04 0835 CH03 11/16/01 10:19 AM Page 137

CHAPTER

3

Manipulating Structured Data

IN THIS CHAPTER

04 0835 CH03 11/16/01 10:19 AM Page 138

All parts should go together without forcing. You must remember that the parts you are reassembling were disassembled by you. Therefore, if you can't get them together again, there must be a reason. By all means, do not use a hammer.

—IBM maintenance manual (1925)

Simple variables are not adequate for real-life programming. Every modern language supports more complex forms of structured data and also provides mechanisms for creating new abstract data types.

Historically, arrays are the earliest known and most widespread of the complex data structures. Long ago, in FORTRAN, they were called *subscripted variables*. Today they have changed somewhat, but the basic idea is the same in all languages.

More recently, the hash has become an extremely popular programming tool. Like an array, a *hash* is an indexed collection of data items; unlike an array, it may be indexed by any arbitrary object. (In Ruby, as in most languages, array elements are accessed by a numerical index.)

Finally, in this chapter we will look at more advanced data structures. Some of these are just special "views" of an array or hash; for example, stacks and queues can be implemented easily using arrays. Other structures such as trees and graphs may be implemented in different ways according to the situation and the programmer's preference.

But let's not get ahead of ourselves. We will begin with arrays.

Working with Arrays

Arrays in Ruby are indexed by integers and are zero based, just like C arrays. The resemblance ends there, however.

A Ruby array is dynamic. It is possible (but not necessary) to specify its size when you create it. After creation, it can grow as needed without any intervention by the programmer.

A Ruby array is *heterogeneous* in the sense that it can store multiple data types rather than just one type. Actually, it stores object references rather than the objects themselves, except in the case of immediate values such as `Fixnum` values.

An array keeps up with its own length so that we don't have to waste our time with calculating it or keeping an external variable in sync with the array. Also, iterators are defined so that, in practice, we rarely need to know the array length anyway.

Finally, the `Array` class in Ruby provides arrays with many useful functions for accessing, searching, concatenating, and otherwise manipulating arrays. We'll spend the remainder of this section exploring the built-in functionality and expanding on it.

Creating and Initializing an Array

The special class method `[]` is used to create an array; the data items listed within the brackets are used to populate the array. The three ways of calling this method are shown here (note that arrays `a`, `b`, and `c` will all be populated identically):

```
a = Array.[](1,2,3,4)
b = Array[1,2,3,4]
c = [1,2,3,4]
```

Also, the class method `new` can take zero, one, or two parameters. The first parameter is the initial size of the array (number of elements). The second parameter is the initial value of each of the elements. Here's an example:

```
d = Array.new              # Create an empty array
e = Array.new(3)           # [nil, nil, nil]
f = Array.new(3, "blah")   # ["blah", "blah", "blah"]
```

Accessing and Assigning Array Elements

Element reference and assignment are done using the class methods `[]` and `[]=`, respectively. Each can take an integer parameter, a pair of integers (start and length), or a range. A negative index counts backward from the end of the array, starting at `-1`.

Also, the special instance method `at` works like a simple case of element reference. Because it can take only a single integer parameter, it is slightly faster. Here's an example:

```
a = [1, 2, 3, 4, 5, 6]
b = a[0]                # 1
c = a.at(0)             # 1
d = a[-2]               # 5
e = a.at(-2)            # 5
f = a[9]                # nil
g = a.at(9)             # nil
h = a[3,3]              # [4, 5, 6]
i = a[2..4]             # [3, 4, 5]
j = a[2...4]            # [3, 4]

a[1] = 8                # [1, 8, 3, 4, 5, 6]
a[1,3] = [10, 20, 30]   # [1, 10, 20, 30, 5, 6]
a[0..3] = [2, 4, 6, 8]  # [2, 4, 6, 8, 5, 6]
a[-1] = 12              # [2, 4, 6, 8, 5, 12]
```

Note in the following example how a reference beyond the end of the array causes the array to grow (note also that a subarray can be replaced with more elements than were originally there, also causing the array to grow):

```
k = [2, 4, 6, 8, 10]
k[1..2] = [3, 3, 3]     # [2, 3, 3, 3, 8, 10]
k[7] = 99               # [2, 3, 3, 3, 8, 10, nil, 99]
```

Finally, we should mention that an array assigned to a single element will actually insert that element as a nested array (unlike an assignment to a range), as shown here:

```
m = [1, 3, 5, 7, 9]
m[2] = [20, 30]         # [1, 3, [20, 30], 7, 9]

# On the other hand...
m = [1, 3, 5, 7, 9]
m[2..2] = [20, 30]      # [1, 3, 20, 30, 7, 9]
```

The method `slice` is simply an alias for the `[]` method:

```
x = [0, 2, 4, 6, 8, 10, 12]
a = x.slice(2)               # 4
b = x.slice(2,4)             # [4, 6, 8, 10]
c = x.slice(2..4)            # [4, 6, 8]
```

The special methods `first` and `last` will return the first and last elements of an array, respectively. They will return `nil` if the array is empty. Here's an example:

```
x = %w[alpha beta gamma delta epsilon]
a = x.first      # "alpha"
b = x.last       # "epsilon"
```

We have seen that some of the element-referencing techniques actually can return an entire subarray. There are other ways to access multiple elements, which we'll look at now.

The method `indices` will take a list of indices (or *indexes*, if you prefer) and return an array consisting of only those elements. It can be used where a range cannot (when the elements are not all contiguous). The alias is called `indexes`. Here's an example:

```
x = [10, 20, 30, 40, 50, 60]
y = x.indices(0, 1, 4)        # [10, 20, 50]
z = x.indexes(2, 10, 5, 4)    # [30, nil, 60, 50]
```

Finding an Array's Size

The method `length` (or its alias `size`) will give the number of elements in an array. Note that this is one less than the index of the last item:

```
x = ["a", "b", "c", "d"]
a = x.length              # 4
b = x.size                # 4
```

The method `nitems` is the same except that it does not count `nil` elements:

```
y = [1, 2, nil, nil, 3, 4]
c = y.size                # 6
d = y.length              # 6
e = y.nitems              # 4
```

Comparing Arrays

Comparing arrays is slightly tricky. If you do it at all, you should do it with caution.

The instance method `<=>` is used to compare arrays. It works the same as the other contexts in which it is used, returning either `-1` (meaning "less than"), `0` (meaning "equal"), or `1` (meaning "greater than"). The methods `==` and `!=` depend on this method.

Arrays are compared in an "elementwise" manner; the first two elements that are not equal will determine the inequality for the whole comparison. (Therefore, preference is given to the leftmost elements, just as if we were comparing two long integers "by eye," looking at one digit at a time.) Here's an example:

```
a = [1, 2, 3, 9, 9]
b = [1, 2, 4, 1, 1]
c = a <=> b            # -1 (meaning a < b)
```

If all elements are equal, the arrays are equal. If one array is longer than another, and they are equal up to the length of the shorter array, the longer array is considered to be greater:

```
d = [1, 2, 3]
e = [1, 2, 3, 4]
f = [1, 2, 3]
if d == f
  puts "d equals f"   # Prints "d equals f"
end
```

Because the `Array` class does not mix in the `Comparable` module, the usual operators, `<`, `>`, `<=`, and `>=`, are not defined for arrays. However, we can easily define them ourselves if we choose:

```
class Array

  def <=> other) == -1
  end

  def <=(other)
    (self < other) or (self == other)
  end

  def >(other)
```

04 0835 CH03 11/16/01 10:19 AM Page 142

```
      (self <=> other) == 1
    end

    def >=(other)
      (self > other) or (self == other)
    end

  end
```

Having defined them, we can use them as you would expect:

```
if a < b
  print "a < b"       # Prints "a < b"
else
  print "a >= b"
end
if d < e
  puts "d < e"        # Prints "d < e"
end
```

It is conceivable that comparing arrays will result in the comparison of two elements for which `<=>` is undefined or meaningless. This will result in a runtime error (a `TypeError`) because the comparison `3 <=> "x"` is problematic:

```
g = [1, 2, 3]
h = [1, 2, "x"]
if g < h             # Error!
  puts "g < h"       # No output
end
```

However, in case you are still not confused, equal and not-equal will still work in this case. This is because two objects of different types are naturally considered to be unequal, even though we can't say which is greater or less than the other:

```
if g != h             # No problem here.
  puts "g != h"       # Prints "g != h"
end
```

Finally, it is conceivable that two arrays containing mismatched data types will still compare with the `<` and `>` operators. In the case shown here, we get a result before we stumble across the incomparable elements:

```
i = [1, 2, 3]
j = [1, 2, 3, "x"]
if i < j             # No problem here.
  puts "i < j"       # Prints "i < j"
end
```

Sorting an Array

The easiest way to sort an array is to use the built-in `sort` method, as shown here:

```
words = %w(the quick brown fox)
list = words.sort  # ["brown", "fox", "quick", "the"]
# Or sort in place:
words.sort!        # ["brown", "fox", "quick", "the"]
```

This method assumes that all the elements in the array are comparable with each other. A mixed array, such as `[1, 2, "three", 4]`, will normally give a type error.

In a case like this one, you can use the block form of the same method call. The example here assumes that there is at least a `to_s` method for each element (to convert it to a string):

```
a = [1, 2, "three", "four", 5, 6]
b = a.sort {|x,y| x.to_s <=> y.to_s}
# b is now [1, 2, 5, 6, "four", "three"]
```

Of course, such an ordering (in this case, depending on ASCII) may not be meaningful. If you have such a heterogeneous array, you may want to ask yourself why you are sorting it in the first place or why you are storing different types of objects.

This technique works because the block returns an integer (`-1`, `0`, or `1`) on each invocation. When a `-1` is returned, meaning that `x` is less than `y`, the two elements are swapped. Therefore, to sort in descending order, we could simply swap the order of the comparison, like this:

```
x = [1, 4, 3, 5, 2]
y = x.sort {|a,b| b <=> a}    # [5, 4, 3, 2, 1]
```

The block style can also be used for more complex sorting. Let's suppose we want to sort a list of book and movie titles in a certain way: We ignore case, we ignore spaces entirely, and we want to ignore any certain kinds of embedded punctuation. Listing 3.1 presents a simple example. (Both English teachers and computer programmers will be equally confused by this kind of alphabetizing.)

LISTING 3.1 Specialized Sorting

```
titles = ["Starship Troopers",
          "A Star is Born",
          "Star Wars",
          "Star 69",
          "The Starr Report"]
sorted = titles.sort do |x,y|
  # Delete leading articles
  a = x.sub(/^(a |an |the )/i, "")
  b = y.sub(/^(a |an |the )/i, "")
```

LISTING 3.1 Continued

```
    # Delete spaces and punctuation
    a.delete!(" .,-?!")
    b.delete!(" .,-?!")
    # Convert to uppercase
    a.upcase!
    b.upcase!
    # Compare a and b
    a <=> b
  end

  # sorted is now:
  # [ "Star 69", "A Star is Born", "The Starr Report"
  #   "Starship Troopers", "Star Wars"]
```

This example is not overly useful, and it could certainly be written more compactly. The point is that any arbitrarily complex set of operations can be performed on two operands in order to compare them in a specialized way. (Note, however, that we left the original operands untouched by manipulating copies of them.) This general technique can be useful in many situations—for example, sorting on multiple keys or sorting on keys that are computed at runtime.

Selecting from an Array by Criteria

Sometimes we want to locate an item or items in an array much as though we were querying a table in a database. There are several ways to do this; the ones we outline here are all mixed in from the `Enumerable` module.

