Fibonacci Heaps

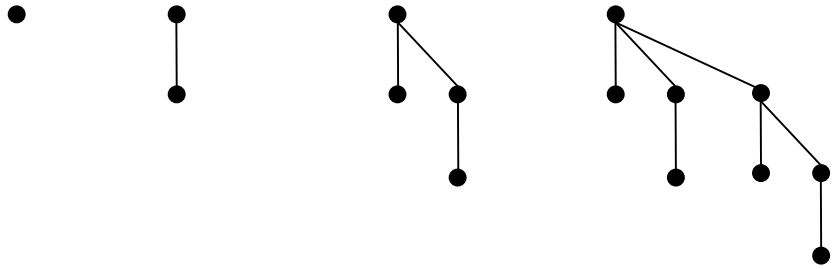
Procedure	Binary Heap Worst Case	**Binomial Heap (worst-case)**	Fibonacci heap (amortized)
Make Heap	Θ(1)	**Θ(1)**	Θ(1)
Insert	Θ(lgn)	**O(lg n)**	Θ(1)
Minimum	Θ(1)	**O(lg N)**	Θ(1)
Extract-Min	Θ(lg n)	**Θ(lg n)**	O(lg n)
Union	Θ(n)	**O(lg n)**	Θ(1)
Decrease-Key	Θ(lg n)	**Θ(lg n)**	Θ(1)
Delete	Θ(lg n)	**Θ(lg n)**	Θ(lg n)

Fibonacci Heaps

Fibonacci heap is a collection of min-heap-ordered trees(unordered binomial trees)

Basic idea.

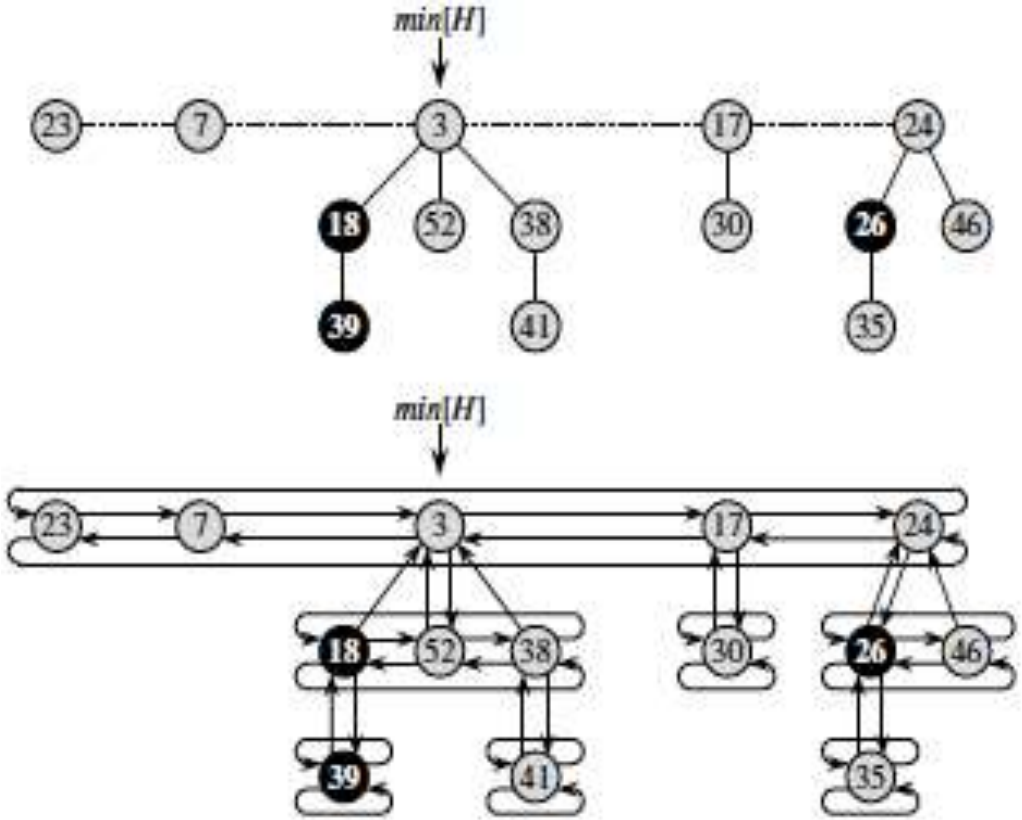
- Similar to binomial heaps, but less rigid structure.
- Binomial heap: eagerly consolidate trees after each insert.



- Fibonacci heap: lazily defer consolidation until next *delete-min*.

- From a theoretical standpoint, Fibonacci heaps are especially desirable when the number of EXTRACT-MIN and DELETE operations is small relative to the number of other operations performed.
- From a practical point of view, however, the constant factors and programming complexity of Fibonacci heaps make them less desirable than ordinary heaps for most applications, except for certain applications that manage large amounts of data.
- Thus, Fibonacci heaps are predominantly of theoretical interest.

- Each node x contains a pointer $p[x]$ to its parent and a pointer $child[x]$ to any one of its children.
- The children of x are linked together in a circular, doubly linked list, which we call the child list of x.
- Each child y in a child list has pointers $left[y]$ and $right[y]$ that point to y's left and right siblings.

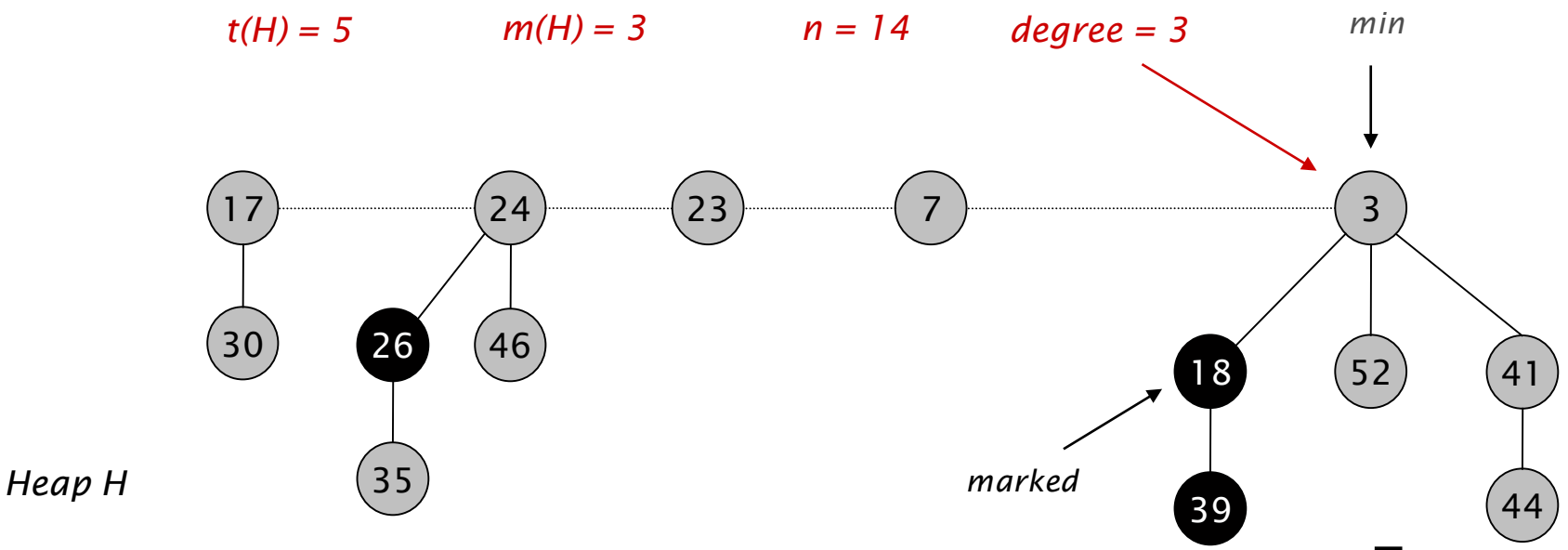


- The number of children in the child list of node *x* is stored in *degree[x]*.
- The boolean-valued field *mark[x]* indicates whether node *x* has lost a child since the last time *x* was made the child of another node.
- Newly created nodes are unmarked, and a node *x* becomes unmarked whenever it is made the child of another node.
- min[H] point to the root of a tree containing a minimum key

Fibonacci Heaps: Potential Function

For a given Fibonacci heap H, we indicate by t (H) the number of trees in the root list of H and by m(H) the number of marked nodes in H.
The potential of Fibonacci heap H is then defined by
$\phi(H) = t(H) + 2m(H)$.

$\Phi(H) = 5 + 2 \cdot 3 = 11$



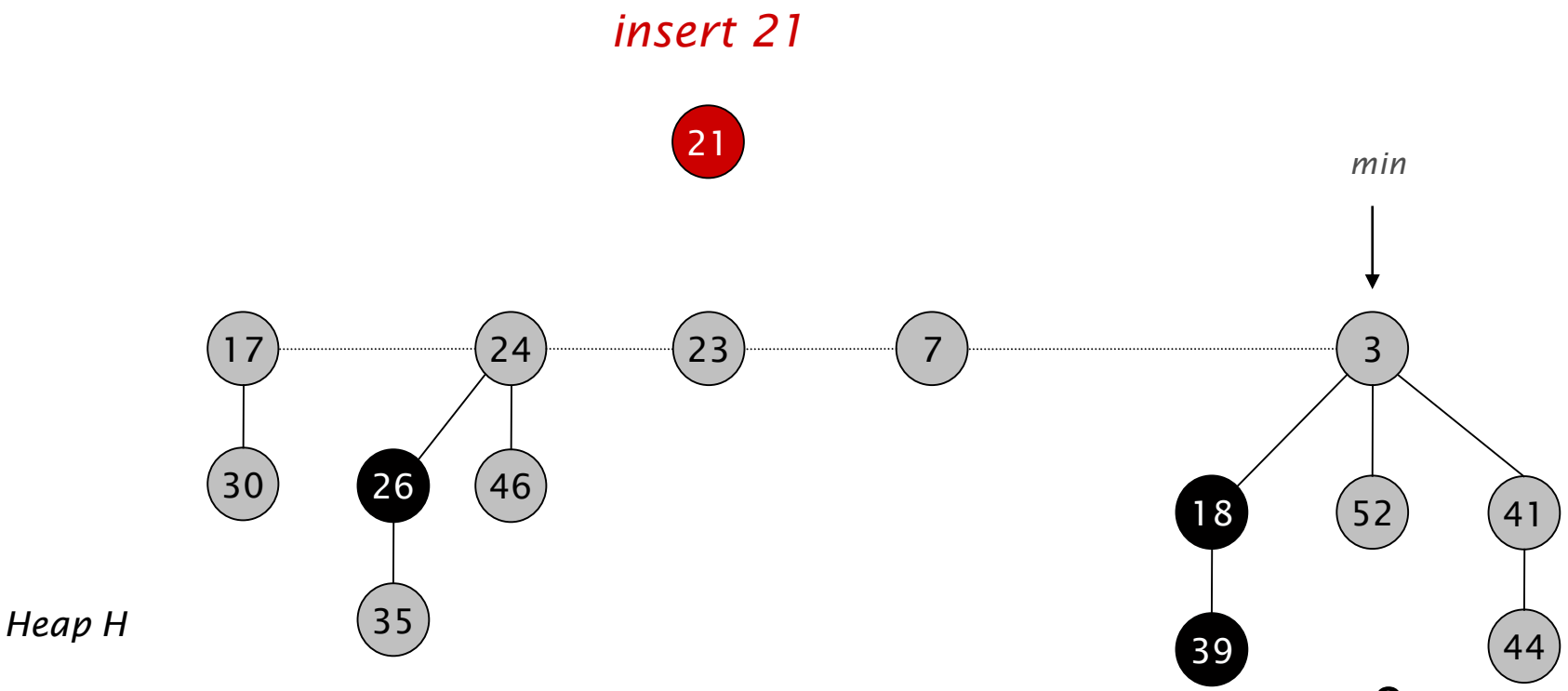
Insert

```
FIB-HEAP-INSERT(H, x)
1   degree[x] ← 0
2   p[x] ← NIL
3   child[x] ← NIL
4   left[x] ← x
5   right[x] ← x
6   mark[x] ← FALSE
7   concatenate the root list containing x with root list H
8   if min[H] = NIL or key[x] < key[min[H]]
9         then min[H]← x
10 n[H]← n[H] + 1
```

Fibonacci Heaps: Insert

Insert.

