Patrick Farley

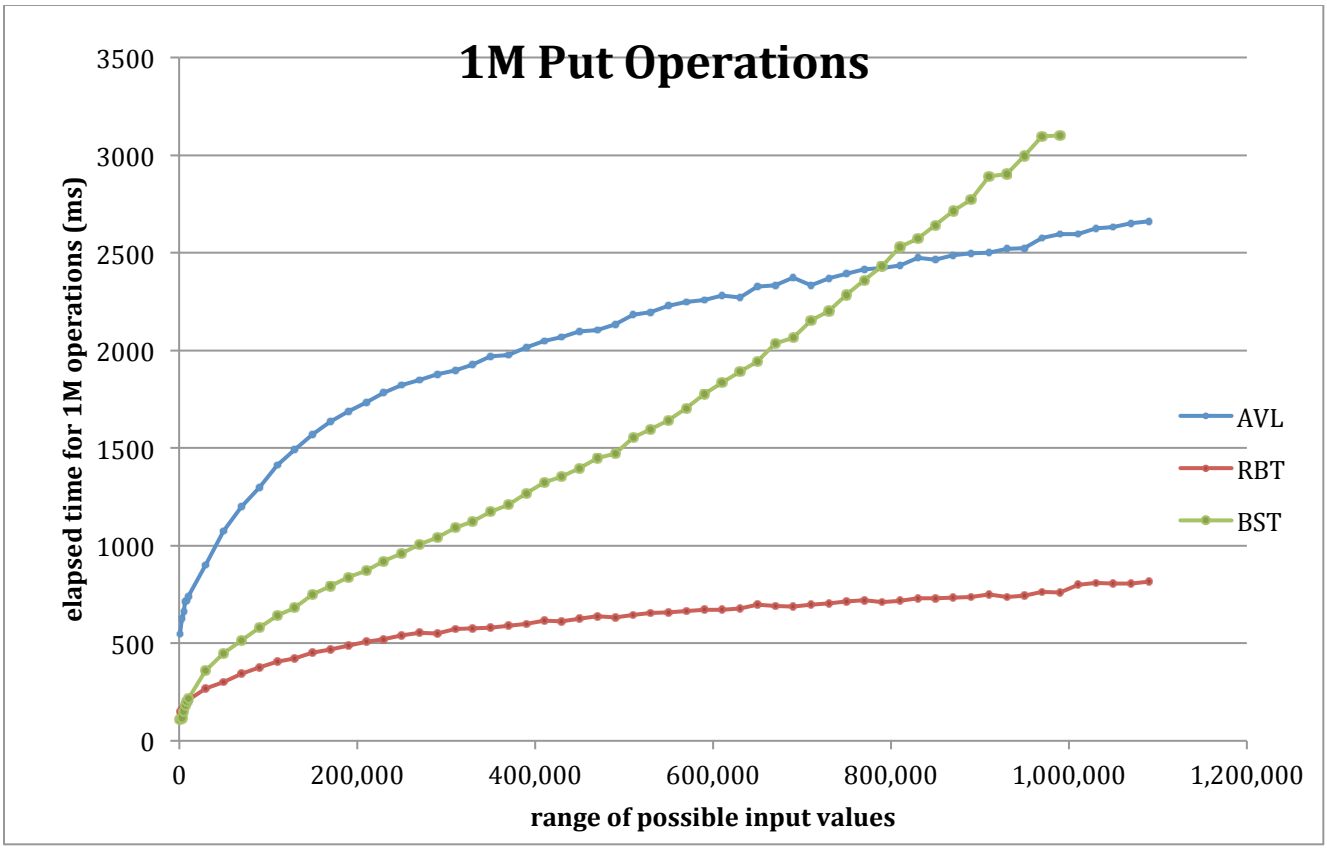
CS 345 Project 3 Report

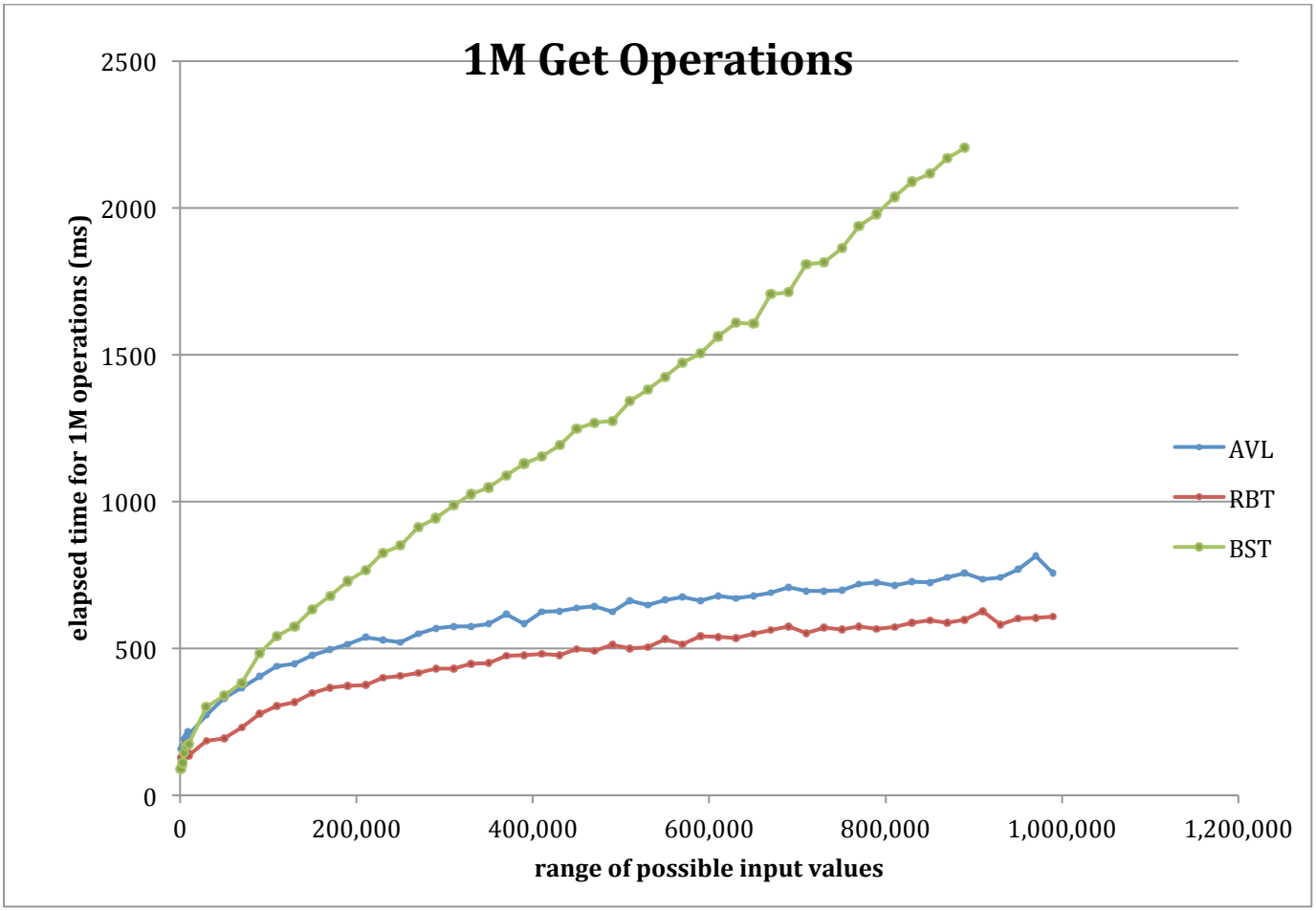
After completing my implementations of AVL Tree and Red Black Tree, I measured the time of a series of put operations and the time of a series of get operations separately for each type of tree, to get a sense of how each tree handled either type of operation. When fiddling around with the input parameters of the tests, I noticed something that I thought peculiar at the time. The relative performances of the AVL Tree, Red Black Tree, and unbalanced implementation of Binary Search Tree depended on the range of values that were inserted and called in the test. Specifically, I was using a random integer function with a parameter for the range of potential values. The total put times as well as get times depended on this range. I soon realized that the range of values that can be inserted into a tree greatly affects the size of the tree, because it determines how many repeated key insertions there will be. In fact, the range of values inserted into a tree can, over time, have a much greater effect on the size of the tree than the raw number of insertions. I decided to focus on this dynamic and do some more complete tests.

