CHAPTER THIRTEEN

Object-based Abstractions

13.1 Introduction

In Chapter 6, we emphasized that each abstract data type should have a collection of operations that was appropriate to how the type needed to be used. This same general principle applies even when we consider types of objects that can be modified. Yet up until now, the only modifiable objects we've seen—vectors and two-dimensional tables—have supported only one particular repertoire of operations. You can put a new value into a numerically specified location or get the current value out from a numerically specified location, which reflects the close link between vectors and the numerically addressed memory of machines like SLIM, as we pointed out in Chapter 11. Yet sometimes our programming would benefit from a different set of operations. For example, we might want an operation that retrieves the most recently stored value, independent of the location at which it was stored.

In this chapter, we'll learn how to work with abstract data types that can be modified, like vectors, but that support operations determined by our needs rather than by the nature of the underlying memory. We'll also see how we can think clearly about objects that undergo change, by focusing on invariant properties that are established when the object is first constructed and preserved by each modification of the object. Finally, we'll see some specific commonly used examples of modifiable data structures. In particular, we'll see a stack-like structure that is useful when evaluating arithmetic expressions, a queue structure that is useful for managing waiting lists fairly, and a tree structure that can be used to efficiently store and retrieve information, such as the collection of movies owned by a video store that constantly acquires new releases. In fact, you'll apply the structure to exactly this problem in the application section at the end of the chapter.

13.2 Arithmetic Expressions Revisited

Recall that we wrote a procedure called `evaluate` in Section 8.3 that computes the value of standard arithmetic expressions that are fully parenthesized. For example, you might have the following interaction:

```
(evaluate '((3 + 4) * (9 - (2 * 3))))
21
```

On the other hand, `evaluate` would be unable to cope with an expression such as

```
'((3 + 4) * 9 - 2 * 3)
```

even though it is a perfectly valid arithmetic expression whose value is 57. Attempting to evaluate the latter expression results in an error because `evaluate` only handles expressions that are numbers or three-element lists. Furthermore, the three-element lists must have a left operand, an operator, and a right operand, in that order. Because the operands had to be expressions of the same form, evaluation was accomplished by recursively applying the value of the operator to the values of the two operands.

We would like to extend our `evaluate` procedure so that it can handle more general arithmetic expressions, such as the preceding one. What makes this difficult is specifying which operands a given operator should operate on. For example, consider the following two expressions:

```
'(3 - 4 + 5)

'(3 - 4 * 5)
```

In the first case, the `-` operates on `3` and `4`, whereas in the second case, the `-` operates on `3` and the result of `4 * 5`. Why? Because the `*` operator has *higher precedence* than the `+` does. Normally, when you have an expression with more than one operator in it, you do the operations with higher precedence first and then do the others, where the precedence convention with the four operators `+ - * /` is that there are two levels, one for `*` and `/` and another for `+` and `-`, and the first level is higher than the second. If you have an expression with two consecutive operators with the same precedence (for instance, `'(10 - 3 - 2)`), you do those operations working from left to right.

There is some flexibility in these rules; for instance in evaluating an expression such as `'(2 + 5 + 5)`, many people would do the second addition first. However, we can always do our operations in a left-to-right order *as long as* we always remember that when we have two consecutive operators and one has higher precedence, we do that one first. Here is an example of figuring out the value of an expression using this approach:

$$\begin{gathered}
3 + \underbrace{2 * 4} - 40/5 \\
\underbrace{3 + 8} - 40/5 \\
11 - \underbrace{40/5} \\
\underbrace{11 - 8} \\
3
\end{gathered}$$

We can therefore view the general evaluation process as a sequence of *reductions*, where each reduction consists of a single operation on two numbers. In the example above, we did four of these reductions.

If we look at expressions with parentheses, such as $3*(2+4)$, we can use a similar process involving reductions. We would reduce $2 + 4$ to 6, yielding $3 * (6)$. Then we could reduce the parenthesized (6) to a plain 6, yielding $3 * 6$, which we would reduce to 18. We'll put off parenthesized expressions for later in this section and stick with unparenthesized expressions for now. However, in both cases the key action is the reduction.

This viewpoint allows us to come up with a method for evaluating unparenthesized expressions from left to right, *provided* we can maintain a little bit of memory. The basic idea is to scan the expression from left to right and do a reduction once we know it should be done. How do we know when to reduce? Consider the example of $3 + 2*4 - 40/5$. Having scanned through $3 + 2$, we need to check the next symbol to determine whether to reduce $3 + 2$. Seeing that the next symbol is an operator of higher precedence, we scan further, eventually reaching $3 + 2*4$. Because the next symbol is an operator of equal or lower precedence, we determine that a reduction is in order and replace the scanned portion with $3 + 8$. This continues through the remainder of the list, reducing until we have a single number.

What sort of storage mechanism do we need? First note that the basic data being manipulated consists of the numbers and operators in the expression. In a sense, numbers and operators are the "words" from which our expressions are formed. We will adopt the common computer science convention of referring to these basic words as *tokens*. Thus, "scanning down the expression" means `cdr`-ing down the list of tokens. As we scan, we'll keep a collection of already scanned (or reduced) tokens. Each time we scan a new token, we either *shift* it onto the collection of already scanned (or reduced) tokens, or we perform a *reduction* on that latter collection. This collection of already scanned or reduced tokens is precisely the memory storage mechanism we need.

What operations must we perform on this collection? Well, we either *shift* something onto it, or we *reduce* the three most recently scanned tokens by performing the operation. In either case, we need to access the most recently scanned tokens.

Figuratively, we can view this collection of scanned or reduced tokens as a *stack* of tokens, where we access the stack from the top (i.e., the most recently scanned or reduced token). Shifting a token means putting it on top of the stack; reducing means removing the top three items from the stack, performing the operation, and putting the resulting value back on top of the stack.

Actually, there are two other actions we might need to do besides shifting and reducing:

- If we have successfully finished evaluating an expression, we should *accept* it and return the top item on the stack as the value.
- If we encounter an *error*, we should report it and stop the processing altogether.

So we have a total of four possible actions during the course of our processing: shift, reduce, accept, and error. Each of these actions can be easily accomplished, provided we can access the top items on the stack of processed tokens.

One question remains before we can view this as a full-blown algorithm: Given our current state (the stack of processed tokens and the newly scanned token), which of the four actions do we take? The key point here is that we can determine the action with only knowledge of the top two elements on the stack and the next scanned token (or knowledge that we have already reached the end of the expression). To illustrate this, consider the table in Figure 13.1, which describes the sequence of actions taken in order to reduce the expression $3+2*4-40/5$. Note that we have added a special "terminating" symbol `$` at the bottom of the expression stack and the end of the

expression stack	rest of expression	next action
`$`	`3 + 2 * 4 - 40 / 5 $`	shift
`$ 3`	`+ 2 * 4 - 40 / 5 $`	shift
`$ 3 +`	`2 * 4 - 40 / 5 $`	shift
`$ 3 + 2`	`* 4 - 40 / 5 $`	shift
`$ 3 + 2 *`	`4 - 40 / 5 $`	shift
`$ 3 + 2 * 4`	`- 40 / 5 $`	reduce
`$ 3 + 8`	`- 40 / 5 $`	reduce
`$ 11`	`- 40 / 5 $`	shift
`$ 11 -`	`40 / 5 $`	shift
`$ 11 - 40`	`/ 5 $`	shift
`$ 11 - 40 /`	`5 $`	shift
`$ 11 - 40 / 5`	`$`	reduce
`$ 11 - 8`	`$`	reduce
`$ 3`	`$`	accept

Figure 13.1 Evaluation of $3 + 2 * 4 - 40/5$

stack	next token		
top	$	op	num
$	error	error	shift
op	error	error	shift
num	reduce or accept	shift or reduce	error

Figure 13.2 Action table for unparenthesized expressions

expression. Strictly speaking, this symbol is not really needed; after all, we could easily test to see whether the expression stack or the rest of the expression is empty. However, having a symbol that indicates these conditions will be helpful when we finally get around to writing the code because we will then always be testing tokens to determine our action.

Although the example in Figure 13.1 gives some notion of how to determine the next action, we need to be more precise. We increase the precision in Figure 13.2, where we give a table that *nearly* specifies which action to take, given the top of the stack and the next scanned token. In this table the row headings refer to the top of the expression stack, the column headings refer to the next token, *op* refers to any of the four operators, and *num* refers to any number. Therefore, if the top of the expression stack is an operator and the next token is a number, we should surely shift; however, if the next token is an operator, we take the error action because no legal expression can have two consecutive operators.

Exercise 13.1

Explain why each of the five error conditions in the table in Figure 13.2 is in fact an error. In each case, give an example of an expression that has the given error, clearly marking where the error occurs.

We said that the table *nearly* specifies the action, because in two cases we need more information:

- If the top of the stack is a number and the next token is $, we accept if the token below the top of the stack is $ (because the expression is then fully reduced), and otherwise we reduce.

- If the top of the stack is a number and the next token is an operator, we shift if the token below the top of the stack is either not an operator or else is an operator of lower precedence than the next token, and otherwise we reduce (this is our evaluation rule).

In both cases, we need only one more piece of information: the token below the top of the stack. This explains our statement that to determine the next action, we at most need to know the top two elements on the stack and the next scanned token.

Exercise 13.2

Work through the steps in evaluating $30 - 7 * 3 - 1$. We recommend that you do this using index cards, at least the first time, to get more of a feel for what is going on. If you want to document your work, you can then do the evaluation a second time in tabular form, using the format shown in Figure 13.1.

To do the evaluation using index cards, you'll use two piles, one for the stack and the other for the remaining input (that is, the two piles of cards correspond to the first two columns in Figure 13.1). The pile that represents the remaining input should start out with eight cards in it, with 30 on the top, then $-$, 7, $*$, 3, $-$, 1, and finally \$ on the bottom. The other pile, representing the stack, should start out with just a \$ card. You'll also need a few blank cards for when you do reductions.

At each step, you should look at the top cards from the two piles and use those to locate the proper row and column in the action table of Figure 13.2. If the action table entry is one of the two with "or" in it, you'll need to peek down at the second card in the stack and use the rules specified above to determine the correct action.

If the action is shift, you just move the top card from the remaining-input pile to the stack. If the action is reduce, you take the top three cards off the stack, do the computation, write the answer on a blank card, and put that onto the stack. (Be sure to get the order right: The card that was on top of the stack is the right operand, whereas the one that was three deep is the left operand.) If the action is accept, the top card on the stack tells you the answer. If the action is error, you must have done something wrong because the expression we started with, $30 - 7 * 3 - 1$, was well formed.

Having pretty much taken care of unparenthesized expressions (except for writing the code), let's now consider expressions that include parentheses, for example the expression $(3 + 2) * 4 - 40/5$. First off, this means we must add new tokens (words) to our expression vocabulary, namely, left and right parentheses. However, this leads to a bit of a problem, because parentheses are not legal symbols in Scheme; after all, they are used to delimit Scheme lists. We will get around this problem by using strings instead of lists to pass our expressions to `evaluate`. Thus, we would compute the value of the example expression by evaluating the expression

```
(evaluate "(3+2)*4-40/5")
```

Rather than getting bogged down with details involving strings and characters, we describe a procedure called `tokenize` in the sidebar Strings and Characters, later in the chapter. It converts a string to the list of tokens it represents. To illustrate how the procedure `tokenize` works, suppose you have the following interaction:

```
(tokenize "(3+2)*4-40/5")
(lparen 3 + 2 rparen * 4 - 40 / 5 $)
```

The return value of `tokenize` is a list consisting of numbers and symbols, where the special symbols `lparen` and `rparen` represent left parentheses and right parentheses, respectively, and the terminating symbol `$` is at the end of the list.

So how do we extend our algorithm to parenthesized expressions? If we want to continue with our left-to-right approach, once we encounter a parenthesized subexpression, we need to fully reduce it to the number it represents before passing beyond it. Figure 13.3 illustrates how the shift/reduce algorithm might work by evaluating the expression $(3 + 2) * 4 - 40/5$. In a sense, a right parenthesis acts much like the $ symbol, forcing reductions until the subexpression has been fully

expression stack	rest of expression	next action
`$`	`( 3 + 2 ) * 4 - 40 / 5 $`	shift
`$ (`	`3 + 2 ) * 4 - 40 / 5 $`	shift
`$ ( 3`	`+ 2 ) * 4 - 40 / 5 $`	shift
`$ ( 3 +`	`2 ) * 4 - 40 / 5 $`	shift
`$ ( 3 + 2`	`) * 4 - 40 / 5 $`	reduce
`$ ( 5`	`) * 4 - 40 / 5 $`	shift
`$ ( 5 )`	`* 4 - 40 / 5 $`	reduce
`$ 5`	`* 4 - 40 / 5 $`	shift
`$ 5 *`	`4 - 40 / 5 $`	shift
`$ 5 * 4`	`- 40 / 5 $`	reduce
`$ 20`	`- 40 / 5 $`	shift
`$ 20 -`	`40 / 5 $`	shift
`$ 20 - 40`	`/ 5 $`	shift
`$ 20 - 40 /`	`5 $`	shift
`$ 20 - 40 / 5`	`$`	reduce
`$ 20 - 8`	`$`	reduce
`$ 12`	`$`	accept

Figure 13.3 Evaluation of $(3 + 2) * 4 - 40/5$

reduced. When that has been accomplished, the right parenthesis is then pushed onto the stack, and the stack is reduced by replacing the parenthesized number with the single number.

Why do we shift a right parenthesis onto the stack, only to immediately throw it away? We are adopting the viewpoint that things get simplified by reduction alone, which occurs at the top of the stack. In our extended algorithm we allow another form of reduction besides performing an arithmetic operation: A parenthesized expression enclosing a number is reduced to the number itself, stripping away the parentheses; this is the reduction that changes the (5) on line 7 to the 5 on line 8 in Figure 13.3. A consequence of this viewpoint is that we must ensure that when the right parenthesis is finally pushed onto the stack, the matching parentheses enclose a simple number, not a more complex expression. This explains why a right parenthesis acts like the $ symbol when it is the next token: It must force a full reduction of the expression on top of the stack back to the matching left parenthesis.

As with unparenthesized expressions, this algorithm is made *nearly* precise by giving a table that explains what to do, given the top of the stack and the next token in the expression. We do this in Figure 13.4, which extends the action table of Figure 13.2 to include left and right parentheses.

Mismatched parentheses are detected by two of the error cases in the num row of the table, that is, when the stack top is a number. If the next token is $ and a left parenthesis lies below the number, we have the kind of error that the input string `"(3"` exemplifies. If, on the other hand, the next token is a right parenthesis and a $ lies below the number on the stack, we have an error like `"3)"`.

stack	next token				
top	$	op	num	(	)
$	error	error	shift	shift	error
op	error	error	shift	shift	error
num	reduce, accept, or error	shift or reduce	error	error	shift, reduce, or error
(	error	error	shift	shift	error
)	reduce	reduce	error	error	reduce

Figure 13.4 Action table for general expressions

Many of the more complicated parenthesization mismatches reduce to one of the above two cases. For example, in the expression

```
( 3 + 3 * 5 ) ) + 56
```

the underscored right parenthesis is erroneous, because it has no matching left parenthesis. How can we detect this? Well, in the course of processing the expression up to, but not including, the erroneous right parenthesis, the expression will be reduced to

```
18 ) + 56
```

Because the expression on the top of the stack, 18, is fully reduced, the error is detected by the fact that the token below the top of the stack is a $ rather than a left parenthesis matching the underscored right parenthesis.

