Runtime Analysis of Different STL containers

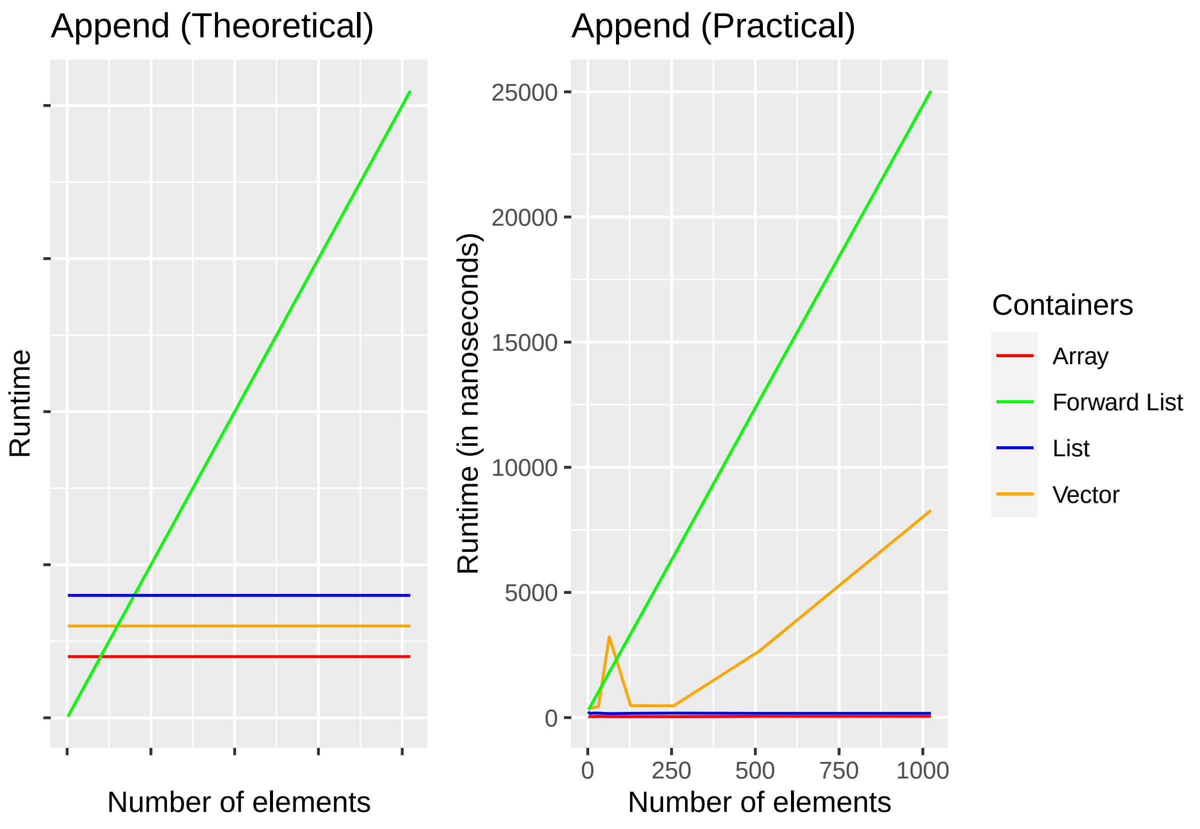
In this project, we will deal with four of the STL containers:
1. Array
2. Vector
3. Forward List
4. List

We will analyze runtimes of the aforementioned containers for performing the following functions:
1. Appending
2. Removing from End
3. Prepending
4. Removing from Front
5. Emptying
6. Swapping
7. Reversing
8. Inserting
9. Removing Duplicates

We will measure the runtimes for each of the containers for executing each of the aforementioned functions with varying numbers of elements. The number of elements assigned to these containers to measure their runtimes will be 2^n where n can take positive integers between 1 and 10 inclusive. To be more specific, the size assigned to these containers will be in the range 2,4,8,16,32,. . . .,1024. For all the functions a trial run will be executed so that unexpected values in runtimes that occur in the beginning can be avoided. The practical runtimes that we have obtained will be compared to the theoretical runtimes of the different containers for performing different functions. Both the practical and theoretical runtimes will be in the worst case scenario unless otherwise specified. When the theoretical runtimes of different containers are projected to be the same, different arbitrary slopes will be assigned to each container's runtime line graph.

1. Append

Lets visualize the runtimes for appending in the four aforementioned containers.



