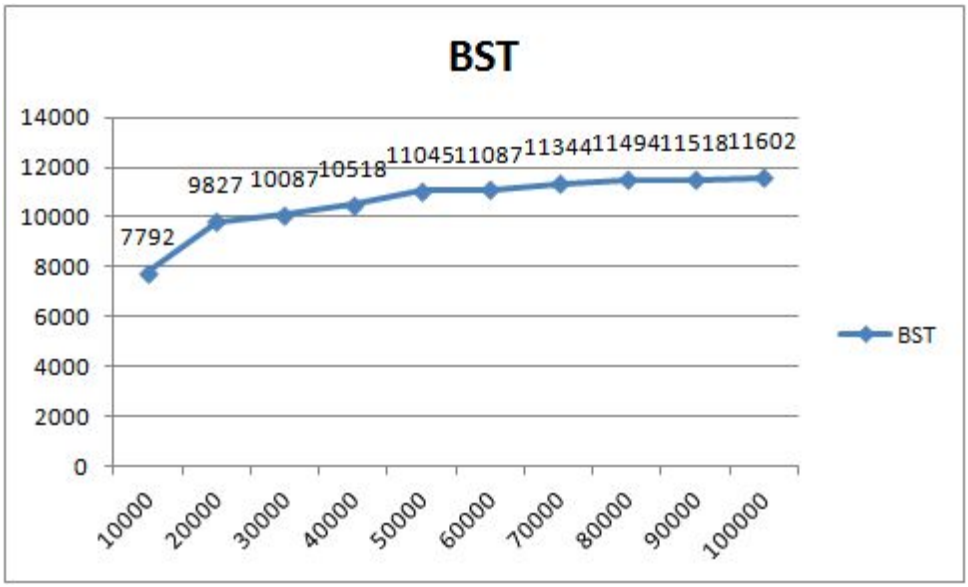
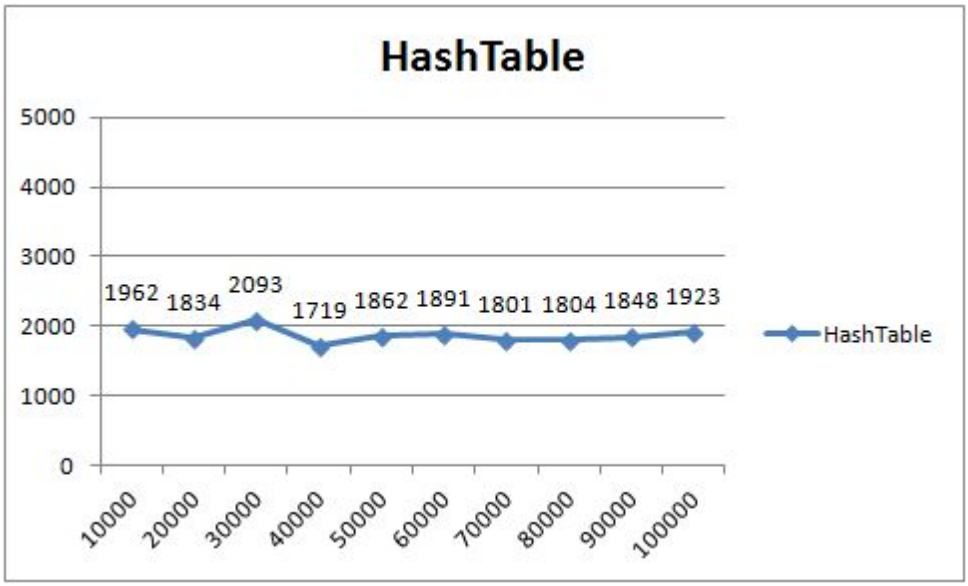
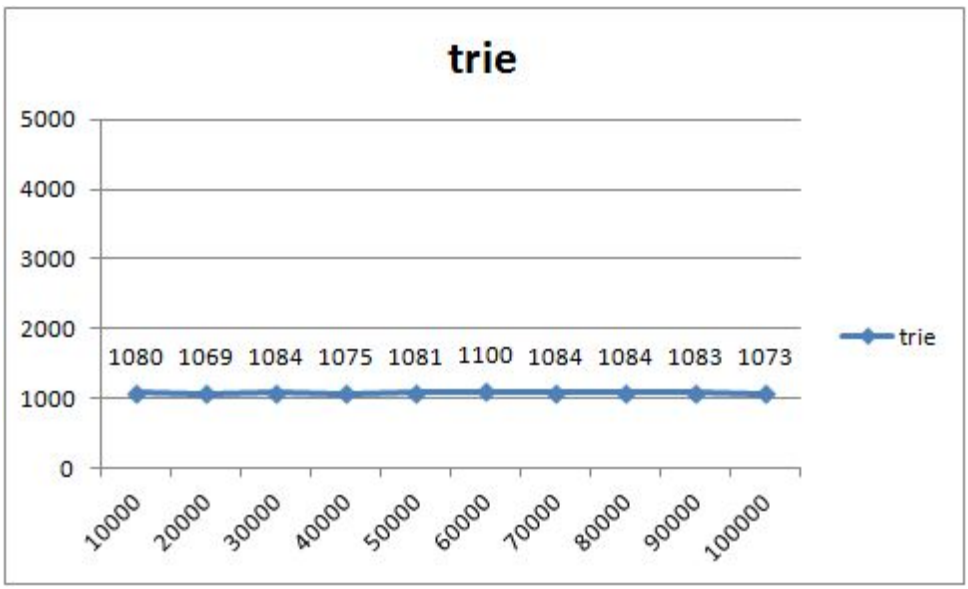

