Partner1:
Name: Jiaqi Fan
PID: A12584051
Partner2:
Name: Xuanru Zhu
PID: A91064234
Date: 2/7/2017
PA2

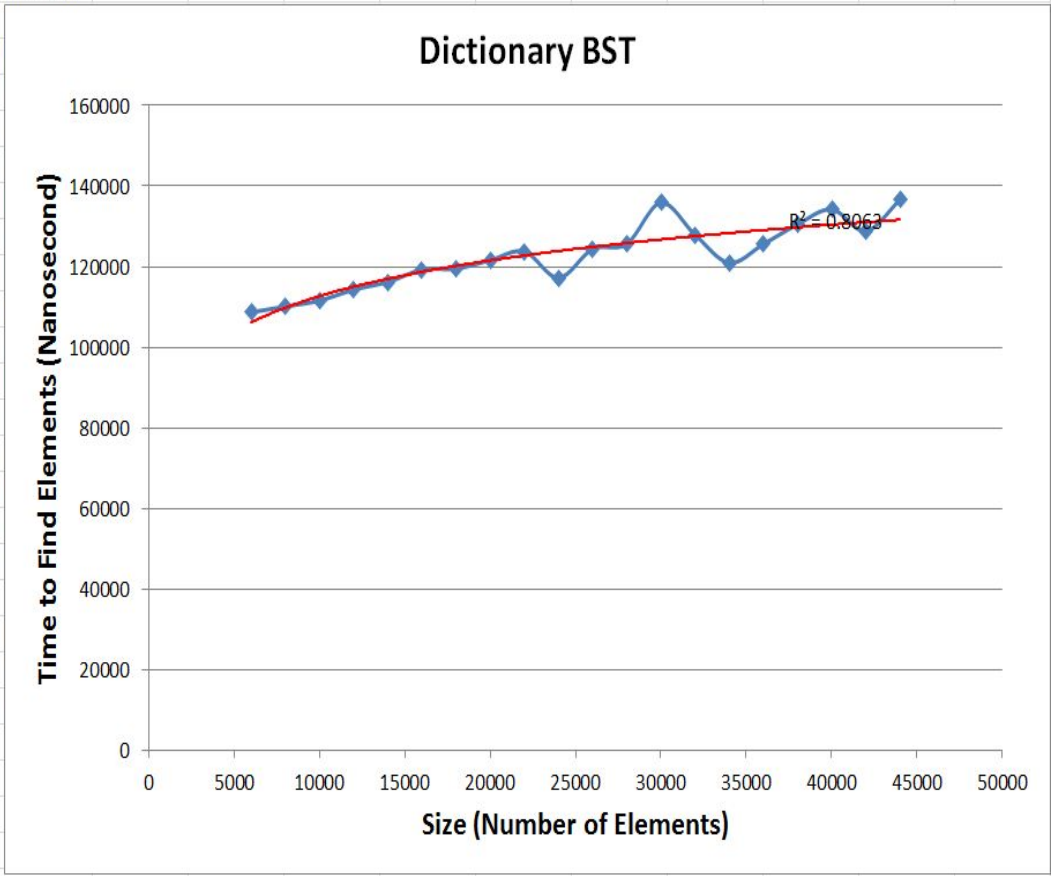
Final Report

Dictionary Part:

1. Plot your results in a graph Using the most reliable run of 3.2 Produce a graph (using google sheets, excel, gnuplot, etc) for each dictionary type. Plots are between Dictionary Size and Run Time.

