The Joys of Hashing

Hash Table Programming with C

Thomas Mailund

Apress®

The Joys of Hashing

Hash Table Programming with C

Thomas Mailund

Apress®

The Joys of Hashing: Hash Table Programming with C

Thomas Mailund
Aarhus N, Denmark

ISBN-13 (pbk): 978-1-4842-4065-6 ISBN-13 (electronic): 978-1-4842-4066-3
https://doi.org/10.1007/978-1-4842-4066-3

Library of Congress Control Number: 2019932793

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484240656. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author vii

About the Technical Reviewer ix

Acknowledgements xi

Chapter 1: The Joys of Hashing 1

Chapter 2: Hash Keys, Indices, and Collisions 7

Mapping from Keys to Indices 9

Risks of Collisions 12

Mapping Hash Keys to Bins 18

Chapter 3: Collision Resolution, Load Factor, and Performance 21

Chaining 21

Linked Lists 22

Chained Hashing Collision Resolution 25

Open Addressing 27

Probing Strategies 32

Load and Performance 35

Theoretical Runtime Performance 35

Experiments 43

Chapter 4: Resizing 49

Amortizing Resizing Costs 50

Resizing Chained Hash Tables 57

Resizing Open Addressing Hash Tables 61
Theoretical Considerations for Choosing the Load Factor 65
Experiments 68
Resizing When Table Sizes Are Not Powers of Two 75
Dynamic Resizing..85

Chapter 5: Adding Application Keys and Values..............................101

Hash Sets 103
Chained Hashing...104
Updating Linked Lists 105
Updating the Hash Table...110
Open Addressing 114
Implementing Hash Maps 120
Chained Hashing...121
Updates to the Linked Lists 121
Updates to the Hash Table..127
Open Addressing 132

Chapter 6: Heuristic Hash Functions 139

What Makes a Good Hash Function? 141
Hashing Computer Words...143
Additive Hashing...146
Rotating Hashing 148
One-at-a-Time Hashing 152
Jenkins Hashing 161
Hashing Strings of Bytes...166

Chapter 7: Universal Hashing 173
Uniformly Distributed Keys 174
Universal Hashing 175
Stronger Universal Families 176
Binning Hash Keys 177
Collision Resolution Strategies 178
Constructing Universal Families 179
Nearly Universal Families 180
Polynomial Construction for *k*-Independent Families 180
Tabulation Hashing 182
Performance Comparison 186
Rehashing 190
Chapter 8: Conclusions 199
Bibliography 201
Index 203

About the Author

Thomas Mailund is an associate professor in bioinformatics at Aarhus University, Denmark. He has a background in math and computer science. For the past decade, his main focus has been on genetics and evolutionary studies, particularly comparative genomics, speciation, and gene flow between emerging species.

He is the author of *Domain-Specific Languages in R, Beginning Data Science in R, Functional Programming in R*, and *Metaprogramming in R*, all from Apress, as well as other books.

About the Technical Reviewer

Michael Thomas has worked in software development for over 20 years as an individual contributor, team lead, program manager, and vice president of engineering. Michael has over 10 years of experience working with mobile devices. His current focus is in the medical sector using mobile devices to accelerate information transfer between patients and health care providers.

Acknowledgments

I am very grateful to Rasmus Pagh for comments on the manuscript, suggestions for topics to add, and correcting me when I have been imprecise or downright wrong. I am also grateful to Anders Halager for many discussions about implementation details and bit fiddling. I am also grateful to Shiella Balbutin for proofreading the book.

CHAPTER 1

The Joys of Hashing

This book is an introduction to the hash table data structure. When implemented and used appropriately, hash tables are exceptionally efficient data structures for representing sets and lookup tables. They provide constant time, low overhead, insertion, deletion, and lookup. The book assumes the reader is familiar with programming and the C programming language. For the theoretical parts of the book, it also assumes some familiarity with probability theory and algorithmic theory.

Hash tables are constructed from two basic ideas: reducing application keys to a *hash key*, a number in the range from 0 to some $N - 1$, and mapping that number into a smaller range from 0 to $m - 1$, $m \ll N$. We can use the small range to index into an array with constant time access. Both ideas are simple, but how they are implemented in practice affects the efficiency of hash tables.