The `detect` method will find at most a single element. It takes a block (into which the elements are passed sequentially) and returns the first element for which the block evaluates to a value that is not `false`. Here's an example:

```
x = [5, 8, 12, 9, 4, 30]
# Find the first multiple of 6
x.detect {|e| e % 6 == 0 }         # 12
# Find the first multiple of 7
x.detect {|e| e % 7 == 0 }         # nil
```

Of course, the objects in the array can be of arbitrary complexity, as can the test in the block.

The `find` method is a synonym for `detect`; the method `find_all` is a variant that will return multiple elements as opposed to a single element. Finally, the method `select` is a synonym for `find_all`. Here's an example:

```
# Continuing the above example...
x.find {|e| e % 2 == 0}            # 8
```

```
    x.find_all {|e| e % 2 == 0}        # [8, 12, 4, 30]
    x.select {|e| e % 2 == 0}          # [8, 12, 4, 30]
```

The `grep` method will invoke the relationship operator to match each element against the pattern specified. In its simplest form, it will simply return an array containing the matched elements. Because the relationship operator (===) is used, the so-called pattern need not be a regular expression. (The name *grep*, of course, comes from the Unix tool of the same name, historically meaning something like *general regular expression pattern-matcher*.) Here's an example:

```
    a = %w[January February March April May]
    a.grep(/ary/)      # ["January, "February"]
    b = [1, 20, 5, 7, 13, 33, 15, 28]
    b.grep(12..24)     # [20, 13, 15]
```

There is a block form that will effectively transform each result before storing it in the array; the resulting array contains the return values of the block rather than the values passed into the block:

```
    # Continuing above example...
    # Let's store the string lengths
    a.grep(/ary/) {|m| m.length}     # [7, 8]
    # Let's square each value
    b.grep(12..24) {|n| n*n}         # {400, 169, 225}
```

The `reject` method is complementary to `select`. It excludes each element for which the block evaluates to `true`. The in-place mutator `reject!` is also defined:

```
    c = [5, 8, 12, 9, 4, 30]
    d = c.reject {|e| e % 2 == 0}    # [5, 9]
    c.reject! {|e| e % 3 == 0}
    # c is now [5, 8, 4]
```

The `min` and `max` methods may be used to find the minimum and maximum values in an array. There are two forms of each of these. The first form uses the "default" comparison, whatever that may be in the current situation (as defined by the <=> method). The second form uses a block to do a customized comparison. Here's an example:

```
    a = %w[Elrond Galadriel Aragorn Saruman Legolas]
    b = a.min                                 # "Aragorn"
    c = a.max                                 # "Saruman"
    d = a.min {|x,y| x.reverse <=> y.reverse} # "Elrond"
    e = a.max {|x,y| x.reverse <=> y.reverse} # "Legolas"
```

Suppose we want to find the *index* of the minimum or maximum element (assuming it is unique). We could use the `index` method for tasks such as this, as shown here:

```
    # Continuing above example...
    i = a.index a.min     # 2
    j = a.index a.max     # 3
```

04 0835 CH03 11/16/01 10:19 AM Page 146

This same technique can be used in other similar situations. However, if the element is not unique, the first one in the array will naturally be the one found.

Using Specialized Indexing Functions

The internals of a language handle the mapping of array indexes to array elements through what is called an *indexing function*. Because the methods that access array elements can be overridden, we can in effect index an array in any way we wish.

For example, in Listing 3.2, we implement an array that is "one-based" rather than "zero-based."

LISTING 3.2 Implementing a One-Based Array

```
class Array2 < Array

  def [](index)
    if index>0
      super(index-1)
    else
      raise IndexError
    end
  end

  def []=(index,obj)
    if index>0
      super(index-1,obj)
    else
      raise IndexError
    end
  end

end


x = Array2.new

x[1]=5
x[2]=3
x[0]=1  # Error
x[-1]=1 # Error
```

Note that the negative indexing (from the end of an array) is disallowed here. Also, be aware that if this were a real-life solution, there would be other changes to make, such as the `slice` method and others. However, this gives the general idea.

A similar approach can be used to implement multidimensional arrays (as you'll see later in the section "Using Multidimensional Arrays").

It is also possible to implement something like a triangular matrix (see Listing 3.3). This is like a special case of a two-dimensional array in which element `x,y` is always the same as element `y,x` (so that only one needs to be stored). This is sometimes useful, for example, in storing an undirected graph (as you'll see toward the end of this chapter).

LISTING 3.3 Triangular Matrix

```
class TriMatrix

  def initialize
    @store = []
  end

  def [](x,y)
    if x > y
      index = (x*x+x)/2 + y
      @store[index]
    else
      raise IndexError
    end
  end

  def []=(x,y,v)
    if x > y
      index = (x*x+x)/2 + y
      @store[index] = v
    else
      raise IndexError
    end
  end

end


t = TriMatrix.new

t[3,2] = 1
puts t[3,2]  # 1

puts t[2,3]  # IndexError
```

04 0835 CH03 11/16/01 10:19 AM Page 148

Here, we have chosen to implement the matrix so that the row number must be greater than or equal to the column number; we also could have coded it so that the same pair of indexes simply mapped to the same element. These design decisions will depend on your use of the matrix.

It would have been possible to inherit from `Array`, but we thought this solution was easier to understand. The indexing formula is a little complex, but 10 minutes with pencil and paper should convince anyone it is correct. Some enhancements could probably be made to this class to make it truly useful, but we will leave that to you, the reader.

Also, it is possible to implement a triangular matrix as an array containing arrays that increase in size as the row number gets higher. This is somewhat similar to what we have done in the section “Using Multidimensional Arrays.” The only tricky part would be to make sure that a row does not accidentally grow past its proper size.

Implementing a Sparse Matrix

Sometimes we need an array that has very few of its elements defined; the rest of its elements can be undefined (or more often zero). This so-called “sparse matrix” has historically been a waster of memory that has led people to seek indirect ways of implementing it.

Of course, in most cases, a Ruby array will suffice, because modern architectures typically have large amounts of memory. An unassigned element will have the value `nil`, which takes only a few bytes to store.

On the other hand, assigning an array element beyond the previous bounds of the array also creates all the `nil` elements in between. For example, if elements `0` through `9` are defined, and we suddenly assign to element `1000`, we have in effect caused elements `10` through `999` to spring into being as `nil` values. If this is unacceptable, you might consider an alternative.

The alternative we have to suggest, however, does not involve arrays at all. If you really need a sparse matrix, a hash might be the best solution. See the section “Using a Hash As a Sparse Matrix” for more information.

Using Arrays As Mathematical Sets

Most languages do not directly implement sets (Pascal being one exception). However, Ruby arrays have some features that make them usable as sets. We’ll present these here and add a few of our own.

First of all, an array can have duplicate entries. If you specifically want to treat the array as a set, you can remove these entries (using `uniq` or `uniq!`).

The two most basic operations performed on sets are union and intersection. These are accomplished by the `|` (`or`) and `&` (`and`) operators, respectively. In accordance with the idea that a set does not contain duplicates, any duplicates will be removed. (This may be contrary to your expectations if you are used to array union and intersection operations in some other language.) Here's an example:

```
a = [1, 2, 3, 4, 5]
b = [3, 4, 5, 6, 7]
c = a | b           # [1, 2, 3, 4, 5, 6, 7]
d = a & b           # [3, 4, 5]
# Duplicates are removed...
e = [1, 2, 2, 3, 4]
f = [2, 2, 3, 4, 5]
g = e & f           # [2, 3, 4]
```

The concatenation operator `+` can be used for set union, but it does *not* remove duplicates.

The `-` method is a "set difference" operator that will produce a set with all the members of the first set except the ones appearing in the second set. (See the section "Finding Elements in One Array but Not Another" for more information.) Here's an example:

```
a = [1, 2, 3, 4, 5]
b = [4, 5, 6, 7]
c = a - b           # [1, 2, 3]
# Note that the extra items 6 and 7 are irrelevant.
```

To "accumulate" sets, you can use the `|=` operator; as expected, `a |= b` simply means `a = a | b`. Likewise `&=` can progressively "narrow down" the elements of a set.

There is no exclusive-or defined for arrays, but we can make our own very easily. In set terms, this corresponds to elements that are in the union of two sets but *not* in the intersection. Here's an example:

```
class Array

  def ^(other)
    (self | other) - (self & other)
  end

end

x = [1, 2, 3, 4, 5]
y = [3, 4, 5, 6, 7]
z = x ^ y           # [1, 2, 6, 7]
```

04 0835 CH03 11/16/01 10:19 AM Page 150

To check for the presence of an element in a set, we can use the method `include?` or `member?` (essentially an alias mixed in from `Comparable`), like so:

```
x = [1, 2, 3]
if x.include? 2
  puts "yes"     # Prints "yes"
else
  puts "no"
end
```

Of course, this is a little backward from what we are used to in mathematics, where the operator resembling a Greek epsilon denotes set membership. It is backward in the sense that the set is on the left rather than on the right; we are not asking "Is this element in this set?" but rather "Does this set contain this element?"

Many people will not be bothered by this at all. However, if you are used to Pascal or Python (or you have ingrained mathematical inclinations), you may want to use a different way. We present two options here:

```
class Object

  def in(other)
    other.include? self
  end

end

x = [1, 2, 3]
if 2.in x
  puts "yes"     # Prints "yes"
else
  puts "no"
end
```

This is still a trifle ugly, but at least the ordering is more familiar. As for making it look "more like an operator," Ruby's amazingly flexible parser allows you to write the expression `2.in x` instead as `2 .in x` or even `2. in x`, should you wish to go that far.

For those who can't stand the presence of that period, it is conceivable that we could overload an operator such as <= for that purpose. However, something like this should be done with caution.

There has been talk of a Python-like (or Pascal-like) `in` operator for Ruby. However, it is no more than talk at this time.

How do we tell whether a set is a subset or a superset of another? There are no built-in methods, but we can do it as demonstrated in Listing 3.4.

LISTING 3.4 Subset and Superset

```
class Array

  def subset?(other)
    self.each  do |x|
      if !(other.include? x)
        return false
      end
    end
    true
  end

  def superset?(other)
    other.subset?(self)
  end

end

a = [1, 2, 3, 4]
b = [2, 3]
c = [2, 3, 4, 5]

flag1 = c.subset? a     # false
flag2 = b.subset? a     # true
flag3 = c.superset? b   # true
```

Note that we've chosen the "natural" ordering—that is, `x.subset? y` means "Is *x* a subset of *y*?" rather than vice versa.

To detect the null set (or empty set), we simply detect the empty array. The `empty?` method will do this.

The concept of set negation (or complement) depends on the concept of a *universal set*. Because in practical terms this will vary from one application or situation to another, the best way is the simplest—define the universe and then do a set difference, as shown here:

```
universe = [1, 2, 3, 4, 5, 6]
a = [2, 3]
b = universe - a   # complement of a = [1, 4, 5, 6]
```

Of course, if you really feel the need, you could define a unary operator (such as `-` or `~`) to do this.

You can iterate through a set just by iterating through the array. The only difference is that the elements will come out in order, which you may not want. To see how to iterate randomly, refer to the section "Iterating over an Array."

04 0835 CH03 11/16/01 10:19 AM Page 152

Finally, we may sometimes want to compute the powerset of a set. This is simply the set of all possible subsets (including the null set and the original set itself). Those familiar with discrete math, especially combinatorics, will see that there must be 2^n of these subsets. We can generate the powerset as demonstrated in Listing 3.5.