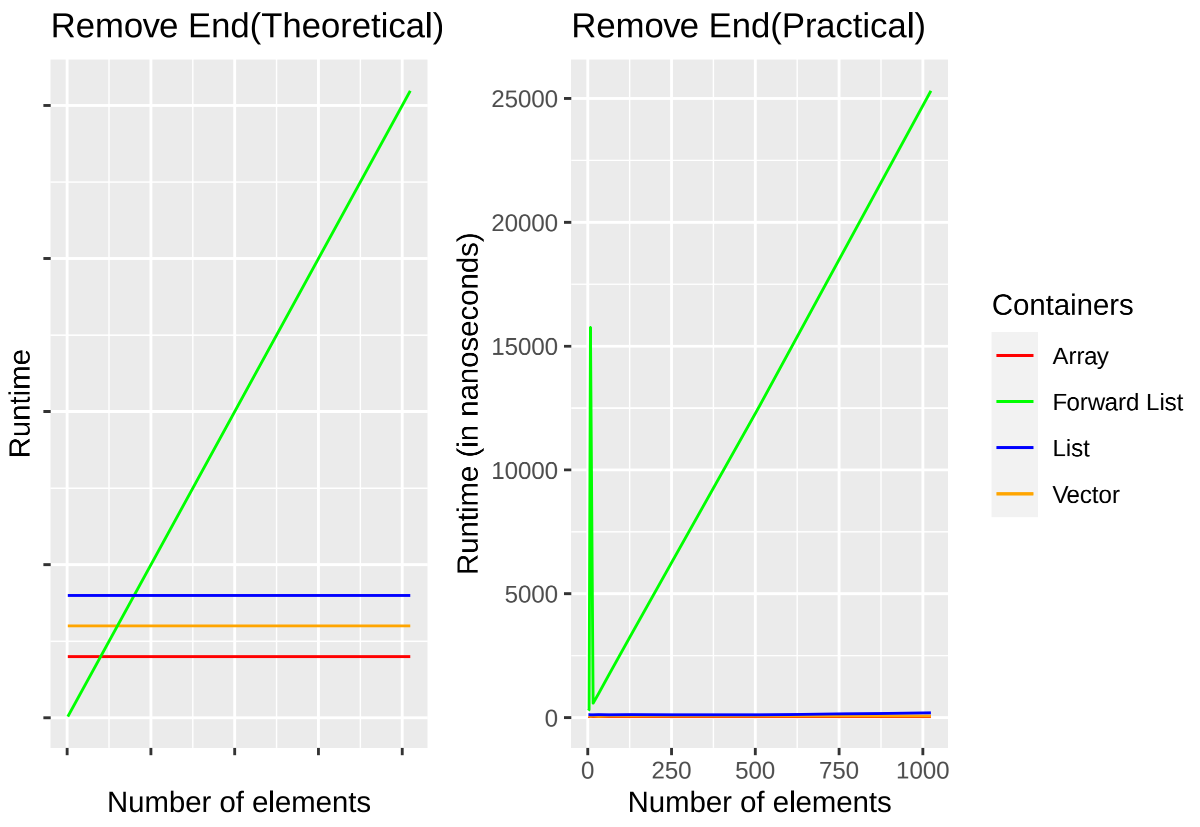
To append the array, we just need to know the size of the elements of the array, then appending will be of constant time as shown in the theoretical graph above. However, if the size of elements has already reached the capacity of the array, then a new array must be created with increased size and all of the elements from the original array need to be copied, including the value to be appended in the new array. This will have linear time complexity.

To append a vector, we can use the push_back function. This is of constant time as shown in the theoretical graph. This holds true as long as the vector has empty space(s) in the end. However, if the vector is already full, then a new vector is automatically created and all the elements including the value to be appended will be added in the end of the new vector that has increased size. This can be clearly seen as sharp rise between 0 and 100 in the line graph of vector. This sudden increase in runtime is due to the creation of new vector and copying of elements to the new vector. If we constantly push elements at the end of the vector, then it might appear that the cost of pushing will be of linear time complexity as shown in the practical line graph for the vector. However, as we deal with much larger number of elements, this will have amortized constant cost as the elements of the vector needs to be allocated less often.
Appending the forward list will be of linear time complexity as shown in the theoretical graph. This is because to add a new node/element at the end of the list, we need to have a pointer point at the last node. For this, we need to advance the pointer all the way from the beginning till it reaches the end node.

Appending the list will be of constant time complexity as shown in the theoretical graph. This is because, unlike forward list, we dont need to advance our iterator from the front till the end. We are already given an iterator that can point to the end node. Its constant time complexity is supported by the practical runtime graph as well.

2. Remove from End

Lets visualize the runtimes for removing elements from the end in the four aforementioned containers.

