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 five chapters present some basic techniques for representing finite dynamic sets and manipulating them on a computer.

Algorithms may require several different 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. A dynamic set that supports these operations is called a ***dictionary***. Other algorithms require more complicated operations. For example, min-priority queues, which were introduced in Chapter 6 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 must be supported.

Elements of a dynamic set

In a typical implementation of a dynamic set, each element is represented by an object whose fields can be examined and manipulated if we have a pointer to the object. (Section 10.3 discusses the implementation of objects and pointers in programming environments that do not contain them as basic data types.) Some kinds of dynamic sets assume that one of the object's fields is an identifying ***key*** field. 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 fields but are otherwise unused by the set implementation. It may also have

fields that are manipulated by the set operations; these fields 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 totally ordered set satisfies the trichotomy property, defined on page 49.) A total ordering allows us to define the minimum element of the set, for example, or 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 $key[x] = k$, or NIL if no such element belongs to S.

INSERT(S, x)
: A modifying operation that augments the set S with the element pointed to by x. We usually assume that any fields 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 uses a pointer to an element x, not a key value.)

MINIMUM(S)
: A query on a totally ordered set S that returns a pointer to the element of S with the smallest key.

MAXIMUM(S)
: A query on a totally ordered set S that returns a pointer to the element of S with the largest 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.

The queries SUCCESSOR and PREDECESSOR are often extended 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.

The time taken to execute a set operation is usually measured in terms of the size of the set given as one of its arguments. For example, Chapter 13 describes a data structure that can support any of the operations listed above on a set of size n in time $O(\lg n)$.

Overview of Part III

Chapters 10–14 describe several data structures that can be used to implement dynamic sets; many of these will be used later to construct efficient algorithms for a variety of problems. Another important data structure—the heap—has already been introduced in Chapter 6.

Chapter 10 presents the essentials of working with simple data structures such as stacks, queues, linked lists, and rooted trees. It also shows how objects and pointers can be implemented in programming environments that do not support them as primitives. Much of this material should be familiar to anyone who has taken an introductory programming course.

Chapter 11 introduces hash tables, which support the dictionary operations INSERT, DELETE, and SEARCH. In the worst case, hashing requires $\Theta(n)$ time to perform a SEARCH operation, but the expected time for hash-table operations is $O(1)$. The analysis of hashing relies on probability, but most of the chapter requires no background in the subject.

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, but on a randomly built binary search tree, the expected time for each operation is $O(\lg n)$. Binary search trees serve as the basis for many other data structures.

Red-black trees, a variant of binary search trees, are introduced in Chapter 13. 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 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, walking through the code can be quite instructive.

In Chapter 14, we show how to augment red-black trees to support operations other than the basic ones listed above. First, we augment them so that we can dynamically maintain order statistics for a set of keys. Then, we augment them in a different way to maintain intervals of real numbers.

10 Elementary Data Structures

In this chapter, we examine the representation of dynamic sets by simple data structures that use pointers. Although many complex data structures can be fashioned using pointers, we present only the rudimentary ones: stacks, queues, linked lists, and rooted trees. We also discuss a method by which objects and pointers can be synthesized from arrays.

10.1 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. In this section we show how to use a simple array to implement each.

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.

As shown in Figure 10.1, we can implement a stack of at most n elements with an array $S[1..n]$. The array has an attribute $top[S]$ that indexes the most recently inserted element. The stack consists of elements $S[1..top[S]]$, where $S[1]$ is the element at the bottom of the stack and $S[top[S]]$ is the element at the top.

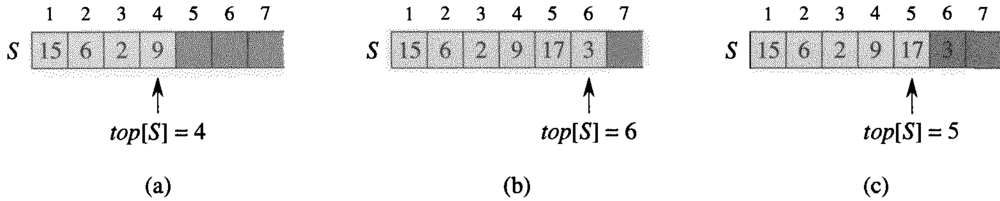

Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH(S, 17) and PUSH(S, 3). **(c)** Stack S after the call POP(S) has returned the element 3, which is the 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 $top[S] = 0$, the stack contains no elements and is ***empty***. The stack can be tested for emptiness by the query operation STACK-EMPTY. If an empty stack is popped, we say the stack ***underflows***, which is normally an error. If $top[S]$ exceeds n, the stack ***overflows***. (In our pseudocode implementation, we don't worry about stack overflow.)