Exercise 13.3

Explain, using examples, the eight additional error conditions in the table in Figure 13.4, beyond those explained in the foregoing and in Exercise 13.1.

Exercise 13.4

Let's consider some of the regularities in this extended table.

a. Why are the columns headed by num and `(` identical?
b. Why are the rows headed by `$` and `(` identical?
c. Why are these latter rows identical to the row headed by op?

All that remains to make the algorithm precise is to complete our explanation of the additional ambiguous entry in the table, namely, when the top of the stack is a number and the next token is a right parenthesis. Because we showed in the preceding how to detect an error in this situation, we need only explain how to distinguish a shift from a reduce. As we said, we must reduce if the top of the stack is a simple arithmetic expression (i.e., an operator and two numeric operands), because we only want to shift the right parenthesis when the parenthesized expression has been fully reduced. This situation can be detected by checking to see whether the token below the top of the stack is an operator or a left parenthesis. If it is an operator, we should reduce, whereas if it is a left parenthesis, we should shift the right parenthesis onto the stack.

Exercise 13.5

Work through the evaluation of $30 - 7 * (3 - 1)$ using the same technique as in Exercise 13.2.

To finally code up this algorithm, we need to clearly specify the abstract data type *Stack*. As the term is commonly used, a stack allows you to access its top element and to add or delete an item at the top (these latter two operations are generally called *push* and *pop*, respectively). However, we could use something slightly more powerful for our program because we will need to access items below the top as well. For this reason, we are going to use an ADT that we call an *RA-stack* (for Random Access stack), which allows access to all of its elements, while still limiting addition and deletion to the top. Using Scheme notation, we specify the operations of random access stacks as follows:

```
(make-ra-stack)
  ;; returns a newly created empty stack.

(empty-ra-stack? ra-stack)
  ;; returns #t if ra-stack is empty, otherwise #f.

(height ra-stack)
  ;; returns the height (i.e., number of elements) in ra-stack.

(top-minus ra-stack offset)
  ;; returns the element which is offset items below the top of
  ;; ra-stack, provided 0 <= offset < (height ra-stack).
  ;; In particular, (top-minus ra-stack 0) returns the top of
  ;; ra-stack, provided ra-stack is non-empty.

(pop! ra-stack)
  ;; removes the top element of ra-stack, provided ra-stack is
  ;; non-empty.
  ;; The return value is the modified ra-stack.

(push! ra-stack item)
  ;; pushes item onto the top of ra-stack.
  ;; The return value is the modified ra-stack.
```

The two operators `pop!` and `push!` are of particular interest because they cause the stack parameter `ra-stack` to change (mutate); in this respect, they are similar to `vector-set!`. Because the ADT RA-stack allows mutation, it is called a *mutable* *data type*. Another way to say this is that RA-stacks are *objects* rather than *values*. Mutable data types are very useful for modeling phenomena that change in time; in our case, the expression stack changes as the evaluator works.

Turning finally to our version of `evaluate`, most of the work is done by the internally defined procedure `process`, which scans down the expression in the manner described above. The list of as-of-yet-unscanned tokens is maintained through the parameter `rest-of-expr`. `Process` is initialized by first using a `let` to define an empty stack `expr-stack`, then pushing the special token `$` onto `expr-stack`, and finally calling `process` with the tokenization of the input string. Here is the code:

```
(define evaluate
  (lambda (expression-string)
    (let ((expr-stack (make-ra-stack)))
      (define process
        (lambda (rest-of-expr)
          (let ((next-token (car rest-of-expr)))
            (cond ((accept? expr-stack next-token)
                   (top-minus expr-stack 0))
                  ((reduce? expr-stack next-token)
                   (reduce! expr-stack)
                   (process rest-of-expr))
                  ((shift? expr-stack next-token)
                   (push! expr-stack next-token)
                   (process (cdr rest-of-expr)))
                  (else  ; error
                   (error "EVALUATE: syntax error"
                          expr-stack rest-of-expr))))))
      (push! expr-stack '$)
      (process (tokenize expression-string)))))
```

Note that the determination of the next action is offloaded to three predicate procedures `reduce?`, `accept?`, and `shift?`. Similarly, the reduce action has been spun off to the procedure `reduce!`.

The three predicate procedures simply implement the action table in Figure 13.4:

```
(define accept?
  (lambda (expr-stack next-token)
    (if (and (number? (top-minus expr-stack 0))
             (equal? next-token '$))
        (equal? (top-minus expr-stack 1) '$)
        #f)))
```

```
(define reduce?
  (lambda (expr-stack next-token)
    (let ((stack-top (top-minus expr-stack 0)))
      (cond ((number? stack-top)
             (let ((stack-second (top-minus expr-stack 1)))
               (cond ((equal? next-token '$)
                      (operator? stack-second))
                     ((operator? next-token)
                      (and (operator? stack-second)
                           (not (lower-precedence?
                                 stack-second
                                 next-token))))
                     ((equal? next-token 'rparen)
                      (operator? stack-second))
                     (else #f))))
            ((equal? stack-top 'rparen)
             (or (equal? next-token '$)
                 (operator? next-token)
                 (equal? next-token 'rparen)))
            (else #f)))))

(define shift?
  (lambda (expr-stack next-token)
    (let ((stack-top (top-minus expr-stack 0)))
      (cond ((or (operator? stack-top)
                 (member stack-top '($ lparen)))
             (or (number? next-token)
                 (equal? next-token 'lparen)))
            ((number? stack-top)
             (let ((stack-second (top-minus expr-stack 1)))
               (cond ((operator? next-token)
                      (or (not (operator? stack-second))
                          (lower-precedence? stack-second
                                             next-token)))
                     ((equal? next-token 'rparen)
                      (equal? stack-second 'lparen))
                     (else #f))))
            (else #f)))))
```

The procedure `reduce!` has two branches, corresponding to whether we are "unparenthesizing" a parenthesized number or performing an arithmetic operation.

```
(define reduce!
  (lambda (expr-stack)
    (cond ((equal? (top-minus expr-stack 0) 'rparen)
           (let ((value (top-minus expr-stack 1)))
             (pop! expr-stack)  ; remove rparen
             (pop! expr-stack)  ; remove the value
             (pop! expr-stack)  ; remove lparen
             (push! expr-stack value)))
          (else ; a simple arithmetic operation
           (let ((left-operand  (top-minus expr-stack 2))
                 (operator      (top-minus expr-stack 1))
                 (right-operand (top-minus expr-stack 0)))
             (pop! expr-stack)  ; remove the right operand
             (pop! expr-stack)  ; remove the operator
             (pop! expr-stack)  ; remove the left operand
             (push! expr-stack
                    ((look-up-value operator)
                     left-operand
                     right-operand)))))))
```

Finally, the procedure `look-up-value` was written in Section 8.3. The remaining auxiliary routines can be implemented as follows:

```
(define operator?
  (lambda (token)
    (member token '(+ - * /))))

(define lower-precedence?
  (lambda (op-1 op-2)
    (and (member op-1 '(+ -))
         (member op-2 '(* /)))))
```

13.3 RA-Stack Implementations and Representation Invariants

We now address the task of implementing RA-stacks. As with all ADTs, we have great freedom in choosing how we represent them and implement their operators; our only real constraint is that RA-stacks must behave as they are supposed to behave. A secondary, though still important, consideration is that they operate efficiently, both in terms of time and memory consumption.

In addition to the RA-stack precedures previously listed, we add one more, `display-ra-stack`, which displays an RA-stack from bottom to top. We can easily

Strings and Characters

Up until this chapter, we only used strings as output arguments in procedures like `display` and `error`. However, the procedure `tokenize` needs to access the contents of a string and construct a list of tokens from it. Therefore, we need to know more about the built-in *String* data type and the operations it supports. We give here a brief overview of strings and the related data type *Character*; much more information is given in the R^{4}RS Scheme standard, which is available from the web site for this book.

Characters are basic Scheme objects that represent textual characters, such as letters and digits. They are denoted in Scheme by preceding them with `#\`, so `#\a` denotes the character *a*. Certain characters have names; for example the "space" character is written `#\space`. The following procedure determines whether `char` is an arithmetic operator:

```
(define operator-char?
  (lambda (char)
    (member char '(#\+ #\- #\* #\/))))
```

Although strings and vectors are distinct types, strings are essentially vectors that contain characters. Most vector procedures (e.g., `make-vector`, `vector-length`, `vector-ref`, and `vector-set!`) have string equivalents (`make-string`, `string-length`, `string-ref`, and `string-set!`). Also, there are some useful conversion procedures such as `string->number`, which takes a numeric string and converts it to a number it represents, and `string->symbol`, which converts a string to the corresponding symbol.

Given this brief overview of strings and characters, we now present the procedure `tokenize`. By way of explanation, the internal procedure `iter` accumulates the list of tokens from `input-string` in reverse order in the parameter `acc-list`. When `iter` completes, it returns this reverse-order list of tokens. We cons a `$` on the front and reverse the result; therefore, the result is the tokens in correct order and with `$` at the end, as was our desire.

The procedure `iter` processes `input-string` character by character, keeping track of the current position with the parameter `i`, and the "previous state" with the parameter `prev-state`. This state variable tells what type of character we just read, and it is used if we need to process a group of characters together (such as a numeric substring) or are moving to a new token (as would be indicated by a having read a space).

(Continued)

Strings and Characters (Continued)

```
(define tokenize
  (lambda (input-string)
    (define iter
      (lambda (i prev-state acc-lst)
        (if (= i (string-length input-string))
            acc-lst
            (let ((next-char (string-ref input-string i)))
              (cond ((equal? next-char #\space)
                     (iter (+ i 1) 'read-space
                           acc-lst))
                    ((char-numeric? next-char) ;next-char is a digit
                     (if (equal? prev-state 'read-numeric)
                         ;; continue constructing the number, digit
                         ;; by digit, by adding the current digit
                         ;; to 10 times the amount read so far
                         (iter (+ i 1) 'read-numeric
                               (cons (+ (* 10 (car acc-lst))
                                        (digit->number next-char))
                                     (cdr acc-lst)))
                         (iter (+ i 1) 'read-numeric
                               (cons (digit->number next-char)
                                     acc-lst))))
                    ((operator-char? next-char)
                     (iter (+ i 1) 'read-operator
                           (cons (string->symbol
                                  (make-string 1 next-char))
                                 acc-lst)))
                    ((equal? next-char #\()
                     (iter (+ i 1) 'read-lparen
                           (cons 'lparen
                                 acc-lst)))
                    ((equal? next-char #\))
                     (iter (+ i 1) 'read-rparen
                           (cons 'rparen
                                 acc-lst)))
                    (else
                     (error "illegal character in input"
                            next-char)))))))
    (reverse (cons '$ (iter 0 'start '())))))

(define digit->number
  (lambda (digit-char)
    (string->number (string digit-char))))
```

implement it in terms of the other operators:

```
(define display-ra-stack
  (lambda (ra-stack)
    (define display-from
      (lambda (offset)
        (cond ((= offset 0)
               (display (top-minus ra-stack 0))
               'done)
              (else
               (display (top-minus ra-stack offset))
               (display " ")
               (display-from (- offset 1))))))
    (if (empty-ra-stack? ra-stack)
        (display "empty-stack")
        (display-from (- (height ra-stack) 1)))))
```

One advantage of writing `display-ra-stack` in terms of the other operators is that we can then use it to help determine whether the other operators are correctly implemented.

How do we ensure that RA-stacks behave as they should? We must first clearly specify how they are *supposed* to behave. Our description of RA-stacks has so far been very informal, relying on some mental image of a stack, say, as a stack of cafeteria trays, and our ADT operations were supposed to conform to this imagined stack. We can make the specification of RA-stacks more formal by writing equations that specify how the RA-stack operations should work together, much as we did in Section 6.3 for the game-state ADT. For example, here are some equations that describe how `push!` and `pop!` work together with `top-minus`:

$$\texttt{(top-minus (push! } \textit{ra-stack item}\texttt{) 0)} = \textit{item}$$

If $1 \leq i \leq$ `(height` *ra-stack*`)` and $k = i - 1$,

$$\texttt{(top-minus (push! } \textit{ra-stack item}\texttt{) } i\texttt{)} = \texttt{(top-minus } \textit{ra-stack } k\texttt{)}$$

If $0 \leq i <$ `(height` *ra-stack*`)` $- 1$ and $k = i + 1$,

$$\texttt{(top-minus (pop! } \textit{ra-stack}\texttt{) } i\texttt{)} = \texttt{(top-minus } \textit{ra-stack } k\texttt{)}$$

Exercise 13.6

Ideally we should give a set of equations that, taken together, fully specifies RA-stacks; such a complete set would be called an *axiomatic system* for RA-stacks. Rather than getting into whether we have such a complete set (or, in fact, precisely what "complete" means), let's instead generate some additional equations for RA-stacks. Keep in mind that an equation needn't be between two numerical quantities; it can also state that two boolean values are equal.

a. Write equations that explain how `push!` and `pop!` work together with `height`.

b. Write an equation that explains how `empty-ra-stack?` and `height` are related.

c. Write an equation that explains how `empty-ra-stack?` and `make-ra-stack` are related.

Exercise 13.7

The two sides of each of the preceding equations are equivalent in the sense that they produce the same value but not in the sense of also having the same effect. We can make improved versions where the effects as well as the values are identical; for example, if $0 \le i <$ `(height` *ra-stack*`)` $- 1$ and $k = i + 1$,

```
(top-minus (pop! ra-stack) i)
   ≡
(let ((value (top-minus ra-stack k)))
  (pop! ra-stack)
  value)
```

a. Rewrite the other two given equations in this style.

b. Rewrite your equations from Exercise 13.6a in this form.

The previous equations will help guide our implementation. But before we get around to actually writing code, we must first consider how RA-stacks will be represented. By this we mean how a given RA-stack should look in terms of more basic Scheme data objects. In order to come up with a representation, let's first consider what specific needs RA-stacks require from their representation. First and foremost is the need for mutability; and because we only know how to mutate vectors, we will therefore represent an RA-stack with one or more vectors. A secondary consideration is that because we do this mutation at the top of the stack, it would be nice to be able to do so without having to change things elsewhere. Finally, we want to be able to access all elements of the stack efficiently.

Our first representation uses two vectors, one with two cells and the other with a large (though fixed) number of cells. The idea is to use the second vector to store the elements of the stack, starting with the bottom element, and let the first vector maintain the height of the stack as well as a reference to the second vector. Figure 13.5 gives a pictorial representation of the stack $\underline{5\ 2\ 9\ 1}$, where 1 is top

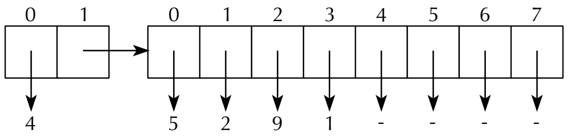


Figure 13.5 Representation of the stack 5 2 9 1, where 1 is top element

element. In this picture, the second vector has eight cells, with the values in the last four cells being immaterial for the stack in question.

