

Introduction to Algorithms
Second Edition

by

Cormen, Leiserson, Rivest & Stein

Chapter 10

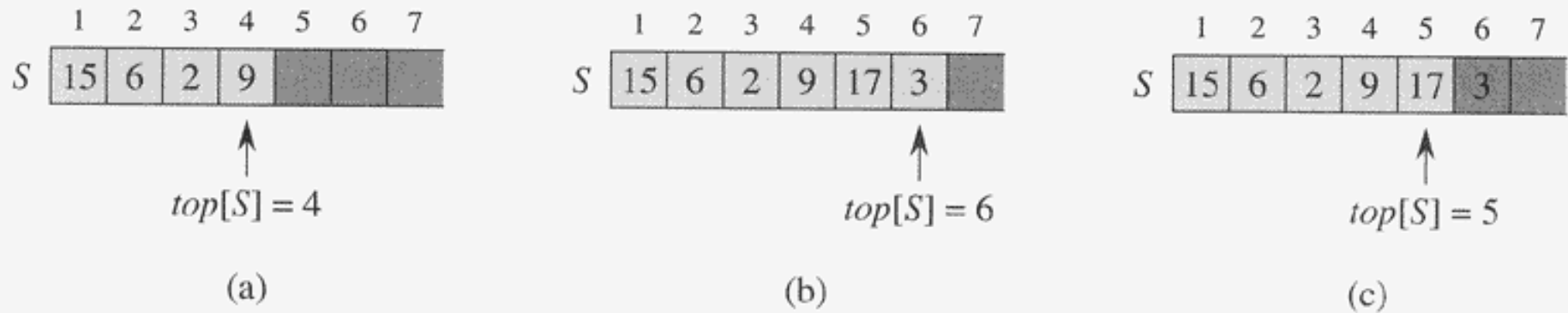


Figure 10.1 An array implementation of a stack S . Stack elements appear only in the lightly shaded positions. (a) Stack S has 4 elements. The top element is 9. (b) Stack S after the calls $PUSH(S, 17)$ and $PUSH(S, 3)$. (c) Stack S after the call $POP(S)$ has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

STACK-EMPTY(S)

1 if $top[S] = 0$

2 then return TRUE

3 else return FALSE

PUSH(S, x)

1 $top[S] \leftarrow top[S] + 1$

2 $S[top[S]] \leftarrow x$

POP(S)

```
1  if STACK-EMPTY( $S$ )  
2      then error “underflow”  
3      else  $top[S] \leftarrow top[S] - 1$   
4          return  $S[top[S] + 1]$ 
```

