8

When Words Collide

8.1 Word Lists

If you have the “check spelling as you type” option turned on in your word processor you are undoubtedly familiar with the idea of a word list. The word processor maintains a list of correctly spelled words, and each time you hit the space bar as you edit a document the program scans the list to look for the word you just typed. If you type a word that is not in the list the program flags it, typically by underlining it in red. The word list is not the same as a dictionary – it doesn’t have any definitions or any other information about a word. The program simply keeps a list of strings it considers to be correctly spelled words.

In principle either of the two methods for searching through lists introduced in previous chapters could be used to search a word list. The simple linear search algorithm is far too inefficient – an English word list can easily have over 50,000 entries, and if a word is misspelled the search algorithm will compare its input to all 50,000 words before it returns. The binary search algorithm is much better, since only \( \log_2 50,000 \approx 16 \) comparisons will be made in an unsuccessful search for a misspelled word.

Binary search is a good choice for any situation where searches are not too frequent. A user might look up a name in an address book or a word in an on-line dictionary a few times a day, and binary search is a very good choice for these situations. The spell checker, on the other hand, needs to search its word list much more often, perhaps once or twice a second during bursts of heavy typing. If the search is not as efficient as possible the word processor will appear sluggish and users will be dissatisfied.

An ideal search algorithm would be one that knows exactly where to look in the list, without doing any comparisons. If all 50,000 words are loaded into an array in alphabetical order, this ideal algorithm might say “the word ‘their’ belongs in location 44,101 – yes, there it is, this is a correctly spelled word” or “the word ‘thier’ should be at location 44,236, but that location contains ‘thieve’ so ‘thier’ must be misspelled.” In mathematical terms, the ideal search would use a function \( f \) which takes a string \( s \) as its input and returns a number \( i \) so the algorithm could look at location \( i \) in its list of words. The cost of this search is the time to compute \( f \) plus the time to do one comparison. If \( f \) can be implemented efficiently this method will be faster than binary search since it will do just one comparison instead of \( \log_2 n \) comparisons.

Unfortunately there is no way to define such a magic function for large lists of arbitrary words. But if we aren’t concerned with keeping the words in alphabetical order, and all we
Figure 8.1: A hash table used to store the list of words used by a spell checker. This table has room for almost 1,000,000 words. When the table is first created it is empty. To add a word \( s \), compute \( i = h(s) \) and then store the string \( s \) in row \( i \). This figure shows the state of the table after adding five words (white rows), while the rest of the table remains empty (gray rows). An effective hash function will scatter words to random locations throughout the table.

want to know is whether or not a word is in the list, then we can define a data structure and associated algorithms that give almost the same performance.

The data structure used to hold the words is known as a hash table, and the function that finds places for words in the table is called a hash function. The name comes from the fact that the function works by breaking words into small pieces, recombining them, and coming up with a seemingly random location (Figure 8.1). The hash function \( h \) takes a string as its input and produces an integer as its output. When adding words to the table we figure out where to place a word \( s \) by computing \( i = h(s) \) and then storing \( s \) in row \( i \) of the table. Later, to see if a string \( t \) is a correctly spelled word, we compute \( j = h(t) \) and then look in row \( j \) of the table. If row \( j \) is empty or contains a different word then we conclude \( t \) is not in the table.

An effective hash function will scatter words all through the table, leaving many of the rows empty. In many cases the search for a misspelled word halts right away, i.e. we can tell immediately that a word is not in the list without doing any comparisons because the row where the word should be stored is empty. In other cases the row will not be empty and we might have to do a few comparisons in the region around the row we're directed to by the hash function. In the project for this chapter you will get a chance to experiment with a program that builds a hash table to store all 235,000 words found in the 2nd Edition of Webster's dictionary. When the table has around 1,000,000 rows, the majority are empty and most searches for misspelled words terminate immediately; in the worst case, an unsuccessful search makes at most five comparisons.

The utility of this new method of organizing data clearly depends on the hash function \( h \). The best hash functions will scatter input strings to random locations throughout the table.
8.1 Word Lists

Figure 8.2: A list of names of fruits stored in a hash table with 10 rows. The hash function uses the first letter in a string to find the location for the string, putting words that start with “a” in the first row, “b” in the second row, and so on. When the function runs out of rows, it “wraps around” to the first row again, so words starting with “k” would go in the first row because “k” is the 11th letter of the alphabet. Because words starting with “u” also go in row 0 the string “ugli” cannot be stored in this table unless there is a strategy for dealing with collisions.

Even two words that are close to each other in the dictionary, such as “abacus” and “abate”, will be widely separated when they are stored in the table (Figure 8.1). A poorly designed function would put several strings in the same location, so that when we go to look up a word we have to search through a list of all the words that map to the same location. When two or more words have the same hash value we say there is a collision.

