Evan Conley
Prof. Sahr
CS 411
3/14/2017

Lab 5 Analysis

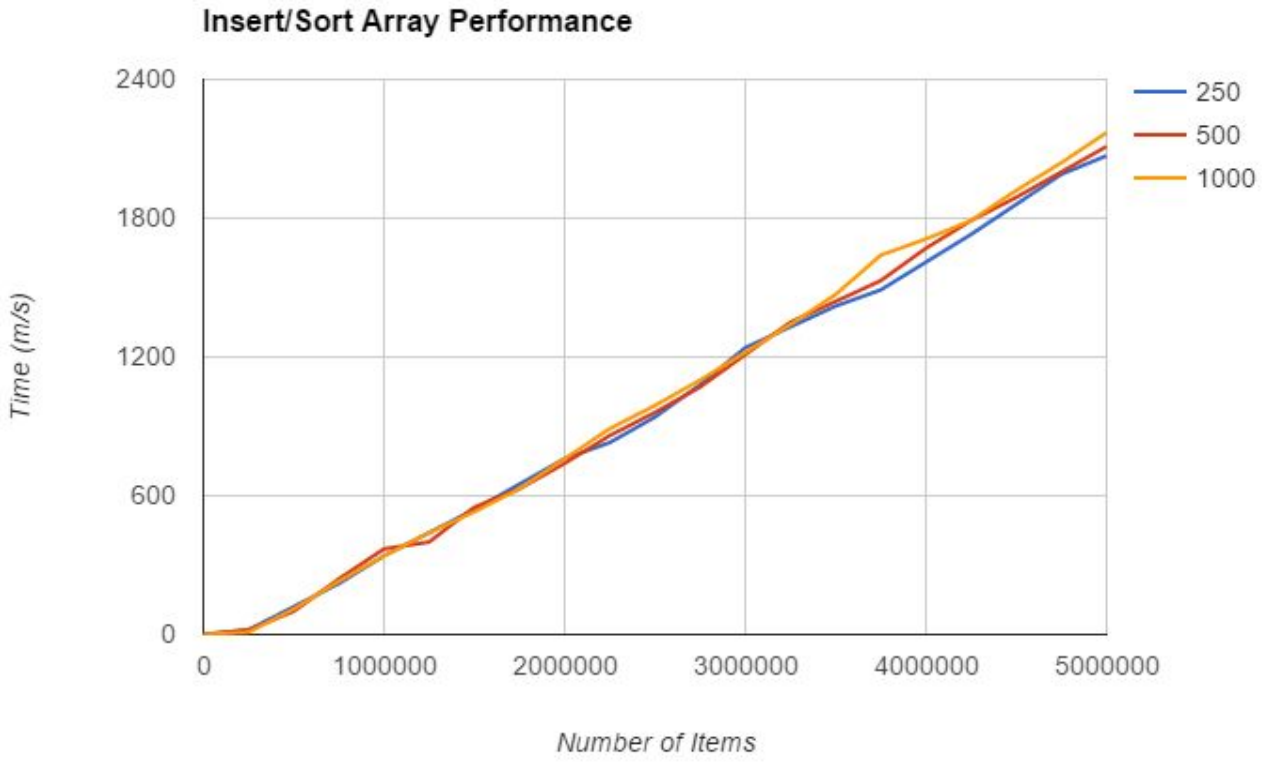
In comparing the overall complexity, and in doing so, the efficiency, of an unordered array insert followed by a counting sort to the insertion into a binary search tree (BST) the expectation was to find that BST would be more efficient. This initial inclination was drawn as, from the lectures, we noted that insertion into a BST, as long as it held balance, would have an average case of O(logn). The discussed complexity of an insertion into an unordered array was O(1) as a bounds check could place the element at the end of the array. Counting sort, though, increases the complexity of the overall algorithm as it is O(n+k) where n is the array size and k is the count of the elements in the array.

Despite the initial hypothesis, the findings of the lab showed that the unordered array insertion followed by counting sort was far more efficient time wise. The complexity O(n+k) seemed to, at first, outgrow that of the O(logn) insert of the BST, but, through analyzing the data structures independently, insert into a BST seemed to grow along with the data set size. Although a singular insert into a BST has average case O(logn), by dealing with a data set of size n, this logn insert is completed n times, making the overall complexity of the insert of a data set O(nlogn), whereas the order of the counting sort insert was only ever reliant on the O(n+k) sorting method.