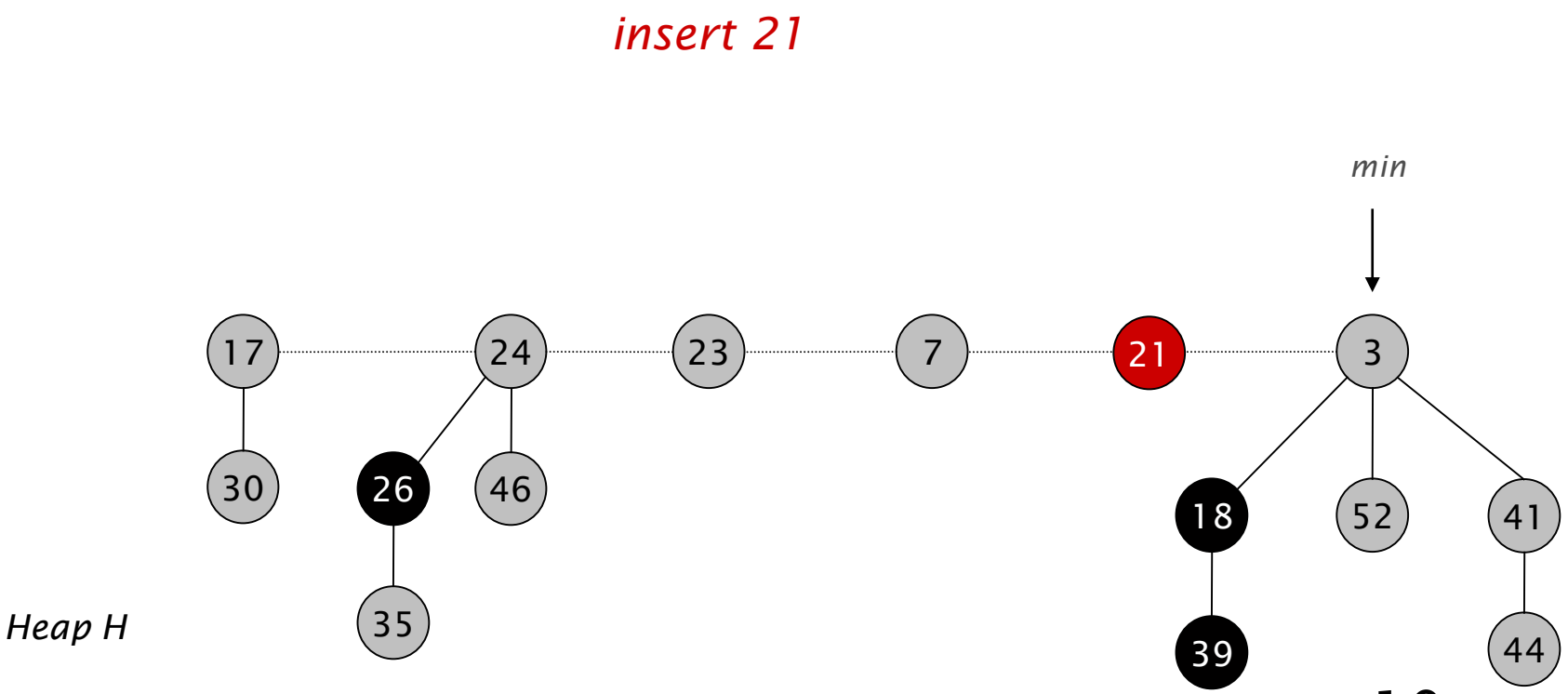
- Create a new singleton tree.
- Add to root list; update min pointer (if necessary).



Fibonacci Heaps: Insert

Insert.

- Create a new singleton tree.
- Add to root list; update min pointer (if necessary).

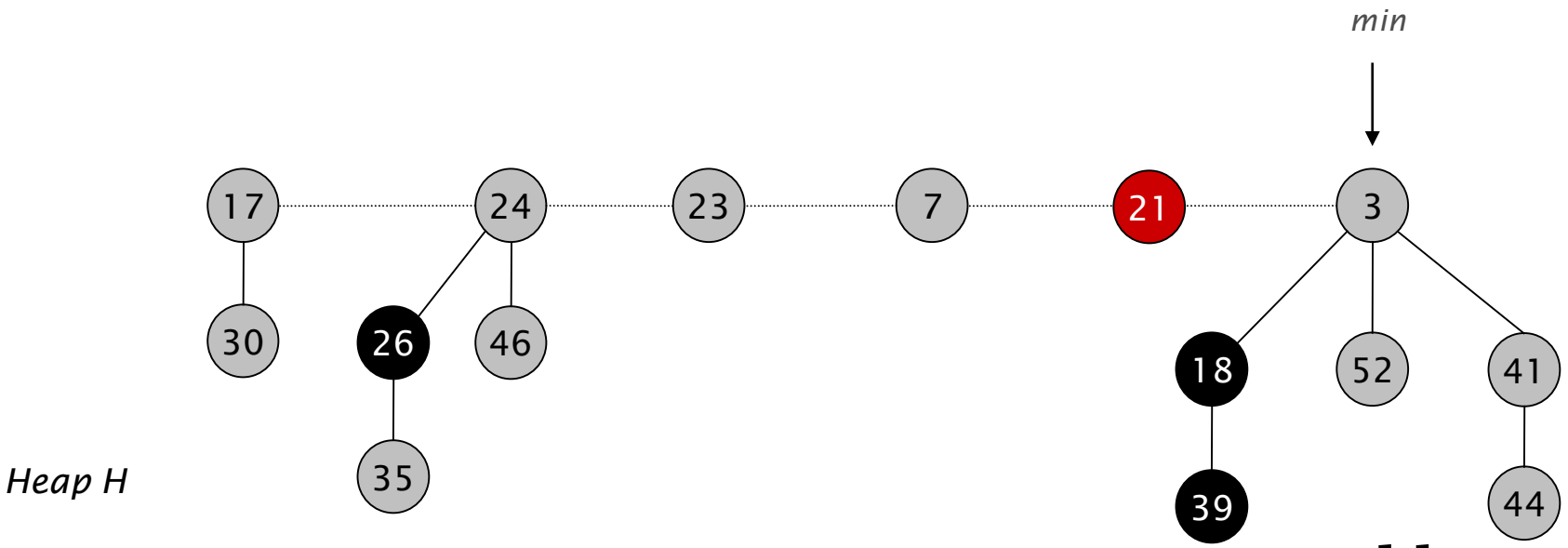


Fibonacci Heaps: Insert Analysis

Actual cost. O(1)

Change in potential. $((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1$

Amortized cost. O(1) + 1 = O(1)



Find Minimum

The minimum node of a Fibonacci heap H is given by the pointer min[H].

ANALYSIS
actual cost- O(1).
potential difference – 0.
amortized cost - O(1).

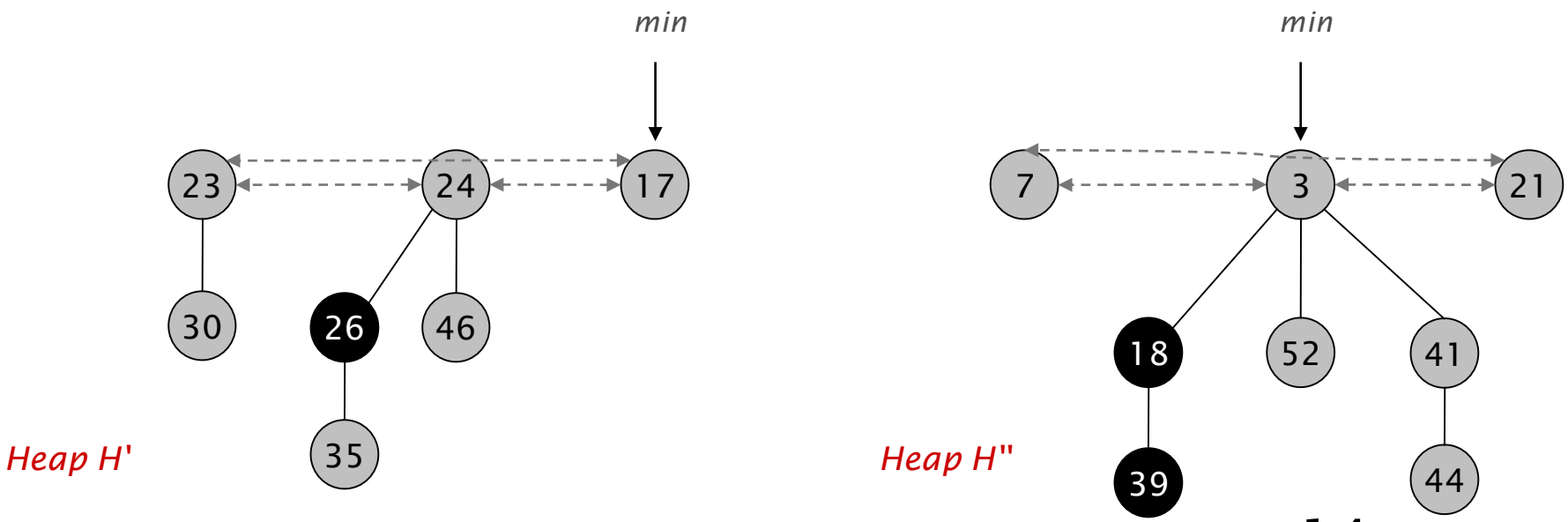
Union

```
FIB-HEAP-UNION(H1, H2)
1    H ← MAKE-FIB-HEAP()
2    min[H]← min[H1]
3    concatenate the root list of H2 with the root list of H
4    if (min[H1] = NIL) or (min[H2] ≠ NIL and key[min[H2]] <
key[min[H1]])
5          then min[H] ← min[H2]
6    n[H]← n[H1] + n[H2]
7    free the objects H1 and H2
8    return H
```

Fibonacci Heaps: Union

Union. Combine two Fibonacci heaps.

