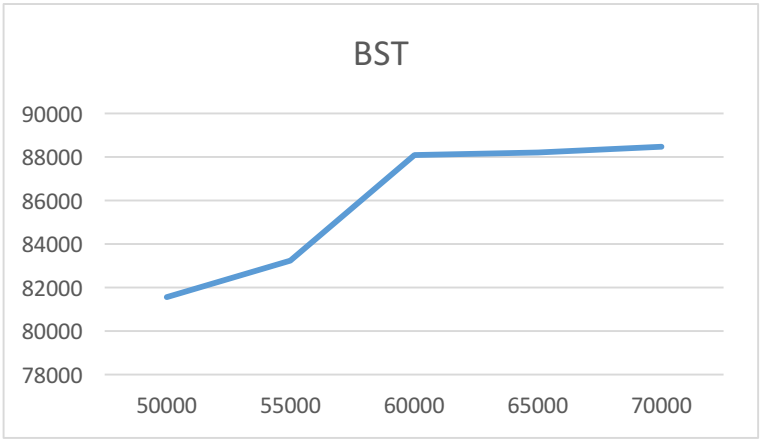
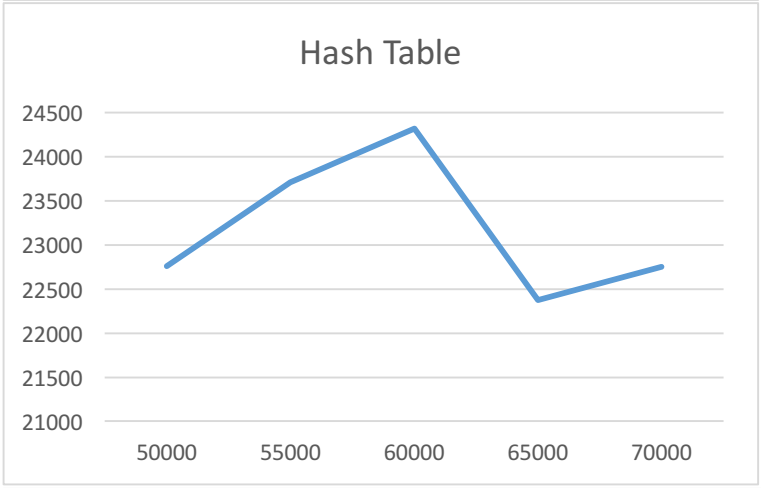
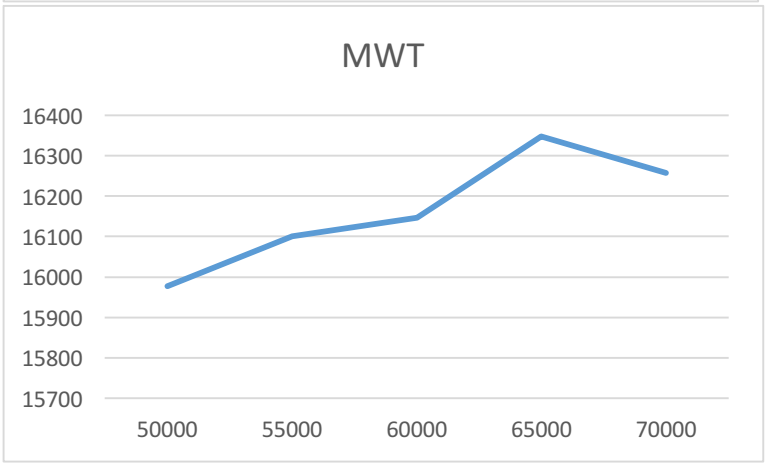
Adrian Cordova y Quiroz A12010305
Jonathan Chiu (A12113428)

Final Report

3.3 Graphs (X-axis: Dict Size; Y-axis: Run Time)







3.4 In class we saw that a Hashtable has expected case time to find of O(1), a BST worst case O(logN) and a MWT worst case O(K), where K is the length of the longest string. We didn't look at the running time of the TST, though the book mentions that its average case time to find is O(logN) (and worst case O(N)). Are your empirical results consistent with these analytical running time expectations? If yes, justify how by making reference to your graphs. If not, explain why not and also explain why you think you did not get the results you expected (also referencing your graphs).

