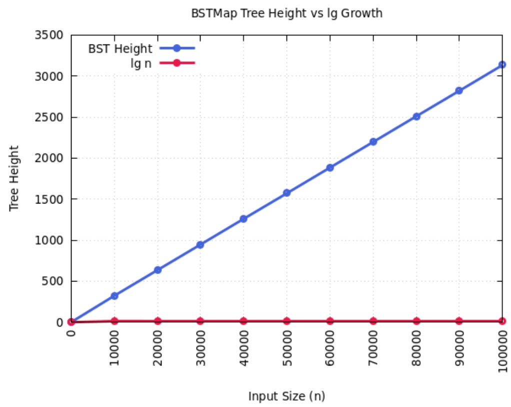
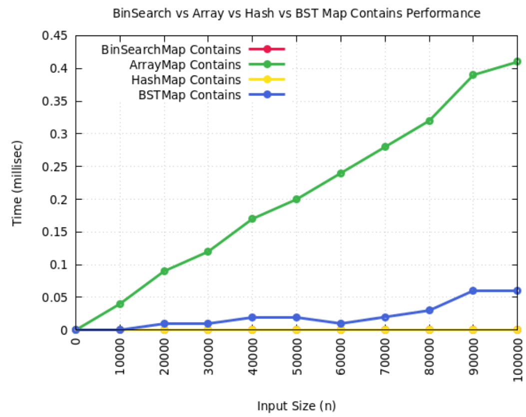
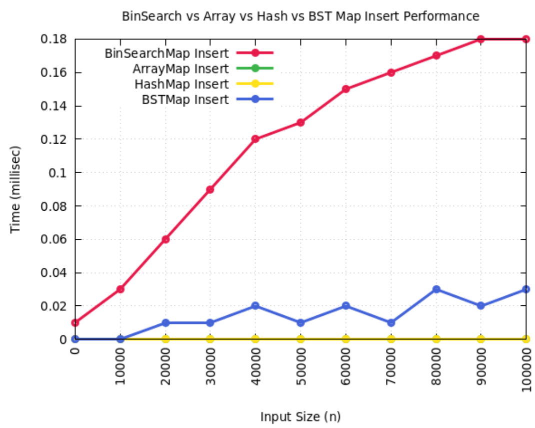
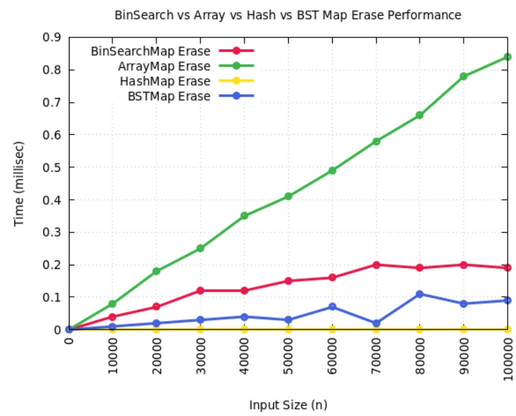
HW8 Writeup (Spring 2022)

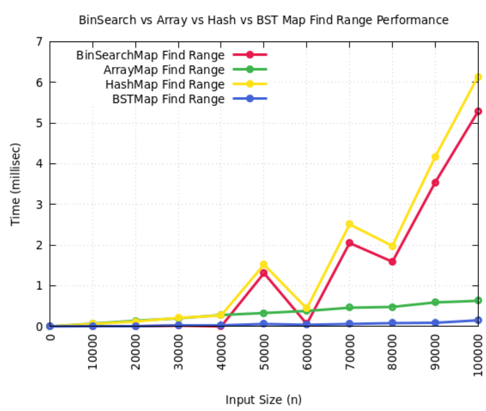
Plots:

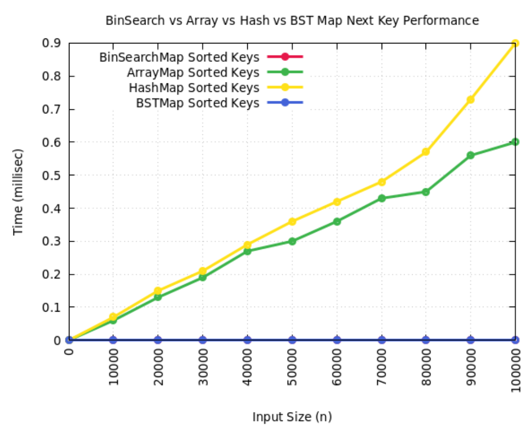


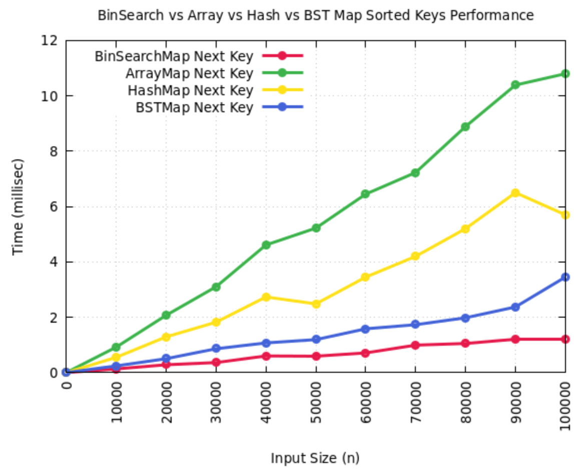












The tree height graph in the top left shows the height as calculated by the height() method within my BSTMap class. This function computes what the max height of the tree generated at each input size used. Compared to lg(n), this class is far from ideal when it comes to generating a tree with reasonable height for increasingly large input sizes.

The contains performance graph above shows the comparison of the contains() method between the four different Map implementations completed in this course. BST Map appears to be more complex than both HashMap and BinSearchMap in terms of time complexity. This is due to the need to navigate a path to a specific node within a tree rather than simply iterating through what is essentially a sequence.

The Erase performance graph in the bottom left shows the same four Map structures. This time, BSTMap beat out BinSearchMap even though erasing in particularly complex in terms of steps in BSTMap. I am not sure why BSTMap is faster in this case, unless BinSearchMap, in utilizing the BinSearch helper method, holds a worse time complexity than using the mix of iteration and recursion found within erase in BSTMap.

The Find Range performance graph shows BSTMap as the faster of the four Map implementations. This is simply due to the strict recursion being used to add only the necessary nodes, not every viewed node from within the tree.

The insert performance graph shows BSTMap as more efficient than BinSearchMap, but worse than ArrayMap and HashMap. This can be attributed to both iterating down a root-to-leaf path and the constraint of only inserting leaf nodes. Whereas BinSearchMap requires an O(nlogn) complexity to search for the key and then insert it at its correct place as required to keep a sorted list for BinSearch.

The sorted keys graph above places BSTMap between BinSearchMap and HashMap in time complexity for sorting a Map. As BinSearchMap maintains a sorted list throughout operations, BSTMap can be said to be the fastest in terms of the structures implemented in this course that sort a list in real time with this function. My implementation of sorted keys in BSTMap utilizes a call to find_keys (find range) after iterating to where a key should go.

Finally, the next key graph displays the performance of each structure in obtaining the next sequential key after any key is passed in as a parameter. BSTMap appears to have an almost constant time complexity, like BinSearchMap. As the other two structures take longer, it is hard to determine what time complexity is help by these two fast next key structures. Though, by looking at my implementation, I can reason that BSTMap follows a path using iteration to obtain the location of the next sequential key in the sequence.

Challenges:

I only struggled to implement next_key and prev_key. These functions do not pass their respective tests. Also, when running valgrind I can see a large error count produced in around 660 contexts. I am unsure as to why these errors are present.