Insertion complexity of C++ unordered associative containers

C++ unordered associative containers make the beautiful promise of (average) constant time insertion provided hashing is done properly. Let's see the algorithmic theory hash table insertion relies on. We restrict ourselves to discussing containers not allowing duplicate elements (`unordered_set` and `unordered_map`, which, for the purpose of this analysis, are entirely equivalent).

So, we want to calculate the average complexity of this model of running insertion into a container:

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

where `rnd` is a random source generating non-repeating integers and `c` is empty upon invoking `insert`. Let B_0 be the initial number of buckets of an empty container and F_{max} its maximum load factor (usually 1): when there are $n\text{-}1 < F_{max}B_0$ elements in `c`, the time taken to insert a new one is

$$t(n) = a + b(n\text{-}1)/B_0,$$

for some constants a and b: the first addend represents the constant insertion time of the element into the structure and the second one accounts for collisions into the destination bucket: if hashing is distributing elements uniformly, there are on average $(n\text{-}1)/B_0$ previous elements in each bucket. So, the time complexity of `insert(n)` (if $n \leq F_{max}B_0$) is

$$f(n) = \Sigma_{i=1,..,n}\, t(i) = an + bn(n\text{-}1)/2B_0.$$

Now, when the maximum load factor is hit ($n = F_{max}B_0$), an additional insertion involves rehashing the table to a new, larger number of buckets B. In this scenario, `insert(n)` complexity is

$$f(n) = a(n\text{-}m) + b(n(n\text{-}1) - m(m\text{-}1))/2B + r(m,B) + f(m),$$

with

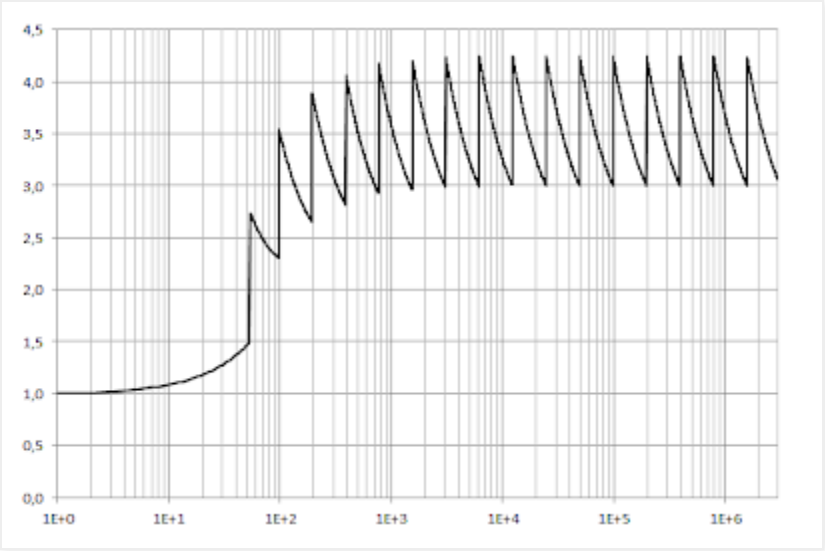
$$m := F_{max}B_0,$$
$$r(m,B) := a'm + b'm(m\text{-}1)/2B.$$

This is simpler than it seems: $f(m)$ is the time taken before rehashing, $r(m,B)$ represents rehashing m elements from B_0 to B (for the sake of generality we allow for a', b' to be different than a, b because no new nodes are allocated, insertion can take some more time

than the regular case due to the creation of the new bucket array, collisions need not be checked, etc.) and the rest of the expression models the insertion of the remaining $n-m$ elements after rehashing. The formula we have derived applies recursively to any n provided that B is the number of buckets used by the container to hold at least n elements and B_0 is the number of buckets after the last rehashing operation (0 if no previous rehashing took place), and defining by convention $r(0,0) = f(0) = 0$. This is a C++ implementation of $f(n)$ (name `running_insertion`) and $f(n)/n$ (`average_running_insertion`):

```
double rehash(unsigned int m,unsigned int B)
{
  static const double aprime=1.0,bprime=1.0;
  return m==0?0:aprime*m+bprime*m*(m-1)/(2*B);
}

double running_insertion(unsigned int n,double Fmax)
{
  static const double a=1.0,b=1.0;
  static const std::array<unsigned int,18> Bs={
    0, 53, 97, 193, 389, 769,
    1543, 3079, 6151, 12289, 24593,
    49157, 98317, 196613, 393241, 786433,
    1572869, 3145739
  };

  if(n==0)return 0.0;

  auto it=std::lower_bound(
    Bs.begin(),Bs.end(),(unsigned int)std::ceil(n/Fmax));
  if(it==Bs.end())throw std::out_of_range("running_insertion");
  unsigned int B=*it,m=(unsigned int)(Fmax*(*(it-1)));

  return a*(n-m)+(b*n*(n-1)-b*m*(m-1))/(2*B)+
         rehash(m,B)+running_insertion(m,Fmax);
}

double average_running_insertion(unsigned int n,double Fmax=1.0)
{
  return running_insertion(n,Fmax)/n;
}
```

53, 97, 193,... is the sequence of bucket array sizes used by Boost.Unordered and other implementations of unordered associative containers. The graphic shows $f(n)/n$ with $F_{max} = a = b = a' = b' = 1$ for $n = 1$ to $3{\cdot}10^6$; the horizontal scale is logarithmic.



