Container Review and Analysis

Rosa Kim Cho
CPSC 131

As depicted in the graphs within this review, the vast spectrum of the representation of data contained in five different data structures is quite diverse. Each function of inserting, removing, and searching carry their own average times to begin and end depending on their respective data structures. Here, we shall look through five different analyses of the relationship between different functions within the same data structure, similar functions among different data structures, and more.

Comparing the cost of time of inserting data between a binary search tree and a hash table is quite evident by their average-times. As both utilize nonlinear algorithms as well as unique structures, there are a few noticeable differences. First off, Binary Search Trees (or BST) here has few spikes in average time with one notable one at around 309 mark on the x-axis. In the case of BST, the big-O analysis that best describes it is log(n) as it mostly remains level with one exception. However, the Hash Table is not as level as BST but there are notable spikes in multiple areas where the hash key is in play. This is to be expected as Hash Tables are usually O(1) average and amortized case complexity but, in the case of the spikes, it reaches an O(n) worst-case complexity. Too many items were hashed at those times thus it reaches a stressful point. A real-life example of this would be the constantly changing inventory of a pharmacy where different prescriptions specific to a client need to be refilled once they are done. Or, in some cases, completely new prescriptions are made so new orders are made to fulfill them. For two data structures that utilize paired nodes, there can be noted differences in their Big-O analyses of their runtimes with insertion.

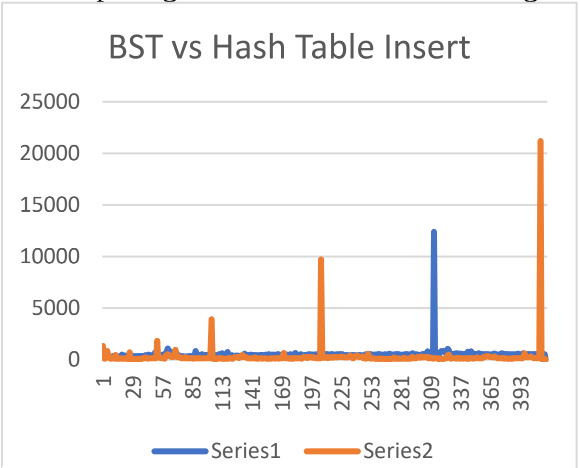
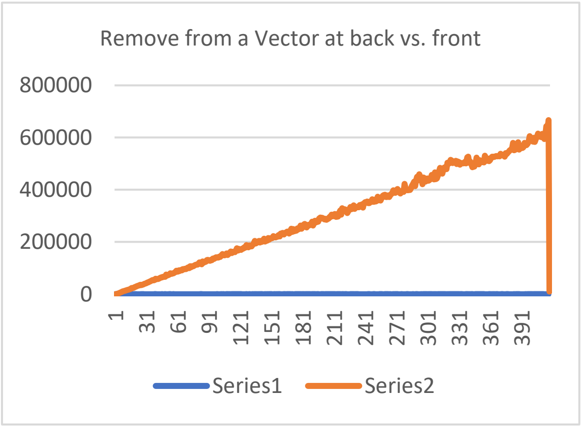
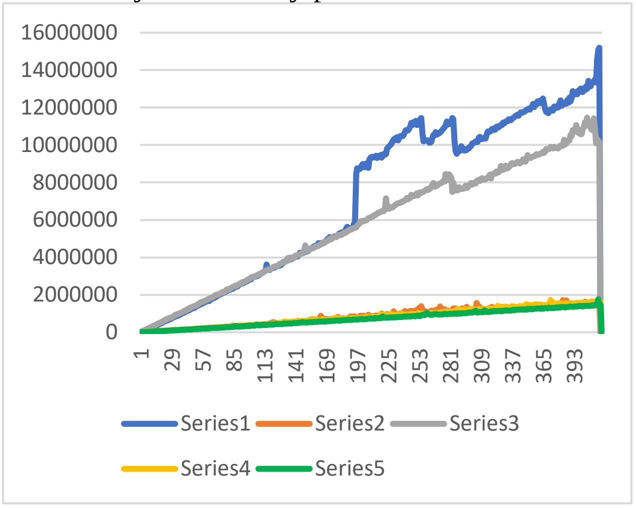


