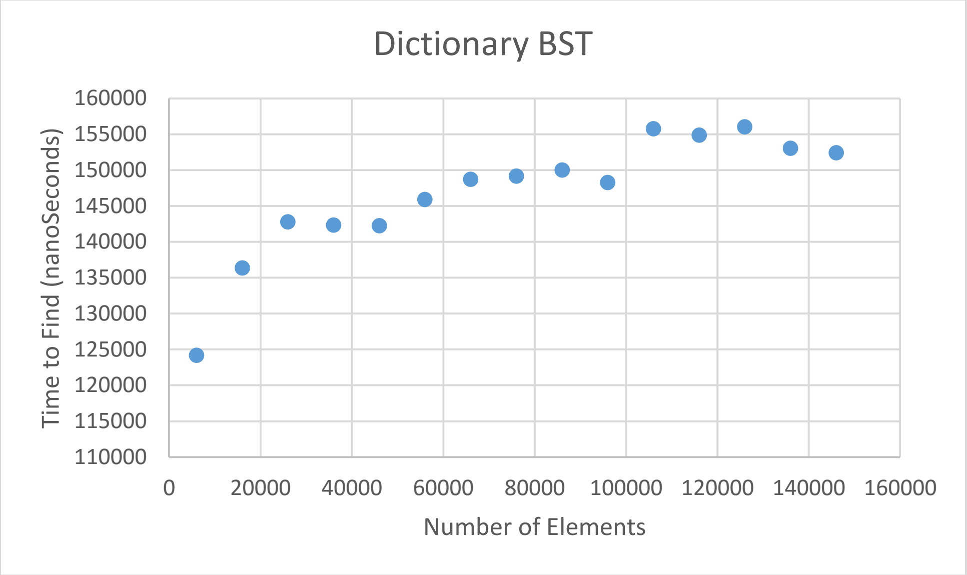
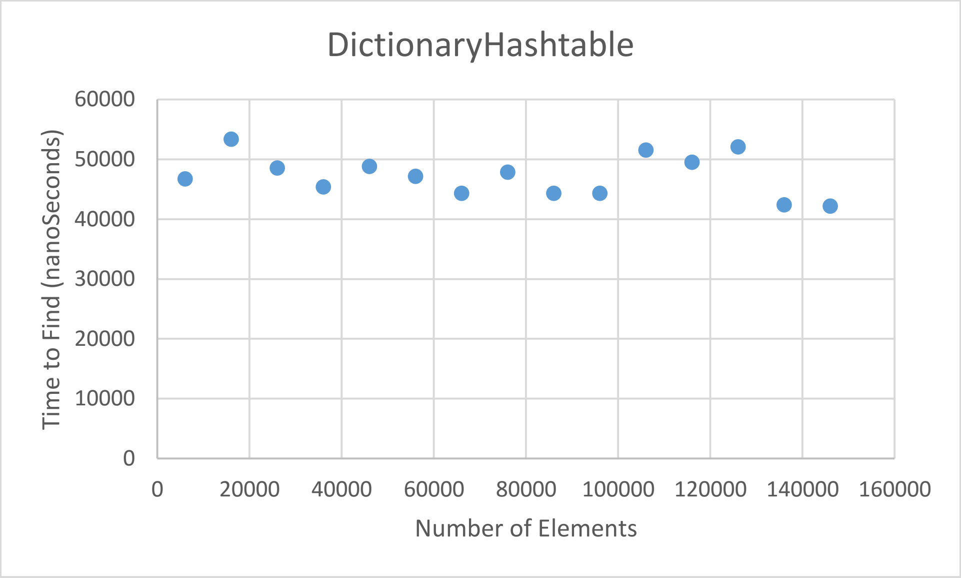
DictionaryBST

6000	124169
16000	136325
26000	142792
36000	142307
46000	142239
56000	145883
66000	148723
76000	149163
86000	149975
96000	148260
106000	155744
116000	154863
126000	156025
136000	153050
146000	152399