As an example of how a hash table might be organized, suppose we want to build a table that will store names of fruits. This is far too small a list of strings to require using a hash table, but it provides a good example of the issues involved in building more realistic tables. A trivial hash function is based on the first letter in the string: words that start with “a” go in the first row, words that start with “b” are placed in the second row, and so on. If there are not enough rows in the table, the function “wraps around.” So if the table has 10 rows, words starting with “j” go in the last row (row 9), and strings starting with “k” wrap around and go in row 0 (Figure 8.2). Thus not only are other “a” words collisions with “apple”, but so are “kiwi”, “ugli”, and any others that start with “k” or “u.”

The project in this chapter explores the use of hash tables for word lists and other large sets of strings. The first part of the chapter is on the organization of the table itself – we’ll look at requirements for our hash tables and how to implement them in Ruby. The next three sections will explore the definition of hash functions and methods for dealing with collisions. The interactive Ruby projects have exercises that will help you understand how a hash function converts a string into a row number and issues that arise if the function is poorly designed. By the end of the chapter we will have an application that will read a list of words and evaluates how well a hash function maps words to different locations in the table.
8.2 Table Structure

To experiment with hash tables in Ruby we can start by making an array to represent the table. Recall from the experiments in Chapter 6 that arrays are collections of data items, and that we can store any type of data, including strings, in an array. If an array has \( n \) objects, the objects are numbered from 0 to \( n - 1 \), so if \( a \) is an array of size 10 the items in the array are referenced as \( a[0] \) to \( a[9] \).

Arrays used in previous projects typically started out as empty collections, and then grew as we inserted new items. Hash tables, on the other hand, have a fixed size that is determined when the table is created. The number of rows in a table is referred to as the capacity of the table. We can pass an integer parameter to the method that creates an array to get a new object with a predetermined number of cells. To make the initial table of Figure 8.2, which has room for 10 items, the expression is simply

\[
> t = \text{Array.new(10)}
\]

The cells in the new table each contain the special object \texttt{nil}. This is the same object we used in previous chapters to indicate a search was unsuccessful. Here we use it for a different purpose—it will be a place-holder to use in a row that does not yet contain a string.

Now that we have a table, we can insert items simply by assigning them to a location in the array. So to store the word “mango” in location 2 in the table we would just type

\[
> t[2] = \text{"mango"}
\]

Ruby has a method named \texttt{nil?} that we can use to see if a cell in the table is empty or not. After filling row 2 with a string, we can see if it's empty:

\[
> t[2].nil? \n\]

\[
> \text{false}
\]

Other rows will still be empty:

\[
> t[3].nil? \n\]

\[
> \text{true}
\]

Another way to verify only one slot in the table has been filled and the others are still empty it to just ask Ruby for the value of the variable that represents the table:

\[
> t \n\]

\[
[\text{nil, nil, "mango", nil, nil, nil, nil, nil, nil, nil}]
\]

(don't forget that the first cell in the table is \( t[0] \), so location 2 is actually the third item in the array). When we start experimenting with larger tables it’s going to be hard to determine which cells are full simply by printing the entire table. The following expression uses an array iterator to scan the list and print the array contents and their locations:

\[
> t.\text{each_with_index} \{ |x,i| \text{printf("%d: %s
", i, x) } \}
\]

The \texttt{each_with_index} iterator executes the expression between \{ and \} once for each item in the array, setting \( x \) to the item and \( i \) to its location.
Tutorial Project

Start a new session with IRB.

T1. Type the following expression to define an array named t with a capacity for 10 words:

```ruby
>> t = Array.new(10)
```

T2. Use assignment statements to insert strings into the array so it looks like the one shown in Figure 8.2. Start by putting the string “apple” in the first location:

```ruby
>> t[0] = "apple"
=> "apple"
```

T3. Make sure that assignment worked by checking the new value of t:

```ruby
>> t
=> ["apple", nil, nil, nil, nil, nil, nil, nil, nil]
```

T4. Continue by typing additional assignments to add “mango”, “orange”, and “strawberry” to the array until it looks like the one in Figure 8.2. When you are done you should see this when you ask for the value of t:

```ruby
>> t
=> ["apple", nil, "mango", nil, "orange", nil, nil, nil, "strawberry", nil]
```

T5. If the array was built correctly the first entry should not be nil:

```ruby
>> t[0].nil?
=> false
```

T6. The second row, however, should still be empty:

```ruby
>> t[1].nil?
=> true
```

T7. Type the expression that prints the array, showing each location and its contents:

```ruby
>> t.each_with_index { |x, i| printf("%d: %s\n", i, x) }
0: apple
1: 
2: mango
...
```

T8. Extra Credit: Can you figure out how to modify the expression that uses the `each_with_index` iterator so it prints only the rows that are not `nil`?

