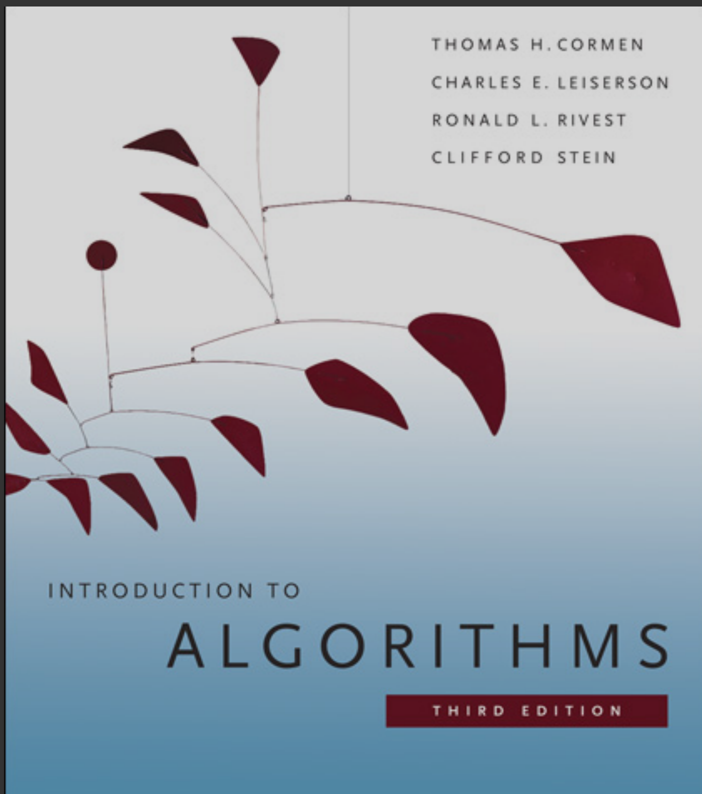


FIBONACCI HEAPS

- *preliminaries*
- *insert*
- *extract the minimum*
- *decrease key*
- *bounding the rank*
- *meld and delete*

Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Last updated on Apr 15, 2013 6:59 AM

Priority queues performance cost summary

operation	linked list	binary heap	binomial heap	Fibonacci heap †
MAKE-HEAP	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IS-EMPTY	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
EXTRACT-MIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
MELD	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
FIND-MIN	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$

† amortized

Ahead. $O(1)$ INSERT and DECREASE-KEY, $O(\log n)$ EXTRACT-MIN.

Fibonacci heaps

Theorem. [Fredman-Tarjan 1986] Starting from an empty Fibonacci heap, any sequence of m INSERT, EXTRACT-MIN, and DECREASE-KEY operations involving n INSERT operations takes $O(m + n \log n)$ time.

this statement is a bit weaker than the actual theorem

Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms

MICHAEL L. FREDMAN

University of California, San Diego, La Jolla, California

AND

ROBERT ENDRE TARJAN

AT&T Bell Laboratories, Murray Hill, New Jersey

Abstract. In this paper we develop a new data structure for implementing heaps (priority queues). Our structure, *Fibonacci heaps* (abbreviated *F-heaps*), extends the binomial queues proposed by Vuillemin and studied further by Brown. F-heaps support arbitrary deletion from an n-item heap in $O(\log n)$ amortized time and all other standard heap operations in $O(1)$ amortized time. Using F-heaps we are able to obtain improved running times for several network optimization algorithms. In particular, we obtain the following worst-case bounds, where n is the number of vertices and m the number of edges in the problem graph:

(1) $O(n \log n + m)$ for the single-source shortest path problem with nonnegative edge lengths, improved from $O(m \log_{(m/n+2)} n)$;
(2) $O(n^2 \log n + nm)$ for the all-pairs shortest path problem, improved from $O(nm \log_{(m/n+2)} n)$;
(3) $O(n^2 \log n + nm)$ for the assignment problem (weighted bipartite matching), improved from $O(nm \log_{(m/n+2)} n)$;
(4) $O(m\beta(m, n))$ for the minimum spanning tree problem, improved from $O(m \log \log_{(m/n+2)} n)$, where $\beta(m, n) = \min \{i \mid \log^{(i)} n \leq m/n\}$. Note that $\beta(m, n) \leq \log^* n$ if $m \geq n$.

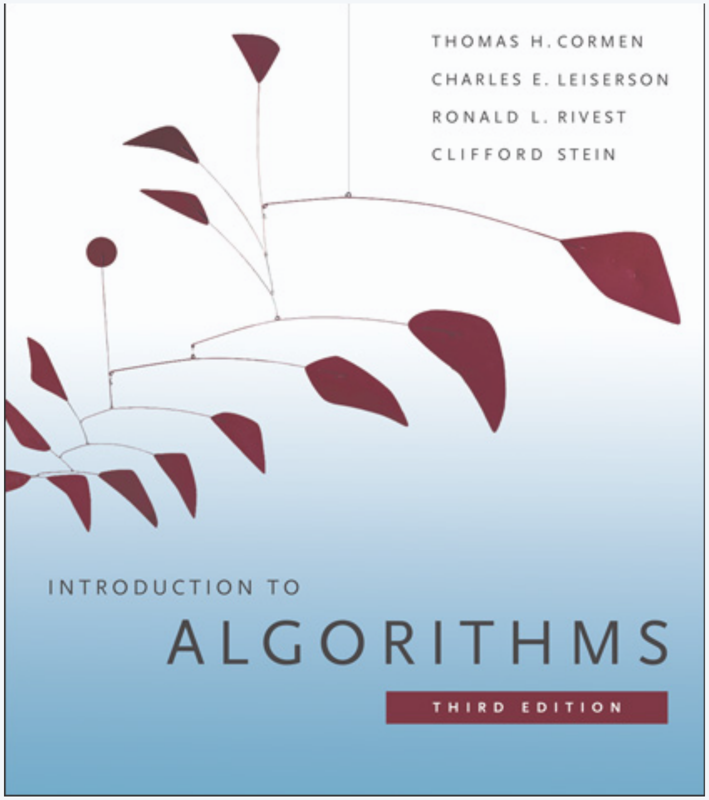
Of these results, the improved bound for minimum spanning trees is the most striking, although all the results give asymptotic improvements for graphs of appropriate densities.

Fibonacci heaps

Theorem. [Fredman-Tarjan 1986] Starting from an empty Fibonacci heap, any sequence of m INSERT, EXTRACT-MIN, and DECREASE-KEY operations involving n INSERT operations takes $O(m + n \log n)$ time.

History.

- Ingenious data structure and application of amortized analysis.
- Original motivation: improve Dijkstra's shortest path algorithm from $O(m \log n)$ to $O(m + n \log n)$.
- Also improved best-known bounds for all-pairs shortest paths, assignment problem, minimum spanning trees.



SECTION 19.1

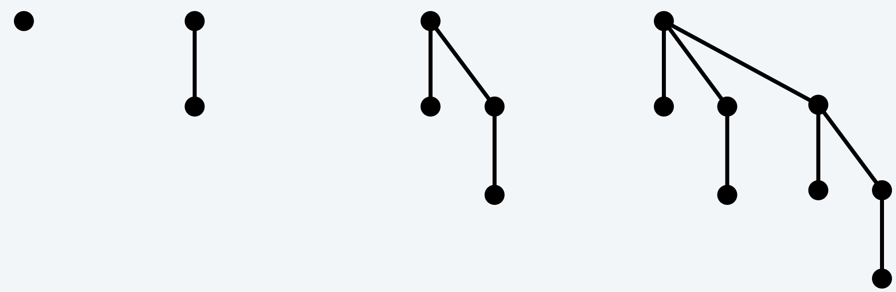
FIBONACCI HEAPS

- *structure*
- *insert*
- *extract the minimum*
- *decrease key*
- *bounding the rank*
- *meld and delete*

Fibonacci heaps

Basic idea.

- Similar to binomial heaps, but less rigid structure.
- Binomial heap: eagerly consolidate trees after each INSERT; implement DECREASE-KEY by repeatedly exchanging node with its parent.



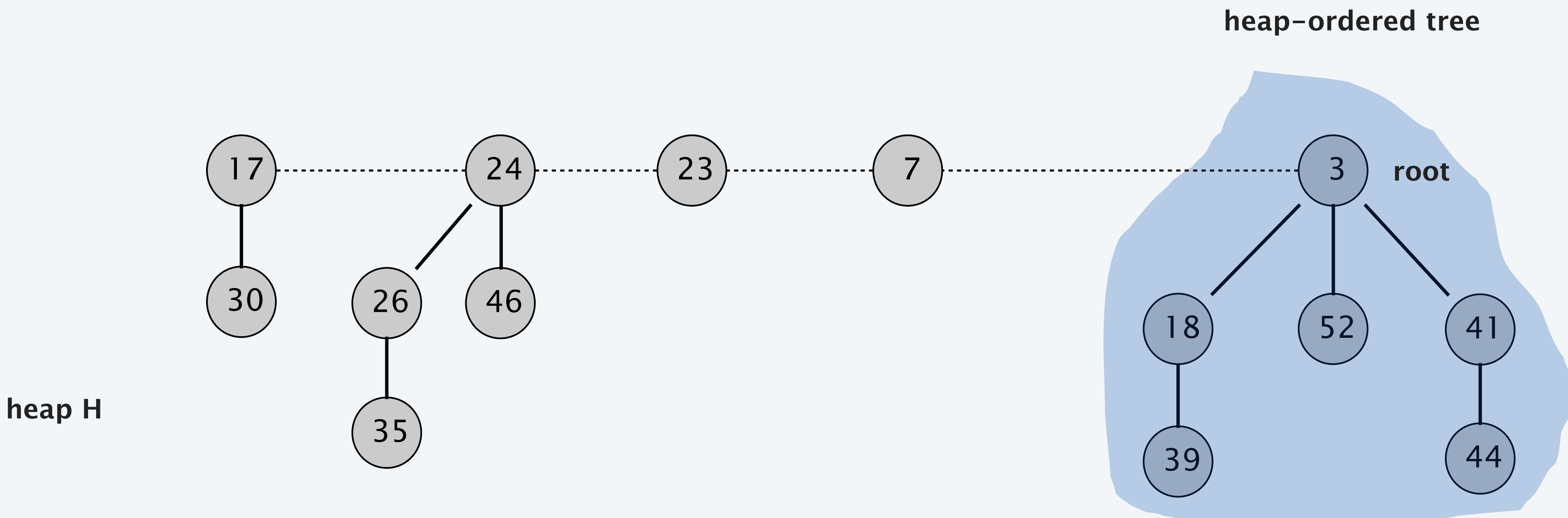
- Fibonacci heap: lazily defer consolidation until next EXTRACT-MIN; implement DECREASE-KEY by cutting off node and splicing into root list.

Remark. Height of Fibonacci heap is $\Theta(n)$ in worst case, but it doesn't use sink or swim operations.

Fibonacci heap: structure

- Set of heap-ordered trees.

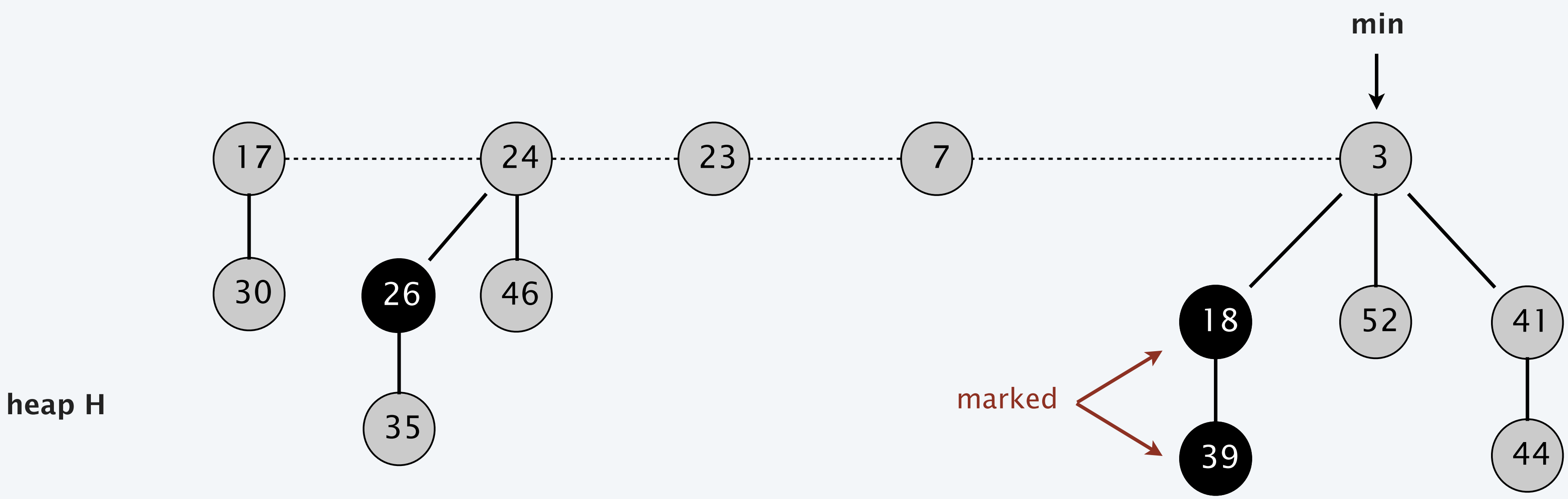
each child no smaller
than its parent



Fibonacci heap: structure

- Set of heap-ordered trees.
- Set of marked nodes.

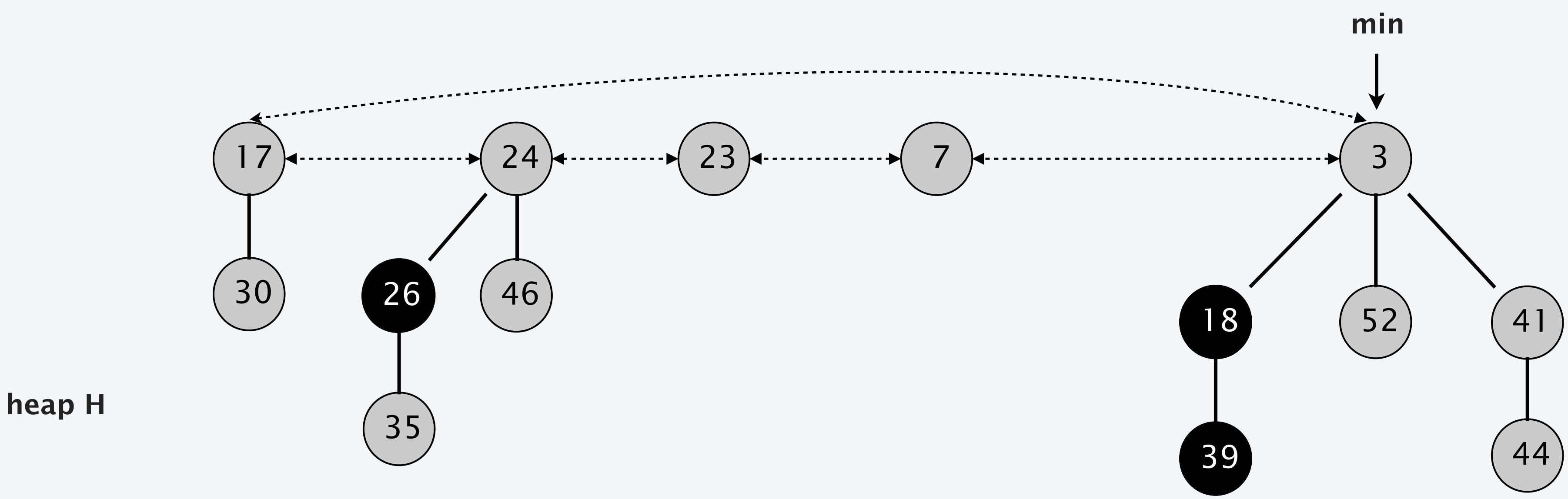
used to keep trees bushy
(stay tuned)



Fibonacci heap: structure

Heap representation.

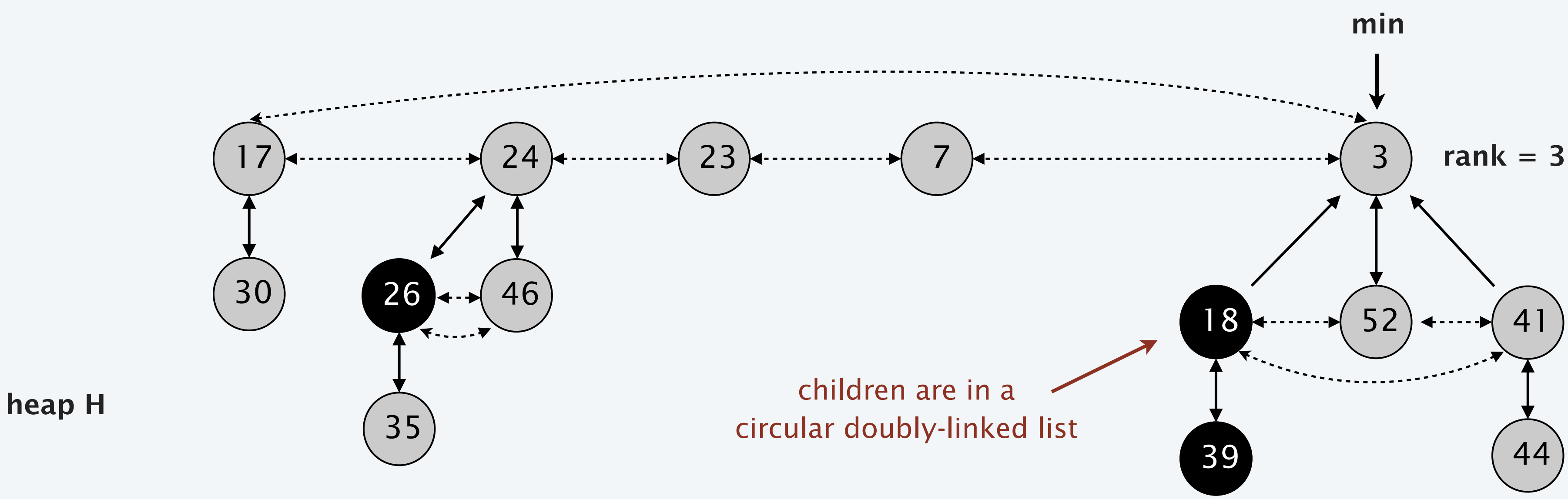
- Store a pointer to the minimum node.
- Maintain tree roots in a circular, doubly-linked list.



Fibonacci heap: representation

Node representation. Each node stores:

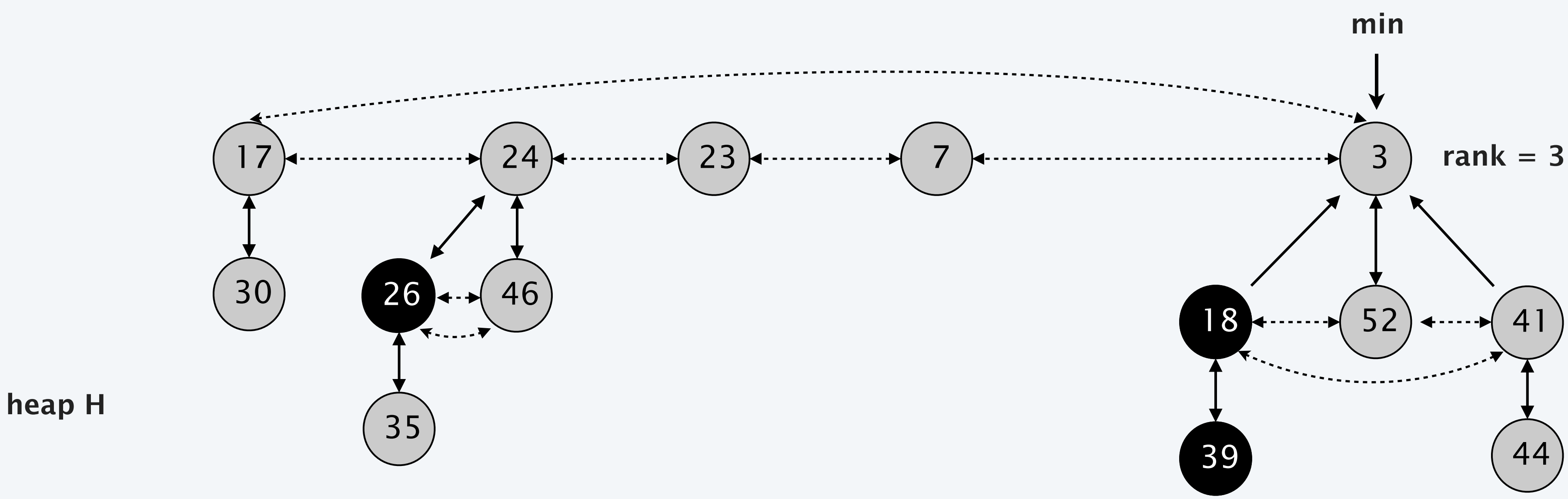
- A pointer to its parent.
- A pointer to any of its children.
- A pointer to its left and right siblings.
- Its rank = number of children.
- Whether it is marked.



Fibonacci heap: representation

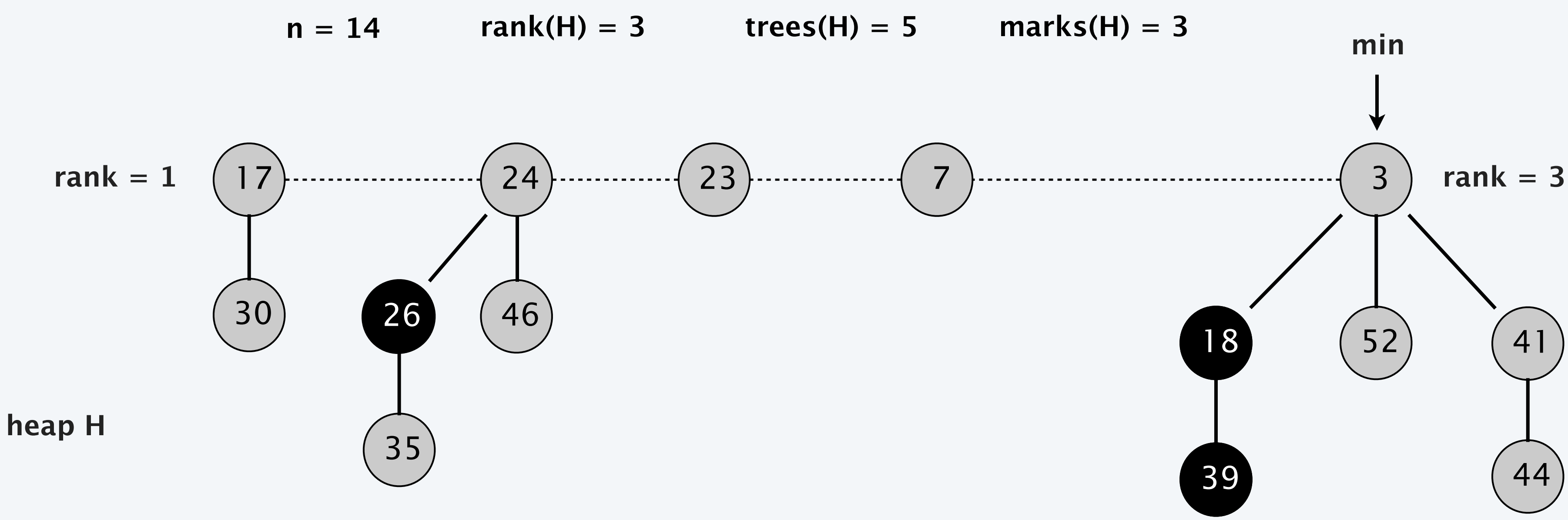
Operations we can do in constant time:

- Find the minimum element.
- Merge two root lists together.
- Determine rank of a root node.
- Add or remove a node from the root list.
- Remove a subtree and merge into root list.
- Link the root of a one tree to root of another tree.



