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.

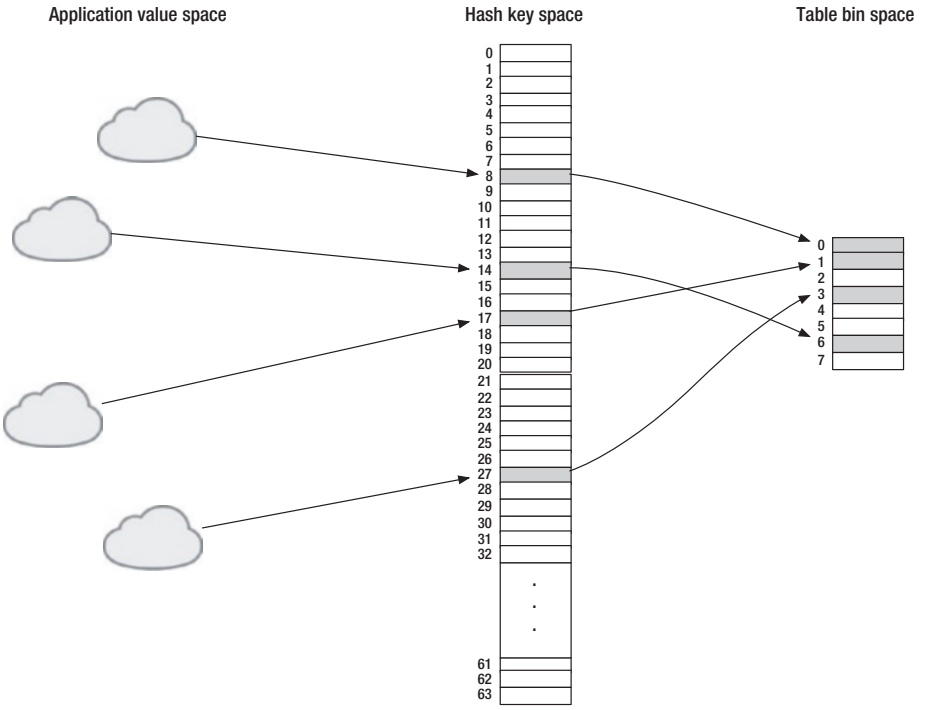

Figure 1-1. *Values map to hash keys that then map to table bins*

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, $[N] = [0, \ldots, N - 1]$, 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 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.

$[N] \to [N]$ 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. 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.

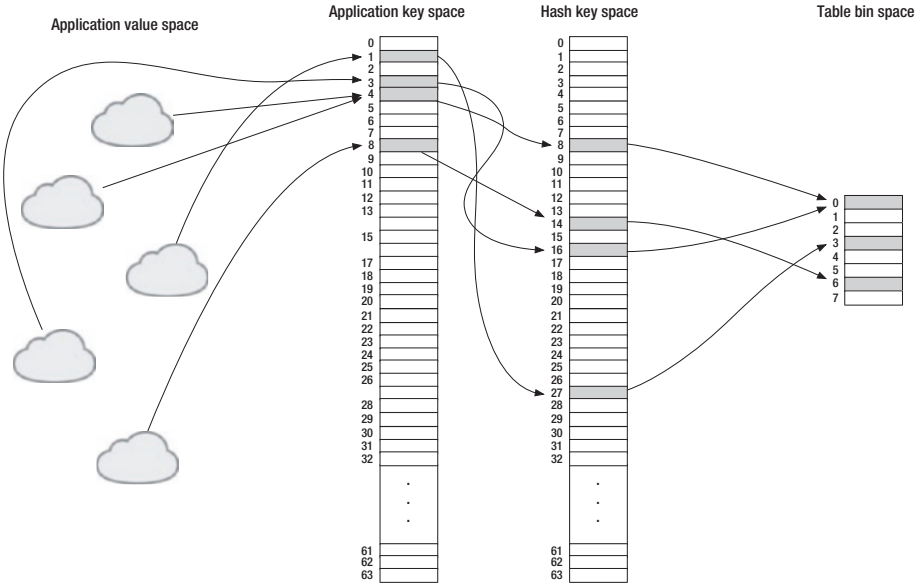

Figure 1-2. *If the application maps values to keys, but they are not uniformly distributed, then a hashing step between the application and the binning can be added*

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.

T. Mailund, *The Joys of Hashing*, https://doi.org/10.1007/978-1-4842-4066-3_2

```
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.

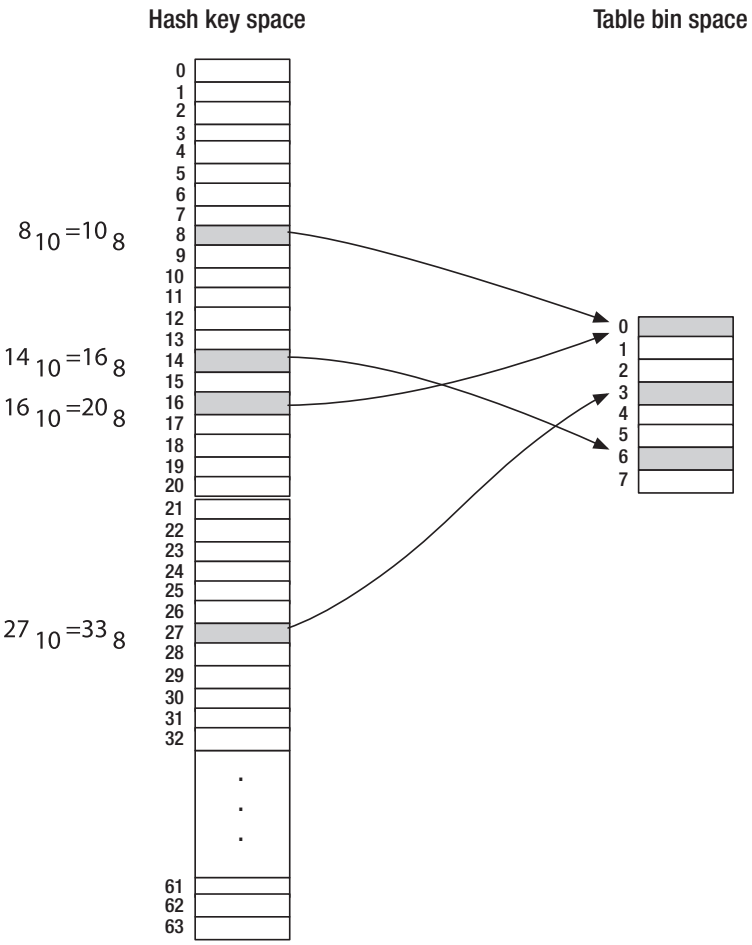

Figure 2-1. *Collisions of hash keys when binning them*

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.

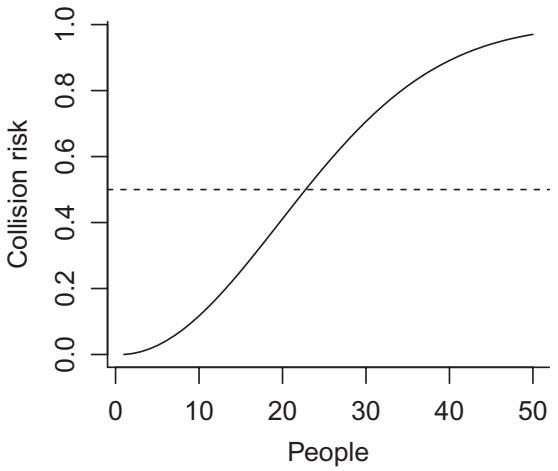

Figure 2-2. *Birthday paradox*

The math carries over to an arbitrary number of "days," m, and tells us what the risk of collision is if we try to insert n elements into a hash table of size m. Provided that the keys are uniformly distributed in the range from 0 to $m - 1$, the probability that there is at least one collision is

$$p(n|m) = 1 - \frac{m!}{m^n (m-n)!}$$

See Figure 2-3 for a few examples of m and n.

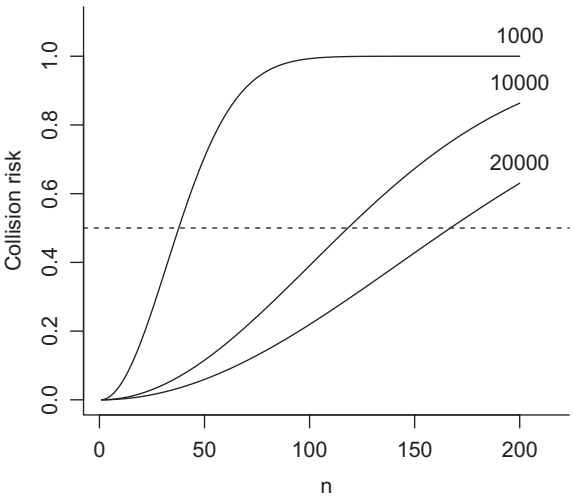

Figure 2-3. *Collision risks for different sizes of tables*

In practice, you are less interested in when the risk of collision reaches any particular probability than you are interested in how many items you can put into a table of size *n* before you get the *first* collision. Let *K* denote the random variable that represents the first time you get a collision when inserting elements into the table. The probability that the first collision is when you add item number *k* is

$$\Pr(K = k|m) = \frac{m!}{m^k (m-k-1)!} \cdot \frac{k-1}{m}$$

where the first term is the probability that there were no collisions in the first $k-1$ insertions and the second term is the probability that the kth element hits one of the $k-1$ slots already occupied. The expected number of inserts you can do until you get the first collision can then be computed as

$$\mathrm{E}[k|m] = \sum_{k=1}^{m+1} k \cdot \Pr(K = k|m)$$

The sum starts at 1 where no collision is possible and ends at $m + 1$ where a collision is guaranteed. Figure 2-4 shows the expected waiting time, together with sampled collision waiting times.

It may not be immediately apparent from Figure 2-3 and Figure 2-4 what the relationship between m and k is for the risk of collision, but it should be evident that it is not linear. In Figure 2-3, increasing m by an order of magnitude when going from 1,000 to 10,000 does not change the k where the risk is above 50% by an order of magnitude; the change is closer to a factor of three. Doubling m when going from 10,000 to 20,000 far from doubles the k where you pass 50%. That the expected number of elements you can insert into a table does not grow linearly with the size of the table is even more apparent from Figure 2-4, but how large should you to have to make a table before you can expect to avoid collisions?

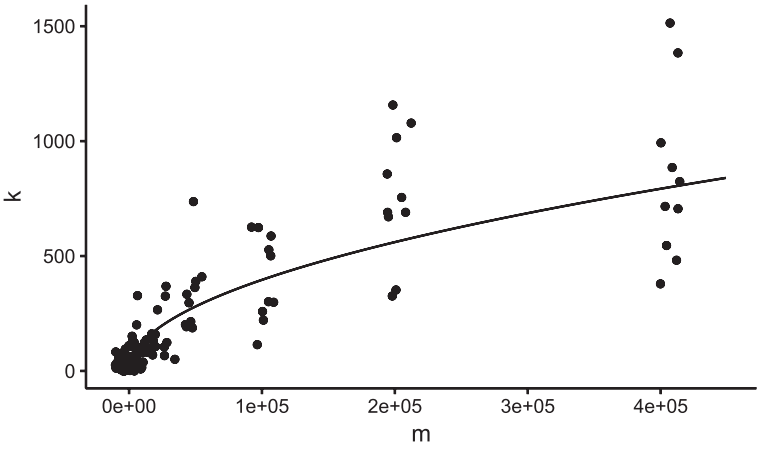

Figure 2-4. *Expected number of insertions before a collision*

An approximation to the collision risk that is reasonably accurate for low probabilities is this:

$$p(k|m) \approx \frac{k^2}{2m}$$

Figure 2-5 shows this approximation as dashed lines.

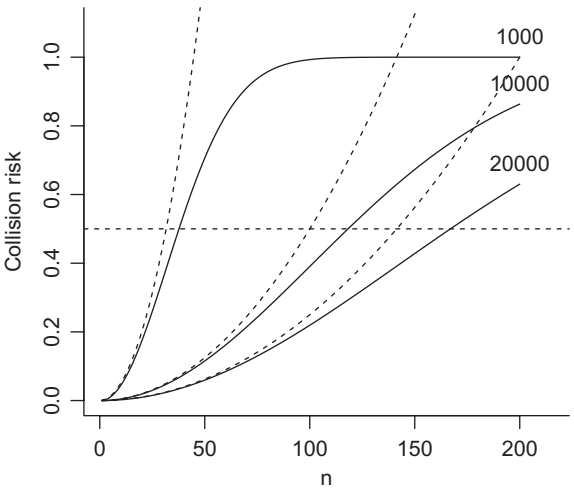

Figure 2-5. *Square approximation*

The approximation is unquestionably very poor at high probabilities (it tends to infinity, which is a bad approximation for a probability), but it is only slightly conservative at low probabilities. The good thing about this approximation is that it is easy to reason about it. You can rewrite it to

$$m \approx \frac{k^2}{2p(k|m)}$$

The formula says that to keep the collision risk low, m has to be proportional to the square of k, with a coefficient that is inversely proportional to how low you want the risk.

This formula is potentially bad news. If you need to initialize the hash table before you can use it,[1] then you automatically have a quadratic time algorithm on your hands. That is a hefty price to pay for constant

[1]It is technically possible to use the array in the table without initializing it, but it requires some trickery that incurs overhead.

time access to the elements you put into the table. Since hash tables are used everywhere, this should tell you that in practice they do not rely on avoiding collisions entirely; they obviously have to deal with them, and most of this book is about how to do so.

Mapping Hash Keys to Bins

Before you move on from the discussion on the size of the tables, however, there is one more topic to address. You are mainly concerned with the relationship between the number of elements you can efficiently put in a hash table, n, and the size of the table, m, but you also need to consider your choice for the value of m.

This is not important in most cases. You want to keep the size of the table linear in the number of elements you put into it. Otherwise, initializing the table will be expensive compared to the number of operations you do on the table. If the reason that you use a hash table is to get constant time operations, and thus a linear time usage for n operations, it would be counter to the purpose to spend more time on initializing them.

In the literature, you will often see suggested that the table size is a prime. There are several reasons for this. One is that if there is some structure to the keys, such as if they tend to be a power of some number more often than not, and that number divides m, then you will only hit a subsection of bins. Such patterns are not uncommon in applications. If you choose a table size that is prime, the keys would have to be powers of that particular prime for this to be a problem.

Sticking to primes has some drawbacks, however. You will often need to resize tables if the number of keys you insert into it is not known ahead of time; you will look at this in Chapter 4. If you want to stick to primes, you need a table of primes to pick from when growing or shrinking your table. If you instead choose table sizes that are powers of two, it is very easy to grow

and shrink them. You can easily combine modulus primes with this idea. If you pick a prime $p > m$, you can index bins as $h(x)$ mod p mod m. Modulus p reduces the problem of regularity in keys and if m is a power of two, you can grow and shrink tables easily and you can mask to get the bins.

If your keys are randomly distributed, you can easily pick table sizes that are powers of two. If so, you can use the lower bits of keys as bin indices and modulus can be replaced by bit masking. If the keys are random, the lower bits will also be random.

You can mask out the lower bits of a key like this:

```
unsigned int mask = table->size - 1;
unsigned int index = key & mask;
```

Subtracting one from table size m will give you the lower k bits, provided m is a power of two, and masking with that, gives you the index.

Masking is a faster operation than modulo. In my experiments, I see about a factor of five in the speed difference. Compilers can optimize modulus to masking if they know that m is a power of two, but if m is a prime (and larger than two), this is of little help. How much of an issue this is depends on your application and choice of hash function. Micro-optimizations will matter very little if you have hash functions that are slow to compute.

There is a trick to avoid modulo, however,[2] and replace it with multiplication. Multiplication is faster than modulus, but not as fast as masking. Still, if you need to have m be a prime, it is still better than modulo.

The first idea is this: you do not need to map keys x to x mod m but you can map them to any bin as long as you do so *fairly*. By fairly I mean that each bin in $[m]$ will contain the same number of keys if you map all of the keys in $[N]$ down to the keys in $[m]$. This is what the expression x mod m does, but you can do it any way that you like.[3]

[2] https://tinyurl.com/yazblf4o

[3] You cannot make a perfectly fair such mapping if N if m does not divide N —x mod m cannot do this either. However, if $N >> m$, then you can get very close.

Typically, N is a power of two, $N = 2^w$ (for example w = 32-bit words). If you want to map w-bit keys, x, to $[m]$, you can use $x \cdot m/2^w$. This is obviously not the same as x mod m but this map is as fair as that. The product $x.m$ will not necessarily fit in w bits, but it will fit into $2w$ bits, so if you use 32-bit words for $[N]$, you can cast them to 64-bit words, do the multiplication, and then divide by 2^w by shifting w bits:

```
((uint64_t) x * (uint64_t) m) >> 32 ;
```

In my experiments, this mapping is only about 50% slower than masking.

Another trick is possible if you use Mersenne primes, which are those on the form $2^s - 1$. One such is $2^{61} - 1$, which can be a good choice for 32-bit words. Let $p = 2^s$ and $x < 2^s - 1$. Let x be on the form $a2^s + b$; that is, let x mod $2^s = b$. Because 2^s mod $p = 1$, x mod $p = a + b$ mod p. Since you use integer division and $b/s^2 < 0$, you also have $x/2^s = a$. Because $x < 2^s - 1$, $x/2^s$ mod $p = x/2^s$ (i.e., it is itself modulo 2^s).

Now, let $y = (x \text{ mod } 2^s) + (x/s^2)$. Again, because $x < 2^s - 1$ you have $a < p$ so $a + b < 2p$. Therefore, either $y \leq p$ or $y \leq 2p$. If the former, that is x mod p, of the latter, then x mod $p = y - p$.

Because x mod 2^x is the same as masking x by p and $x/2^s$ is the same as shifting x by s bits, you can compute modulo as this:

```
uint64_t mod_Mersenne(uint64_t x, uint8_t s)
{
    uint64_t p = (uint64_t)(1 << s) - 1;
    uint64_t y = (x & p) + (x >> s);
    return (y > p) ? y - p : y;
}
```

This avoids multiplications and modulo, and only uses fast bit operations. This will be much faster than modulo.

CHAPTER 3

Collision Resolution, Load Factor, and Performance

Collisions are inevitable when using a hash table, at least if you want the table size, and thus the initialization time for the table, to be linear in the number of keys you put into it. Therefore, you need a way to deal with collisions so you can still insert keys if the bin you map it to is already occupied. There are two classical approaches to collision resolution: chaining (where you use linked lists to store colliding keys) and open addressing (where you find alternative empty slots to store values in when keys collide).

You can download the complete code for this chapter from https://github.com/mailund/JoyChapter3.

Chaining

One of the most straightforward approaches to resolve collisions is to put colliding keys in a data structure that can hold them, and the most straightforward data structure is a linked list.

T. Mailund, *The Joys of Hashing*, https://doi.org/10.1007/978-1-4842-4066-3_3

Linked Lists

The operations you need for storing elements in a list are these:

1. You should be able to add a new key to a list.
2. You should be able to test if a key is already there.
3. You should be able to remove a key from a list.

Also, of course, you should be able to create and delete lists. You can use this data structure:

```
struct linked_list {
    uint32_t key;
    struct linked_list *next;
};
```

This is a single-linked list, which is as simple as lists come. A list consists of links that each contain a key and a pointer to the next element in the list. You will use a null pointer to indicate you are at the end of the list.

You could also use null to represent empty lists, but this has some drawbacks. You would have to make insertion and deletion operations return an updated list; you cannot modify a null pointer, so to insert an element in an empty list you need to return a singleton list. You would also need to return a null pointer when you delete the last link in a list. Doing this is not in itself a problem, but if you update a link somewhere in the middle of a list, returning the updated list is cumbersome.

An alternative approach is to have a sentinel element at the beginning of all lists. You never look at its key, but you know that you never access null lists directly. An empty list is a sentinel whose `next` pointer is null.

```
struct linked_list *new_linked_list() {
    struct linked_list *sentinel =
        (struct linked_list *)malloc(sizeof(struct linked_
        list));
      sentinel->key = 0;
      sentinel->next = 0;
      return sentinel;
}
```

The benefit of using a sentinel is that you can always modify a list in place. It also means that "real" elements in a list always have a predecessor, which simplifies updating lists.

Deleting a list is simple; you only need to delete all the chains in the list:

```
void delete_linked_list(struct linked_list *list) {
    while (list) {
        struct linked_list *next = list->next;
        free(list);
        list = next;
    }
}
```

It is vital that you get hold of the `next` chain in the list before you delete the current one. Other than this, there is nothing complicated in this function.

For the remaining functions, you use an auxiliary function that gives you the link just *before* a link that contains a given key. Because of the sentinel, all links that contain keys will have another link before them, so if a key is in a list, this function will return a link:

```
static struct linked_list *
get_previous_link(struct linked_list *list,
                  uint32_t key) {
```

```
    // because of sentinel list != 0
    while (list->next) {
        if (list->next->key == key)
            return list;
        list = list->next;
    }
    // if we get to list->next == 0, we didn't find the key
    return 0;
}
```

The function returns a link if the key is in the list; otherwise, it returns null. You can exploit this for testing if a key is in a list:

```
bool contains_element(struct linked_list *list, uint32_t key)
{
    return get_previous_link(list, key) != 0;
}
```

You can then use this function to ensure that when you insert a new element, you do not duplicate keys. If you want to use the hash table to represent multi-sets instead of sets, then you should leave that part out. In any case, if you add a new element to the list, you might as well add it to the front of the list. To do this, you must make the new link point to the link the sentinel currently points to and then make the sentinel point to the new link. The function will then look like this:

```
void add_element(struct linked_list *list, uint32_t key)
{
    // Build link and put it at the front of the list.
    // The hash table checks for duplicates if we want to
    // avoid those
```

```
    struct linked_list *link =
    (struct linked_list*)malloc(sizeof(struct linked_list));
    link->key = key;
    link->next = list->next;
    list->next = link;
}
```

Finally, when deleting elements, you see why it is important to get the link *before* the link containing the key. You need to make that link point past the link you want to remove before you delete it. There is nothing complicated in the function, and you can implement it like this:

```
void delete_element(struct linked_list *list, uint32_t key) {
    struct linked_list *link = get_previous_link(list, key);
    if (!link) return; // key isn't in the list

    // we need to get rid of link->next
    struct linked_list *to_delete = link->next;
    link->next = to_delete->next;
    free(to_delete);
}
```

Chained Hashing Collision Resolution

To use linked lists to resolve collisions, you replace the table of keys with a table of lists:

```
struct hash_table {
    struct linked_list **table;
    size_t size;
};
```

Creating and deleting hash tables must be updated accordingly. You need to create a new list for each entry in the table when you construct the table, and you must delete the lists in the table when you delete the table:

```
struct hash_table *empty_table(size_t size)
{
    struct hash_table *table =
      (struct hash_table*)malloc(sizeof(struct hash_table));
    table->table =
      (struct linked_list **)malloc(size * sizeof(struct
      linked_list *));
    for (size_t i = 0; i < size; ++i) {
        table->table[i] = new_linked_list();
    }
    table->size = size;
    return table;
}

void delete_table(struct hash_table *table)
{
    for (size_t i = 0; i < table->size; ++i) {
        delete_linked_list(table->table[i]);
    }
    free(table->table);
    free(table);
}
```

For the other three operations, you map the key to an index into the table as before, and then call the appropriate operation on the linked list at that index:

```
void insert_key(struct hash_table *table, uint32_t key)
{
    uint32_t mask = table->size - 1;
```

```
    uint32_t index = key & mask;
    if (!contains_element(table->table[index], key))
        add_element(table->table[index], key);
}

bool contains_key(struct hash_table *table, uint32_t key)
{
    uint32_t mask = table->size - 1;
    uint32_t index = key & mask;
    return contains_element(table->table[index], key);
}

void delete_key(struct hash_table *table, uint32_t key)
{
    uint32_t mask = table->size - 1;
    uint32_t index = key & mask;
    delete_element(table->table[index], key);
}
```

If you know that your application will never have duplicated keys, you can leave out the check in the insert operation. It is unlikely to matter much for the running time, though, since you aim to keep the lists very short. Because you will keep the lists short, you don't have to worry about the linear search time in each list. If you have an application where you cannot resize your table to keep the number of collisions small, you can replace the linked lists with a data structure aimed at speeding up operations per bin, such as a search tree.

Open Addressing

The open addressing collision resolution does not use an extra data structure but stores keys in the table, as you did with direct addressing (the table you implemented in the previous chapter). If there are collisions,

however, and the desired index is already in use, the trick is to find some other index in which to store the value (somewhere that you can always find again, naturally).

Open addressing requires a strategy for searching for an available index when you want to insert an element. This search is called *probing*. To formalize this, you use a *probing strategy* of $p(k, i)$ that gives you an index that depends on the hash key, k, and an index, i, that goes from zero to m – 1 where m denotes the size of the hash table. When you want to insert k into the table, you first attempt to add it at index $p(k, 0)$. If that slot is occupied, you instead try $p(k, 1)$, and if *that* slot is occupied, you look at $p(k, 2)$, and so on. You want the strategy to probe the entire table eventually. That is, you want the sequence

$$p(k,0),\ p(k,1),\ p(k,2),\ldots,p(k,m-1)$$

to be a permutation of the numbers 0 to m – 1. That way you will eventually find a slot to put the key in, as long as you haven't filled the table.

Probing by iteratively checking if bins in the table are occupied creates a problem with deleting keys. If you remove keys by turning a table entry from occupied to empty, a later search will only get to this point before concluding that there are no more entries to probe. To solve this problem, add another flag to the bin structure you used in direct hashing:

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

The structure for the hash table is the same as when you had no collision resolution:

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

Creating and deleting tables is also the same as before:

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

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

When inserting values, you need to use your probe function to find a free bin. Below, you handle this using the function `p`. Unlike the mathematical function above, the probe function you use takes three arguments: the hash key, the index, and also the size of the hash table, which was implicitly used in the value *m* above.

When inserting an element, you search for the first entry that is either empty, contains a deleted value, or contains the key you want to add (to avoid inserting it twice). You return if you see the key you want to insert. In that case, you do not need to do anything. If you see an empty bin, you break out of the probing loop and insert the key. It is crucial here that the table contains at least *one* empty cell; otherwise, you will overwrite an existing value after you reach the end of the loop. You do not explicitly check for this. You will ensure that you never fill the table later, by resizing it before it fills up. For now, let's ignore this issue.

```
void insert_key(struct hash_table *table, uint32_t key)
{
    uint32_t index;
    struct bin *bin;
    for (size_t i = 0; i < table->size; ++i) {
        index = p(key, i, table->size);
        bin = & table->table[index];
        if (bin->is_free || bin->is_deleted || bin->key == key)
            break;
    }
    bin->is_free = bin->is_deleted = false;
    bin->key = key;
}
```

You also run the probe when checking if a key is already in the table. This time, though, you do not stop the search when you see a deleted value. Searching beyond removed keys to examine bins found after the deleted entry is the whole point of tagging bins containing removed keys.

```
bool contains_key(struct hash_table *table, uint32_t key)
{
    for (size_t i = 0; i < table->size; ++i) {
        uint32_t index = p(key, i, table->size);
        struct bin *bin = & table->table[index];
        if (bin->is_free)
            return false;
        if (!bin->is_deleted && bin->key == key)
            return true;
    }
    return false;
}
```

Finally, for deleting keys, you once again search until you either find the value you are looking for or find a free cell. If you locate the key you want to delete, you update the table entry by tagging the bin as containing a deleted key:

```
void delete_key(struct hash_table *table, uint32_t key)
{
    for (size_t i = 0; i < table->size; ++i) {
        uint32_t index = p(key, i, table->size);
        struct bin *bin = & table->table[index];
        if (bin->is_free) return;
        if (bin->key == key) {
            bin->is_deleted = true;
            return;
        }
    }
}
```

Probing Strategies

Ideally, you want the probing strategy to map each key k to a random permutation of the indices $[m] = 0, 1, \ldots, m - 1$. In practice, this is easier said than done, and you can use simpler strategies. The most straightforward approach is *linear probing*. This strategy is far from providing a random permutation, but it is simple to implement. You search linearly from the index you get from the key to the end of the table and then you wrap around and start from the beginning of the table:

$$p(k,i) = (k+i) \bmod m$$

Assuming you are using table sizes that are powers of two, you can replace modulus with masking and can implement probing like this:

```
static uint32_t
p(uint32_t k, uint32_t i, uint32_t m)
{
    return (k + i) & (m - 1);
}
```

There are two notable drawbacks to linear probing. First, if you have a collision, you not only collide on the first index but the entire probe sequence. This isn't that different from chaining, where you will also need to put colliding keys in the same list, but it is not ideal. You would expect that searching for an available bin would be faster if each key had a different probe sequence. Second, probe collisions tend to cluster. If the linear probe sequence from one index overlaps the probe sequence starting at another index, the two probes will come into conflict. Keys that map to either index will have to probe to the end of the block of occupied bins.

Another strategy, which is closer to the goal of getting a random permutation for each key, is *double hashing*. The idea here is to use two different hash functions to map the key to the initial index and to determine the probe sequence, respectively. The form of the probe is this:

$$p(k,i) = h_1(k) + i \cdot h_2(k) \bmod m$$

For now, let's assume that the keys are already hash keys and thus uniformly distributed, so you would always have h_1 be the identity function. For h_2 you need some value that determines the probe sequence, and you have to make sure that it gives you a permutation of the numbers from zero to m – 1. You get a probe that covers the entire range whenever m and $h_2(k)$ are mutual primes (i.e. their greatest common divisor is 1). Since you are using hash table sizes that are powers of two, any hash function that gives you odd numbers will work, so a simple approach is to turn the key into an odd number by shifting the bits one position and setting the least significant bit to 1:

```
static uint32_t
p(uint32_t k, uint32_t i, uint32_t m)
{
    uint32_t h1 = k;
    uint32_t h2 = (k << 1) | 1;
    return (h1 + i*h2) & (m - 1);
}
```

As a fast but crude evaluation of the two strategies, you can sample the probe lengths in tables where you have inserted random keys. For the experiments in Figure 3-1, I built tables of size $m = 128$ and inserted n elements with $n = 32$, $n = 64$ and $n = 96$. I then sampled 1,000 random keys and measured the probe length for each. For comparison, I also plotted the number of linked list cells examined in the chaining collision resolution strategy. As you can see, the probe lengths do not differ much when n is relatively small compared to m but as n approaches m, the distribution of probe lengths for the linear probe shifts further to the right than the double hashing. The open addressing strategy for both probing approaches generally involves more probes than the chaining approach.

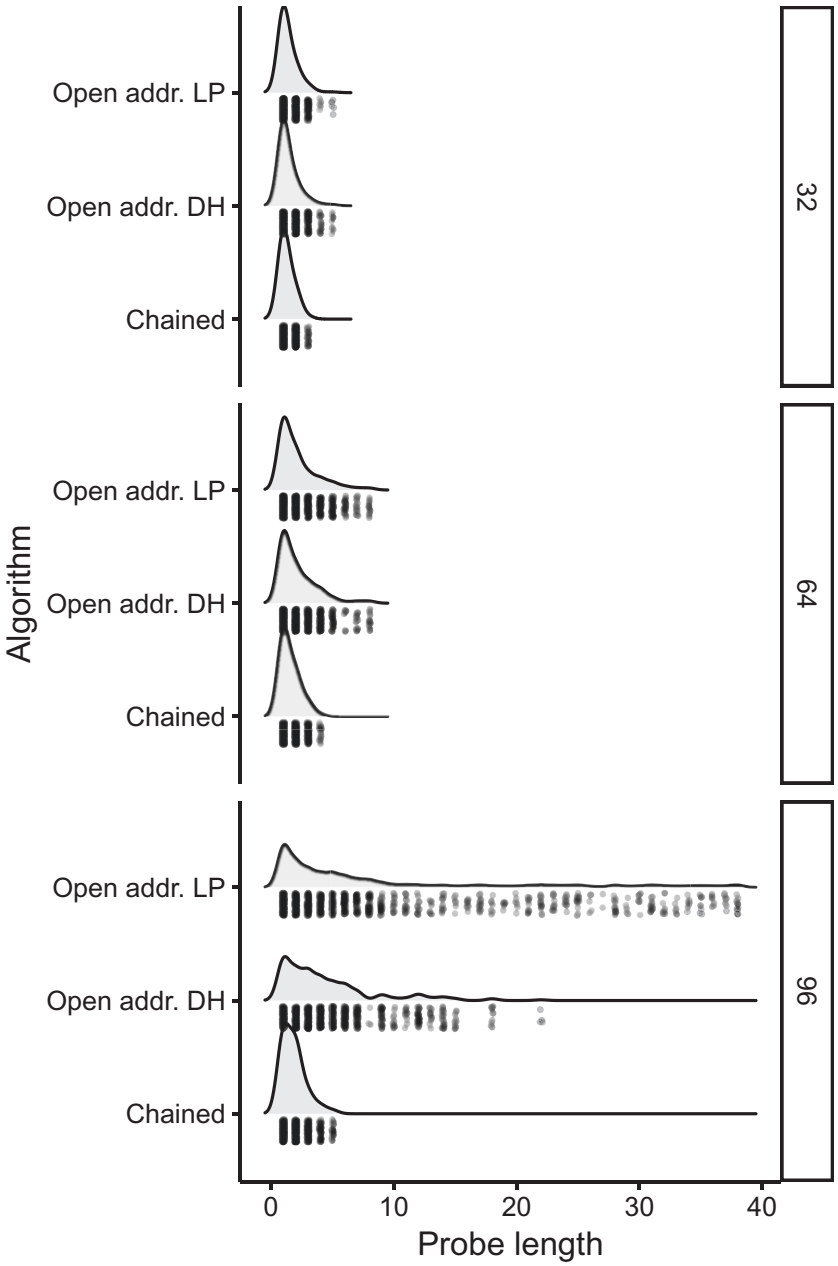

Figure 3-1. *Probe lengths for linear and double hashing probing*

Load and Performance

With conflict resolution strategies you do not need to avoid collisions entirely, but collisions will still incur a performance penalty. If you can avoid collisions altogether, all operations take constant time, but as you start filling the table, the number of collisions will inevitably accumulate, and the running time for each operation will degrade accordingly.

As a measure of how full a table is, you define its *load factor*: Definition: Given a hash table with m slots that store n elements, you define the *load factor* α for the table as n/m.

In this section, you will consider the relationship between the running time and the load factor of a table using one of the two conflict resolution strategies you have implemented. There are many theoretical results for worst-case and average-case performance of these strategies as functions of α, and the proofs can be somewhat involved, so I will not show them here. Instead, I will refer to algorithmic textbooks such as Sedgewick (1998) Chapter 14 and Cormen et al. (2009) Chapter 11 and the references in them.

You will, though, consider the consequences of the theoretical results and then explore performance through experiments.

Theoretical Runtime Performance

The two resolution strategies have different performance penalties as functions of the load factor. You should not be surprised by this, considering that with chained hashing, it is impossible to fill a table to the point where you cannot insert more keys. You can always add new keys to one of the linked lists, regardless of how many keys you have already inserted into the table. With open addressing, you eventually run out of bins to put keys in, at which point probing will either fail or enter infinite loops, depending on the implementation.

Chained Hashing

Chained hashing is the most straightforward strategy. For a chained hashing table, the load factor is also the average number of elements stored per linked list, assuming that keys are uniformly distributed. This follows from the observation that each bin is equally likely to be hit by a key if the keys are perfectly randomly distributed and you map random keys in the key space into random bins in the table. From this observation we get the following:

Property (Cormen et al. 2009 Theorem 11.1): In a hash table in which chaining resolves collisions, both a successful and an unsuccessful search takes time $\Theta(1 + \alpha)$, on average, under the assumption of uniform hashing.

If you are unfamiliar with Θ-notation, $\Theta(f(n))$ means that the running time of an algorithm tends to $c \cdot f(n)$ for some constant c as n tends to infinity. The Θ-notation is part of the terminology and notation known as "big-O" notation, where O-notation is most frequently used. To say an algorithm runs in time $O(f(n))$ means that for some c, $c \cdot f(n)$ is an upper bound for the running time as n tends to infinity. Similarly, you can use $\Omega(f(n))$ to indicate that $c \cdot f(n)$ is a lower bound for the running time of the algorithm as $n \to \infty$. Now, $\Theta(f(n))$ means that the algorithm has both $O(f(n))$ and $\Omega(f(n))$, so that the running time of the algorithm will tend to $c \cdot f(n))$ for some constant c.

When using chaining conflict resolution, you fundamentally rely on linked lists for your table. You use m of them, and so you shave off a factor m in the running time compared to using a single linked list.

Open Addressing Hashing

With open addressing conflict resolution, you cannot reason as directly about conflicts as you can for chaining. Collisions can interfere; the probe starting at one table bin will overlap the probe beginning at another bin. There are theoretical results for the expected running time for table

operations. The proofs are beyond the scope of this book, but these results show the probe length depending on whether a search is successful (i.e., the key you search for is already in the table) or if it is not successful (i.e., the key you seek is not in the table).

Property (Sedgewick 1998 Property 14.3): When collisions are resolved with linear probing, the average number of probes required to search in a hash table of size m that contains $n = \alpha m$ keys is about

$$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$$

and

$$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

for hits and misses, respectively.

Property (Sedgewick 1998 Property 14.4): When collisions are resolved with double hashing, the average number of probes required to search in a hash table of size m that contains $n = \alpha m$ keys is about

$$\frac{1}{\alpha}\log\left(\frac{1}{1-\alpha}\right)$$

and

$$\frac{1}{1-\alpha}$$

for hits and misses, respectively.

Figure 3-2 shows the plotting of these theoretical results.

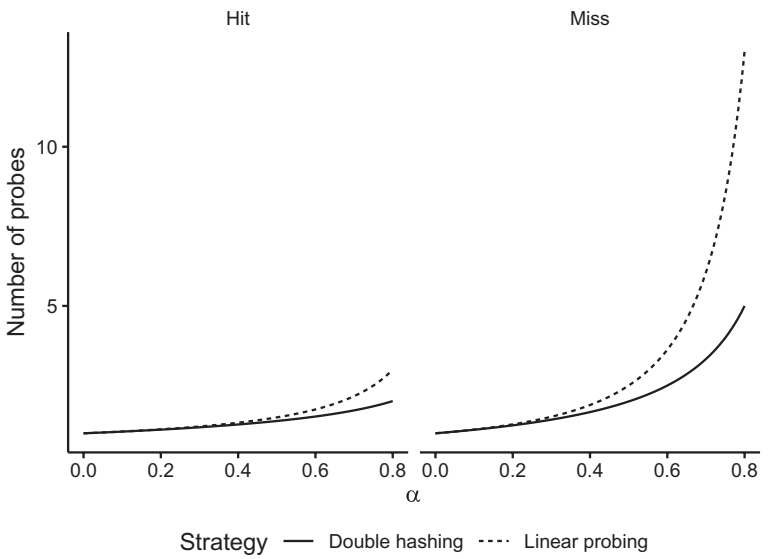

Figure 3-2. *Theoretical probe length as a function of load*

While the running time for chained collision resolution tends to be linear as the load factor grows, and does this even when $\alpha > 1$, the running time for open addressing tends to infinity as α approaches 1. This is only true if you ignore the actual size of the hash table and do not discover infinite loops. If the table has size m, and you avoid infinite loops when probing for an empty slot, you should never have probes longer than m, so in practice, the running time for open address conflict resolution tends toward the size of the table as the load factor tends towards 1. Of course, as α tends to 1, the chained collision resolution doesn't tend towards linear running time in a practical sense either. As $\alpha \rightarrow 1$ the probe length tends towards 1 as well, since, with n keys in a table of size m, you expect the average linked list to have length $\alpha = n/m$, so you expect, on average, to have a probe length of 1. As the load factor tends toward 1, the chained hashing strategy degrades more gracefully than the open addressing strategy, and with the open addressing strategy, the double hashing strategy will give you shorter probes than linear probing.

Using the implementations from the previous section, we can validate the theoretical results experimentally. I constructed tables of size $m = 1024$ and varied n from 32 to 900 (giving α from 0.031 to 0.88) and counted how many probes each method needed when looking up a random key (which, since the space of possible keys is much larger than n, is most likely a miss). Figures 3-3 and 3-4 show the results, with the full range of load factors shown in Figure 3-3 and with load factors smaller than 1/2 shown in Figure 3-4 (in the full range, the results for small load factors are drowned by the long probes at high load factors). The lines are loess-fitted smoothings of the data, roughly showing the mean values along the load axis. The experimental results show the same pattern as you would expect from the theoretical results. The chaining approach has the probe length grow linearly as a function of the load factor, while the open addressing probe length grows super-linearly as it approaches 1, with the linear probe strategy growing faster than the double hashing strategy.

In Figure 3-5, the dots are the mean probe lengths for each load factor and the dashed lines the theoretical expectations.

Probe lengths aren't everything, however. The cost involved with each probe matters as well. For linked lists, there is some overhead involved, although it is relatively minor, and this overhead might make open addressing more appealing as long as you keep $\alpha << 1$. Also, while the double hashing strategy provides shorter probes, this comes at the cost of having to evaluate two hash functions instead of one. On top of this, there is cache efficiency to consider. With chained hashing, you need to allocate list links, and all links in any give list are not necessarily found close together in memory. With double hashing, you jump around in the table of bins, and this is not cache efficient. With linear probing, you search bins close together in memory, which might compensate for the longer probe sequences. The optimal strategy might very well depend on your application and can only be examined by considering actual implementations.

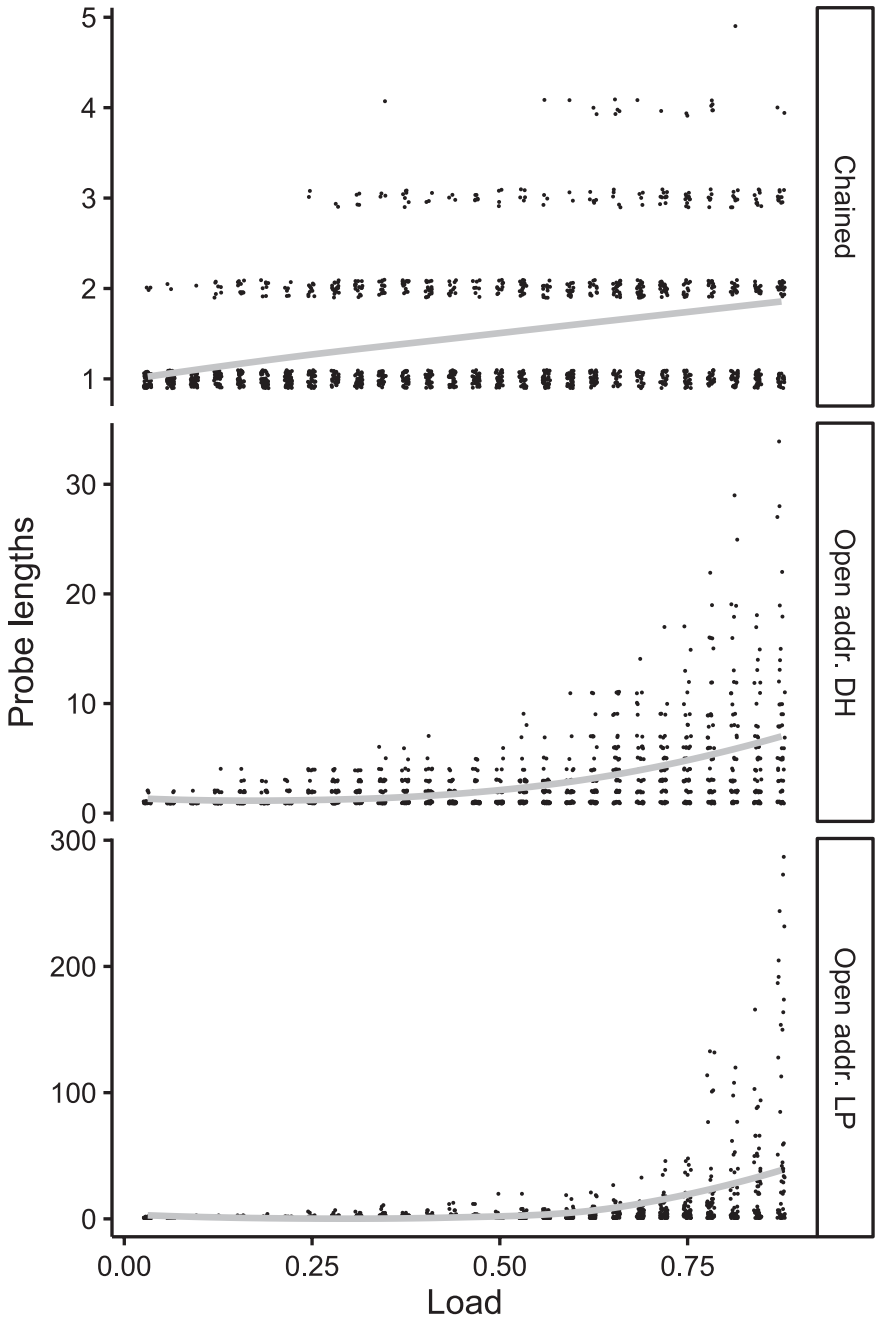

Figure 3-3. *Number of probes for different load factors*

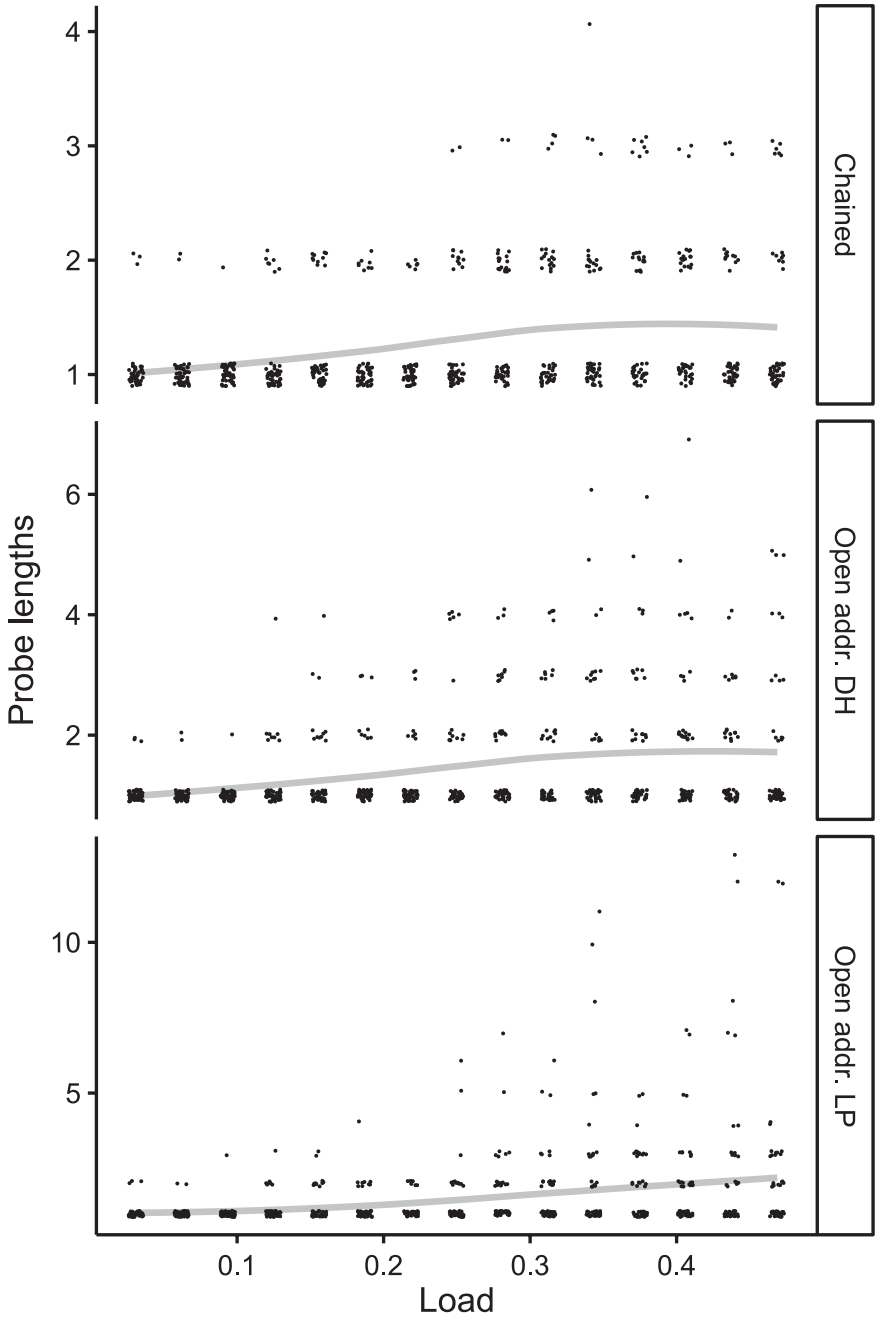

Figure 3-4. *Number of probes for different small load factors*

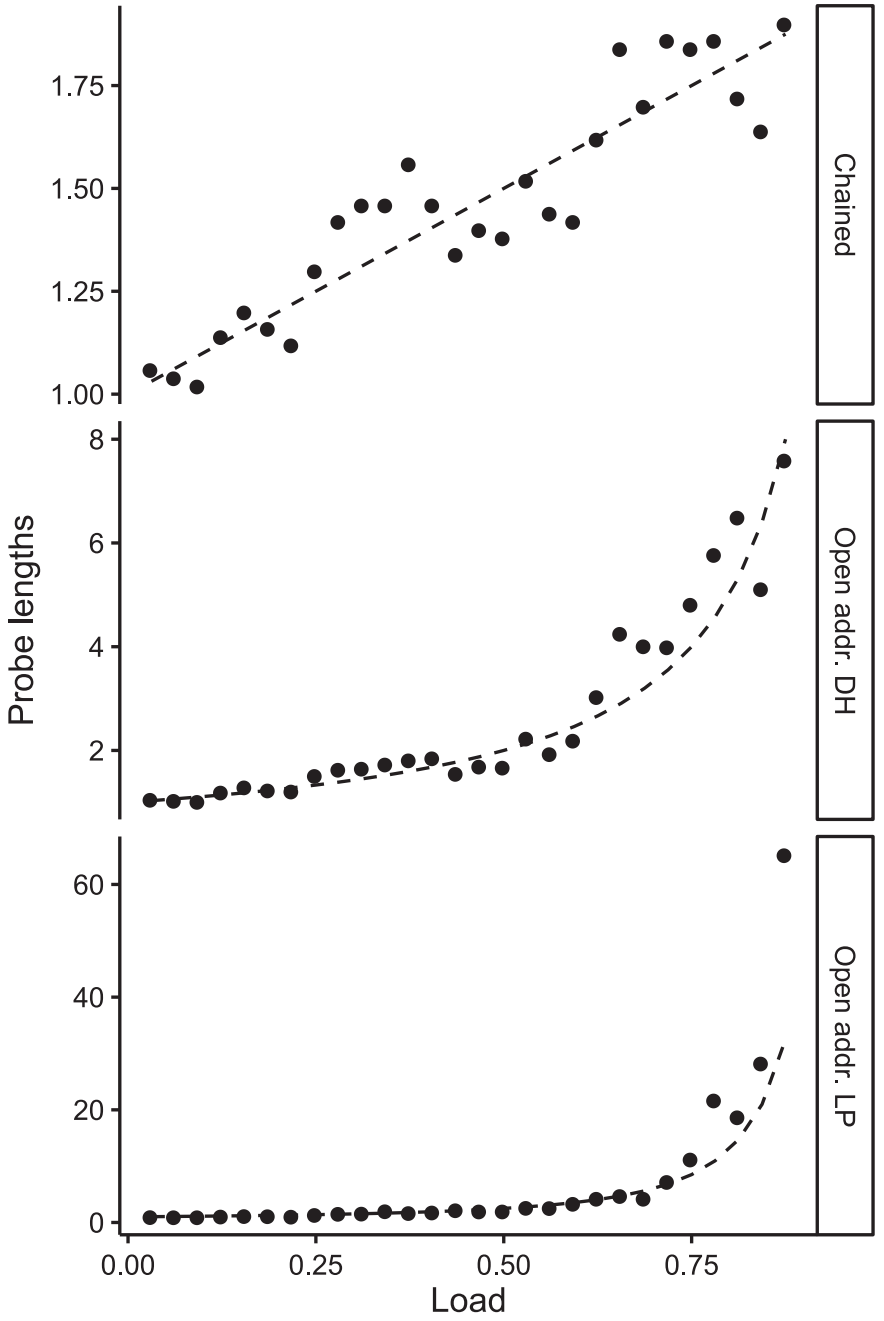

Figure 3-5. *Mean probe lengths vs. theoretical probe lengths*

Experiments

To evaluate the time usage for the three different collision resolution methods, I once again constructed tables of size 1024 (a power of two since we map keys to bins by masking) and then inserted n elements, varying n from 32 to 900 (giving us α from 0.03125 to 16.0). After populating the tables, I performed 1,000 lookups with random keys (which means that we are vastly more likely to have misses in the search than hits, giving us conservative runtime results). Figure 3-6 shows the results for the entire range of load factor. The x-axis is on a log scale, which is why the chained collision resolution strategy does not appear as a line.

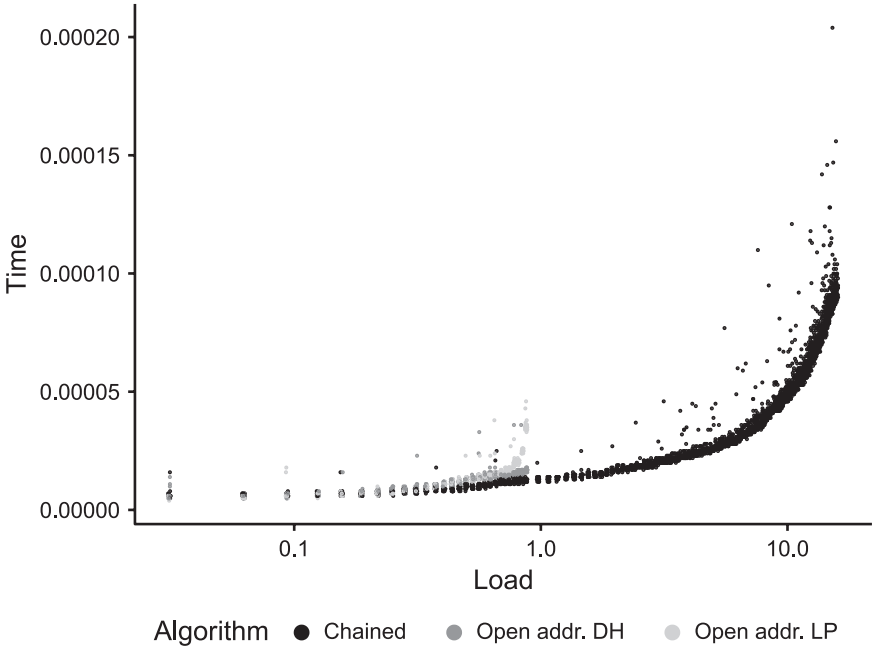

Figure 3-6. *Lookup time usage as a function of load*

Overall, you see the degradation in performance in open addressing collision resolution as the load factor approaches 1 while the chained collision resolution degrades more gracefully. Also observe that the linear probing strategy gets slower than the double hashing strategy as the load factor approaches 1.

You might expect shorter probes with double hashing, but as you observed in the previous section, this comes at the cost of more expensive probe operations. Consider low load factors in Figure 3-7 and Figure 3-8, where the latter displays the same information as the former but is less cluttered since it only shows the mean time for each load factor instead of each replication. These plots focus on small load factors, and you see that at small load factors, $\alpha < 0.2$, the linear probe, with its small computational overhead, is fastest. The double hashing implementation overtakes linear probing around $\alpha = 0.45$, but long before that, at $\alpha = 0.2$, the linked list chaining is fastest (and remains so as α grows).

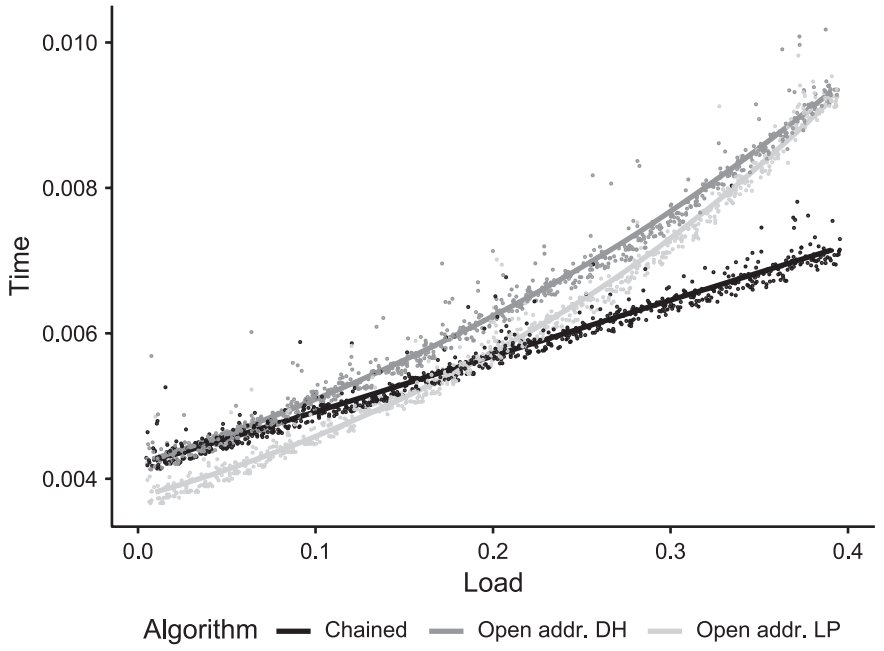

Figure 3-7. *Lookup time usage as a function of load*

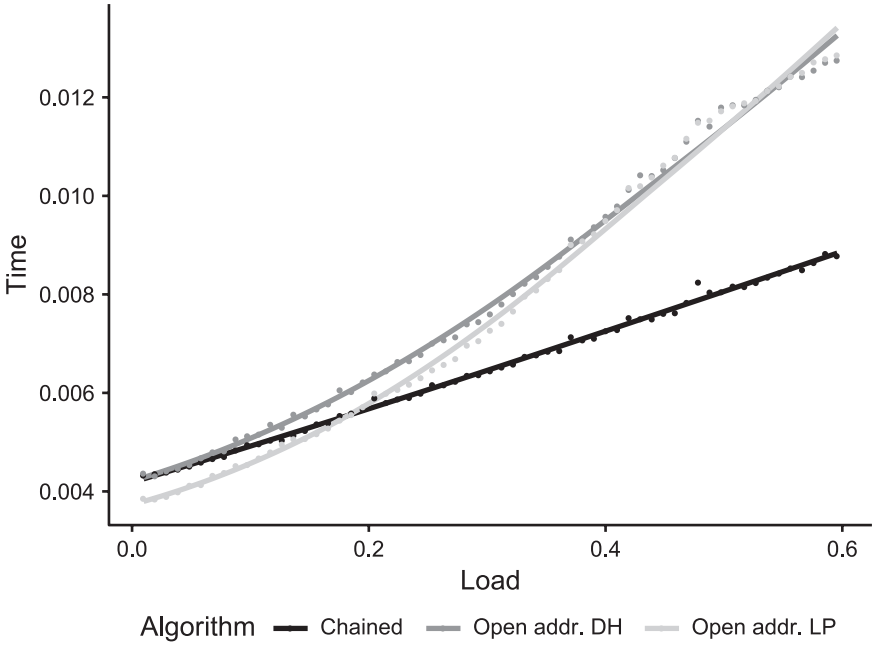

Figure 3-8. *Mean lookup time usage as a function of load*

The exact ranges of load factors at which the different conflict resolution methods dominate in runtime will depend on the implementations, runtime systems, and hardware you run the experiments on. In general, however, expect that at small load factors, linear probing, with the lowest overhead, will be best, and as α approaches 1, the chaining approach will be best (and it will, naturally, be the only approach that works for $\alpha \geq 1$).

However, it is not entirely fair to say that chained hashing is outcompeting the open addressing tables just from these experiments. You should also consider cache efficiency. In the experiments above, the tables are all relatively small, so you do not see a cache effect, but for larger tables, you will. Dynamically allocated links for the lists in chained

hashing are not likely to be optimal for cache usage, not unless you deal with this to explicitly avoid it. Allocating links to minimize cache misses is far from trivial. You will need to allocate pools of memory for the links, and you will want to have separate pools for each bin so searching through the keys in any given bin will involve searching in a list where the links are located close to each other in memory. If you jump around too much in memory as you scan through a list, you will see many cache misses, and the performance will degrade accordingly.

Since the bins in open addressing tables are allocated in contiguous memory locations, caching performance is likely to be better. If all your bins fit into a cache line, open addressing is very efficient. If they do not, linear probing will have fewer cache misses. As you scan linearly through the bins, your probes access nearby memory locations, which is optimal with relation to cache efficiency. With double hashing, you jump around in memory; you would, therefore, expect more cache misses. Although the probe lengths might be longer for linear hashing, the improved cache performance can easily compensate for this. Figure 3-9 shows the runtime for larger table sizes (all with load 0.5). Here you see that open addressing outperforms chained hashing once the tables are large enough where cache efficiency is an issue, even in the load range where chained hashing outperforms open addressing for smaller tables.

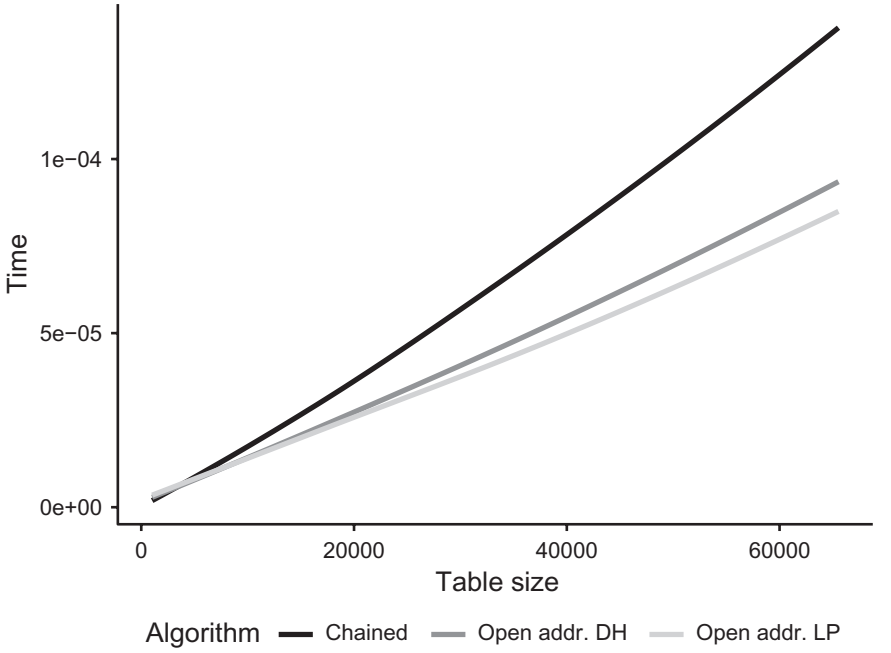

Figure 3-9. *Table size vs. time*

Many considerations will affect the performance of your hash tables, and there are many trade-offs between them. There isn't one best solution because it depends on your application. If the performance of a hash table is critical to your application, it might be worthwhile to experiment with different solutions and engineer your table to be optimal for the specific usage you will subject it to.

CHAPTER 4

Resizing

If you know how performance degrades as the load factor of a hash table increases, you can use this to pick a table size where the expected performance matches your needs, presuming that you know how many keys the table will need to store. If you do not know the number of elements you need to store, n, then you cannot choose a table size, m, that ensures that $\alpha = n/m$ is below a desired upper bound. In most applications, you do *not* know n before you run your program. Therefore, you must adjust m as n increases by resizing the table.

You can download the code from this chapter from `https://github.com/mailund/JoyChapter4`.

Whenever you resize a hash table from size m_{old} to m_{new}, you need to create a new array of length m_{new}. After that, you need to copy all the elements from the old table's bin array into the new table's bin array (where the keys are expected to be more spread out if the new array is larger than the old array). Allocating the new array, and initializing its bins, takes time $O(m_{\text{new}})$ and moving elements from the old array to the new takes time $O(n) = O(\alpha \cdot m_{\text{old}} + m_{\text{old}})$. As long as you keep the load factor bounded by some constant, then resizing a table takes time proportional to the new table size. This runtime cost tells you that you cannot be too aggressive with resizing. If you aim to keep the load factor small to guarantee constant time lookups, you should not pay for this through linear time updates.

T. Mailund, *The Joys of Hashing*, https://doi.org/10.1007/978-1-4842-4066-3_4

Amortizing Resizing Costs

Resizing a table takes time $O(m_{\text{new}})$ (as long as $\alpha \leq 1$), so you cannot guarantee an expected constant running time for all operations if insertion or deletion can trigger resizing. Most operations might take constant time, or be expected to be constant time, as the actual time depends on the length of linked lists or the length of open addressing probes. If an operation requires that you resize the table, however, that operation will not run in constant time. Instead of assuring constant time operations, you can achieve something almost as good: that you can always perform n operations in expected time $O(n)$.

Such a guarantee is known as an *amortized* running time.[1] The way you will amortize the resizing of hash tables is similar to how you implement a "growable array," but the latter is slightly simpler, so let's consider it first.

A growable array is a data structure that you can append to in amortized constant time as well as update and access elements in worst-case constant time. Updating and accessing elements works just as for arrays: you keep values in contiguous memory and can access them through a pointer and an index. Because you use contiguous memory, appending might add an element that doesn't fit in the space you have allocated. When this happens, you need to resize the underlying allocated array.

[1]Strictly speaking, *amortized* means that you write off expensive operations over time, and this suggests that cheaper ones follow costly operations. Doing this would not give you the runtime guarantee you are after. If you stop an algorithm right after an expensive operation and do not follow it with a series of cheap operations, you will be in trouble. What you do with amortized running time is that you save up some "computation" when doing cheap operations such that you can guarantee that you have enough computation in your bank account when you need to pay for an expensive operation.

The interface to a growable array could be this:

```
struct array {
    int *array;
    unsigned int size;
    unsigned int used;
};

struct array *new_array(int initial_size);
void delete_array(struct array *array);

void append (struct array *array, int value);
int get     (struct array *array, int index);
void set    (struct array *array, int index, int value);
```

The implementation is straightforward:

```
struct array *new_array(int initial_size)
{
    struct array *array = (struct array *)malloc(sizeof
    (struct array));
    array->array = (int *)malloc(initial_size * sizeof(int));
    array->used = 0;
    array->size = initial_size;
    return array;
}

void delete_array(struct array *array)
{
    free(array->array);
    free(array);
}
```

```
static void resize(struct array *array, unsigned int new_size)
{
    assert(new_size >= array->used);
    array->array = (int *)realloc(array->array, new_size *
    sizeof(int));
}

void append(struct array *array, int value)
{
    if (array->used == array->size)
        resize(array, 2 * array->size);
    array->array[array->used++] = value;
}
int get(struct array *array, int index)
{
    assert(array->used > index);
    return array->array[index];
}

void set(struct array *array, int index, int value)
{
    assert(array->used > index);
    array->array[index] = value;
}
```

When an append triggers a resize, you double the allocated memory. You use `realloc` to automatically free the old array and automatically copy the old elements into the new when this is necessary. When you resize hash tables later, you need to move elements since you also need to map the keys to new bins. There you cannot use `realloc`, but you can implement resizing by explicitly moving elements.

Growing the size by a constant factor (two when you double the size) is crucial for getting amortized constant time. If you instead chose to increase the length just enough to store the next element, resizing would be prohibitively expensive. Each time you resize the array you need to allocate new memory and move all the existing value to the new array. This takes time proportional to the length of the array. If you start out with an array of size 1 and push n elements onto the array, you will use time $1 + 2 + \ldots + n - 1 + n = n(n-1)/2 = O(n^2)$.

If you double the array size each time you grow, it lets you append m elements in time $O(m)$. In general, increasing the array size by any fixed factor $\beta > 1$ will do this; you will learn more about this later in this chapter. Let's first consider doubling the size. To see if m appends can be done in time $O(m)$, consider the appends between two successive resizing calls. Let the size of the append just after the first size increase be m and the size after the second be $2m$. When you increased the size to m, you did this from $m/2$ and a full array, so the state just after the increase has a half-full array (i.e., the array has length m and contains $m/2$ elements). You need to append an additional m/2 elements to get to the next resizing. The first $m/2 - 1$ of these operations takes constant time. The last takes constant time for appending the last element and then uses time $2m$ for allocating a new array and finally time m for copying all the items to the new array, so in total the $m/2$ operators take time $m/2 - 1 + 1 + 4m/2 + 2m/2 = 7m/2$, which is in $O(m)$.

There is another way to put it: you can make each append cost seven "computations." Of these seven, one is used on the append and six are put in the bank. After the $m/2 - 1$ appends, the bank contains $6(m/2 - 1)$ computations. If you include the seven from the last append, making this operation pay one "computation" immediately and put the remaining six in the bank as the other operations did, you have $6m/2$ left in the bank before you resize. That number of banked operations is what you need to allocate a new array of size $2m$ ($4m/2$) and copy m elements ($2m/2$).

Resizing is illustrated in Figure 4-1. Here, the first (dark grey) block represents the elements that were copied from the previous $m/2$-sized array into the size m array. The next block (light grey) is the $m/2$ long block into which you can insert elements. When you have inserted all of them, you must allocate the $2m$ sized array and move both dark and light grey elements, m elements in total, to the new array.

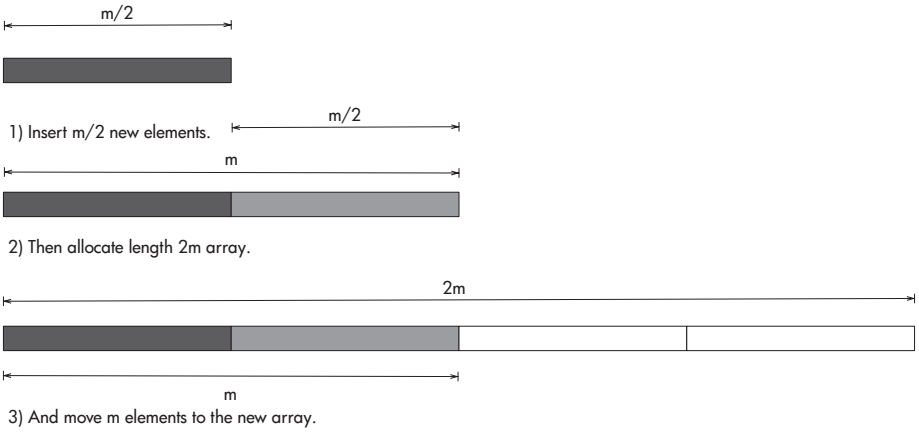

Figure 4-1. *The steps from one enlargement of the array to another*

Figure 4-2 shows the running time when growing an array each time you fill it up. The graph shows the number of operations spent on actually inserting elements, allocating memory for them, and moving them from the old array to the new, and then the cost of all the operations combined in the total running time. The linear upper bound $7m$, which you just derived, is shown as a dashed line.

This analysis assumes that you move from one resize operation to the next as early as you can, by appending $m/2$ times in a row. If you include the other operations, and let them put elements in the bank, you only end up with a larger account before you need to resize.

Growing the array suffices if you only wish to ensure that you can store all the elements you ever need to hold, but it can be a waste of memory if you only hold this maximum number of items early in a program and hold much fewer items after that. As an example of this, you can introduce a "pop" operation that removes the last element and shrinks the array to reduce memory usage. For shrinking, you need to insert coins in the bank when popping to pay for resizing and get amortized running time.

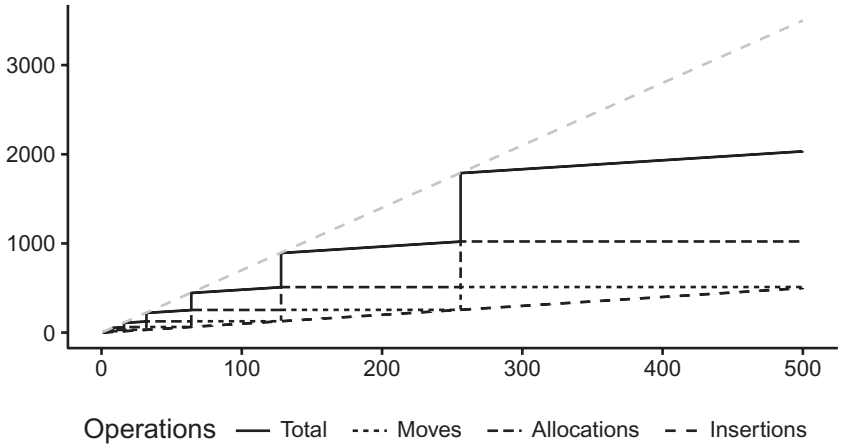

Figure 4-2. *The running time when growing an array by doubling it each time it gets filled*

You halve the length of the array when you pop the array to a sufficiently small size and choose the quarter of the allocated size for "sufficiently low:"