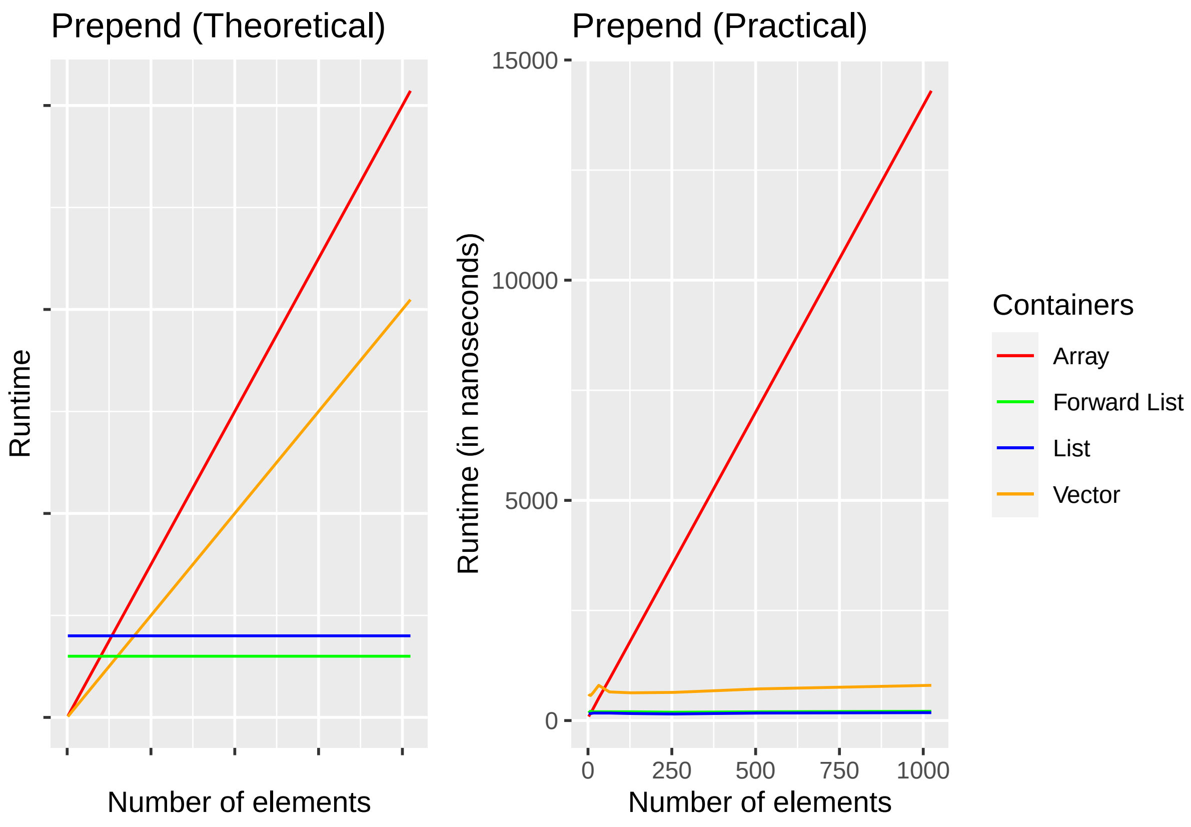


Array, vector, and list have a constant time complexity (the line graph of array is lying underneath that of vector and list) to remove the element from the back. We cannot "remove" the element from the array, however. To mimic removal we assign null value or some other value that is out of the range of the value we have stored (for example, -1 is stored in case we have array which is supposed to store all the positive integers) to the end position. Among the three, list seems to have higher runtime, which can be seen in the practical runtime graph. This might possibly be due to the time spent detructing last the node and assigning pointer to and from second last and first element.

The forward list has linear time complexity as can be seen from the theoretical graph. This is because to remove the last element, we need an iterator that reaches all the way up to the end of the list to point to the last node to delete it. The initial spike in runtime might be due to cache miss when fetching the data from heap. Advancing the iterator needs to be done till the end, hence it is of linear time complexity although insertion itself takes constant time.

3. Prepend

Lets visualize the runtimes for prepending elements in the four aforementioned containers.

