16 Maps And Linked Lists

Overview

The program you write in a text file is called the source code. You then have the compiler translate the source code to object code, stored in another file. This object code is some computer chip's machine code (for Java, it is the machine code of the Java Virtual Machine). By contrast, an **interpreter** translates one line of source code at a time to object code in RAM, executing the result before going on to the next line.

You have been hired to develop a user-friendly interpreter for the Scheme programming language. In the process of developing this software, you will become better at working with linked lists using recursive logic, and you will see the basis for several different implementations of the Map class from the Sun standard library.

This chapter requires that you have a solid understanding of arrays and have learned about recursion, Exceptions, and interfaces. No new Java language features are in this chapter or in any later chapter.

- Sections 16.1-16.2 give a partial development of the Scheme interpreter with four classes implementing the Item interface.
- Section 16.3 presents the standard Sun library Map interface and a home-grown subset of it called Mapping. It also discusses the Iterator interface.
- Sections 16.4-16.6 show how a Mapping can be implemented using a partially-filled array and/or a linked list, including the nested class implementing Iterator.
- Section 16.7 introduces the concept of a hash table to implement a Mapping.
- Section 16.8 develops the Scheme interpreter further using a Mapping.

16.1 Basic Scheme Language Elements

The client has the manual for the Scheme language, which is an exact specification in all details. The client wants precisely what is described there with certain exceptions, such as the error messages being far more helpful than what the manual specifies. Your job is to develop the interpreter the way the manual describes, to a substantial extent, before considering modifications.

The manual says that the Scheme interpreter waits for the user to enter a line of input and then prints a response. It gives these examples of inputs and responses (==> indicates the response for the given user input):

```
6.40                ==> 6.4
"Hello"             ==> Hello
"He \"finks\" out"  ==> He "finks" out
pi                  ==> 3.14159
```

These examples illustrate the **atomic** kinds of things in the Scheme language: a real number, a TextString, and a WordUnit.

- A **real** number is the same as Java's ordinary double value: one or more digits, with an optional decimal point before any one of them, and an optional negative sign at the very beginning. Integers are not a separate class of values from reals.
- A **TextString** is the same as Java's String value, beginning and ending with a quote. You may use `\"` to indicate a quote mark inside the string, `\n` to indicate a newline, `\\` to represent a backslash, etc.
- A **WordUnit** is most anything else that does not involve parentheses, apostrophes, quotes, or spaces. WordUnits are used as names of variables and functions.

Function calls

A list of atomic items inside a matched pair of parentheses has a value that can be computed. Here are some examples of addition, multiplication, subtraction, and finding the square root (we slightly simplify the true Scheme description):

```
(+ 5 3)        ==> 8
(* 5 3)        ==> 15
(- 5 3)        ==> 2
(sqrt 9)       ==> 3
```

In most cases, a WordUnit at the beginning of the list indicates an operation to be carried out on the rest of the values in the list. Each such **list** is essentially a method call, where the method name is `+` or `sqrt` or whatever the initial WordUnit is, and the other values in the list are the actual parameters of the method call.

A method is called a **function** in Scheme, and actual parameters are called **arguments**. Every Scheme function returns a value, i.e., there are no void methods in Scheme.

Assignment to variables

Scheme allows you to assign values to WordUnits, so you can use them like variables in Java. The assignment WordUnit `set!` is the **keyword** used for this purpose:

```
(set! x 5)     ==> x is defined
x              ==> 5
(/ 20 x)       ==> 4
val_3          ==> oh-oh: val_3 has no value
(set! val_3 x) ==> val_3 is defined
val_3          ==> 5
```

From these examples you can see that the `set!` command requires two parameters. It assigns to the first parameter, a WordUnit, the value of the second parameter. The value returned by the `set!` command is just `whatever is defined`. Some WordUnits have predefined values (such as `pi` and `true` and `false`). A WordUnit that is not predefined by Scheme and not yet defined by the user produces an **oh-oh error message** when you ask the interpreter to find its value.

The Scheme interpreter accepts one expression at a time containing either an atomic thing (WordUnit, TextString, or real) or a list (in parentheses), evaluates it, and then prints its value. If the expression the user enters is a `set!` command, the interpreter simply adds the variable (`x` and `val_3` in the preceding examples) to its list of defined variables, along with the value defined for it, and returns `whatever is defined`. Here are some examples of illegal expressions:

```
(6 2)    ==> oh-oh: a function call must start with a WordUnit
(bob 9)     ==> oh-oh: bob is not a function name
(sqrt 4 9)  ==> oh-oh: sqrt has too many parameters
```

Only a WordUnit can be used as a function name, and then only if it has been previously defined as a function. Also, you have to have exactly the number of parameters that the function definition requires. When the Scheme interpreter sees that the line the user entered is wrong, it stops processing the line and produces an oh-oh message.

The things on a list can be called its **elements**. A list can contain lists as its elements as well as atomic things (in other words, an argument of a function call can be a function call, just as in Java). The value of a list whose first element is a function name is the result of applying the function to the values of the rest of the elements of the list:

```
(* 3 (+ 5 2))              ==> 21
(sqrt (/ 50 2))           ==> 5
(+ (sqrt 9) (* 2 4))      ==> 11
```

Defining new functions

But of course, this is not programming, it is just using a fancy calculator. Programming requires that you have a way of writing new functions. Scheme provides that with the `define` keyword. The following Scheme coding defines functions for the cube of a value and the average of two values and illustrates their use:

```
(define (cube p)  (* p (* p p)))  ==> cube is defined
(cube 3)                          ==> 27
(cube val_3)                      ==> 125
(define (avg x y) (/ (+ x y) 2))  ==> avg is defined
(avg 6 13)                        ==> 9.5
```

The general format of a **function definition** is a list of three elements: The WordUnit `define`, then the function heading (a list of WordUnits), then the function body (any legal expression to be evaluated). For instance, a function definition with two parameters always has this form:

```
(define (functionName par1 par2)  anExpressionToEvaluateLater)
```

This list is <u>not</u> a function call. If it were, the interpreter would try to evaluate the second and third elements on the list. Instead, the interpreter stores the two elements away unevaluated for now, so it can use them to evaluate any number of calls of that function later in the program. Similarly,

```
(set!  variableName  anExpressionToEvaluateNow)
```

is not a function call. This non-evaluation of the expressions in a list beginning with `set!` or `define` is why `set!` and `define` are keywords.

Comparison with Java

Pause a moment to admire the simplicity and beauty of this programming language (which is taught in the first computer science course in many universities). The function call `(sqrt p)` corresponds to a Java method call `sqrt(p)` and the function call `(avg 6 13)` corresponds to a Java method call `avg(6,13)`. It may seem strange that addition is `(+ 2 3)` instead of `2 + 3` but it provides uniformity and simplicity: `+` is considered a function name just as are `sqrt` and `avg`. Moreover:

- Scheme has no other punctuation, such as semicolons, brackets, or periods.
- Precedence of operators does not even arise in Scheme.
- The function definition describes the value to be returned, so you do not have to say `return`.
- You do not bother with classes or public vs. private or instance vs. class: All variables and methods are public class methods and variables of one giant class, which you need not explicitly define (actually, this is a big drawback for advanced programming situations, since you do not have encapsulation).
- Scheme has an if-statement, a looping mechanism, boolean functions, etc., which we do not discuss here.
- www.htdp.org has a complete online textbook for Scheme and a free interpreter.

Exercise 16.1 Define a Scheme function that returns the Fahrenheit temperature corresponding to a given Centigrade temperature. Give an example of its use, as shown in the text for `cube` and `avg`.

Exercise 16.2 Define a Scheme function that returns the length of the hypotenuse of a right triangle given two parameters, the two sides of the right triangle. Give an example of its use.
Exercise 16.3* Define a Scheme function that returns the area of a circle given its radius. Give an example of its use.
Reminder: The answers to unstarred exercises are at the end of the chapter.

16.2 Design Of The Scheme Interpreter

You now have enough information to begin the design of your Scheme interpreter. The main logic can display a frame for the graphical interface with a textfield for the input and a textarea for the output. The accompanying design block contains a reasonable design for the `actionPerformed` method that responds to the ENTER key within the textfield and prints the results.

STRUCTURED NATURAL LANGUAGE DESIGN of actionPerformed
1. Get the string of characters from the textfield.
2. Convert that string to a WordUnit, TextString, real, or list, whichever kind of item it represents.
3. If the item is a list with the `set!` or `define` keyword then...
   Update the set of variable or function definitions.
4. Find the value of the item.
5. Print the string representation of that value in a textarea.
6. BUT, if any exceptional situation arises in Steps 2 through 5 then...
   Print an oh-oh! message explaining the problem.

Object design

What objects are needed to implement the main logic? Three objects needed for the graphical user interface are a JFrame object, a JTextField object, and a JTextArea object. For the Scheme language itself, it appears good to have WordUnit objects, TextString objects, Real objects, and List objects.

Since these four kinds of objects can all appear as elements of a List, you should have a class or interface from which those four classes inherit. Call it Item. Since there will be no actual Item instances, only instances of the four kinds of Items, an Item interface seems best. It gives the public instance method headings the four classes have in common, with a semicolon in place of each method body.

Step 4 of the structured design only requires a single Java statement if each of the four Item subclasses has a `getValue()` method that returns the value of that kind of Item. Step 5 of the design also requires only a single Java statement if you have an appropriate `toString()` method for each kind of Item. Listing 16.1 (see next page) gives the design of the Item interface and its implementations so far. The bodies of the methods in the four implementations are stubbed for now.

Step 2 of the structured design seems to require some kind of parser object with a method that accepts one line of characters as input and produces the corresponding Item. The parser needs to call a constructor for each of the four kinds of Item. The constructor for a Real object is passed a double value, and the constructor for a TextString or WordUnit is passed the String of characters forming it. The tricky part is the constructor for a List object. Figure 16.1 shows the set of all object classes so far.

Listing 16.1 Item and its four implementors, stubbed form

```
public interface Item
{
   /** Return the Scheme value of the Item, which is the current
    *  value if a variable, the evaluation if a function call. */

   public Item getValue();

   /** Return the way the Item is presented to the user
    *  on the screen. */

   public String toString();
}
//#################################################################


public class WordUnit implements Item
{
   public WordUnit (String given)              { }
   public Item getValue()           {return null; }
   public String toString()         {return null; }
}


public class TextString implements Item
{
   public TextString (String given)            { }
   public Item getValue()           {return null; }
   public String toString()         {return null; }
}


public class Real implements Item
{
   public Real (double given)                  { }
   public Item getValue()           {return null; }
   public String toString()         {return null; }
}


public class List implements Item
{
   public Item getValue()           {return null; }
   public String toString()         {return null; }
}
```

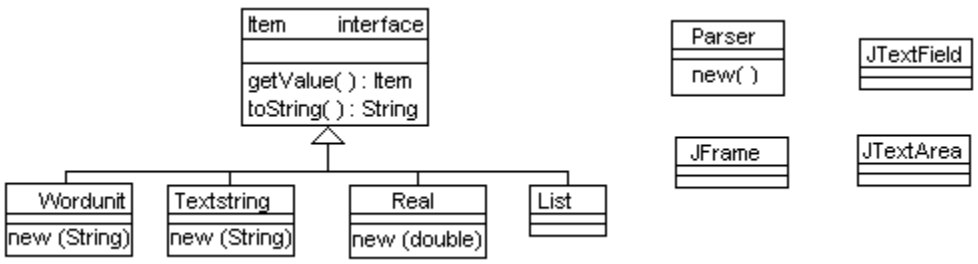

Figure 16.1 UML diagram of object classes needed for the interpreter

The TextString and Real classes

The simplest of the four kinds of Item to implement is TextString. If the constructor simply stores the String parameter away in an instance variable named `itsValue`, the `toString` method can simply return that value when requested. Listing 16.2 shows the complete TextString class.

Listing 16.2 The TextString class

```
public class TextString implements Item
{
   private String itsValue;


   public TextString (String given)
   {  itsValue = given;
   }  //=====================

   public Item getValue()
   {  return this;
   }  //=====================

   public String toString()
   {  return itsValue;
   }  //=====================
}
```

The Real class is not much harder. It should have a double value as its sole instance variable; call it `itsValue`, just as for TextStrings. For function calls such as `+` and `/`, you will need to retrieve the double value from the Real object so you can add or divide the numbers. So the Real class needs a `getNumber` method. These methods are left as exercises.

Top-level implementation

Finding the value of an Item is done by calling its `getValue` method. The interpreter's main logic should have a variable of type Item for which it calls the `getValue` method. This lets the runtime system call the `getValue` method of the appropriate implementation. This is a polymorphic call: The runtime system determines the type of the Item at that point in the execution of the program.

If an Item variable refers to a List object, the call of the `getValue` method in Step 4 of the main logic checks that it is a non-empty list, that the first element on the list is a WordUnit, and that the WordUnit has a meaning. However, all of this also has to be done by Step 3 of the main logic, which determines whether the user's input is a List object beginning with `set!` or `define`. This is duplication of effort. So it seems best to omit Step 3 of the main logic entirely, letting List's `getValue` method do that job.

