The Joys of Hashing

Hash Table Programming with C

—

Second Edition

—

Thomas Mailund

Apress®

The Joys of Hashing

Hash Table Programming with C

Second Edition

Thomas Mailund

Apress®

The Joys of Hashing: Hash Table Programming with C, Second Edition

Thomas Mailund
Aarhus N, Denmark

ISBN-13 (pbk): 979-8-8688-0825-8 ISBN-13 (electronic): 979-8-8688-0826-5
https://doi.org/10.1007/979-8-8688-0826-5

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Melissa Duffy
Development Editor: James Markham
Coordinating Editor: Gryffin Winkler
Copy Editor: Kezia Endsley

Cover designed by eStudioCalamar

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

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. 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 booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail 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 http://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 (https://github.com/Apress). For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

Table of Contents

About the Author

Thomas Mailund is a former associate professor in bioinformatics at Aarhus University, Denmark, and currently a senior software architect at the quantum computing company Kvantify. He has a background in math and computer science, including experience programming and teaching in the C and R programming languages. For the last decade, his main focus has been on genetics and evolutionary studies, particularly comparative genomics, speciation, and gene flow between emerging species.

About the Technical Reviewer

Megan J. Hirni currently teaches and conducts research at the University of Missouri-Columbia, focusing on statistical methodology applied in health, social sciences, and education. Apart from her passion for coding, Megan enjoys meeting new people and exploring diverse research disciplines.

Acknowledgments

I am very grateful to Rasmus Pagh for his comments on the manuscript, for suggestions on topics to add, and for correcting me when I was 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

Introduction

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. This book assumes you are familiar with programming and the C programming language. The theoretical parts of the book also assume some familiarity with probability theory and algorithmic theory, but nothing beyond what you would learn in an introductory course.

Hash tables are constructed from two fundamental ideas: reducing application keys to a *hash key*—a number ranging from 0 to some $N-1$—and mapping that number into a smaller range from 0 to $m-1$, $m \ll N$. You 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, which illustrates the main components of storing values in a hash table. Potentially complex application values are mapped to *hash keys*, which are integer values in a range of size N, usually 0 to $N-1$. In the figure, $N = 64$. Doing this simplifies the representation of the values. You now only have integers as keys, and if N is small, you can store the integers 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 take up more than 4GB of space. You would need between four and eight times as

T. Mailund, *The Joys of Hashing*, https://doi.org/10.1007/979-8-8688-0826-5_1

much memory to store pointers or integers, for example, which are still simple objects. It is impractical to use this size of an array to store some application keys.

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 an efficient table, you should be able to 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 is to keep N large, but include a second step that reduces the hash key range to a smaller bin-range of size m with $m \in O(n)$—this example uses $m = 8$. If you keep m small (i.e., in $O(n)$), you can allocate and initialize it in linear time and 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.

