Implementing Hash Tables in C

October 2, 2021

NOTE(s): The article is in “draft” status. The content was originally written in 2017.

The intended audience for this article is undergrad students or seasoned developers who want to refresh their knowledge on the subject.

The reader should already be familiar with C (pointers, pointer functions, macros, memory management) and basic data structures knowledge (e.g., arrays, linked lists, and binary trees).

Table of contents

Code

If you don't want to read the article, and you just want to jump directly into the code for:

- Separate Chaining (https://github.com/nomemory/chained-hash-table-c): `git clone git@github.com:nomemory/chained-hash-table-c.git`
- Open Addressing (https://github.com/nomemory/open-adressing-hash-table-c): `git clone git@github.com:nomemory/open-adressing-hash-table-c.git`

> *The code needs to be compiled with the `c99` flag.*

Introduction

> *Outside the domain of computers, the word **hash** means to **chop/mix** something.*

In Computer Science, a *hash table* is a fundamental data structure that associates a **set of keys** with a **set of values**. Each pair `<key, value>` is an entry in our **hash table**. Given a **key**, we can get the **value**. Not only that, but we can add and remove `<key, value>` pairs whenever it is needed.

> *Not be confused with hash trees (https://en.wikipedia.org/wiki/Hash_tree) or hash lists (https://en.wikipedia.org/wiki/Hash_list).*

In a way, a hash table share some similarities with the average "array", so let's look at the following code:

```
int arr[] = {100, 200, 300};
printf("%d\n", arr[1]);
```

If we were to run it, the output would be `200`. As we write `arr[<index>]`, we are *peeping* at the value associated with the given `<index>`, and in our case, the value associated with `1` is `200`.

In this regard, a hash table can act very similar to an array, because it will allow us to map a **value** to a given **key**. But there's a catch, compared to an array, the **key** can be *everything* - we are not limited to sorted numerical indexes.

Most modern computer programming languages have a hash table implementation in their standard libraries. The names can be different, but the results are the same.

For example in C++, we have something called `std::unorderd_map` (https://www.cplusplus.com/reference/unordered_map/unordered_map/):

```
#include <iostream>
#include <string>
#include <unordered_map>

int main() {

    std::unordered_map<std::string, std::string> colors = {
        /* associates to the key: */ {"RED", /* the value: */ "#FF0000"},
        {"GREEN", "#00FF00"},
        {"BLUE", "#0000FF"}
    };

    std::cout << colors["RED"] << std::endl;
}
// Output: #FF0000
```

In the Java world, there's `HashMap`:

```java
import java.util.Map;
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        Map<String, String> colors = new HashMap<>();

        colors.put("RED", "#FF0000");
        colors.put("GREEN", "#00FF00");
        colors.put("BLUE", "#0000FF");

        System.out.println(colors.get("RED"));
    }
}
// Output: #FF0000
```

Or in languages like python, support for hash tables is “built-in” in the language itself:

```python
colors = {
        "RED" :  "#FF0000",
        "GREEN" : "#00FF00",
        "BLUE" :"#0000FF"
}

print(colors["RED"])
# Output: #FF0000
```

So regardless of the name (`unordered_map` in C++, `HashMap` in Java, or `dict` in python), if we run all three code snippets, the result will be the same: `"#FF0000"` .

> *Size doesn’t matter.*

What is impressive about hash tables is they are very “efficient” in terms of (average) time complexities. Given a **key**, we can return the **value** associated with it in(almost) constant time, no matter how many pairs we have previously inserted. So they scale exceptionally well with size. In a way, for a **hash table** “size doesn’t matter”, as long as we keep things under control:

Think of it for a moment. For a binary tree, searching for an element is `O(logn)` ; if the trees grow, `n` grows, so searching for an element takes more time. But No!, not for hash tables. As long as we know the key, we can have almost instant access to the stored value.

Operation	Average Time Complexity	Worst-case scenario
Getting a **value**	`O(1)`	`O(n)`
Adding a new pair `<key, value>`	`O(1)`	`O(n)`
Updating a pair `<key, value>`	`O(1)`	`O(n)`
Removing a pair `<key, value>`	`O(1)`	`O(n)`

This remarkable data structure is internally powered by clever mathematical functions: called **hash functions**. They are the “force” behind the fantastic properties of hash tables: the ability to scale even if the number of pairs increases.

At this point, it wouldn’t be wise to jump directly to the implementation of a hash table, so we will make a short mathematical detour into the wonderful world of **hash functions**. They are less scary than you would typically expect, well, at least on the surface.

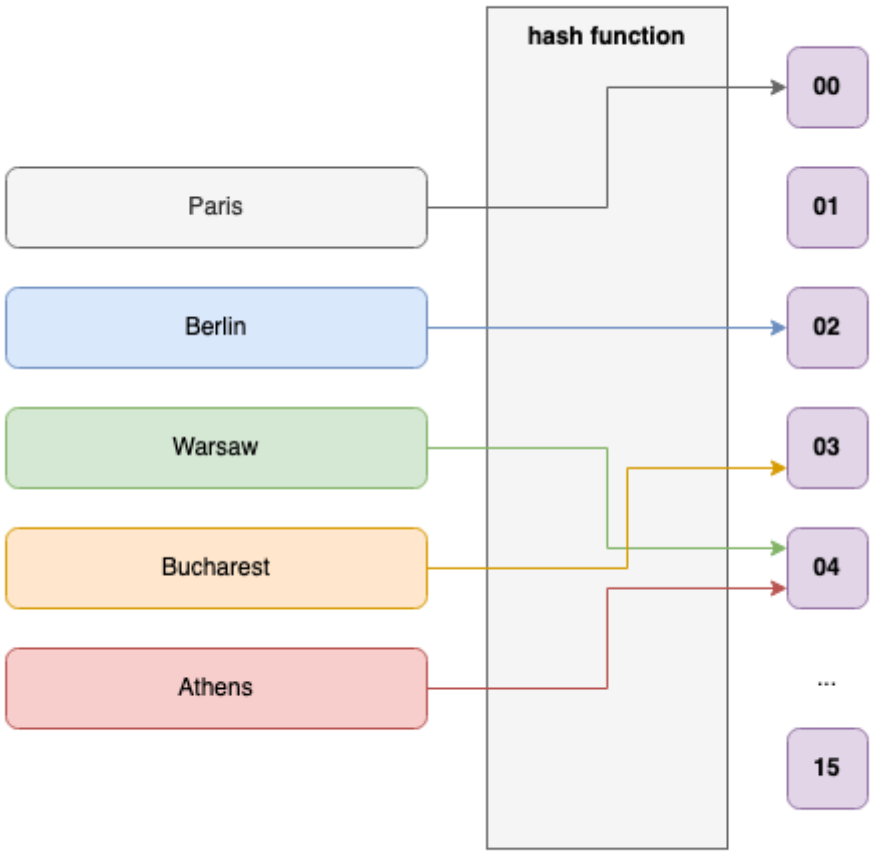
Hash Functions

In *English*, A **hash function** is a function that “chops” data of arbitrary size to data of fixed size.

From a mathematical perspective, A **hash function** is a function $H : X \to [0, M)$, that takes an element in $x \in X$ and associates to it a positive integer $H(x) = m$, where $m \in [0, M)$.

X can be bounded or an *un*-bounded set of values, while M is always a positive finite number $0 < M < \infty$.

Let’s take a look at the following example:



In the above diagram `Paris`, `Berlin`, `Warsaw`, `Bucharest`, `Athens` are all capitals that $\in X$, where X is a set containing all European Capitals, so $n(X) = 48$.

Our hashing function is $H : X \to [0, 16)$, so that:

- `H("Paris")` returns `00`;
- `H("Berlin)` returns `01`;
- `H("Warsaw)` returns `04`;
- `H("Bucharest)` returns `03`;
- `H("Athens")` returns `04`;

In total, there are around 48 countries in Europe, each with its capital. So the number of elements in X is 48, while $M = 16$ so that the possible values are in the interval $[0, 1, 2, \dots 16)$.

Because $48 > 16$, no matter how we write H, some European Capital Cities will share the same number $m \in [0, 1, \dots 16)$.

For our hypothetical example, we can see that happening for `H("Warsaw) = H("Athens") = 4`.

Whenever we have two elements $x_1, x_2 \in X$ so that $H(x_1) = H(x_2) = m_x$, we say we have a **hash collision**.

In our example we can say that H has a **hash collision** for $x_1 = Warsaw$ and $x_2 = Athens$, because $H(x_1) = H(x_2) = 4$.

Hash Collisions are not game-breaking per se, as long as the $n(X) > M$ they might happen. But it's important to understand that a *good* **hash function** creates fewer **hash collisions** than a *bad* one.

Basically the worst hash function we can write is a function that returns a constant value, so that $H(x_1) = H(x_2) = \dots = H(x_n) = c$, where $n = n(X)$, and $c \in [0, M)$. This is just another way of saying that every element $x \in X$ will collide with the others.

Another way we can shoot ourselves in the foot is to pick a **hash function** that is not deterministic. When we call $H(x)$ subsequently, it should render the same results without being in any way affected by external factors.

> ***Cryptographic hash functions*** *are a special family of* ***hash functions****. For security considerations, they exhibit an extra set of properties. The functions used in* ***hash table*** *implementations are significantly less pretentious.*

Another essential aspect when picking the right **hash function** is to pick something that it's not computationally intensive. Preferably it should be something close to `O(1)`. **Hash tables** are using **hash functions** intensively, so we don't want to complicate things too much.

Picking the proper hash function is a tedious job. It involves a lot of statistics, number theory, and empiricism, but generally speaking, when we look for a hash function, we should take into consideration the following requirements:

- It should have a reduced number of collisions;
- It should disperse the hashes uniformly in the $[0, M)$ interval;
- It should be fast to compute;

We can talk about "families" of hashing functions. On the one hand, you have **cryptographic hash functions** that are computationally complex and have to be resistant to preimage attacks (https://en.wikipedia.org/wiki/Preimage_attack) (that's a topic for another day). Then you have simpler **hash functions** that are suitable to be used to implement **hash tables**:

- Cryptographic hash functions;
- All-around / general functions used for **hash table** implementations:
    - Division hashing;
    - Bit shift hash functions
    - Multiplicative hashing;

The advanced math behind **hash functions** eludes me. I am a simple engineer, no stranger to math, but not an expert. There are PHD (https://en.wikipedia.org/wiki/Doctor_of_Philosophy) papers on the subject, and for a select group of people, this is their actual job: finding better and faster ways of **hashing** stuff. So, in the next two paragraphs, I will try to keep things as simple as possible by avoiding making things more mathematical than needed.

Division hashing

The simplest **hash function** that we can write the mod (https://en.wikipedia.org/wiki/Modulo_operation) `%` operation, and it's called **division hashing**.

It works on positive numbers, so let's suppose we can represent our initial input data $x_1, x_2, \dots x_n \in X \subset N$ as a non-negative integers.

Then, the formula for our hash function is $H_{division}(x) = x \mod M$, where $H_{division} : X \to [0, M)$. In English this means that the hash of a given value x is the remainder of x divided with M.

Writing this function in C is trivial:

```
uint32_t hashf_division(uint32_t x, uint32_t M) {
    return x % M;
};
```

On the surface, `hashf_division()` checks all the requirements for a good **hashing function**.

And it's a good hashing function as long as the input data $(x_1, x_2, \dots, x_n)$ is guaranteed to be perfectly random, without obvious numerical patterns.

