



# **Chapter 4**

# **Data Structures**

**Algorithm Theory**  
**WS 2014/15**

**Fabian Kuhn**

# Priority Queue / Heap

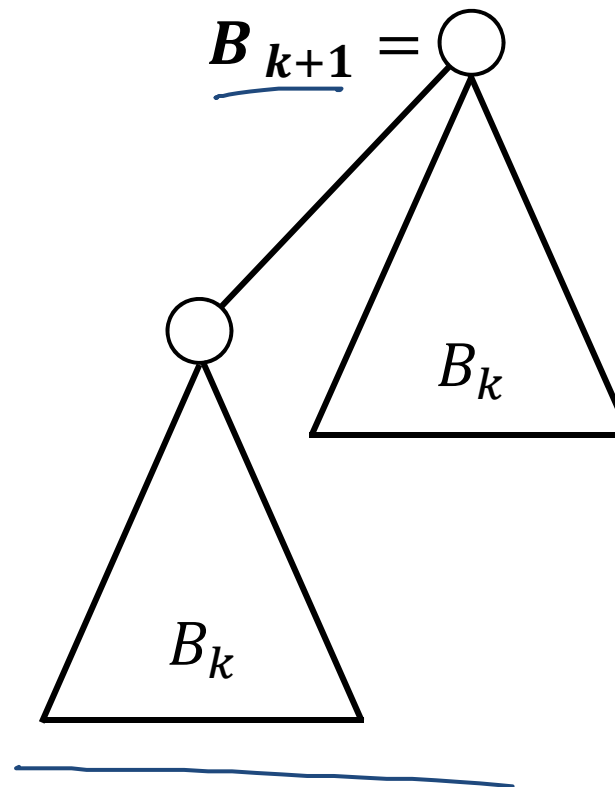
---

- 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** $(\underline{entry}, \underline{newkey})$ : decreases  $key$  of  $entry$  to  $newkey$ 
  - $entry$ : pointer to entry in data structure
- **Merge**: merges two heaps into one

