COMP 2402 – ASSIGNMENT 9

NEM ZUTKOVIC

101085982

All execution times are displayed in **seconds, with 2 seconds as the y-axis limit. The reason for this limit is because no data structure exceeded this run time with any method. Tests were performed by the Tester.java file.

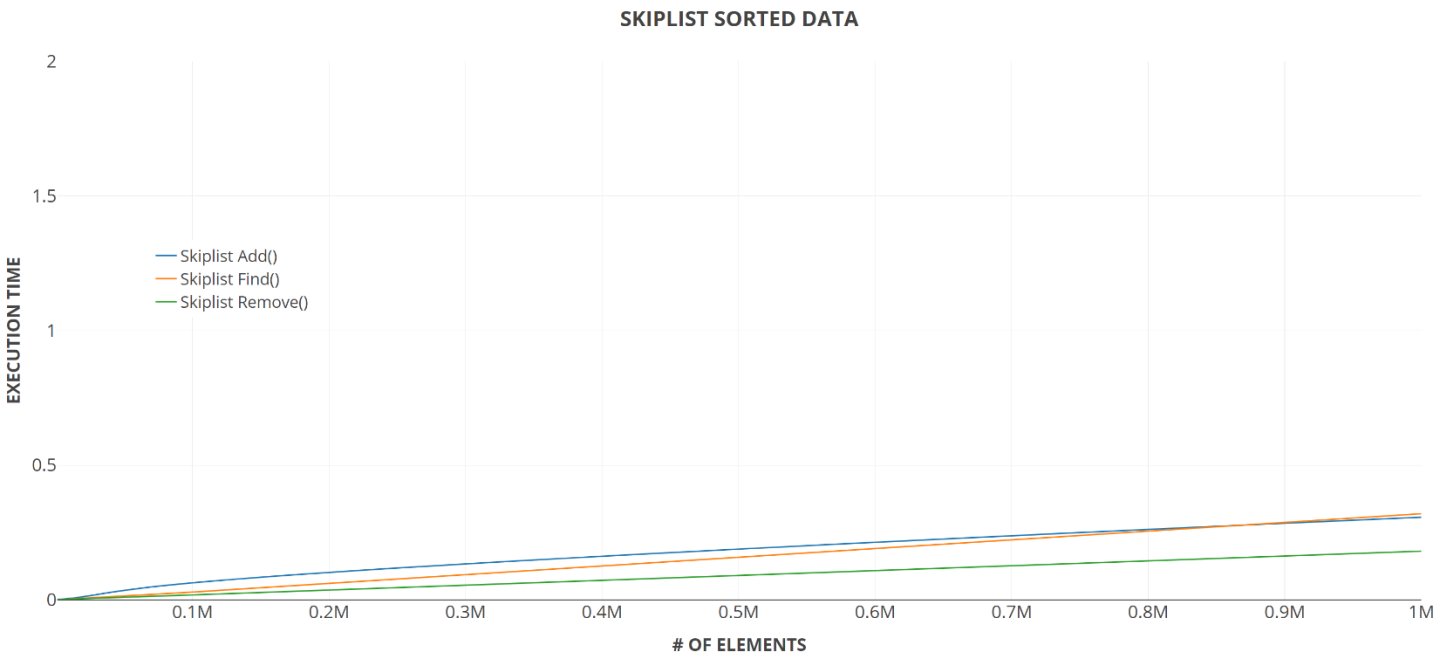

Figure 1. *Skiplist Sorted Data*

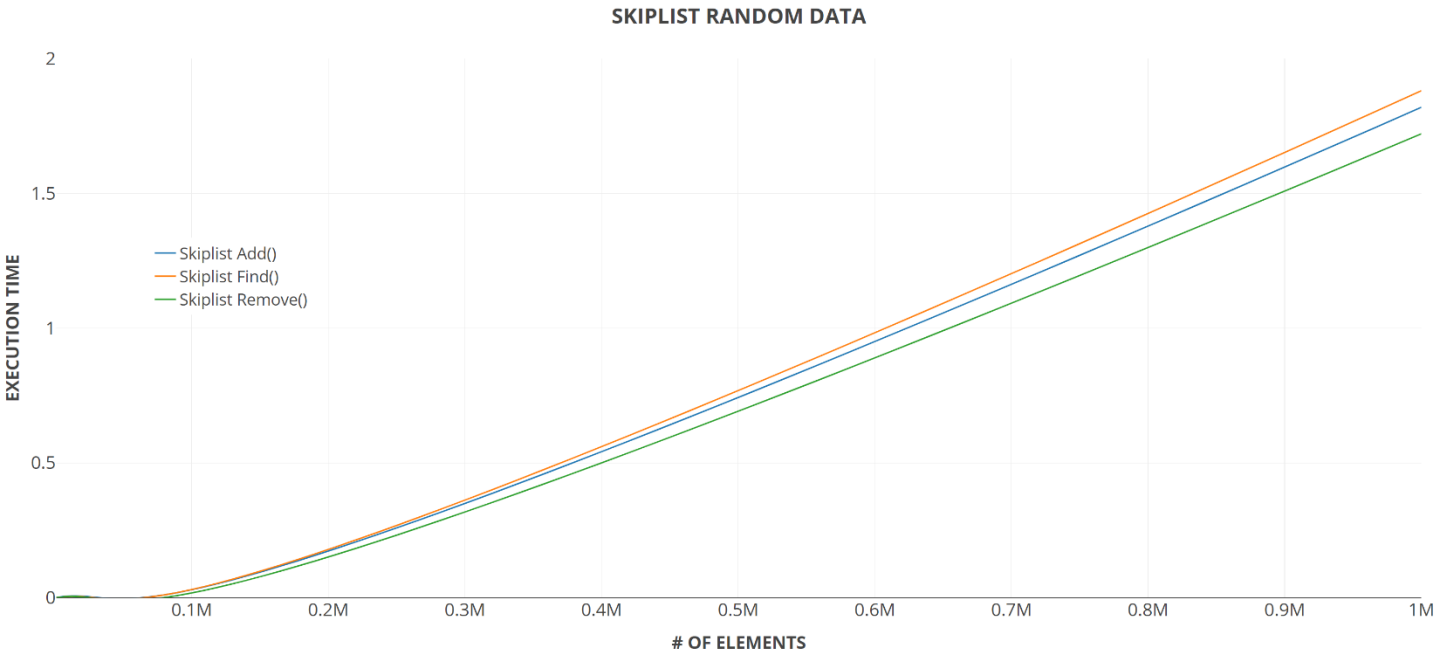

Figure 2. *Skiplist Random Data*

Figure 1 which supports a sorted data set for the Skiplist, falls in line with the **expected** *O(logn)* run time. In contrast, *Figure 2*, which represents a random data set, does not seem to support the *O(logn)* runtime. A possible hypothesis to explain this, is that like all the other data structures here, the Skiplist implements the Sorted Set interface. Starting with sorted data will always be more efficient than starting with completely random data. When every element is randomized, we begin to see the Skiplist slowly approach its worst case run time of *O(n)*. I knew the random data set would take longer to execute than the sorted data set, but am surprised to see the random data set **not** demonstrate an *O(logn)* runtime.

