# COL106
# Data Structures and Algorithms

Subodh Sharma and Rahul Garg

# Fibonacci Heaps

Based on slides by: Kevin Wayne, Princeton University, Data Structures, Stanford University

# Why Binomial/Fibonacci Heaps?

| Operation | Linked List | Binary Heap | Binomial Heap | Fibonacci Heap † | Relaxed Heap |
|---|---|---|---|---|---|
| *make-heap* | 1 | 1 | 1 | 1 | 1 |
| *is-empty* | 1 | 1 | 1 | 1 | 1 |
| *insert* | 1 | log *n* | log *n* | 1 | 1 |
| *delete-min* | *n* | log *n* | log *n* | log *n* | log *n* |
| *decrease-key* | *n* | log *n* | log *n* | 1 | 1 |
| *delete* | *n* | log *n* | log *n* | log *n* | log *n* |
| *union* | 1 | *n* | log *n* | 1 | 1 |
| *find-min* | *n* | 1 | log *n* | 1 | 1 |

*n* = number of elements in priority queue

*Runtime of Dijkstra's/Prim's Algorithm*

O(|V|²)

O(|E|log(|V|)

O(|E| + |V|log(|V|)

***Dijkstra's/Prim's***

**1 make-heap**

**|V| insert**

**|V| delete-min**

**|E| decrease**

# Why Fibonacci Heaps?

- History. [Fredman and Tarjan, 1986]
  - Ingenious data structure and analysis.
  - Original motivation: improve Dijkstra's shortest path algorithm from O($E \log V$) to O($E + V \log V$)
  - Also works for Prims's MST algorithm
- O(1) decrease key operation (amortized)
- Need all your attention

# Fibonacci Heaps: Key Ideas

- Binomial trees
- Lazy merging of trees at the heap root
  - Only merge trees during deleteMin operation
  - Union of two heaps in O(1) time
- Amortized analysis
- Decrease key implemented using direct cutting of the node subtree and moving to root
  - No longer perfect binomial trees
  - Maintain nodes as marked and limit such imperfections
  - Nice properties (degrees, size) of binomial tree are preserved

# Amortized Analysis

- Consider an algorithm taking times $t_1$, $t_2$, ... $t_k$ for performing k operations in sequence
- We cannot bound the worst-case time $t_i$
- Maintain a potential function ϕ(i) at every step
- ϕ(i) is like a computation bank.
  - You may deposit and withdraw from it to bound worst case amortized time $at_i$
  - ϕ(i) may depend on the internal state of the data structures
- Amortized time at step i, $at_i = t_i + \phi(i+1) - \phi(i)$
- Total time = $\sum_{i=1}^{T} t_i = \sum_{i=1}^{T} at_i - \phi(T+1) + \phi(1)$
- If ϕ(T+1) - ϕ(1), then total time is same as total amortized time
- Example: Analysis of dynamic arrays

# Example: Time to Increment a Binary Number

- Given an n-bit binary number
- How long does it take to increment it?

$b_n$ $b_{n-1}$ ...... $b_2$ $b_1$ $b_0$

Worst case time: O(n) because if $b_{n-1}$ ... $b_0$ are all 1's then one needs to flip all the bits

Let $\phi$ = number of bits that are 1

Amortized time:

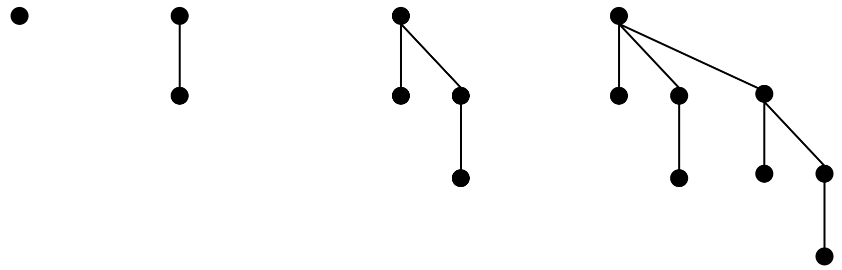
- The algo start with flipping the lsb's that are 1
- Stops when it reaches a bit that is 0 after making it 1
- Time taken: Number of 1's flopped to zero + 1
- Amortized time taken: 2

# Fibonacci Heaps: Lazy Merging at the Root

# Fibonacci Heaps: Lazy Merging at the Root

Basic idea.

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



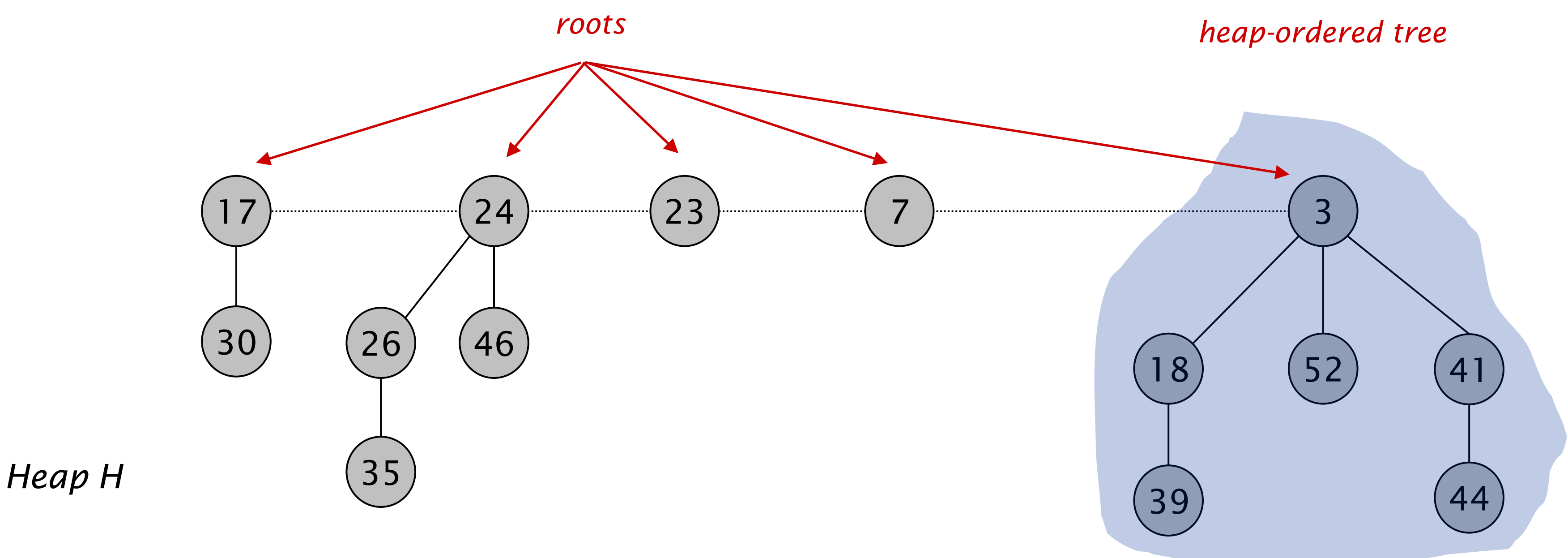
- Fibonacci heap: lazily defer consolidation until next *delete-min*.

# Fibonacci Heaps: Structure

each parent larger than its children

**Fibonacci heap.**

- Set of heap-ordered trees.
- Maintain pointer to minimum element.
- Set of marked nodes.

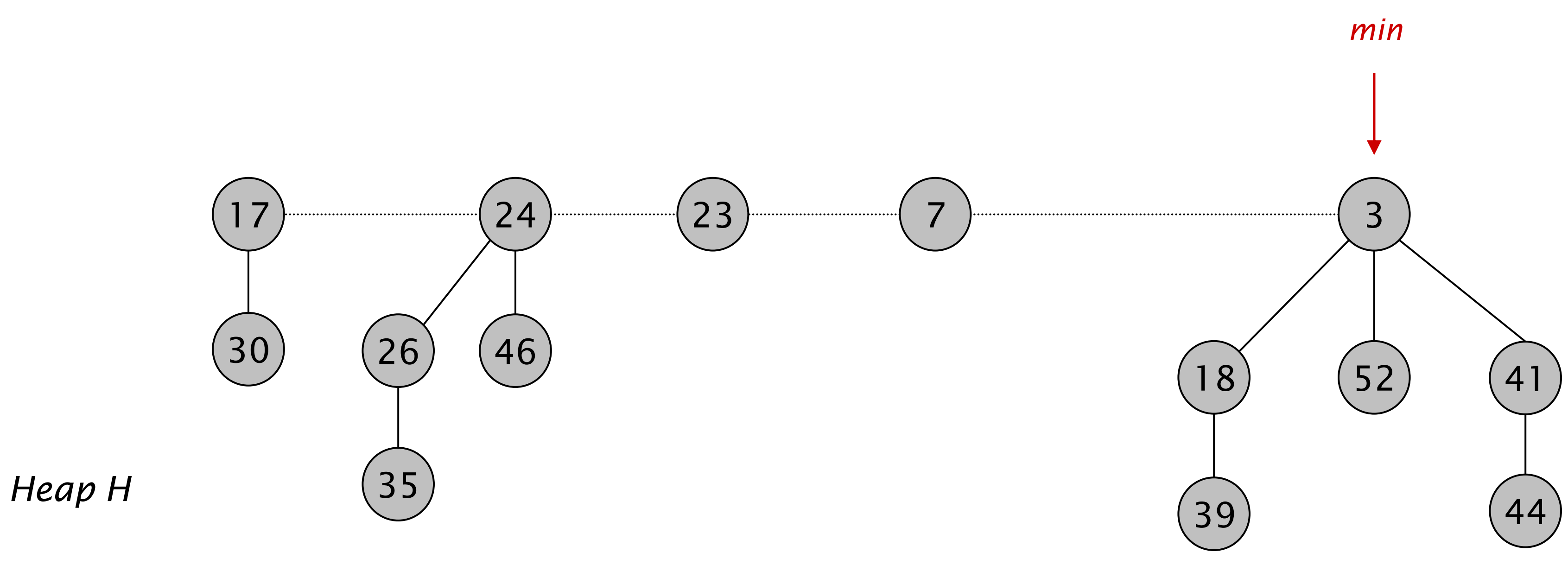


# Fibonacci Heaps: Structure

Fibonacci heap.

- Set of heap-ordered trees.
- Maintain pointer to minimum element.
- Set of marked nodes.

find-min takes O(1) time

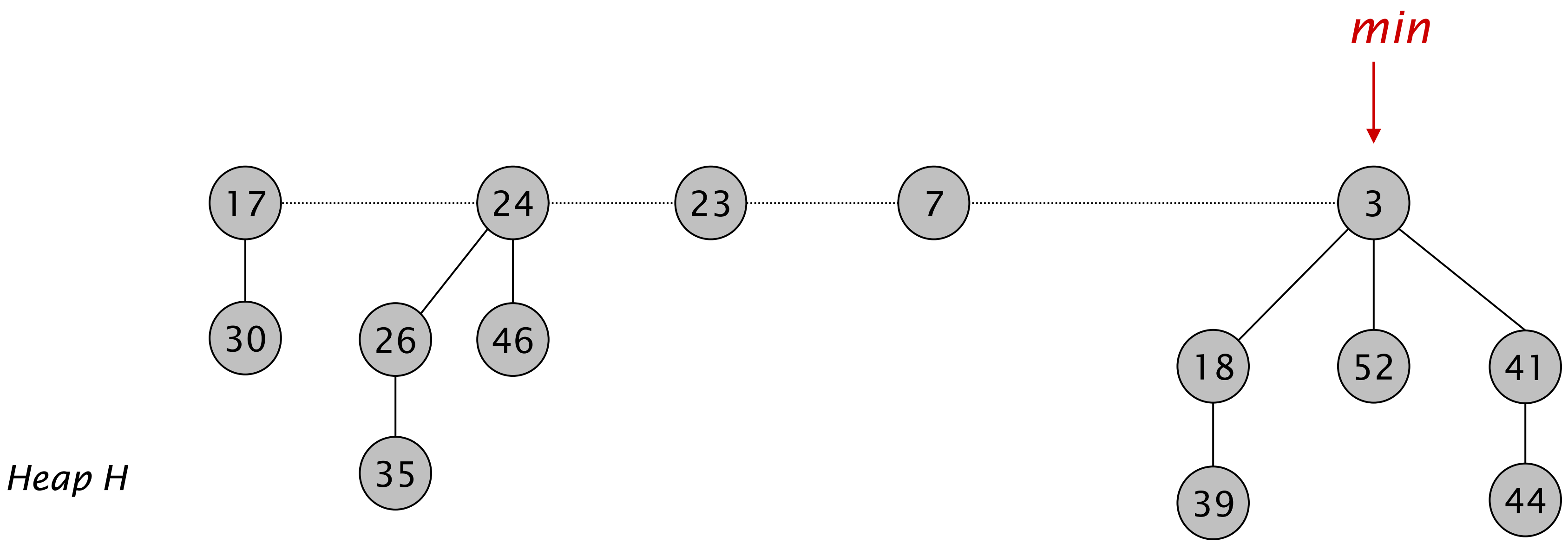


# Fibonacci Heaps: Structure

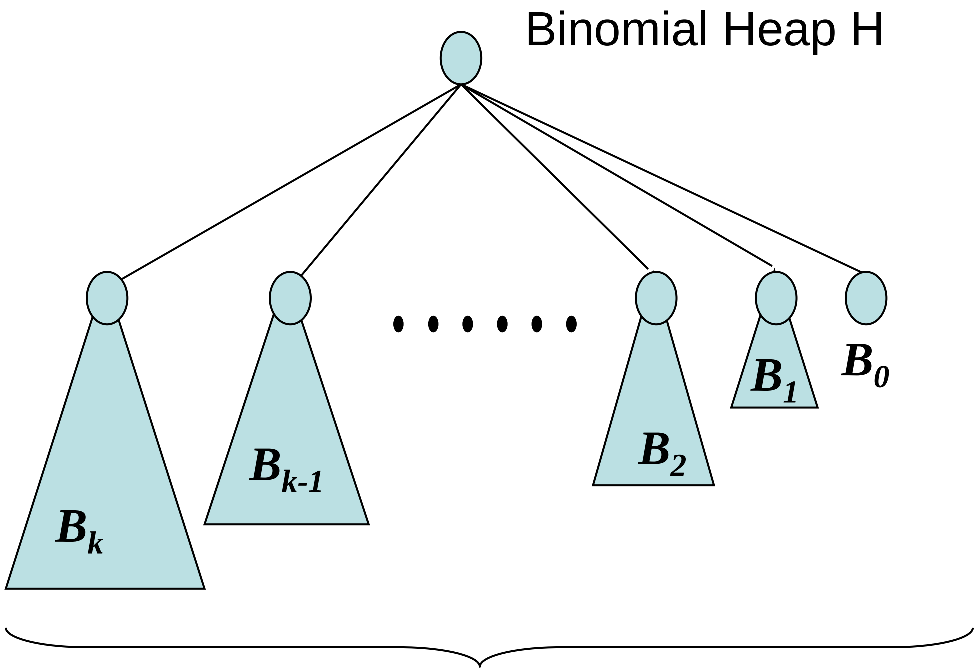
Fibonacci heap.

- Set of heap-ordered trees.
- Maintain pointer to minimum element.
- Set of marked nodes.

merge heaps takes O(1) time

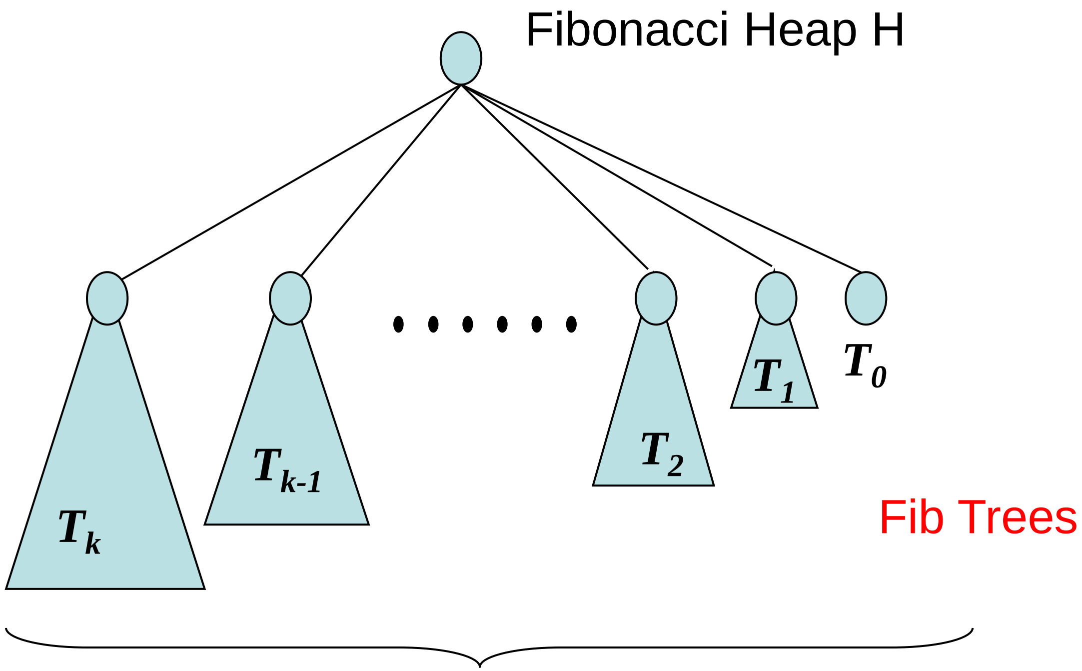


# Binomial Heap




Zero or one copies of binomial trees $B_k$

# Fibonacci Heap




~~Zero or one copies of binomial trees $B_k$~~
Can have many more trees
No longer binomial trees, but “defective” binomial trees
All trees satisfy the heap property

# Fib Trees

- Define rank of a tree, rank(T) as the number of children of the root of the tree T
- The Fib $F_k$ is a tree of rank(k) defined recursively

  $F_o$ Consists of a single node

  $F_1$ A single node with a single child

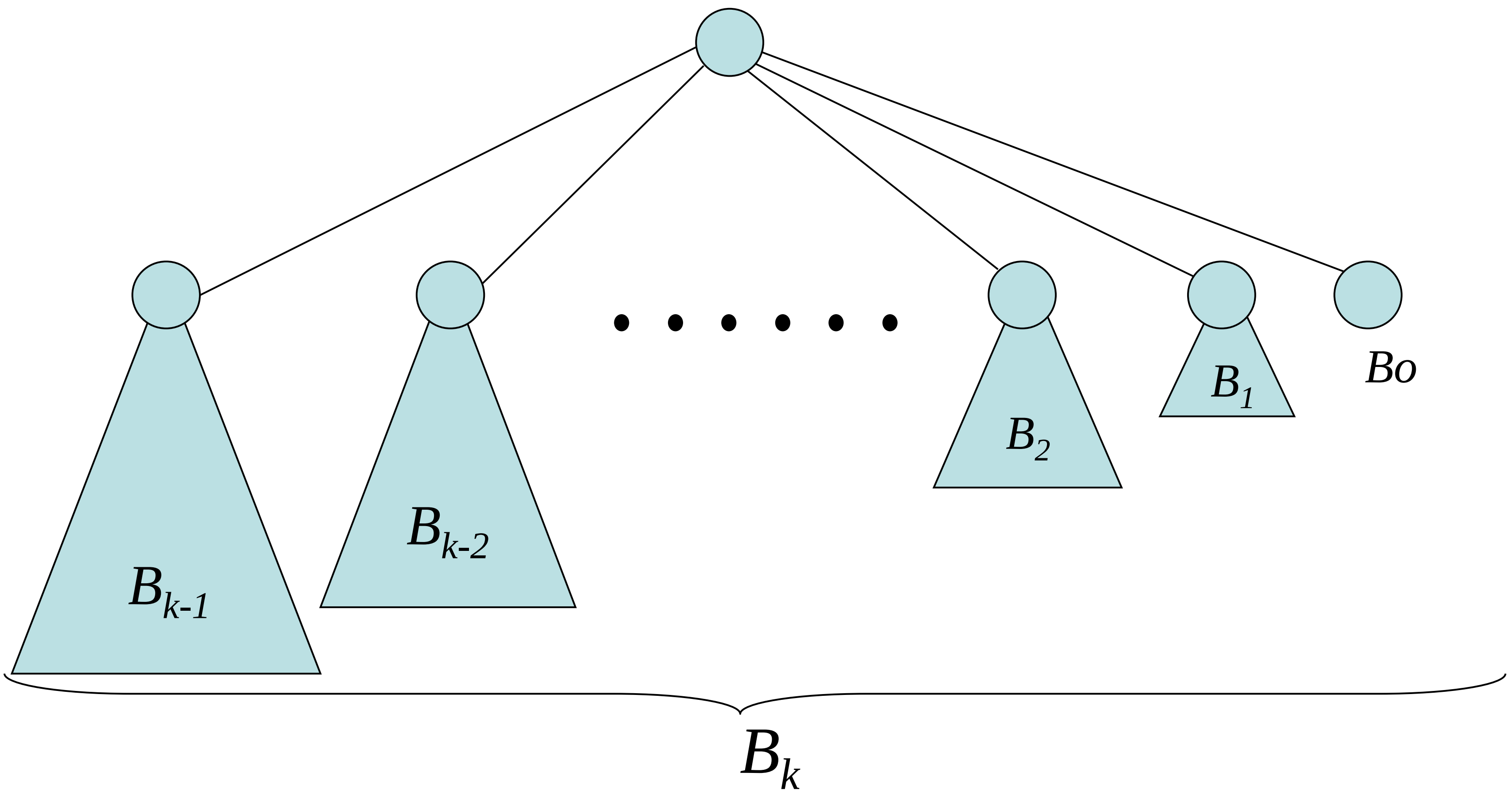
  .

