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.

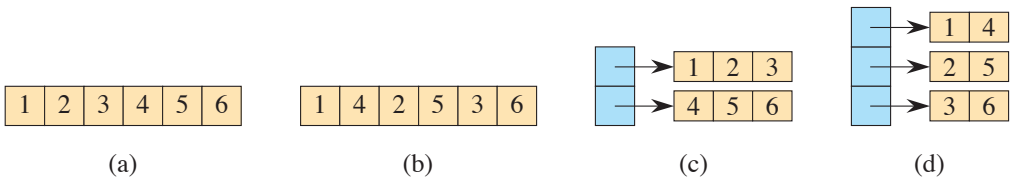

Figure 10.1 Four ways to store the 2×3 matrix M from equation (10.1). **(a)** In row-major order, in a single array. **(b)** In column-major order, in a single array. **(c)** In row-major order, with one array per row (tan) and a single array (blue) of pointers to the row arrays. **(d)** In column-major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

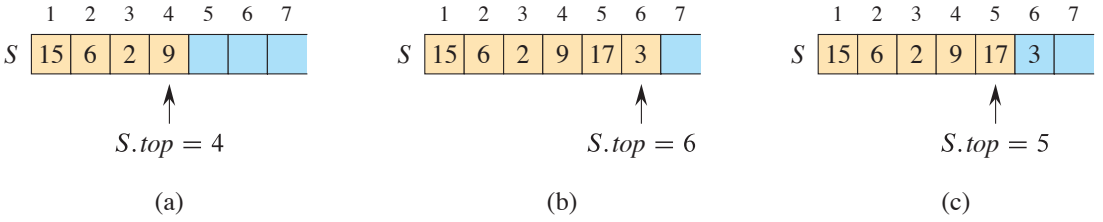

Figure 10.2 An array implementation of a stack S. Stack elements appear only in the tan positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH$(S, 17)$ and PUSH$(S, 3)$. **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most 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]
```

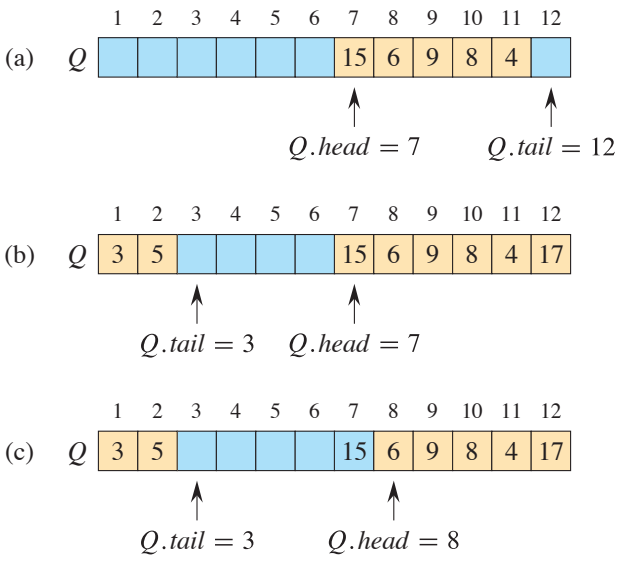

Figure 10.3 A queue implemented using an array $Q[1:12]$. Queue elements appear only in the tan positions. **(a)** The queue has 5 elements, in locations $Q[7:11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The configuration 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 $Q.head$, $Q.head + 1$, $\ldots$, $Q.tail - 1$, 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

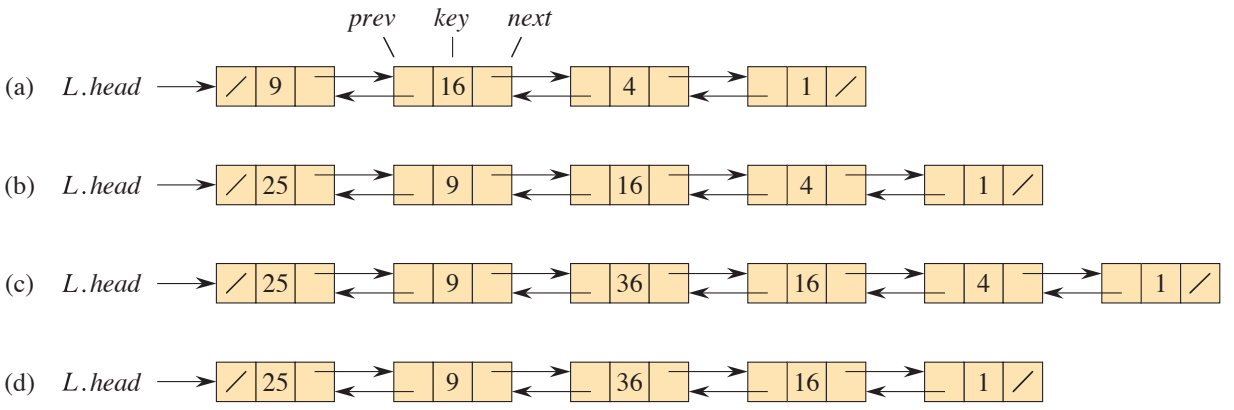

Figure 10.4 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, 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.