```
int pop(struct array *array)
{
    assert(array->used > 0);
    int top = array->array[--(array->used)];
```

```
    if (array->used < array->size / 4)
        resize(array, array->size / 2);
    return top;
}
```

With this choice, you can pop m times in time $O(m)$. Consider the pops between two resize operations. Let the first resizing be one that leaves the length of the array at m and the second one that reduces the size to $m/2$. Between these two resize operations you must have appended $m/4$ elements. The resize to m would have left the table containing $m/2$ elements (regardless of whether the resize operation grew or shrunk the array), and you do not shrink the array to length $m/2$ before it only contains $m/4$ elements.

If you move directly from the m'th resize event to the $m/2$ resizing, you must have performed $m/4$ pops where the first $m/4 - 1$ involves $m/4 - 1$ constant time pops and the last involves one operation for popping and then $m/2$ (for allocating the new array) plus $m/4$ (for copying elements), so in total $m/4 - 1 + 1 + 2m/4 + m/4 = m$. With the banking analogy, you can charge each pop four: one for the pop and three for the bank. If you do this, you have $3m/4$ in the bank when you need to resize and copy. Resizing costs $2m/4$ and copying costs $m/4$.

Now, let's go to hash tables. The resizing strategy for them will be similar to the array, but you cannot allow the hash table to fill up before you grow it, at least not for the open addressing strategies. The performance degrades dramatically as the load factor approaches 1. You need to resize the tables before α gets too close to 1. Any fixed load factor will do, but as an initial choice, let's grow tables when $\alpha = 1/2$ and shrink them when $\alpha = 1/8$. You expect to see excellent performance for the open addressing strategies if you grow whenever the table is half full and shrink whenever $\alpha = 1/8$. If you shrank tables when $\alpha = 1/4$, then a single deletion right after a table has grown will force you to resize; you grow at $\alpha = 1/2$, which will leave you with $\alpha = 1/4$ after resizing. You resize at $\alpha = 1/8$ because of this.

The runtime analysis for resizing works is analogous to the analysis for arrays. You must bank a bit more for each cheap operation, but otherwise, the analysis is the same. Between growing a table from size m to size $2m$ you need to increase the number of keys stored in the table from $m/4$ to $m/2$ ($m/4$ constant time operations) and then allocate the new table ($2m$) and copy the elements ($m/2$), for a total of $m/4 + 8m/4 + 2m/4 = 11m/4$. So, you can charge each insertion 11. The $m/4$ insertions cost $m/4$ directly and leave $10m/4$ in the bank while allocating the new array costs $2m$, which leaves $m/2$ in the bank. You can use this to pay for the $m/2$ elements you need to copy.

For deletion, consider the operations between resizing to m and shrinking to $m/2$. Here, you need to remove $m/8$ elements (after resizing to size m the table contains $m/4$ elements and shrinking when you reach $m/8$). The resizing costs $m/2$ and the copying costs $m/8$, so during the $m/8$ delete operations you need to save up $5m/8$. If you charge each delete $6m/8$, you can pay for the deletion and save sufficiently many computations for the resizing.

Resizing Chained Hash Tables

The overall pattern for resizing hash tables is the same for the different strategies. You check a `used` variable against the `size` variable after each insert or deletion. If you trigger a resize, you allocate a new array of bins, initialize it, and copy all elements from the old array to the new. The details differ slightly between chained hashing and open addressing, however, so let's consider the two strategies separately. Let's start with chained hashing.

First, make sure to keep track of both the size and the number of keys stored in the table. For this, you need the two variables: `size` and `used`:

```
struct hash_table {
    struct linked_list **table;
    size_t size;
    size_t used;
};
```

You then update insert_key and delete_key to trigger resizing:

```
void insert_key(struct hash_table *table, uint32_t key)
{
    uint32_t mask = table->size - 1;
    uint32_t index = key & mask;

    if (!contains_element(table->table[index], key)) {
        add_element(table->table[index], key);
        table->used++;
    }

    if (table->used > table->size / 2)
        resize(table, table->size * 2);
}

void delete_key(struct hash_table *table, uint32_t key)
{
    uint32_t mask = table->size - 1;
    uint32_t index = key & mask;

    if (contains_element(table->table[index], key)) {
        delete_element(table->table[index], key);
        table->used--;
    }

if (table->used < table->size / 8)
        resize(table, table->size / 2);
}
```

You need to allocate and instantiate a new array for resizing. The code for this matches the constructor code. After that, you need to go through all the keys stored in each of the bin's old array and insert them in the new array. You could handle insertions into the new array while you iterate through the old array's linked lists, but if you update the struct_hash

table so it contains the new array *before* you start moving keys, you can reuse its insert_key function. The resizing code looks like this:

```
static void resize(struct hash_table *table,
                   uint32_t new_size)
{
    // Remember these...
    uint32_t old_size = table->size;
    struct linked_list **old_bins = table->table;

    // Set up the new table
    table->table =
        (struct linked_list **)malloc(new_size * sizeof
        (struct linked_list *));
    for (size_t i = 0; i < new_size; ++i) {
        table->table[i] = new_linked_list();
    }
    table->size = new_size;
    table->used = 0;

    // Copy keys
    for (size_t i = 0; i < old_size; ++i) {
    struct linked_list *list = old_bins[i];
        while ( (list = list->next) ) {
            insert_key(table, list->key);
        }
    }

    // Delete old table
    for (size_t i = 0; i < old_size; ++i) {
        delete_linked_list(old_bins[i]);
    }
    free(old_bins);
}
```

With this implementation, you are freeing the links in the old list and allocating links for the new lists. You do this because calling `insert_key` makes it explicit what you are doing. It is also less efficient than moving the links from the old to the new tables and thus avoids memory allocation and deallocation. If you want to move links instead, you can use this `resize` function:

```
static void resize(struct hash_table *table,
                   size_t new_size)
{
    // remember these...
    uint32_t old_size = table->size;
    struct linked_list **old_bins = table->table;

    // set up the new table
    table->table =
        (struct linked_list **)malloc(new_size * sizeof
        (struct linked_list *));
    for (size_t i = 0; i < new_size; ++i) {
        table->table[i] = new_linked_list();
    }
    table->size = new_size;
    table->used = 0;

    // copy keys
    uint32_t mask = table->size - 1;
    for (size_t i = 0; i < old_size; ++i) {
        struct linked_list *sentinel = old_bins[i];
        struct linked_list *list = sentinel->next;
        while (list) {
            uint32_t index = list->key & mask;
            struct linked_list *next_link = list->next;
```

```
            struct linked_list *bin = table->table[index];
            list->next = bin->next;
            bin->next = list;
            table->used++;
            list = next_link;
        }
    }

    // delete old table
    for (size_t i = 0; i < old_size; ++i) {
        // Free instead of delete -- we have moved the links
        // and we only need to remove the sentinels
        free(old_bins[i]);
    }
    free(old_bins);
}
```

It is much harder to read, but it is slightly more efficient.

Resizing Open Addressing Hash Tables

For open addressing, there is a tiny complication: deleted elements still take up space in the table. The load factor indicates how many keys a table holds, but deleted elements slow down `contains_key` as much as keys that are still in the table. For insertions, you do not have to worry about deleted elements influencing searches (you consider the first bin that contains a deleted element as empty) but for lookups, you have to continue searching past deleted elements. To know when to grow the table to ensure good performance, the `used` counter has to count both the number of keys in the table and the number of deleted elements. This means that you cannot decrease `used` when you delete elements, which is a problem if you want to

shrink table sizes as well as grow them. To do this, you must keep track of two counters: the used counter for the number of keys and deleted cells and another counter named active to count the number of keys in the table:

```
struct hash_table {
    struct bin *table;
    size_t size;
    size_t used;
    size_t active;
};
```

When you insert a key, there are three cases to consider. If the key is already in the table, you leave the counters alone. If you insert into an empty bin, you need to update both used and active. Finally, if the key is inserted into a deleted bin, you only increase the active counter.

```
void insert_key(struct hash_table *table, uint32_t key)
{
    if (contains_key(table, key)) return;

    uint32_t index;
    for (uint32_t i = 0; i < table->size; ++i) {
        index = p(key, i, table->size);
        struct bin *bin = & table->table[index];

        if (bin->is_free) {
            bin->key = key;
            bin->is_free = bin->is_deleted = false;

            // We have one more active element
            // and one more unused cell changes character
            table->active++; table->used++;
            break;
        }
```

```
        if (bin->is_deleted) {
            bin->key = key;
            bin->is_free = bin->is_deleted = false;

            // We have one more active element
            // but we do not use more cells since the
            // deleted cell was already used.
            table->active++;
            break;
        }
    }

    if (table->used > table->size / 2)
        resize(table, table->size * 2);
}
```

When deleting, you must decrease active but not used:

```
void delete_key(struct hash_table *table, uint32_t key)
{
    for (uint32_t i = 0; i < table->size; ++i) {
        uint32_t index = p(key, i, table->size);
        struct bin * bin = & table->table[index];
        if (bin->is_free)
            return;
        if (!bin->is_deleted && bin->key == key) {
            bin->is_deleted = true;
            table->active--;
            break;
        }
    }

    if (table->active < table->size / 8)
        resize(table, table->size / 2);
}
```

The resizing function is relatively simple. You copy all the active keys to the new table but leave the deleted keys alone:

```
static void resize(struct hash_table *table, uint32_t new_size)
{
    // remember the old bins until we have moved them.
    struct bin *old_bins = table->table;
    uint32_t old_size = table->size;
    // Update table so it now contains the new bins (that are
       empty)
    table->table = (struct bin *)malloc(new_size *
    sizeof(struct bin));
    struct bin *end = table->table + new_size;
    for (struct bin *bin = table->table; bin != end; ++bin) {
        bin->is_free = true;
        bin->is_deleted = false;
    }
    table->size = new_size;
    table->active = table->used = 0;

    // the move the values from the old bins to the new, using
    the table's
    // insertion function
    end = old_bins + old_size;
    for (struct bin *bin = old_bins; bin != end; ++bin) {
        if (bin->is_free || bin->is_deleted) continue;
        insert_key(table, bin->key);
    }

    // finally, free memory for old bins
    free(old_bins);
}
```

Theoretical Considerations for Choosing the Load Factor

This example, somewhat arbitrarily, grew or shrank the table when the load factor reached 1/2 or 1/8. In the amortized analysis of the running time, you saw that this gave you a linear running time for doing n insert or delete operations, but you didn't explore how the value of α affects this running time.

Let's consider the general case of growing a table when the load factor reaches some α. The case for shrinking the table is similar: before you grew the table to size m, it had size $m/2$ and contained $\alpha m/2$ elements and $(1 - \alpha)m/2$ empty cells; see Figure 4-3 and Figure 4-4. The next time you grow the table, you will have αm elements, so you must have inserted $\alpha m - \alpha m/2$ elements. The resizing then takes $2m$ operations, and you move αm elements to the new table. In total, you do $m(2\alpha + 2 - \alpha/2)$ operations, and you must pay for it in the $\alpha m - \alpha m/2$ insertion operations.

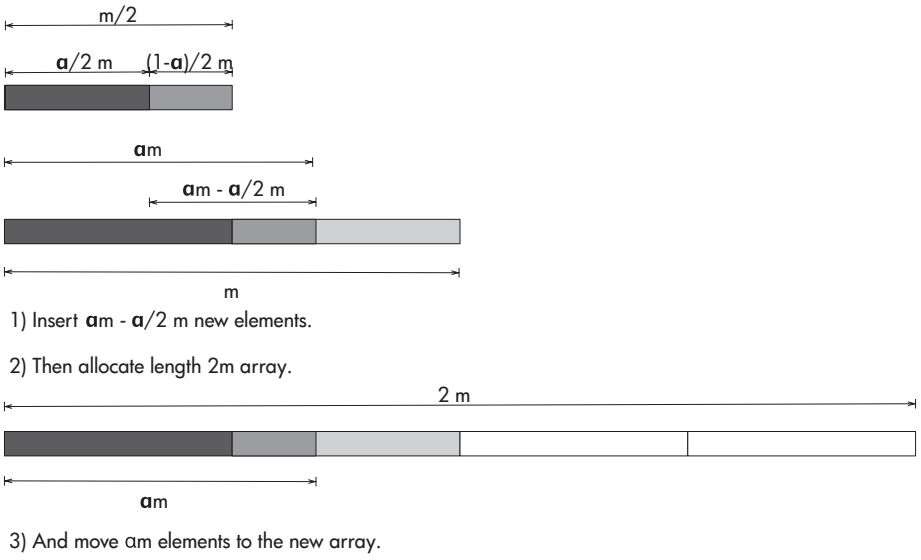

Figure 4-3. *Resizing when you only fill the array up to αn elements before resizing*

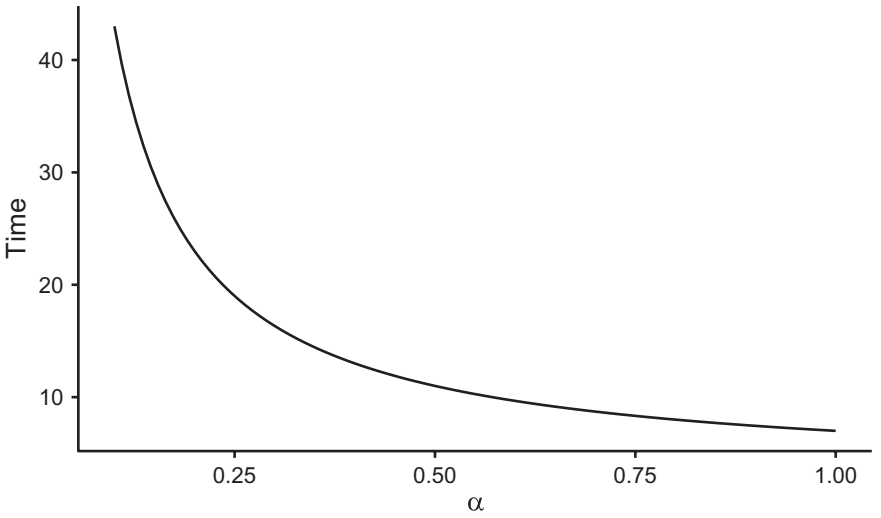

Figure 4-4. *The theoretical running time for growing a hash table, as a function of the load factor threshold, α*

Dividing one by the other, you get

$$\frac{m\left(2\alpha+2-\alpha/2\right)}{m\left(\alpha-\alpha/2\right)}=\frac{2\alpha+2-\alpha/2}{\alpha-\alpha/2}$$

This is the coefficient for the amortized line in the analysis for general α thresholds.

Figure 4-5 plots the theoretical running time for growing a table as a function of α. The figure implies that the higher the load, the better the performance. This shouldn't surprise. The more you fill up the array before you resize, the less relative time you spend on the resizing. The figure is misleading, however. It does not take into account the costs of the probe operations, which you know do also depend on the load factor. If you have an idea of how many successful and unsuccessful searches you expect in a typical run, you can combine the formula from above with the formula for probe lengths from the previous chapter, but it is easier to explore the actual running time via experiments.

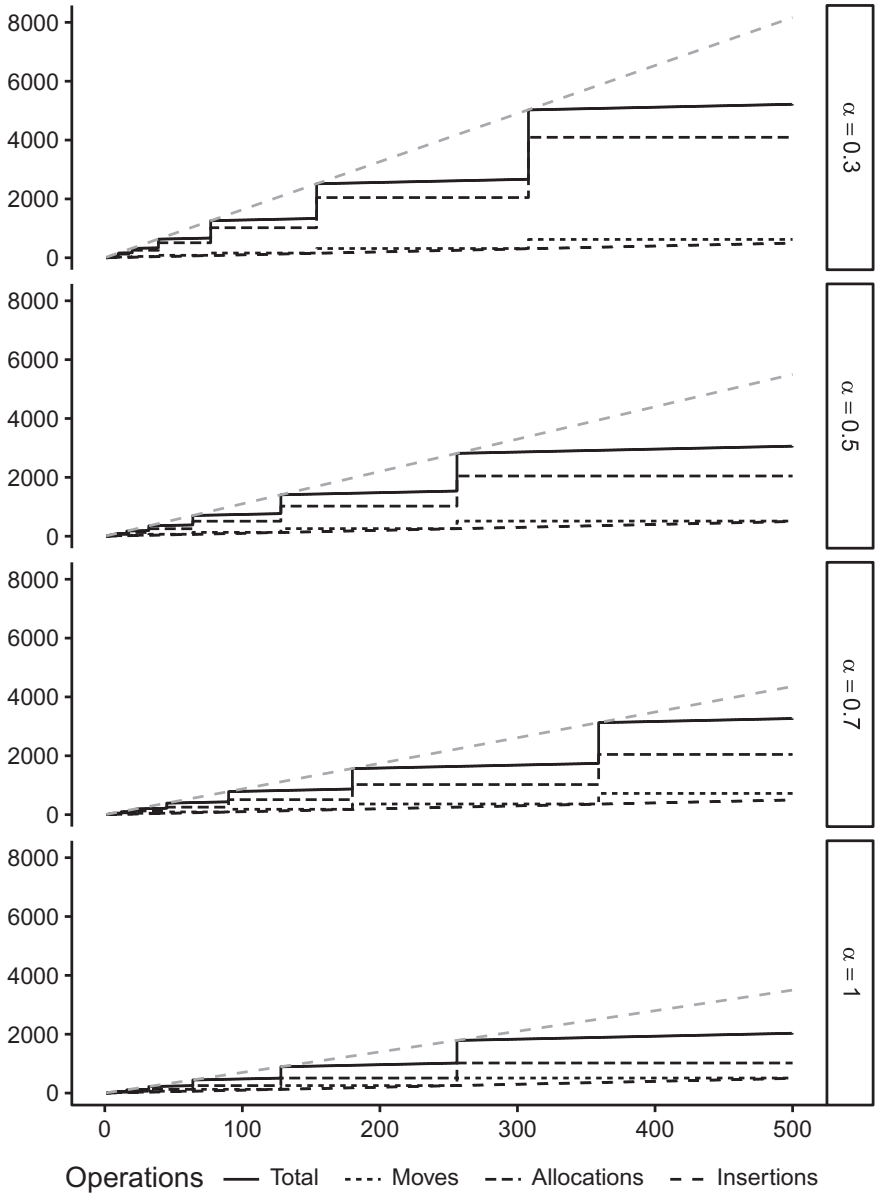

Figure 4-5. *The theoretical running time, split into its different components, for growing a hash table, as a function of the load factor threshold, α*

Experiments

From this theoretical analysis of the performance of the hash tables with resizing, you know that you should be able to insert n elements in time $O(n)$. You should also be able to test if these keys are in the table (they all should be) and do this test in linear time. You should be able to look up n random keys in linear time as well. This is a better measure for the actual performance since the running time guarantees are worse for keys that are not in the table compared to those that are. In either case, each lookup is in $O(1)$ if α is bounded by a constant. Finally, you should be able to delete the n keys stored in the table in time $O(n)$. Let's test this in practice.

Figure 4-6 shows the performance of the three different conflict resolution strategies when you insert, look up, and delete n elements while keeping the load factor $\alpha \leq 1/2$. (Figure 4-7 shows the same experiments but contains only the open addressing strategies, using a different scale on the y-axis to make it easier to see their performance). In these experiments, I initialized all tables with size two. In a real application, you should consider the likely number of keys a table will hold. The table will adjust its size as needed, but if you know how many keys it will hold, you can save some time by initializing it with a capacity around that value.

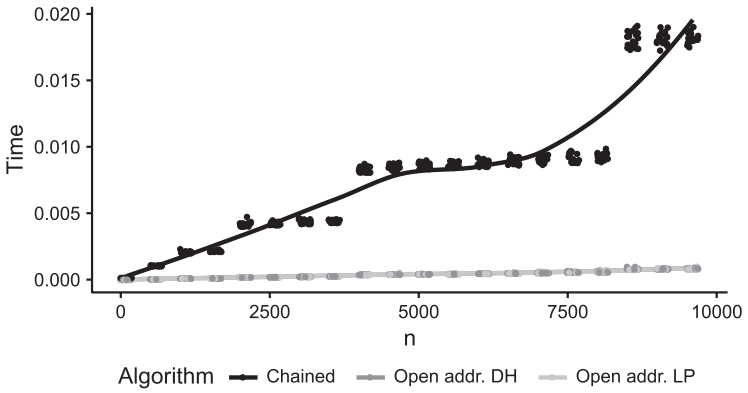

Figure 4-6. *Time usage for inserting n elements, looking them up, and then deleting them again, resizing along the way*

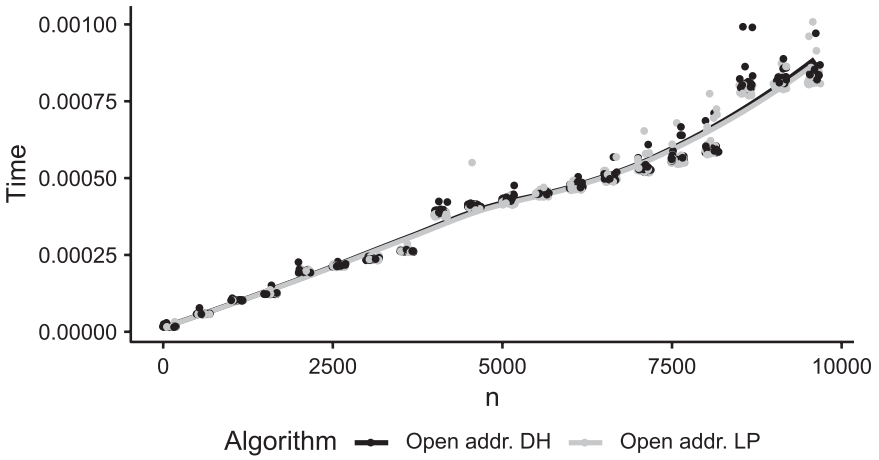

Figure 4-7. *Time usage experiments from Figure 4-6, including only the open addressing strategies*

The time usage looks more stepwise than linear but considering that the amortized analysis only tells you that the time usage should be *bounded* by a line, while you know that resizing operations are expensive and non-resizing inserts and deletions are not, you shouldn't be surprised by this. The stepwise growth of the time usage measurements merely reflects the stepwise function of the smallest powers of two larger than n. Whenever $2^{k-1} < m \leq 2^k$, for some k, you will have to allocate and initialize a table of size 2^k, and this table creation is the most expensive operation in the entire experiment. The steps you see in the experiments are the transitions between different powers of two.

As discussed in the previous section, the choice of the threshold for the load factor α can be any number $0 < \alpha < 1$. In Figure 4-8, you can see the performance of the same experiments as above, with different choices of load factor limits. Figure 4-9 shows the same data (and a few more load factor limits) with smoothed curves, and in Figure 4-10 the data are shown

as functions of the load factor thresholds for a subset of load factors. These experiments require keeping track of the load bound, so you can update the hash table definition to this:

```
struct hash_table {
    struct bin *table;
    size_t size;
    size_t used;
    size_t active;
    double load_limit;
};
```

Next, update the constructor to set the load limit:

```
struct hash_table *empty_table(size_t size, double load_limit)
{
    struct hash_table *table =
        (struct hash_table*)malloc(sizeof(struct hash_table));
    table->table = (struct bin *)malloc(size * sizeof(struct bin));
    struct bin *end = table->table + size;
    for (struct bin *bin = table->table; bin != end; ++bin) {
        bin->is_free = true;
        bin->is_deleted = false;
    }
    table->size = size;
    table->active = table->used = 0;
    table->load_limit = load_limit;
    return table;
}
```

And then update insertion and deletion like this:

```
void insert_key(struct hash_table *table, uint32_t key)
{
    if (contains_key(table, key)) return;

    uint32_t index;
    for (uint32_t i = 0; i < table->size; ++i) {
        index = p(key, i, table->size);
        struct bin *bin = & table->table[index];

        if (bin->is_free) {
            bin->key = key;
            bin->is_free = bin->is_deleted = false;

            // We have one more active element
            // and one more unused cell changes character
            table->active++; table->used++;
            break;
        }

        if (bin->is_deleted) {
            bin->key = key;
            bin->is_free = bin->is_deleted = false;

            // We have one more active element
            // but we do not use more cells since the
            // deleted cell was already used.
            table->active++;
            break;
        }
    }

    if (table->used > table->load_limit * table->size)
        resize(table, table->size * 2);
}
```

```
void delete_key(struct hash_table *table, uint32_t key)
{
    for (uint32_t i = 0; i < table->size; ++i) {
        uint32_t index = p(key, i, table->size);
        struct bin * bin = & table->table[index];
        if (bin->is_free)
            return;
        if (!bin->is_deleted && bin->key == key) {
            bin->is_deleted = true;
            table->active--;
            break;
        }
    }
    if (table->active < table->load_limit / 4 * table->size)
        resize(table, table->size / 2);
}
```

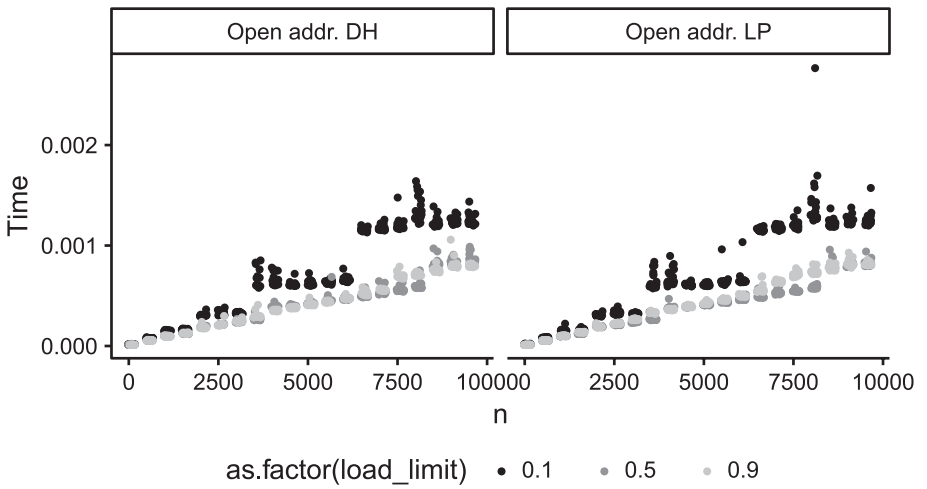

Figure 4-8. *Running time with different thresholds for resizing*

The contains_key function does not change because you add resizing to the implementation:

```
bool contains_key(struct hash_table *table, uint32_t key)
{
    for (uint32_t i = 0; i < table->size; ++i) {
        uint32_t index = p(key, i, table->size);
        struct bin *bin = & table->table[index];
        if (bin->is_free)
            return false;
        if (!bin->is_deleted && bin->key == key)
            return true;
    }
    return false;
}
```

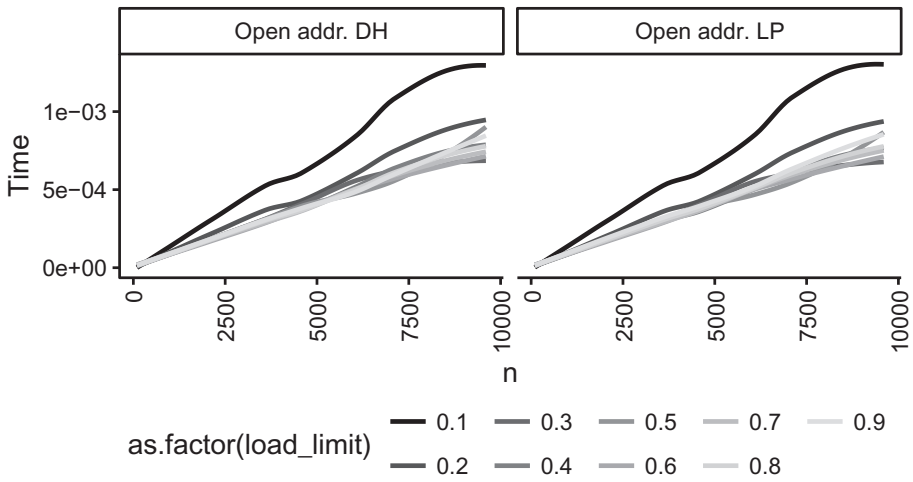

Figure 4-9. *Smoothed running time with different thresholds for resizing*

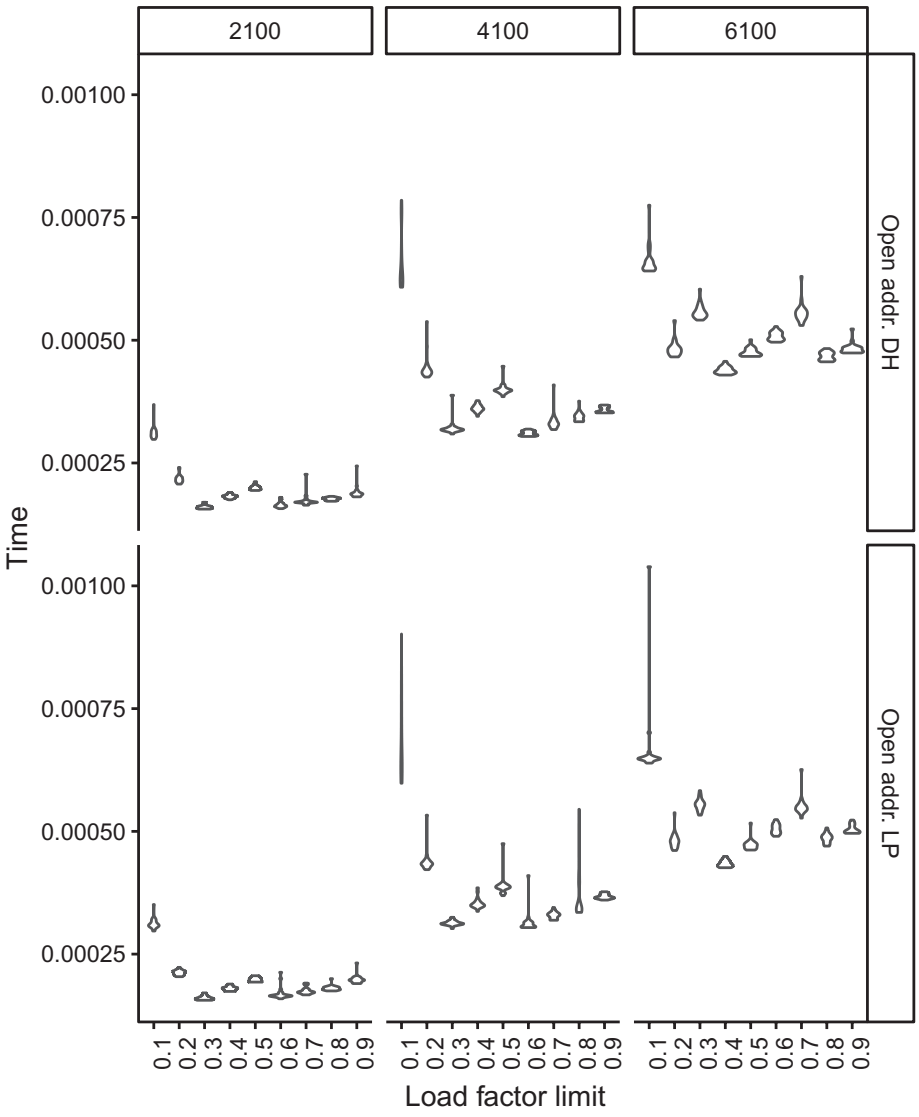

Figure 4-10. *Running time with different thresholds for resizing as functions of the load-factor limits. The different facets show different levels of n.*

You can see that the resize threshold affects the running time substantially. The tradeoff is between the cost of resizing vs. the cost of probing as the load factor increases. If you set this threshold very low, you spend too much time on resizing; if you set it very high, you spend too much time probing.

The optimal choice of load factor threshold depends on your application, the typical sizes of n, and the insertion and deletion patterns. It also depends on your runtime system, which determines the cost of allocating m cells and setting them to 0. As a rule of thumb, though, you are generally best off if you make the threshold at least one-half. Less than that and you always allocate at least twice as much memory as you need—potentially much more, if your threshold is small. The performance does not substantially degrade until you get close to a load factor of 1, so you will get better performance as your threshold approaches 1 than when it approaches 0. You will never do poorly with a threshold around one-half. However, if you have an algorithm that crucially depends on the performance of a hash table, tweaking the threshold is a place to start in your algorithmic engineering.

Resizing When Table Sizes Are Not Powers of Two

When your hash table contains αm or $\alpha m/4$ elements, you grow or shrink the table by a factor of two. As long as you use bit masking to get the bin index for keys, you need the table size to be a factor of two. You can loosen that assumption if you use modulo, and then you can use a prime for m to avoid clustering of occupied bins.

Instead of growing and shrinking the table size in factors of two, you can introduce another parameter, β, and set the table size to $\beta \cdot m$ when growing and m/β when shrinking. Unfortunately, primes are not spread out such that p/β and βp will always be primes when p is, so you cannot

achieve exactly this. The best you can do is pick primes that are close to this and tabulate primes $p_1, p_2, \ldots, p_M$ (for some choice of M)[2] such that $p_{j-1} < p_i/\beta$ and $p_{j+1} > \beta \cdot p_j$.

To handle sizes and a table of primes, you can add a variable, `primes_idx`, to your `struct hash_table`:[3]

```
struct hash_table {
    struct bin *table;
    size_t size;
    size_t used;
    size_t active;
    double load_limit;
    uint32_t primes_idx;
};
```

You can also add a table of primes based on your choice of β. For example, for $\beta = 2$, you can define this table:

```
int primes[] = {
    2, 5, 11, 23, 47, 97, 197, 397,
    797, 1597, 3203, 6421, 12853, 25717, 51437,
    102877, 205759, 411527, 823117, 1646237,
    3292489, 6584983, 13169977 };
```

[2]Technically, you could compute these primes as needed, but this would be much slower than all the other hash table operations, so tabulating the primes you need is the only practical way. You can go to https://primes.utm.edu/lists/ to get a list of the first 1000, 10,000 or 50 million primes and build a table from these by filtering them according to your choice of β.

[3]You do not necessarily need your table size to be prime just because you use modulo and prime to get your bins. You can first get a random key using modulus and then mask out the lower bits. This way, you get a table size that is easier to work with; you can grow it and shrink it by a power of two, but, of course, at the cost of needing two operations to get your bin index. Since getting this index is unlikely to be the most time-critical in using a hash-table, this is a small price to pay.

