan'] = 0.4582

15 : ['communism', 'toaster'] = 0.0559

16 : ['key', 'lock'] = 0.4309

17 : ['key', 'piano'] = 0.1862

18 : ['key', 'heart'] = 0.4214

19 : ['hole', 'fish'] = 0.4219

20 : ['hole', 'wall'] = 0.6157

21 : ['hole', 'window'] = 0.4941

22 : ['book', 'knowledge'] = 0.5801

23 : ['book', 'love'] = 0.7045

24 : ['book', 'camera'] = 0.3806

25 : ['pop', 'knuckle'] = 0.0899

26 : ['pop', 'britney'] = 0.5369

27 : ['pop', 'queue'] = -0.0207

28 : ['studio', 'magic'] = 0.4636

29 : ['studio', 'adidas'] = 0.1150

30 : ['studio', 'record'] = 0.4240

31 : ['engine', 'computer'] = 0.5230

32 : ['engine', 'car'] = 0.6771

33 : ['engine', 'camera'] = 0.4495

34 : ['opine', 'scowl'] = 0.1590

35 : ['hate', 'sisters'] = 0.2842

36 : ['bath', 'frame'] = 0.3244

37 : ['father', 'insert'] = 0.0081

38 : ['scorch', 'crooked'] = -0.0417

39 : ['offer', 'glove'] = 0.0498

40 : ['pay', 'flash'] = 0.1875

41 : ['canvass', 'spy'] = -0.3449

42 : ['wind', 'special'] = 0.2983

43 : ['name', 'groomed'] = 0.0441

44 : ['slim', 'scare'] = 0.0103

45 : ['balance', 'unbiased'] = 0.2617

46 : ['snatch', 'beat'] = 0.4500

47 : ['answer', 'slope'] = 0.1485

48 : ['scarf', 'ethereal'] = 0.2106

49 : ['condemned', 'cheer'] = 0.1387

50 : ['stitch', 'power'] = -0.0451

51 : ['savory', 'celery'] = 0.5940

52 : ['announce', 'airplane'] = 0.2902

53 : ['replace', 'hat'] = 0.2984

54 : ['reach', 'rule'] = 0.4308

55 : ['cheer', 'nifty'] = 0.2005

56 : ['salute', 'line'] = 0.1966

57 : ['comparison', 'wry'] = 0.2577

58 : ['spot', 'female'] = 0.3676

59 : ['spot', 'weights'] = 0.3185

60 : ['spot', 'partner'] = 0.3342

61 : ['spiky', 'rat'] = 0.2862

62 : ['futuristic', 'shock'] = 0.1602

63 : ['beginner', 'grain'] = 0.0669

64 : ['labored', 'girls'] = 0.0479

65 : ['drunk', 'bite'] = 0.3759

66 : ['manage', 'future'] = 0.7155

67 : ['pour', 'education'] = 0.1403

68 : ['holiday', 'force'] = 0.2302

69 : ['spy', 'shock'] = 0.2326

70 : ['mature', 'poison'] = 0.2573

71 : ['position', 'burn'] = 0.1873

72 : ['pushy', 'toad'] = 0.0967

73 : ['slope', 'leg'] = 0.3546

74 : ['vanish', 'north'] = -0.1118

75 : ['modify', 'hew'] = 0.2264

76 : ['position', 'plucky'] = -0.0461

77 : ['poised', 'spot'] = 0.5438

78 : ['purify', 'tendency'] = 0.1341

79 : ['knit', 'recess'] = -0.0641

80 : ['spend', 'sprout'] = 0.2464

81 : ['blush', 'spin'] = -0.0088

82 : ['class', 'smoke'] = 0.2048

83 : ['harmonious', 'transform'] = 0.5328

84 : ['address', 'chickens'] = 0.0814

85 : ['busy', 'colossal'] = 0.0562

86 : ['quickest', 'quickest'] = 1.0000

87 : ['prepare', 'blush'] = -0.0128

88 : ['sprout', 'smoke'] = 0.2358

89 : ['class', 'spin'] = 0.3252

90 : ['love', 'phone'] = 0.3889

91 : ['prepare', 'shame'] = 0.1990

92 : ['busy', 'harmonious'] = 0.2520

93 : ['transform', 'address'] = 0.3256

94 : ['chickens', 'forgetful'] = 0.0221

95 : ['lazy', 'afterthought'] = 0.3887

96 : ['utopian', 'fly'] = -0.1208