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 $x.prev = \text{NIL}$, the element x has no predecessor and is therefore the first element, or ***head***, of the list. If $x.next = \text{NIL}$, 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 $L.head = \text{NIL}$, the list is empty.

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:

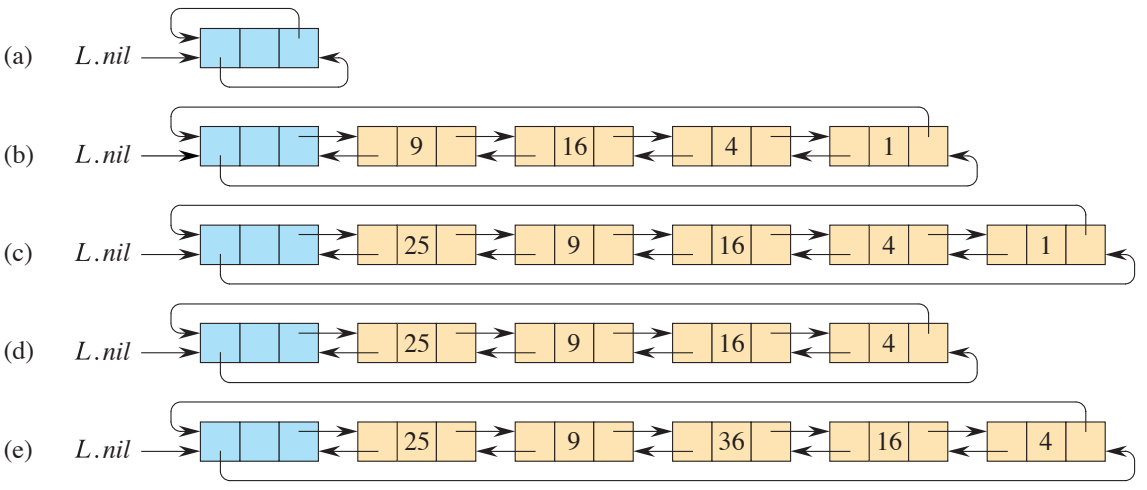

Figure 10.5 A circular, doubly linked list with a sentinel. The sentinel $L.nil$, in blue, appears between the head and tail. The attribute $L.head$ is no longer needed, since the head of the list is $L.nil.next$. **(a)** An empty list. **(b)** The linked list from Figure 10.4(a), with key 9 at the head 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 $x.p = \text{NIL}$, then x is the root. If node x has no left child, then $x.left = \text{NIL}$, and similarly for the right child. The root of the entire tree T is pointed to by the attribute $T.root$. If $T.root = \text{NIL}$, 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 $child_1, child_2, \ldots, child_k$. 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 $x.left\text{-}child = \text{NIL}$, and if node x is the rightmost child of its parent, then $x.right\text{-}sibling = \text{NIL}$.

