**Course Name:** 2302 **Author:** Olugbenga Iyiola **ID:** 80638542 **Instructor:** Olac Fuentes **TA:** Nath Anindita/ Malileh Zargaran **LAB #5 Report**

**Introduction**

The purpose of this lab is to compare the running times of two implementations of tables to retrieve word embeddings to enable the (hopefully fast) comparison of two given words. One of the table implementations uses a binary search tree and the other should uses a hash table with chaining. Word embeddings are a recent advance in Natural Language Processing (NLP) that consists of representing words by vectors (or arrays) of floating-point numbers in such a way that if two words are similar, their embeddings are also similar.

A hash table (hash map) is a [data structure](https://en.wikipedia.org/wiki/Data_structure) that implements an [associative array](https://en.wikipedia.org/wiki/Associative_array) [abstract data type](https://en.wikipedia.org/wiki/Abstract_data_type), a structure that can map [keys](https://en.wikipedia.org/wiki/Unique_key) to [values](https://en.wikipedia.org/wiki/Value_(computer_science)). A hash table uses a [hash function](https://en.wikipedia.org/wiki/Hash_function) to compute an index into an array of buckets or slots, from which the desired value can be found. On the other hand, a binary search tree (BST), also known as an ordered binary tree, is a node-based data structure in which each node has no more than two child nodes. Each child must either be a leaf node or the root of another binary search tree. In many situations, hash tables turn out to be on average more efficient than [search trees](https://en.wikipedia.org/wiki/Search_tree) or any other [table](https://en.wikipedia.org/wiki/Table_(computing)) lookup structure. (Wikipedia, 2019)

**Proposed Solution Design and Implementation**

The Hash table and BST are implemented with python native list with the various operations of the Hash table carried out in O(1) time and those of the BST in O(n) time as expected and the pseudocodes for the various implementations are given below;

**Pseudocode to read text file, create and return table of an word embed object**

*Read text file*

*for line in f: # Loop to traversal*

*Remove the space character after each read line*

*Create array from text file with a delimiter*

*The first element of the array is the word*

*Create numpy array for the embeddings*

*for i in range of numpy array:*

*Append the embeddings*

*Create an object that has word and array as items*

*Append word object to table.*

*return Table*

**Pseudocode to construct binary search tree using word object form table**

*if root is none:*

*Create root if tree is empty*

*elif item of root> item to be inserted:*

*Insert as left child if current item > new item*

*elif item of root < item to be inserted:*

*Insert ad right child if current item < new item*

*else:*

*Do not insert if item is already in tree*

**Pseudocode to construct hash table using word object form table**

*# Inserts k in appropriate bucket (list)*

*Check value of load factor*

*if check > 1:*

*Create new hash table with double size*

*Get new buckets for old items of hash table*

*Append the old items to the new hash table*

*Increment number of items for every item inserted in hash table*

*Append the new item to the new hash table*

*Else:*

*Get bucket for item*

*Append item in the bucket of the hash table*

*Increment number of items for every item inserted in hash table*

*return Hash table*

**Pseudocode to read new text file, find words embeddings and calculate similarity between 2 words**

*Read text file*

*for line in file:*

*Remove the space character after each read line*

*Create array from text file with a delimiter*

*Create variable with line's first word*

*Create variable with line's second word*

*Get first word’s embedding*

*Get second word’s embedding*

*Find Dot product of the embeddings*

*Find Magnitudes of first and second word’s embeddings*

*Similarities = Dot Product/(Magnitude1\*Magnitude2)*

**Pseudocode for returning total number of nodes in BST**

*if root/node is none:*

*return 0*

*else:*

*return 1+ recursive call on left child of node + recursive call on right child of node*

