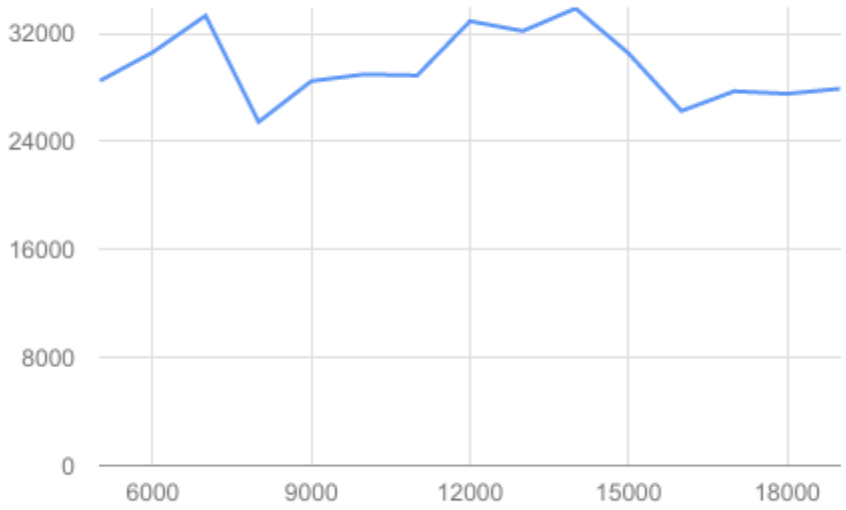
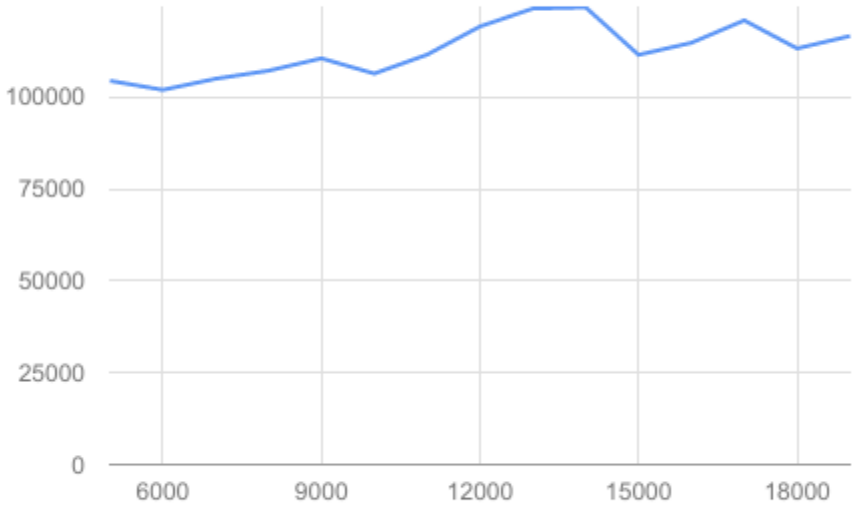
Brian Suh and Jihoon You, CSE 100 Prof. Alvarado, Project 2 Final Report

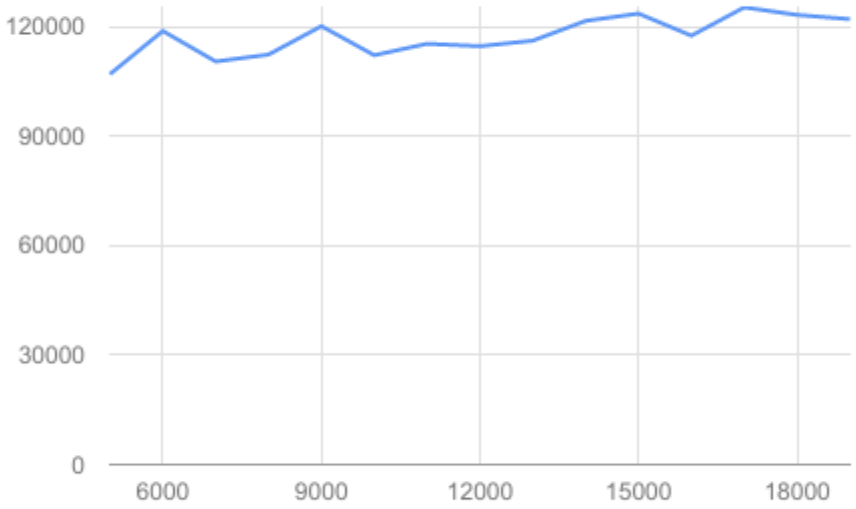
1.




