7 THE HASH TABLE: THE WORKHORSE OF RUBY INTERNALS

Experiment 5-1 showed us how in Ruby 1.9 and 2.0 the `ivptr` member of the `RObject` structure pointed to a simple array of instance variable values. We learned that adding a new value was usually very fast but that Ruby was somewhat slower while saving every third or fourth instance variable because it had to allocate a larger array.

Taking a broader look across Ruby's C source code base, we find that this technique is unusual. Instead, Ruby often uses a data structure called a *hash table*. Unlike the simple array we saw in Experiment 5-1, hash tables can automatically expand to accommodate more values; the client of a hash table doesn't need to worry about how much space is available or about allocating more memory for it.

Among other things, Ruby uses a hash table to hold the data you save in the hash objects you create in your Ruby script. Ruby also saves much of its internal data in hash tables. Every time you create a method or a constant, Ruby inserts a new value in a hash table, and Ruby saves many of the special variables we saw in Experiment 3-2 in hash tables. Additionally, Ruby saves instance variables for generic objects, such as integers or symbols, in hash tables. Thus, the hash table is the workhorse of Ruby internals.

In this chapter I'll begin by explaining how hash tables work: what happens inside the table when you save a new value with a key and what happens when you later retrieve that value using the same key. I'll also explain how hash tables automatically expand to accommodate more values. Finally, we'll look at how hash functions work in Ruby.

ROADMAP

Hash Tables in Ruby

Hash tables are a commonly used, well-known, age-old concept in computer science. They organize values into groups, or *bins*, based on an integer value calculated from each value—a *hash*. When you need to find a value, you can figure out which bin it's in by recalculating its hash value, thus speeding up the search.

Every time you write a method, Ruby creates an entry in a hash table.

Saving a Value in a Hash Table

Figure 7-1 shows a single hash object and its hash table.

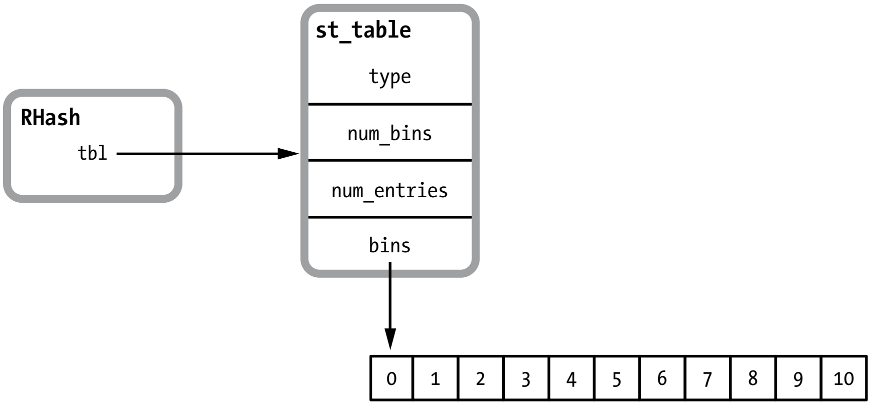

Figure 7-1: A Ruby hash object with an empty hash table

On the left is the `RHash` (short for *Ruby hash*) structure. On the right, you see the hash table used by this hash, represented by the `st_table` structure. This C structure contains the basic information about the hash table, including the number of entries saved in the table, the number of bins, and a pointer to the bins. Each `RHash` structure contains a pointer to a corresponding `st_table` structure. The empty bins on the lower right are there because Ruby 1.8 and 1.9 initially create 11 bins for a new, empty hash. (Ruby 2.0 and later work somewhat differently; see "Hash Optimization in Ruby 2.x" on page 187.)

The best way to understand how a hash table works is by stepping through an example. Suppose I add a new key/value to a hash called `my_hash`:

```
my_hash[:key] = "value"
```

While executing this line of code, Ruby creates a new structure called an `st_table_entry` that it will save into the hash table for `my_hash`, as shown in Figure 7-2.

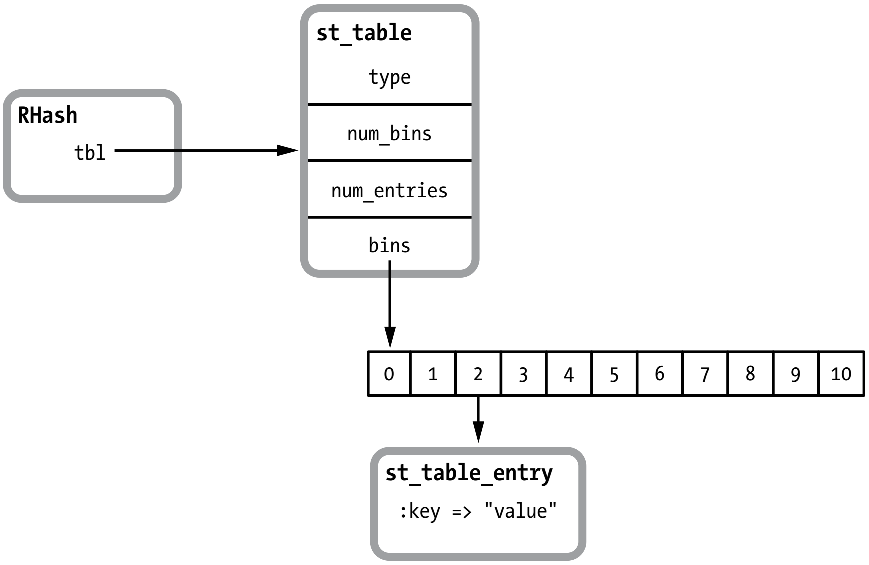

Figure 7-2: A Ruby hash object containing a single value

Here you can see Ruby saved the new key/value pair under the third bucket, number 2. Ruby did this by taking the given key—in this example, the symbol `:key`—and passing it to an internal hash function that returns a pseudorandom integer:

```
some_value = internal_hash_function(:key)
```

Next, Ruby takes the hash value—in this example, `some_value`—and calculates the modulus by the number of bins, which is the remainder after dividing by the number of bins.

```
some_value % 11 = 2
```

NOTE *In Figure 7-2, I assume that the actual hash value for* `:key` *divided by 11 leaves a remainder of 2. Later in this chapter, I'll explore in more detail the hash functions that Ruby actually uses.*

Now let's add a second element to the hash:

```
my_hash[:key2] = "value2"
```

This time let's imagine that the hash value of `:key2` divided by 11 yields a remainder of 5.

```
internal_hash_function(:key2) % 11 = 5
```

Figure 7-3 shows that Ruby places a second `st_table_entry` structure under bin number 5, the sixth bin.

