Chapter 4
Data Structures

Algorithm Theory
WS 2014/15

Fabian Kuhn

Priority Queue / Heap

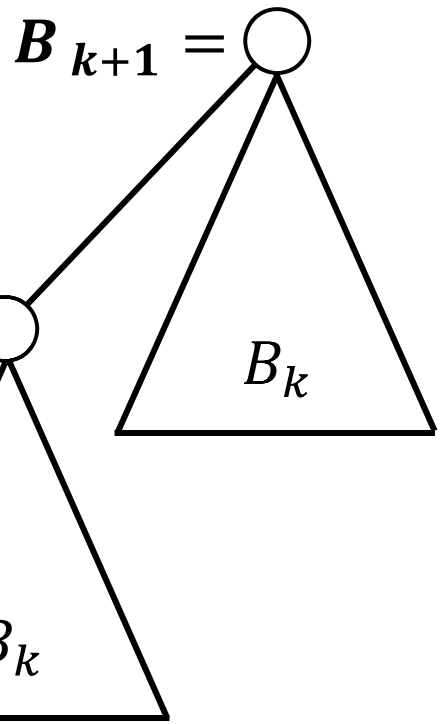
- Stores (*key,data*) pairs (like dictionary)
- But, different set of operations:

- **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*
  - *entry*: pointer to entry in data structure
- **Merge**: merges two heaps into one

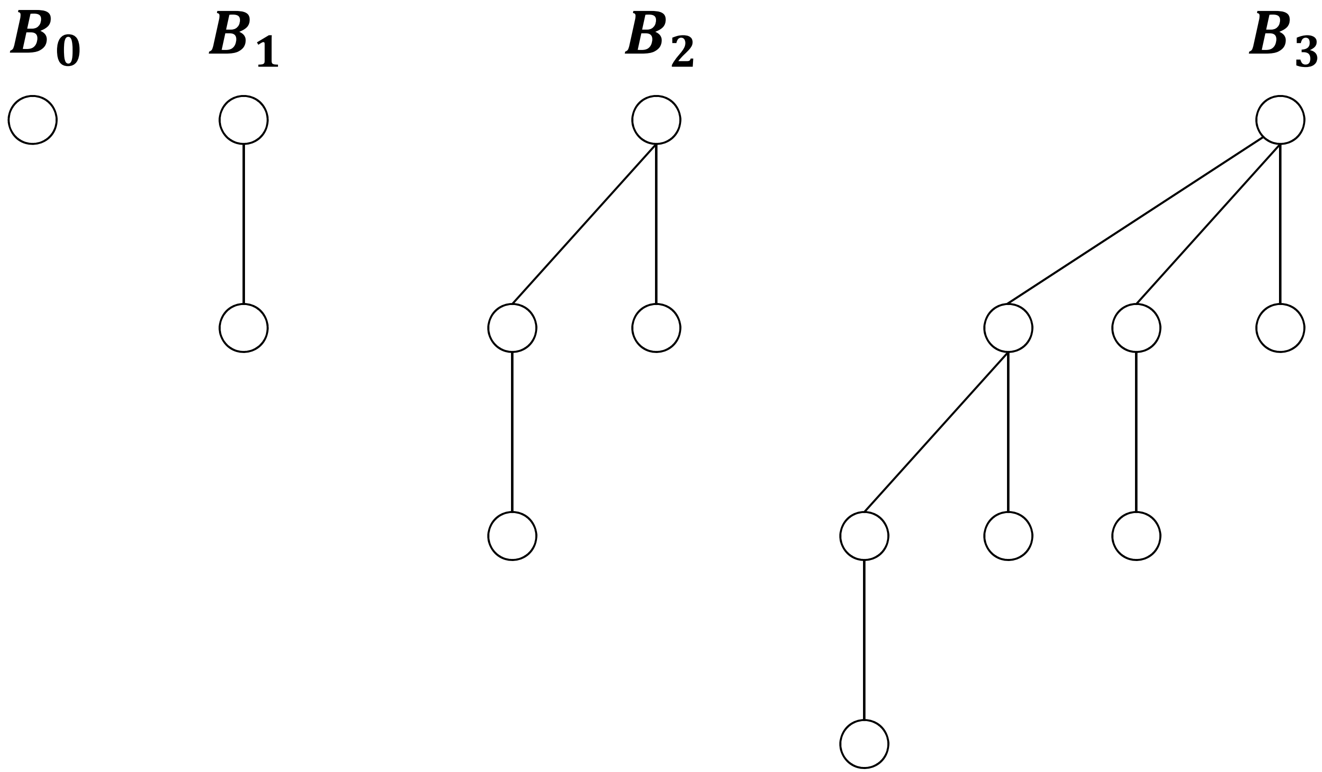
Definition: Binomial Tree

Binomial tree B_k of order k $(n \geq 0)$:

$\boldsymbol{B_0} =$ ○



Binomial Trees



Properties

1. Tree B_k has 2^k nodes

2. Height of tree B_k is k

3. Root degree of B_k is k

4. In B_k, there are exactly $\binom{k}{i}$ nodes at depth i

Binomial Heap

- Keys are stored in nodes of binomial trees of different order

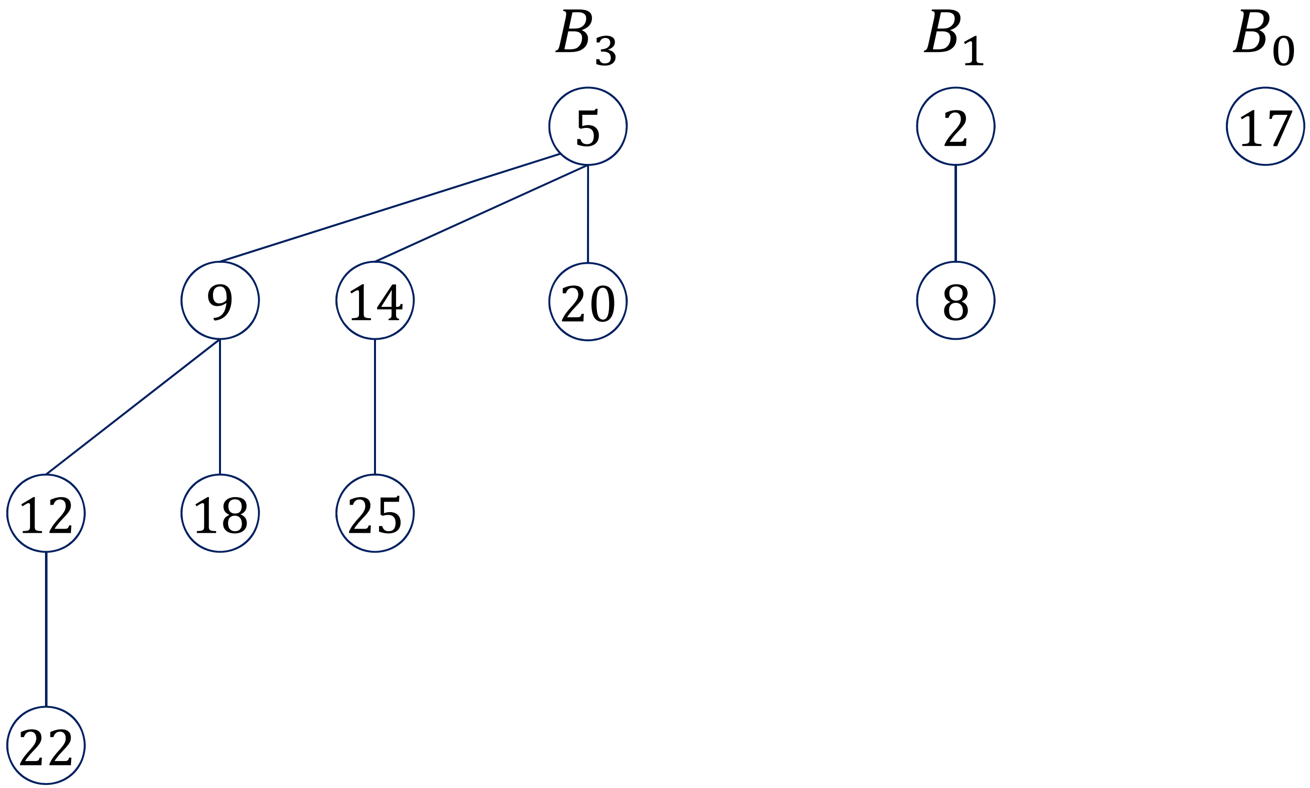
  n nodes: there is a binomial tree B_i of order i iff
  bit i of base-2 representation of n is 1.