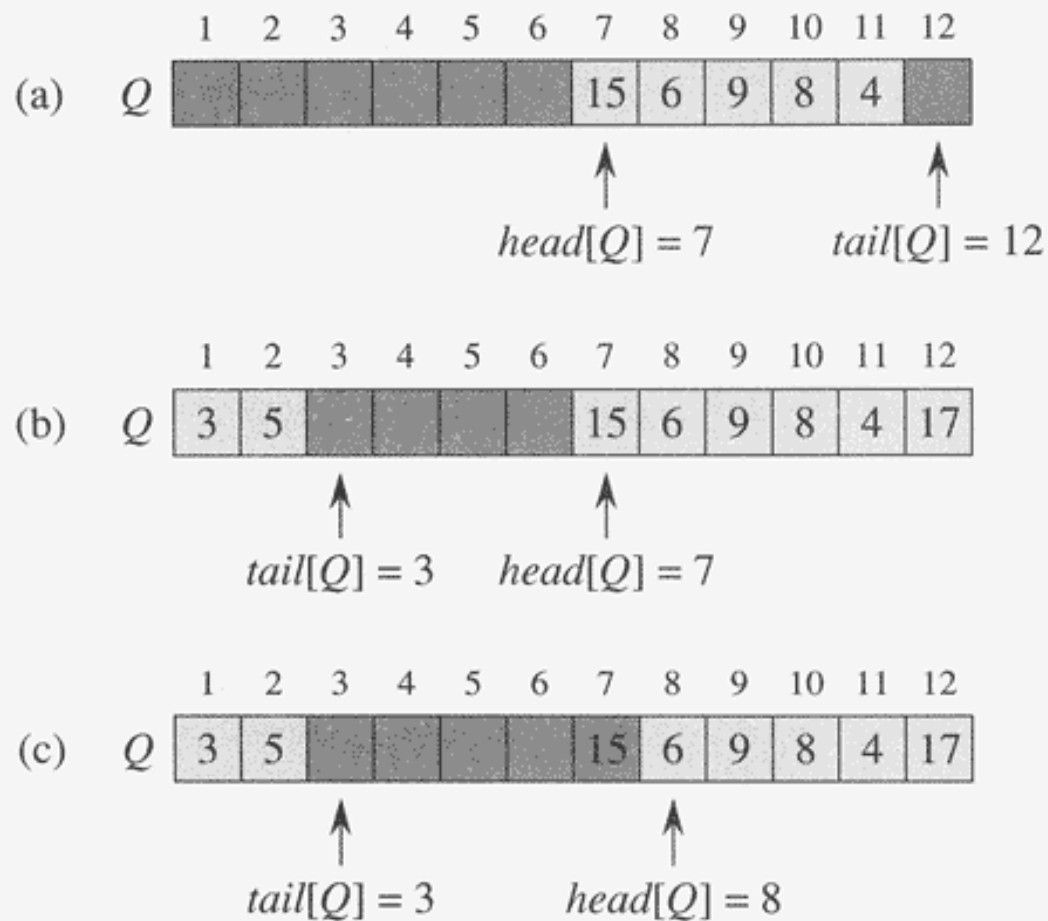


Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. (a) The queue has 5 elements, in locations $Q[7..11]$. (b) The configuration of the queue after the calls $ENQUEUE(Q, 17)$, $ENQUEUE(Q, 3)$, and $ENQUEUE(Q, 5)$. (c) The configuration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

ENQUEUE(Q, x)

1 $Q[\textit{tail}[Q]] \leftarrow x$

2 **if** $\textit{tail}[Q] = \textit{length}[Q]$

3 **then** $\textit{tail}[Q] \leftarrow 1$

4 **else** $\textit{tail}[Q] \leftarrow \textit{tail}[Q] + 1$

DEQUEUE(Q)

```
1   $x \leftarrow Q[\textit{head}[Q]]$   
2  if  $\textit{head}[Q] = \textit{length}[Q]$   
3     then  $\textit{head}[Q] \leftarrow 1$   
4     else  $\textit{head}[Q] \leftarrow \textit{head}[Q] + 1$   
5  return  $x$ 
```

