JC Maes

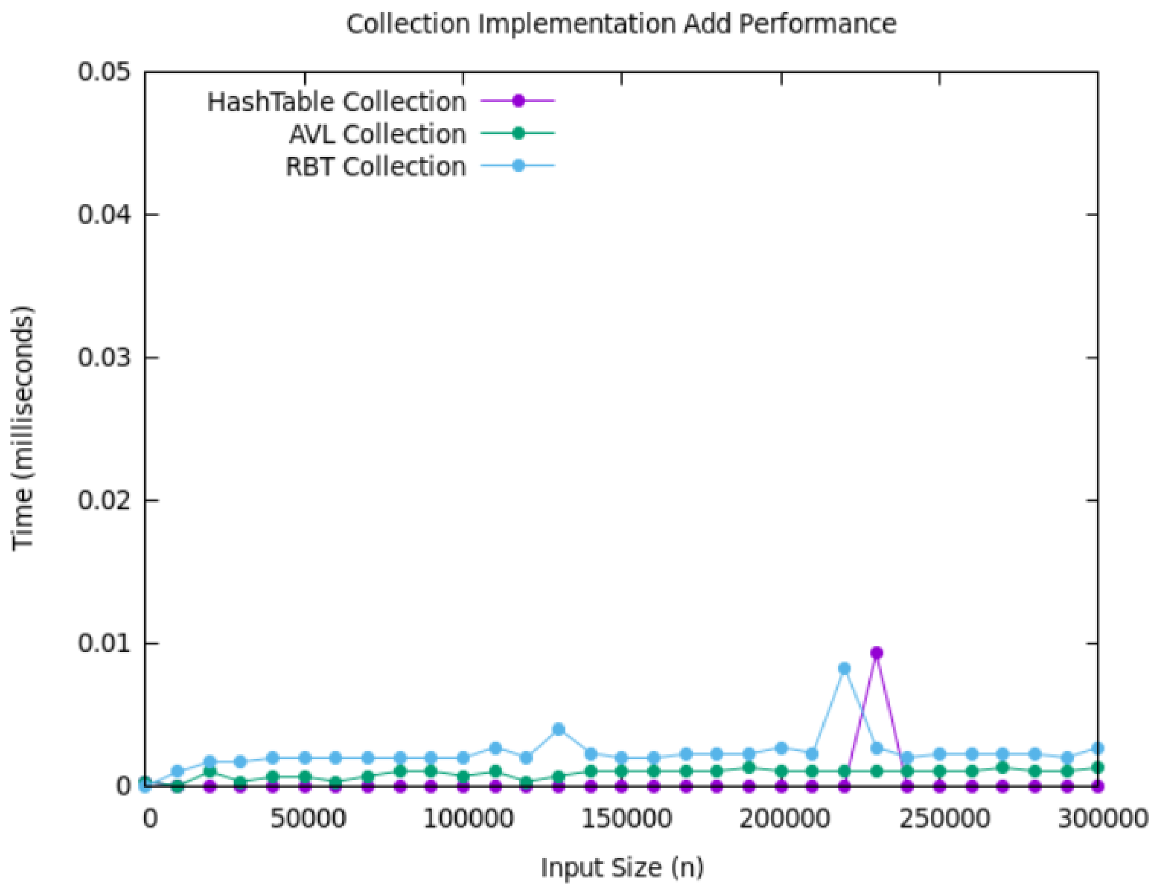
CPSC223

Dr. Bowers

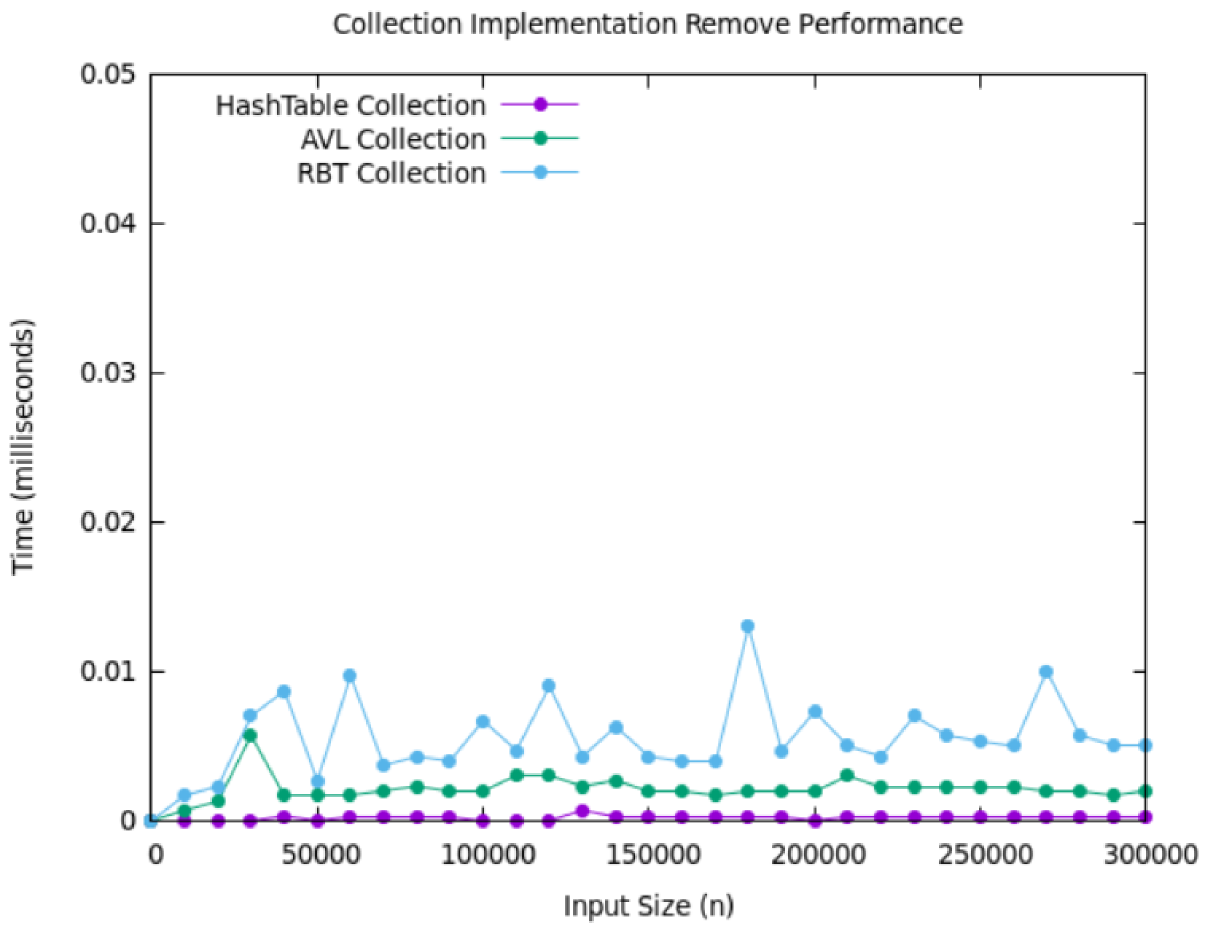