97 : ['resist', 'sail'] = 0.3016

98 : ['behavior', 'cease'] = 0.2517

99 : ['quack', 'sapien'] = 0.0300

100 : ['belligerent', 'lick'] = -0.1649

101 : ['work', 'wave'] = 0.4549

102 : ['seem', 'rub'] = 0.2795

103 : ['hand', 'six'] = 0.6449

104 : ['jellyfish', 'knee'] = -0.0101

105 : ['precious', 'quickest'] = 0.0605

106 : ['ten', 'spin'] = 0.3164

107 : ['class', 'colossal'] = 0.0822

108 : ['prepare', 'tell'] = 0.5782

109 : ['jellyfish', 'six'] = -0.0687

110 : ['chance', 'rattle'] = 0.2189

111 : ['hydrant', 'shame'] = -0.0292

112 : ['busy', 'smoke'] = 0.3902

113 : ['class', 'spin'] = 0.3252

114 : ['extend', 'unit'] = 0.3683

115 : ['stranger', 'momentous'] = 0.2470

116 : ['dad', 'different'] = 0.2794

117 : ['groovy', 'spark'] = -0.0178

118 : ['quiet', 'destroy'] = 0.2012

119 : ['familiar', 'talk'] = 0.7070

120 : ['wonder', 'dream'] = 0.8020

121 : ['oval', 'sleep'] = 0.0621

122 : ['arithmetic', 'enjoy'] = 0.0998

123 : ['marvelous', 'subscribe'] = 0.1014

124 : ['unaccountable', 'forecast'] = -0.1896

125 : ['tan', 'female'] = 0.3061

126 : ['full', 'ticket'] = 0.4987

127 : ['lake', 'remove'] = 0.1731

128 : ['boiling', 'snatch'] = 0.1840

129 : ['yawn', 'shit'] = 0.5201

130 : ['creature', 'contend'] = 0.0676

131 : ['forsake', 'business'] = -0.0443

132 : ['night', 'board'] = 0.4584

133 : ['fit', 'believe'] = 0.6197

134 : ['furniture', 'control'] = 0.2184

135 : ['distribute', 'prohibit'] = 0.5235

136 : ['robin', 'harsh'] = 0.0140

137 : ['conspire', 'persuade'] = 0.3794

138 : ['old', 'encouraging'] = 0.2056

139 : ['cart', 'dire'] = -0.0982

140 : ['water', 'give'] = 0.5000

141 : ['dye', 'add'] = 0.2638

142 : ['awake', 'scent'] = 0.1345

143 : ['burn', 'wound'] = 0.4740

144 : ['cheerful', 'clammy'] = 0.3751

145 : ['unruly', 'mew'] = -0.0386

146 : ['operate', 'fan'] = 0.2246

147 : ['coal', 'surpass'] = 0.2324

148 : ['door', 'balloon'] = 0.3650

149 : ['worship', 'limping'] = -0.1044

150 : ['fall', 'sentence'] = 0.3434

Time Taken To Find 150 Similarites: 0.05207681655883789 Seconds

For this portion of the lab a binary search tree was to be created to find each element. The word and embedding set is sorted by its [0] element where the words are contained. This is done using the list.sort() method from Python. From here a method named listToTree is called with and empty tree and the list containing the words and embeddings. The method finds the middle of the list by finding len(list)//2, from here it checks if T is none and if so a new Binary-Search Tree is created from the middle element (This will only happen in the initial call). From there the left node of the tree is recursively declared as calling the same tree and calling all word and embeddings before the middle element. The same is then done for the right node of the tree, only calling the word and embeddings after the middle of the list. The made tree is then returned.

