Fibonacci Heap: *Functions*

MAKE-HEAP() creates and returns a new heap containing no elements.

INSERT(H, x) inserts element x, whose *key* has already been filled in, into heap H.

MINIMUM(H) returns a pointer to the element in heap H whose key is minimum.

EXTRACT-MIN(H) deletes the element from heap H whose key is minimum, returning a pointer to the element.

UNION(H_1, H_2) creates and returns a new heap that contains all the elements of heaps H_1 and H_2. Heaps H_1 and H_2 are "destroyed" by this operation.

In addition to the mergeable-heap operations above, Fibonacci heaps also support the following two operations:

DECREASE-KEY(H, x, k) assigns to element x within heap H the new key value k, which we assume to be no greater than its current key value.[1]

DELETE(H, x) deletes element x from heap H.

Fibonacci Heap: *Complexity*

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

Fibonacci Heap: *Structure*

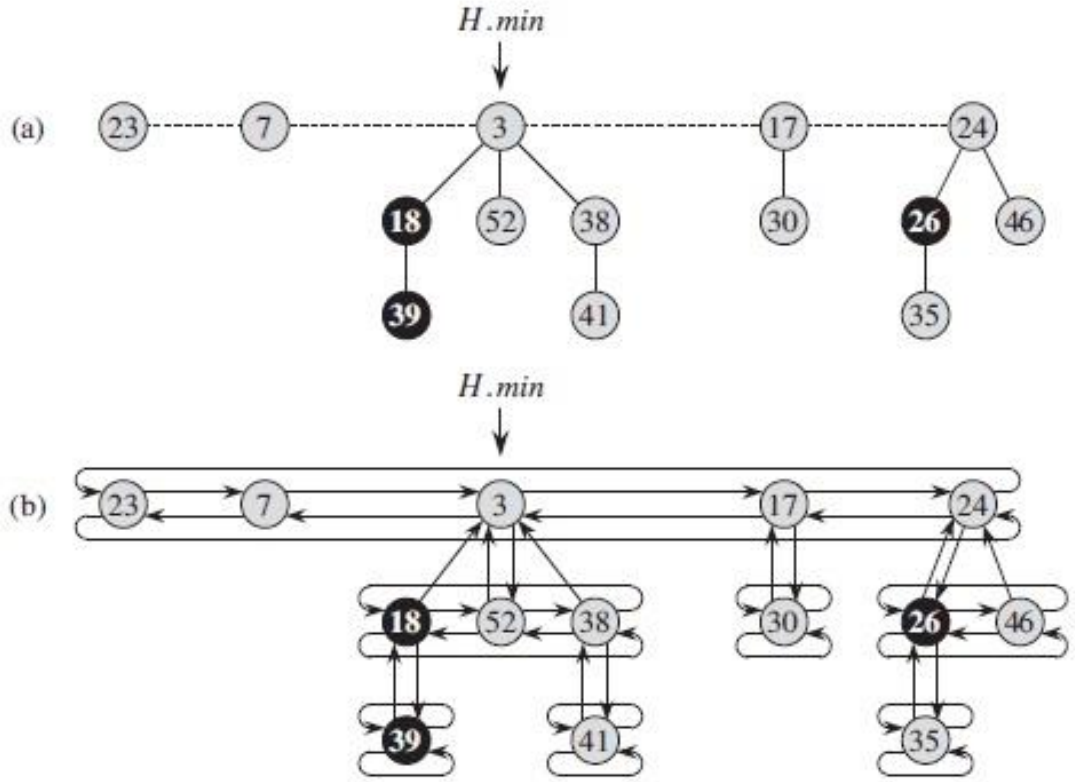


Figure 19.2 (a) A Fibonacci heap consisting of five min-heap-ordered trees and 14 nodes. The dashed line indicates the root list. The minimum node of the heap is the node containing the key 3. Black nodes are marked. The potential of this particular Fibonacci heap is $5 + 2 \cdot 3 = 11$. (b) A more complete representation showing pointers p (up arrows), *child* (down arrows), and *left* and *right* (sideways arrows). The remaining figures in this chapter omit these details, since all the information shown here can be determined from what appears in part (a).

```
FIB-HEAP-INSERT(H, x)
 1  x.degree = 0
 2  x.p = NIL
 3  x.child = NIL
 4  x.mark = FALSE
 5  if H.min == NIL
 6       create a root list for H containing just x
 7       H.min = x
 8  else insert x into H's root list
 9       if x.key < H.min.key
10            H.min = x
11  H.n = H.n + 1
```

