Binary, Binomial and Fibonacci Heaps

These lecture slides are borrowed from CLRS, Chapters 6, 19 and 20.

BY Kevin Wayne

Priority Queues

Supports the following operations.

- Insert element x.
- Return min element.
- Return and delete minimum element.
- Decrease key of element x to k.

Applications.

- Dijkstra's shortest path algorithm.
- Prim's MST algorithm.
- Event-driven simulation.
- Huffman encoding.
- Heapsort.
- . . .

Priority Queues in Action

Dijkstra's Shortest Path Algorithm

```
PQinit()
for each v ∈ V
   key(v) ← ∞
   PQinsert(v)

key(s) ← 0
while (!PQisempty())
   v = PQdelmin()
   for each w ∈ Q s.t (v,w) ∈ E
      if π(w) > π(v) + c(v,w)
         PQdecrease(w, π(v) + c(v,w))
```

Priority Queues

		Heaps			
Operation	**Linked List**	**Binary**	**Binomial**	**Fibonacci ***	**Relaxed**
make-heap	1	1	1	1	1
insert	1	log N	log N	1	1
find-min	N	1	log N	1	1
delete-min	N	log N	log N	log N	log N
union	1	N	log N	1	1
decrease-key	1	log N	log N	1	1
delete	N	log N	log N	log N	log N
is-empty	1	1	1	1	1

Dijkstra/Prim
1 make-heap
|V| insert
|V| delete-min
|E| decrease-key

$O(|V|^2)$

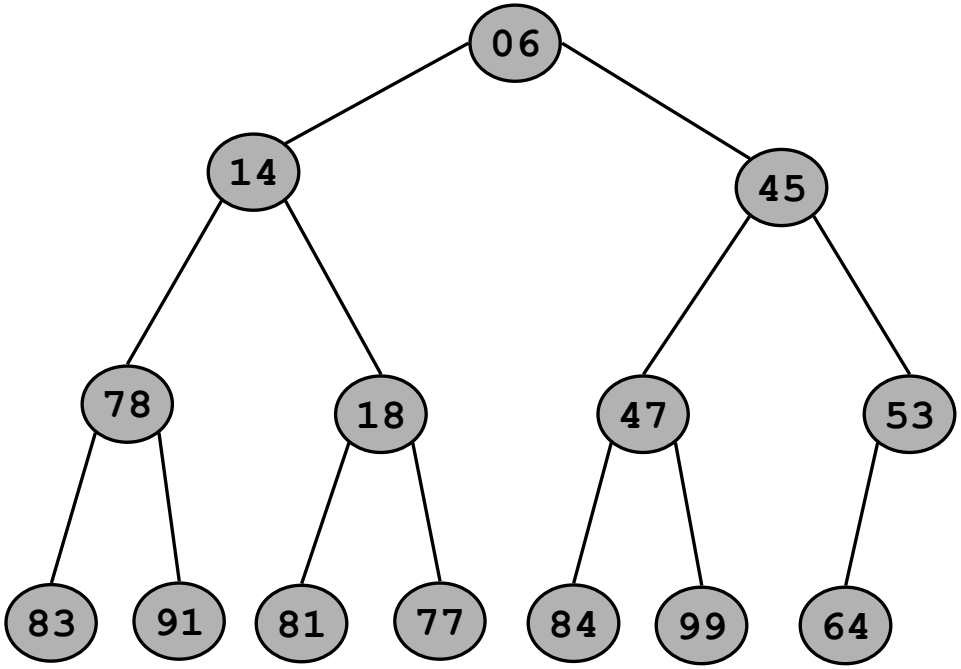
$O(|E| \log |V|)$

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

Binary Heap: Definition

Binary heap.

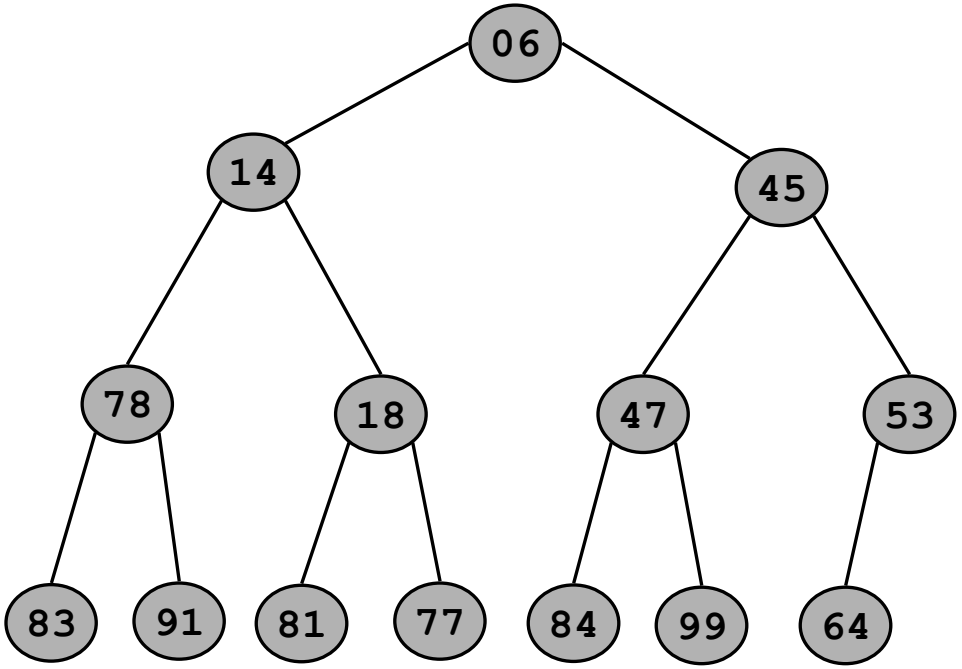
- **Almost complete binary tree.**
  - **filled on all levels, except last, where filled from left to right**
- **Min-heap ordered.**
  - **every child greater than (or equal to) parent**



Binary Heap: Properties

Properties.

- **Min element is in root.**
- **Heap with N elements has height = $\lfloor \log_2 N \rfloor$.**

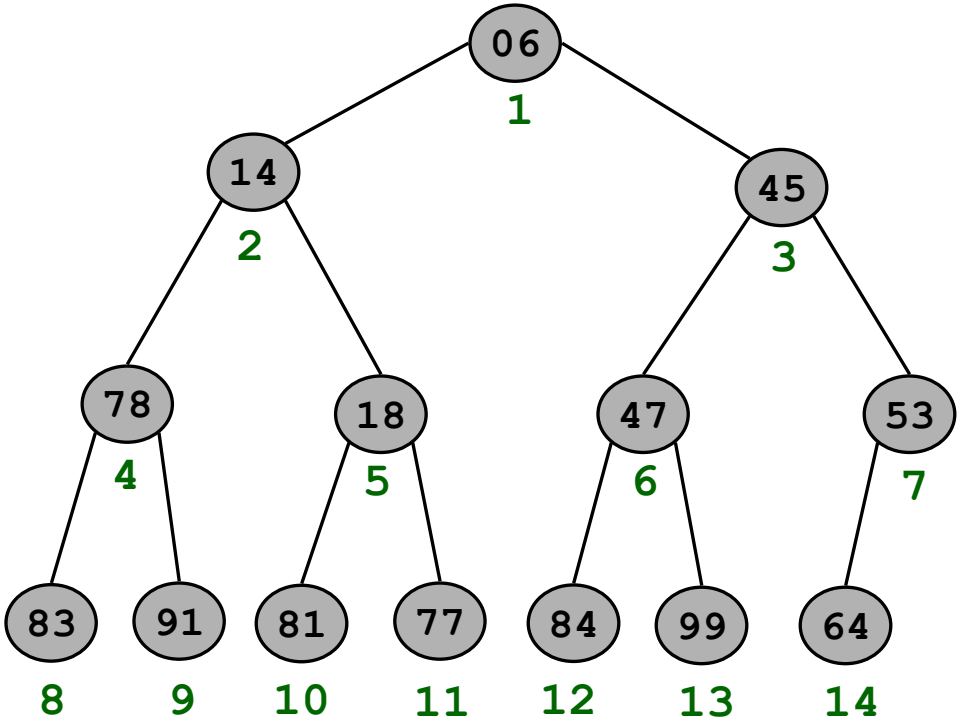


N = 14
Height = 3

Binary Heaps: Array Implementation

Implementing binary heaps.

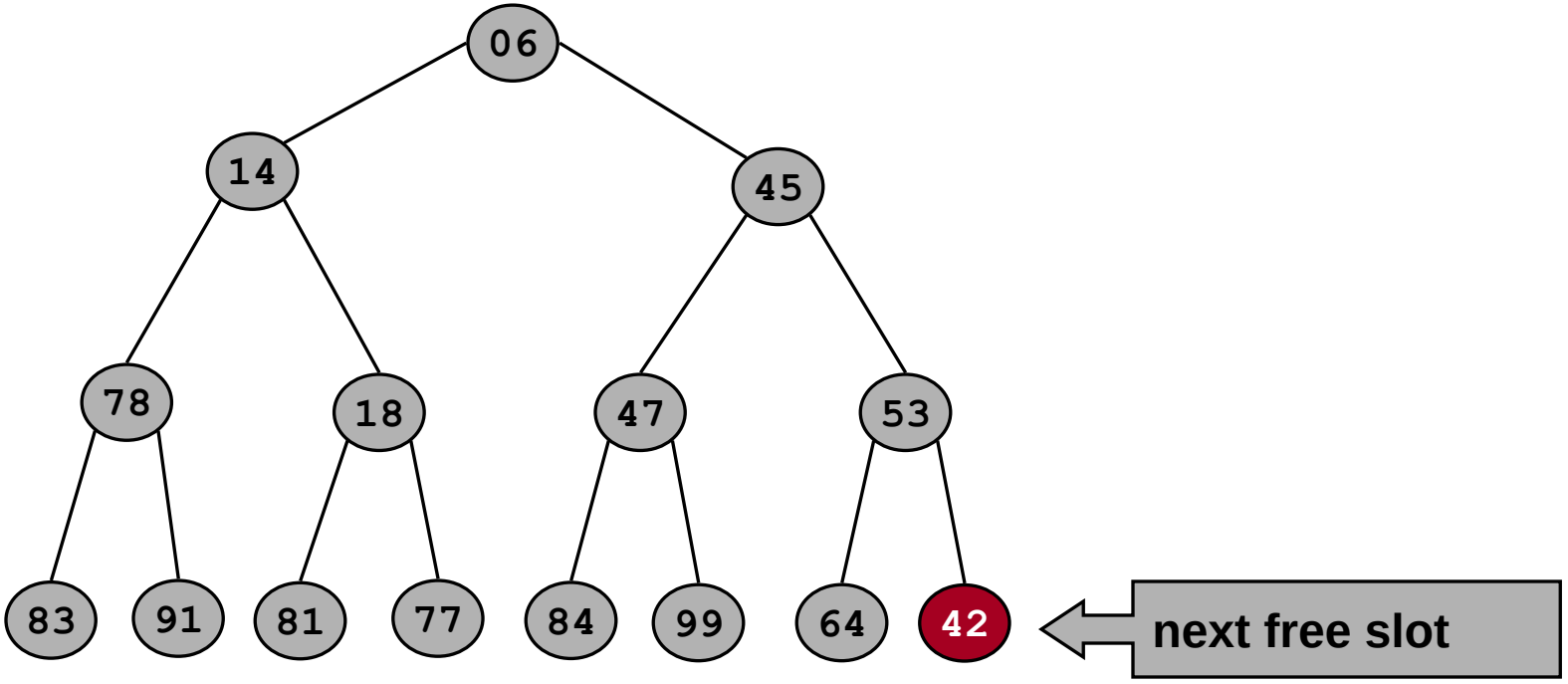
- Use an array: no need for explicit parent or child pointers.
  - `Parent(i) = ⌊i/2⌋`
  - `Left(i) = 2i`
  - `Right(i) = 2i + 1`



Binary Heap: Insertion

Insert element x into heap.

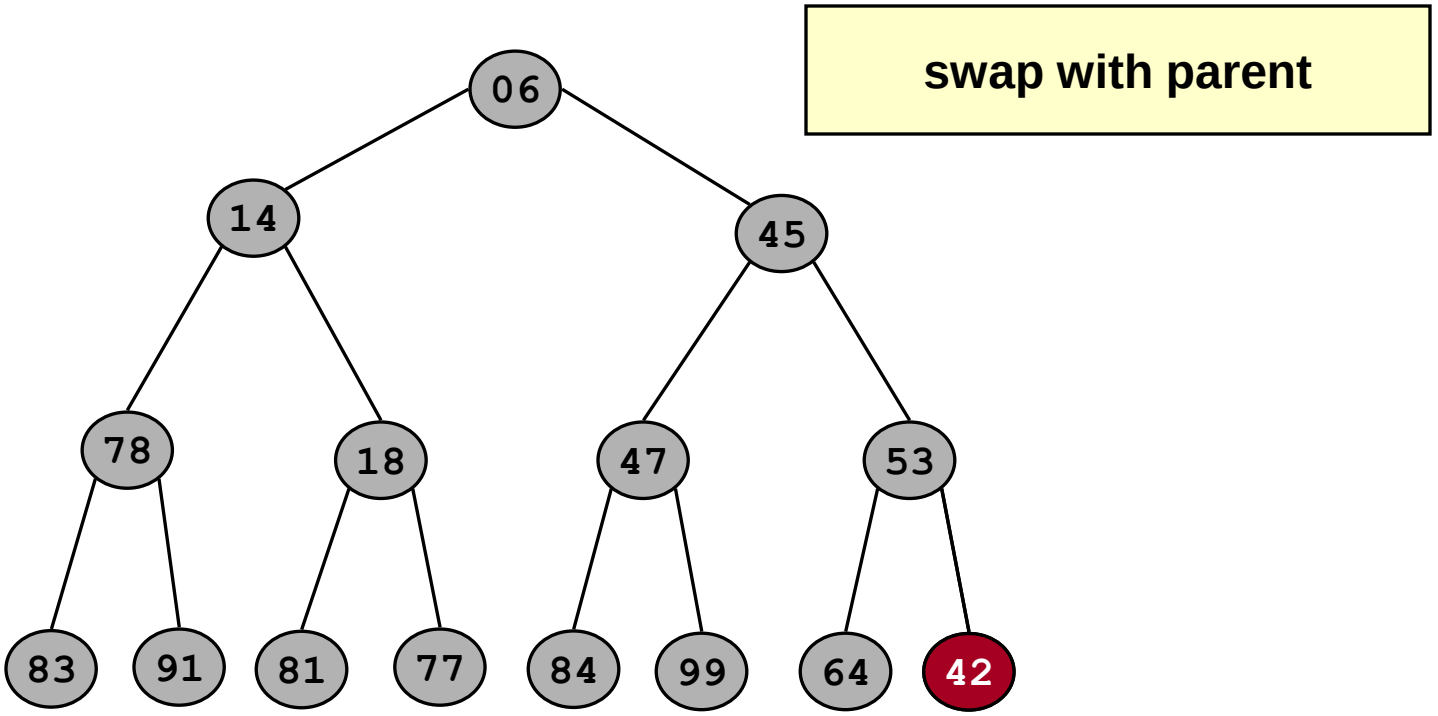
- **Insert into next available slot.**
- **Bubble up until it's heap ordered.**
  - **Peter principle: nodes rise to level of incompetence**



Binary Heap: Insertion

Insert element x into heap.

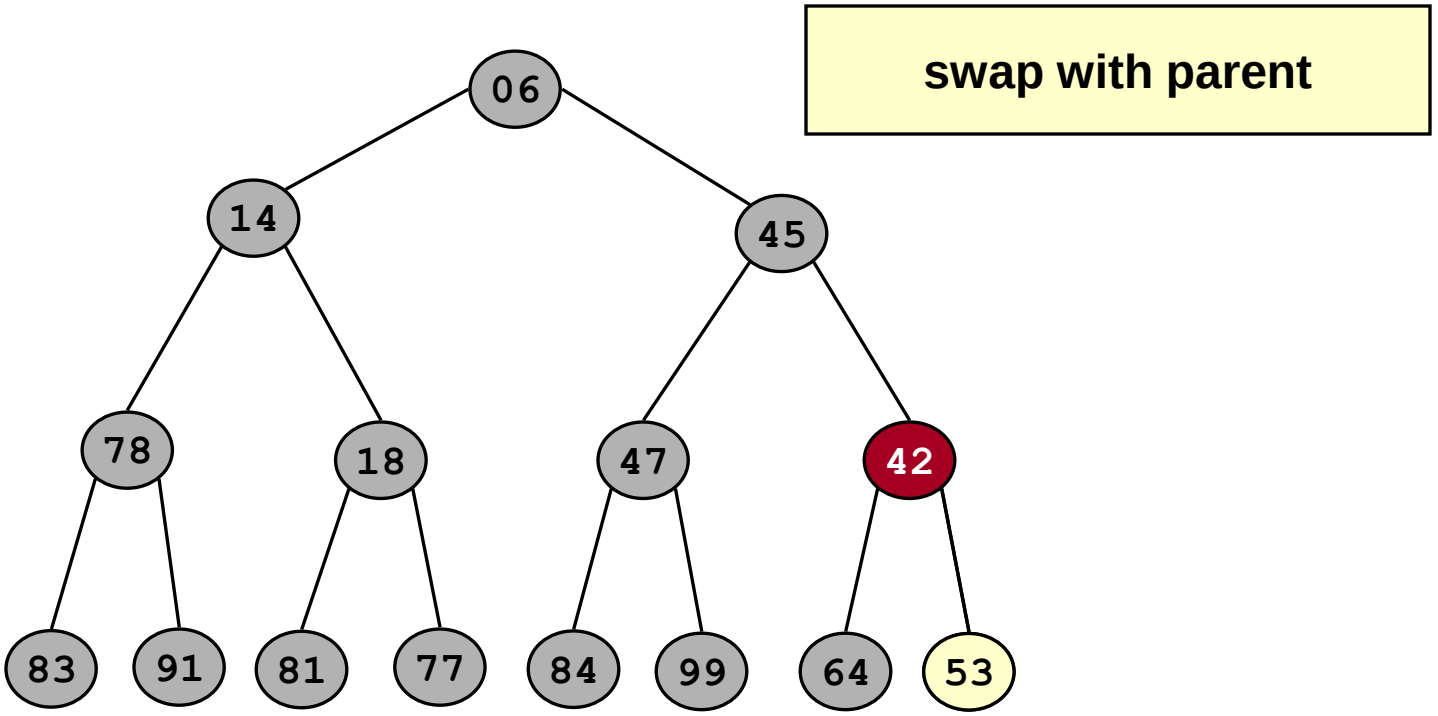
- **Insert into next available slot.**
- **Bubble up until it's heap ordered.**
  - **Peter principle: nodes rise to level of incompetence**



Binary Heap: Insertion

Insert element x into heap.

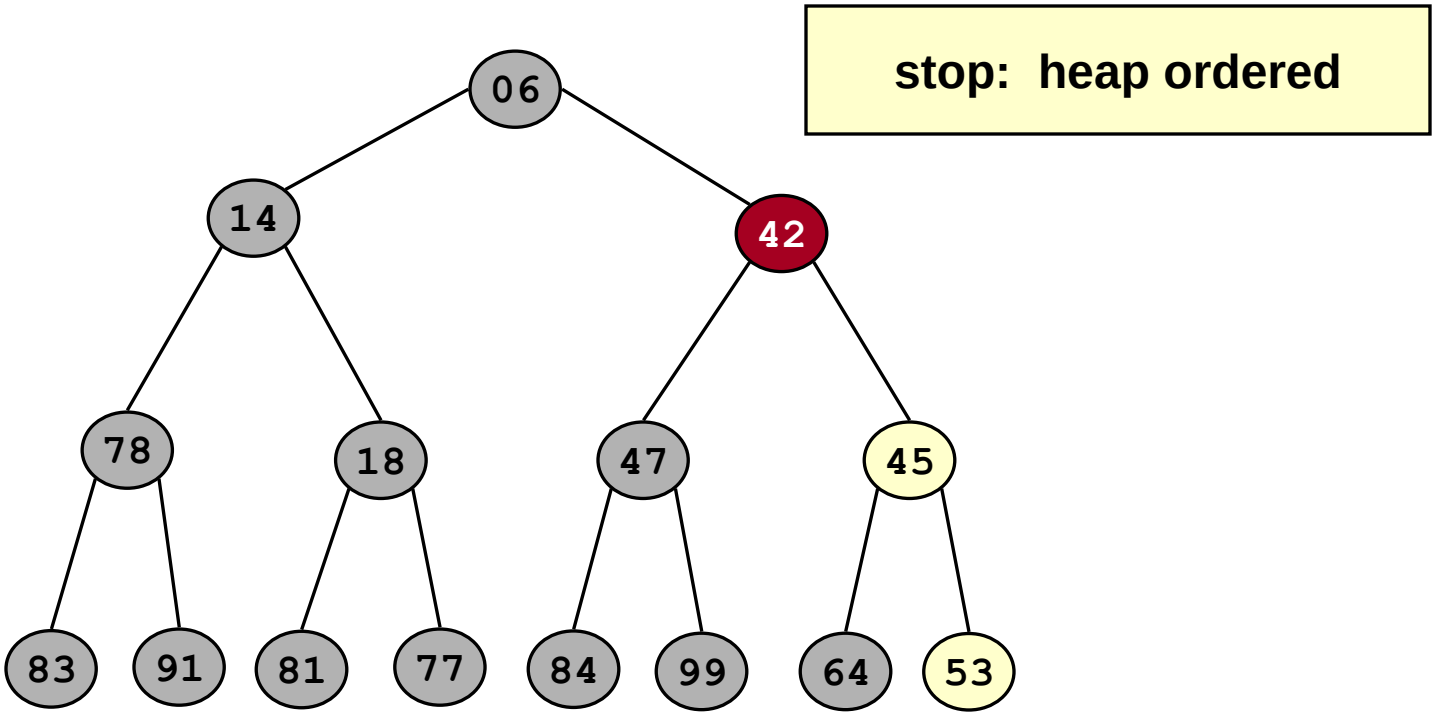
- **Insert into next available slot.**
- **Bubble up until it's heap ordered.**
  - **Peter principle: nodes rise to level of incompetence**



Binary Heap: Insertion

Insert element x into heap.

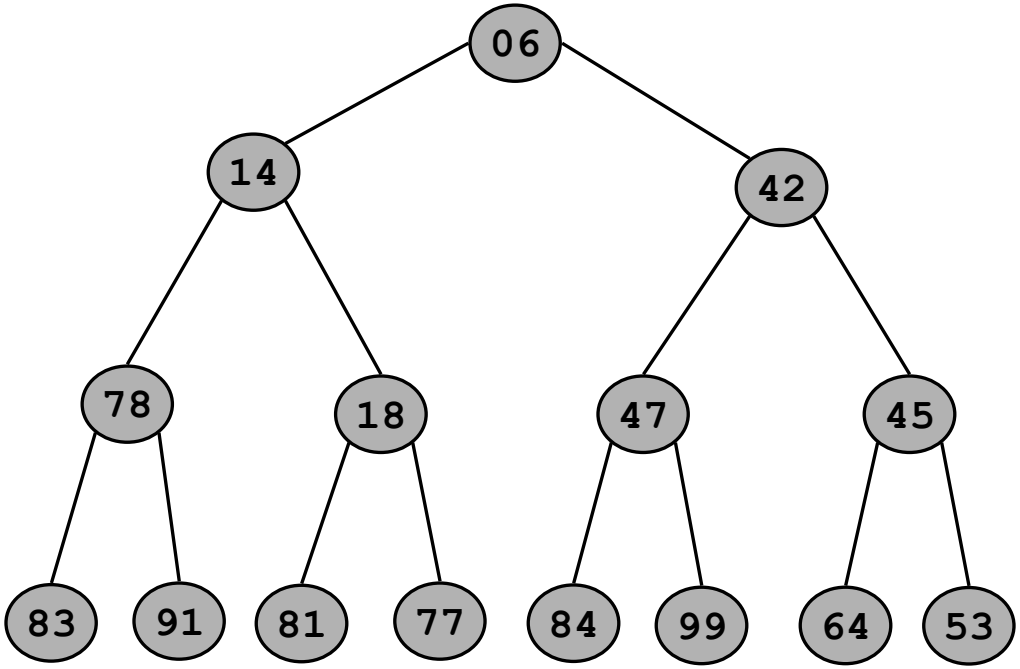
- **Insert into next available slot.**
- **Bubble up until it's heap ordered.**
  - **Peter principle: nodes rise to level of incompetence**
- **O(log N) operations.**



Binary Heap: Decrease Key

Decrease key of element x to k.

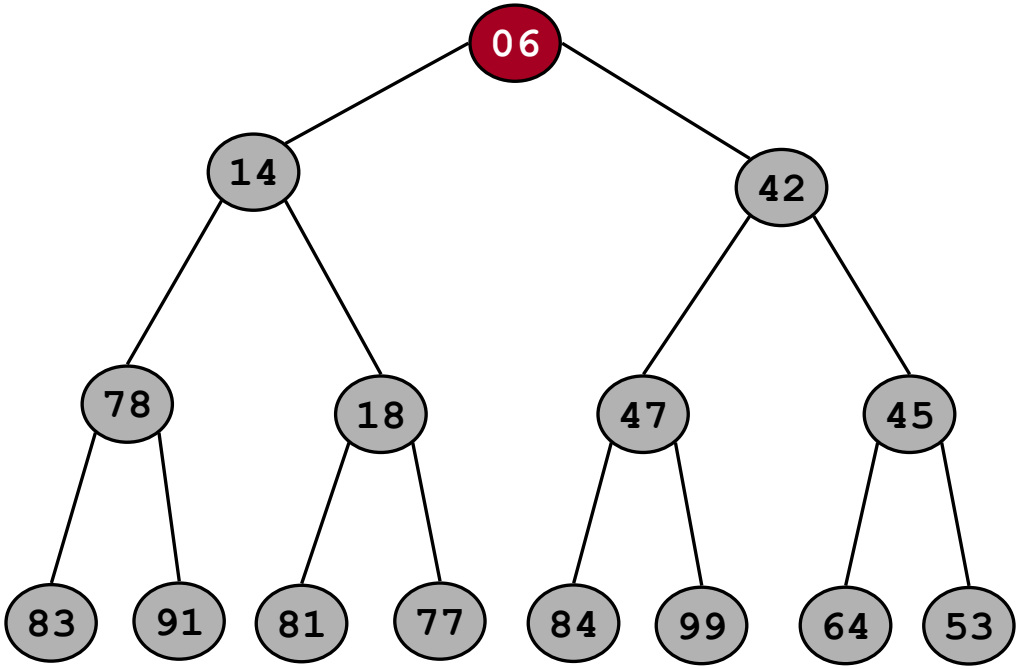
- **Bubble up until it's heap ordered.**
- **O(log N) operations.**



Binary Heap: Delete Min

Delete minimum element from heap.

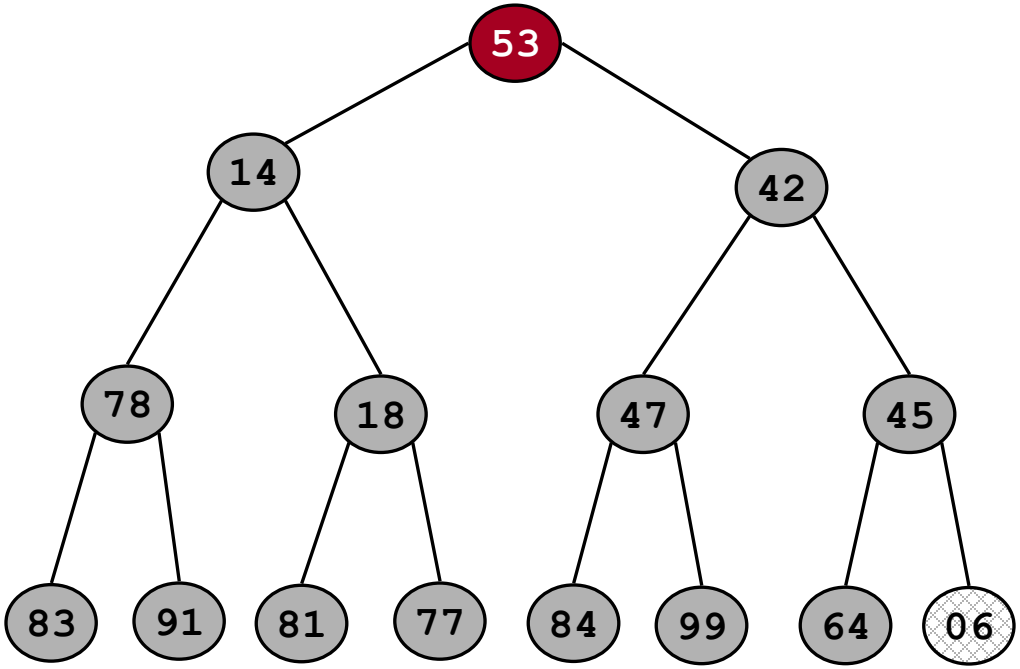
- **Exchange root with rightmost leaf.**
- **Bubble root down until it's heap ordered.**
  - **power struggle principle: better subordinate is promoted**



Binary Heap: Delete Min

Delete minimum element from heap.

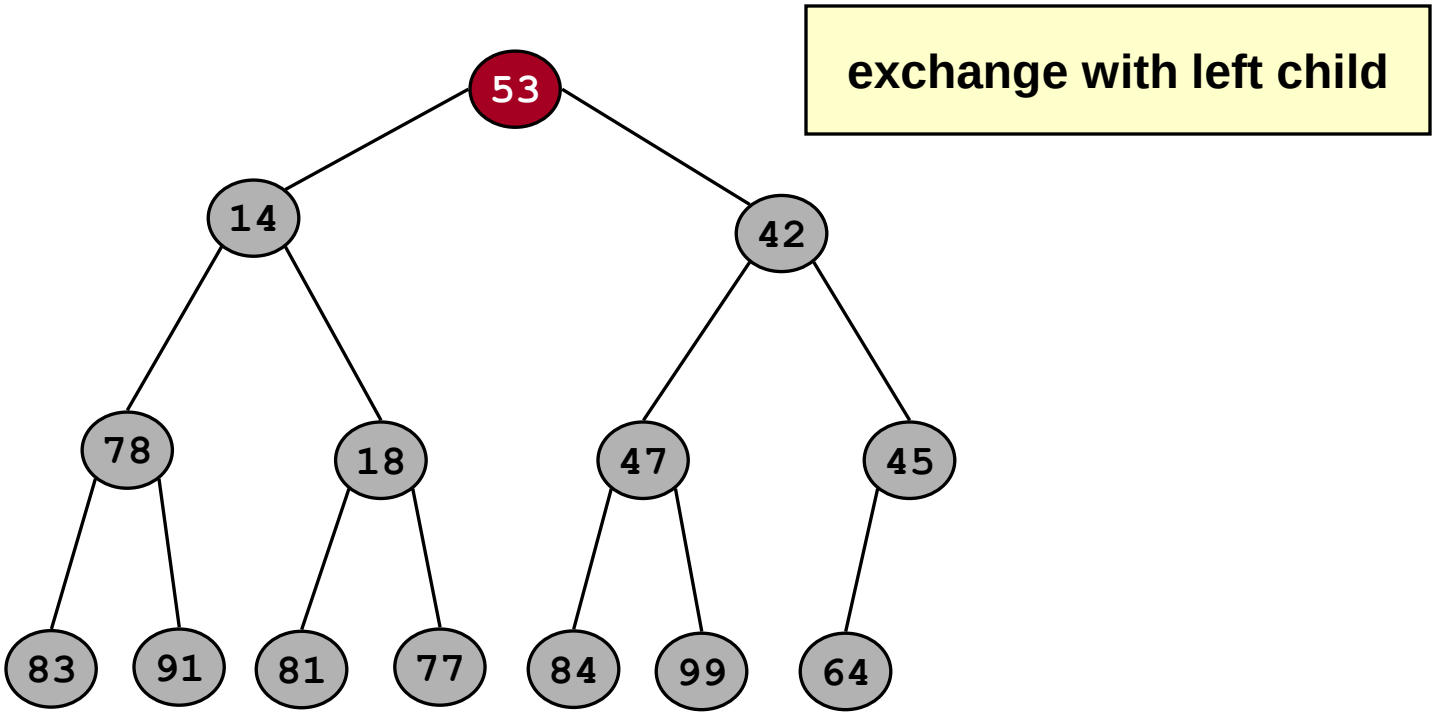
- **Exchange root with rightmost leaf.**
- **Bubble root down until it's heap ordered.**
  - **power struggle principle: better subordinate is promoted**



Binary Heap: Delete Min

Delete minimum element from heap.