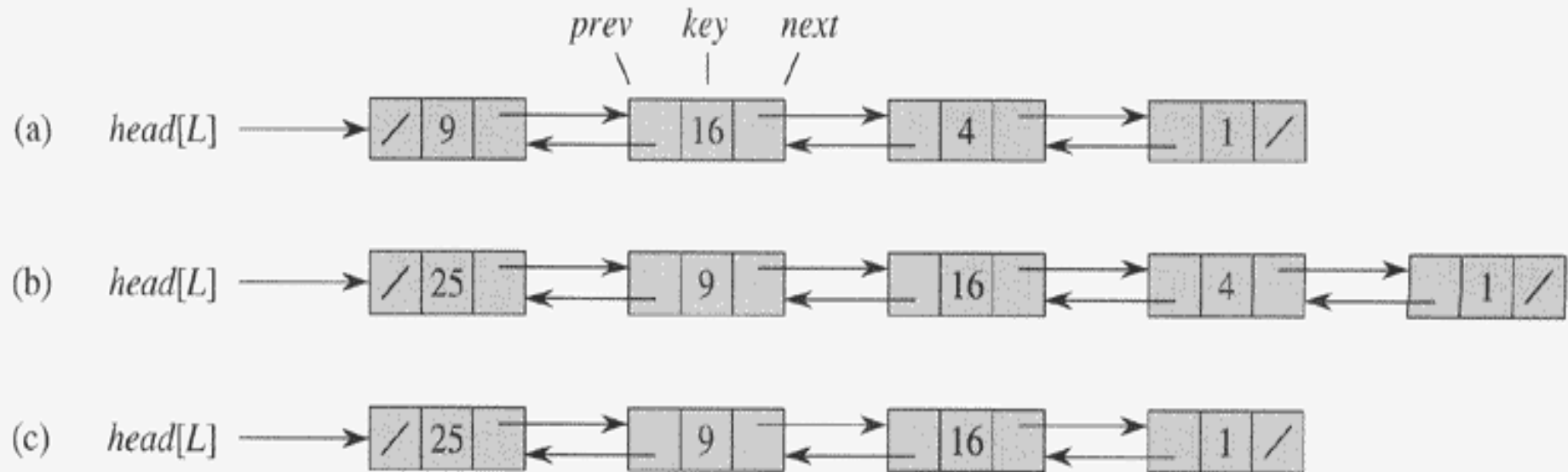


Figure 10.3 (a) A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with fields for the key and pointers (shown by arrows) to the next and previous objects. The $next$ field of the tail and the $prev$ field of the head are NIL, indicated by a diagonal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

LIST-SEARCH(L, k)

1 $x \leftarrow \textit{head}[L]$

2 **while** $x \neq \text{NIL}$ and $\textit{key}[x] \neq k$

3 **do** $x \leftarrow \textit{next}[x]$

4 **return** x

LIST-INSERT(L, x)

1 $next[x] \leftarrow head[L]$

2 **if** $head[L] \neq NIL$

3 **then** $prev[head[L]] \leftarrow x$

4 $head[L] \leftarrow x$

5 $prev[x] \leftarrow NIL$

LIST-DELETE (L, x)

```
1  if  $prev[x] \neq NIL$ 
2      then  $next[prev[x]] \leftarrow next[x]$ 
3      else  $head[L] \leftarrow next[x]$ 
4  if  $next[x] \neq NIL$ 
5      then  $prev[next[x]] \leftarrow prev[x]$ 
```

LIST-DELETE' (L, x)