Consider Figure 1-1. This figure illustrates the main components of storing values in a hash table: application values, which are potentially complex, are mapped to hash keys, which are integer values in a range of size N, usually zero to $N - 1$. In the figure, $N = 64$. Doing this simplifies the representation of the values; now you only have integers as keys, and if N is small, you can store them in an array of size N. You use their hash keys as their index into the array. However, if N is large, this is not feasible. If, for example, the space of hash keys is 32-bit integers, then $N = 4, 294, 967, 295$, slightly more than four billion. An array of bytes of this size would,

T. Mailund, *The Joys of Hashing*, https://doi.org/10.1007/978-1-4842-4066-3_1

therefore, take up more than four gigabytes of space. To be able to store pointers or integers, simple objects, you would need between four and eight times as much memory. It is impractical to use this size of an array to store some application keys.

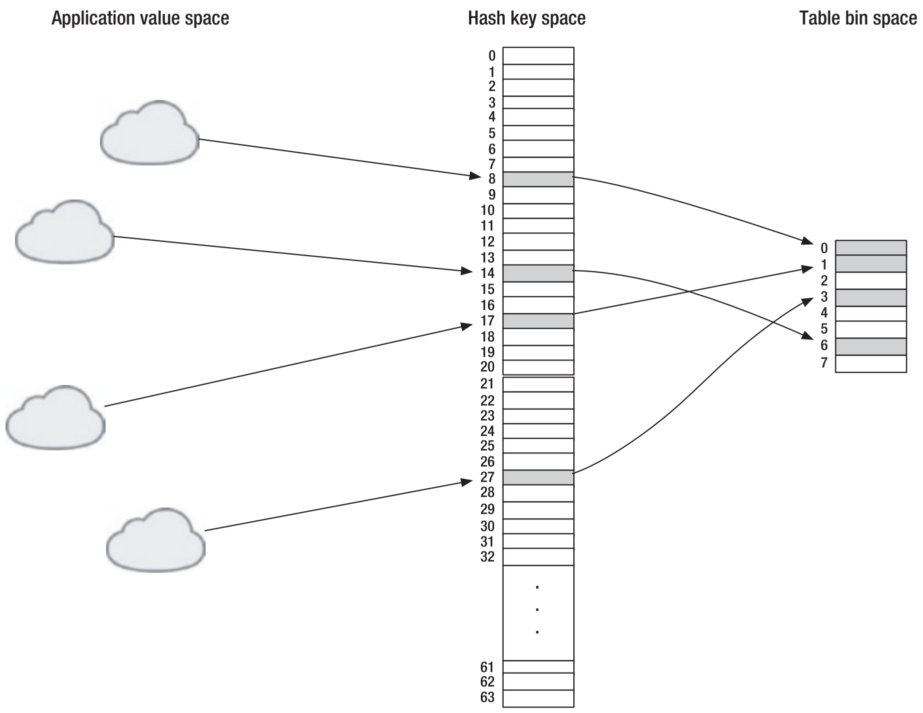

Figure 1-1. *Values map to hash keys that then map to table bins*

Even if N is considerably smaller than four-byte words, if you plan to store $n \ll N$ keys, you waste a lot of space to have the array. Since this array needs to be allocated and initialized, merely creating it will cost you $O(N)$. Even if you get constant time insertion and deletion into such an array, the cost of producing it can easily swamp the time your algorithm will spend while using the array. If you want a table that is efficient, you should be able to both initialize it and use it to insert or delete n keys, all in time $O(n)$. Therefore, N should be in $O(n)$.

The typical solution to this is to keep N large but have a second step that reduces the hash key range down to a smaller bin range of size m with $m \in O(n)$; in the example, you use $m = 8$. If you keep m small, as in $O(n)$, you can allocate and initialize it in linear time, and you can get any bin in it in constant time. To insert, check, or delete an element in the table, you map the application value to its hash key and then map the hash key to a bin index.

You reduce values to bin indices in two steps because the first step, mapping data from your application domain to a number, is program-specific and cannot be part of a general hash table implementation.[1] Moving from large integer intervals to smaller, however, can be implemented as part of the hash table. If you resize the table to adapt it to the number of keys you store in it, you need to change m. You do not want the application programmer to provide separate functions for each m. You can think of the hash key space, $[N] = [0, \dots, N - 1]$, as the interface between the application and the data structure. The hash table itself can map from this space to indices in an array, $[m] = [0, \dots, m - 1]$.

