CMSC 221 - Fall 2017 - Programming Assignment 2

Ryan Namgung

November 5, 2017

Introduction:

The objective of this assignment was to analyze time complexities of five different data structures and analyze the time complexities of the major operations of these structures. In this assignment, I learned the implementation of the methods of the BST functions as well as the creation of unit tests and timing tests of different data structures. My results during my assignments basically showed me exactly what I learned in class. I created a test to find the time complexities of each data structure and from here I got the $n0$ and c values of each graph. Also, I wrote out the expected time results in asymptotic notation.

Implementation Details:

Timing: For the timing implementation, I made a timing of the I basically used the timing done in Programming Assignment 1 as a basis. These timings were done for the put method of each data structure. For the Unsorted Table Map, because it doesn't really matter where we put the key I ran an average case filling the map for $i < k$. For the Best Case Sorted Table Map, we inserted at the front every time. For worst case and average case Sorted Table Map, we put the value at a key where we would have to shift through i keys every time. For our Chain Hash Map the average case was inserting at i every time, this returns log n, because some of these values could have collisions and some of them won't. The average case for the Probe Hash Map is inserting at i every time, for the same reasons. Now for the BST Best case, we created an implementation where the values would input would balance the tree. For the BST average case, we put at random key values causing the tree to be randomized. Finally for the BST worst case, we created a situation where we keep putting values where values go down a single child of the root causing it to run in $O(n^2)$ time.

Unit Tests: The unit test implementation was relatively straightforward considering we just made test cases for each function of all data structures. The get just had us fill in the map with values and getting a value at a specific key. The put function similarly had us input values and then we got the value at a specific key to see if it was the one we input. Remove returned the value of the key we got rid of and also decreased the size by one. Finally, the TestEntrySet method created an entry set and compared the values of the existing map to the entry set. Using two separate for loops, we created a condition where if you could not find a node in the original that was also in the entry set, we return the Boolean variables as false.

BST: The implementation of the BST consisted of the creation of a node class where we initialized the creation of left child, right child and a parent. Then we initialized the get immediate node methods (ex. for getparent we just return parent) and set methods (ex. for setparent we just have parent equal the node we want). From here, we constructed an empty node that has no value, key and points to nothing and set the size to 0. Then the we made an actual get method where we set a node to be the node that is the node of the key we entered. If the key of this node and the key we entered is the same then we return the value of the node. This covers the case where the node of the key we entered might not exist. Otherwise, we return null because the node does not exist. For the put method, if the node with the key value pairs we wanted to put at has an existing key, then we just return the value of that node and replace it. Otherwise, we created a helper method that sets the children of a node we want to external. Therefore, if the node we wanted to put at doesn't exist then we are putting at an external node. In which case we set the children to null

and increment size by one. For the remove method, (I looked in the book) we used a helper method called treeMax which is basically a pseudo-inorder traversal. Then in the actual remove method, we say that if the node we want to remove is external, return null because this value does not exist. Otherwise the node must be internal and if this is the case, we run an inorder traversal of the node and store the value of the node we want to remove in a temp. We give the node we want to remove the value of the next inorder node and set that inorder node to null. We then return the temp and our remove is complete. Finally for the entrySet, we begin by creating the helper method "adds". adds recurses until the nodes it encounters are all external meaning all the values have been filled out. Then in entrySet, we just run adds on a new arraylist "buffer" we want filled.

Theoretical Analysis:

Unsorted Table Map: The put function for Unsorted Table Map runs in $O(n^2)$ time, because we could potentially run into a collision of size n n times which would result in this time complexity. Here the average and worst case would be the same so I simply created a timing for the "average" case.

Sorted Table Map: The put function for Sorted Table Map runs in both best and worst case: $O(nlogn)$ time, because in the best case, we are putting the value we want in an incrementing fashion. In a tree with k elements, it would take (in both best and worst case) a time of $logk$ time to find the node and its value. Because we are repeating this process n times, the complexity becomes $O(nlogn)$.

Chain Hash Map: The put function for Sorted Table Map in the best case runs in $O(n)$ time, the function takes a total of $O(1)$ time to perform because we are incrementing the values we are putting at each time so there should be no collision. However, because we are performing the operation n times, we have a time complexity of $O(n)$.