Graph for Dictionary BST benchmarking :

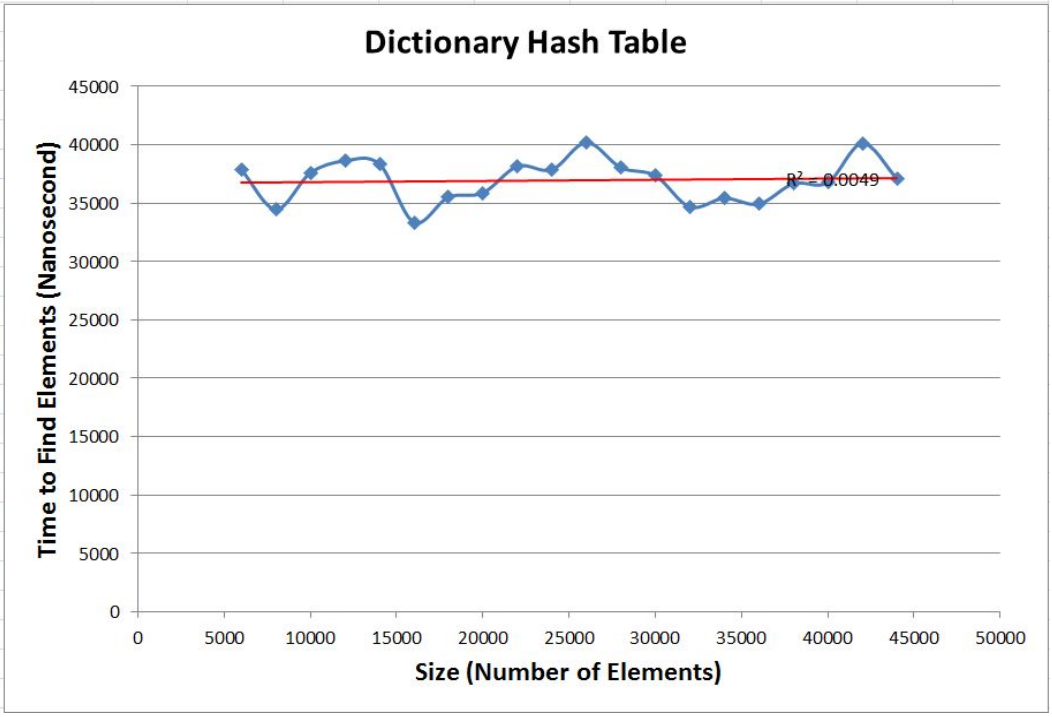
DictionaryBST is benchamrking	
6000	108695
8000	110114
10000	111586
12000	114297
14000	116202
16000	119233
18000	119517
20000	121613
22000	123754
24000	117217
26000	124383
28000	125633
30000	135897
32000	127914
34000	120958
36000	125575
38000	130475
40000	134220
42000	128817
44000	136569



Graph for Dictionary Hash Table benchmarking:

./benchdict 6000 2000 20 shuffled_freq_dict.txt
DictionaryHashTable is benchamrking