- 1 $next[prev[x]] \leftarrow next[x]$
- 2 $prev[next[x]] \leftarrow prev[x]$

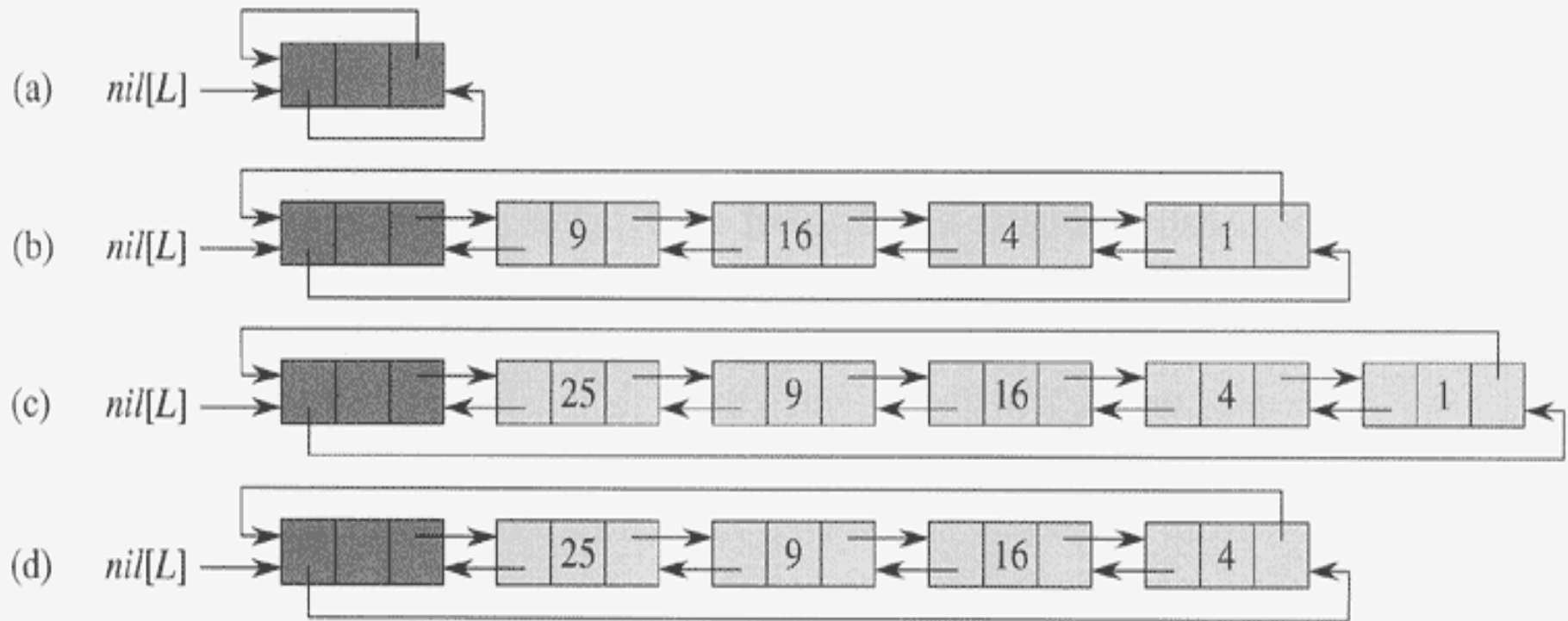


Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel $nil[L]$ appears between the head and tail. The attribute $head[L]$ is no longer needed, since we can access the head of the list by $next[nil[L]]$. (a) An empty list. (b) The linked list from Figure 10.3(a), with key 9 at the head 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.

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

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 $prev[x] \leftarrow nil[L]$

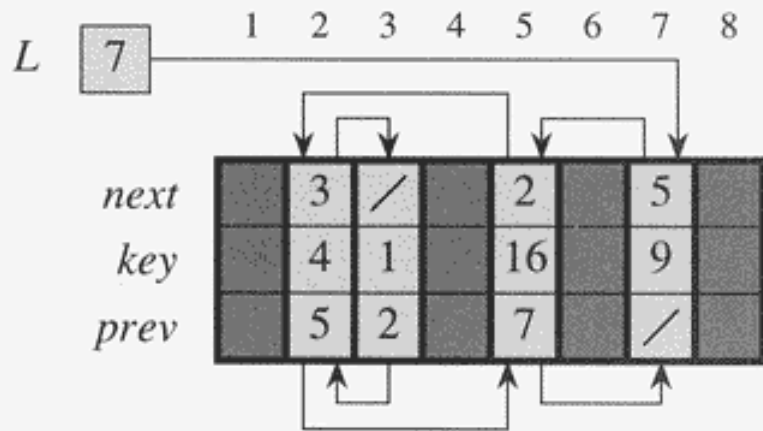


Figure 10.5 The linked list of Figure 10.3(a) represented by the arrays *key*, *next*, and *prev*. Each vertical slice of the arrays represents a single object. Stored pointers correspond to the array indices shown at the top; the arrows show how to interpret them. Lightly shaded object positions contain list elements. The variable *L* keeps the index of the head.