Fibonacci heap: notation

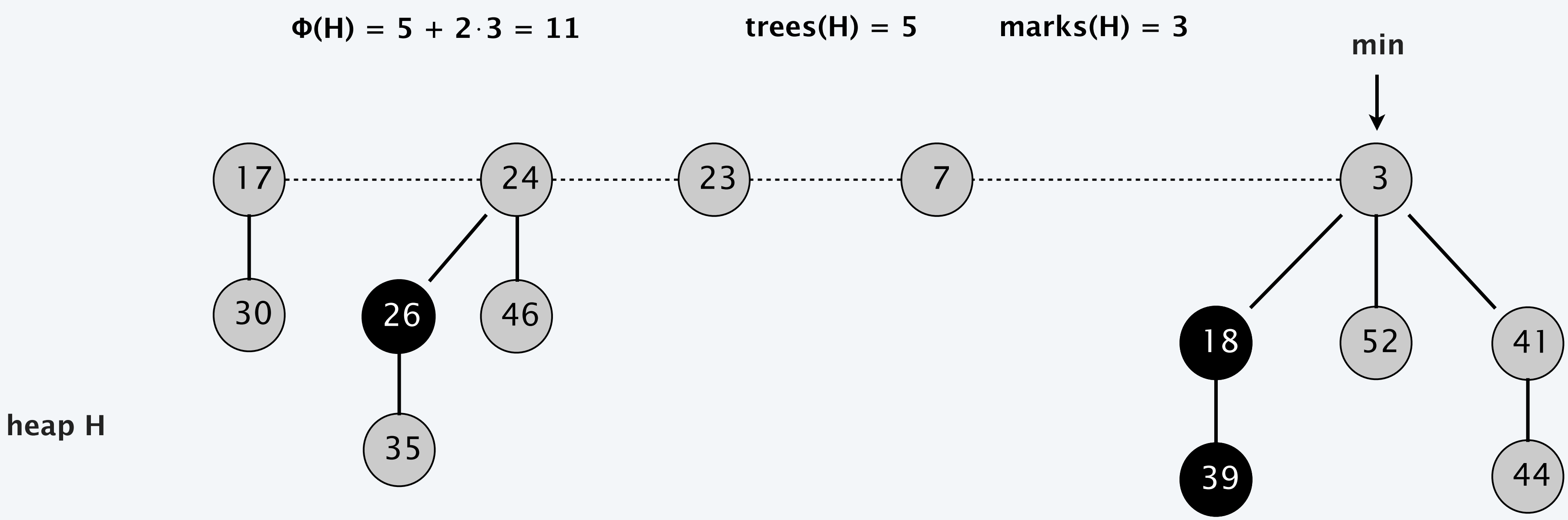
notation	meaning
n	number of nodes
$rank(x)$	number of children of node x
$rank(H)$	max rank of any node in heap H
$trees(H)$	number of trees in heap H
$marks(H)$	number of marked nodes in heap H

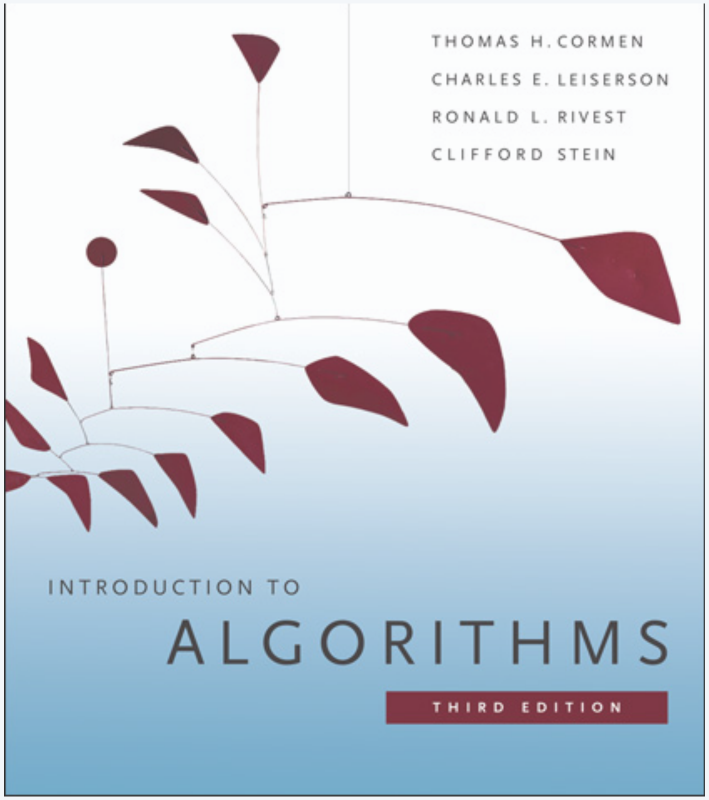


Fibonacci heap: potential function

Potential function.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$





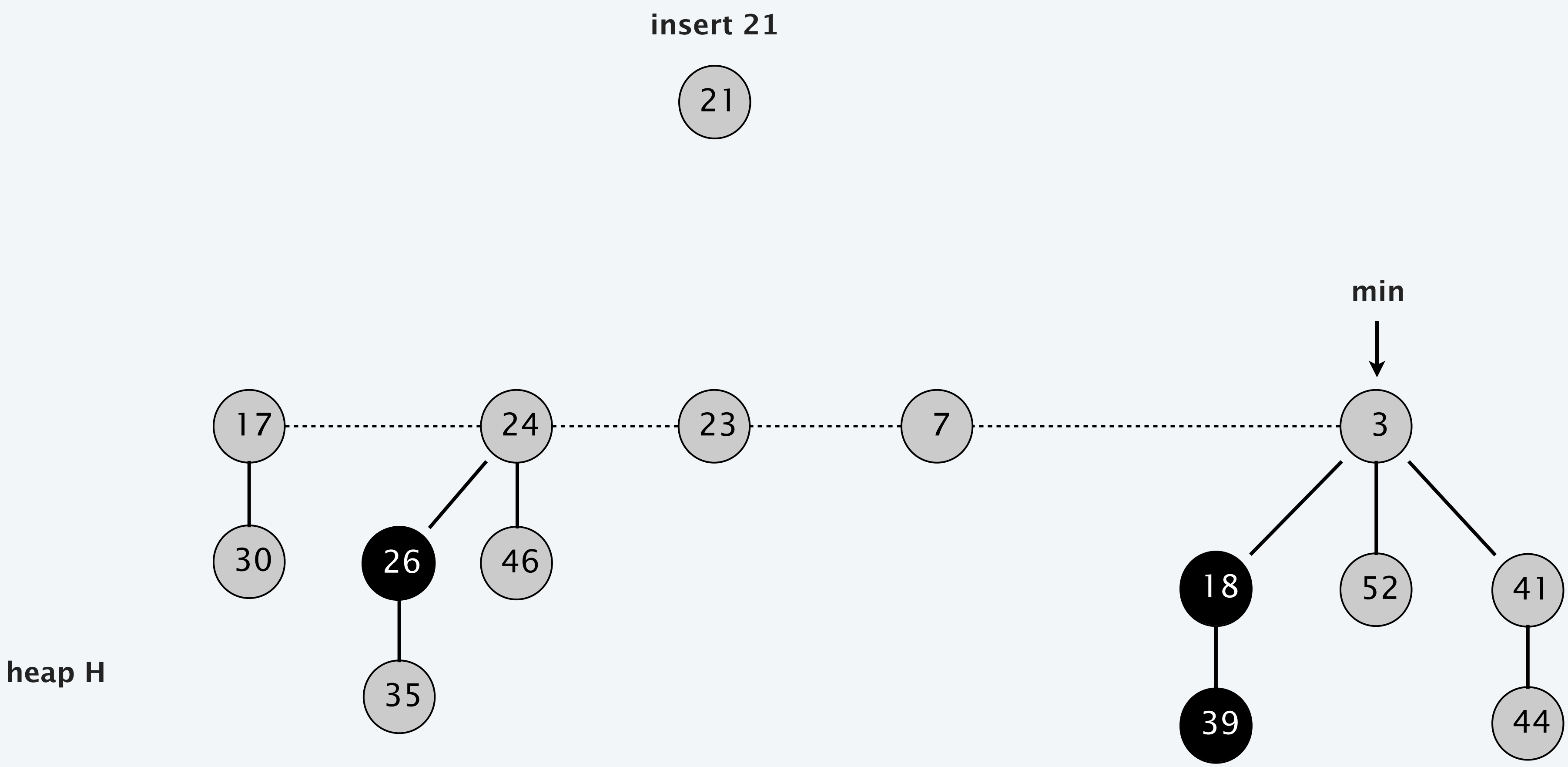
SECTION 19.2

FIBONACCI HEAPS

- *preliminaries*
- *insert*
- *extract the minimum*
- *decrease key*
- *bounding the rank*
- *meld and delete*

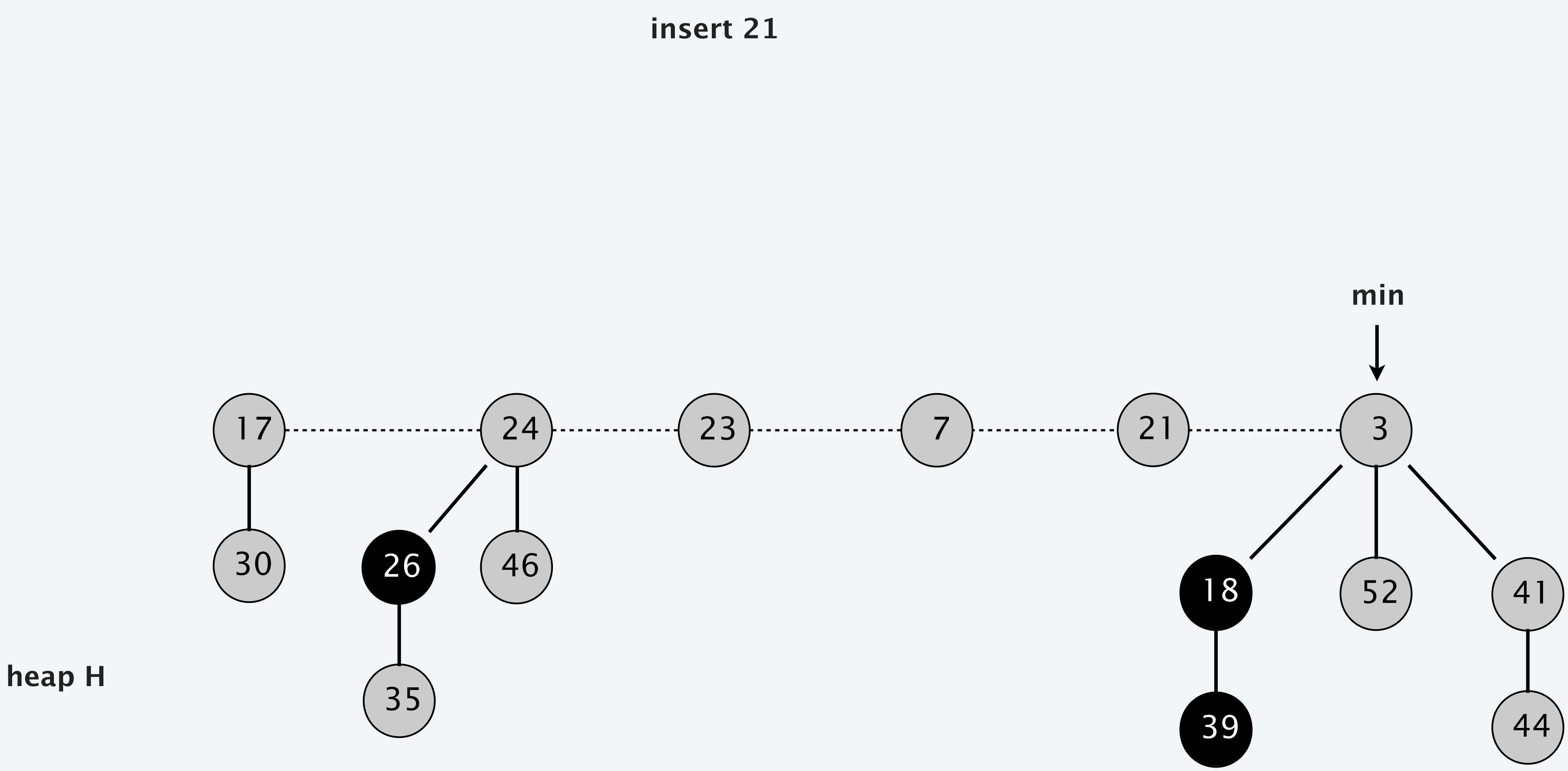
Fibonacci heap: insert

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



Fibonacci heap: insert

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



Fibonacci heap: insert analysis

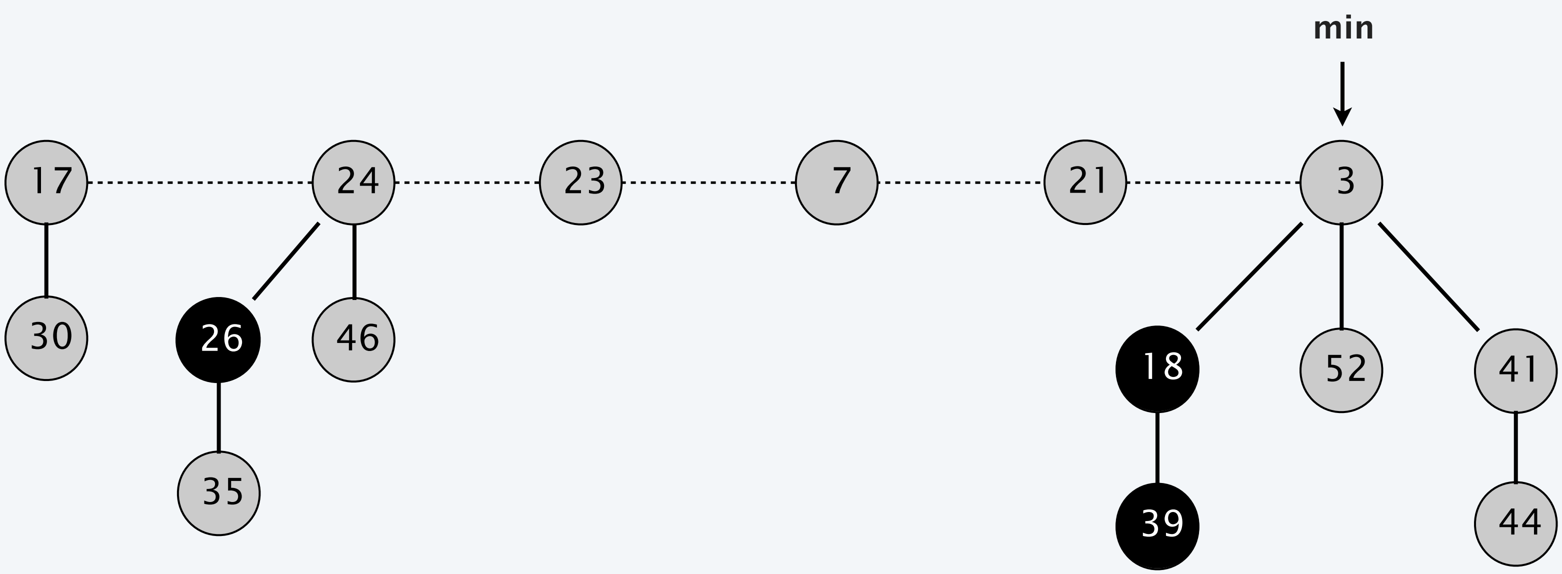
Actual cost. $c_i = O(1)$.

Change in potential. $\Delta\Phi = \Phi(H_i) - \Phi(H_{i-1}) = +1$.

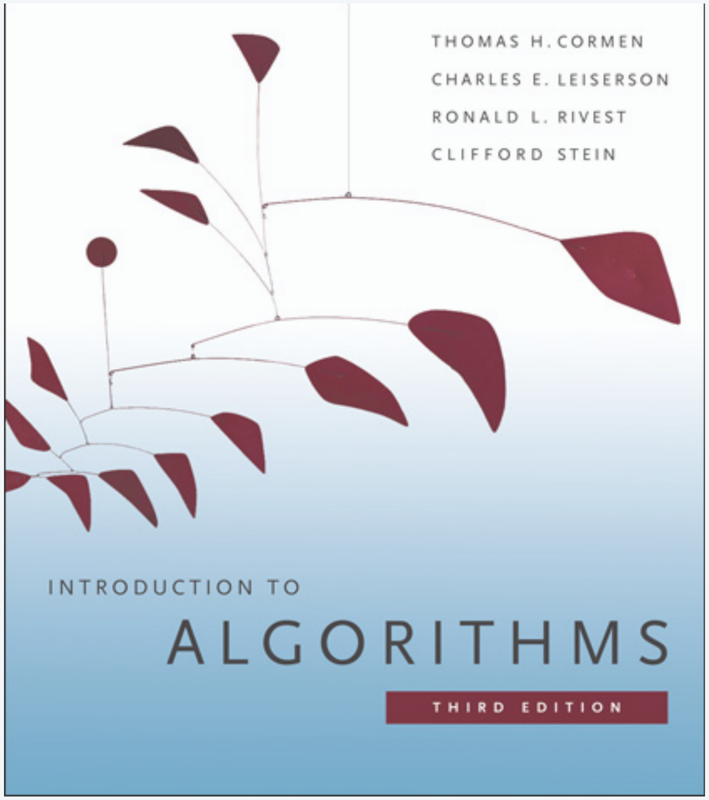
one more tree;
no change in marks

Amortized cost. $\hat{c}_i = c_i + \Delta\Phi = O(1)$.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$



heap H



SECTION 19.2

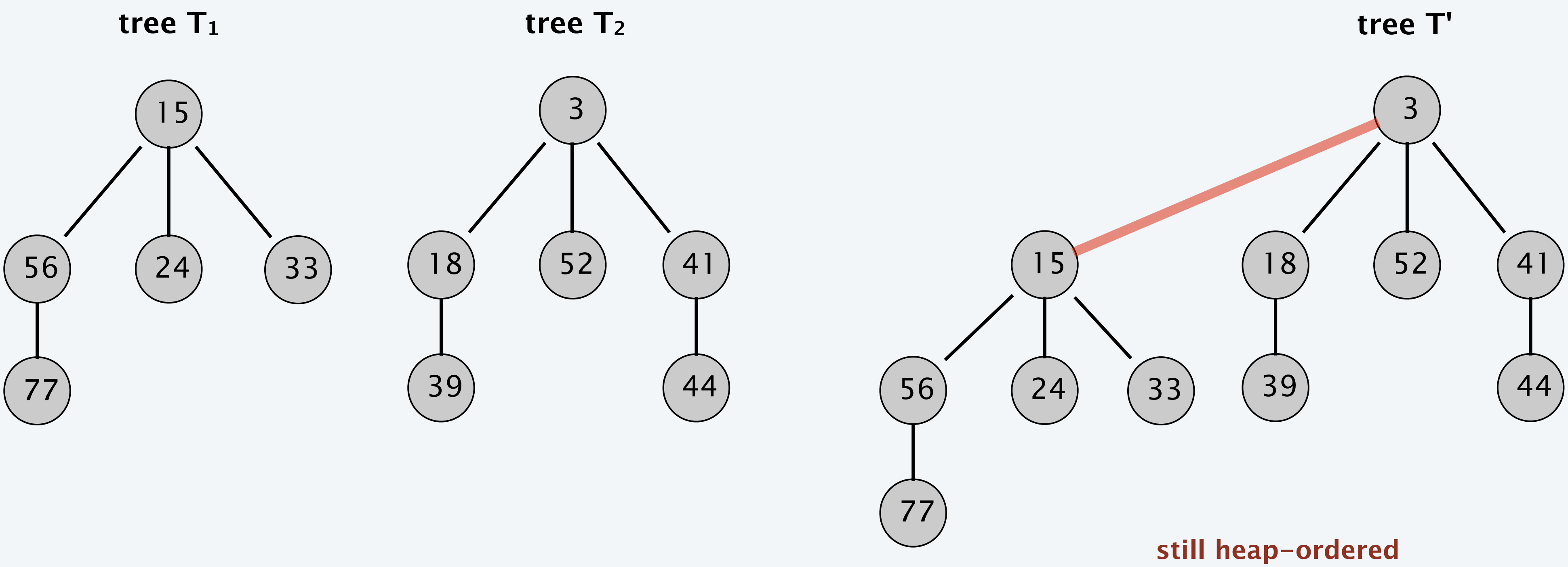
FIBONACCI HEAPS

- *preliminaries*
- *insert*
- *extract the minimum*
- *decrease key*
- *bounding the rank*
- *meld and delete*

Linking operation

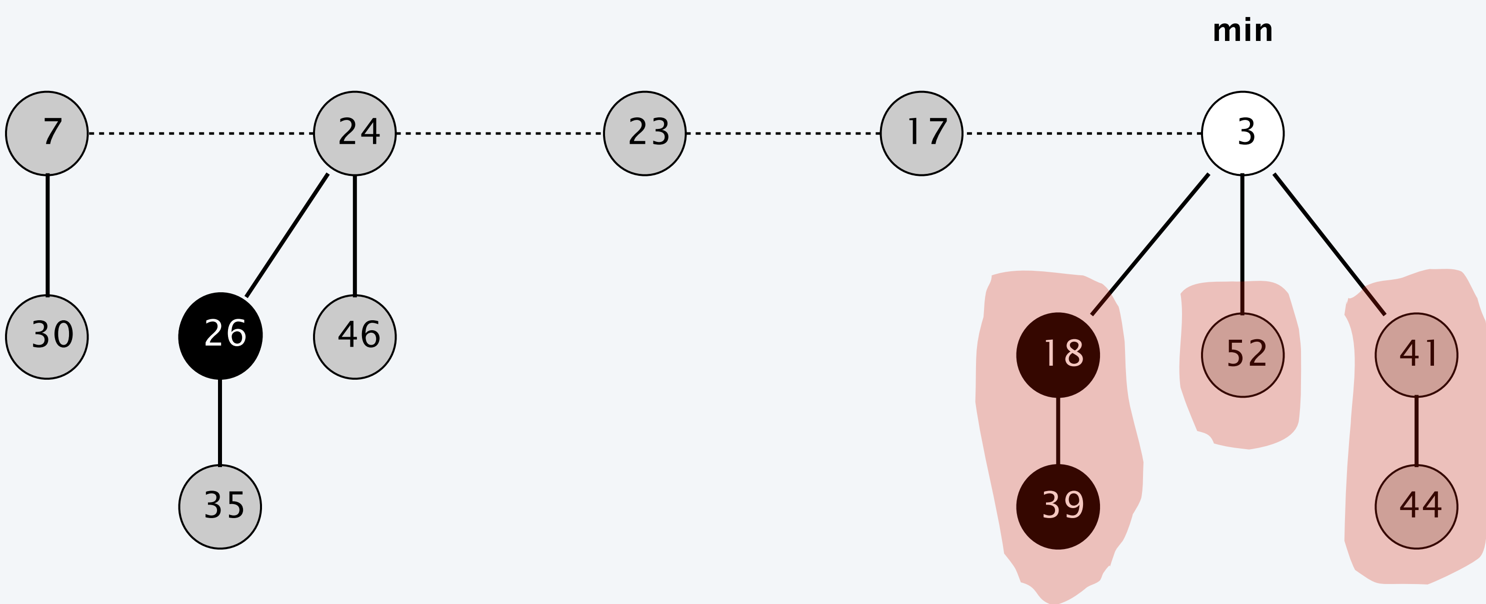
Useful primitive. Combine two trees T_1 and T_2 of rank k.

- Make larger root be a child of smaller root.
- Resulting tree T' has rank $k + 1$.



