HW7 - Binary Search Trees

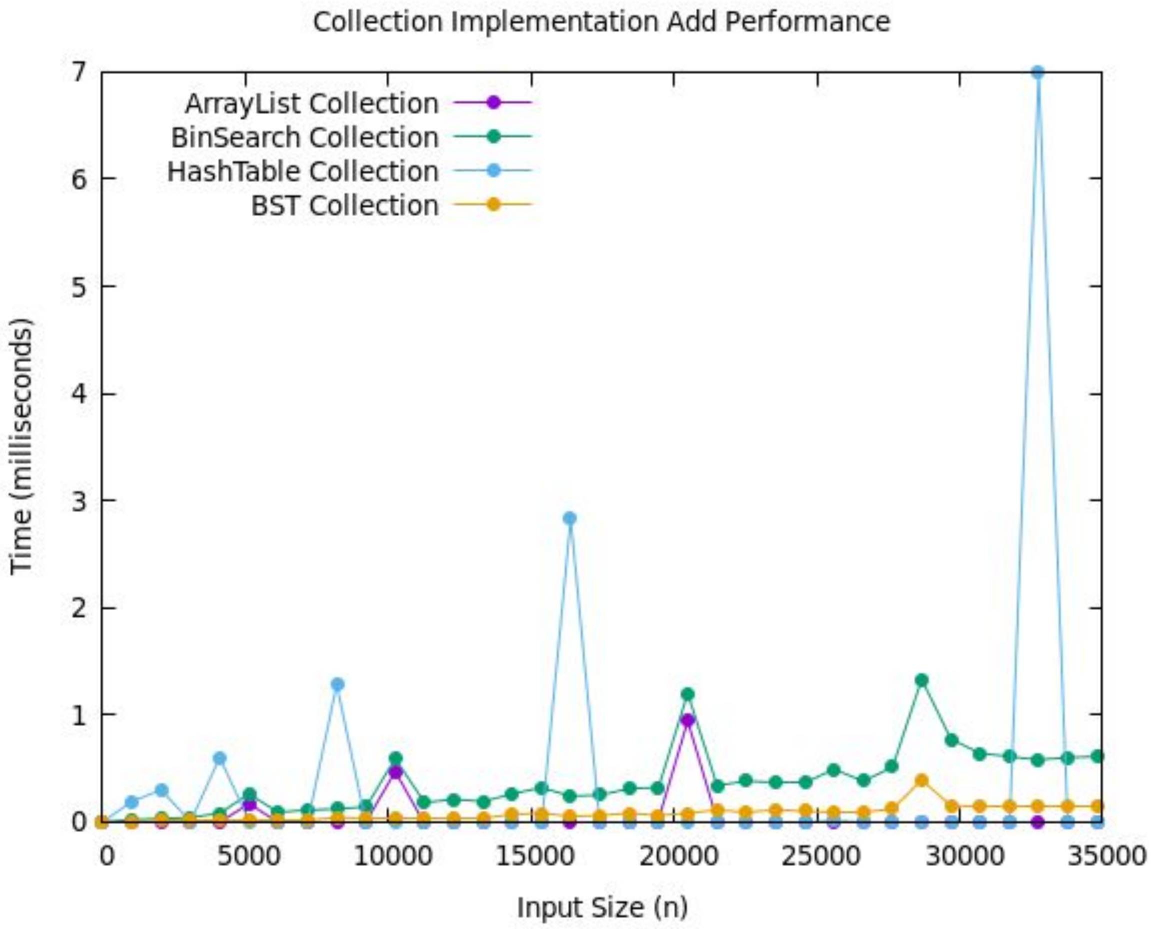
CPSC 223 - HW7 - Wesley Muehlhausen

Challenges and Issues

- When creating this assignment, it went pretty well because I understand the concept pretty well. I would say the hardest part of this was making the assignment operator, copy constructor, and the resize and rehash function. My assignment operator didn’t work because it wasn’t setting the root to nullptr after deleting it with the make empty helper function. This took me a while to figure out because I thought it was my make empty function that wasn’t working. That is where I spent most of my time.
- The add function was fairly easy to implement. I also did good with the remove functions, but I messed up the case where the delete function deletes a higher up node and has to find the inorder successor. That one took me a while to figure out, but eventually I got it all to work.

