

Binary Search

CPSC 223 - HW5 - Wesley Muehlhausen

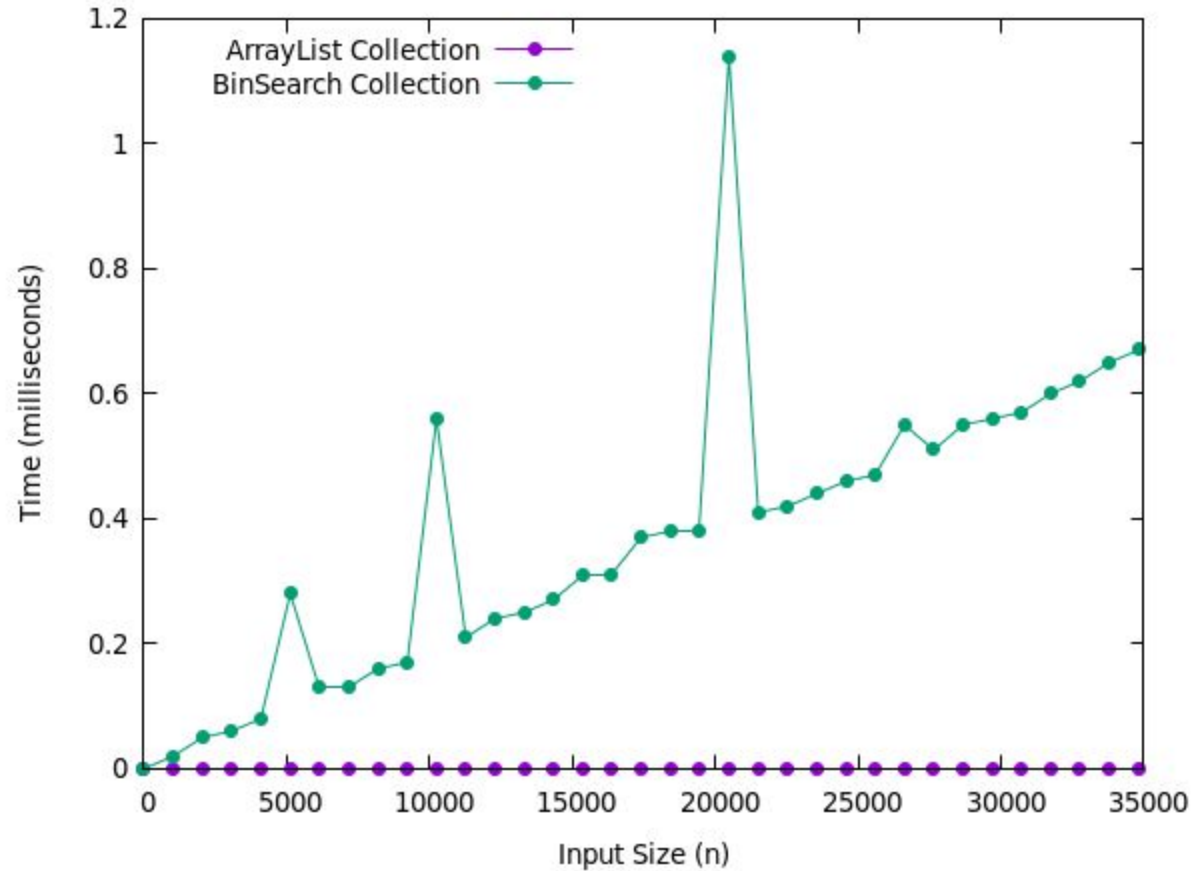
Challenges and Issues

- This assignment went very smoothly for me in terms of setting everything up. Most of the problems came in the binary search class add function. I had to figure out the condition where for example: the list contains an element A and my binary search function (which worked) would tell the add function (Adding B) to add it before the A (at index 0) which is not correct. So I fixed that with an extra case at the end of my binary search function. But that ended up slowing everything down which made the performance tests go super slow, so I had to make it work in my add function
- Another problem I had was coming up with tests. I ended up making tests that would try to exploit the potential flaws such as sorting an empty list or removing from an empty list.
- Finally, my merge sort was messed up because I was out of bounds on my index which caused problems because my index was unsigned.

