Implementation of C++ unordered associative containers with duplicate elements

In a prior entry we reviewed the data structures used by some popular implementations of C++ unordered associative containers without duplicate elements (`unordered_set` and `unordered_map`) and described a new approach introduced by Boost.MultiIndex hashed indices. Let's continue with the corresponding containers allowing for duplicate elements, `unordered_multiset` and `unordered_multimap`.

Hash tables and duplicate elements do not mix well together, the two basic problems being:

- Load factor control based on the total number of elements rather than the number of *different* elements leads to unnecessarily large and sparsely populated bucket arrays;
- Algorithm complexity is not bound by the load factor but instead depends on the average length of groups of equivalent elements.

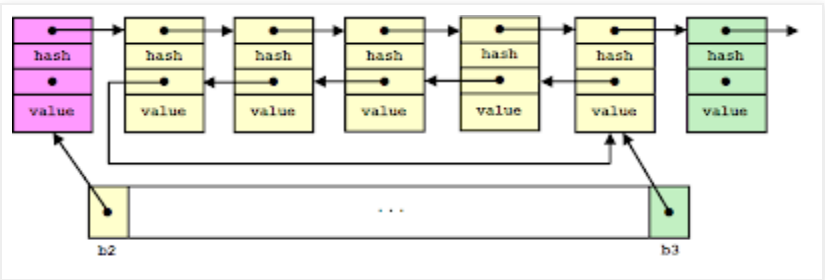
Implementations of `unordered_multiset`/`unordered_multimap` can do nothing to alleviate the difficulties with load factor management: the standard codifies its behavior very precisely with little room for reinterpretation or improvement. An observant user can remedy the situation on her own by setting FG as the load factor, where F controls the level of occupancy (i.e. it is the equivalent load factor for the case where no duplicate elements are allowed) and G is the average length of element groups. As for the second problem, an implementation may choose to enrich the internal data structure so that element groups can be skipped in constant time (the standard specifies that equivalent elements must be packed together): in this case, group length is no longer an issue and the container provides the same performance guarantees as the non-duplicate version. What do popular implementations do in this respect?

Dinkumware, **libc++**, **libstdc++-v3**

None of these libraries makes any special provision to handle groups of equivalent elements in an efficient manner: they rely on the same data structure used for non-duplicate versions of the containers.

Boost.Unordered

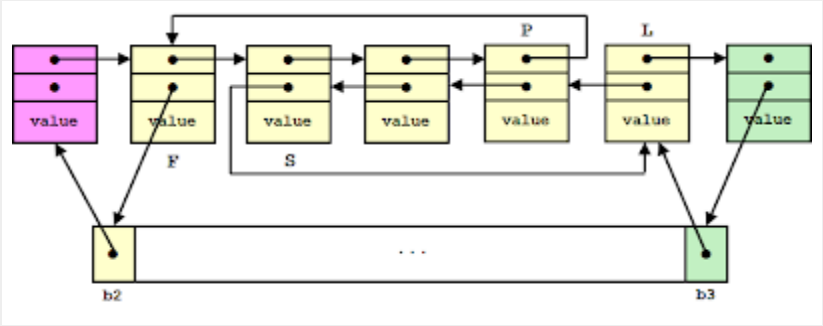
This library does augment its data structure to cope with equivalent elements:



The figure shows a group of five equivalent elements located at bucket `b2`. An additional back pointer is added to the original structure that reverse-links the elements of the group together in the way a circular doubly linked list does. This back pointer allows groups to be treated as whole units: when traversing a bucket for insertion or erasure, only the first element of each group need be inspected, and the rest can be skipped by following the back pointer to the last element. The penalty for this performance improvement is one extra pointer per node.

Boost.MultiIndex

The same data structure used for containers without duplicates can be further tweaked to efficiently handle groups of equivalent elements at no extra memory overhead:



To interpret the tangle of pointers shown in the figure we use the *Xn*, *Xp* terminology introduced in the first article. As there, the first element of each bucket is redirectioned to have *Xp* point to the associated bucket entry. Now consider a group of three or more equivalent elements with *F*, *S*, *P*, *L* pointers to the first, second, penultimate and last node, as marked in the figure (*S* = *P* when the group has length 3); the following redirections are applied:

- $Sp = L$,
- $Pn = F$.

It is not hard to prove that:

- *X* is the first of its bucket iff $Xpnn = X$,
- *X* is the last of its bucket iff $Xnpn = X$,
- *X* is the first of a group (of length ≥3) iff $Xnp \neq X$ and $Xnppn = X$,
- *X* is the second of a group (of length ≥3) iff $Xpn \neq X$ and $Xppnn = X$,
- *X* is the penultimate of a group (of length ≥3) iff $Xnp \neq X$ and $Xnnpp = X$,
- *X* is the last of a group (of length ≥3) iff $Xpn \neq X$ and $Xpnnp = X$,

that is, each special position can be univocally determined. Moreover, given *X* we can apply these properties to locate in constant time the nodes following and preceding *X*:

- The element following *X* is *Xnnp* if *X* is the penultimate of a group (of length ≥3), *Xn* otherwise,
- the element preceding *X* is *Xpn* if *X* is the first of its bucket, *Xppn* if *X* is the second of a group (of length ≥3), *Xp* otherwise.

So, even with all these redirections we can still treat the structure as a circular doubly linked list with O(1) linking/unlinking, and at the same time take benefit of group identification to speed up hash operations. The procedure for insertion follows these steps:

1. Use the hash function to locate the corresponding bucket entry → constant time.
2. Traverse the first elements of each group in the bucket looking for a group of equivalent elements → time linear on the number of *groups* in the bucket.
3. If a group was found, link into it; otherwise, link at the beginning of the bucket → constant time.
4. Adjust bucket entry pointers as needed → constant time.

Group skipping is constant time: if X is the first element of a group of length ≥ 3, the last element is reached with Xnp; if the length is 1 or 2, fast skipping is not available, but the number of elements to check against is, again, 1 or 2, i.e. bounded. The scheme for erasure looks like:

1. Unlink from the (doubly linked) list → constant time.
2. Adjust bucket entry pointers if the element is the first or the last element of its bucket, or both → constant time.

The actual implementation is more complicated than this overall description suggests, since linking and unlinking must maintain the special redirections used to identify groups, but the net result from the point of view of performance is: insertion and erasure times depend on the number of groups rather than the total number of elements and have the same complexity as the non-duplicate case, i.e. O(1) for insertion if the hash function works properly, unconditionally O(1) for erasure.