Throwing an Exception

The parser object could find a syntax error in the input at any of several different stages. And if the expression parses correctly, it may still have a **semantic** error such as the wrong number of parameters for a function, an undefined function name, etc. So errors can arise at many points in the processing of one line of input. Wherever an error arises, you want to terminate all processing back to the main logic and print an oh-oh message. This is a perfect situation for throwing an Exception.

You can create a BadInput subclass of the standard library RuntimeException class. The top part of Listing 16.3 gives the standard way of writing such a subclass of Exception: two constructors and nothing else. Then any time a method finds an error, it need only execute `throw new BadInput (someMessage)` to have all processing terminate back to the `actionPerformed` method. That method contains the try/catch statement that catches all such throws of BadInput Exception objects and displays "oh-oh" followed by that message (Step 6 of the structured design is this try/catch logic).

Listing 16.3 The BadInput class and the main logic

```
public class BadInput extends RuntimeException
{
   public BadInput ()
   {  super();
   }  //====================

   public BadInput (String message)
   {  super (message);
   }  //====================
}
//#############################################################



// the actionPerformed method for the JTextField's ActionListener

   private JTextField field;
   private JTextArea output;
   private Parser parser;

   public void actionPerformed (ActionEvent event)
   {  output.append ("\n" + field.getText() + "    ==> ");
      try
      {  Item userInput = parser.parseInput (field.getText());
         output.append (userInput.getValue().toString());
      }catch (BadInput e)
      {  output.append ("oh-oh: " + e.message());
      }
   }  //====================
```

Converting the string of characters that the user enters into a Scheme expression made up of the various kinds of Items will be fairly complex (though much less so than if it were a Java statement or method definition). So an entirely separate class to convert the string (**parse** it) should be used, with probably only one public method named `parseInput`. This is left as a major programming project.

You now know enough to understand the `actionPerformed` method that reacts to the user's pressing the ENTER key inside the JTextField for the next expression to be processed. The implementation in Listing 16.3 assumes that `field`, `output`, and `parser` are instance variables of a class containing the `actionPerformed` method.

Exercise 16.4 Write the constructor and the `getValue` method for the Real class, assuming a single double instance variable named `itsValue`.
Exercise 16.5 Write the `toString` and `getNumber` method for the Real class under the same assumption.
Exercise 16.6 Write the two statements in the try part of Listing 16.3 as one statement.

16.3 The Map Interface And The Mapping Interface

You need to store the names of Scheme variables that have been defined along with their values. At other times you need to be able to look up the name of a variable to see if it has a value and, if so, what that value is. Also, you need to store the names of functions that have been defined along with their definitions and later look them up. What you need is a Map. The Map interface is part of the Sun standard library, in package `java.util`. It replaces the Dictionary abstract class that was in an earlier version of Java.

A **Map** is a collection of pairs of objects, the first being the ID of the second. An example of one kind of pair is a Social Security Number (SSN) plus the Worker with that SSN. Another example is a dictionary word and its meaning. In general, the ID is called the **key** and the other object is the **value** for that key. Different values must have different keys. Figure 16.2 should give you some idea of what the basic Map methods mean:

- You can add a new Worker to the SSN/Worker Map if you supply its SSN, as in `someMap.put (herSSN, worker)`.
- You can find out whether there is a Worker in the SSN/Worker Map with a given SSN: `someMap.containsKey (herSSN)` returns `true` if `herSSN` is stored as a key, `false` if not.
- You can look up a Worker in a SSN/Worker Map by giving the Social Security Number, as in `worker = someMap.get (herSSN)`; it returns null if there is no Worker with that SSN in the Map.
- You can take a particular SSN/Worker pair out of the Map when you specify the SSN to look for: `someMap.remove (herSSN)` returns the removed value.

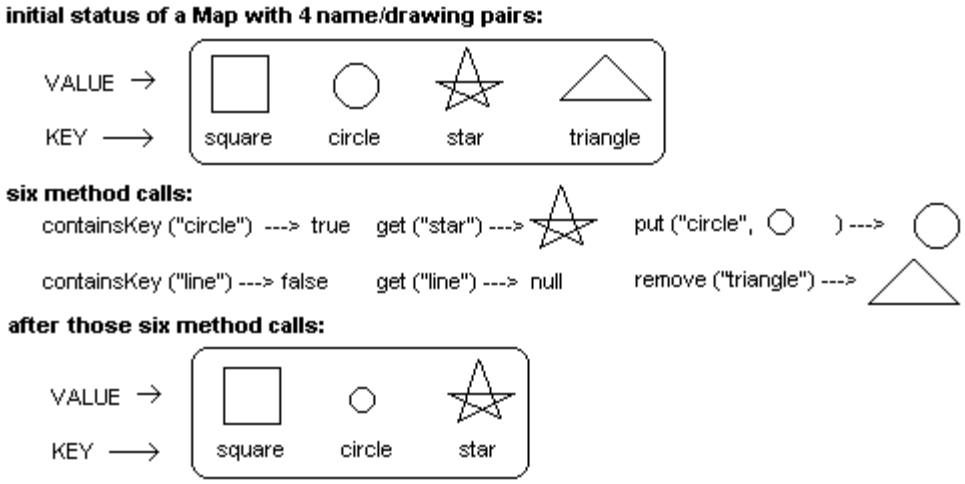

Figure 16.2 Meaning of Map methods; ---> shows what is returned

The Mapping interface

The Sun standard library Map interface prescribes twelve methods that must be implemented. Most implementations do not need all of them. Some of them return Sets or Collections, which for simplicity we do not wish to discuss here. So for most of this chapter, we use a stripped-down form called **Mapping**. This interface is given in the upper part of Listing 16.4 (see next page). Note that it ignores a key/value pair with a null key, although it allows a key/value pair with a null value. By contrast, the standard Map either accepts a null key or throws an Exception when you try to add one.

Listing 16.4 The Mapping and Iterator interfaces

```
public interface Mapping // simplified Map, specific to this book
{
   /** Put this key/value pair in this Mapping if id is not null.
    *  If some existing key/value pair has key.equals(id),
    *  replace the value, otherwise add the key/value pair.
    *  Return the previous value, or null if there was none. */
   public Object put (Object id, Object value);

   /** Tell whether a key/value pair satisfies key.equals(id). */
   public boolean containsKey (Object id);

   /** Return the value in the key/value pair where
    *  key.equals(id).  Return null if no such key/value pair. */
   public Object get (Object id);

   /** Take out the key/value pair where key.equals(id), if any.
    *  Return the existing value, or null if there was none.  */
   public Object remove (Object id);

   /** Tell whether this data structure has no elements. */
   public boolean isEmpty();

   /** Tell how many elements are in this data structure. */
   public int size();

   /** Return an object that can be used to list all the
    *  elements in this data structure one at a time. */
   public java.util.Iterator iterator();
}
//#############################################################


public interface Iterator  // in java.util
{
   /** Tell whether any more elements can be obtained. */
   public boolean hasNext();

   /** Return the next element.
    *  Throw NoSuchElementException if hasNext() is false. */
   public Object next();

   /** Remove the element most recently returned by next.
    *  This operation is optional -- the implementation may
    *  throw UnsupportedOperationException if not available. */
   public void remove();
}
```

The first six methods listed for the Mapping interface are all in the standard library Map interface. Mapping has in addition a method that returns an Iterator object. You use it to progress through the values in the data structure one item at a time in some order. The "official" Map interface provides an Iterator indirectly, in that one Map method returns a Set object which has a method that returns an Iterator. So the Mapping interface is essentially a subset of the "official" Map interface. The full Map interface from the Sun standard library is discussed in the last section of this chapter, along with two useful Sun library implementations of it. Until then, we use this book-specific Mapping only.

The Iterator interface

The **Iterator** interface in the `java.util` package has the three methods described in the lower part of Listing 16.4, but the choice of whether to provide the ability to remove the current element is optional with the implementor. For this chapter, we will simply have the `remove` method throw an **UnsupportedOperationException** (from the `java.lang` package). An iterator is intended to be used in coding such as the following:

```
Iterator it = someMapping.iterator();
while (it.hasNext())
{  Object nextElement = it.next();
   // whatever processing of nextElement is appropriate
}
```

If you construct an Iterator object to use with a collection of objects that has N elements, then you may call `next()` exactly N times; it will give you the N elements one at a time. If you call `next()` N+1 times, it throws a **NoSuchElementException** (from the `java.util` package). Some sample Java logic for an outside class to print all the key/value pairs in a Mapping object is as follows:

```
public static void display (Mapping given)  // independent
{  Iterator it = given.iterator();
   while (it.hasNext())
      System.out.println (it.next());
}  //======================
```

If you wanted to only print the values that are not equal to a particular value `badOne`, you would have to have the body of the method be as follows. You store `it.next()` in a local variable because each call of `it.next()` advances to the next value:

```
Iterator it = given.iterator();
while (it.hasNext())
{  Object nextElement = it.next();
   if ( ! nextElement.equals (badOne))
      System.out.println (nextElement);
}
```

Using Map methods in the WordUnit class

Finding the value of a WordUnit (usually a variable) has to be easier than finding the value of a List (a function call). More important, it should give some insight on the latter. So it is a good idea to start with that and work on the List class later (in Section 16.8).

Clearly you will need to store all the current values of all variables that have been assigned so far. So the WordUnit class should have a Mapping class variable named perhaps `theVariables`. The Mapping methods then allow a simple coding of the `getValue` method of the WordUnit class: If `theVariables` contains the given WordUnit as the key field of a key/value pair, the `getValue` method should return its value, otherwise it should throw a BadInput Exception. Note that you need to cast the Object value returned by `get` to have an Item value that can be returned:

```
public Item getValue()                          // in WordUnit
{  if (theVariables.containsKey (this))
      return (Item) theVariables.get (this);
   else
      throw new BadInput (this.toString() + " has no value");
}  //======================
```

When the List class processes a variable definition, it needs to add a given variable/value pair to `theVariables`. So the WordUnit class needs a public method to provide that service. It only needs to be a simple wrapper around a single statement that tells `theVariables` to include the variable/value pair:

```
public void setVariableValue (Item value)     // in WordUnit
{  theVariables.put (this, value);
}  //======================
```

A Map is not supposed to allow two entries with the same key; if the key is already part of an existing key/value pair in `theVariables`, then the new pair replaces the old pair, which is exactly what you want to have happen when you assign a value to a variable. The rest of the WordUnit methods are much the same as for TextString and Real: The constructor is supplied the String of characters that are the written form of the WordUnit object. That String is stored in an instance variable called `itsValue`, which is the value returned by a call of `toString`.

Services provided by the List class

The `getValue` method in the List class finds the value of a function call or other List. First, though, you have to verify that it is legally-formed. A List is a sequence of a number of Items. For a legally-formed function call, the List object's first element must be a WordUnit that has been defined as a function name. Also, the List object must have the same number of arguments as the function heading has formal parameters.

It is clear that you need to be able to ask a List object for its first element so you can see if it is a WordUnit that has been defined. You also need a List method to retrieve the **sublist** of arguments, i.e., the rest of the List excluding that first element.

If `sam` is a non-empty List object, then `sam.first()` is the first Item in the List. `sam.rest()` is the sublist containing all elements of `sam` except its first element. Figure 16.3 illustrates their meaning for a List of six values. You will also need to be able to tell whether a List is empty; that is the condition `sam.isEmpty()`.

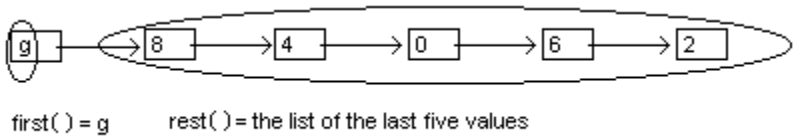

Figure 16.3 Meaning of first() and rest()

Note that, in developing a class such as List, you do not consider the (private) instance variables until after you have most of the public methods. The only reason for having an object know stuff (instance variables) is so it can do stuff (respond to method calls). Once you know what an object has to be able to do, and not before, you will know what the object has to know.

The `set!` and `define` keywords require special handling, as do the predefined functions such as `+` and `sqrt`. You need a different segment of Java coding to process each one. Finding the right code segment can be done easily if you have a different numerical index for each keyword and predefined function name. So the data structure where they are stored should let you look up the numerical index of a WordUnit. Given a WordUnit, first try searching in that data structure to retrieve its numerical index. If you do not find it, treat that WordUnit as a normal function name. Further details on these numerical indexes, and coding for them, is postponed until Section 16.8.

A normal function name should have a definition that has previously been stored, generally using the `define` keyword. The obvious data structure for storing these definitions is a Mapping, perhaps called `theDefs`. And the data structure in which you look up the numerical index of a keyword or predefined function should also be a Mapping, perhaps called `theSpecials`. These will be two class variables in List.