The advantage of this representation is that the RA-stack operations are easy to implement, because they involve straightforward vector index computations. For example, the index of the position where the next element should be added is precisely the stack's height, so pushing an element onto the stack involves placing it there and then incrementing the stack's height by 1. Popping an element is accomplished in a similar manner. Accessing an element in a stack is done through a fairly simple index calculation.

Note that this representation imposes an upper limit on the size of the stack, namely, the number of cells in the second vector. We can reflect this restriction by having the following alternative constructor:

```
(make-ra-stack-with-at-most max-num)
  ;; returns an empty stack that can't grow beyond max-num items
```

We can then implement `make-ra-stack` as follows:

```
(define make-ra-stack
  (lambda ()
    (make-ra-stack-with-at-most 8)))
```

The maximum stack size of 8 was somewhat arbitrarily chosen. It is sufficient for most expressions you are likely to encounter when using stacks in the algorithm from the previous section. However, it is insufficient in general because an expression can have arbitrarily many subexpressions, as illustrated by the following example:

```
(1+(2+(3+(4+(5+(6+(7+(8+9))))))))
```

Exercise 13.8

Let's consider the potential size of the expression stack during the course of processing an expression.

a. What is the maximum size of the expression stack during the processing of the preceding expression?

b. What is the maximum size of the expression stack during the processing of an unparenthesized expression?

Let's now work through this implementation scheme. The constructor `make-ra-stack-with-at-most` is straightforward, given the representation described in Figure 13.5. We first create two vectors, called `header` and `cells`, then appropriately initialize the values in `header`, and finally return `header` as the desired empty RA-stack.

```
(define make-ra-stack-with-at-most
  (lambda (max-height)
    (let ((header (make-vector 2))
          (cells (make-vector max-height)))
      (vector-set! header 0 0)     ; header[0] = height = 0
      (vector-set! header 1 cells) ; header[1] = cells
      header)))
```

Note that we used the notation `header[0]` to signify the element in position 0 of the vector `header`. This is *not* allowable Scheme syntax; it is simply an abbreviation we will use in comments and elsewhere when describing the contents of a vector.

Given this construction, the two procedures `height` and `empty-ra-stack?` are also straightforward:

```
(define height
  (lambda (ra-stack)
    (vector-ref ra-stack 0)))

(define empty-ra-stack?
  (lambda (ra-stack)
    (= 0 (height ra-stack))))
```

Note that we've defined `empty-ra-stack?` using `height` rather than directly in terms of `vector-ref`. In general, it makes the implementation of a mutable data type easier to write, read, understand, and modify if arbitrary numerical vector positions needed for `vector-ref` and `vector-set!` are confined to a limited number of procedures. For this reason, we'll also define an "unadvertised" selector, `cells`, which is intended to be used only internally within the implementation of RA-stacks:

```
(define cells  ; use only within the ADT implementation
  (lambda (ra-stack)
    (vector-ref ra-stack 1)))
```

The other operators are more complicated, because we need to do some index computations. For example, consider the operator `top-minus`, which is supposed to return the element *offset* positions from the top of *ra-stack*. How do we calculate the index of the desired element? Well, we claimed in the foregoing that the index of the position where the next element should be added is precisely the stack's height. If we could count on this, we could then conclude that the top of the stack would be in position $height(ra\text{-}stack) - 1$. Therefore, the element *offset* positions from the top would be in position

$$height(ra\text{-}stack) - 1 - offset = height(ra\text{-}stack) - (offset + 1)$$

This information helps us come up with the following implementation of `top-minus`, which includes some error-checking:

```
(define top-minus
  (lambda (ra-stack offset)
    (cond ((< offset 0)
           (error "TOP-MINUS: offset < 0" offset))
          ((>= offset (height ra-stack))
           (error "TOP-MINUS: offset too large for stack"
                  offset (height ra-stack)))
          (else
           (vector-ref (cells ra-stack)
                       (- (height ra-stack)
                          (+ offset 1)))))))
```

The foregoing reasoning relied on certain assumptions about the representation we are using for RA-stacks, namely, that the index of the position where the next element should be added is the height, which is stored in $ra\text{-}stack[0]$, and that the stack elements are stored in order from bottom to top starting at $cells[0]$, where $cells = ra\text{-}stack[1]$. How can we rely on these assumptions? The answer is that *we* must maintain them as *representation invariants*; representation invariants are important enough that we give the following definition:

Representation invariant: A *representation invariant* is a property of the representation of an ADT that is valid for all legally formed and maintained instances of the ADT. In other words, if an instance of the ADT was legally formed via one of the ADT's constructors, and was only altered by legal calls to its mutators, the property is guaranteed to be valid.

By *legal*, we mean that the arguments to the constructors or mutators satisfy all of the stipulated or implied preconditions. For example, the *max-height* argument in `make-ra-stack-with-at-most` must be nonnegative.

What are the representation invariants for RA-stacks? Here is one that describes more formally the structure we are relying on from our representation:

> **RA-stack representation invariant (representation 1):** Let $height = ra\text{-}stack[0]$ and $cells = ra\text{-}stack[1]$. The elements of *ra-stack*, listed from the bottom of the stack to its top, are in $cells[0], cells[1], \ldots, cells[height - 1]$.

In particular, this invariant implies that the element of the stack that is *offset* elements from the top is stored in $cells[height - (offset + 1)]$, the fact we used in our implementation of `top-minus`.

The key point in the definition is that *we* must ensure through our implementation that the representation invariant is valid for any legally formed and maintained instance of an RA-stack. How can we do this? Well, note that any such instance was first formed by an ADT constructor and then was operated on a finite number of times by certain of the ADT selectors and mutators. Because the selectors do not change the instance, the only changes come from the finite sequence of mutations. We can inductively prove the validity of the invariant if we show that

- The invariant is valid for the value returned by a legal call to an RA-stack constructor.
- If the property is valid for an RA-stack before it is passed in a legal call to an RA-stack mutator, it is also valid after the call.

The first condition corresponds to the base case of an induction proof, whereas the second condition corresponds to the inductive step.

Consider first the base case. Note that the invariant is true for the return value for the RA-stack constructor `make-ra-stack-with-at-most` (and therefore also for `make-ra-stack`): After all, there is no i such that $0 \leq i < height$ because $height = 0$. We say that the invariant is *vacuously* true in this case.

How about the inductive step in the proof of the invariant? Clearly we can't prove it yet because we have not yet written the two mutators `pop!` and `push!`. On the other hand, we can use our need to prove the inductive step to guide our implementation of the two mutators. Take for example `pop!`. The only thing we need to do in order to remove the top element of the stack while maintaining the invariant is to decrease $ra\text{-}stack[0]$ (the height) by 1. After all, the remaining elements of the stack will still be in the required order and will still start at location 0 in the *cells* vector, so the invariant will remain valid assuming it had been valid when `pop!` was called. Therefore, we deduce the following implementation for `pop!`:

```
(define pop!
  (lambda (ra-stack)
    (if (empty-ra-stack? ra-stack)
        (error "POP!: attempted pop from an empty stack")
        (begin
          (set-height! ra-stack
                       (- (height ra-stack) 1))
          ra-stack))))

(define set-height!  ; use only within the ADT implementation
  (lambda (ra-stack new-height)
    (vector-set! ra-stack 0 new-height)))
```

Finally, consider `push!`. Again, the invariant will remain valid if we put the new item in the position with index *height(ra-stack)*. (Recall that the existing elements stop in the location before that one.) After doing the appropriate `vector-set!` to put it there, all we need to do is increase the value of of the height by 1. Hence:

```
(define push!
  (lambda (ra-stack item)
    (if (<= (vector-length (cells ra-stack))
            (height ra-stack))
        (error "PUSH!: attempted push onto a full stack")
        (begin
          (vector-set! (cells ra-stack)
                       (height ra-stack)
                       item)
          (set-height! ra-stack
                       (+ (height ra-stack) 1))
          ra-stack))))
```

That completes our first implementation of RA-stacks. The main advantage of this implementation is its efficiency: Each operator uses only a small, fixed number of operations. However, there is a definite disadvantage: The stack has a limited size.

Exercise 13.9

One way to overcome this size limitation is to increase the size of the vector holding the stack elements whenever that is necessary, which means rewriting the `error` clause of the `if` expression in `push!`. For example, you could create a vector of twice the size of the current `cells` vector, copy the old stack elements into the

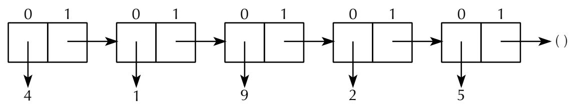


Figure 13.6 Representation of the stack 5 2 9 1, where 1 is top element

new vector, set the new vector as the stack's `cells` vector, and carry on from there. Rewrite `push!` to implement this strategy.

Our second representation of stacks (not counting the one in Exercise 13.9) uses a varying number of two-element vectors. It contains one vector for each element in the stack, plus an additional vector (the *header*) that contains the stack's height and a pointer to the first of the other vectors. Each of the other vectors contains a stack element and a reference to the next vector. In effect, we are implementing something very similar to Scheme lists. Figure 13.6 gives a pictorial representation of the stack 5 2 9 1, where 1 is top element. Notice that the stack is listed from top to bottom, which is the opposite of the first representation. We do this to have easy access to the top of the stack: otherwise we would have to, in effect, "cdr" to the end of the stack in order to add or delete elements. You'll notice in Figure 13.6 that the last two-element vector has the empty list, `()`, in position 1, which plays the same role as in normal Scheme lists. Because the stack's height is explicitly recorded, this end-marker isn't strictly necessary, but it does make debugging and reasoning easier.

Before starting this implementation, we should try to come up with an invariant that describes our representation. But even before working on the invariant, we have a higher priority: coming up with some terminology so that we can conveniently talk about our representation. We will call the two-element vectors *nodes*, and a *linked list* of nodes such as in Figure 13.6 a *node-list*.

Rather than continuing to talk concretely about the nodes as two-element vectors with "element 0" and "element 1," it would be better if we treated nodes as an abstract data type with the two selectors `node-element` and `node-rest`. That way you don't need to keep straight the 0s and 1s, and we also have the flexibility to later switch to a nonvector representation. For now, the implementation of nodes is as follows:

```
(define make-node
  (lambda (element rest)
    (let ((node (make-vector 2)))
      (vector-set! node 0 element)
      (vector-set! node 1 rest)
      node)))
```

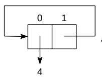
```
(define node-element
  (lambda (node)
    (vector-ref node 0)))

(define node-rest
  (lambda (node)
    (vector-ref node 1)))
```

When we say that an object is a linked list of nodes, or a node-list, we mean that it obeys the following representation invariant:

Node-list representation invariant: A node-list is always represented in one of two ways:

1. As the empty list, `()`, in which case we say the list is of length 0, or contains 0 nodes.
2. As a node that has as its `node-rest` component a node-list of length $n - 1$, where n is a positive integer; in this case we say that the original node represents a node-list of length n, or contains n nodes.

All node-lists must be assigned a unique well-defined length by the above rules; this forbids cycles such as 

.

Because our new representation of RA-stacks is as node-lists, we'll be able to take advantage of the preceding invariant for node-lists but will also have the responsibility for maintaining that invariant. However, not just any node-list is a valid representation of an RA-stack, so there is an additional representation invariant specific to RA-stacks in addition to the generic node-list invariant above:

RA-stack representation invariant (representation 2): Let *height* be the `node-element` component of *ra-stack*. Then *ra-stack* is a node-list containing *height*+1 nodes. Furthermore, the elements of the RA-stack, listed from top to bottom, are the `node-element` components of the nodes in the node-list given by the `node-rest` component of *ra-stack* (that is, the node-list starting with the second node in *ra-stack*).

This invariant already indicates to us how we should implement the operators `make-ra-stack` and `height`. (Note that we no longer have any reason to implement `make-ra-stack-with-at-most`, and `empty-ra-stack?` can remain unchanged, because it is defined in terms of `height`.)

```
(define make-ra-stack
  (lambda ()
    (make-node 0 '()))) ; height 0, no other nodes

(define height
  (lambda (ra-stack)
    (node-element ra-stack)))
```

Given their similarity, it would be very useful if we could mimic some of the functionality of Scheme lists in our node-lists. One such list-like procedure we will use later is `nodes-down`, which is roughly like cdring *n* times down a node-list. Thus, if `ra-stack` is the node-list in Figure 13.6, `(nodes-down 0 ra-stack)` would be `ra-stack` itself, whereas `(nodes-down 2 ra-stack)` would be the node-list starting with the node containing 9.

```
(define nodes-down
  (lambda (n node-list)
    (if (= n 0)
        node-list
        (nodes-down (- n 1) (node-rest node-list)))))
```

This procedure makes `top-minus` quite easy to write, given the invariant describing our current representation:

```
(define top-minus
  (lambda (ra-stack offset)
    (cond ((< offset 0)
           (error "TOP-MINUS: offset < 0" offset))
          ((>= offset (height ra-stack))
           (error "TOP-MINUS: offset too large for stack"
                  offset (height ra-stack)))
          (else
           (node-element (nodes-down (+ offset 1) ra-stack))))))
```

To maintain the invariant in `pop!`, we need to somehow remove the second node in the node-list (because that is where the top element of the stack is contained) and also decrease the stack's height by 1. Both of these tasks involve updating a node, so we'll need the following two mutator procedures for our abstract data type of nodes:

```
(define node-set-element!
  (lambda (node new-element)
    (vector-set! node 0 new-element)))
```

```
(define node-set-rest!
  (lambda (node new-rest)
    (vector-set! node 1 new-rest)))
```

Given a node that represents an RA-stack, the `node-element` component is the height of the stack, so decreasing the height by 1 will be done using `node-set-element!`:

```
(node-set-element! ra-stack (- (height ra-stack) 1))
```

Similarly, the RA-stack's `node-rest` component is what needs updating to reflect the removal of the node containing the top stack element; it should now have as its contents a node-list of all the elements on the stack after the `pop!` (i.e., all except the one that was on top). This node-list can be found using `nodes-down` to skip over the header node and the node containing the top element:

```
(node-set-rest! ra-stack (nodes-down ra-stack 2))
```

Putting these two steps together (with a little error-checking), we get the following:

```
(define pop!
  (lambda (ra-stack)
    (if (empty-ra-stack? ra-stack)
        (error "POP!: attempted pop from an empty stack")
        (begin (node-set-element! ra-stack (- (height ra-stack) 1))
               (node-set-rest! ra-stack (nodes-down 2 ra-stack))
               ra-stack))))
```

Finally, `push!` requires us to first insert a new node containing the new element between the first two nodes of the old stack and then to increase the height by 1. Figure 13.7 illustrates how this would work when pushing 6 onto the stack in Figure 13.6.

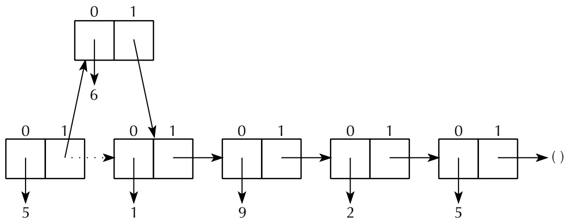


Figure 13.7 Effect of pushing 6 onto the stack from the previous figure

```
(define push!
  (lambda (ra-stack item)
    (let ((new-node (make-node item (node-rest ra-stack))))
      (node-set-rest! ra-stack new-node)
      (node-set-element! ra-stack (+ (height ra-stack) 1))
      ra-stack)))
```

This code completes our second implementation of RA-stacks. It has the advantage of imposing no growth restrictions on RA-stacks. Furthermore, with the exception of `top-minus`, all of the operators are efficient in that they only require a small, fixed number of operations. On the other hand, the procedure `top-minus` has linear complexity, measured in terms of `offset`. In the application from the previous section, this is unimportant, because the largest value of `offset` we used was 2.