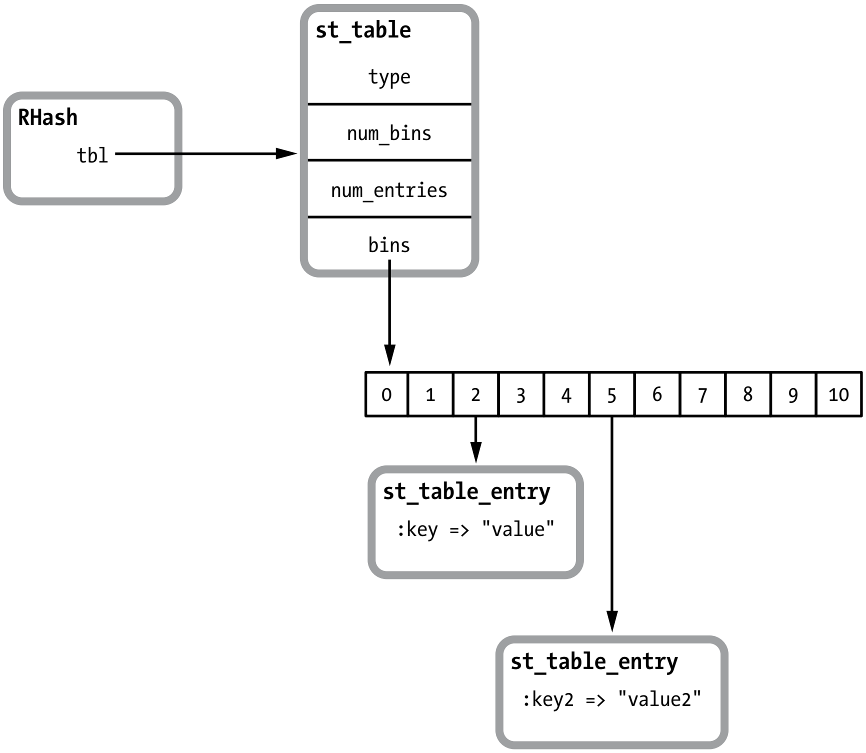

Figure 7-3: A Ruby hash object containing two values

Retrieving a Value from a Hash Table

The benefit of using a hash table becomes clear when you ask Ruby to retrieve the value for a given key. For example:

```
p my_hash[:key]
 => "value"
```

If Ruby had saved all of the keys and values in an array or linked list, it would have to iterate over all the elements in that array or list, looking for `:key`. This might take a very long time, depending on the number of elements. But using a hash table, Ruby can jump straight to the key it needs to find by recalculating the hash value for that key.

To recalculate the hash value for a particular key, Ruby simply calls the hash function again:

```
some_value = internal_hash_function(:key)
```

Then, it redivides the hash value by the number of bins to get the remainder, or the modulus.

```
some_value % 11 = 2
```

At this point, Ruby knows to look in bin number 2 for the entry with the key of `:key`. Ruby can later find the value for `:key2` by repeating the same hash calculation.

```
internal_hash_function(:key2) % 11 = 5
```

NOTE *The C library used by Ruby to implement hash tables was written in the 1980s by Peter Moore from the University of California, Berkeley. Later, it was modified by the Ruby core team. You can find Moore's hash table code in the C code files* st.c *and* include/ruby/st.h. *All of the function and structure names in that code use the naming convention* `st_`*. The definition of the* `RHash` *structure that represents every Ruby Hash object is in the* include/ruby/ruby.h *file. Along with* `RHash`*, this file contains all of the other primary object structures used in the Ruby source code:* `RString`*,* `RArray`*, and so on.*

Experiment 7-1: Retrieving a Value from Hashes of Varying Sizes

This experiment will create hashes of wildly different sizes, from 1 to 1 million elements, and then measure how long it takes to find and return a value from each of these hashes. Listing 7-1 shows the experiment code.

```
  require 'benchmark'

❶ 21.times do |exponent|

    target_key = nil

❷   size = 2**exponent
    hash = {}
❸   (1..size).each do |n|
      index = rand
❹     target_key = index if n > size/2 && target_key.nil?
❺     hash[index] = rand
    end

    GC.disable

    Benchmark.bm do |bench|
      bench.report("retrieving an element
                    from a hash with #{size} elements 10000 times") do
        10000.times do
❻         val = hash[target_key]
        end
      end
    end
  GC.enable
end
```

Listing 7-1: Measuring how long it takes to retrieve an element from hashes of wildly different sizes

At ❶ the outer loop iterates over powers of two, calculating different values for `size` at ❷. These sizes will vary from 1 to about 1 million. Next, the inner loop at ❸ inserts that number of elements into a new empty hash at ❺.

After disabling garbage collection to avoid skewing the results, Experiment 7-1 uses the benchmark library to measure how long it takes to retrieve a value 10,000 times from each hash at ❻. The line of code at ❹ saves one of the random key values to use below at ❻ as `target_key`.

The results in Figure 7-4 show that Ruby can find and return a value from a hash containing over 1 million elements just as fast as it can return one from a small hash.

