Chapter 10. 기본 자료구조 (Elementary Data Structures)

20150413 남윤원

목차

10-0. 기초 이론 – Dynamic과 Dictionary

- 수학의 집합은 변하지 않는 반면, 알고리즘에서 다루는 집합은 시간의 흐름에 따라 확장, 축소 또는 변경될 수 있다. – 동적집합

- 알고리즘은 집합에서 수행되는 여러 종류의 연산들을 요구한다. 이 연산들을 지원하는 동적 집합을 Dictionary라고 부른다.

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. 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 must be supported.

10-0. 기초 이론 – Elements of a dynamic set (동적 집합의 원소)

- 키와 부속 데이터 (Key and satellite data)

In a typical implementation of a dynamic set, each element is represented by an object whose attributes 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 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.

10-0. 기초 이론 – Operations on dynamic sets (동적 집합의 연산)

- 질의(query) : 집합에 대한 정보를 알아보려는 연산
- 변경 연산 : 집합을 바꾸는 연산

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.

10-0. 기초 이론 – Operations on dynamic sets (동적 집합의 연산)

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 augments the set S with the element pointed to by x. 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)
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.

10-0. 기초 이론

- 점화식에서의 기술적 사항 – ex. MergeSort의 Worst Case

In practice, we neglect certain technical details when we state and solve recurrences. For example, if we call MERGE-SORT on n elements when n is odd, we end up with subproblems of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Neither size is actually $n/2$, because $n/2$ is not an integer when n is odd. Technically, the recurrence describing the worst-case running time of MERGE-SORT is really

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1 . \end{cases} \tag{4.3}$$

Boundary conditions represent another class of details that we typically ignore. Since the running time of an algorithm on a constant-sized input is a constant, the recurrences that arise from the running times of algorithms generally have $T(n) = \Theta(1)$ for sufficiently small n. Consequently, for convenience, we shall generally omit statements of the boundary conditions of recurrences and assume that $T(n)$ is constant for small n. For example, we normally state recurrence (4.1) as

$$T(n) = 2T(n/2) + \Theta(n) , \tag{4.4}$$

without explicitly giving values for small n. The reason is that although changing the value of $T(1)$ changes the exact solution to the recurrence, the solution typically doesn't change by more than a constant factor, and so the order of growth is unchanged.

When we state and solve recurrences, we often omit floors, ceilings, and boundary conditions. We forge ahead without these details and later determine whether

10-1. 스택과 큐 (Stacks and Queues)

- 스택(Stack) : 가장 최근에 삽입된 원소가 삭제된다.
  : 후입선출, LIFO

- 큐(Queue) : 가장 오랜 시간 동안 존재한 원소가 삭제된다.
  : 선입선출, FIFO

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.

10-1. 스택과 큐 (Stacks and Queues)

- 스택(Stack)에서 Insert 연산을 PUSH, 인자를 갖지 않는 DELETE 연산을 POP이라 한다.

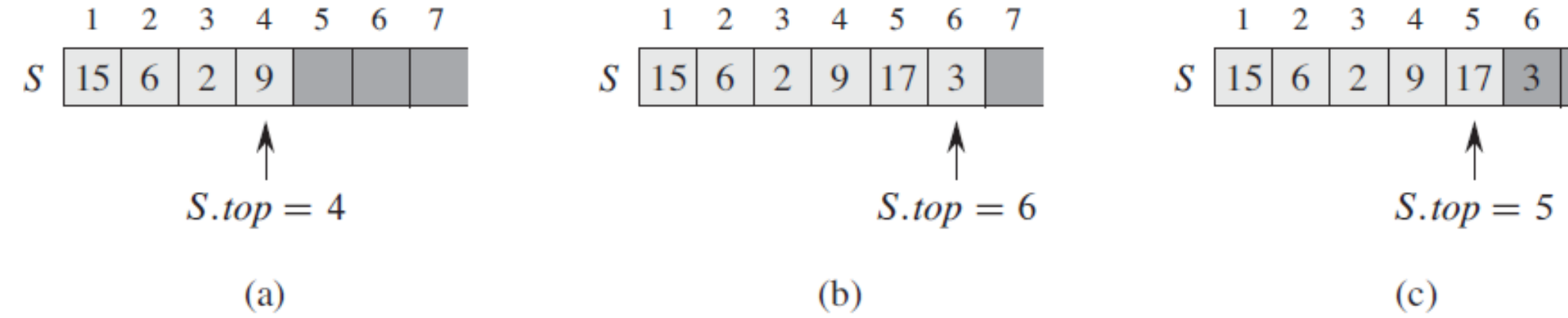

Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH(S, 17) and PUSH(S, 3). **(c)** Stack S after the call POP(S) has returned the element 3, which is the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

10-1. 스택과 큐 (Stacks and Queues)

- 다음 그림과 같이 배열 S[1..n]으로 최대 n개의 원소를 가지는 스택을 구현할 수 있다.

- 배열은 가장 최근에 삽입된 원소를 가리키는 S.top을 속성값으로 가진다.

- 스택은 원소 S[1..S.top]으로 구성되어 있으며, S[1]은 맨 밑에 있는 원소를 나타내고, S[S.top]은 맨 위에 있는 원소를 나타낸다.

10-1. 스택과 큐 (Stacks and Queues)

- S.top = 0일 때, 스택은 원소를 포함하지 않고 비었다(empty)고 한다.

- STACK-EMPTY 연산을 통해 스택이 비었는지 검사할 수 있다.

- 빈 스택에서 원소를 추출하려는 경우 underflow라 하며, 오류로 간주된다.

- S.top이 원소의 개수 n을 초과하면 overflow라 한다.