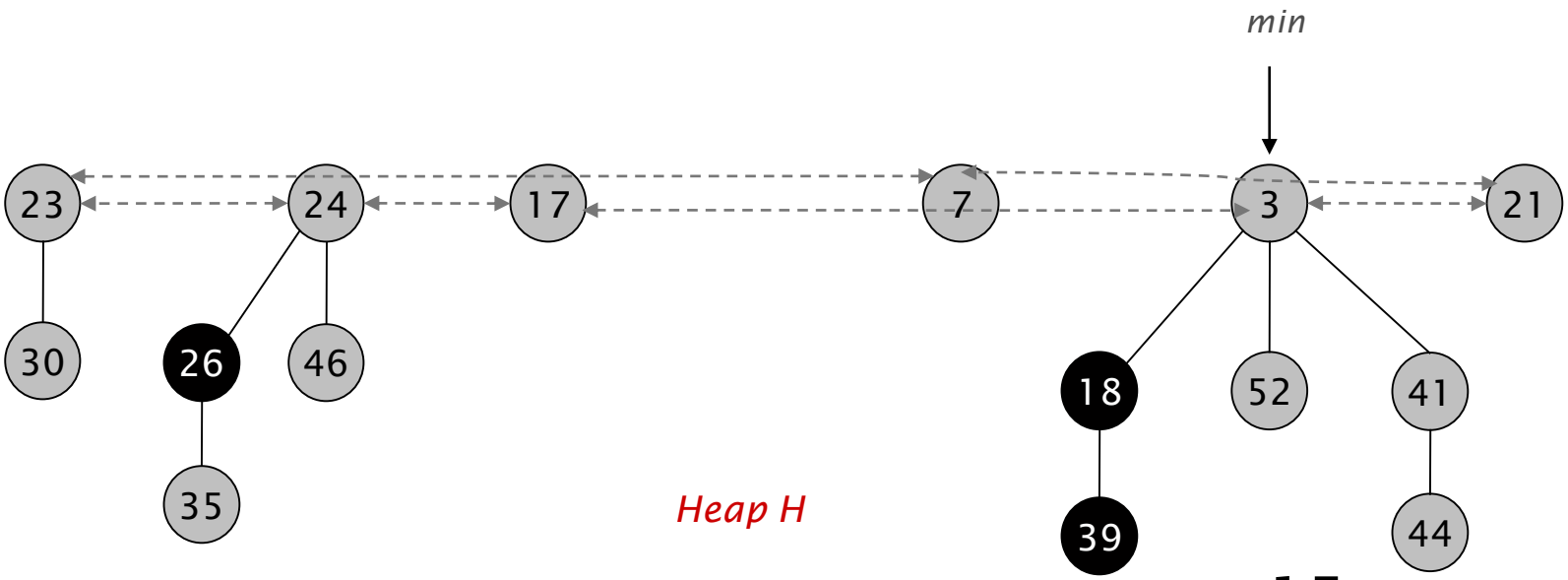
Representation. Root lists are circular, doubly linked lists.



Fibonacci Heaps: Union

Union. Combine two Fibonacci heaps.

Representation. Root lists are circular, doubly linked lists.



Fibonacci Heaps: Union

Actual cost. O(1)

Change in potential. $\phi(H) - (\phi(H1) + \phi(H2))$

$= (t(H) + 2m(H)) - ((t(H1) + 2m(H1)) + (t(H2) + 2m(H2)))$

$= 0$

Amortized cost. O(1)

Extract Min

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

Extract Min

- Consolidating the root list is performed by the call CONSOLIDATE(H).
- Consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a distinct degree value.

1. Find two roots *x* and *y* in the root list with the same degree, where key[x] ≤ key[y].
2. Link *y* to *x*: remove *y* from the root list, and make *y* a child of *x*. This operation is performed by the FIB-HEAP-LINK procedure. This procedure increments the degree of x and clears the mark on y.

Fibonacci Heaps: Extract Min

```
CONSOLIDATE(H)
1   for i ← 0 to D(n[H])
2       do A[i] ← NIL
3   for each node w in the root list of H
4       do x ← w
5          d ← degree[x]
6          while A[d] ≠ NIL
7             do y ← A[d] // Another node with the same degree as x.
8                if key[x] > key[y]
9                   then exchange x ↔ y
10               FIB-HEAP-LINK(H, y, x)
11               A[d] ← NIL
12               d ← d + 1
13         A[d] ← x
14  min[H] ← NIL
```

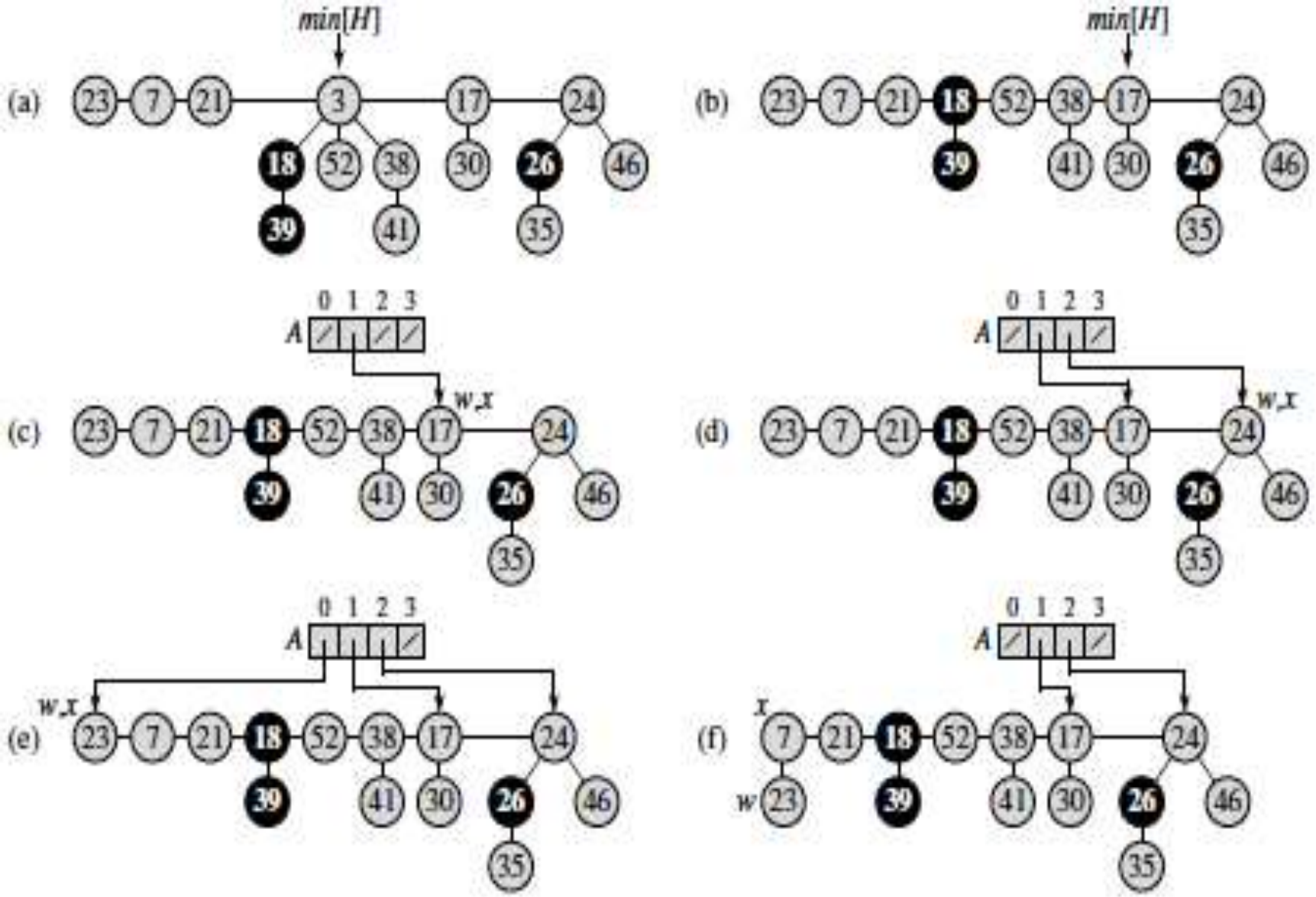
Fibonacci Heaps: Extract Min(contd..)

```
15  for i ← 0 to D(n[H])
16      do if A[i ] ≠NIL
17             then add A[i ] to the root list of H
18                  if min[H] = NIL or key[A[i ] < key[min[H]
19                       then min[H] ← A[i ]
```