BST
14000
12000
10000
8000
6000
4000
2000
0
7792
9827
10087
10518
11045
11087
11344
11494
11515
11602
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
BST


HashTable
5000
4000
3000
2000
1000
0
1962
1834
2093
1719
1862
1891
1801
1804
1848
1923
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
HashTable


trie
5000
4000
3000
2000
1000
0
1080
1069
1084
1075
1081
1100
1084
1084
1083
1073
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
trie

1. What running time function do you expect to see for the find method in each of your three dictionary classes as a function of N (number of words), C (number of unique characters in the dictionary) and/or D (word length), and why? This step requires mathematical analysis. Your function may depend on any combination of N, C, and D, or just one of them.

BST

BST's average-case time cost for finding is **$\Theta(\log_2 N)$**. The best case is Θ(1) and worst case is Θ(N). N is number of words here. We just focus on average case here.

At the worst case of finding, words might be inserted in order or reverse order. Then the tree will look like a straight line to the right or to the left. Therefore, when we are trying to find a word that's on the bottom of the tree. We will go through all the words in the tree and down to the bottom. This will take N comparisons, so the worst case takes Θ(N). Since the dictionary in our case won't insert words like this pattern, the worst case is not likely to happen.

At the best case of finding, the word we are trying to find is on the top of the tree, or the only one word in the tree. Then we only take 1 comparison to finish finding, so best case is Θ(1). Since the dictionary in our case won't just insert one word or just finding the word on top all the time, the best case is not likely to happen.

At the average case of finding, The run time directly relates to number of compares needed to find any word. And number of compares to find a key depends on location (specifically depth) of the key in a BST and structure of the BST. Given a BST having: N nodes $x_1,...x_n$ such that key(x_i) = k_i, probability of searching for key k_i is p_i. The expected number of comparisons to find a key is $\sum p_i$*(No. of comparisons to find k_i). Then we make the first assumption for average case: All keys are equally likely to be searched. Thus p1 = ... = pN = 1/N and the average number of comparisons in a successful find is: $D_{avg}(N) = (1/N)* \sum d(x_i)$ = (total depth of the tree)/N. However, the structure of BST will influence the total depth of BST. Then we need to make second assumption for average case: Any insertion order of the keys is equally likely when building the BST. Then let D(N) be the expected total depth of BSTs with N nodes, over all the N! possible BSTs, assuming that Probabilistic Assumption #2 holds: D(N) = $\sum$ (probability of T_j)(Total Depth(T_j)). If Assumption #1 also holds, the average # comparisons in a successful find is $D_{avg}(N)$ = D(N)/N. To compute D(N), we make a recurrence relation: D(N|i) = D(i) + D(N − i − 1) + N. Solving this equation, finally we get N*D(N) = (N+1)*D(N-1)+2N-1⇒D(N) = 2(N+1)$\sum$1/i - 3N⇒$D_{avg}(N) \approx 1.386 \log_2 N$. So the average case of finding is $\Theta(\log_2 N)$.

HashTable

HashTable's average-case and best case time cost for finding are both **Θ(1)**. The worst case is Θ(N). We just focus on average case here.

