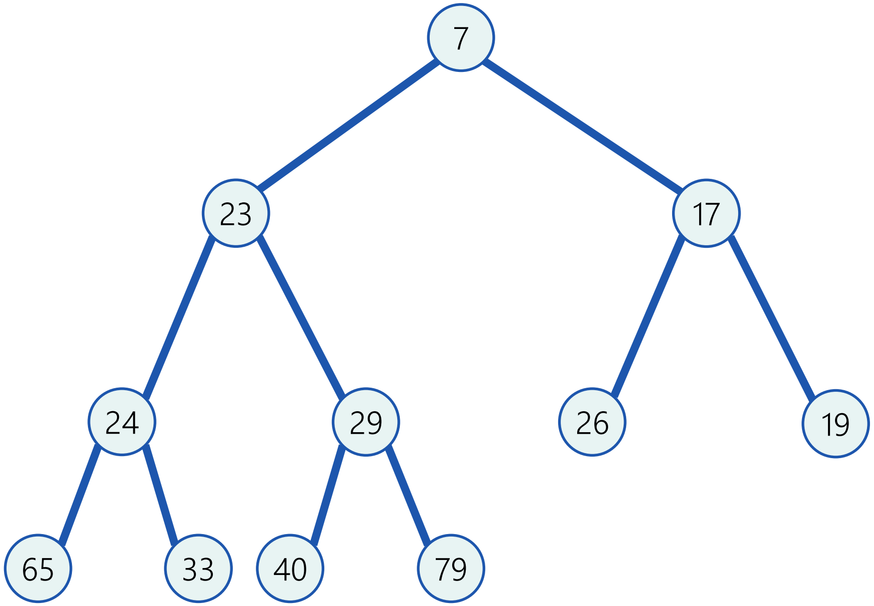
Data Structures

Lecture 8

Lazy Binomial Heaps
Fibonacci Heaps

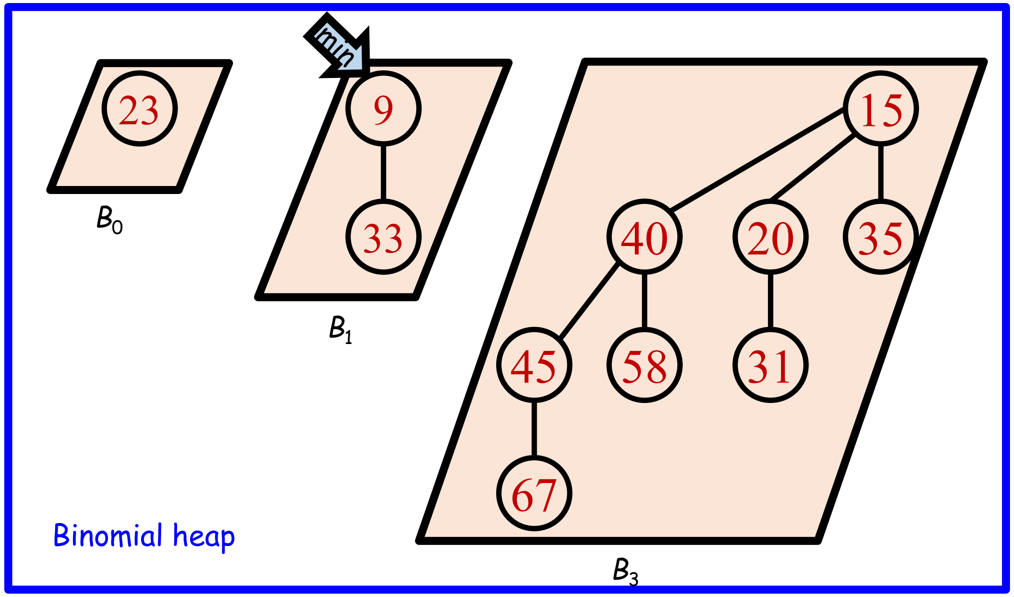
Shiri Chechik, Or Zamir
Winter semester 2025-6

Binary Heap



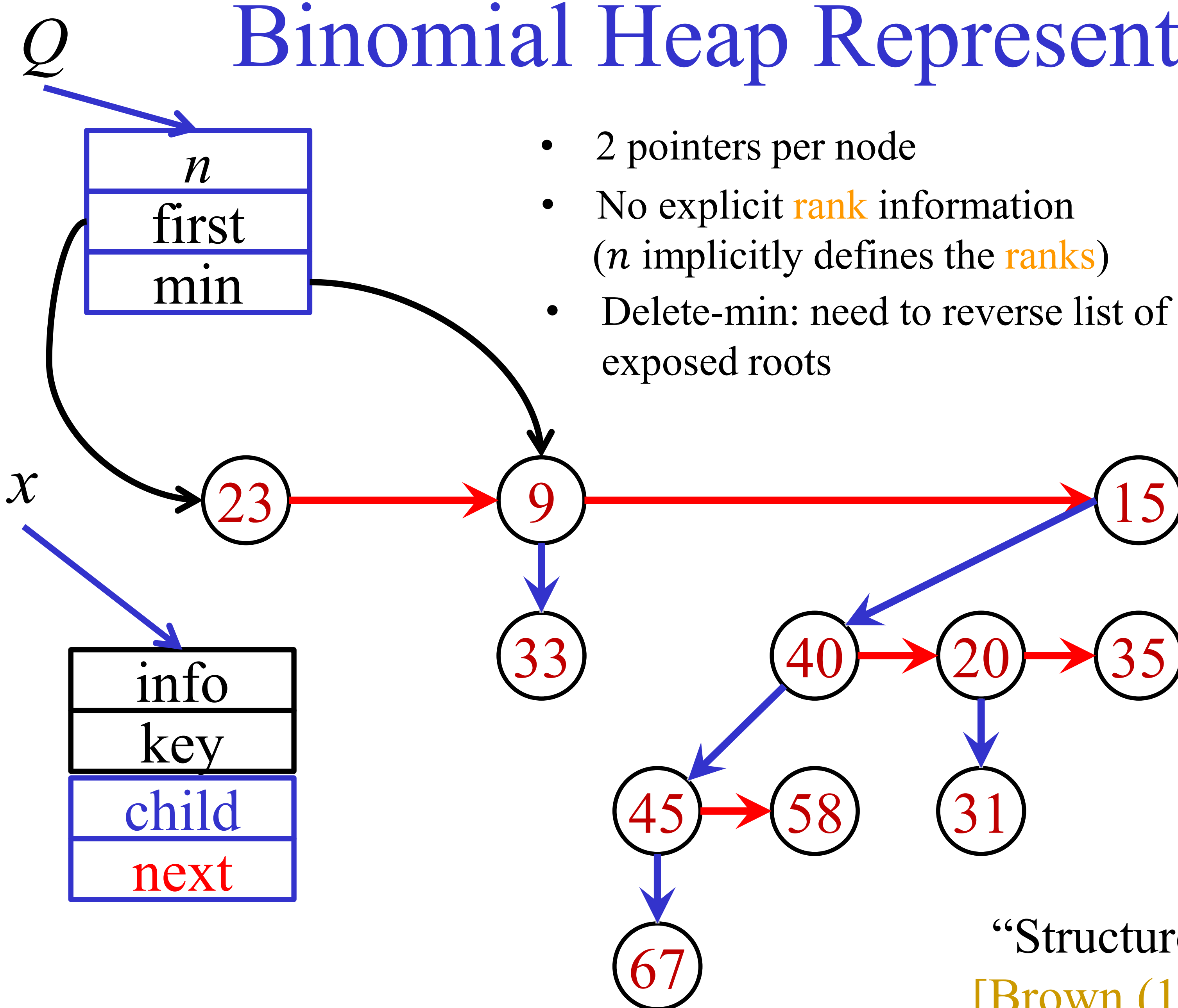
	Binary Heaps
Insert	$O(\log n)$
Find-min	$O(1)$
Delete-min	$O(\log n)$
Decrease-key	$O(\log n)$
Meld / Join	$O(n)$

Binomial Heap



	Binomial Heaps
Insert	$O(\log n)$
Find-min	$O(1)$
Delete-min	$O(\log n)$
Decrease-key	$O(\log n)$
Meld / Join	$O(\log n)$

Binomial Heap Representation

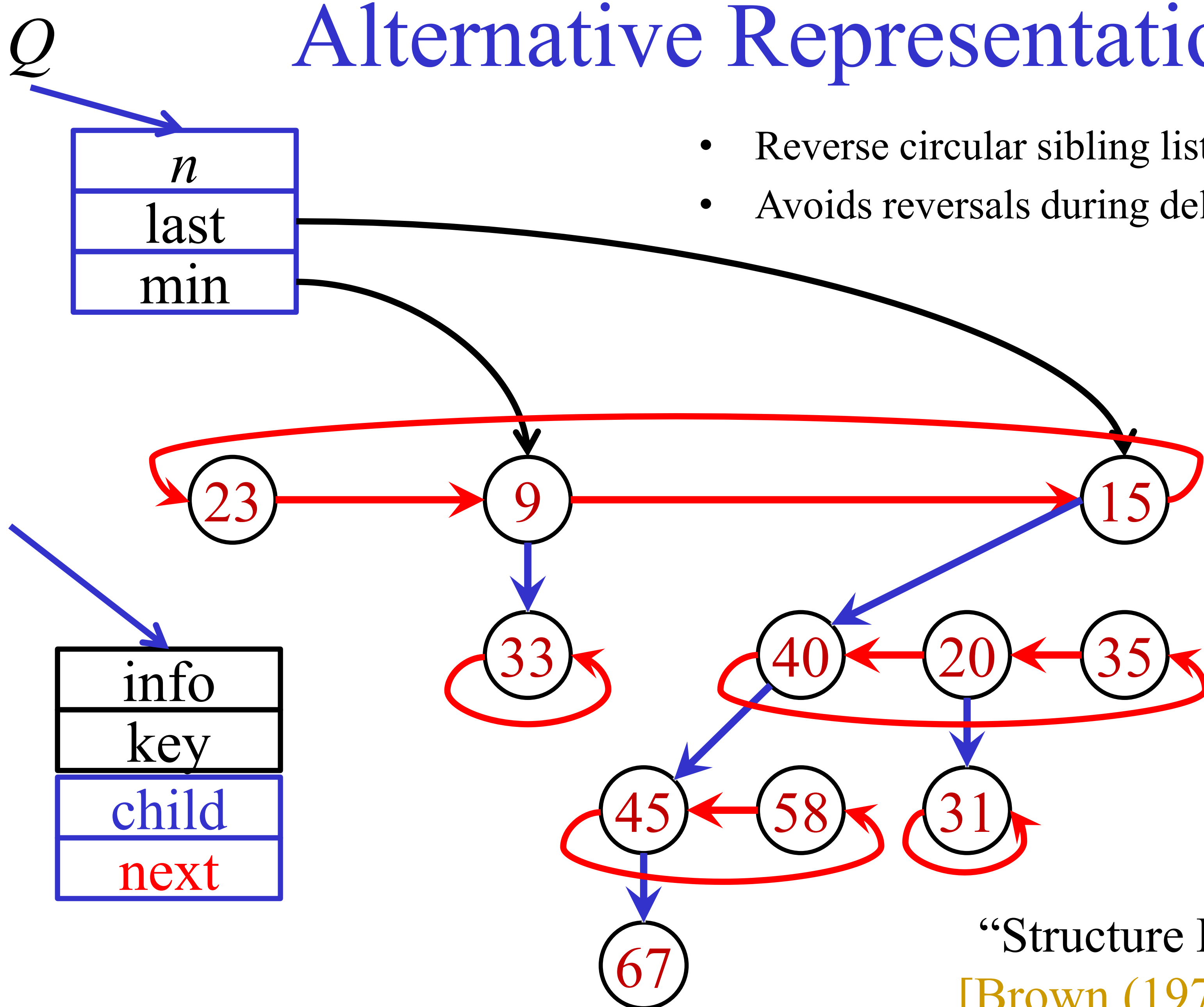


- 2 pointers per node
- No explicit rank information (n implicitly defines the ranks)
- Delete-min: need to reverse list of exposed roots

"Structure V"
[Brown (1978)]

Alternative Representation

- Reverse circular sibling list
- Avoids reversals during delete-min



"Structure R"
[Brown (1978)]

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

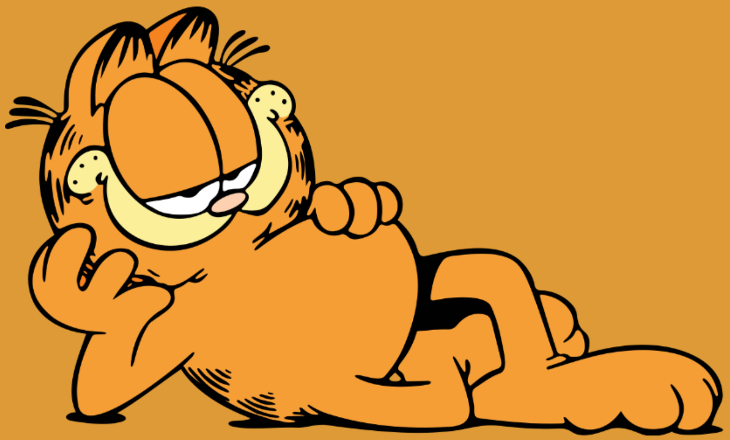
Linking in first representation

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

Lazy Binomial Heaps



Binomial Heaps

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

Worst case Amortized

Intuition

Intuition in a nutshell:

Be less rigid:

- Binomial heap:
  eagerly link heaps at meld/insert/delete-min
- Lazy binomial heap:
  lazily defer linking until next delete-min

Laziness will turn out beneficial (amortized).

Benefits of laziness

Lazy Insert

Add the new item to the list of roots (as B_0)
Update the pointer to root with minimal key

O(1) worst case time

Lazy Meld

Concatenate the two lists of Binomial trees
Update the pointer to root with minimal key

O(1) worst case time

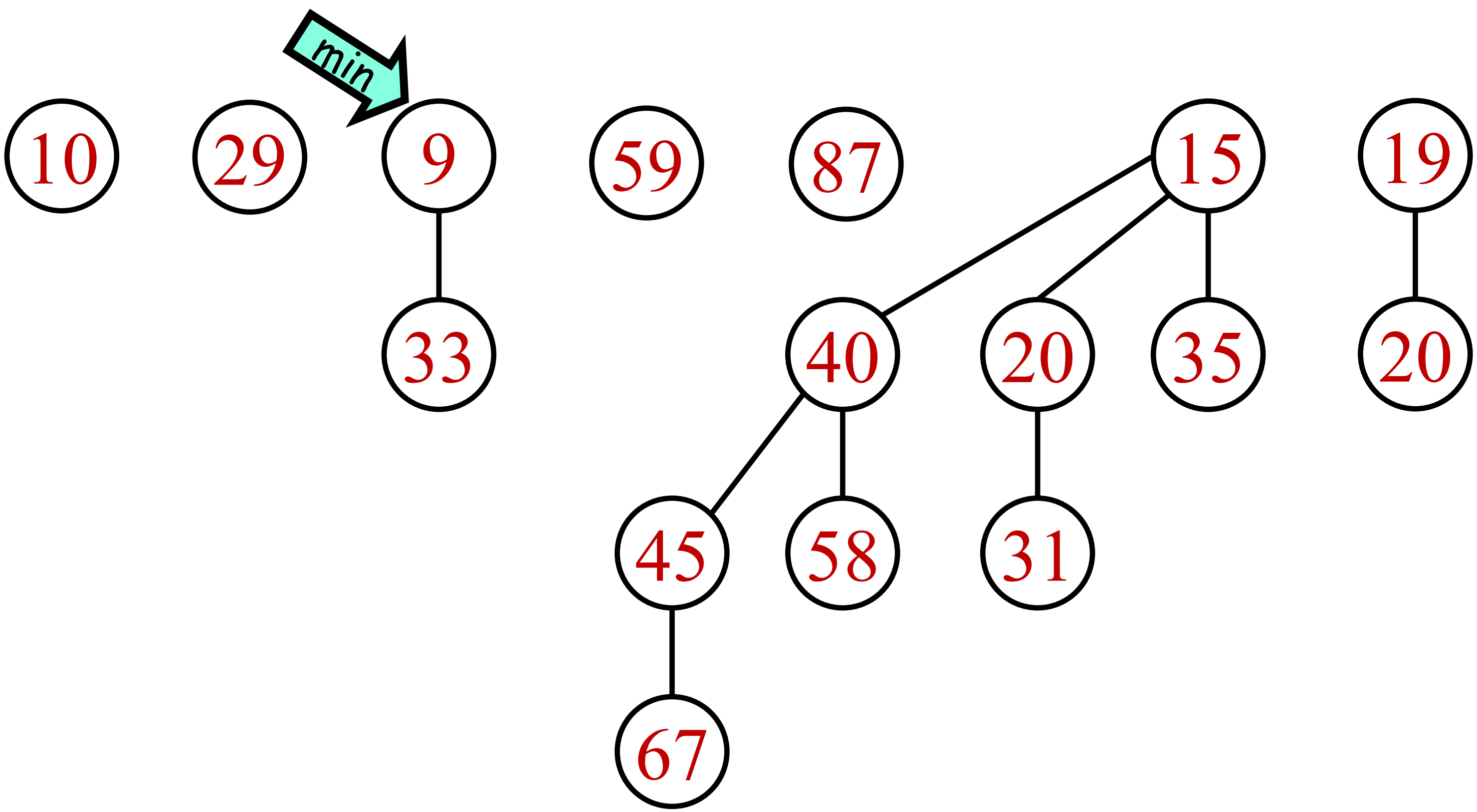
Lazy Binomial Heap

- Binomial heap:
  A list of heap ordered binomial trees,
  at most one of each degree
  (at most O(log n) trees)

- Lazy binomial heap:
  A list of heap ordered binomial trees
  ~~at most one of each degree~~
  (possibly even n trees of size 1)

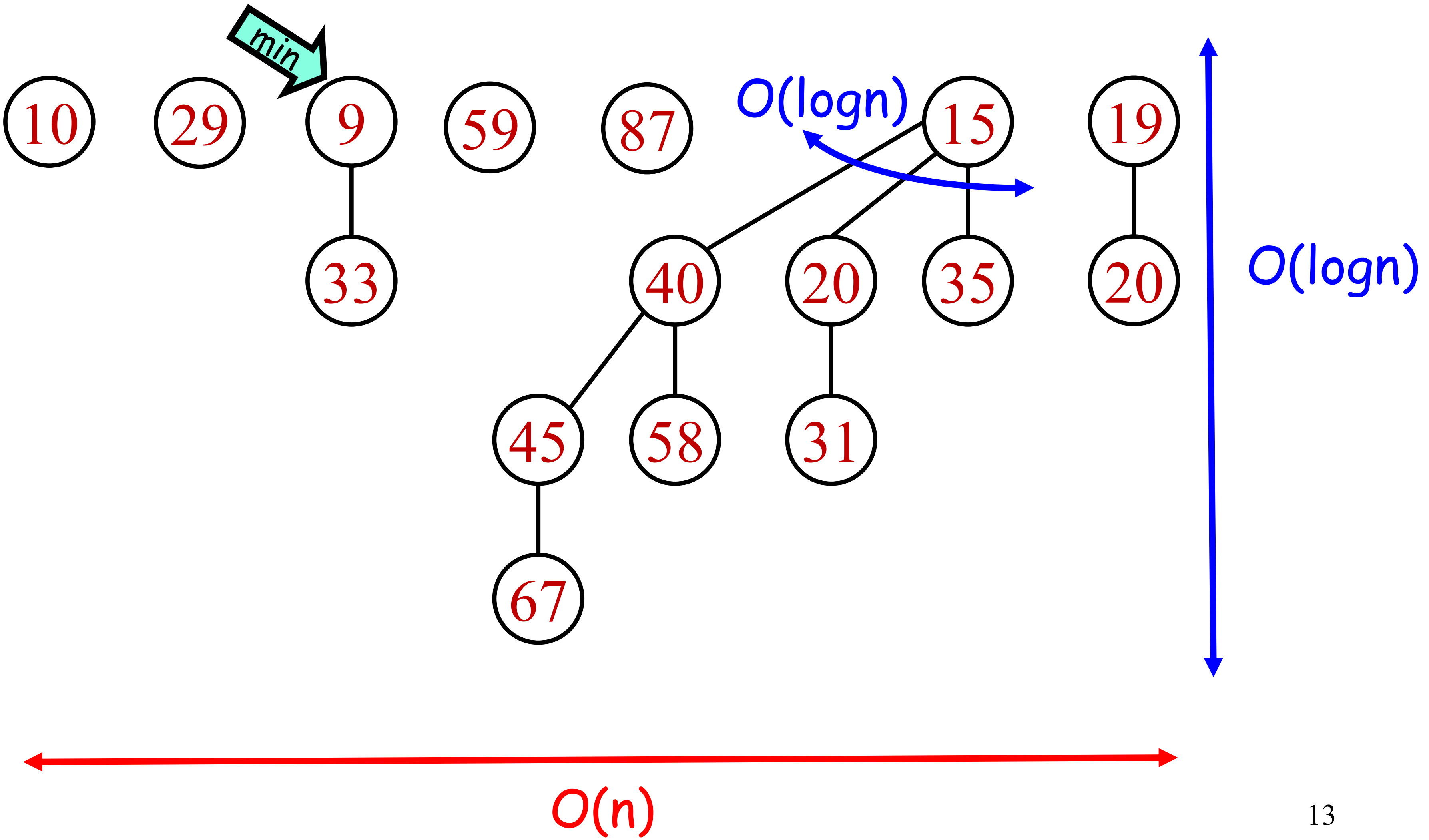
Lazy Binomial Heap

An arbitrary list of heap-ordered binomial trees
+ pointer to root with minimal key



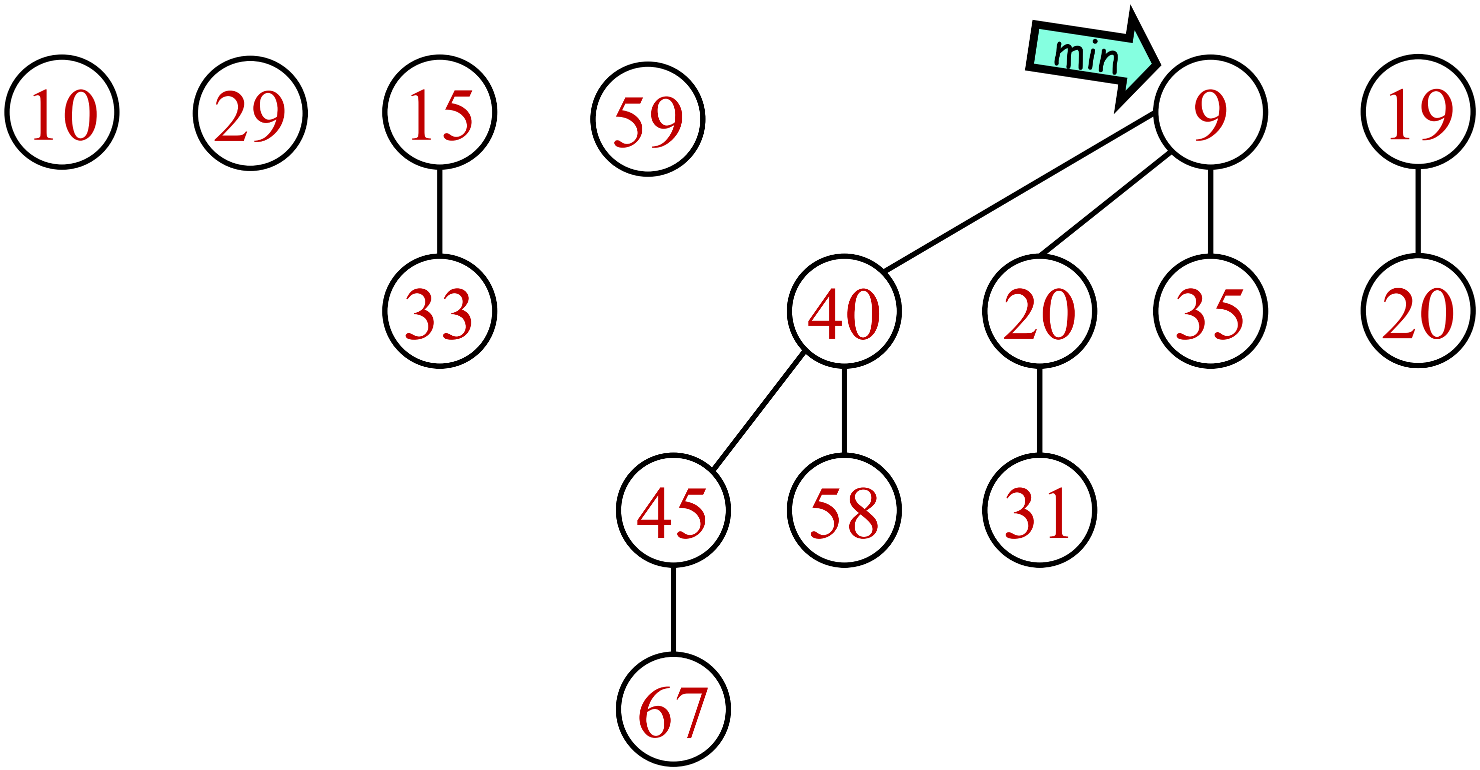
Lazy Binomial Heap - Intuition

- A lazy binomial heap can be wide but not deep



Worst Case for Delete-min

- Remove the minimum root and meld exposed trees to the heap in O(1)

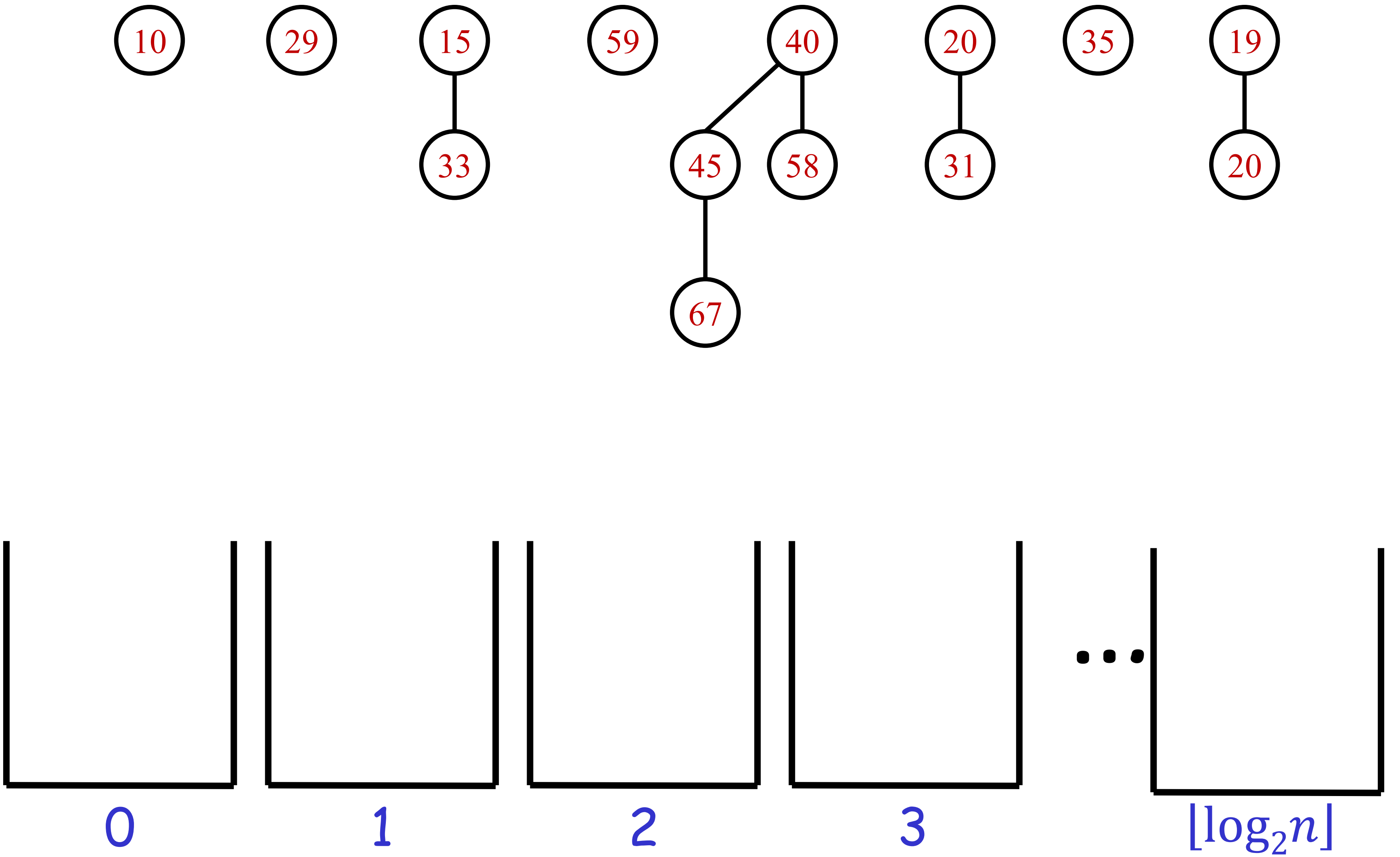


- But ?? time to find the new minimum!

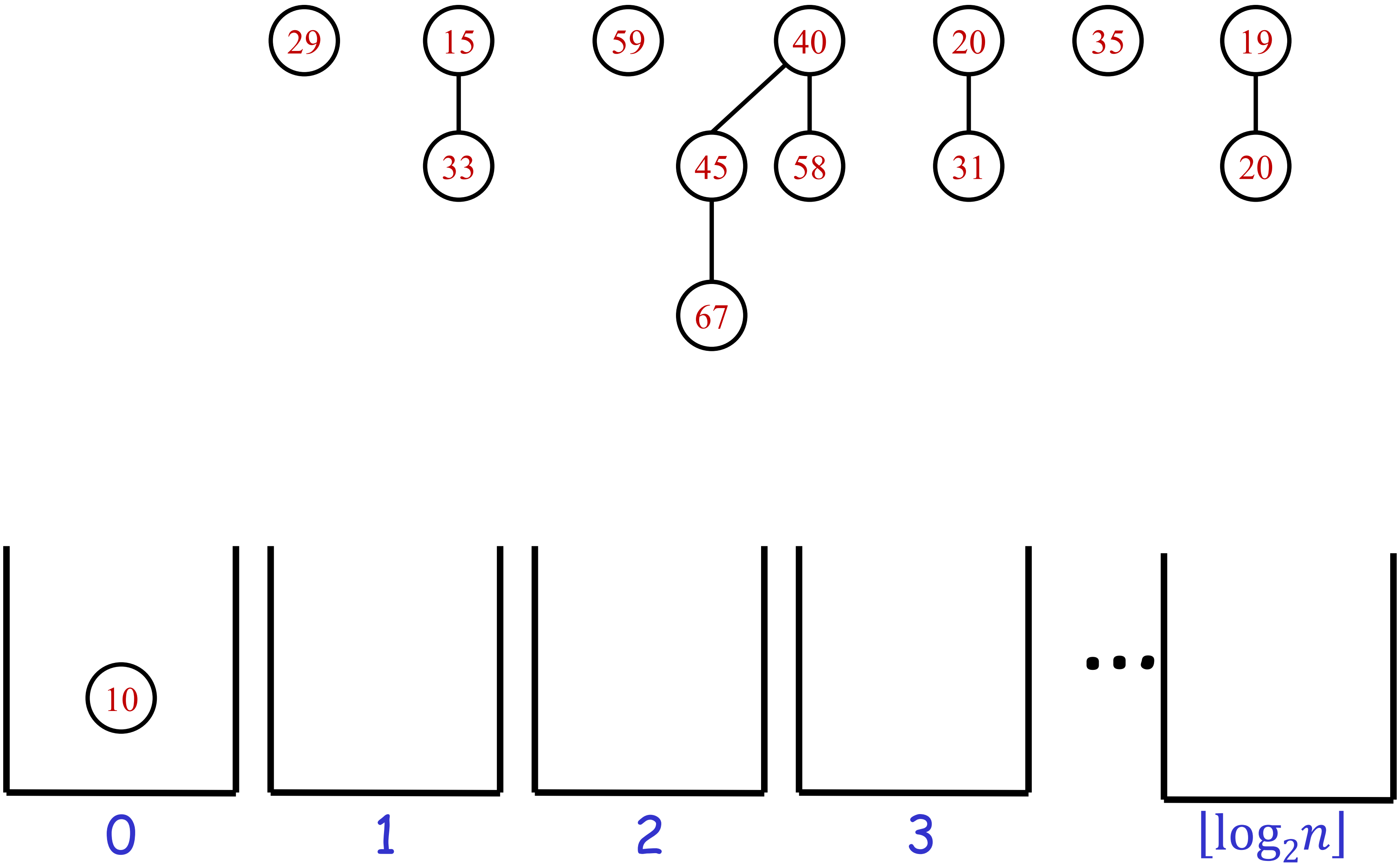
Amortized Delete-min

- Delete-Min is a good opportunity to restore order by linking trees of same degree
  - This is called consolidating / successive linking
  - We'll now show this makes Delete-Min run in $O(logn)$ amortized cost.

