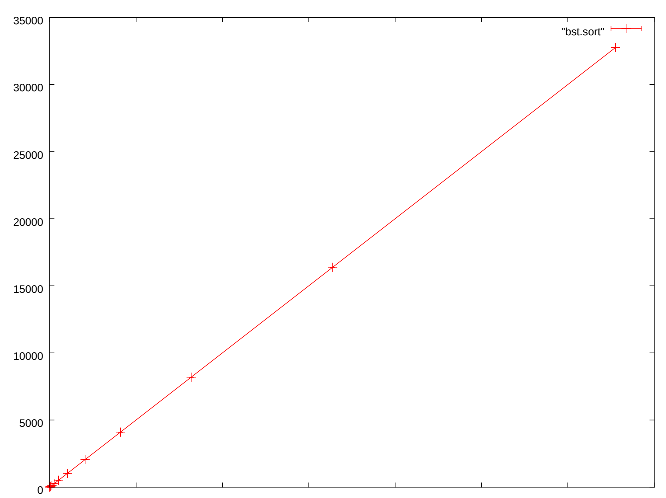
Alexander Niema Moshiri

A09850737 a1moshir@ucsd.edu

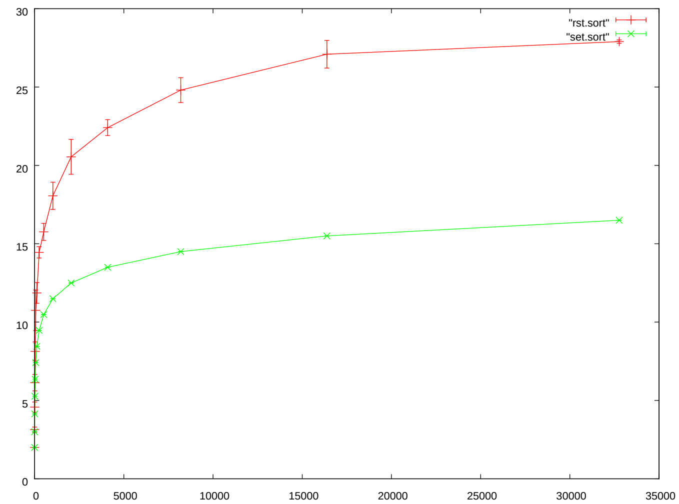
BenchTree – Big-O Time Complexity Analysis of BST, RST, and Set (Red-Black Tree)

In class, we learned about three data types: Binary Search Tree (BST), Randomized Search Tree, and Red-Black Tree. Using C++, we implemented a *BST* in the last programming assignment and an *RST* in this assignment. Then, using a benchmarking program that we wrote, we found information about the efficiency of the *find* functions of our *RST* and *BST* implementations, as well as the *find* function of a C++ *set* (which is implemented as a Red-Black Tree). Now, using that data, we will analyze the efficiency of the three in practice and see if the results match the theoretical Big-O time complexities we learned in class.

A *BST* has a *find* worst-case time complexity of $O(n)$. This is because, in the worst case (which would be if all elements were added in numerical order, either increasing or decreasing), the *BST* would essentially just be a *Linked List* (would be either a right-child or a left-child chain). In this case, a *find* call would have to, in the worst case, cycle through all *n* elements to succeed/fail. In our benchmarking, this linear relation between *find* and *n* elements can be clearly seen in the resulting image below.

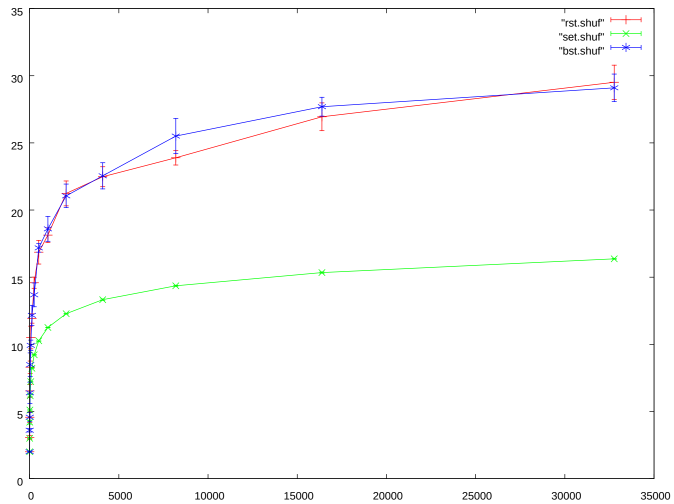


As can be clearly seen, there is a linear relationship between how many elements were added in sorted order and how many comparisons were needed for the *find* function. I benchmarked the *RST* and *set* as well, but was forced to plot them on a separate graph because the $O(n)$ *BST* plot was significantly larger and made them scale to an unnoticeable line. The separate graph for these two is below (red is *RST* and green is *set*).



As can also clearly be seen, there is a logarithmic relationship between how many elements were added in sorted order and how many comparisons were needed for the *find* function for both of these data structures. This makes sense because, although the *RST*'s *insert* function starts identically as the *BST*'s *insert* function, the *RST*'s added *heap* property requirements cause it to self-fix in order to meet these requirements after the *BST*-style add. Since the priorities of an *RST* are randomly generated, this tends to result in a pretty balanced tree (not necessarily perfect, and it could by chance end up being very imbalanced, but it's USUALLY pretty good). Also, since the *set* is implemented as a *Red-Black Tree*, its insert method guarantees a relatively perfectly balanced tree structure, resulting in a more efficient (yet still logarithmic) *find* function. Therefore, both have time complexities of $O(\log_2 n)$.

I then benchmarked how well the three structures did when the elements inserted into them were randomly sorted, and the results can be seen below (red is *RST*, green is *set*, and blue is *BST*).



As can be seen, all three structures have time complexities of $O(\log_2 n)$ when the data is in shuffled order (which is the average case). Again, the *RST* and *BST* are pretty balanced in the average case, while the *set* (which as we know is implemented as a *Red-Black Tree*) is nearly perfectly balanced, resulting in lower (yet still logarithmic) average case results.

In conclusion, as can be seen from the data collected, the actual time complexities of the *find* function of each of our data structures matches the theoretical ones learned in class. In the worst case, the *RST* and *set* (*Red-Black Tree*) have time complexities of $O(\log_2 n)$, while the *BST* has a time complexity of $O(n)$. In the average case, all three structures have time complexities of $O(\log_2 n)$.