At the worst case finding, if the hashtable uses a seperate chaining and the hash function is not a perfect one or the keys used to insert happen to go to the same place. Then when we are trying to find the last element of the linked list in that place, then we need to go through all the keys in HashTable to find the specific key. This takes Θ(N). However, this is very unlikely to happen if we have a good enough hash function. Since we're using the build in unordered_set in C++. We don't have to worry about the worst case. So the expected time complexity for HashTable here is just Θ(1). When we have a good hash function and a HashTable with reasonable load factor so that there won't be too much collision. We can

simply find a word(in our case) from the HashTable because we know the position of it based on the hash function. If no such position exist, we fail to find the word but it still takes constant time.

Trie

Trie's time cost for finding is **Θ(D)** for all the case. D is the word length of the word we want to find.

Since we implement our Trie with multiway trie, we create a TrieNode class which has pointers to other TrieNodes as children. When a node is pointing to a specific position of its child node, this means the word appends one specific character. We can find a word by just simply go down the trie from root character by character. Based on the single character of the word, we will know where to go from a node. We need to find next node D times and see whether the frequency is 0 or not. If the frequency is not 0, then we find the word.

2. Are your results consistent with your 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.

BST

Yes. BST's time cost for finding is **Θ ($\log_2$N)**. The graph of BST has horizontal axis of the number of words that we have loaded and the vertical axis of the time(in nanosecond) that BST takes to find 10 words in worst case(words not in dictionary). From the graph of BST above, we can see the graph appears to be like the pattern of $\log_2$N because as the number of words in dictionary increases, the time it takes to find increases but in a slower pace as N gets larger.

HashTable

Yes. HashTable's time cost for finding is **Θ(1)**. The graph of HashTable has horizontal axis of the number of words that we have loaded and the vertical axis of the time(in nanosecond) that HashTable takes to find 10 words in worst case(words not in dictionary). From the graph of HashTable above, we can see the graph appears to be like a straight and horizontal line, which means the time that HashTable takes to find a word is constant. There's some minor difference between the times. This might be caused by some outliers that would influence the time of finding in HashTable. Those outliers can be collisions, computer system issues etc. But overall this graph make sense because the time of finding in dictionaries with different size is mostly constant.

Trie

Yes. Trie's time cost for finding is **Θ(D)**. The graph of Trie has horizontal axis of the number of words that we have loaded and the vertical axis of the time(in nanosecond) that Trie takes to find 10 words in worst case(words not in dictionary). From the graph of the Trie above, we can see the graph also apeears to be like a straight and horizontal line, which means the time that Trie takes to find a word is constant. Since we use the same words with same length to find each time, according to the time complexity we expected, the graph should

give us an horizontal line since D is treated to be a constant in this case. Therefore, the graph of trie makes sense for our expectation.

3. Explain the algorithm that you used to implement the predictCompletions method in your dictionaryTrie class. What running time function do you expect to see for the predictCompletions as a function of N (number of words), C (number of unique characters in the dictionary) and/or D (word length), and why? This step also requires mathematical analysis. Your function may depend on any combination of N, C, and D, or just one of them.

In the predictCompletions method, we first use the same logic that we used in find method to advance TrieNode curr to point to the node which is the end of the prefix. Then, we create a priority queue(min_heap) that hold pair(string and int). We first insert num_completion times of empty pair into the pq so that we can have some int to compare with in our helper method. After that, we call our searchHelper method to recursivly visit all the node that fit our design. In our design of priority queue, the min frequency pair is at the front of the pq, and max frequency is at the end of the pq. So, everytime we run the helper method, we check if the maxCount of the TrieNode is bigger than the pq's top elment's frequency. If it is larger, we know that we need to visit that node and its child and push the pair if the current prefix's frequency is greater than zero. If it is smaller, we know that we do not need to visit that node. By doing this, we can save a lot of time comparing to the exhaustive search which visit all the node and its child no matter what.

Θ(D)

In our method, the word length plays an important role in run time analysis, since we determine if we will go to one level below if the current node's maxCount is greater than the min frequency of the priority queue. In TrieNode class, the maxCount is the max Frequency of the current node's subtrie. As a result, all the TrieNodes that are with in the word path will have the same maxCount. Thus, when we go to visit one level down from current node, the updated current node still has the same maxCount which is still bigger than the min frequency of the priority queue. And we will have to visit all the node that are in the word path. Therefore, our method's run time will depend on the lenght of the word.