LISTING 3.5 Powerset of a Set

```
class Array

  def powerset
    num = 2**size
    ps = Array.new(num, [])
    self.each_index do |i|
      a = 2**i
      b = 2**(i+1) - 1
      j = 0
      while j < num-1
        for j in j+a..j+b
          ps[j] += [self[i]]
        end
        j += 1
      end
    end
    ps
  end

end

x = [1, 2, 3]
y = x.powerset
# y is now:
#   [[], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]]
```

Randomizing an Array

Sometimes we want to scramble an array into a random order. The first example that might come to mind is a card game, but there are other circumstances, such as presenting a list of questions to a user in a random order, in which we might use this.

To accomplish this task, we can use `rand` in the `Kernel` module. Here's one way to do this:

```
class Array

  def randomize
    arr=self.dup
    arr.collect { arr.slice!(rand arr.length) }
  end

  def randomize!
    arr=self.dup
    result = arr.collect { arr.slice!(rand arr.length) }
    self.replace result
  end

end

x = [1, 2, 3, 4, 5]
y = x.randomize      # [3, 2, 4, 1 ,5]
x.randomize!         # x is now [3, 5, 4, 1, 2]
```

The key to understanding this solution is knowing that the `slice!` method will return the value of an array element and, at the same time, delete that element from the array (so that it cannot be used again).

There are other ways to perform this operation. If you find a better one, let us know.

If we wanted simply to pick an array element at random (without disallowing duplicates), we could do that as follows.

```
class Array

  def pick_random
    self[rand(self.length)]
  end

end
```

Finally, remember that any time you are using `rand`, you can generate a predictable sequence (for example, for testing) simply by seeding with a known seed using `srand`.

Using Multidimensional Arrays

If you want to use multidimensional arrays for numerical purposes, an excellent library in the Ruby Application Archive called `NArray` (by Masahiro Tanaka) is available. If you want to use matrixes, you can use the `matrix.rb` standard library, as mentioned in Chapter 2, “Simple Data Tasks.”

In Listing 3.6, we present a way of handling multidimensional arrays by overloading the `[]` and `[]=` methods to map elements onto a nested array. The class `Array3` presented here will handle three-dimensional arrays in a rudimentary fashion, but it is far from complete.

LISTING 3.6 Three-dimensional Array

```
class Array3

  def initialize
     @store = [[[]]]
  end

  def [](a,b,c)
    if @store[a]==nil ||
       @store[a][b]==nil ||
       @store[a][b][c]==nil
      return nil
    else
      return @store[a][b][c]
    end
  end

  def []=(a,b,c,x)
    @store[a] = [[]] if @store[a]==nil
    @store[a][b] = [] if @store[a][b]==nil
    @store[a][b][c] = x
  end

end


x = Array3.new
x[0,0,0] = 5
x[0,0,1] = 6
x[1,2,3] = 99

puts x[1,2,3]
```

Note that all we really gain here is the convenience of a "comma" notation `[x,y,z]` instead of the more C-like `[x][y][z]`. If the C-style notation is acceptable to you, you can just use nested arrays in Ruby. Another minor benefit is the prevention of the situation in which `nil` is the receiver for the bracket method.

Finding Elements in One Array but Not Another

Finding elements in one array but not another is simpler in Ruby than in many languages. It is a simple "set difference" problem:

```
text = %w[the magic words are squeamish ossifrage]
dictionary = %w[an are magic the them these words]
```

```
# Find potential misspellings
unknown = text - dictionary   # ["squeamish", "ossifrage"]
```

Transforming or Mapping Arrays

The `collect` method (part of `Enumerable`) is a useful little tool that proves to be a time and labor saver in many circumstances. If you are a Smalltalk programmer, this may be more intuitive than if you come from a C background.

This method simply operates on each element of an array in some arbitrary way to produce a new array. In other words, it "maps" an array onto another array (hence the synonym `map`). Here's an example:

```
x = %w[alpha bravo charlie delta echo foxtrot]
# Get the initial letters
a = x.collect {|w| w[0..0]}       # %w[a b c d e f]
# Get the string lengths
b = x.collect {|w| w.length}       # [5, 5, 7, 5, 4, 7]
# map is just an alias
c = x.map {|w| w.length}           # [5, 5, 7, 5, 4, 7]
```

The in-place variant `collect!` (or `map!`) is also defined:

```
x.collect! {|w| w.upcase}
# x is now %w[ALPHA BRAVO CHARLIE DELTA ECHO FOXTROT]
x.map! {|w| w.reverse}
# x is now %w[AHPLA OVARB EILRAHC ATLED OHCE TORTXOF]
```

Removing `nil` Values from an Array

The `compact` method (or its in-place version `compact!`) will remove `nil` values from an array, leaving the rest untouched:

```
a = [1, 2, nil, 3, nil, 4, 5]
b = a.compact     # [1, 2, 3, 4, 5]
a.compact!        # a is now [1, 2, 3, 4, 5]
```

Removing Specific Array Elements

It is easy to delete elements from a Ruby array, and there are many ways to do it. If you want to delete one specific element by index, `delete_at` is a good way:

```
a = [10, 12, 14, 16, 18]
a.delete_at(3)              # Returns 16
# a is now [10, 12, 14, 18]
a.delete_at(9)              # Returns nil (out of range)
```

If you want to delete all instances of a certain piece of data, `delete` will do the job. It will return the value of the objects deleted or `nil` if the value was not found. Here's an example:

```
b = %w(spam spam bacon spam eggs ham spam)
b.delete("spam")            # Returns "spam"
# b is now ["bacon", "eggs", "ham"]
b.delete("caviar")          # Returns nil
```

The `delete` method will also accept a block. This may be a little counterintuitive, though. All that happens is that the block is evaluated (potentially performing a wide range of operations) if the object is not found and the value of the block is returned, as shown here:

```
c = ["alpha", "beta", "gamma", "delta"]
c.delete("delta") { "Nonexistent" }
# Returns "delta" (block is never evaluated)
c.delete("omega") { "Nonexistent" }
# Returns "Nonexistent"
```

The `delete_if` method will pass every element into the supplied block and delete the elements for which the block evaluates to `true`. It behaves similarly to `reject!`, except that the latter can return `nil` when the array remains unchanged. Here's an example:

```
email = ["job offers", "greetings", "spam", "news items"]
# Delete four-letter words
email.delete_if {|x| x.length==4 }
# email is now ["job offers", "greetings", "news items"]
```

The `slice!` method accesses the same elements as `slice` but deletes them from the array as it returns their values:

```
x = [0, 2, 4, 6, 8, 10, 12, 14, 16]
a = x.slice!(2)                         # 4
# x is now [0, 2, 6, 8, 10, 12, 14, 16]
b = x.slice!(2,3)                       # [6, 8, 10]
# x is now [0, 2, 12, 14, 16]
c = x.slice!(2..3)                      # [12, 14]
# x is now [0, 2, 16]
```

The `shift` and `pop` methods can be used for deleting array elements (for more about their intended uses, see the discussion of stacks and queues elsewhere in this chapter):

```
x = [1, 2, 3, 4, 5]
x.pop                  # Delete the last element
# x is now [1, 2, 3, 4]
x.shift                # Delete the first element
# x is now [2, 3, 4]
```

Finally, the `clear` method will delete all the elements in an array. It is equivalent to assigning an empty array to the variable, but it's marginally more efficient. Here's an example:

```
x = [1, 2, 3]
x.clear
# x is now []
```

Concatenating and Appending onto Arrays

Very frequently we want to take an array and append an element or another array. There are many ways to do this with a Ruby array.

The "append" operator `<<` will append an object onto an array; the return value is the array itself so that these operations can be "chained":

```
x = [1, 5, 9]
x << 13        # x is now [1, 5, 9, 13]
x << 17 << 21  # x is now [1, 5, 9, 13, 17, 21]
```

Similar to the append are the `unshift` and `push` methods, which add to the beginning and end of an array, respectively. See the section "Using an Array As a Stack or Queue" for more information.

Arrays may be concatenated with the `concat` method or by using the `+` and `+=` operators:

```
x = [1,2]
y = [3,4]
z = [5,6]
b = y + z         # [3,4,5,6]
b += x            # [3,4,5,6,1,2]
z.concat y        # z is now [5,6,3,4]
```

Using an Array As a Stack or Queue

The basic stack operations are `push` and `pop`, which add and remove items, respectively, at the end of an array. The basic queue operations are `shift` (which removes an item from the beginning of an array) and `unshift` (which adds an element to the beginning). The append operator, can also be used to add an item to the end of an array (basically a synonym for `push`).

Don't get confused. The `shift` and `unshift` methods work on the *beginning* of an array; the `push`, `pop`, and `<<` methods work on the *end*.

For a better discussion of this topic, see the section "Working with Stacks and Queues."

Iterating over an Array

The `Array` class has the standard iterator `each`, as is to be expected. However, it also has other useful iterators.

The `reverse_each` method will iterate in reverse order. It is equivalent to using `reverse` and then `each`, but it is faster. Here's an example:

```
words = %w(Son I am able she said)
str = ""
words.reverse_each { |w| str += "#{w} "}
# str is now "said she able am I Son "
```

If we only want to iterate over the indexes, we can use `each_index`. Saying `x.each_index` is equivalent to saying `(0..(x.size-1)).each` (that is, iterating over the range of indexes).

The iterator `each_with_index` (mixed in from `Comparable`) will pass both the element and the index into the block, as shown here:

```
x = ["alpha", "beta", "gamma"]
x.each_with_index do |x,i|
  puts "Element #{i} is #{x}"
end
# Produces three lines of output
```

Suppose you wanted to iterate over an array in random order? The following example uses the iterator `random_each` (which simply invokes the `randomize` method from section "Randomizing an Array"):

```
class Array

# Assumes we have defined randomize

  def random_each
    temp = self.randomize
    temp.each {|x| yield x}
  end

end

dwarves = %w(Sleepy Dopey Happy Sneezy Grumpy Bashful Doc)
list = ""
dwarves.random_each {|x| list += "#{x} "}
# list is now:
# "Bashful Dopey Sleepy Happy Grumpy Doc Sneezy "
# (Your mileage may vary.)
```

Interposing Delimiters to Form a String

Frequently we will want to insert delimiters in between array elements in a "fencepost" fashion; that is, we want to put delimiters between the elements, but not before the first one or after the last one. The method `join` will do this, as will the `*` operator:

```
been_there = ["Veni", "vidi", "vici."]
journal = been_there.join(", ")        # "Veni, vidi, vici."

# Default delimiter is space
letters = ["Phi","Mu","Alpha"]
musicians = letters.join               # "Phi Mu Alpha"

people = ["Bob","Carol","Ted","Alice"]
movie = people * " and "
# movie is now "Bob and Carol and Ted and Alice"
```

Note that if we really need to treat the last element differently, perhaps by inserting the word *and*, we can do it manually, like so:

```
list = %w[A B C D E F]
with_commas = list[0..-2]*", " + ", and " + list[-1]
# with_commas is now "A, B, C, D, E, and F"
```

Reversing an Array

To reverse the order of an array, use the `reverse` or `reverse!` method:

```
inputs = ["red", "green", "blue"]
outputs = inputs.reverse          # ["green","blue","red"]
priorities = %w(eat sleep code)
priorities.reverse!               # ["code","sleep","eat"]
```

Removing Duplicate Elements from an Array

If you want to remove duplicate elements from an array, the `uniq` method (or its in-place mutator `uniq!`) will do the job:

```
breakfast = %w[spam spam eggs ham eggs spam]
lunch = breakfast.uniq   # ["spam","eggs","ham"]
breakfast.uniq!          # breakfast has changed now
```

Interleaving Arrays

Suppose you want to take two arrays and "interleave" them so that the new array contains alternating elements from each of the two original ones. There must be a hundred ways to do this. Here is one way:

04 0835 CH03 11/16/01 10:19 AM Page 160

```
a = [1, 2, 3, 4]
b = ["a", "b", "c", "d"]
c = []
a.each_with_index { |x,i| c << x << b[i]}
# c is now [1, "a", 2, "b", 3, "c", 4, "d"]
```

Counting Frequency of Values in an Array

There is no `count` method for arrays as there is for strings (to count the occurrences of each data item). Therefore, we've created one here:

```
class Array

  def count
    k=Hash.new(0)
    self.each{|x| k[x]+=1 }
    k
  end

end

meal = %w[spam spam eggs ham eggs spam]
items = meal.count
# items is {"ham" => 1, "spam" => 3, "eggs" => 2}
spams = items["spam"]   # 3
```

Note that a hash is returned. No pun intended.

