CPSC 223 Red Black Trees

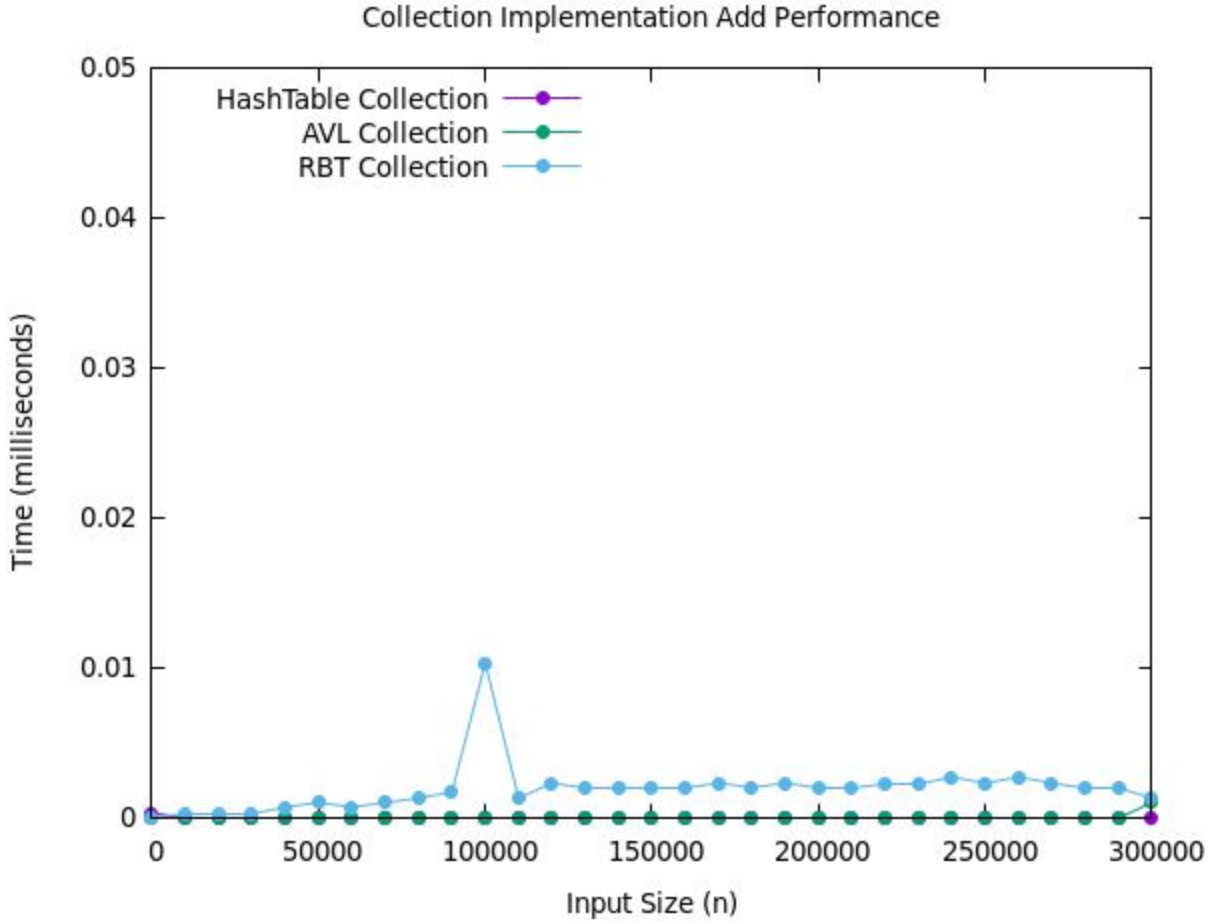
Wesley Muehlhausen

Reflection

- This assignment was fairly tough just because of the amount of conditions there are to consider. I had problems with add but I figured it out by testing using the class example which told me which rotations should be happening when, and subsequently tweaking the if statement conditions to make it work.
- I also had problems in the remove function because I could get most nodes to delete properly, but I kept on running into problems whenever I had deletions at the top of the tree. It took such a long time to work out all of the specific cases but I eventually got it to work. A lot if it was redundant Which made it really long.
- Looking at the notes were extremely necessary for this. This was by far the hardest one to get working. I had to create a print function that made it a lot easier to test with.

