Part III Data Structures

Introduction

Sets are as fundamental to computer science as they are to mathematics. Whereas mathematical sets are unchanging, the sets manipulated by algorithms can grow, shrink, or otherwise change over time. We call such sets ***dynamic***. The next four chapters present some basic techniques for representing finite dynamic sets and manipulating them on a computer.

Algorithms may require several types of operations to be performed on sets. For example, many algorithms need only the ability to insert elements into, delete elements from, and test membership in a set. We call a dynamic set that supports these operations a ***dictionary***. Other algorithms require more complicated operations. For example, min-priority queues, which Chapter 6 introduced in the context of the heap data structure, support the operations of inserting an element into and extracting the smallest element from a set. The best way to implement a dynamic set depends upon the operations that you need to support.

Elements of a dynamic set

In a typical implementation of a dynamic set, each element is represented by an object whose attributes can be examined and manipulated given a pointer to the object. Some kinds of dynamic sets assume that one of the object's attributes is an identifying ***key***. If the keys are all different, we can think of the dynamic set as being a set of key values. The object may contain ***satellite data***, which are carried around in other object attributes but are otherwise unused by the set

implementation. It may also have attributes that are manipulated by the set operations. These attributes may contain data or pointers to other objects in the set.

Some dynamic sets presuppose that the keys are drawn from a totally ordered set, such as the real numbers, or the set of all words under the usual alphabetic ordering. A total ordering allows us to define the minimum element of the set, for example, or to speak of the next element larger than a given element in a set.

Operations on dynamic sets

Operations on a dynamic set can be grouped into two categories: ***queries***, which simply return information about the set, and ***modifying operations***, which change the set. Here is a list of typical operations. Any specific application will usually require only a few of these to be implemented.

SEARCH(S, k)
: A query that, given a set S and a key value k, returns a pointer x to an element in S such that $x.key = k$, or NIL if no such element belongs to S.

INSERT(S, x)
: A modifying operation that adds the element pointed to by x to the set S. We usually assume that any attributes in element x needed by the set implementation have already been initialized.

DELETE(S, x)
: A modifying operation that, given a pointer x to an element in the set S, removes x from S. (Note that this operation takes a pointer to an element x, not a key value.)

MINIMUM(S) and MAXIMUM(S)
: Queries on a totally ordered set S that return a pointer to the element of S with the smallest (for MINIMUM) or largest (for MAXIMUM) key.

SUCCESSOR(S, x)

A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or NIL if x is the maximum element.

PREDECESSOR(S, x)
A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next smaller element in S, or NIL if x is the minimum element.

In some situations, we can extend the queries SUCCESSOR and PREDECESSOR so that they apply to sets with nondistinct keys. For a set on n keys, the normal presumption is that a call to MINIMUM followed by $n - 1$ calls to SUCCESSOR enumerates the elements in the set in sorted order.

We usually measure the time taken to execute a set operation in terms of the size of the set. For example, Chapter 13 describes a data structure that can support any of the operations listed above on a set of size n in $O(\lg n)$ time.

Of course, you can always choose to implement a dynamic set with an array. The advantage of doing so is that the algorithms for the dynamic-set operations are simple. The downside, however, is that many of these operations have a worst-case running time of $\Theta(n)$. If the array is not sorted, INSERT and DELETE can take $\Theta(1)$ time, but the remaining operations take $\Theta(n)$ time. If instead the array is maintained in sorted order, then MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR take $\Theta(1)$ time; SEARCH takes $O(\lg n)$ time if implemented with binary search; but INSERT and DELETE take $\Theta(n)$ time in the worst case. The data structures studied in this part improve on the array implementation for many of the dynamic-set operations.

Overview of Part III

Chapters 10–13 describe several data structures that we can use to implement dynamic sets. We'll use many of these data structures later to construct efficient algorithms for a variety of problems. We already saw another important data structure—the heap—in Chapter 6.

Chapter 10 presents the essentials of working with simple data structures such as arrays, matrices, stacks, queues, linked lists, and rooted trees. If you have taken an introductory programming course, then much of this material should be familiar to you.

Chapter 11 introduces hash tables, a widely used data structure supporting the dictionary operations INSERT, DELETE, and SEARCH. In the worst case, hash tables require $\Theta(n)$ time to perform a SEARCH operation, but the expected time for hash-table operations is $O(1)$. We rely on probability to analyze hash-table operations, but you can understand how the operations work even without probability.

Binary search trees, which are covered in Chapter 12, support all the dynamic-set operations listed above. In the worst case, each operation takes $\Theta(n)$ time on a tree with n elements. Binary search trees serve as the basis for many other data structures.

Chapter 13 introduces red-black trees, which are a variant of binary search trees. Unlike ordinary binary search trees, red-black trees are guaranteed to perform well: operations take $O(\lg n)$ time in the worst case. A red-black tree is a balanced search tree. Chapter 18 in Part V presents another kind of balanced search tree, called a B-tree. Although the mechanics of red-black trees are somewhat intricate, you can glean most of their properties from the chapter without studying the mechanics in detail. Nevertheless, you probably will find walking through the code to be instructive.

10 Elementary Data Structures

In this chapter, we examine the representation of dynamic sets by simple data structures that use pointers. Although you can construct many complex data structures using pointers, we present only the rudimentary ones: arrays, matrices, stacks, queues, linked lists, and rooted trees.

10.1 Simple array-based data structures: arrays, matrices, stacks, queues

10.1.1 Arrays

We assume that, as in most programming languages, an array is stored as a contiguous sequence of bytes in memory. If the first element of an array has index s (for example, in an array with 1-origin indexing, $s = 1$), the array starts at memory address a, and each array element occupies b bytes, then the ith element occupies bytes $a + b(i - s)$ through $a + b(i - s + 1) - 1$. Since most of the arrays in this book are indexed starting at 1, and a few starting at 0, we can simplify these formulas a little. When $s = 1$, the ith element occupies bytes $a + b(i - 1)$ through $a + bi - 1$, and when $s = 0$, the ith element occupies bytes $a + bi$ through $a + b(i + 1) - 1$. Assuming that the computer can access all memory locations in the same amount of time (as in the RAM model described in Section 2.2), it takes constant time to access any array element, regardless of the index.

Most programming languages require each element of a particular array to be the same size. If the elements of a given array might occupy different numbers of bytes, then the above formulas fail to apply, since

the element size b is not a constant. In such cases, the array elements are usually objects of varying sizes, and what actually appears in each array element is a pointer to the object. The number of bytes occupied by a pointer is typically the same, no matter what the pointer references, so that to access an object in an array, the above formulas give the address of the pointer to the object and then the pointer must be followed to access the object itself.

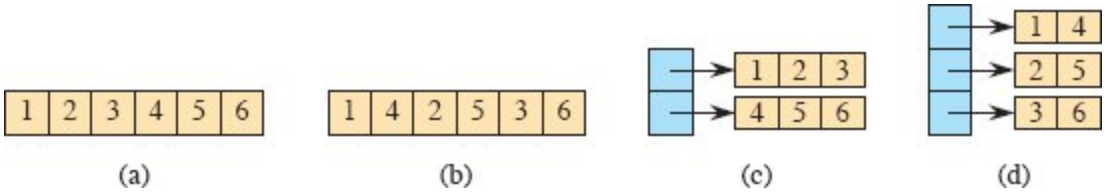

Figure 10.1 Four ways to store the 2 × 3 matrix M from equation (10.1). **(a)** In row-major order, in a single array. **(b)** In column-major order, in a single array. **(c)** In row-major order, with one array per row (tan) and a single array (blue) of pointers to the row arrays. **(d)** In column-major order, with one array per column (tan) and a single array (blue) of pointers to the column arrays.

10.1.2 Matrices

We typically represent a matrix or two-dimensional array by one or more one-dimensional arrays. The two most common ways to store a matrix are row-major and column-major order. Let's consider an $m \times n$ matrix—a matrix with m rows and n columns. In ***row-major order***, the matrix is stored row by row, and in ***column-major order***, the matrix is stored column by column. For example, consider the 2 × 3 matrix

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}. \tag{10.1}$$

Row-major order stores the two rows 1 2 3 and 4 5 6, whereas column-major order stores the three columns 1 4; 2 5; and 3 6.

Parts (a) and (b) of Figure 10.1 show how to store this matrix using a single one-dimensional array. It's stored in row-major order in part (a) and in column-major order in part (b). If the rows, columns, and the single array all are indexed starting at s, then $M[i, j]$—the element in row i and column j—is at array index $s + (n(i - s)) + (j - s)$ with row-

major order and $s + (m(j - s)) + (i - s)$ with column-major order. When $s = 1$, the single-array indices are $n(i - 1) + j$ with row-major order and $i + m(j - 1)$ with column-major order. When $s = 0$, the single-array indices are simpler: $ni + j$ with row-major order and $i + mj$ with column-major order. For the example matrix M with 1-origin indexing, element $M[2, 1]$ is stored at index $3(2 - 1) + 1 = 4$ in the single array using row-major order and at index $2 + 2(1 - 1) = 2$ using column-major order.

Parts (c) and (d) of Figure 10.1 show multiple-array strategies for storing the example matrix. In part (c), each row is stored in its own array of length n, shown in tan. Another array, with m elements, shown in blue, points to the m row arrays. If we call the blue array A, then $A[i]$ points to the array storing the entries for row i of M, and array element $A[i][j]$ stores matrix element $M[i, j]$. Part (d) shows the column-major version of the multiple-array representation, with n arrays, each of length m, representing the n columns. Matrix element $M[i, j]$ is stored in array element $A[j][i]$.

Single-array representations are typically more efficient on modern machines than multiple-array representations. But multiple-array representations can sometimes be more flexible, for example, allowing for "ragged arrays," in which the rows in the row-major version may have different lengths, or symmetrically for the column-major version, where columns may have different lengths.

Occasionally, other schemes are used to store matrices. In the ***block representation***, the matrix is divided into blocks, and each block is stored contiguously. For example, a 4×4 matrix that is divided into 2×2 blocks, such as

$$\left(\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right)$$

might be stored in a single array in the order $\langle 1, 2, 5, 6, 3, 4, 7, 8, 9, 10, 13, 14, 11, 12, 15, 16 \rangle$.

10.1.3 Stacks and queues

Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified. In a ***stack***, the element deleted from the set is the one most recently inserted: the stack implements a ***last-in, first-out***, or ***LIFO***, policy. Similarly, in a ***queue***, the element deleted is always the one that has been in the set for the longest time: the queue implements a ***first-in, first-out***, or ***FIFO***, policy. There are several efficient ways to implement stacks and queues on a computer. Here, you will see how to use an array with attributes to store them.

Stacks

The INSERT operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP. These names are allusions to physical stacks, such as the spring-loaded stacks of plates used in cafeterias. The order in which plates are popped from the stack is the reverse of the order in which they were pushed onto the stack, since only the top plate is accessible.

Figure 10.2 shows how to implement a stack of at most n elements with an array $S[1 : n]$. The stack has attributes $S.top$, indexing the most recently inserted element, and $S.size$, equaling the size n of the array. The stack consists of elements $S[1 : S.top]$, where $S[1]$ is the element at the bottom of the stack and $S[S.top]$ is the element at the top.

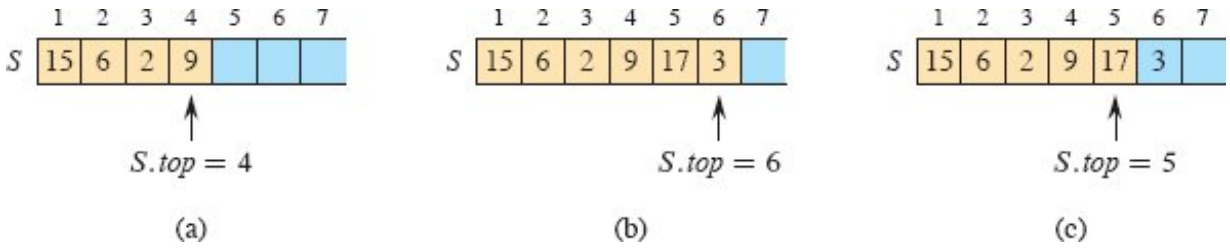

Figure 10.2 An array implementation of a stack S. Stack elements appear only in the tan positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH(S, 17) and PUSH(S, 3). **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack. The top is element 17.

When $S.top = 0$, the stack contains no elements and is ***empty***. We can test whether the stack is empty with the query operation STACK-

EMPTY. Upon an attempt to pop an empty stack, the stack ***underflows***, which is normally an error. If $S.top$ exceeds $S.size$, the stack ***overflows***.

The procedures STACK-EMPTY, PUSH, and POP implement each of the stack operations with just a few lines of code. Figure 10.2 shows the effects of the modifying operations PUSH and POP. Each of the three stack operations takes $O(1)$ time.

```
STACK-EMPTY(S)

1 if S.top == 0
2     return TRUE
3 else return FALSE

PUSH(S, x)

1 if S.top == S.size
2     error "overflow"
3 else S.top = S.top + 1
4     S[S.top] = x

POP(S)

1 if STACK-EMPTY(S)
2     error "underflow"
3 else S.top = S.top – 1
4     return S[S.top + 1]
```

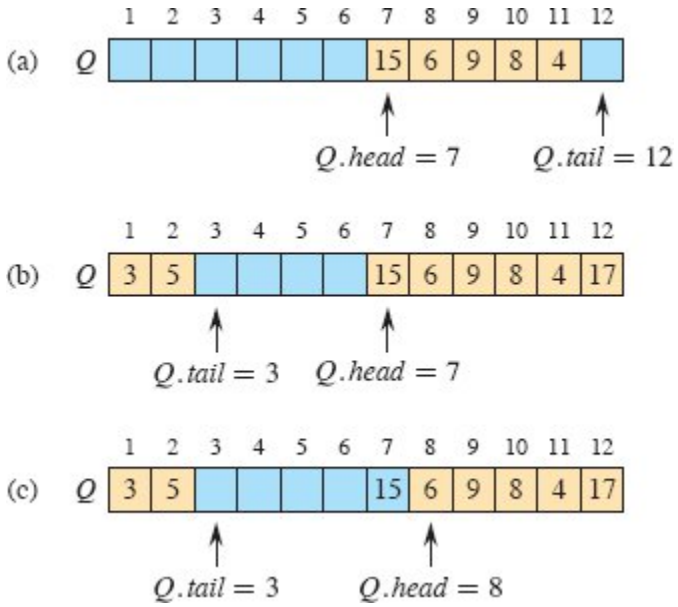

Figure 10.3 A queue implemented using an array $Q[1 : 12]$. Queue elements appear only in the tan positions. **(a)** The queue has 5 elements, in locations $Q[7 : 11]$. **(b)** The configuration of the queue after the calls ENQUEUE(Q, 17), ENQUEUE(Q, 3), and ENQUEUE(Q, 5). **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

Queues

We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE. Like the stack operation POP, DEQUEUE takes no element argument. The FIFO property of a queue causes it to operate like a line of customers waiting for service. The queue has a ***head*** and a ***tail***. When an element is enqueued, it takes its place at the tail of the queue, just as a newly arriving customer takes a place at the end of the line. The element dequeued is always the one at the head of the queue, like the customer at the head of the line, who has waited the longest.

Figure 10.3 shows one way to implement a queue of at most $n - 1$ elements using an array $Q[1 : n]$, with the attribute $Q.size$ equaling the size n of the array. The queue has an attribute $Q.head$ that indexes, or points to, its head. The attribute $Q.tail$ indexes the next location at which a newly arriving element will be inserted into the queue. The

elements in the queue reside in locations $Q.head$, $Q.head + 1$, … , $Q.tail - 1$, where we "wrap around" in the sense that location 1 immediately follows location n in a circular order. When $Q.head = Q.tail$, the queue is empty. Initially, we have $Q.head = Q.tail = 1$. An attempt to dequeue an element from an empty queue causes the queue to underflow. When $Q.head = Q.tail + 1$ or both $Q.head = 1$ and $Q.tail = Q.size$, the queue is full, and an attempt to enqueue an element causes the queue to overflow.

In the procedures ENQUEUE and DEQUEUE, we have omitted the error checking for underflow and overflow. (Exercise 10.1-5 asks you to supply these checks.) Figure 10.3 shows the effects of the ENQUEUE and DEQUEUE operations. Each operation takes $O(1)$ time.

```
ENQUEUE(Q, x)

1 Q[Q.tail] = x
2 if Q.tail == Q.size
3      Q.tail = 1
4 else Q.tail = Q.tail + 1

DEQUEUE(Q)

1 x = Q[Q.head]
2 if Q.head == Q.size
3      Q.head = 1
4 else Q.head = Q.head + 1
5 return x
```

Exercises

10.1-1

Consider an $m \times n$ matrix in row-major order, where both m and n are powers of 2 and rows and columns are indexed from 0. We can represent a row index i in binary by the $\lg m$ bits $\langle i_{\lg m - 1}, i_{\lg m - 2}, \ldots, i_0 \rangle$ and a column index j in binary by the $\lg n$ bits $\langle j_{\lg n - 1}, j_{\lg n - 2}, \ldots, j_0 \rangle$. Suppose that this matrix is a 2×2 block matrix, where each block has

$m/2$ rows and $n/2$ columns, and it is to be represented by a single array with 0-origin indexing. Show how to construct the binary representation of the $(\lg m + \lg n)$-bit index into the single array from the binary representations of i and j.

10.1-2
Using Figure 10.2 as a model, illustrate the result of each operation in the sequence PUSH(S, 4), PUSH(S, 1), PUSH(S, 3), POP(S), PUSH(S, 8), and POP(S) on an initially empty stack S stored in array $S[1 : 6]$

10.1-3
Explain how to implement two stacks in one array $A[1 : n]$ in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The PUSH and POP operations should run in $O(1)$ time.

10.1-4
Using Figure 10.3 as a model, illustrate the result of each operation in the sequence ENQUEUE(Q, 4), ENQUEUE(Q, 1), ENQUEUE(Q, 3), DEQUEUE(Q), ENQUEUE(Q, 8), and DEQUEUE(Q) on an initially empty queue Q stored in array $Q[1 : 6]$.

10.1-5
Rewrite ENQUEUE and DEQUEUE to detect underflow and overflow of a queue.

10.1-6
Whereas a stack allows insertion and deletion of elements at only one end, and a queue allows insertion at one end and deletion at the other end, a ***deque*** (double-ended queue, pronounced like "deck") allows insertion and deletion at both ends. Write four $O(1)$-time procedures to insert elements into and delete elements from both ends of a deque implemented by an array.

10.1-7
Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

10.1-8
Show how to implement a stack using two queues. Analyze the running time of the stack operations.

10.2 Linked lists

A ***linked list*** is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object. Since the elements of linked lists often contain keys that can be searched for, linked lists are sometimes called ***search lists***. Linked lists provide a simple, flexible representation for dynamic sets, supporting (though not necessarily efficiently) all the operations listed on page 250.

As shown in Figure 10.4, each element of a ***doubly linked list*** L is an object with an attribute *key* and two pointer attributes: *next* and *prev*. The object may also contain other satellite data. Given an element x in the list, $x.next$ points to its successor in the linked list, and $x.prev$ points to its predecessor. If $x.prev$ = NIL, the element x has no predecessor and is therefore the first element, or ***head***, of the list. If $x.next$ = NIL, the element x has no successor and is therefore the last element, or ***tail***, of the list. An attribute $L.head$ points to the first element of the list. If $L.head$ = NIL, the list is empty.

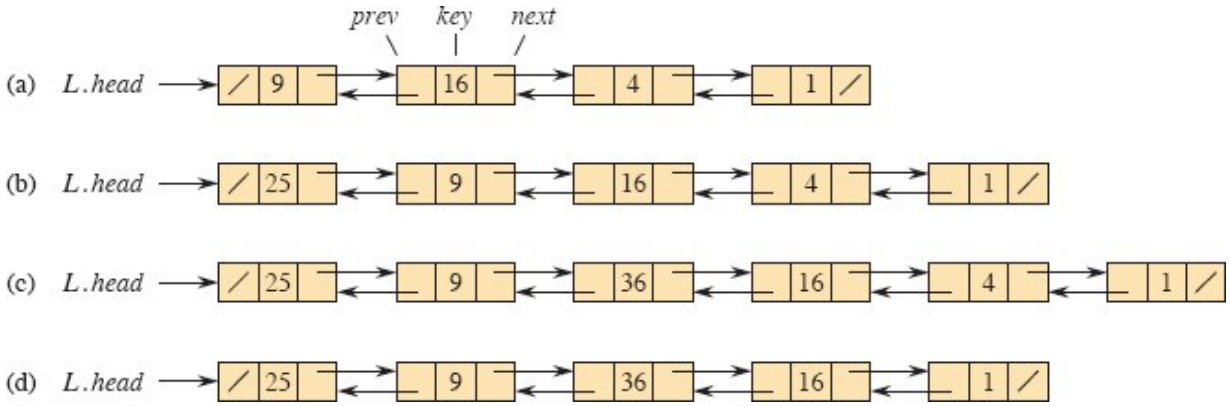

Figure 10.4 (a) A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute *L.head* points to the head. **(b)** Following the execution of LIST-PREPEND(L, x), where $x.key = 25$, the linked list has an object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of calling LIST-INSERT(x, y), where $x.key = 36$ and y points to the object with key 9. **(d)** The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

A list may have one of several forms. It may be either singly linked or doubly linked, it may be sorted or not, and it may be circular or not. If a list is ***singly linked***, each element has a *next* pointer but not a *prev* pointer. If a list is ***sorted***, the linear order of the list corresponds to the linear order of keys stored in elements of the list. The minimum element is then the head of the list, and the maximum element is the tail. If the list is ***unsorted***, the elements can appear in any order. In a ***circular list***, the *prev* pointer of the head of the list points to the tail, and the *next* pointer of the tail of the list points to the head. You can think of a circular list as a ring of elements. In the remainder of this section, we assume that the lists we are working with are unsorted and doubly linked.

Searching a linked list

The procedure LIST-SEARCH(L, k) finds the first element with key k in list L by a simple linear search, returning a pointer to this element. If no object with key k appears in the list, then the procedure returns NIL. For the linked list in Figure 10.4(a), the call LIST-SEARCH(L, 4)

returns a pointer to the third element, and the call LIST-SEARCH(L, 7) returns NIL. To search a list of n objects, the LIST-SEARCH procedure takes $\Theta(n)$ time in the worst case, since it may have to search the entire list.

```
LIST-SEARCH(L, k)

1 x = L.head
2 while x ≠ NIL and x.key ≠ k
3       x = x.next
4 return x
```

Inserting into a linked list

Given an element x whose *key* attribute has already been set, the LIST-PREPEND procedure adds x to the front of the linked list, as shown in Figure 10.4(b). (Recall that our attribute notation can cascade, so that *L.head.prev* denotes the *prev* attribute of the object that *L.head* points to.) The running time for LIST-PREPEND on a list of n elements is $O(1)$.

```
LIST-PREPEND(L, x)

1 x.next = L.head
2 x.prev = NIL
3 if L.head ≠ NIL
4       L.head.prev = x
5 L.head = x
```

You can insert anywhere within a linked list. As Figure 10.4(c) shows, if you have a pointer y to an object in the list, the LIST-INSERT procedure on the facing page "splices" a new element x into the list, immediately following y, in $O(1)$ time. Since LIST-INSERT never references the list object L, it is not supplied as a parameter.

```
LIST-INSERT(x, y)
```

```
1 x.next = y.next
2 x.prev = y
3 if y.next ≠ NIL
4       y.next.prev = x
5 y.next = x
```

Deleting from a linked list

The procedure LIST-DELETE removes an element x from a linked list L. It must be given a pointer to x, and it then "“splices" x out of the list by updating pointers. To delete an element with a given key, first call LIST-SEARCH to retrieve a pointer to the element. Figure 10.4(d) shows how an element is deleted from a linked list. LIST-DELETE runs in $O(1)$ time, but to delete an element with a given key, the call to LIST-SEARCH makes the worst-case running time be $\Theta(n)$.

```
LIST-DELETE(L, x)

1 if x.prev ≠ NIL
2       x.prev.next = x.next
3 else L.head = x.next
4 if x.next ≠ NIL
5       x.next.prev = x.prev
```

Insertion and deletion are faster operations on doubly linked lists than on arrays. If you want to insert a new first element into an array or delete the first element in an array, maintaining the relative order of all the existing elements, then each of the existing elements needs to be moved by one position. In the worst case, therefore, insertion and deletion take $\Theta(n)$ time in an array, compared with $O(1)$ time for a doubly linked list. (Exercise 10.2-1 asks you to show that deleting an element from a singly linked list takes $\Theta(n)$ time in the worst case.) If, however, you want to find the kth element in the linear order, it takes just $O(1)$ time in an array regardless of k, but in a linked list, you'd have to traverse k elements, taking $\Theta(k)$ time.

Sentinels

The code for LIST-DELETE is simpler if you ignore the boundary conditions at the head and tail of the list:

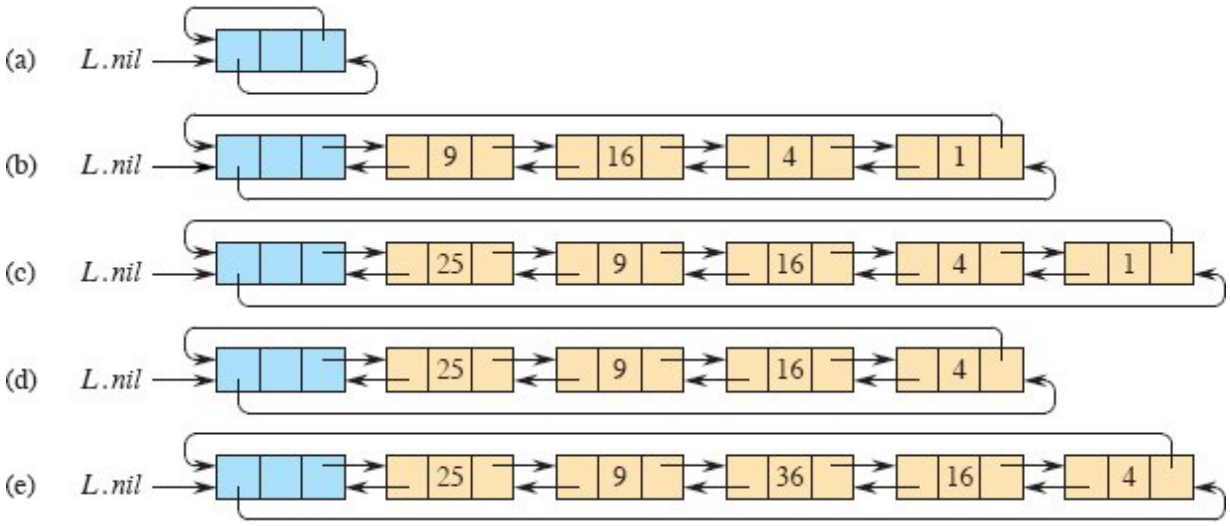

Figure 10.5 A circular, doubly linked list with a sentinel. The sentinel *L.nil*, in blue, appears between the head and tail. The attribute *L.head* is no longer needed, since the head of the list is *L.nil.next*. **(a)** An empty list. **(b)** The linked list from Figure 10.4(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT′ (x, *L.nil*), where $x.key = 25$. The new object becomes the head of the list. **(d)** The list after deleting the object with key 1. The new tail is the object with key 4. **(e)** The list after executing LIST-INSERT′ (x, y), where $x.key = 36$ and y points to the object with key 9.

```
LIST-DELETE′ (x)
1 x.prev.next = x.next
2 x.next.prev = x.prev
```

A ***sentinel*** is a dummy object that allows us to simplify boundary conditions. In a linked list L, the sentinel is an object *L.nil* that represents NIL but has all the attributes of the other objects in the list. References to NIL are replaced by references to the sentinel *L.nil*. As shown in Figure 10.5, this change turns a regular doubly linked list into a ***circular, doubly linked list with a sentinel***, in which the sentinel *L.nil* lies between the head and tail. The attribute *L.nil.next* points to the head of the list, and *L.nil.prev* points to the tail. Similarly, both the *next*

attribute of the tail and the *prev* attribute of the head point to *L.nil*. Since *L.nil.next* points to the head, the attribute *L.head* is eliminated altogether, with references to it replaced by references to *L.nil.next*. Figure 10.5(a) shows that an empty list consists of just the sentinel, and both *L.nil.next* and *L.nil.prev* point to *L.nil*.

To delete an element from the list, just use the two-line procedure LIST-DELETE′ from before. Just as LIST-INSERT never references the list object *L*, neither does LIST-DELETE′. You should never delete the sentinel *L.nil* unless you are deleting the entire list!