Since a Mapping requires that the value stored with a keyword (such as `set!` or `define`) be an object, not a numerical index, `theSpecials` can use a Real object rather than a numerical index. So the `getValue` method in the List class could include this statement (you need the cast because `get` returns an Object):

```
Real index = (Real) theSpecials.get (this.first());
```

If `this.first()` is found, and thus `index` is not null, you need to call a method to get the value by choosing the right segment of Java logic. Otherwise you need to call a completely different method for a normal List object. So the List class needs two private instance methods `specialValue(Real index)` and `normalValue()`.

For the following exercises, suppose the **Roget** class declares `theDictionary`, a class variable that is an implementation of Mapping. The key is a word and the value is its dictionary definition. Both the word and its definition are of type String. The Roget class also has instance variables `int itsSize` and `String[] itsItem`, with some words stored in array components `0...itsSize-1`. A precondition for all exercises is that no parameter is null unless explicitly stated or its type is Object.

Exercise 16.7 Write a Roget method `public static void inOrOut (String word, String definition)`: Put into `theDictionary` the given word and its definition if the given word is not there, otherwise take the word and its definition out.
Exercise 16.8 Write a Roget method `public boolean hasDefinition()`: The executor tells whether at least one of the words in `itsItem` is in `theDictionary`.
Exercise 16.9 Write a WordUnit method `public boolean equals (Object ob)`: The executor tells whether `ob` is a WordUnit object with the same `itsValue`.
Exercise 16.10 If x is the list of five elements `(a b c d e)`, then what is `x.first()? x.rest()? x.rest().first()?`
Exercise 16.11 Write a List method `public boolean hasThree()`: The executor tells whether it has at least three elements.
Exercise 16.12 (harder) Write a generic Mapping method `public void putAll (Mapping given)`: The executor puts all of the parameter's data in its own structure. Hint: Use an iterator to go through all the values in the `given` Mapping one at a time. For this exercise and the next, you need to know that the iterator produces MapEntry objects that have a `getKey()` method and a `getValue()` method.
Exercise 16.13* Write a Roget method `public static String getDef (String one, String two)`: It returns the definition of the first parameter if it is in `theDictionary`, otherwise it returns the definition of the second (or null if neither is in `theDictionary`).
Exercise 16.14* Write a Roget method `public int getCount()`: The executor tells how many of the words in `itsItem` are keys in `theDictionary`.
Exercise 16.15* Write out the complete WordUnit class, assuming a single String instance variable named `itsValue`.
Exercise 16.16* If x is the list of three elements `((ab) (c d) (e f))`, then what is `x.first()? x.rest()? x.first().first()? x.rest().first().rest()?`
Exercise 16.17* Write a List method `public Item third()`: The executor returns its third element. It returns null if it has less than three elements.
Exercise 16.18* Write the Mapping method `public int size()` as a generic method. Hint: Use the Mapping's iterator to go through all the values one at a time.

16.4 Implementing The Mapping Interface With Arrays

The Scheme interpreter software will use an implementation of the Mapping interface named SchemeMap. We now look at how to implement the SchemeMap class, which requires implementing the seven methods described by the earlier Listing 16.4: `containsKey(id)`, `get(id)`, `remove(id)`, `put(id, value)`, `isEmpty()`, `size()`, and `iterator()`.

One way is with a partially-filled array of objects (i.e., only the first n components have useable values). Each object stored in the array is a key/value pair called a MapEntry. The MapEntry class in Listing 16.5 is a straightforward implementation of the Entry interface used for Maps in `java.util` (the `equals` method is complex only because you have to guard against putting a dot on null). Entry is declared as an interface nested in the standard library Map interface. So using the standard library means you must refer to it as `Map.Entry`; the dot is needed. Note that it is a class of immutable objects -- you cannot change the values of the instance variables once you have constructed the object.

Listing 16.5 The MapEntry class of objects

```
public class MapEntry implements java.util.Map.Entry
{
   private final Object itsKey;
   private final Object itsValue;


   public MapEntry (Object key, Object value)
   {  itsKey = key;
      itsValue = value;
   }  //=====================


   public boolean equals (MapEntry given)
   {  return given != null
               && (this.itsKey == given.itsKey
                  || (this.itsKey != null
                      && this.itsKey.equals (given.itsKey)))
               && (this.itsValue == given.itsValue
                  || (this.itsValue != null
                      && this.itsValue.equals (given.itsValue)));
   }  //=====================


   public Object getKey()
   {  return itsKey;
   }  //=====================


   public Object getValue()
   {  return itsValue;
   }  //=====================


   public Object setValue (Object ob)
   {  throw new UnsupportedOperationException ("can't setValue");
   }  //=====================
}
```

The array for storing MapEntries contains key/value objects stored in components indexed 0 through `itsSize-1` of an array named `itsItem`. The `put` operation adds a key/value object at the end of the array, and the `get` operation searches through the array to find the key and return its corresponding value. Figure 16.4 shows MapEntry objects as musical notes for keys and their musical definitions as values.

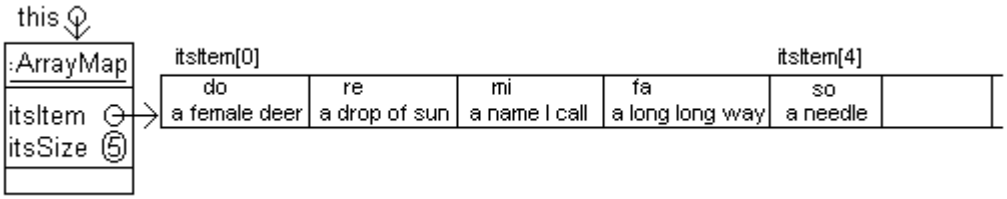

Figure 16.4 Representation of an ArrayMap object

We will be looking at several different ways to implement the SchemeMap class. So we will actually call this first one the ArrayMap class. You can use it for the Scheme interpreter with the following simple name-changing class definition (so `new SchemeMap()` actually makes a new ArrayMap object):

```
public class SchemeMap extends ArrayMap
{  // nothing is needed inside this name-changing class
}
```

The internal invariant for the ArrayMap class, which describes how the abstraction (a data base of MapEntry pairs) corresponds to the concrete (the instance variables) is therefore as follows:

- `itsItem` is an array of MapEntry values.
- `itsSize` is a non-negative int value which is the number of MapEntry pairs in the data structure. The data structure is empty when `itsSize` is zero.
- Each such MapEntry pair is in one of the components `itsItem[0]` through `itsItem[itsSize-1]`, in no specific order.

Implementing the ArrayMap class

Listing 16.6 (see next page) contains several methods of the ArrayMap class; the rest are left as exercises. The `put` method should refuse to add a MapEntry with a null key; the `containsKey` method relies on this refusal. The array length of 100 is arbitrary; an exercise provides for increasing the capacity of the array by 50% when needed. The values are not in any particular order; a major programming project keeps them in order.

The logic for `containsKey` should search through the array from the top down (i.e., higher indexes first), on the assumption that more-recently-defined key/value pairs are used more often. Several other Map methods have to search through the list of values for a particular key, and some of them have to know the exact position where it is found. So this coding uses a private `lookUp` method to find that position if it exists, and `containsKey` simply tells whether the position was found.

The `get` operation also uses the private `lookUp` method. It then returns the value at the position that the `lookUp` method finds (or returns null if it was not found).

Listing 16.6 The ArrayMap class of objects, some methods postponed

```
public class ArrayMap implements Mapping
{
   private MapEntry[] itsItem = new MapEntry [100];
   private int itsSize = 0;  // only has the default constructor


   /** Tell whether a key/value pair satisfies key.equals(id). */

   public boolean containsKey (Object id)
   {  return lookUp (id) >= 0;
   }  //=====================

   /** Return the value in the key/value pair for which
    *  key.equals(id).  Return null if no such key/value pair. */

   public Object get (Object id)
   {  int loc = lookUp (id);
      return (loc < 0)  ?  null  :  itsItem[loc].getValue();
   }  //=====================

   /** Return the index of the key/value pair for which
    *  key.equals(id).  Return -1 if no such key/value pair. */

   private int lookUp (Object id)
   {  int k = itsSize - 1;
      while (k >= 0 && ! itsItem[k].getKey().equals (id))
         k--;
      return k;  // -1 if the object is not in the list
   }  //=====================
}
```

A driver program

When you do the exercises, you will probably want to test out your logic. The program in Listing 16.7 (see next page) is a **driver program**, i.e., it "exercises" several ArrayMap methods in a way that should uncover most bugs in your logic. It will also exercise the methods of the NodeMap class that is discussed in the next section. Figure 16.5 is a UML diagram for it.

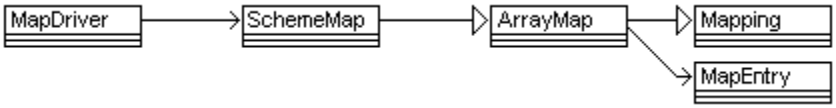

Figure 16.5 Partial UML class diagram for MapDriver

This driver program creates a SchemeMap object. The user inputs several word/definition pairs. Each is added to the SchemeMap, unless the word is already defined, in which case the definition is removed. Consequently, the Map object will contain all the word/definition pairs that have been entered an odd number of times. Exceptions: If the definition is the empty string or null, that is a signal to retrieve the current definition of the word and display it. And if the word is the empty string or null, the program terminates.

Exercise 16.19 Write the ArrayMap method `public int size()`.
Exercise 16.20 Write an ArrayMap method `public int countLess (Comparable id)`: The executor tells how many of its entries have a key that is less than a given key. Throw an Exception if the keys are not from a class that implements Comparable (and so has the usual `compareTo` method).

Listing 16.7 Driver program for an implementation of SchemeMap

```
import javax.swing.JOptionPane;

class MapDriver
{
   public static void main (String [ ] args)
   {  JOptionPane.showMessageDialog (null,
                "Repeatedly enter a word, then its definition.");
      WordProcessor.processManyWords (new SchemeMap());
      System.exit (0);
   }  //====================
}
//#########################################################


public class WordProcessor
{
   public static void processManyWords (Mapping odds)
   {  String word = JOptionPane.showInputDialog ("word?");
      while (word != null && word.length() > 0)
      {  String definition = JOptionPane.showInputDialog
               ("definition in/out?  ENTER key to see it: ");
         if (definition == null || definition.length() == 0)
            System.out.println ("its definition is currently "
                                + (String) odds.get (word));
         else if (odds.containsKey (word))
            System.out.println (word + " had the definition "
                                + (String) odds.remove (word));
         else
         {  odds.put (word, definition);
            System.out.println ("added " + word + " to the map; "
                                + odds.size() + " entries.");
         }
         word = JOptionPane.showInputDialog ("word?");
      }
   }  //====================
}
```

Exercise 16.21 Write the ArrayMap method `public Object remove (Object id)`: The executor first sees if the key is in the data structure. If not, nothing happens except that `remove` returns null. But if the key is there, the executor replaces that key/value pair by the one at the end of the array, makes the array one MapEntry smaller, and returns the value that corresponded to the key. Note that `id` could be null.

Exercise 16.22 (harder) Write an ArrayMap method `public ArrayMap reverse()`: The executor creates and returns a new ArrayMap object with a different array that has the same entries in it in the opposite order.

Exercise 16.23 (harder) Write the ArrayMap method `public Object put (Object id, Object value)`: The executor does nothing but return null if the given key is null. Otherwise it adds the key/value pair unless the key was already in the Mapping, in which case it replaces the pair. It returns null if no pair with that key was initially in the Mapping, otherwise it returns the value that corresponded to the key. It throws an IndexOutOfBoundsException when the array is full (improved in the next exercise).

Exercise 16.24* Modify the `put` method in the preceding exercise to verify that the array is not full; when it is, create a new array 50% larger and transfer the entries to it.

Exercise 16.25* Write an ArrayMap method `public boolean equals (Object ob)`: The executor tests whether the parameter is an ArrayMap object with the same key/value pairs in the same order as the executor.

Exercise 16.26* Write the ArrayMap method `public boolean isEmpty()`.
Exercise 16.27* Write an ArrayMap constructor with an ArrayMap parameter: Construct the new ArrayMap to have the same key/value pairs as the parameter, in the same order.
Exercise 16.28* Write an ArrayMap method `public void display()`: The executor prints all of its keys to `System.out`. This is useful when you are debugging a program.
Exercise 16.29* Omit the private `lookUp` method in ArrayMap. Instead, have a private instance variable `thePosition` that `containsKey` sets to the correct location when it finds the value it is looking for. Code `containsKey`. Then rewrite `get`, `put`, and `remove` (the latter two are in the preceding exercises) to call on `containsKey` and, when the `id` is found, use the value of `thePosition`.
Exercise 16.30** Write a `put` method that keeps the values of the ArrayMap in ascending order. Assume keys are Comparable. Use binary search.
Reminder: Double-starred exercises are harder; the answers are not in the book.