Collection Implementation

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

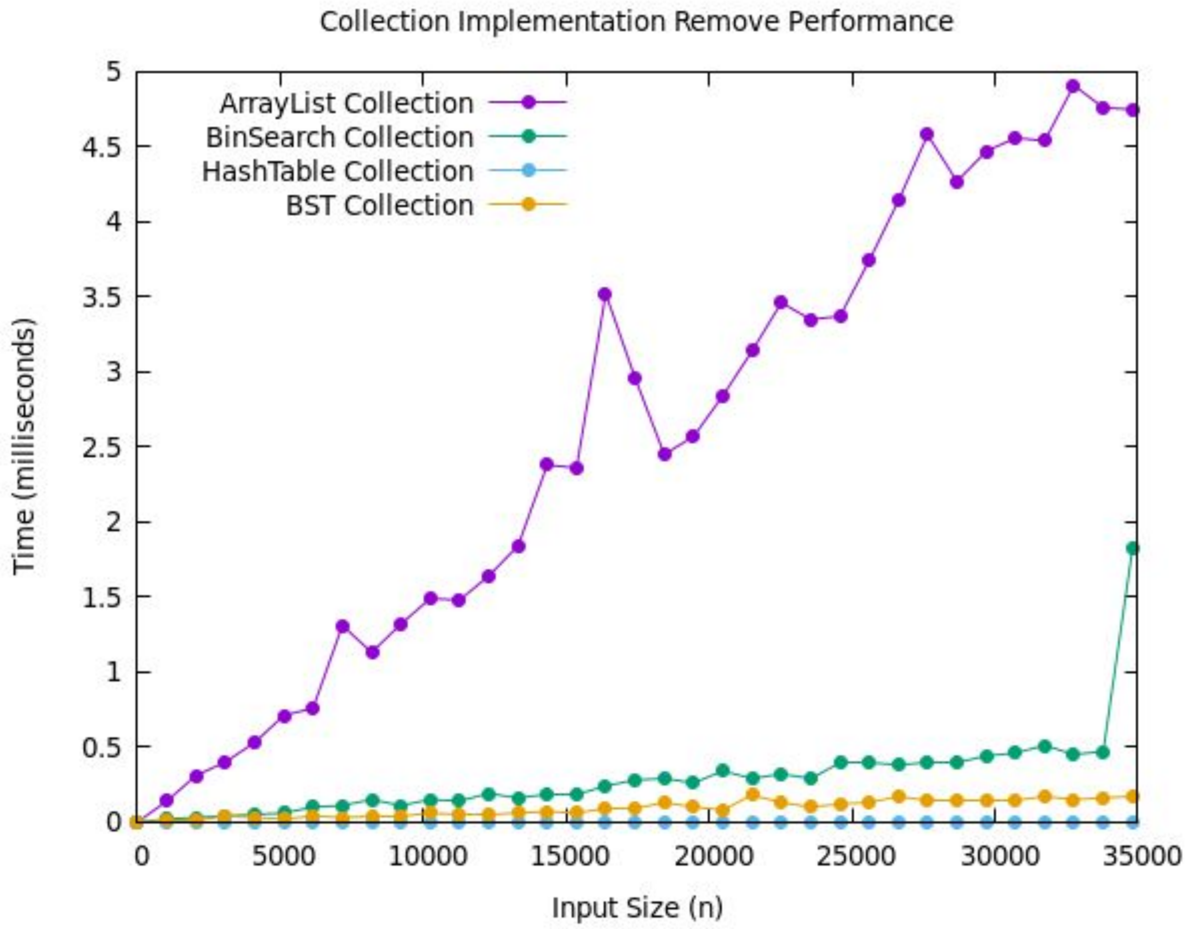


Collection Implementation Add Performance
ArrayList Collection
BinSearch Collection
HashTable Collection
BST Collection
Time (milliseconds)
Input Size (n)
0
1
2
3
4
5
6
7
0
5000
10000
15000
20000
25000
30000
35000

Collection Implementation ADD performance

Graph Hypothesis - Looking at the graph all of the operations did well. This graph makes sense though because ArrayList and HashTable should perform better than Binary Search because it doesn't have to add in order which makes it an O(1), while Binary Search is O(n) because it has to add by index to maintain its sorted list. The spikes in all of the graphs is most likely to do with the resizing. For BST, it could have been worse considering this is an O(n) operation but that should only be the case when adding in a continuous path, such as if every added key is larger than all of the keys on the tree, which basically turns it into a linked list.