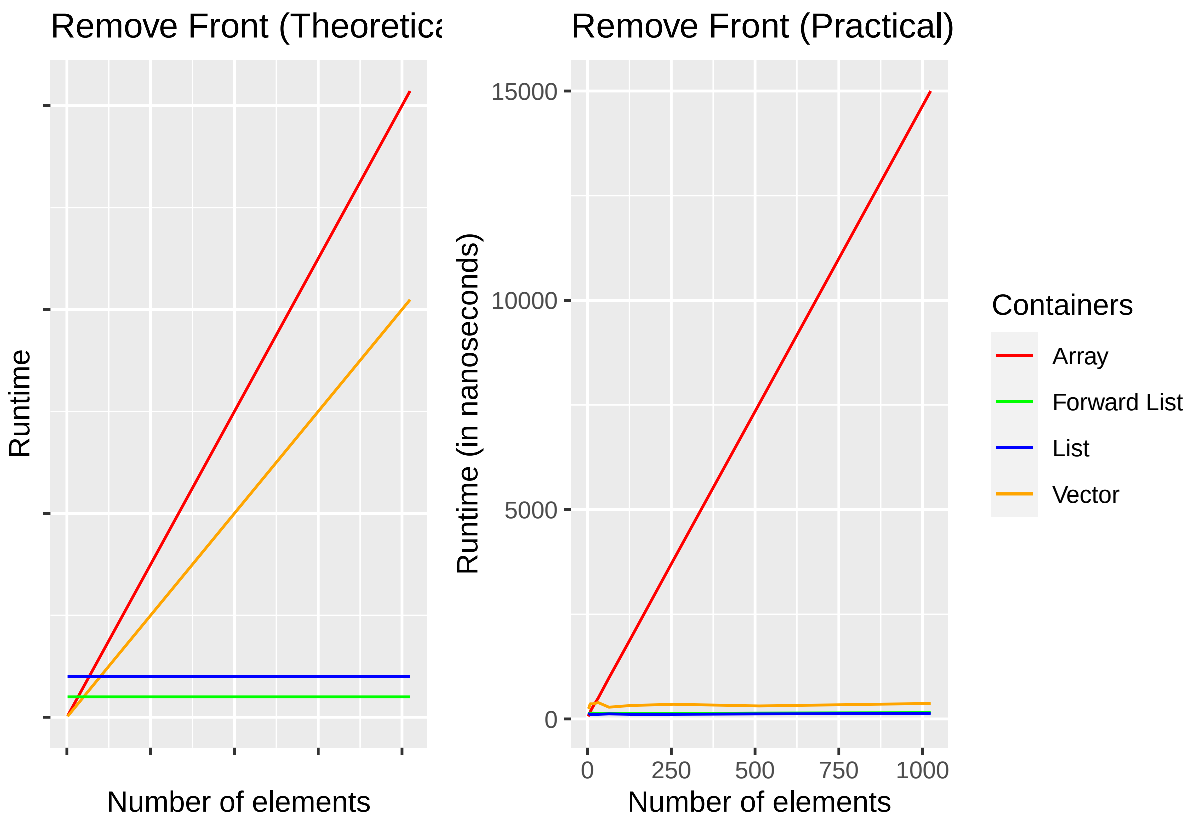


Adding an element to the beginning of the array and vector has a linear time complexity as can be seen from the theoretical graph. It is as such because, when we add a new element at the beginning of the array/vector, we need to shift all the existing elements towards the right by one. If the array doesnt have an empty space, then a new array must be declared and the value to be prepended along with values existing in the initial array must be copied manually. Similarly, if the vector has reached its full capacity, then it will automatically create a new vector and copy the value to be prepended along with the existing values in the initial vector to the new vector. Looking at the practical runtime graph, it appears that vector takes much less time to prepend the elements. This might be because in vectors swap and move operations are instantaneous (copy/swap of three pointers is all it needs) but in array, every single elements must be copied separately. There is a spike in runtime in between value 0 and 1000 which is due to the need to reallocate all the elements of the vector to new one due to insufficient size.

Prepending in forward list and list has a constant time, however. It is as such because, we already have a pointer that can point to the starting position of the list/forward list. All that needs to be done is the insertion of value which has a constant time complexity.

4. Remove from Front

Lets visualize the runtimes for removing elements from the front in the four aforementioned containers.



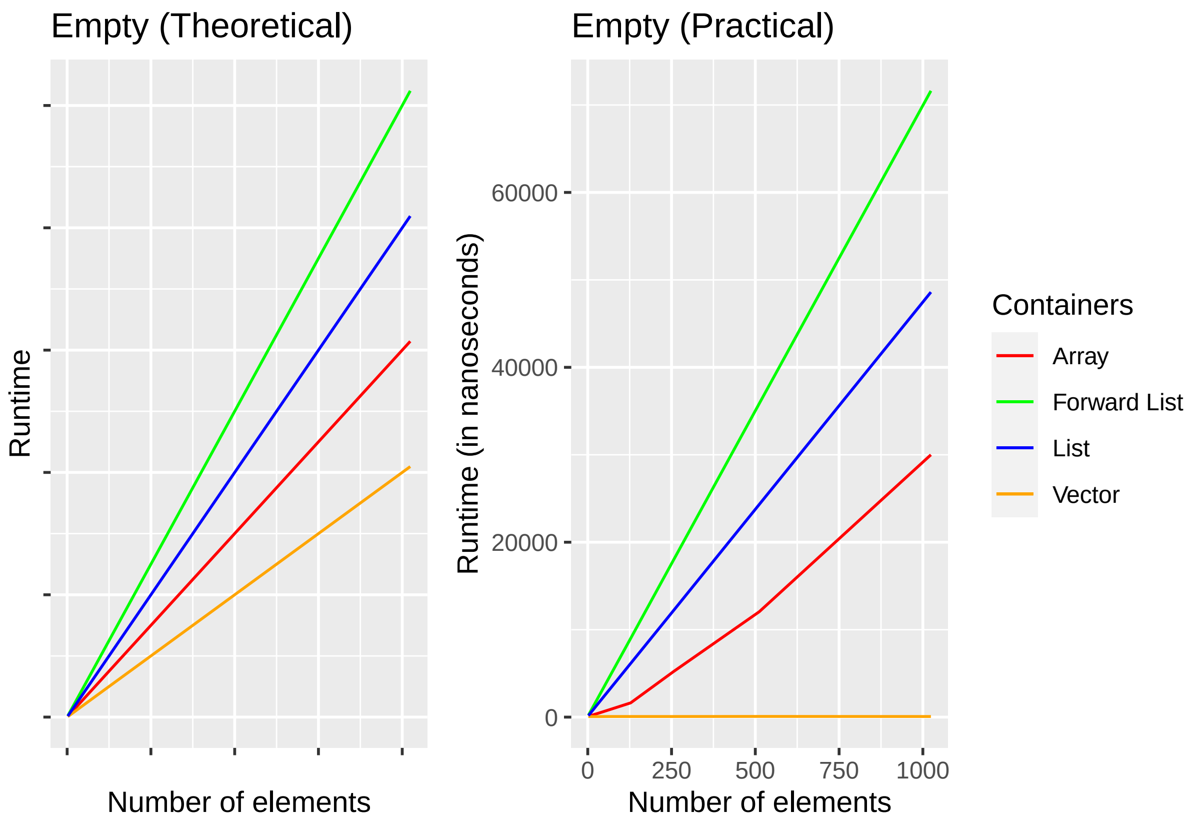
The removal of element from the first position has a linear time complexity of array as when the element in the first position is removed, the subsequent elements need to be shifted to the left/beginning of the array by 1. For this, we need to copy every single value from a given location in the array to its preceding location.

The removal of element from the first position has a linear time complexity for the vector too. Theoretically, we would execute a similar action to what we have performed for the array (Remove the first elements and shift the remaining element by 1). However, in vectors we have a function named erase that can automatically remove any given element, reduce the size of the vector and destroy the removed element. This implementation has much much faster runtime than the one we have performed for the array. One thing must be noted, however. For a smaller number of elements (less than 50), removing from the front of the array is much more efficient than doing so from the vector. Also, when the complexity of the elements inserted in the vector increases, then shifting elements might prove more costly than the array as they might need to call the copy constructors for every single objects stored in the vector when moving them.