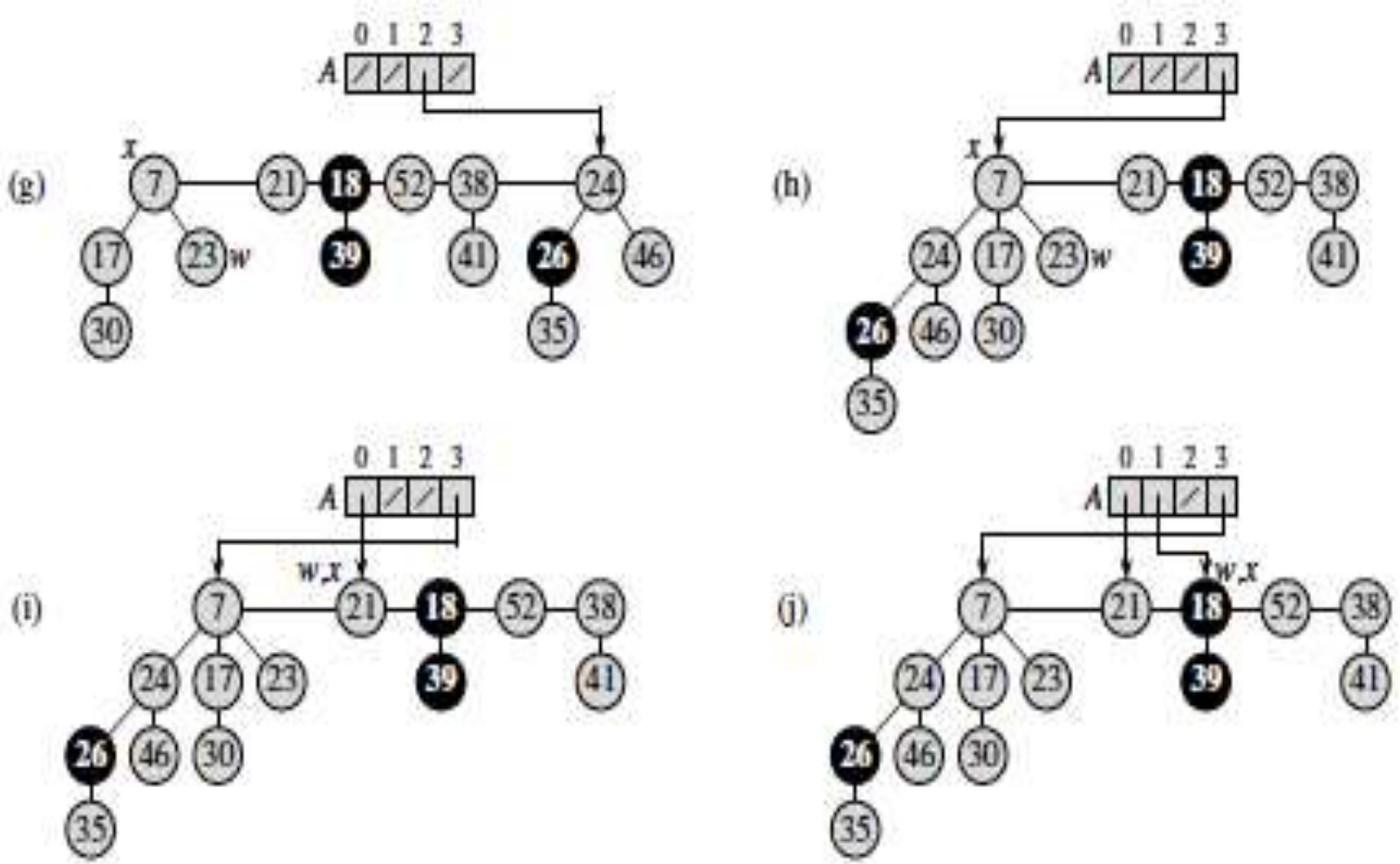
FIB-HEAP-LINK(H, *y, x*)

```
1  remove y from the root list of H
2  make y a child of x, incrementing degree[x]
3  mark[y] ← FALSE
```

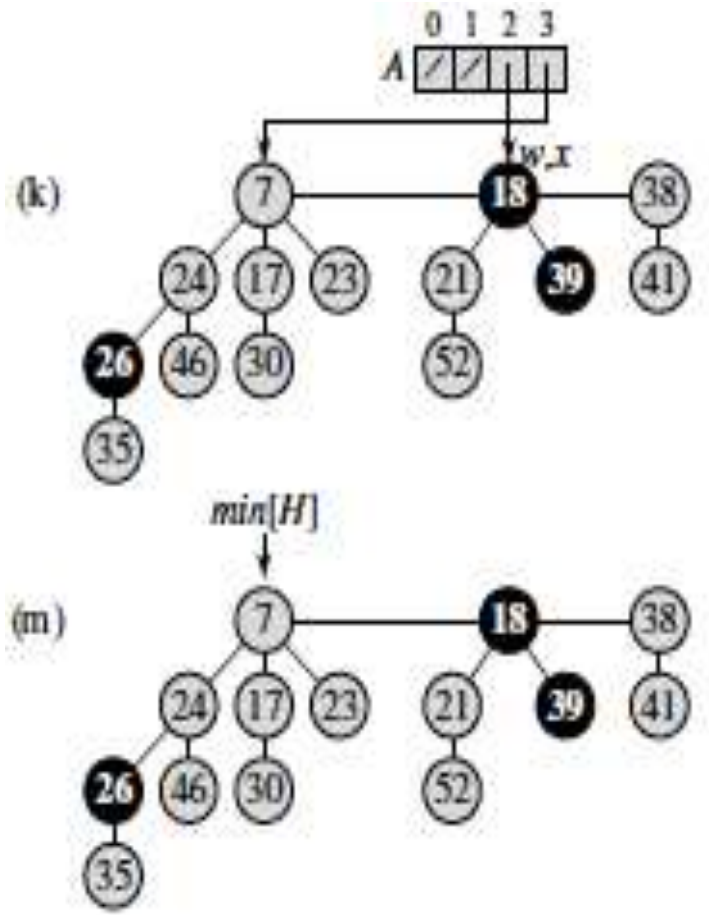
Fibonacci Heaps: Extract Min



Fibonacci Heaps: Extract Min



Fibonacci Heaps: Extract Min



Fibonacci Heaps: Extract Min Analysis

Actual cost.
O(D(n)) - at most D(n) children of the minimum node that are processed in FIB-HEAP-EXTRACT-MIN and from the work in lines 1–2 and 14–19 of CONSOLIDATE.

Consider for loop of lines 3–13(CONSOLIDATE). Size of the root list upon calling CONSOLIDATE is at most D(n) + t (H) − 1, since it consists of the original t (H) root list nodes, minus the extracted root node, plus the children of the extracted node, which number at most D(n).

Every time through the while loop of lines 6–12, one of the roots is linked to another, thus the total amount of work performed is at most proportional to D(n)+ t (H).

Thus, the total actual work in extracting the minimum node is O(D(n)+t(H)).

Fibonacci Heaps: Extract Min Analysis

Change in potential.

The potential before extracting the minimum node is t (H) + 2m(H), and afterward is at most (D(n) + 1) + 2m(H), since at most D(n) + 1 roots remain and no nodes become marked during the operation.

Amortized cost.

$O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H))$

$= O(D(n)) + O(t(H)) - t(H)$

$= O(D(n)) = O(\log n)$.

