Speeding up the insertion of a sorted (or mostly-sorted) key list into a std::map or other ordered associative container

devblogs.microsoft.com/oldnewthing/20230522-00

May 22, 2023

Raymond Chen

Suppose you want to insert or emplace a bunch of items into a `std::map`, `std::multimap`, `std::set`, or `std::multiset` (collectively known as ordered associative containers), and you have the items in sorted or mostly-sorted order by key. It is common to have the items sorted ascending by keys because you got them from someplace else that produces them in ascending order, but this one weird trick works both for sorted and reverse-sorted insertion, as well as nearly-sorted and nearly-reverse-sorted.

The instinctive way to insert or emplace the items is to do so one at a time:

```
for (auto&& v : source)
{
    map.try_emplace(v.key, v.value);
}
```

This results in an $O(n \log n)$ running time because you are running n insertions, each of which is $O(\log n)$.

If you know that the items are being inserted in ascending key order, you can use the hinted insertion operator to say, “I bet this one goes at the end.”

```
for (auto&& v : source)
{
    map.try_emplace(map.end(), v.key, v.value);
}
```

Conversely, if you know that the items are being inserted in descending key order, you can hint “I bet this one goes at the beginning.”

```
for (auto&& v : source)
{
    map.try_emplace(map.begin(), v.key, v.value);
}
```

According to the standard, if the hint is correct, then the insertion cost drops to $O(1)$ amortized time. More generally, if the new element finds itself placed immediately before the hint, then you get constant amortized time.

All three of the major implementations of the C++ standard library go even further: They will also provide $O(1)$ amortized insertion time if new element goes immediately *after* the hint.[1]

This means that your key list can be *nearly* sorted, and the insertion will still have constant amortized insertion time, for a total of $O(n)$. And even if the list is "not quite nearly sorted", you'll still be better than the original $O(n \log n)$ total insertion time, because only the items that are "more than one spot out of sorted order" will pay the $O(\log n)$ cost.

```
// Amortized linear overall insertion time if the range is sorted by key
// or reverse sorted by key. Performance degrades toward O(n log n) the more
// the list is not sorted.
template<typename C, typename Iterator>
auto try_emplace_mostly_sorted(C&& c, Iterator first, Iterator last)
    -> decltype(c.end())
{
    auto prev = c.end();
    for (; first != last; ++first) {
        prev = c.try_emplace(prev, first->key, first->value);
    }
    return prev;
}
```

Note that if your range is not sorted at all, then the total insertion time will be $O(n \log n)$, but with a higher constant than the unhinted insertion, so don't just use this as a drop-in replacement. Use it when the items are mostly sorted or mostly reverse-sorted.

[1] I determined this by reading the code. You can see it happening in the Microsoft STL version, the gcc libstdc++ version, and the LLVM libcxx version.