12/15/2020

HW9 Write-Up

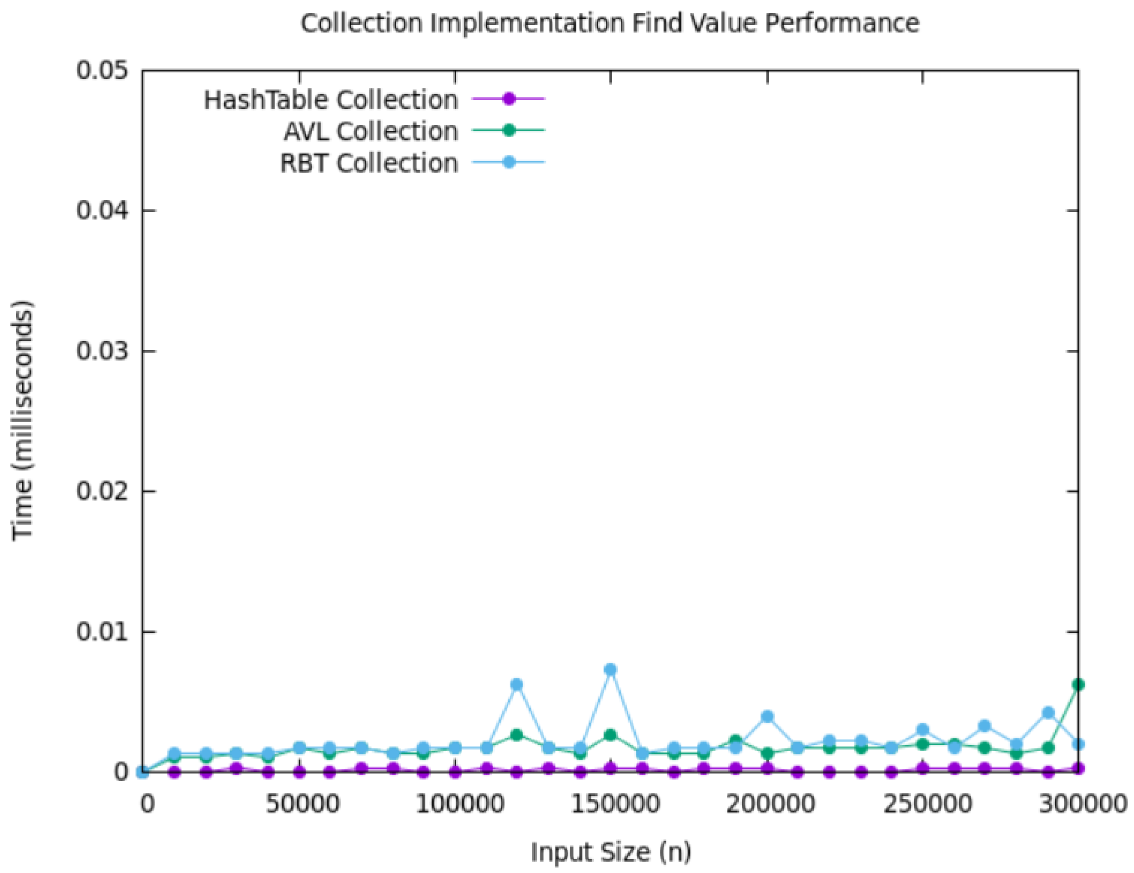
Performance Graphs:



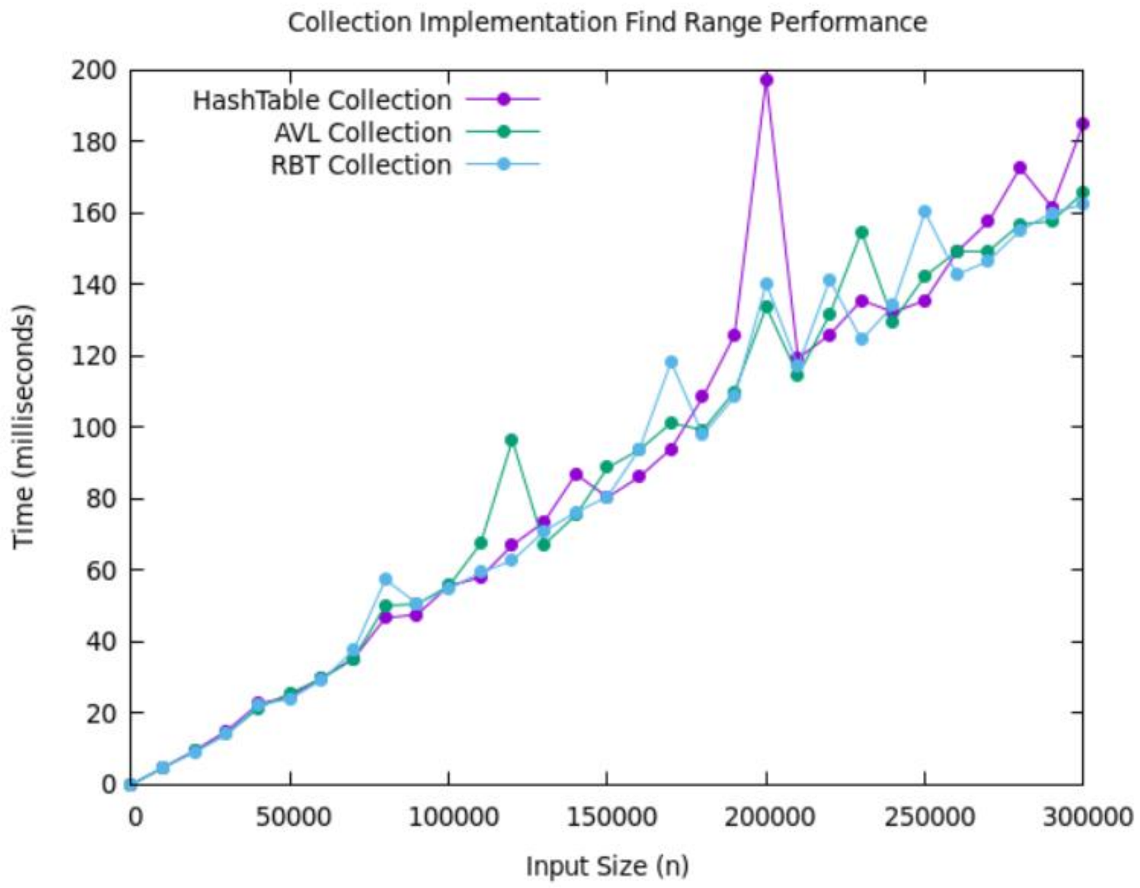
Everything in this graph performs as expected, since hash table's add is O(1) and the other two are O(log n). Also, testing that the tree was valid is more involved with the RBT as opposed to the AVL tree. This is why, although both have a worst case of O(log n), the AVL tree performs slightly better.



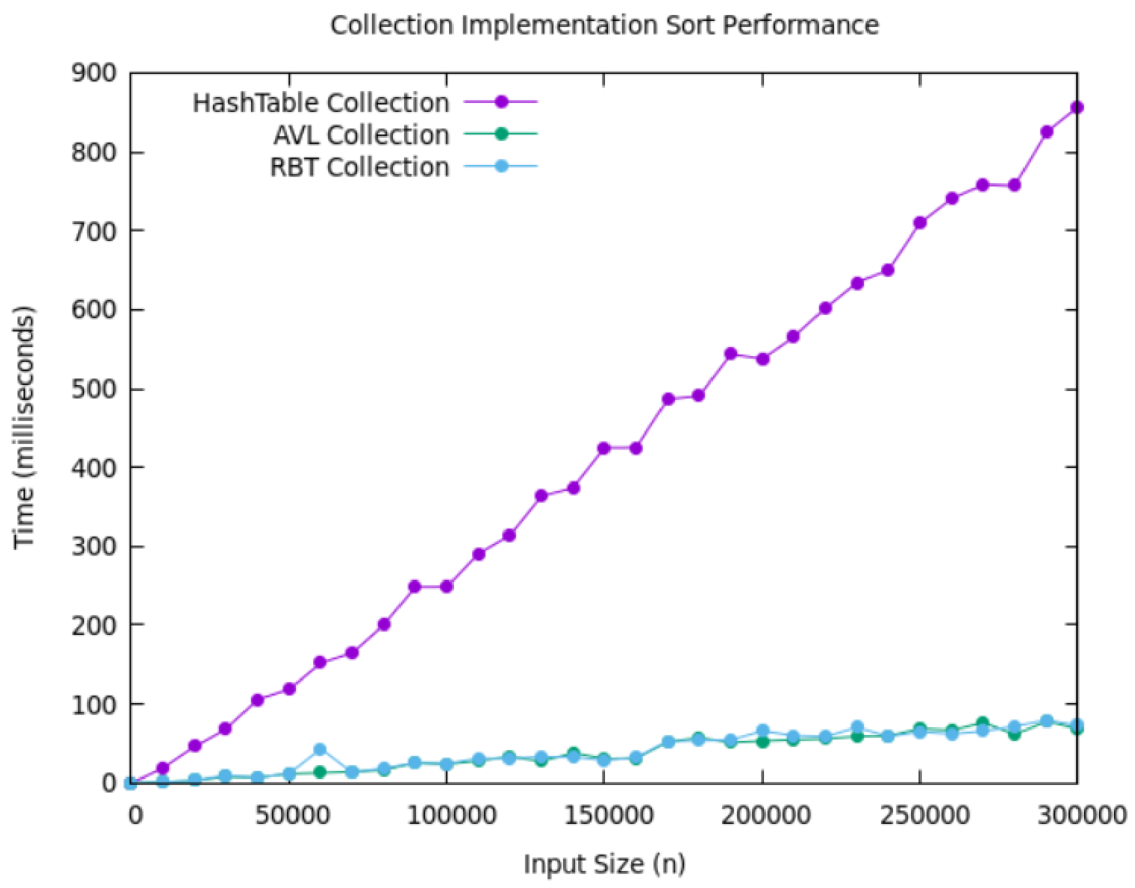
Here, the results are little unexpected, only due to the spikes in the RBT remove. If I had to guess, I'd say this is due to the complexity of the remove_rebalance function in RBT (or just incorrect implementation). Although the spikes are weird, the overall data trends are as expected. AVL and RBT remove at a worst case complexity of O(n). Also, hash tables remove might be worst case O(n), but in practice it performs much better. This is because the worst case accounts for potentially traversing nodes, but since we introduced a load factor there is very little traversing needed.