16.5 Implementing The Mapping Interface With Linked Lists

A more flexible way to implement the SchemeMap is as a linked list, similar to the linked lists in Scheme itself. As was done for the ArrayMap, the actual SchemeMap class will indirectly use a class named NodeMap (in place of the earlier definition of SchemeMap):

```
public class SchemeMap extends NodeMap
{  // nothing is needed inside this name-changing class
}
```

A NodeMap object will have two instance variables, `itsData` and `itsNext`. A NodeMap is conceptually a large number of **nodes** on the linked list where each node contains a single MapEntry (`itsData`) plus a reference to the next node on the linked list (`itsNext`). But in the last node (which is the only node if the map is empty), `itsData` is null and so is `itsNext`. This last node is called a **trailer node**.

The advantage of using a linked list is that you do not have to worry about (a) crashing the program because you made the array too small, or (b) wasting space because you made the array too big. Figure 16.6 shows an example of a linked list with trailer.

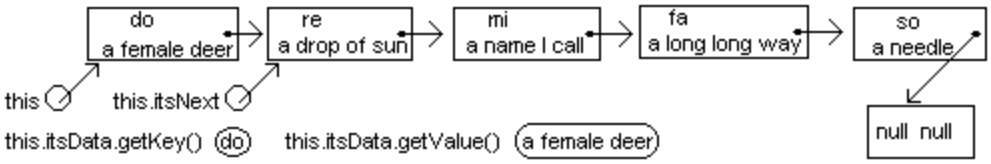

Figure 16.6 Representation of a NodeMap object

The containsKey method

A recursive logic for `containsKey` is now quite straightforward: There are two kinds of lists, an empty list and a nonempty list (this mantra reminds you that you always begin by considering these two alternatives). An empty list does not contain the key. A nonempty list `this` contains the key when and only when `this.itsData` is a MapEntry with that key or `this.itsNext` refers to a sublist that contains the key.

```
public boolean containsKey (Object id)   // in NodeMap
{  return ! isEmpty()
          && (this.itsData.getKey().equals (id)
                      || this.itsNext.containsKey (id));
}  //=====================
```

However, a private method to find the position where a given key appears would be more useful, since it could do most of the work for several different methods:

```
private NodeMap lookUp (Object id)       // in NodeMap
{  return this.isEmpty() || this.itsData.getKey().equals (id)
            ?  this  :  this.itsNext.lookUp (id);
}  //=====================
```

Then `containsKey` only needs one simple return statement, telling whether `lookUp(id)` is empty. You will also find it profitable to study a non-recursive version of this `lookUp` logic, as follows:

```
private NodeMap lookUp (Object id)       // in NodeMap
{  NodeMap p = this;
   while ( ! p.isEmpty() && ! p.itsData.getKey().equals (id))
      p = p.itsNext;
   return p;  // the trailer node if it gets this far
}  //=====================
```

You should compare that logic with the following from Listing 16.6 step by step:

```
private int lookUp (Object id)           // in ArrayMap
{  int k = itsSize - 1;
   while (k >= 0 && ! itsItem[k].getKey().equals (id))
      k--;
   return k; // -1 if the object is not in the list
}  //=====================
```

You can see that `p` corresponds to `k` exactly:

- `p` or `k` indicates your current position in the sequence of values.
- `p` starts at the first position in the list (`this`) and `k` starts at the first position from the end (at `itsSize - 1`).
- The loop continuation condition requires that `p` or `k` not be beyond the last place where a MapEntry occurs (signaled by `p.isEmpty()` and `k < 0`, respectively).
- You move on by changing `p` or `k` to the following position.
- `p.itsData` corresponds to `itsItem[k]` exactly; either way, it is the MapEntry value at the current position. The rest of the logic is identical.

A NodeMap object represents a linked list, that is, a list that contains many entries. But a NodeMap object has only two instance variables `itsData` and `itsNext`, and only one of them is an entry. If X is the executor of a NodeMap method, then X's node and X's linked list are two different mental concepts. In the preceding `lookUp` method, `p` moves through the linked list by being initialized to refer to the first node of the linked list, then to the second node of the linked list, then to the third one, etc.

The get method, the size method, and the constructors

The natural constructor for the NodeMap class fills in its two instance variables with the corresponding parameter values. You also need a public constructor with no parameters; outside classes can use it to make an empty NodeMap object. That would of course be the empty list of values, which can be recognized by the fact that `itsNext` is null. These two constructors are in the upper part of Listing 16.8 (see next page). The natural constructor is private because it is only used internally to modify the data structure. Both are required, even though the public one looks like the default constructor.

Listing 16.8 The NodeMap class, some methods postponed

```
public class NodeMap implements Mapping
{
   private MapEntry itsData;          // trailer node logic
   private NodeMap  itsNext;


   private NodeMap (MapEntry data, NodeMap next)
   {  itsData = data;
      itsNext = next;
   }  //=====================

   public NodeMap()
   {  itsData = null;
      itsNext = null;
   }  //=====================


   public boolean isEmpty()
   {  return itsNext == null;
   }  //=====================

   public boolean containsKey (Object id)
   {  return ! lookUp (id).isEmpty();
   }  //=====================

   public Object get (Object id)
   {  NodeMap loc = lookUp (id);
      return loc.isEmpty() ?  null  :  loc.itsData.getValue();
   }  //=====================

   public int size()
   {  return this.isEmpty() ?  0  :  1 + this.itsNext.size();
   }  //=====================


   /** Return the node containing the key/value pair for which
    *  key.equals(id). Return the trailer node if no such
    *  key/value pair exists.  */

   private NodeMap lookUp (Object id)
   {  return this.isEmpty() || this.itsData.getKey().equals (id)
                ?  this  :  this.itsNext.lookUp (id);
   }  //=====================
}
```

The executor of the `get` method should call `lookUp` to see if the `id` is already in the data structure. If so, it returns the corresponding value, otherwise it returns null. This can be written in Java using the same logic as in the earlier Listing 16.6 except that `itsItem[loc].getValue()` is replaced by `loc.itsData.getValue()`. This `get` method is in the middle part of Listing 16.8.

The `size` method is the simplest kind of recursion: An empty list has no data; a non-empty list has one piece of data (in `itsData`) plus whatever data is in the rest of the list. Listing 16.8 collects together the progress to date on this linked list implementation.

In the Scheme programming language, once you make a linked list, you cannot change it in any way. The List class is immutable the way the String class is. By contrast, you need to be able to modify the state of a NodeMap object.

The put method

The `put` method requires that you change the number of values in the linked list represented by a NodeMap object. But you can never change the value of the executor node by a method call (which is why a statement beginning `this=` is illegal). So you have to change the values of the instance variables of the nodes. The accompanying Structured English design block gives a workable solution. The coding is in Listing 16.8. There are other logics that execute perhaps slightly faster, but they are moderately more complex.

STRUCTURED NATURAL LANGUAGE DESIGN of put for NodeMap

1. Ignore the `put` request if the `id` is null, since Mappings do not allow null ids.
2. If the given `id` does not occur in any MapEntry in the data structure, then...
   - 2a. Create a new node after the first node, containing the first node's data.
   - 2b. Put the given `id` and its value into the first node.
3. Otherwise...
   - 3a. Put the given `id` and its new value into the node containing the `id`.
   - 3b. Return the value that previous was stored with that `id`.

Listing 16.9 The put and remove methods for the NodeMap class

```
// public class NodeMap, continued

   public Object put (Object id, Object value)
   {  if (id == null)   // Mappings ignore a null key        //1
         return null;                                        //2
      NodeMap loc = lookUp (id);                             //3
      if (loc.isEmpty())                                     //4
      {  this.itsNext = new NodeMap (this.itsData, this.itsNext);
         this.itsData = new MapEntry (id, value);            //6
         return null;                                        //7
      }                                                      //8
      Object valueToReturn = loc.itsData.getValue();         //9
      loc.itsData = new MapEntry (id, value);                //10
      return valueToReturn;                                  //11
   }  //====================


   public Object remove (Object id)
   {  NodeMap loc = lookUp (id);                             //12
      if (loc.isEmpty())                                     //13
         return null;                                        //14
      Object valueToReturn = loc.itsData.getValue();         //15
      loc.itsData = loc.itsNext.itsData;                     //16
      loc.itsNext = loc.itsNext.itsNext;                     //17
      return valueToReturn;                                  //18
   }  //====================
```

This logic works because a NodeMap is a linked list with a **trailer node**: When the linked list contains `N` values, it contains `N+1` nodes, the last one having null for both of its instance variables. Figure 16.7 shows the result of calling this `put` method twice.

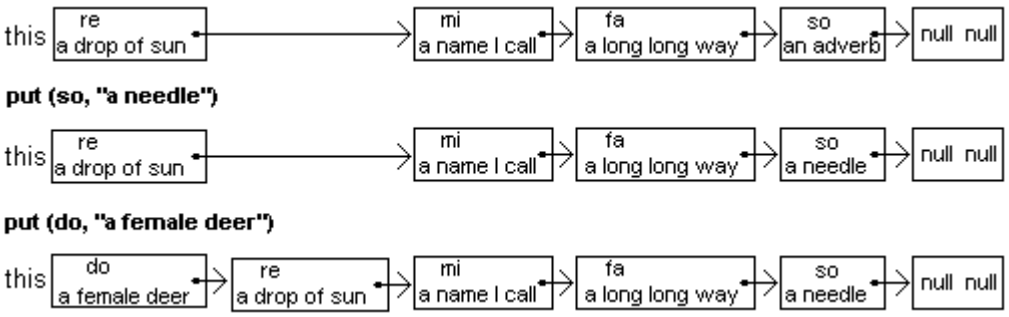

Figure 16.7 Result of two consecutive calls of put(Object, Object)

The remove method

For the `remove` method, if the linked list does not contain the key, then of course you simply return null. Otherwise, `lookUp` gives you a NodeMap value `loc` such that `loc's` node contains the MapEntry value that you want to remove. If it happens to be the first node in the linked list, you cannot delete the node because it is the executor itself (i.e., it is `this`). What you can do is delete the node after the one `loc` refers to. If you first move the MapEntry value from that deleted node up into `loc's` node, you have removed `loc`'s MapEntry value from the list.

Even if `loc` refers to some node other than the first one, you cannot easily delete that node itself from the linked list, because you have no convenient way to hook up the node before it to the node after it. So just do the same as at the first node: Move the MapEntry value in the following node up into `loc`'s node (which removes the entry you wanted to remove) and then remove the node after `loc`'s node from the linked list. This logic is in the lower part of Listing 16.9. Figure 16.8 shows the result of applying this `remove` logic to a linked list.

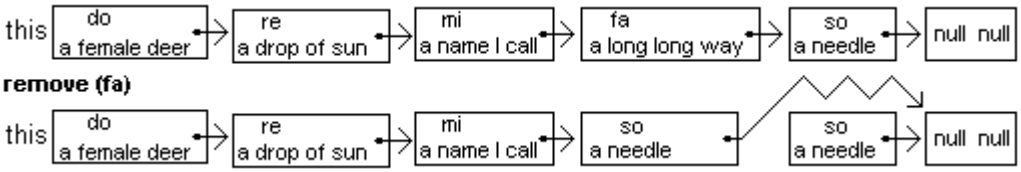

Figure 16.8 Result of a call of remove(Object)

You might think that this logic gives the wrong result when you are removing the last entry in the list, because there is no entry that comes after it. However, the empty list comes after it, consisting of a single node with both of its instance variables being null. So the `remove` logic simply copies both null values up to the current node. That makes the current node be the empty node at the end. The node that was last is no longer referred to by any other node, so garbage collection will recycle it eventually.

An alternative implementation using simple linked lists

Another way to implement a Mapping is to use a **simple linked list**, which is a linked list in which every node contains a data value (thus there is no trailer node). This saves a little space in RAM, but it makes the logic more complex. A Mapping object implemented this way contains just one instance variable `itsFirst` that is the first Node on a list, but is null when the list has no data. Node is a `private static class` **nested** within the Mapping class. That means that its methods can access the instance variables of a Node object but outside classes cannot (and thus encapsulation is maintained).

The `get` method has exactly the same coding as in Listing 16.8 except that the condition `loc.isEmpty()` is replaced by the condition `loc == null`. The `lookUp` method for a simple linked list could have the same logic as the non-recursive version in the previous section, except declare `Node p = this.itsFirst` and have the heading of the while-loop be the following:

```
while (p != null && ! p.itsData.getKey().equals (id))
```

Another alternative is to keep the data in ascending order of IDs, if the ID values are Comparable. This alternative is explored in the exercises. In any case, most methods in both ArrayMap and NodeMap execute in big-oh of N time since `lookUp` does.