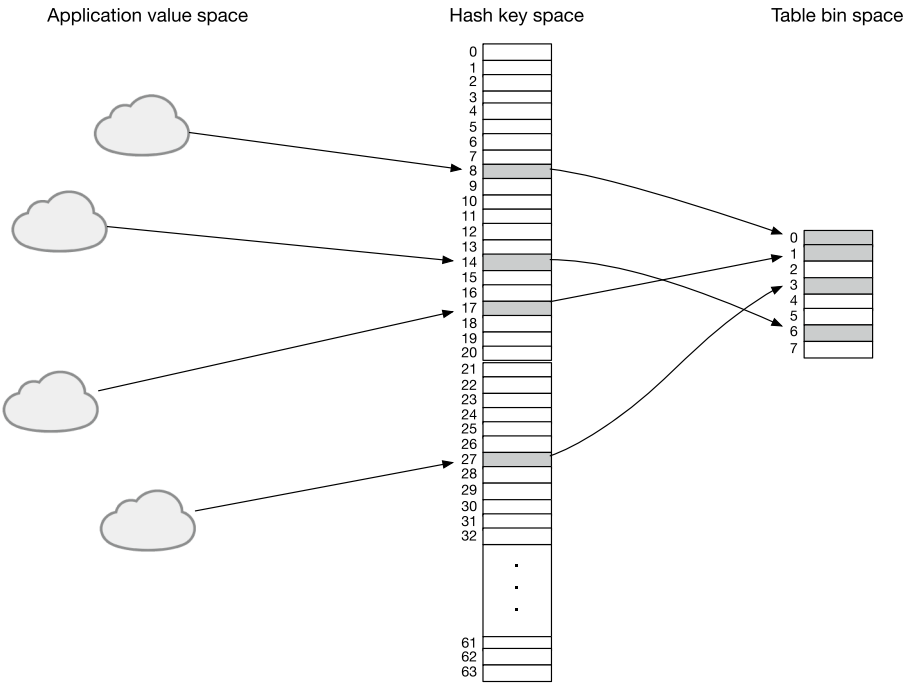

Figure 1-1. *Value maps to hash keys that then maps to table bins*

You can reduce values to bin indices in two steps. 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, you need to change m. You do not want the application programmer to provide separate functions for each m. You can

[1] In some textbooks, you will see the hashing step and the binning step combined, 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 is usually implemented as such.

think of the hash key space, $[N] = [0, \ldots, N-1]$, as the interface between the application and the data structure. The hash table itself can map from this space to indices in an array, $[m] = [0, \ldots, m-1]$.

The primary responsibility of the first step is to reduce potentially complicated application values into simpler hash keys. For example, 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. The 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 evenly into bins. If this is violated, the data structure does not guarantee (expected) constant time operations. Here, you can add a third step between the two previous that maps from $[N] \to [N]$ 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.

Chapters 6 and 7 return to these functions. 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, the middle step cannot do anything but map the same input to the same output.

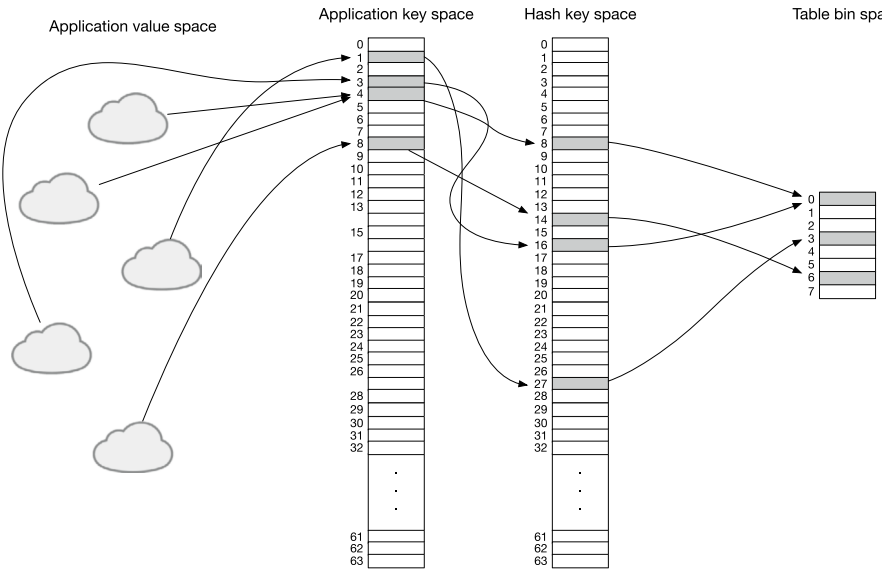

Figure 1-2. *If the application maps values to keys, but they are not uniformly distributed, then a hashing step between the application and the binning can be added*

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 improbable. The goal is to have uniformly distributed hash keys, which are easiest to work with when analyzing theoretical performance. The runtime results in Chapter 3 assume this, and therefore, you can as well. Chapter 7 considers 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. When implementing hash tables, the concerns are these: given hash keys with application values attached to them, how do you represent the data so 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 that the hash and binning functions provide. Still, if m is relatively small compared to the number of data values, you are likely to have *collisions*, which are 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 that's covered in a sizeable portion 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, I initially assume that the data values you store in tables are simply hash keys. Chapter 5 addresses the 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 cover all that you need to implement the concepts in this chapter, you cannot easily compile them from the book, but you can download the complete code listings from `https://github.com/mailund/JoyChapter2`. I did not include the necessary header files in the source code snippets throughout the book, but you can access them in the repository links found at the beginning of each chapter.

