

CS 2302 Data Structures

Fall 2019

Lab Report #5

Due: November 5th, 2019

Professor: Olac Fuentes

TA: Anindita Nath

Introduction

For this lab we were tasked with creating a Hash Table using Linear Probing or Chaining depending on what the user would like to use. Then we would give them six choices on how they would like the data to hash in the list. After the file “glove.6B.50d.txt” has been sorted within the chosen list using the chosen hash the lists stats are displayed then all words in the test file are compared for similarity’s and displayed properly.

Proposed Solution Design and Implementation

Part 1:

First, I reused most of the code from lab 4 main.py file and imported two new files “HashTables_LP.py” and “HashTables_Chaining.py”. These two files contain the classes of both Hash tables. Once the Program loads the chosen method of hashing would be added to the creation of the table along with an appropriate size to create the initial table.

Part 2:

For Hash Tables with Linear Probing I first created a while loop to figure out how many lines where in the glove.6B.50d file and used that number plus one to be the length of the hash table. Next it would go through the file line by line and split each line into a word and a list for the 50-word embeddings. It would then check if, “word.isalpha” to check for any special characters in the sting. If the word had no special characters, it would be added to a word embedding class then that would be added to the Hash Table and if not, the method would move on to the next line. Once all the word-embedding classes are added the tables stats are displayed. Then the similarities file is used to find the words embeddings within the hash table and compared using dot product of the two words embeddings divided by the magnitudes of the two words embeddings. They are then displayed along with the running time to fine all of them.

Part 3:

For Hash Tables with Chaining I first created a while loop to figure out how many lines where in the glove.6B.50d file and used that number divided by twenty to be the length of the hash table. Next it would go through the file line by line and split each line into a word and a list for the 50-word embeddings. It would then check if, “word.isalpha” to check for any special characters in the sting. If the word had no special characters, it would be added to a word embedding class then that would be added to the Hash Table and if not, the method would move on to the next line. Once all the word-embedding classes are added the tables stats are displayed. Then the similarities file is used to find the words embeddings within the hash table and compared using dot product of the two words embeddings divided by the magnitudes of the two words embeddings. They are then displayed along with the running time to fine all of them.

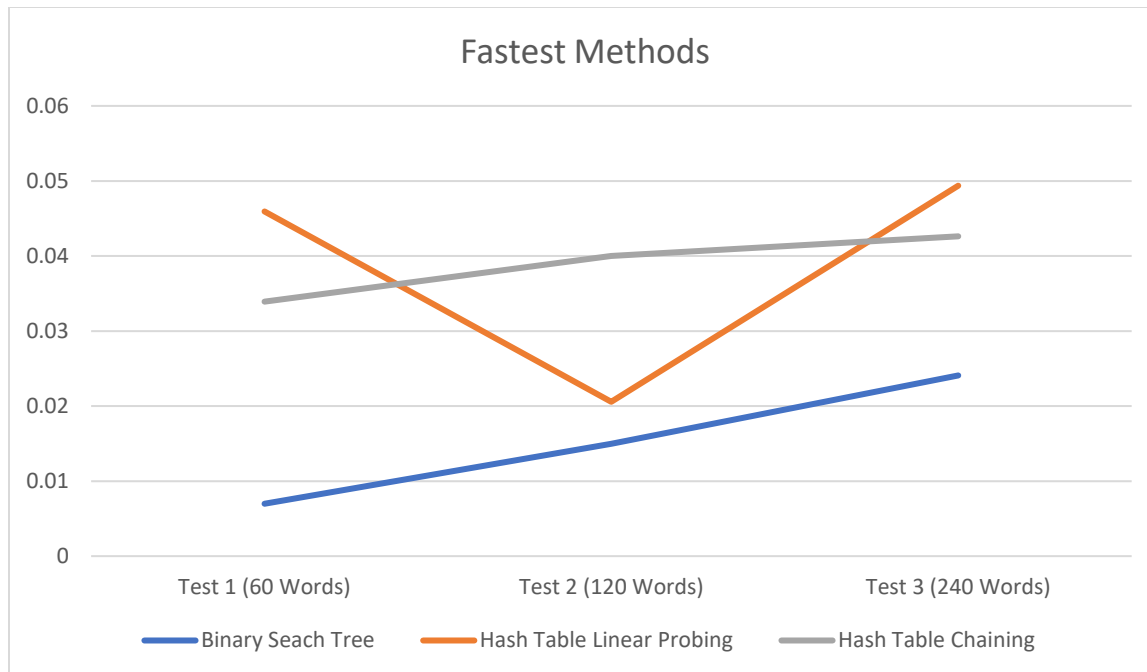
Experimental Results

Hash Tables with Linear Probing:

| Hashing Methods | Test 1 (60 words) | Test 2 (120 words) | Test 3 (240 words) |
|-------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------|-----------------------|-----------------------|
| 1. The length of the string % n (Average time to create table: 6 Hours) | 839.46397 Seconds | 842.14224 Seconds | 876.45831 Seconds |
| 2. The ascii value (ord(c)) of the first character in the string % n (Average time to create table: 2 Hours) | 653.198631 Seconds | 673.416398 Seconds | 690.23659 Seconds |
| 3. The product of the ascii values of the first and last characters in the string % n (Average time to create table: 1.5 Hour) | 371.123642 Seconds | 421.123475 Seconds | 424.123541 Seconds |
| 4. The sum of the ascii values of the characters in the string % n (Average time to create table: 45 Min) | 0.2932154 Seconds | 0.3247586 Seconds | 0.3842327 Seconds |
| 5. The recursive formulation $h("",n) = 1$; $h(S,n) = (\text{ord}(s[0]) + 255 * h(s[1:],n)) \% n$ (Average time to create table: 32 Sec) | 0.0385463 Seconds | 0.0540861 Seconds | 0.0493659 Seconds |
| 6. The ascii value of each letter of the string times 2302 % n (Average time to create table: 1.5 Min) | 0.2349631 Seconds | 0.2314586 Seconds | 0.2546923 Seconds |

Hash Tables with Chaining:

| Hashing Methods | Test 1 (60 words) | Test 2 (120 words) | Test 3 (240 words) |
|-------------------------------------------------------------------------------------------------------------------------------------------------|----------------------|-----------------------|-----------------------|
| 1. The length of the string % n (Average time to create table: 4.5 Min) | | 8.0543294 Seconds | 7.9071414 Seconds |
| 2. The ascii value (ord(c)) of the first character in the string % n (Average time to create table: 2 Min) | 4.1035479 Seconds | 4.0026483 Seconds | 4.3181658 Seconds |
| 3. The product of the ascii values of the first and last characters in the string % n (Average time to create table: 1 Min) | 2.1264592 Seconds | 2.4851369 Seconds | 2.3752146 Seconds |
| 4. The sum of the ascii values of the characters in the string % n (Average time to create table: 18 Sec) | 0.1547962 Seconds | 0.1746324 Seconds | 0.189647 Seconds |
| 5. The recursive formulation $h("",n) = 1$; $h(S,n) = (\text{ord}(s[0]) + 255 * h(s[1:],n)) \% n$ (Average time to create table: 13 Sec) | 0.0155804 Seconds | 0.0237725 Seconds | 0.048619 Seconds |
| 6. The ascii value of each letter of the string times 2302 % n (Average time to create table: 18 Sec) | 0.1296547 Seconds | 0.1322591 Seconds | 0.1913447 Seconds |



Conclusion

In Conclusion the both methods were able to find the similarities in a relatively fast time compared to Btrees but where longer faster than Binary search trees. However, the time taken to create a hash Table of this size could take a massive amount of time if the hash method isn't good. As with the case of using the length of the string divided by n the program took about 6 hours to create the table as compared to the hash using the recursive formulation which created it in 30 seconds. Also the implementation of a very sophisticated hash function could result in the best case of a data search of $O(1)$ so theoretically of applied well a hash function has the potential to beat the run time of a binary search tree.

Appendix

Lab5.py:

```

1  '''
2      Cs2302 Data Structures
3      Issac Rivas
4      Lab 4
5      Dr.Fuentues
6  '''
7  import Binary_Search_Tree as BSTree
8  import BTree
9  import HashTables_Chaining as HTC
10 import HashTables_LP as HTLP
11 import numpy as np
12 import time as t

