

C++ benchmark – std::vector VS std::list VS std::deque

Baptiste Wicht — 2012-12-03 08:58 — 44 Comments

Last week, I wrote a benchmark comparing the performance of std::vector and std::list on different workloads. This previous article received a lot of comments and several suggestions to improve it. The present article is an improvement over the previous article.

In this article, I will compare the performance of std::vector, std::list and std::deque on several different workloads and with different data types. In this article, when I talk about a list refers to std::list, a vector refers to std::vector and deque to std::deque.

It is generally said that a list should be used when random insert and remove will be performed (performed in $O(1)$ versus $O(n)$ for a vector or a deque). If we look only at the complexity, the scale of linear search in both data structures should be equivalent, complexity being in $O(n)$. When random insert/replace operations are performed on a vector or a deque, all the subsequent data needs to be moved and so each element will be copied. That is why the size of the data type is an important factor when comparing those two data structures. Because the size of the data type will play an important role on the cost of copying an element.

However, in practice, there is a huge difference: the usage of the memory caches. All the data in a vector is contiguous where the std::list allocates separately memory for each element. How does that change the results in practice ? The deque is a data structure aiming at having the advantages of both data structures without their drawbacks, we will see how it perform in practice. Complexity analysis does not take the memory hierarchy into level. I believe that in practice, memory hierarchy usage is as important as complexity analysis.

Keep in mind that all the tests performed are made on vector, list and deque even if other data structures could be better suited to the given workload.

In the graphs and in the text, n is used to refer to the number of elements of the collection.

All the tests performed have been performed on an Intel Core i7 Q 820 @ 1.73GHz. The code has been compiled in 64 bits with GCC 4.7.2 with -O2 and -march=native. The code has been compiled with C++11 support (-std=c++11).

For each graph, the vertical axis represent the amount of time necessary to perform the operations, so the lower values are the better. The horizontal axis is always the number of elements of the collection. For some graph, the logarithmic scale could be clearer, a button is available after each graph to change the vertical scale to a logarithmic scale.

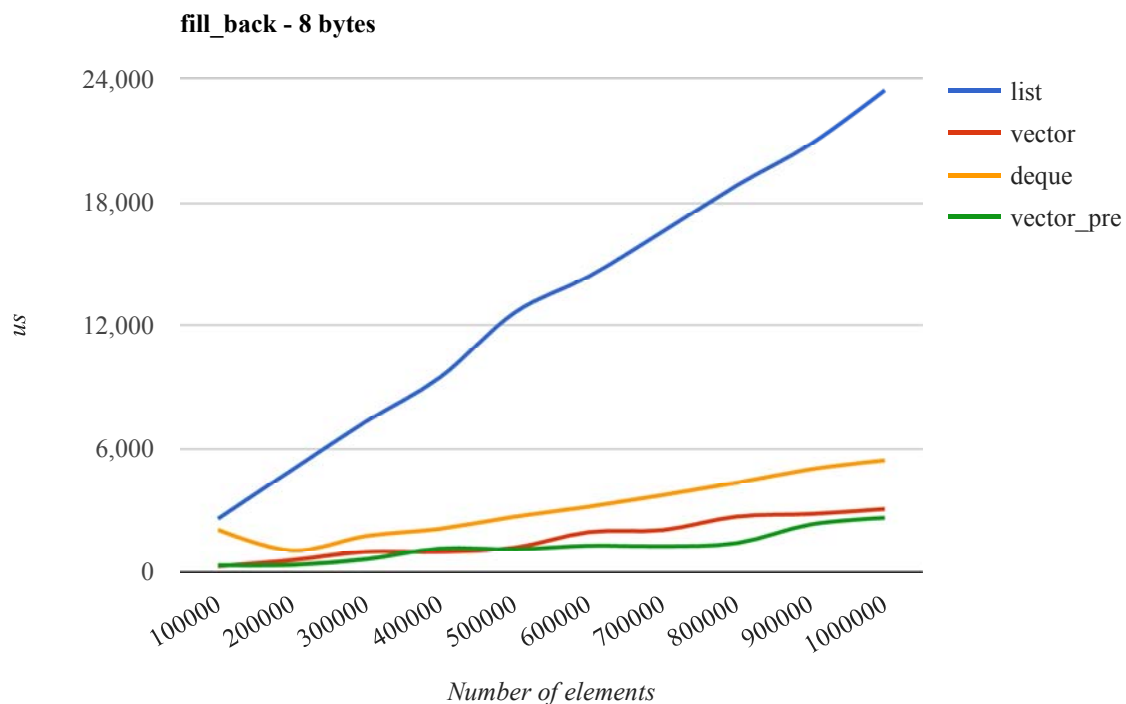
The data types are varying in size, they hold an array of longs and the size of the array varies to change the size of the data type. The non-trivial data type is made of two longs and has very stupid assignment operator and copy constructor that just does some maths (totally meaningless

but costly). One may argue that is not a common copy constructor neither a common assignment operator and one will be right, however, the important point here is that it is costly operators which is enough for this benchmark.

Fill

The first test that is performed is to fill the data structures by adding elements to the back of the container (using *push_back*). Two variations of vector are used, *vector_pre* being a `std::vector` using `vector::reserve` at the beginning, resulting in only one allocation of memory.

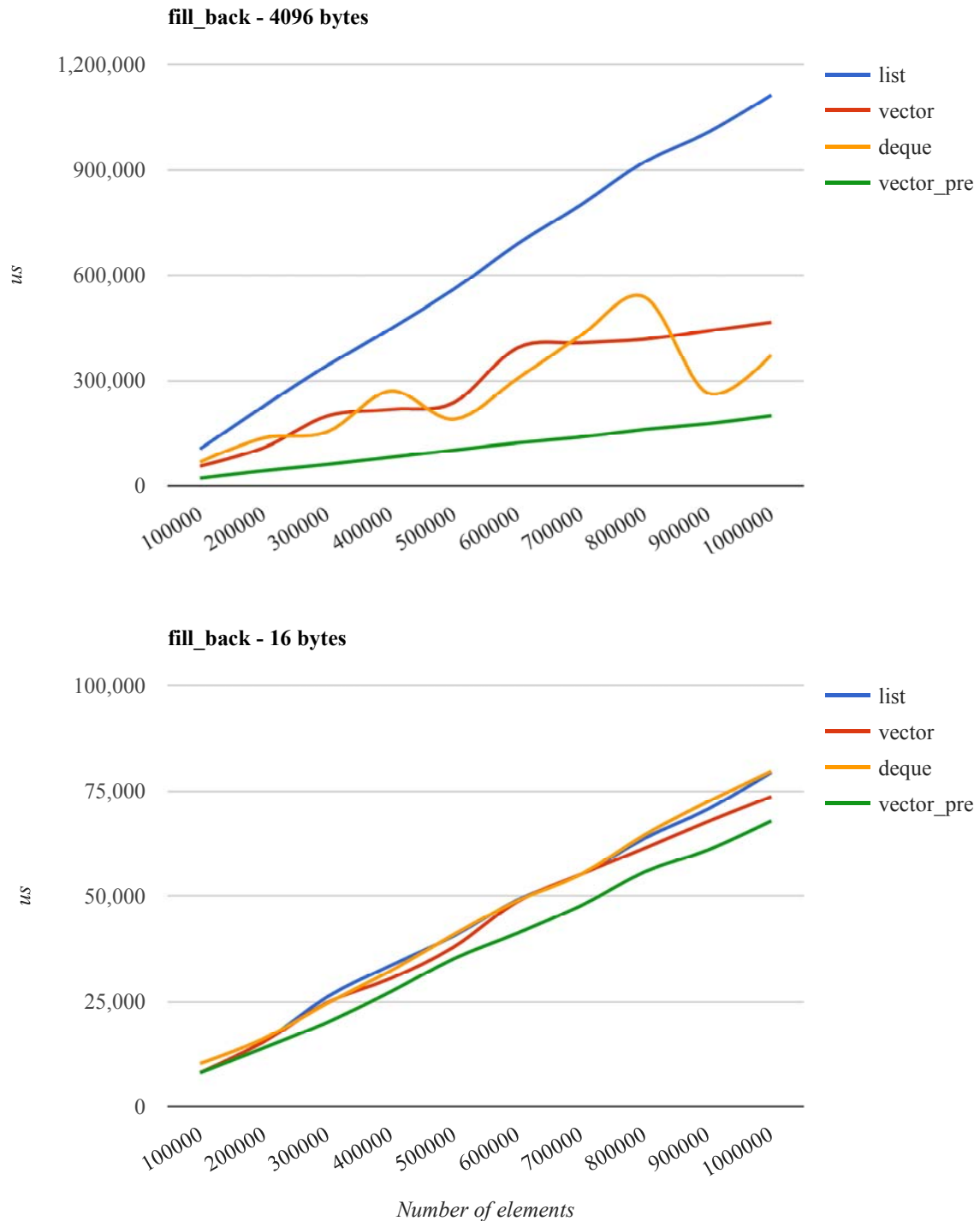
Lets see the results with a very small data type:



Logarithmic scale

The pre-allocated vector is the fastest by a small margin and the list is 3 times slower than a vector. deque and vector.

