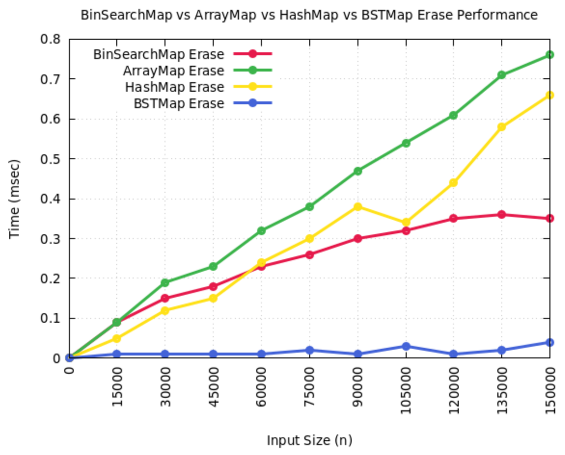
Colleen Lemak
Professor Bowers
CPSC 223
12 December 2021

Written Report

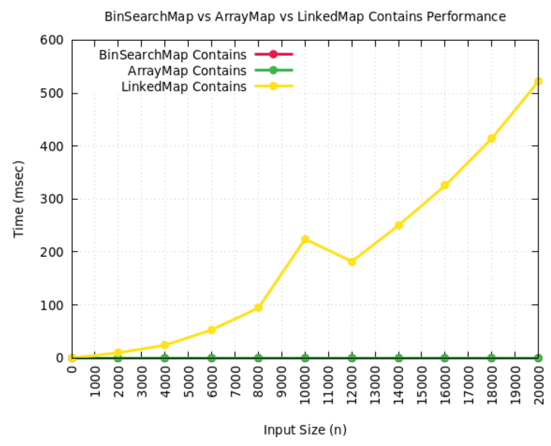
1. Comparison of Performance and Analytical Results

Operation	ArrayMap	LinkedMap	BinSearch	HashMap	BSTMap	AVLMap
insert	O(n)	O(1)	O(n)	O(1)	O(n)	O(logn)
erase	O(n)	O(n)	O(n)	O(n)	O(n)	O(logn)
contains	O(n)	O(n)	O(logn)	O(n)	O(n)	O(logn)
find_keys	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)
sorted_keys	O(n^2)	O(n^2)	O(n)	O(nlogn)	O(n)	O(n)

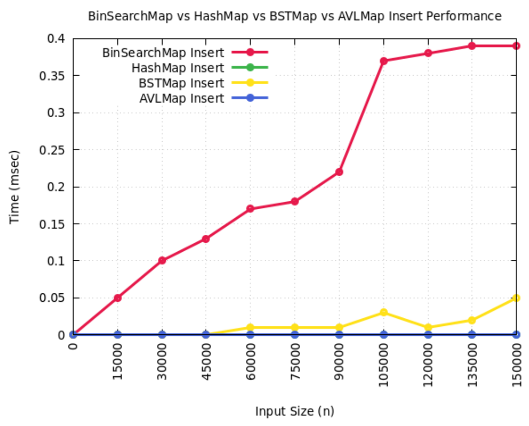
Discussion:
Starting with ArrayMap, some disadvantages are that erase costs O(n) because you have to linearly iterate through every element to check its value as shown below; this iterative process is common for other functions including insert and contains. However, ArrayMap provides accessing advantages as you can directly access a value by an array key or index.



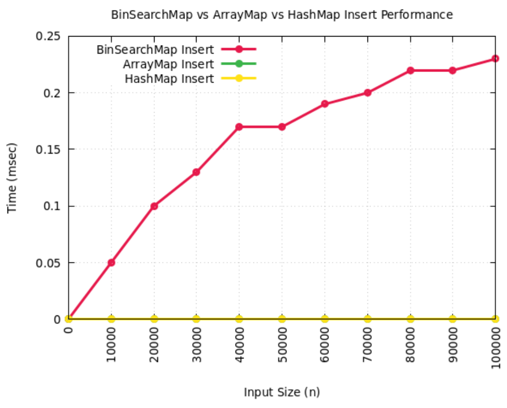
In LinkedMap, many functions are expensive because of the structure of a LinkedMap; sorted keys is O(n^2) because the quick sort algorithm is expensive in the worst case, especially if we pick a random pivot. However, its insert function is fast because you are allowed to insert at the head of the list, minimizing the cost to O(1).



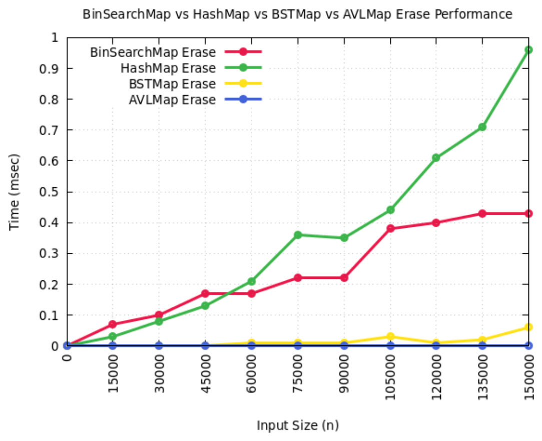
For BinSearch, insert is relatively expensive, as shown below, as you have to traverse to find the position to insert, by calling bin_search; this algorithm will be costly because you must search the left or right half of the array, and then afterwards, BinSearch's insert calls another insert which has to traverse through the list again. Insert is the main disadvantage to BinSearch, as other functions such as contains are improved. Because of BinSearch's ability to split the list in half and search based on values, it is much better in a worst case contains, O(logn), as opposed to an O(n) operation in ArrayMap contains.