8.3 Radix-26

From the exercises in the previous section we can see that Ruby arrays can be used to hold the items. Now let’s turn our attention to the function that determines where items will be stored. The goal is to define a hash function that determines a unique position for a string so we can insert it and later retrieve it simply by computing the value of the function. We will begin by defining a trivial hash function, just to see that it is possible to create a function that takes a string as its input and generates a row number as its output. Then as we gain experience we will add more complexity until we have a function that scatters words randomly throughout an arbitrarily large table.

The tables we are using for these experiments are represented by arrays with a capacity to hold n objects. Since array indices run from 0 to n – 1, our goal is to define a function named h that will take a string s as its input and return an integer between 0 and n – 1. There are two steps in this process. First we will look at issues related to converting a string
of characters to a number, and then in the next section we will see how to make sure the
number is between 0 and \( n - 1 \).

The simplest possible way to convert a word to a number is the one used in the fruit list
example. A single letter corresponds to a number between 0 and 25, using the convention
that \( a = 0 \), \( b = 1 \), and so on, up to \( z = 25 \). A common name for this function that converts
a letter to a number is \( \text{ord} \), which stands for “ordinal” – it just means we want the relative
position of the letter in the English alphabet. A trivial hash function uses the first letter of
the input word to figure out which row to place the word in. We’ll call this function \( h_0 \):

\[
h_0(s) = \text{ord}(s_0)
\]

where the notation \( s_0 \) means “the first character in string \( s \).”

The obvious problem with this function is that every word that starts with “a” will be
assigned to row 0, every word starting with “b” would be placed in row 1, and so on. To
avoid a huge number of collisions we need a hash function based on more than just the first
letter in a word.

Let’s extend the hash function so it looks at the first two letters. One approach is that if
a word starts with “aa” it will go in row 0; if it starts “ab” it will go in row 1, and so on, up
to words starting “az”, which will go in row 25. Continue by using row 26 for a word that
starts with “ba”, 27 for words starting “bb”, and so on (Figure 8.3).

To use this function a table needs \( 26 \times 26 = 676 \) rows, numbered from 0 to 675. One way
to look at the new table, shown in Figure 8.3, is that it is a set of 26-word blocks, where
each block has words that start with the same letter. The first block of 26 rows is reserved
for words that start with “a”, the next block for words that start with “b”, and the last block
for words starting with “z.” From this point of view, the function that use the first two letters
to find a place for a word finds the start of the block defined by the first letter, and then the
row within the block using the second letter.

The row numbers for the first location in each block follow a pattern: block “a” starts
in row 0, block “b” in row 26, block “c” in row 52. We can take advantage of this regular
pattern by describing it mathematically: the \( i^{th} \) block starts in row \( 26 \times i \).
8.3 Radix-26

Putting this all together, the hash function that uses the first two letters in the string \( s \) is

\[
\text{h}_1(s) = \text{ord}(s_0) \times 26 + \text{ord}(s_1)
\]

where \( s_0 \) is the first letter in \( s \) and \( s_1 \) is the second letter. Some examples of how this function would work on a table being built for a spell-checker:

\[
\begin{align*}
\text{h}_1(\text{bed}) &= 1 \times 26 + 4 = 30 \\
\text{h}_1(\text{cnidarian}) &= 2 \times 26 + 13 = 65 \\
\text{h}_1(\text{zymurgy}) &= 25 \times 26 + 24 = 674
\end{align*}
\]

This method of assigning a numeric value to a set of two letters is the same method used to determine the value of two-digit numbers in a positional number system. For example, in the octal (base 8) number system the digits “47” represent the number 39 because the digit 4 in the “eights column” has a weight of \( 4 \times 8 = 32 \) and the digit 7 in the “ones column” has a weight of 7. If we are using hexadecimal (base 16) the same string represents the number 71 because now the 4 has a weight of 16 and \( 4 \times 16 + 7 = 71 \).

The new two-letter hash function is thus equivalent to finding the value of a “number” in a base-26 number system, where the “digits” are the letters from “a” to “z” and they have weights from 0 to 25. A common name for this function is radix-26, to reflect the fact that it is the same as assigning a value to a number in a base 26 number system.

