Josh Seward

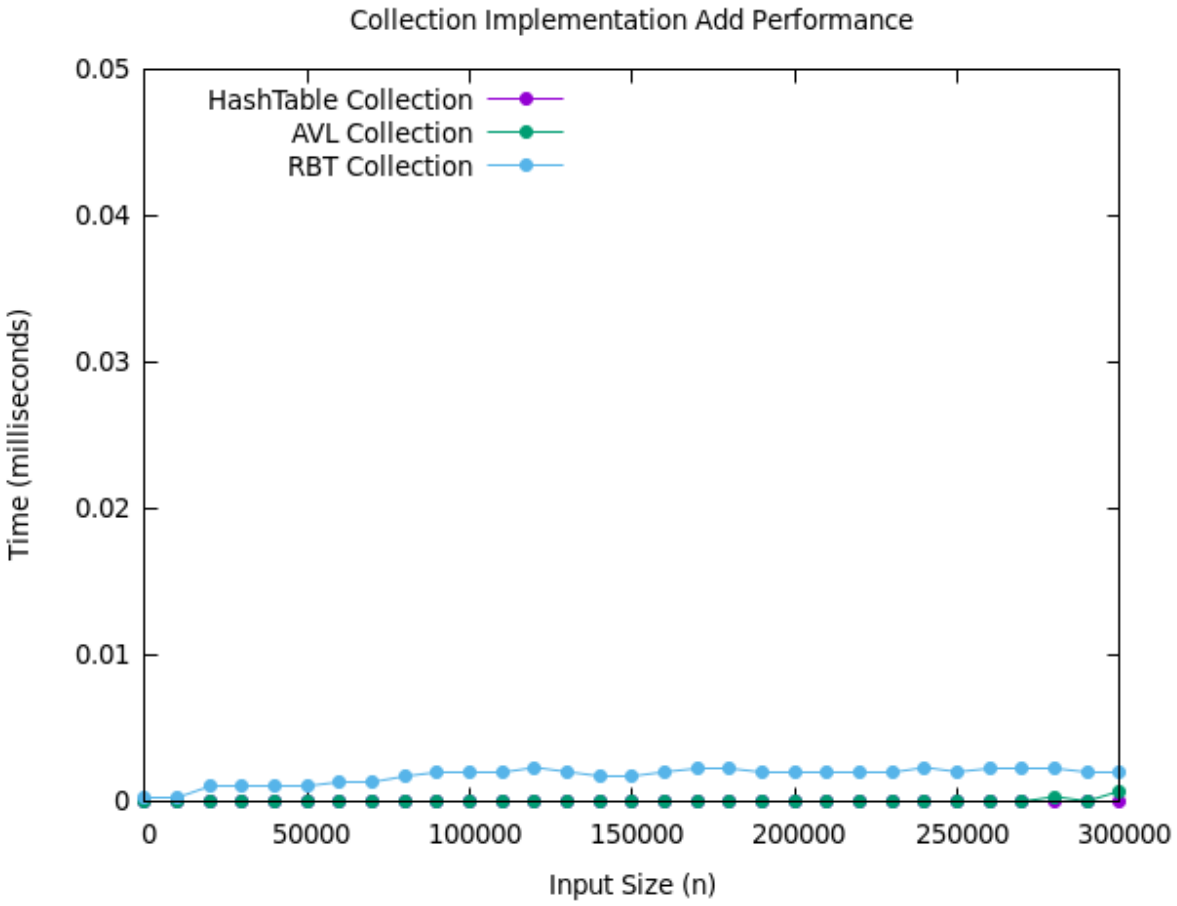
Dr. Bowers

CPSC223 (1)