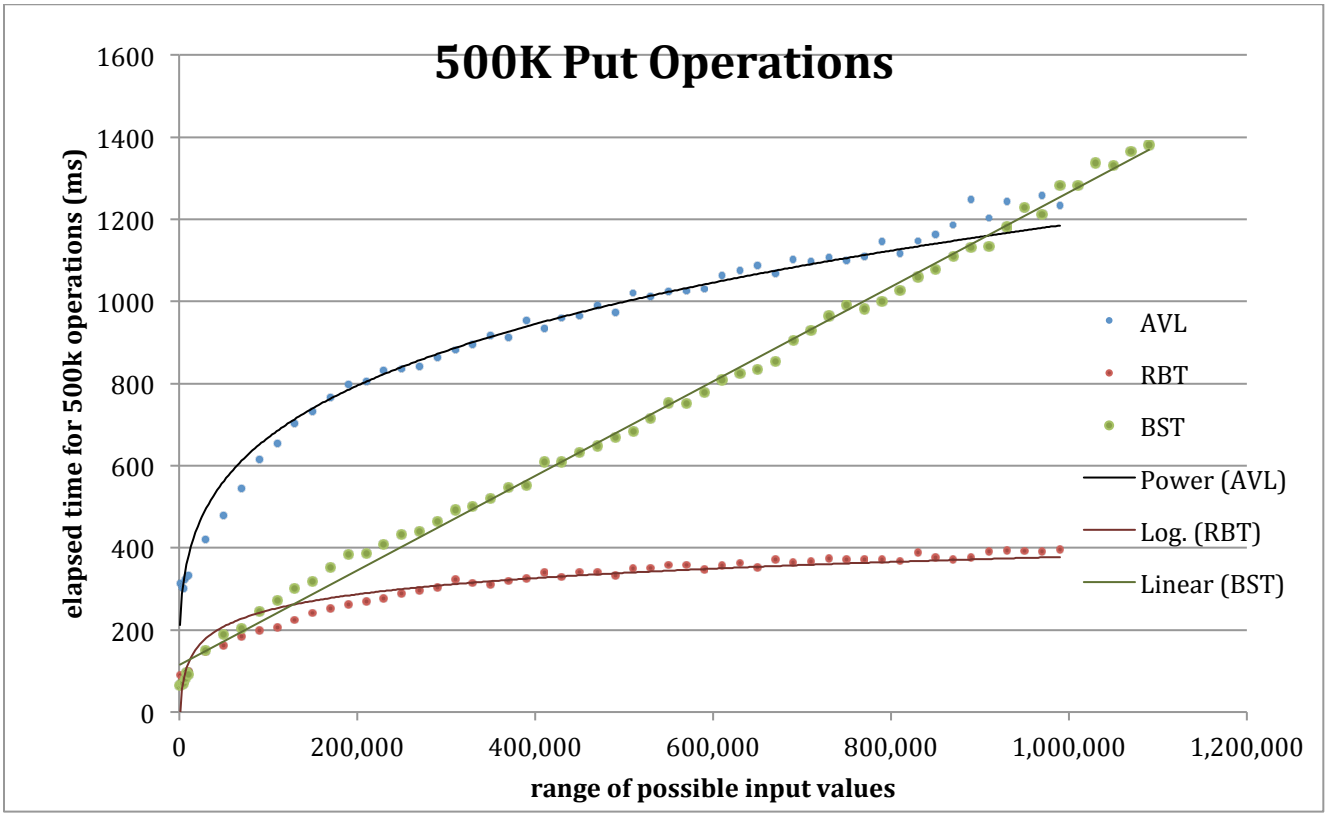
My tests had two independent variables: the number of insertions or retrievals per test, and the range of possible values to be inserted or retrieved. I decided to pick three fixed values for the raw number of insertions and retrievals: 100,000, 500,000, and 1,000,000. For each of those, I tested times over a wide range of input value ranges.

