**Lab 4 Report – Hash Tables**

For Lab 4, our objective was to use the English word files used in Lab 3 to populate a Hash Table instead of an AVL Tree or a Red-Black Tree. It required the formulation of a few hash functions in order to compare the average number of comparisons performed by each hash function. Insertion and search were the primary methods use from the Hash Table function class, which was implemented largely based off of Professor Aguirre’s Hash Table class for the second exam (per his permission).

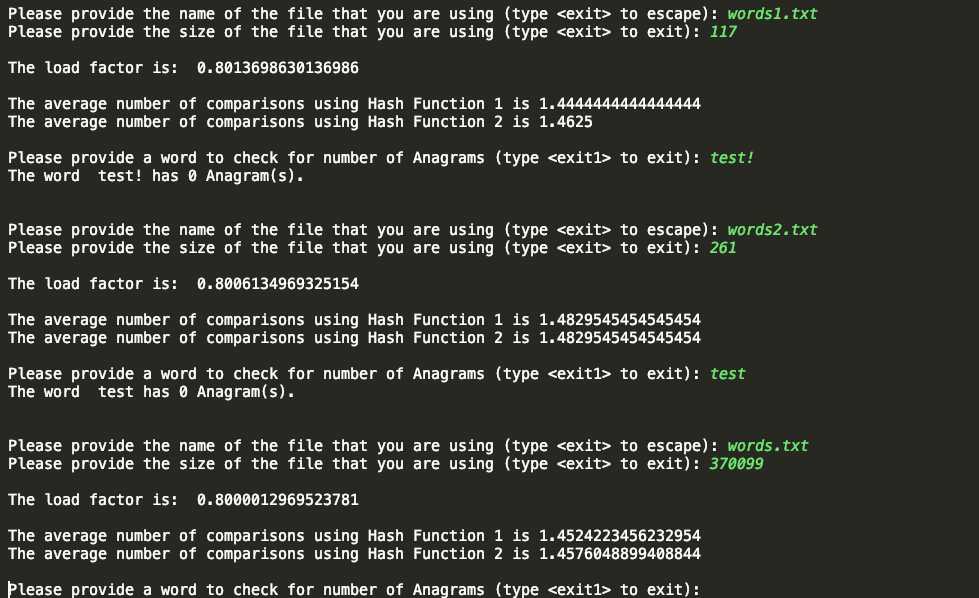
Implementing this lab was relatively simple, since the Hash Table class required very few changes in order to customize it for Lab 4. The only real change was to the hash function(s) and also to the search method, which now returns a Boolean variable as opposed to the memory indexed of the searched node. The first of the two hash functions used obtained the ASCII representation of a given English word, appended each of the letters values to a string. This string was then cast as an integer and finally “modded” by the table size. The second hash function was a tad bit more complicated to implement and it actually provided a slightly higher average number of comparisons.

Each word in the English language was converted to a base-26 number. Each letter’s ASCII value was “standardized,” so to speak, by subtracting the ASCII value of the letter ‘a’ from each ASCII value, there ‘a’ had a true value of 0, ‘b’ of 1, ‘c’ of 2 and so on and so forth. This is done in each iteration of a for-loop that runs in the range of the word’s length. A new variable is created that sums the multiplication of each letter’s “new,” standardized, ASCII value and its respective base to its position (i.e. word = ‘test’, word[0] should be ASCII value of t \* 264). This was tested by making these conversions by hand and comparing expected outputs.

In addition, this lab required the computation of the load factor of the hash table created and also the “average number of comparisons.” The load factor was relatively simple, since a total number of elements count can be easily generated by iterating through the entirety of the hash table, finding total number of elements, and dividing that by the size of the hash table itself. My program was designed to initialize any new hash table created to **1.25\*the user specified size of the file.** Therefore, you ensure enough space for the elements while keeping collisions to a minimum

The average number of collisions was computed by taking the total number of elements and simply diving that by the number of spaces in the hash table that were actually USED. This calculation would actually provide an “upper bound” of sorts for the average number of comparisons, since most elements may not exist that the “extreme” ends of a linked list found in any given hash table bucket. In fact, most elements may be toward the center of the lists. Regardless, a **number\_comparisons** method was created that mimicked the **get\_load\_factor** method, however, a counter was only updated when it was discovered that the “head” of the each hash table index was not **NONE**, ensuring only occupied indices were accounted for. Once these occupied number of indices was obtained, it was divided by the total number of elements and thus, the average number of comparisons is computed.