The stack operations can each be implemented with a few lines of code.

```
STACK-EMPTY(S)
1  if top[S] = 0
2     then return TRUE
3     else return FALSE
```

```
PUSH(S, x)
1  top[S] ← top[S] + 1
2  S[top[S]] ← x
```

```
POP(S)
1  if STACK-EMPTY(S)
2     then error "underflow"
3     else top[S] ← top[S] − 1
4          return S[top[S] + 1]
```

Figure 10.1 shows the effects of the modifying operations PUSH and POP. Each of the three stack operations takes $O(1)$ time.

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

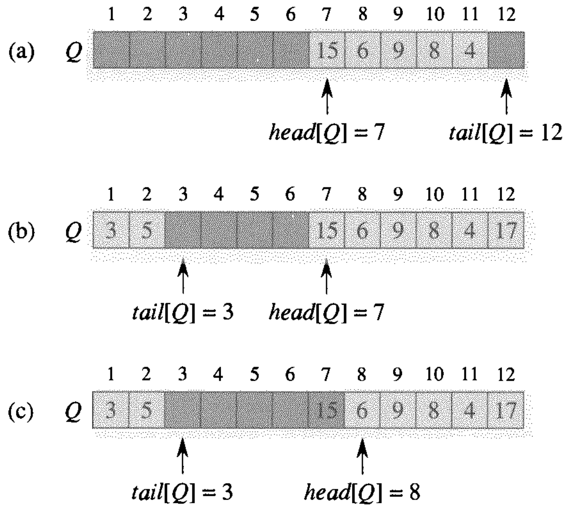

Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE(Q, 17), ENQUEUE(Q, 3), and ENQUEUE(Q, 5). **(c)** The configuration 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.

argument. The FIFO property of a queue causes it to operate like a line of people in the registrar's office. 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 student takes a place at the end of the line. The element dequeued is always the one at the head of the queue, like the student at the head of the line who has waited the longest. (Fortunately, we don't have to worry about computational elements cutting into line.)

Figure 10.2 shows one way to implement a queue of at most $n - 1$ elements using an array $Q[1..n]$. The queue has an attribute $head[Q]$ that indexes, or points to, its head. The attribute $tail[Q]$ indexes the next location at which a newly arriving element will be inserted into the queue. The elements in the queue are in locations $head[Q], head[Q]+1, \ldots, tail[Q] - 1$, where we "wrap around" in the sense that location 1 immediately follows location n in a circular order. When $head[Q] = tail[Q]$, the queue is empty. Initially, we have $head[Q] = tail[Q] = 1$. When the queue is empty, an attempt to dequeue an element causes the queue to underflow. When $head[Q] = tail[Q] + 1$, the queue is full, and an attempt to enqueue an element causes the queue to overflow.

In our procedures ENQUEUE and DEQUEUE, the error checking for underflow and overflow has been omitted. (Exercise 10.1-4 asks you to supply code that checks for these two error conditions.)

```
ENQUEUE(Q, x)
1  Q[tail[Q]] ← x
2  if tail[Q] = length[Q]
3     then tail[Q] ← 1
4     else tail[Q] ← tail[Q] + 1
```

```
DEQUEUE(Q)
1  x ← Q[head[Q]]
2  if head[Q] = length[Q]
3     then head[Q] ← 1
4     else head[Q] ← head[Q] + 1
5  return x
```

Figure 10.2 shows the effects of the ENQUEUE and DEQUEUE operations. Each operation takes $O(1)$ time.

Exercises

10.1-1

Using Figure 10.1 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 $S[1..6]$.

10.1-2

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.

10.1-3

Using Figure 10.2 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 $Q[1..6]$.

10.1-4

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

10.1-5

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) 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 constructed from an array.

10.1-6

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

10.1-7

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, though, 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. Linked lists provide a simple, flexible representation for dynamic sets, supporting (though not necessarily efficiently) all the operations listed on page 198.

As shown in Figure 10.3, each element of a ***doubly linked list*** L is an object with a *key* field and two other pointer fields: *next* and *prev*. The object may also contain other satellite data. Given an element x in the list, $next[x]$ points to its successor in the linked list, and $prev[x]$ points to its predecessor. If $prev[x] = \text{NIL}$, the element x has no predecessor and is therefore the first element, or ***head***, of the list. If $next[x] = \text{NIL}$, the element x has no successor and is therefore the last element, or ***tail***, of the list. An attribute $head[L]$ points to the first element of the list. If $head[L] = \text{NIL}$, 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***, we omit the *prev* pointer in each element. 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 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. The list may thus be viewed as a ring of elements. In the remainder of this section, we assume that the lists with which we are working are unsorted and doubly linked.

