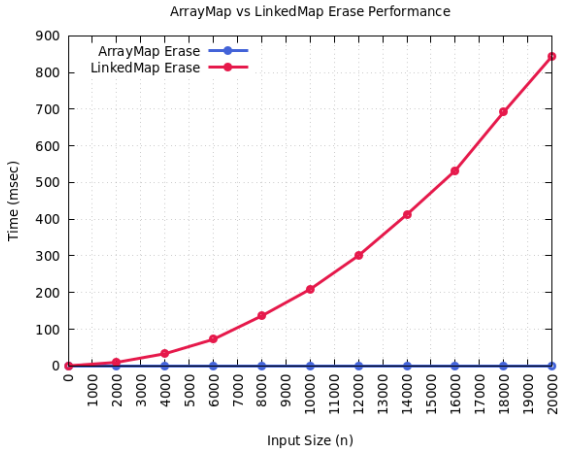
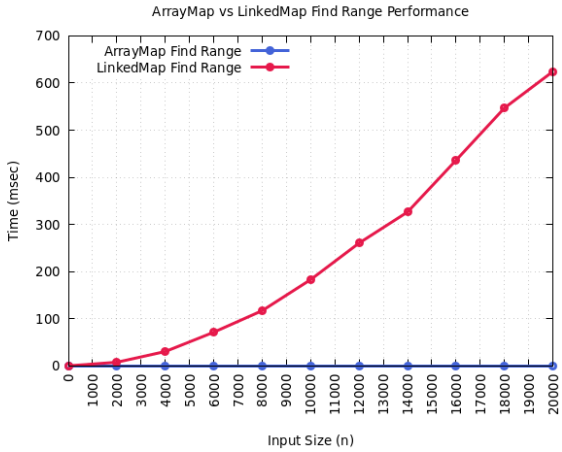
Final Report

Comparison of Performance and Analytical Results

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

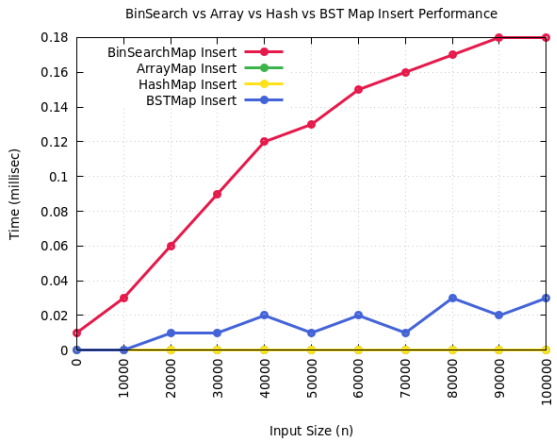
To start off, LinkedMap is not the ideal in many of the main functions of the Map interface. Even with a near-identical implementation to ArrayMap, the simple swap from resizable array to Linked List (sequence) as the base list type used made the time complexity increase dramatically. While inserting, searching, and sorting remain the same at O(1), O(n), and O(n^2), both erasing and finding a range of keys see performance improvements when using ArrayMap. The improvement to erasing can be attributed to a need to traverse a linked list to get to the desired key rather than accessing the key at any given index in constant time. This effect causes time complexity to be reduced from O(n^2) to O(n) as shown in the graph. Similarly, the improvement to find range is also due to direct access at indices rather than traversing a linked list and it has the same effect.