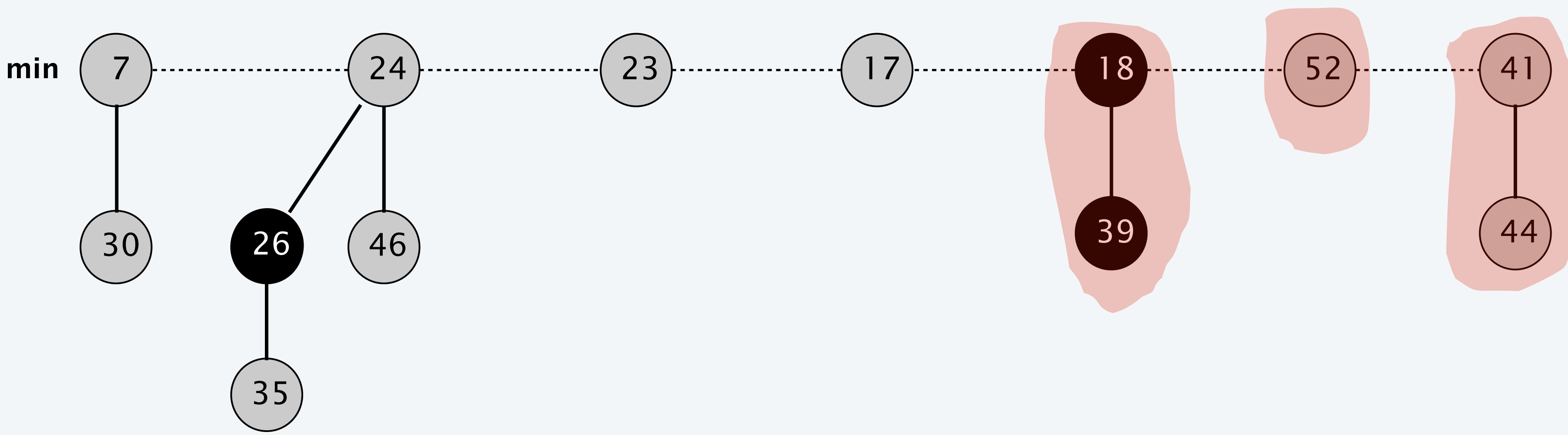
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



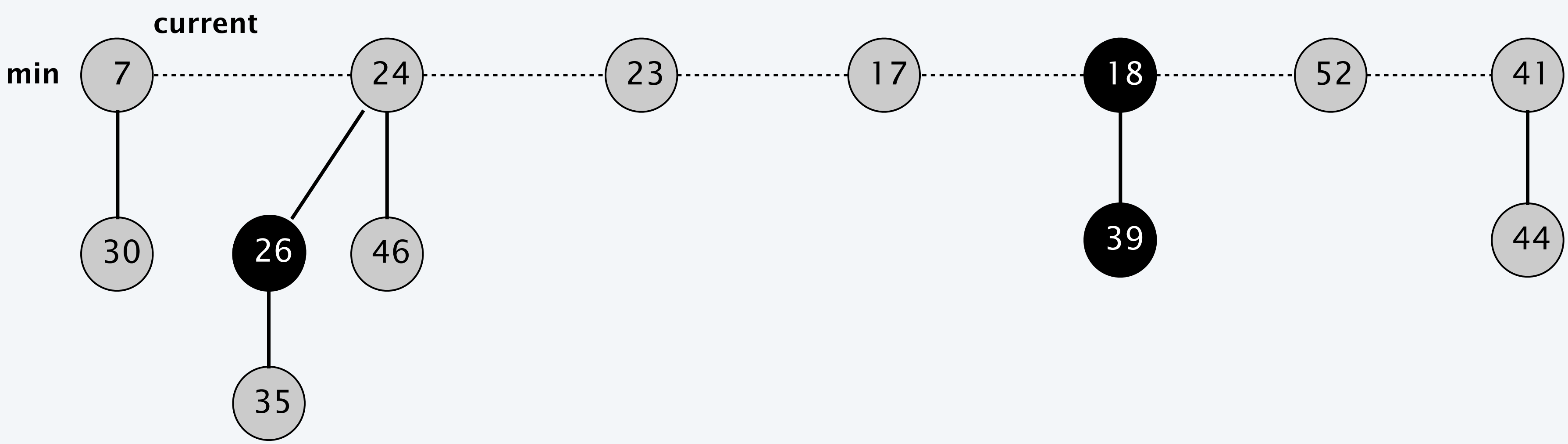
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



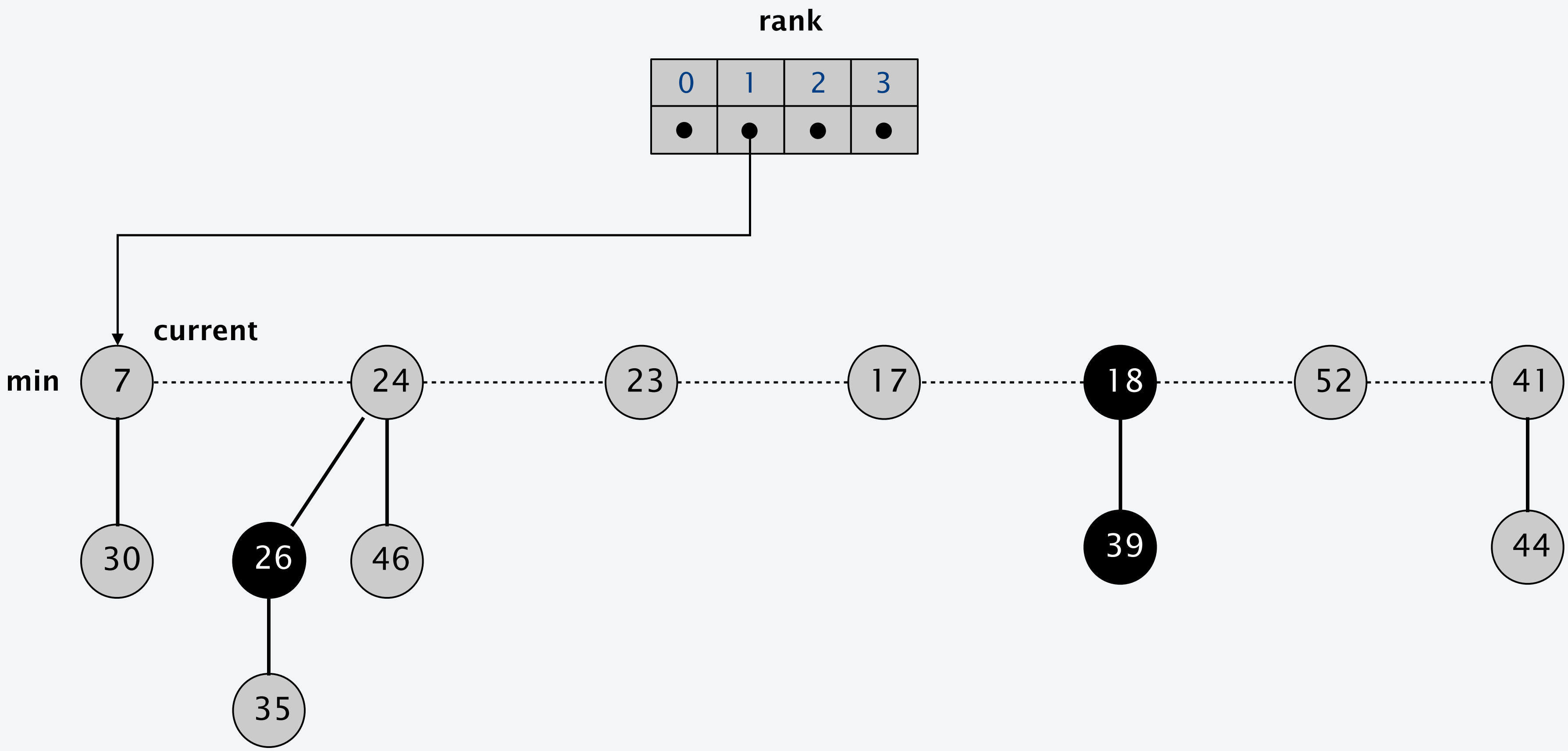
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



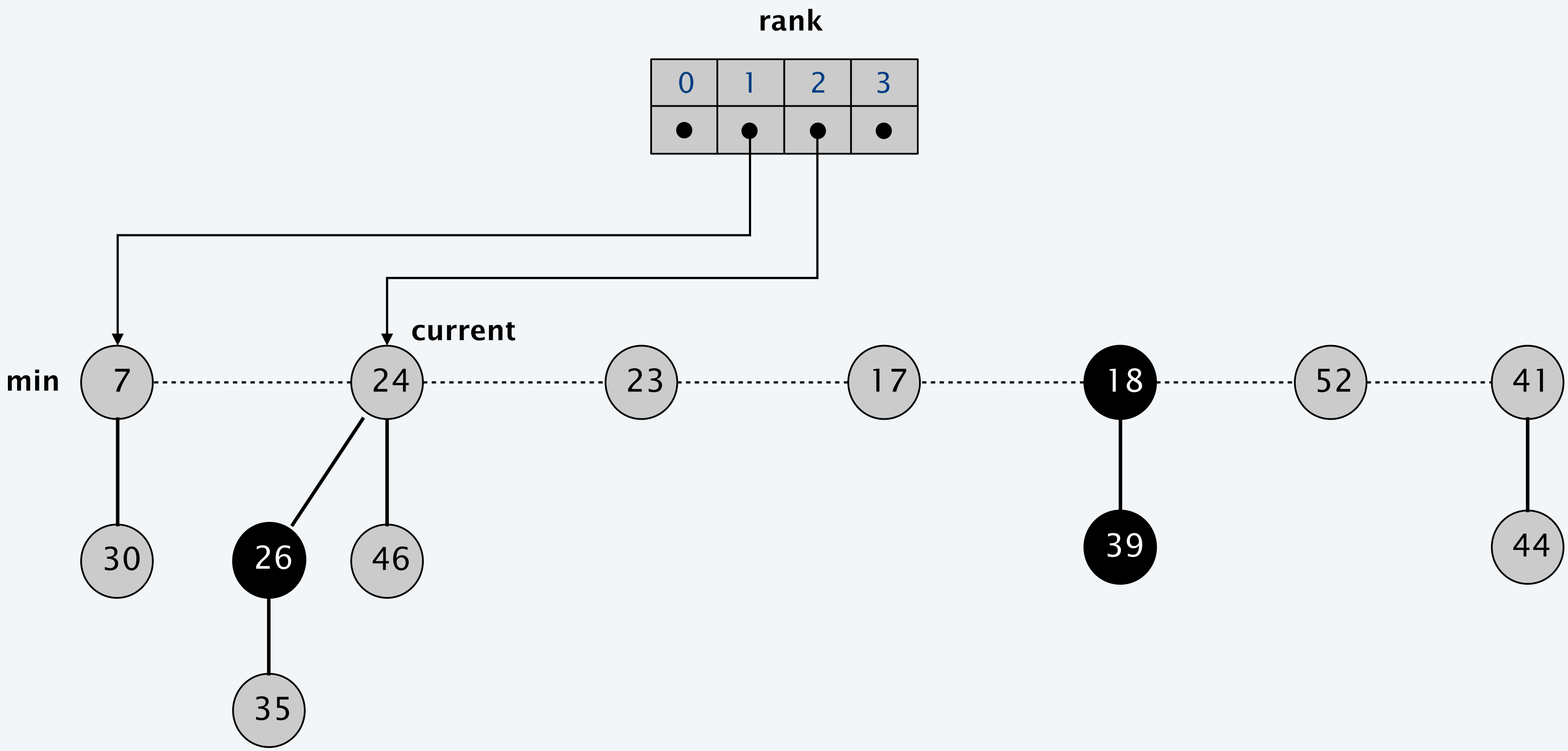
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



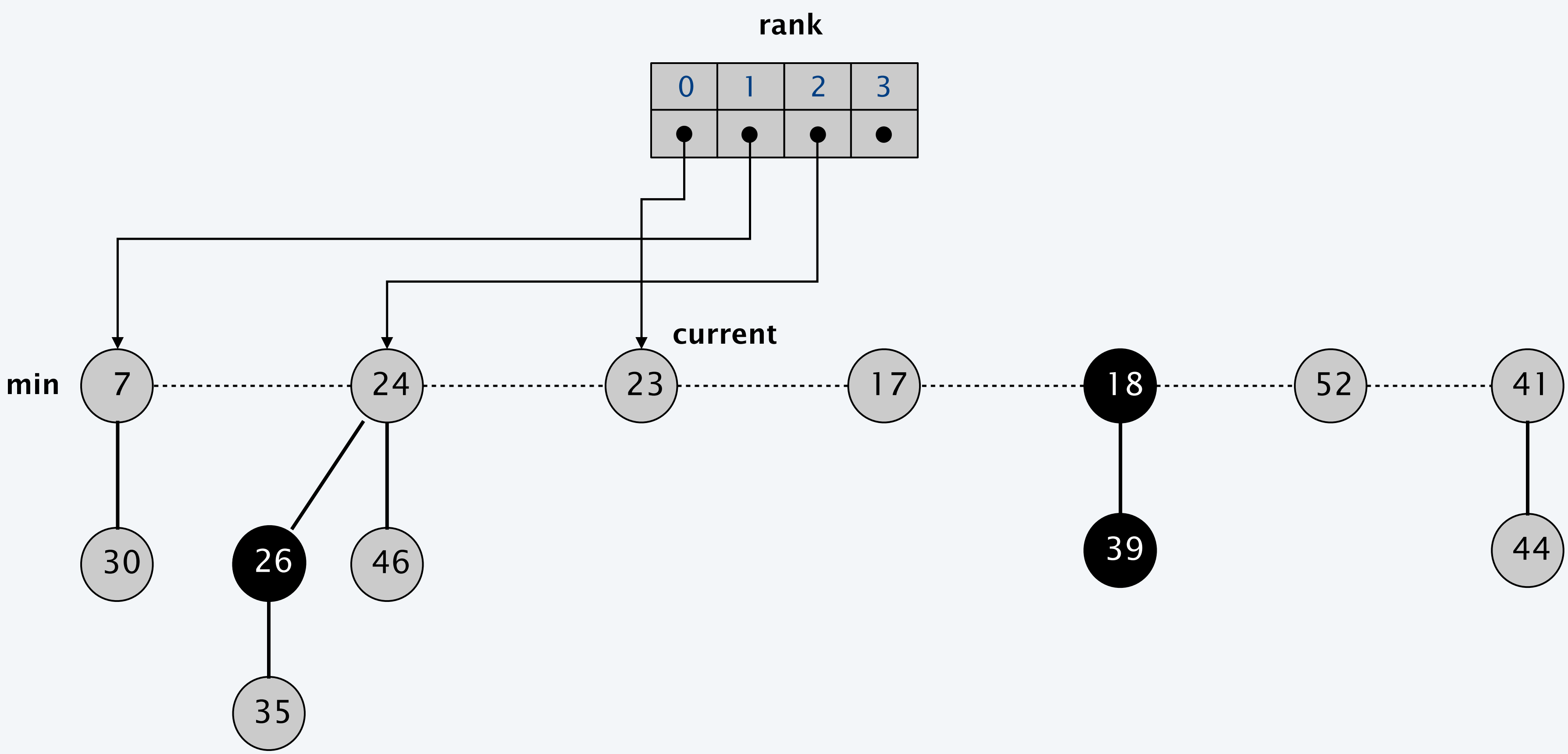
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



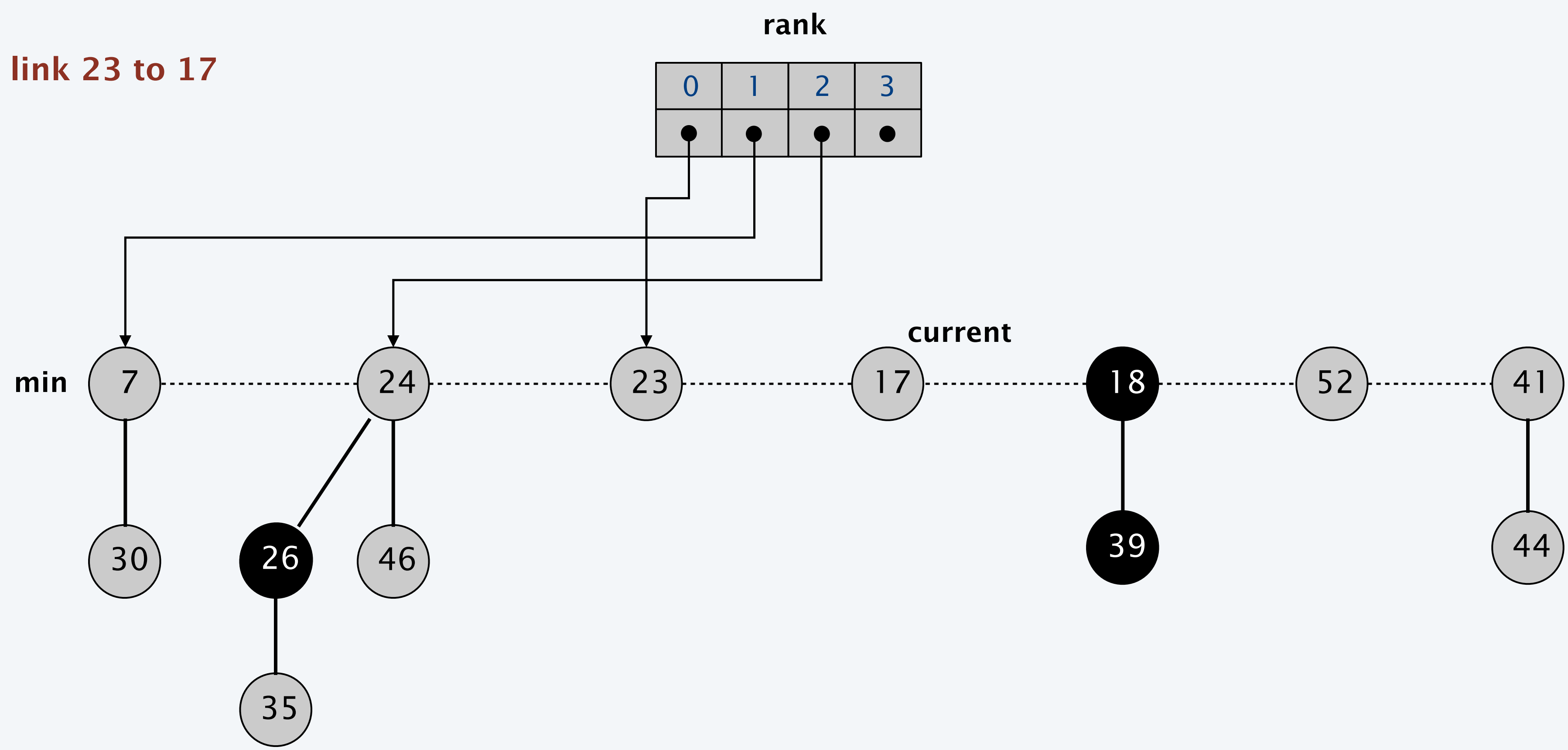
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



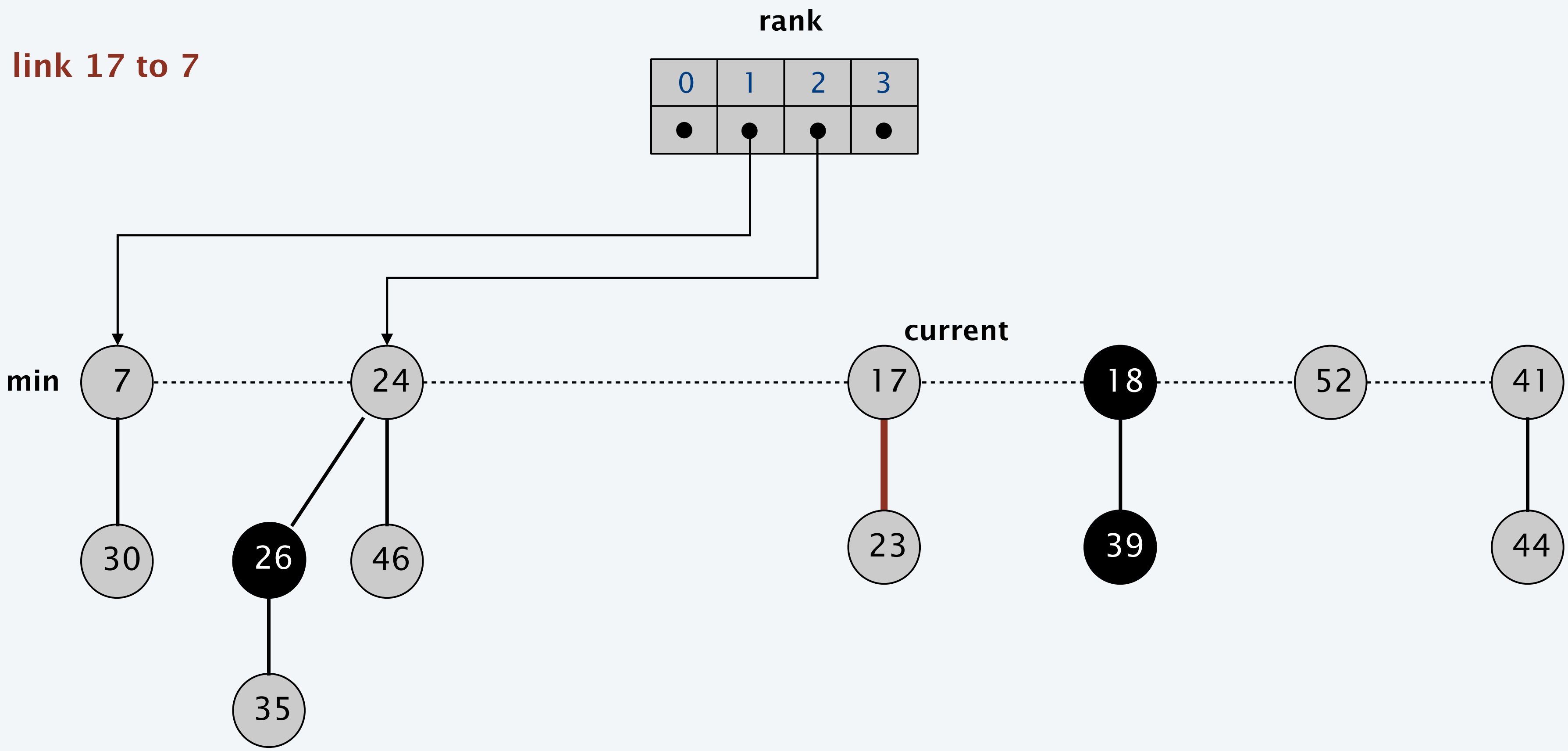
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



Fibonacci heap: extract the minimum

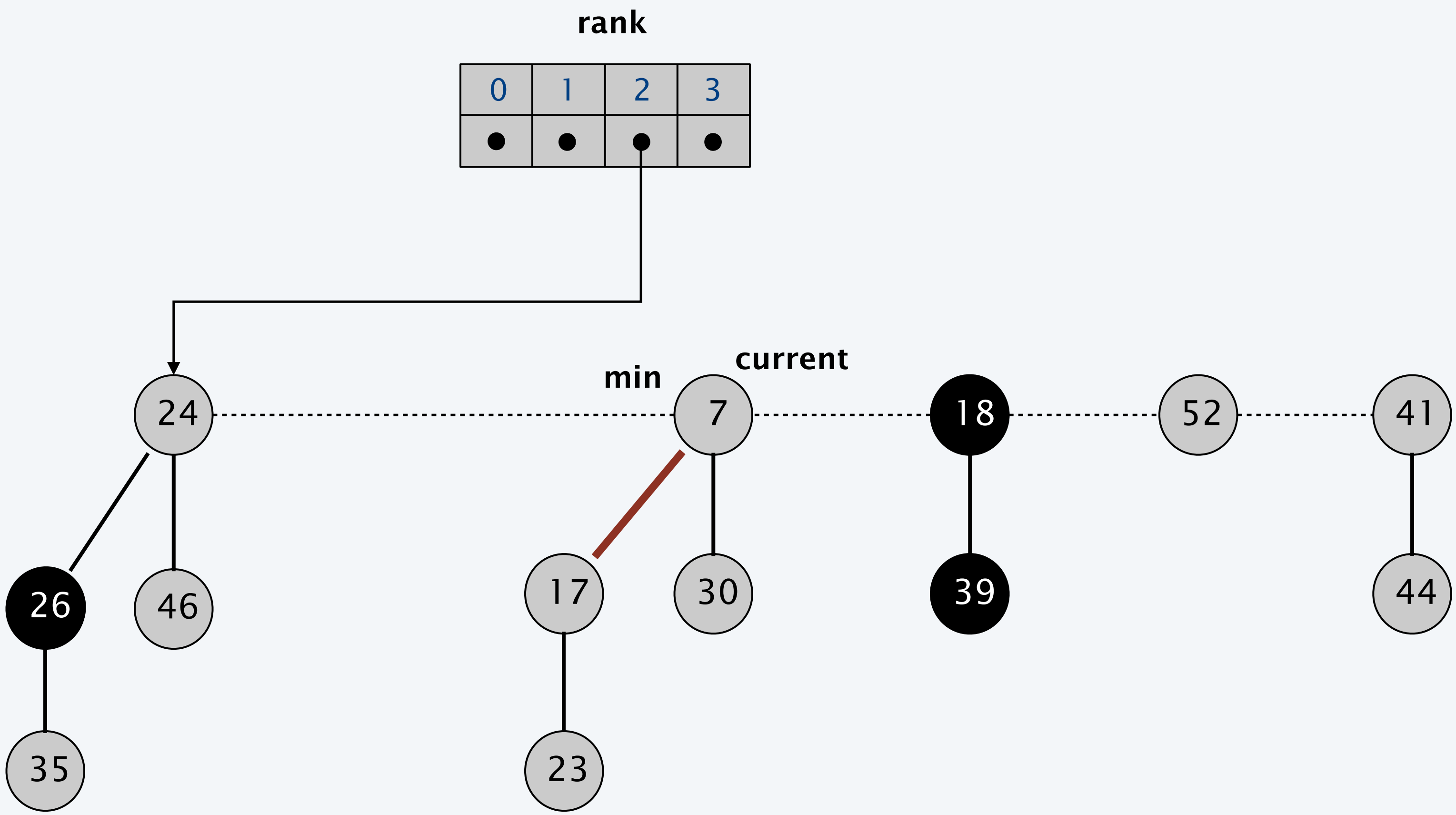
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



Fibonacci heap: extract the minimum

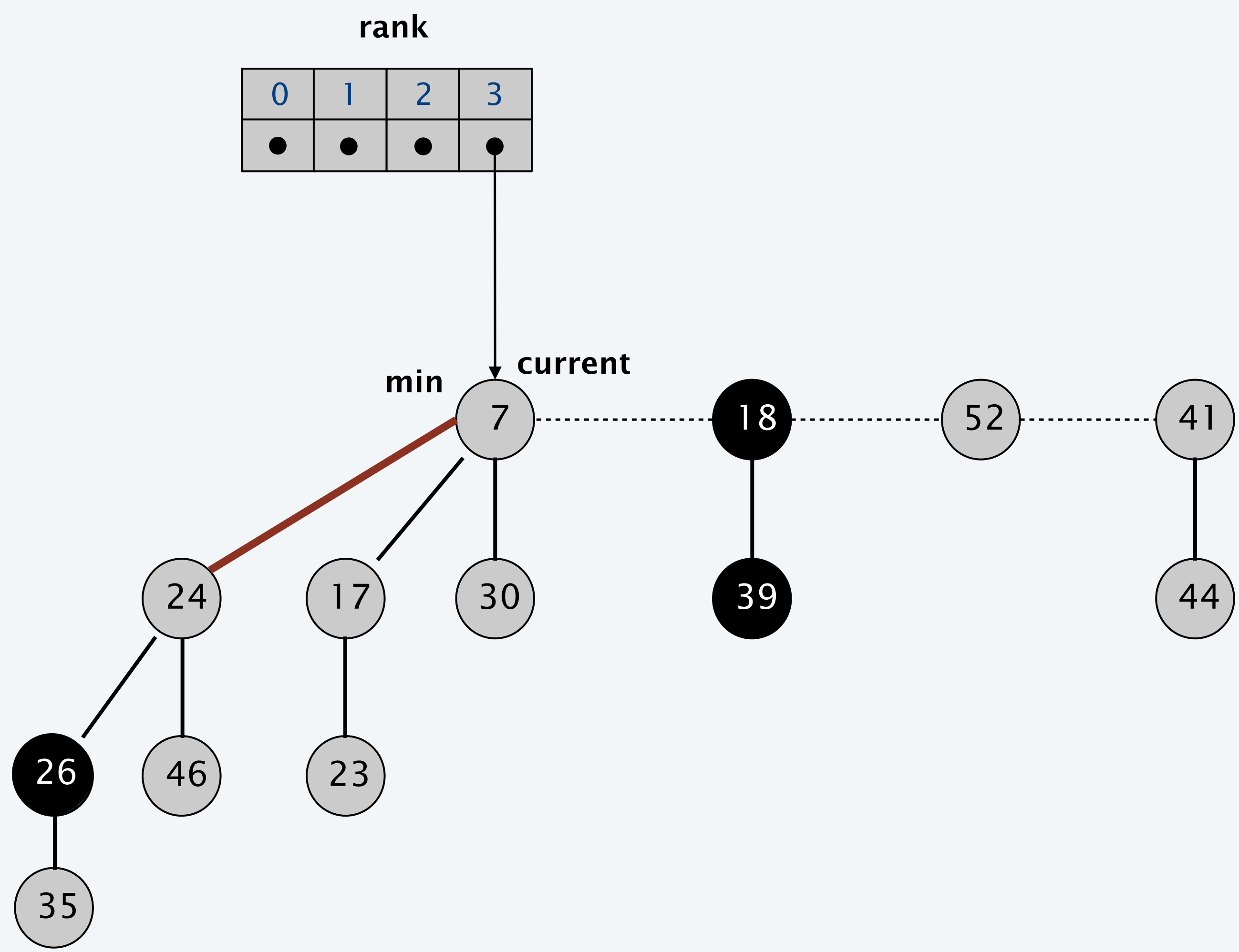
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.