Removing from the front of the forward list and list has a constant time complexity as can be seen from the theoretical graph. The theoretical runtime line graphs are supported by the practical runtime line graph because we already have a pointer that points to the front of the list/forward list and destructing the node for removing it from the list takes constant time.

5. Empty

Lets visualize the runtimes for emptying the four aforementioned containers.



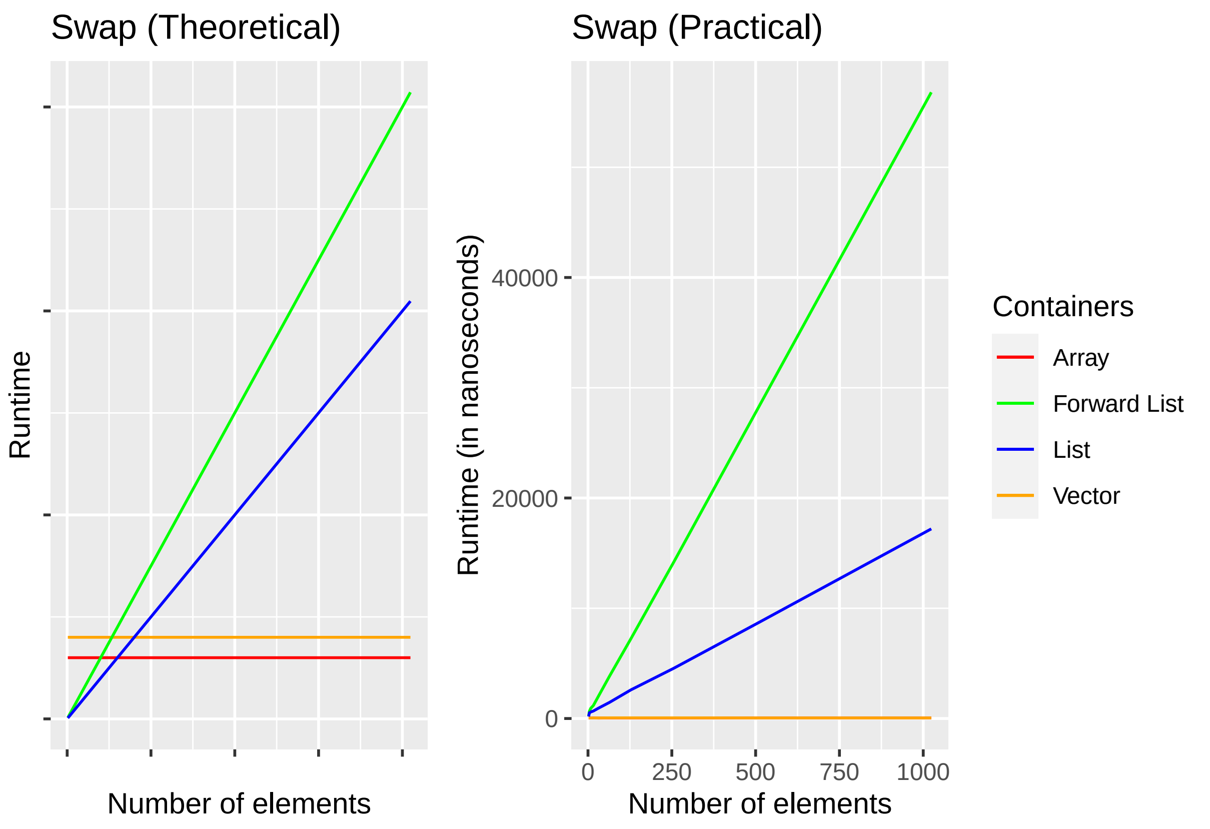
The cost of emptying elements from the array is linear as we need to access every single element and set them to null. This is supported both by our theoretical and practical runtime graph. Similarly, the cost of erasing all the nodes from link and forward list is also linear as each node needs to be accessed at least once and deleted. While, theoretically we might expect the runtime of lists to be higher than that of forward lists due to the overhead of dealing with almost double the pointers, the runtime graphs shows the contrary. It is as such because when we delete elements/node from the forward list one by one, we also need to keep information of the preceding node and to keep track of this we need to move iterators which are costly. As a result, the runtime for erasing all the elements from forward list become more expensive depsite half the number of pointers.

Theoretically, we might assume that the cost of erasing all the elements in a vector is also linear owing to the similar reason given to other containers. However, the clear() function in the vector removes all the elements of the vector in constant time. This is possible as we have stored integers in the vectors and these integers are examples of Plain Old Data (POD) where elements do not need to be destroyed.

It should be noted that array and vector are much faster at erasing the contents as they dont need to deal with pointers and their elements are stored in contiguous manner which can be accessed pretty quickly through the “prefetcher” in the CPU unlike in lists/forward lists where CPU has to jump back and forth to the heap to fetch the memory.

6. Swap

Lets visualize the runtimes for swapping elements/nodes in the four aforementioned containers.