```
static unsigned int no_primes = sizeof(primes)/sizeof(int);
```

In insert_key you can update the resizing code to this:

```
if (table->used > table->load_limit * table->size) {
    assert(table->primes_idx + 1 < no_primes);
    resize(table, primes[++(table->primes_idx)]);
}
```

In delete_key you can use

```
if (table->active < table->load_limit / 4 * table->size &&
    table->primes_idx > 0) {
    resize(table, primes[--(table->primes_idx)]);
}
```

You can consider the theoretical amortized time analysis when you have a growth factor β added to the story. First, let's ignore α and assume you fill the table before you resize, similar to the growing array. The case is shown in Figure 4-11 and the reasoning is similar to what you have done before to get the amortized running time. Between growing the array to size m and growing it to size βm, you must insert $m(1\text{-}1/\beta)$ elements. These elements must pay for the $m(1\text{-}1/\beta)$ insertions, then the allocation of an array of size βn, and finally for moving m elements to the new array. If you divide the total cost by the number of insertion operations, you get

$$\frac{m(2+\beta-1/\beta)}{m(1-1/\beta)}.$$

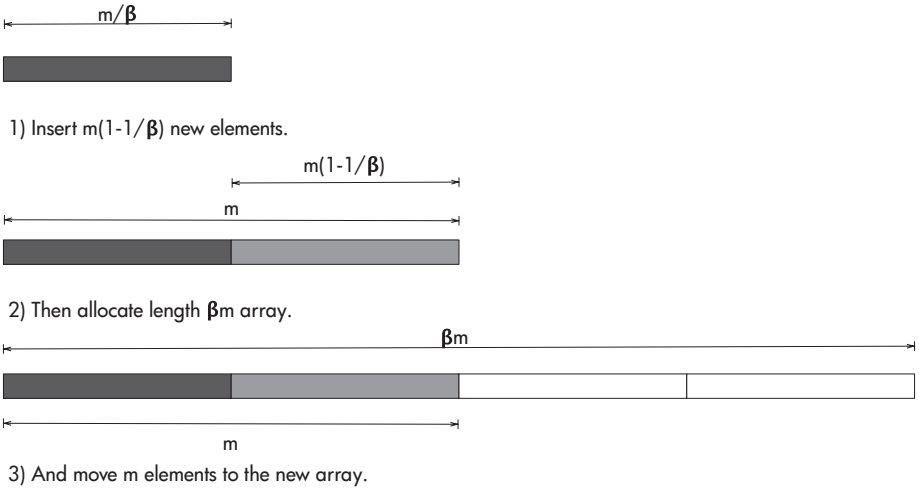

Figure 4-11. *Growing an array by a factor of β*

Figure 4-12 shows this running time as a function of β while Figure 4-13 shows the components of the time usage for different values of β. When β is close to 0, you grow the array by a tiny amount each time you resize, and consequently, you have to reallocate memory frequently, which will give you a runtime penalty. When β grows to infinity, the running time degrades, simply because the cost of a single allocation will grow linearly in β.[4] The expression has a minimum at $\beta = 1 + \sqrt{2}$, shown as the black dot in the figure. Since $\beta = 1 + \sqrt{2} \approx 2.41$, your choice of $\beta = 2$ was not far from optimal, but could be better.

Do not rely too much on this analytical result for the optimal choice of β, however. It assumes that all the operations you perform have precisely the same cost, which is unlikely to be true. The insertion cost depends on the price for updating linked lists or for probing the open addressing table;

[4]The reason we say that n insertion takes (amortized) linear time is that the cost per operation does not depend on n. It does depend on β, however, as you can see from the figure.

the movement cost will depend on this cost as well. The allocation cost depends both on the operating and runtime system. You need experiments to get an accurate measurement of the performance in practice.

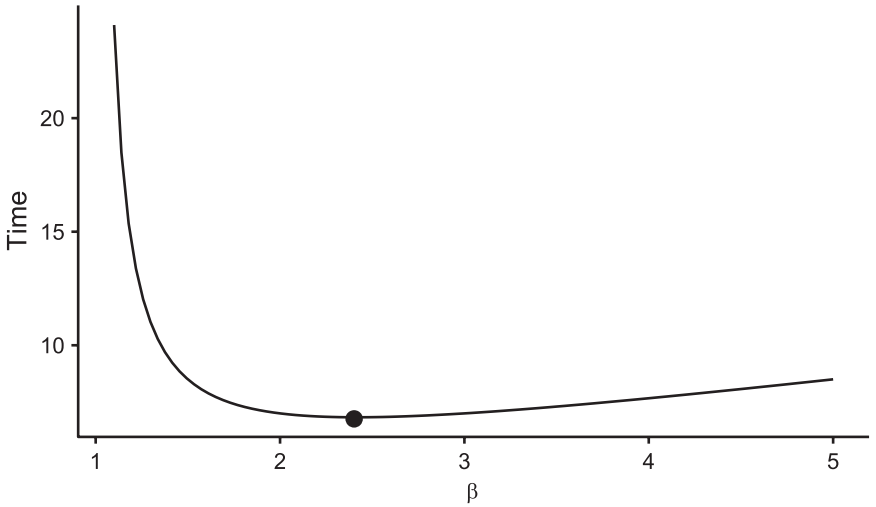

Figure 4-12. *Amortized running time for rescaling as a function of β*

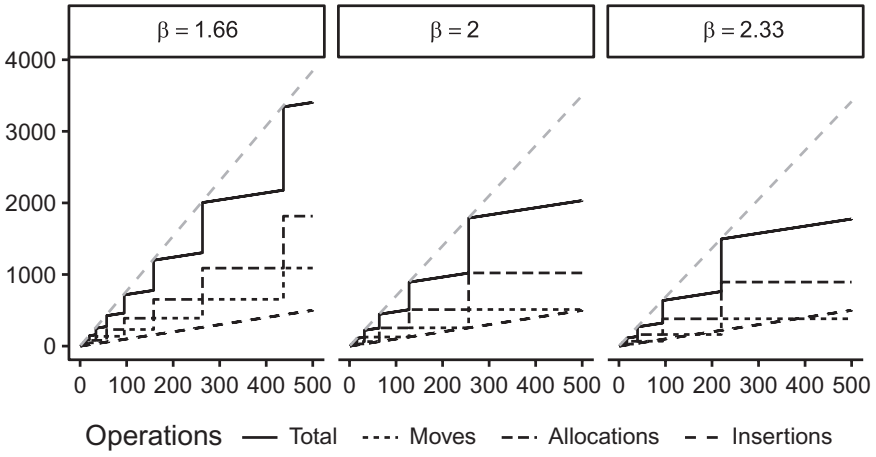

Figure 4-13. *Details of the time usage when growing by a factor β*

If you want your experiments to include open addressing, however, you cannot handle $\alpha = 1$, as you just did when you resized the table when it is full. So you need to add α to your analysis again. The full setup is shown in Figure 4-14 and Figure 4-15. You derive the linear cost per insertion operation as before, just with $\alpha m - \alpha/\beta n$ insertions, allocation to a size βm array, and moving αn elements. The slope for the resulting line is

$$\frac{2\alpha + \beta - \alpha / \beta}{\alpha - \alpha / \beta}$$

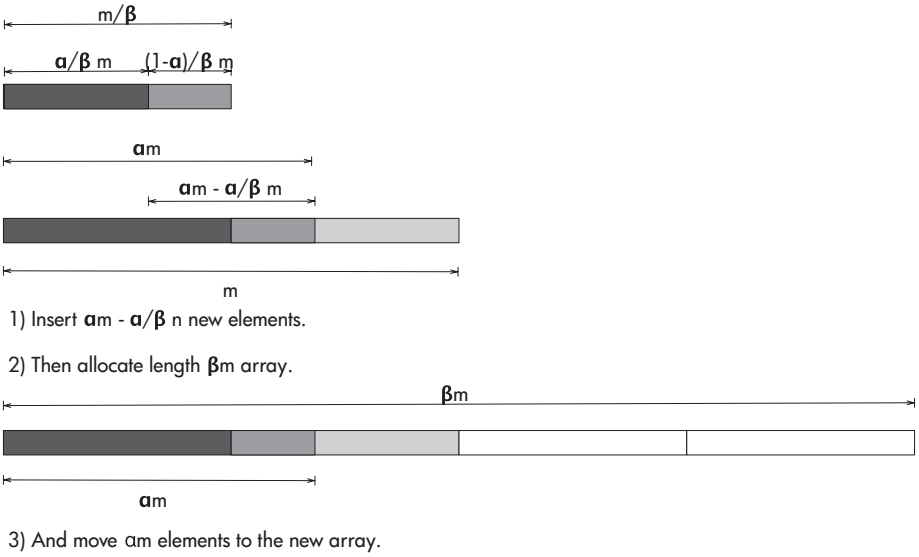

Figure 4-14. *Resizing when both α and β are taken into account*

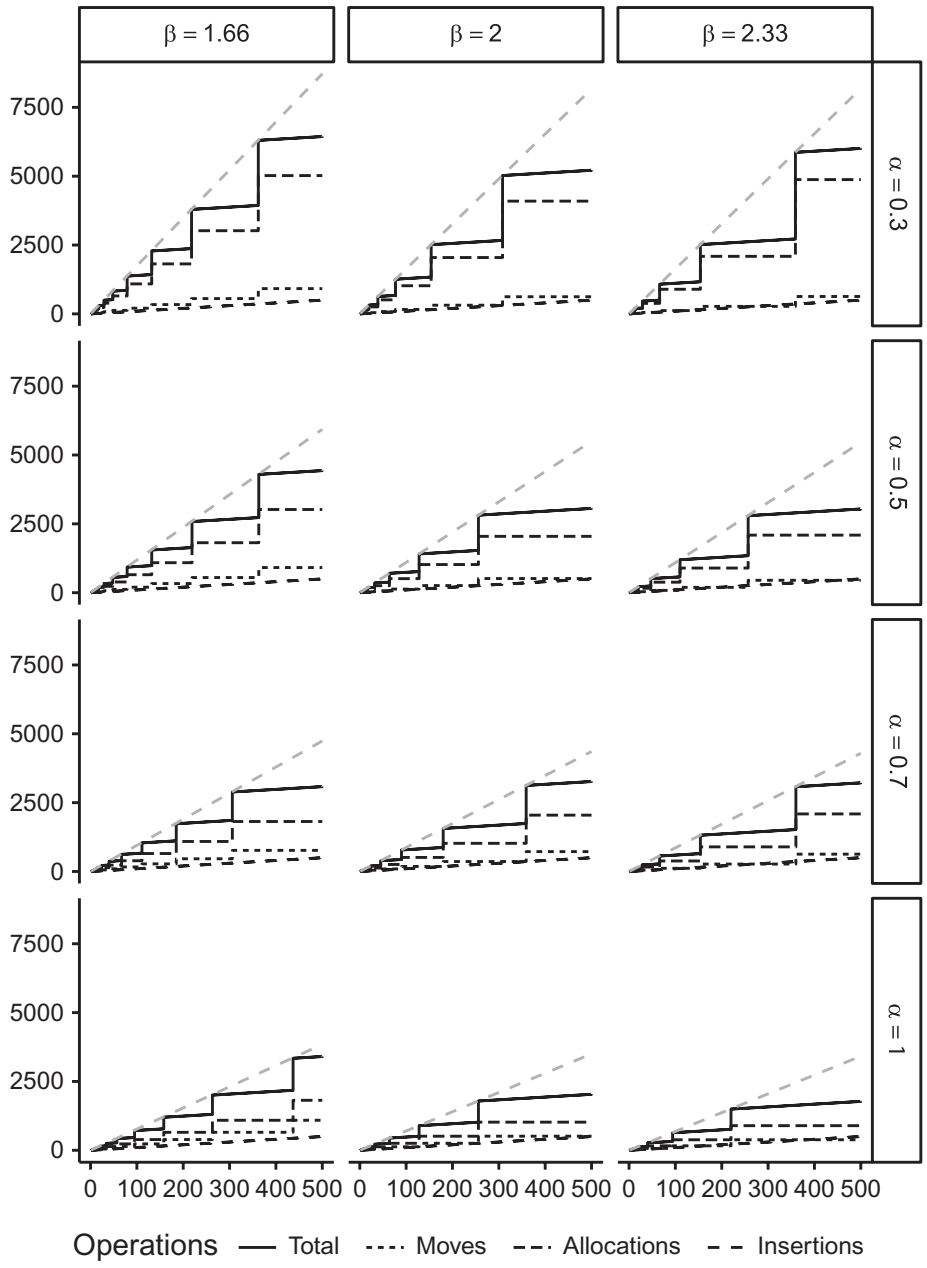

Figure 4-15. *The different components of the running time for a growing table when both α and β are taken into account*

Figure 4-16 plots this amortized operation cost as a function of both α and β. On the left is a range of α values for different choices of β. On the right is a range of β values for different choices of α. As discussed earlier, this formula suggests that you should always make α as large as possible, which you cannot since you need to keep the load factor low. For any given choice of α, however, you have an optimal β at $1-\sqrt{1+\alpha}$. This optimal value is shown as dots on the plot to the right. This optimum, however, requires that all operations take the same time, which they don't, so you must use experiments to see how the actual running time varies for different choices of α and β.

In my experiments, using tables of sizes that are powers of two, and binning based on bit masking, performs better than tables of prime size with modulus (see Figure 4-17) but this can vary. All measurements in Figure 4-17 used linear probing. You saw that linear probing was slightly superior to double hashing for the load-factor thresholds you have used, so you have chosen the fastest solution. You also did this to ensure that m and $h_2(k)$ were mutual primes and thus that the double hashing probe can scan the entire table. To guarantee this is trivial when m is a factor of two but more complicated otherwise.

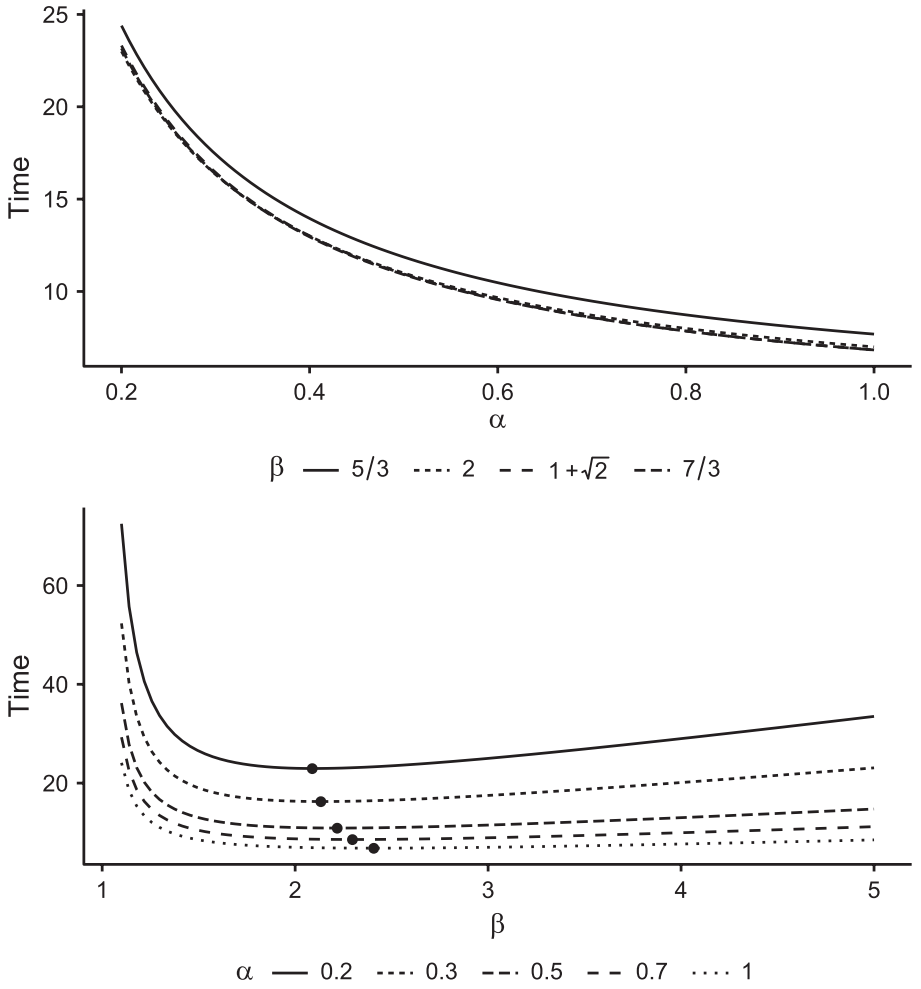

Figure 4-16. *Amortized operation cost when varying α and β*

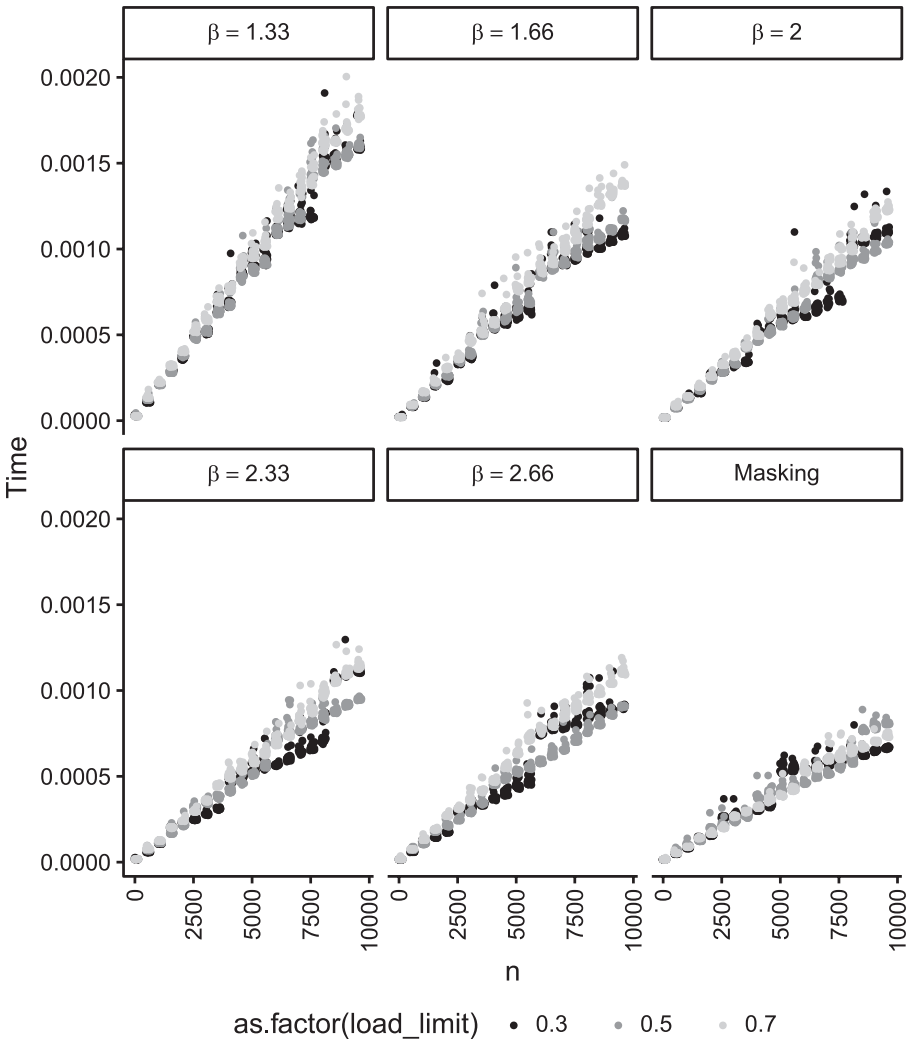

Figure 4-17. *Time performance for tables of prime size with different choices of load factor thresholds, α, and resizing scales, β. Masking denotes the powers-of-two table with masking.*

Dynamic Resizing

Doubling and halving tables when you resize them gives you amortized constant time operations, but the resizing will be slow. This can be remedied by incrementally growing and shrinking a table, one bucket at a time. One approach to this is *linear hashing* (Litwin 1980) (not to be confused with linear probing for open addressing hashing). You will still need the amortization trick to a much smaller degree, but you do not need to initialize tables when you resize.

The underlying idea is this: if you have $m = 2^b$ and you have *two* tables, T_0 and T_1, then you can map $b + 1$ bit keys $x = x_b x_{b-1} \cdots x_0$ to the two tables by using the first bit, x_b, to pick the table and the remaining b bits to pick the index in the chosen table:

$$h(x) = T_{xb}\left[x_{b-1} \cdots x_0\right]$$

Or, in general, if you use $b + k$ bit keys, you can use $x_{k+b-1} \cdots x_b$ to pick one of 2^k tables, and then $x_{b-1} \cdots x_0$ to pick an index into the table:

$$h(x) = T_{x_{k+b-1} \cdots x_b}\left[x_{b-1} \cdots x_0\right]$$

See Figure 4-18.

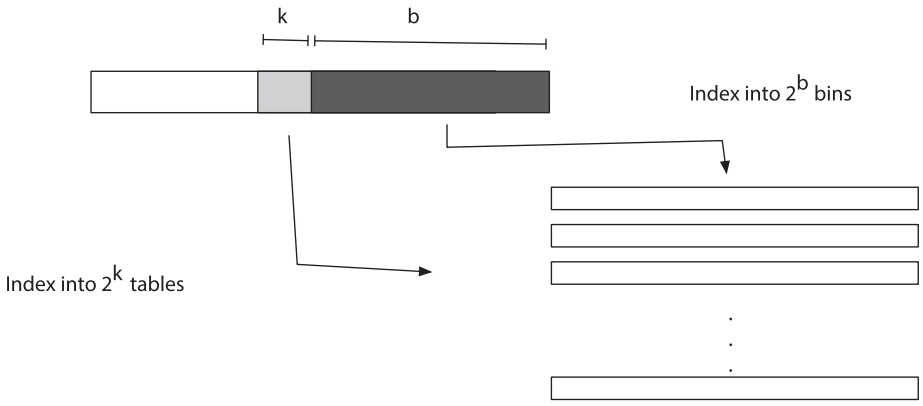

Figure 4-18. *Indexing keys*

This setup alone does not give you dynamic resizing, but it lets you represent a hash table as smaller segments of contiguous memory blocks. You can use 2^k tables of 2^b-sized subtables, where each subtable can be allocated independently of the others.

Keys with the same b least significant bits might sit in different tables, but they will always sit at the same index into those tables. If the next k bits are the same, they will also be in the same table. But imagine if you extend the key by one additional bit, x_k. Then the keys that would otherwise be in the same table at the same index, because they agree on the first k bits, would sit in one of two tables, depending on which bit they have at position k. See Figure 4-19. If the keys are random, about half will sit in the first table and half in the second.

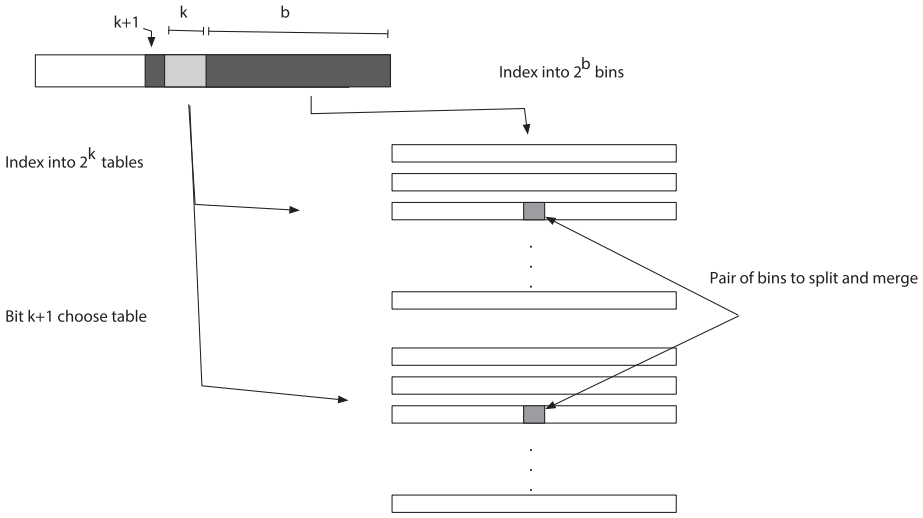

Figure 4-19. *Pairs of bins to split and merge*

If you grow and shrink a table you have allocated this way, you do not need to touch the subtables to double the size of the table. You need to allocate a new "table of tables," and you need to move pointers to the subtables to the new table, but this table is likely to be small compared

to the total table size, at least if you keep subtables reasonably large compared to the number of keys you will put into the table. You can still use amortization to grow and shrink this table, but it will be less work than copying the entire hash table.

You can double the size of a hash table by increasing the number of subtables by a power of two and use one additional bit from the hash keys. Similarly, you can shrink a table to half its size by removing half the subtables and use one bit less from the hash keys.

This is the same amortized behavior as before if you do this. You allocate the memory for the table in a different way and split hash keys into two segments to index into the table, but you still need to rehash all the keys when you change the table's size.

You can, however, exploit that keys that agree on the first $k + b$ bits have to map to one of two bins in a table that is twice as large when you use one additional bit from the hash key. Similarly, you map two bins in the larger table into a single bin in the smaller table when you shrink the table. You do not otherwise shuffle keys around.

The operation that maps one bin in a $k + b$ bit table into two in a $k + b + 1$ bit table is called a *split*, and the operation that maps two bins in a $k+b+1$ bit table into one in the $k + b$ bit table is called a *merge*.

The trick to incrementally grow and shrink is this: you start by allocating two tables of size m, and you put a counter at the first bin in the first table. Each time you insert a key into the table, you also split the bin at the counter, moving half the keys in that bin into the first bin in the second table. Then you increment the counter and do the same for the second bin at the next insertion, and so on. When you look up a key, you need to know if that key should be mapped only to the first table or to both tables. You can do this by checking if the keys are smaller than or larger than the counter. When you have inserted m keys, you have also split m bins, and the active size of the table is $2m$; you have put keys in bins from both tables. You can then double the size of the table by allocating two more tables of size m. Once you have done this, you reset the counter, and now,

when you insert keys, you let it increment to $2m$, splitting the bins in the first two tables to put keys in all four tables.

Similarly, you merge the bin pointed to by the counter and the corresponding bin in the next level of tables when you remove keys. When you have merged sufficiently many bins, you can free the tables you no longer need. As for the resizing strategy from the previous sections, you do not want to shrink a table every time you only use half the tables; if you did, you might end up growing and shrinking the table too often, but you can shrink the table when you use a quarter of the allocated subtables.

Because you split and merge at every insertion and deletion, and use twice as many bins as you have inserted keys, you keep the load under one half.

It is relatively simple to implement this idea with chained hashing because each bin when using k table bits only maps to two other bins when using $k + 1$ bits. The probing complicates initialization with open addressing. You can keep track of how much of a table is initialized by another counter if you use linear probing, but with double hashing, you either need to initialize tables when you allocate them, or you need to use complicated bookkeeping. I will only present chained hashing in the implementation below.

Use this structure to hold the tables:

```
typedef struct linked_list * subtable;
struct hash_table {
    subtable *tables;

    uint32_t table_bits;
    uint32_t k;

    uint32_t subtable_mask;
    uint32_t old_tables_mask;
    uint32_t new_tables_mask;
```

```
    uint32_t allocated_tables;

    uint32_t split_count;
    uint32_t max_bin_initialised;
};
```

This is an array of tables, but you won't follow the exact approach from the previous chained hashing implementation of making the individual tables' arrays of pointers to linked lists. Instead, you embed the sentinel in the tables, not a pointer to it. This is to avoid allocating and freeing sentinels every time you split and merge bins.

The rest of the variables are for bookkeeping. The `table_bits` contains the number of bits set aside for indexing into individual tables and `k` holds the additional number of bits used to identify the tables. The `subtable_mask` is the mask to get out the bits to index into subtables, the `old_tables_mask` is the mask for getting the table index for those keys that should only be mapped to the first half of the subtables, and the `new_tables_mask` is the mask for indexing into all the tables. The `allocated_tables` counter keeps track of how many tables you have allocated; you use this to keep track of when to shrink the table. Finally, `split_count` keeps track of which bin you should split or merge next, and `max_bin_initialised` keeps track of the number of bins you have initialized in all the tables. You only initialize a bin when you split another, so you need to know how many bins you need to free again later.

Most of the work for implementing this idea is bit manipulation. You will use a few helper functions. First, one that masks out a fixed number of bits. You use this function for creating a mask:

```
static inline uint32_t
mask_bits(uint32_t no_bits)
{ return (1 << no_bits) - 1; }
```

Then, given a hash key *x* you can get the table index and the index into a subtable like this:

```
static inline uint32_t
tables_idx(uint32_t x, uint32_t mask, uint32_t bits)
{ return (x & mask) >> bits; }
static inline uint32_t
sub_idx(uint32_t x, uint32_t mask)
{ return x & mask; }
```

Here, `bits` is the number of bits you use to index into a subtable and `mask` is either the mask for the *b* bits that give you the subtable index or the *k* + *b* bits that give you the full bin index (table index plus subtable index).

The number of bits you will use for subtables will be a parameter for the constructor and *k* a counter that you increase and decrease as you resize the table. The `subtable_mask` is fixed once you know how many bits are used for the subtables and the other masks can be set from `table_bits` and k:

```
static inline void
set_table_masks(struct hash_table *table)
{
    uint32_t old_bits = table->table_bits + table->k;
    uint32_t new_bits = old_bits + 1;
    table->old_tables_mask = mask_bits(old_bits);
    table->new_tables_mask = mask_bits(new_bits);
}
```

To get the table index from a key, *x*, you must first check if the key is larger or smaller than the split counter. You cannot use all of *x* to make this test, though. You need to know if *x* is mapped into a bin that you have split or not, and that information is found in the lower *k* + *b* bits of *x*. So, you mask out those bits and compare with the split counter to determine which of the table masks to use.

```
static inline uint32_t
tables_idx_from_table(struct hash_table *table, uint32_t x)
{
    uint32_t old_masked_key = x & table->old_tables_mask;
    uint32_t tmask = (table->split_count <= old_masked_key) ?
                        table->old_tables_mask :
                        table->new_tables_mask;
    return tables_idx(x, tmask, table->table_bits);
}
```

The name of the function doesn't roll off the tongue, but it does what it says it does. It gives you the table index from a key based on the bookkeeping info you have in the table.

You initialize the table like this:

```
struct hash_table *empty_table(size_t table_bits)
{
    struct hash_table *table = malloc(sizeof(struct hash_
    table));
    table->table_bits = table_bits;
    table->subtable_mask = (1 << table_bits) - 1;
    table->k = 0;
    set_table_masks(table);
    table->split_count = 0;

    uint32_t size = 1 << table_bits;
    uint32_t no_tables = 1 << (table->k + 1);

    table->tables = malloc(no_tables * sizeof(subtable));
    for (uint32_t i = 0; i < no_tables; ++i) {
        table->tables[i] = malloc(size * sizeof(struct linked_
        list));
    }
```

```
    // initialise the first table
    for (uint32_t j = 0; j < size; ++j) {
        table->tables[0][j].next = 0;
    }
    table->allocated_tables = 2;
    table->max_bin_initialised = size;

    return table;
}
```

Notice here that you only initialize the first table. The other will be initialized incrementally as you split bins. You initialize bins by setting the `next` pointer of the sentinel to null; you do not allocate the sentinel on the heap.

When you free a table, you need to free the lists in the initialized bins only. The simplest approach is to extract table and subtable indices from a counter that runs up to `max_bin_initialised` and free the lists like this:

```
void delete_table(struct hash_table *table)
{
    uint32_t t_idx = 0;
    uint32_t s_idx = 0;
    uint32_t size = 1 << table->table_bits;
    for (uint32_t i = 0; i < table->max_bin_initialised; ++i) {
        subtable subtab = table->tables[t_idx];
        for (u_int32_t j = 0; j < size; ++j) {
            new_delete_linked_list(subtab[s_idx].next);
        }
        s_idx++;
        if (s_idx == size) {
            free(subtab);
            t_idx++;
            s_idx = 0;
        }
    }
```

```
    free(table->tables);
    free(table);
}
```

The simplest function in the implementation, `contains_key`, only uses the double-indexing to pick a table and a bin:

```
bool contains_key(struct hash_table *table, uint32_t key)
{
    uint32_t t_idx = tables_idx_from_table(table, key);
    uint32_t s_idx = sub_idx(key, table->subtable_mask);
    subtable subtab = table->tables[t_idx];
    return contains_element(&subtab[s_idx], key);
}
```

The inserting and deleting functionality is not more complicated, but you need to call functions for splitting and merging in them:

```
void insert_key(struct hash_table *table, uint32_t key)
{
    uint32_t t_idx = tables_idx_from_table(table, key);
    uint32_t s_idx = sub_idx(key, table->subtable_mask);

    subtable subtab = table->tables[t_idx];
    if (!contains_element(&subtab[s_idx], key)) {
        add_element(&subtab[s_idx], key);
            split_bin(table);
    }
}

void delete_key(struct hash_table *table, uint32_t key)
{
    uint32_t t_idx = tables_idx_from_table(table, key);
    uint32_t s_idx = sub_idx(key, table->subtable_mask);
```

```
    subtable subtab = table->tables[t_idx];
    if (contains_element(&subtab[s_idx], key)) {
        delete_element(&subtab[s_idx], key);
        merge_bin(table);
    }
}
```

When you split a bin, you get the old and new table indices from the `split_count` variable, then run through the bin in the old table and move links to the new based on their keys, masked by the bit at position $b + k$.

When the `split_count` reaches the end of the first half of the allocated tables, which is the same as the old table mask, you grow the table:

```
static inline void split_bin(struct hash_table *table)
{
    uint32_t old_t_idx = tables_idx(
        table->split_count,
        table->old_tables_mask,
        table->table_bits
    );
    uint32_t new_t_idx = (1 << table->k) + old_t_idx;
    uint32_t s_idx = sub_idx(table->split_count, table->
    subtable_mask);

    struct linked_list *old_bin =
        &table->tables[old_t_idx][s_idx];
    struct linked_list *new_bin =
        &table->tables[new_t_idx][s_idx];
    new_bin->next = 0;
    table->max_bin_initialised++;
```

```
uint32_t ti_mask = 1 << (table->k + table->table_bits);

struct linked_list *list = old_bin->next; // save link to old
old_bin->next = 0; // reset old bin

while (list != 0) {
    struct linked_list *next = list->next;
    if (list->key & ti_mask) {
        list->next = new_bin->next;
        new_bin->next = list;
    } else {
        list->next = old_bin->next;
        old_bin->next = list;
    }
    list = next;
}

table->split_count++;
// The old mask is also the maximum index into old tables
if (table->split_count > table->old_tables_mask) {
    grow_tables(table);
    }
}
```

Growing the table involves allocating a new array to hold pointers to the subtables, copying the old tables to the new array, and allocating (but not initializing) the new tables:

```
static void grow_tables(struct hash_table *table)
{
    table->split_count = 0;
    table->k++;
    set_table_masks(table);
```

```
    uint32_t old_no_tables = 1 << table->k;
    uint32_t new_no_tables = 1 << (table->k + 1);
    uint32_t size = 1 << table->table_bits;

    subtable *old_tables = table->tables;
    subtable *new_tables = malloc(new_no_tables *
sizeof(subtable));
    for (uint32_t i = 0; i < old_no_tables; ++i) {
        new_tables[i] = old_tables[i];
    }
    for (uint32_t i = old_no_tables; i < new_no_tables; ++i) {
        new_tables[i] = malloc(size * sizeof(struct linked_
list));
    }
    table->allocated_tables *= 2;

    table->tables = new_tables;
    free(old_tables);
}
```

When you merge bins, you also get the new and the old table index. This time, you move all the keys in the new table to the old.

You need to keep track of the `split_count` again. If it reaches 0, you must move from a k-bit table index to a $k - 1$ bit index. You can extract the number of tables that have active bins from the k counter. This can be less than the number of tables you have allocated, and you use this to recognize when to shrink the table. Strictly speaking, the right time to shrink is when the `split_count` points to the first element in an inactive table, not when it points to the last in another. Since it is easier to recognize the latter, it is what I use. It means you shrink one deletion too late to match the case when you grow the table, but it doesn't change how you resize.

```
static inline void merge_bin(struct hash_table *table)
{
    bool moved_table = false;
    if (table->split_count > 0) {
        table->split_count--; // FIXME: decease anyway...
    } else {
        table->k--;
        set_table_masks(table);
        table->split_count = table->old_tables_mask;
        moved_table = true;
    }

    // we use the number of tables the k indicate
    // to compare with the totally allocated tables.
    uint32_t total_no_tables = 1 << (table->k + 1);

    uint32_t old_t_idx = tables_idx(
        table->split_count,
        table->old_tables_mask,
        table->table_bits
    );

    uint32_t new_t_idx = (1 << table->k) + old_t_idx;
    uint32_t s_idx = sub_idx(table->split_count, table->
    subtable_mask);

    struct linked_list *old_bin =
        &table->tables[old_t_idx][s_idx];
    struct linked_list *new_bin =
        &table->tables[new_t_idx][s_idx];
    struct linked_list *list = new_bin->next;
```

```
    while (list) {
        struct linked_list *next = list->next;
        list->next = old_bin->next;
        old_bin->next = list;
        list = next;
    }
    new_bin->next = 0;

    if (total_no_tables <= 2) {
        // Do not merge below the smallest table
        return;
    }
    if (moved_table && total_no_tables <= table->allocated_
    tables / 4) {
        // I'm moving one index after I should, but this
        // point is slightly easier to recognize than the
        // one before.
        shrink_tables(table);
    }
}
```

Finally, shrinking the table follows the pattern of growing it. You allocate a new array of subtable pointers and move the old tables. When you shrink the table, there are no keys in the tables you remove since you have merged them into the other tables. Since the sentinels are embedded in the subtables, you do not need to free them. You only need to free the subtables.