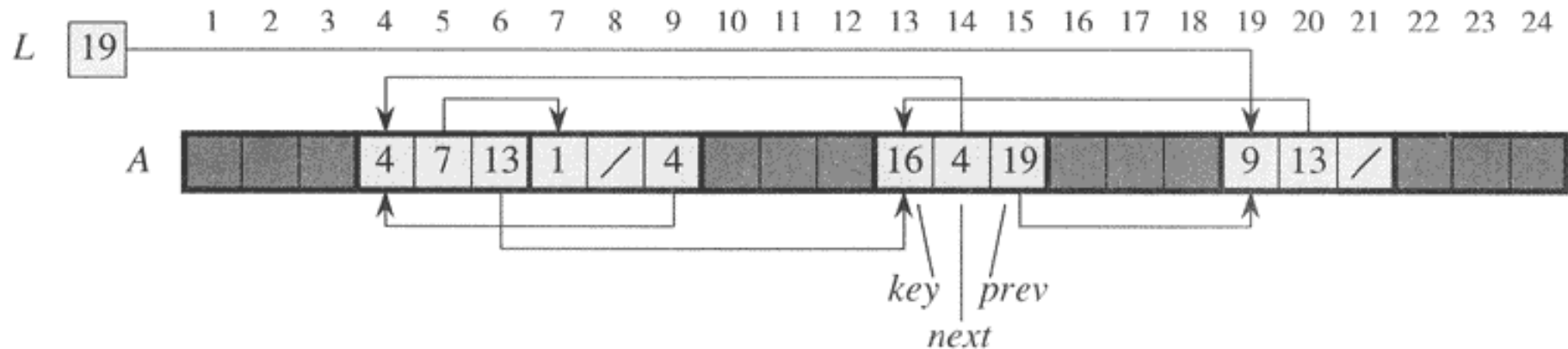


Figure 10.6 The linked list of Figures 10.3(a) and 10.5 represented in a single array A . Each list element is an object that occupies a contiguous subarray of length 3 within the array. The three fields *key*, *next*, and *prev* correspond to the offsets 0, 1, and 2, respectively. A pointer to an object is an index of the first element of the object. Objects containing list elements are lightly shaded, and arrows show the list ordering.

ALLOCATE-OBJECT()

```
1  if free = NIL
2      then error “out of space”
3      else  $x \leftarrow free$ 
4           $free \leftarrow next[x]$ 
5          return  $x$ 
```