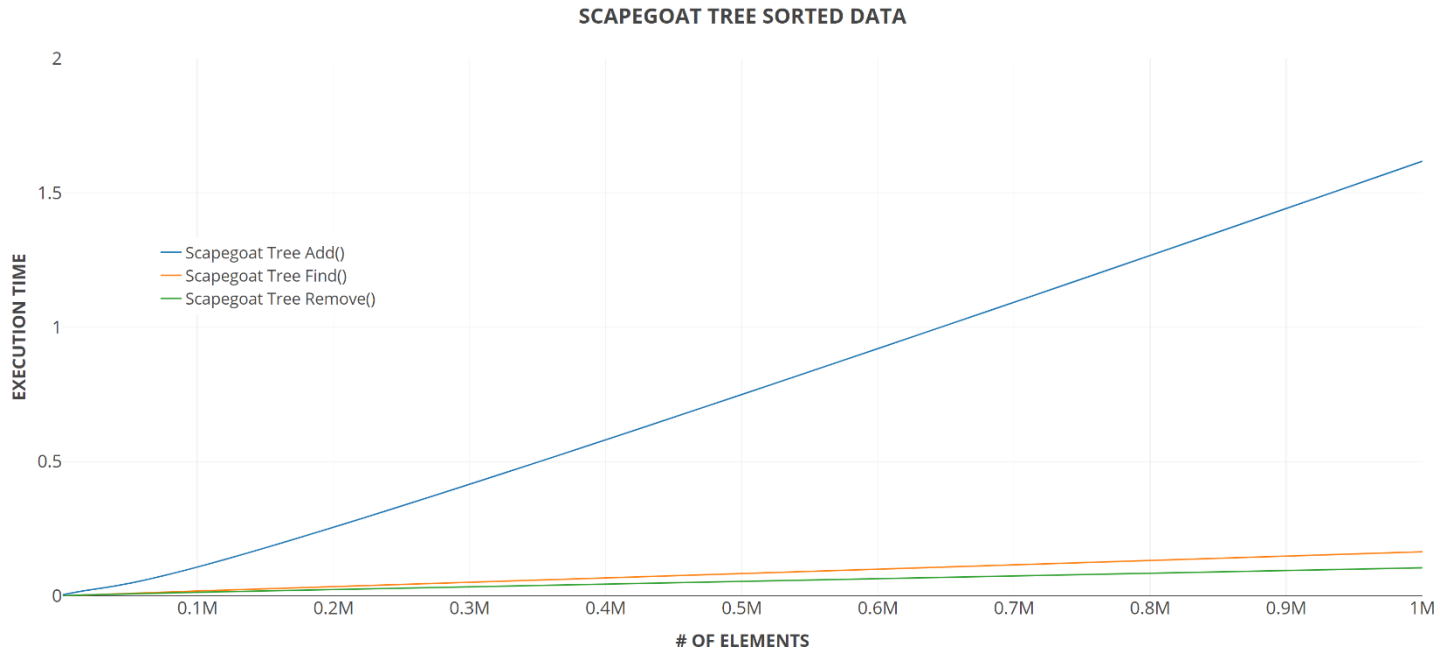

Figure 3. *Scapegoat Tree Sorted Data Plot*

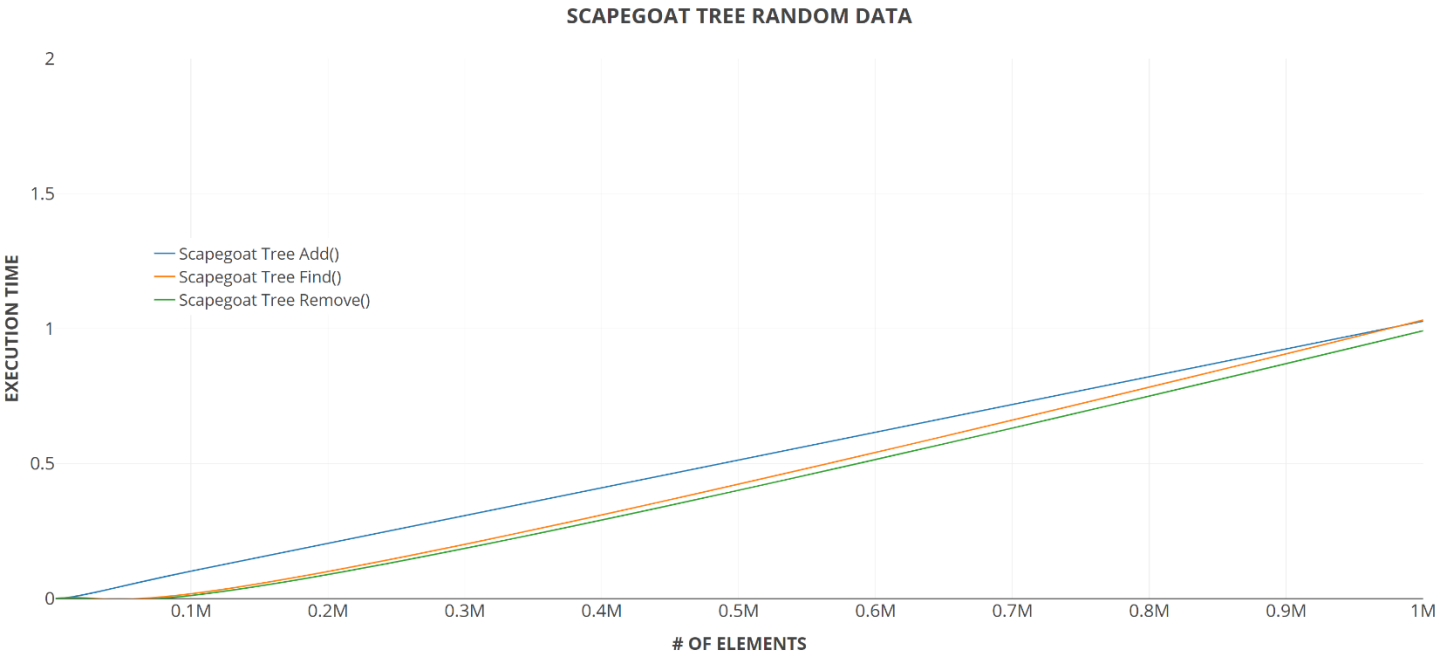

Figure 4. *Scapegoat Tree Random Data Plot*

Excluding the notion that starting with sorted data will always be more efficient than starting with completely random data, the Scapegoat Tree performs as expected. The **expected** runtime of a Scapegoat Tree is *O(logn)*. This is something we see very obviously with the sorted data set in *Figure 3*, with the exception of the add() method. The reason the add() method deviates so much from the other methods is because the scapegoat tree begins to expensively rebuild the subtree at the scapegoat it chooses. The more elements in the data structure the more runtime the *add()* method may consume. For this reason, the Scapegoat tree has an *O(n)* worst case time complexity, and that is becoming more and more prevalent as the number of calls to *add()* keep increasing. I am not surprised by the *add()* method in the sorted data set, but once again, I am surprised by the random data set. I was expecting the Scapegoat Tree to maintain a better logarithmic pattern.

