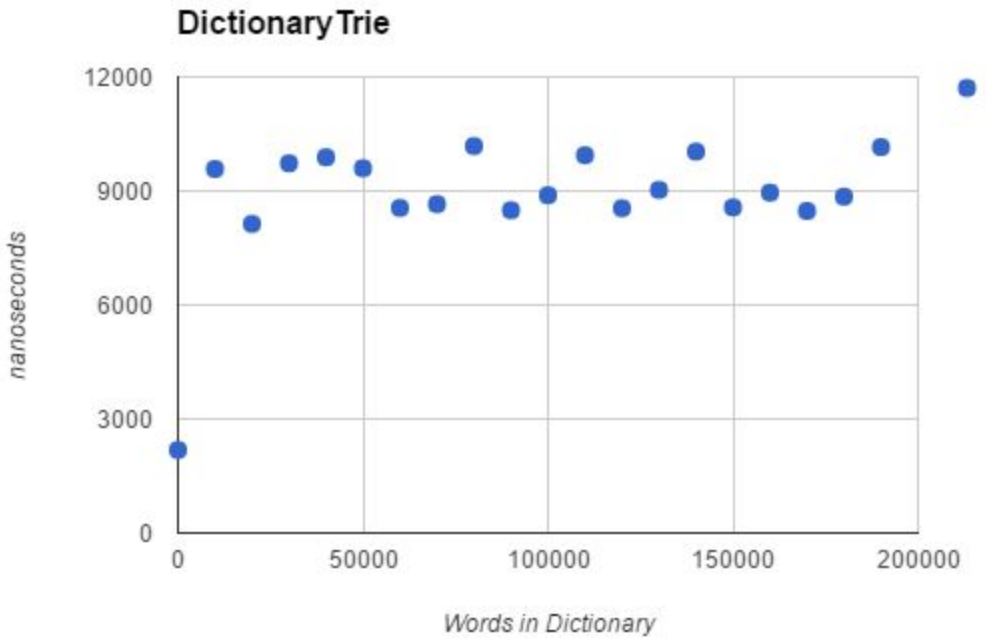
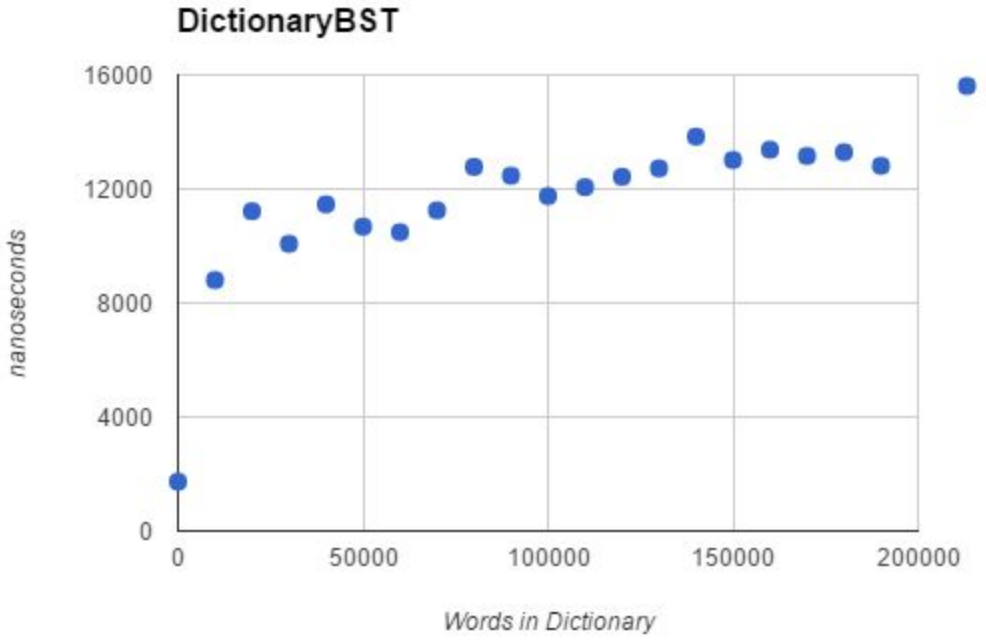
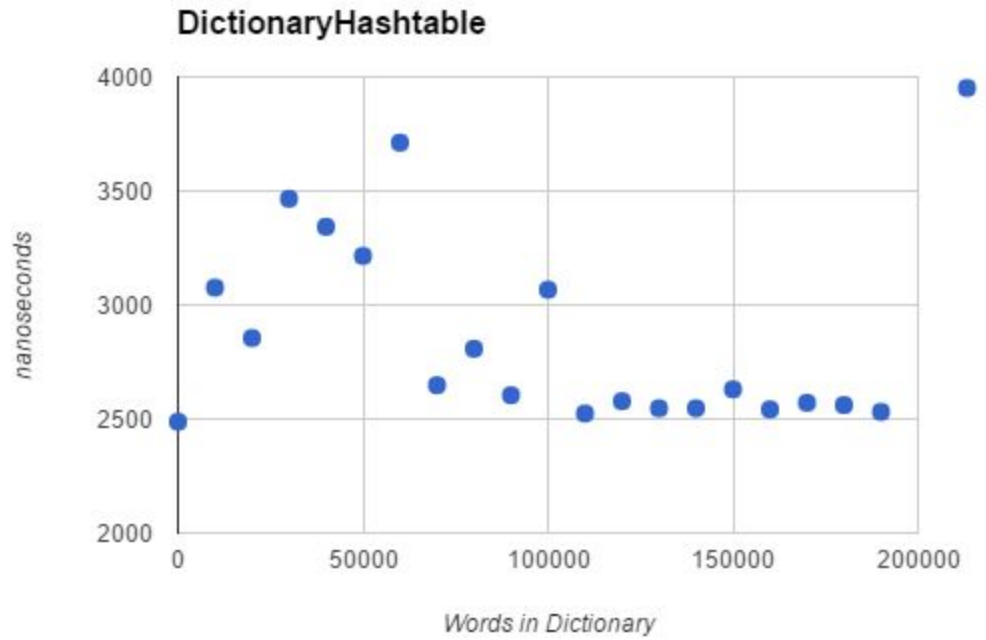
Lucas Blumhardt
A1202745

```
./benchdict 0 10000 20 shuffled_freq_dict.txt
DictionaryTrie
0       2184
10000   9579
20000   8134
30000   9728
40000   9886
50000   9598
60000   8554
70000   8654
80000   10182
90000   8494
100000  8888
110000  9941
120000  8546
130000  9028
140000  10036
150000  8567
160000  8956
170000  8474
180000  8851
190000  10155
DictionaryBST
0       1731
10000   8805
20000   11219
30000   10075
40000   11460
50000   10680
60000   10481
70000   11250
80000   12776
90000   12465
100000  11752
110000  12064
120000  12429
130000  12724
140000  13737
150000  13018
160000  13379
170000  13159
```

