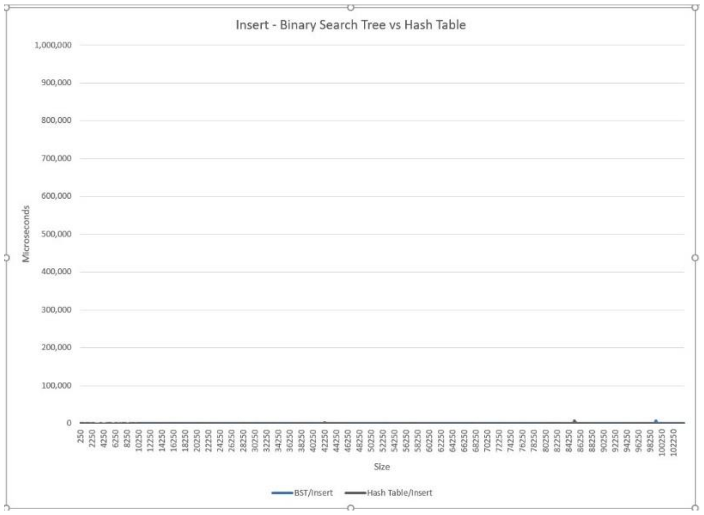
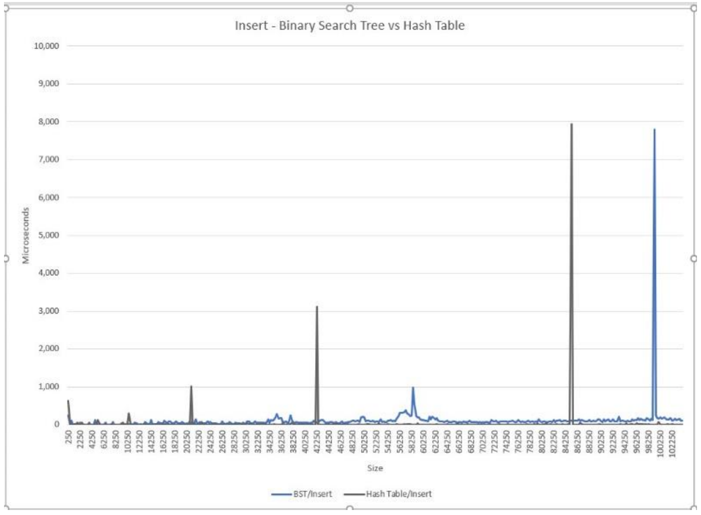
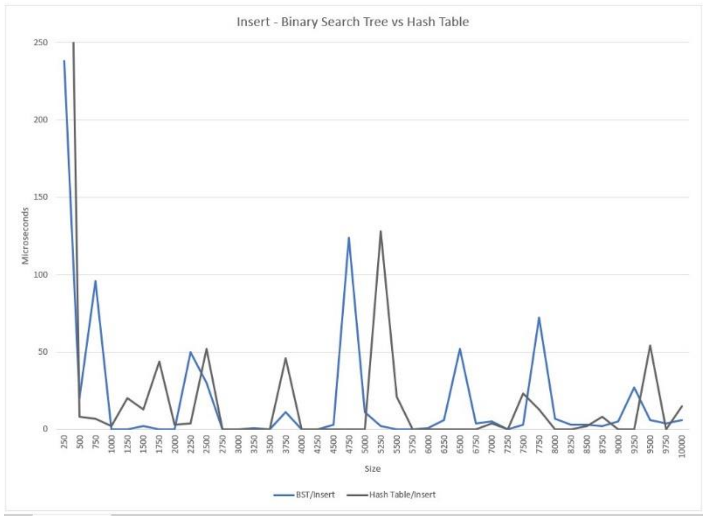
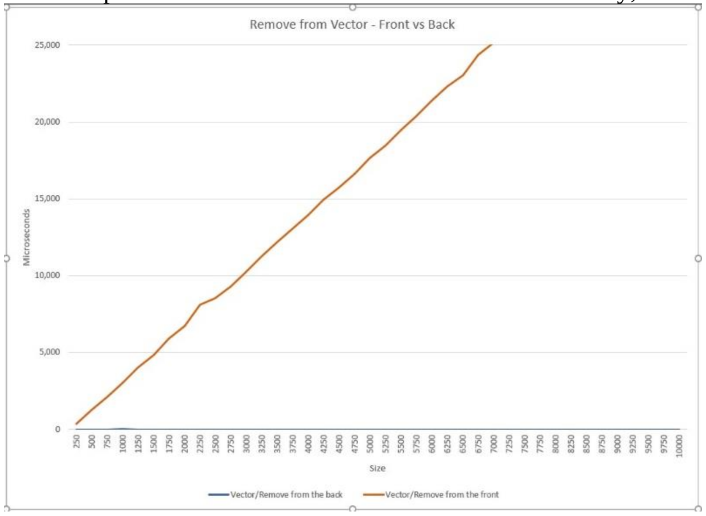
Jason Angel

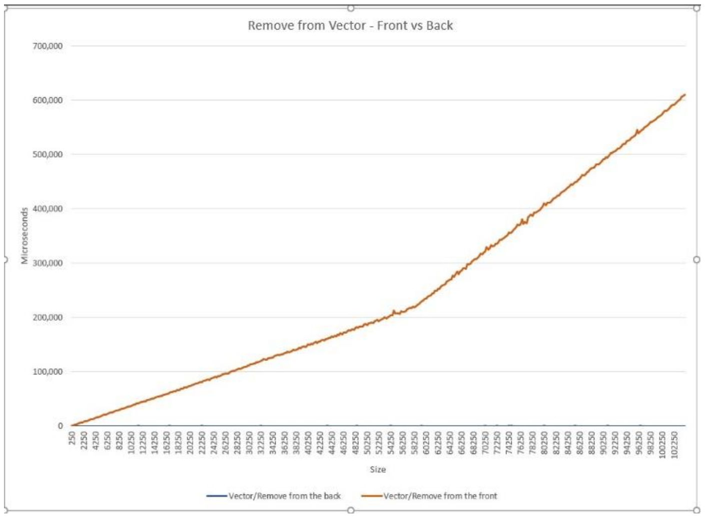


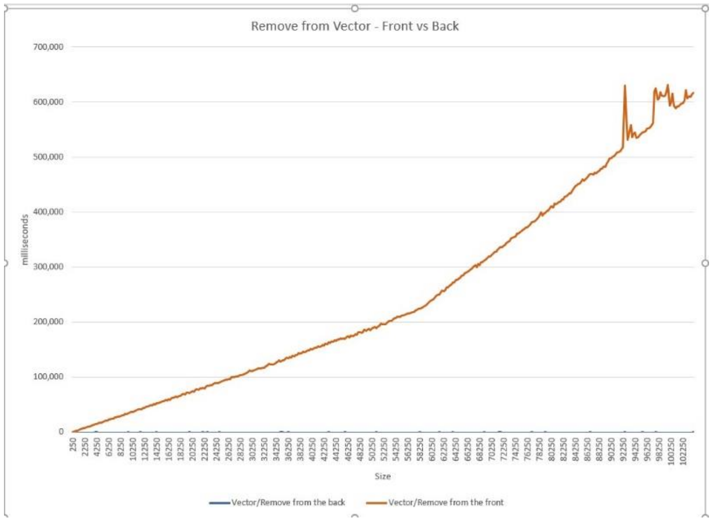




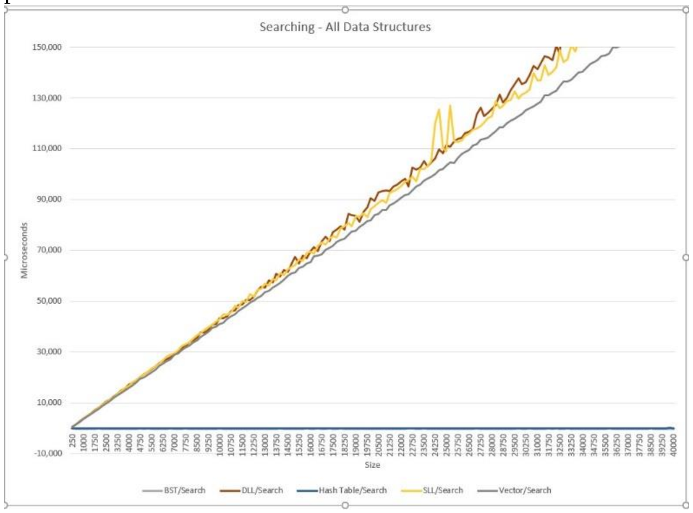
With the dark grey line representing the Binary Search Tree and the blue line representing the Hash Table, it’s clear to assume insertion of both assistive containers is extremely fast. The BST sorts data and compares it in a way that it allows for faster time in several operations such as insertion compared to sequential containers. Hash Tables on the other hand do not sort data. Instead Hash Tables combine linked lists and vectors along with some modular arithmetic. This allows for faster times than a BST, which is already an extremely fast data structure. The top left graph initially gives us the impression that both data structures are constant. This of course isn’t the case as the complexity for insertion is O (log n) amortized for a BST and O (1) amortized for a Hash Table. Because both data structures have amortized complexity, it means that the time complexity can change for both data structures depending on how the insertion operation is utilized and the design of the data structure such as its size. BSTs can have O(n) worst cast time for insertion and so too for Hash Tables. This explains the arbitrary peaks that are shown in the top right and bottom graph. Depending on if the BST is balanced and what is inserted, the BST can lose its O (log n) performance easily. If an element is less than every single element that exists in the BST, it would have to be inserted all the way to the bottom left of the tree. Similarly, if an element is greater than every single element that exists in the BST, it would have to be inserted