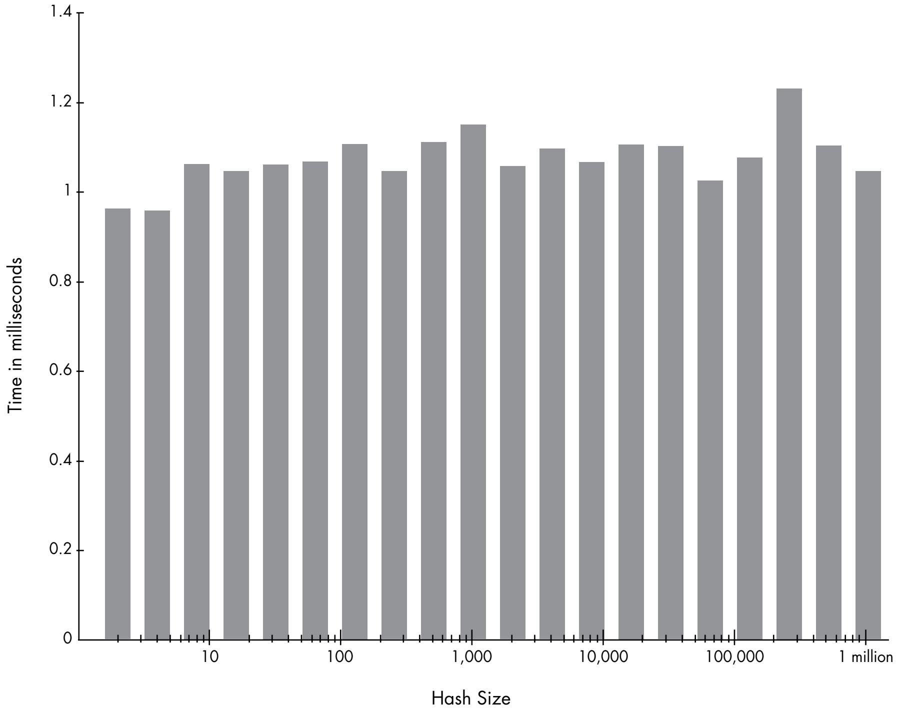

Figure 7-4: Time to retrieve 10,000 values (ms) vs. hash size for Ruby 2.0

Clearly Ruby's hash function is very fast, and once Ruby identifies the bin containing the target key, it can very quickly find the corresponding value and return it. What's remarkable here is that the chart is more or less flat.

How Hash Tables Expand to Accommodate More Values

If there are millions of `st_table_entry` structures, why does distributing them among 11 bins help Ruby search quickly? Because even if the hash function is fast, and even if Ruby distributes the values evenly among the 11 bins in the hash table, Ruby still has to search among almost 100,000 elements in each bin to find the target key if there are 1 million elements overall.

Something else must be going on here. It seems that Ruby must add more bins to the hash table as more and more elements are added. Let's look again at how Ruby's internal hash table code works. Continuing with the example from Figures 7-1 through 7-3, suppose I keep adding more and more elements to my hash.

```
my_hash[:key3] = "value3"
my_hash[:key4] = "value4"
my_hash[:key5] = "value5"
my_hash[:key6] = "value6"
```

As we add more elements, Ruby continues to create more `st_table_entry` structures and add them to different bins.

Hash Collisions

Eventually two or more elements might be saved into the same bin. When this happens, we have a *hash collision*. This means that Ruby is no longer able to uniquely identify and retrieve a key based solely on the hash function.

Figure 7-5 shows the linked list Ruby uses to track the entries in each bin. Each `st_table_entry` structure contains a pointer to the next entry in the same bin. As you add more entries to the hash, the linked lists get longer and longer.

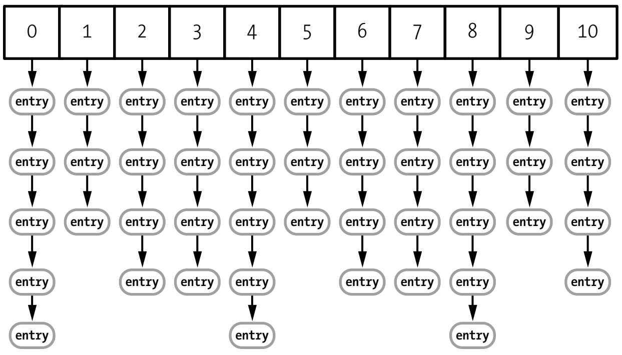

Figure 7-5: A hash table containing 44 values

To retrieve a value, Ruby needs to iterate over the linked list and compare each key with the target. This isn't a serious problem as long as the number of entries in a single bin doesn't grow too large. For integers or symbols, which are typically used as hash keys, this is a simple numerical comparison. However, if you use a more complex data type, such as a custom object, Ruby calls the `eql?` method on the keys to check whether each key in the list is the target. As you might guess, `eql?` returns *true* if two values are equal and *false* if they are not.

Rehashing Entries

To keep these linked lists from growing out of control, Ruby measures the *density*, or average number of entries per bin. In Figure 7-5 you can see that the average number of entries per bin is about 4. This means that the hash value modulus 11 has started to return repeated values for different keys and hash values; thus, there have been some hash collisions.

Once the density exceeds 5, a constant value in Ruby's C source code, Ruby allocates more bins and then *rehashes*, or redistributes, the existing entries across the new bin set. If we keep adding more key/value pairs, for example, Ruby eventually discards the array of 11 bins and allocates an array of 19 bins, as shown in Figure 7-6.

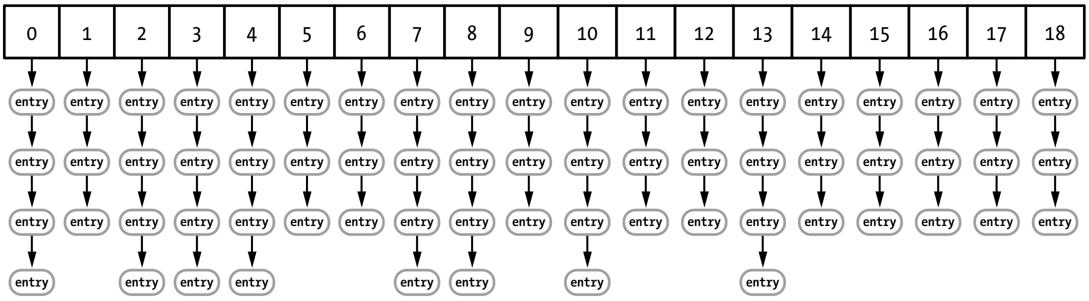

Figure 7-6: A hash table containing 65 values

In this figure the bin density has dropped to about 3.

By monitoring bin density, Ruby guarantees that the linked lists remain short and that retrieving a hash element is always fast. After calculating the hash value, Ruby just needs to step through one or two elements to find the target key.