15 December 2020

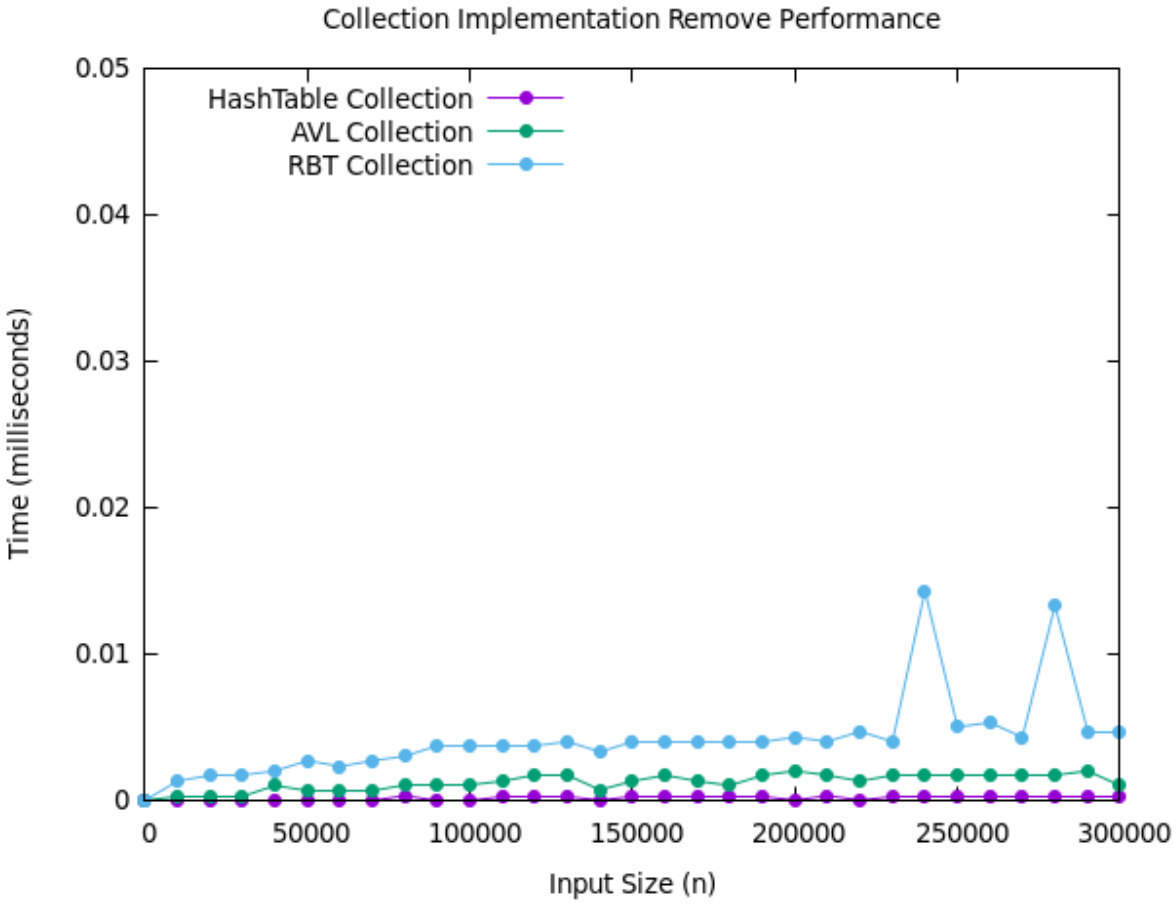
HW9 Write Up

Test 1: Add



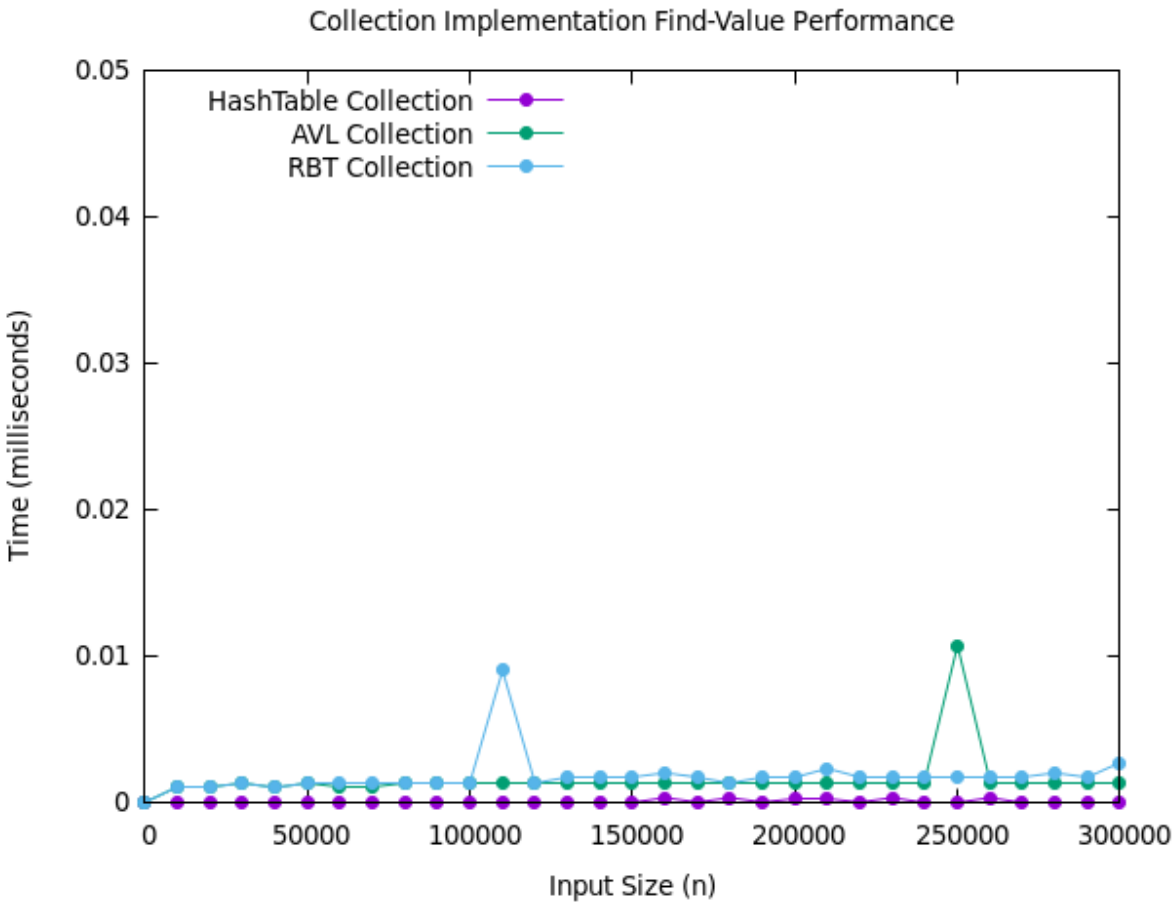
The RBT implementation of add is O(log(n)). This is because of how the tree is structured. It is slightly less efficient than the AVL Tree implementation because the height of a RBT is, on average, longer than the height of an AVL Tree even though the complexity of their add functions comes out to be O(log(n)). This influences the efficiency of the RBT implementation of add more than its other functions because nodes are always added as leaves, which means a root-to-leaf path must be taken for every new node added to the tree.