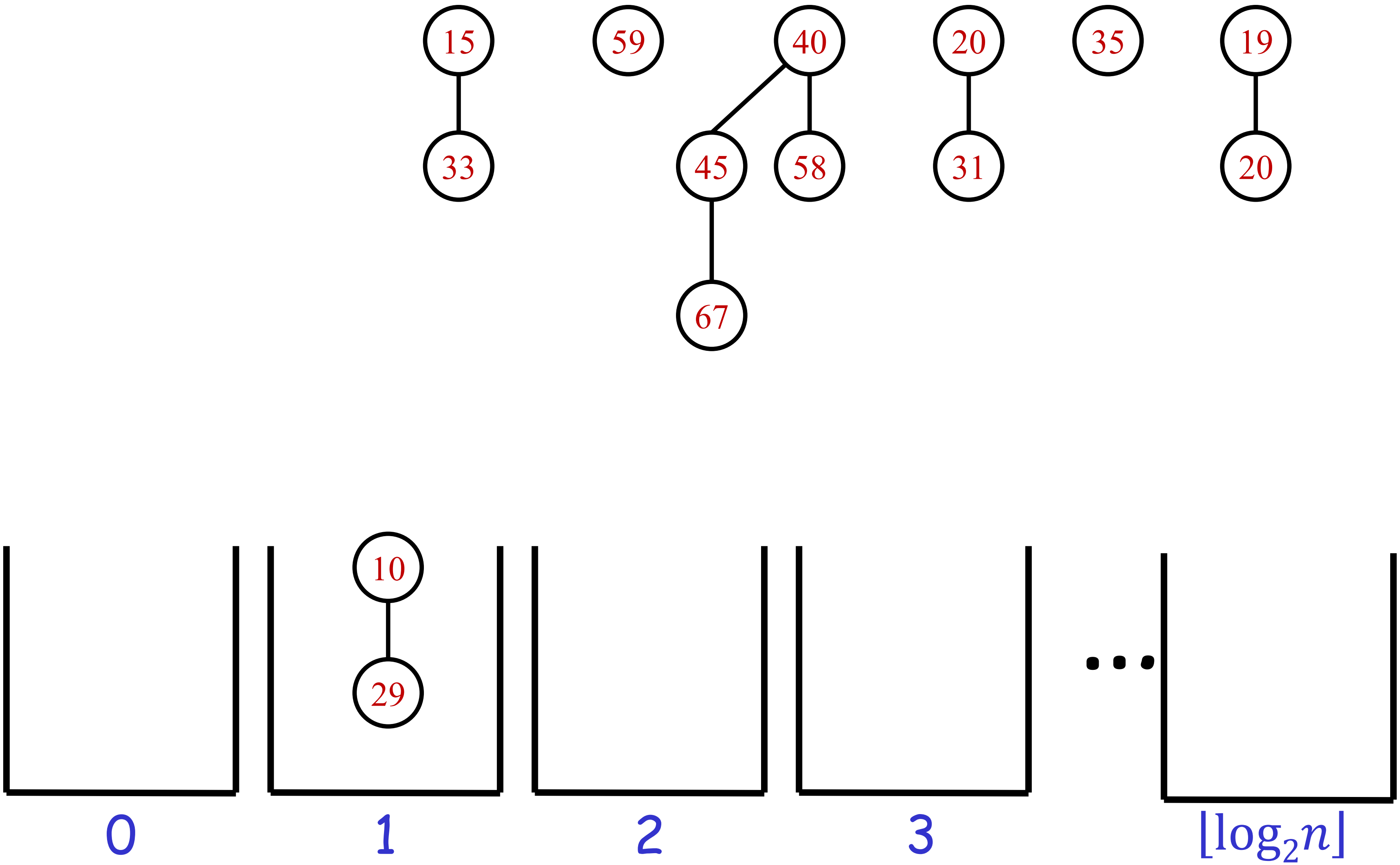
Consolidating / Successive Linking



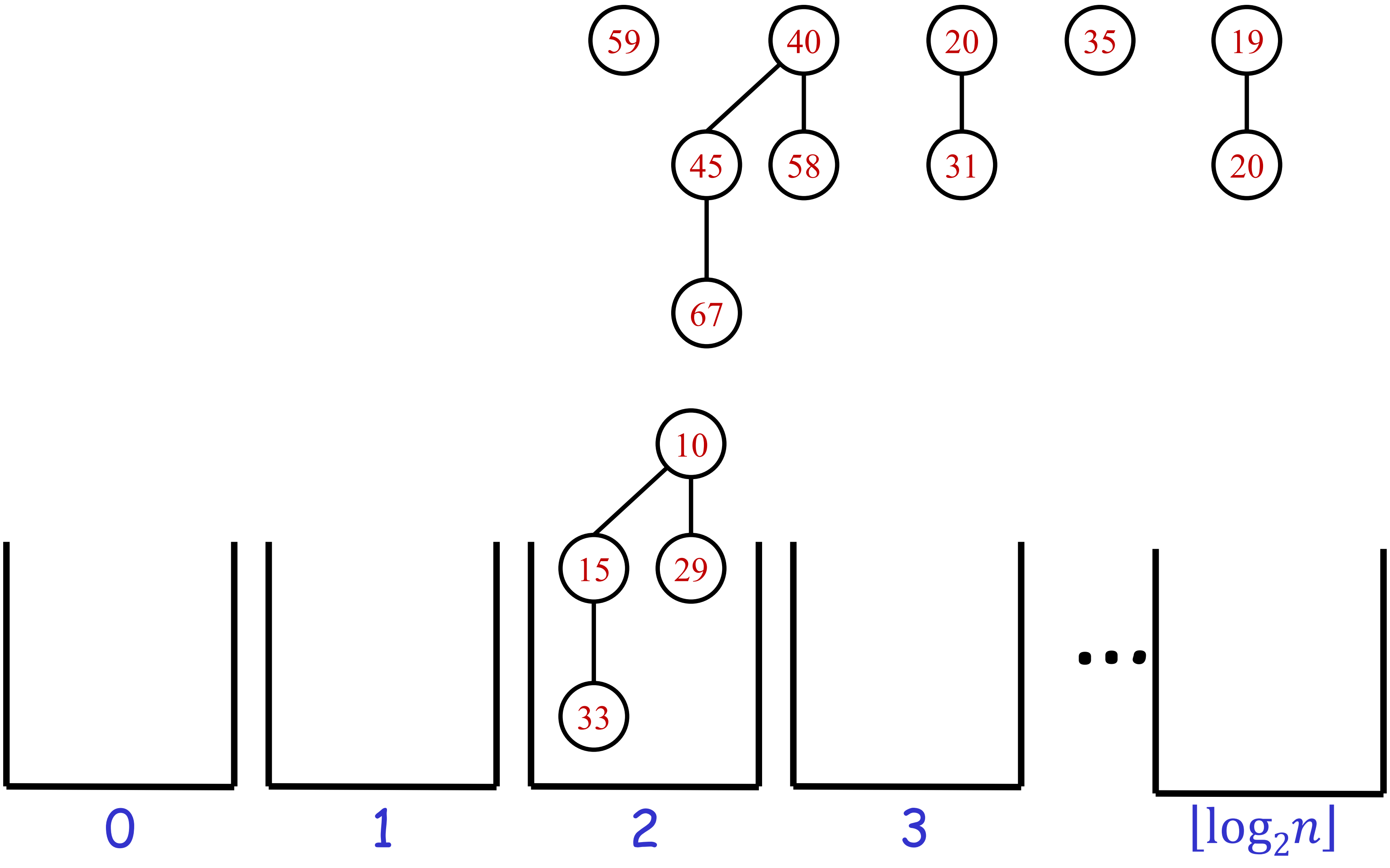
Consolidating / Successive Linking



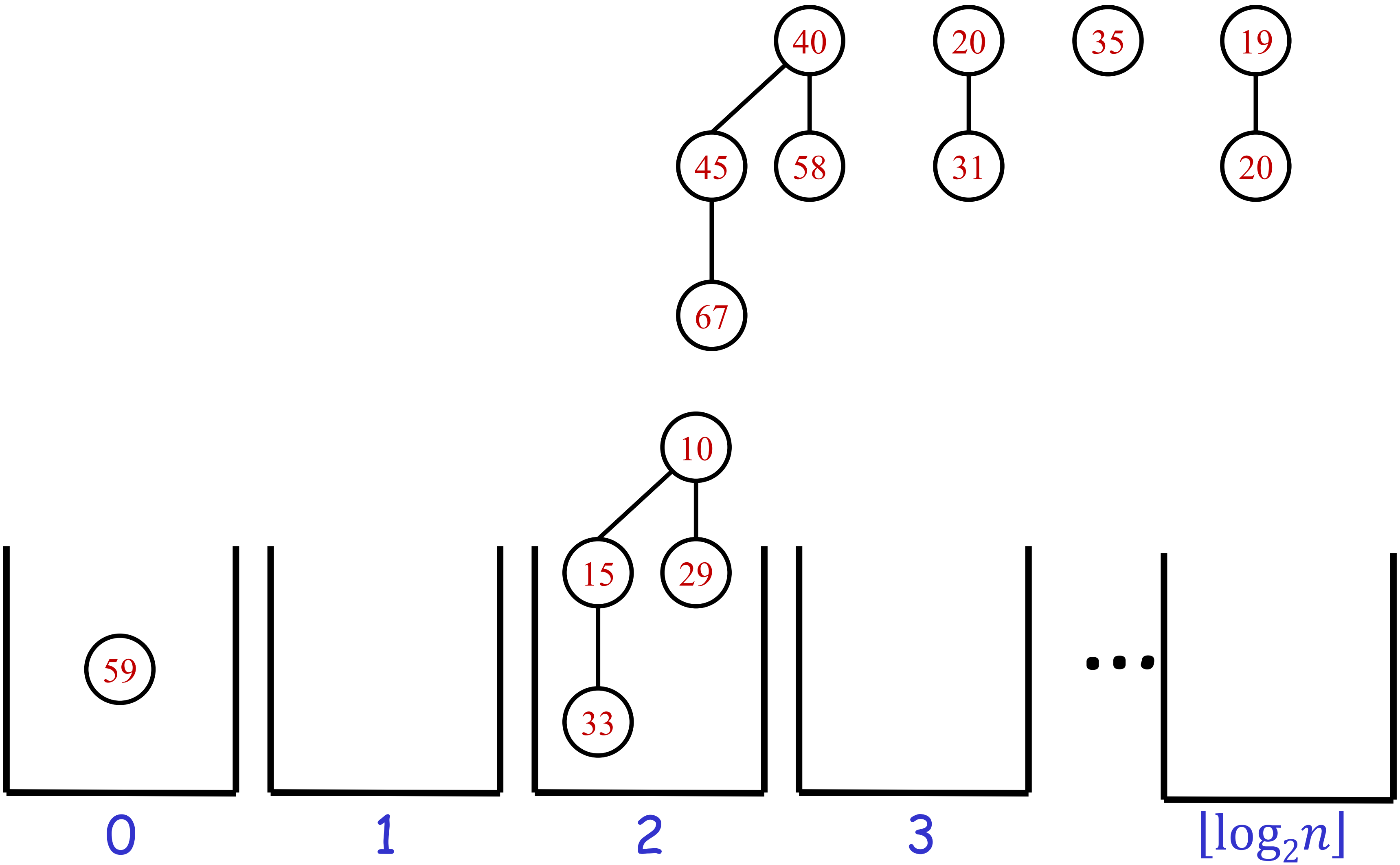
Consolidating / Successive Linking



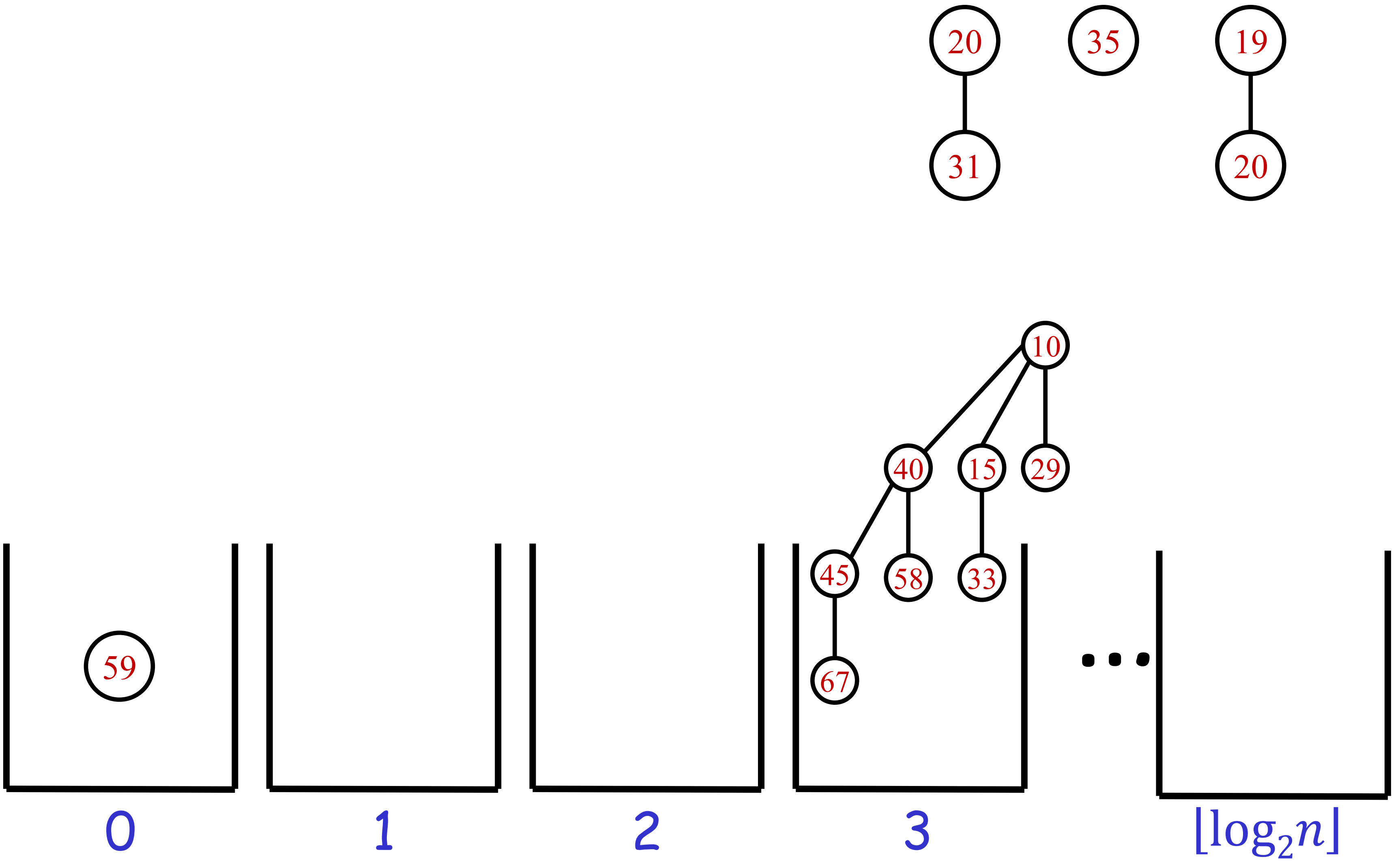
Consolidating / Successive Linking



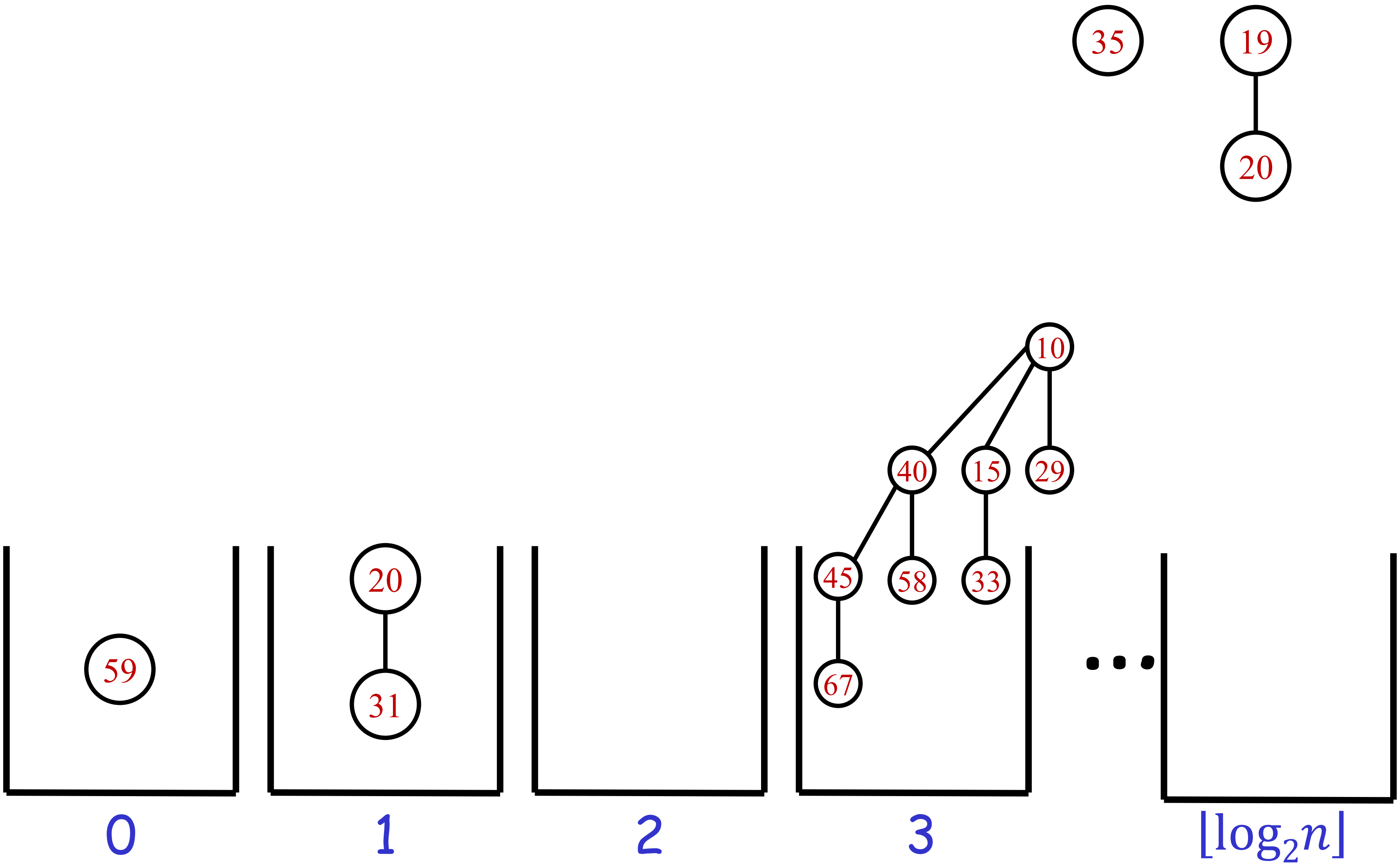
Consolidating / Successive Linking



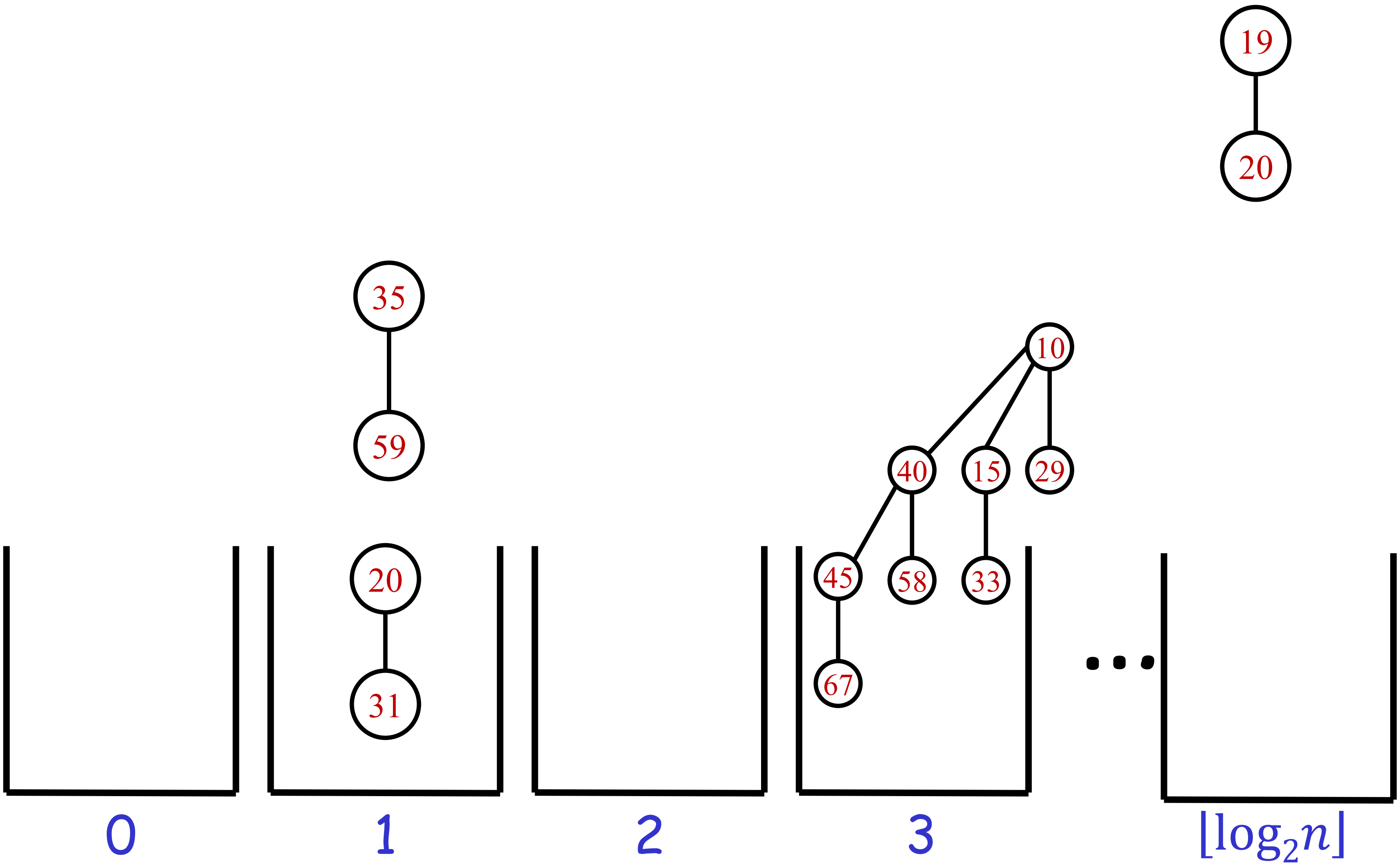
Consolidating / Successive Linking



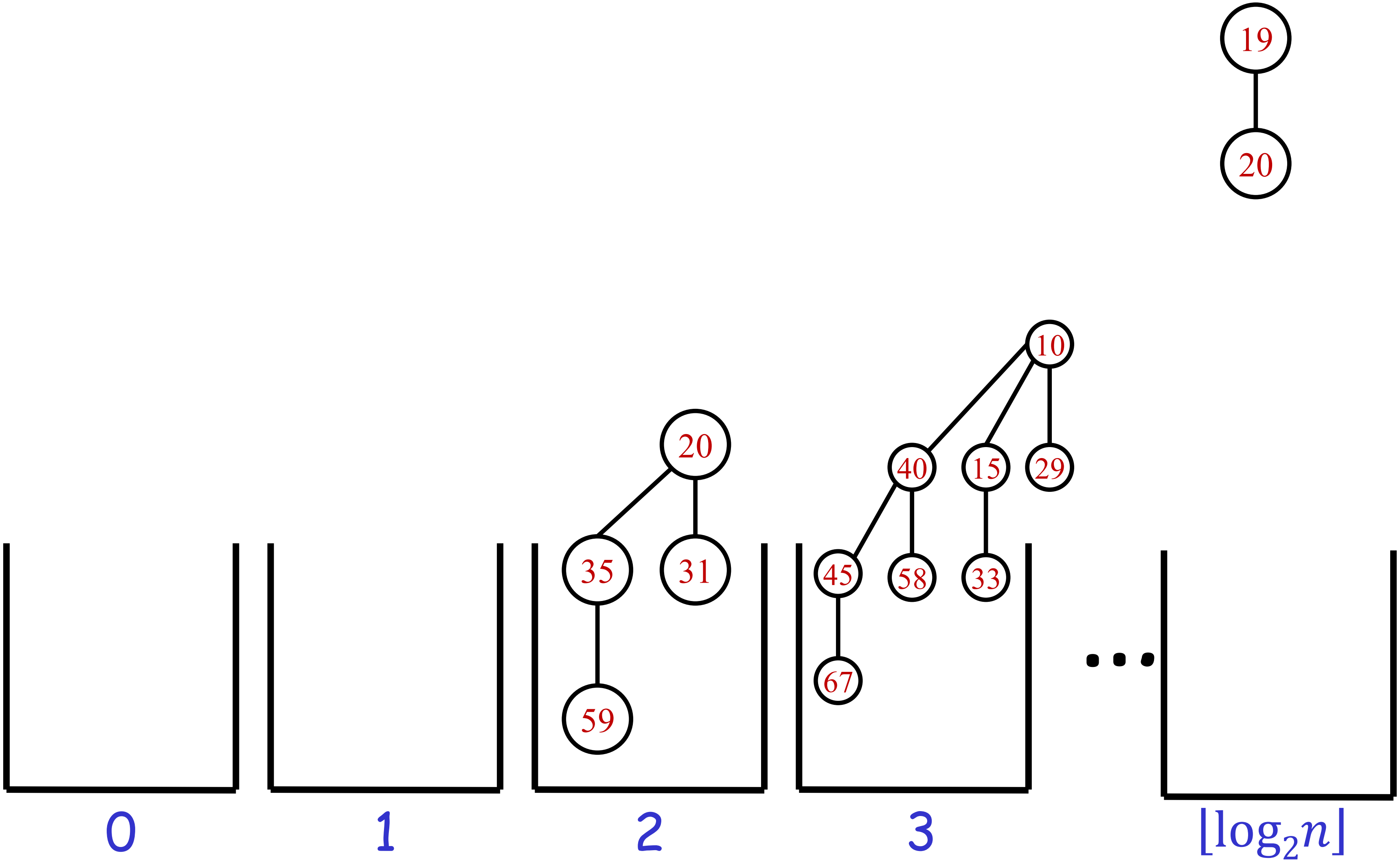
Consolidating / Successive Linking



Consolidating / Successive Linking



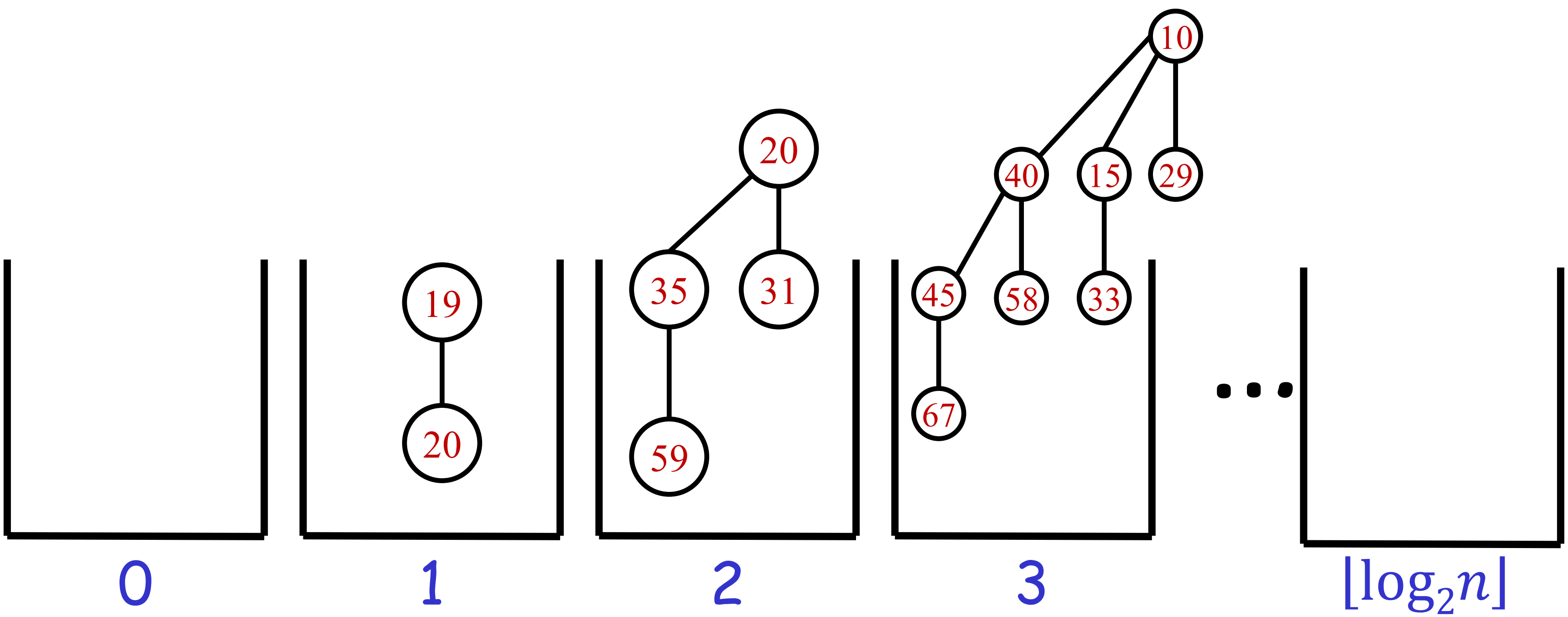
Consolidating / Successive Linking



Consolidating / Successive Linking

At the end of the process, we obtain a non-lazy binomial heap

(at most $\lfloor \log_2 n \rfloor + 1$ trees, at most 1 of each degree)



Worst Case Complexity of Consolidating

- Worst case (process all trees + all linkings):

$$T_0 - 1 + \log_2 n + L \leq 2(T_0 + \log_2 n) = O(n)$$

#trees before Delete-Min

#new trees exposed in Delete-Min $\leq \lfloor \log_2 n \rfloor$

total #links through consolidating

$L \leq \mathrm{T}_0 + logn$
why?

$T_0 \leq n$

Amortized Complexity of Consolidating

- So actual worst case cost:

  (scaled) $T_0 + \log_2 n$

- Claim: amortized complexity $O(logn)$
  - Intuition?
    Who can "pay for" consolidation?
    Potential function?

