Josh Ibad
CPSC 131-05; Spring 2020

Analysis Report
Container Review and Analysis Homework.pdf

An analysis program measuring the time it took to execute insertion, removal, and search operations, was performed on multiple C++ Standard Library Data Structures, namely, on the Binary Search Tree (std::map), Hash Table (std::unordered_map), Doubly Linked List (std::list), Singly Linked List (std::forward_list), and Vector (std::vector). Some of these operations were performed on both front and back ends to truly find the potential worst case scenario in time complexity. The program used GroceryItem objects as the elements and time was measured with various numbers of elements or sizes in the data structures under test. After performing these tests, the data was compiled into a TSV file, which was plotted on google sheets by operation type. The plots are depicted three times below, with various ranges so as to allow the viewer to actually see all results, as the differences often are too great between the worst case scenario and the best case scenario as to render some of the data unreadable, without the replotting in different intervals. Within this analysis report, to get the full picture, the time complexity of each operation shall firstly be mentioned by data structure. Afterwards, the time complexity shall be looked at comparatively between data structures, by operation, to give insight as to how data structures within the same time complexity classes perform against each other. The results of the analysis are as follows:

For the Binary Search Tree (std::map), both the insertion and removal operations evidently follow a logarithmic path, with the increase in y slowing down as x grows. Hence, for the BST, insertion is of the time complexity class O(log n) and removal operation is of the time complexity class O(log n). The search however is very obscure, most likely because the average case grows too slowly and because the average case is the most occurent scenario. Merely from the data, the search operation looks like it is a constant O(1), however it is worth noting that theoretically, the search operation of the BST is truly O(log n).

As for the Hash Table (std::unordered_map), all throughout the test, within all operations, insertion, removal, and search, the hash table plots remained fairly constant in the very bottom of the charts, staying very close to the x axis. Yet it is also worth noting that though it remains fairly constant, there appears to be a very small increase in run time throughout the data, for the insertion and removal operations, especially towards the end, possibly since the hash table data structure does at times depend on the load ratio which is affected at time by the size. Regardless of this note, the growth is small, especially compared to the others, that it is negligible, hence we shall make the conclusion that for the Hash Table, insertion is of the O(1) time complexity, removal is O(1), and search is O(1).

In the Doubly Linked List data structure (implemented as std::list), the insertion and removal operations are fairly obscure and pattern, however since they fall in the very bottom close to the x axis, with the variation being negligible, it is fair to say the insertion and removal operations, both for the front and the back, prove to be constant in time complexity. As for the search operation, the data plot clearly follows a linear pattern, being better than SLL's linear search but worse than Vector's linear search, as will be detailed later. Hence we conclude for DLL, insertion for front and back is O(1), removal for front and back is O(1), and search is O(n).

For the Singly Linked List data structure (implemented as std::forward_list), the insertion at front operation clearly follows a constant time complexity, appearing to be among the best, while the removal at front is obscure, but with negligible variation staying close to the x-axis so as to warrant being considered as a constant operation. However, for the insertion at back, the structure appears to follow a constant pattern near the bottom, while removal at back follows a linear pattern that proves worse than all removal operations. As for the search, SLL has the worst time complexity, following a linear pattern growing steeper than the others. To summarize for the SLL, insertion at front and back appears to be O(1), while for removal, removal at front is O(1) while removal at back is O(n). Search is O(n).

Finally, for the Vector (std::vector), insertion at front is clearly constant, worse than SLL insertion at front but at times better than Hash insertion. Insertion at back however is the worst, following a linear pattern unlike all others in the test. As for removal, removal at front follows a linear pattern performing better than SLL removal at back, while removal at back for the vector performs the best out of all, as the quickest constant time complexity. For search, it follows a linear pattern in the best of its complexity class. Thus, vector insertion at front is O(n), insertion at back is O(1), removal at front is O(n), removal at back is O(1), and search is O(n).

To summarize, according to the findings of the experimental analysis:

	Insertion (front, back)	Removal (front, back)	Search
BST	O(log n)	O(log n)	O(log n)
Hash	~O(1)	~O(1)	O(1)
DLL	{F,B} = { O(1), O(1) }	{F,B} = { O(1), O(1) }	O(n)
SLL	{F,B} = { O(1), O(n) }	{F,B} = { O(1), O(1) }	O(n)
Vector	F = O(n) B = {Best & Ave Case: O(1), Worst Case: O(n) }	{F,B} = { O(n), O(1) }	O(n)

Now, complexity class classification is a great tool, but within the same complexity classes, Big-O notation buries the differences, hence the following will now list them comparatively experimentally:

Insertion:

For insertion, vector at front is obviously the worst with the only O(n) runtime. BST follows as the next worst with O(log n), compared to the other's O(1) complexity. Within the constant time class, SLL appears to perform the best overall, with vector at back following it, succeeded by Hash, then DLL. In absolute terms, vector at front performs worst while SLL performs best in insertion.