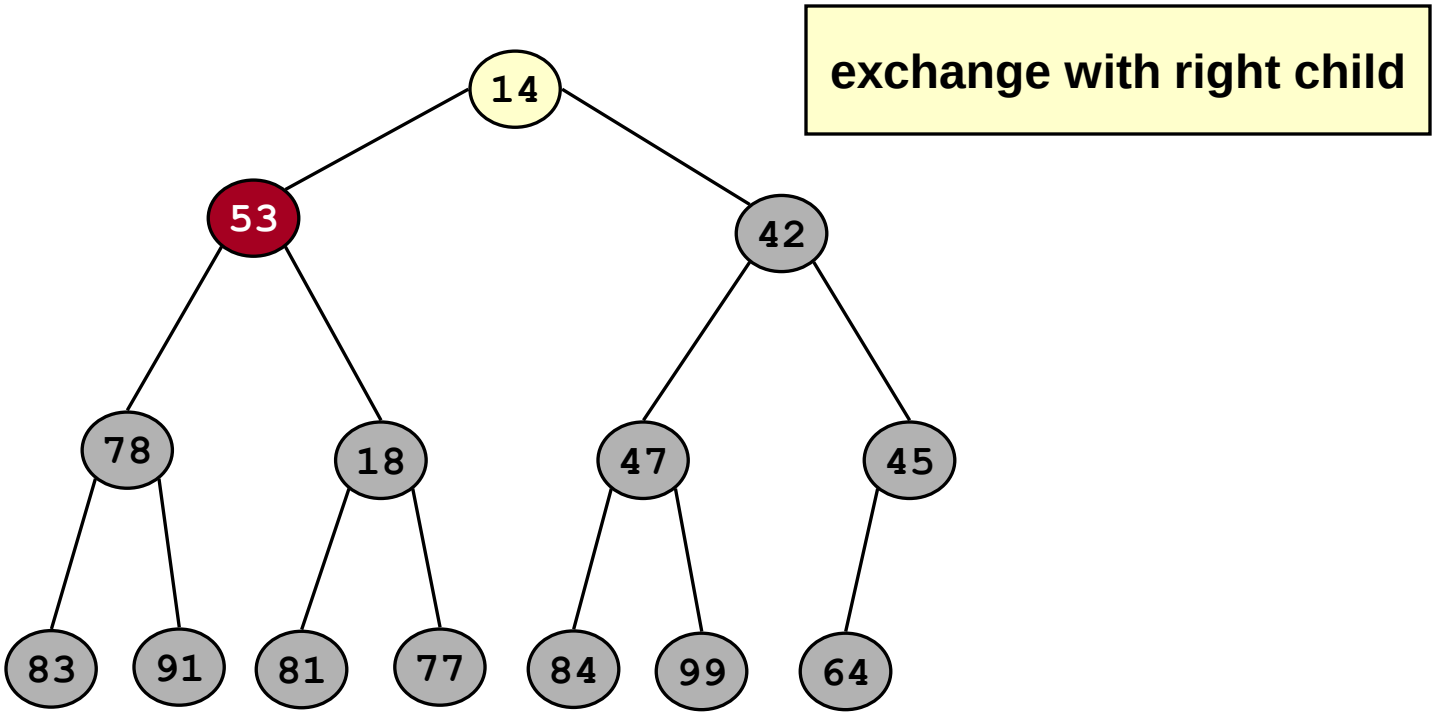
- **Exchange root with rightmost leaf.**
- **Bubble root down until it's heap ordered.**
  - **power struggle principle: better subordinate is promoted**



Binary Heap: Delete Min

Delete minimum element from heap.

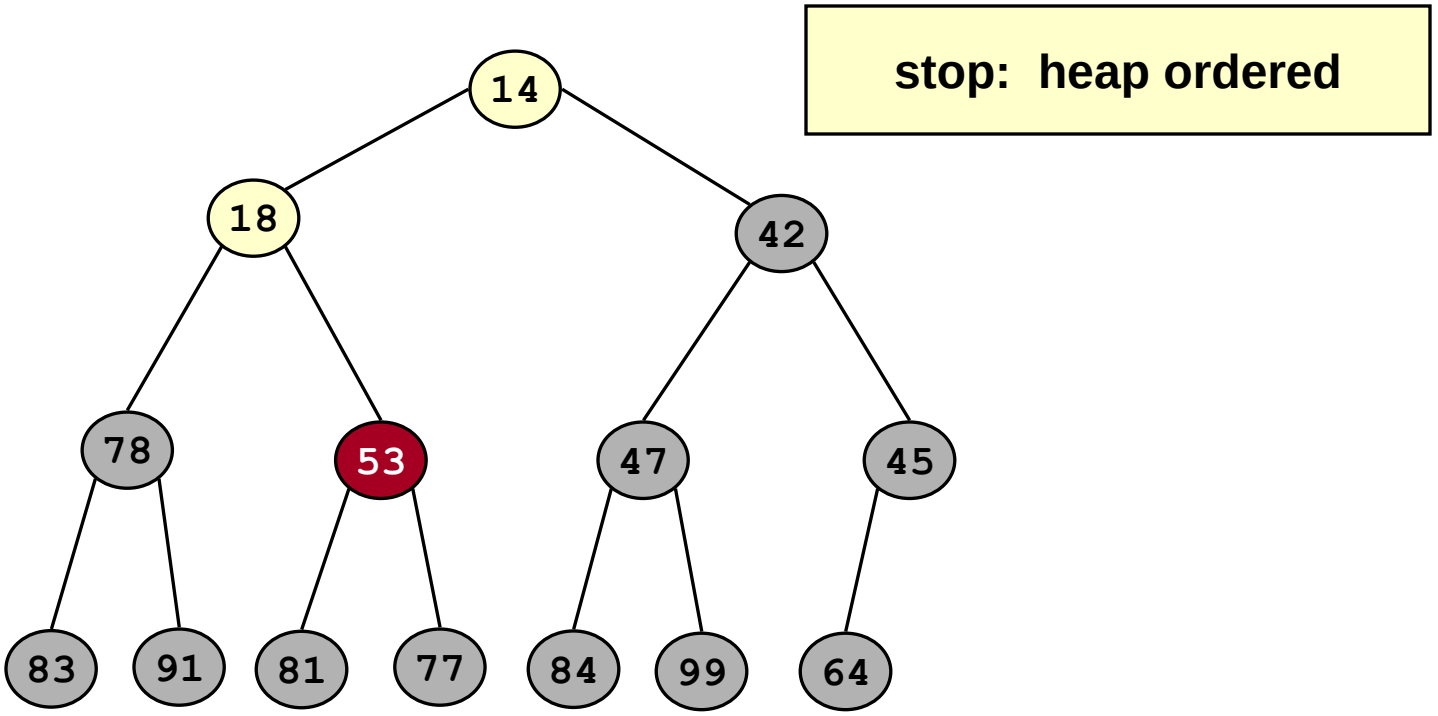
- **Exchange root with rightmost leaf.**
- **Bubble root down until it's heap ordered.**
  - **power struggle principle: better subordinate is promoted**



Binary Heap: Delete Min

Delete minimum element from heap.

- **Exchange root with rightmost leaf.**
- **Bubble root down until it's heap ordered.**
  - **power struggle principle: better subordinate is promoted**
- **O(log N) operations.**



Binary Heap: Heapsort

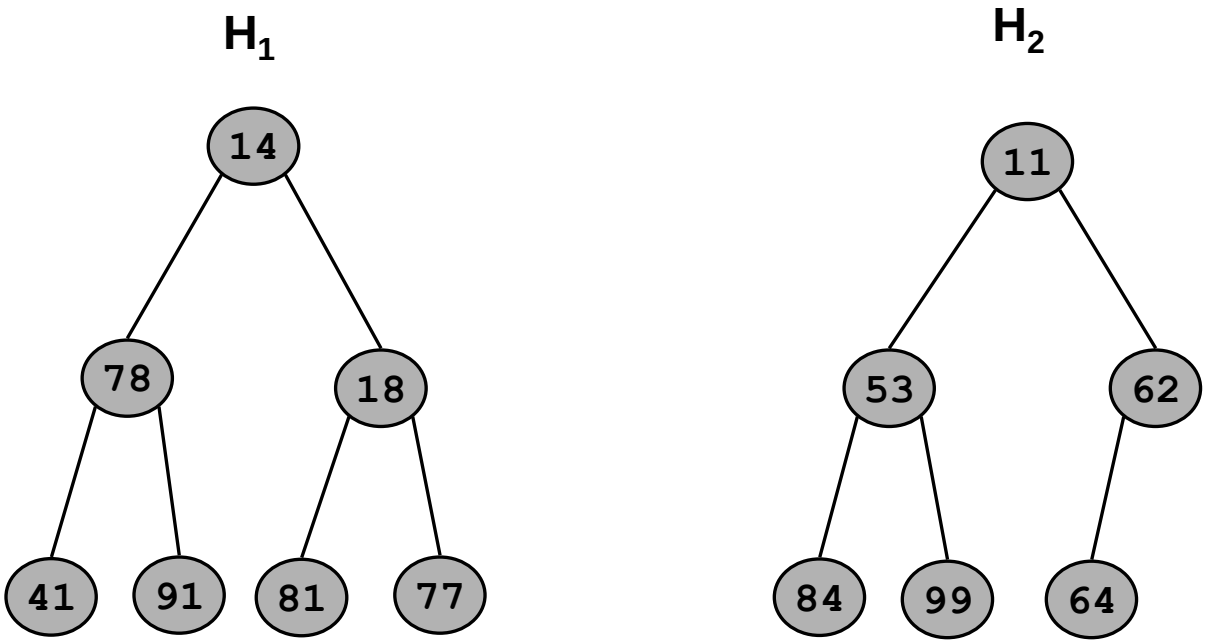
Heapsort.

- **Insert N items into binary heap.**
- **Perform N delete-min operations.**
- **O(N log N) sort.**
- **No extra storage.**

Binary Heap: Union

Union.

- Combine two binary heaps H_1 and H_2 into a single heap.
- No easy solution.
  - $\Omega(N)$ operations apparently required
- Can support fast union with fancier heaps.



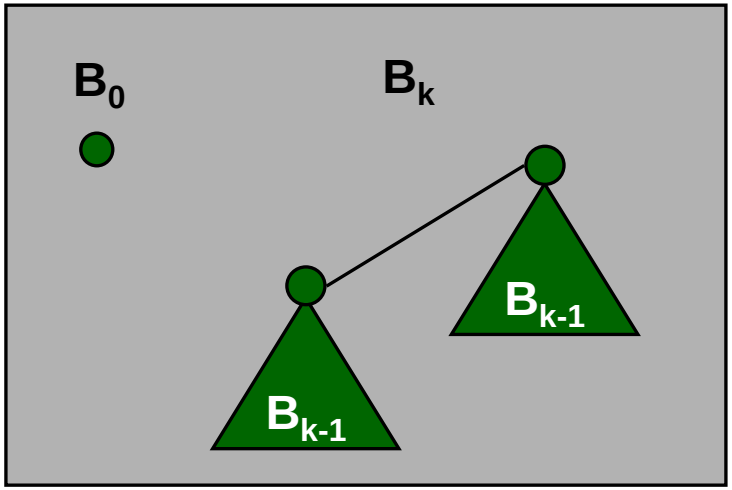
Priority Queues

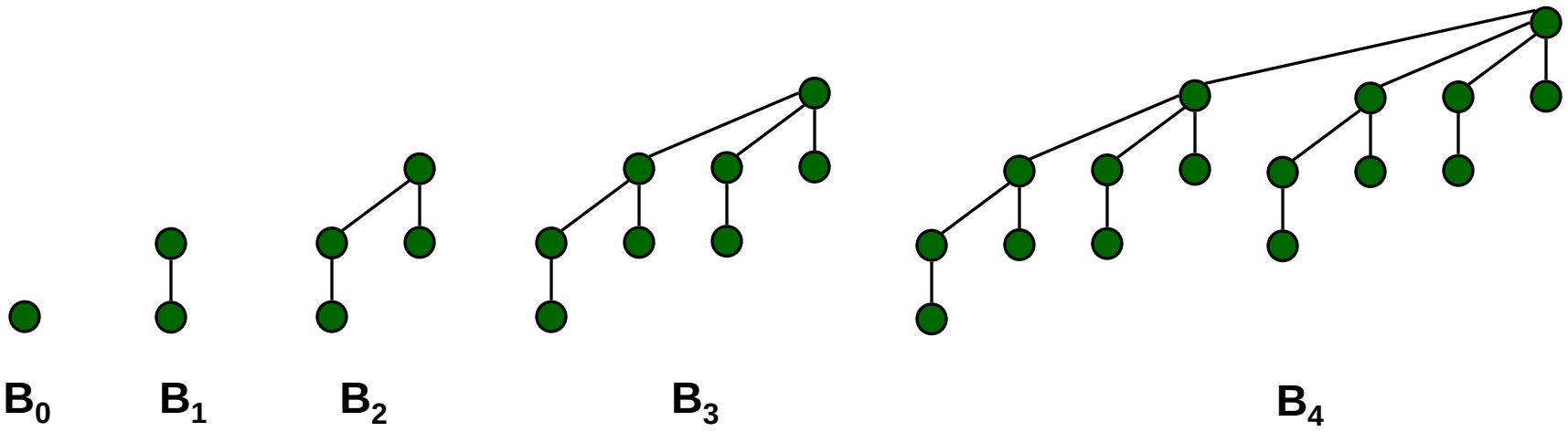
		Heaps			
Operation	Linked List	Binary	Binomial	Fibonacci *	Relaxed
make-heap	1	1	1	1	1
insert	1	log N	log N	1	1
find-min	N	1	log N	1	1
delete-min	N	log N	log N	log N	log N
union	1	N	log N	1	1
decrease-key	1	log N	log N	1	1
delete	N	log N	log N	log N	log N
is-empty	1	1	1	1	1

Binomial Tree

Binomial tree.

- **Recursive definition:**

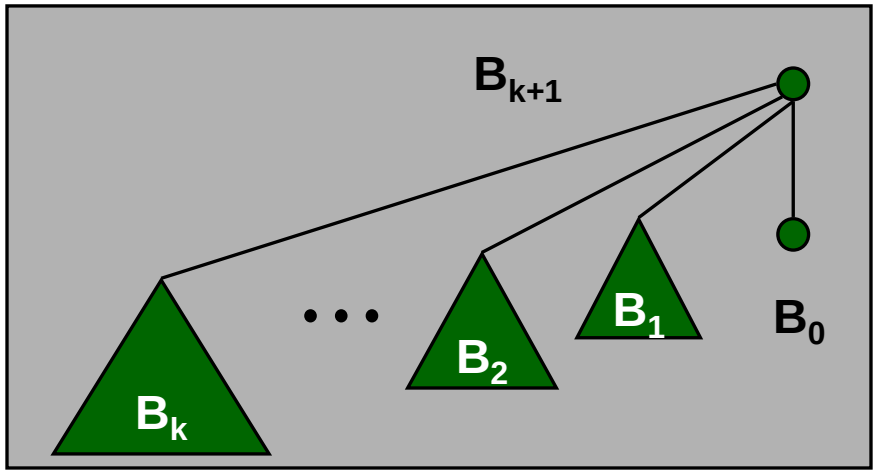




Binomial Tree

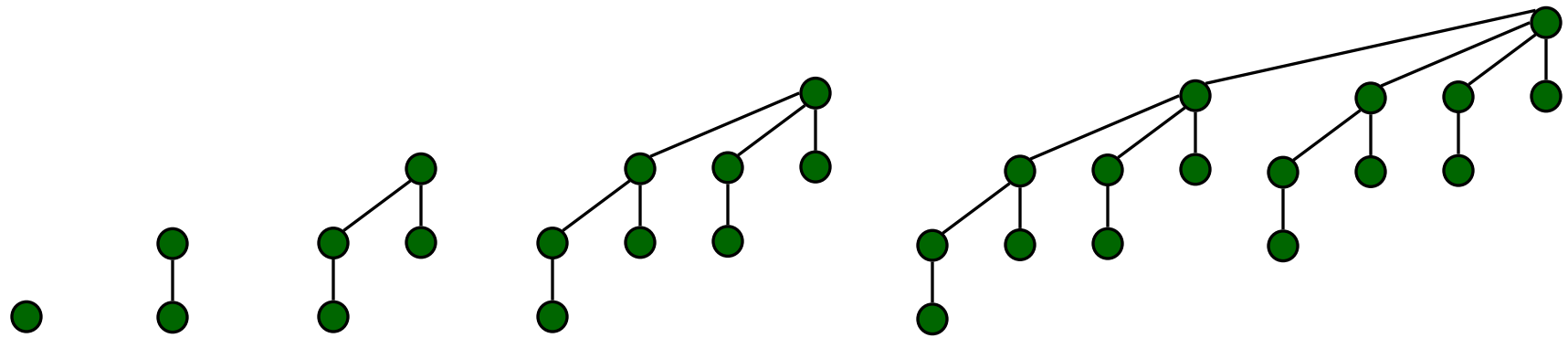
Useful properties of order k binomial tree B_k.

- **Number of nodes = 2^k.**
- **Height = k.**
- **Degree of root = k.**
- **Deleting root yields binomial trees B_{k-1}, … , B_0.**



Proof.

- **By induction on k.**

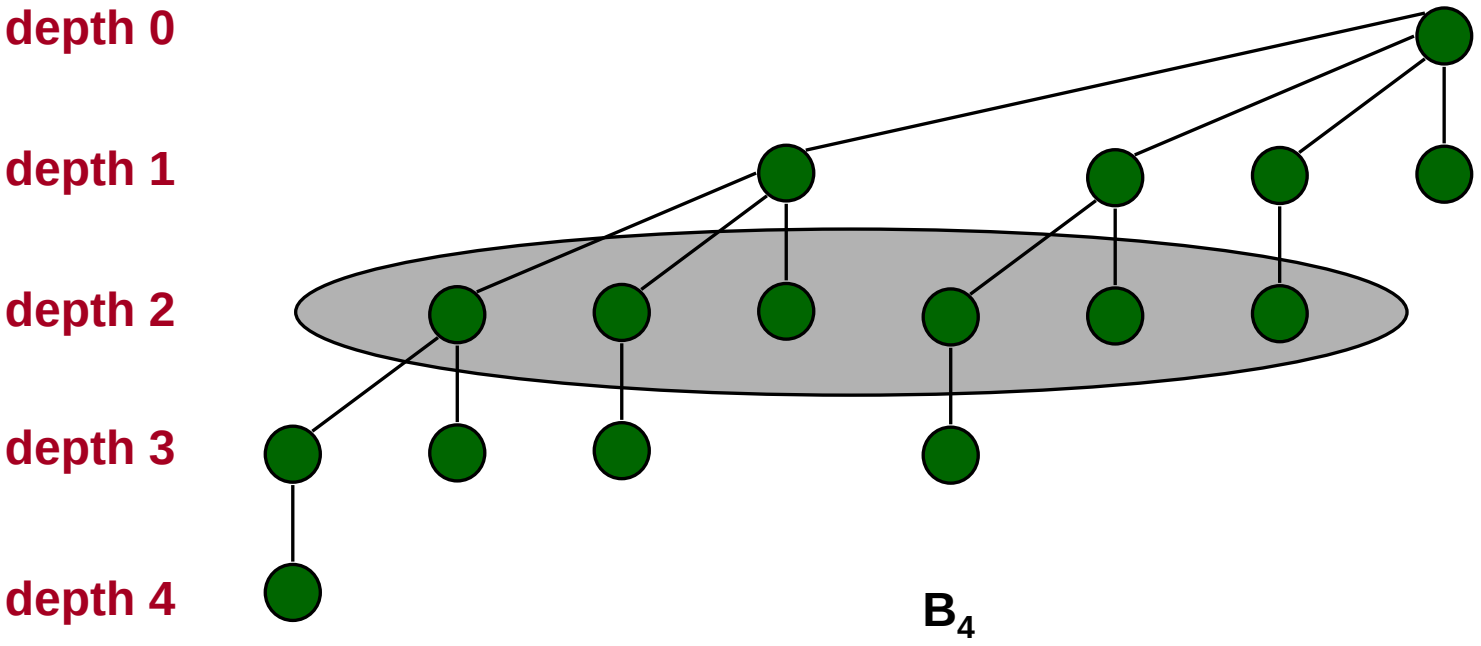


Binomial Tree

A property useful for naming the data structure.

- B_k has $\binom{k}{i}$ nodes at depth i.

$$\binom{4}{2} = 6$$

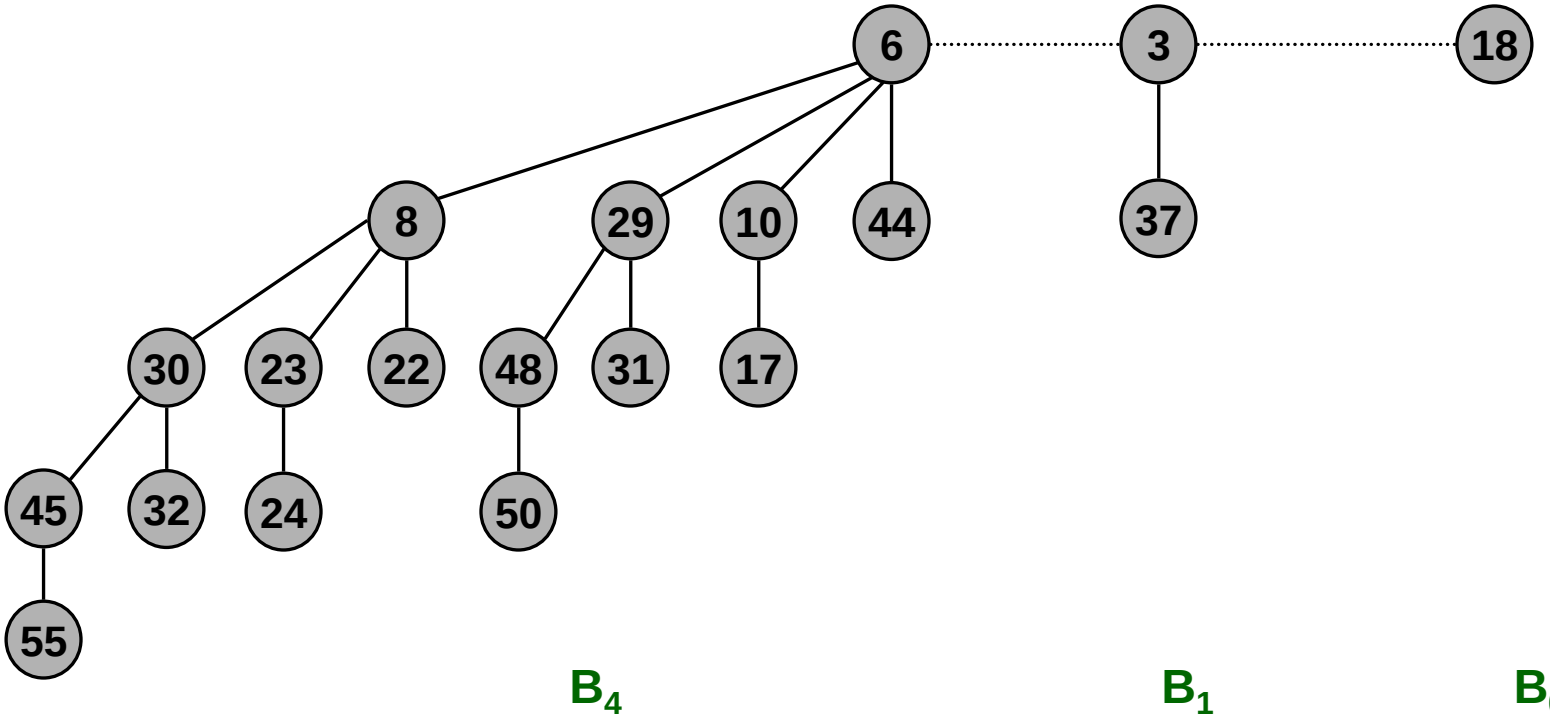


B_4

Binomial Heap

Binomial heap. Vuillemin, 1978.

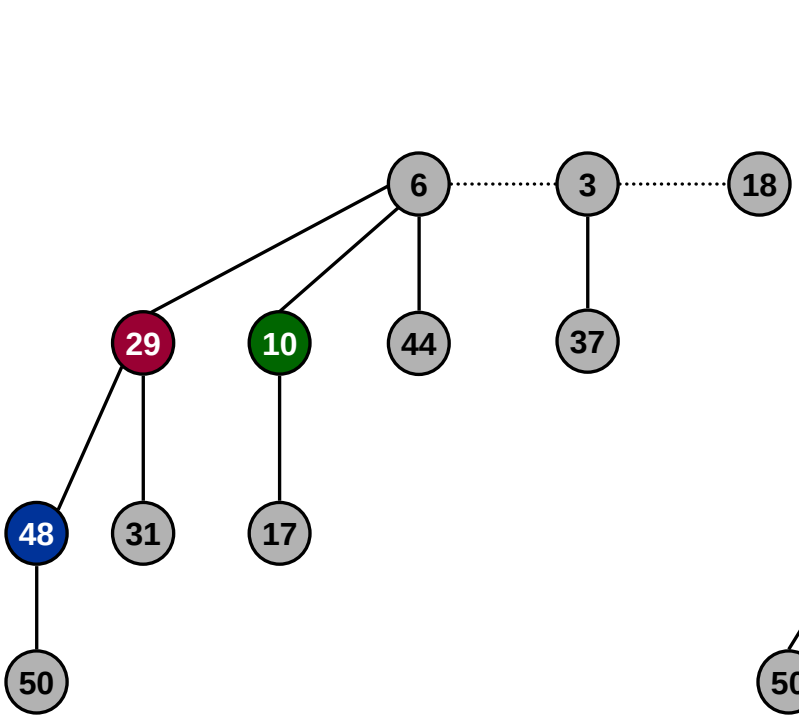
- Sequence of binomial trees that satisfy binomial heap property.
  - each tree is min-heap ordered
  - 0 or 1 binomial tree of order k



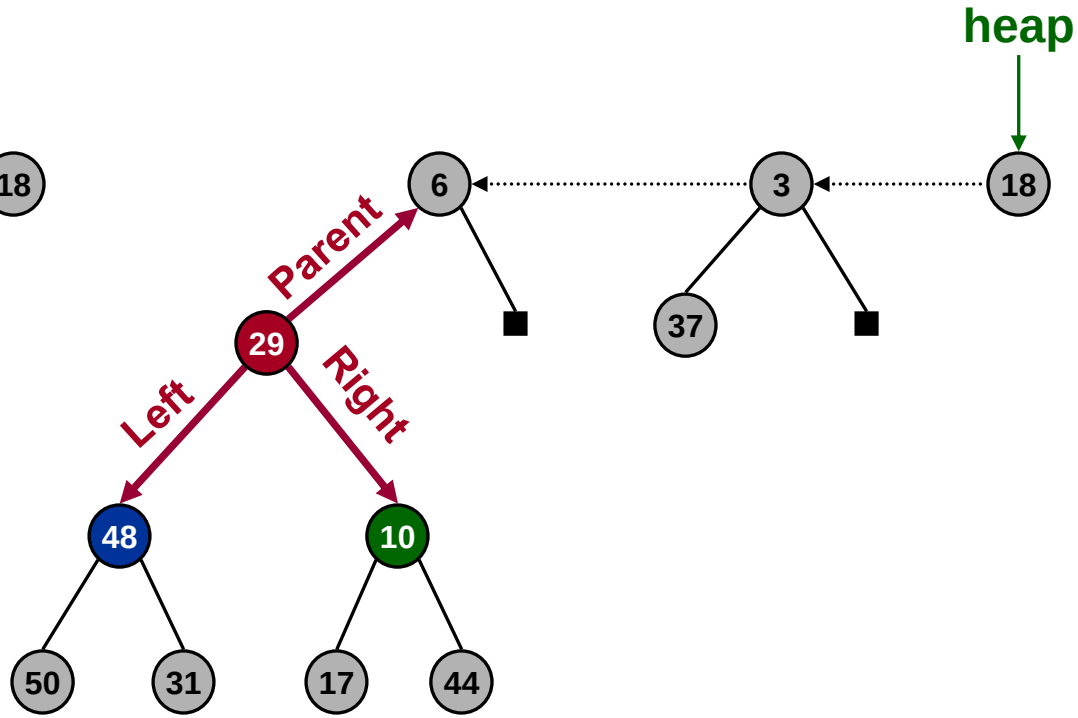
Binomial Heap: Implementation

Implementation.

- Represent trees using left-child, right sibling pointers.
  - three links per node (parent, left, right)
- Roots of trees connected with singly linked list.
  - degrees of trees strictly decreasing from left to right



Binomial Heap

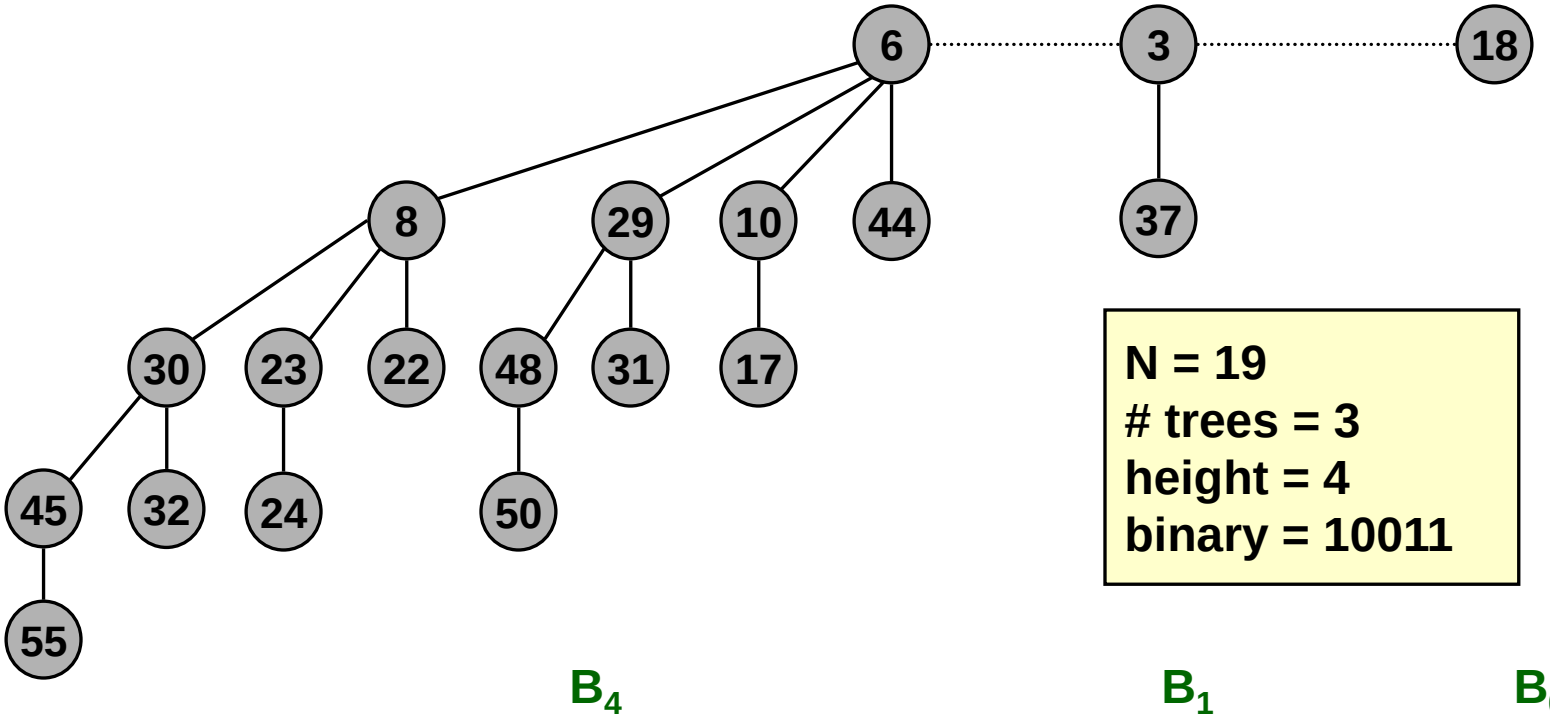


Leftist Power-of-2 Heap

Binomial Heap: Properties

Properties of N-node binomial heap.

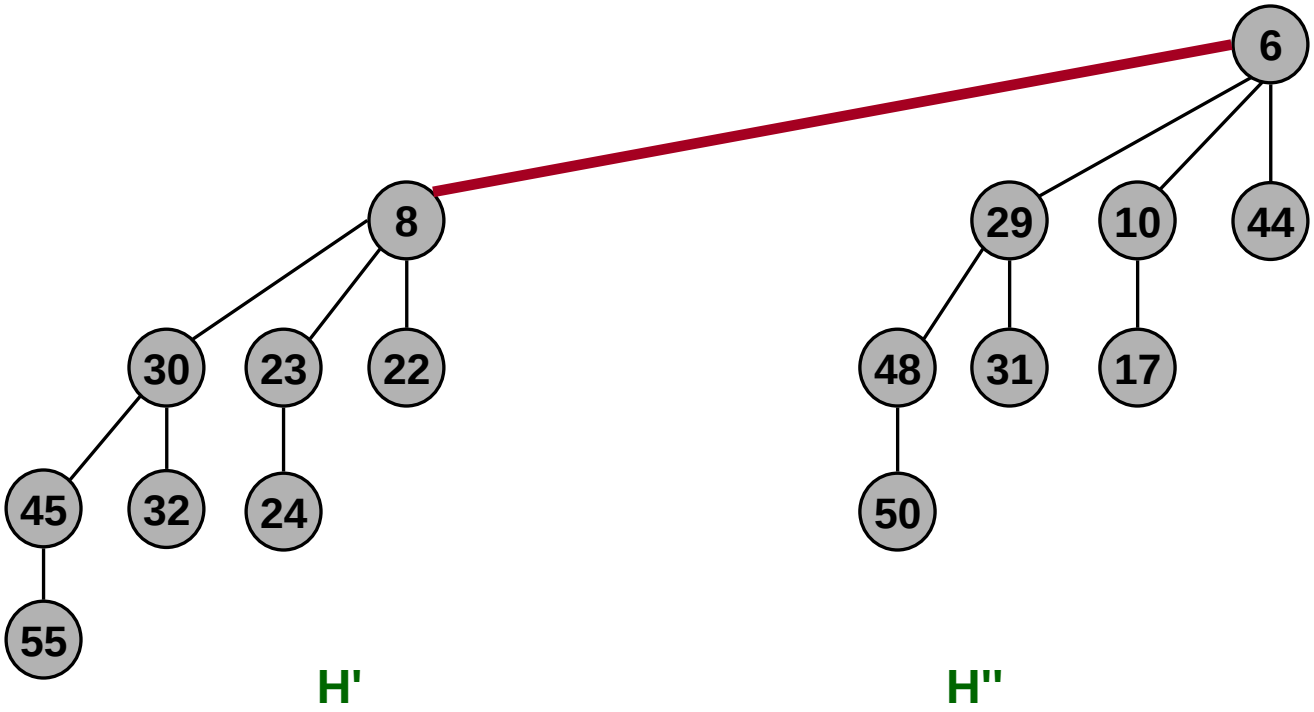
- Min key contained in root of $B_0, B_1, \ldots, B_k$.
- Contains binomial tree B_i iff $b_i = 1$ where $b_n \cdot b_2 b_1 b_0$ is binary representation of N.
- At most $\lfloor \log_2 N \rfloor + 1$ binomial trees.
- Height $\leq \lfloor \log_2 N \rfloor$.