- **Min-Heap Property:**

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

Example

- 11 keys: $\{2, 5, 8, 9, 12, 14, 17, 18, 20, 22, 25\}$
- Binary representation of n: $(11)_2 = 1011$
  → trees B_0, B_1, and B_3 present

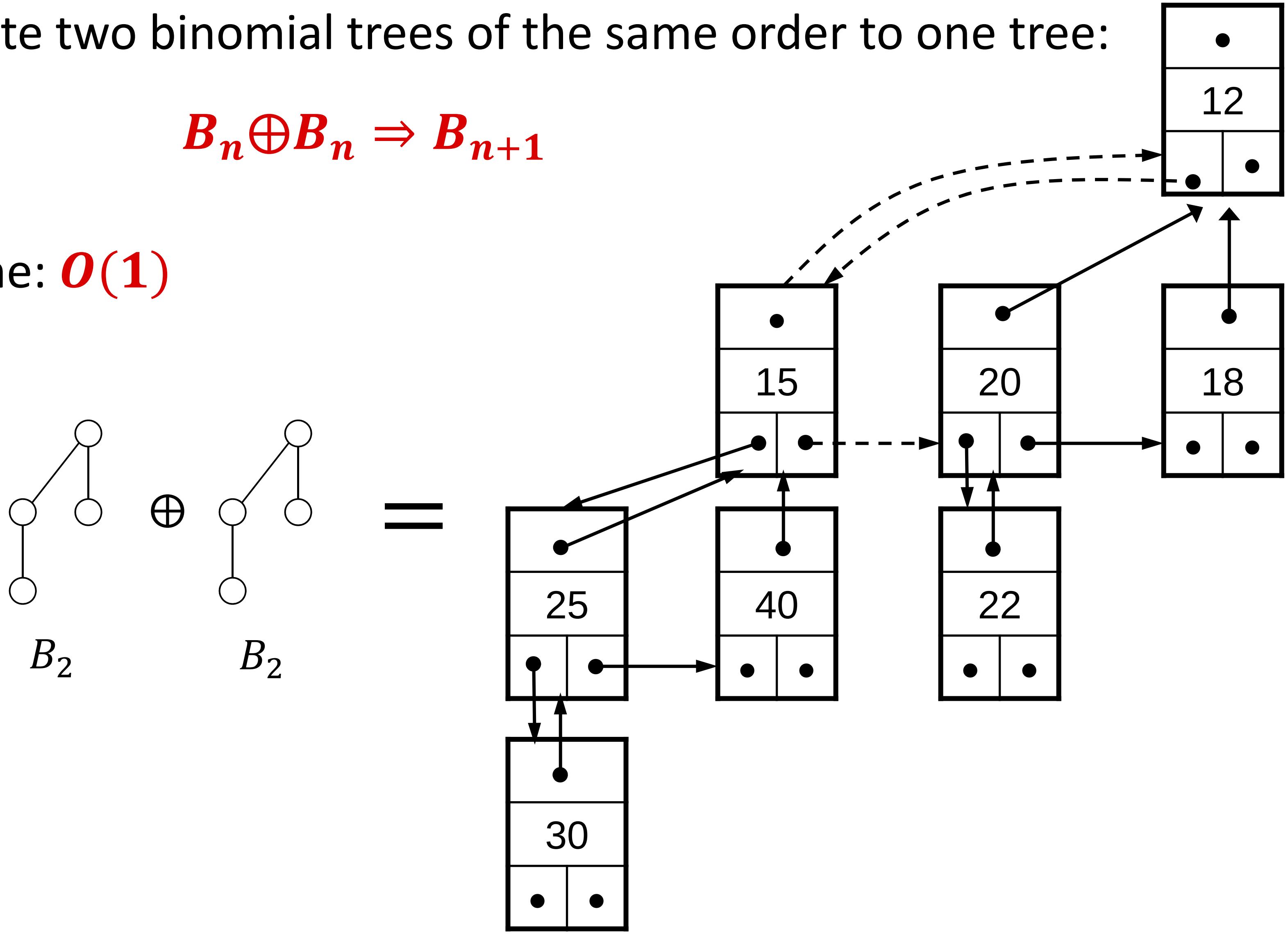


Link Operation

- Unite two binomial trees of the same order to one tree:

$$B_n \oplus B_n \Rightarrow B_{n+1}$$

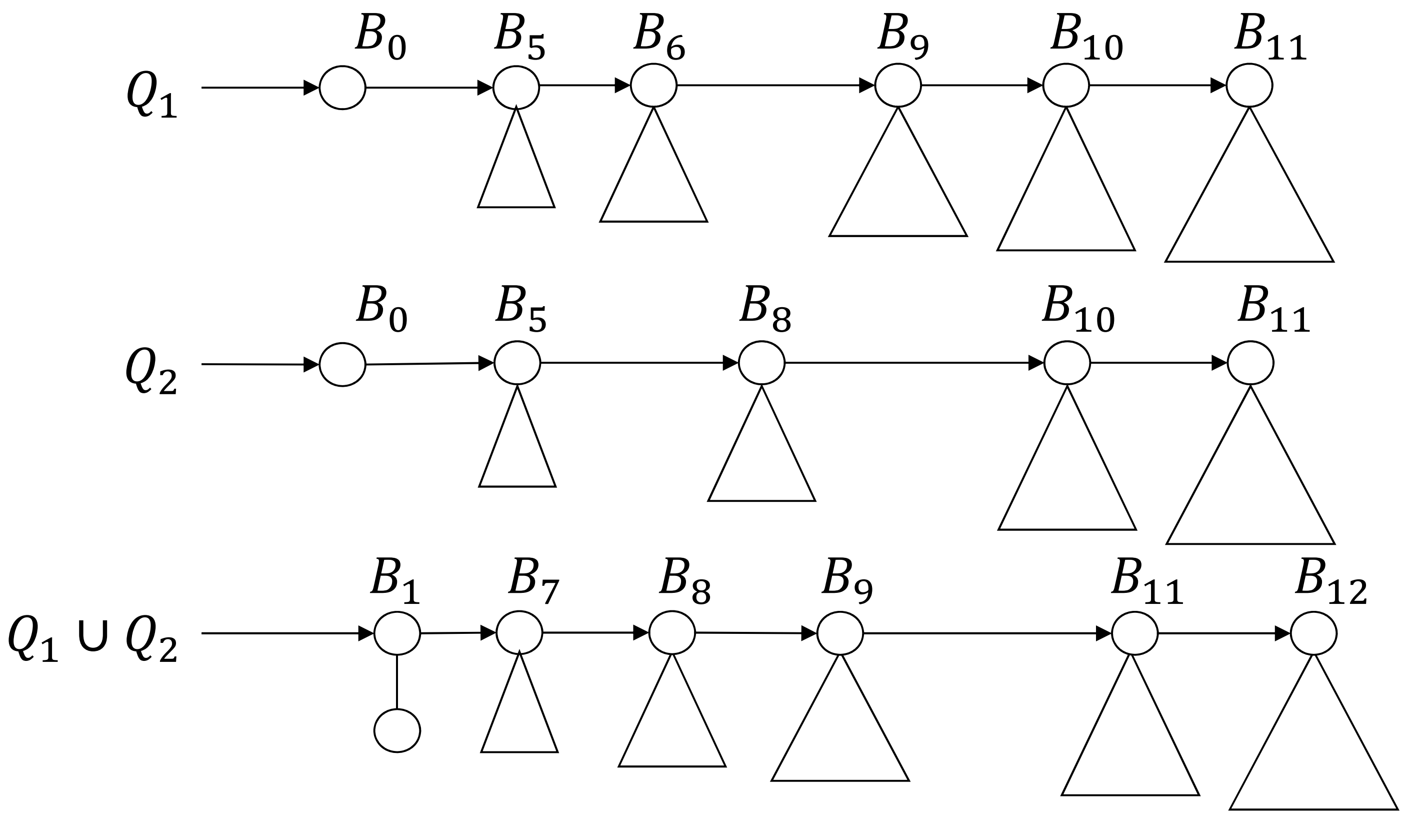
- Time: $O(1)$



Merge Operation

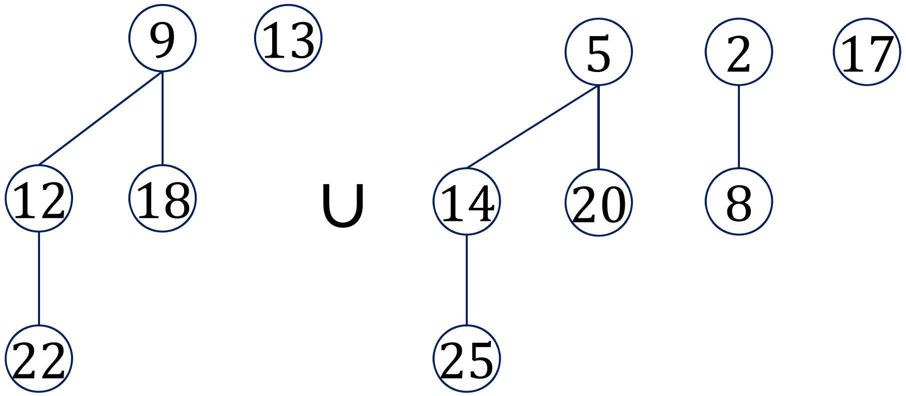
Merging two binomial heaps:

- **For $\boldsymbol{i = 0, 1, \dots, \log n}$:**
  If there are 2 or 3 binomial trees B_i: apply link operation to merge 2 trees into one binomial tree B_{i+1}



Time: $\boldsymbol{O(\log n)}$

Example



Operations

Initialize: create empty list of trees

Get minimum of queue: time $O(1)$ (if we maintain a pointer)

Decrease-key at node v:

- Set *key* of node v to new key
- Swap with parent until min-heap property is restored
- Time: $O(\log n)$

Insert *key* x into queue Q:

1. Create queue Q' of size 1 containing only x
2. Merge Q and Q'

- Time for insert: $O(\log n)$

Operations

Delete-Min Operation:

- Smallest key is at the root of some tree
- Removing the root of a binomial tree:

Operations

Delete-Min Operation:

1. Find tree B_i with minimum root r

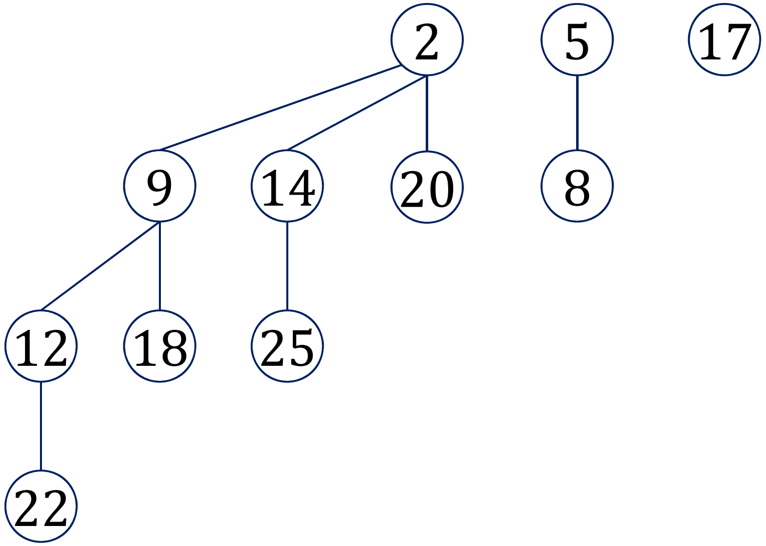
2. Remove B_i from queue Q → queue Q'

3. Children of r form new queue Q''

4. Merge queues Q' and Q''

- **Overall time: $\boldsymbol{O(\log n)}$**

Delete-Min Example



Complexities Binomial Heap

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

Can We Do Better?

- Binomial heap:
  insert, delete-min, and decrease-key cost $O(\log n)$

- One of the operations insert or delete-min must cost $\Omega(\log n)$:
  - Heap-Sort:
    Insert n elements into heap, then take out the minimum n times
  - (Comparison-based) sorting costs at least $\Omega(n \log n)$.

- But maybe we can improve decrease-key and one of the other two operations?

- Structure of binomial heap is not flexible:
  - Simplifies analysis, allows to get strong worst-case bounds
  - **But**, operations almost inherently need at least logarithmic time

Fibonacci Heaps

Lacy-merge variant of binomial heaps:

- Do not merge trees as long as possible…

Structure:

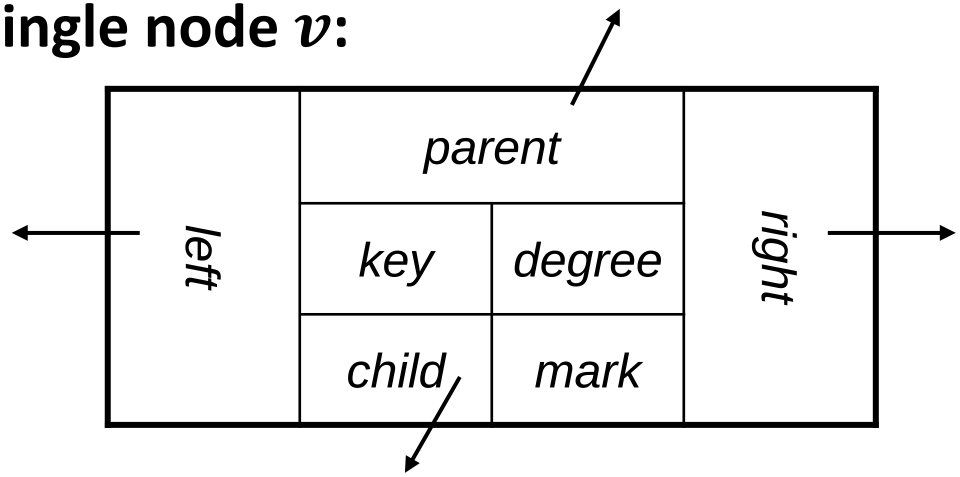
A Fibonacci heap H consists of a collection of trees satisfying the min-heap property.

