CHAPTER EIGHT

Trees

8.1 Binary Search Trees

Joe S. Franksen, one of the co-owners of the video store that uses your query system, has been getting a lot of customer complaints that searching for a video by director takes too long. Now he's hired us to try to fix the problem. The problem doesn't appear to be in the query-matching part of our system. Therefore, we will need to look at the procedures we used for looking up a particular director or video.

Recall that we used a list of video records, and in Exercise 7.22c you wrote a procedure for searching for the ones by a given director. This procedure has to search through the entire list of movies, even if the ones by the specified director happen to be near the front, because it has no way of knowing that there aren't any more movies by the same director later in the list. When Franksen's video rental business was only a small part of his gas station/convenience store, this was no big deal because he only had about a hundred videos. But now that he's expanded his business and acquired 10,000 videos, the time it takes to find one becomes noticeably long.

Are there better ways to structure the list of videos so that finding those by a particular director won't take so long? One idea would be to sort the list, say, alphabetically by the director's name. When we search for a particular director, we can stop when we reach the first video by a director alphabetically "greater than" the one we're searching for.

Is this approach any better? A lot depends on the name of the director. If we're searching for videos directed by Alfred Hitchcock, the search will be relatively quick

(because this name begins with an A), whereas if we're looking for videos directed by Woody Allen, we will still need to search through essentially the entire list to get to W. We can show that, on the average, using a sorted list will take about half the time that using an unsorted one would. From an asymptotic point of view, this is not a significant improvement.

We can find things in a sorted list much faster using what's called a *divide and conquer* approach. The main idea is to divide the list of records we're searching through in half at each point in our search. We start by looking in the middle of the list. If the record we're looking for is the same as the middle element of the list, we are done. If it's smaller than the middle record, we only need to look in the first half of the list, and if it's bigger, we only need to look in the second half. This way of searching for something is often called *binary search*. Because each pass of binary search at worst splits the search space in half, we would expect the time taken to be at worst a multiple of $\log(n)$, where n is the size of the list. (In symbols we say that the time is $O(\log(n))$, pronounced "big oh of log en," which means that for all but perhaps finitely many exceptions, it is known to lie below a constant multiple of $\log(n)$.) For large values of n, this is an enormous improvement because, for example, $\log_2(1{,}000{,}000) \approx 20$, a speed-up factor of $1{,}000{,}000/20 = 50{,}000$.

But we run into trouble when we try to code this up because we can't get to the middle of a list quickly. In fact, the time it takes to get that middle element is long enough to make the binary search algorithm as slow as doing the straightforward linear search that constituted our first and second approaches. Can we do something to our list that is more drastic than just sorting it? In other words, can we somehow arrange the video records so that we could efficiently implement the binary search algorithm? We would need to be able to easily access the middle element (i.e., the one where half the remaining records are larger than it and half are smaller). We would also need to be able to access the records that are smaller than the middle record, as well as those which are larger. Furthermore, both halves should be structured in exactly the same way as the whole set of video records, so we can search the relevant half in the same way.

How do we create such a structure? The answer is to use a data structure based on the above description. Our new data type will have three elements: one movie record (the "middle" one) and two collections of movie records (those that are smaller and those that are larger). This way, we can get at any of the three parts we need by just using the appropriate selector.

This type of structure is called a *binary search tree*. There is the hint of a recursive definition in the preceding discussion: Most binary search trees have a middle element and two subtrees, which are also binary search trees. We need to make this more precise. First, we skipped over the base case: an empty tree. Secondly, we need to define what we mean by a middle element. This is simply one that is greater than every element in one subtree and less than every element in the other subtree. Thus we can make the following definition:

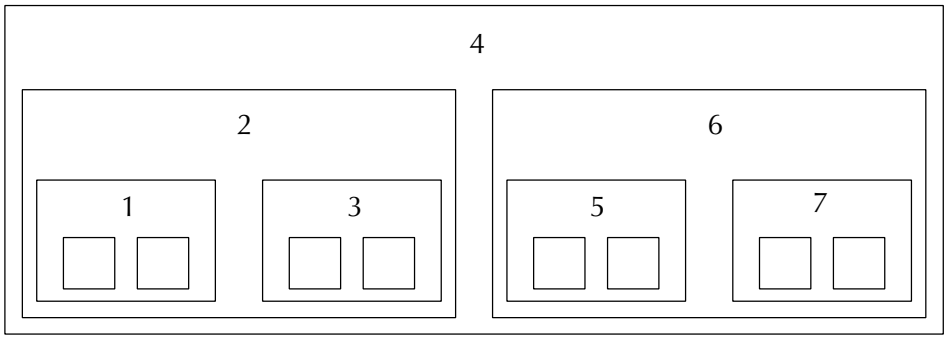
> **Binary search tree:** A binary search tree is either empty or it consists of three parts: the root, the left subtree, and the right subtree. The left and right subtrees are themselves binary search trees. The root is an element that is greater than or equal to each of the elements in the left subtree and less than or equal to each of the elements in the right subtree.

Notice that there is no guarantee in this definition that the root is the median element (i.e., that half of the elements in the tree are less than it and half are greater than it). When the root of a tree is the median, and similarly for the roots of the subtrees, sub-subtrees, etc., the tree will be as short as possible. We will see in the next section that such trees are the binary search trees that are most efficient for searching.

For the remainder of this section, we will work with two kinds of binary search trees, ones that have numbers as their elements and ones that have video records. Because trees with numbers are easier to conceptualize, we will write procedures that work with them first. Then we can easily modify these procedures to work with trees of video records.

In the numerical trees, we will assume that there are no duplicate items. In this case, we say that the tree is strictly ordered. In the video record trees, there are probably lots of "duplicates." Recall that we compare two records by comparing their directors. Because some people direct many videos, we would expect to see one entry for each of these videos in the tree.

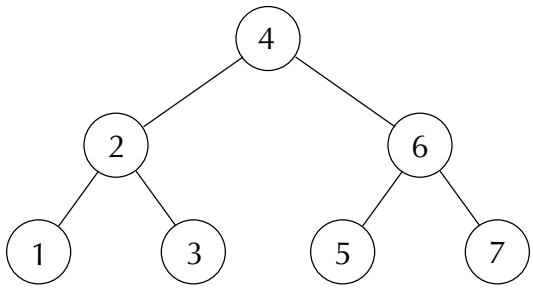
Binary search trees can be represented visually by diagrams in which each tree is a box. Empty trees are represented by empty boxes, and nonempty trees are represented by boxes containing the root value and the boxes for the two subtrees. For example, a small binary search tree with seven elements looks like the following:



Note that the root of the tree, which is 4, is at the top, and the subtrees branch downward. For some obscure reason, mathematicians and computer scientists almost always draw their trees so that they grow upside down. The left subtree of this example tree has 2 for its root. Notice that this subtree is a box much like the outer one, and

so we can talk about its subtrees in turn (with roots 1 and 3), just as we talked about the subtrees of the original tree.