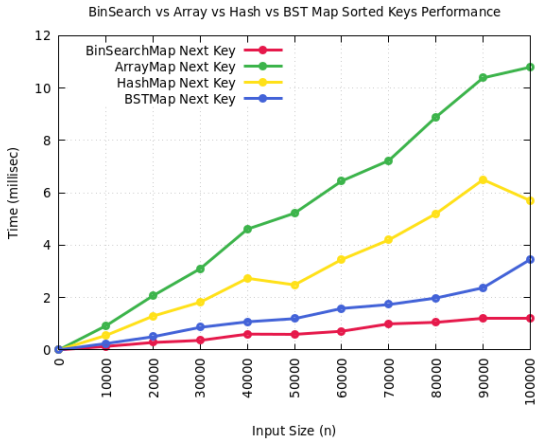




The next pairing of similar methods would be those that utilize the concept of Searching by Binaries, BinSearchMap and BSTMap. The main difference between the previous two implementations and these is how much of the data structure is covered to perform an operation. See, the binary search method compares a search key with the key in the location where a traversal currently sits. Doing so, the algorithm reduces the amount of area to cover when searching for keys or sufficient locations for inserting. Funny enough, BinSearchMap maintains a sorted state over the list for the entire time it is used. BSTMap does something similar in checking for larger or smaller keys. However, BSTMap takes on the tree structure rather than a list so searching, in general, appears to be more complex than in BinSearchMap. However, BinSearchMap, as shown in the graphs below and the table above, takes a hit in insert going

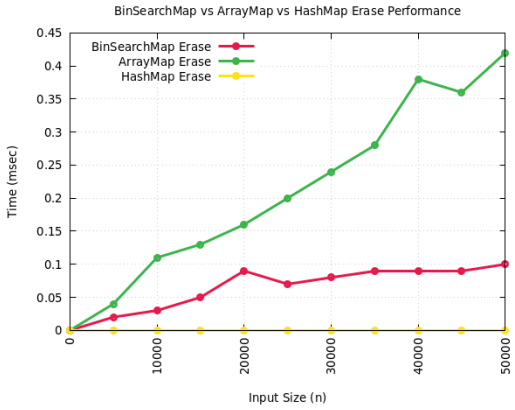
from O(1) in ArrayMap or O(logn) in BSTMap to O(n). That said, sorting is the best it will be with BinSearchMap where the time complexity is O(logn) as compared to the usual O(n^2).

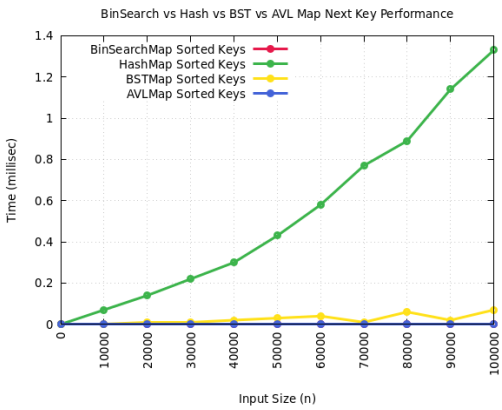




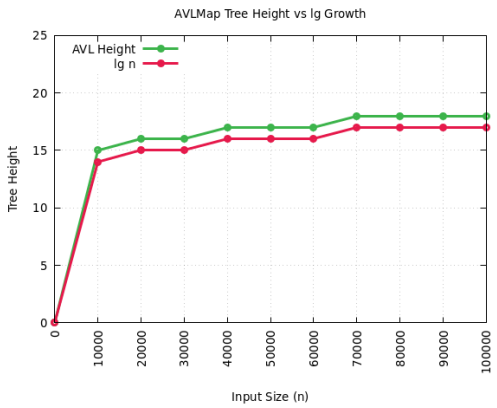
Finally, there is the HashMap and the AVLMap. These can easily be labeled the Map implementations with the lowest overall time complexities. Each achieves this goal in its own way.

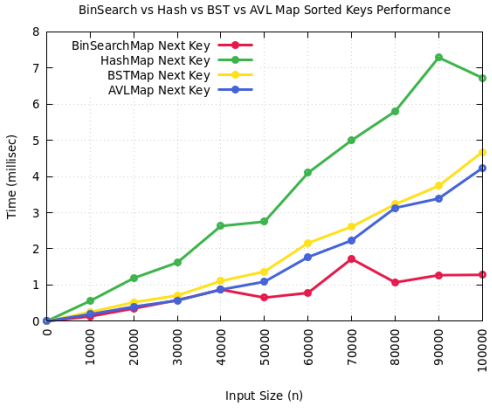
For HashMap, this is reached by hashing a code by utilizing the standard hash function method to quickly index a table of values. In addition, the ability to resize on occasions of rehashing allows space to open in the table. Thus, searching occurs over less lengthy linked lists when utilizing the separate chaining method of implementing a hash table. In other words, each spot in the table is reduced to such a low number of nodes that the time complexity is reduced to O(1). This behavior, as documented in the table at the start of this document, appears to be the case as the graphs produced when running performance tests on this data structure show HashMap as the ideal over the likes of ArrayMap and BinSearchMap for insert, erase, and contains. However, finding and sorting keys are more complex in comparison because one must iterate through the capacity and the count of nodes in each spot in the hash table. This produces key-wise functions with time complexities of O(n^2) rather than the ideal O(1) or even O(logn) for that matter.





