HW4 - KV Pairs

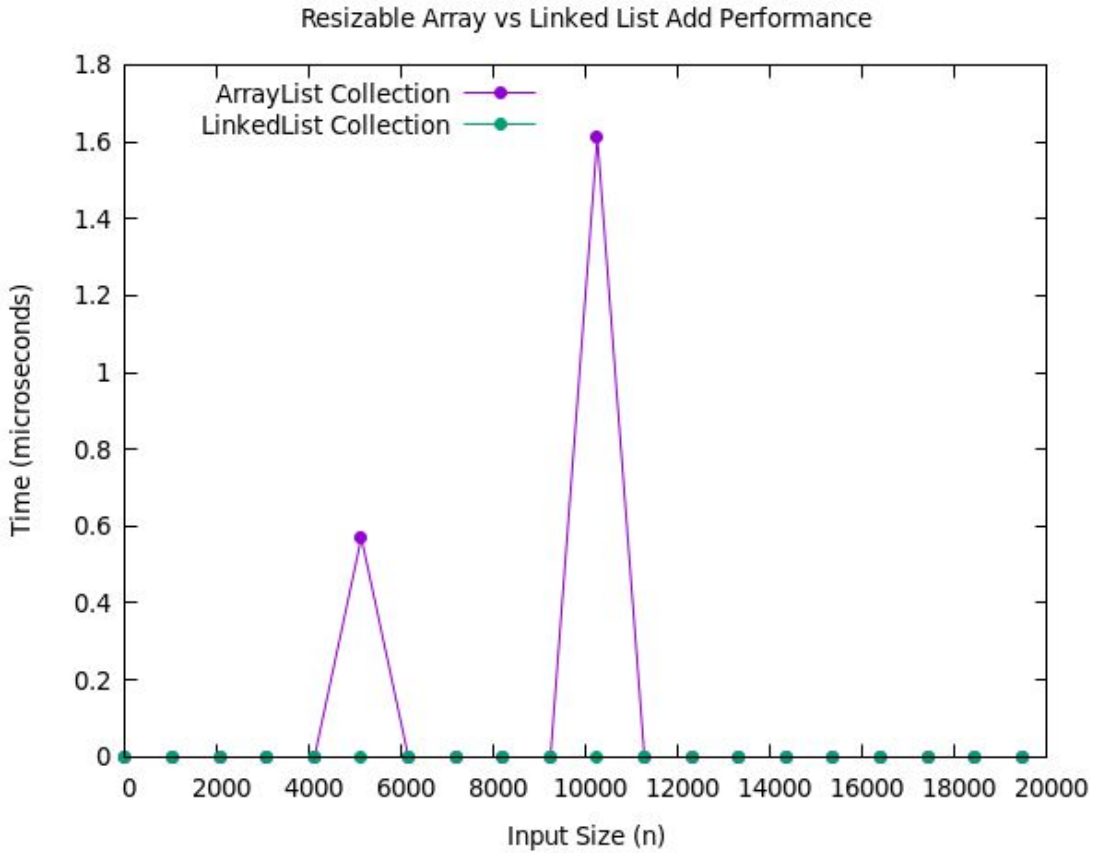
CPSC 223 - Wesley Muehlhausen

Collection Implementation

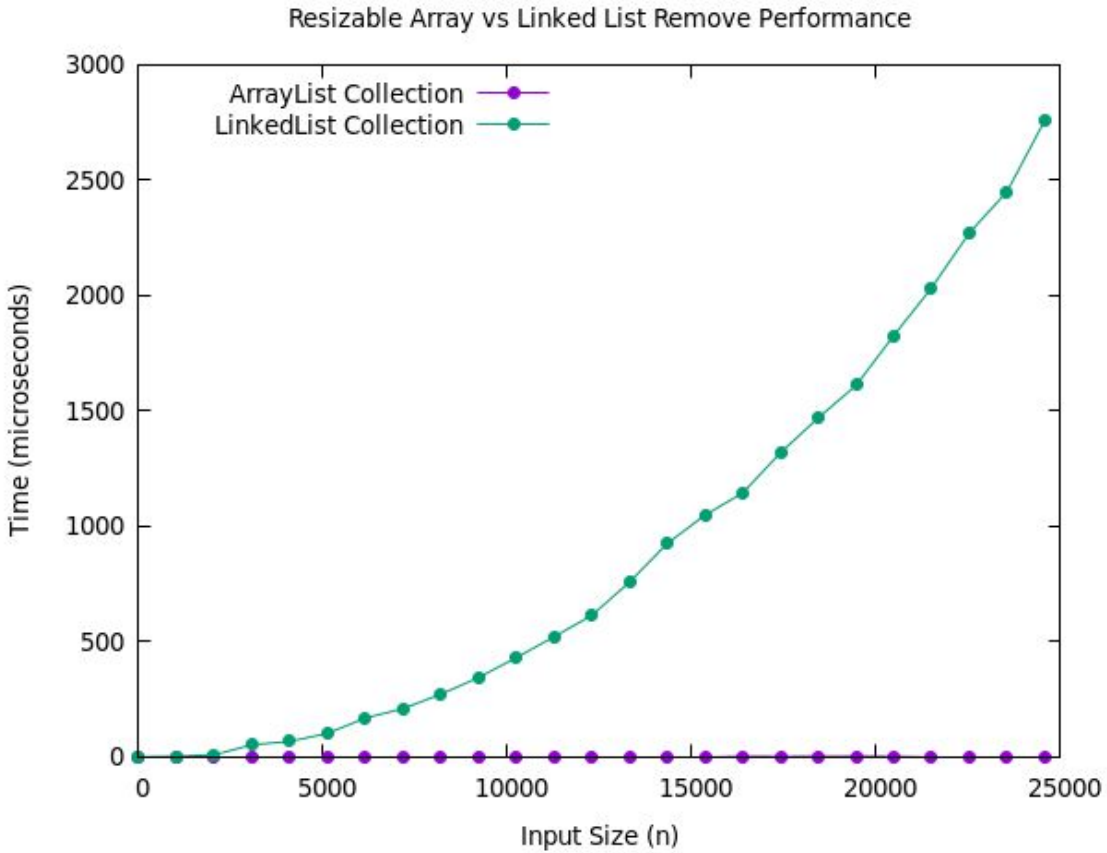
Operation	ArrayList C.	LinkedList C.	Sorted Array (Bin Search)	Hash Table	BST	AVL
Add	O(1)	O(1)	O(n)	O(1)	O(n)	O(logn)
Remove	O(n)	$O(n^2)$	O(n)	O(1)	O(n)	O(logn)
Find-value	O(n)	$O(n^2)$	O(logn)	O(1)	O(n)	O(logn)
Find-range	O(n)	$O(n^2)$	O(n)	O(n)	O(n)	O(n)
Sort	$O(n^2)$ Quick Sort	$O(n^2)$ See explanation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

Issues and Challenges

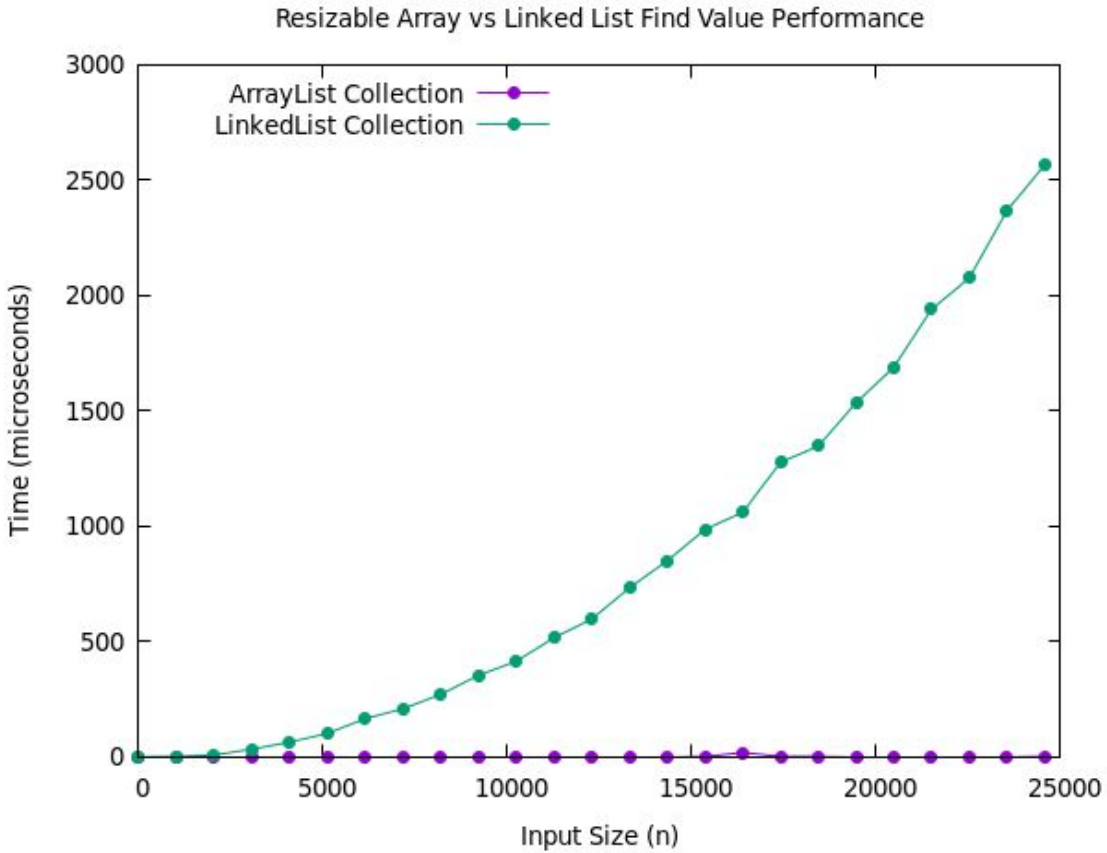
- My huge problem in this project was just setting it up. I was a little confused on how to set up the functions and also using inheritance for using functions from ArrayList.h and LinkedList.h. That lead to huge problems which made it so I couldn't even get it to compile to start with.
- Another problem or stem of confusion was with using an ArrayList in the LinkedList Collection implementation. It didn't make sense, but after working with it in multiple projects it makes perfect sense. Also, creating a pair was something new and was overall confusing which halted my progression.
- Creating 10 tests was a lot for me. Making the tests wasn't hard, it was just thinking of them. I ended up making tests for specific conditions like lists of size 2 and 3, because after that there are less specific conditions to worry about.
- Looking back on it, this homework was super easy, it just didn't seem like it because I got a little lost.