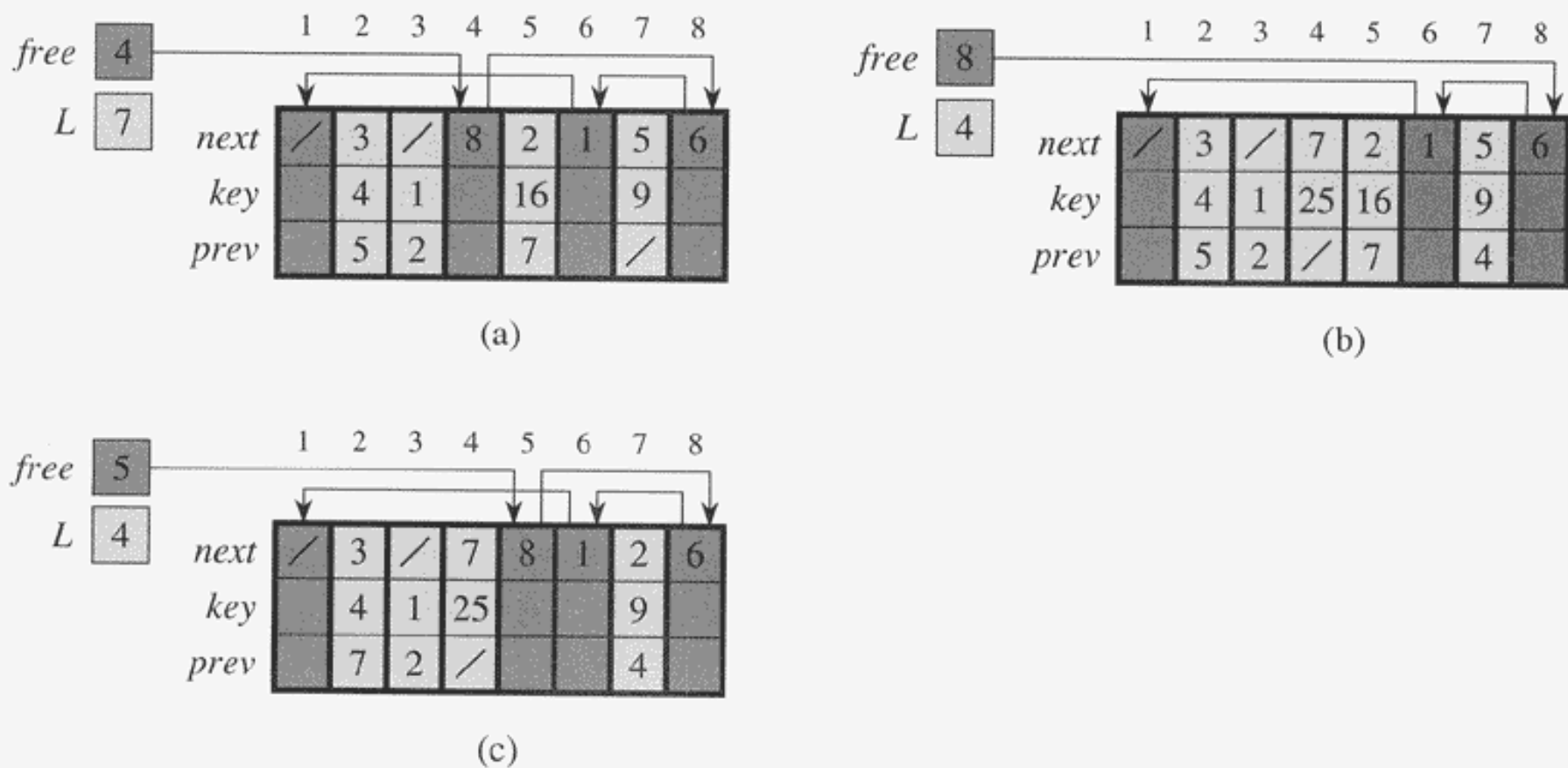


Figure 10.7 The effect of the ALLOCATE-OBJECT and FREE-OBJECT procedures. (a) The list of Figure 10.5 (lightly shaded) and a free list (heavily shaded). Arrows show the free-list structure. (b) The result of calling ALLOCATE-OBJECT() (which returns index 4), setting $key[4]$ to 25, and calling LIST-INSERT($L, 4$). The new free-list head is object 8, which had been $next[4]$ on the free list. (c) After executing LIST-DELETE($L, 5$), we call FREE-OBJECT(5). Object 5 becomes the new free-list head, with object 8 following it on the free list.

FREE-OBJECT (x)

1 $next[x] \leftarrow free$

2 $free \leftarrow x$



Figure 10.8 Two linked lists, L_1 (lightly shaded) and L_2 (heavily shaded), and a free list (darkened) intertwined.