all the way to bottom right of the tree. Such behavior leads to O(n) worst cast time, similar to that of a linked list or vector which we don’t want. A Hash Table can have O(n) worst case time because too many elements are either hashed in the same key or the Hash Table needs a bigger table, which requires for every element to be reinserted. It should also be noted that a Hash Table is much more random because of its use of modular arithmetic when constructing the data structure. The way a Hash Table is constructed is not predictable as opposed to a BST, which we already know is sorted. Another explanation for the peaks for both data structures is the size. Some elements will be inserted more quickly than others and this will become more prominent as the size of the data structures grows. This could explain the two tall peaks shown in the top right graph for both the Hash Table and BST. Nevertheless, the times for insertion for both data structures when they grow in size are still fast and seem consistent to when they were a much smaller size. Because the way the BST and Hash Table work, such behavior is expected and normal. With all that said, because Hash Tables are O (1) amortized time when inserting, it performs much better than BSTs. This is especially noticeable in the top right and bottom graph. The bottom graph, which is scaled to 250 microseconds, shows the peaks of the BST and Hash Table in greater detail. In most cases the blue line (the BST) is above the grey line (the Hash Table). One real world example where insertion can be applicable to a BST and Hash Table is a dictionary. This type of example would go well for a BST because we create new words all the time and everything is sorted so there will be no randomness that a Hash Table experiences. The BST might be slower as the graphs show, but it is predictable. Additionally, one would have to create a specific Hash function to work with the dictionary, which can be difficult.