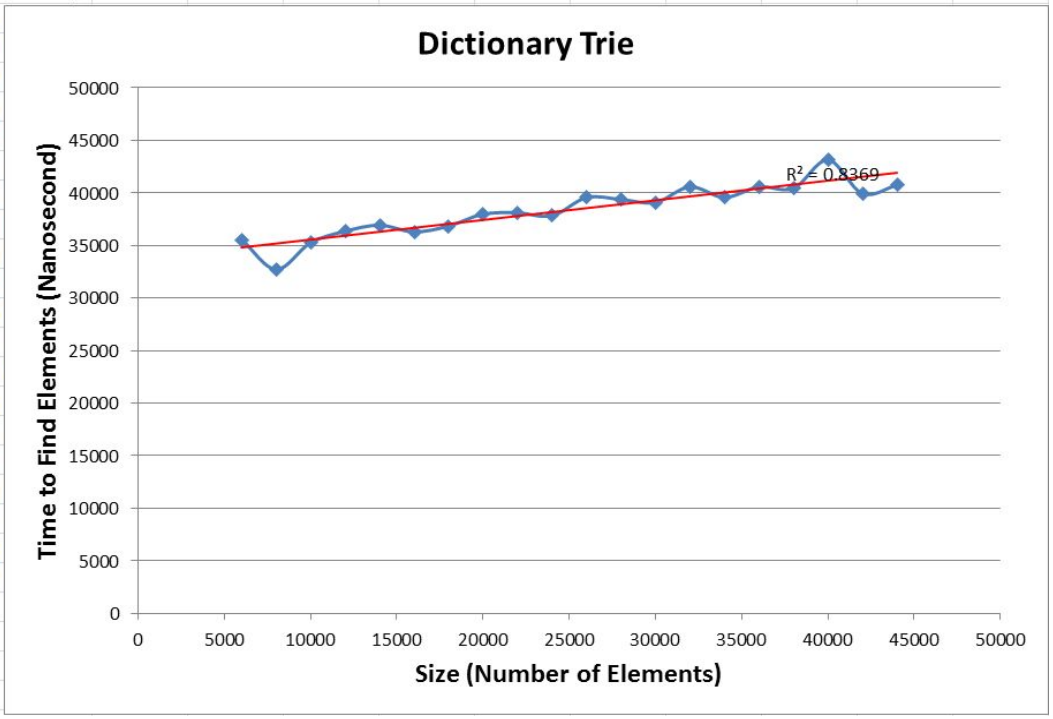
6000	37847
8000	34474
10000	37593
12000	38624
14000	38338
16000	33363
18000	35512
20000	35880
22000	38157
24000	37868
26000	40202
28000	38043
30000	37353
32000	34675
34000	35420
36000	34926
38000	36666
40000	36766
42000	40117
44000	37125



Graph for Dictionary Trie (MWT) benchmarking:

./benchdict 6000 2000 20 shuffled_freq_dict.txt
DictionaryTrie is benchamrking

6000	35565
8000	32730
10000	35269
12000	36355
14000	36932
16000	36302
18000	36852
20000	37983
22000	38110
24000	37837
26000	39566
28000	39382
30000	39100
32000	40578
34000	39576
36000	40584
38000	40440
40000	43142
42000	39938
44000	40749



2. 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).

For a Binary Search Tree the worst case of search is O(logN). According to our graph the result consistent with the analytical running time which is O(logN), with the size of DictionaryBST grows in each iteration, we expect to see the search time starting growing slowly. For example, let's take a look of the data in the beginning with minimum size 6000 the search time is 108695 ns and for size 10000 the search time is 111586 and we subtract the search time (111586 - 108695 = 2891 ns) is the difference. Then we take a look at the somewhere that is middle of the data let's take size 28000 and 32000 the run time for both is 125633 and 127914 so find the difference (127914 - 125633 = 2281) we can see the the time growth start decreasing. Just like the growth rate of logN. For graph accuracy we also inserted the graph trendline, and the R^2 of the trendline is 0.81 for a log function which means that the data is really reliable. Because when the R^2 value of trendline getting close to 1 then the data is really reliable. However, there are some peak points around in our data, that might highly caused by the background program running in the school ieng6 sever and instability of ssh terminal. If the program runs on clean server with less users than the result will be much better.

For Hash Table the worse find is going to O(1) a constant time. According to to our graph the result consistent with the analytical running time which is O(1), when the size of hash table grows the search time is nearly unchanged. Again we can pick some points and do a graph data analysis. Let's pick the size 6000 and 10000 the runtime for them are 37847 and 37593 (37847 - 37593 = 254) the time difference is about 254 nanosecond. Picking some data near end at size 38000 and 40000 the runtimes are 36666 and 36766 take the difference (36766 - 36666 = 100) by using this data analysis method we can see the the difference between datas are not increasing. Therefore the runtime of this algorithm must be constant which is O(C). As explained in BST the peak points might caused by the server and ssh terminal.