Before the first rehashing happens (that is, when $n \le 53$), $f(n)/n$ grows as collisions are more frequent, but the slope of the function is negative after every subsequent rehashing because $f(n)/n$ *amortizes* such steps. From 1,000 elements onwards, the average value of $f(n)/n$ stabilizes at around 3.6 and does not grow indefinitely. This can be proved formally:

Proposition. $f(n)/n$ is bounded if bucket array sizes follow a geometric progression.

Proof. By hypothesis, bucket array sizes form a sequence $B_i = B_0K^i$ for some $K > 1$. Let's call $N_i := F_{\max}B_i$. Now, when $n > N_1$ the recursive calculation of $f(n)$ can be unfolded to

$$\begin{gathered} f(n) = a(n\text{-}N_{k\text{-}1}) + b(n(n\text{-}1) - N_{k\text{-}1}(N_{k\text{-}1}\text{-}1))/2B_k + \\ + \Sigma_{i=0,..,k\text{-}2}(a(N_{i+1}\text{-}N_i) + b(N_{i+1}(N_{i+1}\text{-}1) - N_i(N_i\text{-}1))/2B_{i+1}) + \\ + aN_0 + bN_0(N_0\text{-}1)/2B_0 + \\ + \Sigma_{i=0,..,k\text{-}1}(a'N_i + b'N_i(N_i\text{-}1)/2B_{i+1}), \end{gathered}$$

where $k = \text{ceil}(\log_K n/N_0)$, that is, $n/N_0 \le K^k < nK/N_0$. In a $[N_i + 1, N_{i+1} + 1]$ interval, $\Delta(f(n)/n)$ is monotonically non-decreasing, i.e. $f(n)/n$ is convex (proof omitted), so its maximum happens at $N_i + 1$ or $N_{i+1} + 1$ and hence we can restrict our study of boundedness of $f(n)/n$ to the points $n = N_i + 1$. Approximating x-1 to x for $x \gg 1$, dropping the terms outside of the summatories (as they are constant or grow slower than n) and taking into account the relationship between bucket array sizes, we have

$$\begin{gathered} f(N_i + 1) \simeq \Sigma_{i=0,..,k\text{-}2}(a(N_{i+1}\text{-}N_i) + bF_{\max}(N_{i+1}/2 - N_i/2K)) + \\ + \Sigma_{i=0,..,k\text{-}1}(a'N_i + b'F_{\max}N_i/2K) = \\ = (a(K\text{-}1) + bF_{\max}(K^2\text{-}1)/2K)\,\Sigma_{i=0,..,k\text{-}2}\,N_i + \\ + (a' + b'F_{\max}/2K)\,\Sigma_{i=0,..,k\text{-}1}\,N_i = \\ = (a(K\text{-}1) + bF_{\max}(K^2\text{-}1)/2K)N_0(K^{k\text{-}1}\text{-}1)/(K\text{-}1) + \\ + (a' + b'F_{\max}/2K)N_0(K^k\text{-}1)/(K\text{-}1) < \end{gathered}$$

$$< n\big(a(K\text{-}1) + bF_{\max}(K^2\text{-}1)/2K + a'K + b'F_{\max}/2\big)/(K\text{-}1),$$

thus $f(n)/n$ is bounded.

The (approximate) bounding formula yields 4.25 for the function depicted above ($K \simeq 2$), in excellent agreement with the graph.

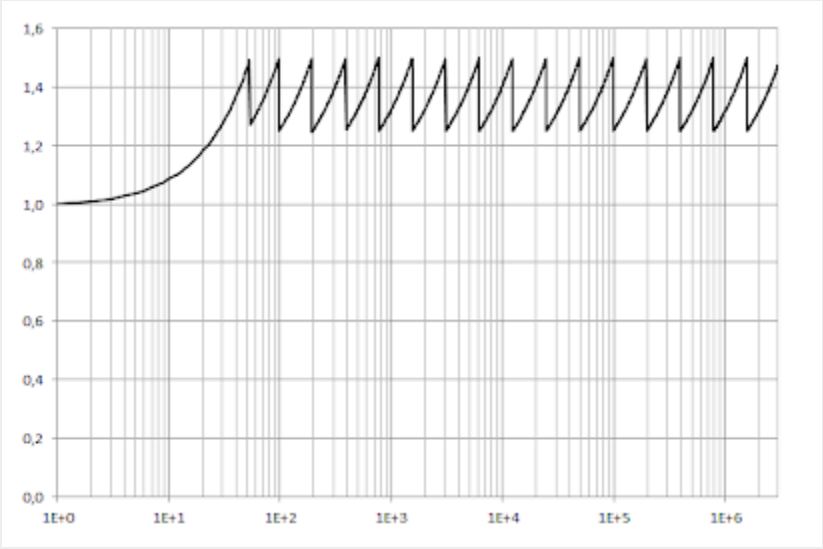
There is an alternative insertion scenario where the total number of elements is known in advance and the user can reserve space to avoid rehashing:

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

The associated complexity of this operation is

$$f(n) = an + bn(n\text{-}1)/2B,$$

where B is the minimum bucket size such that $n \le F_{\max}B$. $f(n)/n$ is, again, bounded, as shown in the figure.



We omit the proof, which is much simpler than the previous one. In this case, the bound on $f(n)/n$ is $a + bF_{\max}/2$. Note that this does not depend on K precisely because there is no rehashing involved.

So, the promise holds true: if the container uses bucket array sizes following a geometric progression and hashing behaves well, insertion is O(1) on average (either with or without rehashing). In a later entry we will see whether real implementations of unordered associative containers live up to theory.