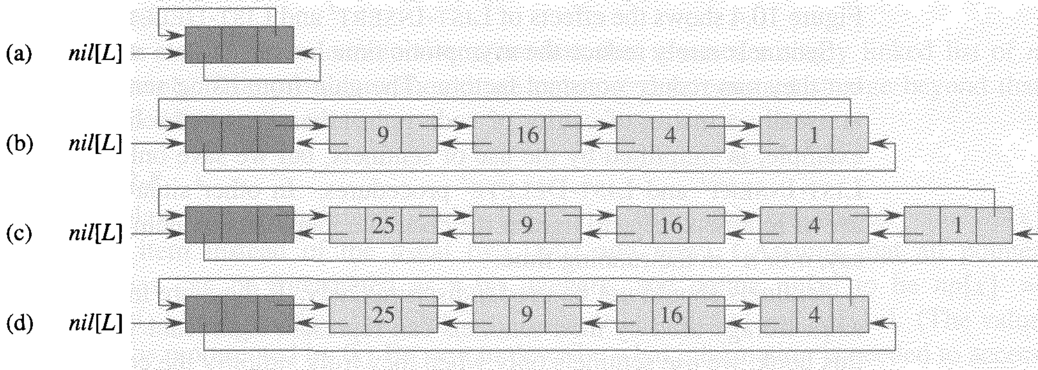

Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel $nil[L]$ appears between the head and tail. The attribute $head[L]$ is no longer needed, since we can access the head of the list by $next[nil[L]]$. **(a)** An empty list. **(b)** The linked list from Figure 10.3(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT$'(L, x)$, where $key[x] = 25$. 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.

tail; the field $next[nil[L]]$ points to the head of the list, and $prev[nil[L]]$ points to the tail. Similarly, both the *next* field of the tail and the *prev* field of the head point to $nil[L]$. Since $next[nil[L]]$ points to the head, we can eliminate the attribute $head[L]$ altogether, replacing references to it by references to $next[nil[L]]$. An empty list consists of just the sentinel, since both $next[nil[L]]$ and $prev[nil[L]]$ can be set to $nil[L]$.

The code for LIST-SEARCH remains the same as before, but with the references to NIL and $head[L]$ changed as specified above.

LIST-SEARCH$'(L, k)$

1 $x \leftarrow next[nil[L]]$
2 **while** $x \neq nil[L]$ and $key[x] \neq k$
3   **do** $x \leftarrow next[x]$
4 **return** x

We use the two-line procedure LIST-DELETE$'$ to delete an element from the list. We use the following procedure to insert an element into the list.

LIST-INSERT$'(L, x)$

1 $next[x] \leftarrow next[nil[L]]$
2 $prev[next[nil[L]]] \leftarrow x$
3 $next[nil[L]] \leftarrow x$
4 $prev[x] \leftarrow nil[L]$

Figure 10.4 shows the effects of LIST-INSERT′ and LIST-DELETE′ on a sample list.

Sentinels rarely reduce the asymptotic time bounds of data structure operations, but they can reduce constant factors. The gain from using sentinels within loops is usually a matter of clarity of code rather than speed; the linked list code, for example, is simplified by the use of sentinels, but we save only $O(1)$ time in the LIST-INSERT′ and LIST-DELETE′ procedures. In other situations, however, the use of sentinels helps to tighten the code in a loop, thus reducing the coefficient of, say, n or n^2 in the running time.

Sentinels should not be used indiscriminately. If 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 truly simplify the code.

Exercises

10.2-1

Can the dynamic-set operation INSERT be implemented on a singly linked list in $O(1)$ time? How about DELETE?

10.2-2

Implement a stack using a singly linked list L. The operations PUSH and POP should still take $O(1)$ time.

10.2-3

Implement a queue by a singly linked list L. The operations ENQUEUE and DEQUEUE should still take $O(1)$ time.

10.2-4

As written, each loop iteration in the LIST-SEARCH′ procedure requires two tests: one for $x \neq nil[L]$ and one for $key[x] \neq k$. Show how to eliminate the test for $x \neq nil[L]$ in each iteration.

10.2-5

Implement the dictionary operations INSERT, DELETE, and SEARCH using singly linked, circular lists. What are the running times of your procedures?

10.2-6

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.