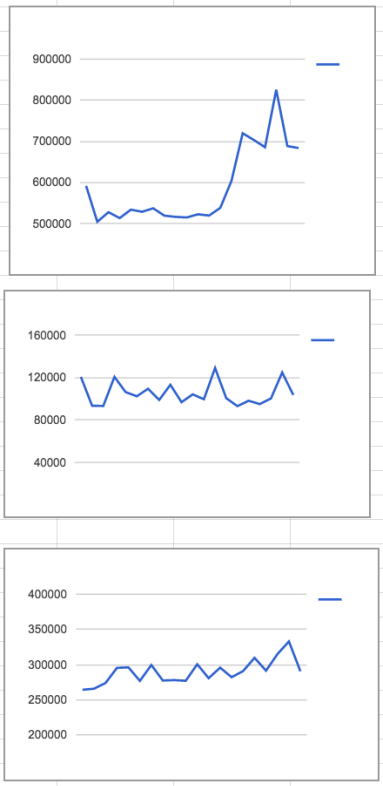
3. Benchmark Dictionaries

The below is the three graphs that are drawn by Google sheets, along with the most reliable run produced by my program.

Size	DictionaryBST	Size	DictionaryHashtable	Size	DictionaryTrie
7000	591478	7000	120474	7000	264128
7200	503958	7200	93316	7200	265629
7400	527326	7400	93091	7400	273686
7600	512946	7600	120457	7600	295203
7800	533523	7800	106179	7800	295985
8000	528337	8000	102208	8000	276771
8200	536777	8200	109245	8200	299321
8400	518919	8400	98723	8400	277270
8600	515923	8600	113047	8600	277832
8800	514475	8800	96653	8800	276915
9000	522173	9000	104026	9000	300389
9200	519182	9200	99395	9200	280588
9400	537901	9400	128879	9400	295606
9600	603709	9600	100378	9600	282000
9800	719473	9800	92929	9800	290664
10000	703283	10000	98011	10000	309507
10200	685510	10200	94840	10200	291261
10400	825439	10400	100205	10400	314925
10600	688391	10600	124620	10600	332841
10800	683556	10800	103399	10800	290396



3 graphs from top to bottom are BST, Hashtable, and Trie, respectively.

We would analyze the graph one by one. First, we look at the top graph, which is from the BST dictionary. The running time stays relatively steady before the size gets larger than 9400. After that we observe a spike in the running time and, despite showing some fluctuation, it never drops back down to the previous level of running time. Therefore, it is clear that the running time tends to increase as the dictionary size increases. However, my graph looks closer than an exponential growth rather than the theoretical O(log N). I think, in order to get a “flatter” curve that is more similar than the theoretical curve, I would have to further tweak the initial size of the dictionary and the step parameter. In my many runs, while I was not able to perfectly reproduce the O(log N) curve, note the incremental in the running time is not as significant as the shape of the graph may suggest, which is due to the scale of the y-axis. We can also calculate the

theoretical runtime at size=10800: log(10800/7200)=0.176(omit the first run as it seems like a outlier).
And 503985x (1+0.176)=592654(extrapolated runtime). And we got around 68xxxx as our empirical value, which is not that far off considering all other programs running on my system concurrently (the “noise”).

As for the dictionary using hash table as its backing data structure, the result is pretty much consistent with what I expect. The hash table should have constant running time and we see, indeed, that my graph shows that every run took around 10,000 nanoseconds. The curve is relatively flat in this case.

Lastly, we know that for MWT the theoretical running time is O(k), where k is the length of the longest string in the dictionary. According to my graph, we again see a pretty flat curve and stays steady around the 270,000-300,000 mark. This aligns with my expectation since we could verify the the strings in the shuffled_freq_dict.txt do not have a crazy wide range of lengths. Therefore, the graph would look similar to the one with constant running time.

4. Research and compare the performance of different hash functions for strings

For my first hash function, I use the one from our lecture 8, on slide 14. The function simply sums up all the ASCII values of every character in the string and modulo the size of the table.