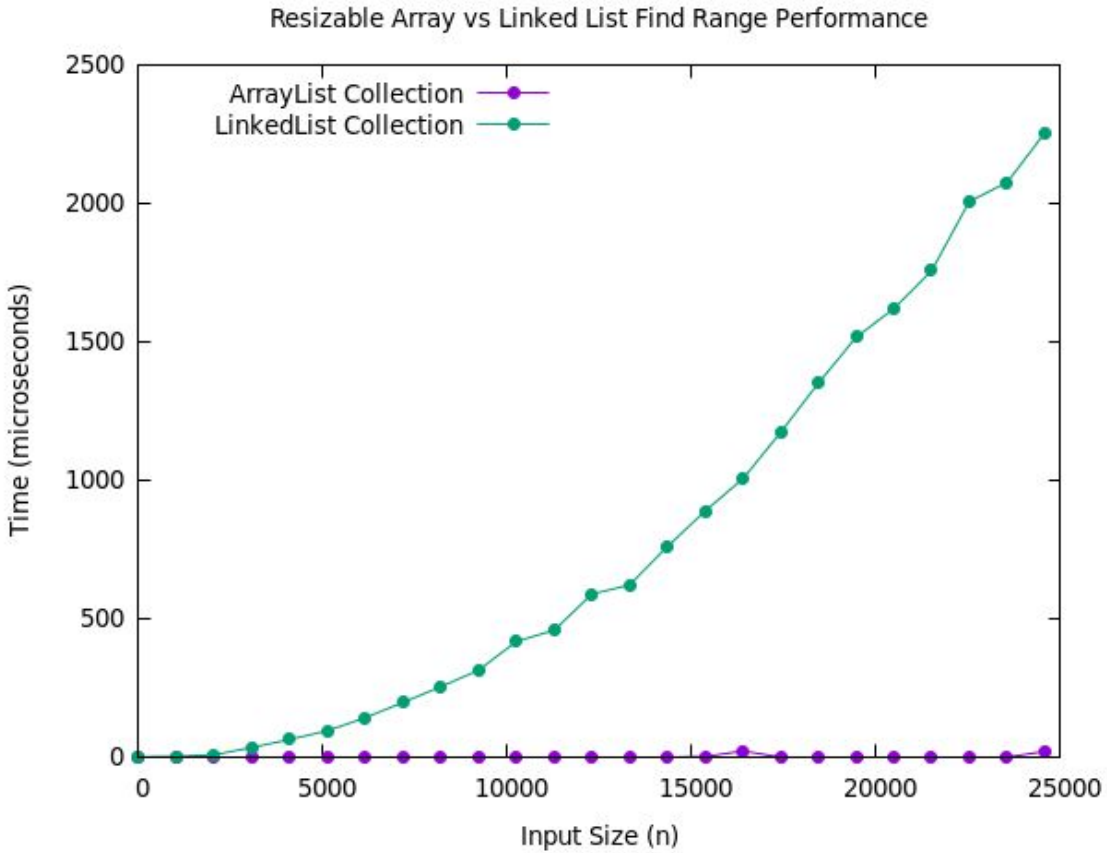
-This graph makes sense. Both of the add implementations are O(1). In this case though, LinkedList is faster than ArrayList because ArrayList still has to resize, this is why we see the spikes in the graph. Both overall do this very fast.