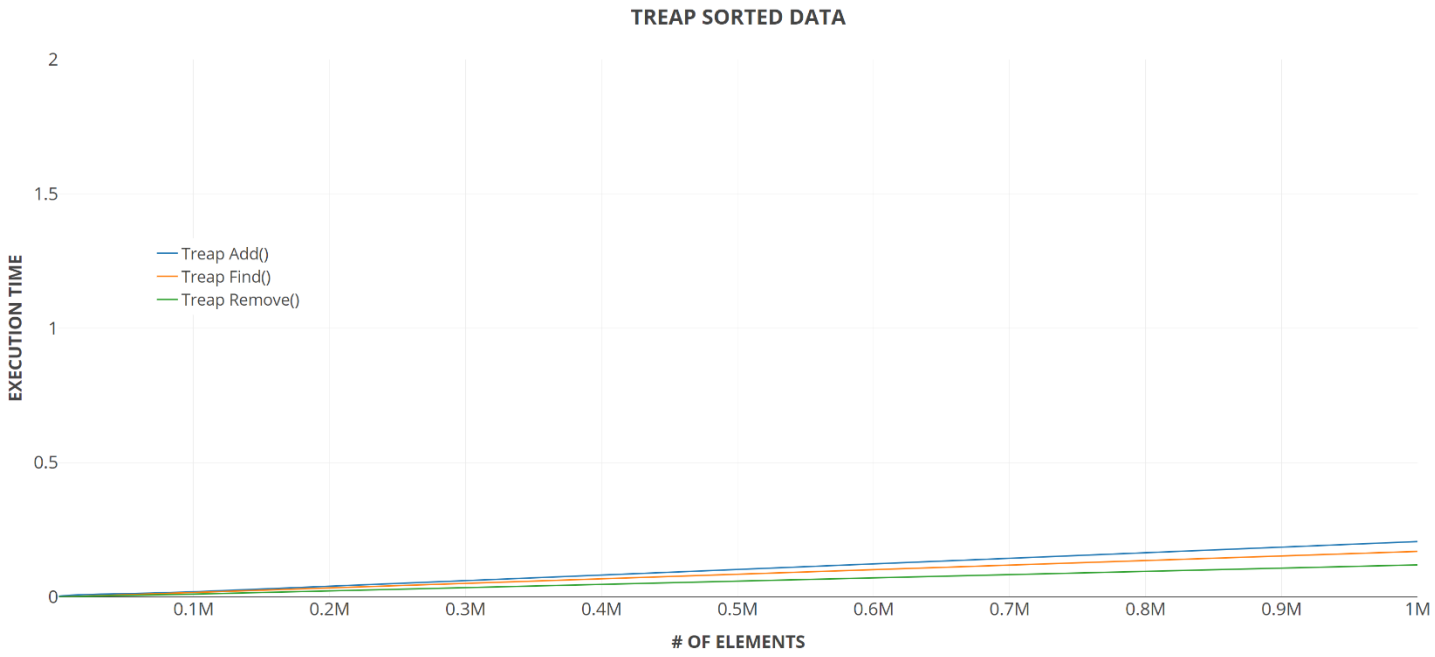

Figure 5. *Treap Sorted Data Plot*

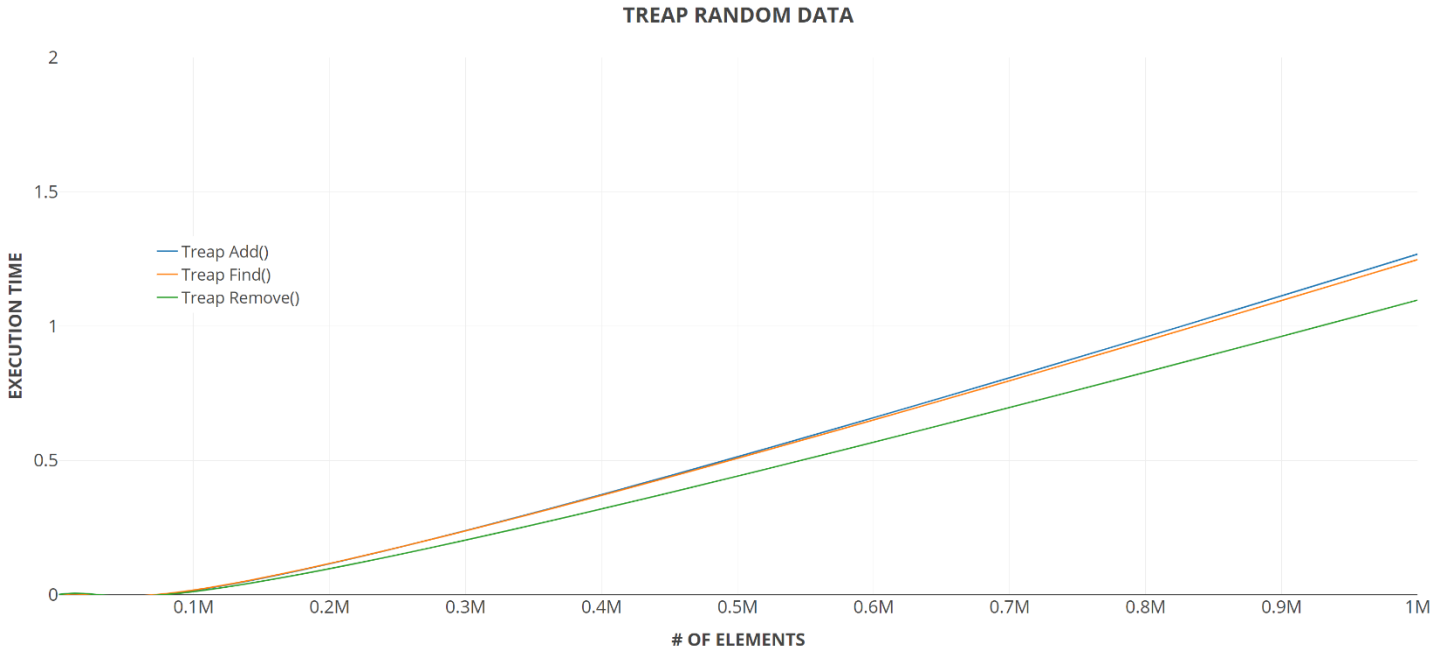

Figure 6. *Treap Random Data Plot*

The Treap has **expected** runtimes of *O(logn)* when the data is sorted (*Figure 5*), but when the data is completely random (*Figure 6*), the Treap time complexity deviates exactly like the Skiplist. The Skiplist and Treap both exhibit similar plots whereas the Treap performs slightly better than the Skiplist. You can notice at 1 million elements the Treap will run in about 1.25 seconds, but the Skiplist will run over 1.75 seconds. This slight difference in performance can be easily seen between *Figure 2* and *Figure 6*. *Figure 1* and *Figure 5* also exhibit this pattern but it is more difficult to interpret. If this conclusion is not entirely evident visually, the Treap outperforming the Skiplist can be seen in the attached sample data set used to build these plots. This comparison of runtimes between data structures supports the notion of the Treap's expected search path length of *1.386logn + O(1)*, compared to the Skiplist's search path length of *1.844logn + O(1)*.