This sort of boxes-within-boxes diagram is probably the best way to think of a tree because it emphasizes the recursive three-part structure. However, another style of diagram is so traditional that it is worth getting used to as well. In this traditional style of tree diagram, the same binary search tree would look like the following:



Here you have to mentally recognize the whole collection of seven "nodes" as a single tree, with the top node as the root, the three nodes on the left grouped together in your mind as one subtree, and the three nodes on the right similarly grouped together as the other subtree. You also have to remember that the "leaves" at the bottom of the tree (1, 3, 5, and 7) are really roots of trees with empty subtrees that are invisible in this style of diagram.

We can implement binary search trees by using lists with three elements. Using the convention that the first element is the root, and the second and third elements are the left and right subtrees, respectively, the list representation of the preceding tree would be

```
(4 (2 (1 () ()) (3 () ())) (6 (5 () ()) (7 () ())))
```

Its tree structure is much easier to see if we write it on several different lines:

```
(4
 (2
  (1 () ())
  (3 () ()))
 (6
  (5 () ())
  (7 () ())))
```

What sort of operations do we need to implement binary search trees? We use two constructors:

```
(define make-empty-tree
  (lambda () '()))

(define make-nonempty-tree
  (lambda (root left-subtree right-subtree)
    (list root left-subtree right-subtree)))
```

and four selectors:

```
(define empty-tree? null?)

(define root car)

(define left-subtree cadr)

(define right-subtree caddr)
```

These procedures are all we need to implement the binary search algorithm given above. Initially, we assume that we're dealing with a binary search tree that has numerical elements and does not have duplicate entries:

```
(define in?
  (lambda (value tree)
    (cond
     ((empty-tree? tree) #f)
     ((= value (root tree)) #t)
     ((< value (root tree)) (in? value (left-subtree tree)))
     (else ; the value must be greater than the root
      (in? value (right-subtree tree))))))
```

Notice how closely this procedure follows the definition of binary search trees. If the tree is empty, the value can't be in the tree. On the other hand, if the tree is not empty, the value is either equal to the root or it's in one of the subtrees. Furthermore, we can tell which subtree it's in by how it compares to the root.

There are two related points worth noting here because they will crop up time and time again. One is the parallelism between the recursive structure of the data and that of the procedure that operates on it. The other is that our one-layer thinking about the design of the procedure goes along with a one-layer perspective on the structure of the data. We don't think about searching through a succession of values in the tree, but rather about looking at the root and then one or the other subtree. Similarly, we don't view the tree as composed of a bunch of values, but rather of a root and two subtrees. We can summarize these points as a general principle for future reference:

The one-layer data structure principle: Hierarchical data structures should not be thought of in their entirety but rather in a one-layer fashion, as a recursive composition of substructures. This one-layer thinking guides one to write recursive procedures that naturally parallel the recursive structure of the data.

Exercise 8.1

Write a procedure called `minimum` that will find the smallest element in a nonempty binary search tree of numbers.

Exercise 8.2

Write a procedure called `number-of-nodes` that will count the number of elements in a binary search tree.

In the video catalog example, we will want a version of `in?` that returns a list of all the videos directed by a given person. Therefore, we will need a procedure that takes a video record and a director's name and determines how the name of the director of the video compares alphabetically to the given name. The director field of the video record is often called the *key* field; the particular name that we're searching for is called the *key value*. Now, any two names could be identical, the first one could come before the second in alphabetical order, or the first one could come after the second. Therefore we'll assume our comparison procedure returns one of three symbols, `=`, `<`, or `>`.

```
(define compare-by-director
  (lambda (video-record name)
    ; Returns one of the symbols <, =, or > according to how the
    ; director in video-record compares alphabetically to name.
    ; For example, if video-record's director alphabetically
    ; precedes name, < would be returned.
    the code implementing this would go here))
```

We're now in a position to modify `in?` so that it can list all of the videos in a binary search tree that are directed by a given person. The basic idea is to traverse the tree looking for a node whose director is the same as the given key value. Once we find such a subtree, we must still search both halves of it, looking for all of the other records that match the key value. This may seem to defeat the efficiency of the procedure. However, it can be shown that so long as the tree isn't unnecessarily tall and skinny, this search method is in fact very efficient.

To make our procedure work generally, and not just for the director, let's suppose that we have a general comparison operator (such as `compare-by-director`). Such

a procedure takes a video record and a key value, compares the appropriate field of the record to the key value, and returns exactly one of the symbols =, <, or >. We can then write a procedure that returns the list of records matching a given key value as follows:

```
(define list-by-key
  (lambda (key-value comparator tree)
    (if (empty-tree? tree)
        '()
        (let ((comparison-result (comparator (root tree)
                                             key-value)))
          (cond
           ((equal? comparison-result '=)
            (cons (root tree)
                  (append (list-by-key key-value comparator
                                       (left-subtree tree))
                          (list-by-key key-value comparator
                                       (right-subtree tree)))))
           ((equal? comparison-result '<)
            (list-by-key key-value comparator
                         (right-subtree tree)))
           (else ;it must be the symbol >
            (list-by-key key-value comparator
                         (left-subtree tree))))))))
```

Of course, because we haven't explained how to do alphabetical comparison, you're not in a very good position to complete the `compare-by-director` procedure above. You could, of course, try `list-by-key` out with an analogous `compare-by-year` instead, or alternatively consult a Scheme reference manual to learn how to do alphabetical comparisons. However, our main point was to illustrate the nature of accessing a binary search tree, not to get into the details of the specific kind of comparison used.

The procedure `list-by-key` typifies a process called *tree traversal*. We call it a *preorder* traversal because we consider the root of the tree first and then the left and right subtrees, in that order. When the root of the tree should be included in the result, it is consed on in front of the elements from the left and right subtrees. The lists from the left and right subtrees are appended together using a built-in procedure we haven't seen before, `append`. Here is a simpler example of `append`:

```
(append '(a b c) '(1 2 3 4))
(a b c 1 2 3 4)
```

We can use this idea of preorder traversal with `cons` and `append` to produce a list of all the nodes in the tree:

```
(define preorder
  (lambda (tree)
    (if (empty-tree? tree)
        '()
        (cons (root tree)
              (append (preorder (left-subtree tree))
                      (preorder (right-subtree tree)))))))
```

The `append` in this procedure can be avoided if we generalize to a `preorder-onto` procedure that conses the tree's nodes onto the front of a specified list. This is analogous to our definition of `reverse` in terms of `reverse-onto` and is motivated by the same concern: efficiency.

```
(define preorder
  (lambda (tree)
    (preorder-onto tree '())))

(define preorder-onto
  (lambda (tree list)
    (if (empty-tree? tree)
        list
        (cons (root tree)
              (preorder-onto (left-subtree tree)
                             (preorder-onto (right-subtree tree)
                                            list))))))
```

Exercise 8.3

Use this technique to eliminate the `append` from `list-by-key`.