Collection Implementation

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

ArrayList vs. Binary Search ADD performance



Collection Implementation ADD performance

Graph Hypothesis - Looking at the graph all of the operations did well. This graph makes sense though because ArrayList 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 Binary Search are because of the resizing. What is odd is that ArrayList doesn't show any resizing spikes. This could be because of the scale of the graph.

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 $O(\log n)$ 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.

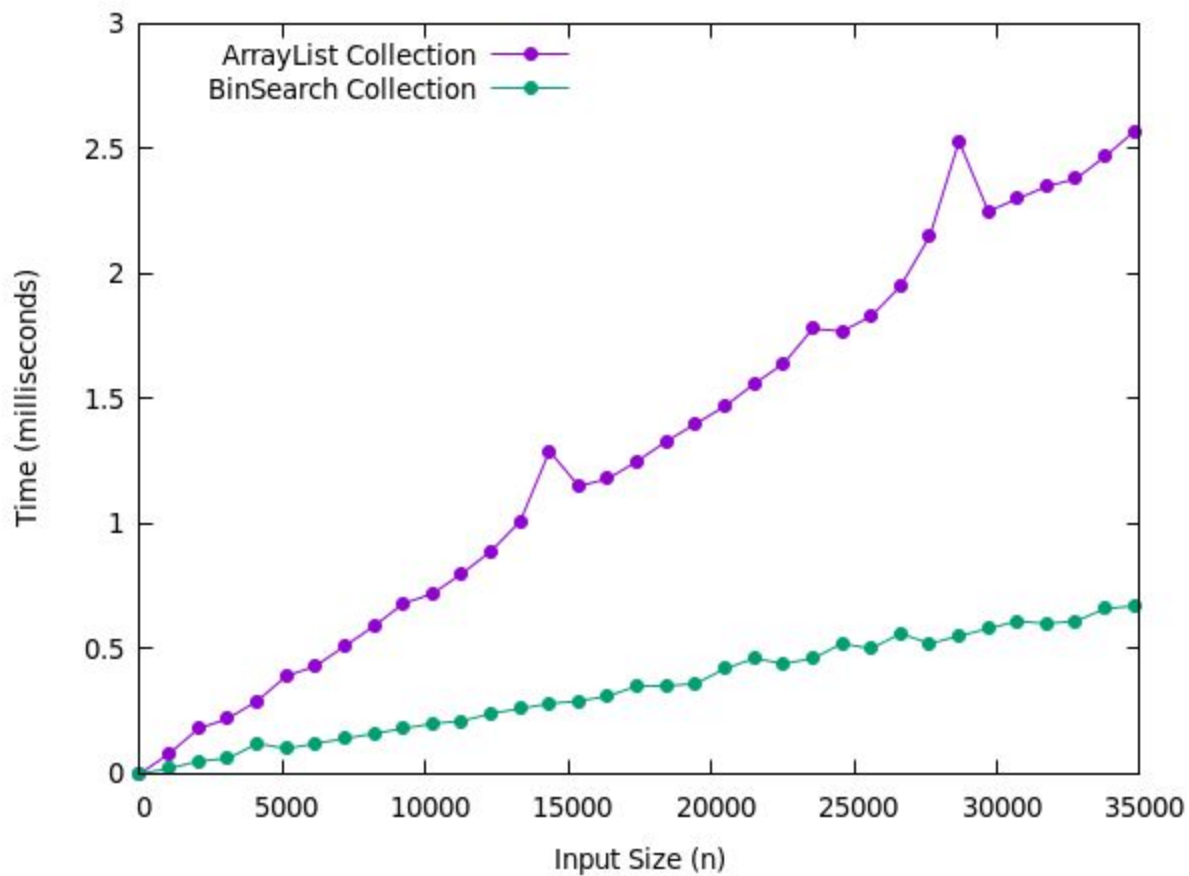
LinkedList Collection (for personal use) - Uses the LinkedList add function which is $O(1)$ because it adds to the end of the list using the tail pointer. This means no traversal is necessary which is why it is not n moves.