Collection Implementation Worst Case Time Complexities

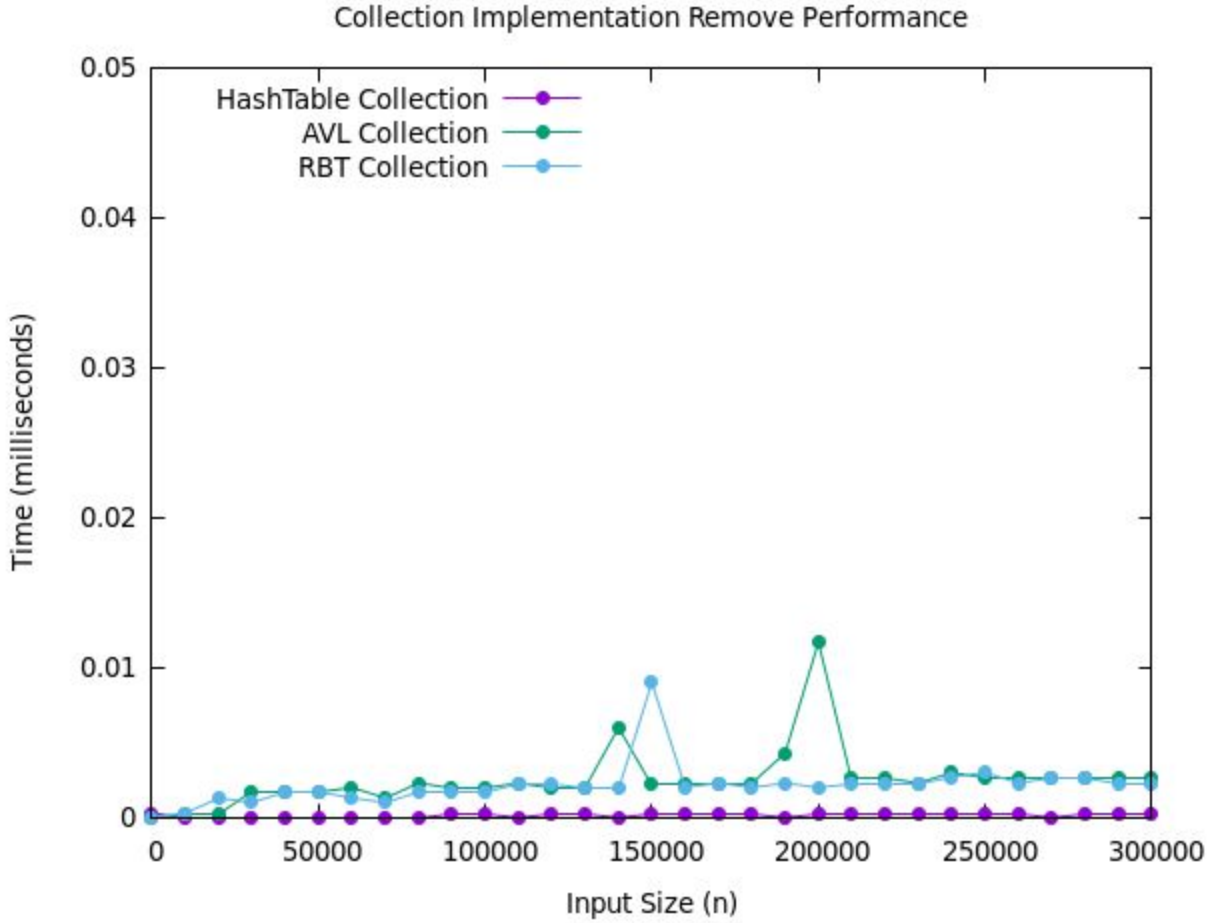
Operation	ArrayList C.	LinkedList C.	Sorted Array (Bin Search)	Hash Table	BST	AVL	RBT
Add	O(1)	O(1)	O(n)	O(1)	O(n)	O(logn)	O(logn)
Remove	O(n)	$O(n^2)$	O(n)	O(1)	O(n)	O(logn)	O(logn)
Find-value	O(n)	$O(n^2)$	O(logn)	O(1)	O(n)	O(logn)	O(logn)
Find-range	O(n)	$O(n^2)$	O(n)	O(n)	O(n)	O(n)	O(n)
Sort	$O(n^2)$ Quick Sort	$O(n^2)$ See explanation	O(n)	$O(n^2)$ Quick Sort	O(n)	O(n)	O(n)


