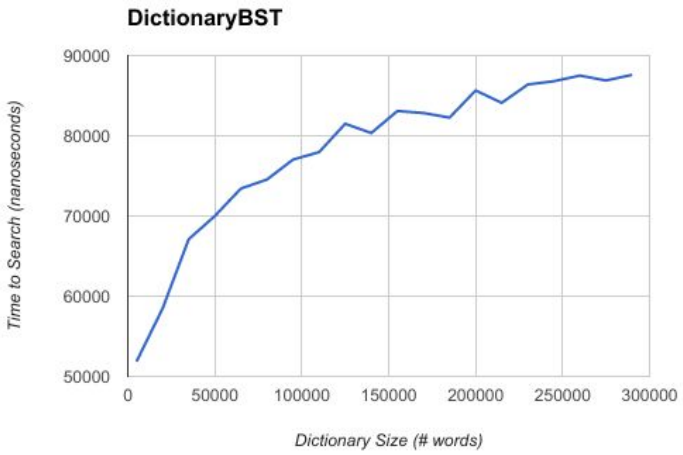
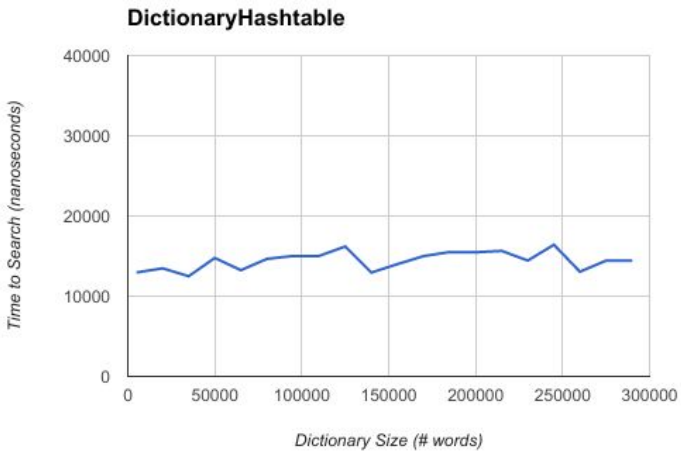
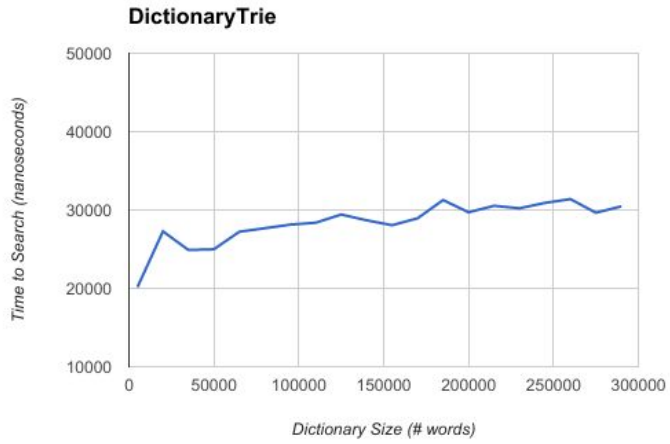
/*
* Name: Megan Chu
* PID: A12814536
* Date: April 28, 2017
* CSE 100 PA3 C++ Autocomplete
*/







All graphs have min dictionary size as 5000, step size as 15000, and number of iterations as 20

1. We can see that the BST graph increases drastically at first and gradually starts leveling off, ending at a runtime of 86000-87000 nanoseconds, which is representative of logarithmic runtime growth. This matches up with the analytical running time for BST, O(logn).

We see that the hashtable graph slightly increases in runtime for each growing dictionary size, however this increase is barely noticeable compared to the drastic stepsize of 15000 words and the growth of the other graphs. The hashtable runtime for find may not be not be exactly O(1), but it is representative of constant runtime by its near 0 average slope. I think we may account for the increase in runtime due to collision handling. The more items there are in the dictionary, the more collisions there are, and the more lines of code that need to be executed to find a certain word. I also conclude that the hashtable graph is a near enough match for the analytical running time for hash table, O(1).