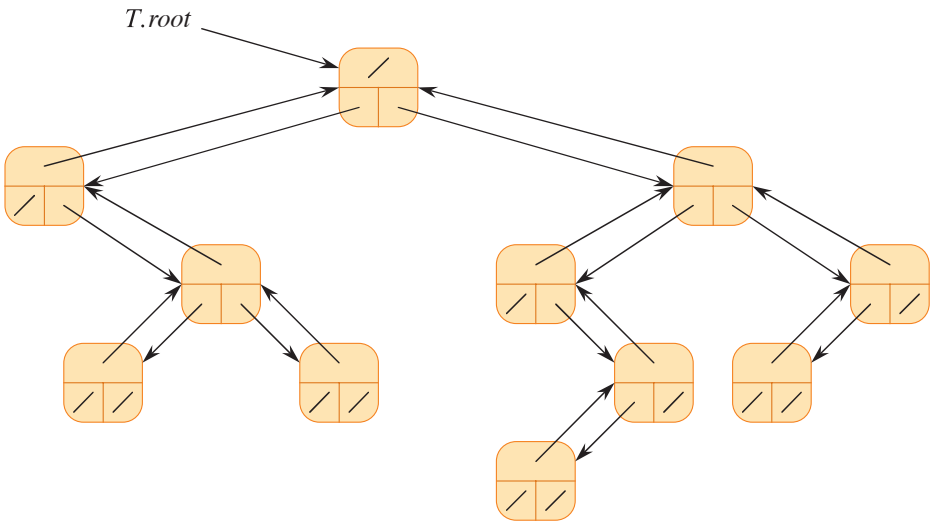

Figure 10.6 The representation of a binary tree T. Each node x has the attributes $x.p$ (top), $x.left$ (lower left), and $x.right$ (lower right). The *key* attributes are not shown.

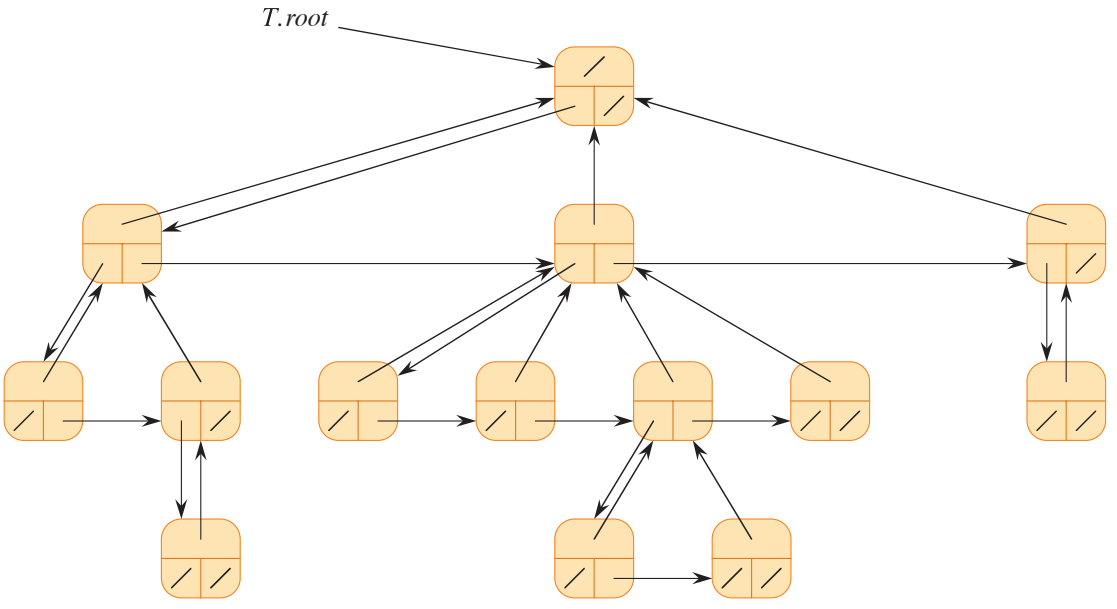

Figure 10.7 The left-child, right-sibling representation of a tree T. Each node x has attributes $x.p$ (top), $x.left\text{-}child$ (lower left), and $x.right\text{-}sibling$ (lower right). The *key* attributes are not shown.

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]

[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***.

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 $\text{RANDOM}(1, n)$ 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 $O(t + \text{E}[X_t])$.

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

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

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

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.