Amortized Complexity of Consolidating

$$\Phi = \text{Number of Trees}$$
$$\Delta\Phi = T_1 - T_0$$

#trees after the process = $O(logn)$

- Amortized cost = actual cost + $\Delta\Phi$

$$= (T_0 + \log n) + (T_1 - T_0)$$

*up to scaling

$$= \log n + T_1$$
$$= O(logn)$$

Lazy Binomial Heaps

- Inserts pays for Del-min:

	Actual cost	Potential: Δ Trees*	Amortized cost
Lazy Insert	O(1)	+1	O(1)
Find-min	O(1)	0	O(1)
Delete-min	$T_0 + log n$	$T_1 - T_0$	$O(\log n)$
Decrease-key	$O(\log n)$	0	$O(\log n)$
Lazy Meld	O(1)	0	O(1)

* up to scaling

Lazy Binomial Heaps

	Binary Heaps ➡	**Binomial Heaps** ➡	**Lazy Binomial Heaps** ➡	**Fibonacci Heaps**
Insert	$O(\log n)$	←	$\mathbf{O(1)}$	
Find-min	$O(1)$	←	←	
Delete-min	$O(\log n)$	←	←	
Decrease-key	$O(\log n)$	←	←	
Meld / Join	$O(n)$	$\mathbf{O(\log n)}$	$\mathbf{O(1)}$	

Worst case

Amortized

Fibonacci Heaps

[Fredman-Tarjan (1987)]

Fibonacci Heaps

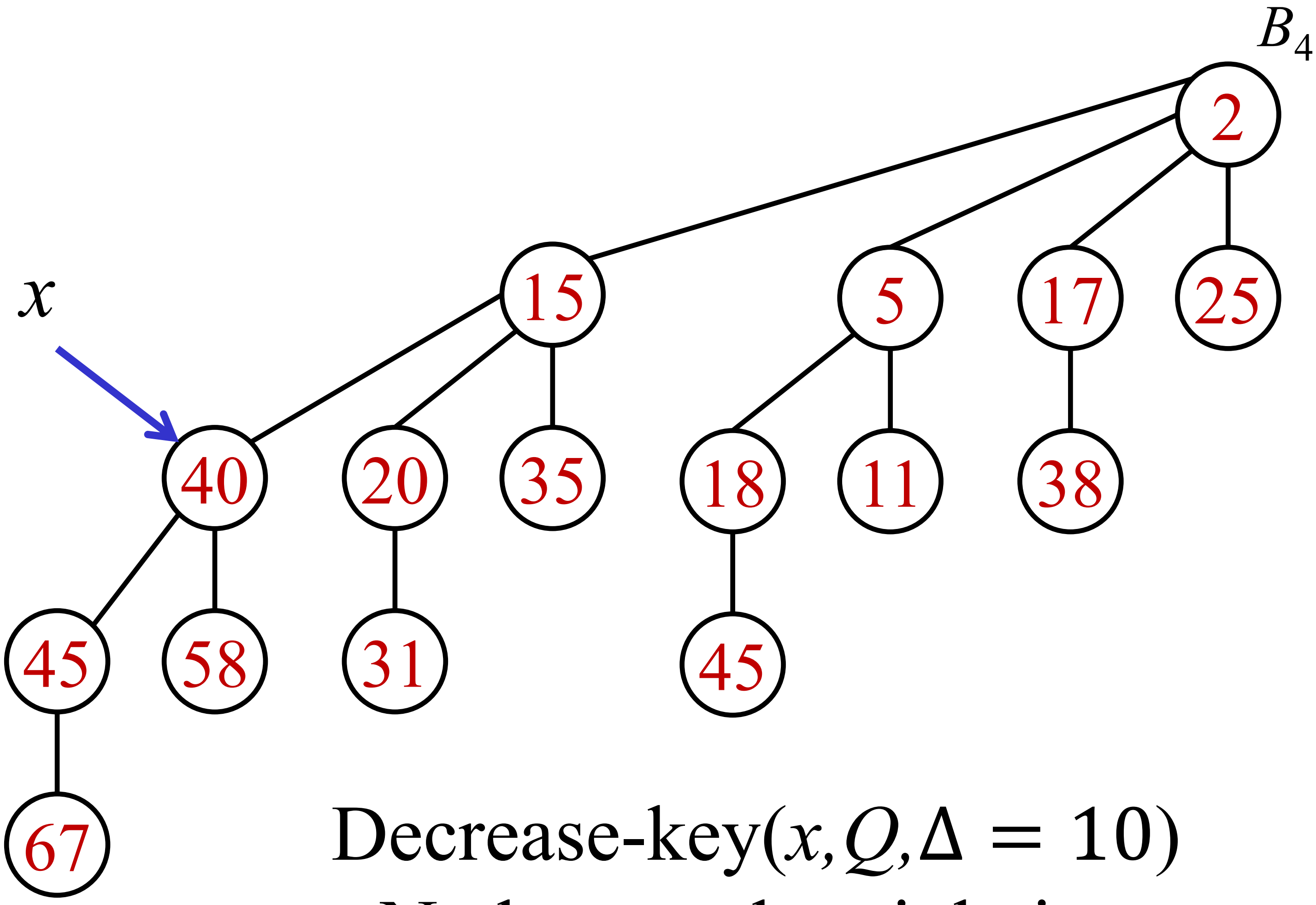
	Binary Heaps ➡	Binomial Heaps ➡	Lazy Binomial Heaps ➡	Fibonacci Heaps
Insert	O(log n)	←	**O(1)**	←
Find-min	O(1)	←	←	←
Delete-min	O(log n)	←	←	←
Decrease-key	O(log n)	←	←	**O(1)**
Meld / Join	O(n)	**O(log n)**	**O(1)**	←

Worst case | Amortized

Intuition in a nutshell

- Decrease-key: we do not want to fix heap order all the way up ($O(\log n)$).

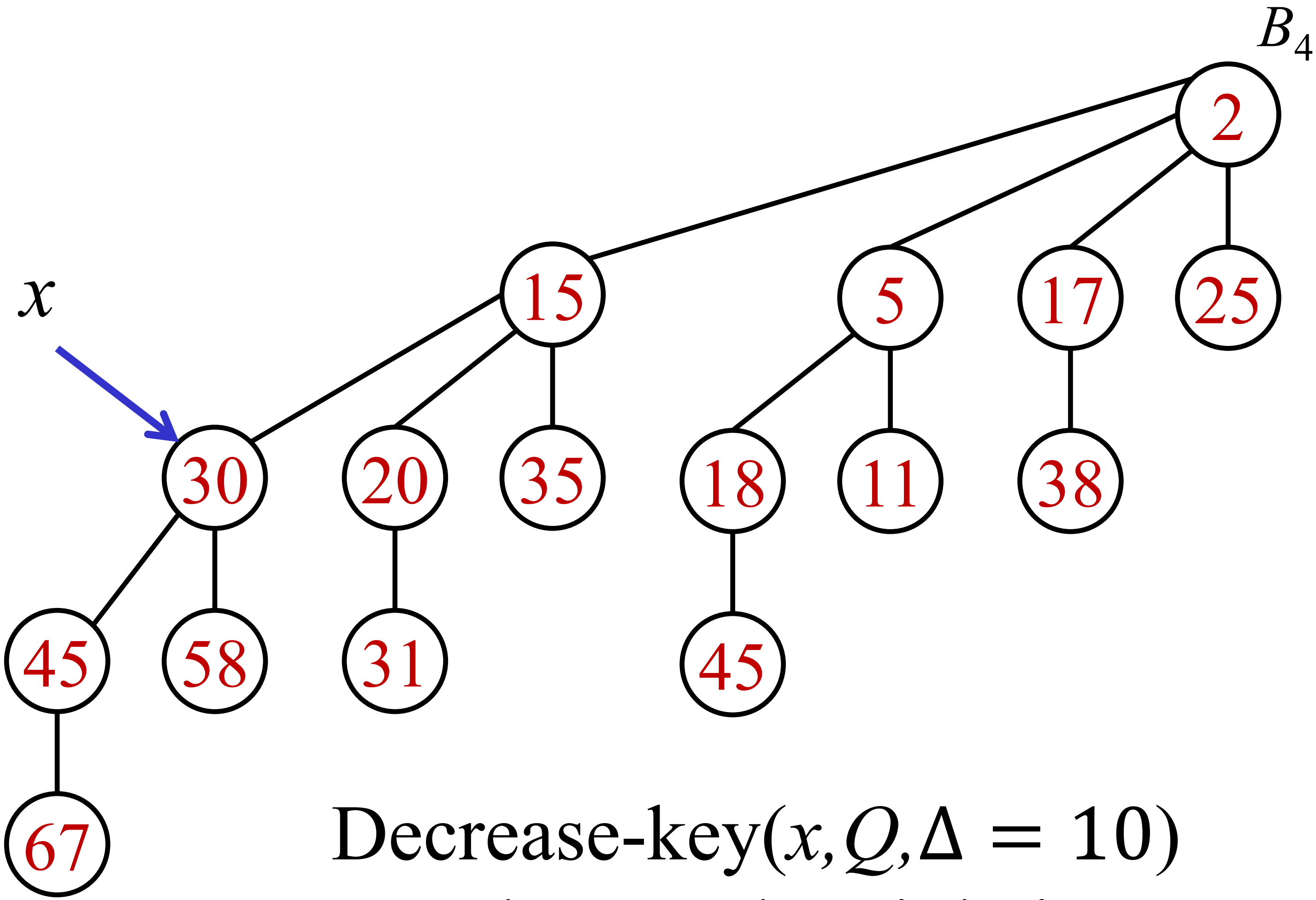
Decrease-key in O(1) time?



Decrease-key$(x, Q, \Delta = 10)$

No heap order violation

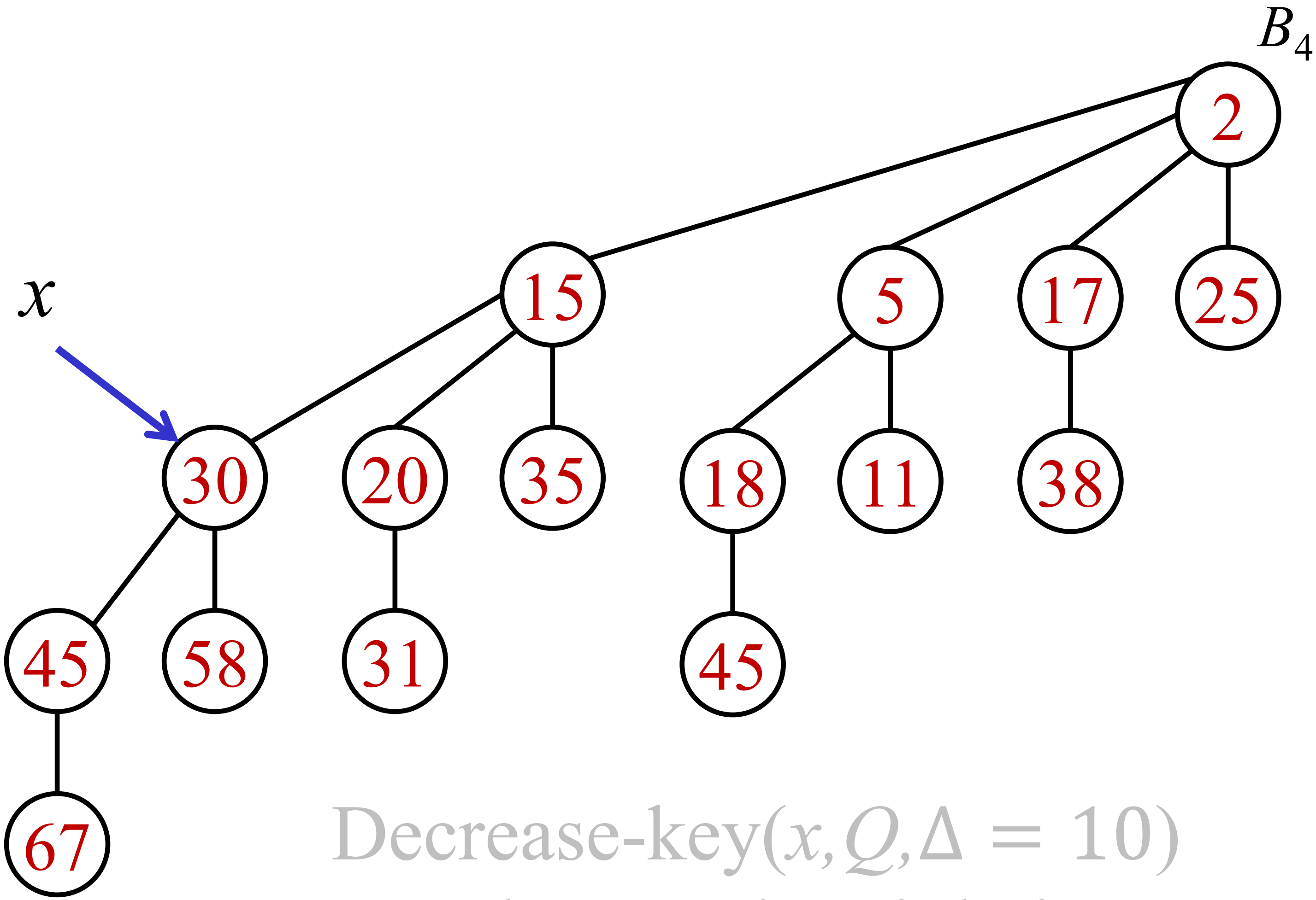
Decrease-key in O(1) time?



Decrease-key$(x, Q, \Delta = 10)$

No heap order violation

Decrease-key in O(1) time?

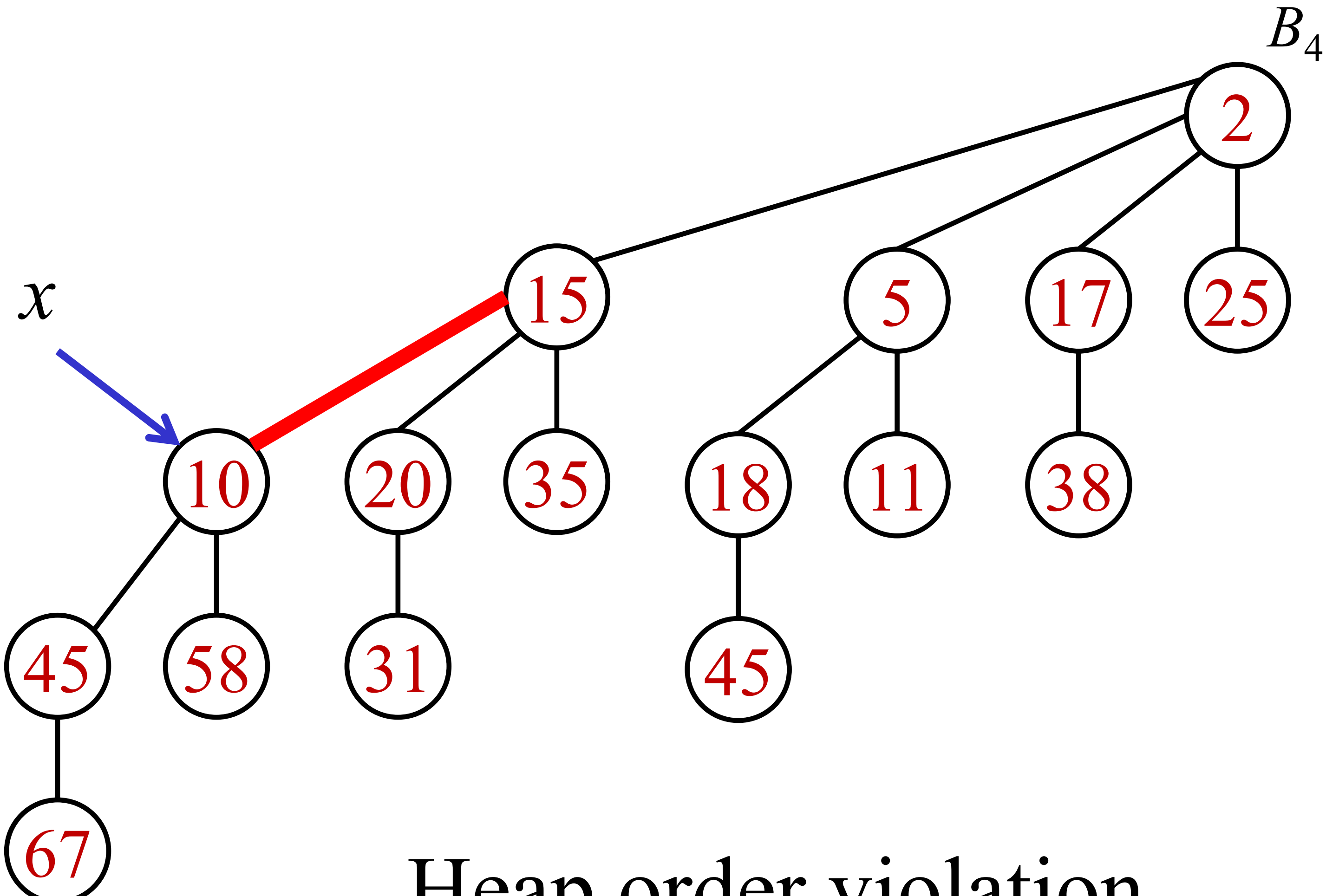


Decrease-key($x, Q, \Delta = 10$)

No heap order violation

Decrease-key($x, Q, \Delta = 20$)

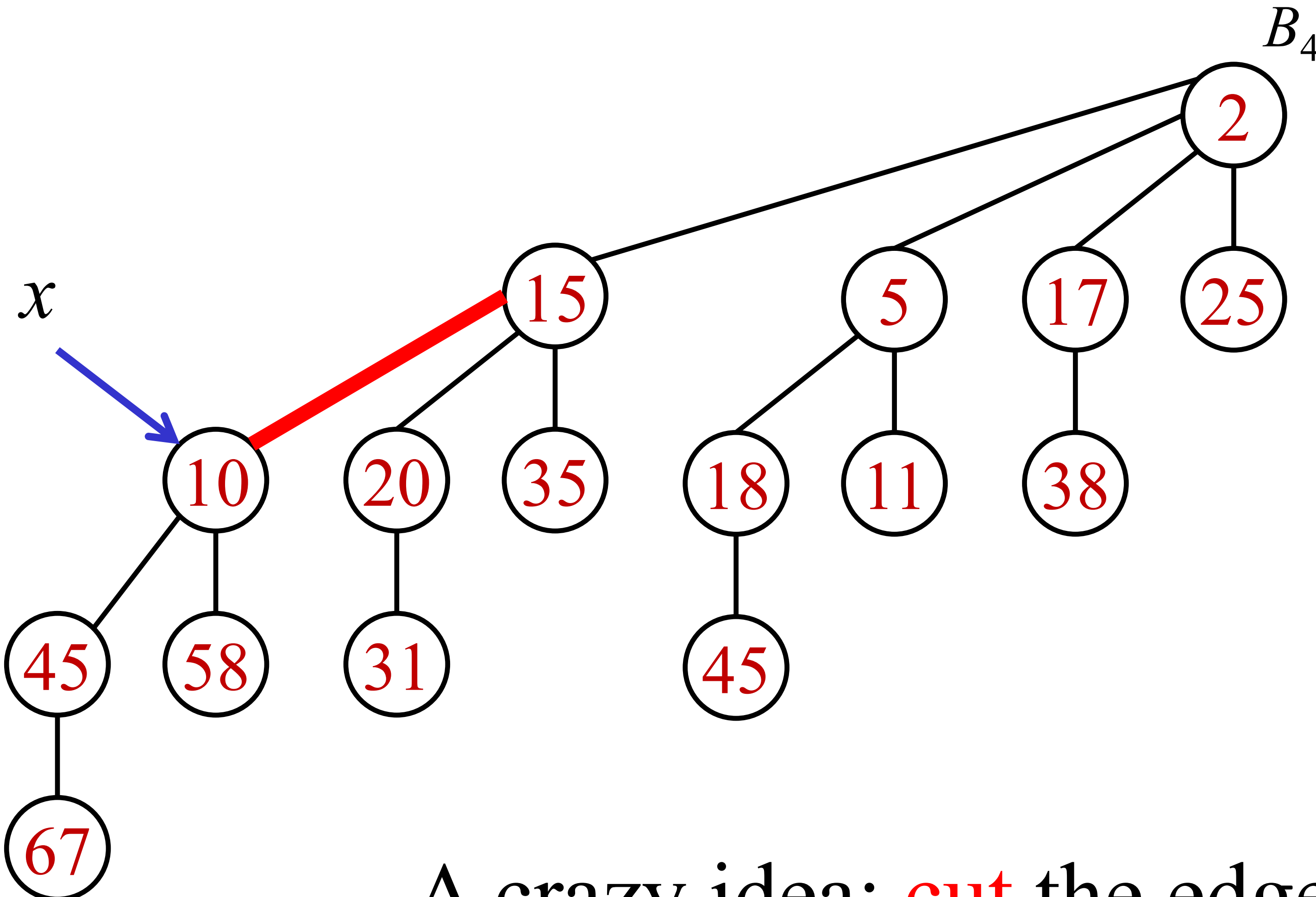
Decrease-key in O(1) time?



Heap order violation

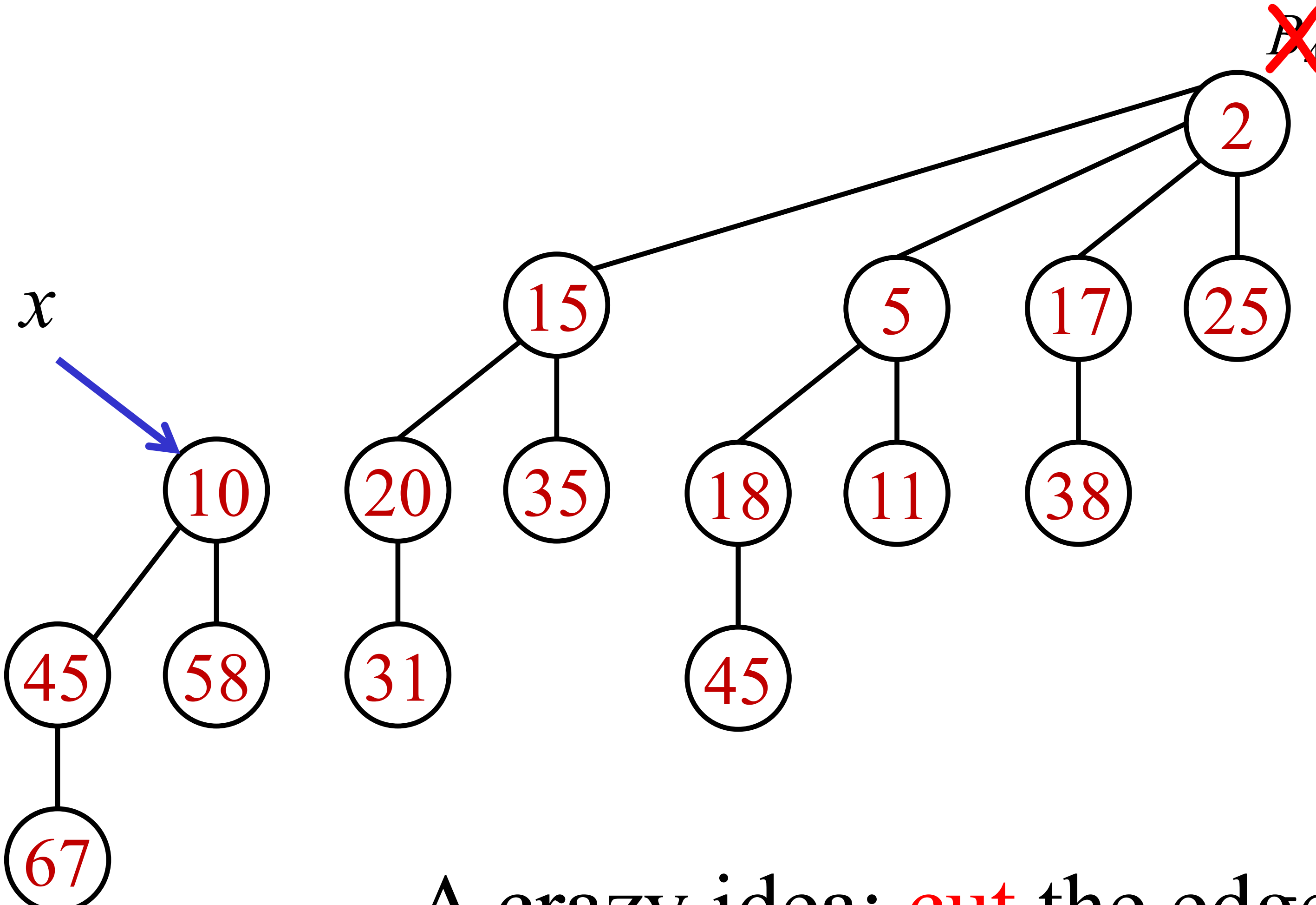
Can we avoid the $O(logn)$?

Decrease-key in O(1) time?



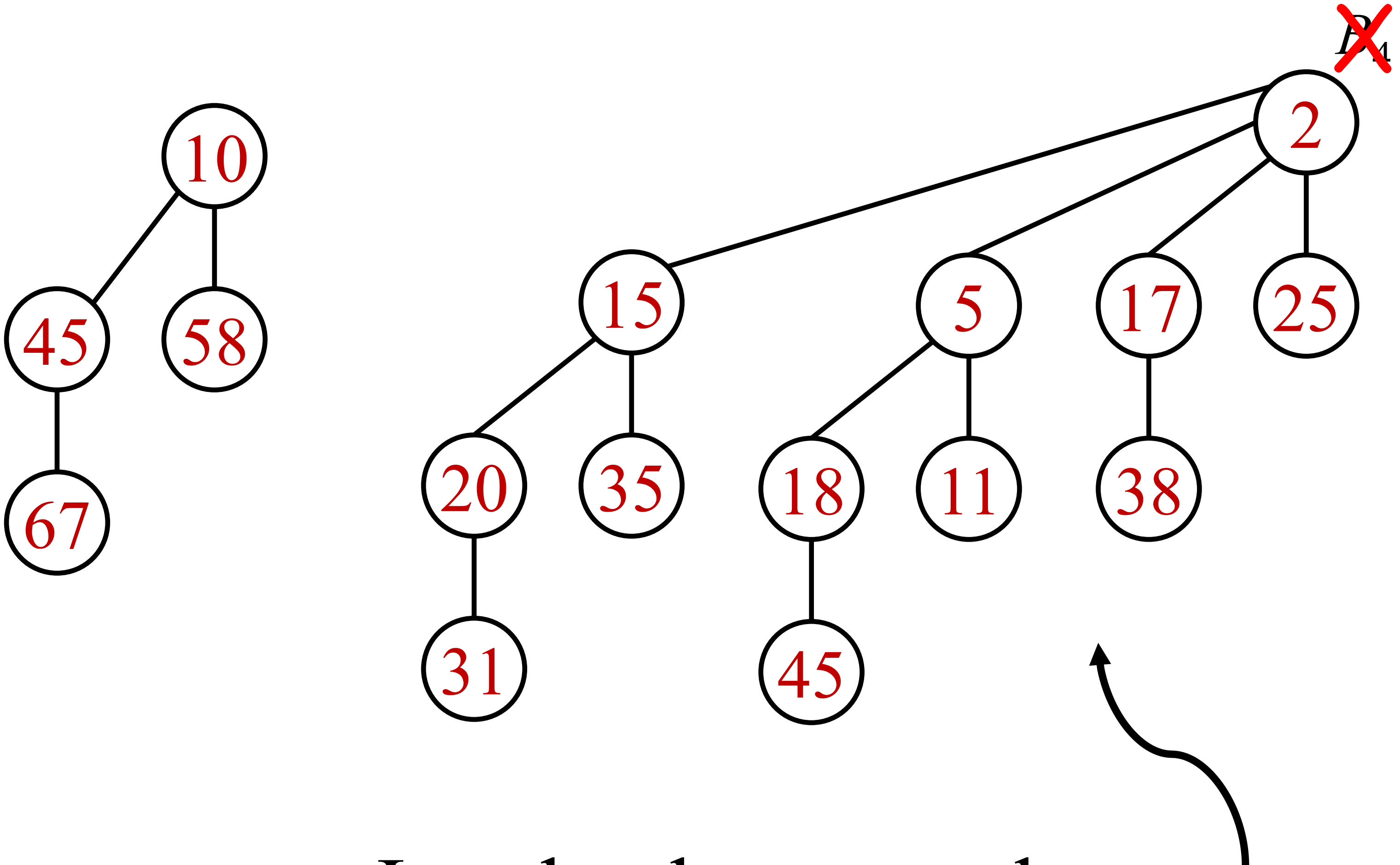
A crazy idea: cut the edge
(add x to root list)

Decrease-key in O(1) time?



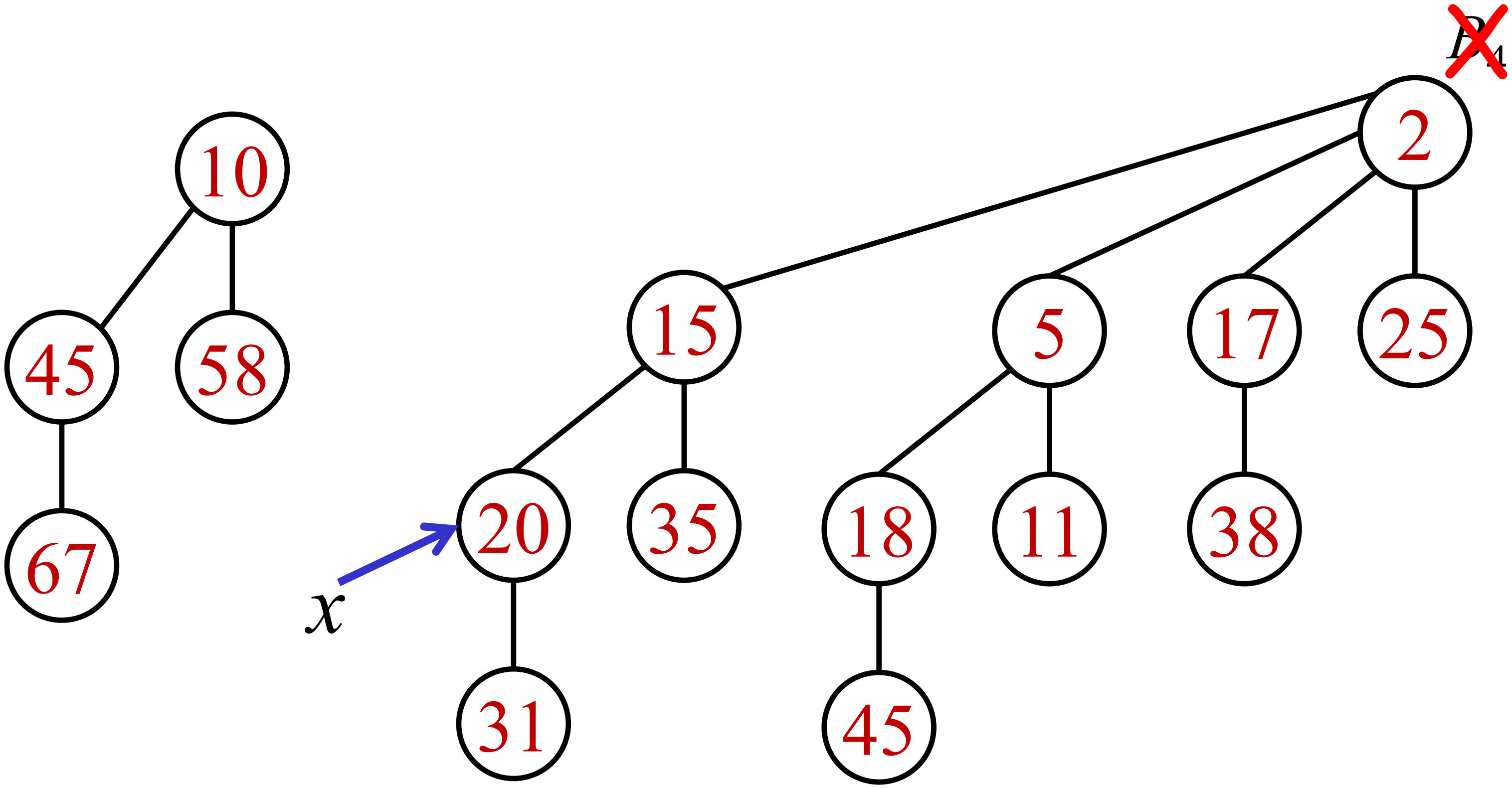
A crazy idea: cut the edge
(add x to root list)

Decrease-key in O(1) time?



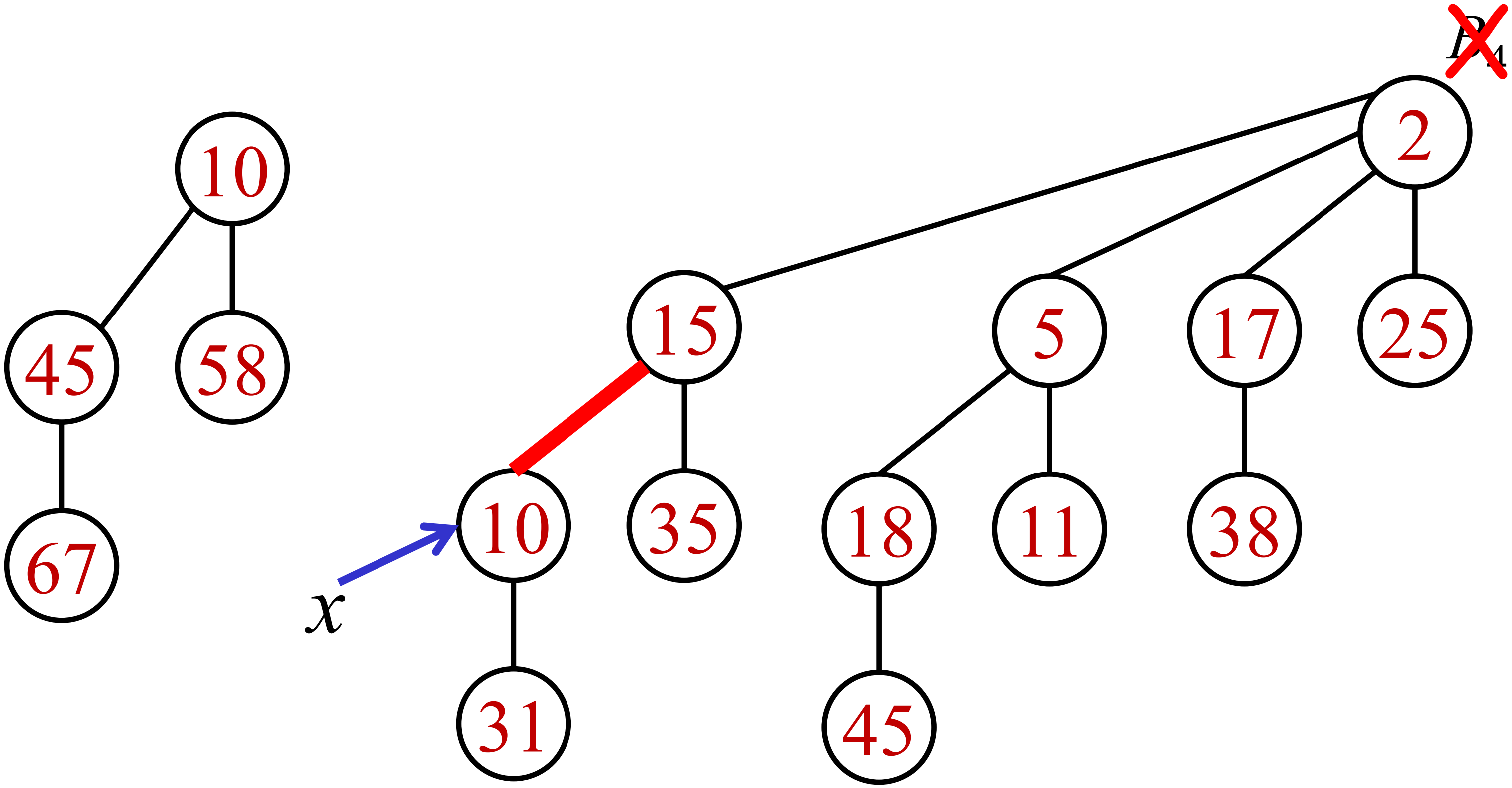
Involved trees no longer binomial

Decrease-key in O(1) time?