Collection Implementation Add Performance
HashTable Collection
AVL Collection
RBT Collection
Time (milliseconds)
0.05
0.04
0.03
0.02
0.01
0
0
50000
100000
150000
200000
250000
300000
Input Size (n)

Add Operation Graph Explanations

RBT - Traverses the tree iteratively similar to BST. The reason why this operation is O(logn) and isn't O(n) is because the height of the tree is managed which makes cases where BST turns into a linked list not possible. So the height will cause the add to at most need logn moves to get to the bottom of the list.
Hash Table - Uses the hash function to find the index of the bucket where the kv pair goes. Since the load factor threshold is set at 0.75, on average, there will mostly be buckets size 0 or 1. Regardless, adding to the front of the bucket means that it will be O(1). When the load factor threshold is above 0.75, the hashtable needs to resize and rehash which is why there are spikes in the graph, because this takes n moves because every value needs to be rehashed.
AVL - Traverses the tree recursively. Since the list's height can be no worse than 1.5 the best case height, the height cannot get out of hand like the binary search tree. This means that in the worst case, the add function takes O(logn) which is why it performs so well in the graph.

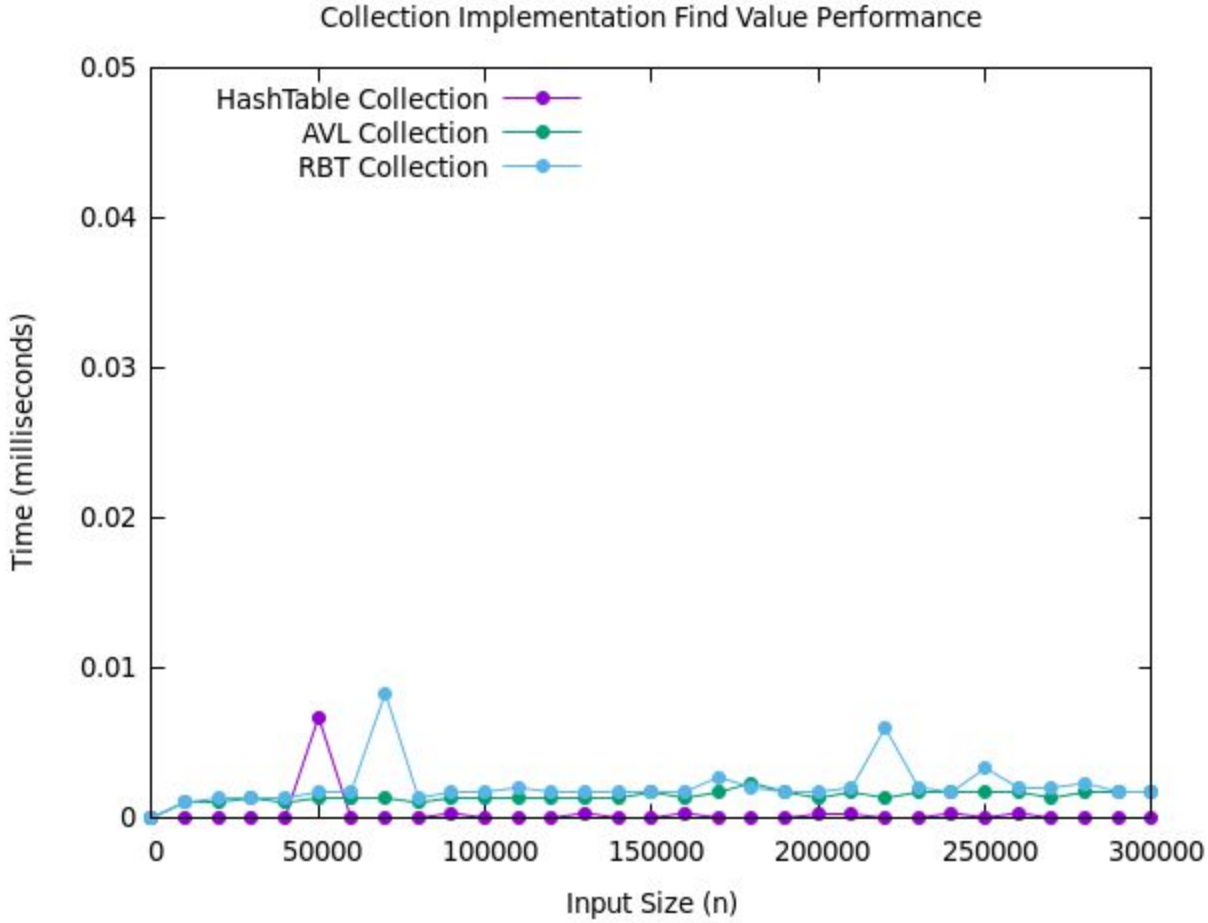
Graph Hypothesis - Looking at graph 2, we have the top performers. These adds are either O(logn) or for Hash Table, O(1). Hash Table performs the best which makes sense since there is no traversal (except adding to buckets which should only have 0 or 1 nodes in it). Since RBT has less height constraints as AVL, I would guess that this is the reason for AVL outperforming it; due to having to traverse less tree levels. Hash Table though clearly outperforms them with it's O(1) adding.