Now that we know assigning a value to a 2-letter string is the same as finding its value in a base 26 number system it’s clear how to extend the function to strings of any length: just assign a weight of \( 26^i \) to the letter that is \( i \) places from the right end of the string. Some examples:

\[
\begin{align*}
\text{radix}_{26}(\text{bed}) &= 1 \times 26^2 + 4 \times 26 + 3 = 783 \\
\text{radix}_{26}(\text{cnidarian}) &= 2 \times 26^8 + 13 \times 26^7 + \ldots + 13 = 524,574,935,989 \\
\text{radix}_{26}(\text{zymurgy}) &= 25 \times 26^6 + 24 \times 26^5 + \ldots + 24 = 8,013,894,328
\end{align*}
\]

The radix-26 function shown above assigns a unique integer to every word, or more precisely to every string that has only the 26 lower case letters of the English alphabet. But as the examples show the number can be pretty big – far too big to be used as row numbers for tables with only a few hundred thousand entries. We’ll address this problem in the next section.

**Tutorial Project**

The methods required for this part of the tutorial project are in a file named `hashlab.rb`. Download this file and save it in the folder you are using for this project.

T9. Make a string named \( s \) to use to test the radix-26 function:

\[
\text{\texttt{\textgreater\textgreater s = \textquotesingle\textquotesingle hello\textquotesingle\textquotesingle}}
\]

T10. We can access the individual letters in a string the same way we access items in an array – write the name of the string followed by a location enclosed in brackets. To get the first letter in the string \( s \):

\[
\text{\texttt{s[0]}}
\]

© 2008 John S. Conery. Please do not distribute without permission.
11. You might have expected Ruby to print something like ‘h’ as the result of that previous expression, and in many languages¹ that is what you would see. But in Ruby the individual pieces of a string are numbers that correspond to the ASCII code for the letters. If you look at a table of ASCII codes you’ll see that the code for lower case h is 104, and that’s why Ruby printed 104 as the value of the previous expression.

12. Look up the ASCII code² for the letter e. Ask Ruby to print the value of the second letter in s:

```ruby
>> s[1]
=> 101
```

Did you get what you expected?

13. Look up the ASCII codes for the third, fourth, and fifth letters in s, and verify your results by asking Ruby to print these values.

14. The file hashlab.rb defines a method named `ord` that will convert a single letter into a number between 0 and 25. Load this file:

```ruby
>> load "hashlab.rb"
=> true
```

15. Test the `ord` on the first letter in s:

```ruby
>> s[0].ord
=> 7
```

16. What are the ordinal values of the second and third letters in s?

17. The hash function that uses the first two letters of a string s is \(ord(s_0) \times 26 + ord(s_1)\). The equivalent expression in Ruby is

```ruby
>> s[0].ord * 26 + s[1].ord
=> 186
```

18. Define a string named `t` that starts with something besides “he”, for example

```ruby
>> t = "bonjour"
=> "bonjour"
```

What do you think the two-letter hash function will produce for your string? Verify your answer by asking Ruby to evaluate `t[0].ord * 26 + t[1].ord`

19. The Ruby version of the radix-26 function is a method named `radix26` (this method is also defined in hashlab.rb). Test the method on one of the small strings used an an example earlier in the chapter:

```ruby
>> radix26("bed")
=> 783
```

20. Compute the radix-26 representation of the test string s:

```ruby
>> radix26(s)
=> 3276872
```

Look carefully at the result. Can you see how Ruby got this number?

21. Try calling the `radix26` method a few more times, making up your own test strings, until you are sure you know how the radix-26 function converts a string of letters into an integer.

22. What do you think Ruby will print as the value of `radix26("bee")`? You should be able to answer this question without asking IRB to compute it, since you already know `radix26("bed")` from an earlier exercise.

---

¹Including Ruby, starting with version 1.9.
²One place to find this table is to search for ASCII on Wikipedia.
8.4 The mod Function

To recap what we’ve seen so far:

- The goal is to store words in a table and to use a function to compute the location of a word so we don’t have to search for it.
- When we first make the table all the rows will be empty, and as new words are added the table will gradually fill up.
- A word \( s \) is inserted in row \( i = h(s) \).

The first step in finding the place for a word is to convert the sequence of letters into a number. The radix-26 function does this very well – it comes up with a unique number for each input string – but the number it makes for long words is very large, much larger than the number of rows in a table that can be reasonably be stored in a computer’s memory. The next step in our exploration of hash tables is to look at how values produced by radix-26 can be turned into row numbers for a hash table (Figure 8.4).

The key idea is to use the “mod” function introduced in the Sieve of Eratosthenes project. Recall that this function is defined to be the remainder of a division. For example, \( 123 \mod 7 = 4 \) because 123 divided by 7 is 17 with a remainder of 4. The useful fact about division modulo \( n \) for this project is that the result is always a number between 0 and \( n - 1 \).