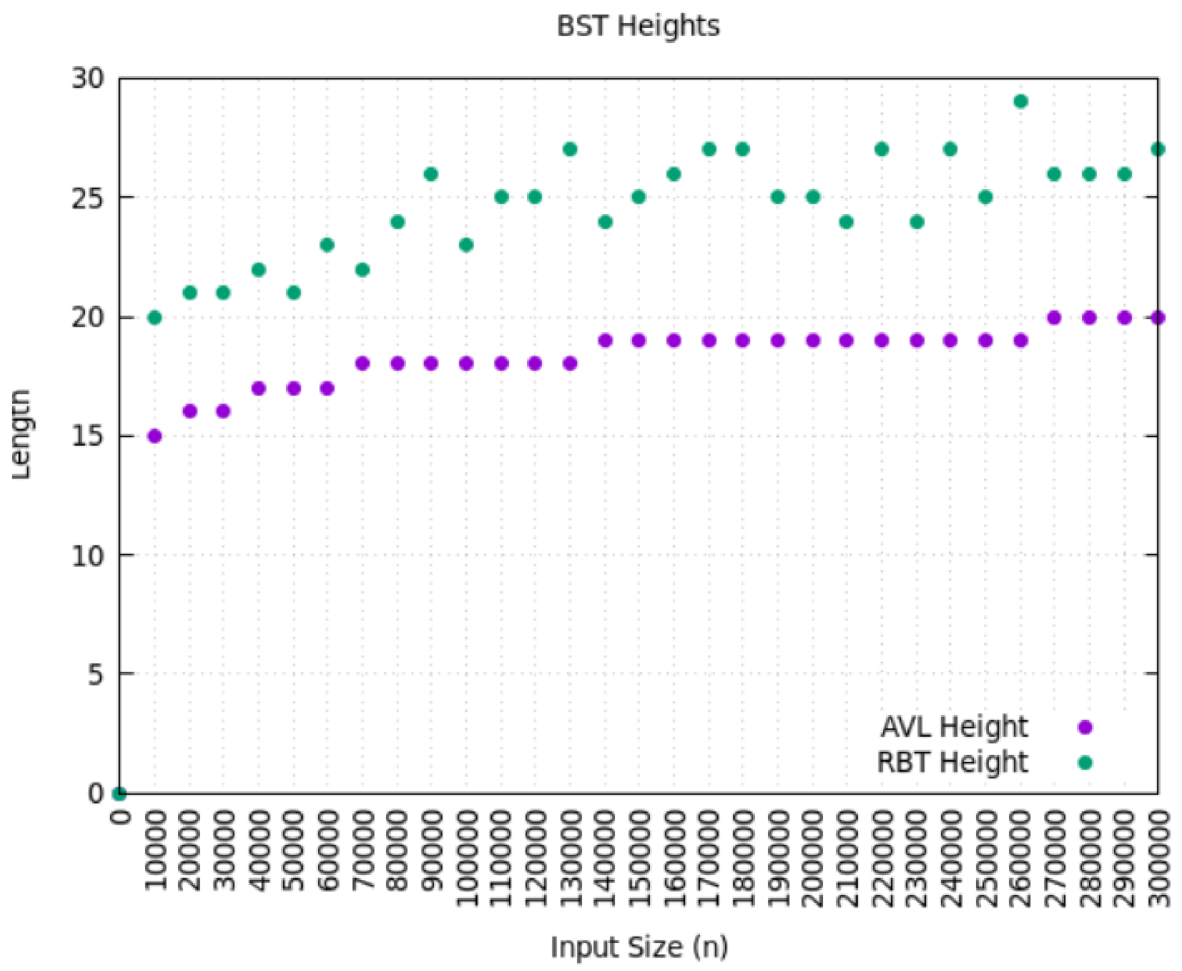
This graph came out better than the Remove graph, as the data is relatively uniform other than a few spikes here and there. Both AVL and RBT find values the same way, which has a complexity of O(log n) which can be seen in the graph. Just like remove, hash table's worst case complexity with find-value is much worse than the average case. If there were no load factor, and each bucket had multiple nodes, the data would resemble O(n) much closer.