Several Binary Tree Stats are also displayed.

**Number of Nodes**

This method is simple as it returns the number of nodes in the Binary-Search Tree. It checks if T is empty and if so returns zero, if not it returns one plus a recursive call to the left node plus a recursive call to the right node.

**Height of Tree**

This method returns the height of the given Binary-Search Tree. It calls a method that recursively calls itself to both the left and right, making these two different variables. It then compares the two at the end to see which is bigger and returns the number accordingly.

**Time Taken To Create the Tree**

This is a simple statistic used to get the amount of time taken to create the tree. It takes the start of the number of methods as time.time() and the end as time.time() and from there prints the end-start.

**Comparing Similarities**

This method compares two elements by finding their position in the Binary-Search Tree. It does this by finding the alphabetical position of each element and once one comparison is not equal, the node moves left or right accordingly (left for less, right for more). Once the two words are found, the method then finds the dot product of the two embeddings. In addition, another method is called to find the magnitudes of the word’s embeddings. From here the cosine distance is returned using the formula Dr. Fuentes shared with us.

**Time Taken To Find Similarities**

This simply puts a time.time() at the start of the for loop and one at the end and subtracts the second one from the first.

**Hash-Table**

**Example Output**

What type of structure would you like?:

1. Binary Tree

2. Hash Table

Choice: 2

Creating Hash Table...

Hash Table Stats:

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

Intial Table Size: 11

Final Table Size: 393215

Load Factor: 0.8318375443459685

Percent of Empty Lists: 99.9173480157166 %

Standard Deviation of Length of Lists: 142.04472410698668

Time Taken To Build Hash Table: 33.11059260368347 Seconds

Reading Word File To Determine Similarities...

Word Similarities Found:

1 : ['keyboard', 'keyboard'] = 1.0000

2 : ['keyboard', 'computer'] = 0.5768

3 : ['keyboard', 'music'] = 0.5730

4 : ['remote', 'television'] = 0.3916

5 : ['remote', 'isolated'] = 0.7149

6 : ['remote', 'sofa'] = 0.1483

7 : ['ant', 'shopping'] = 0.1597

8 : ['ant', 'beetle'] = 0.6550

9 : ['ant', 'alarm'] = 0.1792

10 : ['computer', 'data'] = 0.7605

11 : ['computer', 'strap'] = 0.1247

12 : ['computer', 'platypus'] = -0.1277

13 : ['communism', 'stalin'] = 0.7117

14 : ['communism', 'reagan'] = 0.4582

15 : ['communism', 'toaster'] = 0.0559

16 : ['key', 'lock'] = 0.4309

17 : ['key', 'piano'] = 0.1862

18 : ['key', 'heart'] = 0.4214

19 : ['hole', 'fish'] = 0.4219

20 : ['hole', 'wall'] = 0.6157

21 : ['hole', 'window'] = 0.4941

22 : ['book', 'knowledge'] = 0.5801

23 : ['book', 'love'] = 0.7045

24 : ['book', 'camera'] = 0.3806

25 : ['pop', 'knuckle'] = 0.0899

26 : ['pop', 'britney'] = 0.5369

27 : ['pop', 'queue'] = -0.0207

28 : ['studio', 'magic'] = 0.4636

29 : ['studio', 'adidas'] = 0.1150

30 : ['studio', 'record'] = 0.4240

31 : ['engine', 'computer'] = 0.5230

32 : ['engine', 'car'] = 0.6771

33 : ['engine', 'camera'] = 0.4495

34 : ['opine', 'scowl'] = 0.1590

35 : ['hate', 'sisters'] = 0.2842

36 : ['bath', 'frame'] = 0.3244

37 : ['father', 'insert'] = 0.0081

38 : ['scorch', 'crooked'] = -0.0417

39 : ['offer', 'glove'] = 0.0498

40 : ['pay', 'flash'] = 0.1875

41 : ['canvass', 'spy'] = -0.3449

42 : ['wind', 'special'] = 0.2983

43 : ['name', 'groomed'] = 0.0441

44 : ['slim', 'scare'] = 0.0103

45 : ['balance', 'unbiased'] = 0.2617

46 : ['snatch', 'beat'] = 0.4500

47 : ['answer', 'slope'] = 0.1485

48 : ['scarf', 'ethereal'] = 0.2106

49 : ['condemned', 'cheer'] = 0.1387

50 : ['stitch', 'power'] = -0.0451

51 : ['savory', 'celery'] = 0.5940

52 : ['announce', 'airplane'] = 0.2902

53 : ['replace', 'hat'] = 0.2984

54 : ['reach', 'rule'] = 0.4308

55 : ['cheer', 'nifty'] = 0.2005

56 : ['salute', 'line'] = 0.1966

57 : ['comparison', 'wry'] = 0.2577

58 : ['spot', 'female'] = 0.3676

59 : ['spot', 'weights'] = 0.3185

60 : ['spot', 'partner'] = 0.3342

61 : ['spiky', 'rat'] = 0.2862

62 : ['futuristic', 'shock'] = 0.1602

63 : ['beginner', 'grain'] = 0.0669

64 : ['labored', 'girls'] = 0.0479

65 : ['drunk', 'bite'] = 0.3759

66 : ['manage', 'future'] = 0.7155

67 : ['pour', 'education'] = 0.1403

68 : ['holiday', 'force'] = 0.2302

69 : ['spy', 'shock'] = 0.2326

70 : ['mature', 'poison'] = 0.2573

71 : ['position', 'burn'] = 0.1873

72 : ['pushy', 'toad'] = 0.0967

73 : ['slope', 'leg'] = 0.3546

74 : ['vanish', 'north'] = -0.1118

75 : ['modify', 'hew'] = 0.2264

76 : ['position', 'plucky'] = -0.0461

77 : ['poised', 'spot'] = 0.5438

78 : ['purify', 'tendency'] = 0.1341

79 : ['knit', 'recess'] = -0.0641

80 : ['spend', 'sprout'] = 0.2464

81 : ['blush', 'spin'] = -0.0088

82 : ['class', 'smoke'] = 0.2048

83 : ['harmonious', 'transform'] = 0.5328

84 : ['address', 'chickens'] = 0.0814

85 : ['busy', 'colossal'] = 0.0562

86 : ['quickest', 'quickest'] = 1.0000

87 : ['prepare', 'blush'] = -0.0128

88 : ['sprout', 'smoke'] = 0.2358

89 : ['class', 'spin'] = 0.3252

90 : ['love', 'phone'] = 0.3889

91 : ['prepare', 'shame'] = 0.1990

92 : ['busy', 'harmonious'] = 0.2520

93 : ['transform', 'address'] = 0.3256

94 : ['chickens', 'forgetful'] = 0.0221

95 : ['lazy', 'afterthought'] = 0.3887

96 : ['utopian', 'fly'] = -0.1208

97 : ['resist', 'sail'] = 0.3016

98 : ['behavior', 'cease'] = 0.2517

99 : ['quack', 'sapien'] = 0.0300

100 : ['belligerent', 'lick'] = -0.1649

101 : ['work', 'wave'] = 0.4549

102 : ['seem', 'rub'] = 0.2795

103 : ['hand', 'six'] = 0.6449

104 : ['jellyfish', 'knee'] = -0.0101

105 : ['precious', 'quickest'] = 0.0605

106 : ['ten', 'spin'] = 0.3164

107 : ['class', 'colossal'] = 0.0822

108 : ['prepare', 'tell'] = 0.5782

109 : ['jellyfish', 'six'] = -0.0687

110 : ['chance', 'rattle'] = 0.2189

111 : ['hydrant', 'shame'] = -0.0292