Before we leave the linked-list representation entirely, we can make one other interesting observation. The node-lists we have been using are extremely similar to normal Scheme lists; wouldn't it be nice if they really could be lists? That is, we would like to use pairs (of the kind `cons` creates) rather than two-element vectors as the representation of the abstract data type of nodes. The constructor and selectors are no problem—`cons`, `car`, and `cdr` correspond naturally to `make-node`, `node-element`, and `node-rest`. The only problem is with the mutators. But, Scheme has mutators for pairs too—a secret we've been hiding thus far. They are called `set-car!` and `set-cdr!`, and they allow us to reimplement nodes as follows:

```
(define make-node cons)
(define node-element car)
(define node-rest cdr)
(define node-set-element! set-car!)
(define node-set-rest! set-cdr!)
```

With these definitions in place, the RA-stack procedures will work as before, except now the node-lists will be ordinary lists made out of `cons` pairs. The pictures would lose the "0" and "1" labels over the boxes, which were our way of distinguishing two-element vectors from pairs.

13.4 Queues

Stacks have the property that the last item pushed onto the stack is the first one popped off; for this reason, they are also known as *LIFO* structures, for *last in first out*. Sometimes we'd rather store information in a *first in first out*, or *FIFO* fashion. This typically arises from fairness considerations. For example, imagine storing the names of the students waiting to get into a popular course. If a space opens up, we'd like to retrieve the name of the student who has been waiting the longest.

The traditional name for a data structure that works in this way is a *queue* (which is pronounced like the letter Q). In this section we'll look at queues as another example of how representation invariants can guide us in implementing a mutable data type. As with RA-stacks, we'll look at two different styles of representation. In one, we store the elements in consecutive positions within a vector. In the other, we store each element in a separate node, with the nodes linked together into a list.

We'll start by giving a list of operations for the queue ADT:

```
(make-queue)
  ;; returns a newly created empty queue.

(empty-queue? queue)
  ;; returns #t if queue is empty, otherwise #f.

(head queue)
  ;; returns the element which is at the head of queue,
  ;; that is, the element that has been waiting the longest,
  ;; provided queue is nonempty.

(dequeue! queue)
  ;; removes the head of queue, provided queue is
  ;; nonempty. The return value is the modified queue.

(enqueue! queue item)
  ;; inserts item at the tail of queue, that is, as the most
  ;; recent arrival. The return value is the modified queue.
```

The two mutators are pronounced like the letters DQ and NQ.

Now consider representing queues like our first representation of RA-stacks. In that representation, we stored the items in consecutive positions of a "cells" vector and used a two-element "header" vector to store the number of items in the RA-stack and the cells vector. If we used this same format for queues, and also maintained the representation invariant that the head of the queue is in cell number 0 and the remaining elements follow in consecutive cells, we might wind up with a picture like Figure 13.8 for a queue that had 5 enqueued, then 2, then 9, and finally 1.

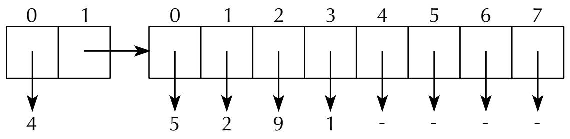


Figure 13.8 Initial, suboptimal, idea for how to represent the queue 5 2 9 1, where 5 is the oldest element (head) and 1 is the newest (tail)

In this representation for queues, all the operations except `dequeue!` would be relatively straightforward. However, because `dequeue!` is supposed to remove the 5 from the head of the queue, in this representation it would be necessary to shift the remaining elements all down by one position. For this reason, the representation isn't a good one. The basic problem is that maintaining the representation invariant is too expensive, given that the elements of the queue should start at position 0 in the cells vector.

One way to cope with an expensive-to-maintain representation invariant is to redesign the representation to be more flexible so that we don't have as constraining of an invariant to maintain. In particular, we'd like to have the flexibility to start the queue at any point in the cells vector rather than always at position 0. That way when a `dequeue!` operation is done, we wouldn't have to shift the remaining elements down. In order to support this flexibility, we'll extend the header vector to now contain three pieces of information. It will still contain the number of elements in the queue and the cells vector. However, it will also contain the position number within the cells vector that the queue's head is at. For example, we could now `dequeue!` the element 5 from the queue 5 2 9 1 as shown in Figure 13.9, changing from having four elements starting in position 0 to having three elements starting in position 1.

Suppose, having dequeued 5 from our example queue, we now were to enqueue some additional elements. Because the cells vector in the figure has four unused cells after the one containing 1, we could insert four more items without any problem. What about adding a fifth item, bringing the total length of the queue to eight? It should be possible to store an eight-element queue in an eight-element cells vector. The trick is to consider the queue's storage as "wrapping around" to the beginning of the vector. Because the queue starts in position 1 within the cells vector, it can continue to positions 2, 3, 4, 5, 6, 7, and then 0, in that order. Similarly, if we dequeued the 2, we would then have freed up space to enqueue one more item, and the queue would now go from position 2 to 3, 4, 5, 6, 7, 0, and 1. This wrapping around of positions can be expressed using modular arithmetic. We can write the representation invariant as follows:

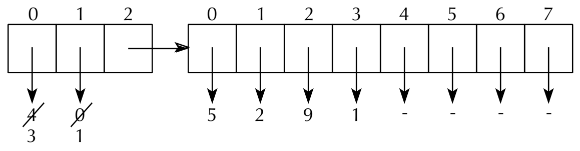


Figure 13.9 Improved idea for how to represent the queue 5 2 9 1, where 5 is the oldest element (head) and 1 is the newest (tail); the indicated changes correspond to using `dequeue!` to remove the 5, changing the queue to 2 9 1

Queue representation invariant (representation 1): Let *queuelength* = *queue*[0], *start* = *queue*[1], and *cells* = *queue*[2]. Let *cellslength* = `(vector-length` *cells*`)`. The following restrictions are all met:

- $0 \leq queuelength \leq cellslength$
- $0 \leq start < cellslength$
- There are *queuelength* elements in *queue*. For each *i* in the range $0 \leq i < queuelength$, the element that is *i* elements after the head of *queue* is stored in $cells[(start + i) \bmod cellslength]$.

We can use this representation invariant to guide us in writing the operations as follows:

```
(define queue-length ; use only within the ADT implementation
  (lambda (queue)
    (vector-ref queue 0)))

(define set-queue-length! ; use only within the ADT implementation
  (lambda (queue new-length)
    (vector-set! queue 0 new-length)))

(define queue-start ;use only within the ADT implementation
  (lambda (queue)
    (vector-ref queue 1)))

(define set-queue-start! ; use only within the ADT implementation
  (lambda (queue new-start)
    (vector-set! queue 1 new-start)))

(define queue-cells ; use only within the ADT implementation
  (lambda (queue)
    (vector-ref queue 2)))

(define set-queue-cells! ; use only within the ADT implementation
  (lambda (queue new-cells)
    (vector-set! queue 2 new-cells)))
```

```
(define make-queue
  (lambda ()
    (let ((cells (make-vector 8)) ; 8 is arbitrary
          (header (make-vector 3)))
      (set-queue-length! header 0)
      (set-queue-start! header 0) ; arbitrary start
      (set-queue-cells! header cells)
      header)))
```

```
(define empty-queue?
  (lambda (queue)
    (= (queue-length queue) 0)))
```

```
(define head
  (lambda (queue)
    (if (empty-queue? queue)
        (error "attempt to take head of an empty queue")
        (vector-ref (queue-cells queue)
                    (queue-start queue)))))
```

```
(define enqueue!
  (lambda (queue new-item)
    (let ((length (queue-length queue))
          (start (queue-start queue))
          (cells (queue-cells queue)))
      (if (= length (vector-length cells))
          (begin
            (enlarge-queue! queue)
            (enqueue! queue new-item))
          (begin
            (vector-set! cells
                         (remainder (+ start length)
                                    (vector-length cells))
                         new-item)
            (set-queue-length! queue (+ length 1))
            queue)))))
```

```
(define enlarge-queue! ;use only within the ADT implementation
  (lambda (queue)
    (let ((length (queue-length queue))
          (start (queue-start queue))
          (cells (queue-cells queue)))
      (let ((cells-length (vector-length cells)))
        (let ((new-cells (make-vector (* 2 cells-length))))
          (from-to-do
           0 (- length 1)
           (lambda (i)
             (vector-set! new-cells i
                          (vector-ref cells
                                      (remainder (+ start i)
                                                 cells-length)))))
          (set-queue-start! queue 0)
          (set-queue-cells! queue new-cells)
          queue)))))
```

Exercise 13.10

The `enlarge-queue!` procedure is used when the cells vector is full. It makes a new cells vector twice as large and copies the queue's elements into it. It copies the elements into positions starting at the beginning of the new cells vector and correspondingly sets the queue's start to be 0. Explain why the queue's elements can't just be copied into the same positions within the new vector that they occupied in the old vector.

Exercise 13.11

We've left out `dequeue!`. Write it. If the queue is empty, you should signal an error. Be sure to maintain the representation invariant by adjusting the start of the queue appropriately. You can't just add 1 to it because you have to keep it in the proper range, $0 \leq \mathit{start} < \mathit{cellslength}$.

Now let's turn our attention to designing an alternative queue representation using a node list. We'll store each element of the queue in one node of the node list, in some order; we still have to decide whether it should be head to tail or tail to head. Recall that node lists are inherently asymmetrical: One end of the node list is the beginning, from which one can start cdring down the list. Queues need to be operated on at both ends because enqueuing happens at the tail end, and dequeuing happens at the head end. Thus, to support both operations efficiently, we'll need

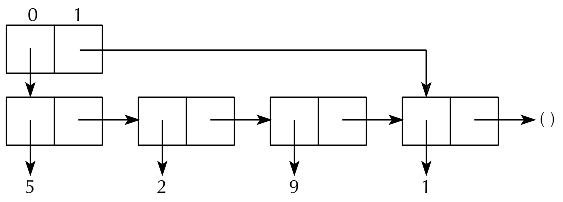


Figure 13.10 Representation of the queue 5 2 9 1 as a header vector that contains the first and last nodes of a node list

some quick way to get directly to the last node in the node list, without cdring down to it starting from the first node. This is true no matter which order we pick; the order just determines which operation's efficiency is at stake. The easiest way to have quick access to both ends of the node list is by using a header vector that directly contains both the first and the last node of the node list. That is, we would have a situation like that shown in Figure 13.10.

If you consider what it would take to maintain the representation invariant, you can figure out which end of the node list should be the queue's head and which should be the queue's tail. Remember that nodes will get added at the queue's tail and removed at its head. So, we have to consider how easily we can update the picture shown in Figure 13.10 under the two options:

- If the beginning of the node list is the head of the queue, we can dequeue by simply adjusting the header vector to point at the next node instead. We can enqueue by adding a new node after the current last node (found using the header vector) and adjusting the header vector to reflect this new last node.
- If the beginning of the node list is the tail of the queue, enqueuing would still be easy because we can tack a new node onto the front of the node list and adjust the header vector to make it the new starting node. However, dequeuing is another matter. There is no efficient way to get to the second to last node, which should now become the last node.

Having considered these options, we see that it is superior to consider the start of the node list the head of the queue. That is, in Figure 13.10, 5 is the head element of the queue. Having made this decision, we should formalize it in a representation invariant.

Exercise 13.12

Write the representation invariant for this second representation for queues. Be sure to specify what the two elements of the header vector should be when the queue is empty.

Exercise 13.13

Now write the queue ADT procedures based on this new representation invariant. Be sure that you maintain the invariant. For example, when you enqueue, you will need not only to make use of the header's information about which node is last but also to update that information.

13.5 Binary Search Trees Revisited

One common problem in computer programming is maintaining large amounts of data in a manner that allows the individual records in the data to be retrieved efficiently. For example, states maintain driver's license information, schools maintain student records, dictionaries maintain definitions, card catalogs maintain book records, and Joe Franksen's video store (Section 8.1) maintains its video records. A data structure that holds this information should allow efficient construction, retrieval, and maintenance.

When we considered this problem in Section 8.1, we investigated binary search trees as such a storage mechanism. (Recall that binary search trees are binary trees where each node has a value that is greater than those in its left subtree and less than those in its right subtree.) Binary search trees have the potential for very efficient data retrieval. Specifically, we showed that searching for an element in such a tree takes $O(h)$ time, where h is the height of the tree. We also showed that the minimum height for a binary tree with n nodes is exactly $\lfloor \log_2(n) \rfloor$ (where $\lfloor \log_2(n) \rfloor$ is the largest integer $\leq \log_2(n)$). Thus, searching for an element in a minimum-height tree would take $O(\log(n))$ time. We even wrote a procedure `optimize-bstree` in Exercise 8.13 on page 224 that produced a minimum-height binary search tree containing the same nodes as a given binary search tree.

Unfortunately, the methods developed in Sections 8.1 and 8.2 did not adequately address the problem of maintenance, by which we mean adding and deleting records when necessary. In particular, the `insert` procedure in Exercise 8.6 on page 220 did not keep the height of the tree as small as possible, and calling the `optimize-bstree` procedure after each insertion would prove time-consuming. What should we do? Well, various strategies have been devised for maintaining binary search trees so that their height is $O(\log(n))$, which will suffice to allow us to write retrieval, insertion, and deletion procedures that have time complexity $O(\log(n))$. We describe one such strategy here, one using *red-black trees*.

Red-black trees are a special subclass of binary search trees. That is, they obey an additional, more restrictive, representation invariant above and beyond the structural invariant that all binary trees satisfy and the ordering invariants that all binary search trees satisfy. Every node in a red-black tree has an additional field, its *color*, which is either red or black. This includes also the "empty nodes" at the bottom of the tree, which we treat as the leaves of the tree. The representation invariant is that the following three conditions hold must hold, in addition to the binary search condition:

- Each leaf (empty) node is black.
- The number of black nodes along a path from the root node to any of the leaf nodes is the same as for any of the other leaves.
- No red node has a red parent.

Figure 13.11 gives an example of a red-black tree containing numbers (the only type of red-black trees we will consider in this section).

We need to show that the height of a red-black tree with n nonempty nodes is $O(\log(n))$. Let h denote the height of our tree. When we say that this tree has height h, we mean that the deepest of the empty nodes is at depth h. For example, in Figure 13.11 the deepest empty node is at depth 5, so the tree has height 5. How about the shallowest empty node? The tree in that figure has its shallowest empty node at depth 3; we will use the name d for the depth of the shallowest empty node. Because d is the depth of the shallowest empty node, all the nodes at depth $d-1$ must be nonempty. There are 2^{d-1} of these; thus, the number of nonempty nodes, n, is at least this big, and we have $n \geq 2^{d-1}$. Taking the log of both sides we have $\log_2(n) \geq d-1$, or $\log_2(n)+1 \geq d$, so we know that d can be no bigger than $\log_2(n)+1$. When $n \geq 2$, this means that $d \leq 2\log_2(n)$.

