CHAPTER

# 8b. Hashtables (Part 2)

## 8.6 Fixed-size chaining hashtable

We'll break the implementation into 2 phases:

1. Fixed-size hashtable with intrusive linked lists.
2. Resizable hashtable with progressive rehashing.

### Step 0: Choose a hash function

There are several classes of hash functions with different properties for different uses:

- Cryptographic hash functions, such as MD5, SHA1.
- Checksum hash functions, such as CRC32, Adler32.
- Hash functions for hashtables, such as FNV, Murmur. This is what to use!

They are all called hash functions, but their uses do not overlap. Do not use cryptographic hash functions for hashtables because they are slow and overkill. Do not use hashtable hash functions for anything else because they are not strong enough.

### Step 1: Define the intrusive list node

```
struct HNode {
    HNode *next = NULL;
    uint64_t hcode = 0; // hash value
};
```

Just an intrusive linked list node with the hash value of the key. An intrusive hashtable doesn't care about the data, but it still needs the hash value for the insertion.

### Step 2: Define the fixed-size hashtable

```
struct HTab {
    HNode **tab = NULL; // array of slots
```

```
    size_t mask = 0;    // power of 2 array size, 2^n - 1
    size_t size = 0;    // number of keys
};
```

`hash(key) % N` maps a hash value to a slot. Modulo or division are slow CPU operations, so it's common to use powers of 2 sizes. Modulo by a power of 2 is just taking the lower bits, so it can be done by a fast bitwise and: `hash(key) & (N-1)`.

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

### Step 3: Linked list insertion

```
static void h_insert(HTab *htab, HNode *node) {
    size_t pos = node->hcode & htab->mask;  // node->hcode & (n - 1)
    HNode *next = htab->tab[pos];
    node->next = next;
    htab->tab[pos] = node;
    htab->size++;
}
```

New nodes are inserted at the front, so insertion is $O(1)$ in the worst case. Being intrusive means that there's no allocation in the data structure code, unlike `std::list`.

Linked lists are the most common collections in large C projects like the Linux kernel, and **almost all practical linked lists are intrusive**, non-intrusive linked lists like `std::list` are rarely useful.

### Step 6: Hashtable lookup

The generic search function needs a callback to do the equality comparison, as it knows nothing about the data. Note that it compares the hash value before calling the callback, this is an optimization to rule out candidates early.

```
static HNode **h_lookup(HTab *htab, HNode *key, bool (*eq)(HNode *, HNode *)) {
    if (!htab->tab) {
        return NULL;
    }
    size_t pos = key->hcode & htab->mask;
    HNode **from = &htab->tab[pos];     // incoming pointer to the target
    for (HNode *cur; (cur = *from) != NULL; from = &cur->next) {
        if (cur->hcode == key->hcode && eq(cur, key)) {
            return from;    // Q: Why not return `cur`? This is for deletion.
        }
    }
    return NULL;
}
```

The function should return the node found, but it actually returns the **address of its parent pointer**, requiring a pointer dereference to get the target pointer. Why? Because we need the address of the pointer to delete it.

### Step 7: Hashtable deletion

To remove a linked list node, we need to update the pointer in its parent node. That's why `h_lookup()` returns the address of the parent pointer. No need for a separate search-and-delete function!

```
static HNode *h_detach(HTab *htab, HNode **from) {
    HNode *node = *from;    // the target node
    *from = node->next;     // update the incoming pointer to the target
    htab->size--;
    return node;
}
```

But what if we remove the first node without a parent node? This seems to be a special case.

- If the node is the first node, update the array slot.
- Otherwise, update its parent node.

The above cases are unnecessary, because `h_lookup()` returns the address of the to-be-updated pointer, it doesn't matter if it's from a slot or a node.

```
    HNode **from = &htab->tab[pos];     // incoming pointer to the target
    for (HNode *cur; (cur = *from) != NULL; from = &cur->next) {
        if (cur->hcode == key->hcode && eq(cur, key)) {
            return from;                // may be a node, may be a slot
        }
    }
```

When you think harder, special cases may not be special at all.

- Removing the first list node is not special.
- No need for a search-and-delete function.

## 8.7 Resizable hashtable

### Step 8: Define the hashtable interfaces

The resizable `HMap` is based on the fixed-size `HTab`. It contains 2 of them for the progressive rehashing.

```
struct HMap {
    HTab newer;
    HTab older;
    size_t migrate_pos = 0;
};
```

The get, set, del interfaces:

```
HNode *hm_lookup(HMap *hmap, HNode *key, bool (*eq)(HNode *, HNode *));
void   hm_insert(HMap *hmap, HNode *node);
HNode *hm_delete(HMap *hmap, HNode *key, bool (*eq)(HNode *, HNode *));
```

### Step 9: Deal with 2 hashtables during rehashing

Normally, `HMap::newer` is used while `HMap::older` is unused. When the load factor is too high, `HMap::newer` is moved to `HMap::older`, and `HMap::newer` is replaced by a larger, empty table. We used power of 2 sizes, so the newer table is simply doubled.

```
static void hm_trigger_rehashing(HMap *hmap) {
```

```
    hmap->older = hmap->newer;  // (newer, older) <- (new_table, newer)
    h_init(&hmap->newer, (hmap->newer.mask + 1) * 2);
    hmap->migrate_pos = 0;
}
```