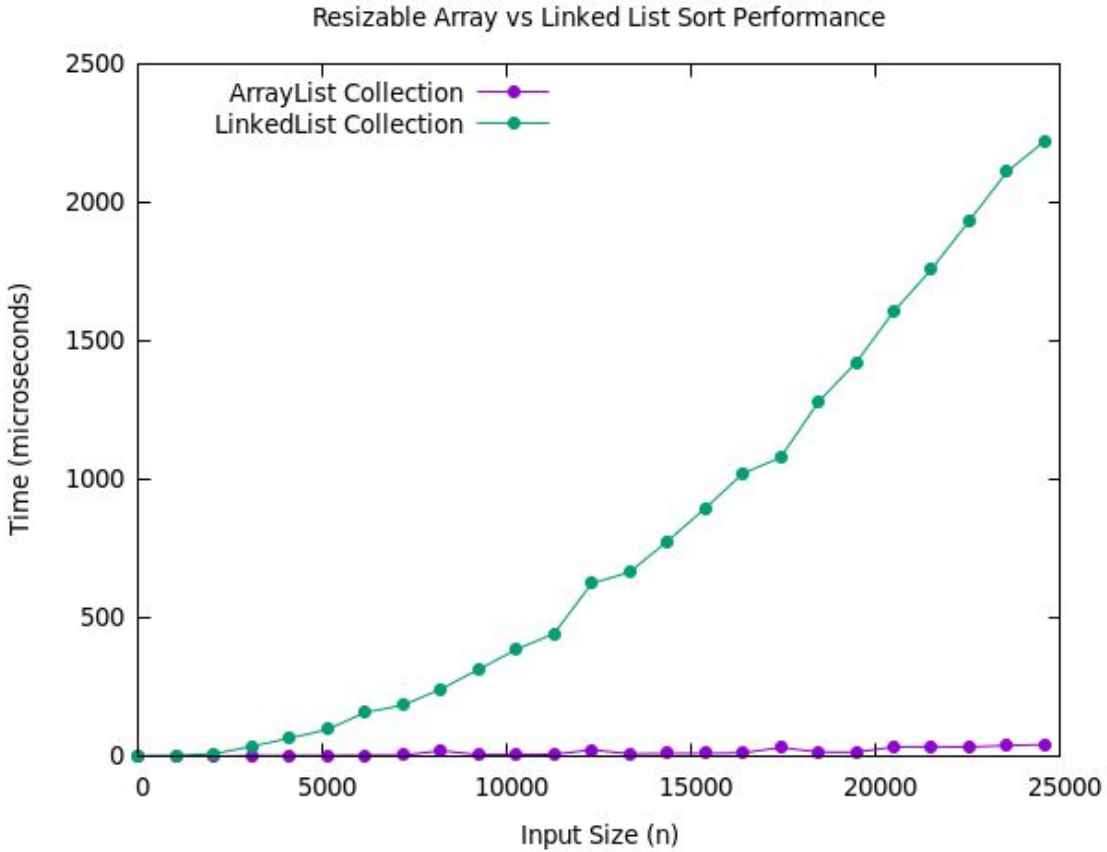
-This also makes sense. ArrayList is going to be faster than LinkedList in this operation. ArrayList Collection is O(n) because in the any case, you have to traverse to the item and remove it, and then after that, you have to shift over the other side of the list which in total is n moves. That is still better than LinkedList Collection which if removing the last item has to traverse to the end of the list which costs n, and within each iteration, the LinkedList we use uses get() which also can take n moves which is worst case O(n^2). So the ArrayList collection is a lot better and it shows.



-The find value is very similar in performance to remove. The ArrayList and linked list both have to search for the item iteratively which takes n moves. But LinkedList performs worse because get() for the linkedlist is also n moves and is called for every iteration in find. This is again why the graph shows LinkedList being so slow.



-The find range is exactly the same performance as the find value. It iterates through the list for both implementations which takes n moves. What makes LinkedList so slow is that the get() method is called within each iteration which makes it O(n^2).



-The ArrayList Collection defers to quick sort while the LinkedList defers to merge sort. Both of the time complexities should be the same. I think the reason the graph is the way it is, is because quicksort only is O(n^2) in very worst case so it is not always like that. On the other hand, LinkedList uses merge sort which is O(nlogn). But the collection calls keys and then sort. Keys is always O(n^2) because it iterates through the list calling get() on every iteration. This is why I think LinkedList performs so bad on the graph.