One of the problems with `preorder` is that the list it produces is not sorted. We can get a list of the nodes that's sorted by doing what's called an *in order* traversal of the tree. The "in" refers to the fact that you include the root of the tree in between the left and right subtrees:

```
(define inorder
  (lambda (tree)
    (if (empty-tree? tree)
        '()
        (append (inorder (left-subtree tree))
                (cons (root tree)
                      (inorder (right-subtree tree)))))))
```

Now when we call `inorder` on a binary search tree, the resulting list has the elements in it listed in increasing order.

Exercise 8.4

Again, eliminate `append` by using an "onto" parameter.

Exercise 8.5

The third standard way of traversing a tree is called a *postorder* traversal. Here, you enumerate the left subtree, then the right subtree, and finally the root. Write a procedure that takes a binary search tree and produces the list of nodes that describe a postorder traversal of the tree.

Exercise 8.6

Suppose we want to create a new binary search tree by adding another element to an already existing binary search tree. Where is the easiest place to add such an element? Write a procedure called `insert` that takes a number and a binary search tree of numbers and returns a new binary search tree whose elements consist of the given number together with all of the elements of the binary search tree. You may assume that the given number isn't already in the tree.

Exercise 8.7

Using the procedure `insert`, write a procedure called `list->bstree` that takes a list of numbers and returns a binary tree whose elements are those numbers. Try this on several different lists and draw the corresponding tree diagrams. What kind of list gives you a short bushy tree? What kind of list gives a tall skinny tree?

8.2 Efficiency Issues with Binary Search Trees

Now that we have some experience with binary search trees, we need to ask if they really are a better structure for storing our catalog of videos than sorted lists. In order to do that, we first look at a general binary tree and get some estimates on the number of nodes in a tree. We start with some definitions.

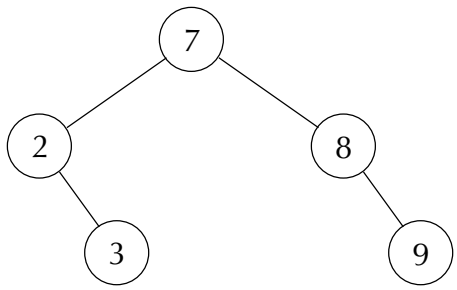
If we ignore the ordering properties that are part of a binary search tree's definition, we get something called a *binary tree*. More precisely,

Binary tree: A binary tree is either empty or it consists of three parts: the root, the left subtree, and the right subtree. The left and right subtrees are themselves binary trees.

Needless to say, binary search trees are special cases of binary trees. Furthermore, we set up the basic constructors and selectors for binary search trees so that they work equally well for implementing binary trees.

There is an enormous amount of terminology commonly used with binary trees. The elements that make up roots of binary trees (or roots of subtrees of binary trees) are called the *nodes* of the tree. In the graphical representation of a tree, the nodes are often represented by circles with values inside them. If a particular node in a binary tree is the root of a subtree that has two empty subtrees, that node is called a *leaf*. On the other hand, if a node is the root of a subtree that has at least one nonempty subtree, that node is called an *internal node*. If you look at the graphical representation, the leaves of a tree are the nodes at the very bottom of the tree and all of the rest of the nodes are internal ones. Of course, if we drew out trees with the root at the bottom of the diagram, the leaves would correspond more closely to real leaves on real trees. The two subtrees of a binary tree are often labeled as the left subtree and the right subtree. Sometimes these subtrees are called the left child or the right child. More commonly, we define a parent-child relationship between nodes. If an internal node has a nonempty left subtree, the root of that left subtree is called the *left child* of the node. The right child is similarly defined. The internal node is the parent node of its children. The parent-child relationship is indicated graphically by drawing an edge between the two nodes. The root of the whole tree has no parent, all internal nodes have at least one and at most two children, and the leaves in a tree have no children at all.

Imagine traveling through a binary tree starting at the root. At each point, we make a choice to go either left or right. If we only travel downward (i.e., without backing up), there is a unique path from the root to any given node. The *depth* of a node is the length of the path from the root to that node, where we define the *length of a path* to be the number of edges that we passed along. For example, if we travel from 7 to 2 to 3 in the tree



we take a path of length 2. The *height* of a tree is the length of the longest path from the root down to a leaf without any doubling back. In other words, it is the maximum depth of the nodes in the tree. Thus, the height of the above tree is 2 because every path from the root to a leaf has length 2. According to our definition, a tree having a single node will have height 0. The height of an empty tree is undefined; in the remainder of this section, we'll assume all the trees we're talking about are nonempty.

Exercise 8.8

Write a predicate that will return true if the root node of a tree is a leaf (i.e., the tree has only one node).

Exercise 8.9

Write a procedure that will compute the height of a tree.

Suppose we have a binary tree of height h. What is the maximum number of nodes that it can have? What is the maximum number of leaves that it can have? These maximum values occur for *complete trees*, where a complete tree of height h is one where all of the leaves occur at depth h and all of the internal nodes have exactly two children. (Why is the number of leaves maximum then?) Let's let $leaves(h)$ and $nodes(h)$, respectively, denote the maximum number of leaves and nodes of a tree of height h and look at a few small examples to see if we can determine a general formula. A tree of height 0 has one node and one leaf. A tree of height 1 can have at most two leaves, and those plus the root make a total of three nodes. A tree of height 2 can have at most four leaves, and those plus the three above make a maximum of seven nodes.

In general, the maximum number of leaves doubles each time h is increased by 1. This combined with the fact that $leaves(0) = 1$ implies that $leaves(h) = 2^h$. On the other hand, because every node in a complete tree is either a leaf or a node that would remain were the tree shortened by 1, the maximum number of nodes of a tree of height $h > 0$ is equal to the maximum number of leaves of a tree of height h plus the maximum number of nodes of a tree of height $h - 1$. Thus, we have derived the following recursive formula, or *recurrence relation*:

$$nodes(h) = \begin{cases} 1 & \text{if } h = 0 \\ leaves(h) + nodes(h-1) & \text{if } h > 0 \end{cases}$$

If we take the second part of this recurrence relation, $nodes(h) = leaves(h) + nodes(h-1)$, and substitute in our earlier knowledge that $leaves(h) = 2^h$, it follows that when h is positive, $nodes(h) = 2^h + nodes(h-1)$. Similarly, for $h > 1$, we could

show that $nodes(h-1) = 2^{h-1} + nodes(h-2)$, so $nodes(h) = 2^h + 2^{h-1} + nodes(h-2)$. Continuing this substitution process until we reach the base case of $nodes(0) = 1$, we find that $nodes(h) = 2^h + 2^{h-1} + 2^{h-2} + \cdots + 4 + 2 + 1$. This sum can be simplified by taking advantage of the fact that multiplying it by 2 effectively shifts it all over by one position, that is, $2 \times nodes(h) = 2^{h+1} + 2^h + 2^{h-1} + \cdots + 8 + 4 + 2$. The payoff comes if we now subtract $nodes(h)$ from this:

$$\begin{array}{rrl} 2 \times nodes(h) = & 2^{h+1} + 2^h + 2^{h-1} + \cdots + 4 + 2 & \\ - \quad nodes(h) = & \quad 2^h + 2^{h-1} + \cdots + 4 + 2 + 1 & \\ \hline nodes(h) = & 2^{h+1} & - 1 \end{array}$$

Exercise 8.10

You can also use the recurrence relation together with induction to prove that $nodes(h) = 2^{h+1} - 1$. Do so.