So let's test how it behaves if we pick:

- `M=4`;
- `1000000` uniformly distributed positive integers as input (X)

Without writing any code, we would infer that all the input will be evenly distributed between 4 hash values: `0`, `1`, `2` and `3`. There are going to be collisions (as $n(X)$ is 250000 bigger than `4`), but theoretically, we can reduce them by increasing `M` further down the road.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define M_VAL 4
#define X_VAL 1000000

uint32_t hashf_division(uint32_t x, uint32_t M) {
    return x % M;
};

int main(int argc, char* argv[]){

    // initiate rand
    srand(time(0));

    unsigned int i, hash;

    // buckets
    int buckets[M_VAL];

    for(i = 0; i < X_VAL; i++) {
        hash = hashf_division(rand(), M_VAL);
        buckets[hash]++;
    }

    for(i = 0; i < M_VAL; i++) {
        printf("bucket[%u] has %u elements\n", i, buckets[i]);
    }
}
```

If we run the above program, a possible output would be:

```
bucket[0] has 250146 elements
bucket[1] has 249361 elements
bucket[2] has 250509 elements
bucket[3] has 249984 elements
```

The results are OK:

- All input has been evenly distributed between the four values (buckets);
- The `%` operation is quite efficient (although it's usually a number of times less efficient than multiplication);
- Collisions are there, but they can be controlled by increasing the value `M` to accommodate the input size.

> *Multiplication and division take longer time. Integer multiplication takes 11 clock cycles on Pentium 4 processors, and 3 - 4 clock cycles on most other microprocessors. Integer division takes 40 - 80 clock cycles, depending on the microprocessor. Integer division is faster the smaller the integer size on AMD processors, but not on Intel processors. Details about instruction latencies are listed in manual 4: "Instruction tables". Tips about how to speed up multiplications and divisions are given on page 146 and 147, respectively. (source (https://www.agner.org/optimize/optimizing_cpp.pdf))*

> *We call the resulting hashes **buckets**, and once we start implementing the actual hash table, this will make more sense.*

But what happens if our (input) data is not that random after all. What if the data follows a particular obvious (or not so obvious) pattern? How is this pattern going to affect the distribution of computed hashes?

Let's change this line:

```c
hash = hashf_division(rand(), M_VAL);
```

To this:

```c
hash = hashf_division(rand()&-2, M_VAL)
```

Now, all our *randomly generated* numbers will be even. If that's not an obvious pattern, I don't know what is.

Let's see how our hashing function behaves in this scenario and how well the hashes are distributed:

```
bucket[0] has 500810 elements
bucket[1] has 0 elements
bucket[2] has 499190 elements
bucket[3] has 0 elements
```

We see that values `1` and `3` are never used, which is unfortunate, but an expected consequence of how the input is constructed. If all the input numbers are even, then their remainder is either `0` or `2`.

Mathematically we can prove there's not a single $x_i \in X \subset N$, for our function $H_{division}(x) = x \mod 4$, where $H : X \to [0, 4)$, so that $H_{division}(x_i) = 1$ or $H_{division}(x_i) = 3$.

So this means our function is not so good after all because it's extremely sensitive to "input patterns"?

The answer is not that simple, but what's for sure is that changing `M=4` to `M=7` will render different results.

This time, the output will be:

```
bucket[0] has 142227 elements
bucket[1] has 143056 elements
bucket[2] has 143592 elements
bucket[3] has 142721 elements
bucket[4] has 142722 elements
bucket[5] has 142662 elements
bucket[6] has 143020 elements
```

The results look promising again. The hash values are again "evenly" distributed, but all kinds of (subtle) problems can appear based on our choice of `M`.

Normally, `M` should be a prime number, and some prime numbers will work in practice better than others, e.g.:

$$H'_{division}(x) = x \mod 127$$
$$H''_{division}(x) = x \mod 511$$
$$H'''_{division}(x) = x \mod 2311$$

To make the results even better, Donald Knuth (https://en.wikipedia.org/wiki/Donald_Knuth), proposed an alternative solution, where $H^{Knuth}_{division}(x) = x(x+3) \mod M$.

In practice, **division hashing** is not that commonly used. The reason is simple, even the results are satisfactory (especially when `M` is chosen wisely), division and modulo operations are more "expensive" than addition or multiplication.

Multiplicative hashing

A common (and practical) approach for generating relatively uniform hash values is called **multiplicative hashing**.

Similar to **division hashing**, the formula works for positive integers. So we assume that we have a mechanism that converts our input space to positive numbers only.

The following formula usually describes a **multiplicative hash** function:

$H_{multip}(x) = \frac{M}{W} * (Ax \mod W)$, where $H_{multip} : X \in N \to [0, M)$.

- $A \in R^+_*$ is a constant;
- M is usually a power of `2`, so that $M = 2^m$.
- $W = 2^w$, where w is the machine word size (https://en.wikipedia.org/wiki/Word_(computer_architecture)). In C we can look for the max value of an `unsigned int` in the header file `limits.h`: `UINT_MAX`. In this case `W=UINT_MAX+1`.

So our function becomes:

$$H_{multip}(x) = \frac{2^m}{2^w} * (Ax \mod 2^w) = 2^{m-w} * (Ax \mod 2^w) = \frac{Ax \mod 2^w}{2^{w-m}}$$

By the magic of bitwise operations, $Ax \mod 2^w$ is just a way of getting the w low order bits of Ax. Think of it for a second, `x % 2` returns the last bit of x, `x % 4` returns the last 2 bits of x, etc.

Dividing a number b by 2^{w-m} actually means shifting right `w-m` bits. So our method actually becomes:

$$H_{multip}(x) = A * x >> (w - m)$$

If we were to write this in C:

```
uint32_t hashf_multip(uint32_t x, uint32_t p, uint32_t w, uint32_t m, uint32_t A)  {
    return (x * A) >> (w-m);
}
```

I won't get into all the mathematical details, but a good choice for A is $A = \phi * 2^w$, where w is the word machine word size (https://en.wikipedia.org/wiki/Word_(computer_architecture)), and ϕ (phi) is the golden ratio (https://en.wikipedia.org/wiki/Golden_ratio).

ϕ, just like its more famous brother π (pi (https://en.wikipedia.org/wiki/Pi)), is an irrational number, $\phi \in Q'$, that is a solution to the equation: $x^2 - x - 1 = 0 => \phi \approx 0.61803398874989484820458683436 56$.

Depending on the word size, A can be computed as follows:

w	$A = 2^w * \phi \approx$
16	40,503
32	2,654,435,769
64	11,400,714,819,323,198,485

When we pick $A = \phi * 2^w$, our general **multiplicative hash function** is called a **Fibonacci hash function** (https://en.wikipedia.org/wiki/Hash_function#Fibonacci_hashing).

Having this said, we can now simplify the signature of our C function. We don't need `A`, `m`, `w` anymore as input params because they can be `#defined` as constants.

After the signature change, our function becomes:

```
#define hash_a (uint32_t) 2654435769
#define hash_w 32
#define hash_m 3

uint32_t hashf_multip(uint32_t x, uint32_t m) {
    return (x * hash_a) >> (hash_w - m);
}
```

As it's mentioned here (https://probablydance.com/2018/06/16/fibonacci-hashing-the-optimization-that-the-world-forgot-or-a-better-alternative-to-integer-modulo/), **Fibonacci Hashing** has one "small" issue. Poor diffusion happens as higher-value bits do not affect lower-value bits. In this regard, it can further be improved by shifting the span of retained higher bits and then `XOR` ing them to the key before the actual **hash multiplication** happens:

```
#define hash_a (uint32_t) 2654435769
#define hash_w 32
#define hash_m 3

uint32_t hashf_multip2(uint32_t x, uint32_t m) {
    x ^= x >> (hash_w - m);
    return (x * hash_a) >> (hash_w - m);
}
```

With `hashf_multip2()`, we achieve better diffusion with a price: more operations.

Hashing strings

Converting non-numerical data to positive integers (`uint32_t`) is quite simple. After all, everything is a sequence of bits.

In K&R Book (https://en.wikipedia.org/wiki/The_C_Programming_Language) (1st ed) a simple (and ineffective) hashing algorithm was proposed: *What if sum the numerical values of all characters from a string?*

```
uint32_t hashf_krlose(char *str) {
    uint32_t hash = 0;
    char c;
    while((c=*str++)) {
        hash+=c;
    }
    return hash;
}
```

Unfortunately, the output for `hashf_krlose` will be "extremely" sensitive to input patterns. It's easy to apply a little **Gematria** (https://en.wikipedia.org/wiki/Gematria) ourselves to create input that will repeatedly return identical hashes.

For example:

```
char* input[] = { "IJK", "HJL", "GJM", "FJN" };
uint32_t i;
for(i = 0; i < 4; i++) {
    printf("%d\n", hashf_krlose(input[i]));
}
```

The hash values for `"IJK"`, `"HJL"`, `"GJM"`, `"FJN"` are all `222`.

A proposal to improve the function is to replace `+=` (summing) with `^=` (XORing), so that `hash+=c` becomes `hash^=c`. But again, patterns that break our **hash function** are easy to create, so it doesn't make a big practical difference.

The good news is that there's a common type of **hash function** that works quite well with strings (`char*`).

The *template* for those functions look like:

```
#define INIT <some_value>
#define MULTIP <some_value>

uint32_t hashf_generic(char* str) {
    uint32_t hash = INIT;
    char c;
    while((c*=str++)) {
        hash = MULTIP * hash + c;
    }
    return hash;
}
```

djb2

If `INIT=5381` and `MULT=33`, the function is called **Bernstein hash djb2**, which dates 1991.

> *If you find better values, chances are your name will remain in the history books of Computer Science.*

Implementing it in C is quite straight-forward:

```
#define INIT 5381
#define MULT 33

uint32_t hashf_djb2_m(char *str) {
    uint32_t hash = INIT;
    char c;
    while((c=*str++)) {
        hash = hash * MULT + c;
    }
    return hash;
}
```

If you look over the internet for **djb2**, you will find a different implementation that uses one clever, simple trick. The code would be:

```
#define INIT 5381

uint32_t hashf_djb2(char *str) {
    uint32_t hash = INIT;
    char c;
    while((c=*str++)) {
        hash = ((hash << 5) + hash) + c;
    }
    return hash;
}
```

If we write `a << x`, we are shifting `x` bits of `a` to the left. By the magic of bitwise operations, this is equivalent to multiplying `a * 2^x`.

So, our expression `hash = ((hash << 5) + hash) + c` is equivalent to `hash = (hash * 2^5) + hash + c`, that is equivalent to `hash = hash * (2^5 + 1) + c`, that is equivalent `hash = hash * 33 +c`.

This is **not** just a fancy way of doing things. Historically speaking, most CPUs were performing bitshifts faster than multiplication or division. They still do.

In modern times, modern compilers can perform all kinds of optimizations, including this one. So it's up to you to decide if making things harder to read is worth it. Again, some benchmarking is recommended.

sdbm

If you search the internet for **sdbm** you won't find a lot of details, except (http://www.cse.yorku.ca/~oz/hash.html):

> *This algorithm was created for sdbm (a public-domain re-implementation of ndbm) database library. It was found to do well in scrambling bits, causing a better distribution of the keys and fewer splits. It also happens to be a good general hashing function with good distribution.*

The function looks like this: z

```
uint32_t hashf_sdbm(char *str) {
    uint32_t hash = 0;
    char c;
    while((c=*str++)) {
        hash = c + (hash << 6) + (hash << 16) - hash;
    }
    return hash;
}
```

Reducing the number of collisions - Finalizers

After publishing the article draft on reddit/r/C_Programming (https://www.reddit.com/r/C_Programming/comments/q88m49/implementing_hash_tables_in_c_an_article_ive/), the community (u/skeeto) suggested the **hash functions** for strings can be improved even further with a simple trick: *finalizers*.

A common *finalizer* found in MurmurHash (https://en.wikipedia.org/wiki/MurmurHash), looks like:

```
uint32_t ch_hash_fmix32(uint32_t h) {
    h ^= h >> 16;
    h *= 0x3243f6a9U;
    h ^= h >> 16;
    return h;
}
```

In practice, we can improve results generated by djb2, sdbm by applying `ch_hash_fmix32()` on their output:

```
uint32_t final_hash = ch_hash_fmix32(hashf_sdbm("some string");
```

More exploration

What we've discussed so far about **hash functions** only scratches the surface of a vast subject. There are hundreds of functions, some of them better than others. But, as Software Engineers, we need to be pragmatic, know about them, ponder over their pros and cons and, in the end, take things for granted. Most of the time, simple is better.

For high(er)-level programming languages (C++, C#, Java, python), the "hashing" problem is already sorted out at "language" or "standard library" level, so we rarely have to write (or copy-paste) a hash function by hand.

If you want to explore this topic more, I suggest you also take a look at the following articles:

- FNV Hash (http://www.isthe.com/chongo/tech/comp/fnv/) - a popular **hash function** designed to be fast while maintaining a low collision rate;
- SipHash (https://github.com/skeeto/scratch/blob/master/siphash/siphash-embed.h#L8)
- Murmurhash (https://en.wikipedia.org/wiki/MurmurHash) - is a non-cryptographic hash function suitable for general hash-based lookup. Source code here (https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp).
- Zobrist Hashing (https://en.wikipedia.org/wiki/Zobrist_hashing) - a **hash function** used in computer programs that play abstract board games, such as chess and Go, to implement transposition tables, a special kind of hash table that is indexed by a board position and used to avoid analyzing the same position more than once.
- chunk64 (https://github.com/skeeto/scratch/blob/master/misc/chunky64.c#L7) - a **hash function** designed by Christopher Wellons (https://nullprogram.com/)
- Integer hash function (https://gist.github.com/badboy/6267743)
- 4-byte Integer hashing (http://burtleburtle.net/bob/hash/integer.html)

Implementing hash tables

Hash tables are fundamental data structures that associate a set of keys with a set of values. Each `<key, value>` pair is an entry in the table. Knowing the key, we can look for its corresponding value (*GET*). We can also add (*PUT*) or remove (*DEL*) `<key, value>` entries just by knowing the key.

In a **hash table**, data is stored in an array (not surprisingly), where each pair `<key, value>` has its own unique index (or *bucket*). Accessing the data becomes highly efficient as long as we know the *bucket*.

The *bucket* is always computed by applying a hash function over a key: $hash_{function}(key) \rightarrow \text{bucket}[<\text{key}, \text{value}>)]$.

Depending on how we plan to tackle potential **hash collisions**, there are various ways to implement a **hash table**:

- *Separate Chaining (using linked lists)*
  - Each *bucket* references a **linked list** (https://en.wikipedia.org/wiki/Linked_list) that contains *none*, *one* or more `<key, value>` entries;
  - To add a new entry (*PUT*) we compute the bucket (`hash(key)`), and then we append the `<key, value>` to the corresponding **linked list**. If the `key` already exists, we just update the `value` ;
  - To identify an entry (*GET*), we compute the *bucket*, traverse the corresponding **linked list** until we find the `key` .
- Separate Chaining (dynamically growing array buckets)
  - Each *bucket* is an array that grows dynamically if needed;
- Separate Chaining (red black trees optimization)
  - The *buckets* are being transformed into a *red black tree* if the number of elements contained exceeds a certain threshold.
- *Open Addressing*
  - There are no **linked lists** involved - there's only one `<key, value>` entry per bucket;
  - In case of collisions, we *probe* the array to find another suitable *bucket* for our entry, and then we add the entry at this new-found *empty* location;
  - Various algorithms for "probing" exists; the simplest one is called *linear probing* (https://en.wikipedia.org/wiki/Linear_probing) - in case of collision, we just jump to the next available *bucket*;
  - Deleting an existing entry is a complex operation.

Each strategy has its PROs and CONs. For example, the creators of Java preferred to use *Separate Chaining* in their `HashMap` implementation, while the creators of python went with *Open Addressing* for their `dict` .

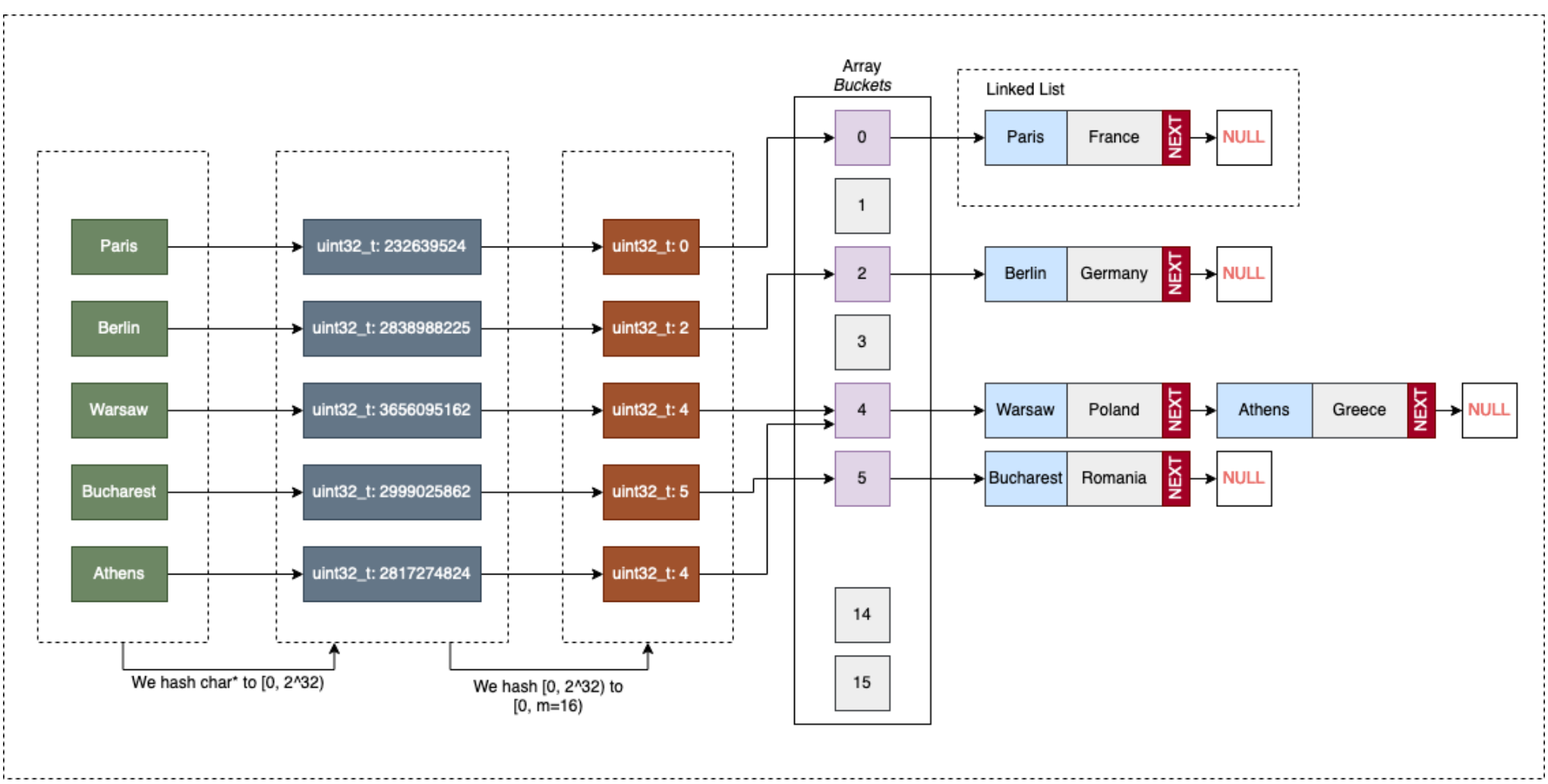
Separate Chaining is simpler to implement, and in case we have a high frequency **hash collisions**, performance degradation is more *graceful* - not as exacerbated as for *Open Addressing*.

Open Addressing is more complex to implement, but if we keep **hash collisions** at bay, it's very *cache-friendly*.

Separate Chaining (using Linked Lists)

To better understand how *Separate Chaining* works, let's represent in a visual way a **hash table** that associates the capitals of Europe (the keys) to their corresponding countries (the values):

European Capital	European Country
"Paris"	"France"
"Berlin"	"Germany"
"Warsaw"	"Poland"
"Bucharest"	"Romania"
"Athens"	"Greece"



Both keys and values are strings (`char*`).

The first step is to reduce the keyspace to a numeric value to hash the keys further and associate them with buckets. For this, we can use something like djb2 to hash the string to a `uint32_t` value. We already know that **djb** works well with string inputs.

This step is not "part" of the **hash table** itself. It's impossible for a **hash table** to understand all data formats and reduce them to 32-bit unsigned integers(`uint32_t`). In this regard, the *data* needs to come with its mechanisms for this. So except the keys are `uint32_t` , additional **hash functions** should be provided to the **hash table** (for the *data*, by the *data*).

After the keys are "projected" to the `uint32_t` space, we must map them to *buckets* (indices in our array). Another **hash function** will reduce the *key-space* further: from `uint32_t` to `[0, M)` , where `M` is the total number of buckets. For simplicity, we can use **division hashing**, **multiplicative hashing**, or whatever hash function we think it's good.

After the *bucket* is identified, we check if it's empty (`NULL`). If it's `NULL` , we create a new *linked list*, and add the pair `<key, value>` to it. If it's not empty, we append the entry to the existing structure.

A generic implementation

Preferably, our **hash table** will be as *generic* as possible so that it can be re-used for various key/values combinations. We want to write code once and re-use it later. Writing a new **hash table** every time the key/value combination changes it's not ideal.

The bad news, C is not exactly the most generic-friendly programming language there is.

The good news, C is flexible enough to make generic programming happen (with some effort) by:

- using `void*` to pass data around,
- using `#macros` that generate *specialized* code after pre-processing (something similar to C++ templates).

> *I've recently found an interesting generic data structures library called STC (https://github.com/tylov/STC); looking at the code can be pretty inspirational.*

I am the camp that prefers `#macros` over `void*`, but **not** when it comes to writing tutorials. With `#macros`, everything becomes messy fast, and the code is hard to debug and maintain. So, for this particular implementation, we will use `void*` for passing/retrieving *generic data* to/from the **hash table**.

The model

We start by defining the `struct`(s) behind **hash tables**. The main structure (`ch_hash`) will contain:

- `capacity` - the total number of available buckets;
- `size` - the total number of elements;
- `buckets` - a dynamic (expandable if needed) array of linked lists;
- Structures to keep tracking of data-specific operations (e.g., a function that checks if two values are equal, etc.);

```
typedef struct ch_hash_s {
    size_t capacity;
    size_t size;
    ch_node **buckets;
    ch_key_ops key_ops;
    ch_val_ops val_ops;
} ch_hash;
```

As we said, the *buckets* are composed of linked lists:**

```
typedef struct ch_node_s {
    uint32_t hash;
    void *key;
    void *val;
    struct ch_node_s *next;
} ch_node;
```

The `ch_node` structure contains:

- `uint32_t hash` is not the actual bucket index, but it's the hash of the data that is going inside the **hash table**. For example, if our keys are strings, `uint32_t hash` is obtained by applying a **hashing function** that works for strings (e.g.: djb2). This is only computed once, and it's (re)used whenever we need to ** rehash** our table;
- `key` and `val` of a `void*` type. This gives us enough flexibility to store almost anything in the table.
- `struct ch_node_s *next` which is a reference to the next node in the list.

The only thing left to explain are the two "unknown" members: `ch_hash->ch_key_ops` and `ch_hash->ch_val_ops`:

```
typedef struct ch_key_ops_s {
    uint32_t (*hash)(const void *data, void *arg);
    void* (*cp)(const void *data, void *arg);
    void (*free)(void *data, void *arg);
    bool (*eq)(const void *data1, const void *data2, void *arg);
    void *arg;
} ch_key_ops;

typedef struct ch_val_ops_s {
    void* (*cp)(const void *data, void *arg);
    void (*free)(void *data, void *arg);
    bool (*eq)(const void *data1, const void *data2, void *arg);
    void *arg;
} ch_val_ops;
```

The two types, `ch_key_ops` and `ch_val_ops` are simple structures (`struct`) used to group functions specific to the data inserted in the **hash table**. As we previously stated, it's impossible to think of all the possible combinations of keys/pairs and their types, so we let the user supply us with the truth.

> *In case the above code is confusing, please refer to the following article: Function pointers (https://en.wikipedia.org/wiki/Function_pointer); it will explain to you in great detail how we can pass functions around through "pointers".*

> *It's funny to think C had its first functional programing feature implemented decades before Java...*

Think of `ch_key_ops` and `ch_val_ops` as *traits* (https://en.wikipedia.org/wiki/Trait_(computer_programming)), bits of logic that are external to the **hash table** itself, but by defining them, we are creating a simple contract between our structure and the data we are inserting:

> *Look, if you want to add a `chr*` as a `key` in our **hash table**, please tell us first how do you: compute its hash, copy it, and free the memory. Our table will do the heavy work for you, but first, we need to know this !?*

As an example, the required functions for strings (`chr*`) can be implemented like this:

```
//Finalizer
static uint32_t ch_hash_fmix32(uint32_t h) {
    h ^= h >> 16;
    h *= 0x3243f6a9U;
    h ^= h >> 16;
    return h;
}

// Returns the uint32_t hash value of a string computed using djb2
uint32_t ch_string_hash(const void *data, void *arg) {

    //djb2
    uint32_t hash = (const uint32_t) 5381;
    const char *str = (const char*) data;
    char c;
    while((c=*str++)) {
        hash = ((hash << 5) + hash) + c;
    }

    return ch_hash_fmix32(hash);
}

// Returns a copy of the *data string
void* ch_string_cp(const void *data, void *arg) {
    const char *input = (const char*) data;
    size_t input_length = strlen(input) + 1;
    char *result;
    result = malloc(sizeof(*result) * input_length);
    if (NULL==result) {
        fprintf(stderr,"malloc() failed in file %s at line # %d", __FILE__,__LINE__);
        exit(EXIT_FAILURE);
    }
    strcpy(result, input);
    return result;
}

// Check if two strings are equal
bool ch_string_eq(const void *data1, const void *data2, void *arg) {
    const char *str1 = (const char*) data1;
    const char *str2 = (const char*) data2;
    return !(strcmp(str1, str2)) ? true : false;
}