The primary responsibility of the first step is to reduce potentially complicated application values to simpler hash keys, such as to map application-relevant information like positions on a board game or connections in a network down to integers. These integers can then be handled by the hash table data structure. A second responsibility of the function is to make the hash keys uniformly distributed in the range $[N]$. The binning strategy for mapping hash keys to bins assumes that the hash keys are uniformly distributed to distribute keys into bins evenly. If this is violated, the data structure does not guarantee (expected) constant time operations. Here, you can add a third, middle step that maps from

[1]In some textbooks, you will see the hashing step and the binning step combined and called hashing. Then you have a single function that maps application-specific keys directly to bins. I prefer to consider this as two or three separate functions, and it usually is implemented as such.

$[N] \rightarrow [N]$ and scrambles the application hash keys to hash keys with a better distribution; see Figure 1-2. These functions can be application-independent and part of a hash table library. You will return to such functions in Chapter 6 and Chapter 7. Having a middle step does not eliminate the need for application hash functions. You still need to map complex data into integers. The middle step only alleviates the need for an even distribution of keys. The map from application keys to hash keys still has some responsibility for this, though. If it maps different data to the same hash keys, then the middle step cannot do anything but map the same input to the same output.

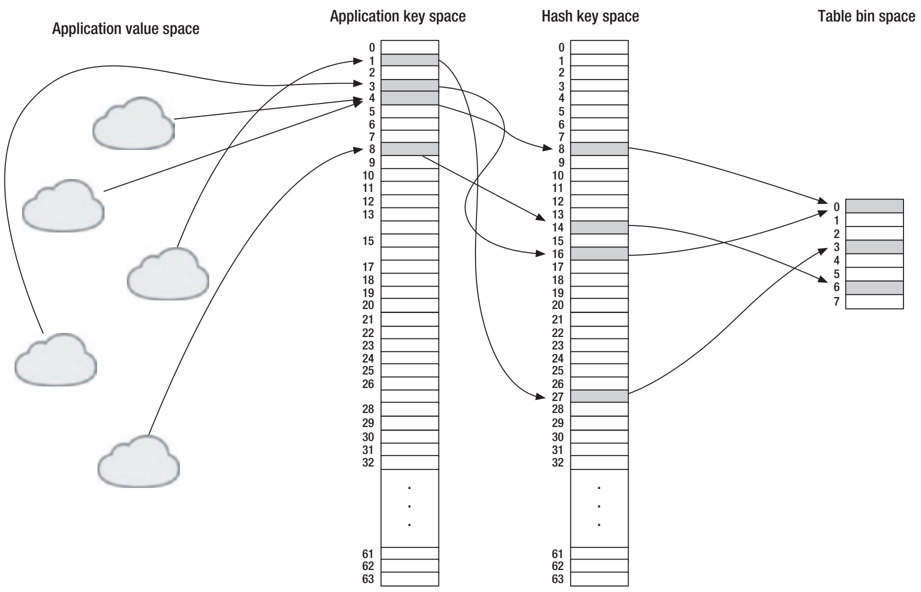

Figure 1-2. *If the application maps values to keys, but they are not uniformly distributed, then a hashing step between the application and the binning can be added*

Strictly speaking, you do not need the distribution of hash keys to be uniform as long as the likelihood of two different values mapping to the same key is highly unlikely. The goal is to have uniformly distributed hash keys, as these are easiest to work with when analyzing theoretical performance. The runtime results referred to in Chapter 3 assume this, and therefore, we will as well. In Chapter 7, you will learn techniques for achieving similar results without the assumption.

The book is primarily about implementing the hash table data structure and only secondarily about hash functions. The concerns when implementing hash tables are these: given hash keys with application values attached to them, how do you represent the data such that you can update and query tables in constant time? The fundamental idea is, of course, to reduce hash keys to bins and then use an array of bins containing values. In the purest form, you can store your data values directly in the array at the index the hash function and binning functions provide but if m is relatively small compared to the number of data values, then you are likely to have collisions: cases where two hash keys map to the same bin. Although different values are unlikely to hash to the same key in the range $[N]$, this does not mean that collisions are unlikely in the range $[m]$ if m is smaller than N (and as the number of keys you insert in the table, n, approaches m, collisions are guaranteed). Dealing with collisions is a crucial aspect of implementing hash tables, and a topic we will deal with for a sizeable part of this book.