```

```

13
14 #Creates an object that holds the word as well as 50 element embedding list
15 class WordEmbedding(object):
16     def __init__(self, word, embedding):
17         self.word = word
18         self.emb = np.array(embedding, dtype=np.float32)
19
20 #Splits a text file line into a word and a float list of the remaining 50 elements
21 def lineSplit(line):
22     emb = line.split()
23     word = emb[0]
24     del emb[0]
25     for x in range(0, len(emb)):
26         emb[x] = float(emb[x])
27     return word, emb
28
29 #Calculates for the Similarity of two words
30 def Similarity(word0, word1):
31     dotProduct = np.dot(word0.emb, word1.emb)
32     magnitude = np.linalg.norm(word0.emb) * np.linalg.norm(word1.emb)
33     return dotProduct/magnitude
34
35 if __name__ == "__main__":
36     print("Choose table implementation:")
37     print("Enter the number '1' for a Binary Search Tree")
38     print("'2' for a B-tree")
39     print("'3' for a Hash Table with Linear Probing")
40     print("'4' for a Hash Table with Chaining")
41     ans = int(input("Input: "))
42     while(ans != 1 and ans != 2 and ans != 3 and ans != 4):
43         print()
44         print("Incorrect input entered please Try again.")
45         print("Type 1 for binary search or 2 for B-tree")
46         ans = int(input("Input: "))
47
48     filepath = 'C:/Users/Issac/Desktop/Lab_5/glove.6B.50d.txt'
49     testpath = 'C:/Users/Issac/Desktop/Lab_5/test3.txt'
50
51     #If user wants to create a BST
52     if ans == 1:
53         start = t.time()
54         T = None
55         with open(filepath, encoding="utf8") as fp:
56             for line in fp:
57                 word, emb = lineSplit(line)
58                 if word.isalpha():
59                     temp = WordEmbedding(word, emb)
60                     T = BSTree.Insert(T, temp)
61         Total = t.time() - start
62         print()
63         print("Building Binary Search Tree")
64         print()
65         print("Binary Tree Stats:")
66         print("Number of Nodes: ", BSTree.numOfNodes(T))
67         print("Tree Height is: ", BSTree.treeHeight(T))
68         print("Running time for Binary Search Tree construction:", round(Total, 4), " seconds")
69         print()
70         print("Reading word file to determine similarities")
71         print()
72         print("Word similarities found:")
73         start = t.time()
74         with open(testpath) as tp:
75             for line in tp:
76                 words = line.split()
77                 temp0 = BSTree.Search(T, words[0])
78                 temp1 = BSTree.Search(T, words[1])
79                 sim = Similarity(temp0.data, temp1.data)
80                 print("Similarity [" + words[0] + ", " + words[1] + "] = ", round(sim, 4))
81         Total = t.time() - start
82         print()
83         print("Running time for Binary Search Tree query processing: ", round(Total, 7))

```

```

83         print("Running time for Binary Search Tree query processing: ", round(Total, 7))
84
85     #If user wants to create a B-tree
86     elif ans == 2:
87         max_data = int(input("Please enter the maximum number of items per node: "))
88         while (max_data < 1):
89             print("Error please insert a number equal to or greater then 1.")
90             max_data = int(input("Please enter the maximum number of items per node: "))
91         start = t.time()
92         T = BTree.BTree([], max_data)
93         with open(filepath, encoding="utf8") as fp:
94             for line in fp:
95                 word, emb = lineSplit(line)
96                 if word.isalpha():
97                     temp = WordEmbedding(word, emb)
98                     BTree.Insert(T, temp)
99
100         print()
101         print("Building B-Tree")
102         print()
103         print("B-Tree Stats:")
104         print("Number of Nodes: ", BTree.NumItems(T))
105         print("Tree Height is: ", BTree.Height(T))
106         Total = t.time() - start
107         print("Running time for B-tree construction (with max_items = " + str(max_data) + "):", round(Total, 4), "seconds")
108         print()
109         print("Reading word file to determine similarities")
110         print()
111         print("Word similarities found:")
112         start = t.time()
113         with open(testpath) as tp:
114             for line in tp:
115                 words = line.split()
116                 wordemb1 = WordEmbedding(words[0], [])
117                 wordemb2 = WordEmbedding(words[1], [])
118                 temp0 = BTree.Search(T, wordemb1)
119                 temp1 = BTree.Search(T, wordemb2)
120                 sim = Similarity(temp0, temp1)
121                 print("Similarity [" + words[0] + ", " + words[1] + "] = ", round(sim, 4))
122         Total = t.time() - start
123         print()
124         print("Running time for B-Tree query processing (with max_items = " + str(max_data) + "):", round(Total, 7), "seconds")
125
126     #If the user want to create a Hash Table with Linear Probing
127     else:
128         print("Choose Hash Function you would like to use:")
129         print("Enter the number")
130         print("'1' for The length of the string % n")
131         print("'2' for The ascii value (ord(c)) of the first character in the string % n")
132         print("'3' for The product of the ascii values of the first and last characters in the string % n")
133         print("'4' for The sum of the ascii values of the characters in the string % n")
134         print("'5' for The recursive formulation h(", n) = 1; h(S, n) = (ord(s[0]) + 255*h(s[1:], n))% n")
135         print("'6' for The ascii value of each letter of the string times 2302 % n")
136         ans2 = int(input("Input: "))
137         while (ans2 != 1 and ans2 != 2 and ans2 != 3 and ans2 != 4 and ans2 != 5 and ans2 != 6):
138             print()
139             print("Enter the number")