  .

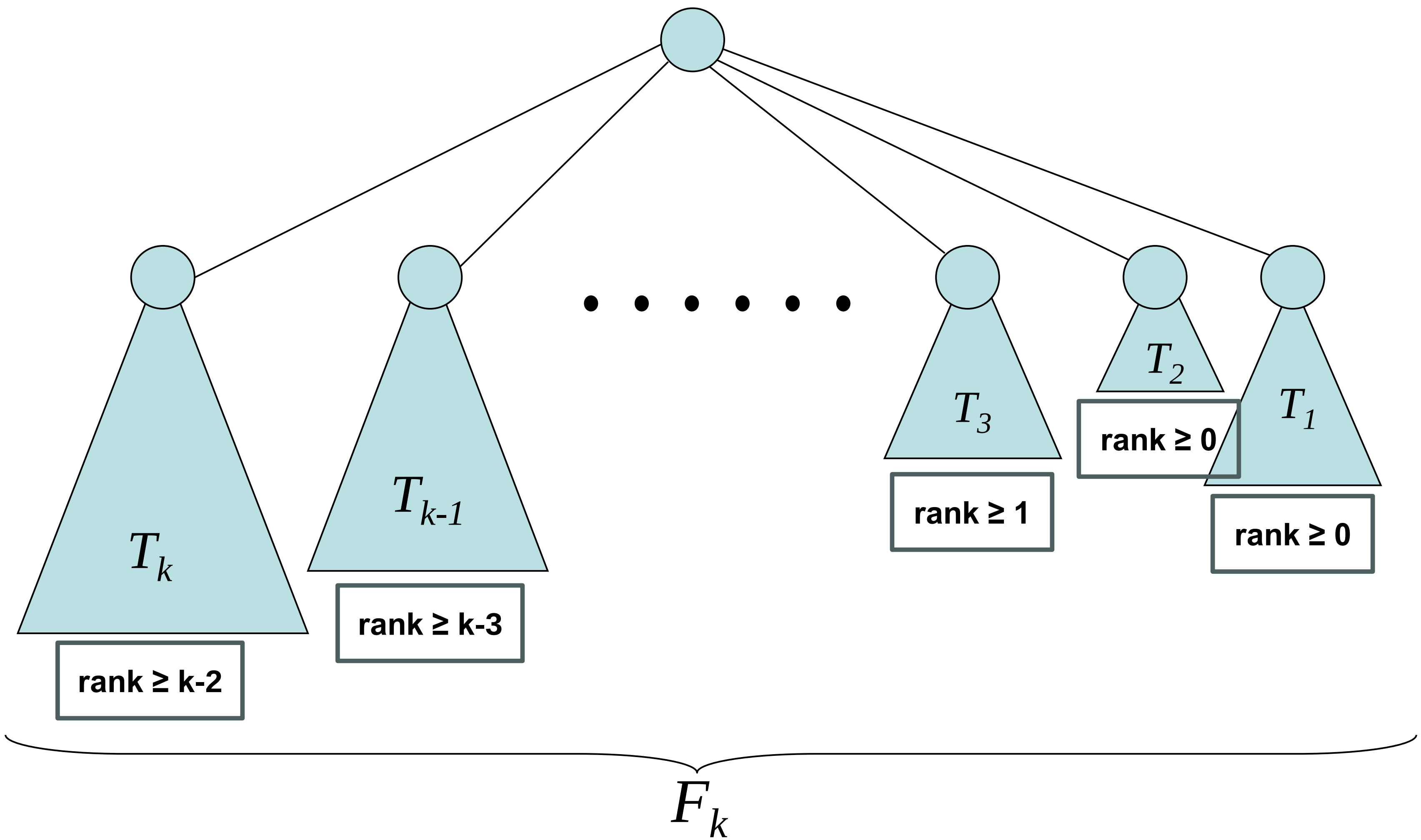
  $F_k$ is a Fib tree of rank k with children $T_1$, $T_2$, ..., $T_k$ such that

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

# Binomial Trees



# Fib Trees



# Properties of Fib Trees

*Define: size(T) as the number of nodes in T*

*D(T) = max degree of the nodes in T*

*Lemma:* For the Fib tree $F_k$

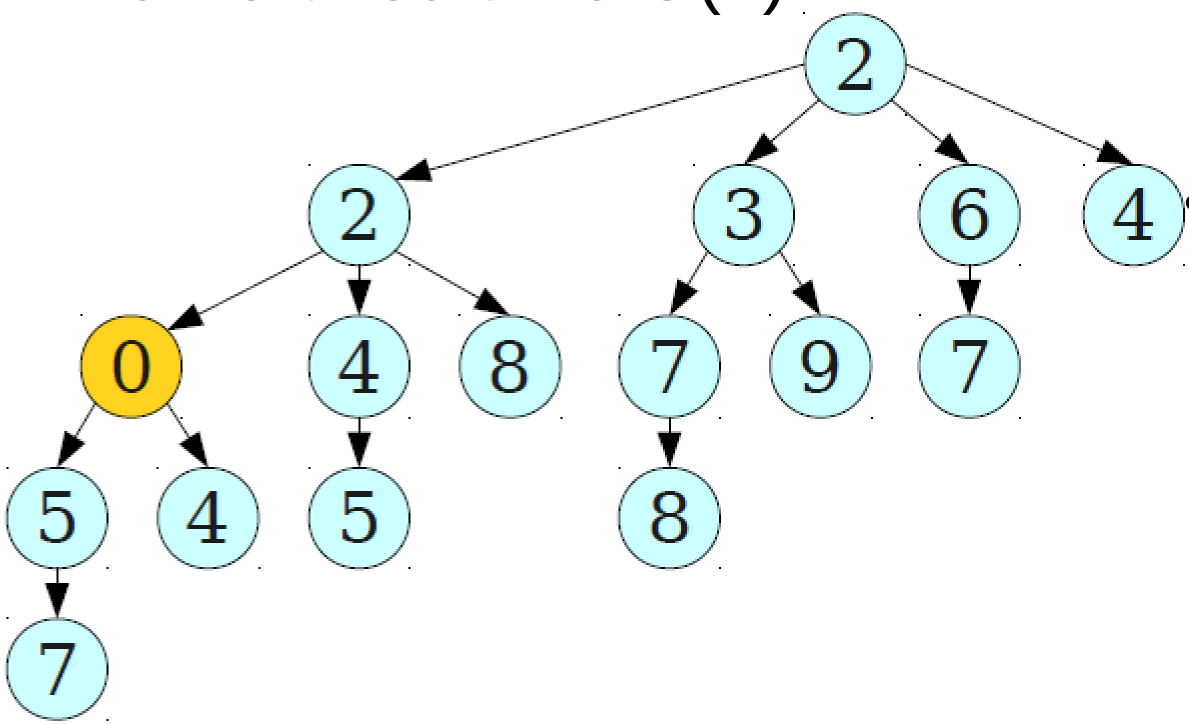
1. $\text{size}(F_k) \geq \phi^k$ where $\phi = (1 + \sqrt{5}) / 2$
2. $D(F_k) \leq \log_\phi(\text{size}(F_k))$

# Why Fib Trees?

- Goal: Implement decrease-key in amortized time O(1)

# Why Fib Trees?

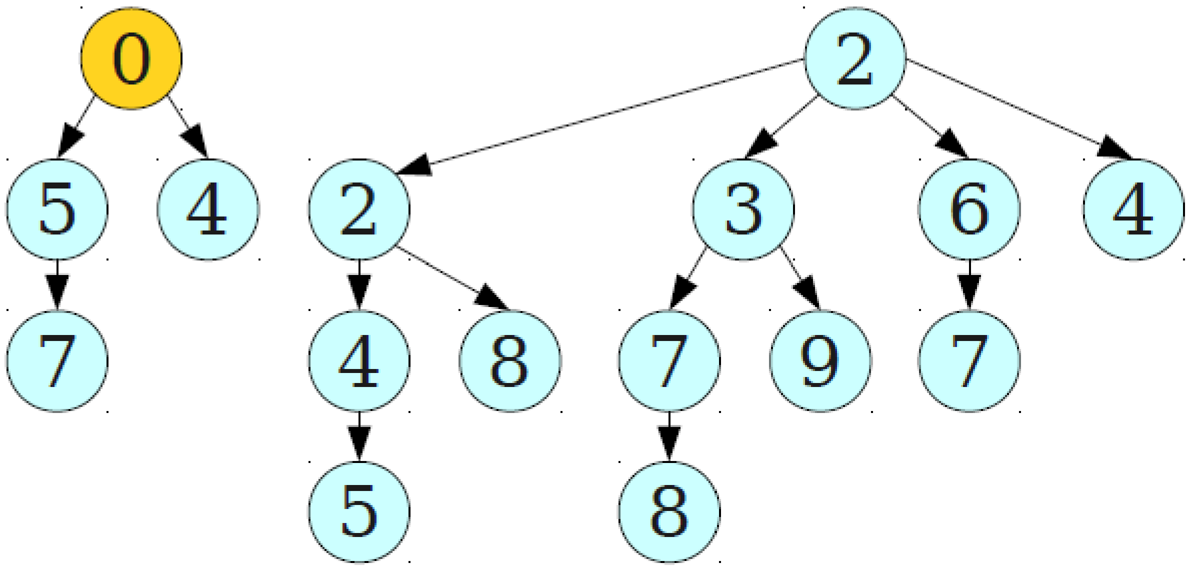
- Goal: Implement decreaseKey in amortized time O(1)




- Naive implementation
- Compare with the parent and bubble up
- May need O(log(n)) time

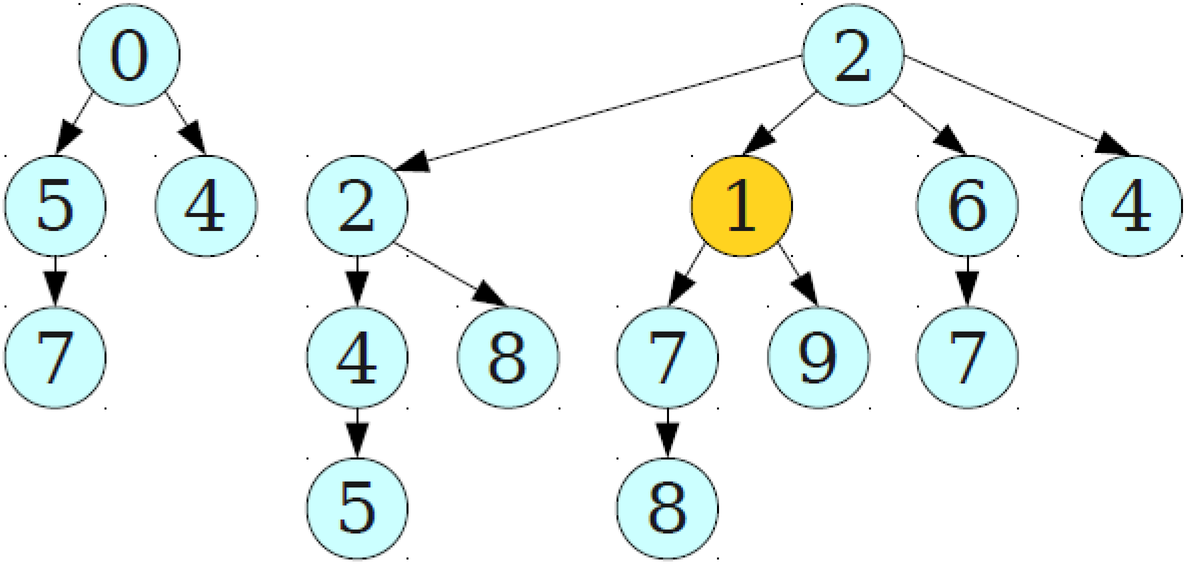
# Our Idea

- Goal: Implement decreaseKey in amortized time O(1)



# Our Idea

- Goal: Implement decreaseKey in amortized time O(1)

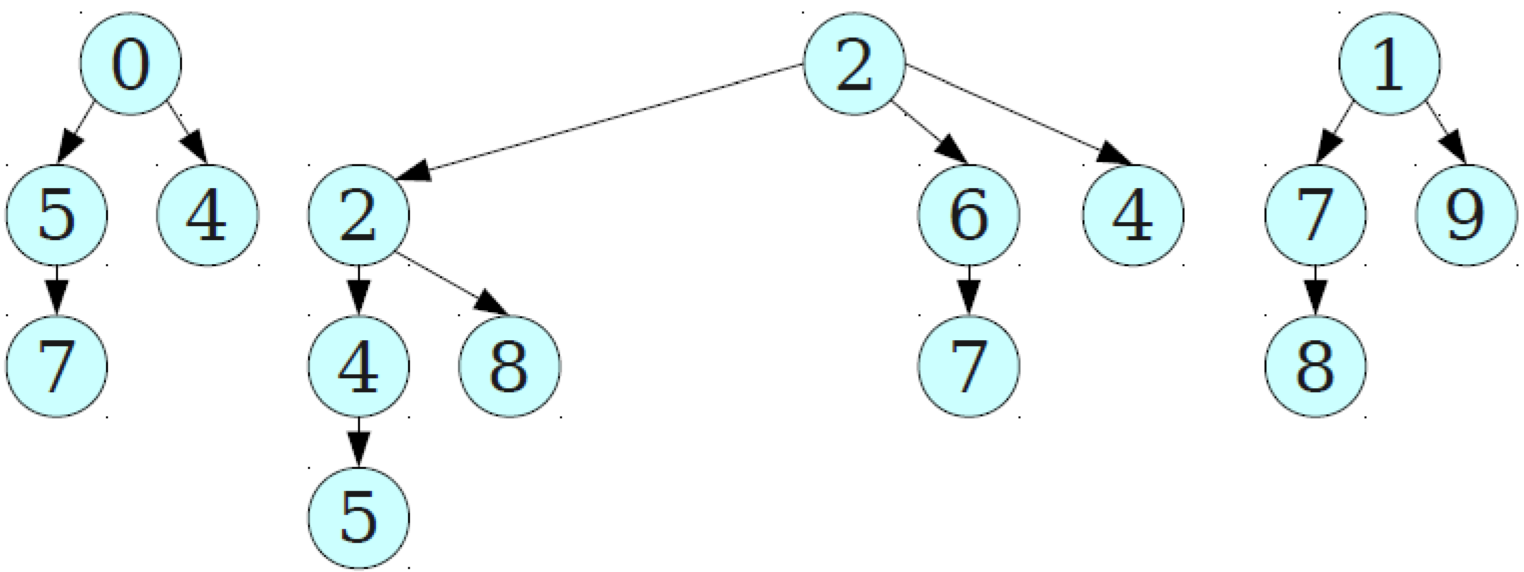


# Our Idea

- Goal: Implement decreaseKey in amortized time O(1)

decreaseKey(x)

- Cut the node x and move subtree to root
- O(1) operation



# decreaseKey

- To implement ***decrease-key*** efficiently:
  - Lower the key of the specified node
  - If its key is greater than or equal to its parent's key, we're done
  - Otherwise, cut that node from its parent and hoist it up to the root list, optionally updating the min pointer
  - Time required: O(1)
- This requires some changes to the tree representation

# Fibonacci Heaps: Structure

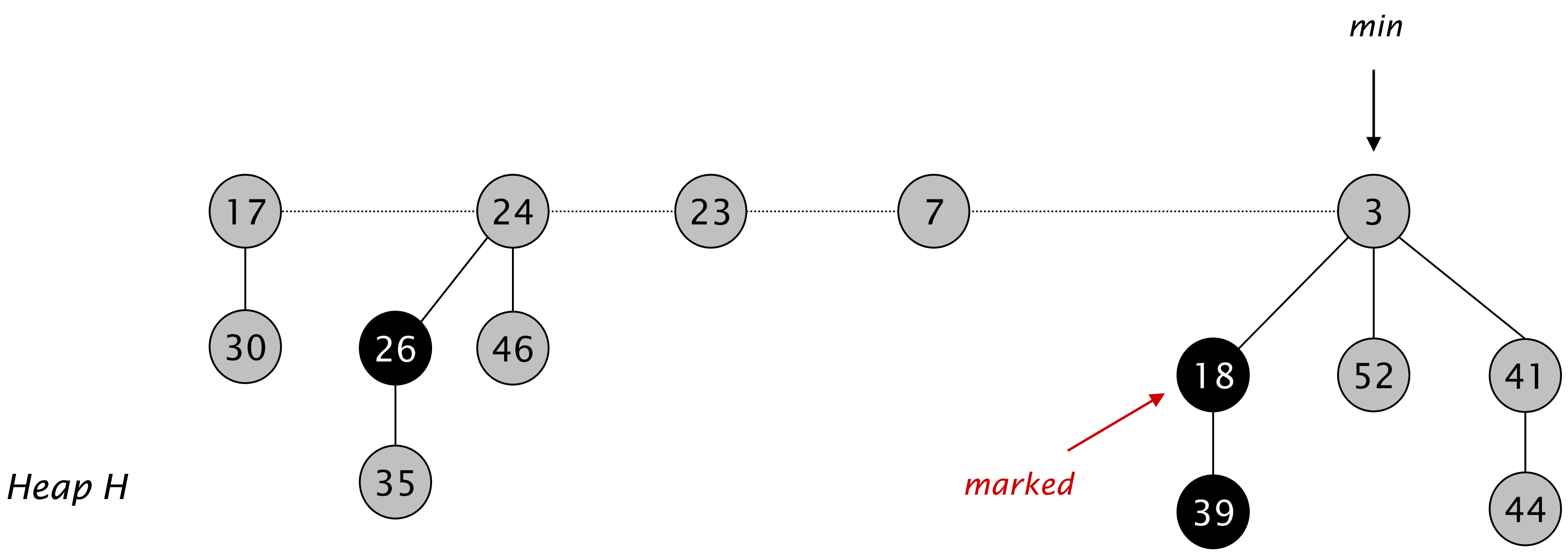
## Fibonacci heap.

- Set of heap-ordered trees.
- Maintain pointer to minimum element.
- Set of marked nodes.

If a node's child has been cut it is marked

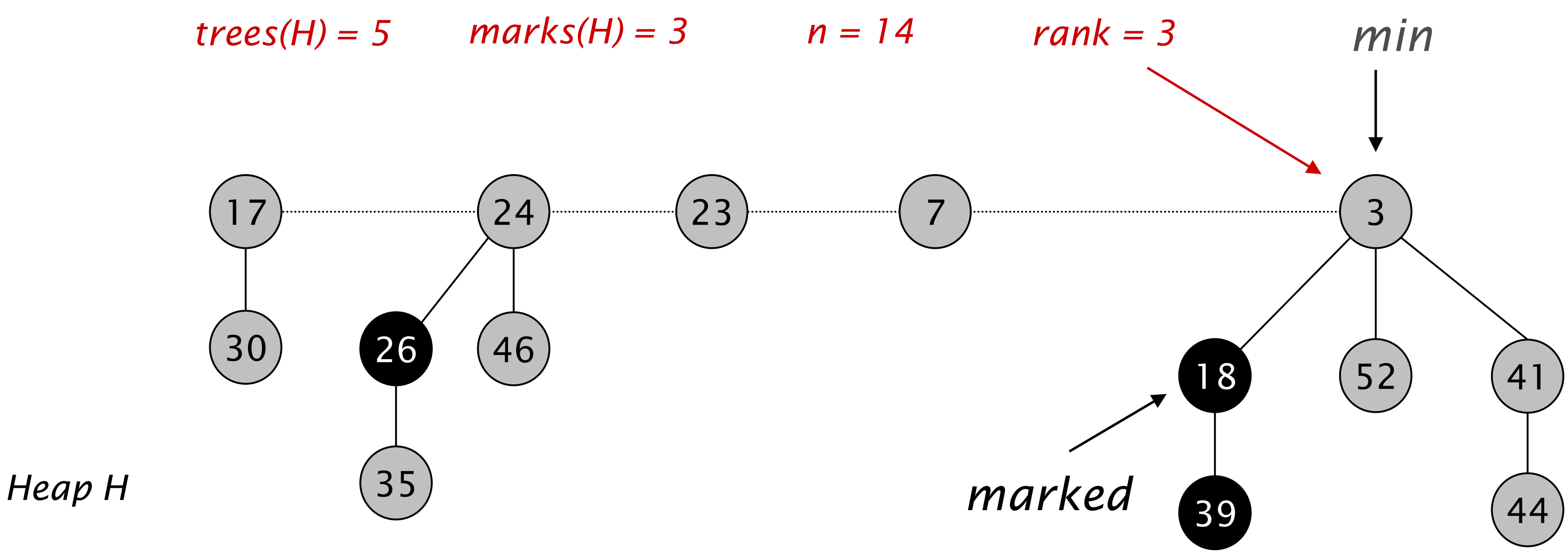
No node can have more than one child cut

Used to keep heaps flat (stay tuned)



## Notation.

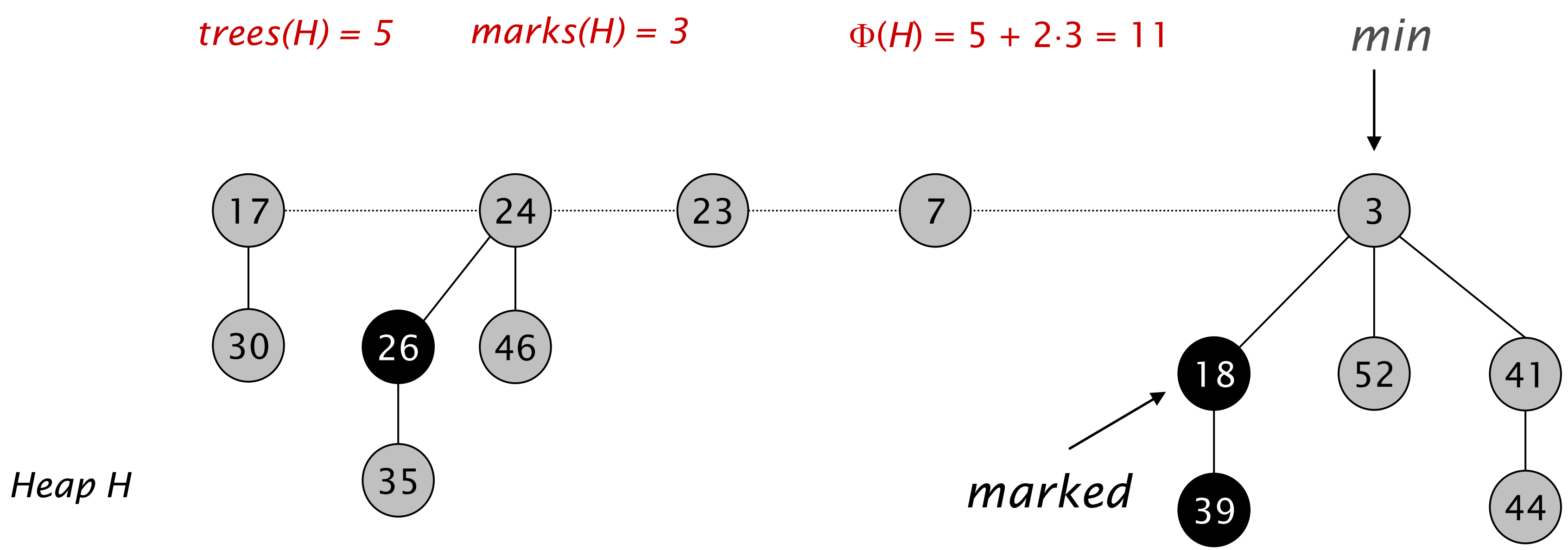
- $n$ = number of nodes in heap.
- $rank(x)$ = number of children of node $x$.
- $rank(H)$ = max rank of any node in heap $H$.
- $trees(H)$ = number of trees in heap $H$.
- $marks(H)$ = number of marked nodes in heap $H$.