If we consider higher data type:



Logarithmic scale

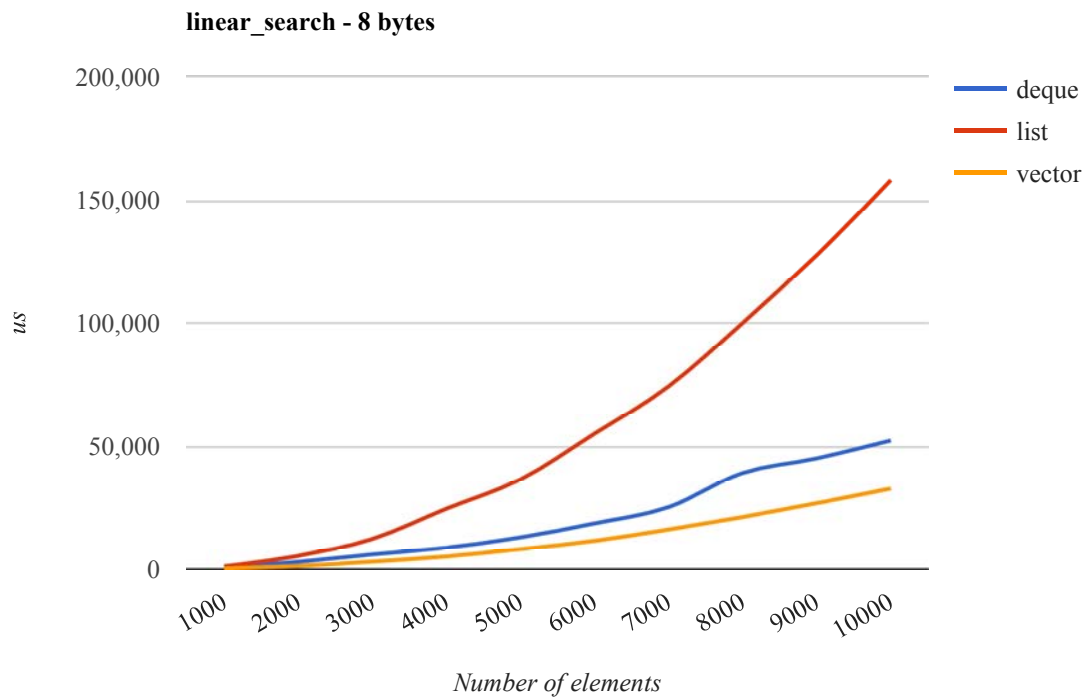
All data structures are performing more or less the same, with `vector_pre` being the fastest.

For `push_back` operations, pre-allocated vectors is a very good choice if the size is known in advance. The others performs more or less the same.

I would have expected a better result for pre-allocated vector. If someone find an explanation for such a small margin, I'm interested.

Linear Search

The first operation that is tested is the search. The container is filled with all the numbers in $[0, N]$ and shuffled. Then, each number in $[0, N]$ is searched in the container with `std::find` that performs a simple linear search. In theory, all the data structures should perform the same if we consider their complexity.



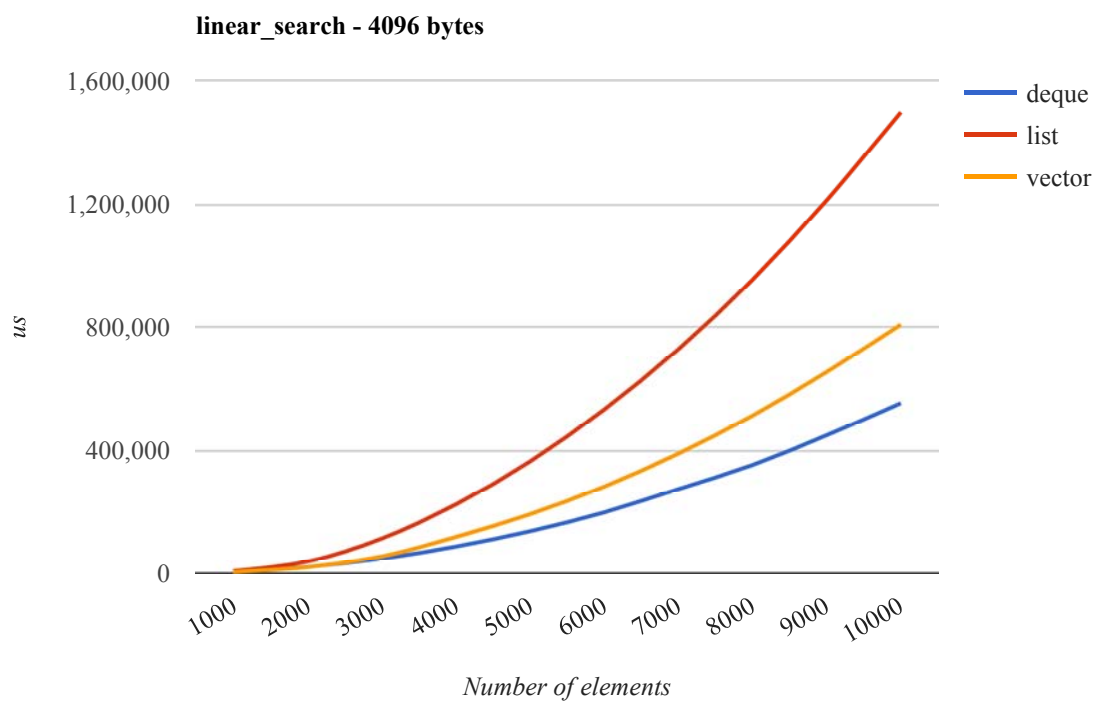
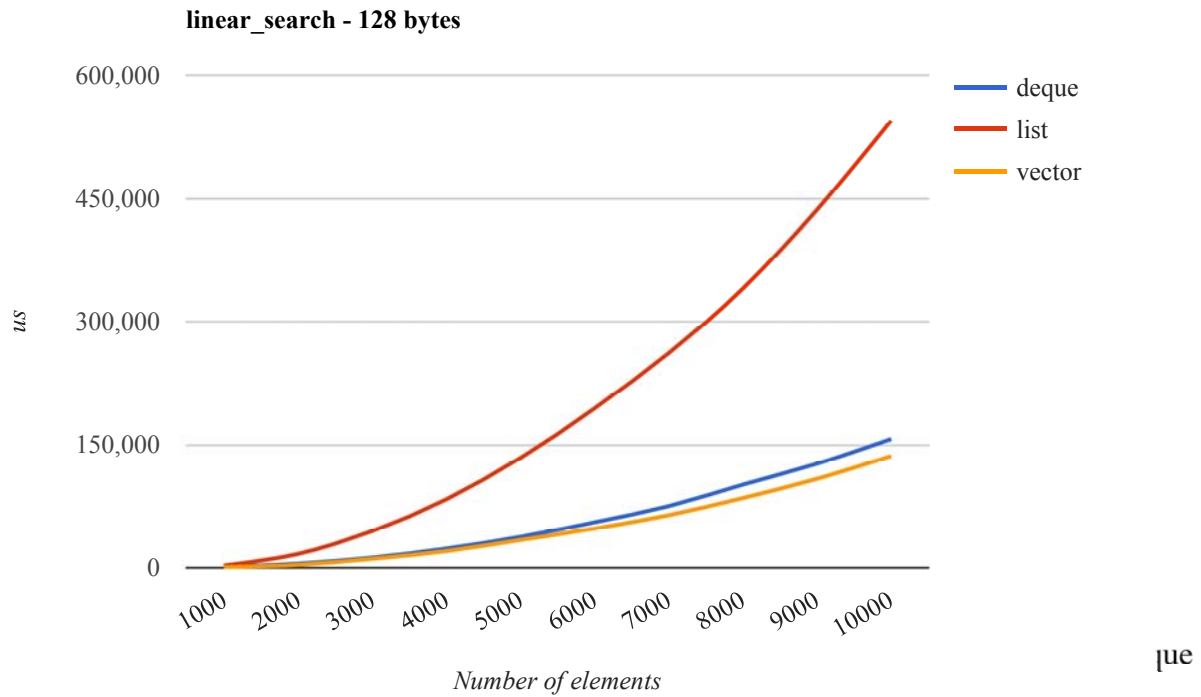
Logarithmic scale

It is clear from the graph that the list has very poor performance for searching. The growth is much worse for a list than for a vector or a deque.