// Free the memory associated with a string
void ch_string_free(void *data, void *arg) {
    free(data);
}
```

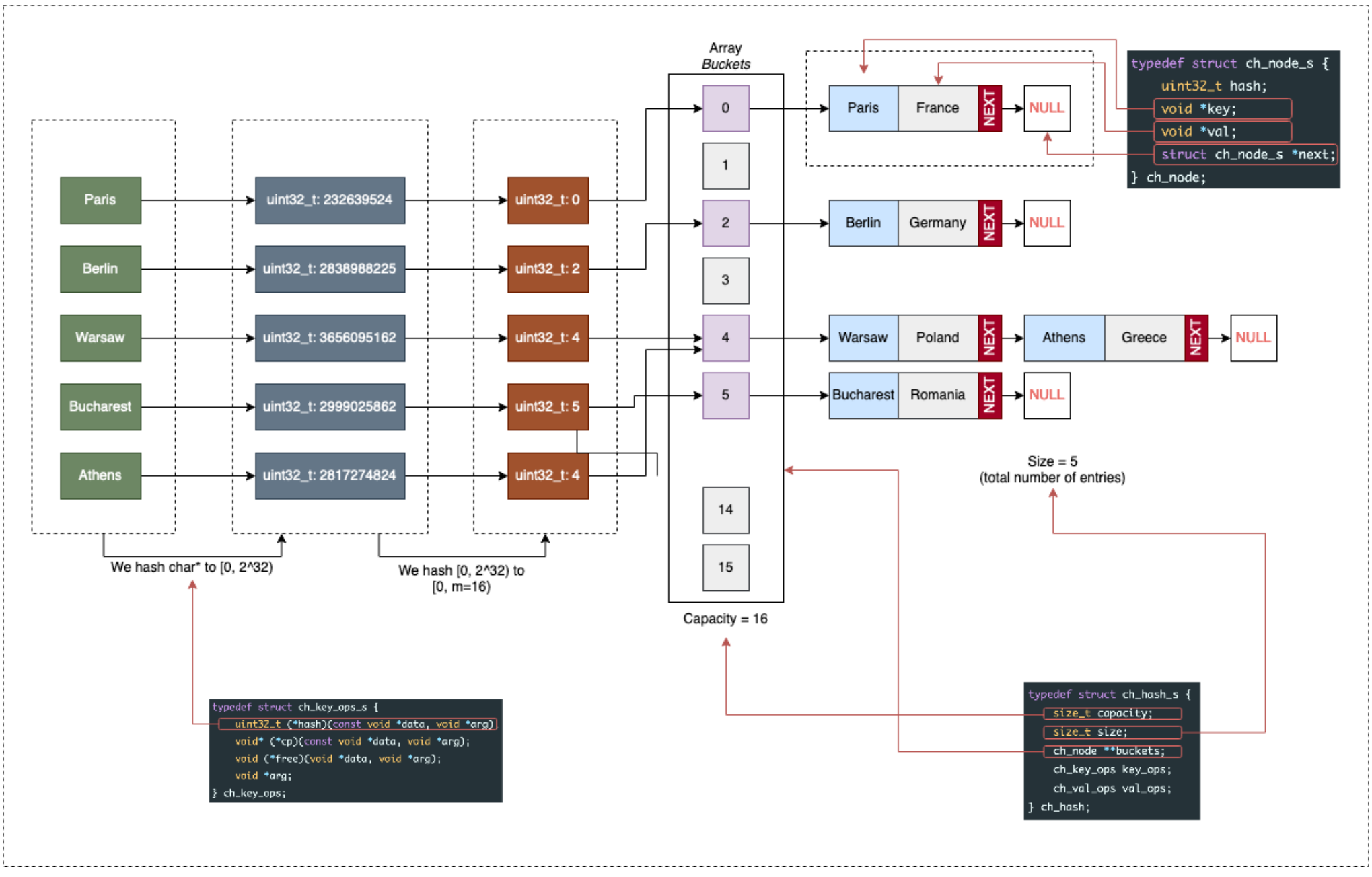
As you can see, even if our methods are going to be used for strings (`chr*`) we are forced to use `void*` instead and cast data manually.

Sadly, `void*` is not the `T` we know from *C++ Templates* or *Java Generics*. No compile-time validations are performed. It's up to us to use it accordingly.

And then, what it remains is two keep two instances around (one for keys and one for values):

```
ch_key_ops ch_key_ops_string = { ch_string_hash, ch_string_cp, ch_string_free, ch_string_eq, NULL};
ch_val_ops ch_val_ops_string = { ch_string_cp, ch_string_free, ch_string_eq, NULL};
```

Now let's retake a look at the initial diagram and how are structures fit together with the visual representation:



The interface

Our **hash table** (`ch_hash`) will support and publicly expose the following functions (interface):

```
// Creates a new hash table
ch_hash *ch_hash_new(ch_key_ops k_ops, ch_val_ops v_ops);

// Free the memory associated with the hash (and all of its contents)
void ch_hash_free(ch_hash *hash);

// Gets the value coresponding to a key
// If the key is not found returns NULL
void* ch_hash_get(ch_hash *hash, const void *k);

// Checks if a key exists or not in the hash table
bool ch_hash_contains(ch_hash *hash, const void *k);

// Adds a <key, value> pair to the table
void ch_hash_put(ch_hash *hash, const void *k, const void *v);

// Prints the contents of the hash table
void ch_hash_print(ch_hash *hash, void (*print_key)(const void *k), void (*print_val)(const void *v));

// Get the total number of collisions
uint32_t ch_hash_numcol(ch_hash *hash);
```

As you might've noticed, there's no function for deleting an entry. That's intentionally left out as a proposed exercise.

Before implementing the enumerated methods, it's good to clarify a few things that are not obvious from the interface itself.

Just for fun (best practices are fun!), our **hash table** will grow automatically if its `size` reaches a certain threshold. So every time we insert a new element, we check if the threshold has been reached and if it's time to increase the capacity (and the number of available buckets). A rehashing of everything will also be performed - old entries might go to new buckets.

In this regard let's define the following constants (we can tweak their values later):

```
#define CH_HASH_CAPACITY_INIT (32)
#define CH_HASH_CAPACITY_MULT (2)
#define CH_HASH_GROWTH (1)
```

And then perform the following check each time we insert a new item:

```
// Grow if needed
if (hash->size > hash->capacity * CH_HASH_GROWTH) {
    ch_hash_grow(hash);
        // -> The function will perform a full rehashing to a new array of buckets
        // of size [hash->capacity * CH_HASH_CAPACITY_MULT]
}
```

The `ch_hash_grow(ch_hash *hash)` function it's not defined as part of the interface; it's `private`. We won't be exposing it to the header file.

Another function that's not exposed in our (public) interface is: `ch_node* ch_hash_get_node(ch_hash*, const void*)`. This one is used to check if a node exists or not. In case it exists, it retrieves the node. Otherwise, it returns `NULL`.

The reason we have two functions for retrieving data is simple:

- `void* ch_hash_get(ch_hash *hash, const void *k);` (public)
- `ch_node* ch_hash_get_node(ch_hash *hash, const void *key)` (private)

`ch_hash_get_node` works on a structure (`ch_node`) that is internal for our implementation, while `ch_hash_get` will use `ch_hash_get_node` to retrieve the actual value.

Creating/Destroying a hash table

`ch_hash_new` is a constructor-like function that dynamically allocates memory for a new `ch_hash`.

`ch_hash_free` is a destructor-like function that `free`s all the memory associated with a `ch_hash`. `ch_hash_free` goes even deeper and de-allocates memory for the internal *buckets* and all the existing entries.

The code for `ch_hash_new` is:

```
ch_hash *ch_hash_new(ch_key_ops k_ops, ch_val_ops v_ops) {
    ch_hash *hash;

    hash = malloc(sizeof(*hash));
    if(NULL == hash) {
        fprintf(stderr,"malloc() failed in file %s at line # %d", __FILE__,__LINE__);
        exit(EXIT_FAILURE);
    }

    hash->size = 0;
    hash->capacity = CH_HASH_CAPACITY_INIT;
    hash->key_ops = k_ops;
    hash->val_ops = v_ops;

    hash->buckets = malloc(hash->capacity * sizeof(*(hash->buckets)));
    if (NULL == hash->buckets) {
        fprintf(stderr,"malloc() failed in file %s at line # %d", __FILE__,__LINE__);
        exit(EXIT_FAILURE);
    }
    for(int i = 0; i < hash->capacity; i++) {
        // Initially all the buckets are NULL
        // Memory will be allocated for them when needed
        hash->buckets[i] = NULL;
    }

    return hash;
}
```

Note: using `exit(EXIT_FAILURE);` is not ideal, and it's not a good practice when you are writing libraries you want to share with the public. Basically, you are telling the program to **stop**, without giving it any chance to do some cleaning first. Don't do this if you plan to make your own **hash table** library and share it with a larger audience.

The code for `ch_hash_free` is:

```
void ch_hash_free(ch_hash *hash) {

    ch_node *crt;
    ch_node *next;

    for(int i = 0; i < hash->capacity; ++i) {
        // Free memory for each bucket
        crt = hash->buckets[i];
        while(NULL!=crt) {
            next = crt->next;

            // Free memory for key and value
            hash->key_ops.free(crt->key, hash->key_ops.arg);
            hash->val_ops.free(crt->val, hash->val_ops.arg);

            // Free the node
            free(crt);
            crt = next;
        }
    }
    // Free the buckets and the hash structure itself
    free(hash->buckets);
    free(hash);
}
```

In terms of what's happening in `ch_hash_free`, things are quite straightforward, except for those two lines:

```
hash->key_ops.free(crt->key, hash->key_ops.arg);
hash->val_ops.free(crt->val, hash->val_ops.arg);
```

Because the **hash table** doesn't know what type of data it holds in the entries (`void*` is not very explicit, isn't it), it's impossible to `free` the memory correctly.

So in this regard, we use the `free` functions referenced inside `key_ops` (for keys) and `val_ops` (for values).

Retrieving a value from the hash table

For the sake of simplicity, the function that translates the `uint32_t` hash of the key to the `[0, hash->capacity)` space is `%`.

We will use **division hashing** in our implementation.

```
static ch_node* ch_hash_get_node(ch_hash *hash, const void *key) {

    ch_node *result = NULL;
    ch_node *crt = NULL;
    uint32_t h;
    size_t bucket_idx;

    // We compute the hash of the key to check for it's existence
    h = hash->key_ops.hash(key, hash->key_ops.arg);
    // We use simple division hashing for determining the bucket
    bucket_idx = h % hash->capacity;
    crt = hash->buckets[bucket_idx];

    while(NULL!=crt) {
        // Iterated through the linked list found at the bucket
        // to determine if the element is present or not
        if (crt->hash == h && hash->key_ops.eq(crt->key, key, hash->val_ops.arg)) {
            result = crt;
            break;
        }
        crt = crt->next;
    }

    // If the while search performed in the while loop was successful,
    // `result` contains the node
    // otherwise it's NULL
    return result;
}
```

`ch_hash_get` is just a wrapper function built on top of `ch_hash_get_node`. It has filtering purposes only: retrieving the value (`ch_node->val`) instead of the internal `ch_node` representation.

```
void* ch_hash_get(ch_hash *hash, const void *k) {
    ch_node *result = NULL;
    if (NULL!=(result=ch_hash_get_node(hash, k))) {
        return result->val;
    }
    return NULL;
}
```

Adding an entry to the hash table

The `ch_hash_put` method is responsible for adding new entries to the **hash table**.

```c
void ch_hash_put(ch_hash *hash, const void *k, const void *v) {
    ch_node *crt;
    size_t bucket_idx;
    crt = ch_hash_get_node(hash, k);
    if (crt) {
        // Key already exists
        // We need to update the value
        hash->val_ops.free(crt->val, hash->val_ops.arg);
        crt->val = v ? hash->val_ops.cp(v, hash->val_ops.arg) : 0;
    }
    else {
        // Key doesn't exist
        // - We create a node
        // - We add a node to the corresponding bucket
        crt = malloc(sizeof(*crt));
        if (NULL == crt) {
            fprintf(stderr,"malloc() failed in file %s at line # %d", __FILE__,__LINE__);
            exit(EXIT_FAILURE);
        }
        crt->hash = hash->key_ops.hash(k, hash->key_ops.arg);
        crt->key = hash->key_ops.cp(k, hash->key_ops.arg);
        crt->val = hash->val_ops.cp(v, hash->val_ops.arg);

        // Simple division hashing to determine the bucket
        bucket_idx = crt->hash % hash->capacity;
        crt->next = hash->buckets[bucket_idx];
        hash->buckets[bucket_idx] = crt;

        // Element has been added successfully
        hash->size++;

        // Grow if needed
        if (hash->size > hash->capacity * CH_HASH_GROWTH) {
            ch_hash_grow(hash);
        }
    }
}
```

`ch_hash_grow` is an internal method (not exposed in the public API) responsible for scaling up the number of buckets (`hash->buckets`) based on the number of elements contained in the table (`hash->size`).

`ch_hash_grow` allocates memory for a new array `ch_node **new_buckets`, and then rehashes all the elements from the *old* array (`hash->buckets`) by projecting them in the *new buckets*.

In regards to this, 3 constants are being used:

```c
#define CH_HASH_CAPACITY_INIT (32)
#define CH_HASH_CAPACITY_MULT (2)
#define CH_HASH_GROWTH (1)
```

`CH_HASH_CAPACITY_INIT` is the initial size of the array (`hash->buckets`).

`CH_HASH_CAPACITY_MULT` is the growth multiplier: `hash->capacity *= CH_HASH_CAPACITY_MULT`. Normally, it would've been better to grow to a bigger prime number (because of **division hashing**), but that would've been more complicated to implement in the code.

`CH_HASH_GROWTH` is the load factor - a constant that influences the growth condition: `hash->size > hash->capacity * CH_HASH_GROWTH`.

If allocating a new array (`new_buckets`) fails, we will keep the old one - that's a pragmatic decision to make - so, instead of crashing the program, we accept a potential drop in performance.

```c
static void ch_hash_grow(ch_hash *hash) {

    ch_node **new_buckets;
    ch_node *crt;
    size_t new_capacity;
    size_t new_idx;

    new_capacity = hash->capacity * CH_HASH_CAPACITY_MULT;
    new_buckets = malloc(sizeof(*new_buckets) * new_capacity);
    if (NULL==new_buckets) {
        fprintf(stderr, "Cannot resize buckets array. Hash table won't be resized.\n");
        return;
    }
    for(int i = 0; i < new_capacity; ++i) {
        new_buckets[i] = NULL;
    }

    // Rehash
    // For each bucket
    for(int i = 0; i < hash->capacity; i++) {
        // For each linked list
        crt = hash->buckets[i];
        while(NULL!=crt) {
            // Finding the new bucket
            new_idx = crt->hash % new_capacity;
            ch_node *cur = crt;
            crt = crt->next;
            cur->next = new_buckets[new_idx];
            new_buckets[new_idx] = cur;
        }
    }

    hash->capacity = new_capacity;

    // Free the old buckets
    free(hash->buckets);

    // Update with the new buckets
    hash->buckets = new_buckets;
}
```

> *Because there's no `ch_hash_delete` method, there's no `ch_hash_shrink` method.*

Printing the contents of a hash table

And last but not least, `ch_hash_print` is a util method that allows us to print the contents of our chained **hash table** to `stdout`. Because we don't know how the keys and values look like, we expect the user to supply us with the corresponding *printing functions*.