112 : ['busy', 'smoke'] = 0.3902

113 : ['class', 'spin'] = 0.3252

114 : ['extend', 'unit'] = 0.3683

115 : ['stranger', 'momentous'] = 0.2470

116 : ['dad', 'different'] = 0.2794

117 : ['groovy', 'spark'] = -0.0178

118 : ['quiet', 'destroy'] = 0.2012

119 : ['familiar', 'talk'] = 0.7070

120 : ['wonder', 'dream'] = 0.8020

121 : ['oval', 'sleep'] = 0.0621

122 : ['arithmetic', 'enjoy'] = 0.0998

123 : ['marvelous', 'subscribe'] = 0.1014

124 : ['unaccountable', 'forecast'] = -0.1896

125 : ['tan', 'female'] = 0.3061

126 : ['full', 'ticket'] = 0.4987

127 : ['lake', 'remove'] = 0.1731

128 : ['boiling', 'snatch'] = 0.1840

129 : ['yawn', 'shit'] = 0.5201

130 : ['creature', 'contend'] = 0.0676

131 : ['forsake', 'business'] = -0.0443

132 : ['night', 'board'] = 0.4584

133 : ['fit', 'believe'] = 0.6197

134 : ['furniture', 'control'] = 0.2184

135 : ['distribute', 'prohibit'] = 0.5235

136 : ['robin', 'harsh'] = 0.0140

137 : ['conspire', 'persuade'] = 0.3794

138 : ['old', 'encouraging'] = 0.2056

139 : ['cart', 'dire'] = -0.0982

140 : ['water', 'give'] = 0.5000

141 : ['dye', 'add'] = 0.2638

142 : ['awake', 'scent'] = 0.1345

143 : ['burn', 'wound'] = 0.4740

144 : ['cheerful', 'clammy'] = 0.3751

145 : ['unruly', 'mew'] = -0.0386

146 : ['operate', 'fan'] = 0.2246

147 : ['coal', 'surpass'] = 0.2324

148 : ['door', 'balloon'] = 0.3650

149 : ['worship', 'limping'] = -0.1044

150 : ['fall', 'sentence'] = 0.3434

Time Taken To Find 60 Similarites: 0.19785499572753906 Seconds

In this portion of the lab, a Hash-Table is to be created using the words in the word and embedding list. The method called crtHT creates a Hash-Table from the elements given in the word and embeddings list in the list element [0]. It starts by declaring an empty hash table of size 11, this allows for the hash value to be a prime number. The method then runs a loop traversing the length of the word and embeddings list, inserting each word into a correct bucket. It does this by using the InsertC method given to us by Dr. Fuentes in the Hash-Table code, which calls a method that determines what bucket the element is supposed to be placed into. After that the number of items in the Hash-Table is increased by one. Once the number of items in the Hash-Table is equal to the size of the Hash-Table, this means the load factor is equal to one. From there the size of the Hash-Table is increased by multiplying it by two and adding one to that. This keeps the list size a prime number.

I did run into some problems in the number of empty lists being absurdly high and the standard deviation of the lists being very high too. What this means is that though the length of the size of the Hash-Table was being increased, many of the new indices in the lists were redundant. I would pin-point that the issue lied in the h function given to us by Dr. Fuentes, which would determine which bucket the word would be inserted into though I am not sure how to fix it. I tried things like adding all ord elements of the word and finding the modulo of that, though it would end up not inserting all items of the words into the list. I also tried changing the variable n in the h function when multiplying it times r, but doing that messed up the whole program leaving the find function unable to work for some reason.

I also display several stats of the Hash-Table:

**Initial Table Size**

The initial table size was hard coded to always be eleven as this is what the table size is always initialized to when first calling the crtHT method which creates the Hash Table.

**Final Table Size**

The final table size is received by the len(Hash-Table) that is initialized as hTable in the program main area which contains all the word and embedding indices.

**Load Factor**

The load factor is received from a method named LoadFac that sums all of the elements that lie in the list, then divides this by the size of the Hash-Table.

