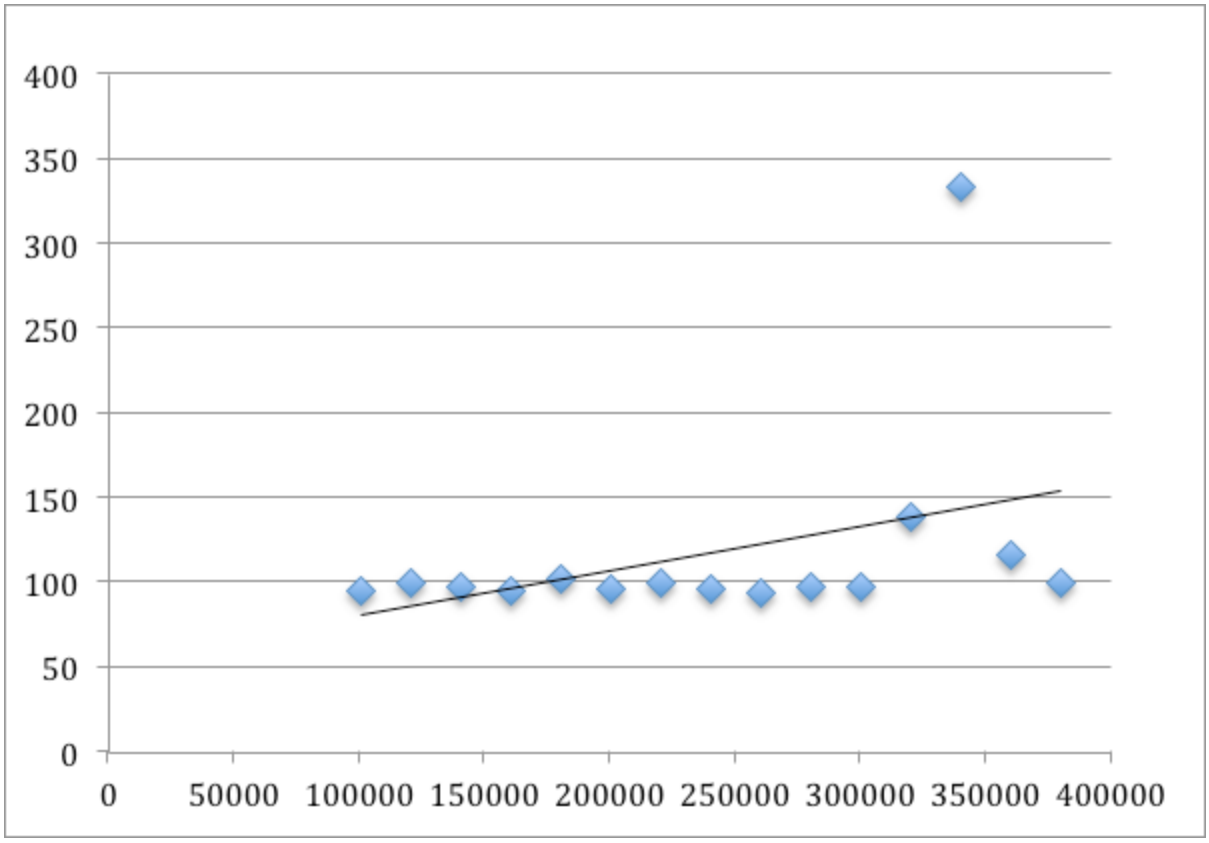
Names: Louis Leung, Kyle Sit

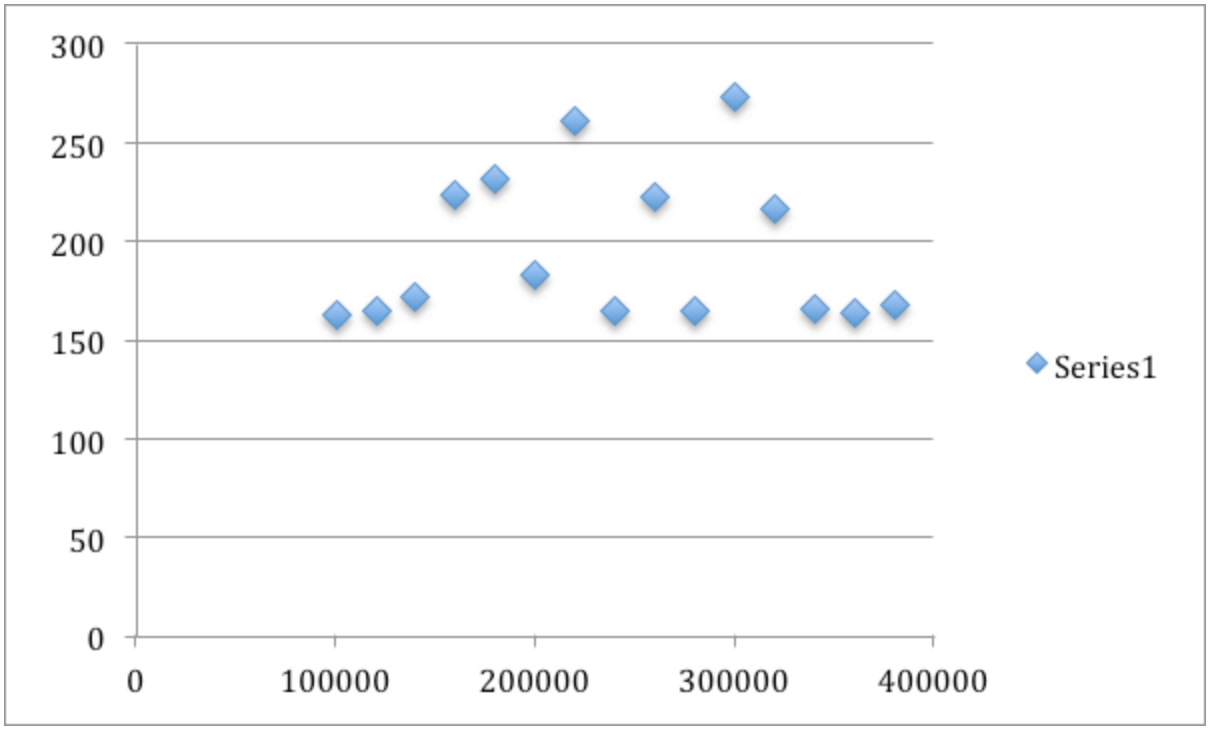
Dictionary Trie:



X Axis: Lines Read

Y Axis: Nanoseconds taken to fail to find 10 words averaged through 50 times.

