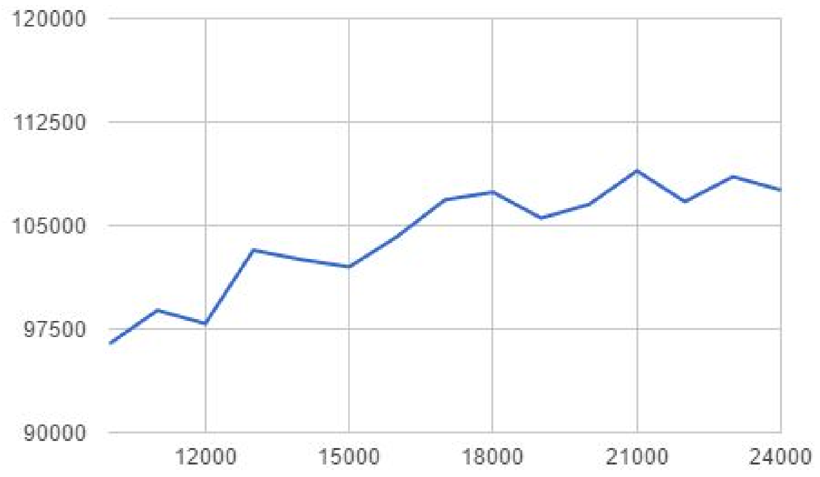
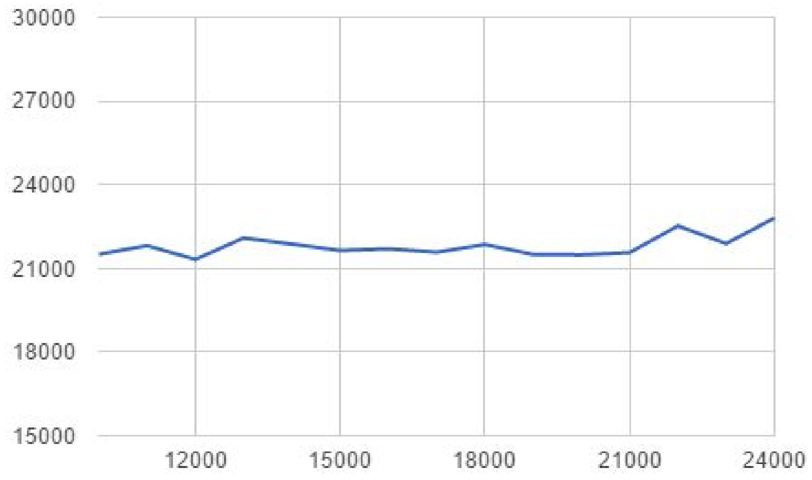
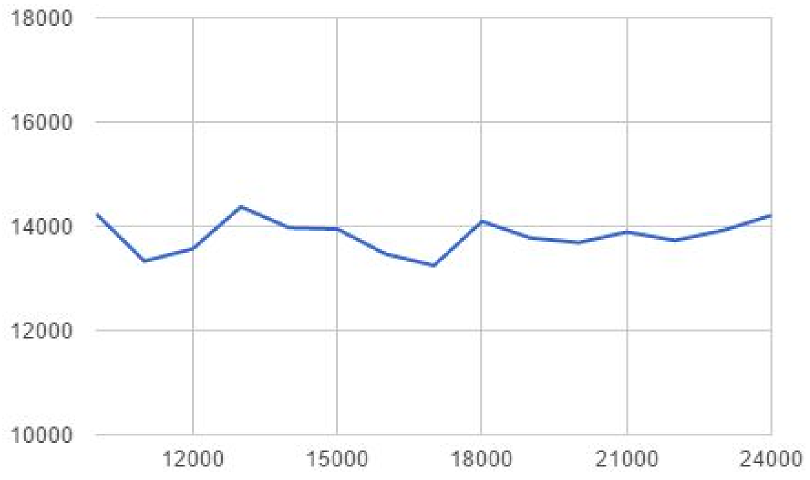
Dictionary BST: 10000 to 24000



Dictionary Hashtable: 10000 to 24000



Dictionary Trie: 10000 to 24000



1.) Our results for our benchmarks are somewhat consistent with the expected average results. For our BST, we find that the running times roughly reflect a logarithmic growth pattern, where growth is somewhat fast at lower values and rise very slowly on higher inputs. As you can observe in our graphs, our initial inputs grow somewhat substantially, but later inputs have very minor differences.