10-1. 스택과 큐 (Stacks and Queues)

- 스택의 각 연산

```
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]
```

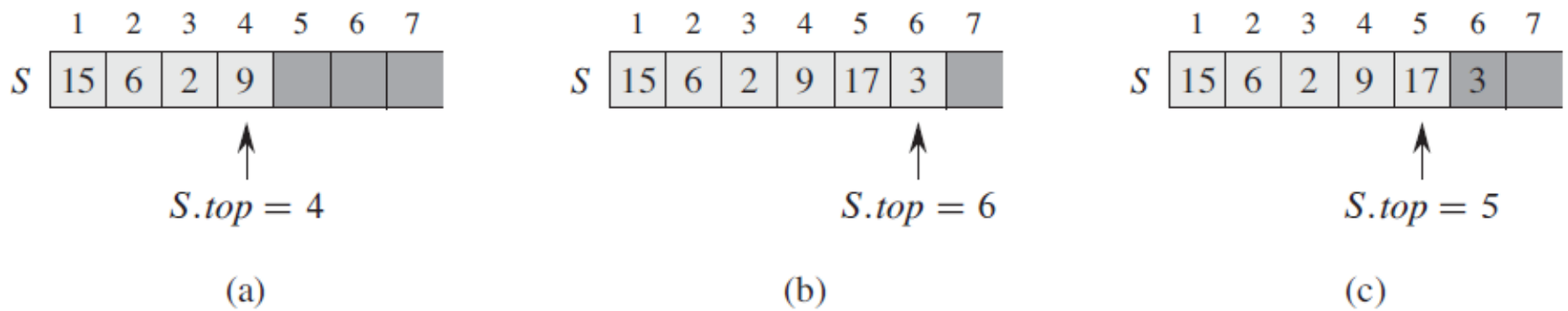

Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH$(S, 17)$ and PUSH$(S, 3)$. **(c)** Stack S after the call POP(S) has returned the element 3, which is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

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

10-1. 스택과 큐 (Stacks and Queues)

- 큐(Queue)에서는 INSERT 연산을 ENQUEUE, DELETE 연산을 DEQUEUE라고 한다.

- 스택의 POP 연산과 동일하게 연산 DEQUEUE도 인자를 갖지 않는다.

- 큐는 머리(head)와 꼬리(tail)라는 인자를 가진다.

- 큐에서의 삭제는 가장 오래 기다린, 대기열의 맨 앞에 있는 손님처럼 항상 큐의 머리 위치에 있는 원소를 삭제한다.

10-1. 스택과 큐 (Stacks and Queues)

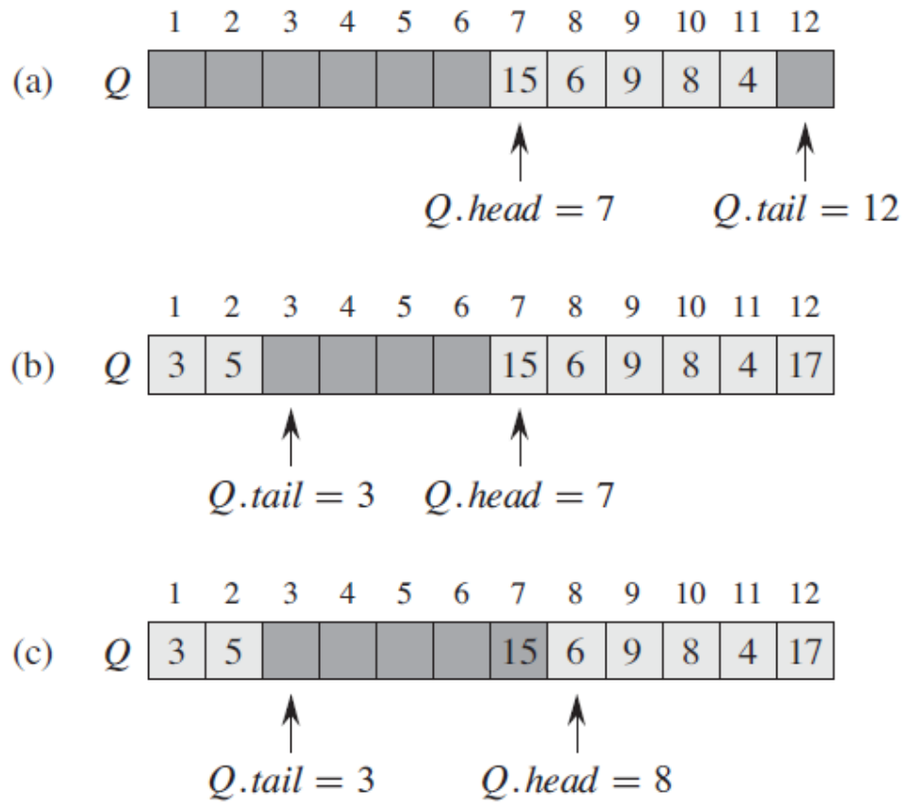

Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The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

10-1. 스택과 큐 (Stacks and Queues)

- 배열 Q[1..n]을 이용해 원소가 최대 n-1인 큐를 구현할 수 있다.
- Head를 가리키는 index 또는 pointer를 속성값 Q.head에, 새로운 원소가 삽입될 위치는 속성값 Q.tail에 저장한다.
- 큐의 원소들은 Q.head, Q.head+1, …, Q.tail -1 위치에 존재한다.
- 위치 1은 위치 n 다음에 존재하는 고리 모양의 순환 순서를 가진다.
- Q.head = Q.tail인 경우 큐는 비었다.
  초기에 큐는 Q.head = Q.tail = 1로 시작한다.
- 빈 큐에서 원소를 삭제하려고 하면 Underflow가 발생
  Q.head = Q.taili + 1 또는 Q.head = 1 and Q.tail =
  Q.length는 큐가 가득찬 상태이고, 이때 원소 삽입 시도시 overflow

10-1. 스택과 큐 (Stacks and Queues)

- 연습문제

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. 스택과 큐 (Stacks and Queues)

- 연습문제

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 연결 리스트 (Linked Lists)

- 연결 리스트(Linked List)는 객체가 선형적 순서를 가지도록 배치된 자료구조이다.

