Data Structures

Binomial Heaps
Fibonacci Heaps

Haim Kaplan & Uri Zwick

December 2013

Heaps / Priority queues

	Binary Heaps	**Binomial Heaps**	**Lazy Binomial Heaps**	**Fibonacci Heaps**
Insert	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
Find-min	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Delete-min	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Decrease-key	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Meld	—	$O(\log n)$	$O(1)$	$O(1)$

Worst case　　　　Amortized

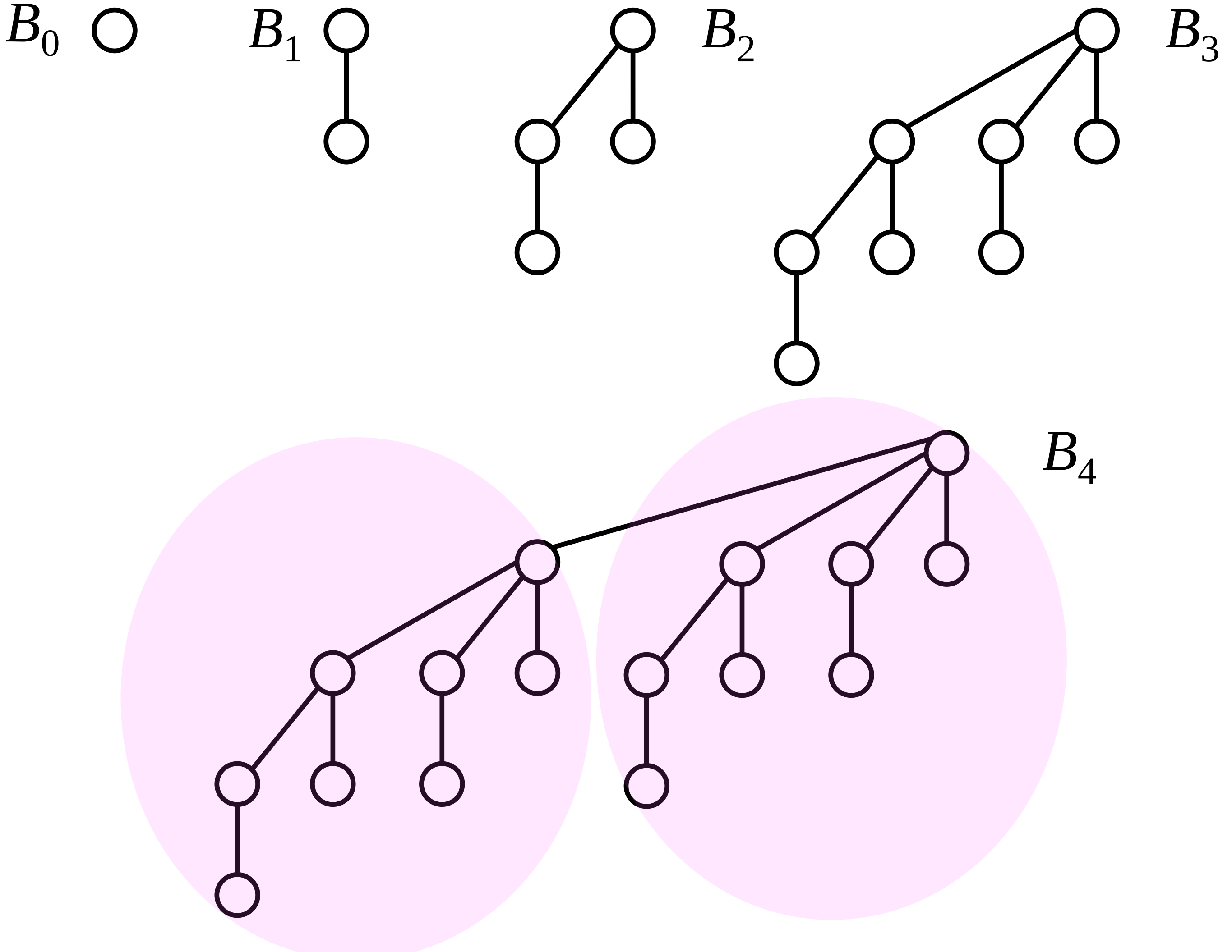
Delete can be implemented using Decrease-key + Delete-min

Decrease-key in O(1) time important for Dijkstra and Prim

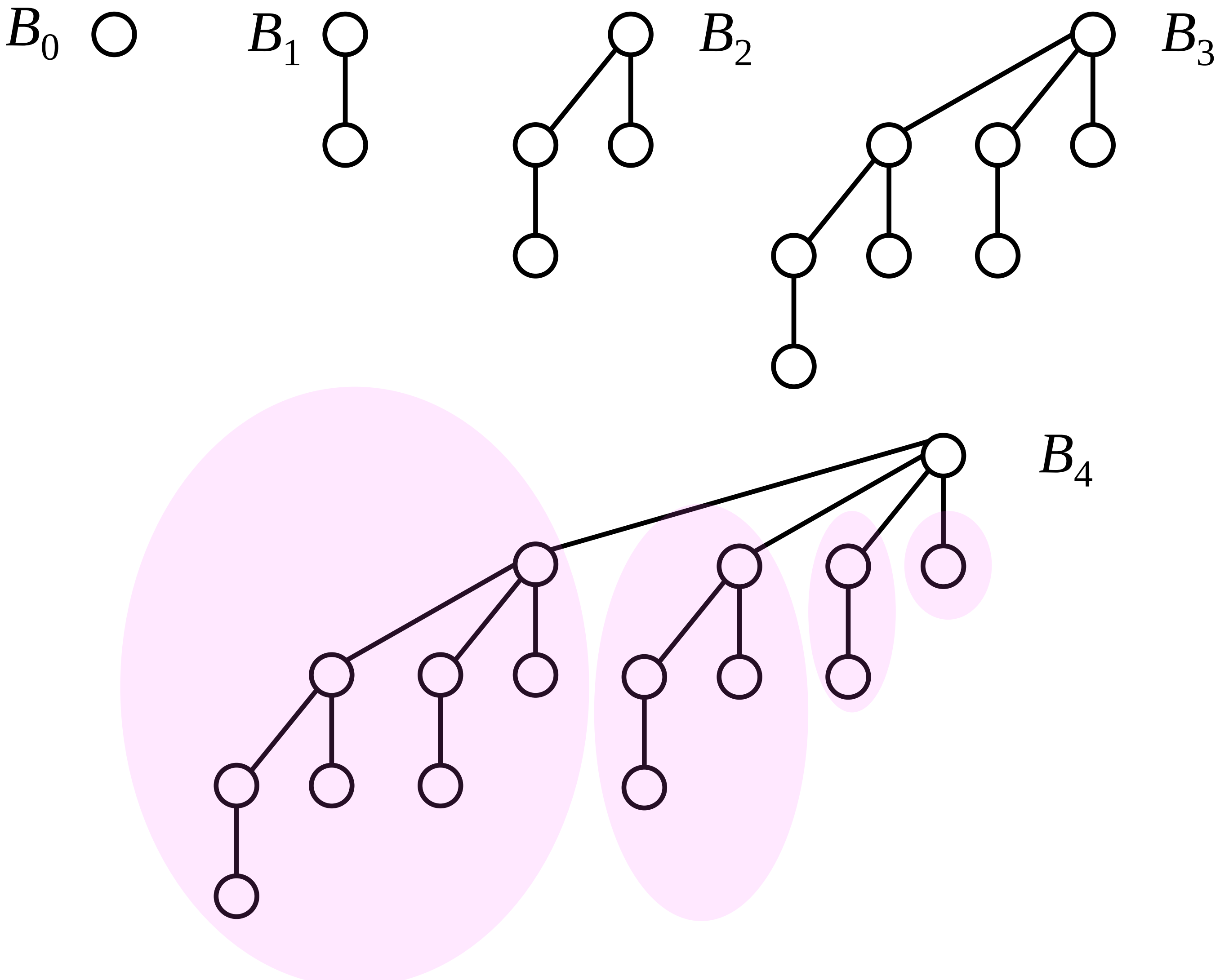
Binomial Heaps

[Vuillemin (1978)]

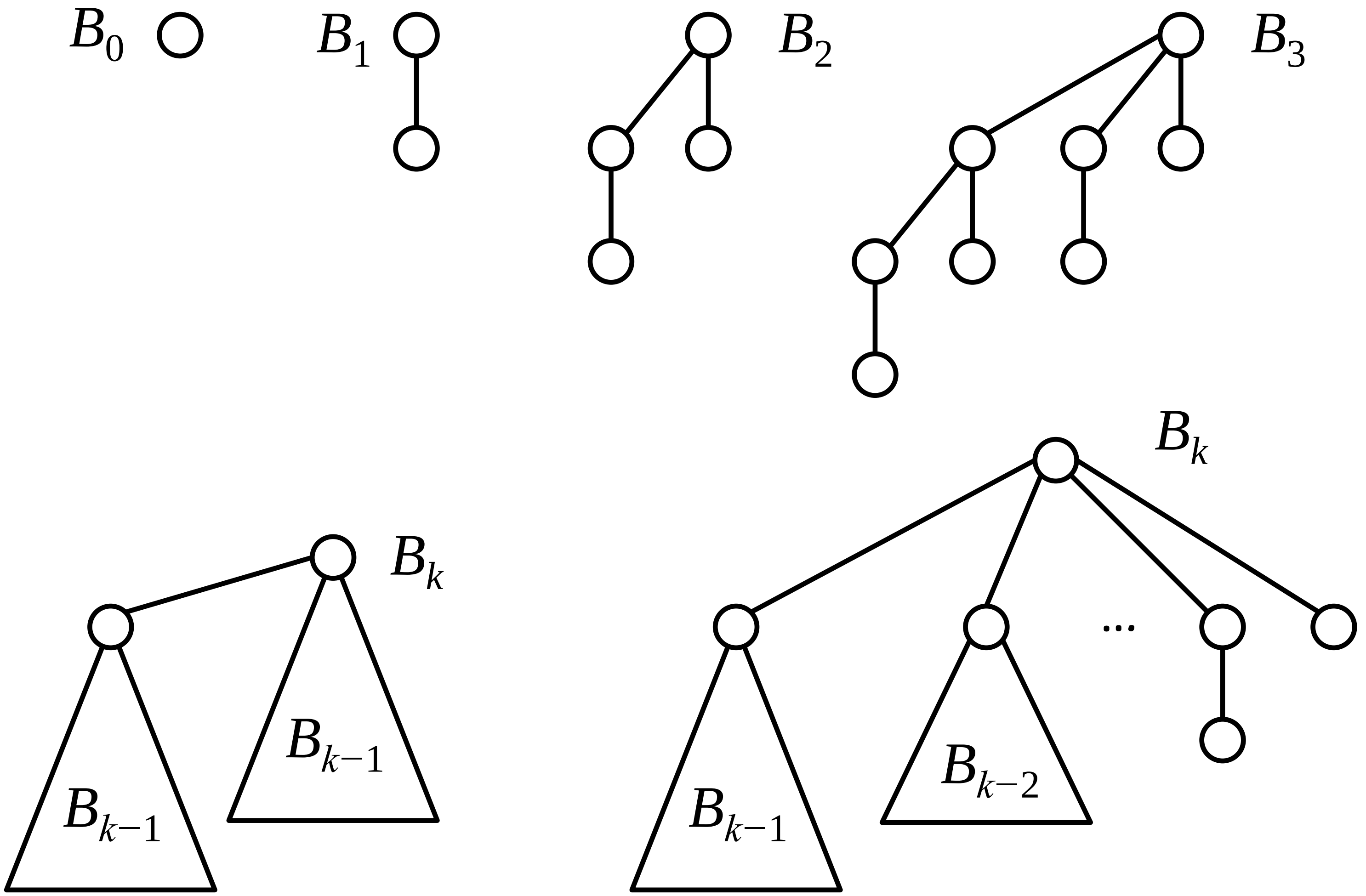
Binomial Trees



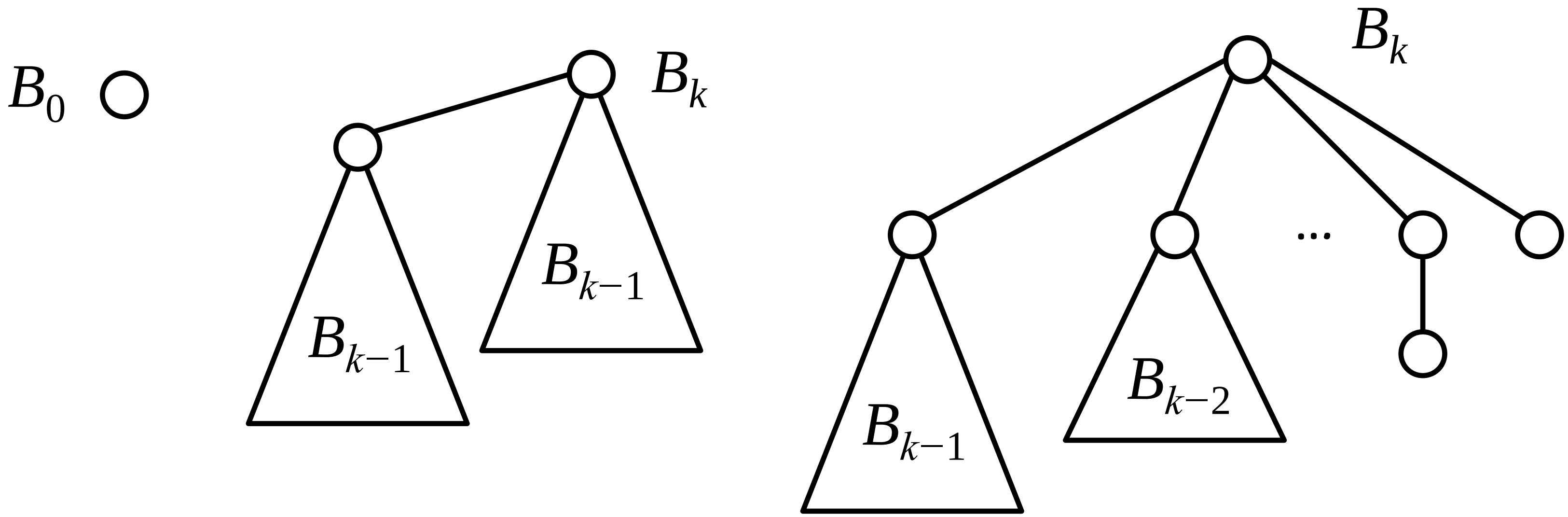
Binomial Trees



Binomial Trees



Binomial Trees



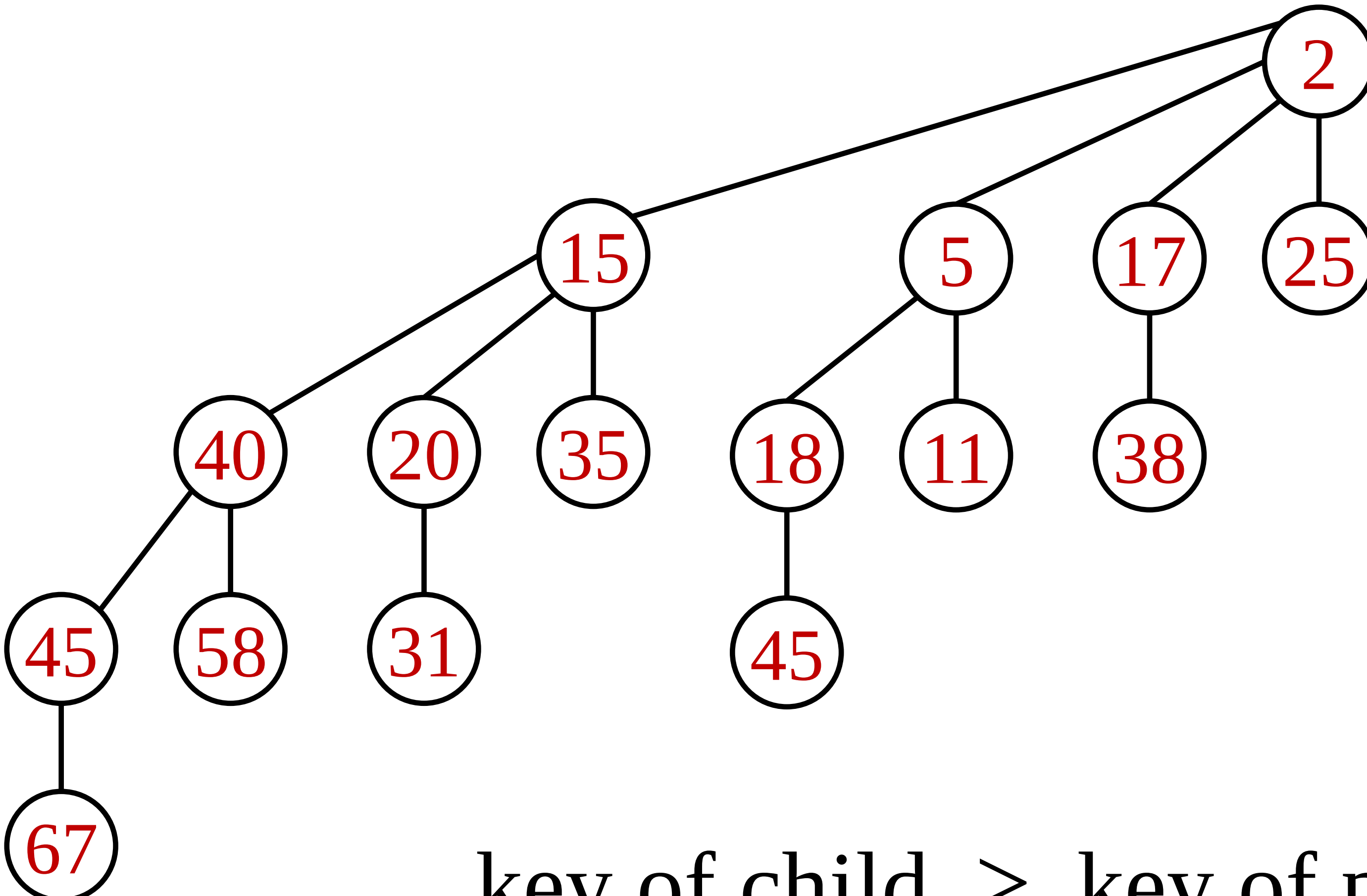
B_k contains 2^k nodes and its depth is k

$\binom{k}{i}$ of the nodes of B_k are at level i

The root of B_k has k children

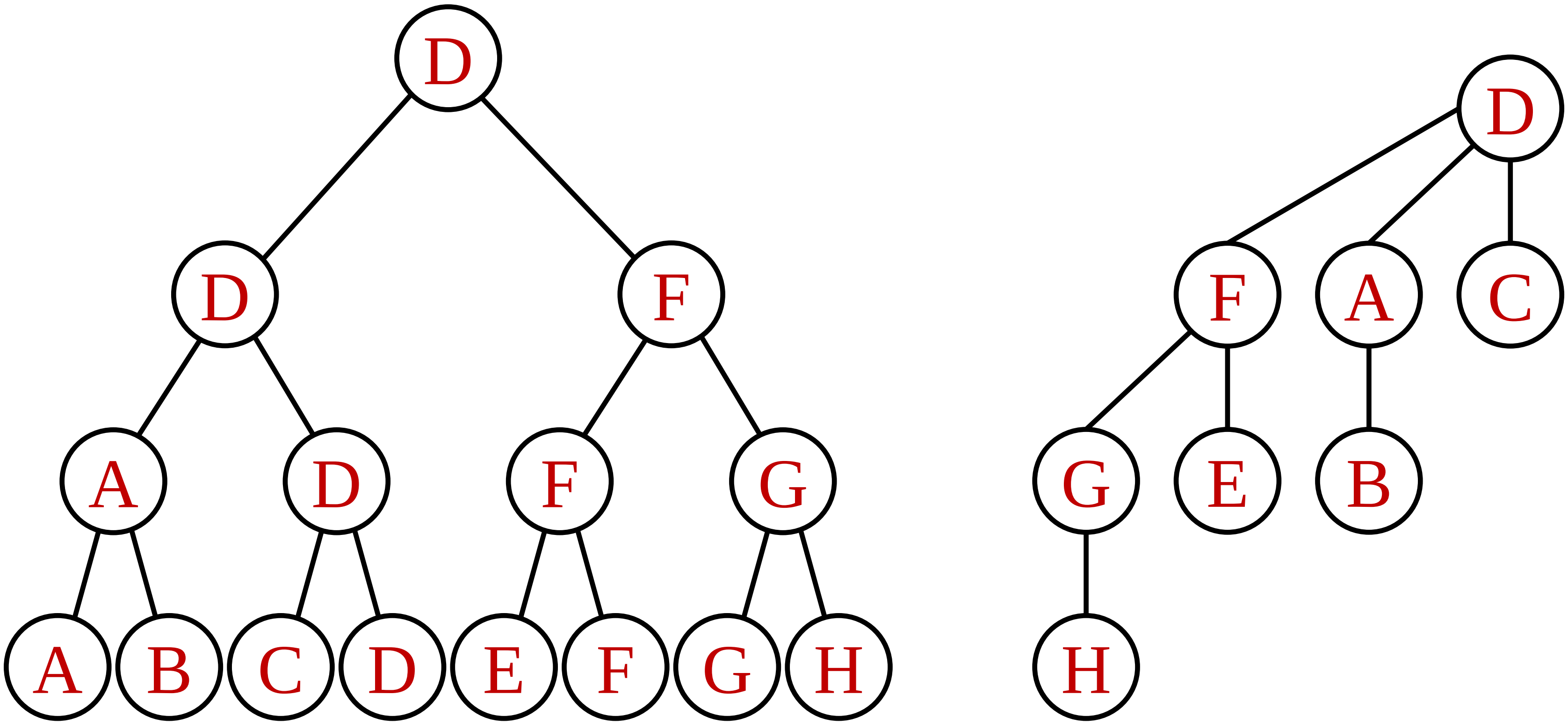
$\sum_{i=0}^{k} \binom{k}{i} = 2^k \qquad \binom{k}{i} = \binom{k-1}{i} + \binom{k-1}{i-1}$

Min-heap Ordered Binomial Trees



key of child $\geq$ key of parent

Tournaments ⇔ Binomial Trees

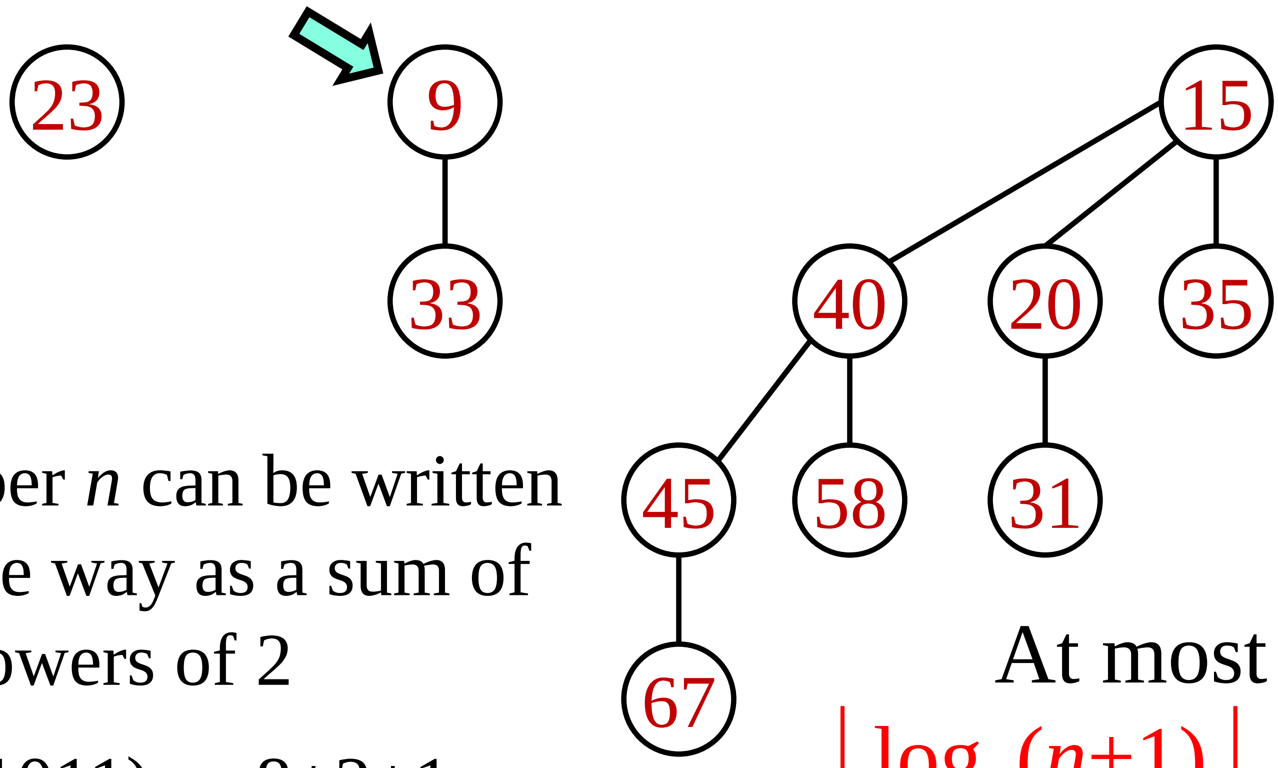


The children of *x* are the items that lost matches with *x*, in the order in which the matches took place.