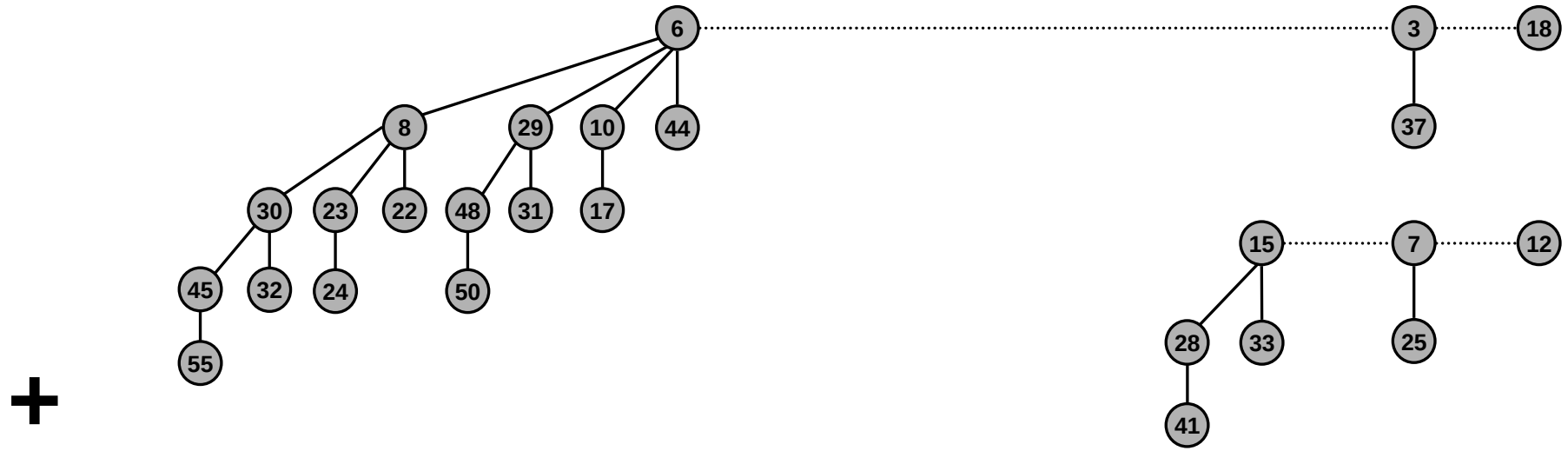
Binomial Heap: Union

Create heap H that is union of heaps H' and H''.

- **"Mergeable heaps."**
- **Easy if H' and H'' are each order k binomial trees.**
  - **connect roots of H' and H''**
  - **choose smaller key to be root of H**



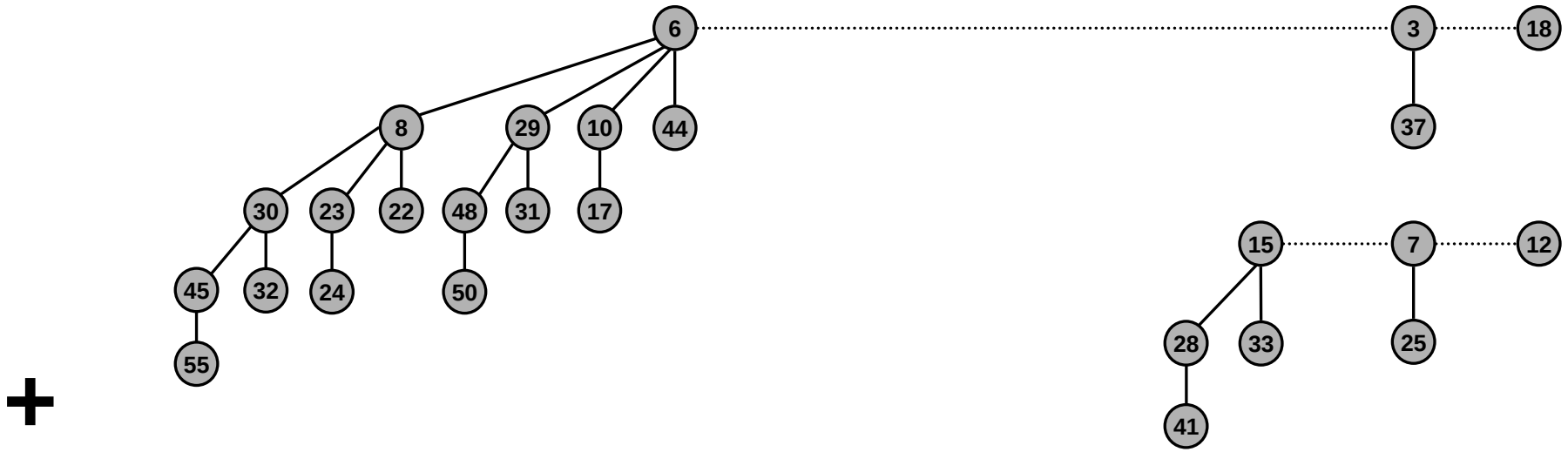
Binomial Heap: Union



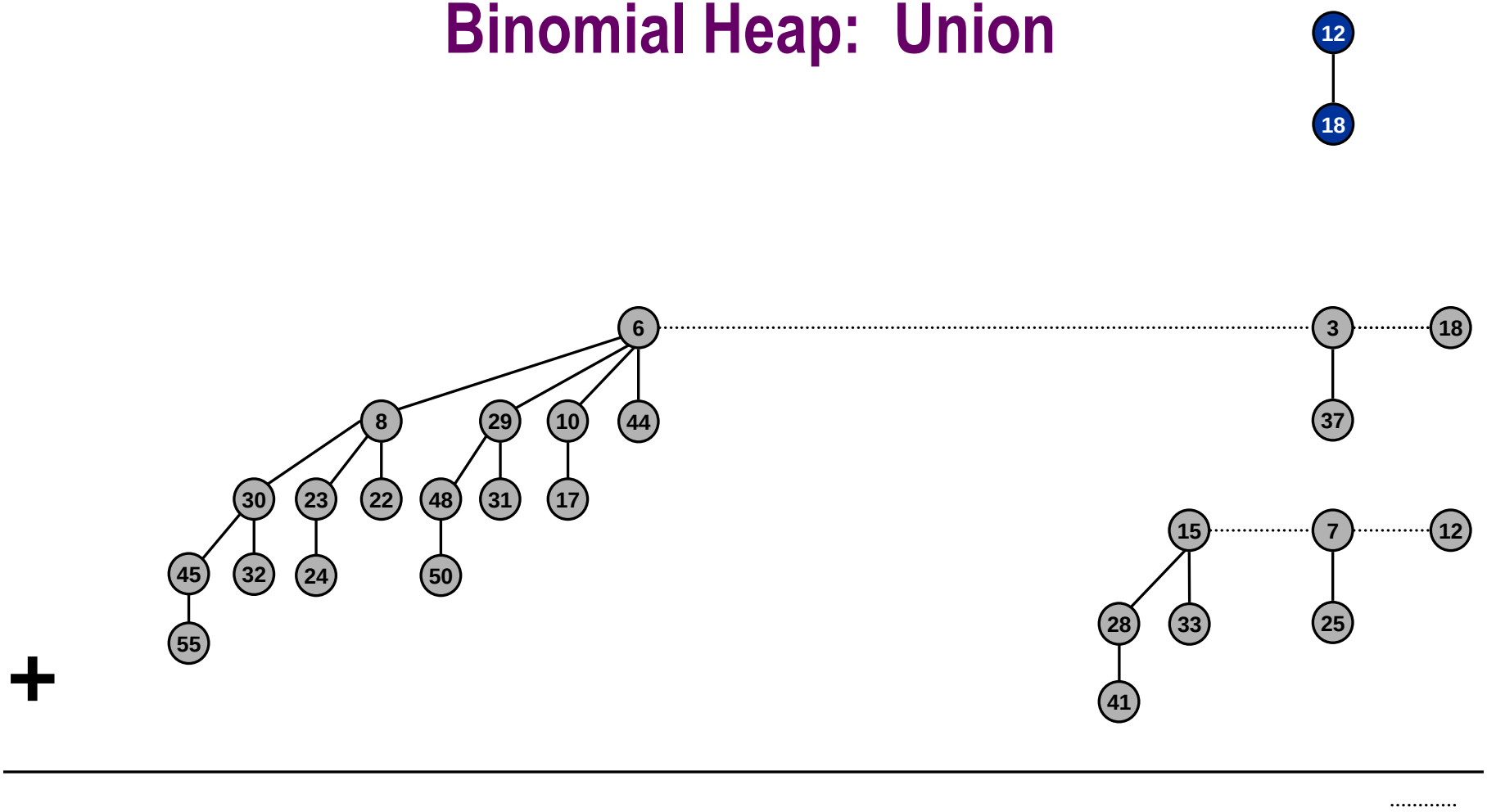
19 + 7 = 26

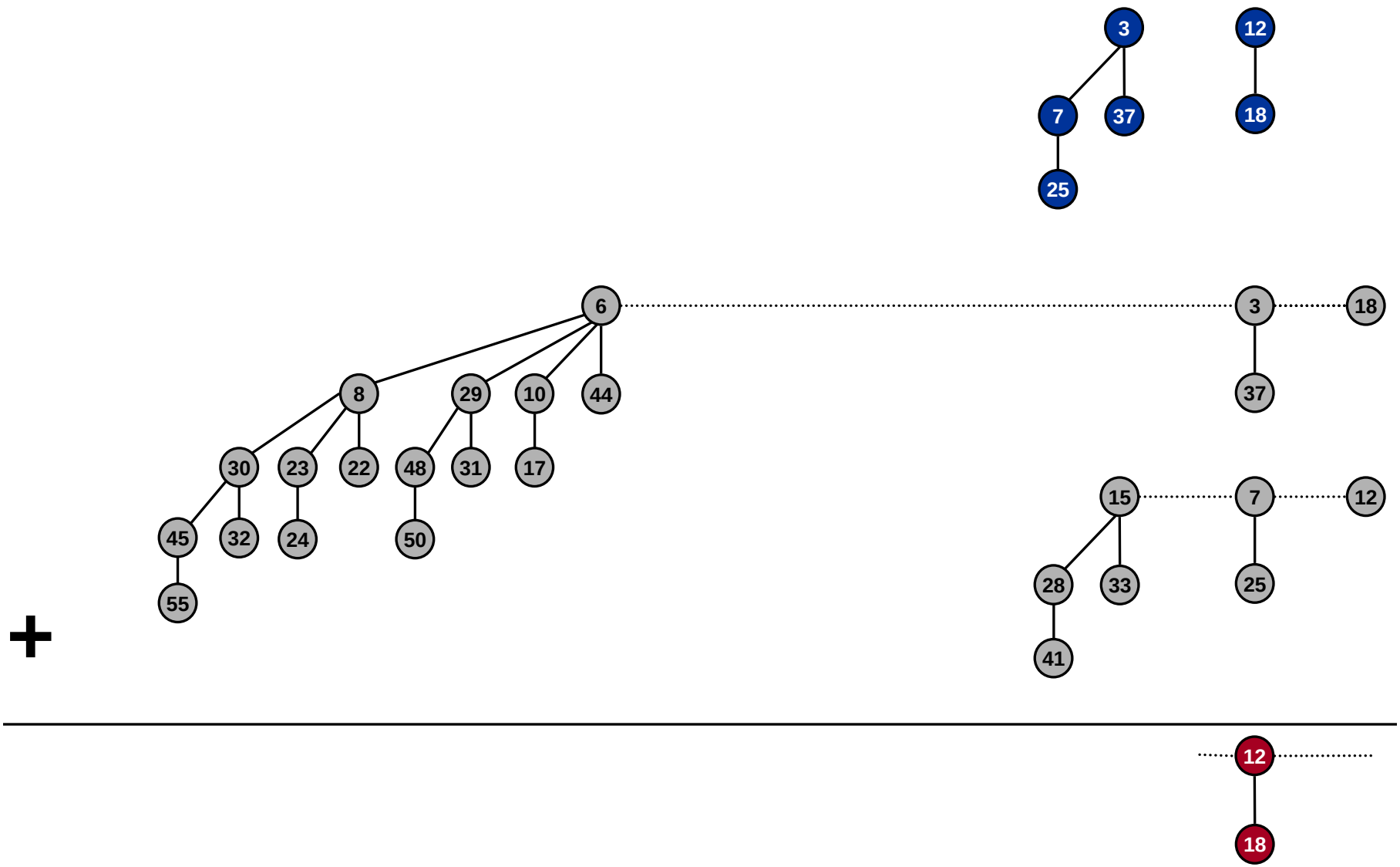
		1	1	1	
	1	0	0	1	1
+	0	0	1	1	1
	1	1	0	1	0

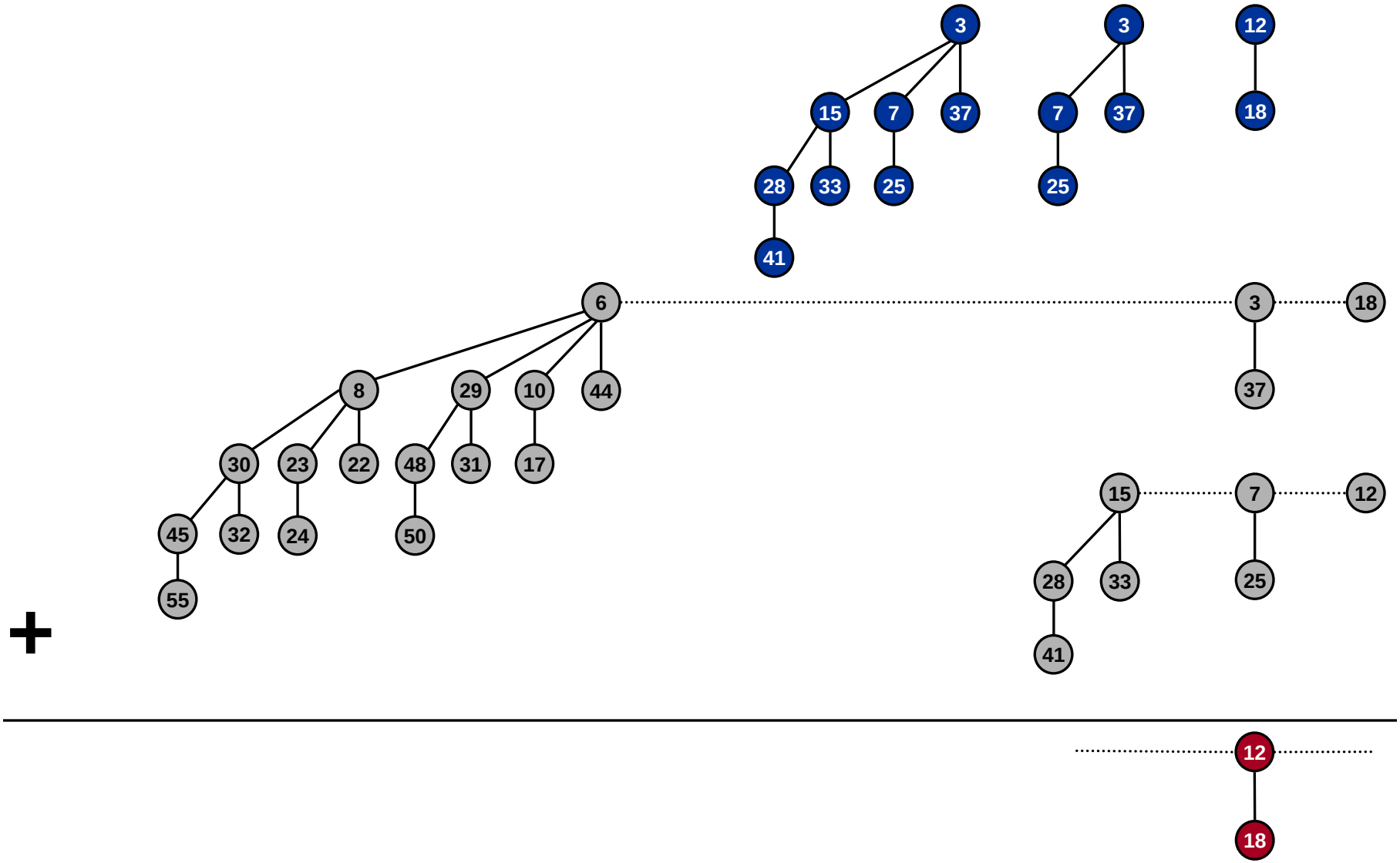
Binomial Heap: Union

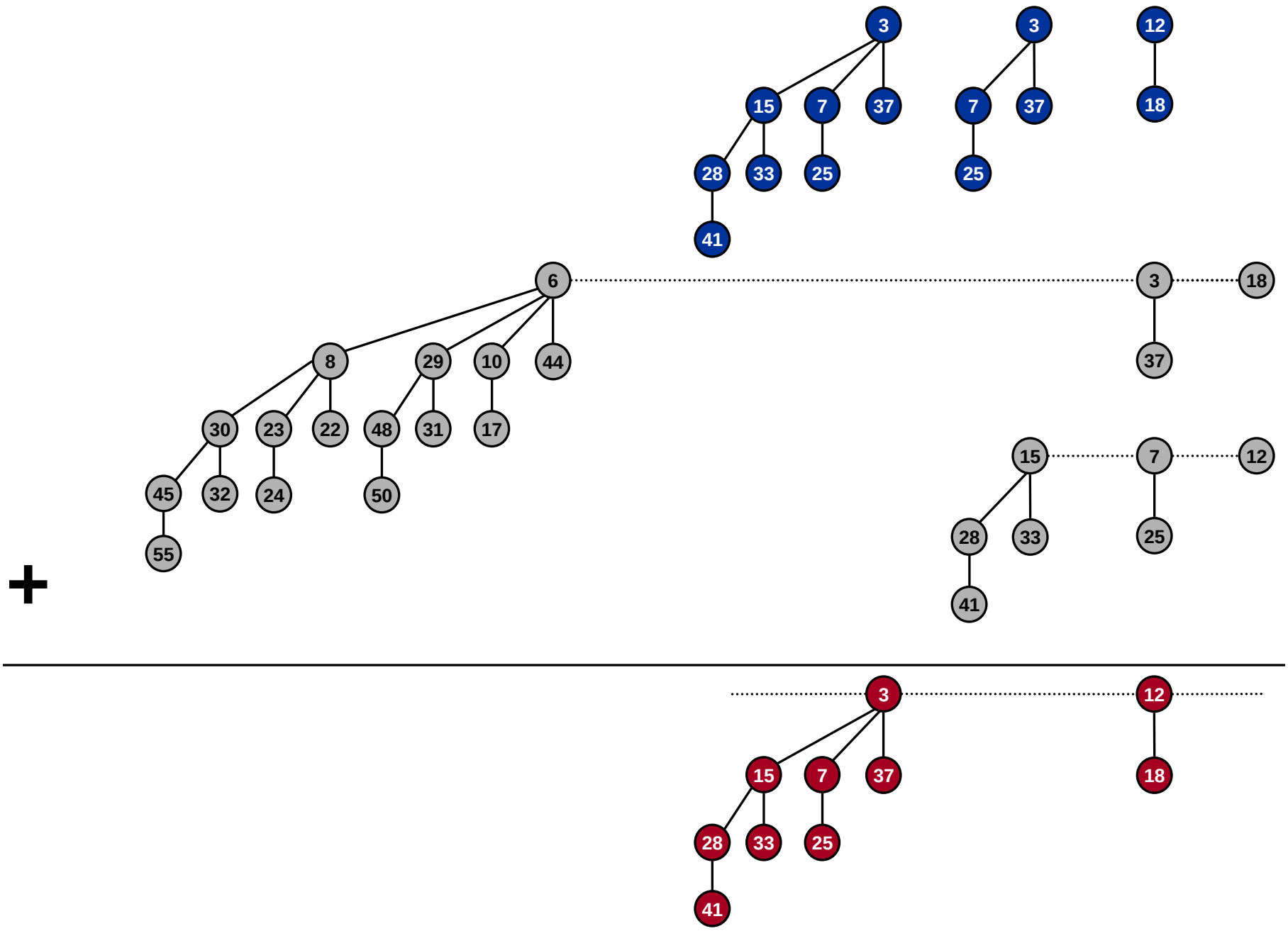


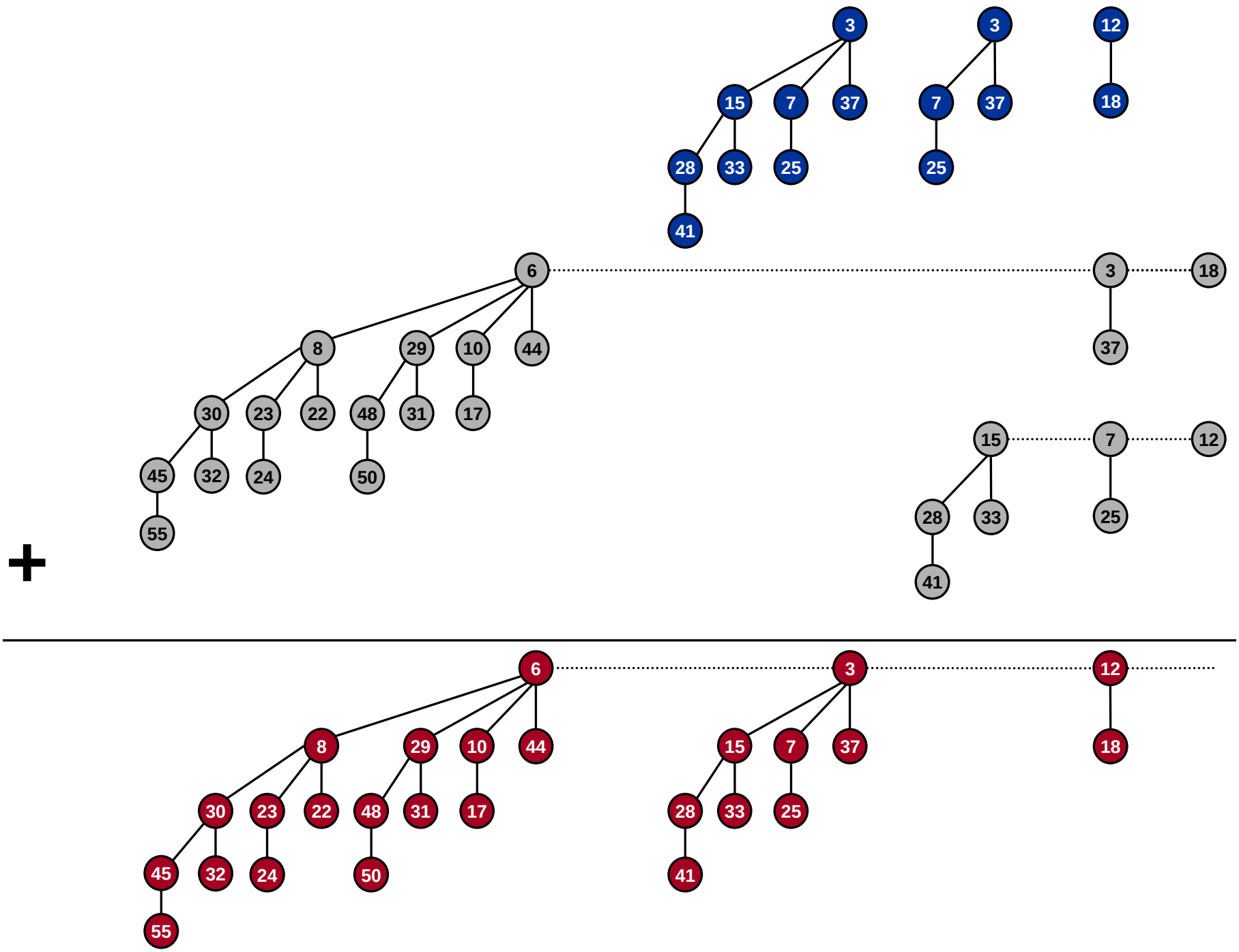
Binomial Heap: Union




3
12
7
37
18
25
6
3
18
8
29
10
44
37
30
23
22
48
31
17
15
7
12
45
32
24
50
28
33
25
55
41
+
12
18


3
15
7
37
28
33
25
41
3
7
37
25
12
18
6
8
29
10
44
30
23
22
48
31
17
45
32
24
50
55
3
18
37
15
7
12
28
33
25
41
12
18


3
15
7
37
28
33
25
41
3
7
37
25
12
18
6
8
29
10
44
30
23
22
48
31
17
45
32
24
50
55
3
18
37
15
7
12
28
33
25
41
+
3
15
7
37
28
33
25
41
12
18


3
15
7
37
28
33
25
41
3
7
37
25
12
18
6
8
29
10
44
30
23
22
48
31
17
45
32
24
50
55
3
18
37
15
7
12
28
33
25
41
+
6
8
29
10
44
30
23
22
48
31
17
45
32
24
50
55
3
15
7
37
28
33
25
41
12
18

Binomial Heap: Union

Create heap H that is union of heaps H' and H''.

- **Analogous to binary addition.**

Running time. O(log N)

- **Proportional to number of trees in root lists $\leq 2(\lfloor \log_2 N \rfloor + 1)$.**

19 + 7 = 26

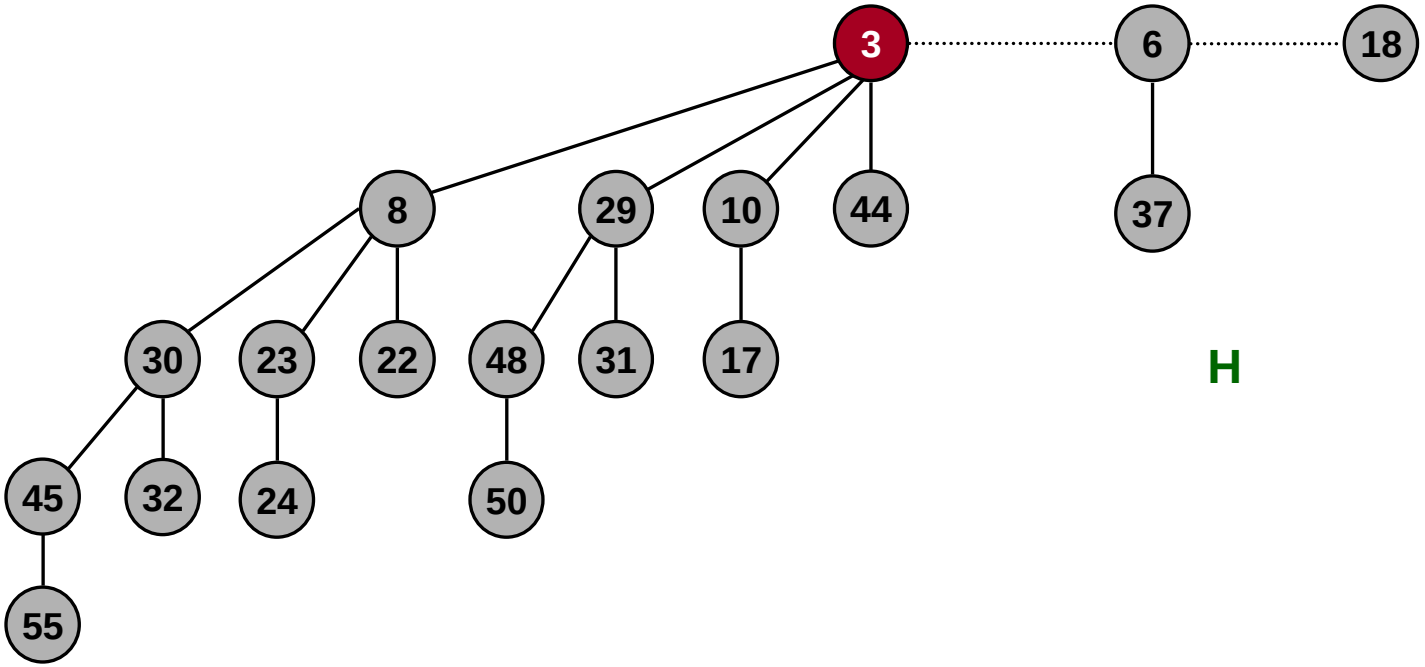
		1	1	1	
	1	0	0	1	1
+	0	0	1	1	1
	1	1	0	1	0

Binomial Heap: Delete Min

Delete node with minimum key in binomial heap H.

- **Find root x with min key in root list of H, and delete**
- **H' ← broken binomial trees**
- **H ← Union(H', H)**

Running time. **O(log N)**

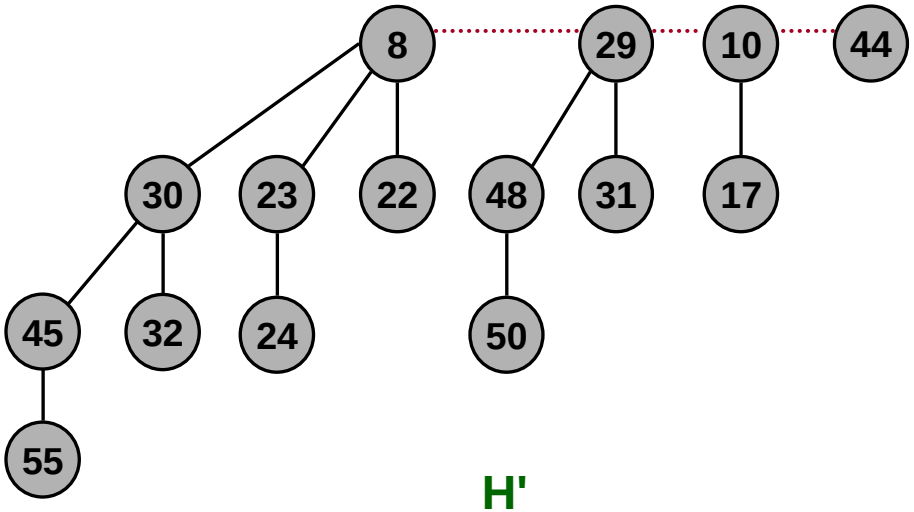


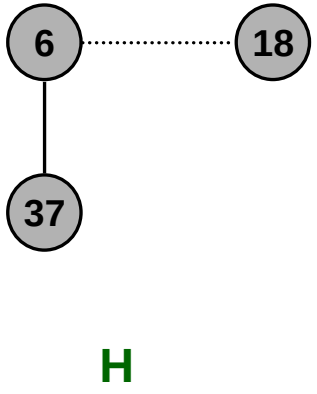
Binomial Heap: Delete Min

Delete node with minimum key in binomial heap H.

- **Find root x with min key in root list of H, and delete**
- **H' ← broken binomial trees**
- **H ← Union(H', H)**

Running time. **O(log N)**





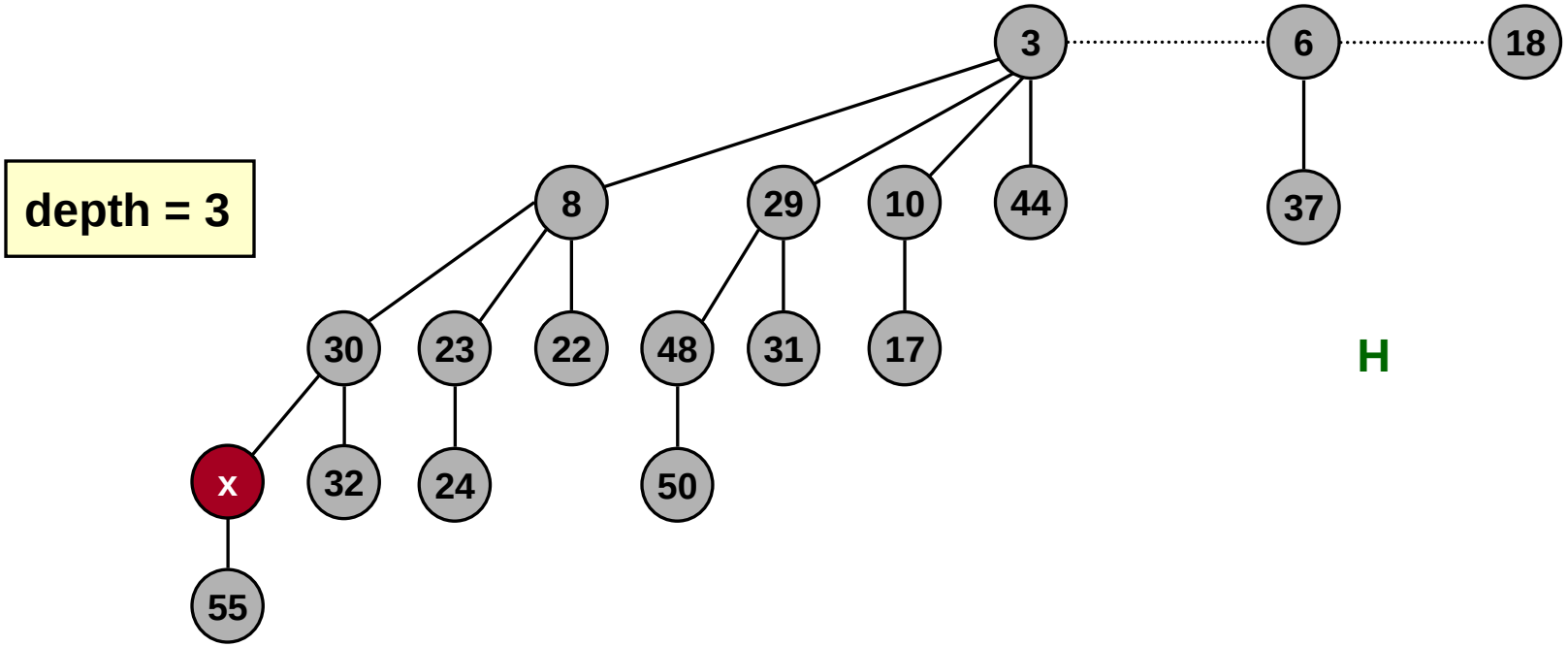
Binomial Heap: Decrease Key

Decrease key of node x in binomial heap H.