Exercise 8.11

In many applications, binary trees aren't sufficient because we need more than two subtrees. An *m*-ary tree is a tree that is either empty or has a root and *m* subtrees, each of which is an *m*-ary tree. Generalize the previous results to *m*-ary trees.

Now suppose we have a binary tree that has *n* nodes total. What could the height of the tree be? In the worst-case scenario, each internal node has one nonempty child and one empty child. For example, imagine a tree where the left subtree of every node is empty (i.e., it branches only to the right). (This will happen with a binary search tree if the root at each level is always the smallest element.) In this case, the resulting tree is essentially just a list. Thus the maximum height of a tree with *n* nodes is $n - 1$.

What about the minimum height? We saw that a tree of height *h* can accommodate up to $2^{h+1} - 1$ nodes. On the other hand, if there are fewer than 2^h nodes, even a tree of height $h - 1$ would suffice to hold them all. Therefore, for *h* to be the minimum height of any tree with *n* nodes, we must have $2^h \leq n < 2^{h+1}$. If we take the logarithm base 2 of this inequality, we find that

$$h \leq \log_2(n) < h + 1$$

In other words, the minimum height of a tree with *n* nodes is $\lfloor \log_2(n) \rfloor$. (The expression $\lfloor \log_2(n) \rfloor$ is pronounced "the floor of log en." In general, the floor of a real number is the greatest integer that is less than or equal to that real number.)

Because searching for an element in a binary search tree amounts to finding a path from the root node to a node containing that element, we will clearly prefer

trees of *minimum height* for the given number of nodes. In some sense, such trees will be as short and bushy as possible. There are several ways to guarantee that a tree with n nodes has minimum height. One is given in Exercise 8.12. In Chapter 13 we'll consider the alternative of settling for trees that are no more than 4 times the minimum height.

We now have all of the mathematical tools we need to discuss why and when binary search trees are an improvement over straightforward lists. We will consider the procedure `in?` because it is somewhat simpler than `list-by-key`. However, similar considerations apply to the efficiency of `list-by-key`, just with more technical difficulties. Remember that with the `in?` procedure, we are only concerned with whether or not a given element is in a binary search tree, whereas with `list-by-key` we want to return the list of all records matching a given key.

Let's consider the time taken by the procedure `in?` on a tree of height h having n nodes. Searching for an element that isn't in the tree is equivalent to traveling from the root of the tree to one of its leaves. In this case, we will pass through at most $h + 1$ nodes. If we're searching for an element that is in the tree, we will encounter it somewhere along a path from the root to a leaf. Because the number of operations performed by `in?` is proportional to the number of nodes encountered, we conclude that in either case, searching for an element in the tree takes $O(h)$ time. If the tree has minimum height, this translates to $O(\log(n))$. In the worst case, where the height of the tree is $n - 1$, this becomes $O(n)$.

In Exercise 8.7, you wrote a procedure `list->bstree` that created a binary search tree from a list by successively inserting the elements into the tree. This procedure can lead to trees that are far from minimum height—surprisingly, the worst case occurs if the list is in sorted order. However, if you know the list is already in sorted order, you can do much better: Write a procedure `sorted-list->min-height-bstree` that creates a minimum height binary search tree from a sorted list of numbers. *Hint:* If the list has more than one element, split it into three parts: the middle element, the elements before the middle element, and the elements after. Construct the whole tree by making the appropriate recursive calls on these sublists and combining the results.

Using `sorted-list->min-height-bstree` and `inorder` (which constructs a sorted list from a binary search tree), write a procedure `optimize-bstree` that optimizes a binary search tree. That is, when given an arbitrary binary search tree, it should produce a minimum-height binary search tree containing the same nodes.

Exercise 8.14

Using `list->bstree` and `inorder`, write a procedure `sort` that sorts a given list.

Privacy Issues

How would you feel if you registered as a child at a chain ice-cream parlor for their "birthday club" by providing name, address, and birth date, only to find years later the Selective Service using that information to remind you of your legal obligation to register for the draft?

This case isn't a hypothetical one: It is one of many real examples of personal data voluntarily given to one organization for one purpose being used by a different organization for a different purpose.

Some very difficult social, ethical, and legal questions occur here. For example, did the ice-cream chain "own" the data it collected and hence have a right to sell it as it pleased? Did the the government step outside of the Bill of Rights restrictions on indiscriminate "dragnet" searches? Did the social good of catching draft evaders justify the means? How about if it had been tax or welfare cheats or fathers delinquent in paying child support? (All of the above have been tracked by computerized matching of records.) Should the computing professionals who wrote the "matching" program have refused to do so?

The material we have covered on binary search trees may help you to define efficient structures to store and retrieve data. However, because many information storage and retrieval systems are used to store personal information, we urge you to also take the following to heart when and if you undertake such a design. The Code of Ethics and Professional Conduct of the Association for Computing Machinery, or ACM (which is the major computing professional society) contains as General Moral Imperative 1.7:

Respect the privacy of others

Computing and communication technology enables the collection and exchange of personal information on a scale unprecedented in the history of civilization. Thus there is increased potential for violating the privacy of individuals and groups. It is the responsibility of professionals to maintain the privacy and integrity of data describing individuals. This includes taking precautions to ensure the accuracy of data, as well as protecting it from unauthorized access or accidental disclosure to inappropriate individuals. Furthermore, procedures must be established to allow individuals to review their records and correct inaccuracies.