```c
void ch_hash_print(ch_hash *hash, void (*print_key)(const void *k), void (*print_val)(const void *v)) {

    ch_node *crt;

    printf("Hash Capacity: %lu\n", hash->capacity);
    printf("Hash Size: %lu\n", hash->size);

    printf("Hash Buckets:\n");
    for(int i = 0; i < hash->capacity; i++) {
        crt = hash->buckets[i];
        printf("\tbucket[%d]:\n", i);
        while(NULL!=crt) {
            printf("\t\thash=%" PRIu32 ", key=", crt->hash);
            print_key(crt->key);
            printf(", value=");
            print_val(crt->val);
            printf("\n");
            crt=crt->next;
        }
    }
}
```

A possible implementation for the `print_key` function:

```c
void ch_string_print(const void *data) {
    printf("%s", (const char*) data);
}
```

Calling `ch_hash_print` is then as simple as: `ch_hash_print(htable, ch_hash_print)`.

Calculating the number of collisions from the hash table

The function (`ch_hash_numcol`) is quite simple to implement:

- We will create an internal function `uint32_t ch_node_numcol(ch_node* node)` that counts the number of collisions per *bucket*;
- We sum the number of collisions per *bucket*

The equivalent C code is:

```c
static uint32_t ch_node_numcol(ch_node* node) {
    uint32_t result = 0;
    if (node) {
        while(node->next!=NULL) {
            result++;
            node = node->next;
        }
    }
    return result;
}

uint32_t ch_hash_numcol(ch_hash *hash) {
    uint32_t result = 0;
    for(int i = 0; i < hash->capacity; ++i) {
        result += ch_node_numcol(hash->buckets[i]);
    }
    return result;
}
```

Using the hash table

The code put together can be found here:

- chained_hash.c (https://github.com/nomemory/chained-hash-table-c/blob/main/chained_hash.c)
- chained_hash.h (https://github.com/nomemory/chained-hash-table-c/blob/main/chained_hash.h)

Using the **hash table** is quite trivial. Let's take for example the following:

```
ch_hash *htable = ch_hash_new(ch_key_ops_string, ch_val_ops_string);

ch_hash_put(htable, "Paris", "France");
ch_hash_put(htable, "Berlin", "Germany");
ch_hash_put(htable, "Warsaw", "Poland");
ch_hash_put(htable, "Bucharest", "Romania");
ch_hash_put(htable, "Athens", "Greece");

printf("%s\n", (char*) ch_hash_get(htable, "Athens"));
printf("%s\n", (char*) ch_hash_get(htable ,"Bucharest"));

ch_hash_print(htable, ch_string_print, ch_string_print);

return 0;
```

The output will be:

```
Greece
Romania
Hash Capacity: 32
Hash Size: 5
Hash Buckets:
        bucket[0]:
        bucket[1]:
                hash=2838988225, key=Berlin, value=Germany
        bucket[2]:
        bucket[3]:
        bucket[4]:
                hash=232639524, key=Paris, value=France
        bucket[5]:
        bucket[6]:
                hash=2999025862, key=Bucharest, value=Romana
        bucket[7]:
        bucket[8]:
                hash=2817274824, key=Athens, value=Greece
// ...
// and so on
```

Separate Chaining (Dynamically growing array buckets)

The previous implementation is rather *naive*, so don't judge it too harshly. I would be a little concerned if you will use it in practice.

The *elephant in the room* when it comes to **Linked Lists** is how unfriendly they are to caching. A **linked list** is suitable for inserting items (depending on the scenario). Still, they are not well equipped for current hardware architectures, especially when it comes to iteration and reading elements.

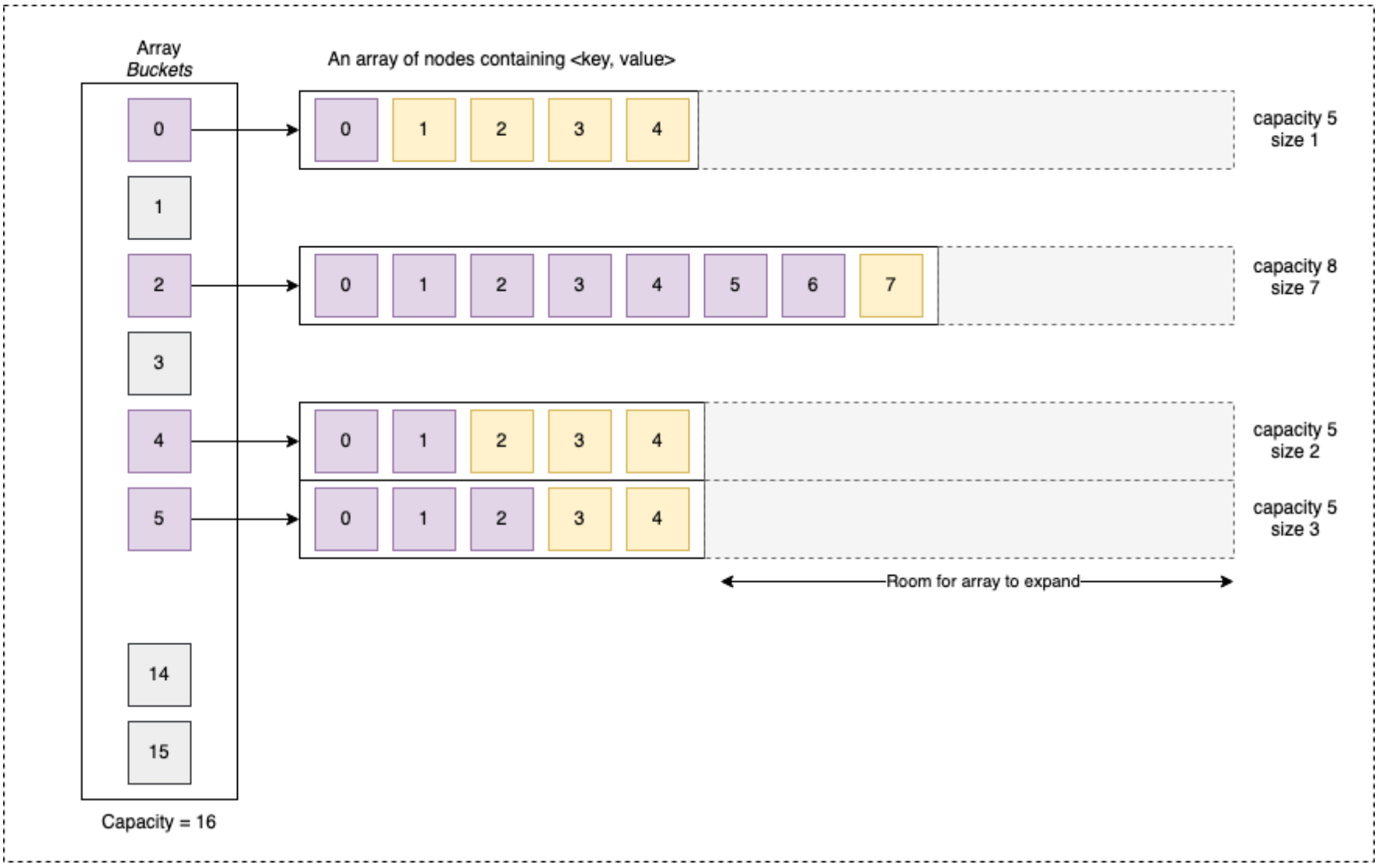
CPUs are usually looking to cache two things: the recently accessed memory, and then it tries to predict which memory will be used next. For **linked lists**, the data is not contiguous; nodes are somewhat scattered (also depends on the `malloc()` implementation), so calling `node->next` might be subject to cache misses.

So what if we plan to use a "self-expanding" array instead of a **linked list**. When I say self-expanding array, I am thinking something akin to C++'s `std::vector` or Java's `ArrayList`.

The number of *cache misses* will decrease, but we will need to grow the array; `realloc` is expensive.

This "optimization" will improve the read times, but it will increase (significantly) the writes.

Visually our new **hash table** will look like this:



Writing a `vector`-like structure for our buckets: `ch_vect`

The model

Implementing a `vector`-like structure is straightforward. We will use the same approach as before to achieve *genericity* (is this a word?!) as before, but we will keep the interface and the model as simple as possible this time.

Each of our buckets will be a *vector*, with the following internal structure:

```
#define VECT_INIT_CAPACITY (1<<2)
#define VECT_GROWTH_MULTI (2)

typedef struct ch_vect_s {
        size_t capacity;
        size_t size;
        void **array;
} ch_vect;
```

The `capacity` is the actual memory allocated for the `ch_vect` vector internal `ch_vect->array`. This may, or may be not fully used.

The `size` is the actual number of elements in the `ch_vect`. It's the exact length we use for iteration (`i<vect->size`).

`VECT_INIT_CAPACITY` represents the initial `capacity` for the `ch_vect`.

`VECT_GROWTH_MULTI` is the multiplicator the `ch_vect->array`. Whenever we are about to remain without space, we perform another allocation of size `VECT_GROWTH_MULTI * ch_vect->capacity`.

The value of `VECT_GROWTH_MULTI` is ideally a number in the `[1, 2]` interval. For simplicity we've picked `2`, but there's a risk our `array` will grow faster than it's needed; doubling its `size` every time might be an exaggeration.

The interface

When it comes to the interface, for simplicity, we will limit ourselves in implementing only the functions we are going to use:

```
ch_vect* ch_vect_new(size_t capacity);
ch_vect* ch_vect_new_default();
void ch_vect_free(ch_vect *vect);
void* ch_vect_get(ch_vect *vect, size_t idx);
void ch_vect_set(ch_vect *vect, size_t idx, void *data);
void ch_vect_append(ch_vect *vect, void *data);
```

`ch_vect_new` is the constructor-like function. It accepts only a parameter which is the initial `ch_vect->capacity`.

`ch_vect_newdefault` would've been the overloaded `ch_vect_new` method that uses `capacity=VECT_INIT_CAPACITY` as the default param value. Unfortunately, C doesn't support function overloading (https://en.wikipedia.org/wiki/Function_overloading), so we had to give it another name.

`ch_vect_free` is the destructor-like function. It only de-allocates the memory of the structure itself, but not for the data.

`ch_vect_get` is used to access an item from the internal array (`ch_vect->array`) at the specified index `idx`.

`ch_vect_set` is used to set the value of an item from the internal array (`ch_vect->array`) at the specified index.

`ch_vect_append` appends an element at the end of the `ch_vect->array`. In case there's not enough allocated space for it, it allocates a new array.

It's important to understand there's not a good idea to interact with `ch_vect->array` directly. If you do this, `ch_vect->size` won't be updated, so growth is not guaranteed to work.

Allocating an de-allocating memory

The code is definitely standard for both the constructor and the destructor:

```
ch_vect* ch_vect_new(size_t capacity) {
    ch_vect *result;
    result = malloc(sizeof(*result));
    if (NULL==result) {
        fprintf(stderr,"malloc() failed in file %s at line # %d", __FILE__,__LINE__);
        exit(EXIT_FAILURE);
    }
    result->capacity = capacity;
    result->size = 0;
    result->array = malloc(result->capacity * sizeof(*(result->array)));
    if (NULL == result->array) {
        fprintf(stderr,"malloc() failed in file %s at line # %d", __FILE__,__LINE__);
        exit(EXIT_FAILURE);
    }
    return result;
}

ch_vect* ch_vect_new_default() {
    return ch_vect_new(VECT_INIT_CAPACITY);
}

void ch_vect_free(ch_vect *vect) {
    free(vect->array);
    free(vect);
}
```

> *It's a bad idea to call `exit(EXIT_FAILURE)` if you are building a library. You need to give the program a chance to recover from an error. So don't do it in practice.*

Reading and setting data

Again, there's nothing fancy.