Binomial Heap

A list of binomial trees, at most one of each rank

Pointer to root with minimal key

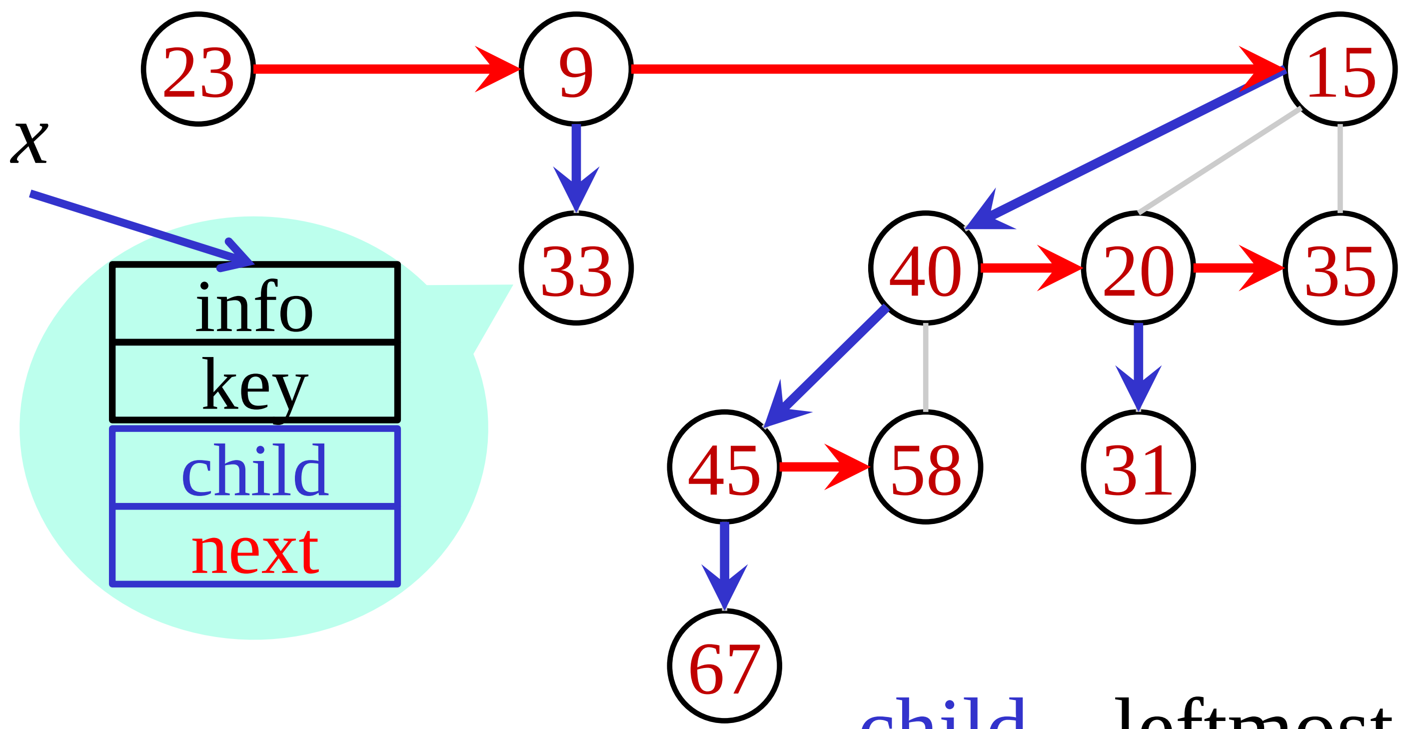


Each number n can be written in a unique way as a sum of powers of 2

$11 = (1011)_2 = 8+2+1$

At most $\lfloor \log_2(n+1) \rfloor$ trees

Ordered forest → Binary tree

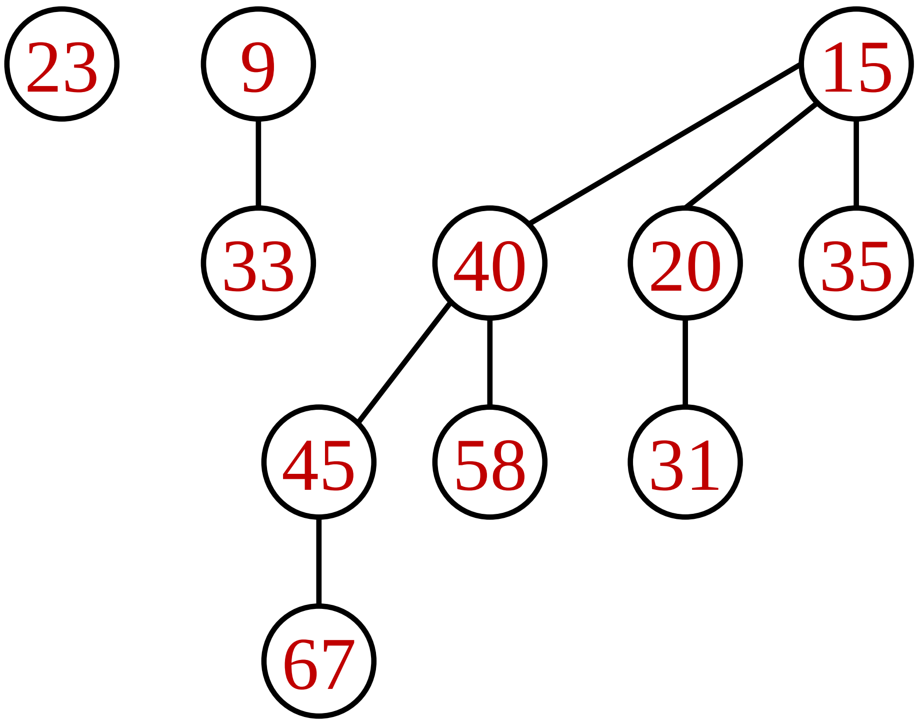


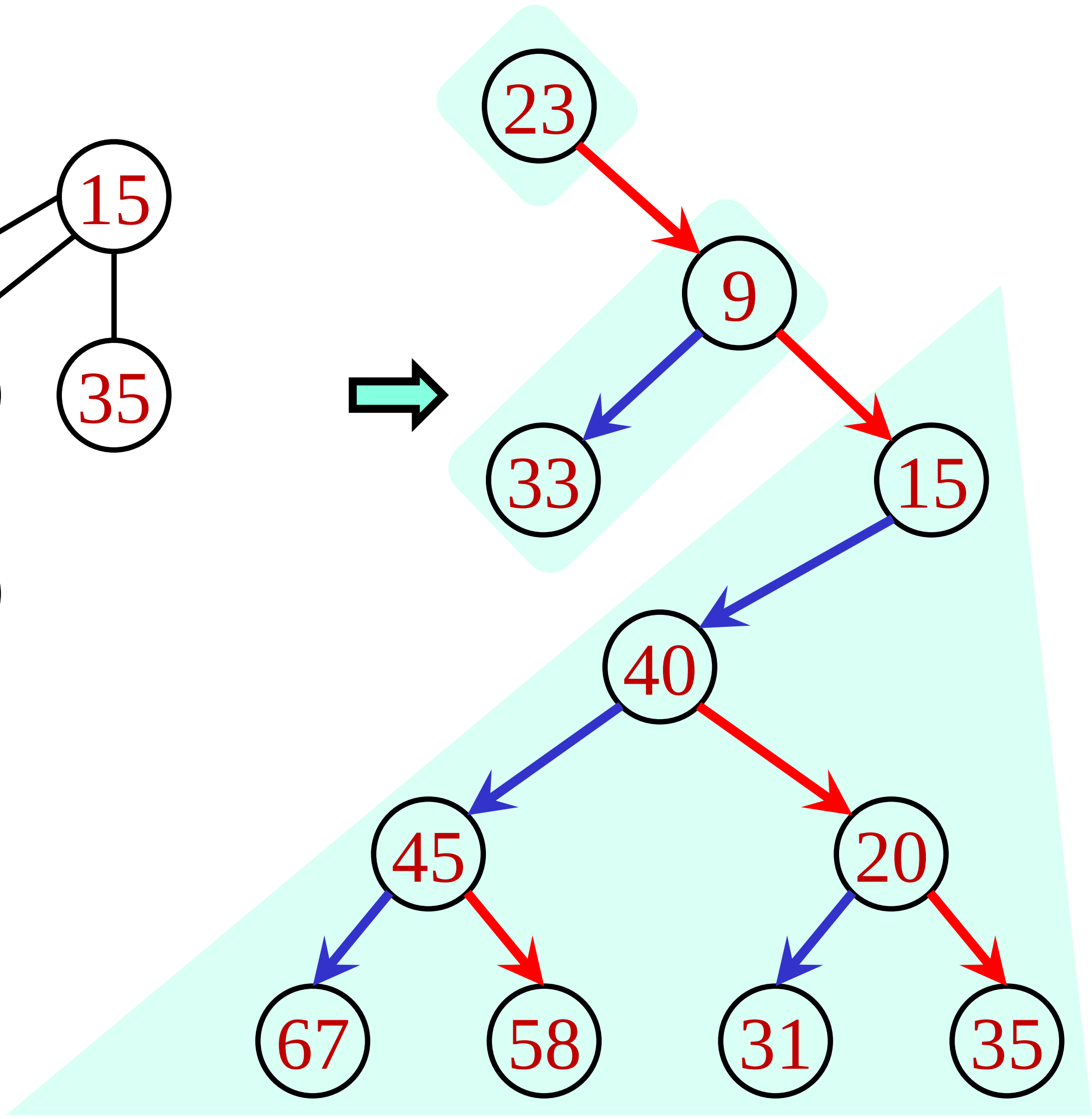
child – leftmost child

next – next "sibling"

2 pointers per node

Forest → Binary tree





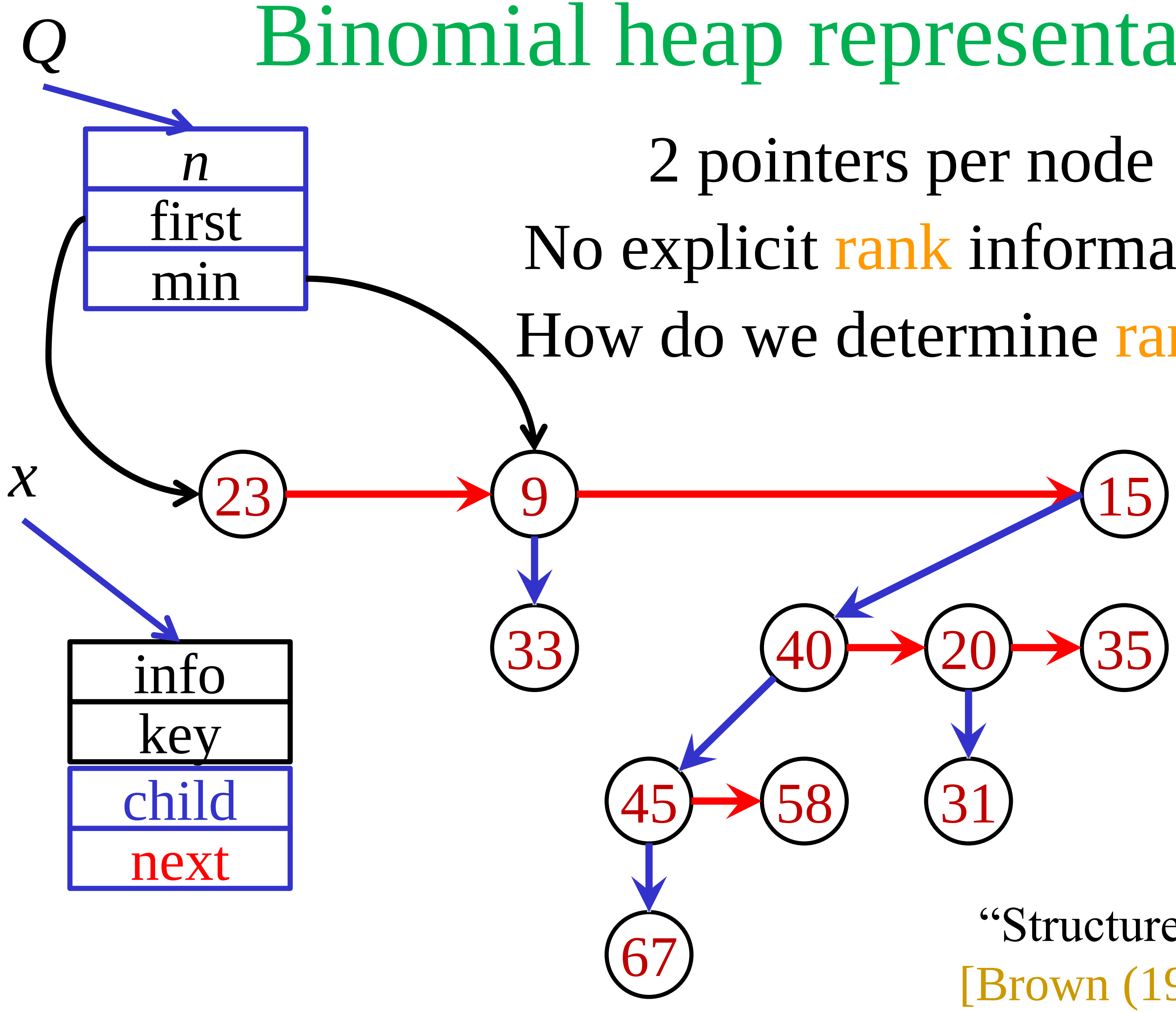
Heap order → "half ordered"

Binomial heap representation

2 pointers per node

No explicit rank information

How do we determine ranks?



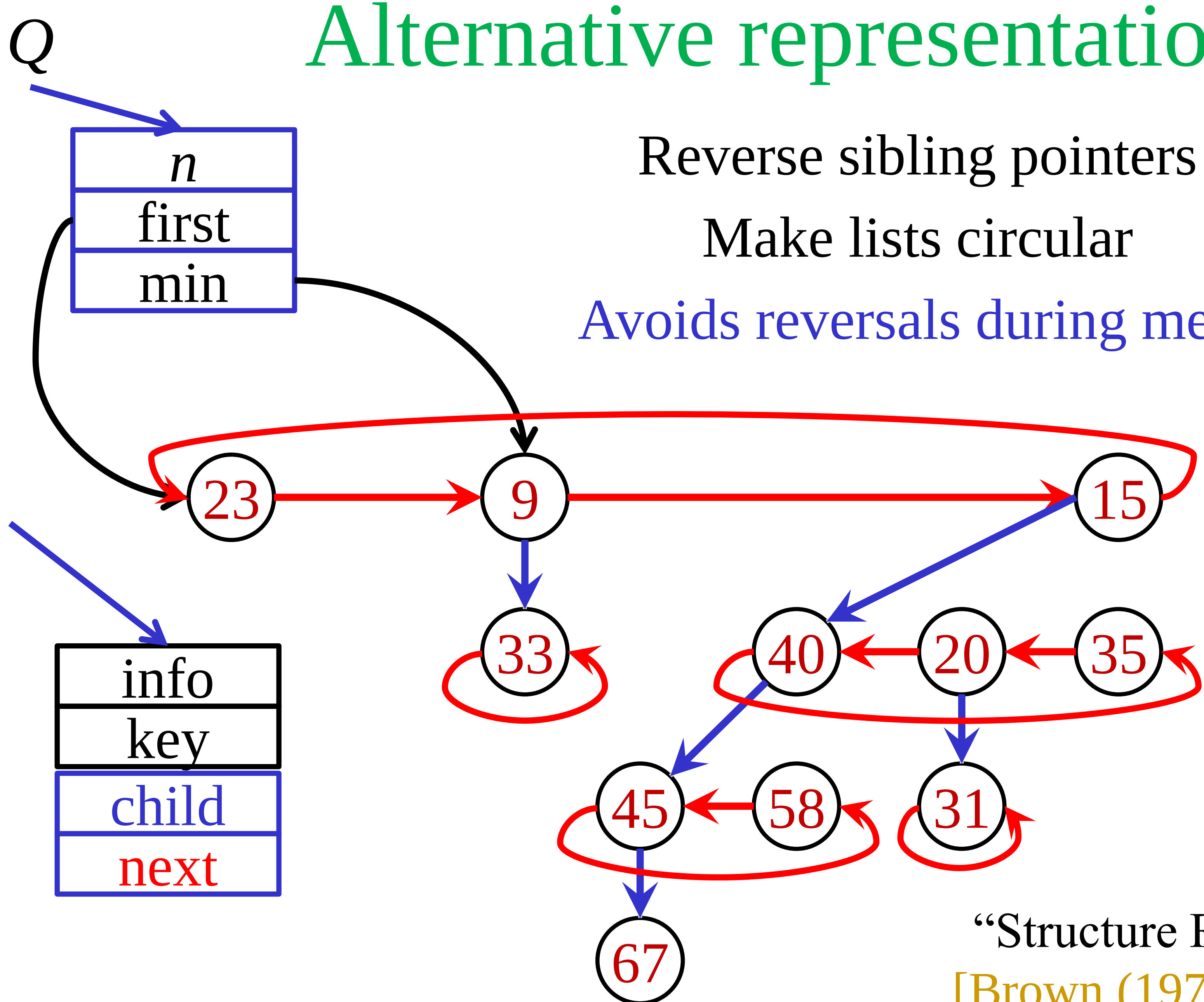
"Structure V"
[Brown (1978)]

Alternative representation

Reverse sibling pointers

Make lists circular

Avoids reversals during meld

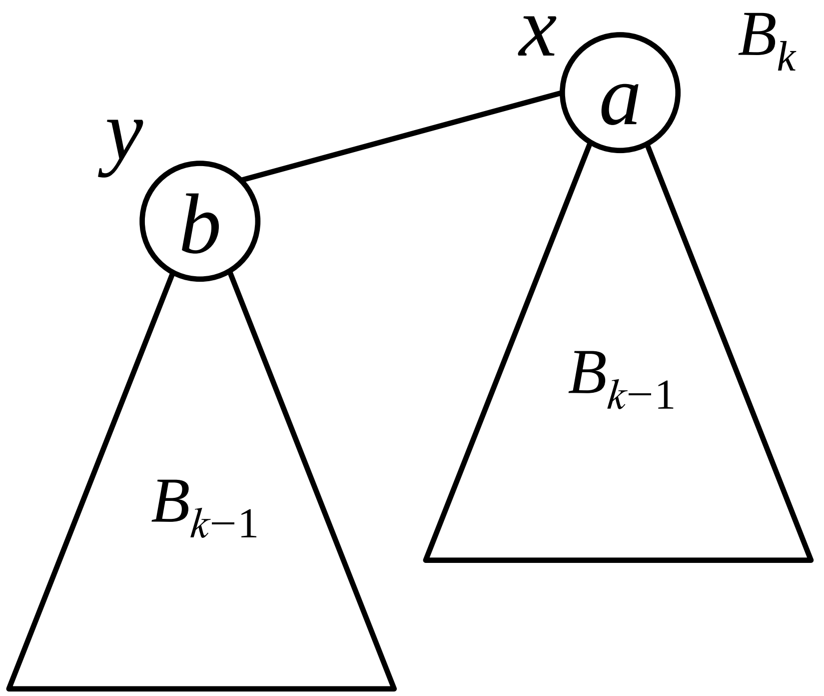


"Structure R"

[Brown (1978)]

Linking binomial trees

$a \le b$



O(1) time

Linking binomial trees

Function `link`(x, y)

if $x.key > y.key$ **then**
  $x \leftrightarrow y$
$y.next \leftarrow x.child$
$x.child \leftarrow y$
return x

Linking in first representation

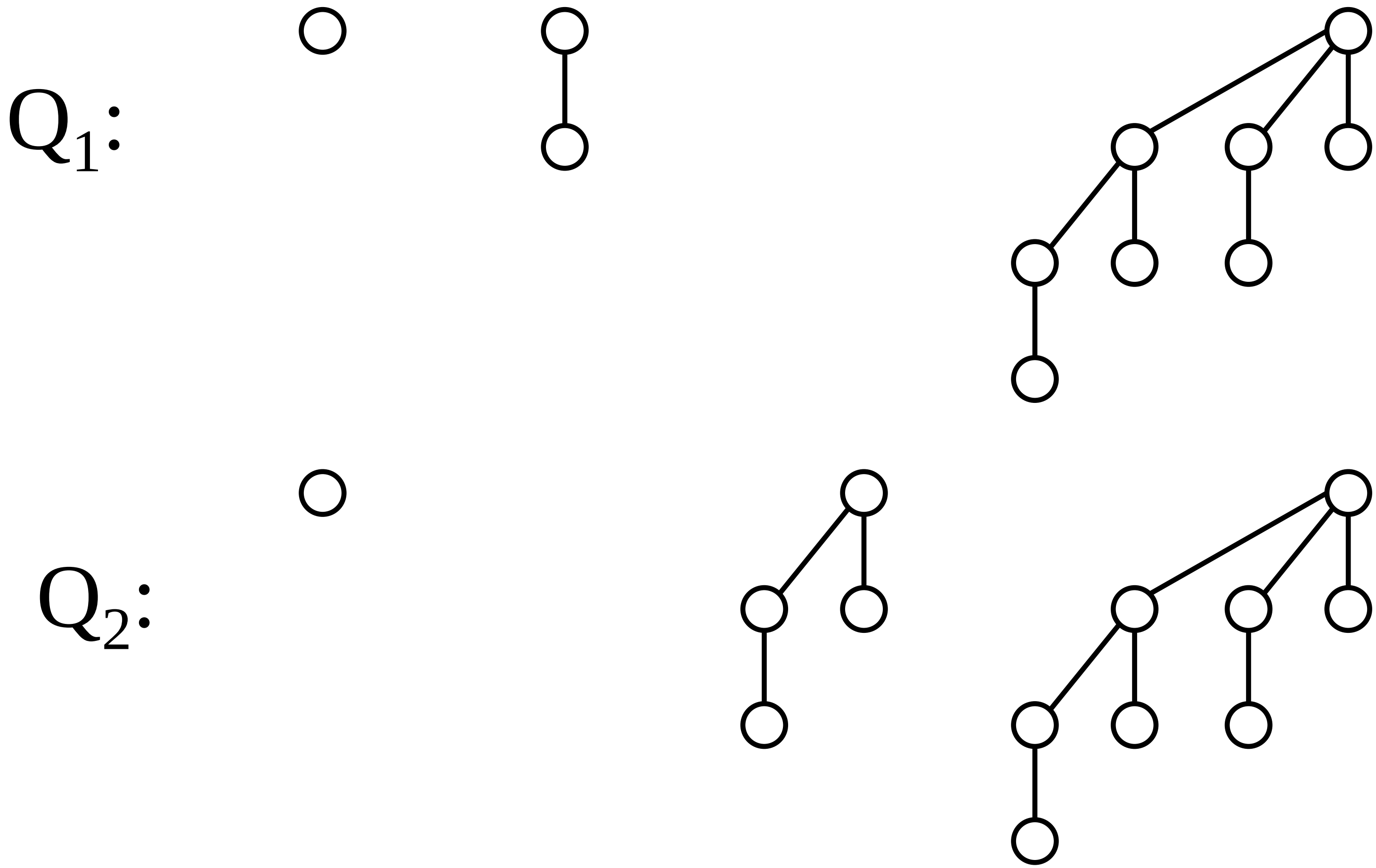
Function `link`(x, y)

if $x.key > y.key$ **then**
  $x \leftrightarrow y$
if $x.child = null$ **then**
  $y.next \leftarrow y$
else
  $y.next \leftarrow x.child.next$
  $x.child.next \leftarrow y$
$x.child \leftarrow y$
return x

Linking in second representation

Melding binomial heaps

Link trees of same degree



Melding binomial heaps

Link trees of same degree

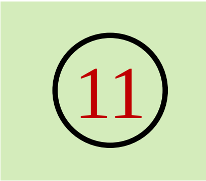
		B_1	B_2	B_3	
Q_1:	B_0	B_1	–	B_3	
Q_2:	B_0	–	B_2	B_3	
	–	–	–	B_3	B_4