HOW DOES RUBY REHASH ENTRIES IN A HASH TABLE?

You can find the `rehash` function (the code that loops through the `st_table_entry` structures and recalculates which bin to put the entry into) in the *st.c* source file. To keep things simple, Listing 7-2 shows the version of `rehash` from Ruby 1.8.7. While Ruby 1.9 and 2.0 work largely the same way, their C rehash code is somewhat more complex.

```
  static void
  rehash(table)
      register st_table *table;
  {
      register st_table_entry *ptr, *next, **new_bins;
      int i, old_num_bins = table->num_bins, new_num_bins;
      unsigned int hash_val;
❶     new_num_bins = new_size(old_num_bins+1);
      new_bins = (st_table_entry**)Calloc(new_num_bins,
                                          sizeof(st_table_entry*));
❷     for(i = 0; i < old_num_bins; i++) {
          ptr = table->bins[i];
          while (ptr != 0) {
              next = ptr->next;
❸             hash_val = ptr->hash % new_num_bins;
❹             ptr->next = new_bins[hash_val];
              new_bins[hash_val] = ptr;
              ptr = next;
          }
      }
❺     free(table->bins);
      table->num_bins = new_num_bins;
      table->bins = new_bins;
  }
```

Listing 7-2: The C code inside Ruby 1.8.7 that rehashes a hash table

In this listing, the `new_size` method call at ❶ returns the new bin count. Once Ruby has the new bin count, it allocates the new bins and then iterates over all the existing `st_table_entry` structures (all the key/value pairs in the hash) beginning at ❷. For each `st_table_entry` Ruby recalculates the bin position using the same modulus formula at ❸: `hash_val = ptr->hash % new_num_bins`. Then, Ruby saves each entry in the linked list for that new bin at ❹. Finally, Ruby updates the `st_table` structure and frees the old bins at ❺.

Experiment 7-2: Inserting One New Element into Hashes of Varying Sizes

One way to test whether this rehashing, or redistribution, of entries really occurs is to measure the amount of time Ruby takes to save one new element into existing hashes of different sizes. As we add more elements to the same hash, we should eventually see evidence that Ruby is taking extra time to rehash the elements.

The code for this experiment is shown in Listing 7-3.

```
  require 'benchmark'

❶ 100.times do |size|

    hashes = []
❷   10000.times do
      hash = {}
      (1..size).each do
        hash[rand] = rand
      end
      hashes << hash
    end

    GC.disable

    Benchmark.bm do |bench|
      bench.report("adding element number #{size+1}") do
        10000.times do |n|
❸         hashes[n][size] = rand
        end
      end
    end

    GC.enable
  end
```

Listing 7-3: Adding one more element to hashes of different sizes

At ❶ the outer loop iterates over hash sizes from 0 to 100, and at ❷ the inner loop creates 10,000 hashes of the given size. After disabling garbage collection, this experiment uses the benchmark library to measure how long it takes Ruby to insert a single new value at ❸ into all 10,000 hashes of the given size.

The results are surprising! Figure 7-7 shows the results for Ruby 1.8.

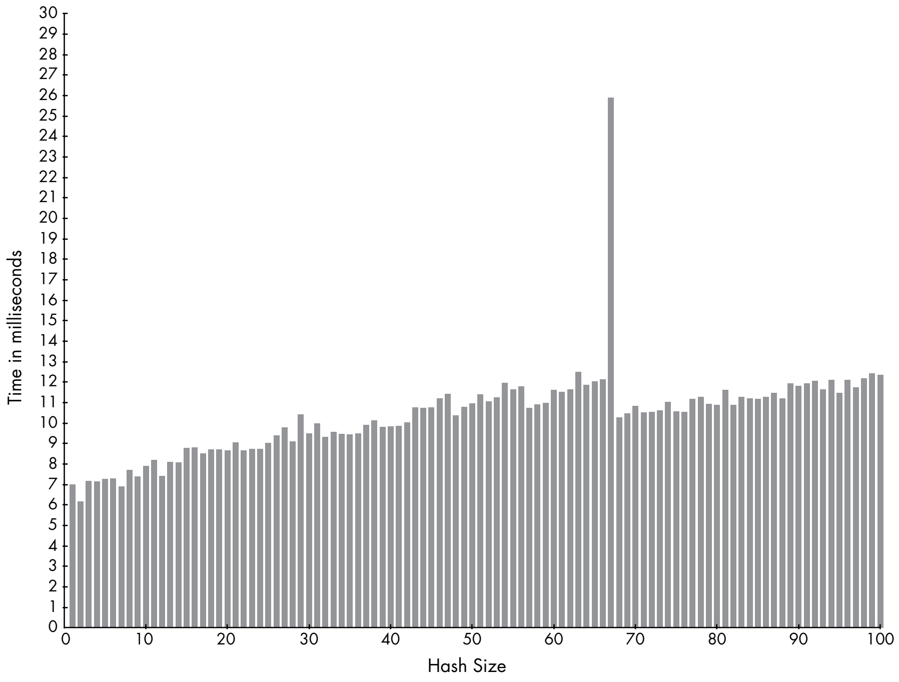

Figure 7-7: Time to add 10,000 key/value pairs vs. hash size (Ruby 1.8)

Interpreting these data values from left to right, we see the following:

- It takes about 7 ms to insert the first element into an empty hash (10,000 times).
- As the hash size increases from 2 to 3 and then up to about 60 or 65, the amount of time required to insert a new element slowly increases.
- It takes around 11 to 12 ms to insert each new key/value pair into a hash that contains 64, 65, or 66 elements (10,000 times).
- A huge spike! Inserting the 67th key/value pair takes over twice as much time: about 26 ms instead of 11 ms for 10,000 hashes!
- After inserting the 67th element, the time required to insert additional elements drops to about 10 ms or 11 ms and then slowly increases again from there.

What's going on here? Well, Ruby spends the extra time required to insert that 67th key/value pair reallocating the bin array from 11 to 19 bins and then reassigning the `st_table_entry` structures to the new bin array.