**DictionaryHashtable** – A hash table has an expected case time to find of O(1). Based on the results of our graph above, we can see that this holds true as our hash table is the fastest amongst the three data structures and is steady in its rate.




**DictionaryBST** - A BST has an expected worst case time to find of O(log n). Referring to our graph above, we can see that our results justify the worst case. Despite the sudden spike in the middle of the run, the BST overall is faster compared to the TST and much slower compared to the hash table.




**DictionaryTrie** - A trie has been mentioned to have an average case time of O(log n) and worst case O(n). Our results show the expected average case time of O(log n) as it is only slightly slower compared to the BST. However, it is still far slower compared to the hash table.

a.

Hash Function 1:

Our first hash function is the djb2 hash function. It takes a hash value of 5381 and goes through each of the characters in the string, adding the value of the current character to the product of the hash and 33 (33 is consider a magic number) . This hash is then updated with this new value.

Source: http://www.cse.yorku.ca/~oz/hash.html

Hash Function 2:

Our second hash function is the sdbm hash function. It takes a hash value of 0 and like djb2 it goes through each character of the string, only this time using a different method. The function essentially updates hash to the outcome of this particular equation: hash(i-1)*65599 + str[i]. 65599 is the magic number for this function, str represents the string, and the letter i is an int representing the index of the current character the function is looking at.

Source: http://www.cse.yorku.ca/~oz/hash.html

b.

In the main of our benchhash.cpp file, we passed three different strings to both hash function 1 and hash function 2. These three strings were “say,” “hi,” and “bye.”

To verify that the hash functions gave the desired out, we manually calculated the hash values of the three strings by hand:

Hash function 1:

“say” -> s is 115, a is 97, y is 121 -> (((5381*33+115)*33+97)*33+121) % 2000 = 1554

“hi” -> h is 104, i is 105 -> ((5381*33+104)*33+105) % 2000 = 1446

“bye” -> b is 98, y is 121, e is 101 -> (((5381*33+98)*33+121)*33+101) % 2000 = 1813

Hash function 2:

“say” -> s is 115, a is 97, y is 121 -> (((0*65599+115)*65599+97)*65599+121) % 2000 = 299

“hi” -> h is 104, i is 105 -> ((0*65599+104)*65599+105) % 2000 = 401

“bye” -> b is 98, y is 121, e is 101 -> (((0*65599+98)*65599+121)*65599+101) % 2000 = 1070

c.

**Freq1**

hashFunction1 with hash table size 2000

#hits slots receiving the #hits

0 597

1 157

2 27

3 2

Printing the statistics for hashFunction2

#hits slots receiving the #hits

0 595

1 162

2 24

3 1

4 1

**Freq2**

hashFunction1 with hash table size 2000

#hits slots receiving the #hits

0 589

1 160

2 29

3 1

hashFunction2

#hits slots receiving the #hits

0 626

1 161

2 16

3 1

**Freq3**

hashFunction1 with hash table size 2000

#hits slots receiving the #hits

0 598

1 152

2 26

3 5

hashFunction2

#hits slots receiving the #hits

0 616

1 160

2 20

3 1

d.

Based on our results from running the hash functions on different data. We have come to the conclusion the hash function 1 is superior to hash function 2. The reason is that for the Freq1.txt file, hash function 2 had more hits compared to hash function 1. In all other cases, they were pretty close, but nevertheless hash function 1 has come out the victor. We expected hash function 1 to be better, so our results solidifies our expectation.

e.

READ IT!