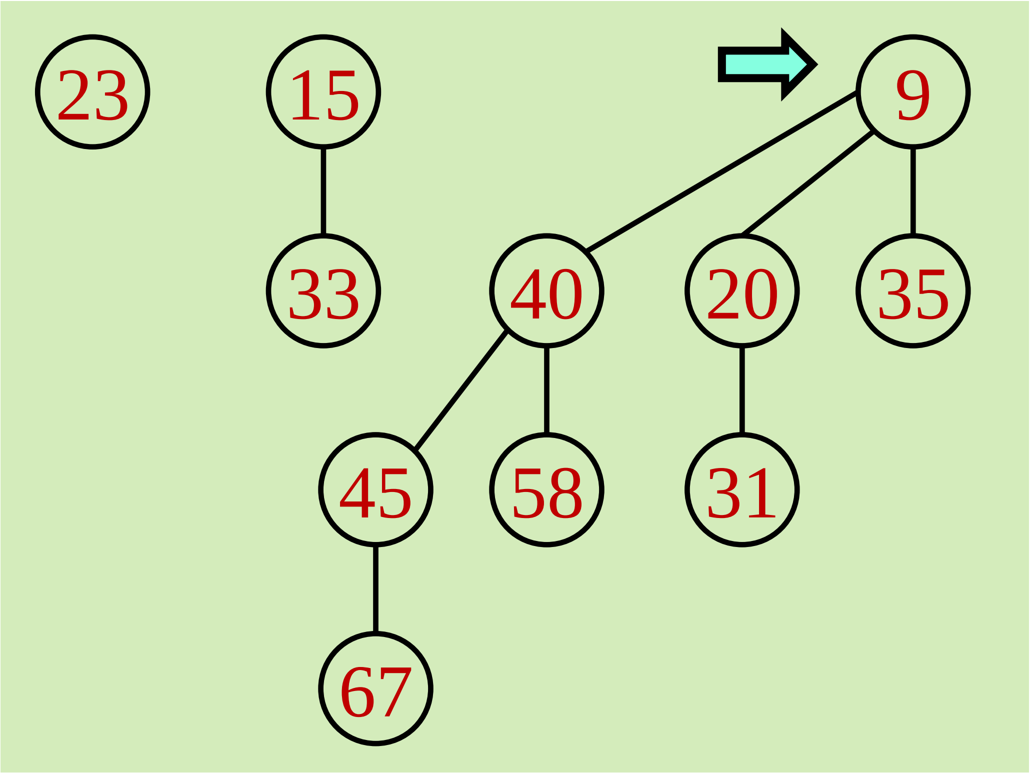
Like adding binary numbers

Maintain a pointer to the minimum

$O(\log n)$ time

Insert



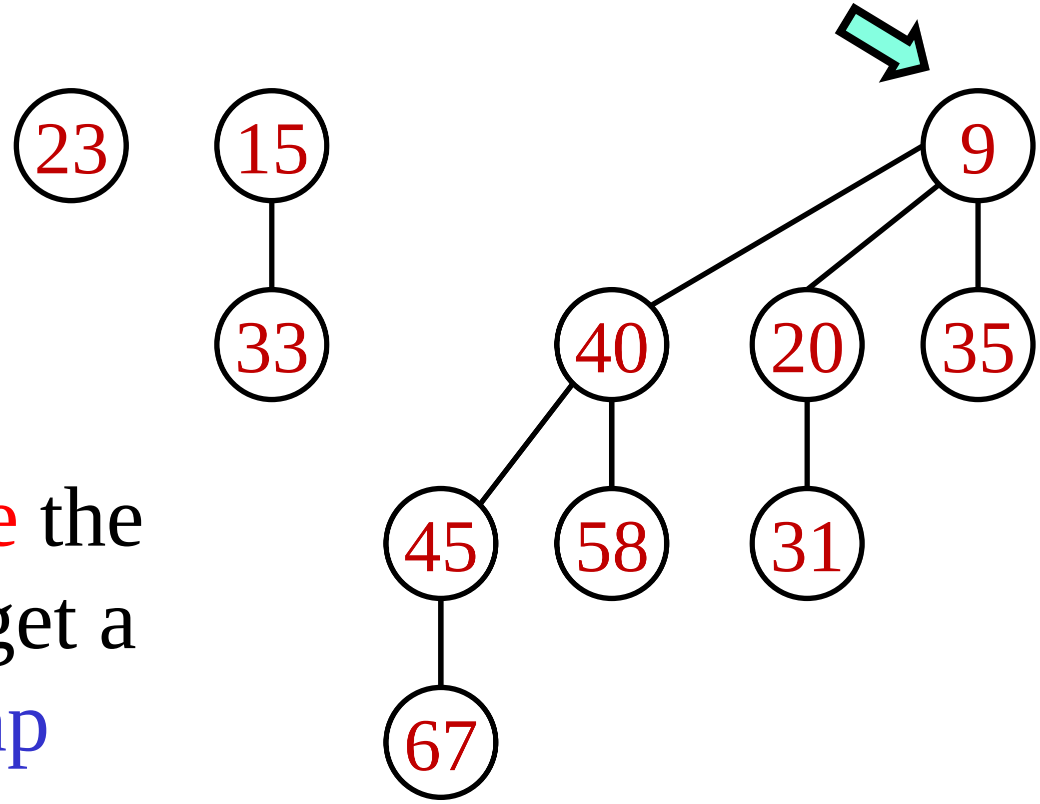


New item is a one tree binomial heap

Meld it to the original heap

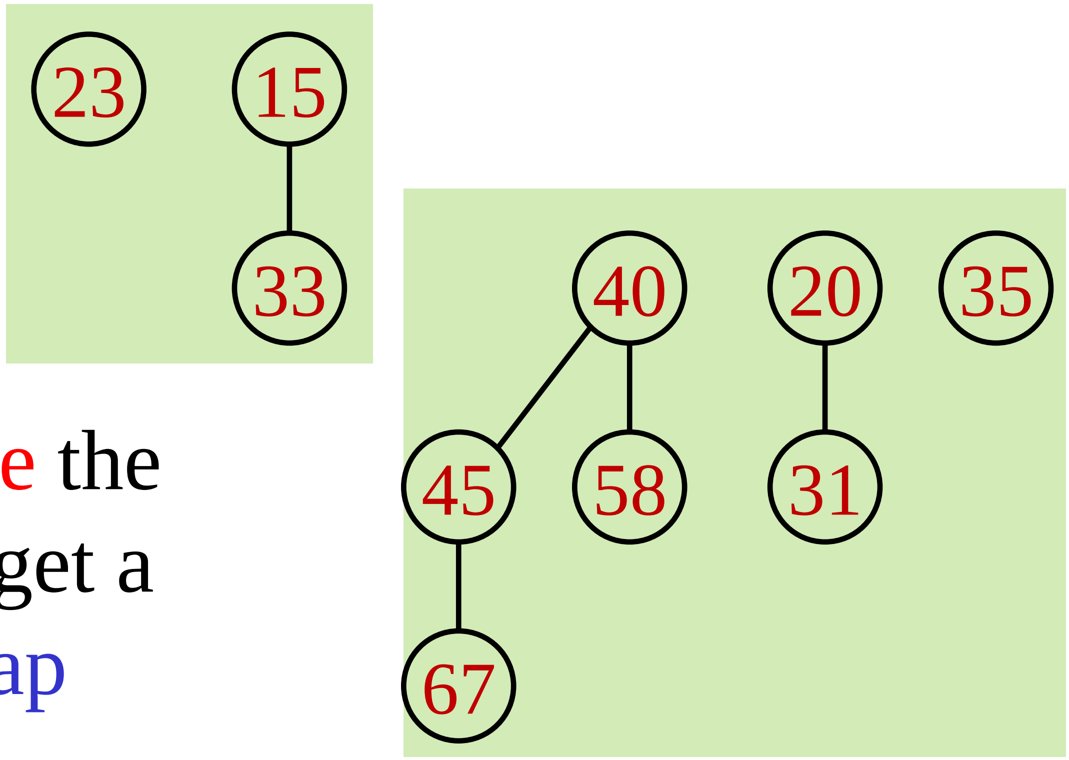
$O(\log n)$ time

Delete-min



When we delete the minimum, we get a binomial heap

Delete-min



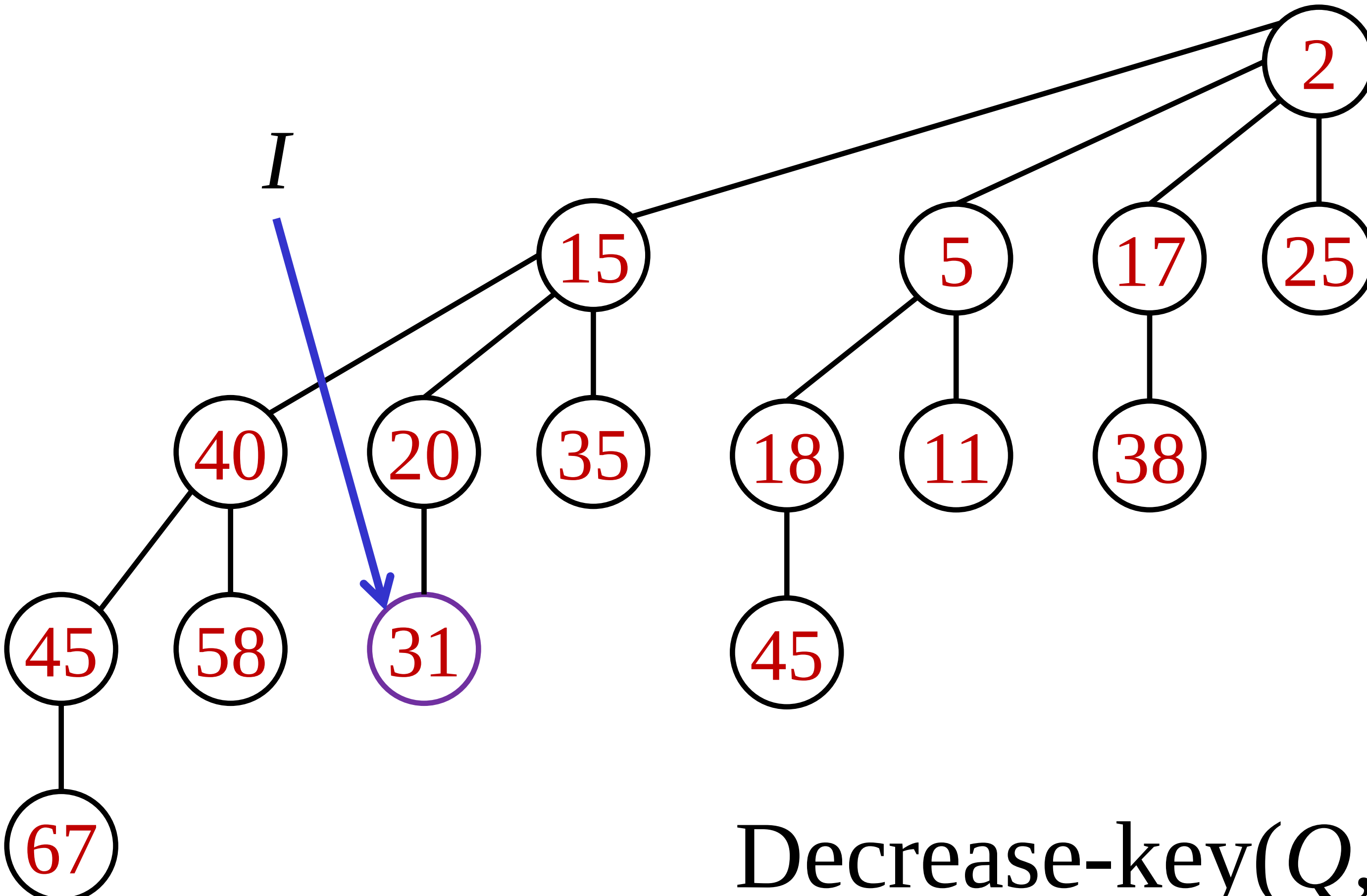
When we delete the minimum, we get a binomial heap

Meld it to the original heap

(Need to reverse list of roots in first representation)

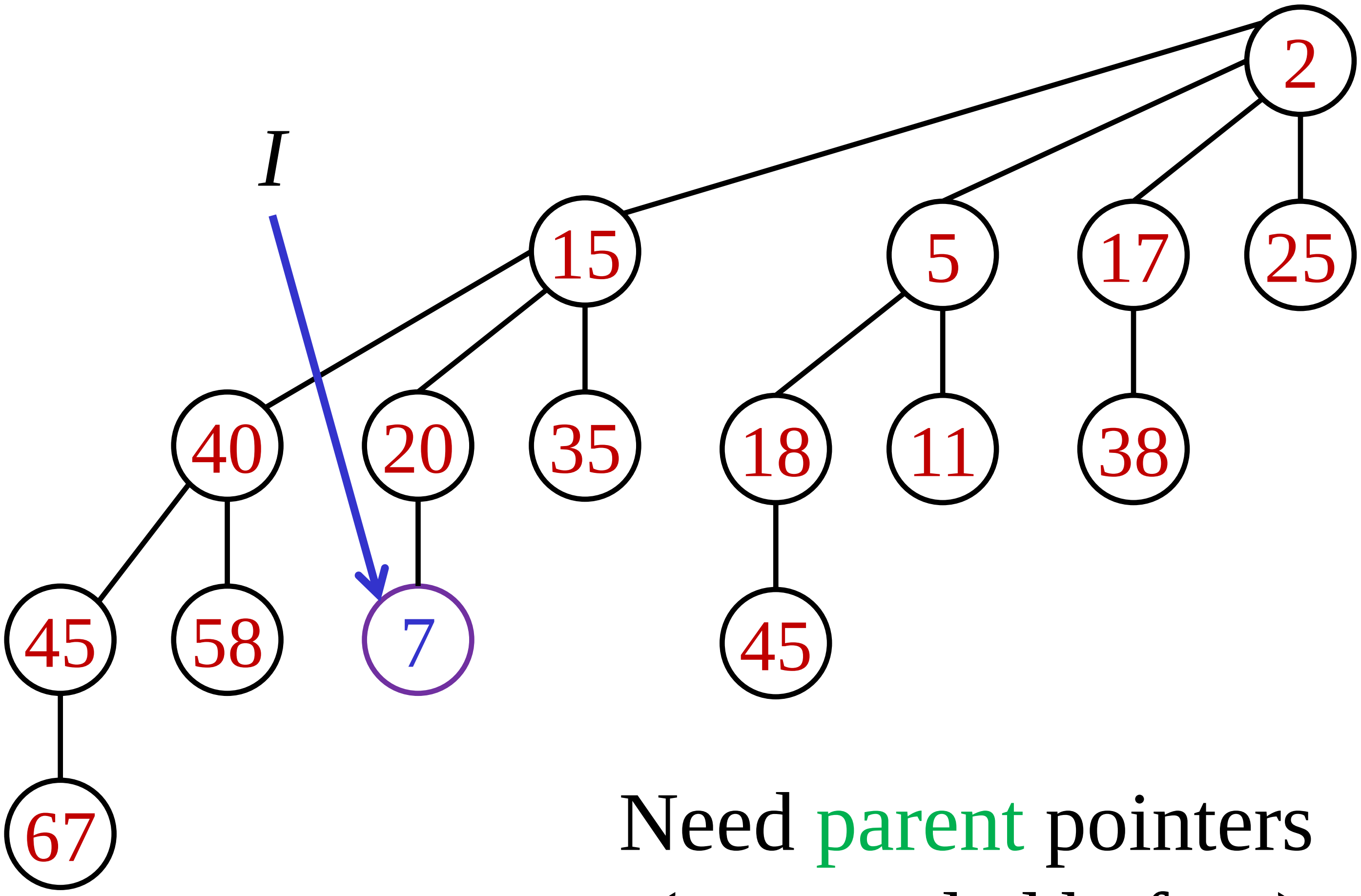
O(log$n$) time

Decrease-key using "sift-up"



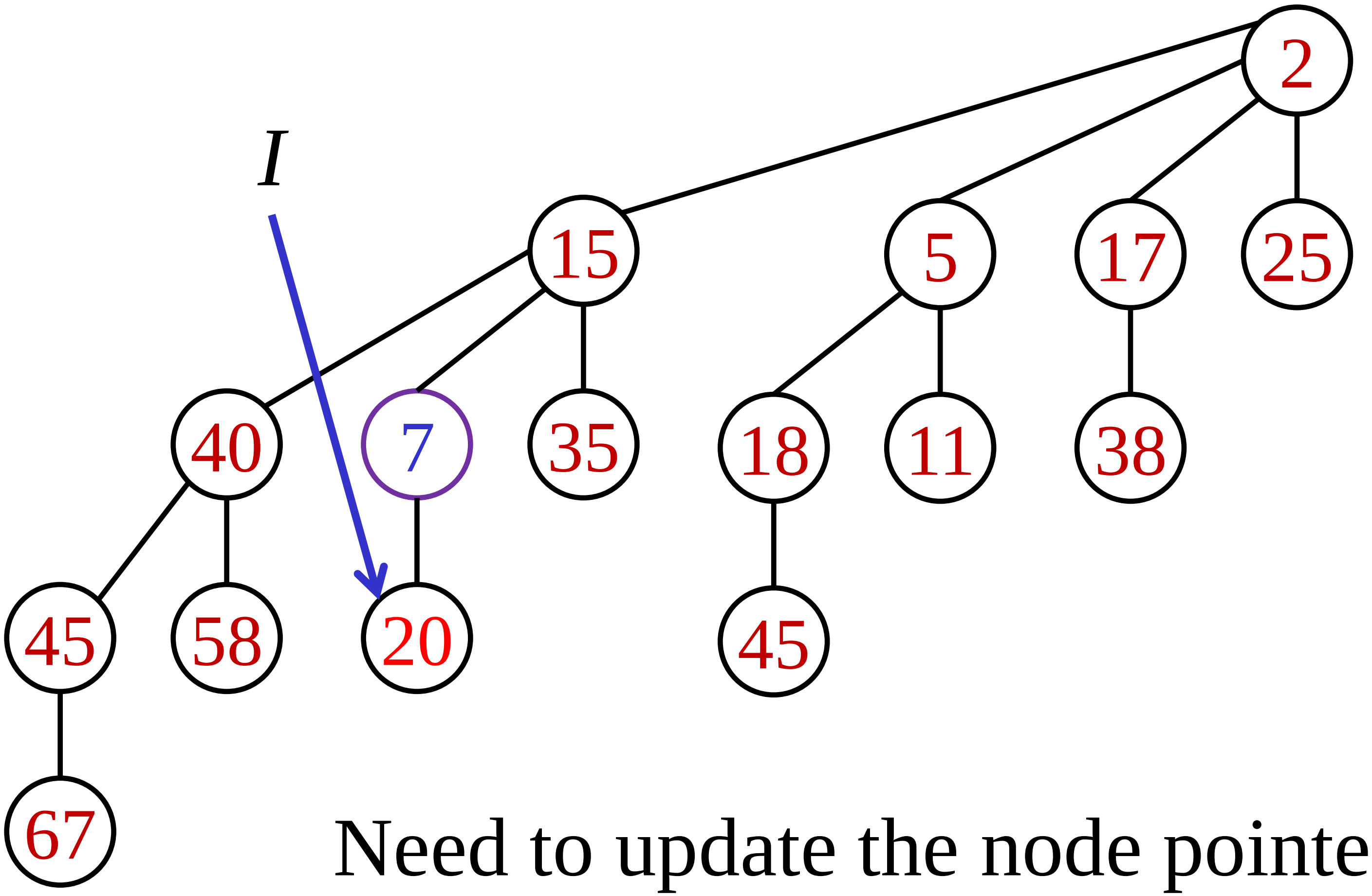
Decrease-key(Q,I,7)

Decrease-key using “sift-up”



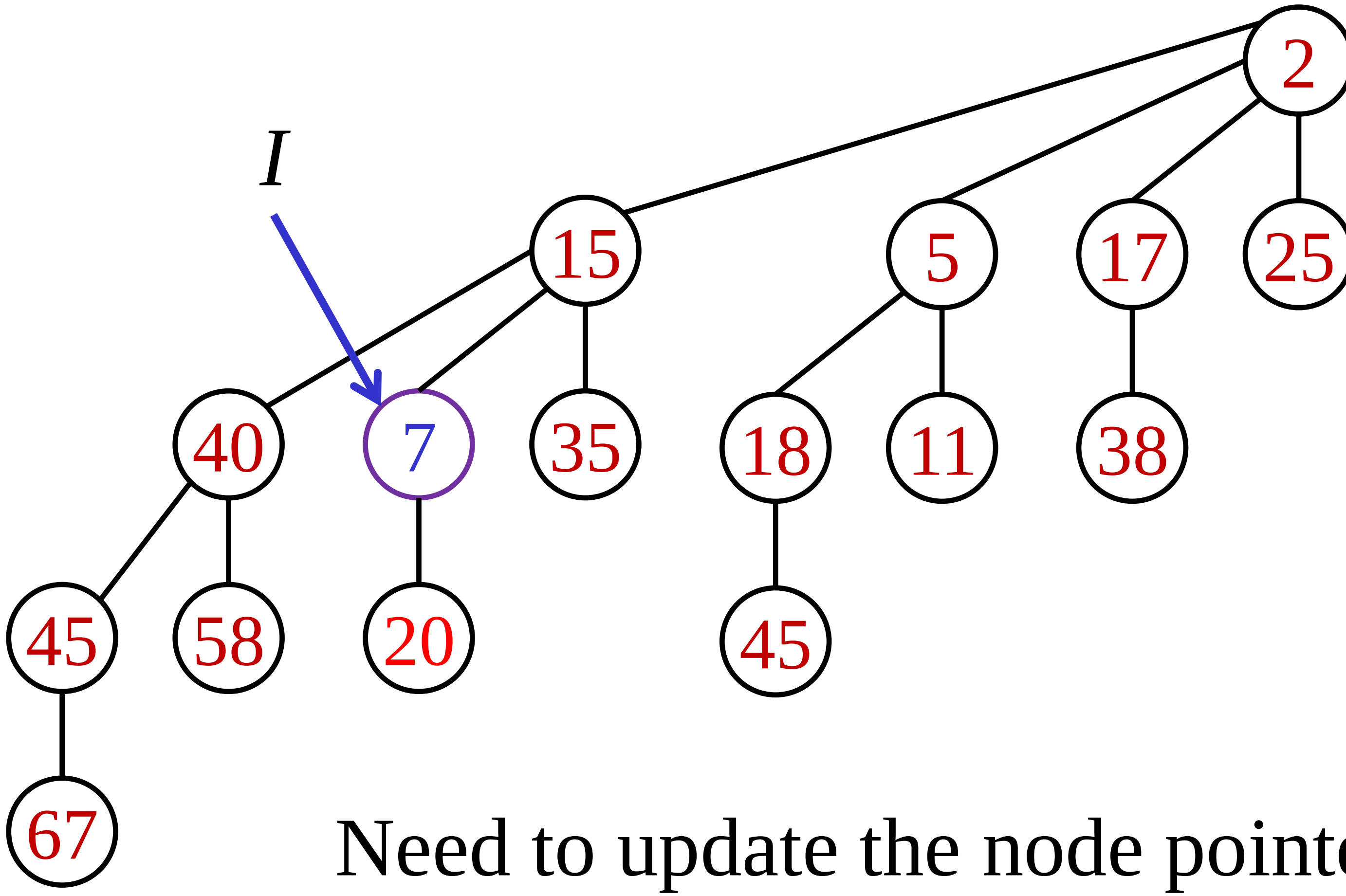
Need parent pointers
(not needed before)

Decrease-key using “sift-up”



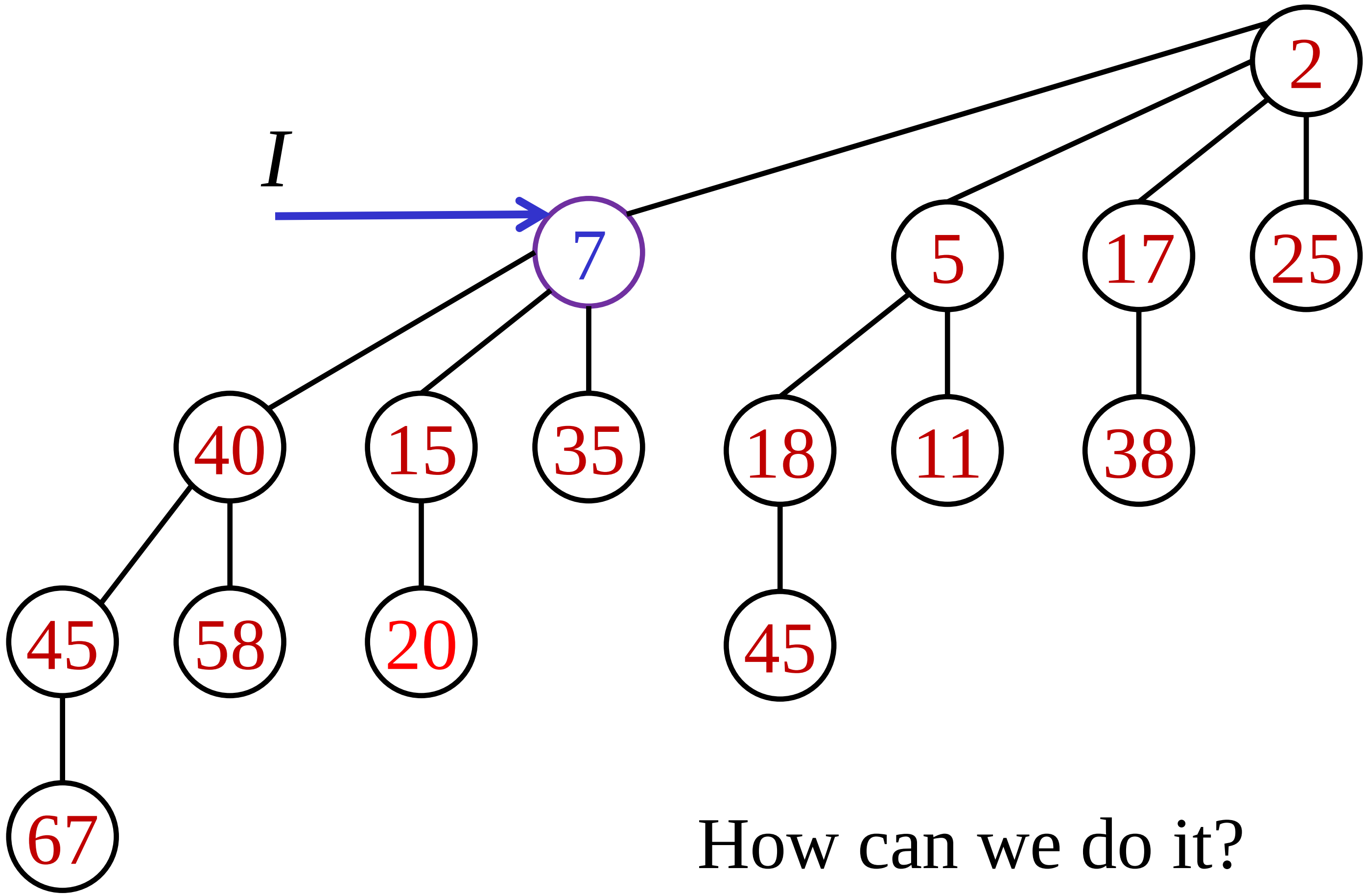
Need to update the node pointed by *I*

Decrease-key using "sift-up"