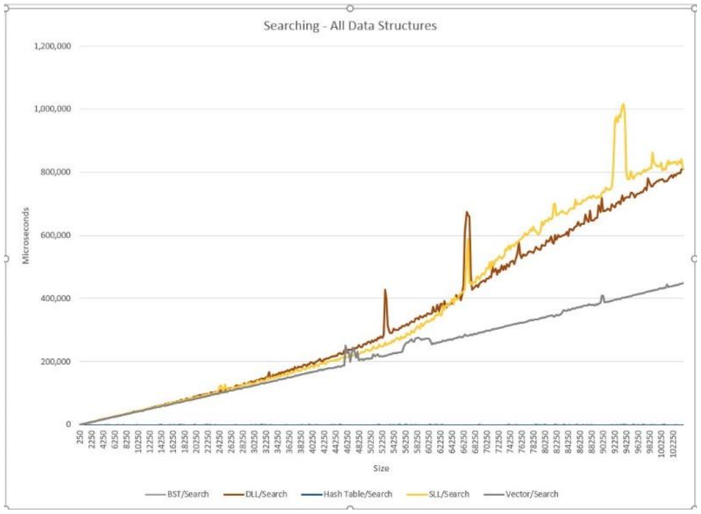


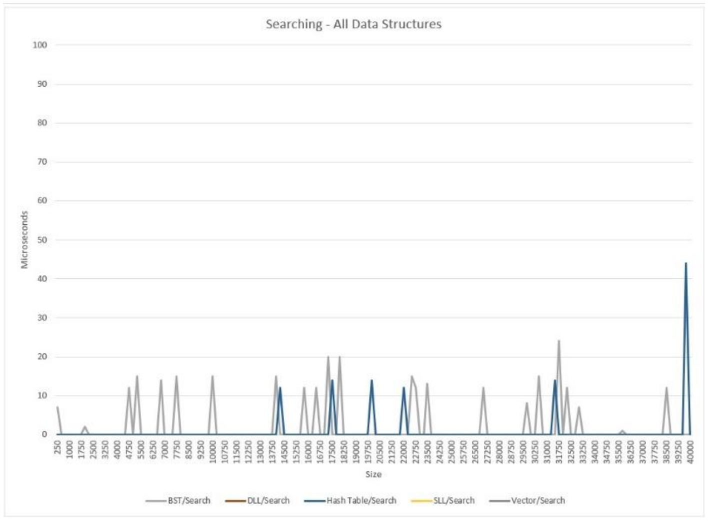




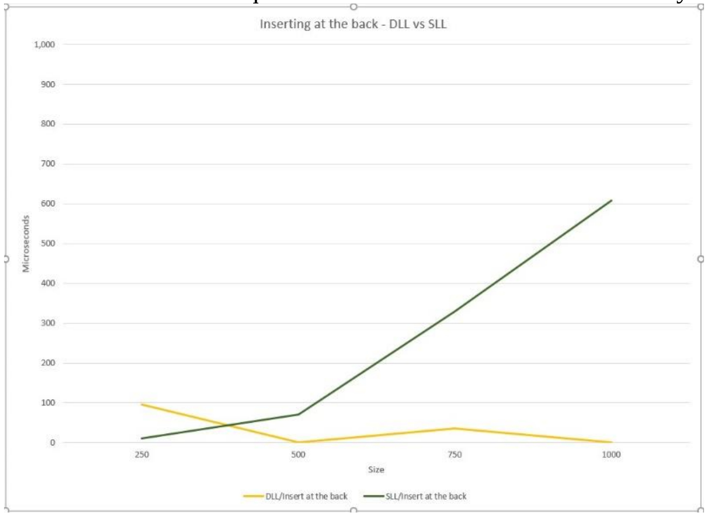
With the orange line representing removing from the front of the vector and the blue line representing removing from the back of the vector, we can immediately tell what has better performance. The orange line appears linear and the blue line appears constant, which is and should be expected. Since the vector is a sequential data structure, removing from the back of the vector has O(n) complexity and removing from the back is O (1) constant time. Why the performance between both operations is so different makes a lot of sense if you understand how the vector works. Since the vector is sequential, it stores elements according to how the client wishes. Additionally, one can access elements through random access. But just like any other data structure, the vector has its drawbacks, especially when it comes to insertion, removal, and searching within the vector. The time complexity of the removal operation depends on which end of the vector is affected. Removing from the back of the vector does not require shifting elements whereas removing from the front does. Removing from the back of the vector does not affect the data structure unless the client shrinks the capacity of the vector accordingly. For the first graph there is nothing special going on other than some small negligible peaks (note that the graph is scaled to 20,000 microseconds). I say they are negligible because we are so zoomed in into the graph where such peaks are most likely natural as they are probably from the fault of the computer. Upon zooming out and looking at the whole lifetime of the vector object however, which