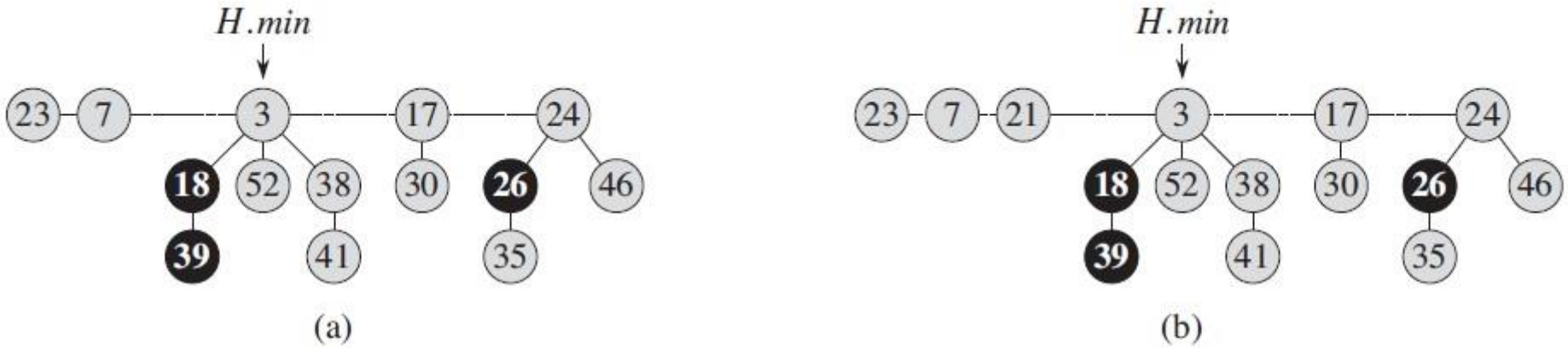


Figure 19.3 Inserting a node into a Fibonacci heap. **(a)** A Fibonacci heap H. **(b)** Fibonacci heap H after inserting the node with key 21. The node becomes its own min-heap-ordered tree and is then added to the root list, becoming the left sibling of the root.

```
FIB-HEAP-UNION(H_1, H_2)
1  H = MAKE-FIB-HEAP()
2  H.min = H_1.min
3  concatenate the root list of H_2 with the root list of H
4  if (H_1.min == NIL) or (H_2.min ≠ NIL and H_2.min.key < H_1.min.key)
5       H.min = H_2.min
6  H.n = H_1.n + H_2.n
7  return H
```

```
FIB-HEAP-EXTRACT-MIN(H)
 1  z = H.min
 2  if z ≠ NIL
 3       for each child x of z
 4            add x to the root list of H
 5            x.p = NIL
 6       remove z from the root list of H
 7       if z == z.right
 8            H.min = NIL
 9       else H.min = z.right
10            CONSOLIDATE(H)
11       H.n = H.n − 1
12  return z
```