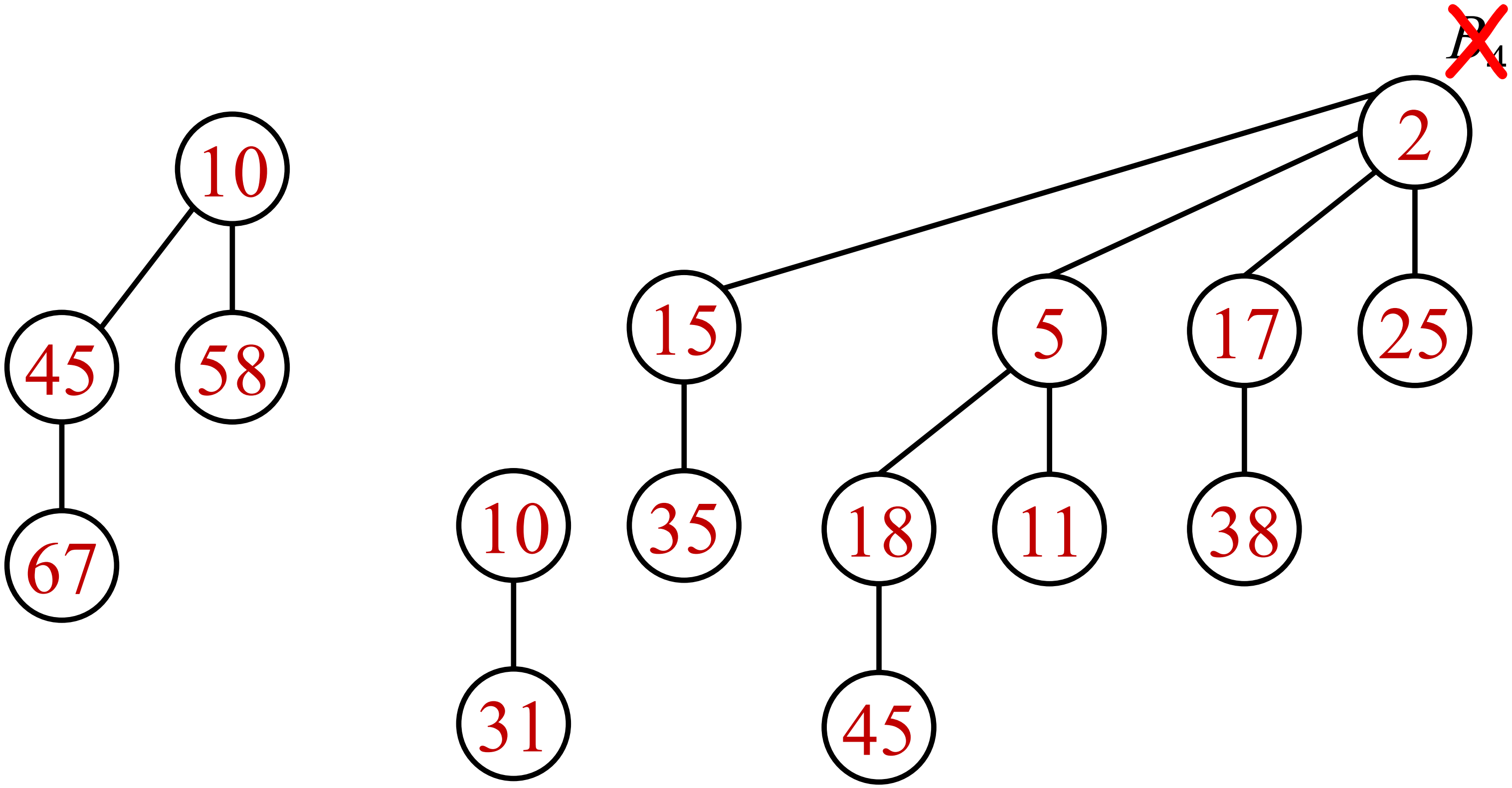
$$\text{Decrease-key}(x, Q, \Delta = 10)$$

Decrease-key in O(1) time?



Cut the edge

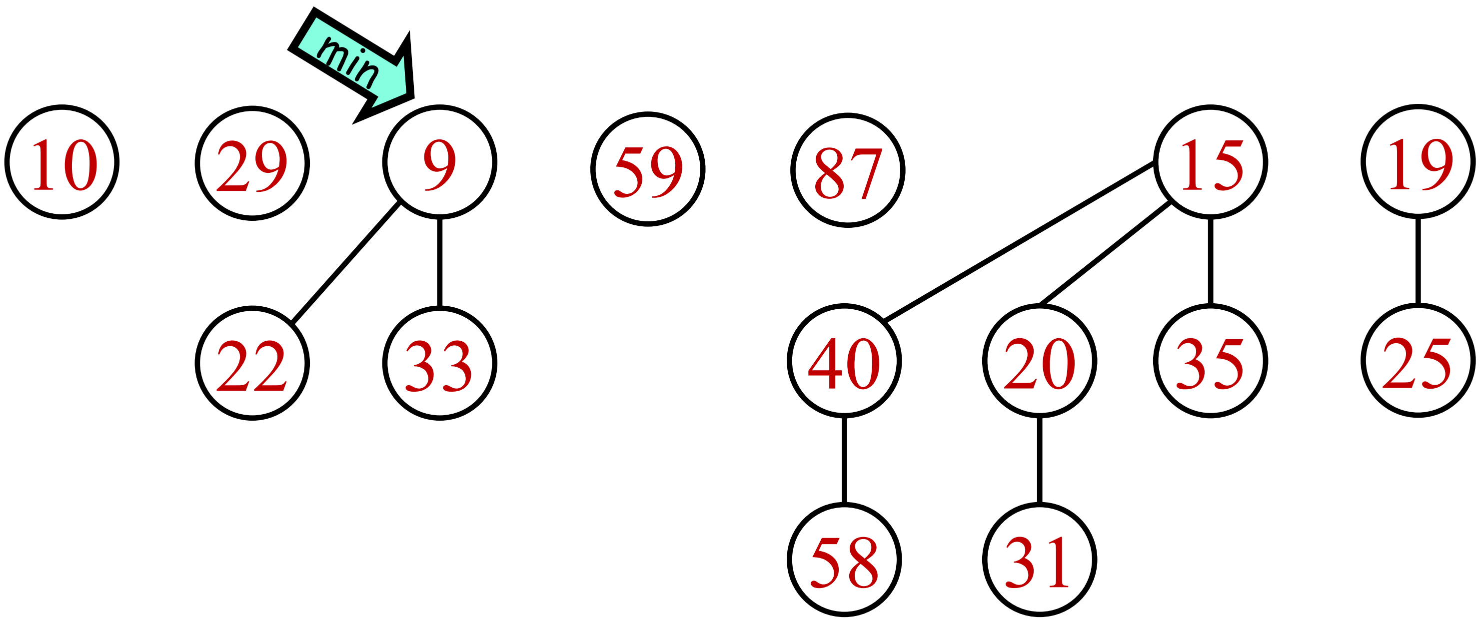
Decrease-key in O(1) time?



Cut the edge

Fibonacci Heaps

A list of heap-ordered ~~binomial~~ trees
+ pointer to root with minimal key

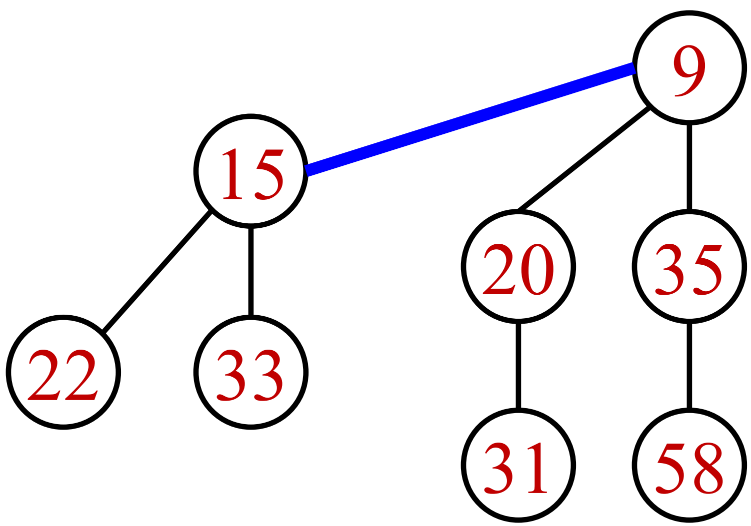


All operations, except decrease-key are the same as in lazy binomial heaps

Note on Linking in Fibonacci Heaps

- Linking 2 trees (after Delete-min) is done the same as in lazy binomial heaps:
  - Link 2 trees of same degree

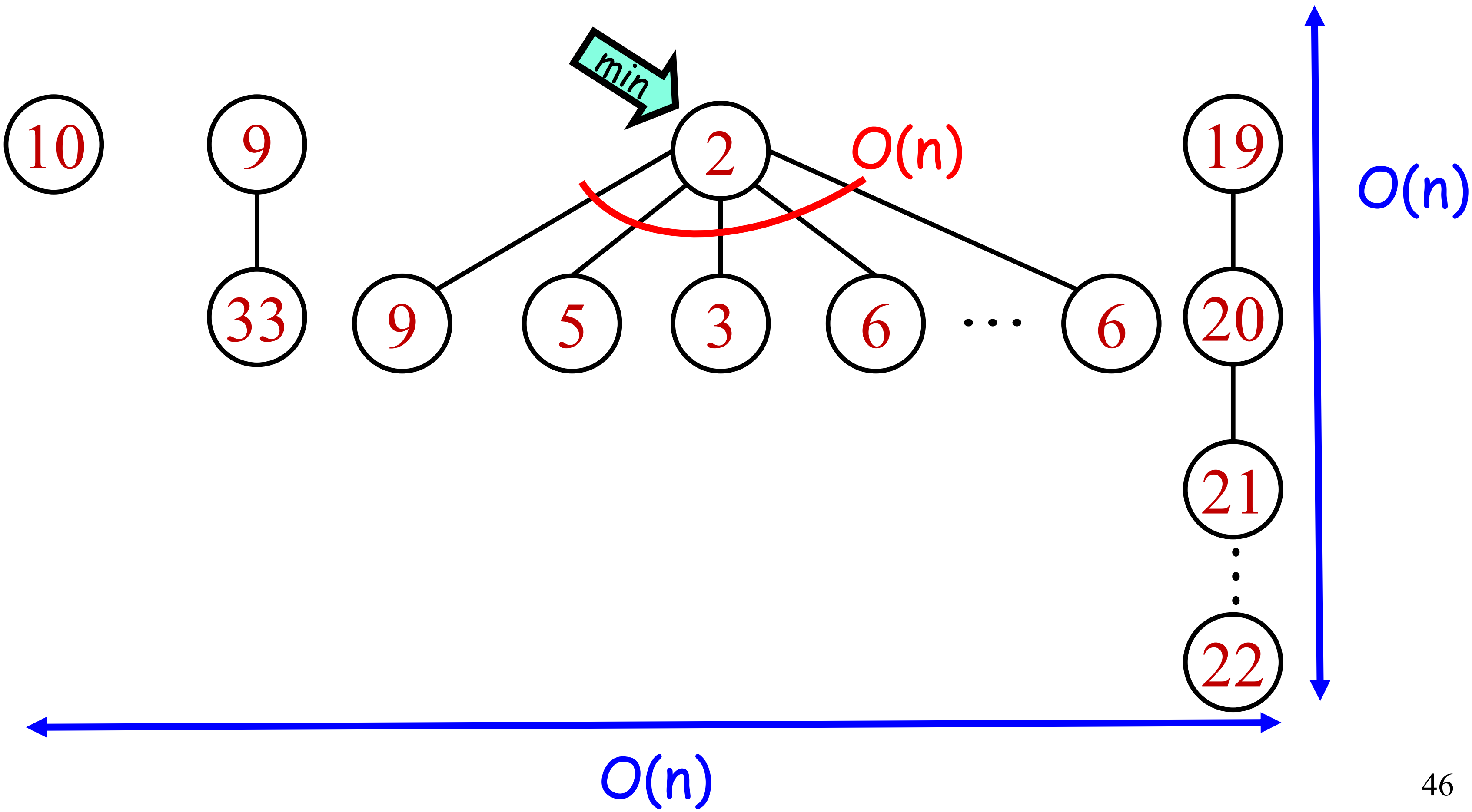
- Only difeerence: trees not necessarily binomial



Linking 2 trees of degree 2

Fibonacci Heap - Intuition

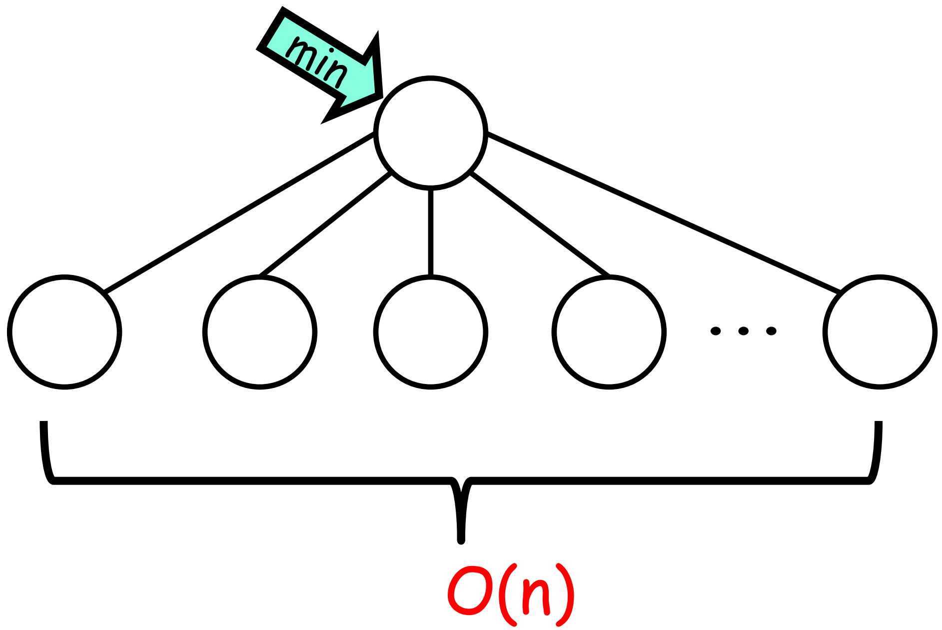
- In a Fibonacci heap we may get almost any tree



The Problem with Cuts:
Wide Shallow Trees

Intuition in a Nutshell

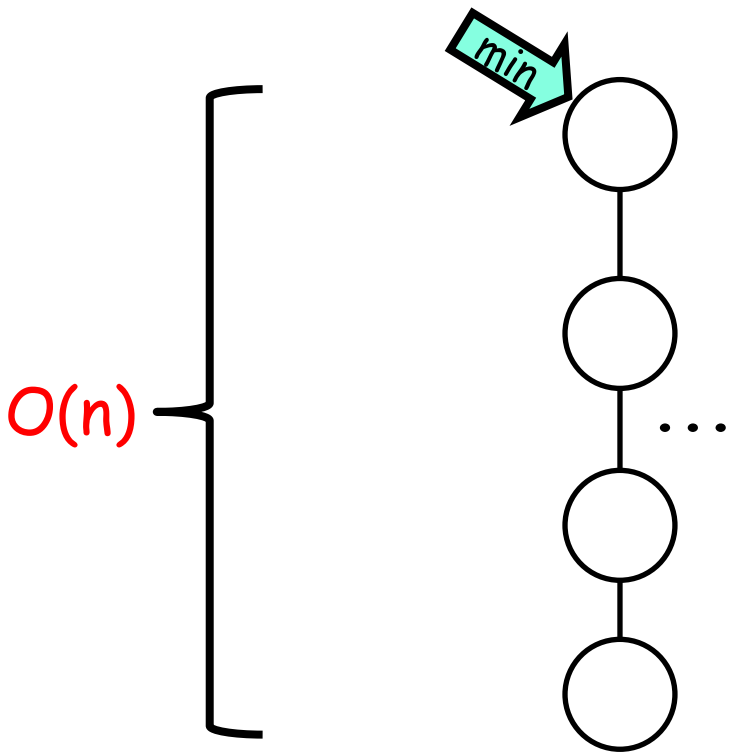
With many cuts, we may get shallow wide trees:



Which operation suffers from this?

Intuition in a Nutshell (2)

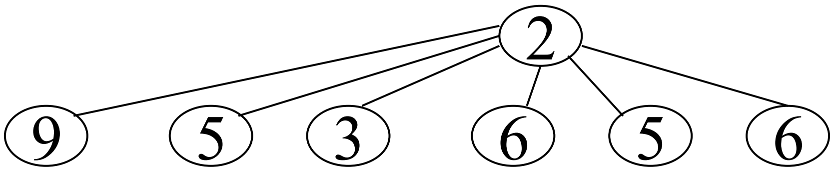
With many cuts, we may also get deep narrow trees:



Currently this is OK since Decrease-key cuts edges

Simple cuts create wide shallow trees

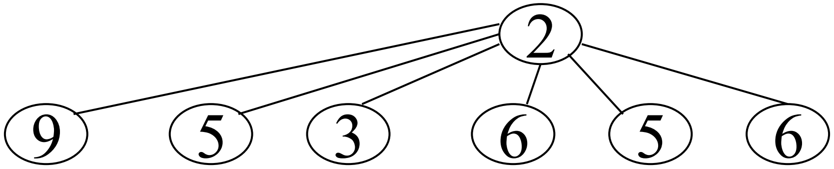
- Recall: a binomial tree of degree k contains 2^k nodes, so all degrees $= O(\log n)$
- However, with cuts, we may get trees of degree k containing as few as $k+1$ nodes, so degrees may be $O(n)$!
- Previous analysis of Del-min breaks down



Shallow wide tree

How to eliminate wide shallow trees

- We don't want a node of degree k to have only $O(k)$ decendants.



Shallow wide tree

- We want it to have $\Omega(c^k)$ decendants, for some constant $c > 1$, so $k = O(\log_c n)$.

Eliminating Wide Shallow Trees: Cascading Cuts (via Decrease-Key)

How to eliminate wide shallow nodes

In Decrease-Key:

- When a node loses 2^{nd} child cut it too and add to root list
  - so a non-root node cannot lose many children without becoming a root itself

- Then we can prove: a node of degree k in a Fibonacci Heap has at least ϕ^k nodes, so $k = O(logn)$ again!

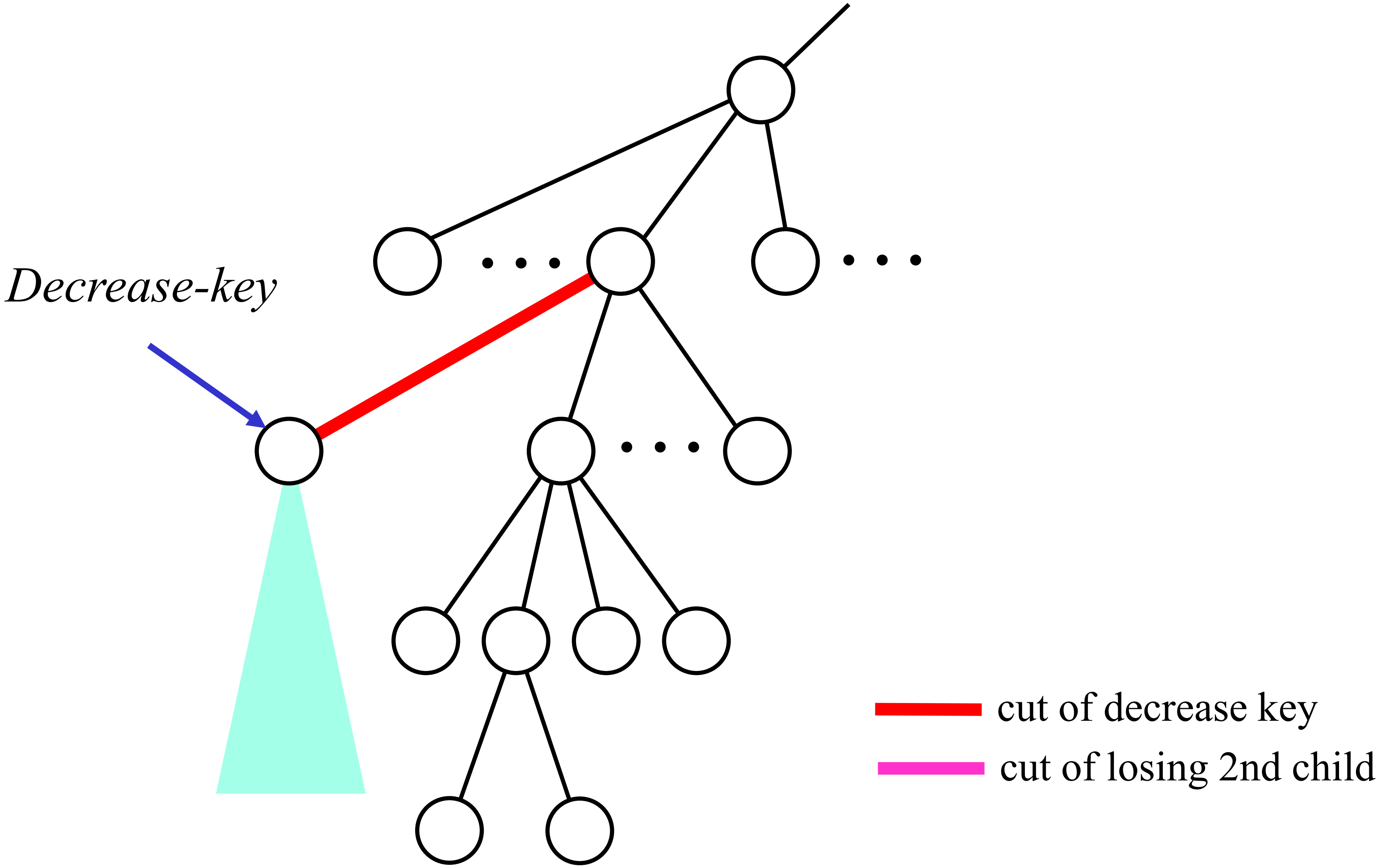
Cascading cuts (eliminate wide shallow nodes)

Desired property:

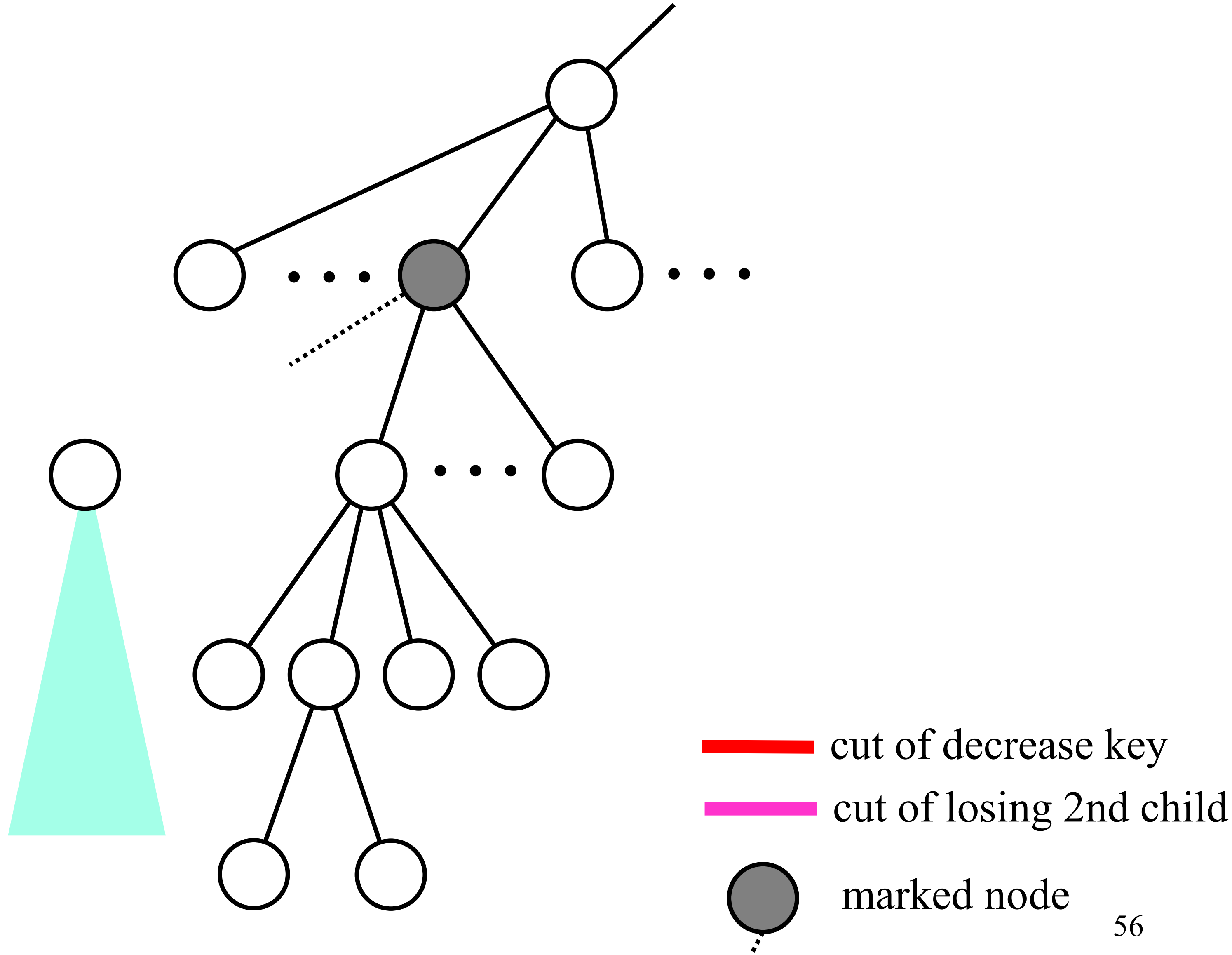
Each node, except root, looses at most one child

- Rule 1: lose 1 child and you are marked LOSER
- Rule 2: lose 2^{nd} child and you're dumped into root list (and unmarked)