I assume that the keys are uniformly distributed in the interval $[N] = [0, \ldots, N-1]$, where N is the maximum `unsigned int`, and consider the most straightforward hash table I can imagine. 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, and that is why you store it. If you always had the same table size, you wouldn't

T. Mailund, *The Joys of Hashing*, https://doi.org/10.1007/979-8-8688-0826-5_2

even need that, and a hash table would be an array. But you will allow for different table sizes (when you get to Chapter 4), so you need to store the number m in the variable `size` using the following structure:

```
struct bin { ... };
struct hash_table {
  struct bin *table;
  unsigned int size;
};
```

If your bins are just an array of hash keys with no further information, you have an interesting problem. If you find a key k in the bin where you expect to find k, does that mean it is actually there? After all, an array is usually uninitialized memory, so it *could* happen that k was there by pure chance. Admittedly, this is extremely unlikely to happen, and I wouldn't worry about it happening in real life if the space of keys is large, but we might as well consider and deal with the issue.

If the bits you have in a bin are precisely the bits you have for hash keys, there is little that you can do about it. You need at least one bit of information to indicate whether an array entry is initialized. There are clever ways of representing such information without putting it in bins, but that puts the extra information elsewhere, in auxiliary data structures. You need a simple table here, so I do not want to go there, now or ever.

A simple solution is to add one bit of information to each bin:

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

struct hash_table {
  struct bin *table;
  unsigned int size;
};
```

That increases the size of the bins and leaves enough bits for the keys and the initialization indicator. Unfortunately, even though you only ask for one bit for the `is_free` flag, you can potentially get a lot more. The `struct bin` has to contain enough memory for both `is_free` and `key`, but your computer does not allocate memory in bit-sized chunks, so the size must be rounded up. Furthermore, the memory alignment of various types will usually result in even more rounding up. If your computer stores integers as four bytes, it might also demand that all integers are at offsets that are multiples of fours, and when it sees a `struct` like this, it will set aside two integers per `struct bin`. So, by adding one bit, you have doubled the bin size.

You should rarely worry about this, but it can be wasteful. Instead, you could remove one bit from the hash keys, using, for example, 31 bits for keys, and then one bit for `is_free`, packing both neatly into a 32-bit integer. In practice, there is not much difference between 31-bit and 32-bit keys, but you have just halved the space of keys, which also feels a bit dramatic. So I won't go there, especially because cutting the key space in half is unnecessary to represent whether a bin is initialized or not. You could reserve a unique key value to indicate that and require that no one uses that hash key for anything else. Zero, for example. Then bins can be `unsigned int`, and you don't need extra space.

```
#define RESERVED_KEY ((unsigned int)0)
struct hash_table {
  unsigned int size;
  unsigned int *bins;
};
```

For the user who has to generate hash keys, avoiding a reserved key is a potential problem, but if that is the case, the previous solution is an adequate fallback choice. In any case, once you get to more complicated tables, you will need more data in bins in any case, and then the extra `is_free` bit will be free, or you will need more special values for reserved

keys, and you will need to deal with these anyway. So, I go with the two cases without complicating it further, and later in the book, you will see more variations on both themes.

A function for allocating a table can then look like this for the variant with `struct bin`:

```
struct hash_table *
new_table(unsigned int size)
{
  // Allocate table and bins
  struct hash_table *table = malloc(sizeof *table);
  table->size = size;
  table->bins = malloc(size * sizeof *table->bins);