CHAPTER 2

Hash Keys, Indices, and Collisions

As mentioned in the introduction, this book is primarily about implementing hash tables and not hash functions. So to simplify the exposition, assume that the data values stored in tables are identical to the hash keys. In Chapter 5, you will address which changes you have to make to store application data together with keys, but for most of the theory of hash tables you only need to consider hash keys; everywhere else, you will view additional data as black box data and just store their keys. While the code snippets below cover all that you need to implement the concepts in the chapter, you cannot easily compile them from the book, but you can download the complete code listings from `https://github.com/mailund/JoyChapter2`.

Assume that the keys are uniformly distributed in the interval $[N] = [0, \ldots, N - 1]$ where N is the maximum `uint32_t` and consider the most straightforward hash table. It consists of an array where you can store keys and a number holding the size of the table, m. To be able to map from the range $[N]$ to the range $[m]$, you need to remember m. You store this number in the variable `size` in the structure below. You cannot use a special key to indicate that an entry in the table is not occupied, so you will use a structure called `struct bin` that contains a flag for this.

```
struct bin {
    int is_free : 1;
    uint32_t key;
};

struct hash_table {
    struct bin *table;
    uint32_t size;
};
```

Functions for allocating and deallocating tables can then look like this:

```
struct hash_table *empty_table(uint32_t size)
{
    struct hash_table *table =
        (struct hash_table*)malloc(sizeof(struct hash_table));
    table->table = (struct bin *)malloc(size * sizeof
    (struct bin));
    for (uint32_t i = 0; i < size; ++i) {
        struct bin *bin = & table->table[i];
        bin->is_free = true;
    }
    table->size = size;
    return table;
}
void delete_table(struct hash_table *table)
{
    free(table->table);
    free(table);
}
```

The operations you want to implement on hash tables are the insertion and deletion of keys and queries to test if a table holds a given key. You use this interface to the three operations:

```
void insert_key (struct hash_table *table, uint32_t key);
bool contains_key (struct hash_table *table, uint32_t key);
void delete_key (struct hash_table *table, uint32_t key);
```

Mapping from Keys to Indices

When you have to map a hash key from [*N*] down to the range of the indices in the array, [*m*], the most straightforward approach is to take the remainder of a division by *m*:

```
    unsigned int index = key % table->size;
```

You then use that index to access the array. Assuming that you never have collisions when doing this, the implementation of the three operations would then be as simple as this:

```
void insert_key(struct hash_table *table, uint32_t key)
{
    uint32_t index = key % table->size;
    struct bin *bin = & table->table[index];
    if (bin->is_free) {
        bin->key = key;
        bin->is_free = false;
    } else {
        // There is already a key here, so we have a
        // collision. We cannot deal with this yet.
    }
}
```

```
bool contains_key(struct hash_table *table, uint32_t key)
{
    uint32_t index = key % table->size;
    struct bin *bin = & table->table[index];
    if (!bin->is_free && bin->key == key) {
        return true;
    } else {
        return false;
    }
}

void delete_key(struct hash_table *table, uint32_t key)
{
    uint32_t index = key % table->size;
    struct bin *bin = & table->table[index];
    if (bin->key == key) {
        bin->is_free = true;
    }
}
```

When inserting an element, you place the value at the index given by the mapping from the space of hash keys to the range of the array. Deleting a key is similarly simple: you set the flag in the bin to `false`. To check if the table contains a key, you check that the bin is not free and that it contains the right key. If you assume that the only way you can get an index at a given index is if you have the key value, this would be correct. However, you also usually check that the application keys match the key in the hash table, not just that the hash keys match. In this implementation, the application keys and hash keys are the same, so you check if the hash keys are identical only. Because, of course, you *could* have a situation where two different hash keys would map to the same bin index, even if the hash keys never collide. The space of bin indices, after all, is much smaller than the space of keys.

Collisions of hash values are rare events if they are the results of a well-designed hash function. Although collisions of hash keys are rare, it does not imply that you cannot get collisions in the indices. The range $[N]$ is usually vastly larger than the array indices in the range $[m]$. Two different hash keys can easily end up in the same hash table bin; see Figure 2-1. Here, you have hash keys in the space of size $N = 64$ and only $m = 8$ bins. The numbers next to the hash keys are written in octal, and you map keys to bins by extracting the lower eight bits of the key, which corresponds to the last digit in the octal representation. The keys 8 and 16, or 10_8 and 20_8 in octal, both map to bin number 0, so they collide in the table.