Collection Implementation Remove Performance
HashTable Collection
AVL Collection
RBT Collection
Time (milliseconds)
0.05
0.04
0.03
0.02
0.01
0
0
50000
100000
150000
200000
250000
300000
Input Size (n)

Remove Operation Graph Explanations

AVL - Traverses the tree recursively. Since the list's height can be no worse than 1.5 the best case height, the height cannot get out of hand like the binary search tree. This means that in the worst case, the remove function takes O(logn) which is why it performs so well in the graph.
RBT - This remove is similar to AVL remove in complexity, but to uses iterative traversal to find the node to remove. This is a O(logn) operation because it, like AVL, manages its height with rotations in order to keep a O(logn) height.
Hash Table - Uses the hash function to find the index of the bucket where the kv pair goes. Since the load factor threshold is set at 0.75, on average, there will mostly be buckets size 0 or 1. This means that removing from buckets of size 0 or 1 which is O(1). Since the hash table does not get any bigger, there is no need to resize and rehash which is why there are no spikes in the graph and makes it incredibly efficient.

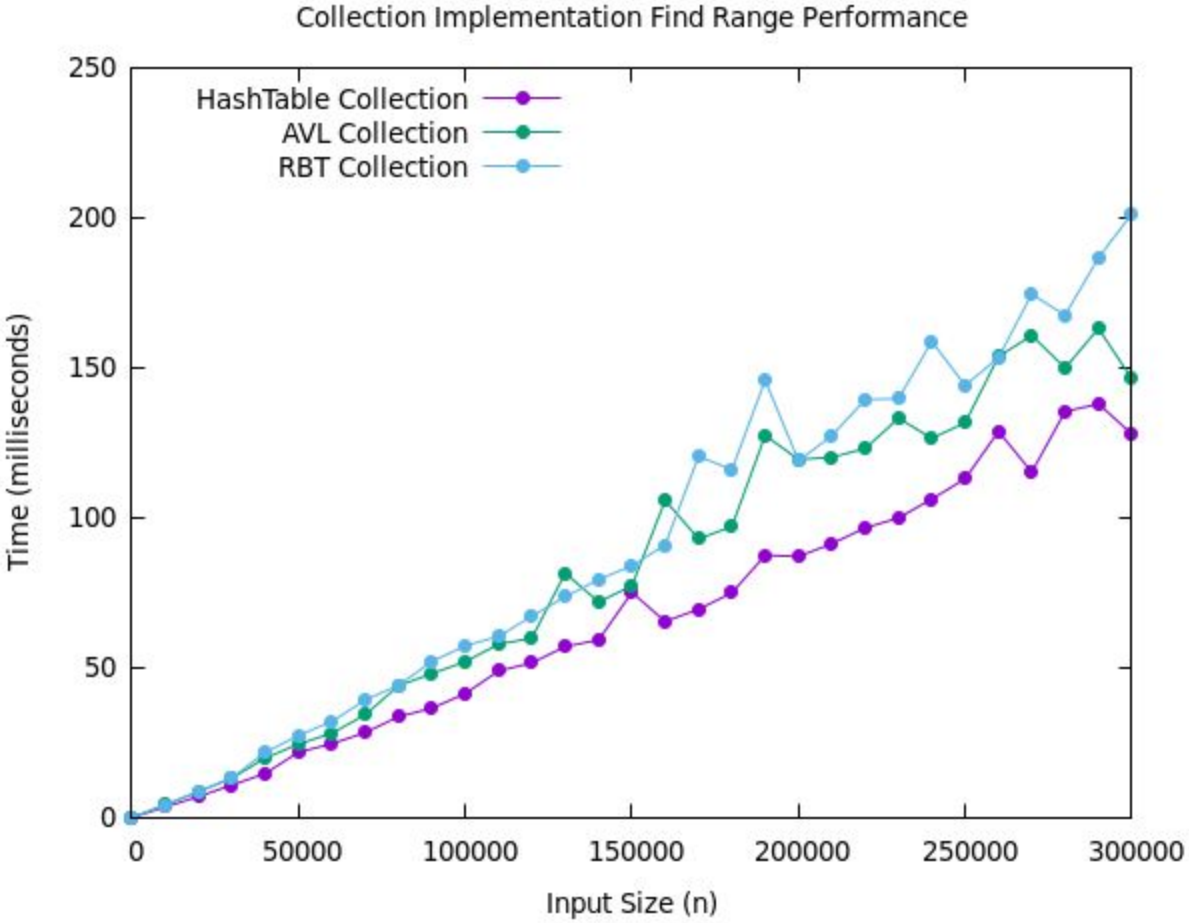
Graph shows the best performers. Hash Table again outperforms AVL and RBT. This is again due to its ability to keep its buckets filled with around 1 or 0 nodes as well as its hash function which gives an index in one move which all contributes to its O(1) remove. Also it doesn't not have to resize and rehash which is another time cut. AVL and RBT seem to have similar performances probably because of how similar the structures and heights are to each other.