```c
void* ch_vect_get(ch_vect *vect, size_t idx) {
    if (idx >= vect->size) {
        fprintf(stderr, "cannot get index %lu from vector.\n", idx);
        exit(EXIT_FAILURE);
    }
    return vect->array[idx];
}

void ch_vect_set(ch_vect *vect, size_t idx, void *data) {
    if (idx >= vect->size) {
        fprintf(stderr, "cannot get index %lu from vector.\n", idx);
        exit(EXIT_FAILURE);
    }
    vect->array[idx] = data;
}
```

Appending an element

Appending a new element to the `ch_vect` should take into consideration the following:

- It actual `size` of the vector is equal or becomes bigger than the `capacity` we need to:
    - Compute a `new_capacity`;
        - Before computing it, take care of potential overflows;
    - Allocate a new array `new_array` with `new_capacity`;
    - Copy all elements from the old array `ch_vect->array` to the new array `new_array`;
    - Free the memory associated with the old array `ch_vect->array`;
    - Update the `ch_vect->array` so it points to the `new_array`.

The equivalent code for what has been described above is the following:

```c
void ch_vect_append(ch_vect *vect, void *data) {
    if (!(vect->size < vect->capacity)) {
        // Check for a potential overflow
        uint64_t tmp = (uint64_t) VECT_GROWTH_MULTI * (uint64_t) vect->capacity;
        if (tmp > SIZE_MAX) {
            fprintf(stderr, "size overflow\n");
            exit(EXIT_FAILURE);
        }
        size_t new_capacity = (size_t) tmp;
        void *new_array = malloc(new_capacity * sizeof(*(vect->array)));
        if (NULL==new_array) {
            fprintf(stderr,"malloc() failed in file %s at line # %d", __FILE__,__LINE__);
            exit(EXIT_FAILURE);
        }
        memcpy(new_array, vect->array, vect->size * sizeof(*(vect->array)));
        free(vect->array);
        vect->array = new_array;
        vect->capacity = new_capacity;
    }
    vect->array[vect->size] = data;
    vect->size++;
}
```

`SIZE_MAX` is a C99 macro that describes the maximum possible value for a `size_t` type.

Updating the existing `ch_hash` structure to use `ch_vect` for buckets

Our initial `ch_hash` structure from the previous chapter Separate Chaining (using Linked Lists), and its associated interface won't suffer a lot of changes.

We will simply create another version, and to make a distinction between the two we will name it: `ch_hashv` (with `v` from `v`ector at the end):

```c
typedef struct ch_hashv_s {
    size_t capacity;
    size_t size;
    ch_vect **buckets;
    ch_key_ops key_ops;
    ch_val_ops val_ops;
} ch_hashv;

// VERSUS

typedef struct ch_hash_s {
    size_t capacity;
    size_t size;
    ch_node **buckets;
    ch_key_ops key_ops;
    ch_val_ops val_ops;
} ch_hash;
```

As you can see, only the type of the `buckets` changes: `ch_node **buckets` vs. `ch_vect **buckets`.

The interface will remain identical as well.

For the actual methods, we are going to keep 90% of the code from the previous implementation. But, instead of using a `while` loop for iterating through the **linked list**, a `for` loop will be used for iterating through the **dynamically-expanding array** (`ch_vect`).

In case you are curious about the changes and the implementation of `ch_hashv`, you can take a look at the code directly on github (https://github.com/nomemory/chained-hash-table-c):

- chained_hashv.c (https://github.com/nomemory/chained-hash-table-c/blob/main/chained_hashv.c)
- chained_hash.h (https://github.com/nomemory/chained-hash-table-c/blob/main/chained_hashv.h)
- ch_vect.c (https://github.com/nomemory/chained-hash-table-c/blob/main/vect.c)
- ch_vect.h (https://github.com/nomemory/chained-hash-table-c/blob/main/vect.h)

Separate Chaining (Red Black Trees optimization)

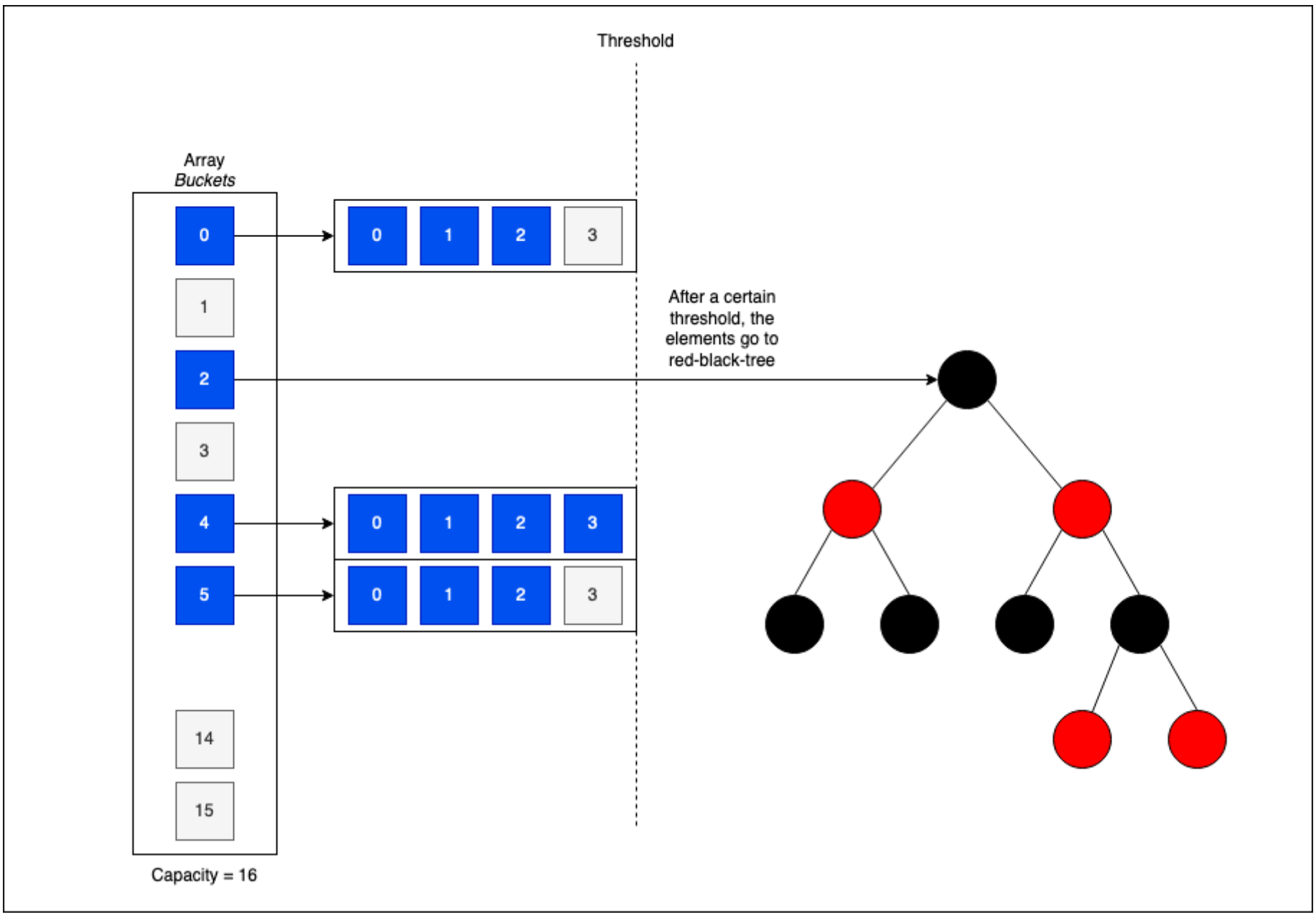
From a practical perspective, the `ch_hashv` improves the read times for the **hash table** because it reduces the number of *cache misses* introduced by our use of **linked lists**.

But from a theoretical perspective searching in a bucket is still `O(N)`, where `N` is the number of elements in the bucket. So, how can we improve the reads further (`ch_hash_get()`)?

In this regard, we can introduce another optimization:

- We can measure the number of colliding elements inserted in a bucket;
- If this number increases, after a certain threshold, we can "morph" our bucket into a red-black tree (https://en.wikipedia.org/wiki/Red%E2%80%93black_tree).

Without getting into many details, **red-black trees** are binary search trees are self-balancing themselves during the insertion and deletion of new elements. This means that searching performance doesn't degrade, and it will remain in logarithmic bounds.



This is the route the creators of Java took when `HashMap` was implemented.

Open Addressing

Compared to **Separate Chaining**, a **hash table** implemented with **Open Adressing** stores only one element per bucket. We handle collisions by inserting the colliding pairs to unused slots in the *bucket* array.

Finding unused locations in the *bucket* array is achieved through a probing algorithm that determines what the is second (or the *nth*) most *natural* index if the previous ones are occupied.

The most accessible probing algorithm is called **linear probing**. In case of collision, we iterate over each bucket (starting with the first bucket computed), until we find an empty slot to make the insertion.

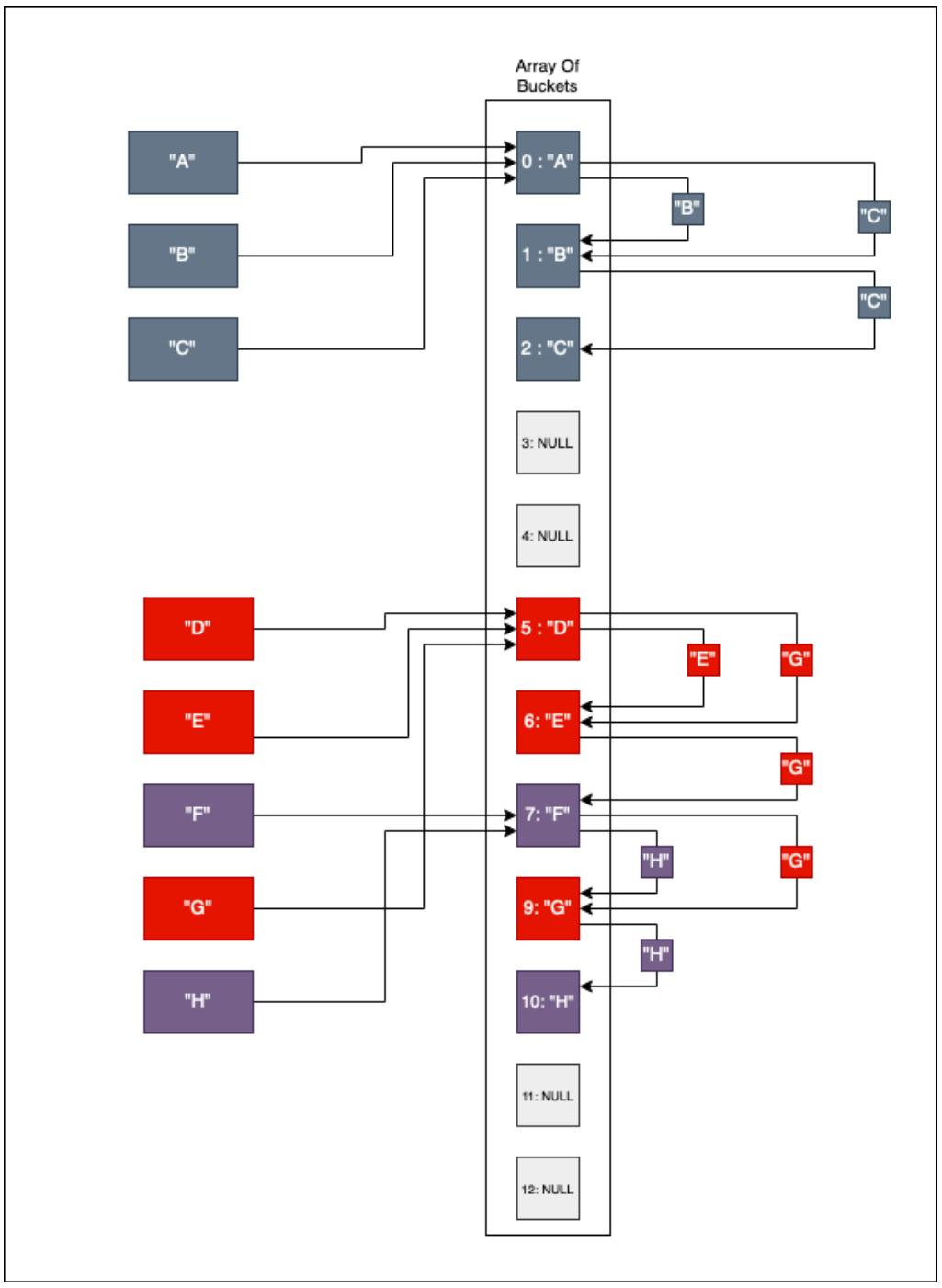
To avoid probing for new slots in excess, we need to keep the load factor of the **hash table** at a satisfactory level, and the hash function should have remarkable diffusion properties. If the load factor (defined as `size/capacity`) reaches a certain threshold, we increase the buckets and perform full-rehashing.

If we fail to increase the number of buckets, clusters of adjacent elements will form. ** Clustering** diminishes the performance of both read and insert operations because we will need to iterate more over more buckets to obtain the correct value.

For example, python's `dict` implementation uses a more advanced variant of **Open Adressing**.

Compared to the **Separate Chaining** approach, cache misses are reduced because we operate on a single contiguous block of memory (the array of buckets). And, as long as we use a decent **hash function**, and the load factor is low, it can lead to superior performance, especially for read operations.

To better understand let's take a look at the following diagram:



We want to insert `<key, value>` pairs with the keys (`char*`) from the set: [`"A"` , `"B"` , `"C"` , etc.]:

- First, we insert `"A"` . The corresponding bucket for `"A"` (after we compute the hash) is `0` ;
- Then we insert `"B"` . The corresponding bucket for `"B"` is `0` . But `0` is occupied by `"A"` . So, by applying **linear probing** we go to the next index, which `1` , and it's free. So `"B"` is now inserted at bucket `1` ;
- Then we insert `"C"` . The corresponding bucket for `"C"` is `0` . But `0` is occupied by `"A"` , the next bucket, `1` is occupied by `"B"` , so we end-up inserting `"C"` to 2.

Let's look further; what is happening for `"D"` , `"E"` , `"F"` , `"G"` , `"H"` is the beginning of a *not* so lovely cluster of elements: areas in the array of buckets where intertwined elements are grouped together.

Having intertwined groups leads to decreased performance. Think of it, for reading the value associated with `"G"` we have to iterate over an area that is naturally associated with bucket number `7` (eg., `"F"`).

Clustering usually happens for two reasons:

- The load factor (size/capacity) is close to `1` , and we need to allocate more empty buckets to increase the *spacing* between elements;
- The hash function doesn't have good diffusion properties, and it's biased towards specific buckets.

Reducing Clustering can also be achieved by changing **linear probing** to another algorithm to identify the next good bucket (e.g., **quadratic probing**).

Tombstones

Compared to **Separate Chaining**, deleting elements in a **hash table** that uses **Open Addressing**, is relatively more challenging to implement.

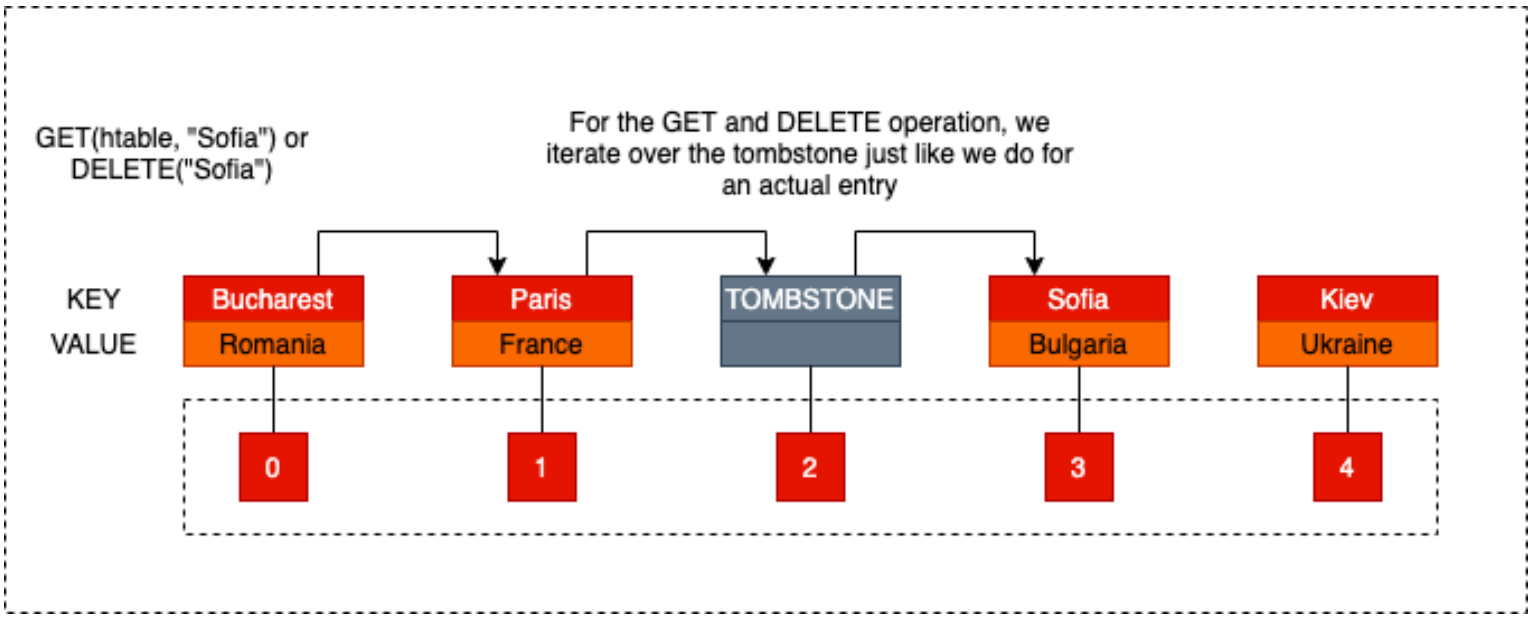
After a delete operation occurs, we cannot simply empty the bucket (associate `NULL` with it). If we do that, we might break a sequence of entries and unjustly isolate some. The separated entries will become unreachable for the following read operations. They become reachable only if the previous *sequence* is somehow *restored* through a *lucky* insertion.

One solution would be to rehash everything after each delete. But think of it for a moment; this means to allocate a new array of buckets, calculate the new index (bucket) for each element, and then free the memory associated with the old and *broken* array. It's a costly and unacceptable endeavor, especially when there are better alternatives.

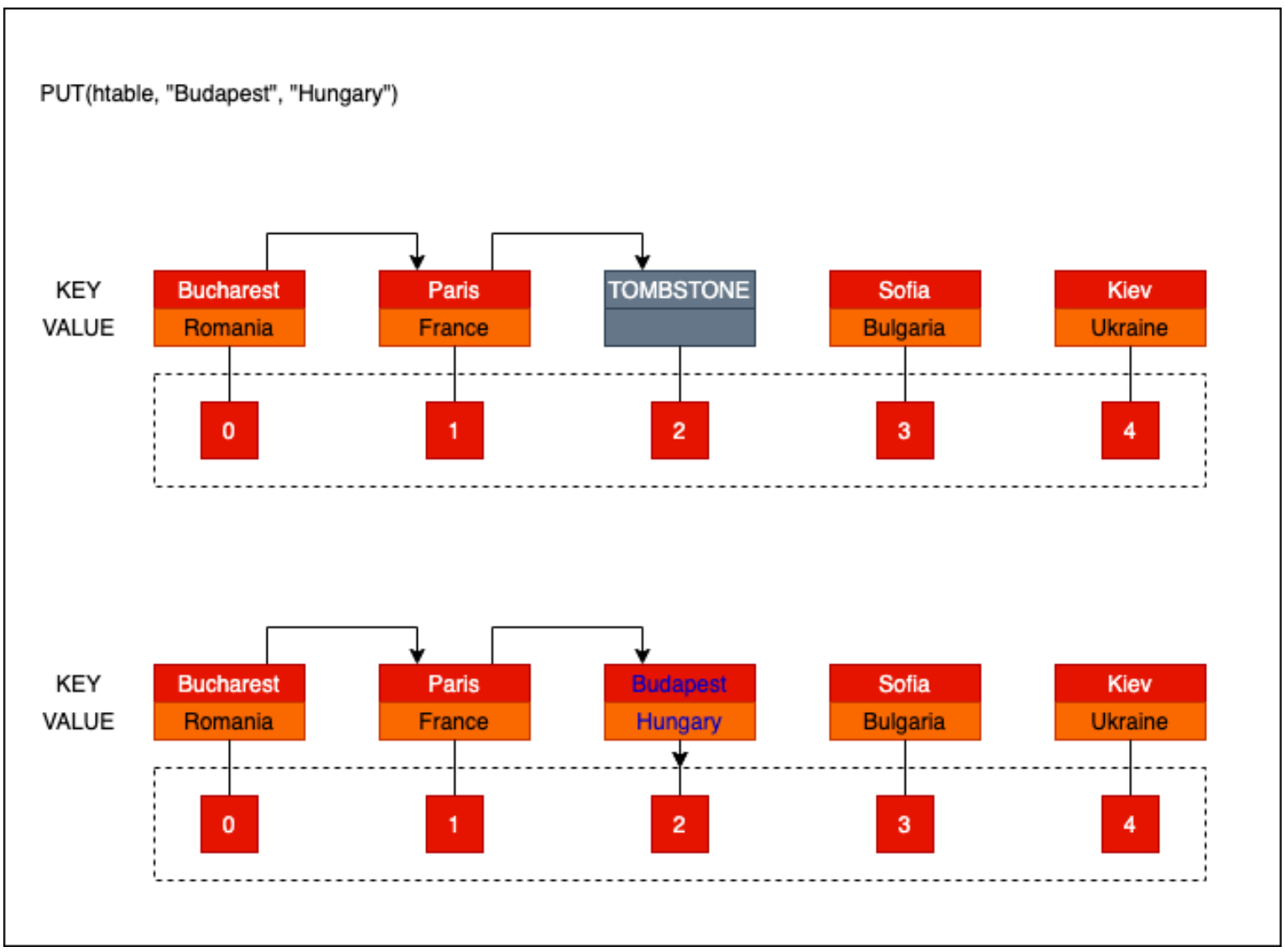
One alternative for avoiding full rehashing is to introduce the notion of **tombstones**. Each time a delete operation occurs, instead of emptying the bucket, we put a sentinel value, metaphorically called **a tombstone**.

The tombstone will behave differently based on the operation.

If we want to perform a `read` operation or another `delete`, the **tombstone** will act just like an actual entry; probing will ignore it and continue further. This way, the rest of the entries are not isolated from the others.



If we want to perform a `put` (insert) operation, the **tombstone** will act as a very inviting empty slot.



A high density of **tombstones** will increase the load factor, just like the actual entries do, so an intelligent **hash table** implementation needs to consider this.

There are ways of avoiding **tombstones** altogether, and if you are interested to read about this, check this article (https://attractivechaos.wordpress.com/2019/12/28/deletion-from-hash-tables-without-tombstones/). But, for simplicity and academic purposes, our **hash table** will use **tombstones**.

A generic implementation

If you want to jump directly into the code, without reading the explanation,s you can clone the following repo (https://github.com/nomemory/open-adressing-hash-table-c):

