Algorithm Theory

Chapter 5
Data Structures

Part IV:
Fibonacci Heaps, Algorithm Description

Fabian Kuhn

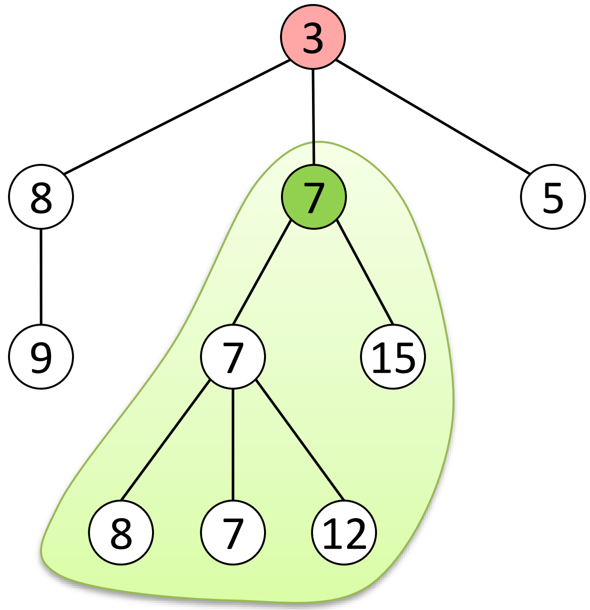
Fibonacci Heaps

Structure:

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

Min-Heap Property:

Key of a node $v \leq$ keys of all nodes in any sub-tree of v



Fibonacci Heaps

Structure:

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

Variables:

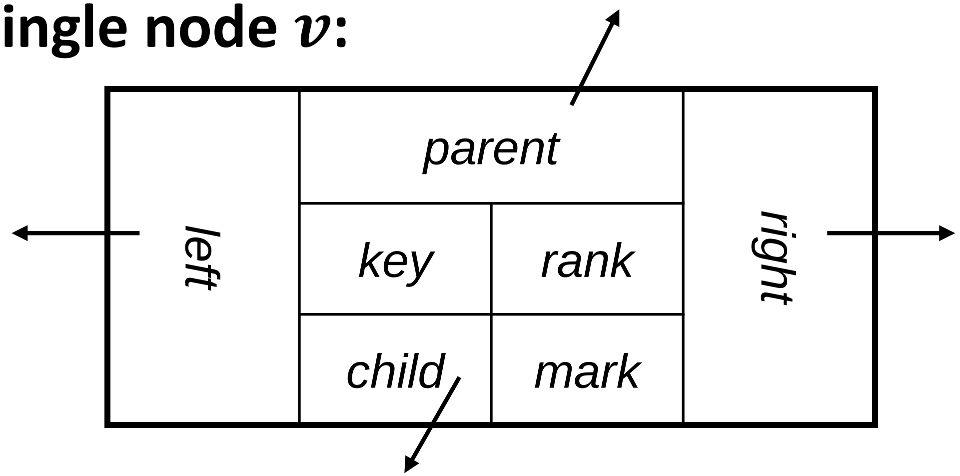
- $H.min$: root of the tree containing the (a) minimum key
- $H.rootlist$: circular, doubly linked, unordered list containing the roots of all trees
- $H.size$: number of nodes currently in H

Lazy Merging:

- To reduce the number of trees, sometimes, trees need to be merged
- Lazy merging: Do not merge as long as possible...

Trees in Fibonacci Heaps

Structure of a single node v:



- $v.child$: points to circular, doubly linked and unordered list of the children of v
- $v.left, v.right$: pointers to siblings (in doubly linked list)
- $v.mark$: will be used later…

Advantages of circular, doubly linked lists:

- Deleting an element takes constant time
- Concatenating two lists takes constant time