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

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

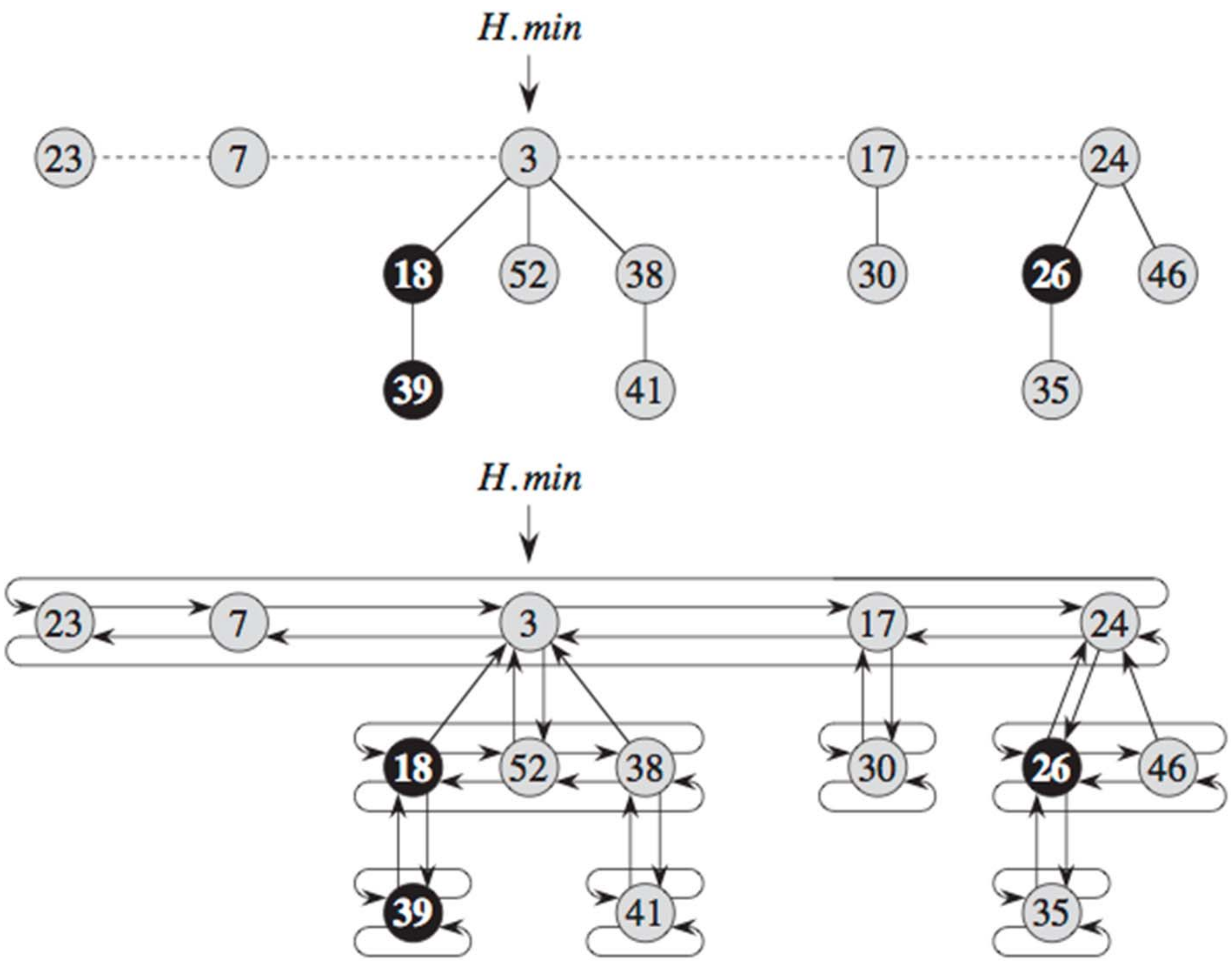


Figure: Cormen et al., Introduction to Algorithms

Simple (Lazy) Operations

Initialize-Heap H:

- $H.rootlist \coloneqq H.min \coloneqq null$

Merge heaps H and H':

- concatenate root lists
- update $H.min$

Insert element e into H:

- create new one-node tree containing e → H'
- 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:

- $rank(v)$: degree of v

Tree T:

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

Heap H:

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

Assumption (n: number of nodes in H):

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

– for a known function $D(n)$

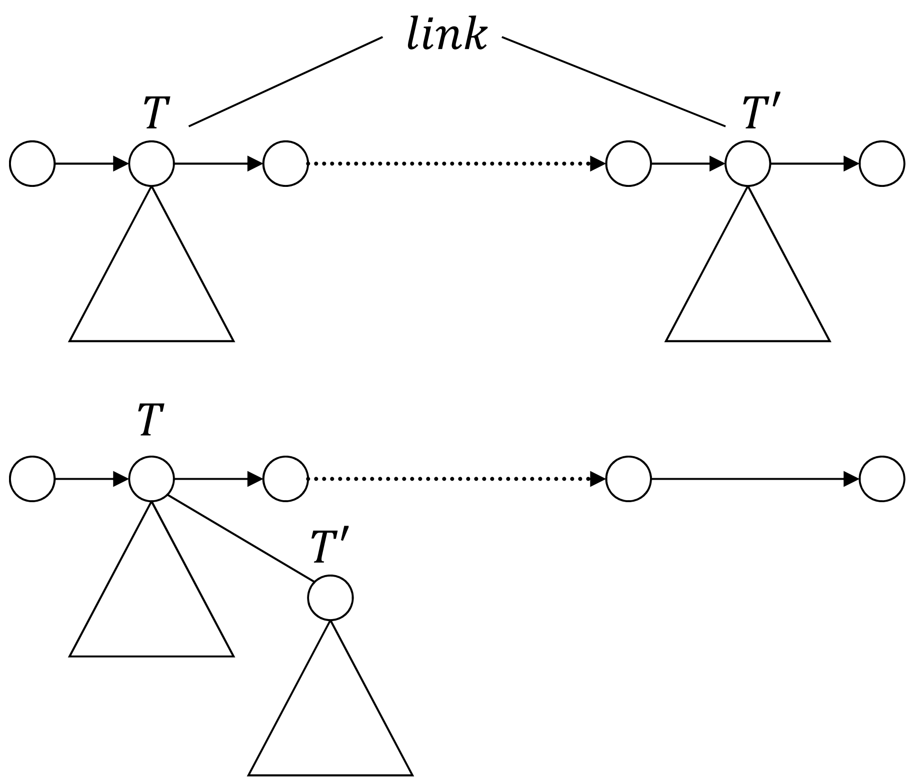
Merging Two Trees

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

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

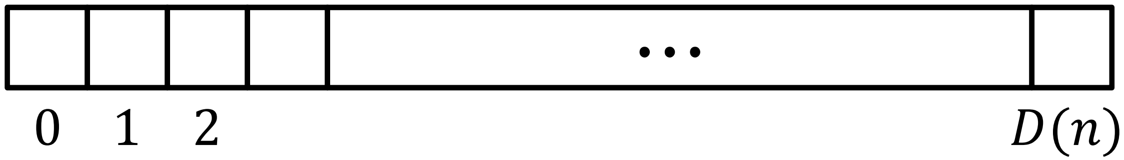
Operation $\boldsymbol{link(T, T')}$:

- Removes tree T' from root list and adds T' to child list of T
- $rank(T) \coloneqq rank(T) + 1$
- $T'.mark \coloneqq \textbf{false}$



Consolidation of Root List

Array A pointing to find roots with the same rank:

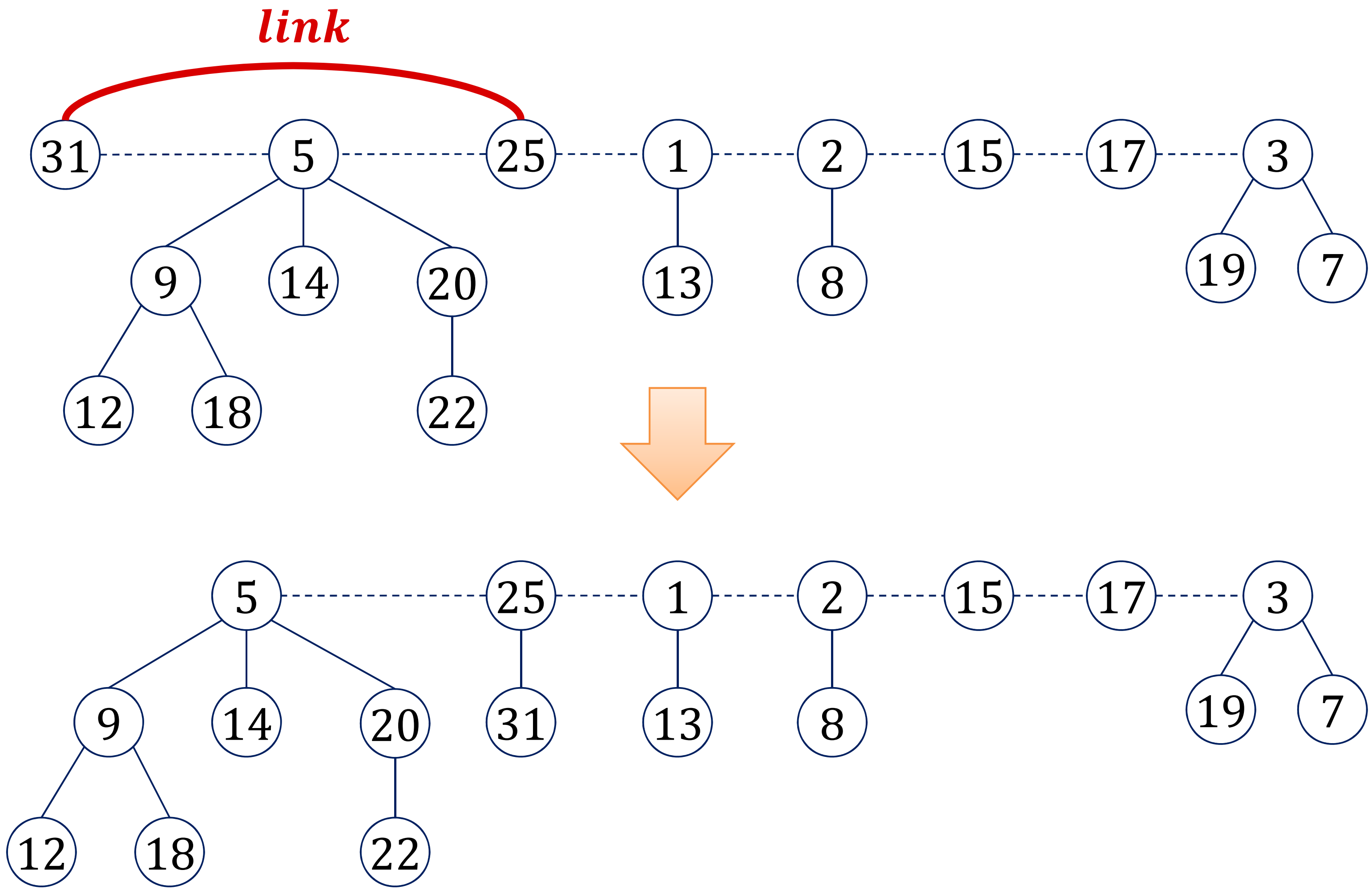


Consolidate:

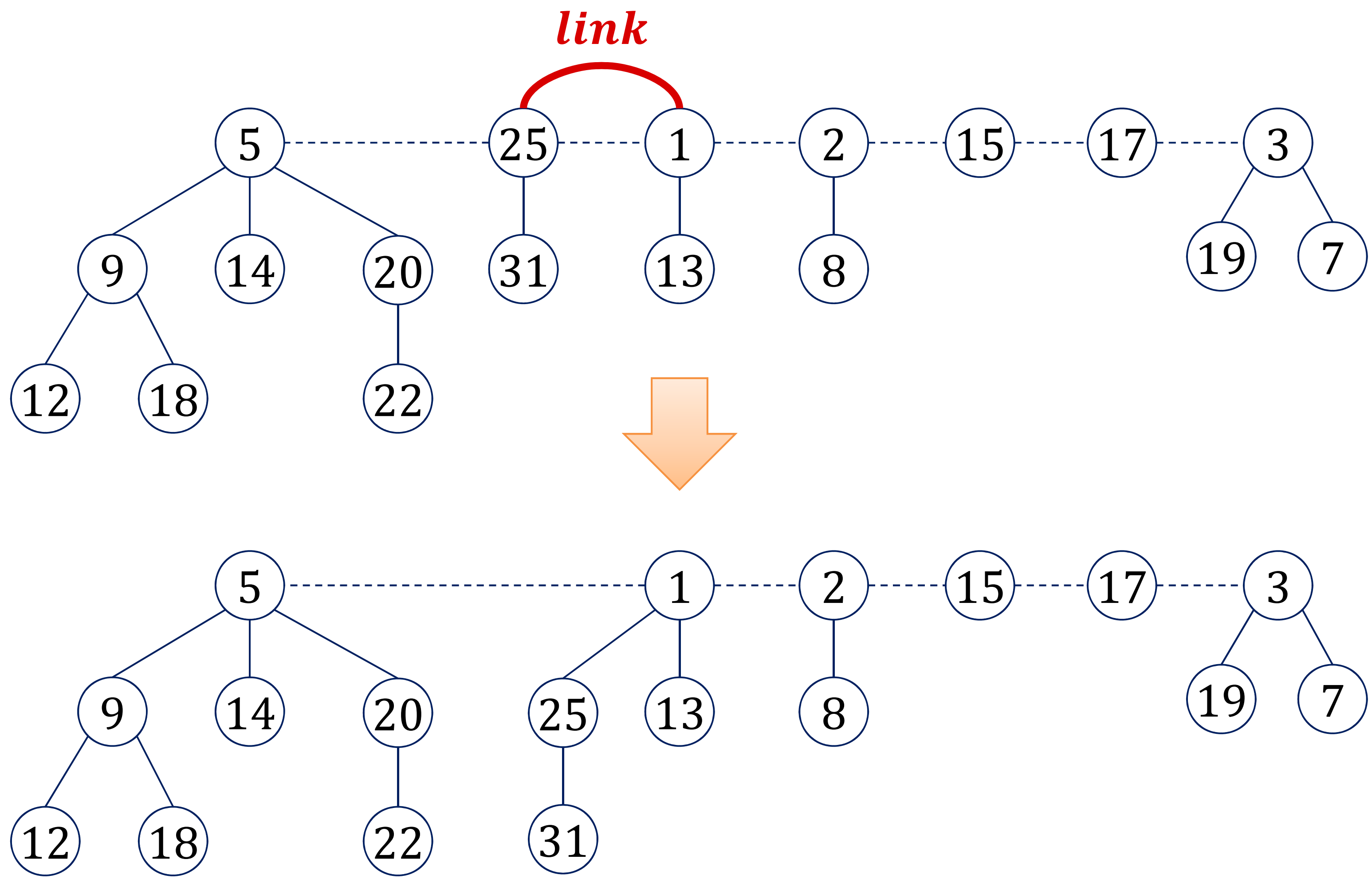
for $i := 0$ **to** $D(n)$ **do** $A[i] :=$ null;
while $H.rootlist \neq$ null **do**
    $T :=$ "delete and return first element of $H.rootlist$"
    while $A[rank(T)] \neq$ null **do**
        $T' := A[rank(T)]$;
        $A[rank(T)] := null$;
        $T := link(T, T')$
    $A[rank(T)] := T$
Create new $H.rootlist$ and $H.min$