Inverting an Array to Form a Hash

An array is used to associate an integer index with a piece of data. However, what if you want to invert that association (that is, associate the data with the index, thus producing a hash)? The following method will do just that:

```
class Array

  def invert
    h={}
    self.each_with_index{|x,i| h[x]=i}
    h
  end

end

a = ["red","yellow","orange"]
h = a.invert     # {"orange"=>2, "yellow"=>1, "red"=>0}
```

04 0835 CH03 11/16/01 10:19 AM Page 161

Synchronized Sorting of Multiple Arrays

Suppose you want to sort an array, but you have other arrays that corresponded with this one on an element-for-element basis. In other words, you don't want to get them out of sync. How would you do this?

The solution we present in Listing 3.7 will sort an array and gather the resulting set of indexes. The list of indexes (itself an array) can be applied to any other array to put its elements in the same order.

LISTING 3.7 Synchronized Array Sorting

```
class Array

  def sort_index
    d=[]
    self.each_with_index{|x,i| d[i]=[x,i]}
    if block_given?
      d.sort {|x,y| yield x[0],y[0]}.collect{|x| x[1]}
    else
      d.sort.collect{|x| x[1]}
    end
  end

  def sort_by(ord=[])
    return nil if self.length!=ord.length
    self.indexes(*ord)
  end

end


a = [21, 33, 11, 34, 36, 24, 14]
p a
p b=a.sort_index
p a.sort_by b
p c=a.sort_index {|x,y| x%2 <=> y%2}
p a.sort_by c
```

Establishing a Default Value for New Array Elements

When an array grows and new (unassigned) elements are created, these elements default to `nil` values:

```
a = Array.new
a[0]="x"
```

```
a[3]="y"
# a is now ["x", nil, nil, "y"]
```

What if we want to set those new elements to some other value? As a specific instance of a general principle, we offer the `ZArray` class in Listing 3.8, which will default new unassigned elements to `0`.

LISTING 3.8 Specifying a Default for Array Elements

```
class ZArray < Array

  def [](x)
    if x > size
      for i in size+1..x
        self[i]=0
      end
    end
    v = super(x)
  end

  def []=(x,v)
    max = size
    super(x,v)
    if size - max > 1
      (max..size-2).each do |i|
        self[i] = 0
      end
    end
  end

end


num = ZArray.new
num[1] = 1
num[2] = 4
num[5] = 25
# num is now [0, 1, 4, 0, 0, 25]
```

Working with Hashes

Hashes are known in some circles as *associative arrays*, *dictionaries*, and by various other names. Perl and Java programmers in particular will be familiar with this data structure.

Think of an array as an entity that creates an association between index *x* and data item *y*. A hash creates a similar association, with at least two exceptions. First, for an array, *x* is always an integer; for a hash, it need not be. Second, an array is an ordered data structure; a hash typically has no ordering.

A hash key can be of any arbitrary type. As a side effect, this makes a hash a nonsequential data structure. In an array, we know that element 4 follows element 3; but in a hash, the key may be of a type that does not define a real predecessor or successor. For this reason (and others), there is no notion in Ruby of the pairs in a hash being in any particular order.

You may think of a hash as an array with a specialized index, or as a database "synonym table" with two fields, stored in memory. Regardless of how you perceive it, it is a powerful and useful programming construct.

Creating a New Hash

As with `Array`, the special class method `[]` is used to create a hash. The data items listed in the brackets are used to form the mapping of the hash.

Six ways of calling this method are shown here (note that hashes `a1` through `c2` will all be populated identically):

```
a1 = Hash.[]("flat",3,"curved",2)
a2 = Hash.[]("flat"=>3,"curved"=>2)
b1 = Hash["flat",3,"curved",2]
b2 = Hash["flat"=>3,"curved"=>2]
c1 = {"flat",3,"curved",2}
c2 = {"flat"=>3,"curved"=>2}
# For a1, b1, and c1: There must be
# an even number of elements.
```

Also, the class method `new` can take a parameter specifying a *default* value. Note that this default value is not actually part of the hash; it is simply a value returned in place of `nil`. Here's an example:

```
d = Hash.new         # Create an empty hash
e = Hash.new(99)     # Create an empty hash
f = Hash.new("a"=>3) # Create an empty hash
e["angled"]          # 99
e.inspect            # {}
f["b"]               # {"a"=>3} (default value is
                     #   actually a hash itself)
f.inspect            # {}
```

Specifying a Default Value for a Hash

The default value of a hash is an object that is referenced in place of `nil` in the case of a missing key. This is useful if you plan to use methods with the hash value that are not defined for `nil`. It can be assigned upon creation of the hash or at a later time using the `default=` method.

All missing keys point to the same default value object, so changing the default value has a side effect:

```
a = Hash.new("missing")  # default value object is "missing"
a["hello"]               # "missing"
a.default="nothing"
a["hello"]               # "nothing"
a["good"] << "bye"       # "nothingbye"
a.default                # "nothingbye"
```

The special instance method `fetch` raises an `IndexError` exception if the key does not exist in the `Hash` object. It takes a second parameter that serves as a default value. Also, `fetch` optionally accepts a block to produce a default value in case the key is not found. This is in contrast to `default`, because the block allows each missing key to have its own default. Here's an example:

```
a = {"flat",3,"curved",2,"angled",5}
a.fetch("pointed")                   # IndexError
a.fetch("curved","na")               # 2
a.fetch("x","na")                    # "na"
a.fetch("flat") {|x| x.upcase}       # 3
a.fetch("pointed") {|x| x.upcase}    # "POINTED"
```

Accessing and Adding Key/Value Pairs

`Hash` has class methods `[]` and `[]=`, just as `Array` has; they are used much the same way, except that they accept only one parameter. The parameter can be any object, not just a string (although string objects are commonly used). Here's an example:

```
a = {}
a["flat"] = 3        # {"flat"=>3}
a.[]=("curved",2)    # {"flat"=>3,"curved"=>2}
a.store("angled",5)  # {"flat"=>3,"curved"=>2,"angled"=>5}
```

The method `store` is simply an alias for the `[]=` method, both of which take two arguments, as shown in the example.

The method `fetch` is similar to the `[]` method, except that it raises an `IndexError` for missing keys. It also has an optional second argument (or alternatively a code block) for dealing with default values (see the section "Specifying a Default Value for a Hash"). Here's an example:

```
a["flat"]       # 3
a.[]("flat")    # 3
```

```
a.fetch("flat")  # 3
a["bent"]        # nil
```

Suppose you are not sure whether the `Hash` object exists, and you would like to avoid clearing an existing hash. The obvious way is to check whether the hash is defined, as shown here:

```
unless defined? a
  a={}
end
a["flat"] = 3
```

Another way to do this is as follows:

```
a ||= {}
a["flat"] = 3
```

The same problem can be applied to individual keys, where you only want to assign a value if the key does not exist:

```
a=Hash.new(99)
a[2]             # 99
a                # {}
a[2] ||= 5       # 99
a                # {}
b=Hash.new
b                # {}
b[2]             # nil
b[2] ||= 5       # 5
b                # {2=>5}
```

Note that `nil` may be used as either a key or an associated value:

```
b={}
b[2]       # nil
b[3]=nil
b          # {3=>nil}
b[2].nil? # true
b[3].nil? # true
b[nil]=5
b          # {3=>nil,nil=>5}
b[nil]     # 5
b[b[3]]    # 5
```

Deleting Key/Value Pairs

Key/value pairs of a `Hash` object can be deleted using `clear`, `delete`, `delete_if`, `reject`, `reject!`, and `shift`.

Use `clear` to remove all key/value pairs. This is essentially the same as assigning a new empty hash, but it's marginally faster.

Use `shift` to remove an unspecified key/value pair. This method returns the pair as a two-element array (or `nil` if no keys are left):

```
a = {1=>2, 3=>4}
b = a.shift       # [1,2]
# a is now {3=>4}
```

Use `delete` to remove a specific key/value pair. It accepts a key and returns the value associated with the key removed (if found). If the key is not found, the default value is returned. It also accepts a block to produce a unique default value rather than just a reused object reference. Here's an example:

```
a = {1=>1, 2=>4, 3=>9, 4=>16}
a.delete(3)                     # 9
# a is now {1=>1, 2=>4, 4=>16}
a.delete(5)                     # nil in this case
a.delete(6) { "not found" }     # "not found"
```

Use `delete_if`, `reject`, or `reject!` in conjunction with the required block to delete all keys for which the block evaluates to `true`. The method `reject` uses a copy of the hash, and `reject!` returns `nil` if no changes were made.

Iterating over a Hash

The `Hash` class has the standard iterator `each`, as is to be expected. It also has `each_key`, `each_pair`, and `each_value` (`each_pair` is an alias for `each`). Here's an example:

```
{"a"=>3,"b"=>2}.each do |key, val|
  print val, " from ", key, "; "    # 3 from a; 2 from b;
end
```

The other two provide only one or the other (the key or the value) to the block:

```
{"a"=>3,"b"=>2}.each_key do |key|
  print "key = #{key};"      # Prints: key = a; key = b;
end

{"a"=>3,"b"=>2}.each_value do |value|
  print "val = #{value};"    # Prints: val = 3; val = 2;
end
```

Inverting a Hash

Inverting a hash in Ruby is trivial with the `invert` method:

```
a = {"fred"=>"555-1122","jane"=>"555-7779"}
b = a.invert
b["555-7779"]     # "jane"
```

Because hashes have unique keys, there is potential for data loss when doing this—duplicate associated values will be converted to a unique key using only one of the associated keys as its value. There is no predictable way to tell which one will be used.