is shown in the second graph, there is a noticeable sharp change in steepness for the remove from the front operation. It likely has to do with the size of the vector. The orange line continues to be linear shortly after. Either a hiccup happened, or the size really affects inserting at the front that badly. But if the latter was the case, why did it occur in a certain size interval? That is why I ran the program for a second time. The third bottom graph represent the results of the same operations. The orange line has a similar trend. It starts linear and eventually has a sharp change in steepness but then continues to be linear. Therefore, I believe the size is what causes the abrupt change in steepness. However, there are some irregularities in the orange line near the end of its lifetime. Because that didn't occur on the first test, I'm going to assume those irregularities are just at the fault of the computer. Of course, I would need to run more tests in order to get a better understanding. On the other hand, the remove from the front operation appears constant as it should be. There is nothing funny going on. You would have to zoom in extremely close to the graph in order to see detail of the operation. But at that point such peaks are negligible. The application of a vector is broad as vectors are intended to hold any object. One real world example of removing from the vector is removing a book from a list of books. If one day we decided to do some clean up on the list, removing from the back sounds like a better idea than removing from the front, as shown by the graphs. Of course, it would all depend on how the vector is sorted and what is trying to be accomplished. But if we ever have the option between removing from back or front, removing from the back is the ideal choice because of its constant time performance.

