**Lab 4 Report**

The main purpose of this lab is first to improve the running time of the previous lab by using a hash-table instead of using a binary search tree. The second problem is that it needs to use chaining in order to solve collisions if any is presented. One of the things I struggle the most at the beginning was to write effective hash tables that will test successfully a result my hash tables were throwing results which were wrong based on the math in my implementation. In addition, with the math the average number of comparisons was wrong at first because the tables were giving wrong results and would have the program be all over the place. However, by breaking down the problem I could implement different functions that actually allowed me to compare the output and running time. I also implemented a table which would allow me to read and store the values and the compared them, looking for any collisions and finally determining the average of the numbers that were compared. What I learned from this lab is that it is actually possible to compare with only 1 access and reduce drastically the running time. Personally, I feel that I still need more practice with this lab and this type of implementations with hash-tables but, was able to pull out the most important and the basic concepts.

# CS2302 Data Structures  
# Programmed by Luis Garcia.  
# Last modified November 21, 2018.  
# Instructor Diego Aguirre.  
# Implementation of hashes in order to compare the time complexity  
#lab 4  
  
class HTNode(object):  
  
 def \_\_init\_\_(self, item, next):  
 self.item = item  
 self.next = next  
  
  
'''HashInsert(hashTable, item) {  
 if (HashSearch(hashTable, item->key) == null) {  
 bucketList = hashTable[Hash(item->key)]  
 node = Allocate new linked list node  
 node->next = null  
 node->data = item  
 ListAppend(bucketList, node)  
 }  
}  
  
HashRemove(hashTable, item) {  
 bucketList = hashTable[Hash(item->key)]  
 itemNode = ListSearch(bucketList, item->key)  
 if (itemNode is not null) {  
 ListRemove(bucketList, itemNode)  
 }   
}  
  
HashSearch(hashTable, key) {  
 bucketList = hashTable[Hash(key)]  
 itemNode = ListSearch(bucketList, key)  
 if (itemNode is not null)  
 return itemNode->data  
 else  
 return null  
}  
'''  
  
  
class hashTable:  
  
 def \_\_init\_\_(self, table\_size):  
 self.size = 0  
 self.table = [None] \* table\_size  
  
 def hashMultiplication(self, word, first\_letter):  
 return (word \* first\_letter) % len(self.table)  
 '''def hashMultiplication(self, word, first\_letter):  
 return (word \*\* first\_letter) % len(self.table)  
 def hashMultiplication(self, word, first\_letter):  
 return (word + first\_letter) % len(self.table)  
 def hashMultiplication(self, word, first\_letter):  
 return (word - first\_letter) % len(self.table)  
'''  
 def stringToASCII(self, word):  
 sum = 0  
 for char in word:  
 sum += ord(char)  
 return sum  
  
# The insert method will insert the word where it corresponds.  
 def insert(self, item):  
 self.size += 1  
 word = self.stringToASCII(item)  
 first\_letter = self.stringToASCII(item[:1])  
 position = self.hashMultiplication(word, first\_letter)  
 self.table[position] = HTNode(item,self.table[position])  
  
  
def comparisons(self):  
 num\_nodes = 0  
 num = 0  
 for i in range(len(self.table)):  
 temp = self.table[i]  
 if temp is not None:  
 num += 1  
 while temp is not None:  
 num\_nodes += 1  
 temp = temp.next  
 print('Number of nodes counted:', num)  
 print('Total number of nodes:', num\_nodes)  
 print('Average:', num\_nodes / num)  
  
  
def load\_factor(self):  
 print('Load Factor:', self.size / len(self.table))  
  
  
class better\_hash\_table:  
 def \_\_init\_\_(self, table\_size):  
 self.size = 0  
 self.table = [None] \* table\_size  
  
 def hash\_1(self, word):  
 init\_size = 5381  
 for char in word:  
 init\_size = (init\_size \* 33) + self.stringToASCII(char)  
 return init\_size % len(self.table)  
  
 def stringToASCII(self, word):  
 sum = 0  
 for char in word:  
 sum += ord(char)  
 return sum  
  
 def insert(self, item):  
 self.size += 1  
 position = self.hash\_1(item)  
 self.table[position] = HTNode(item, self.table[position])  
  
 def load\_factor(self):  
 print('Load Factor:', self.size / len(self.table))  
  
 def comparisons(self):  
 num\_nodes = 0  
 num = 0  
 for i in range(len(self.table)):  
 temp = self.table[i]  
 if temp is not None:  
 num += 1  
 while temp is not None:  
 num\_nodes += 1  
 temp = temp.next  
 print('Number of nodes counted:', num)  
 print('Total number of nodes:', num\_nodes)  
 print('Average:', num\_nodes / num)  
  
  
size = int(input('What is the size of the table?'))  
table = hashTable(size)  
second\_table = better\_hash\_table(size)  
words\_file = open('words.txt', 'r')  
for line in words\_file:  
 table.insert(line)  
 second\_table.insert(line)  
print()  
print('Hashed with the first hash function:')  
table.load\_factor()  
table.comparisons()  
print()  
print('Hashed with the second hash function:')  
second\_table.load\_factor()  
second\_table.comparisons()

“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 class.”