  // Set all bins to free
  struct bin *beg = table->bins, *end = beg + size;
  for (struct bin *bin = beg; bin != end; bin++) {
    bin->is_free = true;
  }

  return table;
}
```

And it can look like this for the variant with a reserved key:

```
struct hash_table *
new_table(unsigned int size)
{
  // Allocate table and bins
  struct hash_table *table = malloc(sizeof *table);
  table->bins = malloc(size * sizeof *table->bins);

  // Initialize the bins with the reserved key
  unsigned int *beg = table->bins, *end = beg + size;
```

```
  for (unsigned int *bin = beg; bin != end; bin++) {
    *bin = RESERVED_KEY;
  }

  return table;
}
```

They are pretty similar. In both cases, you allocate the `hash_table` structure and then allocate the bins, after which you iterate through all the bins to initialize them.

I haven't dealt with allocation errors (`malloc()` returning `NULL`) in either function. You could easily do it here. For example, the "reserved key" initialization could look like this:

```
struct hash_table *
new_table(unsigned int size)
{
  // Allocate table and bins
  struct hash_table *table = malloc(sizeof *table);
  unsigned int *bins = malloc(size * sizeof *bins);
  if (!table || !bins) goto error;

  *table = (struct hash_table){.size = size, .bins = bins};

  unsigned int *beg = table->bins, *end = beg + size;
  for (unsigned int *bin = beg; bin != end; bin++) {
    *bin = RESERVED_KEY;
  }

  return table;

error:
  free(table);
  free(bins);
  return NULL;
}
```

However, once you start resizing tables in Chapter 4, dealing with allocation errors gets far more complicated. Especially when every allocation error potentially has to propagate out from deeply nested function calls, and C doesn't have any convenient mechanism for error propagation. While I believe that learning how to handle allocation errors is important, my attempts at doing that for the more complicated code you will see in that chapter overshadowed the hash-table lessons, and the book is about hash tables and not error handling in C. That may be an exciting topic for a later book, but it will not be this one. What I am saying is that I won't be handling `malloc()` errors in the book. If you want, pretend that my `malloc()` is a variant that calls `exit()` if it fails.

One more thing I want to say about memory allocation is this: if you can pack your data into fewer allocations, it is easier to work with. You could have done that by putting the bins in a "flexible array member" as so:

```
struct hash_table {
  unsigned int size;
  struct bin bins[];  // flexible array member
};
```

A flexible array member is an array you declare at the end of a `struct` without specifying its length. If you have such a member, you can allocate the `hash_table` and the `bins` in a single call to `malloc()`:

```
struct hash_table *
new_table(unsigned int size)
{
  // Allocate table and bins
  struct hash_table *table =
    malloc(sizeof *table + size * sizeof *table->bins);

  if (table) {
    table->size = size;
    struct bin *beg = table->bins, *end = beg + size;
```

```
    for (struct bin *bin = beg; bin != end; bin++) {
      bin->is_free = true;
    }
  }

  return table;
}
```

The trick is to allocate enough memory in the `malloc()` call for both the `struct` and the elements you want to put in the array. Here I do that by simply adding the size of the `struct` to the size of the bins array. Depending on the memory layout of the `struct` members, this *might* be slightly more than I need, and I could instead add the offset of the array to the size of the array, but the difference hardly matters.

I don't use a flexible array member in this book, and it is for the same reason that I don't include allocation error handling. While the flexible array member is often helpful, it can get complicated if you need to reallocate memory to grow or shrink your tables. Suppose you allocate one block of memory for the table plus the bins. In that case, you cannot easily add or remove bins later because every pointer to the table has to be updated to point to the newly allocated version. If you have a pointer to a table, and it has a pointer to its bins, you can update the bins pointer once, and everyone will have access to it. Because of this, I allocate bins separately from the `hash_table` structure.

To free a table's memory again, you need to free both the table structure and the bins array. For the two first versions, where you allocated the bins separately, it looks like this:

```
void
delete_table(struct hash_table *table)
{
  free(table->bins);
  free(table);
}
```