The LIST-INSERT′ procedure inserts an element x into the list following object y. No separate procedure for prepending is necessary: to insert at the head of the list, let y be *L.nil*; and to insert at the tail, let y be *L.nil.prev*. Figure 10.5 shows the effects of LIST-INSERT′ and LIST-DELETE′ on a sample list.

```
LIST-INSERT′ (x, y)
1 x.next = y.next
2 x.prev = y
3 y.next.prev = x
4 y.next = x
```

Searching a circular, doubly linked list with a sentinel has the same asymptotic running time as without a sentinel, but it is possible to decrease the constant factor. The test in line 2 of LIST-SEARCH makes two comparisons: one to check whether the search has run off the end of the list and, if not, one to check whether the key resides in the current element x. Suppose that you *know* that the key is somewhere in the list. Then you do not need to check whether the search runs off the end of the list, thereby eliminating one comparison in each iteration of the **while** loop.

The sentinel provides a place to put the key before starting the search. The search starts at the head *L.nil.next* of list *L*, and it stops if it finds the key somewhere in the list. Now the search is guaranteed to find the key, either in the sentinel or before reaching the sentinel. If the key is found before reaching the sentinel, then it really is in the element where

the search stops. If, however, the search goes through all the elements in the list and finds the key only in the sentinel, then the key is not really in the list, and the search returns NIL. The procedure LIST-SEARCH′ embodies this idea. (If your sentinel requires its *key* attribute to be NIL, then you might want to assign $L.nil.key$ = NIL before line 5.)

```
LIST-SEARCH′ (L, k)
1 L.nil.key = k    // store the key in the sentinel to guarantee it is in list
2 x = L.nil.next   // start at the head of the list
3 while x.key ≠ k
4     x = x.next
5 if x == L.nil    // found k in the sentinel
6     return NIL   // k was not really in the list
7 else return x    // found k in element x
```

Sentinels often simplify code and, as in searching a linked list, they might speed up code by a small constant factor, but they don't typically improve the asymptotic running time. Use them judiciously. When there are many small lists, the extra storage used by their sentinels can represent significant wasted memory. In this book, we use sentinels only when they significantly simplify the code.

Exercises

10.2-1

Explain why the dynamic-set operation INSERT on a singly linked list can be implemented in $O(1)$ time, but the worst-case time for DELETE is $\Theta(n)$.

10.2-2

Implement a stack using a singly linked list. The operations PUSH and POP should still take $O(1)$ time. Do you need to add any attributes to the list?

10.2-3

Implement a queue using a singly linked list. The operations ENQUEUE and DEQUEUE should still take $O(1)$ time. Do you need to add any attributes to the list?

10.2-4
The dynamic-set operation UNION takes two disjoint sets S_1 and S_2 as input, and it returns a set $S = S_1 \cup S_2$ consisting of all the elements of S_1 and S_2. The sets S_1 and S_2 are usually destroyed by the operation. Show how to support UNION in $O(1)$ time using a suitable list data structure.

10.2-5
Give a $\Theta(n)$-time nonrecursive procedure that reverses a singly linked list of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.

★ ***10.2-6***
Explain how to implement doubly linked lists using only one pointer value $x.np$ per item instead of the usual two (*next* and *prev*). Assume that all pointer values can be interpreted as k-bit integers, and define $x.np = x.next$ XOR $x.prev$, the k-bit "exclusive-or" of $x.next$ and $x.prev$. The value NIL is represented by 0. Be sure to describe what information you need to access the head of the list. Show how to implement the SEARCH, INSERT, and DELETE operations on such a list. Also show how to reverse such a list in $O(1)$ time.

10.3 Representing rooted trees

Linked lists work well for representing linear relationships, but not all relationships are linear. In this section, we look specifically at the problem of representing rooted trees by linked data structures. We first look at binary trees, and then we present a method for rooted trees in which nodes can have an arbitrary number of children.

We represent each node of a tree by an object. As with linked lists, we assume that each node contains a *key* attribute. The remaining

attributes of interest are pointers to other nodes, and they vary according to the type of tree.

Binary trees

Figure 10.6 shows how to use the attributes p, *left*, and *right* to store pointers to the parent, left child, and right child of each node in a binary tree T. If $x.p$ = NIL, then x is the root. If node x has no left child, then $x.left$ = NIL, and similarly for the right child. The root of the entire tree T is pointed to by the attribute $T.root$. If $T.root$ = NIL, then the tree is empty.

Rooted trees with unbounded branching

It's simple to extend the scheme for representing a binary tree to any class of trees in which the number of children of each node is at most some constant k: replace the *left* and *right* attributes by $child_1$, $child_2$, … , $child_k$. This scheme no longer works when the number of children of a node is unbounded, however, since we do not know how many attributes to allocate in advance. Moreover, if k, the number of children, is bounded by a large constant but most nodes have a small number of children, we may waste a lot of memory.

Fortunately, there is a clever scheme to represent trees with arbitrary numbers of children. It has the advantage of using only $O(n)$ space for any n-node rooted tree. The ***left-child, right-sibling representation*** appears in Figure 10.7. As before, each node contains a parent pointer p, and $T.root$ points to the root of tree T. Instead of having a pointer to each of its children, however, each node x has only two pointers:

1. $x.left\text{-}child$ points to the leftmost child of node x, and
2. $x.right\text{-}sibling$ points to the sibling of x immediately to its right.

If node x has no children, then $x.left\text{-}child$ = NIL, and if node x is the rightmost child of its parent, then $x.right\text{-}sibling$ = NIL.

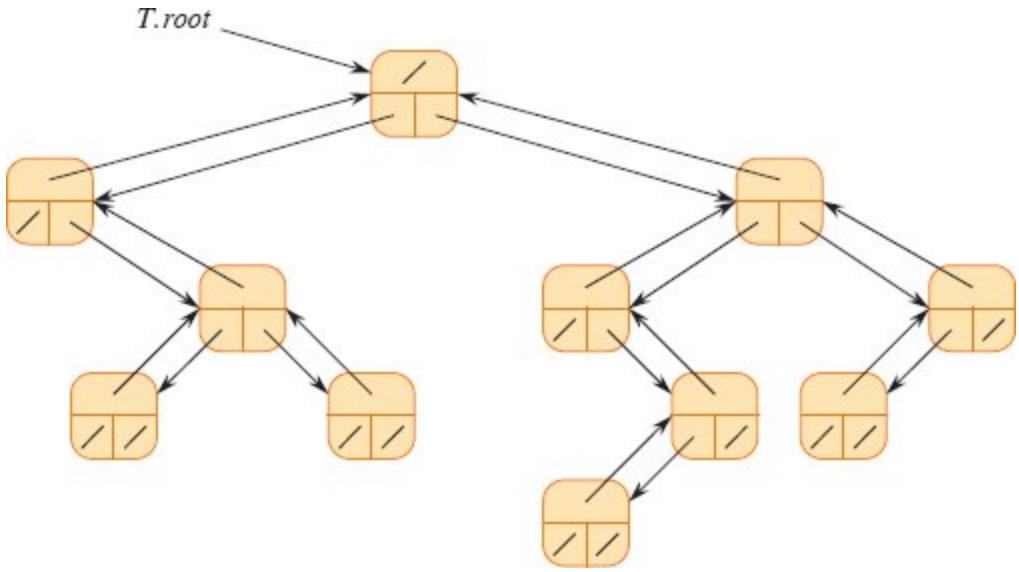

Figure 10.6 The representation of a binary tree T. Each node x has the attributes $x.p$ (top), $x.left$ (lower left), and $x.right$ (lower right). The *key* attributes are not shown.

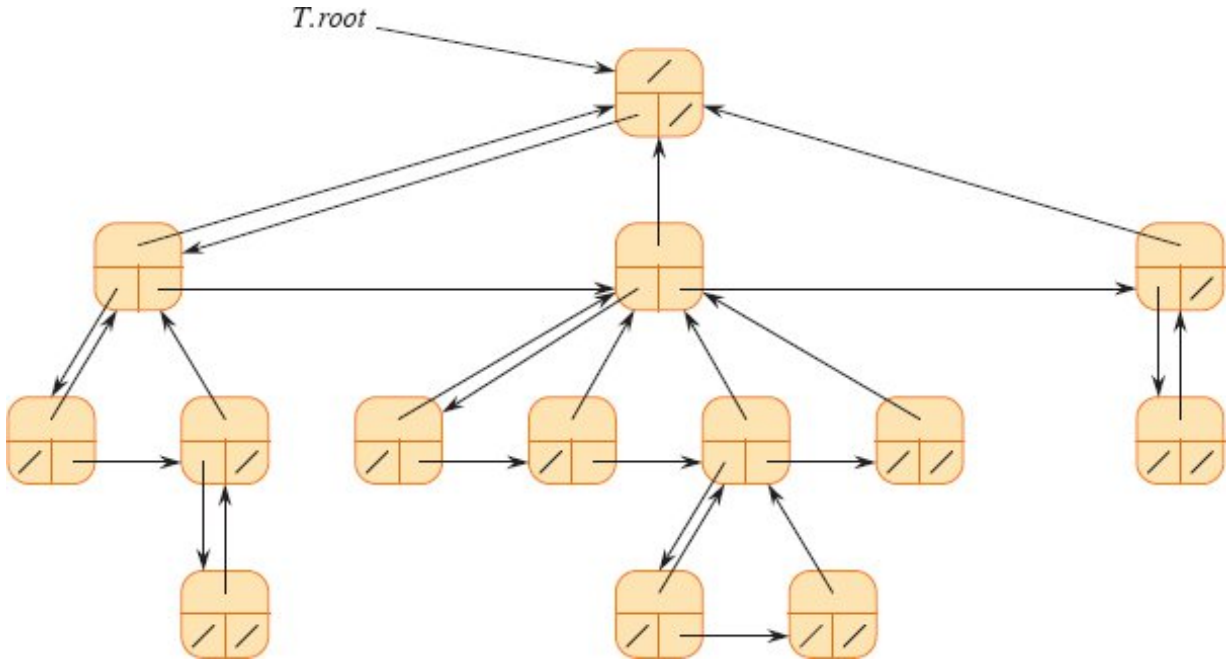

Figure 10.7 The left-child, right-sibling representation of a tree T. Each node x has attributes $x.p$ (top), $x.left\text{-}child$ (lower left), and $x.right\text{-}sibling$ (lower right). The *key* attributes are not shown.

Other tree representations

We sometimes represent rooted trees in other ways. In Chapter 6, for example, we represented a heap, which is based on a complete binary tree, by a single array along with an attribute giving the index of the last node in the heap. The trees that appear in Chapter 19 are traversed only toward the root, and so only the parent pointers are present: there are no pointers to children. Many other schemes are possible. Which scheme is best depends on the application.

Exercises

10.3-1

Draw the binary tree rooted at index 6 that is represented by the following attributes:

index	*key*	*left*	*right*
1	17	8	9
2	14	NIL	NIL
3	12	NIL	NIL
4	20	10	NIL
5	33	2	NIL
6	15	1	4
7	28	NIL	NIL
8	22	NIL	NIL
9	13	3	7
10	25	NIL	5

10.3-2

Write an $O(n)$-time recursive procedure that, given an n-node binary tree, prints out the key of each node in the tree.

10.3-3

Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node in the tree. Use a stack as an auxiliary data structure.

10.3-4

Write an $O(n)$-time procedure that prints out all the keys of an arbitrary rooted tree with n nodes, where the tree is stored using the left-child, right-sibling representation.

★ ***10.3-5***

Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node. Use no more than constant extra space outside of the tree itself and do not modify the tree, even temporarily, during the procedure.

★ ***10.3-6***

The left-child, right-sibling representation of an arbitrary rooted tree uses three pointers in each node: *left-child*, *right-sibling*, and *parent*. From any node, its parent can be accessed in constant time and all its children can be accessed in time linear in the number of children. Show how to use only two pointers and one boolean value in each node x so that x's parent or all of x's children can be accessed in time linear in the number of x's children.

Problems

10-1 Comparisons among lists

For each of the four types of lists in the following table, what is the asymptotic worst-case running time for each dynamic-set operation listed?

	unsorted, singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked
SEARCH				
INSERT				
DELETE				
SUCCESSOR				
PREDECESSOR				

MINIMUM				
MAXIMUM				

10-2 Mergeable heaps using linked lists

A ***mergeable heap*** supports the following operations: MAKE-HEAP (which creates an empty mergeable heap), INSERT, MINIMUM, EXTRACT-MIN, and UNION.[1] Show how to implement mergeable heaps using linked lists in each of the following cases. Try to make each operation as efficient as possible. Analyze the running time of each operation in terms of the size of the dynamic set(s) being operated on.

a. Lists are sorted.

b. Lists are unsorted.

c. Lists are unsorted, and dynamic sets to be merged are disjoint.

10-3 Searching a sorted compact list

We can represent a singly linked list with two arrays, *key* and *next*. Given the index i of an element, its value is stored in $key[i]$, and the index of its successor is given by $next[i]$, where $next[i] = \text{NIL}$ for the last element. We also need the index *head* of the first element in the list. An n-element list stored in this way is ***compact*** if it is stored only in positions 1 through n of the *key* and *next* arrays.

Let's assume that all keys are distinct and that the compact list is also sorted, that is, $key[i] < key[next[i]]$ for all $i = 1, 2, \ldots, n$ such that $next[i] \neq \text{NIL}$. Under these assumptions, you will show that the randomized algorithm COMPACT-LIST-SEARCH searches the list for key k in $O(\sqrt{n})$ expected time.

```
COMPACT-LIST-SEARCH(key, next, head, n, k)
1 i = head
2 while i ≠ NIL and key[i] < k
3     j = RANDOM(1, n)
4     if key[i] < key[j] and key[j] ≤ k
5         i = j
```

```
6          if key[i] == k
7              return i
8      i = next[i]
9  if i == NIL or key[i] > k
10     return NIL
11 else return i
```

If you ignore lines 3–7 of the procedure, you can see that it's an ordinary algorithm for searching a sorted linked list, in which index i points to each position of the list in turn. The search terminates once the index i "falls off" the end of the list or once $key[i] \geq k$. In the latter case, if $key[i] = k$, the procedure has found a key with the value k. If, however, $key[i] > k$, then the search will never find a key with the value k, so that terminating the search was the correct action.

Lines 3–7 attempt to skip ahead to a randomly chosen position j. Such a skip helps if $key[j]$ is larger than $key[i]$ and no larger than k. In such a case, j marks a position in the list that i would reach during an ordinary list search. Because the list is compact, we know that any choice of j between 1 and n indexes some element in the list.

Instead of analyzing the performance of COMPACT-LIST-SEARCH directly, you will analyze a related algorithm, COMPACT-LIST-SEARCH′, which executes two separate loops. This algorithm takes an additional parameter t, which specifies an upper bound on the number of iterations of the first loop.

```
COMPACT-LIST-SEARCH′ (key, next, head, n, k, t)
1  i = head
2  for q = 1 to t
3      j = RANDOM(1, n)
4      if key[i] < key[j] and key[j] ≤ k
5          i = j
6          if key[i] == k
7              return i
8  while i ≠ NIL and key[i] < k
9      i = next[i]
```

```
10 if i == NIL or key[i] > k
11     return NIL
12 else return i
```

To compare the execution of the two algorithms, assume that the sequence of calls of RANDOM(1, n) yields the same sequence of integers for both algorithms.

a. Argue that for any value of t, COMPACT-LIST-SEARCH(key, $next$, $head$, n, k) and COMPACT-LIST-SEARCH′ (key, $next$, $head$, n, k, t) return the same result and that the number of iterations of the **while** loop of lines 2–8 in COMPACT-LIST-SEARCH is at most the total number of iterations of both the **for** and **while** loops in COMPACT-LIST-SEARCH′.

In the call COMPACT-LIST-SEARCH′ (key, $next$, $head$, n, k, t), let X_t be the random variable that describes the distance in the linked list (that is, through the chain of $next$ pointers) from position i to the desired key k after t iterations of the **for** loop of lines 2–7 have occurred.

b. Argue that COMPACT-LIST-SEARCH′ (key, $next$, $head$, n, k, t) has an expected running time of $O(t + \mathrm{E}[X_t])$.

c. Show that $\mathrm{E}[X_t] = \sum_{r=1}^{n}(1-r/n)^t$. (*Hint:* Use equation (C.28) on page 1193.)

d. Show that $\sum_{r=0}^{n-1} r^t \le n^{t+1}/(t+1)$. (*Hint:* Use inequality (A.18) on page 1150.)

e. Prove that $\mathrm{E}[X_t] \le n/(t+1)$.

f. Show that COMPACT-LIST-SEARCH′ (key, $next$, $head$, n, k, t) has an expected running time of $O(t + n/t)$.

g. Conclude that COMPACT-LIST-SEARCH runs in $O(\sqrt{n})$ expected time.

h. Why do we assume that all keys are distinct in COMPACT-LIST-SEARCH? Argue that random skips do not necessarily help

asymptotically when the list contains repeated key values.

Chapter notes

Aho, Hopcroft, and Ullman [6] and Knuth [259] are excellent references for elementary data structures. Many other texts cover both basic data structures and their implementation in a particular programming language. Examples of these types of textbooks include Goodrich and Tamassia [196], Main [311], Shaffer [406], and Weiss [452, 453, 454]. The book by Gonnet and Baeza-Yates [193] provides experimental data on the performance of many data-structure operations.

The origin of stacks and queues as data structures in computer science is unclear, since corresponding notions already existed in mathematics and paper-based business practices before the introduction of digital computers. Knuth [259] cites A. M. Turing for the development of stacks for subroutine linkage in 1947.

Pointer-based data structures also seem to be a folk invention. According to Knuth, pointers were apparently used in early computers with drum memories. The A-1 language developed by G. M. Hopper in 1951 represented algebraic formulas as binary trees. Knuth credits the IPL-II language, developed in 1956 by A. Newell, J. C. Shaw, and H. A. Simon, for recognizing the importance and promoting the use of pointers. Their IPL-III language, developed in 1957, included explicit stack operations.

[1] Because we have defined a mergeable heap to support MINIMUM and EXTRACT-MIN, we can also refer to it as a ***mergeable min-heap***. Alternatively, if it supports MAXIMUM and EXTRACT-MAX, it is a ***mergeable max-heap***.

11 Hash Tables

Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE. For example, a compiler that translates a programming language maintains a symbol table, in which the keys of elements are arbitrary character strings corresponding to identifiers in the language. A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list—$\Theta(n)$ time in the worst case—in practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$. Indeed, the built-in dictionaries of Python are implemented with hash tables.

A hash table generalizes the simpler notion of an ordinary array. Directly addressing into an ordinary array takes advantage of the $O(1)$ access time for any array element. Section 11.1 discusses direct addressing in more detail. To use direct addressing, you must be able to allocate an array that contains a position for every possible key.

When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored. Instead of using the key as an array index directly, we *compute* the array index from the key. Section 11.2 presents the main ideas, focusing on "chaining" as a way to handle "collisions," in which more than one key maps to the same array index. Section 11.3 describes how to compute array indices from keys using hash functions. We present and analyze several

variations on the basic theme. Section 11.4 looks at "open addressing," which is another way to deal with collisions. The bottom line is that hashing is an extremely effective and practical technique: the basic dictionary operations require only $O(1)$ time on the average. Section 11.5 discusses the hierarchical memory systems of modern computer systems have and illustrates how to design hash tables that work well in such systems.

11.1 Direct-address tables

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small. Suppose that an application needs a dynamic set in which each element has a distinct key drawn from the universe $U = \{0, 1, \ldots, m - 1\}$, where m is not too large.

To represent the dynamic set, you can use an array, or ***direct-address table***, denoted by $T[0 : m - 1]$, in which each position, or ***slot***, corresponds to a key in the universe U. Figure 11.1 illustrates this approach. Slot k points to an element in the set with key k. If the set contains no element with key k, then $T[k] = \text{NIL}$.

The dictionary operations DIRECT-ADDRESS-SEARCH, DIRECT-ADDRESS-INSERT, and DIRECT-ADDRESS-DELETE on the following page are trivial to implement. Each takes only $O(1)$ time.

For some applications, the direct-address table itself can hold the elements in the dynamic set. That is, rather than storing an element's key and satellite data in an object external to the direct-address table, with a pointer from a slot in the table to the object, save space by storing the object directly in the slot. To indicate an empty slot, use a special key. Then again, why store the key of the object at all? The index of the object *is* its key! Of course, then you'd need some way to tell whether slots are empty.

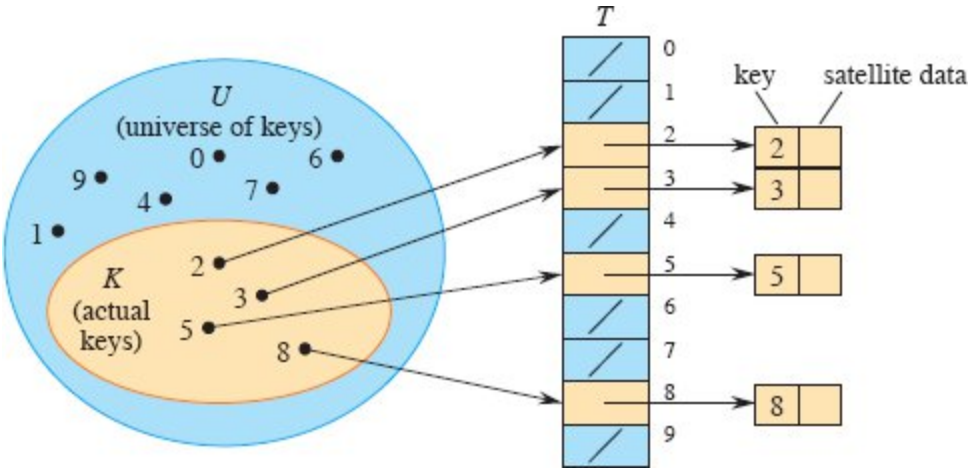

Figure 11.1 How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index into the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, in blue, contain NIL.

```
DIRECT-ADDRESS-SEARCH(T, k)
1 return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1 T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
1 T[x.key] = NIL
```

Exercises

11.1-1

A dynamic set S is represented by a direct-address table T of length m. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?

11.1-2

A ***bit vector*** is simply an array of bits (each either 0 or 1). A bit vector of length m takes much less space than an array of m pointers. Describe

how to use a bit vector to represent a dynamic set of distinct elements drawn from the set $\{0, 1, \ldots, m - 1\}$ and with no satellite data. Dictionary operations should run in $O(1)$ time.

11.1-3

Suggest how to implement a direct-address table in which the keys of stored elements do not need to be distinct and the elements can have satellite data. All three dictionary operations (INSERT, DELETE, and SEARCH) should run in $O(1)$ time. (Don't forget that DELETE takes as an argument a pointer to an object to be deleted, not a key.)

★ 11.1-4

Suppose that you want to implement a dictionary by using direct addressing on a *huge* array. That is, if the array size is m and the dictionary contains at most n elements at any one time, then $m \gg n$. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct-address dictionary on a huge array. Each stored object should use $O(1)$ space; the operations SEARCH, INSERT, and DELETE should take $O(1)$ time each; and initializing the data structure should take $O(1)$ time. (*Hint:* Use an additional array, treated somewhat like a stack whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.)

11.2 Hash tables

The downside of direct addressing is apparent: if the universe U is large or infinite, storing a table T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. Furthermore, the set K of keys *actually stored* may be so small relative to U that most of the space allocated for T would be wasted.

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct-address table. Specifically, the storage requirement

reduces to $\Theta(|K|)$ while maintaining the benefit that searching for an element in the hash table still requires only $O(1)$ time. The catch is that this bound is for the *average-case time*,[1] whereas for direct addressing it holds for the *worst-case time*.

With direct addressing, an element with key k is stored in slot k, but with hashing, we use a ***hash function*** h to compute the slot number from the key k, so that the element goes into slot $h(k)$. The hash function h maps the universe U of keys into the slots of a ***hash table*** $T[0 : m - 1]$:

$$h : U \to \{0, 1, \ldots, m - 1\},$$

where the size m of the hash table is typically much less than $|U|$. We say that an element with key k ***hashes*** to slot $h(k)$, and we also say that $h(k)$ is the ***hash value*** of key k. Figure 11.2 illustrates the basic idea. The hash function reduces the range of array indices and hence the size of the array. Instead of a size of $|U|$, the array can have size m. An example of a simple, but not particularly good, hash function is $h(k) = k \bmod m$.

There is one hitch, namely that two keys may hash to the same slot. We call this situation a ***collision***. Fortunately, there are effective techniques for resolving the conflict created by collisions.

Of course, the ideal solution is to avoid collisions altogether. We might try to achieve this goal by choosing a suitable hash function h. One idea is to make h appear to be "random," thus avoiding collisions or at least minimizing their number. The very term "to hash," evoking images of random mixing and chopping, captures the spirit of this approach. (Of course, a hash function h must be deterministic in that a given input k must always produce the same output $h(k)$.) Because $|U| > m$, however, there must be at least two keys that have the same hash value, and avoiding collisions altogether is impossible. Thus, although a well-designed, "random"-looking hash function can reduce the number of collisions, we still need a method for resolving the collisions that do occur.

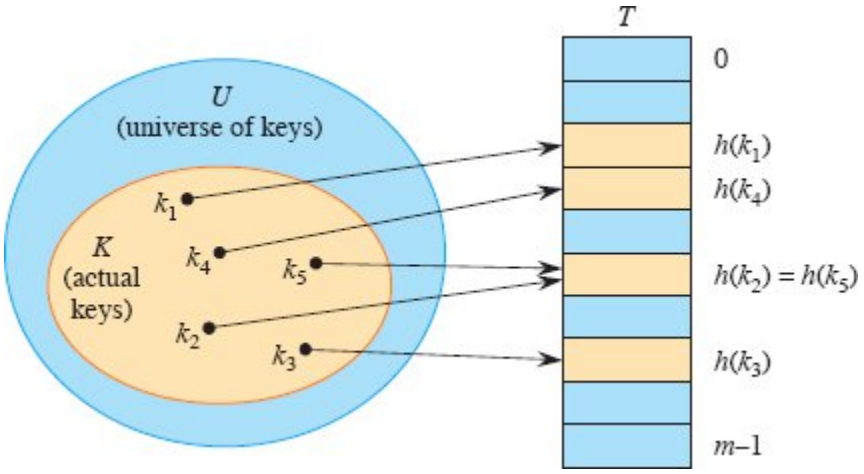

Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

The remainder of this section first presents a definition of "independent uniform hashing," which captures the simplest notion of what it means for a hash function to be "random." It then presents and analyzes the simplest collision resolution technique, called chaining. Section 11.4 introduces an alternative method for resolving collisions, called open addressing.

Independent uniform hashing

An "ideal" hashing function h would have, for each possible input k in the domain U, an output $h(k)$ that is an element randomly and independently chosen uniformly from the range $\{0, 1, \ldots, m - 1\}$. Once a value $h(k)$ is randomly chosen, each subsequent call to h with the same input k yields the same output $h(k)$.

We call such an ideal hash function an ***independent uniform hash function***. Such a function is also often called a ***random oracle*** [43]. When hash tables are implemented with an independent uniform hash function, we say we are using ***independent uniform hashing***.

Independent uniform hashing is an ideal theoretical abstraction, but it is not something that can reasonably be implemented in practice. Nonetheless, we'll analyze the efficiency of hashing under the

assumption of independent uniform hashing and then present ways of achieving useful practical approximations to this ideal.

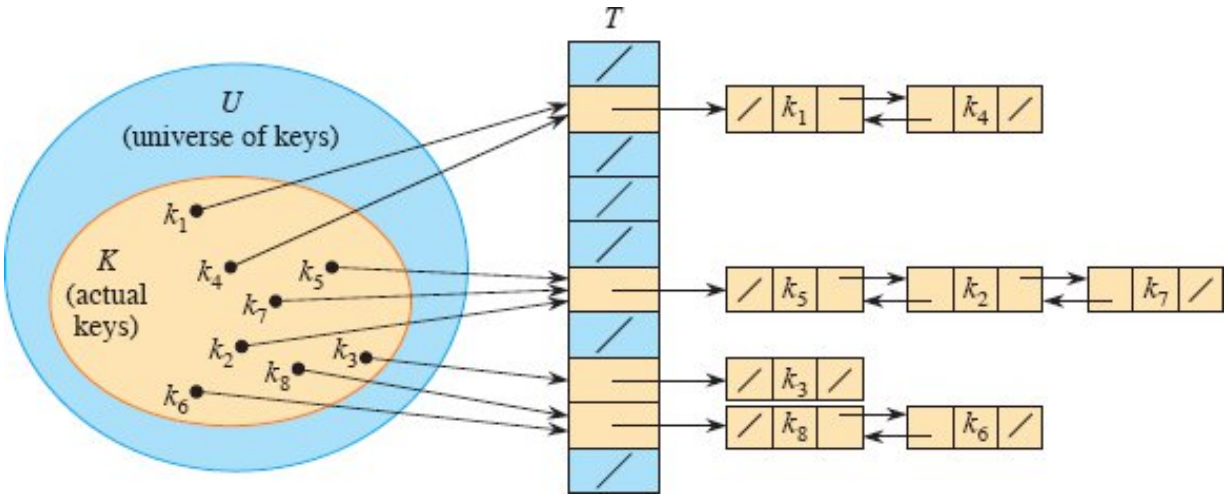

Figure 11.3 Collision resolution by chaining. Each nonempty hash-table slot $T[j]$ points to a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$. The list can be either singly or doubly linked. We show it as doubly linked because deletion may be faster that way when the deletion procedure knows which list element (not just which key) is to be deleted.