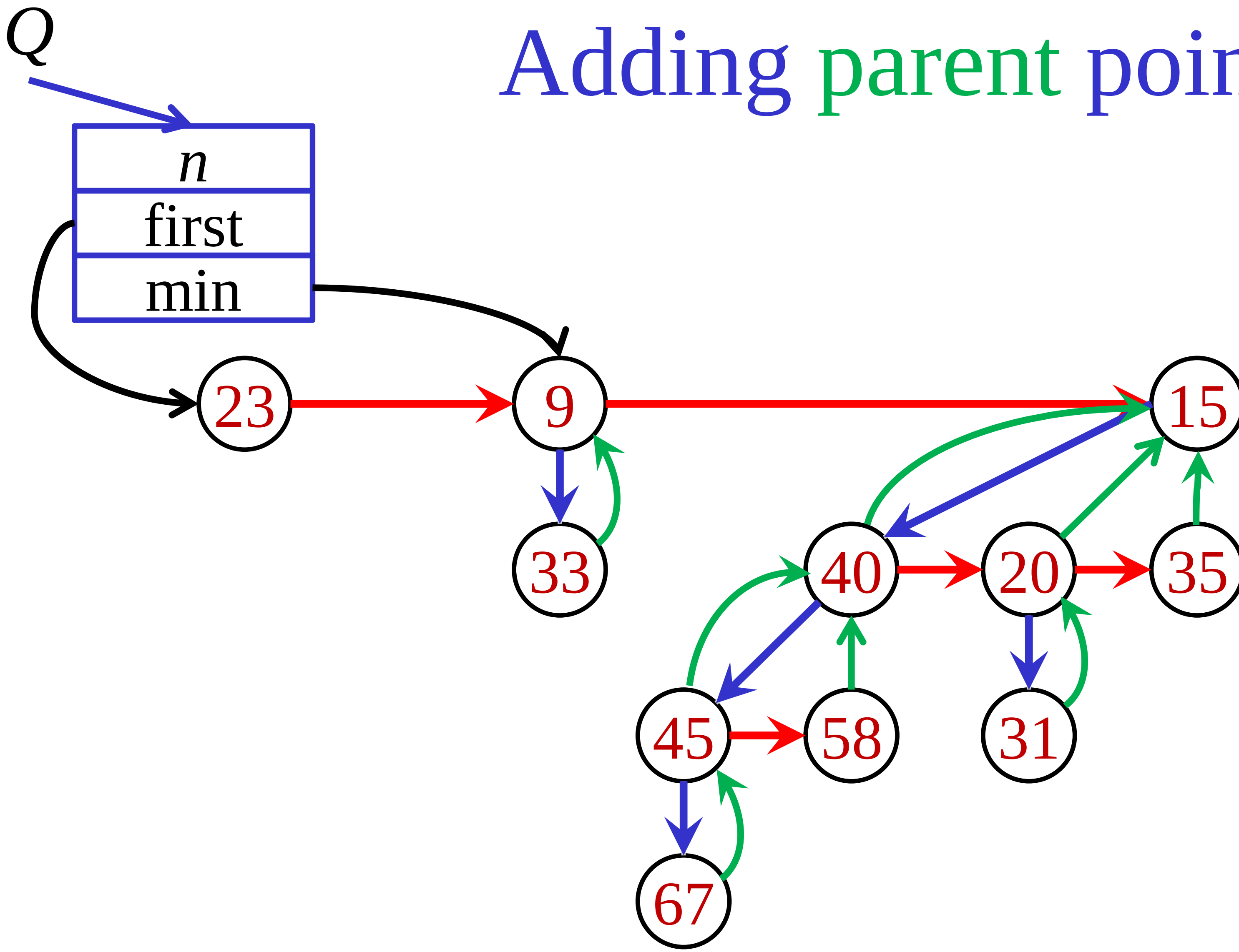
Need to update the node pointed by *I*

Decrease-key using "sift-up"



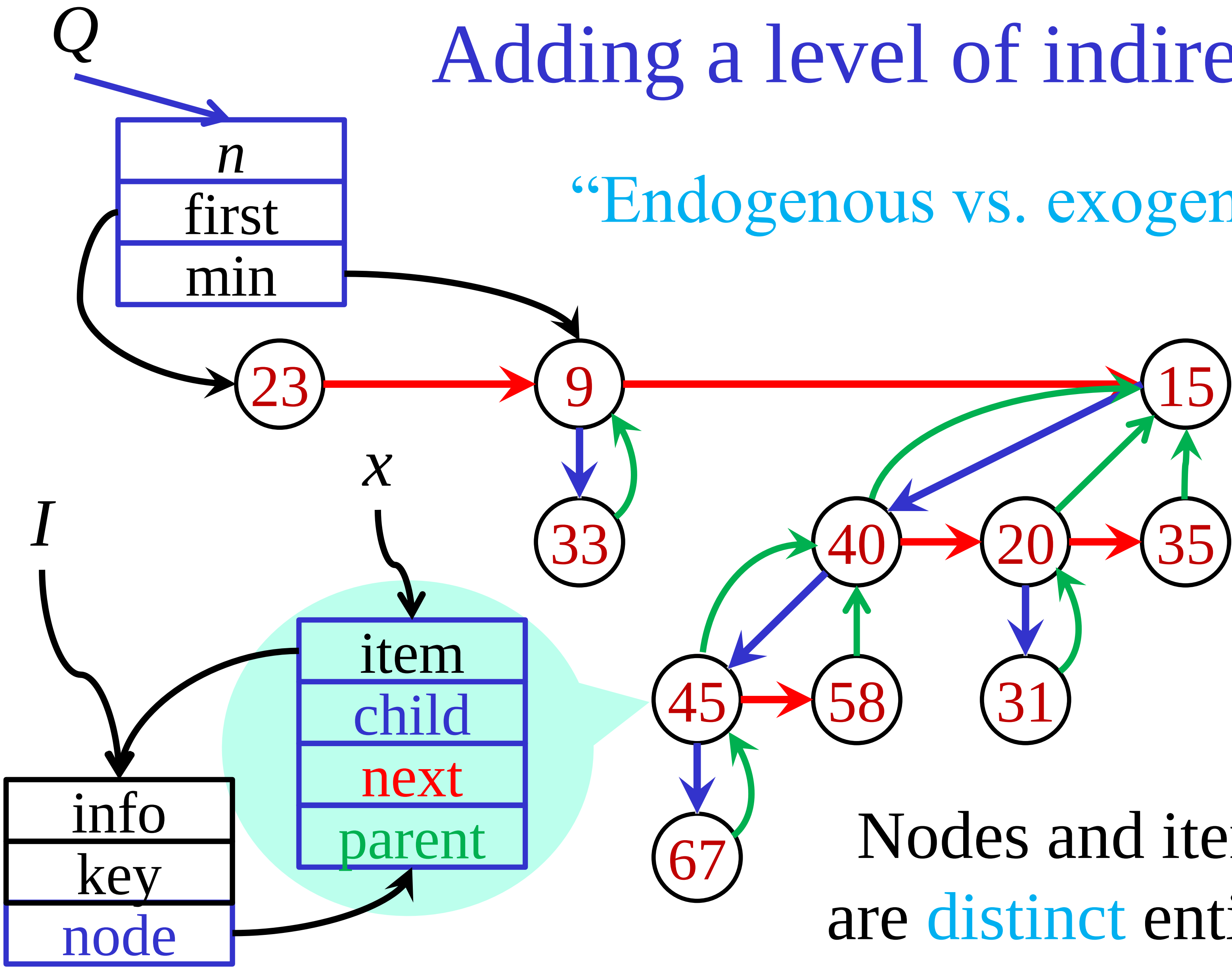
How can we do it?

Adding parent pointers



Adding a level of indirection

"Endogenous vs. exogenous"

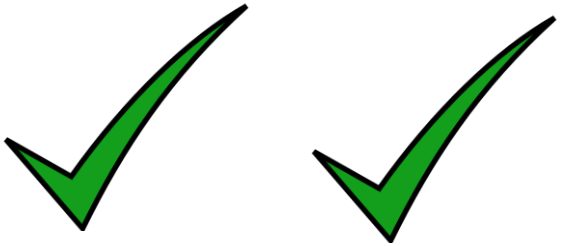


Nodes and items are distinct entities

Heaps / Priority queues

	Binary Heaps	**Binomial Heaps**	**Lazy Binomial Heaps**	**Fibonacci Heaps**
Insert	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
Find-min	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Delete-min	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Decrease-key	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Meld	–	$O(\log n)$	$O(1)$	$O(1)$

Worst case | Amortized



Lazy Binomial Heaps

Binomial Heaps

A list of binomial trees,
at most one of each rank, sorted by rank
(at most $O(\log n)$ trees)

Pointer to root with minimal key

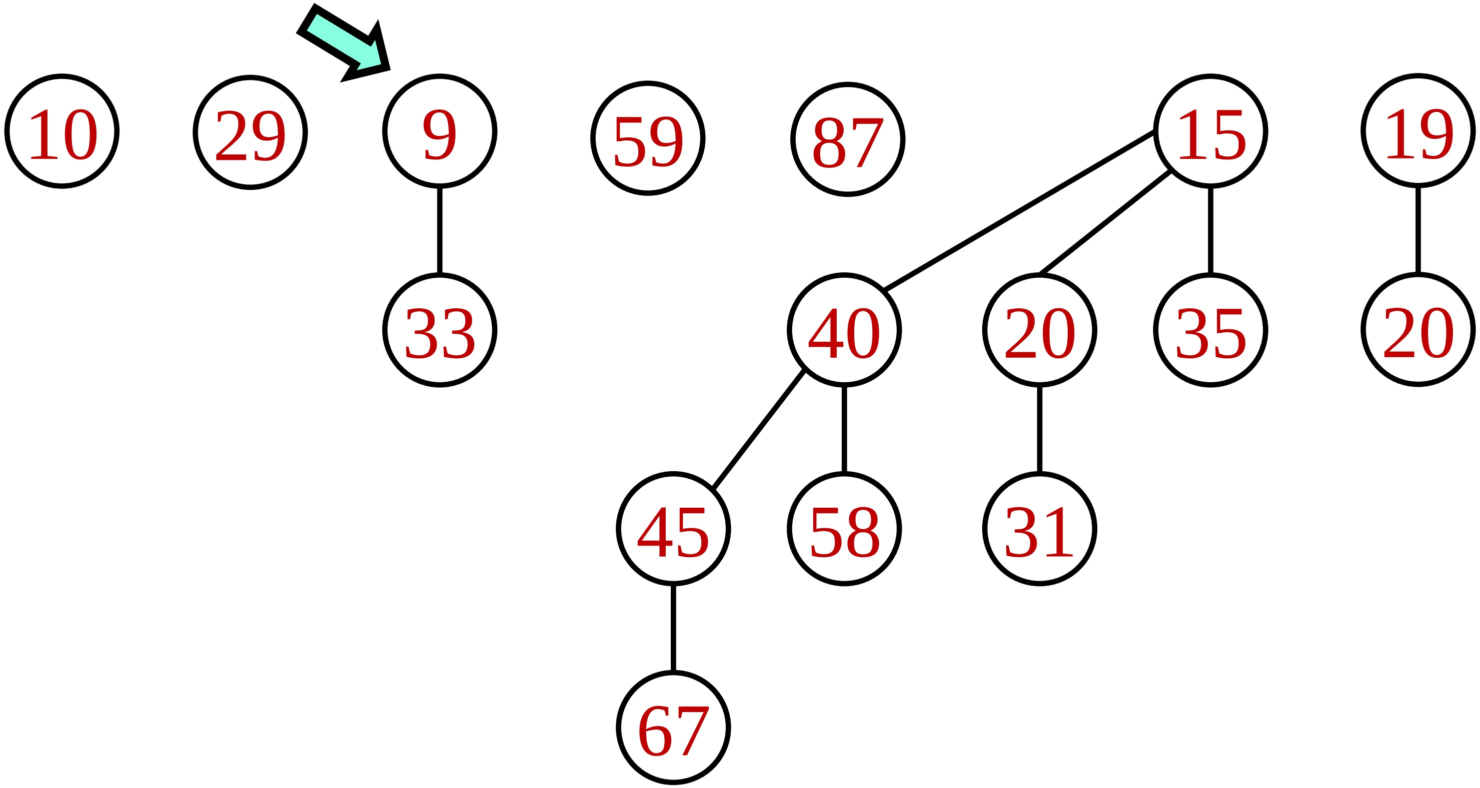
Lazy Binomial Heaps

An arbitrary list of binomial trees
(possibly n trees of size 1)

Pointer to root with minimal key

Lazy Binomial Heaps

An arbitrary list of binomial trees
Pointer to root with minimal key



Lazy Meld

Concatenate the two lists of trees

Update the pointer to root with minimal key

$O(1)$ worst case time

Lazy Insert

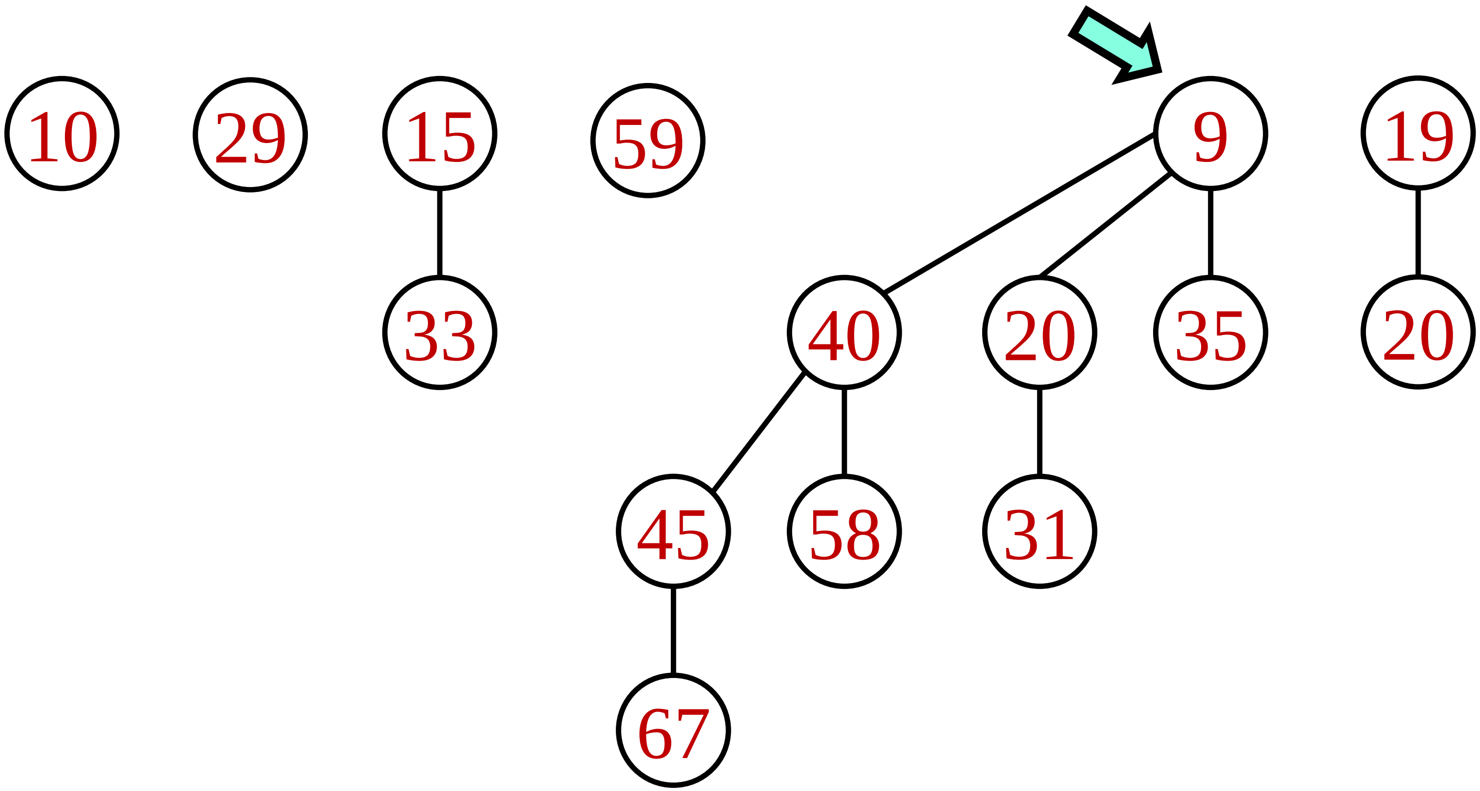
Add the new item to the list of roots

Update the pointer to root with minimal key

$O(1)$ worst case time

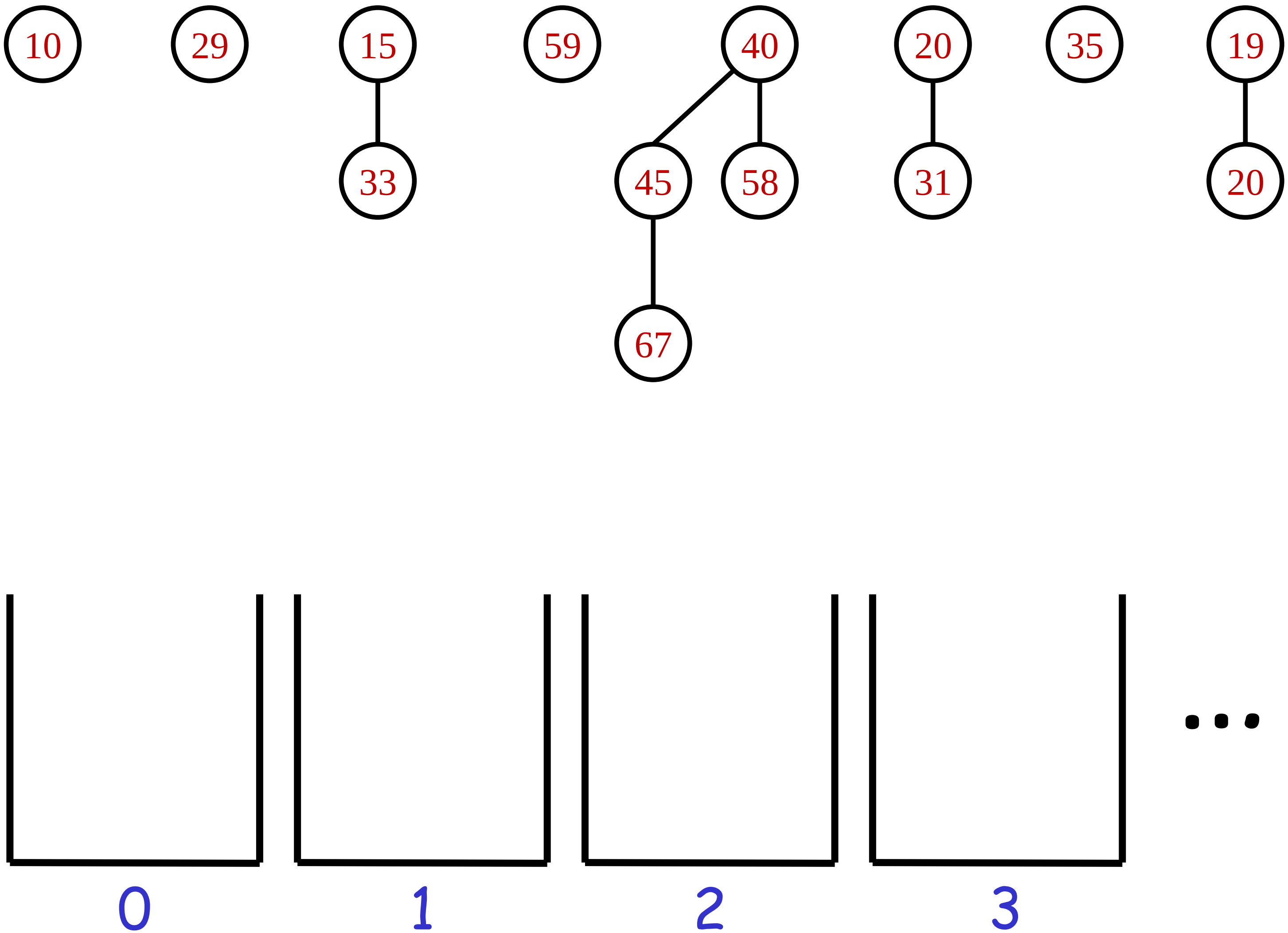
Lazy Delete-min ?

Remove the minimum root and meld ?