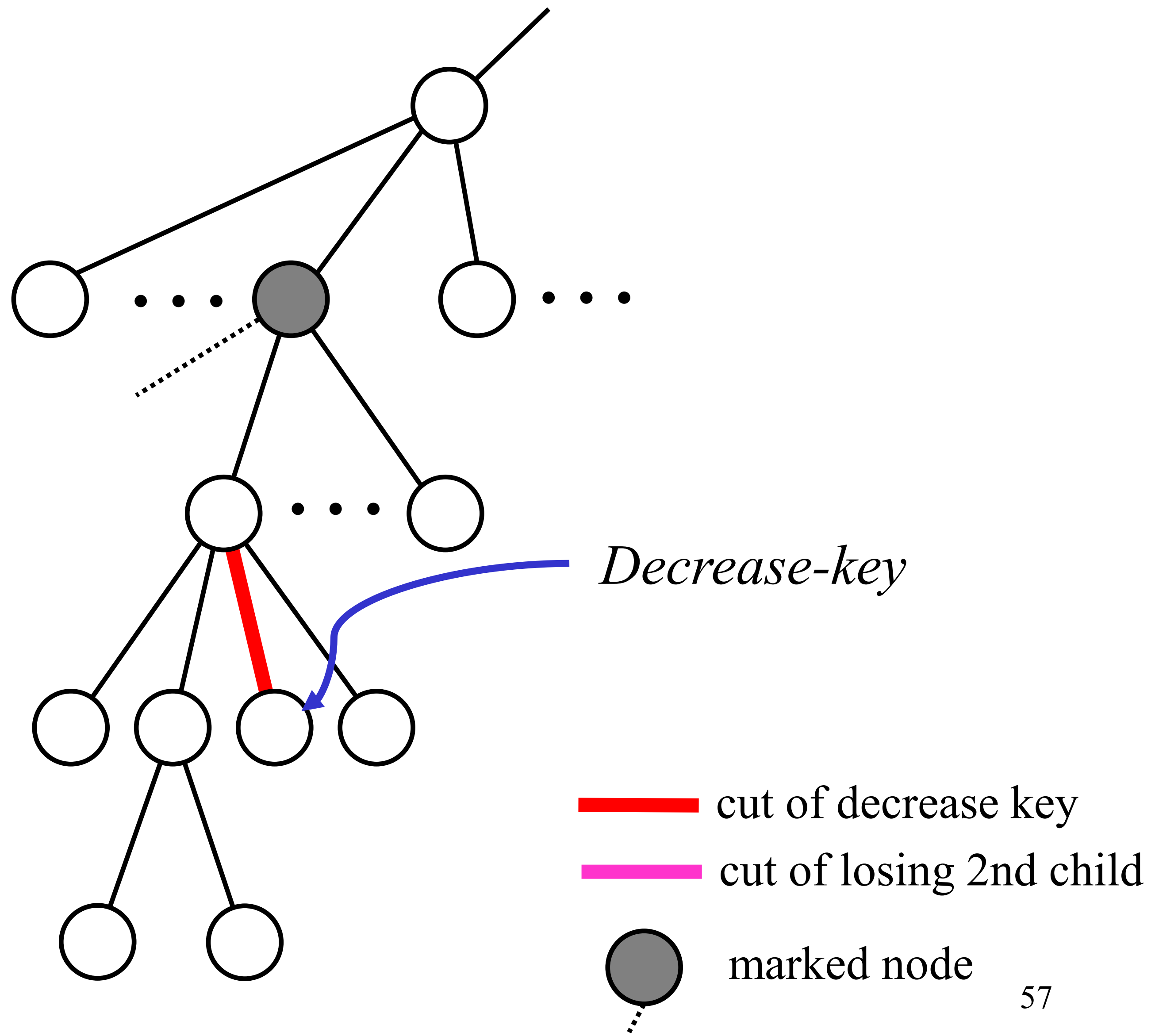
Cascading cuts



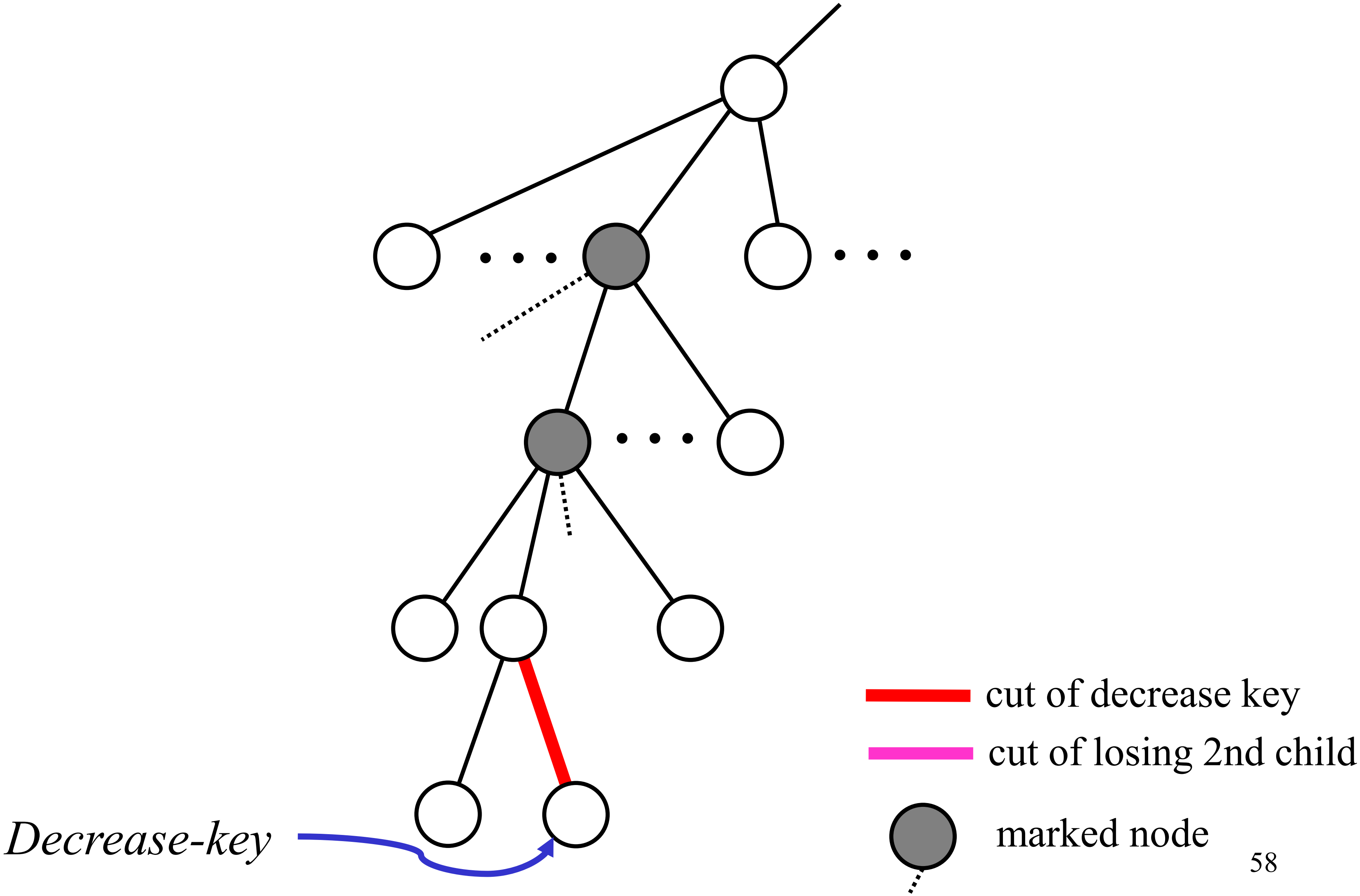
Cascading cuts



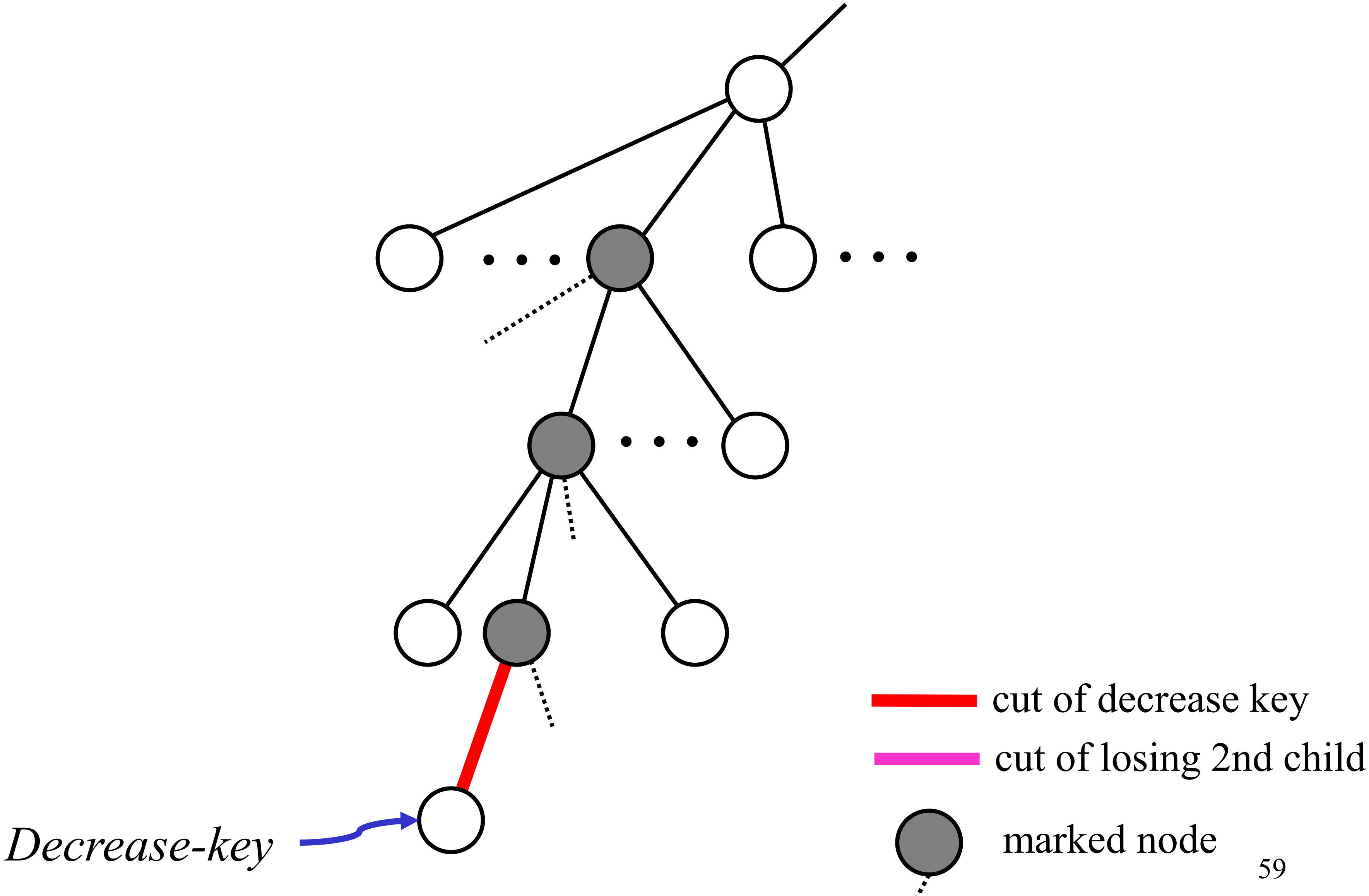
Cascading cuts



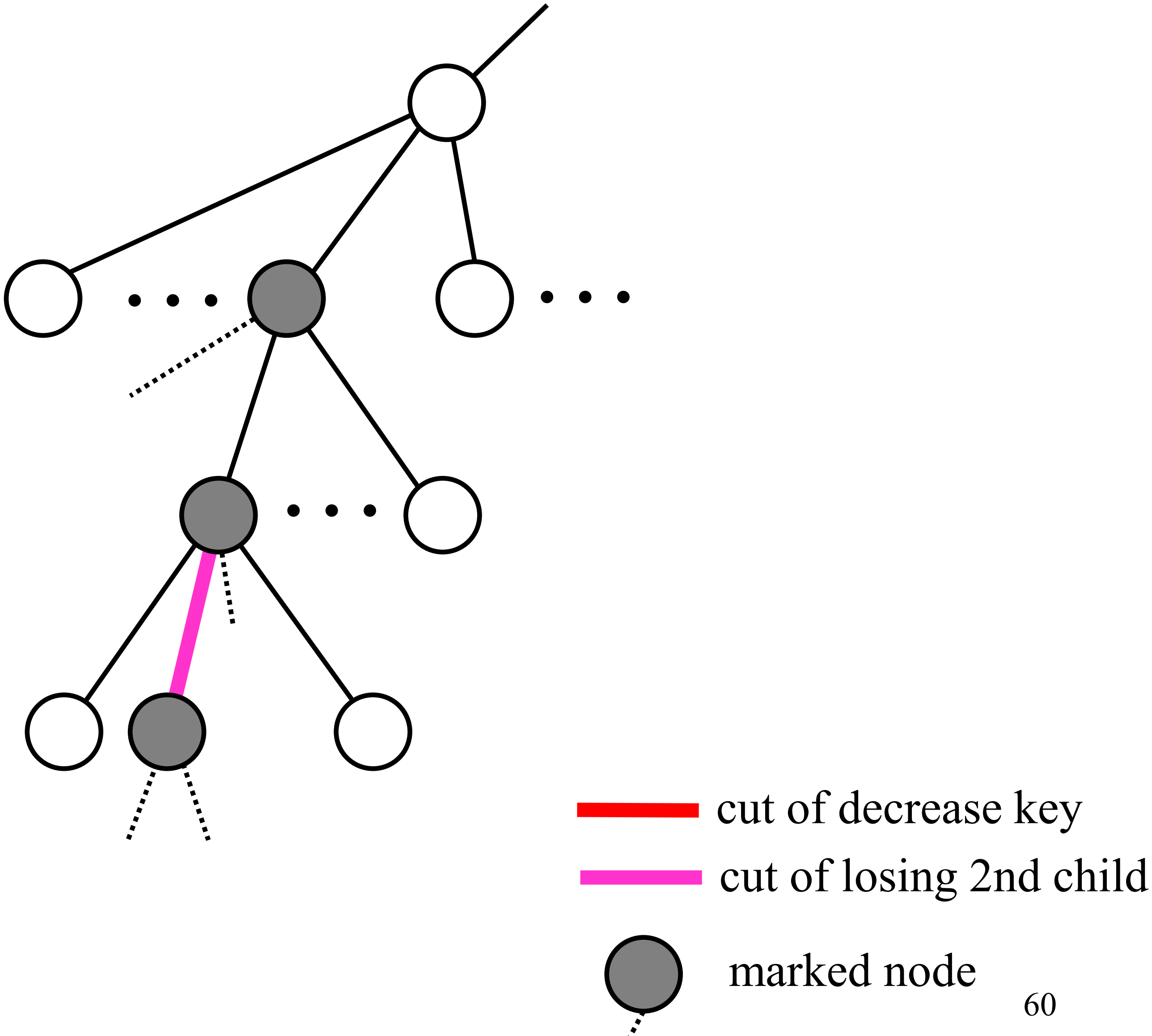
Cascading cuts



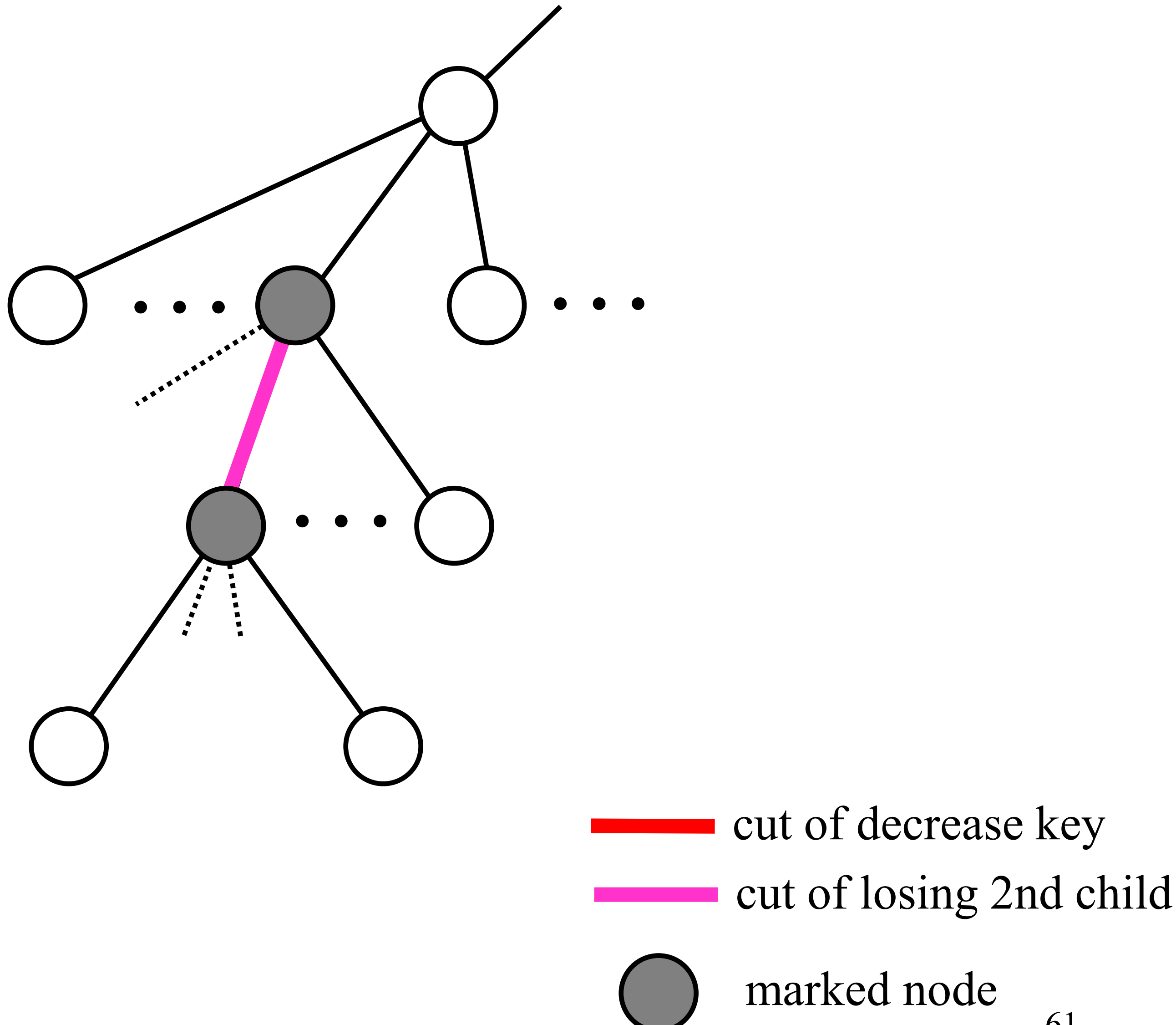
Cascading cuts



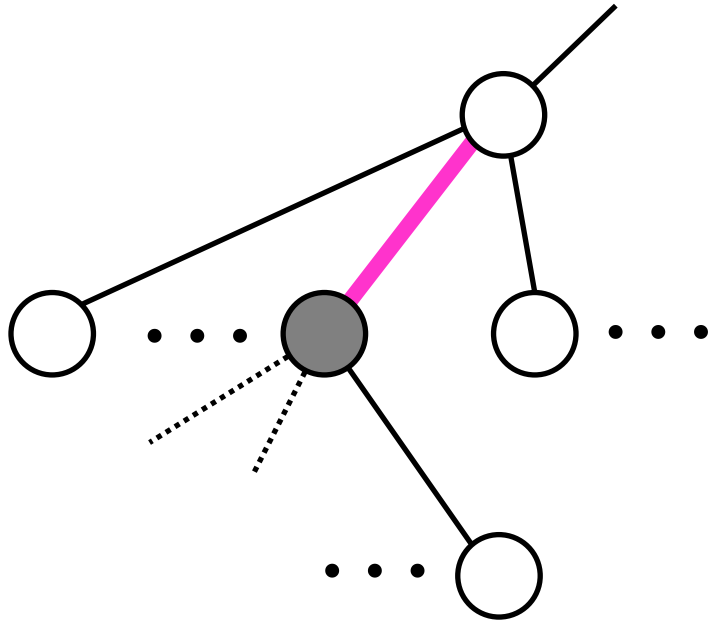
Cascading cuts



Cascading cuts



Cascading cuts

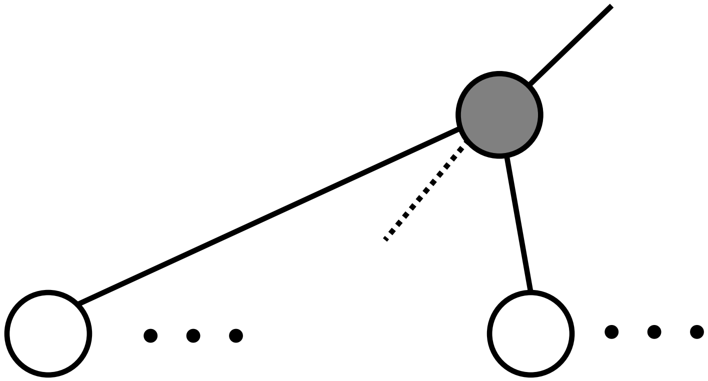


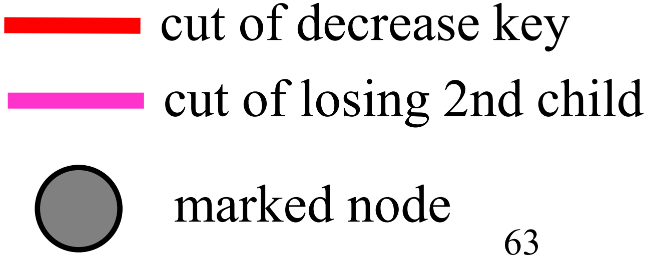
cut of decrease key

cut of losing 2nd child

marked node

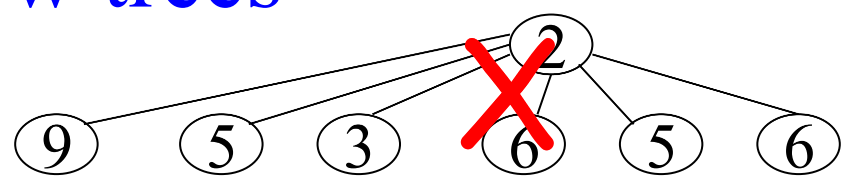
Cascading cuts





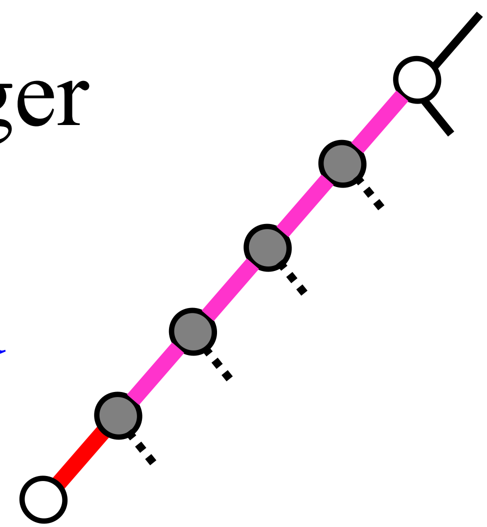
Plan for the Rest of this Lecture

1) Cascading cuts indeed prevent wide shallow trees



Shallow wide tree

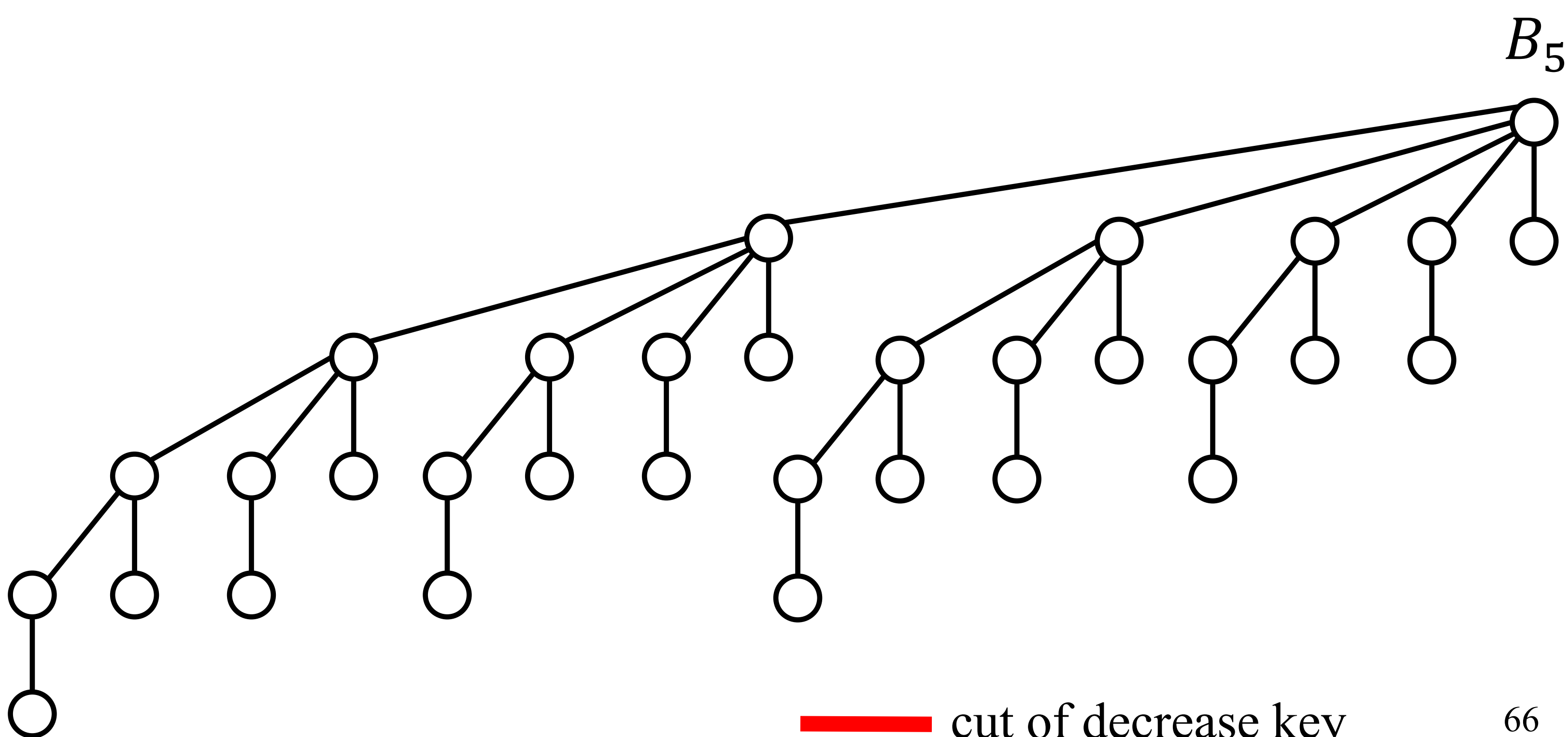
2) Decrease-Key may trigger a cascade of $O(n)$ cuts, but only O(1) amortized



1) Cascading Cuts
Prevent Wide Shallow Trees

Maximally “damaged” trees

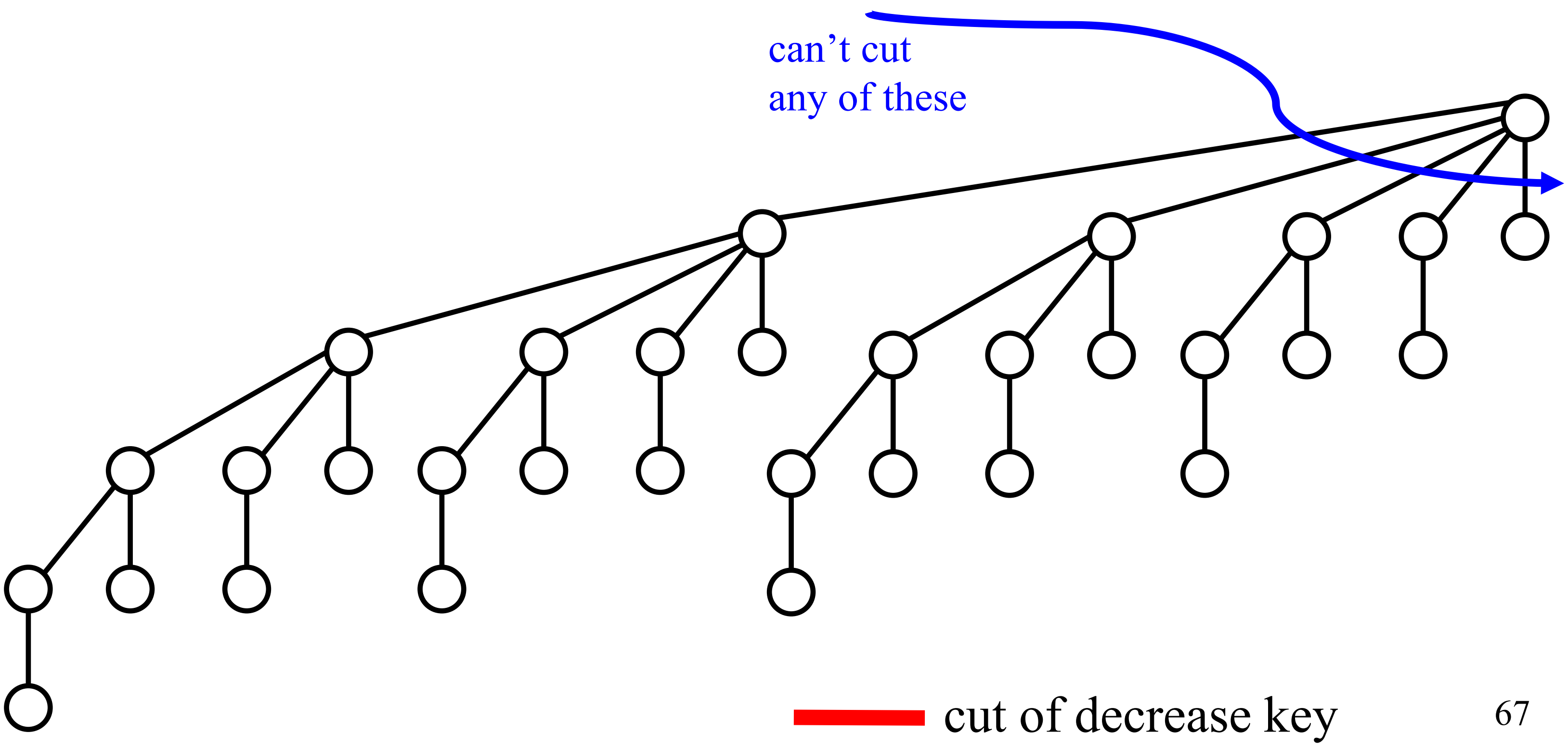
Let’s take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



cut of decrease key

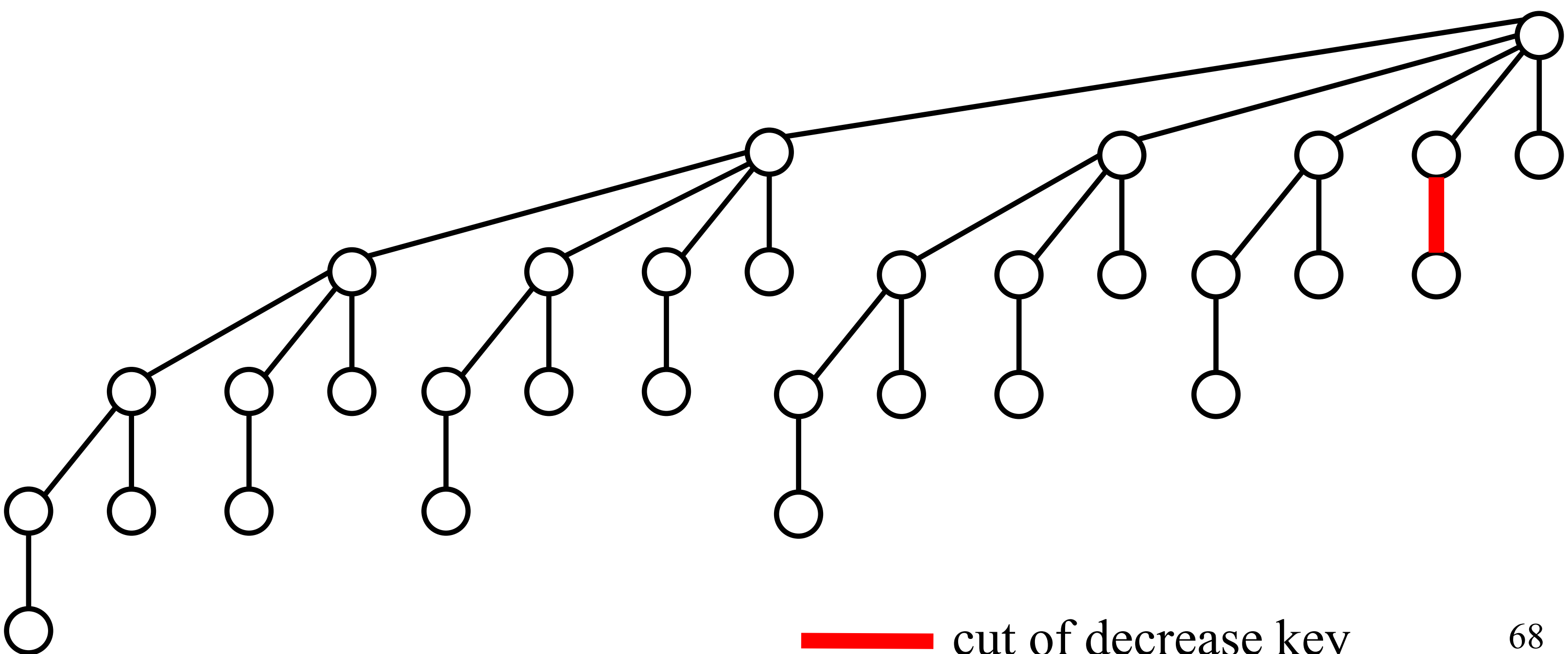
Maximally "damaged" trees

Let's take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



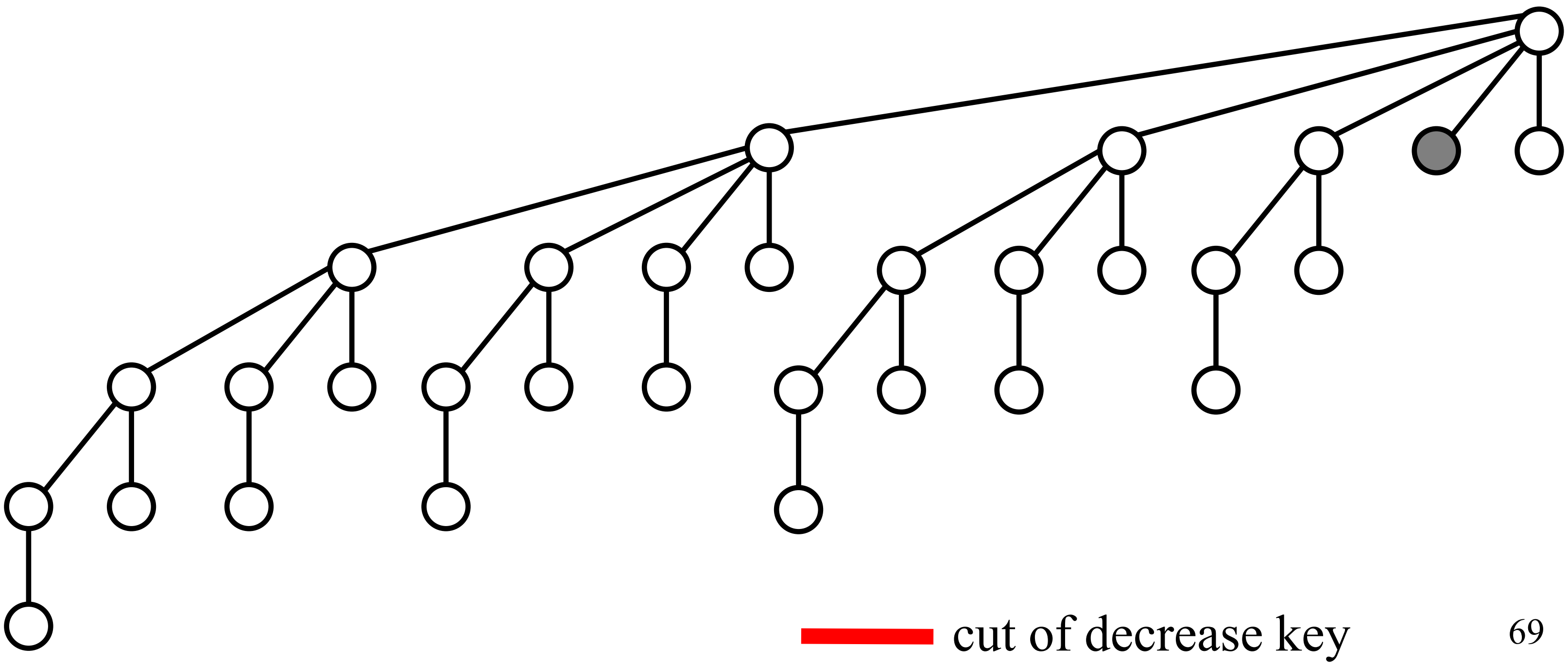
Maximally "damaged" trees

Let's take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



Maximally "damaged" trees

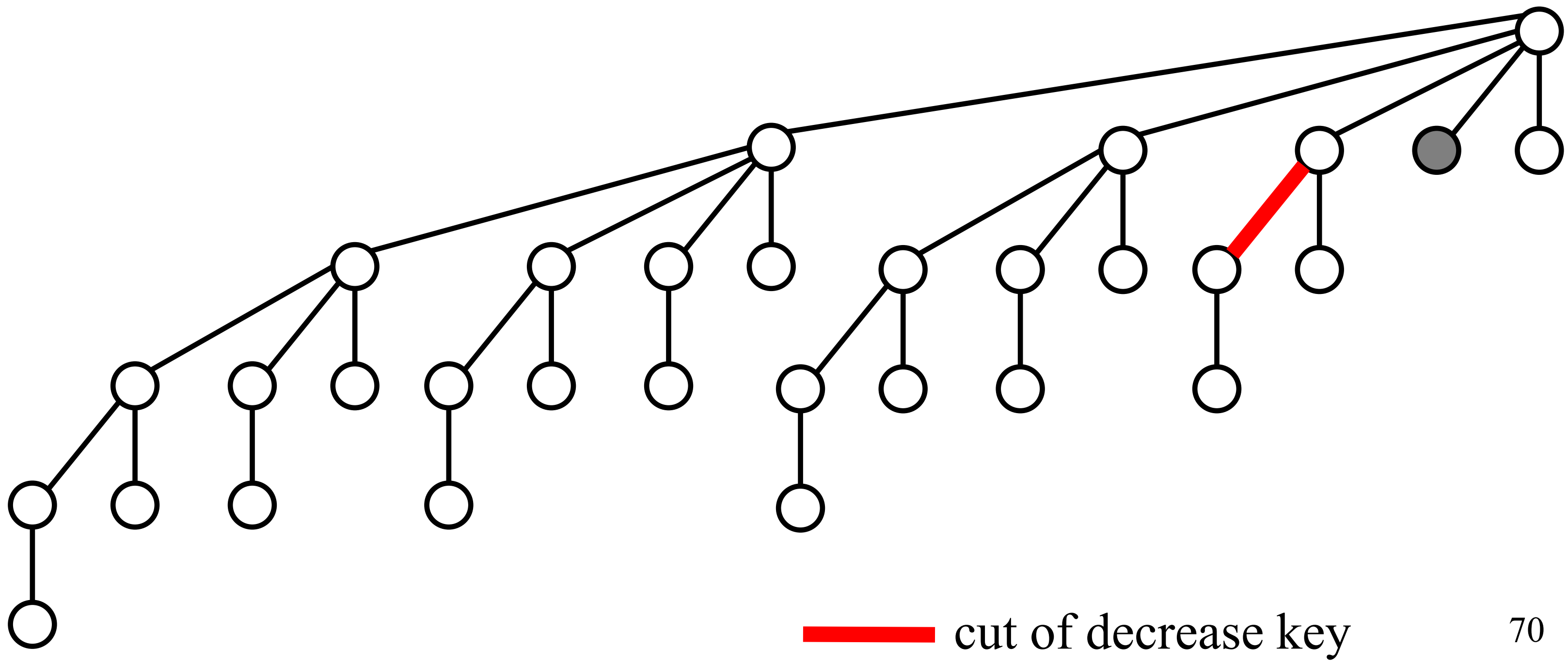
Let's take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