- 인덱스에 의해 선형적 순서가 결정되는 배열과는 달리, 연결 리스트에서는 각 객체에 있는 포인터에 의해 순서가 결정된다.

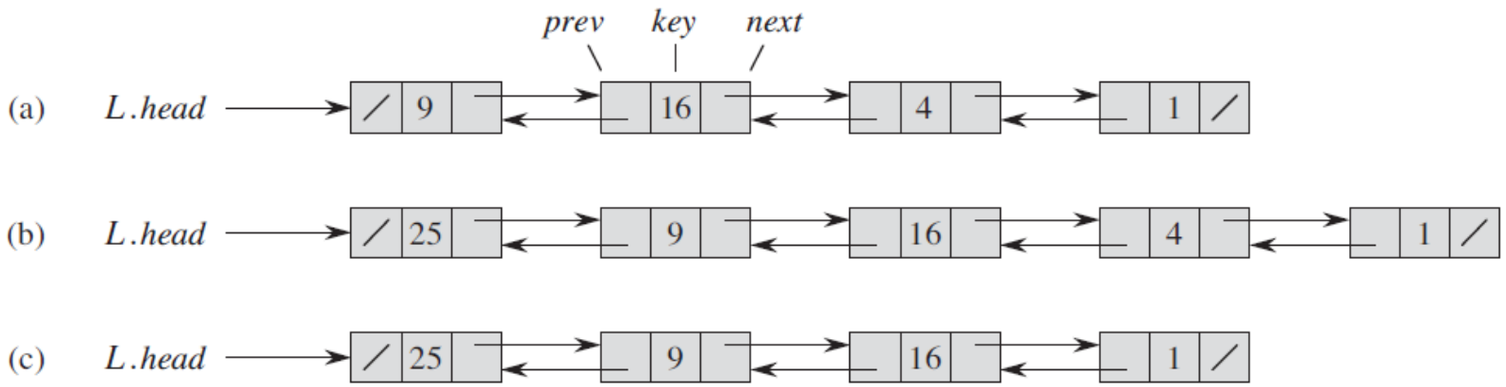

Figure 10.3 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, 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.

10.2 연결 리스트 (Linked Lists)

- 양방향 연결 리스트(doubly linked list) L의 각 원소는 key 속성값과 두 개의 포인터인 prev와 next를 속성값으로 가지는 객체다. (이 객체는 부가 데이터를 가질 수도 있다.)

- X.next는 다음 원소를, X.prev는 직전 원소를 가리킨다.

- X.prev가 NIL이라면 원소 x는 바로 이전 원소가 없으므로, 이 리스트의 첫 번째 원소 또는 머리(head)라 한다.

- 또한 x.next = NIL인 경우, 원소 x는 바로 다음 원소가 존재하지 않으므로 이 리스트의 마지막 원소 또는 꼬리(tail)라 한다. 속성값 L.head는 리스트의 첫 번째 원소를 가리킨다. L.head = NIL인 경우는 리스트가 비었음을 의미한다.

10.2 연결 리스트 (Linked Lists)

- 리스트는 형태가 다양하다.
- 1. 단순 연결 리스트 (singly linked list)
- 2. 양방향 연결 리스트 (doubly linked list)
- 3. 환형 연결 리스트 (circular linked list)
- 리스트가 정렬되어 있다. (sorted) -> 리스트의 순서가 각 원소의 키 순서대로 저장되어 있다.
- 정렬되지 않은(unsorted) 리스트의 경우, 원소가 아무 위치에나 존재할 수 있다.

10.2 연결 리스트 (Linked Lists)

- 연결 리스트에서의 검색

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 ≠ NIL and x.key ≠ 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.

10.2 연결 리스트 (Linked Lists)

- 연결 리스트에서의 삽입

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 ≠ NIL
3        L.head.prev = x
4  L.head = x
5  x.prev = 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 $O(1)$.

10.2 연결 리스트 (Linked Lists)

- 연결 리스트에서의 삭제

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

10.2 연결 리스트 (Linked Lists)

- 경계 원소

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

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 $L.nil$.

10.2 연결 리스트 (Linked Lists)