The only reason is the usage of the cache line. When a data is accessed, the data is fetched from the main memory to the cache. Not only the accessed data is accessed, but a whole cacheline is fetched. As the elements in a vector are contiguous, when you access an element, the next element is automatically in the cache. As the main memory is orders of magnitude slower than the cache, this makes a huge difference. In the list case, the processor spends its whole time waiting for data being fetched from memory to the cache, at each fetch, the processor fetches a lot of unnecessary data that are almost always useless.

The deque is a bit slower than the vector, that is logical because here there are more cache misses due to the segmented parts.

If we take a bigger data type:



Logarithmic scale

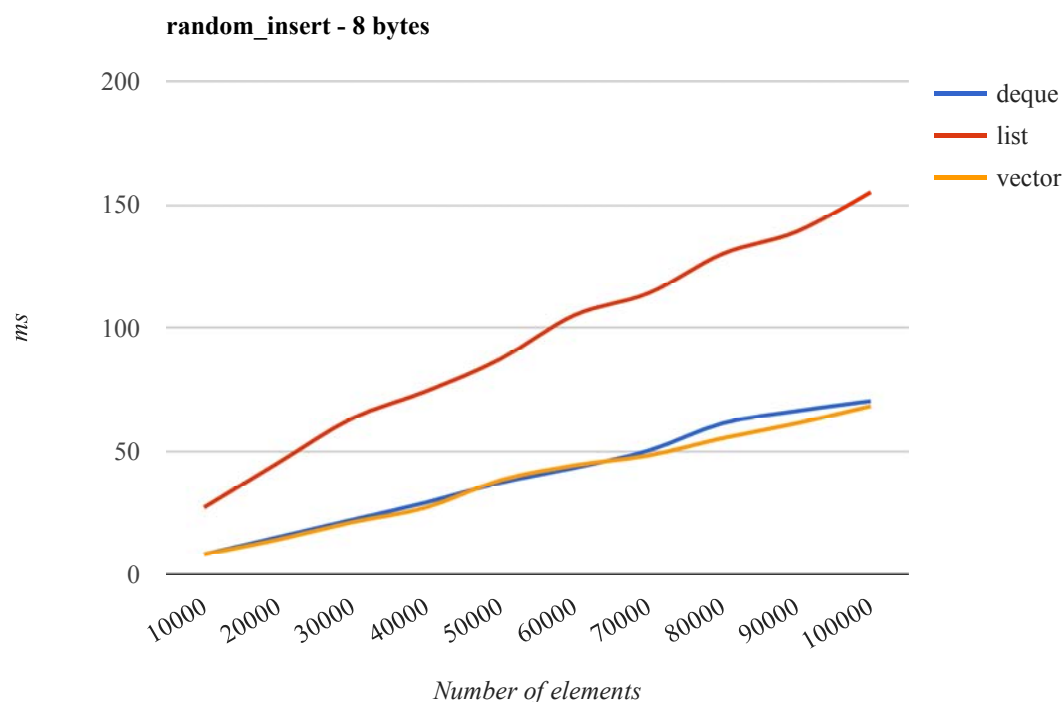
The performance of the list are still poor but the gap is decreasing. The interesting point is that deque is now faster than vector. I'm not really sure of the reason of this result. It is possible that it comes only from this special size. One thing is sure, the bigger the data size, the more cache misses the processor will get because elements don't fit in cache lines.

For search, list is clearly slow where deque and vector have about the same performance. It seems that deque is faster than a vector for very large data sizes.

Random Insert (+Linear Search)

In the case of random insert, in theory, the list should be much faster, its insert operation being in $O(1)$ versus $O(n)$ for a vector or a deque.

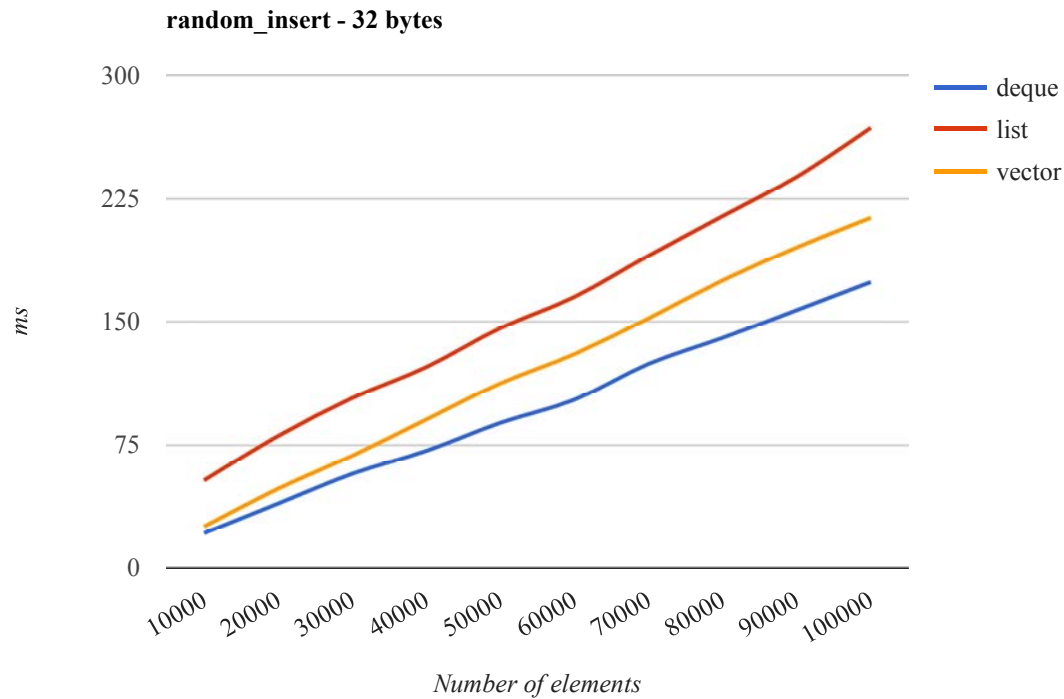
The container is filled with all the numbers in $[0, N]$ and shuffled. Then, 1000 random values are inserted at a random position in the container. The random position is found by linear search. In both cases, the complexity of the search is $O(n)$, the only difference comes from the insert that follow the search. We saw before that the performance of the list were poor for searching, so we'll see if the fast insertion can compensate the slow search.



Logarithmic scale

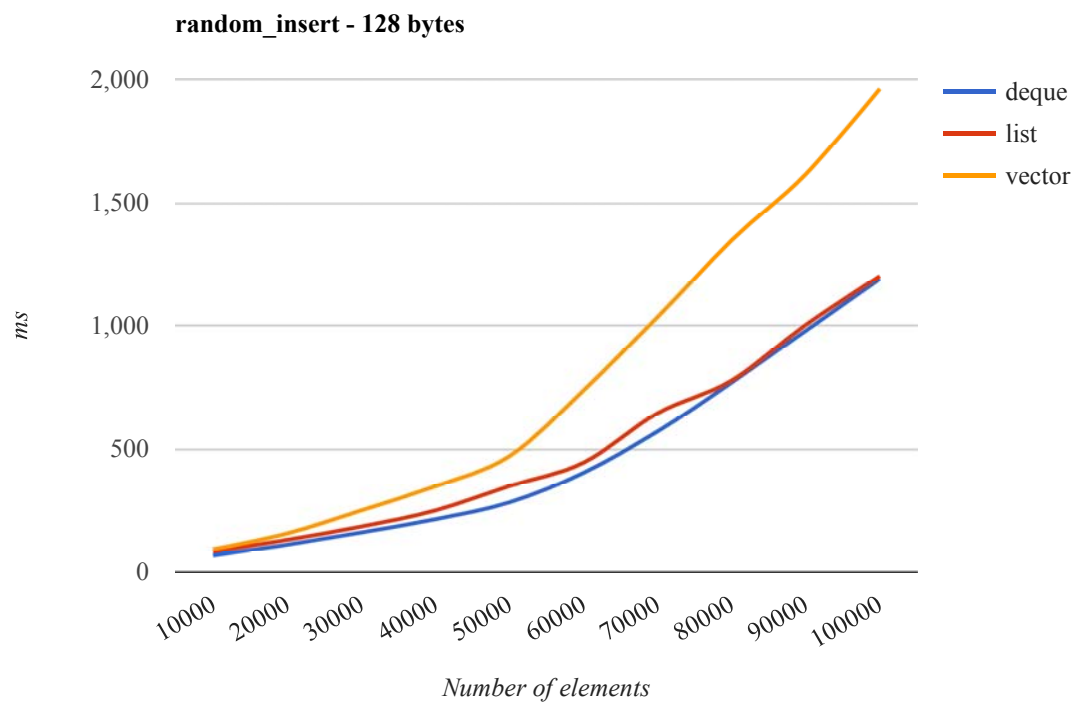
List is clearly slower than the other two data structures that exhibit the same performance. This comes from the very slow linear search. Even if the two other data structures have to move a lot of data, the copy is cheap for small data types.