Figure 7-8 shows the same graph for Ruby 2.0. This time the bin density threshold is different. Instead of taking extra time to reallocate the elements into bins on the 67th insert, Ruby 2.0 does it when the 57th element is inserted. Later Ruby 2.0 performs another reallocation after the 97th element is inserted.

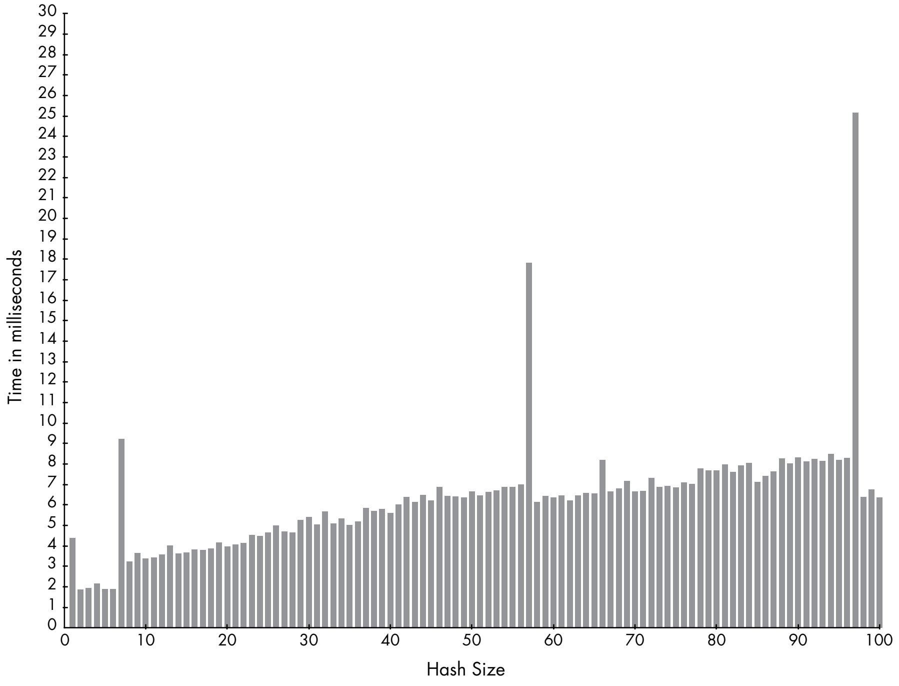

Figure 7-8: Time required to add 10,000 key/value pairs vs. hash size (Ruby 2.0)

The two smaller spikes on the 1st and 7th insert in this figure are curious. While not as pronounced as the spikes at the 57th and 97th elements, these smaller spikes are nonetheless noticeable. As it turns out, Ruby 2.0 contains another optimization that speeds up hash access even more for small hashes that contain less than 7 elements. I'll discuss this further in "Hash Optimization in Ruby 2.x" on page 187.

WHERE DO THE MAGIC NUMBERS 57 AND 67 COME FROM?

To see where these magic numbers come from (57, 67, and so on), look at the top of the *st.c* code file for your version of Ruby. You should find a list of prime numbers like the ones shown in Listing 7-4.

```
/*
Table of prime numbers 2^n+a, 2<=n<=30.
*/
static const unsigned int primes[] = {
❶  8 + 3,
❷  16 + 3,
❸  32 + 5,
  64 + 3,
  128 + 3,
  256 + 27,
  512 + 9,
--snip--
```

Listing 7-4: Ruby uses an algorithm based on prime numbers to determine the number of buckets required in each hash table.

This C array lists some prime numbers that occur near powers of 2. Peter Moore's hash table code uses this table to decide how many bins to use in the hash table. For example, the first prime number in the list above is 11 at ❶, which is why Ruby hash tables start with 11 bins. Later, as the number of elements increases, the number of bins increases to 19 at ❷, then to 37 at ❸, and so on.

Ruby always sets the number of hash table bins to a prime number in order to make it more likely that the hash values will be evenly distributed among the bins. Mathematically, prime numbers help here because they are less likely to share a common factor with the hash values, should a poor hash function return not entirely random values. Remember Ruby divides the hash values by the number of bins while calculating which bin to place the value into. If the hash values and bin count shared a factor, or even worse if the hash values were multiples of the bin count, the bin number (modulus) might always be the same. This would lead to the table entries being unevenly distributed among the bins.

Elsewhere in the *st.c* file, you should see this C constant:

```
#define ST_DEFAULT_MAX_DENSITY 5
```

This constant defines the maximum allowed density, or the average number of elements per bin.

Finally, you should see the code that decides when to perform a bin reallocation by finding where the constant `ST_DEFAULT_MAX_DENSITY` is used in *st.c*. For Ruby 1.8, you'll find this code:

```
if (table->num_entries/(table->num_bins) > ST_DEFAULT_MAX_DENSITY) {
  rehash(table);
```

Ruby 1.8 rehashes from 11 to 19 bins when the value `num_entries/11` is greater than 5—that is, when it equals 66. As this check is performed before a new element is added, the condition becomes true when you add the 67th element because `num_entries` would then be 66.

For Ruby 1.9 and Ruby 2.0, you'll find this code instead:

```
if ((table)->num_entries >
    ST_DEFAULT_MAX_DENSITY * (table)->num_bins) {
  rehash(table);
```

You can see that Ruby 2.0 rehashes for the first time when `num_entries` is greater than 5*11, or when you insert the 57th element.

How Ruby Implements Hash Functions

Now for a closer look at the actual hash function Ruby uses to assign keys and values to bins in hash tables. This function is central to the way the hash object is implemented—if it works well, Ruby hashes are fast, but a poor hash function can cause severe performance problems. Furthermore, Ruby uses hash tables internally to store its own information, in addition to the data values you save in hash objects. Clearly having a good hash function is very important!

Hash functions allow Ruby to find which bin contains a given key and value.

Let's review how Ruby uses hash values. Remember that when you save a new element in a hash—a new key/value pair—Ruby assigns that element to a bin inside the internal hash table used by that hash object, as shown in Figure 7-9.