The figure is slightly misleading since the hash space is only a factor of eight larger than the size of the hash table. In any real application, the keys range over a much wider interval than could ever be represented in a table. In the setup in this book, the range $[N]$ maps over all possible unsigned integers, which in most C implementations means all possible memory addresses on your computer. This space is much larger than what you could reasonably use for an array; if you had to use your entire computer memory for a hash table, you would have no space for your actual computer program. Each value might map to a unique hash key, but when you have to map the hash keys down to a smaller range to store values in a table, you are likely to see collisions.

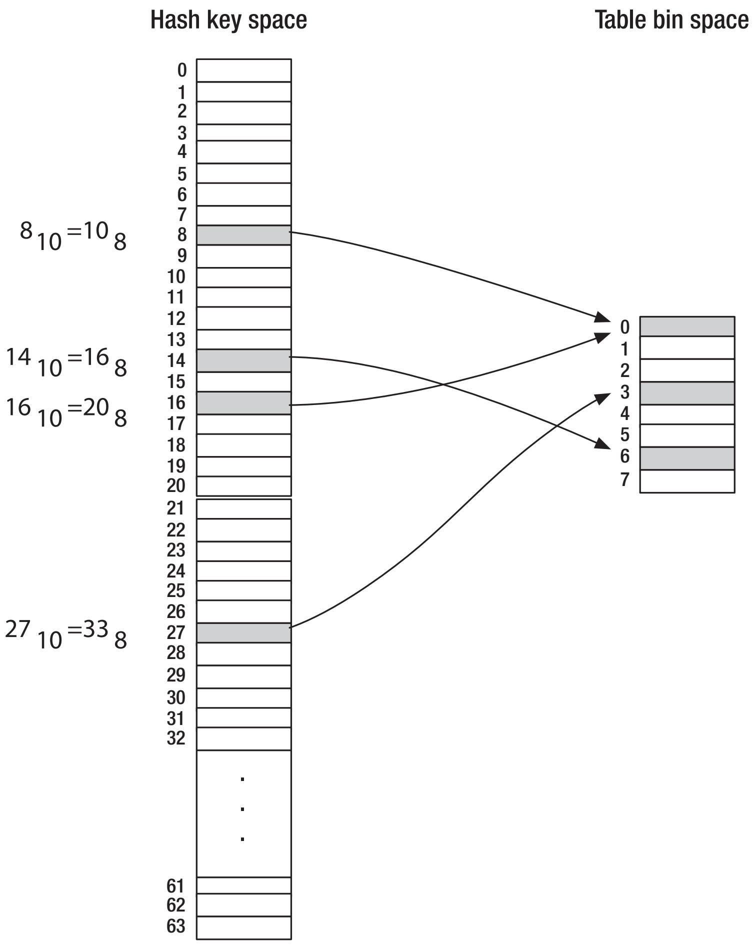

Figure 2-1. *Collisions of hash keys when binning them*

Risks of Collisions

Assuming a uniform distribution of hash keys, let's do some back-of-the-envelope calculations of collisions probabilities. The chances of collisions are surprisingly high once the number of values approaches even a small fraction of the number of indices we can hit. To figure out the chances of collisions, let's use the *birthday paradox* (`https://en.wikipedia.org/wiki/Birthday`). In a room of *n* people, what is the probability that two or more have the same birthday? Ignoring leap years, there are 365 days in a year, so how many people do we need for the chance that at least two have the same birthday is above one half? This number, *n*, turns out to be very low. If we assume that each date is equally likely as a birthday, then with only 23 people we would expect a 50% chance that at least two share a birthday.

Let's phrase the problem of "at least two having the same birthday" a little differently. Let's ask "what is the probability that all *n* people have *different* birthdays?" The answer to the first problem will then be one minus the answer to the second.

To answer the second problem, we can reason like this: out of the *n* people, the first birthday hits 1 out of 365 days without any collisions. The second person, if we avoid collisions, has to hit 1 of the remaining 364 days. The third person has to have his birthday on 1 of the 363 remaining days. Continuing this reasoning, the probability of no collisions in birthdays of *n* people is

$$\frac{365}{365}\times\frac{364}{365}\times\ldots\times\frac{365-n+1}{365}.$$

where 1 minus this product is then the risk of at least one collision when there are *n* people in the room. Figure 2-2 shows this probability as a function of the number of people. The curve crosses the point of 50% collision risk between 22 and 23.