Measuring insertion times for C++ unordered associative containers with duplicate elements

We continue our measuring plan for C++ unordered associative containers and focus now on insertion in containers with duplicate elements as implemented by Dinkumware, Boost.Unordered and Boost.MultiIndex. As before, we consider non-rehashing and rehashing scenarios

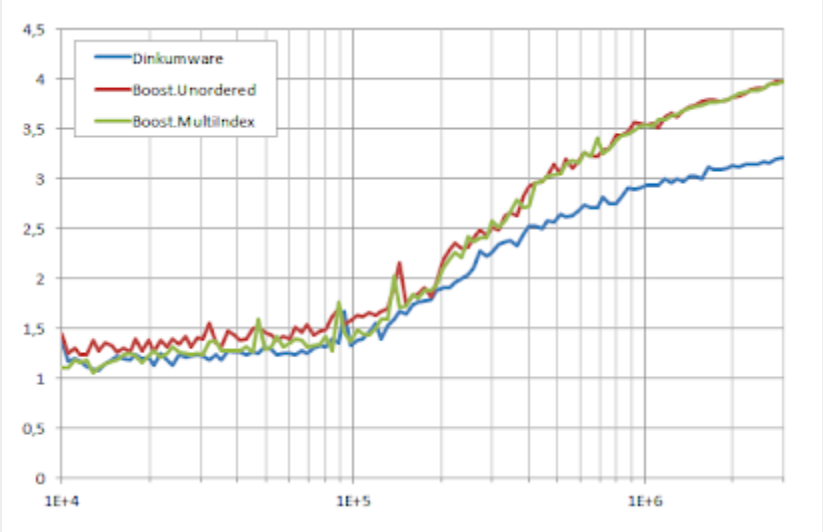
```
// non-rehashing
void insert(container& c,unsigned int n,float Fmax)
{
  c.max_load_factor(Fmax);
  c.reserve(n);
  while(n--)c.insert(rnd());
}

// rehashing
void insert(container& c,unsigned int n,float Fmax)
{
  c.max_load_factor(Fmax);
  while(n--)c.insert(rnd());
}
```

with two additional complications:

- The random source produces duplicate elements where each different value is repeated $G = 5$ times on average within a run of n invocations.
- We study the performance with maximum load factors $F_{\max} = 1$ (the default specified by the standard) and $F_{\max} = G = 5$. The latter case results in an element distribution across buckets equivalent to that of a container *without* duplicate elements being fed the same random source (i.e. where each group of equivalent elements is reduced to just one element). As studied before, this favors implementations that are able to skip groups of equivalent elements in constant time, which is the case for Boost.Unordered and Boost.MultiIndex.

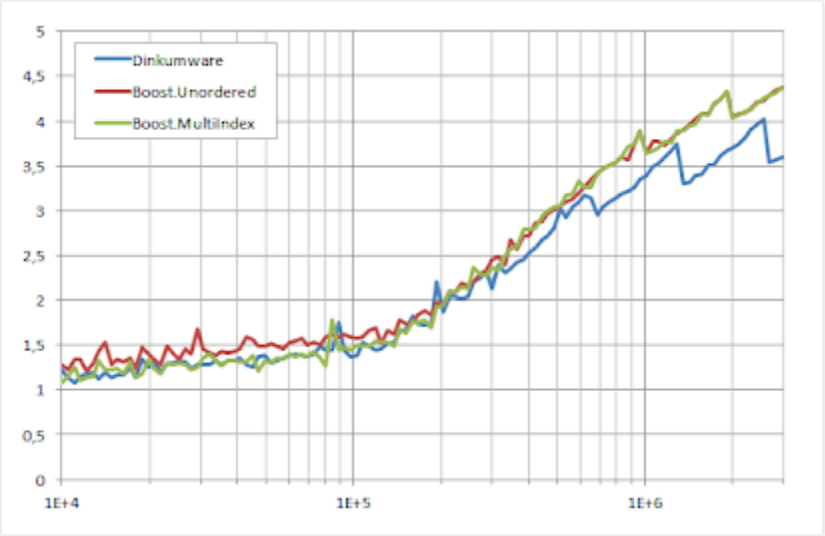
The test program was compiled and run on the same environment as in our previous entry. The first figure shows the results for the non-rehashing scenario and $F_{\max} = 1$, times in microseconds/element.



The sigmoid shapes due to CPU memory cache effects should be familiar by now. Boost.Unordered and Boost.MultiIndex perform very similarly despite their different data structures and insertion algorithms: the former is slower at determining the end of a bucket whereas the latter does worse for groups of size < 3, and these two effects seem to cancel each other out. Why is Dinkumware as fast or even faster than the other two libraries?