Ruby calculates the modulus of the key's hash value based on the number of bins.

```
bin_index = internal_hash_function(key) % bin_count
```

Using the same example values we used earlier, this formula becomes:

```
2 = hash(:key) % 11
```

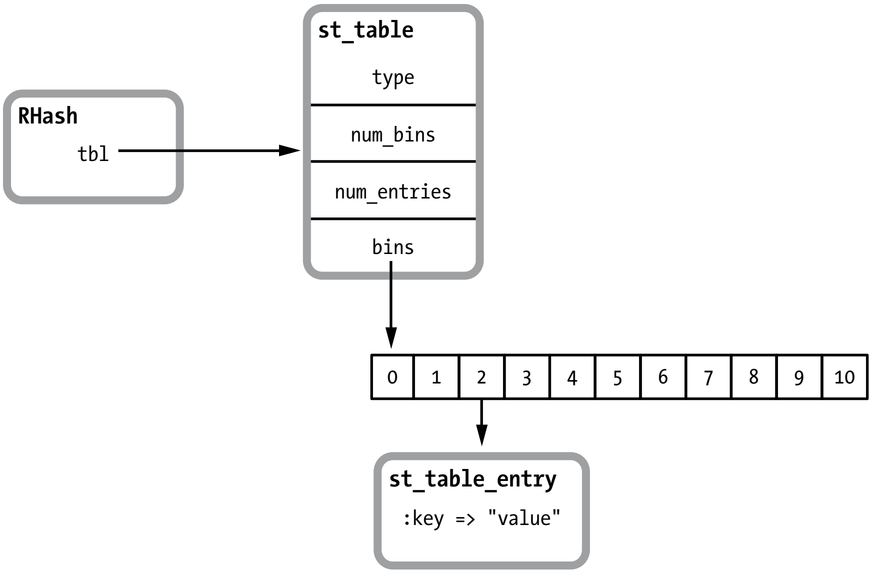

Figure 7-9: A Ruby hash object containing a single value (repeated from Figure 7-2)

This formula works well because Ruby's hash values are basically random integers for any given input data. To get a feel for how Ruby's hash function works, call the `hash` method, as shown in Listing 7-5.

```
$ irb
> "abc".hash
 => 3277525029751053763
> "abd".hash
 => 234577060685640459
> 1.hash
 => -3466223919964109258
> 2.hash
 => -2297524640777648528
```

Listing 7-5: Displaying the hash value for different Ruby objects

Here, even similar values have very different hash values. And if we call `hash` again, we always get the same integer value for the same input data.

```
> "abc".hash
 => 3277525029751053763
> "abd".hash
 => 234577060685640459
```

Here's how Ruby's hash function actually works for most Ruby objects:

- When you call `hash`, Ruby finds the default implementation in the `Object` class. You can override this if you want to.
- The C code used by the `Object` class's implementation of the `hash` method gets the C pointer value for the target object—that is, the actual memory address of that object's `RValue` structure. This is essentially a unique ID for that object.
- Ruby passes the pointer value through a complex C function (the hash function), which scrambles the bits in the value, producing a pseudorandom integer in a repeatable way.

In the case of strings and arrays, Ruby actually iterates through all of the characters in the string or the elements in the array and calculates a cumulative hash value. This guarantees that the hash will always be the same for any instance of a string or array and that it will change if any of the values in the string or array change. Integers and symbols are another special case. Ruby just passes their values right to the hash function.

To calculate hashes from values, Ruby 1.9 and 2.0 use a hash function called *MurmurHash*, which was invented by Austin Appleby in 2008. The name *Murmur* comes from the machine language operations used in the algorithm: *multiply* and *rotate*. (To learn how the Murmur algorithm actually works, read its C code in the *st.c* Ruby source code file. Or read Austin's web page on Murmur: *http://sites.google.com/site/murmurhash/*.)

Ruby 1.9 and 2.0 initialize MurmurHash using a random seed value that is reinitialized each time you restart Ruby. This means that if you stop and restart Ruby, you'll get different hash values for the same input data. It also means that if you try this yourself, you'll get different values than those above, but the hash values will always be the same within the same Ruby process.

Experiment 7-3: Using Objects as Keys in a Hash

Because hash values are evenly distributed, once Ruby divides them by the bin count, say 11, the remaining values (the modulus values) are random numbers between 0 and 10. This means that the `st_table_entry` structures are evenly distributed over the available bins as they are saved in the hash table, which ensures that Ruby will be able to quickly find any given key. The number of entries per bin will always be small.

But what if Ruby's hash function didn't return random integers but rather returned the same integer for every input data value? What would happen?

In that case, every time you added a key/value to a hash, it would always be assigned to the same bin. Ruby would end up with all of the entries in a single long list under that one bin, with no entries in any other bin, as shown in Figure 7-10.

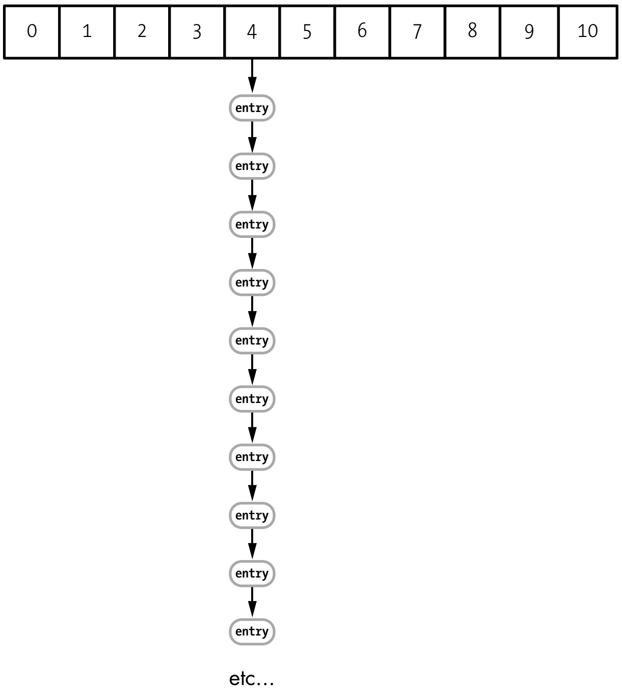

Figure 7-10: A hash table created with a very poor hash function

If you tried to retrieve a value from this hash, Ruby would have to look through this long list, one element at a time, to find the requested key. In this scenario, loading a value from the hash would be very, very slow.