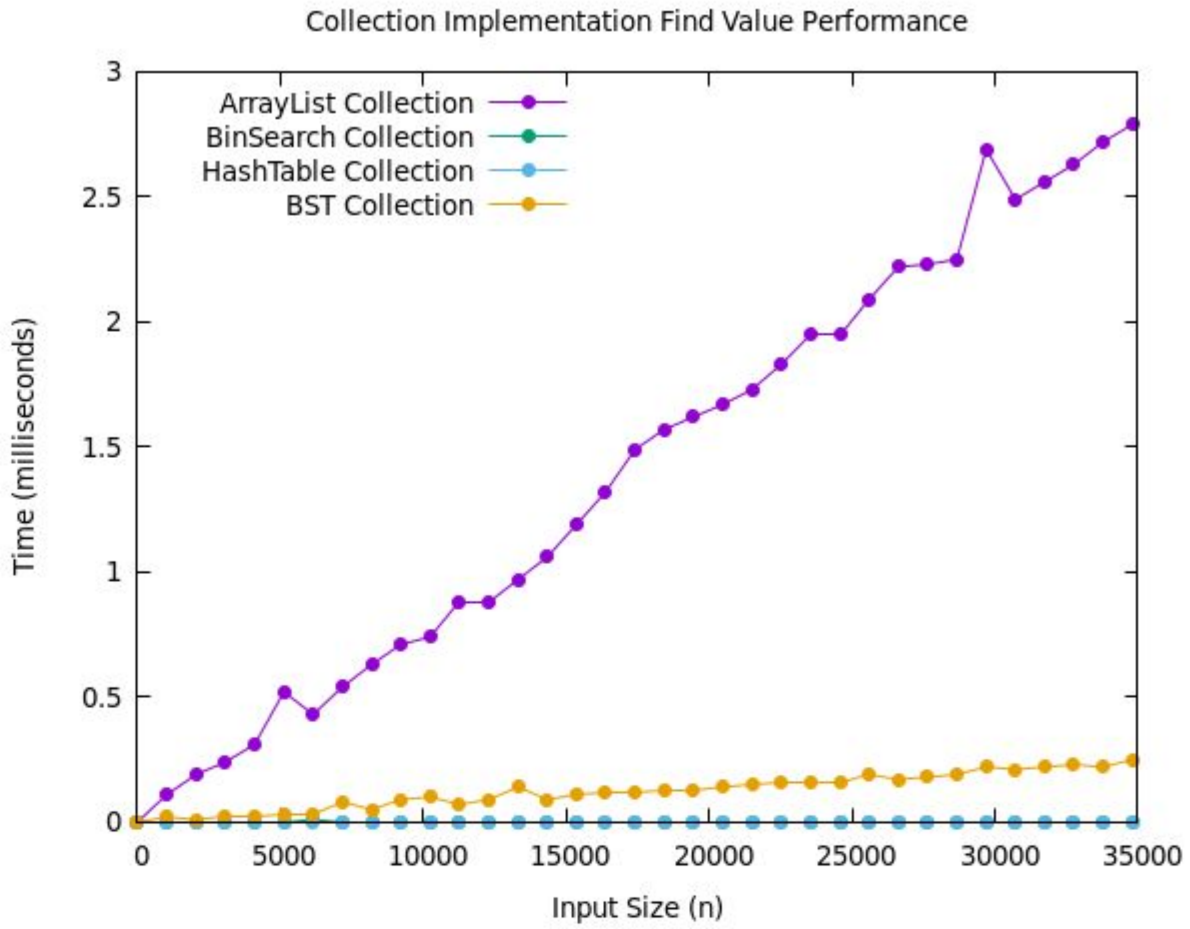
AVL - Traverses the tree recursively. Since the list's height can be no worse than 1.5 the best case height, the height cannot get out of hand like the binary search tree. This means that in the worst case, the add function takes O(logn) which is why it performs so well in the graph.
ArrayList Collection - Uses the ArrayList add function which is O(1) because it can add by index without any need for traversal. We do see minor spikes in the graph because when the ArrayList gets filled up, it has to resize which means all of the values have to be copied over to the bigger ArrayList which takes n moves.
LinkedList Collection (for personal use) - Uses the LinkedList add function which is O(1) because it adds to the end of the list using the tail pointer. This means no traversal is necessary which is why it is not n moves.
Binary Search - Uses the binary search function to find the index of where it should be added in the list which costs O(logn), with the index, the ArrayList add function is called which is time complexity O(n) because it is add by index. Also, there is a potential call to get() which is also O(1) so overall it is O(n). When there are spikes in the graph is when the ArrayList needs to resize.
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.
BST - Traverses the tree to find the next open place to add. Since there is no height regulation, The worst case add could be O(n) because there could be one continuous string of parent to child which would be length n. If there is a more normal variety of kv pairs added, then the add function would perform better as O(logn)


