CHAPTER

08. Hashtables

Let’s replace the placeholder map from the previous chapter. Prerequisites: Basic data structures, big-O notations.

8.1 Hashtable quick review

Data structures for KV

There are 2 classes of data structures for KV stores:

- Sorting data structures, such as AVL tree, Treap, Trie, B-tree.
- 2 types of hashtables: open addressing & chaining.

Sorting data structures maintain order and use *comparisons* to narrow the search, their complexity is $O(\log(N))$. While hashtables rely on *uniformly* distributed hash values to achieve $O(1)$.

How to choose between them? The primary concern is whether the keys are sorted. For top-level keys in Redis, there is no such requirement, so we choose the faster one.

From arrays to hashtables

Let’s say you need a map where keys are *unique integers*, this is trivially satisfied with an array. We have done this with the fd-to-connection map:

```
std::vector<Conn *> fd2conn;    // like map<int, Conn *>
```

Now consider use cases where keys are *not unique*, the map is 1-to-many, so we need to store a list of values instead:

```
std::vector<std::vector<T>> m;
```

Now consider that keys can be *non-integer* types such as strings, structs, we can *reduce arbitrary types to integers* with a *hash function*. Different keys may hash to the same integer, this is called a *collision*, but that's no problem for an array-of-arrays, we just need to store both keys and values to distinguish them.

```
std::vector<std::vector<std::pair<K, T>>> m;
m.resize(N);
m[hash(key) % N].push_back(std::pair{key, val});    // insert


for (std::pair<K, T> &p : m[hash(key) % N]) {       // lookup
    if (p.first == key) { /* found */ }
}
```

This is how to invent hashtables in 3 lines. No excuse for not understanding.

2 types of hashtables: chaining vs. open addressing

The array-of-arrays is a type of *chaining* hashtable. The nested collection doesn't have to be arrays, array-of-linked-lists is the most common, array-of-trees are also viable.

```
std::vector<std::vector<std::pair<K, T>>> m;    // array of arrays
std::vector<std::list<std::pair<K, T>>> m;      // array of linked lists
std::vector<std::map<K, T>> m;                  // array of trees
```

Another type of hashtable is called *open addressing*, which doesn't use nested collections and stores KV pairs directly in the array. In case of a collision, it simply finds another array slot and uses it if it's empty, otherwise, it keeps probing for more slots. The scheme for probing for the next slot is deterministic, so given a hash value, it can find all colliding keys.

There are different probing schemes such as linear probing, double hashing, but we won't go into details because we'll choose chaining instead.

Why are hash functions essential?

Typical hash functions produce a *uniform* distribution over the full range of the integer. However, the array size N is far smaller than the integer range, so the range is reduced by `hash(key) % N`.

A hash function is needed even if the keys are already integers, because it reduces collisions. For example, if keys are pointers, then keys are likely to be multiples of 8 due to alignment, which means `key % N` can only use 1/8 slots, which means each used slot has 8x more keys on average, which means searching existing keys takes longer.

The worst case is that all keys are mapped to the same slot, making searches $O(N)$ instead of $O(1)$. This is less likely to happen with a hash function because the key-to-slot mapping is effectively (pseudo) random even with skewed key distributions.

The maximum load factor

Why are hashtables $O(1)$ on average? Because there is a limit to the number of keys relative to the number of slots. Let's say N slots can store $k \times N$ keys, then each slot has on average k keys. This ratio is called $\text{load\_factor} = \text{keys}/\text{slots}$.

When the maximum load factor is reached, keys are migrated to a larger hashtable (called *rehashing*), which can be triggered by insertion. Like dynamic arrays, the new hashtable is exponentially larger, so the amortized insertion cost is still $O(1)$.

8.2 Hashtables for Redis

Worst case latency causes scalability issues

The principles of hashtables are simple, as the 3-line hashtable shows. So why bother coding one? Why not just use `std::unordered_map` from C++ STL? Because if Redis is just STL collections over a network, there will be issues when deployed at scale.

There are 2 types of scalability problems: *throughput* and *latency*. Throughput problems often have generic, easy solutions, such as sharding and read-only replicas, while latency problems are often domain-specific and harder. Throughput is an average case problem while latency is often a worst case problem.

For hashtables, the largest latency issue comes from insertion, which may trigger an $O(N)$ resize. A modern machine can easily have hundreds of GB of memory, but resizing a hashtable of this scale takes a long time, rendering the service unavailable.