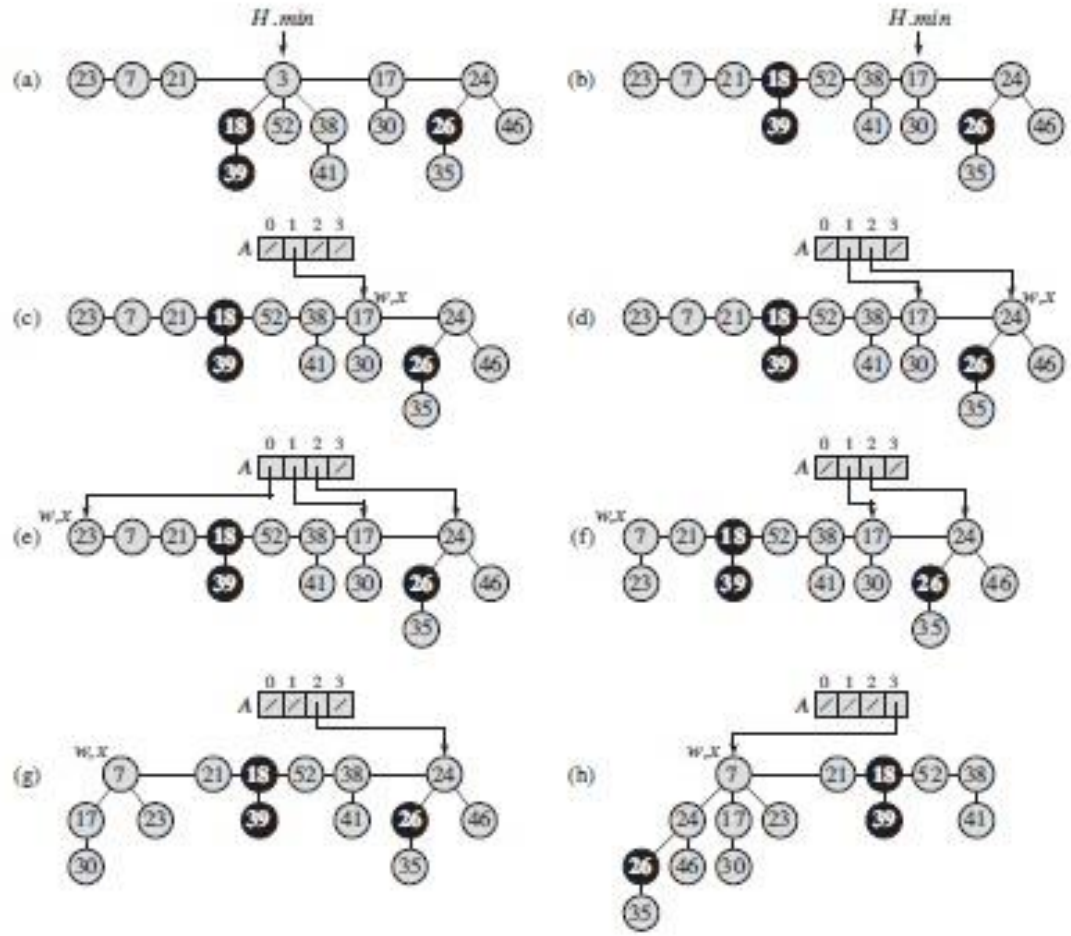


Figure 19.4 The action of FIB-HEAP-EXTRACT-MIN. (a) A Fibonacci heap H. (b) The situation after removing the minimum node z from the root list and adding its children to the root list. (c)–(e) The array A and the trees after each of the first three iterations of the **for** loop of lines 4–14 of the procedure CONSOLIDATE. The procedure processes the root list by starting at the node pointed to by $H.min$ and following *right* pointers. Each part shows the values of w and x at the end of an iteration. (f)–(h) The next iteration of the **for** loop, with the values of w and x shown at the end of each iteration of the **while** loop of lines 7–13. Part (f) shows the situation after the first time through the **while** loop. The node with key 23 has been linked to the node with key 7, which x now points to. In part (g), the node with key 17 has been linked to the node with key 7, which x still points to. In part (h), the node with key 24 has been linked to the node with key 7. Since no node was previously pointed to by $A[3]$, at the end of the **for** loop iteration, $A[3]$ is set to point to the root of the resulting tree.