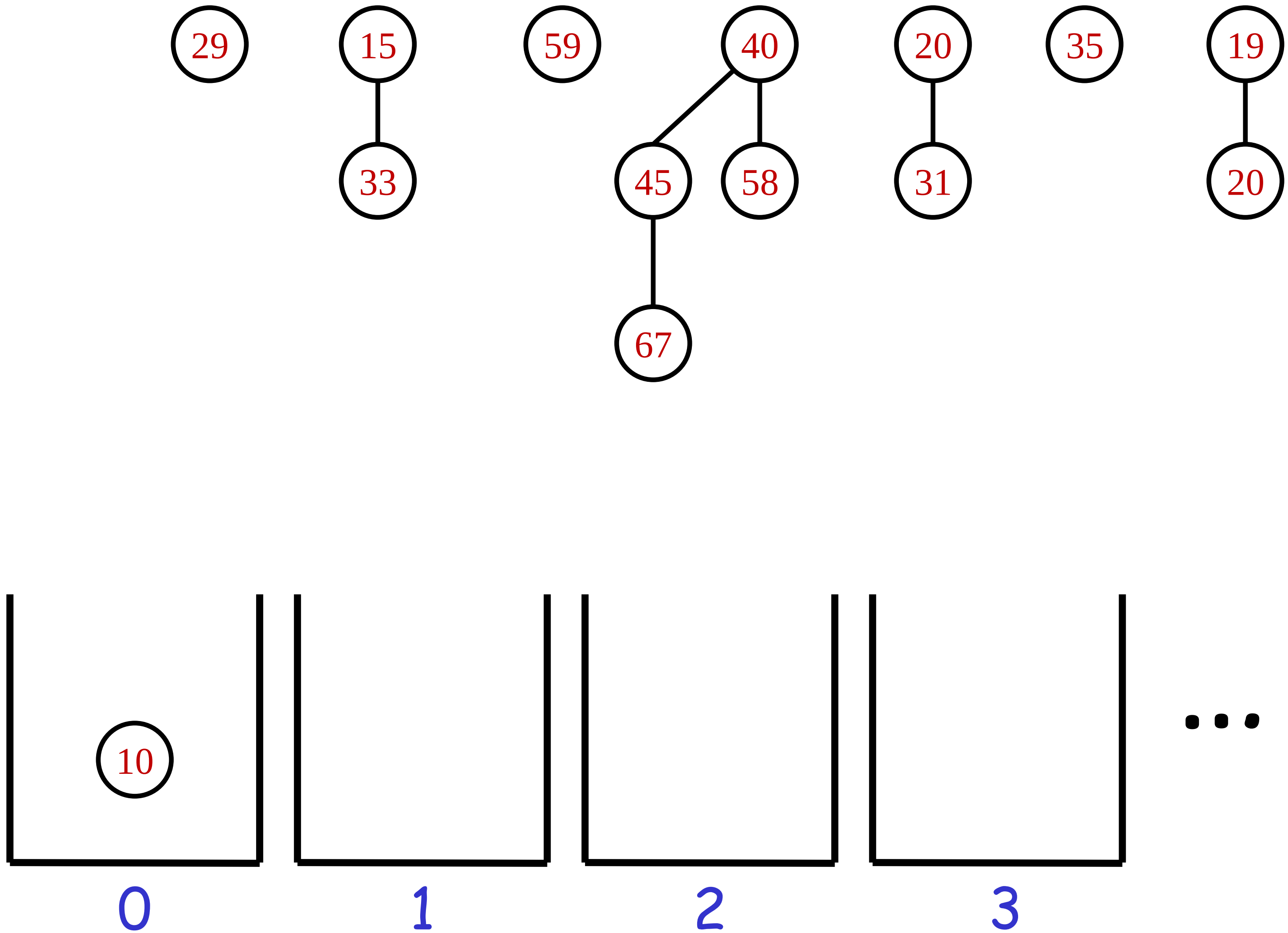


May need $\Omega(n)$ time to find the new minimum

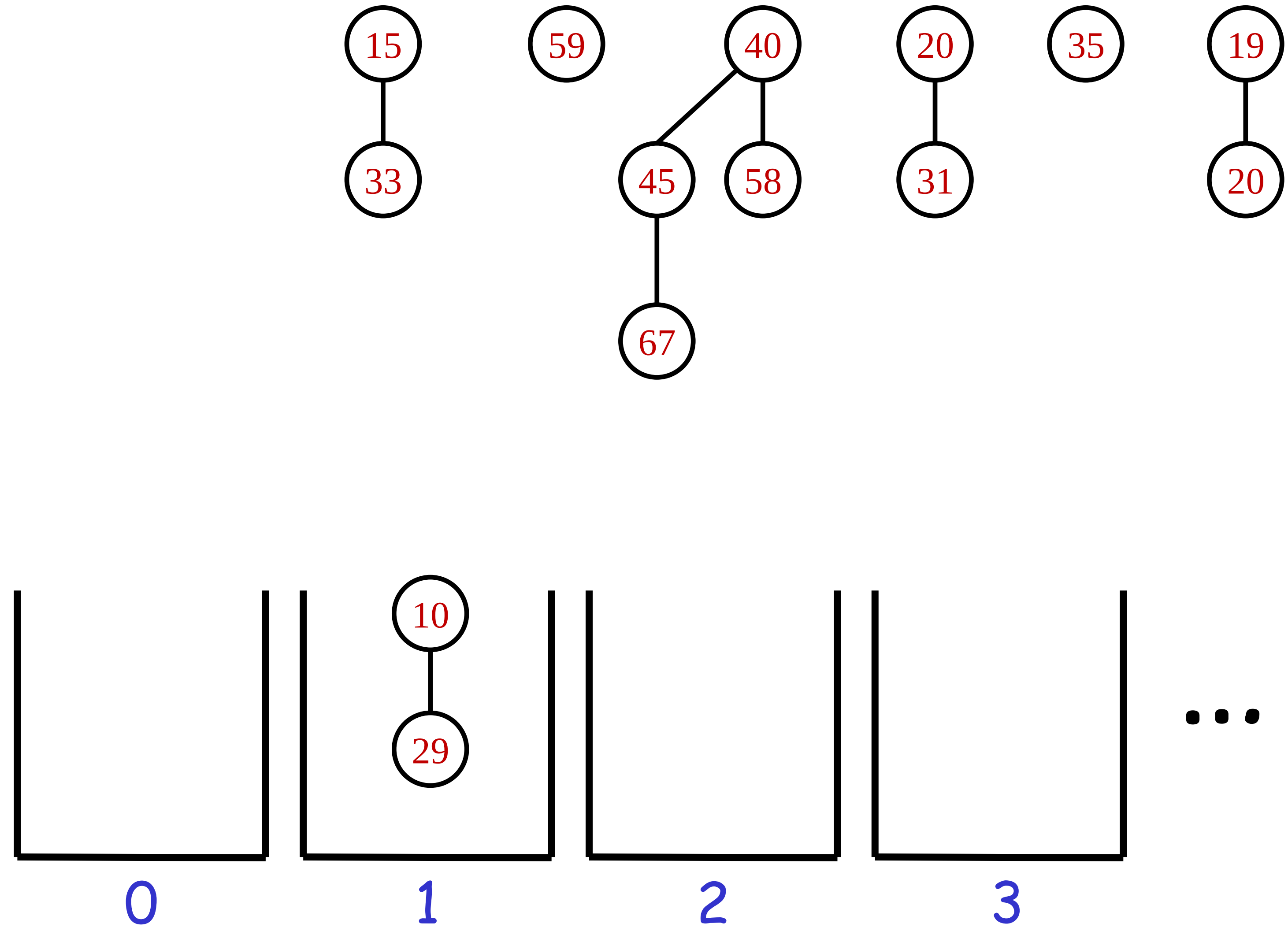
Consolidating / Successive Linking



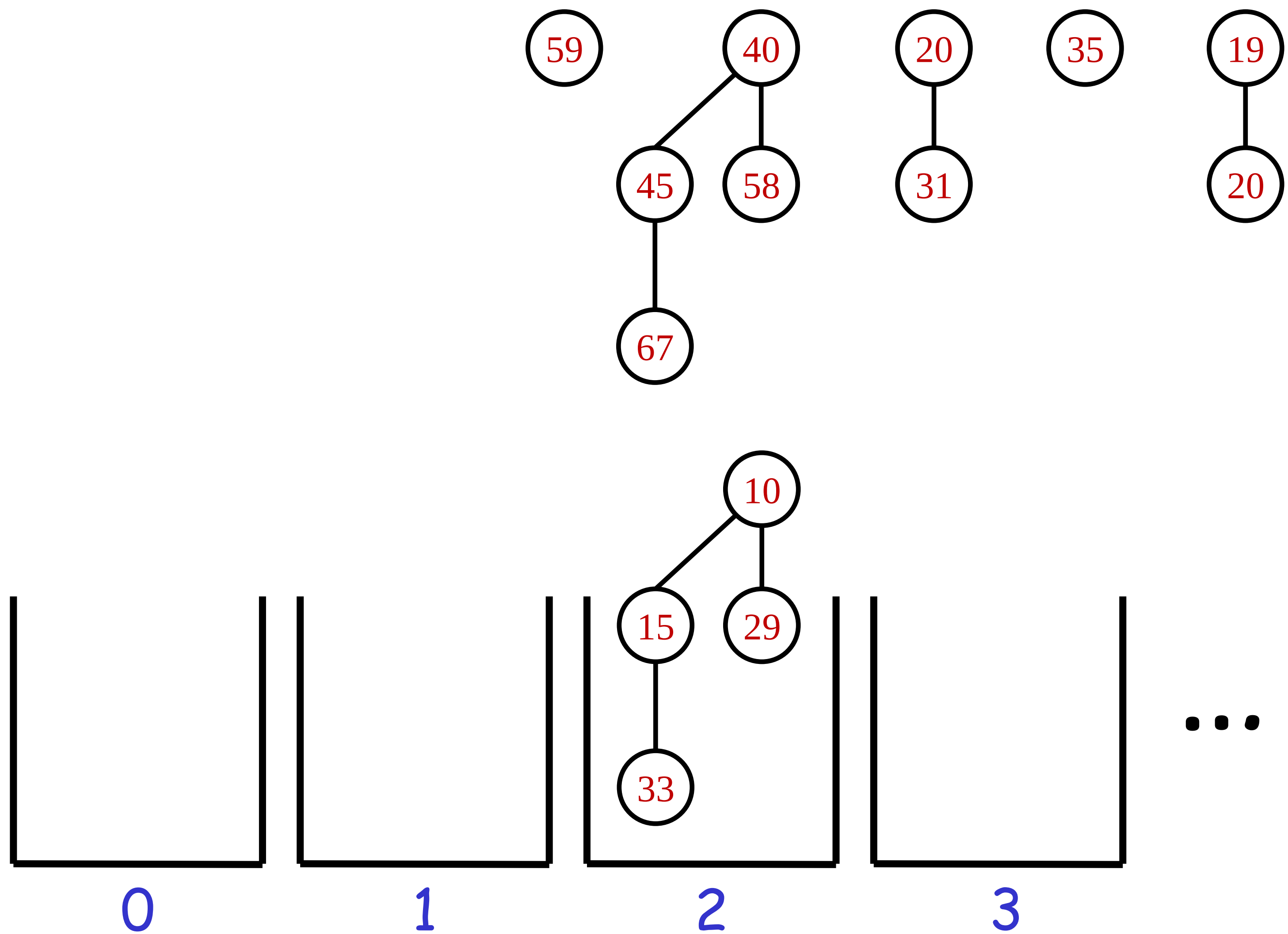
Consolidating / Successive Linking



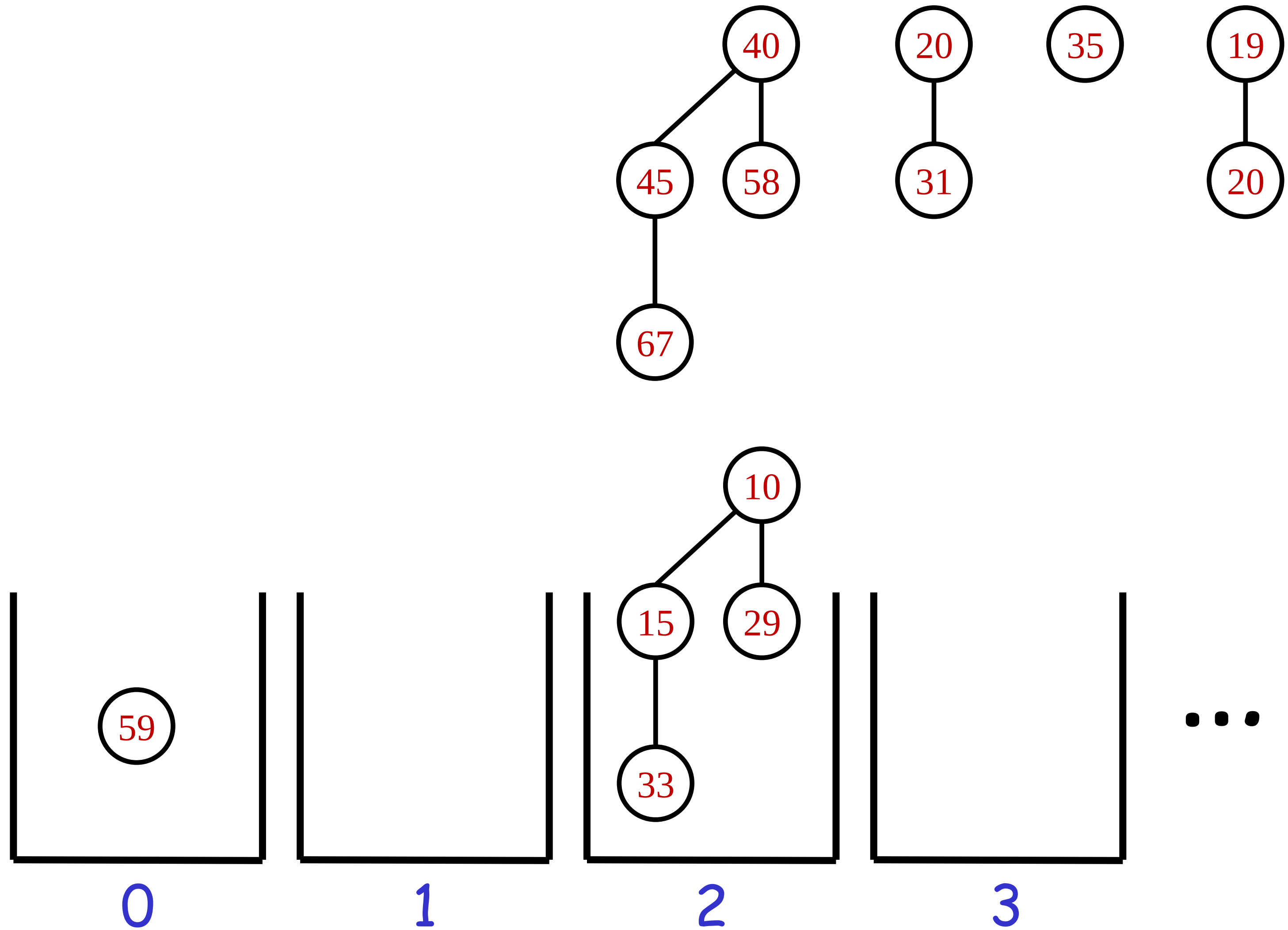
Consolidating / Successive Linking



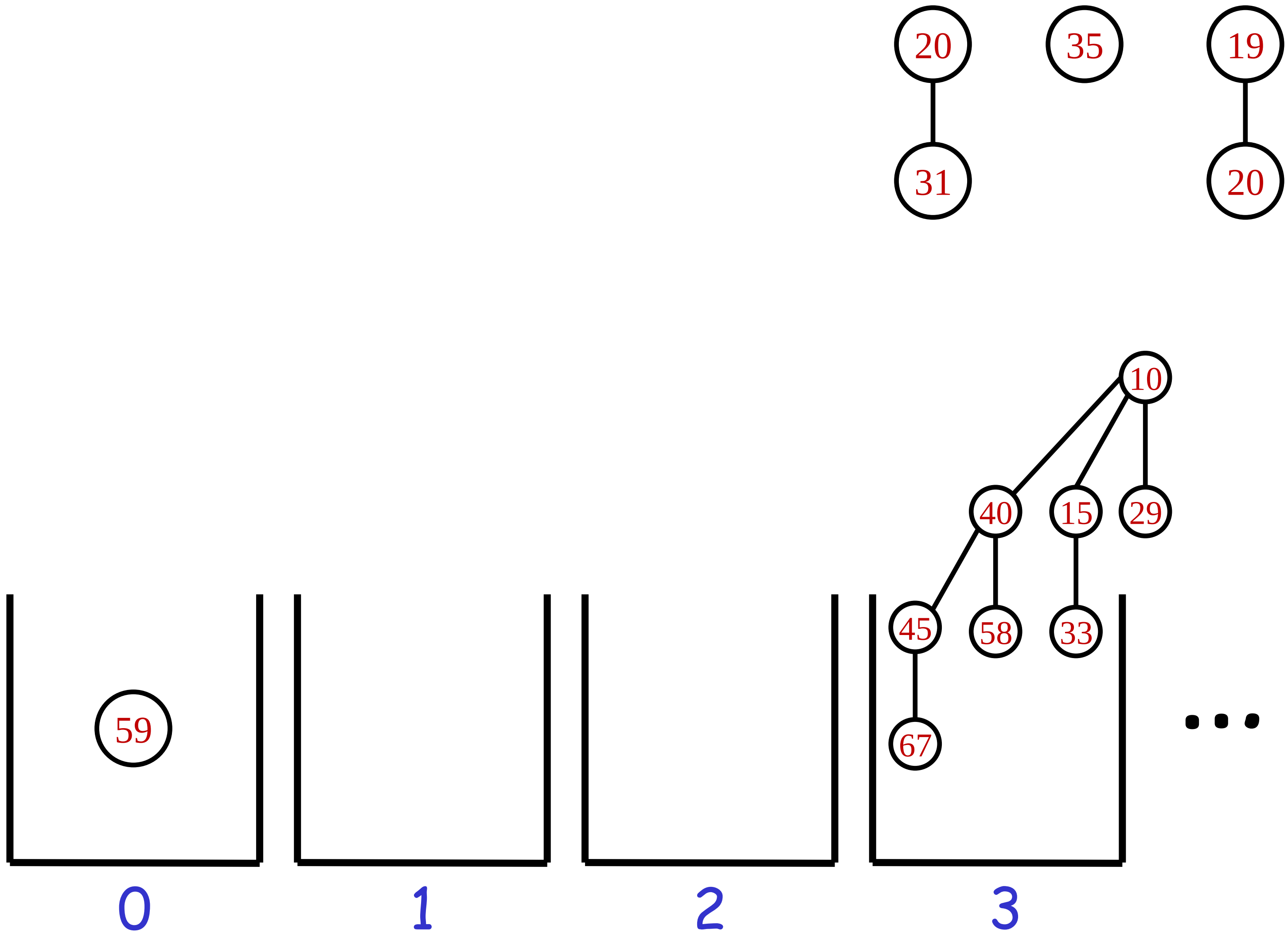
Consolidating / Successive Linking



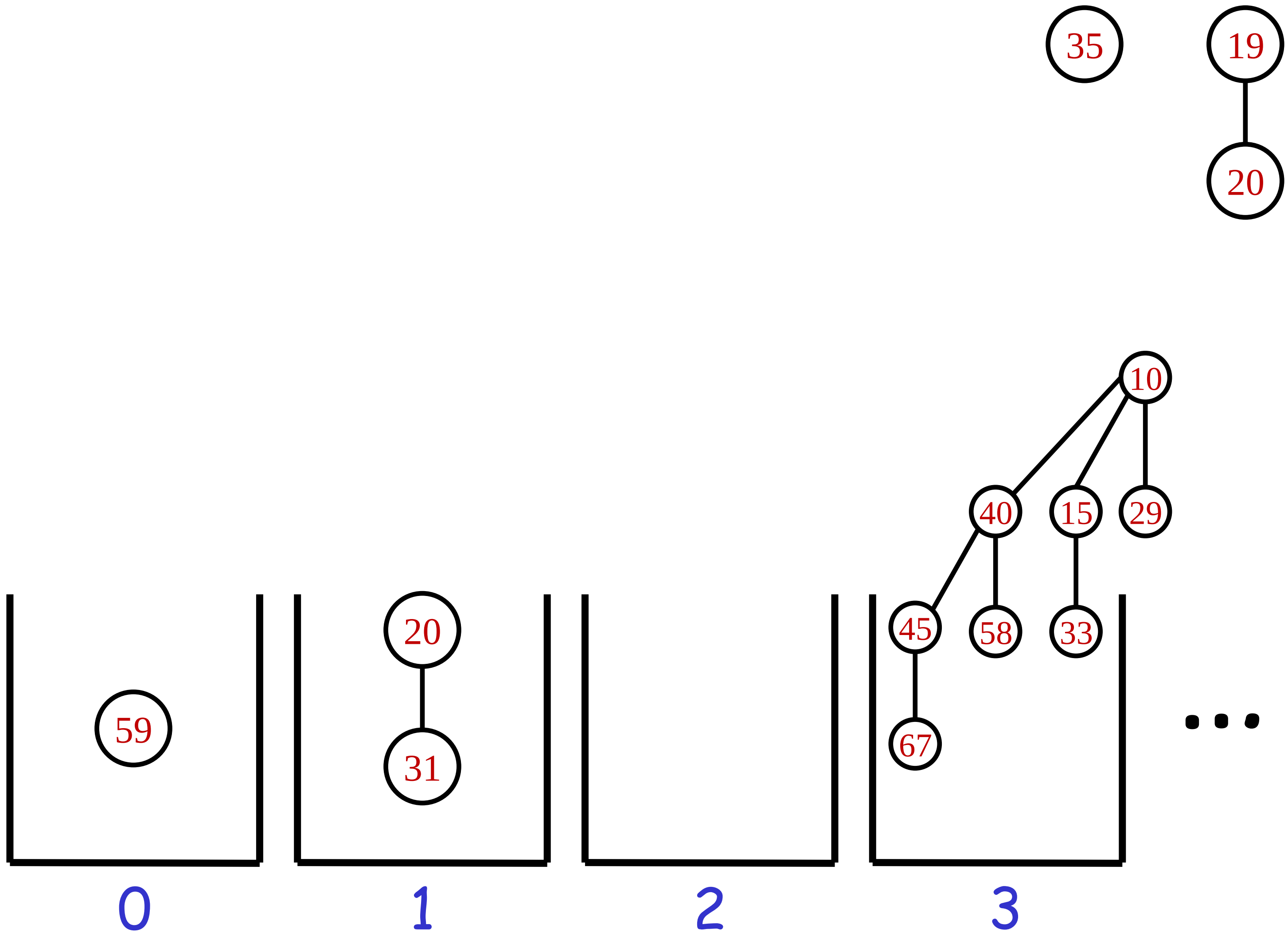
Consolidating / Successive Linking



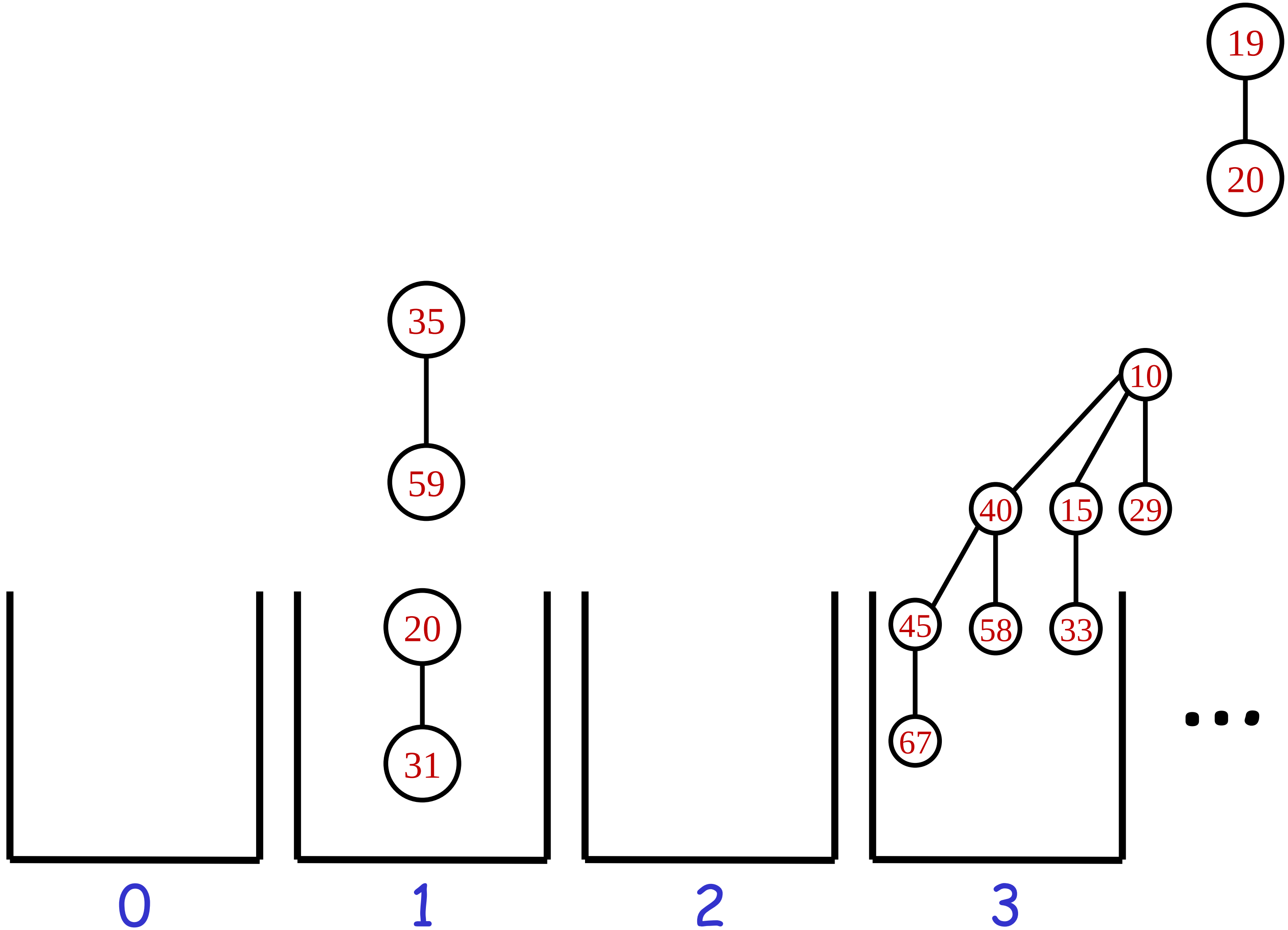
Consolidating / Successive Linking



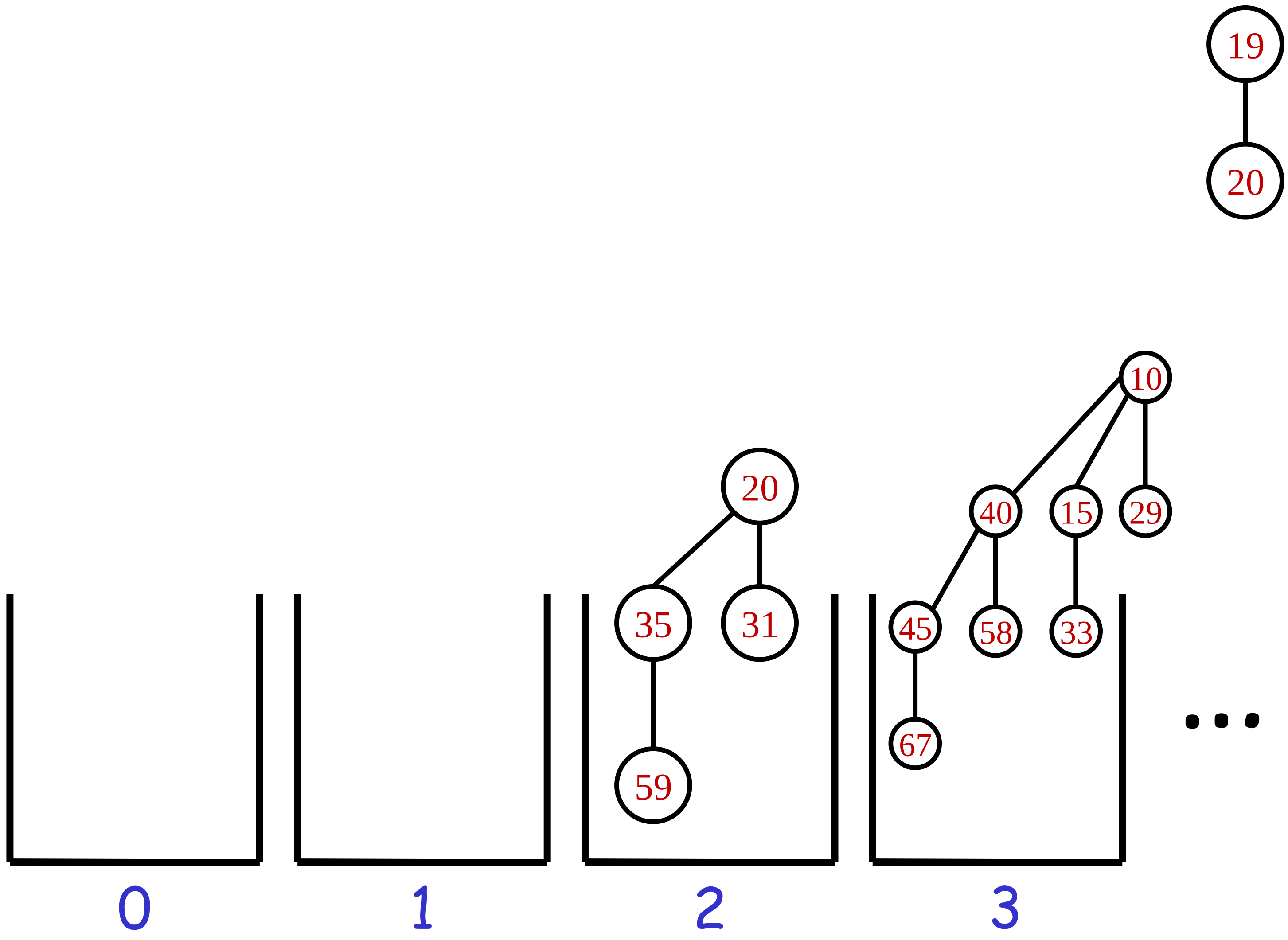
Consolidating / Successive Linking



Consolidating / Successive Linking

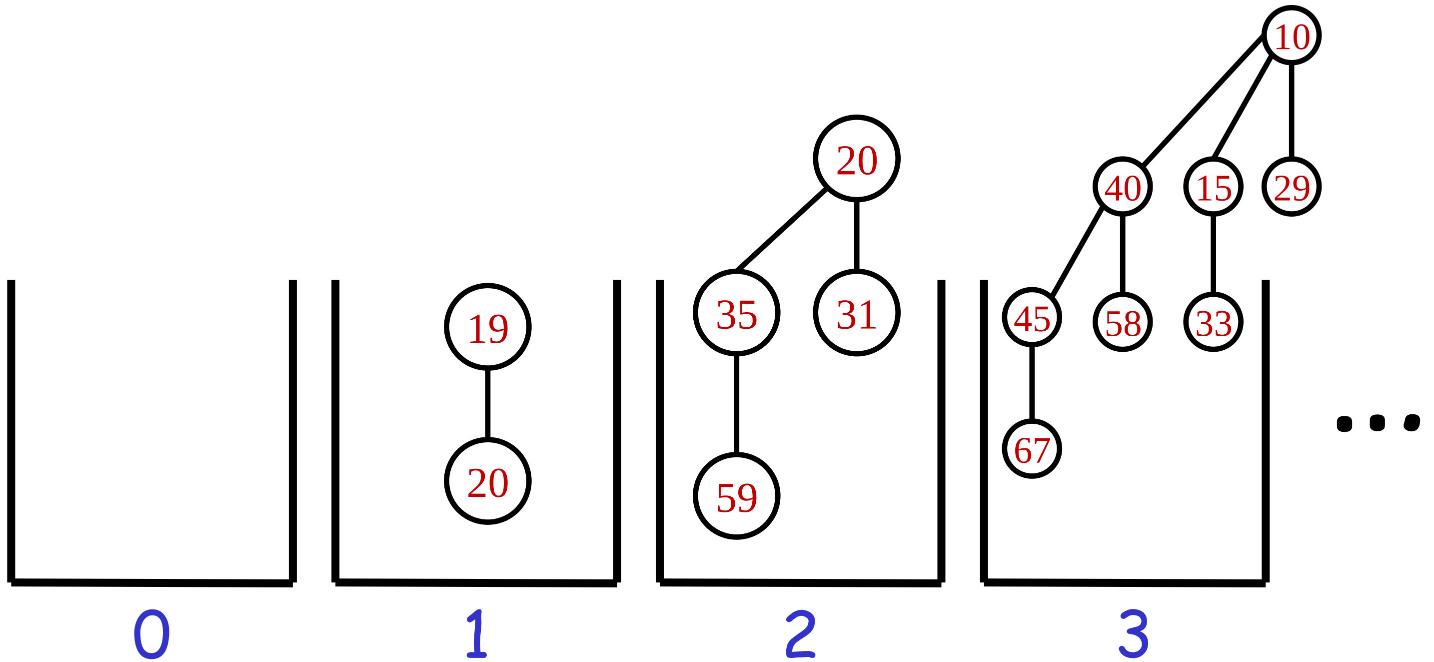


Consolidating / Successive Linking



Consolidating / Successive Linking

At the end of the process, we obtain a non-lazy binomial heap containing at most $\log(n+1)$ trees, at most one of each rank



Consolidating / Successive Linking

At the end of the process, we obtain a non-lazy binomial heap containing at most $\log n$ trees, at most one of each degree

Worst case cost – $O(n)$

Amortized cost – $O(\log n)$

Potential = Number of Trees

Cost of Consolidating

T_0 – Number of trees before

T_1 – Number of trees after

L – Number of links

$T_1 = T_0 - L$ (Each link reduces the number of tree by 1)

Total number of trees processed – $T_0 + L$
(Each link creates a new tree)

Putting trees into buckets or finding trees to link with

Linking

Handling the buckets

$$\text{Total cost} = O(\ (T_0+L) + L + \lceil \log_2 n \rceil\ )$$

$$= O(\ T_0 + \lceil \log_2 n \rceil\ )$$

As $L \le T_0$

Amortized Cost of Consolidating

(Scaled) actual cost = $T_0 + \lceil \log_2 n \rceil$

Change in potential = $\Delta\Phi = T_1 - T_0$

$$\begin{aligned} \text{Amortized cost} &= (T_0 + \lceil \log_2 n \rceil) + (T_1 - T_0) \\ &= T_1 + \lceil \log_2 n \rceil \\ &\leq 2\lceil \log_2 n \rceil \end{aligned}$$

As $T_1 \leq \lceil \log_2 n \rceil$

Another view: A link decreases the potential by 1.
This can pay for handling all the trees involved in the link.
The only "unaccounted" trees are those that
were not the input nor the output of a link operation.

