John Ahlgren

A blog mostly about mathematics, computer science, and programming.

Sunday, October 13, 2013

STL Container Performance

STL Container Performance Table

There's already a good table at Stack Overflow listing time complexity (in Big O notation) of common operations with C++ STL containers for comparison, although it's structured in a more abstract way and a little hard to read because it's not a HTML table. Here's my version, which also includes priority_queue (technically an adaptor).

Persistent iterators means that the iterators are not invalidated by insertions and erase (except when erasing the element referred to by the iterator, which is necessarily invalidated).

Container	Insertion	Access	Erase	Find	Persistent Iterators
vector / string	Back: O(1) or O(n) Other: O(n)	O(1)	Back: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	No
deque	Back/Front: O(1) Other: O(n)	O(1)	Back/Front: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	Pointers only
list / forward_list	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	O(n)	Yes
set / map	O(log n)	-	O(log n)	O(log n)	Yes
unordered_set / unordered_map	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	Pointers only
priority_queue	O(log n)	O(1)	O(log n)	-	-

Always O(1): begin(), end(), empty(), size(), push_back()

The following operations are always O(1) when they exist:

1. begin(), end()
2. empty()
3. size() (note that list::size() was not necessarily O(1) prior to C++11)
4. push_front() (note that std::vector does not have push_front(), as it would not be O(1))
5. push_back()

Some Additional Notes

Adaptors: queue and stack

For std::queue and std::stack, complexity depends on the underlying container used (by default std::deque).

vector

std::vector has constant time (O(1)) back insertion provided no reallocation needs to take place (use reserve/capacity to allocate/check). When reallocation is necessary, all elements are copied (or moved, if possible) to a need memory location. It is guaranteed that back insertion is amortized constant, meaning: "if we perform a large amount of back insertions, the average time for back insertion is constant".

Insertion does not invalidate iterators as long as no reallocation is performed (when reallocating, all iterators become invalid). Deletion invalidates all iterators after the deleted element, iterators to elements before are still valid.

deque

Insertion and deletion of elements in a std::deque may invalidate all its iterators. Pointers are however persistent. In practice accessing / iterating over a std::vector is faster than std::deque.

All iterators may become invalid after an insertion or deletion, but pointers/references are always valid.

list

If you have an iterator to an element, you can insert right after (or before) that element in constant time (O(1)). Of course, you can also erase it or access it directly (O(1)) using the iterator (or any adjacent element, as ++iterator / --iterator are constant time operations).

If you only know the index, e.g. that you wish to insert/retrieve/erase the 4th element, you'll need to iterate the list until you reach that element. Put differently: std::list does not provide random access.

sorted vector and deque

To search for an element in a sorted std::vector or std::deque, use std::equal_range. If only the first element is needed, there is std::lower_bound. If you only want to know whether an element exists or not, there is std::binary_search.

set and map

Requires a less-than comparison function. Complexities also apply to multiset and multimap.

unordered_set and unordered_map (hash tables)

unordered_set and unordered_map has constant time performance on all operations provided no collisions occur. When collisions occur, traversal of a linked list containing all elements of the same bucket (those that hash to the same value) is necessary, and in the worst case, there is only one bucket; hence O(n).

Requires a hash function and equality comparison function. Complexities also apply to unordered_multiset and unordered_multimap.

Deletion does not invalidate any iterators (other than erased element). Insertion keeps all iterators valid as long as no rehashing is done. When rehashing is performed, all iterators become invalid. Pointers/references to elements always remain valid.

multiset, multimap, unordered_multiset, unordered_multimap

std::multiset and std::multimap follow the same rules as std::set and std::map. std::unordered_multiset and std::unordered_multimap follow the same rules as std::unordered_set and std::unordered_map.

The only reason they are not listed in the table/throughout this document is to save space.

basic_string

Strictly speaking std::string and std::wstring are typedefs for basic_string, which is a container. What applies to string above applies more generally to basic_string (and hence, to std::wstring too).

Note that prior to C++11 basic_string was not required to store its elements (characters) contiguously. Now it acts as std::vector, except its only valid for POD types and some tricks that don't violate the constraints may be employed (in practice: small string optimization).

priority_queue

Requires a less-than comparison function. Always gives the greatest element (according to comparison function) when top() is called. top() is constant time, but pop() requires O(log n) as the queue needs to be rearranged (to ensure next top() correctly gives greatest element).

Posted by Unknown at 2:35 PM

Labels: C plus plus (Code), code efficiency

7 comments:

Anonymous October 14, 2013 at 8:38 AM

BTW, unordered_map/set do not have 'persistent' iterators. References/pointers are always valid, however.

Reply