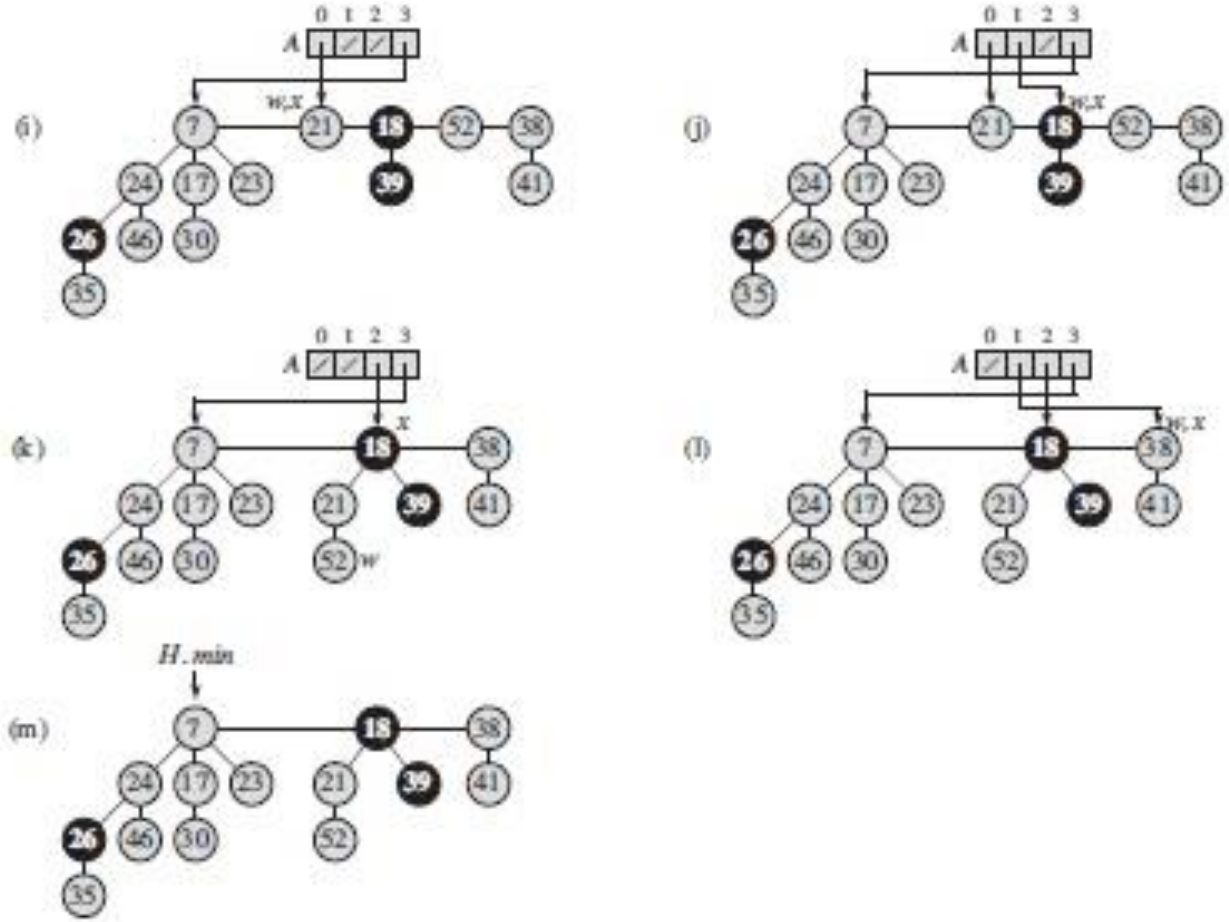


Figure 19.4, continued (i)–(l) The situation after each of the next four iterations of the **for** loop. (m) Fibonacci heap H after reconstructing the root list from the array A and determining the new $H.min$ pointer.