- Suppose x is in binomial tree B_k.
- Bubble node x up the tree if x is too small.

Running time. O(log N)

- Proportional to depth of node x $\leq \lfloor \log_2 N \rfloor$.



Binomial Heap: Delete

Delete node x in binomial heap H.

- **Decrease key of x to -∞.**
- **Delete min.**

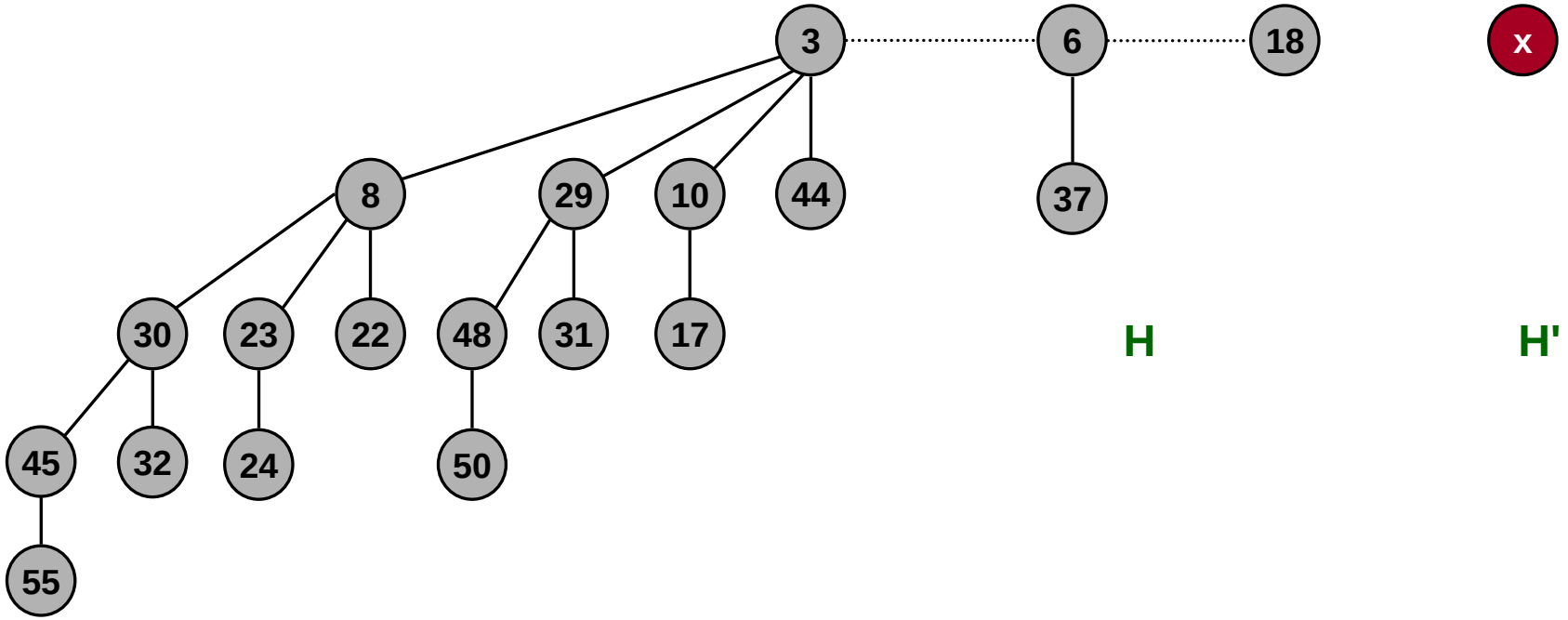
Running time. **O(log N)**

Binomial Heap: Insert

Insert a new node x into binomial heap H.

- **H' ← MakeHeap(x)**
- **H ← Union(H', H)**

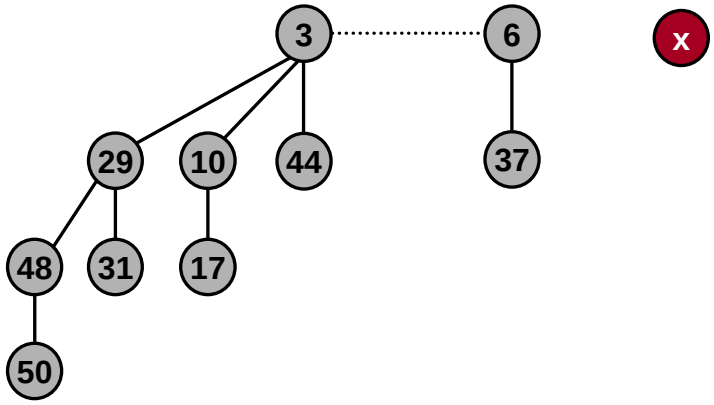
Running time. O(log N)



Binomial Heap: Sequence of Inserts

Insert a new node x into binomial heap H.

- If N = `.......0`, then only 1 steps.
- If N = `......01`, then only 2 steps.
- If N = `.....011`, then only 3 steps.
- If N = `....0111`, then only 4 steps.



Inserting 1 item can take Ω(log N) time.

- If N = `11...111`, then $\log_2 N$ steps.

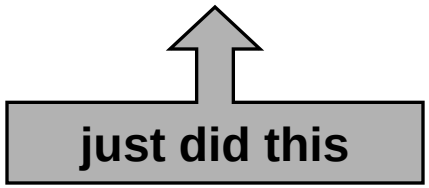
But, inserting sequence of N items takes O(N) time!

- (N/2)(1) + (N/4)(2) + (N/8)(3) + . . . ≤ 2N
- Amortized analysis.
- Basis for getting most operations down to constant time.

$$\sum_{n=1}^{N} \frac{n}{2^n} = 2 - \frac{N}{2^N} - \frac{1}{2^{N-1}} \leq 2$$

Priority Queues

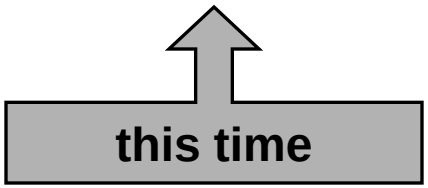
		Heaps			
Operation	**Linked List**	**Binary**	**Binomial**	**Fibonacci ***	**Relaxed**
make-heap	1	1	1	1	1
insert	1	log N	log N	1	1
find-min	N	1	log N	1	1
delete-min	N	log N	log N	log N	log N
union	1	N	log N	1	1
decrease-key	1	log N	log N	1	1
delete	N	log N	log N	log N	log N
is-empty	1	1	1	1	1



Priority Queues

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

† amortized



Fibonacci Heaps

Fibonacci heap history. Fredman and Tarjan (1986)

- Ingenious data structure and analysis.
- Original motivation: O(m + n log n) shortest path algorithm.
  - also led to faster algorithms for MST, weighted bipartite matching
- Still ahead of its time.

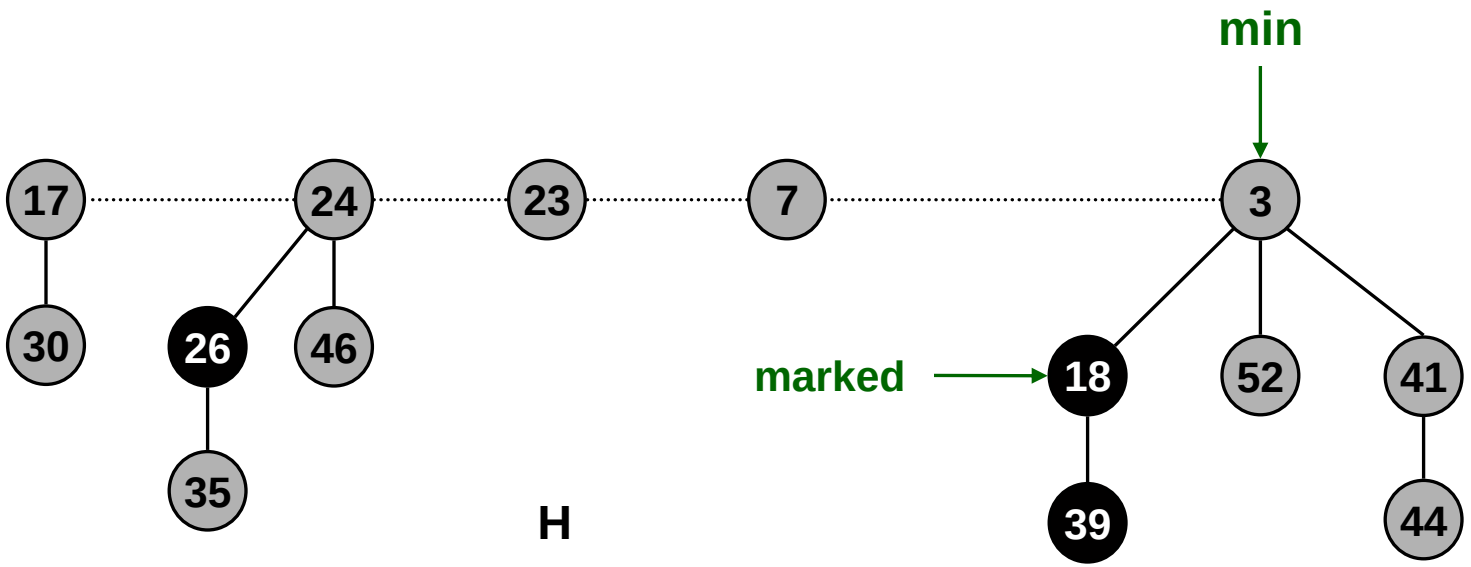
Fibonacci heap intuition.

- Similar to binomial heaps, but less structured.
- Decrease-key and union run in O(1) time.
- "Lazy" unions.

Fibonacci Heaps: Structure

Fibonacci heap.

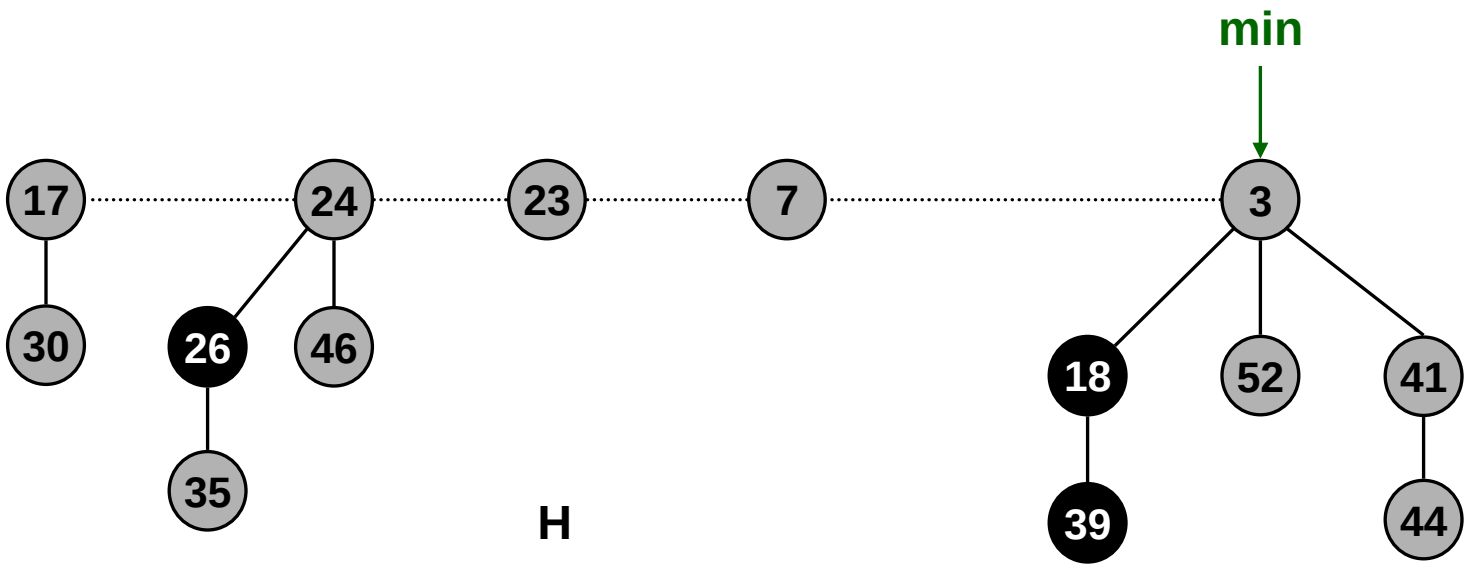
- **Set of min-heap ordered trees.**



Fibonacci Heaps: Implementation

Implementation.

- **Represent trees using left-child, right sibling pointers and circular, doubly linked list.**
  - **can quickly splice off subtrees**
- **Roots of trees connected with circular doubly linked list.**
  - **fast union**
- **Pointer to root of tree with min element.**
  - **fast find-min**



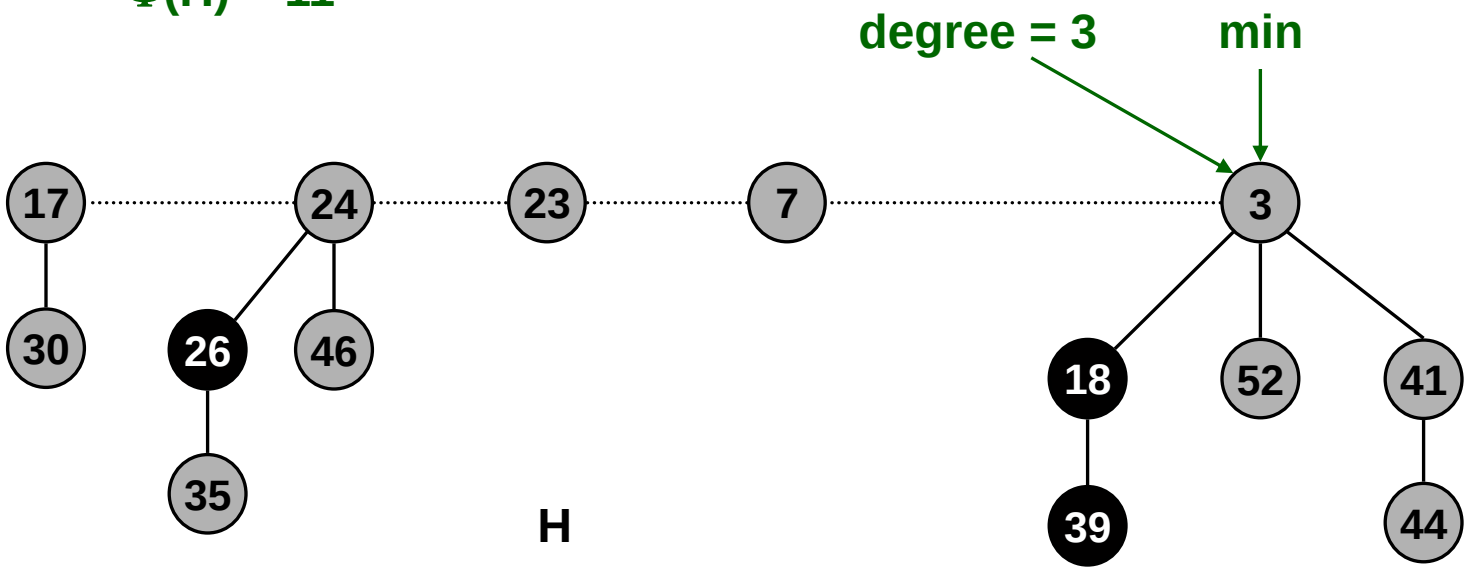
Fibonacci Heaps: Potential Function

Key quantities.

- **Degree[x] = degree of node x.**
- **Mark[x] = mark of node x (black or gray).**
- **t(H) = # trees.**
- **m(H) = # marked nodes.**
- **Φ(H) = t(H) + 2m(H) = potential function.**

t(H) = 5, m(H) = 3

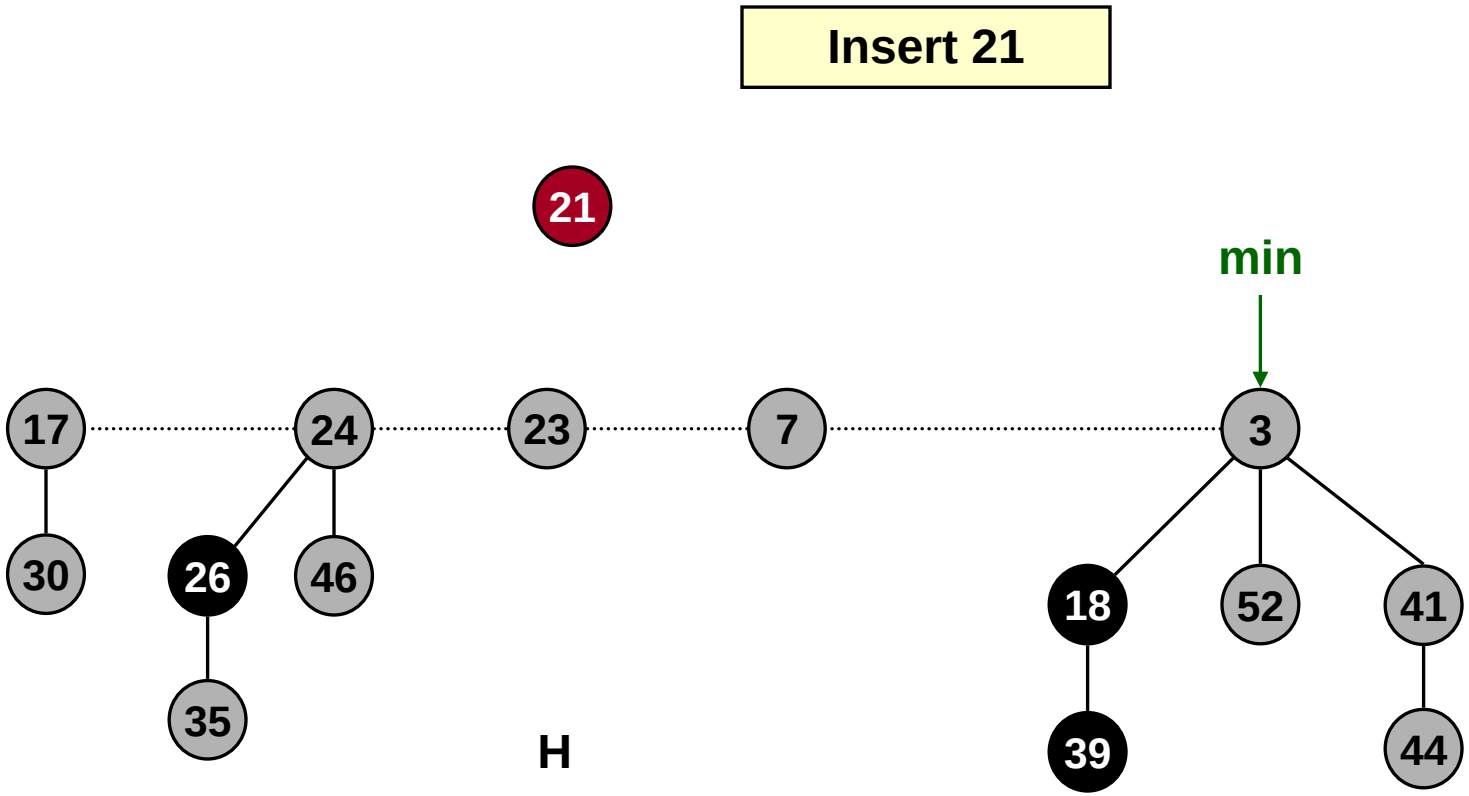
Φ(H) = 11



Fibonacci Heaps: Insert

Insert.

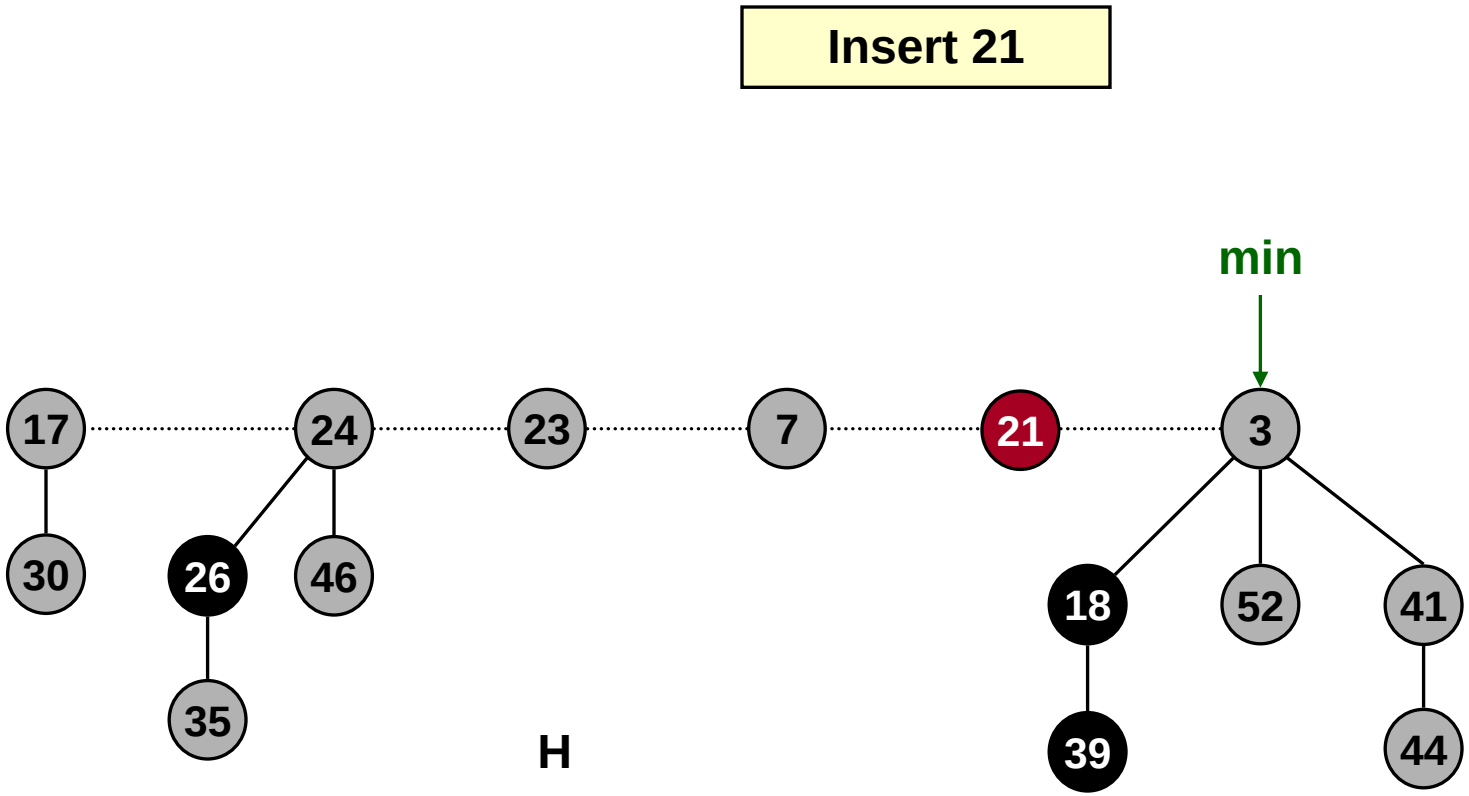
- Create a new singleton tree.
- Add to left of min pointer.
- Update min pointer.



Fibonacci Heaps: Insert

Insert.

- Create a new singleton tree.
- Add to left of min pointer.
- Update min pointer.



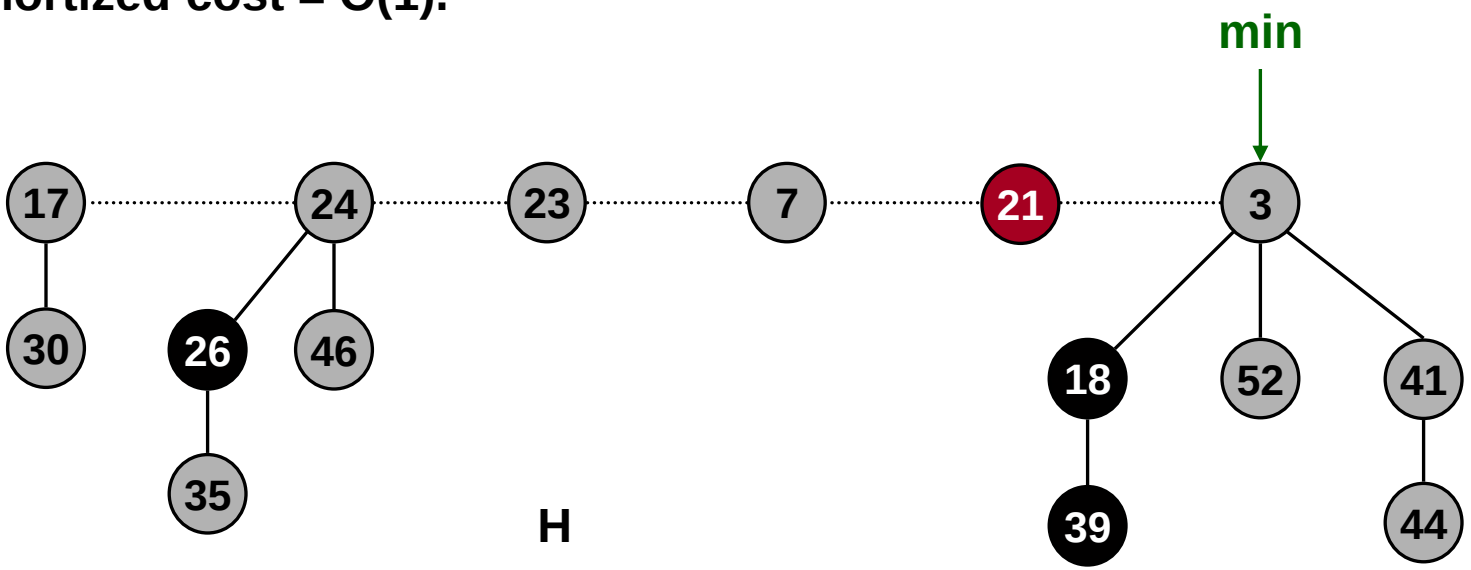
Fibonacci Heaps: Insert

Insert.

- Create a new singleton tree.
- Add to left of min pointer.
- Update min pointer.

Running time. O(1) amortized

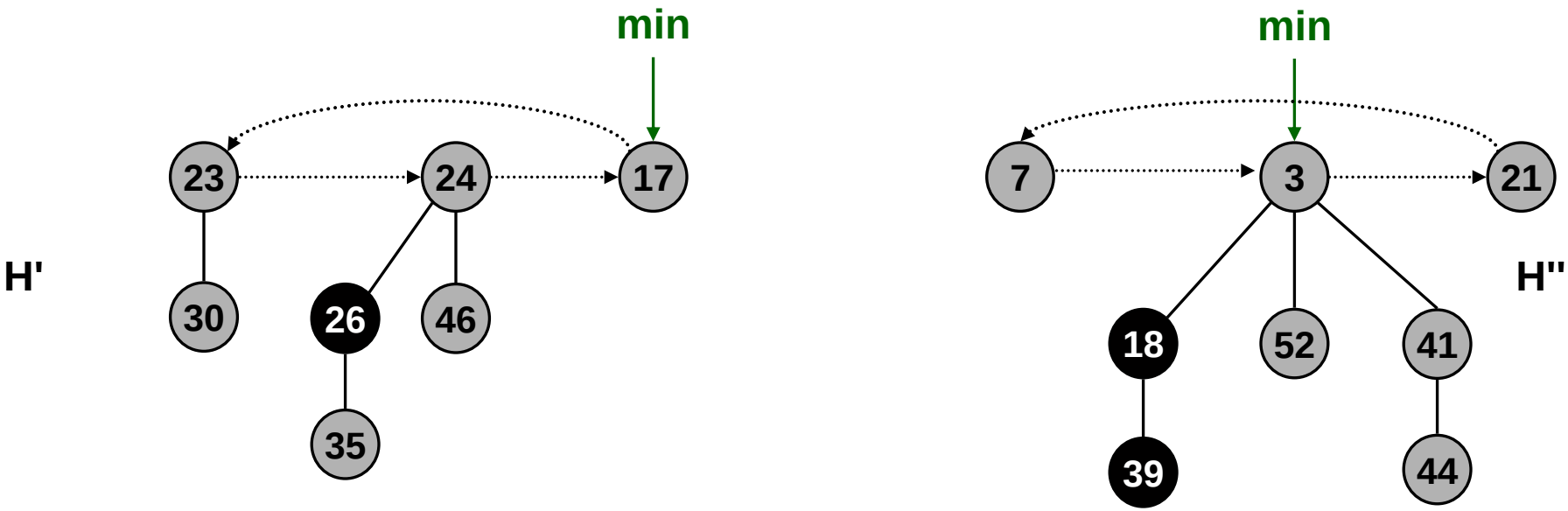
- Actual cost = O(1).
- Change in potential = +1.
- Amortized cost = O(1).



Fibonacci Heaps: Union

Union.

- Concatenate two Fibonacci heaps.
- Root lists are circular, doubly linked lists.



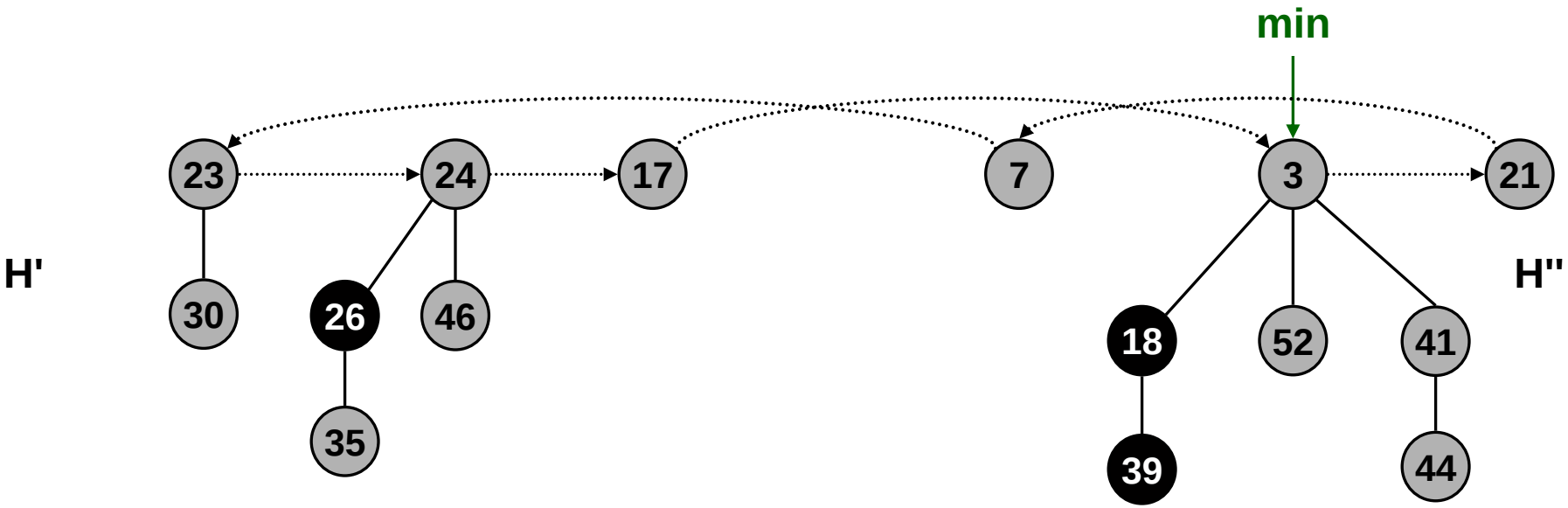
Fibonacci Heaps: Union

Union.

- Concatenate two Fibonacci heaps.
- Root lists are circular, doubly linked lists.