During rehashing, lookup or delete may need to query both tables.

```
HNode *hm_lookup(HMap *hmap, HNode *key, bool (*eq)(HNode *, HNode *)) {
    HNode **from = h_lookup(&hmap->newer, key, eq);
    if (!from) {
        from = h_lookup(&hmap->older, key, eq);
    }
    return from ? *from : NULL;
}


HNode *hm_delete(HMap *hmap, HNode *key, bool (*eq)(HNode *, HNode *)) {
    if (HNode **from = h_lookup(&hmap->newer, key, eq)) {
        return h_detach(&hmap->newer, from);
    }
    if (HNode **from = h_lookup(&hmap->older, key, eq)) {
        return h_detach(&hmap->older, from);
    }
    return NULL;
}
```

### Step 10: Trigger rehashing by the load factor

Insertions always update the newer table. It triggers rehashing when the load factor is high.

```
void hm_insert(HMap *hmap, HNode *node) {
    if (!hmap->newer.tab) {
        h_init(&hmap->newer, 4);    // initialized it if empty
    }
    h_insert(&hmap->newer, node);   // always insert to the newer table
    if (!hmap->older.tab) {         // check whether we need to rehash
        size_t shreshold = (hmap->newer.mask + 1) * k_max_load_factor;
        if (hmap->newer.size >= shreshold) {
            hm_trigger_rehashing(hmap);
```

```
        }
    }
    hm_help_rehashing(hmap);        // migrate some keys
}
```

For a chaining hashtable, the load factor limit should be greater than 1 because each slot is intended to hold multiple items.

```
const size_t k_max_load_factor = 8;
```

`hm_help_rehashing()` moves some keys to the newer table. We can also trigger this from lookup and delete, it doesn't hurt as it does $O(1)$ work.

```
HNode *hm_lookup(HMap *hmap, HNode *key, bool (*eq)(HNode *, HNode *)) {
    hm_help_rehashing(hmap);
    // ...
}
HNode *hm_delete(HMap *hmap, HNode *key, bool (*eq)(HNode *, HNode *)) {
    hm_help_rehashing(hmap);
    // ...
}
```

### Step 11: Progressive rehashing

`hm_help_rehashing()` scans each slot, moves a constant number of items, and then exits. It remembers the last slot index in `HMap::migrate_pos`.

```
const size_t k_rehashing_work = 128;    // constant work


static void hm_help_rehashing(HMap *hmap) {
    size_t nwork = 0;
    while (nwork < k_rehashing_work && hmap->older.size > 0) {
        // find a non-empty slot
        HNode **from = &hmap->older.tab[hmap->migrate_pos];
        if (!*from) {
            hmap->migrate_pos++;
            continue;   // empty slot
```

```
            }
            // move the first list item to the newer table
            h_insert(&hmap->newer, h_detach(&hmap->older, from));
            nwork++;
        }
        // discard the old table if done
        if (hmap->older.size == 0 && hmap->older.tab) {
            free(hmap->older.tab);
            hmap->older = HTab{};
        }
    }
```

## 8.8 Redis command: get, set, del

### Step 1: Define data types

```
static struct {
    HMap db;    // top-level hashtable
} g_data;

// KV pair for the top-level hashtable
struct Entry {
    struct HNode node;  // hashtable node
    std::string key;
    std::string val;
};
```

`struct Entry` is the KV pair with an embedded hashtable node. The equality callback shows how to use the `container_of` macro.

```
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 le->key == re->key;
}
```

### Step 2: Implement get, set, del

We used a dummy `Entry` with just a key in it for the comparisons during the lookup.

```
static void do_get(std::vector<std::string> &cmd, Response &out) {
    // a dummy `Entry` just for the lookup
    Entry key;
    key.key.swap(cmd[1]);
    key.node.hcode = str_hash((uint8_t *)key.key.data(), key.key.size());
    // hashtable lookup
    HNode *node = hm_lookup(&g_data.db, &key.node, &entry_eq);
    if (!node) {
        out.status = RES_NX;
        return;
    }
    // copy the value
    const std::string &val = container_of(node, Entry, node)->val;
    assert(val.size() <= k_max_msg);
    out.data.assign(val.begin(), val.end());
}
```

Note that the key doesn't need to be an `Entry`, the key can be other types that embed an `HNode`, as long as you update `entry_eq()` accordingly. Right now `Entry` is just a KV pair, it will get larger as we go. We can define a leaner struct for lookups.

```
struct LookupKey {  // for lookup only
    HNode node;
    std::string key;
};

LookupKey key = {...};
HNode *node = hm_lookup(&g_data.db, &key.node, &key_eq);
```

## 8.8 What we learned

- Scalable hashtable implementations.
- Avoid special cases via pointer indirection in linked list code.

Source code:

> - 08_server.cpp

- hashtable.cpp
- hashtable.h