(Continued)

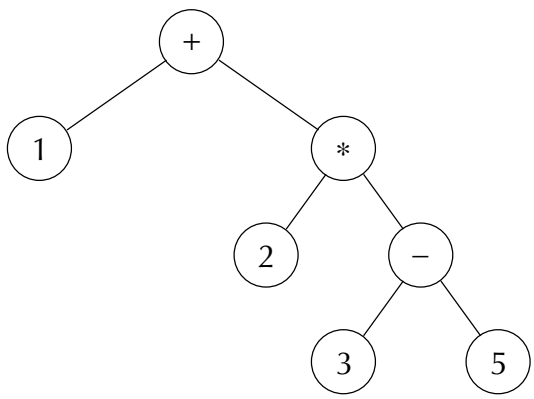
Privacy Issues (Continued)

This imperative implies that only the necessary amount of personal information be collected in a system, that retention and disposal periods for that information be clearly defined and enforced, and that personal information gathered for a specific purpose not be used for other purposes without consent of the individual(s). These principles apply to electronic communications, including electronic mail, and prohibit procedures that capture or monitor electronic user data, including messages, without the permission of users or bona fide authorization related to system operation and maintenance. User data observed during the normal duties of system operation and maintenance must be treated with strictest confidentiality, except in cases where it is evidence for the violation of law, organizational regulations, or this Code. In these cases, the nature or contents of that information must be disclosed only to proper authorities.

8.3 Expression Trees

So far, we've used binary trees and binary search trees as a way of storing a collection of numbers or records. What makes these trees different from lists is the way we can access the elements. A list has one special element, the first element, and all the rest of the elements are clumped together into another list. Binary trees also have a special element, the root, but they divide the rest of the elements into *two* subtrees, instead of just one, which gives a hierarchical structure that is useful in many different settings. In this section we'll look at another kind of tree that uses this hierarchical structure to represent arithmetical expressions. In these trees, the way a tree is structured indicates the operands for each operation in the expression.

Consider an arithmetic expression, such as the one we'd write in Scheme notation as `(+ 1 (* 2 (- 3 5)))`. We can think of this as being a tree-like structure with numbers at the leaves and operators at the other nodes:



Such a structure is often called an *expression tree*. As we did with binary trees, we can define an expression tree more precisely:

> **Expression tree:** An expression tree is either a number or it has three parts, the name of an operator, a left operand and a right operand. Both the left and right operands are themselves expression trees.

There are several things to notice about this definition:

- We are restricting ourselves to expressions that have *binary operators* (i.e., operators that take exactly two operands).
- We are also restricting ourselves to having numbers as our *atomic expressions*. In general, expression trees also include other kinds of constants and variable names as well.
- There is nothing in the definition that says an expression tree must be written in *prefix* order, that is, with the name of the operator preceding the two operands. Indeed, most people would find *infix* order more natural. An infix expression has the name of the operator in between the two operands.

How do we implement expression trees? We will do it in much the same way that we implemented binary trees, except that we will follow the idea of the last note in the preceeding list and list the parts of an expression in infix order:

```
(define make-constant
  (lambda (x) x))

(define constant? number?)

(define make-expr
  (lambda (left-operand operator right-operand)
    (list left-operand operator right-operand)))

(define operator cadr)

(define left-operand car)

(define right-operand caddr)
```

Now that we have a way of creating expressions, we can write the procedures necessary to evaluate them using the definition to help us decide how to structure our code. To buy ourselves some flexibility, we'll use a procedure called `look-up-value` to map an operator name into the corresponding operator procedure. Then the main `evaluate` procedure just needs to apply that operator procedure to the values of the operands:

```
(define evaluate
  (lambda (expr)
    (cond ((constant? expr) expr)
          (else ((look-up-value (operator expr))
                 (evaluate (left-operand expr))
                 (evaluate (right-operand expr)))))))

(define look-up-value
  (lambda (name)
    (cond ((equal? name '+) +)
          ((equal? name '*) *)
          ((equal? name '-) -)
          ((equal? name '/) /)
          (else (error "Unrecognized name" name)))))
```

With these definitions, we would have the following interaction:

```
(evaluate '(1 + (2 * (3 - 5))))
-3
```

Exercise 8.15

In the preceding example, we've "cheated" by using a quoted list as the expression to evaluate. This method relied on our knowledge of the representation of expression trees. How could the example be rewritten to use the constructors to form the expression?

We can do more with expression trees than just finding their values. For example, we could modify the procedure for doing a postorder traversal of a binary search tree so that it works on *expression trees* instead. In this case, our base case will be when we have a constant, or a leaf, instead of an empty tree:

```
(define post-order
  (lambda (tree)
    (define post-order-onto
      (lambda (tree list)
        (if (constant? tree)
            (cons tree list)
            (post-order-onto (left-operand tree)
                             (post-order-onto
                              (right-operand tree)
                              (cons (operator tree) list))))))
    (post-order-onto tree '())))
```

If we do a postorder traversal of the last tree shown, we get:

```
(post-order '(1 + (2 * (3 - 5))))
(1 2 3 5 - * +)
```

This result is exactly the sequence of keys that you would need to punch into a Hewlett-Packard calculator in order to evaluate the expression. Such an expression is said to be a *postfix* expression.

Exercise 8.16

Define a procedure for determining which operators are used in an expression.

Exercise 8.17

Define a procedure for counting how many operations an expression contains.

Note that all of the operators in our expressions were binary operators, and thus we needed nodes with two children to represent them; we say the operator nodes all have *degree* 2. If we had operators that took m expressions instead of just two, we would need nodes with degree m (i.e., trees that have m subtrees).

The kind of tree we've been using in this section differs subtly from the binary and m-ary trees we saw earlier in the chapter. In those *positional trees*, it was possible to have a node with a right child but no left child, for example. In the *ordered trees* we're using for expressions, on the other hand, there can't be a second operand unless there is a first operand. Other kinds of trees exist as well, for example, trees in which no distinction is made among the children—none is first or second, left or right; they are all just children. Most of the techniques and terminology carry over for all kinds of trees.

8.4 An Application: Automated Phone Books

Have you ever called a university's information service to get the phone number of a friend and, instead of talking to a human operator, found yourself following instructions given by a computer? Perhaps you were even able to look up the friend's phone number using the numbers on the telephone keypad. Such automated telephone directory systems are becoming more common. In this section we will explore one version of how such a directory might be implemented.

In this version, a user looks up the telephone number of a person by spelling the person's name using the numbers on the telephone keypad. When the user has entered enough numbers to identify the person, the system returns the telephone

number. Can we rephrase this problem in a form that we can treat using Scheme? Suppose that we have a collection of pairs, where each pair consists of a person's name and phone number. How could we store the pairs so that we can easily retrieve a person's phone number by giving the sequence of digits (from 2 to 9) corresponding to the name? Perhaps our system might do even more: For example, we could have our program repeatedly take input from the user until the identity of the desired person is determined, at which point the person's name and phone number is given.

Notice the similarity between this problem and the video catalog problem considered in Section 8.1. There we wanted to store the videos in a way that allowed us to efficiently find all videos with a given director. Our desire to implement binary search led us to develop the binary search tree ADT. Searching was accomplished by choosing the correct child of each subtree and therefore amounted to finding the path from the root node to the node storing the desired value.