Running time. O(1) amortized

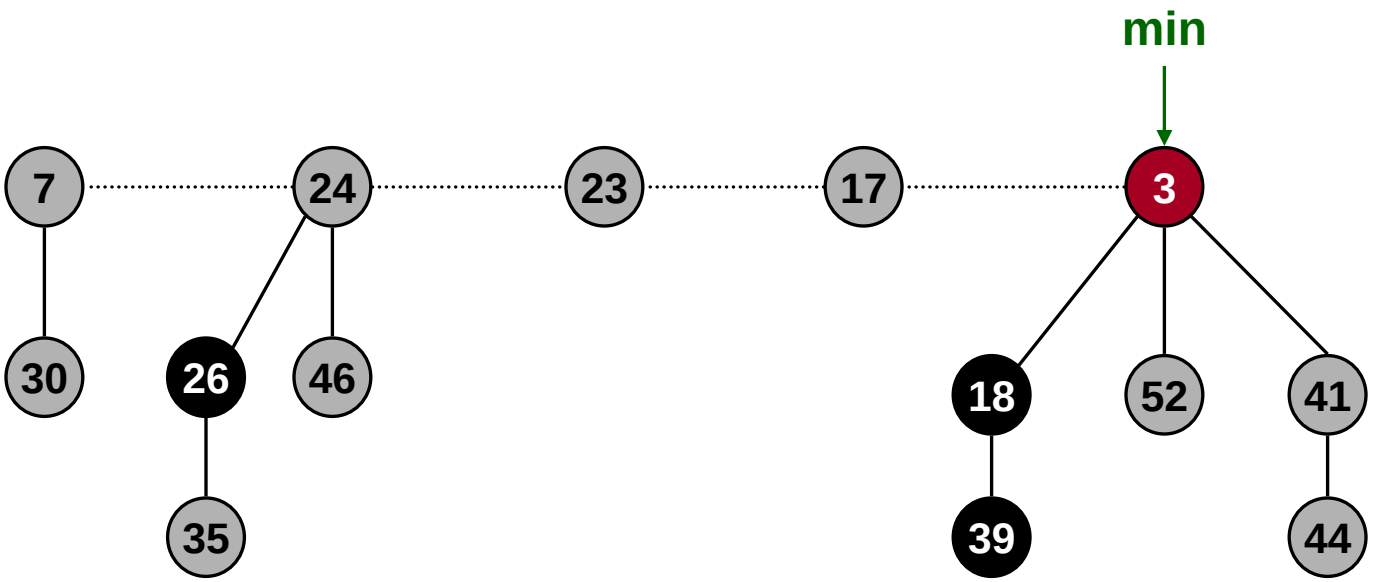
- Actual cost = O(1).
- Change in potential = 0.
- Amortized cost = O(1).



Fibonacci Heaps: Delete Min

Delete min.

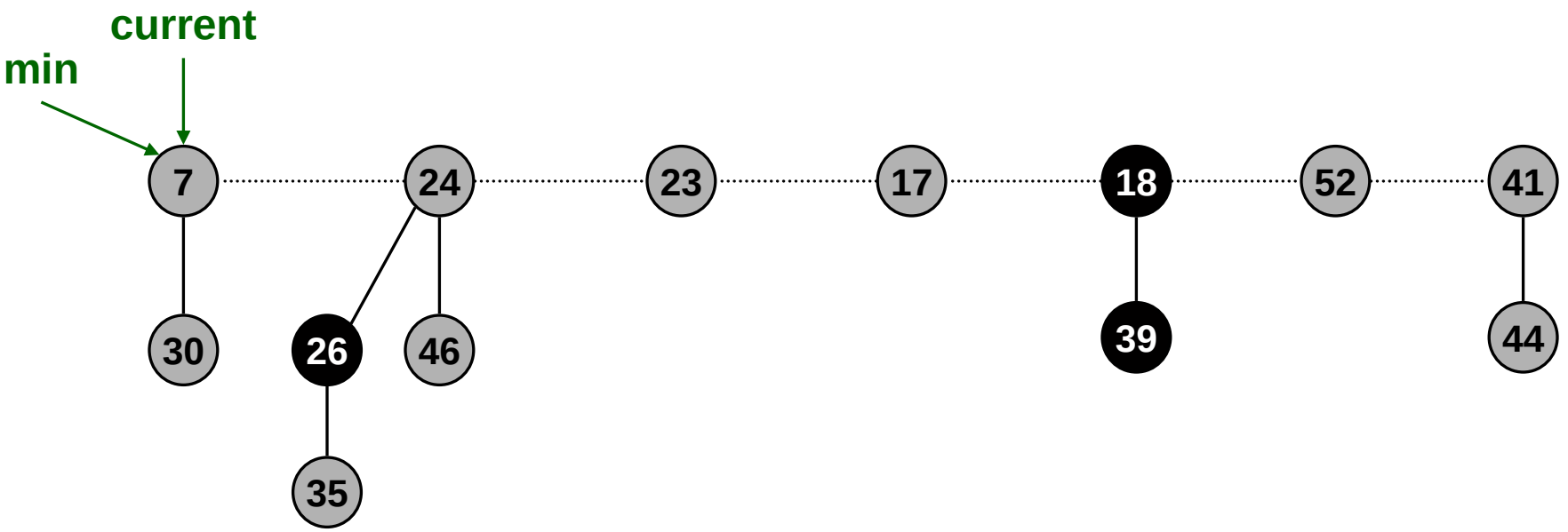
- **Delete min and concatenate its children into root list.**
- **Consolidate trees so that no two roots have same degree.**



Fibonacci Heaps: Delete Min

Delete min.

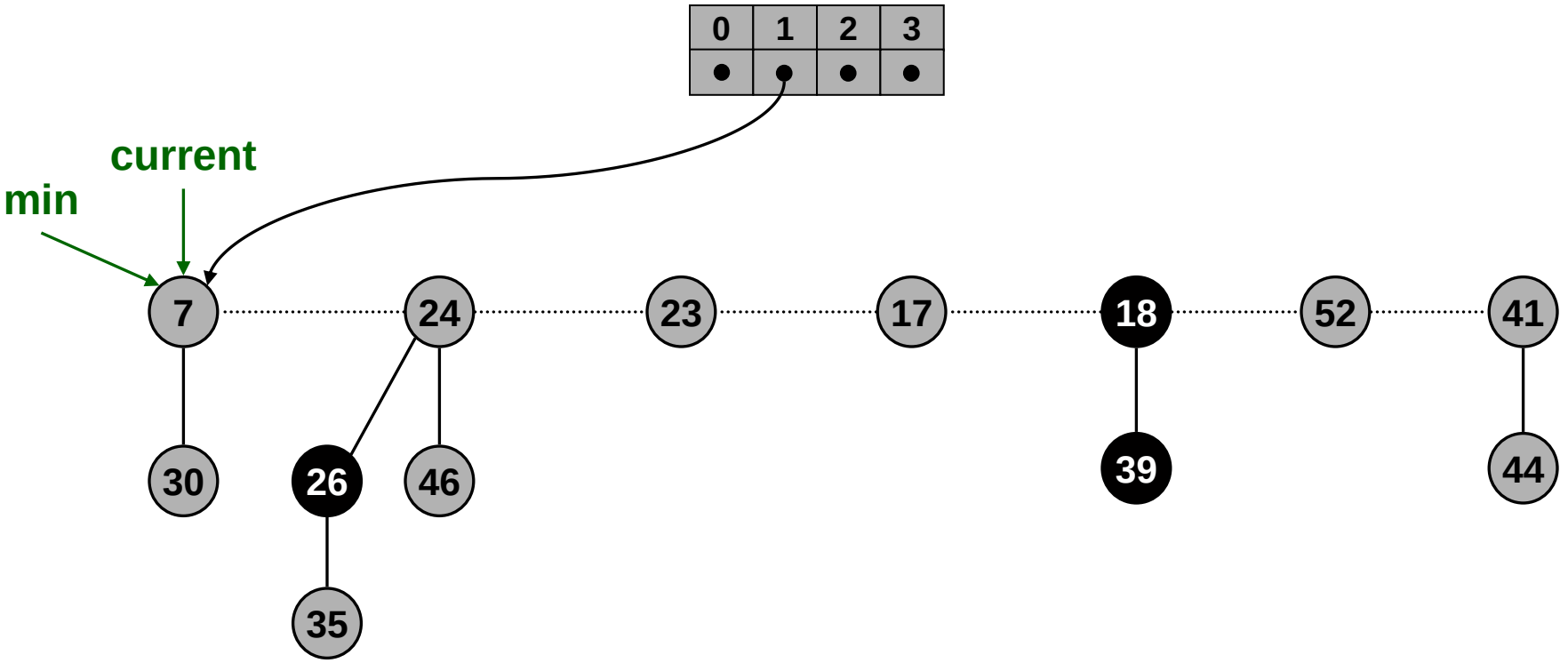
- Delete min and concatenate its children into root list.
- **Consolidate trees so that no two roots have same degree.**



Fibonacci Heaps: Delete Min

Delete min.

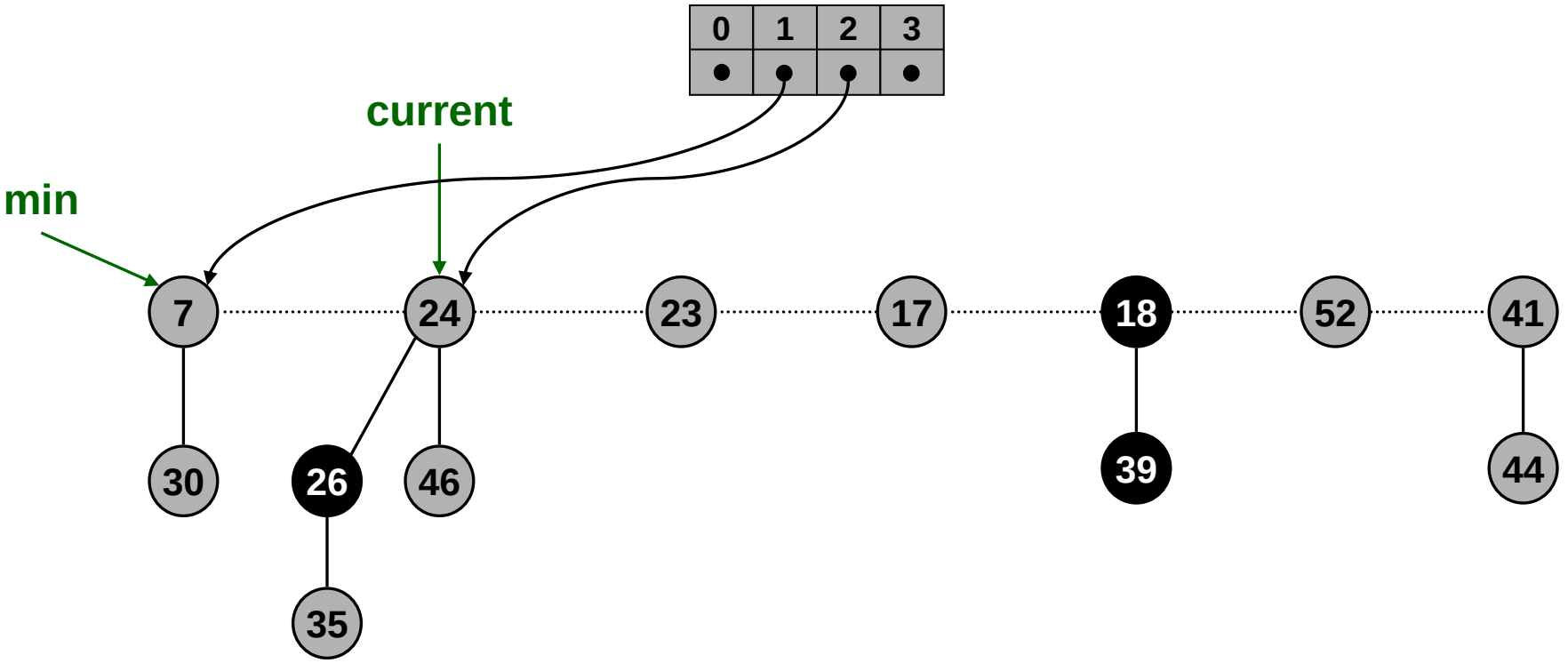
- Delete min and concatenate its children into root list.
- **Consolidate trees so that no two roots have same degree.**



Fibonacci Heaps: Delete Min

Delete min.

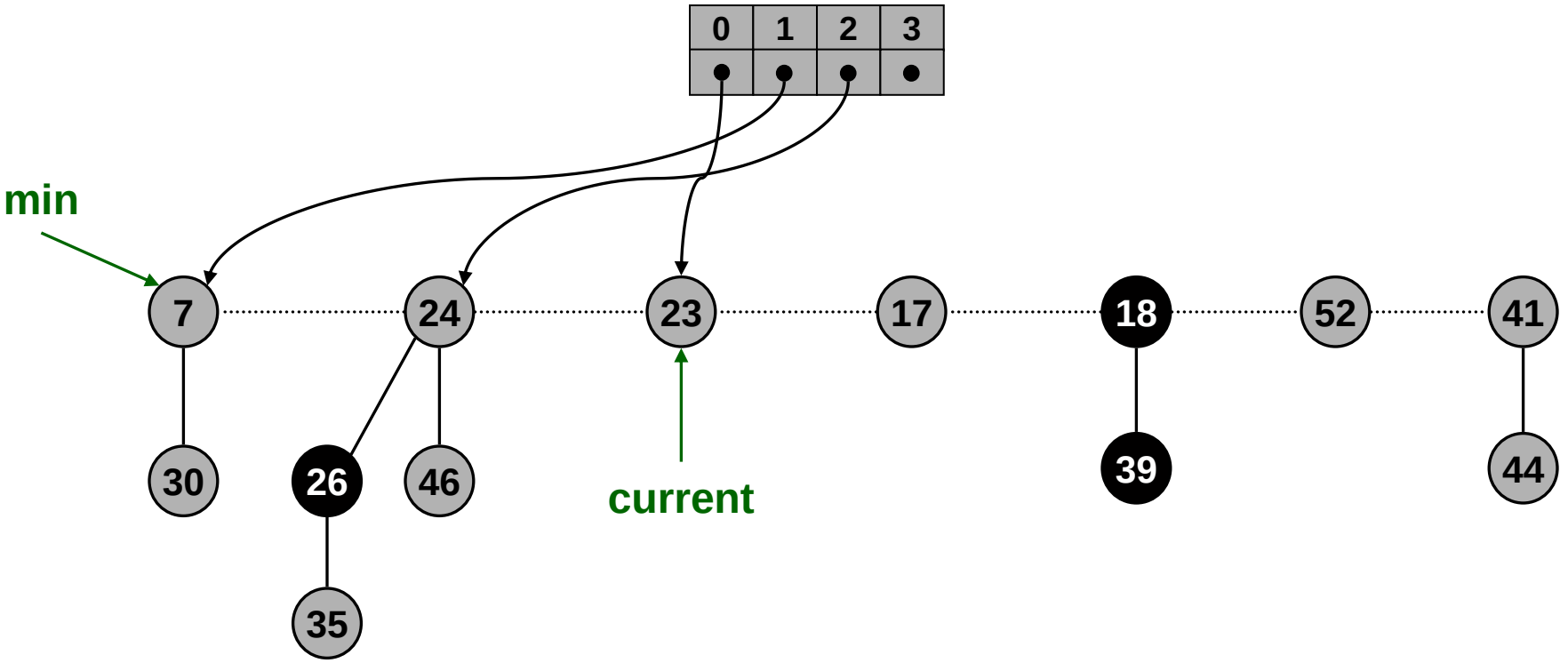
- Delete min and concatenate its children into root list.
- **Consolidate trees so that no two roots have same degree.**



Fibonacci Heaps: Delete Min

Delete min.

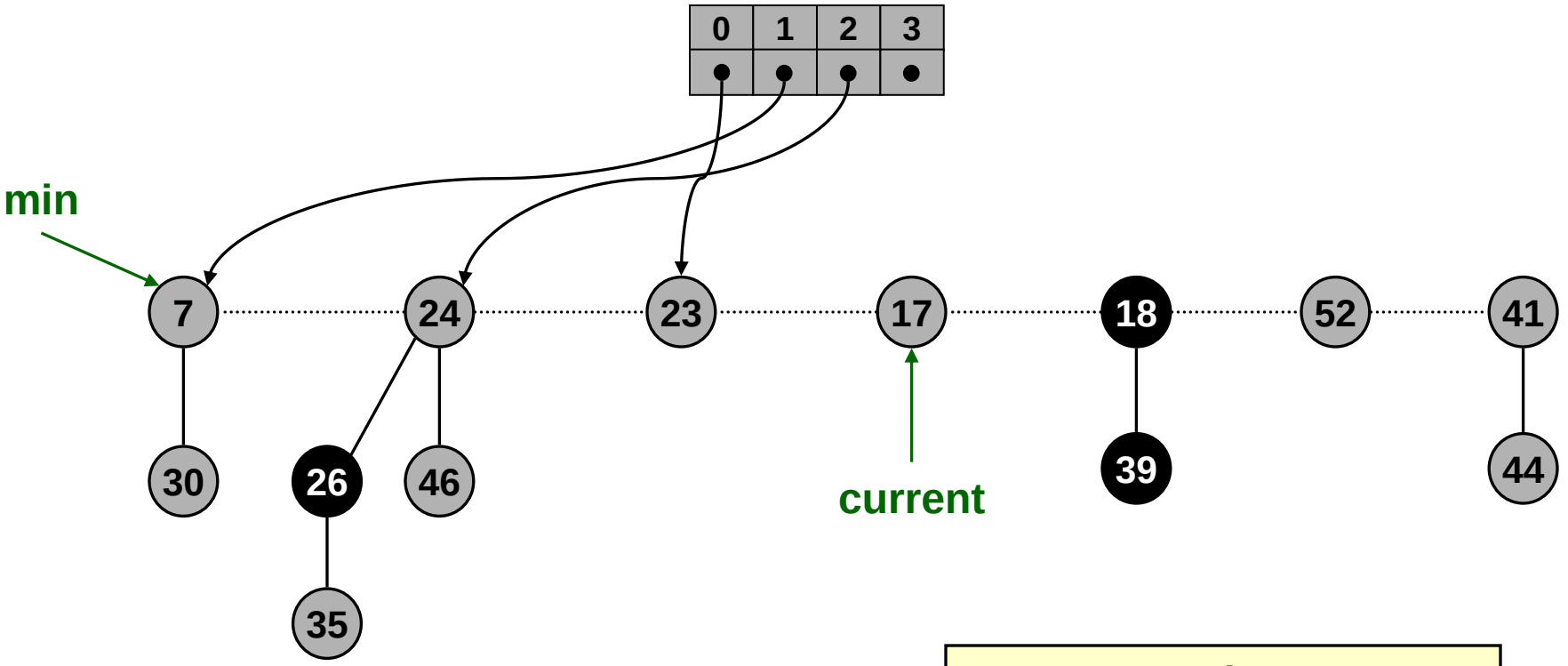
- Delete min and concatenate its children into root list.
- **Consolidate trees so that no two roots have same degree.**



Fibonacci Heaps: Delete Min

Delete min.

- Delete min and concatenate its children into root list.
- **Consolidate trees so that no two roots have same degree.**

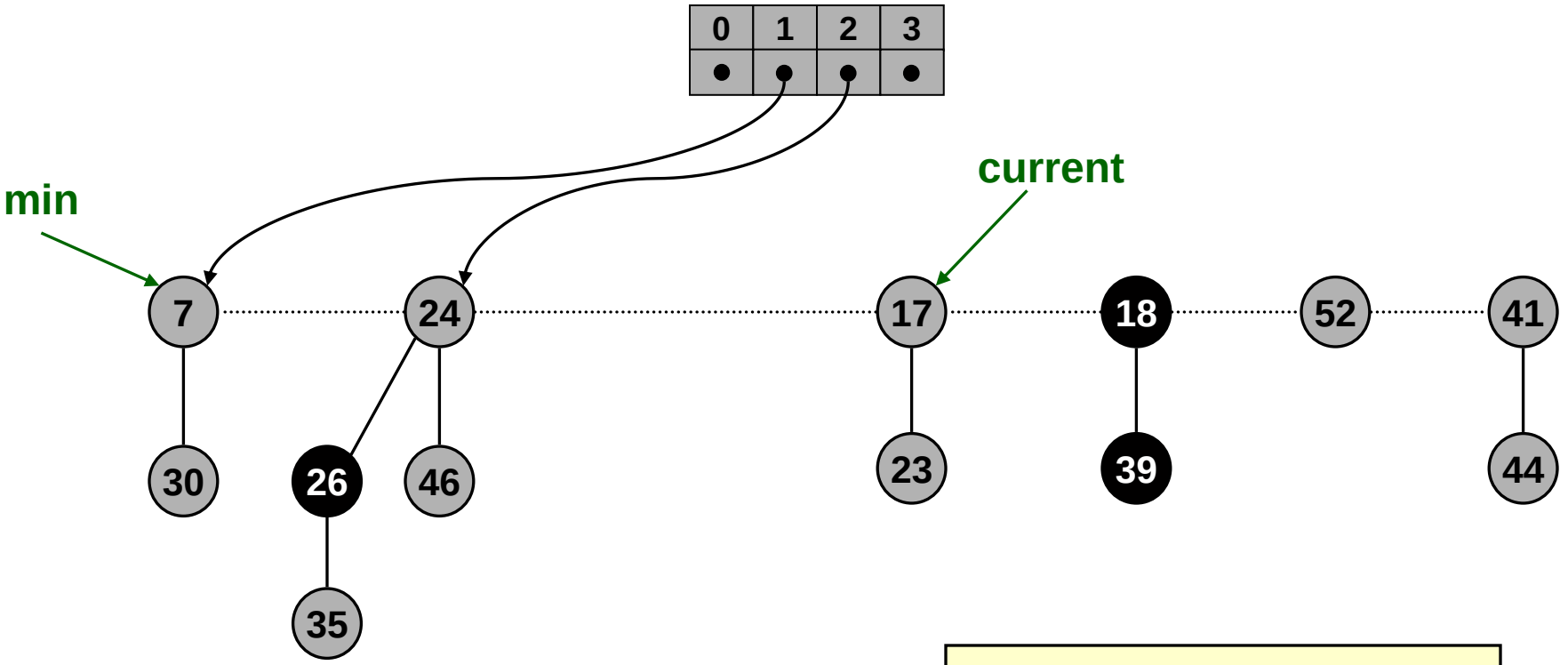


Merge 17 and 23 trees.

Fibonacci Heaps: Delete Min

Delete min.

- Delete min and concatenate its children into root list.
- **Consolidate trees so that no two roots have same degree.**

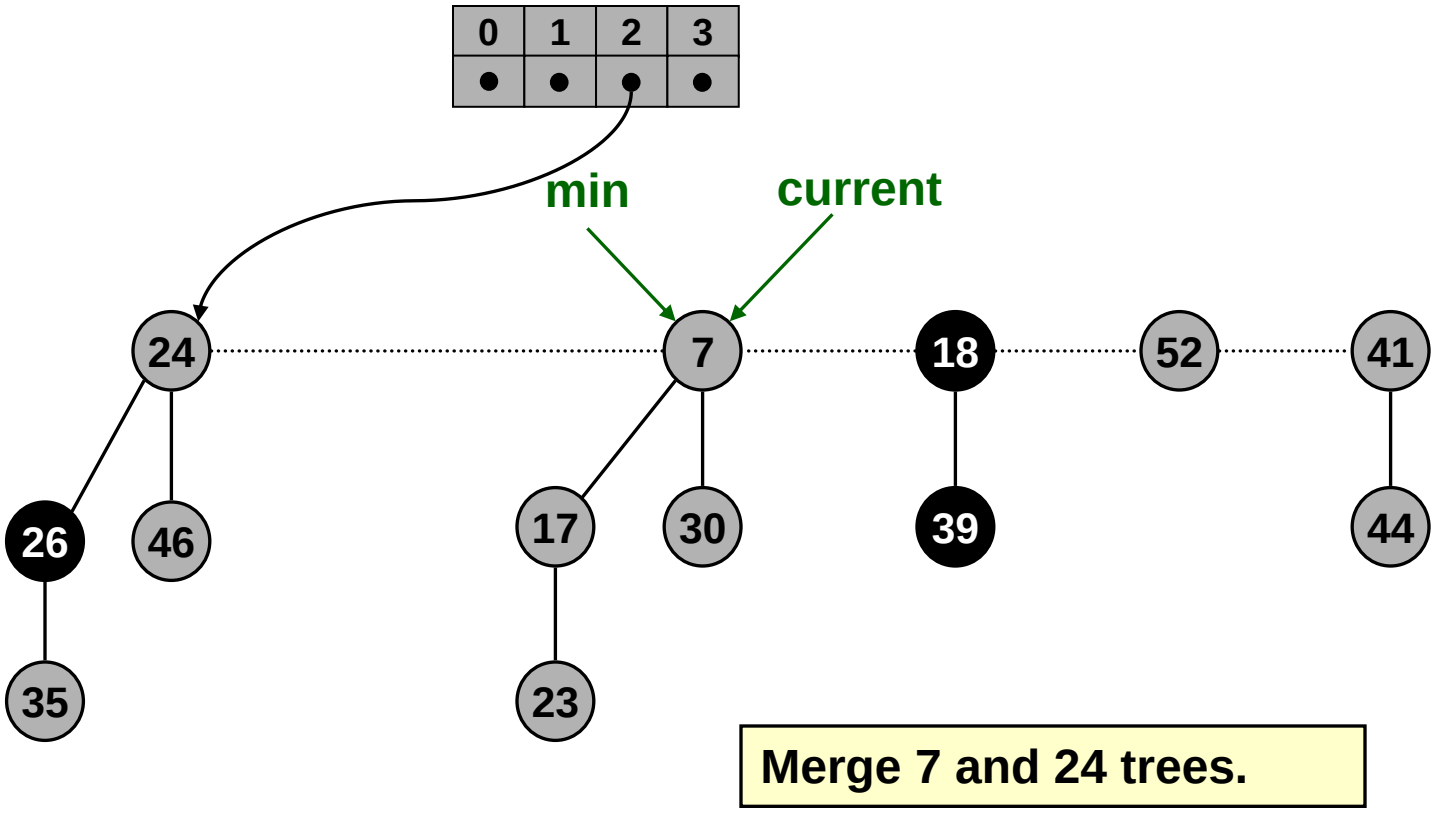


Merge 7 and 17 trees.

Fibonacci Heaps: Delete Min

Delete min.

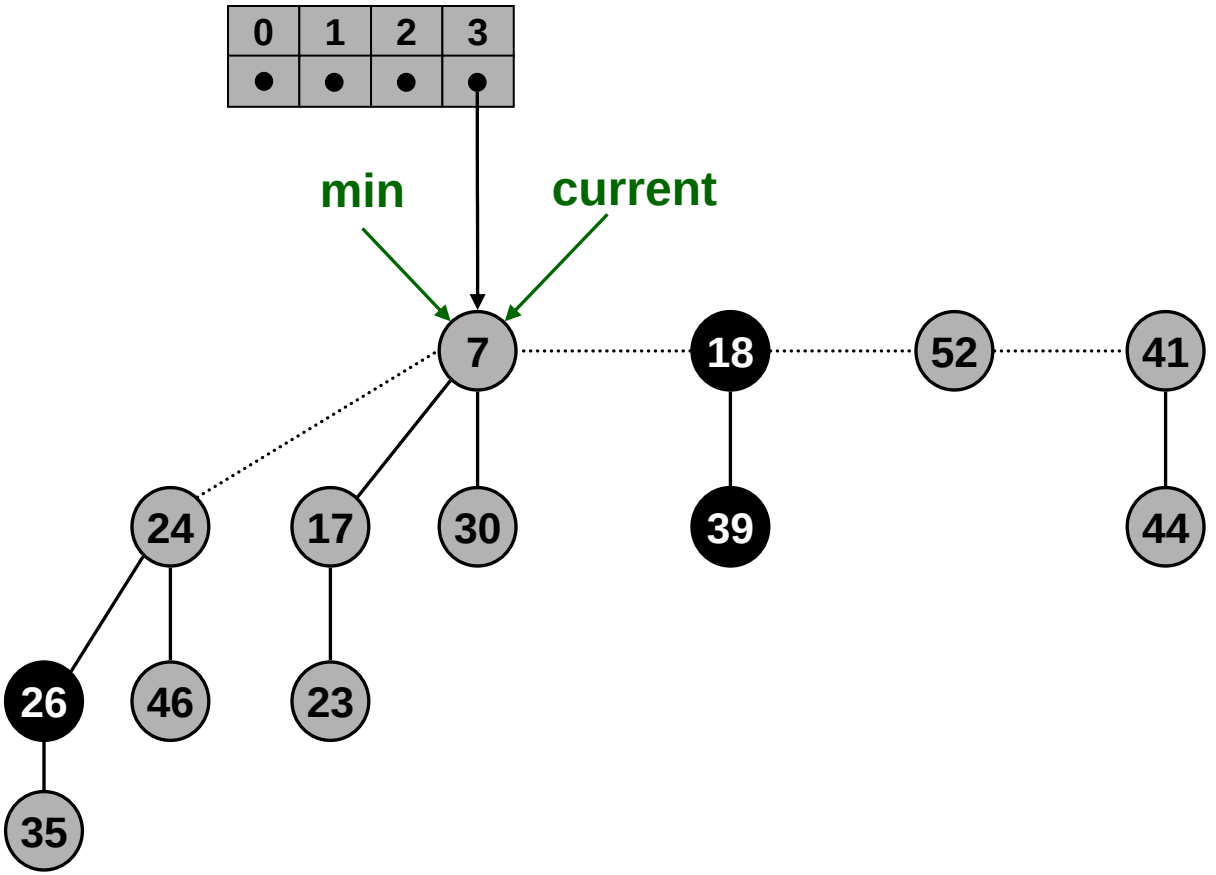
- Delete min and concatenate its children into root list.
- **Consolidate trees so that no two roots have same degree.**



Fibonacci Heaps: Delete Min

Delete min.

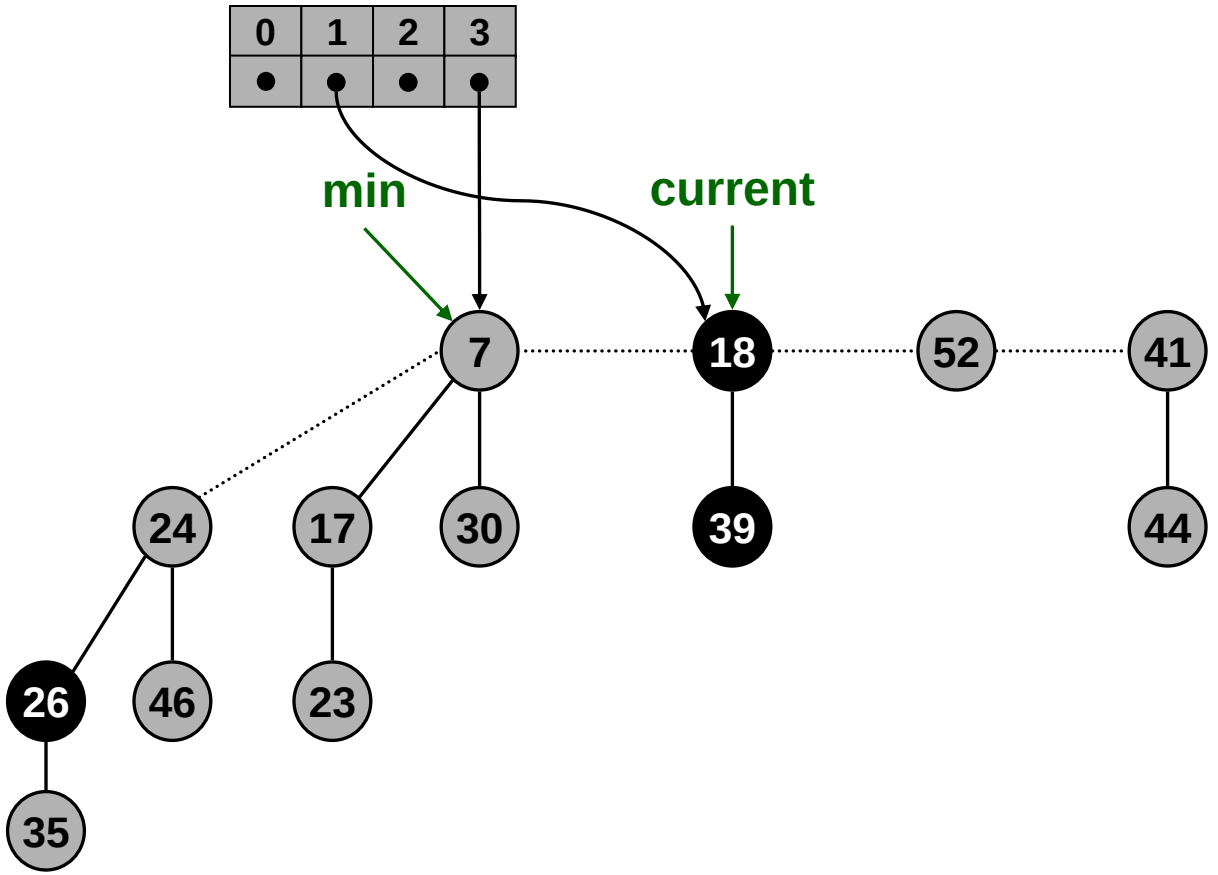
- Delete min and concatenate its children into root list.
- **Consolidate trees so that no two roots have same degree.**



Fibonacci Heaps: Delete Min

Delete min.