The mod function is exactly what we need for the hash function. If the table has \( n \) rows, we want the hash function to compute a number between 0 and \( n - 1 \). All we have to do is use the radix-26 function to turn the string into a number, and then use the mod function to trim the result to a value between 0 and \( n - 1 \):

\[
h(s) = \text{radix}_{26}(s) \mod n
\]
Tutorial Project

T23. If you started a new IRB session load the file that has the definition of the radix-26 function:

```ruby
>> load "hashlab.rb"
=> true
```

T24. Recall that Ruby uses a percent sign as its operator for the mod function. Since 20 ÷ 7 = 2 R 6 we expect Ruby to print 6 when it is asked to evaluate 20 % 7. Type this expression and a few others that do divisions with integers to make sure you understand the mod function:

```ruby
>> 20 / 7
=> 2
>> 20 % 7
=> 6
>> 21 % 7
=> 0
>> 22 % 7
=> 1
```

T25. The `times` method used in previous projects is one way of asking Ruby to evaluate an expression several times. To print the numbers between 0 and 9:

```ruby
>> 10.times { |i| puts i }
0
1
2
...
```

T26. Call `times` again, but modify the expression in the body of the loop so Ruby prints i mod 4:

```ruby
>> 10.times { |i| puts i % 4 }
0
1
2
3
0
1
...
```

The motivation for using the mod function to implement a hash function was that the result of i mod n is a number between 0 and n − 1. Is that the case for the expression you just typed? i.e. are all the numbers printed by the call to `times` between 0 and 3?

T27. The table shown in Figure 8.4 has room for 10,000 words. That means the location for a word s would be radix26(s) mod 10,000. To find the table location for the word “zest” we simply use the remainder after dividing radix26(“zest”) by 10,000:

```ruby
>> radix26("zest")
=> 442591
>> radix26("zest") % 10000
=> 2591
```

Note that programming languages don’t use commas in large numbers, so 10,000 is entered in Ruby by typing a 1 and 4 zeroes (just like on a hand-held calculator).

T28. Use the expression above to find the table location for the remaining strings shown in Figure 8.4.

8.5 A New Type of Object

Before continuing with the project that will explore the ideas behind hash tables it is worth taking some time to understand how a programmer would tackle the problem of imple-
menting a hash table in a spell checker or other program that needs to quickly look up information in a table.

As the previous section showed, it is possible to use an array to represent the hash table. The problem, however, is that arrays can be used for many different purposes, and there are several operations on arrays that do not make sense for a hash table. For example, suppose we want to use an array named `a` to represent a table. Since `a` is an array, Ruby will let us call `a.length`, `a.first`, and other operations that make sense for most arrays but do not really apply to hash tables. Even worse, there are methods like `sort` and `reverse` that will completely alter the structure of the array and make it useless as a hash table. In a large program it is very easy to make a mistake and call one of these methods that does not make sense when the array is supposed to be used only as a table.

One of the most useful features of an object-oriented programming language like Ruby is that it is possible to define new types of object. For the tutorial project in this chapter we will be defining a new type of object specifically to represent hash tables. We will define this type so it allows us to create new tables, and add items and later retrieve them by using a hash function, but there are very few other things we need to do. None of the extra operations that Ruby provides for arrays will apply to the objects we will make to represent tables.

In an object-oriented language, a data type is known as a `class`, and we say an individual object “belongs to” a class or is an `instances` of a class.

When we ask Ruby to make a new object, Ruby uses instructions that are part of the class definition to build the new object. For example, in the expression

```ruby
>> s = "hello"
=> "hello"
```

Ruby recognizes that the letters between quotes are going to be in a string, so it uses instructions specific to the String class (class names start with upper case letters) to create a new object. The new object is then placed in the object store (Figure 8.5a).

We can ask Ruby to tell us which class an object belongs to by calling a method named `class`:

```ruby
>> s = "hello"
=> "hello"
>> s.class
=> String
>> a = [1,2,3]
=> [1, 2, 3]
>> a.class
=> Array
```

The result of evaluating the expression `a.class` is Ruby’s way of telling us that `a` is a reference to an Array object, i.e. that `a` is an instance of the Array class.

The new class we will create for this project will be named `HashTable`. After loading `HashTable.rb`, the file that contains the class definition, we will be able to make new hash tables by calling a special method named `new` that makes instances of the class:

```ruby
>> load "HashTable.rb"
=> true
>> t = HashTable.new(10)
=> #<HashTable:0x803a4>
```
(a) Two objects that belong to built-in classes: \( x \) is an instance of the Fixnum class and \( s \) is an instance of the String class. (b) The object store after creating an instance of the HashTable class, a new class defined for this project.

