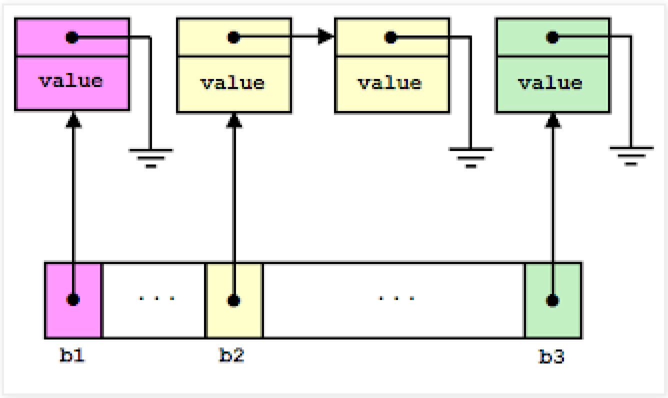
# Implementation of C++ unordered associative containers

*Unordered associative container* is a fancy name adopted by C++ for what the rest of the world refers to as a *hash table*. With the 2011 revision of the standard, the language provides four different unordered associative containers:

- `unordered_set` and `unordered_map`,
- `unordered_multiset` and `unordered_multimap`, allowing for duplicate elements.

Given the particular interface these containers have, their internal representation is universally based on closed addressing, of which the following is a straightforward implementation.

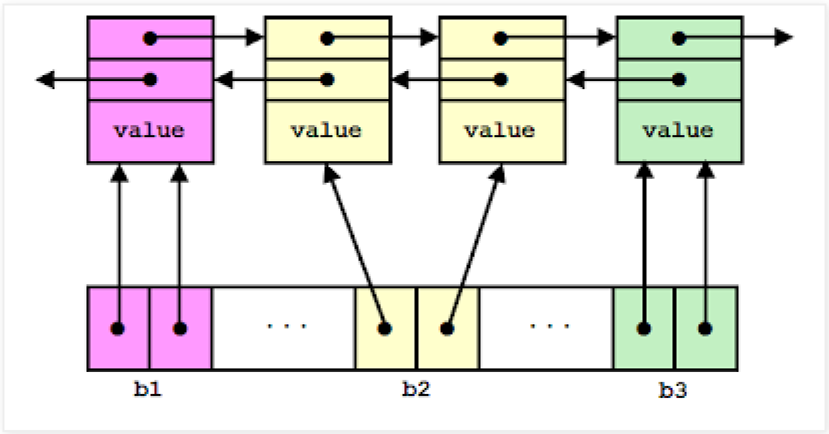


In this data structure, each bucket points to a null-terminated singly linked list of elements. If the provided hash function works as it should, the number of elements in each bucket only rarely exceeds by much the specified load factor, so insertion and erasure of elements is O(1) on average. In the figure above, buckets `b1` and `b3` have one element each, and bucket `b2` holds two elements. Simple and memory-efficient as this structure is, it turns out C++ unordered associative containers cannot use it directly: containers are required to be *traversable*, i.e. they must provide an associated iterator type which can go through all the elements of a container `c` as comprised in the range [`c.begin()`, `c.end()`). So, how does an iterator go from say bucket `b1` to `b2`? The obvious solution of scanning the bucket array until the next non-empty bucket is found does not work, since the average time this operation takes grows as the bucket array is depleted and more buckets need to be walked through to reach the following element, whereas the standard requires that iterator increment be O(1). As this has been discussed in extenso elsewhere, we'll make a long story short and simply conclude that, in order to meet the requirements of C++ unordered associative containers, *all the elements must be linked together*.

Let's see the resulting data structures used by some popular implementations of unordered associative containers, plus a new approach adopted by Boost.MultiIndex. We consider containers without duplicate elements fist (`unordered_set` and `unordered_map`), and study `unordered_multiset` and `unordered_multimap` in a further article. In what follows we call *N* the number of elements in the container and *B* the number of its buckets.

## Dinkumware

These are the providers of the Microsoft Visual Studio C++ standard library implementation. Their hash tables are designed as shown in the figure:



(A note on this and the following diagrams: although seemingly implied by the pictures, bucket entries need not be arranged in the same order as their corresponding buckets are traversed.)