Detecting Keys and Values in a Hash

Determining whether a key has been assigned can be done with `has_key?` or any one of its aliases: `include?`, `key?`, or `member?`. Here's an example:

```
a = {"a"=>1,"b"=>2}
a.has_key? "c"        # false
a.include? "a"        # true
a.key? 2              # false
a.member? "b"         # true
```

You can also use `empty?` to see whether there are any keys at all left in the hash; `length` or its alias `size` can be used to determine how many there are, as shown here:

```
a.empty?        # false
a.length        # 2
```

Alternatively, you can test for the existence of an associated value using `has_value?` or `value?`:

```
a.has_value? 2        # true
a.value? 99           # false
```

Extracting Hashes into Arrays

To convert the entire hash into an array, use the `to_a` method. In the resulting array, keys will be even-numbered elements (starting with 0) and values will be odd-numbered elements of the array:

```
h = {"a"=>1,"b"=>2}
h.to_a        # ["a",1,"b",2]
```

It is also possible to convert only the keys or only the values of the hash into an array:

```
h.keys         # ["a","b"]
h.values       # [1,2]
```

Finally, you can extract an array of values selectively based on a list of keys, using the `indices` method. This works for hashes much as the method of the same name works for arrays (the alias is `indexes`):

```
h = {1=>"one",2=>"two",3=>"three",4=>"four","cinco"=>"five"}
h.indices(3,"cinco",4)     # ["three","five","four"]
h.indexes(1,3)             # ["one","three"]
```

Selecting Key/Value Pairs by Criteria

The `Hash` class mixes in the `Enumerable` module, so you can use `detect` (`find`), `select` (`find_all`), `grep`, `min`, `max`, and `reject` as with arrays.

The `detect` method (whose alias is `find`) finds a single key/value pair. It takes a block (into which the pairs are passed one at a time) and returns the first pair for which the block evaluates to `true`. Here's an example:

```
names = {"fred"=>"jones","jane"=>"tucker",
         "joe"=>"tucker","mary"=>"SMITH"}
# Find a tucker
names.detect {|k,v| v=="tucker" }    # ["joe","tucker"]
# Find a capitalized surname
names.find {|k,v| v==v.upcase }    # ["mary", "SMITH"]
```

Of course, the objects in the hash can be of arbitrary complexity, as can the test in the block, but comparisons between differing types can cause problems.

The `select` method (whose alias is `find_all`) will return multiple matches, as opposed to a single match:

```
names.select {|k,v| v=="tucker" }
# [["joe", "tucker"], ["jane", "tucker"]]
names.find_all {|k,v| k.count("r")>0}
# [["mary", "SMITH"], ["fred", "jones"]]
```

Sorting a Hash

Hashes are by their nature not ordered according to the value of their keys or associated values. In performing a sort on a hash, Ruby converts the hash to an array and then sorts that array. The result is naturally an array:

```
names = {"Jack"=>"Ruby","Monty"=>"Python",
         "Blaise"=>"Pascal", "Minnie"=>"Perl"}
list = names.sort
# list is now:
# [["Blaise","Pascal"], ["Jack","Ruby"],
#  ["Minnie","Perl"], ["Monty","Python"]]
```

Merging Two Hashes

Merging hashes may be useful sometimes. Ruby's `update` method will put the entries of one hash into the target hash, overwriting any previous duplicates:

```
dict = {"base"=>"foundation", "pedestal"=>"base"}
added = {"base"=>"non-acid", "salt"=>"NaCl"}
dict.update(added)
# {"base"=>"non-acid", "pedestal"=>"base", "salt"=>"NaCl"}
```

Creating a Hash from an Array

The easiest way to create a hash from an array is to remember the bracket notation for creating hashes. This works if the array has an even number of elements. Here's an example:

```
array = [2, 3, 4, 5, 6, 7]
hash = Hash[*array]
# hash is now: {2=>3, 4=>5, 6=>7}
```

Finding Difference or Intersection of Hash Keys

Because the keys of a hash can be extracted as a separate array, the extracted arrays of different hashes can be manipulated using the `Array` class methods `&` and `-` to produce the intersection and difference of the keys. The matching values can be generated with the `each` method performed on a third hash representing the merge of the two hashes (to ensure all keys can be found in one place):

```
a = {"a"=>1,"b"=>2,"z"=>3}
    b = {"x"=>99,"y"=>88,"z"=>77}
    intersection = a.keys & b.keys
    difference = a.keys - b.keys
    c = a.dup.update(b)
    inter = {}
    intersection.each {|k| inter[k]=c[k] }
    # inter is {"z"=>77}
    diff={}
    difference.each {|k| diff[k]=c[k] }
    # diff is {"a"=>1, "b"=>2}
```

Using a Hash As a Sparse Matrix

Often we want to make use of an array or matrix that is nearly empty. We could store it in the conventional way, but this is often wasteful of memory. A hash provides a way to store only the values that actually exist.

Here is an example in which we are assuming that the nonexistent values should default to zero:

```
values = Hash.new(0)
values[1001] = 5
values[2010] = 7
values[9237] = 9
x = values[9237]      # 9
y = values[5005]      # 0
```

Obviously in this example, an array would have created over 9,000 unused elements. This may not be acceptable.

What if we want to implement a sparse matrix of two or more dimensions? All we need do is use arrays as the hash keys, like so:

```
cube = Hash.new(0)
cube[[2000,2000,2000]] = 2
z = cube[[36,24,36]]       # 0
```

In this case, we see that literally *billions* of array elements would need to be created if this three-dimensional array is to be complete.

Implementing a Hash with Duplicate Keys

Purists would likely say that if a hash has duplicate keys, it isn't really a hash. We don't want to argue. Call it what you will, there might be occasions when you want a data structure that offers the flexibility and convenience of a hash but allows duplicate key values.

We offer a partial solution here (see Listing 3.9). It is partial for two reasons. First, we have not bothered to implement all the functionality that could be desired, but only a good representative subset. Second, the inner workings of Ruby are such that a hash literal is always an instance of the `Hash` class, and even though we were to inherit from `Hash`, a literal would not be allowed to contain duplicates. (We're thinking about this one further.)

But as long as you stay away from the hash-literal notation, this problem is doable. Here we implement a class that has a "store" (`@store`) that is a simple hash; each value in the hash is an array. We control access to the hash in such a way that when we find ourselves adding a key that already exists, we add the value to the existing array of items associated with that key.

What should `size` return? Obviously, the "real" number of key/value pairs *including* duplicates. Likewise, the `keys` method returns a value potentially containing duplicates. The iterators behave as expected; as with a normal hash, there is no predicting the order in which the pairs will be visited.

Besides the usual `delete`, we have implemented a `delete_pair` method. The former will delete *all* values associated with a key; the latter will delete only the specified key/value pair. (Note that it would have been difficult to make a single method such as `delete(k,v=nil)` because `nil` is a valid value for any hash.)

For brevity, we have not implemented the entire class; frankly, some of the methods, such as `invert`, would require some design decisions as to what their behavior should be. If you're interested, you can flesh out the rest as needed.

LISTING 3.9 Hash with Duplicate Keys

```
class HashDup

  def initialize(*all)
    raise IndexError if all.size % 2 != 0
    @store = {}
    if all[0]  # not nil
      keyval = all.dup
      while !keyval.empty?
        key = keyval.shift
        if @store.has_key?(key)
          @store[key] += [keyval.shift]
        else
          @store[key] = [keyval.shift]
        end
      end
    end
  end

  def store(k,v)
    if @store.has_key?(k)
      @store[k] += [v]
    else
      @store[k] = [v]
    end
  end

  def [](key)
    @store[key]
  end

  def []=(key,value)
    self.store(key,value)
  end
```

LISTING 3.9 Continued

```
  def to_s
    @store.to_s
  end

  def to_a
    @store.to_a
  end

  def inspect
    @store.inspect
  end

  def keys
    result=[]
    @store.each do |k,v|
      result += ([k]*v.size)
    end
    result
  end

  def values
    @store.values.flatten
  end

  def each
    @store.each {|k,v| v.each {|y| yield k,y}}
  end

  alias each_pair each

  def each_key
    self.keys.each {|k| yield k}
  end

  def each_value
    self.values.each {|v| yield v}
  end

  def has_key? k
    self.keys.include? k
  end

  def has_value? v
    self.values.include? v
  end
```

04 0835 CH03 11/16/01 10:19 AM Page 173

LISTING 3.9 Continued

```
    def length
      self.values.size
    end

    alias size length

    def delete k
      val = @store[k]
      @store.delete k
      val
    end

    def delete k,v
      @store[k] -= [v] if @store[k]
      v
    end

    # Other methods omitted here...

  end



  # This won't work... dup key will ignore
  # first occurrence.
  h = {1=>1, 2=>4, 3=>9, 4=>16, 2=>0}

  # This will work...
  h = HashDup.new(1,1, 2,4, 3,9, 4,16, 2,0)

  k = h.keys         # [4, 1, 2, 2, 3]
  v = h.values       # [16, 1, 4, 0, 9]

  n = h.size         # 5

  h.each {|k,v| puts "#{k} => #{v}"}
  # Prints:
  # 4 => 16
  # 1 => 1
  # 2 => 4
  # 2 => 0
  # 3 => 9
```

04 0835 CH03 11/16/01 10:19 AM Page 174

Working with Stacks and Queues

Stacks and queues are the first entities we have discussed that are not strictly built in to Ruby. By this we mean that Ruby does not have `Stack` and `Queue` classes as it has `Array` and `Hash` classes.

And yet, in a way, they are built in to Ruby after all. In fact, the `Array` class implements all the functionality we need to treat an array as a stack or a queue. You'll see this in detail shortly.

A *stack* is a last-in first-out (LIFO) data structure. The traditional everyday example is a stack of cafeteria trays on its spring-loaded platform; trays are added at the top and also taken away from the top.

There is a limited set of operations that can be performed on a stack. These include *push* and *pop* (to add and remove items) at the very least; usually there is a way to test for an empty stack, and there may be a way to examine the top element without removing it. A stack implementation never provides a way to examine an item in the middle of the stack.

You might ask how an array can implement a stack given that array elements may be accessed randomly and stack elements may not. The answer is simple: A stack sits at a higher level of abstraction than an array; it is a stack only so long as you treat it as one. The moment you access an element illegally, it ceases to be a stack.

Of course, you can easily define a `Stack` class so that elements can only be accessed legally. We will show how this is done.

It is worth noting that many algorithms that use a stack also have elegant recursive solutions. The reason for this becomes clear with a moment's reflection. Function or method calls result in data being pushed onto the system stack, and this data is popped upon return. Therefore, a recursive algorithm simply trades an explicit user-defined stack for the implicit system-level stack. Which is better? That depends on how you value readability, efficiency, and other considerations.

A *queue* is a first-in first-out (FIFO) data structure. It is analogous to a group of people standing in line at, for example, a movie theater. Newcomers go to the end of the line, whereas those who have waited longest are the next served. In most areas of programming, these are probably used less often than stacks.

Queues are useful in more real-time environments where entities are processed as they are presented to the system. They are useful in producer/consumer situations (especially where threads or multitasking is involved). A printer queue is a good example; print jobs are added to one end of the queue, and they "stand in line" until they are removed at the other end.

The two basic queue operations are usually called *enqueue* and *dequeue* in the literature. The corresponding instance methods in the `Array` class are called `shift` and `unshift`, respectively.

Note that `unshift` could serve as a companion for `shift` in implementing a stack, not a queue, because `unshift` adds to the same end from which `shift` removes. There are various combinations of these methods that could implement stacks and queues, but we will not concern ourselves with all the variations.

That ends our introductory discussion of stacks and queues. Now let's look at some examples.

Implementing a Stricter Stack

We promised earlier to show how a stack could be made "idiot-proof" against illegal access. We may as well do that now (see Listing 3.10). We present here a simple class that has an internal array and manages access to that array. (There are other ways of doing this—by delegating, for example—but what we show here is simple and works fine.)

LISTING 3.10 Stack

```
class Stack

  def initialize
    @store = []
  end

  def push(x)
    @store.push x
  end

  def pop
    @store.pop
  end

  def peek
    @store.last
  end

  def empty?
    @store.empty?
  end

end
```

We have added one more operations that are not defined for arrays; `peek` will simply examine the top of the stack and return a result without disturbing the stack.