The program was tested with word files of various sizes. As the following screen shots will show, the load factor remains largely the same for a file of any size, since the table will always be initialized to **1.25\* the size of the file**. This ensures ample space for the elements and **a load factor of around 0.8 most of the times**. As for the average number of comparisons, the first hash function (the one that casts ASCII value strings as integers) has a slightly lower number of comparisons:



S

The program still requests a word to check against the **english\_words** hash table, as the specifics of the lab were unknown.

All in all, the insert and search methods run very closely to O(1). In a worst case scenario, the search method may be O(n), where n is the length of the largest possible list in a given index. All in all, however, an incease in speed to create the hash table and search/compare against it was noticed.

This lab really drove home the efficiency of a hash table. While it was noted in class that a good hash table with a good hash function provided faster access, search, and insertion, it was definitely seen first-hand in this lab. The program took a while longer to populate the data structures when it came to AVL trees and Red-Black trees than this one. In addition, it was very fun having to think of what ways words could be converted into relevant numbers that could be used in adequate hash functions. All in all, this was definitely a very fun and informative lab.

**APPENDIX**

#!/usr/bin/env python3  
# -\*- coding: utf-8 -\*-  
"""  
File: Main.py  
Name: Angel Villalpando  
Date: 11/11/2018  
Course: CS 2302 - Data Structures  
Description: This program provides the load factor and the average number of comparisons for two hash tables created using  
a user provided .txt file. The first one uses a hash function that creates a string of the ASCII representation of the word,  
casts it as an integer and mods that number by the size of the table. The second converts the words into a base 26 number  
using their ASCII value to determine their multiplication value against their respective power of 26.  
"""  
*from* HashTable *import* HashTable  
*from* HashTable2 *import* HashTable2  
  
anagramCount = 0  
  
*def* count\_anagrams(*word*, *word\_list*, *prefix*=""): # this is the modified method that now counts number of anagrams  
 *global* anagramCount  
 *if* len(*word*) <= 1:  
 str = *prefix* + *word  
  
 if word\_list*.search(str):  
 anagramCount += 1  
 *else*:  
 *for* i *in* range(len(*word*)):  
 curr = *word*[i: i + 1]  
 before = *word*[0: i]  
 after = *word*[i + 1:]  
  
 *if* curr *not in* before:  
 count\_anagrams(before + after, *word\_list*, *prefix* + curr)  
  
 *return* anagramCount  
  
  
*def* main():  
  
  
 *while True*:  
 usr\_file = input("\n\nPlease provide the name of the file that you are using (type <exit> to escape): ")  
 *if* str(usr\_file) == "exit" *or* str(usr\_file) == "Exit":  
 exit(0)  
 usr\_size = input("Please provide the size of the file that you are using (type <exit> to exit): ")  
 *if* str(usr\_size) == "exit" *or* str(usr\_file) == "Exit":  
 exit(0)  
  
 table\_size = int(usr\_size) \* 1.25  
 english\_words = HashTable(int(table\_size)) # Hash table created for the first hash function  
 english\_words2 = HashTable2(int(table\_size)) # Hash table created for the second hash function  
  
  
 file = open(usr\_file, "r")  
 *for* line *in* file:  
 word = line.split('\n')  
 english\_words.insert(word[0])  
 english\_words2.insert(word[0])  
 file.close()  
  
 print("\nThe load factor is: ", english\_words.get\_load\_factor()) # load factor is the same for both hash functions  
  
  
 comp\_total = english\_words.number\_comparisons()  
 print("\nThe average number of comparisons using Hash Function 1 is %s" % comp\_total) ## displays avg comparisons for hash function 1  
  
 comp\_total2 = english\_words2.number\_comparisons()  
 print("The average number of comparisons using Hash Function 2 is %s" %comp\_total2) ## displays avg comparisons for hash function 2  
  
 usr\_word = input("\nPlease provide a word to check for number of Anagrams (type <exit1> to exit): ") # anagram count prompt  
 *if* usr\_word == "exit1" *or* usr\_word == "Exit1":  
 exit(0)  
 anagrams = count\_anagrams(str(usr\_word), english\_words)  
 print("The word ", usr\_word, "has", anagrams, "Anagram(s).")  
  