As anticipated, all the elements are linked together, although in this case they are doubly so: this allows for bidirectional iteration at the expense of one more pointer per element, and, more importantly, makes unlinking elements extremely easy. As for the bucket array, each entry holds two pointers to the first and last element of the bucket, respectively (the latter pointer is needed to determine the extent of the bucket). Insertion of a new element is simple enough:

1. Use the hash function to locate the corresponding bucket entry → constant time.
2. Look for duplicate elements in the bucket and abort insertion if found → time linear on the number of elements in the bucket.
3. Link into the element list at the beginning of the bucket → constant time.
4. Adjust bucket entry pointers as needed → constant time.

And erasure goes as follows:

1. Use the hash function to locate the corresponding bucket entry → constant time.
2. Unlink from the element list → constant time.
3. Adjust bucket entry pointers as needed → constant time.

The resulting complexity of these operations is then O(1) (in the case of insertion, under the assumption that the hash function is well behaved and does not overpopulate buckets) and the memory overhead is two pointers per element plus two pointers per bucket entry:

$$2N + 2B,$$

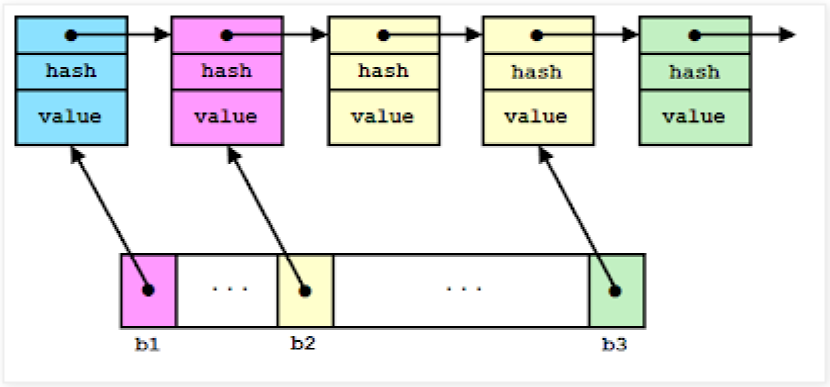
All is well then, right? Well, actually not: Dinkumware's scheme has a serious problem with erasure, as the first step of the procedure

1. Use the hash function to locate the corresponding bucket entry.

will fail if the user-provided hash function throws an exception, and the C++ standard requires that erasure work unconditionally and *never throw*. Blunt as it sounds, it is my opinion that Dinkuware's implementation of C++ unordered associative containers is broken.

**Boost.Unordered, libc++, libstdc++-v3**

Boost.Unordered and libc++ (Clang's standard library implementation) use basically the same data structure, whose design is a direct application of three guiding principles:



1. All elements are linked (here, singly linked) together.
2. Recognizing the fact that, in order to implement erasure without throwing, we must connect an element with its bucket entry without invoking the hash function, the associated hash value is stored as part of the element node.
3. This is the trickiest part: in a singly linked list, erasure of an element takes linear time unless we know the preceding element in advance. So, instead of having each bucket entry point to the first element in the bucket, we make it point to the element before that one. The figure can be more easily understood if color codes are taken into account: for instance, `b2`, which is yellow, points to a fuchsia element (in `b1`).

The algorithm for insertion is based on the same guidelines as those of Dinkumware (with the complication that the end of the bucket is located by comparing stored hash values), whereas erasure follows these steps:

1. Locate the associated bucket entry via the stored hash value → constant time, never throws.
2. Locate the preceding element → linear on the number of elements in the bucket.
3. Use the preceding element to unlink from the element list → constant time.
4. Adjust bucket entry pointers as needed → constant time.