Example

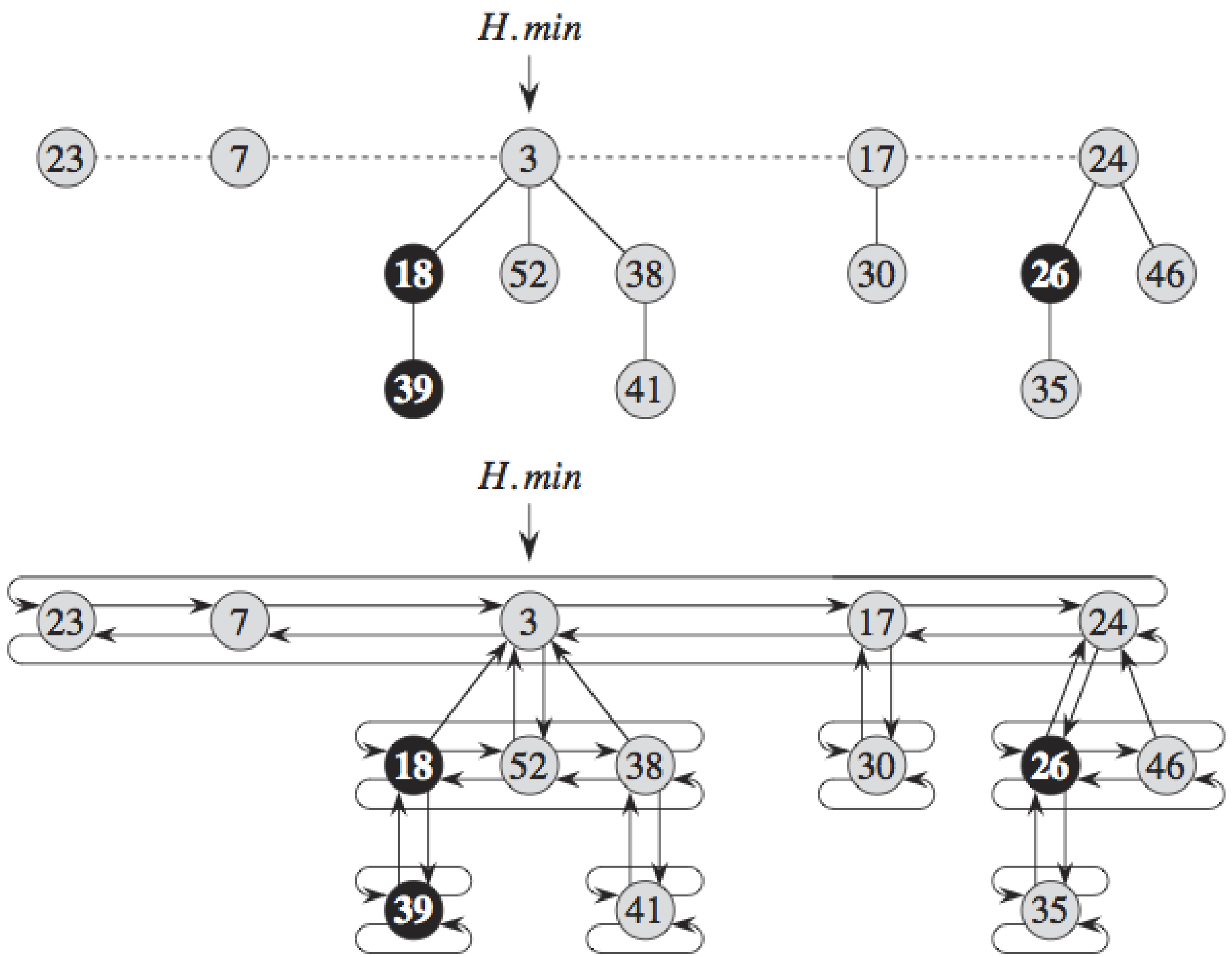


Figure: Cormen et al., Introduction to Algorithms

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:

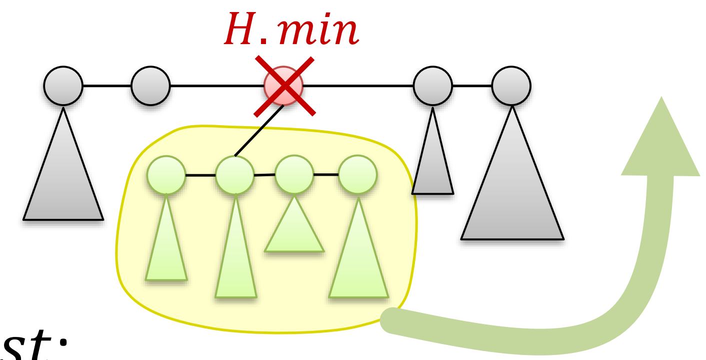
- create new one-node tree containing e → H'
  - mark of root node is set to $\mathbf{false}$
- merge heaps H and H'

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. $\boldsymbol{H.Consolidate();}$

// 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:

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

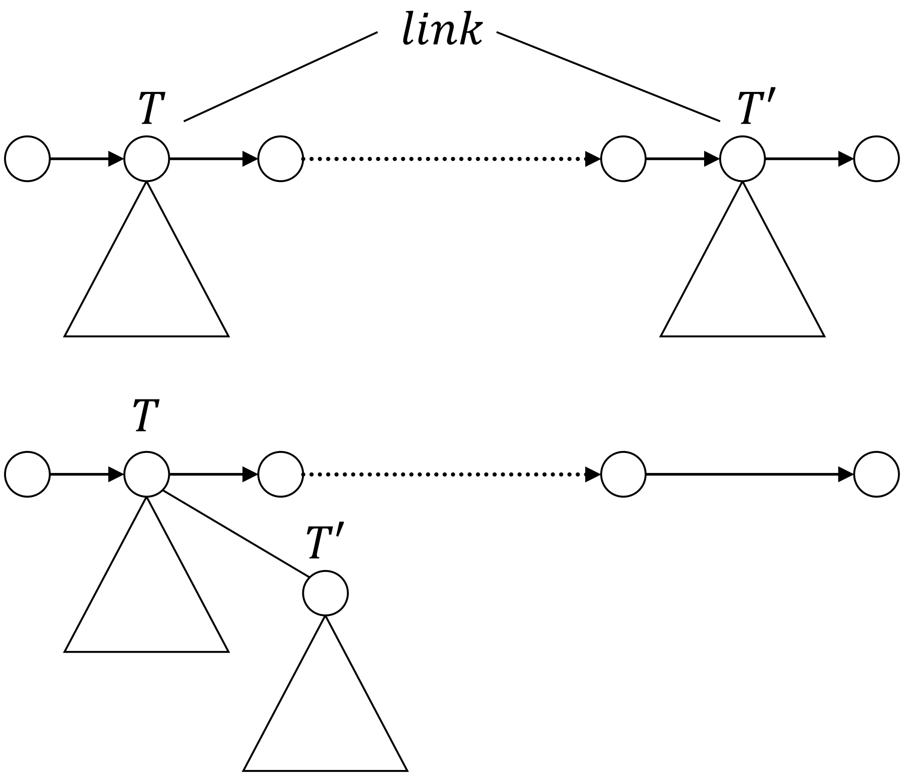
Merging Two Trees

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

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

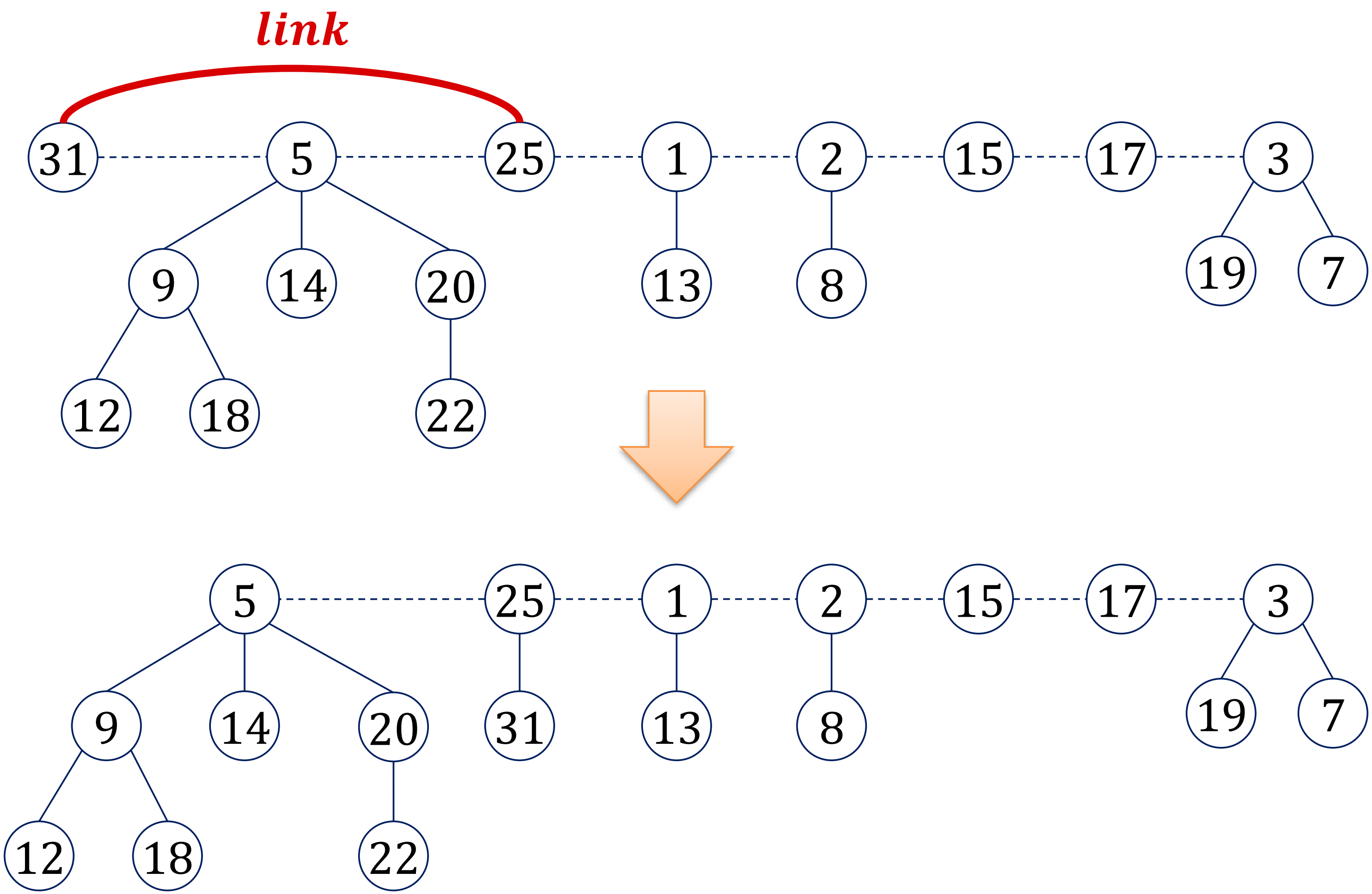
Operation $\boldsymbol{link(T, T')}$:

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

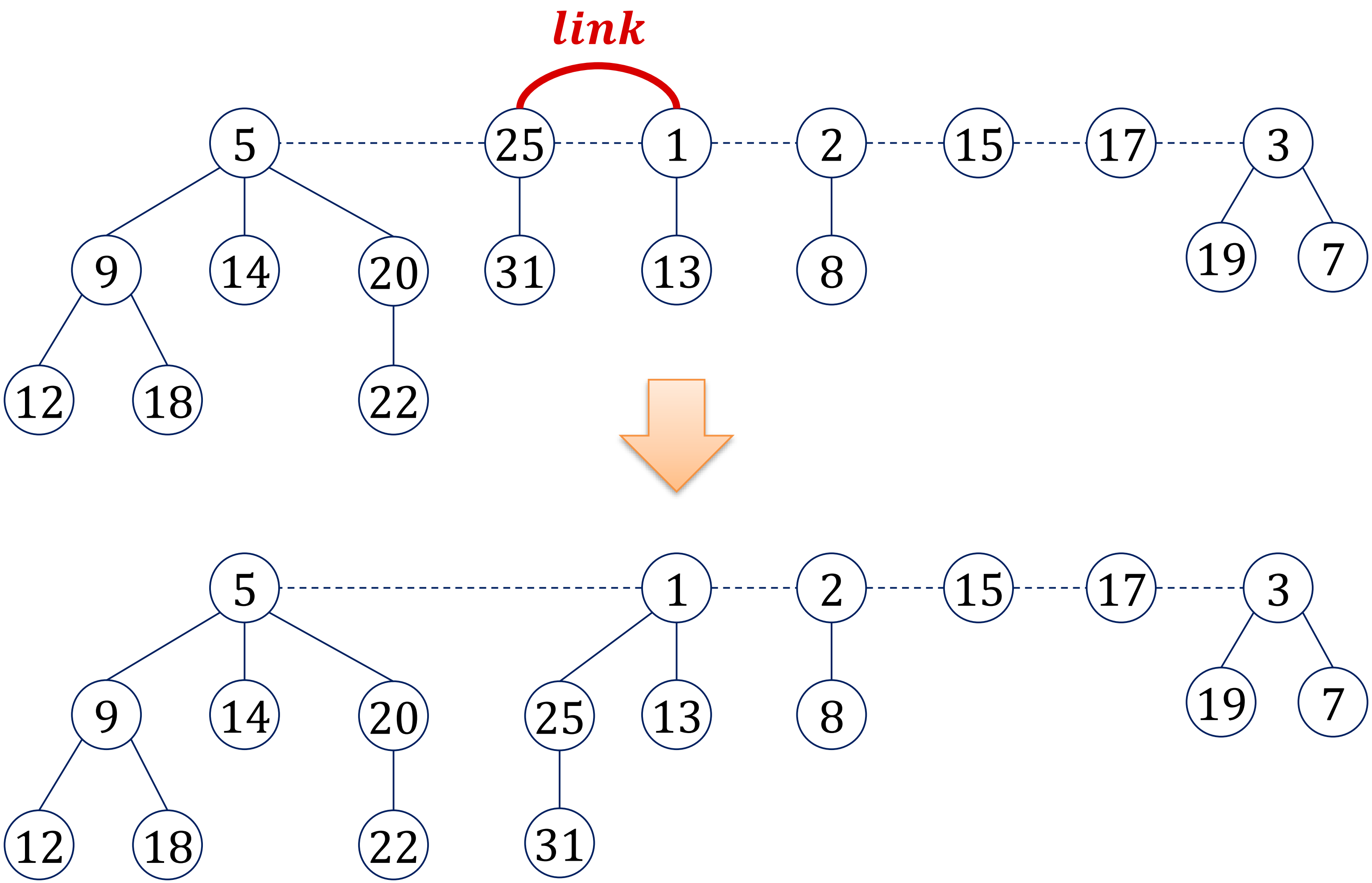


- $rank(T) \coloneqq rank(T) + 1$
- $(T'.mark = \textbf{false})$

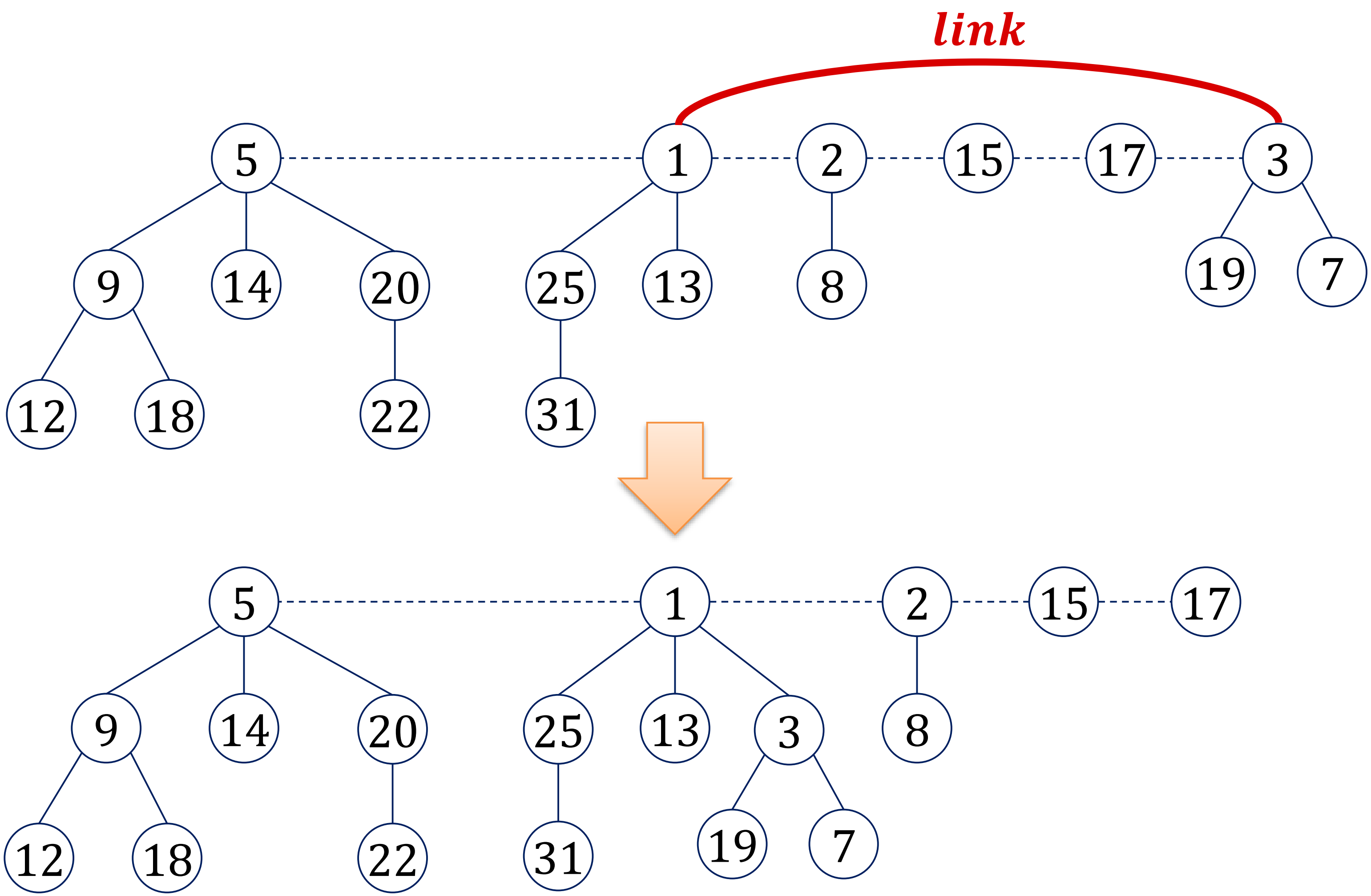
Consolidate Example



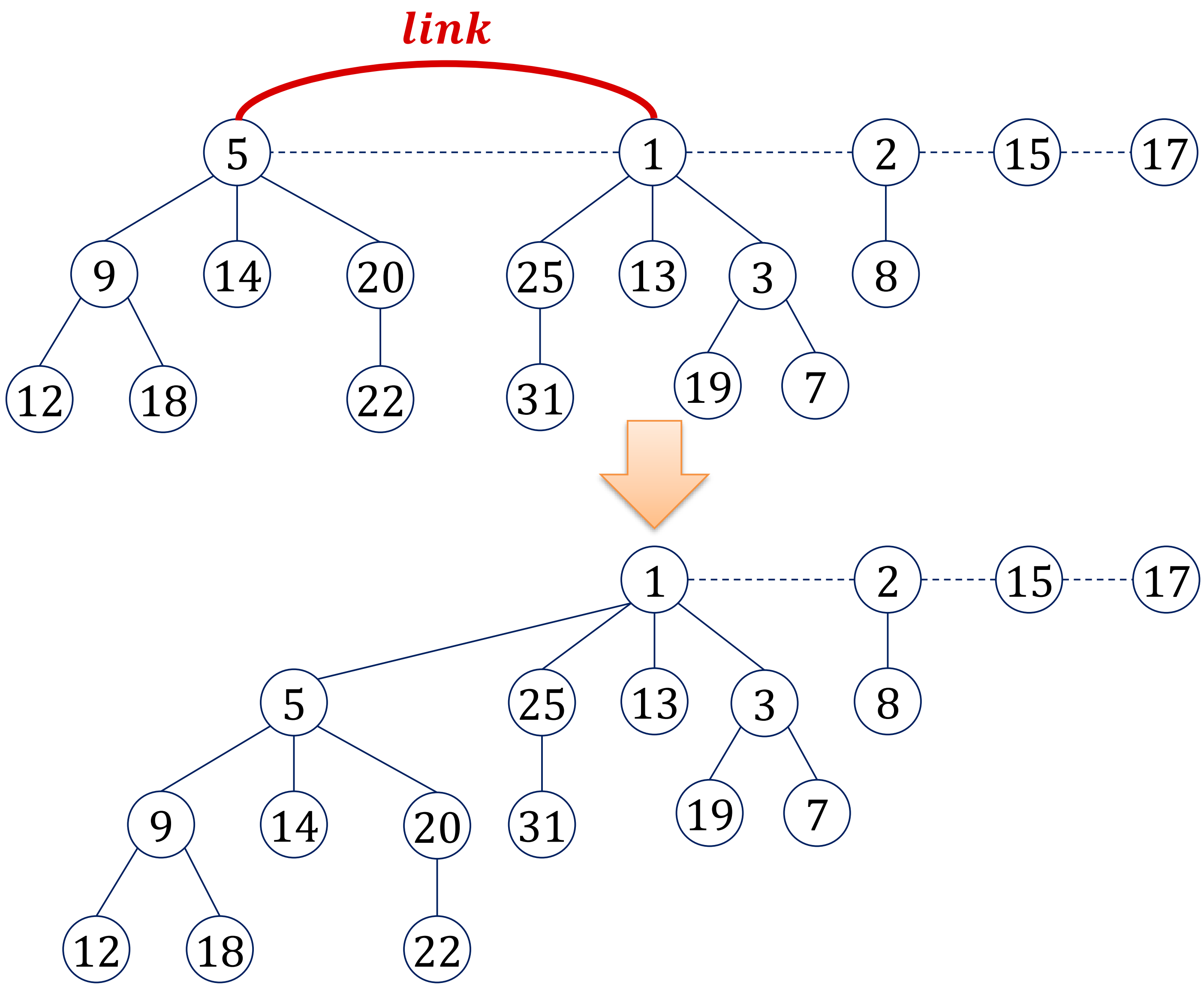
Consolidate Example



