



Chapter 5

Data Structures

Algorithm Theory
WS 2015/16

Fabian Kuhn

Priority Queue / Heap

- Stores $(key, data)$ pairs (like dictionary)
- But, different set of operations:
- **Initialize-Heap**: creates new empty heap
- **Is-Empty**: returns true if heap is empty
- **Insert** $(key, data)$: inserts $(key, data)$ -pair, returns pointer to entry
- **Get-Min**: returns $(key, data)$ -pair with minimum key
- **Delete-Min**: deletes minimum $(key, data)$ -pair
- **Decrease-Key** $(entry, newkey)$: decreases key of $entry$ to $newkey$
- **Merge**: merges two heaps into one

Analysis

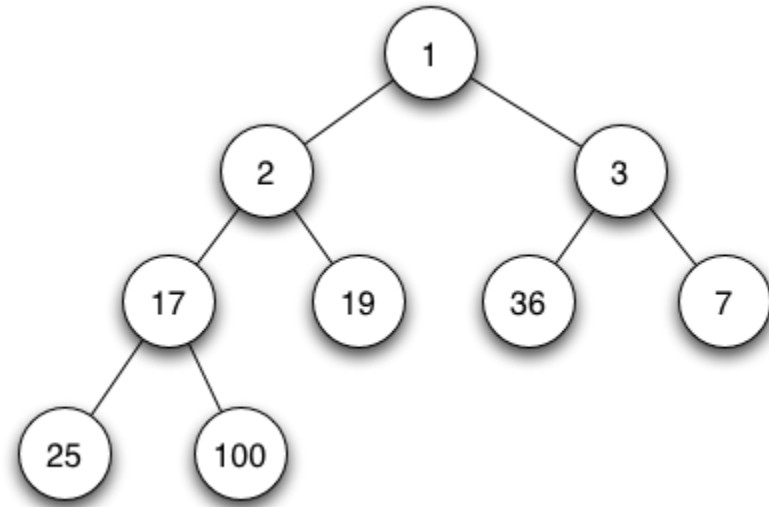
Number of priority queue operations for Dijkstra:

- **Initialize-Heap:** **1**
- **Is-Empty:** **$|V|$**
- **Insert:** **$|V|$**
- **Get-Min:** **$|V|$**
- **Delete-Min:** **$|V|$**
- **Decrease-Key:** **$|E|$**
- **Merge:** **0**

Priority Queue Implementation

Implementation as min-heap:

→ complete binary tree,
e.g., stored in an array



- **Initialize-Heap:** $O(1)$
- **Is-Empty:** $O(1)$
- **Insert:** $O(\log n)$
- **Get-Min:** $O(1)$
- **Delete-Min:** $O(\log n)$
- **Decrease-Key:** $O(\log n)$
- **Merge** (heaps of size m and n , $m \leq n$): $O(m \log n)$

Fibonacci Heaps

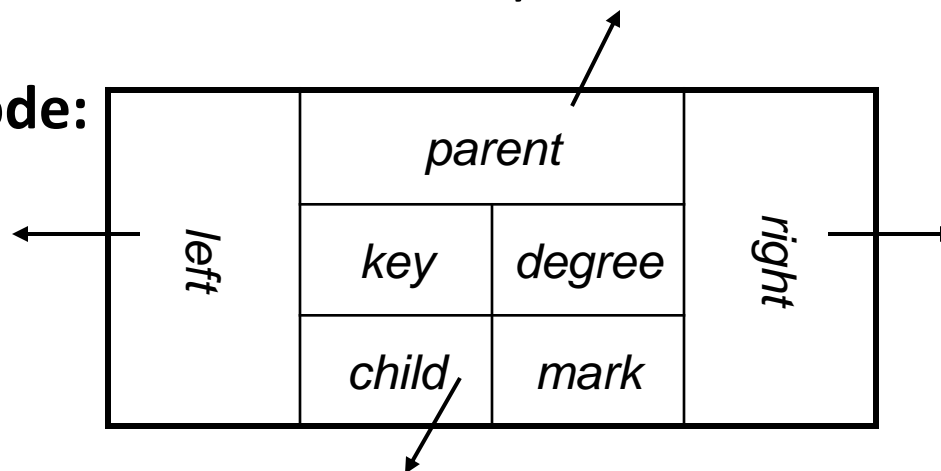
Structure:

A Fibonacci heap H consists of a collection of trees satisfying the min-heap property.

Global Variables:

- $H.min$: root of the tree containing the (a) minimum key
- $H.rootlist$: circular, doubly linked, unordered list containing the roots of all trees
- $H.size$: number of nodes currently in H

Structure of a Node:



Fibonacci Heaps

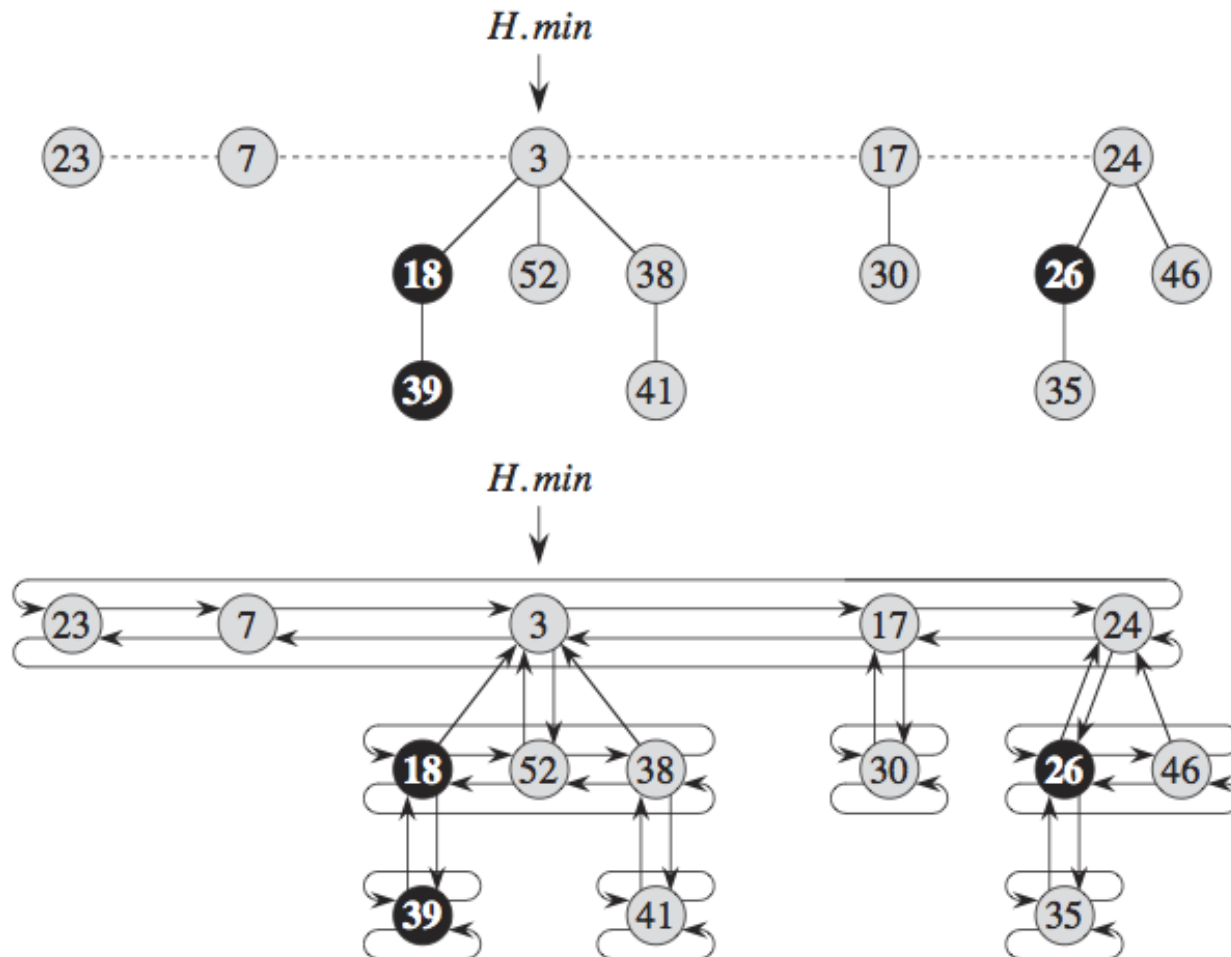


Figure: Cormen et al., Introduction to Algorithms

Simple (Lazy) Operations

Initialize-Heap H :

- $H.rootlist := H.min := null$

Merge heaps H and H' :

- concatenate root lists
- update $H.min$

Insert element e into H :

- create new one-node tree containing $e \rightarrow H'$
 - mark of root node is set to **false**
- merge heaps H and H'

Get minimum element of H :

- return $H.min$

Operation Delete-Min

Delete the node with minimum key from H and return its element:

1. $m := H.min;$
2. **if** $H.size > 0$ **then**
3. remove $H.min$ from $H.rootlist$;
4. add $H.min.child$ (list) to $H.rootlist$
5. **$H.Consolidate()$** ;

 // Repeatedly merge nodes with equal degree in the root list
 // until degrees of nodes in the root list are distinct.
 // Determine the element with minimum key
6. **return** m

Rank and Maximum Degree

Ranks of nodes, trees, heap:

Node v :

- $rank(v)$: degree of v (number of children of v)

Tree T :

- $rank(T)$: rank (degree) of root node of T

Heap H :

- $rank(H)$: maximum degree (#children) of any node in H

Assumption (n : number of nodes in H):

$$rank(H) \leq D(n)$$

- for a known function $D(n)$

Merging Two Trees

Given: Heap-ordered trees T, T' with $rank(T) = rank(T')$

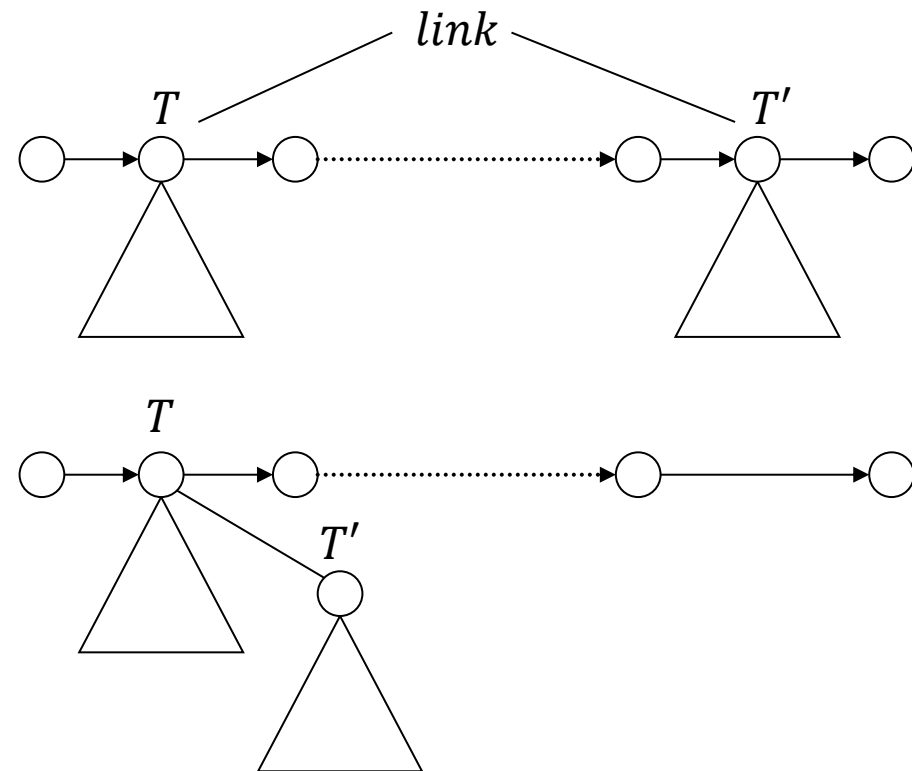
- Assume: min-key of $T <$ min-key of T'

Operation $link(T, T')$:

- Removes tree T' from root list and adds T' to child list of T

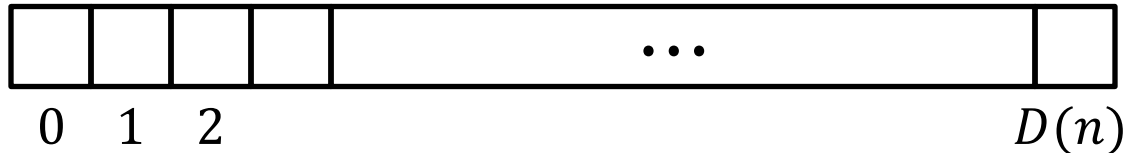
- $rank(T) := rank(T) + 1$

- $(T'.mark := \mathbf{false})$



Consolidation of Root List

Array A pointing to find roots with the same rank:



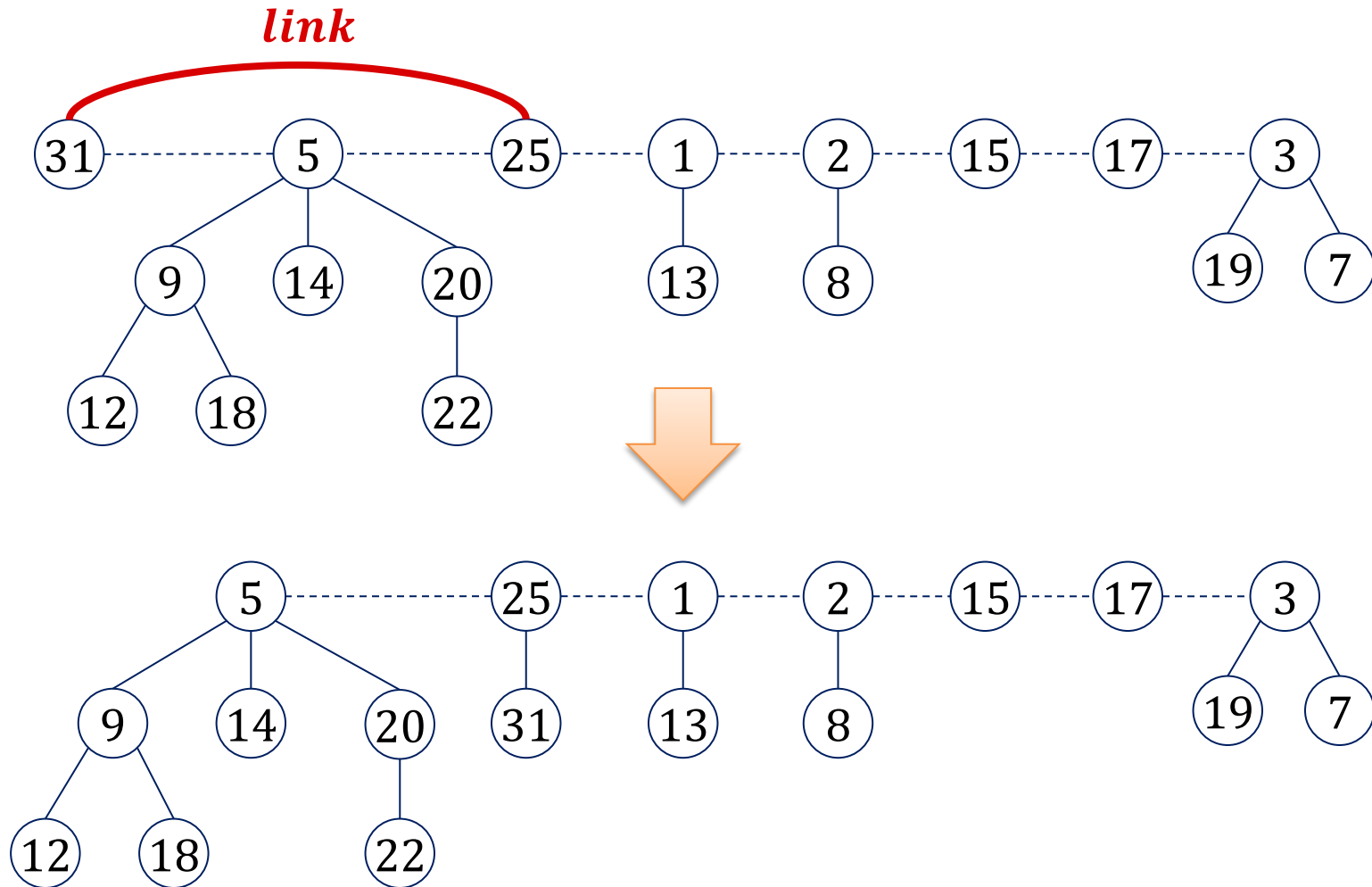
Consolidate:

1. **for** $i := 0$ **to** $D(n)$ **do** $A[i] := \text{null}$;
2. **while** $H.\text{rootlist} \neq \text{null}$ **do**
3. $T :=$ “delete and return first element of $H.\text{rootlist}$ ”
4. **while** $A[\text{rank}(T)] \neq \text{null}$ **do**
5. $T' := A[\text{rank}(T)]$;
6. $A[\text{rank}(T)] := \text{null}$;
7. $T := \text{link}(T, T')$
8. $A[\text{rank}(T)] := T$
9. Create new $H.\text{rootlist}$ and $H.\text{min}$

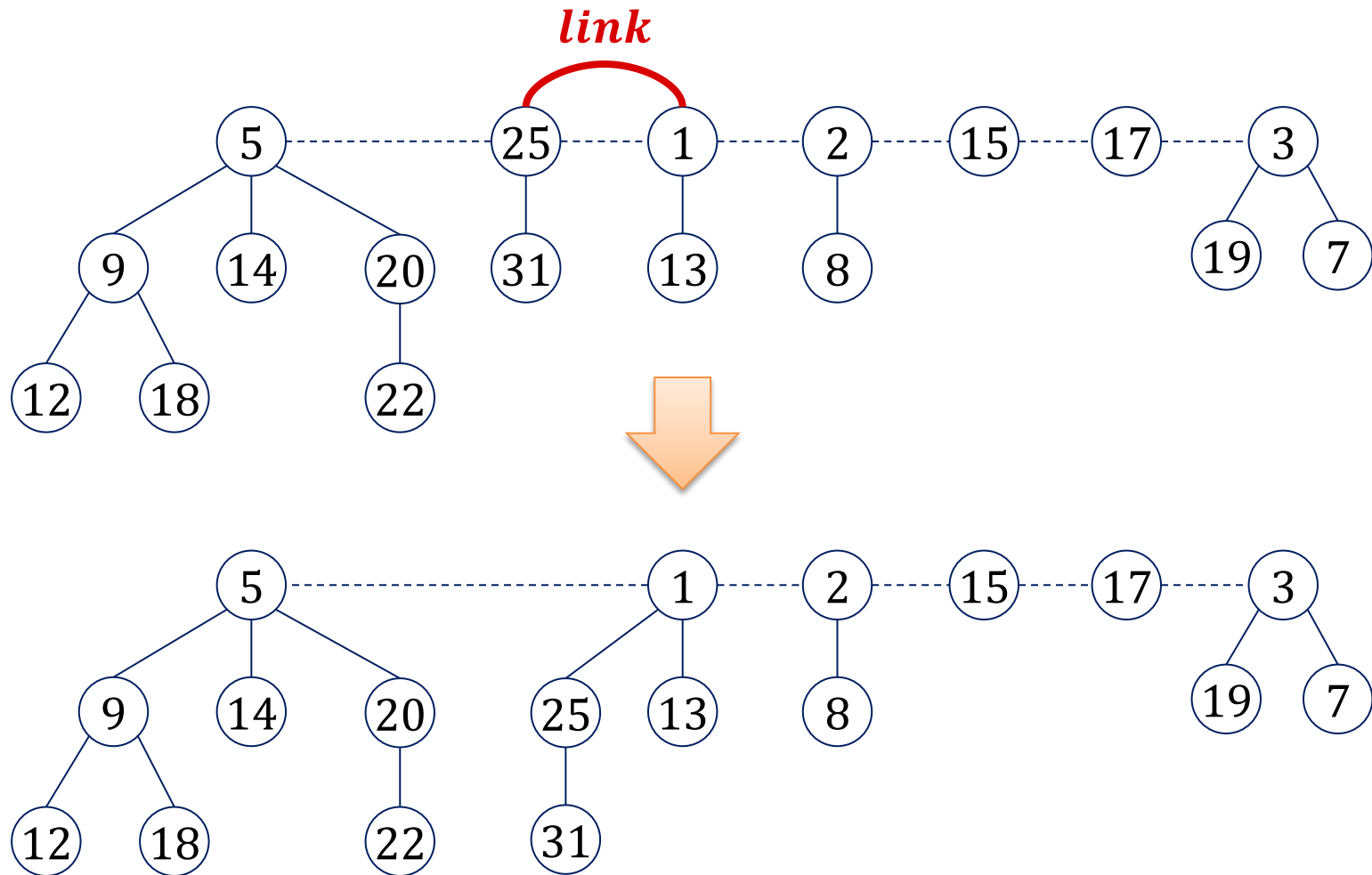
Time:

$O(|H.\text{rootlist}| + D(n))$

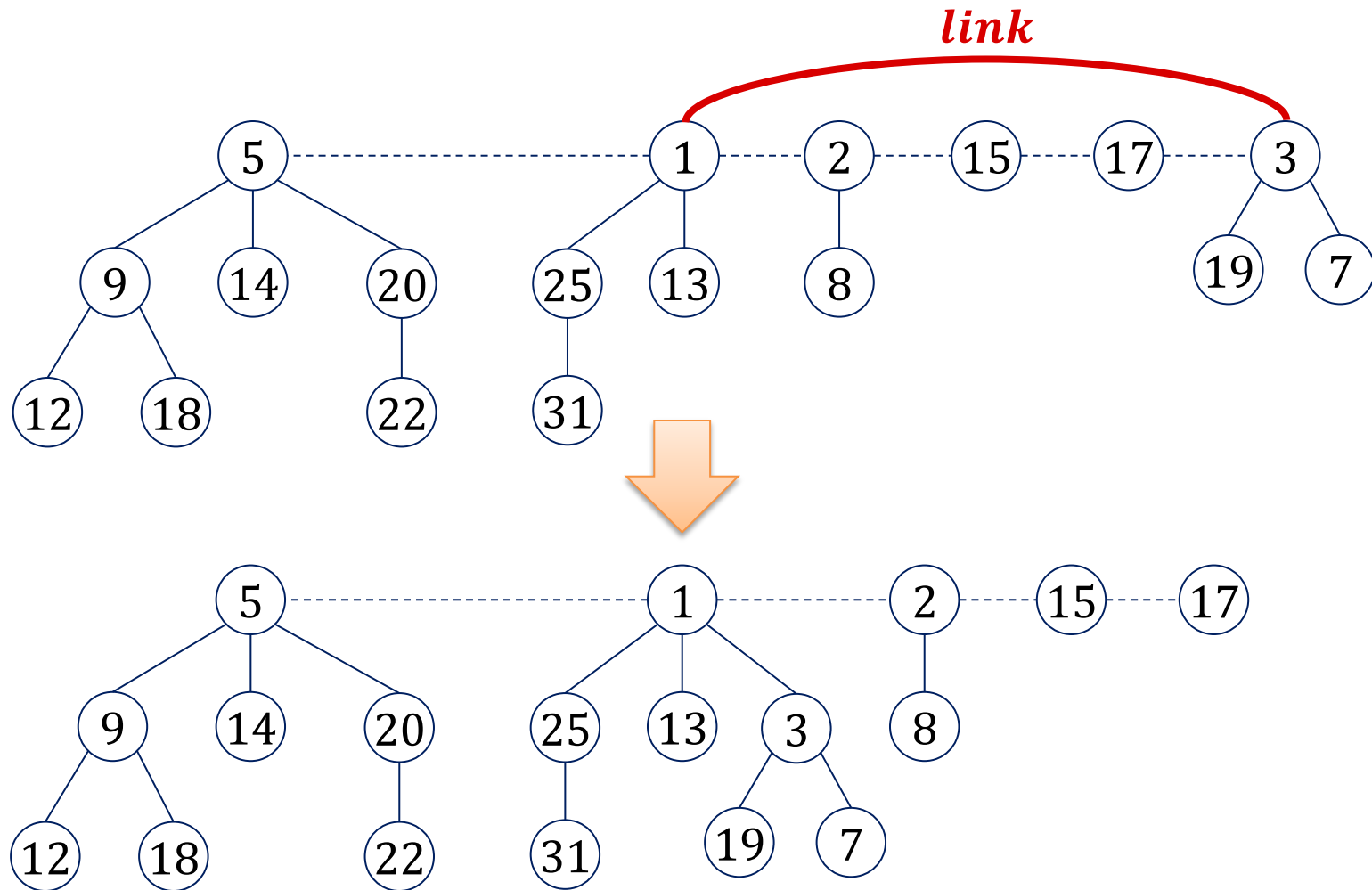
Consolidate Example



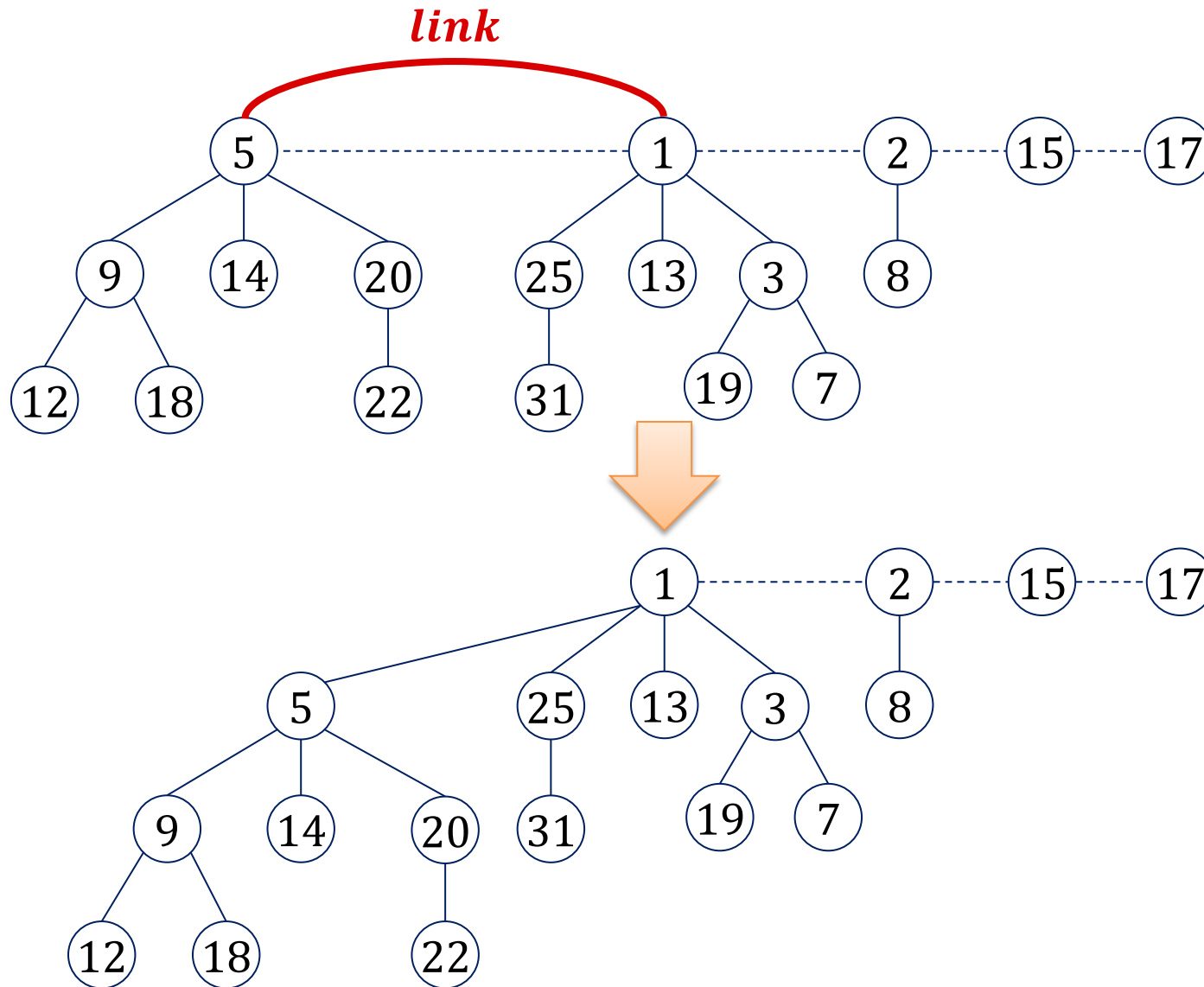
Consolidate Example



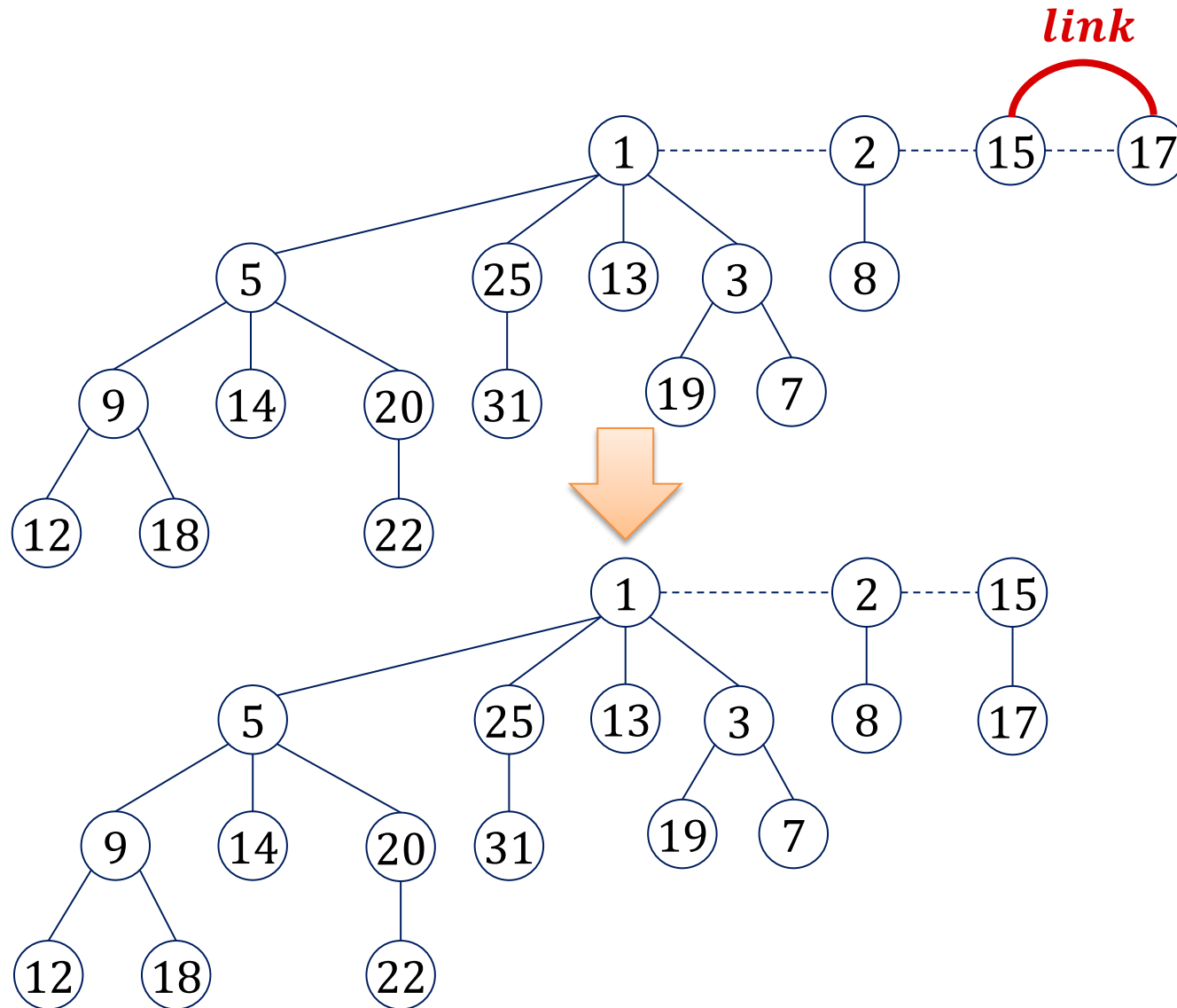
Consolidate Example



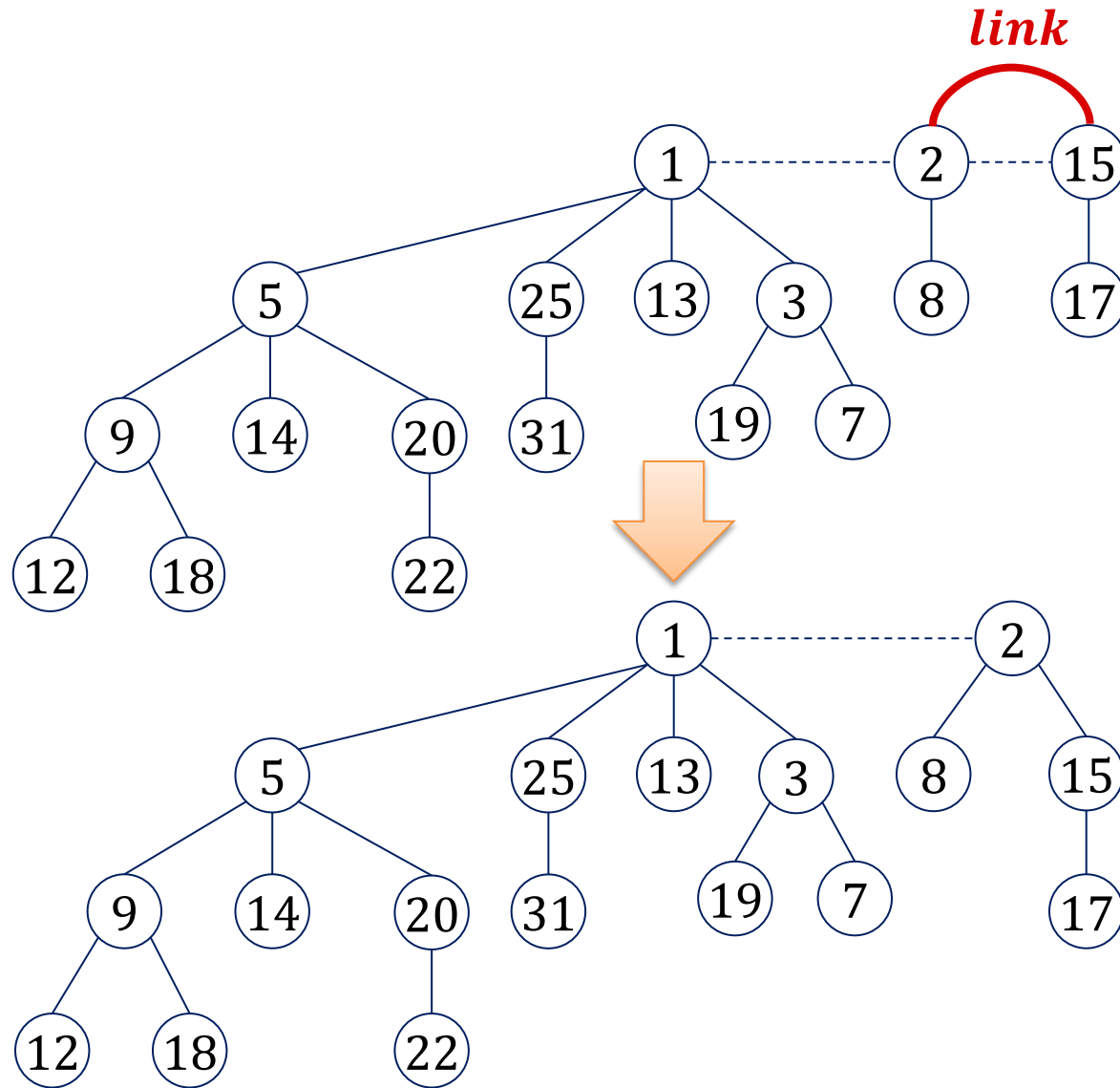
Consolidate Example



Consolidate Example