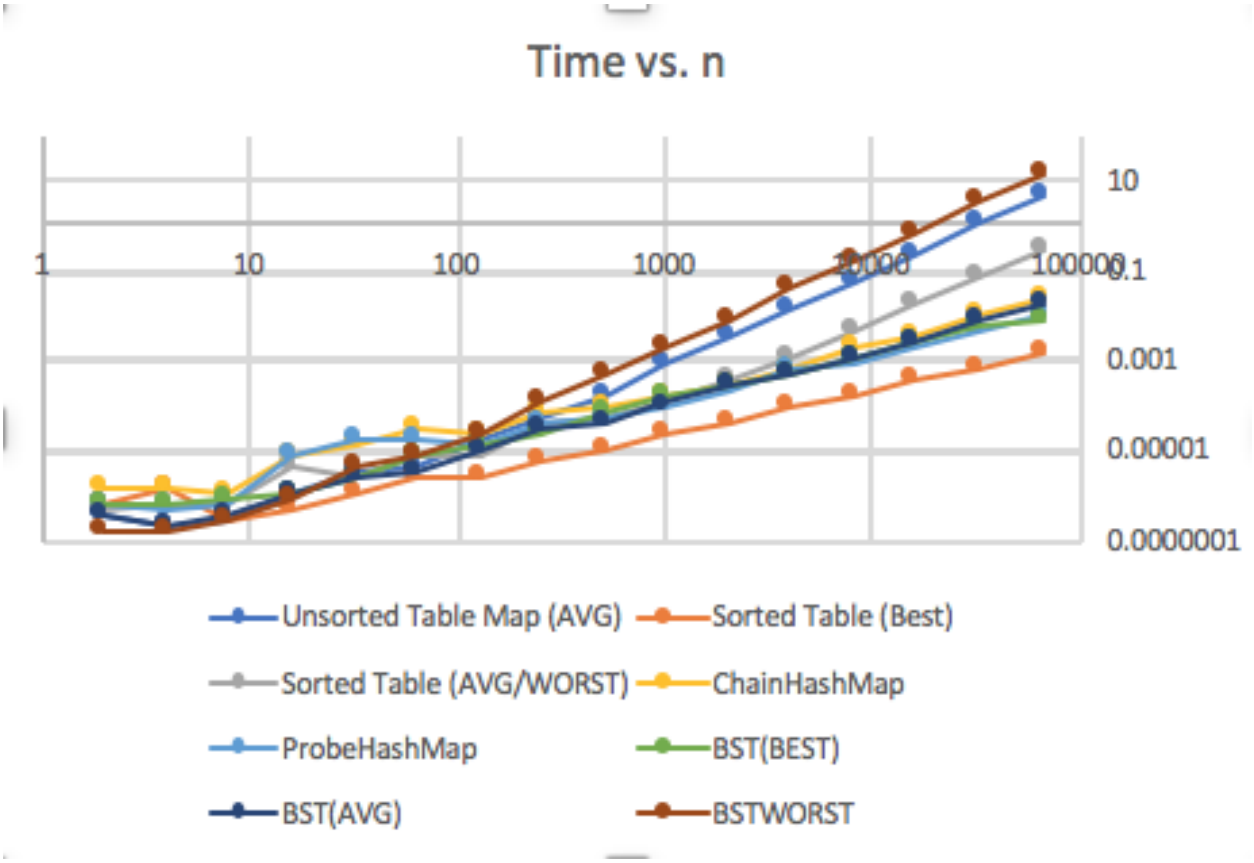
Probe Hash Map: Similarly, the put function for Sorted Table Map in the best case runs in $O(n)$ time, because the function will encounter no collisions in the best case so we will not have to worry about iterating the value we want to store until we find an empty key. This has a complexity of $O(1)$, but because we perform this operation n times we have a time complexity of $O(n)$.

BST: The put function for the Binary Search Tree in best case and average case runs in $O(nlogn)$, because we are finding the node which in best case will be in a balanced tree and in average case an unbalanced but not a tree with n nodes going down one path. Therefore the get takes $O(logn)$ and we perform this operation n times to get a time complexity of $O(nlogn)$. For the worst case, we have all the nodes going down one path, so it will take n time to get the key of the node we want giving us a complexity of $O(n^2)$.

Experimental Setup: My setup is a 2015 MacBook Pro with a 2.9 GHz intel Core i5 processor. It has a memory of 8 GB 1867 MHz DDR3. It runs on OS X El Capitan. We used Java 9 for the creation of this assignment.

