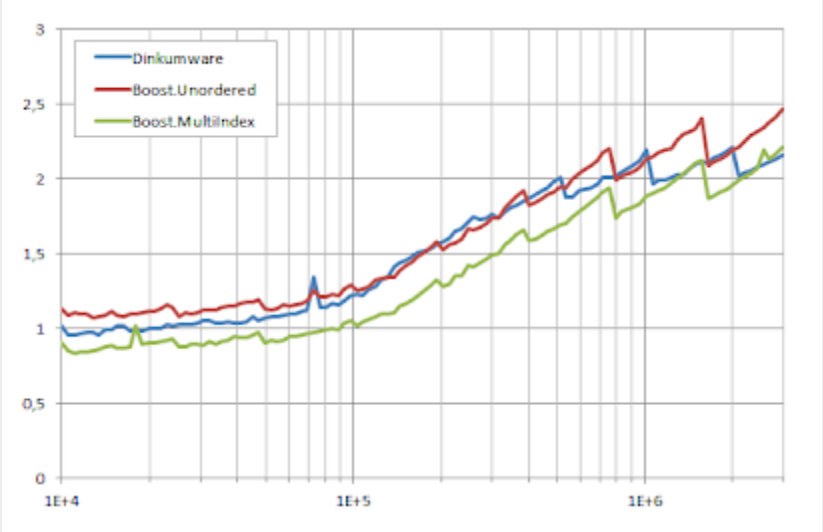
Measuring insertion times for C++ unordered associative containers

Equipped with our theoretical analysis on the complexity of insertion for C++ unordered associative containers without duplicate elements, let us now measure the real thing. I have written a program that profiles running insertion times for Dinkumware, Boost.Unordered and Boost.MultiIndex. Tests were compiled with Microsoft Visual Studio 2012, default release mode settings, and run on a Windows machine with an Intel Core i5-2520M processor running at 2.50GHz.

We begin with the non-rehashing scenario:

```
void insert(container& c,unsigned int n)
{
  c.reserve(n);
  while(n--)c.insert(rnd());
}
```

The results for n = 10,000 to 3 million elements are depicted in the figure. Times are microseconds/element.



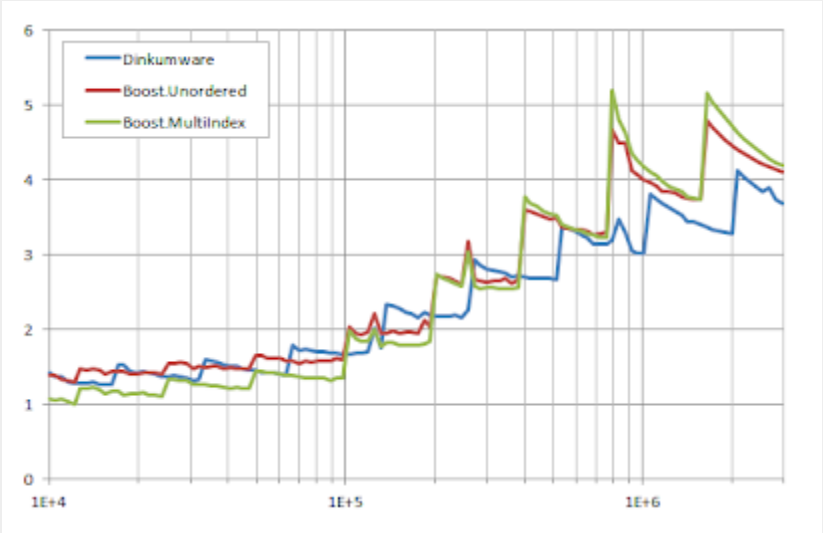
Well, this is anything but average constant time! What we are seeing is in most likelihood the effect of the CPU memory cache, which in our testing environment has a size of 3MB. This means that the active memory set fits into the CPU cache up to $n \approx 150{,}000$ for Dinkumware and $n \approx 200{,}000$ for Boost.Unordered and Boost.MultiIndex: beyond that point average memory read/write time gradually increases and is expected to stabilize when the size of the memory set is much larger than the CPU cache. Leaving these effects aside, what the figure tells us about the performance of the containers is:

- Boost.MultiIndex does slightly faster than Boost.Unordered because determining the end of a bucket only involves some pointer checks in the former lib whereas the latter has to use more expensive verifications based on stored hash values.

- Dinkumware bucket arrays sizes are powers of two whereas Boost.Unordered and Boost.MultiIndex resort to an (approximately) geometric progression of prime numbers: the former scenario allows for a much faster mapping of hash values to bucket entries because `n%m`, which is in general an expensive operation, can be efficiently implemented as `n&(m-1)` when `m` is a power of two. On the other hand, Dinkumware's pointer manipulation is the heaviest of all three libs, and the net result is that its observed insertion performance sits more or less between those of the other two.

Now we consider the rehashing scenario:

```
void insert(container& c,unsigned int n)
{
  while(n--)c.insert(rnd());
}
```



The differences with respect to the previous case are due, obviously, to the presence of rehashing:

- Dinkumware excels when *n* is large because of the advantage we mentioned about using powers of two for bucket array sizes, more prevalent under rehashing conditions.
- Unlike Boost.Unordered, Boost.MultiIndex needs to calculate and temporarily store hash values each time a rehashing occurs, which eats up its head start when *n* grows.

Note that the test uses `unsigned int`s as elements, whose associated hash function is trivial: if the elements were harder to hash (like in the case of `std::string`s), Boost.Unordered, which is the only lib that stores hash values, should beat the other two for large numbers of elements.