    For our hash table, we observe that our running times are consistent regardless of input size. However, the running times for our hash table are significantly higher than the expected O(1). We suspect that server stress and an excess of inputs may be slowing down the running time, since the worst case runtime is O(n) due to separate chaining. As observed in our graphs, our run times are constant regardless of our input size, though they are higher than expected.

    For out MWT, we observe extremely fast run times, which are consistent with O(K) average case. This run time seems to be constant regardless of input size, which is also consistent with expected behavior. We attribute minor variations to outside factors such as server stress, other processes, etc. . As you can observe in our graph, our run times are very fast in comparison, and are constant regardless of input size.

Hash Benchmarking:

a. Bernstein hashing works by multiplying the current hash key by a constant and adding the ascii value of each letter in every iteration. Additive hashing works by adding together the ascii values of every character. **(Source:** http://www.eternallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx)
b. To verify the hash function, we manually computed the expected hash keys of various words and used print statements to compare the expected to the actual results. For example, we hashed the words:
   - "aaaa", which has an ascii value of 97*4 in the additive case, and 33*(97+33(97+33*(97)))+97,
   - "cat", which has an ascii value of 99+97+116 in the additive case and 33*(97+37*(99))+116
   - "dog", which has an ascii value of 100+111+103 in the additive case, and 33*(111+33*(100))+103

   After comparing these expected results with the actuals, we confirmed that they were equivalent. Therefore, we were confident that the functions were operating correctly.

C. We ran three different inputs, listed below:

freq1.txt 50			
# of hits	# of slots received hits	# of hits	# slots received hits
0	76	0	80
1	14	1	11
2	8	2	6
3	1	3	0
4	0	4	0
5	0	5	0
6	0	6	0
7	0	7	1
8	0	8	1
9	0	9	0
10	0	10	0
11	0	11	0
12	0	12	1
13	0		Average: 0.88
14	0		
15	0		
16	0		

17	1		
	Average: 3.94		

shuffled_freq_dict.txt 250			
# of hits	# of slots received hits	# of hits	# slots received hits
0	353	0	367
1	104	1	85
2	28	2	28
3	6	3	7
4	2	4	1
5	1	5	4
6	2	6	3
7	1	7	1
8	1	8	1
9	0	9	2
10	0	10	0
11	0	11	0
12	0	12	0
13	0	13	1
14	0		Average: 1.46
15	1		
16	0		
17	1		
	Average: 2.552		

shuffled_freq_dict.txt 400			
# of hits	# of slots received hits	# of hits	# of slots received hits
0	590	0	615
1	140	1	110
2	38	2	33

3	15	3	18
4	5	4	8
5	3	5	4
6	1	6	4
7	1	7	1
8	3	8	1
9	0	9	2
10	0	10	1
11	1	11	0
12	0	12	0
13	0	13	0
14	0	14	0
15	1	15	0
16	1	16	1
17	1	17	1
18	0	18	1
19	0		Average: 2.295
20	0		
21	0		
22	0		
23	0		
24	0		
25	1		
	Average = 3.1705		

D. The Bernstein hashing function seems to operate more effectively, it causes less collisions, most likely due to it's greater variation in hash keys. This causes the lower average number of steps required to find a value, which will allow this function to operate faster than its additive counterpart. This is somewhat inline with my prior expectations; because every character is multiplied by a constant, it becomes impossible for non-identical inputs to hash to the same keys. However, the degree to which the hash functions differ was somewhat surprising; the degree of variation is much smaller than anticipated. However, this may be because the keys had to be modded by the size of the array, which produced collisions that otherwise would not have occurred.