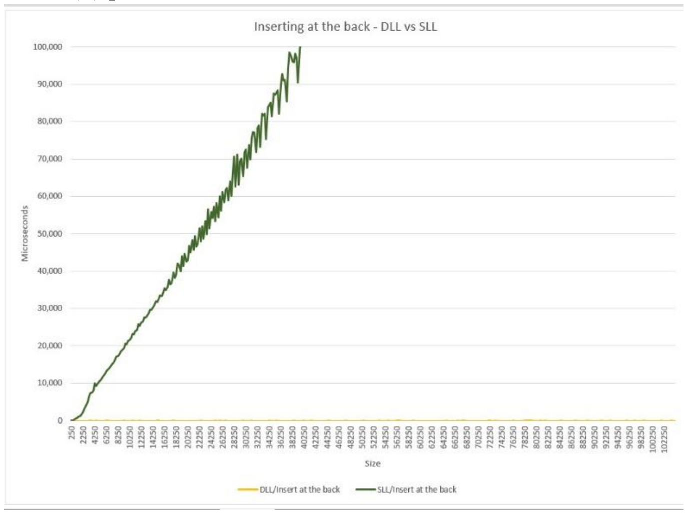


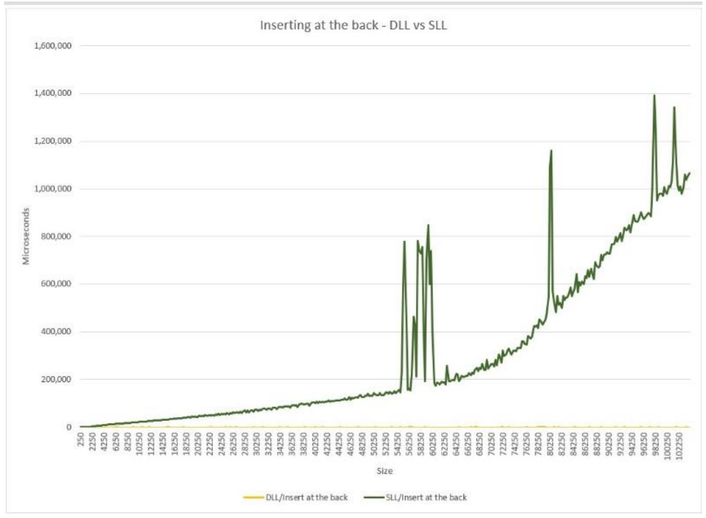




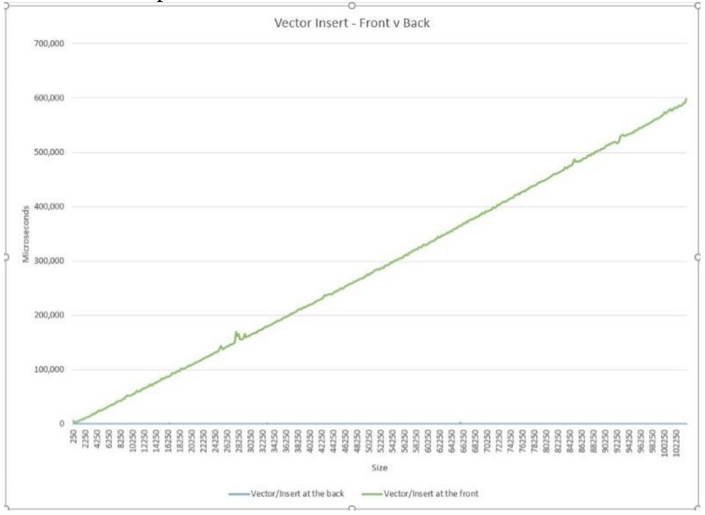
When it comes to searching, the Hash Table and BST easily outperform all other data structures. You must zoom in significantly to even begin to see detail of the Hash Table and BST to the point where the rest of the data structures seem non-existent. That is how badly the DLL, SLL, and Vector lag behind the Hash Table and BST when it comes to searching. From the top left graph, we can see that the DLL, SLL, and Vector trend similarly, with some performing better than others. The similarities in performance make sense as all three mentioned data structures are sequential, whereas the BST and Hash Table are not. However, a SLL is more similar to a DLL than a Vector. The SLL and DLL are node-based data structures with the SLL being the simpler of the two. The Standard Library SLL maintains only a head pointer, whereas the Standard Library DLL maintains a head and tail pointer. That difference alone brings in much more complexity, but there shouldn't be any difference in the search operation because they're still both sequential. Since the SLL and DLL are node-based data structures, they are different from a vector in that they do not have the ability of random access when it comes to retrieving or modifying elements. Still, at the end of the day, all three data structures are still sequential and so their complexity when it comes to searching should be the same. Therefore, before I saw the graphs, I was expecting all the sequential data structures to perform about the same. But the top right graph, which is scaled to the lifetime of the objects, says otherwise. The difference in performance for the sequential data structures becomes even bigger. The SLL ends up with a higher time average as the data structure grows bigger. The time average for the DLL also increases, but like just said, it is taken over by the SLL. Meanwhile the performance of the vector doesn't have any notable irregularities. All the sequential data structures should be O(n) when it comes to searching, but we're seeing some notable difference in performance starting at the top right graph. This of course could be because how I designed my search algorithm for the sequential data structures, which is likely the case. But even then they all had essentially the same algorithm, so the difference in performance between the sequential data structures caught me by surprise. The BST and Hash Table on the other hand seem to show expected performance. The third graph shows an extremely zoomed in view so that we can see detail of the time average for the BST and Hash Table. The peaks that are shown are likely because of worst case scenarios happening. We aren't seeing continuous increase of time, however. It goes to show how fast the search operation is for these two data structures. This is because they are not sequential. Instead, they search through elements by either sorting (BST) or by hashing (Hash Table), which eliminates having to go through each element unlike the sequential data structures. When zoomed out the BST and Hash Table appear constant because size isn't a big player unlike the sequential data structures. The sequential data structures continue to increase their time but the associative data structures stay flat. One real world application of searching that can be applicable to all of these data structures is checking