```
static void shrink_tables(struct hash_table *table)
{
    uint32_t total_no_tables = 1 << (table->k + 1);
    uint32_t new_no_tables = 2 * total_no_tables;
    uint32_t size = 1 << table->table_bits;
```

```
    subtable *old_tables = table->tables;
    subtable *new_tables = malloc(new_no_tables *
    sizeof(subtable));
    for (uint32_t i = 0; i < new_no_tables; ++i) {
        new_tables[i] = old_tables[i];
    }

    for (uint32_t i = new_no_tables; i < 2*new_no_tables; ++i)
{
        subtable subtab = table->tables[i];
        free(subtab);
    }
    free(old_tables);
    table->tables = new_tables;
    table->allocated_tables = new_no_tables;

    // When we shrink, all lists in the old tables
    // must have been initialised
    table->max_bin_initialised = size * new_no_tables;
}
```

Incrementally growing and shrinking tables reduces the time it takes to resize, but all operations get more complicated and thus slower. Unless you want to reduce the time each operation takes, then using more time on resizing and less on all the others is preferable. Over a series of operations, the latter will be faster.

CHAPTER 5

Adding Application Keys and Values

So far, you have only considered storing keys in your hash tables. Most of the techniques for implementing hash tables do not depend on whether you store simple keys or whether you associate application values with them. The setup where you only store keys that you can also use as hash keys, however, is practically never used in real-world applications. This chapter is about storing application values in bins together with their hash keys. You can download the code at https://github.com/mailund/JoyChapter5.

You generally use hash tables for two things: to implement a *set* data structure or to implement a *map* data structure; the setup where you only store hash keys implements neither. What you have seen so far is implementations of sets of hash keys, but remember that you compute hash keys from some other data. Hash keys are the result from applying a hash function to your application keys, and they represent a simplification of your original data.

Implementing a set while storing hash keys alone does not guarantee that membership tests will work. They are likely to work since you expect different keys to map to the same key with small probability, but you cannot rule out collisions. You will need to compare application keys when you search for membership or when you delete keys; only comparing hash keys will not suffice. Consider Figure 5-1. Here, you can see collisions at

T. Mailund, *The Joys of Hashing*, https://doi.org/10.1007/978-1-4842-4066-3_5

two levels. Different application keys can map to the same hash key and different hash keys map to the same table bin. You have solved the second problem; to solve the first, you need to store the actual keys in the table.

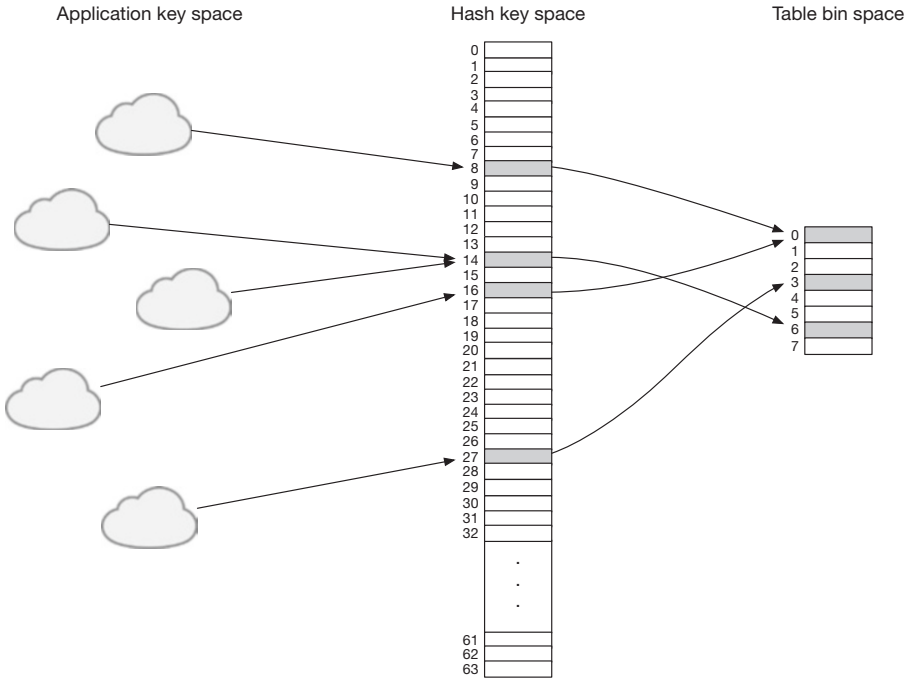

Figure 5-1. *Hash keys and application keys*

If you store the application keys, do you still need to store the hash keys? You do, for two reasons. First, you need the hash keys when you resize the tables. If you didn't store the hash keys, you would need to recompute them every time you resize a table, and recomputing hash keys is a potentially slow operation compared to moving keys and values between arrays or lists. Second, comparing hash keys involves comparing two computer words, a fast operation. Comparing application keys, on the other hand, might not be fast; it would depend on their complexity.

Hash Sets

To add application keys to a hash table, you must modify the interface a bit. The actual `struct` depends on the collision resolution strategy, but as an abstract data structure, you will use this interface:

```
typedef void (*destructor_func)(void *);
typedef bool (*compare_func)(void *, void *);

struct hash_table *
empty_table(uint32_t size,
            compare_func cmp,
            destructor_func destructor);
void delete_table(struct hash_table *table);

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

Note the third parameter, the application key. The application key is a `void` pointer since this is the only "generic" type in C. When you store generic application keys that the hash table cannot know how to deal with, you need a way to compare and delete them. For this, you add functions to the table. You do not want to add them to the table operations, so you put the pointers there. Let's add these key comparison and deletion functions to the constructor.

You need the comparison function if the application keys are complex. If you can compare them with `==`, you can leave this as null. If the comparison is not a simple constant or pointer comparison, for example, if they are strings and must be compared using `strcmp`, you need a function for comparison.

The destructor function is there for memory management. When you remove a key from the table using `delete_key` or free the entire table using `delete_table`, you need to know what to do with the application keys.

If they do not need to be deleted, for example, if the `void` pointers are computer words used as numbers, or they are stack-allocated data, you do not need to worry about them. Their memory management is taken care of elsewhere in the application, and you can leave the destructor as a null pointer. If the keys are allocated on the heap, and the hash table is the only reference to them, you need to free the memory they occupy. In general, you cannot simply call `free` on the values. If you do, you will not free memory that is only pointed to by the key and not directly nested.

Chained Hashing

You do not need to do much to modify the chained hashing data structure because a modified linked list does all the heavy lifting. You only need to add pointers for the two application-key functions to the table structure:

```
struct hash_table {
    struct bin *table;
    uint32_t size;
    uint32_t used;
    uint32_t active;
    compare_func cmp;
    destructor_func destructor;
};
```

You also need to add application keys to the table operations, of course, which is a question of propagating hash keys and application keys to list functions, so you consider the updates to the list structure first.

Updating Linked Lists

To store application keys in a linked list, you need to add this to the link structure:

```
struct linked_list {
    uint32_t hash_key;
    void *key;
    struct linked_list *next;
};
```

You also need to update the interface to the lists. For all the functions that used to take a hash key as an argument, you now need both a hash key and an application key. You also have to add application key comparisons and deletion functions where appropriate.

```
void delete_linked_list(struct linked_list *list,
                        destructor_func destructor,
                        bool free_keys);

void list_insert_key   (struct linked_list *list,
                        uint32_t hash_key, void *key,
                        compare_func cmp);
bool list_contains_key (struct linked_list *list,
                        uint32_t hash_key, void *key,
                        compare_func cmp);
void list_delete_key   (struct linked_list *list,
                        uint32_t hash_key, void *key,
                        compare_func cmp,
                        destructor_func destructor);
```

The flag in the deletion function is needed later. Sometimes you want to delete the links in a chain without freeing the keys they store. You do this when resizing a table, when you move the keys to another table before you remove the old links.

In the hash table interface, you provided the function arguments to the constructor and remembered them for where you needed to use them. You could also do that for linked lists, but this would add a substantial memory overhead.

You used three computer words in the way you implemented links: one each for the hash key, the application key, and the next pointer. As adding two function pointers would almost double the size, it is better to provide the function pointers where needed. Furthermore, you have function pointers in the hash table so they will always be available when you access the lists. The relative overhead of storing two function pointers is insignificant in the hash table; storing them there, as opposed to requiring the user to provide them in each operation for the lists, reduces the risk of mixing up functions and accidentally using the wrong comparison or destructor function somewhere in your program.

The link constructor doesn't change, except that you set the sentinel's application key to null. You never actually look at the keys in the sentinel, so this isn't necessary, but it leaves the sentinel in a consistent state.

```
struct linked_list *new_linked_list()
{
    struct linked_list *sentinel =
        (struct linked_list*)malloc(sizeof(struct linked_list));
    sentinel->hash_key = 0;
    sentinel->key = 0;
    sentinel->next = 0;
    return sentinel;
}
```

The destructor has to destroy application keys if the `free_keys` flag is true, so you update it to this:

```
void delete_linked_list(struct linked_list *list,
                        destructor_func destructor,
                        bool free_keys)
{
    while (list) {
        struct linked_list *next = list->next;
        if (free_keys && destructor && list->key)
            destructor(list->key);
        free(list);
        list = next;
    }
}
```

You allow the destructor to be a null pointer and interpret this to mean that the application keys do not need to be deallocated. You then assume that the memory management is taken care of elsewhere. If the destructor is provided, though, you need to call it on the list link's application key before you free the link.

The `get_previous_link` function needs to be updated to check application key equality as well as hash key equality. You call the comparison function for this. You check the equality of hash keys first, and you do so using the `&&` operator. By using logical AND you exploit that `&&` expressions do not evaluate their right-hand side if the left-hand side is false. If the hash keys differ, you do not call the potentially expensive application key comparison function.

```
static struct linked_list *
get_previous_link(struct linked_list *list,
                  uint32_t hash_key, void *key,
                  compare_func cmp)
{
```

```
    while (list->next) {
        if (list->next->hash_key == hash_key &&
                cmp(list->next->key, key))
            return list;
        list = list->next;
    }
    return 0;
}
```

You do not worry about duplications when you add keys. If you want the hash table to avoid them, it must check for containment itself by using the `list_contains_key`. If you allow lists to contain duplicates, you can implement both sets and multisets. Therefore, the only change to `add_key` is setting the application key in the new link you prepend to the list. If you wish to avoid duplicates, you need to check and remove the key in the hash table insertion function.

```
void list_insert_key(struct linked_list *list,
                     uint32_t hash_key,
                     void *key,
                     compare_func cmp)
{
    // build link and put it at the front of the list.
    // the hash table checks for duplicates if we want to avoid
    those
    struct linked_list *new_link =
        (struct linked_list*)malloc(sizeof(struct linked_list));
    new_link->hash_key = hash_key;
    new_link->key = key;
    new_link->next = list->next;
    list->next = new_link;
}
```

When deleting a key, you need to propagate the comparison function to `get_previous_link`. If you find the key, you need to call the destructor function if it is not null.

```
void list_delete_key(struct linked_list *list,
                     uint32_t hash_key,
                     void *key,
                     compare_func cmp,
                     destructor_func destructor)
{
    struct linked_list *link =
        get_previous_link(list, hash_key, key, cmp);
    if (!link) return;

    struct linked_list *to_delete = link->next;
    link->next = to_delete->next;
    if (destructor) destructor(to_delete->key);
    free(to_delete);
}
```

The reason you check that both the destructor and the key are not null is that even if you never insert a null pointer as a key, the sentinel will still hold one, and you use this function to delete it as well.

Finally, for `list_contains_key`, you only need to update it by passing the comparison function on to `get_previous_link`:

```
bool list_contains_key(struct linked_list *list,
                       uint32_t hash_key,
                       void *key,
                       compare_func cmp)
{
    return get_previous_link(list, hash_key, key, cmp) != 0;
}
```

Updating the Hash Table

With the updated linked list implementation in place, you need minimal changes to the hash table. You need to make the constructor store the two function pointers:

```
struct hash_table *empty_table(uint32_t size,
                               compare_func cmp,
                               destructor_func destructor)
{
    struct hash_table *table =
        (struct hash_table *)malloc(sizeof(struct hash_table));
    table->table =
        (struct linked_list **)malloc(size * sizeof(struct
        linked_list *));
    for (int i = 0; i < size; ++i) {
        table->table[i] = new_linked_list();
    }
    table->size = size;
    table->used = 0;
    table->cmp = cmp;
    table->destructor = destructor;
    return table;
}
```

You need to pass the functions on to the list functions where necessary. For deleting the entire table you need the destructor:

```
void delete_table(struct hash_table *table)
{
    for (int i = 0; i < table->size; ++i) {
        delete_linked_list(table->table[i], table->destructor,
        true);
    }
```

```
    free(table->table);
    free(table);
}
```

The resize function needs to pass the destructor function and the application keys on to the list:

```
static void resize(struct hash_table *table, uint32_t new_size)
{
    // Remember these...
    uint32_t old_size = table->size;
    struct linked_list **old_bins = table->table;

    // Set up the new table
    table->table =
        (struct linked_list **)malloc(new_size * sizeof(struct
        linked_list *));
    for (int i = 0; i < new_size; ++i) {
        table->table[i] = new_linked_list();
    }
    table->size = new_size;
    table->used = 0;

    // Copy keys
    for (int i = 0; i < old_size; ++i) {
        struct linked_list *list = old_bins[i];
        while ( (list = list->next) ) {
            insert_key(table, list->hash_key, list->key);
        }
    }
```

```
    // Delete old table but not the resources
    // (we have moved those)
    for (int i = 0; i < old_size; ++i) {
        delete_linked_list(old_bins[i], table->destructor, false);
    }
    free(old_bins);
}
```

For the remaining functions, you only need to pass application keys and functions on to the corresponding list functions:

```
void insert_key(struct hash_table *table,
                uint32_t hash_key, void *key)
{
    uint32_t mask = table->size - 1;
    uint32_t index = hash_key & mask;

    if (list_contains_key(table->table[index],
                          hash_key, key, table->cmp)) {
        list_delete_key(table->table[index],
                        hash_key, key,
                        table->cmp, table->destructor);
        list_insert_key(table->table[index],
                        hash_key, key, table->cmp);
    } else {
        list_insert_key(table->table[index],
                        hash_key, key, table->cmp);
        table->used++;
    }

    if (table->used > table->size / 2)
        resize(table, table->size * 2);
}
```

```
bool contains_key(struct hash_table *table,
                  uint32_t hash_key, void *key)
{
    uint32_t mask = table->size - 1;
    uint32_t index = hash_key & mask;
    return list_contains_key(table->table[index],
                             hash_key, key,
                             table->cmp);
    }

    void delete_key(struct hash_table *table,
                    uint32_t hash_key, void *key)
    {
        uint32_t mask = table->size - 1;
        uint32_t index = hash_key & mask;

        if (list_contains_key(table->table[index],
                              hash_key, key, table->cmp)) {
            list_delete_key(table->table[index],
                            hash_key, key,
                            table->cmp,
                            table->destructor);
                            table->used--;
        }

    if (table->used < table->size / 8)
        resize(table, table->size / 2);
}
```

In the insertion function, there is a design choice that you may want to change in your own application. If the key is already in the table, you delete it before you insert the new key. This is to free the memory allocated for the old key. In the hash table, you do not know if identical keys contain resources that are acquired separately in each key or if they share them.

This is why you have a comparison function instead of comparing keys by pointer equality. In this implementation, you assume that keys must be deleted when you insert an equivalent copy.

Open Addressing

For open addressing hash tables, the changes are no harder than for the chaining table. You first add application keys to the bins:

```
struct bin {
    bool is_free : 1;
    bool is_deleted : 1;
    uint32_t hash_key;
    void *key;
};
```

You also need to remember the comparison and destructor function pointers in the table:

```
struct hash_table {
    struct bin *table;
    uint32_t size;
    uint32_t used;
    uint32_t active;
    compare_func cmp;
    destructor_func destructor;
};
```

For the table constructor, you also need to store the function pointers:

```
struct hash_table *empty_table(uint32_t size,
                               compare_func cmp,
                               destructor_func destructor)
{
```

```
    struct hash_table *table =
        (struct hash_table*)malloc(sizeof(struct hash_table));
    table->table =
        (struct bin *)malloc(size * sizeof(struct bin));
    struct bin *end = table->table + size;
    for (struct bin *bin = table->table; bin != end; ++bin) {
        bin->is_free = true;
        bin->is_deleted = false;
    }
    table->size = size;
    table->active = table->used = 0;
    table->cmp = cmp;
    table->destructor = destructor;
    return table;
}
```

The destructor gets slightly more complicated than what you had before. Earlier, you could just `free` the array and then the table structure, but now you need to iterate over the entire table to call the destructor on the keys stored there:

```
void delete_table(struct hash_table *table)
{
    if (table->destructor) {
        struct bin *end = table->table + table->size;
        for (struct bin *bin = table->table; bin != end; ++bin)
{
            if (bin->is_free || bin->is_deleted) continue;
            table->destructor(bin->key);
        }
    }
    free(table->table);
    free(table);
}
```

When you resize the array in the hash table, you need to copy the keys to the new array. You should *not*, however, call the key destructor on the keys in it. You must `free` the memory used by the old array, but the keys have moved to the new array and that array is now responsible for deallocating them.

```
static void resize(struct hash_table *table, uint32_t new_size)
{
    // Remember the old bins until we have moved them.
    struct bin *old_bins = table->table;
    uint32_t old_size = table->size;

    // Update table so it now contains the new bins
    table->table =
      (struct bin *)malloc(new_size * sizeof(struct bin));
    struct bin *end = table->table + new_size;
    for (struct bin *bin = table->table; bin != end; ++bin) {
        bin->is_free = true;
        bin->is_deleted = false;
    }
    table->size = new_size;
    table->active = table->used = 0;

    // Move the values from the old bins to the new,
    // using the table's insertion function
    end = old_bins + old_size;
    for (struct bin *bin = old_bins; bin != end; ++bin) {
        if (bin->is_free || bin->is_deleted) continue;
        insert_key(table, bin->hash_key, bin->key);
    }

    // Finally, free memory for old bins
    free(old_bins);
}
```

In the insertion operation, you need to update the case when you find a matching hash key. Then, you need to compare the application keys to see if this key is already in the table. If it is, you do nothing. If you want a multiset instead of a set, you should ignore this check and insert. Otherwise, when you find an empty bin, you have to save both the application keys and the hash key, but otherwise, it works as before.

```
void insert_key(struct hash_table *table,
                uint32_t hash_key, void *key)
{
    uint32_t index;
    bool contains = contains_key(table, hash_key, key);
    for (uint32_t i = 0; i < table->size; ++i) {
        index = p(hash_key, i, table->size);
        struct bin *bin = & table->table[index];

        if (bin->is_free) {
            bin->hash_key = hash_key; bin->key = key;
            bin->is_free = bin->is_deleted = false;

            // we have one more active element
            // and one more unused cell changes character
            table->active++; table->used++;
            break;
        }

        if (bin->is_deleted && !contains) {
            bin->hash_key = hash_key; bin->key = key;
            bin->is_free = bin->is_deleted = false;

            // we have one more active element
            // but we do not use more cells since the
            // deleted cell was already used.
```

```
            table->active++;
            break;
        }

        if (bin->hash_key == hash_key) {
            if (table->cmp(bin->key, key)) {
                delete_key(table, hash_key, key);
                insert_key(table, hash_key, key);
                return; // Done
            } else {
                // we have found the key but with as
                // different value...
                // find an empty bin later.
                continue;
            }
        }
    }

    if (table->used > table->size / 2)
        resize(table, table->size * 2);
}
```

Like the chained hashing case, you delete an old key when the new one matches it to free resources it might have allocated.

For the `contains_key` function, you need to compare application keys when hash keys match:

```
bool contains_key(struct hash_table *table,
                  uint32_t hash_key, void *key)
{
    for (uint32_t i = 0; i < table->size; ++i) {
        uint32_t index = p(hash_key, i, table->size);
        struct bin *bin = & table->table[index];
```

```
        if (bin->is_free)
            return false;
        if (!bin->is_deleted && bin->hash_key == hash_key &&
            table->cmp(bin->key, key))
            return true;
    }
    return false;
}
```

Finally, for deletion, you again need to compare application keys when hash keys match, and you need to call the key destructor if necessary:

```
void delete_key(struct hash_table *table,
                uint32_t hash_key, void *key)
{
    for (uint32_t i = 0; i < table->size; ++i) {
        uint32_t index = p(hash_key, i, table->size);
        struct bin * bin = & table->table[index];

        if (bin->is_free) return;


        if (!bin->is_deleted && bin->hash_key == hash_key &&
            table->cmp(bin->key, key)) {
            bin->is_deleted = true;
            if (table->destructor)
                table->destructor(table->table[index].key);
                table->active--;
                break;
            }
        }

        if (table->active < table->size / 8)
            resize(table, table->size / 2);
    }
```

Implementing Hash Maps

A common use of hash tables is mapping keys to values where both keys and values are application-dependent. You associate values with keys and allow the table to delete them if you remove the corresponding key. You do not implement lookup or deletion based on values. You can return the value associated with a key, but any use of values is not the table's responsibility.

The hash map interface looks like this:

```
struct hash_table *
empty_table(uint32_t size,
            compare_func key_cmp,
            destructor_func key_destructor,
            destructor_func val_destructor);

void delete_table(struct hash_table *table);


void add_map     (struct hash_table *table,
                  uint32_t hash_key,
                  void *key, void *value);
void delete_key  (struct hash_table *table,
                  uint32_t hash_key, void *key);
void *lookup_key (struct hash_table *table,
                  uint32_t hash_key, void *key);
```

You only need to compare keys, so the comparison function in the constructor is for that. You will need to delete both application keys and values, so the constructor requires two functions.

When you insert values, you need to know the hash key, the application key, and the application value. When you look up or delete, you use the keys but not the value.

You no longer have a function that tests if a key is in the table. You have replaced it with a function that gets the value associated with a key. If the key is not in the table, you will get a null pointer back so the lookup function can be used to check membership as well.[1]

Chained Hashing

For the chained hashing table, you update the structure to hold the key-comparison function and the two destructors:

```
struct hash_table {
    struct linked_list **table;
    uint32_t size;
    uint32_t used;
    compare_func key_cmp;
    destructor_func key_destructor;
    destructor_func val_destructor;
};
```

Most of the updates to this hash table are found in the linked list implementation, as was the case for sets.

Updates to the Linked Lists

The updated interface to lists is this:

```
struct linked_list {
    uint32_t hash_key;
    void *key;
```

[1]This will fail if you try to map keys to null pointers; then you would need to add the contains operation as a separate function.

```
    void *value;
    struct linked_list *next;
};

typedef void (*destructor_func)(void *);
typedef bool (*compare_func)(void *, void *);

struct linked_list *new_linked_list();

void delete_linked_list(struct linked_list *list,
                        destructor_func key_destructor,
                        destructor_func val_destructor,
                        bool free_resouces);
void list_add_map      (struct linked_list *list,
                        uint32_t hash_key,
                        void *key, void *value);
bool list_delete_key   (struct linked_list *list,
                        uint32_t hash_key, void *key,
                        compare_func key_cmp,
                        destructor_func key_destructor,
                        destructor_func value_destructor);
struct linked_list *
list_lookup_key        (struct linked_list *list,
                        uint32_t hash_key, void *key,
                        compare_func key_cmp);
```

You need destructors for both keys and values when deleting elements. When you add elements, you prepend the list with a new link, so you do not need comparisons or destructors; you only need to provide the hash key, the application key, and the value. When deleting elements you need the key-comparison function to identify the link you should remove, and you need the two destructor functions to free heap-allocated memory.

Finally, you replace the contains operation with a lookup operation. For the lookup operation, you need the application key comparison function as well as the application key and the hash key.

With this interface, you make a few assumptions about how you use the lists in the hash table. You are assuming that you only store one link with any given key. The list implementation does not guarantee this; the `add_map` function inserts the key-value pair without checking if the key already maps to a value. To avoid duplications, you must check if a key is already in the table in the table's `insert` function. If you do not, then the old map will be left in the table if you delete the key again because it can be found in a later link in the same list.

In the set implementation, this was also the case. The set interface would not break if you violated this invariant. The performance might have degraded since you would store the same values multiple times and thus increase the load factor, but that would be all. The `contains_key` function would work correctly even if the same key was inserted multiple times. The `contains_key` could easily be modified to count how many occurrences of a given key was in the list, in case you wanted a multiset instead of a set.

With `lookup_key`, you get the value associated with the first link with the given key. The behavior you expect for a map is that the value associated with a key is the last value you mapped the key to, but since you prepend new links to lists, this is also what you get. If you never delete keys, as for `contains_key`, the function will work even if you do not enforce that each key is only found once in the list. The only effect you should see, if any, is a degradation in performance. If you delete a key-value map, however, you also only delete the first link in the list. If there is another key-value map later in the list, the key will still be in the map but it will point to the old value. You explicitly take care of this issue in the table's insertion function.

Anyway, on to the implementation of the updated linked lists. You add the value to new links; otherwise, there is nothing new in the constructor:

```
struct linked_list *new_linked_list()
{
    struct linked_list *sentinel =
        (struct linked_list*)malloc(sizeof(struct linked_list));
    sentinel->hash_key = 0;
    sentinel->key = 0;
    sentinel->value = 0;
    sentinel->next = 0;
    return sentinel;
}
```

For the destructor, you now need to delete both application keys and application values:

```
void delete_linked_list(struct linked_list *list,
                        destructor_func key_destructor,
                        destructor_func val_destructor,
                        bool free_resources)
{
    while (list != 0) {
        struct linked_list *next = list->next;
        if (free_resources && key_destructor && list->key)
            key_destructor(list->key);
        if (free_resources && val_destructor && list->value)
            val_destructor(list->value);
        free(list);
        list = next;
    }
}
```

When you insert elements, you again have both an application key and an application value:

```
void list_add_map(struct linked_list *list,
                  uint32_t hash_key,
                  void *key, void *value)
{
    struct linked_list *link =
        (struct linked_list*)malloc(
            sizeof(struct linked_list)
        );
    link->hash_key = hash_key;
    link->key = key;
    link->value = value;
    link->next = list->next;
    list->next = link;
}
```

When you get the previous link to a key, you need to compare hash keys and application keys. You do not compare values. You do not care about them when you want to find the link for a key.

```
static struct linked_list *
get_previous_link(struct linked_list *list,
                  uint32_t hash_key, void *key,
                  compare_func key_cmp)
{
    while (list->next) {
        if (list->next->hash_key == hash_key &&
            key_cmp(list->next->key, key))
            return list;
        list = list->next;
    }
    return 0;
}
```

For deletion, you need to update the call to the previous link function. In that link, you need to destroy both keys and values (if the destructors are there).

```
bool list_delete_key(struct linked_list *list,
                     uint32_t hash_key, void *key,
                     compare_func key_cmp,
                     destructor_func key_destructor,
                     destructor_func val_destructor)
{
    struct linked_list *link =
        get_previous_link(list, hash_key, key, key_cmp);
    if (!link) return false;

    // we need to get rid of link->next
    struct linked_list *to_delete = link->next;
    link->next = to_delete->next;
    if (key_destructor) key_destructor(to_delete->key);
    if (val_destructor) val_destructor(to_delete->value);
    free(to_delete);
    return true;
}
```

Finally, for the `lookup` function, you modify the code from the `contains` function. You get the previous link to the key, using the application key and the application key comparison function. If the link is null, the table does not contain the key, so you return a null pointer. If the link is there, you return it.

```
struct linked_list *
list_lookup_key(struct linked_list *list,
                uint32_t hash_key, void *key,
                compare_func key_cmp)
{
```

```
    struct linked_list *link =
        get_previous_link(list, hash_key, key, key_cmp);
    return link ? link->next : 0;
}
```

Updates to the Hash Table

You've finally gotten to the updated hash table implementation. The updates to the constructor are almost trivial. You need to store a key-comparison function, and you need to store destructors for both keys and values. Other than that, there is nothing new.

```
struct hash_table *empty_table(uint32_t size,
                               compare_func key_cmp,
                               destructor_func key_destructor,
                               destructor_func val_destructor)
{
    struct hash_table *table =
        (struct hash_table *)malloc(sizeof(struct hash_table));
    table->table =
        (struct linked_list **)malloc(size * sizeof(struct
        linked_list *));
    for (int i = 0; i < size; ++i) {
        table->table[i] = new_linked_list();
    }
    table->size = size;
    table->used = 0;
    table->key_cmp = key_cmp;
    table->key_destructor = key_destructor;
    table->val_destructor = val_destructor;
    return table;
}
```

In the destructor, you update the parameters to the list destructor:

```
void delete_table(struct hash_table *table)
{
    for (int i = 0; i < table->size; ++i) {
        delete_linked_list(table->table[i],
                           table->key_destructor,
                           table->val_destructor,
                           true);
    }
    free(table->table);
    free(table);
}
```

The only changes to the resize function are also updates to the calls to list functions:

```
static void resize(struct hash_table *table, uint32_t new_size)
{
    // Remember these...
    uint32_t old_size = table->size;
    struct linked_list **old_bins = table->table;

    // Set up the new table
    table->table =
        (struct linked_list **)malloc(
            new_size * sizeof(struct linked_list *)
        );
    for (int i = 0; i < new_size; ++i) {
        table->table[i] = new_linked_list();
    }
```

```
    table->size = new_size;
    table->used = 0;

    // Copy maps
    for (int i = 0; i < old_size; ++i) {
        struct linked_list *list = old_bins[i];
        while ( (list = list->next) ) {
            add_map(table, list->hash_key,
                    list->key, list->value);
        }
    }

    // Delete old table
    for (int i = 0; i < old_size; ++i) {
        delete_linked_list(old_bins[i],
                           table->key_destructor,
                           table->val_destructor,
                           false);
    }
    free(old_bins);
}
```

For the insert function, you approach the operation a little differently. You want to replace the value if the list has an existing value associated with a key. So, you get the link from the list; if the key is already in the list, you free the old key and value, and insert the new key-value map. If the link you get from the list is null, then you know the key is not in the list, and you prepend a new link containing the new key and value. If you used the old contains interface in this function, you need to find the link holding the key twice. First, you call the function in the test to see if the key is already in the table—something you need to know in order to update the `used`

counter when necessary. Second, you call it to find the link to update. Since `link` will be the null pointer if the key is not already in the table, you can get away with a single call to `lookup_key`.[2]

```
void add_map(struct hash_table *table,
             uint32_t hash_key,
             void *key, void *value)
{
    uint32_t mask = table->size - 1;
    uint32_t index = hash_key & mask;
    struct linked_list *list = table->table[index];

    struct linked_list *link =
        list_lookup_key(list, hash_key, key, table->key_cmp);

    if (link != 0) {
        // the key exists in the table, replace the value.
        if (table->key_destructor)
            table->key_destructor(link->key);
        if (table->val_destructor)
            table->val_destructor(link->value);
        link->key = key;
        link->value = value;
    } else {
        // the key is new, so insert it and the value
        list_add_map(list, hash_key, key, value);

        table->used++;
    }
```

[2]If you worry that the earlier implementations where you *did* use the `contains` function are inefficient, don't. There, you would not update the link if it was already found in the list, and if it was not, you prepended a new link. You wouldn't search through the linked list twice in those operations.

```
    if (table->used > table->size / 2)
        resize(table, table->size * 2);
}
```

The updates to the deletion operation are straightforward. You only update the call to the linked list function:

```
void delete_key(struct hash_table *table,
                uint32_t hash_key, void *key)
{
    uint32_t mask = table->size - 1;
    uint32_t index = hash_key & mask;
    struct linked_list *list = table->table[index];
    bool deleted = list_delete_key(
        list, hash_key, key, table->key_cmp,
        table->key_destructor, table->val_destructor
    );

    if (deleted) table->used--;

    if (table->used < table->size / 8)
        resize(table, table->size / 2);
}
```

Finally, the `lookup` function forwards the operation to the linked list. If it gets a link back, you return the link's value. If the link you get from the list is null, you return null.

```
void *lookup_key(struct hash_table *table,
                 uint32_t hash_key, void *key)
{
    uint32_t mask = table->size - 1;
    uint32_t index = hash_key & mask;
    struct linked_list *list = table->table[index];
```

```
    struct linked_list *link =
        list_lookup_key(list, hash_key, key, table->key_cmp);
    return link ? link->value : 0;
}
```

Open Addressing

For the open addressing strategy, you have the same interface as the chained hash map, and you define the data structures like this:

```
struct bin {
    bool is_free : 1;
    bool is_deleted : 1;
    uint32_t hash_key;
    void *key;
    void *value;
};

struct hash_table {
    struct bin *table;
    uint32_t size;
    uint32_t used;
    uint32_t active;
    compare_func key_cmp;
    destructor_func key_destructor;
    destructor_func val_destructor;
};
```

You update the `bin` structure to contain both an application key and a value, and you update the `hash_table` structure to contain a key-comparison function and two destructors. In the constructor, all you need is to set the new functions in the `struct`.

```
struct hash_table *empty_table(uint32_t size,
                               compare_func key_cmp,
                               destructor_func key_destructor,
                               destructor_func val_destructor)
{
    struct hash_table *table =
        (struct hash_table*)malloc(sizeof(struct hash_table));
    table->table =
        (struct bin *)malloc(size * sizeof(struct bin));

    struct bin *end = table->table + size;
    for (struct bin *bin = table->table; bin != end; ++bin) {
        bin->is_free = true;
        bin->is_deleted = false;
    }
    table->size = size;
    table->active = table->used = 0;
    table->key_cmp = key_cmp;
    table->key_destructor = key_destructor;
    table->val_destructor = val_destructor;
    return table;
}
```

For the destructor, you need to iterate through all the bins if any of the destructors are defined and then invoke both (if defined) when you find a bin containing active maps:

```
void delete_table(struct hash_table *table)
{
    if (table->key_destructor || table->val_destructor) {
        struct bin *end = table->table + table->size;
        for (struct bin *bin = table->table; bin != end; ++bin)
{
            if (bin->is_free || bin->is_deleted) continue;
```

```
            if (table->key_destructor)
                table->key_destructor(bin->key);
            if (table->val_destructor)
                table->val_destructor(bin->value);
        }
    }
    free(table->table);
    free(table);
}
```

In the resize function, you need to update the call to add_map to match the new interface:

```
static void resize(struct hash_table *table, uint32_t new_size)
{
struct bin *old_bins = table->table;
uint32_t old_size = table->size;

table->table = (struct bin *)malloc(new_size * sizeof(struct bin));
struct bin *end = table->table + new_size;
for (struct bin *bin = table->table; bin != end; ++bin) {
bin->is_free = true;
bin->is_deleted = false;
}
table->size = new_size;
table->active = table->used = 0;
    end = old_bins + old_size;
    for (struct bin *bin = old_bins; bin != end; ++bin) {
        if (bin->is_free || bin->is_deleted) continue;
        add_map(table, bin->hash_key, bin->key, bin->value);
    }

    free(old_bins);
}
```

For add_map, you need to insert values as well as keys:

```
void add_map(struct hash_table *table,
             uint32_t hash_key,
             void *key, void *value)
{
    uint32_t index;
    bool contains = (bool)lookup_key(table, hash_key, key);
    for (uint32_t i = 0; i < table->size; ++i) {
        index = p(hash_key, i, table->size);
        struct bin *bin = & table->table[index];

        if (bin->is_free) {
            bin->is_free = bin->is_deleted = false;
            bin->hash_key = hash_key;
            bin->key = key;
            bin->value = value;
            // we have one more active element
            // and one more unused cell changes character
            table->active++; table->used++;
            break;
        }

        if (bin->is_deleted && !contains) {
            bin->is_free = bin->is_deleted = false;
            bin->hash_key = hash_key;
            bin->key = key;
            bin->value = value;
            // we have one more active element
            // but we do not use more cells since the
            // deleted cell was already used.
            table->active++;
            break;
        }
```

```
        if (bin->hash_key == hash_key) {
            if (table->key_cmp(bin->key, key)) {
                delete_key(table, hash_key, key);
                add_map(table, hash_key, key, value);
                return; // Done
            } else {
                // we have found the key but with a
                // different value...
                continue;
            }
        }
    }

    if (table->used > table->size / 2)
        resize(table, table->size * 2);
}
```

For delete_key, you have to deallocate both the application key and application value if the destructors are not null:

```
void delete_key(struct hash_table *table,
                uint32_t hash_key, void *key)
{
    for (uint32_t i = 0; i < table->size; ++i) {
        uint32_t index = p(hash_key, i, table->size);
        struct bin *bin = & table->table[index];

        if (bin->is_free) return;

        if (!bin->is_deleted
            && bin->hash_key == hash_key
            && table->key_cmp(bin->key, key)) {
            bin->is_deleted = true;
```

```
            if (table->key_destructor)
                table->key_destructor(bin->key);
            if (table->val_destructor)
                table->val_destructor(bin->value);
            table->active--;
            break;
        }
    }

    if (table->active < table->size / 8)
        resize(table, table->size / 2);
}
```

Finally, for `lookup_key`, you essentially do the same as you did for `contains_key`, except that you return the bin's value instead of `true` if you find the key in the table:

```
void *lookup_key (struct hash_table *table,
                  uint32_t hash_key, void *key)
{
    for (uint32_t i = 0; i < table->size; ++i) {
        uint32_t index = p(hash_key, i, table->size);
        struct bin *bin = & table->table[index];
        if (bin->is_free)
            return 0;
        if (!bin->is_deleted
            && bin->hash_key == hash_key
            && table->key_cmp(bin->key, key))
            return bin->value;
    }
    return 0;
}
```

CHAPTER 6

Heuristic Hash Functions

The main topic of this book is implementing hash tables; it's only secondarily about hash functions. This is why you have assumed a priori that you have uniformly distributed hash keys. In reality, this is unlikely to be the case; real data are rarely random samples from the space of possible data values. In this chapter, you will learn about commonly used heuristic hash functions. In the next chapter, you will see an approach to achieving stronger probabilistic guarantees.