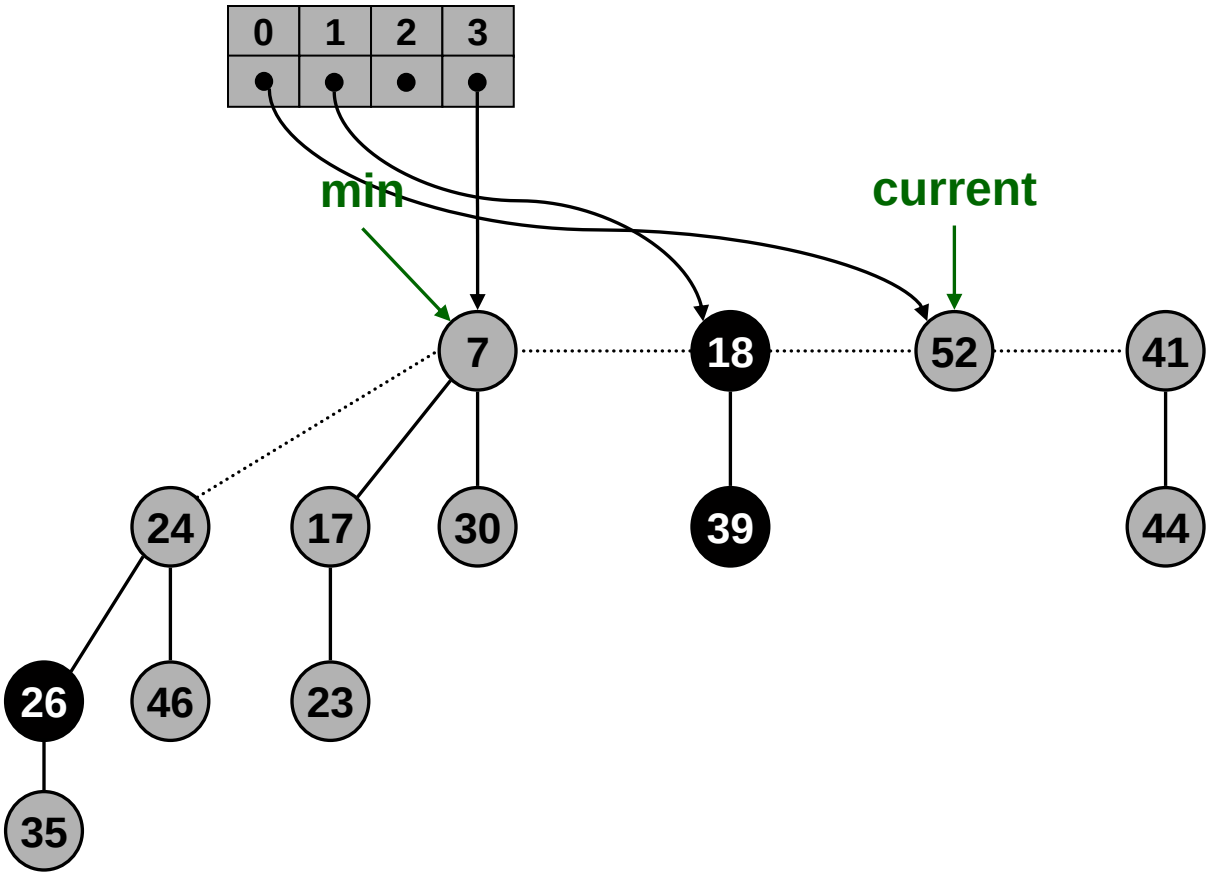
- Delete min and concatenate its children into root list.
- **Consolidate trees so that no two roots have same degree.**



Fibonacci Heaps: Delete Min

Delete min.

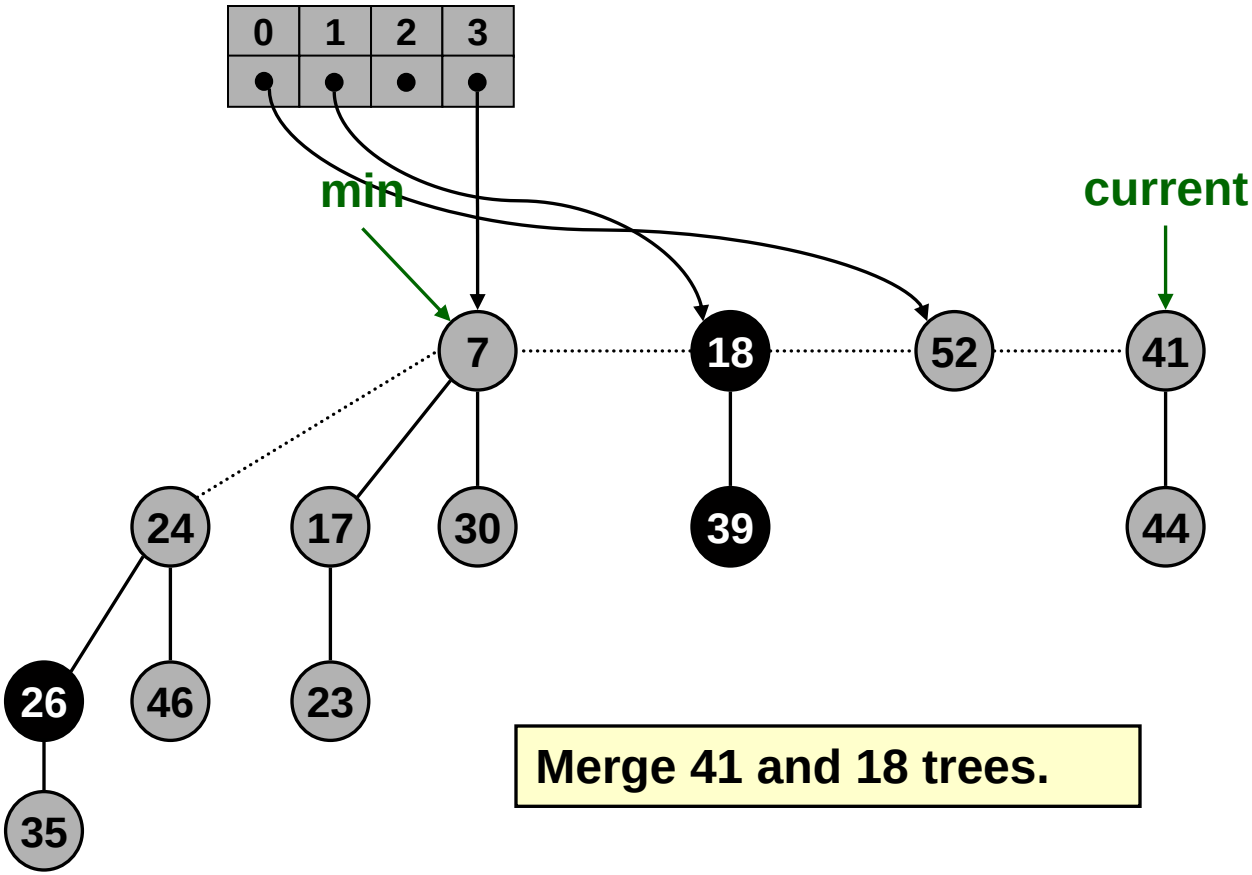
- Delete min and concatenate its children into root list.
- **Consolidate trees so that no two roots have same degree.**



Fibonacci Heaps: Delete Min

Delete min.

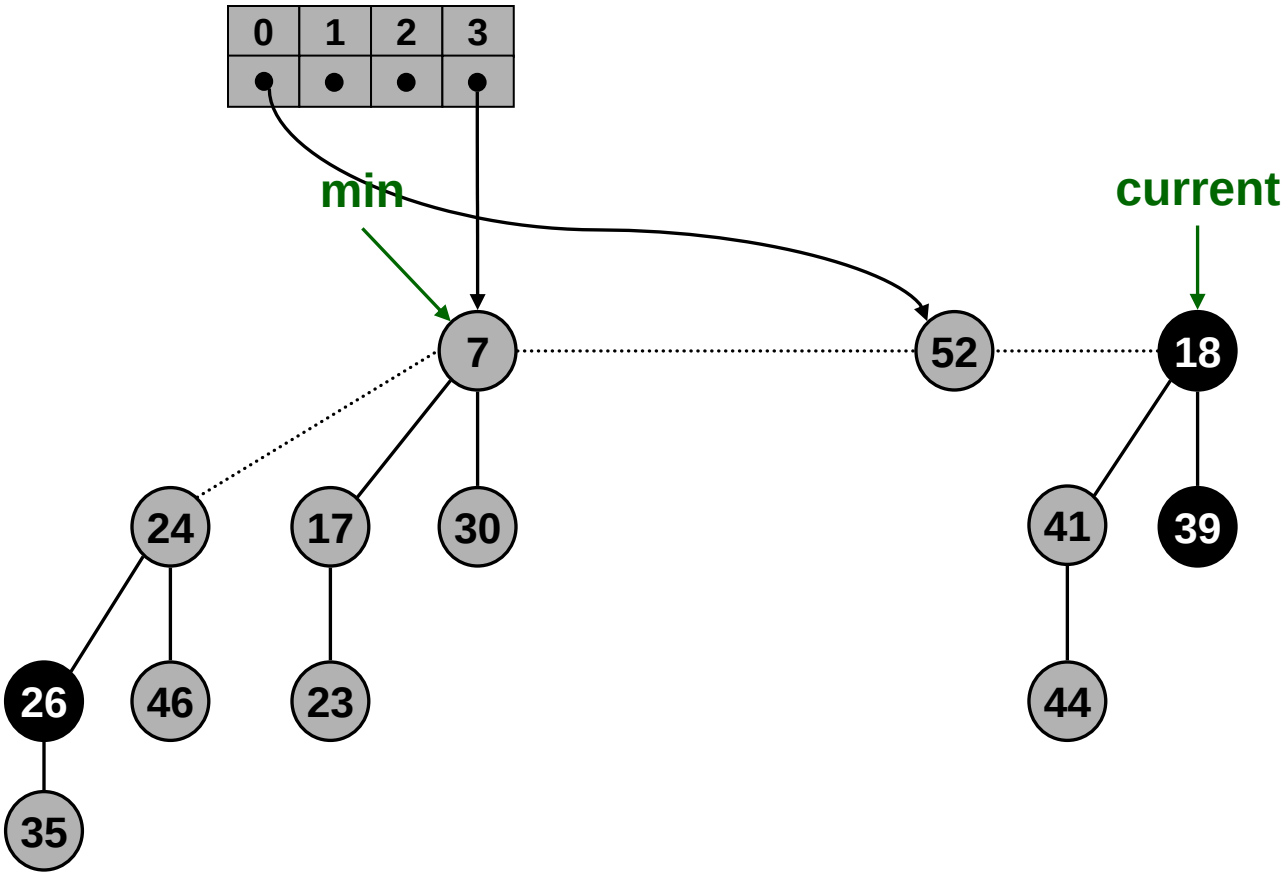
- Delete min and concatenate its children into root list.
- **Consolidate trees so that no two roots have same degree.**



Fibonacci Heaps: Delete Min

Delete min.

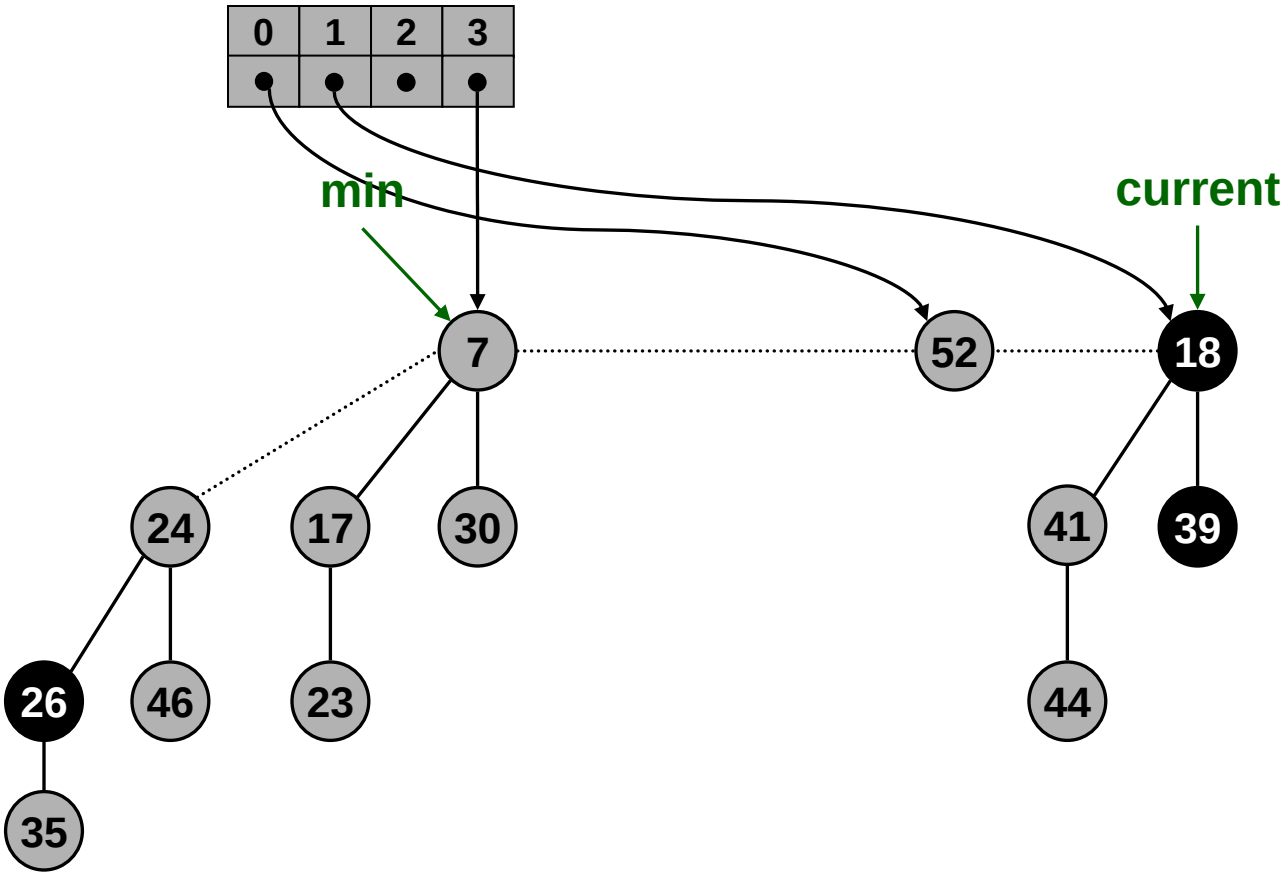
- Delete min and concatenate its children into root list.
- **Consolidate trees so that no two roots have same degree.**



Fibonacci Heaps: Delete Min

Delete min.

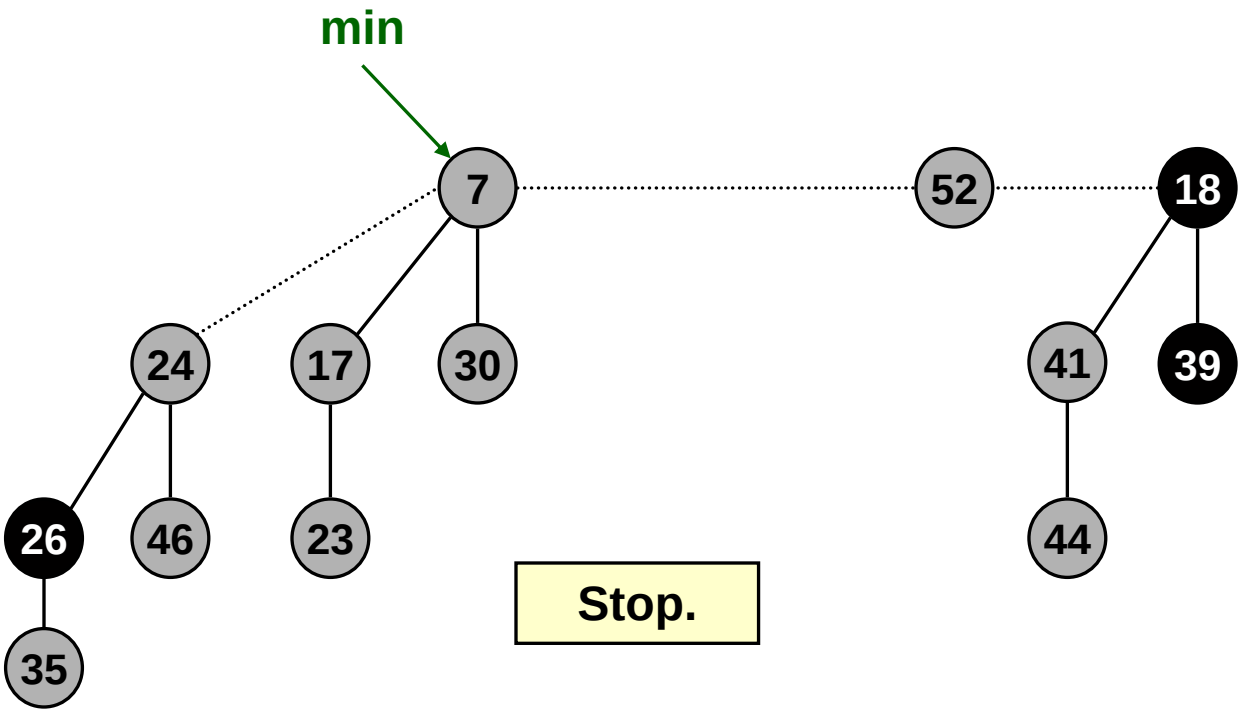
- Delete min and concatenate its children into root list.
- **Consolidate trees so that no two roots have same degree.**



Fibonacci Heaps: Delete Min

Delete min.

- Delete min and concatenate its children into root list.
- **Consolidate trees so that no two roots have same degree.**



Fibonacci Heaps: Delete Min Analysis

Notation.

- D(n) = max degree of any node in Fibonacci heap with n nodes.
- t(H) = # trees in heap H.
- $\Phi(H) = t(H) + 2m(H)$.

Actual cost. O(D(n) + t(H))

- O(D(n)) work adding min's children into root list and updating min.
  - at most D(n) children of min node
- O(D(n) + t(H)) work consolidating trees.
  - work is proportional to size of root list since number of roots decreases by one after each merging
  - $\leq$ D(n) + t(H) - 1 root nodes at beginning of consolidation

Amortized cost. O(D(n))

- t(H') $\leq$ D(n) + 1 since no two trees have same degree.
- $\Delta\Phi(H) \leq$ D(n) + 1 - t(H).

Fibonacci Heaps: Delete Min Analysis

Is amortized cost of O(D(n)) good?

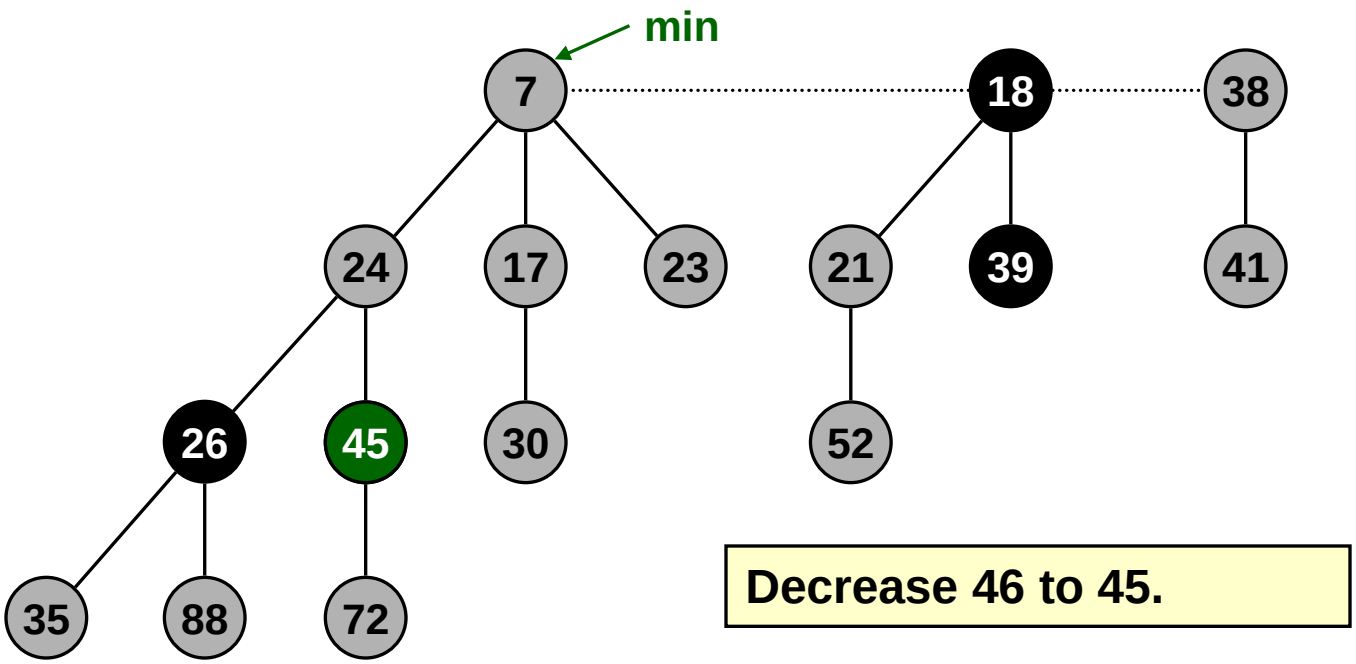
- **Yes, if only Insert, Delete-min, and Union operations supported.**
  - **in this case, Fibonacci heap contains only binomial trees since we only merge trees of equal root degree**
  - **this implies $D(n) \leq \lfloor \log_2 N \rfloor$**

- **Yes, if we support Decrease-key in clever way.**
  - **we'll show that $D(n) \leq \lfloor \log_\phi N \rfloor$, where ϕ is golden ratio**
  - **$\phi^2 = 1 + \phi$**
  - **$\phi = (1 + \sqrt{5}) / 2 = 1.618\ldots$**
  - **limiting ratio between successive Fibonacci numbers!**

Fibonacci Heaps: Decrease Key

Decrease key of element x to k.

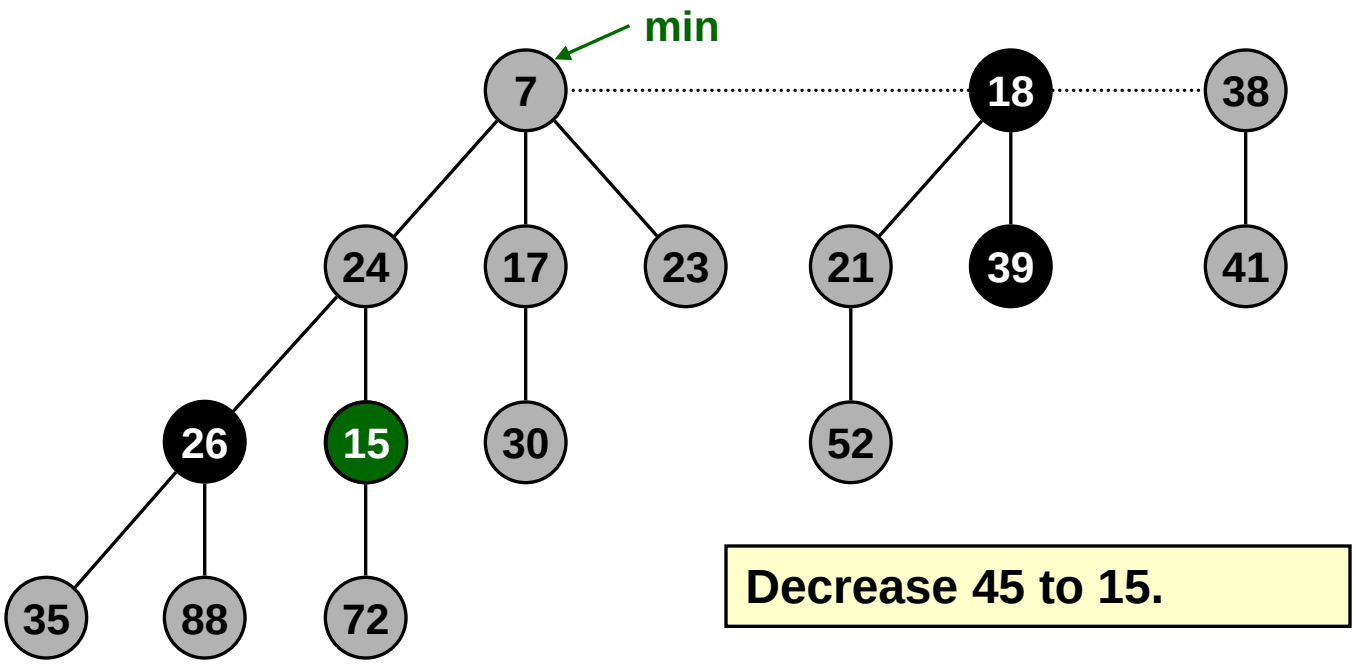
- **Case 0: min-heap property not violated.**
  - **decrease key of x to k**
  - **change heap min pointer if necessary**



Fibonacci Heaps: Decrease Key

Decrease key of element x to k.

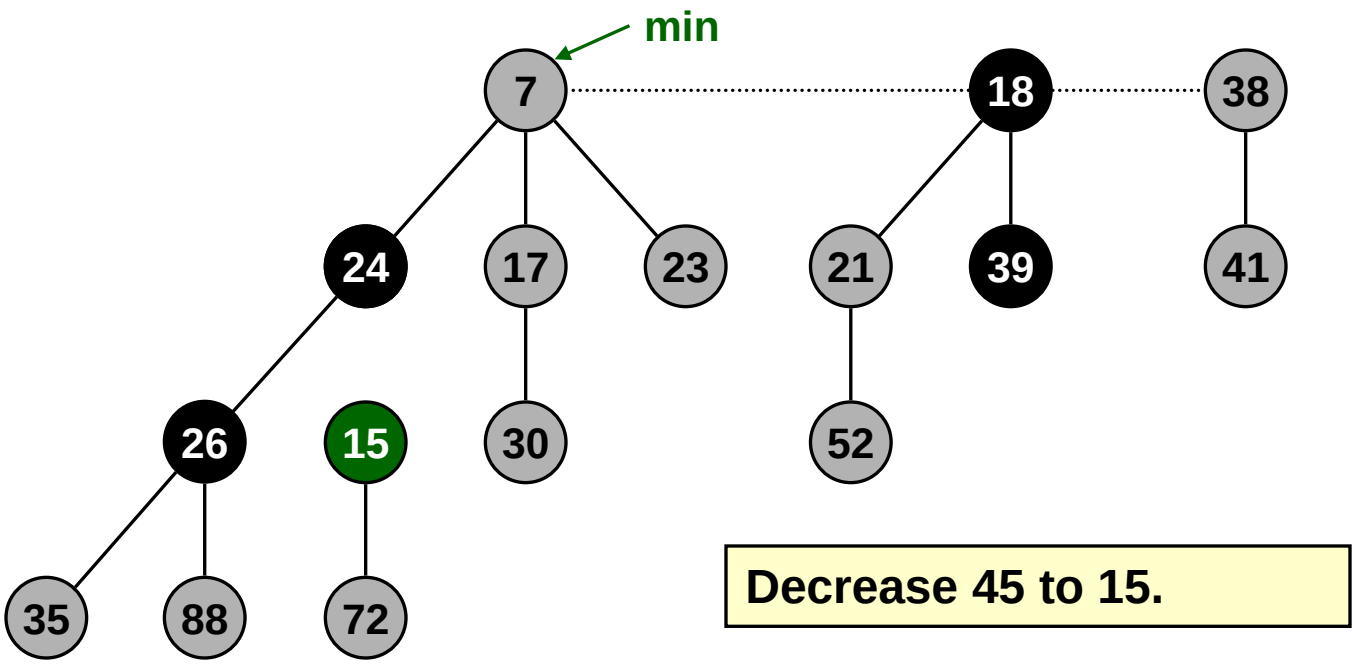
- **Case 1: parent of x is unmarked.**
  - **decrease key of x to k**
  - **cut off link between x and its parent**
  - **mark parent**
  - **add tree rooted at x to root list, updating heap min pointer**



Fibonacci Heaps: Decrease Key

Decrease key of element x to k.

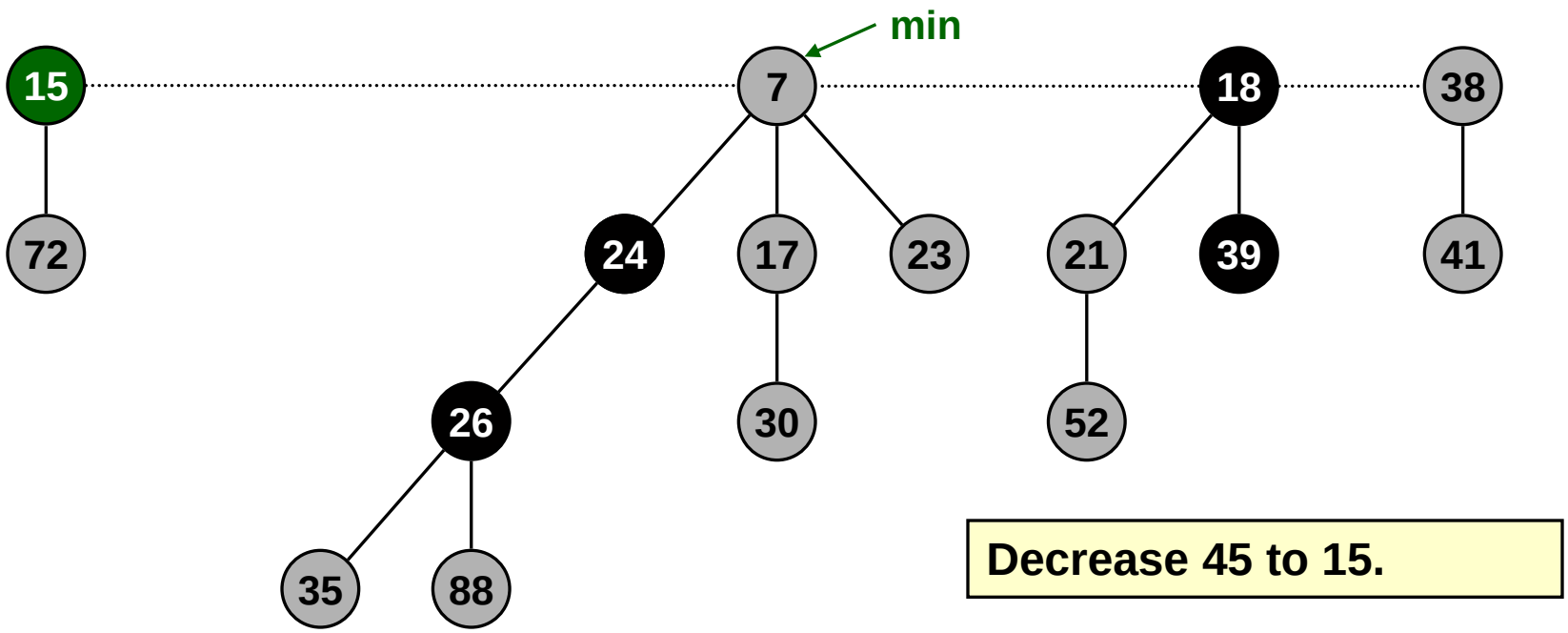
- **Case 1: parent of x is unmarked.**
  - **decrease key of x to k**
  - **cut off link between x and its parent**
  - **mark parent**
  - **add tree rooted at x to root list, updating heap min pointer**



Fibonacci Heaps: Decrease Key

Decrease key of element x to k.

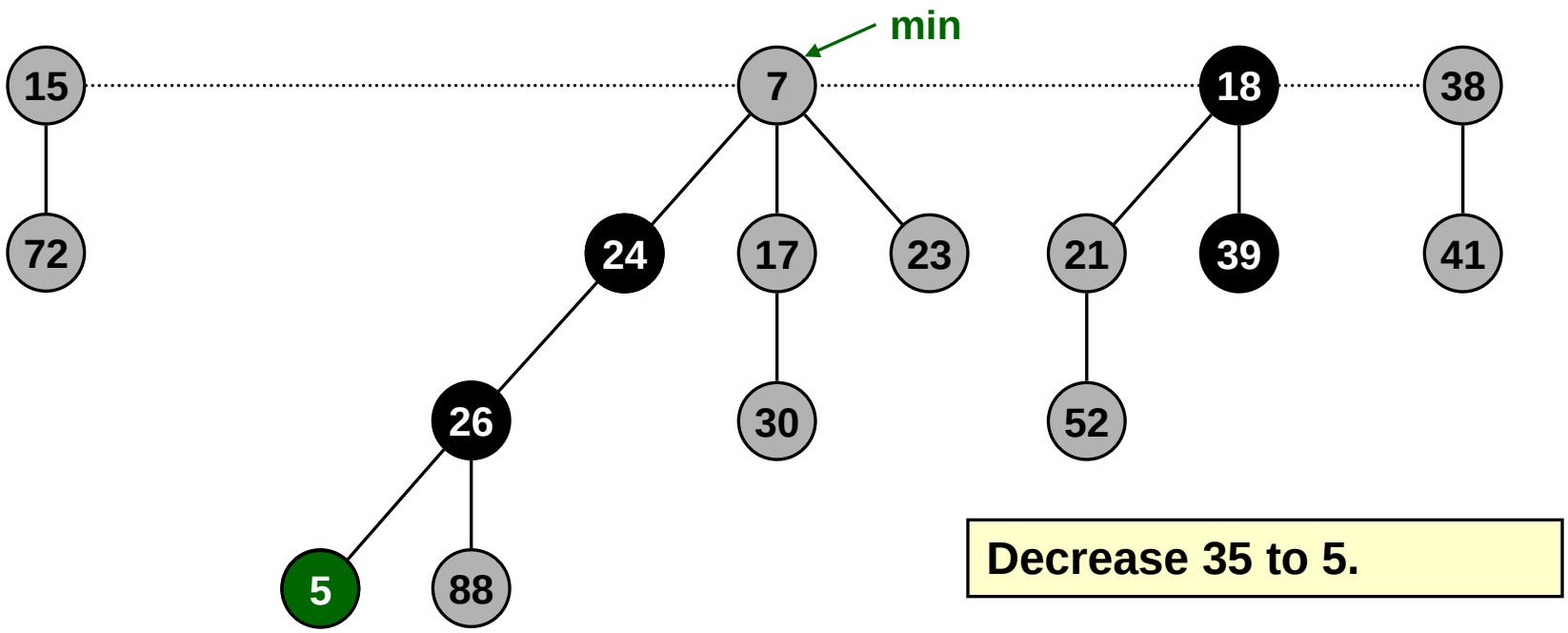
- **Case 1: parent of x is unmarked.**
  - **decrease key of x to k**
  - **cut off link between x and its parent**
  - **mark parent**
  - **add tree rooted at x to root list, updating heap min pointer**



Fibonacci Heaps: Decrease Key

Decrease key of element x to k.