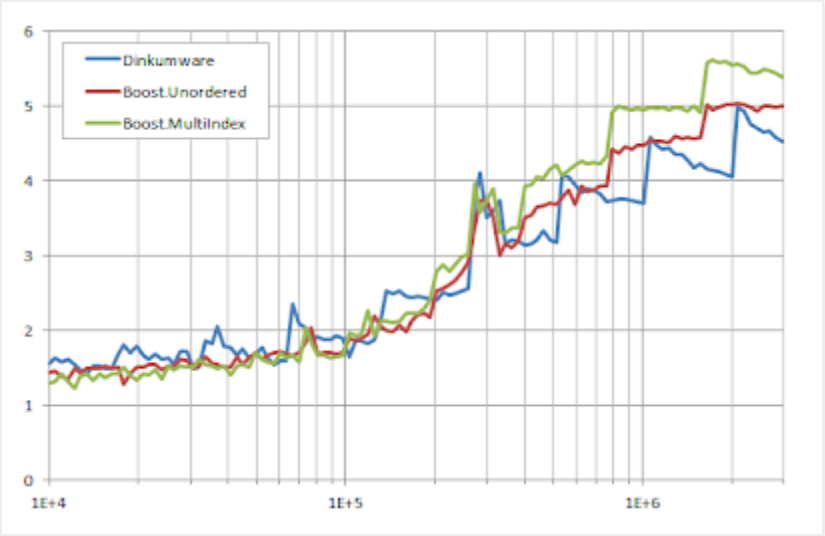
- As we have proved in another entry, the distribution of elements when duplicates are allowed varies significantly with respect to the non-duplicate situation for the same load factor: here many more buckets are empty and the probability that two groups of elements end up in the same bucket is much lower: consequently, most of the time the insertion procedure needs not traverse the entire bucket and the fact that Dinkumware cannot skip groups in constant time is irrelevant.
- Furthermore, the locality of Dikumware is better than that of Boost.Unordered and Boost.MultiIndex, which need to access the first *and* the last element of a group upon each new insertion; this shows up as an additional advantage for Dinkumware when n is large and most of the memory active set lies outside the CPU cache.

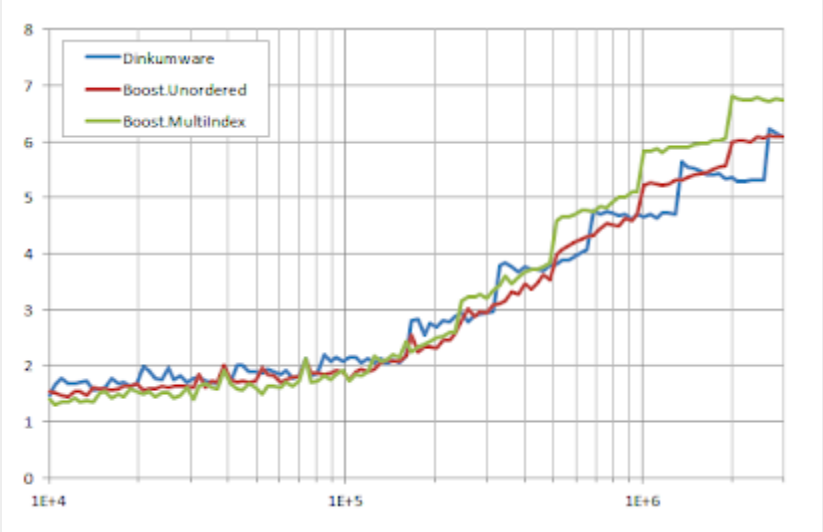
When we set $F_{\max} = 5$, results look like this:



Although the level of bucket occupancy is five times higher, Boost.Unordered and Boost.MultiIndex insertion times does not grow much, precisely because they are not affected by the size of groups of equivalent elements. On the other hand, Dinkumware performance, as expected, degrades noticeably: when two groups occupy the same bucket, which is now more likely, reaching for the second group implies traversing all the elements of the first.

The following figures show the the inserton times for the rehashing scenario and $F_{\max} = 1$ and 5, respectively.





The results are not so easy to interpret as before. There are several competing forces acting here:

- Dinkumware rehashing procedure is the slowest from an algorithmical point of view (each element is rehashed separately) but has the best locality, which compensates its initial disadvantage when n grows.
- Boost.Unordered rehashes fastest of all three libraries because the implementation stores precomputed hash values within element nodes. This is more apparent for large values of n, as evidenced by the height of the graph "steps", and would show even more distinctly if the elements were harder to hash than the plain `unsigned int`s we have used (for instance, if we had dealt with `std::string`s).

In future entries we will go on to profiling erasure and lookup times.