Decrease Key

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

Fibonacci Heaps: Decrease Key

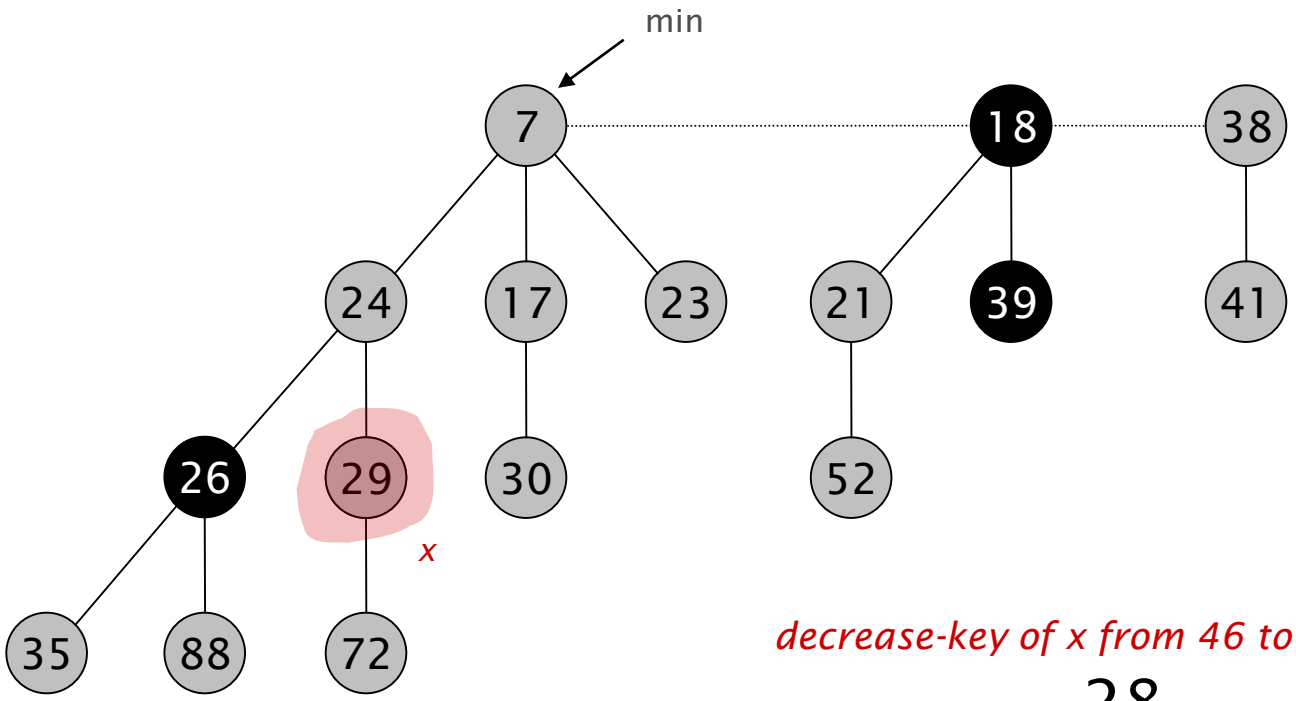
```
CUT(H, x, y)
1   remove x from the child list of y, decrementing degree[y]
2    add x to the root list of H
3    p[x] ← NIL
4   mark[x] ← FALSE


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

Fibonacci Heaps: Decrease Key

Case 1. [heap order not violated]

- Decrease key of *x*.
- Change heap min pointer (if necessary).

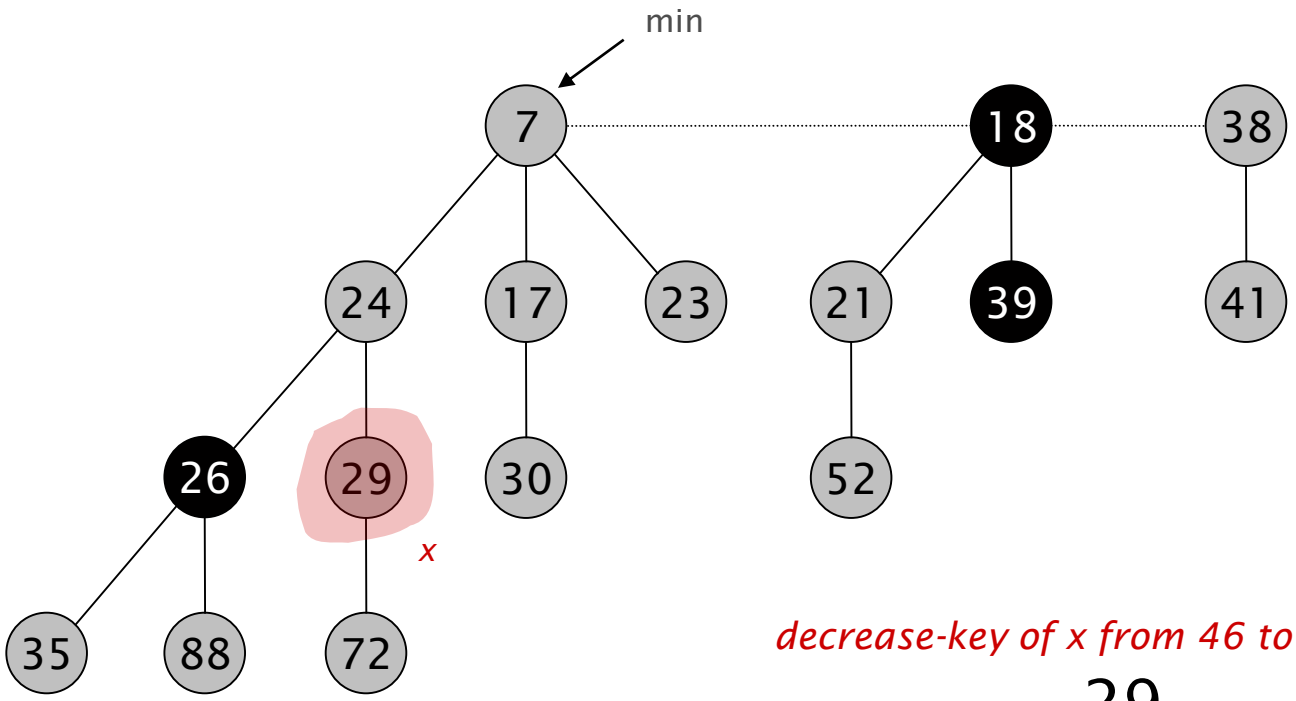


decrease-key of x from 46 to 29

Fibonacci Heaps: Decrease Key

Case 1. [heap order not violated]

- Decrease key of *x*.
- Change heap min pointer (if necessary).

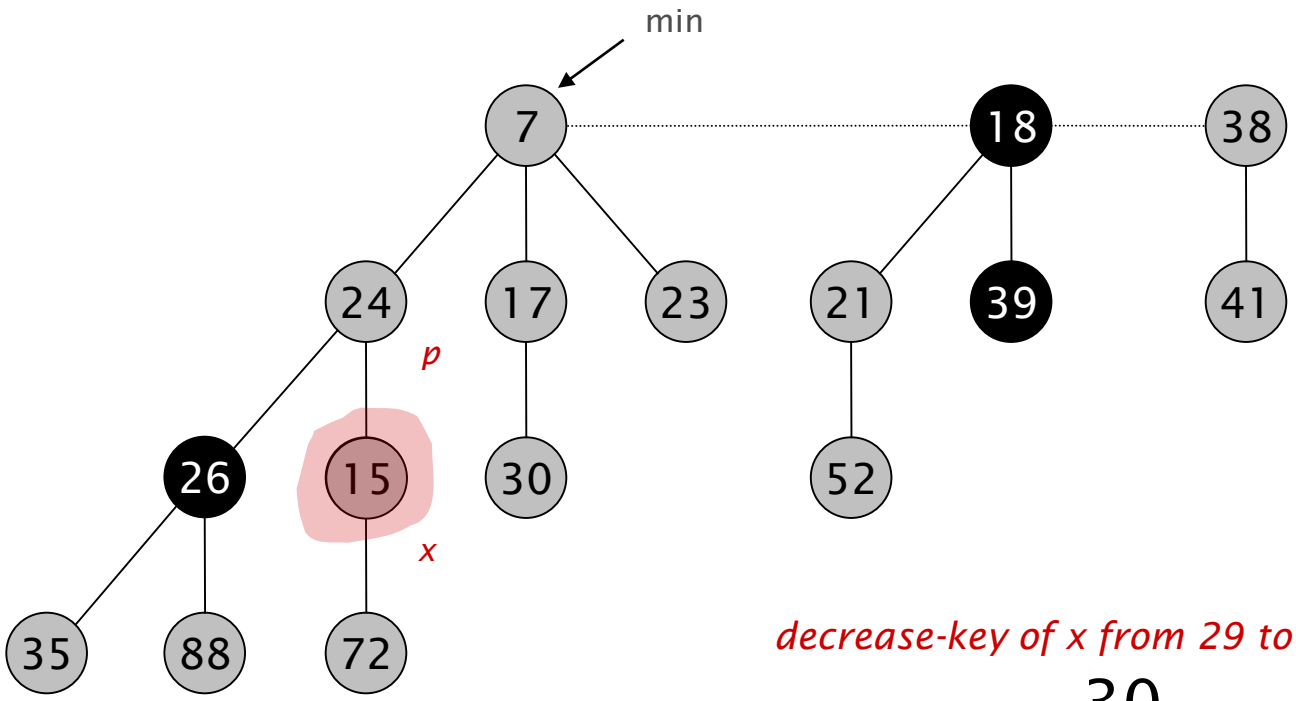


decrease-key of x from 46 to 29

Fibonacci Heaps: Decrease Key

Case 2a. [heap order violated]

- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

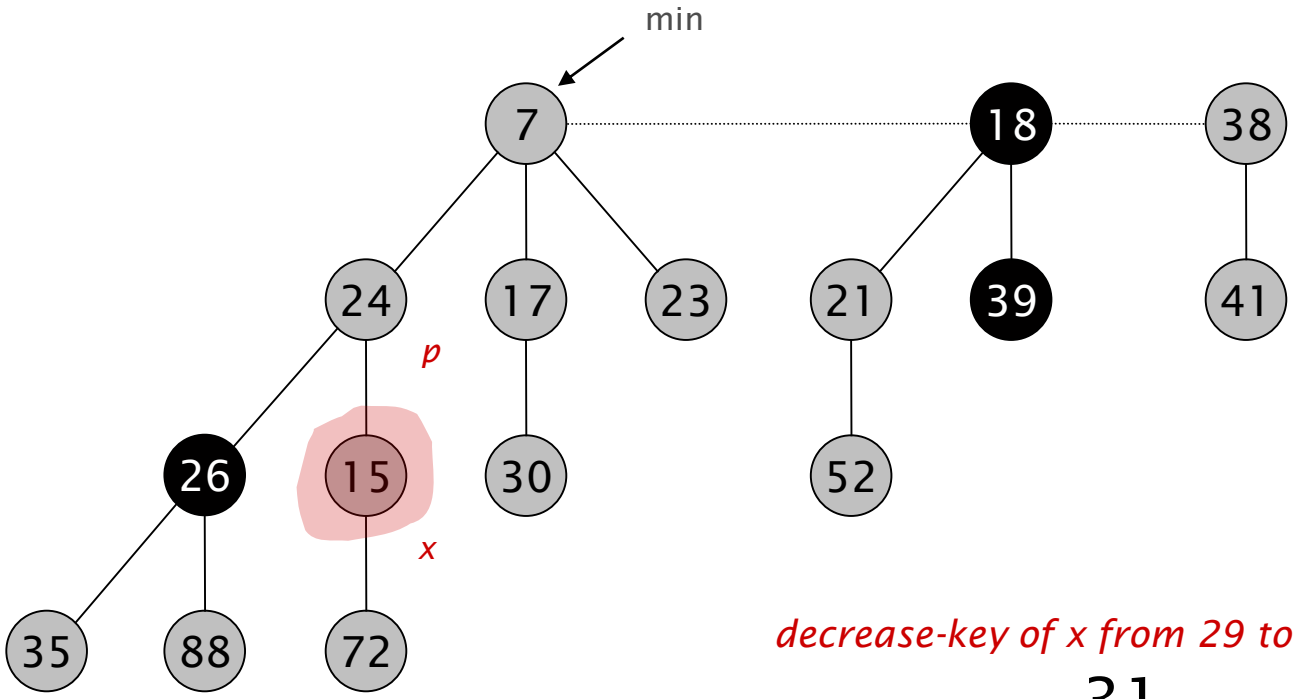


decrease-key of x from 29 to 15

Fibonacci Heaps: Decrease Key

Case 2a. [heap order violated]

- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

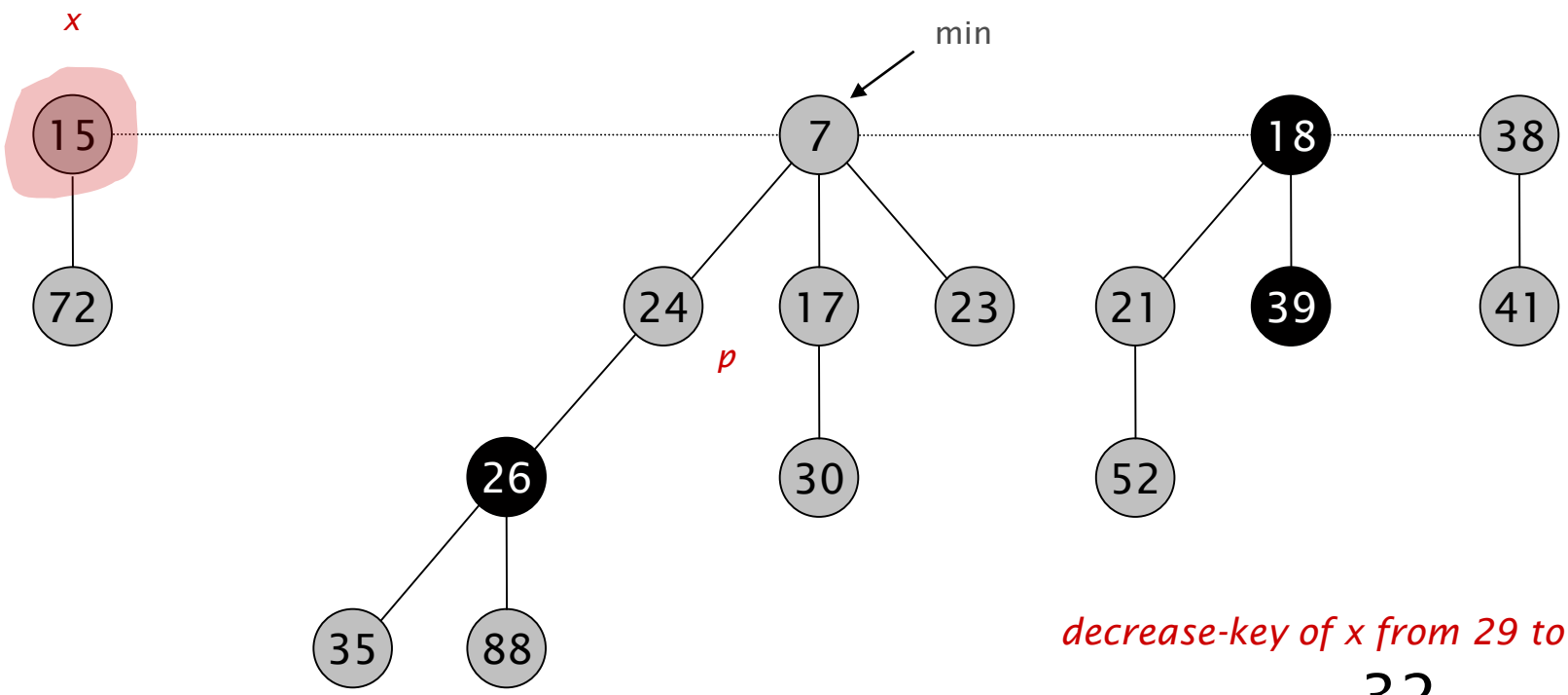


decrease-key of x from 29 to 15