# Fibonacci Heaps: Potential Function

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

*potential of heap H*

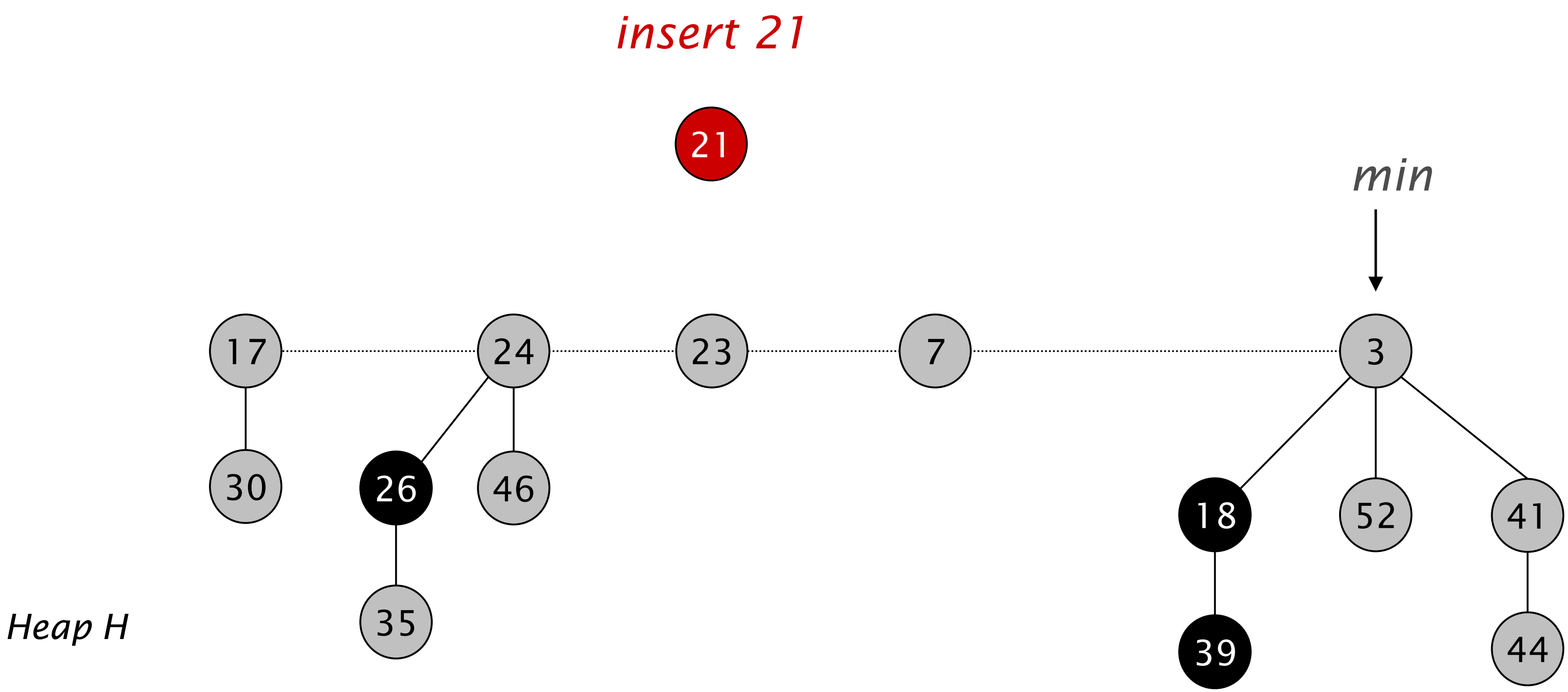


# Insert

# Fibonacci Heaps: Insert

Insert.

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

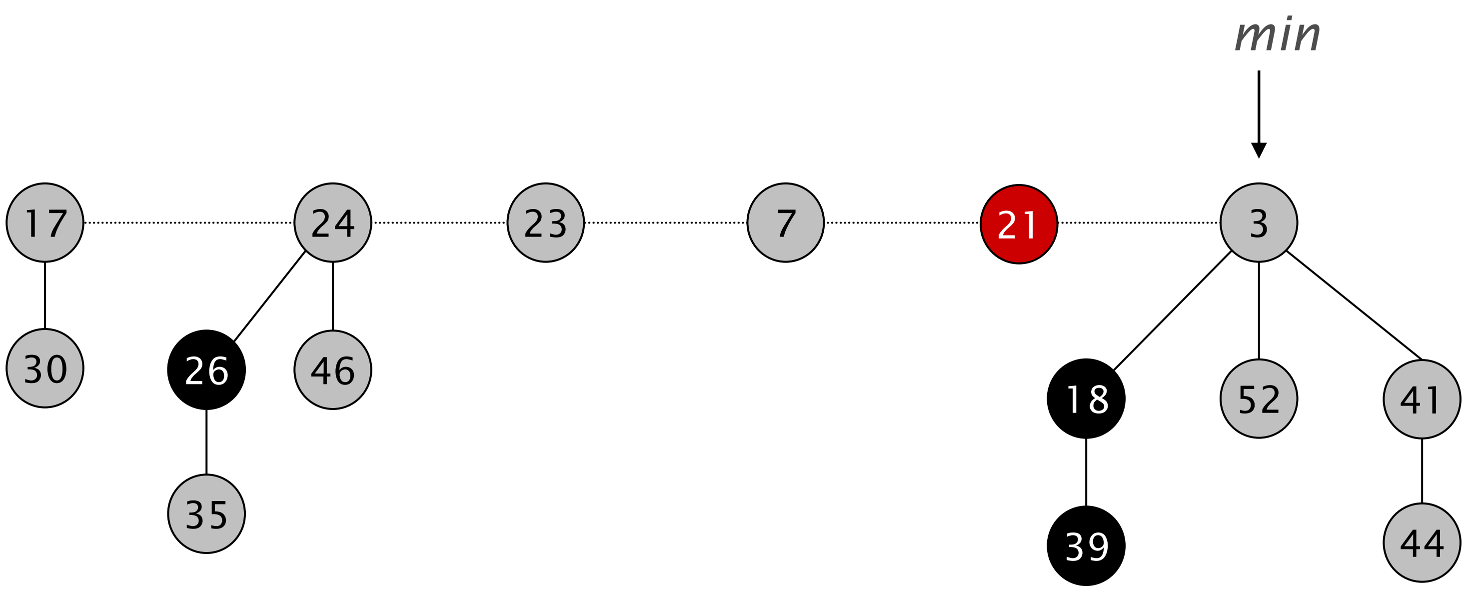


# Fibonacci Heaps: Insert

Insert.

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

*insert 21*




*Heap H*

# Fibonacci Heaps: Insert Analysis

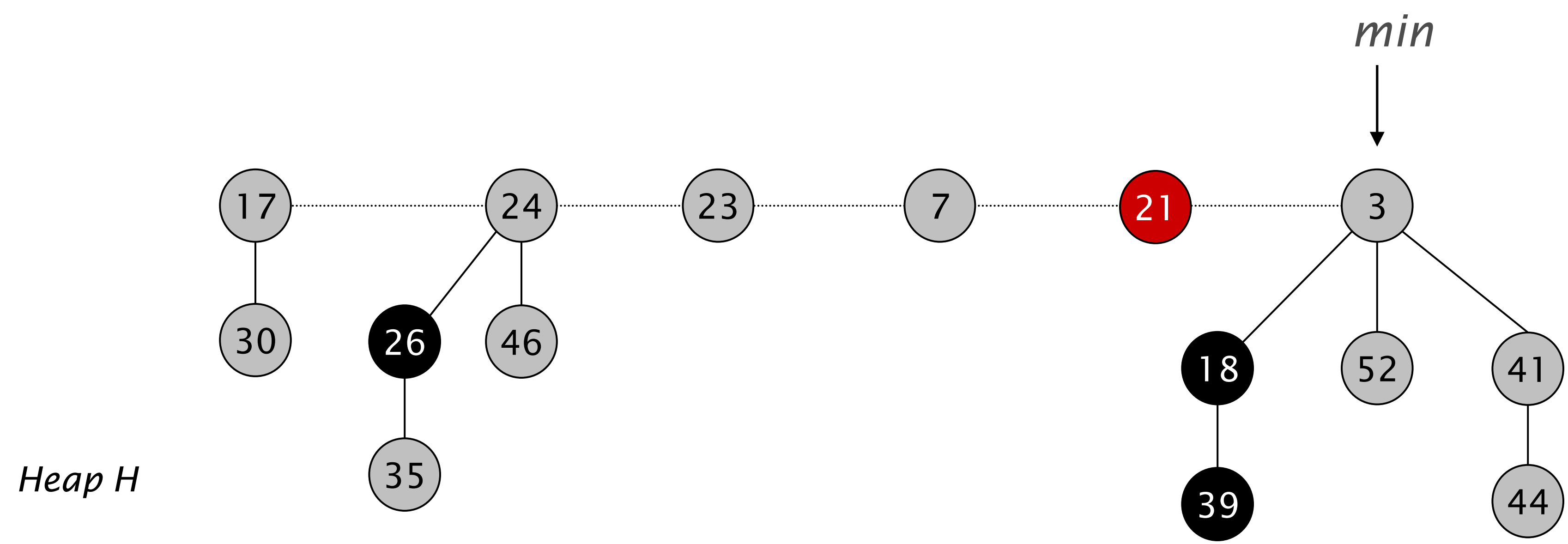
**Actual cost.** O(1)

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

*potential of heap H*

**Change in potential.** +1

**Amortized cost.** O(1)



# Delete Min

# Delete Min

We merge all the trees in the Heap in the deleteMin operation

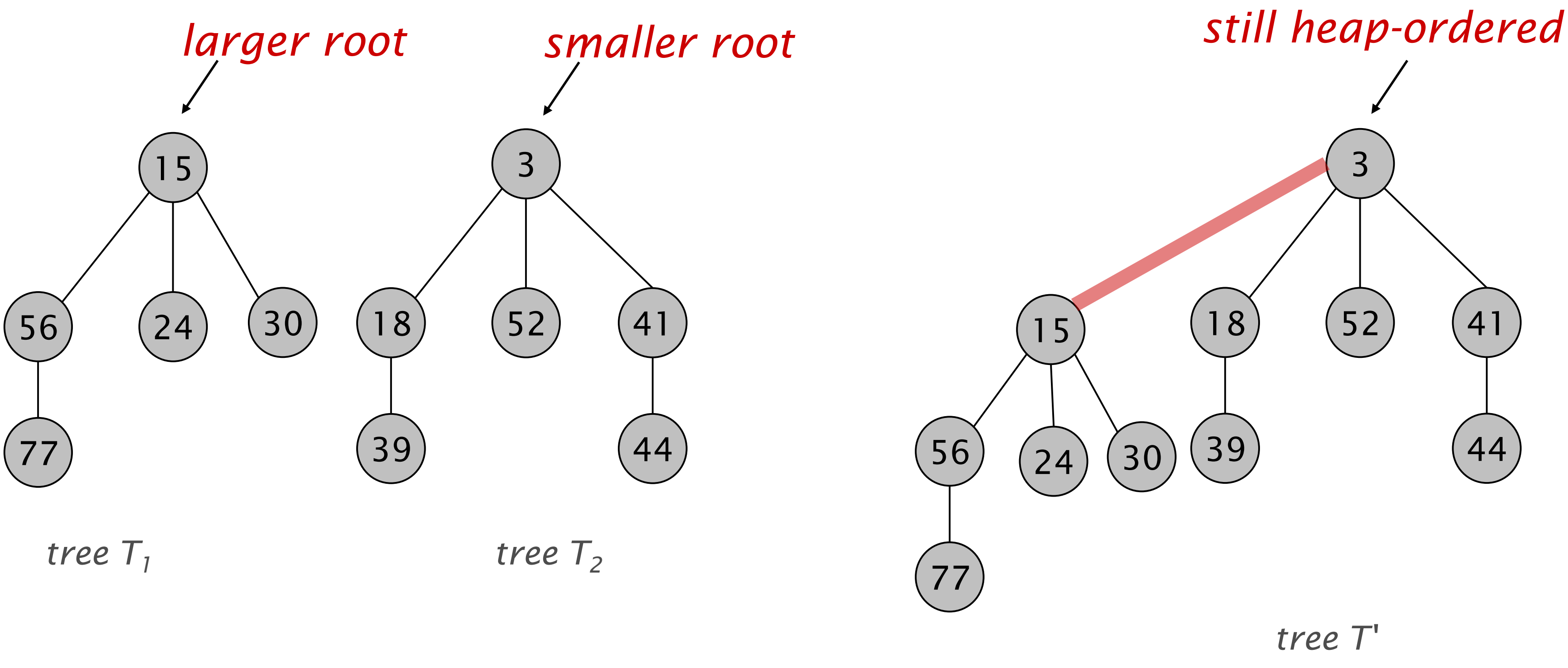
Only merge trees of the same rank

Recall rank(T) is the number of children in the root of T

Make the larger root the child of the smaller root

# Linking Operation

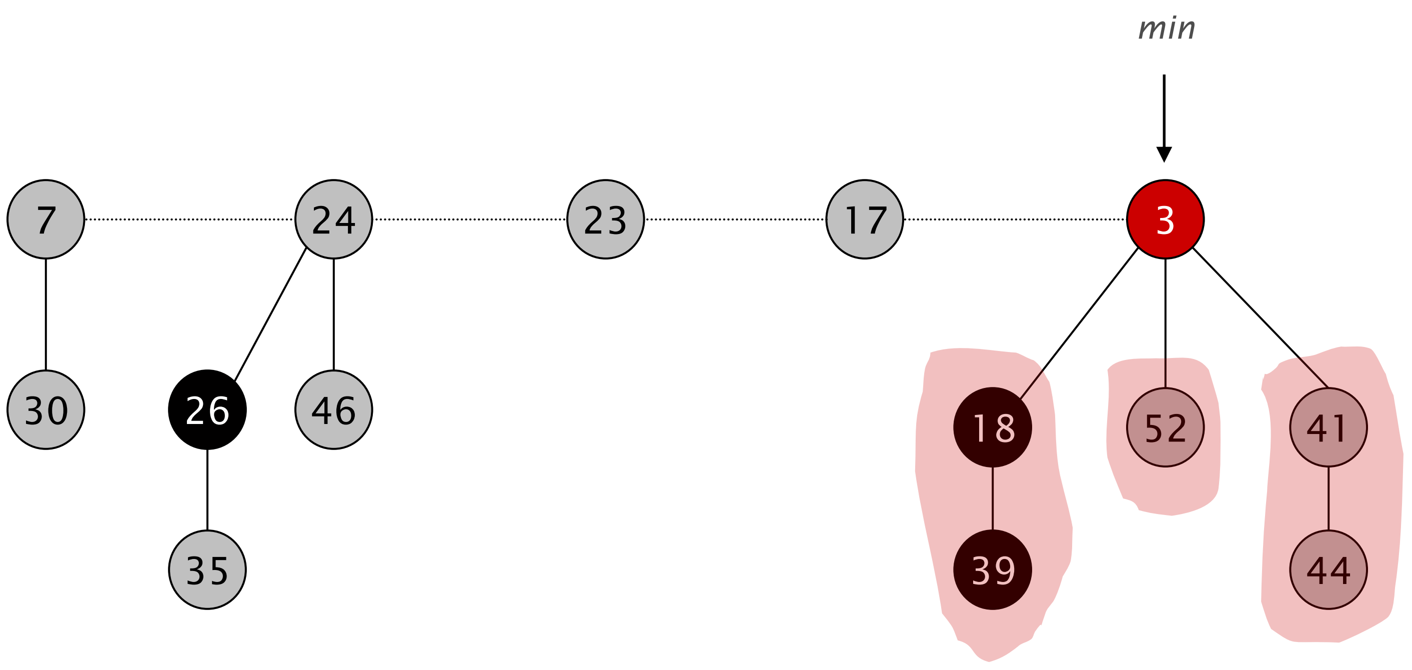
**Linking operation.** Make larger root be a child of smaller root.



# Fibonacci Heaps: Delete Min

**Delete min.**

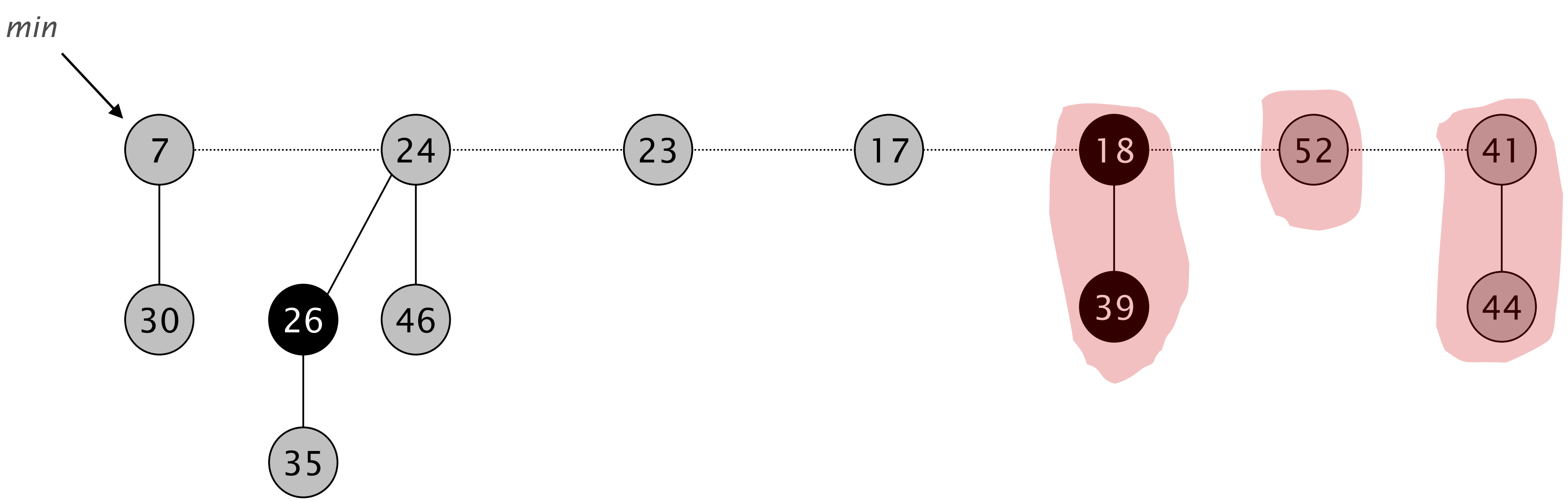
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



# Fibonacci Heaps: Delete Min

## Delete min.

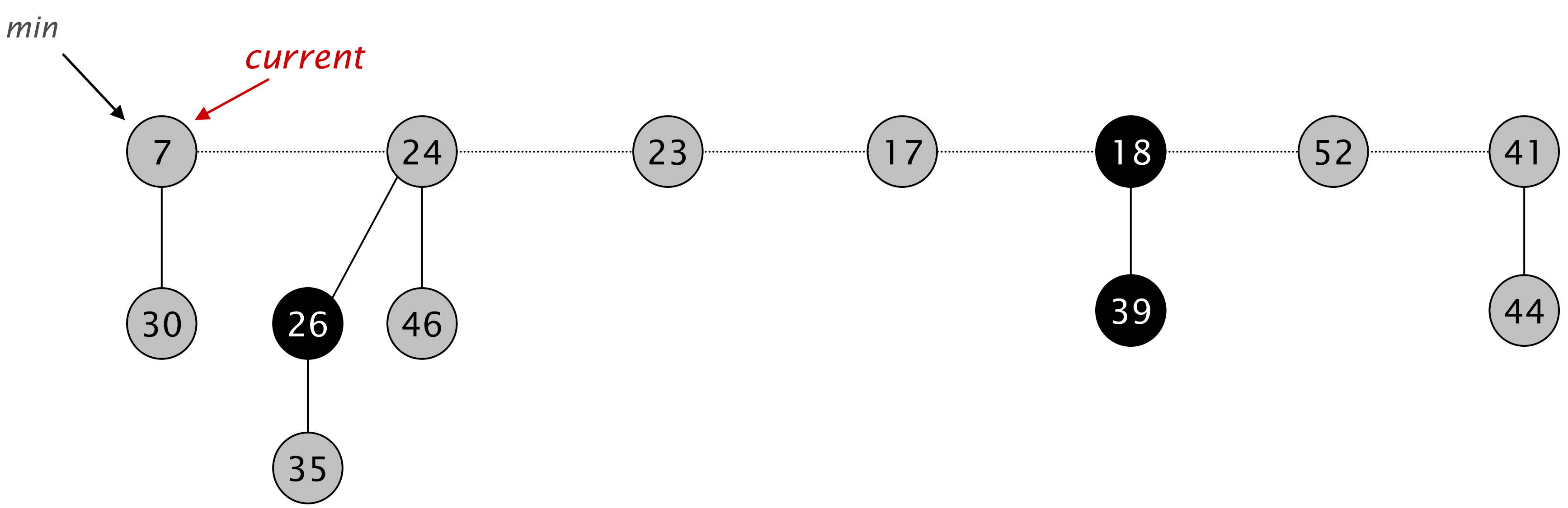
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



