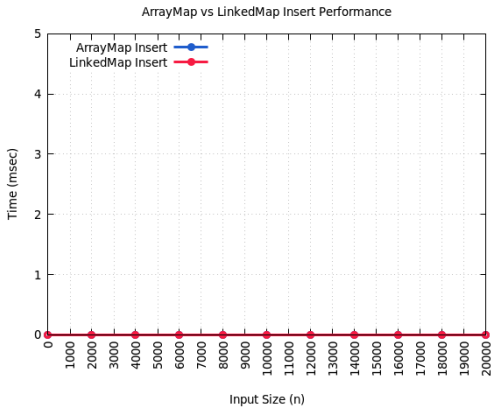
-- CPSC 223 Report--

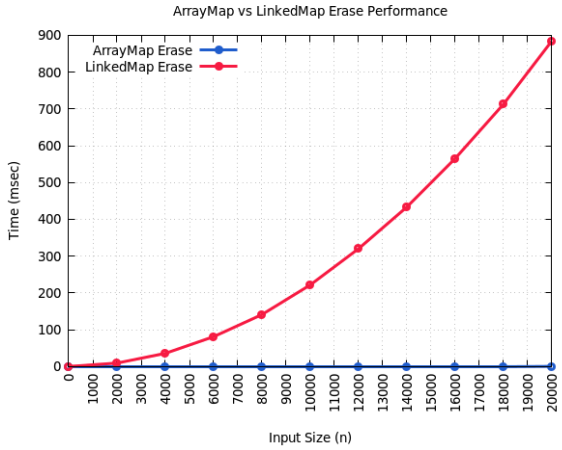
By Samuel Sovi

Over the course of this semester, we covered many different data structures, each with their own advantages and disadvantages. Utilizing the structure of a Map (having a key value pair), we used many different data structures to compare the efficiency in making a Map perform the operations we needed them to do such as 1. Inserting key-value pairs, 2. Erasing key-value pairs, 3. Checking if keys were contained within the map, 4. Finding all keys within a given range and 5. Returning all keys in sorted order. Each of our key-value pair Map implementations contains these five main operations and each have differing trade offs.

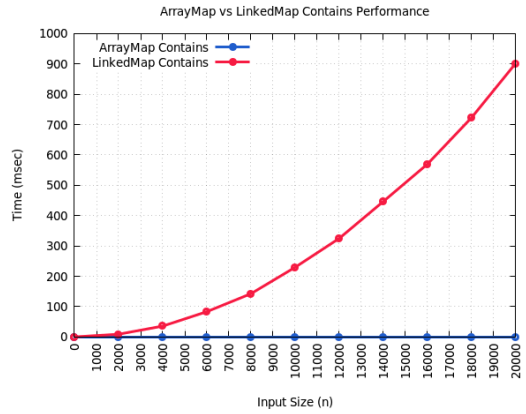
The first two key-value pair Maps we implemented were LinkedMap and ArrayMap. Our LinkedMap used a Linked List to store the key-value pairs while our ArrayMap used a resizable array to store them. .



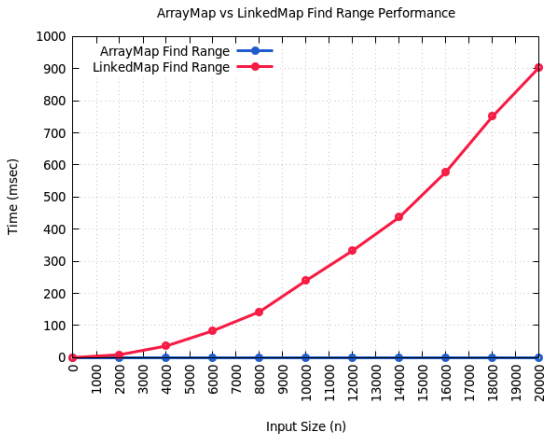
For insertion, both our LinkedMap and ArrayMap implementations performed their function with a worst case O(1). This is because we could insert at the end of our array with an amortized cost of O(1) for our ArrayMap whereas our tail pointer for our LinkedSeq in our LinkedMap allowed us to insert at the end of our LinkedMap.