Lazy Binomial Heaps

	Actual cost	Change in potential	Amortized cost
Insert	O(1)	1	O(1)
Find-min	O(1)	0	O(1)
Delete-min	$O(k+T_0+\log n)$	$k-1+T_1-T_0$	$O(\log n)$
Decrease-key	$O(\log n)$	0	$O(\log n)$
Meld	O(1)	0	O(1)

Rank of deleted root

Heaps / Priority queues

	Binary Heaps	Binomial Heaps	Lazy Binomial Heaps	Fibonacci Heaps
Insert	O(log n)	O(log n)	O(1)	O(1)
Find-min	O(1)	O(1)	O(1)	O(1)
Delete-min	O(log n)	O(log n)	O(log n)	O(log n)
Decrease-key	O(log n)	O(log n)	O(log n)	O(1)
Meld	–	O(log n)	O(1)	O(1)

Worst case

Amortized

One-pass successive linking

A tree produced by a link is immediately put in the output list and not linked again

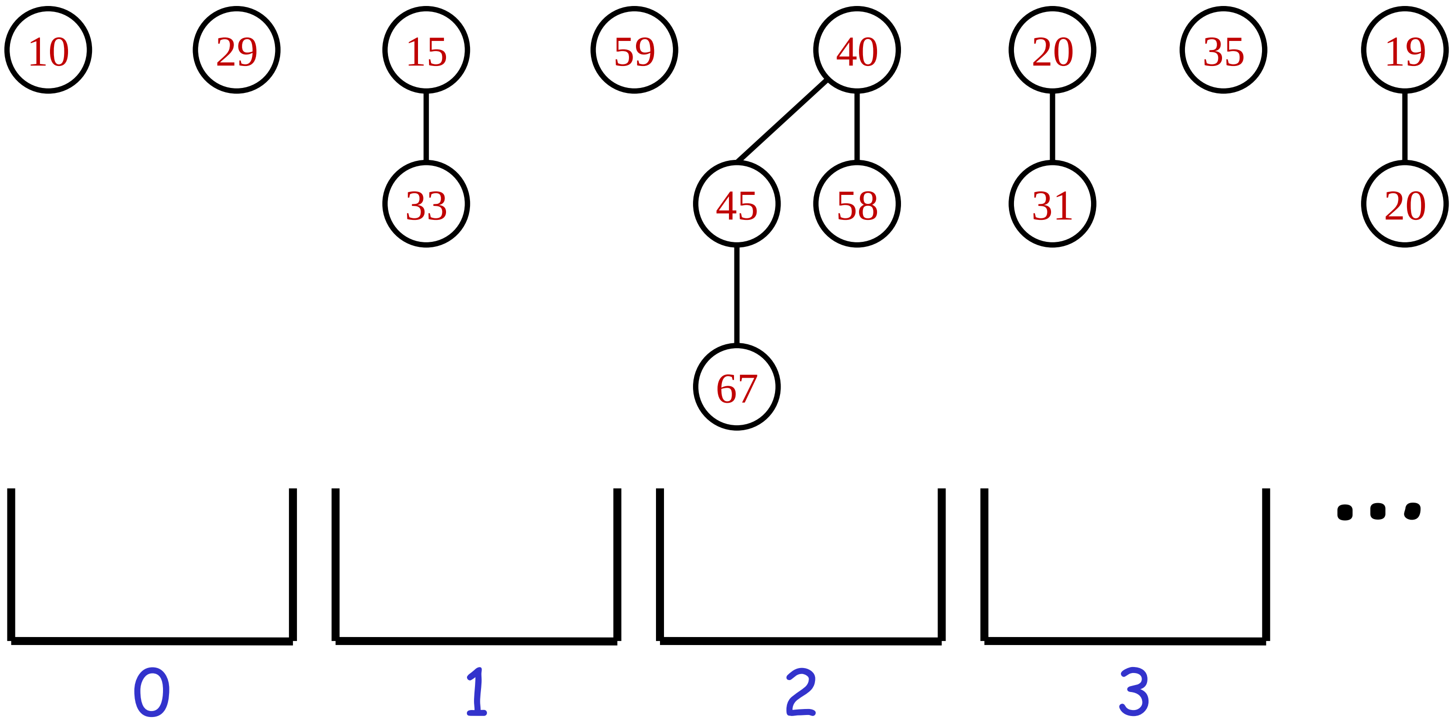
Worst case cost – $O(n)$

Amortized cost – $O(\log n)$

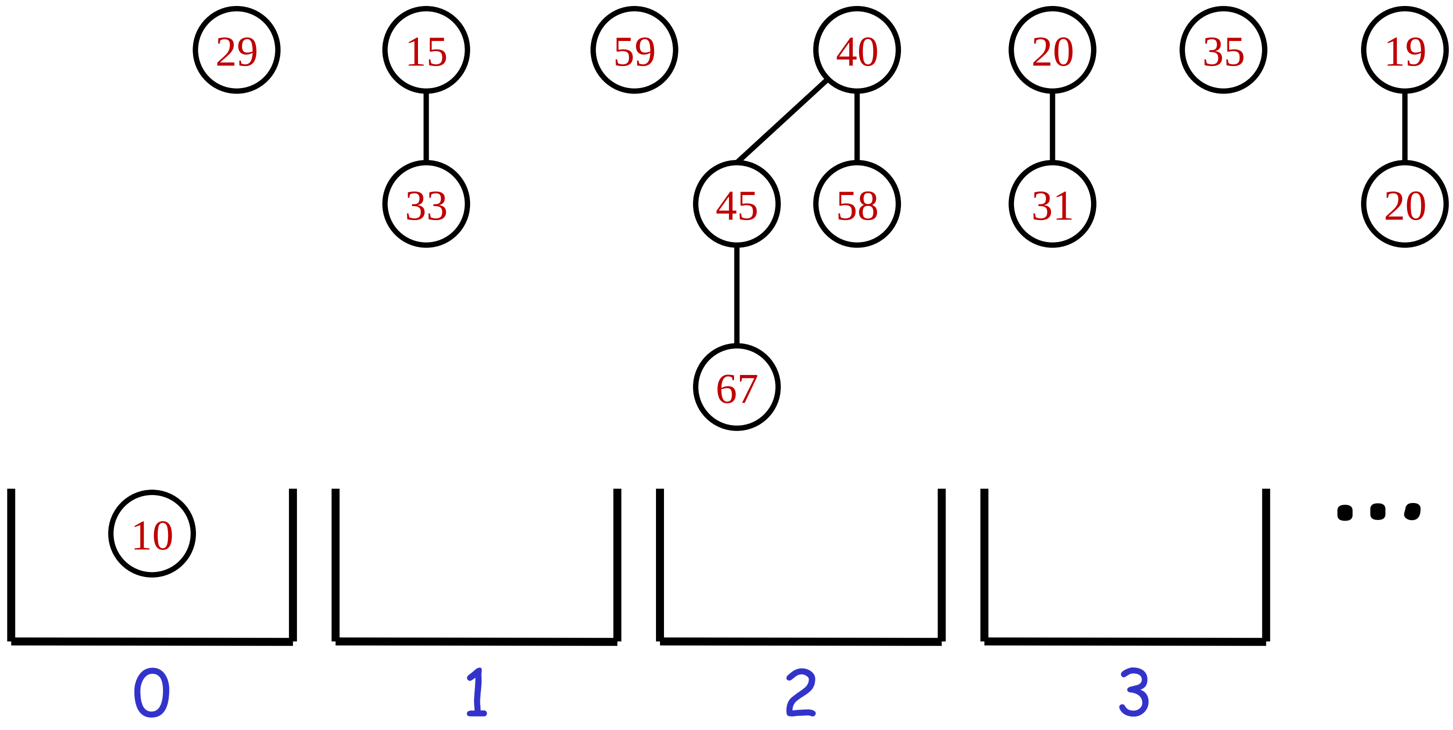
Potential = Number of Trees

Exercise: Prove it!

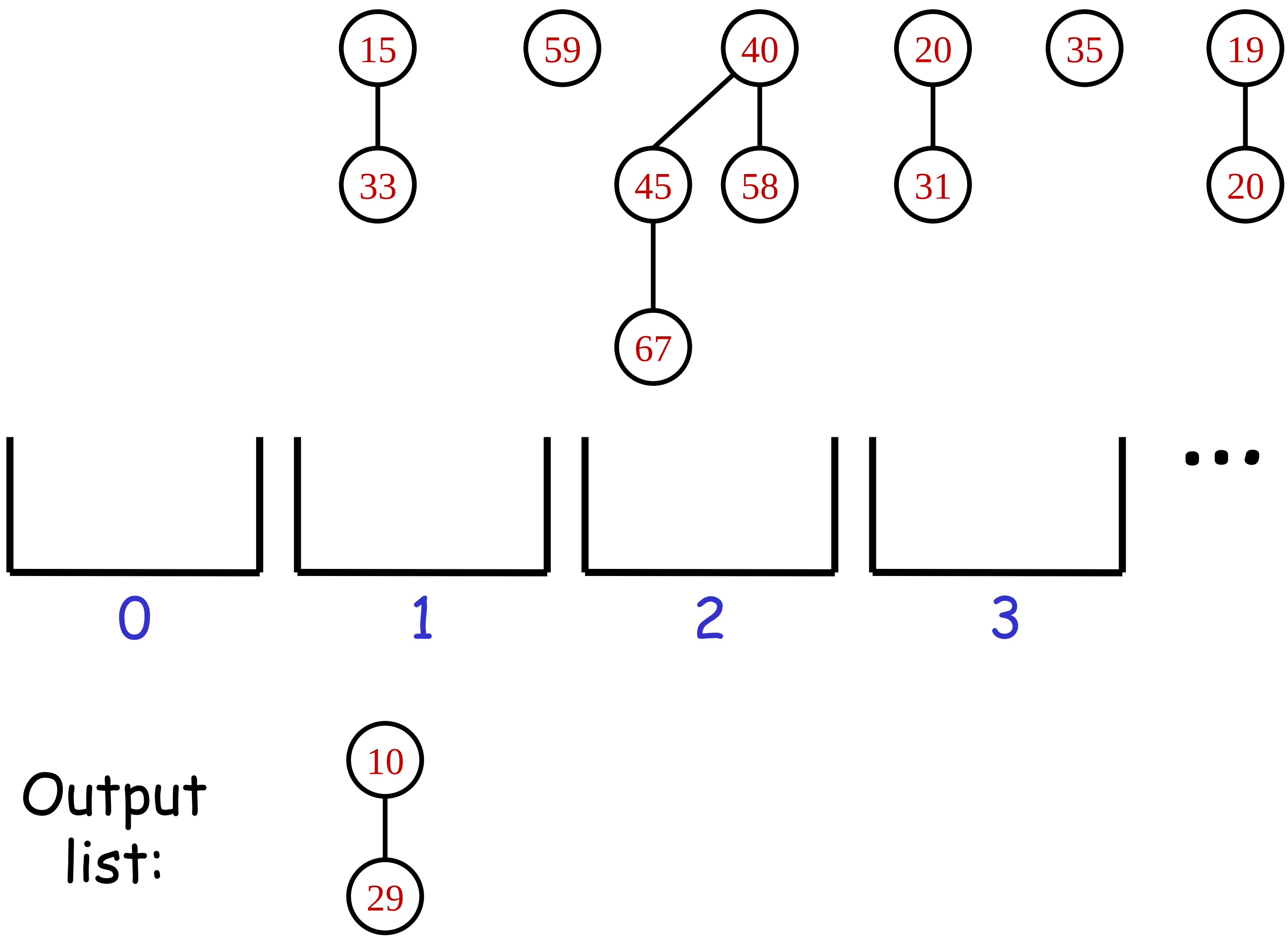
One-pass successive Linking



One-pass successive Linking



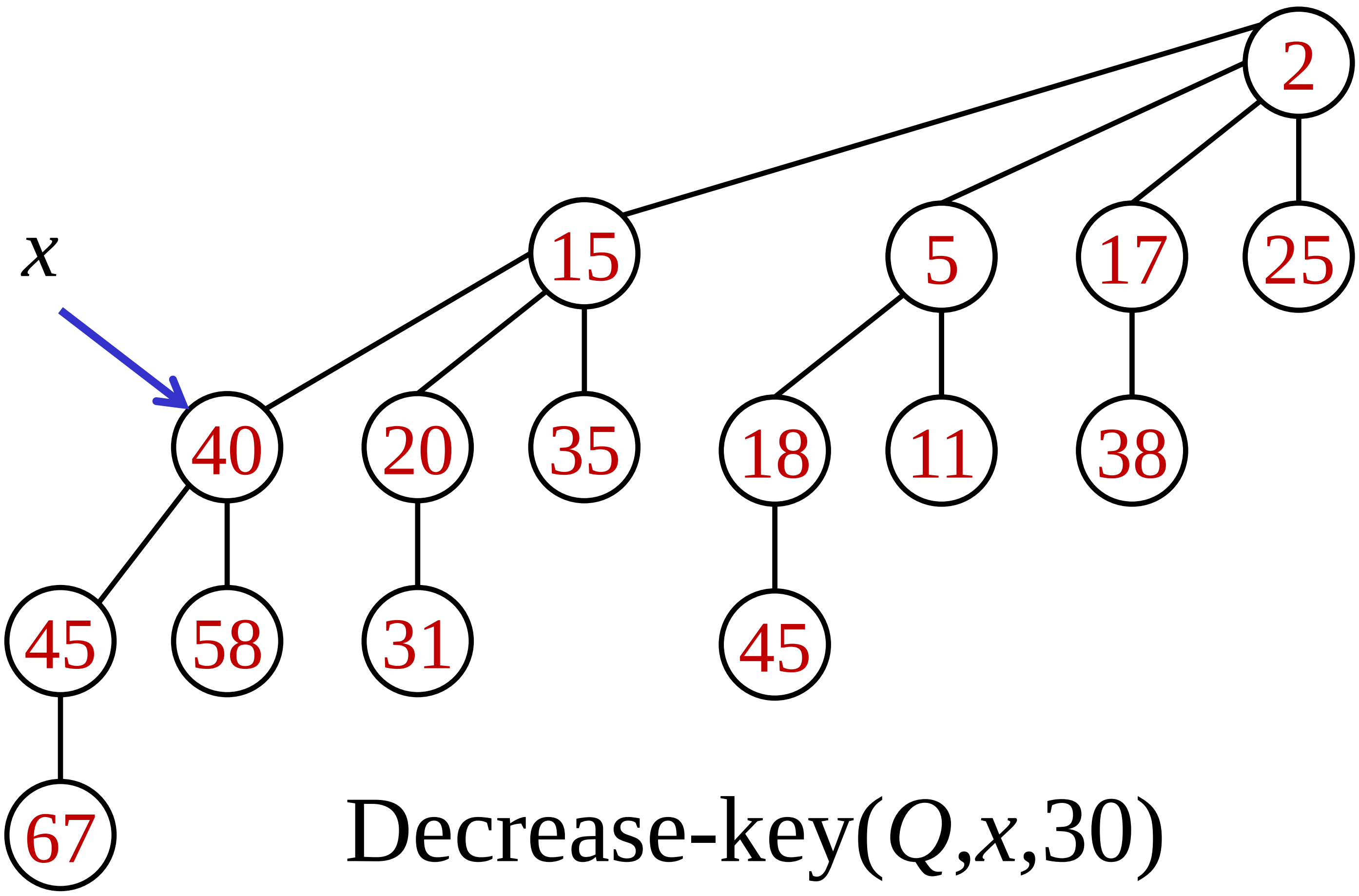
One-pass successive Linking



Fibonacci Heaps

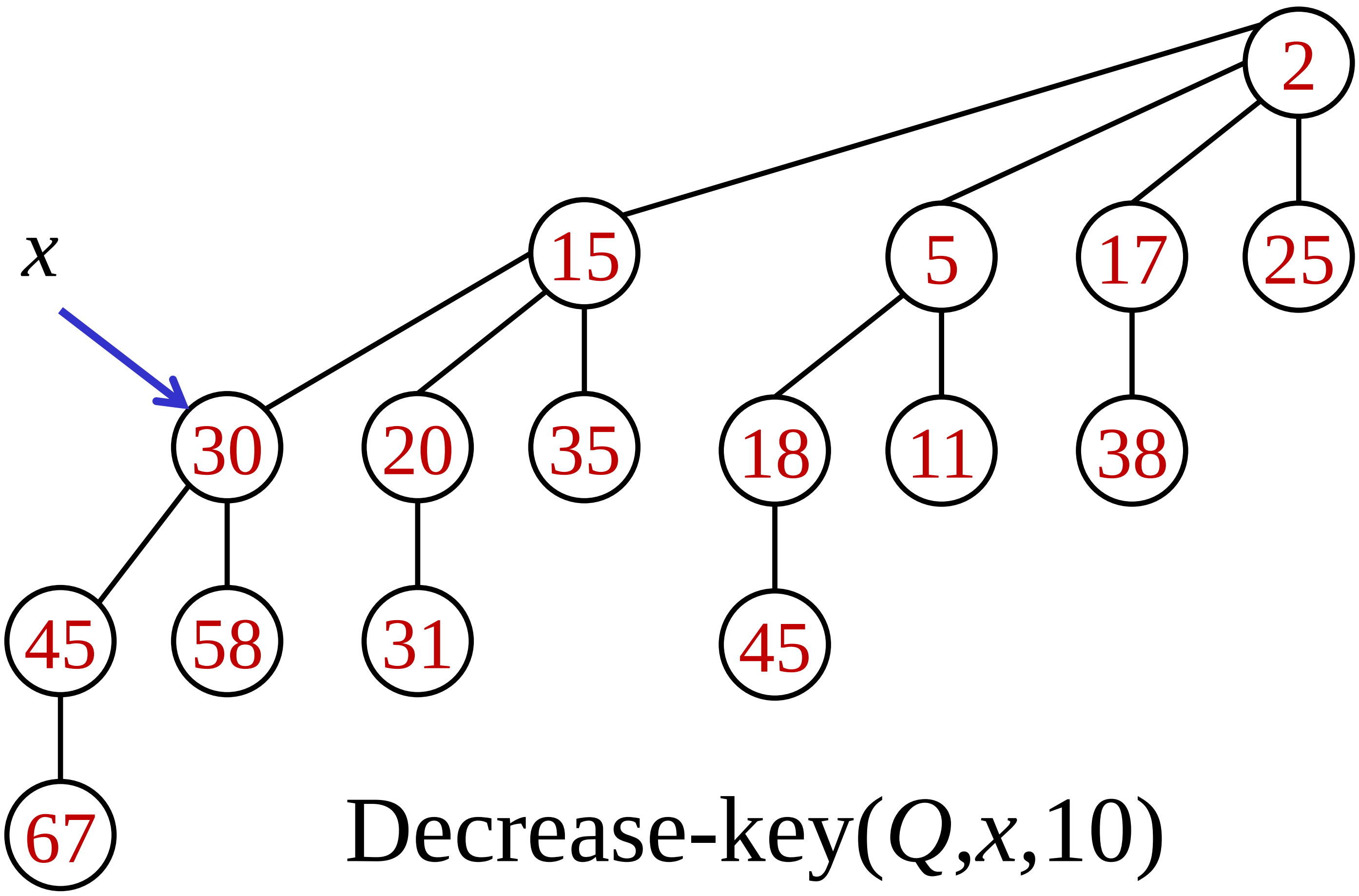
[Fredman-Tarjan (1987)]

Decrease-key in O(1) time?



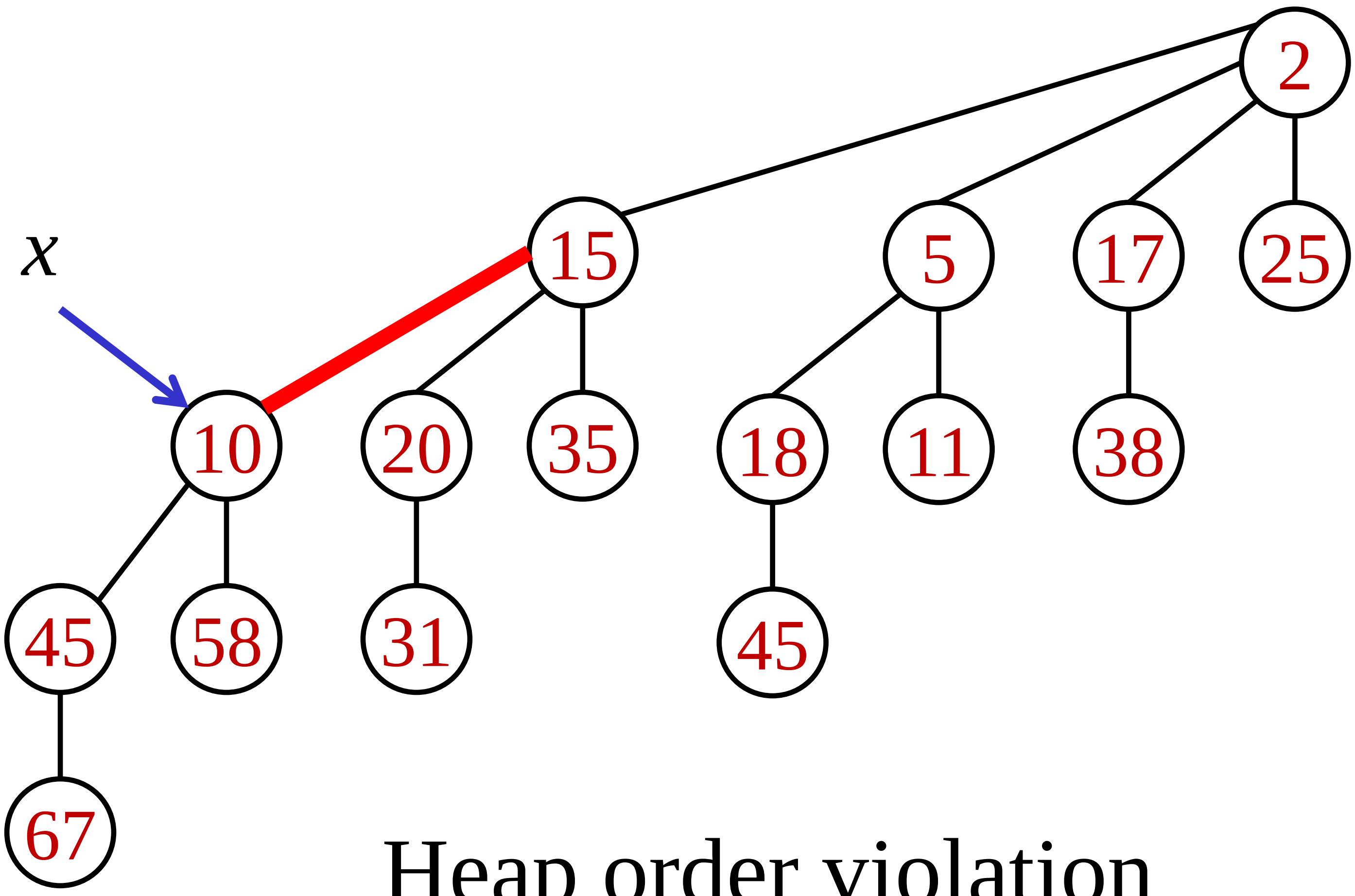
Decrease-key(Q,x,30)

Decrease-key in O(1) time?