**Pseudocode to find standard deviation of the length of the hash table** **lists**

*Create list to store length of the lists in the hash table*

*Traverse the hash table*

*Append the table list lengths to the list*

*Compute the standard deviation of the lengths of the lists in the hash table*

**Pseudocode to find height of BST**

*if root/ node is None:*

*return -1;*

*Get height of left tree by making recursive call with node’s left child*

*Get height of right tree by making recursive call with node’s right child*

*Return the greater of right and left trees*

**Pseudocode to find percentage of list that is empty**

*Create counter to count number of positions in table*

*Traverse the hash table*

*Increment counter for every position in the hash*

*Percent Empty = number of items divided capacity of hash table*

**Experimental Result**

System Specification: HP Windows 10, 1.60GHZ Intel® Celeron® , 4.GB RAM, 64-bit operating system

The file glove.6B.50d.txt was too big for my RAM so I only used a part of it for the experiment.

The results of the various test cases using different sizes from the file for each of the algorithms are shown below:

**Creating of BST**

|  |  |
| --- | --- |
| **Number of Nodes(Input)** | **Runtime in nanoseconds** |
| **232** | **423457022** |
| **4021** | **7433361985** |
| **6780** | **11984155643** |
| **18708** | **33555058757** |
| **27592** | **49704872287** |
|  |  |

Recurrence Equation: T(n) = 2T(n/2) + 1 which, using the master theorem, gives us O(log n)

**Finding similarities using BST**

|  |  |
| --- | --- |
| **Number of nodes(Input)** | **Runtime in nanoseconds** |
| **232** | **4937121** |
| **4021** | **4853918** |
| **6780** | **66324725** |
| **18708** | **19739525** |
| **27592** | **4873759** |
|  |  |

Recurrence Equation: T(n) = 2T(n/2) + n which, using the master theorem, gives us O(n)

**Creating Hash Table**

|  |  |
| --- | --- |
| **Number of items(Input)** | **Runtime in Nanoseconds** |
| **232** | **450962559** |
| **4021** | **7119768389** |
| **6780** | **12360886660** |
| **18708** | **34976116980** |
| **25003** | **954831104248** |
|  |  |

This gives us O(n).

**Finding similarities using Hash Table**

|  |  |
| --- | --- |
| **Number of items(Input)** | **Runtime in Nanoseconds** |
| **232** | **4972962** |
| **4021** | **4712474** |
| **6780** | **141466377** |
| **18708** | **18791014** |
| **25003** | **3779963** |
|  |  |

This gives us O( N).

**CONCLUSION**

In summary if you know how many data to maintain and have enough space to store hash table and do not need data to be sorted, hash table is always a good choice. Because, hash table provides constant time operation for insertion, retrieve and deletion. On the other hand, if items will be consistently added, binary search tree’s O(log(n)) operation is acceptable, comparing with rehashing operation during running time.

Besides, if you actually do not know size of input items, but after inserting, most operations are item looking up, hash table is preferred due to constant retrieve time. However, if items are continuously added or removed, tree’s O(log(n)) insertion and deletion time are more suitable in this condition (Brack’s blog).

**Appendix**

***Programmed by Olac Fuentes***

# Implementation of hash tables with chaining using strings

class HashTableC(object):

# Builds a hash table of size 'size'

# Item is a list of (initially empty) lists

# Constructor

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

self.item = []

for i in range(size):

self.item.append([])

def InsertC(H,k,l):

# 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([k,l])

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] == k:

return b, i, H.item[b][i][1]

return b, -1, -1

def h(s,n):

r = 0

for c in s:

r = (r\*255 + ord(c))% n

return r

H = HashTableC(11)

A = ['data','structures','computer','science','university','of','texas','at','el','paso']

for a in A:

InsertC(H,a,len(a))

print(H.item)

for a in A: # Prints bucket, position in bucket, and word length

print(a,FindC(H,a))

**Wikipedia**

[**https://en.wikipedia.org/wiki/Sorting\_algorithm#Comparison\_of\_algorithms**](https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms)

**Academic Dishonesty**

This work was done by me without any act or practice of academic dishonesty

**SIGNATURE**

**OLUGBENGA IYIOLA(OT)**

**……………………………………………….**