There is a solution to this issue, but it's not in C++ STL, or most general-purpose libraries, because most data structure libraries are optimized for throughput, not latency. The common belief is that there's a library for every need, but the more you learn, the less it's true.

Progressive hashtable resizing

The solution to resize latency problem is to do it *progressively*. After allocating the new hashtable, don't move all the keys at once, only move a fixed number of keys. Then, each time the hashtable is used, move some more keys. This can slow down lookups during resizing because there are 2 hashtables to query.

After allocating a hashtable, the slots must be initialized. So *triggering* a resize still seems to be $O(N)$. This is avoided with `calloc()`. When allocating a large array, `calloc()` gets the memory from `mmap()`, and the pages from `mmap()` are allocated and zeroed on the first access, which is effectively a progressively zero-initialized array.

For smaller arrays, `calloc()` gets the memory from the heap, which requires immediate zeroing, but the latency is bounded. The threshold between large & small is determined by the libc.

`calloc()` should be used instead of `malloc()` + `memset()` to avoid the $O(N)$ initialization latency. But if you are using a data structure library or a high-level language, you probably don't have this level of control. That's where low-level C code is needed.

Chaining hashtables are robust, simple, flexible

The next latency issue is collision. The possibility of a very long chain of collision always exists. Chaining hashtables are less prone to collisions because colliding keys share the same slot, whereas with open addressing, colliding keys take multiple slots, further increasing the probability of collision.

There are many probing strategies to choose from. Simple probing strategies, such as linear probing, result in a higher probability of collision, so they are mostly exist only in textbooks. While chaining hashtables do not have this consideration.

We'll code a *chaining hashtable with linked lists*. But why linked lists over arrays?

- Latency: Insertion is always $O(1)$, deletion is $O(1)$ after finding the node.
- Reference stability: Pointers to data remain valid even after resizing.
- Can be implemented as an intrusive data structure, explained later.

Reference stability is the reason why STL uses chaining hashtables. It makes hashtables more flexible and less error-prone to use.

8.3 Generic collections in C

Our hashtable will be used with different data types later, but C++ generics aren't the only way; we'll learn another, more flexible solution called *intrusive data structures*. It's an overlooked technique because textbooks don't teach it, but it's used by many practical C projects, such as the Linux kernel.

But first, let's review common methods for making data structures generic and their drawbacks, using linked list as the example.

Method 1: `void *` pointers

The most common method in C is to use the typeless pointer `void *`.

```
struct Node {
    void *data; // points to anything
    struct Node *next;
};
```

If the data happens to be pointer-sized or smaller, it can be stored in it. But often the data is larger, or variable-sized, so the data is probably allocated from the heap, which has some drawbacks:

- A level of pointer indirection to access the data.
- More dynamic memory management.
- No type checking.

Method 2: Generate code with C macros, code generators

C++ templates generate per-type code, which can be emulated with C macros.

```
#define DEFINE_NODE(T) struct Node_ ## T { \
    T data; \
    struct Node_ ## T *next; \
} \
// and all the list functions...
```

This can also be done without macros using a code generator. From the compiler's point of view, this is the same as C++ generics. But from the programmers' point of view, this is an undebuggable, unmaintainable mess. Don't bother with this, we'll learn a better way.

8.4 Intrusive data structures

Embedding structures within data

The 2 methods above are *encapsulating* data with structures:

```
template <class T>
struct Node {
    T data; // or `T *data`
    struct Node *next;
};
```

What if we add structures to data without encapsulation?

```
struct Node {
    Node *next;
};

struct MyData {
    int foo;    // data
    Node node;  // embedded structure
    // more data ...
};
```

`struct Node` contains no data, instead the data includes the structure. How is this generic? Example:

```
size_t list_size(struct Node *node) {
    size_t cnt = 0;
    for (; node != NULL; node = node->next) {
        cnt++;
    }
    return cnt;
}
```

`list_size()` touches no data; it operates on the sole node type `struct Node`. Data structures are about structures, not data types, so data structure code can be made generic by not including the data in it.

In contrast to the encapsulated data approach:

```
template <class T>
size_t list_size(struct Node<T> *node);
```

The templated function depends on the data type T, even if it's not used.