```

```

139     print("'1' for The length of the string % n")
140     print("'2' for The ascii value (ord(c)) of the first character in the string % n")
141     print("'3' for The product of the ascii values of the first and last characters in the string % n")
142     print("'4' for The sum of the ascii values of the characters in the string % n")
143     print("'5' for The recursive formulation  $h(n) = 1; h(S,n) = (ord(s[0]) + 255 * h(s[1:],n)) \% n$ ")
144     print("'6' for The ascii value of each letter of the string times 2302 % n")
145     ans2 = int(input("Input: "))
146     if ans == 3:
147         start = t.time()
148         count = 0
149         with open(filepath, encoding="utf8") as fp:
150             for line in fp:
151                 count += 1
152         T = HTLP.HashTableLP(int(count+1), ans2)
153         counter = 0
154         with open(filepath, encoding="utf8") as fp:
155             for line in fp:
156                 word, emb = lineSplit(line)
157                 if word.isalpha():
158                     temp = WordEmbedding(word, emb)
159
160                     #If the used buckets/Unused is grteater then 0.8 precent resize the bucket
161                     if (counter / len(T.item)) >= 0.95:
162                         T = HTLP.resize(T)
163                     HTLP.insert(T, temp)
164                     counter += 1
165         Total = t.time() - start
166         print()
167         print("Building Hash Table with Linear Probing")
168         print()
169         print("Hash Table Stats:")
170         print("Length of the Hash Table with Linear Probing: ", len(T.item))
171         print("Number of items in the Hash Table with Linear Probing: ", counter)
172         print("Running time for Hash Table with Linera Probing construction:", round(Total, 4), " seconds")
173         print()
174         print("Reading word file to determine similarities")
175         print()
176         print("Word similarities found:")
177         start = t.time()
178         with open(testpath) as tp:
179             for line in tp:
180                 words = line.split()
181                 w1 = WordEmbedding(words[0], [])
182                 w2 = WordEmbedding(words[1], [])
183                 temp0 = HTLP.find(T, w1)
184                 temp1 = HTLP.find(T, w2)
185                 sim = Similarity(T.item[temp0], T.item[temp1])
186                 print("Similarity [" + words[0] + ", " + words[1] + "] = ", round(sim, 4))
187         Total = t.time() - start
188         print()
189         print("Running time for Hash Table with Linear Probing query processing: ", round(Total, 7))
190
191     #If the user want to create a Hash Table with Chaining
192     elif ans == 4:
193         start = t.time()
194         count = 0

```

```

195         with open(filepath, encoding="utf8") as fp:
196             for line in fp:
197                 count += 1
198         T = HTC.HashTableChain(int(count/20), ans2)
199         counter = 0
200         with open(filepath, encoding="utf8") as fp:
201             for line in fp:
202                 word, emb = lineSplit(line)
203                 if word.isalpha():
204                     temp = WordEmbedding(word, emb)
205                     HTC.insert(T, temp)
206                     counter += 1
207         Total = t.time() - start
208         print()
209         print("Building Hash Table with Chaining")
210         print()
211         print("Hash Table Stats:")
212         print("Length of the Hash Table with Chaining: ", len(T.bucket))
213         print("Number of items in the Hash Table with Chaining: ", counter)
214         print("Running time for Hash Table with Chaining construction:", round(Total, 4), " seconds")
215         print()
216         print("Reading word file to determine similarities")
217         print()
218         print("Word similarities found:")
219         start = t.time()
220         with open(testpath) as tp:
221             for line in tp:
222                 words = line.split()
223                 W1 = WordEmbedding(words[0], [])
224                 W2 = WordEmbedding(words[1], [])
225                 temp0, ind0 = HTC.find(T, W1)
226                 temp1, ind1 = HTC.find(T, W2)
227                 sim = Similarity(T.bucket[temp0][ind0], T.bucket[temp1][ind1])
228                 print("Similarity [" + words[0] + ", " + words[1] + "] = ", round(sim, 4))
229         Total = t.time() - start
230         print()
231         print("Running time for Hash Table with Chaining query processing: ", round(Total, 7))