I wasn’t expecting the different performances of find value to be so similar, but I suppose this is because regardless of the data structure, you still have to traverse the whole list. Either that or poor implementation, because hash tables complexity of O(n) should be slower than AVL collection and RBT collection O(log n). This could also point to me being wrong about RBT and AVL’s find-range complexity, because they appear closer to O(n) in the graph.



This graph came out as expected, and the results are pretty self-explanatory. The AVL and RBT trees are, by nature, in sorted order, and so the only "work" being done is grabbing the keys. The hash table on the other hand is not sorted, and after finding each key, we have to sort that sublist of keys.



This is fairly straight forward and expected. The AVL tree is much more rigidly balanced than the RBT tree, and a balanced tree is at its minimum height. Because the RBT isn't always balanced, it isn't always at its optimal height, hence why in the graph it is consistently taller.

Worst-Case Performance (Big O)

	Resizeable Array	Linked List	Sorted Array	Hash Table	BST	AVL	RBT
Add	O(1)	O(1)	O(n)	O(1)	O(log n)	O(log n)	O(log n)
Remove	O(n)	O(n^2)	O(log n)	O(n)	O(log n)	O(log n)	O(log n)
Find Value	O(n)	O(n^2)	O(log n)	O(n)	O(log n)	O(log n)	O(log n)
Find Range	O(n)	O(n^2)	O(log n)	O(n)	O(log n)	O(log n)	O(log n)
Sort	O(n^2)	O(n^2)	O(1)	O(log n)	O(1)	O(1)	O(1)

Implementation difficulties/challenges:

This was easily the hardest assignment of the semester, and about 50% of the work was dealing with seg_faults. To fix these, I had print statements all throughout my code that would tell me right where the program was before crashing. Usually this would lead me to an area when I was trying to access an element of a Null node (majority of my errors). Another difficult aspect was all of the different checks we had to do throughout the code. To solve this, I looked at the hw9_test file and would draw exactly what should be happening, then compared that drawing with a printout of what was actually happening in the code. This was the easiest way to try to figure out a weird error, and also helped build knowledge of the system as I was progressing with my implementation.