- 경계 원소

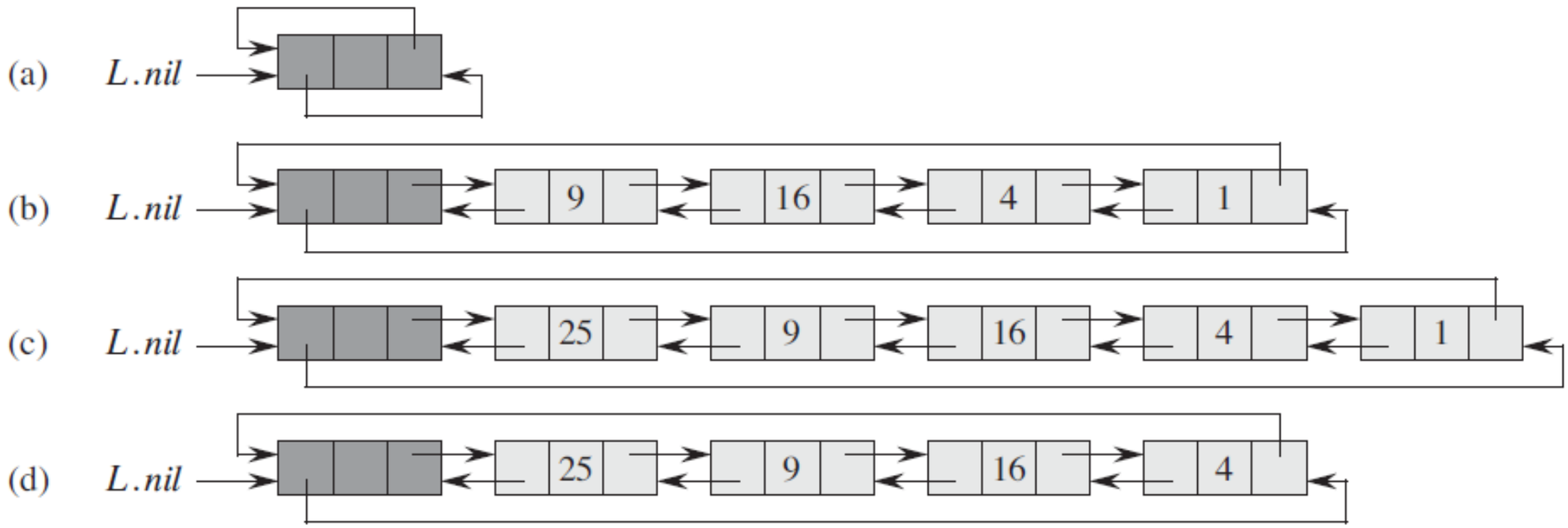

Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel $L.nil$ appears between the head and tail. The attribute $L.head$ is no longer needed, since we can access the head of the list by $L.nil.next$. **(a)** An empty list. **(b)** The linked list from Figure 10.3(a), with key 9 at the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

10.2 연결 리스트 (Linked Lists)

- 경계 원소

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 ≠ L.nil and x.key ≠ 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.

10.2 연결 리스트 (Linked Lists)

- 경계 원소는 신중하게 사용해야 한다. 크기가 작은 리스트가 많은 경우, 경계 원소를 사용하기 위한 추가 저장 공간의 사용이 상당한 메모리 낭비를 초래할 수 있다. 이 책에서는, 코드를 단순화하는 경우에만 경계 원소를 사용하였다.

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.

10.2 연결 리스트 (Linked Lists)

- 연습문제

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 연결 리스트 (Linked Lists)

- 연습문제

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 $x.np = x.next$ 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 포인터와 객체 구현하기 (Implementing pointers and objects)

- 명시적인 포인터 데이터 타입 없이 연결된 자료구조를 구현하는 방법 : 배열과 배열의 인덱스를 이용해 객체와 포인터를 만든다.

- 1. 객체의 다중 배열 표현

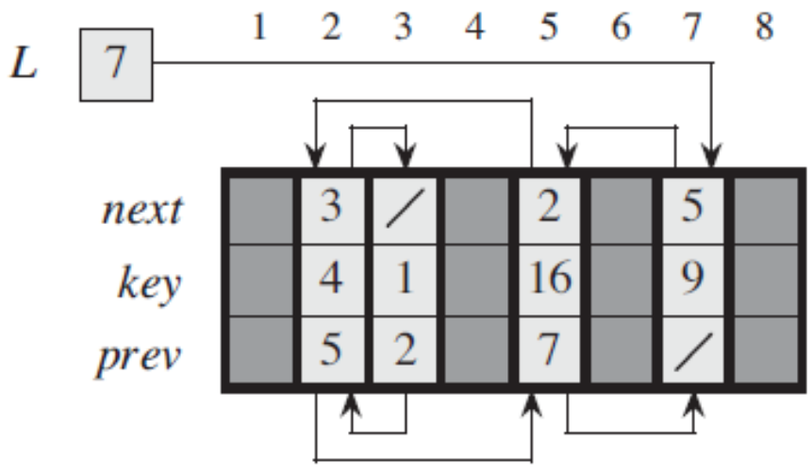

Figure 10.5 The linked list of Figure 10.3(a) represented by the arrays *key*, *next*, and *prev*. Each vertical slice of the arrays represents a single object. Stored pointers correspond to the array indices shown at the top; the arrows show how to interpret them. Lightly shaded object positions contain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

10.3 포인터와 객체 구현하기 (Implementing pointers and objects)

- 명시적인 포인터 데이터 타입 없이 연결된 자료구조를 구현하는 방법 : 배열과 배열의 인덱스를 이용해 객체와 포인터를 만든다.

- 2. 객체의 단일 배열 표현

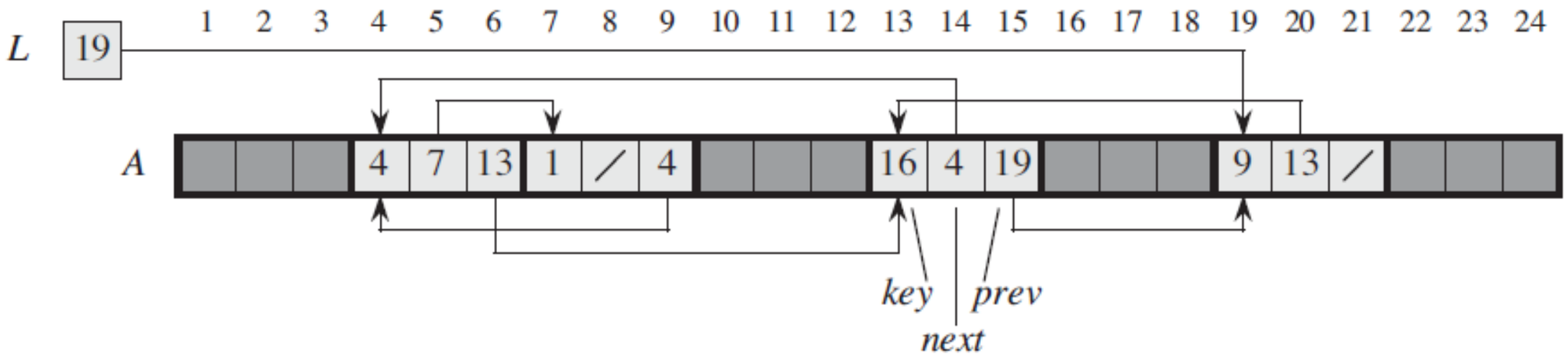

Figure 10.6 The linked list of Figures 10.3(a) and 10.5 represented in a single array A. Each list element is an object that occupies a contiguous subarray of length 3 within the array. The three attributes *key*, *next*, and *prev* correspond to the offsets 0, 1, and 2, respectively, within each object.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