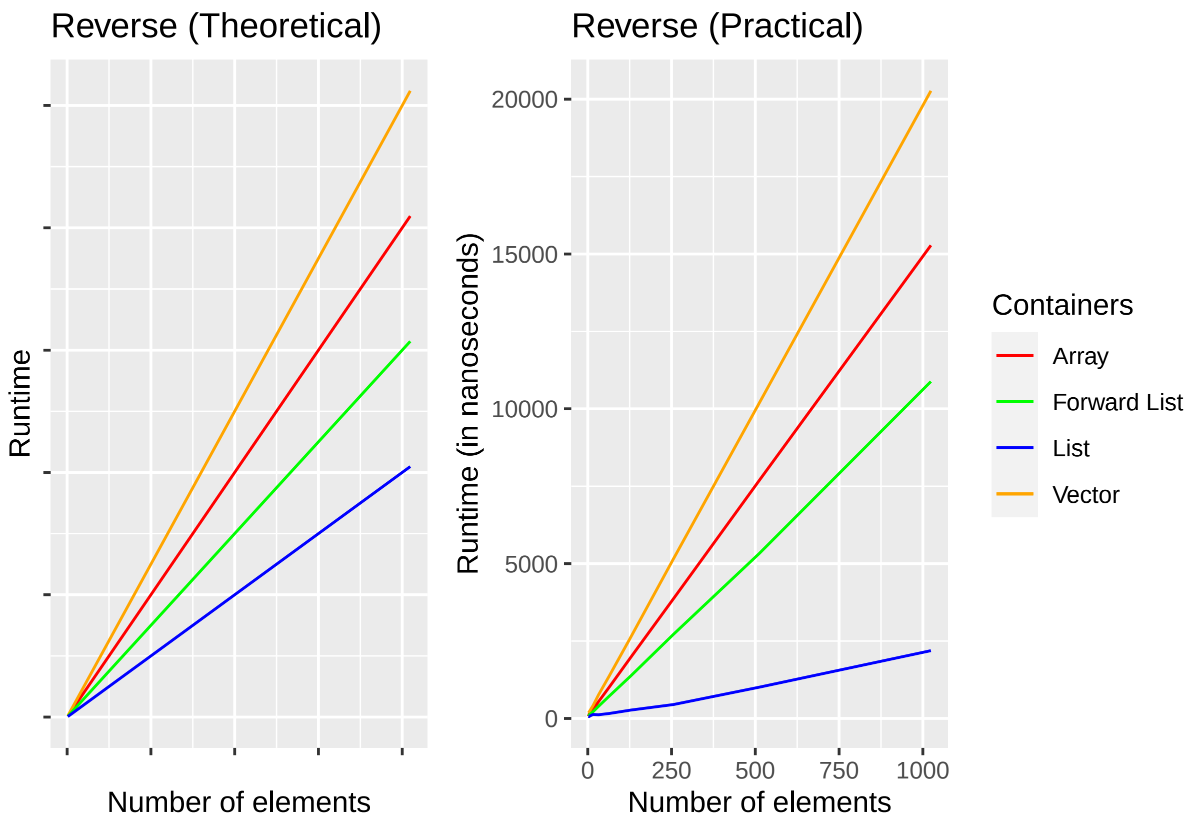
Swapping two elements from the beginning and end has a constant time complexity in both the array and vector as these elements, no matter what their position is, can be accessed quickly with index or offset.

Swapping two nodes in linked list and forward list do not have a constant time complexity, however. The worst case for a forward list occurs when we want to swap the last node with the third last node. We dont swap the last node with the second last node because doing so will essentially be the same as moving just one node and not “swapping”. This has a linear time complexity as we need to advance the iterators till they reach towards the end.

Similarly, the worst case in a list occurs when the nodes in the left and right of the middle position are swapped. We do not choose the last and third last elements in case of lists because they can be easily accessed by the help of reverse iterators. This swapping in the lists also has a linear time complexity, as their iterators need to be advanced from the beginning and end position towards the end. Comparing the time worst case time complexity of list and forward list, its is evident that list is taking almost as half the time taken by forward list. This is because, we are only advancing till the nodes at the half of the list.

7. Reverse

Lets visualize the runtimes for reversing the elements in the four aforementioned containers.