  
  
  
main()

*class* HashTableNode:  
 *def \_\_init\_\_*(self, *item*, *next*):  
 self.item = *item* self.next = *next  
  
  
class* HashTable:  
  
 *def \_\_init\_\_*(self, *table\_size*):  
 self.table = [*None*] \* *table\_size* self.occupied = 0  
  
  
  
 # for my initial hash function, the concated ASCII values of each word's letters are casted as 'int' and divided by the table size.  
 *def* hash(self, *k*):  
 numK = ''.join(str(ord(c)) *for* c *in k*) # turns word into a string of ASCII representation  
 *return* int(numK) % len(self.table) # cast of ASCII string as int, mod by table size  
  
 *def* hash2(self, *k*):  
 key = 0  
 standard = ord('a') # since letters analyzed are lower case, will use only ASCII value 97  
  
 *for* i *in* range(len(*k*)):  
 base = ord(*k*[i]) - standard  
 key += base \* pow(26, (len(*k*) - 1 - i))  
  
 *return* key % len(self.table)  
  
  
 *def* insert(self, *k*):  
 pos = self.hash(*k*)  
 self.table[pos] = HashTableNode(*k*, self.table[pos])  
  
  
 *def* search(self, *k*):  
 loc = self.hash(*k*)  
  
 temp = self.table[loc]  
  
 *while* temp *is not None*:  
 *if* temp.item == *k*:  
 *return True* # ***TODO: Two lines missing*** temp = temp.next  
  
 *return False  
  
  
 def* no\_duplicates\_insert(self, *k*):  
 loc = self.hash(*k*)  
  
 temp = self.table[loc]  
  
 *while* temp *is not None*:  
 *if* temp.item == *k*:  
 *return* # ***TODO: Two Lines missing*** temp = temp.next  
  
 self.table[loc] = HashTableNode(*k*, self.table[loc])  
  
  
 *def* get\_load\_factor(self):  
  
 num\_elements = 0  
 *for* i *in* range(len(self.table)):  
 temp = self.table[i]  
  
 *while* temp *is not None*:  
 num\_elements += 1 # ***TODO: Replace return with body of the loop (more than 1 line missing)*** temp = temp.next  
 *return* num\_elements / len(self.table)  
  
 *def* number\_comparisons(self):  
  
 total\_elem = 0  
 num\_occupied = 0  
 *for* i *in* range(len(self.table)):  
 temp = self.table[i]  
 *if* temp *is not None*:  
 num\_occupied += 1  
  
 *while* temp *is not None*:  
 total\_elem += 1  
 temp = temp.next  
  
 *return* total\_elem/num\_occupied

*class* HashTableNode2:  
 *def \_\_init\_\_*(self, *item*, *next*):  
 self.item = *item* self.next = *next  
  
  
class* HashTable2:  
  
 *def \_\_init\_\_*(self, *table\_size*):  
 self.table = [*None*] \* *table\_size* self.occupied = 0  
  
  
 *def* hash2(self, *k*): # converts word into a base 26 number  
 key = 0  
 standard = ord('a') # since letters analyzed are lower case, will use only ASCII value 97  
  
 *for* i *in* range(len(*k*)):  
 base = ord(*k*[i]) - standard # base number determined to multiply against respective power of 26  
 key += base \* pow(26, (len(*k*) - 1 - i)) # key becomes the new, base 26 number  
  
 *return* key % len(self.table)  
  
  
 *def* insert(self, *k*):  
 pos = self.hash2(*k*)  
 self.table[pos] = HashTableNode2(*k*, self.table[pos])  
  
  
 *def* search(self, *k*):  
 loc = self.hash2(*k*)  
  
 temp = self.table[loc]  
  
 *while* temp *is not None*:  
 *if* temp.item == *k*:  
 *return True* # ***TODO: Two lines missing*** temp = temp.next  
  
 *return False  
  
  
 def* no\_duplicates\_insert(self, *k*):  
 loc = self.hash2(*k*)  
  
 temp = self.table[loc]  
  
 *while* temp *is not None*:  
 *if* temp.item == *k*:  
 *return* # ***TODO: Two Lines missing*** temp = temp.next  
  
 self.table[loc] = HashTableNode2(*k*, self.table[loc])  
  
  
 *def* get\_load\_factor(self):  
  
 num\_elements = 0  
 *for* i *in* range(len(self.table)):  
 temp = self.table[i]  
  
 *while* temp *is not None*:  
 num\_elements += 1 # ***TODO: Replace return with body of the loop (more than 1 line missing)*** temp = temp.next  
 *return* num\_elements / len(self.table)  
  
 *def* number\_comparisons(self):  
  
 total\_elem = 0  
 num\_occupied = 0  
 *for* i *in* range(len(self.table)):  
 temp = self.table[i]  
 *if* temp *is not None*:  
 num\_occupied += 1  
  
 *while* temp *is not None*:  
 total\_elem += 1  
 temp = temp.next  
  
 *return* total\_elem/num\_occupied

**Certification**

“I certify that this project is entirely my 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.”

-Angel Villalpando

11/13/2018