link 24 to 7



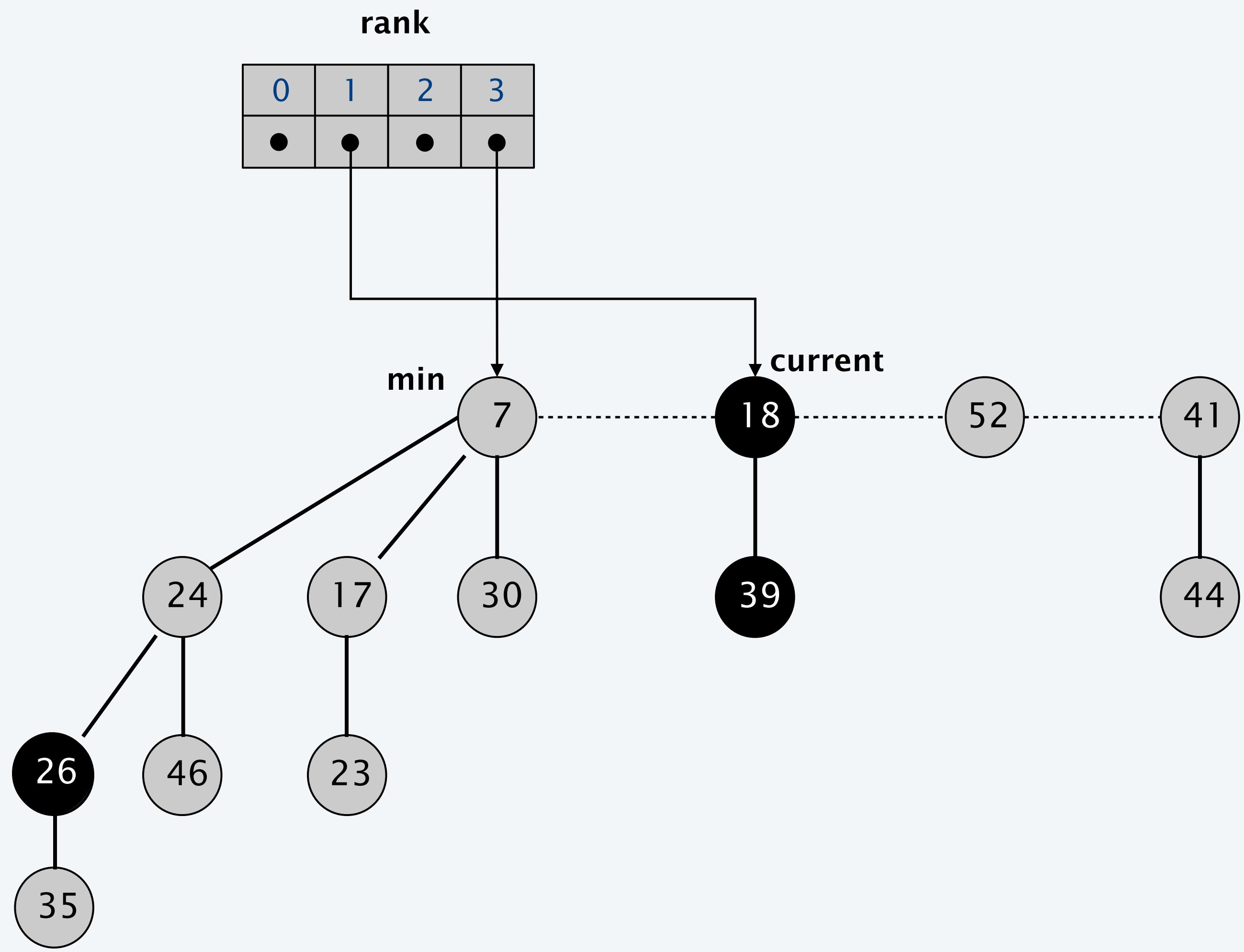
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



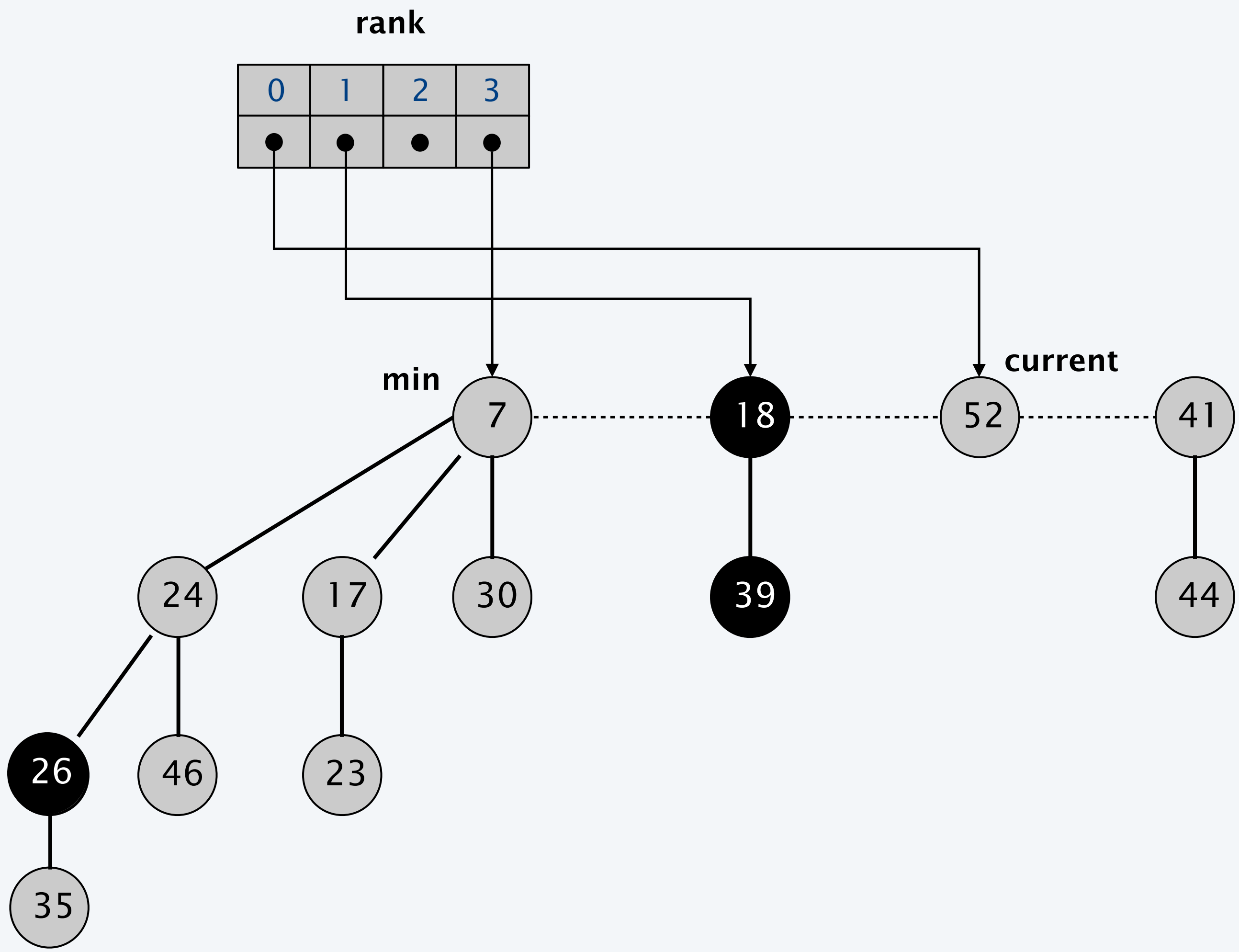
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



Fibonacci heap: extract the minimum

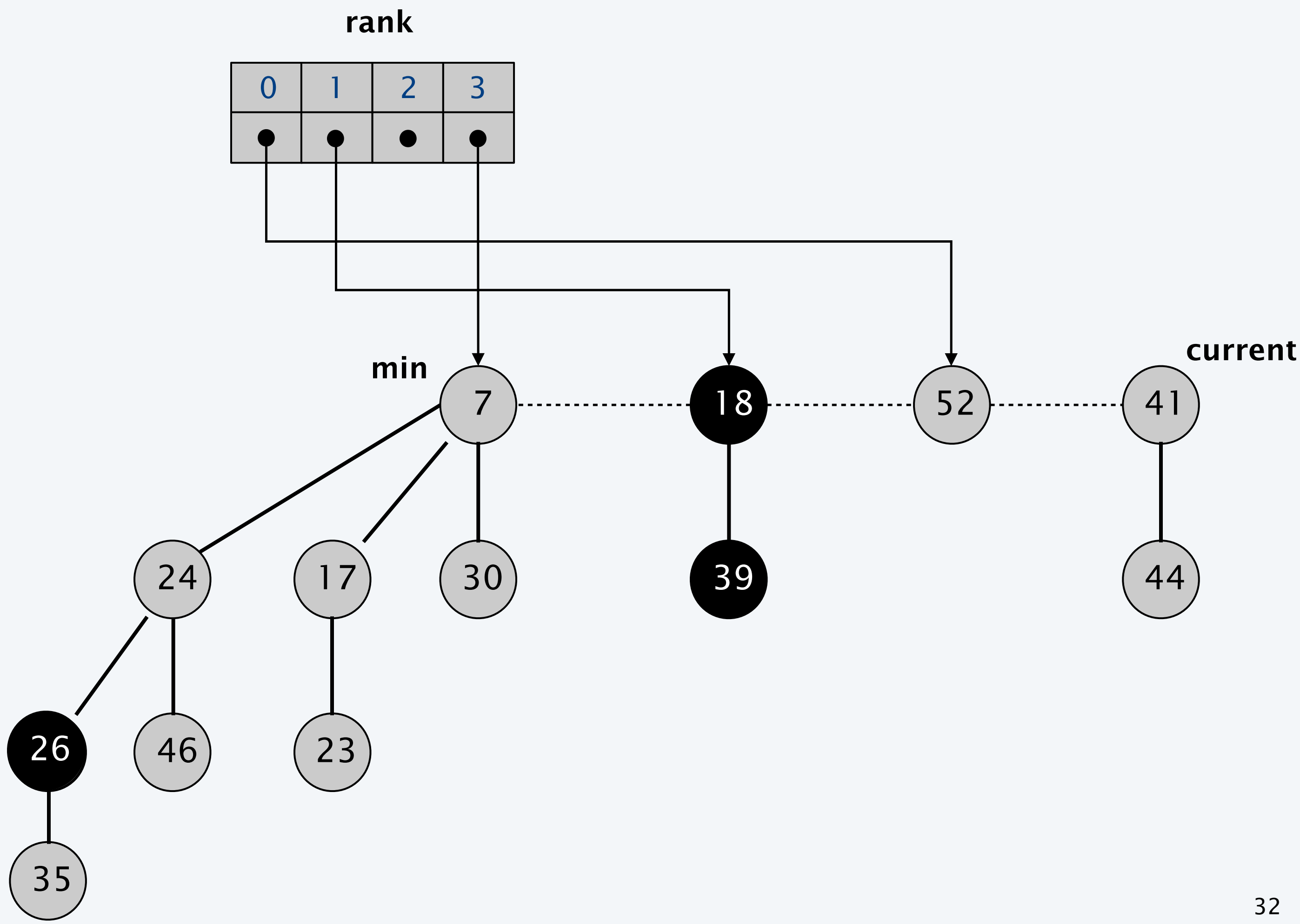
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



Fibonacci heap: extract the minimum

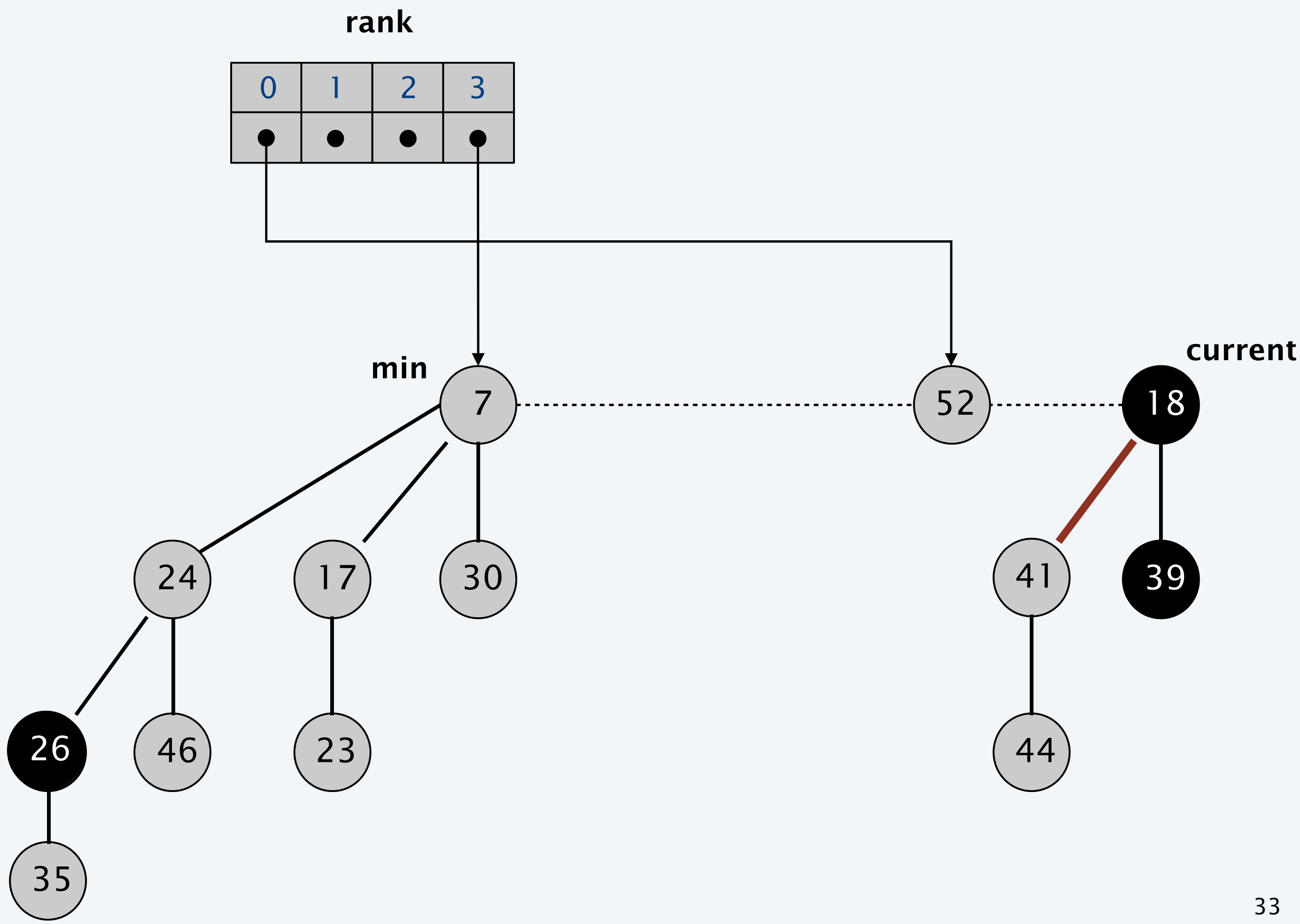
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.

link 41 to 18



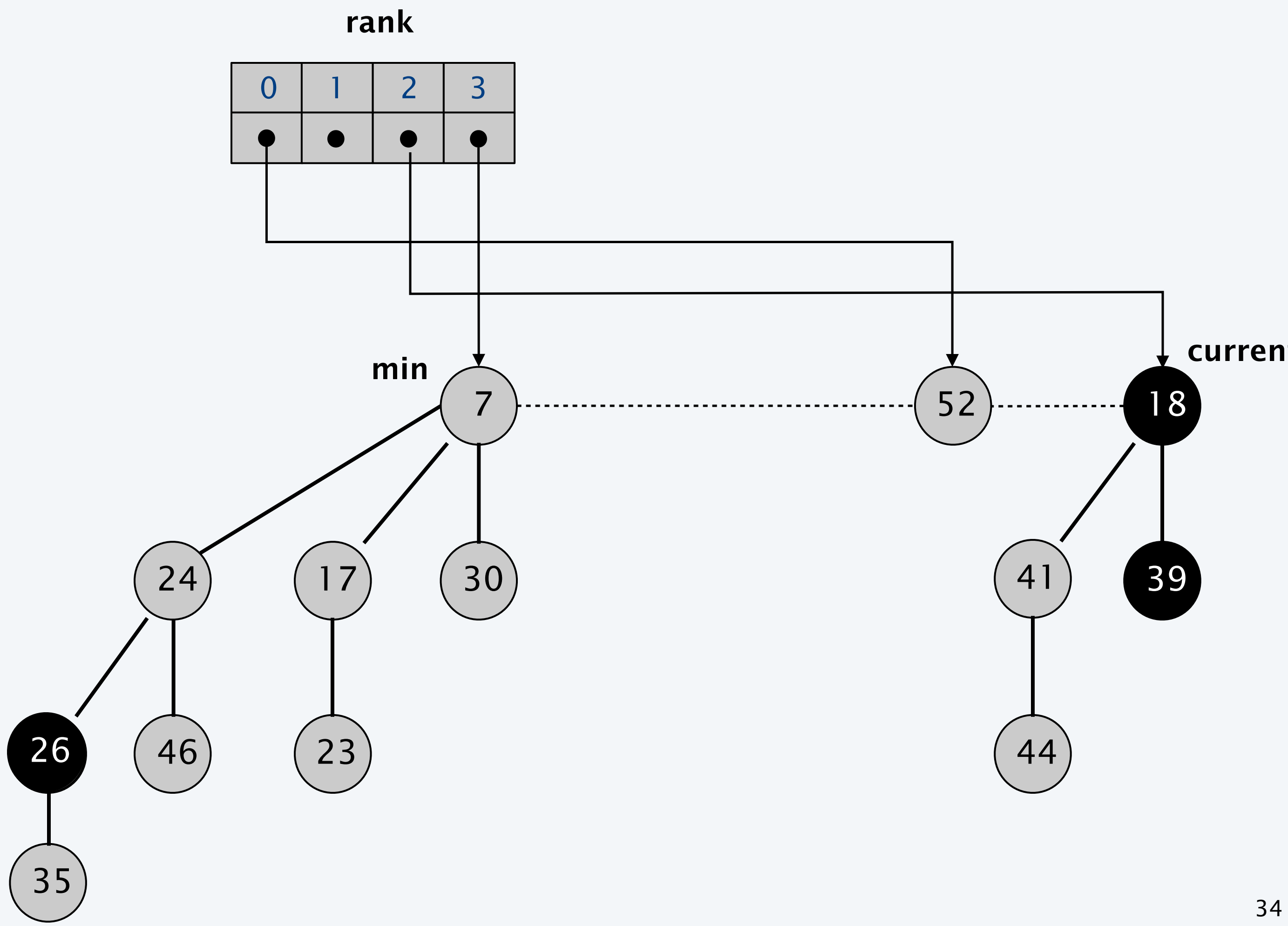
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



Fibonacci heap: extract the minimum

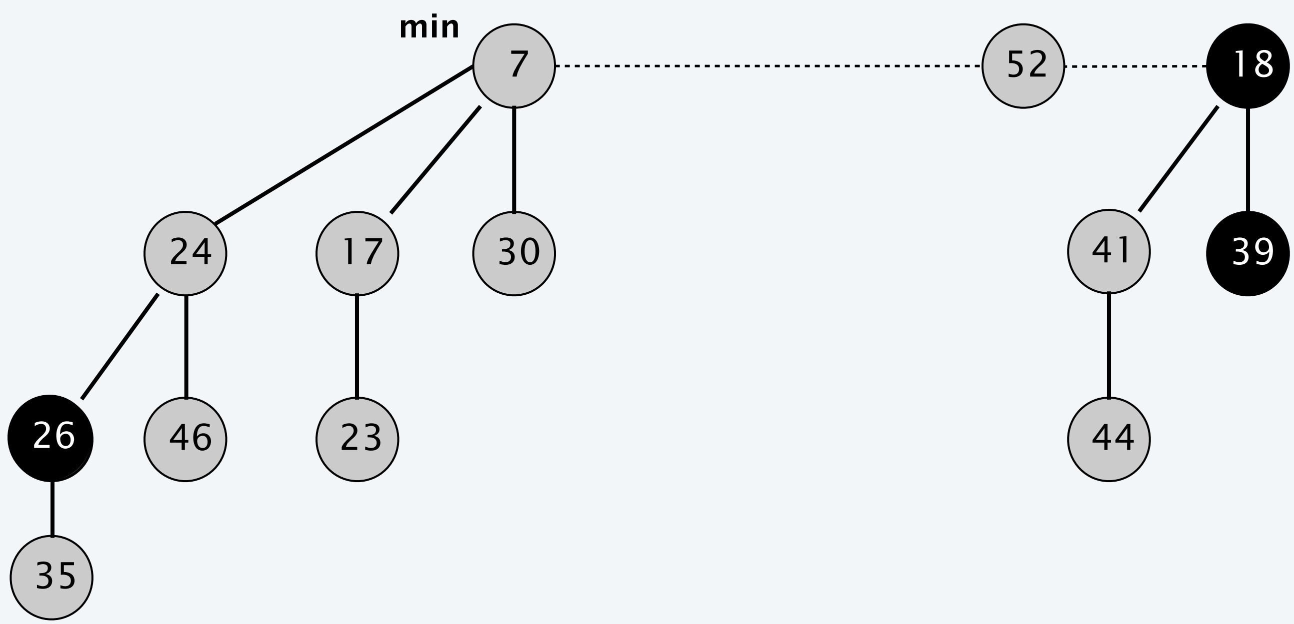
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.

stop (no two trees have same rank)



Fibonacci heap: extract the minimum analysis

Actual cost. $c_i = O(rank(H)) + O(trees(H))$.

- $O(rank(H))$ to meld min's children into root list. ← $\leq$ rank(H) children
- $O(rank(H)) + O(trees(H))$ to update min. ← $\leq$ rank(H) + trees(H) − 1 root nodes
- $O(rank(H)) + O(trees(H))$ to consolidate trees. ← number of roots decreases by 1 after each linking operation

Change in potential. $\Delta\Phi \leq rank(H') + 1 - trees(H)$.

- No new nodes become marked.
- $trees(H') \leq rank(H') + 1$. ← no two trees have same rank after consolidation

Amortized cost. $O(\log n)$.

- $\hat{c}_i = c_i + \Delta\Phi = O(rank(H)) + O(rank(H'))$.
- The rank of a Fibonacci heap with n elements is $O(\log n)$.

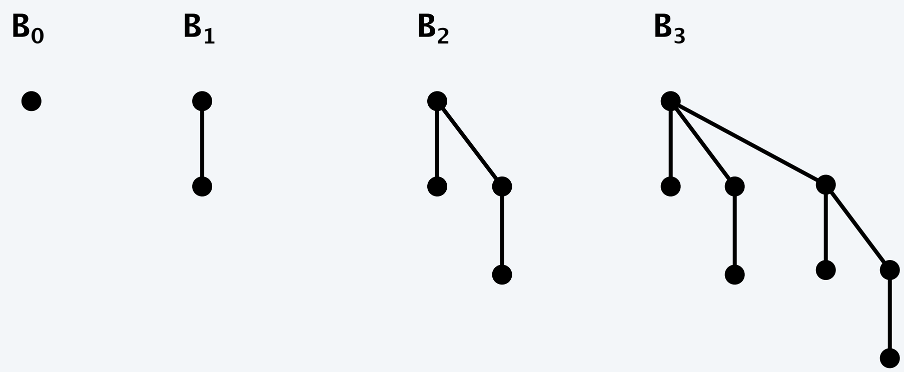
Fibonacci lemma (stay tuned)

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Fibonacci heap vs. binomial heaps

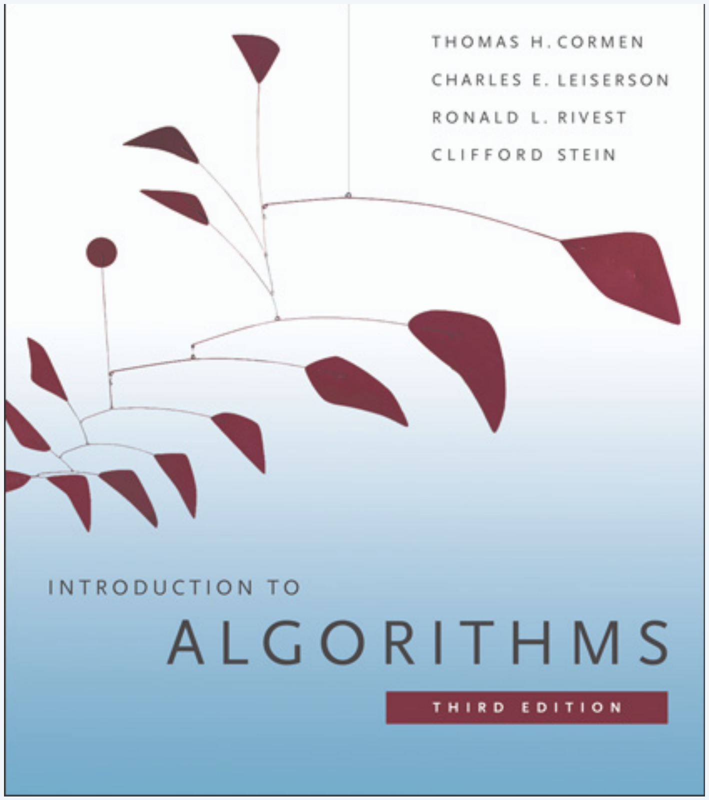
Observation. If only INSERT and EXTRACT-MIN operations, then all trees are binomial trees.

we link only trees of equal rank



Binomial heap property. This implies $rank(H) \leq \log_2 n$.