**Percent Of Empty Lists**

This is received by a method named perEmpty which iterates through all buckets in the Hash-Table given and stores plus one for every element whose length is equal to zero. From there the number is multiplied by one hundred and divided by the size of the Hash-Table, returning the percent of empty lists.

**Standard Deviation**

This is given by a method named standDevH that takes a Hash-Table and finds the average of the length of the items squared then returns the sqaure root of this average, which returns the Standard Deviation.

**Time Taken To Create Hash-Table**

This method just takes the time.time() at the beginning of the creation of the Hash-Table and subtracts that from the time.time() at the end of the creation of the Hash-Table.

**Comparing Similarities**

This method is run through a for loop that spans the length of the file that contains all the words that are chosen to be compared. It is the same as the Binary-Search Tree as it finds the two words and their embeddings, calculates their dot product and each magnitude and returns a number that resembles the formula given in the Lab sheet.

**Time Taken To Find Similarities**

This simply puts a time.time() at the start of the for loop and one at the end and subtracts the second one from the first.

**Creation Run-Times (Seconds)**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test 1** | **Test 2** | **Test 3** |
| **Binary Tree Creation** | 17.325868606567383 | 25.732795000076294 | 23.01021957397461 |
| **Hash Table Creation** | 19.184810400009155 | 17.450879335403442 | 16.39420986175537 |

**Word Search Processing 150 Items (Seconds)**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test 1** | **Test 2** | **Test 3** |
| **Binary Tree** | .02997422218322754 | .04278039932250976 | .02404165267944336 |
| **Hash Table** | .10058283805847168 | .14780235290527344 | .12822198867797852 |

**Word Search Processing 300 Items (Seconds)**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test 1** | **Test 2** | **Test 3** |
| **Binary Tree** | .12040472030639648 | .11874532699584961 | .11184978485107422 |
| **Hash Table** | .3732333183288574 | .3890531063079834 | .3939049243927002 |

From the above data collected for the run-times of the two different algorithms, we can conclude that the Binary-Search Tree’s creation didn’t remain a nearly identical runtime. And the Hash Table’s creation seemed to remain nearly the same with each test of the creation. This is due to the properties of each data structure and shows in terms of creation a Hash-Table is more efficient.

As we see in the tests of the items word processing, the Binary-Search Tree’s run-time is significantly better than the Hash-Table's. This is due to the difference in Big(O) notations of the structures. Though the Binary Search Trees could be inconsistent due to wherever the word may be in the list, the Hash-Table's for the most part, will be consistent.

The O run-time of the Binary Tree’s word-processing would-be O(logn) due to the list

always being cut in two each time a new node is added or searched for in the list. And the O run-time of the Hash-Table's word-processing would be something close to constant as we can see the number in the run-time does not vary too heavily in the results.

Over all it was interesting to work with the two different data structures and compare their efficiencies. This lab helped me to understand Has-Tables better and I felt it very helpful.

#CS2302

#Tyler Salas

#Lab4

#Dr.Fuentes

#Anindita Nath

#Create and implement different dadta structures to store words and their embeddings

import numpy as np

import math

import time

class BST(object):

# Constructor

def \_\_init\_\_(self, item, left=None, right=None):

self.item = item

self.left = left

self.right = right

def Insert(T,newItem):

if T == None:

T = BST(newItem)

elif T.item > newItem:

T.left = Insert(T.left,newItem)

else:

T.right = Insert(T.right,newItem)

return T

#Returns position in alphabet

def char\_position(letter):

return ord(letter) - 97

#Returns true if item is to the left and false if it is to the right

def compElem(word1,word2):

if word1 > word2:

return True

return False

#Converts List of sorted words into A Tree

def listToTree(T,A):

if len(A) == 0:

return None

mid = len(A)//2

if T is None:

head = BST(A[mid])

head.left = listToTree(T,A[:mid])

head.right = listToTree(T,A[mid+1:])

return head

#Returns the number of nodes in Given Tree

def numNodes(T):

if T is None:

return 0

return 1 + numNodes(T.left) + numNodes(T.right)