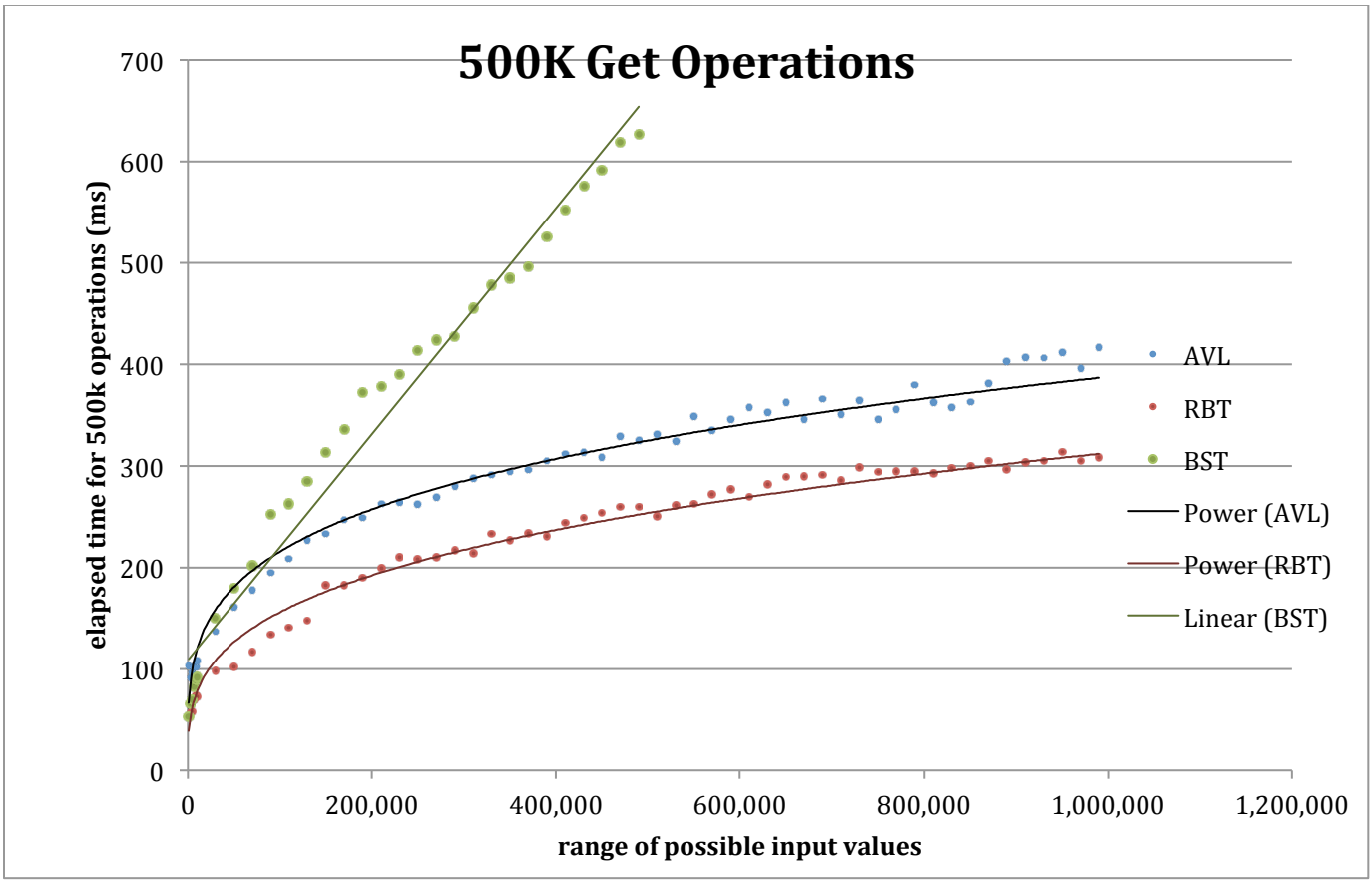
One important feature of my tests was a built-in option to run redundant tests and only return the average. I used this feature as I saw fit to get cleaner data, without taking too much time per test.