To set the tone for this chapter, you will first consider two cases of data that are not randomly distributed. To make it easier to visualize the data, assume that the data takes values that can be represented in 16 bits, for example, in the range of [0, 65535], and that you have 64 data points. You will map down these data points within length ranges of 8, 16, 32, and 64 for powers of two, or 7, 17, 31, and 67 for tables with lengths that are prime numbers.[1]

[1]For all but two of the tables, that of size 64 and that of size 67, this means that the load is higher than 1, so this obviously will only work for chained hashing. The purpose of the examples in this chapter, however, is merely to show how keys are distributed over bins with tables of different sizes, so don't worry about conflict resolution and load.

T. Mailund, *The Joys of Hashing*, https://doi.org/10.1007/978-1-4842-4066-3_6

Two pathological cases have been set up: one with consecutive numbers from 0 to 128, and one with the same numbers but shifted 2 bits. The latter is to emulate the case where you have pointers that are 4 bytes apart; usually, you will have pointers that are aligned with computer words, which are likely to be 4 or 8 bytes apart, so the lowest 2 or 4 bits will be 0. The numbers can all be represented in 8 bits, but you will allow the hash keys to take values in 16 bits. Although you compute the hash keys in 32-bit words, you mask out the 2 least-significant bytes.

Figure 6-1 plots the first case and Figure 6-2 the second. The input (application) keys are not plotted. You would not be able to see them in either plot; they are at the very left end of the x-axis when $N = 2^{16}$. You can see this in the hash keys plot when you use the identity hash function. The hash keys are plotted so you can see how different functions will spread them over the hash key space.

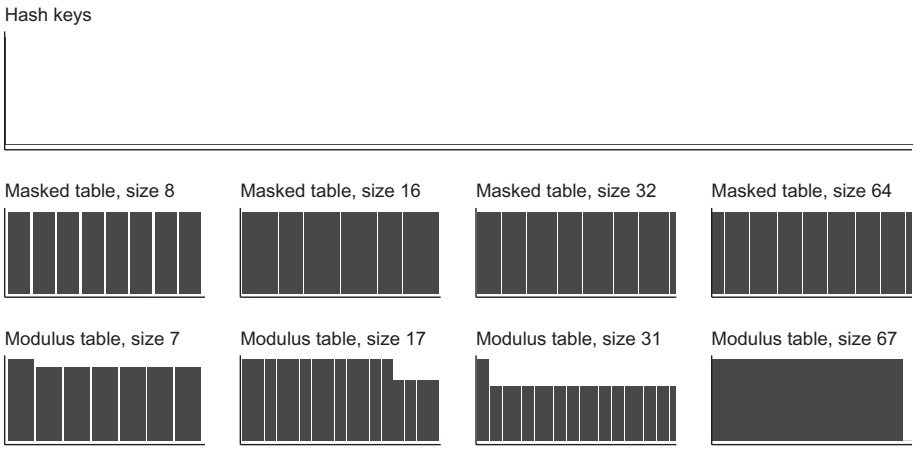

Figure 6-1. *Consecutive numbers directly mapped to hash keys*

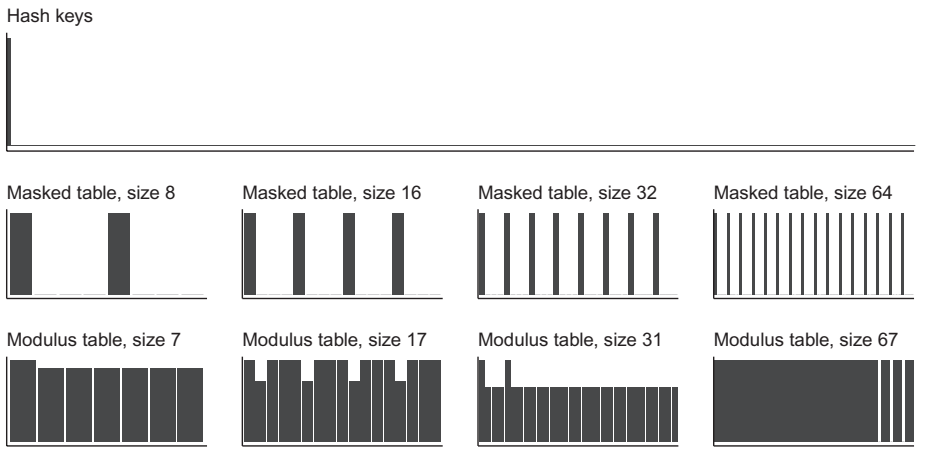

Figure 6-2. *Numbers shifted two bits directly mapped to hash keys*

As you can see, the hash keys are far from uniform in their range. For the first case, there are good spreads for both the masked and the prime number hash tables while in the second case, there's only a good spread for the prime number-sized hash tables. This should not be a surprise. You base your bins on the lower bits when you mask them, and you get a poor distribution when you shift them. The second case is the worst setups for masking, thus the need for better hash functions. One of the reasons people prefer to use hash keys modulo a prime is precisely to avoid this problem. With good hash functions, however, you should be able to get closer to the goal of randomly distributed hash keys when you mask to get the lowest bits.

What Makes a Good Hash Function?

Before you start engineering hash functions, you should consider these properties:

1. They should be fast to compute.

2. They should be deterministic.[2]
3. They should aim at distributing values uniformly in their target domain.

You always want fast computations, and also for hash functions. The goal of hash tables is to achieve constant time lookups, but if the hash function is slow at computing the hash key, then much of the efficiency is lost. If time is so vital that you prefer masking bits instead of calculating the remainder modulus a prime, then the hash function shouldn't be slower than the modulus operation. This means that you should prefer bit-wise operations over arithmetic operations as well as adding and subtracting over multiplying and dividing.

The second property is essential. If the hash function is not deterministic, then you might end up with two different keys for the same value. If the hash key changes each time you want to look up a value, the hash table will not be able to do lookups. This point does not mean, however, that you cannot use randomization. As you will see later, you can use randomization to avoid poor performance of hash functions on pathological data where values map to a small range of hash table bins. If you use random values, however, they need to be parameters to the hash function so you can get deterministic behavior out of them.

The third property is why you need hash functions, and this property is the hardest to achieve. You get the best performance in a hash table when keys are spread uniformly over the hash table bin. If the hash table

[2]When I say deterministic here, I mean that a hash function should always produce the same output on the same input. There are plenty of randomized hash functions, in the sense that they use random numbers as part of their construction. You fix these random numbers when you use the function to hash application keys. You can change from one hash function to another by picking new random numbers, but you can't change them at arbitrary times if you want your function to consistently give you the same output for the same input. Universal hashing, which will be discussed in the next chapter, uses random numbers to create deterministic hash functions.

produces random keys, you will also get a uniform spread in bins. If it does not, there are no runtime guarantees. This is unfortunately impossible to guarantee with a single deterministic hash function. If you map k-bit values to an l-bit range, you are mapping 2^k possible values into 2^l keys. The best you can achieve, if the map is uniform, is to map 2^{k-l} values to each key. An adversary that knows your hash function can exploit this and maximize the number of collisions you get. With an ensemble of different hash functions, you can mitigate this by choosing a hash function dependent on the data. You can avoid adversarial data by randomly selecting (deterministic) hash functions. With a single good hash function that tends to map similar data to very different hash keys, though, you usually get good performance without randomization tricks.

Hashing Computer Words

For the remainder of this chapter, you'll consider two cases. The first case is where you have values that you can fit into a single computer word and want to scramble up the values there to make them evenly distributed. The second case is the more general case where you have a sequence of bytes for each value and where the length varies from value to value. Any case that does not fit the first can be handled by the second since you can serialize any data to a stream of bytes, even though it might require some programming to get there for some data structures.

In the following, you will only consider the case where you shifted the numbers 2 bits to the right. When you use consecutive numbers from 0 to 63 and shift them by 2 bits, the result still fits into a single byte. Although you will use 32-bit words in all the functions, remember that in the input, the only non-zero bits are in the least-significant byte. Also, remember that you mask the hash keys to the 16 least-significant bits for plotting purposes.

Now, from the identity hash function, you observe that the problem you had for the masked tables was that the lower bits were all 0. Having 3/4 of the bytes in the input values identical makes the test data somewhat adversarial. If you are hashing something like pointers, it is not unusual that the least significant bits are identical. If some bits in the input are always the same, the input domain is effectively smaller, and no hash function can compensate for this when scrambling the input. Still, if the hash function scrambles the remaining bits well, you should still be able to get good performance.

Before you get to the hash functions, let's go over some terminology. Consider Figure 6-3, which shows the components of a hash function when you deal with an entire computer word as a single unit. Assume that you have a parameter that goes into the function. This gives you a way to parameterize a function, and this value can be random as long as you use the same parameter every time you hash a value. The parameter can potentially be used to randomize a function if you get poor performance.

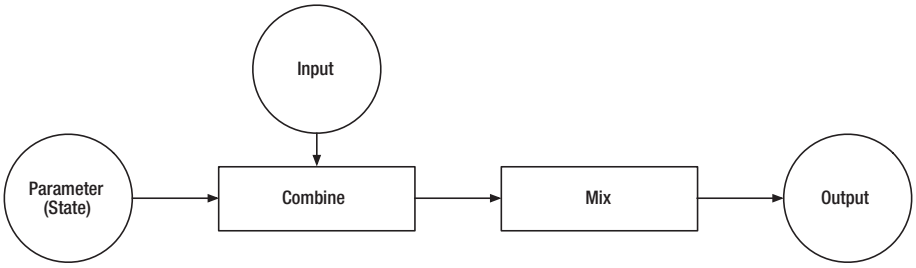

Figure 6-3. *Components of a single-word hash function*

This parameter is called a *state* (or *initial state*) because it works as an intermediate state when you hash multiple words; see Figure 6-4. Multiple words do not necessarily refer to complete computer words here, but also to individual subkeys, such as the 4 bytes that make up a 32-bit computer word. When you split a word into individual bytes or when you hash over multiple-word values, the output of a single computation in the hash

function behaves as the input of the next, and you call such values the states of the function as you process the input.

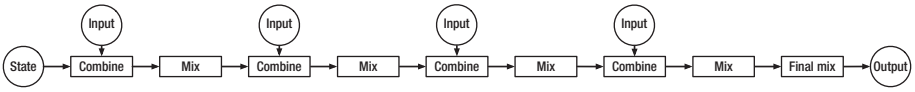

Figure 6-4. *Components of a multiple-words hash function*

When you hash, you first combine the input with the state of the function, starting from the initial state. Combining means that you XOR or add the state to the input. For scrambling the bits in the input, adding the input to the state is a slightly better choice since a single bit in the input will only affect a single bit in the output with XOR. When you add, a carry bit can propagate a single input bit several bits to the left in the result. For maximum speed, on the other hand, XOR is preferable.

You mix up the result after combining the input and state. In this step, you attempt to modify the state such that each bit in the state will affect several other bits in the result of the mixing. If you hash a value consisting of multiple components (bytes or words), you perform several combine and mix steps, and you might have some additional mixing after you have processed all the input.

Most functions presented in this chapter are taken from Bob Jenkins' excellent web page, `www.burtleburtle.net/bob/hash/doobs.html`, in some cases with minor modifications. I have not included all hash functions described there, but I have selected a few that tend to perform well. If you wish to explore more functions, Jenkins' web page is a good starting point. All functions take a 32-bit integer as its `state` input (even when this value is ignored), a 32-bit bit word for the `input`, and produce a 32-bit integer as output. I mask the last two bytes of the hash values for plotting purposes. I do this to limit the keys range to 65,536 values to make the plots easier to read. For the masking tables, this does not affect the bins

since they all mask out fewer than 8 bits (3 to 6 for the four sizes). For the modulo computation, the remainders are taken from the least significant bits, so while the table of size 67 has information from more than 6 bits, it does not vary when you modify bits higher than 7, so these functions are not affected either.

Additive Hashing

One of the simplest hash functions is the *additive hashing* function.[3] This function, shown below, combines the input and the state by addition and with no mixing. It does move all the 4 bits in a 32-bit word to the least significant byte, so the lower bits are potentially affected by the full 32-bit input; the higher bits would not affect the lower bits if you simply added `hash` and `input` in the function. For your test input, where the 3 most significant bytes are all 0, it behaves exactly as the identity function when the `state` parameter is 0; see Figure 6-5. When state is not 0, though, it still leaves the 2 lowest bits constant on your input. The 2 lowest bits will be copied directly from the `state` parameter and will not be affected by the `input`.

```
uint32_t additive_hash(uint32_t state, uint32_t input)
{
    uint32_t hash = state;
    uint8_t *p = (uint8_t*)&input;

    // combine
    hash += *(p++);
```

[3]The simplest I have seen was used to hash ASCII strings and only used the first character. For standard ASCII, there are only 128 characters (they use 7 bits per character), while for Extended ASCII there are 256. That is not the bad part, however. If you hash common words, such as variable names in a program, then they do not use the full set of ASCII characters. Using only the first character of a string is a very poor hash function.

```
    hash += *(p++);
    hash += *(p++);
    hash += *p;

    return hash;
}
```

Since the sizes of your masked hash tables are 8, 16, 32, and 64, the binned keys are 3, 4, 5, and 6 bits. Of these, the additive hashing function can only modify 1, 2, 3, and 4, respectively, since the lower 2 bits are constant. Your input spans all possible bit patterns of them, so you already have the best possible spread you can get using this hash function regardless of the `state` parameter. It cannot perform better than the identity hash function on your test data.

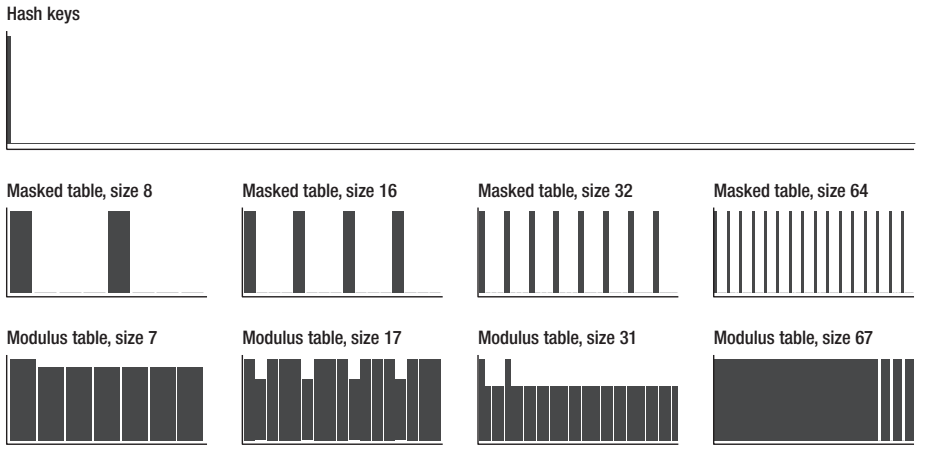

Figure 6-5. *Additive hashing*

Rotating Hashing

On hardware architectures where you have a rotate operation, implemented using shift and OR below, you can write a very fast hash function that operates using rotate and XOR:

```
#define rot(x,k) (((x)<<(k)) | ((x)>>(32-(k))))
uint32_t rotating_hash(uint32_t state, uint32_t input)
{
    uint32_t hash = state;
    uint8_t *p = (uint8_t*)&input;

    //     mix          ; combine
    hash ^=                *(p++);
    hash += rot(hash, 4) ^ *(p++);
    hash += rot(hash, 4) ^ *(p++);
    hash += rot(hash, 4) ^ *p;

    return hash;
}
```

Big-endian and small-endian architectures will combine the input bytes in different order. To reverse the order in which you add the bytes, you can implement the function like this:

```
uint32_t rotating_hash(uint32_t state, uint32_t input)
{
    uint32_t hash = state;
    uint8_t *p = ((uint8_t*)&input) + 3;

    //     mix          ; combine
    hash ^=                *(p--);
    hash += rot(hash, 4) ^ *(p--);
    hash += rot(hash, 4) ^ *(p--);
```

```
    hash += rot(hash, 4) ^ *p;
    return hash;
}
```

This function rotates the hash function state in each mixing step and then combines one byte at a time using XOR. By rotating, it can preserve input bits through many cycles of input, but for a single computer word, and in this application, it does not work well. If you first combine with the byte that contains different data, as you would do on a big-endian computer, the mixing operations shift out of the lower bits entirely, and the function would only depend on the addition that preserves bit positions through the three operations; see Figure 6-6. The input bytes are highlighted using boxes; the boxes over the right-most 8 bits show where the input bytes enter the function. With the first 3 bytes set to 0, as in the test data, and the 3 least-significant bits at 0 as well, you will not get a better spread over bins that you get with additive hashing, although the hash keys are spread out more than with additive hashing; see Figure 6-7.

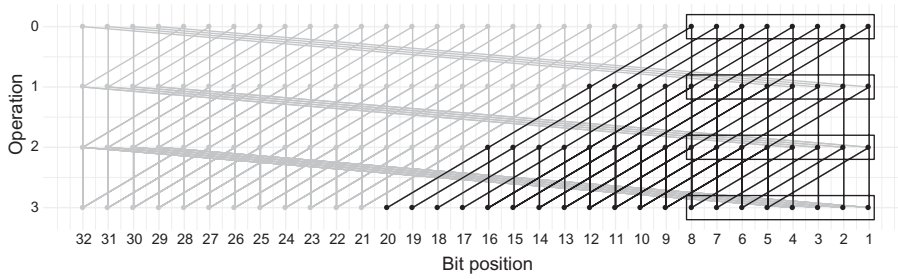

Figure 6-6. *Bits affected by the first input byte (shown in black). Addition (vertical edges) is shown as if it only affects single bits. In actuality, some bits to the left of an addition will be affected.*

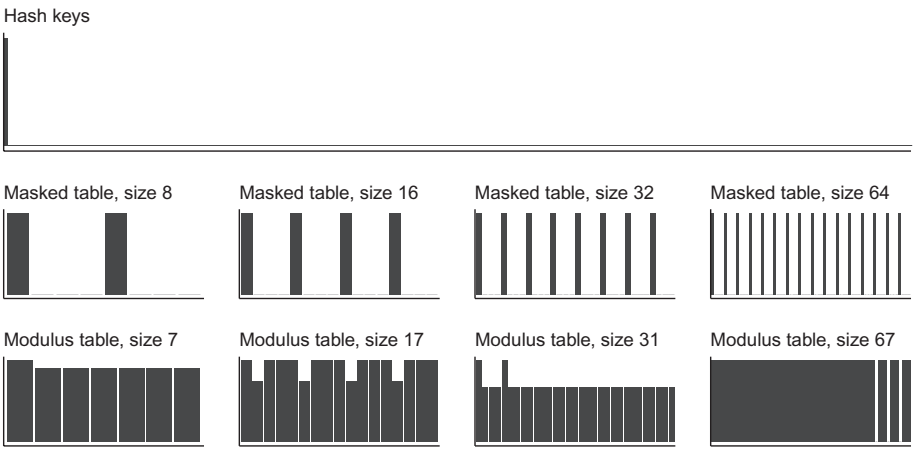

Figure 6-7. *Rotate hashing with the last byte carrying information*

With more bytes in the input, the first byte will wrap around and start affecting later bytes, but the periodicity in when the first byte affects the least significant bytes will be an issue. The best you can hope for is to return the initial byte to the lowest bits minus 2, where your 64 keys would take all possible values for your mapped bits. This solution, however, is only applicable to this test data and does not generalize.

When the informative byte in your test data is added in the last combination step, the rotation hash function is a simple XOR between a rotation of the initial state and the variable byte. If the initial state is 0, you will get the same performance as with the additive hashing; see Figure 6-8. Changing the initial state will affect the keys, because of the XOR operation, but as with the additive hashing, no choice of initial state will allow you to have anything but a constant for the last 2 bits in the key when the last 2 bits are constant in your input.

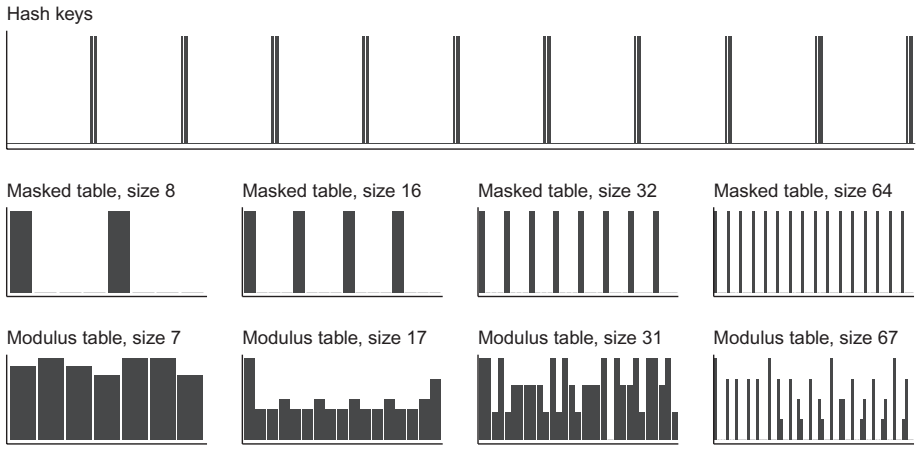

Figure 6-8. *Rotate hashing with the first byte carrying information*

Another way to plot the performance of the hash function is to show how the input bit patterns translate to output bit patterns. For the rotating hash function, see Figure 6-9 and Figure 6-10. The performance of the masking hash tables can be seen by looking at the last 3, 4, 5, and 6 bits. Here, you can see that the 2 least-significant bits do not change for the example input, which is why you get the poor performance.

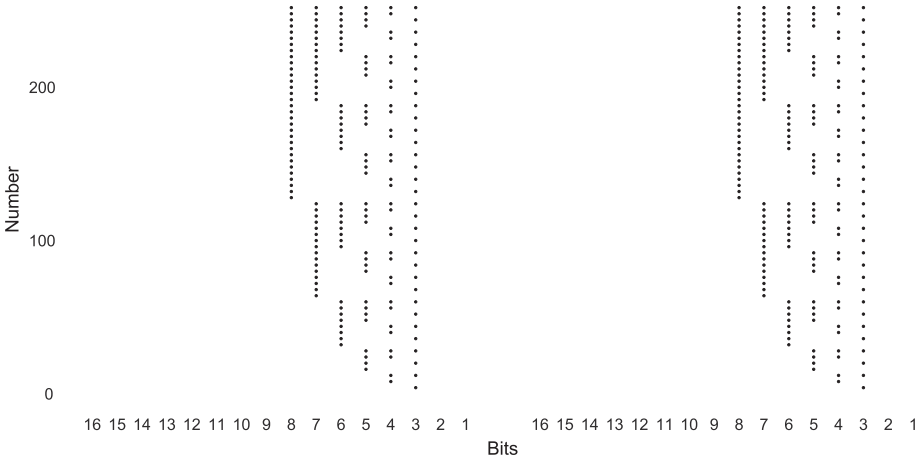

Figure 6-9. *Input and output bit patterns for the rotating hash function when the input is least-significant-byte last*

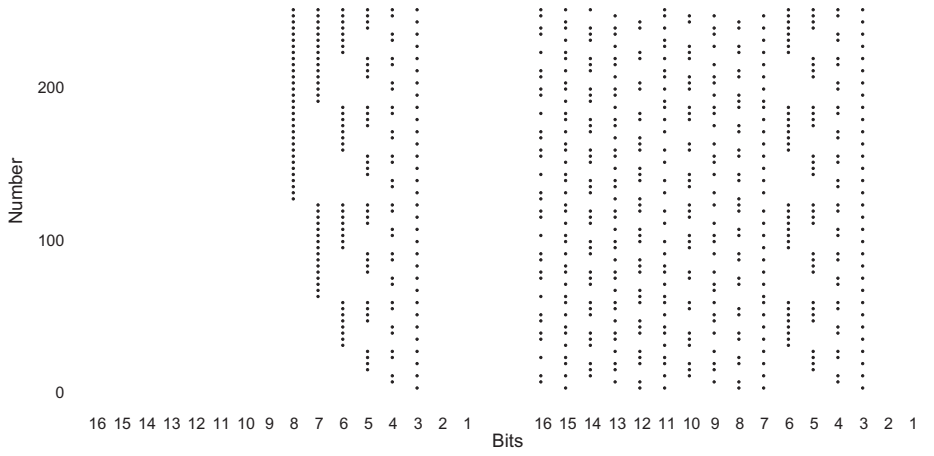

Figure 6-10. *Input and output bit patterns for the rotating hash function when the input is least-significant-byte first*

One-at-a-Time Hashing

The one-at-a-time hash function, developed by Bob Jenkins, uses addition to combine the state and the input, 1 byte at a time. It then mixes the result using bit-wise shift, addition, and XOR. You can implement it in two different ways, varying in the order in which you add the bytes in the input, so either

```
uint32_t one_at_a_time_hash(uint32_t state, uint32_t input)
{
    uint32_t hash = state;
    uint8_t *p = (uint8_t*)&input;

    // combine ; mix
    hash += *(p++); hash += (hash << 10); hash ^= (hash >> 6);
    hash += *(p++); hash += (hash << 10); hash ^= (hash >> 6);
    hash += *(p++); hash += (hash << 10); hash ^= (hash >> 6);
    hash += *p; hash += (hash << 10); hash ^= (hash >> 6);
```

```
    // final mix
    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);
    return hash;
}
```

or

```
uint32_t one_at_a_time_hash(uint32_t state, uint32_t input)
{
    uint32_t hash = state;
    uint8_t *p = ((uint8_t*)&input) + 3;

    // combine ; mix
    hash += *(p--); hash += (hash << 10); hash ^= (hash >> 6);
    hash += *(p--); hash += (hash << 10); hash ^= (hash >> 6);
    hash += *(p--); hash += (hash << 10); hash ^= (hash >> 6);
    hash += *p; hash += (hash << 10); hash ^= (hash >> 6);

    // final mix
    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);

    return hash;
}
```

Assuming you can assign and perform an operation in one instruction, so += and ^= are one operation, you spend 7 operations on combining, 4 × 6 on mixing, and 6 operations on the final mix, for a total of 37 operations.

Similar to how you visualized the rotating hash, you can see how the one-at-a-time hash function moves bits around. Figure 6-11 shows how the bits in the first byte propagate down through the operations and Figure 6-12 shows how the bits in the last byte propagate. Each mixing step consists of 2 operations, so the input bytes are added as operations 0, 2, 4, and 6. The last 3 operations are the final mixing.

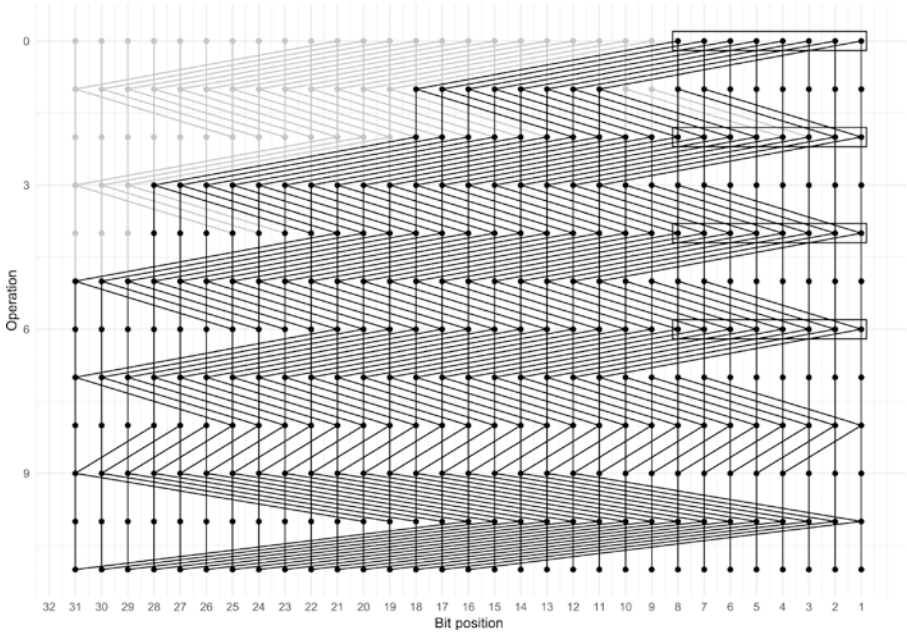

Figure 6-11. *Bits affected by the first input byte (shown in black) using one-at-a-time hashing*

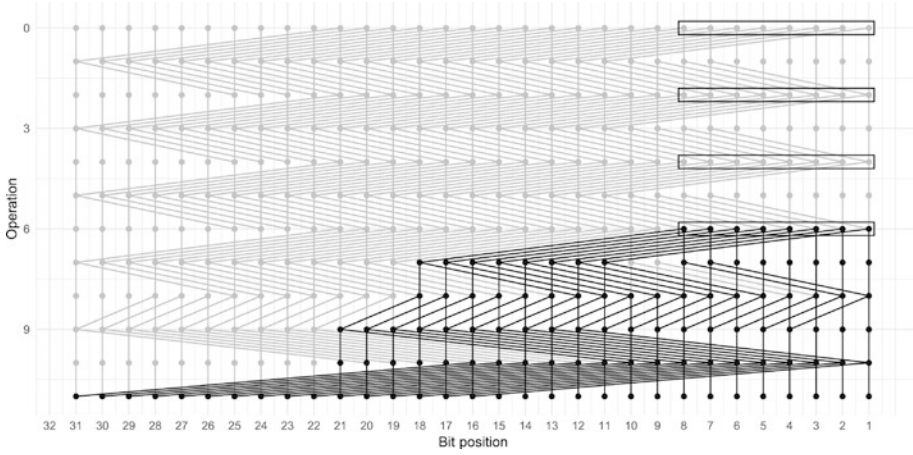

Figure 6-12. *Bits affected by the first last byte (shown in black) using one-at-a-time hashing*

Plotting how input bits propagate to the output bits tells you how many output bits are affected by the input and more importantly, how many input bits each output bit depends on. Figure 6-13 and Figure 6-14 plot this dependency for the least and the second-least significant bit—the bits that were constant in your previous attempts at hash functions. This isn't the entire story since some of the operations can cancel each other, but you can see that the least significant bit depends on the input bit from the first byte and all except bit 3 for the last byte. For the second-least significant bit, you can see that it depends on all the bits in the first byte and half of the bits of the last byte.

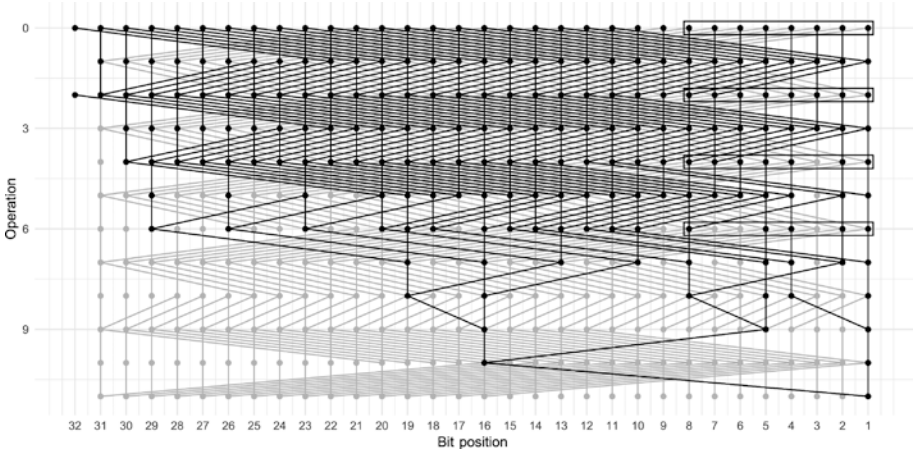

Figure 6-13. *Dependencies for the least significant bit in one-at-a-time hashing*

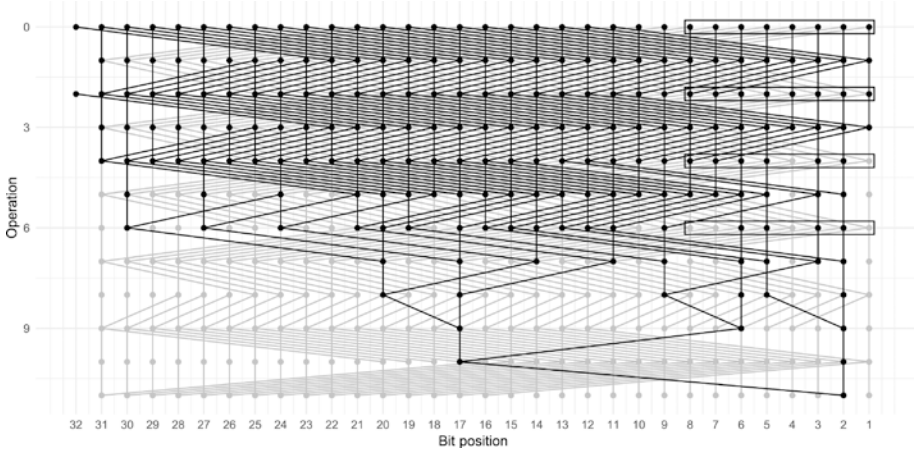

Figure 6-14. *Dependencies for the second-least significant bit in one-at-a-time hashing*

Based on these observations, you would expect that this hash function performs better on your test data, which seems to be the case, as shown in Figure 6-15 and Figure 6-16. Since the output bits depend on all the bits in the first byte and only some of the bits in the last byte, you might expect

that putting the informative byte in your test data as the first byte would be slightly better, but both options seem to work well.

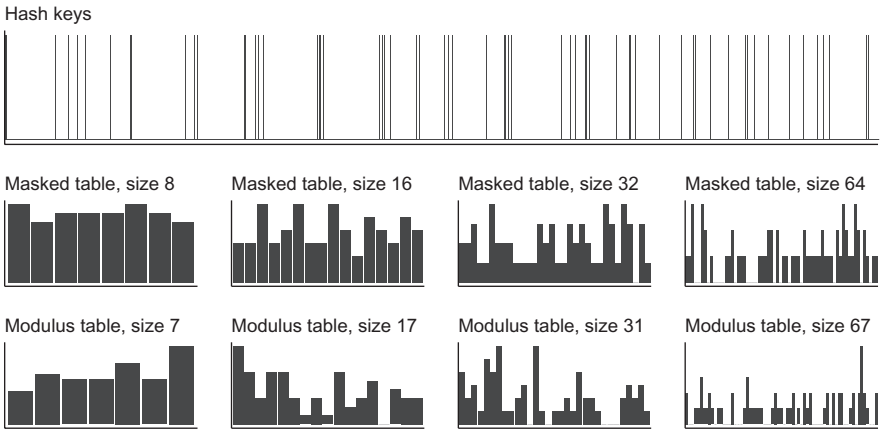

Figure 6-15. *One-at-a-time hashing adding the informative byte in the first combine operation*

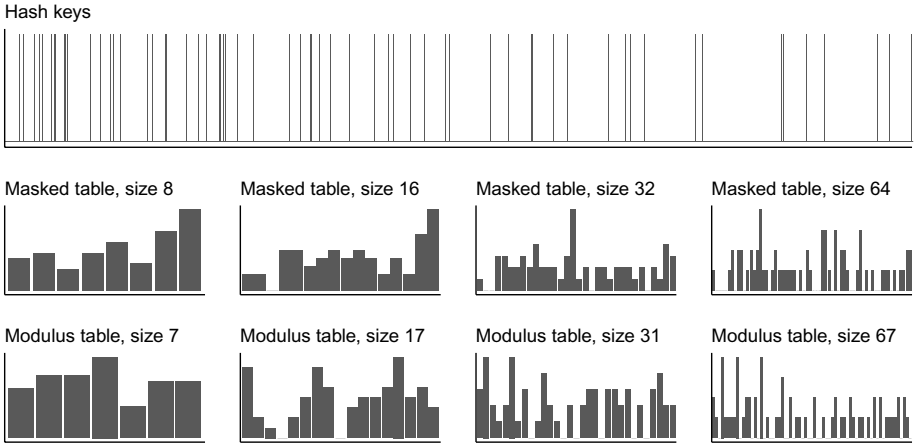

Figure 6-16. *One-at-a-time hashing adding the informative byte in the last combine operation*

You can also see that the output bits depend on combinations of the initial state and the input, suggesting that poor performance on adversarial data can be improved by changing the initial state.