Test 2: Remove



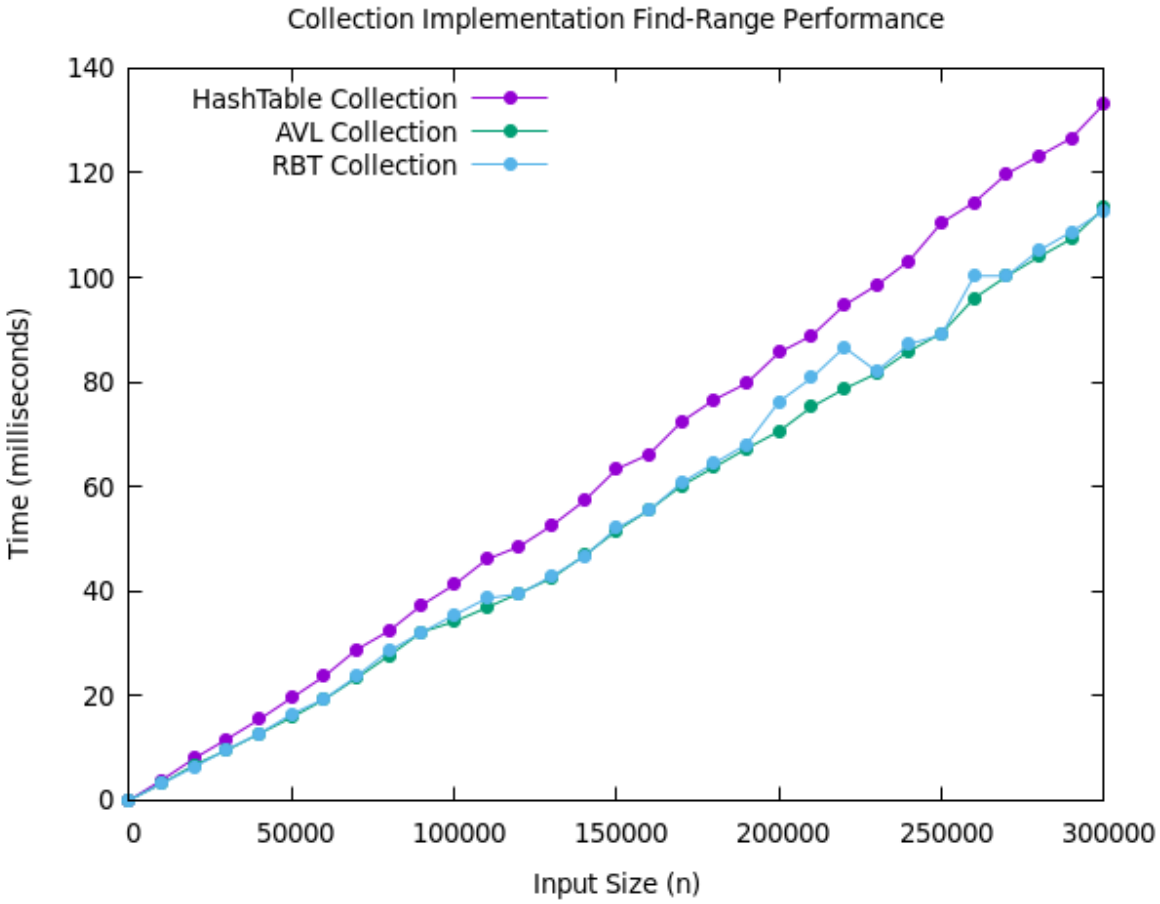
The RBT implementation of remove has a complexity of O(log(n)). This is because the average height of a RBT is log(n), and the longest path that would have to be traveled to find a node would be the length of the height. It is less efficient than the AVL collection because the constraints for a RBT are looser, and allow for trees with larger heights which makes it take longer to find the node to remove.