10.3 포인터와 객체 구현하기 (Implementing pointers and objects)

- 객체의 할당과 해제 (garbage collector가 관리하는 OS있음)

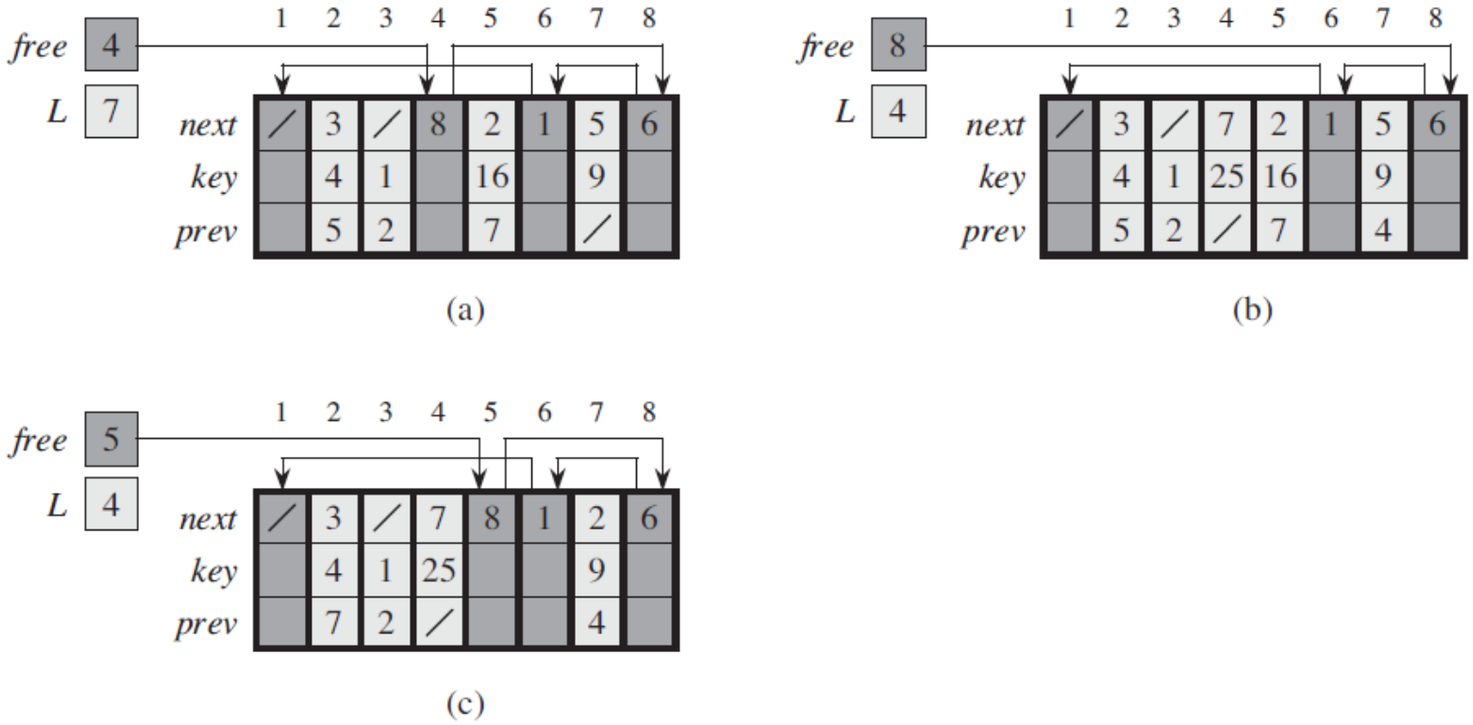

Figure 10.7 The effect of the ALLOCATE-OBJECT and FREE-OBJECT procedures. **(a)** The list of Figure 10.5 (lightly shaded) and a free list (heavily shaded). Arrows show the free-list structure. **(b)** The result of calling ALLOCATE-OBJECT() (which returns index 4), setting *key*[4] to 25, and calling 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.

- 단순 연결 리스트로 객체를 관리하면 free list라 함.

10.3 포인터와 객체 구현하기 (Implementing pointers and objects)

- 객체의 할당과 해제 (garbage collector가 관리하는 OS있음)