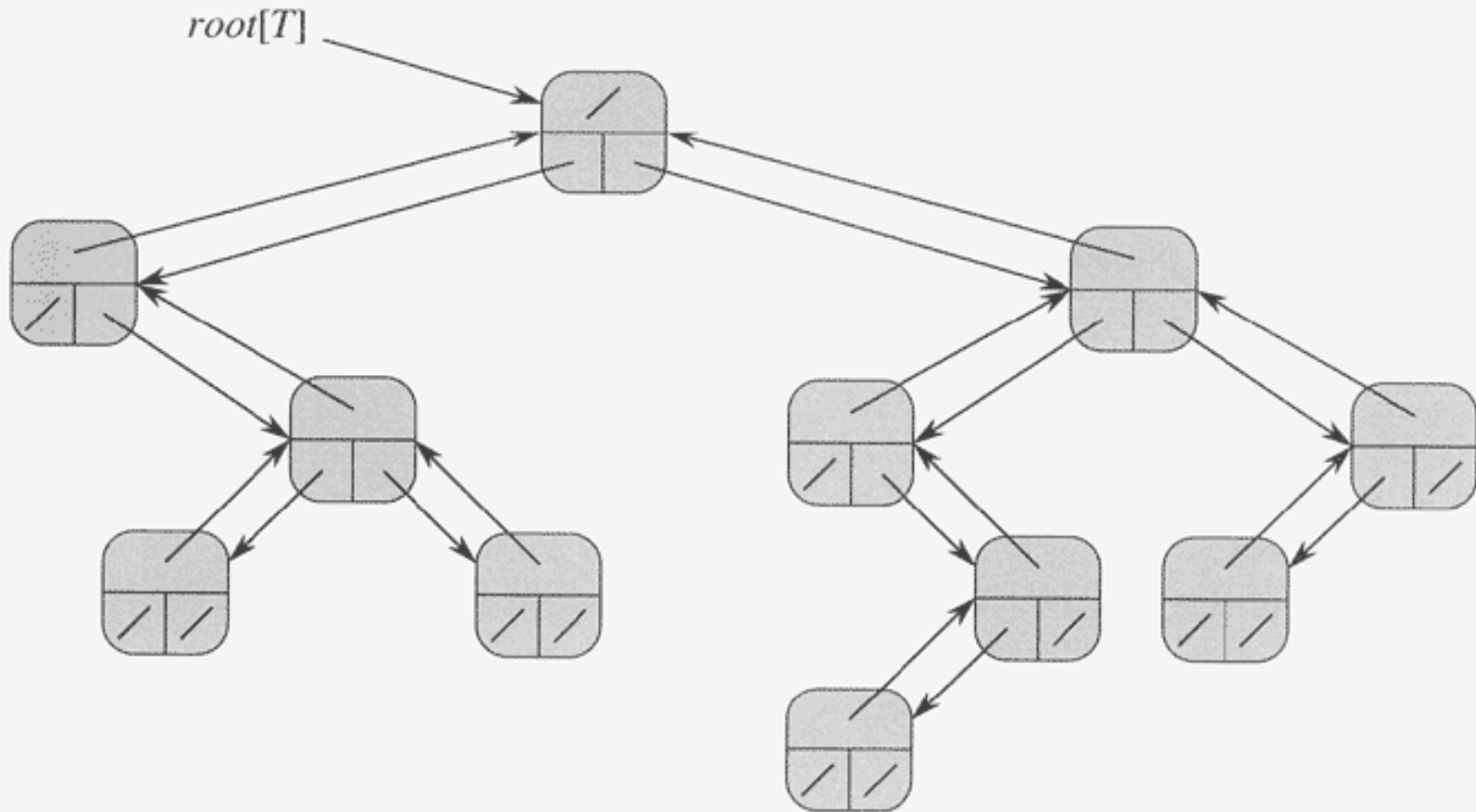


Figure 10.9 The representation of a binary tree T . Each node x has the fields $p[x]$ (top), $left[x]$ (lower left), and $right[x]$ (lower right). The *key* fields are not shown.