We see that the trie graph increasing in runtime by about 10000 nanoseconds over a difference of 285000 words in the dictionary. It also levels off at around 30000 nanoseconds. Which is reasonable because there is a certain min and max length of how long a string can be, and therefore the runtime should not go out of bounds of the shortest and longest word lengths. I conclude that at a search time of ~30000 nanoseconds is the search time for one of the longest length words in the dictionary, and thus this matches with the analytical running time for MultiwayTrie, O(k), where k is the length of the longest word.

a. hashFunc1 was taken from the following source:
http://stackoverflow.com/questions/8317508/hash-function-for-a-string
It works by summing up all the ASCII key values of each char in the string input, and then returning that sum mod size of the table so that the returned int is between 0 and the table size.

hashFunc2 was thought up on my own based on hashFunc1. The order of the characters in each string have no significance in hashFunc1, so in hashFunc2, we multiply the ASCII key of each char by an integer based on its position in the string. So the char in the first position gets multiplied by 1, and the char in the second position gets multiplied by 2, and so on. This makes the position of the char in the string relevant, although even with this implementation there is still a possibility that different strings will somehow sum up to the same value. The total sum is again mod by table size so that the returned int is between 0 and the table size.

b. I verified the correctness of each hash function's implementation by calculating the hash values by hand, and then running the hash functions with the test cases in a main method and printing out the results. I made sure the printed results matched with my calculations. The three test cases used were "apple", "banana", and "orange" for both hash functions.

For hashFunc1, the expected hashed value was
(97+112+112+108+101)%10 = 0 for apple,
(98+97+110+97+110+97)%10 = 9 for banana,
(111+114+97+110+103+101)%10 = 6 for orange.

For hasFunc2, the expected hashed value was
[(1)97+(2)112+(3)112+(4)108+(5)101]%10 = 4 for apple,
[(1)98+(2)97+(3)110+(4)97+(5)110+(6)97]%10 = 2 for banana,
[(1)111+(2)114+(3)97+(4)110+(5)103+(6)101]%10 = 1 for orange.

To verify that hashFunc1 and hashFunc2 gave the expected values, I included a main method in benchhash.cpp and had it print out the hashed values with the specific string inputs. I tweaked my code until the hash functions returned the correct value I calculated by hand.

c.

freq1.txt

num_words	1000										
hashFunc1	#hits	0	1	2	3	4	5	7			
Avg: 1.498	#slots	1330	449	146	50	20	3	2			
hashFunc2	#hits	0	1	2	3	4					
Avg: 1.227	#slots	1200	626	149	24	1					
num_words	2000										
hashFunc1	#hits	0	1	2	3	4	5	6	7	8	9
Avg: 1.911	#slots	2976	535	220	142	69	37	13	5	2	1
hashFunc2	#hits	0	1	2	3	4					
Avg: 1.235	#slots	2422	1202	332	42	2					

freq2.txt

num_words	500									
hashFunc1	#hits	0	1	2	3	4	6			
Avg: 1.396	#slots	645	251	72	25	6	1			
hashFunc2	#hits	0	1	2	3	4				
Avg: 1.244	#slots	609	294	86	10	1				
num_words	1000									
hashFunc1	#hits	0	1	2	3	4	5	6	7	8
Avg: 1.672	#slots	1402	366	130	54	36	7	3	1	1
hashFunc2	#hits	0	1	2	3	4	5			
Avg: 1.264	#slots	1221	596	149	31	2	1			

freq3.txt

num_words	250							
hashFunc1	#hits	0	1	2	3	4		
Avg: 1.284	#slots	311	137	44	7	1		
hashFunc2	#hits	0	1	2	3	4		
Avg: 1.316	#slots	313	137	40	7	3		

num_words	500							
hashFunc1	#hits	0	1	2	3	4	5	6
Avg: 1.596	#slots	694	185	71	34	10	5	1
hashFunc2	#hits	0	1	2	3	4		
Avg: 1.384	#slots	650	236	84	24	6		

d. hashFunc2 is noticeably better for larger table sizes or more words. The worst case # hits it had in all tests was 5 or less, compared to hashFunc1 whose worst case #hits increased as the table size/# words increased. The average steps for hashFunc2 is also noticeably better as it grows at a much slower rate than hashFunc1's average steps as the table size/#words increases. Smaller worst case and average steps for hashFunc2 allows for a faster search time and a greater efficiency than hashFunc1. This matched my expectation because hashFunc2 gives a hash value that depends on the position of a letter in the string, versus hashFunc1 that only gave a hash value based on the characters that composed the string in any order. For example, hashFunc1 would give the same value for eat and ate, but hashFunc2 would give different hash values. This is obviously better. I thought hashFunc2 would perform better than hashFunc1 in all cases, but this was not true for small table sizes as seen in the test with freq3.txt and 250 words in the table. However, since the table size is so small, the difference in efficiency of different hash functions are not as noticeable, and by testing the larger table sizes/number of words we gain a greater insight that for the most part hashFunc2 is better for a large range of table size/# words.