Exercise 16.31 Write a NodeMap method `public Object secondOne()`: The executor returns the value part of its second entry, except it returns null if its map has less than two values.
Exercise 16.32 Write a NodeMap method `public void clear()`: The executor makes the linked list empty.
Exercise 16.33 Write a NodeMap method `public void swapTwo()`: The executor swaps the first and second entries in its map, except it has no effect if its map has less than two values.
Exercise 16.34 What changes would you make in the `put` method (Listing 16.9) to add the new MapEntry object directly after the first NodeMap object using `itsNext = new NodeMap (new MapEntry (id, value), itsNext)` whenever appropriate?
Exercise 16.35 (harder) Write a NodeMap method `public boolean equals (NodeMap given)`: The executor tells whether it has the same data values as the parameter in the same order. Use recursion. Precondition: `given` is not null.
Exercise 16.36 (harder) Suppose the keys in a NodeMap are in ascending order and are Comparable. Write a recursive `putRecursive` method that inserts the new MapEntry in a way that maintains order, called from `put` with the following statement: `return id == null ? null : putRecursive ((Comparable) id, value);`.
Exercise 16.37 (harder) Revise all NodeMap coding to keep the keys in ascending order. Assume that all keys are mutually Comparable. Hint: Have `lookUp` stop earlier.
Exercise 16.38 (harder) Suppose the keys in every NodeMap are in ascending order and are Comparable. Write a NodeMap method `public NodeMap intersection (NodeMap given)` that returns a new NodeMap that contains all MapEntries that are in both `given` and the executor. Use recursion. When two MapEntries have the same key, keep only the value that goes with the executor's key. Precondition: `given` is not null.
Exercise 16.39* Write a NodeMap method `public boolean isAscending()`: The executor tells whether or not the keys in its map are in ascending order (the comparison of any key to the next does not give a positive value). Precondition: All keys can be cast to Comparable (i.e., have the usual `compareTo` method).
Exercise 16.40* Write a NodeMap method `public int countLess (Comparable id)`: The executor tells how many of its entries have a key that is less than the given key. Precondition: All keys can be cast to Comparable.
Exercise 16.41* Write a NodeMap method `public Comparable firstKey()`: The executor returns the smallest key in its map. Precondition: All keys can be cast to Comparable. Return null if the map is empty.
Exercise 16.42* Write a NodeMap constructor with a NodeMap parameter: Construct the new NodeMap to have the same key/value pairs as the parameter.
Exercise 16.43* Write a NodeMap method `public void display()`: The executor prints all of its keys to `System.out`. This is a useful method when you are debugging a program using NodeMaps. Use recursion.
Exercise 16.44** Write a NodeMap `public void removeValue (Object ob)`: the executor removes every MapEntry object that has `ob` as its value.
Exercise 16.45** Write a NodeMap method `public NodeMap reverse()`: the executor returns a new NodeMap with the same entries in it but in the opposite order.

Part B Enrichment And Reinforcement

16.6 Implementing Mapping Iterators; Ordered Tables

In many situations, you need to go through the entries stored in a Mapping object one at a time. An Iterator object is the appropriate way to do this. The Iterator interface is part of the standard library in the `java.util` package. As Listing 16.4 showed, it requires three methods to be in any class that would implement it, though `remove` can be an unsupported operation (i.e., can always throw an Exception):

```
public interface Iterator
{  public Object next();     // return next available element
   public boolean hasNext(); // tell whether next() is allowed
   public void remove();   // remove what next() last returned
}
```

Listing 16.10 contains an implementation of the Iterator interface that can be placed inside the ArrayMap class (`remove` is deactivated). That lets it access the instance variables of an ArrayMap object. The `itsPos` instance variable gives the position of the next value that the Iterator returns. The `next` method returns the value stored at that position after advancing `itsPos` by 1. The `java.lang` package contains the UnsupportedOperationException class.

Listing 16.10 The MapIt class nested in the ArrayMap class

```
private static class MapIt implements java.util.Iterator
{
   private int itsPos = 0;
   private ArrayMap itsMap;


   public MapIt (ArrayMap given)  // given will not be null
   {  itsMap = given;
   }  //=====================

   /** Return true if the Iterator can produce another one.*/

   public boolean hasNext()
   {  return itsPos < itsMap.itsSize;
   }  //=====================

   /** Return the next element in the sequence, if any. */

   public Object next()
   {  if ( ! hasNext())
         throw new java.util.NoSuchElementException
                           ("iterator has no next element!");
      itsPos++;
      return itsMap.itsItem [itsPos - 1];
   }  //=====================

   /** Remove the element last returned by a call of next().*/

   public void remove()
   {  throw new UnsupportedOperationException ("no remove!");
   }  //=====================
}
```

The MapIt classes

The **MapIt** class is declared with the heading `private static class MapIt` inside the ArrayMap class. This kind of declaration is called a **nested class**. Now the iterator method inside the ArrayMap class is easy to code:

```
public java.util.Iterator iterator()
{  return new MapIt (this);
}  //======================
```

The **MapIt** class nested in the NodeMap class has the same methods as shown in Listing 16.10; only the coding is different. The implementation in Listing 16.11 has the instance variable `itsPos` be the NodeMap object containing the next value that the Iterator will return. So `itsPos` is initially the first node on the linked list in the NodeMap. Then the `hasNext` method simply tests whether `itsPos` contains data, and the `next` method returns the data in that node after moving forward to the next node.

Listing 16.11 The MapIt class nested in the NodeMap class

```
public java.util.Iterator iterator()  // member of NodeMap
{  return new MapIt (this);
}  //======================


private static class MapIt implements java.util.Iterator
{
   private NodeMap itsPos;


   public MapIt (NodeMap given)  // given will not be null
   {  itsPos = given;
   }  //======================


   public boolean hasNext()
   {  return itsPos.itsNext != null;
   }  //======================


   public Object next()
   {  if ( ! hasNext())
         throw new java.util.NoSuchElementException
                              ("iterator has no next element!");
      Object valueToReturn = itsPos.itsData;
      itsPos = itsPos.itsNext;
      return valueToReturn;
   }  //======================


   public void remove()
   {  throw new UnsupportedOperationException ("no remove!");
   }  //======================
}
```

The following could be a constructor in the NodeMap class that creates a new NodeMap object containing all the values in a given Mapping. It goes through the data values in the given Mapping one at a time, inserting them in newly-created NodeMap without regard to maintaining a specific order.

```
public NodeMap (Mapping given)
{  java.util.Iterator it = given.iterator();
   if ( ! it.hasNext())
      return;
   itsData = (MapEntry) it.next();  // the first data value
   itsNext = new NodeMap();         // the new trailer node
   while (it.hasNext())
      itsNext = new NodeMap ((MapEntry) it.next(), itsNext);
}  //======================
```

Ordered tables

Sometimes it is essential to be able to iterate through the data values in ascending order of IDs, where the IDs are Comparable values. Frequent iteration implies you should store them in ascending order, for efficiency. Doing so for the NodeMap class would improve execution of the search methods (`get` and `containsKey`) when searching for an ID that is in the **table** (another name for a map), as well as `put` and `remove`.

One of the fastest-executing ways of maintaining an ordered table is the following, which we will implement as the **BinarySearchListMap** class. The following internal invariant for this class describes the relation between the abstraction (the data base of MapEntry pairs) and the concrete (the instance variables for the implementation):

1. All of the MapEntry values are kept in ascending order of ids in a linked list of nodes that has a trailer node (the trailer node makes insertions and deletions easier). That is, for each node `p`, `((Comparable) p.itsData.getKey()).compareTo (p.itsNext.getKey()) < 0` as long as neither node is the trailer node.
2. An array named `itsItem` contains references to approximately every fourth node in the linked list (although any number from 3 to 6 would work about as well) starting with the first node. As data values are inserted and deleted, the portions of the linked list between two consecutive array positions will vary from four. But at any given time, `itsItem[k+1]` never refers to a node that comes before `itsItem[k]`.
3. A non-negative int `itsSize` is the number of MapEntry values in the data structure.

Therefore, to find an ID, you can perform binary search on the array until you find the node p such that the ID you want is or should be in the four or so nodes beginning at p. Then perform a sequential search on the linked list starting at p.

Execution time to find or insert or remove a MapEntry is big-oh of log(N) time where N is the number of data values, as long as hardly any of those portions are more than seven or eight nodes in length. The reason is that binary search of up to eight values takes about as many comparisons of IDs as the average number of comparisons required for sequential search. This is far better than the big-oh of N time for NodeMap. Figure 16.9 shows how this might look for 19 data values, using integers to represent the keys. Listing 16.12 (see next page) contains part of the coding for this class.

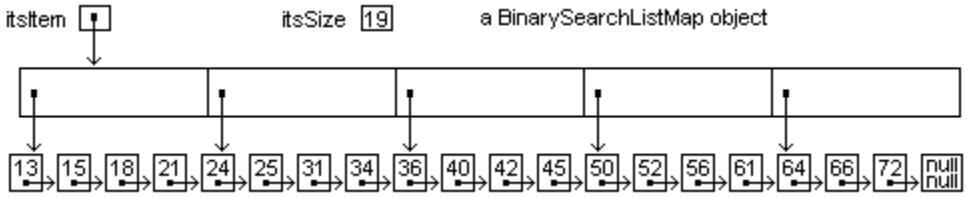

Figure 16.9 a BinarySearchList object with 19 data values

Listing 16.12 The BinarySearchList class

```
public class BinarySearchListMap implements Mapping
{
   private Node[] itsItem;
   private int itsSize = 0;

   public Object put (Object id, Object value)
   {  if (id == null)                                                  //1
         return null;  // Mappings ignore a null key                  //2
      Node start = itsItem [lastLessEqual ((Comparable) id)];//3
      Node loc = firstGreaterEqual ((Comparable) id, start); //4
      if (loc.itsNext == null                                          //5
                     || ! loc.itsData.getKey().equals (id))  //6
      {  itsSize++;                                                    //7
         loc.itsNext = new Node (loc.itsData, loc.itsNext);  //8
         loc.itsData = new MapEntry (id, value);             //9
         return null;                                                  //10
      }                                                                //11
      Object valueToReturn = loc.itsData.getValue();          //12
      loc.itsData = new MapEntry (id, value);                 //13
      return valueToReturn;                                           //14
   }  //=====================


   /** Create a Mapping that contains the data values produced
    *  by the iterator.  Precondition:  The iterator produces
    *  MapEntry objects in ascending order of keys. */

   public BinarySearchListMap (java.util.Iterator it)
   {  Node header = new Node (null, null);                            //15
      Node p = header;                                                //16
      while (it.hasNext())                                            //17
      {  p.itsNext = new Node ((MapEntry) it.next(), null);  //18
         itsSize++;                                                   //19
         p = p.itsNext;                                               //20
      }                                                               //21
      p.itsNext = new Node (null, null);  // trailer node    //22
      createArrayContainingEveryFourthOne (header.itsNext);  //23
   }  //=====================


   private static class Node
   {  public MapEntry itsData;
      public Node itsNext;

      public Node (MapEntry data, Node next)
      {  itsData = data;                                              //24
         itsNext = next;                                              //25
      }  //=====================
   }
}
```

The put method for BinarySearchListMap

The `put` method in Listing 16.12 is quite similar to the NodeMap `put` method in the earlier Listing 16.9. In fact, lines 8-14 are exactly the same as lines 5-11 in Listing 16.9 except that here we insert at position `loc` and there we insert at position `this` (the beginning of the list).

The `put` method calls on two private methods: The `lastLessEqual` method performs a binary search through the array to find the node of highest index containing an ID less than or equal to the given ID. If all IDs whose nodes are in the array are larger than the given ID, `lastLessEqual` returns index zero. Then `firstGreaterEqual` does a sequential search starting from the node that was returned, until it finds a node `loc` containing an ID equal to or greater than the given ID (but stopping at the trailer node). The new MapEntry goes right before the MapEntry in `loc` (lines 8-9), except it may have the same ID, in which case it replaces the MapEntry in `loc` (line 13).

The constructor given an iterator

The constructor's parameter is an object that iterates through some Mapping or Collection of MapEntries in ascending order. In practice, you might store the information from an ordered Mapping in a file at the end of a business week, then read the file in again the next day to build a new ordered table as a BinarySearchListMap object. This class is best used for situations where the data structure does not grow or shrink drastically during use; more than a fifty percent change in one business week can reduce the efficiency of the data structure. But an exercise suggests how to rewrite `put` to allow for a great increase in size.

This constructor simply creates the linked list with trailer node from the values the iterator produces, meanwhile counting them. This linked list's first node that contains data is in `header.itsNext`. Then it calls a private method to store references to the first, fifth, ninth, thirteenth, etc. nodes in the array, starting at `itsItem[0]`. The rest of this BinarySearchListMap class is left as a major programming project. A related project uses the same idea for a very fast implementation of set operations.

The basic operations in BinarySearchListMap execute in big-oh of log(N) time. You should research the concept of "skip lists" if you want to see another popular way of maintaining an ordered table with expected big-oh of log(N) time for all operations.