cut of decrease key

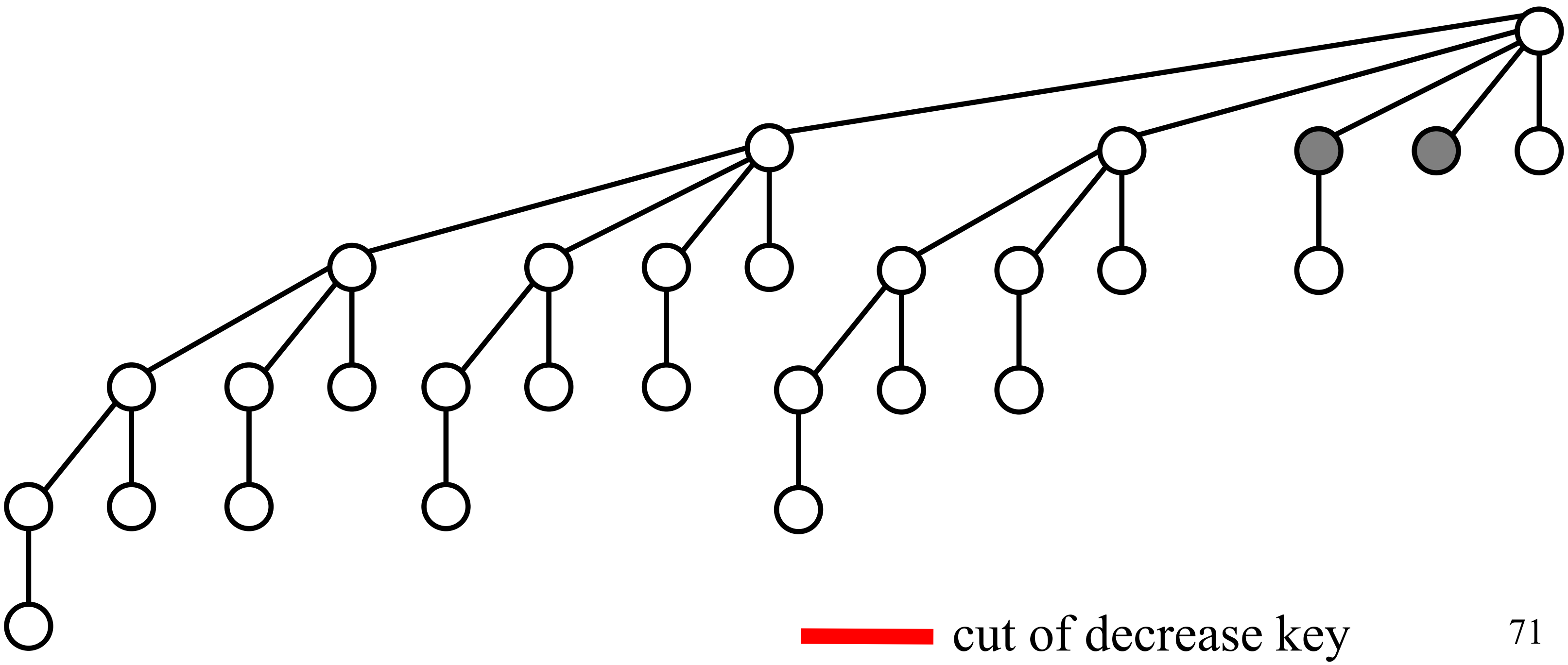
Maximally "damaged" trees

Let's take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



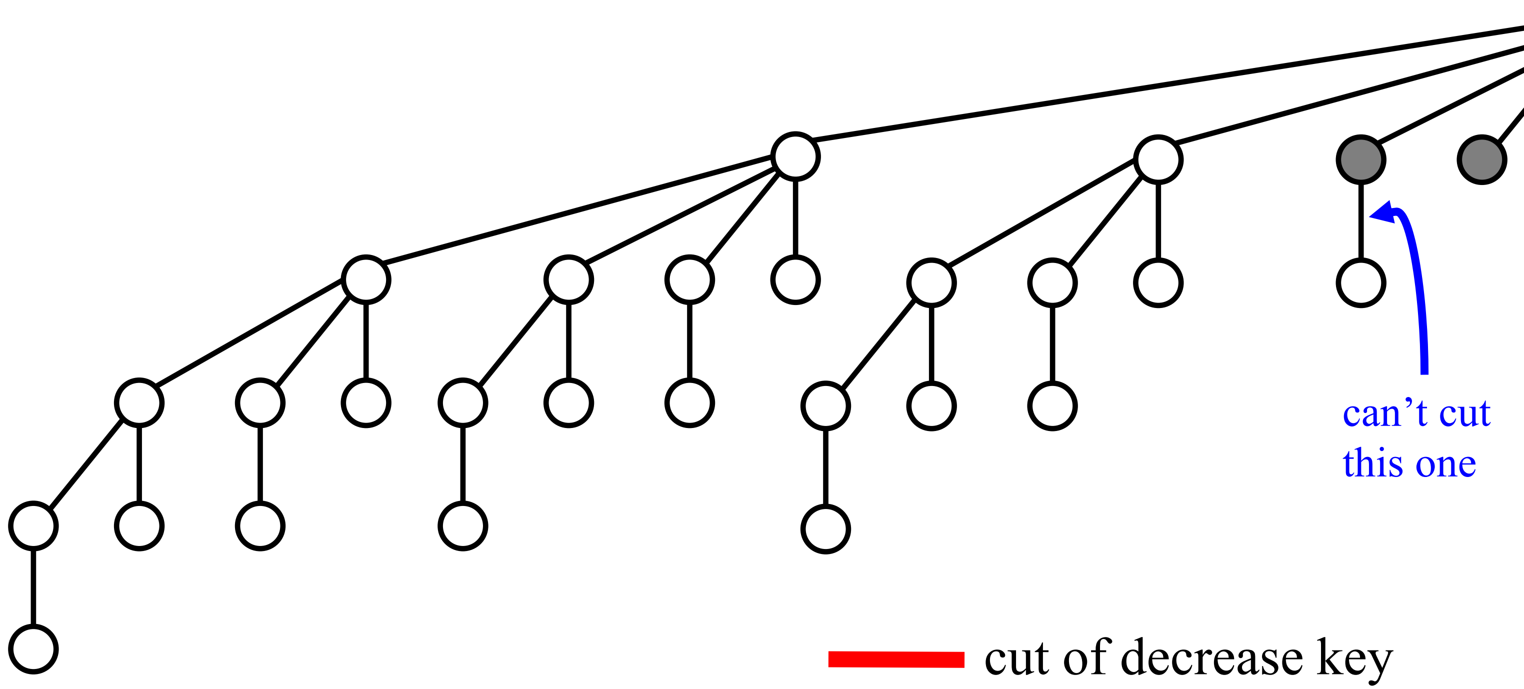
Maximally "damaged" trees

Let's take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



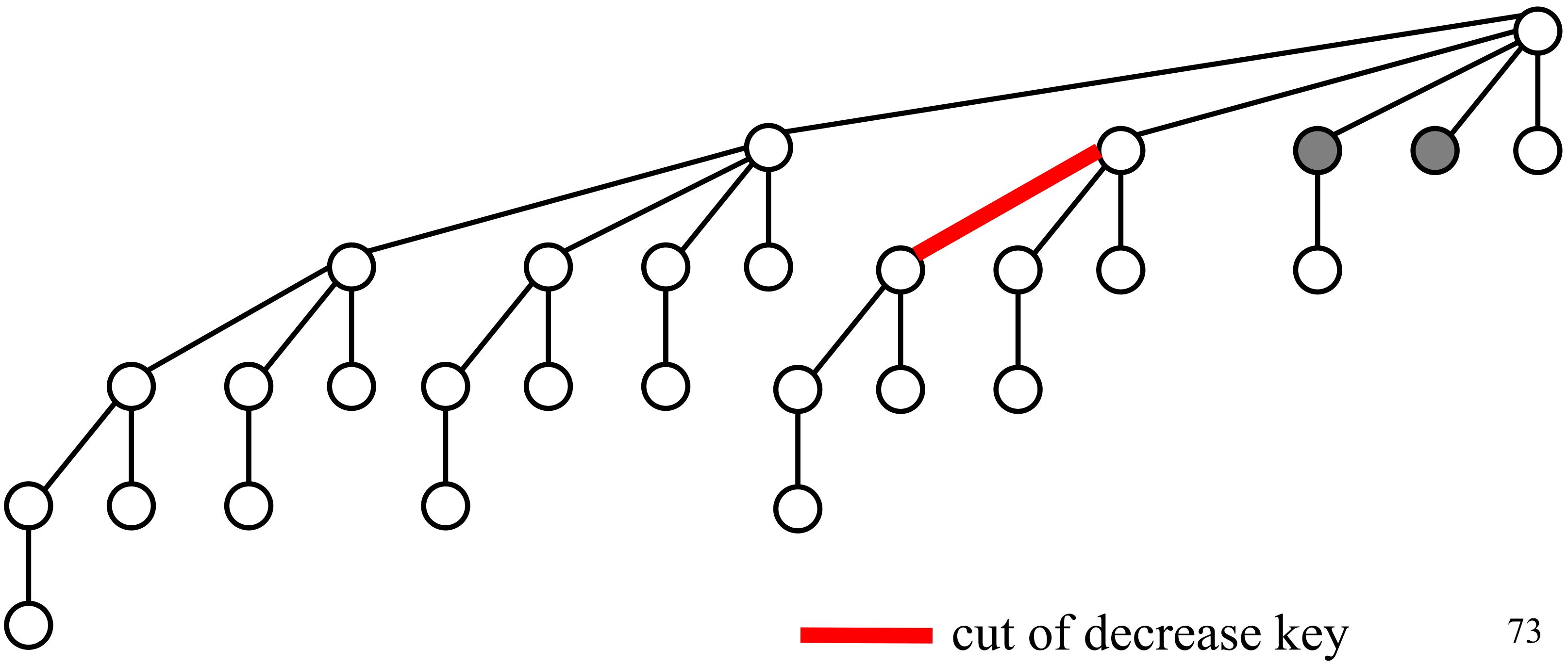
Maximally “damaged” trees

Let’s take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



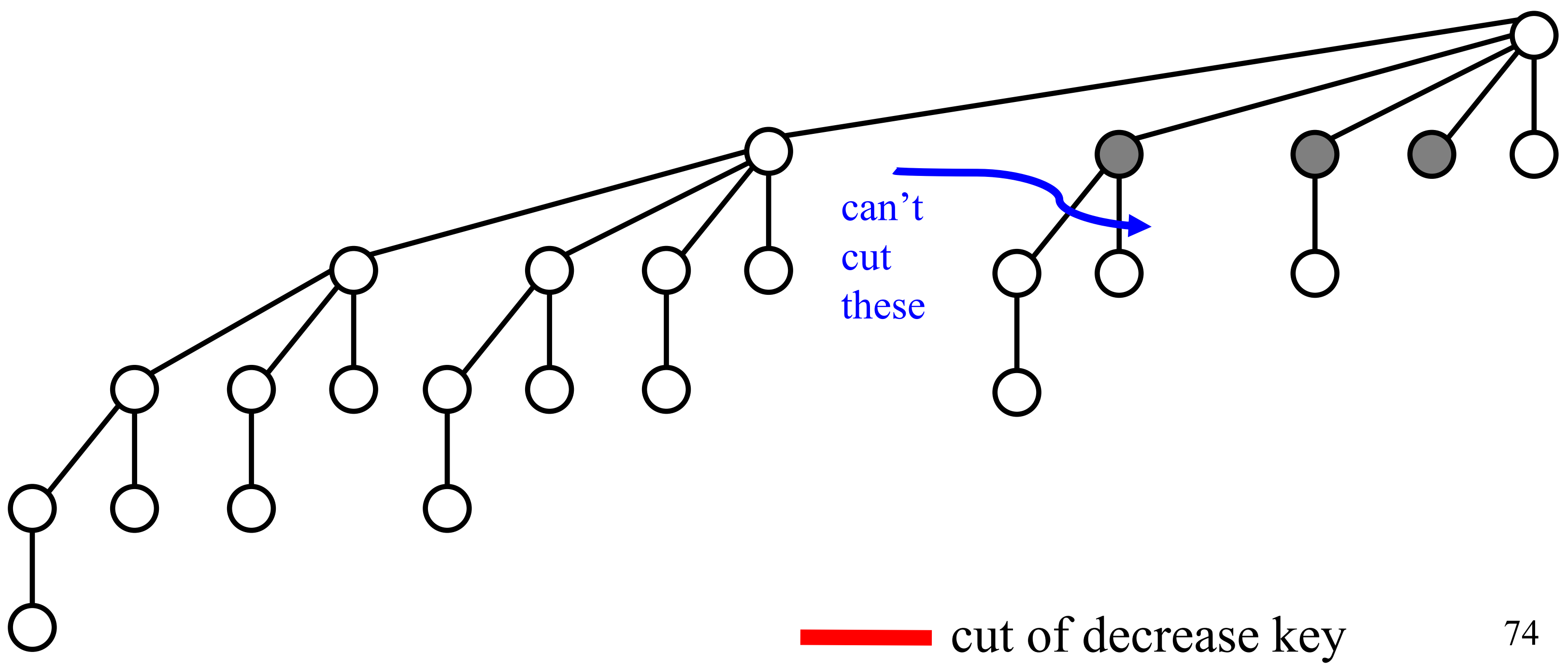
Maximally "damaged" trees

Let's take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



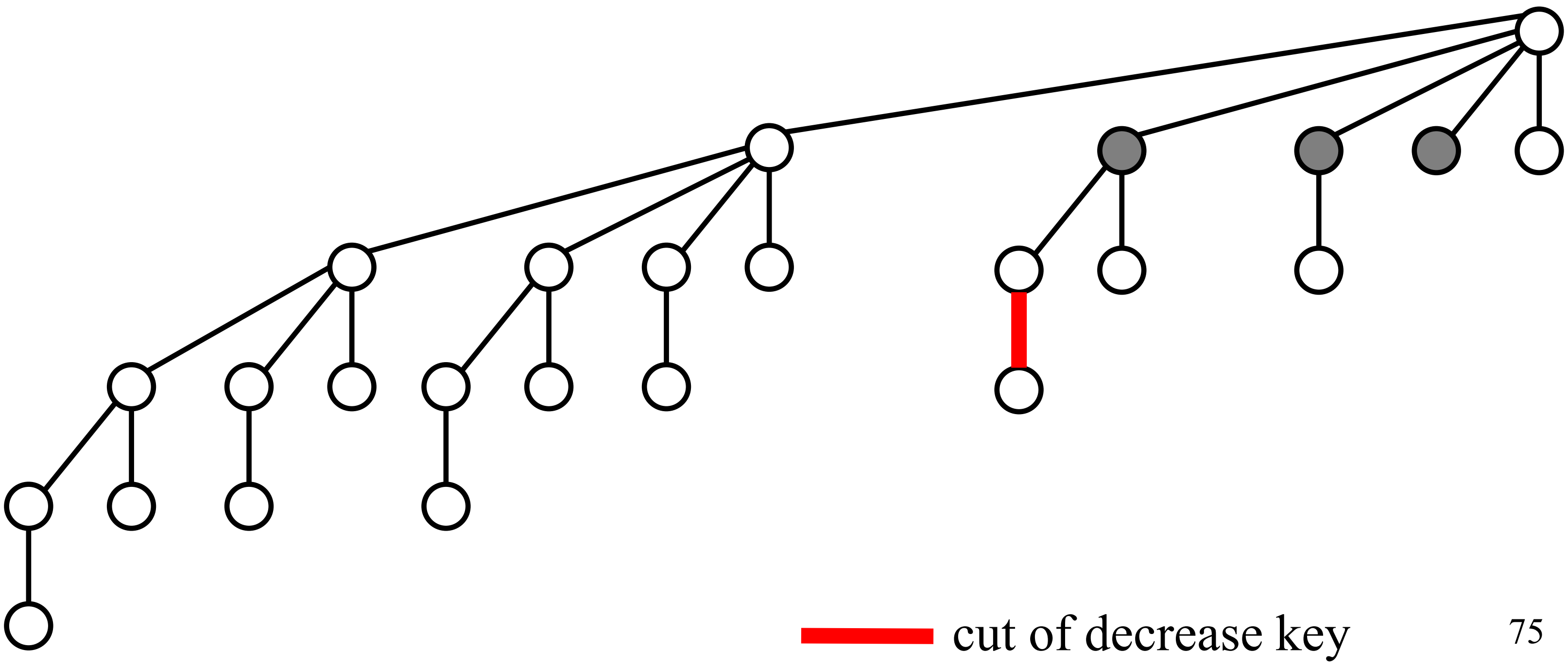
Maximally “damaged” trees

Let’s take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



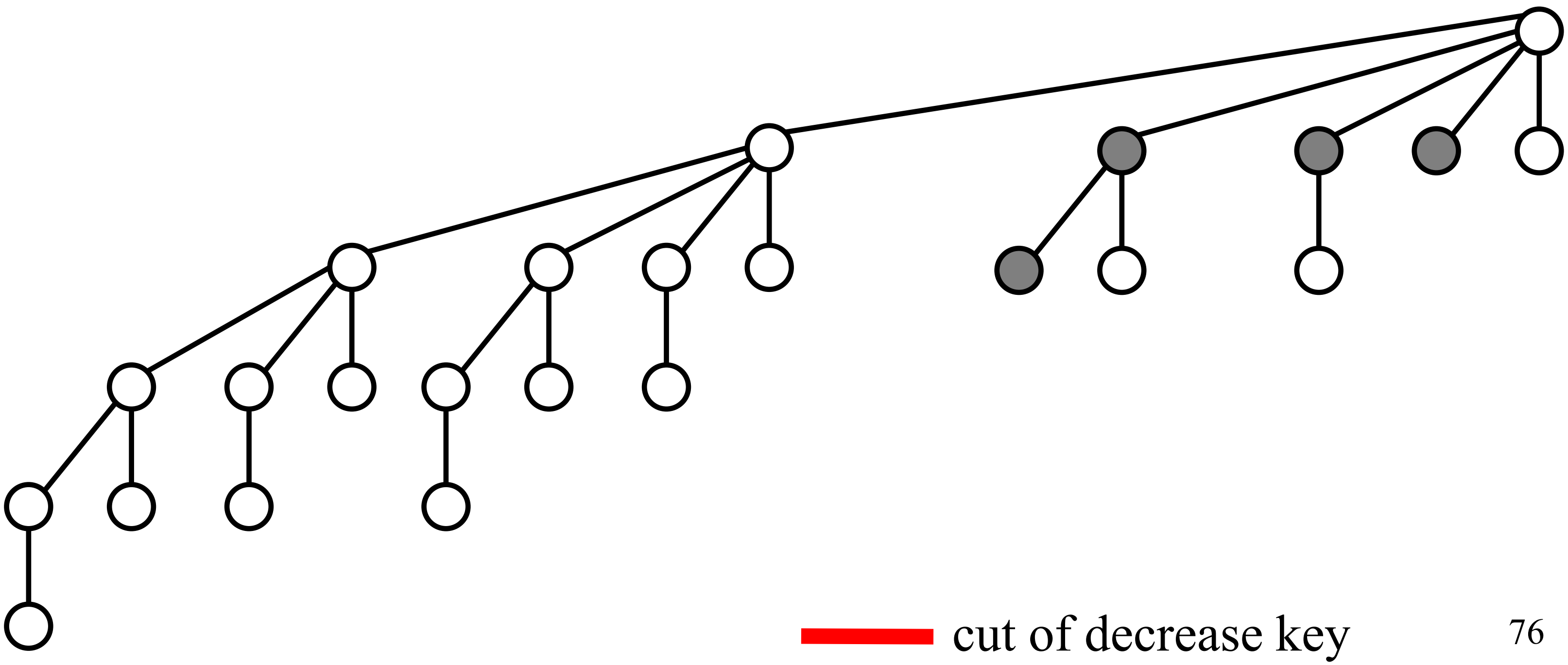
Maximally “damaged” trees

Let’s take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



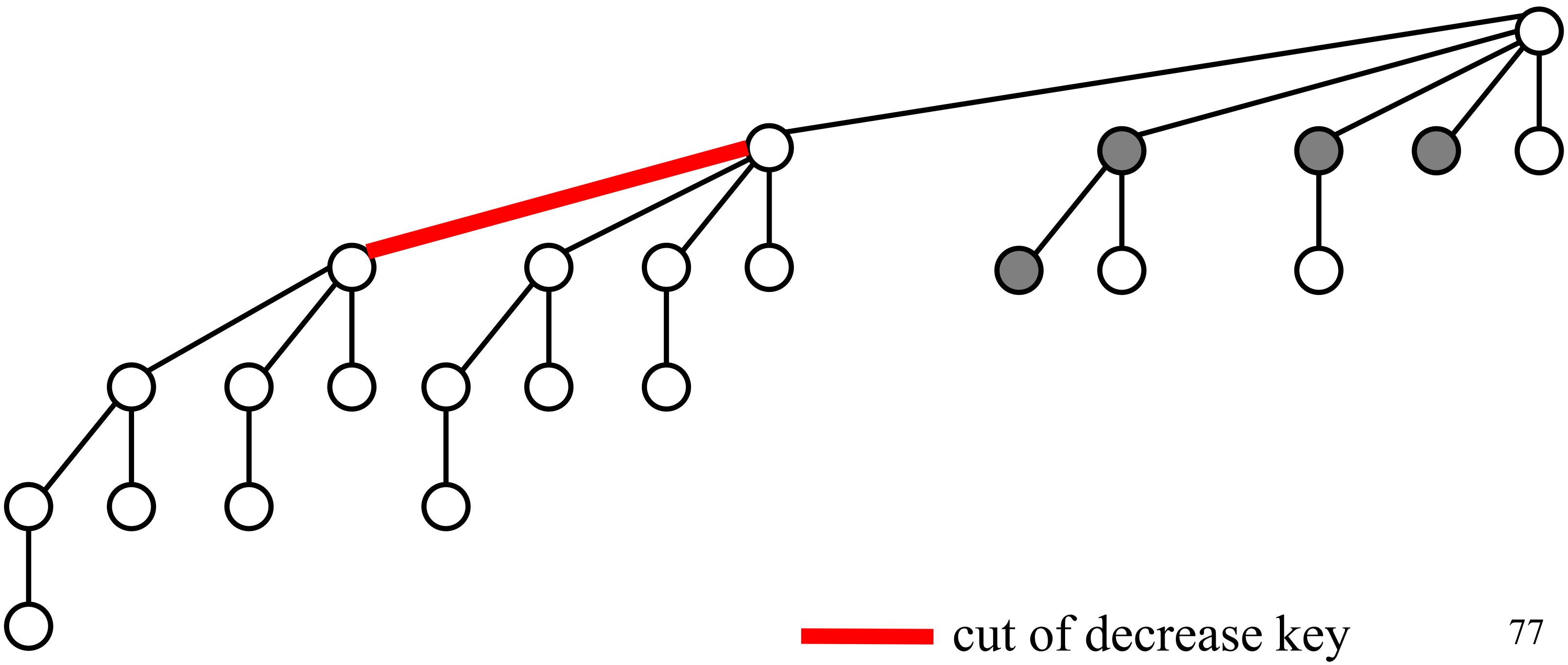
Maximally "damaged" trees

Let's take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



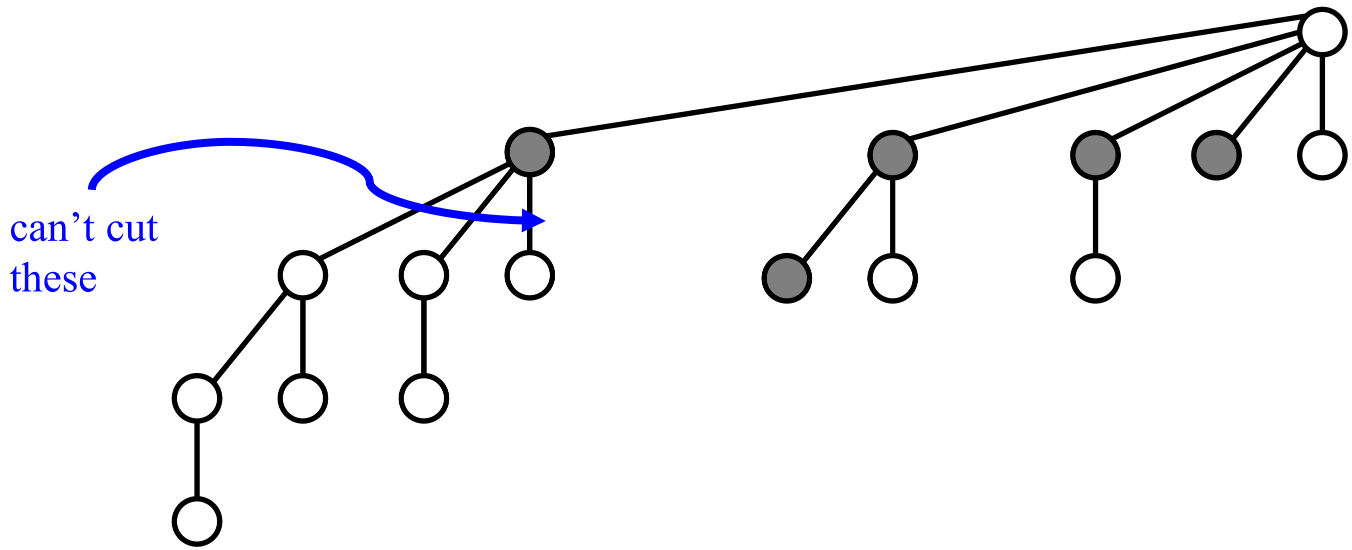
Maximally “damaged” trees

Let’s take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



Maximally “damaged” trees

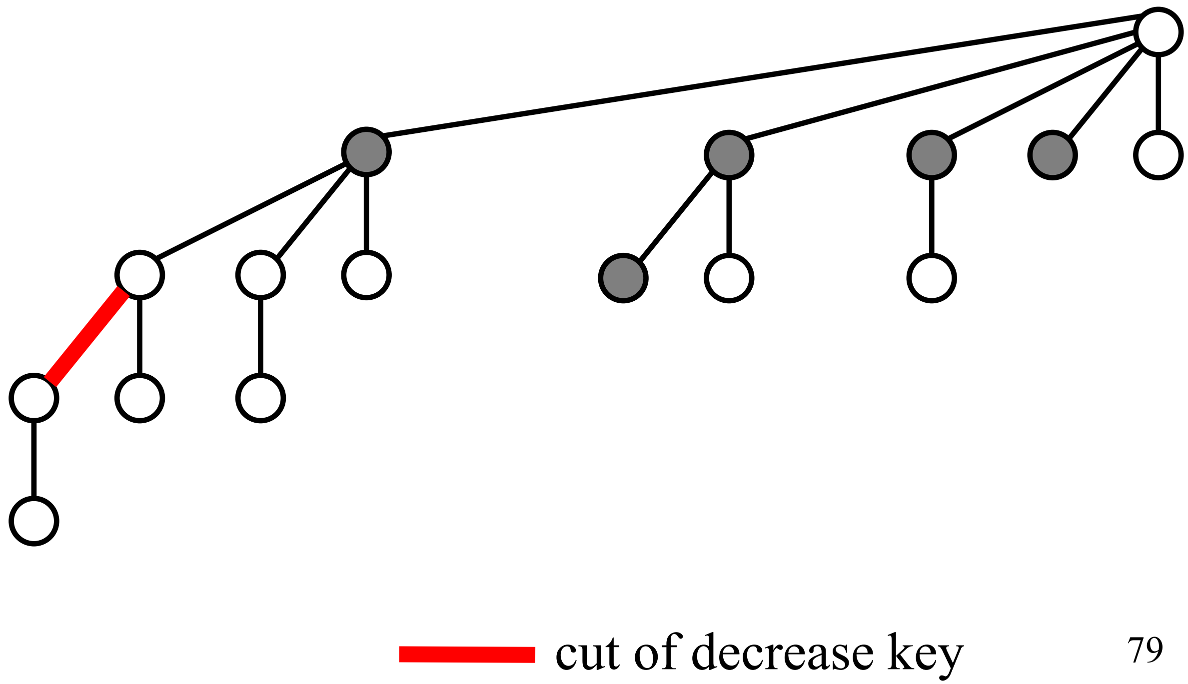
Let’s take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



cut of decrease key

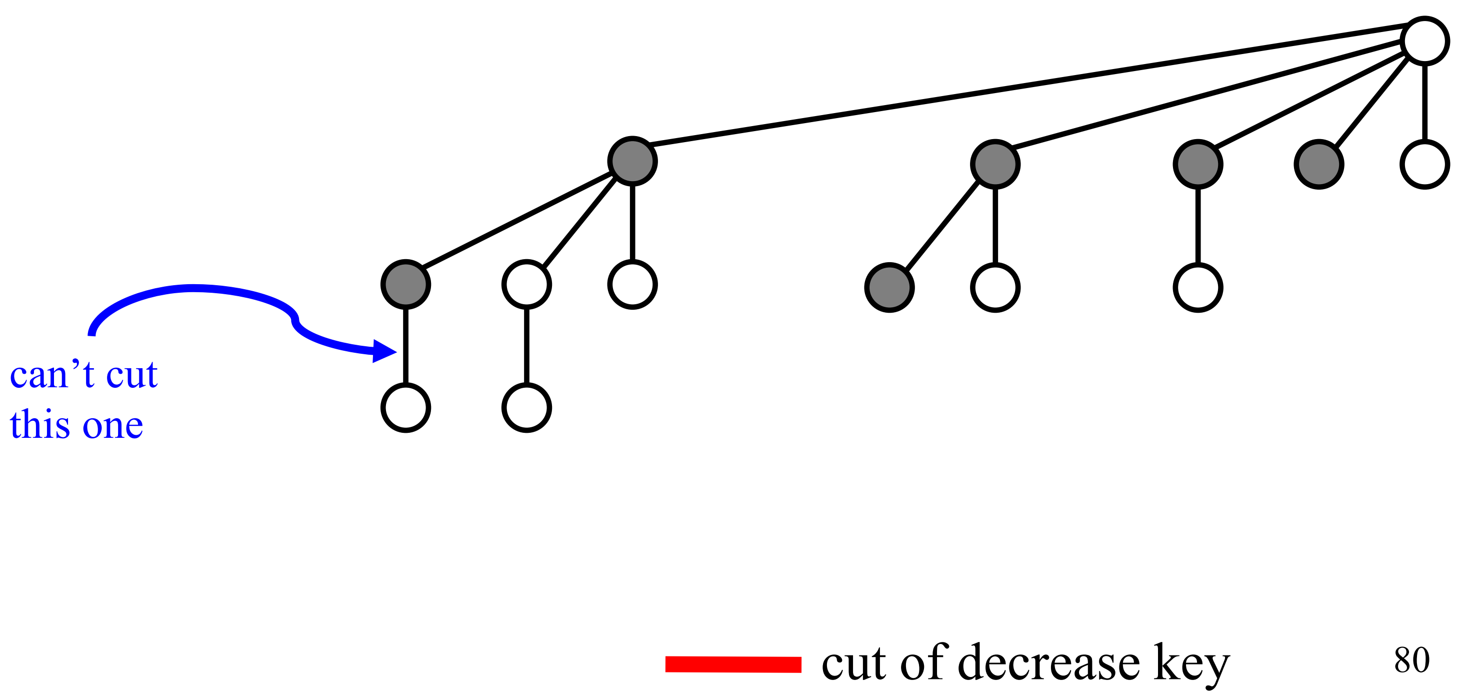
Maximally "damaged" trees

Let's take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



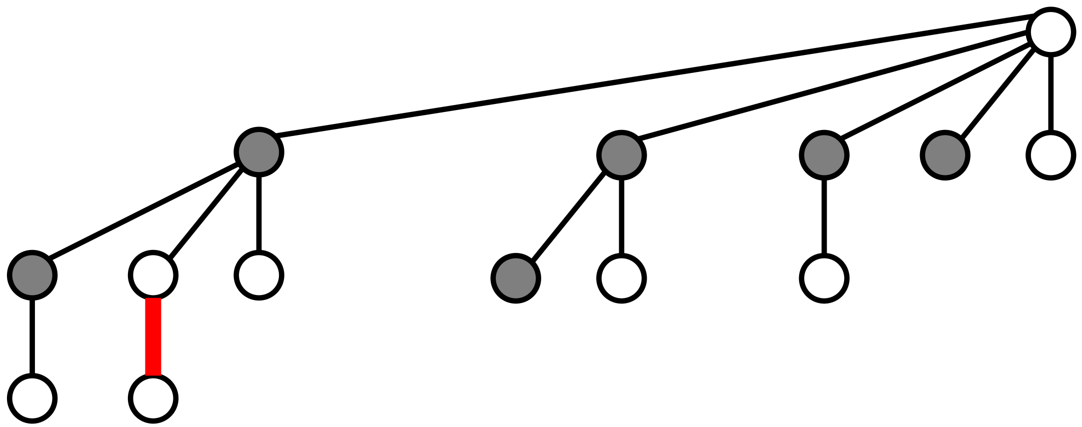
Maximally “damaged” trees

Let’s take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



Maximally "damaged" trees

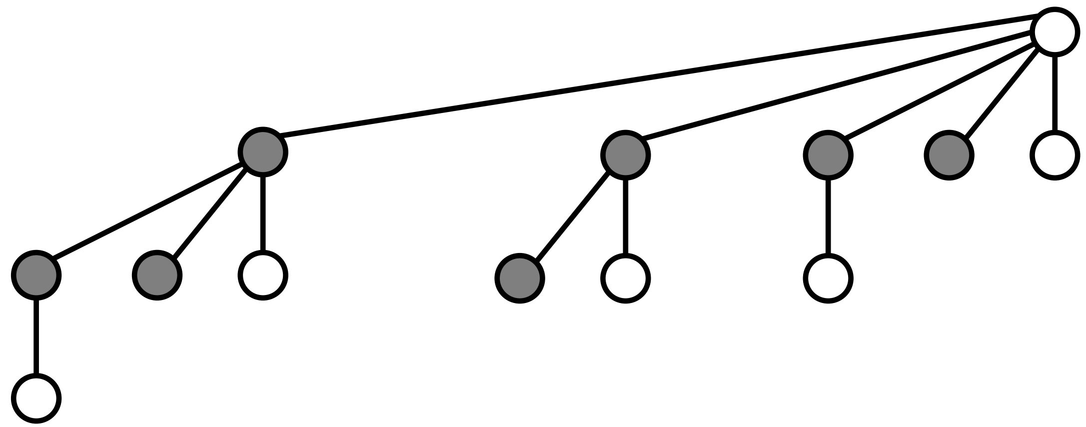
Let's take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