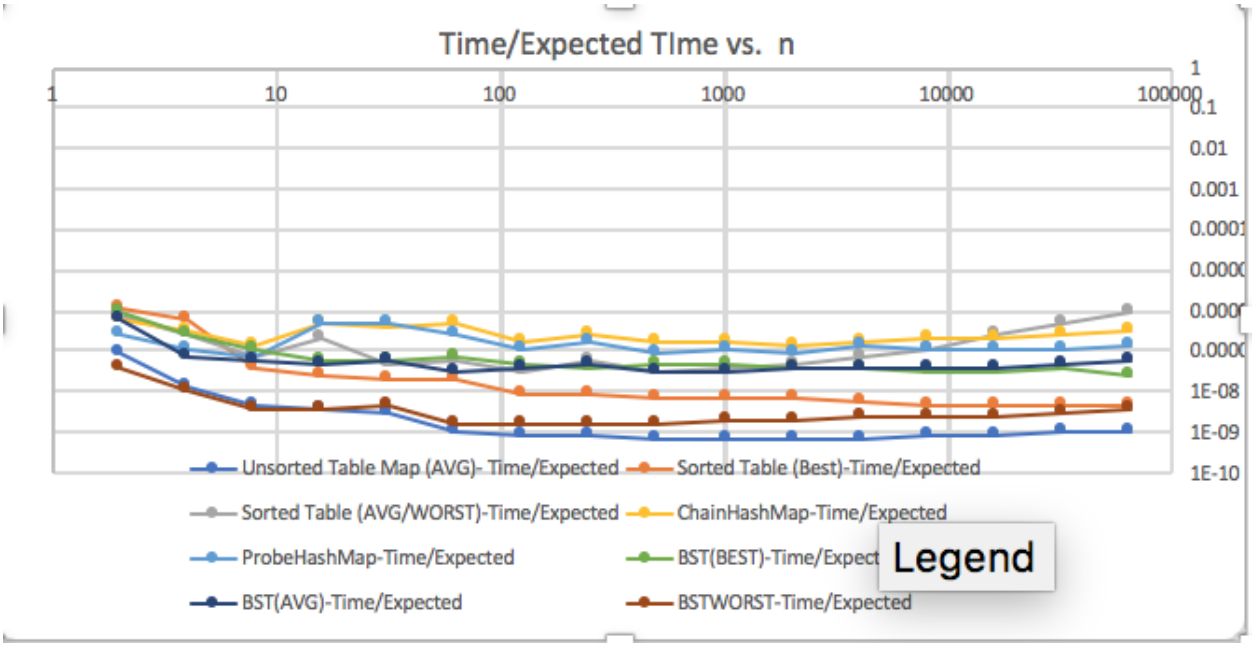
Results and Discussion:

Figure 1:



As we can see in the figure here, in the time vs. n graph, we can see that the Worst Case of the BST is in $O(n^2)$ time due to its also parabolic structure, which the BST Best and Average Cases run in $O(nlogn)$ time which we can see by the small dip into a near linear shape. Similarly, the unsorted table map also has a parabolic structure curving upwards showing that its time complexity is $O(n^2)$. The Sorted Table Map also has a small dip followed by a stabilizing curve showing that it has a time complexity of $O(nlogn)$. Finally, we can observe by the linear shape of the Chain Hash Map and the Probe Hash Map, that they do indeed, have an overall complexity of $O(n)$. The pros of the unsorted hash map is that it will have a good run time given that the amount of times that it is run is relatively few, but in the long run it won't have a very good time complexity. The Chain Hash Map and Probe Hash Map will have a linear trend always giving a stable time complexity. The Sorted Table Map shows that it is probably one of the best data structures in terms of time complexity in this case, because it will be below linear given a small sample, and eventually reach linear which still out performs $O(n^2)$. On the other hand, the BST really depends on whether it encounters a best, average or worst case. If it encounters a best or average case, it outperforms almost all other data structures, but in the worst case it will be outperformed. The $n0$ values of these graphs are visible from the point where the graphs start to stabilize, and for the majority of them this happens where n is equal to 3.87E-06. Meanwhile, the c values of these graphs generally occur at the point that surpasses the $n0$ after the graph stabilizes, so it exists for all possible graphs in this case.

Figure 2:



By observing the figure of the Time/Expected Time vs. n graph we can observe the run time of every data structure. Because Unsorted Table Map and the worst case BST will have $O(n^2)$ values, they will be towards the bottom of the graph. We can see that the time expected for probe hash map is rather high because of the number of potential collisions it will encounter. Other than that, we can see that the time complexities from the Time vs. n graph are holding up to the time/expected time graph.

Conclusion:

Overall, we learned that the complexities of each Data Structure will differ and that each will perform at a different level (time wise). We also learned that overall, some data structures are simply more efficient than others (sorted table map vs. unsorted table map).