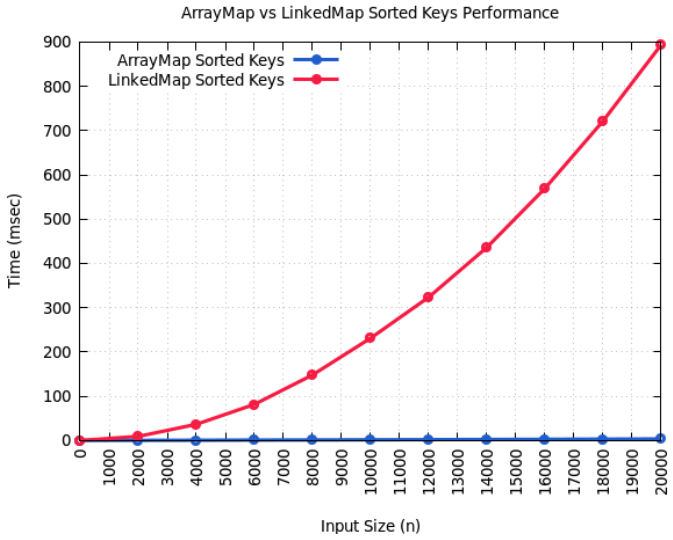
When it came to erase, however, our LinkedMap had an O(n^2) while ArrayMap had a worst case O(n). This is because erasing from both implementations using a key requires nodes to be visited sequentially and LinkedSeqs must visit every previous node to get to the next node whereas ArraySeqs just need to shift elements after erasing.



Contains had the same logic as erase, with LinkedSeqs needing to go to every previous node to move to the next node whereas ArraySeqs needed to go through each element sequentially to check if it matched the key that was passed in.

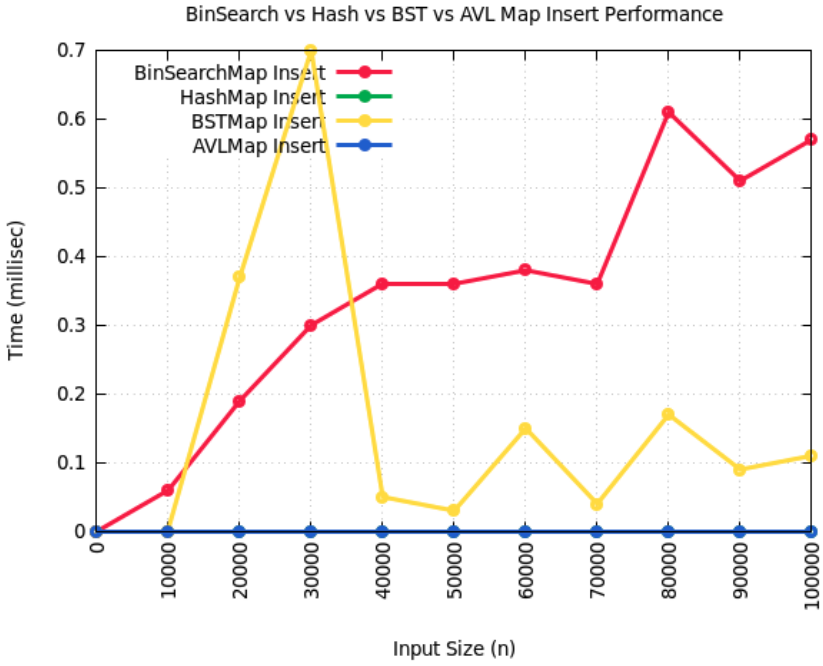


For finding all keys in a given range, we must iterate through our entire data structure finding all keys that match the given range since both structures are not sorted. As a result, we once again get O(n^2) for LinkedMap and O(n) for ArrayMap



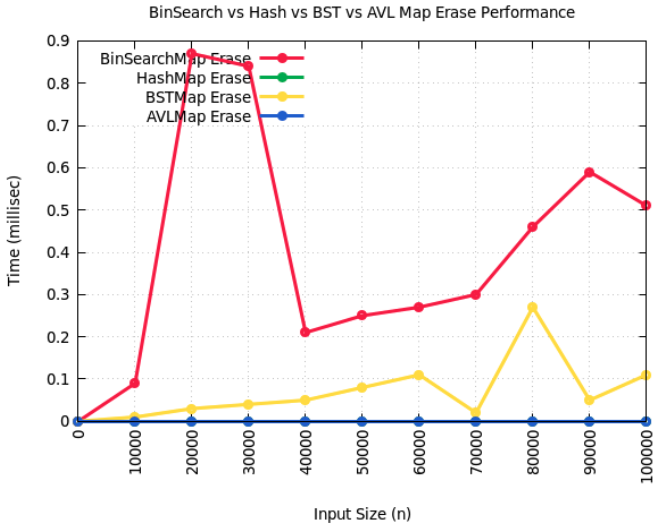
While both curves look different, both ArrayMap and LinkedMap have a worst case of O(n^2) for sorted keys. LinkedMap has O(n^2) because of the time taken to iterate through the LinkedList to collect all keys while ArrayMap has O(n^2) since it utilizes quick_sort_random() as its sorting algorithm. The graphed curves look much different though since quick_sort_random() is much closer to O(nlogn) in practice and only very rarely hits the O(n^2) case.