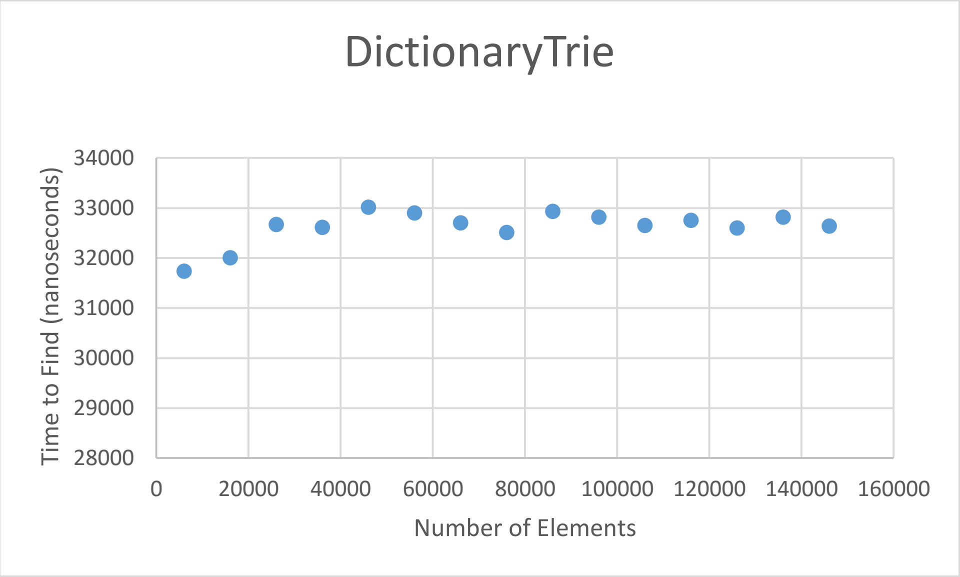
DictionaryHashtable

6000	46704
16000	53343
26000	48525
36000	45383
46000	48798
56000	47176
66000	44304
76000	47832
86000	44329
96000	44301
106000	51557
116000	49533
126000	52070
136000	42401
146000	42155



DictionaryTrie

6000	31740
16000	32004
26000	32671
36000	32615
46000	33016
56000	32905
66000	32707
76000	32513
86000	32935
96000	32821
106000	32654
116000	32757
126000	32600
136000	32817
146000	32639



1&2) Running time expectations:

For a hash table there should be a constant look up time O(1), the element is hashed to a location, and then you look to see if the element is there. The results show this is true on page two, if you were to draw a line through all the points it would be horizontal.

Next, the Dictionary BST should have log(N) look up time, because it is a balanced binary search tree, and worst case it would have to span the whole height of the tree, the height being log(N), the data shows this, if you were to fit a curve to the data, a log curve would fit it nicely.

Finally the DictionaryTrie look up time is independent from the number of elements, rather it is the length of the word that dictates the look up time, it has to go from node to node, letter to letter seeing if the whole word is in the trie structure, so, if you have a million words, but all the words are 10 letters long, it would take the same amount of time to find a word if only 10 words were loaded in, and all the words were 10 letters long. So, the worst case, is the longest word in the set.

3) The algorithm I used to implement the predictCompletions method. First, I added a maxFreq element to each node in my tree, the maxFreq element of a node keeps track of the biggest frequency of all its children. First step of the algorithm, find the prefix in the tree, if its not in the tree, return an empty array of words. After the position is found, add its middle child to a priority queue that sorts based on the maxFreq element. Then pop the top of the maxFreq priority queue, if the top is a word add it to another priority queue of words that sorts based on the frequency of that particular word, not the max frequency. Then add all of its children nodes (left right middle) to the maxFreq priority queue, and keep popping the maxFreq priority queue til the word priority queue has reached the desired size, or if the maxFreq priority queue is empty. Finally add the prefix to the word priority queue if the prefix is a valid word. Finally pop the number of words the user was looking for or until the word priority queue is empty. The amount of time it takes for the algorithm to complete, it doesn't matter how many words are added to the tree, but rather the deepest the tree goes, so, the length of the word, and then the number of words that you are looking for, so if you are looking for E number of words, and they are all length L, then it would take O(EL), but the number of words in the trie doesn't matter.