Collision resolution by chaining

At a high level, you can think of hashing with chaining as a nonrecursive form of divide-and-conquer: the input set of n elements is divided randomly into m subsets, each of approximate size n/m. A hash function determines which subset an element belongs to. Each subset is managed independently as a list.

Figure 11.3 shows the idea behind ***chaining***: each nonempty slot points to a linked list, and all the elements that hash to the same slot go into that slot's linked list. Slot j contains a pointer to the head of the list of all stored elements with hash value j. If there are no such elements, then slot j contains NIL.

When collisions are resolved by chaining, the dictionary operations are straightforward to implement. They appear on the next page and use the linked-list procedures from Section 10.2. The worst-case running time for insertion is $O(1)$. The insertion procedure is fast in part because

it assumes that the element x being inserted is not already present in the table. To enforce this assumption, you can search (at additional cost) for an element whose key is $x.key$ before inserting. For searching, the worst-case running time is proportional to the length of the list. (We'll analyze this operation more closely below.) Deletion takes $O(1)$ time if the lists are doubly linked, as in Figure 11.3. (Since CHAINED-HASH-DELETE takes as input an element x and not its key k, no search is needed. If the hash table supports deletion, then its linked lists should be doubly linked in order to delete an item quickly. If the lists were only singly linked, then by Exercise 10.2-1, deletion could take time proportional to the length of the list. With singly linked lists, both deletion and searching would have the same asymptotic running times.)

```
CHAINED-HASH-INSERT(T, x)
1 LIST-PREPEND(T[h(x.key)], x)

CHAINED-HASH-SEARCH(T, k)
1 return LIST-SEARCH(T[h(k)], k)

CHAINED-HASH-DELETE(T, x)
1 LIST-DELETE(T[h(x.key)], x)
```

Analysis of hashing with chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given a hash table T with m slots that stores n elements, we define the ***load factor*** α for T as n/m, that is, the average number of elements stored in a chain. Our analysis will be in terms of α, which can be less than, equal to, or greater than 1.

The worst-case behavior of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n. The worst-case time for searching is thus $\Theta(n)$ plus the time to compute the hash function—no better than using one linked list for all the elements. We clearly don't use hash tables for their worst-case performance.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average (meaning with respect to the distribution of keys to be hashed and with respect to the choice of hash function, if this choice is randomized). Section 11.3 discusses these issues, but for now we assume that any given element is equally likely to hash into any of the m slots. That is, the hash function is ***uniform***. We further assume that where a given element hashes to is *independent* of where any other elements hash to. In other words, we assume that we are using ***independent uniform hashing***.

Because hashes of distinct keys are assumed to be independent, independent uniform hashing is ***universal***: the chance that any two distinct keys k_1 and k_2 collide is at most $1/m$. Universality is important in our analysis and also in the specification of universal families of hash functions, which we'll see in Section 11.3.2.

For $j = 0, 1, \ldots, m-1$, denote the length of the list $T[j]$ by n_j, so that

$$n = n_0 + n_1 + \cdots + n_{m-1} , \tag{11.1}$$

and the expected value of n_j is $\mathrm{E}[n_j] = \alpha = n/m$.

We assume that $O(1)$ time suffices to compute the hash value $h(k)$, so that the time required to search for an element with key k depends linearly on the length $n_{h(k)}$ of the list $T[h(k)]$. Setting aside the $O(1)$ time required to compute the hash function and to access slot $h(k)$, we'll consider the expected number of elements examined by the search algorithm, that is, the number of elements in the list $T[h(k)]$ that the algorithm checks to see whether any have a key equal to k. We consider two cases. In the first, the search is unsuccessful: no element in the table has key k. In the second, the search successfully finds an element with key k.

Theorem 11.1

In a hash table in which collisions are resolved by chaining, an unsuccessful search takes $\Theta(1 + \alpha)$ time on average, under the assumption of independent uniform hashing.

Proof Under the assumption of independent uniform hashing, any key k not already stored in the table is equally likely to hash to any of the m slots. The expected time to search unsuccessfully for a key k is the expected time to search to the end of list $T[h(k)]$, which has expected length $E[n_{h(k)}] = \alpha$. Thus, the expected number of elements examined in an unsuccessful search is α, and the total time required (including the time for computing $h(k)$) is $\Theta(1 + \alpha)$.

■

The situation for a successful search is slightly different. An unsuccessful search is equally likely to go to any slot of the hash table. A successful search, however, cannot go to an empty slot, since it is for an element that is present in one of the linked lists. We assume that the element searched for is equally likely to be any one of the elements in the table, so the longer the list, the more likely that the search is for one of its elements. Even so, the expected search time still turns out to be $\Theta(1 + \alpha)$.

Theorem 11.2
In a hash table in which collisions are resolved by chaining, a successful search takes $\Theta(1 + \alpha)$ time on average, under the assumption of independent uniform hashing.

Proof We assume that the element being searched for is equally likely to be any of the n elements stored in the table. The number of elements examined during a successful search for an element x is 1 more than the number of elements that appear before x in x's list. Because new elements are placed at the front of the list, elements before x in the list were all inserted after x was inserted. Let x_i denote the ith element inserted into the table, for $i = 1, 2, \ldots, n$, and let $k_i = x_i.key$.

Our analysis uses indicator random variables extensively. For each slot q in the table and for each pair of distinct keys k_i and k_j, we define the indicator random variable

$X_{ijq} = I\{\text{the search is for } x_i, h(k_i) = q, \text{ and } h(k_j) = q\}$.

That is, $X_{ijq} = 1$ when keys k_i and k_j collide at slot q and the search is for element x_i. Because $\Pr\{\text{the search is for } x_i\} = 1/n$, $\Pr\{h(k_i) = q\} = 1/m$, $\Pr\{h(k_j) = q\} = 1/m$, and these events are all independent, we have that $\Pr\{X_{ijq} = 1\} = 1/nm^2$. Lemma 5.1 on page 130 gives $\mathrm{E}[X_{ijq}] = 1/nm^2$.

Next, we define, for each element x_j, the indicator random variable

$$\begin{aligned} Y_j &= \mathrm{I}\{x_j \text{ appears in a list prior to the element being searched for}\} \\ &= \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq} , \end{aligned}$$

since at most one of the X_{ijq} equals 1, namely when the element x_i being searched for belongs to the same list as x_j (pointed to by slot q), and $i < j$ (so that x_i appears after x_j in the list).

Our final random variable is Z, which counts how many elements appear in the list prior to the element being searched for:

$$Z = \sum_{j=1}^{n} Y_j .$$

Because we must count the element being searched for as well as all those preceding it in its list, we wish to compute $\mathrm{E}[Z + 1]$. Using linearity of expectation (equation (C.24) on page 1192), we have

$$\begin{aligned} \mathrm{E}[Z+1] &= \mathrm{E}\left[1 + \sum_{j=1}^{n} Y_j\right] \\ &= 1 + \mathrm{E}\left[\sum_{j=1}^{n} \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq}\right] \\ &= 1 + \mathrm{E}\left[\sum_{q=0}^{m-1} \sum_{j=1}^{n} \sum_{i=1}^{j-1} X_{ijq}\right] \\ &= 1 + \sum_{q=0}^{m-1} \sum_{j=1}^{n} \sum_{i=1}^{j-1} \mathrm{E}[X_{ijq}] \qquad \text{(by linearity of expectation)} \end{aligned}$$

Thus, the total time required for a successful search (including the time for computing the hash function) is $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$.

■

What does this analysis mean? If the number of elements in the table is at most proportional to the number of hash-table slots, we have $n = O(m)$ and, consequently, $\alpha = n/m = O(m)/m = O(1)$. Thus, searching takes constant time on average. Since insertion takes $O(1)$ worst-case time and deletion takes $O(1)$ worst-case time when the lists are doubly linked (assuming that the list element to be deleted is known, and not just its key), we can support all dictionary operations in $O(1)$ time on average.

The analysis in the preceding two theorems depends only on two essential properties of independent uniform hashing: uniformity (each key is equally likely to hash to any one of the m slots), and independence (so any two distinct keys collide with probability $1/m$).

Exercises

11.2-1

You use a hash function h to hash n distinct keys into an array T of length m. Assuming independent uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of $\{\{k_1, k_2\} : k_1 \neq k_2 \text{ and } h(k_1) = h(k_2)\}$?

11.2-2

Consider a hash table with 9 slots and the hash function $h(k) = k \bmod 9$. Demonstrate what happens upon inserting the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 with collisions resolved by chaining.

11.2-3

Professor Marley hypothesizes that he can obtain substantial performance gains by modifying the chaining scheme to keep each list in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?

11.2-4

Suggest how to allocate and deallocate storage for elements within the hash table itself by creating a "free list": a linked list of all the unused slots. Assume that one slot can store a flag and either one element plus a pointer or two pointers. All dictionary and free-list operations should run in $O(1)$ expected time. Does the free list need to be doubly linked, or does a singly linked free list suffice?

11.2-5

You need to store a set of n keys in a hash table of size m. Show that if the keys are drawn from a universe U with $|U| > (n-1)m$, then U has a subset of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\Theta(n)$.

11.2-6

You have stored n keys in a hash table of size m, with collisions resolved by chaining, and you know the length of each chain, including the length L of the longest chain. Describe a procedure that selects a key uniformly at random from among the keys in the hash table and returns it in expected time $O(L \cdot (1 + 1/\alpha))$.

11.3 Hash functions

For hashing to work well, it needs a good hash function. Along with being efficiently computable, what properties does a good hash function have? How do you design good hash functions?

This section first attempts to answer these questions based on two ad hoc approaches for creating hash functions: hashing by division and hashing by multiplication. Although these methods work well for some sets of input keys, they are limited because they try to provide a single fixed hash function that works well on any data—an approach called ***static hashing***.

We then see that provably good average-case performance for *any* data can be obtained by designing a suitable *family* of hash functions and choosing a hash function at random from this family at runtime, independent of the data to be hashed. The approach we examine is

called random hashing. A particular kind of random hashing, universal hashing, works well. As we saw with quicksort in Chapter 7, randomization is a powerful algorithmic design tool.

What makes a good hash function?

A good hash function satisfies (approximately) the assumption of independent uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other keys have hashed to. What does "equally likely" mean here? If the hash function is fixed, any probabilities would have to be based on the probability distribution of the input keys.

Unfortunately, you typically have no way to check this condition, unless you happen to know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently.

Occasionally you might know the distribution. For example, if you know that the keys are random real numbers k independently and uniformly distributed in the range $0 \le k < 1$, then the hash function

$$h(k) = \lfloor km \rfloor$$

satisfies the condition of independent uniform hashing.

A good static hashing approach derives the hash value in a way that you expect to be independent of any patterns that might exist in the data. For example, the "division method" (discussed in Section 11.3.1) computes the hash value as the remainder when the key is divided by a specified prime number. This method may give good results, if you (somehow) choose a prime number that is unrelated to any patterns in the distribution of keys.

Random hashing, described in Section 11.3.2, picks the hash function to be used at random from a suitable family of hashing functions. This approach removes any need to know anything about the probability distribution of the input keys, as the randomization necessary for good average-case behavior then comes from the (known) random process used to pick the hash function from the family of hash functions, rather than from the (unknown) process used to create the input keys. We recommend that you use random hashing.

Keys are integers, vectors, or strings

In practice, a hash function is designed to handle keys that are one of the following two types:

- A short nonnegative integer that fits in a w-bit machine word. Typical values for w would be 32 or 64.
- A short vector of nonnegative integers, each of bounded size. For example, each element might be an 8-bit byte, in which case the vector is often called a (byte) string. The vector might be of variable length.

To begin, we assume that keys are short nonnegative integers. Handling vector keys is more complicated and discussed in Sections 11.3.5 and 11.5.2.

11.3.1 Static hashing

Static hashing uses a single, fixed hash function. The only randomization available is through the (usually unknown) distribution of input keys. This section discusses two standard approaches for static hashing: the division method and the multiplication method. Although static hashing is no longer recommended, the multiplication method also provides a good foundation for "nonstatic" hashing—better known as random hashing—where the hash function is chosen at random from a suitable family of hash functions.

The division method

The ***division method*** for creating hash functions maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

$h(k) = k \bmod m$.

For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$. Since it requires only a single division operation, hashing by division is quite fast.

The division method may work well when m is a prime not too close to an exact power of 2. There is no guarantee that this method provides good average-case performance, however, and it may complicate applications since it constrains the size of the hash tables to be prime.

The multiplication method

The general ***multiplication method*** for creating hash functions operates in two steps. First, multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then, multiply this value by m and take the floor of the result. That is, the hash function is

$$h(k) = \lfloor m\,(kA \bmod 1) \rfloor,$$

where "kA mod 1" means the fractional part of kA, that is, $kA - \lfloor kA \rfloor$. The general multiplication method has the advantage that the value of m is not critical and you can choose it independently of how you choose the multiplicative constant A.

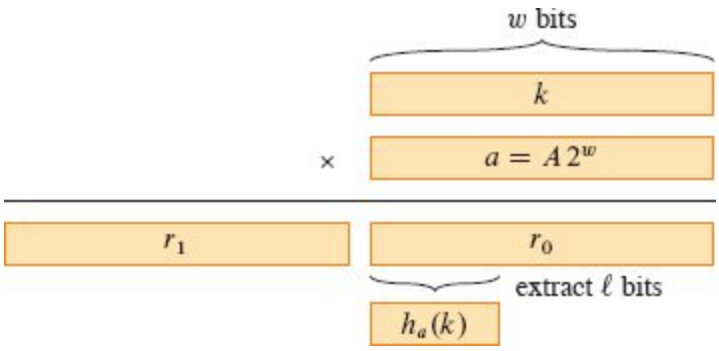

Figure 11.4 The multiply-shift method to compute a hash function. The w-bit representation of the key k is multiplied by the w-bit value $a = A \cdot 2^w$. The ℓ highest-order bits of the lower w-bit half of the product form the desired hash value $h_a(k)$.

The multiply-shift method

In practice, the multiplication method is best in the special case where the number m of hash-table slots is an exact power of 2, so that $m = 2^\ell$ for some integer ℓ, where $\ell \le w$ and w is the number of bits in a machine

word. If you choose a fixed w-bit positive integer $a = A\,2^w$, where $0 < A < 1$ as in the multiplication method so that a is in the range $0 < a < 2^w$, you can implement the function on most computers as follows. We assume that a key k fits into a single w-bit word.

Referring to Figure 11.4, first multiply k by the w-bit integer a. The result is a $2w$-bit value $r_1 2^w + r_0$, where r_1 is the high-order w-bit word of the product and r_0 is the low-order w-bit word of the product. The desired ℓ-bit hash value consists of the ℓ most significant bits of r_0. (Since r_1 is ignored, the hash function can be implemented on a computer that produces only a w-bit product given two w-bit inputs, that is, where the multiplication operation computes modulo 2^w.)

In other words, you define the hash function $h = h_a$, where

$$h_a(k) = (ka \bmod 2^w) \ggg (w - \ell) \tag{11.2}$$

for a fixed nonzero w-bit value a. Since the product ka of two w-bit words occupies $2w$ bits, taking this product modulo 2^w zeroes out the high-order w bits (r_1), leaving only the low-order w bits (r_0). The $\ggg$ operator performs a logical right shift by $w - \ell$ bits, shifting zeros into the vacated positions on the left, so that the ℓ most significant bits of r_0 move into the ℓ rightmost positions. (It's the same as dividing by $2^{w-\ell}$ and taking the floor of the result.) The resulting value equals the ℓ most significant bits of r_0. The hash function h_a can be implemented with three machine instructions: multiplication, subtraction, and logical right shift.

As an example, suppose that $k = 123456$, $\ell = 14$, $m = 2^{14} = 16384$, and $w = 32$. Suppose further that we choose $a = 2654435769$ (following a suggestion of Knuth [261]). Then $ka = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$, and so $r_1 = 76300$ and $r_0 = 17612864$. The 14 most significant bits of r_0 yield the value $h_a(k) = 67$.

Even though the multiply-shift method is fast, it doesn't provide any guarantee of good average-case performance. The universal hashing

approach presented in the next section provides such a guarantee. A simple randomized variant of the multiply-shift method works well on the average, when the program begins by picking a as a randomly chosen odd integer.

11.3.2 Random hashing

Suppose that a malicious adversary chooses the keys to be hashed by some fixed hash function. Then the adversary can choose n keys that all hash to the same slot, yielding an average retrieval time of $\Theta(n)$. Any static hash function is vulnerable to such terrible worst-case behavior. The only effective way to improve the situation is to choose the hash function *randomly* in a way that is *independent* of the keys that are actually going to be stored. This approach is called ***random hashing***. A special case of this approach, called ***universal hashing***, can yield provably good performance on average when collisions are handled by chaining, no matter which keys the adversary chooses.

To use random hashing, at the beginning of program execution you select the hash function at random from a suitable family of functions. As in the case of quicksort, randomization guarantees that no single input always evokes worst-case behavior. Because you randomly select the hash function, the algorithm can behave differently on each execution, even for the same set of keys to be hashed, guaranteeing good average-case performance.

Let $\mathscr{H}$ be a finite family of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m - 1\}$. Such a family is said to be ***universal*** if for each pair of distinct keys $k_1, k_2 \in U$, the number of hash functions $h \in \mathscr{H}$ for which $h(k_1) = h(k_2)$ is at most $|\mathscr{H}|/m$. In other words, with a hash function randomly chosen from $\mathscr{H}$, the chance of a collision between distinct keys k_1 and k_2 is no more than the chance $1/m$ of a collision if $h(k_1)$ and $h(k_2)$ were randomly and independently chosen from the set $\{0, 1, \ldots, m - 1\}$.

Independent uniform hashing is the same as picking a hash function uniformly at random from a family of m^n hash functions, each member

of that family mapping the n keys to the m hash values in a different way.

Every independent uniform random family of hash function is universal, but the converse need not be true: consider the case where $U = \{0, 1, \ldots, m - 1\}$ and the only hash function in the family is the identity function. The probability that two distinct keys collide is zero, even though each key is hashes to a fixed value.

The following corollary to Theorem 11.2 on page 279 says that universal hashing provides the desired payoff: it becomes impossible for an adversary to pick a sequence of operations that forces the worst-case running time.

Corollary 11.3
Using universal hashing and collision resolution by chaining in an initially empty table with m slots, it takes $\Theta(s)$ expected time to handle any sequence of s INSERT, SEARCH, and DELETE operations containing $n = O(m)$ INSERT operations.

Proof The INSERT and DELETE operations take constant time. Since the number n of insertions is $O(m)$, we have that $\alpha = O(1)$. Furthermore, the expected time for each SEARCH operation is $O(1)$, which can be seen by examining the proof of Theorem 11.2. That analysis depends only on collision probabilities, which are $1/m$ for any pair k_1, k_2 of keys by the choice of an independent uniform hash function in that theorem. Using a universal family of hash functions here instead of using independent uniform hashing changes the probability of collision from $1/m$ to at most $1/m$. By linearity of expectation, therefore, the expected time for the entire sequence of s operations is $O(s)$. Since each operation takes $\Omega(1)$ time, the $\Theta(s)$ bound follows.

■

11.3.3 Achievable properties of random hashing

There is a rich literature on the properties a family $\mathscr{H}$ of hash functions can have, and how they relate to the efficiency of hashing. We summarize a few of the most interesting ones here.

Let $\mathscr{H}$ be a family of hash functions, each with domain U and range $\{0, 1, \ldots, m-1\}$, and let h be any hash function that is picked uniformly at random from $\mathscr{H}$. The probabilities mentioned are probabilities over the picks of h.

- The family $\mathscr{H}$ is ***uniform*** if for any key k in U and any slot q in the range $\{0, 1, \ldots, m-1\}$, the probability that $h(k) = q$ is $1/m$.
- The family $\mathscr{H}$ is ***universal*** if for any distinct keys k_1 and k_2 in U, the probability that $h(k_1) = h(k_2)$ is at most $1/m$.
- The family $\mathscr{H}$ of hash functions is ***ϵ-universal*** if for any distinct keys k_1 and k_2 in U, the probability that $h(k_1) = h(k_2)$ is at most ϵ. Therefore, a universal family of hash functions is also $1/m$-universal.[2]
- The family $\mathscr{H}$ is ***d-independent*** if for any distinct keys $k_1, k_2, \ldots, k_d$ in U and any slots $q_1, q_2, \ldots, q_d$, not necessarily distinct, in $\{0, 1, \ldots, m-1\}$ the probability that $h(k_i) = q_i$ for $i = 1, 2, \ldots, d$ is $1/m^d$.

Universal hash-function families are of particular interest, as they are the simplest type supporting provably efficient hash-table operations for any input data set. Many other interesting and desirable properties, such as those noted above, are also possible and allow for efficient specialized hash-table operations.

11.3.4 Designing a universal family of hash functions

This section present two ways to design a universal (or ϵ-universal) family of hash functions: one based on number theory and another based on a randomized variant of the multiply-shift method presented in Section 11.3.1. The first method is a bit easier to prove universal, but the second method is newer and faster in practice.

A universal family of hash functions based on number theory

We can design a universal family of hash functions using a little number theory. You may wish to refer to Chapter 31 if you are unfamiliar with basic concepts in number theory.

Begin by choosing a prime number p large enough so that every possible key k lies in the range 0 to $p - 1$, inclusive. We assume here that p has a "reasonable" length. (See Section 11.3.5 for a discussion of methods for handling long input keys, such as variable-length strings.) Let $\mathbb{Z}_p$ denote the set $\{0, 1, \ldots, p - 1\}$, and let $\mathbb{Z}_p^*$ denote the set $\{1, 2, \ldots, p - 1\}$. Since p is prime, we can solve equations modulo p with the methods given in Chapter 31. Because the size of the universe of keys is greater than the number of slots in the hash table (otherwise, just use direct addressing), we have $p > m$.

Given any $a \in \mathbb{Z}_p^*$ and any $b \in \mathbb{Z}_p$, define the hash function h_{ab} as a linear transformation followed by reductions modulo p and then modulo m:

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m \ . \tag{11.3}$$

For example, with $p = 17$ and $m = 6$, we have

$$\begin{aligned} h_{3,4}(8) &= ((3 \cdot 8 + 4) \bmod 17) \bmod 6 \\ &= (28 \bmod 17) \bmod 6 \\ &= 11 \bmod 6 \\ &= 5. \end{aligned}$$

Given p and m, the family of all such hash functions is

$$\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\} \ . \tag{11.4}$$

Each hash function h_{ab} maps $\mathbb{Z}_p$ to $\mathbb{Z}_m$. This family of hash functions has the nice property that the size m of the output range (which is the size of the hash table) is arbitrary—it need not be prime. Since you can choose from among $p - 1$ values for a and p values for b, the family $\mathcal{H}_{pm}$ contains $p(p - 1)$ hash functions.

Theorem 11.4

The family $\mathscr{H}_{pm}$ of hash functions defined by equations (11.3) and (11.4) is universal.

Proof Consider two distinct keys k_1 and k_2 from $\mathbb{Z}_p$, so that $k_1 \neq k_2$. For a given hash function h_{ab}, let

$r_1 = (ak_1 + b) \bmod p$,

$r_2 = (ak_2 + b) \bmod p$.

We first note that $r_1 \neq r_2$. Why? Since we have $r_1 - r_2 = a(k_1 - k_2) \pmod p$, it follows that $r_1 \neq r_2$ because p is prime and both a and $(k_1 - k_2)$ are nonzero modulo p. By Theorem 31.6 on page 908, their product must also be nonzero modulo p. Therefore, when computing any $h_{ab} \in \mathscr{H}_{pm}$, distinct inputs k_1 and k_2 map to distinct values r_1 and r_2 modulo p, and there are no collisions yet at the "mod p level." Moreover, each of the possible $p(p - 1)$ choices for the pair (a, b) with $a \neq 0$ yields a *different* resulting pair (r_1, r_2) with $r_1 \neq r_2$, since we can solve for a and b given r_1 and r_2:

$a = ((r - r_2)((k_1 - k_2)^{-1} \bmod p)) \bmod p$,

$b = (r_1 - ak_1) \bmod p$,

where $((k_1 - k_2)^{-1} \bmod p)$ denotes the unique multiplicative inverse, modulo p, of $k_1 - k_2$. For each of the p possible values of r_1, there are only $p - 1$ possible values of r_2 that do not equal r_1, making only $p(p - 1)$ possible pairs (r_1, r_2) with $r_1 \neq r_2$. Therefore, there is a one-to-one correspondence between pairs (a, b) with $a \neq 0$ and pairs (r_1, r_2) with $r_1 \neq r_2$. Thus, for any given pair of distinct inputs k_1 and k_2, if we pick (a, b) uniformly at random from $\mathbb{Z}_p^* \times \mathbb{Z}_p$, the resulting pair (r_1, r_2) is equally likely to be any pair of distinct values modulo p.

Therefore, the probability that distinct keys k_1 and k_2 collide is equal to the probability that $r_1 = r_2 \pmod m$ when r_1 and r_2 are randomly

chosen as distinct values modulo p. For a given value of r_1, of the $p - 1$ possible remaining values for r_2, the number of values r_2 such that $r_2 \neq r_1$ and $r_2 = r_1 \pmod{m}$ is at most

$$\left\lceil \frac{p}{m} \right\rceil - 1 \le \frac{p+m-1}{m} - 1 \quad \text{(by inequality (3.7) on page 64)}$$
$$= \frac{p-1}{m} .$$

The probability that r_2 collides with r_1 when reduced modulo m is at most $((p - 1)/m)/(p - 1) = 1/m$, since r_2 is equally likely to be any of the $p - 1$ values in Z_p that are different from r_1, but at most $(p - 1)/m$ of those values are equivalent to r_1 modulo m.

Therefore, for any pair of distinct values $k_1, k_2 \in \mathbb{Z}_p$,

$$\Pr\{h_{ab}(k_1) = h_{ab}(k_2)\} \le 1/m,$$

so that $\mathscr{H}_{pm}$ is indeed universal.

■

A 2/*m*-universal family of hash functions based on the multiply-shift method

We recommend that in practice you use the following hash-function family based on the multiply-shift method. It is exceptionally efficient and (although we omit the proof) provably 2/*m*-universal. Define $\mathscr{H}$ to be the family of multiply-shift hash functions with odd constants a:

$$\mathcal{H} = \{h_a : a \text{ is odd}, 1 \le a < m, \text{ and } h_a \text{ is defined by equation (11.2)}\} . \qquad (11.5)$$

Theorem 11.5

The family of hash functions $\mathscr{H}$ given by equation (11.5) is 2/*m*-universal.

■

That is, the probability that any two distinct keys collide is at most 2/*m*. In many practical situations, the speed of computing the hash

function more than compensates for the higher upper bound on the probability that two distinct keys collide when compared with a universal hash function.

11.3.5 Hashing long inputs such as vectors or strings

Sometimes hash function inputs are so long that they cannot be easily encoded modulo a reasonably sized prime number p or encoded within a single word of, say, 64 bits. As an example, consider the class of vectors, such as vectors of 8-bit bytes (which is how strings in many programming languages are stored). A vector might have an arbitrary nonnegative length, in which case the length of the input to the hash function may vary from input to input.

Number-theoretic approaches

One way to design good hash functions for variable-length inputs is to extend the ideas used in Section 11.3.4 to design universal hash functions. Exercise 11.3-6 explores one such approach.

Cryptographic hashing

Another way to design a good hash function for variable-length inputs is to use a hash function designed for cryptographic applications. ***Cryptographic hash functions*** are complex pseudorandom functions, designed for applications requiring properties beyond those needed here, but are robust, widely implemented, and usable as hash functions for hash tables.

A cryptographic hash function takes as input an arbitrary byte string and returns a fixed-length output. For example, the NIST standard deterministic cryptographic hash function SHA-256 [346] produces a 256-bit (32-byte) output for any input.

Some chip manufacturers include instructions in their CPU architectures to provide fast implementations of some cryptographic functions. Of particular interest are instructions that efficiently implement rounds of the Advanced Encryption Standard (AES), the "AES-NI" instructions. These instructions execute in a few tens of nanoseconds, which is generally fast enough for use with hash tables. A

message authentication code such as CBC-MAC based on AES and the use of the AES-NI instructions could be a useful and efficient hash function. We don't pursue the potential use of specialized instruction sets further here.