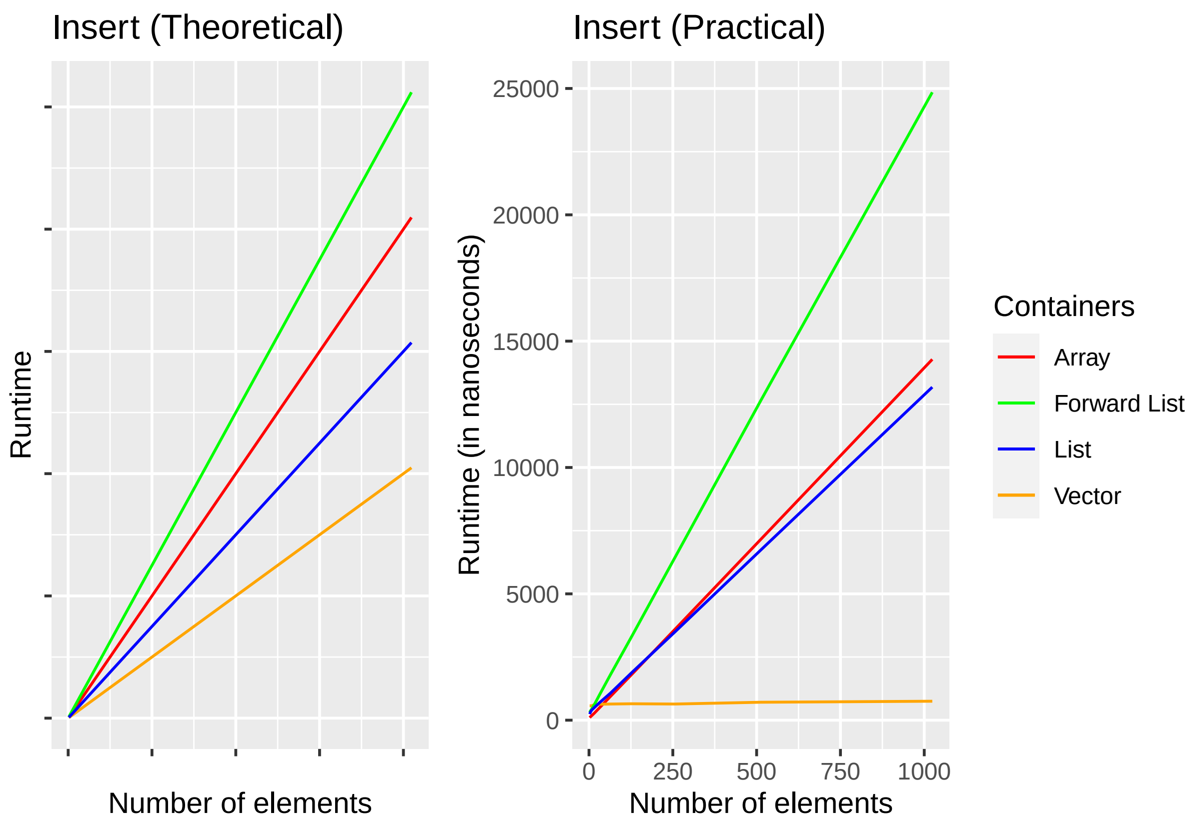
For reversing the values in the array, we simply swapped values from the beginning of the array with the values from the end till our iterators reached the middle position. This has a linear time complexity which is supported by both of our line graphs above.

For reversing the value, we used the reverse() function found in stl library(for the vector) and reverse() function for each of the containers in forward list and list. All of these have a linear time complexity. Since the reverse function for the vector constructs a new vector of same length and copies all the content, it appears to the most costly than performing the same operation for the other three containers.

Performing the reversal in the list appears to be the most runtime efficient which is evident from our practical graph. Despite having more overhead of pointers, list appear to be more runtime efficient. This is becausing while reversing the elements, the lists simply need to swap the values pointed by iterator from the beginning and reverse iterator from the end. However, in the case of forward list, there are not reverse iterators as any node stores the address of the next node only. Hence reversing is more costly in forward list despite having half the number of pointers.

8. Insert

Lets visualize the runtimes for inserting elements in the four aforementioned containers.



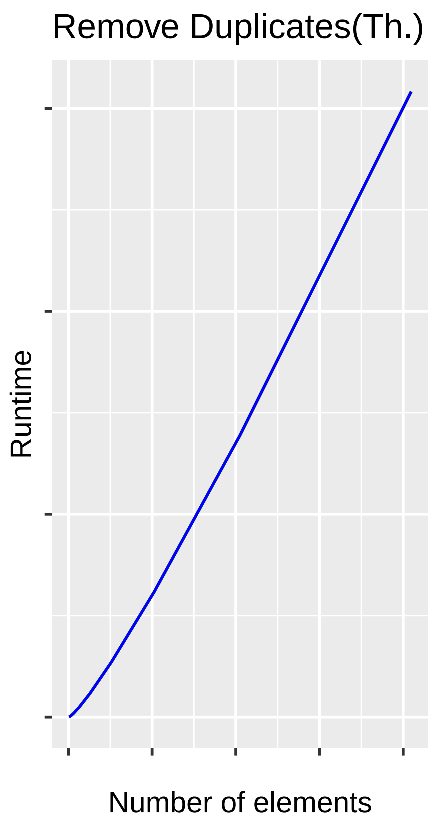
The worst case for array and vector occurs when we insert elements in the beginning. When we insert element at the beginning of these containers, all the existing elements will have to be shifted towards the right by 1. In case of array, if we do not have enough space then we might have to manually create another array and move all the values to the new array. In the case of vector, vector automatically does this for us. Hence the time complexity for both is linear. The runtime line graph for the vector might appear to be constant but it is slightly increasing and its small gradient has been undermined by other line graphs.