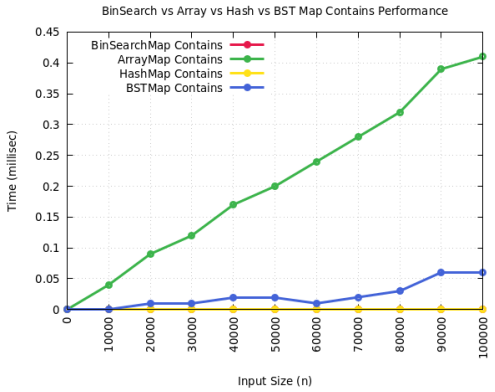
For AVLMap, the low time complexity is achieved by keeping a balanced tree such that the heights needed to navigate from the one root to several different leaves do not grow to unreasonable lengths. Allowing such, like in the BSTMap implementation increases the time it takes to traverse through a single root-to-leaf path. Graphically, the height growth of AVLMap mirrors that of the 'logn' function whereas the BSTMap height growth performs at O(n) complexity. Because AVLMap has an optimized height growth, it excels at many of the basic map functions. The only real issue it has is in sorting where, while it beats out every other implementation, it just cannot match the sorting maintained in the BinSearchMap. In the technical realm, most AVLMap functions appear to have O(logn) time complexities with sorting not far behind at O(nlogn). This produces graphs that have AVLMap as king of insert, searching, and erasing.

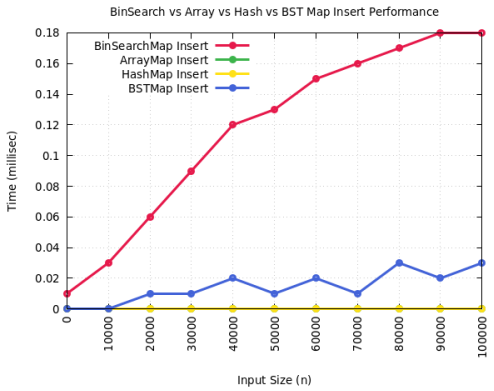




Application Scenarios

(a) The map implementation best suited for an application dealing primarily with insertion and searching at the same frequency would be the HashMap. The table above depicts the insert and contains functions of HashMap with identical worst-case time complexities at O(1). One might notice that the LinkedMap and ArrayMap also have insert complexities of O(1), this makes them viable candidate for this function. However, their contains complexity is worse than HashMap at O(n). Though it is not a significant increase in time complexity, all the other implementations take longer to perform the contains function, making HashMap excel at this scenario. For even further assurance, the performance graphs below also show HashMap as better than other implementations at these functions.





(b) If desiring an application that is still good at insert, but better at finding a range of keys, I recommend implementing a BinSearchMap. Above it is listed with time complexities of O(n) for insert and O(logn) for find keys. This is due to the nature of BinSearchMap, characterized by it keeping a sorted list throughout use. As an extended need of this application is to return keys in sorted order, BinSearchMap quickly becomes the favorite option available. It does have one of the worst insert complexities out of the implementations listed as shown in the right graph above, but this is solely due to searching for the correct place to insert a single bit of data in a list.

(c) However, if an application needs to do a set of inserts and then turn around and do a set of erases, I will switch back to the HashMap. As it is a good assumption that the factor of list size would need to grow rather slowly, HashMap proves its worth by resizing and rehashing elements when a specified capacity is met. This allows the space to perform inserts such that they begin to overpower the erasures. The cost of insert has not shifted from O(1) like in scenario ‘a’. However, erase also has a complexity of O(1) as there is no need to iterate or recursively traverse when specifying index using a hash function. As seen below, erase is proven to have a time complexity of O(1) for HashMap. A performance that outweighs even an implementation that remains sorted indefinitely.