Cryptographic hash functions are useful because they provide a way of implementing an approximate version of a random oracle. As noted earlier, a random oracle is equivalent to an independent uniform hash function family. From a theoretical point of view, a random oracle is an unachievable ideal: a deterministic function that provides a randomly selected output for each input. Because it is deterministic, it provides the same output if queried again for the same input. From a practical point of view, constructions of hash function families based on cryptographic hash functions are sensible substitutes for random oracles.

There are many ways to use a cryptographic hash function as a hash function. For example, we could define

$h(k) = \text{SHA-256}(k) \bmod m$.

To define a family of such hash functions one may prepend a "salt" string a to the input before hashing it, as in

$h_a(k) = \text{SHA-256}(a \parallel k) \bmod m$,

where $a \parallel k$ denotes the string formed by concatenating the strings a and k. The literature on message authentication codes (MACs) provides additional approaches.

Cryptographic approaches to hash-function design are becoming more practical as computers arrange their memories in hierarchies of differing capacities and speeds. Section 11.5 discusses one hash-function design based on the RC6 encryption method.

Exercises

11.3-1

You wish to search a linked list of length n, where each element contains a key k along with a hash value $h(k)$. Each key is a long character string.

How might you take advantage of the hash values when searching the list for an element with a given key?

11.3-2

You hash a string of r characters into m slots by treating it as a radix-128 number and then using the division method. You can represent the number m as a 32-bit computer word, but the string of r characters, treated as a radix-128 number, takes many words. How can you apply the division method to compute the hash value of the character string without using more than a constant number of words of storage outside the string itself?

11.3-3

Consider a version of the division method in which $h(k) = k \bmod m$, where $m = 2^p - 1$ and k is a character string interpreted in radix 2^p. Show that if string x can be converted to string y by permuting its characters, then x and y hash to the same value. Give an example of an application in which this property would be undesirable in a hash function.

11.3-4

Consider a hash table of size $m = 1000$ and a corresponding hash function $h(k) = \lfloor m\ (kA \bmod 1) \rfloor$ for $A = (\sqrt{5} - 1)/2$. Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped.

★ 11.3-5

Show that any ϵ-universal family $\mathscr{H}$ of hash functions from a finite set U to a finite set Q has $\epsilon \geq 1/|Q| - 1/|U|$.

★ 11.3-6

Let U be the set of d-tuples of values drawn from $\mathbb{Z}_p$, and let $Q = \mathbb{Z}_p$, where p is prime. Define the hash function $h_b : U \to Q$ for $b \in \mathbb{Z}_p$ on an input d-tuple $\langle a_0, a_1, \ldots, a_{d-1} \rangle$ from U as

$$h_b(\langle a_0, a_1, \ldots, a_{d-1}\rangle) = \left(\sum_{j=0}^{d-1} a_j b^j\right) \bmod p\ ,$$

and let $\mathscr{H} = \{h_b : b \in \mathbb{Z}_p\}$. Argue that $\mathscr{H}$ is ϵ-universal for $\epsilon = (d-1)/p$. (*Hint:* See Exercise 31.4-4.)

11.4 Open addressing

This section describes open addressing, a method for collision resolution that, unlike chaining, does not make use of storage outside of the hash table itself. In ***open addressing***, all elements occupy the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. No lists or elements are stored outside the table, unlike in chaining. Thus, in open addressing, the hash table can "fill up" so that no further insertions can be made. One consequence is that the load factor α can never exceed 1.

Collisions are handled as follows: when a new element is to be inserted into the table, it is placed in its "first-choice" location if possible. If that location is already occupied, the new element is placed in its "second-choice" location. The process continues until an empty slot is found in which to place the new element. Different elements have different preference orders for the locations.

To search for an element, systematically examine the preferred table slots for that element, in order of decreasing preference, until either you find the desired element or you find an empty slot and thus verify that the element is not in the table.

Of course, you could use chaining and store the linked lists inside the hash table, in the otherwise unused hash-table slots (see Exercise 11.2-4), but the advantage of open addressing is that it avoids pointers altogether. Instead of following pointers, you compute the sequence of slots to be examined. The memory freed by not storing pointers provides the hash table with a larger number of slots in the same amount of memory, potentially yielding fewer collisions and faster retrieval.

To perform insertion using open addressing, successively examine, or ***probe***, the hash table until you find an empty slot in which to put the key. Instead of being fixed in the order 0, 1, …, $m - 1$ (which implies a $\Theta(n)$ search time), the sequence of positions probed depends upon the key being inserted. To determine which slots to probe, the hash function includes the probe number (starting from 0) as a second input. Thus, the hash function becomes

$$h : U \times \{0, 1, \ldots, m - 1\} \rightarrow \{0, 1, \ldots, m - 1\}.$$

Open addressing requires that for every key k, the ***probe sequence*** $\langle h(k, 0), h(k, 1), \ldots, h(k, m - 1)\rangle$ be a permutation of $\langle 0, 1, \ldots, m - 1\rangle$, so that every hash-table position is eventually considered as a slot for a new key as the table fills up. The HASH-INSERT procedure on the following page assumes that the elements in the hash table T are keys with no satellite information: the key k is identical to the element containing key k. Each slot contains either a key or NIL (if the slot is empty). The HASH-INSERT procedure takes as input a hash table T and a key k that is assumed to be not already present in the hash table. It either returns the slot number where it stores key k or flags an error because the hash table is already full.

```
HASH-INSERT(T, k)
1 i = 0
2 repeat
3     q = h(k, i)
4     if T[q] == NIL
5         T[q] = k
6         return q
7     else i = i + 1
8 until i == m
9 error "hash table overflow"

HASH-SEARCH(T, k)
1  i = 0
2  repeat
```

```
3       q = h(k, i)
4       if T[q] == k
5           return q
6       i = i + 1
7   until T[q] == NIL or i == m
8   return NIL
```

The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted. Therefore, the search can terminate (unsuccessfully) when it finds an empty slot, since k would have been inserted there and not later in its probe sequence. The procedure HASH-SEARCH takes as input a hash table T and a key k, returning q if it finds that slot q contains key k, or NIL if key k is not present in table T.

Deletion from an open-address hash table is tricky. When you delete a key from slot q, it would be a mistake to mark that slot as empty by simply storing NIL in it. If you did, you might be unable to retrieve any key k for which slot q was probed and found occupied when k was inserted. One way to solve this problem is by marking the slot, storing in it the special value DELETED instead of NIL. The HASH-INSERT procedure then has to treat such a slot as empty so that it can insert a new key there. The HASH-SEARCH procedure passes over DELETED values while searching, since slots containing DELETED were filled when the key being searched for was inserted. Using the special value DELETED, however, means that search times no longer depend on the load factor α, and for this reason chaining is frequently selected as a collision resolution technique when keys must be deleted. There is a simple special case of open addressing, linear probing, that avoids the need to mark slots with DELETED. Section 11.5.1 shows how to delete from a hash table when using linear probing.

In our analysis, we assume ***independent uniform permutation hashing*** (also confusingly known as ***uniform hashing*** in the literature): the probe sequence of each key is equally likely to be any of the $m!$ permutations of $\langle 0, 1, \ldots, m - 1 \rangle$. Independent uniform permutation hashing generalizes the notion of independent uniform hashing defined earlier to

a hash function that produces not just a single slot number, but a whole probe sequence. True independent uniform permutation hashing is difficult to implement, however, and in practice suitable approximations (such as double hashing, defined below) are used.

We'll examine both double hashing and its special case, linear probing. These techniques guarantee that $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1)\rangle$ is a permutation of $\langle 0, 1, \ldots, m-1\rangle$ for each key k. (Recall that the second parameter to the hash function h is the probe number.) Neither double hashing nor linear probing meets the assumption of independent uniform permutation hashing, however. Double hashing cannot generate more than m^2 different probe sequences (instead of the $m!$ that independent uniform permutation hashing requires). Nonetheless, double hashing has a large number of possible probe sequences and, as you might expect, seems to give good results. Linear probing is even more restricted, capable of generating only m different probe sequences.

Double hashing

Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. ***Double hashing*** uses a hash function of the form

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m,$$

where both h_1 and h_2 are ***auxiliary hash functions***. The initial probe goes to position $T[h_1(k)]$, and successive probe positions are offset from previous positions by the amount $h_2(k)$, modulo m. Thus, the probe sequence here depends in two ways upon the key k, since the initial probe position $h_1(k)$, the step size $h_2(k)$, or both, may vary. Figure 11.5 gives an example of insertion by double hashing.

In order for the entire hash table to be searched, the value $h_2(k)$ must be relatively prime to the hash-table size m. (See Exercise 11.4-5.) A convenient way to ensure this condition is to let m be an exact power of 2 and to design h_2 so that it always produces an odd number. Another

way is to let m be prime and to design h_2 so that it always returns a positive integer less than m. For example, you could choose m prime and let

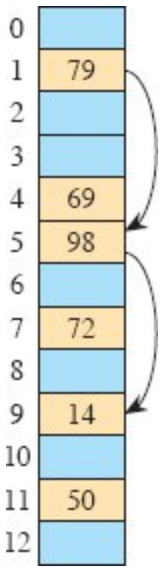

Figure 11.5 Insertion by double hashing. The hash table has size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 = 1 \pmod{13}$ and $14 = 3 \pmod{11}$, the key 14 goes into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

$$h_1(k) = k \bmod m,$$

$$h_2(k) = 1 + (k \bmod m'),$$

where m' is chosen to be slightly less than m (say, $m - 1$). For example, if $k = 123456$, $m = 701$, and $m' = 700$, then $h_1(k) = 80$ and $h_2(k) = 257$, so that the first probe goes to position 80, and successive probes examine every 257th slot (modulo m) until the key has been found or every slot has been examined.

Although values of m other than primes or exact powers of 2 can in principle be used with double hashing, in practice it becomes more difficult to efficiently generate $h_2(k)$ (other than choosing $h_2(k) = 1$, which gives linear probing) in a way that ensures that it is relatively prime to m, in part because the relative density $\phi(m)/m$ of such numbers for general m may be small (see equation (31.25) on page 921).

When m is prime or an exact power of 2, double hashing produces $\Theta(m^2)$ probe sequences, since each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence. As a result, for such values of m, double hashing appears to perform close to the "ideal" scheme of independent uniform permutation hashing.

Linear probing

Linear probing, a special case of double hashing, is the simplest open-addressing approach to resolving collisions. As with double hashing, an auxiliary hash function h_1 determines the first probe position $h_1(k)$ for inserting an element. If slot $T[h_1(k)]$ is already occupied, probe the next position $T[h_1(k) + 1]$. Keep going as necessary, on up to slot $T[m - 1]$, and then wrap around to slots $T[0]$, $T[1]$, and so on, but never going past slot $T[h_1(k) - 1]$. To view linear probing as a special case of double hashing, just set the double-hashing step function h_2 to be fixed at 1: $h_2(k) = 1$ for all k. That is, the hash function is

$$h(k, i) = (h_1(k) + i) \bmod m \tag{11.6}$$

for $i = 0, 1, \ldots, m - 1$. The value of $h_1(k)$ determines the entire probe sequence, and so assuming that $h_1(k)$ can take on any value in $\{0, 1, \ldots, m - 1\}$, linear probing allows only m distinct probe sequences.

We'll revisit linear probing in Section 11.5.1.

Analysis of open-address hashing

As in our analysis of chaining in Section 11.2, we analyze open addressing in terms of the load factor $\alpha = n/m$ of the hash table. With open addressing, at most one element occupies each slot, and thus $n \le m$, which implies $\alpha \le 1$. The analysis below requires α to be strictly less than 1, and so we assume that at least one slot is empty. Because deleting from an open-address hash table does not really free up a slot, we assume as well that no deletions occur.

For the hash function, we assume independent uniform permutation hashing. In this idealized scheme, the probe sequence $\langle h(k, 0), h(k, 1),$

…, $h(k, m - 1)\rangle$ used to insert or search for each key k is equally likely to be any permutation of $\langle 0, 1, \ldots, m - 1\rangle$. Of course, any given key has a unique fixed probe sequence associated with it. What we mean here is that, considering the probability distribution on the space of keys and the operation of the hash function on the keys, each possible probe sequence is equally likely.

We now analyze the expected number of probes for hashing with open addressing under the assumption of independent uniform permutation hashing, beginning with the expected number of probes made in an unsuccessful search (assuming, as stated above, that $\alpha < 1$).

The bound proven, of $1/(1 - \alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$, has an intuitive interpretation. The first probe always occurs. With probability approximately α, the first probe finds an occupied slot, so that a second probe happens. With probability approximately α^2, the first two slots are occupied so that a third probe ensues, and so on.

Theorem 11.6

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1 - \alpha)$, assuming independent uniform permutation hashing and no deletions.

Proof In an unsuccessful search, every probe but the last accesses an occupied slot that does not contain the desired key, and the last slot probed is empty. Let the random variable X denote the number of probes made in an unsuccessful search, and define the event A_i, for $i = 1, 2, \ldots$, as the event that an ith probe occurs and it is to an occupied slot. Then the event $\{X \geq i\}$ is the intersection of events $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$. We bound $\Pr\{X \geq i\}$ by bounding $\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$. By Exercise C.2-5 on page 1190,

$$\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\}.$$

Since there are n elements and m slots, $\Pr\{A_1\} = n/m$. For $j > 1$, the probability that there is a jth probe and it is to an occupied slot, given that the first $j - 1$ probes were to occupied slots, is $(n - j + 1)/(m - j + 1)$. This probability follows because the jth probe would be finding one of the remaining $(n - (j - 1))$ elements in one of the $(m - (j - 1))$ unexamined slots, and by the assumption of independent uniform permutation hashing, the probability is the ratio of these quantities. Since $n < m$ implies that $(n - j)/(m - j) \le n/m$ for all j in the range $0 \le j < m$, it follows that for all i in the range $1 \le i \le m$, we have

$$\begin{aligned}
\Pr\{X \ge i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\
&\le \left(\frac{n}{m}\right)^{i-1} \\
&= \alpha^{i-1} .
\end{aligned}$$

The product in the first line has $i - 1$ factors. When $i = 1$, the product is 1, the identity for multiplication, and we get $\Pr\{X \ge 1\} = 1$, which makes sense, since there must always be at least 1 probe. If each of the first n probes is to an occupied slot, then all occupied slots have been probed. Then, the $(n + 1)$st probe must be to an empty slot, which gives $\Pr\{X \ge i\} = 0$ for $i > n + 1$. Now, we use equation (C.28) on page 1193 to bound the expected number of probes:

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \ge i\} \\
&= \sum_{i=1}^{n+1} \Pr\{X \ge i\} + \sum_{i>n+1} \Pr\{X \ge i\} \\
&\le \sum_{i=1}^{\infty} \alpha^{i-1} + 0 \\
&= \sum_{i=0}^{\infty} \alpha^{i} \\
&= \frac{1}{1-\alpha} \qquad \text{(by equation (A.7) on page 1142 because } 0 \le \alpha < 1) .
\end{aligned}$$

■

If α is a constant, Theorem 11.6 predicts that an unsuccessful search runs in $O(1)$ time. For example, if the hash table is half full, the average number of probes in an unsuccessful search is at most $1/(1 - .5) = 2$. If it is 90% full, the average number of probes is at most $1/(1 - .9) = 10$.

Theorem 11.6 yields almost immediately how well the HASH-INSERT procedure performs.

Corollary 11.7
Inserting an element into an open-address hash table with load factor α, where $\alpha < 1$, requires at most $1/(1 - \alpha)$ probes on average, assuming independent uniform permutation hashing and no deletions.

Proof An element is inserted only if there is room in the table, and thus $\alpha < 1$. Inserting a key requires an unsuccessful search followed by placing the key into the first empty slot found. Thus, the expected number of probes is at most $1/(1 - \alpha)$.

■

It takes a little more work to compute the expected number of probes for a successful search.

Theorem 11.8
Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1 - \alpha},$$

assuming independent uniform permutation hashing with no deletions and assuming that each key in the table is equally likely to be searched for.

Proof A search for a key k reproduces the same probe sequence as when the element with key k was inserted. If k was the $(i + 1)$st key inserted into the hash table, then the load factor at the time it was inserted was i/m, and so by Corollary 11.7, the expected number of probes made in a search for k is at most $1/(1 - i/m) = m/(m - i)$. Averaging over all n keys in the hash table gives us the expected number of probes in a successful search:

$$
\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\
&\le \frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}\,dx \qquad \text{(by inequality (A.19) on page 1150)} \\
&= \frac{1}{\alpha}(\ln m - \ln(m-n)) \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}\,.
\end{aligned}
$$

■

If the hash table is half full, the expected number of probes in a successful search is less than 1.387. If the hash table is 90% full, the expected number of probes is less than 2.559. If $\alpha = 1$, then in an unsuccessful search, all m slots must be probed. Exercise 11.4-4 asks you to analyze a successful search when $\alpha = 1$.

Exercises

11.4-1

Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m = 11$ using open addressing. Illustrate the result of inserting these keys using linear probing with $h(k, i) = (k + i) \bmod m$ and using double hashing with $h_1(k) = k$ and $h_2(k) = 1 + (k \bmod (m - 1))$.

11.4-2

Write pseudocode for HASH-DELETE that fills the deleted key's slot with the special value DELETED, and modify HASH-SEARCH and HASH-INSERT as needed to handle DELETED.

11.4-3

Consider an open-address hash table with independent uniform permutation hashing and no deletions. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is 3/4 and when it is 7/8.

11.4-4
Show that the expected number of probes required for a successful search when $\alpha = 1$ (that is, when $n = m$), is H_m, the mth harmonic number.

★ 11.4-5
Show that, with double hashing, if m and $h_2(k)$ have greatest common divisor $d \geq 1$ for some key k, then an unsuccessful search for key k examines $(1/d)$th of the hash table before returning to slot $h_1(k)$. Thus, when $d = 1$, so that m and $h_2(k)$ are relatively prime, the search may examine the entire hash table. (*Hint:* See Chapter 31.)

★ 11.4-6
Consider an open-address hash table with a load factor α. Approximate the nonzero value α for which the expected number of probes in an unsuccessful search equals twice the expected number of probes in a successful search. Use the upper bounds given by Theorems 11.6 and 11.8 for these expected numbers of probes.

11.5 Practical considerations

Efficient hash table algorithms are not only of theoretical interest, but also of immense practical importance. Constant factors can matter. For this reason, this section discusses two aspects of modern CPUs that are not included in the standard RAM model presented in Section 2.2:

Memory hierarchies: The memory of modern CPUs has a number of levels, from the fast registers, through one or more levels of ***cache memory***, to the main-memory level. Each successive level stores more

data than the previous level, but access is slower. As a consequence, a complex computation (such as a complicated hash function) that works entirely within the fast registers can take less time than a single read operation from main memory. Furthermore, cache memory is organized in ***cache blocks*** of (say) 64 bytes each, which are always fetched together from main memory. There is a substantial benefit for ensuring that memory usage is local: reusing the same cache block is much more efficient than fetching a different cache block from main memory.

The standard RAM model measures efficiency of a hash-table operation by counting the number of hash-table slots probed. In practice, this metric is only a crude approximation to the truth, since once a cache block is in the cache, successive probes to that cache block are much faster than probes that must access main memory.

Advanced instruction sets: Modern CPUs may have sophisticated instruction sets that implement advanced primitives useful for encryption or other forms of cryptography. These instructions may be useful in the design of exceptionally efficient hash functions.

Section 11.5.1 discusses linear probing, which becomes the collision-resolution method of choice in the presence of a memory hierarchy. Section 11.5.2 suggests how to construct "advanced" hash functions based on cryptographic primitives, suitable for use on computers with hierarchical memory models.

11.5.1 Linear probing

Linear probing is often disparaged because of its poor performance in the standard RAM model. But linear probing excels for hierarchical memory models, because successive probes are usually to the same cache block of memory.

Deletion with linear probing

Another reason why linear probing is often not used in practice is that deletion seems complicated or impossible without using the special DELETED value. Yet we'll now see that deletion from a hash table

based on linear probing is not all that difficult, even without the DELETED marker. The deletion procedure works for linear probing, but not for open-address probing in general, because with linear probing keys all follow the same simple cyclic probing sequence (albeit with different starting points).

The deletion procedure relies on an "inverse" function to the linear-probing hash function $h(k, i) = (h_1(k) + i) \bmod m$, which maps a key k and a probe number i to a slot number in the hash table. The inverse function g maps a key k and a slot number q, where $0 \le q < m$, to the probe number that reaches slot q:

$$g(k, q) = (q - h_1(k)) \bmod m.$$

If $h(k, i) = q$, then $g(k, q) = i$, and so $h(k, g(k, q)) = q$.

The procedure LINEAR-PROBING-HASH-DELETE on the facing page deletes the key stored in position q from hash table T. Figure 11.6 shows how it works. The procedure first deletes the key in position q by setting $T[q]$ to NIL in line 2. It then searches for a slot q' (if any) that contains a key that should be moved to the slot q just vacated by key k. Line 9 asks the critical question: does the key k' in slot q' need to be moved to the vacated slot q in order to preserve the accessibility of k'? If $g(k', q) < g(k', q')$, then during the insertion of k' into the table, slot q was examined but found to be already occupied. But now slot q, where a search will look for k', is empty. In this case, key k' moves to slot q in line 10, and the search continues, to see whether any later key also needs to be moved to the slot q' that was just freed up when k' moved.

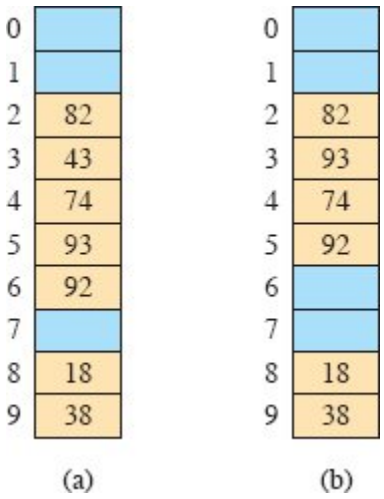

Figure 11.6 Deletion in a hash table that uses linear probing. The hash table has size 10 with $h_1(k) = k \bmod 10$. **(a)** The hash table after inserting keys in the order 74, 43, 93, 18, 82, 38, 92. **(b)** The hash table after deleting the key 43 from slot 3. Key 93 moves up to slot 3 to keep it accessible, and then key 92 moves up to slot 5 just vacated by key 93. No other keys need to be moved.

```
LINEAR-PROBING-HASH-DELETE(T, q)
 1 while TRUE
 2     T[q] = NIL                    // make slot q empty
 3     q' = q                        // starting point for search
 4     repeat
 5         q' = (q' + 1) mod m       // next slot number with linear probing
 6         k' = T[q']                // next key to try to move
 7         if k' == NIL
 8             return                // return when an empty slot is found
 9     until g(k', q) < g(k', q')    // was empty slot q probed before q'?
10     T[q] = k'                     // move k' into slot q
11     q = q'                        // free up slot q'
```

Analysis of linear probing

Linear probing is popular to implement, but it exhibits a phenomenon known as ***primary clustering***. Long runs of occupied slots build up,

increasing the average search time. Clusters arise because an empty slot preceded by i full slots gets filled next with probability $(i + 1)/m$. Long runs of occupied slots tend to get longer, and the average search time increases.

In the standard RAM model, primary clustering is a problem, and general double hashing usually performs better than linear probing. By contrast, in a hierarchical memory model, primary clustering is a beneficial property, as elements are often stored together in the same cache block. Searching proceeds through one cache block before advancing to search the next cache block. With linear probing, the running time for a key k of HASH-INSERT, HASH-SEARCH, or LINEAR-PROBING-HASH-DELETE is at most proportional to the distance from $h_1(k)$ to the next empty slot.

The following theorem is due to Pagh et al. [351]. A more recent proof is given by Thorup [438]. We omit the proof here. The need for 5-independence is by no means obvious; see the cited proofs.

Theorem 11.9

If h_1 is 5-independent and $\alpha \leq 2/3$, then it takes expected constant time to search for, insert, or delete a key in a hash table using linear probing.

■

(Indeed, the expected operation time is $O(1/\epsilon^2)$ for $\alpha = 1 - \epsilon$.)

★ 11.5.2 Hash functions for hierarchical memory models

This section illustrates an approach for designing efficient hash tables in a modern computer system having a memory hierarchy.

Because of the memory hierarchy, linear probing is a good choice for resolving collisions, as probe sequences are sequential and tend to stay within cache blocks. But linear probing is most efficient when the hash function is complex (for example, 5-independent as in Theorem 11.9). Fortunately, having a memory hierarchy means that complex hash functions can be implemented efficiently.

As noted in Section 11.3.5, one approach is to use a cryptographic hash function such as SHA-256. Such functions are complex and

sufficiently random for hash table applications. On machines with specialized instructions, cryptographic functions can be quite efficient.

Instead, we present here a simple hash function based only on addition, multiplication, and swapping the halves of a word. This function can be implemented entirely within the fast registers, and on a machine with a memory hierarchy, its latency is small compared with the time taken to access a random slot of the hash table. It is related to the RC6 encryption algorithm and can for practical purposes be considered a "random oracle."

The wee hash function

Let w denote the word size of the machine (e.g., $w = 64$), assumed to be even, and let a and b be w-bit unsigned (nonnegative) integers such that a is odd. Let swap(x) denote the w-bit result of swapping the two $w/2$-bit halves of w-bit input x. That is,

$$\text{swap}(x) = (x \ggg (w/2)) + (x \lll (w/2))$$

where "$\ggg$" is "logical right shift" (as in equation (11.2)) and "$\lll$ is "left shift." Define

$$f_a(k) = \text{swap}((2k^2 + ak) \bmod 2^w).$$

Thus, to compute $f_a(k)$, evaluate the quadratic function $2k^2 + ak$ modulo 2^w and then swap the left and right halves of the result.

Let r denote a desired number of "rounds" for the computation of the hash function. We'll use $r = 4$, but the hash function is well defined for any nonnegative r. Denote by $f_a^{(r)}(k)$ the result of iterating f_a a total of r times (that is, r rounds) starting with input value k. For any odd a and any $r \geq 0$, the function $f_a^{(r)}$, although complicated, is one-to-one (see Exercise 11.5-1). A cryptographer would view $f_a^{(r)}$ as a simple block cipher operating on w-bit input blocks, with r rounds and key a.

We first define the wee hash function h for short inputs, where by "short" we means "whose length t is at most w-bits," so that the input fits within one computer word. We would like inputs of different lengths

to be hashed differently. The ***wee hash function*** $h_{a,b,t,r}(k)$ for parameters a, b, and r on t-bit input k is defined as

$$h_{a,b,t,r}(k) = \left(f^{(r)}_{a+2t}(k+b)\right) \bmod m \ . \tag{11.7}$$

That is, the hash value for t-bit input k is obtained by applying $f^{(r)}_{a+2t}$ to $k + b$, then taking the final result modulo m. Adding the value b provides hash-dependent randomization of the input, in a way that ensures that for variable-length inputs the 0-length input does not have a fixed hash value. Adding the value $2t$ to a ensures that the hash function acts differently for inputs of different lengths. (We use $2t$ rather than t to ensure that the key $a + 2t$ is odd if a is odd.) We call this hash function "wee" because it uses a tiny amount of memory—more precisely, it can be implemented efficiently using only the computer's fast registers. (This hash function does not have a name in the literature; it is a variant we developed for this textbook.)

Speed of the wee hash function

It is surprising how much efficiency can be bought with locality. Experiments (unpublished, by the authors) suggest that evaluating the wee hash function takes less time than probing a *single* randomly chosen slot in a hash table. These experiments were run on a laptop (2019 MacBook Pro) with $w = 64$ and $a = 123$. For large hash tables, evaluating the wee hash function was 2 to 10 times faster than performing a single probe of the hash table.

The wee hash function for variable-length inputs

Sometimes inputs are long—more than one w-bit word in length—or have variable length, as discussed in Section 11.3.5. We can extend the wee hash function, defined above for inputs that are at most single w-bit word in length, to handle long or variable-length inputs. Here is one method for doing so.

Suppose that an input k has length t (measured in bits). Break k into a sequence $\langle k_1, k_2, \ldots, k_u \rangle$ of w-bit words, where $u = \lceil t/w \rceil$, k_1 contains the least-significant w bits of k, and k_u contains the most significant

bits. If t is not a multiple of w, then k_u contains fewer than w bits, in which case, pad out the unused high-order bits of k_u with 0-bits. Define the function chop to return a sequence of the w-bit words in k:

$$\text{chop}(k) = \langle k_1, k_2, \ldots, k_u \rangle.$$

The most important property of the chop operation is that it is one-to-one, given t: for any two t-bit keys k and k', if $k \neq k'$ then $\text{chop}(k) \neq \text{chop}(k')$, and k can be derived from $\text{chop}(k)$ and t. The chop operation also has the useful property that a single-word input key yields a single-word output sequence: $\text{chop}(k) = \langle k \rangle$.