Binary Search - Uses the binary search function to find the index of where it should be added in the list which costs $O(\log n)$, 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 $O(\log n)$

ArrayList vs. Binary Search REMOVE performance



Collection Implementation REMOVE performance

Graph Hypothesis - This graph makes sense. ArrayList performs worse than Binary Search even though they are both $O(n)$ operations. 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 $O(\log n)$ 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 $O(\log n)$ in that scenario.

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 $O(\log n)$ 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.

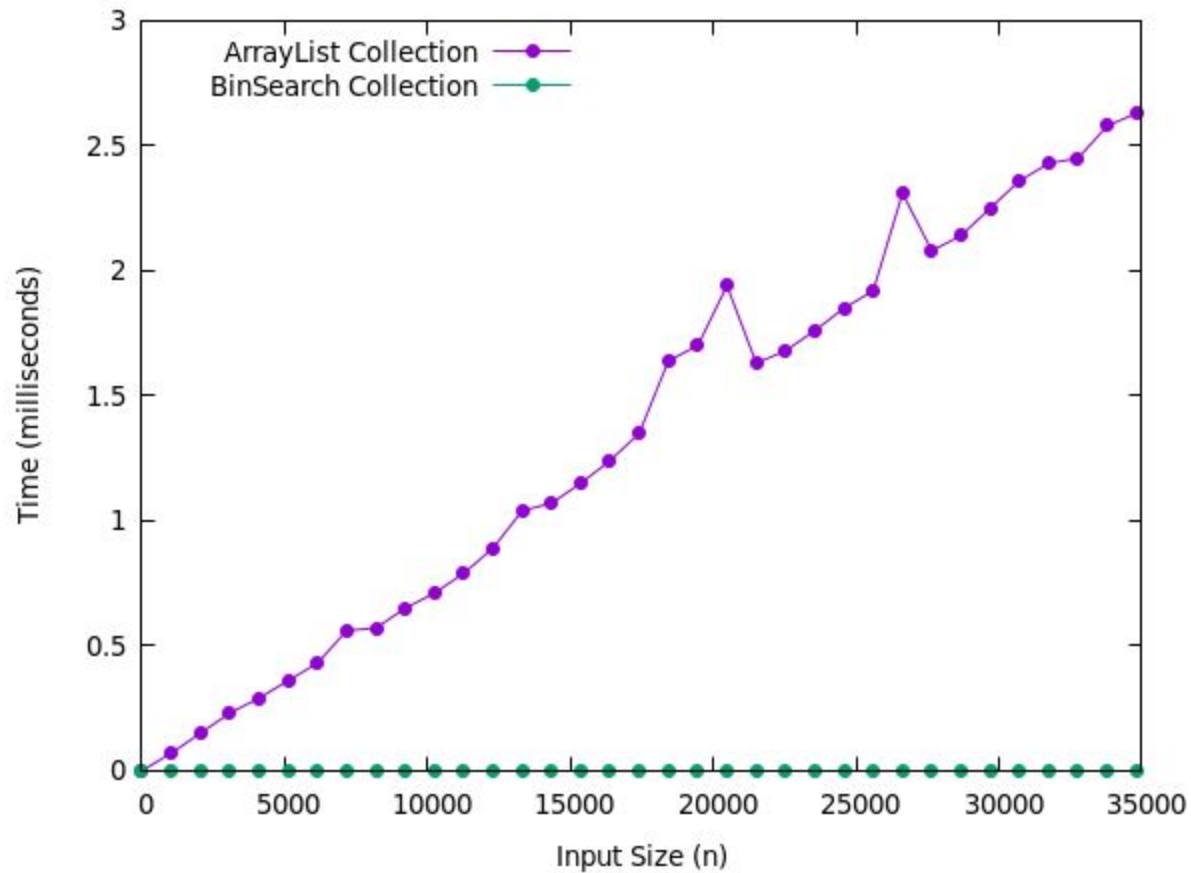
LinkedList Collection (for personal use) - Uses the LinkedList remove function which is $O(n^2)$ because it has to traverse the list to find and remove the item which takes n moves. But also in the loop of size n , each iteration calls `get()` which is also n moves which makes it worst case $O(n^2)$.