# Fibonacci Heaps: Delete Min

## Delete min.

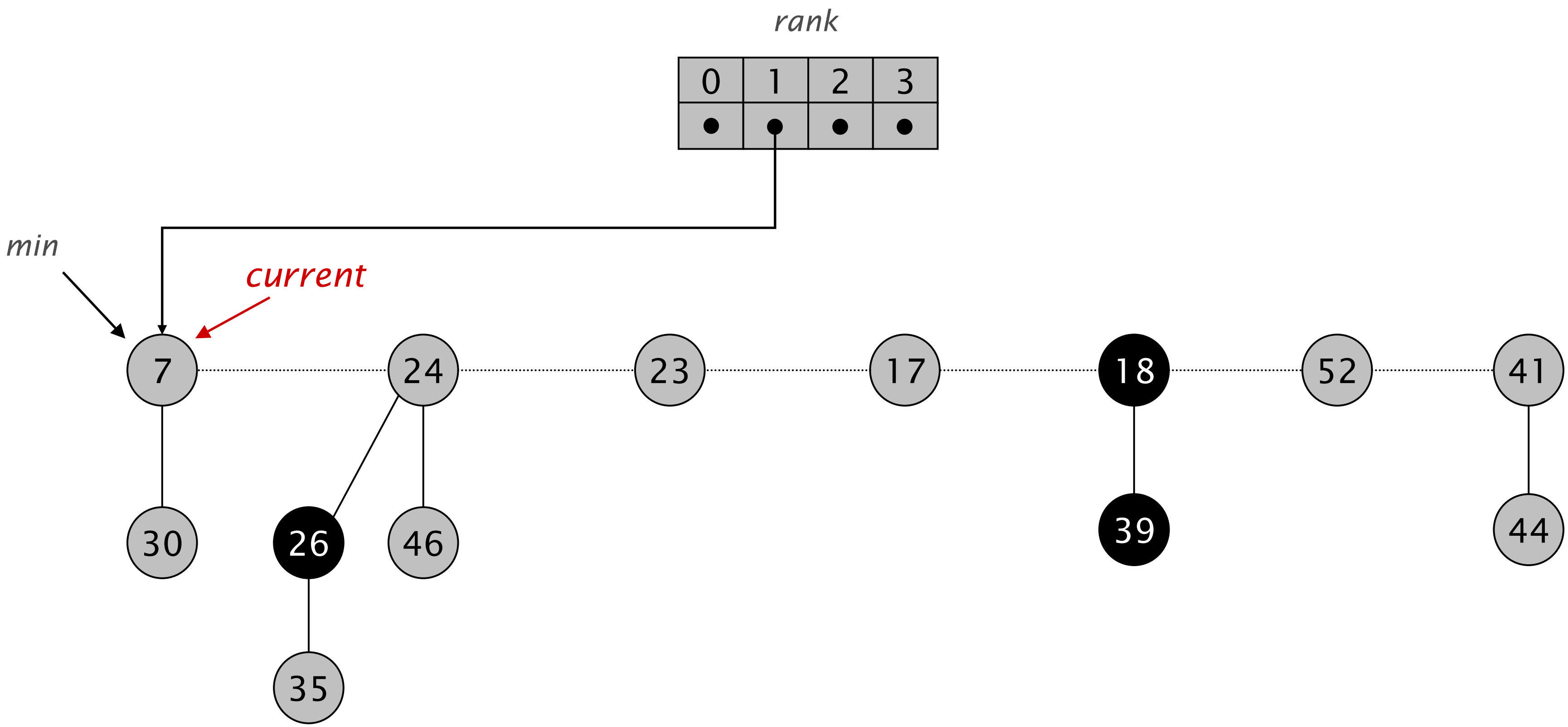
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



# Fibonacci Heaps: Delete Min

## Delete min.

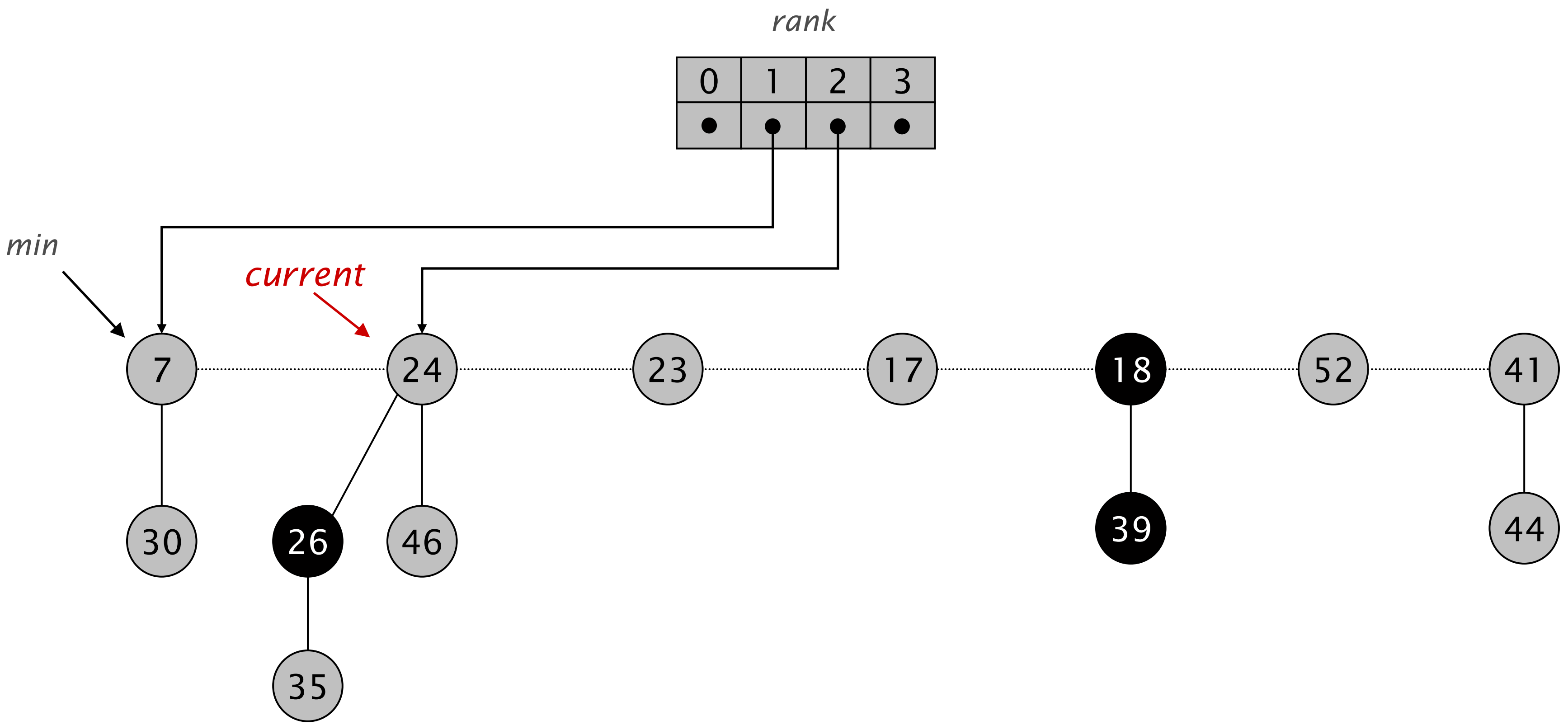
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



# Fibonacci Heaps: Delete Min

## Delete min.

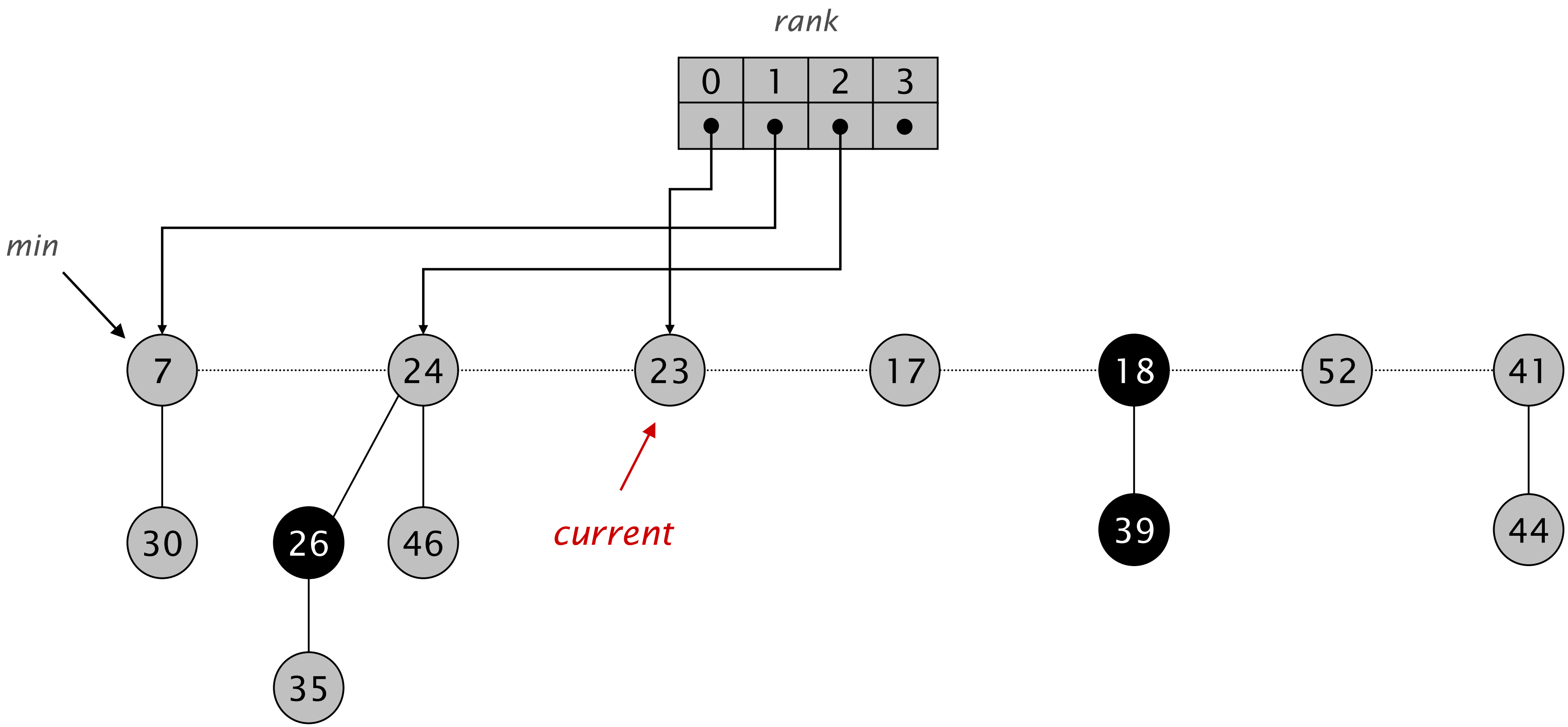
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



# Fibonacci Heaps: Delete Min

## Delete min.

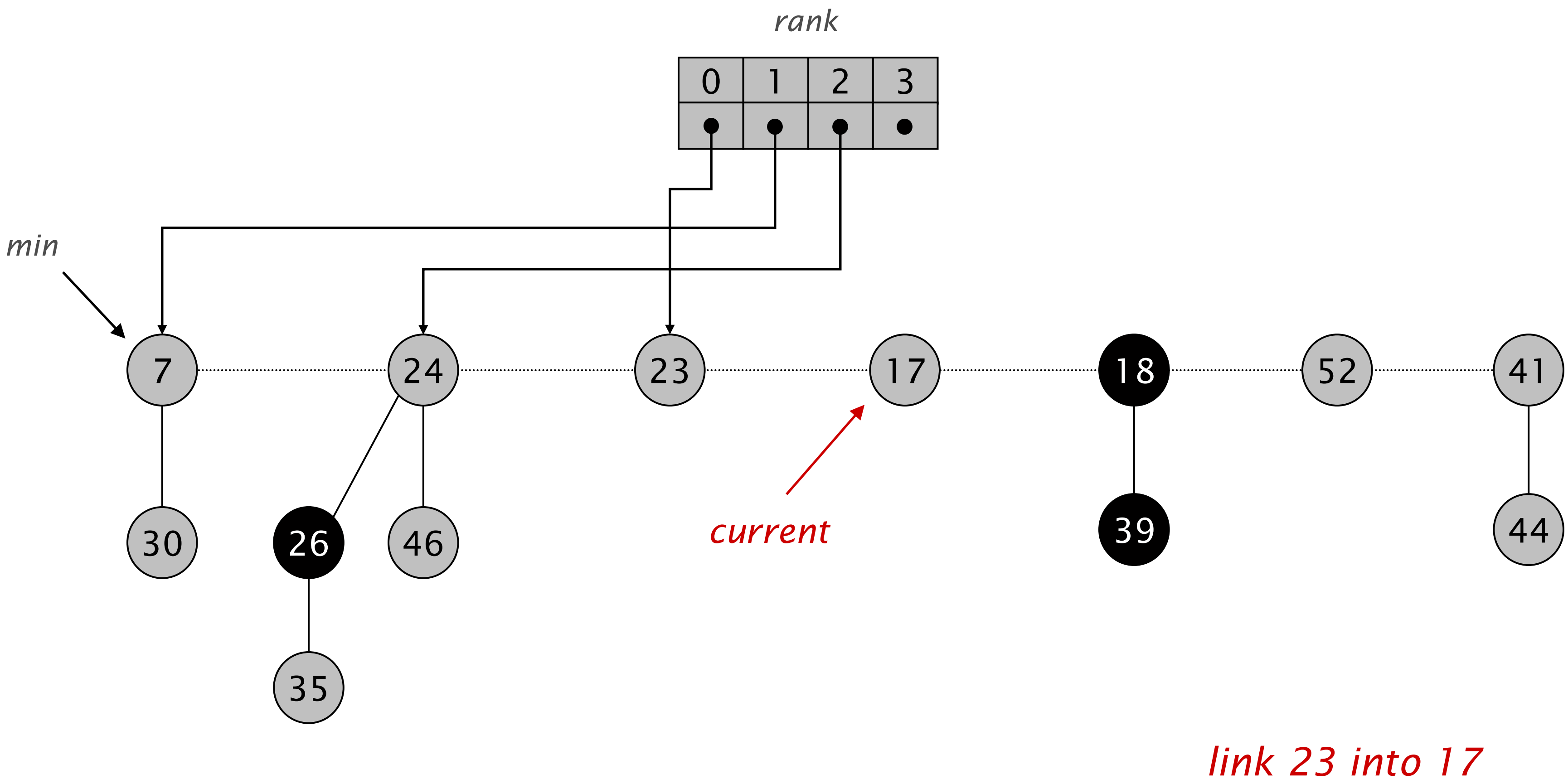
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



# Fibonacci Heaps: Delete Min

Delete min.

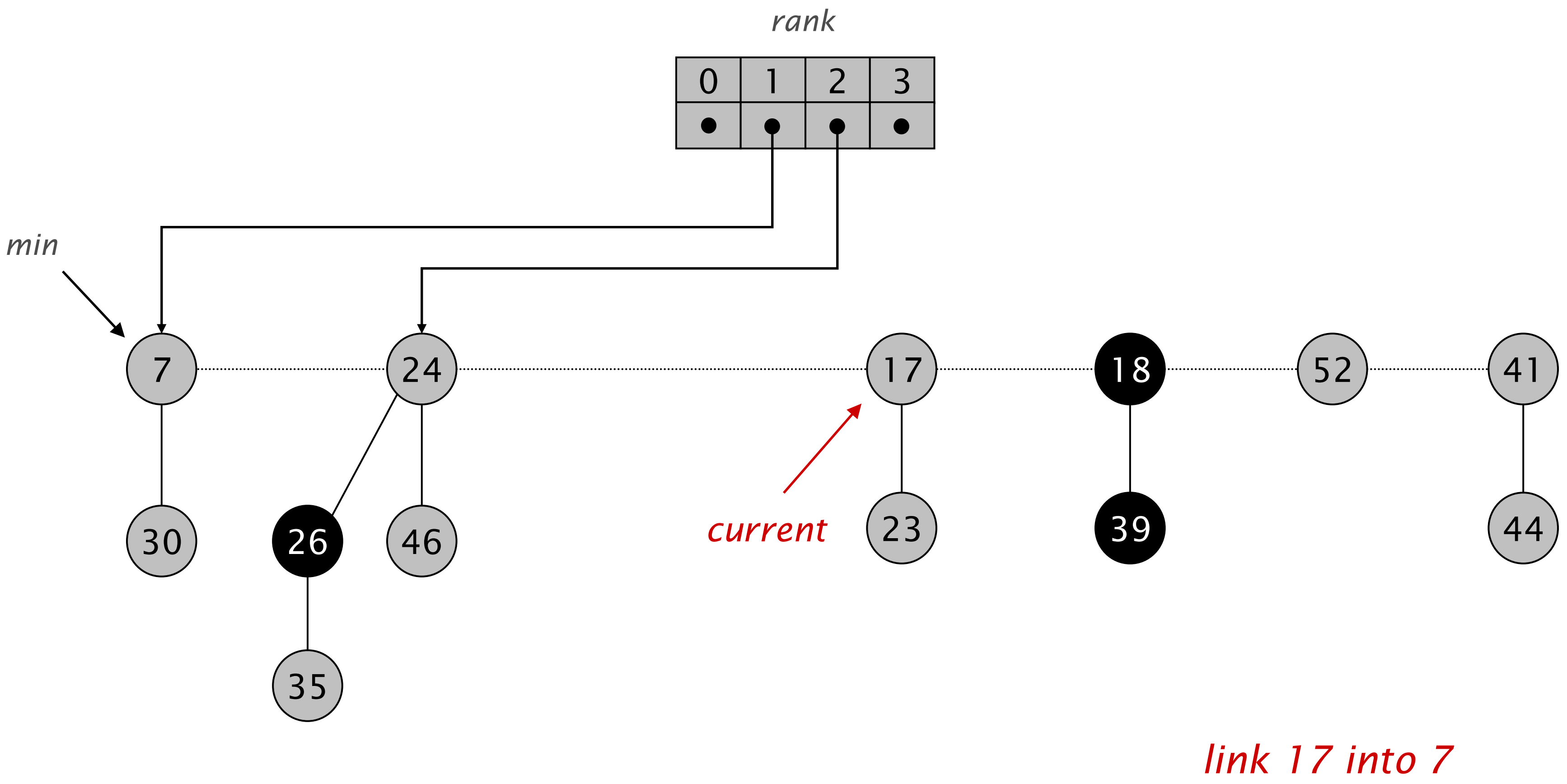
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



# Fibonacci Heaps: Delete Min

## Delete min.

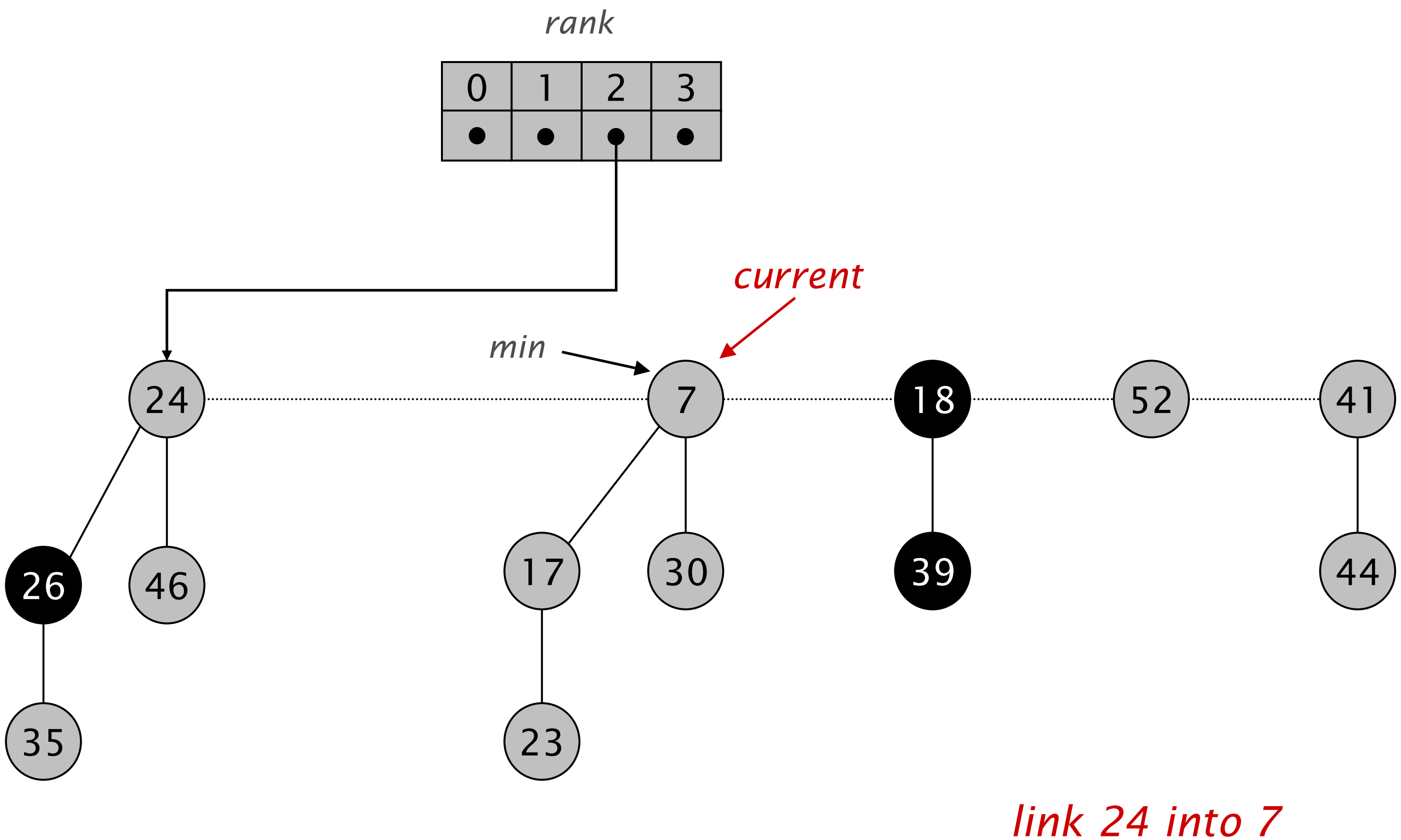
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