With the chop function in hand, we specify the wee hash function $h_{a,b,t,r}(k)$ for an input k of length t bits as follows:

$$h_{a,b,t,r}(k) = \text{WEE}(k, a, b, t, r, m),$$

where the procedure WEE defined on the facing page iterates through the elements of the w-bit words returned by $\text{chop}(k)$, applying f_a^r to the sum of the current word k_i and the previously computed hash value so far, finally returning the result obtained modulo m. This definition for variable-length and long (multiple-word) inputs is a consistent extension of the definition in equation (11.7) for short (single-word) inputs. For practical use, we recommend that a be a randomly chosen odd w-bit word, b be a randomly chosen w-bit word, and that $r = 4$.

Note that the wee hash function is really a hash function family, with individual hash functions determined by parameters a, b, t, r, and m. The (approximate) 5-independence of the wee hash function family for variable-length inputs can be argued based on the assumption that the 1-word wee hash function is a random oracle and on the security of the cipher-block-chaining message authentication code (CBC-MAC), as studied by Bellare et al. [42]. The case here is actually simpler than that studied in the literature, since if two messages have different lengths t and t', then their "keys" are different: $a + 2t \neq a + 2t'$. We omit the details.

WEE(k, a, b, t, r, m)

1 $u = \lceil t/w \rceil$
2 $\langle k_1, k_2, \ldots, k_u \rangle = \text{chop}(k)$
3 $q = b$
4 **for** $i = 1$ **to** u
5 $\quad q = f^{(r)}_{a+2t}(k_i + q)$
6 **return** $q \bmod m$

This definition of a cryptographically inspired hash-function family is meant to be realistic, yet only illustrative, and many variations and improvements are possible. See the chapter notes for suggestions.

In summary, we see that when the memory system is hierarchical, it becomes advantageous to use linear probing (a special case of double hashing), since successive probes tend to stay in the same cache block. Furthermore, hash functions that can be implemented using only the computer's fast registers are exceptionally efficient, so they can be quite complex and even cryptographically inspired, providing the high degree of independence needed for linear probing to work most efficiently.

Exercises

★ *11.5-1*

Complete the argument that for any odd positive integer a and any integer $r \geq 0$, the function $f^{(r)}_a$ is one-to-one. Use a proof by contradiction and make use of the fact that the function f_a works modulo 2^w.

★ *11.5-2*

Argue that a random oracle is 5-independent.

★ *11.5-3*

Consider what happens to the value $f^{(r)}_a(k)$ as we flip a single bit k_i of the input value k, for various values of r. Let $k = \sum_{i=0}^{w-1} k_i 2^i$ and $g_a(k) = \sum_{j=0}^{w-1} b_j 2^j$ define the bit values k_i in the input (with k_0 the least-

significant bit) and the bit values b_j in $g_a(k) = (2k^2 + ak) \bmod 2^w$ (where $g_a(k)$ is the value that, when its halves are swapped, becomes $f_a(k)$). Suppose that flipping a single bit k_i of the input k may cause any bit b_j of $g_a(k)$ to flip, for $j \geq i$. What is the least value of r for which flipping the value of any single bit k_i may cause *any* bit of the output $f_a^{(r)}(k)$ to flip? Explain.

Problems

11-1 Longest-probe bound for hashing

Suppose you are using an open-addressed hash table of size m to store $n \leq m/2$ items.

a. Assuming independent uniform permutation hashing, show that for $i = 1, 2, \ldots, n$, the probability is at most 2^{-p} that the ith insertion requires strictly more than p probes.

b. Show that for $i = 1, 2, \ldots, n$, the probability is $O(1/n^2)$ that the ith insertion requires more than $2 \lg n$ probes.

Let the random variable X_i denote the number of probes required by the ith insertion. You have shown in part (b) that $\Pr\{X_i > 2 \lg n\} = O(1/n^2)$. Let the random variable $X = \max\{X_i : 1 \leq i \leq n\}$ denote the maximum number of probes required by any of the n insertions.

c. Show that $\Pr\{X > 2 \lg n\} = O(1/n)$.

d. Show that the expected length $\mathrm{E}[X]$ of the longest probe sequence is $O(\lg n)$.

11-2 Searching a static set

You are asked to implement a searchable set of n elements in which the keys are numbers. The set is static (no INSERT or DELETE operations), and the only operation required is SEARCH. You are given

an arbitrary amount of time to preprocess the n elements so that SEARCH operations run quickly.

a. Show how to implement SEARCH in $O(\lg n)$ worst-case time using no extra storage beyond what is needed to store the elements of the set themselves.

b. Consider implementing the set by open-address hashing on m slots, and assume independent uniform permutation hashing. What is the minimum amount of extra storage $m - n$ required to make the average performance of an unsuccessful SEARCH operation be at least as good as the bound in part (a)? Your answer should be an asymptotic bound on $m - n$ in terms of n.

11-3 Slot-size bound for chaining

Given a hash table with n slots, with collisions resolved by chaining, suppose that n keys are inserted into the table. Each key is equally likely to be hashed to each slot. Let M be the maximum number of keys in any slot after all the keys have been inserted. Your mission is to prove an $O(\lg n / \lg \lg n)$ upper bound on $\mathrm{E}[M]$, the expected value of M.

a. Argue that the probability Q_k that exactly k keys hash to a particular slot is given by

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k} .$$

b. Let P_k be the probability that $M = k$, that is, the probability that the slot containing the most keys contains k keys. Show that $P_k \le nQ_k$.

c. Show that $Q_k < e^k/k^k$. *Hint:* Use Stirling's approximation, equation (3.25) on page 67.

d. Show that there exists a constant $c > 1$ such that $Q_{k_0} < 1/n^3$ for $k_0 = c \lg n / \lg \lg n$. Conclude that $P_k < 1/n^2$ for $k \ge k_0 = c \lg n / \lg \lg n$.

e. Argue that

$$\mathrm{E}[M] \le \Pr\left\{M > \frac{c \lg n}{\lg \lg n}\right\} \cdot n + \Pr\left\{M \le \frac{c \lg n}{\lg \lg n}\right\} \cdot \frac{c \lg n}{\lg \lg n} .$$

Conclude that $\mathrm{E}[M] = O(\lg n / \lg \lg n)$.

11-4 Hashing and authentication

Let $\mathscr{H}$ be a family of hash functions in which each hash function $h \in \mathscr{H}$ maps the universe U of keys to $\{0, 1, \ldots, m - 1\}$.

a. Show that if the family $\mathscr{H}$ of hash functions is 2-independent, then it is universal.

b. Suppose that the universe U is the set of n-tuples of values drawn from $\mathbb{Z}_p = \{0, 1, \ldots, p - 1\}$, where p is prime. Consider an element $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle \in U$. For any n-tuple $a = \langle a_0, a_1, \ldots, a_{n-1} \rangle \in U$, define the hash function h_a by

$$h_a(x) = \left(\sum_{j=0}^{n-1} a_j x_j \right) \bmod p .$$

Let $\mathscr{H} = \{h_a : a \in U\}$. Show that $\mathscr{H}$ is universal, but not 2-independent. (*Hint:* Find a key for which all hash functions in $\mathscr{H}$ produce the same value.)

c. Suppose that we modify $\mathscr{H}$ slightly from part (b): for any $a \in U$ and for any $b \in \mathbb{Z}_p$, define

$$h'_{ab}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b \right) \bmod p$$

and $\mathscr{H}' = \{h'_{ab} : a \in U \text{ and } b \in \mathbb{Z}_p\}$. Argue that $\mathscr{H}'$ is 2-independent. (*Hint:* Consider fixed n-tuples $x \in U$ and $y \in U$, with $x_i \neq y_i$ for some i. What happens to $h'_{ab}(x)$ and $h'_{ab}(y)$ as a_i and b range over $\mathbb{Z}_p$?)

d. Alice and Bob secretly agree on a hash function h from a 2-independent family $\mathscr{H}$ of hash functions. Each $h \in \mathscr{H}$ maps from a

universe of keys U to $\mathbb{Z}_p$, where p is prime. Later, Alice sends a message m to Bob over the internet, where $m \in U$. She authenticates this message to Bob by also sending an authentication tag $t = h(m)$, and Bob checks that the pair (m, t) he receives indeed satisfies $t = h(m)$. Suppose that an adversary intercepts (m, t) en route and tries to fool Bob by replacing the pair (m, t) with a different pair (m', t'). Argue that the probability that the adversary succeeds in fooling Bob into accepting (m', t') is at most $1/p$, no matter how much computing power the adversary has, even if the adversary knows the family $\mathscr{H}$ of hash functions used.

Chapter notes

The books by Knuth [261] and Gonnet and Baeza-Yates [193] are excellent references for the analysis of hashing algorithms. Knuth credits H. P. Luhn (1953) for inventing hash tables, along with the chaining method for resolving collisions. At about the same time, G. M. Amdahl originated the idea of open addressing. The notion of a random oracle was introduced by Bellare et al. [43]. Carter and Wegman [80] introduced the notion of universal families of hash functions in 1979.

Dietzfelbinger et al. [113] invented the multiply-shift hash function and gave a proof of Theorem 11.5. Thorup [437] provides extensions and additional analysis. Thorup [438] gives a simple proof that linear probing with 5-independent hashing takes constant expected time per operation. Thorup also describes the method for deletion in a hash table using linear probing.

Fredman, Komlós, and Szemerédi [154] developed a perfect hashing scheme for static sets—"perfect" because all collisions are avoided. An extension of their method to dynamic sets, handling insertions and deletions in amortized expected time $O(1)$, has been given by Dietzfelbinger et al. [114].

The wee hash function is based on the RC6 encryption algorithm [379]. Leiserson et al. [292] propose an "RC6MIX" function that is essentially the same as the wee hash function. They give experimental

evidence that it has good randomness, and they also give a "DOTMIX" function for dealing with variable-length inputs. Bellare et al. [42] provide an analysis of the security of the cipher-block-chaining message authentication code. This analysis implies that the wee hash function has the desired pseudorandomness properties.

---

[1] The definition of "average-case" requires care—are we assuming an input distribution over the keys, or are we randomizing the choice of hash function itself? We'll consider both approaches, but with an emphasis on the use of a randomly chosen hash function.

[2] In the literature, a (c/m)-universal hash function is sometimes called c-universal or c-approximately universal. We'll stick with the notation (c/m)-universal.

12 Binary Search Trees

The search tree data structure supports each of the dynamic-set operations listed on page 250: SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. Thus, you can use a search tree both as a dictionary and as a priority queue.

Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in $\Theta(\lg n)$ worst-case time. If the tree is a linear chain of n nodes, however, the same operations take $\Theta(n)$ worst-case time. In Chapter 13, we'll see a variation of binary search trees, red-black trees, whose operations guarantee a height of $O(\lg n)$. We won't prove it here, but if you build a binary search tree on a random set of n keys, its expected height is $O(\lg n)$ even if you don't try to limit its height.

After presenting the basic properties of binary search trees, the following sections show how to walk a binary search tree to print its values in sorted order, how to search for a value in a binary search tree, how to find the minimum or maximum element, how to find the predecessor or successor of an element, and how to insert into or delete from a binary search tree. The basic mathematical properties of trees appear in Appendix B.

12.1 What is a binary search tree?

A binary search tree is organized, as the name suggests, in a binary tree, as shown in Figure 12.1. You can represent such a tree with a linked

data structure, as in Section 10.3. In addition to a *key* and satellite data, each node object contains attributes *left*, *right*, and *p* that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate attribute contains the value NIL. The tree itself has an attribute *root* that points to the root node, or NIL if the tree is empty. The root node *T.root* is the only node in a tree *T* whose parent is NIL.

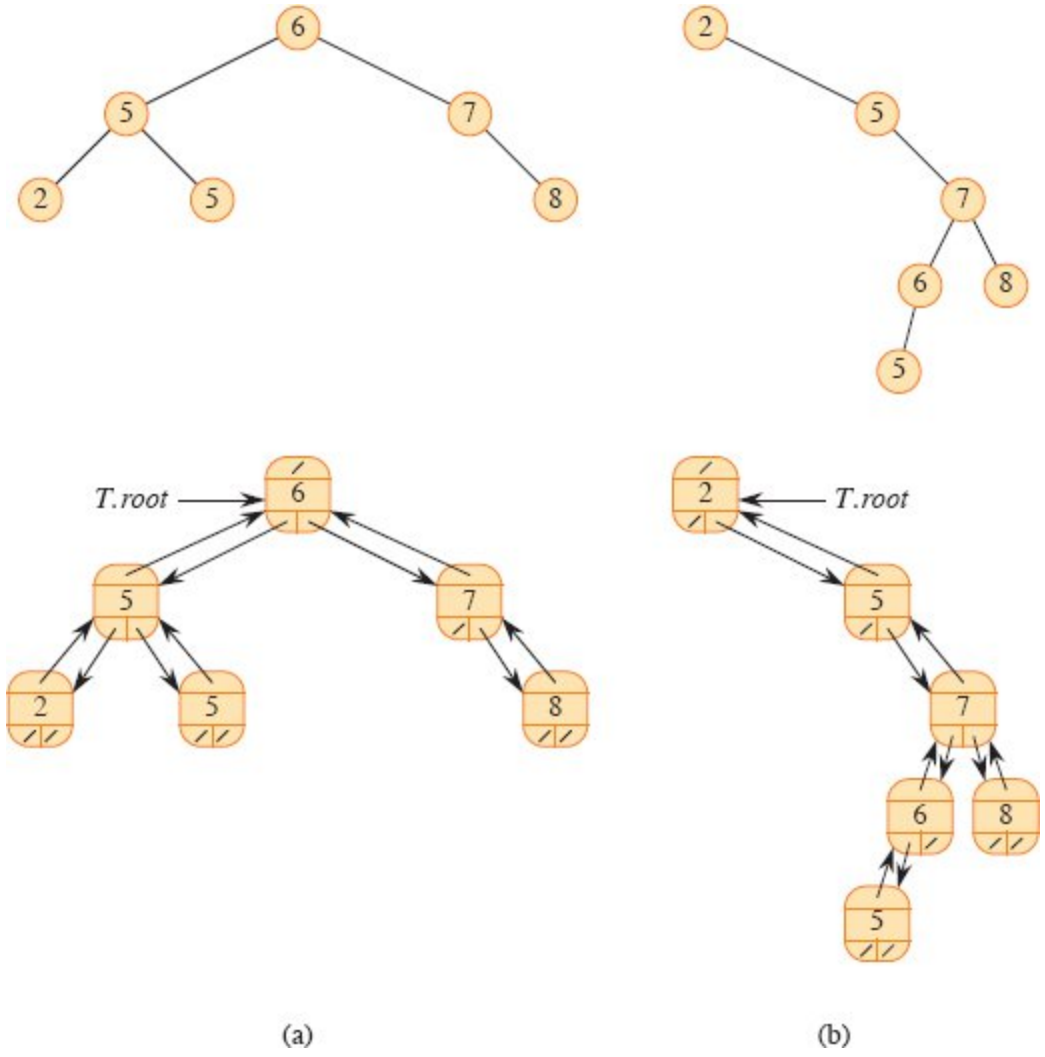

Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. The top figure shows how to view the tree conceptually, and the bottom figure shows the *left*, *right*, and p attributes in each node, in the style of Figure 10.6 on page 266. **(b)** A less efficient binary search tree, with height 4, that contains the same keys.

The keys in a binary search tree are always stored in such a way as to satisfy the ***binary-search-tree property***:

> Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \le x.key$. If y is a node in the right subtree of x, then $y.key \ge x.key$.

Thus, in Figure 12.1(a), the key of the root is 6, the keys 2, 5, and 5 in its left subtree are no larger than 6, and the keys 7 and 8 in its right subtree are no smaller than 6. The same property holds for every node in the tree. For example, looking at the root's left child as the root of a subtree, this subtree root has the key 5, the key 2 in its left subtree is no larger than 5, and the key 5 in its right subtree is no smaller than 5.

Because of the binary-search-tree property, you can print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an ***inorder tree walk***, given by the procedure INORDER-TREE-WALK. This algorithm is so named because it prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree. (Similarly, a ***preorder tree walk*** prints the root before the values in either subtree, and a ***postorder tree walk*** prints the root after the values in its subtrees.) To print all the elements in a binary search tree T, call INORDER-TREE-WALK($T.root$). For example, the inorder tree walk prints the keys in each of the two binary search trees from Figure 12.1 in the order 2, 5, 5, 6, 7, 8. The correctness of the algorithm follows by induction directly from the binary-search-tree property.

```
INORDER-TREE-WALK(x)
1 if x ≠ NIL
2     INORDER-TREE-WALK(x.left)
3     print x.key
4     INORDER-TREE-WALK(x.right)
```

It takes $\Theta(n)$ time to walk an n-node binary search tree, since after the initial call, the procedure calls itself recursively exactly twice for each node in the tree—once for its left child and once for its right child. The following theorem gives a formal proof that it takes linear time to perform an inorder tree walk.

Theorem 12.1

If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ time.

Proof Let $T(n)$ denote the time taken by INORDER-TREE-WALK when it is called on the root of an n-node subtree. Since INORDER-TREE-WALK visits all n nodes of the subtree, we have $T(n) = \Omega(n)$. It remains to show that $T(n) = O(n)$.

Since INORDER-TREE-WALK takes a small, constant amount of time on an empty subtree (for the test $x \neq \text{NIL}$), we have $T(0) = c$ for some constant $c > 0$.

For $n > 0$, suppose that INORDER-TREE-WALK is called on a node x whose left subtree has k nodes and whose right subtree has $n - k - 1$ nodes. The time to perform INORDER-TREE-WALK(x) is bounded by $T(n) \leq T(k) + T(n - k - 1) + d$ for some constant $d > 0$ that reflects an upper bound on the time to execute the body of INORDER-TREE-WALK(x), exclusive of the time spent in recursive calls.

We use the substitution method to show that $T(n) = O(n)$ by proving that $T(n) \leq (c + d)n + c$. For $n = 0$, we have $(c + d) \cdot 0 + c = c = T(0)$. For $n > 0$, we have

$$\begin{aligned} T(n) &\leq T(k) + T(n - k - 1) + d \\ &\leq ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\ &= (c + d)n + c - (c + d) + c + d \\ &= (c + d)n + c, \end{aligned}$$

which completes the proof.

■

Exercises

12.1-1

For the set {1, 4, 5, 10, 16, 17, 21} of keys, draw binary search trees of heights 2, 3, 4, 5, and 6.

12.1-2

What is the difference between the binary-search-tree property and the min-heap property on page 163? Can the min-heap property be used to print out the keys of an n-node tree in sorted order in $O(n)$ time? Show how, or explain why not.

12.1-3
Give a nonrecursive algorithm that performs an inorder tree walk. (*Hint:* An easy solution uses a stack as an auxiliary data structure. A more complicated, but elegant, solution uses no stack but assumes that you can test two pointers for equality.)

12.1-4
Give recursive algorithms that perform preorder and postorder tree walks in $\Theta(n)$ time on a tree of n nodes.

12.1-5
Argue that since sorting n elements takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \lg n)$ time in the worst case.

12.2 Querying a binary search tree

Binary search trees can support the queries MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR, as well as SEARCH. This section examines these operations and shows how to support each one in $O(h)$ time on any binary search tree of height h.

Searching

To search for a node with a given key in a binary search tree, call the TREE-SEARCH procedure. Given a pointer x to the root of a subtree and a key k, TREE-SEARCH(x, k) returns a pointer to a node with key k if one exists in the subtree; otherwise, it returns NIL. To search for key k in the entire binary search tree T, call TREE-SEARCH($T.root$, k).

```
TREE-SEARCH(x, k)
1 if x == NIL or k == x.key
2     return x
3 if k < x.key
4     return TREE-SEARCH(x.left, k)
```

```
5 else return TREE-SEARCH(x.right, k)

ITERATIVE-TREE-SEARCH(x, k)
1 while x ≠ NIL and k ≠ x.key
2     if k < x.key
3         x = x.left
4     else x = x.right
5 return x
```

The TREE-SEARCH procedure begins its search at the root and traces a simple path downward in the tree, as shown in Figure 12.2(a). For each node x it encounters, it compares the key k with $x.key$. If the two keys are equal, the search terminates. If k is smaller than $x.key$, the search continues in the left subtree of x, since the binary-search-tree property implies that k cannot reside in the right subtree. Symmetrically, if k is larger than $x.key$, the search continues in the right subtree. The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of TREE-SEARCH is $O(h)$, where h is the height of the tree.

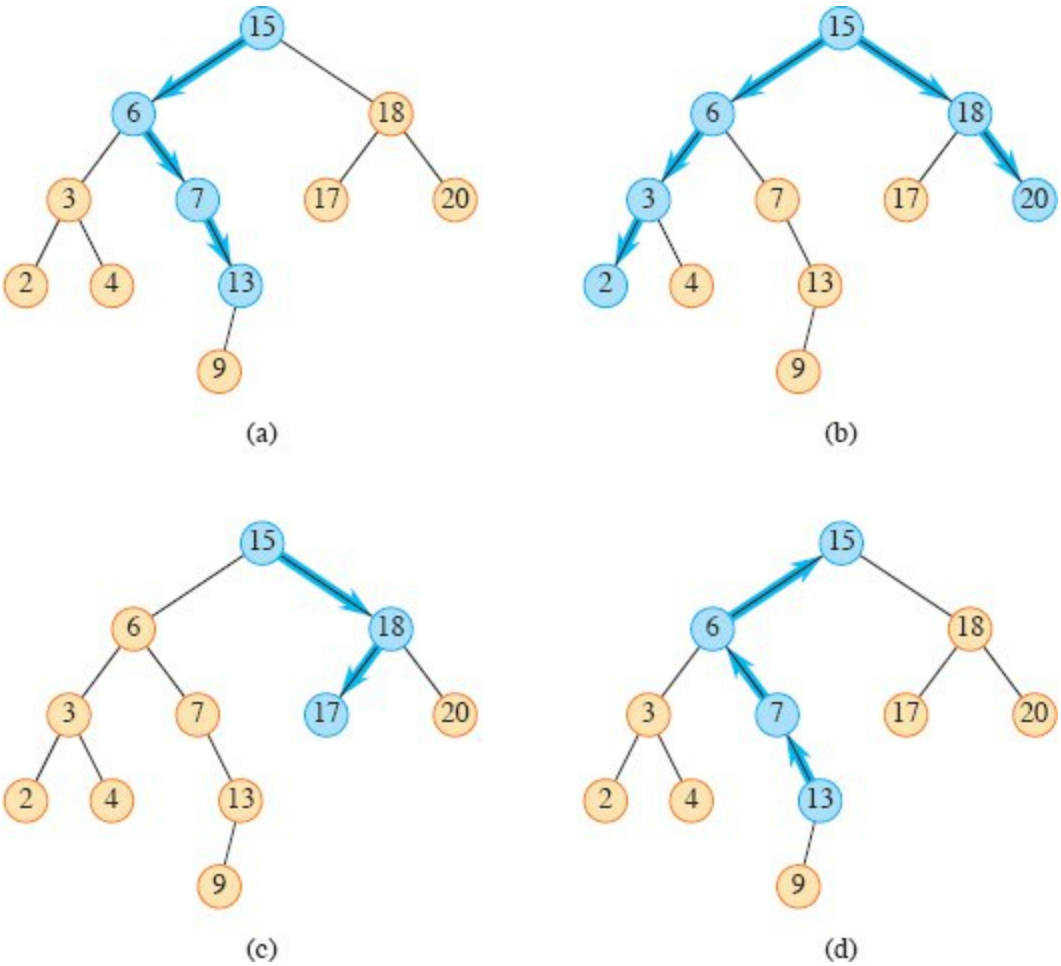

Figure 12.2 Queries on a binary search tree. Nodes and paths followed in each query are colored blue. **(a)** A search for the key 13 in the tree follows the path 15 → 6 → 7 → 13 from the root. **(b)** The minimum key in the tree is 2, which is found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. **(c)** The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. **(d)** The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

Since the TREE-SEARCH procedure recurses on either the left subtree or the right subtree, but not both, we can rewrite the algorithm to "unroll" the recursion into a **while** loop. On most computers, the ITERATIVE-TREE-SEARCH procedure on the facing page is more efficient.

Minimum and maximum

To find an element in a binary search tree whose key is a minimum, just follow *left* child pointers from the root until you encounter a NIL, as shown in Figure 12.2(b). The TREE-MINIMUM procedure returns a pointer to the minimum element in the subtree rooted at a given node x, which we assume to be non-NIL.

```
TREE-MINIMUM(x)
1 while x.left ≠ NIL
2     x = x.left
3 return x

TREE-MAXIMUM(x)
1 while x.right ≠ NIL
2     x = x.right
3 return x
```

The binary-search-tree property guarantees that TREE-MINIMUM is correct. If node x has no left subtree, then since every key in the right subtree of x is at least as large as $x.key$, the minimum key in the subtree rooted at x is $x.key$. If node x has a left subtree, then since no key in the right subtree is smaller than $x.key$ and every key in the left subtree is not larger than $x.key$, the minimum key in the subtree rooted at x resides in the subtree rooted at $x.left$.

The pseudocode for TREE-MAXIMUM is symmetric. Both TREE-MINIMUM and TREE-MAXIMUM run in $O(h)$ time on a tree of height h since, as in TREE-SEARCH, the sequence of nodes encountered forms a simple path downward from the root.

Successor and predecessor

Given a node in a binary search tree, how can you find its successor in the sorted order determined by an inorder tree walk? If all keys are distinct, the successor of a node x is the node with the smallest key greater than $x.key$. Regardless of whether the keys are distinct, we define the ***successor*** of a node as the next node visited in an inorder tree walk. The structure of a binary search tree allows you to determine the

successor of a node without comparing keys. The TREE-SUCCESSOR procedure on the facing page returns the successor of a node x in a binary search tree if it exists, or NIL if x is the last node that would be visited during an inorder walk.

The code for TREE-SUCCESSOR has two cases. If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in x's right subtree, which line 2 finds by calling TREE-MINIMUM($x.right$). For example, the successor of the node with key 15 in Figure 12.2(c) is the node with key 17.

On the other hand, as Exercise 12.2-6 asks you to show, if the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. In Figure 12.2(d), the successor of the node with key 13 is the node with key 15. To find y, go up the tree from x until you encounter either the root or a node that is the left child of its parent. Lines 4–8 of TREE-SUCCESSOR handle this case.

```
TREE-SUCCESSOR(x)
1 if x.right ≠ NIL
2     return TREE-MINIMUM(x.right)   // leftmost node in right
          subtree
3 else // find the lowest ancestor of x whose left child is an ancestor of
  x
4     y = x.p
5     while y ≠ NIL and x == y.right
6         x = y
7         y = y.p
8     return y
```

The running time of TREE-SUCCESSOR on a tree of height h is $O(h)$, since it either follows a simple path up the tree or follows a simple path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in $O(h)$ time.

In summary, we have proved the following theorem.

Theorem 12.2
The dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR can be implemented so that each one runs in $O(h)$ time on a binary search tree of height h.

■

Exercises

12.2-1
You are searching for the number 363 in a binary search tree containing numbers between 1 and 1000. Which of the following sequences *cannot* be the sequence of nodes examined?

a. 2, 252, 401, 398, 330, 344, 397, 363.

b. 924, 220, 911, 244, 898, 258, 362, 363.

c. 925, 202, 911, 240, 912, 245, 363.

d. 2, 399, 387, 219, 266, 382, 381, 278, 363.

e. 935, 278, 347, 621, 299, 392, 358, 363.

12.2-2
Write recursive versions of TREE-MINIMUM and TREE-MAXIMUM.

12.2-3
Write the TREE-PREDECESSOR procedure.

12.2-4
Professor Kilmer claims to have discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up at a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Kilmer claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \le b \le c$. Give a smallest possible counterexample to the professor's claim.

12.2-5

Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

12.2-6
Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. (Recall that every node is its own ancestor.)

12.2-7
An alternative method of performing an inorder tree walk of an n-node binary search tree finds the minimum element in the tree by calling TREE-MINIMUM and then making $n - 1$ calls to TREE-SUCCESSOR. Prove that this algorithm runs in $\Theta(n)$ time.

12.2-8
Prove that no matter what node you start at in a height-h binary search tree, k successive calls to TREE-SUCCESSOR take $O(k + h)$ time.

12.2-9
Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y be its parent. Show that $y.key$ is either the smallest key in T larger than $x.key$ or the largest key in T smaller than $x.key$.

12.3 Insertion and deletion

The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change. The data structure must be modified to reflect this change, but in such a way that the binary-search-tree property continues to hold. We'll see that modifying the tree to insert a new element is relatively straightforward, but deleting a node from a binary search tree is more complicated.