Binary Search - Uses the binary search function to find the index of where it should be removed in the list which costs $O(\log n)$, 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 $\log n$ 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.

ArrayList vs. Binary Search FIND VALUE performance



Collection Implementation FIND VALUE performance

Graph Hypothesis - Binary Search dominates in this operation and it shows on the graph. Binary Search find only has to use the binary search function which is $O(\log n)$. On the other hand, ArrayList has to search iteratively which is $O(n)$ moves which is why it performs so bad in the graph.

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 $O(\log n)$ 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.

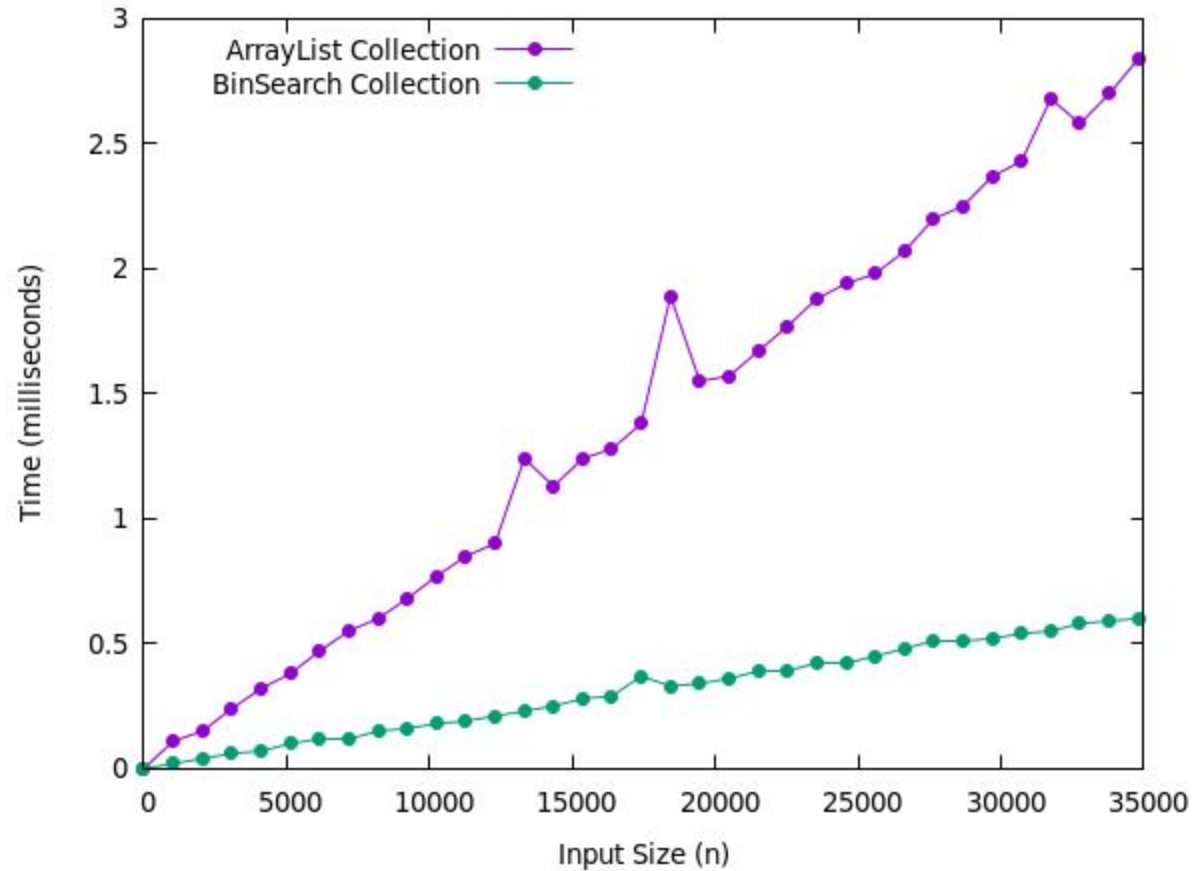
LinkedList Collection (for personal use) - Uses the LinkedList find value function which is $O(n)$ because of the need to traverse the list to find the value of the key, which is why it is worst case n moves. But also has to call get which is another n moves so $O(n^2)$

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 $O(\log n)$ 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 $\log n$ 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.