```
git clone git@github.com:nomemory/open-adressing-hash-table-c.git
```

Similar to the implementations for **Separate Chaining**, we will use the `void*` pointer to achieve some sort of genericity.

The model

For the model layer, we will keep almost the same structures as before.

```
typedef struct oa_key_ops_s {
    uint32_t (*hash)(const void *data, void *arg);
    void* (*cp)(const void *data, void *arg);
    void (*free)(void *data, void *arg);
    bool (*eq)(const void *data1, const void *data2, void *arg);
    void *arg;
} oa_key_ops;

typedef struct oa_val_ops_s {
    void* (*cp)(const void *data, void *arg);
    void (*free)(void *data, void *arg);
    bool (*eq)(const void *data1, const void *data2, void *arg);
    void *arg;
} oa_val_ops;
```

`oa_key_ops_s` and `oa_val_ops_s` are collections of functions that are related to the data kept inside the keys and values.

```
typedef struct oa_pair_s {
    uint32_t hash;
    void *key;
    void *val;
} oa_pair;
```

The `oa_pair_s` is the structure corresponding to the actual `<key, value>` entry.

```
typedef struct oa_hash_s {
    size_t capacity;
    size_t size;
    oa_pair **buckets;
    void (*probing_fct)(struct oa_hash_s *htable, size_t *from_idx);
    oa_key_ops key_ops;
    oa_val_ops val_ops;
} oa_hash;
```

`oa_hash_s` is the actual structure behind our **hash table** implementation:

- `capacity` - the number of buckets;
- `size` - the number of elements (gets incremented each time we add a new element, and decremented when a removal is performed);
- `buckets` - the actual references to our `oa_pairs` ;
- `probing_fct` - the probing algorithm (eg., linear probing);
- `key_ops` - key related operations;
- `val_ops` - value related operations;

`probing_fct` is a *pointer function* that allows our users to choose the algorithm for probing.

For example, for linear probing, we can come with something like this:

```
static inline void oa_hash_lp_idx(oa_hash *htable, size_t *idx) {
    (*idx)++;
    if ((*idx)==htable->capacity) {
        (*idx) = 0;
    }
}
```

`oa_hash_lp_idx` is a simple function that increments the index (`idx`). If we reach the end of the *bucket array* we simply start again.

We can also write a function for **quadratic probing** later and pass it to the **hash table**. It's good to offer some flexibility, after all.

A few constants are also defined:

```
#define OA_HASH_LOAD_FACTOR (0.75)
#define OA_HASH_GROWTH_FACTOR (1<<2)
#define OA_HASH_INIT_CAPACITY (1<<12)
```

- `OA_HASH_LOAD_FACTOR` - The maximum accepted load factor.
- `OA_HASH_GROWTH_FACTOR` - The growth factor.
- `OA_HASH_INIT_CAPACITY` - The initial capacity (nuber of buckets for the **hash table**).

The interface

The interface is again relatively straightforward:

```
// Creating  anew hash table
oa_hash* oa_hash_new(oa_key_ops key_ops, oa_val_ops val_ops, void (*probing_fct)(struct oa_hash_s *htable, size_t *from_idx));
oa_hash* oa_hash_new_lp(oa_key_ops key_ops, oa_val_ops val_ops);
// Destructor like method for destroying the hashtable
void oa_hash_free(oa_hash *htable);
// Adding, Reading, Deleting entries
void oa_hash_put(oa_hash *htable, const void *key, const void *val);
void *oa_hash_get(oa_hash *htable, const void *key);
void oa_hash_delete(oa_hash *htable, const void *key);
void oa_hash_print(oa_hash *htable, void (*print_key)(const void *k), void (*print_val)(const void *v));

// Pair related
oa_pair *oa_pair_new(uint32_t hash, const void *key, const void *val);

// String operations
uint32_t oa_string_hash(const void *data, void *arg);
void* oa_string_cp(const void *data, void *arg);
bool oa_string_eq(const void *data1, const void *data2, void *arg);
void oa_string_free(void *data, void *arg);
void oa_string_print(const void *data);
```

Creating/Destroying a hash table

It's always a good practice to start writing constructor-like and destructor-like functions for `oa_hash` structure.

- `oa_hash_new` is the constructor-like function for dynamically allocating an `oa_hash` function on the heap;
- `oa_hash_free` is a destructor-like function that frees the memory associated with `oa_hash` and its elements.