Let's increase the size a bit:



The result are interesting. The list is still the slowest but with a smaller margin. This time deque is faster than the vector by a small margin.

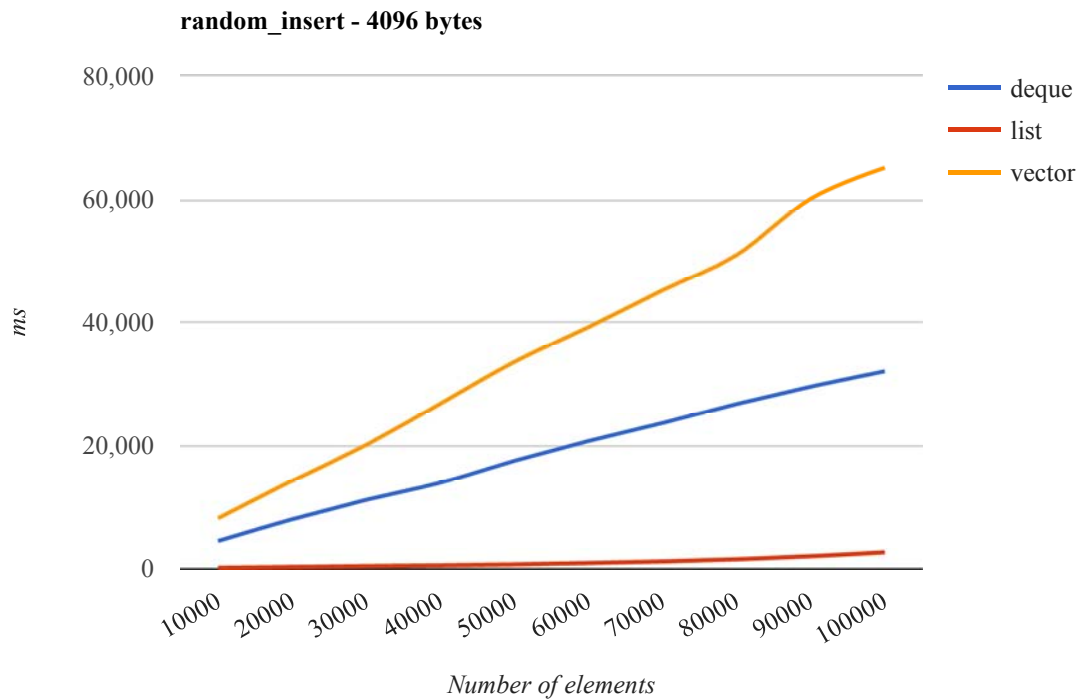
Again, increasing the data size:



Logarithmic scale

This time, the vector is clearly the looser and deque and list have the same performance. We can say that with a size of 128 bytes, the time to move a lot of the elements is more expensive than searching in the list.

A huge data type gives us clearer results:

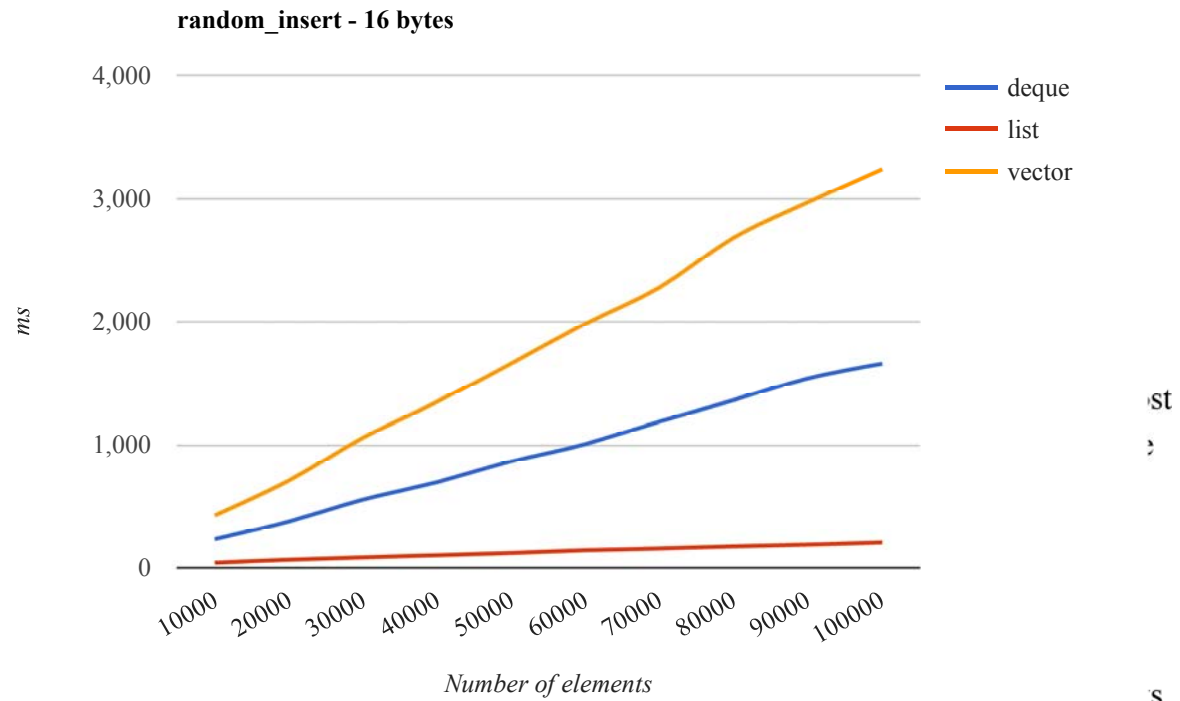


Logarithmic scale

The list is more than 20 times faster than the vector and an order of magnitude faster than the deque ! The deque is also twice faster than the vector.

The fact that the deque is faster than vector is quite simple. When an insertion is made in a deque, the elements can either be moved to the end or the beginning. The closer point will be chosen. An insert in the middle is the most costly operation with $O(n/2)$ complexity. It is always more efficient to insert elements in a deque than in vector because at least twice less elements will be moved.

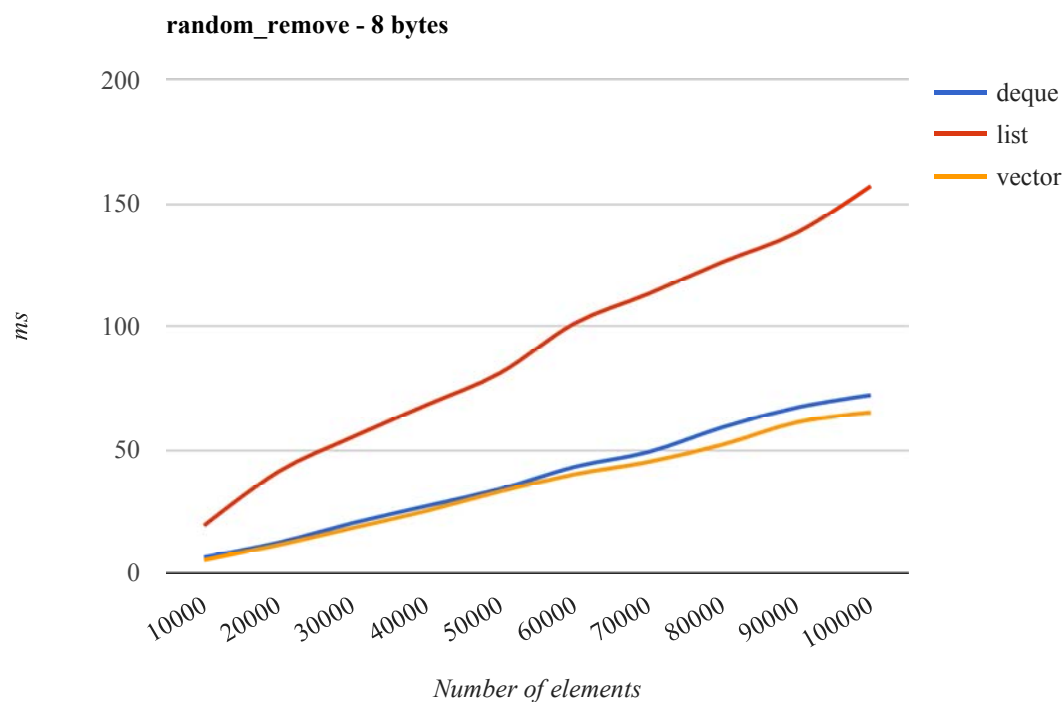
If we look at the non-trivial data type:



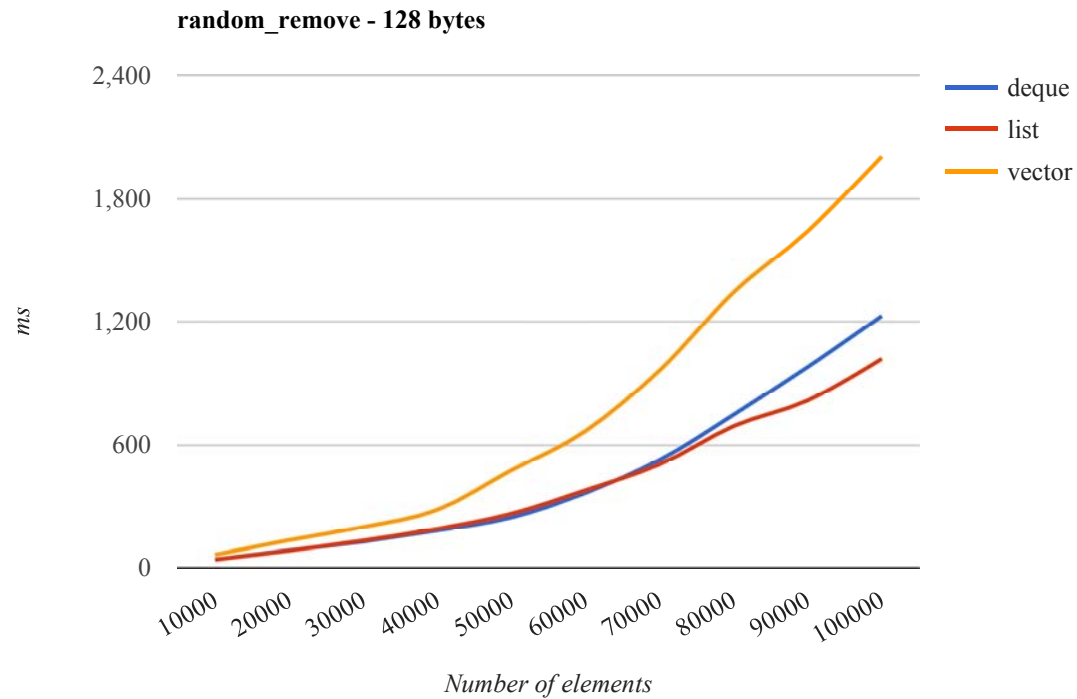
with random insert, we could expect the same behavior for random remove.

The container is filled with all the numbers in $[0, N]$ and shuffled. Then, 1000 random values are removed from a random position in the container.

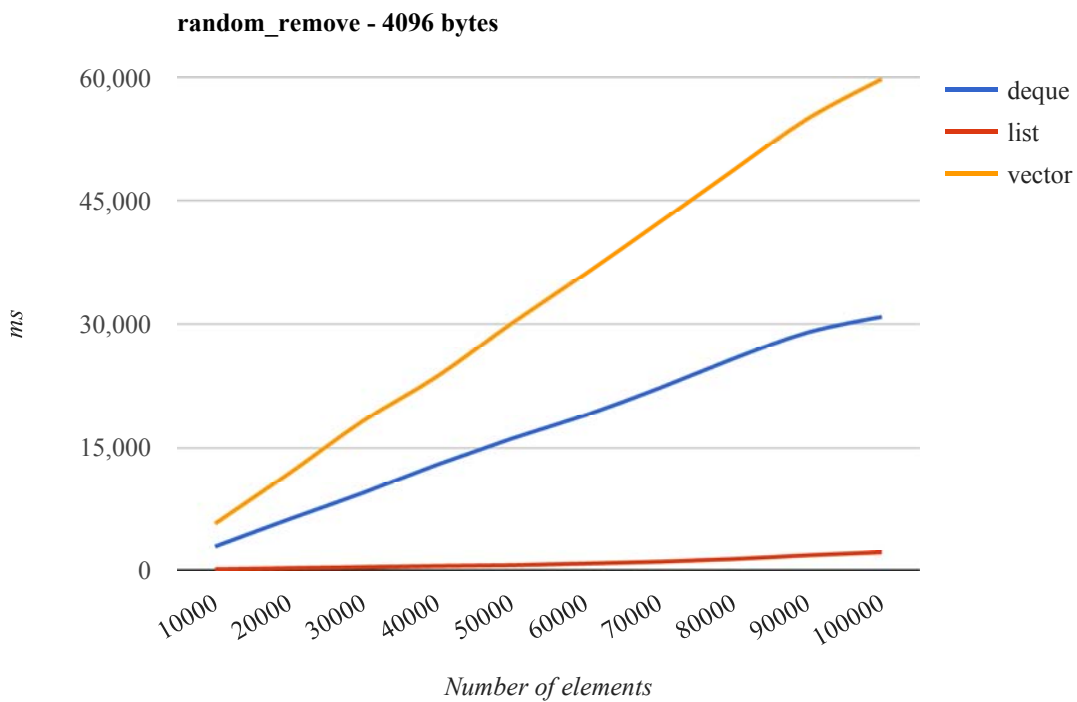
If we take the same data sizes as the random insert case:



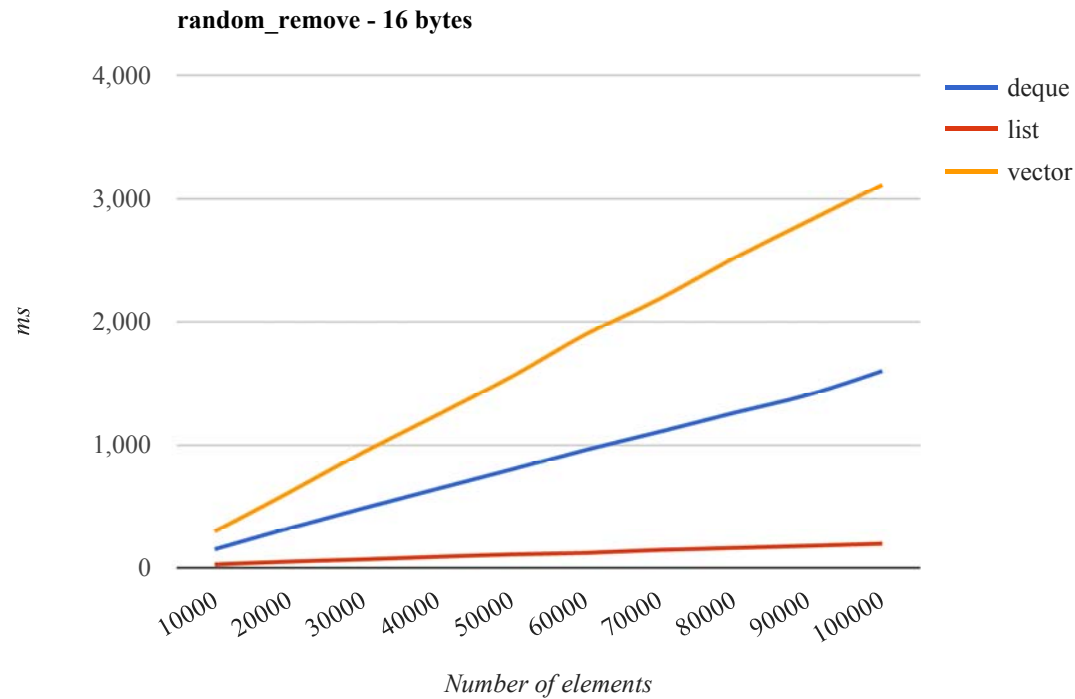
Logarithmic scale



Logarithmic scale



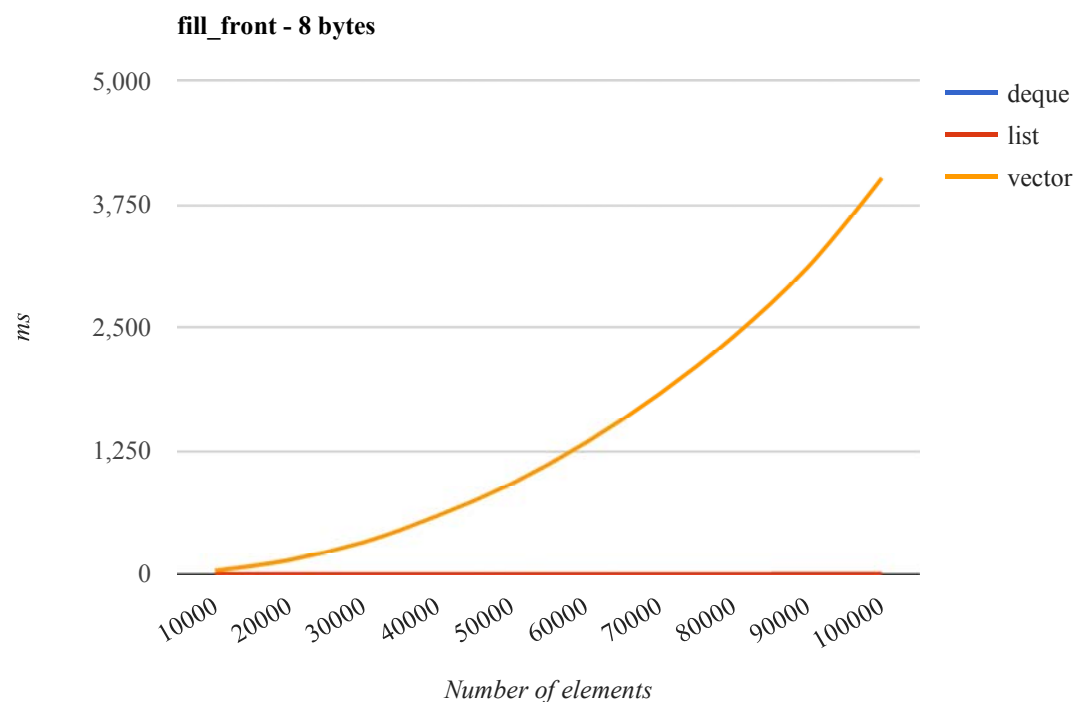
Logarithmic scale



Push Front

The next operation that we will compare is inserting elements in front of the collection. This is the worst case for vector, because after each insertion, all the previously inserted will be moved and copied. For a list or a deque, it does not make a difference compared to pushing to the back.

So let's see the results:

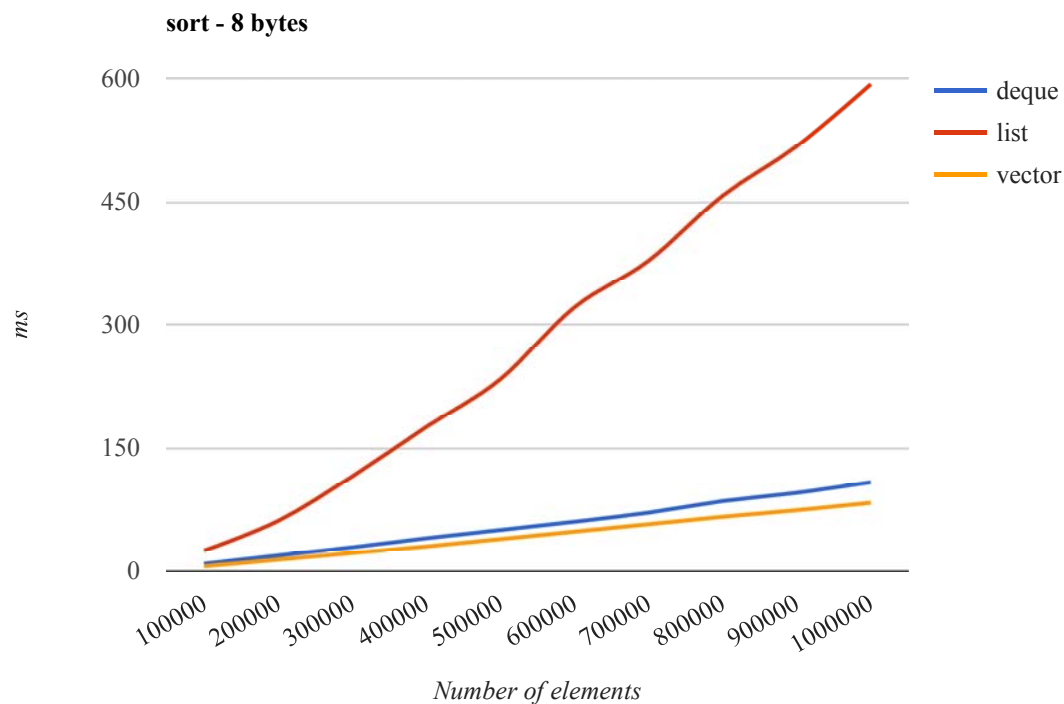


Logarithmic scale

The results are crystal-clear and as expected, vector is very bad at inserting elements to the front. The list and the deque results are almost invisible in the graph because it is a free operation for the two data structures. This does not need further explanations. There is no need to change the data size, it will only make vector much slower and my processor hotter.

Sort

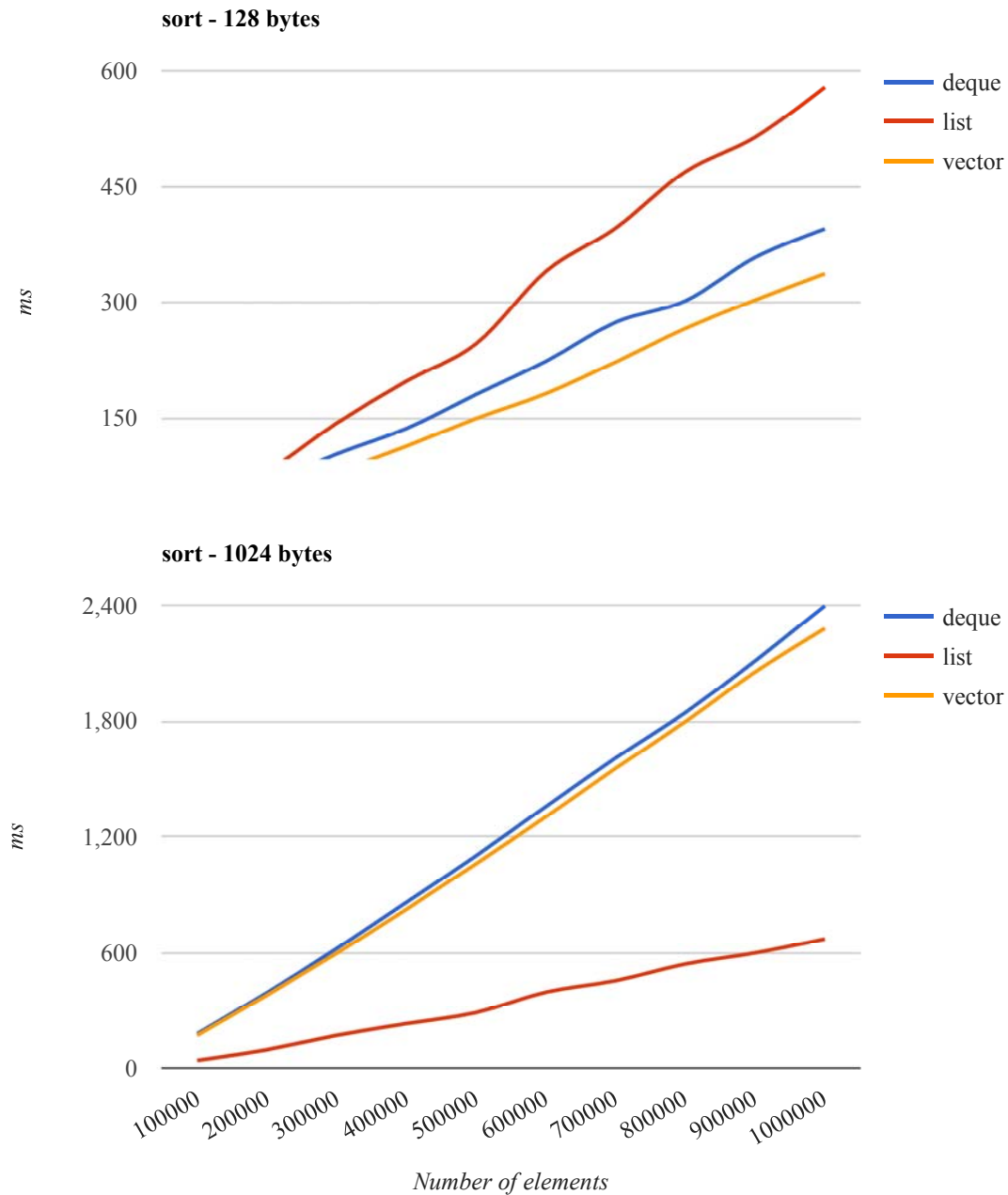
The next operation that is tested is the time necessary to sort the data structures. For the vector and the deque `std::sort` is used and for a list the member function `sort` is used.