We are also searching for things with the automated phone book, but the difference is the method of retrieval: we want to retrieve a phone number by successively giving the digits corresponding to the letters in the person's name. How should we structure our data in a way that facilitates this type of retrieval? Suppose we use a tree to store the phone numbers. What type of tree would lend itself to such a search?

If we are going to search by the sequence of digits corresponding to the person's name, then these digits could describe the path from the root node to the node storing the desired value. Each new digit would get us closer to our goal. The easiest way to accomplish this is to have the subtrees of a given node labeled (indexed) by the digits themselves. Then the sequence of digits would exactly describe the path to the desired node because we would always choose the subtree labeled by the next digit in our sequence. Such a tree is called a *trie*. This name is derived from the word re*trie*val, though the conventional pronunciation has become "try" rather than the logical but confusing "tree." More precisely,

> **Trie:** A trie is either empty or it consists of two parts: a list of root values and a list of subtries, which are indexed by labels. Each subtrie is itself a trie.

Because we have the eight digits from 2 to 9 as labels in our example, our tries will be 8-ary trees. The first child of a node will be implicitly labeled as the "2" child, the second as the "3" child, etc. In other words, the digits the user enters describe a path starting from the root node. If the user types a 2, we move to the first child of the root node. If the user types a 3 next, we then move to the second child of that node.

The values stored at a particular node are those corresponding to the path from the root of the trie to that node. If anyone had an empty name (i.e., zero letters long), that name and number would be stored on the root node of the trie. Anyone with the one-letter name A, B, or C would be on the first child of the root (the one for the digit 2 on the phone keypad, which is also labeled ABC). Anyone with the

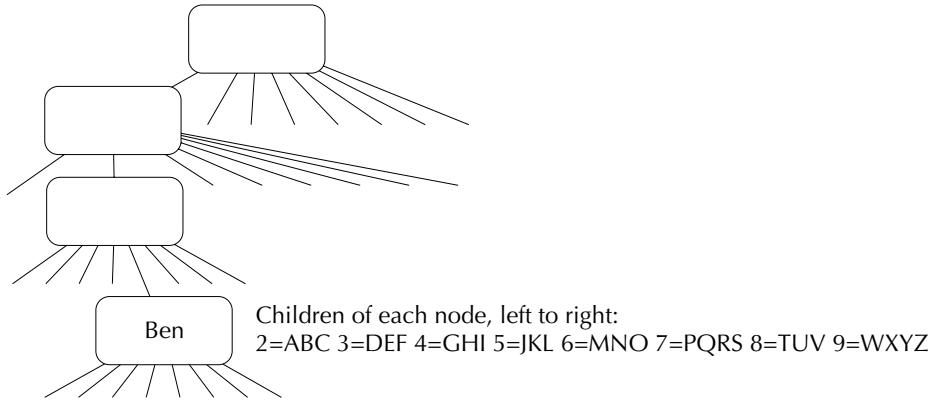


Figure 8.1 An example phone trie, with Ben's position indicated

one-letter name D, E, or F would be on the second child of the root. Anyone with any of the two-letter names Ad, Ae, Af, Bd, Be, Bf, Cd, Ce, or Cf would be on the second child of the first child of the root. For example, the trie in Figure 8.1 shows where the name and number of someone named Ben would be stored.

Note that a given node may or may not store a value: In our example, the nodes encountered on the way to Ben's node don't have any values because no one has an empty name, the one-letter name A, B, or C, or any of the 9 two-letter names listed above. Not all the values need be at leaf nodes, however. For example, Ben's name corresponds on a phone to the digits 2-3-5. However, these are also the first three digits in the name Benjamin, and in fact even the first three digits in the name Adonis, because B and A share a phone digit, as do E and D and also N and O. Therefore, the node in our trie that stores the value Ben may also be encountered along a path to a deeper node that stores Benjamin or Adonis.

We must also allow more than one value to be stored at a given node, because, for example, Jim and Kim would be specified by the same sequence of digits (5-4-6) on the telephone. Therefore, we have a *list* of root values in our definition.

How can we implement tries? As described above, we will implement them as 8-ary trees, where every tree has *exactly* eight subtrees, even if some (or all) of them are empty. These subtrees correspond to the digits 2 through 9, which have letters on a phone keypad. We call these digits 2 through 9 the "labels" of the subtrees and define a selector called `subtrie-with-label` that returns the subtrie of a nonempty trie that corresponds to a given label:

```
(define make-empty-trie
  (lambda () '()))
```

```
(define make-nonempty-trie
  (lambda (root-values ordered-subtries)
    (list root-values ordered-subtries)))

(define empty-trie? null?)

(define root-values car)

(define subtries cadr)

(define subtrie-with-label
  (lambda (trie label)
    (list-ref (subtries trie) (- label 2))))
```

Note that the constructor `make-nonempty-trie` assumes that the subtries are given to it in order (including possibly some empty subtries). Constructing a specific phone trie is a somewhat difficult task that we will consider later in this section. In fact, we will write a procedure `values->trie` that takes a list of values (people's names and phone numbers) and returns the trie containing those values. Note also that the procedure `subtrie-with-label` must subtract 2 from the label because list convention refers to the first element (corresponding to the digit 2) as element number zero.

The values in our automated phone book are the phone numbers of various people. In order to store the person's name and phone number together, we create a simple record-structured ADT called *person*:

```
(define make-person
  (lambda (name phone-number)
    (list name phone-number)))

(define name car)

(define phone-number cadr)
```

How do we construct the trie itself? As we said in the preceeding, we will do this later in the section by writing a procedure `values->trie` that creates a trie from a list of values. For example, a definition of the form:

```
(define phone-trie
  (values->trie (list (make-person 'lindt          7483)
                      (make-person 'cadbury        7464)
                      (make-person 'wilbur         7466)
                      (make-person 'hershey        7482)
```

```
(make-person 'spruengli         7009)
(make-person 'merkens           7469)
(make-person 'baker             7465)
(make-person 'ghiradelli        7476)
(make-person 'tobler            7481)
(make-person 'suchard           7654)
(make-person 'callebaut         7480)
(make-person 'ritter            7479)
(make-person 'maillard          7477)
(make-person 'see               7463)
(make-person 'perugina          7007))))
```

will define `phone-trie` to be the trie containing the given people, which can then be used to look up phone numbers. You can work on other exercises involving tries before we write `values->trie` because we've included an alternate definition on the web site for this book, which simply defines `phone-trie` as a quoted list.