180000 13290
190000 12814
DictionaryHashtable
0 2488
10000 3076
20000 2855
30000 3466
40000 3343
50000 3215
60000 3712
70000 2648
80000 2808
90000 2604
100000 3067
110000 2524
120000 2578
130000 2547
140000 2547
150000 2630
160000 2542
170000 2571
180000 2561
190000 2531







N = number of words
C = number of unique characters

D = word length

1.
DictionaryTrie:

I implemented a Multiway Trie for the my DictionaryTrie class. Find() in a MWT should depend only on the length of the word you're finding, D. This is because the find method just has to iterate through the word to find which index to traverse from the current node to a child. Therefore find() for DictionaryTrie should be O(D)

DictionaryBST:

DictionaryBST uses a set, or a balanced BST class. The words are stored in BST nodes, so find would take the standard BST find time which is O(log elem) where elem is the number of elements in the BST. The DictionaryBST holds words so elem is number of words. Thererfore find() for DictionaryBST should be O(log N).

DictionaryHashtable:

DictionaryHashtable utilizes an unordered set which is essentially a hashtable. These have constant find times in the average case, but worst case depends on how they resolve collisions. cplusplus.com's page on unordered sets doesn't state how the unordered set RESOLVES its collisions. If it were through linear probing, find could take up to O(N). We don't know the hash function and how it handles the words in the dictionary.txt files. Let's assume the hash function is decent and there aren't a crazy amount of collisions that compromise its average case find. Therefore find() for DictionaryHashtable should be O(1).

2.
DictionaryTrie:

Because this find() method depends on the word's length, it's hard to tell much about the accuracy from the graph because word length wasn't a variable All lookups took around the same amount of time. In my benchmark testing I asked DictionaryTrie to find invalid words that were all around the same length. That would mean our times should be all around the same length. For that reason I think the graph is reason to believe my find() method has the correct runtime, but not sufficient evidence.

DictionaryBST:

The DictionaryBST graph looks like a model log function. The x-axis is number of words in the dictionary, and the bst's find() function is supposed to take O(log (number of words)) time. Therefore the graph proves DictionaryBST to have accurate results.

DictionaryHashtable:

This graph's results are hard to tell if the method is accurate to the predicted runtime. The first half of the graph seems random, but the second half looks like constant runtime. My prediction is that the servers were busier for the first half of the tests than the second.

3.

My algorithm is a standard depth-first search. This traverses every node with a stack and adds the nodes that are words to a max heap. The runtime of DFS is O(Vertices + Edges). This value is hard to determine for a graph that is a multiway trie. It's pretty much every unique character at a certain index. It's impossible to represent that in the variables N C and D without knowing the dictionary we are referring to (and even then it'd be pretty impossible by hand). The number of nodes and edges is dependent on the number of words though, so I will simplify DFS runtime to O(N).

Along with traversing nodes, we have to take into account the constant reordering the heap is doing. Reordering the heap is O(log (number of words in the heap)). The words going into the heap are words with the specified prefix, so I will simplify this to O(log N).

Therefore the runtime for my predictCompletions algorithm is O(N log N).

My method is very simple compared to the way I was first implementing it. I had to give up because two days of work had gotten me nowhere and I couldn't handle all the debugging. I switched to a standard dfs algorithm, but even that has been imperfect. My logic is fine I believe. There must be something wrong with either my insert method or my heap comparator. I'm working on that until the deadline, but turning this pdf in now.