Soon after implementing these two Maps in our first Map-related assignment, we started to work with other more efficient data structures for our Maps. These included AVL Trees for AVLMap, Binary Search Trees for BSTMap, a Hash Table for HashMap and a Binary Search ArraySeq for BinSearchMap. For inserting into these structures, we have a worst case of O(n) for BinSearchMap, O(1) for HashMap, O(n) for BSTMap and O(logn) for AVLMap. A performance graph is shown below:



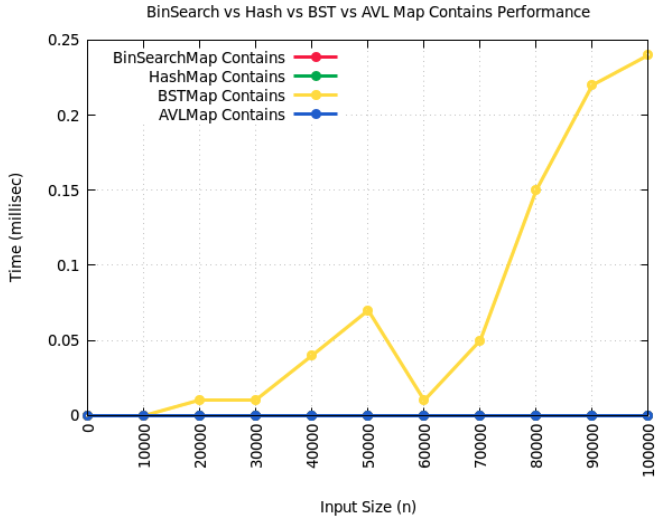
BinSearchMap has a cost of O(n) since inserting to the beginning of the ArraySeq would require the data structure to shift all of its elements which is O(n). HashMap is O(1) since our Hashing Function should be decreasing the amount of keys per bucket and resizing when necessary. BSTMap is O(n) for inserting because in the case where every element is input is strictly increasing or strictly decreasing which makes the BSTMap effectively a linked list. AVLMap has a worst case of O(logn) for inserting since the tree is balanced and only takes the Tree logn steps to get from the root to any given node for inserting.

Erasing from these data structures, we have a worst case of O(n) for BinSearchMap, O(1) for HashMap, O(n) for BSTMap and O(logn) for AVLMap. A performance graph is shown below:



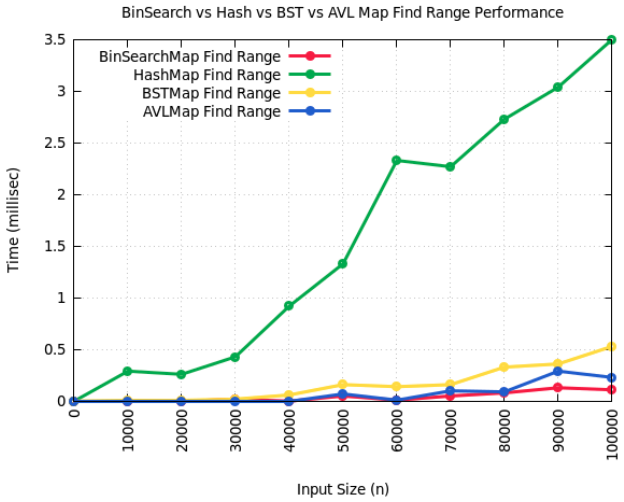
BinSearchMap has a cost of O(n) because erasing from the beginning would require the data structure to shift all elements left which is O(n). HashMap is O(1) for a similar reason to insertion where the HashFunction directs us to the exact bucket we need to search in. BSTMap is O(n) since strictly increasing or strictly decreasing element insertions would make the BSTMap have to go through every single element to get to the last Node. AVLMap is O(logn) for erase as well for the same balancing reason as insert.

For finding if a given key is contained within the structure, we have a worst case of O(logn) for BinSearchMap, O(1) for HashMap, O(n) for BSTMap and O(logn) for AVLMap. A performance graph is shown below:



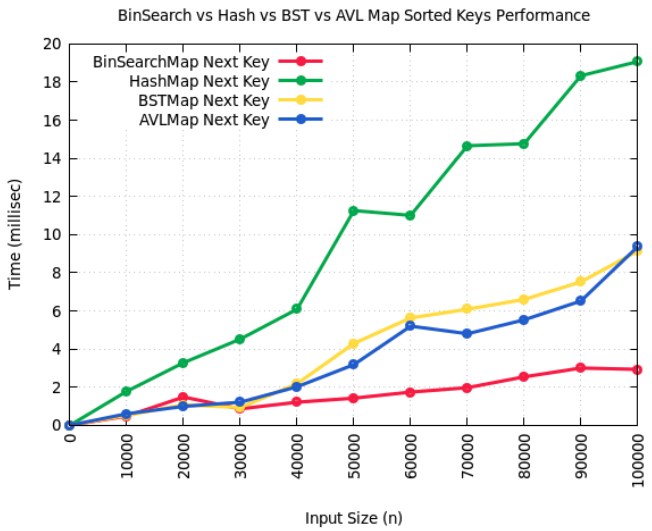
BinSearchMap has a cost of O(logn) because we just need to perform a binary search to find the element. HashMap is O(1) for a similar reason to insertion and erasing where the Hashing Function directs us to the exact bucket we need to search in. BSTMap is O(n) since strictly increasing or strictly decreasing element insertions would make the BSTMap have to go through every single element to get to the last Node. AVLMap is O(logn) for contains as well for the same balancing reason as insert and erase.

For finding all keys within a given range, we have a worst case of O(n) for all 5 of these data structures' Map implementations. A performance graph is shown below:



BinSearchMap has a worst case of O(n) since we could need to check every key in the ArraySeq. HashMap requires us to check every bucket because the hashing function does not tell us enough about the key's value. BSTMap and AVLMap can also require us to visit every Node if all the keys are within range.

Finally, for finding if a given key is contained within the structure, we have a worst case of O(n) for BinSearchMap, BSTMap and AVLMap while HashMap has a worst case of O(n^2). A performance graph is shown below:



For BinSearchMap, BSTMap and AVLMap, since the keys are stored in a sorted fashion, we can just enter all the keys in order into an ArraySeq and return it. For HashMap, we have to get all the keys, then call sort() on the ArraySeq which has a worst case of O(n^2) if all of the randomized pivots are in ascending or descending or alternating ascending/descending order.

The following table can be used to summarize our findings

	LinkedMap	ArrayMap	BinSearchMap	HashMap	BSTMap	AVLMap
Insert	O(1)	O(1)	O(n)	O(1)	O(n)	O(logn)
Erase	O(n^2)	O(n)	O(n)	O(1)	O(n)	O(logn)
Contains	O(n^2)	O(n)	O(logn)	O(1)	O(n)	O(logn)
Find Keys	O(n^2)	O(n)	O(n)	O(n)	O(n)	O(n)
Sorted Keys	O(n^2)	O(n^2) (quicksort)	O(n)	O(n^2) (quicksort)	O(n)	O(n)

For the next part of the report, I will explain my idea of the best implementation of a Map for each function listed below. For each instance, I will assume that the Map is already bulk loaded (as instructed).

(a). An application where the two main operations are insert and contains, which are performed at approximately the same frequency.

I believe that the best Map implementation of the ones we have done so far for this purpose would be a HashMap. The HashMap is perfect for both insert() and contains() since it is O(1) for both operations. This beats out the other Map implementations by quite a bit as shown in the earlier table since none of the other implementations can match HashMap contains() cost. Since the hashmap is already bulk loaded, there will also be less reszie_and_rehash() function calls needed. However, it is worth noting that HashMap is not very memory-friendly since it takes up a lot more space than the size of elements in the HashMap.

(b). An application where the two main operations are insert and find keys, but where the keys need to be returned in sorted order. Assume find keys is performed much more frequently than insert. Since some map implementations don't return the results of a range search in sorted order, an additional step may be needed to do the sort (which could potentially still lead to more efficient overall operation cost).

I believe that the best Map implementation of the ones we have done so far for this purpose would be an AVLMap. This is due to the balanced nature of the AVLMap which beats out the similar competitors of BSTMap and BinSearchMap since its worst case of insert is O(logn). All three of these options are the only implementations listed above that will return the found keys in sorted order and they all have the same find_keys() worst case which is O(n). BSTMap performs much worse when elements are inserted in ascending or descending order which increases insert()'s cost to O(n) whereas BinSearchMap insert()'s cost is O(n) due to

the need to shift up to n-1 elements when inserting at the beginning. As such, the AVLMap is best for this scenario since it negates the need for any of these cost increases with the only drawback being the balancing.

(c). An application where a frequent number of insert and erase operations are performed such that a series of inserts are done, followed by a series of erases, and so on. Assume that the overall size of the map ends up growing slowly (with slightly more insert than erase calls over time).

Once again, I believe that the HashMap is the best Map implementation of the ones above for this scenario since it has an insert cost of O(1) as well as an erase cost of O(1). Since the overall size of the Map is growing slowly and the Map is also already bulk loaded (as per assumption in instructions), there will be less of a need for calling resize_and_rehash(), further increasing the efficiency. None of the other Map implementations listed above can beat the Erase cost of the HashMap while only some can even match the insert cost. However, it is worth noting once again that HashMap is not very memory-friendly since it takes up a lot more space than the size of elements in the HashMap.