Exercise 16.46 (harder) Revise the MapIt class for NodeMap to have the `remove` method work right. It should throw an Exception if it cannot remove the MapEntry most recently returned by a call of `next()`. Add another instance variable `itsPrevious` which is the node containing the MapEntry that can be removed, when one exists.
Exercise 16.47 (harder) Write a BinarySearchListMap method `public void putAll (Mapping given)`: The executor adds to itself all the MapEntries in `given`, replacing the value where duplicate keys occur. Precondition: All keys are mutually Comparable.
Exercise 16.48* Revise the `public NodeMap (Mapping given)` constructor in the text to keep the same order of data values as are in the given Mapping.
Exercise 16.49* Write another constructor for Listing 16.12 to make an empty Mapping.
Exercise 16.50* Rewrite the constructor in Listing 16.12 to not create the extra Node.
Exercise 16.51* Write a BinarySearchListMap method `public void clear()`: The executor removes all of its MapEntries, leaving itself empty.
Exercise 16.52* Rewrite the `put` method in Listing 16.12 to discard the current array and create a new one when you reach seven times as many MapEntries as components (this requires roughly 1 second per 10 million data values with a 1400MHz PC chip). Include the coding of the `createArrayContainingEveryFourthOne` method.

16.7 Hash Tables

Say you need to store around 100,000 Info values in a Mapping. Then `lookUp()` needs an average of up to 50,000 comparisons to search for a value in the simplest implementations, and even binary search through the data will require at least 17 comparisons for the average look-up (since 50,000 is nearly 2^{16}). Wouldn't it be great if you had an implementation of the Mapping interface that could `get` or `put` or `remove` a given data value with only one comparison?

One way to do this is to use the ID of each data value as the index of an array where data values are stored. For instance, if a company has employee IDs that range in value from 1 to 99,999, you could have an array of size 100,000 and store e.g. the data value with ID 7413 in `itsItem[7413]` (you store null there if that ID is not in use). Or if each data value corresponds to a particular day in the year, you could have an array of size 366. Listing 16.13 shows a start on this kind of data structure.

Listing 16.13 Two methods in the IndexedMap class of objects

```
public class IndexedMap implements Mapping
{
   private MapEntry[] itsItem = new MapEntry [100000]; //all null


   /** All methods throw a RuntimeException if the given id is
    *  not an Integer object with intValue() in 0...99999. */

   public boolean containsKey (Object id)
   {  int k = ((Integer) id).intValue();
      return itsItem[k] != null;
   }  //=====================


   public Object get (Object id)
   {  int k = ((Integer) id).intValue();
      return itsItem[k] != null ? itsItem[k].getValue() : null;
   }  //=====================
}
```

The collision problem

The kind of situation just described is quite rare, so we will not carry it further in this book. You would have to know that no ID is larger than `itsItem.length-1`. Now you might be thinking that Social Security Numbers (abbreviation SSN) would usually do, since they never exceed 1 billion (SSNs have nine digits). But they cause a different problem: You would need an array of size 1 billion. If you are using four bytes for each reference to a MapEntry, that is a total of four gigabytes of RAM, which may not be available. And even if it were available, it would be a horrendous waste to use that size of array to store just a few hundreds of thousands of values.

One approach is to use the last 5 digits of the SSN for the index into an array with 100,000 components. This is called **open addressing**. But some of the tens of thousands of people will surely have the same last 5 digits. Even if we use the last 6 digits (thus requiring an array of size 1 million), probability theory tells us we are virtually guaranteed to have quite a few cases of people with the same last 6 digits.

This is the **collision problem**: Several data values "collide" when they all try to get into the same storage space. One collision-resolution solution is **linear probing**: When you have a data value X that goes in a position at index K, and some data value is already there, put it at K+1; but if there is something there, put it at K+2 instead, or at K+3 if necessary, etc. If you go beyond the end of the array, start searching for a space at index 0, then 1, etc. Of course, this causes all sorts of problems when you try to find the data value later, since it is not in the right spot. And deletions cause even more problems.

The chained hash table

The **chained hash table** solution in Listing 16.14 (see next page) is a **ChainedHashMap** with an array of 100,000 linked lists. Each list contains all the data values whose SSN has the last 5 digits equal to the index (e.g., 271-82-8182 will be stored in the linked list at index 28,182). This approach executes faster than open addressing. The array is named `itsItem` and the number of data values is `itsSize`, as illustrated in Figure 16.10.

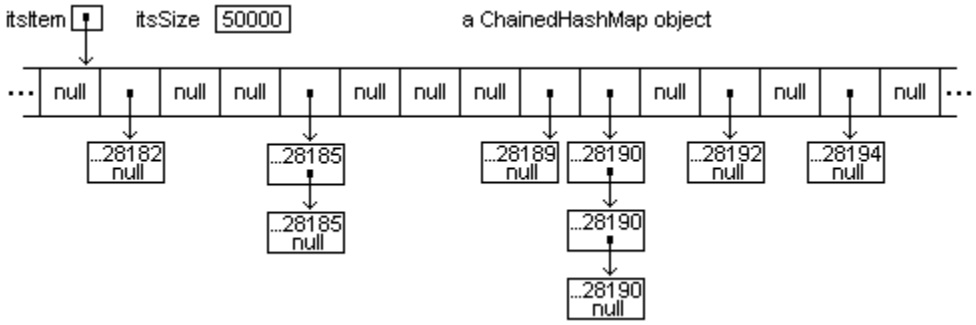

Figure 16.10 Example of ChainedHashMap with an array of 100,000 components

If the IDs are strings of characters (such as names), you calculate a number in the range from 0 to 99,999 from the Unicode values of the letters of the ID and store the data value in the linked list at that array index. Java provides a useful function for this purpose:

- The Object class has a method **`hashCode`**`()` that returns an int value. If `x.equals(y)` then `x.hashCode()` and `y.hashCode()` are the same int value.
- The Integer, Double, and Long classes override this `hashCode` method with an appropriate calculation from the bits in the corresponding primitive value.
- The String class overrides this `hashCode` method with a calculation from the Unicode values of the letters of the characters. Any class you use for IDs in a hash table should define a `hashCode` method for which `x.equals(y)` means that `x` and `y` have the same hash code value. If equal objects have the same `toString` value, you could use `x.toString().hashCode()` for your hash codes.

The calculation that Java's String class provides is as follows. For the chained hash table with an array of size 100,000, divide this `hashCode` value by 100,000 and use the remainder as the index into the array:

```
public int hashCode()
{  int result = 0;
   for (int k = 0;  k < length();  k++)
      result = result * 31 + charAt (k);
   return result;
}  //======================
```

Listing 16.14 The ChainedHashMap class of objects, some methods left for exercises

```
public class ChainedHashMap implements Mapping
{
   private Node[] itsItem = new Node[100000];    // all null
   private int itsSize = 0;


   /** Tell whether a key/value pair satisfies key.equals(id). */

   public boolean containsKey (Object id)
   {  if (id == null)
         return false;
      int index = Math.abs (id.hashCode()) % itsItem.length;
      return lookUp (id, itsItem[index]) != null;
   }  //======================


   /** Return the value in the key/value pair for which
    *  key.equals(id).  Return null if no such key/value pair. */

   public Object get (Object id)
   {  if (id == null)
         return null;
      int index = Math.abs (id.hashCode()) % itsItem.length;
      Node p = lookUp (id, itsItem[index]);
      return (p == null)  ?  null  :  p.itsData.getValue();
   }  //======================


   /** Return the Node containing the key/value pair for which
    *  key.equals(id).  Return null if no such key/value pair. */

   private static Node lookUp (Object id, Node list)
   {  return (list == null)  ?  null
             : list.itsData.getKey().equals (id)  ?  list
                   : lookUp (id, list.itsNext);
   }  //======================


   private static class Node
   {
      public MapEntry itsData;
      public Node itsNext;

      public Node (MapEntry data, Node next)
      {  itsData = data;
         itsNext = next;
      }  //======================
   }
}
```

You choose the array size of 100,000 when you think that the data set will normally contain between 50,000 and 100,000 data values. In general, choose an array size only somewhat more than the maximum you expect to store in the Mapping structure, not more than twice as large. For instance, you might use an array of size 10,000 for a company with five to eight thousand employees.

There will tend to be many cases where several people have the same last five digits of their SSNs, or have the same `hashCode` value for their names, so many array components will be empty and many will have more than one data value. However, it should be quite rare to have more than 3 or 4 data values stored on the same linked list. Thus the look-up process should take on average well under two comparisons, which is much better than the seventeen or more that binary search would require.

The average distribution of 100,000 randomly-chosen values in an array of size 100,000 will have 99.5% of data values in linked lists of length 4 or less, almost never with a linked list of more than 9 data values. The average search time will be about 1.5 comparisons. The average distribution of 50,000 randomly-chosen values in an array of 100,000 is roughly as follows, with an average search time of about 1.25 comparisons:

- 30,600 values are in 1-node lists.
- 15,000 values are in 7500 2-node lists.
- 3600 values are in 1200 3-node lists.
- 700 values are in 175 4-node lists.
- 100 values are in about 19 lists of 5 to 7 nodes (almost never more than 7).
- Over 60,000 array entries are null.

Warning Do not use a hash table unless you know that the keys being inserted have overridden the `hashCode()` method that the Object class provides. That basic `hashCode()` method is calculated from the RAM address of the object, not from the values stored in the object.

You will usually have fewer collisions if you use a prime number for the size of a hash table. Most sets of data have patterns where some sets of integers occur much less frequently than others as hash codes. For instance, all of them might be even numbers, which leaves half the table empty if its size is 10,000. If you choose a prime number such as 10,007 or 10,009 for the array length, you do not have this problem.

RAM usage

The space requirements for various Mapping implementations can be calculated with some effort. A **word** of storage is 4 bytes, sufficient space for a reference to an object or for an int value. If N is the number of data values stored, then an ArrayMap (Listing 16.6) takes between N and 1.5*N words of storage for `itsItem` (since the array grows by 50% each time it fills up). A NodeMap (Listing 16.8) takes 2*N words of storage for the N Nodes in use. A ChainedHashMap (Listing 16.14) or a java.util.HashMap (described in Section 16.9) takes about 1.5*N words of storage for `itsItem` and 2*N words of storage for the N Nodes in use, a total of 3.5*N words. A java.util.TreeMap (described in Section 16.9) takes about 4*N words (the tree Nodes store 3 values instead of 2, plus the red-black information it requires).

However, this count is misleading in that each object carries around an extra word of storage containing a reference to its own class. So you have to add N more words for each of the implementations just mentioned other than the ArrayMap.

This is where open addressing has a great advantage over chaining for maps that are based on hash tables: Open addressing uses less than half the space. You can strike a nice balance if you store a partially-filled array of entries in each component of `itsItem`:

- To add an entry where none was before, create an array of length 1 for it.
- To add an entry when you already have an array holding some entries for that same index, put it at the next available place in the array, unless the array is full, in which case you transfer the information to another array twice as large.
- To delete an entry, replace it by the entry at the end of its array (no shifting around).

This way, you use less than three-quarters of the space of chaining and you avoid probing, but at the cost of some execution time. The development of this **ArrayHashMap** implementation is left as a major programming project.

More on open addressing

Even if the collision problem is not too severe for open addressing, linear probing causes **clustering**: Putting an ID elsewhere fills up a space that many other IDs would otherwise go into. This can mean that, for data such that chaining might not have more than five data values for any one index, you could still have to search through dozens of entries to find what you are looking for with open addressing.

This problem can be avoided by using a probe interval that depends on the key you are putting in, instead of always using 1 or some other number. This solution is called **double-hashing**. For instance, you could probe forward by jumps of 1 to 13, where the size of the jump is calculated from the hash code of the ID you are looking for. 13 was chosen here because it is significantly larger than the largest number of collisions you expect in any one component (7 to 9 for arrays with more components than data values). The size of `itsItem` should always be a prime number when using double-hashing.

The accompanying structured design is a reasonable algorithm for finding the data value. The corresponding coding is in Listing 16.15, which is a start on an implementation of Mapping named **OpenHashMap** that uses this double-hashing technique.

STRUCTURED NATURAL LANGUAGE DESIGN of containsKey for OpenHashMap

1. Return `false` if the `id` is null, since Mappings do not allow null ids.
2. Use the hash code of the given `id` to calculate the `index` where the MapEntry with that `id` should be stored.
3. Repeat until you find at `index` the given `id` or an empty component...
   3a. Increment `index` by a jump value in the range 1 to 13, depending on the hash code of the `id`.
4. Return `true` if the given `id` was found, `false` if it was not.

Listing 16.15 The containsKey method in the OpenHashMap class of objects

```
public class OpenHashMap implements Mapping
{
   private MapEntry[] itsItem = new MapEntry[100000]; // all null
   private int itsSize = 0;


   public boolean containsKey (Object id)
   {  if (id == null)
         return false;
      int index = Math.abs (id.hashCode()) % itsItem.length;
      int jump  = Math.abs (id.hashCode()) % 13 + 1;
      while (itsItem[index] != null
                     && ! id.equals (itsItem[index].getKey())
         index = (index + jump) % itsItem.length;
      return itsItem[index] != null;
   }  //=====================
}
```

That still leaves a serious problem for open addressing that does not exist for chaining: What if you delete an entry? Say you wanted to put X at index 5000 and there was already a value there, so you tried index 5009 and found a value already there. Then you tried index 5018 and that was available. So you put X at index 5018. Later you remove the entry that is at index 5009. Later still you look for X, starting at 5000 of course. You see something there, so you look at index 5009. You see nothing there, so you think that X is not in the data set. But it is.