Some of the rest of our examples will assume this class definition.

04 0835 CH03 11/16/01 10:19 AM Page 176

Converting Infix to Postfix

In writing algebraic expressions, we commonly use *infix* notation, with the operator in between the operands. Often it is more convenient to store an expression in *postfix* form, a parenthesis-free form in which the operator follows both the operands. (This is sometimes called *Reverse Polish Notation*.)

In Listing 3.11, we present a simple routine for converting infix to postfix notation using a stack. We make the simplifying assumptions that all operands are lowercase letters and the only operators are `*`, `/`, `+`, and `-`.

LISTING 3.11 Infix to Postfix

```
# Define level of precedence

def level(opr)
  case opr
    when "*", "/"
      2
    when "+", "-"
      1
    when "("
      0
  end
end


# "Main"

infix = "(a+b)*(c-d)/(e-(f-g))"
postfix = ""

stack = Stack.new

infix.each_byte do |sym|
  sym = "" << sym   # Convert to string
  case sym
    when "("
      stack.push sym

    when /[a-z]/
      postfix += sym

    when "*", "/", "+", "-"
      finished = false
```

LISTING 3.11 Continued

```
        until finished or stack.empty?
          if level(sym) > level(stack.peek)
            finished = true
          else
            postfix += stack.pop
          end
        end
        stack.push sym

      when ")"
        while stack.peek != "("
          postfix += stack.pop
        end
        stack.pop  # Get rid of paren on stack
    end
  end

  while !stack.empty?
    postfix += stack.pop
  end

  puts postfix            # Prints "ab+cd-*efg--/"
```

Detecting Unbalanced Punctuation in Expressions

Because of the nature of grouped expressions, such as parentheses and brackets, their validity can be checked using a stack (see Listing 3.12). For every level of nesting in the expression, the stack will grow one level higher; when we find closing symbols, we can pop the corresponding symbol off the stack. If the symbol does not correspond as expected, or if there are symbols left on the stack at the end, we know the expression is not well formed.

LISTING 3.12 Detecting Unbalanced Punctuation

```
  def paren_match str
    stack = Stack.new
    lsym = "{[(<"
    rsym = "}])>"
    str.each_byte do |byte|
      sym = byte.chr
      if lsym.include? sym
        stack.push(sym)
      elsif rsym.include? sym
        top = stack.peek
```

04 0835 CH03 11/16/01 10:19 AM Page 178

LISTING 3.12 Continued

```
          if lsym.index(top) != rsym.index(sym)
            return false
          else
            stack.pop
          end
          # Ignore non-grouped characters...
        end
      end
      # Ensure stack is empty...
      return stack.empty?
    end

    str1 = "Hello (yes, ¨ you) there!"
    str2 = "(((a+b))*((c-d)-(e*f))"
    str3 = "[[(a-(b-c))], [[x,y]]]"

    paren_match str1          # true
    paren_match str2          # false
    paren_match str3          # true
```

Detecting Unbalanced Tags in HTML and XML

The example shown in Listing 3.13 is essentially the same as Listing 3.12. We include it only to give a hint that this task is possible (that is, that a stack is useful for validating HTML and XML).

In the old days, a string was considered, at best, a special case of an array. Your opinion may vary depending on your language background. In Ruby, strings are not arrays; however, it is a tribute to the orthogonality of the language when we see how similar these two examples turned out. This is because, after all, there is a certain isomorphism between strings and arrays. They are both ordered sequences of elements, where in the case of a string, the element is a character.

Because we are talking about stacks and not HTML/XML, we have made a huge truckload of simplifying assumptions here. (If you're interested in real-life HTML and XML examples, refer to later chapters.) First of all, we assume that the text has already been parsed and stuck into an array. Second, we only care about a limited subset of the many tags possible. Third, we ignore the possibility of attributes and values associated with the tags.

In short, this is not a real-life example at all; however, like the previous example, it shows the underlying principle.

04 0835 CH03 11/16/01 10:19 AM Page 179

LISTING 3.13 Detecting Unbalanced Tags

```
def balanced_tags list
  stack = Stack.new
  opening = %w[ <html> <body> <b> <i> <u> <sub> <sup> ]
  closing = %w[ </html> </body> </b> </i> </u> </sub> </sup> ]
  list.each do |word|
    if opening.include? word
      stack.push(word)
    elsif closing.include? word
      top = stack.peek
      if closing.index(top) != opening.index(word)
        return false
      else
        stack.pop
      end
      # Ignore other words
    end
  end
  # Ensure stack is empty...
  return stack.empty?
end

text1 = %w[ <html> <body> This is <b> only </b>
            a test. </body> </html> ]

text2 = %w[ <html> <body> Don't take it <i> too </i>
            seriously... </html> ]

balanced_tags(text1)    # true
balanced_tags(text2)    # false
```

Understanding Stacks and Recursion

As an example of the isomorphism between stack-oriented algorithms and recursive algorithms, we will take a look at the classic “Tower of Hanoi” problem.

According to legend, there is a Buddhist temple somewhere in the Far East, where monks have the sole task of moving disks from one pole to another while obeying certain rules about the moves they can make. There were originally 64 disks on the first pole; when they finish the task, the world will come to an end.

As an aside, we like to dispel myths when we can. It seems that in reality, this puzzle originated with the French mathematician Edouard Lucas in 1883 and has no actual basis in eastern culture. What’s more, Lucas himself named the puzzle the “Tower of Hanoi” (in the singular).

04 0835 CH03 11/16/01 10:19 AM Page 180

So if you were worried about the world ending, don't worry on that account. Anyway, 64 disks would take 2^{64}-1 moves. A few minutes with a calculator will reveal that those monks would be busy for millions of years.

But on to the rules of the game. (We'll explain this even though every first-year computer science student in the world has already seen the puzzle.) We have a pole with a certain number of varying-sized disks stacked on it; call this the *source pole*. We want to move all these disks to the *destination pole*, using a third pole (called the *auxiliary pole*) as an intermediate resting place. The catch is that you can only move one disk at a time, and you cannot ever place a larger disk onto a smaller one.

The following example uses a stack to solve the problem. We use only three disks here because 64 would occupy a computer for centuries:

```
def towers2(list)
  while !list.empty?
    n, src, dst, aux = list.pop
    if n == 1
      puts "Move disk from #{src} to #{dst}"
    else
      list.push [n-1, aux, dst, src]
      list.push [1, src, dst, aux]
      list.push [n-1, src, aux, dst]
    end
  end
end


list = []
list.push([3, "a", "c", "b"])

towers2(list)
```

Here's the output that's produced:

```
Move disk from a to c
Move disk from a to b
Move disk from c to b
Move disk from a to c
Move disk from b to a
Move disk from b to c
Move disk from a to c
```

Of course, the classic solution to this problem is recursive. As we already pointed out, the close relationship between the two algorithms is no surprise because recursion implies the use of an invisible system-level stack. Here's an example:

```
def towers(n, src, dst, aux)
  if n==1
    puts "Move disk from #{src} to #{dst}"
  else
    towers(n-1, src, aux, dst)
    towers(1, src, dst, aux)
    towers(n-1, aux, dst, src)
  end
end

towers(3, "a", "c", "b")
```

The output produced here is the same. And it may interest you to know that we tried commenting out the output statements and comparing the runtimes of these two methods. Don't tell anyone, but the recursive version is twice as fast.

Implementing a Stricter Queue

We define a queue here in much the same way we defined a stack earlier. If you want to protect yourself from accessing such a data structure in an illegal way, we recommend this practice (see Listing 3.14).

LISTING 3.14 A Stricter Queue

```
class Queue

  def initialize
    @store = []
  end

  def enqueue(x)
    @store << x
  end

  def dequeue
    @store.shift
  end

  def peek
    @store.first
  end
```

LISTING 3.14 Continued

```
    def length
      @store.length
    end

    def empty?
      @store.empty?
    end

  end
```

We should mention that there is a `Queue` class in the `thread` library that works very well in threaded code.

A Token Queue Example: Traffic Light Simulation

We offer here a fairly contrived example of using a queue. This code will simulate the arrival of cars at a traffic light and store the arrival times in four queues. At the end, it prints some (presumably meaningful) statistics about the queue lengths and wait times.

A number of simplifying assumptions have been made. Time is granularized at the level of one second. There are no threads involved; all car movements are serialized in a reasonable way. Cars turn neither left nor right, they never go through a yellow or red light, and so on. The code is shown in Listing 3.15.

LISTING 3.15 Traffic Light Simulation with a Queue

```
  #
  # Program: Traffic light simulation
  #          (Queue example)
  #
  # The traffic light has this behavior:
  # Green north/south for 40 seconds
  # Pause 2 seconds
  # Green east/west for 45 seconds
  # Pause 2 seconds
  # Repeat
  #
  # The traffic behaves this way:
  # A northbound car arrives at the traffic light
  #   every 3 seconds;
  # Southbound, every 5 seconds;
  # Eastbound, every 4 seconds;
  # Westbound, every 6 seconds.
  # All times are approximate (random).
```

LISTING 3.15 Continued

```
# Assume no cars turn at the light.
#
# Cars pass through the light at a rate of
# one per second.
#
# Let's run for 8900 seconds (100 full cycles or
# more than two hours) and answer these questions:
# How long on the average is each line of cars
# when the light turns green? What is the average
# wait time in seconds? What is the longest wait
# time?
#


# Direction constants

NORTH, SOUTH, EAST, WEST = 0, 1, 2, 3
dirs = %w[North South East West]

# Probabilities for car arriving
# from each direction:

p = Array.new(4)
p[NORTH] = 1.0/3.0
p[SOUTH] = 1.0/5.0
p[EAST]  = 1.0/4.0
p[WEST]  = 1.0/6.0

# Queues:

waiting = Array.new(4)
waiting[NORTH] = Queue.new
waiting[SOUTH] = Queue.new
waiting[EAST]  = Queue.new
waiting[WEST]  = Queue.new

lengths = [0, 0, 0, 0]  # How long is queue
                        # when light turns green?
greens  = [0, 0, 0, 0]  # How many times did
                        # light turn green?
times   = [0, 0, 0, 0]  # How long did cars wait?
ncars   = [0, 0, 0, 0]  # Count cars through light.
maxtime = [0, 0, 0, 0]  # Max wait time?

# Looping...
```

04 0835 CH03 11/16/01 10:19 AM Page 184

LISTING 3.15 Continued

```
time=0
while time < 8900

  change = true  # Light changed
  for time in time..time+40          # North/south green
    # Enqueue all arrivals
    for dir in NORTH..WEST
      waiting[dir].enqueue(time) if rand < p[dir]
    end

    # Record queue lengths, counts
    if change
      for dir in NORTH..SOUTH
        lengths[dir] += waiting[dir].length
        greens[dir] += 1
      end
      change = false
    end

    # N/S can leave, one per second...
    for dir in NORTH..SOUTH
      if !waiting[dir].empty?
        car = waiting[dir].dequeue
        wait = time - car
        ncars[dir] += 1
        times[dir] += wait
        maxtime[dir] = [maxtime[dir],wait].max
      end
    end
  end

  for time in time..time+2           # Yellow/red
    # Nothing happens...
  end

  change = true  # Light changed
  for time in time..time+45          # East/west green
    # Enqueue all arrivals
    for dir in NORTH..WEST
      waiting[dir].enqueue(time) if rand < p[dir]
    end

    # Record queue lengths, counts
    if change
      for dir in EAST..WEST
```

LISTING 3.15 Continued

```
          lengths[dir] += waiting[dir].length
          greens[dir] += 1
        end
        change = false
      end

      # N/S can leave, one per second...
      for dir in EAST..WEST
        if !waiting[dir].empty?
          car = waiting[dir].dequeue
          wait = time - car
          ncars[dir] += 1
          times[dir] += wait
          maxtime[dir] = [maxtime[dir],wait].max
        end
      end
    end

    for time in time..time+2           # Yellow/red
      # Nothing happens...
    end

  end

  # Display results...

  puts "Average queue lengths:"
  for dir in NORTH..WEST
    printf "  %-5s %6.1f\n", dirs[dir],
           lengths[dir]/greens[dir].to_f
  end

  puts "Max wait times:"
  for dir in NORTH..WEST
    printf "  %-5s %4d\n", dirs[dir],
           maxtime[dir]
  end

  puts "Average wait times:"
  for dir in NORTH..WEST
    printf "  %-5s %6.1f\n", dirs[dir],
           times[dir]/ncars[dir].to_f
  end
```

04 0835 CH03 11/16/01 10:19 AM Page 186

Here is the output this example produces (which will vary because of the use of the pseudorandom number generator `rand`):

```
Average queue lengths:
  North   15.6
  South    9.5
  East    10.8
  West     7.3
Max wait times:
  North   51
  South   47
  East    42
  West    42
Average wait times:
  North   19.5
  South   16.2
  East    13.7
  West    12.9
```

You may at once see a dozen ways in which this program could be improved. However, it serves its purpose, which is to illustrate a simple queue.