Collection Implementation Find Value Performance
HashTable Collection
AVL Collection
RBT Collection
Time (milliseconds)
0.05
0.04
0.03
0.02
0.01
0
0
50000
100000
150000
200000
250000
300000
Input Size (n)

Find Value Operation Graph Explanations

AVL - Traverses the tree recursively. Since the list's height can be no worse than 1.5 the best case height, the height cannot get out of hand like the binary search tree. This means that in the worst case, the find function takes O(logn) which is why it performs so well in the graph.
RBT - Traverses the tree iteratively. Height cannot get longer than logn which makes this also a O(logn) operation because it searches down the list. In this case, recursion isn't any faster which is why it's similar to AVL.
Hash Table - Uses the hash function to find the index of the bucket where the kv pair is at. Since the load factor threshold is set at 0.75, on average, there will mostly be buckets size 0 or 1. This means that finding from buckets of size 0 or 1 will be worst case O(1).

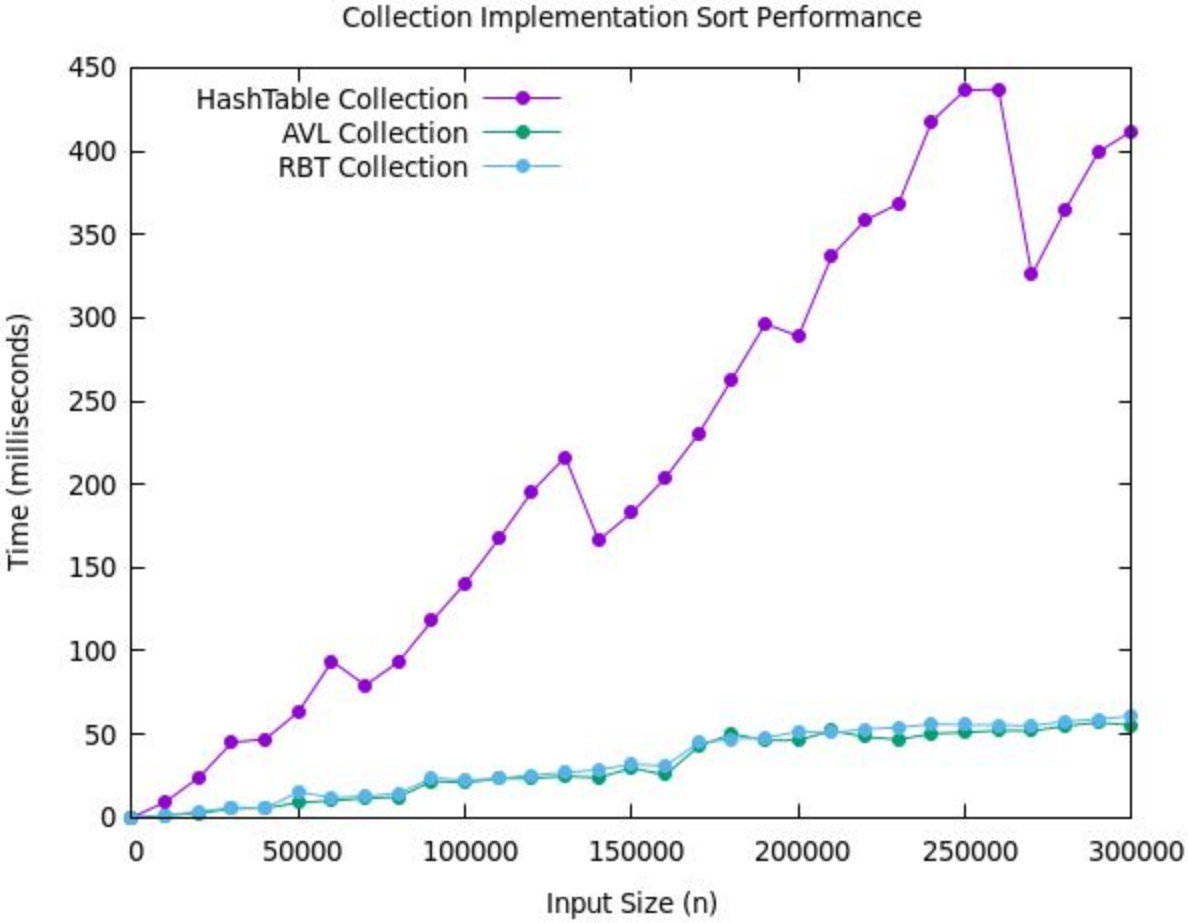
AVL and RBT don't have a problem with turning into a linked list because of height management which makes find much quicker. Looking at the graph, it is very similar to the add and remove operations of this graph where Hash Table has the best performance followed by AVL and then RBT. Hash Table is clearly faster considering it is O(1). As for the two trees, I expect the RBT to be a little slower because the height of the tree is less strict so it could be slightly bigger and cause a slower find time.