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

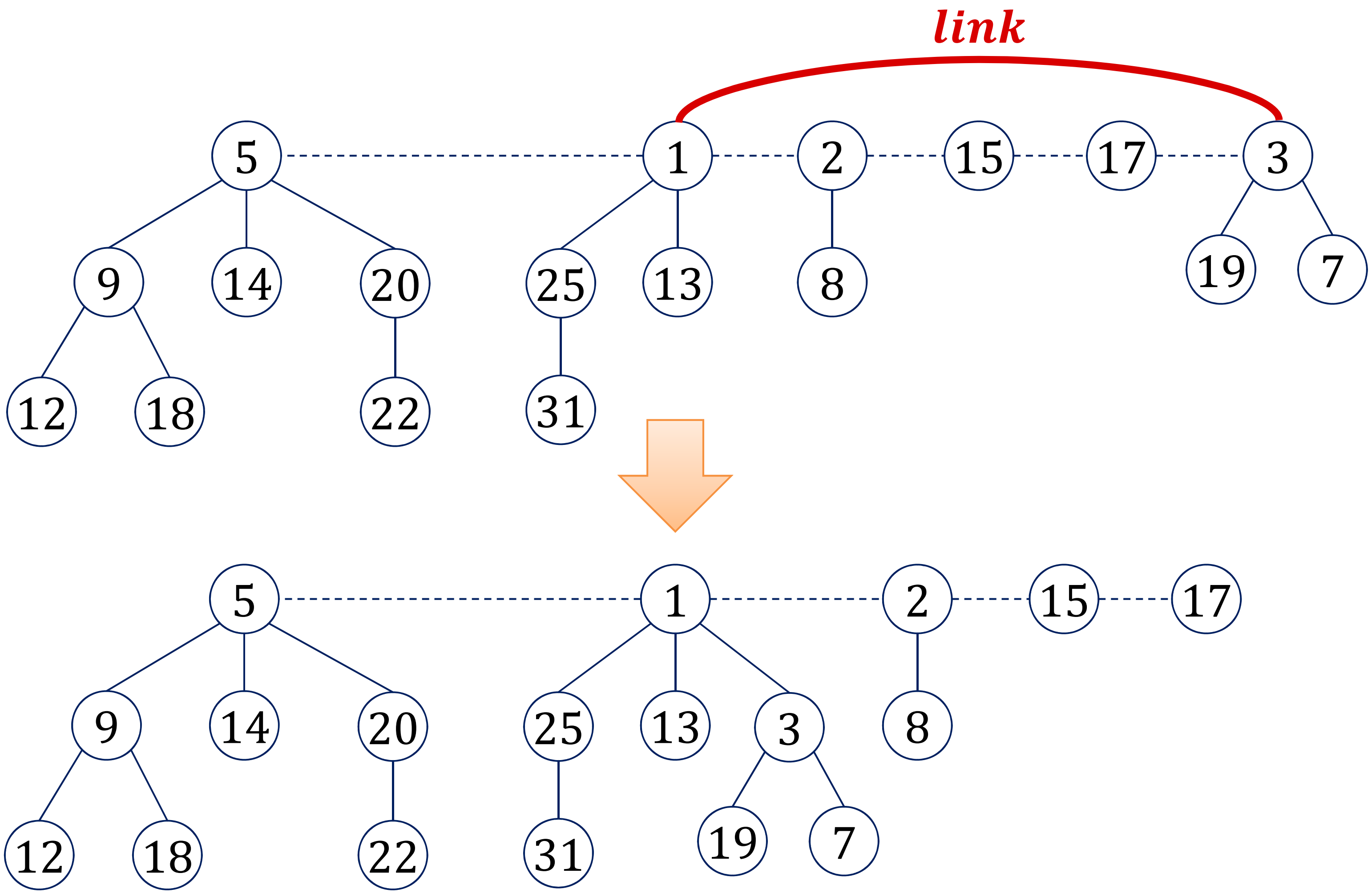
Consolidate Example



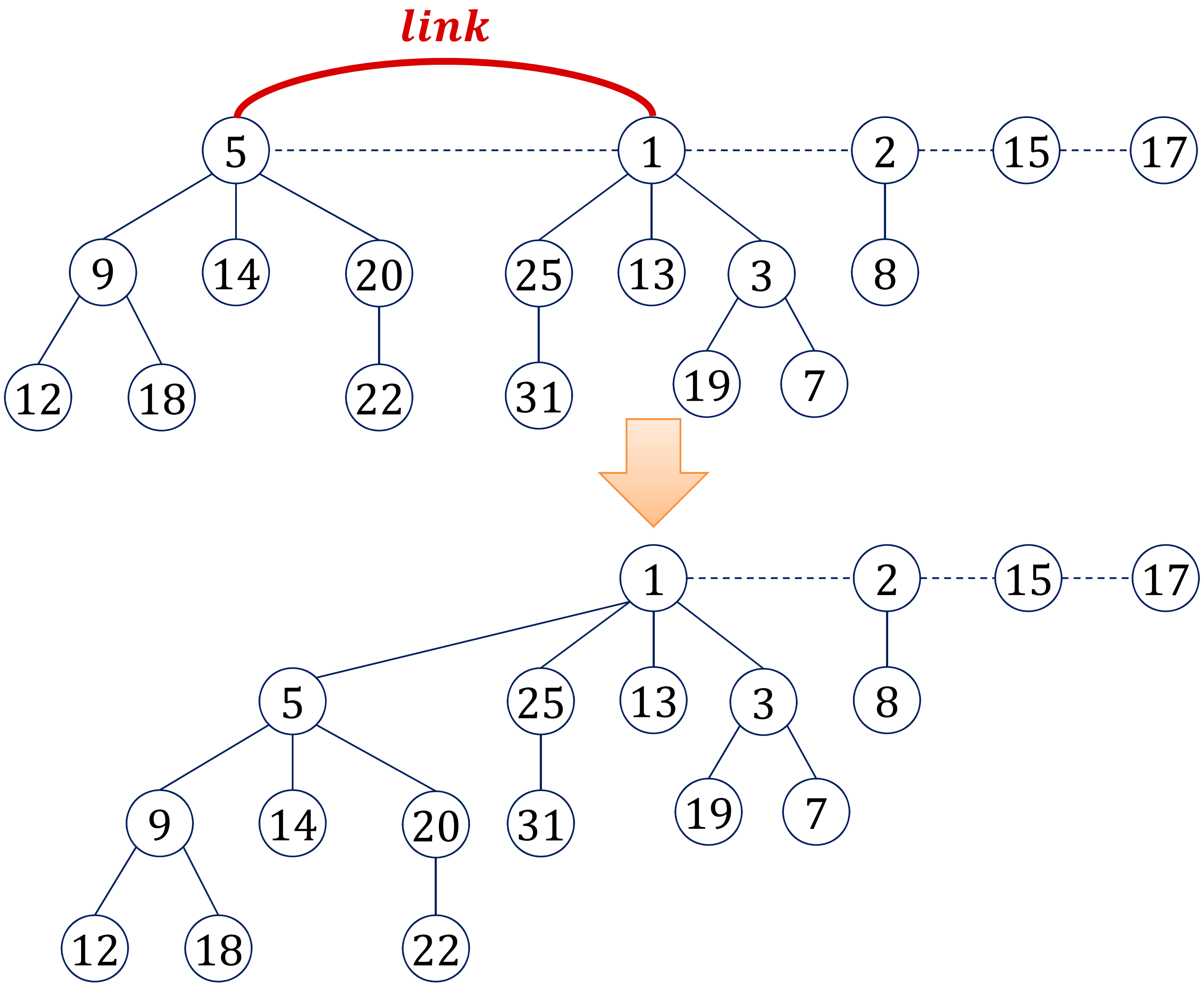
Consolidate Example



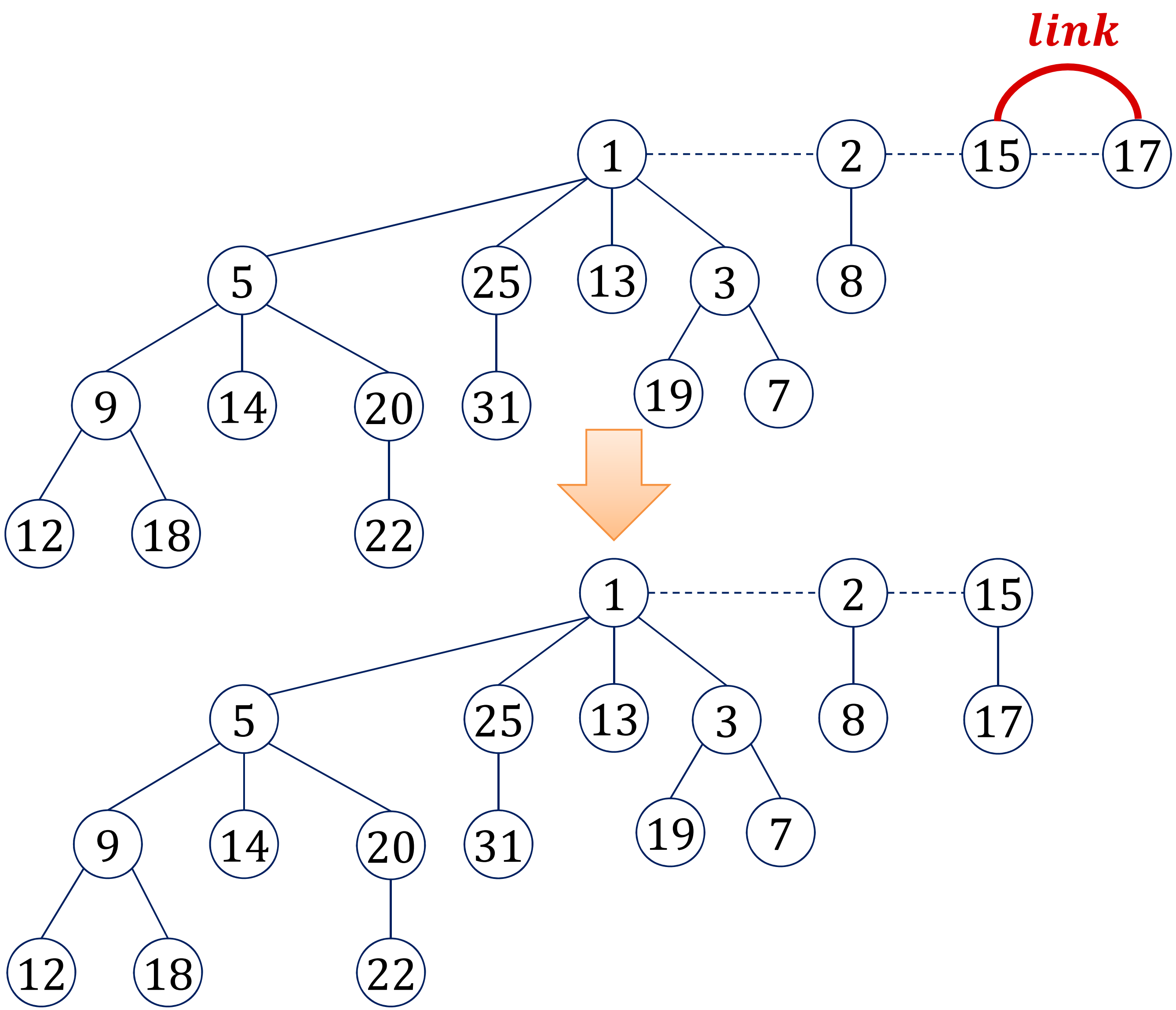
Consolidate Example



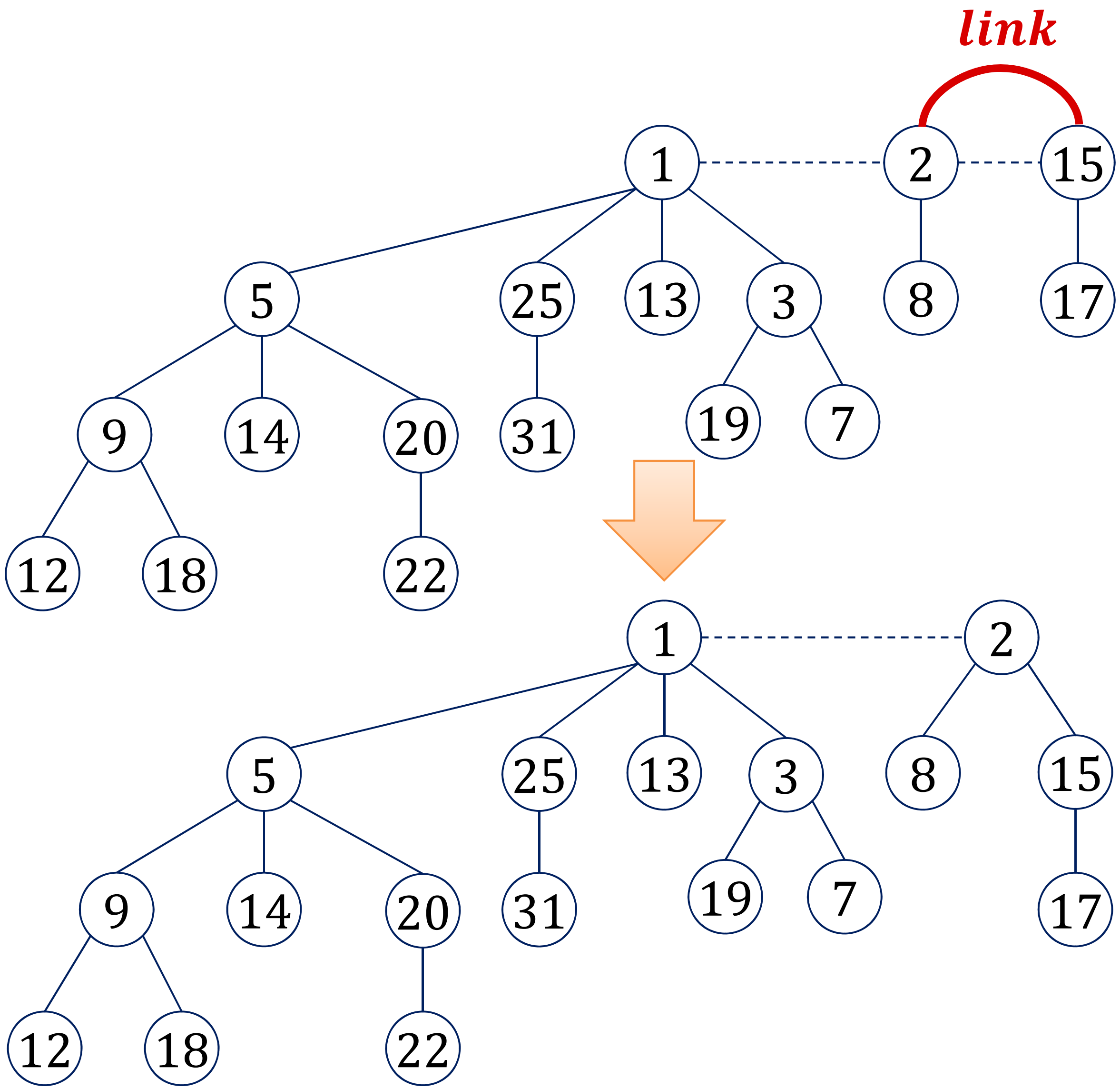
Consolidate Example



Consolidate Example



Consolidate Example



Operation Decrease-Key

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

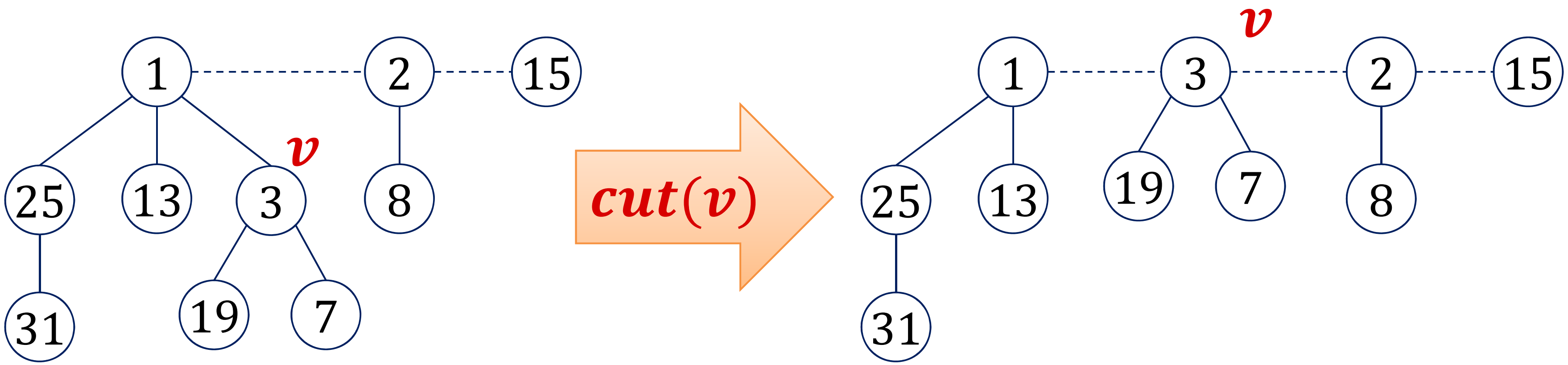