Using what we have already developed, we can implement a simple automated phone book as follows:

```
(define look-up-with-menu
  (lambda (phone-trie)
    (menu)
    (look-up-phone-number phone-trie)))

(define menu
  (lambda ()
    (newline)
    (display "Enter the name, one digit at a time.")
    (newline)
    (display "Indicate you are done by 0.")
    (newline)))

(define look-up-phone-number
  (lambda (phone-trie)
    (newline)
    (if (empty-trie? phone-trie)
        (display "Sorry we can't find that name.")
        (let ((user-input (read)))
          (if (= user-input 0)
              (display-phone-numbers (root-values phone-trie))
              (look-up-phone-number (subtrie-with-label
                                     phone-trie
                                     user-input)))))))
```

```
(define display-phone-numbers
  (lambda (people)
    (define display-loop
      (lambda (people)
        (cond ((null? people) 'done)
              (else (newline)
                    (display (name (car people)))
                    (display "'s phone number is ")
                    (display (phone-number (car people)))
                    (display-loop (cdr people))))))
    (if (null? people)
        (display "Sorry we can't find that name.")
        (display-loop people))))
```

Here is how you could use `look-up-with-menu` to look up the telephone number of Spruengli, for example:

```
(look-up-with-menu phone-trie)
Enter the name, one digit at a time.
Indicate you are done with 0.

7
7
7
8
3
6
4
5
4
0
spruengli's phone number is 7009
```

This method is certainly progress, but it is also somewhat clunky. After all, in our example Spruengli is already determined by the first two digits (7 and 7). It seems silly to require the user to enter more digits than are necessary to specify the desired person. We could make our program better if we had a procedure that tells us when we have exactly one remaining value in a trie, and another procedure that returns that value.

We can write more general versions of both of these procedures; one would return the number of values in a trie and the other the list of values. Notice that these two procedures are quite similar. In either case you can compute the answer by taking

the number of values (respectively, the list of values) at the root node and adding that to the number of values (respectively, the list of values) in each of the subtries. The difference is that in the former case you add the numbers by regular addition, whereas in the latter case you add by appending the various lists.

Exercise 8.18

Write the procedure `number-in-trie` that calculates the total number of values in a trie. *Hint:* In the general case, you can compute the list of numbers in the various subtries by using `number-in-trie` in conjunction with the built-in Scheme procedure `map`. The total number of values in all the subtries can then be gotten by applying the `sum` procedure from Section 7.3. Of course, you have to take into account the values that are at the root node of the trie.

Exercise 8.19

Write the procedure `values-in-trie` that returns the list of all values stored in a given trie. It should be very similar in form to `number-in-trie`. You may find your solution to Exercise 7.5 on page 173 useful. In fact, if you rewrote `number-in-trie` to use Exercise 7.5's solution in place of `sum`, `values-in-trie` would be nearly identical in form to `number-in-trie`.

Exercise 8.20

Let's use these procedures to improve what is done in the procedure `look-up-phone-number`.

a. Use `number-in-trie` to determine if there are fewer than two values in *phone-trie* and immediately report the appropriate answer if so, using `values-in-trie` and `display-phone-numbers`.

b. Further modify `look-up-phone-number` so that if the user enters 1, the names of all the people in the current trie will be reported, but the procedure `look-up-phone-number` will continue to read input from the user. You will also want to make appropriate changes to `menu`.

We now confront the question of how these tries we have been working with can be created in the first place. As we indicated earlier, we will write a procedure `values->trie` that will take a list of values (i.e., people) and will return the trie containing them. First some remarks on vocabulary: Because we have so many different data types floating around (and we will soon define one more), we need to be careful about the words we use to describe them. A *value* is a single data item (in

our case a person, that is, name and phone number) being stored in a trie. A *label* is in our case a digit from 2 to 9; it is what is used to select a subtrie. Plurals will always indicate lists; for example, *values* will mean a list of values and *labels* will mean a list of labels. This may seem trivial, but it will prove very useful for understanding the meanings of the following procedures and their parameters.

Exercise 8.21

Write a procedure `letter->number` that takes a letter (i.e., a one-letter symbol) and returns the number corresponding to it on the telephone keypad. For *q* and *z* use 7 and 9, respectively. *Hint:* The easiest way to do this exercise is to use a `cond` together with the list membership predicate `member` we introduced in the previous chapter.

Exercise 8.22

To break a symbol up into a list of one-character symbols, we need to use some features of Scheme that we'd rather not talk about just now. The following `explode-symbol` procedure uses these magic features of Scheme so that `(explode-symbol 'ritter)` would evaluate to the list of one-letter symbols `(r i t t e r)`, for example:

```
(define explode-symbol
  (lambda (sym)
    (map string->symbol
         (map string
              (string->list (symbol->string sym))))))
```

Use this together with `letter->number` to write a procedure `name->labels` that takes a name (symbol) and returns the list of numbers corresponding to the name. You should see the following interaction:

```
(name->labels 'ritter)
(7 4 8 8 3 7)
```

To make a trie from a list of values, we will need to work with the labels associated with each of the values. One way is to define a simple ADT called *labeled-value* that packages these together. This could be done as follows:

```
(define make-labeled-value
  (lambda (labels value)
    (list labels value)))
```

```
(define labels car)

(define value cadr)
```

Because we will use this abstraction to construct tries, we will need some procedures that allow us to manipulate labeled values.

Exercise 8.23

Write a procedure `empty-labels?` that takes a labeled value and returns true if and only if its list of labels is empty.

Exercise 8.24

Write a procedure `first-label` that takes a labeled value and returns the first label in its list of labels.

Exercise 8.25

Write a procedure `strip-one-label` that takes a labeled value and returns the labeled value with one label removed. For example, you would have the following interaction:

```
(define labeled-ritter
  (make-labeled-value '(7 4 8 8 3 7)
                      (make-person 'ritter 7479)))

(labels (strip-one-label labeled-ritter))
(4 8 8 3 7)

(name (value (strip-one-label labeled-ritter)))
ritter

(phone-number (value (strip-one-label labeled-ritter)))
7479
```

Exercise 8.26

Write a procedure `value->labeled-value` that takes a value (person) and returns the labeled value corresponding to it. You must of course use the procedure `name->labels`.

We can now write `values->trie` in terms of a yet to be written procedure that operates on labeled values:

```
(define values->trie
  (lambda (values)
    (labeled-values->trie (map value->labeled-value
                               values))))
```

How do we write `labeled-values->trie`? The argument to this procedure is a list of labeled values, and we must clearly use the labels in the trie construction. If a given labeled value has the empty list of labels (in other words, we have gotten to the point in the recursion where all of the labels have been used), the associated value should be one of the values at the trie's root node. We can easily isolate these labeled values using the `filter` procedure from Section 7.3, as in:

```
(filter empty-labels? labeled-values)
```

We can similarly isolate those with nonempty labels, which belong in the subtries; the first label of each labeled value determines which subtrie it goes in.

Exercise 8.27

Write a procedure `values-with-first-label` that takes a list of labeled values and a label and returns a list of those labeled values that have the given first label, but with that first label removed. You may assume that none of the labeled values has an empty list of labels. Thus, the call `(values-with-first-label labeled-values 4)` should return the list of those labeled values in `labeled-values` with a first label of `4`, but with the `4` removed from the front of their lists of labels. (This would only be legal assuming each labeled value in `labeled-values` has a nonempty list of labels.) Stripping off the first label makes sense because it was used to select out the relevant labeled values, which will form one subtrie of the overall trie. Within the subtrie, that first label no longer plays a role.