Collection Implementation Find Range Performance
HashTable Collection
AVL Collection
RBT Collection
Time (milliseconds)
Input Size (n)
0
50
100
150
200
250
0
50000
100000
150000
200000
250000
300000

Find Range Operation Graph Explanations

AVL, BST, RBT - Recursively goes through the tree and finds the values of keys within a certain range. There is no extra fast way to do a find range operation because every node needs to be visited unless out of range. But regardless, if the range spans the entire tree, then each node will be included in the range which means it will be O(n).
Hash Table - This also cannot use the hash function for the find range operation. This means that hash table also has to iteratively goes through all of the values to see if they are in the range, which is why this is also O(n).

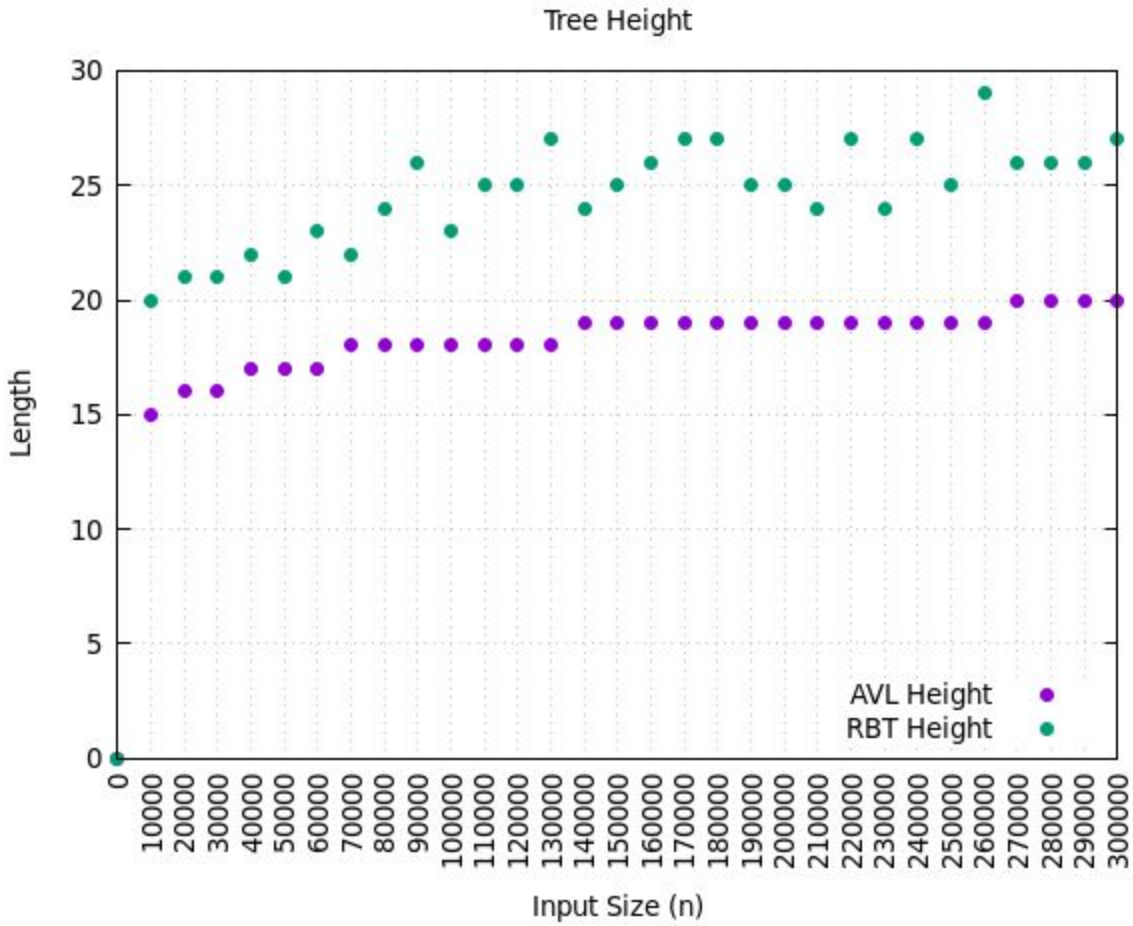
All of the graphs perform pretty similarly. There is no fast way to do find range since every node has to be found so worst case will be finding the whole list which is O(n). Hash table search is the fastest probably because it can hash to the location of the first key which gives it a headstart, and then after it is out of range it can stop searching. Both of the trees search recursively but that doesn't help and does not allow for a jump to first key or exit early option like in Hash Table.


Collection Implementation Sort Performance
HashTable Collection
AVL Collection
RBT Collection
Time (milliseconds)
Input Size (n)
0
50
100
150
200
250
300
350
400
450
0
50000
100000
150000
200000
250000
300000

Sort Operation Worst Case Explanations

AVL, BST, RBT - Recursively goes through the tree and gathers all of the keys in the tree because the tree is already sorted. There is no extra fast way to do a sort operation because every node needs to be visited. But regardless, it will be O(n).
Hash Table - Same as ArrayList. Calls keys and defers to quicksort which is $O(n^2)$. I think the reason that this performs better than ArrayList because buckets are somewhat sorted already.