# Fibonacci Heaps: Delete Min

## Delete min.

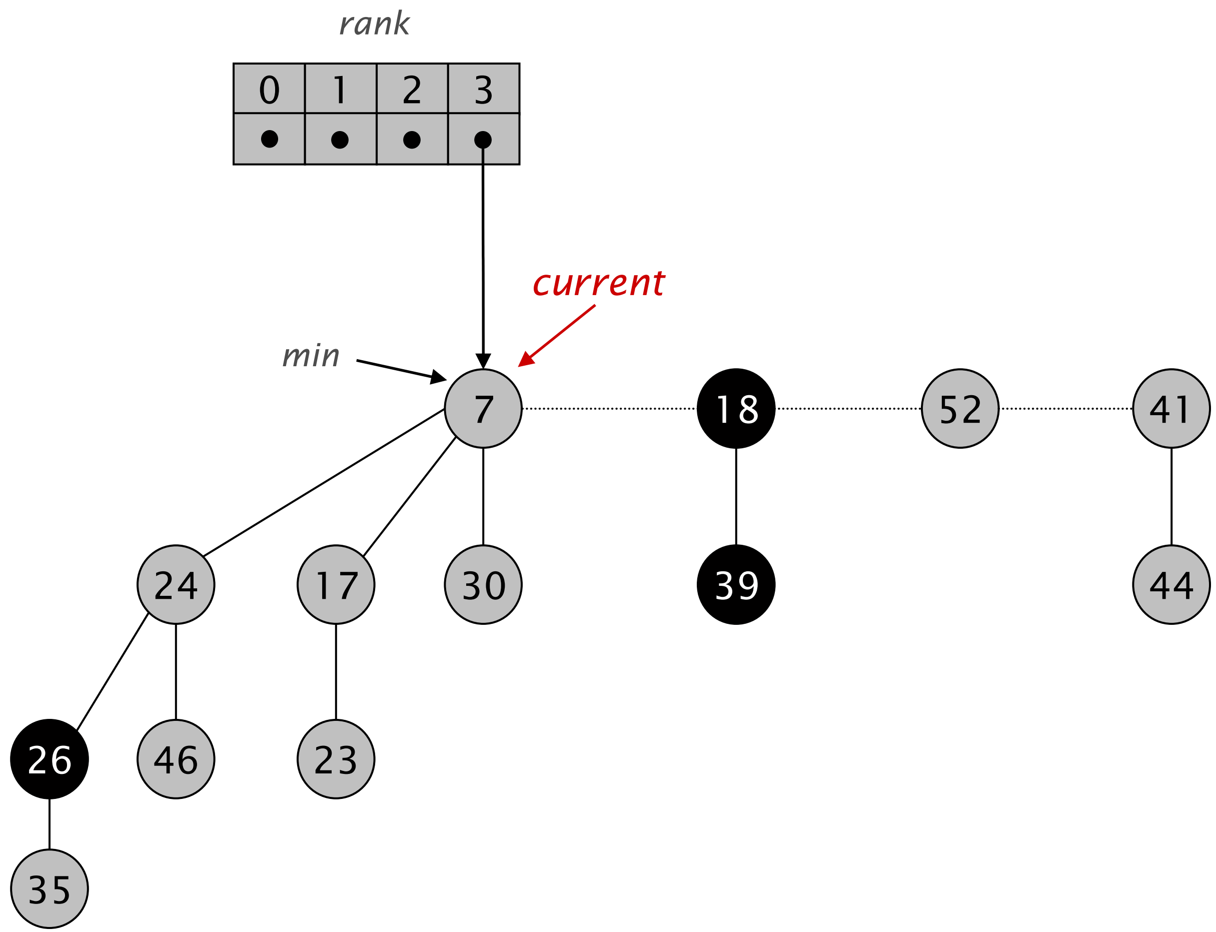
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



# Fibonacci Heaps: Delete Min

## Delete min.

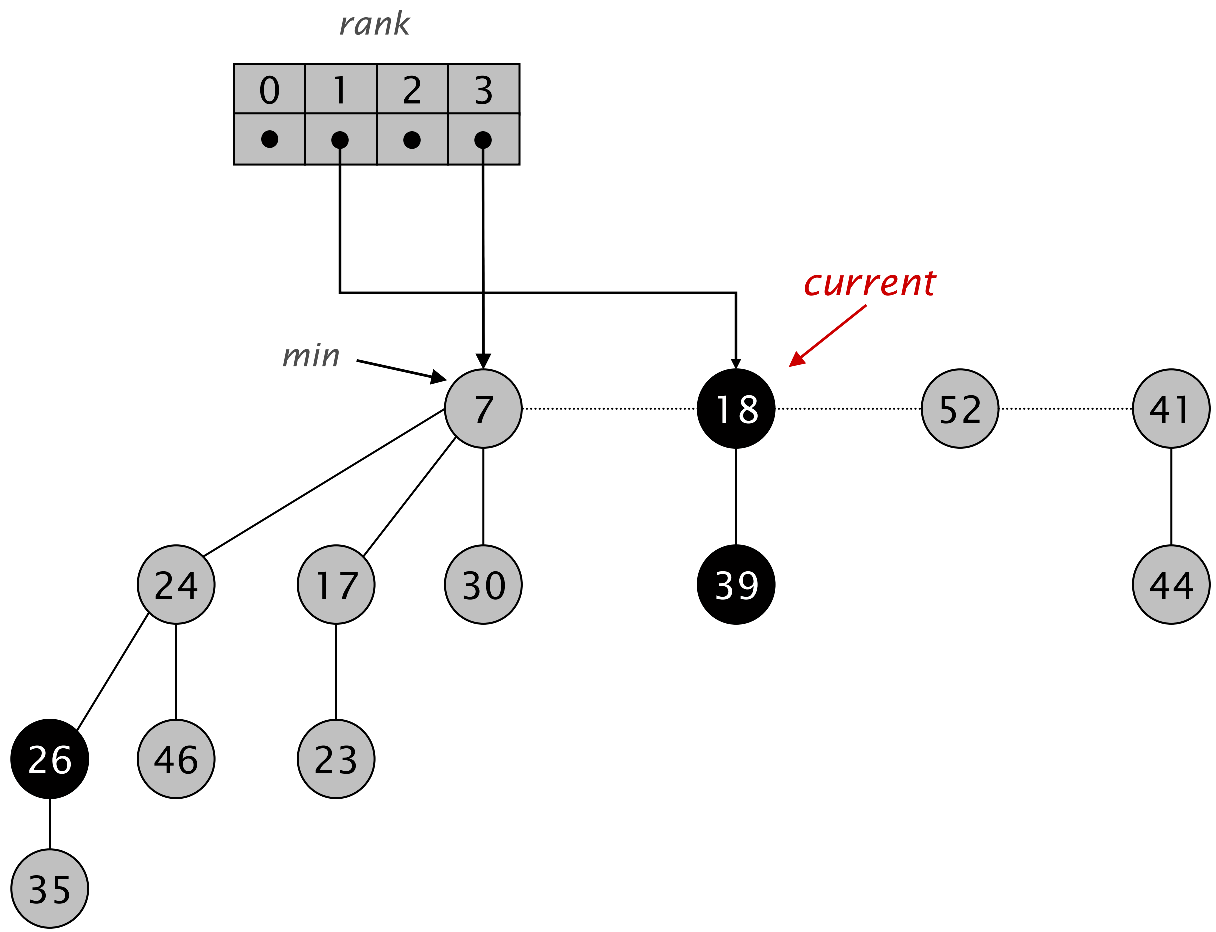
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



# Fibonacci Heaps: Delete Min

## Delete min.

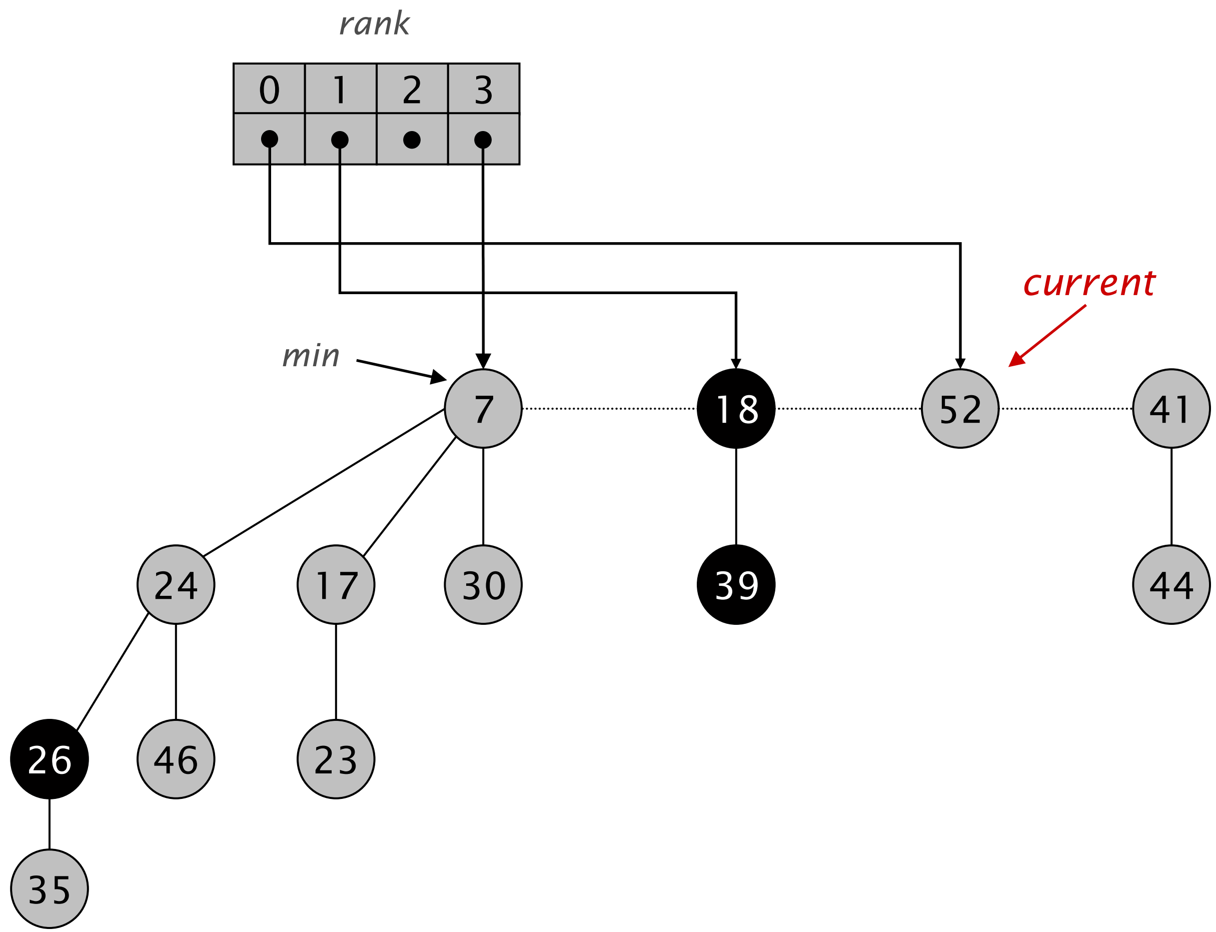
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



# Fibonacci Heaps: Delete Min

## Delete min.

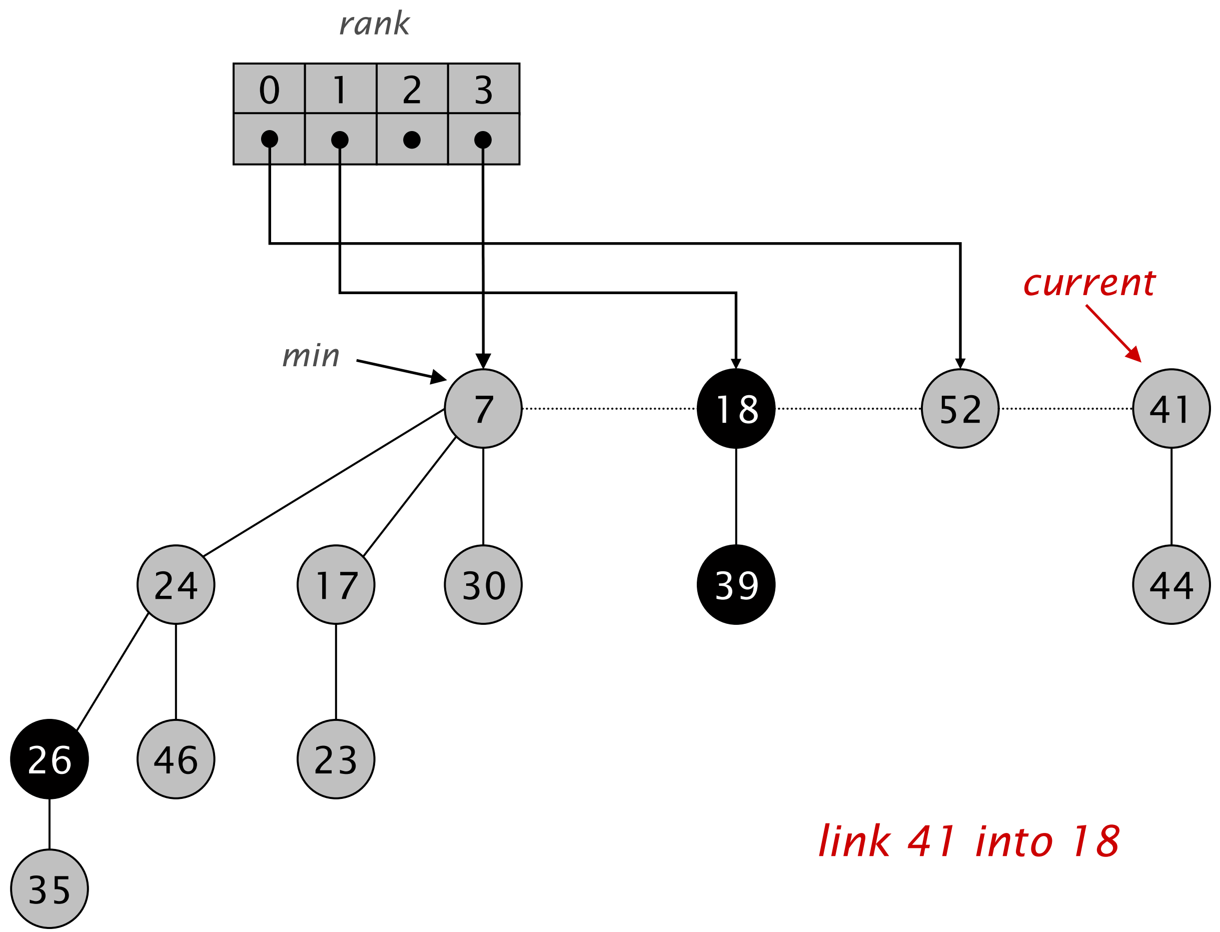
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



# Fibonacci Heaps: Delete Min

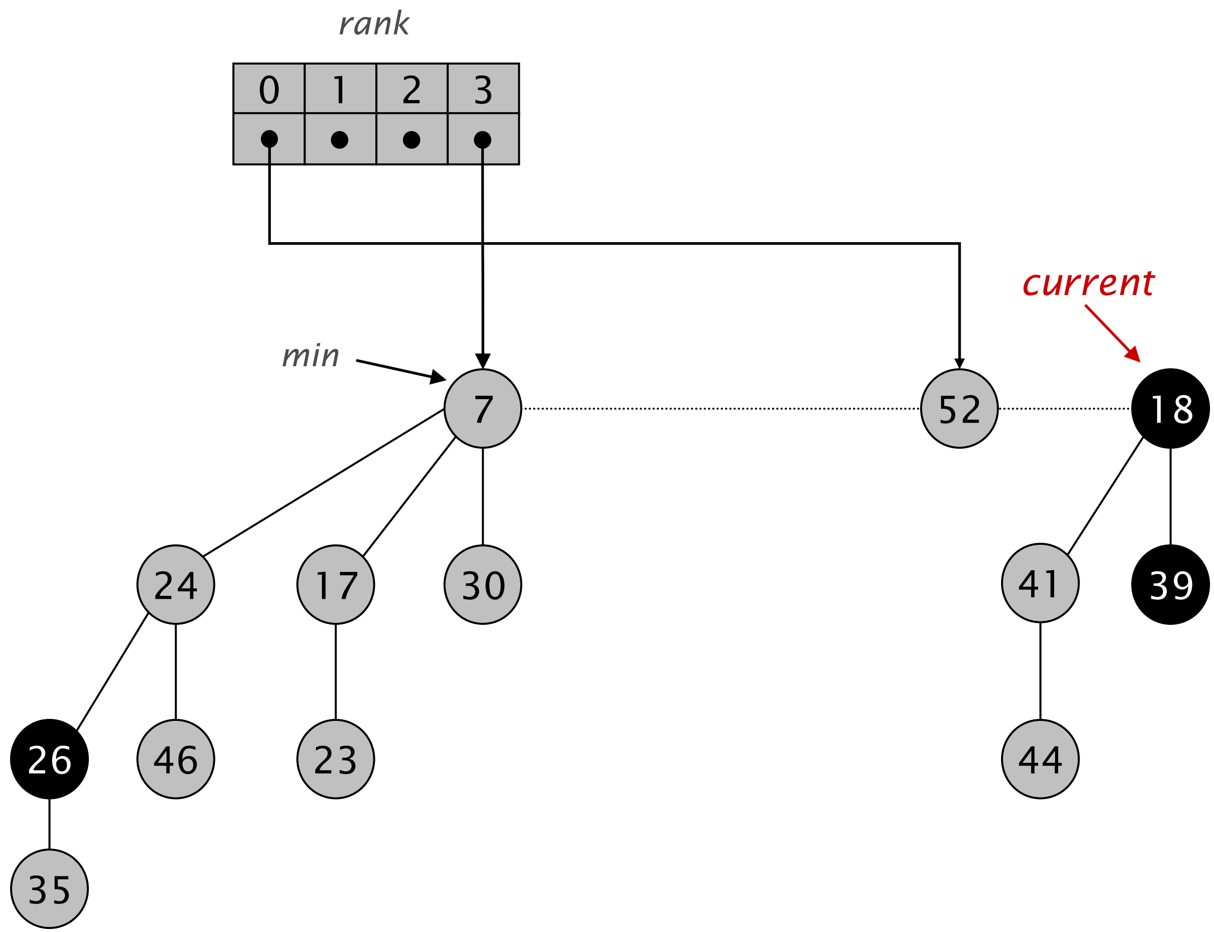
## Delete min.