This is all well and good for the shallowest empty node, at depth d, but what about the deepest, at depth h? The red-black properties come to our rescue here: There are an equal number of black nodes on any path down from the root to a leaf, and at worst every other node on such a path can be red, because red nodes cannot

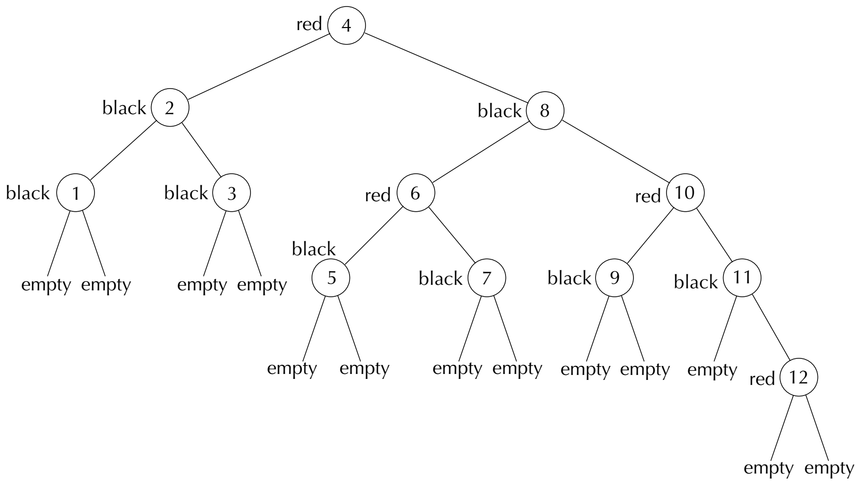


Figure 13.11 A red-black tree containing numbers

have red parents. Thus, the deepest empty node can be at most twice as deep as the shallowest (which would happen if there were no red nodes at all on the path to the shallowest empty node and every other node was red on the path to the deepest empty node). Therefore we have $h \leq 2d$ and hence $h \leq 4\log_2(n)$ for $n \geq 2$. From the foregoing we conclude that h, and therefore also the complexity of retrieval, is $O(\log(n))$ in red-black trees.

We next turn to the insertion algorithm for red-black trees. As a mutator of red-black trees, `red-black-insert!` will take a number and a red-black tree and then insert the number into the tree in a manner that maintains the binary search and red-black invariants. We would naturally want the complexity of `red-black-insert!` to be $O(\log(n))$ as well. But before actually describing the insertion algorithm, we give an equivalent definition of red-black trees that will prove to be better suited for both describing and implementing the algorithm.

According to the new definition, a red-black tree is a binary search tree where every node has an additional field, its *rank*, that is a nonnegative integer. (The rank is in place of the color, not in addition to it.) Again, this definition includes also the "empty nodes" at the bottom of the tree. Furthermore, the following three conditions must hold (in addition to the binary search condition):

- Each leaf (empty) node has rank 0 and each parent of a leaf has rank 1.
- $rank(node) \leq rank(parent(node)) \leq rank(node) + 1$, provided *node* has a parent.
- $rank(node) < rank(parent(parent(node)))$, provided *node* has a grandparent

Briefly, the latter two conditions say that the rank can either stay the same or increase by 1 when going to a node's parent, but it can't stay the same through all three of the node, its parent, and its grandparent. Figure 13.12 gives the example from Figure 13.11 according to this new definition of red-black trees.

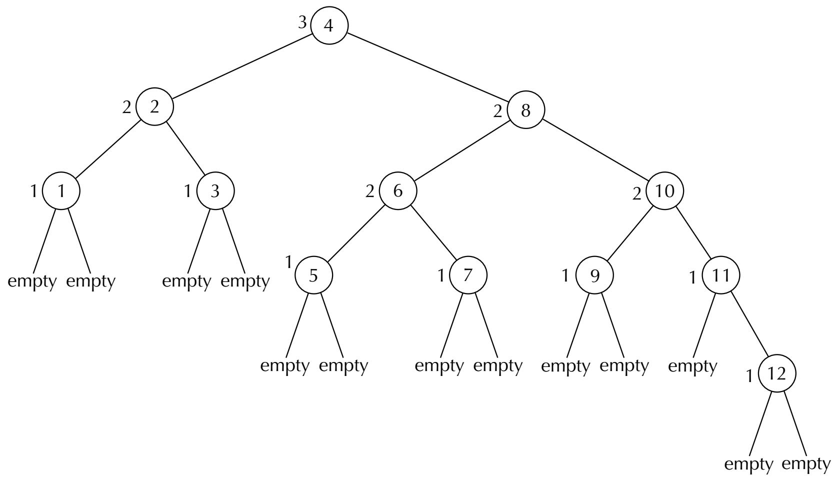


Figure 13.12 Previous red-black tree recast according to new definition

Why are these two definitions of red-black trees equivalent? If you have a red-black tree according to the second definition, color a node black if its rank is different from its parent, and otherwise color it red. (The root node must be black if either of its children are red, but otherwise it can be either red or black.) Because leaves have rank 0 and their parents have rank 1, all leaf nodes are black. Furthermore, the ranks along any path from the root to a leaf will decrease k times, where k is the rank of the root, and each decrease corresponds to a black node; hence, the number of black nodes is the same going to any leaf. Finally, the prohibition against three consecutive generations sharing a rank implies that the parent of a red node is necessarily black.

Exercise 13.14

If you have a red-black tree according to the first definition, you can define the rank of a node to be the number of black nodes encountered along any path from the node down to any descendant leaf (not counting the node itself). Explain why the foregoing results in a red-black tree according to the second definition.

We finally turn to the algorithm for insertion into a red-black tree. The idea is to use the binary search condition to move down the tree until we find a leaf (empty) node where the new item can be inserted while maintaining the binary search condition. We then insert the item, giving it rank 1 and two new rank 0 empty children. Thus far, the insertion is much as for binary search trees. However, the additional red-black invariants may have become violated. Therefore, before calling the red-black tree insertion complete, we repair the damage by performing a sequence of simple "rebalancings" that progress upward until the invariants have been restored, possibly moving as far up the tree as the root node. Just as the number of steps going down the tree was $O(\log(n))$, so too the number of rebalancing steps moving back up the tree is also $O(\log(n))$.

What operations can we do at a given node and how do we determine which one to do in order to rebalance at a given node? The point is that after we have done the binary search insert, only the third of the red-black conditions might fail (the prohibition against three consecutive equal-rank generations), and if it does fail, it will only do so at the newly inserted node. Our strategy will be to move this failure upward in the tree until it finally disappears.

This condition can fail in exactly four ways, each of which is illustrated in Figure 13.13. (The triangles in this diagram correspond to possibly empty subtrees, and the letter k corresponds to a rank). We therefore need rebalancing procedures that will deal with each of these four cases. If we consider the first of these cases, we see that it can be broken down into the two subcases illustrated in Figure 13.14. In the first of these, our only choice is to increase the rank of the grandparent by 1 and continue upward from there. We call the process of increasing a node's rank by one *promotion*.

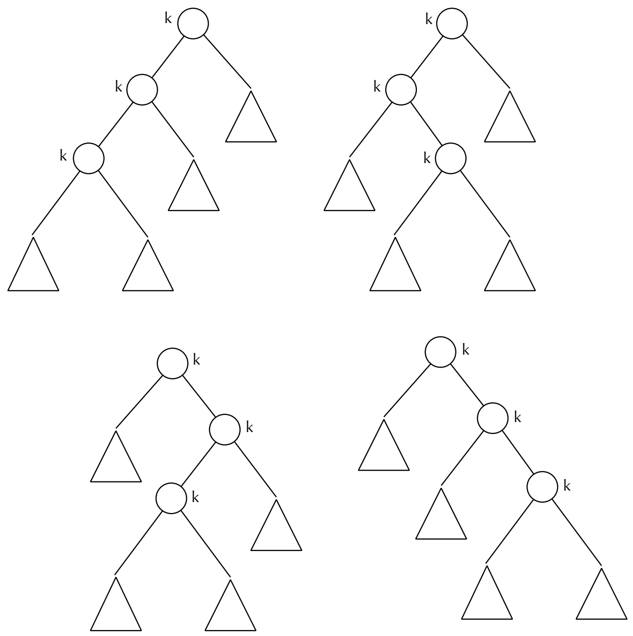


Figure 13.13 Four ways to fail the third red-black condition

The second subcase is more difficult because promoting the grandparent would cause it to have a rank that is 2 larger than the rank of its right child, thereby violating the second red-black condition. We therefore need some operation that will "move things around" slightly. Two such operations, called *right rotation* and *left rotation*, are illustrated in Figure 13.15. Notice that the nodes in the two trees in Figure 13.15 satisfy the following condition (where b and d denote the values at the two displayed nodes, and A, C, and E represent values at any node in their respective subtrees):

$$A < b < C < d < E$$

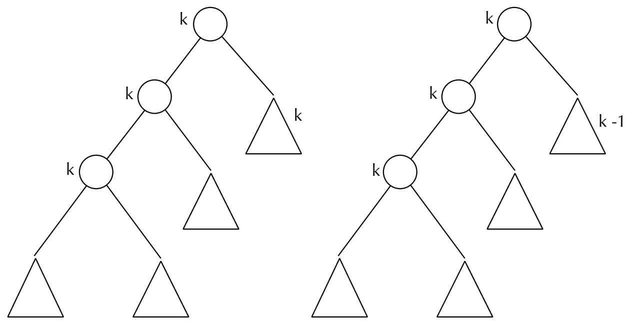


Figure 13.14 Two subcases for failing the third red-black condition

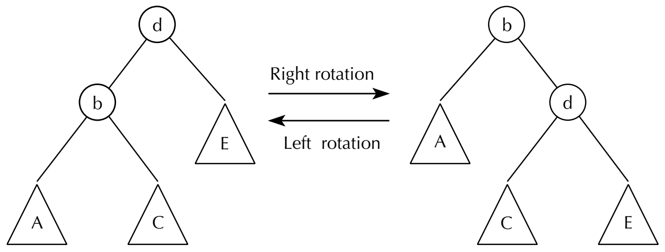


Figure 13.15 Right and left rotation

This condition means that the binary search condition is maintained under both left and right rotation. Figure 13.16 illustrates how right rotation applied to the second tree in Figure 13.14 completely fixes the red-black failure.

What about the other possible red-black failures illustrated in Figure 13.13? Each of these again has a subcase where grandparent promotion applies; we will focus here only on the second subcase of each case, where we can't promote the grandparent. The last case shown in Figure 13.13, which is the mirror image of the first, can be fixed by left rotation. The other cases appear more complicated but can be solved by a couple of rotations. For example, the way to fix the second case is illustrated in Figure 13.17.

As an example of how insertion into a red-black tree works, consider starting with an empty tree and inserting the numbers 1, 2, 3, and 4 in that order. This is illustrated in Figure 13.18. Inserting the 1 puts the value of 1 at the root node, with a rank of 1. Because this node has no grandparent, there is no potential for it to have the same rank as its grandparent, and hence there is no failure of the red-black invariant. Therefore, no rebalancing action is necessary. Now we insert the 2; because 2 is greater than 1, it goes in as the right child of the root, again with rank 1. (Remember, all insertions are at rank 1.) Again, this node has no grandparent, so there can be no problem. Next we insert 3; because it is greater than both 2 and 1, it goes in as the right child of the 2 node, as usual with rank 1. Now we have three

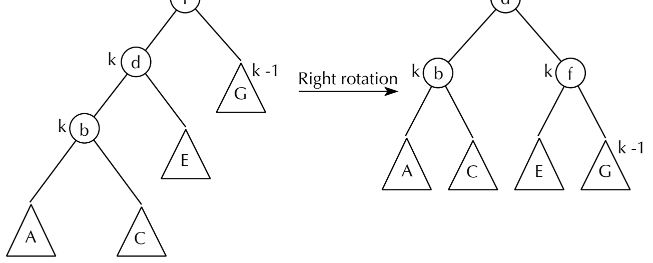


Figure 13.16 Using right rotation to fix case 1, subcase 2

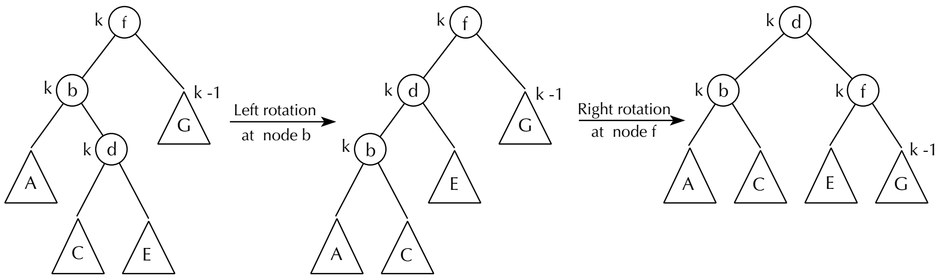


Figure 13.17 Using left and right rotations to fix case 2

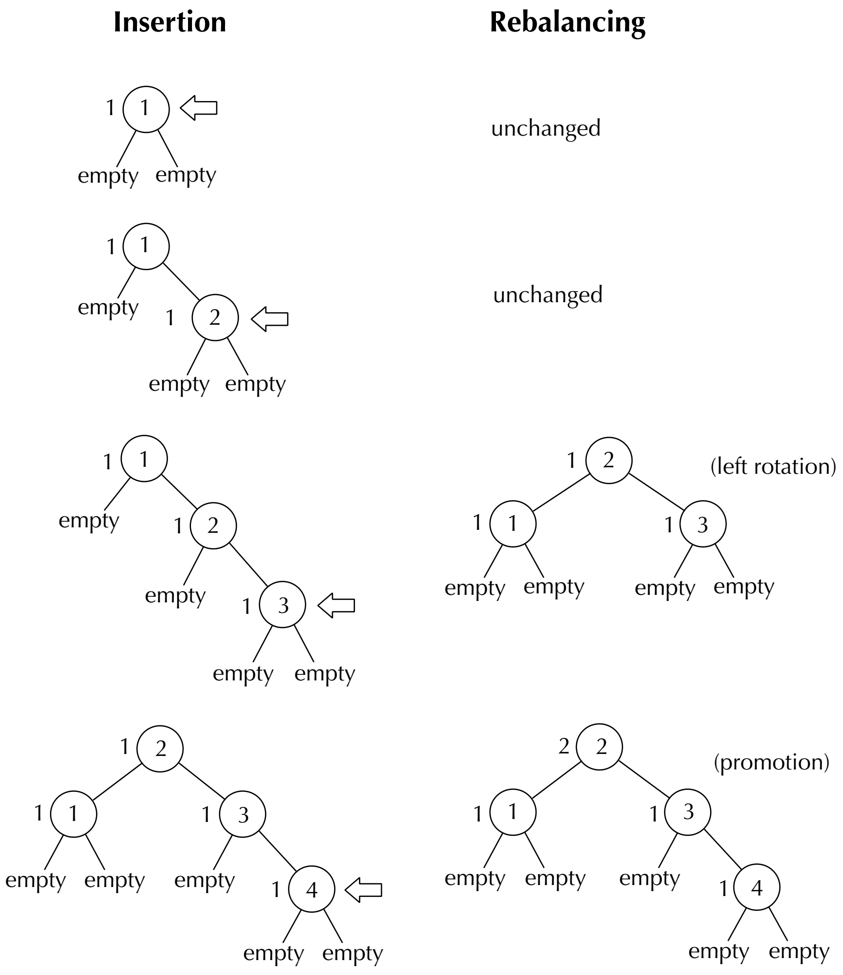


Figure 13.18 Inserting 1, 2, 3, and 4 into an empty red-black tree

nodes in a row all of rank 1: the newly inserted 3, the 2 that is its parent, and the 1 that is its grandparent. This is a violation of the invariant. Because the new node's uncle doesn't share the rank of 1 (it is an empty tree, with rank 0), it isn't permissible to promote the grandparent to rank 2. Instead we do a left rotation at the new node's grandparent, resulting in the node with value 2 now being the root, with the 1 on its left and the 3 on its right. All three of these nodes retain their ranks, (i.e., all are still rank 1). Now the 4 is inserted, and it goes to the right of the 3, with rank 1. Again we have an invariant violation, because three generations in a row share the rank 1: the new 4, its parent 3, and its grandparent 2. However, this time the uncle (the node with value 1) also is of rank 1, so we simply promote the grandparent to rank 2. Because this node is the root of the tree, it doesn't itself have a grandparent, so we can't possibly have introduced a new invariant violation in fixing the old one. Thus, we are done.