To prove this is the case—and to illustrate just how important Ruby's hash function really is—we'll use objects with poor hash functions as keys in a hash. We'll repeat Experiment 7-1 here, but we'll use instances of a class I defined as the key values instead of random numbers. Listing 7-6 shows the code from Experiment 7-1, updated in two places.

```
  require 'benchmark'

❶ class KeyObject
    def eql?(other)
      super
    end
  end

  21.times do |exponent|

    target_key = nil

    size = 2**exponent
    hash = {}
    (1..size).each do |n|
❷     index = KeyObject.new
      target_key = index if n > size/2 && target_key.nil?
    hash[index] = rand
  end

  GC.disable

  Benchmark.bm do |bench|
    bench.report("retrieving an element
                  from a hash with #{size} elements 10000 times") do
      10000.times do
        val = hash[target_key]
      end
    end
  end

  GC.enable

end
```

Listing 7-6: Measuring how long it takes to retrieve an element from hashes of wildly different sizes. This is the same as Listing 7-1, but using instances of `KeyObject` as keys.

At ❶ we define an empty class called `KeyObject`. Note that I implemented the `eql?` method; this allows Ruby to search for the target key properly when I retrieve a value. However, in this example, I don't have any interesting data in `KeyObject`, so I simply call `super` and use the default implementation of `eql?` in the `Object` class.

Then, at ❷ we use new instances of `KeyObject` as the keys for my hash values. Figure 7-11 shows the results of this test.

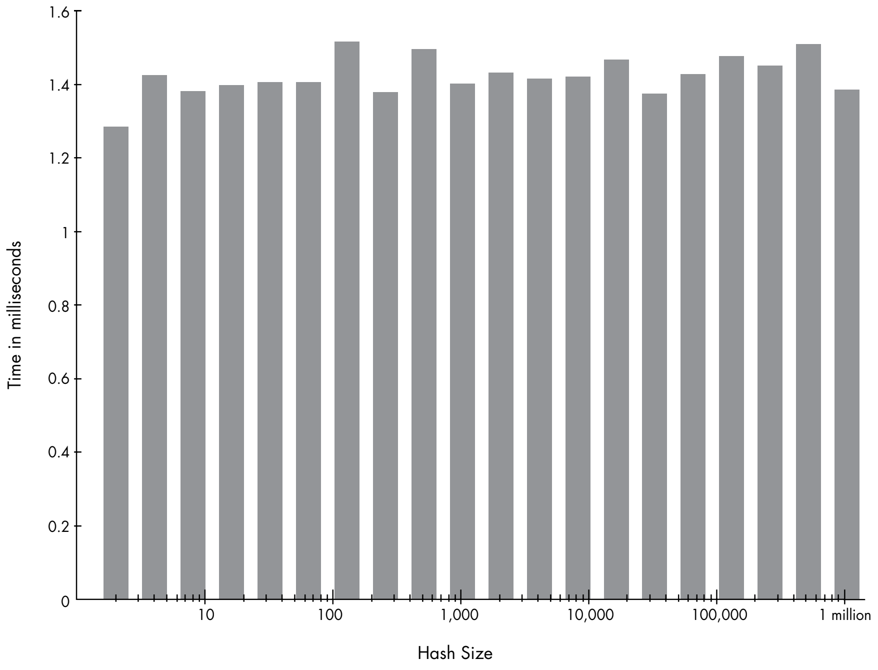

Figure 7-11: Time to retrieve 10,000 values vs. hash size, using objects as keys (Ruby 2.0)

As you can see, the results are very similar to those in Figure 7-4. The chart is more or less flat. It takes about the same amount of time to retrieve a value from a hash with 1 million elements as it does for a hash with just 1 element. No surprise there; using objects as keys hasn't slowed down Ruby at all.

Now let's change the `KeyObject` class and try again. Listing 7-7 shows the same code with a new hash function added at ❶.

```
require 'benchmark'

class KeyObject
  def hash
❶   4
  end
  def eql?(other)
    super
  end
end

21.times do |exponent|

  target_key = nil

  size = 2**exponent
  hash = {}
  (1..size).each do |n|
    index = KeyObject.new
    target_key = index if n > size/2 && target_key.nil?
    hash[index] = rand
  end

  GC.disable

  Benchmark.bm do |bench|
    bench.report("retrieving an element
                  from a hash with #{size} elements 10000 times") do
      10000.times do
        val = hash[target_key]
      end
    end
  end

  GC.enable
end
```

Listing 7-7: `KeyObject` now has a very poor hash function.

I've purposefully written a very poor hash function. Instead of returning a pseudorandom integer, the hash function in Listing 7-7 always returns the integer 4 at ❶, regardless of which `KeyObject` object instance you call it on. Now Ruby will always get 4 when it calculates the hash value. It will have to assign all of the hash elements to bin number 4 in the internal hash table, as in Figure 7-10.

Let's try this to see what happens! Figure 7-12 shows the results of running the code from Listing 7-7.