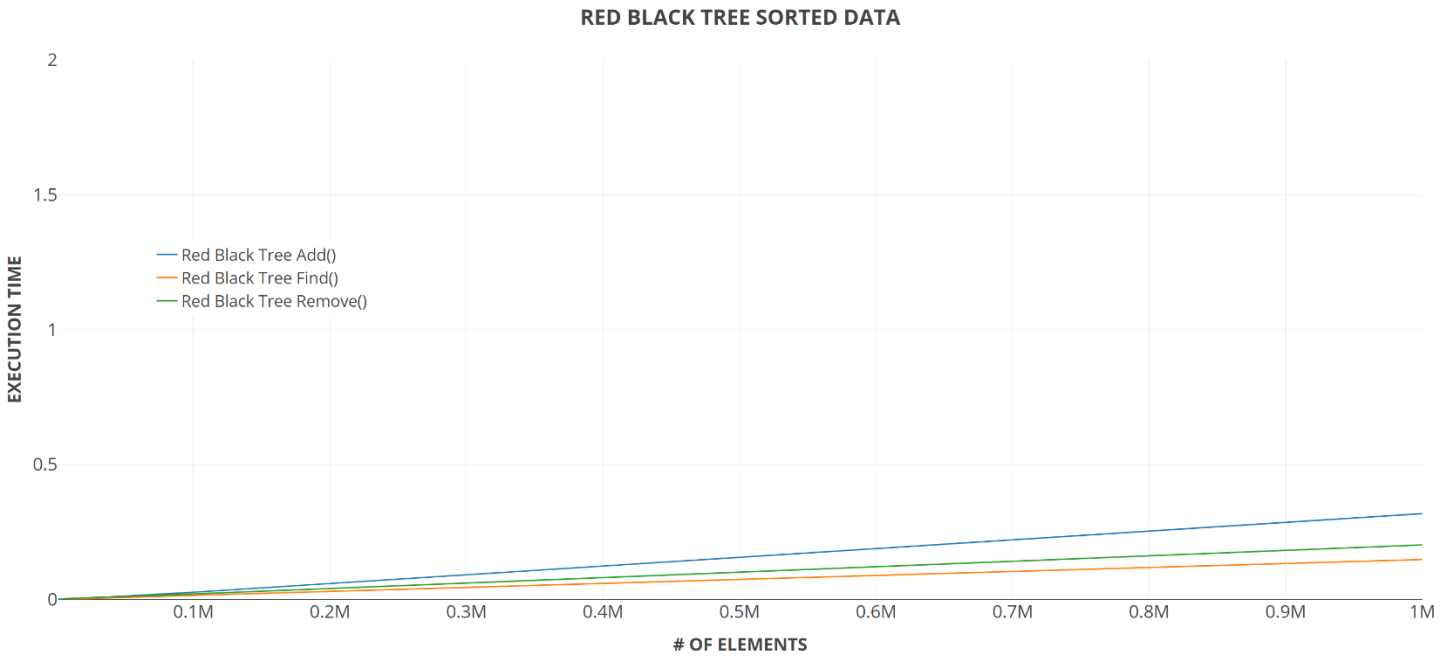

Figure 7. *Red Black Tree Sorted Data*

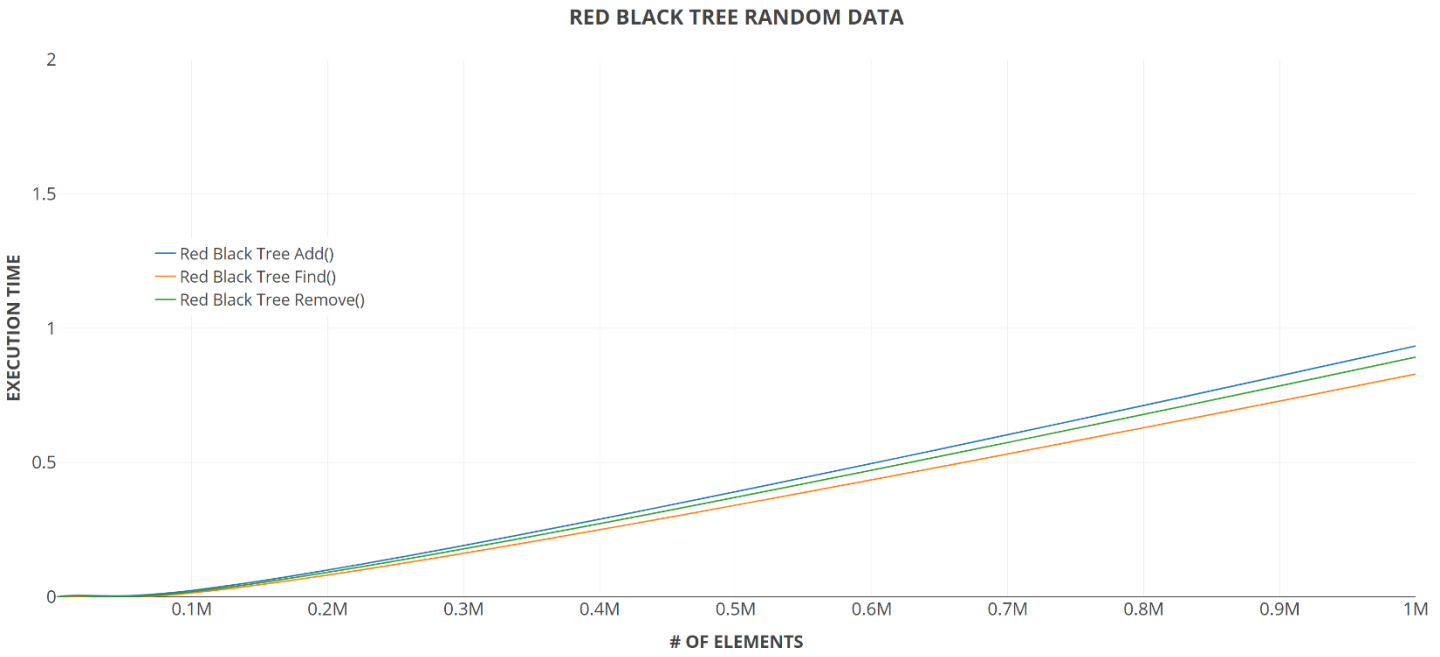

Figure 8. *Red Black Tree Random Data*

Lastly, the Red Black Tree performs as expected, with an **expected** runtime of *O(logn)* when dealing with sorted data. Runtimes with sorted data is faster than runtimes with random data like all of the other data structures. Something interesting about the Red Black Tree is that it has this element of consistency between all its' methods. In terms of performance, the Red Black Tree single-handily beats out the other data structures in many, if not all, of the methods.

To conclude the analysis of all these data structures, the major aspect of all these plots that surprise me, and maybe it shouldn't, is that the runtime of each method when dealing with random data does not appear to maintain the **expected** *O(logn)* time complexity. Maybe this is beginning to show the real-world performance of these data structures as we were only really studying their **expected** time complexities, or maybe I just did something terribly wrong when setting up my random data.