# 08. Data Structure: Hashtables

## 8.1 Introduction

This chapter fills the placeholder code in the last chapter's server. We'll start by implementing a hashtable. Hashtables are often the obvious data structure for holding an unknown amount of key-value data that does not require ordering.

There are two kinds of hashtables: chaining and open addressing. Their primary difference is collision resolution. Open addressing seeks another free slot in the event of a collision while chaining simply groups conflicting keys with a linked list. There are many variants of open addressing due to the need to find free slots, while the chaining hashtable is pretty much a fixed design.

The hashtable used in our server is a chaining one. A chaining hashtable is easy to code; it doesn't require much choice-making.

The definition of our data types:

```
// hashtable node, should be embedded into the payload
struct HNode {
    HNode *next = NULL;
    uint64_t hcode = 0;
};

// a simple fixed-sized hashtable
struct HTab {
    HNode **tab = NULL;
    size_t mask = 0;
```

```
    size_t size = 0;
};
```

## 8.2 Query, Insertion and Deletion

When the size of the hashtable is the power of two, the indexing operation is a simple bit mask with the hash code.

```
// n must be a power of 2
static void h_init(HTab *htab, size_t n) {
    assert(n > 0 && ((n - 1) & n) == 0);
    htab->tab = (HNode **)calloc(sizeof(HNode *), n);
    htab->mask = n - 1;
    htab->size = 0;
}

// hashtable insertion
static void h_insert(HTab *htab, HNode *node) {
    size_t pos = node->hcode & htab->mask;
    HNode *next = htab->tab[pos];
    node->next = next;
    htab->tab[pos] = node;
    htab->size++;
}
```

The lookup subroutine is simply a list traversal:

```
// hashtable look up subroutine.
// Pay attention to the return value. It returns the address of
// the parent pointer that owns the target node,
// which can be used to delete the target node.
static HNode **h_lookup(
    HTab *htab, HNode *key, bool (*cmp)(HNode *, HNode *))
{
    if (!htab->tab) {
        return NULL;
    }

    size_t pos = key->hcode & htab->mask;
    HNode **from = &htab->tab[pos];
    while (*from) {
```

```
        if (cmp(*from, key)) {
            return from;
        }
        from = &(*from)->next;
    }
    return NULL;
}
```

Deleting is easy. Notice how the use of pointers enables succinct code. The `from` pointer can be either an item of the array or from a node, yet the code doesn't differentiate.

```
// remove a node from the chain
static HNode *h_detach(HTab *htab, HNode **from) {
    HNode *node = *from;
    *from = (*from)->next;
    htab->size--;
    return node;
}
```

### 8.3 Progressive Resizing

Our hashtable is fixed in size, we need to migrate to a bigger one when the load factor is too high. There is an extra consideration when using hashtables in Redis. Resizing a large hashtable requires moving a lot of nodes to a new table, which can stall the server for some time. This shall be avoided by not moving everything at once, instead, we keep two hashtables and gradually move nodes between them.

Here is the final hashtable interface:

```
// the real hashtable interface.
// it uses 2 hashtables for progressive resizing.
struct HMap {
    HTab ht1;
    HTab ht2;
    size_t resizing_pos = 0;
};
```

The lookup subroutine now help with resizing:

```c
HNode *hm_lookup(
    HMap *hmap, HNode *key, bool (*cmp)(HNode *, HNode *))
{
    hm_help_resizing(hmap);
    HNode **from = h_lookup(&hmap->ht1, key, cmp);
    if (!from) {
        from = h_lookup(&hmap->ht2, key, cmp);
    }
    return from ? *from : NULL;
}
```

The `hm_help_resizing` function is the subroutine for gradually moving nodes:

```c
const size_t k_resizing_work = 128;

static void hm_help_resizing(HMap *hmap) {
    if (hmap->ht2.tab == NULL) {
        return;
    }

    size_t nwork = 0;
    while (nwork < k_resizing_work && hmap->ht2.size > 0) {
        // scan for nodes from ht2 and move them to ht1
        HNode **from = &hmap->ht2.tab[hmap->resizing_pos];
        if (!*from) {
            hmap->resizing_pos++;
            continue;
        }

        h_insert(&hmap->ht1, h_detach(&hmap->ht2, from));
        nwork++;
    }

    if (hmap->ht2.size == 0) {
        // done
        free(hmap->ht2.tab);
        hmap->ht2 = HTab{};
    }
}
```