We can work through another example, in which the course of events is slightly different. Suppose we again start with an empty tree, but this time insert 12, 1, 4, and 3 in that order, as illustrated in Figure 13.19. The 12 becomes the root and the 1 becomes its left child. Both nodes have rank 1, and no rebalancing is necessary, because neither has a grandparent. Now we insert the 4; because it is smaller than 12 but bigger than 1, it goes to the right of the 1, with rank 1. This action leads to a three-generation chain of rank 1, so we have an invariant violation. Again the uncle is an empty tree of rank 0, so promotion isn't an option. Instead we can first do a left rotation at the parent (the node containing 1) and then a right rotation at the grandparent (the node containing 12). The net result is that the node with value 4 is now the root, with 1 on its left and 12 on its right. All three are still of rank 1. Now when 3 is inserted, it goes to the right of the 1 node, and the resulting invariant violation can be fixed simply by promoting the root node (the new node's grandparent) to rank 2.

We can summarize the rebalancing process as shown in Figure 13.20. Notice that we have two basic kinds of rebalancing, depending on whether the node's uncle shares its rank (which is also shared by the grandparent, or we'd have no problem). If the uncle has the same rank, we promote the grandparent; otherwise we rotate. We mentioned earlier that in the case where we rotate, promotion wouldn't work because it would leave the grandparent with a rank 2 greater than the uncle. In the case where we promote, rotation would just shift the problem from one side of the tree to the other rather than making any headway on solving it. (To see this, consider Figure 13.16, but with G relabeled to be of rank *k*.) Thus, we never really have any choice—one situation needs promotion and the other needs rotation.

Exercise 13.15

Insert the following numbers one by one into the red-black tree shown in Figure 13.12. After each one is inserted, do the appropriate rotation(s) and/or promo-

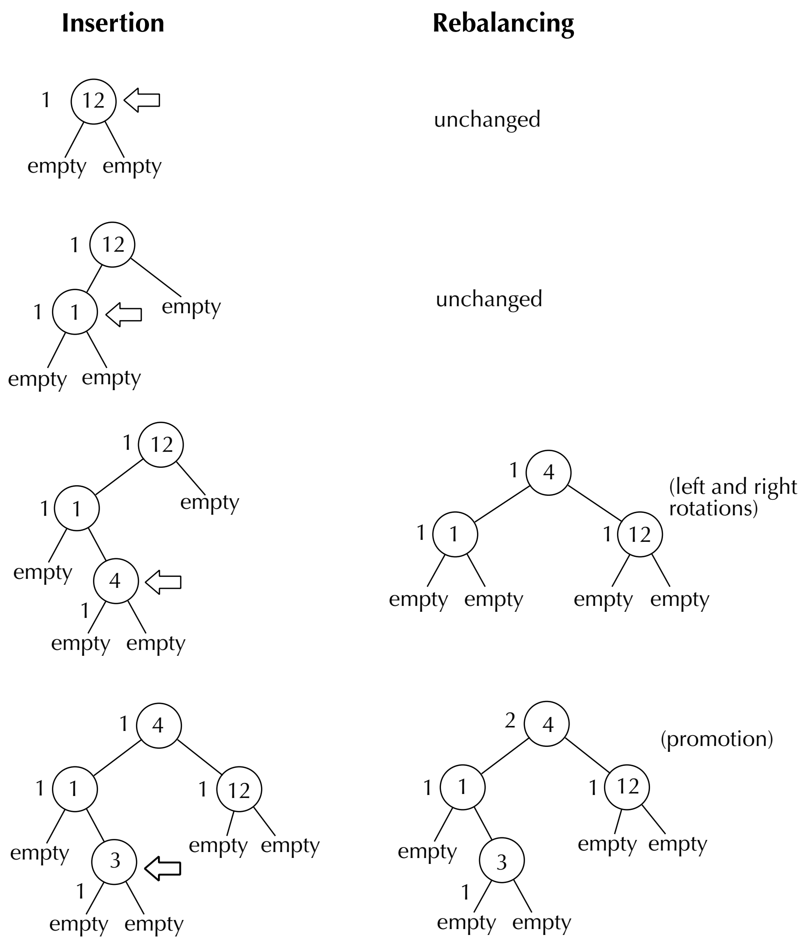


Figure 13.19 Inserting 12, 1, 4, and 3 into an empty red-black tree

tion(s) (if any), as previously described. Remember that after you do a promotion, you need to check to see whether it introduced a new invariant violation, necessitating further action.

a. 13

b. 14

c. −10

d. −5

e. 15

f. 16

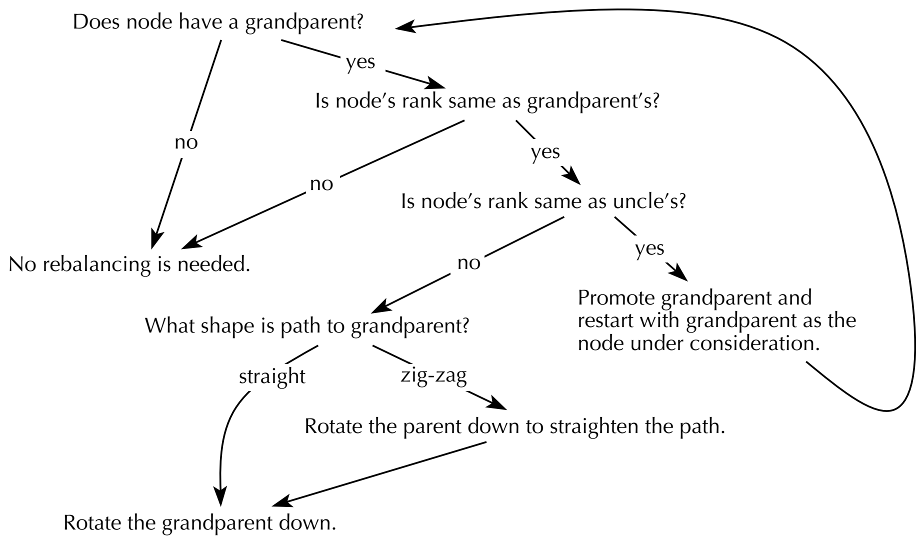

Does node have a grandparent?
yes
no
Is node's rank same as grandparent's?
yes
no
Is node's rank same as uncle's?
yes
no
No rebalancing is needed.
Promote grandparent and restart with grandparent as the node under consideration.
What shape is path to grandparent?
straight
zig-zag
Rotate the parent down to straighten the path.
Rotate the grandparent down.

Figure 13.20 A summary of how to rebalance a red-black tree, starting from an inserted node.

Armed with this background, we now turn to the actual implementation of red-black trees. As indicated in the foregoing, we will make the simplifying assumption that the values in our red-black trees are numbers, inserted according to their numeric value. (The final section in this chapter will consider how red-black trees can be extended to more general data, such as movie databases.) Furthermore, we will allow multiple copies of an element to be inserted into the tree. Therefore, our basic operators for red-black trees are as follows:

```
(make-red-black-tree)
  ;; returns a newly created empty red-black tree.

(red-black-in? item rb-tree)
  ;; returns #t if item is in rb-tree, otherwise #f.

(red-black-insert! item rb-tree)
  ;; inserts item into rb-tree, maintaining red-black invariants.
  ;; If item is already in rb-tree, another copy of item
  ;; is inserted.
```

Note that we implement `red-black-in?` instead of an operation to do a lookup and return what is found. (We could call such an operation `red-black-retrieve`.) The two procedures are very similar, and retrieval makes little sense for pure numeric trees. We will consider how to convert `red-black-in?` into `red-black-retrieve`

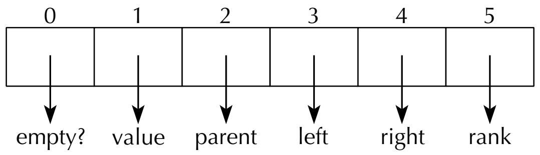


Figure 13.21 Representation of ranked binary trees

in the last section of this chapter. Furthermore, for simplicity, we have decided to not implement deletion.

We have noted that red-black trees are a special class of binary search trees, which in turn are a special class of binary trees. This suggests a "layered" strategy for implementing red-black trees: First we implement ranked binary trees, which are simply mutable binary trees where every node has a rank and where we can access a node's parent as well as its left and right subtrees. None of the binary search or rank conditions hold for these trees; they are just the low-level stratum on which we will construct binary search and red-black trees. On top of this layer we build binary search trees, and on top of that layer we build red-black trees.

So we first turn to the implementation of ranked binary trees. Conceptually, we are extending the binary trees of Chapter 8 by allowing mutation as well as access to a node's parent and rank. In particular, we need something to mutate, so an empty tree cannot simply be the empty list; it should have the same structure as nonempty trees. Figure 13.21 describes the representation we will use for ranked binary trees as six-element vectors, where the names at the end of the arrows indicate the meanings of the various cells.

Figure 13.22 gives an implementation for ranked binary trees in terms of this representation. The code is fairly straightforward because most of what we do involves the selection and mutation of the various attributes of ranked binary trees. The mutators take care to maintain the simple representation invariants that apply to all binary trees. For example, it is impossible using these mutators to set the value without marking the tree as nonempty, and (even more importantly) if *node*1 is made a child of *node*2, *node*2 is automatically made the parent of *node*1. Note, however, that not all cells in a vector need to be set; for example, the first cell being `#t` indicates that the tree is empty, so we don't care about the values in cells 1, 3, and 4 (the value, the left-subtree, and the right-subtree, respectively). Also note that we use `#f` in cell 2 (the parent cell) to indicate that the node has no parent (i.e., that we are at the root node of the tree). This is a subtle difference from the binary trees of Chapter 8, because there we had no absolute notion of root: Each node was the root of its own subtree. Here we consider the root node to be the "top-most" node in the tree, that is, the node you get to by following up the parent links as far as possible. We therefore include a selector `root?`, which determines whether we are at the root node of the tree.

```
(define make-empty-ranked-btree
  (lambda ()
    (let ((tree (make-vector 6)))
      (vector-set! tree 0 #t) ; empty-tree? = true
      (vector-set! tree 2 #f) ; has no parent
      (vector-set! tree 5 0)  ; rank = 0
      tree)))

(define empty-tree?          (define set-empty! ;makes tree empty
  (lambda (tree)               (lambda (tree)
    (vector-ref tree 0)))        (vector-set! tree 0 #t)))

(define value                (define set-value!
  (lambda (tree)               (lambda (tree item)
    (vector-ref tree 1)))        (vector-set! tree 0 #f) ;not empty
                                 (vector-set! tree 1 item)))

(define parent               (define root?
  (lambda (tree)               (lambda (tree)
    (vector-ref tree 2)))        (not (vector-ref tree 2))))

(define left-subtree         (define set-left-subtree!
  (lambda (tree)               (lambda (tree new-subtree)
    (vector-ref tree 3)))        (vector-set! new-subtree 2 tree) ;parent
                                 (vector-set! tree 3 new-subtree)))

(define right-subtree        (define set-right-subtree!
  (lambda (tree)               (lambda (tree new-subtree)
    (vector-ref tree 4)))        (vector-set! new-subtree 2 tree) ;parent
                                 (vector-set! tree 4 new-subtree)))

(define rank                 (define set-rank!
  (lambda (tree)               (lambda (tree rank)
    (vector-ref tree 5)))        (vector-set! tree 5 rank)))
```

Figure 13.22 Basic operators for ranked binary trees

Although the procedures in Figure 13.22 give a complete implementation of ranked binary trees, there are certain procedures that will prove useful later when we use ranked binary trees to implement binary search trees and red-black trees. In particular, the insertion algorithm in red-black trees requires us to know where we are in the tree (for example, is the current node the left or right child of its parent?) and also to move around easily (for example, to the current node's sibling). The following two procedures accomplish these tasks (note that we use the built-in

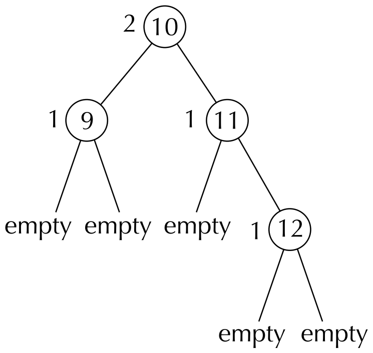
Scheme predicate `eq?`, which tests whether its two arguments actually are the same Scheme object):

```
(define which-subtree
  (lambda (tree)
    ;; Returns the symbol left if tree is left-subtree of its
    ;; parent and the symbol right if it is the right-subtree
    (cond ((root? tree)
           (error "WHICH-SUBTREE called at root of tree."))
          ((eq? tree (left-subtree (parent tree)))
           'left)
          (else 'right))))

(define sibling
  (lambda (tree)
    (cond ((root? tree)
           (error "SIBLING called at root of tree."))
          ((equal? (which-subtree tree) 'left)
           (right-subtree (parent tree)))
          (else
           (left-subtree (parent tree))))))
```

Exercise 13.16

Write `display-ranked-btree` so that it produces output such as that shown in Figure 13.23 when given the tree shown in that figure. Each line of output corresponds to one node; the indentation level indicates the depth of the node in the tree, and the value (or emptiness) and rank are shown explicitly. Each node is followed by its left-subtree descendants and then its right-subtree descendants.



```
10 (rank 2)
  9 (rank 1)
    empty (rank 0)
    empty (rank 0)
  11 (rank 1)
    empty (rank 0)
    12 (rank 1)
      empty (rank 0)
      empty (rank 0)
```

Figure 13.23 An example of `display-ranked-btree`

We next turn to the implementation of binary search trees. As with red-black trees, we will make the simplifying assumption that the values in our trees are numbers, and we will allow multiple copies of an element to be inserted into the tree. Therefore, our basic operators for binary search trees are as follows:

```
(make-binary-search-tree)
  ;; returns a newly created empty binary search tree.

(binary-search-in? item bs-tree)
  ;; returns #t if item is in bs-tree, otherwise #f.

(binary-search-insert! item bs-tree)
  ;; inserts item into bs-tree, maintaining the binary search
  ;; invariant. If item is already in bs-tree, another
  ;; copy of item is inserted.
```

Again, for simplicity, we will not implement deletion.