Fibonacci Heaps: Decrease Key

Case 2a. [heap order violated]

- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

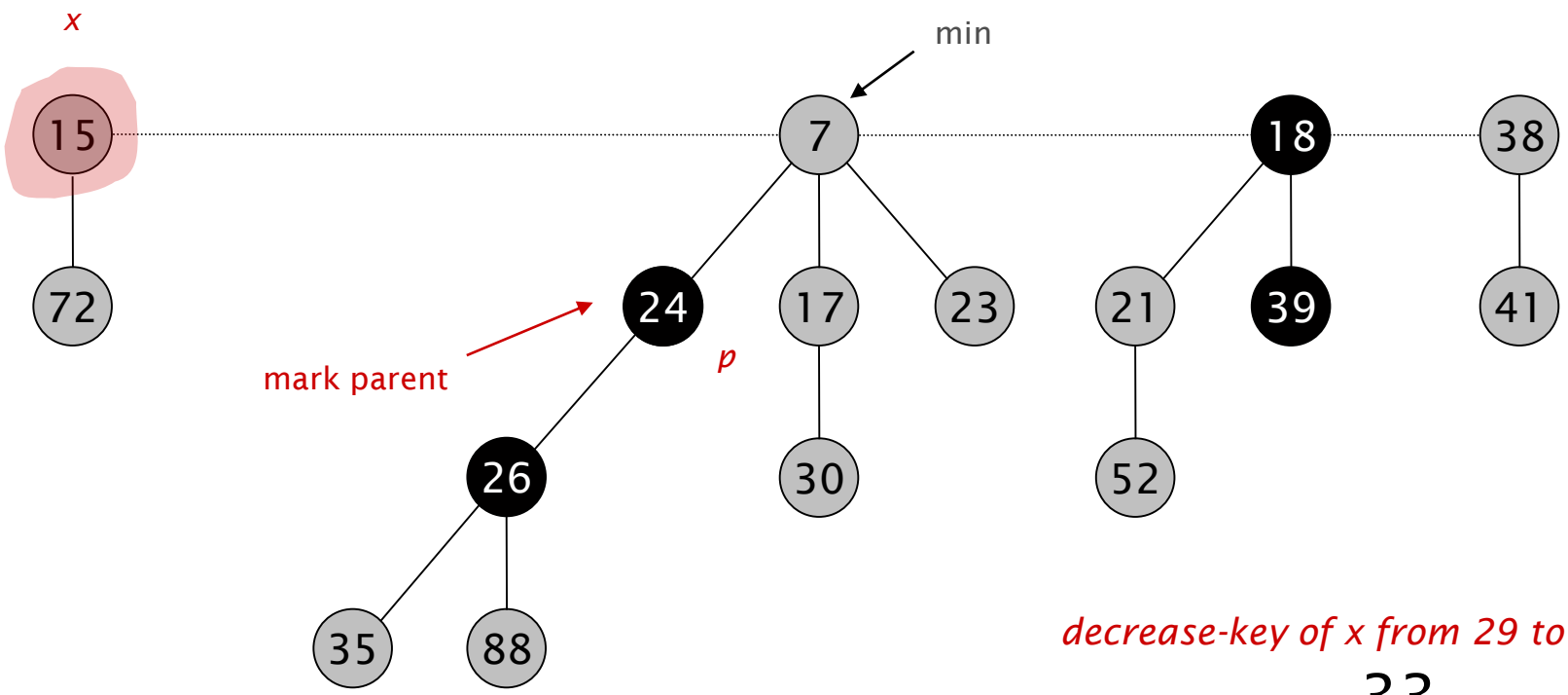


decrease-key of x from 29 to 15

Fibonacci Heaps: Decrease Key

Case 2a. [heap order violated]

- Decrease key of x.
- Cut tree rooted at x, meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut p, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

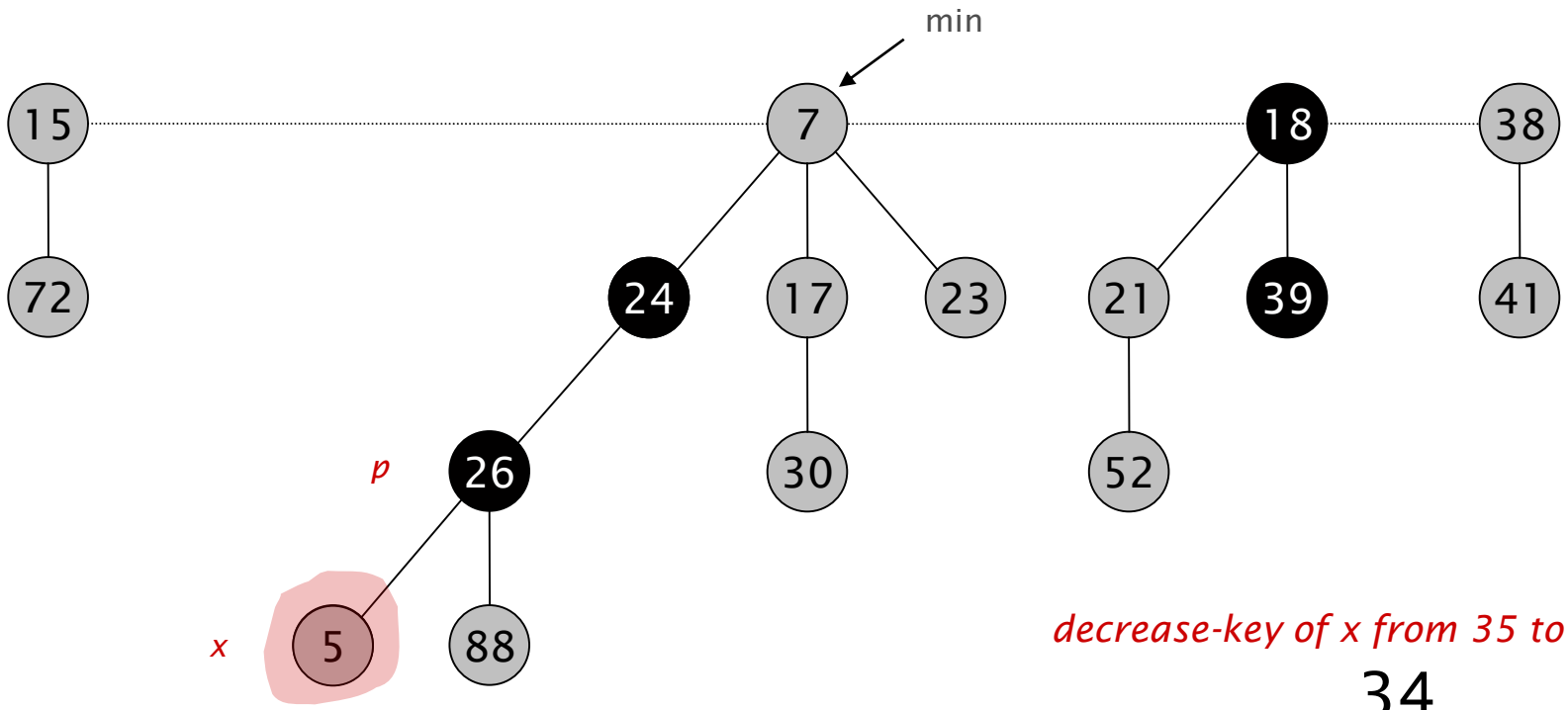


decrease-key of x from 29 to 15

Fibonacci Heaps: Decrease Key

Case 2b. [heap order violated]

- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

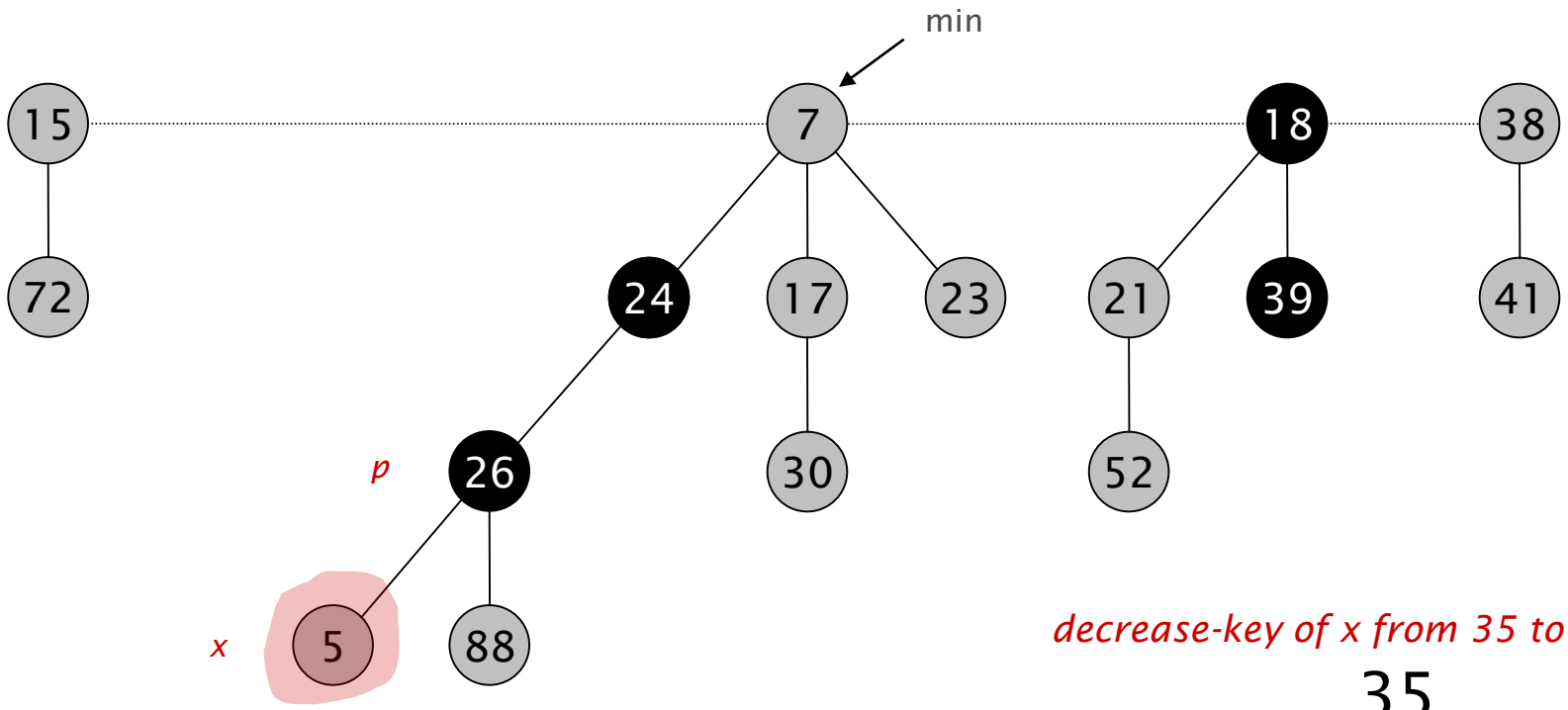


decrease-key of x from 35 to 5

Fibonacci Heaps: Decrease Key

Case 2b. [heap order violated]

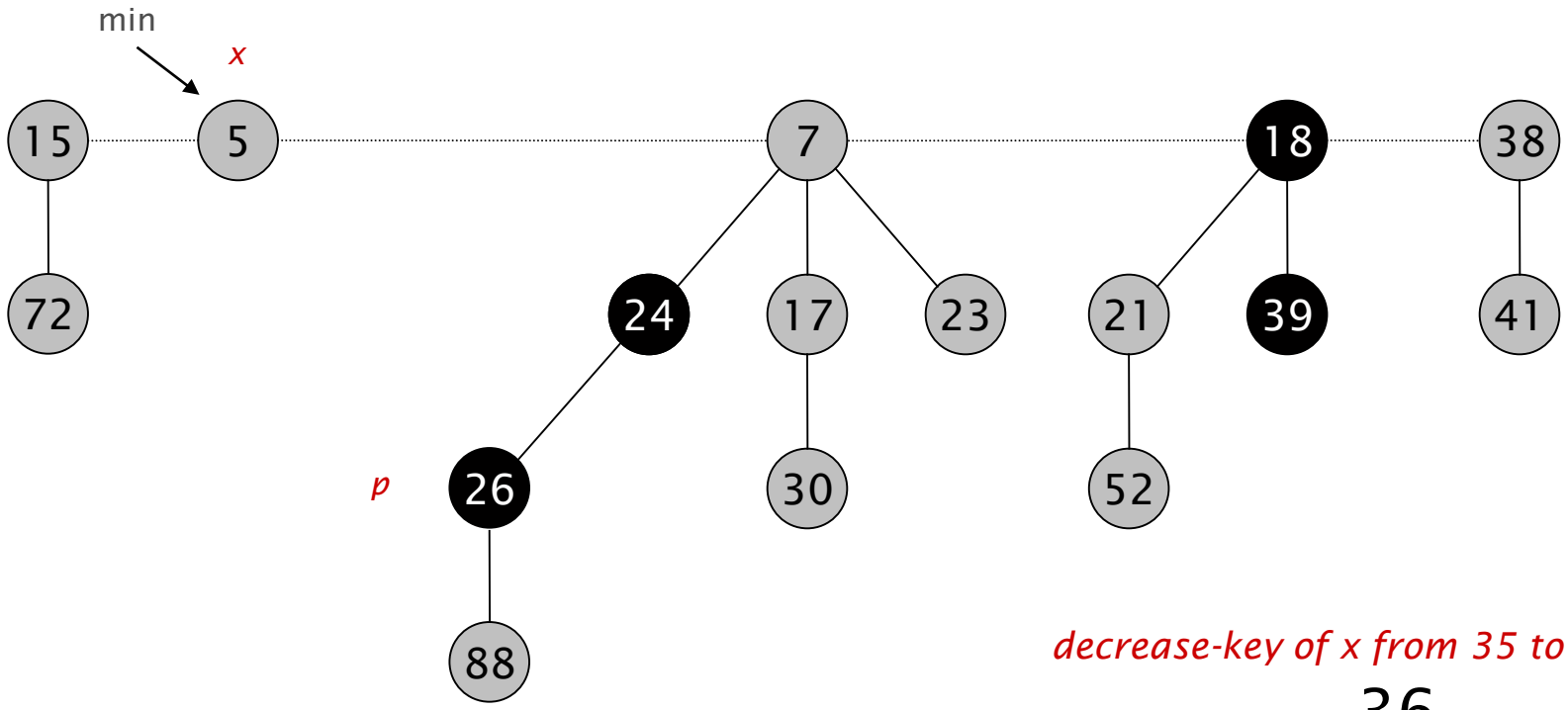
- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).



Fibonacci Heaps: Decrease Key

Case 2b. [heap order violated]

- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

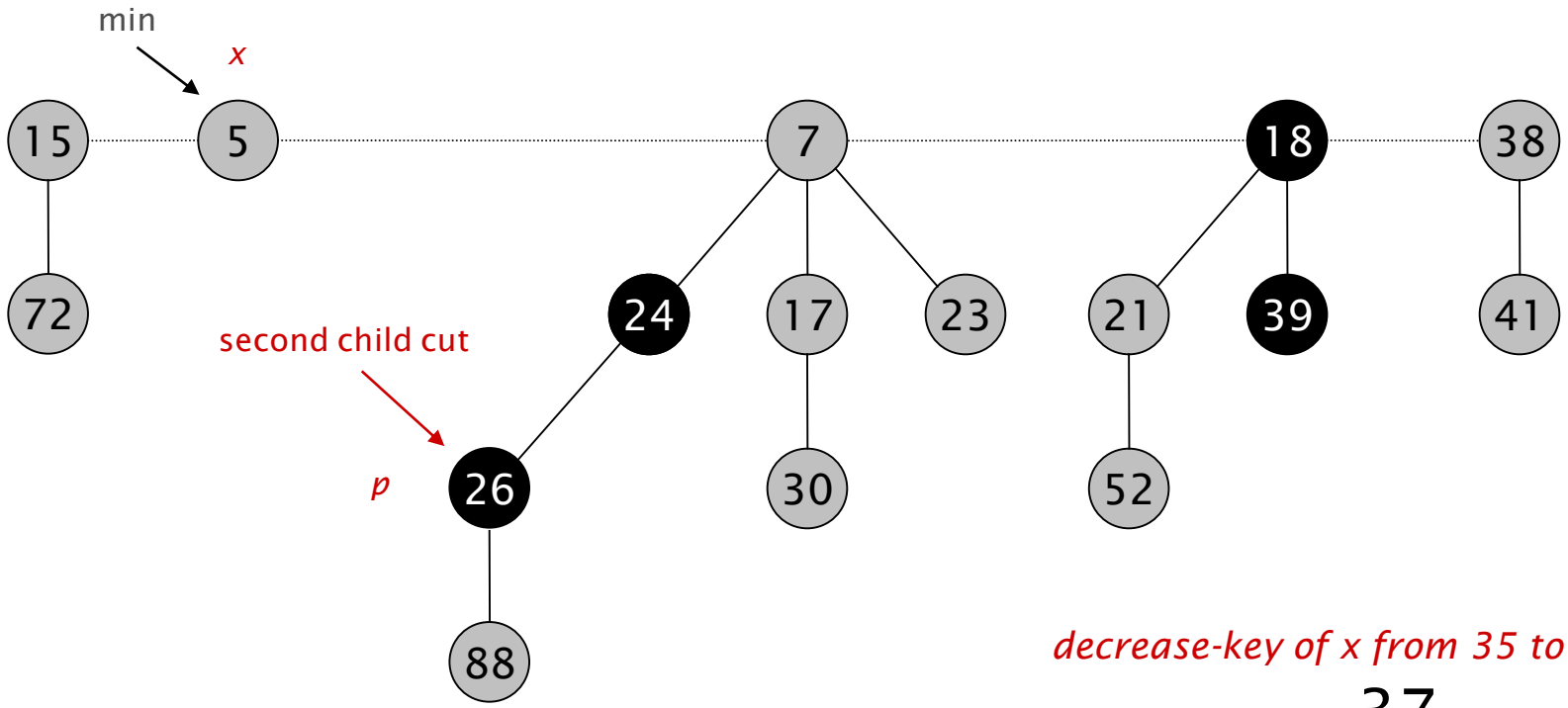


decrease-key of x from 35 to 5

Fibonacci Heaps: Decrease Key

Case 2b. [heap order violated]

- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).

