Jae Song, A12042160

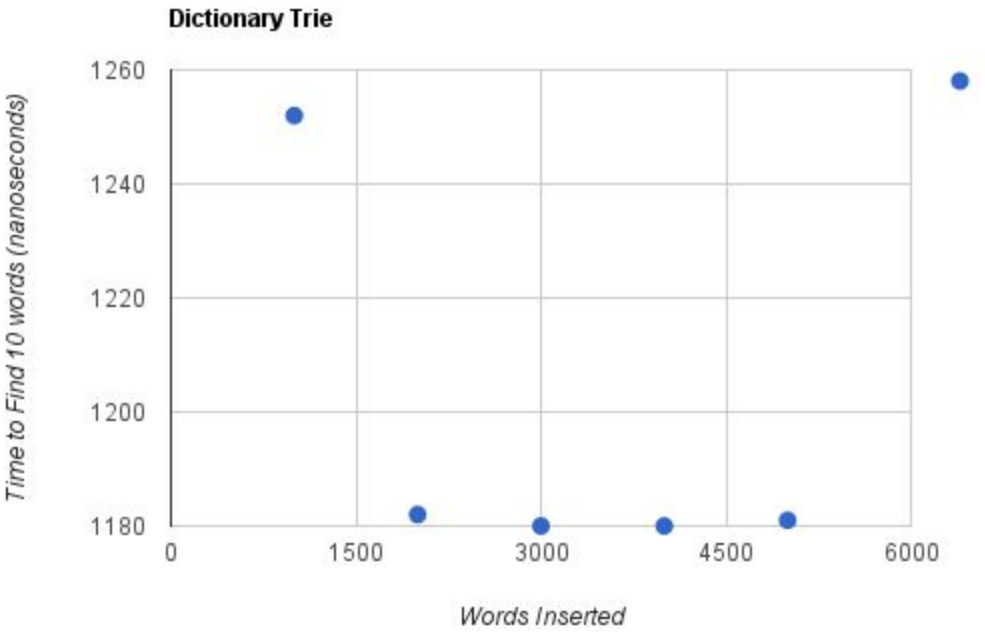
Professor Alvarado, CSE 100 PA 3

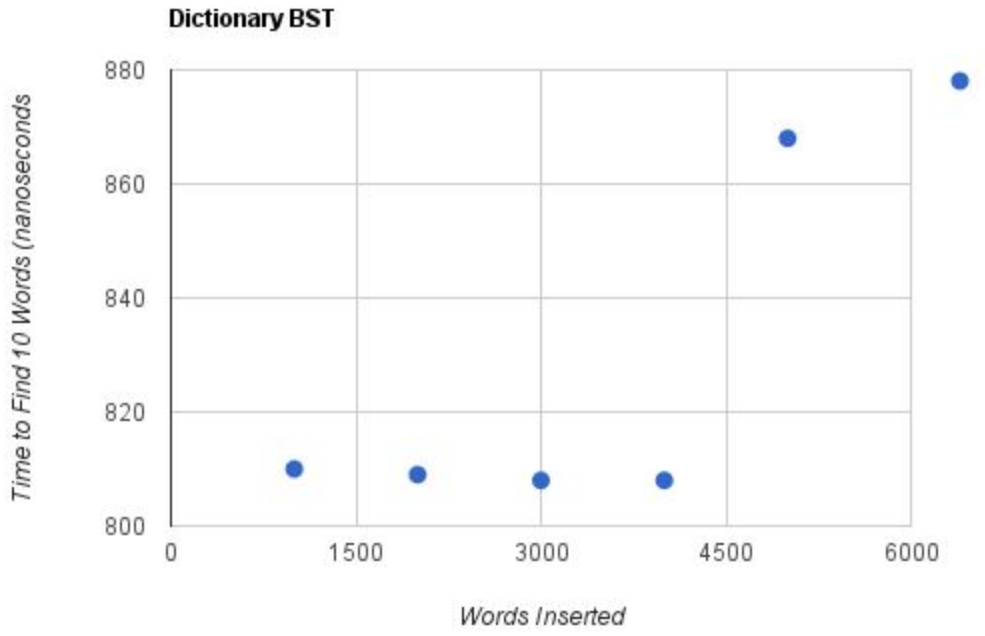
Final Report

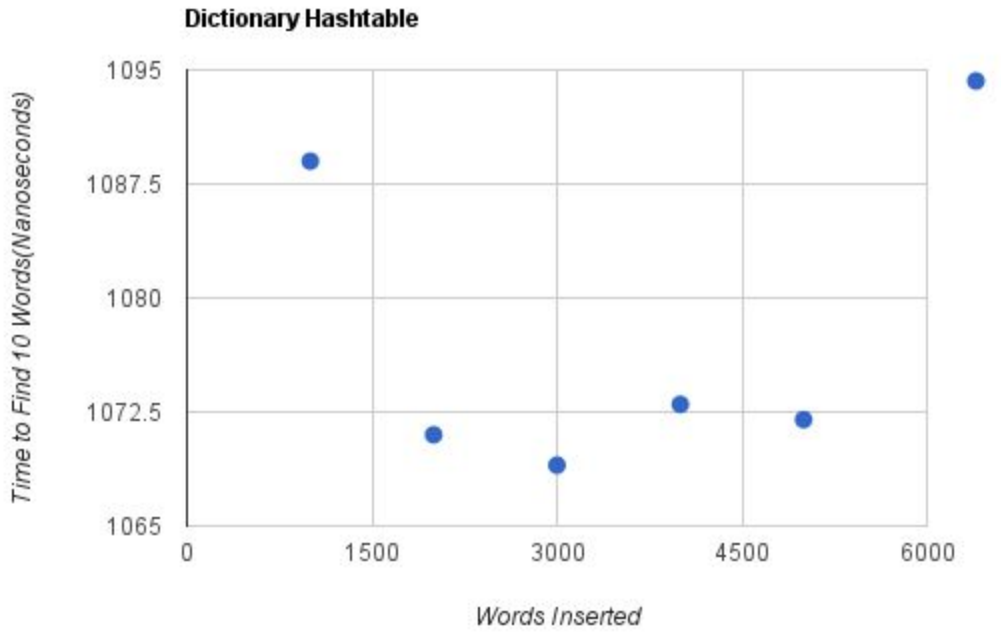
1) I expect the Dictionary_BST to run at a runtime function of O(Log(N)). We know that the BST used in the c++ set is a balanced BST so the worst case time would be Log(N). Because each node either has two children or no children at all, every time a decision to go either right or left is made, essentially half of the nodes can be excluded in the search. So dividing by half every time you move until you reach the goal would essentially make the function of balanced BST's runtime to be **O(Log(N)).** The Dictionary_Hashtable should be a runtime function of O(1). That is the essential feature of a hashtable, its find function's should be a runtime function of **O(1)**. Because the Dictionary_Hashtable was implemented as a unordered_set of strings, it should not be a mathematical function of N, C, or D but just simple constant runtime as it will turn each string into a hash code. I would assume that C++ STL does a good job of this, but if there are collisions, then the worst case runtime can be O(N). But this should be very unlikely as that would defeat the purpose of hashtables and the C++ STL would do a good job of hashing and alleviating collisions but it is still possible. Finally my Dictionary_Trie class would at worst take O(D). I have a unordered_map as a member variable of TrieNode, it would look up each corresponding char and continue to go until it got to the last char or until the Trie has no more children. So the runtime for each unordered_map find for one char should be O(1), and the word length is D, so combined would be **O(D)**. Again, this is mostly extra, but in some rare and absurd case where C++ STL's unordered_map has collsions for 27

characters, then the runtime could be O(D*N), but this would be very very unlikely, and most likely not true.

2)







The data I've retrieved were not consistent with the predictions that I made. I've predicted DictionaryHashtable to take the shortest time with O(1), but the results were not aligned with the prediction. A reason why may be because of the large number of people that is running the program on the server; it could also be because of collisions in the hash table, but again I would assume the C++ STL hashtables would handle collisions well so that the runtimes would not be affected. Also my DictionaryTrie took the longest, I did expect Dictionary Trie to take longer than hashtables since it looks up from a hashtable D(length of word) times, but I did not expect it to run slower than DictionaryBST. Again this could be due to collision or maybe the busy servers' slowing of the programs. I've also tested the benchdict program with 100,000 min_size, 500,000 step_size, 5 iterations, and freq_dict.txt, but the numbers were very close to what I have in the graphs. Now in this case, I would expect Dictionary Hashtable and Dictionary Trie to take