Fibonacci heap property. Our DECREASE-KEY implementation will not preserve this property, but we will implement it in such a way that $rank(H) \leq \log_\phi n$.



SECTION 19.3

FIBONACCI HEAPS

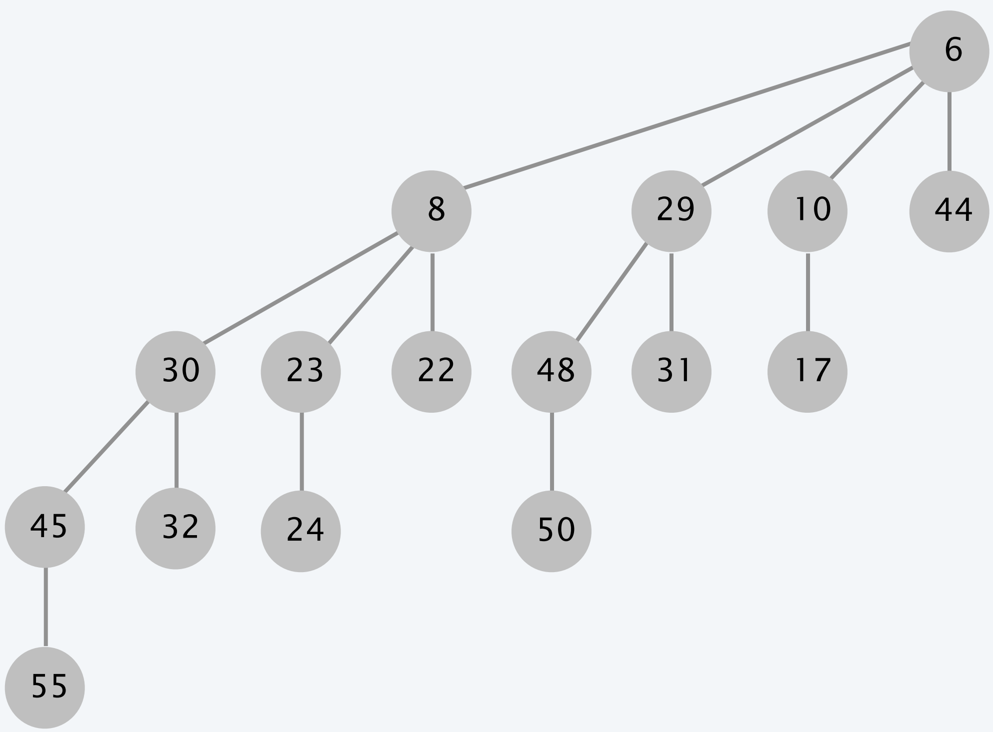
- *preliminaries*
- *insert*
- *extract the minimum*
- *decrease key*
- *bounding the rank*
- *meld and delete*

Fibonacci heap: decrease key

Intuition for deceasing the key of node x.

- If heap-order is not violated, decrease the key of x.
- Otherwise, cut tree rooted at x and meld into root list.

decrease-key of x from 30 to 7

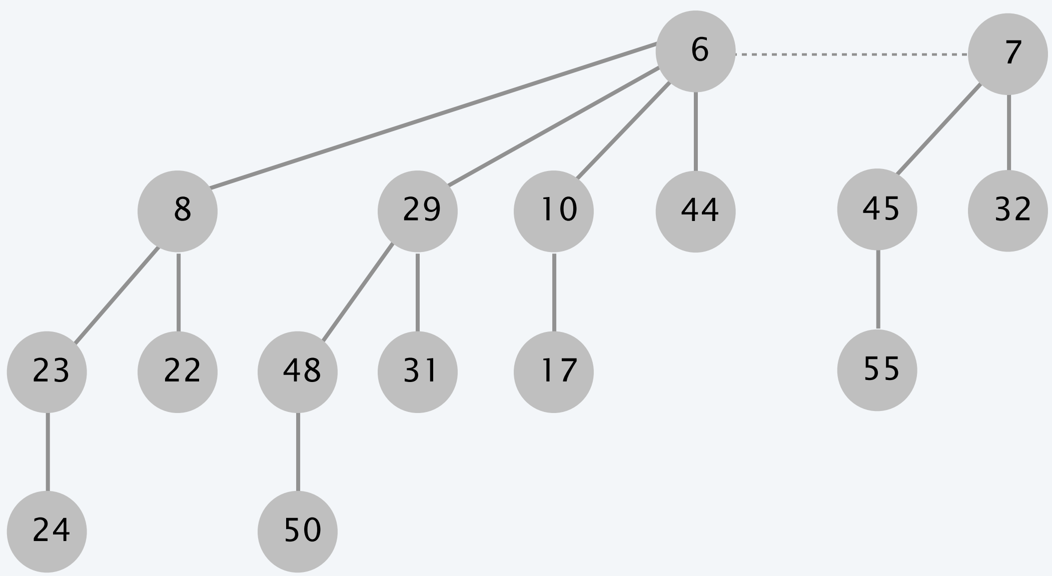


Fibonacci heap: decrease key

Intuition for deceasing the key of node x.

- If heap-order is not violated, decrease the key of x.
- Otherwise, cut tree rooted at x and meld into root list.

decrease-key of x from 23 to 5

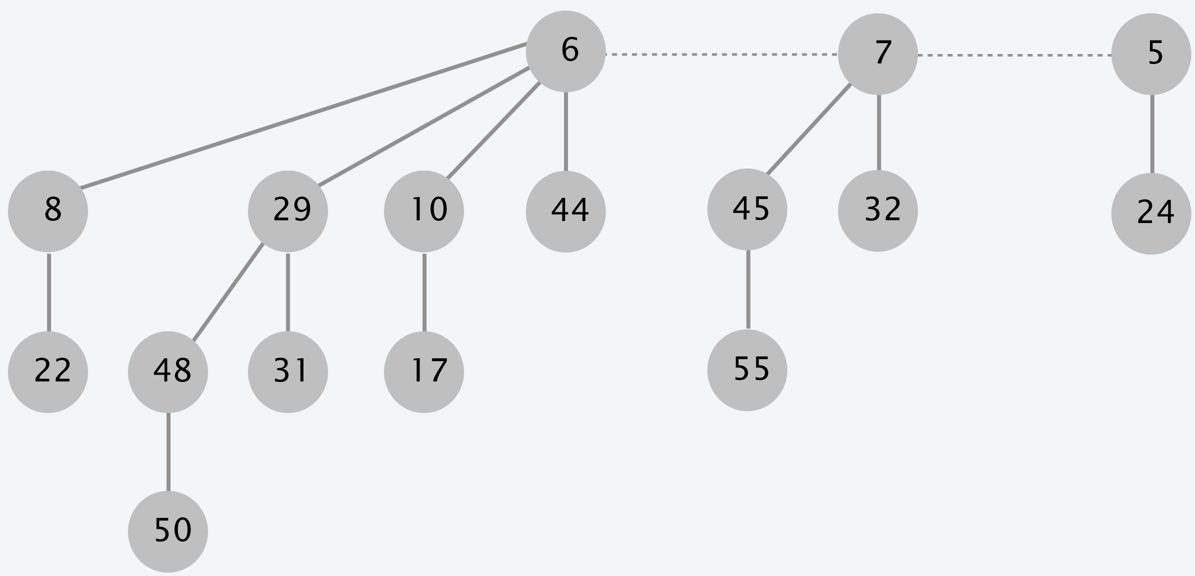


Fibonacci heap: decrease key

Intuition for deceasing the key of node x.

- If heap-order is not violated, decrease the key of x.
- Otherwise, cut tree rooted at x and meld into root list.

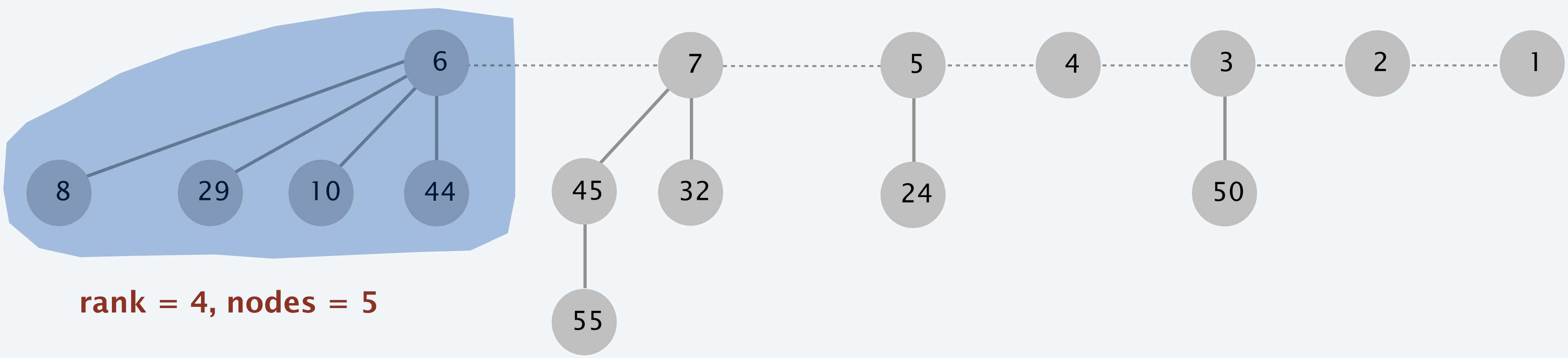
decrease-key of 22 to 4
decrease-key of 48 to 3
decrease-key of 31 to 2
decrease-key of 17 to 1



Fibonacci heap: decrease key

Intuition for deceasing the key of node x.

- If heap-order is not violated, decrease the key of x.
- Otherwise, cut tree rooted at x and meld into root list.
- Problem: number of nodes not exponential in rank.

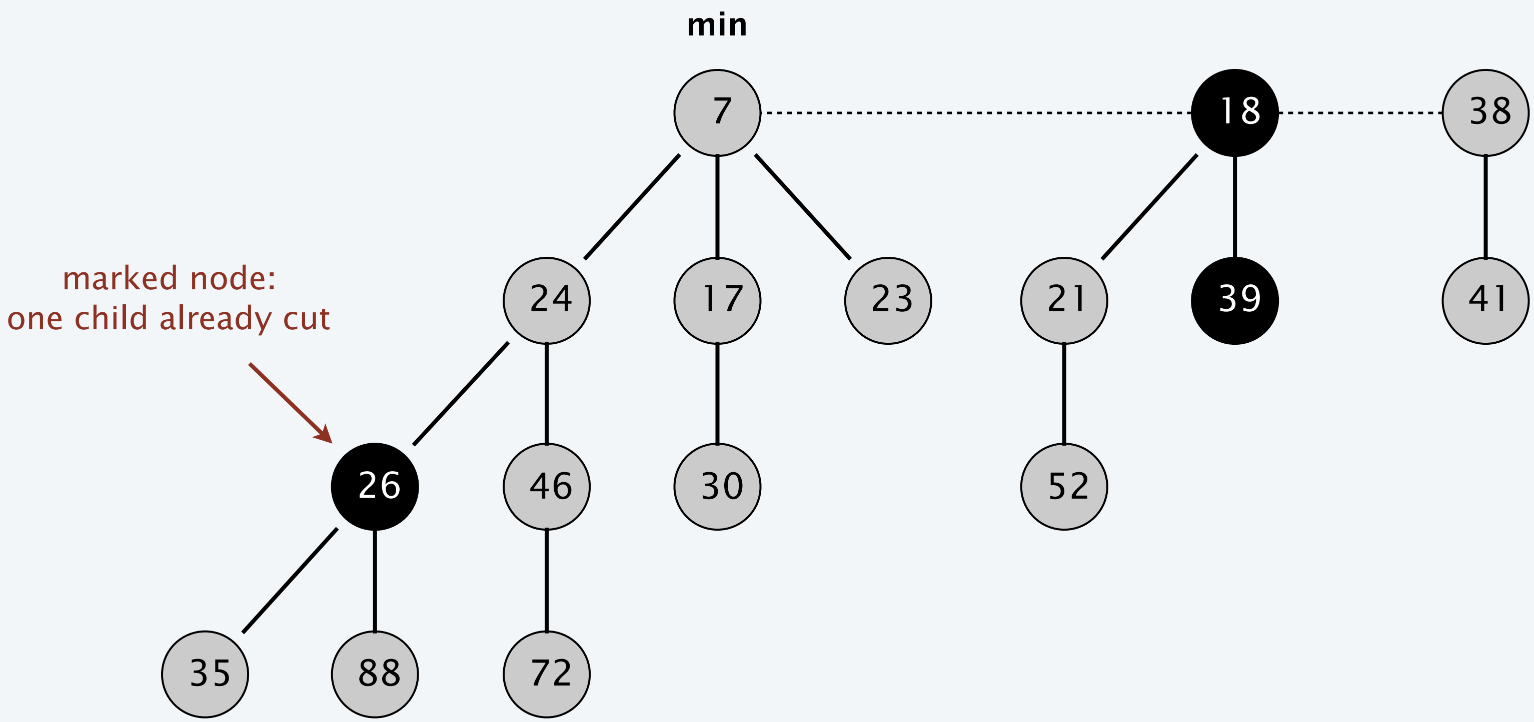


rank = 4, nodes = 5

Fibonacci heap: decrease key

Intuition for deceasing the key of node x.

- If heap-order is not violated, decrease the key of x.
- Otherwise, cut tree rooted at x and meld into root list.
- Solution: as soon as a node has its second child cut, cut it off also and meld into root list (and unmark it).

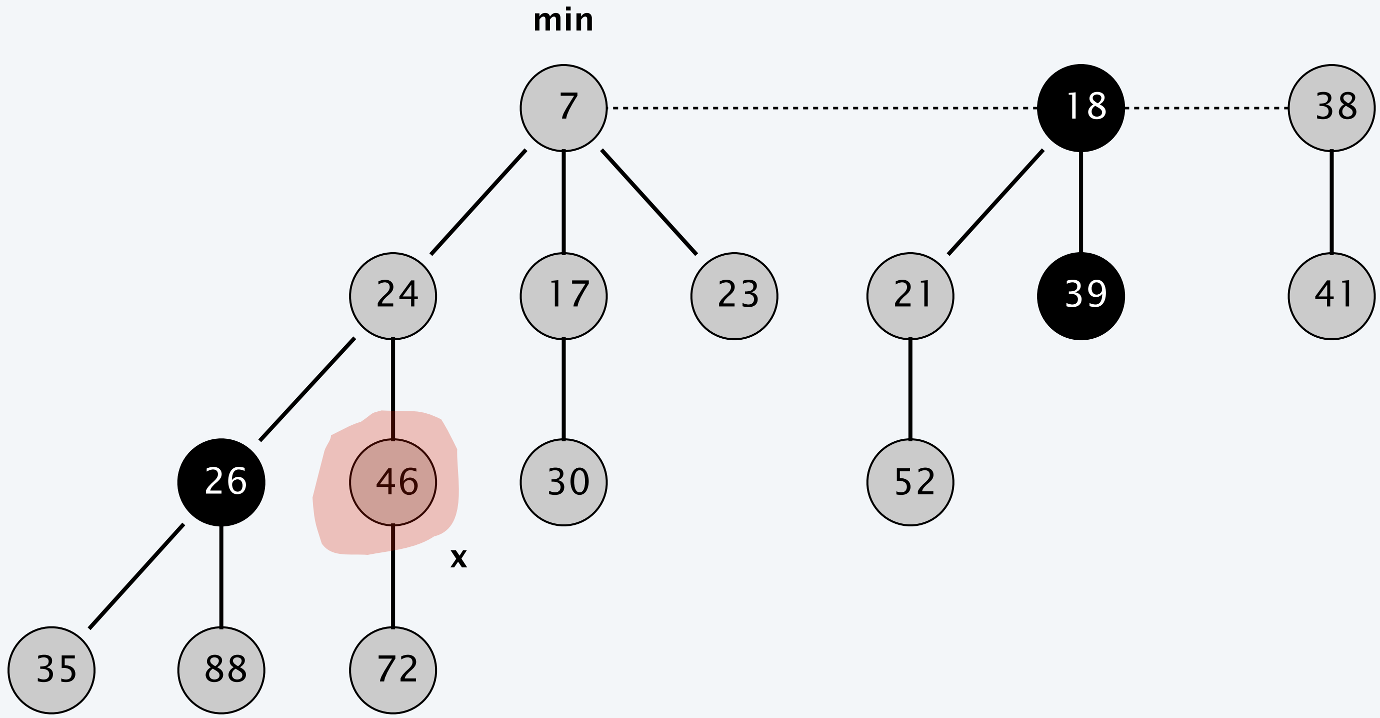


Fibonacci heap: decrease key

Case 1. [heap order not violated]

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

decrease-key of x from 46 to 29

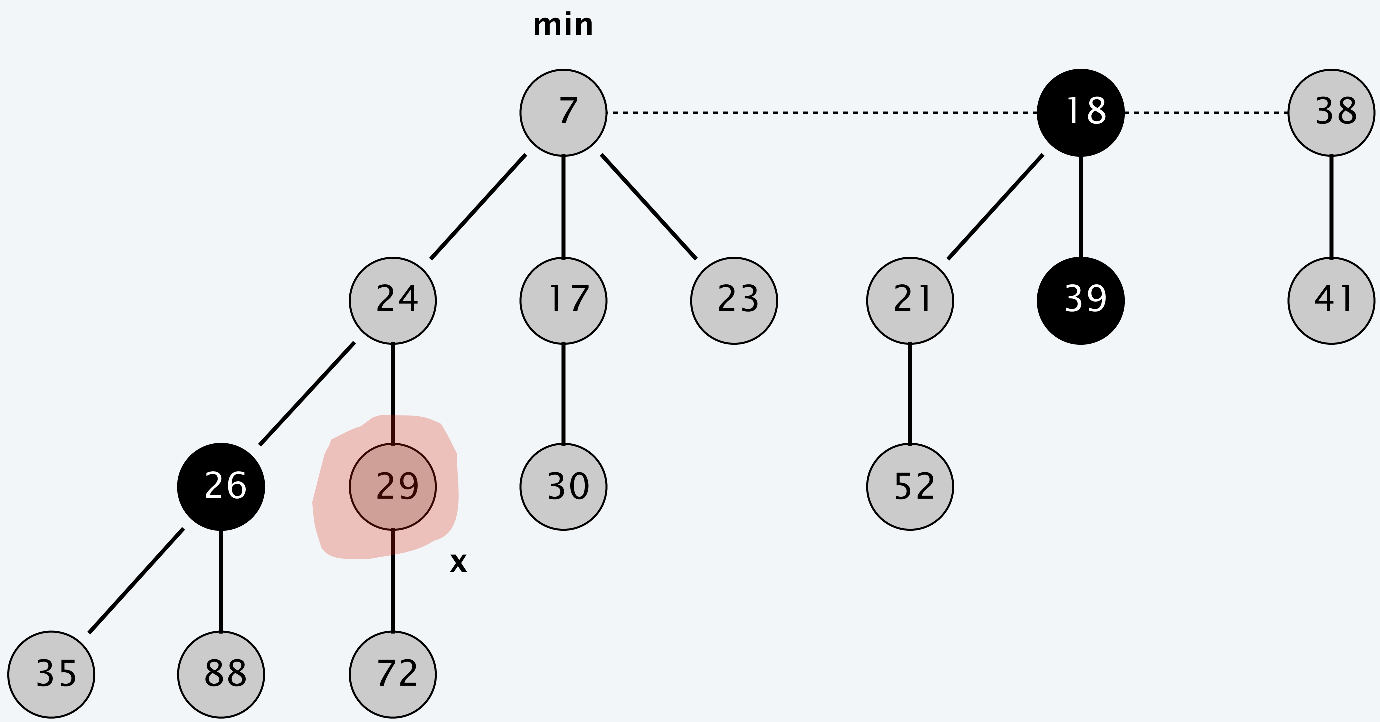


Fibonacci heap: decrease key

Case 1. [heap order not violated]