Consolidate Example



Operation Decrease-Key

Decrease-Key(v, x): (decrease key of node v to new value x)

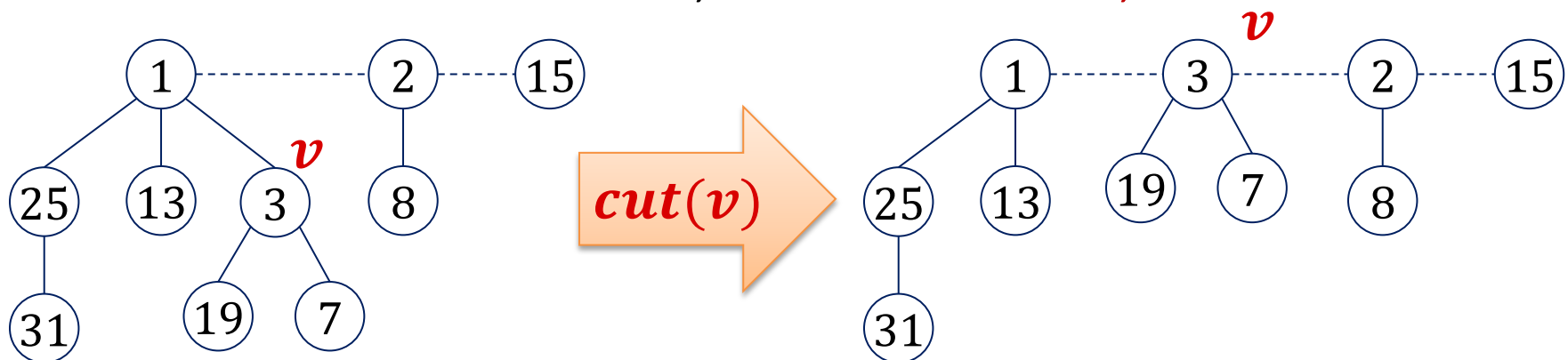
1. **if** $x \geq v.key$ **then return**;
2. $v.key := x$; update $H.min$;
3. **if** $v \in H.rootlist \vee x \geq v.parent.key$ **then return**
4. **repeat**
5. $parent := v.parent$;