The second hash function is found on the website called “General Purpose Hash Function Algorithms”. There are several files of source code that can be downloaded. I picked the JS hash function from there, which is a bitwise hash function written by Justin Sobel.
This hash function incorporates several bitwise operations on the original hash value(initialized to 1315423911), such as shift 5 bits to the left and 2 bits to the right.
We can reference the code snippet below:

```
for(std::size_t i = 0; i < key.length(); i++)
  {
     hash ^= ((hash << 5) + key[i] + (hash >> 2));
  }
```

We see that it loops over all characters in the string (the key) and use the OR operator “^” to update the hash value in every iteration. The line “hash ^= ((hash << 5) + key[i] + (hash >> 2));” is equivalent to “hash= hash ^ ((hash << 5) + key[i] + (hash >> 2));”

Test cases: “APPLE” “dam” “yeah” . table size=10
1. hashFunc1:
With the help of Ascii table, “APPLE” should have 65+80+80+76+69=370 , 370%10=0.

And my main function in benchhash.cpp would print out the hash value of “APPLE” to the terminal to verify that hashFunc1 indeed returns the expected hash value, 0.

for “dam”: 100+97+109= 306, 306%10= 6. It also matches the standard output when running main.
for “yeah”: 121+101+97+104= 423, 423%10=3. Verified!

2. hashFunc2 :

For this function, I calculated the following values by hand in each iteration with the aid of the Ascii table. (We know that to shift left one bit is to multiply the number by 2 in decimal representation. To shift right one bit would be to divide the number by 2 in decimal representation.)
“APPLE” :
2935292013
2762493542
446410447
1082616592
1636324889—–> 1636324889%10=9

“dam” :
2935291978
2762492409
446372850 –>446372850%10=0

“yeah” :

2935292325
2762503371
445694520
1110226382–>1110226382%10=2

So, in the end, we have 9,0,2 as our three hash values for the three strings in my test. They do match the output from my program.

The following lines of code are the ones that I used to verify my expected hash value. And they were commented out later for a cleaner output format.

```
 cout<<"testing the first hash function"<<endl;

    //print them all out and verify the hash values by hand
    string s1="APPLE";
    cout<< hashFunc1(s1, 10)<<endl;

    string s2="dam";
    cout<< hashFunc1(s2, 10)<<endl;

    string s3="yeah";
```

```
cout<< hashFunc1(s3, 10)<<endl;

cout<<"testing the second hash function"<<endl;

//print them all out and verify the hash values by hand
cout<< hashFunc2(s1, 10)<<endl;

cout<< hashFunc2(s2, 10)<<endl;


cout<< hashFunc2(s3, 10)<<endl;

cout<<endl;
```

————————————————-

Now moving on to compare the performance of my two hash functions.

Below is some data that I collect:

When using freq1.txt as our dictionary, we see the worst case of search for hashFunc1 is 1002 steps and for hashFunc2 is 1000 steps.
When using freq3.txt as our dictionary, we see the worst case of search for hashFunc1 is 1000 steps and for hashFunc2 is also 1000 steps.

A	B	C	D	E
hashFun1			hashFun2	
dictfile	num_words		dictfile	num_words
freq1.txt	1000		freq1.txt	1000
#hits	#slots receiving the #hits		#hits	#slots receiving the #hits
0	1330		0	1220
1	449		1	597
2	145		2	145
3	50		3	35
4	20		4	2
5	3		1000	1
7	2			
1002	1			
verage steps: 2.497			Average steps: 2.272	

A	B	C	D	E
hashFun1			hashFun2	
dictfile	num_words		dictfile	num_words
freq3.txt	1000		freq3.txt	1000
#hits	#slots receiving the #hits		#hits	#slots receiving the #hits
0	1519		0	1218
1	239		1	593
2	95		2	163
3	76		3	19
4	33		4	6
5	22		1000	1
6	8			
7	3			
8	4		Average steps: 2.266	
1000	1			
Average steps:	3.072			

Despite the worst cases for both functions seem pretty close, it is true that the second function consistently performs better than the first function. It aligns with my original expectation because we would guess that, by utilizing more complicated bitwise operation, hashFun2 (JS function) is likely to have a better (more even) distribution than hashFunc1, which can be poorly distributed due to the fact that the range of the sums of all possible Ascii values in strings is rather narrow.

In short, we can conclude hashFunc2 is a better hash function of the two.