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

decrease-key of x from 46 to 29

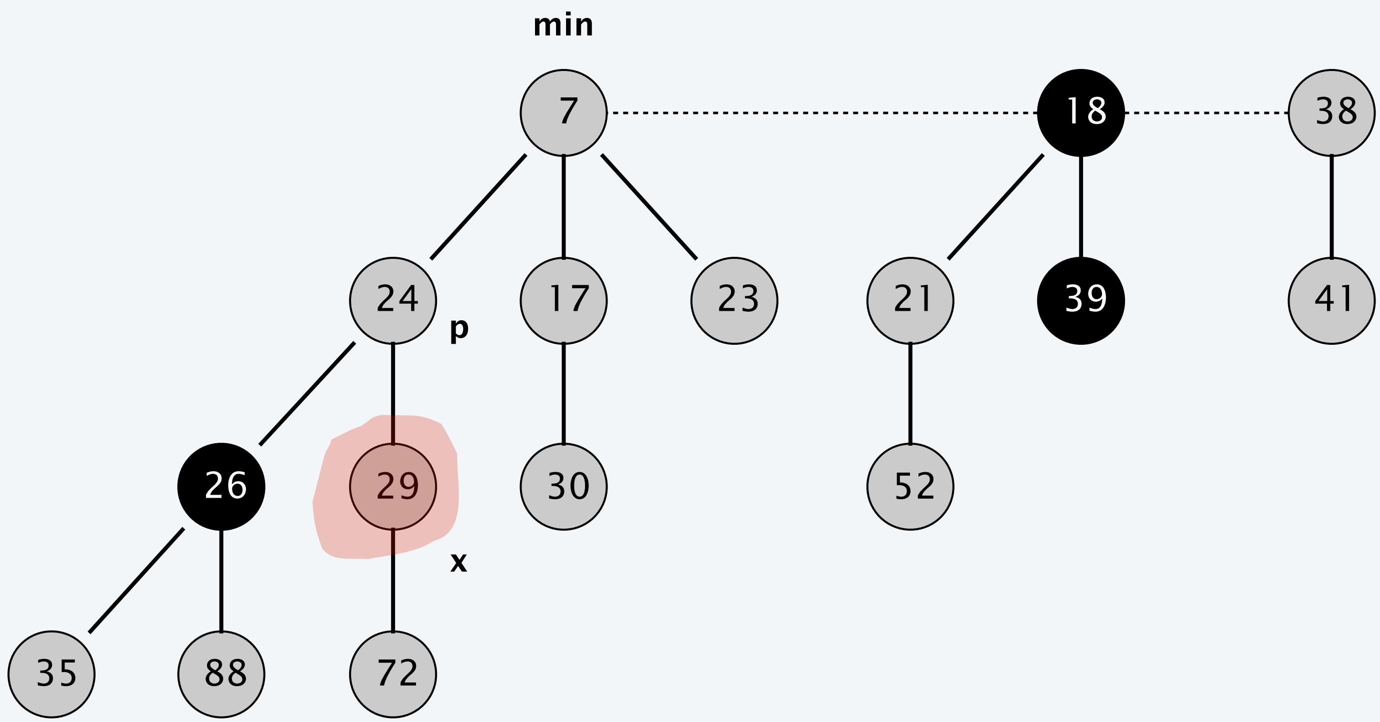


Fibonacci heap: decrease key

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

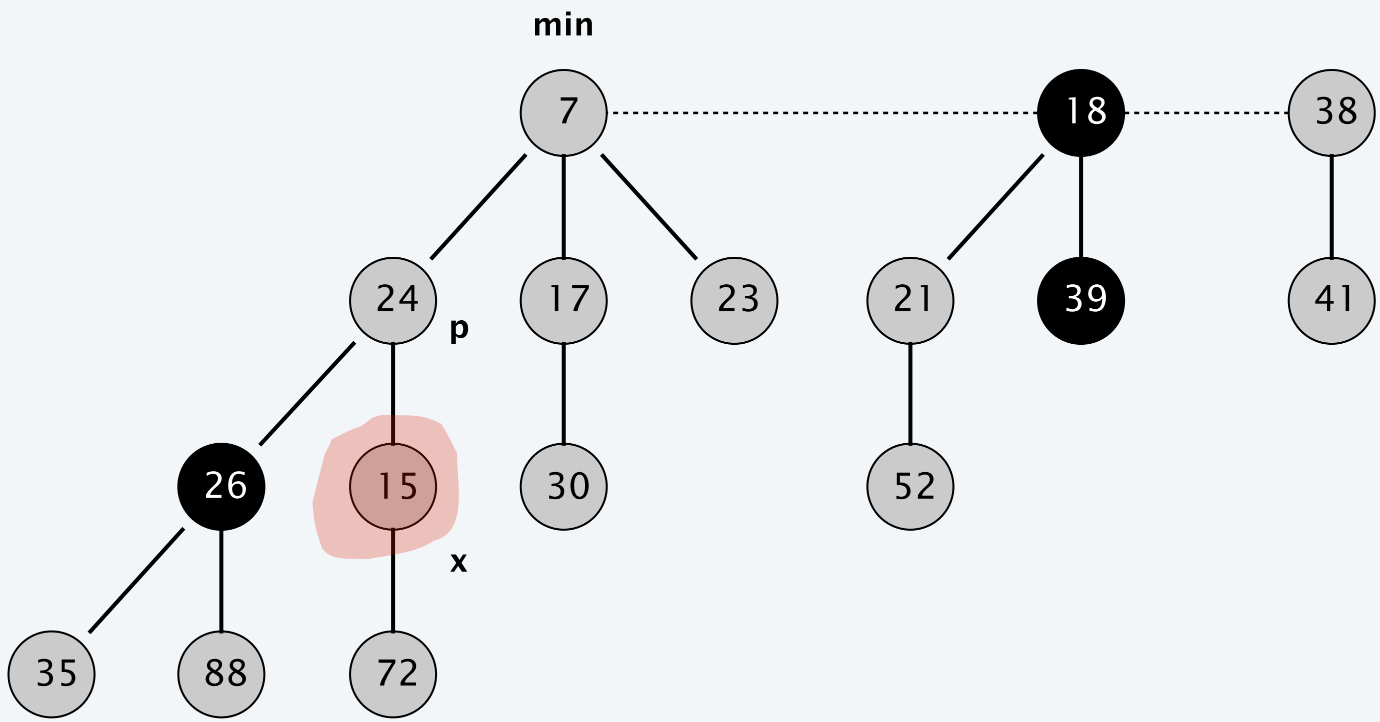


Fibonacci heap: decrease key

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

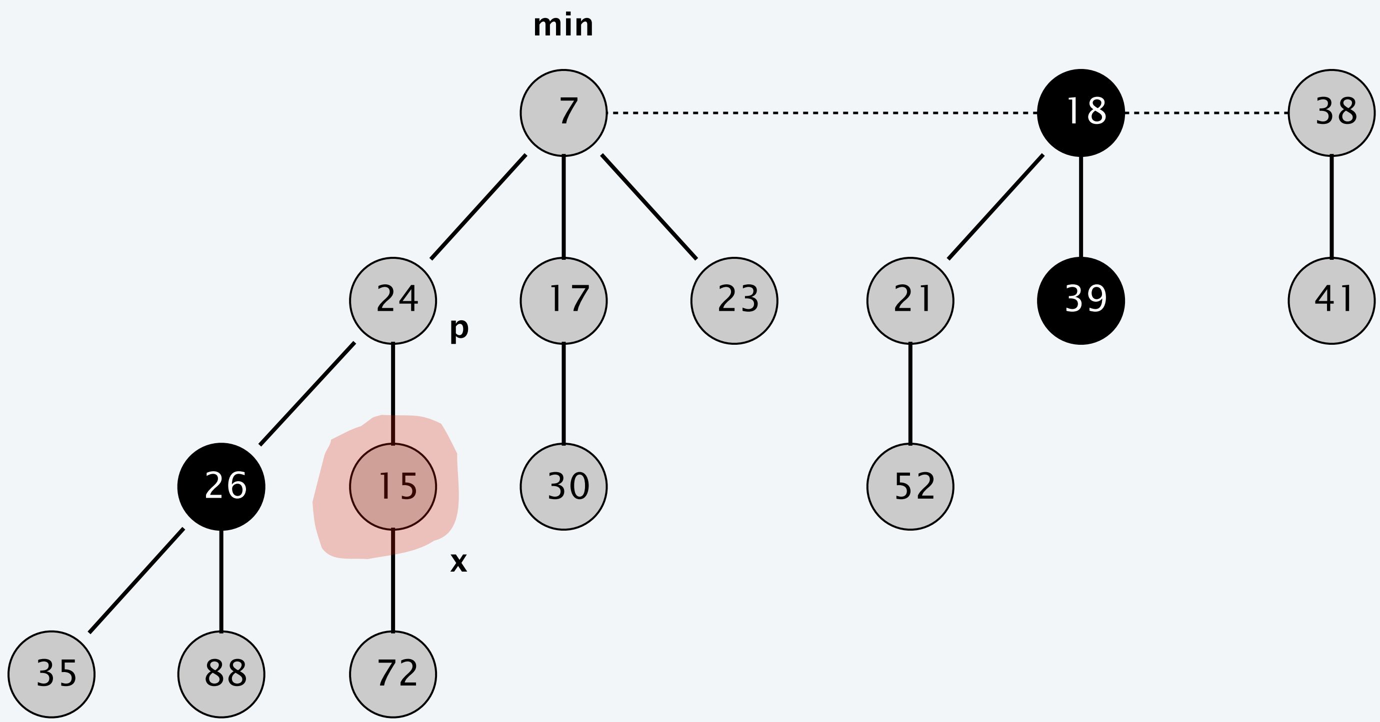


Fibonacci heap: decrease key

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

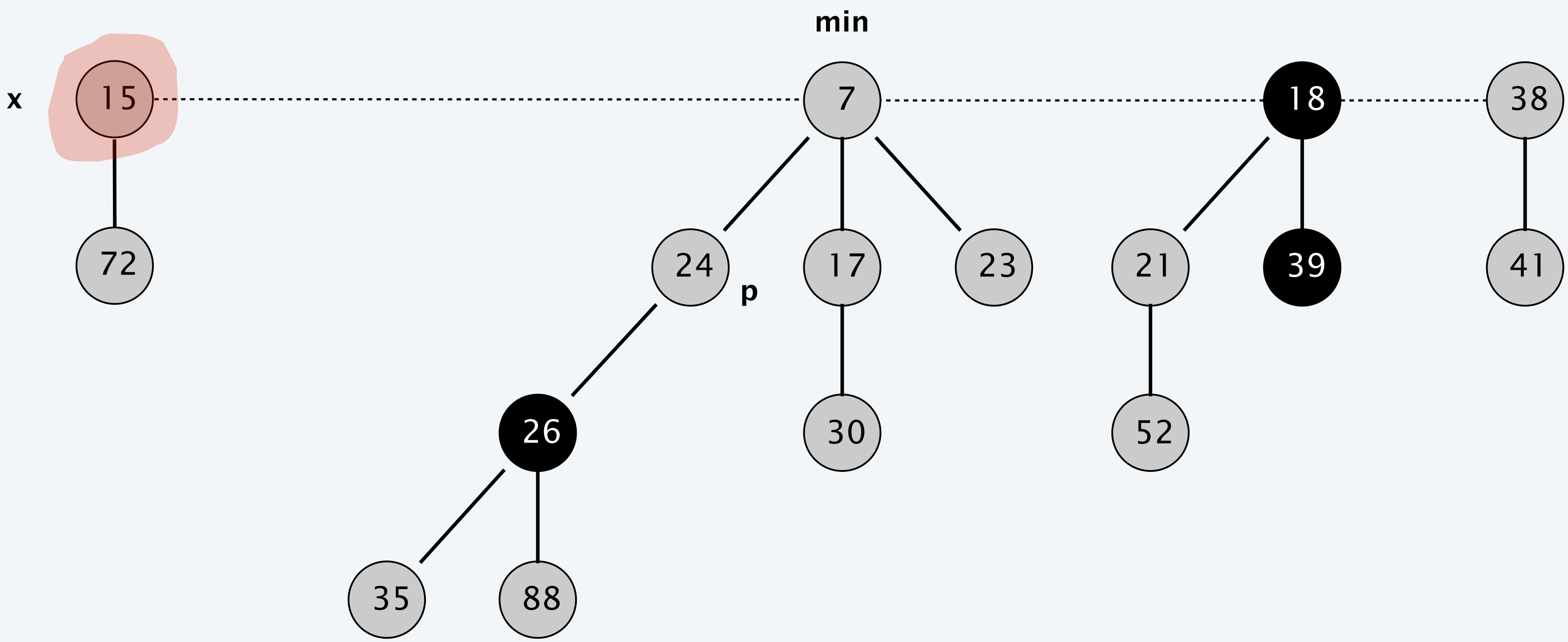


Fibonacci heap: decrease key

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

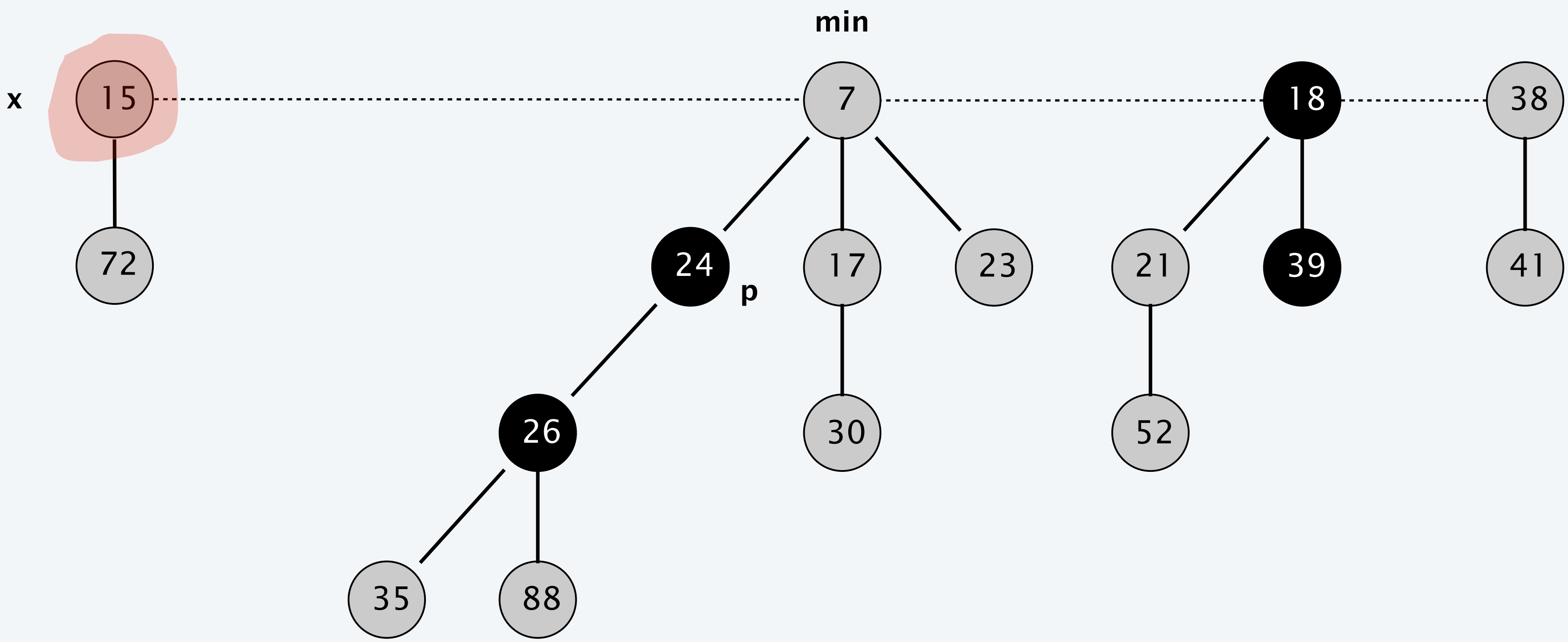


Fibonacci heap: decrease key

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

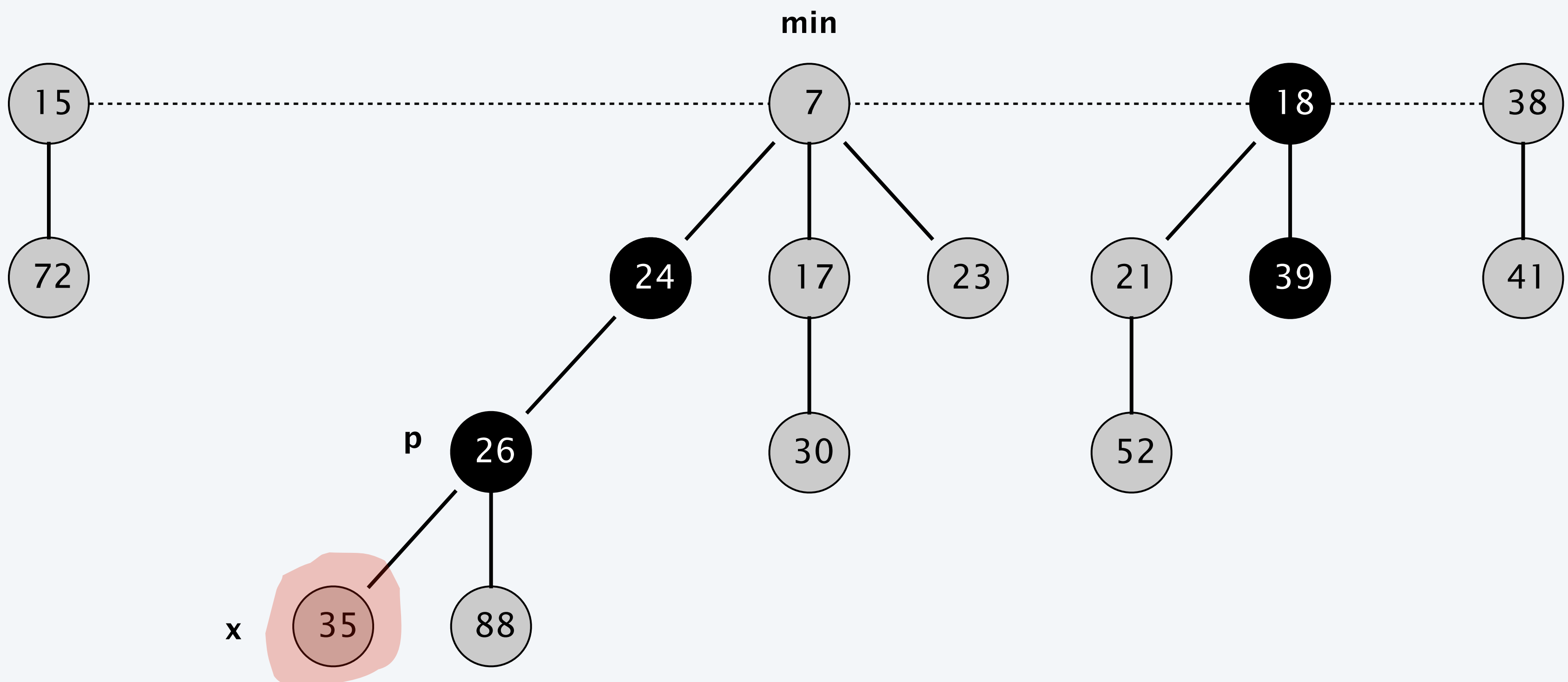


Fibonacci heap: decrease key

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

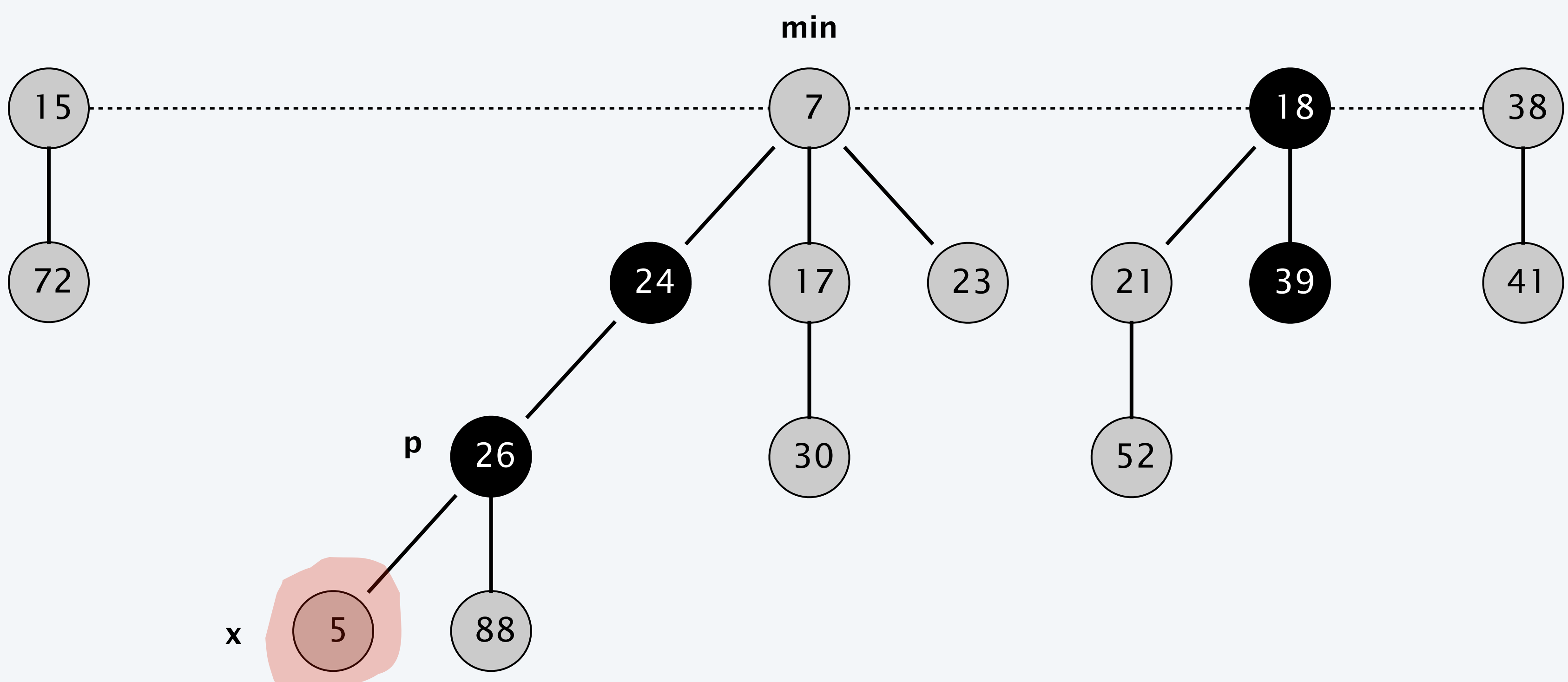


Fibonacci heap: decrease key

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

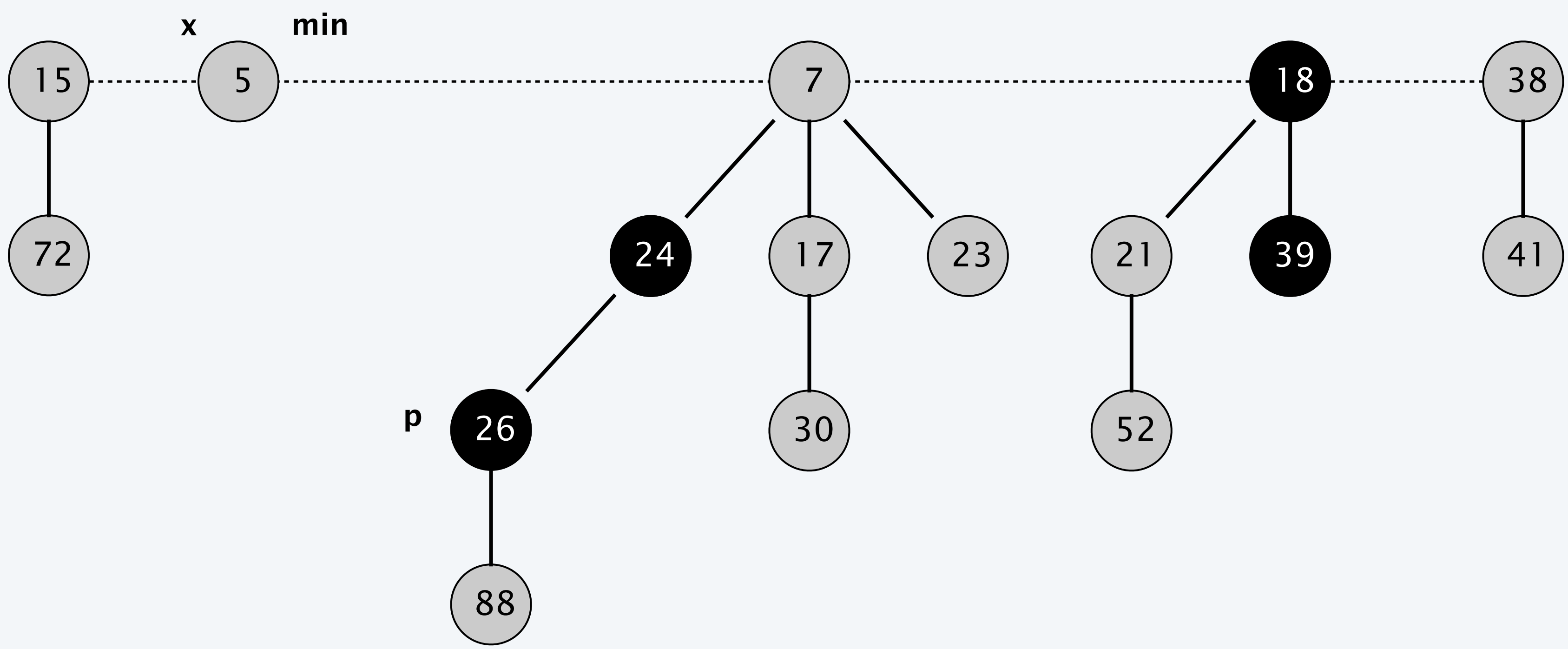


Fibonacci heap: decrease key

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

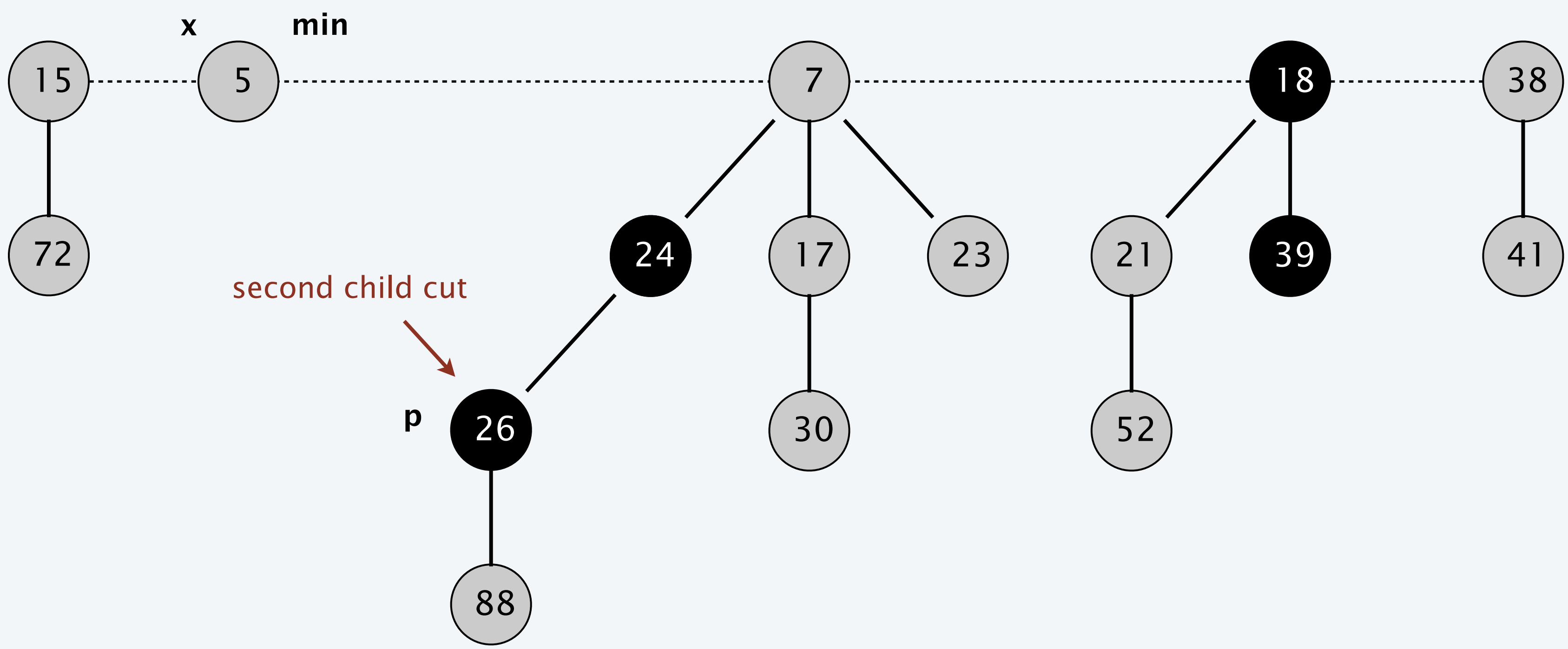


Fibonacci heap: decrease key

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 heap: decrease key

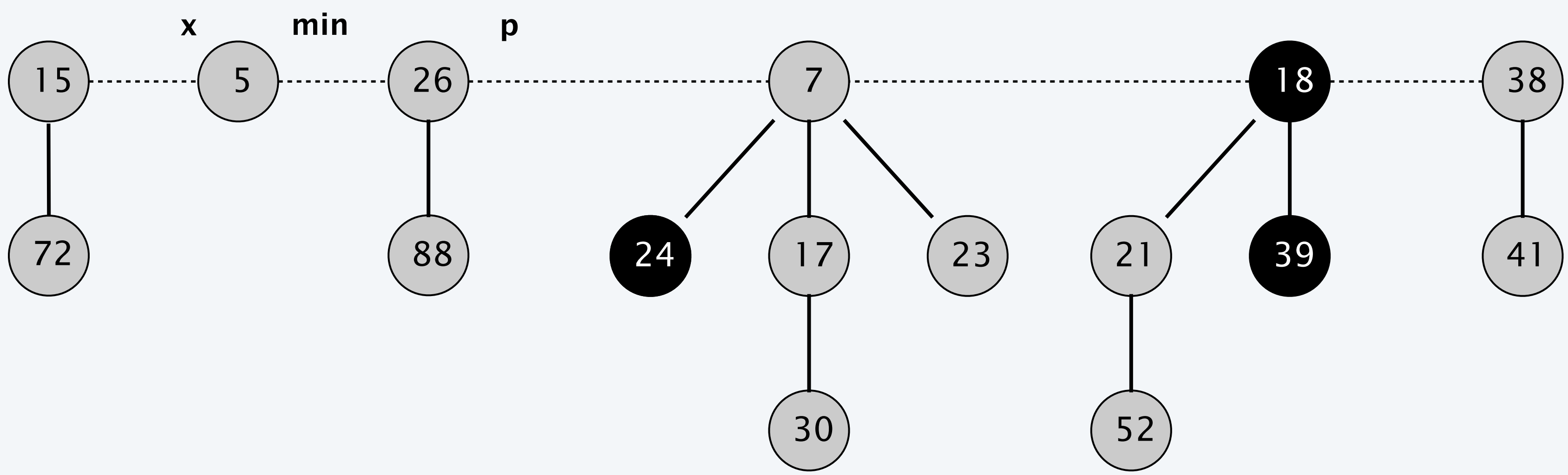
Case 2b. [heap order violated]