The bit patterns for the input and output of this hash function are shown in Figure 6-17 and Figure 6-18. You can see that you propagate some of the variation in the input to the least-significant bits.

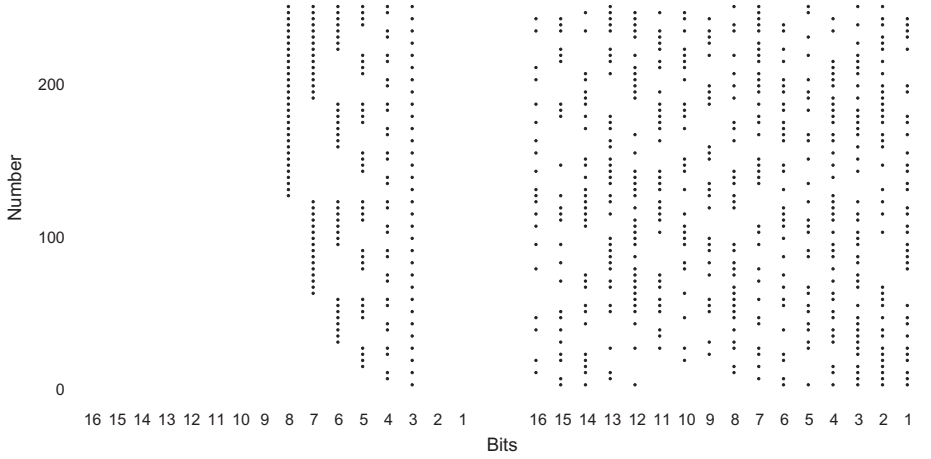

Figure 6-17. *Input and output bit patterns for the one-at-a-time hash function when the input is least-significant-byte first*

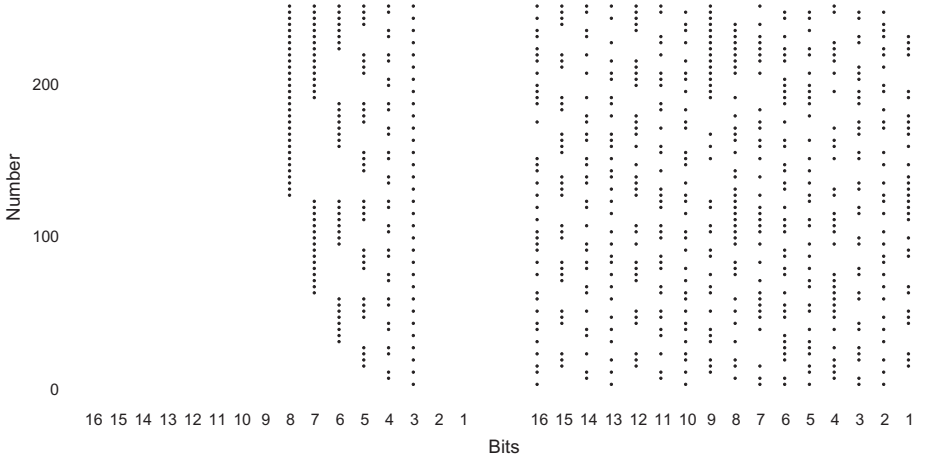

Figure 6-18. *Input and output bit patterns for the one-at-a-time hash function when the input is least-significant-byte last*

The initial state affects the hash keys when you use rotating hashing, but it will not change the two least-significant bits, which are constant in the example input. For one-at-a-time hashing, these bits *do* vary with the initial state; see Figure 6-19, Figure 6-20, Figure 6-21, and Figure 6-22. This provides some hope that, if the hash function performs poorly on specific data, you can change the initial state and get better performance.

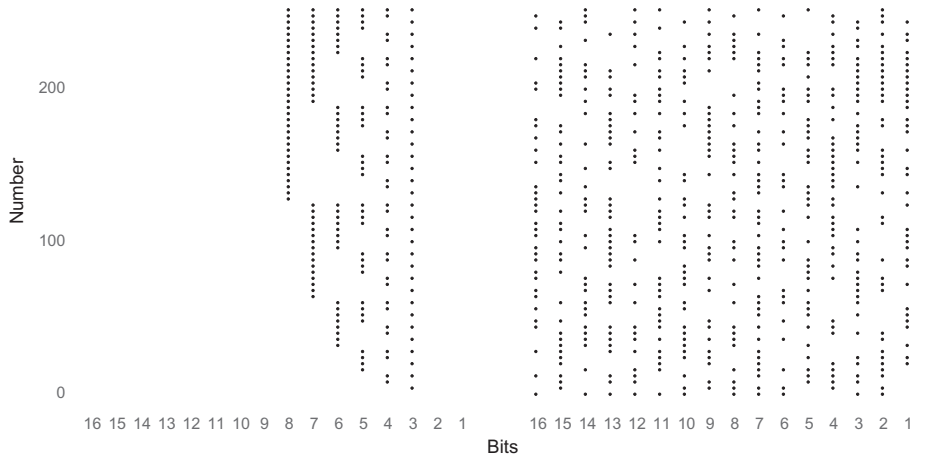

Figure 6-19. *Input and output bit patterns for the one-at-a-time hash function when the input is least-significant-byte first and initial state is set to 1*

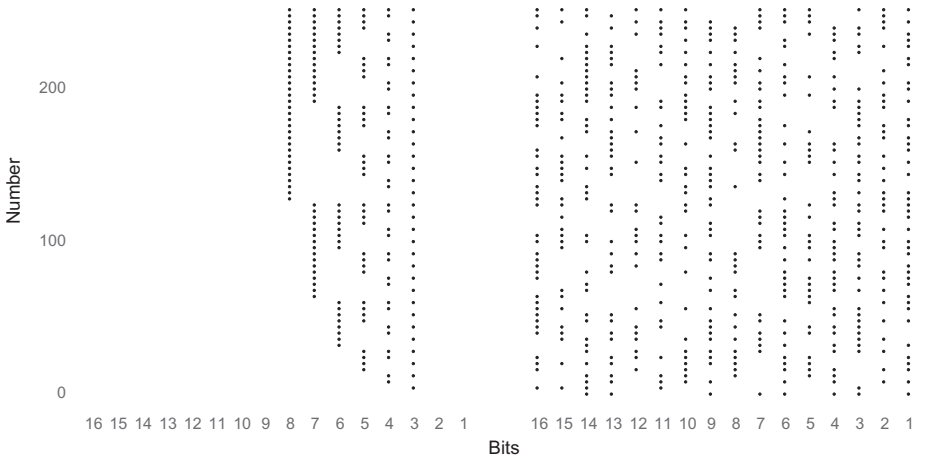

Figure 6-20. *Input and output bit patterns for the one-at-a-time hash function when the input is least-significant-byte first and initial state is set to 0x9e3779b9*

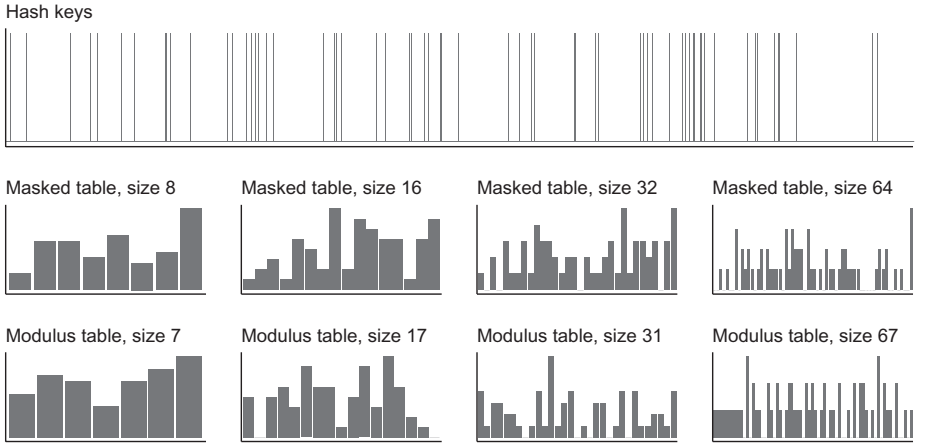

Figure 6-21. *One-at-a-time hashing adding the informative byte in the last combine operation and the initial state is set to 1*

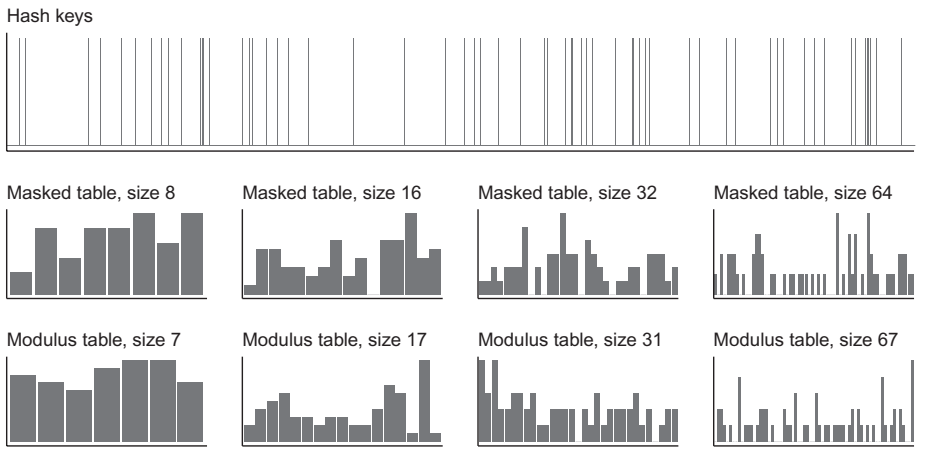

Figure 6-22. *One-at-a-time hashing adding the informative byte in the last combine operation and the initial state is set to 0x9e3779b9*

Jenkins Hashing

The Jenkins' `loopup2` function operates on full computer words and looks as follows:

```
uint32_t jenkins_hash(uint32_t state, uint32_t input)
{
    uint32_t a, b; a = b = 0x9e3779b9;
    uint32_t c = state;

    // combine
    a += input;

    // mix
    a -= b; a -= c; a ^= (c>>13);
    b -= c; b -= a; b ^= (a<<8);
    c -= a; c -= b; c ^= (b>>13);
    a -= b; a -= c; a ^= (c>>12);
    b -= c; b -= a; b ^= (a<<16);
```

```
    c -= a; c -= b; c ^= (b>>5);
    a -= b; a -= c; a ^= (c>>3);
    b -= c; b -= a; b ^= (a<<10);
    c -= a; c -= b; c ^= (b>>15);
    return c;
}
```

This hash function uses more operations than one-at-a-time. It uses one operation for combining and 9 × 4 on mixing, so a total of 37. For larger keys, however, you can operate on data in chunks of 12 bytes, where you can combine 12 bytes in 3 operations and still mix in 36 operations to get a performance of $36/12n = 3n$ operations for keys of n bytes. The one-at-a-time function will use $2n - 1$ operations for combining, $20n$ for mixing, and 6 for the final mix, with a total of $40n + 5$ operations. You will consider hashing variable length keys later in this chapter.

In this implementation, you set the variable `c` to the initial state, but in reality, all three variables, `a`, `b`, and `c`, should be considered the state of the function. When you hash more than a single word, all three variables move the state from one word to the next.

A plot of how individual bits move through this function's mixing step gets very complicated and does not provide much insight into the function. You can, however, plot the input and output bit-patterns; see Figure 6-23 and Figure 6-24 and the corresponding hash table performance in Figure 6-25 and Figure 6-26.

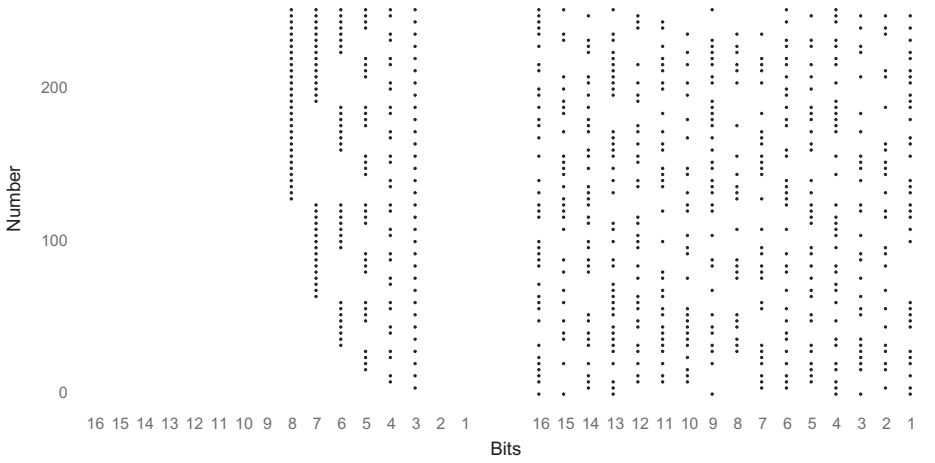

Figure 6-23. *Input and output bit patterns for Jenkins' `lookup2` hash function when the initial state is set to 0*

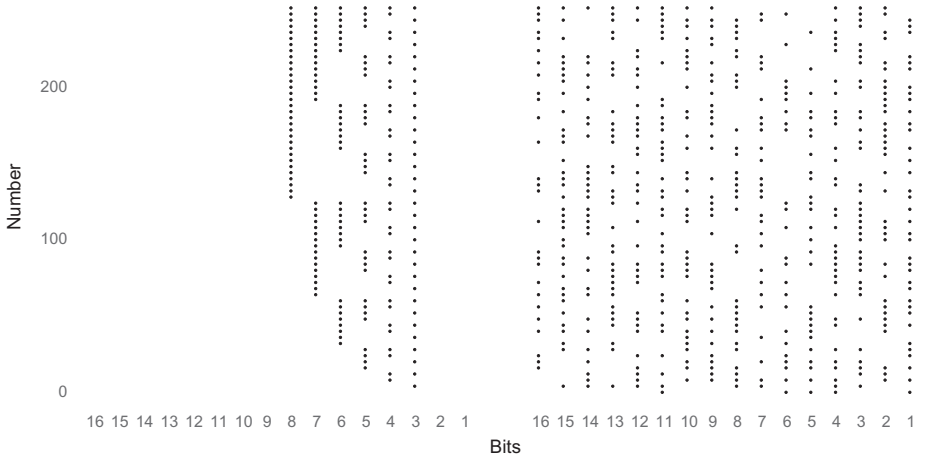

Figure 6-24. *Input and output bit patterns for Jenkins' `lookup2` hash function when the initial state is set to 0x9e3779b9*

The performance of this function on the test data is shown in Figure 6-25 and Figure 6-26 for two different initial states. You get a good spread for either initial state, and the difference between using one state and another is apparent.

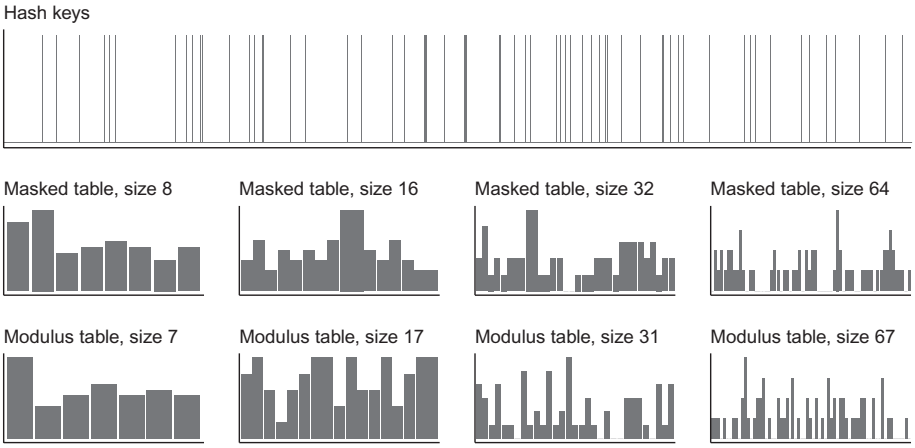

Figure 6-25. *Jenkins' `lookup2` hashing with initial state 0*

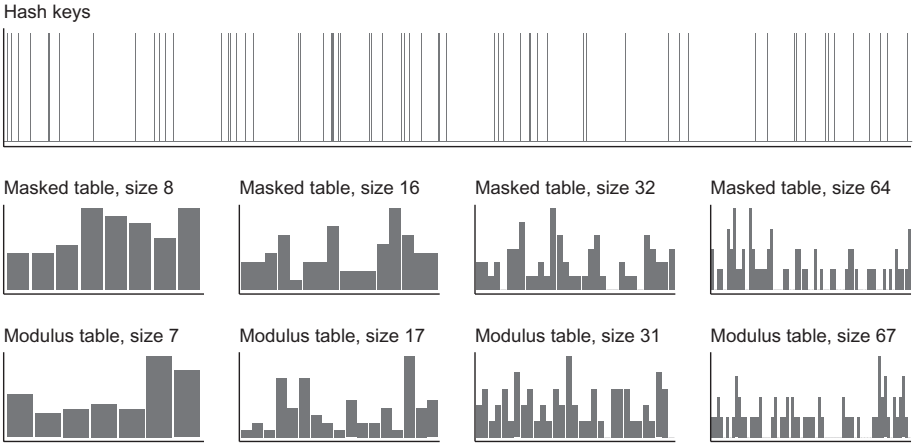

Figure 6-26. *Jenkins' `lookup2` hashing with initial state 0x9e3779b9*

A successor, `lookup3`, is more complex but also faster on larger input data than what you consider here. It is a good choice for hashing entire files. For a hash table, however, lookup2 is a good choice and more straightforward to implement.

Figure 6-27 shows the performance of the different hash functions as implemented above. The time measures have been normalized, so they are relative to the mean of the identity function. As expected, the complex functions are slower than the simplest functions, with the Jenkins function about a factor ten slower than the identity.

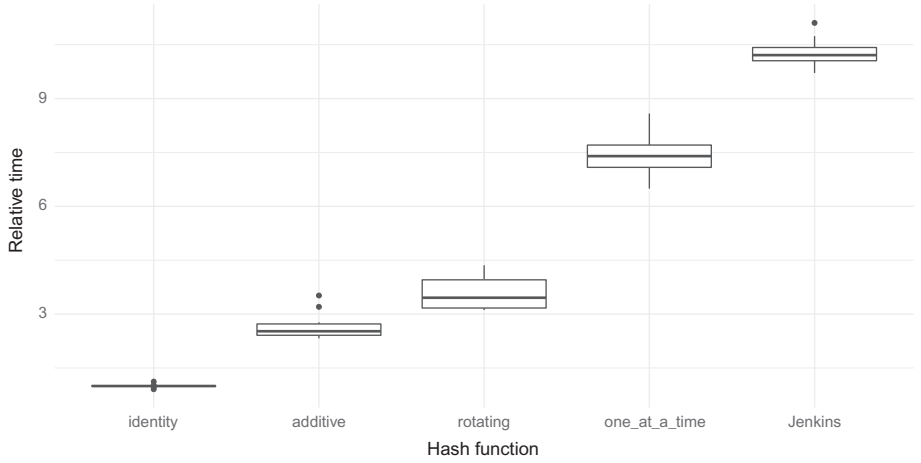

Figure 6-27. *Hash function speed (normalized by the mean performance of the identity function)*

Hashing Strings of Bytes

There is not much difference between hashing a single computer word and a string of bytes of variable length except for a loop. You do not have to worry about the endianness of byte keys since byte keys come in the same order on all software. You can, of course, iterate through the bytes in any order, but there is less reason to worry about it since you would assume that all bytes in such keys would carry information.

You include the length of the key in the signature of hash functions on byte keys. For C strings, you can exploit that these are null terminated, but this will only work when the keys *are* strings. It will not work if you serialize a general data structure and then hash it. For the Jenkins hash function, you also need to know the length of the input to handle the input 12 bytes at a time.

The first three functions, `additive_hash`, `rotating_hash`, and `one_at_a_time_hash`, are easy to translate into versions that iterate over a sequence of bytes:

```
uint32_t additive_hash(uint32_t state, char *input, int len)
{
    uint32_t hash = state;
    for (int i = 0; i < len; i++) {
        // combine
        hash += input[i];
    }
    return hash;
}

#define rot(x,k) (((x)<<(k)) | ((x)>>(32-(k))))
uint32_t rotating_hash(uint32_t state, char *input, int len)
{
    uint32_t hash = state;
    for (int i = 0; i < len; i++) {
```

```
        //     mix            combine
        hash += rot(hash, 4) ^ input[i];
    }
    return hash;
}
uint32_t one_at_a_time_hash(uint32_t state, char *input, int len)
{
    uint32_t hash = state;
    for (int i = 0; i < len; i++) {
        // combine
        hash += input[i];
        // mix
        hash += (hash << 10); hash ^= (hash >> 6);
    }

    // final mix
    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);

    return hash;
}
```

The `jenkins_hash` function takes a little more work since it handles 12 bytes at a time. It reads these into the three state variables, `a`, `b`, and `c`, when there are 12 bytes left, and when there are less than 12 bytes, it reads in as many as it can using a `switch` statement:

```
#define mix(a,b,c) \
{ \
    a -= b; a -= c; a ^= (c>>13); \
    b -= c; b -= a; b ^= (a<<8); \
    c -= a; c -= b; c ^= (b>>13); \
```

```
    a -= b; a -= c; a ^= (c>>12); \
    b -= c; b -= a; b ^= (a<<16); \
    c -= a; c -= b; c ^= (b>>5); \
    a -= b; a -= c; a ^= (c>>3); \
    b -= c; b -= a; b ^= (a<<10); \
    c -= a; c -= b; c ^= (b>>15); \
}
uint32_t jenkins_hash(uint32_t state, char *input, int len)
{
    uint32_t a, b; a = b = 0x9e3779b9;
    uint32_t c = state;
    int k = 0;

    // handle most of the key
    while (len >= 12)
    {
        a += *((uint32_t*)input);
        b += *((uint32_t*)input + 4);
        c += *((uint32_t*)input + 8);
        mix(a,b,c);
        input += 12;
        len -= 12;
    }

    // handle the last 11 bytes
    c += len;
    switch(len) // all the case statements fall through
    {
        case 11: c += input[10] << 24;
        case 10: c += input[9] << 16;
        case 9 : c += input[8] << 8;
        case 8 : b += input[7] << 24;
```

```
            case 7 : b += input[6] << 16;
            case 6 : b += input[5] << 8;
            case 5 : b += input[4];
            case 4 : a += input[3] << 24;
            case 3 : a += input[2] << 16;
            case 2 : a += input[1] << 8;
            case 1 : a += input[0];
                // case 0: nothing left to add
        }
        mix(a,b,c);

        return c;
}
```

Figure 6-28, Figure 6-29, Figure 6-30, and Figure 6-31 show the result of the four hash functions where each word in the poem *The Walrus and the Carpenter* (taken from www.poetryfoundation.org/poems/43914/the-walrus-and-the-carpenter-56d222cbc80a9) is hashed. All functions work well on these words.

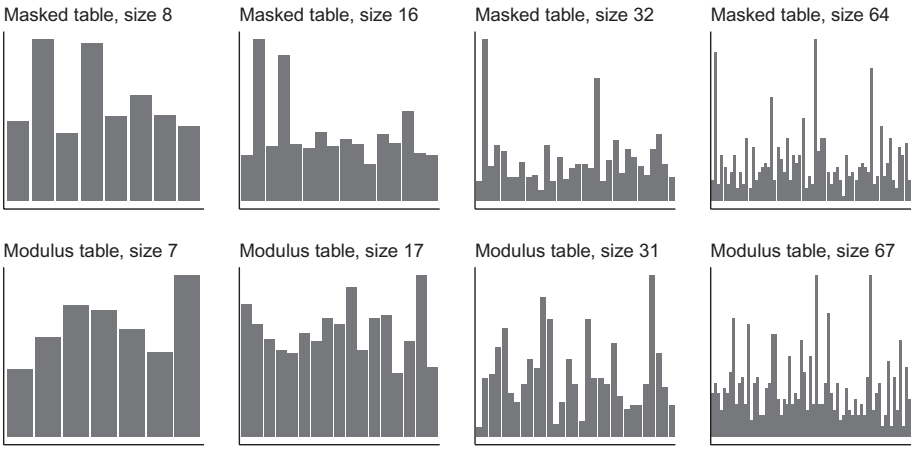

Figure 6-28. *Hashing words using additive hashing*

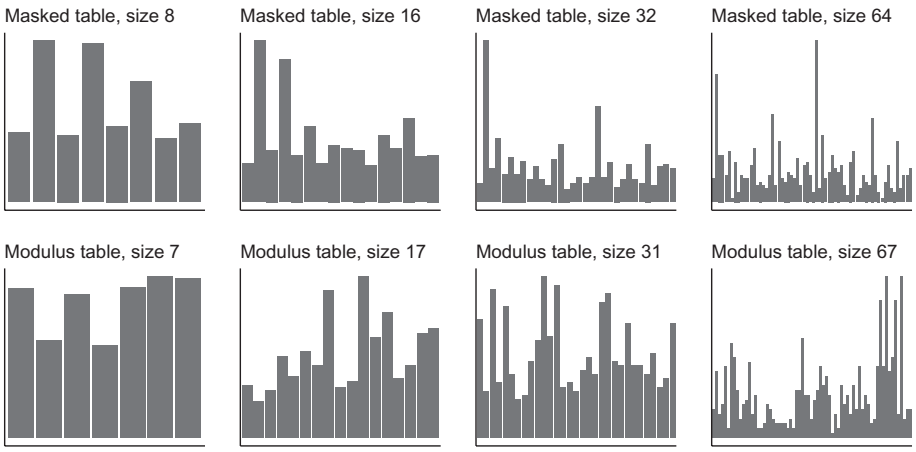

Figure 6-29. *Hashing words using rotating hashing*

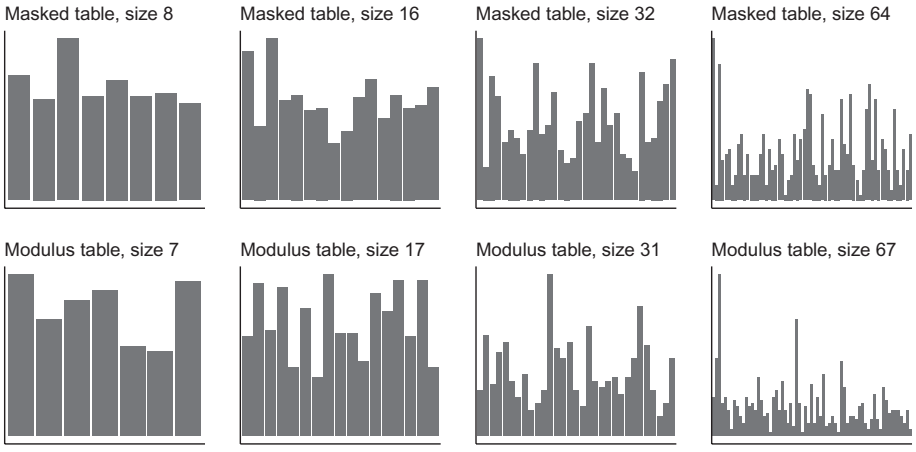

Figure 6-30. *Hashing words using one-at-a-time hashing*

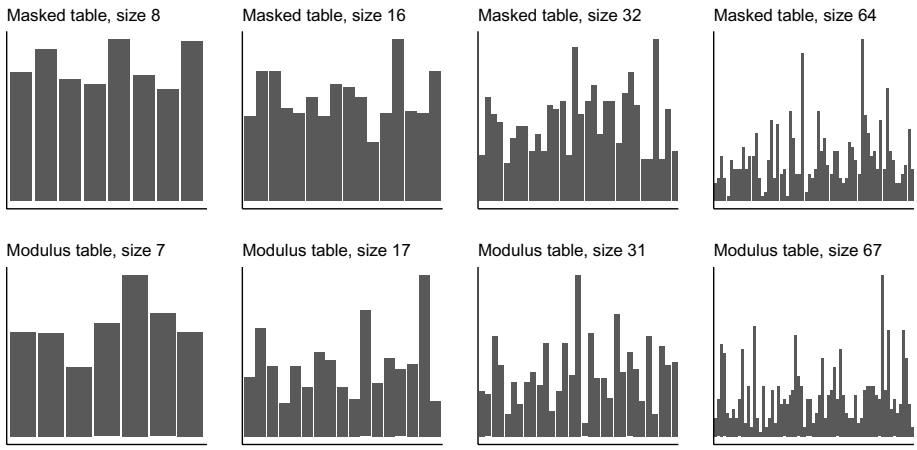

Figure 6-31. *Hashing words using Jenkins' hashing*

The functions you have seen in this chapter are fast to evaluate and widely used, but they do guarantee that hash keys are evenly distributed. In general, any fixed hash function, h, cannot guarantee that it maps all keys uniformly over the range $[m]$ for all sets of keys. After all, if keys are taken from N possible values and put into m bins, then h must map $N = m$ keys to at least one bin. If an adversary knew which hash function you would use, and could pick the keys to give you the worst performance, he could choose the keys such that you get the most collisions possible. Randomized algorithms avoid adversarial scenarios by adding stochasticity into the analysis. For hashing, you can pick random functions h.

The adversary might know from which family of functions you sample h, but not which function you will use. You do not use worst-case running time in the analysis of these algorithms; the worst case would be the same as if the adversary knew your hash function. Instead, you consider the expected running time over the distribution of functions.

Rehashing the functions you have seen so far do not give you sufficient guarantees to use them in a randomized algorithm analysis. You do not know how different two random choices of a function will be when you choose different function parameters. Universal families of hash functions *do* give you guarantees. This is the topic of Chapter 7.

CHAPTER 7

Universal Hashing

Generally, you cannot assume that your application can produce uniformly distributed keys; the hash functions in Chapter 6 are only heuristics. They make no guarantees about the results of hashing application keys and thus risk pathological cases where operations are linear rather than constant.

Since you cannot make assumptions about your hash keys, there is another technique you can employ: randomizing the hash *functions*. Instead of using a fixed hash function that might be sensitive to pathological keys, you can use a family of functions and sample from it. You can rely on random functions to give you expected constant-time operations. The family of functions needs to satisfy specific properties to provide you with this. You need them to be so-called *universal*, and it is conditional on them being universal that you get guarantees for the expected running times. Of course, expected running times are not worst-case running times; you only see the expected performance on average. You can still risk pathological cases. If that happens, however, you can sample new functions. If you resample functions sufficiently often, you will see the average performance over a long run of table operations.

Because universal hashing is heavily based on probability theory, this chapter is more mathematical than the previous chapters. It is mainly concerned with how to construct hash function families to implement universal hashing, and not with proving the probabilistic expectations results.

T. Mailund, *The Joys of Hashing*, https://doi.org/10.1007/978-1-4842-4066-3_7

As before, n refers to the number of keys that you insert into a table, m the size of the table, and $\alpha = n/m$ the load of the table. I use $x_1, \ldots, x_n$ to denote keys from the application universe and $y_1, \ldots, y_n$ to denote hash keys in the range $[m]$. I use $\mathbf{1}$ as the indicator function; for example, $\mathbf{1}_{\text{event}}$ is 1 when the event occurs and 0 when the event does not occur.

Uniformly Distributed Keys

To motivate universal hashing, let's first revisit random keys. Consider chained hashing in a case with load $\alpha < 1$, and consider N operations on the table. Let O_i be operation number i, $h(x_i) = y_i$ the keys involved in operation O_i, S_i the set of keys in the table after operation $i - 1$, and $T(O_i)$ the time it takes to perform operation O_i.

If the hash keys $y_1, \ldots, y_n$ are independent and uniformly distributed, you can show that the expected time for each operation is amortized constant time. Consider the expected running time of the N operations:

$$\mathbf{E}\left[\sum_{i=1}^{N} T(O_i)\right]$$

By linearity of expectation, you have

$$\mathbf{E}\left[\sum_{i=1}^{N} T(O_i)\right] = \sum_{i=1}^{N} \mathbf{E}\left[T(O_i)\right]$$

Since the cost of O_i is the number of keys in the bin y_i maps to, you get

$$\begin{aligned}\mathbf{E}\left[T\left(O_i\right)\right] &= 1+\mathbf{E}\left[\#\left\{y \in S_i \middle| y = y_i\right\}\right] \\ &= 1+\mathbf{E}\left[\sum_{y \in S_i} 1\left(y = y_i\right)\right] \\ &= 1+\sum_{y \in S_i} \mathbf{E}\left[1\left(y = y_i\right)\right] \\ &\leq 1+1+\sum_{y \in S_i,\, y \neq y_i} \Pr\left(1_{y=y_i}\right) \\ &\leq 1+1+m \cdot \frac{1}{m} = 3\end{aligned}$$

In the last step, you see that $|S_i|$ must be less than m when the load is less than 1 and that the keys are uniform so Pr $(y_i = y_j) = 1/m$ for $y_i \neq y_j$.

Universal Hashing

You cannot assume that keys are random because they depend on the application. Instead, you can sample random hash functions from a family of functions, H. In the proof above, you didn't need to know that the keys were independent and uniformly distributed; you only needed Pr $(y_i = y_j) = 1/m$.

You can say that a family of hash functions, H, is *universal* if

$$\Pr\left(h\left(x_i\right) = h\left(x_j\right)\right) \leq \frac{1}{m}$$

when $x_i \neq x_j$ and h is chosen at random from H.

To get amortized constant time operations in a chained hash table with a load less than 1, you only need the family of hash functions to be universal. You do not need the hash f unction to map application keys to uniformly distributed hash keys. Universality is also sufficient to show that the expected amortized time for each operation is n/m when the load is larger than 1.

A family of hash functions H is *nearly universal* if

$$\Pr\left(h\left(x_i\right)=h\left(x_j\right)\right)\leq\frac{k}{m}$$

for some constant k when $x_i \neq x_j$ and h is chosen at random from H.

You can repeat the proof from above with nearly universal hash functions and still get constant time operations. The cost will be bounded by 2 + k instead of 3.

Stronger Universal Families

A universal family of hash functions is far from giving you uniformly distributed hash keys. If you have a family of hash functions that genuinely give you random hash keys, then for any n application keys $x_1, \ldots, x_n$ and hash keys $y_1, \ldots, y_n$ (which can be selected before you sample the hash function, h) you will have

$$\Pr\left(h\left(x_1\right)=y_1,\ldots,h\left(x_n\right)=y_n\right)=\frac{1}{m}\cdot\frac{1}{m}\cdot\ldots\cdot\frac{1}{m}=1/m^n$$

If it holds for any number of keys, n is a very strong property of the family of functions, especially considering that you have to create H and be able to sample from it. In general, you cannot sample functions entirely at random. However, you can create and sample from function families with weaker properties that are still stronger than universal families.

A family of hash functions is *k-independent* if for any k fixed application and hash keys, $x_1, \ldots, x_k$ and $y_1, \ldots, y_k$

$$\Pr\left(h\left(x_1\right)=y_1,\ldots,h\left(x_k\right)=y_k\right)=1/m^k$$

Families that are 2-independent are also called *pairwise independent* or *strong universal*. Pairwise independent families are also universal, but

universal families are not necessarily pairwise independent.
Any k-independent family is also k'-independent for $k' < k$.

Binning Hash Keys

When you map from application keys to hash keys, it is convenient to first map the keys to a large set, $[N]$, and then bin them in $m < N$ bins; for example, you map the hash keys from the large range down to the smaller range.