ArrayList vs. Binary Search FIND RANGE performance



Collection Implementation FIND RANGE performance

Graph Hypothesis - This graph makes sense. Although they are both $O(1)$, binary search is faster 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 does so well in this 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 $O(n)$.

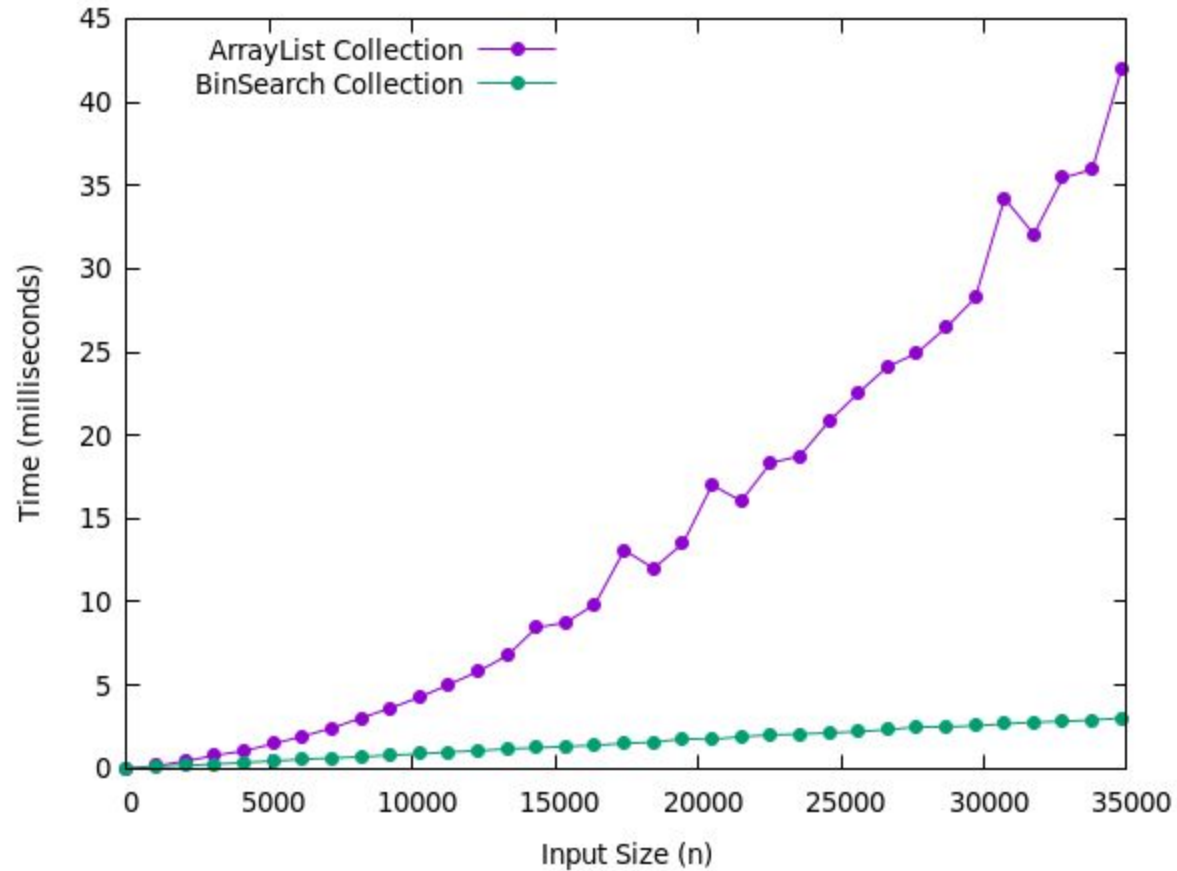
LinkedList Collection (for personal use) - Iteratively goes through the list and adds keys which in the worst case will visit every node which is $O(n)$. Within each iteration, the `get()` function is called which is also n operations worst case which means that this is $O(n^2)$.

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 go 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 $O(n)$.

ArrayList vs. Binary Search SORT performance



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.

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 $n \log n$, 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 $O(n^2)$.

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 $O(n)$.