- 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

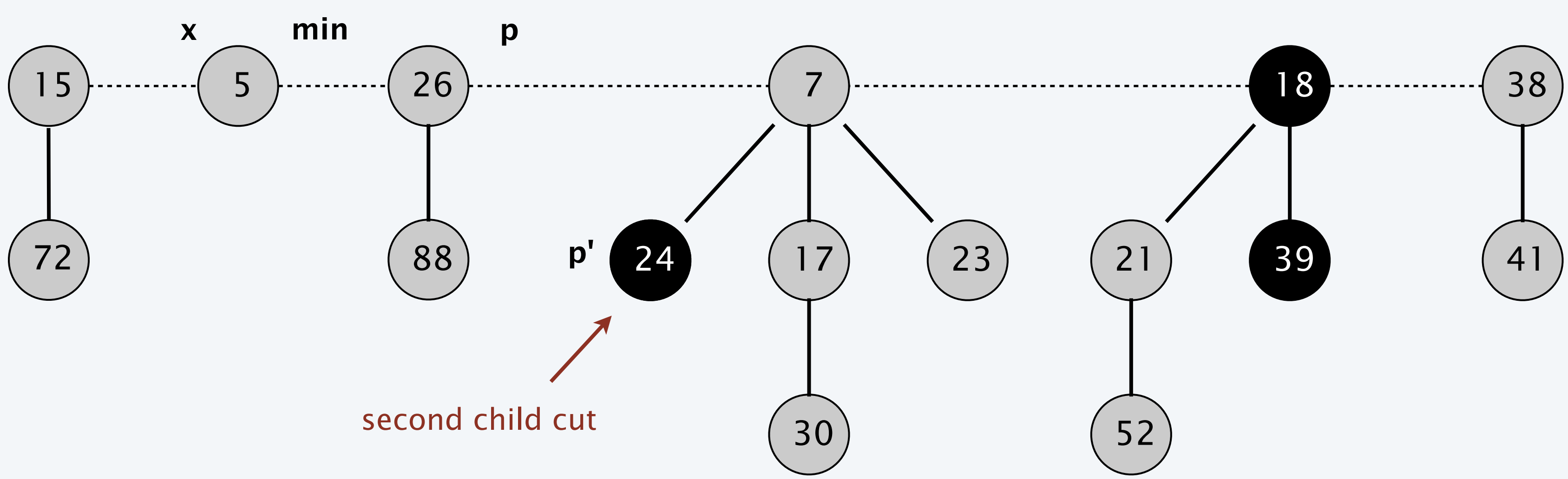


Fibonacci heap: decrease key

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

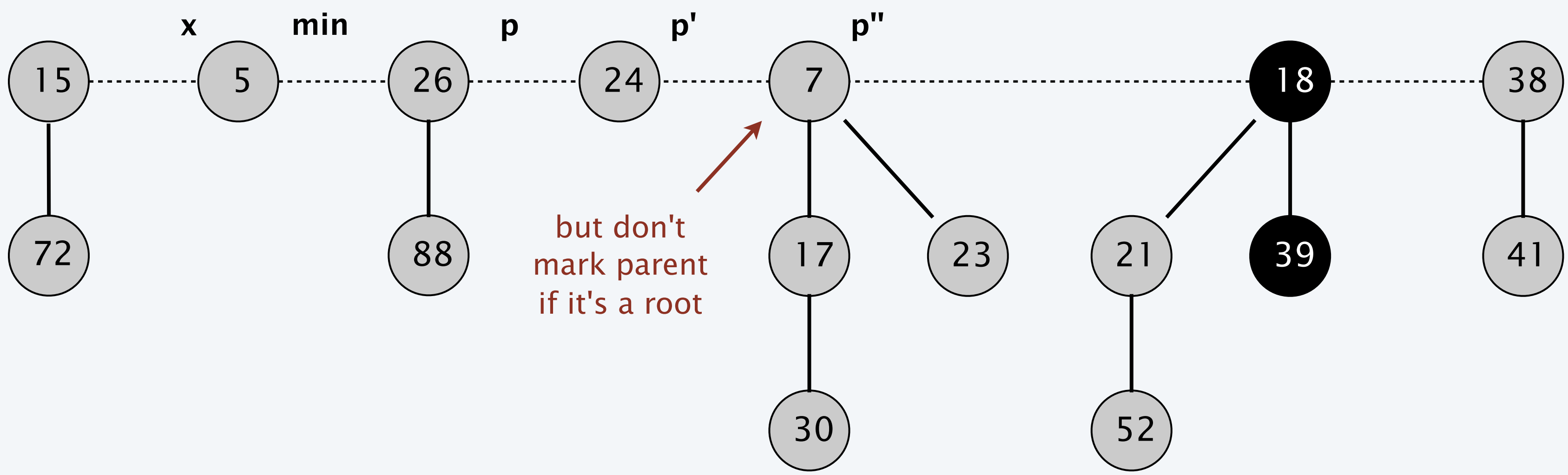


Fibonacci heap: decrease key

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 heap: decrease key analysis

Actual cost. $c_i = O(c)$, where c is the number of cuts.

- $O(1)$ time for changing the key.
- $O(1)$ time for each of c cuts, plus melding into root list.

Change in potential. $\Delta\Phi = O(1) - c$.

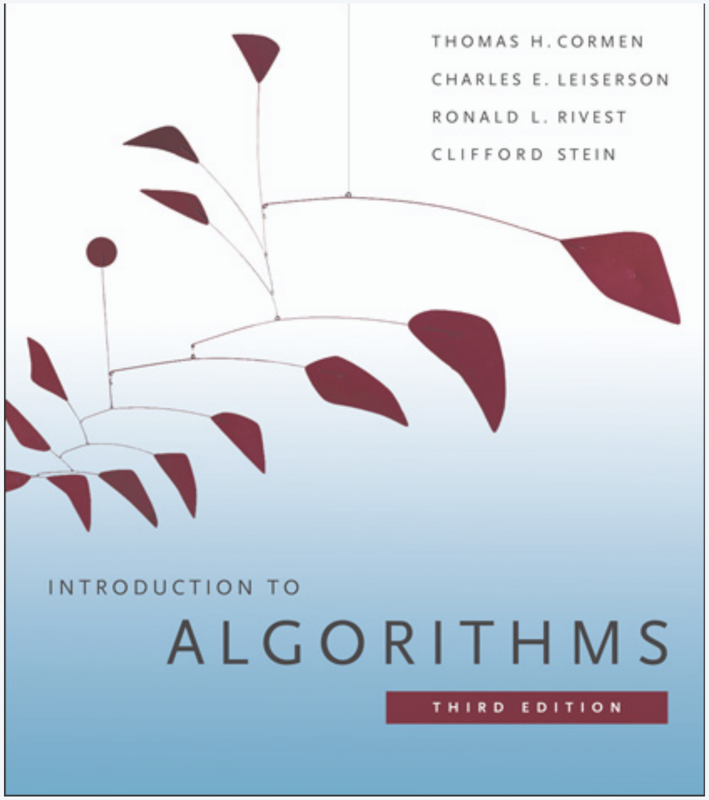
- $trees(H') = trees(H) + c$.
- $marks(H') \le marks(H) - c + 2$.

each cut (except first) unmarks a node
last cut may or may not mark a node

- $\Delta\Phi \le c + 2 \cdot (-c + 2) = 4 - c$.

Amortized cost. $\hat{c}_i = c_i + \Delta\Phi = O(1)$.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$



SECTION 19.4

FIBONACCI HEAPS

- *preliminaries*
- *insert*
- *extract the minimum*
- *decrease key*
- ***bounding the rank***
- *meld and delete*

Analysis summary

Insert. $O(1)$.

Delete-min. $O(rank(H))$ amortized.

Decrease-key. $O(1)$ amortized.

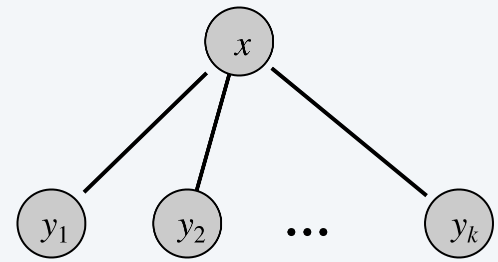
Fibonacci lemma. Let H be a Fibonacci heap with n elements.
Then, $rank(H) = O(\log n)$.

number of nodes is
exponential in rank

Bounding the rank

Lemma 1. Fix a point in time. Let x be a node of rank k, and let $y_1, \ldots, y_k$ denote its current children in the order in which they were linked to x. Then:

$$rank(y_i) \ \geq \ \begin{cases} 0 & \text{if } i = 1 \\ i - 2 & \text{if } i \geq 2 \end{cases}$$



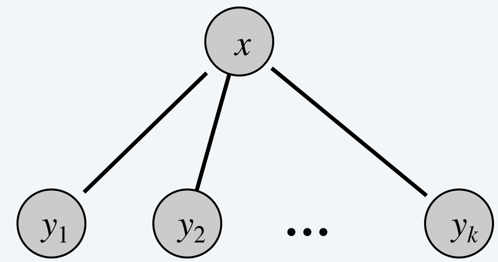
Pf.

- When y_i was linked into x, x had at least $i - 1$ children $y_1, \ldots, y_{i-1}$.
- Since only trees of equal rank are linked, at that time
  $rank(y_i) \ = \ rank(x) \ \geq \ i - 1$.
- Since then, y_i has lost at most one child (or y_i would have been cut).
- Thus, right now $rank(y_i) \ \geq \ i - 2$. ▪

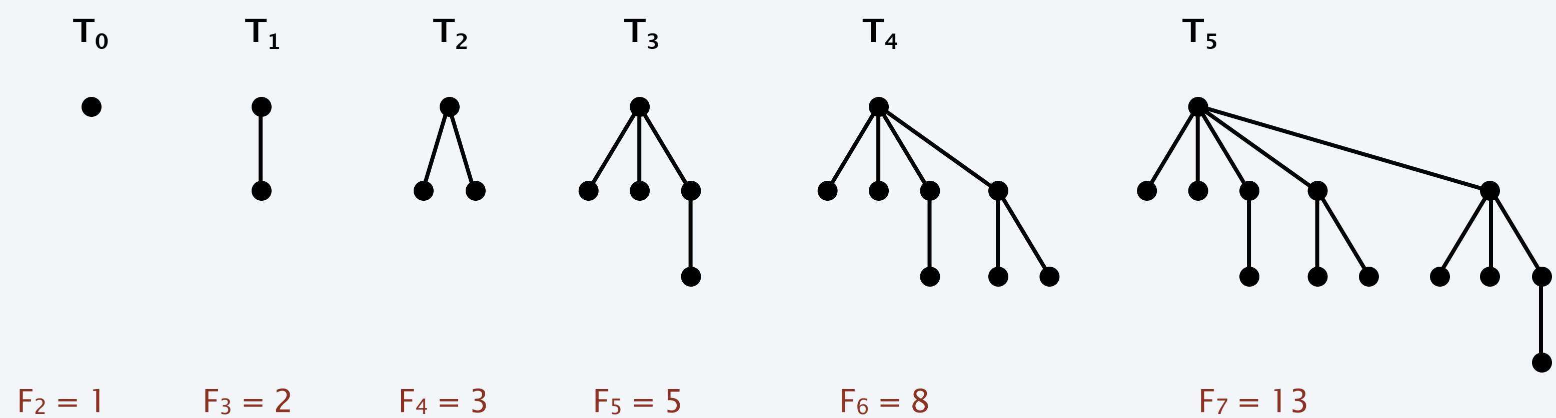
Bounding the rank

Lemma 1. Fix a point in time. Let x be a node of rank k, and let $y_1, \ldots, y_k$ denote its current children in the order in which they were linked to x. Then:

$$rank(y_i) \;\geq\; \begin{cases} 0 & \text{if } i = 1 \\ i-2 & \text{if } i \geq 2 \end{cases}$$



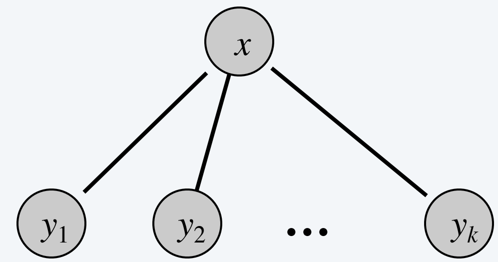
Def. Let T_k be smallest possible tree of rank k satisfying property.



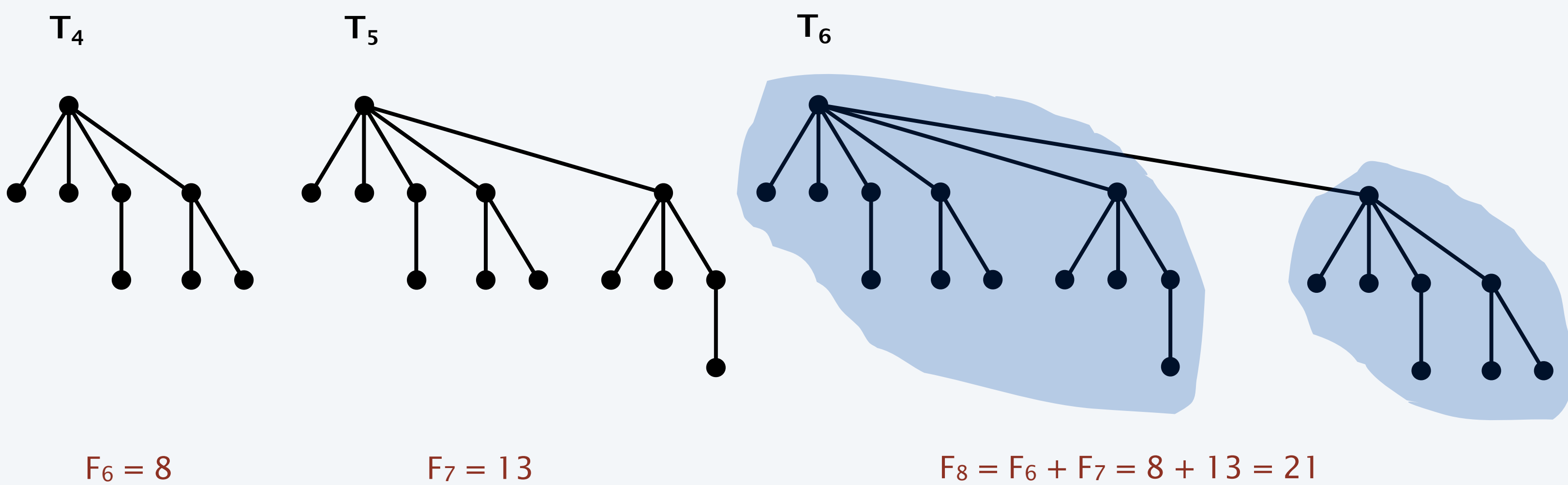
Bounding the rank

Lemma 1. Fix a point in time. Let x be a node of rank k, and let $y_1, \ldots, y_k$ denote its current children in the order in which they were linked to x. Then:

$$rank(y_i) \ \geq \begin{cases} 0 & \text{if } i = 1 \\ i-2 & \text{if } i \geq 2 \end{cases}$$



Def. Let T_k be smallest possible tree of rank k satisfying property.



Bounding the rank

Lemma 2. Let s_k be minimum number of elements in any Fibonacci heap of rank k. Then $s_k \geq F_{k+2}$, where F_k is the k^{th} Fibonacci number.

Pf. [by strong induction on k]

- Base cases: $s_0 = 1$ and $s_1 = 2$.
- Inductive hypothesis: assume $s_i \geq F_{i+2}$ for $i = 0, \ldots, k-1$.
- As in Lemma 1, let let $y_1, \ldots, y_k$ denote its current children in the order in which they were linked to x.

$$
\begin{aligned}
s_k &\geq 1 + 1 + (s_0 + s_1 + \ldots + s_{k-2}) && \text{(Lemma 1)} \\
&\geq (1 + F_1) + F_2 + F_3 + \ldots + F_k && \text{(inductive hypothesis)} \\
&= F_{k+2}. \quad \blacksquare && \text{(Fibonacci fact 1)}
\end{aligned}
$$

Bounding the rank

Fibonacci lemma. Let H be a Fibonacci heap with n elements.
Then, $rank(H) \le \log_\phi n$, where ϕ is the golden ratio $= (1 + \sqrt{5}) / 2 \approx 1.618$.

Pf.

- Let H is a Fibonacci heap with n elements and rank k.
- Then $n \underset{\uparrow \text{ Lemma 2}}{\ge} F_{k+2} \underset{\uparrow \text{ Fibonacci Fact 2}}{\ge} \phi^k$.
- Taking logs, we obtain $rank(H) = k \le \log_\phi n$. ▪

Fibonacci fact 1

Def. The Fibonacci sequence is: $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

Fibonacci fact 1. For all integers $k \geq 0$, $F_{k+2} = 1 + F_0 + F_1 + \ldots + F_k$.

Pf. [by induction on k]

- Base case: $F_2 = 1 + F_0 = 2$.
- Inductive hypothesis: assume $F_{k+1} = 1 + F_0 + F_1 + \ldots + F_{k-1}$.

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} && \text{(definition)} \\ &= F_k + (1 + F_0 + F_1 + \ldots + F_{k-1}) && \text{(inductive hypothesis)} \\ &= 1 + F_0 + F_1 + \ldots + F_{k-1} + F_k. \quad \blacksquare && \text{(algebra)} \end{aligned}$$

Fibonacci fact 2

Def. The Fibonacci sequence is: $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

Fibonacci fact 2. $F_{k+2} \geq \phi^k$, where $\phi = (1 + \sqrt{5}) / 2 \approx 1.618$.

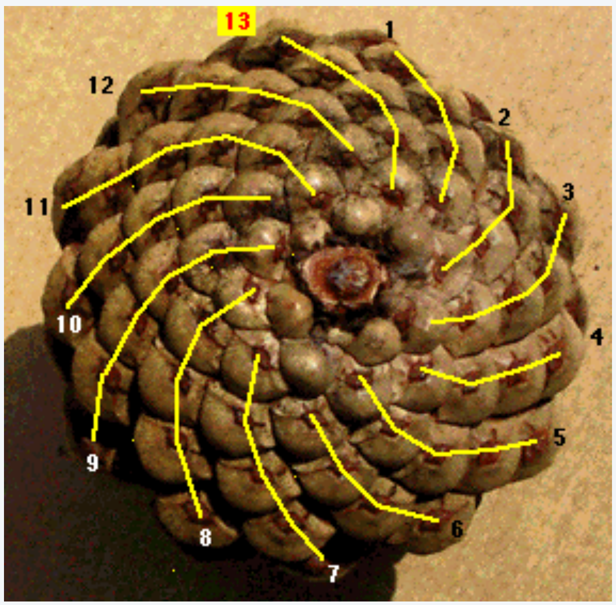
Pf. [by induction on k]

- Base cases: $F_2 = 1 \geq 1$, $F_3 = 2 \geq \phi$.
- Inductive hypotheses: assume $F_k \geq \phi^k$ and $F_{k+1} \geq \phi^{k+1}$

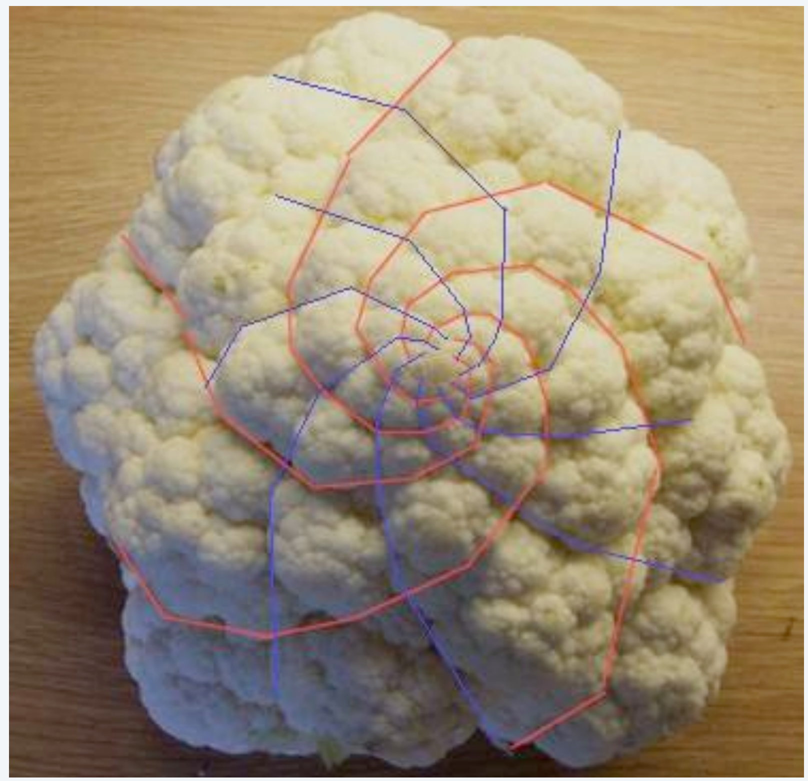
$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} && \text{(definition)} \\ &\geq \phi^{k-1} + \phi^{k-2} && \text{(inductive hypothesis)} \\ &= \phi^{k-2}(1 + \phi) && \text{(algebra)} \\ &= \phi^{k-2}\,\phi^2 && (\phi^2 = \phi + 1) \\ &= \phi^k. \;\blacksquare && \text{(algebra)} \end{aligned}$$

Fibonacci numbers and nature

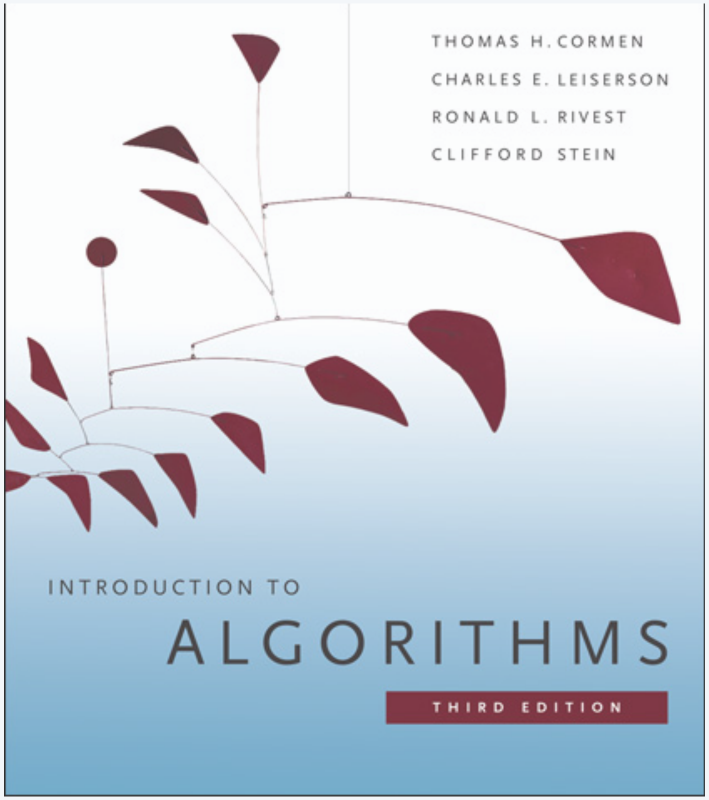
Fibonacci numbers arise both in nature and algorithms.



pinecone



cauliflower



SECTION 19.2, 19.3

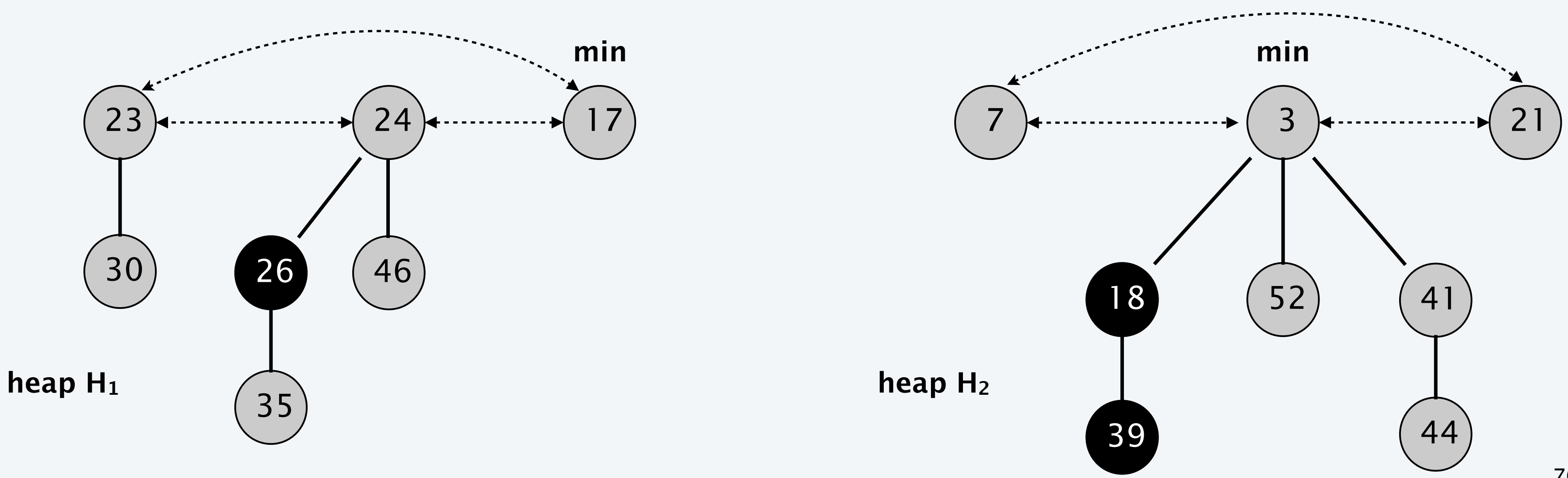
FIBONACCI HEAPS

- *preliminaries*
- *insert*
- *extract the minimum*
- *decrease key*
- *bounding the rank*
- *meld and delete*

Fibonacci heap: meld

Meld. Combine two Fibonacci heaps (destroying old heaps).