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



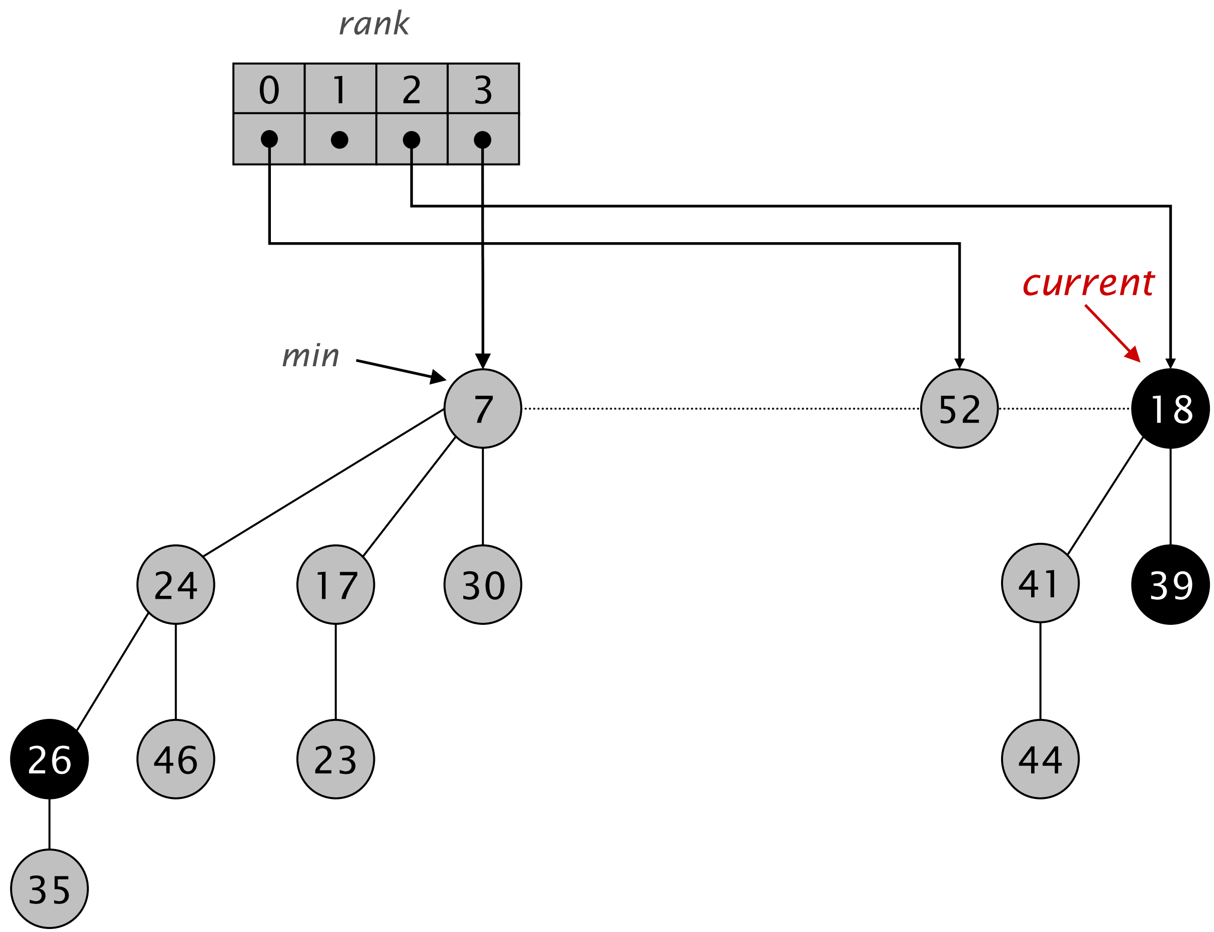
## Delete min.

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



Delete min.

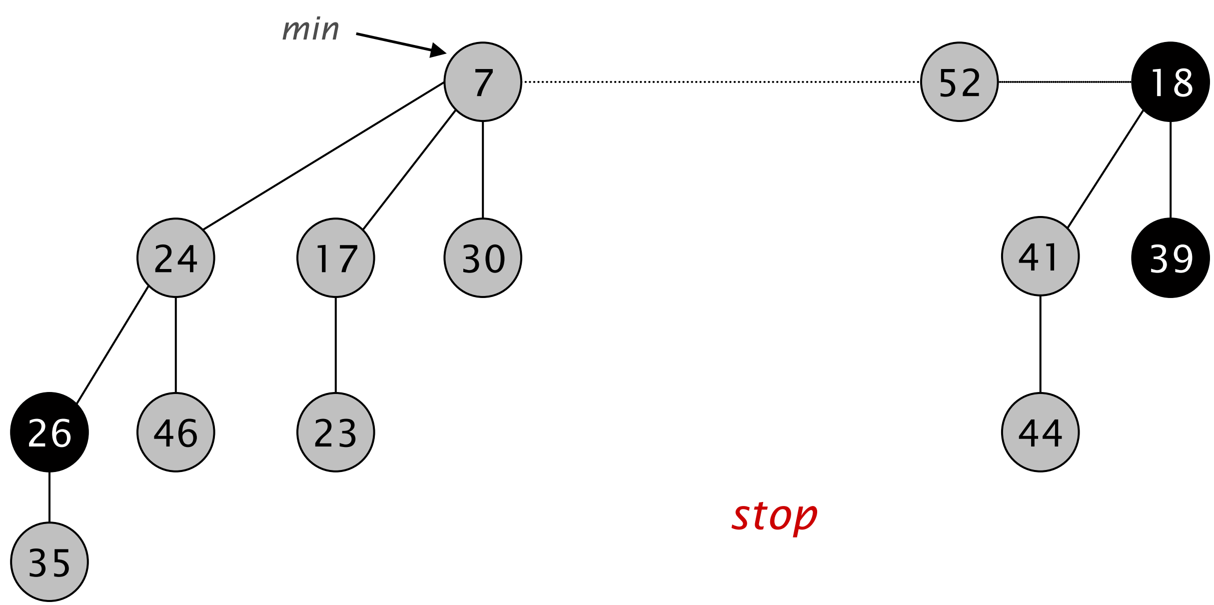
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



# Fibonacci Heaps: Delete Min

## Delete min.

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



# Fibonacci Heaps: Delete Min Analysis

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

*potential function*

**Delete min.**

**Actual cost.** $O(rank(H)) + O(trees(H))$

- $O(rank(H))$ to meld min's children into root list.
- $O(rank(H)) + O(trees(H))$ to update min.
- $O(rank(H)) + O(trees(H))$ to consolidate trees.

**Change in potential.** $O(rank(H)) - trees(H)$

- *No change in marks(H)*
- $trees(H') \leq rank(H) + 1$ since no two trees have same rank.
- $\Delta\Phi(H) \leq rank(H) + 1 - trees(H)$.

**Amortized cost.** $O(rank(H))$

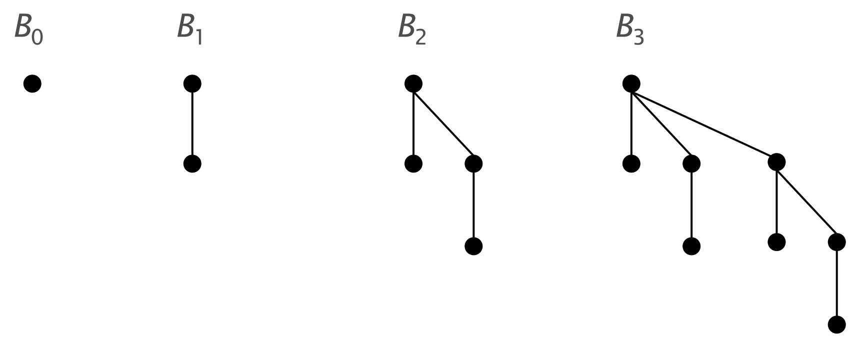
# Fibonacci Heaps: Delete Min Analysis

Q. Is amortized cost of O(*rank*(*H*)) good?

A. Yes, if only *insert* and *delete-min* operations.

- In this case, all trees are binomial trees.
- This implies *rank*(*H*) $\leq$ log *n*.

we only link trees of equal rank



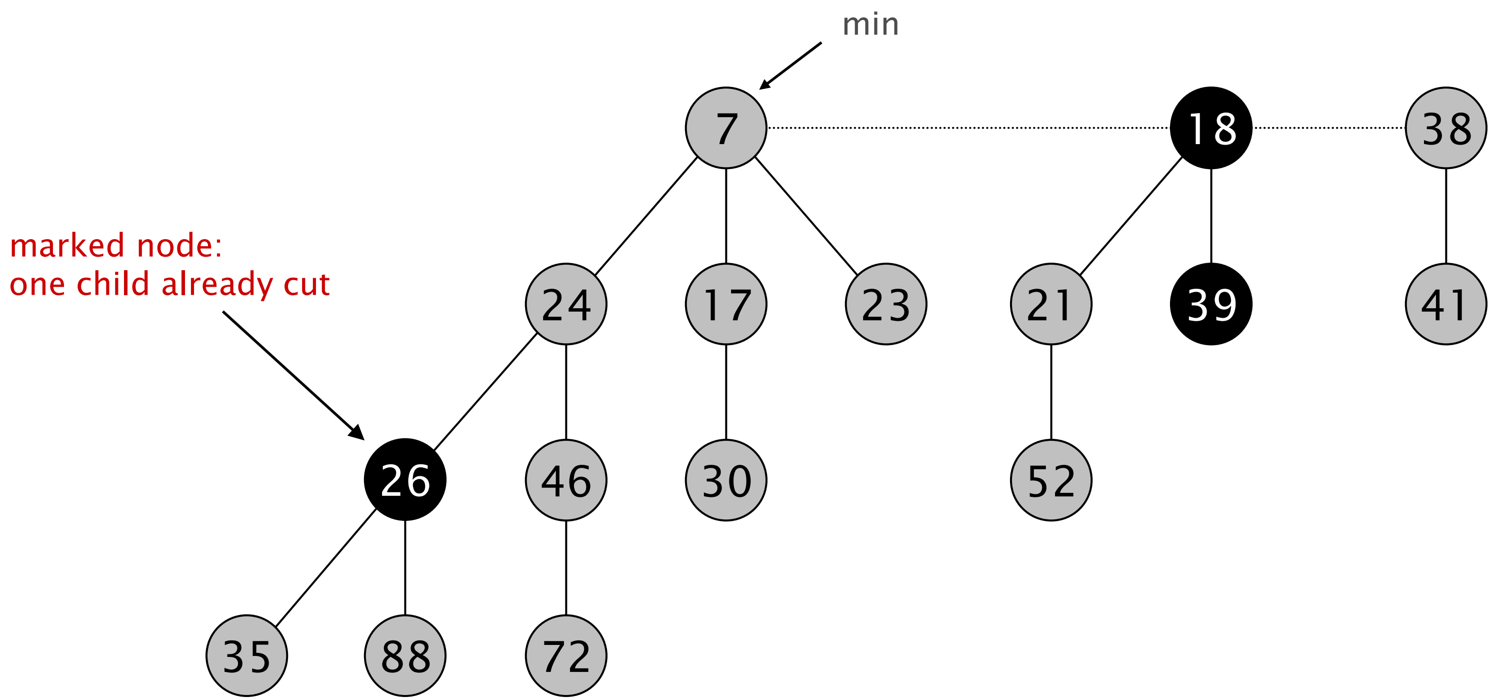

A. Yes, with Fib trees *rank*(*H*) = O(log *n*). Need to ensure that trees remain as Fib trees if not binomial trees

# Decrease Key

# Fibonacci Heaps: Decrease Key

Intuition for deceasing the key of node *x*.

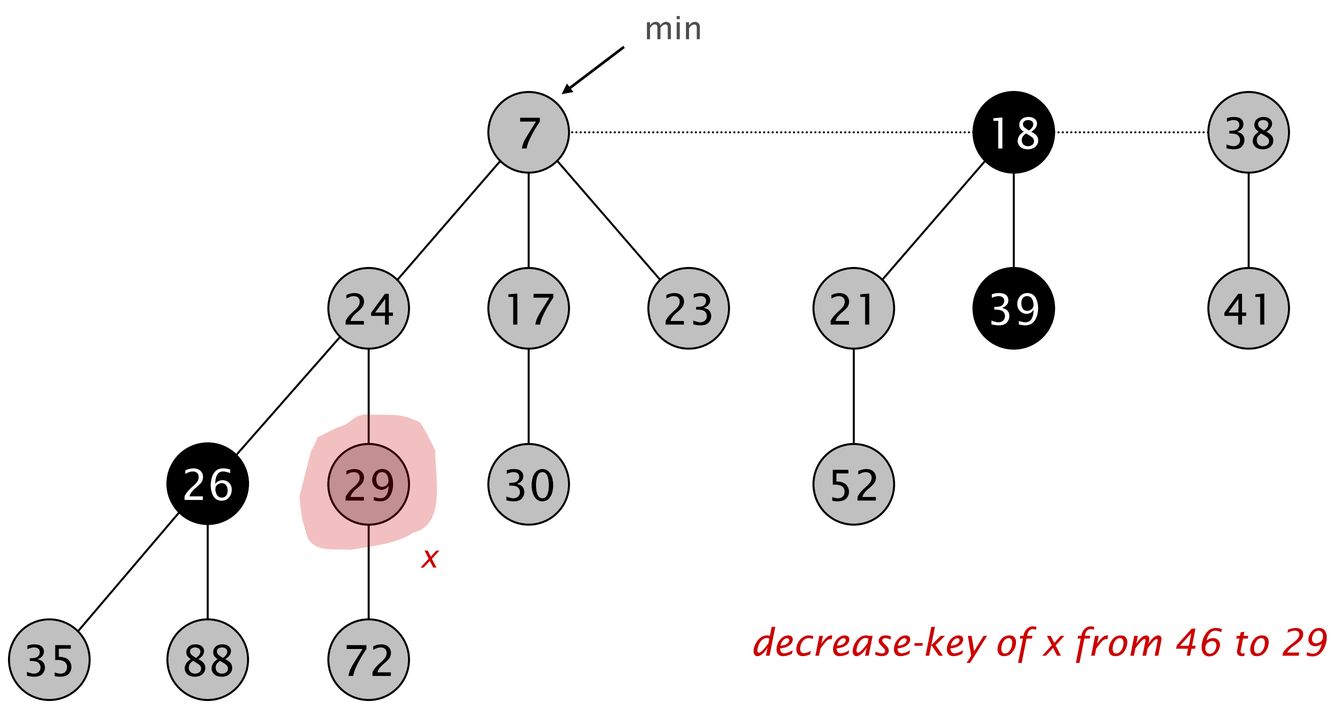
- If heap-order is not violated, just decrease the key of *x*.
- Otherwise, cut tree rooted at *x* and meld into root list.
- To keep trees flat: as soon as a node has its second child cut, cut it off and meld into root list (and unmark it).



# Fibonacci Heaps: Decrease Key

Case 1. [heap order not violated]

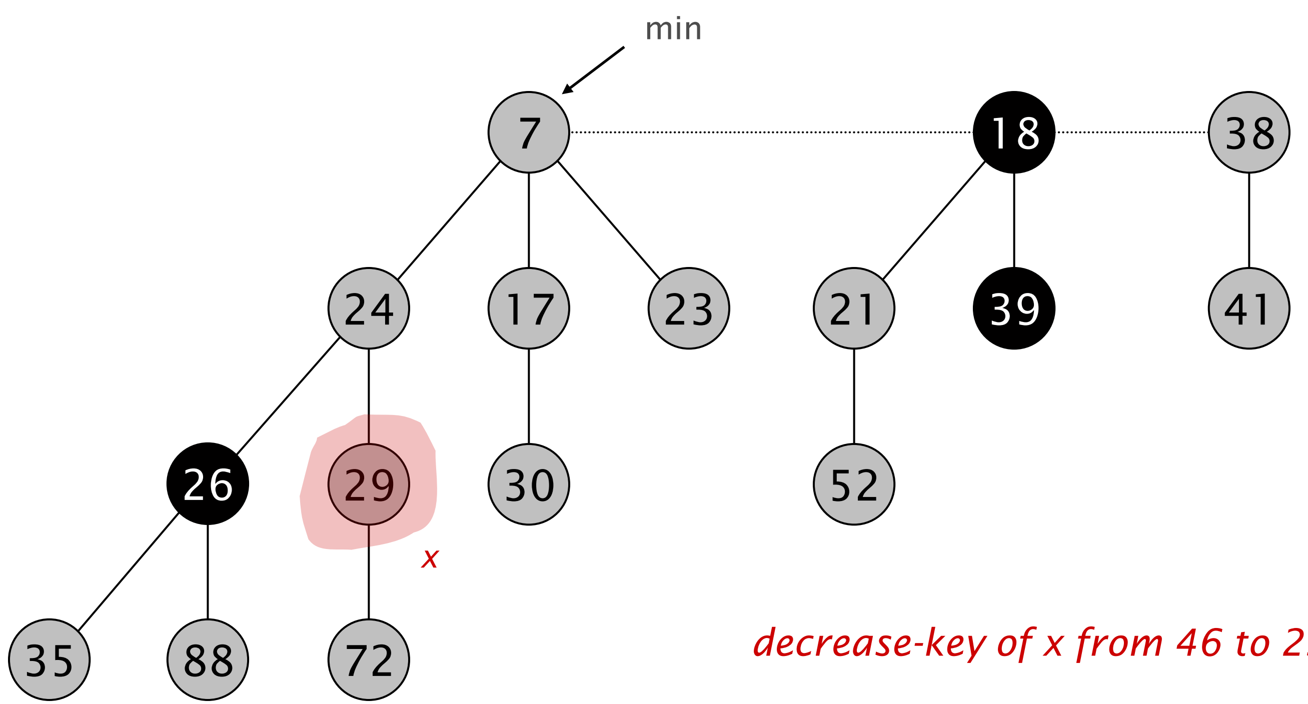
- Decrease key of *x*.
- Change heap min pointer (if necessary).



# Fibonacci Heaps: Decrease Key

Case 1. [heap order not violated]

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

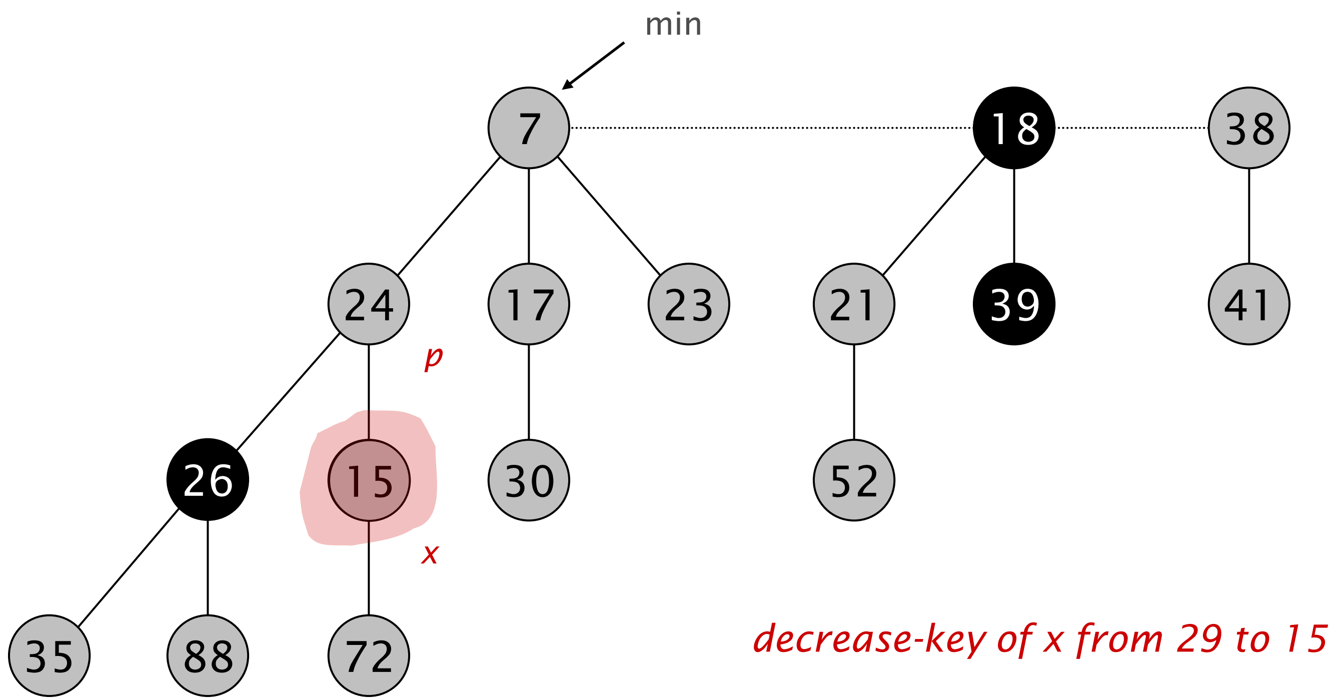



*decrease-key of x from 46 to 29*

# Fibonacci Heaps: Decrease Key

Case 2a. [heap order violated]

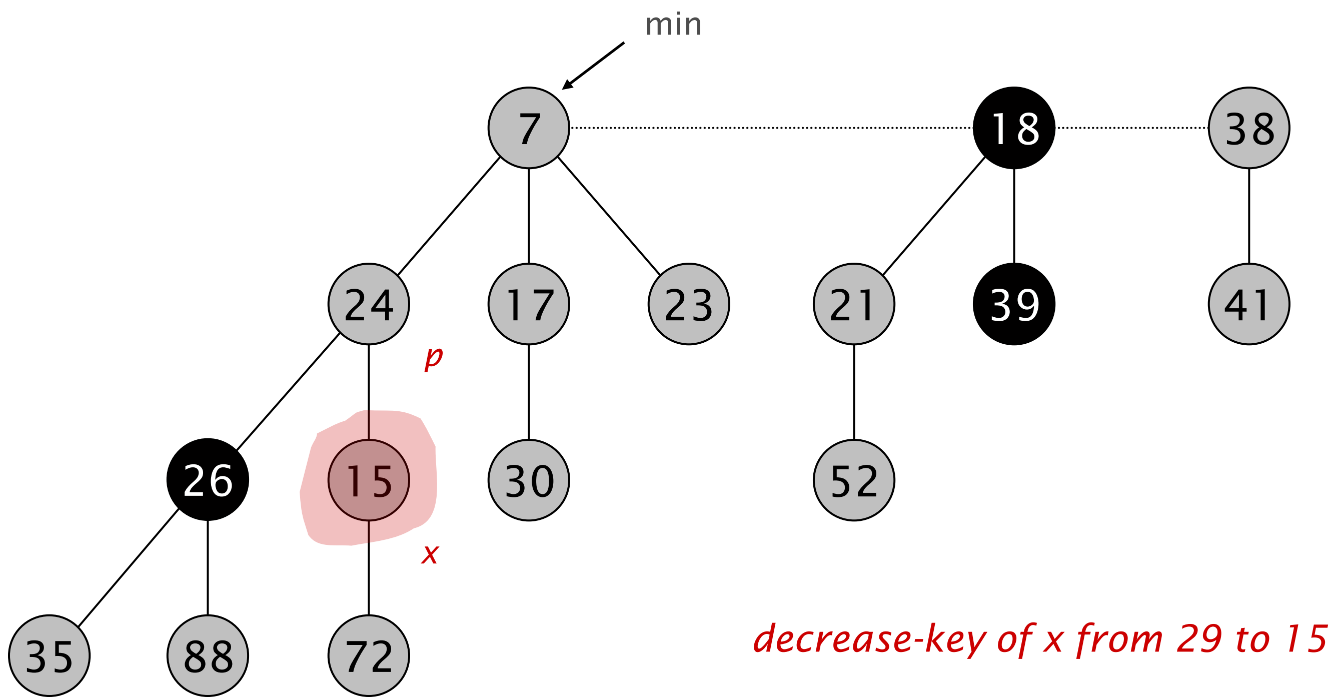
- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).