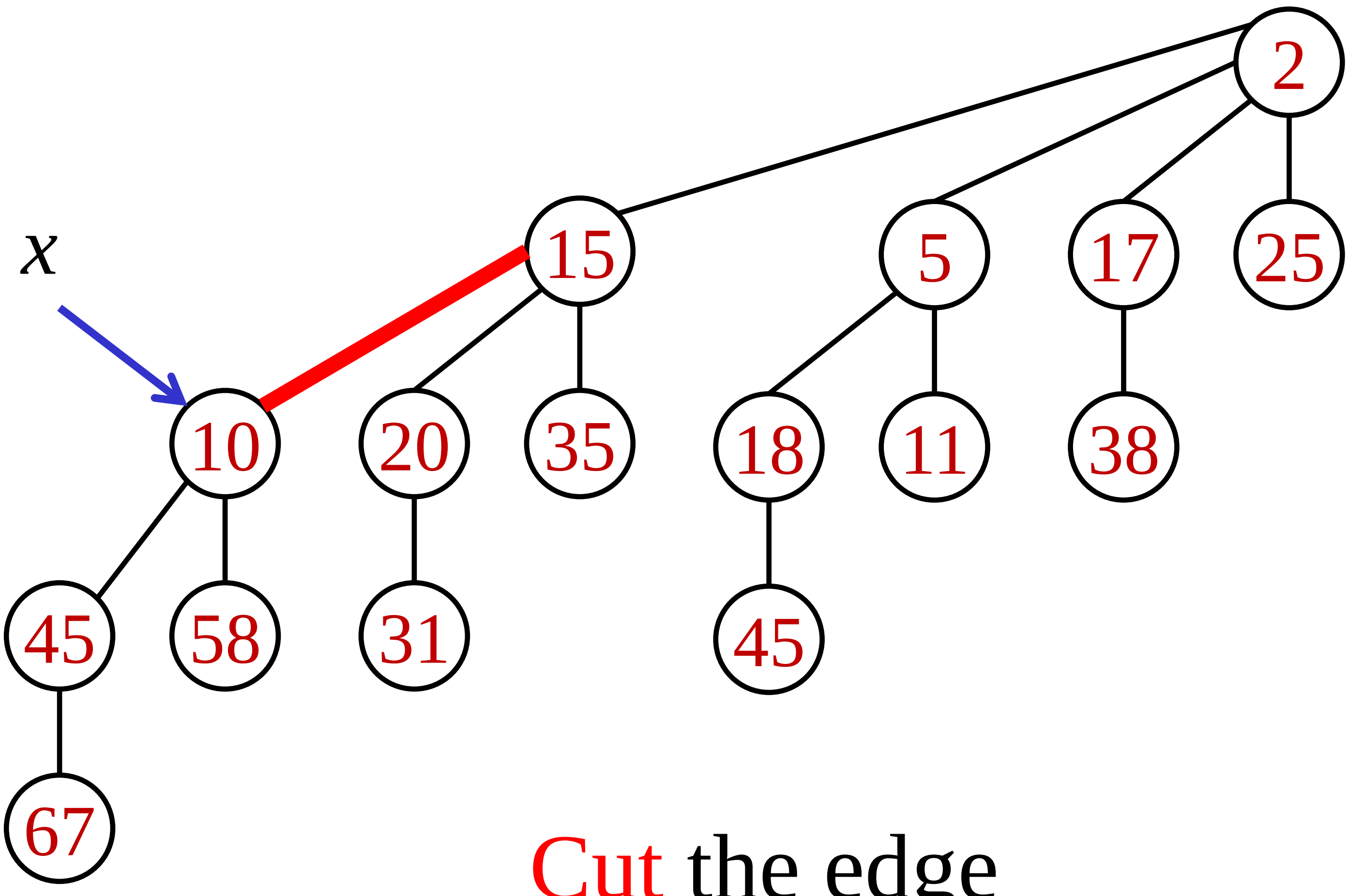
Decrease-key(Q,x,10)

Decrease-key in O(1) time?



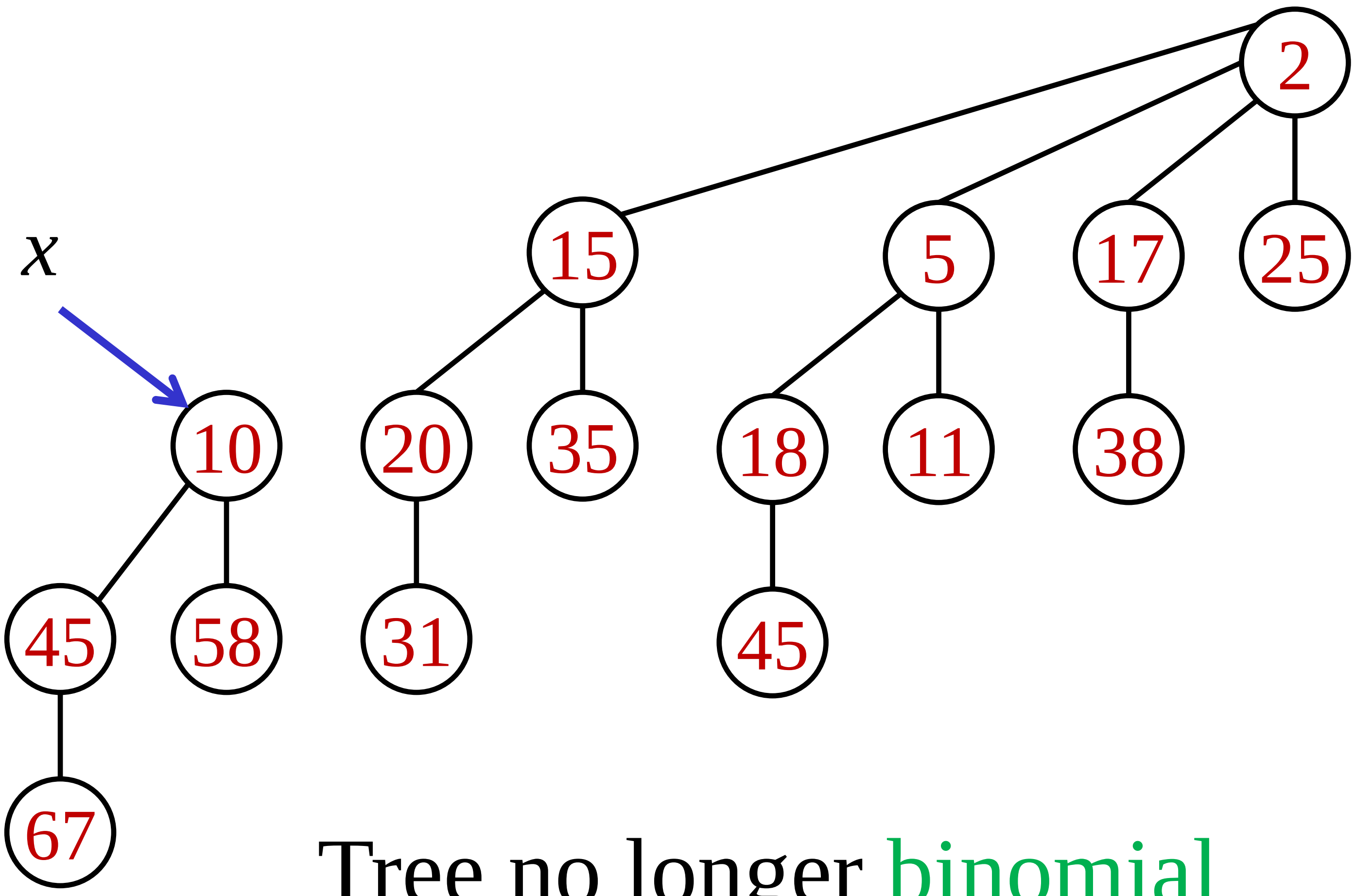
Heap order violation

Decrease-key in O(1) time?



Cut the edge

Decrease-key in O(1) time?

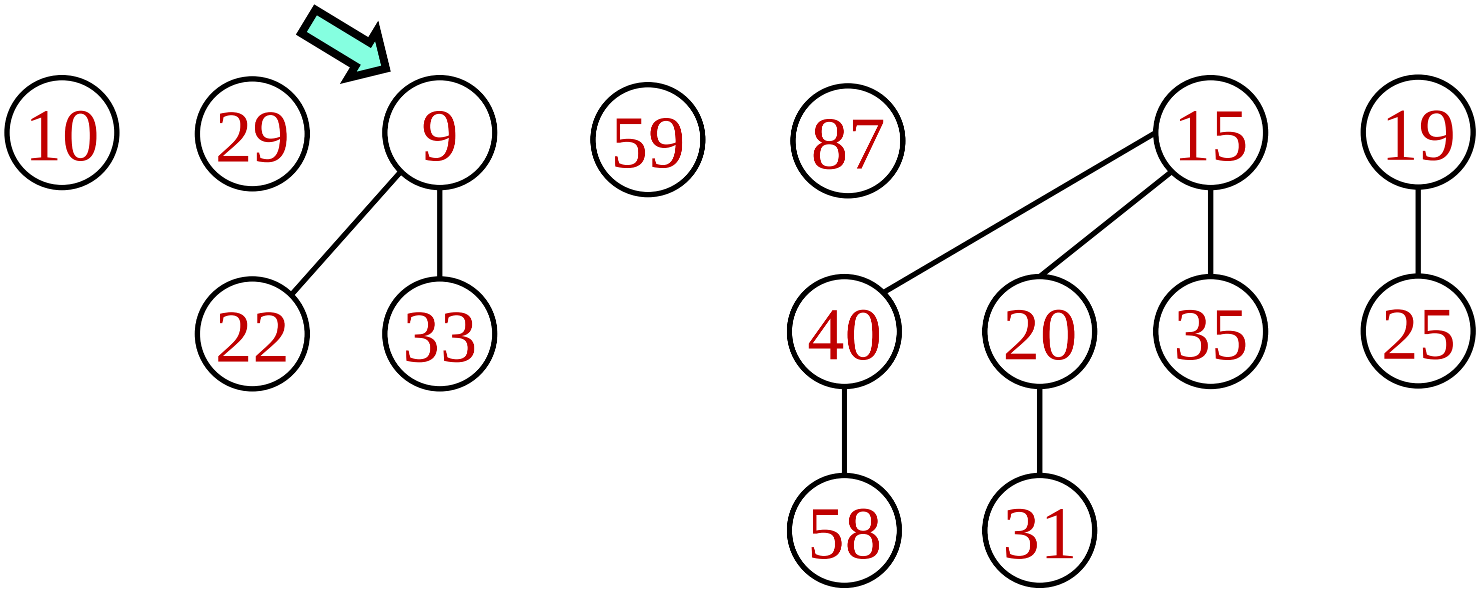


Tree no longer binomial

Fibonacci Heaps

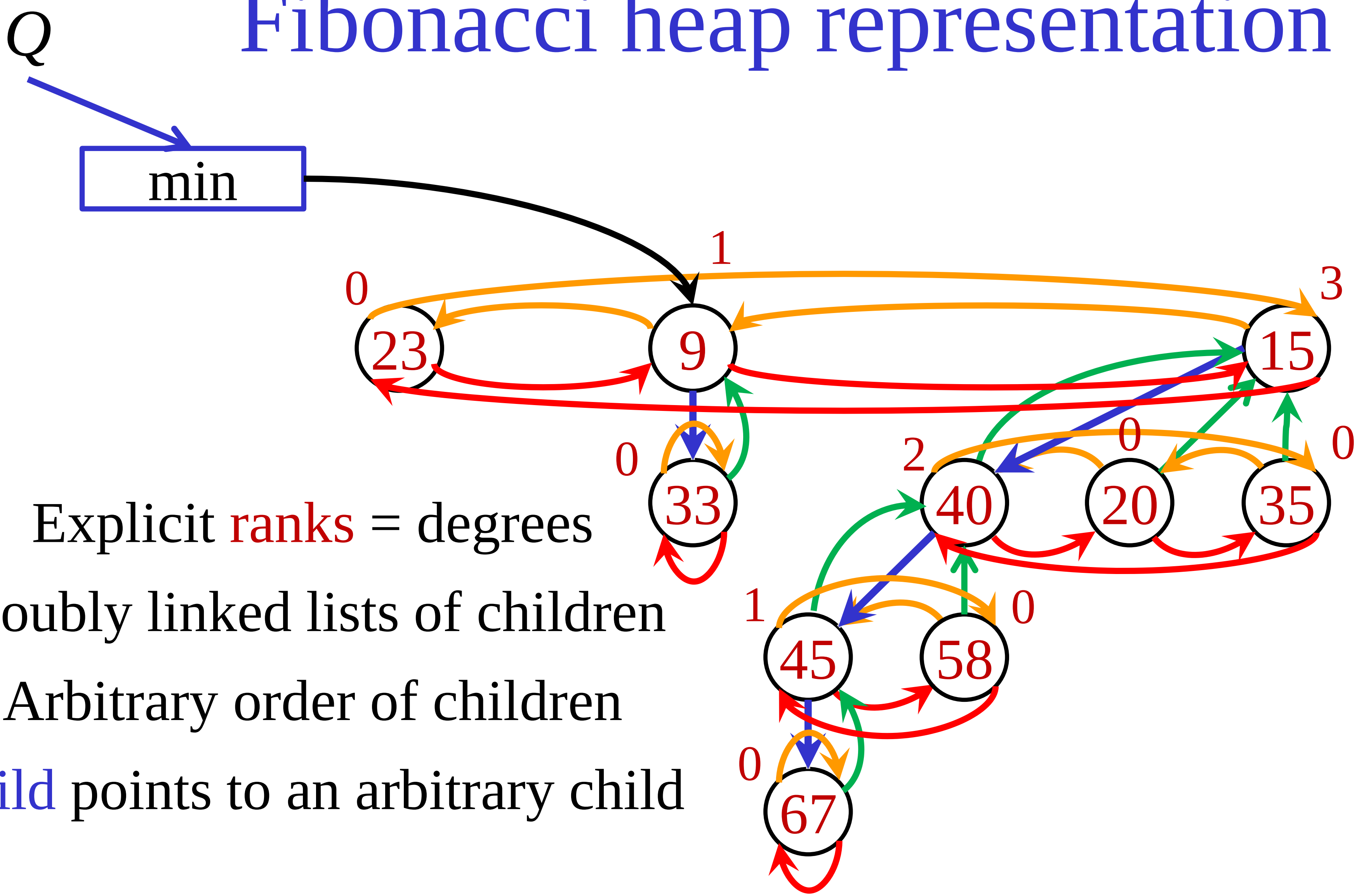
A list of heap-ordered trees

Pointer to root with minimal key



Not all trees may appear in Fibonacci heaps

Fibonacci heap representation



Explicit ranks = degrees

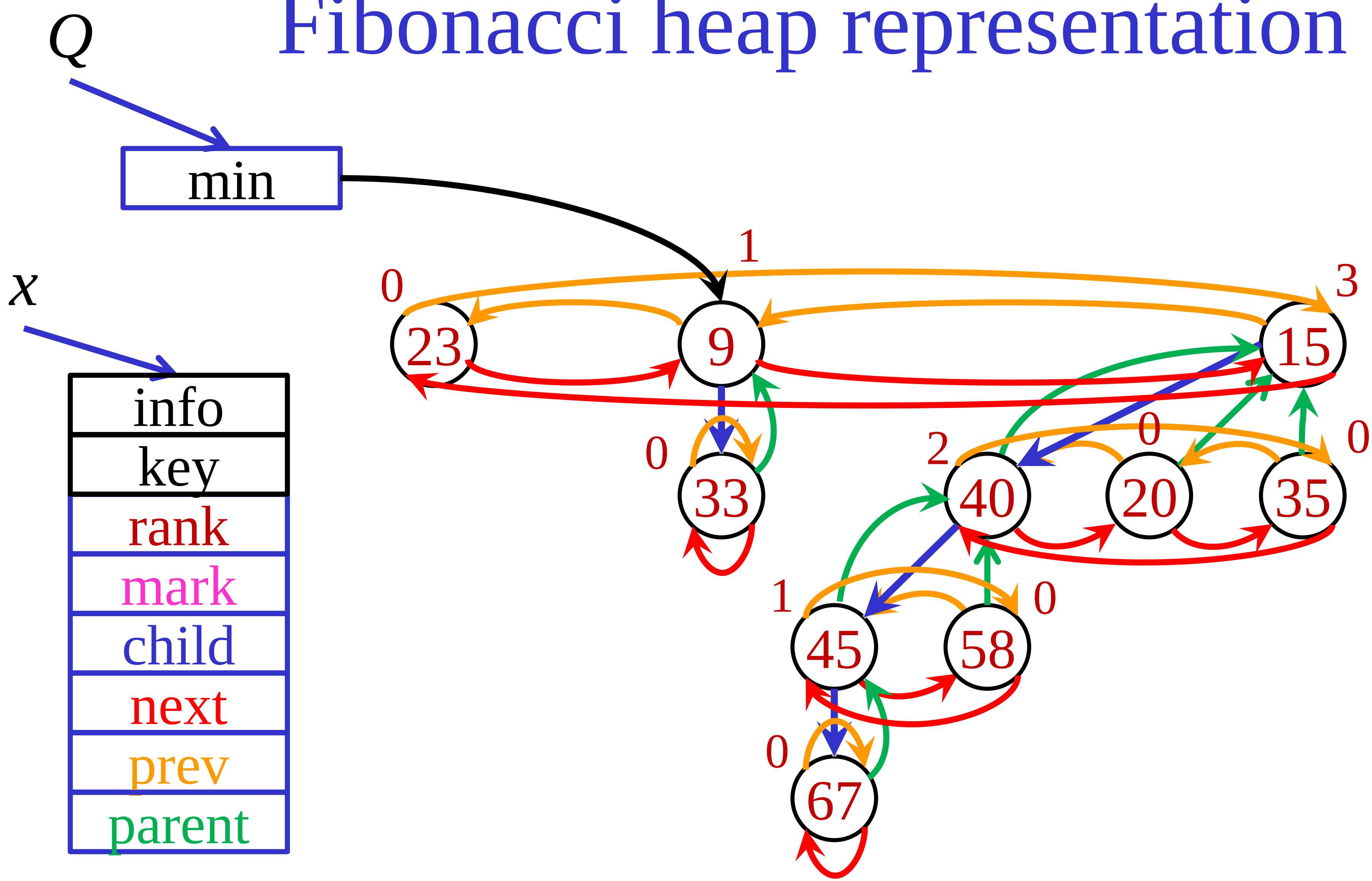
Doubly linked lists of children

Arbitrary order of children

child points to an arbitrary child

4 pointers + rank + mark bit per node

Fibonacci heap representation



4 pointers + rank + mark bit per node

Are simple cuts enough?

A binomial tree of rank k contains at least 2^k

We may get trees of rank k containing only $k+1$ nodes

Ranks not necessarily $O(\log n)$

Analysis breaks down

Cascading cuts

Invariant: Each node looses at most one child after becoming a child itself

To maintain the invariant, use a mark bit

Each node is initially unmarked.

When a non-root node looses its first child, it becomes marked

When a marked node looses a second child, it is cut from its parent

Cascading cuts

Invariant: Each node looses at most one child after becoming a child itself

When $x \rightarrow y$ is cut:

x becomes unmarked

If y is unmarked, it becomes marked

If y is marked, it is cut from its parent

Our convention: Roots are unmarked

Cascading cuts

Function `cut`(x, y)

$x.parent \leftarrow null$
$x.mark \leftarrow 0$
$y.rank \leftarrow y.rank - 1$
if $x.next = x$ **then**
  $y.child \leftarrow null$
else
  $y.child \leftarrow x.next$
  $x.prev.next \leftarrow x.next$
  $x.next.prev \leftarrow x.prev$

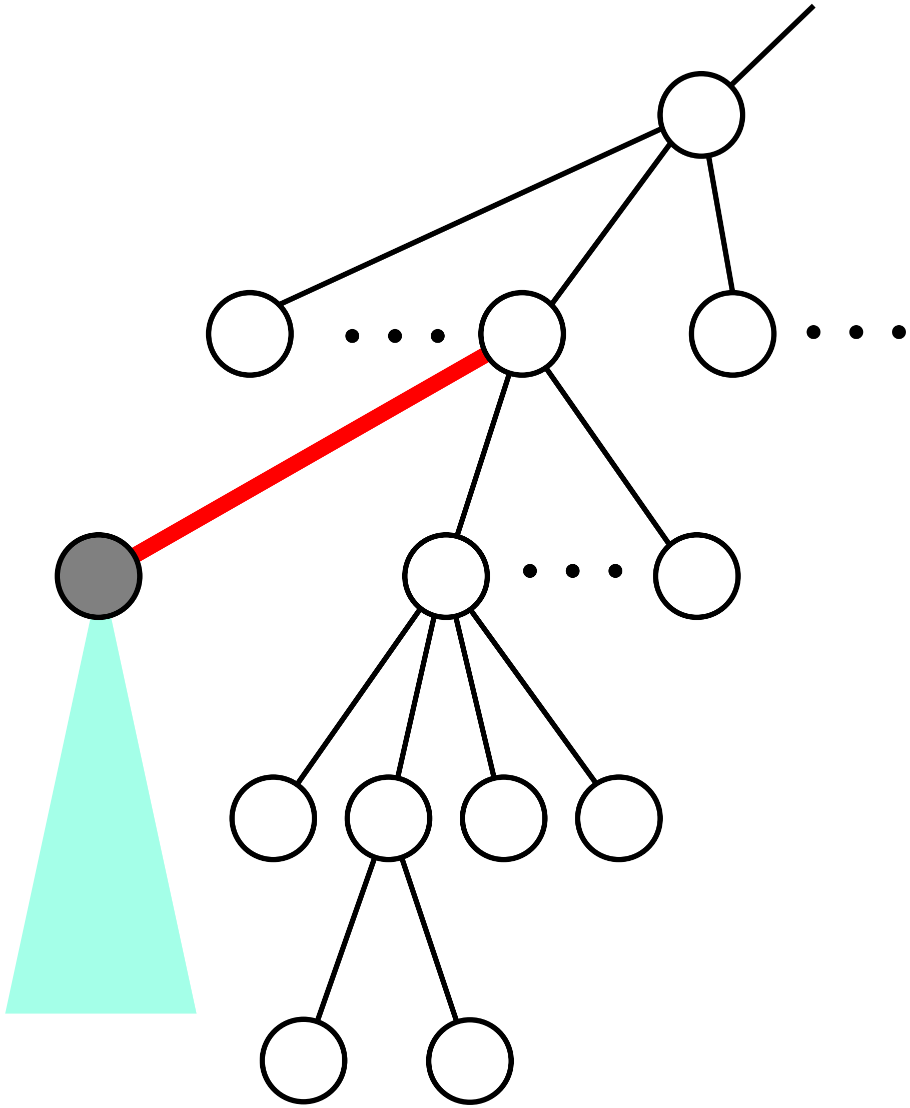
Cut x from its parent y

Function `cascading-cut`(x, y)

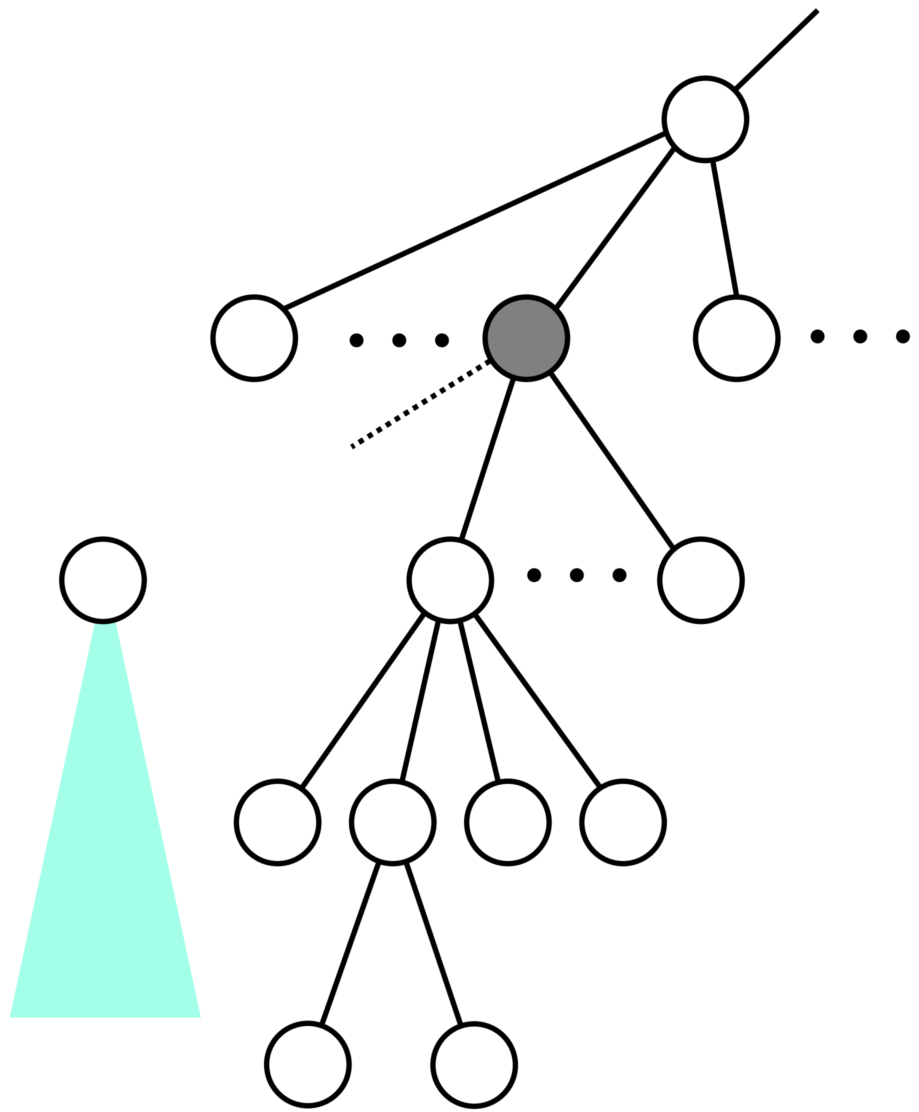
`cut`(x, y)
if $y.parent \neq null$ **then**
  if $y.mark = 0$ **then**
    $y.mark \leftarrow 1$
  else
    `cascading-cut`$(y, y.parent)$