#Returns Hieght of Given Tree

def getHeight(T):

if T is None:

return 0

else:

leftHeight = getHeight(T.left)

rightHeight = getHeight(T.right)

if leftHeight > rightHeight :

return leftHeight+1

else:

return rightHeight+1

#Finds A Node In A BST

def findB(T,k):

if T is None:

return -1

if T.item[0] == k:

return T.item

cur = T.item[0]

if compElem(cur,k):

return findB(T.left,k)

return findB(T.right,k)

#Calculates The Similarites Of Two Words In A Binary Search Tree

def compSim(T,word1,word2):

#Declaring Two words To Work With

word1 = findB(T,word1)

word2 = findB(T,word2)

#Finding Dot Product of Word Embeddings

dp = dotProduct(word1,word2)

#Finding Magnitudes of two words

mag1 = Magnitude(word1)

mag2 = Magnitude(word2)

denom = mag1 \* mag2

return dp/denom

#Gets The Dot Product of Two Words from a Binary-Tree

def dotProduct(word1,word2):

dp = 0

for i in range(len(word1[1])):

dp += word1[1][i] \* word2[1][i]

return dp

#Calculates the Magnitude Of A Word From A Binary-Tree

def Magnitude(word):

mag = 0

for i in range(len(word[1])):

mag += word[1][i] \* word[1][i]

return math.sqrt(mag)

#Functions concerning hash tables

#------------------------------------------------------------------------------

class HashTableC(object):

# Builds a hash table of size 'size'

# Item is a list of (initially empty) lists

# Constructor

def \_\_init\_\_(self,size,num\_Items):

self.item = []

self.size = size

self.num\_Items = 0

for i in range(size):

self.item.append([])

def NumItems(H):

count = 0

for i in range(len(H.item)):

count += len(H.item[i])

return count

def LoadFac(H):

count = 0

for i in range(len(H.item)):

count += len(H.item[i])

num\_Items = count

return num\_Items/len(H.item)

def InsertC(H,k,l,e):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

b = h(k,len(H.item))

H.item[b].append([e])

def FindC(H,k):

# Returns bucket (b) and index (i)

# If k is not in table, i == -1

b = h(k,len(H.item))

for i in range(len(H.item[b])):

if H.item[b][i][0][0] == k:

return H.item[b][i]

return -1

def h(s,n):

r = 0

for c in s:

r = (r\*n + ord(c)) % n

return r

#Turns list with words and embeddings into a Hash-Table

def crtHT(A):

H = HashTableC(11,0)

for i in range(len(A)):

#inserting elements into Hash-Table

elem = [A[i][0],A[i][1]]

InsertC(H,elem[0],len(elem[0]),elem)

H.num\_Items += 1

#Checks if load factor is = 1 and if so makes list size larger by \*2+1

if(H.num\_Items == H.size):

for i in range(H.size+1):

H.item.append([])

H.size = (H.size\*2)+1

return H

#Gets The Dot Product of Two Words in A Hash-Table

def dotProductH(word1,word2):

dp = 0

for i in range(len(word1[1])):

dp += word1[1][i] \* word2[1][i]

return dp

#Calculates the Magnitude Of A Word In A Hash-Table

def MagnitudeH(word):

mag = 0

for i in range(len(word[1])):

mag += word[1][i] \* word[1][i]

return math.sqrt(mag)

#Compares Similarities In A Hash-Table

def compSimH(H,word1,word2):

#Declaring Two Words

word1 = FindC(H,word1)[0]

word2 = FindC(H,word2)[0]

#FInding Dot Product Of Words Embeddings

dp = dotProductH(word1,word2)