cut of decrease key

Maximally “damaged” trees

Let’s take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible

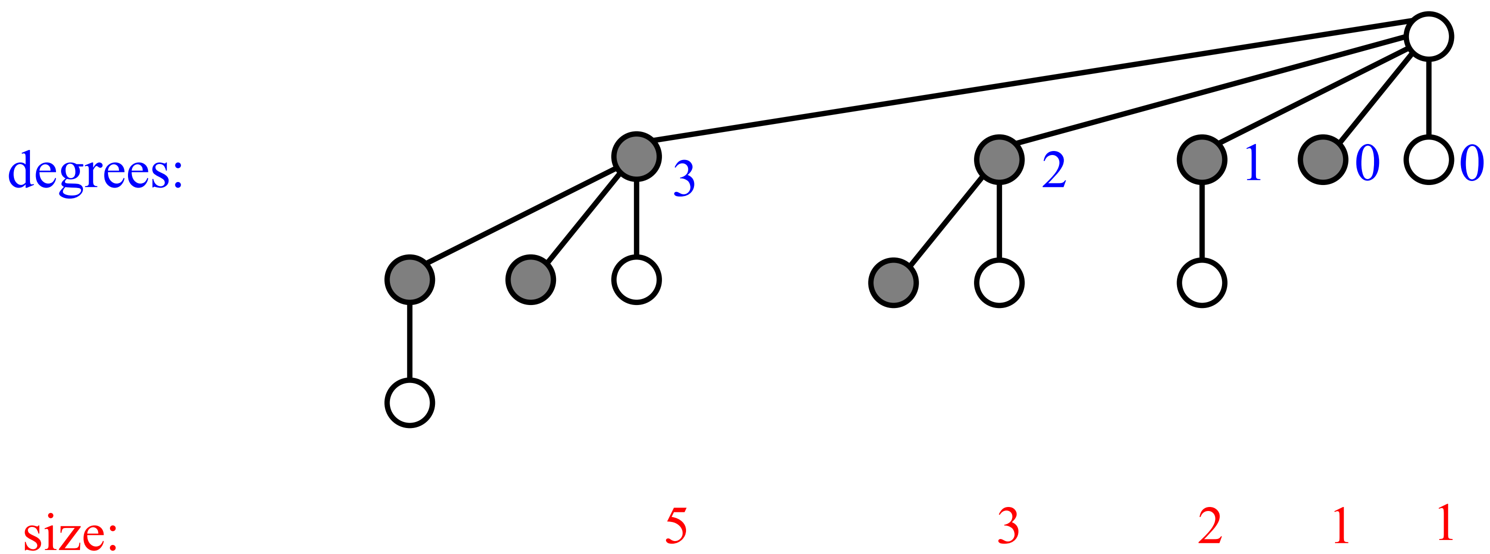


can’t cut anymore!

cut of decrease key

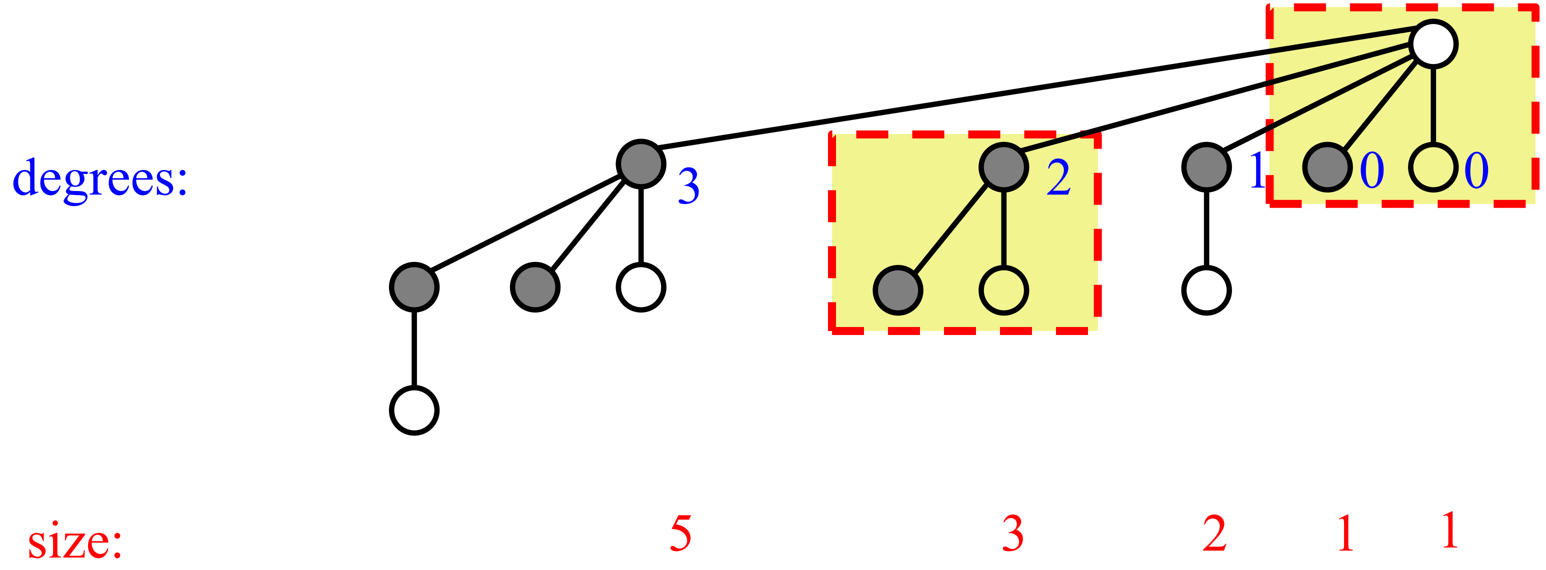
Maximally "damaged" trees

Let's take a binomial tree of degree $k = 5$ and make it lose as many descendants as possible



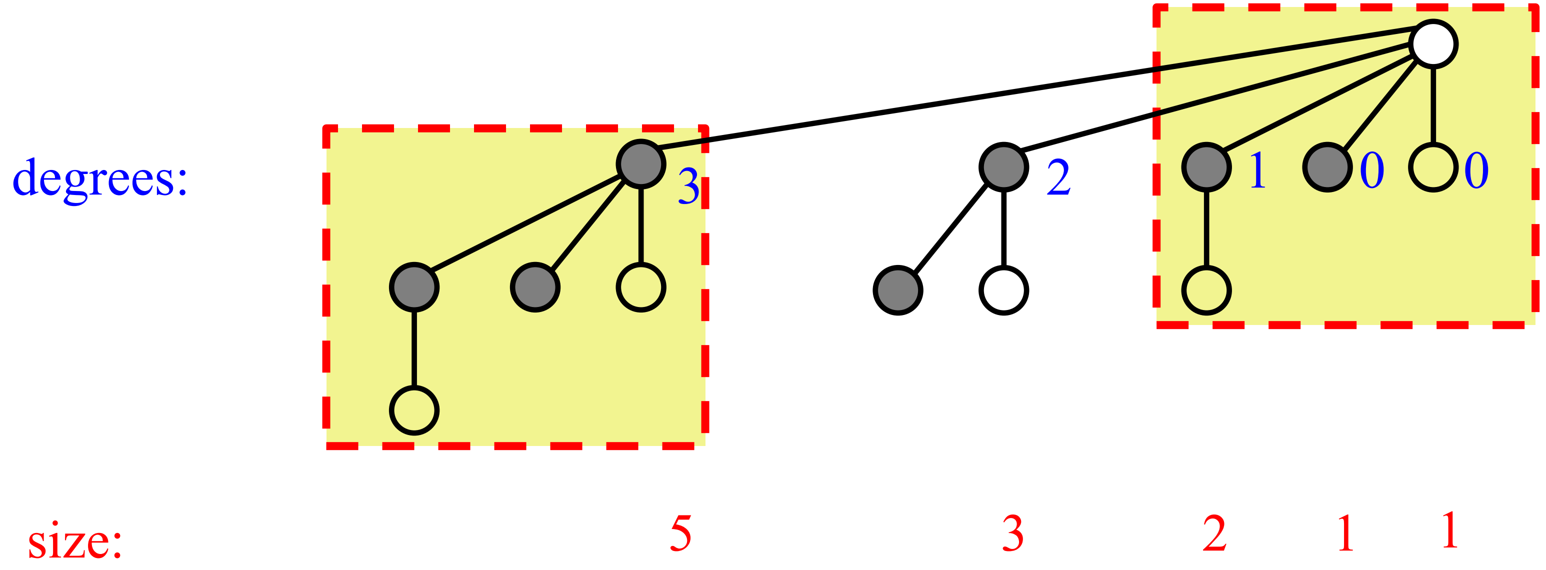
Maximally "damaged" trees

Note the recursive structure



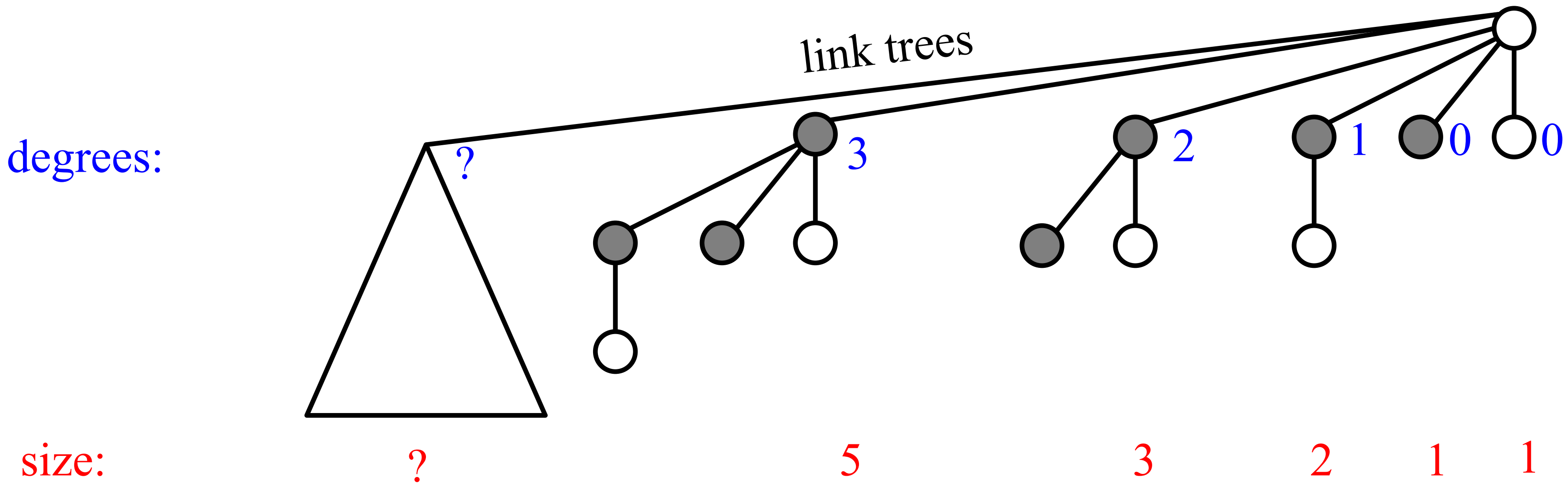
Maximally "damaged" trees

Note the recursive structure



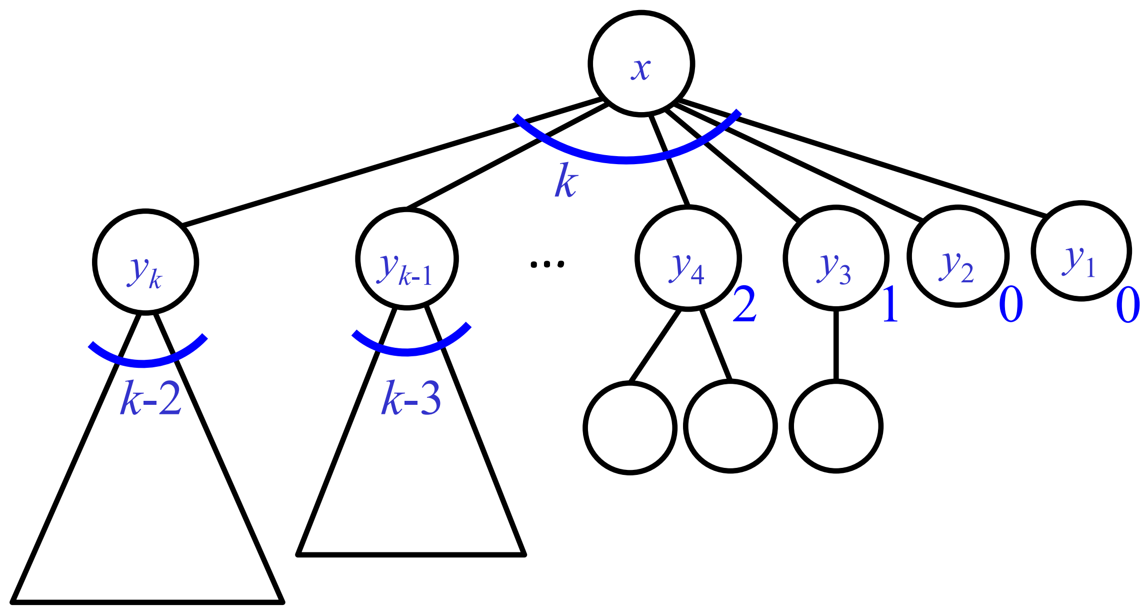
Maximally “damaged” trees

What if we linked another tree
(of same degree $k = 5$),
and cut as much as possible?



Degrees in Maximally "damaged" trees

Lemma 1: Let x be a node of degree 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 degree of y_i is at least $i–2$.



Degrees in Maximally "damaged" trees

Lemma 1: Let x be a node of degree 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 degree 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 degree of x was $i-1$. This was also the degree of y_i (why?) As y_i is still a child of x, it lost at most one child.

Size of Maximally "damaged" trees

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

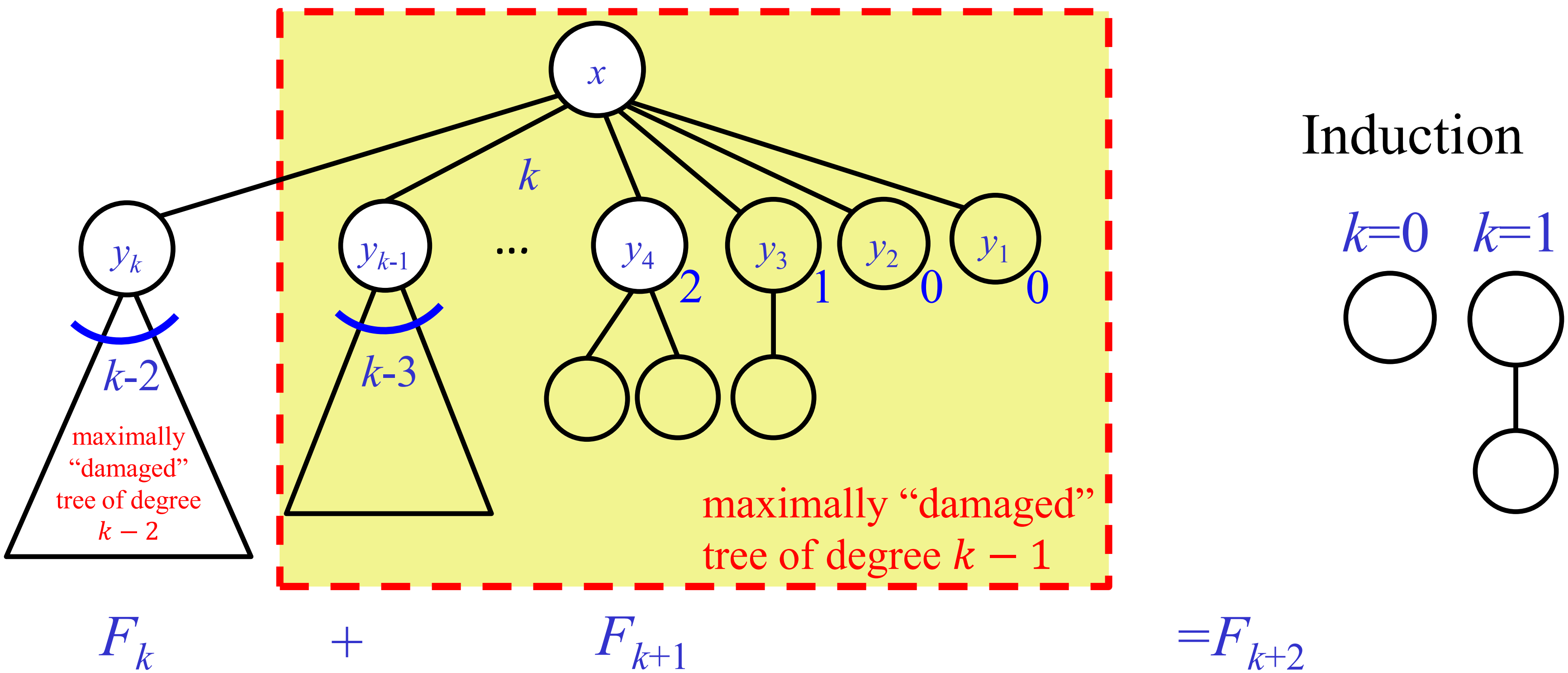
$$F_0 = 0 \quad F_1 = 1$$
$$F_k = F_{k-1} + F_{k-2}\ , \ k \geq 2$$

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

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

Size of Maximally “damaged” trees

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



Size of Maximally “damaged” trees

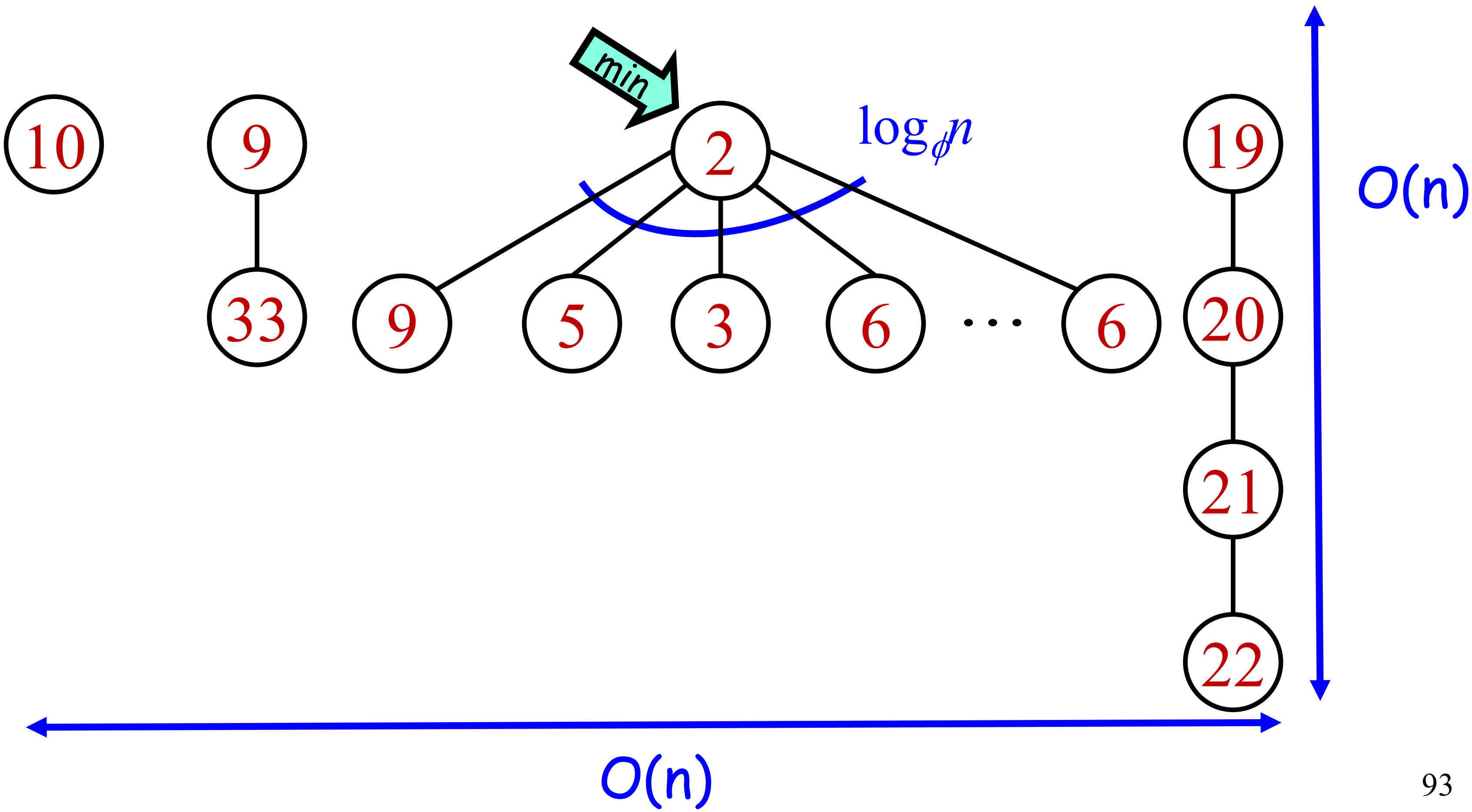
Lemma 2: A node of degree k in a Fibonacci Heap has at least $F_{k+2} \geq \phi^k$ descendants, including itself.

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

Degrees are again $O(\log n)$

Fibonacci Heap - Intuition

- In a Fibonacci heap a tree cannot be too wide!



2) Decrease-key is O(1) amortized

Amortized Cost of Decrease-key

- A decrease-key operation may trigger $O(n)$ cuts.

- However, we want to prove the cost of Decrease-key is $O(1)$ amortized
  - Who can pay for all these cuts?
  - How should Φ be defined?

Potential – first try

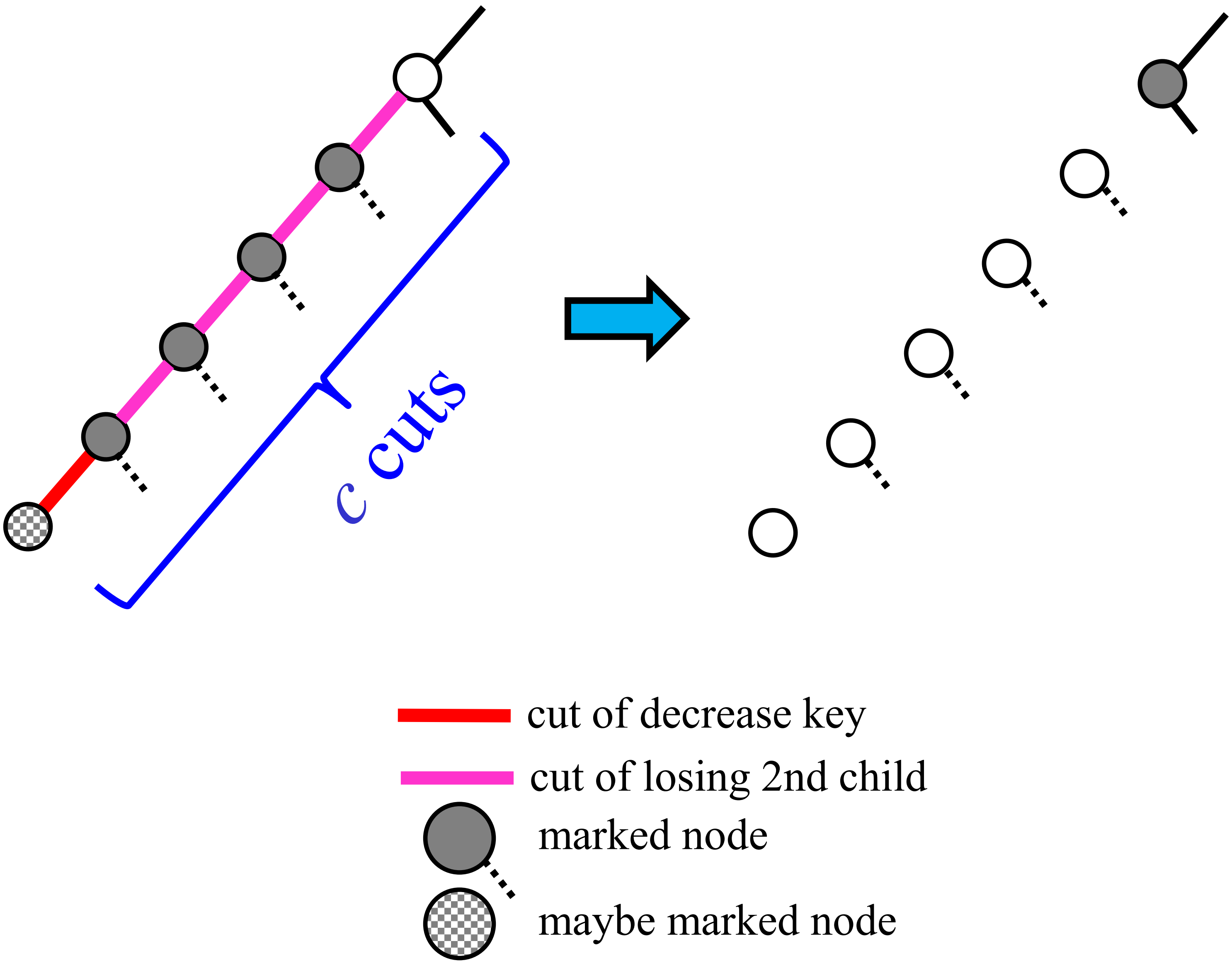
$$\Phi = \#\text{trees} + \#\text{marked nodes}$$

We need this to pay for Delete-Min, remember?

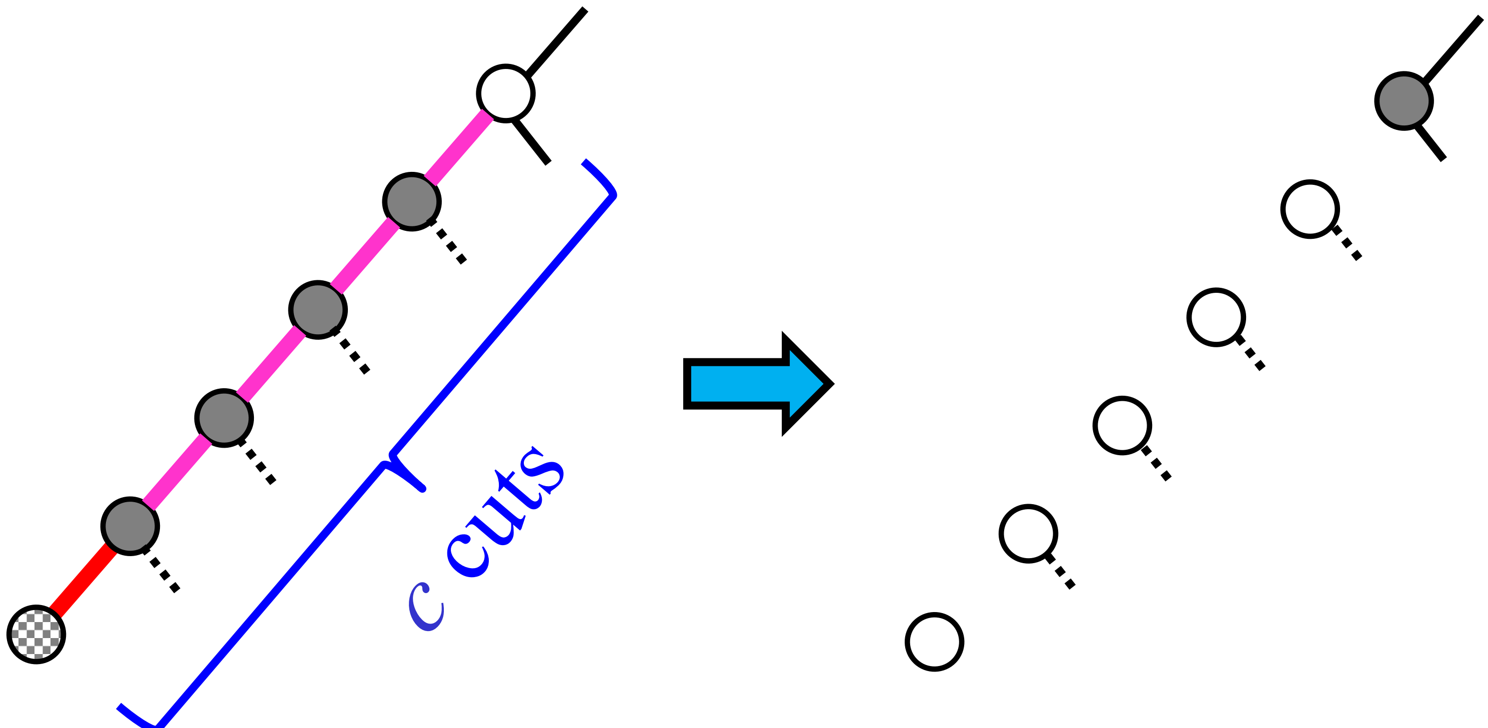
So Decrease-key will pay for itself

- We currently have 2 sources to potential change. We have to see if previous amortized analysis has changed

Amortized Cost of Decrease-key



Amortized Cost of Decrease-key



$\underline{\Delta\Phi}$

- c new trees $\quad +c$
- ≤ 1 new mark (last cut), and
  c or $c-1$ marks removed $\quad \leq 2-c$

Amortized Cost of Decrease-key

	Actual cost	**potential: Δ Trees**	**Potential: ΔMarks**	**Amortized cost**
Decrease-key	c	$+c$	$\leq 2-c$	$O(c) = O(n)$

number of cuts performed

Potential - Solution

Potential = #trees + 2·#marked nodes

note this 2

- We currently have 2 sources to potential change. We have to see if previous amortized analysis has changed