around the same time since the word I'm looking for does not change so D doesn't change, and increasing words inserted would only make the hash table bigger, but the hash function should still take around the same time. Dictionary BST gave back similar value on this input as well which leads me to strongly believe that it is the slowing down of the servers as a Balanced BST should have taken a lot longer as more word were inserted. All in all, I believe that Dictionary hashtable should be O(1), Dictionary Trie should be O(D), and Dictinoary BST should be O(Log(N)) and the reason why the data is not aligned with my prediction could be due to the slowing down of the servers and collisions.

3) For my Dictionary_Trie's predictCompletions the runtime would be O(log(N)). The Algorithm goes like this. I have a counter called arbitrary size that starts out as number of completions. Then, I would start with the top node(node with the prefix), and insert it to a priority queue that compares the maxFreq (max freq number of the subtree). I also insert it into the priority queue that pops out the highest frequency if it's a word, and increment arbitrary size by 1. It will then add its children as well into the priority queue that compares maxFreq, so when I pop it again it will pop out the one with the same maxFreq since that's the one with the biggest maxFreq as it was maxFreq of the parent. I will then use this priority queue to go through the tree to find a TrieNode with a freq that matches maxFreq. On the way to find that particular TrieNode, I will add all the visited TrieNode(words or not) into the priority queue that compares maxFreq and TrieNodes that are words into both of two different priority queues, one that pops out the highest frequency node and one that pops out smallest frequency node. I will insert them if the size of those queues are less than the number of completions I have to return or if the

frequency of the visited TrieNode is greater than the smallest one already in the priority queue(that's why I have the priority queue that pops out the smallest). And I increment the counter by 1. The only time I don't increment my counter is when I actually find the maxFreq. That's already numerous instances of O(Log(N)) for pushing and popping into the priority queue, however I stop this loop once the size of the priority queue has reached arbitrary size, or if I run out of nodes in the priority queue that compares maxFreq(however it will most likely stop when size reaches arbitrary size), and since I don't increment my counter only when I find the TrieNode that matches maxFreq, it guarantees I will at least find all the possible words with a constant number of number of completions, or just possible words if there are less than number of completions. Then the runtime becomes, O(K*Log(N)) where K is some constant time, or more specifically the number of completions given by the argument. Since in Big O notation constant multipliers can be dropped, the runtime becomes O(Log(N)).