
10

Elementary Data Structures

In this chapter, we examine the representation of dynamic sets by simple data structures that use pointers. Although we can construct many complex data structures using pointers, we present only the rudimentary ones: stacks, queues, linked lists, and rooted trees. We also show ways to synthesize objects and pointers 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 Figure 10.1 shows, we can implement a stack of at most n elements with an array $S[1..n]$. The array has an attribute $S.top$ that indexes the most recently

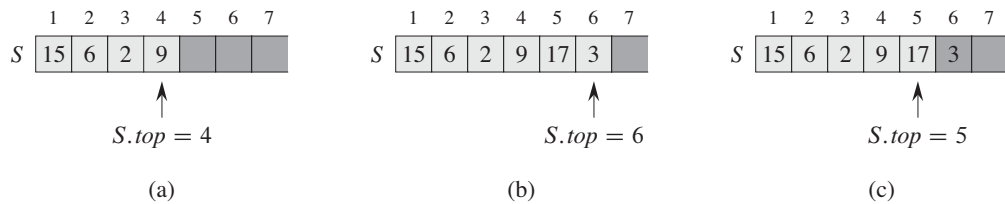


Figure 10.1 An array implementation of a stack S . Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls $PUSH(S, 17)$ and $PUSH(S, 3)$. **(c)** Stack S after the call $POP(S)$ has returned the element 3, which 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.

inserted element. The stack consists of elements $S[1 \dots S.top]$, where $S[1]$ is the element at the bottom of the stack and $S[S.top]$ is the element at the top.

When $S.top = 0$, the stack contains no elements and is *empty*. We can test to see whether the stack is empty by query operation `STACK-EMPTY`. If we attempt to pop an empty stack, we say the stack *underflows*, which is normally an error. If $S.top$ exceeds n , the stack *overflows*. (In our pseudocode implementation, we don't worry about stack overflow.)

We can implement each of the stack operations with just a few lines of code:

`STACK-EMPTY(S)`

```

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

`PUSH(S, x)`

```

1   $S.top = S.top + 1$ 
2   $S[S.top] = x$ 
```

`POP(S)`

```

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

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

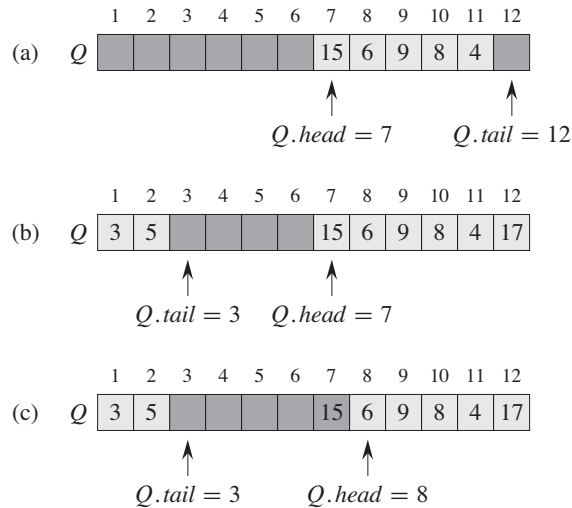


Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. (a) The queue has 5 elements, in locations $Q[7..11]$. (b) The configuration of the queue after the calls $ENQUEUE(Q, 17)$, $ENQUEUE(Q, 3)$, and $ENQUEUE(Q, 5)$. (c) The configuration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

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 to pay a cashier. 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 10.2 shows one way to implement a queue of at most $n - 1$ elements using an array $Q[1..n]$. The queue has an attribute $Q.head$ 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 $Q.head, Q.head + 1, \dots, Q.tail - 1$, 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 $Q.head = Q.tail = 1$. If we attempt to dequeue an element from an empty queue, the queue underflows.

When $Q.head = Q.tail + 1$, the queue is full, and if we attempt to enqueue an element, then the queue overflows.

In our procedures ENQUEUE and DEQUEUE, we have omitted the error checking for underflow and overflow. (Exercise 10.1-4 asks you to supply code that checks for these two error conditions.) The pseudocode assumes that $n = Q.length$.

```

ENQUEUE( $Q, x$ )
1   $Q[Q.tail] = x$ 
2  if  $Q.tail == Q.length$ 
3       $Q.tail = 1$ 
4  else  $Q.tail = Q.tail + 1$ 

DEQUEUE( $Q$ )
1   $x = Q[Q.head]$ 
2  if  $Q.head == Q.length$ 
3       $Q.head = 1$ 
4  else  $Q.head = Q.head + 1$ 
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 implemented by 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, 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. Linked lists provide a simple, flexible representation for dynamic sets, supporting (though not necessarily efficiently) all the operations listed on page 230.

As shown in Figure 10.3, each element of a **doubly linked list** L is an object with an attribute *key* and two other 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 $x.prev = \text{NIL}$, 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**, 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 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. We can think of a circular list as a ring of

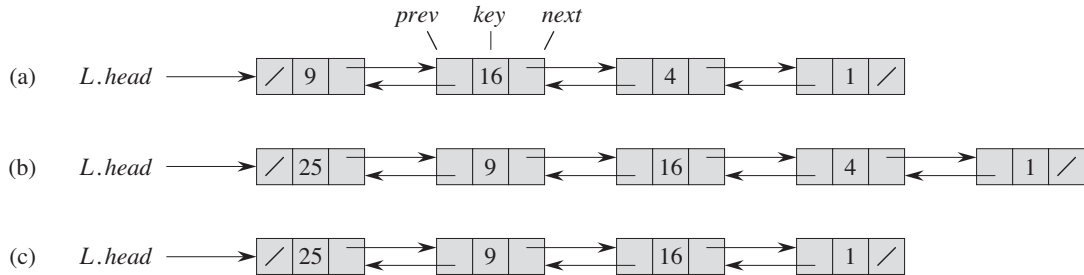


Figure 10.3 (a) A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. (b) Following the execution of $LIST-INSERT(L, x)$, where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. (c) The result of the subsequent call $LIST-DELETE(L, x)$, where x points to the object with key 4.

elements. In the remainder of this section, we assume that the lists with which we are working 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 10.3(a), the call $LIST-SEARCH(L, 4)$ returns a pointer to the third element, and the call $LIST-SEARCH(L, 7)$ returns NIL.

$LIST-SEARCH(L, k)$

```

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

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.

