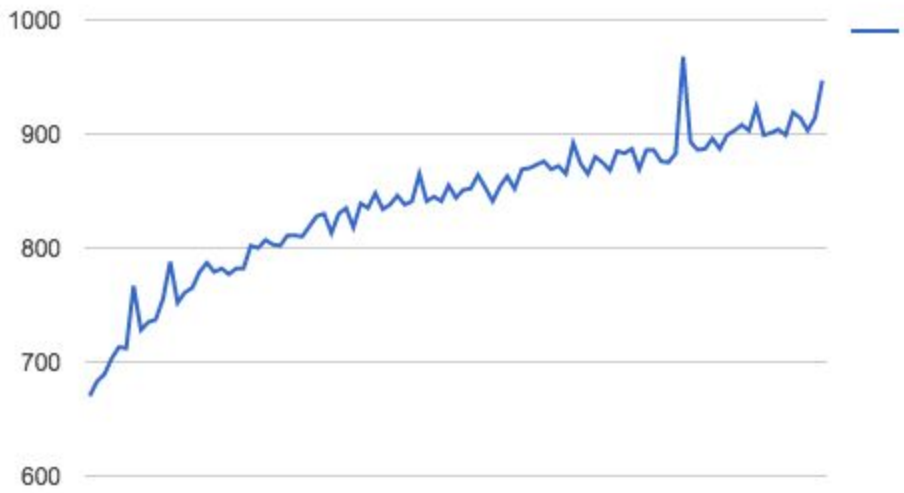
I. Bench Dict

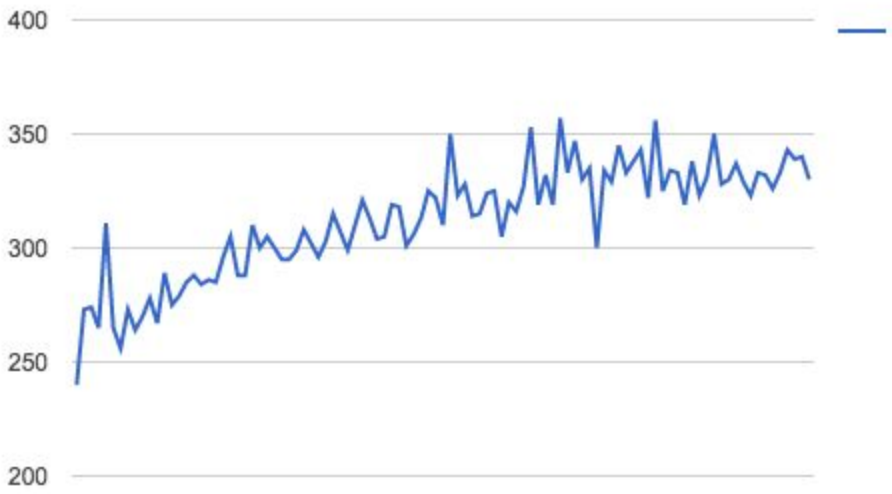
A. Dictionary BST



Comment: We can see that the line is characteristic of a logarithmic function. I used 100 words to insert at a busy time in the labs, so that is what may have caused some of the spikes. However, the graph overall seems to be that of a log function.

B. DictionaryHashtable

C. DictionaryTrie



Comment: The curve here also seems to be characteristic of a logarithmic function. If we ignore the spikes, the graph creeps up slowly and then levels off just like a log function. The spikes might be due to the height of the tree being different each time we step in a way that doesn't follow a log trend. For example, the heights might be 5000: 30, 6000: 35, 7000:36, 8000: 40.

1. I implemented a TST so I expect a logarithmic running time, which did happen. So I am satisfied. Because the graph makes its way up in the shape of a logarithmic function functions, although I am a little worried about the spikes. However, the TST still did find is what seems to be O(log(N)) running time where N is the number of elements.

II. BenchHash

a. The first hash function works by multiplying the hash value at the previous index of the string by 33 and adding the value at the current index of the string to the result(hash*33+c) at each step. Before this loop, it will first be initialized to 5381 which is a prime number. After going through all the letters, we mod the key yo the size of the table.

The second hash function, for every letter in the string, it will first shift the previous hash value left 5 times and add it to the result of the previous hashvalue shifted right 2 times. That will be added to the ASCII value of the current letter in the index. The result will be XOR with the previous hashvalue. After that, like the first hash function, we mod it with the size of the table.

b. K
c. Results

Hashfunction 1

Shuffled_freq_dict with 1000 words

#hits	#slots receiving #hits
0	1209
1	611
2	155
3	21
4	4

Average steps: 1.242

freq_dict with 1000 words

#hits	#slots receiving #hits
0	1209
1	611
2	155
3	21
4	4

Average steps: 1.26

freq1 with 1000 words

#hits	#slots receiving #hits
0	1215
1	608
2	145
3	26
4	6

Average steps: 1.259

freq3 with 1000 words

#hits	#slots receiving #hits
0	12016
1	610
2	135
3	36
4	3

Average steps: 1.261

freq3 with 1000 words

#hits	#slots receiving #hits
0	1204
1	625
2	141
3	27
4	3

Average steps: 1.24

Hashfunction 2

Shuffled_freq_dict with 1000 words

#hits	#slots receiving #hits
0	1218

freq_dict with 1000 words

#hits	#slots receiving #hits
0	1204

1	594
2	161
3	24
4	3

Average steps: 1.251

1	621
2	150
3	22
4	2
5	1

Average steps: 1.238

freq1 with 1000 words

#hits	#slots receiving #hits
0	1203
1	628
2	137
3	30
4	2

Average steps: 1.239

freq2 with 1000 words

#hits	#slots receiving #hits
0	1214
1	604
2	155
3	23
4	3
5	1

Average steps: 1.252

freq3 with 1000 words

#hits	#slots receiving #hits
0	1203
1	594
2	144
3	31
4	5
5	1

Average steps: 1.277	

d. Based on the outputs, the second hash function seems to be a better function for resolving collisions, because comparing the first to the second hashfunction results, the second one has a better step averages. We can see that this is not true for freq3.txt, which is a little disappointing but since the second hash function was a little more complex, it did better as expected.