# Fibonacci Heaps: Decrease Key

Case 2a. [heap order violated]

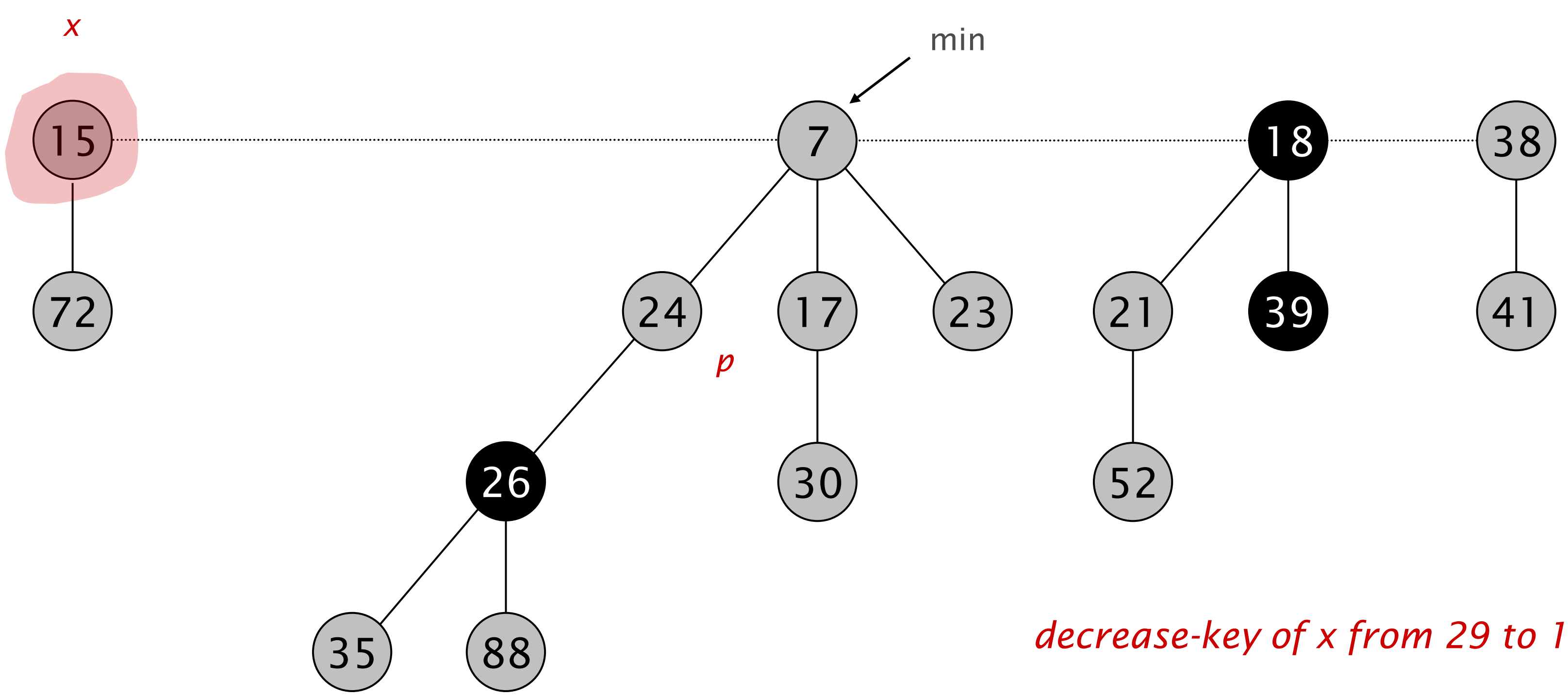
- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).



# 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).

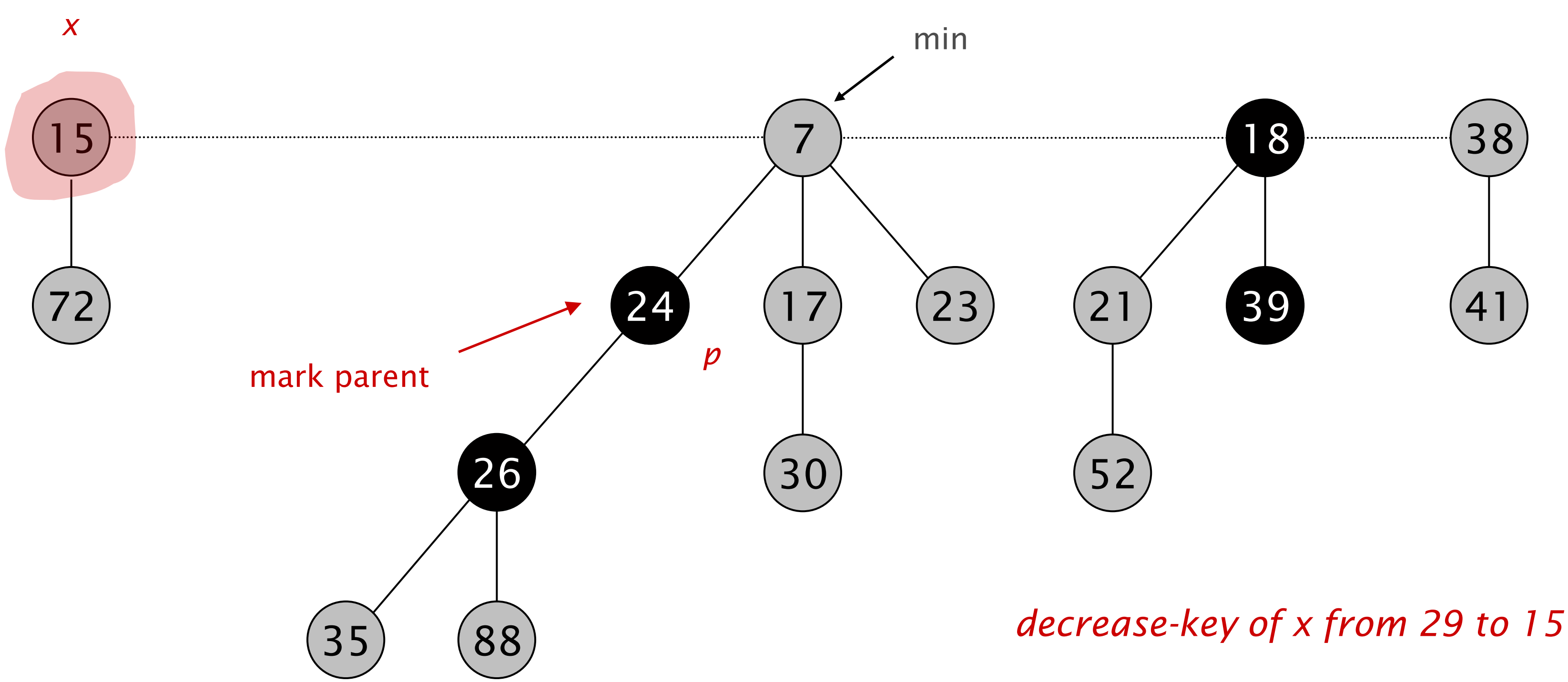



*decrease-key of x from 29 to 15*

# Fibonacci Heaps: Decrease Key

**Case 2a.** [heap order violated]

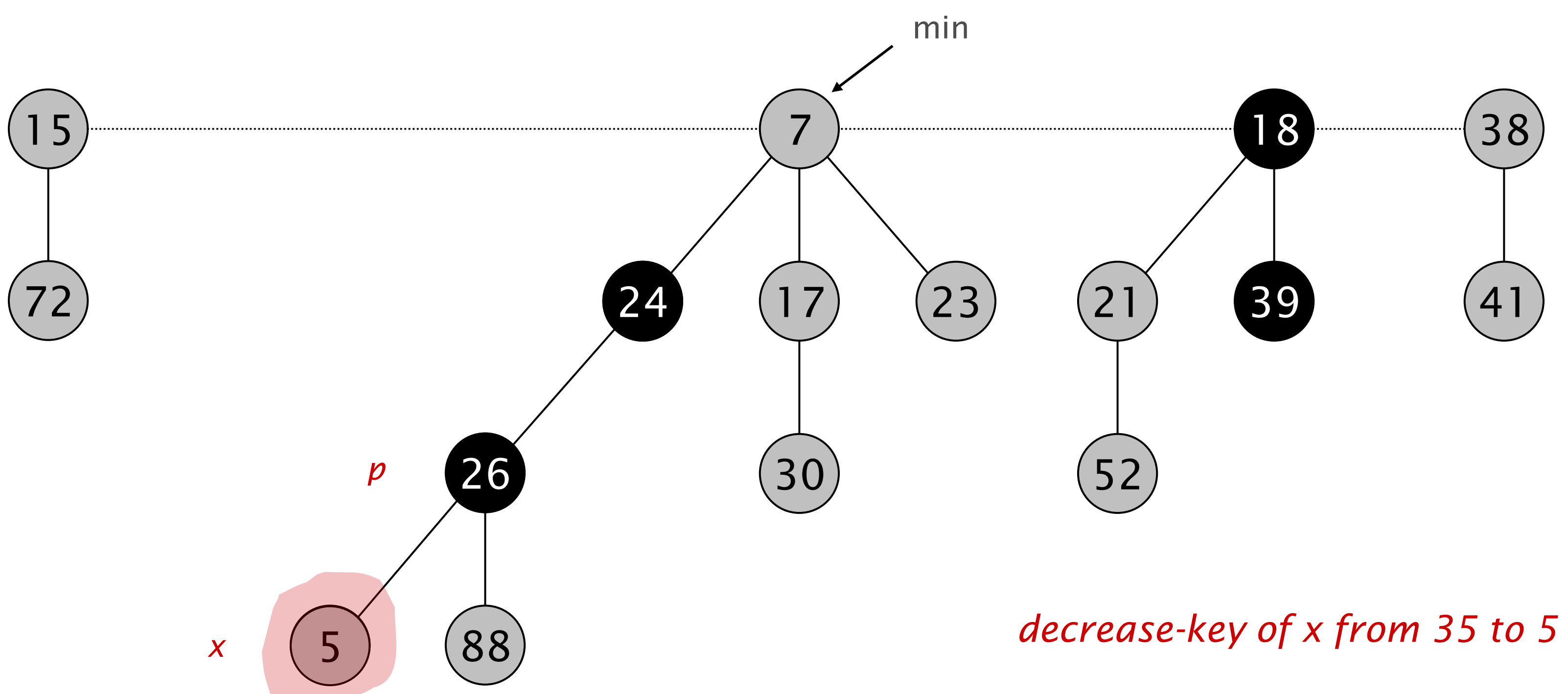
- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).



# Fibonacci Heaps: Decrease Key

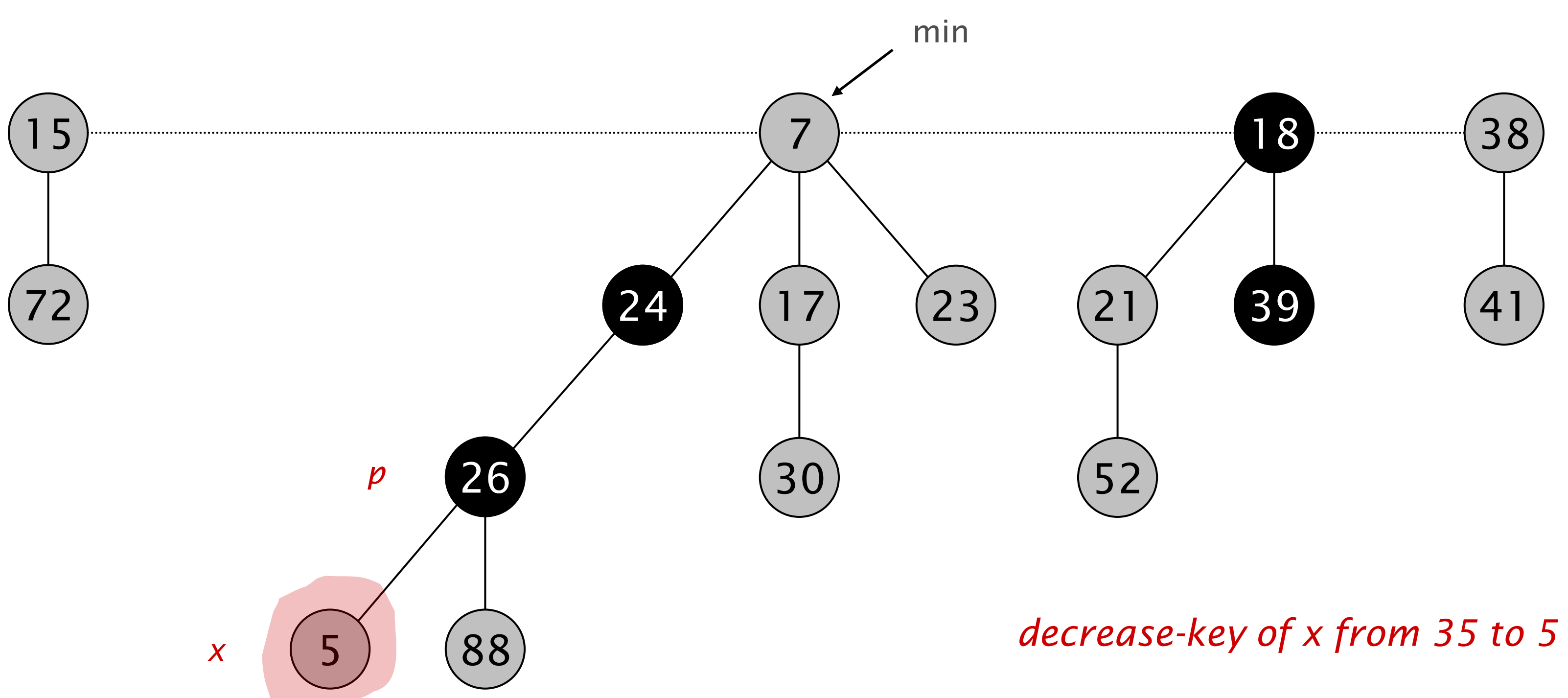
Case 2b. [heap order violated]

- 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).



**Case 2b.** [heap order violated]

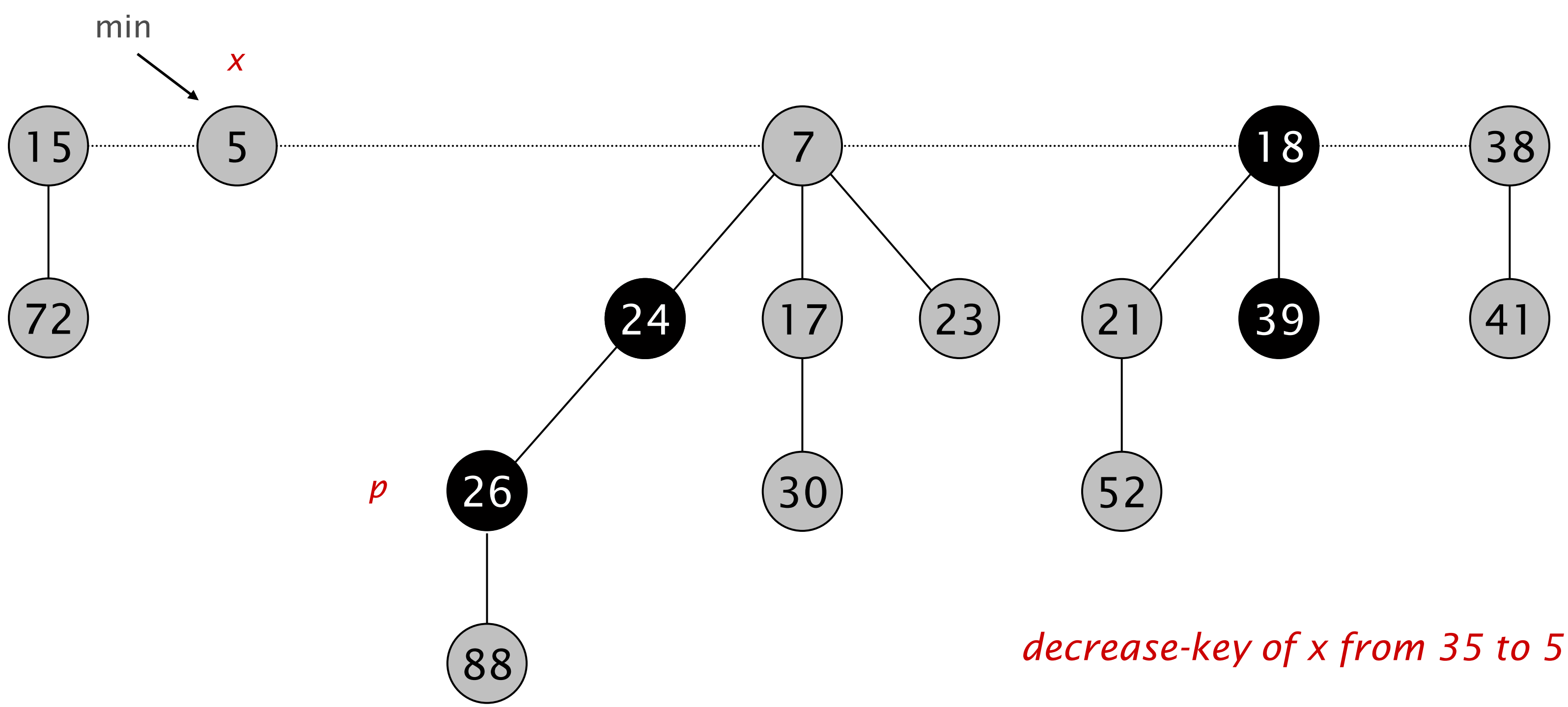
- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).



# Fibonacci Heaps: Decrease Key

Case 2b. [heap order violated]

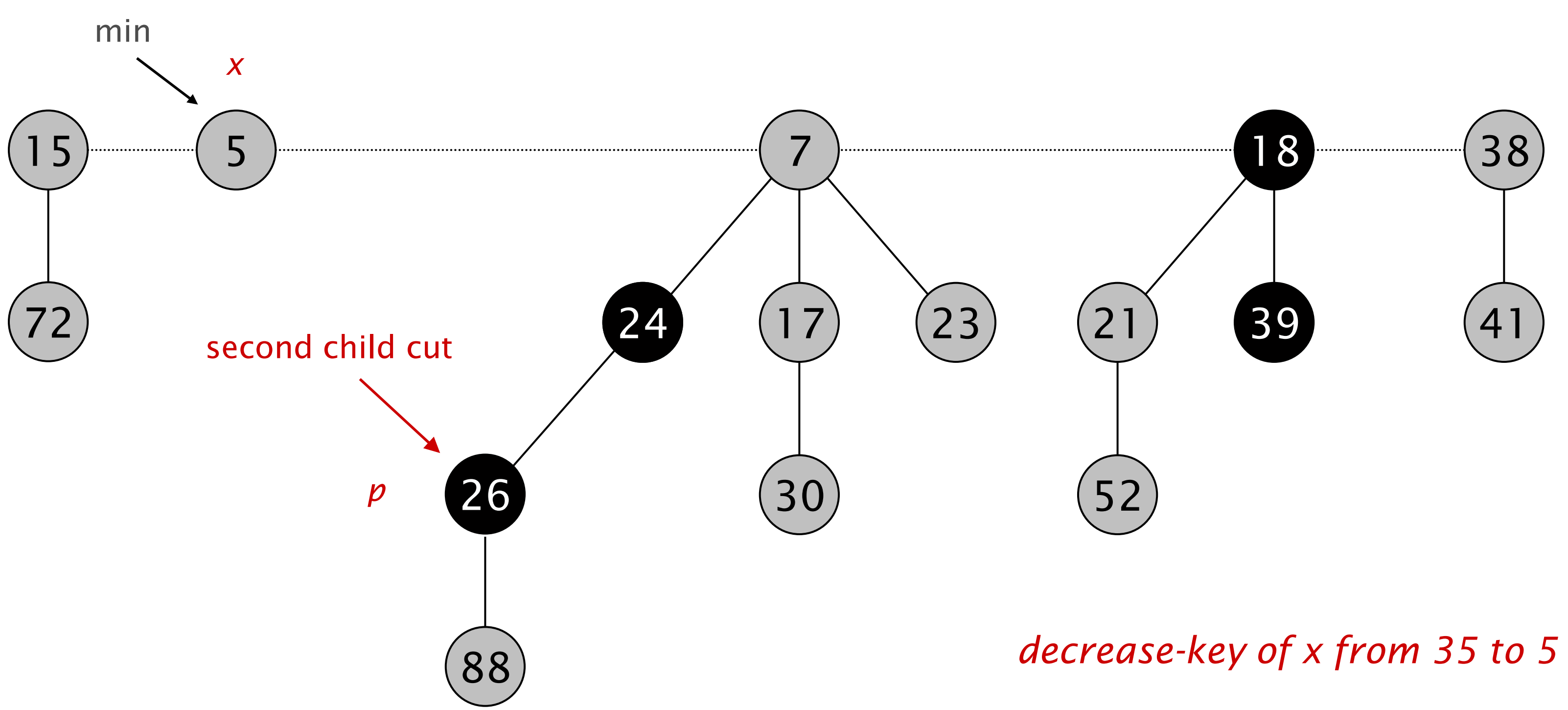
- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).