Inserting into a linked list

Given an element x whose *key* attribute has already been set, the $LIST-INSERT$ procedure “splices” x onto the front of the linked list, as shown in Figure 10.3(b).

```

LIST-INSERT( $L, x$ )
1   $x.next = L.head$ 
2  if  $L.head \neq \text{NIL}$ 
3       $L.head.prev = x$ 
4   $L.head = x$ 
5   $x.prev = \text{NIL}$ 

```

(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-INSERT on a list of n elements is $O(1)$.

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. If we wish to delete an element with a given key, we must first call LIST-SEARCH to retrieve a pointer to the element.

```

LIST-DELETE( $L, x$ )
1  if  $x.prev \neq \text{NIL}$ 
2       $x.prev.next = x.next$ 
3  else  $L.head = x.next$ 
4  if  $x.next \neq \text{NIL}$ 
5       $x.next.prev = x.prev$ 

```

Figure 10.3(c) shows how an element is deleted from a linked list. LIST-DELETE runs in $O(1)$ time, but if we wish to delete an element with a given key, $\Theta(n)$ time is required in the worst case because we must first call LIST-SEARCH to find the element.

Sentinels

The code for LIST-DELETE would be simpler if we could ignore the boundary conditions at the head and tail of the list:

```

LIST-DELETE'( $L, x$ )
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. For example, suppose that we provide with list L an object $L.nil$ that represents NIL

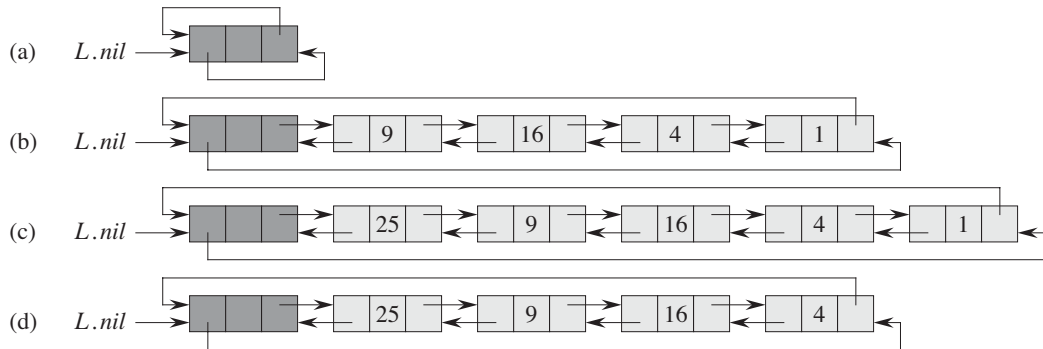


Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel $L.nil$ appears between the head and tail. The attribute $L.head$ is no longer needed, since we can access the head of the list by $L.nil.next$. (a) An empty list. (b) The linked list from Figure 10.3(a), with key 9 at the head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

but has all the attributes of the other objects in the list. Wherever we have a reference to NIL in list code, we replace it by a reference to the sentinel $L.nil$. As shown in Figure 10.4, 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, we can eliminate the attribute $L.head$ altogether, replacing references to it by references to $L.nil.next$. Figure 10.4(a) shows that an empty list consists of just the sentinel, and both $L.nil.next$ and $L.nil.prev$ point to $L.nil$.

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

```
LIST-SEARCH'(L, k)
1   $x = L.nil.next$ 
2  while  $x \neq L.nil$  and  $x.key \neq k$ 
3       $x = x.next$ 
4  return  $x$ 
```

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

LIST-INSERT'(L, x)

```
1   $x.next = L.nil.next$   
2   $L.nil.next.prev = x$   
3   $L.nil.next = x$   
4   $x.prev = L.nil$ 
```

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, becomes simpler when we use 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.

We should use sentinels 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 truly simplify the code.

Exercises

10.2-1

Can you implement the dynamic-set operation INSERT 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 $x \neq L.nil$ and one for $x.key \neq k$. Show how to eliminate the test for $x \neq L.nil$ 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.

10.2-7

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.

10.2-8 ★

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 $x.np$ to be $x.np = x.next \text{ XOR } x.prev$, 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 Implementing pointers and objects

How do we implement pointers and objects in languages that do not provide them? In this section, we shall see two ways of implementing linked data structures without an explicit pointer data type. We shall synthesize objects and pointers from arrays and array indices.

A multiple-array representation of objects

We can represent a collection of objects that have the same attributes by using an array for each attribute. As an example, Figure 10.5 shows how we can implement the linked list of Figure 10.3(a) with three arrays. The array *key* holds the values of the keys currently in the dynamic set, and the pointers reside in the arrays *next* and *prev*. For a given array index x , the array entries $key[x]$, $next[x]$, and $prev[x]$ represent an object in the linked list. Under this interpretation, a pointer x is simply a common index into the *key*, *next*, and *prev* arrays.

In Figure 10.3(a), the object with key 4 follows the object with key 16 in the linked list. In Figure 10.5, key 4 appears in $key[2]$, and key 16 appears in $key[5]$, and so $next[5] = 2$ and $prev[2] = 5$. Although the constant NIL appears in the *next*