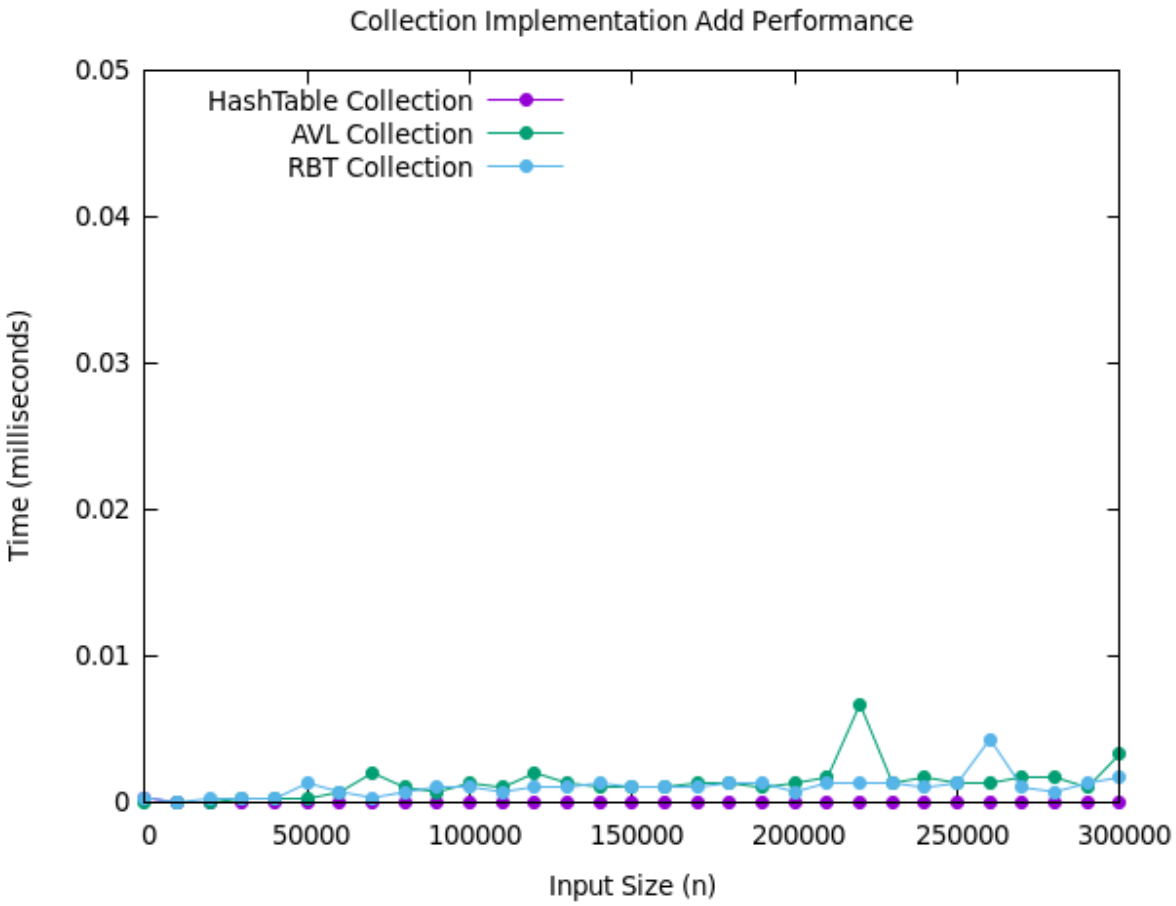
Scott Tornquist

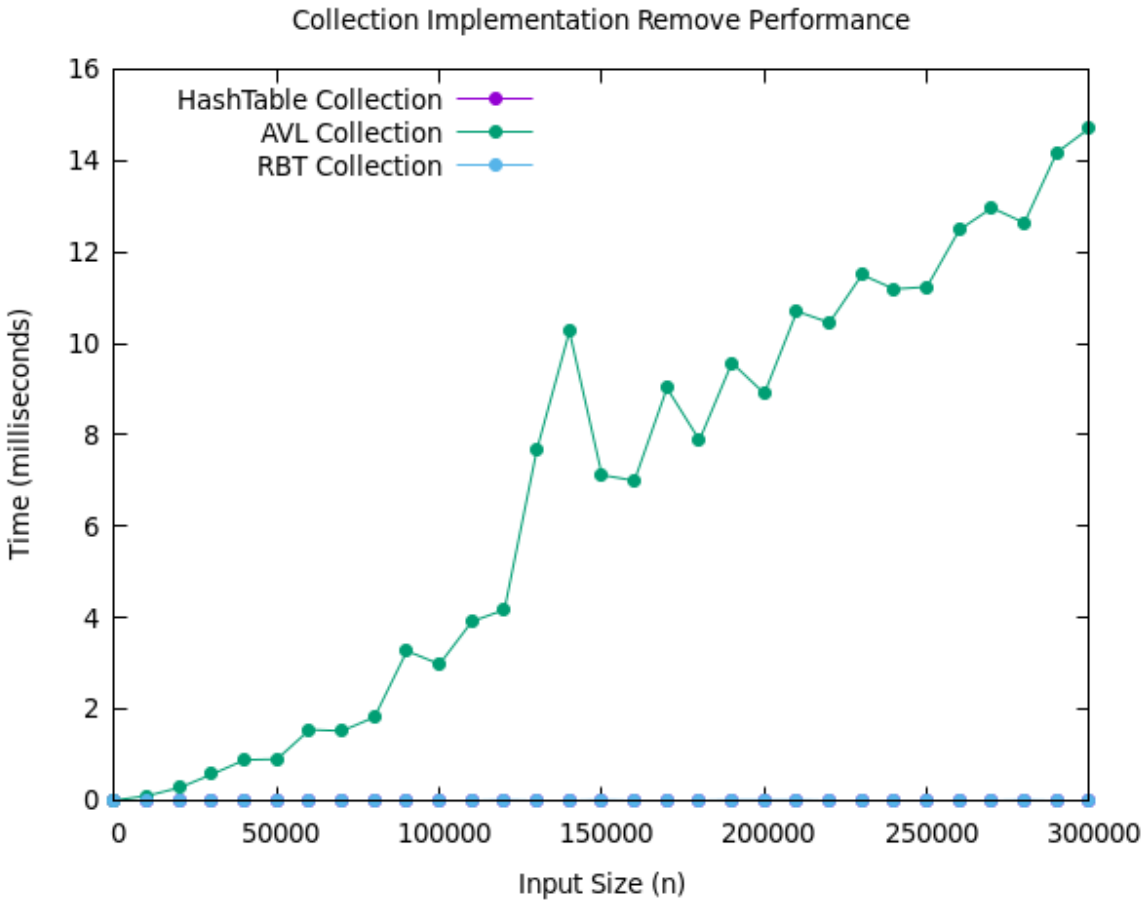
Abstract Data Structures

HW #9