Insertion

The TREE-INSERT procedure inserts a new node into a binary search tree. The procedure takes a binary search tree T and a node z for which

$z.key$ has already been filled in, $z.left$ = NIL, and $z.right$ = NIL. It modifies T and some of the attributes of z so as to insert z into an appropriate position in the tree.

```
TREE-INSERT(T, z)
 1 x = T.root              // node being compared with z
 2 y = NIL                 // y will be parent of z
 3 while x ≠ NIL           // descend until reaching a leaf
 4     y = x
 5     if z.key < x.key
 6         x = x.left
 7     else x = x.right
 8 z.p = y                 // found the location—insert z with parent y
 9 if y == NIL
10     T.root = z          // tree T was empty
11 elseif z.key < y.key
12     y.left = z
13 else y.right = z
```

Figure 12.3 shows how TREE-INSERT works. Just like the procedures TREE-SEARCH and ITERATIVE-TREE-SEARCH, TREE-INSERT begins at the root of the tree and the pointer x traces a simple path downward looking for a NIL to replace with the input node z. The procedure maintains the ***trailing pointer*** y as the parent of x. After initialization, the **while** loop in lines 3–7 causes these two pointers to move down the tree, going left or right depending on the comparison of $z.key$ with $x.key$, until x becomes NIL. This NIL occupies the position where node z will go. More precisely, this NIL is a *left* or *right* attribute of the node that will become z's parent, or it is $T.root$ if tree T is currently empty. The procedure needs the trailing pointer y, because by the time it finds the NIL where z belongs, the search has proceeded one step beyond the node that needs to be changed. Lines 8–13 set the pointers that cause z to be inserted.

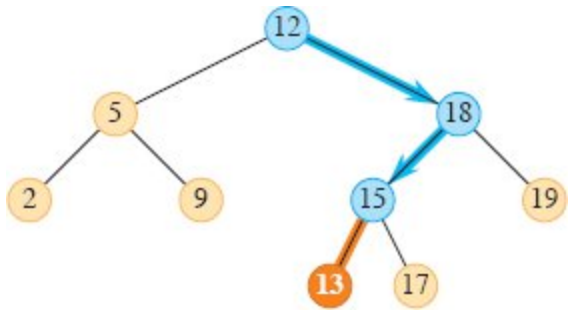

Figure 12.3 Inserting a node with key 13 into a binary search tree. The simple path from the root down to the position where the node is inserted is shown in blue. The new node and the link to its parent are highlighted in orange.

Like the other primitive operations on search trees, the procedure TREE-INSERT runs in $O(h)$ time on a tree of height h.

Deletion

The overall strategy for deleting a node z from a binary search tree T has three basic cases and, as we'll see, one of the cases is a bit tricky.

- If z has no children, then simply remove it by modifying its parent to replace z with NIL as its child.
- If z has just one child, then elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.
- If z has two children, find z's successor y—which must belong to z's right subtree—and move y to take z's position in the tree. The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree. Because y is z's successor, it cannot have a left child, and y's original right child moves into y's original position, with the rest of y's original right subtree following automatically. This case is the tricky one because, as we'll see, it matters whether y is z's right child.

The procedure for deleting a given node z from a binary search tree T takes as arguments pointers to T and z. It organizes its cases a bit differently from the three cases outlined previously by considering the four cases shown in Figure 12.4.

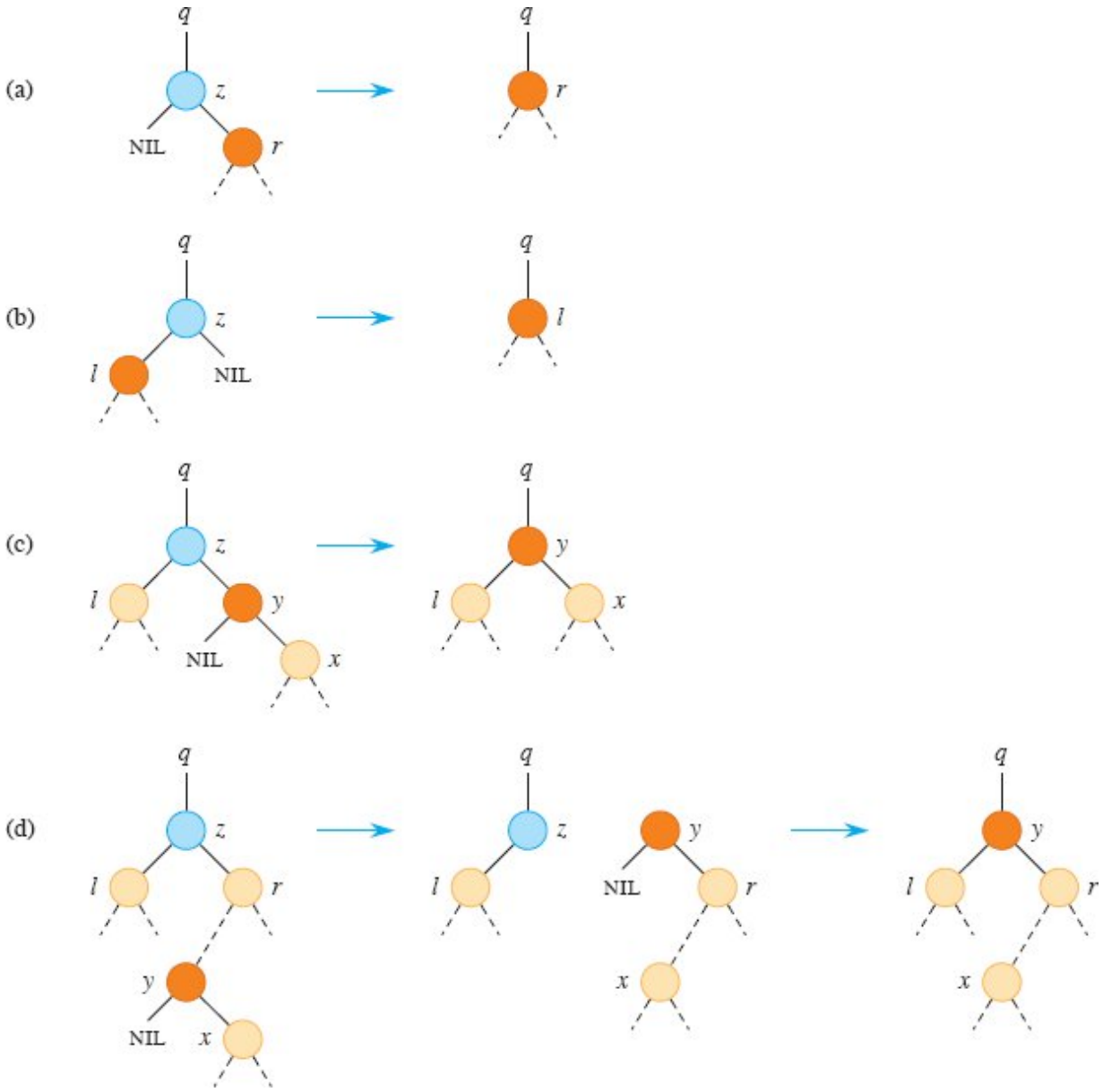

Figure 12.4 Deleting a node z, in blue, from a binary search tree. Node z may be the root, a left child of node q, or a right child of q. The node that will replace node z in its position in the tree is colored orange. **(a)** Node z has no left child. Replace z by its right child r, which may or may not be NIL. **(b)** Node z has a left child l but no right child. Replace z by l. **(c)** Node z has two children. Its left child is node l, its right child is its successor y (which has no left child), and y's right child is node x. Replace z by y, updating y's left child to become l, but leaving x as y's right child. **(d)** Node z has two children (left child l and right child r), and its successor $y \neq r$ lies within the subtree rooted at r. First replace y by its own right child x, and set y to be r's parent. Then set y to be q's child and the parent of l.

- If z has no left child, then as in part (a) of the figure, replace z by its right child, which may or may not be NIL. When z's right child is NIL, this case deals with the situation in which z has no children. When z's right child is non-NIL, this case handles the situation in which z has just one child, which is its right child.
- Otherwise, if z has just one child, then that child is a left child. As in part (b) of the figure, replace z by its left child.
- Otherwise, z has both a left and a right child. Find z's successor y, which lies in z's right subtree and has no left child (see Exercise 12.2-5). Splice node y out of its current location and replace z by y in the tree. How to do so depends on whether y is z's right child:
    - If y is z's right child, then as in part (c) of the figure, replace z by y, leaving y's right child alone.
    - Otherwise, y lies within z's right subtree but is not z's right child. In this case, as in part (d) of the figure, first replace y by its own right child, and then replace z by y.

As part of the process of deleting a node, subtrees need to move around within the binary search tree. The subroutine TRANSPLANT replaces one subtree as a child of its parent with another subtree. When TRANSPLANT replaces the subtree rooted at node u with the subtree rooted at node v, node u's parent becomes node v's parent, and u's parent ends up having v as its appropriate child. TRANSPLANT allows v to be NIL instead of a pointer to a node.

```
TRANSPLANT(T, u, v)
1 if u.p == NIL
2     T.root = v
3 elseif u == u.p.left
4     u.p.left = v
5 else u.p.right = v
6 if v ≠ NIL
7     v.p = u.p
```

Here is how TRANSPLANT works. Lines 1–2 handle the case in which u is the root of T. Otherwise, u is either a left child or a right child of its parent. Lines 3–4 take care of updating $u.p.left$ if u is a left child, and line 5 updates $u.p.right$ if u is a right child. Because v may be NIL, lines 6–7 update $v.p$ only if v is non-NIL. The procedure TRANSPLANT does not attempt to update $v.left$ and $v.right$. Doing so, or not doing so, is the responsibility of TRANSPLANT's caller.

The procedure TREE-DELETE on the facing page uses TRANSPLANT to delete node z from binary search tree T. It executes the four cases as follows. Lines 1–2 handle the case in which node z has no left child (Figure 12.4(a)), and lines 3–4 handle the case in which z has a left child but no right child (Figure 12.4(b)). Lines 5–12 deal with the remaining two cases, in which z has two children. Line 5 finds node y, which is the successor of z. Because z has a nonempty right subtree, its successor must be the node in that subtree with the smallest key; hence the call to TREE-MINIMUM($z.right$). As we noted before, y has no left child. The procedure needs to splice y out of its current location and replace z by y in the tree. If y is z's right child (Figure 12.4(c)), then lines 10–12 replace z as a child of its parent by y and replace y's left child by z's left child. Node y retains its right child (x in Figure 12.4(c)), and so no change to $y.right$ needs to occur. If y is not z's right child (Figure 12.4(d)), then two nodes have to move. Lines 7–9 replace y as a child of its parent by y's right child (x in Figure 12.4(c)) and make z's right child (r in the figure) become y's right child instead. Finally, lines 10–12 replace z as a child of its parent by y and replace y's left child by z's left child.

```
TREE-DELETE(T, z)
 1 if z.left == NIL
 2     TRANSPLANT(T, z, z.right)          // replace z by its right child
 3 elseif z.right == NIL
 4     TRANSPLANT(T, z, z.left)           // replace z by its left child
 5 else y = TREE-MINIMUM(z.right)         // y is z's successor
 6     if y ≠ z.right                     // is y farther down the tree?
 7         TRANSPLANT(T, y, y.right)      // replace y by its right child
```

```
 8        y.right = z.right                  // z's right child becomes
 9        y.right.p = y                      //      y's right child
10    TRANSPLANT(T, z, y)                    // replace z by its successor y
11    y.left = z.left                        // and give z's left child to y,
12    y.left.p = y                           //      which had no left child
```

Each line of TREE-DELETE, including the calls to TRANSPLANT, takes constant time, except for the call to TREE-MINIMUM in line 5. Thus, TREE-DELETE runs in $O(h)$ time on a tree of height h.

In summary, we have proved the following theorem.

Theorem 12.3

The dynamic-set operations INSERT and DELETE can be implemented so that each one runs in $O(h)$ time on a binary search tree of height h.

■

Exercises

12.3-1

Give a recursive version of the TREE-INSERT procedure.

12.3-2

Suppose that you construct a binary search tree by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is 1 plus the number of nodes examined when the value was first inserted into the tree.

12.3-3

You can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

12.3-4

When TREE-DELETE calls TRANSPLANT, under what circumstances can the parameter v of TRANSPLANT be NIL?

12.3-5

Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give a counterexample.

12.3-6

Suppose that instead of each node x keeping the attribute $x.p$, pointing to x's parent, it keeps $x.succ$, pointing to x's successor. Give pseudocode for TREE-SEARCH, TREE-INSERT, and TREE-DELETE on a binary search tree T using this representation. These procedures should operate in $O(h)$ time, where h is the height of the tree T. You may assume that all keys in the binary search tree are distinct. (*Hint:* You might wish to implement a subroutine that returns the parent of a node.)

12.3-7

When node z in TREE-DELETE has two children, you can choose node y to be its predecessor rather than its successor. What other changes to TREE-DELETE are necessary if you do so? Some have argued that a fair strategy, giving equal priority to predecessor and successor, yields better empirical performance. How might TREE-DELETE be minimally changed to implement such a fair strategy?

Problems

12-1 Binary search trees with equal keys

Equal keys pose a problem for the implementation of binary search trees.

a. What is the asymptotic performance of TREE-INSERT when used to insert n items with identical keys into an initially empty binary search tree?

Consider changing TREE-INSERT to test whether $z.key = x.key$ before line 5 and to test whether $z.key = y.key$ before line 11. If equality holds, implement one of the following strategies. For each strategy, find the asymptotic performance of inserting n items with identical keys into an initially empty binary search tree. (The strategies are described for line 5, which compares the keys of z and x. Substitute y for x to arrive at the strategies for line 11.)

b. Keep a boolean flag $x.b$ at node x, and set x to either $x.left$ or $x.right$ based on the value of $x.b$, which alternates between FALSE and TRUE each time TREE-INSERT visits x while inserting a node with the same key as x.

c. Keep a list of nodes with equal keys at x, and insert z into the list.

d. Randomly set x to either $x.left$ or $x.right$. (Give the worst-case performance and informally derive the expected running time.)

12-2 Radix trees

Given two strings $a = a_0a_1 \ldots a_p$ and $b = b_0b_1 \ldots b_q$, where each a_i and each b_j belongs to some ordered set of characters, we say that string a is ***lexicographically less than*** string b if either

1. there exists an integer j, where $0 \le j \le \min\{p, q\}$, such that $a_i = b_i$ for all $i = 0, 1, \ldots, j - 1$ and $a_j < b_j$, or
2. $p < q$ and $a_i = b_i$ for all $i = 0, 1, \ldots, p$.

For example, if a and b are bit strings, then $10100 < 10110$ by rule 1 (letting $j = 3$) and $10100 < 101000$ by rule 2. This ordering is similar to that used in English-language dictionaries.

The ***radix tree*** data structure shown in Figure 12.5 (also known as a ***trie***) stores the bit strings 1011, 10, 011, 100, and 0. When searching for a key $a = a_0a_1 \ldots a_p$, go left at a node of depth i if $a_i = 0$ and right if $a_i = 1$. Let S be a set of distinct bit strings whose lengths sum to n. Show how to use a radix tree to sort S lexicographically in $\Theta(n)$ time. For the

example in Figure 12.5, the output of the sort should be the sequence 0, 011, 10, 100, 1011.

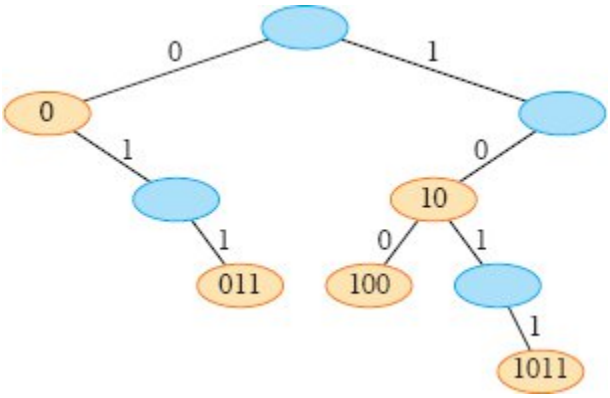

Figure 12.5 A radix tree storing the bit strings 1011, 10, 011, 100, and 0. To determine each node's key, traverse the simple path from the root to that node. There is no need, therefore, to store the keys in the nodes. The keys appear here for illustrative purposes only. Keys corresponding to blue nodes are not in the tree. Such nodes are present only to establish a path to other nodes.

12-3 Average node depth in a randomly built binary search tree

A ***randomly built binary search tree*** on n keys is a binary search tree created by starting with an empty tree and inserting the keys in random order, where each of the $n!$ permutations of the keys is equally likely. In this problem, you will prove that the average depth of a node in a randomly built binary search tree with n nodes is $O(\lg n)$. The technique reveals a surprising similarity between the building of a binary search tree and the execution of RANDOMIZED-QUICKSORT from Section 7.3.

Denote the depth of any node x in tree T by $d(x, T)$. Then the ***total path length*** $P(T)$ of a tree T is the sum, over all nodes x in T, of $d(x, T)$.

a. Argue that the average depth of a node in T is

$$\frac{1}{n}\sum_{x \in T} d(x, T) = \frac{1}{n} P(T) .$$

Thus, you need to show that the expected value of $P(T)$ is $O(n \lg n)$.

b. Let T_L and T_R denote the left and right subtrees of tree T, respectively. Argue that if T has n nodes, then

$P(T) = P(T_L) + P(T_R) + n - 1$.

c. Let $P(n)$ denote the average total path length of a randomly built binary search tree with n nodes. Show that

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1) .$$

d. Show how to rewrite $P(n)$ as

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n) .$$

e. Recalling the alternative analysis of the randomized version of quicksort given in Problem 7-3, conclude that $P(n) = O(n \lg n)$.

Each recursive invocation of randomized quicksort chooses a random pivot element to partition the set of elements being sorted. Each node of a binary search tree partitions the set of elements that fall into the subtree rooted at that node.

f. Describe an implementation of quicksort in which the comparisons to sort a set of elements are exactly the same as the comparisons to insert the elements into a binary search tree. (The order in which comparisons are made may differ, but the same comparisons must occur.)

12-4 Number of different binary trees

Let b_n denote the number of different binary trees with n nodes. In this problem, you will find a formula for b_n, as well as an asymptotic estimate.

a. Show that $b_0 = 1$ and that, for $n \geq 1$,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} .$$

b. Referring to Problem 4-5 on page 121 for the definition of a generating function, let $B(x)$ be the generating function

$$B(x) = \sum_{n=0}^{\infty} b_n x^n .$$

Show that $B(x) = xB(x)^2 + 1$, and hence one way to express $B(x)$ in closed form is

$$B(x) = \frac{1}{2x}\left(1 - \sqrt{1-4x}\right) .$$

The ***Taylor expansion*** of $f(x)$ around the point $x = a$ is given by

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k ,$$

where $f^{(k)}(x)$ is the kth derivative of f evaluated at x.

c. Show that

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

(the nth ***Catalan number***) by using the Taylor expansion of $\sqrt{1-4x}$ around $x = 0$. (If you wish, instead of using the Taylor expansion, you may use the generalization of the binomial theorem, equation (C.4) on page 1181, to noninteger exponents n, where for any real number n and for any integer k, you can interpret $\binom{n}{k}$ to be $n(n-1)\ldots(n-k+1)/k!$ if $k \geq 0$, and 0 otherwise.)

d. Show that

$$b_n = \frac{4^n}{\sqrt{\pi} n^{3/2}}\left(1 + O(1/n)\right) .$$

Chapter notes

Knuth [261] contains a good discussion of simple binary search trees as well as many variations. Binary search trees seem to have been independently discovered by a number of people in the late 1950s. Radix trees are often called "tries," which comes from the middle letters in the word *retrieval*. Knuth [261] also discusses them.

Many texts, including the first two editions of this book, describe a somewhat simpler method of deleting a node from a binary search tree when both of its children are present. Instead of replacing node z by its successor y, delete node y but copy its key and satellite data into node z. The downside of this approach is that the node actually deleted might not be the node passed to the delete procedure. If other components of a program maintain pointers to nodes in the tree, they could mistakenly end up with "stale" pointers to nodes that have been deleted. Although the deletion method presented in this edition of this book is a bit more complicated, it guarantees that a call to delete node z deletes node z and only node z.

Section 14.5 will show how to construct an optimal binary search tree when you know the search frequencies before constructing the tree. That is, given the frequencies of searching for each key and the frequencies of searching for values that fall between keys in the tree, a set of searches in the constructed binary search tree examines the minimum number of nodes.

13 Red-Black Trees

Chapter 12 showed that a binary search tree of height h can support any of the basic dynamic-set operations—such as SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, and DELETE—in $O(h)$ time. Thus, the set operations are fast if the height of the search tree is small. If its height is large, however, the set operations may run no faster than with a linked list. Red-black trees are one of many search-tree schemes that are "balanced" in order to guarantee that basic dynamic-set operations take $O(\lg n)$ time in the worst case.

13.1 Properties of red-black trees

A ***red-black tree*** is a binary search tree with one extra bit of storage per node: its ***color***, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately ***balanced***. Indeed, as we're about to see, the height of a red-black tree with n keys is at most $2 \lg(n + 1)$, which is $O(\lg n)$.

Each node of the tree now contains the attributes *color*, *key*, *left*, *right*, and p. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. Think of these NILs as pointers to leaves (external nodes) of the binary

search tree and the normal, key-bearing nodes as internal nodes of the tree.

A red-black tree is a binary search tree that satisfies the following ***red-black properties***:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Figure 13.1(a) shows an example of a red-black tree.

As a matter of convenience in dealing with boundary conditions in red-black tree code, we use a single sentinel to represent NIL (see page 262). For a red-black tree T, the sentinel $T.nil$ is an object with the same attributes as an ordinary node in the tree. Its *color* attribute is BLACK, and its other attributes—p, *left*, *right*, and *key*—can take on arbitrary values. As Figure 13.1(b) shows, all pointers to NIL are replaced by pointers to the sentinel $T.nil$.

Why use the sentinel? The sentinel makes it possible to treat a NIL child of a node x as an ordinary node whose parent is x. An alternative design would use a distinct sentinel node for each NIL in the tree, so that the parent of each NIL is well defined. That approach needlessly wastes space, however. Instead, just the one sentinel $T.nil$ represents all the NILs—all leaves and the root's parent. The values of the attributes p, *left*, *right*, and *key* of the sentinel are immaterial. The red-black tree procedures can place whatever values in the sentinel that yield simpler code.

We generally confine our interest to the internal nodes of a red-black tree, since they hold the key values. The remainder of this chapter omits the leaves in drawings of red-black trees, as shown in Figure 13.1(c).

We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the ***black-height*** of the node, denoted $\mathrm{bh}(x)$. By property 5, the notion of black-height is well defined, since all

descending simple paths from the node have the same number of black nodes. The black-height of a red-black tree is the black-height of its root.

The following lemma shows why red-black trees make good search trees.

Lemma 13.1
A red-black tree with n internal nodes has height at most $2 \lg(n + 1)$.

Proof We start by showing that the subtree rooted at any node x contains at least $2^{\mathrm{bh}(x)} - 1$ internal nodes. We prove this claim by induction on the height of x. If the height of x is 0, then x must be a leaf (*T.nil*), and the subtree rooted at x indeed contains at least $2^{\mathrm{bh}(x)} - 1 = 2^0 - 1 = 0$ internal nodes. For the inductive step, consider a node x that has positive height and is an internal node. Then node x has two children, either or both of which may be a leaf. If a child is black, then it contributes 1 to x's black-height but not to its own. If a child is red, then it contributes to neither x's black-height nor its own. Therefore, each child has a black-height of either $\mathrm{bh}(x) - 1$ (if it's black) or $\mathrm{bh}(x)$ (if it's red). Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{\mathrm{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{\mathrm{bh}(x)-1} - 1) + (2^{\mathrm{bh}(x)-1} - 1) + 1 = 2^{\mathrm{bh}(x)} - 1$ internal nodes, which proves the claim.

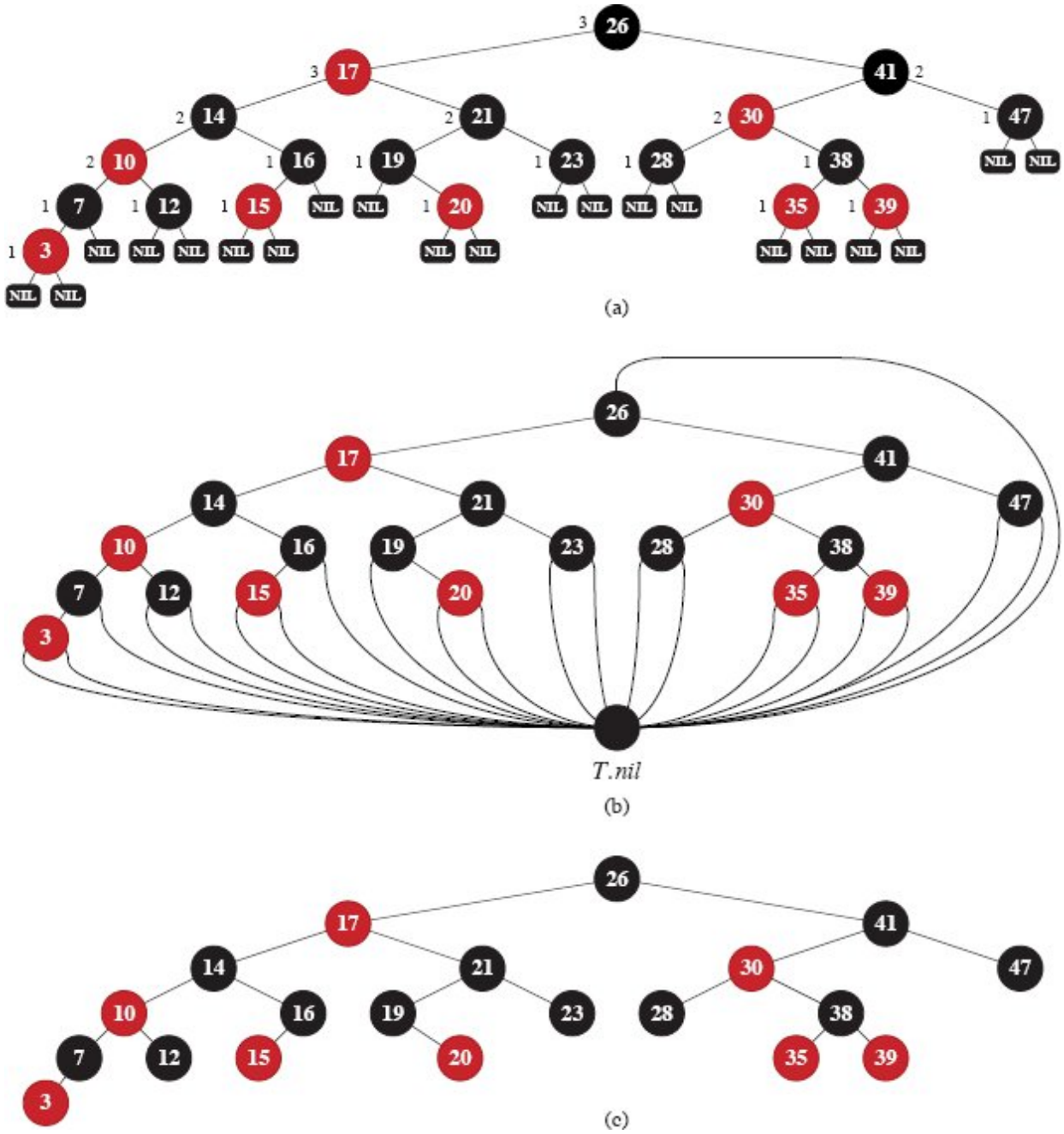

Figure 13.1 A red-black tree. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height, where NILs have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *T.nil*, which is always black, and with black-heights omitted. The root's parent is also the sentinel. **(c)** The same red-black tree but with leaves and the root's parent omitted entirely. The remainder of this chapter uses this drawing style.

To complete the proof of the lemma, let h be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least $h/2$, and thus,

$$n \geq 2^{h/2} - 1.$$

Moving the 1 to the left-hand side and taking logarithms on both sides yields $\lg(n + 1) \geq h/2$, or $h \leq 2 \lg(n + 1)$.

■

As an immediate consequence of this lemma, each of the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR runs in $O(\lg n)$ time on a red-black tree, since each can run in $O(h)$ time on a binary search tree of height h (as shown in Chapter 12) and any red-black tree on n nodes is a binary search tree with height $O(\lg n)$. (Of course, references to NIL in the algorithms of Chapter 12 have to be replaced by *T.nil*.) Although the procedures TREE-INSERT and TREE-DELETE from Chapter 12 run in $O(\lg n)$ time when given a red-black tree as input, you cannot just use them to implement the dynamic-set operations INSERT and DELETE. They do not necessarily maintain the red-black properties, so you might not end up with a legal red-black tree. The remainder of this chapter shows how to insert into and delete from a red-black tree in $O(\lg n)$ time.

Exercises

13.1-1

In the style of Figure 13.1(a), draw the complete binary search tree of height 3 on the keys $\{1, 2, \ldots, 15\}$. Add the NIL leaves and color the nodes in three different ways such that the black-heights of the resulting red-black trees are 2, 3, and 4.