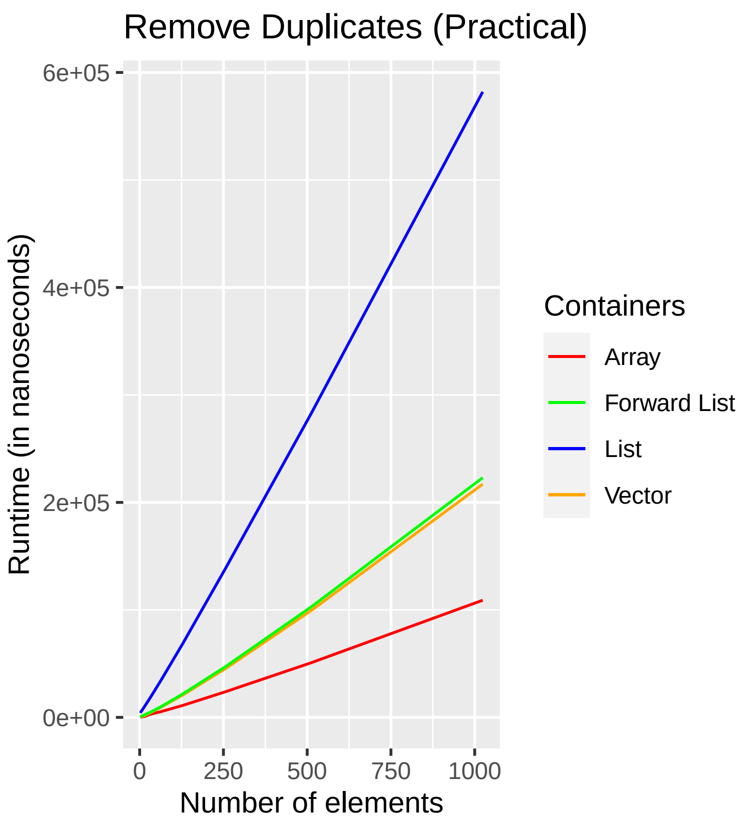
Vector has a considerably less time in inserting in the beginning than the array which might change upon changing the complexity of elements inserted in it. Please note that, when the number of elements are less than 50, inserting element at the beginning of the array is much more runtime efficient than the vector. Inserting elements in the beginning of the list/forwards list has a constant time complexity, however. We must note that, theoretically we might assume inserting elements in the vector in the middle to be less efficient than inserting the element in the middle of forward list or list. However, our practical graph shows that vectors are, quite contrarily, efficient in inserting even in the beginning than inserting elements in the middle in the list. This is because of the data type that we have inserted in the vector. Since copying of integers in the succeeding position when we insert element in the beginning doesnt need to invoke constructors, it becomes much more efficient than traversing the iterators in the list/forward list. However, if the nature of our data type is changed and more complex objects are stored in the vector, than inserting in the beginning/middle will prove to be more expensive than inserting in the beginning/middle of the list/forward list.

The worst case for forward list occurs when we want to insert an element to the end of the list. We need to advance the iterator all the way to the end of the list and this has a linear time complexity. The worst case for list occurs when we want to insert the element in the middle of the list as inserting in the end of the list has constant time complexity for list because of its ability to directly point to the last element. To insert an element in the middle, we need to advance the iterator half the total size of the list. This too has a linear time complexity. Note that the practical runtime line graph of the list is almost as much as half of that of the forward list. This is because, the iterator in the list only has to advance half the value of the number of advances made by iterator in the forward list.

9. Remove Duplicates

Lets visualize the runtimes for removing duplicates from the four aforementioned containers.





For removing duplicates from an array, we sort the elements in the array at first. The we compare each element with its subsequent element and only if they are not same, they are inserted into the array in place. The vacant places in the array are filled with null value. This altogether has an average time complexity of θ(nlogn).

Similarly, to remove the duplicates from the vector we first sort them. This has the average case time complexity of θ(nlogn). Then we use erase function combined with unique function. This has a time complexity of θ(n). We also resize the vector but this has a negligible time complexity in our case as we have only assigned 2 repeated values (We have done this so as to keep the number of repeated values constant over all the size for a given container since the smallest size assigned to any container is 2, the most number of repeated values that we could have made was 2). Overall, the worst case time complexity become θ(n + nlogn) = θ(nlogn). This is supported by both the theoretical and practical runtime graph.

The removal of duplicate values of list and forward list can be implemented first by sorting them at first. This has approximately θ(nlogn) time complexity on average. Then we use the unique() function in the list/forward list which has the linear time complexity. The total average time complexity for removing duplicates in the list/forward list become θ(n + nlogn) = θ(nlogn) where n is the number of nodes/elements. This is supported by both the theoretical and practical runtime graph.(Please note, all the containers have nlogn average time complexity hence their theoretical graphs are the same.)

Comparing the practical values, array appears to be the most runtime efficient and list appear to be the most expensive in terms of time. The runtimes for vectors and forward list are similar and lie in between the most expensive and least expensive runtimes. List might have taken the highest runtime since our data were stored in reverse order and when executing sort function, it might have had to move essentially all the nodes and change the value of almost all the pointers. The vector might have taken much larger value than that of array because of the difference in sorting algorithm used, and need to resize it after the removal of duplicates.

Conclusion

If you have small number of elements(less than 50) of fixed size and you want random access to these elements without insertions or removals, then choosing std::array might be a good option. Note, however, that you will need to predefine your own functions for several actions to be performed. For instances, for removing duplicates, reversing„ etc std::array doesn't have built in functions. If you want random access and have large number of elements/objects of non-conventional type, you dont know the size of the elements and only prefer insertions at the end (not at the beginning or at random), then std:: vector might be the best option. Better yet, if you know the tentative size of the proposed vector use resize in the beginning to avoid reallocations while assigning values to the vector. For cases like swapping values and erasing elements again std::vectors are the most runtime efficient option. Even in the cases like removing from the front and prepending std::vector outperforms the std::array. If random access is not desired but insertions in the middle or beginning is to be done and elements are of complex data types, then std::list or std::forward list can be chosen. Since std::list has more number of built-in functions, choosing it over forward list might be convenient if space is not the concern. Note, however, that both std::list and std::forward list have overhead because of the pointers they need to store so only use them where std::vectors and std::arrays fail to perform efficiently in case of runtime (for insertion in the beginning and in random positions) and space is not the concern. Lastly, if your system uses cache then prefer vectors over std::list/forward list as std::list/forward list are more prone to cache miss. Overall, std::vector seems to be suited to perform multiple functions efficiently and conveniently hence choose it as your default container. As for specific tasks, choose the containers based on the insights given above.