The insertion subroutine will trigger resizing should the table become too full:

```
const size_t k_max_load_factor = 8;

void hm_insert(HMap *hmap, HNode *node) {
    if (!hmap->ht1.tab) {
        h_init(&hmap->ht1, 4);
    }
    h_insert(&hmap->ht1, node);

    if (!hmap->ht2.tab) {
        // check whether we need to resize
        size_t load_factor = hmap->ht1.size / (hmap->ht1.mask + 1);
        if (load_factor >= k_max_load_factor) {
            hm_start_resizing(hmap);
        }
    }
    hm_help_resizing(hmap);
}

static void hm_start_resizing(HMap *hmap) {
    assert(hmap->ht2.tab == NULL);
    // create a bigger hashtable and swap them
    hmap->ht2 = hmap->ht1;
    h_init(&hmap->ht1, (hmap->ht1.mask + 1) * 2);
    hmap->resizing_pos = 0;
}
```

The subroutine for removing a key. Nothing interesting.

```
HNode *hm_pop(
    HMap *hmap, HNode *key, bool (*cmp)(HNode *, HNode *))
{
    hm_help_resizing(hmap);
    HNode **from = h_lookup(&hmap->ht1, key, cmp);
    if (from) {
        return h_detach(&hmap->ht1, from);
    }
    from = h_lookup(&hmap->ht2, key, cmp);
    if (from) {
        return h_detach(&hmap->ht2, from);
```

```
    }
    return NULL;
}
```

## 8.3 Intrusive Data Structures

The hashtable implementation is done. Let's add them to the server. Looking at the `struct HNode` again, this structure contains no data, how do we actually use that? The answer is called "intrusive data structure":

```
// the structure for the key
struct Entry {
    struct HNode node;
    std::string key;
    std::string val;
};
```

Instead of making our data structure contain data, the hashtable node structure is embedded into the payload data. This is the standard way of creating generic data structures in C.

Besides making the data structure fully generic, this technique also has the advantage of reducing unnecessary memory management. The structure node is not separately allocated but is part of the payload data, and the data structure code does not own the payload but merely organizes the data. This may be quite a new idea to you if you learned data structures from textbooks, which is probably using `void *` or C++ templates or even macros.

Listing the `do_get` function to see how the intrusive data structure is used:

```
// The data structure for the key space.
static struct {
    HMap db;
} g_data;

static uint32_t do_get(
    std::vector<std::string> &cmd, uint8_t *res, uint32_t *reslen)
{
    Entry key;
```

```
    key.key.swap(cmd[1]);
    key.node.hcode = str_hash((uint8_t *)key.key.data(), key.key.size());

    HNode *node = hm_lookup(&g_data.db, &key.node, &entry_eq);
    if (!node) {
        return RES_NX;
    }

    const std::string &val = container_of(node, Entry, node)->val;
    assert(val.size() <= k_max_msg);
    memcpy(res, val.data(), val.size());
    *reslen = (uint32_t)val.size();
    return RES_OK;
}

static bool entry_eq(HNode *lhs, HNode *rhs) {
    struct Entry *le = container_of(lhs, struct Entry, node);
    struct Entry *re = container_of(rhs, struct Entry, node);
    return lhs->hcode == rhs->hcode && le->key == re->key;
}
```

The `hm_lookup` function returns a pointer to `HNode`, which is a member of the `Entry`, we need some pointer arithmetics to convert that pointer to an `Entry` pointer. The `container_of` macro is commonly used in C projects for this purpose:

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

The `do_set` and `do_del` are both trivial.

```
static uint32_t do_set(
    std::vector<std::string> &cmd, uint8_t *res, uint32_t *reslen)
{
    (void)res;
    (void)reslen;

    Entry key;
    key.key.swap(cmd[1]);
    key.node.hcode = str_hash((uint8_t *)key.key.data(), key.key.size());
```

```
    HNode *node = hm_lookup(&g_data.db, &key.node, &entry_eq);
    if (node) {
        container_of(node, Entry, node)->val.swap(cmd[2]);
    } else {
        Entry *ent = new Entry();
        ent->key.swap(key.key);
        ent->node.hcode = key.node.hcode;
        ent->val.swap(cmd[2]);
        hm_insert(&g_data.db, &ent->node);
    }
    return RES_OK;
}

static uint32_t do_del(
    std::vector<std::string> &cmd, uint8_t *res, uint32_t *reslen)
{
    (void)res;
    (void)reslen;

    Entry key;
    key.key.swap(cmd[1]);
    key.node.hcode = str_hash((uint8_t *)key.key.data(), key.key.size());

    HNode *node = hm_pop(&g_data.db, &key.node, &entry_eq);
    if (node) {
        delete container_of(node, Entry, node);
    }
    return RES_OK;
}
```

Exercises:

1. Our hashtable triggers resizing when the load factor is too high, should we also shrink the hashtable when the load factor is too low? Can the shrinking be performed automatically?

Source code:

> - [08_server.cpp]()
> - [hashtable.cpp]()
> - [hashtable.h]()

See also:

**codecrafters.io** offers “Build Your Own X” courses in many programming languages. Including Redis, Git, SQLite, Docker, and more.

Check it out