Collection Implementation Remove Performance
ArrayList Collection
BinSearch Collection
HashTable Collection
BST Collection
Time (milliseconds)
Input Size (n)
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
0
5000
10000
15000
20000
25000
30000
35000

Collection Implementation REMOVE performance

Graph Hypothesis - This graph makes sense. ArrayList performs worse than Binary Search, BST, and Hash Table even though they are both O(n) operations and Hash Table is O(1). I think this is because ArrayList has to traverse to the key, remove, and then shift the rest of the list over which is always going to take n moves, while on the other hand, Binary Search uses the binary search function to find the index of the array which is only O(logn) and then has to shift over what is to the right of the remove which in the best case it can remove the last key in the list so there is no need for shifting so it would only be O(logn) in that scenario. Hash Table is no question faster since it does almost no traversal. BST could be O(n) in the same case as the adding operation, but in general, it will be closer to a logn operation.

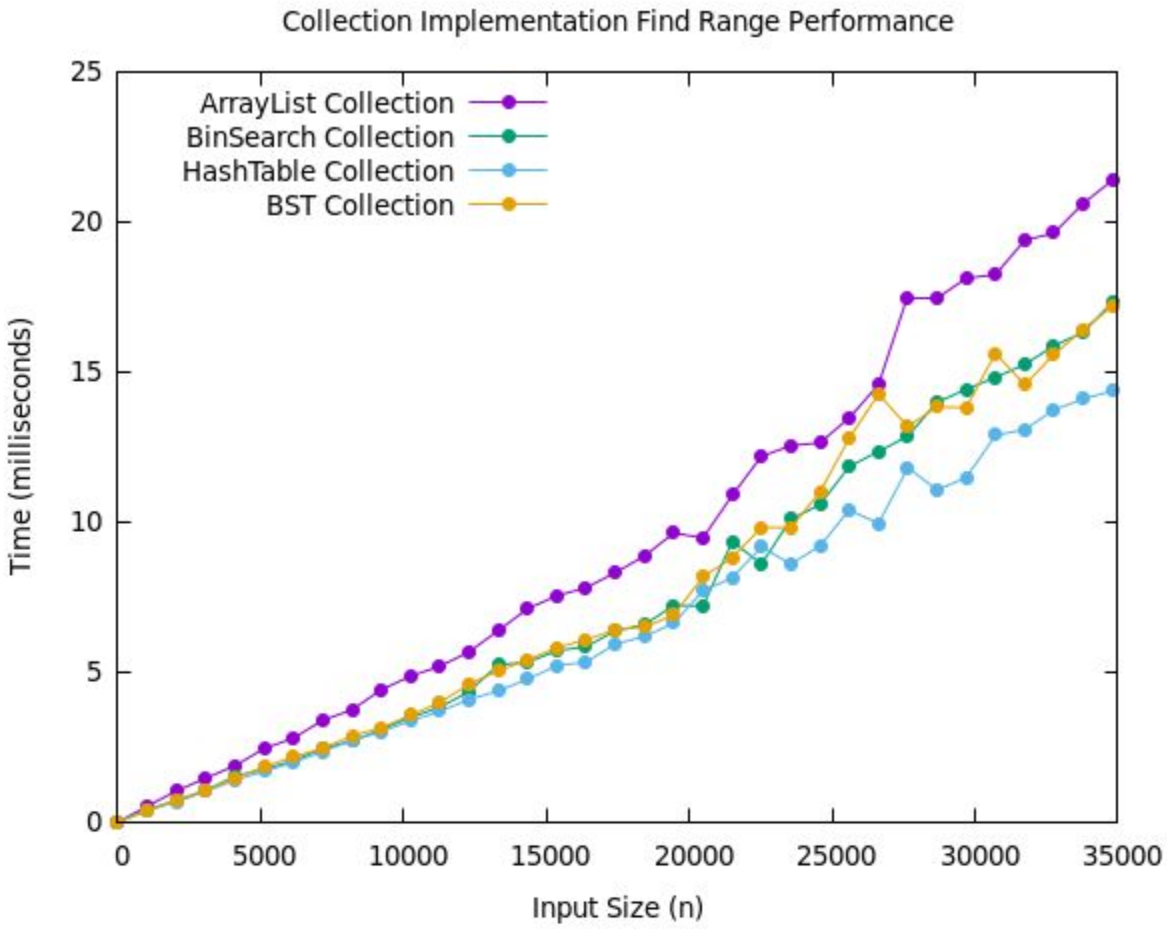
AVL - Traverses the tree recursively. Since the list's height can be no worse than 1.5 the best case height, the height cannot get out of hand like the binary search tree. This means that in the worst case, the remove function takes O(logn) which is why it performs so well in the graph.
ArrayList Collection - The collection iterates through the list and sees if the input key is in the list. This can potentially take n moves. If not, it will still take up to n moves total because the list to the right of the remove has to be shifted over.
LinkedList Collection (for personal use) - Uses the LinkedList remove function which is O(n^2) because it has to traverse the list to find and remove the item which takes n moves. But also in the loop of size n, each iteration calls get() which is also n moves which makes it worst case O(n^2).
Binary Search - Uses the binary search function to find the index of where it should be removed in the list which costs O(logn), with the index, binary search implementation calls get on the list which costs O(n) which makes this an O(n) time complexity.
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.
BST - Traverses the tree to find the item to be removed. Most of the time this will take logn operations which is why it performs so well, but in the worst case, there will be a tree which has a line of size n with no distribution which would take O(n) to remove from. This could be why there is a spike in the graph.