```
oa_hash* oa_hash_new(
    oa_key_ops key_ops,
    oa_val_ops val_ops,
    void (*probing_fct)(struct oa_hash_s *htable, size_t *from_idx))
{
    oa_hash *htable;

    htable = malloc(sizeof(*htable));
    if (NULL==htable) {
        fprintf(stderr,"malloc() failed in file %s at line # %d", __FILE__,__LINE__);
        exit(EXIT_FAILURE);
    }

    htable->size = 0;
    htable->capacity = OA_HASH_INIT_CAPACITY;
    htable->val_ops = val_ops;
    htable->key_ops = key_ops;
    htable->probing_fct = probing_fct;

    htable->buckets = malloc(sizeof(*(htable->buckets)) * htable->capacity);
    if (NULL==htable->buckets) {
        fprintf(stderr,"malloc() failed in file %s at line # %d", __FILE__,__LINE__);
        exit(EXIT_FAILURE);
    }

    for(int i = 0; i < htable->capacity; i++) {
        htable->buckets[i] = NULL;
    }

    return htable;
}
```

In practice, it's not a good idea to free the memory associated with the pairs, but for simplicity our implementation frees the memory:

```
void oa_hash_free(oa_hash *htable) {
    for(int i = 0; i < htable->capacity; i++) {
        if (NULL!=htable->buckets[i]) {
            htable->key_ops.free(htable->buckets[i]->key, htable->key_ops.arg);
            htable->val_ops.free(htable->buckets[i]->val, htable->val_ops.arg);
        }
        free(htable->buckets[i]);
    }
    free(htable->buckets);
    free(htable);
}
```

Given the fact we are going to use **linear probing** for our implementation we can *overload* the allocator to use the previously defined algorithm for **linear probing**:

```
oa_hash* oa_hash_new_lp(oa_key_ops key_ops, oa_val_ops val_ops) {
    return oa_hash_new(key_ops, val_ops, oa_hash_lp_idx);
}
```

Modeling Tombstones

We will need a way to model **tombstones** for our **hash table**. In this regard we will call a tombstone a non-`NULL` `oa_pair` element with `oa_pair->hash=0`, `oa_pair->key=NULL` and `oa_pair->val=NULL`.

Then we write a function that checks if a certain bucket is a **tombstone**:

```
inline static bool oa_hash_is_tombstone(oa_hash *htable, size_t idx) {
    if (NULL==htable->buckets[idx]) {
        return false;
    }
    if (NULL==htable->buckets[idx]->key &&
        NULL==htable->buckets[idx]->val &&
        0 == htable->buckets[idx]->key) {
            return true;
    }
    return false;
}
```

And another function that puts a tombstone at given bucket index every time we delete an element:

```
inline static void oa_hash_put_tombstone(oa_hash *htable, size_t idx) {
    if (NULL != htable->buckets[idx]) {
        htable->buckets[idx]->hash = 0;
        htable->buckets[idx]->key = NULL;
        htable->buckets[idx]->val = NULL;
    }
}
```

Growing the bucket capacity if needed

Let's define a metric to track down the load on the *buckets*: `load_factor=size/capacity`. If the load factor is bigger than `0.75`, our **hash table** performance might drop.

In this regard, we will define a function `oa_hash_should_grow` that will check after each addition if we need to grow the **hash table**.

```
#define OA_HASH_LOAD_FACTOR (0.75)
inline static bool oa_hash_should_grow(oa_hash *htable) {
    return (htable->size / htable->capacity) > OA_HASH_LOAD_FACTOR;
}
```

We will continue by writing the growth method `oa_hash_grow`. This method will:

- check if the new bucket capacity doesn't overflow;
- allocate a new memory zone for the new buckets;
- perform a complete rehash of all the elements (skipping any potential tombstone).
- free the memory associated with the old buckets.

```c
inline static void oa_hash_grow(oa_hash *htable) {
    uint32_t old_capacity;
    oa_pair **old_buckets;
    oa_pair *crt_pair;

    // Check if the new bucket capacity doesn't overflow;
    uint64_t new_capacity_64 = (uint64_t) htable->capacity * OA_HASH_GROWTH_FACTOR;
    if (new_capacity_64 > SIZE_MAX) {
        fprintf(stderr, "re-size overflow in file %s at line # %d", __FILE__,__LINE__);
        exit(EXIT_FAILURE);
    }

    old_capacity = htable->capacity;
    old_buckets = htable->buckets;

    // Allocate a new memory zone for the new buckets;
    htable->capacity = (uint32_t) new_capacity_64;
    htable->size = 0;
    htable->buckets = malloc(htable->capacity * sizeof(*(htable->buckets)));

    if (NULL == htable->buckets) {
        fprintf(stderr,"malloc() failed in file %s at line # %d", __FILE__,__LINE__);
        exit(EXIT_FAILURE);
    }

    for(int i = 0; i < htable->capacity; i++) {
        htable->buckets[i] = NULL;
    };

    // Perform a complete rehash of all the elements (skipping any potential tombstone).
    for(size_t i = 0; i < old_capacity; i++) {
        crt_pair = old_buckets[i];
        if (NULL!=crt_pair && !oa_hash_is_tombstone(htable, i)) {
            oa_hash_put(htable, crt_pair->key, crt_pair->val);
            htable->key_ops.free(crt_pair->key, htable->key_ops.arg);
            htable->val_ops.free(crt_pair->val, htable->val_ops.arg);
            free(crt_pair);
        }
    }

    // Free the memory associated with the old buckets.
    free(old_buckets);
}
```

Getting the correct index

The next step is to implement a function that retrieves the bucket where we perform `oa_hash_get`, `oa_hash_put` or `oa_hash_delete` operations.

```c
static size_t oa_hash_getidx(oa_hash *htable, size_t idx, uint32_t hash_val, const void *key, enum oa_ret_ops op) {
    do {
        if (op==PUT && oa_hash_is_tombstone(htable, idx)) {
            break;
        }
        if (htable->buckets[idx]->hash == hash_val &&
            htable->key_ops.eq(key, htable->buckets[idx]->key, htable->key_ops.arg)) {
            break;
        }
        htable->probing_fct(htable, &idx);
    } while(NULL!=htable->buckets[idx]);
    return idx;
}
```

The input params of the function are:

- `htable` - this represents the **hash table** where we operate;
- `idx` - this is the starting index from where we will start probing (using `htable->probing_fct`, the **pointer function** supplied at construction-time);
- `hash_val` - the `hash` of the value we are searching;
- `key` - the `key` for which we are operating;
- `op` - the operation. For example, if we `PUT` an element, tombstones won't be ignored, but if we use this function tp `GET` or `DELETE` elements, tombstones will act just as normal elements.

`idx`, or the starting index is usually obtained like this:

```
uint32_t hash_val = htable->key_ops.hash(key, htable->key_ops.arg);
size_t idx = hash_val % htable->capacity;
```

`idx` is the *natural* choice for each entry, but in the case of **hash collisions**, we need to find another free index. Here's where the `oa_hash_getidx` functions come into play. It returns the most natural choice starting from `idx`.

Adding an element to the hash table

The steps to add (`oa_hash_put`) an element inside the **hash table** are the following:

- We grow the **hash table** if necesary;
- We check if there is an empty bucket to insert the element;
- If it's not empty, we probe for another suitable bucket;
- We insert the element;

```
void oa_hash_put(oa_hash *htable, const void *key, const void *val) {

    if (oa_hash_should_grow(htable)) {
        oa_hash_grow(htable);
    }

    uint32_t hash_val = htable->key_ops.hash(key, htable->key_ops.arg);
    size_t idx = hash_val % htable->capacity;

    if (NULL==htable->buckets[idx]) {
        // Key doesn't exist & we add it anew
        htable->buckets[idx] = oa_pair_new(
                hash_val,
                htable->key_ops.cp(key, htable->key_ops.arg),
                htable->val_ops.cp(val, htable->val_ops.arg)
        );
    } else {
        // // Probing for the next good index
        idx = oa_hash_getidx(htable, idx, hash_val, key, PUT);

        if (NULL==htable->buckets[idx]) {
            htable->buckets[idx] = oa_pair_new(
                hash_val,
                htable->key_ops.cp(key, htable->key_ops.arg),
                htable->val_ops.cp(val, htable->val_ops.arg)
            );
        } else {
            // Update the existing value
            // Free the old values
            htable->val_ops.free(htable->buckets[idx]->val, htable->val_ops.arg);
            htable->key_ops.free(htable->buckets[idx]->key, htable->key_ops.arg);
            // Update the new values
            htable->buckets[idx]->val = htable->val_ops.cp(val, htable->val_ops.arg);
            htable->buckets[idx]->key = htable->val_ops.cp(key, htable->key_ops.arg);
            htable->buckets[idx]->hash = hash_val;
        }
    }
     htable->size++;
}
```

Removing an element from the hash table

The steps to remove (`oa_hash_delete`) an element from the **hash table** are the following:

- We check if the bucket is empty. If it is, we simply return;
- If the bucket is not empty we check if the element should be removed or not;
- If it's not the good element, it means we need to probe to find it.
- We free the memory associated with the element;

```
void oa_hash_delete(oa_hash *htable, const void *key) {
    uint32_t hash_val = htable->key_ops.hash(key, htable->key_ops.arg);
    size_t idx = hash_val % htable->capacity;

    if (NULL==htable->buckets[idx]) {
        return;
    }

    idx = oa_hash_getidx(htable, idx, hash_val, key, DELETE);
    if (NULL==htable->buckets[idx]) {
        return;
    }

    htable->val_ops.free(htable->buckets[idx]->val, htable->val_ops.arg);
    htable->key_ops.free(htable->buckets[idx]->key, htable->key_ops.arg);

    oa_hash_put_tombstone(htable, idx);
}
```

Retrieving an element from the hash table

Retrieving an element is again a straight-forward function to implement:

```
void *oa_hash_get(oa_hash *htable, const void *key) {
    uint32_t hash_val = htable->key_ops.hash(key, htable->key_ops.arg);
    size_t idx = hash_val % htable->capacity;

    if (NULL==htable->buckets[idx]) {
        return NULL;
    }

    idx = oa_hash_getidx(htable, idx, hash_val, key, GET);

    return (NULL==htable->buckets[idx]) ?
         NULL : htable->buckets[idx]->val;
}
```

Putting all together

```
int main(int argc, char *argv[]) {
    oa_hash *h = oa_hash_new(oa_key_ops_string, oa_val_ops_string, oa_hash_lp_idx);

    oa_hash_put(h, "Bucharest", "Romania");
    oa_hash_put(h, "Sofia", "Bulgaria");

    printf("%s\n", oa_hash_get(h, "Bucharest"));
    printf("%s\n", oa_hash_get(h, "Sofia"));

    return 0;
}
```

If you want to read more about **Open Addressing** techniques, checkout my new blog post. The code is in Java, but hey, if you can read C, Java should be easier to grasp.

References

- Hash Functions and Hash Tables, Breno Helfstein Moura (https://linux.ime.usp.br/~brelf/mac0499/monografia.pdf)
- Crafting Interpreters - Hash Tables, Robert Nystrom (https://craftinginterpreters.com/hash-tables.html)
- Hash functions (https://www.cs.hmc.edu/~geoff/classes/hmc.cs070.200101/homework10/hashfuncs.html)
- CS240 – Lecture Notes: Hashing (https://www.cpp.edu/~ftang/courses/CS240/lectures/hashing.htm)
- Hash Functions: An Empirical Comparison (https://www.codeproject.com/Articles/32829/Hash-Functions-An-Empirical-Comparison)
- CSci 335 Software Design and Analysis - Chapter 5 Hashing and Hash Tables, Steward Weiss (http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci335/lecture_notes/chapter05.pdf)
- CSE 241 Algorithms and Data Structures - Chosing hash functions (https://classes.engineering.wustl.edu/cse241/handouts/hash-functions.pdf)
- Integer hash functions, Thomas Wang (https://gist.github.com/badboy/6267743)
- Notes on Data Structures and Programming Techniques, James Aspnes (http://www.cs.yale.edu/homes/aspnes/classes/223/notes.html#hashTables)
- The FNV Non-Cryptographic Hash Algorithm (https://datatracker.ietf.org/doc/html/draft-eastlake-fnv-17)
- Hash Tables - Open Addressing vs Chaining (https://www.reddit.com/r/algorithms/comments/9bwzj5/hash_tables_open_addressing_vs_chaining/);
- Optimizing software in C++, Agner Fog (https://www.agner.org/optimize/optimizing_cpp.pdf)
- Why did the designers of Java preferred chaining over open addressing (https://stackoverflow.com/questions/12019434/why-did-the-language-designers-of-java-preferred-chaining-over-open-addressing-f)
- Deletion from hash tables without tombstones (https://attractivechaos.wordpress.com/2019/12/28/deletion-from-hash-tables-without-tombstones/)
- Traits (https://en.wikipedia.org/wiki/Trait_(computer_programming))

Updated: October 2, 2021

COMMENTS