The first two operators are easy because we will define an empty binary search tree to be an empty ranked binary tree, and `binary-search-in?` can be implemented the same way as the procedure `in?` was in Chapter 8:

```
(define make-binary-search-tree make-empty-ranked-btree)


(define binary-search-in?
  (lambda (item bs-tree)
    (cond ((empty-tree? bs-tree)
           #f)
          ((= item (value bs-tree))
           #t)
          ((< item (value bs-tree))
           (binary-search-in? item (left-subtree bs-tree)))
          (else
           (binary-search-in? item (right-subtree bs-tree))))))
```

To insert something into a binary search tree, we must first find the point where it should be inserted. In other words, we move down the tree, using the tree's order condition (i.e., exploiting the representation invariant), until we finally arrive at the (empty) leaf node where the item should go. Because we are allowing multiple copies of an item to be inserted, we need to decide in which direction to go if we encounter the item while moving downward. Our choice is to move rightward if the item is encountered; that way, the new node will occur "later" in the tree.

We determine the point at which the item should be inserted through a procedure `insertion-point`, thereby simplifying the code for `binary-search-insert!`:

```
(define insertion-point
  (lambda (item bs-tree)
    ;; This procedure finds the point at which item should be
    ;; inserted in bs-tree. In other words, it finds the empty
    ;; leaf node where it should be inserted so that the
    ;; binary search condition still holds after it is inserted.
    ;; If item is already in bs-tree, then the insertion
    ;; point will be found by searching to the right so that
    ;; the new copy will occur "later" in bs-tree.
    (cond ((empty-tree? bs-tree) bs-tree)
          ((< item (value bs-tree))
           (insertion-point item (left-subtree bs-tree)))
          (else
           (insertion-point item (right-subtree bs-tree))))))

(define binary-search-insert!
  (lambda (item bs-tree)
    ;; This procedure will insert item into bs-tree at a leaf
    ;; (using the procedure insertion-point), maintaining
    ;; the binary search condition on bs-tree. The return value
    ;; is the subtree that has item at its root.
    ;; If item occurs in bs-tree, another copy of item
    ;; is inserted into bs-tree
    (let ((insertion-tree (insertion-point item bs-tree)))
      (set-value! insertion-tree item)
      (set-left-subtree! insertion-tree
                         (make-binary-search-tree))
      (set-right-subtree! insertion-tree
                          (make-binary-search-tree))
      insertion-tree)))
```

A couple of remarks need to be made about `binary-search-insert!`. First, we have specified its return value, the newly inserted node (rather than the `bs-tree` itself, for example), because our red-black insertion procedure will need to readjust the tree starting at the insertion point, and it would be handy to know where that insertion point is.

The second remark is a warning. Nonempty binary trees, as we have implemented them, are examples of *cyclic* structures, meaning that it is possible to move around the nodes in the tree, eventually returning to the starting node. An example would be simply going from the root node to one of its children and then back again through the parent link. This might seem innocuous enough, and in fact this cyclicality is important for our needs. However, this property could be disastrous if we allow the read-eval-print loop to display a tree. After all, to print out a node would require that its children be printed out, which in turn requires that the children's parent be printed out, thereby leading to an infinite loop. The moral of this story is never to let the read-eval-print loop display a cyclic structure. In our case, we can use the procedure `display-ranked-btree` from Exercise 13.16.

We finally turn to the implementation of the red-black tree operations listed earlier. Two of these operations are trivial, because red-black trees are a special class of binary search trees:

```
(define make-red-black-tree make-binary-search-tree)

(define red-black-in? binary-search-in?)
```

That leaves only `red-black-insert!` yet to be implemented. As we said in the foregoing, our strategy will be to first use `binary-search-insert!` to insert the node and then to use promotion, right rotation, and left rotation to rebalance the tree, starting at the newly inserted node and progressing upward. Hence we must implement these three operations before going on to `red-black-insert!`. Of these three, promotion is the easiest:

```
(define promote!
  (lambda (node)
    (set-rank! node (+ (rank node) 1))))
```

To implement `rotate-left!` and `rotate-right!`, we need to move things around in the tree. We choose to do this through two more elementary procedures. The first one, `exchange-values!`, takes two nonempty nodes and exchanges their respective values, as illustrated in Figure 13.24. We can implement `exchange-values!` as follows:

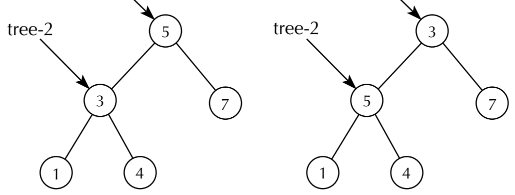


Figure 13.24 Effect of `(exchange-values! tree-1 tree-2)`

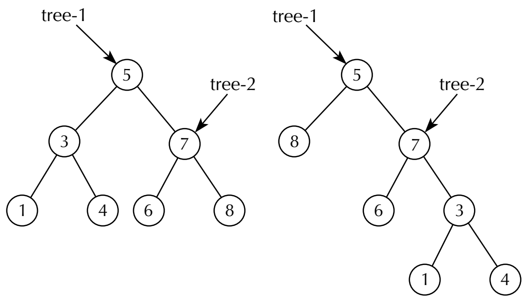


Figure 13.25 Effect of `(exchange-left-with-right! tree-1 tree-2)`

```
(define exchange-values!
  (lambda (node-1 node-2)
    (let ((value-1 (value node-1)))
      (set-value! node-1 (value node-2))
      (set-value! node-2 value-1))))
```

The other procedure, `exchange-left-with-right!`, takes two nonempty trees and exchanges the left subtree of the first with the right subtree of the second, as illustrated in Figure 13.25. In particular, `(exchange-left-with-right! tree tree)` "flips" the two children of `tree`.

```
(define exchange-left-with-right!
  (lambda (tree-1 tree-2)
    (let ((left (left-subtree tree-1))
          (right (right-subtree tree-2)))
      (set-left-subtree! tree-1 right)
      (set-right-subtree! tree-2 left))))
```

The two rotation procedures become fairly straightforward using `exchange-values!` and `exchange-left-with-right!`. For example, Figure 13.26 illustrates how `rotate-left!` can be accomplished through a sequence of exchanges. The corresponding code for `rotate-left!` (and by analogy, for `rotate-right!`) follows:

```
(define rotate-left!
  (lambda (bs-tree)
    (exchange-left-with-right! bs-tree
                               (right-subtree bs-tree))
    (exchange-left-with-right! (right-subtree bs-tree)
                               (right-subtree bs-tree))
```

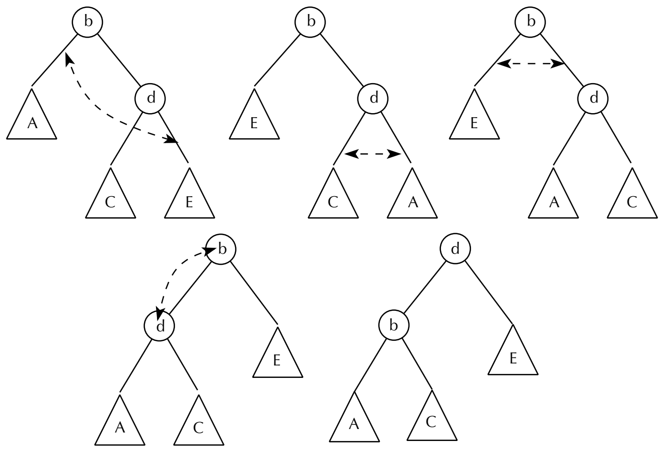


Figure 13.26 Left rotation through a sequence of exchanges

```
    (exchange-left-with-right! bs-tree
                               bs-tree)
    (exchange-values! bs-tree (left-subtree bs-tree))
    'done))

(define rotate-right!
  (lambda (bs-tree)
    (exchange-left-with-right! (left-subtree bs-tree)
                               bs-tree)
    (exchange-left-with-right! (left-subtree bs-tree)
                               (left-subtree bs-tree))
    (exchange-left-with-right! bs-tree
                               bs-tree)
    (exchange-values! bs-tree (right-subtree bs-tree))
    'done))
```

Exercise 13.17

Other sequences of exchanges also exist that will accomplish left rotation. Map one of them out analogously to Figure 13.26 and then write the corresponding alternate definition for `rotate-left!`.

We finally arrive at the procedure `red-black-insert!`, which is now accomplished fairly easily using the tools we have developed:

```
(define red-black-insert!
  (lambda (item red-black-tree)
    (define rebalance!
      (lambda (node)
        (cond ((root? node)
               'done)
              ((root? (parent node))
               'done)
              ((< (rank node) (rank (parent (parent node))))
               'done)
              ((= (rank node) (rank (sibling (parent node))))
               (promote! (parent (parent node)))
               (rebalance! (parent (parent node))))
              (else
               (let ((path-from-grandparent
                      (list (which-subtree (parent node))
                            (which-subtree node))))
                 (cond ((equal? path-from-grandparent '(left left))
                        (rotate-right! (parent (parent node))))
                       ((equal? path-from-grandparent '(left right))
                        (rotate-left! (parent node))
                        (rotate-right! (parent (parent node))))
                       ((equal? path-from-grandparent '(right left))
                        (rotate-right! (parent node))
                        (rotate-left! (parent (parent node))))
                       (else ; '(right right)
                        (rotate-left! (parent (parent node))))))))))
    (let ((insertion-node (binary-search-insert! item
                                                 red-black-tree)))
      (set-rank! insertion-node 1)
      (rebalance! insertion-node))
    'done))
```

Notice that each of the three kinds of trees we layered on top of one another—ranked binary trees, binary search trees, and red-black trees—had mutators that took care to maintain the appropriate invariant. At the ranked binary tree level, the mutators ensured that *node*1 couldn't become a child of *node*2 without *node*2 becoming the parent of *node*1, thus maintaining an important structural invariant. At the binary search tree level, the insertion procedure made sure to maintain the ordering invariant that the `binary-search-in?` procedure relied upon for correct operation. And at the red-black tree level, the `red-black-insert!` procedure maintained the additional invariant properties needed to guarantee $O(\log(n))$ time operation.

13.6 An Application: Dictionaries

To make the exposition clearer in Section 13.5, we restricted the red-black trees to storing numbers rather than more complex records. Some more interesting (and typical) examples of how red-black trees can be applied once the restriction to numbers is lifted were cited at the beginning of that section: databases consisting of driver's license information or student records, dictionaries containing definitions, card catalogs containing book records, or the movie database in Joe Franksen's video store (Section 8.1). In this section we will modify the red-black trees to accommodate the construction of, retrieval from, and maintenance of Joe's movie database.

In each of the cited examples, something is used to look up the records: for drivers' licenses, perhaps the license number; for student records, perhaps the student's name; for dictionaries, the word being defined; etc. In some cases, there might be more than one thing that we could use for looking up: For example, we might look up a movie record either by its title or by its director. The aspect of the record we use for retrieval is called the *key*. Thus, we retrieve a *record* from a *database* by its *key*. Several records may share the same key, in which case retrieval using that key should obtain all those records.

We will use the term *dictionary* as a general term to refer to a mutable data type that stores records and allows retrieval by key, even if the keys aren't words and the records aren't definitions. How can we use keys to organize and retrieve data? Can we be more specific about how we operate on keys? Well, we need to be able to *extract* the key from any given record, and we need to be able to *compare* two different keys to see which one is larger or if they are equal. We will call the procedure that gets the key from the record the *key-extractor*. For example, if we were keying on the movie's title, then the movie ADT selector `movie-title` would be the key-extractor. On the other hand, we will call the procedure that compares two key values the *key-comparator*.

How should we compare two key values? Of course that depends on what the keys are, but we must give a specification of the general form of the comparison procedures. Keeping in mind that keys can compare in three ways ($<$, $=$, or $>$), we will specify that the key-comparator should take two key arguments and return one of the symbols `<`, `=`, or `>` according to whether the first key is less than, equal to, or greater than the second key. For example, if our keys were strings, then we could use the built-in Scheme procedures `string<?` and `string=?` to implement `string-comparator`:

```
(define string-comparator
  (lambda (string-1 string-2)
    (cond ((string<? string-1 string-2) '<)
          ((string=? string-1 string-2) '=)
          (else '>))))
```

In summary, a key-extractor takes a data record and returns a key, whereas a key-comparator takes two keys and returns one of the symbols `<`, `=`, and `>`.

Exercise 13.18

Scheme has a built-in procedure `symbol->string` that takes a symbol and returns the corresponding string. Use `symbol->string` and `string-comparator` to write the procedure `symbol-comparator`, that compares two Scheme symbols. Thus, you should have the following interaction:

```
(symbol-comparator 'erick 'karl)
<

(symbol-comparator 'barbara 'Barbara)
=
```

Note that `symbol-comparator` returned `=` in the latter case because in Scheme the two expressions `'barbara` and `'Barbara` evaluate to the exact same symbol. (The name of that symbol, returned by `symbol->string`, can be either `"barbara"` or `"BARBARA"`, depending on the particular Scheme implementation.)

Exercise 13.19

Use `symbol-comparator` to write `symbol-list-comparator`, which takes two lists of symbols and returns the appropriate comparison symbol. You should have the following interaction:

```
(symbol-list-comparator '(karl wesley)
                        '(karl knight))
>

(symbol-list-comparator '(abba dabba)
                        '(abba dabba doo))
<
```

We will make it the responsibility of the dictionary to store the key-extractor and key-comparator in addition to the underlying database, which allows the dictionary to make use of comparisons between keys in organizing the database. For example, we will create two dictionaries in this section: one that allows us to retrieve movie records by title and the other one by director. Although they will share the same underlying data, it will be organized in two different ways.

Following are the basic operators for dictionaries:

```
(make-dictionary key-comparator key-extractor)
  ;; returns a newly created empty dictionary with given
  ;; key-comparator and key-extractor

(dictionary-retrieve key dictionary)
  ;; returns the list of all items in dictionary matching key

(dictionary-insert! item dictionary)
  ;; inserts item into dictionary, allowing multiple copies
```

Note that as in the last section, we will not implement deletion.

Because we are going to layer dictionaries on red-black trees, which are in turn layered on binary search trees, we need to next extend binary search trees so that they operate on keys. We will not have binary search trees nor red-black trees store the key-extractor and key-comparator; that will be additional information stored by dictionaries. As a result, the constructors `make-binary-search-tree` and `make-red-black-tree` will remain unmodified. Instead, we will have to modify the other operators so that they will take two additional arguments, the key-comparator and key-extractor.

Exercise 13.20

Modify the procedure `insertion-point` so that a call of the form

```
(insertion-point item bs-tree
                 key-comparator key-extractor)
```

will find the appropriate empty leaf node where the `item` should be inserted.

Exercise 13.21

Modify the procedure `binary-search-insert!` so that a call of the form

```
(binary-search-insert! item bs-tree
                       key-comparator key-extractor)
```

will properly insert `item` into `bs-tree`.