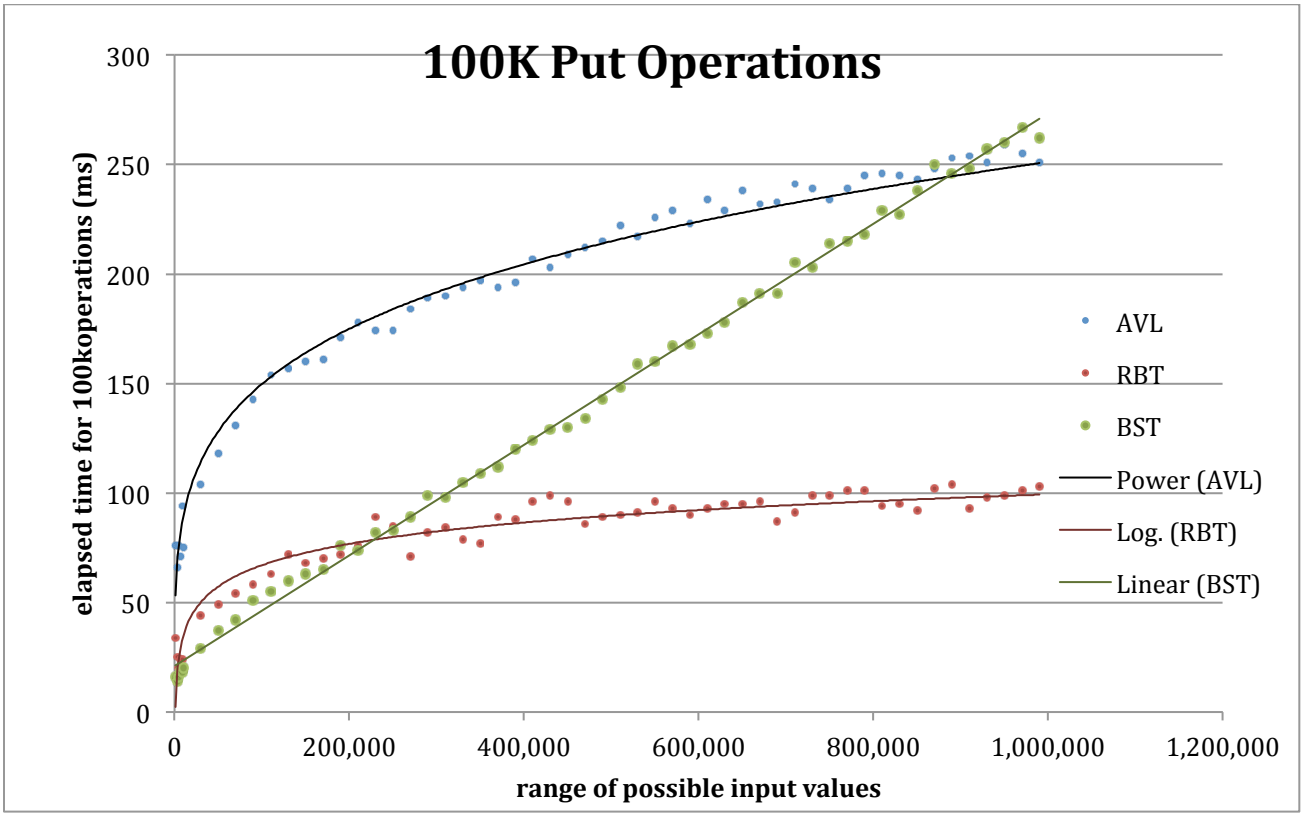
My data yielded plots similar to what I'd expect to see if the x-axis had been total number of items added/retrieved and there were no repeats in the data set. That is, the AVL Tree and Red Black Tree both yielded log plots for the time elapsed vs. range of possible input values, and the unbalanced implementation yielded linear plots. This was true for trees of 100,000, 500,000, and 1,000,000 insertions. See all six plots below:

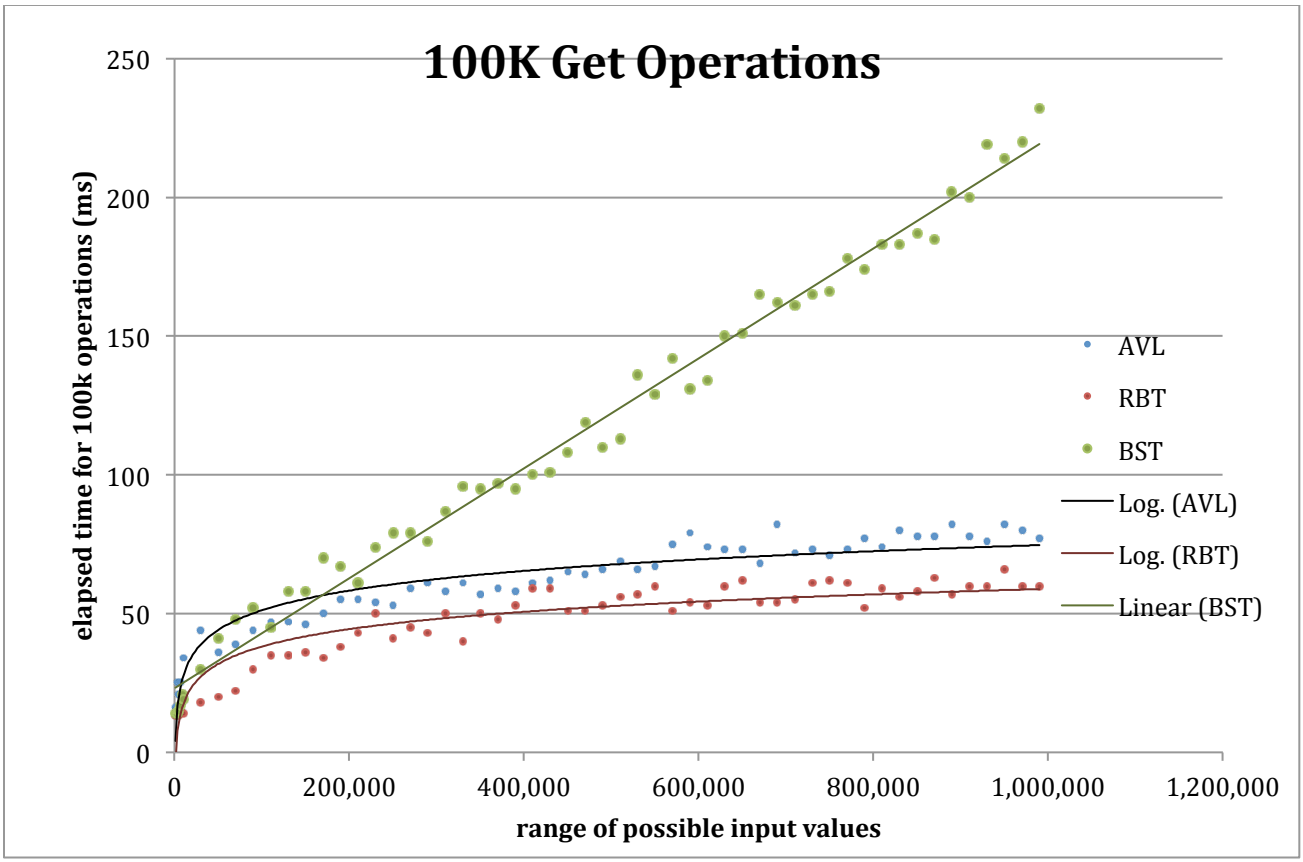

1M Put Operations
elapsed time for 1M operations (ms)
3500
3000
2500
2000
1500
1000
500
0
0
200,000
400,000
600,000
800,000
1,000,000
1,200,000
range of possible input values
AVL
RBT
BST


1M Get Operations
elapsed time for 1M operations (ms)
2500
2000
1500
1000
500
0
0
200,000
400,000
600,000
800,000
1,000,000
1,200,000
range of possible input values
AVL
RBT
BST


500K Put Operations
elapsed time for 500k operations (ms)
range of possible input values
0
200
400
600
800
1000
1200
1400
1600
0
200,000
400,000
600,000
800,000
1,000,000
1,200,000
AVL
RBT
BST
Power (AVL)
Log. (RBT)
Linear (BST)


500K Get Operations
elapsed time for 500k operations (ms)
range of possible input values
0
100
200
300
400
500
600
700
0
200,000
400,000
600,000
800,000
1,000,000
1,200,000
AVL
RBT
BST
Power (AVL)
Power (RBT)
Linear (BST)