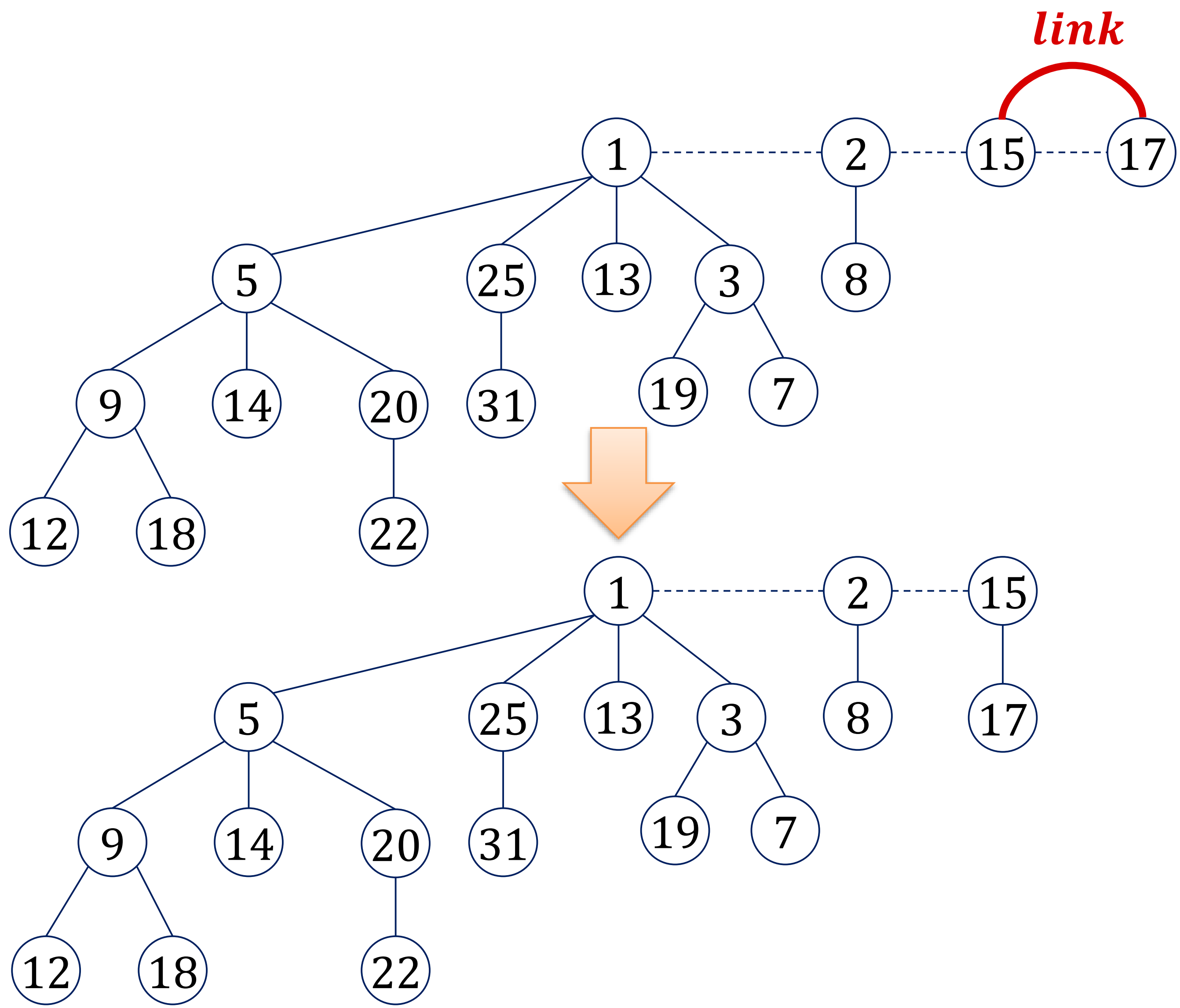
Consolidate Example



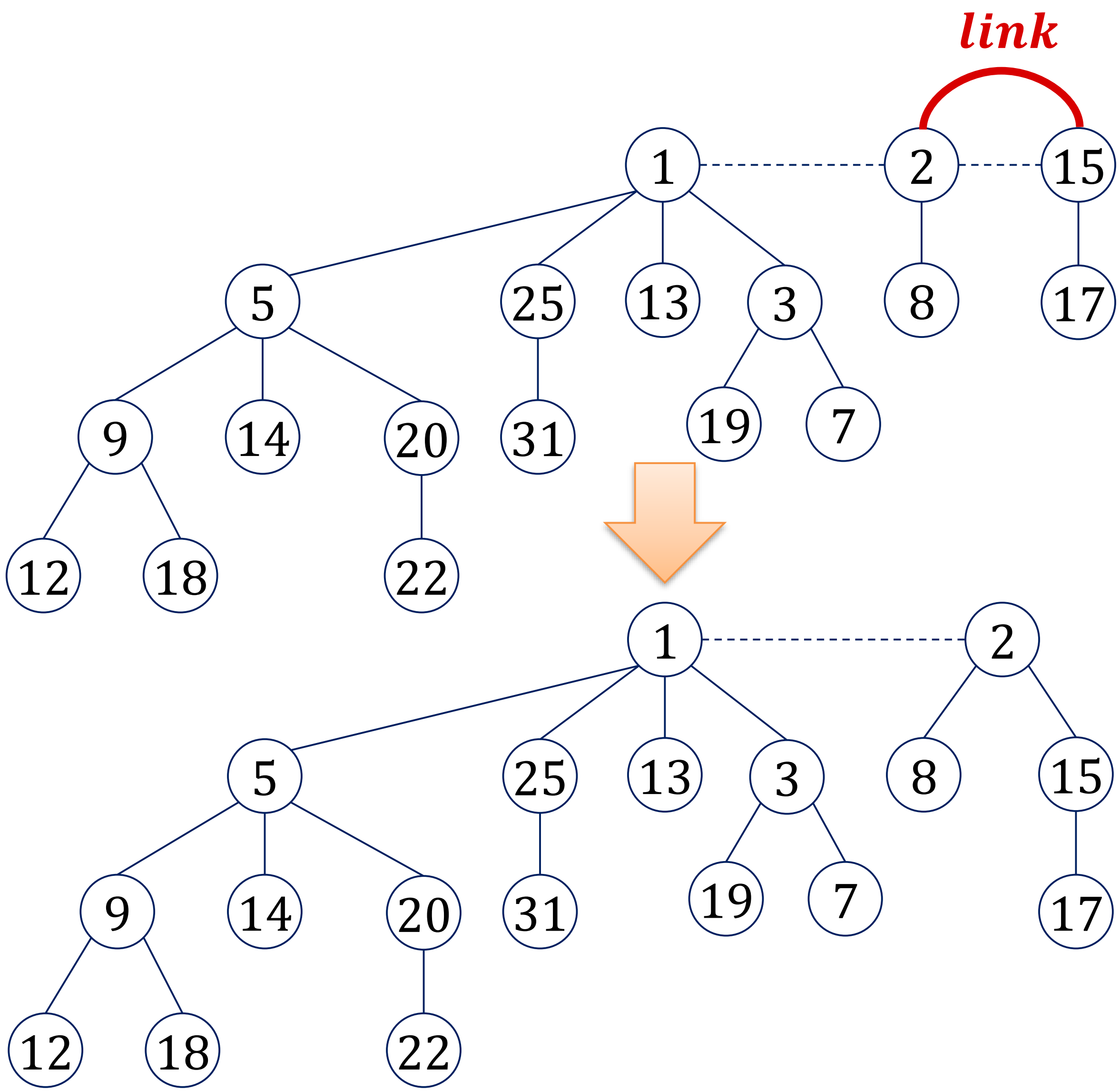
Consolidate Example



Consolidate Example

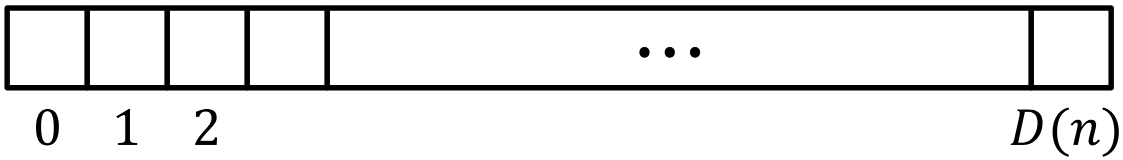


Consolidate Example



Consolidation of Root List

Array A pointing to find roots with the same rank:



Consolidate:

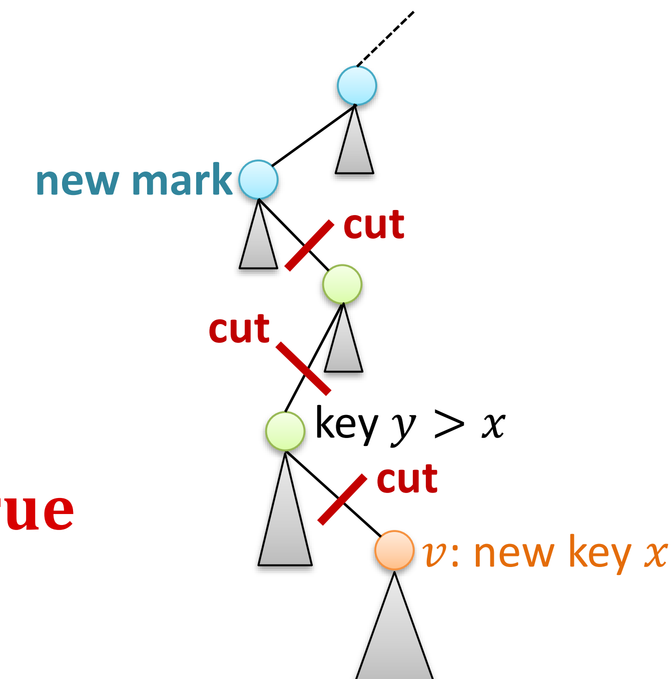
1. **for** $i := 0$ **to** $D(n)$ **do** $A[i] :=$ null
2. **while** $H.rootlist \neq$ null **do**
3. $T :=$ "delete and return first element of $H.rootlist$"
4.   **while** $A[rank(T)] \neq$ null **do**
5. $T' := A[rank(T)]$
6. $A[rank(T)] := null$
7. $T := link(T, T')$
8. $A[rank(T)] := T$
9. Create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

Time:
$\boldsymbol{O(|H.rootlist| + D(n))}$

Operation Decrease-Key

Decrease-Key$(\boldsymbol{v}, \boldsymbol{x})$**:** (decrease key of node v to new value x)