Insertion and erasure are then O(1) for reasonably good hash distributions, which is the minimum mandated by the C++ standard. The memory overhead is, using the same terminology as before,

$$2N + 1B,$$

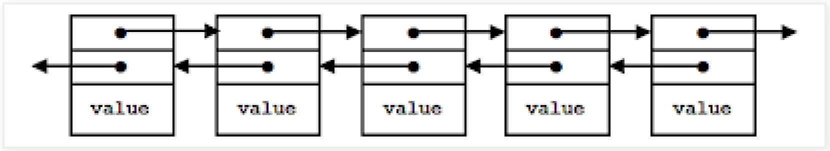
(here we are assuming that storing a hash value takes the same space as a pointer, namely a word.)

GNU Standard C++ Library v3, also known as libstdc++-v3, provides a memory optimization over this data structure: if the hash function is determined to not throw (by `noexcept` specification) and is marked as "fast" (via some dedicated type trait), hash values are not stored and the function is safely used instead. The way this optimization is currently implemented looks to me to a bit dangerous, as the `__is_fast_hash` type trait is true by default except for some basic types, which means that all `noexcept` user-provided hash functions will be deemed as fast unless the user is aware of this implementation-specific trait and states otherwise.

So, leaving libstdc++-v3 optimizations aside, $2N + 1B$ seems to be the minimum overhead required to implement C++ unordered associative containers based on closed addressing and meeting the complexity requirements of the standard. In fact, we can do better in the area of complexity without giving up extra memory.

## A novel data structure

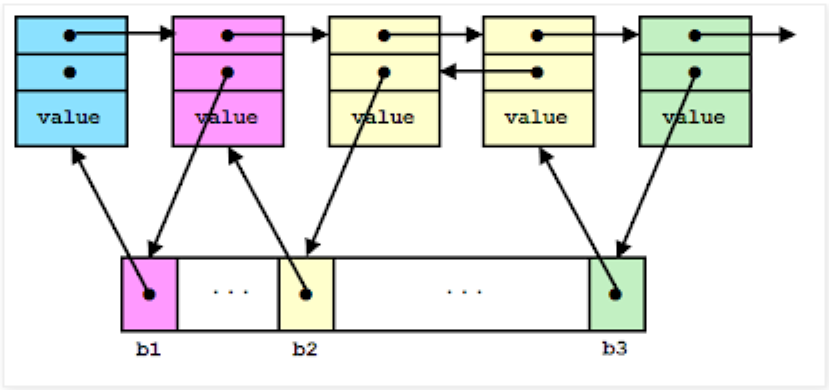
As of Boost 1.56, Boost.MultiIndex hashed indices internally use a new hash table design with some nice complexity characteristics. The key insight behind the approach taken is this: a circular doubly linked list has redundancies that we can take advantage of to accomodate extra information:



Given a node pointer $X$, let's call $Xn$ and $Xp$ the pointers to the next and prior node, respectively. A circular doubly-linked list maintains this invariant:

$$X = Xnp = Xpn \text{ for every } X \text{ in the list.}$$

With some care, we can redirect $Xn$ or $Xp$ (which is detected by the breach of the former invariant) provided we know how to get back to $X$. This is precisely what we are doing to associate elements to bucket entries in a hash table:



Think of this data structure as a simple circular doubly linked list where some redirections have been applied in the following case:

- If an element *X* is the first of its bucket, *Xp* points to the bucket entry.

Besides, bucket entries point to the element preceding the first one in the bucket, just as Boost.Unordered does. Extending our terminology so that if *X* points to a bucket entry then *Xn* denotes the element the entry points to, we have these formal properties:

- *X* is the first element of its bucket iff $Xpn \neq X$,
- *X* is the last element of its bucket iff $Xnp \neq X$,
- the element following *X* is always *Xn*,
- the element preceding *X* is *Xpn* if *X* is the first of its bucket, *Xp* otherwise.

Note that the evaluation of all these properties can be done in constant time, so we can treat our element list as a circular doubly linked list with special operations for accessing the preceding and following nodes and use it for implementing hash table insertion (following the same general steps as with the other data structures) and erasure:

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.

Erasure of an element does not imply traversing the bucket, so unlinking happens in constant time even if the bucket is crowded by a bad behaved hash function; we have then designed a hash table with *unconditional* O(1) erasure complexity at no extra cost in terms of memory overhead ($2N + 1B$).