One standard solution for this problem is to simply forbid removal of a data value once you have put it in the data structure. That means that you have the `remove` method throw an UnsupportedOperationException. But that is a cop-out.

A better way is to create a special MapEntry different from all others. Call it NOENTRY. For instance, you could define `NOENTRY = new MapEntry(null, null)`. Now each time you remove a data value, you replace it by NOENTRY instead of by null. Each time you search for a data value, you treat NOENTRY as a non-match rather than as indicating that your search should stop. And each time you put a new value in, you replace NOENTRY if you find it while you are searching.

Note that the coding for the `containsKey` method for the OpenHashMap class does not have to be changed in any way to allow for this NOENTRY value. The rest of the development of OpenHashMap is left as a major programming project.

Exercise 16.53 Write the `size` and `isEmpty` methods for the ChainedHashMap class.
Exercise 16.54 (harder) Write the `put` method for the ChainedHashMap class.
Exercise 16.55* Write a ChainedHashMap `public void clear()`: The executor removes all of its MapEntries, leaving it empty.
Exercise 16.56* Write a ChainedHashMap `public boolean containsValue (Object ob)`: The executor tells whether it contains a MapEntry with the given value. Note that the `hashCode` method cannot be used here.
Exercise 16.57** Write a ChainedHashMap `public Object firstKey()`: The executor returns the key that is less than all other keys in the Mapping. Throw an Exception if the executor is empty. Assume `compareTo` does not return zero.
Exercise 16.58** Write a ChainedHashMap `public Mapping subMap (Object firstKey, Object lastKey)`: The executor returns a new Mapping with all the MapEntries whose key is greater-equal `firstKey` and less than `lastKey`. Throw an Exception if the usual exceptional situations arise.
Exercise 16.59** Write the `remove` method for the ChainedHashMap class.
Exercise 16.60** Write an Iterator nested class for the ChainedHashMap class, leaving `remove` unsupported. Have three instance variables: `itsPos` is the Node containing the data value that `next()` will return, `itsIndex` is the index of that Node's linked list in the array, and `itsItem` is the array itself.
Exercise 16.61** Revise the entire ChainedHashMap class to allow an ID to be null: Do not put it in the array, but instead add an instance variable `itsNullValue` to record the value that corresponds to a null key.
Exercise 16.62** Write a program to distribute 50,000 values randomly in an array of size 100,000, simulating the results for ChainedHashMap, and report the number of linked lists of each size. Generalize it to distribute N values in an array of size LEN. Run it with several different pairs of values N and LEN. See if your results confirm those reported in the text of this section. How do the results compare with 5000 distributed in 10,000 components and with 10,000 distributed in 20,000 components?

16.8 Further Development Of Scheme's List Class

The earlier Scheme analysis found most of the public methods that a List needs (Section 16.3), so you could now decide what a List object needs to know to perform those methods (its instance variables). It need not have an array. It is enough that a List object store just two Items: the first element on the List plus the list of all the rest of the elements. The latter gives you access to the second, third, or whatever elements. These two instance variables correspond to the `first` and `rest` methods in the List class.

You could name the List instance variables `itsFirst` and `itsRest`. The natural List constructor therefore has two parameters that supply the values for these two instance variables. Since there is no reason to change these values once they are assigned, you might as well make them final instance variables. Final instance variables do not have to be initialized in their declarations, as long as they are initialized in every constructor.

An empty list has no first element, so `itsFirst` is null. The value of `itsRest` for the empty List should also be null. There is no point in wasting space by having many empty lists, so you can have just one class variable `EMPTY` that is the only empty List in use.

The List class also needs a class variable `theDefs` to hold the definition of each function and a class variable `theSpecials` to hold the list of special keywords and predefined function names. Both of these are SchemeMaps, an implementation of the Mapping interface. `theDefs` starts off as an empty SchemeMap, but `theSpecials` has to have the various keywords and predefined function names put in it.

This all leads to the List class in Listing 16.16 (see next page), with the obvious methods filled in. Only three of the special WordUnits are listed in the `addThemAll` method; more can be added later. This coding leaves only two List methods to work on: `getValue` and `toString`.

The toString method in List

The `toString` method should print the List value the way the user would have typed it in. It is normal in Scheme to use the WordUnit `empty` to stand for the empty list, rather than two parentheses with nothing inside them. The structured design could therefore be as shown in the accompanying design block. The coding is in the lower part of Listing 16.16; note that `p` must be declared outside the for-statement.

STRUCTURED NATURAL LANGUAGE DESIGN of List's toString method

1. If the list is empty, then...
   1a. Return the String value "empty".
2. Start with a left parenthesis stored in some String variable; call it `result`.
3. For each element of the list except the last one, do...
   3a. Add its String form to `result`, followed by a blank to separate it from the next one.
4. Add the String form of the last element of the list to `result`, followed by a right parenthesis.
5. Return `result` as the answer.

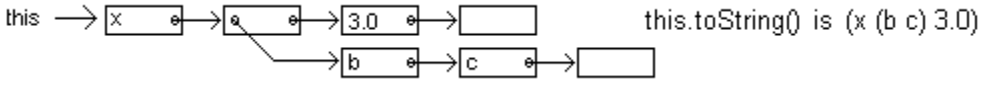

Figure 16.11 Effect of toString()

Listing 16.16 The List class except for the getValue method

```
public class List implements Item
{
   public static final List EMPTY = new List (null, null);
   private static SchemeMap theDefs = new SchemeMap();
   private static SchemeMap theSpecials = addThemAll();
   ////////////////////////////////
   private final Item itsFirst;
   private final List itsRest;


   public List (Item first, List rest)
   {  itsFirst = first;
      itsRest  = rest;
   }  //======================

   private static SchemeMap addThemAll()
   {  SchemeMap result = new SchemeMap();
      result.put (new WordUnit ("set!"), new Real (1));
      result.put (new WordUnit ("define"), new Real (2));
      result.put (new WordUnit ("+"), new Real (3));
      // many more will be added later
      return result;
   }  //======================


   public Item first()
   {  return itsFirst;
   }  //======================

   public List rest()
   {  return itsRest;
   }  //======================

   public boolean isEmpty()
   {  return itsRest == null;
   }  //======================


   public String toString()
   {  if (this.isEmpty())
         return "empty";
      String result = "(";
      List p;
      for (p = this;  ! p.itsRest.isEmpty();  p = p.itsRest)
         result += p.itsFirst.toString() + " ";
      return result + p.itsFirst.toString() + ")";
   }  //======================
}
```

Any function name defined by the user must be stored in `theDefs` along with its definition. The function name is the key in the Map object; the value stored with it is a two-element List object, one element for the heading and the other for the body. For instance, `(define (cube p) (* p (* p p)))` would be stored in `theDefs` with the key `cube` and the value `((cube p) (* p (* p p)))`. So when you get the value `def` from `theDefs` corresponding to `cube`, `def.itsFirst` is the heading `(cube p)` and `def.itsRest.itsFirst` is the body `(* p (* p p))`.

The getValue method

An empty list (normally used as a parameter of a function call) has the value EMPTY. A non-empty list might begin with a keyword or other previously-defined WordUnit, so `getValue` looks up that WordUnit in `theSpecials` Mapping. If it is there, the corresponding value will be a Real (a number) indexing the code segment that tells how it is to be evaluated. Otherwise `getValue` calculates the normal value.

The normalValue method

A reasonable design for the `normalValue` method is in the accompanying design block. Note the great advantage of throwing BadInputs when necessary. The `getValue` method might throw an Exception, or it might call the `normalValue` method which throws an Exception. Instead of the `getValue` method having to handle both of those situations with an if-statement, the `normalValue` method terminates without returning a value; the `getValue` method terminates without returning a value; and the main logic catches the throw and handles it.

STRUCTURED NATURAL LANGUAGE DESIGN of List's normalValue method

1. Retrieve the definition of `this.itsFirst` from `theDefs`.
2. If the definition is not found for this function call then...
   2a. Throw an Exception.
3. Find the value of each of the actual parameters, storing those values in a new List of arguments. Call it `actuals`.
4. Assign the arguments of `actuals` to the corresponding formal parameters. Throw an Exception if the two lists of parameters are not the same length.
5. Evaluate the body of the function using the newly-assigned values of the formal parameters.
6. Return the value obtained.

Step 5 of the `normalValue` design calls the `getValue` method in the class of the Item listed as the function body. The function body might be `(avg 3 2)`, which is a List object. So it calls List's `getValue` method, which calls the `normalValue` method. So `normalValue` can conceivably call a method that calls `normalValue`. Recursion will be needed here several times, because a List is a highly recursive structure: A List is a sequence of elements some of which may be Lists.

Steps 3 and 4 of the design cannot be done in just one or two Java statements, so it makes sense to have them be method calls that are developed next. The upper part of Listing 16.17 (see next page) gives the completed logic for the `normalValue` method. As an example, the `normalValue` method applied to `(avg 6 b)` when `b` is a variable with the value 13 produces the following:

- `this.itsFirst` is `avg`, the name of the function being called.
- `this.itsRest` is `(6 b)`, the list of arguments of the function call.
- `this.itsRest.allValues()` produces `(6 13)`, which is assigned to `actuals`.
- `def.itsFirst` is `(avg x y)`, the heading of the function.
- `def.itsFirst.itsRest` is `(x y)`, the list of formal parameters in the heading.
- The call of `assignValues` stores 6 in `x` and 13 in `y`.
- The call of `getValue` applies to `def.itsRest.itsFirst`, i.e. `(/ (+ x y) 2)`, which evaluates as `(/ (+ 6 13) 2)` and thus as 9.5.

Listing 16.17 The normalValue method in the List class

```
// public class List implements Item, continued

   public Item getValue()
   {  if (this.isEmpty())                                        //1
         return EMPTY;                                           //2
      Real index = (Real) theSpecials.get (this.first());        //3
      return index != null ? this.rest().specialValue (index)//4
                           : this.normalValue();                 //5
   }  //====================

   /** Find the definition of the function, evaluate the
    *  function call, and return the calculated value. */

   private Item normalValue()
   {  List def = (List) theDefs.get (this.itsFirst);             //6
      if (def == null)                                           //7
         throw new BadInput (this.itsFirst.toString()            //8
                             + " is not a function name");       //9
      List actuals = this.itsRest.allValues();                   //10
      ((List) def.itsFirst).itsRest.assignValues (actuals);      //11
      return def.itsRest.itsFirst.getValue();                    //12
   }  //====================

   private List allValues()
   {  return this.isEmpty()  ?  EMPTY  :  new List               //13
           (this.itsFirst.getValue(), this.itsRest.allValues());
   }  //====================

   private void assignValues (List actuals)
      // Protecting against different lengths is an exercise
   {  if ( ! this.isEmpty())                                     //15
      {  WordUnit parameter = (WordUnit) this.itsFirst;          //16
         parameter.setVariableValue (actuals.itsFirst);          //17
         this.itsRest.assignValues (actuals.itsRest);            //18
      }                                                          //19
   }  //====================
```

Private methods used by the normalValue method

The `normalValue` method calls `allValues()` to get a List of the values of all the elements in `this.itsRest`, which is the list of arguments. The logic of `allValues` is: There are two kinds of Lists, empty Lists and nonempty Lists. The list of values for an empty List `this` is the empty List itself. The list of values for a nonempty List `this` is the List consisting of the value of the first element of `this` followed by the List of the values of all the elements in the rest of `this`. And that is recursion! The logic is in the middle part of Listing 16.17, with the recursive call in boldface.

The `normalValue` method then calls `assignValues(actuals)`. This method call is to assign each formal parameter in its executor List the corresponding value in the `actuals` List. The logic of `assignValues` is: There are two kinds of Lists, empty Lists and nonempty Lists. An empty List need do nothing. A nonempty List `this` needs to assign `actuals.itsFirst` to its first element and then have the rest of the List be the executor of the method call `assignValues(actuals.itsRest)`. And that is recursion! The logic is in the lower part of Listing 16.17, with the recursive call in boldface. Figure 16.12 shows what happens during a call of `assignValues`.

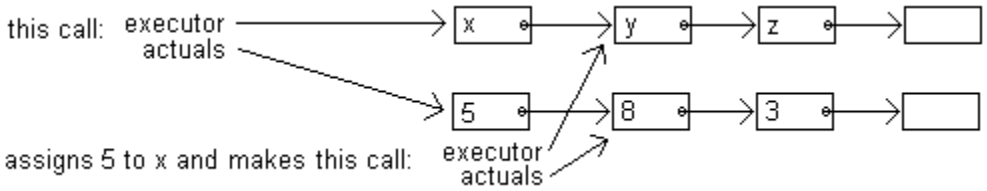

Figure 16.12 Example of a call of the assignValues(List) method

The logic developed here only gives the right results if every parameter of every Scheme function the user defines has a name different from any globally defined variable and from any other parameter. This problem can be corrected. A simple solution is to have the interpreter store the name of each parameter in a distinctive way, e.g. `p` in the `cube` function would be stored as `cube.p`. That solves all problems until the user writes a recursive Scheme function. We will not get into that in this book, however.