Although, HashTable also defers to Quick Sort, I think the reason it performs so much better than ArrayList is because the keys are somewhat sorted into their buckets which makes it so the worst case (sort a reversed list) won't happen like it could in ArrayList. BST, RBT, and AVL have no performance problems since they are already sorted and only has to call keys which returns the list. But overall, Hash Table is still an O(n^2) method which shows why it does exponentially worse than the two tree sorting methods which is O(n).


Tree Height
Length
Input Size (n)
AVL Height
RBT Height
0
5
10
15
20
25
30
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
110000
120000
130000
140000
150000
160000
170000
180000
190000
200000
210000
220000
230000
240000
250000
260000
270000
280000
290000
300000

Collection Implementation TREE HEIGHT performance

Graph Hypothesis - This graph makes sense. As the number of nodes gets incredibly big, the trees size grows slower and slower for AVL. This is because for every new tree level, the number of leaf nodes are doubled from previous. At the beginning that does not seem like much, but hypothetically if a tree had 20,000 leaf nodes on the same level, one more level of height would allow the tree 40,000 new nodes. This shows how fast this is O(logn) compared to Arrays and Linked lists O(n) (in some cases). Adding all those nodes would only make tree traversing operations only have to go down one more level. This same principle applies to Red Black Trees too. Even though the heights are a little greater, they are still comparable and still O(logn). This really exploits the problem with the BST. If adding in order or reverse order, the tree will continuously grow in one direction with no two child parents which means it is essentially a list. Looking at the height, for 30,000 nodes AVL and RBT are in their 20's for height. On the other hand, BST is at a height of 3700. This shows the power of a balanced tree and shows how much better an O(logn) root to leaf traversal is than a O(n) one.