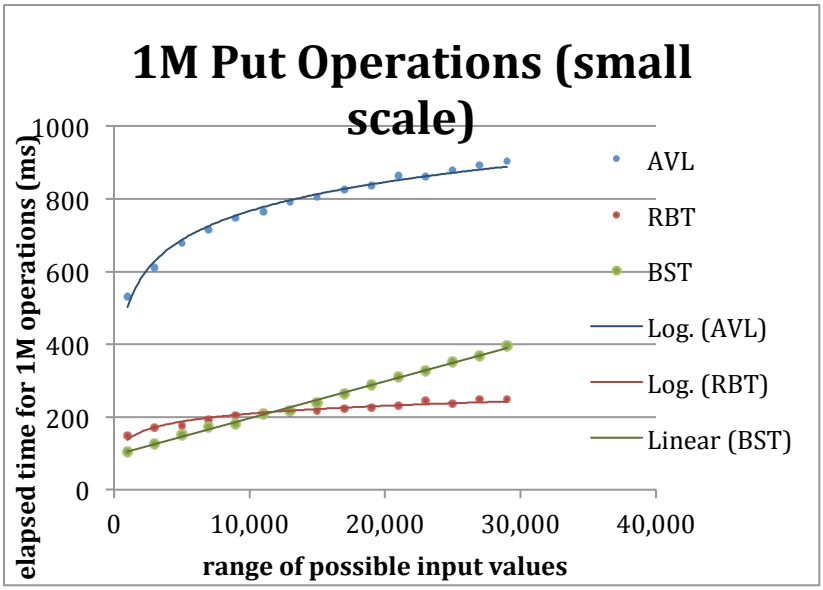

100K Put Operations
elapsed time for 100koperations (ms)
300
250
200
150
100
50
0
0
200,000
400,000
600,000
800,000
1,000,000
1,200,000
range of possible input values
AVL
RBT
BST
Power (AVL)
Log. (RBT)
Linear (BST)

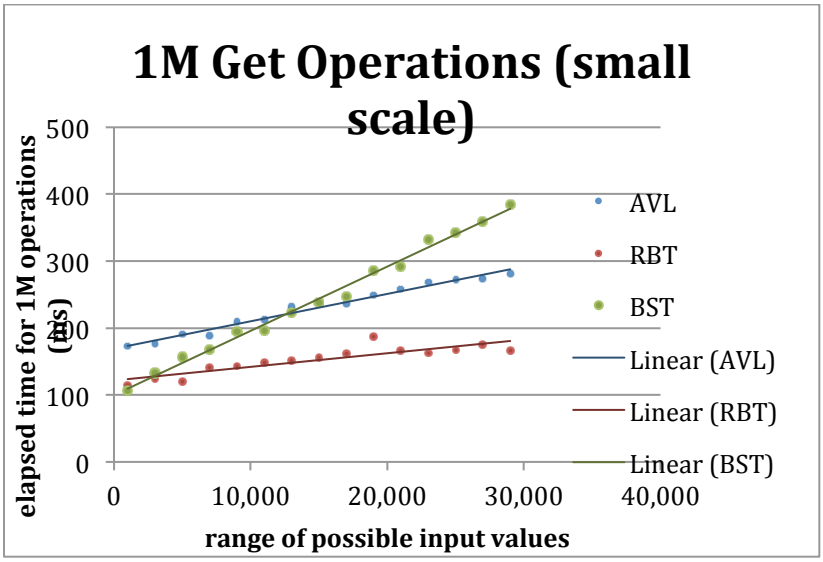

100K Get Operations
elapsed time for 100k operations (ms)
250
200
150
100
50
0
0
200,000
400,000
600,000
800,000
1,000,000
1,200,000
range of possible input values
AVL
RBT
BST
Log. (AVL)
Log. (RBT)
Linear (BST)

Note: when I did tests with a lower number of insertions/retrievals, the randomness of the values inserted played a larger role in the natures of the trees, and even after averaging data points, the operation times were more sporadic for these trees. I used fitting functions to map the general behavior of these plots.

The first thing we may notice about this data is that my implementation of the Red Black Tree was significantly faster than my implementation of the AVL Tree in every test. I was expecting my AVL Tree to be faster at insertions, at least for some of the time, but this was simply not the case.

The significant feature of these graphs is that they show exactly where one type of tree becomes faster than another. The graphs for 1 million entries were scaled too large to see these points exactly, so I also made some plots at a smaller scale. View them below.





The points of intersection on each of these plots show that, for a given number of insertions/retrievals, there is a certain range of values below which the unbalanced implementation of the Binary Search Tree will actually be the fastest at insertions. That is, if there are enough repeated values being inserted, the unbalance implementation can be faster than both AVL and Red Black Trees, even if a great many insertions are made (because the overall size of the tree will remain small enough).

More interesting still, the unbalanced implementation is also faster at *retrieval* than the AVL Tree, provided the range of possible values is small enough. Some of the plots suggest it is also faster at retrieval than the Red Black Tree at certain points, but I'd deem these too close to call from the given data.

I was hoping to find a constant ratio of the number of insertions done to the range of possible values that would reliably mark where one type of tree became faster than another. Unfortunately, the data I took did not suggest any such constant ratio. I suspect if I were to take further data at different numbers of insertions/retrievals, I would start to see a pattern there.

My tests demonstrate the significance of the range of possible values on a tree's performance. They show that when designing a structure to store simple data pairs, one needs to know more than just the expected number of insertions and retrievals. If the incoming data is expected to have a high number of repeats, the programmer may be able to save both time and effort by implementing a simple unbalanced tree instead of designing a self-balancing one.