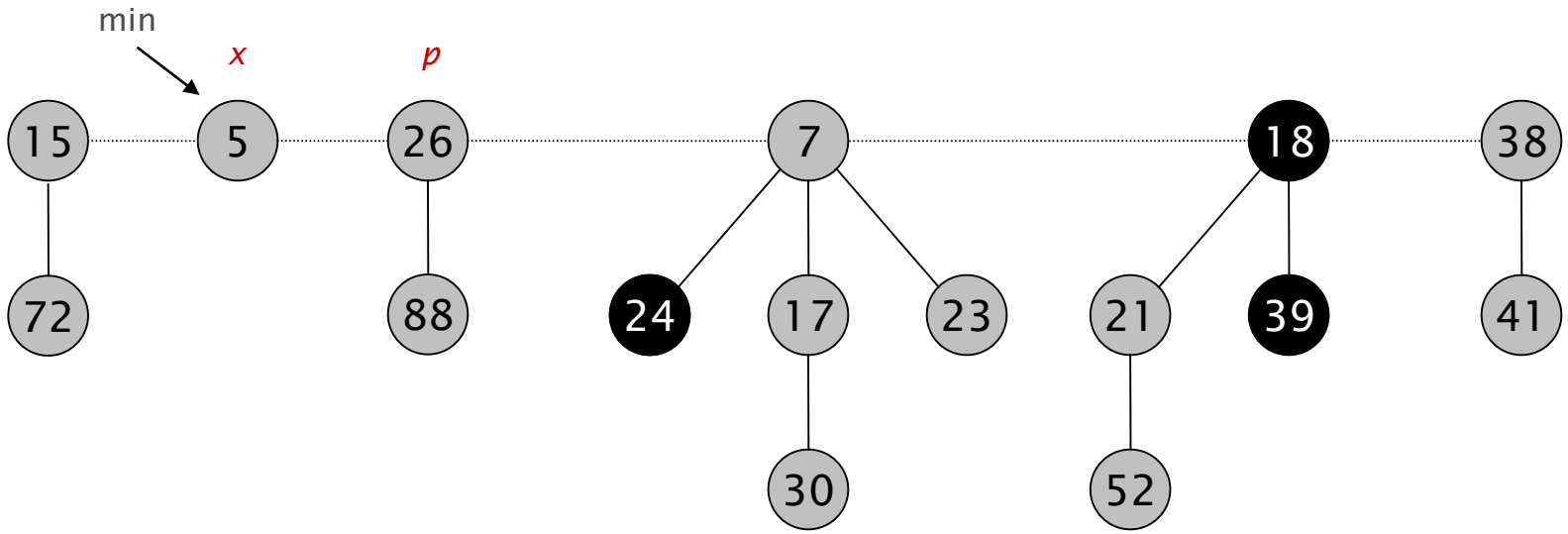


decrease-key of x from 35 to 5

Fibonacci Heaps: Decrease Key

Case 2b. [heap order violated]

- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).



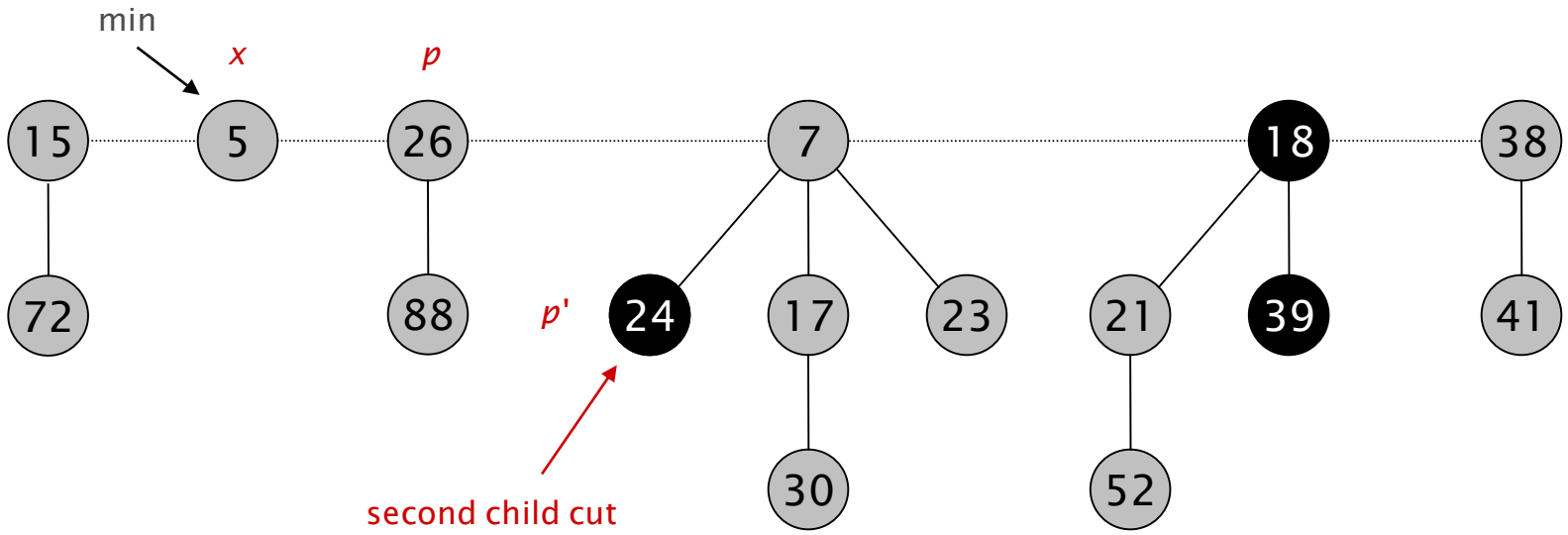
decrease-key of x from 35 to 5

Fibonacci Heaps: Decrease Key

Case 2b. [heap order violated]

- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark

  (and do so recursively for all ancestors that lose a second child).



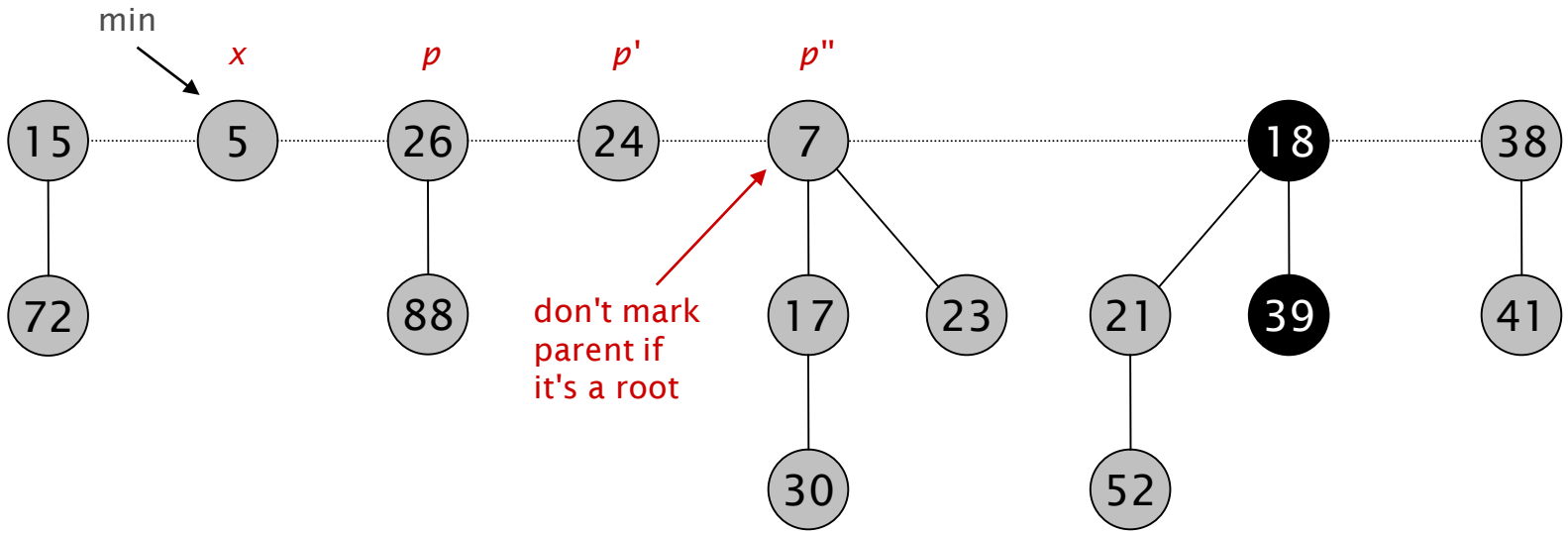
decrease-key of x from 35 to 5

Fibonacci Heaps: Decrease Key

Case 2b. [heap order violated]

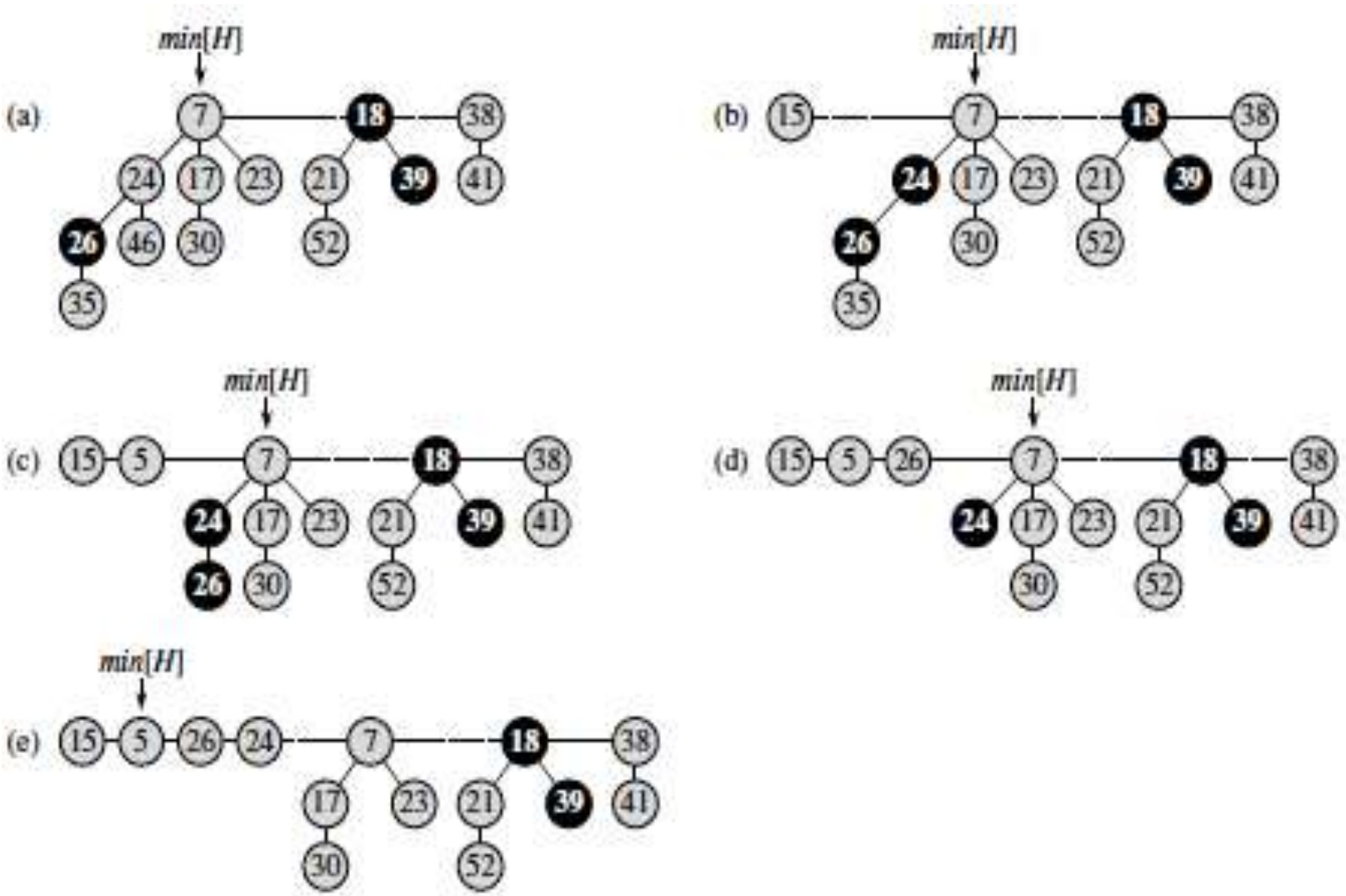
- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark

  (and do so recursively for all ancestors that lose a second child).



decrease-key of x from 35 to 5

Fibonacci Heaps: Decrease Key



a) Decrease key 46 to 15. b) Decrease key 35 to 5.

Fibonacci Heaps: Decrease Key

We use *mark* fields to obtain the desired time bounds.

They record a little piece of the history of each node.

Initially, the *mark* field of a node is set to FALSE.

When its first child is lost,*mark* field is set to TRUE.