6. **$H.cut(v)$** ;
7. $v := parent$;
8. **until** $\neg(v.mark) \vee v \in H.rootlist$;
9. **if** $v \notin H.rootlist$ **then** $v.mark := true$;

Operation $\text{Cut}(v)$

Operation $H.\text{cut}(v)$:

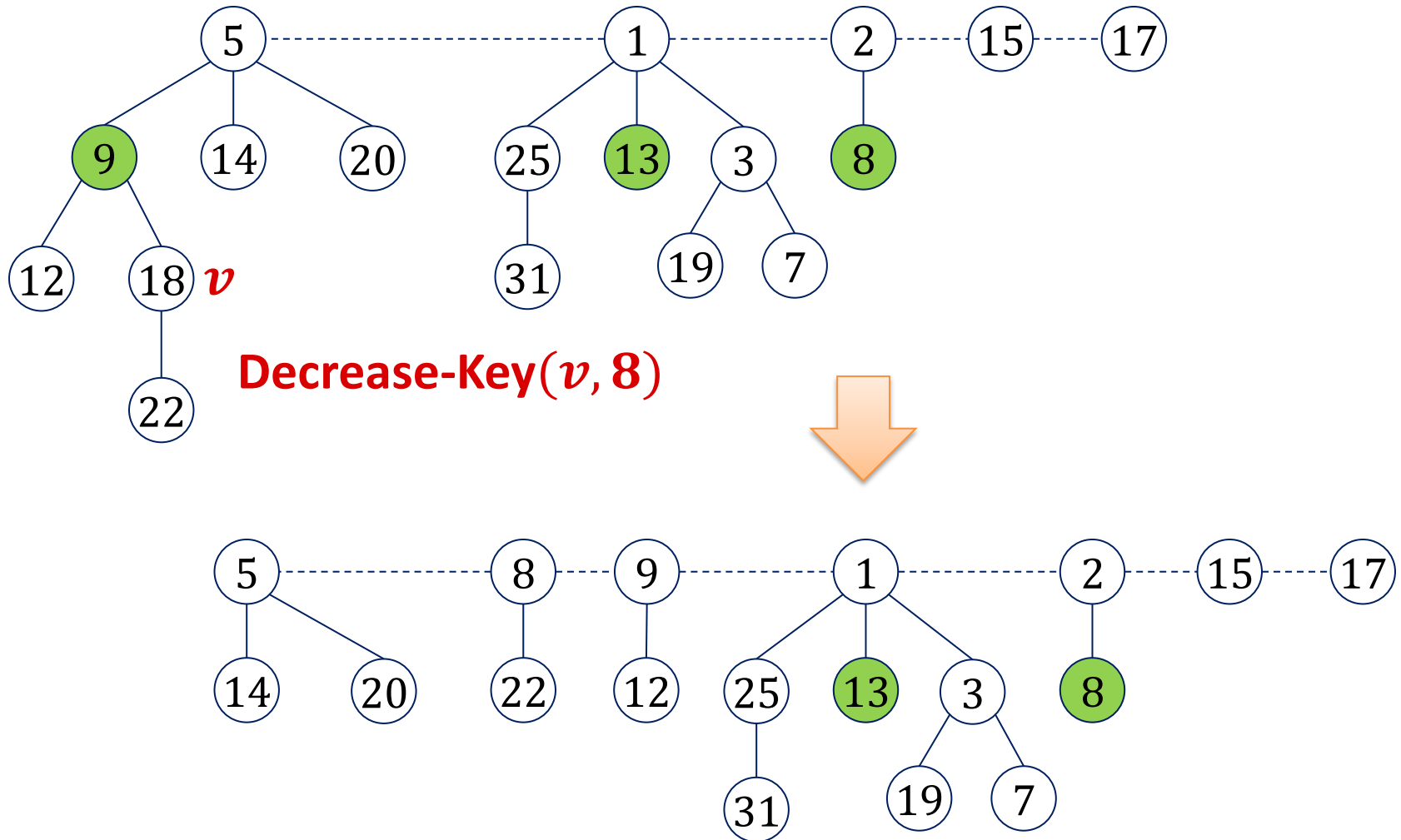
- Cuts v 's sub-tree from its parent and adds v to rootlist