Exercise 8.28

Using the procedure `values-with-first-label`, write a procedure `categorize-by-first-label` that takes a list of labeled values, each with a nonempty list of labels, and returns a *list of lists* of labeled values. The first list in the list of lists should contain all those labeled values with first label 2, the next list, those that start with 3, etc. (If there are no labeled values with a particular first label, the corresponding list will be empty. There will always be eight lists, one for each possible first label,

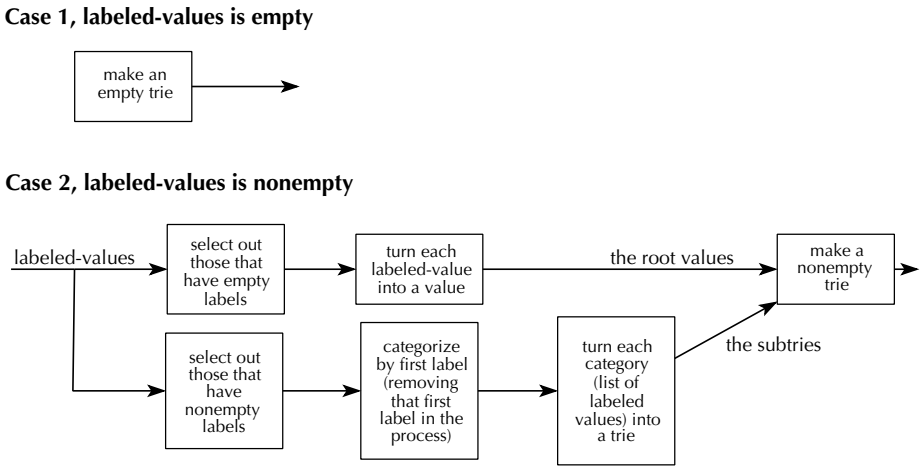


Figure 8.2 The design of the `labeled-values->trie` procedure

ranging from 2 to 9.) Each labeled value should have its first label stripped off, which `values-with-first-label` takes care of. (Thus the labeled values in the first list, for example, no longer have the label of 2 on the front.)

Exercise 8.29

Finally, write the procedure `labeled-values->trie`. If the list of labeled values is empty, you can just use `make-empty-trie`. On the other hand, if the list is not empty, you can isolate those labeled values with empty labels and those with nonempty labels, as indicated above. You can turn the ones with empty labels into the root values by applying `value` to each of them. You can turn the ones with nonempty labels into the subtries by using `categorize-by-first-label`, `map`, and `labeled-values->trie`. Once you have the root values and the subtries, you can use `make-nonempty-trie` to create the trie. Figure 8.2 illustrates this design.

Review Problems

Exercise 8.30

Fill in the following definition of the procedure `successor-of-in-or`. This procedure should take three arguments: a value (*value*), a binary search tree (*bst*), and a value to return if no element of the tree is larger than *value* (*if-none*). If there is any element, x, of *bst* such that $x > value$, the smallest such element should be returned. Otherwise, *if-none* should be returned.

```
(define successor-of-in-or
  (lambda (value bst if-none)
    (cond ((empty-tree? bst)
           _____________________)
          ((<= (root bst) value)
           (successor-of-in-or _____________________
                               _____________________
                               _____________________))
          (else
           (successor-of-in-or _____________________
                               _____________________
                               _____________________)))))
```

Exercise 8.31

Write a procedure that takes as arguments a binary search tree of numbers, a lower bound, and an upper bound and counts how many elements of the tree are greater than or equal to the lower bound and less than or equal to the upper bound. Assume that the tree may contain duplicate elements. Make sure your procedure doesn't examine more of the tree than is necessary.

Exercise 8.32

Write a procedure that takes as arguments a binary search tree of numbers, a lower bound, and an upper bound and returns an ordered list of those elements of the tree that are greater than or equal to the lower bound and less than or equal to the upper bound. Assume that the tree may contain duplicate elements. Use the technique of an "onto" parameter to avoid unnecessary appending of lists, and make sure your procedure doesn't examine more of the tree than is necessary.

Chapter Inventory

Vocabulary

divide and conquer
binary search
root
subtree
strictly ordered
node
leaf
internal node
child
parent
tree traversal
preorder

in-order
postorder
depth
length of a path
height
complete tree
recurrence relation
⌊ ⌋ (floor)
minimum height binary tree
The Code of Ethics and Professional Conduct
Association for Computing Machinery (ACM)
binary operator
atomic expression
prefix
infix
postfix
degree
positional tree
ordered tree

Slogans

The one-layer data structure principle

Abstract Data Types

binary search tree
binary tree
expression tree
trie
person
labeled value

New Predefined Scheme Names

`append`

Scheme Names Defined in This Chapter

`make-empty-tree`
`make-nonempty-tree`
`empty-tree?`
`root`
`left-subtree`
`right-subtree`
`in?`
`minimum`
`number-of-nodes`
`list-by-key`
`preorder`
`preorder-onto`
`inorder`
`insert`
`list->bstree`
`sorted-list->min-height-bstree`
`optimize-bstree`
`sort`
`make-constant`
`constant?`
`make-expr`
`operator`
`left-operand`
`right-operand`
`look-up-value`
`evaluate`
`post-order`
`make-empty-trie`
`make-nonempty-trie`
`empty-trie?`
`root-values`
`subtries`
`subtrie-with-label`
`values->trie`

```
make-person
name
phone-number
phone-trie
look-up-with-menu
menu
look-up-phone-number
display-phone-numbers
number-in-trie
values-in-trie
letter->number
explode-symbol
name->labels
make-labeled-value
labels
value
empty-labels?
first-label
strip-one-label
value->labeled-value
labeled-values->trie
values-with-first-label
categorize-by-first-label
successor-of-in-or
```

Sidebars

Privacy Issues

Notes

As with Θ in Chapter 4, the conventional definition of O allows any number of exceptions up to some cutoff point, rather than finitely many exceptions as we do. Again, so long as n is restricted to the nonnegative integers, our definition is equivalent.

The example of personal information divulged to an ice-cream parlor "birthday club" winding up in the hands of the Selective Service is reported in [24].

The ACM Code of Ethics and Professional Conduct can be found in [17]; a set of illustrative case studies accompanies it in [4].

Regarding the pronunciation of "trie," we've had to take Aho and Ullman's word for it—none of us can recall ever having heard "trie" said aloud. Aho and Ullman should know, though and they write on page 217 of their *Foundations of Computer Science* [3] that "it was originally intended to be pronounced 'tree.' Fortunately, common parlance has switched to the distinguishing pronunciation 'try.'"