whether someone is a member of an online store. From the graphs shown, it would be wise to pick the Hash Table or BST. Those would definitely be good choices as they are far faster than O(n). But what if we want a simpler data structure to work with? Since the other data structures are sequential would it matter what we pick? Based on the graphs it seems it does. If we want a simple data structure to work with and don't care about performance as much then the vector seems like a good choice for searching. With that said, if we're working with very large data we'd likely would want to avoid the vector and the other sequential data structures because how they suffer O(n) performance.

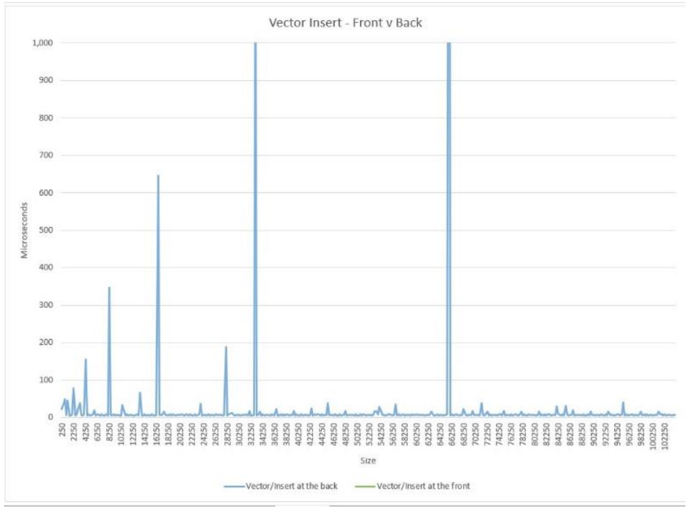






Inserting at the back of a DLL and SLL are very different in performance as expected. A SLL is simpler version of the DLL because it only maintains one pointer, which is the head. Of course, there is nothing stopping a person from implementing a tail pointer in a SLL. The Standard Library SLL does not do that, and as a result, pushing to the back of the linked list is not possible unless you use iterators. For a DLL the time complexity for inserting at the back is O(1) and for a SLL it is O(n). Looking at these graphs however we can tell that the performance for the SLL isn't necessarily O(n). The top left graph, which only consists of 1000 items in size, as well as being extremely zoomed in, shows how quickly the SLL changes in performance relative to its size. The DLL on the other hand, although initially starting out with a higher height than the SLL, does not show any increase in time average of its remove operation. As a matter of fact, the height of the yellow line trends downward relative to its size in the top left graph. This makes sense because the remove operation for a DLL is constant. Also, the initial height of the yellow line is likely the fault of the computer anyway. If the yellow line was anything other than constant that would mean there is something wrong with the program. The top right graph and bottom graph show the same behavior of the top left graph but in a bigger picture. We can see that the SLL increasingly has higher time averages relative to its size while the DLL remains constant. With the top right graph, we can initially assume that the SLL is linear, but that logic seems inconsistent with the third bottom graph. There are several peaks throughout the green line in all three graphs. In the top right graph, all the peaks are clumped together but in the bottom graph we can see tall peaks at certain size intervals. This is most likely because how I designed the algorithm for inserting at the back of the SLL. Because the SLL does not maintain a tail pointer, the node's next pointer must be checked if it points to null. I used std::next for that, all the while traversing the list like normal. The specific line of code is the following: std::next(iterator) != container.end();. I'm not 100% sure if that line of code is constant or not, and whether it affects the traditional traversing of the linked list. Regardless, for now I am going to assume that line of code is the reason for the abnormal linear line of the SLL. The DLL on the other hand does not have unusual behavior. When zoomed out the yellow line is constant as expected. Because the DLL maintains a tail pointer, there is no need to traverse the whole list. The tail pointer gives access to the end of the list, so inserting an element at the back requires no work. One real world example of inserting that can applied here is to maintain items in a shopping cart. Judging by the graphs, if we were given the option to insert at the back of a DLL or a SLL, it would be wise to insert items at the back of a DLL. If we inserted at the back of a SLL, we would suffer O(n) performance, which essentially removes the practicality linked lists offer. If we inserted at the back of a DLL, we would have constant performance.