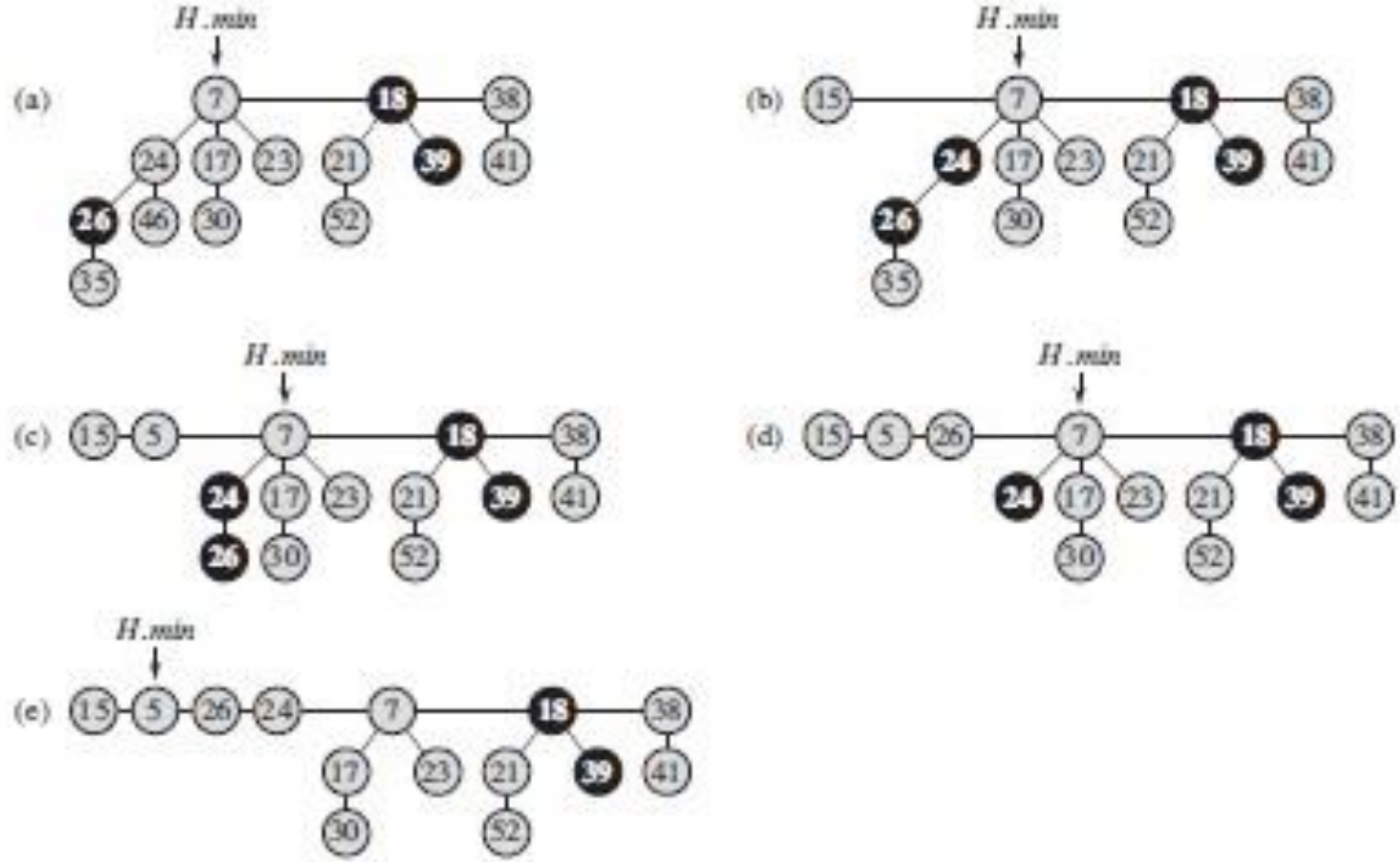
```
CONSOLIDATE(H)
 1  let A[0..D(H.n)] be a new array
 2  for i = 0 to D(H.n)
 3      A[i] = NIL
 4  for each node w in the root list of H
 5      x = w
 6      d = x.degree
 7      while A[d] ≠ NIL
 8          y = A[d]        // another node with the same degree as x
 9          if x.key > y.key
10              exchange x with y
11          FIB-HEAP-LINK(H, y, x)
12          A[d] = NIL
13          d = d + 1
14      A[d] = x
15  H.min = NIL
16  for i = 0 to D(H.n)
17      if A[i] ≠ NIL
18          if H.min == NIL
19              create a root list for H containing just A[i]
20              H.min = A[i]
21          else insert A[i] into H's root list
22              if A[i].key < H.min.key
23                  H.min = A[i]
```

```
FIB-HEAP-LINK(H, y, x)
1  remove y from the root list of H
2  make y a child of x, incrementing x.degree
3  y.mark = FALSE
```

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > x.key
2      error "new key is greater than current key"
3  x.key = k
4  y = x.p
5  if y ≠ NIL and x.key < y.key
6      CUT(H, x, y)
7      CASCADING-CUT(H, y)
8  if x.key < H.min.key
9      H.min = x

CUT(H, x, y)
1  remove x from the child list of y, decrementing y.degree
2  add x to the root list of H
3  x.p = NIL
4  x.mark = FALSE

CASCADING-CUT(H, y)
1  z = y.p
2  if z ≠ NIL
3      if y.mark == FALSE
4          y.mark = TRUE
5      else CUT(H, y, z)
6          CASCADING-CUT(H, z)
```


(a)
H.min
7
18
38
24
17
23
21
39
41
26
46
30
52
35
(b)
H.min
15
7
18
38
24
17
23
21
39
41
26
30
52
35
(c)
H.min
15
5
7
18
38
24
17
23
21
39
41
26
30
52
(d)
H.min
15
5
26
7
18
38
24
17
23
21
39
41
30
52
(e)
H.min
15
5
26
24
7
18
38
17
23
21
39
41
30
52