If you can create a family that is k-independent on the larger range, you also need it to be k-independent on the smaller range. This property is not true for universal functions, but it is for strong universal families if m divides N.[1] For example, if N is a power of two and the range $[m]$ is picked from the lower bits of keys in the range $[m]$, then k-independent families remain k-independent families; so if $N = 2^L$ and $m = 2^{L'}$, $L' < L$ and h is a k-independent family on $[N]$, then $h'(x) = h(x) \bmod m$ is a k-independent family on $[m]$.

For $x_1, \ldots, x_n$ distinct application keys there are $n(n-1)/2$ pairs of keys. A *collision* occurs when $h(x_i) = h(x_j)$ for $i \neq j$. Let X be the number of collisions. The expected number of collisions is

$$\begin{aligned}\mathbf{E}[X] &= \mathbf{E}\left[\sum_{i \neq j} 1_{h(x_i)=h(x_j)}\right] \\ &= \sum_{i \neq j} \Pr\left(h(x_i) = h(x_j)\right) \\ &= \frac{n(n-1)}{2} \Pr\left(h(x_i) = h(x_j)\right) \\ &= \frac{n(n-1)}{2m}.\end{aligned}$$

[1] If m does not divide N, you cannot make universal families. You simply cannot get the same number of keys mapped to each bin. If N is much larger than m, however, you get sufficiently close enough that it doesn't matter in practice.

So if $m \in O(n)$ then $\mathbf{E}[X] \in O(n)$. Also, if $m = n^2$ then $\mathbf{E}[X] = 1/2 - 1/2n < 1/2$, thus the expected number of collisions is less than one half. Furthermore, since the probability of no collisions is $\Pr(X = 0) = 1 - \Pr(X > 0)$ and

$$\Pr(X > 0) = \sum_{x>1} \Pr(X = x) < \sum_{x>1} x \cdot \Pr(X = x) = \mathbf{E}[X] < \frac{1}{2}$$

(i.e. the probability of more than one collision is less than one half), then the probability of no collisions is more than one half. If you pick hash functions at random, then you expect to get a function that gives you no collisions on the second pick.

Requiring that $m \in O(n^2)$, however, means that you must spend time $O(n^2)$ to initialize and resize tables. That is a high price to pay unless you expect to do more than $O(n^2)$ operations on a table (while still bounding the number of keys that are in the table to $O(n)$).

Expecting zero collisions with high probability is a strong requirement. If you allow collisions and resolve them using the strategies you saw in Chapter 3, you can still get expected constant time operations, although you must make assumptions about k-independence for open addressing.

Collision Resolution Strategies

Using chained hashing, a nearly universal family will give you expected constant time operations, as you saw above. You cannot guarantee this with open addressing unless you make stronger assumptions about the family of functions.

With *double hashing*, $h(x) = h_1(x) + i \cdot h_2(x)$, you get constant time operations if both functions are drawn from 2-independent families (Bradford and Katehakis, 2007).

For linear probing, a 5-independent family is needed for expected constant time operations (Pagh, Pagh, and Ruzic, 2011); with 5-wise independence the expected probe length is $O((1-\alpha)^{-5/2})$. For k-independence, $k < 5$, there exist function families that result in logarithmic length probe sequences (Patrashcu and Thorup, 2016). In general, the expected number of operations to query a table or construct a table with n elements, as a function of k, are shown in Table 7-1.

Table 7-1. *Worst-case operation time for k-independence with k from 2 to 5*

Independence	2	3	4	5
Query:	$\ominus(\sqrt{n})$	$\ominus(\log n)$	$\ominus(\log n)$	$\ominus(1)$
Construction:	$\ominus(n \log n)$	$\ominus(n \log n)$	$\ominus(n)$	$\ominus(n)$

The results that require 5-independence for constant time operations *guarantees* expected constant time as long as the function family is 5-independent. The result that 4-independent families do not have this property only shows that *some* 4-independent families do not guarantee constant time operations. Some families can have $k < 5$ and still give expected constant time operations. For example, 3-independent *tabulation hashing* does (Patrashcu and Thorup, 2012). You will consider tabulation hashing later in this chapter.

Constructing Universal Families

All these probabilistic properties you get from universal hashing are only of interest if you can create function families with these properties, and preferably functions that are fast to evaluate. You can do this in multiple ways.

Nearly Universal Families

For constant time operations in chained hashing, nearly universal functions suffice. Dietzfelbinger et al. (1997) showed that if $N = 2^p$ and $m = 2^q$, then function

$$h(x) = \left(ax \bmod 2^p\right) / 2^{p-q}$$

is nearly universal if a is a random odd number $0 < a < 2^p$. They showed that

$$\Pr\left(h(x_1) = h(x_2)\right) \le 1/2^{q-1} = 2/2^q.$$

You need one multiplication, ax, one mask, $a \bmod 2^p$, and one shift, 2^{p-q}, to compute this function. If p is the number of bits in a computer word, then $ax \bmod 2p$ is just one multiplication in p-bit words, since these are multiplication modulo $2p$, and then you avoid masking.

Polynomial Construction for *k*-Independent Families

A common way of creating k-independent hash functions for any k is based on $k - 1$ order polynomials (Wegman and Carter, 1981). The construction works as follows: pick a prime, $p > m$. You can pick this prime to be larger than any m you expect to use in your application. Keep it fixed for the algorithm where you use your table. To sample a function, you pick k random integers in $[p]$, $a_0, \ldots, a_{k-1}$. Your function is

$$h(x) = \left(\sum_{i=1}^{k-1} a_i x^i\right) \bmod p \bmod m$$

Allocating the data you need to represent a polynomial hash function is trivial. If you use 32-bit numbers it is simple as this:

```
malloc(sizeof(uint32_t) * k);
```

Once you have allocated the memory, you can sample functions by picking k random numbers and putting them in this array:

```
void poly_sample(uint32_t *a, int k, uint32_t p)
{
    for (int i = 0; i < k; ++i) {
        a[i] = rand() % p;
    }
}
```

This assumes that you do the mapping into m bins as a separate operation. The mapping will be the same for all hash functions, so it is not specific to universal hashing.

For Mersenne primes, primes on the form $2^l - 1$, you can avoid the modulo operation and replace it with hashing and shifting, as mentioned in Chapter 2:

```
uint64_t mod_Mersenne(uint64_t x, uint8_t s)
{
    uint64_t p = (uint64_t)(1 << s) - 1;
    uint64_t y = (x & p) + (x >> s);
    return (y > p) ? y - p : y;
}

uint32_t poly_hash2_Mersenne(uint32_t x, uint32_t *a, uint8_t s)
{
    uint64_t ax1 = mod_Mersenne((uint64_t)a[1] * (uint64_t)x, s);
    uint32_t y = (uint32_t) mod_Mersenne(a[0] + ax1, s);
    return y;
}
uint32_t poly_hash5_Mersenne(uint32_t x, uint32_t *a, uint8_t s)
{
    // No need for % p for the first value, it will fit
    // in 64-bit.
```

```
    uint64_t x1 = (uint64_t)x;
    uint64_t x2 = mod_Mersenne(x1 * (uint64_t)x, s);
    uint64_t x3 = mod_Mersenne(x2 * (uint64_t)x, s);
    uint64_t x4 = mod_Mersenne(x3 * (uint64_t)x, s);

    uint64_t a0 = (uint64_t)a[0];
    uint64_t ax1 = (uint64_t)a[1] * x1;
    uint64_t ax2 = (uint64_t)a[2] * x2;
    uint64_t ax3 = (uint64_t)a[3] * x3;
    uint64_t ax4 = (uint64_t)a[4] * x4;

    // Since all values fit in 32 bits we
    // can add them in 64 bits without overflow
    uint64_t y = a0 + ax1 + ax2 + ax3 + ax4;
    return (uint32_t) mod_Mersenne(y, s);
}
```

For 32-bit words, you need to do multiplication in 64-bit words, and you need the modulo after each operation to keep them in 64-bit words.

Tabulation Hashing

Tabulation hashing (Carter and Wegman, 1979) is another way to construct a universal family. Tabulation hashing only gives you 3-independence but still provides the expected constant time operations for linear probing.

Tabulation hashing uses a table and has more initialization overhead than the polynomial construction, but it compensates for this with faster evaluation times. It avoids expensive multiplication and modulus operations, and replaces them with table lookups and XOR operations.

Tabulation hashing maps p-bit words ($N = 2^p$) to q-bit words ($m = 2^q$) by splitting application keys into r-bit chunks; there are $t = p/r$ of them. You build a table T with one row for each of the t chunks and 2^r columns. In each of the cells of T, you put a random q-bit number.

For the key x, let x_0 denote the first r bits in x, x_1 the next r bits, and so on until x_{t-1}. Since each of the x_i r-bit chunks can be used to index into an array of length 2^r, you can get a q-bit word from $T[i, x_i]$ for each $i = 0, \ldots, t - 1$. You XOR these together to get the hash key

$$h(x) = T[0, x_0] \oplus T[1, x_1] \oplus \ldots \oplus T[t-1, x_{t-1}]$$

The 2^r number of columns might scare you—exponential numbers always should—but you are working with small r values, which keeps the problem under control.

Indexing into q-bit words requires a lot of bit fiddling, but if you stick to the number of bits available as C data types, you can handle it by casting a pointer, as you will see shortly.

You can treat all tables as bytes when you allocate them. You need to pick an r value. That also defines $t = p/r$ (let's assume that p is always 32). Then, for q, you can pick 8-, 16-, and 32-bit words, corresponding to `uint8_t`, `uint16_t`, and `uint32_t`.

For example, for $r = 2$ and $q = 16$, (i.e. `uint16_t`) you allocate like this:

```
int p = 32;
int r = 2
int q = 16;
int no_cols = (1 << r);
int t = p / r;
no_cols = (1 << r);
bytes = t * no_cols * q / 8;
uint8_t *T8 = malloc(bytes);
```

The table is called T8 to indicate that it contains bytes, `uint8_t`. You always allocate a byte array, but you will cast it to different types for different q.

You can treat the table as an array of 32-bit words and sample like this:

```
void tabulation_sample(uint32_t *start, uint32_t *end)
{
    while (start != end)
        *(start++) = rand();
}
```

Once the byte array is allocated, you cast it to 32-bit numbers and sample into it:

```
        int32_t *T32 = (uint32_t*)T8;
        int32_t *T32_end = (uint32_t*)(T8 + bytes);
        tabulation_sample(T32, T32_end);
```

A straightforward approach to hashing numbers, for example 32-bit numbers using `uint32_t`, is this:

```
uint32_t tabulation_hash(uint32_t x, uint32_t *T, int p, int r)
{
    int t = p / r;
    int no_cols = 1 << r;
    uint32_t r_mask = (1 << r) - 1;

    uint32_t y = 0;
    for (int i = 0; i < t; ++i) {
        y ^= T[i * no_cols + (x & r_mask)];
        x >>= r;
    }

    return y;
}
```

This, however, involves multiplications to compute the indices into T, and more than for the polynomial function for most choices of r. You can, however, fix the indices at compile time if you use a specialized function for each r and q combination. Hashing an $r = 8$ table with $q = 16$ looks like this:

```
uint32_t tabulation_hash_r8_q16(uint32_t x, uint8_t *T)
{
    // these are all known at compile time
    const int r = 8;
    const uint32_t no_cols = 1 << r;
    const uint32_t mask = (1 << r) - 1;

    // q == 16
    uint16_t *T_ = (uint16_t*)T;
    // r == 8 -> t == 4
    uint32_t y = T_[0 * no_cols + (x & mask)]; x >>= r;
    y ^= T_[1 * no_cols + (x & mask)]; x >>= r;
    y ^= T_[2 * no_cols + (x & mask)]; x >>= r;
    y ^= T_[3 * no_cols + (x & mask)];

    return y;
}
```

I have unrolled the loop here to gain more speed.

Different choices of r will have different tradeoffs, but you can specialize functions to any given application or, based on experiments, pick an r that is generally good. If you know what q values you are going to need, you can also fix that. The easiest, however, is to use 32-bit numbers. It will make the tables larger than for smaller q, and the initialization correspondently slower, but you will never use an m larger than q^{32}, so it will always work.

You can also adjust the function as the table grows to larger m, but that will require calling the hash function through a pointer, and computing jump points like that can confuse the CPU's pipelining and slow down the hashing.

Performance Comparison

Figure 7-1 shows the cost of sampling functions both for polynomials and tabulation hashing. The time measurements have been normalized by dividing by the mean of the degree two polynomial computation, which is the fastest.

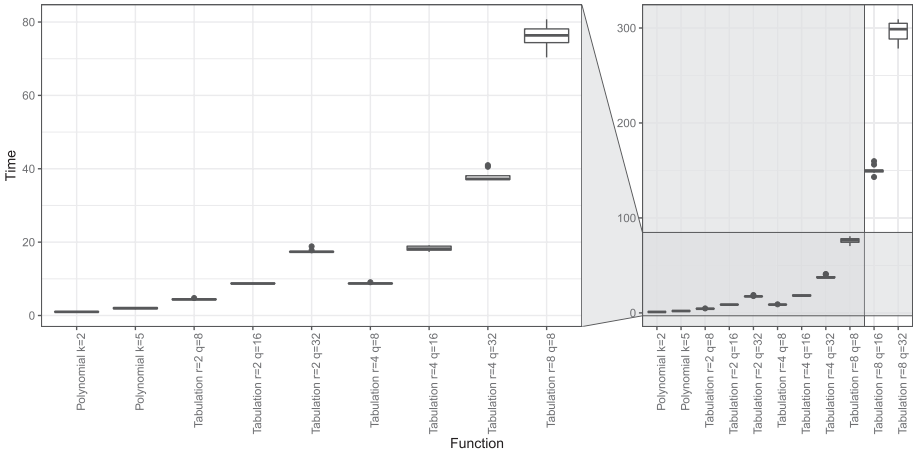

Figure 7-1. *Initialization performance relative to sampling two random integers (Polynomial with k = 2)*

This way, each of the other times is relative to it and shows how much slower they are. The mean times are shown in Table 7-2.

Table 7-2. *Relative initialisation time*

Function	Time
Polynomial k=2	1.00
Polynomial k=5	1.99
Tabulation r=2 q=8	4.45
Tabulation r=2 q=16	8.75

(*continued*)

Table 7-2. (*continued*)

Function	Time
Tabulation r=2 q=32	17.55
Tabulation r=4 q=8	8.76
Tabulation r=4 q=16	18.27
Tabulation r=4 q=32	38.01
Tabulation r=8 q=8	76.18
Tabulation r=8 q=16	150.40
Tabulation r=8 q=32	296.41

There is a larger overhead in filling the tables compared to sample coefficients for the polynomials, and for large r, where the 2^r columns in the table are problematic, this is substantial. If you stick to $q = 32$, to get a value that will work for all choices of m, you have 20 to 40 times the allocation cost. This needs to be compensated for by the speed of the hash functions. Luckily, tabulation hashing is *much* faster than computing polynomials.

Figure 7-2 shows a comparison of the functions, here normalized with the performance of tabulation with $p = 16$ and $r = 8$.

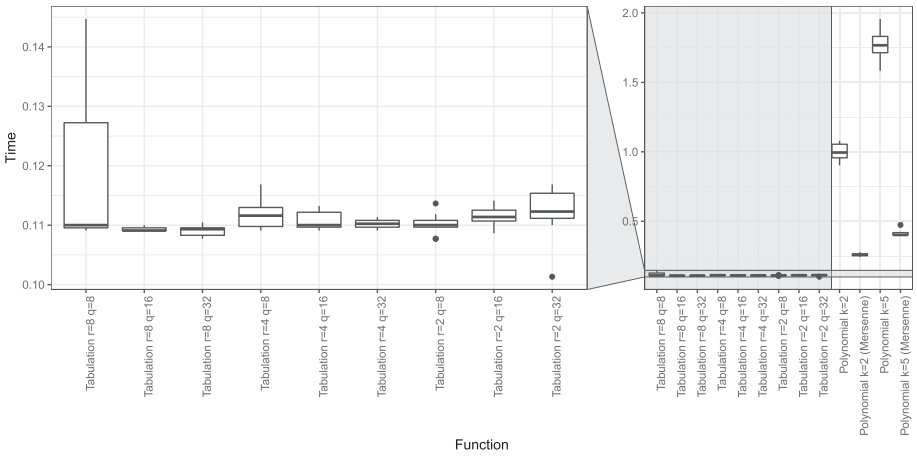

Figure 7-2. *Hashing performance relative the degree two polynomial computation*

Table 7-3 shows the mean of the measurements. The tabulation hash functions are an order of magnitude faster than the degree two polynomial and twice as fast at the Mersenne prime degree two function. For the degree five polynomial, necessary for constant time linear probing, the tabulation functions are four times as fast.

Table 7-3. *Relative function evaluation time*

Function	Mean
Tabulation r=8 q=8	0.118
Tabulation r=8 q=16	0.109
Tabulation r=8 q=32	0.109
Tabulation r=4 q=8	0.112
Tabulation r=4 q=16	0.111
Tabulation r=4 q=32	0.110
Tabulation r=2 q=8	0.110

(*continued*)

Table 7-3. (*continued*)

Function	Mean
Tabulation r=2 q=16	0.111
Tabulation r=2 q=32	0.112
Polynomial k=2	1.00
Polynomial k=2 (Mersenne)	0.259
Polynomial k=5	1.78
Polynomial k=5 (Mersenne)	0.414

Figure 7-3, for comparison with the heuristic hash functions from Chapter 6, shows the $q = 32$ bit tables and the polynomials together with the identity function—a baseline that does nothing—and the Jenkins hashing, the slowest from Chapter 6.

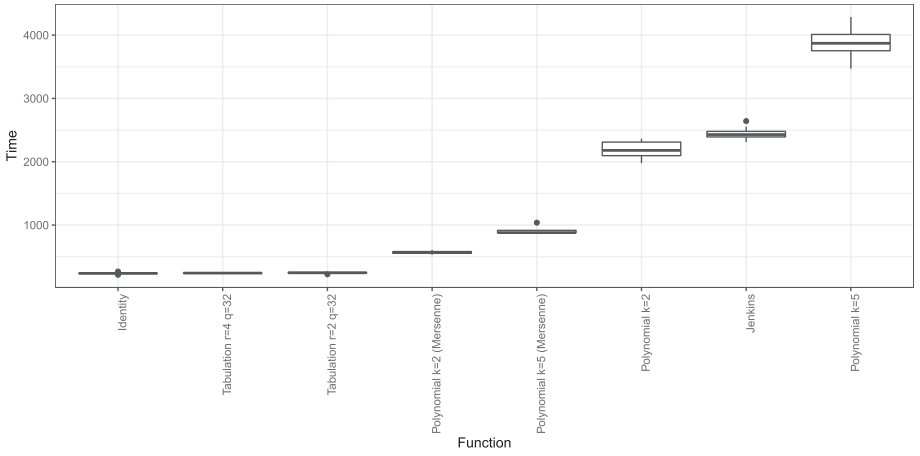

Figure 7-3. *Heuristic hash functions vs. universal hash functions*

It is clear that the universal hashing functions are competitive as long as you use Mersenne primes for the polynomials. They are faster to compute than the Jenkins hashing while providing stronger probabilistic guarantees.

With the large initialization cost for tabulation hashing but faster hashing operations you can consider how many operations you need to do before tabulation hashing outperforms the polynomial method. You can see this in Figure 7-4. Again, the time measures are relative; they have been normalized by the initialization cost for the degree two polynomial. If you do not rehash more often than about every hundredth operation, the tabulation hashing is generally faster than the polynomial.

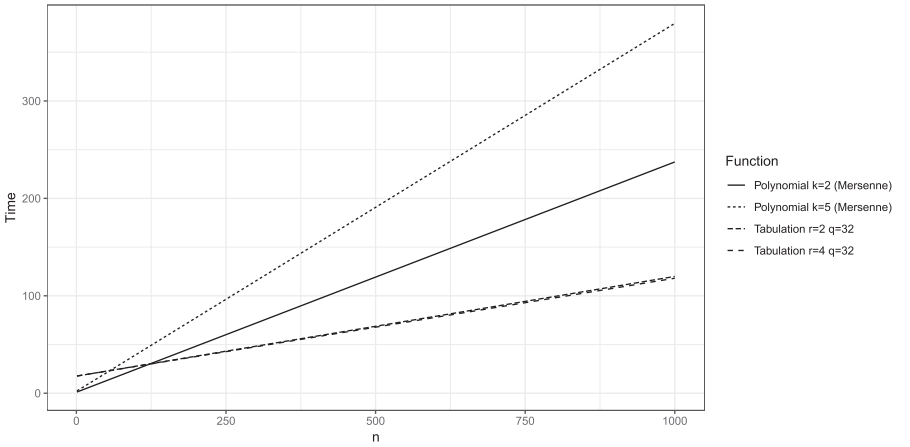

Figure 7-4. *Method performance as a function of n*

Rehashing

Having guarantees on the expected running times does not mean that you have guarantees for any specific choice of hash function that you have sampled, of course. You only get the average behavior over many samples. One technique for getting average behavior is to resample from time to time. If you do this, then one unlucky sample will only affect some operations and will with high probability be replaced by a better choice when you resample.

You cannot rehash too often since rehashing is a linear time operation; you need to move all the keys from the bins where the old hash function

mapped them and to the bins the new hash function is assigning them. If you do not frequently rehash, though, you do not get the average behavior.

If you rehash every time you have spent some $O(n)$ time on hash table operations, then you have amortized the cost of rehashing. What factor you will multiply to n for this depends on the hash function sampling and the cost of moving keys. You can experiment to find a good value for your choices.

To add rehashing to your table, you need a few more items in the structure. You need the hash function information; below you can also see tabulation hashing and some bookkeeping for keeping track of the number of operations you have made when to rehash:

```
struct hash_table {
    struct bin *table;
    unsigned int size;
    unsigned int used;
    unsigned int active;

    // For tabulation hashing.
    uint8_t *T, *T_end;

    float rehash_factor;
    unsigned int probe_limit;
    unsigned int operations_since_rehash;
};
```

In the table constructor, you need to initialize these new entries. Note that the scale factor for when to rehash a parameter is part of the function. You can easily hardcode it. The size is yet again a parameter, but you will want to put a minimum on the size according to the hash function you choose. For tabulation hashing, where the initialization cost is large compared to the polynomial construct, you might want to use a minimum value of 256 (assuming that the size is a power of two here, since you will use masking to map from $[N]$ to $[m]$).

```
struct hash_table *empty_table(unsigned int size, float rehash_
factor)
{
    struct hash_table *table =
        (struct hash_table*)malloc(sizeof(struct hash_table));
    table->table = (struct bin *)malloc(size * sizeof(struct bin));
    struct bin *end = table->table + size;
    for (struct bin *bin = table->table; bin != end; ++bin) {
        bin->is_free = true;
        bin->is_deleted = false;
    }

    table->size = size;
    table->active = table->used = 0;

    // setting up tabulation hashing table
    int p = 32;
    int r = 4;
    int q = 32;
    int no_cols = (1 << r);
    int t = p / r;
    int bytes = t * no_cols * q / 8;
    table->T = malloc(bytes);
    table->T_end = table->T + bytes;
    tabulation_sample((uint32_t*)table->T,
                      (uint32_t*)table->T_end);

    table->rehash_factor = rehash_factor;
    table->probe_limit = rehash_factor * size;
    table->operations_since_rehash = 0;

    return table;
}
```

For the destructor, you need to free the memory that holds the hash function:

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

The rehash code needs to insert elements when moving from one hash function to another. Here the actual insertion is moved into a separate function and you only check whether a rehash is needed in the insert function that users can call. The insertion function also needs to update the rehash bookkeeping. Otherwise, the insertion is the same as before.

```
void insert_key(struct hash_table *table, uint32_t key)
{
    table->operations_since_rehash++;
    if (table->operations_since_rehash > table->probe_limit) {
        rehash(table);
    }
    insert_key_internal(table, key);
    if (table->active > table->size / 2) {
        resize(table, table->size * 2);
    }
}

static void insert_key_internal(struct hash_table *table,
uint32_t key)
{
    uint32_t hash_key = hash(key, table->T);
    uint32_t index;
    for (unsigned int i = 0; i < table->size; ++i) {
```

```
        index = p(hash_key, i, table->size);
        struct bin *bin = & table->table[index];

        if (bin->is_free) {
            bin->key = key;
            bin->is_free = bin->is_deleted = false;

            // we have one more active element
            // and one more unused cell is occupied
            table->active++; table->used++;
            break;
        }

        if (bin->is_deleted) {
            bin->key = key;
            bin->is_free = bin->is_deleted = false;

            // we have one more active element
            // but we do not use more cells since the
            // deleted cell was already used.
            table->active++;
            break;
        }

        if (bin->key == key) {
            return; // nothing to be done
        }
    }
}
```

You need to update the rehash bookkeeping in the contains_key and delete_key functions, but otherwise, they are unchanged:

```
bool contains_key(struct hash_table *table, uint32_t key)
{
    table->operations_since_rehash++;
    if (table->operations_since_rehash > table->probe_limit) {
        rehash(table);
    }

    uint32_t hash_key = hash(key, table->T);

    for (unsigned int i = 0; i < table->size; ++i) {
        unsigned int index = p(hash_key, i, table->size);
        struct bin *bin = & table->table[index];
        if (bin->is_free)
            return false;
        if (!bin->is_deleted && bin->key == key)
            return true;
    }
    return false;
}

void delete_key(struct hash_table *table, uint32_t key)
{
    table->operations_since_rehash++;
    if (table->operations_since_rehash > table->probe_limit) {
        rehash(table);
    }
```

```
    uint32_t hash_key = hash(key, table->T);
    for (unsigned int i = 0; i < table->size; ++i) {
        unsigned int index = p(hash_key, i, table->size);
        struct bin * bin = & table->table[index];

        if (bin->is_free) return;

        if (!bin->is_deleted && bin->key == key) {
            bin->is_deleted = true;
            table->active--;
            break;
        }
    }
    if (table->active < table->size / 8)
        resize(table, table->size / 2);
}
```

Whenever you resize the table, you do all the same work as for rehashing, so you might as well rehash there as well and reset the rehash count:

```
static void resize(struct hash_table *table, unsigned int new_size)
{
    // nothing good comes from tables of size 0
    if (new_size == 0) return;
    // remember the old bins until we have moved them.
    struct bin *old_bins = table->table;
    unsigned int old_size = table->size;

    // Update table so it now contains the new bins (that are empty)
    table->table = (struct bin *)malloc(new_size *
    sizeof(struct bin));
```

```
    struct bin *end = table->table + new_size;
    for (struct bin *bin = table->table; bin != end; ++bin) {
        bin->is_free = true;
        bin->is_deleted = false;
    }
    table->size = new_size;
    table->active = table->used = 0;

    // Update hash function
    tabulation_sample((uint32_t*)table->T,
                      (uint32_t*)table->T_end);

    // Then move the values from the old bins to the new,
    // using the new hash function from the insertion function
    end = old_bins + old_size;
    for (struct bin *bin = old_bins; bin != end; ++bin) {
        if (bin->is_free || bin->is_deleted) continue;
        insert_key_internal(table, bin->key);
    }

    // Update rehash limit
    table->probe_limit = table->rehash_factor * new_size;
    table->operations_since_rehash = 0;
    // finally, free memory for old bins
    free(old_bins);
}
```

Finally, rehashing follows the pattern in the resize function. You allocate a new table, to avoid probes that interfere with old bins when moving keys to new bins, and then you move the keys to the new table.

You need to sample a new function before you move them and update the bookkeeping after.

```
static void rehash(struct hash_table *table)
{
    struct bin *old_bins = table->table;
    table->table = (struct bin *)malloc(table->size *
    sizeof(struct bin));
    struct bin *end = table->table + table->size;
    for (struct bin *bin = table->table; bin != end; ++bin) {
        bin->is_free = true;
        bin->is_deleted = false;
    }
    // Update hash function
    tabulation_sample((uint32_t*)table->T,
                      (uint32_t*)table->T_end);
    table->operations_since_rehash = 0;

    struct bin *old_end = old_bins + table->size;
    for (struct bin *bin = old_bins; bin != old_end; ++bin) {
        if (bin->is_free || bin->is_deleted) continue;
        insert_key_internal(table, bin->key);
    }
    free(old_bins);

    // Update rehash limit
    table->probe_limit = table->rehash_factor * table->size;
    table->operations_since_rehash = 0;
}
```

You can leave the size, used, and active counters alone. They do not change when you rehash.

CHAPTER 8

Conclusions

In this book, you explored the hash table data structure. You learned how to map keys from a large space, in which you assume that keys are uniformly distributed, into a small space of table bins and considered a table's performance as a function of the number of bins vs. how many keys you store in a table. You covered strategies for handling collisions when two or more different keys map to the same bin and the performance consequences of the choice of strategy. You also learned how to dynamically adjust the size of tables to avoid them filling up and incurring high runtime performance penalties as a consequence, while at the same ensuring that you do not allocate tables larger than necessary and thus incur memory penalties as a consequence.

It is a strong assumption that keys are uniformly distributed before you map them to bins, but for natural keys in most applications, where keys might be strings or numbers, it is not true. Generally, it is necessary to first map the application keys into a uniformly distributed space of keys before you can use these "random" keys in a hash table. In the literature, hash functions are often considered functions that map application keys to bins. In this book, you used this map as two or three separate functions. The first step is application-dependent and reduces your data to a number. The (optional) second step scrambles the keys, getting them closer to uniformly distributed. The last step then maps the hash keys to bins. I refer to the first two steps as hash functions but not the third.

T. Mailund, *The Joys of Hashing*, https://doi.org/10.1007/978-1-4842-4066-3_8

Constructing hash functions that create close-to-uniformly-distributed hash keys is a research field in its own right and an essential part of modern cryptography research but beyond the scope of this book. Here, I have chosen to focus mainly on the hash table data structure, and I have covered a few functions for scrambling application hash functions.

This is a first edition of the book, and it resembles a beta release of software. There are likely errors I haven't caught, so I will add an errata section on GitHub for each of the chapter-code repositories, and if the error is an implementation I will update the code there as well.

In each chapter, I focused on a subset of hash table features to prevent unrelated details overshadowing the relevant topic. This means that there is no single implementation of a table that combines them all. You can download full-featured hash tables based on this book at https://github.com/mailund/hash. Please let me know if you discover any errors, either in this book or in the code at GitHub.

Bibliography

Bradford, Philip G., and Michael N. Katehakis. 2007. "A Probabilistic Study on Combinatorial Expanders and Hashing." *SIAM J. Comput.* 317 (1):83–111.

Carter, J. Lawrence, and Mark N. Wegman. 1979. "Universal Classes of Hash Functions." *Journal of Computer and System Sciences* 18 (2):143–54.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition*. 3rd ed. The MIT Press.

Dietzfelbinger, Martin, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. 1997. "A Reliable Randomized Algorithm for the Closest-Pair Problem." *Journal of Algorithms* 25 (1):19–51.

Litwin, Withold. 1980. "Linear Hashing: a new tool for file and table addressing." In *Conference on Very Large Databases*, 212–23.

Pagh, Anna, Rasmus Pagh, and Milan Ruzic. 2011. "Linear Probing with 5-Wise Independence." *SIAM Rev.* 53 (3):547–58.

Patrashcu, Mihai, and Mikkel Thorup. 2012. "The Power of Simple Tabulation Hashing." *J. ACM* 59 (3):14:1–14:50.

———. 2016. "On the k-Independence Required by Linear Probing and Minwise Independence." *ACM Transactions on Algorithms* 12 (1):1–27.

Sedgewick, Robert. 1998. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching, Third Edition*. Pearson Education.

Wegman, Mark N., and J. Lawrence Carter. 1981. "New Hash Functions and Their Use in Authentication and Set Equality." *Journal of Computer and System Sciences* 22 (3):265–79.

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

Index

A

Additive hashing function, 146–147
allocated_tables counter, 89
Allocating and deallocating tables, 8
Amortized operation cost, 82–83
Amortizing resizing costs
  computations, 53
  deletion, 57
  growable array, 51
  hash tables, 52, 56
  memory waste, 55
  running time, 55
  runtime analysis, 57
  shrinking, 55
Application keys, 10, 102, 199

B

Birthday paradox, 14

C

Chained hashing, 104, 121
Chained hash tables, resizing
  code, 59
  constructor code, 58
  delete_key, 58
  insert_key, 58, 60
  memory allocation, 60
  resize function, 60, 61
Collision resolution, 21
  cache efficiency, 45–46
  chaining, 25–27
  linked lists, 22–25
  load factor
    double hashing, 44
    open addressing, 44
    runtime, 45
    time usage, 43–44
  open addressing, 27–31
  performance of hash tables, 47
  probing strategies, 32–34
  strategy, 103
  table size *vs.* time, 46, 47
Collision risk
  birthday paradox, 13–14
  number of insertions, 16
  probability, 15
  sizes of tables, 15
  square approximation, 17
contains_key function, 123

D, E, F

Destructor function, 104, 109
Double hashing, 32–34, 88, 178

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

Dynamic resizing
bookkeeping, 89
chained hashing, 88–89
contains_key, 93
extract table, 92
grow and shrink, 87
growing table, 94–96
hash key, 87, 90
hash table, 86
initialize, table, 91–92
inserting and deleting, 93
keys, 86
merge, 87–88
merge bins, 96, 98
shrinking table, 98–99
split, 87
split counter, 90
subtable index, 90
table size, 87

G

get_previous_link function, 107
Growable array, 50

H, I, J, K

Hash functions, 5
collision resolution, 178–179
random, 175
universal family, 176–177
Hash keys, 102
binning, 177–178
collision, 12
maps values, 4
responsibilities, 3
values map, 1, 2
Hash maps
chained hashing table, 121
hash table implementation, updation, 127
linked lists, 121
open addressing strategy, 132
Hash sets
chained hashing, 104
hash table, updating, 110
linked lists, updating, 105
open addressing hash tables, 114
Hash tables, 1, 5
insertion function, 108
operations, 9
Heuristic hash functions
additive hashing, 146–147
cases, 140, 143–144
components, 144–145
hashing string of bytes, 166–171
hash keys, 141
Jenkins' loopup2, 161–165
one-at-a-time, 152–157, 159–160
properties, 141–143
rotate operation, 148–150, 152
Heuristic hash functions *vs.* universal hash functions, 189

L

Linear hashing, 85
Linear probing, 82, 88
Linked lists, 22–25
list_contains_key, 108
Load factor and runtime performance, 49, 61
  chained hashing, 35–36
  definition, 35
  open addressing hashing
    double hashing, 37, 39
    linear probing, 37
    linked list, 38
    mean probe lengths *vs.* theoretical probe lengths, 42
    probe lengths, 39–41
    size of hash table, 38
    theoretical probe length, 37–38
    theoretical results, 36
  probe operations, 66
  resizing, 65
  theoretical running time, 66–67
  worst-case and average-case, 35
Jenkins' loopup2 function
  bit patterns, 163
  on computer words, 161
  performance, 165

M, N

map data structure, 101
Mapping, hash keys
  bins, 18–20
  indices, 9–11
Masking, 19
Modulus, 19

O

old_tables_mask, 89
One-at-a-time hash function, 159–160
  adding informative byte, 161
  bit patterns, 158–160
  bits affected by first last byte, 155
  dependencies for the least significant bit, 156
  dependencies for the second-least significant bit, 156
  implementation, 152–153
  informative byte, 157
  bits affected by first input byte, 154
Open addressing hash tables, 27–31, 114

P, Q

pop operation, 55
primes_idx variable, 76
Probing strategies, 32–34

R

Rehashing, 190–198
resize function, 111

Resizing
chained hash tables, 57, 59–61
open addressing hash tables, 61–64
table sizes (*see* Table sizes, resizing)
Resizing experiments
amortized analysis, 69
conflict resolution strategies, 68
constructor, 70
contains_key function, 73
cost of resizing *vs.* probing, 75
hash table, 70, 75
insertion and deletion, 71–72
linear time, 68
load factor α, 69
load factor threshold, 75
running time, 72–74
time usage, 69
Resizing function, 64

S

split_count variable, 89, 94, 96
Square approximation, 17
struct bin, 7
struct hash_table, 76
subtable_mask, 89–90

T

Table index, 89–91
Table sizes, resizing
bit masking, 75, 82
growth factor β, 77, 78
insertion cost, 78
insert_key, 77
linear cost, 80
open addressing, 80
optimal value, 82
parameter, β, 75
prime size, 82, 84
running time, 78–79, 81
table of primes, 76
time usage components, 78–79
Tabulation hashing, 179, 182–185

U, V, W, X, Y, Z

Universal hashing construction
hashing performance, 186–190
nearly universal functions, 180
polynomials, 180–182