13.1-2

Draw the red-black tree that results after TREE-INSERT is called on the tree in Figure 13.1 with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?

13.1-3

Define a ***relaxed red-black tree*** as a binary search tree that satisfies red-black properties 1, 3, 4, and 5, but whose root may be either red or black. Consider a relaxed red-black tree T whose root is red. If the root of T is changed to black but no other changes occur, is the resulting tree a red-black tree?

13.1-4

Suppose that every black node in a red-black tree "absorbs" all of its red children, so that the children of any red node become children of the black parent. (Ignore what happens to the keys.) What are the possible degrees of a black node after all its red children are absorbed? What can you say about the depths of the leaves of the resulting tree?

13.1-5

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

13.1-6

What is the largest possible number of internal nodes in a red-black tree with black-height k? What is the smallest possible number?

13.1-7

Describe a red-black tree on n keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

13.1-8

Argue that in a red-black tree, a red node cannot have exactly one non-NIL child.

13.2 Rotations

The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with n keys, take $O(\lg n)$ time. Because they modify the tree, the result may violate the red-black properties

enumerated in Section 13.1. To restore these properties, colors and pointers within nodes need to change.

The pointer structure changes through ***rotation***, which is a local operation in a search tree that preserves the binary-search-tree property. Figure 13.2 shows the two kinds of rotations: left rotations and right rotations. Let's look at a left rotation on a node x, which transforms the structure on the right side of the figure to the structure on the left. Node x has a right child y, which must not be $T.nil$. The left rotation changes the subtree originally rooted at x by "twisting" the link between x and y to the left. The new root of the subtree is node y, with x as y's left child and y's original left child (the subtree represented by β in the figure) as x's right child.

The pseudocode for LEFT-ROTATE appearing on the following page assumes that $x.right \neq T.nil$ and that the root's parent is $T.nil$. Figure 13.3 shows an example of how LEFT-ROTATE modifies a binary search tree. The code for RIGHT-ROTATE is symmetric. Both LEFT-ROTATE and RIGHT-ROTATE run in $O(1)$ time. Only pointers are changed by a rotation, and all other attributes in a node remain the same.

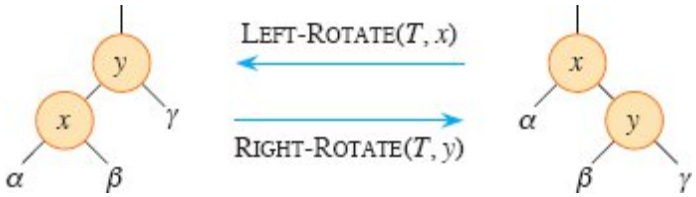

Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation RIGHT-ROTATE(T, y) transforms the configuration on the left into the configuration on the right. The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede $x.key$, which precedes the keys in β, which precede $y.key$, which precedes the keys in γ.

```
LEFT-ROTATE(T, x)
1 y = x.right
2 x.right = y.left      // turn y's left subtree into x's right subtree
```

```
3 if y.left ≠ T.nil      // if y's left subtree is not empty …
4     y.left.p = x       // … then x becomes the parent of the subtree's
                         root
5 y.p = x.p              // x's parent becomes y's parent
6 if x.p == T.nil        // if x was the root …
7     T.root = y         // … then y becomes the root
8 elseif      x      ==  // otherwise, if x was a left child …
  x.p.left
9     x.p.left = y       // … then y becomes a left child
10 else x.p.right = y    // otherwise, x was a right child, and now y is
11 y.left = x            // make x become y's left child
12 x.p = y
```

Exercises

13.2-1

Write pseudocode for RIGHT-ROTATE.

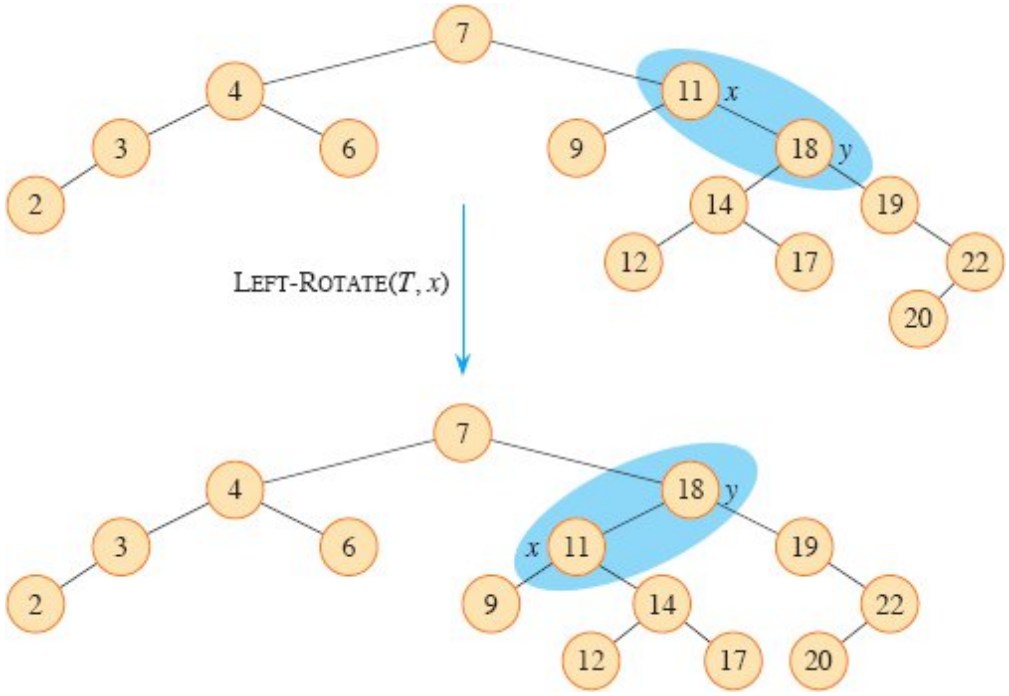

Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

13.2-2

Argue that in every n-node binary search tree, there are exactly $n - 1$ possible rotations.

13.2-3

Let a, b, and c be arbitrary nodes in subtrees α, β, and γ, respectively, in the right tree of Figure 13.2. How do the depths of a, b, and c change when a left rotation is performed on node x in the figure?

13.2-4

Show that any arbitrary n-node binary search tree can be transformed into any other arbitrary n-node binary search tree using $O(n)$ rotations. (*Hint:* First show that at most $n - 1$ right rotations suffice to transform the tree into a right-going chain.)

★ *13.2-5*

We say that a binary search tree T_1 can be ***right-converted*** to binary search tree T_2 if it is possible to obtain T_2 from T_1 via a series of calls to RIGHT-ROTATE. Give an example of two trees T_1 and T_2 such that T_1 cannot be right-converted to T_2. Then, show that if a tree T_1 can be right-converted to T_2, it can be right-converted using $O(n^2)$ calls to RIGHT-ROTATE.

13.3 Insertion

In order to insert a node into a red-black tree with n internal nodes in $O(\lg n)$ time and maintain the red-black properties, we'll need to slightly modify the TREE-INSERT procedure on page 321. The procedure RB-INSERT starts by inserting node z into the tree T as if it were an ordinary binary search tree, and then it colors z red. (Exercise 13.3-1 asks you to explain why to make node z red rather than black.) To guarantee that the red-black properties are preserved, an auxiliary procedure RB-INSERT-FIXUP on the facing page recolors nodes and performs rotations. The call RB-INSERT(T, z) inserts node z, whose *key* is assumed to have already been filled in, into the red-black tree T.

```
RB-INSERT(T, z)
 1 x = T.root                  // node being compared with z
 2 y = T.nil                   // y will be parent of z
 3 while x ≠ T.nil             // descend until reaching the sentinel
 4     y = x
 5     if z.key < x.key
 6         x = x.left
 7     else x = x.right
 8 z.p = y                     // found the location—insert z with parent
                               y
 9 if y == T.nil
10     T.root = z             // tree T was empty
11 elseif z.key < y.key
12     y.left = z
```

```
13 else y.right = z
14 z.left = T.nil                  // both of z's children are the sentinel
15 z.right = T.nil
16 z.color = RED                  // the new node starts out red
17 RB-INSERT-FIXUP(T, z)          // correct any violations of red-black properties
```

The procedures TREE-INSERT and RB-INSERT differ in four ways. First, all instances of NIL in TREE-INSERT are replaced by *T.nil*. Second, lines 14–15 of RB-INSERT set *z.left* and *z.right* to *T.nil*, in order to maintain the proper tree structure. (TREE-INSERT assumed that *z*'s children were already NIL.) Third, line 16 colors *z* red. Fourth, because coloring *z* red may cause a violation of one of the red-black properties, line 17 of RB-INSERT calls RB-INSERT-FIXUP(*T*, *z*) in order to restore the red-black properties.

```
RB-INSERT-FIXUP(T, z)
 1 while z.p.color == RED
 2     if z.p == z.p.p.left                  // is z's parent a left child?
 3         y = z.p.p.right                   // y is z's uncle
 4         if y.color == RED                 // are z's parent and uncle both red?
 5             z.p.color = BLACK             \
 6             y.color = BLACK               | case 1
 7             z.p.p.color = RED             |
 8             z = z.p.p                     /
 9         else
10             if z == z.p.right
11                 z = z.p                   \ case 2
12                 LEFT-ROTATE(T, z)         /
13             z.p.color = BLACK             \
14             z.p.p.color = RED             | case 3
15             RIGHT-ROTATE(T,               |
                   z.p.p)                    /
16     else // same as lines 3–15, but with "right" and "left" exchanged
```

```
17          y = z.p.p.left
18          if y.color == RED
19              z.p.color = BLACK
20              y.color = BLACK
21              z.p.p.color = RED
22              z = z.p.p
23          else
24              if z == z.p.left
25                  z = z.p
26                  RIGHT-ROTATE(T,
                        z)
27              z.p.color = BLACK
28              z.p.p.color = RED
29              LEFT-ROTATE(T, z.p.p)
30  T.root.color = BLACK
```

To understand how RB-INSERT-FIXUP works, let's examine the code in three major steps. First, we'll determine which violations of the red-black properties might arise in RB-INSERT upon inserting node z and coloring it red. Second, we'll consider the overall goal of the **while** loop in lines 1–29. Finally, we'll explore each of the three cases within the **while** loop's body (case 2 falls through into case 3, so these two cases are not mutually exclusive) and see how they accomplish the goal.

In describing the structure of a red-black tree, we'll often need to refer to the sibling of a node's parent. We use the term ***uncle*** for such a node.[1] Figure 13.4 shows how RB-INSERT-FIXUP operates on a sample red-black tree, with cases depending in part on the colors of a node, its parent, and its uncle.

What violations of the red-black properties might occur upon the call to RB-INSERT-FIXUP? Property 1 certainly continues to hold (every node is either red or black), as does property 3 (every leaf is black), since both children of the newly inserted red node are the sentinel $T.nil$. Property 5, which says that the number of black nodes is the same on every simple path from a given node, is satisfied as well, because node z replaces the (black) sentinel, and node z is red with sentinel children.

Thus, the only properties that might be violated are property 2, which requires the root to be black, and property 4, which says that a red node cannot have a red child. Both possible violations may arise because z is colored red. Property 2 is violated if z is the root, and property 4 is violated if z's parent is red. Figure 13.4(a) shows a violation of property 4 after the node z has been inserted.

The **while** loop of lines 1–29 has two symmetric possibilities: lines 3–15 deal with the situation in which node z's parent $z.p$ is a left child of z's grandparent $z.p.p$, and lines 17–29 apply when z's parent is a right child. Our proof will focus only on lines 3–15, relying on the symmetry in lines 17–29.

We'll show that the **while** loop maintains the following three-part invariant at the start of each iteration of the loop:

a. Node z is red.

b. If $z.p$ is the root, then $z.p$ is black.

c. If the tree violates any of the red-black properties, then it violates at most one of them, and the violation is of either property 2 or property 4, but not both. If the tree violates property 2, it is because z is the root and is red. If the tree violates property 4, it is because both z and $z.p$ are red.

Part (c), which deals with violations of red-black properties, is more central to showing that RB-INSERT-FIXUP restores the red-black properties than parts (a) and (b), which we'll use along the way to understand situations in the code. Because we'll be focusing on node z and nodes near it in the tree, it helps to know from part (a) that z is red. Part (b) will help show that z's grandparent $z.p.p$ exists when it's referenced in lines 2, 3, 7, 8, 14, and 15 (recall that we're focusing only on lines 3–15).

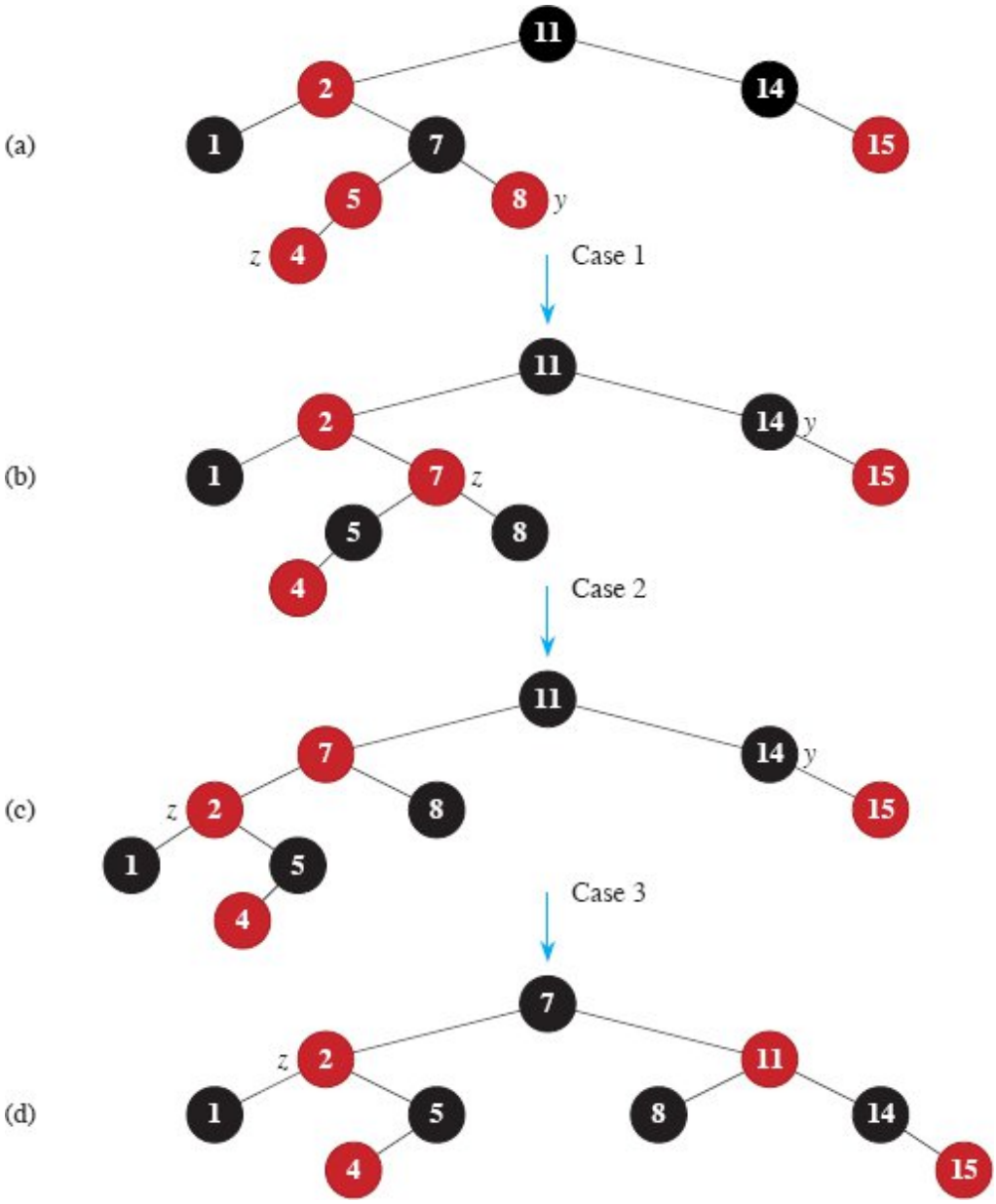

Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Because both z and its parent $z.p$ are red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code applies. Node z's grandparent $z.p.p$ must be black, and its blackness transfers down one level to z's parent and uncle. Once the pointer z moves up two levels in the tree, the tree shown in **(b)** results. Once again, z and its parent are both red, but this time z's uncle y is black. Since z is the right child of $z.p$, case 2 applies. Performing a left rotation results in the tree in **(c)**. Now z is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in **(d)**, which is a legal red-black tree.

Recall that to use a loop invariant, we need to show that the invariant is true upon entering the first iteration of the loop, that each iteration maintains it, that the loop terminates, and that the loop invariant gives us a useful property at loop termination. We'll see that each iteration of the loop has two possible outcomes: either the pointer z moves up the tree, or some rotations occur and then the loop terminates.

Initialization: Before RB-INSERT is called, the red-black tree has no violations. RB-INSERT adds a red node z and calls RB-INSERT-FIXUP. We'll show that each part of the invariant holds at the time RB-INSERT-FIXUP is called:

a. When RB-INSERT-FIXUP is called, z is the red node that was added.

b. If $z.p$ is the root, then $z.p$ started out black and did not change before the call of RB-INSERT-FIXUP.

c. We have already seen that properties 1, 3, and 5 hold when RB-INSERTFIXUP is called.

If the tree violates property 2 (the root must be black), then the red root must be the newly added node z, which is the only internal node in the tree. Because the parent and both children of z are the sentinel, which is black, the tree does not also violate property 4 (both children of a red node are black). Thus this violation of property 2 is the only violation of red-black properties in the entire tree.

If the tree violates property 4, then, because the children of node z are black sentinels and the tree had no other violations prior to z being added, the violation must be because both z and $z.p$ are red. Moreover, the tree violates no other red-black properties.

Maintenance: There are six cases within the **while** loop, but we'll examine only the three cases in lines 3–15, when node z's parent $z.p$ is a left child of z's grandparent $z.p.p$. The proof for lines 17–29 is symmetric. The node $z.p.p$ exists, since by part (b) of the loop invariant, if $z.p$ is the root, then $z.p$ is black. Since RB-INSERT-FIXUP enters a loop

iteration only if $z.p$ is red, we know that $z.p$ cannot be the root. Hence, $z.p.p$ exists.

Case 1 differs from cases 2 and 3 by the color of z's uncle y. Line 3 makes y point to z's uncle $z.p.p.right$, and line 4 tests y's color. If y is red, then case 1 executes. Otherwise, control passes to cases 2 and 3. In all three cases, z's grandparent $z.p.p$ is black, since its parent $z.p$ is red, and property 4 is violated only between z and $z.p$.

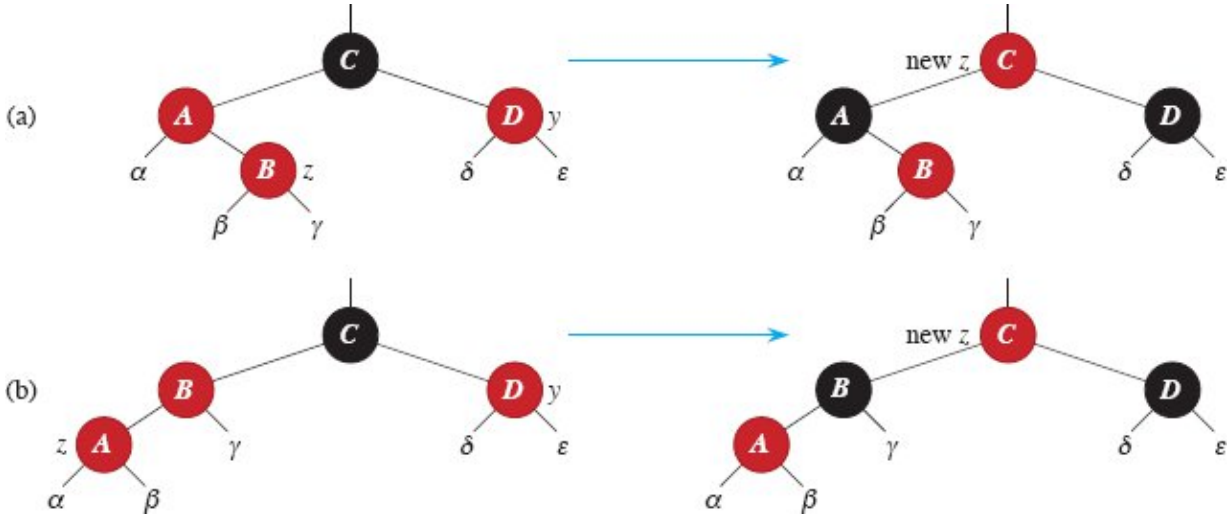

Figure 13.5 Case 1 of the procedure RB-INSERT-FIXUP. Both z and its parent $z.p$ are red, violating property 4. In case 1, z's uncle y is red. The same action occurs regardless of whether **(a)** z is a right child or **(b)** z is a left child. Each of the subtrees α, β, γ, δ, and ϵ has a black root—possibly the sentinel—and each has the same black-height. The code for case 1 moves the blackness of z's grandparent down to z's parent and uncle, preserving property 5: all downward simple paths from a node to a leaf have the same number of blacks. The **while** loop continues with node z's grandparent $z.p.p$ as the new z. If the action of case 1 causes a new violation of property 4 to occur, it must be only between the new z, which is red, and its parent, if it is red as well.

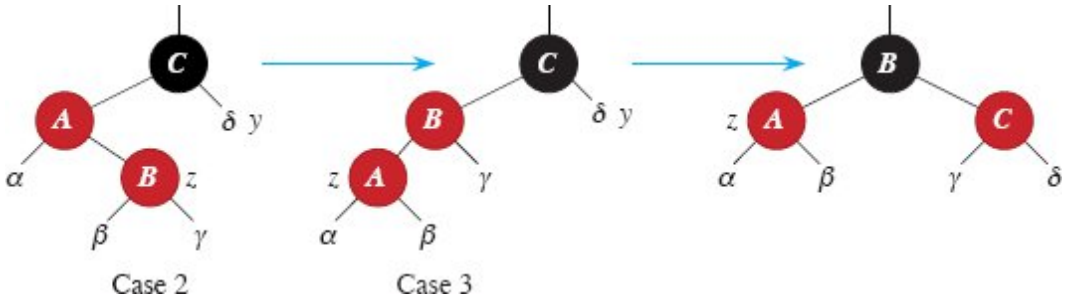

Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT-FIXUP. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent $z.p$ are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from property 4, and δ because otherwise case 1 would apply), and each has the same black-height. Case 2 transforms into case 3 by a left rotation, which preserves property 5: all downward simple paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The **while** loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.

Case 1. z's uncle y is red

Figure 13.5 shows the situation for case 1 (lines 5–8), which occurs when both $z.p$ and y are red. Because z's grandparent $z.p.p$ is black, its blackness can transfer down one level to both $z.p$ and y, thereby fixing the problem of z and $z.p$ both being red. Having had its blackness transferred down one level, z's grandparent becomes red, thereby maintaining property 5. The **while** loop repeats with $z.p.p$ as the new node z, so that the pointer z moves up two levels in the tree.

Now, we show that case 1 maintains the loop invariant at the start of the next iteration. We use z to denote node z in the current iteration, and $z' = z.p.p$ to denote the node that will be called node z at the test in line 1 upon the next iteration.

a. Because this iteration colors $z.p.p$ red, node z' is red at the start of the next iteration.

b. The node $z'.p$ is $z.p.p.p$ in this iteration, and the color of this node does not change. If this node is the root, it was black prior to this iteration, and it remains black at the start of the next iteration.

c. We have already argued that case 1 maintains property 5, and it does not introduce a violation of properties 1 or 3.

   If node z' is the root at the start of the next iteration, then case 1 corrected the lone violation of property 4 in this iteration. Since z' is red and it is the root, property 2 becomes the only one that is violated, and this violation is due to z'.

   If node z' is not the root at the start of the next iteration, then case 1 has not created a violation of property 2. Case 1 corrected the lone violation of property 4 that existed at the start of this iteration. It then made z' red and left $z'.p$ alone. If $z'.p$ was black, there is no violation of property 4. If $z'.p$ was red, coloring z' red created one violation of property 4, between z' and $z'.p$.

Case 2. z's uncle y is black and z is a right child
Case 3. z's uncle y is black and z is a left child

In cases 2 and 3, the color of z's uncle y is black. We distinguish the two cases, which assume that z's parent $z.p$ is red and a left child, according to whether z is a right or left child of $z.p$. Lines 11–12 constitute case 2, which is shown in Figure 13.6 together with case 3. In case 2, node z is a right child of its parent. A left rotation immediately transforms the situation into case 3 (lines 13–15), in which node z is a left child. Because both z and $z.p$ are red, the rotation affects neither the black-heights of nodes nor property 5. Whether case 3 executes directly or through case 2, z's uncle y is black, since otherwise case 1 would have run. Additionally, the node $z.p.p$ exists, since we have argued that this node existed at the time that lines 2 and 3 were executed, and after moving z up one level in line 11 and then down one level in line 12, the identity of $z.p.p$ remains unchanged. Case 3 performs some color changes and a right rotation, which preserve property 5. At this point, there are no longer two red nodes in a row. The **while** loop terminates upon the next test in line 1, since $z.p$ is now black.

We now show that cases 2 and 3 maintain the loop invariant. (As we have just argued, $z.p$ will be black upon the next test in line 1, and the loop body will not execute again.)

a. Case 2 makes z point to $z.p$, which is red. No further change to z or its color occurs in cases 2 and 3.

b. Case 3 makes $z.p$ black, so that if $z.p$ is the root at the start of the next iteration, it is black.

c. As in case 1, properties 1, 3, and 5 are maintained in cases 2 and 3.

Since node z is not the root in cases 2 and 3, we know that there is no violation of property 2. Cases 2 and 3 do not introduce a violation of property 2, since the only node that is made red becomes a child of a black node by the rotation in case 3.

Cases 2 and 3 correct the lone violation of property 4, and they do not introduce another violation.

Termination: To see that the loop terminates, observe that if only case 1 occurs, then the node pointer z moves toward the root in each iteration, so that eventually $z.p$ is black. (If z is the root, then $z.p$ is the sentinel $T.nil$, which is black.) If either case 2 or case 3 occurs, then we've seen that the loop terminates. Since the loop terminates because $z.p$ is black, the tree does not violate property 4 at loop termination. By the loop invariant, the only property that might fail to hold is property 2. Line 30 restores this property by coloring the root black, so that when RB-INSERT-FIXUP terminates, all the red-black properties hold.

Thus, we have shown that RB-INSERT-FIXUP correctly restores the red-black properties.

Analysis

What is the running time of RB-INSERT? Since the height of a red-black tree on n nodes is $O(\lg n)$, lines 1–16 of RB-INSERT take $O(\lg n)$

time. In RB-INSERTFIXUP, the **while** loop repeats only if case 1 occurs, and then the pointer z moves two levels up the tree. The total number of times the **while** loop can be executed is therefore $O(\lg n)$. Thus, RB-INSERT takes a total of $O(\lg n)$ time. Moreover, it never performs more than two rotations, since the **while** loop terminates if case 2 or case 3 is executed.

Exercises

13.3-1

Line 16 of RB-INSERT sets the color of the newly inserted node z to red. If instead z's color were set to black, then property 4 of a red-black tree would not be violated. Why not set z's color to black?

13.3-2

Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.

13.3-3

Suppose that the black-height of each of the subtrees α, β, γ, δ, ϵ in Figures 13.5 and 13.6 is k. Label each node in each figure with its black-height to verify that the indicated transformation preserves property 5.

13.3-4

Professor Teach is concerned that RB-INSERT-FIXUP might set *T.nil.color* to RED, in which case the test in line 1 would not cause the loop to terminate when z is the root. Show that the professor's concern is unfounded by arguing that RB-INSERT-FIXUP never sets *T.nil.color* to RED.

13.3-5

Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue that if $n > 1$, the tree has at least one red node.

13.3-6

Suggest how to implement RB-INSERT efficiently if the representation for red-black trees includes no storage for parent pointers.

13.4 Deletion

Like the other basic operations on an n-node red-black tree, deletion of a node takes $O(\lg n)$ time. Deleting a node from a red-black tree is more complicated than inserting a node.