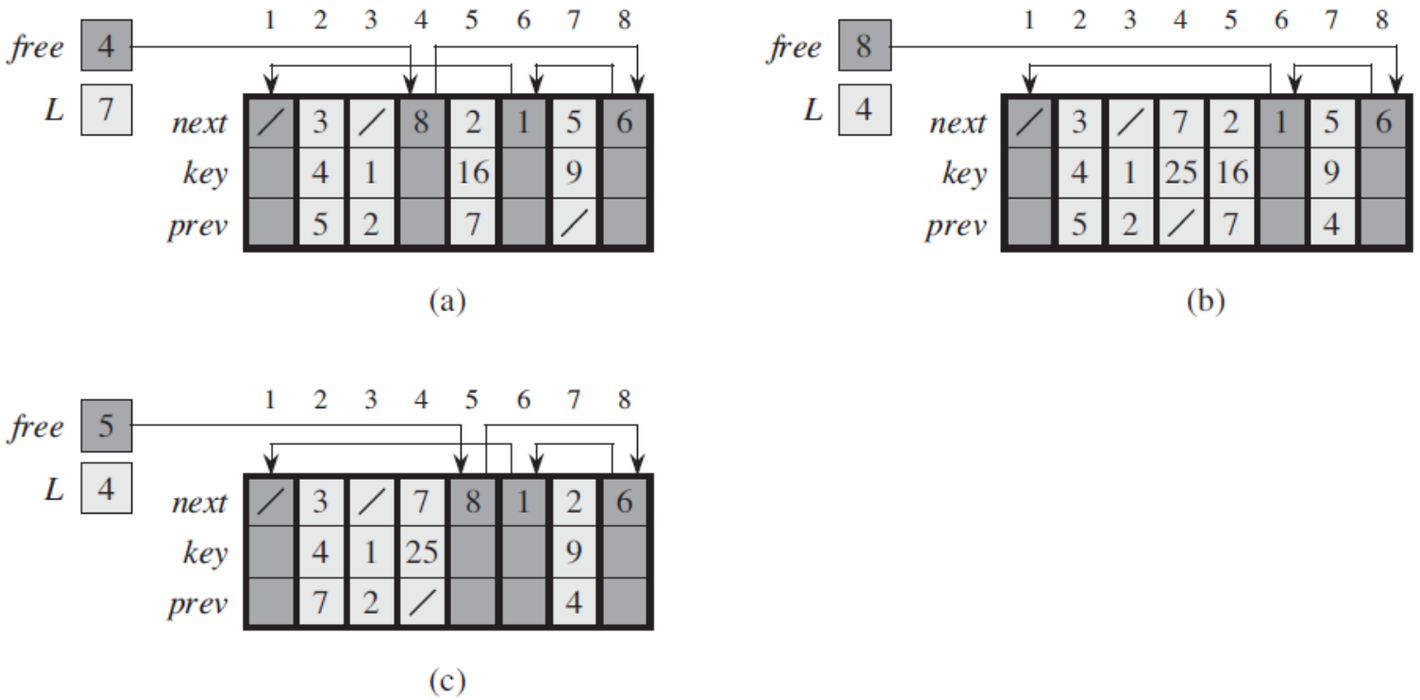

Figure 10.7 The effect of the ALLOCATE-OBJECT and FREE-OBJECT procedures. **(a)** The list of Figure 10.5 (lightly shaded) and a free list (heavily shaded). Arrows show the free-list structure. **(b)** The result of calling ALLOCATE-OBJECT() (which returns index 4), setting *key*[4] to 25, and calling 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.

- 단순 연결 리스트로 객체를 관리하면 free list라 함. Free list는 스택처럼 동작함

10.3 포인터와 객체 구현하기 (Implementing pointers and objects)

- Free list – 아래 두 연산 모두 O(1) 안에 동작함

```
ALLOCATE-OBJECT()
1  if free == NIL
2        error "out of space"
3  else x = free
4       free = x.next
5       return x
```

```
FREE-OBJECT(x)
1  x.next = free
2  free = x
```

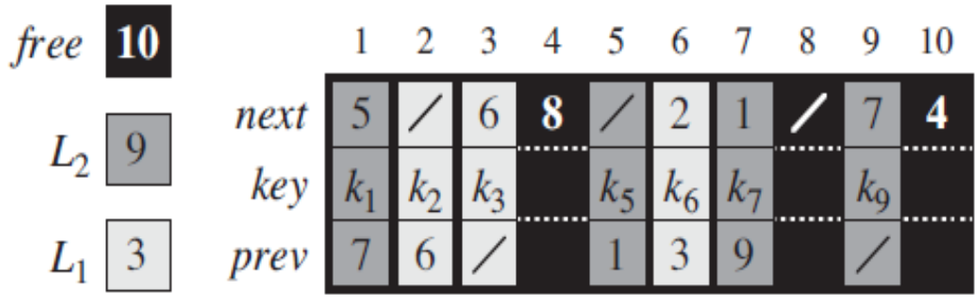

Figure 10.8 Two linked lists, L_1 (lightly shaded) and L_2 (heavily shaded), and a free list (darkened) intertwined.

10.3 포인터와 객체 구현하기 (Implementing pointers and objects)

- 연습 문제 생략

10.3 포인터와 객체 구현하기 (Implementing pointers and objects)

- 연습 문제 생략

10.4 루트 있는 트리 표현하기 (Representing rooted trees)

- 이진 트리와 제한 없는 가지를 가지는 루트 있는 트리

Binary trees

Figure 10.9 shows how we 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

We can 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: we 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, since we do not know how many attributes (arrays in the multiple-array representation) to allocate in advance. Moreover, even if the number of children k 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 10.10. 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 $x.right\text{-}sibling = \text{NIL}$.

10.4 루트 있는 트리 표현하기 (Representing rooted trees)