Logarithmic scale

For a small data type, the list is several times slower than the other two data structures. This is again due to the very poor spatial locality of the list during the search. vector is slightly faster than a deque, but the difference is not very significant.

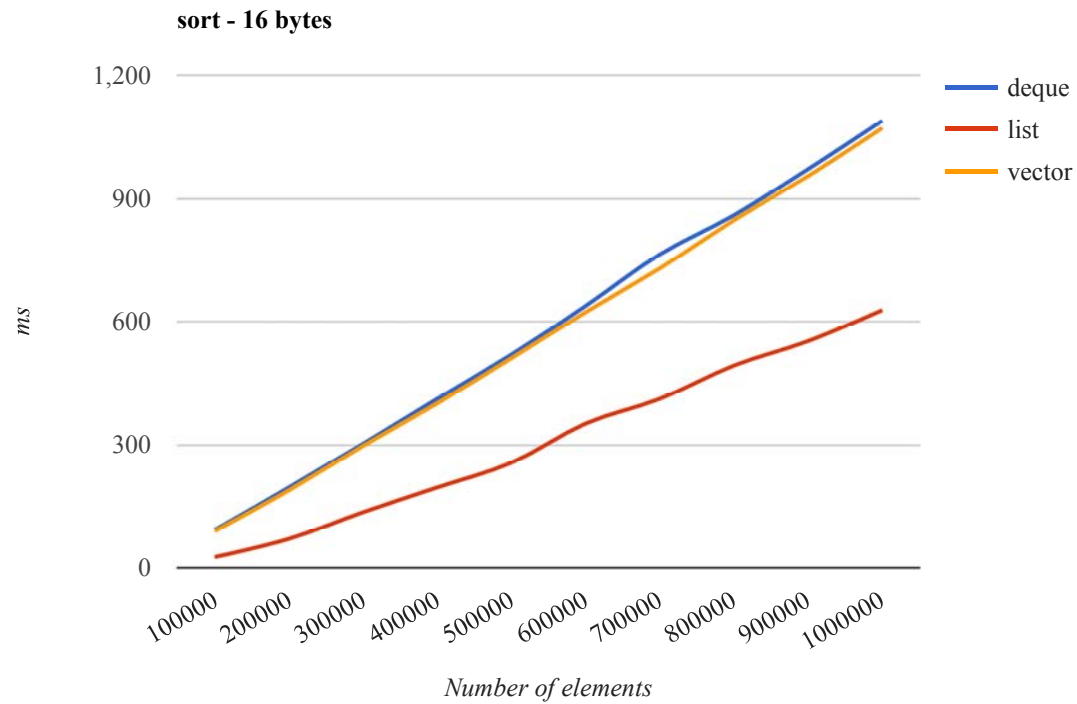
If we increase the size:



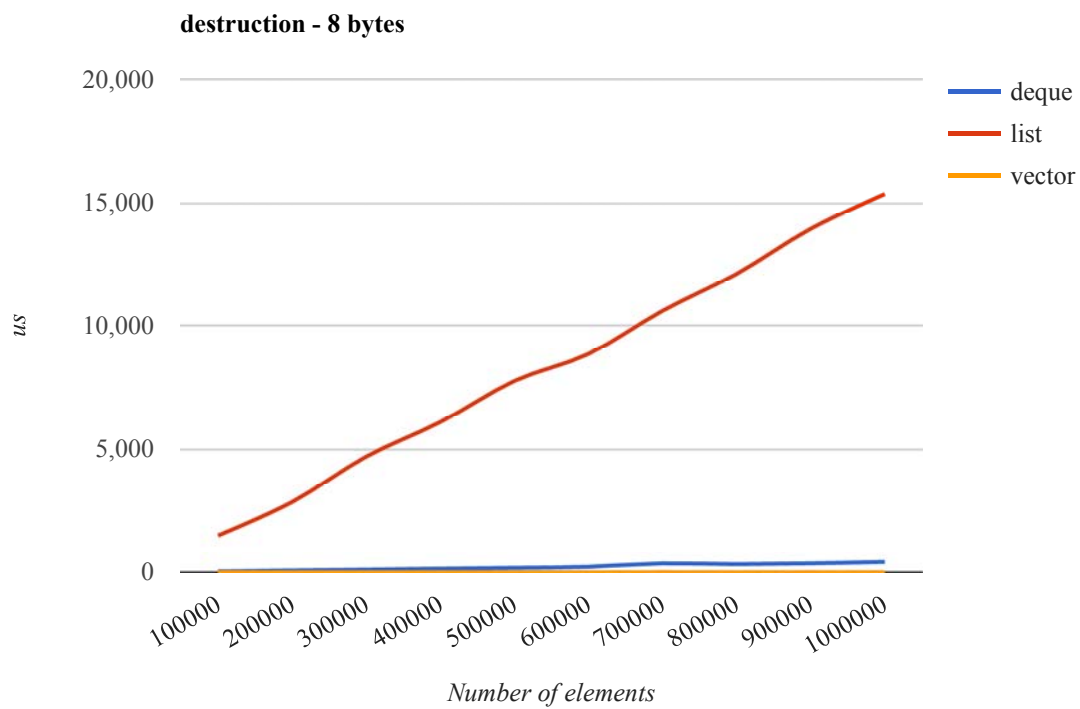
Logarithmic scale

The list is almost five times faster than the vector and the deque which are both performing the same (with a very slight advantage for vector).

If we use the non-trivial data type:



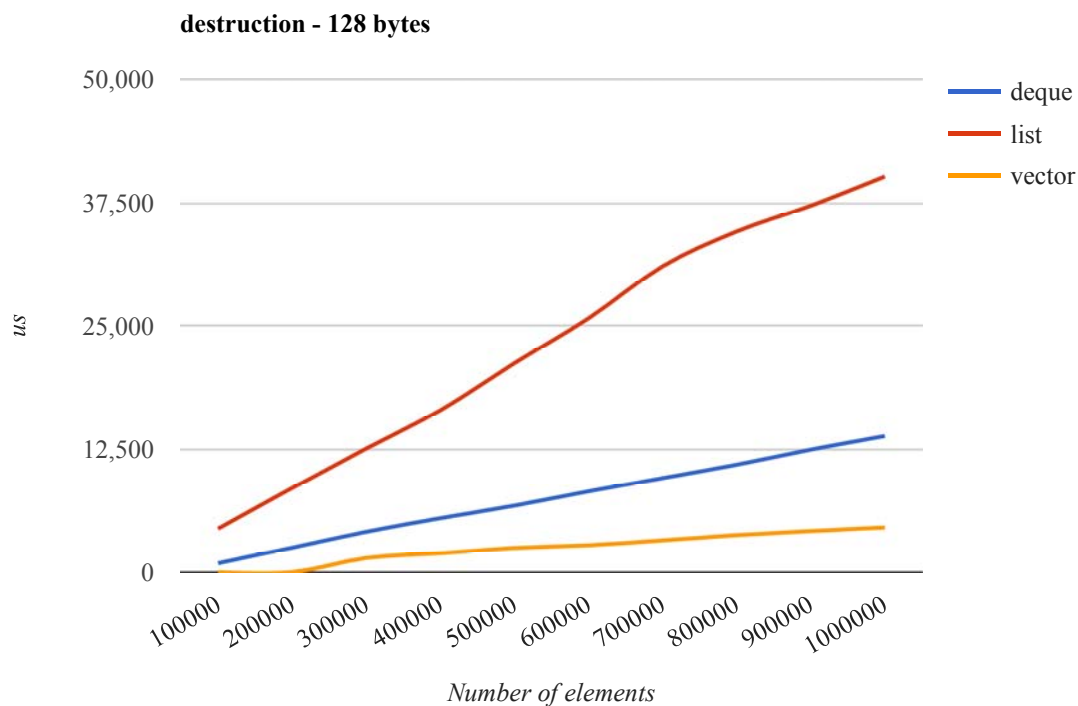
is computed.