That funny looking string that starts with \# is just Ruby’s way of saying the value of \( t \) is a HashTable object. Inside this object there will eventually be an array and other pieces of data to represent a hash table, but for now it’s just another “cloud” in the object store (Figure 8.5b).

**Tutorial Project**

The outline of the HashTable class is in a file named HashTable.rb. There are several pieces missing from this file, but it is still worthwhile to load it into Ruby and use it to make new objects. Download HashTable.rb and put it in the folder you are using for this project (the same folder you used for hashlab.rb).

T29. To make sure the definitions in HashTable.rb are accessible to IRB type the command that loads the class definition:

```ruby
>> load "HashTable.rb"
true
```

T30. You should now be able to create a new HashTable object:

```ruby
>> t = HashTable.new(10)
#<HashTable:0x803a4>
```

Note: the result you see in your IRB session will not be exactly the same; the important part to look for is the word HashTable which indicates the new object was made by this class.

T31. Call the `class` method to ask Ruby which class this object belongs to:

```ruby
>> t.class
=> HashTable
```

T32. There is not much you can do with this object, since the methods that add items to the table and look up items have not been filled in yet:
8.6 The HashTable Class

The version of HashTable.rb you downloaded from the web site has only the bare bones outline of a class definition:

```ruby
class HashTable
  def initialize(n)
    @tbl = nil
    @size = 0
  end

  def find(s)
  end

  def insert(s)
  end

end

def h(s,n)
  return 0
end
```

In this section we are going to fill in the pieces of the three main methods inside the class and the method named \( h \) that will implement the hash function. The \texttt{initialize} method is called automatically by Ruby whenever a new object is created. The \texttt{insert} and \texttt{find} methods are the two main operations we want to perform on hash tables; as you might have guessed from their names these are the methods that add strings and look them up.

The two variables defined in the \texttt{initialize} method both have names that start with an @ sign. Variables that have names that start with this symbol are known as \textit{instance variables}. They are special variables that are "hidden" inside the class and are only accessible to methods like \texttt{find} and \texttt{insert} that are also defined inside the class.

For the HashTable class, we want every object to have an array that will hold the items being stored in the table. For convenience, we'll define a second variable that will quickly tell us how many rows are in the table. The parameter passed to the \texttt{initialize} method is the number of rows to use for the table, so the code we need to put in the body of \texttt{initialize} is trivial:

```ruby
def initialize(n)
  @tbl = Array.new(n)
end
```

These results might not look like much, but in fact they are very reassuring – we know IRB was able to load the class definition and it is able to make a HashTable object. If instead you got an error message (something like “undefined method”) if probably means the two files are not in a location where Ruby could find them.
Now when we call `HashTable.new(10)`, Ruby will create the new object in the object store for us, then call `initialize` and pass it the value 10. Our code will then create the array that will be used to hold the values in the table.

The `insert` and `find` methods will both call the hash function to figure out where a string is located. For now, let’s implement simple versions of each of these methods; in the next section we will revisit these methods and fix them so they deal with collisions. If we ignore the possibility that two words hash to the same location these two methods are very simple. To save a string `s` in the table, call the hash function to figure out the row for `s` and update that row in the array:

```ruby
def insert(s)
  i = h(s, @size)
  @tbl[i] = s
end
```

Note that both the string to insert and the table size are passed to `h`.

To see if a string `s` is in the table, call the hash function, and then, if the row is empty return `nil`, otherwise return the row number:

```ruby
def find(s)
  i = h(s, @size)
  if @tbl[i] == s
    return i
  else
    return nil
  end
end
```

The equation to use for the hash function was defined in Section 8.4, but for now let’s implement a very simple function so we can tell whether the `insert` and `find` methods are working the way we expect. To use the `h_0` function that simply looks at the first letter define `h` as

```ruby
def h(s,n)
  return s[0].ord % n
end
```

**Tutorial Project**

T33. Open the file named `HashTable.rb` and edit the definitions of the methods named `initialize`, `insert`, `find`, and `h` so the body of each method contains the expressions given in this section.

T34. Save the changes, then load the new definition into IRB:

```ruby
>> load "HashTable.rb"
=> true
```

T35. Make a new `HashTable` object using the latest definition:

---

3If you still have your previous session running you can type the `load` command to reload the file and overwrite the old definitions.
8.6 The HashTable Class

```ruby
>> t = HashTable.new(10)
=> #<HashTable:0x4e054>
```

T36. Since the current version is using the simple hash function that puts a word in a row that is
defined by the first letter, a word that starts with “a” should go in the first row in the table.
Verify this by inserting a word that starts with “a”:

```ruby
>> t.insert("apple")
=> "apple"
```

T37. Look up the word you just inserted.

```ruby
>> t.find("apple")
=> 0
```

The 0 returned by the call to find means the item was found in row 0, which is what we
expected to see.