```

HasTables_LP:

```

1  import numpy as np
2
3  class HashTableLP(object):
4      def __init__(self, size, hashNum):
5          self.item = np.zeros(size, dtype=np.object)-1
6          self.hashNum = hashNum
7
8      def insert(self, k):
9          for i in range(len(self.item)):
10             pos = h(self, k)+i
11             if pos >= len(self.item):
12                 pos = pos - len(self.item)
13             if self.item[pos] == -1:
14                 self.item[pos] = k
15             return pos
16         return -1

```



```

17
18 def find(self,k):
19     for i in range(len(self.item)):
20         pos = h(self, k)+i
21         if pos >= len(self.item):
22             pos = pos - len(self.item)
23         if self.item[pos] == -1:
24             return -1
25         if self.item[pos].word == k.word:
26             return pos
27     return -1
28
29 def delete(self,k):
30     f = self.find(k)
31     if f >=0:
32         self.item[f] = -2
33     return f
34
35 def h(self,k):
36     if self.hashNum == 1:
37         return ((len(k.word)-1)) % len(self.item)
38     elif self.hashNum == 2:
39         return ord(k.word[0]) % len(self.item)
40     elif self.hashNum == 3:
41         return (ord(k.word[0])+ord(k.word[-1])) % len(self.item)
42     elif self.hashNum == 4:
43         sum = 0
44         for x in range(len(k.word)):
45             sum = sum + ord(k.word[x])
46         if sum > len(self.item):
47             sum = sum - len(self.item)
48         return (sum % len(self.item))
49     elif self.hashNum == 5:
50         return recursiveForm(self, len(self.item), k.word)
51     else:
52         mx = 0
53         for x in range(len(k.word)):
54             temp = ord(k.word[x])
55             mx += temp * 2302
56         return mx % len(self.item)
57
58 def recursiveForm(self, n, s):
59     if s == '':
60         return 1
61     else:
62         return ((ord(s[0]) + 255*recursiveForm(self, n, s[1:])) % n)
63
64 def print_table(self):
65     print('Table contents:')
66     print(self.item)
67
68 def n(self, k):
69     temp = []
70     hash = self.h(k)
71     for b in self.item:
72         if self.h(b) == hash:
73             temp.append(b)
74     if len(temp) < 1:
75         return -1
76     else:
77         temp.sort()
78         return temp[len(temp)-1]
79
80 def resize(H):
81     new = HashTableLP(len(H.item)*2, H.hashNum)
82     for x in range(len(H.item)):
83         if H.item[x] != -1:
84             insert(new, H.item[x])
85     return new

```

HashTables_Chaining:

```
1 class HashTableChain(object):
2     def __init__(self, size, hashNum):
3         self.bucket = [[] for i in range(size)]
4         self.hashNum = hashNum
5
6     def insert(T, k):
7         b = h(T, k)
8         if len(T.bucket[b]) >= len(T.bucket)*3:
9             T = resize(T)
10        if not k in T.bucket[b]:
11            T.bucket[b].append(k)
12
13
14    def find(T, k):
15        b = h(T, k)
16        i = -1
17        for x in range(len(T.bucket[b])):
18            if T.bucket[b][x].word == k.word:
19                i = x
20        return b, i
21
22    def print_table(T):
23        print('Table contents:')
24        for b in T.bucket:
25            print(b)
26
27    def delete(T, k):
28        b = T.h(k)
29        try:
30            T.bucket[b].remove(k)
31            return 1
32        except:
33            return -1
34
35    def resize(H):
36        new = HashTableChain(len(H.bucket)*2, H.hashNum)
37        for x in range(len(H.bucket)):
38            if H.bucket[x] != None:
39                for y in range(len(H.bucket[x])):
40                    insert(new, H.bucket[x][y])
41        return new
42
43    def h(self, k):
44        if self.hashNum == 1:
45            return ((len(k.word)-1)) % len(self.bucket)
46        elif self.hashNum == 2:
47            return ord(k.word[0]) % len(self.bucket)
48        elif self.hashNum == 3:
49            return (ord(k.word[0])+ord(k.word[-1])) % len(self.bucket)
50        elif self.hashNum == 4:
51            sum = 0
52            for x in range(len(k.word)):
53                sum = sum + ord(k.word[x])
54            return sum % len(self.bucket)
55        elif self.hashNum == 5:
56            return recursiveForm(self, len(self.bucket), k.word)
57        else:
58            mx = 0
59            for x in range(len(k.word)):
60                temp = ord(k.word[x])
```

```
61         mx += temp * 2302
62         return mx % len(self.bucket)
63
64     def recursiveForm(self, n, s):
65         if s == '':
66             return 1
67         else:
68             return (ord(s[0]) + 255*recursiveForm(self, n, s[1:])) % n
```

I Issac Rivas, certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, preformed 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.