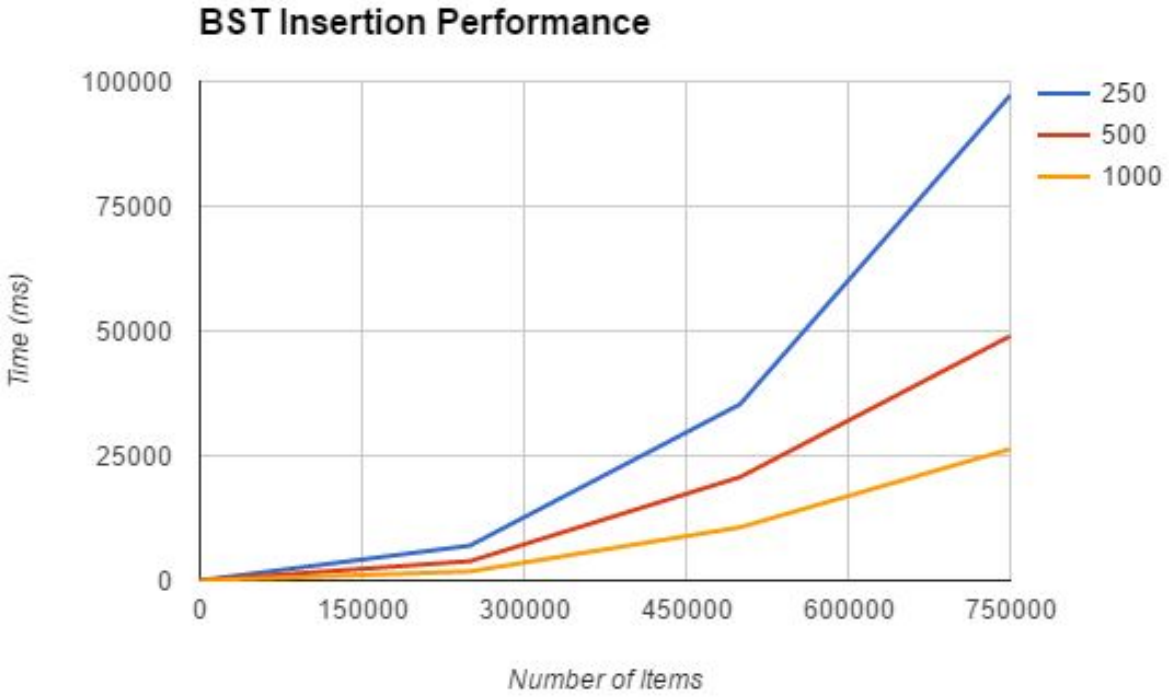
Further, during the experiment, I varied the number of items to see the efficiency of each data structure with respect to greater data set sizes. For the array, I was able to vary between data set sizes of 0 to 5000000. I could not get passed 750000 for the data set size for the BST insertion trials, though, as my attempt at 1000000 or over caused the program to run for too

long. As to memory, I believe that they both would take up a similar amount of memory. The memory complexity of a BST would be O(n) and the, although temporarily created for the sake of the counting sort, the count array of size k would mean, at maximum, the memory complexity of the array would be O(n+k) during counting sort and O(n) otherwise.

In regards to changes in performance with varied values for the maximum ID, only one of the data structures had any noticeable effect. The counting sort array showed little to no variation, but the BST insert began to take noticeable less time with each doubling of the maximum ID. With each doubling of the maximum ID, the time for the insertion into the structure was cut nearly in half. This could be due to the balancing of the tree. At smaller ID's with higher data set sizes, more ID's were more likely to branch out in an unbalanced manner as repetitions of potential ID's would increase the height of the tree. In the alternate case, the higher the maximum ID, the less likely there are to be a high number of duplicates, reducing the height of the tree overall and keeping it balanced.



Number of Items	Maximum ID		
	250	500	1000
0	0	0	0
250000	20	20	10
500000	120	100	110
750000	220	240	230
1000000	340	370	340
1250000	440	400	440
1500000	540	550	530
1750000	650	630	630
2000000	760	740	760
2250000	830	860	890
2500000	940	960	990
2750000	1080	1070	1100
3000000	1240	1210	1220
3250000	1330	1350	1340
3500000	1420	1440	1470
3750000	1490	1530	1640
4000000	1610	1670	1710
4250000	1730	1790	1790
4500000	1860	1890	1920
4750000	1990	2000	2040
5000000	2070	2110	2170



Number of Items	Max Item ID		
	250	500	1000
0	0	0	0
250000	6990	3870	1820
500000	35320	20720	10650
750000	97230	48990	26340