# Fibonacci Heaps: Decrease Key

Case 2b. [heap order violated]

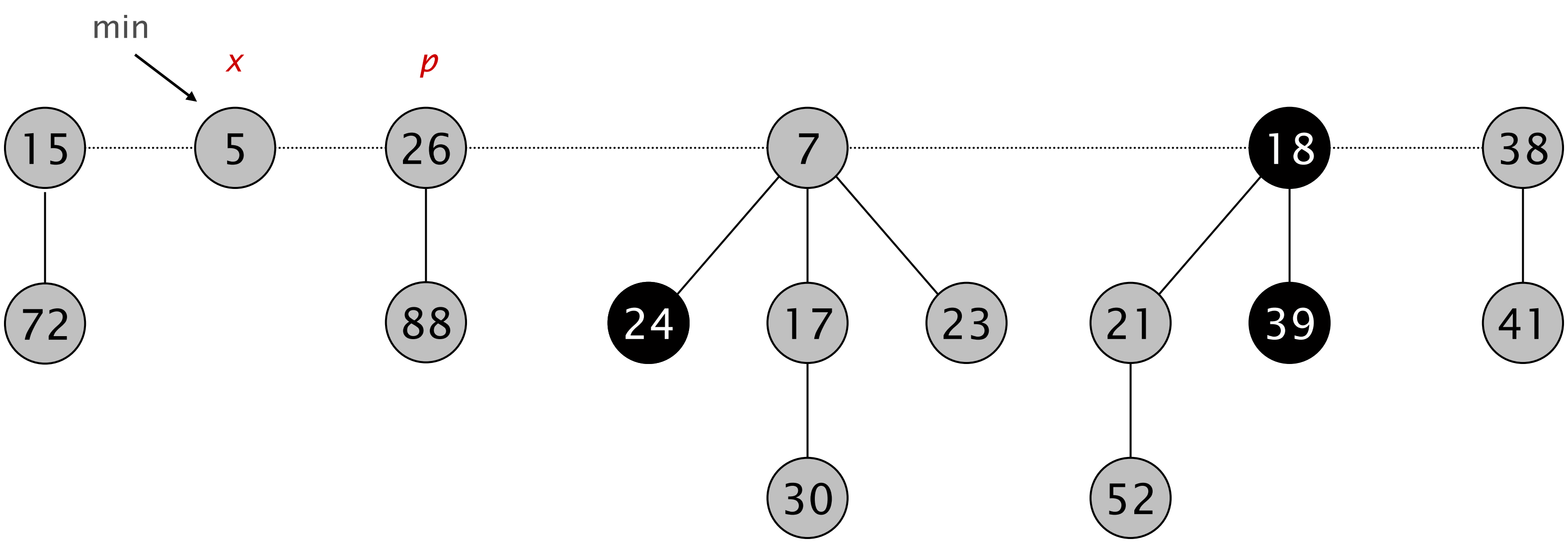
- Decrease key of *x*.
- Cut tree rooted at *x*, meld into root list, and unmark.
- If parent *p* of *x* is unmarked (hasn't yet lost a child), mark it;
  Otherwise, cut *p*, meld into root list, and unmark
  (and do so recursively for all ancestors that lose a second child).



# 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).

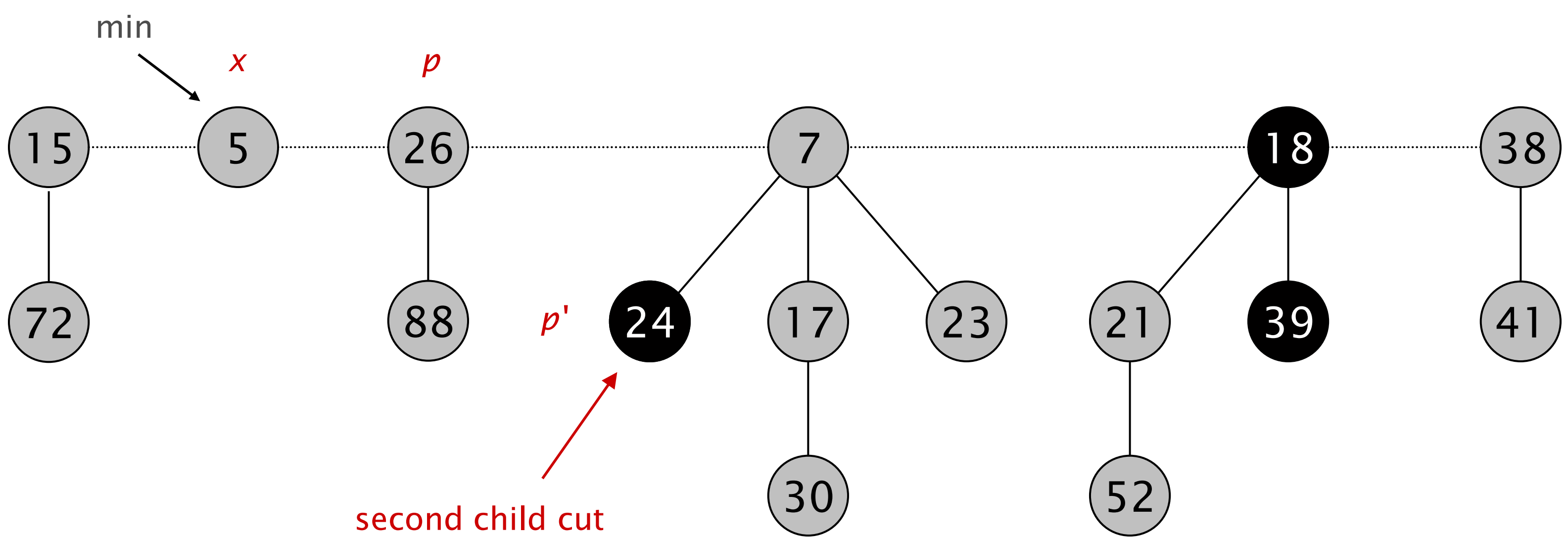



*decrease-key of x from 35 to 5*

# 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).

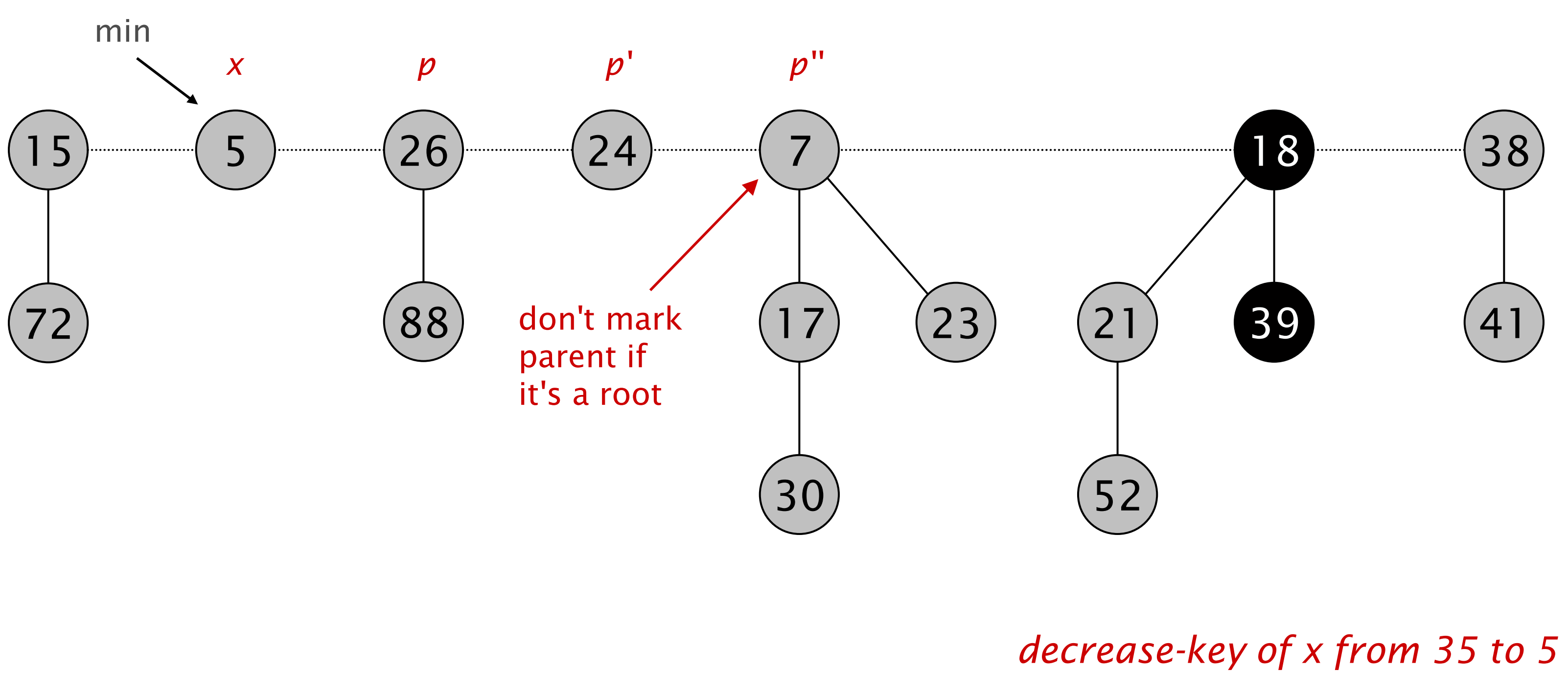



*decrease-key of x from 35 to 5*

# 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).



# Fibonacci Heaps: Decrease Key Analysis

**Decrease-key.**

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

*potential function*

**Actual cost.** $O(c)$, where c is the number of cuts needed

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

**Change in potential.** $O(1) - c$

- $trees(H') = trees(H) + c.$
- $marks(H') \le marks(H) - c + 2.$
- $\Delta\Phi \le c + 2 \cdot (-c + 2) = 4 - c.$

**Amortized cost.** $O(1)$

# Analysis

- Need to show that all the trees are Fib trees
- Need to prove the two properties of Fib trees

# Proving Fib Trees

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

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

Proof.

- When $y_i$ is linked to x, $y_1, \ldots, y_{i-1}$ already linked to x,
  $\Rightarrow$ In that step, rank(x) $\geq$ i - 1
  $\Rightarrow$ rank($y_i$) $\geq$ i - 1 since we only link nodes of equal degree
- Since then, $y_i$ has lost at most one child
  - otherwise it would have been cut from x
- Thus, rank($y_i$) $\geq$ i - 2

# Properties of Fib Trees

*Define: size(T) as the number of nodes in T*

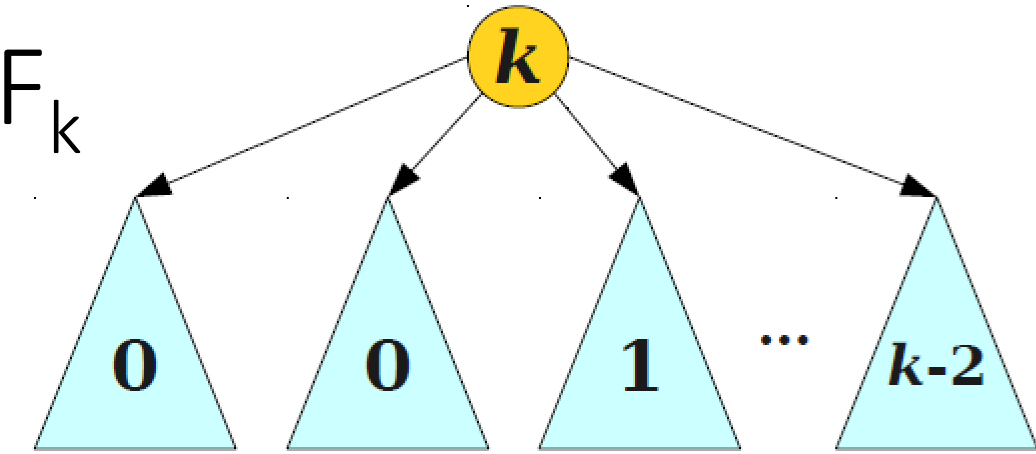
*D(T) = max degree of the nodes in T*

*Lemma:* For the Fib tree $F_k$

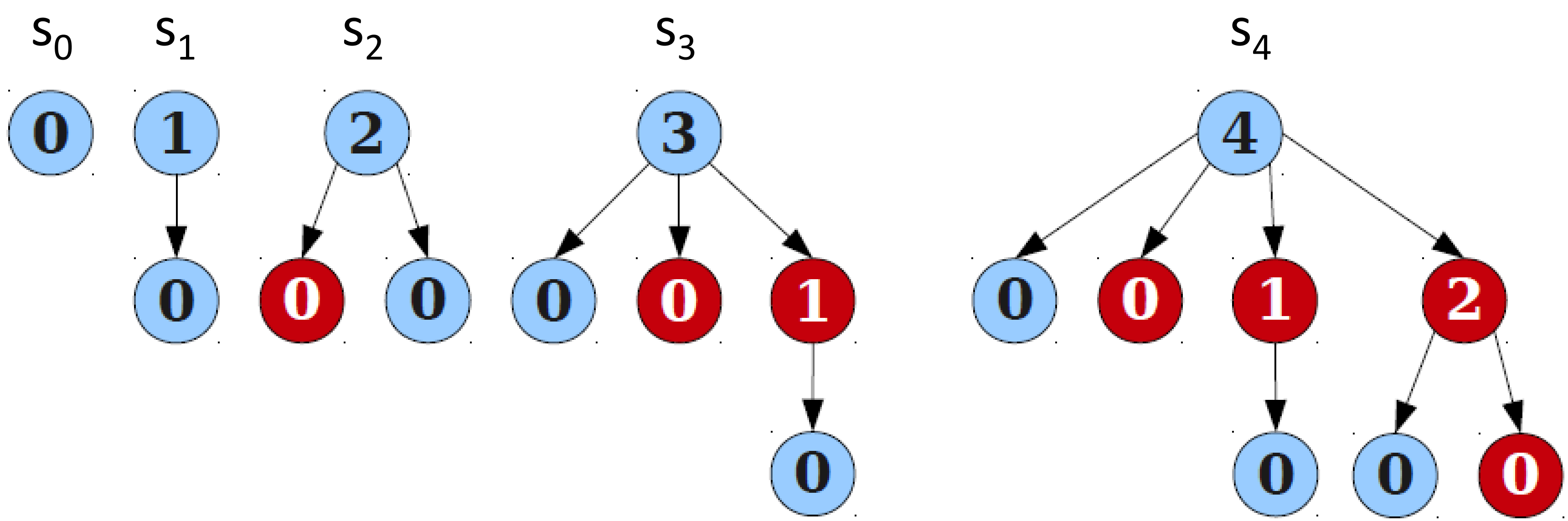
1. $\text{size}(F_k) \geq \phi^k$ where $\phi = (1 + \sqrt{5}) / 2$
2. $D(F_k) \leq \log_\phi(\text{size}(F_k))$

Homework exercise

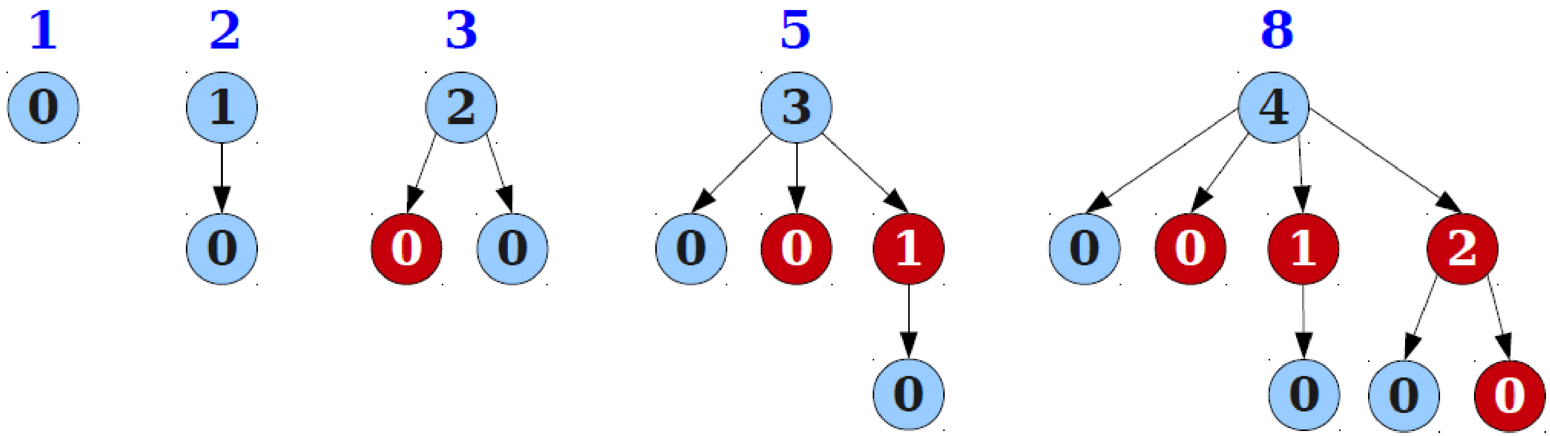
# Bounding Size of $F_k$




Let $s_k$ be the smallest possible size of $F_k$



# Bounding Size of $F_k$

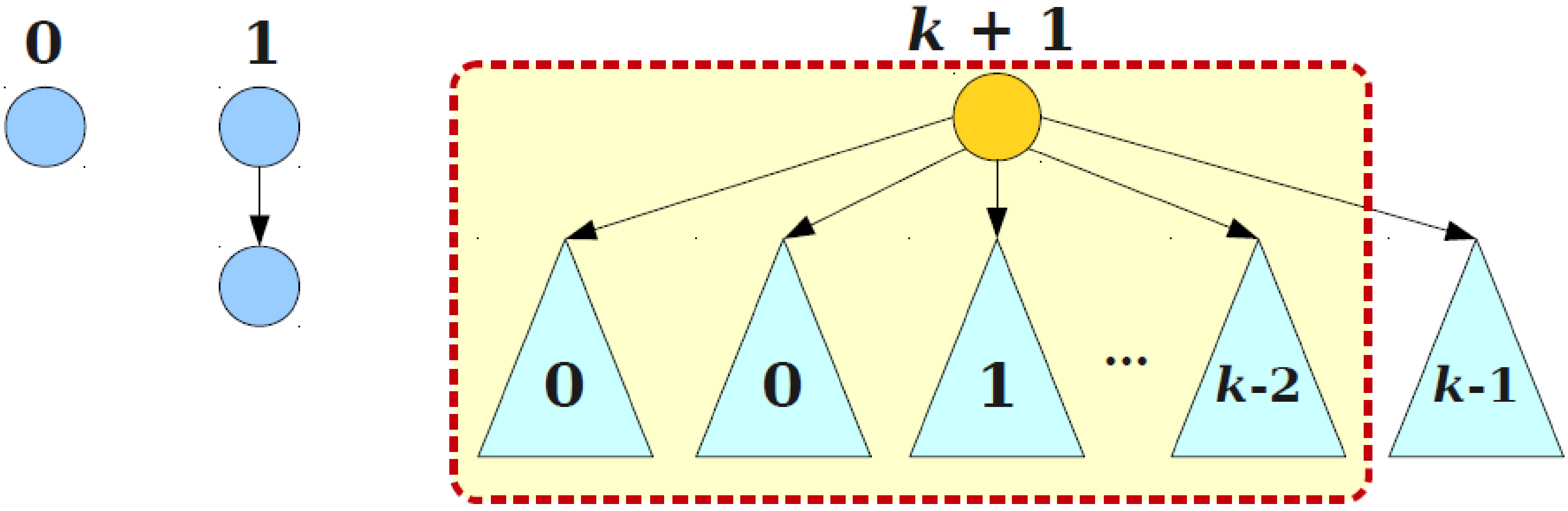



**Claim:** The minimum number of nodes in a tree of rank *k* is $\boldsymbol{F_{k+2}}$

# Bounding Size of $F_k$

***Theorem:*** The number of nodes in a Fib tree of rank *k* is $F_{k+2}$.

***Proof:*** Induction.




$F_2$ $F_3$ $F_{k+2}$ $+$ $F_{k+1}$

# Bounding the Rank of Nodes in $F_k$

- **Fact:** For $n \geq 2$, we have $F_n \geq \phi^{n-2}$, where $\phi$ is the golden ratio: $\phi \approx 1.61803398875...$
- **Claim:** In our modified data structure, we have $rank(T) = O(\log n)$.
- **Proof:** In a tree of rank $k$, there are at least $F_{k+2} \geq \phi^k$ nodes. Therefore, the maximum rank of a tree in our data structure is $\log_\phi n = O(\log n)$.

# Fibonacci Heaps: Summary

| Operation | Linked List | Binary Heap | Binomial Heap | Fibonacci Heap † | Relaxed Heap |
|---|---|---|---|---|---|
| *make-heap* | 1 | 1 | 1 | 1 | 1 |
| *is-empty* | 1 | 1 | 1 | 1 | 1 |
| *insert* | 1 | log *n* | log *n* | 1 | 1 |
| *delete-min* | *n* | log *n* | log *n* | log *n* | log *n* |
| *decrease-key* | *n* | log *n* | log *n* | 1 | 1 |
| *delete* | *n* | log *n* | log *n* | log *n* | log *n* |
| *union* | 1 | *n* | log *n* | 1 | 1 |
| *find-min* | *n* | 1 | log *n* | 1 | 1 |

*n* = number of elements in priority queue

*Runtime of Dijkstra's/Prim's Algorithm*

$O(|V|^2)$

$O(|E|\log(|V|))$

$O(|E| + |V|\log(|V|))$

# Thank You