Since the vector is a sequential data structure, the location where insertion happens in a vector is important. With the green line representing insertion at the front and the blue line representing insertion at the back, it's clear which one performs better. Inserting at the front seems to have linear performance as expected and inserting at the back seems to have constant amortized performance as expected. The performance of both type of insertions are consistent with how the vector works. Inserting at the front requires to shift all other elements to the right, which means inserting at the front has O(n) complexity. Inserting at the back on the other hand requires no shifting of elements meaning that inserting at the back has O(1) complexity. With that said, inserting at the back isn't always O(1). There are times where the vector needs to be resized which requires a bigger array to store the elements. Every element would need to be reinserted into the new bigger array and as a result resizing of a vector has O(n) complexity. As such, when we average O(1) complexity and O(n) complexity out, inserting at the back of a vector is O(1) amortized. The right graph proves this. The tall peaks of the line are when resizing occurs. Notice that the first couple tall peaks are shorter than the subsequent tall peaks. This makes sense because as size grows, so will the time it takes to resize. There are several small peaks clumped together at the bottom of the right graph but keep in mind the graph only has a scale of 1,000 microseconds. It's likely that those small peaks are just natural. A perfect constant line is very difficult to achieve, especially when you're running on a virtual machine. Meanwhile the green line, which represents inserting at the front, is not even visible in the right graph. We must zoom out a whole lot in order to get a good representation of the green line, which is shown in the left graph. Although the green line appears linear, it also has peaks in certain size intervals just like the blue line. At first it seems abnormal as to why the green line would have peaks. But we must realize that resizing can also occur when inserting at the front as well. The vector will eventually reach its capacity regardless of where insertion happens. Therefore, those peaks likely represent intervals where resizing of the vector occurs. Keep in mind that the peaks of the green line don't appear as significant as the peaks of the blue line to the right because the left graph is scaled to 700,000 microseconds, whereas the right graph is scaled to 1,000 microseconds. One real world example of inserting into a vector is inserting names of students in a university. As shown by the graphs, inserting at the back of the vector seems like a better idea. There is the possibility that inserting at the back would suffer O(n) performance every now and then, but it is still miles better than having O(n) performance all the time.