Working with Trees

I think that I shall never see
A poem as lovely as a tree….

—[Alfred] Joyce Kilmer, "Trees"

A *tree* in computer science is a relatively intuitive concept (except that it is usually drawn with the "root" at the top and the "leaves" at the bottom). This is because we are familiar with so many kinds of hierarchical data in everyday life—from the family tree, to the corporate org chart, to the directory structures on our hard drives.

The terminology of trees is rich but easy to understand. Any item in a tree is a *node*; the first or topmost node is the *root*. A node may have *descendants* that are below it, and the immediate descendants are called *children*. Conversely, a node may also have a *parent* (only one) and *ancestors*. A node with no child nodes is called a *leaf*. A *subtree* consists of a node and all its descendants. To travel through a tree (for example, to print it out) is called *traversing the tree*.

We will look mostly at binary trees, although in practice a node can have any number of children. You will see how to create a tree, populate it, and traverse it. Also, we will look at a few real-life tasks that use trees.

We should mention here that in many languages, such as C and Pascal, trees are implemented using true address pointers. However, in Ruby (as in Java, for instance), we don't use pointers; object references work just as well or even better.

Implementing a Binary Tree

There is more than one way to implement a binary tree in Ruby. For example, we could use an array to store the values. Here, we use a more traditional approach, coding much as we would in C, except that pointers are replaced with object references.

What is required in order to describe a binary tree? Well, each node needs an attribute of some kind for storing a piece of data. Each node also needs a pair of attributes for referring to the left and right subtrees under that node.

We also need a way to insert into the tree and a way of getting information out of the tree. A pair of methods will serve these purposes.

The first tree we'll look at will implement these methods in a slightly unorthodox way. We will expand on the `Tree` class in later examples.

A tree is, in a sense, defined by its insertion algorithm and by how it is traversed. In this first example, shown in Listing 3.16, we define an `insert` method that inserts in a *breadth-first* fashion (that is, top to bottom and left to right). This guarantees that the tree grows in depth relatively slowly and is always balanced. Corresponding to the `insert` method, the `traverse` iterator will iterate over the tree in the same breadth-first order.

LISTING 3.16 Breadth-First Insertion and Traversal in a Tree

```
class Tree

attr_accessor :left
attr_accessor :right
attr_accessor :data

def initialize(x=nil)
  @left = nil
  @right = nil
  @data = x
end

def insert(x)
  list = []
  if @data == nil
    @data = x
  elsif @left == nil
```

04 0835 CH03 11/16/01 10:19 AM Page 188

LISTING 3.16 Continued

```
      @left = Tree.new(x)
    elsif @right == nil
      @right = Tree.new(x)
    else
      list << @left
      list << @right
      loop do
        node = list.shift
        if node.left == nil
          node.insert(x)
          break
        else
          list << node.left
        end
        if node.right == nil
          node.insert(x)
          break
        else
          list << node.right
        end
      end
    end
  end

  def traverse()
    list = []
    yield @data
    list << @left if @left != nil
    list << @right if @right != nil
    loop do
      break if list.empty?
      node = list.shift
      yield node.data
      list << node.left if node.left != nil
      list << node.right if node.right != nil
    end
  end

  end


  items = [1, 2, 3, 4, 5, 6, 7]

  tree = Tree.new
```

LISTING 3.16 Continued

```
items.each {|x| tree.insert(x)}

tree.traverse {|x| print "#{x} "}
print "\n"

# Prints "1 2 3 4 5 6 7 "
```

This kind of tree, as defined by its insertion and traversal algorithms, is not especially interesting. However, it does serve as an introduction and something on which we can build.

Sorting Using a Binary Tree

For random data, using a binary tree is a good way to sort. (Although in the case of already-sorted data, it degenerates into a simple linked list.) The reason, of course, is that with each comparison, we are eliminating half the remaining alternatives as to where we should place a new node.

Although it might be fairly rare to sort using a binary tree nowadays, it can't hurt to know how. The code in Listing 3.17 builds on the previous example.

LISTING 3.17 Sorting with a Binary Tree

```
class Tree

  # Assumes definitions from
  # previous example...

  def insert(x)
    if @data == nil
      @data = x
    elsif x <= @data
      if @left == nil
        @left = Tree.new x
      else
        @left.insert x
      end
    else
      if @right == nil
        @right = Tree.new x
      else
        @right.insert x
      end
    end
  end
```

04 0835 CH03 11/16/01 10:19 AM Page 190

LISTING 3.17 Continued

```
    def inorder()
      @left.inorder {|y| yield y} if @left != nil
      yield @data
      @right.inorder {|y| yield y} if @right != nil
    end

    def preorder()
      yield @data
      @left.preorder {|y| yield y} if @left != nil
      @right.preorder {|y| yield y} if @right != nil
    end

    def postorder()
      @left.postorder {|y| yield y} if @left != nil
      @right.postorder {|y| yield y} if @right != nil
      yield @data
    end

  end

  items = [50, 20, 80, 10, 30, 70, 90, 5, 14,
           28, 41, 66, 75, 88, 96]

  tree = Tree.new

  items.each {|x| tree.insert(x)}

  tree.inorder {|x| print x, " "}
  print "\n"
  tree.preorder {|x| print x, " "}
  print "\n"
  tree.postorder {|x| print x, " "}
  print "\n"
```

Using a Binary Tree As a Lookup Table

Suppose we have a tree already sorted. Traditionally, this has made for a good lookup table; for example, a balanced tree of a million items would take no more than 20 comparisons (the depth of the tree or log base 2 of the number of nodes) to find a specific node. For this to be useful, we assume that the data for each node is not just a single value but has a key value and other information associated with it.

In most if not all situations, a hash or even an external database table will be preferable. However, we present this code to you anyhow (see Listing 3.18).

LISTING 3.18 Searching a Binary Tree

```
class Tree

  # Assumes definitions
  # from previous example...

  def search(x)
    if self.data == x
      return self
    else
      ltree = left != nil ? left.search(x) : nil
      return ltree if ltree != nil
      rtree = right != nil ? right.search(x) : nil
      return rtree if rtree != nil
    end
    nil
  end

end

keys = [50, 20, 80, 10, 30, 70, 90, 5, 14,
        28, 41, 66, 75, 88, 96]

tree = Tree.new

keys.each {|x| tree.insert(x)}

s1 = tree.search(75)   # Returns a reference to the node
                       # containing 75...

s2 = tree.search(100)  # Returns nil (not found)
```

Converting a Tree to a String or Array

The same old tricks that allow us to traverse a tree will allow us to convert it to a string or array if we wish, as shown in Listing 3.19. Here, we assume an *inorder* traversal, although any other kind could be used.

LISTING 3.19 Converting a Tree to a String or Array

```
class Tree

  # Assumes definitions from
  # previous example...

  def to_s
    "[" +
    if left then left.to_s + "," else "" end +
    data.inspect +
    if right then "," + right.to_s else "" end + "]"
  end

  def to_a
    temp = []
    temp += left.to_a if left
    temp << data
    temp += right.to_a if right
    temp
  end

end

items = %w[bongo grimace monoid jewel plover nexus synergy]

tree = Tree.new
items.each {|x| tree.insert x}

str = tree.to_s * ","
# str is now "bongo,grimace,jewel,monoid,nexus,plover,synergy"
arr = tree.to_a
# arr is now:
# ["bongo",["grimace",[["jewel"],"monoid",[["nexus"],"plover",
#  ["synergy"]]]]]
```

Note that the resulting array is as deeply nested as the depth of the tree from which it came. You can, of course, use `flatten` to produce a non-nested array.

Storing an Infix Expression in a Tree

Here is another little contrived problem illustrating how a binary tree might be used (see Listing 3.20). We are given a prefix arithmetic expression and want to store it in standard infix form in a tree. (This is not completely unrealistic because the Ruby interpreter itself stores expressions in a tree structure, although it is a couple of orders of magnitude greater in complexity.)

We define a "standalone" method called `addnode` that will add a node to a tree in the proper place. The result will be a tree in which every leaf is an operand and every non-leaf node is an operator. We also define a new `Tree` method called `infix`, which will traverse the tree in order and act as an iterator. One twist is that it adds in parentheses as it goes, because prefix form is "parenthesis free" but infix form is not. The output would look more elegant if only necessary parentheses were added, but we added them indiscriminately to simplify the code.

LISTING 3.20 Storing an Infix Expression in a Tree

```
class Tree

  # Assumes definitions from
  # previous example...

  def infix()
    if @left != nil
      flag = %w[* / + -].include? @left.data
      yield "(" if flag
      @left.infix {|y| yield y}
      yield ")" if flag
    end
    yield @data
    if @right != nil
      flag = %w[* / + -].include? @right.data
      yield "(" if flag
      @right.infix {|y| yield y} if @right != nil
      yield ")" if flag
    end
  end

end

def addnode(nodes)
  node = nodes.shift
  tree = Tree.new node
  if %w[* / + -].include? node
    tree.left  = addnode nodes
    tree.right = addnode nodes
  end
  tree
end


prefix = %w[ * + 32 * 21 45 - 72 + 23 11 ]
```

04 0835 CH03 11/16/01 10:19 AM Page 194

LISTING 3.20 Continued

```
    tree = addnode prefix

    str = ""
    tree.infix {|x| str += x}
    # str is now "(32+(21*45))*(72-(23+11))"
```

Additional Notes on Trees

We'll mention a few more notes on trees here. First of all, a tree is a special case of a graph (as you will see shortly); in fact, it is a *directed acyclic graph* (DAG). Therefore, you can learn more about trees by researching graph algorithms in general.

There is no reason that a tree should necessarily be binary; this is a convenient simplification that frequently makes sense. However, it is conceivable to define a *multiway tree* in which each node is not limited to two children but may have an arbitrary number. In such a case, you would likely want to represent the child node pointers as an array of object references.