We implemented Multiple Way Trie (MWT) and the worse case for search is O(k) k which is the longest string. Our data is consistent with the analytical running time which is O(k). According to the trendline we can see the R^2 value for the data is 0.84 and it is

really close to 1 which means datas are very reliable. however , for MWT we can not use the data analysis method above because the worst runtime is based on the longest word. In the dictionary file shuffled_freq_dict.txt the words are sometime very long sometime very short. By looking at the graph we obtained the beginning part is pretty decent however start from middle the runtime grows pretty fast. We noticed that when we run the program on ieng first two function BST and Hash output result very fast when the time to run trie is taking longer to output the result I think is because the we have used too much memory and cpu on the server. When we continue to run the program the memory usage stack up. This is not because we have a memory leak in our program. That is the main reason the latter half runtime is not that accurate.

Hash Function Part:

1. Describe how each hash function works, and cite the source where you found this information. Your description of how the hash function works should be in your own words.

We found the hash function online at stackoverflow the website is:
http://stackoverflow.com/questions/8317508/hash-function-for-a-string
The first hash function just simply getting the ASCII value for all the characters of the input string and sum them up by using a for loop and use the total number to mod the size of the hashtable. That will generate a hash key by the first hash function. The second function is more complicated than the first one therefore we expect the number of hits are lower than the first hash function. The second hash function generates hash key for each word using a for loop with size of string of iterations. Before the loop, we initialize a value to 0. Within each loop, we let that value equal to the value times the certain number we used for seed and add the result with the ASCII value of word[i]. Lastly, use the special hash value that we calculated in the loop and mod the size of the hashtable which is the second argument that we enter in the command line. We believe the second one is better.

2. 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.

What we did for testing the function correctness of each hash function by writing some simple test case the input string will be one letter or few letters “a”, “ab”, and “apple”.

Test Case 1:

Hash Function 1

Input string = “a” table size = 100, the hash value that we calculated by hand is 97 (the ascii value of “a” is 97 and 97 % 100 = 97). When we run the program with the same input we get the return value of 97 which match our expectation. Just in case we run the same input on the same hash function three times we get the same hash value 97 for all three runs.

Hash Function 2 input string = “a” table size = 100, the hash value that we calculate by hand is (((131*0) + 97) % 100 = 97) When we run the program with the same input we get the return value of 97 which match our expectation. Just in case we run the same input on the same hash function three times we get the same hash value 97 for all three runs.

Test Case 2:

Hash Function 1

Input string = “ab” table size = 100 the hash value that we calculated by hand is 95 (the ascii value of “a” is 97 for “b” is 98 so (97+98) % 100 = 95). When we run the program with the same input we get the return value of 95 which match our expectation. Just in case we run the same input on the same hash function several times we get the same hash value 95 for all runs.

Hash Function 2 input string = “ab” table size = 100, the hash value that we calculate by hand is (((131*0) + 97*131 + 98) % 100 = 5) When we run the program with the same input we get the return value of 5 which match our expectation. Just in case we run the same input on the same hash function several times we get the same hash value 5 for all runs.

Test Case 3:

Hash Function 1

Input string = “apple” table size = 100 same thing with above we calculate the hash value by hand ((97 + 112 + 112 + 108 + 101) % 100 = 30) and the value that we get from running the program is also 30.

Hash Function 2

Input string = “apple” table size = 100 same thing with above we calculate the hash value by hand (((131 *0)+97) *(131 + 112)*(131 + 112)*(131 + 108)*(131 + 101) % 100 = 30) and the value that we get from running the program is also 30. We also run the program with the same input value several times to make sure that the hash value does not change.

The above test are only based on determine the hash function is working or not. We are required to test the whole class. Therefore, step 3 will inspect the correctness of the whole function. On how to handle dictionaries with a lot of words. I believe that we are follow the writeup instruction that will lead us to the correct solution.

3. 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.