The procedure for deleting a node from a red-black tree is based on the TREE-DELETE procedure on page 325. First, we need to customize the TRANSPLANT subroutine on page 324 that TREE-DELETE calls so that it applies to a red-black tree. Like TRANSPLANT, the new procedure RB-TRANSPLANT replaces the subtree rooted at node u by the subtree rooted at node v. The RB-TRANSPLANT procedure differs from TRANSPLANT in two ways. First, line 1 references the sentinel $T.nil$ instead of NIL. Second, the assignment to $v.p$ in line 6 occurs unconditionally: the procedure can assign to $v.p$ even if v points to the sentinel. We'll take advantage of the ability to assign to $v.p$ when $v = T.nil$.

```
RB-TRANSPLANT(T, u, v)

1 if u.p == T.nil
2     T.root = v
3 elseif u == u.p.left
4     u.p.left = v
5 else u.p.right = v
6 v.p = u.p
```

The procedure RB-DELETE on the next page is like the TREE-DELETE procedure, but with additional lines of pseudocode. The additional lines deal with nodes x and y that may be involved in violations of the red-black properties. When the node z being deleted has at most one child, then y will be z. When z has two children, then, as in TREE-DELETE, y will be z's successor, which has no left child and moves into z's position in the tree. Additionally, y takes on z's color. In either case, node y has at most one child: node x, which takes y's place in the tree. (Node x will be the sentinel $T.nil$ if y has no children.) Since node y will be either removed from the tree or moved within the tree, the

procedure needs to keep track of y's original color. If the red-black properties might be violated after deleting node z, RB-DELETE calls the auxiliary procedure RB-DELETE-FIXUP, which changes colors and performs rotations to restore the red-black properties.

Although RB-DELETE contains almost twice as many lines of pseudocode as TREE-DELETE, the two procedures have the same basic structure. You can find each line of TREE-DELETE within RB-DELETE (with the changes of replacing NIL by *T.nil* and replacing calls to TRANSPLANT by calls to RB-TRANSPLANT), executed under the same conditions.

In detail, here are the other differences between the two procedures:

- Lines 1 and 9 set node y as described above: line 1 when node z has at most one child and line 9 when z has two children.
- Because node y's color might change, the variable *y-original-color* stores y's color before any changes occur. Lines 2 and 10 set this variable immediately after assignments to y. When node z has two children, then nodes y and z are distinct. In this case, line 17 moves y into z's original position in the tree (that is, z's location in the tree at the time RB-DELETE was called), and line 20 gives y the same color as z. When node y was originally black, removing or moving it could cause violations of the red-black properties, which are corrected by the call of RB-DELETE-FIXUP in line 22.

```
RB-DELETE(T, z)
 1 y = z
 2 y-original-color = y.color
 3 if z.left == T.nil
 4     x = z.right
 5     RB-TRANSPLANT(T, z, z.right) // replace z by its right child
 6 elseif z.right == T.nil
 7     x = z.left
 8     RB-TRANSPLANT(T, z, z.left)  // replace z by its left child
 9 else y = TREE-MINIMUM(z.right)  // y is z's successor
10     y-original-color = y.color
```

```
11    x = y.right
12    if y ≠ z.right                    // is y farther down the tree?
13        RB-TRANSPLANT(T, y,           // replace y by its right child
              y.right)
14        y.right = z.right             // z's right child becomes
15        y.right.p = y                 //   y's right child
16    else x.p = y                      // in case x is T.nil
17    RB-TRANSPLANT(T, z, y)            // replace z by its successor y
18    y.left = z.left                   // and give z's left child to y,
19    y.left.p = y                      //       which had no left child
20    y.color = z.color
21 if y-original-color == BLACK         // if any red-black violations
                                           occurred,
22    RB-DELETE-FIXUP(T, x)             //      correct them
```

- As discussed, the procedure keeps track of the node x that moves into node y's original position at the time of call. The assignments in lines 4, 7, and 11 set x to point to either y's only child or, if y has no children, the sentinel $T.nil$.
- Since node x moves into node y's original position, the attribute $x.p$ must be set correctly. If node z has two children and y is z's right child, then y just moves into z's position, with x remaining a child of y. Line 12 checks for this case. Although you might think that setting $x.p$ to y in line 16 is unnecessary since x is a child of y, the call of RB-DELETE-FIXUP relies on $x.p$ being y even if x is $T.nil$. Thus, when z has two children and y is z's right child, executing line 16 is necessary if y's right child is $T.nil$, and otherwise it does not change anything.

  Otherwise, node z is either the same as node y or it is a proper ancestor of y's original parent. In these cases, the calls of RB-TRANSPLANT in lines 5, 8, and 13 set $x.p$ correctly in line 6 of RB-TRANSPLANT. (In these calls of RB-TRANSPLANT, the third parameter passed is the same as x.)
- Finally, if node y was black, one or more violations of the red-black properties might arise. The call of RB-DELETE-FIXUP in

line 22 restores the red-black properties. If y was red, the red-black properties still hold when y is removed or moved, for the following reasons:

1. No black-heights in the tree have changed. (See Exercise 13.4-1.)
2. No red nodes have been made adjacent. If z has at most one child, then y and z are the same node. That node is removed, with a child taking its place. If the removed node was red, then neither its parent nor its children can also be red, so moving a child to take its place cannot cause two red nodes to become adjacent. If, on the other hand, z has two children, then y takes z's place in the tree, along with z's color, so there cannot be two adjacent red nodes at y's new position in the tree. In addition, if y was not z's right child, then y's original right child x replaces y in the tree. Since y is red, x must be black, and so replacing y by x cannot cause two red nodes to become adjacent.
3. Because y could not have been the root if it was red, the root remains black.

If node y was black, three problems may arise, which the call of RB-DELETE-FIXUP will remedy. First, if y was the root and a red child of y became the new root, property 2 is violated. Second, if both x and its new parent are red, then a violation of property 4 occurs. Third, moving y within the tree causes any simple path that previously contained y to have one less black node. Thus, property 5 is now violated by any ancestor of y in the tree. We can correct the violation of property 5 by saying that when the black node y is removed or moved, its blackness transfers to the node x that moves into y's original position, giving x an "extra" black. That is, if we add 1 to the count of black nodes on any simple path that contains x, then under this interpretation, property 5 holds. But now another problem emerges: node x is neither red nor black, thereby violating property 1. Instead, node x is either "doubly black" or "red-and-black," and it contributes either 2 or 1, respectively, to the count of black nodes on simple paths containing x. The *color*

attribute of x will still be either RED (if x is red-and-black) or BLACK (if x is doubly black). In other words, the extra black on a node is reflected in x's pointing to the node rather than in the *color* attribute.

The procedure RB-DELETE-FIXUP on the next page restores properties 1, 2, and 4. Exercises 13.4-2 and 13.4-3 ask you to show that the procedure restores properties 2 and 4, and so in the remainder of this section, we focus on property 1. The goal of the **while** loop in lines 1–43 is to move the extra black up the tree until

1. x points to a red-and-black node, in which case line 44 colors x (singly) black;
2. x points to the root, in which case the extra black simply vanishes; or
3. having performed suitable rotations and recolorings, the loop exits.

Like RB-INSERT-FIXUP, the RB-DELETE-FIXUP procedure handles two symmetric situations: lines 3–22 for when node x is a left child, and lines 24–43 for when x is a right child. Our proof focuses on the four cases shown in lines 3–22.

Within the **while** loop, x always points to a nonroot doubly black node. Line 2 determines whether x is a left child or a right child of its parent $x.p$ so that either lines 3–22 or 24–43 will execute in a given iteration. The sibling of x is always denoted by a pointer w. Since node x is doubly black, node w cannot be $T.nil$, because otherwise, the number of blacks on the simple path from $x.p$ to the (singly black) leaf w would be smaller than the number on the simple path from $x.p$ to x.

Recall that the RB-DELETE procedure always assigns to $x.p$ before calling RB-DELETE-FIXUP (either within the call of RB-TRANSPLANT in line 13 or the assignment in line 16), even when node x is the sentinel $T.nil$. That is because RB-DELETE-FIXUP references x's parent $x.p$ in several places, and this attribute must point to the node that became x's parent in RB-DELETE—even if x is $T.nil$.

Figure 13.7 demonstrates the four cases in the code when node x is a left child. (As in RB-INSERT-FIXUP, the cases in RB-DELETE-FIXUP are not mutually exclusive.) Before examining each case in

detail, let's look more generally at how we can verify that the transformation in each of the cases preserves property 5. The key idea is that in each case, the transformation applied preserves the number of black nodes (including x's extra black) from (and including) the root of the subtree shown to the roots of each of the subtrees $\alpha, \beta, \ldots, \zeta$. Thus, if property 5 holds prior to the transformation, it continues to hold afterward. For example, in Figure 13.7(a), which illustrates case 1, the number of black nodes from the root to the root of either subtree α or β is 3, both before and after the transformation. (Again, remember that node x adds an extra black.) Similarly, the number of black nodes from the root to the root of any of γ, δ, ϵ, and ζ is 2, both before and after the transformation.[2] In Figure 13.7(b), the counting must involve the value c of the *color* attribute of the root of the subtree shown, which can be either RED or BLACK.

```
RB-DELETE-FIXUP(T, x)
 1 while x ≠ T.root and x.color == BLACK
 2     if x == x.p.left                          // is x a left child?
 3         w = x.p.right                         // w is x's sibling
 4         if w.color == RED
 5             w.color = BLACK                   } case 1
 6             x.p.color = RED                   }
 7             LEFT-ROTATE(T, x.p)               }
 8             w = x.p.right
 9         if w.left.color == BLACK and w.right.color ==
               BLACK
10             w.color = RED                     } case 2
11             x = x.p                           }
12         else
13             if w.right.color == BLACK
14                 w.left.color = BLACK          } case 3
15                 w.color = RED                 }
16                 RIGHT-ROTATE(T, w)            }
```

```
17                  w = x.p.right
18              w.color = x.p.color
19              x.p.color = BLACK                    } case 4
20              w.right.color = BLACK
21              LEFT-ROTATE(T, x.p)
22              x = T.root
23      else // same as lines 3–22, but with "right" and "left" exchanged
24          w = x.p.left
25          if w.color == RED
26              w.color = BLACK
27              x.p.color = RED
28              RIGHT-ROTATE(T, x.p)
29              w = x.p.left
30          if w.right.color == BLACK and w.left.color == BLACK
31              w.color = RED
32              x = x.p
33          else
34              if w.left.color == BLACK
35                  w.right.color = BLACK
36                  w.color = RED
37                  LEFT-ROTATE(T, w)
38                  w = x.p.left
39              w.color = x.p.color
40              x.p.color = BLACK
41              w.left.color = BLACK
42              RIGHT-ROTATE(T, x.p)
43              x = T.root
44  x.color = BLACK
```

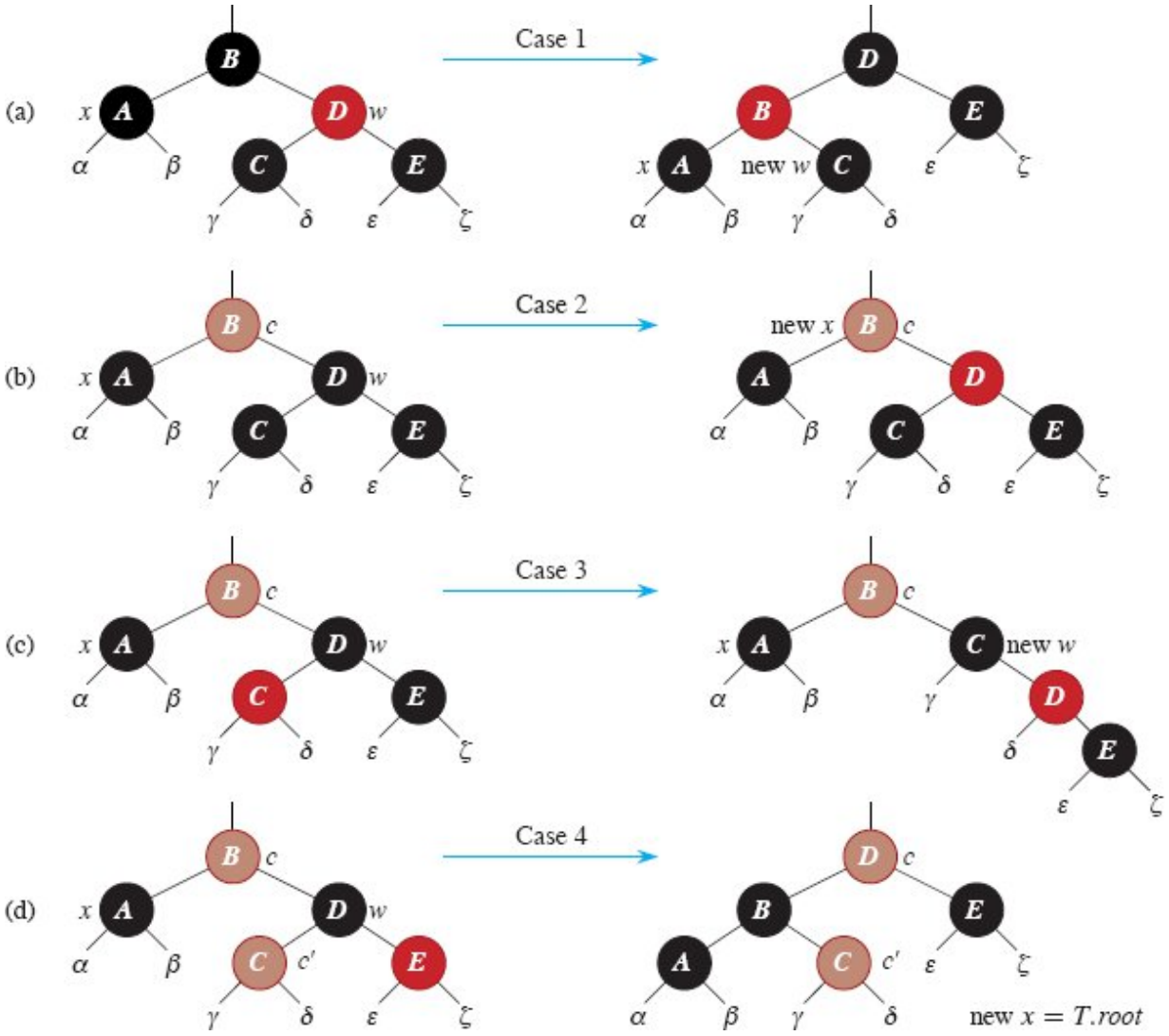

Figure 13.7 The cases in lines 3–22 of the procedure RB-DELETE-FIXUP. Brown nodes have *color* attributes represented by c and c', which may be either RED or BLACK. The letters α, β, …, ζ represent arbitrary subtrees. Each case transforms the configuration on the left into the configuration on the right by changing some colors and/or performing a rotation. Any node pointed to by x has an extra black and is either doubly black or red-and-black. Only case 2 causes the loop to repeat. **(a)** Case 1 is transformed into case 2, 3, or 4 by exchanging the colors of nodes B and D and performing a left rotation. **(b)** In case 2, the extra black represented by the pointer x moves up the tree by coloring node D red and setting x to point to node B. If case 2 is entered through case 1, the **while** loop terminates because the new node x is red-and-black, and therefore the value c of its *color* attribute is RED. **(c)** Case 3 is transformed to case 4 by exchanging the colors of nodes C and D and performing a right rotation. **(d)** Case 4 removes the extra black represented by x by changing some colors and performing a left rotation (without violating the red-black properties), and then the loop terminates.

If we define count(RED) = 0 and count(BLACK) = 1, then the number of black nodes from the root to α is 2 + count(c), both before and after the transformation. In this case, after the transformation, the new node x has *color* attribute c, but this node is really either red-and-black (if c = RED) or doubly black (if c = BLACK). You can verify the other cases similarly (see Exercise 13.4-6).

Case 1. x's sibling w is red

Case 1 (lines 5–8 and Figure 13.7(a)) occurs when node w, the sibling of node x, is red. Because w is red, it must have black children. This case switches the colors of w and $x.p$ and then performs a left-rotation on $x.p$ without violating any of the red-black properties. The new sibling of x, which is one of w's children prior to the rotation, is now black, and thus case 1 converts into one of cases 2, 3, or 4.

Cases 2, 3, and 4 occur when node w is black and are distinguished by the colors of w's children.

Case 2. x's sibling w is black, and both of w's children are black

In case 2 (lines 10–11 and Figure 13.7(b)), both of w's children are black. Since w is also black, this case removes one black from both x and w, leaving x with only one black and leaving w red. To compensate for x and w each losing one black, x's parent $x.p$ can take on an extra black. Line 11 does so by moving x up one level, so that the **while** loop repeats with $x.p$ as the new node x. If case 2 enters through case 1, the new node x is red-and-black, since the original $x.p$ was red. Hence, the value c of the *color* attribute of the new node x is RED, and the loop terminates when it tests the loop condition. Line 44 then colors the new node x (singly) black.

Case 3. x's sibling w is black, w's left child is red, and w's right child is black

Case 3 (lines 14–17 and Figure 13.7(c)) occurs when w is black, its left child is red, and its right child is black. This case switches the colors of w and its left child $w.left$ and then performs a right rotation on w without violating any of the red-black properties. The new sibling w of x is now a

black node with a red right child, and thus case 3 falls through into case 4.

Case 4. x's sibling w is black, and w's right child is red

Case 4 (lines 18–22 and Figure 13.7(d)) occurs when node x's sibling w is black and w's right child is red. Some color changes and a left rotation on $x.p$ allow the extra black on x to vanish, making it singly black, without violating any of the red-black properties. Line 22 sets x to be the root, and the **while** loop terminates when it next tests the loop condition.

Analysis

What is the running time of RB-DELETE? Since the height of a red-black tree of n nodes is $O(\lg n)$, the total cost of the procedure without the call to RB-DELETE-FIXUP takes $O(\lg n)$ time. Within RB-DELETE-FIXUP, each of cases 1, 3, and 4 lead to termination after performing a constant number of color changes and at most three rotations. Case 2 is the only case in which the **while** loop can be repeated, and then the pointer x moves up the tree at most $O(\lg n)$ times, performing no rotations. Thus, the procedure RB-DELETE-FIXUP takes $O(\lg n)$ time and performs at most three rotations, and the overall time for RB-DELETE is therefore also $O(\lg n)$.

Exercises

13.4-1

Show that if node y in RB-DELETE is red, then no black-heights change.

13.4-2

Argue that after RB-DELETE-FIXUP executes, the root of the tree must be black.

13.4-3

Argue that if in RB-DELETE both x and $x.p$ are red, then property 4 is restored by the call to RB-DELETE-FIXUP(T, x).

13.4-4

In Exercise 13.3-2 on page 346, you found the red-black tree that results from successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order 8, 12, 19, 31, 38, 41.

13.4-5
Which lines of the code for RB-DELETE-FIXUP might examine or modify the sentinel *T.nil*?

13.4-6
In each of the cases of Figure 13.7, give the count of black nodes from the root of the subtree shown to the roots of each of the subtrees α, β, …, ζ, and verify that each count remains the same after the transformation. When a node has a *color* attribute c or c', use the notation count(c) or count(c') symbolically in your count.

13.4-7
Professors Skelton and Baron worry that at the start of case 1 of RB-DELETE-FIXUP, the node $x.p$ might not be black. If $x.p$ is not black, then lines 5–6 are wrong. Show that $x.p$ must be black at the start of case 1, so that the professors need not be concerned.

13.4-8
A node x is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE. Is the resulting red-black tree always the same as the initial red-black tree? Justify your answer.

★ 13.4-9
Consider the operation RB-ENUMERATE(T, r, a, b), which outputs all the keys k such that $a \le k \le b$ in a subtree rooted at node r in an n-node red-black tree T. Describe how to implement RB-ENUMERATE in $\Theta(m + \lg n)$ time, where m is the number of keys that are output. Assume that the keys in T are unique and that the values a and b appear as keys in T. How does your solution change if a and b might not appear in T?

Problems

13-1 Persistent dynamic sets

During the course of an algorithm, you sometimes find that you need to maintain past versions of a dynamic set as it is updated. We call such a set ***persistent***. One way to implement a persistent set is to copy the entire set whenever it is modified, but this approach can slow down a program and also consume a lot of space. Sometimes, you can do much better.

Consider a persistent set S with the operations INSERT, DELETE, and SEARCH, which you implement using binary search trees as shown in Figure 13.8(a). Maintain a separate root for every version of the set. In order to insert the key 5 into the set, create a new node with key 5. This node becomes the left child of a new node with key 7, since you cannot modify the existing node with key 7. Similarly, the new node with key 7 becomes the left child of a new node with key 8 whose right child is the existing node with key 10. The new node with key 8 becomes, in turn, the right child of a new root r' with key 4 whose left child is the existing node with key 3. Thus, you copy only part of the tree and share some of the nodes with the original tree, as shown in Figure 13.8(b).

Assume that each tree node has the attributes *key*, *left*, and *right* but no parent. (See also Exercise 13.3-6 on page 346.)

a. For a persistent binary search tree (not a red-black tree, just a binary search tree), identify the nodes that need to change to insert or delete a node.

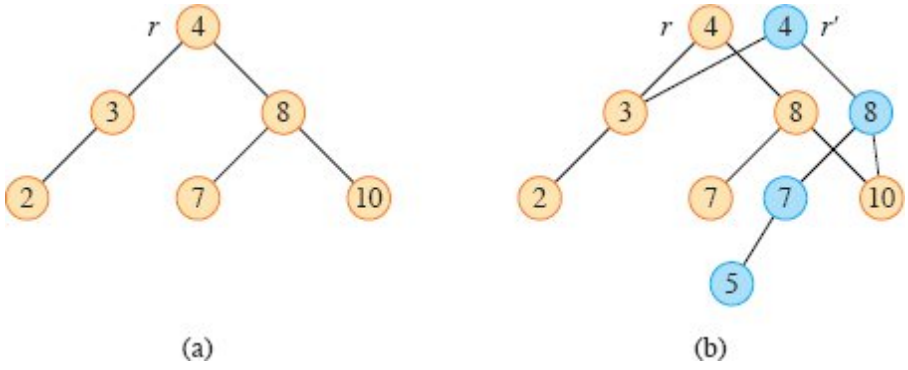

Figure 13.8 (a) A binary search tree with keys 2, 3, 4, 7, 8, 10. **(b)** The persistent binary search tree that results from the insertion of key 5. The most recent version of the set consists of the nodes reachable from the root r', and the previous version consists of the nodes reachable from r. Blue nodes are added when key 5 is inserted.

b. Write a procedure PERSISTENT-TREE-INSERT(T, z) that, given a persistent binary search tree T and a node z to insert, returns a new persistent tree T' that is the result of inserting z into T. Assume that you have a procedure COPY-NODE(x) that makes a copy of node x, including all of its attributes.

c. If the height of the persistent binary search tree T is h, what are the time and space requirements of your implementation of PERSISTENT-TREE-INSERT? (The space requirement is proportional to the number of nodes that are copied.)

d. Suppose that you include the parent attribute in each node. In this case, the PERSISTENT-TREE-INSERT procedure needs to perform additional copying. Prove that PERSISTENT-TREE-INSERT then requires $\Omega(n)$ time and space, where n is the number of nodes in the tree.

e. Show how to use red-black trees to guarantee that the worst-case running time and space are $O(\lg n)$ per insertion or deletion. You may assume that all keys are distinct.

13-2 Join operation on red-black trees

The ***join*** operation takes two dynamic sets S_1 and S_2 and an element x such that for any $x_1 \in S_1$ and $x_2 \in S_2$, we have $x_1.key \le x.key \le x_2.key$. It returns a set $S = S_1 \cup \{x\} \cup S_2$. In this problem, we investigate how to implement the join operation on red-black trees.

a. Suppose that you store the black-height of a red-black tree T as the new attribute $T.bh$. Argue that RB-INSERT and RB-DELETE can maintain the bh attribute without requiring extra storage in the nodes of the tree and without increasing the asymptotic running times. Show how to determine the black-height of each node visited while descending through T, using $O(1)$ time per node visited.

Let T_1 and T_2 be red-black trees and x be a key value such that for any nodes x_1 in T_1 and x_2 in T_2, we have $x_1.key \le x.key \le x_2.key$. You will show how to implement the operation RB-JOIN(T_1, x, T_2), which

destroys T_1 and T_2 and returns a red-black tree $T = T_1 \cup \{x\} \cup T_2$. Let n be the total number of nodes in T_1 and T_2.

b. Assume that $T_1.bh \geq T_2.bh$. Describe an $O(\lg n)$-time algorithm that finds a black node y in T_1 with the largest key from among those nodes whose black-height is $T_2.bh$.

c. Let T_y be the subtree rooted at y. Describe how $T_y \cup \{x\} \cup T_2$ can replace T_y in $O(1)$ time without destroying the binary-search-tree property.

d. What color should you make x so that red-black properties 1, 3, and 5 are maintained? Describe how to enforce properties 2 and 4 in $O(\lg n)$ time.

e. Argue that no generality is lost by making the assumption in part (b). Describe the symmetric situation that arises when $T_1.bh \leq T_2.bh$.

f. Argue that the running time of RB-JOIN is $O(\lg n)$.

13-3 AVL trees

An ***AVL tree*** is a binary search tree that is ***height balanced***: for each node x, the heights of the left and right subtrees of x differ by at most 1. To implement an AVL tree, maintain an extra attribute h in each node such that $x.h$ is the height of node x. As for any other binary search tree T, assume that $T.root$ points to the root node.

a. Prove that an AVL tree with n nodes has height $O(\lg n)$. (*Hint:* Prove that an AVL tree of height h has at least F_h nodes, where F_h is the hth Fibonacci number.)

b. To insert into an AVL tree, first place a node into the appropriate place in binary search tree order. Afterward, the tree might no longer be height balanced. Specifically, the heights of the left and right children of some node might differ by 2. Describe a procedure BALANCE(x), which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ by at most 2,

so that $|x.right.h - x.left.h| \le 2$, and alters the subtree rooted at x to be height balanced. The procedure should return a pointer to the node that is the root of the subtree after alterations occur. (*Hint:* Use rotations.)

c. Using part (b), describe a recursive procedure AVL-INSERT(T, z) that takes an AVL tree T and a newly created node z (whose key has already been filled in), and adds z into T, maintaining the property that T is an AVL tree. As in TREE-INSERT from Section 12.3, assume that $z.key$ has already been filled in and that $z.left$ = NIL and $z.right$ = NIL. Assume as well that $z.h = 0$.

d. Show that AVL-INSERT, run on an n-node AVL tree, takes $O(\lg n)$ time and performs $O(\lg n)$ rotations.

Chapter notes

The idea of balancing a search tree is due to Adel'son-Vel'skiĭ and Landis [2], who introduced a class of balanced search trees called "AVL trees" in 1962, described in Problem 13-3. Another class of search trees, called "2-3 trees," was introduced by J. E. Hopcroft (unpublished) in 1970. A 2-3 tree maintains balance by manipulating the degrees of nodes in the tree, where each node has either two or three children. Chapter 18 covers a generalization of 2-3 trees introduced by Bayer and McCreight [39], called "B-trees."

Red-black trees were invented by Bayer [38] under the name "symmetric binary B-trees." Guibas and Sedgewick [202] studied their properties at length and introduced the red/black color convention. Andersson [16] gives a simpler-to-code variant of red-black trees. Weiss [451] calls this variant AA-trees. An AA-tree is similar to a red-black tree except that left children can never be red.

Sedgewick and Wayne [402] present red-black trees as a modified version of 2-3 trees in which a node with three children is split into two nodes with two children each. One of these nodes becomes the left child of the other, and only left children can be red. They call this structure a "left-leaning red-black binary search tree." Although the code for left-

leaning red-black binary search trees is more concise than the red-black tree pseudocode in this chapter, operations on left-leaning red-black binary search trees do not limit the number of rotations per operation to a constant. This distinction will matter in Chapter 17.

Treaps, a hybrid of binary search trees and heaps, were proposed by Seidel and Aragon [404]. They are the default implementation of a dictionary in LEDA [324], which is a well-implemented collection of data structures and algorithms.

There are many other variations on balanced binary trees, including weight-balanced trees [344], k-neighbor trees [318], and scapegoat trees [174]. Perhaps the most intriguing are the "splay trees" introduced by Sleator and Tarjan [418], which are "self-adjusting." (See Tarjan [429] for a good description of splay trees.) Splay trees maintain balance without any explicit balance condition such as color. Instead, "splay operations" (which involve rotations) are performed within the tree every time an access is made. The amortized cost (see Chapter 16) of each operation on an n-node tree is $O(\lg n)$. Splay trees have been conjectured to perform within a constant factor of the best offline rotation-based tree. The best known competitive ratio (see Chapter 27) for a rotation-based tree is the Tango Tree of Demaine et al. [109].

Skip lists [369] provide an alternative to balanced binary trees. A skip list is a linked list that is augmented with a number of additional pointers. Each dictionary operation runs in $O(\lg n)$ expected time on a skip list of n items.

---

[1] Although we try to avoid gendered language in this book, the English language lacks a gender-neutral word for a parent's sibling.

[2] If property 5 holds, we can assume that paths from the roots of γ, δ, ϵ, and ζ down to leaves contain one more black than do paths from the roots of α and β down to leaves.

Part IV Advanced Design and Analysis Techniques