During the course of modifying binary search trees and red-black trees to operate on keys, you will need to test that the procedures work correctly. You can do this using data from the movie database, `our-movie-database` in Section 7.6, which is included in the software on the web site for this book. You could then do the following calls to create a new binary search tree `bs-tree` and insert two elements into it (note that we wrap the calls to `binary-search-insert!` inside a `begin` expression that ends with `'done`, because otherwise the read-eval-print loop would have problems displaying the cyclic structure returned by `binary-search-insert!`):

```
(define bs-tree (make-binary-search-tree))

(begin (binary-search-insert! (make-movie '(amarcord)
                                           '(federico fellini)
                                           1974
                                           '((magali noel)
                                             (bruno zanin)
                                             (pupella maggio)
                                             (armando drancia)))
                              bs-tree
                              symbol-list-comparator
                              movie-title)
       'done)

(begin (binary-search-insert! (make-movie '(the big easy)
                                           '(jim mcbride)
                                           1987
                                           '((dennis quaid)
                                             (ellen barkin)
                                             (ned beatty)
                                             (lisa jane persky)
                                             (john goodman)
                                             (charles ludlam)))
                              bs-tree
                              symbol-list-comparator
                              movie-title)
       'done)
```

Exercise 13.22

In the course of testing your procedures, you will often need to display the trees you are manipulating. You could use the procedure `display-ranked-btree` from Exercise 13.16. Unfortunately, that procedure would display each entire movie record, which would make examining the output difficult.

Make a variant of the procedure `display-ranked-btree` called `display-ranked-btree-by` that takes an additional argument, a selector operating on records. For each nonempty node of the tree, the selector is used to obtain what should be displayed. For example, you should get the following output given that you had defined `bs-tree` as above:

```
(display-ranked-btree-by bs-tree movie-title)
(amarcord) (rank 0)
  empty (rank 0)
  (the big easy) (rank 0)
    empty (rank 0)
    empty (rank 0)
```

Exercise 13.23

Using `binary-search-in?` as a model, write the procedure `binary-search-retrieve`, that will additionally take a key-comparator and key-extractor, and will return a list of all the records matching the key. Thus, you should have the following interaction using the previously defined `bs-tree`:

```
(binary-search-retrieve '(the big easy)
                        bs-tree
                        symbol-list-comparator
                        movie-title)
(((the big easy) (jim mcbride) 1987 ((dennis quaid)
(ellen barkin) (ned beatty) (lisa jane persky) (john goodman)
(charles ludlam))))
```

For efficiency you should use the “onto” parameter idea introduced in Section 8.1, rather than using `append`.

We next need to extend red-black trees so that they operate on keys. Two of the operators remain the same because they are lifted directly from binary search trees:

```
(define make-red-black-tree make-binary-search-tree)

(define red-black-retrieve binary-search-retrieve)
```

Exercise 13.24

Modify `red-black-insert!` so that it additionally takes a key-comparator and a key-extractor as arguments. Thus, you can construct and put one element into a red-black tree as follows:

```
(define rb-tree (make-red-black-tree))

(red-black-insert! (make-movie '(amarcord)
                               '(federico fellini)
                               1974
                               '((magali noel)
                                 (bruno zamin)
                                 (pupella maggio)
                                 (armando drancia)))
                   rb-tree
                   symbol-list-comparator
                   movie-title)
```

We are finally at the point where we can implement dictionaries. Because we require that a dictionary keeps track of its key-comparator and key-extractor, we start the implementation of dictionaries as follows:

```
(define make-dictionary
  (lambda (key-comparator key-extractor)
    (vector key-comparator
            key-extractor
            (make-red-black-tree))))

(define key-comparator
  (lambda (dictionary)
    (vector-ref dictionary 0)))

(define key-extractor
  (lambda (dictionary)
    (vector-ref dictionary 1)))

(define red-black-tree
  (lambda (dictionary)
    (vector-ref dictionary 2)))
```

Note that the three selectors are for internal usage by dictionaries. A person using dictionaries would use the operators `make-dictionary`, `dictionary-insert!`, and `dictionary-retrieve`. Thus, we would create our desired movie dictionaries as follows (though they don't yet contain the data):

```
(define our-movies-by-title
  (make-dictionary symbol-list-comparator movie-title))
```

```
(define our-movies-by-director
  (make-dictionary symbol-list-comparator movie-director))
```

Exercise 13.25

Implement the procedure `dictionary-insert!`.

Exercise 13.26

Implement the procedure `dictionary-retrieve`.

Exercise 13.27

Scheme has a built-in procedure `for-each` that takes a procedure and a list and applies the procedure to each element of the list. `For-each` is very similar to `map`, except that it is done for effect, not for its return value. Thus, you would have the following interaction:

```
(for-each (lambda (n)
            (newline)
            (display (* n n)))
          '(1 2 3 4))
1
4
9
16
```

(In this example, all the output is produced by explicit `newline` and `display` invocations. If you try this evaluation, you'll probably also see a value returned by the `for-each` procedure itself, but the Scheme standard leaves that value unspecified.) Use `for-each` to insert the data from `our-movie-database` appropriately into our two dictionaries `our-movies-by-title` and `our-movies-by-director`.

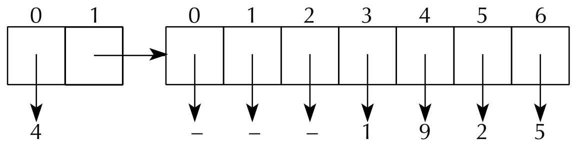
Review Problems

Exercise 13.28

Suppose we want to add the factorial operation, indicated by a postfix `!`, to the shift/reduce evaluator of Section 13.2. That is, because 4! = 24, we should have `(evaluate "2*4!")` produce 48 and `(evaluate "(2*2)!")` produce 24. Notice (from the first example) that `!` has higher precedence even than `*` and `/`, so to apply the factorial operation to anything other than a single number or another factorial, you need parentheses. We'll now have a new kind of reduction. If the top item on the stack is a `!` and a number is below it, we can reduce by popping the two of them off and pushing the factorial on. Extend Figure 13.4 with an extra column and an extra row, each labeled with `!`. Fill in each of the new blank cells in this row and column with shift, reduce, or error as appropriate; you should detect all errors as early as possible.

▷ Exercise 13.29

Suppose we changed our first representation of RA-stacks so that the stack's elements are stored at the end of the *cells* vector, with the bottom element of the stack in the last element of the vector. For example, a representation of the stack <u>5 2 9 1</u>, where 1 is the top element, could look like the following:



Of the procedures `make-ra-stack-with-at-most`, `height`, `top-minus`, `pop!`, and `push!`, which ones would need changing (relative to the versions from Section 13.3's representation 1) and which wouldn't? Justify your answer.

▷ Exercise 13.30

Reimplement the abstract data type for movies in a mutable version that supports all the same operations (the `make-movie` constructor and such selectors as `movie-title`) and also the following additional operations: `check-movie-out-to!`, `check-movie-in!`, and `movie-status`. Initially, a newly created movie should be considered checked in. The `check-movie-out-to!` operation can only be done on a movie that is currently checked in; otherwise an error is signaled. Conversely, only a movie that is currently checked out can be checked in, or `check-movie-in!` will signal an error. The `check-movie-out-to!` procedure takes a person's name as a second argument and records that information in the movie object. The `movie-status` procedure returns the name of the person to whom the movie is checked out, if it is checked out, or `#f` if it is checked in.

▷ Exercise 13.31

In Chapter 1 we emphasized that the `quarter-turn-right` procedure didn't really rotate an image a quarter turn to the right in the sense of changing the image; a new image was created instead, looking like the original image would have had it been turned.

Now we have the ability to make objects that can really be turned in the sense of themselves changing. Define a mutable abstract data type *turnable image* with a constructor `make-turnable-image` that takes a normal image as its argument, a mutator `quarter-turn-right!` that updates the turnable image, and selector `get-image` that returns a normal image showing the current status of the turnable image.

Write three versions, and make sure they all work indistinguishably. They should use the following approaches:

a. The object contains just a normal image. The turn mutator replaces this image with a new one reflecting the turn. The selector returns it.

b. The object contains both a normal image and a turn count in the range 0 to 3. The turn mutator leaves the image unchanged and instead updates the turn count. The selector uses the turn count and image in order to produce the properly turned image to return.

c. The object contains both a normal image and a turn count in the range 0 to 3, as in part **b**. The mutator acts as in part **b**, updating the turn count. However, the selector is different. It not only uses the turn count and image to calculate the turned image to return, but it also then stores that turned image into the object and sets the turn count to 0.

Exercise 13.32

Recall that in Chapter 6 we wrote a two-pile Nim game that used a game-state ADT. We implemented the ADT in various different ways, including using `cons`-pairs. Suppose we wanted to model game states, but *with mutation*. In other words, we will implement the following constructor, selector, and mutator:

```
(make-game-state n m)
  ;; returns a game state with n coins in the first
  ;; pile and m coins in the second pile

(size-of-pile game-state p)
  ;; returns an integer equal to the number of coins
  ;; in the p-th pile of the game-state (p = 1 or 2)

(remove-coins-from-pile! game-state n p)
  ;; changes game-state so that there are n fewer
  ;; coins in pile p (p = 1 or 2). The return value of
  ;; remove-coins-from-pile! is unspecified.
```

Note that `remove-coins-from-pile!` actually alters `game-state` so that pile *p* has *n* fewer coins.

Implement this mutable game state ADT using vectors.

Exercise 13.33

The three procedures below are the constructor, mutator, and selector for a new kind of object, the widget. Describe in English how widgets behave, from the standpoint of someone using these three procedures but not knowing what is going on inside them or how the widgets are being represented. That is, your explanation shouldn't talk about vectors or vector positions at all but instead should talk about how widget insertion and retrieval relate. If some insertions and retrievals are done, how could you predict what each retrieval was going to retrieve? Once you've provided this outsider's perspective, provide a justification of it in terms of the internal behavior of the procedures. That is, explain how it is that the vector operations these procedures do result in the previously stated external behavior.

```
(define make-widget
  (lambda ()
    (let ((widget (make-vector 3)))
      (vector-set! widget 0 'empty)
      (vector-set! widget 1 'empty)
      (vector-set! widget 2 0)
      widget)))

(define insert-into-widget!
  (lambda (widget value)
    (let ((place (vector-ref widget 2)))
      (vector-set! widget place value)
      (vector-set! widget 2 (remainder (+ place 1) 2))
      'done)))

(define retrieve-from-widget
  (lambda (widget)
    (vector-ref widget (vector-ref widget 2))))
```

Exercise 13.34

In Chapter 10 we turned Micro-Scheme into Mini-Scheme by introducing definitions and used global environments to hold the name/value associations resulting from those definitions.

Unfortunately, there was a serious modularity problem. Only the read-eval-print-loop ever added an association to the global environment, and only the evaluation of name ASTs ever looked a name up in the global environment. Yet the global environments needed to be passed around throughout the rest of the program, in particular through all the other kinds of ASTs.

Now that we know how to define mutable data types, we can implement Mini-Scheme much more cleanly. We can start with Micro-Scheme and change only the two communicating partners (read-eval-print-loop and the name ASTs), leaving the rest unchanged, because the communication can now be done through an object with state.

In particular, suppose we do the following definitions:

```
(define make-read-eval-print-loop-state
  (lambda ()
    (make-vector 1)))

(define set-global-environment!
  (lambda (repl-state new-global-env)
    (vector-set! repl-state 0 new-global-env)))

(define get-global-environment
  (lambda (repl-state)
    (vector-ref repl-state 0)))

(define repl-state (make-read-eval-print-loop-state))
```

At this point, the read-eval-print-loop can take charge of setting the global environment into the repl-state, and the name ASTs can get the current global environment back out from there.

As a starting point, you should use all the code from the Micro-Scheme implementation except read-eval-print-loop, as well as the four preceding definitions and the following procedures from Chapter 10's Mini-Scheme implementation: read-eval-print-loop, definition?, definition-name, definition-expression, look-up-value-in, make-initial-global-environment, and extend-global-environment-with-naming.

a. Modify make-name-ast so the name ASTs use look-up-value-in rather than look-up-value. They should get the global environment from the repl-state.

b. Modify the Mini-Scheme read-eval-print-loop so that it uses evaluate rather than evaluate-in and so that at the top of the loop it sets the global-environment into the repl-state.

Exercise 13.35

Insert the following numbers in order into an initially empty red-black tree, rebalancing after each insertion. Show the tree after each insertion, but before the rebalancing, and again after each rebalancing. You must show the rank of each node as well as the value stored in it but may omit the empty leaf nodes from your diagrams if you prefer. The numbers are 5, 1, −5, 0, −2, and −1.

Chapter Inventory

Vocabulary

precedence
reduce
token
shift
accept
mutable data type
object
value
axiomatic system
representation invariant
legal
vacuous
header
linked list
cycle
last in first out (LIFO)
first in first out (FIFO)
rank
promotion
right rotation
left rotation
cyclic
key
record
database
key-extractor
key-comparator

Abstract Data Types

RA-stacks
nodes
node-lists
queues
binary search trees
red-black trees
ranked binary trees
dictionaries
turnable images
mutable game states
widgets
read-eval-print-loop states

New Predefined Scheme Names

```
make-string
string-length
string-ref
string-set!
string->number
string->symbol
char-numeric?
string
set-car!
set-cdr!
eq?
string<?
```

```
string=?
symbol->string
for-each
```

New Scheme Syntax

character constants (`#\`)

Scheme Names Defined in This Chapter

```
tokenize
make-ra-stack
empty-ra-stack?
height
top-minus
pop!
push!
reduce?
accept?
shift?
reduce!
operator?
lower-precedence?
operator-char?
digit->number
display-ra-stack
make-ra-stack-with-at-most
cells
set-height!
make-node
node-element
node-rest
nodes-down
node-set-element!
node-set-rest!
make-queue
empty-queue?
head
dequeue!
enqueue!
queue-length
set-queue-length!
queue-start
queue-cells
set-queue-cells!
enlarge-queue!
red-black-insert!
make-red-black-tree
red-black-in?
root?
which-subtree
make-empty-ranked-btree
empty-tree?
set-empty!
value
set-value!
parent
left-subtree
set-left-subtree!
right-subtree
set-right-subtree!
rank
set-rank!
sibling
display-ranked-btree
make-binary-search-tree
binary-search-in?
binary-search-insert!
insertion-point
promote
rotate-left!
rotate-right!
exchange-values!
exchange-left-with-right!
string-comparator
symbol-comparator
symbol-list-comparator
make-dictionary
```

`dictionary-retrieve`	`check-movie-in!`
`dictionary-insert!`	`movie-status`
`bs-tree`	`make-turnable-image`
`display-ranked-btree-by`	`quarter-turn-right!`
`binary-search-retrieve`	`get-image`
`red-black-retrieve`	`remove-coins-from-pile!`
`rb-tree`	`make-widget`
`key-comparator`	`insert-into-widget!`
`key-extractor`	`retrieve-from-widget`
`red-black-tree`	`make-read-eval-print-loop-state`
`our-movies-by-title`	`set-global-environment!`
`our-movies-by-director`	`get-global-environment`
`check-movie-out-to!`	`repl-state`

Sidebars

Strings and Characters

Notes

Our treatment of red-black trees is patterned rather closely on Tarjan's [50], so that would be one place to turn for guidance on the deletion operation, which we've omitted. However, it is quite dense reading; for a more lengthy treatment, you could turn to an algorithms and data structures textbook, such as Cormen, Leiserson, and Rivest [14].