- if $v \notin H.\text{rootlist}$ then**
- // cut the link between v and its parent
- $\text{rank}(v.\text{parent}) := \text{rank}(v.\text{parent}) - 1$;
- remove v from $v.\text{parent}.\text{child}$ (list)
- $v.\text{parent} := \text{null}$;
- add v to $H.\text{rootlist}$; $v.\text{mark} := \text{false}$;



Decrease-Key Example

- Green nodes are marked



Fibonacci Heaps Marks

- Nodes in the root list (the **tree roots**) are always **unmarked**
→ If a node is added to the root list (insert, decrease-key), the mark of the node is set to false.
- Nodes not in the root list can only get **marked** when a **subtree is cut** in a decrease-key operation
- A node v is **marked** if and only if v is **not in the root list** and v **has lost a child** since v was attached to its current parent
 - a node can only change its parent by being moved to the root list

Fibonacci Heap Marks

History of a node v :

v is being linked to a node



$v.mark = false$

a child of v is cut



$v.mark := true$

a second child of v is cut



**$H.cut(v);$
 $v.mark := false$**

- Hence, the boolean value $v.mark$ indicates whether node v has lost a child since the last time v was made the child of another node.
- Nodes v in the root list always have $v.mark = false$

Cost of Delete-Min & Decrease-Key

Delete-Min:

1. Delete min. root r and add $r.child$ to $H.rootlist$
time: $O(1)$
 2. Consolidate $H.rootlist$
time: $O(\text{length of } H.rootlist + D(n))$
- Step 2 can potentially be linear in n (size of H)

Decrease-Key (at node v):

1. If new key $<$ parent key, cut sub-tree of node v
time: $O(1)$
 2. Cascading cuts up the tree as long as nodes are marked
time: $O(\text{number of consecutive marked nodes})$
- Step 2 can potentially be linear in n

Exercises: Both operations can take $\Theta(n)$ time in the worst case!

Cost of Delete-Min & Decrease-Key

- Cost of delete-min and decrease-key can be $\Theta(n)$...
 - Seems a large price to pay to get insert and merge in $O(1)$ time
- Maybe, the operations are efficient most of the time?
 - It seems to require a lot of operations to get a long rootlist and thus, an expensive consolidate operation
 - In each decrease-key operation, at most one node gets marked: We need a lot of decrease-key operations to get an expensive decrease-key operation
- Can we show that the **average cost** per operation is small?
- We can \rightarrow requires **amortized analysis**

Fibonacci Heaps Complexity

- Worst-case cost of a single delete-min or decrease-key operation is $\Omega(n)$
- Can we prove a small worst-case amortized cost for delete-min and decrease-key operations?

Recall:

- Data structure that allows operations O_1, \dots, O_k
- We say that operation O_p has amortized cost a_p if for every execution the total time is

$$T \leq \sum_{p=1}^k n_p \cdot a_p ,$$

where n_p is the number of operations of type O_p

Amortized Cost of Fibonacci Heaps

- Initialize-heap, is-empty, get-min, insert, and merge have **worst-case cost $O(1)$**
- Delete-min has **amortized cost $O(\log n)$**
- Decrease-key has **amortized cost $O(1)$**
- Starting with an empty heap, any sequence of n operations with at most n_d delete-min operations has total cost (time)

$$T = O(n + n_d \log n).$$

- We will now need the marks...
- Cost for Dijkstra: $O(|E| + |V| \log |V|)$

Fibonacci Heaps: Marks

Cycle of a node:

1. Node v is removed from root list and linked to a node
 $v.mark = false$
2. Child node u of v is cut and added to root list
 $v.mark := true$
3. Second child of v is cut
node v is cut as well and moved to root list
 $v.mark := false$

The boolean value $v.mark$ indicates whether node v has lost a child since the last time v was made the child of another node.

Potential Function

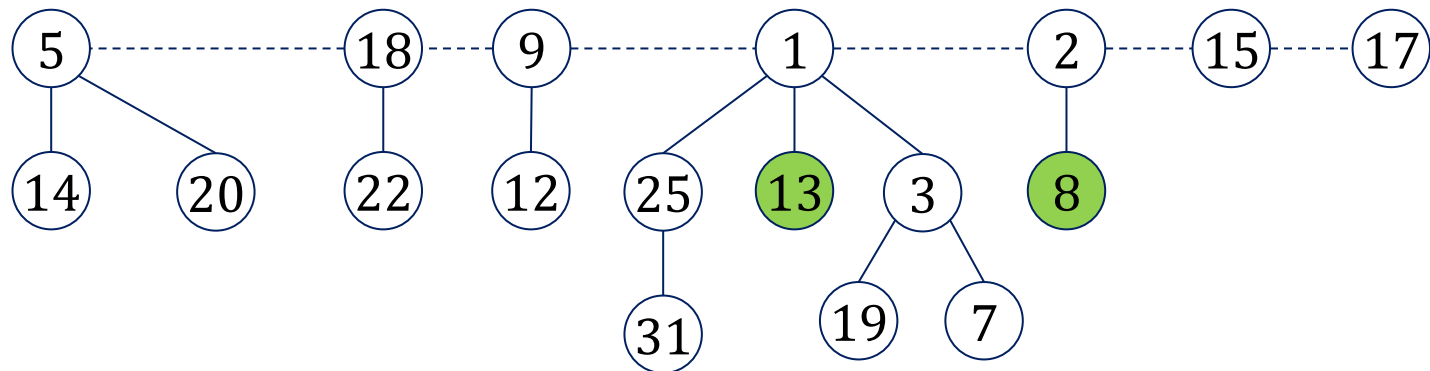
System state characterized by two parameters:

- **R** : number of trees (length of $H.rootlist$)
- **M** : number of marked nodes (not in the root list)

Potential function:

$$\Phi := R + 2M$$

Example:



- $R = 7, M = 2 \rightarrow \Phi = 11$

Actual Time of Operations

- Operations: ***initialize-heap, is-empty, insert, get-min, merge***

actual time: $O(1)$

- Normalize unit time such that

$$t_{init}, t_{is-empty}, t_{insert}, t_{get-min}, t_{merge} \leq 1$$

- Operation ***delete-min***:

- Actual time: $O(\text{length of } H.\text{rootlist} + D(n))$
- Normalize unit time such that

$$t_{del-min} \leq D(n) + \text{length of } H.\text{rootlist}$$

- Operation ***decrease-key***:

- Actual time: $O(\text{length of path to next unmarked ancestor})$
- Normalize unit time such that

$$t_{decr-key} \leq \text{length of path to next unmarked ancestor}$$

Amortized Times

Assume operation i is of type:

- **initialize-heap:**
 - actual time: $t_i \leq 1$, potential: $\Phi_{i-1} = \Phi_i = 0$
 - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$
- **is-empty, get-min:**
 - actual time: $t_i \leq 1$, potential: $\Phi_i = \Phi_{i-1}$ (heap doesn't change)
 - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$
- **merge:**
 - Actual time: $t_i \leq 1$
 - combined potential of both heaps: $\Phi_i = \Phi_{i-1}$
 - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$

Amortized Time of Insert

Assume that operation i is an *insert* operation:

- **Actual time:** $t_i \leq 1$
- **Potential function:**
 - M remains unchanged (no nodes are marked or unmarked, no marked nodes are moved to the root list)
 - R grows by 1 (one element is added to the root list)

$$M_i = M_{i-1}, \quad R_i = R_{i-1} + 1$$
$$\Phi_i = \Phi_{i-1} + 1$$

- **Amortized time:**

$$a_i = t_i + \Phi_i - \Phi_{i-1} \leq 2$$

Amortized Time of Delete-Min

Assume that operation i is a *delete-min* operation:

Actual time: $t_i \leq D(n) + |H.rootlist|$

Potential function $\Phi = R + 2M$:

- R : changes from $|H.rootlist|$ to at most $D(n)$
- M : (# of marked nodes that are not in the root list)
 - no new marks
 - if node v is moved away from root list, $v.mark$ is set to false
→ value of M does not increase!

$$M_i \leq M_{i-1}, \quad R_i \leq R_{i-1} + D(n) - |H.rootlist|$$
$$\Phi_i \leq \Phi_{i-1} + D(n) - |H.rootlist|$$

Amortized Time:

$$a_i = t_i + \Phi_i - \Phi_{i-1} \leq 2D(n)$$

Amortized Time of Decrease-Key

Assume that operation i is a *decrease-key* operation at node u :

Actual time: $t_i \leq$ length of path to next unmarked ancestor v

Potential function $\Phi = R + 2M$:

- Assume, node u and nodes u_1, \dots, u_k are moved to root list
 - u_1, \dots, u_k are marked and moved to root list, v . mark is set to true

Amortized Time of Decrease-Key

Assume that operation i is a *decrease-key* operation at node u :

Actual time: $t_i \leq$ length of path to next unmarked ancestor v

Potential function $\Phi = R + 2M$:

- Assume, node u and nodes u_1, \dots, u_k are moved to root list
 - u_1, \dots, u_k are marked and moved to root list, v . mark is set to true
- $\geq k$ marked nodes go to root list, ≤ 1 node gets newly marked
- R grows by $\leq k + 1$, M grows by 1 and is decreased by $\geq k$


$$R_i \leq R_{i-1} + k + 1, \quad M_i \leq M_{i-1} + 1 - k$$

$$\Phi_i \leq \Phi_{i-1} + (k + 1) - 2(k - 1) = \Phi_{i-1} + 3 - k$$

Amortized time:

$$a_i = t_i + \Phi_i - \Phi_{i-1} \leq k + 1 + 3 - k = 4$$

Complexities Fibonacci Heap

- Initialize-Heap: $O(1)$
 - Is-Empty: $O(1)$
 - Insert: $O(1)$
 - Get-Min: $O(1)$
 - Delete-Min: $O(D(n))$
 - Decrease-Key: $O(1)$
 - Merge (heaps of size m and $n, m \leq n$): $O(1)$
 - How large can $D(n)$ get?
- amortized**
- 

Rank of Children

Lemma:

Consider a node v of rank k and let u_1, \dots, u_k be the children of v in the order in which they were linked to v . Then,

$$\mathit{rank}(u_i) \geq i - 2.$$

Proof:

Size of Trees

Fibonacci Numbers:

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

Lemma:

In a Fibonacci heap, the size of the sub-tree of a node v with rank k is at least F_{k+2} .

Proof:

- S_k : minimum size of the sub-tree of a node of rank k

$$S_0 = 1, \quad S_1 = 2, \quad \forall k \geq 2: S_k \geq 2 + \sum_{i=0}^{k-2} S_i$$

- Claim about Fibonacci numbers:

$$\forall k \geq 0: F_{k+2} = 1 + \sum_{i=0}^k F_i$$

Size of Trees

$$S_0 = 1, S_1 = 2, \forall k \geq 2: S_k \geq 2 + \sum_{i=0}^{k-2} S_i, \quad F_{k+2} = 1 + \sum_{i=0}^k F_i$$

- Claim of lemma: $S_k \geq F_{k+2}$

Size of Trees

Lemma:

In a Fibonacci heap, the size of the sub-tree of a node v with rank k is at least F_{k+2} .

Theorem:

The maximum rank of a node in a Fibonacci heap of size n is at most

$$D(n) = O(\log n).$$

Proof:

- The Fibonacci numbers grow exponentially:

$$F_k = \frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1 + \sqrt{5}}{2} \right)^k - \left(\frac{1 - \sqrt{5}}{2} \right)^k \right)$$

- For $D(n) \geq k$, we need $n \geq F_{k+2}$ nodes.

Summary: Binomial and Fibonacci Heaps

	Binary Heap	Fibonacci Heap
<i>initialize</i>	$O(1)$	$O(1)$
<i>insert</i>	$O(\log n)$	$O(1)$
<i>get-min</i>	$O(1)$	$O(1)$
<i>delete-min</i>	$O(\log n)$	$O(\log n)$ *
<i>decrease-key</i>	$O(\log n)$	$O(1)$ *
<i>merge</i>	$O(m \cdot \log n)$	$O(1)$
<i>is-empty</i>	$O(1)$	$O(1)$

* amortized time