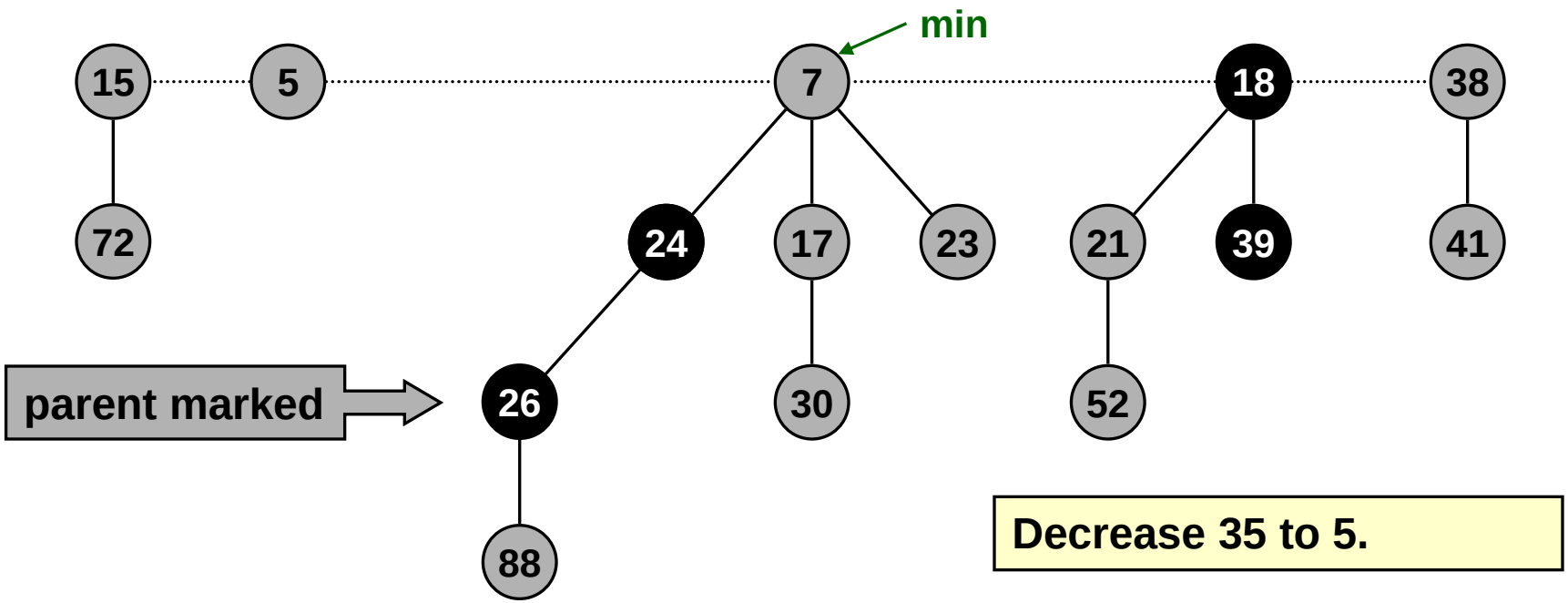
- **Case 2: parent of x is marked.**
  - **decrease key of x to k**
  - **cut off link between x and its parent p[x], and add x to root list**
  - **cut off link between p[x] and p[p[x]], add p[x] to root list**
    - **If p[p[x]] unmarked, then mark it.**
    - **If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.**



Fibonacci Heaps: Decrease Key

Decrease key of element x to k.

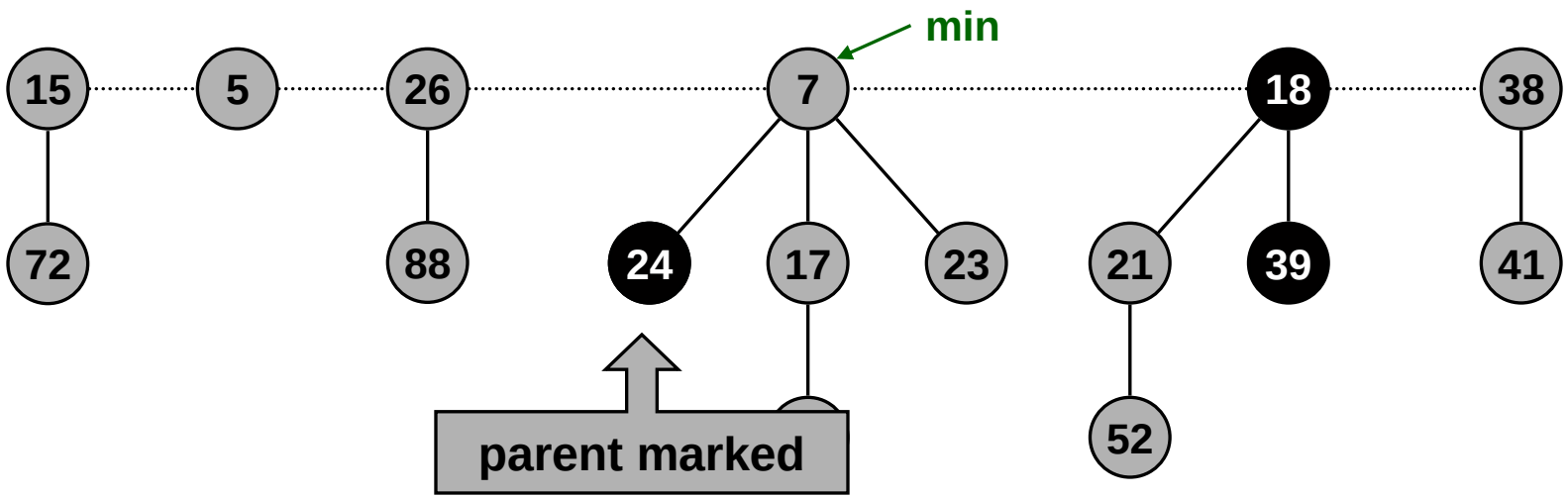
- **Case 2: parent of x is marked.**
  - **decrease key of x to k**
  - **cut off link between x and its parent p[x], and add x to root list**
  - **cut off link between p[x] and p[p[x]], add p[x] to root list**
    - **If p[p[x]] unmarked, then mark it.**
    - **If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.**



Fibonacci Heaps: Decrease Key

Decrease key of element x to k.

- **Case 2: parent of x is marked.**
  - **decrease key of x to k**
  - **cut off link between x and its parent p[x], and add x to root list**
  - **cut off link between p[x] and p[p[x]], add p[x] to root list**
    - **If p[p[x]] unmarked, then mark it.**
    - **If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.**

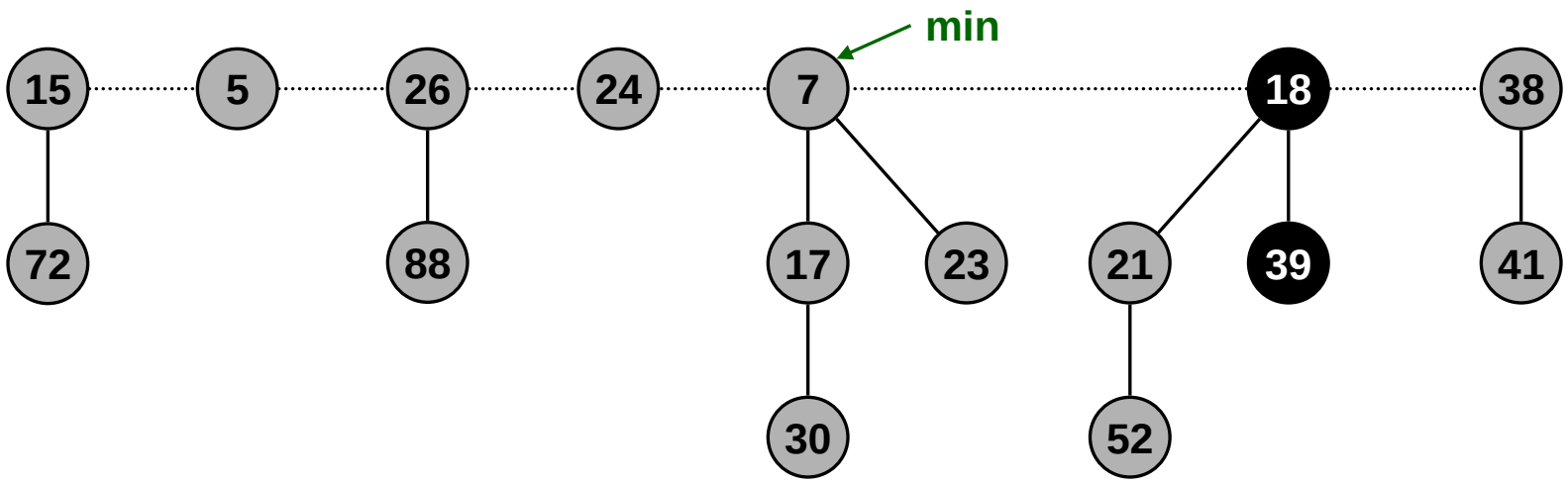


Decrease 35 to 5.

Fibonacci Heaps: Decrease Key

Decrease key of element x to k.

- **Case 2: parent of x is marked.**
  - **decrease key of x to k**
  - **cut off link between x and its parent p[x], and add x to root list**
  - **cut off link between p[x] and p[p[x]], add p[x] to root list**
    - **If p[p[x]] unmarked, then mark it.**
    - **If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.**



Decrease 35 to 5.

Fibonacci Heaps: Decrease Key Analysis

Notation.

- $t(H)$ = # trees in heap H.
- $m(H)$ = # marked nodes in heap H.
- $\Phi(H) = t(H) + 2m(H)$.

Actual cost. O(c)

- O(1) time for decrease key.
- O(1) time for each of c cascading cuts, plus reinserting in root list.

Amortized cost. O(1)

- $t(H') = t(H) + c$
- $m(H') \leq m(H) - c + 2$
  - each cascading cut unmarks a node
  - last cascading cut could potentially mark a node
- $\Delta\Phi \leq c + 2(-c + 2) = 4 - c$.

Fibonacci Heaps: Delete

Delete node x.

- Decrease key of x to $-\infty$.
- Delete min element in heap.

Amortized cost. O(D(n))

- O(1) for decrease-key.
- O(D(n)) for delete-min.
- D(n) = max degree of any node in Fibonacci heap.

Fibonacci Heaps: Bounding Max Degree

Definition. D(N) = max degree in Fibonacci heap with N nodes.

Key lemma. $D(N) \leq \log_\phi N$, where $\phi = (1 + \sqrt{5}) / 2$.

Corollary. Delete and Delete-min take O(log N) amortized time.

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

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

Proof.

- When y_i is linked to x, $y_1, \ldots, y_{i-1}$ already linked to x,
  $\Rightarrow$ degree(x) = i - 1
  $\Rightarrow$ degree(y_i) = 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, degree(y_i) = i - 1 or i - 2

Fibonacci Heaps: Bounding Max Degree

Key lemma. In a Fibonacci heap with N nodes, the maximum degree of any node is at most $\log_\phi N$, where $\phi = (1 + \sqrt{5}) / 2$.

Proof of key lemma.

- For any node x, we show that size(x) $\geq \phi^{\text{degree(x)}}$.
  - size(x) = # node in subtree rooted at x
  - taking base ϕ logs, degree(x) $\leq \log_\phi$ (size(x)) $\leq \log_\phi N$.
- Let s_k be min size of tree rooted at any degree k node.
  - trivial to see that $s_0 = 1$, $s_1 = 2$
  - s_k monotonically increases with k
- Let x* be a degree k node of size s_k, and let $y_1, \ldots, y_k$ be children in order that they were linked to x*.

Assume k ≥ 2

$$
\begin{aligned}
s_k &= \text{size}(x^*) \\
&= 2 + \sum_{i=2}^{k} size(y_i) \\
&\geq 2 + \sum_{i=2}^{k} s_{\deg[y_i]} \\
&\geq 2 + \sum_{i=2}^{k} s_{i-2} \\
&= 2 + \sum_{i=0}^{k-2} s_i
\end{aligned}
$$

Fibonacci Facts

Definition. The Fibonacci sequence is:

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

- 1, 2, 3, 5, 8, 13, 21, . . .
- Slightly nonstandard definition.

Fact F1. $F_k \geq \phi^k$, where $\phi = (1 + \sqrt{5}) / 2 = 1.618\ldots$

Fact F2. $\text{For } k \geq 2,\ F_k = 2 + \sum_{i=0}^{k-2} F_i$

Consequence. $s_k \geq F_k \geq \phi^k$.

- This implies that size(x) $\geq \phi^{\text{degree(x)}}$ for all nodes x.

$$\begin{aligned} s_k &= \text{size}(x^*) \\ &= 2 + \sum_{i=2}^{k} size(y_i) \\ &\geq 2 + \sum_{i=2}^{k} s_{\deg[y_i]} \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &= 2 + \sum_{i=0}^{k-2} s_i \end{aligned}$$

Fibonacci Proofs

Fact F1. $F_k \geq \phi^k$.

Proof. (by induction on k)

- Base cases:
  - $F_0 = 1$, $F_1 = 2 \geq \phi$.
- Inductive hypotheses:
  - $F_k \geq \phi^k$ and $F_{k+1} \geq \phi^{k+1}$

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &\geq \varphi^k + \varphi^{k+1} \\ &= \varphi^k(1+\varphi) \\ &= \varphi^k(\varphi^2) \\ &= \varphi^{k+2} \end{aligned}$$

$\phi^2 = \phi + 1$

Fact F2. For $k \geq 2$, $F_k = 2 + \sum_{i=0}^{k-2} F_i$

Proof. (by induction on k)

- Base cases:
  - $F_2 = 3$, $F_3 = 5$
- Inductive hypotheses:

$$F_k = 2 + \sum_{i=0}^{k-2} F_i$$

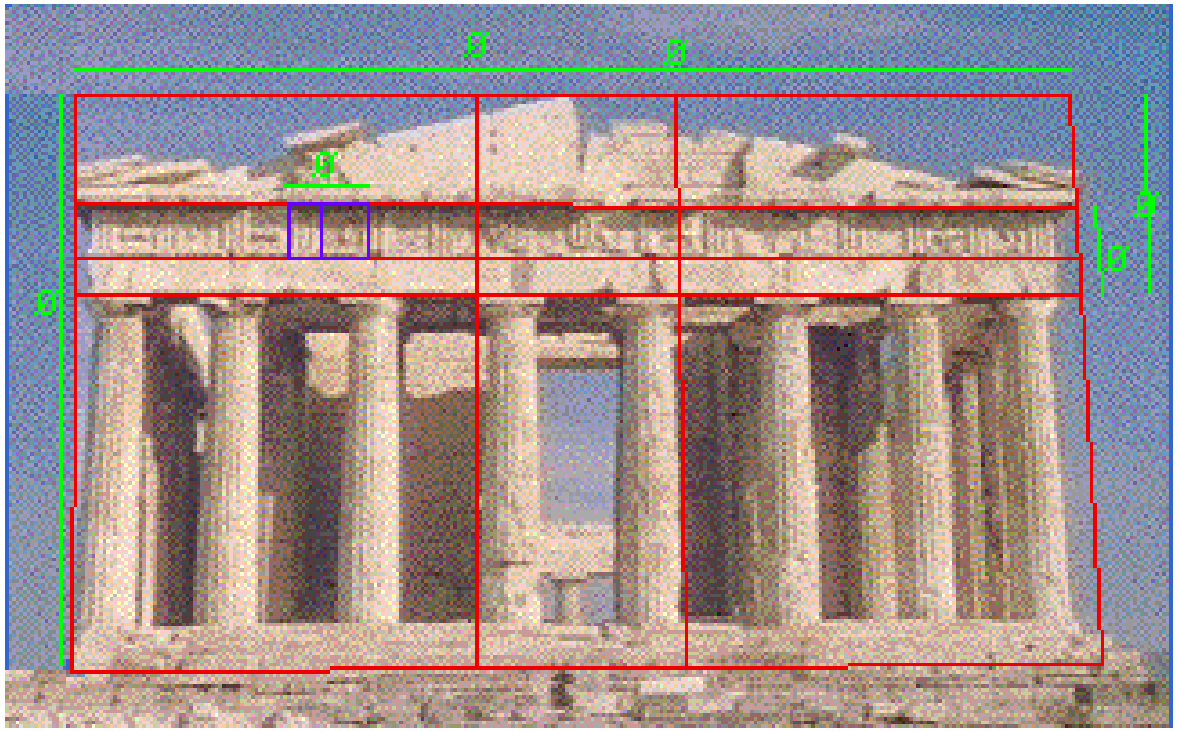
$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= 2 + \sum_{i=0}^{k-2} F_i + F_{k+1} \\ &= 2 + \sum_{i=0}^{k} F_k \end{aligned}$$

Golden Ratio

Definition. The Fibonacci sequence is: 1, 2, 3, 5, 8, 13, 21, . . .

Definition. The golden ratio $\phi = (1 + \sqrt{5}) / 2 = 1.618\ldots$

- Divide a rectangle into a square and smaller rectangle such that the smaller rectangle has the same ratio as original one.



Parthenon, Athens Greece

Golden Ratio

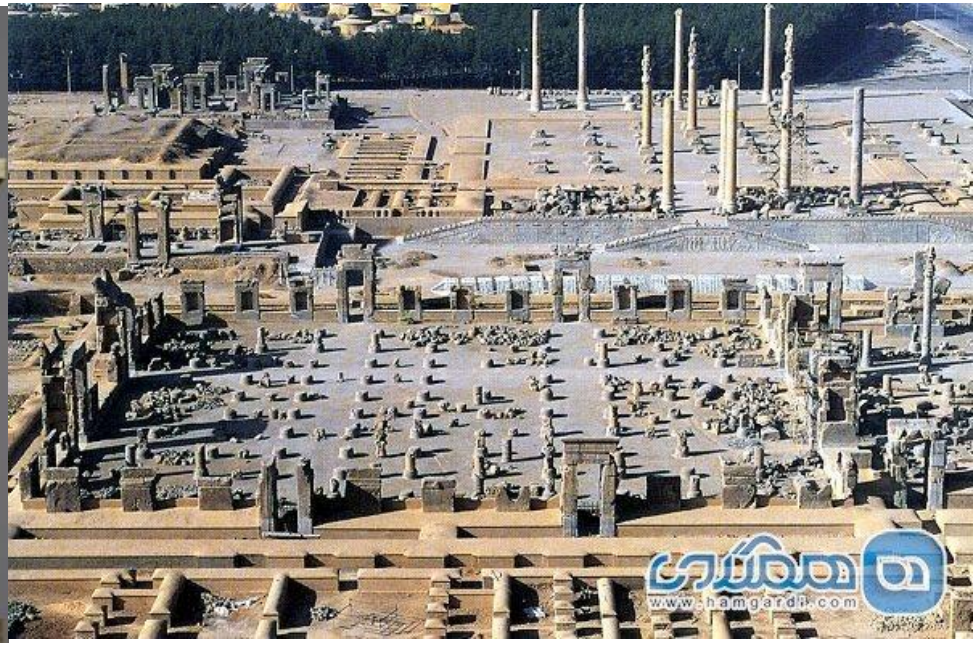
Definition. **The Fibonacci sequence is: 1, 2, 3, 5, 8, 13, 21, . . .**

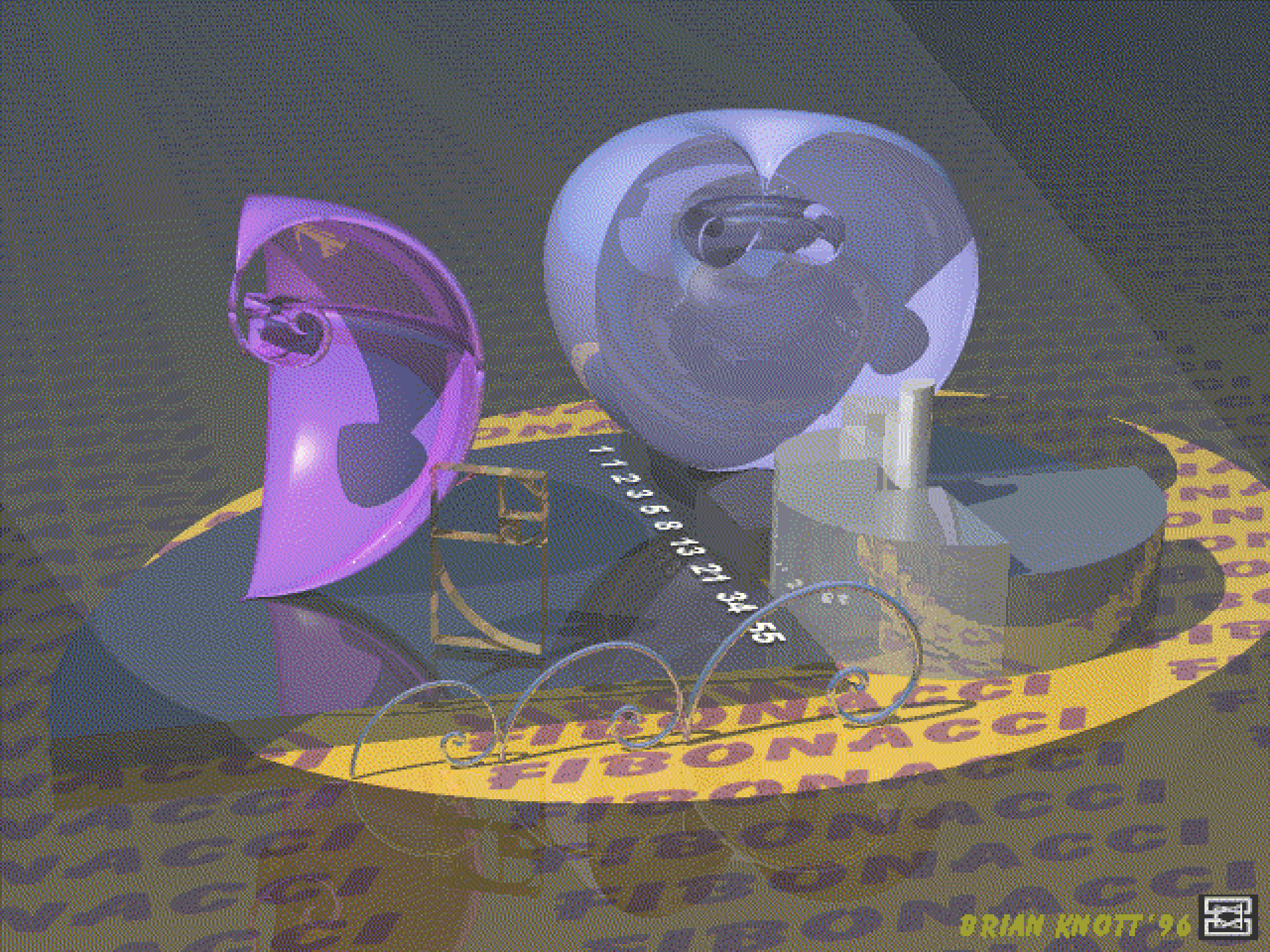
Definition. **The golden ratio $\phi = (1 + \sqrt{5}) / 2 = 1.618\ldots$**

- **Divide a rectangle into a square and smaller rectangle such that the smaller rectangle has the same ratio as original one.**

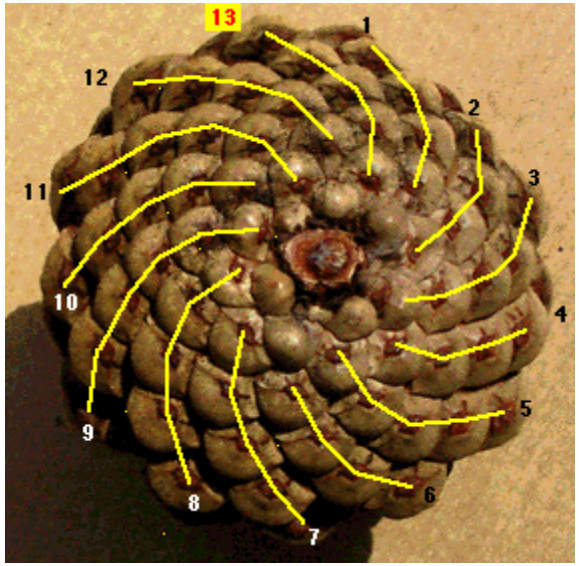
یکی از هنرهای معماری در تخت جمشید این است که نسبت ارتفاع سر درها به عرض آن‌ها و همین‌طور نسبت ارتفاع ستون‌ها به فاصلهٔ بین دو ستون *نسبت طلایی* است. نسبت طلایی نسبت مهمی در هندسه است که در طبیعت وجود دارد. این نشانگر هنر ایرانیان باستان در معماری است.[۵]

برگرفته از ویکیپدیا

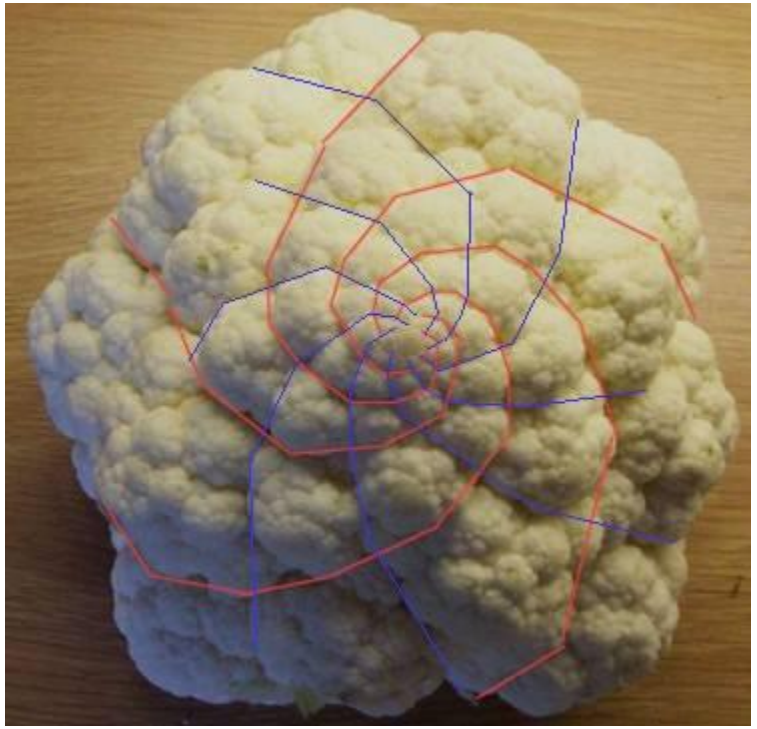



1 1 2 3 5 8 13 21 34 55
FIBONACCI
FIBONACCI
FIBONACCI
BRIAN KNOTT '96

Fibonacci Numbers and Nature



Pinecone



Cauliflower

Fat heaps
(K & Tarjan 96)

Goal

Want to achieve the performance of Fibonnaci heaps but on the worst case.

Why ?

Theoretical curiosity and

some applications

Redundant binary counters (Clancy & Knuth 77)

Go back to binomial heaps.

Lets get started from insert, How do we make insert O(1) on the worst case and keeping logarithmic worst case bounds for the other operations.

Insert is more or less like adding one to a binary number

So maybe we want first to be able to get a binary counter supporting increment in O(1) time on the worst case.

Redundant binary counters (cont)

We shall allow an additional digit.

Use 0,1, and 2, but keep the base to be 2 !

$012110 = 0*2^0 + 1*2^1 + 1*2^2 + 2*2^3 + 1*2^4 = 38$

The representation now is not unique

Can represent 38 also by 020110 or 100110

We shall exploit the redundancy.

Exploiting the redundancy

We are not going to use all possible representations

Define a representation to be regular if it does not contain the following patterns

……211111112…...
…….211111111

When you walk from left to right after you see a 2 you should see a 0 before hitting another 2 or running out of digits

Exploiting the redundancy (cont)

Note: every number has a regular representation.

Given a regular representation for x, how do you get a regular representation for x+1.

1) increment the rightmost digit.

We get a representation for x+1 which may not be regular

…….2101210 ==> ……..2101211 ==> ……..2102011

2) If indeed so, then replace the rightmost 2 with a 0 and increment the digit to its left. (Fix)

Exploiting the redundancy (cont)

Lemma: we end up with a regular representation for x+1.

Proof: Clear if we have not created a 2.

Suppose that we create a 2:

……121111 ==> …….20111

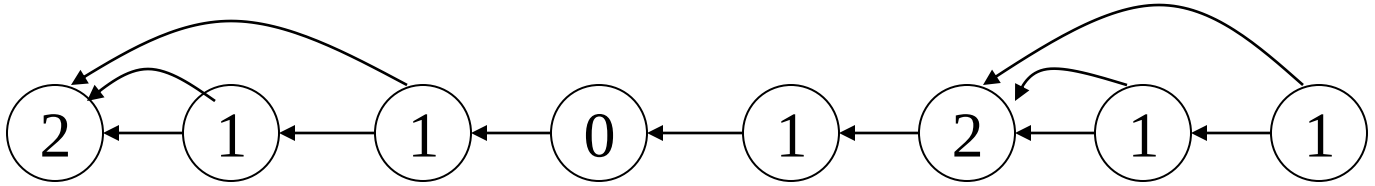
There is a zero between the new 2 and a 2 to its left because there was a zero between the old 2 and the 2 to its left.

Exploiting the redundancy (cont)

We change only O(1) digits but can we implement it in O(1) time ?

Yes!

Represent the number as a sorted list with forward pointers:



If a 1 is followed by a sequence of 1's and a 2 then its forward pointer point to that 2.

Have to update forward pointer when you create a one.

Can incrementing an arbitrary digit

To increment d_i:

Fix d_i if $d_i = 2$

increment d_i

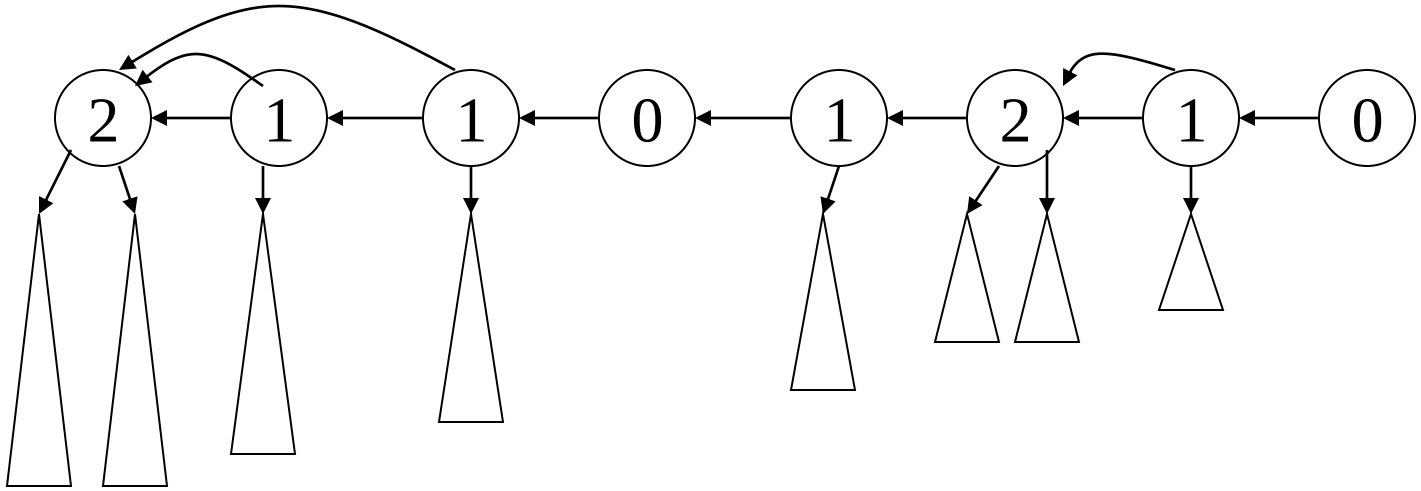
if $d_{i+1} = 1$ and the first d_j, $j>i+1$ such $d_j \neq 1$ is 2 then fix d_j

Fix d_i if $d_i = 2$

Back to heaps

Allow 0,1,2 binomial heaps of each rank.