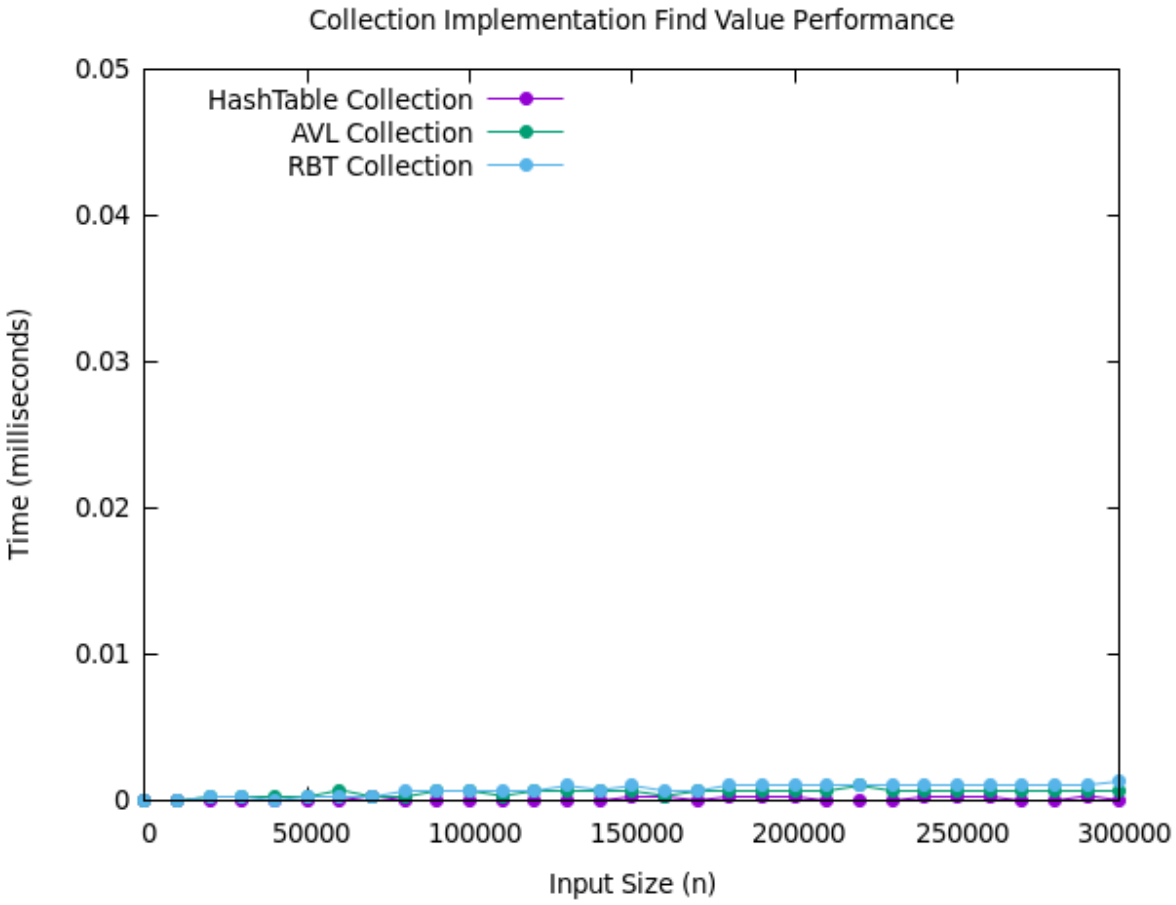
Write up



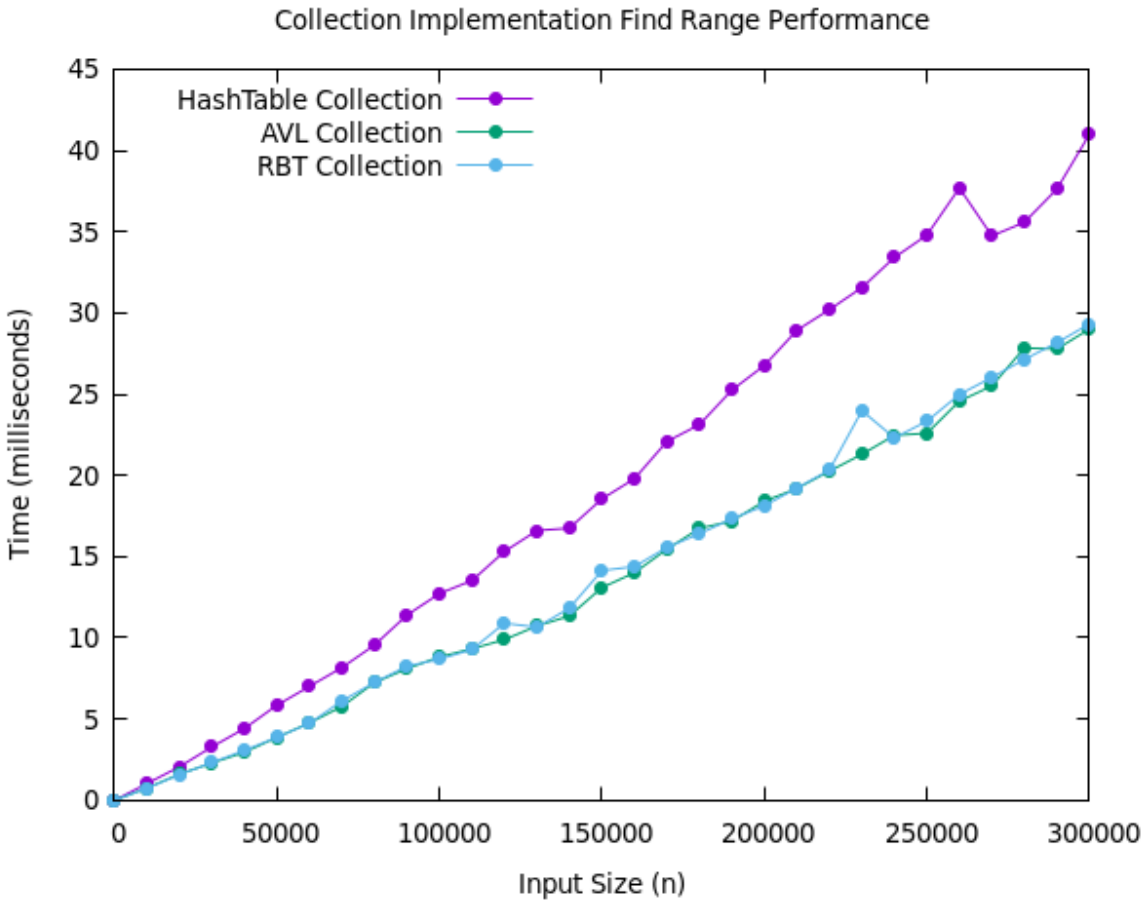
This graph looks this way because in the worst case to add to a RB Tree we must iterate down log(n) nodes to reach the end to insert. This is because the tree is practically balanced. It is actually fast than the AVL tree add because we use iteration and rebalance on the way down so there is no backtracking in the RB Tree. Rebalancing does not affect the big-O.