Collection Implementation Find Value Performance
ArrayList Collection
BinSearch Collection
HashTable Collection
BST Collection
Time (milliseconds)
Input Size (n)
0
0.5
1
1.5
2
2.5
3
0
5000
10000
15000
20000
25000
30000
35000

Collection Implementation FIND VALUE performance

Graph Hypothesis - Binary Search and Hash Table dominate in this operation and it shows on the graph. Binary Search find only has to use the binary search function which is O(logn). Hash Table only has to hash to the correct bucket which is O(1), and there is only 1 or 2 keys in that bucket. On the other hand BST performs pretty well and ArrayList does badly. ArrayList has to search iteratively which is O(n) moves which is why it performs so bad in the graph. BST is also O(n), but in most cases (when the tree is distributed well) it will take logn moves which makes it faster in general.

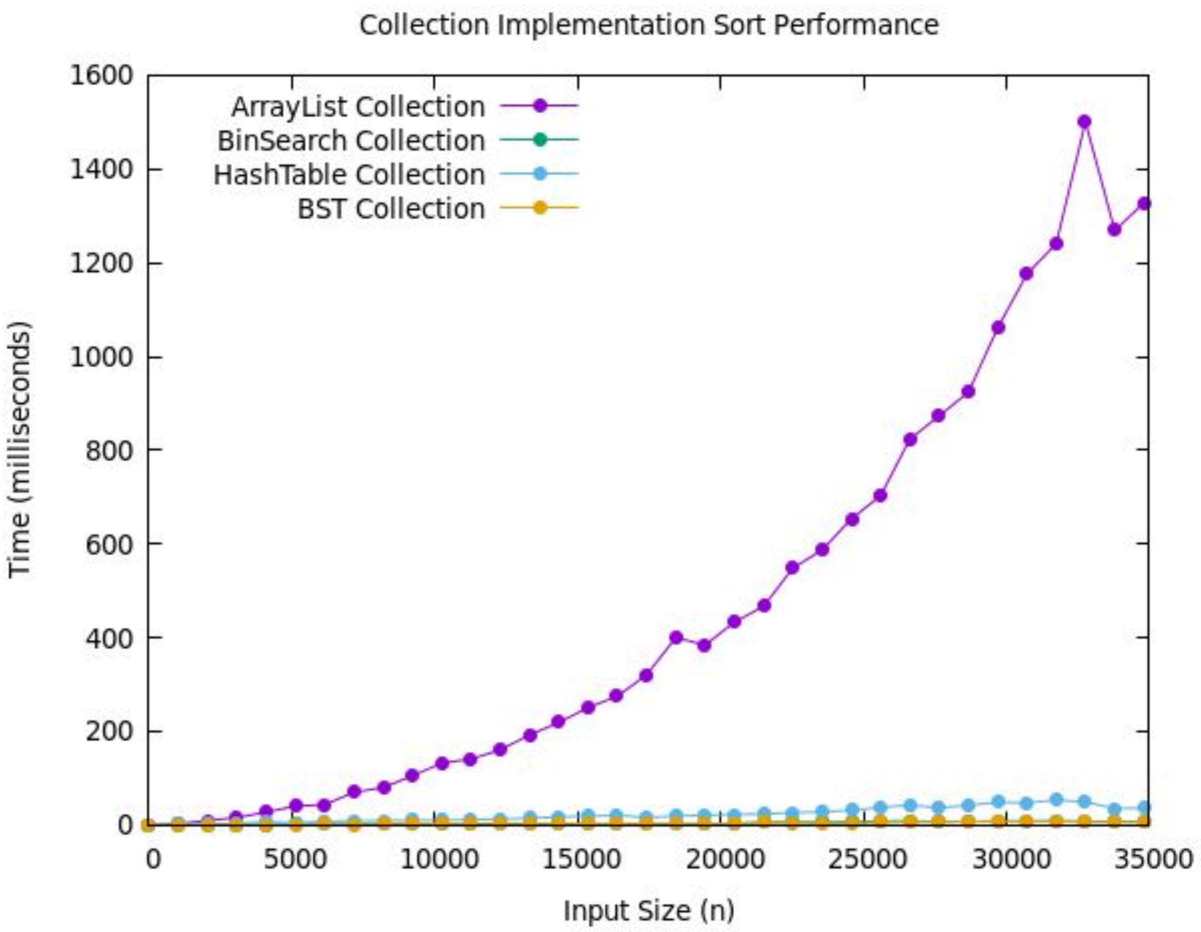
AVL - Traverses the tree recursively. Since the list's height can be no worse than 1.5 the best case height, the height cannot get out of hand like the binary search tree. This means that in the worst case, the find function takes O(logn) which is why it performs so well in the graph.
ArrayList Collection - Uses the ArrayList find value function which is O(n) because it iterates through the list until it finds the value or gets to the end of the list which is why it is classified as n operations worst case.
LinkedList Collection (for personal use) - Uses the LinkedList find value function which is O(n) because of the need to traverse the list to find the value of the key, which is why it is worst case n moves. But also has to call get which is another n moves so O(n^2)
Binary Search - Relies on the binary search function to get the index of where the key will be in the list. This binary search function is O(logn) and once the index is found, the array can go straight to that index without any traversal.
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).
BST - Traverses the tree to find the item to be removed. Most of the time this will take logn operations which is why it performs so well, but in the worst case, there will be a tree which has a line of size n with no distribution which would take O(n) to find an item at the end of the tree.


Collection Implementation Find Range Performance
ArrayList Collection
BinSearch Collection
HashTable Collection
BST Collection
Time (milliseconds)
Input Size (n)
0
5
10
15
20
25
0
5000
10000
15000
20000
25000
30000
35000

Collection Implementation FIND RANGE performance

Graph Hypothesis - This graph makes sense. Although they are all O(1), hash table search is the fastest probably because it can hash to the location of the first key which gives it a headstart. Similarly, binary search is fast because it can use the binary search function in order to find the starting key to add to the list. It also breaks the iteration when it goes out of range instead of iterating to the end of the list like ArrayList does. That is probably why the Binary Search and Hash Table do so well in this graph. BST uses recursion which does not really speed it up, so overall all of the speeds are fairly similar as seen on the graph.