Our empirical results are consistent with the analytical running time expectations for the BST, the Hashtable, and the MWT. The BST has a higher running time as the dictionary size gets bigger, and the Hashtable and MWT are relatively constant as the dictionary increases in size.
4.

a. **summationHash** - http://research.cs.vt.edu/AVresearch/hashing/strings. php
summationHash calculates the sum of the ASCII values of the letters in a given string. If the size of the hash table is lesser compared to the resultant summations, then summationHash should efficiently allocate strings evenly between the hash table slots, since it provides uniform weight to all characters in the string.

bernsteinHash - http://www.eternallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx
bernsteinHash does the same as summationHash, but multiplies the sum by 11 upon each iteration before adding the ASCII value. The function works well for small character keys, where it can perform better than algorithms that show a more random allocation.

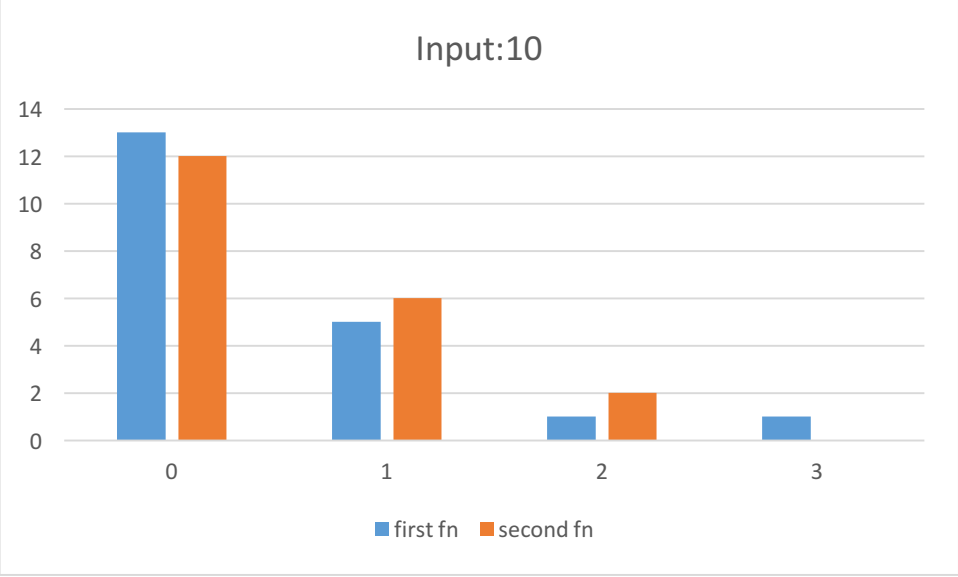
TO DO!!!!!!