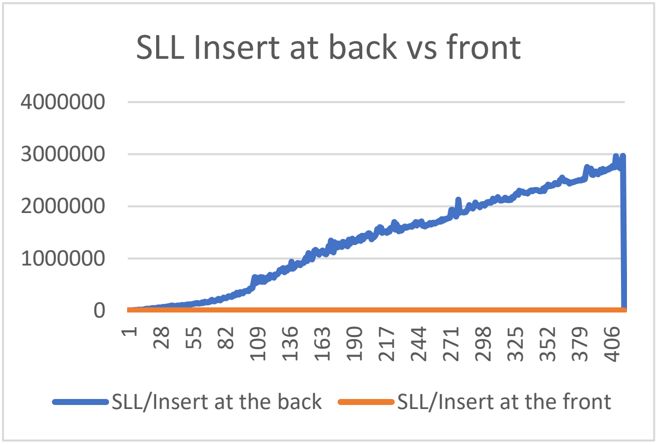
Figure 1: Series1=BST, Series2=Hash Table



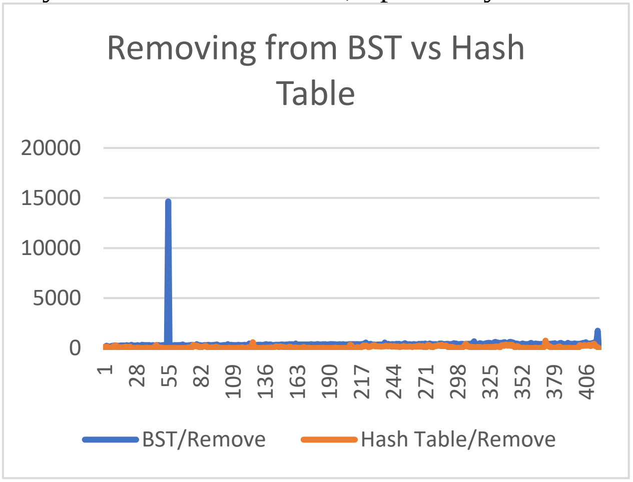
Removing from vectors at the back (*Series1*) vs. the front (*Series2*) shows a huge difference in run time between the two. Removing at back is a simple-enough procedure that uses a constant/O(1) time. However, none could be said for removing from the front as it reaches a zenithal peak with O(n) average. As the vector grows, the longer it takes for the container to go through each element within until it reaches its ascribed point to remove a new element. Still, certain pieces of data need to be removed whether it is the oldest, outdated piece of information (remove from at front) or a new piece of data that, unfortunately, is not wholly accurate and needs to be purged (remove from at back). It is lengthy and certainly expensive time-wise with the at front function. Hence the reason why it is usually preferred to remove from a vector at the end.



Searching through a vector (*series5)*, doubly-linked list (DLL/*series2*), singly-linked list (SLL/*series4*), BST (*series1)*, and a Hash Table (*series3*) demonstrates the fully vast and diverse range of big-O complexities across each different data structure. The vector search reveals a linear/O(n) complexity with a steady slope. DLL also demonstrate a linear/O(n) complexity with little difference. In both cases, they are preferably to SLL when looking for both the front and back of the data structures in search of archived items like books. SLL also displays a linear/O(n) complexity. Despite its limitations, SLL still remains useful when one needs to search for specific items organized in order through time. The Hash Table here has a log(n)/worst case scenario happening which is noted by a steep slope with 2 drops. Usually, it is demonstrated by a linear/O(n) slope but here, there was a sheer amount of data involved with a particularly tricky hash key to delve with. Searching within a hash table though can be quite useful for items organized like databases in which there is a key (or an ID) and its object. The most dramatic of them all is the BST which has a log(n) complexity as demonstrate by the table-like shifts in its slope. The BST remains favorable in terms of searching for paired objects which, like the keys of Hash tables, can be its primary identifier yet is organized to minimize wait time through a nonlinear way.



The SLL insert at the back vs the front, like removing, displays a sheer difference on how position matters which is one reason why I chose this. However, I primarily chose this trend because it reveals a complete opposite in big-O trends for, unlike the remove function, inserting at the front proves to have a more constant/O(1) slope in comparison to inserting at back which has a notable linear/O(n) slope. The reason lies in the fact that, as the container grows, so does the length of time it takes to travel through the SLL to get to the end. It also demonstrates just how much more cost-effective other data structures are at inserting and effectively growing a container with a huge amount of data.



I chose removing from a BST vs a Hash

Table to demonstrate another dramatic different between the two data structures. As noted by the spikes in slope of the BST, there is a demonstrably O(log n) slope which steadily changes but eventually evens out as there is more data involved to parse through. As the function doesn't have to travel through every single branch of the tree, it evens itself out in the end. However, things are much more constant/O(1) with the Hash Table for removing an element in a non-linear fashion proven to be quite cost-effective. Either way, both unconventional narratives show that, as more data is increased, it is neither preferred nor necessary to parse through every single one of them. Their big-o complexity proves it so.