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

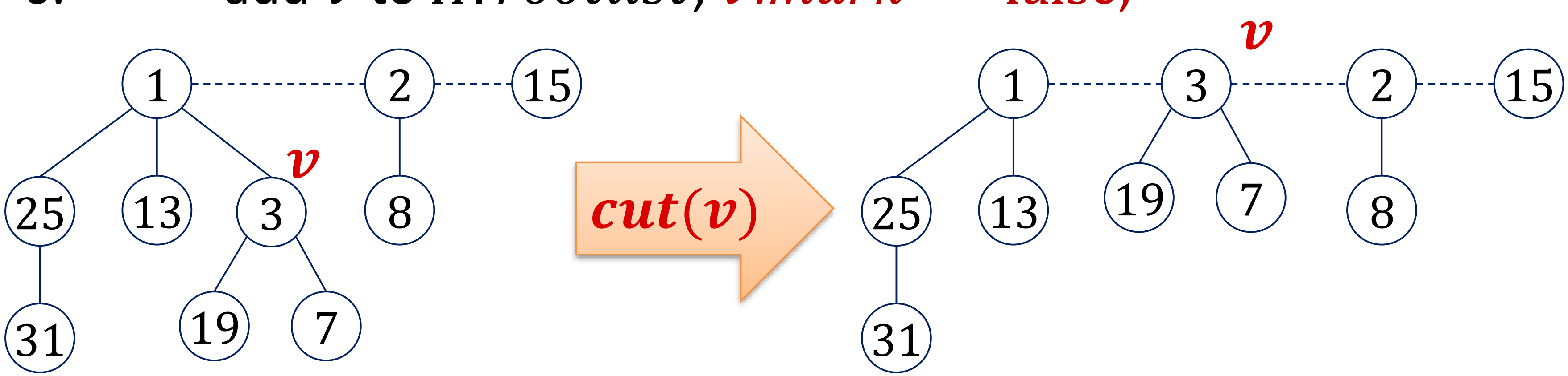


Operation Cut(v)

Operation $H.cut(v)$:

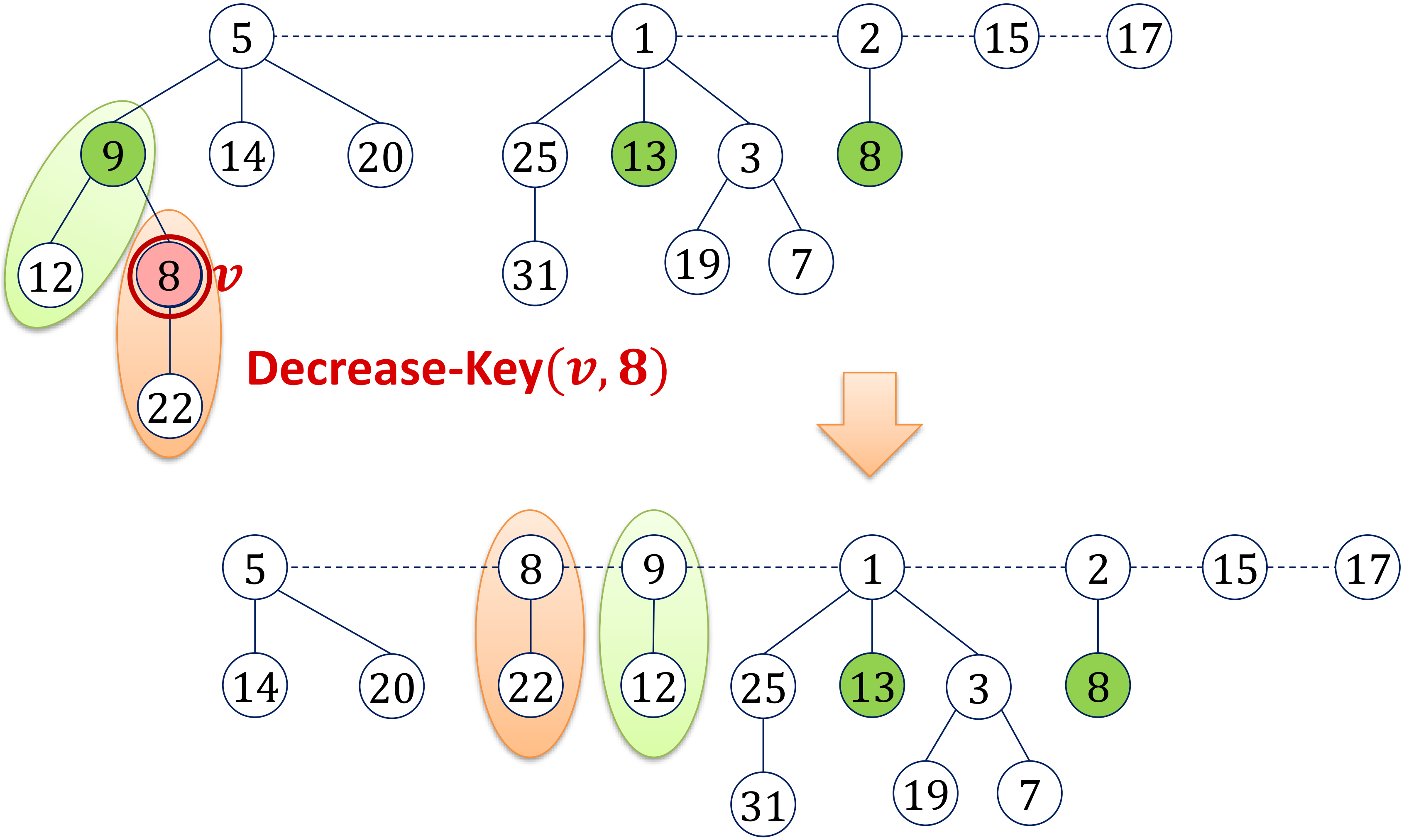
- Cuts v's sub-tree from its parent and adds v to rootlist

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



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 ⇨ $\boldsymbol{v.mark = \mathbf{false}}$

a child of v is cut ⇨ $\boldsymbol{v.mark \coloneqq \mathbf{true}}$

a second child of v is cut ⇨ $\boldsymbol{H.cut(v);}$
$\boldsymbol{v.mark \coloneqq false}$

- 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 $v.mark = \text{false}$