Using intrusive data structures

Embedding "dataless" structures into the data type is called *intrusive data structures*, because you need to modify your data type to use it. Now that the data structure is free of data, to get the data back, just offset the address of the struct.

```
Node *pnode = some_data_structure_code();
MyData *pdata = (MyData *)((char *)pnode - offsetof(MyData, node));
```

Make this less verbose and error-prone with a macro:

```
#define container_of(ptr, T, member) \
    ((T *)( (char *)ptr - offsetof(T, member) ))

MyData *pdata = container_of(pnode, MyData, node);
```

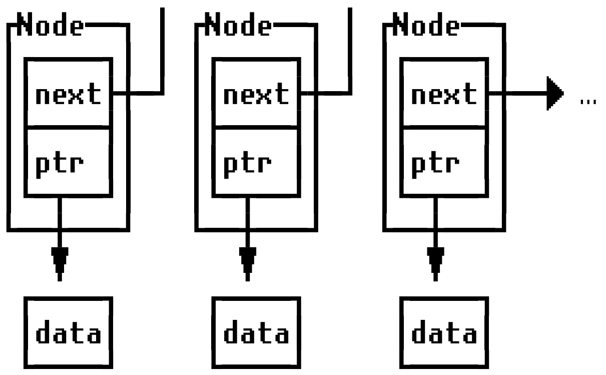
container_of is the de factor standard for intrusive data structures. "Container" refers to the data type containing the structure. In Linux kernel, container_of is defined as:

```
#define container_of(ptr, T, member) ({                 \
    const typeof( ((T *)0)->member ) *__mptr = (ptr);    \
    (T *)( (char *)__mptr - offsetof(T, member) ); })
```

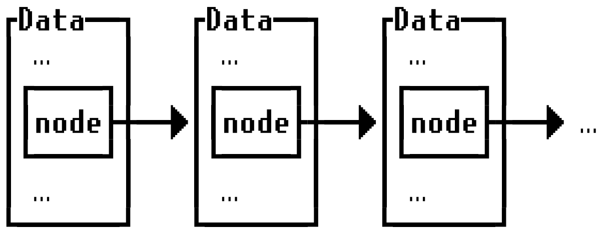
It has an extra type check to ensure that ptr matches T::member with the GCC extension typeof. The ({...}) syntax is a GCC extension that allows statements within an expression.

Advantage 1: Fast data access without indirections

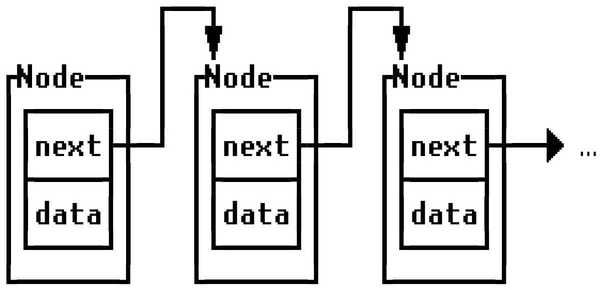
The void * pointer requires an extra pointer chasing to access the data:



Intrusive data structures avoid the extra indirection because the structure and the data are the same node.



From the computer’s point of view, this is as good as templated code:



Advantage 2: Memory management is minimized

Manual memory management is error-prone, so the less of it the better. Intrusive data structure code doesn’t need to allocate structure nodes because it borrows space from the data; how the data is allocated doesn’t even matter.

For our hashtable code, we only need to manage the array of slots, not the linked list nodes.

Advantage 3: Share data nodes between multiple collections

It’s possible for a data node to belong to multiple data structures.

```
struct MultiIndexedData {
    // data...
    HashNode node1; // embedded structure
    TreeNode node2; // another structure
};
```

For example, Redis sorted set is indexed both by name and by score, requiring 2 data structures. With non-intrusive data structures, one index contains the data, the other index contains a pointer to the data, requiring an extra allocation and indirection. Whereas with intrusive data structures, you can add many data structures to a single data node. This flexibility is unique to intrusive data structures.

Advantage 4: Multiple data types in the same collection

While intrusive data structures are not type-safe, they has the flexibility to use different data types in the same collection, as long as you have a way to distinguish between them. For example, you can use a different data type as the first linked list item.

8.5 Summary of hashtables

- 2 classes of hashtables: open addressing and chaining.
- Principles of chaining hashtables: array of collections.
- Challenges of using hashtables at scale: worst case latency.
- Intrusive data structures for making generic data structures.

That’s all you need to know. You can start coding, or see the next chapter for the code.

Source code:

> - 08_server.cpp
> - hashtable.cpp
> - hashtable.h