Test 3: Find Value



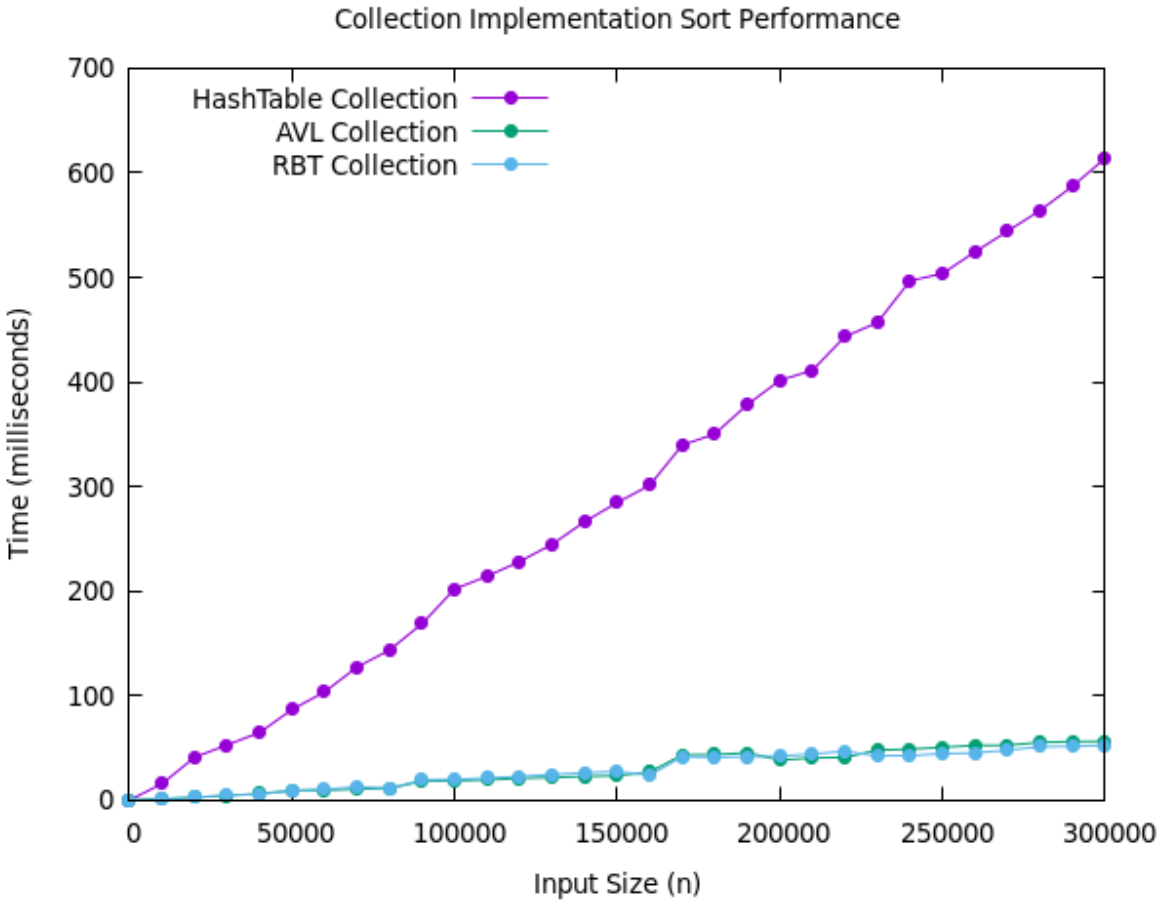
The RBT implementation of find-value is just as efficient as the AVL Tree implementation. This is because the average height of the trees comes out to O(log(n)). The difference between the AVL Tree and RBT implementations of find-value that emerge as the input size becomes increasingly larger comes from how the RBT is structured. The RBT root-to-leaf paths are typically longer than those of the AVL Tree, which means finding nodes further along these paths becomes more complex.