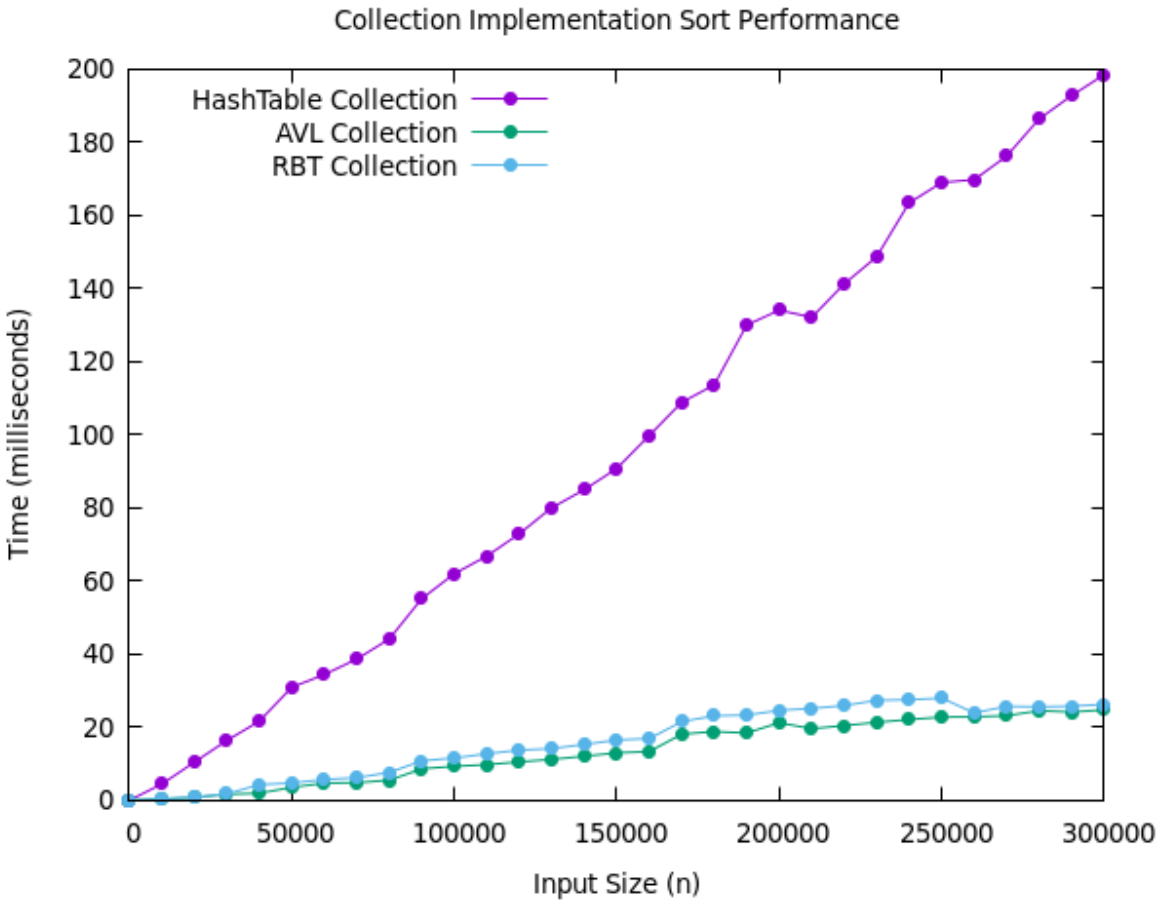
This graph looks this way because in the worst case to remove from an RB Tree we must iterate down log(n) nodes to reach the end to remove. This is because the tree is practically balanced. It is faster than the AVL tree remove because we use iteration and rebalance on the way down so there is no backtracking in the RB Tree. I know the graph for the AVL remove is messed up here. It should be down with the others.