Dictionary BST

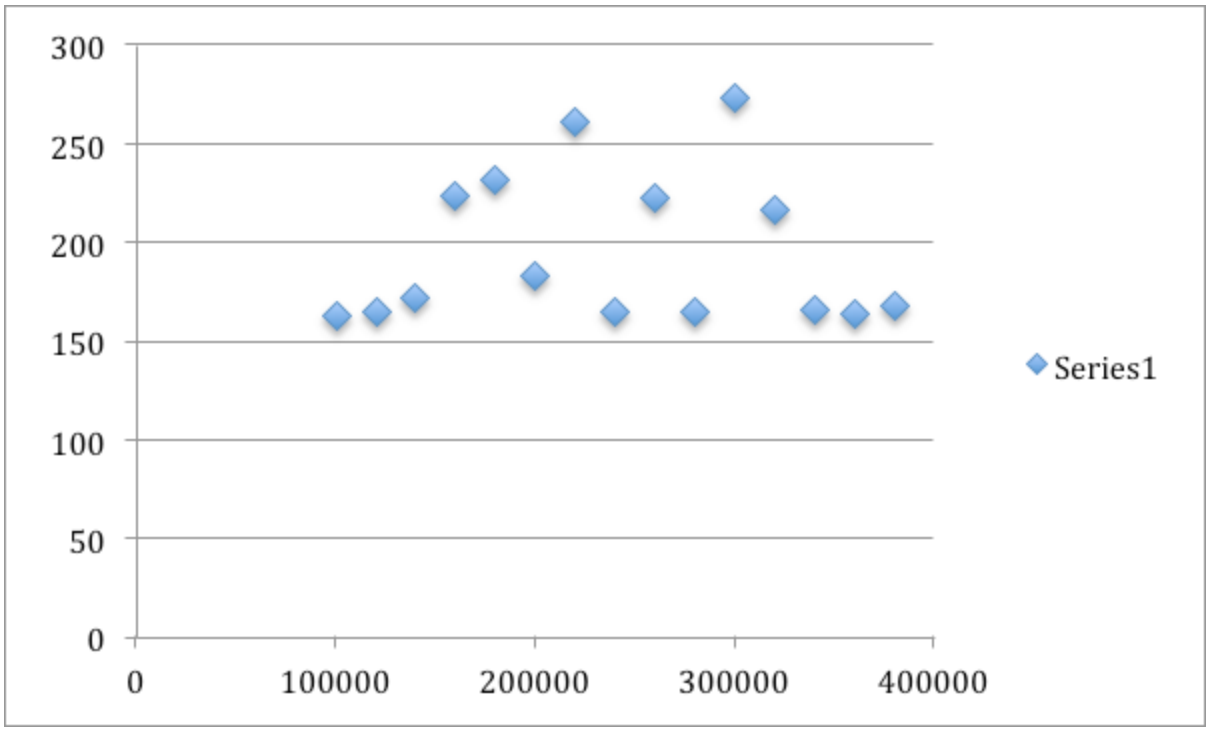


X

Axis: Lines Read

Y Axis: Nanoseconds taken to fail to find 10 words averaged through 50 times.

Dictionary Hash



X Axis: Lines Read

Y Axis: Nanoseconds taken to fail to find 10 words averaged through 50 times.

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: Since our binary search tree is balanced, we only need to complete log N comparisons before we are sure we failed, because when n = height of the tree the total number of nodes is 2^n – 1, where the root is at height 1.

Therefore, the runtime should be O(log N)

Hash: Searching for something in a hash always has a runtime of O(1).

Trie: Since the find method in our trie basically goes takes each letter from our word and goes follows the trie down until we find a null node pointer where we shouldn't or we have found the completed word at a leaf, the find method should be a function of D, the word length.

1. 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.

Yes and no.

For our trie, this did meet our expectations. Because regardless of how many number of lines we had read to create our multi way trie, the length of the letters in the word (or 10 words in our case) D stayed constant, therefore our find time stayed around constant. The random spike was probably just due to some external process on the server.

For hash, this met our expectations as well. Hash should always take constant time and isn't a function of how many dictionary entries we read in.

For BST, we were kind of confused since this didn't seem to follow a O(log N) type format, and we actually dropped in search time for the highest number of entries. Really, we're just attributing this to random noise in the servers and random noise. Stuff happens. We ran this a ton of times and couldn't come up with a better explanation please give us points.

In general, it was interesting that the hash was slower than our Trie, since hash should always be a constant low value since we're just checking if a key exists. We're attributing this to possibly how our hashing find algorithm handles collisions – maybe if it doesn't find the word at a spot it goes through extra steps to check the rest of the hash.

1. Explain the algorithm that you used to implement the predictCompletions method in your odictionaryTrie 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.

Originally we tried to use a BFS method that would only search through the trees with the largest frequency nodes however we could not get it to work correctly. We ended up going with a typical BFS. We expect a runtime of O(D * N) if D is the average word length since we will search through every node of every word.