1. **if** $x \geq v.key$ **then return**;
2. $v.key \coloneqq x$; update $H.min$;
3. **if** $v \in H.rootlist \lor x \geq v.parent.key$ **then return**
4. **repeat**
5. $\quad parent \coloneqq v.parent$;
6. $\quad \boldsymbol{H.cut(v)}$;
7. $\quad v \coloneqq parent$;
8. **until** $\neg(\boldsymbol{v.mark})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

Operation Cut(v)

Operation $H.cut(v)$:

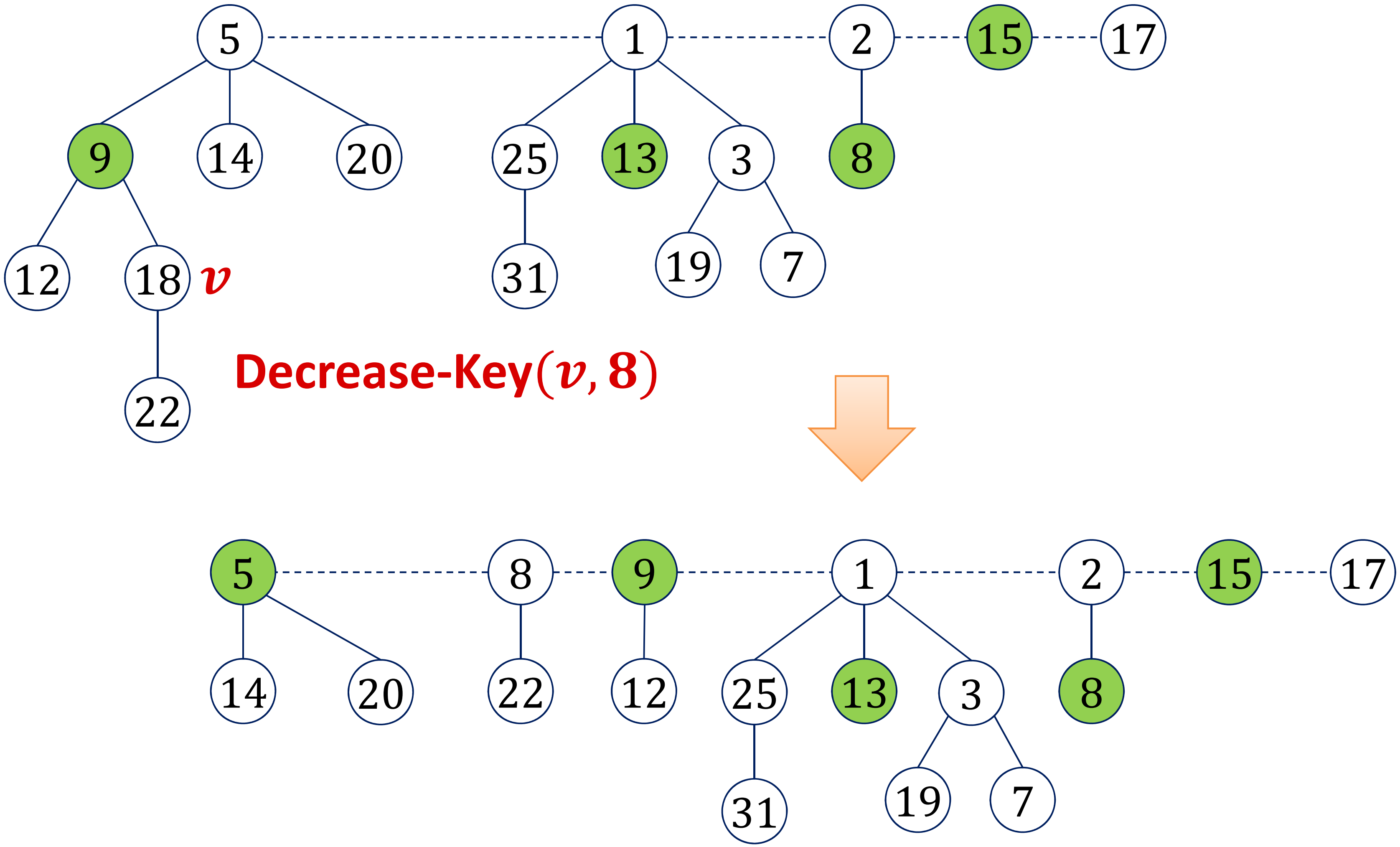
- Cuts v's sub-tree from its parent and adds v to rootlist

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



Decrease-Key Example

- Green nodes are marked



Fibonacci Heap Marks

History of a node v:

v is being linked to a node ⇨ $\boldsymbol{v.mark} \coloneqq \textbf{false}$

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

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

- 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.

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 → requires **amortized analysis**