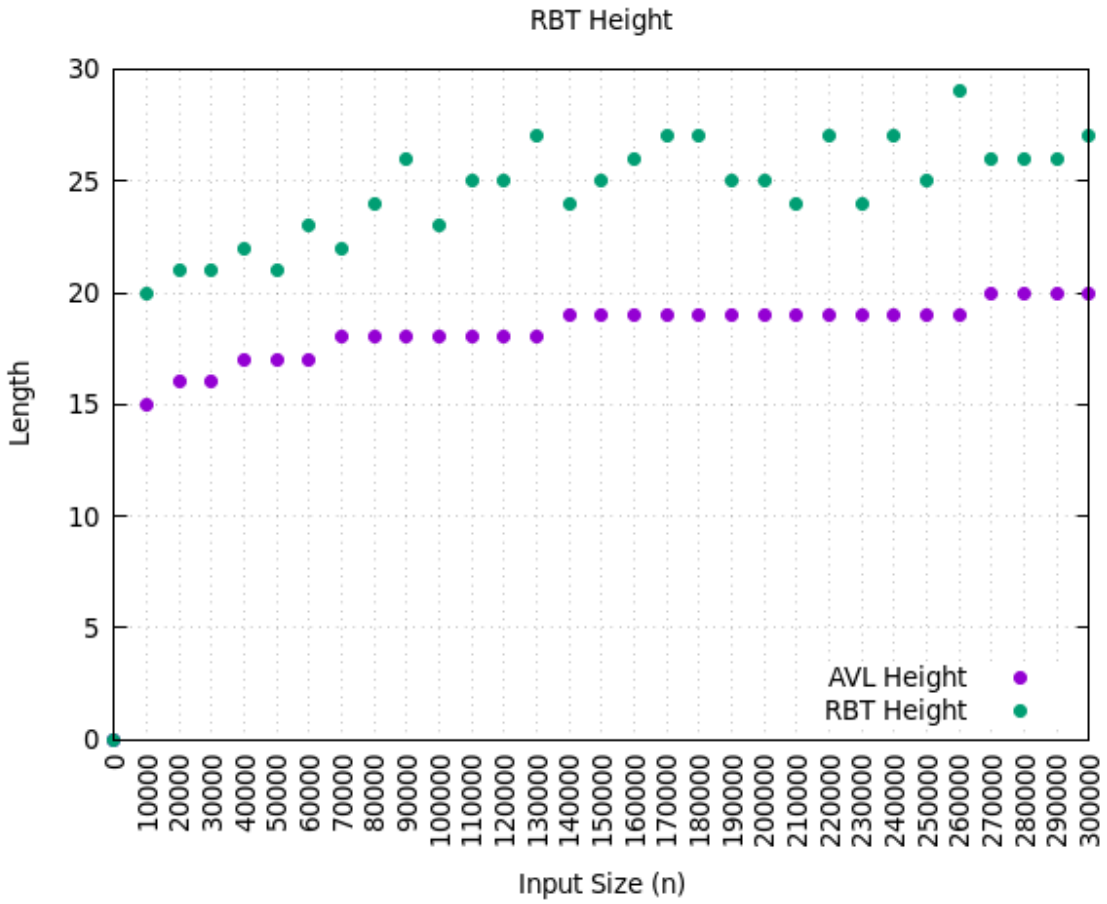
This graph looks this way because in the worst case to find a value from an RB Tree we must iterate down log(n) nodes to reach the node to find. This is because the tree is practically balanced and has a height of log(n). It is slightly slower than the AVL tree because RB Trees can be a little bit longer.



This graph looks this way because in the worst case to find a range from an RB Tree we must navigate down n nodes to reach all the nodes and recursively add every node to the list in the worst case. This is because the tree is practically balanced. It is slightly slower than the AVL tree because RB Trees can be a little bit longer.



This graph looks this way because in the worst case to sort every node from an RB Tree we must navigate down n nodes to reach all the node to add to the return list. The tree is always sorted so the actual sort takes no time. It is slightly slower than the AVL tree because RB Trees can be a little bit longer.



This graph turned out this way because the constraints for RB trees are not as tight at AVL. This means they tend to be taller trees. This is shown in this graph.

Operation	Collection Implementation						
	Array List	Linked List	Sorted Array	Hash Table	Binary Search Tree	AVL Tree	RB Tree
Add	O(1)	O(1)	O(n)	O(1)	O(n)	O(Log(n))	O(Log(n))
Remove	O(n)	O(n^2)	O(n)	O(n)	O(n)	O(Log(n))	O(Log(n))
Find-Value	O(n)	O(n^2)	O(log(n))	O(n)	O(n)	O(Log(n))	O(Log(n))
Find-Range	O(n)	O(n^2)	O(n)	O(n)	O(n)	O(n)	O(n)
Sort	O(n^2)	O(nlog(n))	O(n)	O(nlog(n))	O(n)	O(n)	O(n)

RB trees are special because to automatically rebalance themselves during add and remove on the way down to operate. This means they can be recursive and do not have to back track back up. This means even though they tend to be taller trees and the Big-O results are the same as AVL trees, they are faster to add and remove. This is because RB Trees have a faster log(n) that does not back track. They are identical for finding and sorting.

This project was definitely the hardest of the year. But I have gotten so much better and learned so much from this class that I think this was one of my best projects. I realized that it is better to take my time to write high quality code that I understand and is documented well so I can go back and debug much easier. Because of my improvement this project went pretty well for me. I am proud. Thank Dr. Bowers. This was definitely my fav class this semester.

THANK YOU DR. BOWERS!