Amortized Cost of Decrease-key

	Actual cost	potential: Δ Trees	Potential: 2 ·ΔMarks	Amortized cost
Decrease-key	c	$+c$	$\leq 2 \cdot (2 - c)$	$O(1)$

number of cuts performed

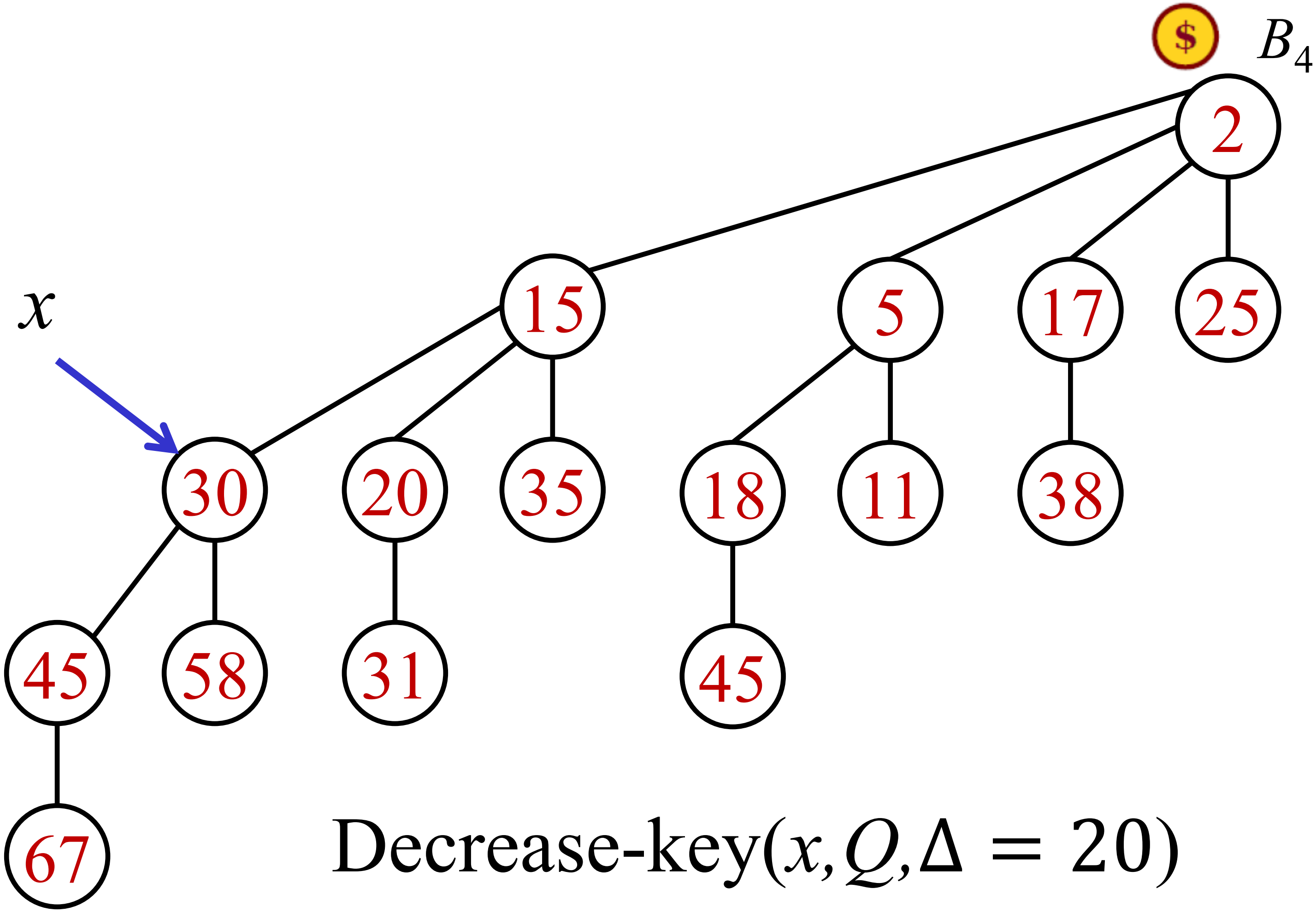
Fibonacci heaps operations

- Inserts pays for Del-min, Decrease-key pays for itself:

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

* up to scaling

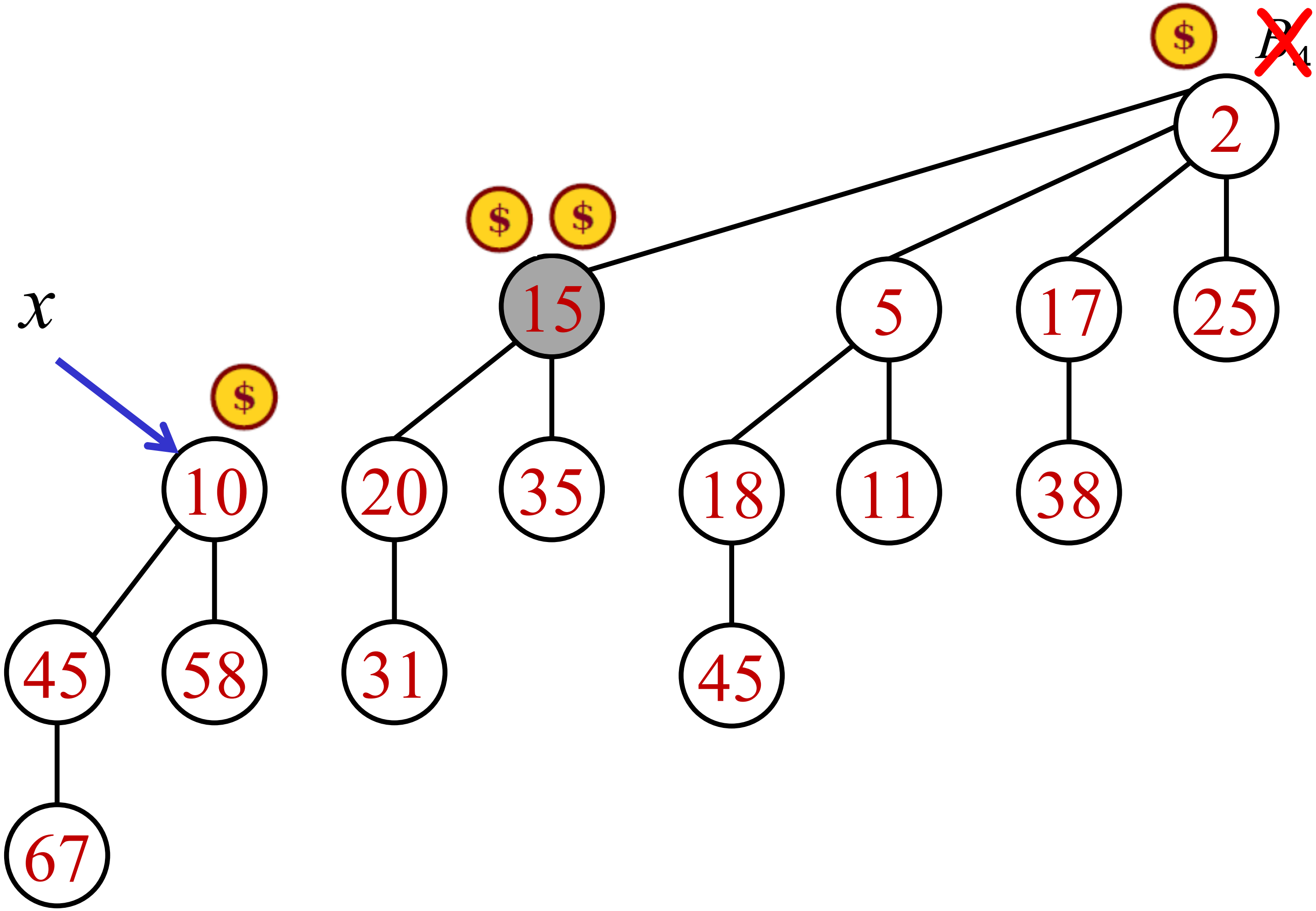
Putting it all together (account method)



Decrease-key$(x, Q, \Delta = 20)$

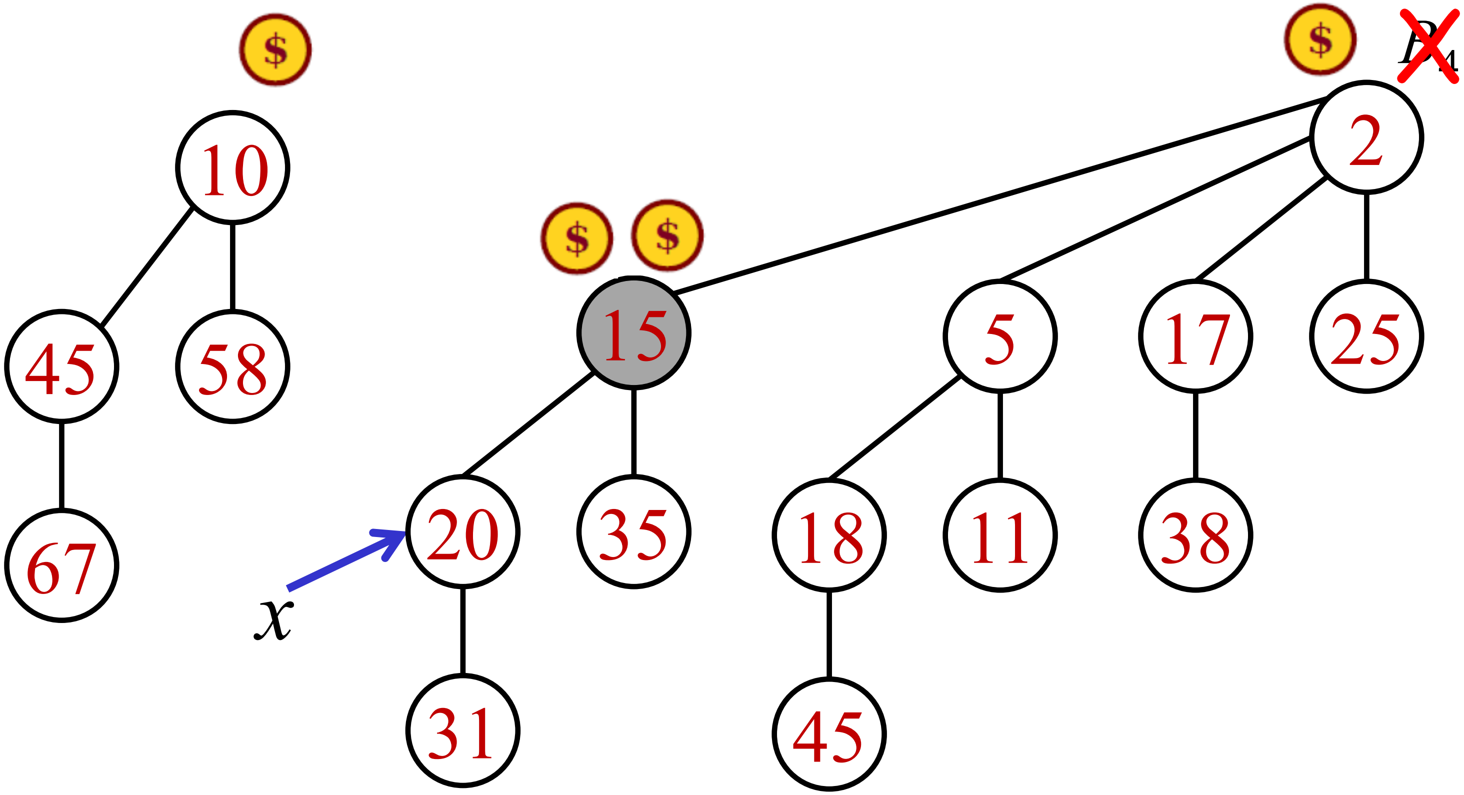
Potential = #trees + 2 #marked nodes = $1 + 2 \cdot 0 = 1$

Putting it all together (account method)



Potential = #trees + 2 #marked nodes $= 2 + 2 \cdot 1 = 4$

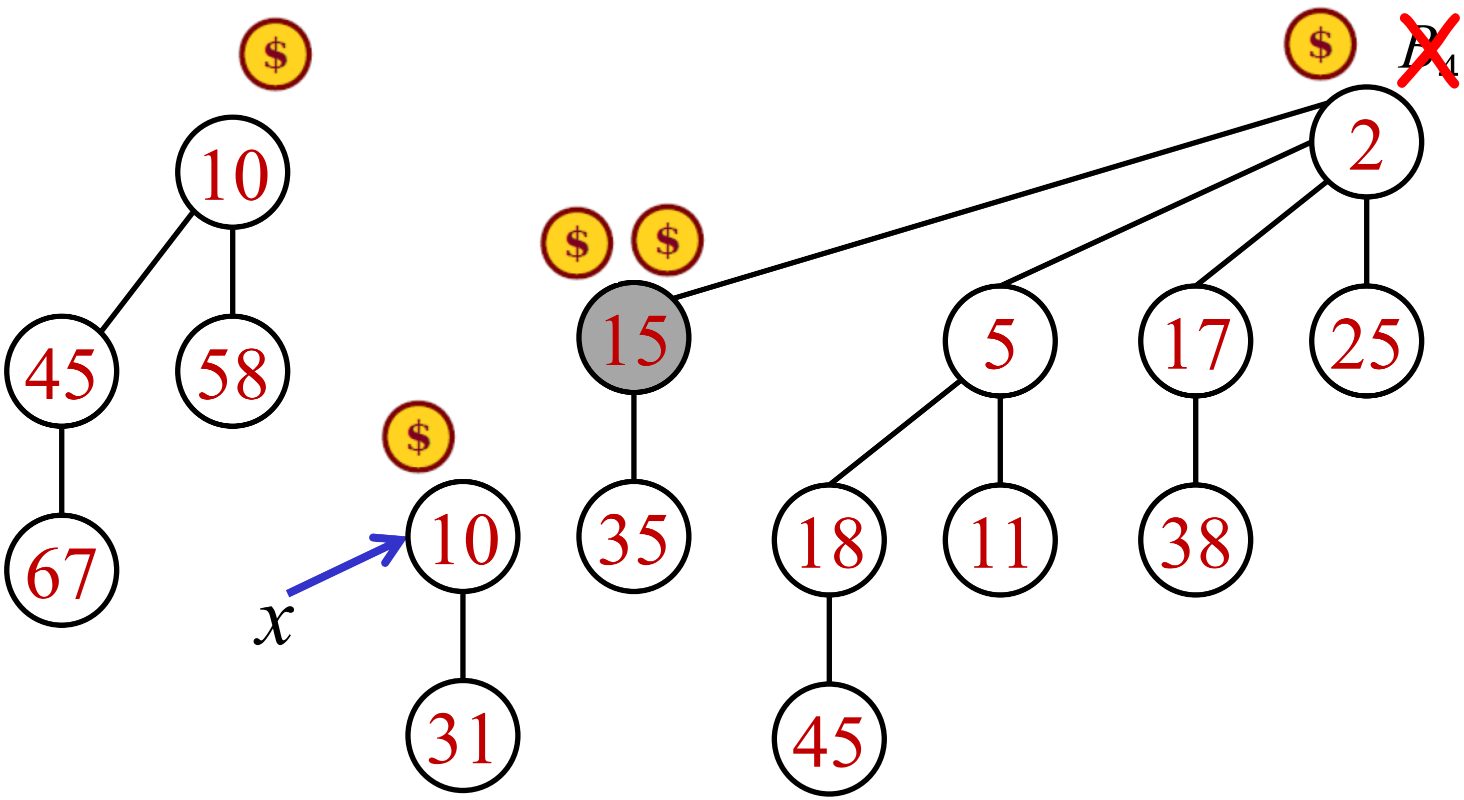
Putting it all together (account method)



Decrease-key$(x, Q, \Delta = 10)$

Potential = #trees + 2 #marked nodes = $2 + 2 \cdot 1 = 4$

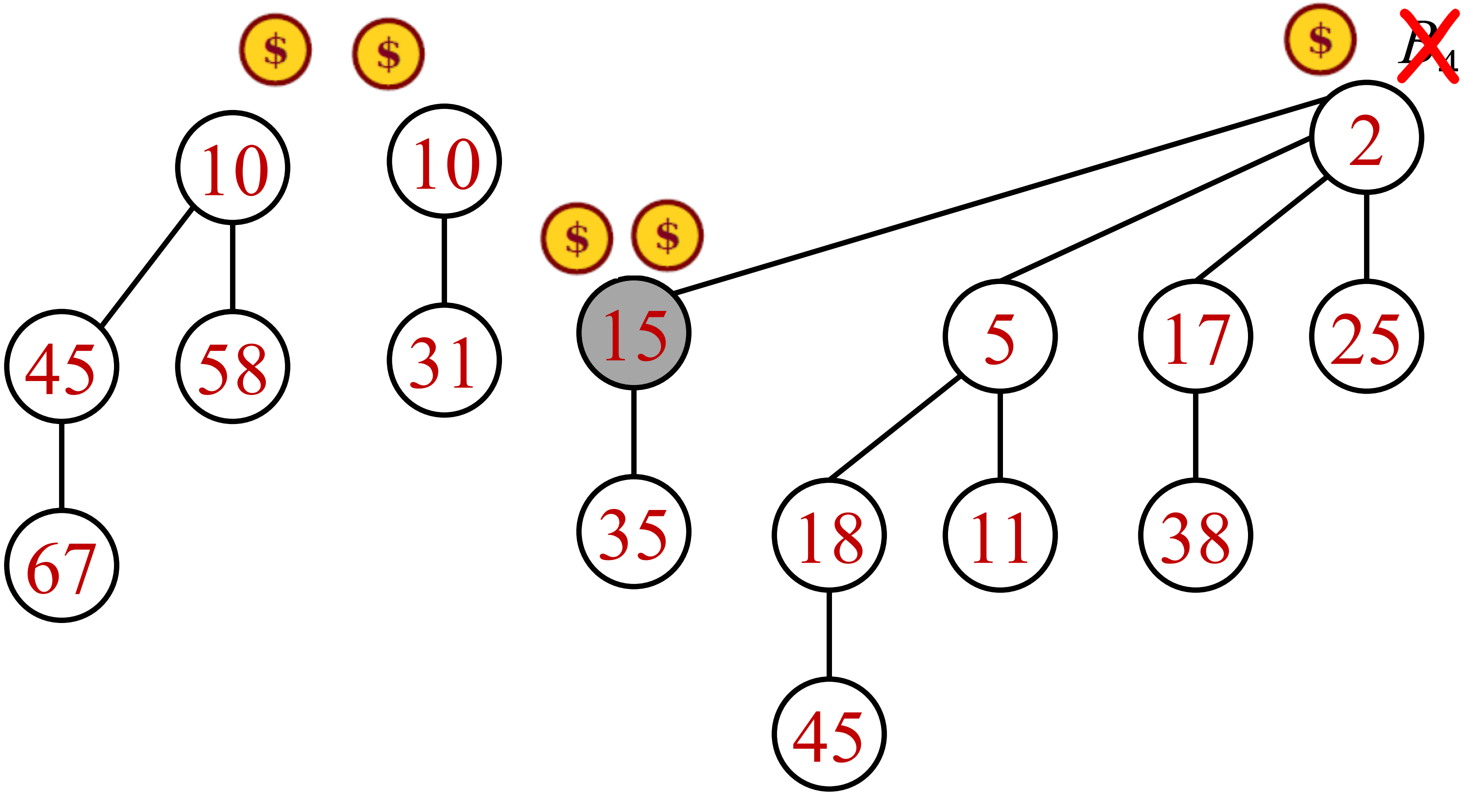
Putting it all together (account method)



$$\text{Decrease-key}(x, Q, \Delta = 10)$$

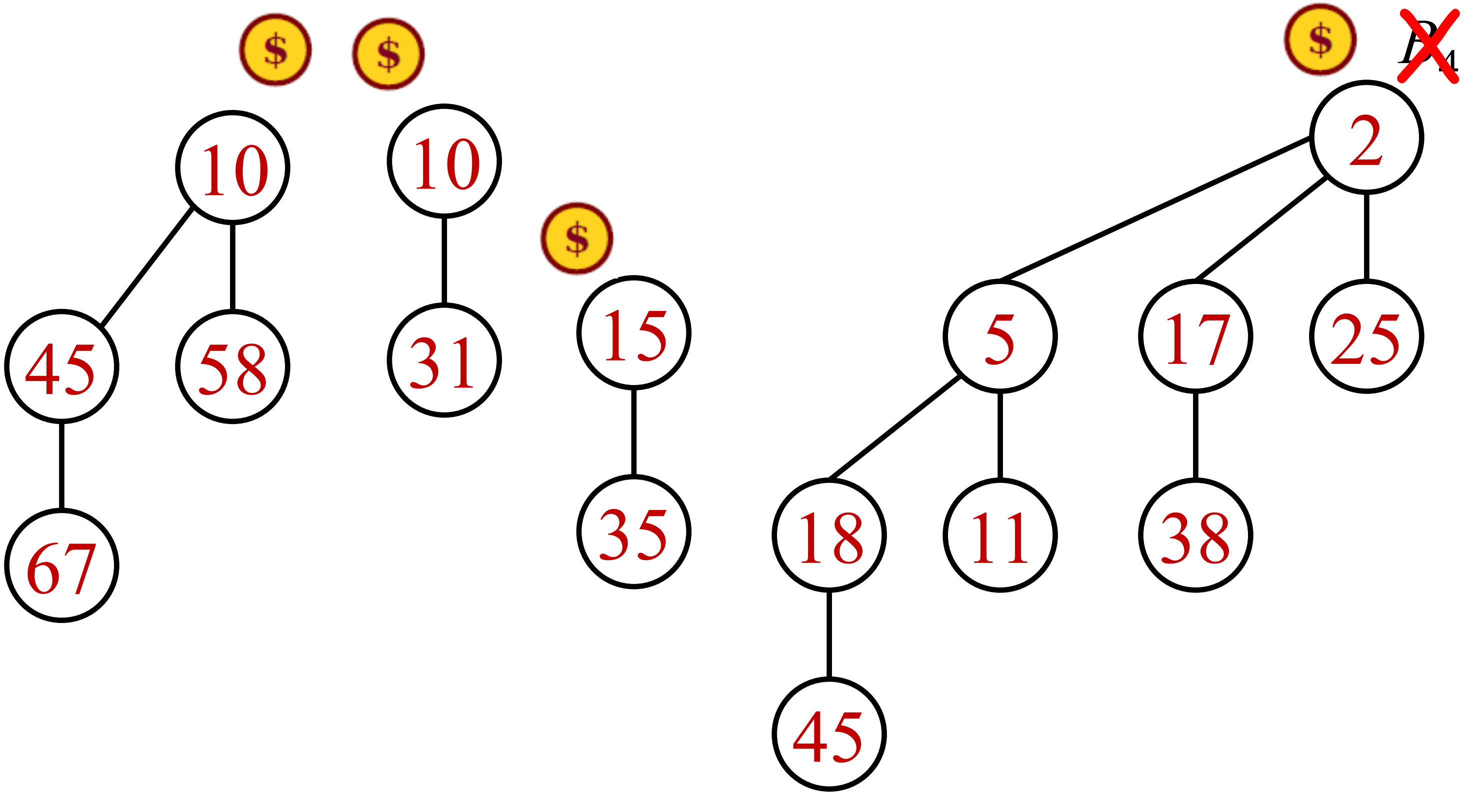
Potential = #trees + 2 #marked nodes = $3 + 2 \cdot 1 = 5$

Putting it all together (account method)



Potential = #trees + 2 #marked nodes = $4 + 0 \cdot 1 = 4$

Putting it all together (account method)



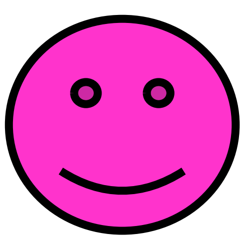
Potential = #trees + 2 #marked nodes = $4 + 0 \cdot 1 = 4$

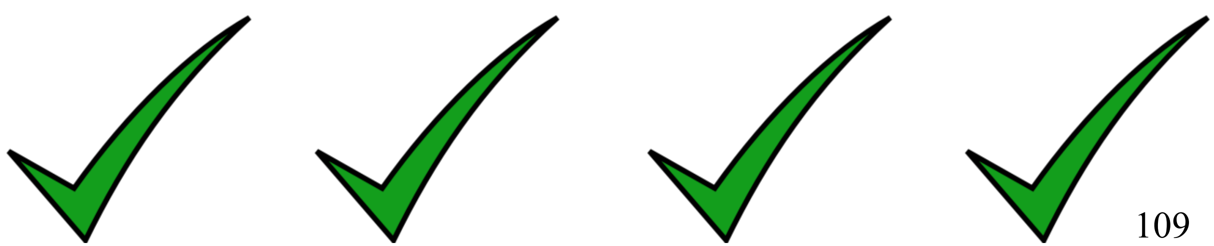
Heaps / Priority queues

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

Worst case

Amortized





Summary: Life is all about Tradeoffs

- If we impose no structural constraints on our trees, then trees of large degree may have too few nodes. This leads to wrecking our runtime bounds for extract-min.

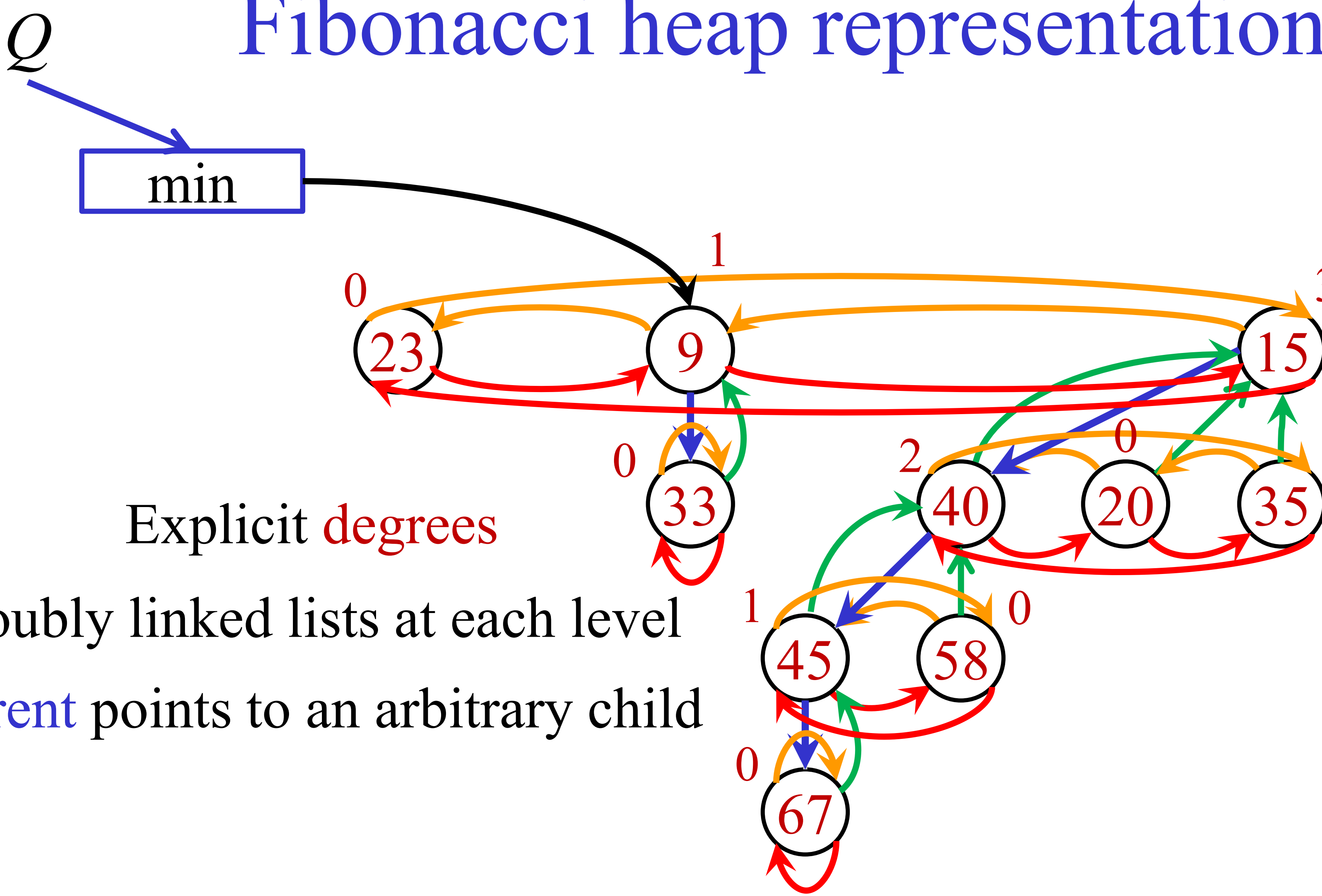
- If we impose too many structural constraints on our trees, then we have to spend too much time fixing up trees. This leads to decrease-key taking too long.

- Fibonacci heaps strike a balance
  - If we do a few decrease-keys, then the tree won't lose "too many" nodes.
  - If we do many decrease-keys, the information slowly propagates to the root (its degree slowly decreases).

Fibonacci Heaps: theory vs. practice

- Theoretically they look good
- Practically:
  - Less efficient than “simpler” heaps
  - Complicated to code and use a lot of memory for each node (see next slide)
  - Still $O(n)$ worst case for Delete-min

Fibonacci heap representation



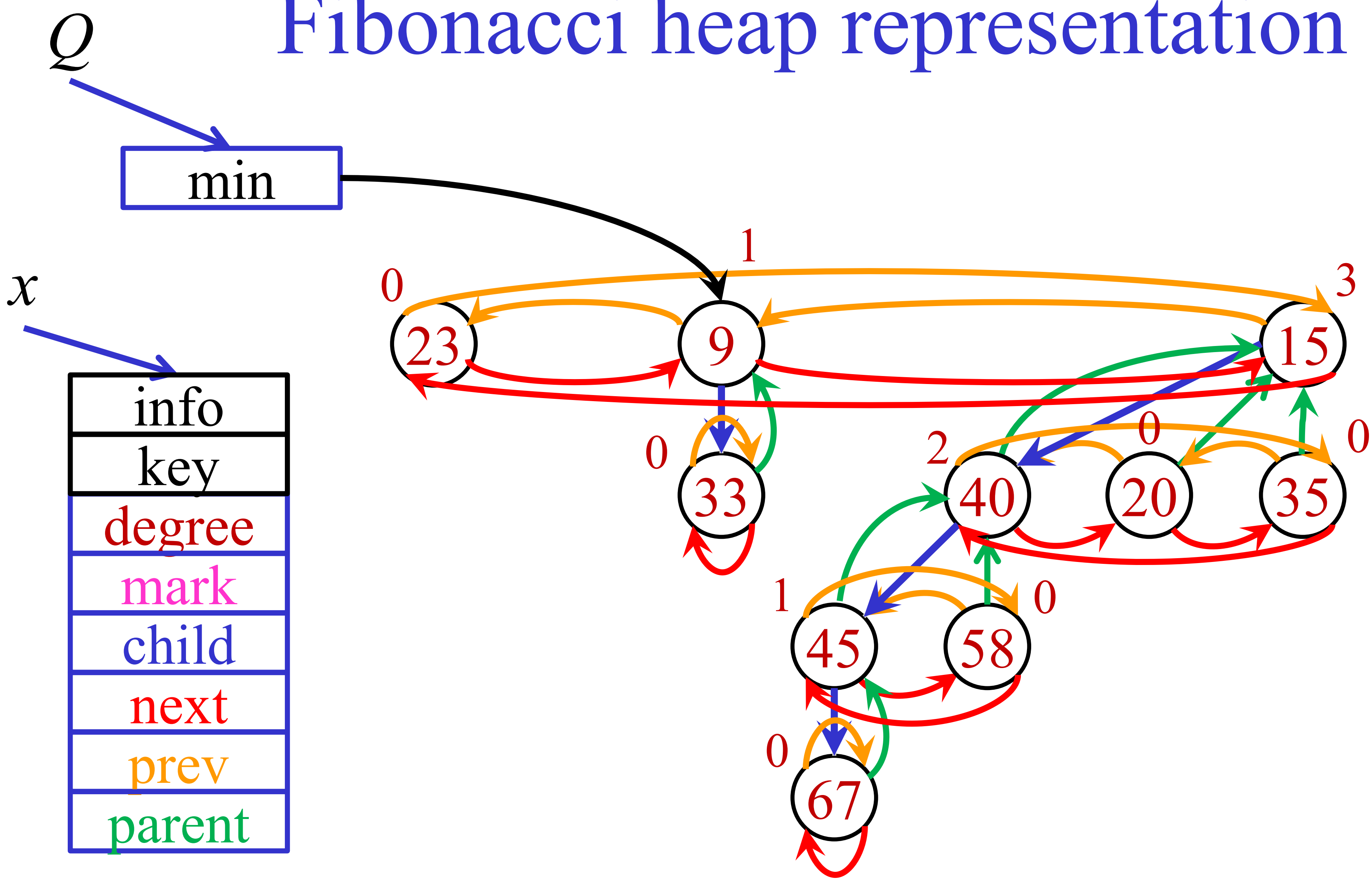
Explicit degrees

Doubly linked lists at each level

Parent points to an arbitrary child

4 pointers + degree + mark bit per node

Fibonacci heap representation



4 pointers + degree + mark bit per node

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

Consolidating / Successive linking

```
Function consolidate(x)
    to-buckets(x)
    return from-buckets()
```

```
Function to-buckets(x)
    for i ← 0 to log_φ n do
        B[i] ← null

    x.prev.next ← null
    while x ≠ null do
        y ← x
        x ← x.next
        while B[y.rank] ≠ null do
            y ← link(y, B[y.rank])
            B[y.rank − 1] ≥ null
        B[y.rank] ← y
```

```
Function from-buckets()
    x ← null
    for i ← 0 to log_φ n do
        if B[i] ≠ null then
            if x = null then
                x ← B[i]
                x.next ← x
                x.prev ← x
            else
                insert-after(x, B[i])
                if B[i].key < x.key then
                    x ← B[i]
    return x
```

Heaps: famous last words...

- Binary heaps, binomial heaps, and Fibonacci heaps are all inefficient in their support of Search
  - Operations such as Decrease-key Delete-min require a pointer to the node

- Min vs. max

- A highly recommended summary:
  https://en.wikipedia.org/wiki/Fibonacci_heap