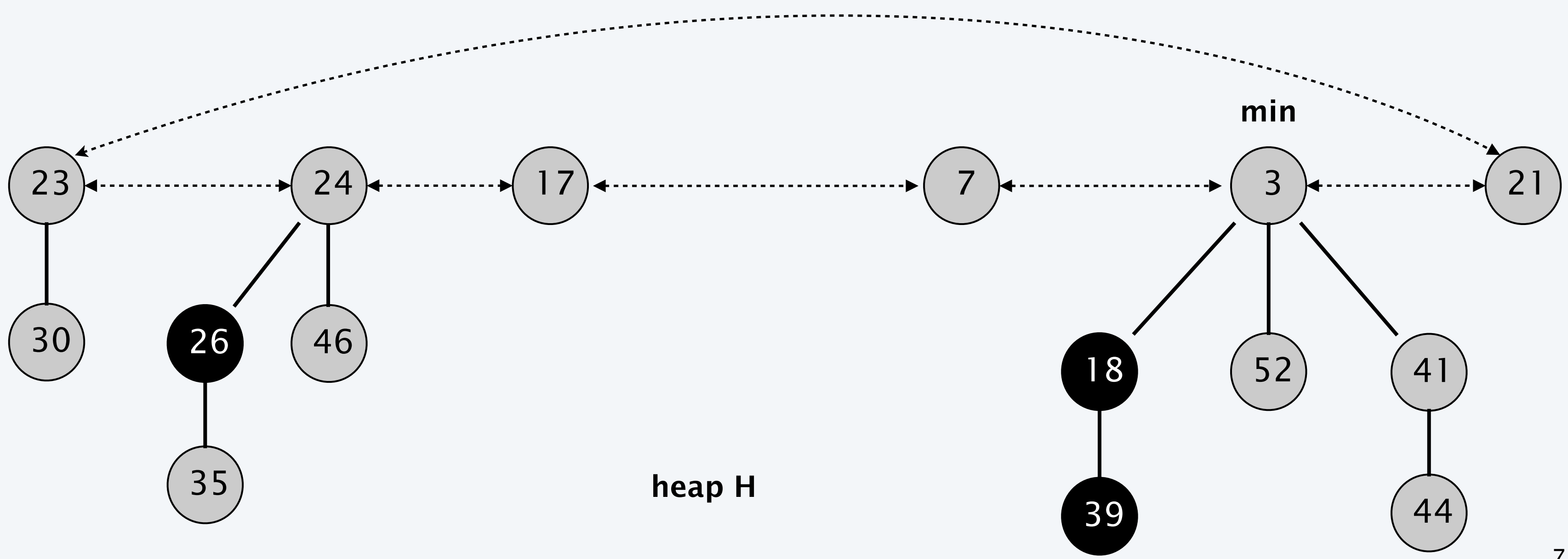
Recall. Root lists are circular, doubly-linked lists.



Fibonacci heap: meld

Meld. Combine two Fibonacci heaps (destroying old heaps).

Recall. Root lists are circular, doubly-linked lists.



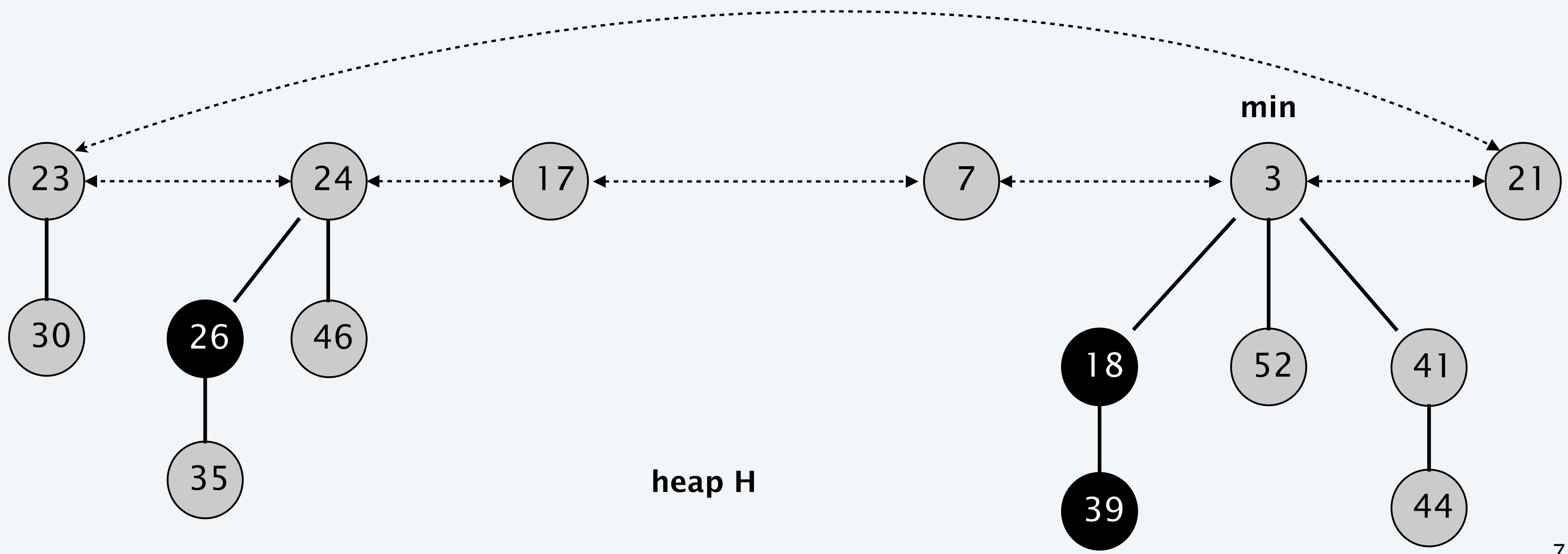
Fibonacci heap: meld analysis

Actual cost. $c_i = O(1)$.

Change in potential. $\Delta\Phi = 0$.

Amortized cost. $\hat{c}_i = c_i + \Delta\Phi = O(1)$.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$



Fibonacci heap: delete

Delete. Given a handle to an element x, delete it from heap H.

- DECREASE-KEY($H, x, -\infty$).
- EXTRACT-MIN(H).

Amortized cost. $\hat{c}_i = O(rank(H))$.

- $O(1)$ amortized for DECREASE-KEY.
- $O(rank(H))$ amortized for EXTRACT-MIN.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Priority queues performance cost summary

operation	linked list	binary heap	binomial heap	Fibonacci heap †
MAKE-HEAP	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IS-EMPTY	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
EXTRACT-MIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
MELD	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
FIND-MIN	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$

† amortized

Accomplished. $O(1)$ INSERT and DECREASE-KEY, $O(\log n)$ EXTRACT-MIN.

PRIORITY QUEUES

- binary heaps
- d-ary heaps
- binomial heaps
- Fibonacci heaps
- advanced topics

Heaps of heaps

- b-heaps.
- Fat heaps.
- 2-3 heaps.
- Leaf heaps.
- Thin heaps.
- Skew heaps.
- Splay heaps.
- Weak heaps.
- Leftist heaps.
- Quake heaps.
- Pairing heaps.
- Violation heaps.
- Run-relaxed heaps.
- Rank-pairing heaps.
- Skew-pairing heaps.
- Rank-relaxed heaps.
- Lazy Fibonacci heaps.

Brodal queues

Q. Can we achieve same running time as for Fibonacci heap but with worst-case bounds per operation (instead of amortized)?

Theory. [Brodal 1996] Yes.

Worst-Case Efficient Priority Queues*

Gerth Stølting Brodal†

Abstract

An implementation of priority queues is presented that supports the operations MakeQueue, FindMin, Insert, Meld and DecreaseKey in worst case time $O(1)$ and DeleteMin and Delete in worst case time $O(\log n)$. The space requirement is linear. The data structure presented is the first achieving this worst case performance.

Practice. Ever implemented? Constants are high (and requires RAM model).

Strict Fibonacci heaps

Q. Can we achieve same running time as for Fibonacci heap but with worst-case bounds per operation (instead of amortized) in pointer model?

Theory. [Brodal-Lagogiannis-Tarjan 2002] Yes.

Gerth Stølting Brodal
MADALGO*
Dept. of Computer Science
Aarhus University
Åbogade 34, 8200 Aarhus N
Denmark
gerth@cs.au.dk

George Lagogiannis
Agricultural University
of Athens
Iera Odos 75, 11855 Athens
Greece
lagogian@aua.gr

Robert E. Tarjan†
Dept. of Computer Science
Princeton University
and HP Labs
35 Olden Street, Princeton
New Jersey 08540, USA
ret@cs.princeton.edu

ABSTRACT

We present the first pointer-based heap implementation with time bounds matching those of Fibonacci heaps in the worst case. We support make-heap, insert, find-min, meld and decrease-key in worst-case $O(1)$ time, and delete and delete-min in worst-case $O(\lg n)$ time, where n is the size of the heap. The data structure uses linear space.

A previous, very complicated, solution achieving the same time bounds in the RAM model made essential use of arrays and extensive use of redundant counter schemes to maintain balance. Our solution uses neither. Our key simplification is to discard the structure of the smaller heap when doing a meld. We use the pigeonhole principle in place of the redundant counter mechanism.

Fibonacci heaps: practice

Q. Are Fibonacci heaps useful in practice?

A. They are part of LEDA and Boost C++ libraries.
(but other heaps seem to perform better in practice)

Pairing heaps

Pairing heap. A self-adjusting heap-ordered general tree.

The Pairing Heap: A New Form of Self-Adjusting Heap

Michael L. Fredman[1,4], Robert Sedgewick[2,5], Daniel D. Sleator[3], and Robert E. Tarjan[2,3,6]

Abstract. Recently, Fredman and Tarjan invented a new, especially efficient form of heap (priority queue) called the *Fibonacci heap*. Although theoretically efficient, Fibonacci heaps are complicated to implement and not as fast in practice as other kinds of heaps. In this paper we describe a new form of heap, called the *pairing heap*, intended to be competitive with the Fibonacci heap in theory and easy to implement and fast in practice. We provide a partial complexity analysis of pairing heaps. Complete analysis remains an open problem.

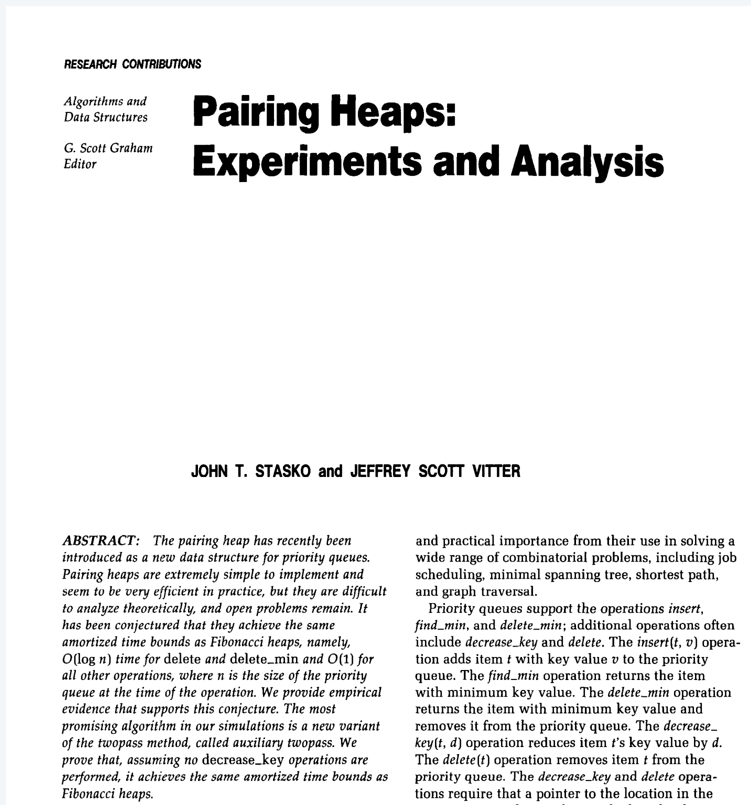
Theory. Same amortized running times as Fibonacci heaps for all operations except DECREASE-KEY.

- $O(\log n)$ amortized. [Fredman et al. 1986]
- $\Omega(\log \log n)$ lower bound on amortized cost. [Fredman 1999]
- $2^{\sqrt{O(\log \log n)}}$ amortized. [Pettie 2005]

Pairing heaps

Pairing heap. A self-adjusting heap-ordered general tree.

Practice. As fast as (or faster than) the binary heap on some problems.
Included in GNU C++ library and LEDA.



RESEARCH CONTRIBUTIONS

Algorithms and Data Structures

G. Scott Graham Editor

Pairing Heaps: Experiments and Analysis

JOHN T. STASKO and JEFFREY SCOTT VITTER

ABSTRACT: The pairing heap has recently been introduced as a new data structure for priority queues. Pairing heaps are extremely simple to implement and seem to be very efficient in practice, but they are difficult to analyze theoretically, and open problems remain. It has been conjectured that they achieve the same amortized time bounds as Fibonacci heaps, namely, O(log n) time for delete *and* delete_min *and O(1) for all other operations, where n is the size of the priority queue at the time of the operation. We provide empirical evidence that supports this conjecture. The most promising algorithm in our simulations is a new variant of the twopass method, called auxiliary twopass. We prove that, assuming no* decrease_key *operations are performed, it achieves the same amortized time bounds as Fibonacci heaps.*

and practical importance from their use in solving a wide range of combinatorial problems, including job scheduling, minimal spanning tree, shortest path, and graph traversal.

Priority queues support the operations *insert*, *find_min*, and *delete_min*; additional operations often include *decrease_key* and *delete*. The *insert*(t, v) operation adds item t with key value v to the priority queue. The *find_min* operation returns the item with minimum key value. The *delete_min* operation returns the item with minimum key value and removes it from the priority queue. The *decrease_key*(t, d) operation reduces item t's key value by d. The *delete*(t) operation removes item t from the priority queue. The *decrease_key* and *delete* operations require that a pointer to the location in the

Priority queues performance cost summary

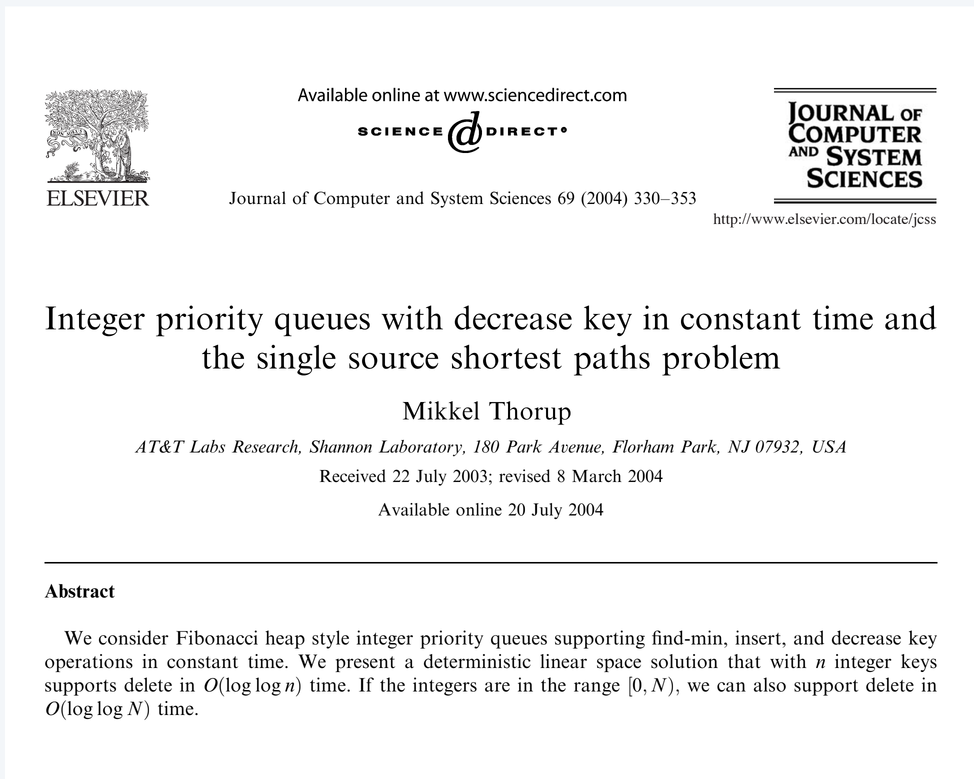
operation	linked list	binary heap	binomial heap	pairing heap †	Fibonacci heap †	Brodal queue
MAKE-HEAP	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IS-EMPTY	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$
EXTRACT-MIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$	$O(\log n)$	$2^{\sqrt{O(\log \log n)}}$	$O(1)$	$O(1)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
MELD	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$
FIND-MIN	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$

† amortized

Priority queues with integer priorities

Assumption. Keys are integers between 0 and C.

Theorem. [Thorup 2004] There exists a priority queue that supports INSERT, FIND-MIN, and DECREASE-KEY in constant time and EXTRACT-MIN and DELETE-KEY in either $O(\log \log n)$ or $O(\log \log C)$ time.



Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Journal of Computer and System Sciences 69 (2004) 330–353

JOURNAL OF COMPUTER AND SYSTEM SCIENCES

http://www.elsevier.com/locate/jcss

Integer priority queues with decrease key in constant time and the single source shortest paths problem

Mikkel Thorup

AT&T Labs Research, Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932, USA

Received 22 July 2003; revised 8 March 2004

Available online 20 July 2004

Abstract

We consider Fibonacci heap style integer priority queues supporting find-min, insert, and decrease key operations in constant time. We present a deterministic linear space solution that with n integer keys supports delete in $O(\log \log n)$ time. If the integers are in the range $[0, N)$, we can also support delete in $O(\log \log N)$ time.

Priority queues with integer priorities

Assumption. Keys are integers between 0 and C.

Theorem. [Thorup 2004] There exists a priority queue that supports INSERT, FIND-MIN, and DECREASE-KEY in constant time and EXTRACT-MIN and DELETE-KEY in either $O(\log \log n)$ or $O(\log \log C)$ time.

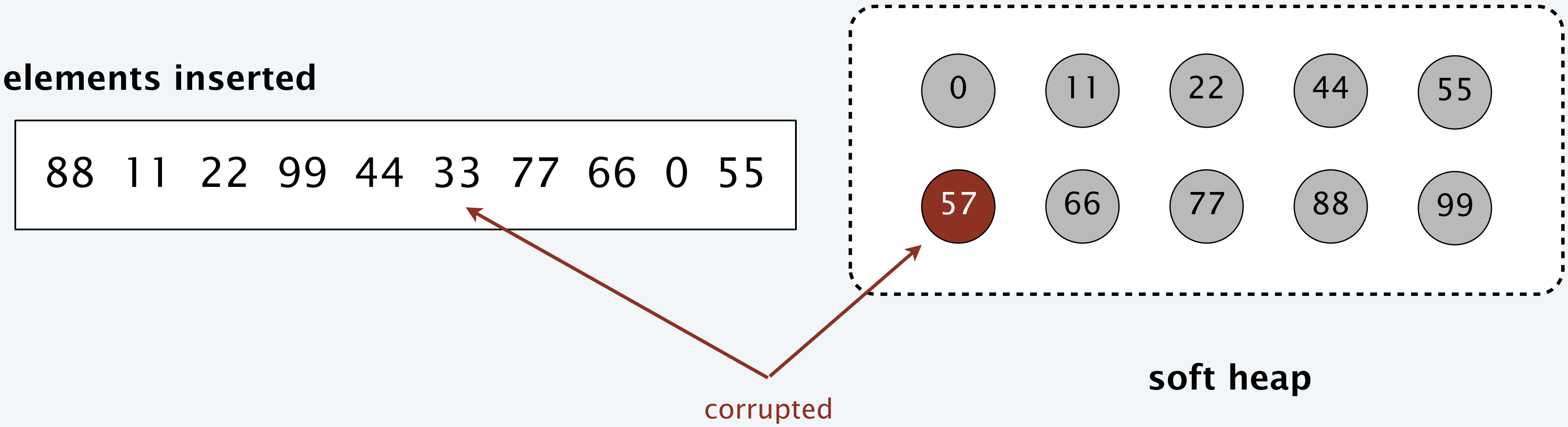
Corollary 1. Can implement Dijkstra's algorithm in either $O(m \log \log n)$ or $O(m \log \log C)$ time.

Corollary 2. Can sort n integers in $O(n \log \log n)$ time.

Computational model. Word RAM.

Soft heaps

Goal. Break information-theoretic lower bound by allowing priority queue to corrupt 10% of the keys (by increasing them).



Soft heaps

Goal. Break information-theoretic lower bound by allowing priority queue to corrupt 10% of the keys (by increasing them).

Representation.

- Set of binomial trees (with some subtrees missing).
- Each node may store several elements.
- Each node stores a value that is an upper bound on the original keys.
- Binomial trees are heap-ordered with respect to these values.

Soft heaps

Goal. Break information-theoretic lower bound by allowing priority queue to corrupt 10% of the keys (by increasing them).

Theorem. [Chazelle 2000] Starting from an empty soft heap, any sequence of n INSERT, MIN, EXTRACT-MIN, MELD, and DELETE operations takes $O(n)$ time and at most 10% of its elements are corrupted at any given time.

The Soft Heap: An Approximate Priority Queue with Optimal Error Rate

BERNARD CHAZELLE

Princeton University, Princeton, New Jersey, and NEC Research Institute

Abstract. A simple variant of a priority queue, called a *soft heap*, is introduced. The data structure supports the usual operations: insert, delete, meld, and findmin. Its novelty is to beat the logarithmic bound on the complexity of a heap in a comparison-based model. To break this information-theoretic barrier, the entropy of the data structure is reduced by artificially raising the values of certain keys. Given any mixed sequence of n operations, a soft heap with error rate ε (for any $0 < \varepsilon \leq 1/2$) ensures that, at any time, at most εn of its items have their keys raised. The amortized complexity of each operation is constant, except for insert, which takes $O(\log 1/\varepsilon)$ time. The soft heap is optimal for any value of ε in a comparison-based model. The data structure is purely pointer-based. No arrays are used and no numeric assumptions are made on the keys. The main idea behind the soft heap is to move items across the data structure not individually, as is customary, but in groups, in a data-structuring equivalent of "car pooling." Keys must be raised as a result, in order to preserve the heap ordering of the data structure. The soft heap can be used to compute exact or approximate medians and percentiles optimally. It is also useful for approximate sorting and for computing minimum spanning trees of general graphs.

Soft heaps

Goal. Break information-theoretic lower bound by allowing priority queue to corrupt 10% of the keys (by increasing them).

Q. Brilliant. But how could it possibly be useful?

Ex. Linear-time deterministic selection. To find k^{th} smallest element:

- Insert the n elements into soft heap.
- Extract the minimum element $n / 2$ times.
- The largest element deleted $\geq 4n / 10$ elements and $\leq 6n / 10$ elements.
- Can remove $\geq 5n / 10$ of elements and recur.
- $T(n) \leq T(3n/5) + O(n) \Rightarrow T(n) = O(n)$. ▪

Soft heaps

Theorem. [Chazelle 2000] There exists an $O(m\,\alpha(m, n))$ time deterministic algorithm to compute an MST in a graph with n nodes and m edges.

Algorithm. Borůvka + nongreedy + divide-and-conquer + soft heap + ...

A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity

BERNARD CHAZELLE

Princeton University, Princeton, New Jersey, and NEC Research Institute

Abstract. A deterministic algorithm for computing a minimum spanning tree of a connected graph is presented. Its running time is $O(m\alpha(m, n))$, where α is the classical functional inverse of Ackermann's function and n (respectively, m) is the number of vertices (respectively, edges). The algorithm is comparison-based: it uses pointers, not arrays, and it makes no numeric assumptions on the edge costs.