T38. Insert the words “banana”, “pear”, and “tangerine”, and then call find to look them up. Are
they where you expected them to be?

T39. The HashTable class has several other methods that have already been written. These are
all defined in the file named hashlab.rb, which is automatically loaded every time you load
HashTable.rb. One of these extra methods is named print; when it is called, it prints every
non-empty row in the table. Call print to see the current state of your table:

```ruby
>> t.print
0: "apple"
1: "banana"
5: "pear"
9: "tangerine"
```

Does this view of the table correspond to what you learned by calling find?

T40. The way we defined insert in this section does not handle collisions. If a new word hashes
to the same location as an existing word the new one just overwrites the old one. Verify this
statement by adding “kiwi” to the table:

```ruby
>> t.insert("kiwi")
=> "kiwi"
```

T41. Where did the insert method place this string? Is it where you expected? What is the new
state of the table?

T42. Once you are satisfied the insert and find methods are working the way you expect them
to, open HashTable.rb with your text editor and change the definition of the hash function
so it computes a table location using the formula from Section 8.4. We’re going to change this
definition back to the simpler version for a future project, so instead of deleting the current
version it would be a good idea to just “comment it out” by putting a # at the front of the
line. Then add a second line that has the new equation. Your method should look like this:

```ruby
def h(s,n)
    # return s[0].ord % n
    return radix26(s) % n
end
```

T43. Save the file, and reload it in your current IRB session:

```ruby
>> load "HashTable.rb"
=> true
```

T44. Test the new hash function by seeing where it will place two of the words shown in Figure 8.4:

```ruby
>> h("bed",10000)
=> 783
>> h("cnidarian",10000)
=> 5989
```
Are these the row numbers shown in the figure?

T45. Make a new, larger table that has room for 10,000 entries:

```python
t = HashTable.new(10000)
=> #<HashTable:0x4f81c>
```

T46. Insert the strings shown in Figure 8.4: “a”, “bed”, “cnidarian”, and “zest”.

T47. Call the `print` method to look at the contents of the new table:

```plaintext
>>> t.print
0: "a"
783: "bed"
2591: "zest"
5989: "cnidarian"
```

### 8.7 Dealing with Collisions

At first it might seem like we should be able to avoid collisions in a hash table simply by making the table big enough so the probability of two words landing in the same row is very small. But it turns out that this is a case where our intuition leads us astray. The difficulty in making a sufficiently large table is illustrated by an example from probability theory known as the “birthday paradox.”

Imagine there is a lecture hall with 365 chairs, each labeled with a different day of the year. As students arrive for a class, they are directed to the chair that is labeled with their birthday. The question is, how many students can we expect to seat before two students have the same birthday, i.e. before an arriving student finds there is another student in their assigned seat? The answer may surprise you: the odds are better than 50% that by the time 23 students have arrived two of them will have the same birthday! By the time the 60th student arrives the probability of two students having the same birthday is well over 99%.

Our intuition tells us that since there are 59 students in the room and they are all in different seats the probability of the 60th student finding an empty seat should be \( p = \frac{306}{365} = 83.5\% \). That would be the case if we told each student “reach into this hat and pull out a seat number at random and take that seat if it’s empty.” But the 60th student does not get to choose a seat at random, and their seat is empty only if the first student did not take it and the second student did not take it and the third student... The odds of all these students taking a different seat are considerably smaller, which is why the 60th student will almost always find their seat taken.

The same situation occurs in a hash table. As words arrive to be placed in the table, the hash function computes a location, and the word is sent to that row in the table. If we have a hash table with 365 rows, there is a better than 50% chance there will be a collision by the time we add the 25th word, and we have less than a 1% chance of adding 60 words without a collision. The table in Figure 8.6 shows the probability of having a collision when placing 100 words in hash tables of varying sizes. When the table has 1,000 rows the probability of a collision is .9929. Even when there are 1,000,000 rows, which seems extravagant for a collection of only 100 words, there is a slight \( (p = .0049) \) chance of a collision.

If making the table larger is not a practical way to avoid collisions, what about designing a better hash function? There are algorithms for creating perfect hash functions, where each word in the input vocabulary is placed in a different row of a table, but they are practical only for small sets of words and cannot be used if the set is not known in advance. A spell
checker not only has a very large number of words to deal with, it will have to find places for new words as users add special vocabulary, names, and other strings that occur in the documents they write.