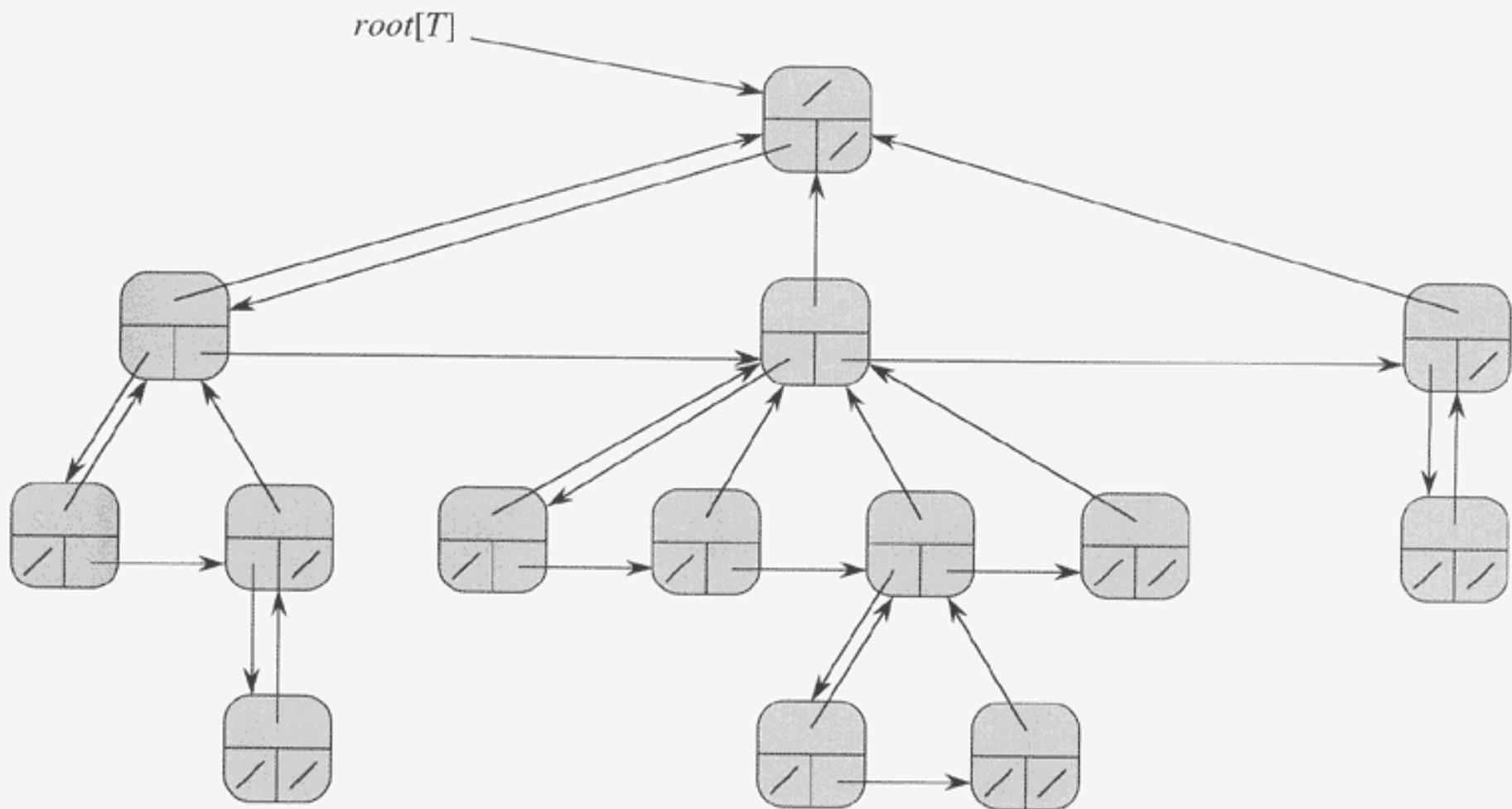


Figure 10.10 The left-child, right-sibling representation of a tree T . Each node x has fields $p[x]$ (top), $left-child[x]$ (lower left), and $right-sibling[x]$ (lower right). Keys are not shown.

COMPACT-LIST-SEARCH(L, n, k)

```
1   $i \leftarrow head[L]$ 
2  while  $i \neq NIL$  and  $key[i] < k$ 
3      do  $j \leftarrow RANDOM(1, n)$ 
4          if  $key[i] < key[j]$  and  $key[j] \leq k$ 
5              then  $i \leftarrow j$ 
6                  if  $key[i] = k$ 
7                      then return  $i$ 
8               $i \leftarrow next[i]$ 
9  if  $i = NIL$  or  $key[i] > k$ 
10     then return  $NIL$ 
11     else return  $i$ 
```

COMPACT-LIST-SEARCH' (L, n, k, t)

```
1   $i \leftarrow head[L]$ 
2  for  $q \leftarrow 1$  to  $t$ 
3      do  $j \leftarrow \text{RANDOM}(1, n)$ 
4          if  $key[i] < key[j]$  and  $key[j] \leq k$ 
5              then  $i \leftarrow j$ 
6                  if  $key[i] = k$ 
7                      then return  $i$ 
8  while  $i \neq \text{NIL}$  and  $key[i] < k$ 
9      do  $i \leftarrow next[i]$ 
10 if  $i = \text{NIL}$  or  $key[i] > k$ 
11     then return  $\text{NIL}$ 
12     else return  $i$ 
```