#Returning Magnitudes of Words

mag1 = MagnitudeH(word1)

mag2 = MagnitudeH(word2)

denom = mag1 \* mag2

return dp/denom

#Returns the standard deviation in a Hash-Table

def standDevH(H):

count = 0

#Summing length of lists

for i in H.item:

count += len(i)

avg = count/len(H.item)

count = 0

#Squaring Lengths

for i in H.item:

count += (len(i) - avg)\*(len(i)-avg)

avg = count/len(H.item)

return math.sqrt(avg)

#Returns percent of empty lists

def perEmpty(H):

count = 0

for i in H.item:

if len(i) == 0:

count += 1

return (count\*100)/len(H.item)

#-------------------------------------------------------------------------------

#Functions that read the file and make them into nodes

#Converts the given file to an array of each line

def fileToArray(filename):

file = open(filename, encoding="utf8")

A = file.readlines()

file.close

return A

#Splits the elements of the string list into individuals

def arrSplit(A):

B = []

for i in range(len(A)):

sp = A[i].split()

B.append(sp)

return B

#Creates a list with the string word element in one field (Word) and an float array in the second (Embedding)

def wrdEmb(A):

B = []

for i in range(len(A)):

if A[i][0].isalpha():

#Putting float elements in one list

ls = np.array(A[i][1:])

#Changing the elements from strings to float points

lsr = ls.astype(np.float)

lis = [A[i][0],lsr]

B.append(lis)

return B

#------------------------------------------------------------------------------

print()

print("What type of structure would you like?: ")

print("1. Binary Tree ")

print("2. Hash Table ")

txt = input("Choice: ")

txt = int(txt)

if txt == 1:

words = fileToArray('Lab5 words.txt')

words = arrSplit(words)

print()

print("Building Binary Search Tree...")

print()

start = time.time()

T = None

filename = 'glove.6B.50d.txt'

inArr = fileToArray(filename)

splitArr = arrSplit(inArr)

wrdAndEm = wrdEmb(splitArr)

#Sorting words from txt file

wrdAndEm.sort()

T = listToTree(T,wrdAndEm)

end = time.time()

#Stats of the BST

print("BST Stats:")

print("----------------------------------------------------------------------")

print("Number of Nodes: ",numNodes(T))

print("Height Of Binary Tree: ",getHeight(T))

print("Time Taken To Build BST: ",end-start, " Seconds")

print()

print("Reading Word File To Determine Similarities...")

print()

print("Word Similarities Found:")

start = time.time()

#Printing Words and Displaying Similarities

for i in range(len(words)):

print(i+1,': ',words[i],end='')

print(" = ",end='')

print("{0:.4f}".format(compSim(T,words[i][0],words[i][1])))

end = time.time()

print()

print("Time Taken To Find 60 Similarites: ",abs(start-end), " Seconds")

elif txt == 2:

words = fileToArray('Lab5 words.txt')

words = arrSplit(words)

print()

print("Creating Hash Table...")

print()

start = time.time()

filename = 'glove.6B.50d.txt'

inArr = fileToArray(filename)

splitArr = arrSplit(inArr)

wrdAndEm = wrdEmb(splitArr)

hTable = crtHT(wrdAndEm)

end = time.time()

#Stats Of The Hash-Table

print("Hash Table Stats: ")

print("------------------------------------------------------------------")

print("Intial Table Size: ", 11)

print("Final Table Size: ", len(hTable.item))

print("Load Factor: ",LoadFac(hTable))

print("Percent of Empty Lists: ", perEmpty(hTable),'%')

print("Standard Deviation of Length of Lists: ",standDevH(hTable))

print("Time Taken To Build Hash Table: ",abs(start-end)," Seconds")

print()

print("Reading Word File To Determine Similarities...")

print()

print("Word Similarities Found:")

start = time.time()

#Displaying Words and Finding Similarities

for i in range(len(words)):

print(i+1,': ',words[i],end='')

print(" = ",end='')

print("{0:.4f}".format(compSimH(hTable,words[i][0],words[i][1])))

end = time.time()

print()

print("Time Taken To Find 150 Similarites: ",abs(start-end), " Seconds")

else:

print("Invalid Number")

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.



-Tyler Salas