A *B-tree* is a specialized form of multiway tree. It is an improvement over a binary tree in that it is always balanced (that is, its depth is minimal), whereas a binary tree in a degenerate case can have a depth that is equal to the number of nodes it has. There is plenty of information on the Web and in textbooks if you need to learn about B-trees. Also, the principles we've applied to ordinary binary trees can be extended to B-trees as well.

A *red-black tree* is a specialized form of binary tree in which each node has a color (red or black) associated with it. In addition, each node has a pointer back to its parent (meaning that it is arguably not a tree at all because it isn't truly acyclic). A red-black tree maintains its balance through rotations of its nodes; that is, if one part of the tree starts to get too deep, the nodes can be rearranged so that depth is minimized (and in-order traversal ordering is preserved). The extra information in each node aids in performing these rotations.

Another tree that maintains its balance in spite of additions and deletions is the *AVL tree*. This structure is named for its discoverers, the two Russian researchers Adel'son-Vel'skii and Landis. An AVL tree is a binary tree that uses slightly more sophisticated insertion and deletion algorithms to keep the tree balanced. It performs rotation of subtrees similar to that done for red-black trees.

All these and more are potentially useful tree structures. If you need more information, search the Web or consult any book on advanced algorithms.

Working with Graphs

A *graph* is a collection of nodes that interconnect with each other arbitrarily. (A tree is a special case of a graph.) We will not get deeply into graphs because the theory and terminology can have a steep learning curve. Before long, we would find ourselves wandering out of the field of computer science entirely and into the province of mathematicians.

Yet, graphs do have many practical applications. Consider any ordinary highway map, with highways connecting cities, or consider a circuit diagram. These are both best represented as graphs. A computer network can be thought of in terms of graph theory, whether it is a LAN of a dozen systems or the Internet itself with its countless millions of nodes.

When we say "graph," we usually mean an *undirected graph*. In simplistic terms, this is a graph in which the connecting lines don't have arrows; two nodes are either connected or they are not. By contrast, a *directed graph* or *digraph* can have "one-way streets;" just because node *x* is connected to node *y* doesn't mean that the reverse is true. (A node is also commonly called a *vertex*.) Finally, a weighted graph has connections (or edges) that have weights associated with them; these weights may express, for instance, the "distance" between two nodes. We won't go beyond these basic kinds of graphs; if you're interested in learning more, you can refer to the numerous references in computer science and mathematics.

In Ruby, as in most languages, a graph can be represented in multiple ways—for example, as a true network of interconnected objects or as a matrix storing the set of edges in the graph. We will look at both of these as we show a few practical examples of manipulating graphs.

Implementing a Graph As an Adjacency Matrix

The example here builds on two previous examples. In Listing 3.21, we implement an undirected graph as an adjacency matrix, using the `ZArray` class to make sure new elements are zero and inheriting from `TriMatrix` to get a *lower triangular matrix* form.

Note that in the kind of graph we are implementing here, a node cannot be connected to itself, and two nodes can be connected by only one edge.

We provide a way to specify edges initially by passing pairs into the constructor. We also provide a way to add and remove edges and detect the presence of edges. The `vmax` method will return the highest-numbered vertex in the graph. The `degree` method will find the *degree* of the specified vertex (that is, the number of edges that connect to it).

Finally, we provide two iterators, `each_vertex` and `each_edge`. These will iterate over vertexes and edges, respectively.

LISTING 3.21 Adjacency Matrix

```
class LowerMatrix < TriMatrix

  def initialize
    @store = ZArray.new
  end

end


class Graph

  def initialize(*edges)
    @store = LowerMatrix.new
    @max = 0
    for e in edges
      e[0], e[1] = e[1], e[0] if e[1] > e[0]
      @store[e[0],e[1]] = 1
      @max = [@max, e[0], e[1]].max
    end
  end

  def [](x,y)
    if x > y
      @store[x,y]
    elsif x < y
      @store[y,x]
    else
      0
    end
  end

  def []=(x,y,v)
    if x > y
      @store[x,y]=v
    elsif x < y
      @store[y,x]=v
    else
      0
    end
  end

  def edge? x,y
    x,y = y,x if x < y
    @store[x,y]==1
  end
```

04 0835 CH03 11/16/01 10:19 AM Page 197

LISTING 3.21 Continued

```
  def add x,y
    @store[x,y] = 1
  end

  def remove x,y
    x,y = y,x if x < y
    @store[x,y] = 0
    if (degree @max) == 0
      @max -= 1
    end
  end

  def vmax
    @max
  end

  def degree x
    sum = 0
    0.upto @max do |i|
      sum += self[x,i]
    end
    sum
  end

  def each_vertex
    (0..@max).each {|v| yield v}
  end

  def each_edge
    for v0 in 0..@max
      for v1 in 0..v0-1
        yield v0,v1 if self[v0,v1]==1
      end
    end
  end

end


mygraph = Graph.new([1,0],[0,3],[2,1],[3,1],[3,2])

# Print the degrees of all the vertices: 2 3 3 2
mygraph.each_vertex {|v| puts mygraph.degree(v)}
```

LISTING 3.21 Continued

```
# Print the list of edges
mygraph.each_edge do |a,b|
  puts "(#{a},#{b})"
end

# Remove a single edge
mygraph.remove 1,3

# Print the degrees of all the vertices: 2 2 2 2
mygraph.each_vertex {|v| p mygraph.degree v}
```

Determining Whether a Graph Is Fully Connected

Not all graphs are fully connected. That is, sometimes “you can’t get there from here” (there may be vertexes that are unreachable from other vertexes no matter what path you try). Connectivity is an important property of a graph to be able to assess, telling whether the graph is “of one piece.” If it is, every node is ultimately reachable from every other node.

We won’t explain the algorithm; you can refer to any discrete math book. However, we offer the Ruby method in Listing 3.22.

LISTING 3.22 Determining Whether a Graph Is Fully Connected

```
class Graph

  def connected?
    x = vmax
    k = [x]
    l = [x]
    for i in 0..@max
      l << i if self[x,i]==1
    end
    while !k.empty?
      y = k.shift
      # Now find all edges (y,z)
      self.each_edge do |a,b|
        if a==y || b==y
          z = a==y ? b : a
          if !l.include? z
            l << z
            k << z
          end
        end
```

04 0835 CH03 11/16/01 10:19 AM Page 199

LISTING 3.22 Continued

```
        end
      end
      if l.size < @max
        false
      else
        true
      end
    end

  end


  mygraph = Graph.new([0,1], [1,2], [2,3], [3,0], [1,3])

  puts mygraph.connected?     # true

  puts mygraph.euler_path?    # true

  mygraph.remove 1,2
  mygraph.remove 0,3
  mygraph.remove 1,3

  puts mygraph.connected?     # false

  puts mygraph.euler_path?    # false
```

A refinement of this algorithm could be used to determine the set of all connected components (or *cliques*) in a graph that is not overall fully connected. We won't do this here.

Determining Whether a Graph Has an Euler Circuit

There is no branch of mathematics, however abstract, which may not some day be applied to phenomena of the real world.

—Nikolai Lobachevsky

Sometimes we want to know whether a graph has an *Euler circuit*. This term comes from the mathematician Leonhard Euler who essentially founded the field of topology by dealing with a particular instance of this question. (A graph of this nature is sometimes called a *unicursive graph* because it can be drawn without lifting the pen from the paper or retracing.)

04 0835 CH03 11/16/01 10:19 AM Page 200

In the German town of Konigsberg is an island in the middle of a river (near where the river splits into two parts). Seven bridges crisscross at various places between opposite shores and the island. The townspeople wondered whether it was possible to make a walking tour of the city in such a way that you would cross each bridge exactly once and return to your starting place. In 1735, Euler proved that this was impossible. This, then, is not just a classic problem, but the *original* graph theory problem.

And, as with many things in life, once you know the answer, it is easy. It turns out that for a graph to have an Euler circuit, it must possess only vertexes with *even degrees*. Here, we add a little method to check that property:

```
class Graph

  def euler_circuit?
    return false if !connected?
    for i in 0..@max
      return false if degree(i) % 2 != 0
    end
    true
  end

end


mygraph = Graph.new([1,0],[0,3],[2,1],[3,1],[3,2])

flag1 =  mygraph.euler_circuit?    # false

mygraph.remove 1,3

flag2 =  mygraph.euler_circuit?    # true
```

Determining Whether a Graph Has an Euler Path

An *Euler path* is not quite the same as an Euler circuit. The word *circuit* implies that you must return to your starting point; with a *path*, we are really only concerned with visiting each edge exactly once. The following code fragment illustrates the difference:

```
class Graph

  def euler_path?
    return false if !connected?
    odd=0
    each_vertex do |x|
      if degree(x) % 2 == 1
```

```
          odd += 1
        end
      end
      odd <= 2
    end

  end


  mygraph = Graph.new([0,1],[1,2],[1,3],[2,3],[3,0])

  flag1 =  mygraph.euler_circuit?    # false
  flag2 =  mygraph.euler_path?       # true
```

Hints for More Complex Graphs

It would be possible to write an entire book about graph algorithms. There are many good ones out there already, and we are certainly not going to range that far outside our realm of expertise.

However, we will offer a few hints for dealing with more sophisticated graphs. These hints should get you started if you need to tackle a more advanced problem.

Suppose you want a directed graph rather than an undirected one. Just because vertex *x* points to vertex *y* doesn't mean the converse is true. You should no longer use a lower triangular matrix form but rather a full-fledged two-dimensional array (see the section "Using Multidimensional Arrays"). You may still find `ZArray` useful (see the section "Establishing a Default Value for New Array Elements"). You will likely want to implement not just a `degree` method but rather a pair of them; in a directed graph, a vertex has an *in-degree* and an *out-degree*.

Suppose you have a directed graph in which a node or vertex is allowed to point to itself. Now you have potential nonzero numbers in the diagonal of your matrix rather than just zeroes. Be sure your code doesn't disallow access to the diagonal.

Suppose you want a *weighted graph*, where each edge has a weight associated with it. Now you would store the weight itself in the array rather than just a `1` or `0` (present or absent).

What about a *multigraph*, in which there can be multiple connections (edges) between the same pair of vertexes? If it is undirected, a lower triangular matrix will suffice, and you can store the number of edges in each element (rather than just a `1` or `0`). If it is directed, you will need a two-dimensional array, and you can still store the number of edges in each respective element.

What about bizarre combinations of these? For example, it is certainly conceivable to have a weighted, directed multigraph (and if you have a valid everyday need for one, let us know about your application). In this case, you would need a more complex data structure. One possibility would be to store a small array in each element of the matrix.

For example, suppose vertex 3 has five edges connecting it with vertex 4; then element `3,4` of the adjacency matrix might store an array containing the associated weights.

The possibilities are endless and are beyond the scope of this book.

Summary

In this chapter, we've taken a good look at arrays, hashes, and more complex data structures. You've seen some similarities between arrays and hashes (many of which are due to the fact that both mix in `Enumerable`) as well as some differences. We've looked at converting between arrays and hashes, and you've learned some interesting ways of extending their standard behavior.

Where more advanced data structures are concerned, you've seen examples of inheriting from an existing class and examples of limited delegation by encapsulating an instance of another class. You've seen ways to store data creatively, ways to make use of various data structures, and how to create iterators for these classes.

In the next chapter, we are again covering the topic of manipulation of data. However, where we have so far been concerned with objects stored in memory, we will now be looking at secondary storage—working with files (and I/O in general), databases, and persistent objects.