Logarithmic scale

The results are already interesting. The vector is almost free to destroy, which is logical because that incurs only freeing one array and the vector itself. The deque is slower due to the freeing of each segments. But the list is much more costly than the other two, more than an order of magnitude slower. This is expected because the list have to free the dynamic memory of each node and also has to iterate through all the elements which we saw was slow.

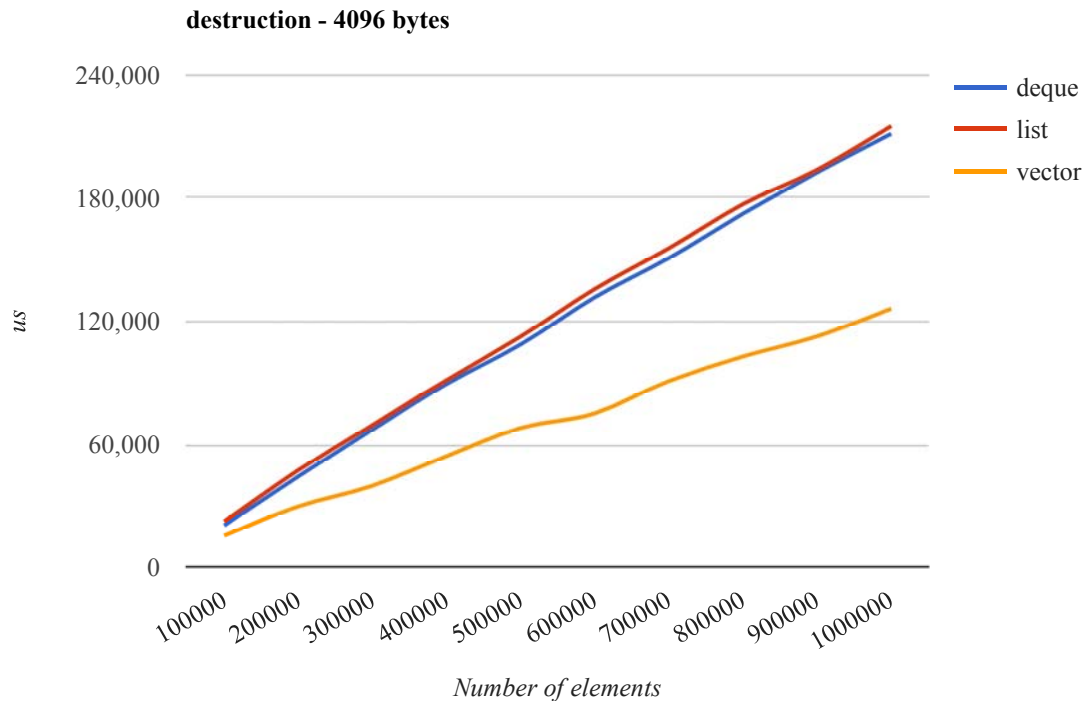
If we increase the data type:



Logarithmic scale

This time we can see that the deque is three times slower than a vector and that the list is still an order of magnitude slower than a vector ! However, the is less difference than before.

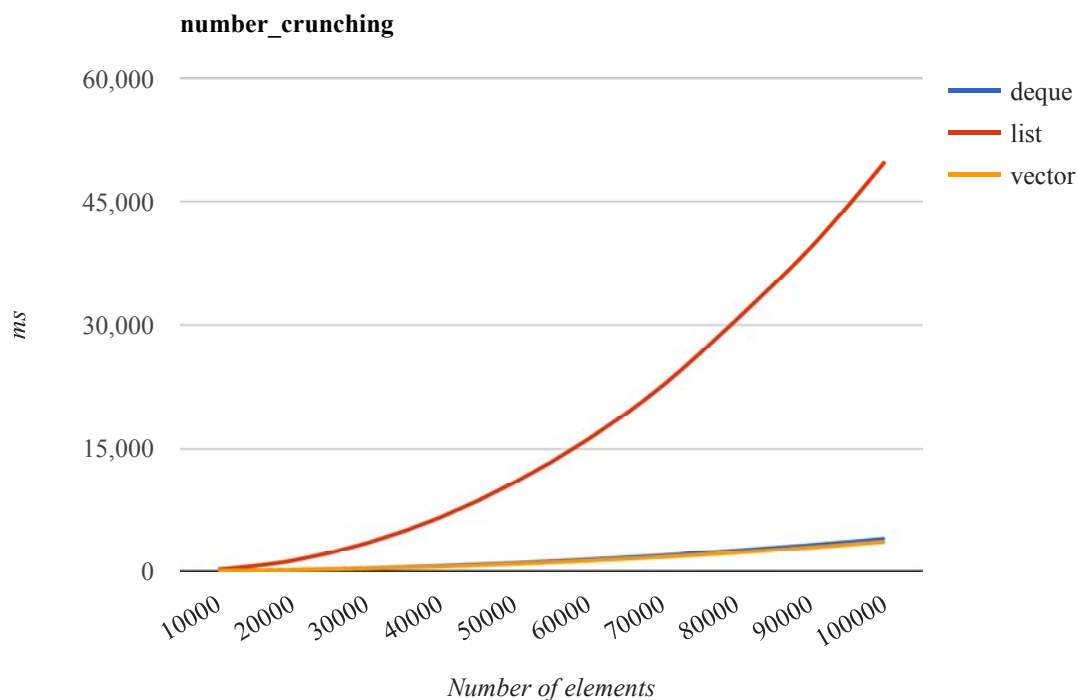
With our biggest data type, now:



1.

ito
ad

crunching, only 8 bytes elements are tested.



Logarithmic scale

Even if there is only 100'000 elements, the list is already an order of magnitude slower than the other two data structures. If we look at the curves of the results, it is easy to see that this will be only worse with higher collection sizes. The list is absolutely not adapted for number crunching operations due to its poor spatial locality.

Conclusion

To conclude, we can get some facts about each data structure:

- `std::list` is very very slow to iterate through the collection due to its very poor spatial locality.
- `std::vector` and `std::deque` perform always faster than `std::list` with very small data
- `std::list` handles very well large elements
- `std::deque` performs better than a `std::vector` for inserting at random positions (especially at the front, which is constant time)
- `std::deque` and `std::vector` do not support very well data types with high cost of copy/assignment

This draw simple conclusions on usage of each data structure:

- Number crunching: use `std::vector` or `std::deque`
- Linear search: use `std::vector` or `std::deque`
- Random Insert/Remove:
 - Small data size: use `std::vector`
 - Large element size: use `std::list` (unless if intended principally for searching)
- Non-trivial data type: use `std::list` unless you need the container especially for searching. But for multiple modifications of the container, it will be very slow.
- Push to front: use `std::deque` or `std::list`

I have to say that before writing this new version of the benchmark I did not know `std::deque` a lot. This is a very good data structure that is very good at inserting at both ends and even in the middle while exposing a very good spatial locality. Even if sometimes slower than a vector, when the operations involves both searching and inserting in the middle, I would say that this structure should be preferred over vectors, especially for data types of medium sizes.

If you have the time, in practice, the best way to decide is always to benchmark each version, or even to try another data structures. Two operations with the same Big O complexity can perform quite differently in practice.

I hope that you found this article interesting. If you have any comment or have an idea about an other workload that you would like to test, don't hesitate to post a comment ;) If you have a question on results, don't hesitate as well.


The code source of the benchmark is available online:

https://github.com/wichtounet/articles/blob/master/src/vector_list/bench.cpp

(https://github.com/wichtounet/articles/blob/master/src/vector_list/bench.cpp)

The older version of the article is still available: C++ benchmark – `std::vector` VS `std::list` (<http://www.baptiste-wicht.com/2012/11/cpp-benchmark-vector-vs-list/>)

Benchmarks C++ C++11 Performances

Contents © 2018 Baptiste Wicht (mailto:baptistewicht@gmail.com) - Powered by Nikola
(http://getnikola.com) - License:  (http://creativecommons.org/licenses/by/4.0/)

Source (cpp-benchmark-vector-list-deque.wp)