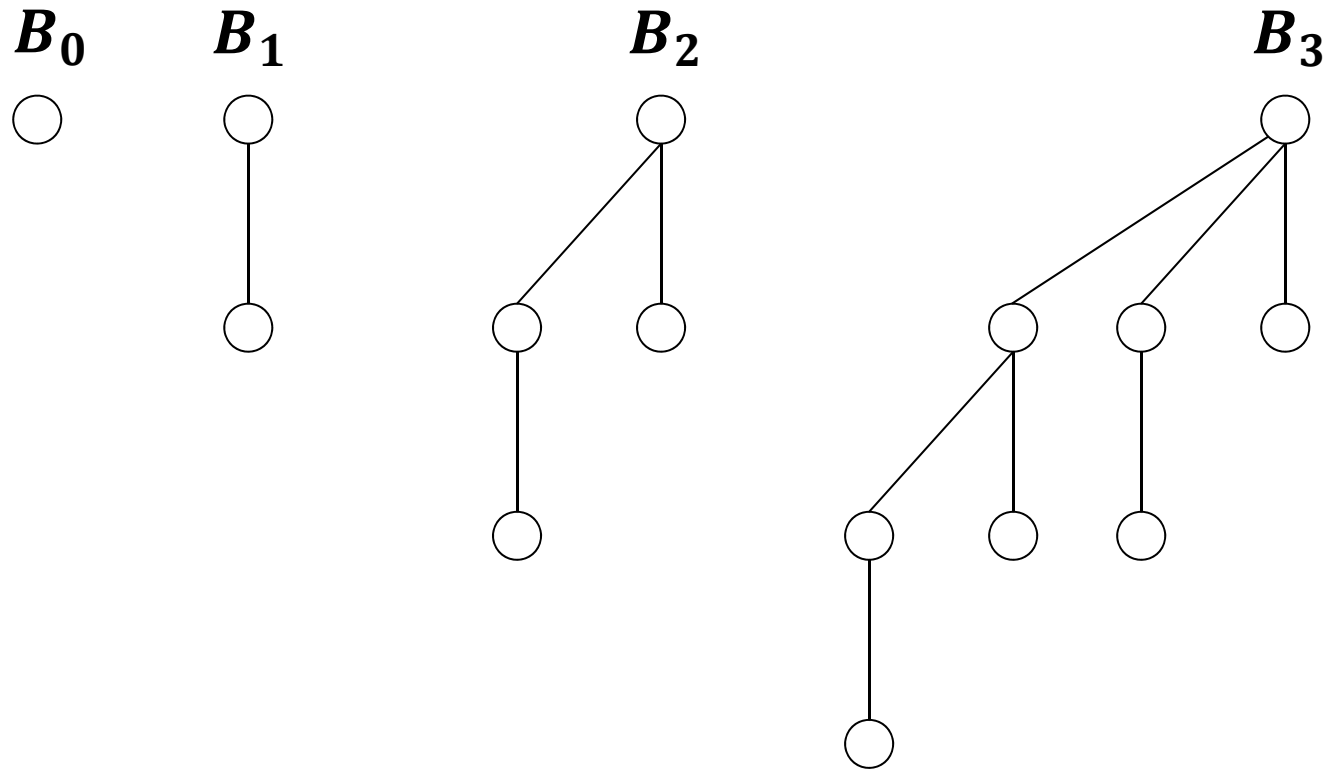
# Definition: Binomial Tree

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

$$\underline{B_0} = \bigcirc$$



# 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

$$|B_i| = 2^i$$



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

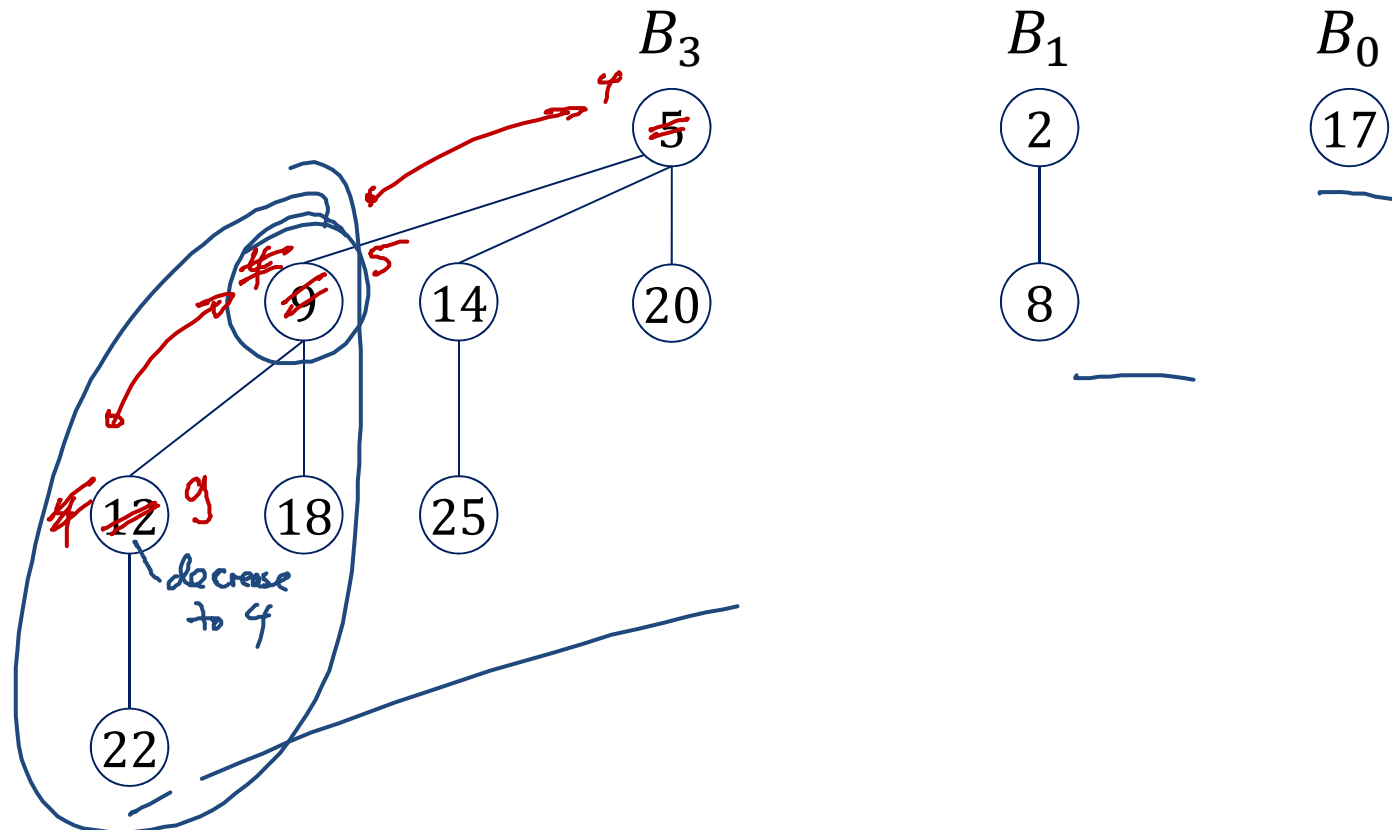
$n$  nodes: there is a binomial tree  $B_i$  of order  $i$  iff  
bit  $i$  of base-2 representation of  $n$  is 1.