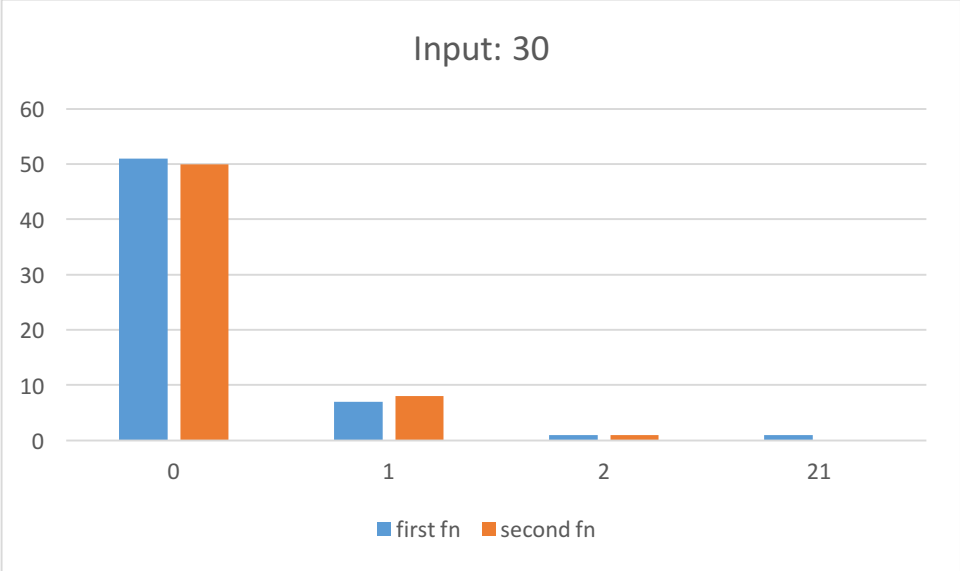
a. Describe how you verified the correctness of each hash function's implementation. Describe at least 3 test cases you used, what value you expected for each hash function on each test case, and the process you used to verify that the functions gave this desired output.
The way we verified the correctness of the hash function's' implementation was 3 different tests. The first test hashed a single letter, in our case 'h'. The second test hashed a single lower case letter with no spaces, in our case 'hello'. The third test hashed a single all-caps word with no spaces, in our case "HELLO".

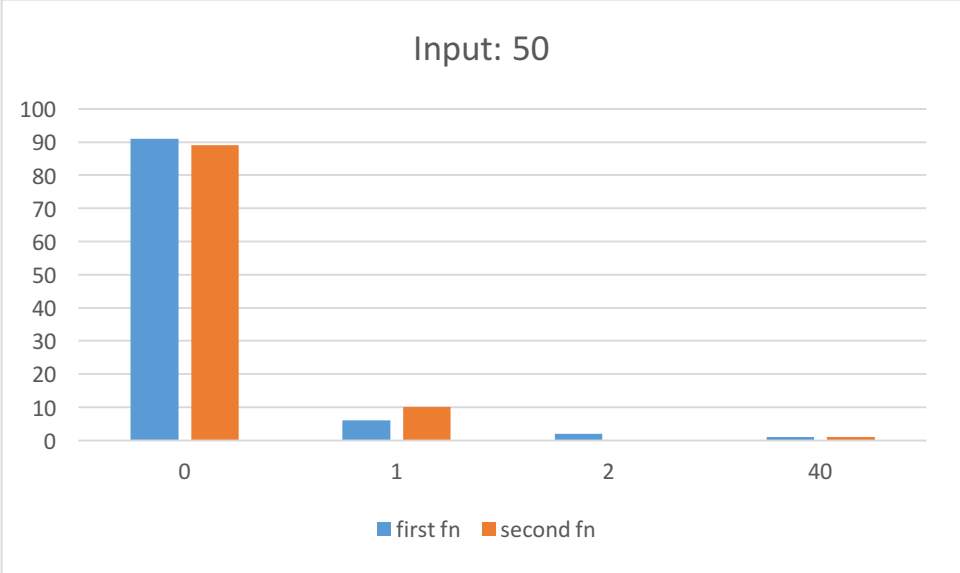
b. Run your benchhash program multiple times with different data and include a table that summarizes the results of several runs of each hash function. Format the output nicely--don't just copy and paste from your program's output.

	First fn	Second Fn
Number of hits	Input: 10	Input: 10
0	13	12
1	5	6
2	1	2
3	1	0
Number of hits	Input: 30	Input: 30
0	51	50
1	7	8
2	1	1
21	1	0
Number of hits	Input: 50	Input: 50
0	91	89
1	6	10
2	2	0
40	1	1

Bar Charts for functions (X-axis: Number of hits; Y-axis: Number of slots reciving the number of hits)







c. Comment on which hash function is better, and why, based on your output. Comment on whether this matched your expectation or not.

The superior hash function is the function that has the lesser number of slots getting the hits for the bigger hits. In our case, both functions are relatively equal in terms of efficiency and collisions, but the second function: bernsteinHash, is a little bit better than the summationHash which does correlate with our expectations since BernsteinHash is expected to run faster than most other algorithms.