Test 4: Find Range



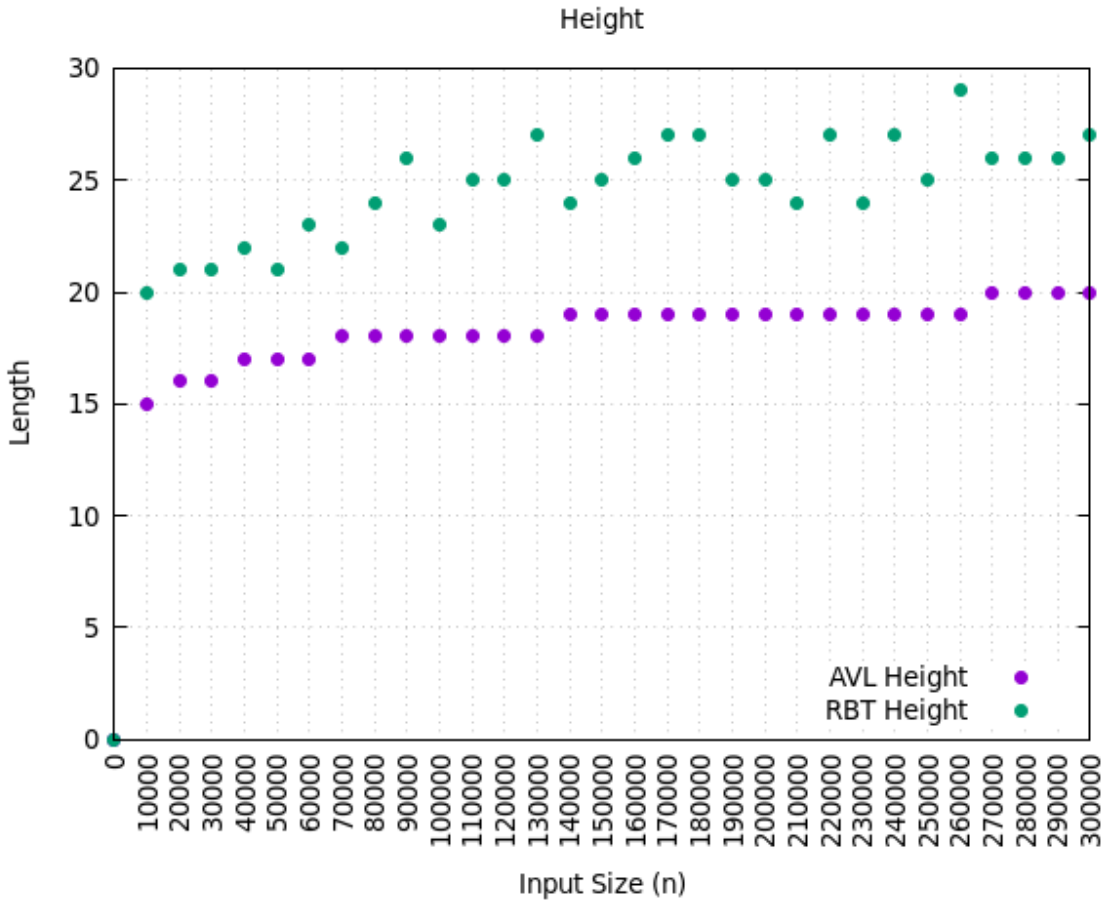
The RBT find-range implementation is O(nlog(n)) This is because the worst case for the find-range function is if the entire tree is in range, and therefore it all must be traversed and added to the output Array List. This is almost identical to the AVL Tree implementation, however it can be slightly slower in some instances because of the relaxed constraints of a RBT.

Test 5: Sort



The RBT implementation of sort has a complexity of O(nlog(n)). This is the same as the AVL Tree implementation since both trees are kept in sorted order. The RBT implementation is faster than the Hash Table implementation even though the complexity for the Hash Table sort function is O(nlog(n)) as well. This is because the Hash Table Collection is not kept in any sorted order, therefore the a separate sorting algorithm must be called, which adds in a bit more work, and decreases the efficiency of the sort function as a whole.

Test 6: Height



The average height grows at a rate of O(log(n)) for a RBT. The height values are much more spread than that of the AVL Tree because the RBT constraints are looser than those of the AVL Tree. These relaxed constraints are also the explanation for why the height of a RBT is larger than the height of an AVL Tree with the same input size.

Operation	*Collection Implementation*						
	Array List	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)	O(log(n))	O(1)	O(log(n))	O(log(n))	O(log(n))
Find Value	O(n)	O(n)	O(log(n))	O(1)	O(log(n))	O(log(n))	O(log(n))
Find Range	O(n)	O(n)	O(n)	O(n)	O(nlog(n))	O(nlog(n))	O(nlog(n))
Sort	O(n)	O(n)	O(nlog(n))	O(n)	O(nlog(n))	O(nlog(n))	O(nlog(n))

Challenges:

The main challenge I faced during this assignment was with the rebalancing. I had a bit of trouble understanding how to implement the different cases where we need to rebalance the tree, mainly having to do with the implementation of a parent pointer in the rotation helper functions and the remove cases.