- 이진 트리와 제한 없는 가지를 가지는 루트 있는 트리

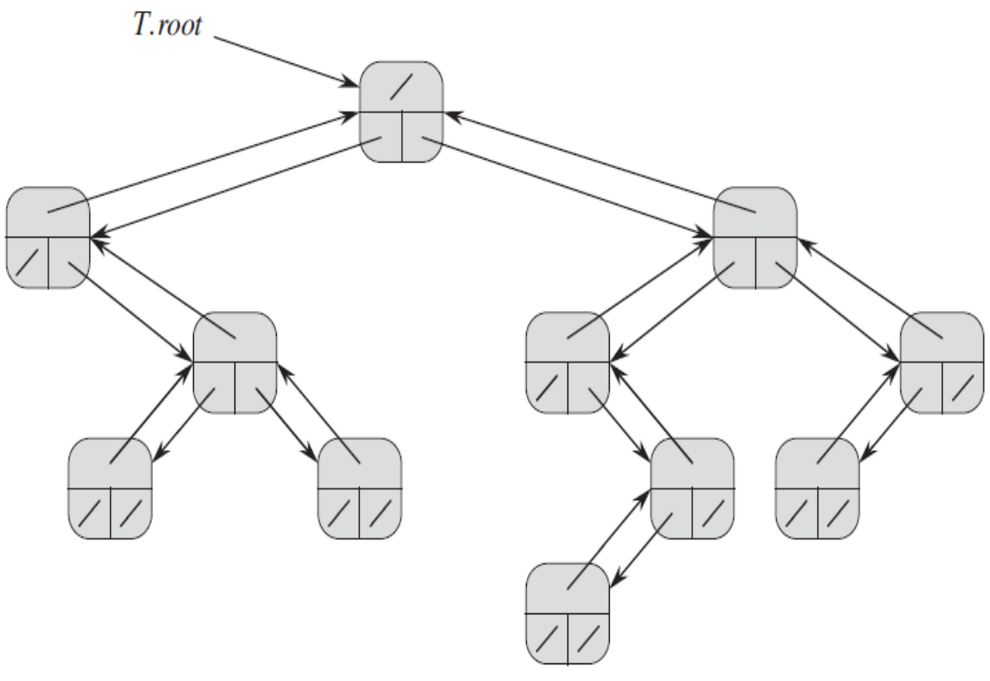

Figure 10.9 The representation of a binary tree T. Each node x has the attributes $x.p$ (top), $x.left$ (lower left), and $x.right$ (lower right). The *key* attributes are not shown.

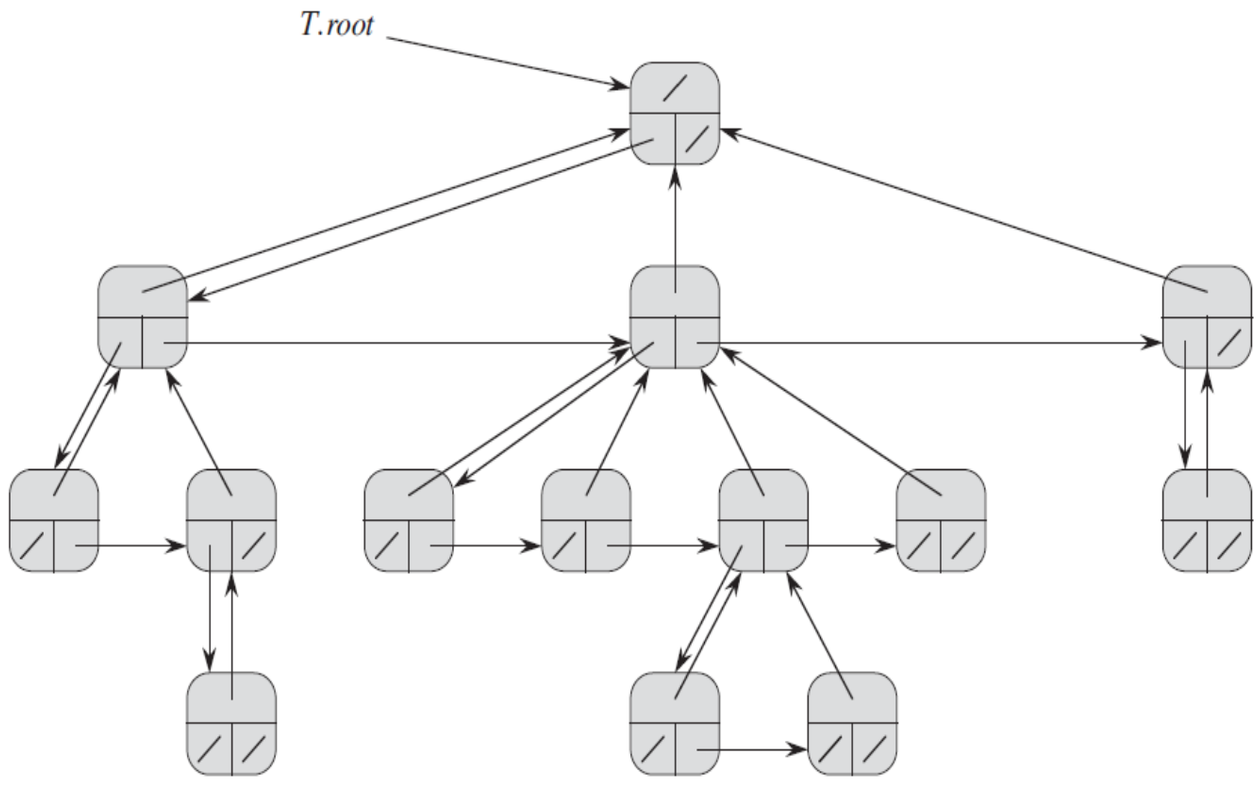

Figure 10.10 The left-child, right-sibling representation of a tree T. Each node x has attributes $x.p$ (top), $x.left\text{-}child$ (lower left), and $x.right\text{-}sibling$ (lower right). The *key* attributes are not shown.