On losing the second child, the node(the subtree) is cut and connected to the root list and *mark* field is set to FALSE.

This process continues upward till the root of the tree is reached.

Fibonacci Heaps: Decrease Key Analysis

Actual cost. O(c)

O(1) time for changing the key(FIB-HEAP-DECREASE-KEY & CUT).

O(1) time for each of c cuts(CASCADING-CUT is recursively called c times), plus melding into root list(CUT).

Change in potential. 4 - c

t(H') = t(H) + c (the original t (H) trees, c−1 trees produced by cascading cuts, and the tree rooted at x).

m(H') <= m(H) - c + 2 (c−1 were unmarked by cascading cuts and the last call of CASCADING-CUT may have marked a node).

((t (H) + c) + 2(m(H) − c + 2)) − (t (H) + 2m(H)) = 4 − c

Amortized cost. O(c) + 4 − c = O(1)

Delete

We assume that there is no key value of $-\infty$ currently in the Fibonacci heap.

FIB-HEAP-DELETE(H, x)

1 FIB-HEAP-DECREASE-KEY(H, x,$-\infty$)

2 FIB-HEAP-EXTRACT-MIN(H)

Fibonacci Heaps: Delete

Delete node x.

decrease-key of x to -∞.

- delete-min element in heap.

Amortized cost.

- O(1) amortized for decrease-key.
- O(D(n)) = O(log n) amortized for extract-min.
- O(log n) amortized for delete .