Test Case 1:
Input Data: ./benchhash shuffled_freq_dict.txt 1000
Printing the statistics for **hashFunction1** with hash table

# Hits	# Slots Receiving # Hits
0	1252
1	547
2	158
3	38
4	3
5	1
6	1

The average number of steps for a successful search for hash function 1 would be 1.315
The worst case steps that would be needed to find a word is 6

Printing the statistics for **hashFunction2** with hash table size 2000

# Hits	# Slots Receiving # Hits
0	1213
1	607
2	149
3	29
4	2

The average number of steps for a successful search for hash function 2 would be 0.624
The worst case steps that would be needed to find a word is 4

Test Case 2:
Input Data: ./benchhash freq_dict.txt 1000
Printing the statistics for **hashFunction1** with hash table

# Hits	# Slots Receiving # Hits
0	1289
1	509
2	140
3	43
4	15
5	2
6	2

The average number of steps for a successful search for hash function 1 would be 1.409
The worst case steps that would be needed to find a word is 6

Printing the statistics for **hashFunction2** with hash table size 2000

# Hits	# Slots Receiving # Hits
0	1211
1	605
2	162
3	18
4	3
5	1

The average number of steps for a successful search for hash function 2 would be 0.622
The worst case steps that would be needed to find a word is 5

Test Case 3:
Input Data: ./benchhash freq1.txt 1000
Printing the statistics for **hashFunction1** with hash table

# Hits	# Slots Receiving # Hits
0	1330
1	449
2	146
3	50
4	20
5	3
7	2

The average number of steps for a successful search for hash function 1 would be 1.488
The worst case steps that would be needed to find a word is 7

Printing the statistics for **hashFunction2** with hash table size 2000

# Hits	# Slots Receiving # Hits
0	1213
1	600
2	165
3	18
4	4

The average number of steps for a successful search for hash function 2 would be 0.6215
The worst case steps that would be needed to find a word is 4

Test Case 4:

Input Data: ./benchhash freq2.txt 1000
Printing the statistics for **hashFunction1** with hash table size 2000

# Hits	# Slots Receiving # Hits
0	1402
1	366
2	130
3	54
4	36
5	7
6	3
7	1
8	1

The average number of steps for a successful search for hash function 1 would be 1.672
The worst case steps that would be needed to find a word is 8

Printing the statistics for **hashFunction2** with hash table size 2000

# Hits	# Slots Receiving # Hits
0	1211
1	607
2	158
3	20
4	3
5	1

The average number of steps for a successful search for hash function 2 would be 0.623
The worst case steps that would be needed to find a word is 5

Test Case 5:
Input Data: ./benchhash freq3.txt 1000
Printing the statistics for **hashFunction1** with hash table size 2000

# Hits	# Slots Receiving # Hits
0	1520
1	239
2	95
3	76
4	33
5	22
6	8
7	3
8	4

The average number of steps for a successful search for hash function 1 would be 2.036
The worst case steps that would be needed to find a word is 8

Printing the statistics for **hashFunction2** with hash table size 2000

# Hits	# Slots Receiving # Hits
0	1207
1	617
2	147
3	27
4	2

The average number of steps for a successful search for hash function 2 would be 0.62
The worst case steps that would be needed to find a word is 4

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

Before the testing and data collect for both hash function, by the time we found those two hash function online we already have an idea about which hash function is better. We have learned that in CSE 12 basic data structure the hash function that will generate a more unique hash key is the better hash function. By comparing the those two hash function it is absolutely that the second hash function will generate a more unique hash key for the input string. Therefore the second hash function is better than the first one. According to our test data in the step three above, we can see that the max number of hits of the second hash function is always smaller than the first hash function with the same input. For example, on test case 5 the first hash function have a maximum hit number of 8, but the second hash function only have a maximum of 4. This is because the second hash function can handle collision better than the first one. Also the average number of the steps for successful search for hash function 2 is much smaller than hash function 1. It is clear that the second function is better than the first one. In conclusion, the test result matches our expectation perfectly.