Perform a cascading-cut process starting at x

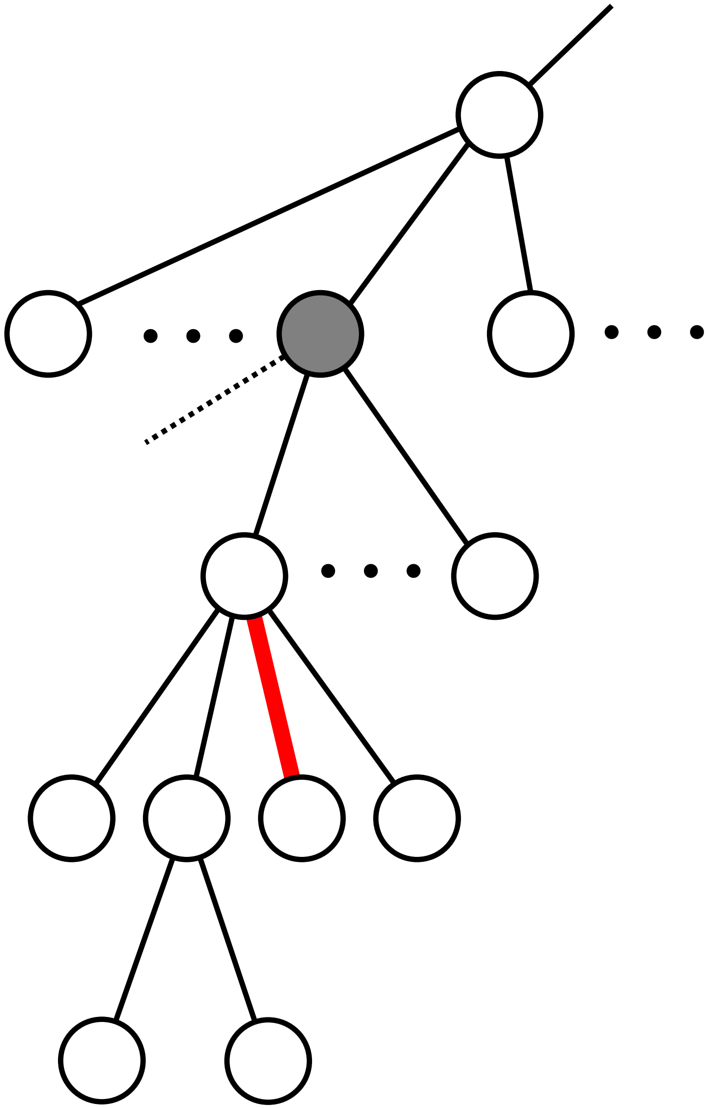
Cascading cuts



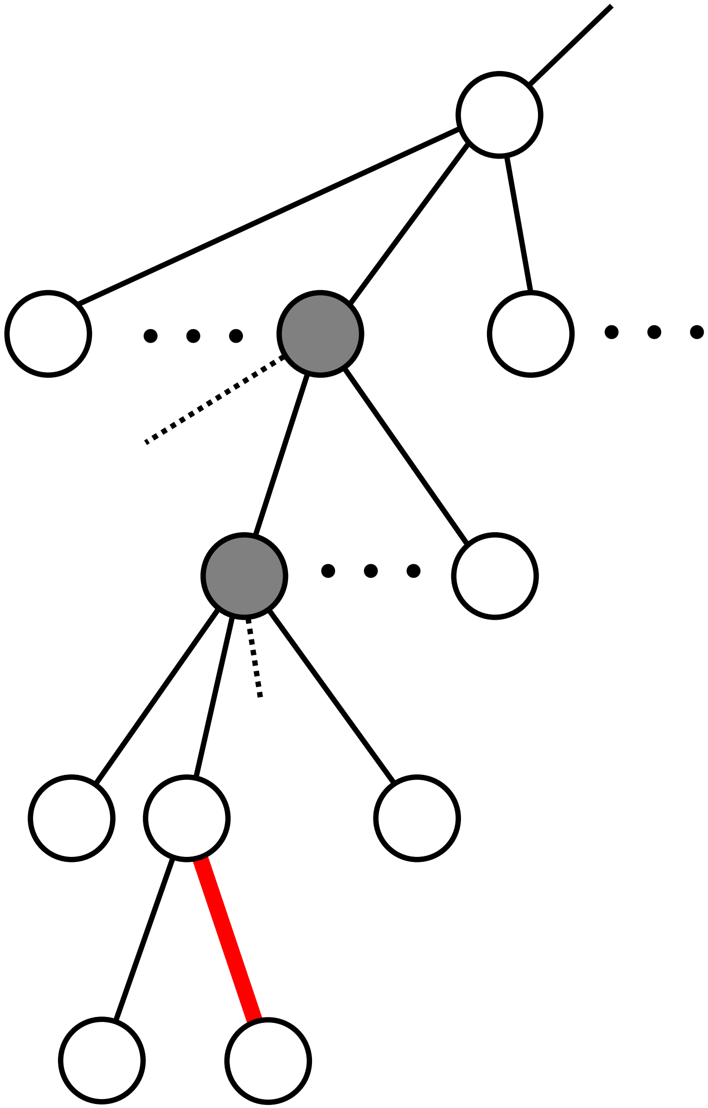
Cascading cuts



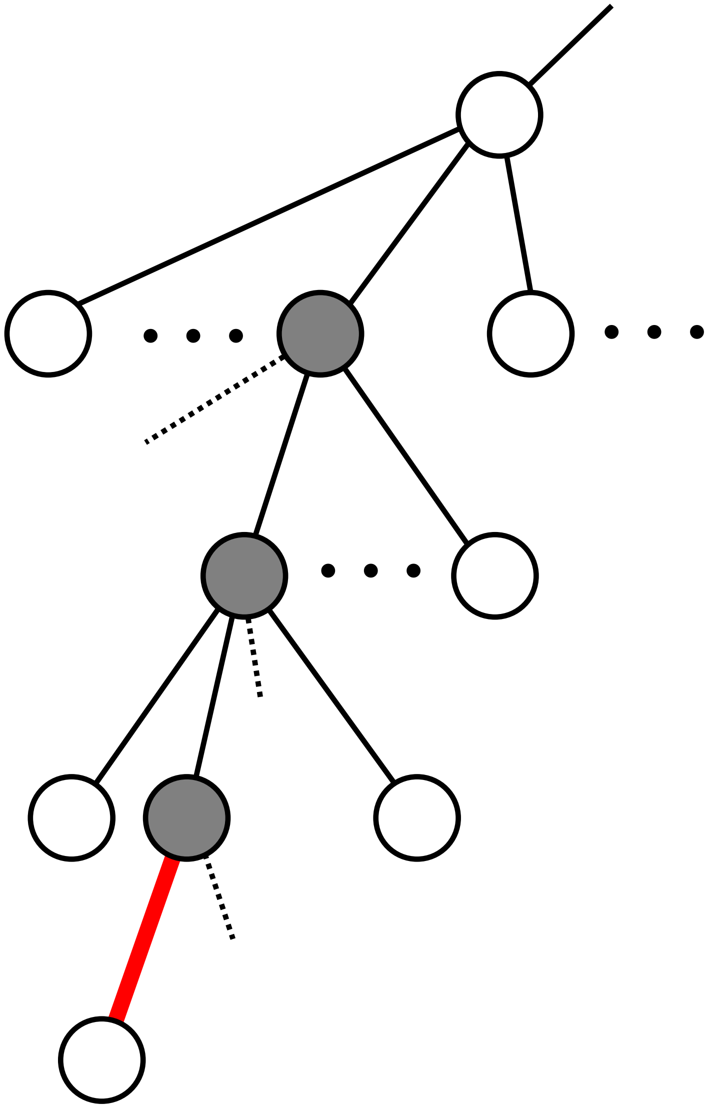
Cascading cuts



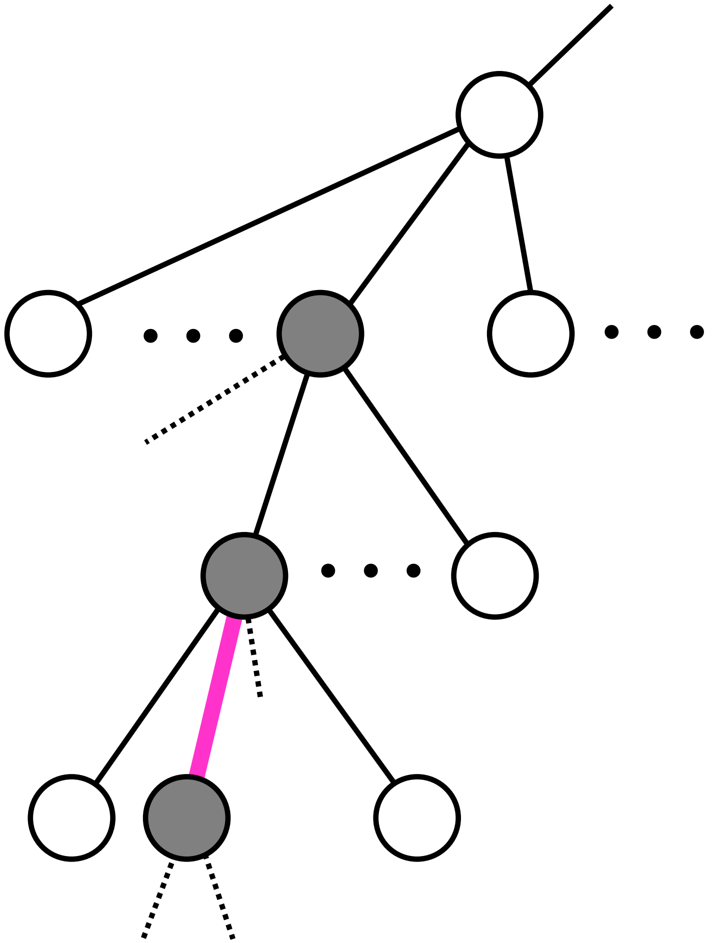
Cascading cuts



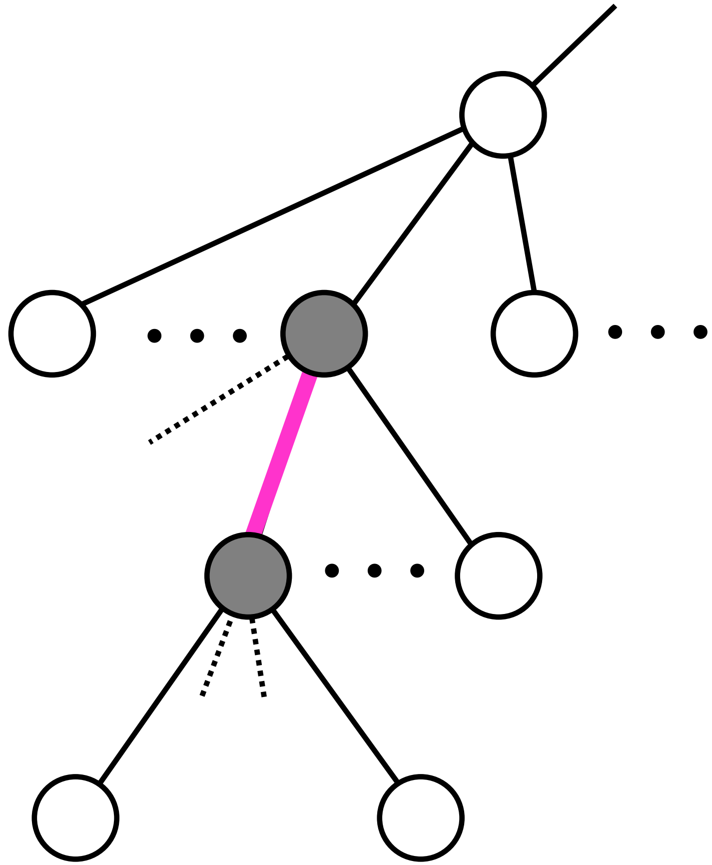
Cascading cuts



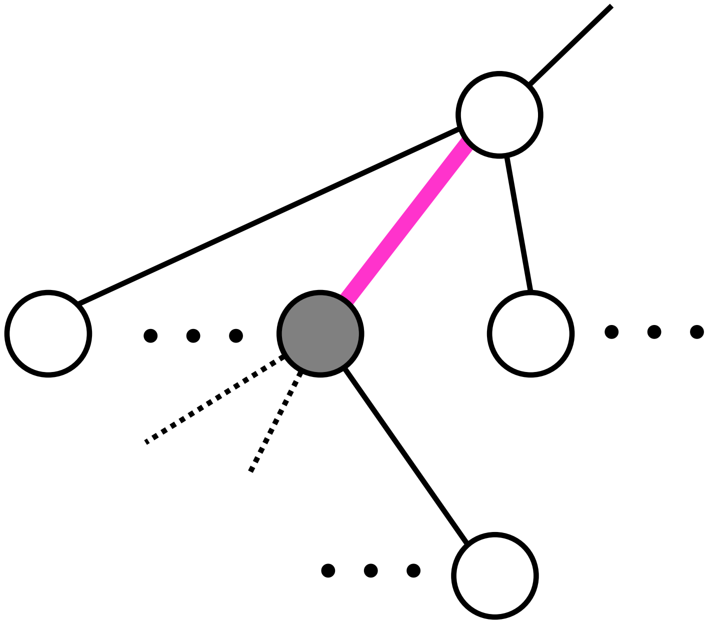
Cascading cuts



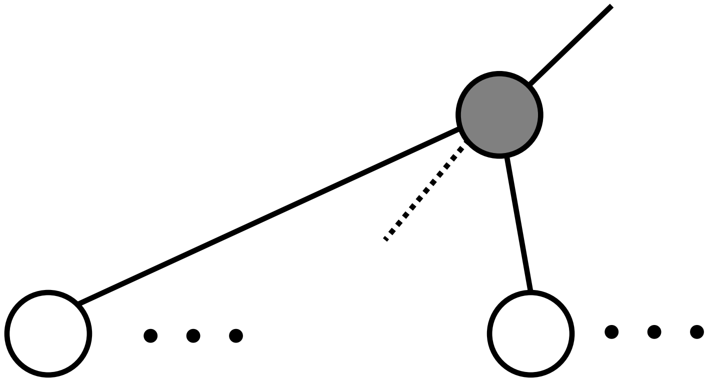
Cascading cuts



Cascading cuts



Cascading cuts



Number of cuts

A decrease-key operation may trigger many cuts

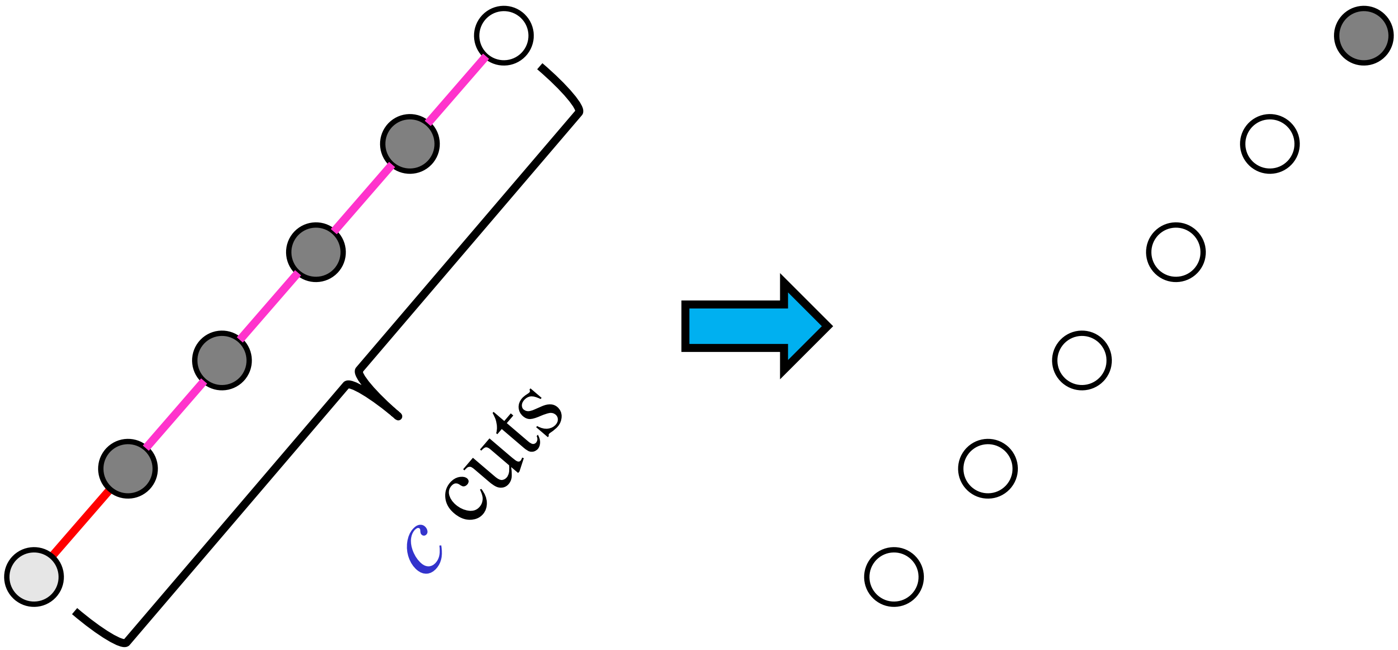
Lemma 1: The first d decrease-key operations trigger at most $2d$ cuts

Proof in a nutshell:
Number of times a second child is lost is at most the number of times a first child is lost

Potential = Number of marked nodes

Number of cuts

Potential = Number of marked nodes



Amortized number of cuts $\leq\ c + (1-(c-1))\ =\ 2$

Trees formed by cascading cuts

Lemma 2: Let x be a node of rank k and let $y_1, y_2, \ldots, y_k$ be the current children of x, in the order in which they were linked to x.
Then, the rank of y_i is at least $i–2$.

Proof: When y_i was linked to x, $y_1, \ldots y_{i–1}$ were already children of x.
At that time, the rank of x and y_i was at least $i–1$.
As y_i is still a child of x, it lost at most one child.

Trees formed by cascading cuts

Lemma 3: A node of rank k in a Fibonacci Heap has at least $F_{k+2} \geq \phi^k$ descendants, including itself.

$$F_0 = 0 \quad F_1 = 1$$
$$F_k = F_{k-1} + F_{k-2} \ , \ k \geq 2$$

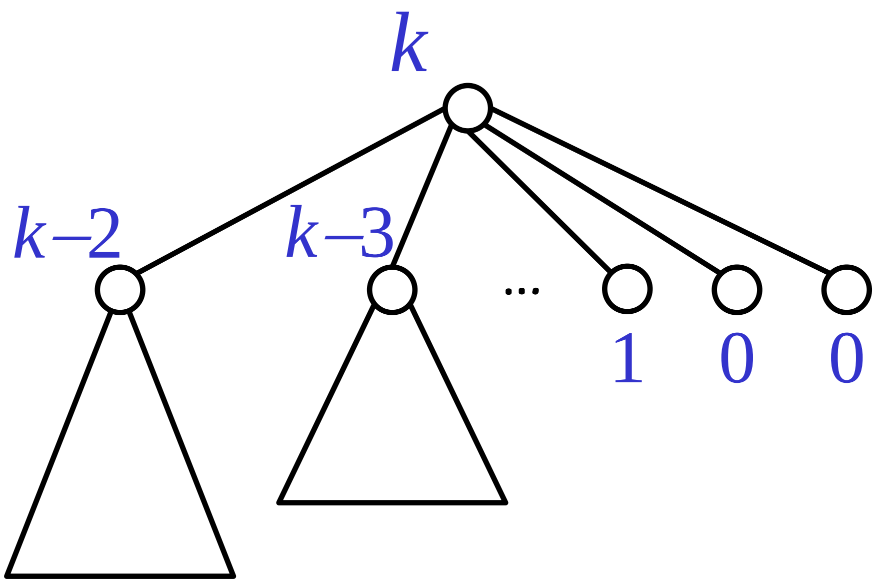
$$\phi = \tfrac{1+\sqrt{5}}{2} \simeq 1.618$$

$$F_{k+2} \ = \ 2 + \sum_{i=2}^{k} F_i \ , \ k \geq 2$$

n	0	1	2	3	4	5	6	7	8	9
F_n	0	1	1	2	3	5	8	13	21	34

Trees formed by cascading cuts

Lemma 3: A node of rank k in a Fibonacci Heap has at least $F_{k+2} \geq \phi^k$ descendants, including itself.



Let S_k be the minimum number of descendants of a node of rank at least k

$$S_0 = 1 \quad S_1 = 2$$

$$S_k \;\geq\; 2 + \sum_{i=0}^{k-2} S_i \;, \; k \geq 2$$

$$S_k \;\geq\; 2 + \sum_{i=0}^{k-2} S_i \;\geq\; 2 + \sum_{i=0}^{k-2} F_{i+2} \;=\; 2 + \sum_{i=2}^{k} F_i \;=\; F_{k+2}$$

Trees formed by cascading cuts

Lemma 3: A node of rank k in a Fibonacci Heap has at least $F_{k+2} \geq \phi^k$ descendants, including itself.

Corollary: In a Fibonacci heap containing n items, all ranks are at most $\log_\phi n \leq 1.4404 \log_2 n$

Ranks are again $O(\log n)$

Are we done?

Putting it all together

Are we done?

A cut increases the number of trees…

We need a potential function that gives good amortized bounds on both successive linking and cascading cuts

Potential = #trees + 2 #marked

Cost of Consolidating

T_0 – Number of trees before

T_1 – Number of trees after

L – Number of links

$T_1 = T_0 - L$ (Each link reduces the number of tree by 1)

Total number of trees processed – $T_0 + L$
(Each link creates a new tree)

Putting trees into buckets or finding trees to link with

Linking

Handling the buckets

$$\text{Total cost} = O(\,(T_0 + L) + L + \lceil \log_\phi n \rceil\,)$$

$$= O(\,T_0 + \lceil \log_\phi n \rceil\,)$$

As $L \le T_0$

Cost of Consolidating

T_0 – Number of trees before

T_1 – Number of trees after

L – Number of links

$T_1 = T_0 - L$ (Each link reduces the number of tree by 1)

Total number of trees processed – $T_0 + L$
(Each link creates a new tree)

Only change:
$\log_\phi n$ *instead of* $\log_2 n$

Total cost $= O(\,(T_0+L) + L + \lceil \log_\phi n \rceil\,)$

$= O(\,T_0 + \lceil \log_\phi n \rceil\,)$

As $L \leq T_0$

Fibonacci heaps

	Actual cost	Δ Trees	Δ Marks	Amortized cost
Insert	O(1)	1	0	O(1)
Find-min	O(1)	0	0	O(1)
Delete-min	$O(k+T_0+\log n)$	$k-1+T_1-T_0$	≤ 0	$O(\log n)$
Decrease-key	$O(c)$	c	$\leq 2-c$	O(1)
Meld	O(1)	0	0	O(1)

Rank of deleted root

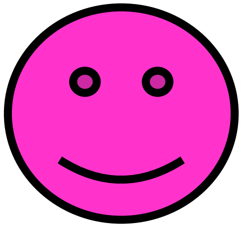
Number of cuts performed

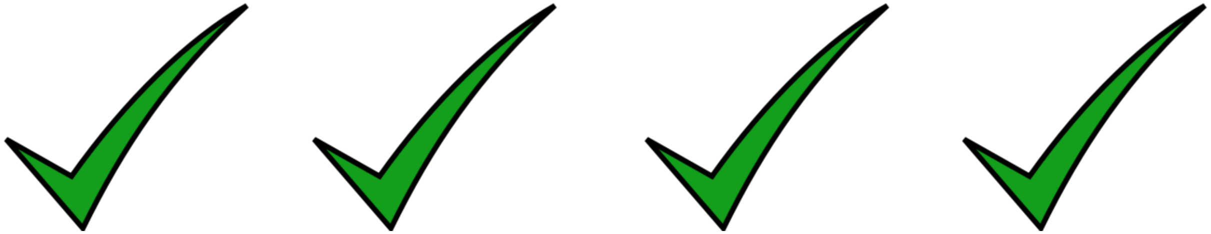
Heaps / Priority queues

	Binary Heaps	**Binomial Heaps**	**Lazy Binomial Heaps**	**Fibonacci Heaps**
Insert	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
Find-min	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Delete-min	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Decrease-key	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Meld	–	$O(\log n)$	$O(1)$	$O(1)$

Worst case

Amortized





Consolidating / Successive linking

Function `consolidate`(x)

- `to-buckets`(x)
- **return** `from-buckets`()

Function `to-buckets`(x)

- **for** $i \leftarrow 0$ `to` $\log_\phi n$ **do**
  - $B[i] \leftarrow \mathit{null}$
- $x.\mathit{prev}.\mathit{next} \leftarrow \mathit{null}$
- **while** $x \neq \mathit{null}$ **do**
  - $y \leftarrow x$
  - $x \leftarrow x.\mathit{next}$
  - **while** $B[y.\mathit{rank}] \neq \mathit{null}$ **do**
    - $y \leftarrow$ `link`$(y, B[y.\mathit{rank}])$
    - $B[y.\mathit{rank} - 1] \geq \mathit{null}$
  - $B[y.\mathit{rank}] \leftarrow y$

Function `from-buckets`()

- $x \leftarrow \mathit{null}$
- **for** $i \leftarrow 0$ `to` $\log_\phi n$ **do**
  - **if** $B[i] \neq \mathit{null}$ **then**
    - **if** $x = \mathit{null}$ **then**
      - $x \leftarrow B[i]$
      - $x.\mathit{next} \leftarrow x$
      - $x.\mathit{prev} \leftarrow x$
    - **else**
      - `insert-after`$(x, B[i])$
      - **if** $B[i].\mathit{key} < x.\mathit{key}$ **then**
        - $x \leftarrow B[i]$
- **return** x