Insertions are required when we maintain a short list of items which is to be printed. In such a case, insertions are required, while removal and search are not. Sequential direct access may also be required. Hence, in such a scenario, SLL would be used, as it rapidly inserts all items, and in the end, a single traversal would be performed in printing it's contents, rendering the drawbacks in lack of backward accessing and lack of direct access useless. With only insertion as the operation in use, all the other data structures were not chosen since they do not perform as well as SLL in insertion.

Removal:

In removal, both SLL at back and Vector at back classify as O(n) linear complexity, but SLL at proves the worst overall, with vector following it in place of worst at removal complexity. Following those, is BST removal with it's logarithmic complexity of O(log n), beaten by the constant complexity of which, obscurely, Vector at back appears the best overall, followed by Hash, then by SLL at front, then by DLL. In absolute terms, SSL at back performs worst, and Vector at back appears best in removal.

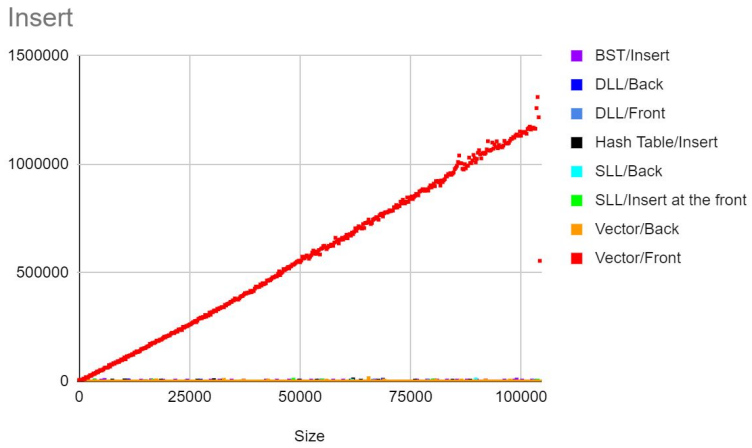
Removal is required in a queue such as one that tracks a line of people's requests or orders or calls. In such a case, not only removal is required, but also insertion, while search and direct access is unnecessary. Hence, a doubly linked list proves best in this use case, as it performs decently in both insertion and removal and is decently efficient with its space. SLL and Vector prove most inefficient in at least of the necessary operations which is why they were not picked, and ordering is not apparent rendering BST's improper in this use case. Hash tables would do great in runtime but the space complexity seems an unnecessary cost for such a use case.

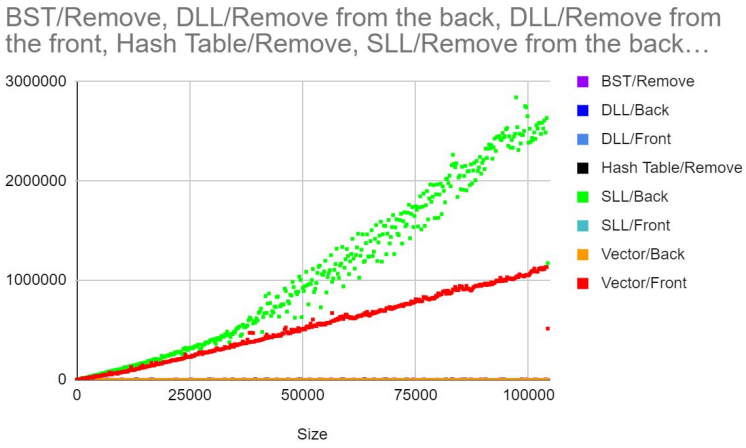
Search:

In search, SLL, DLL, and Vector all classify as linear O(n) complexity. However, SLL proves to be the overall worst, followed by DLL, then Vector. Far below in the bottom, Hash proves to be the best overall with constant O(1) complexity, with BST following close behind it in an obscure pattern that seems constant but is theoretically logarithmic. In absolute terms, SSL proves worst in search operations while Hash proves best.

Searching is required when maintaining a database of items in an online store or inventory of a store. Such a case requires constant insertions, removal, and often access to modify information about its items. In such a case, a Hash Table may prove the best for time complexity, since it will rapidly insert new items with constant time, remove old items in constant time, and access a desired item in constant time due to constant time searching.

Collectively, it appears that Hash tables (std::unordered_map) proves the best in time complexity overall maintaining a theoretically, and practically constant complexity. Though often best, it sometimes costs more space depending on implementation and the use case. Conversely, the worst would have to be either the vector or the SLL.





Josh Ibad
CPSC 131-05; Spring 2020