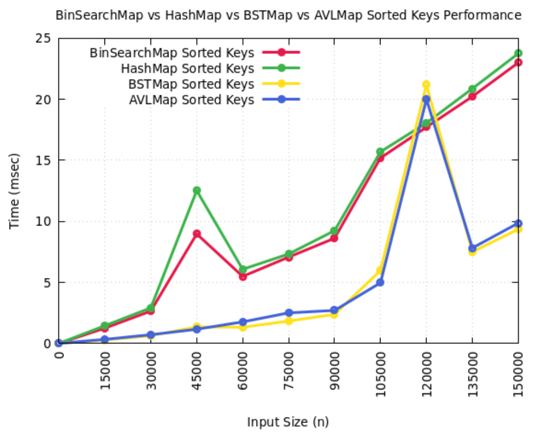
HashMap has many advantages such as O(1) inserting, however, it also has disadvantages as a result with find keys and contains being primarily an O(n) cost. Additionally, we call merge sort which is a O(nlogn) expense. With the table and bucket set up, you are able to quickly insert by hashing the index to get directly or very close to the proper bucket, as shown below as an O(1) operation. However, erase could be an expensive worst case because you may have to navigate to the end of the chain in a bucket, which may end up being O(n).

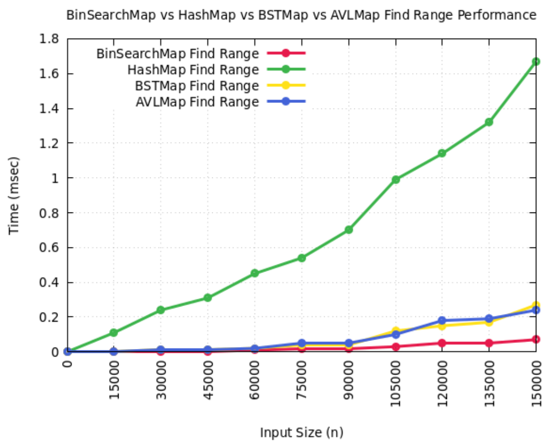


Another map implementation is BinSearchMap; a useful feature of this map is that it is easy to go through paths of O(logn) to insert or erase nodes. However, if the list is structured more like a linked list, it would be costly at O(n) to perform functions such as erase. A linked list would be the worst case because the tree would be heavy on one side and so we would need to traverse through which is expensive.



Another map implementation is AVLMap, which builds off of our BSTMap structure. In this map, you are able to erase and insert values and find keys quickly because you would be navigating a path which costs O(logn). Because this map must also be balanced with no balance factor over 1, you do not need to worry about traversing an entire list like BSTMap's implementation. However, AVLMap can be the most expensive when finding and sorting keys as shown below. This is because we need to visit every node which is O(n).





2. Application Scenarios

(a). An application where the two main operations are insert and contains, which are performed at approximately the same frequency.
- For this scenario, AVLMap is the best map implementation because its insert and contains are not very expensive in comparison to other maps. Others typically have at least one O(n) function cost, whereas AVLMap averages to about two O(logn) operations. When inserting, navigating a path costs O(logn) which may take more time compared to other maps like LinkedMap, however, when you consider contains will also be performed at the same frequency, it is easier to see how AVLMap's cost will not add up as quickly as any of the other implementations. A close second choice of implementation would have been BSTMap because of its similar structure, however, in the worst case scenario, you would have to traverse through the entire tree because it does not implement a rebalance like AVLMap. Because of AVLMap's unique rebalance function and its division of nodes to better navigate to a key, I would recommend AVLMap for this case.

(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 chose BSTMap for this situation because when you call find keys, the array sequence that is returned will already be sorted with an in-order traversal, so there is no need to call merge or quicksort on it, minimizing the cost of the find keys operation. The worst case of this implementation would yield an O(n) operation because it will have to visit every node to add it to our returned array, but it does not require an additional expensive operation to sort. The insert cost could at worst be O(n) as well, but the majority of the time it is relatively fast, O(logn), especially if the tree is not stacked like a linked list. Find keys on other implementations would

be slower in the worst case because an algorithm like quicksort could have a reversed list, in which case would be O(n). Because find keys is called more than insert, BSTMap would be the best option in comparison with HashMap and other implementations because of its unique ability to return a sorted list of keys.

(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 inserts than removes).
- For insert and erase, HashMap has a very efficient algorithm, especially if there are no repeated hash codes and it maintains a relatively smaller HashMap size. With this map, we are able to directly insert into the table and create chains within buckets which makes this operation fairly quick. Inserting is an O(1) operation as you can simply insert at the beginning of the bucket's chain, and erase is typically fast with the hashing index function, but can be O(n) in the worst case. Other functions in HashMap are slow, like contains, however, in this situation we are simply inserting and erasing with a slow growing map size, so using these HashMap features make the most sense and minimize the overall cost.