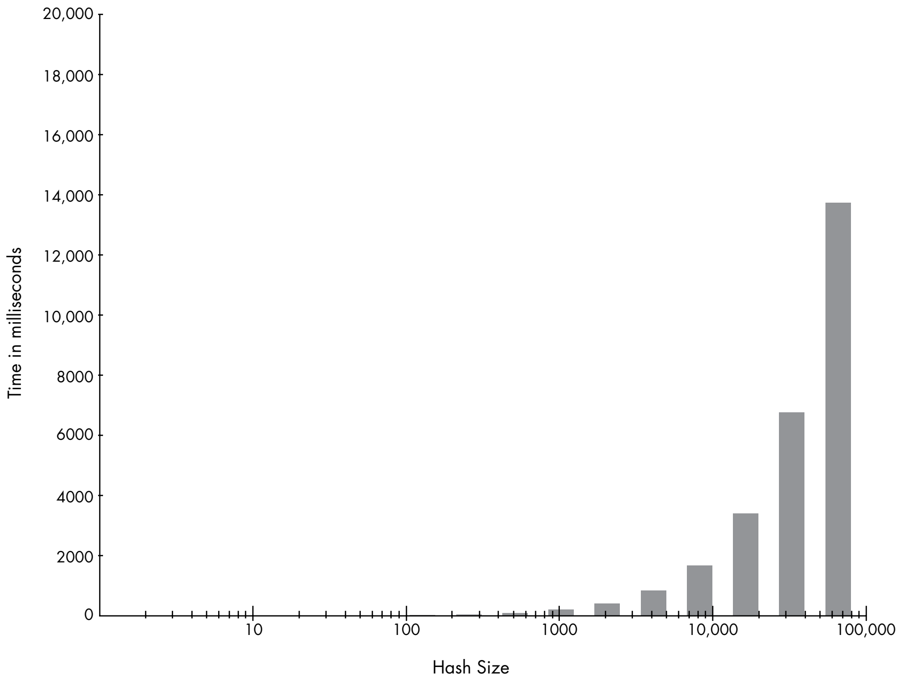

Figure 7-12: Time to retrieve 10,000 values vs. hash size, using a poor hash function (Ruby 2.0)

Figure 7-12 is very different from Figure 7-11! Notice the scale of the graph. The y-axis shows milliseconds, and the x-axis shows the number of elements in the hash on a logarithmic scale. But this time, notice that we have thousands of milliseconds—which means actual seconds—on the y-axis!

With one or a few elements, we can retrieve the 10,000 values very quickly—so quickly that the time is too small to appear on this graph. In fact, it takes about the same 1.5 ms. However, when the number of elements increases past 100 and especially 1,000, the time required to load the 10,000 values increases linearly with the hash size. For a hash containing about 10,000 elements, it takes over 1.6 full seconds to load the 10,000 values. If we continued the test with larger hashes, it would take minutes or even hours to load the values.

What's happening here is that all of the hash elements are saved into the same bin, forcing Ruby to search through the list one key at a time.

Hash Optimization in Ruby 2.x

Starting with version 2.0, Ruby introduced a new optimization to make hashes work even faster. For hashes that contain 6 or fewer elements, Ruby now avoids calculating the hash value entirely and simply saves the hash data in an array. These are known as *packed hashes*. Figure 7-13 shows a packed hash.

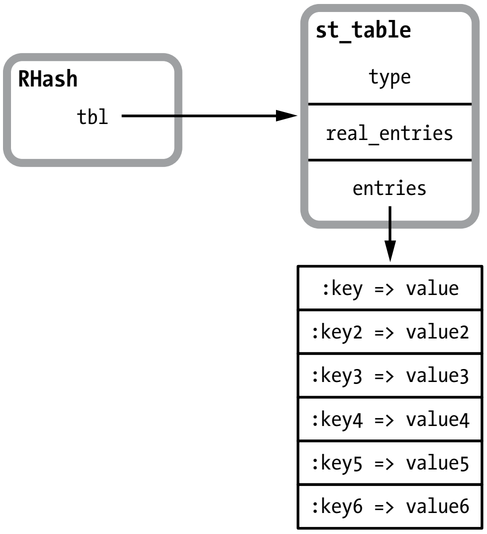

Figure 7-13: Internally, Ruby 2.x saves small hashes with 6 or fewer elements as arrays.

Ruby 2.x doesn't use the `st_table_entry` structure for small hashes, nor does it create a table of bins. Instead, it creates an array and saves the key/value pairs directly into this array. The array is large enough to fit 6 key/value pairs; once you insert a 7th key and value, Ruby discards the array, creates the bin array, and moves all 7 elements into `st_table_entry` structures as usual by calculating hash values. This explains the small spike we saw inserting the 7th element in Figure 7-8 (page 179). `real_entries` saves the number of values saved in the array between 0 and 6.

In a packed hash, there are only 6 or fewer elements; thus, it's faster for Ruby to iterate over the key values looking for a target value than it would be to calculate a hash value and use a bin array. Figure 7-14 shows how Ruby 2.x retrieves an element from a packed hash.

To find the value for a given key of `target`, Ruby iterates through the array and calls the `eql?` method on each key value if the values are objects. For simple values, such as integers or symbols, Ruby just uses a numerical comparison. Ruby 2.x never calls the hash function at all for packed hashes.

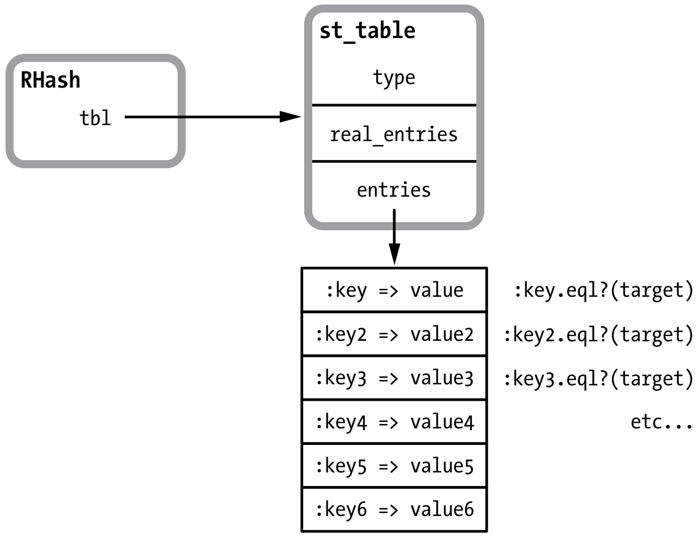

Figure 7-14: For small hashes, Ruby 2.x iterates over the array to find a given key.

Summary

Understanding hash tables is key to understanding how Ruby works internally because the speed and flexibility of hash tables allow Ruby to use them in many ways.

At the beginning of this chapter, we learned how hash tables are able to return values quickly, regardless of how many elements are in the table. Next, we learned how Ruby automatically increases the size of a hash table as you add more and more elements to it. The user of the hash table doesn't need to worry about how fast or large the table is. Hash tables will always be fast and will automatically expand as necessary.

Finally, we looked at the importance of Ruby's hash function. The hash table's algorithm depends on the underlying hash function. With an effective hash function, values are evenly distributed across the bins in the hash table with few collisions, allowing them to be saved and retrieved quickly. However, with a poor hash function, values would be saved in the same bin, leading to poor performance.