10.4 루트 있는 트리 표현하기 (Representing rooted trees)

- 이진 트리와 제한 없는 가지를 가지는 루트 있는 트리

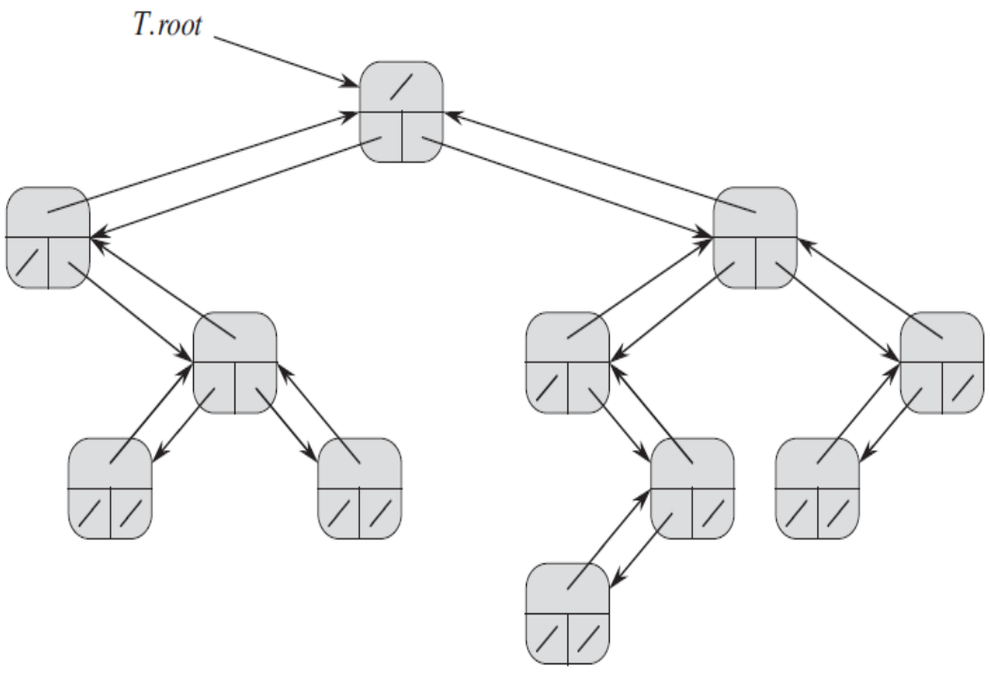

Figure 10.9 The representation of a binary tree T. Each node x has the attributes $x.p$ (top), $x.left$ (lower left), and $x.right$ (lower right). The *key* attributes are not shown.

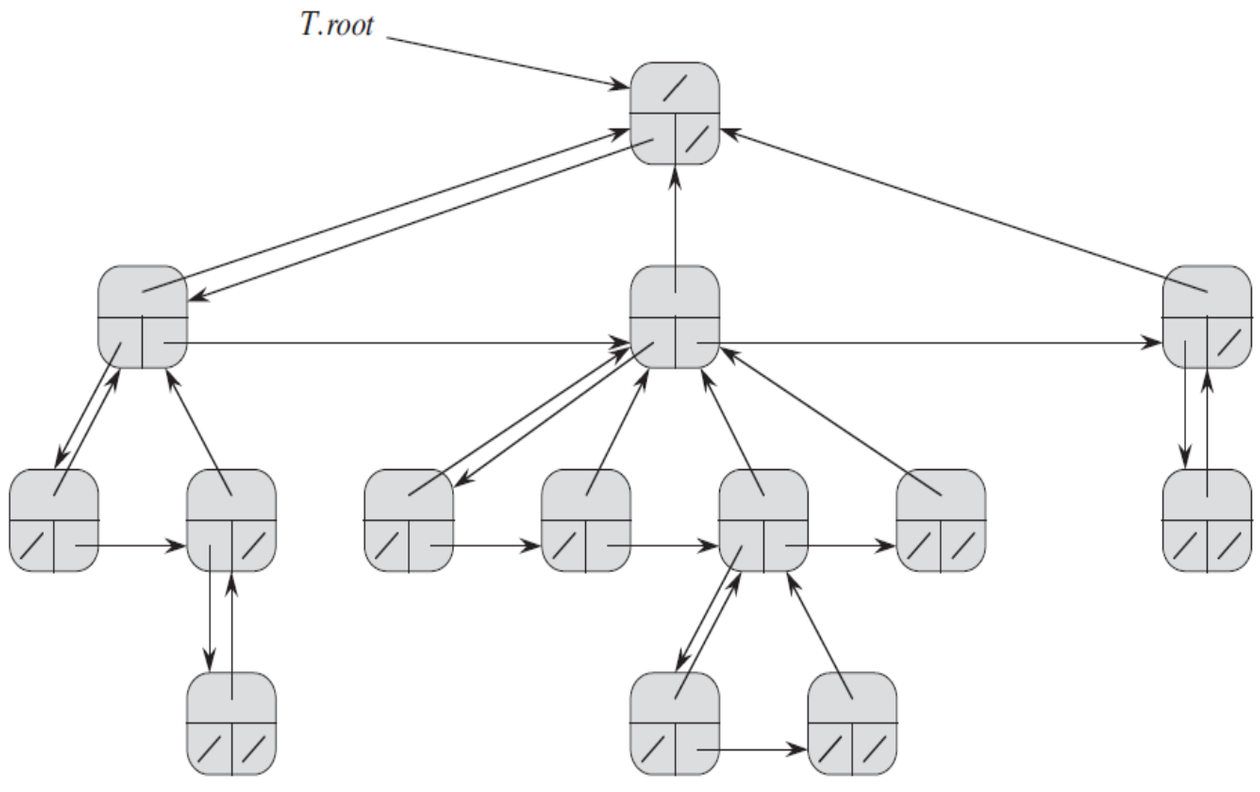

Figure 10.10 The left-child, right-sibling representation of a tree T. Each node x has attributes $x.p$ (top), $x.left\text{-}child$ (lower left), and $x.right\text{-}sibling$ (lower right). The *key* attributes are not shown.

10.4 루트 있는 트리 표현하기 (Representing rooted trees)

- 연습문제

Exercises

10.4-1

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

index	*key*	*left*	*right*
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

10.4-2

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

10.4-3

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.