Exercise 16.63* Rewrite the List method `public String toString()` to use a StringBuffer object (described in Section 7.12) for efficiency.
Exercise 16.64* Revise the `assignValues` method in Listing 16.17 to throw a BadInput Exception if either List is longer than the other.
Exercise 16.65* Rewrite the `allValues` method without using recursion.
Exercise 16.66* Rewrite the `assignValues` method without using recursion.

*16.9 About Map, AbstractMap, HashMap, And TreeMap (*Sun Library)*

The **java.util.Map** interface prescribes twelve methods (in addition to those that any object has: `equals` and `hashCode`) -- the six described in Listing 16.4 (`put`, `remove`, `get`, `containsKey`, `isEmpty`, and `size`) plus another six:

- `someMap.clear()` removes all entries from the data structure.
- `someMap.containsValue(valueObject)` tells whether some entry contains `valueObject` as a value.
- `someMap.putAll(aMap)` in effect repeatedly calls `put` for each entry in `aMap`.
- `someMap.entrySet()` returns a Set object that contains all of the executor's entries. Sets are described in Chapter Fourteen; they are Collections that do not allow duplicate values (since any two entries have to have different keys).
- `someMap.keySet()` returns a Set object that contains all of the executor's keys.
- `someMap.values()` returns a Collection object containing all of the executor's values.

Note that Map does not have a method for directly obtaining an Iterator over the MapEntries. However, you can obtain an Iterator indirectly with the following statement:

```
Iterator it = someMap.entrySet().iterator();
```

The **java.util.Map.Entry** interface is a sub-interface of Map. It has the `getKey()` and `getValue()` methods described in Listing 16.5, as well as a `setValue()` method so you can change the value associated with a given ID.

The **java.util.AbstractMap** class makes it easy to implement the Map interface. It has only one abstract method, the `entrySet` method. If you have a class that extends AbstractMap and replaces the `entrySet` method, it will be a full implementation of Map, though it will not allow modifications of the Entries in the Map. If you also replace the `put` method (which otherwise throws an UnsupportedOperationException) in AbstractMap and/or the `remove` method, the Map will be modifiable.

TreeMap objects (from java.util)

The **TreeMap** class is a full implementation of Map that uses a binary search tree (to be discussed in detail in the next chapter). Keys must be Comparable. In addition, it colors each node either red or black according to a set of rules that guarantees that the tree cannot become too out of balance. Specifically, the worst-case time for `get`, `put`, and `containsKey` can never be more than twice as much as log2(N), where N is the number of entries in the tree. The class includes the following constructors:

- `new TreeMap()` creates an empty data structure.
- `new TreeMap(aMap)` creates a data structure containing each entry in `aMap`.

Actually, TreeMap implements **SortedMap**, an extension of the Map interface that prescribes that the iterator will produce Entry objects in ascending order of keys. The additional methods include `firstKey()` and `lastKey()` as well as `subMap (fromKey, toKey)` that returns a view of the MapEntries whose keys are at least `fromKey` but less than `toKey`.

HashMap objects (from java.util)

The **HashMap** class is another full implementation of Map provided in the Sun library. The `get`, `put`, and `containsKey` operations have an execution time that is big-oh of 1, i.e., in constant time independent of the number of values in the Map. The HashMap class allows a null key and null values. It includes the following constructors:

- `new HashMap()` creates an empty data structure.
- `new HashMap(aMap)` creates a data structure containing each entry in `aMap`.

The HashMap class uses a hash table and each Object's `hashCode` method to get this speed, similar to ChainedHashMap. This implementation of Map makes no guarantee as to the order of the entries. The order might even change substantially over time.

16.10 Review Of Chapter Sixteen

About the Sun standard library java.util.Map interface:

- `someMap.containsKey(Object id)` tells whether id/someValue is in the Map.
- `someMap.get(Object id)` returns `someValue` when id/someValue is in the Map, and otherwise returns null.
- `someMap.put(Object id, Object someValue)` adds that id/someValue pair to the Map, replacing any id/otherValue pair with the same id.
- `someMap.remove(Object id)` removes the id/someValue pair from the Map, returning `someValue`; but it returns null if the id/someValue pair was not in the Map.
- `someMap.isEmpty()` tells whether the Map object contains no id/someValue pairs at all.
- `someMap.size()` tells the number of id/someValue pairs in the Map.
- The Map interface has several more methods, for which the AbstractMap class gives generic implementations as long as you provide coding for the `entrySet` method.

About the Sun standard library Iterator class:

- `someIterator.hasNext()` tells whether a collection of objects has an object that `next` will retrieve.
- `someIterator.next()` advances to the next element and returns that Object, if there is one. It throws `java.util.NoSuchElementException` if there is none.
- `someIterator.remove()` takes out of the collection the element just returned by a call of `next`. It throws `java.lang.IllegalStateException` if that cannot be done. An Iterator implementation that does not allow removal should have `remove` throw a `java.lang.UnsupportedOperationException`.

Other vocabulary to remember:

- An **interpreter** translates one line of source code (written by the programmer) to object code (executable by the chip) at a time. It executes that one line before going on to translate the next line.
- A **naturally recursive** data structure is one that contains a smaller such data structure as part of itself, except when it is empty.
- A linked list is made up of nodes, each of which contains a reference to one piece of data and a reference to another node (or to null if it is the last node in the list). A **simple linked list** has data in each node. If N is the number of data values in a list, then a **linked list with trailer node** has N+1 nodes, the last node having null in place of its data.

Answers to Selected Exercises

16.1
```
(define (cent-to-fahr x)  (+  32  (*  1.8  x)))    ==> cent-to-fahr is defined
(cent-to-fahr 20)                                   ==> 68
```
16.2
```
(define (hypotenuse a b) (sqrt (+ (* a a) (* b b))))    ==> hypotenuse is defined
(hypotenuse 6 8)                                         ==> 10
```
16.4
```
public Real (double par)
{     itsValue = par;
}
public Item getValue()
{     return this;    // because a Real is a constant
}
```
16.5
```
public String toString()
{     return "" + itsValue;
}
public double getNumber()
{     return itsValue;
}
```
16.6
```
output.append (parser.parseInput (field.getText()).getValue().toString());
```
16.7
```
public static void inOrOut (String word, String definition)
{     if (theDictionary.containsKey (word))
            theDictionary.remove (word);
      else
            theDictionary.put (word, definition);
}
```
16.8
```
public boolean hasDefinition()
{     for (int k = 0;  k < itsSize;  k++)
      {     if (theDictionary.containsKey (itsItem[k]))
                  return true;
      }
      return false;
}
```
16.9
```
public boolean equals (Object ob)
{     return ob instanceof WordUnit && this.itsValue.equals (((WordUnit) ob).itsValue);
}
```
16.10 x.first() is a. x.rest() is (b c d e). x.rest().first() is b.

16.11
```
public boolean hasThree()
{     return ! isEmpty() && ! rest().isEmpty() && ! rest().rest().isEmpty();
}
```

```
16.12   public void putAll (Mapping given)
        {   Iterator it = given.iterator();
            while (it.hasNext())
            {   MapEntry entry = (MapEntry) it.next();
                this.put (entry.getKey(), entry.getValue());
            }
        }
16.19   public int size()
        {   return itsSize;
        }
16.20   public int countLess (Comparable id)
        {   int count = 0;
            for (int k = 0;  k < itsSize;  k++)
            {   if (((Comparable) itsItem[k].getKey()).compareTo (id) < 0)
                    count++;
            }
            return count;
        }
16.21   public Object remove (Object id)
        {   int loc = lookUp (id);
            if (loc < 0)
                return null;
            Object valueToReturn = itsItem[loc].getValue();
            itsItem[loc] = itsItem[itsSize - 1];
            itsSize--;
            return valueToReturn;
        }
16.22   public ArrayMap reverse()
        {   ArrayMap valueToReturn = new ArrayMap();
            valueToReturn.itsItem = new MapEntry [this.itsItem.length];
            valueToReturn.itsSize = this.itsSize;
            for (int k = 0;  k < this.itsSize;  k++)
                valueToReturn.itsItem[k] = this.itsItem[this.itsSize - 1 - k];
            return valueToReturn;
        }
16.23   public Object put (Object id, Object value)
        {   if (id == null)
                return null;
            int loc = lookUp (id);
            if (loc < 0)
            {   itsItem[itsSize] = new MapEntry (id, value);
                itsSize++;
                return null;
            }
            Object valueToReturn = itsItem[loc].getValue();
            itsItem[loc] = new MapEntry (id, value);
            return valueToReturn;
        }
16.31   public Object secondOne()
        {   return (itsData == null || itsNext.itsData == null)  ?  null  :  itsNext.itsData.getValue();
        }
16.32   public void clear()
        {   itsData = null;
            itsNext = null;   // the other nodes are garbage-collected.
        }
16.33   public void swapTwo()
        {   if (this.itsData == null || itsNext.itsData == null)
                return;   // a void method forbids returning a value
            MapEntry saved = this.itsData;
            this.itsData = itsNext.itsData;
            itsNext.itsData = saved;
        }
16.34   Insert the following directly after the only left brace inside the body of that put method:
        if ( ! isEmpty())
        {   itsNext = new NodeMap (new MapEntry (id, value), itsNext);
            return null;
        }
16.35   public boolean equals (NodeMap given)
        {   return this.isEmpty()  ?  given.isEmpty()  :  ! given.isEmpty()
                    && this.itsData.equals (given.itsData) && this.itsNext.equals (given.itsNext);
        }
```

16.36
```
private Object putRecursive (Comparable id, Object value)
{   if (this.isEmpty()  ||  id.compareTo (this.itsData.getKey()) < 0)
    {   this.itsNext = new NodeMap (this.itsData, this.itsNext);
        this.itsData = new MapEntry (id, value);
        return null;
    }
    if (id.compareTo (this.itsData.getKey()) > 0)
        return this.itsNext.putRecursive (id, value);
    MapEntry save = this.itsData;
    this.itsData = new MapEntry (id, value);
    return save.getValue();
}
```
16.37 Use the put method of the preceding exercise. Replace the body of lookUp by the following coding:
```
return this.isEmpty() || ((Comparable) this.itsData.getKey()).compareTo (id) >= 0
            ?  this  :  this.itsNext.lookUp (id);
```
Replace the body of the containsKey method by the following:
```
NodeMap loc = lookUp (id);
return ! loc.isEmpty() && loc.itsData.getKey().equals (id);
```
Replace the loc.isEmpty() condition in the get and remove coding by the following condition:
```
loc.isEmpty() || ! loc.itsData.getKey().equals (id)
```
16.38
```
public NodeMap intersection (NodeMap given)
{   if (this.isEmpty() || given.isEmpty())
        return new NodeMap (null, null);
    int comp = ((Comparable) this.itsData.getKey()).compareTo (given.itsData.getKey());
    return comp > 0  ?  this.intersection (given.itsNext)
                :  comp < 0  ?  this.itsNext.intersection (given)
                :  new NodeMap (this.itsData, this.itsNext.intersection (given.itsNext));
}
```
16.46 Declare in the MapIt class nested in NodeMap:
```
private Node itsPrevious;
public void remove()
{   if (itsPrevious == itsPos)   // to signal that you have already removed it or never called next()
        throw new IllegalStateException ("nothing to remove");
    itsPrevious.itsData = itsPos.itsData;
    itsPrevious.itsNext = itsPos.itsNext;
    itsPos = itsPrevious;   // a signal that no further remove is allowed
}
```
Add this statement as the third-to-last statement in next and as the last one in the constructor:
```
itsPrevious = itsPos;
```
16.47
```
public void putAll (Mapping given)
{   java.util.Iterator it = given.iterator();
    while (it.hasNext())
    {   MapEntry toAdd = (MapEntry) it.next();
        this.put (toAdd.getKey(), toAdd.getValue());
    }
}
```
16.53
```
public int size()
{   return itsSize;
}
public boolean isEmpty()
{   return itsSize == 0;
}
```
16.54
```
public Object put (Object id, Object value)  // compare with put in Listing 16.9
{   if (id == null)
        return null;
    int index = Math.abs (id.hashCode()) % itsItem.length;
    Node loc = lookUp (id, itsItem[index]);
    if (loc == null)
    {   itsItem[index] = new Node (new MapEntry (id, value), itsItem[index]);
        itsSize++;
        return null;
    }
    Object valueToReturn = loc.itsData.getValue();
    loc.itsData = new MapEntry (id, value);
    return valueToReturn;
}
```