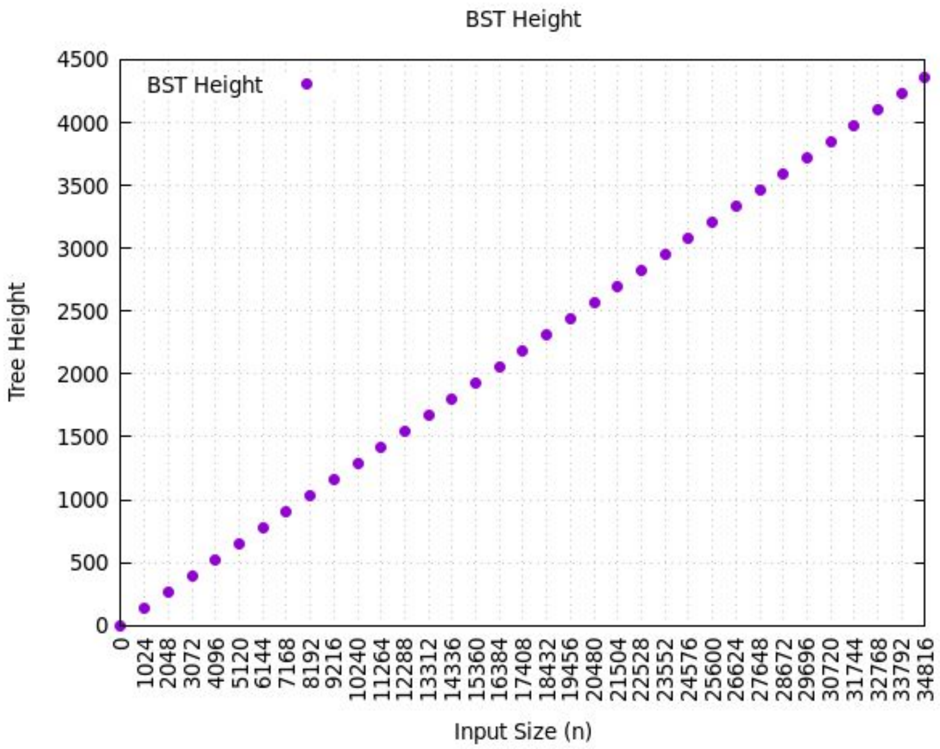
AVL - Recursively goes through the tree and finds the values of keys within a certain range. There is no extra fast way to do a find range operation because every node needs to be visited unless out of range. But regardless, if the range spans the entire tree, then each node will be included in the range which means it will be O(n).
ArrayList Collection - Iteratively goes through the list and adds keys which in the worst case will visit every index which is why it is O(n).
LinkedList Collection (for personal use) - Iteratively goes through the list and adds keys which in the worst case will visit every node which is is O(n). Within each iteration, the get() function is called which is also n operations worst case which means that this is O(n^2).
Binary Search - The binary search function is only used to find the index of the first key which makes it a little faster, but it cannot be used in the rest of this operation which means that the find range function defers to the same one as the ArrayList find range. This 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).
BST - 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).


Collection Implementation Sort Performance
ArrayList Collection
BinSearch Collection
HashTable Collection
BST Collection
Time (milliseconds)
Input Size (n)
0
200
400
600
800
1000
1200
1400
1600
0
5000
10000
15000
20000
25000
30000
35000

Collection Implementation SORT performance

Graph Hypothesis - Once again, Arraylist struggles. Arraylist relies on Quick Sort which is worst case O(n^2). As for Binary Search, they only have to call keys which is O(n) which is significantly faster and is why it does so well on the graph. Spikes could be the very worst case for ArrayList. 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 has no performance problems since it is already sorted and only has to call keys which returns the list.

AVL - 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).
ArrayList Collection - Uses quicksort from ArrayList which with worst case will be O(n^2).
LinkedList Collection (for personal use) - Calls keys which loops through the list which costs n at most, and within each iteration of the loop, get() is called which makes it O(n^2). Since the sorting algorithm called is not any slower than O(n^2), the operation is worst case O(n^2). Even though it uses Merge sort which is worst case nlogn, the fetching part of it in the collection is slower which overrules it.
Binary Search - This function calls keys which iterates through the list. This can cost n moves at most and the list is already sorted so O(n).
Hash Table - Same as ArrayList. Calls keys and defers to quicksort which is O(n^2). But performs better than ArrayList because buckets are somewhat sorted already.
BST - Recursively goes through the tree and finds all of keys. There is no extra fast way to do a do this operation because every node needs to be visited unless out of range. But regardless, it will be O(n).


BST Height
BST Height ●
Tree Height
Input Size (n)
0
500
1000
1500
2000
2500
3000
3500
4000
4500
0
1024
2048
3072
4096
5120
6144
7168
8192
9216
10240
11264
12288
13312
14336
15360
16384
17408
18432
19456
20480
21504
22528
23552
24576
25600
26624
27648
28672
29696
30720
31744
32768
33792
34816

Collection Implementation SIZE performance

Graph Hypothesis - This graph shows the problem with BST that AVL and RBT fix. BST has no height regulations so it can easily turn into a linked list which defeats the purpose of the tree. For example, if you add keys 1-1000 in this order, every key added will go to the very right child's right because they are always bigger than the previous which is why the height in the graph grows constantly.