Two different ways of handling collisions are (a) if a table row is full, use a second algorithm to search for a nearby empty row, or (b) instead of placing strings in the table, have each row be a small list called a "bucket", and when adding a word to the table just append it to the end of one of the buckets. In the rest of this section we will look at how the second strategy can be implemented in our tutorial project (see Figure 8.7).

Making a table is the same as before: just use the Array constructor to make an array object with a specified number of rows, each of which will be set to nil:

```ruby
@tbl = Array.new(n)
```

To add a string `s` to the table, compute `i = h(s)`. Then, check to see if row `i` is nil. If it is, this is the first word for row `i`, and we need to create the list that will represent the bucket. Then, whether this is a new bucket or not, we just have to append `s` to the end. Expressing this algorithm in Ruby is straightforward:

```ruby
def insert(s)
  i = h(s, @size)
  @tbl[i] = [] if @tbl[i].nil?
  @tbl[i] << s
end
```

© 2008 John S. Conery. Please do not distribute without permission.
The first line sets i to the row number, the second creates a new array to use as the bucket for row i if there isn’t one there already, and the third adds the string to the end of the bucket.

The technique for checking to see if an item is in the table is also straightforward. Again start by using the hash function to compute the row number. Then if the row is empty (i.e. @tbl still contains nil at this location) the item is not there. If the row is not empty, scan the array to see if the item is in the bucket. The following code sets a variable named found to true if the string s is in the table, or to false if s is not in the table. Then, as before, if the item is in the table return its location, otherwise return nil:

```ruby
def find(s)
  i = h(s, @size)
  found = (@tbl[i].class == Array) && (@tbl[i].include?(s))
  if found
    return i
  else
    return nil
  end
end
```

The `include?` method is a built-in method that does a linear search through an array. The expression in the second line is true only if the table row is not empty and the bucket in that row contains the string s.

**Tutorial Project**

T48. It will be easier to verify the new versions of `insert` and `find` work as expected if we change the hash function back to the simpler version that uses only the first letter in a string. Remove the comment symbol from the front of the first line in the definition of h and insert a comment symbol before the second line, so the method looks like this:

```ruby
def h(s,n)
  return s[0].ord % n
  # return radix26(s) % n
end
```

T49. Edit the definitions of the `insert` and `find` methods so they contain the new code that stores items in buckets, as shown earlier in this section. When you are done your program should look like the one in Figure 8.8.

T50. To test the new methods, load the updated version of your file:

```ruby
>> load "HashTable.rb"
=> true
```

T51. Make a new table with a capacity to hold 10 items:

```ruby
>> t = HashTable.new(10)
=> #<HashTable:0x8fb88>
```

T52. Call the `insert` method to add a string to the table:

```ruby
>> t.insert("apple")
=> "apple"
```

T53. Now add a string that we expect to have a collision with the one already in the table:

```ruby
>> t.insert("kiwi")
=> "kiwi"
```
# HashTable.rb

```ruby
require "hashlab.rb"

class HashTable
  # a new table has an initially empty array of size n
  def initialize(n)
    @tbl = Array.new(n)
    @size = n
  end

  # return the row number that contains s, or nil if s not found
  def find(s)
    i = h(s, @size)
    found = (@tbl[i].class == Array) && (@tbl[i].include?(s))
    if found
      return i
    else
      return nil
    end
  end

  # add string s to the table
  def insert(s)
    i = h(s, @size)
    @tbl[i] = [] if @tbl[i].nil?
    @tbl[i] << s
    return s
  end

  # Hash function -- computes the row for a string s in a table
  # of size n.
  def h(s,n)
    # return s[0].ord % n
    return radix26(s) % n
  end
end
```

**Figure 8.8:** The final definition of the HashTable class, which uses “buckets” to save all the strings that hash to the same location in the table. Note: some methods are defined in the file hashlab.rb, which is included automatically when this file is loaded into IRB.
T54. Since the new methods are using buckets both strings should be in the table. Check to make sure that's the case:

```ruby
>> t.find("apple")
=> 0
>> t.find("kiwi")
=> 0
```

Did these two strings go where you expected?

T55. Call the `print` method to display the table:

```ruby
>> t.print
 0: ["apple", "kiwi"]
```

It looks like it worked – there is only one non-empty row, and it contains the two strings we just added.

T56. Add several more strings, making sure they are stored in the row you expect them to go into, that the first item in a row starts a new bucket, and that collisions are resolved by adding new strings to the end of existing rows.

```ruby
>> fruits = ["banana", "mango", "orange", "ugli", "cherry", "lime", "tangerine"]
>> fruits.each { |x| t.insert(x) }
```

T57. As a final test, try looking up some strings that are not in the table:

```ruby
>> t.find("peach")
=> nil
>> t.find("olive")
=> nil
```