Trees are hanging off a redundant counter. The i-th digit of the counter corresponds to the number of rank i trees

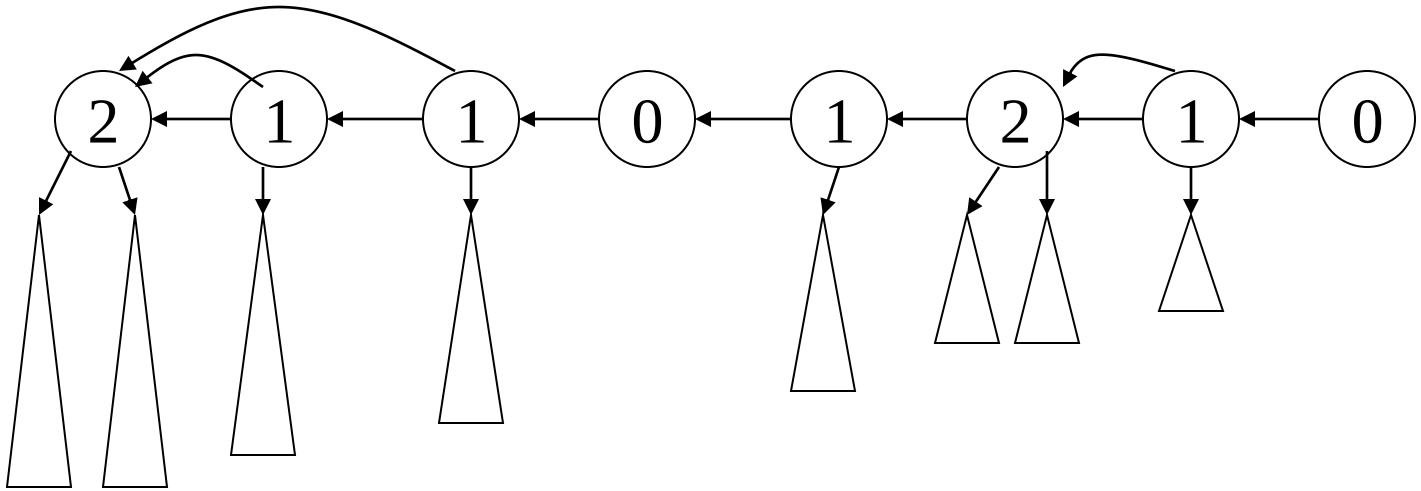


The representation in the counter must be regular.

Back to heaps (cont)

Insert now is like adding one to the redundant binary counter

While incrementing if we need to fix a 2 then it corresponds to linking two binomial trees of rank i to a binomial tree of rank i+1.



Back to heaps (cont)

Do other operations essentially as before.

Next, lets try to add a decrease key, in O(1) worst case time, without hurting other operations

Some additional facts about redundant counters

The idea extends from base 2 to any base.

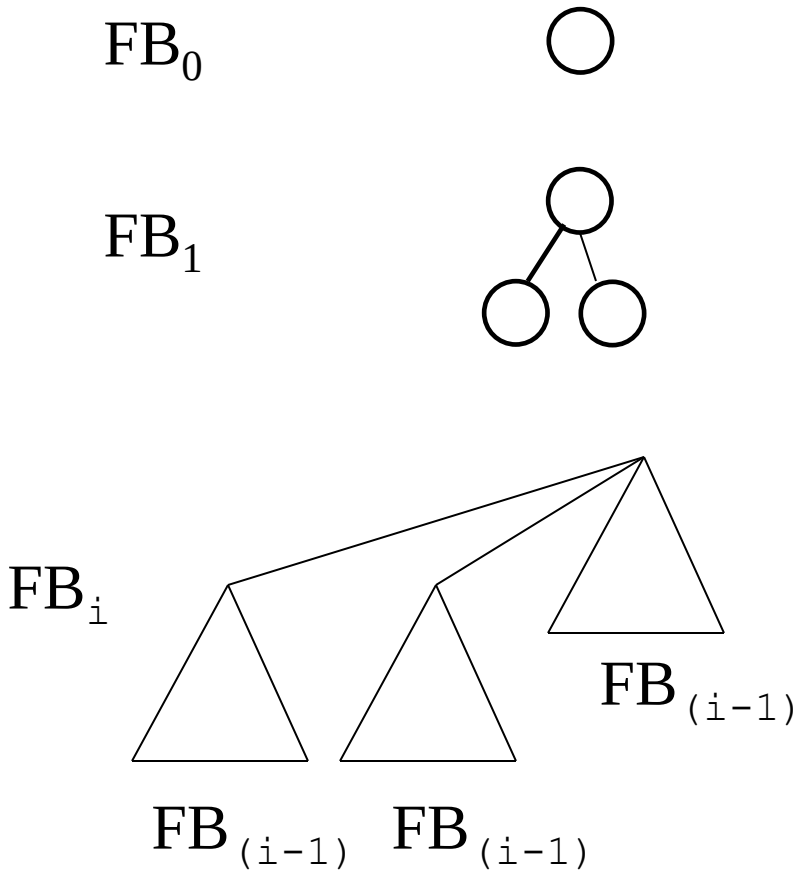
For example in base 3 the forbidden configurations would be

……3222222223….

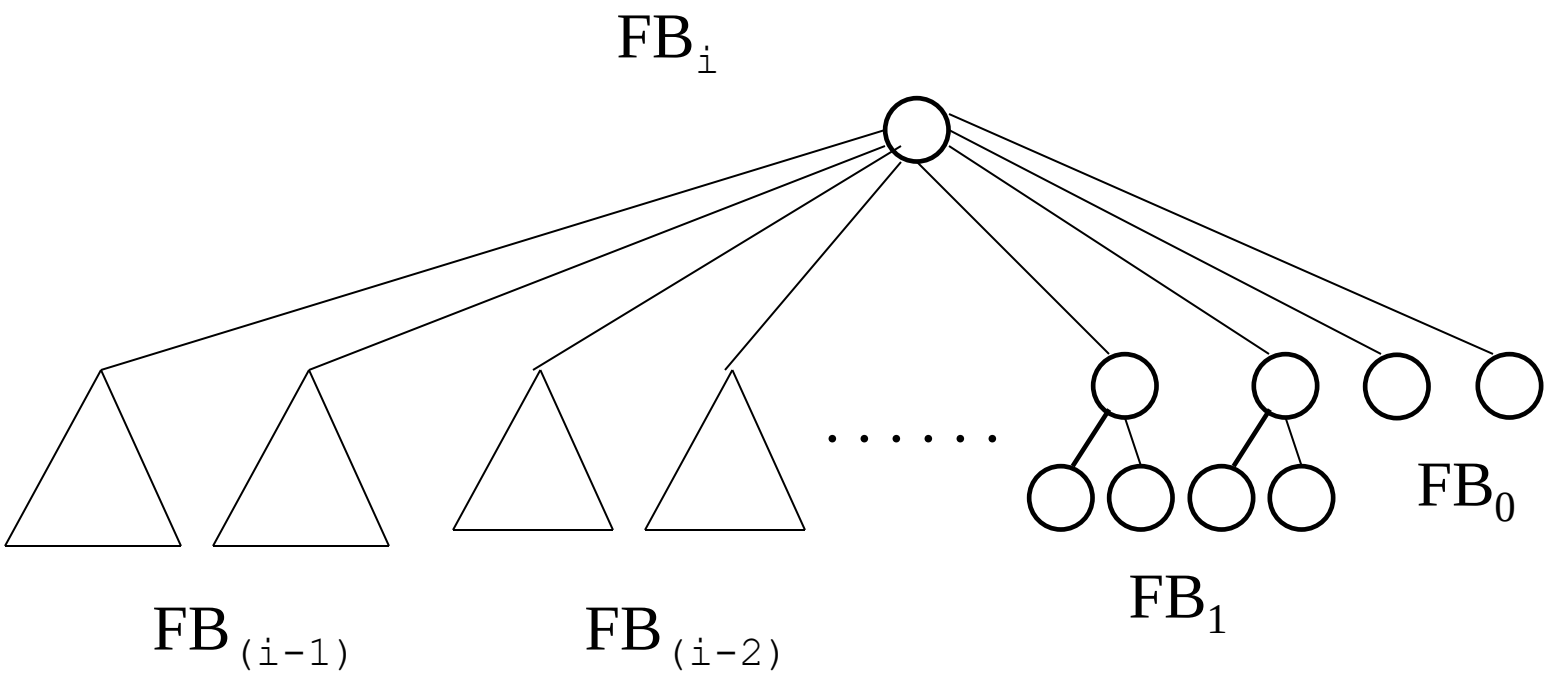
…….322222222

We can increment any digit in O(1) time if we get access to it.

Fat binomial trees



Fat binomial trees



Properties of fat binomial trees

1) $| FB_k | = 3^k$
2) $\text{degree}(\text{root}(FB_k)) = 2k$
3) $\text{depth}(FB_k) = k$

==> The degree and depth of a binomial tree with at most n nodes is O(log(n)).

Define the rank of FB_k to be k

Fat heaps (def)

A collection of fat binomial trees at most 3 of every rank.

Items in the nodes.

Trees are hanging off of a redundant base 3 counter which is regular.

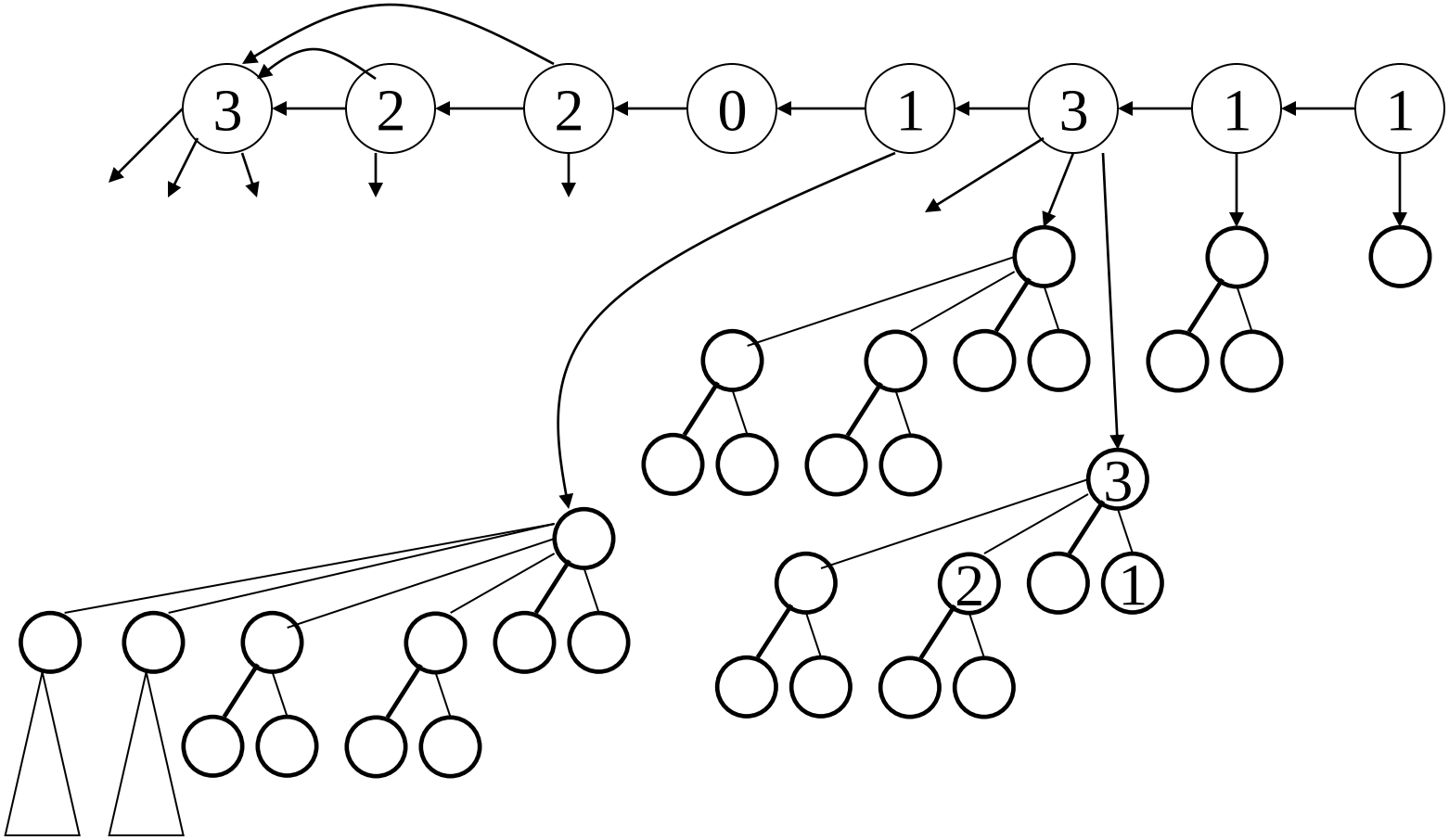
So far, the bounds for all operations are as before.

For efficient decrease key, allow violations to the heap order.

A violation is a node that is smaller than its parent.

Allow at most two such nodes of every rank in all trees

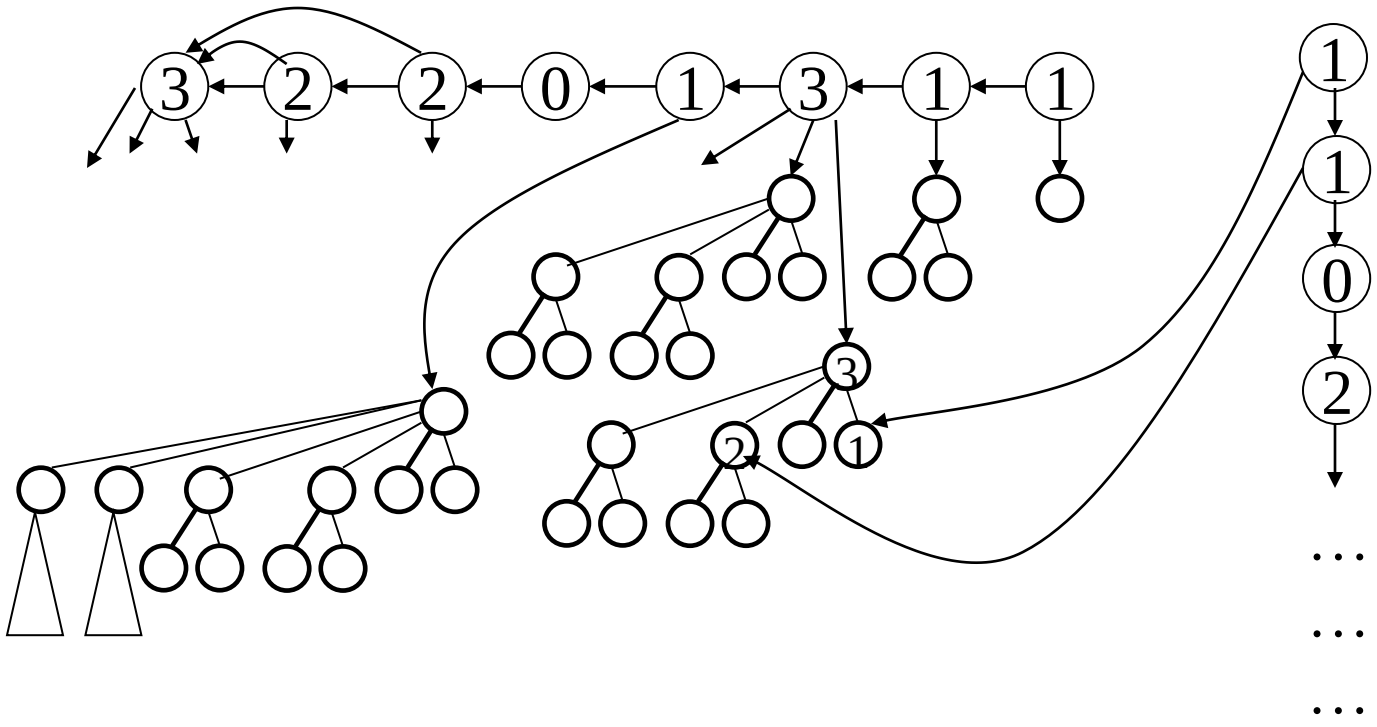
Fat heaps (example)



Organizing violations

Use another redundant binary counter.

The i-th digit corresponds to violations of rank i and stores pointers to these violating nodes. Counter must be regular.



Access the counters via arrays

Operations

Insert(x,h): essentially as before. O(1)

Deletemin(h): add the subtrees by performing increments to the root counter.

The new minimum is either at a root or at a violating node. We search for it and if it is in a violating node we swap it with one of the roots. O(log n)

Delete(x,h): decrease-key(x,h,∞), deletemin(h)

Decrease key(x,h,δ)

Decrease the key of x by δ. If x becomes the minimum swap it with the minimum item.

We may have created new rank i violation. So we need to increment digit i of the violation counter.

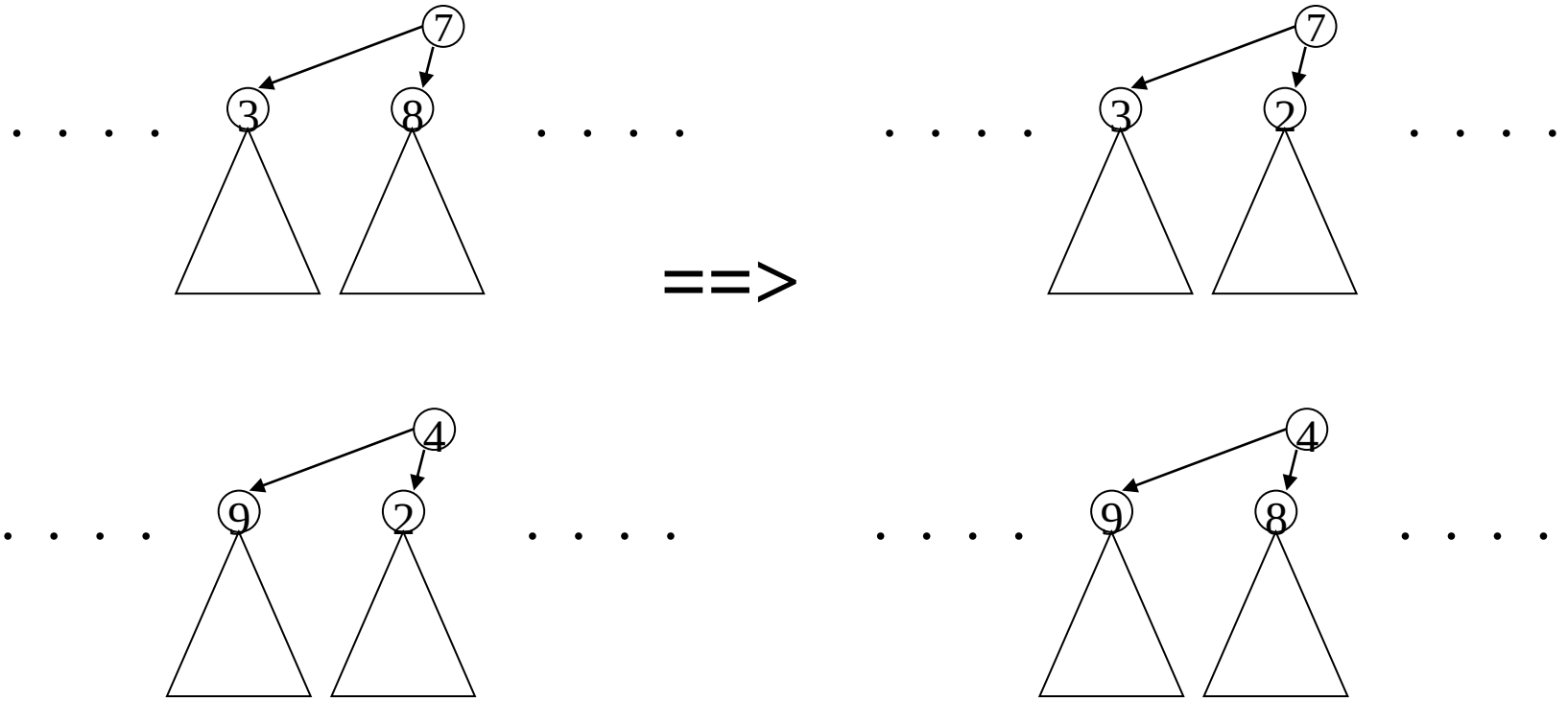
This may require a fix operation on some digits.

A fix operation corresponds to converting two violations of rank i to one violation of rank i+1.

How do we do that ?

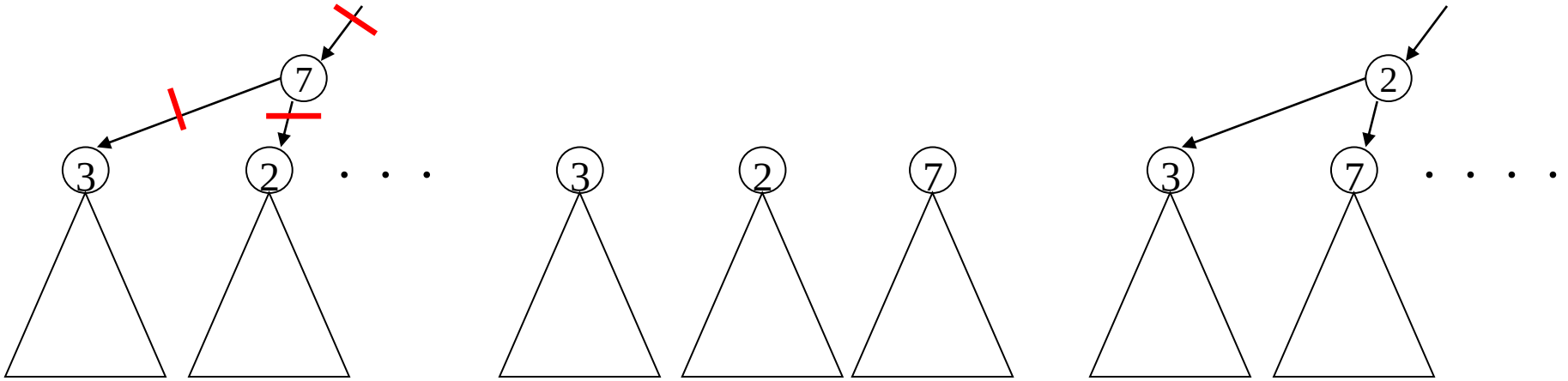
Decrease key(x,h,δ) (cont)

First arrange the two violations to have the same root. (the one with larger key)



Decrease key(x,h,δ) (cont)

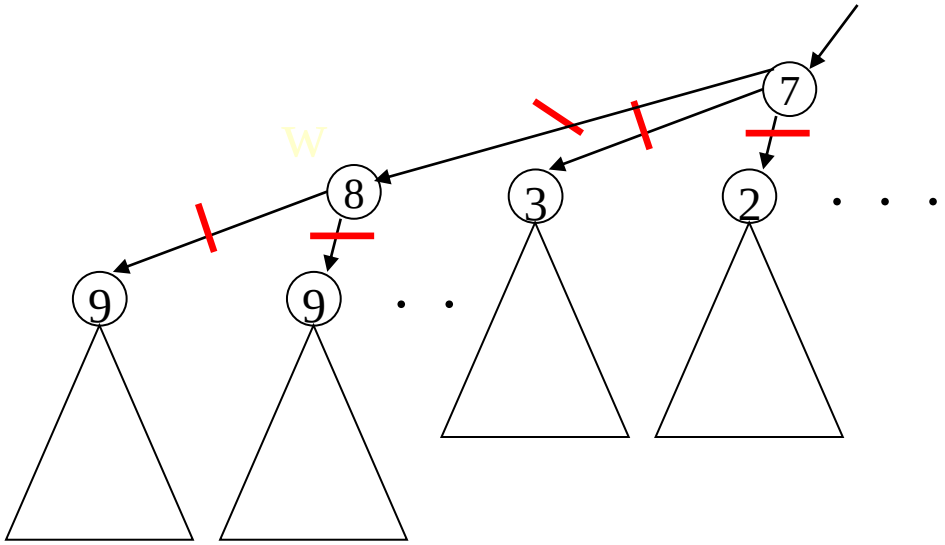
Case 1: The parent of the two rank i violations is of rank i+1.



Cut the two violating nodes and their parent. That gives 3 rank i trees. Link them to one rank i+1 tree which replaces the parent of the two violations.

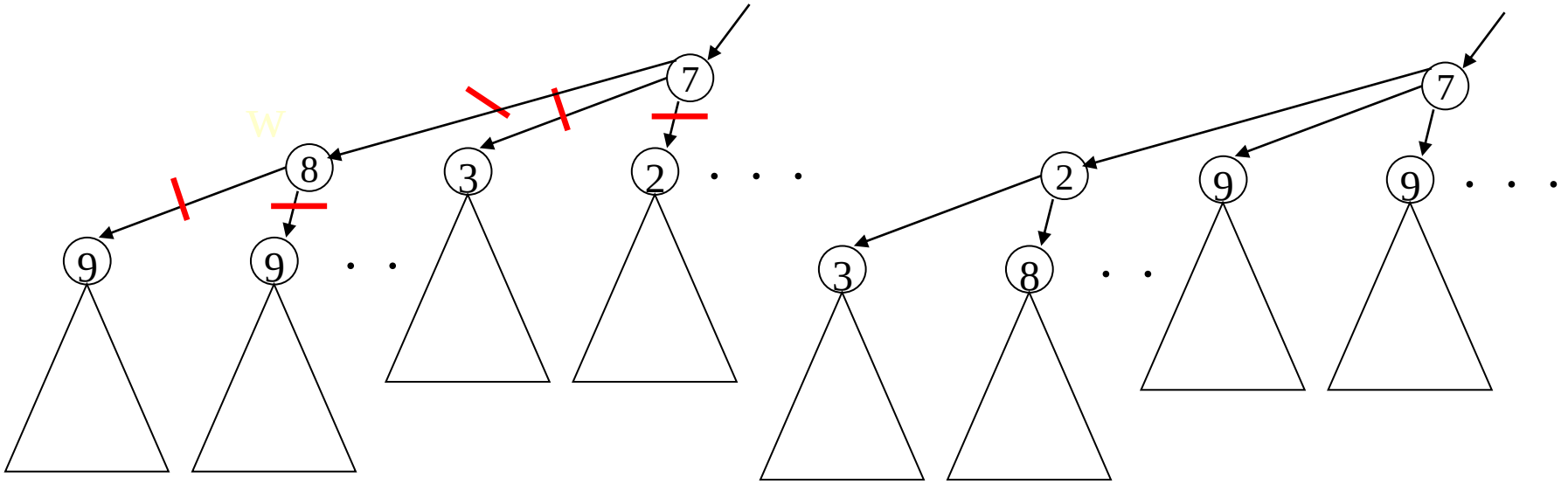
Decrease key(x,h,δ) (cont)

Case 2: The parent of the two rank i violations is of rank at least i+2.

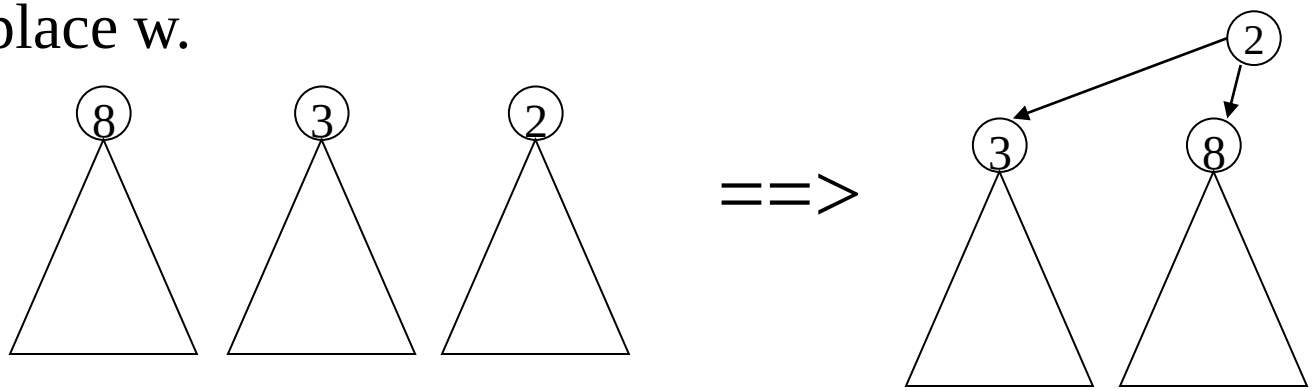


This parent must have a non-violating child w of rank i+1.
This child must have two non violating rank i children.

Decrease key(x,h,δ) (cont)

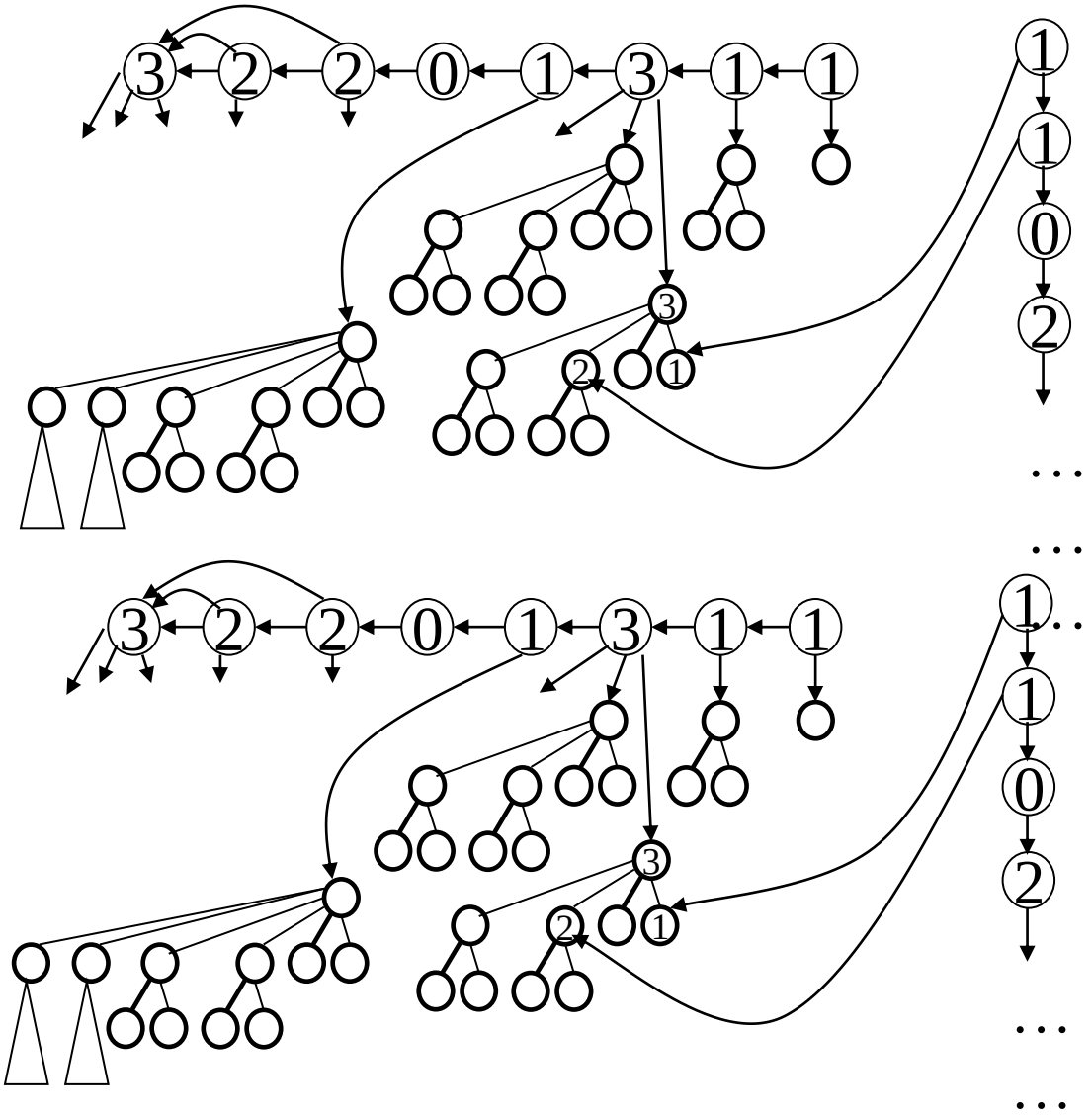


Replace the violating subtrees with the rank i subtrees of w. Cut w (without its leftmost subtrees). Link the subtree rooted by w with the violating subtrees. Use the resulting subtree to replace w.



119

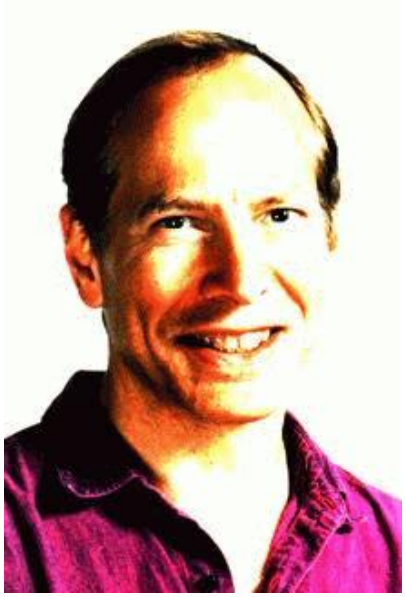
Union(h1,h2)



Fix all violations in h1 and then insert the subtrees of h1 into the counter of h2 by doing increments

On Complicated Algorithms

"Once you succeed in writing the programs for [these] complicated algorithms, they usually run extremely fast. The computer doesn't need to understand the algorithm, its task is only to run the programs."



R. E. Tarjan