- 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  $n$ :  $(11)_2 = 1011$   
 $\rightarrow$  trees  $B_0, B_1,$  and  $B_3$  present

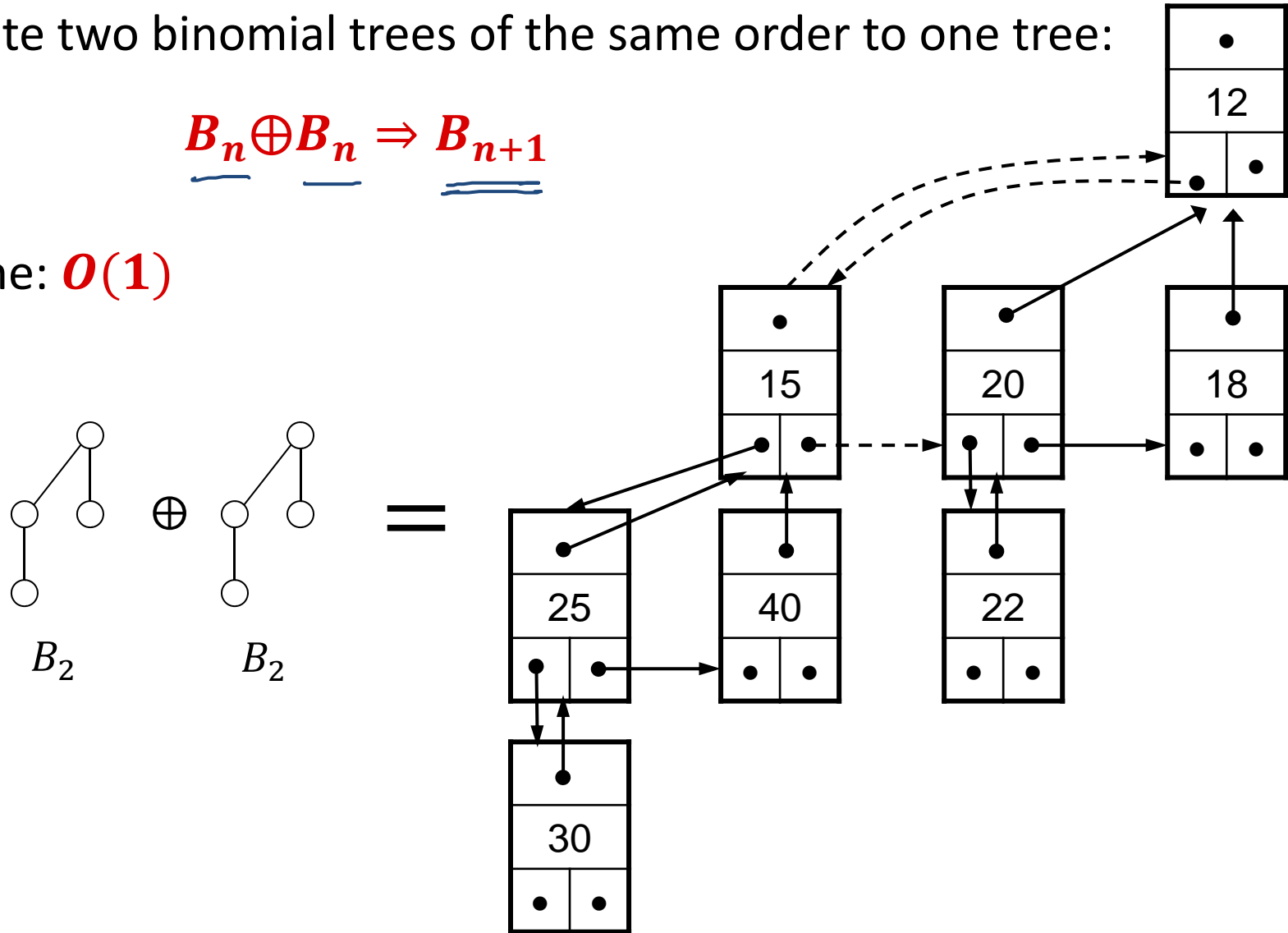


# Link Operation

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

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

- Time:  $O(1)$



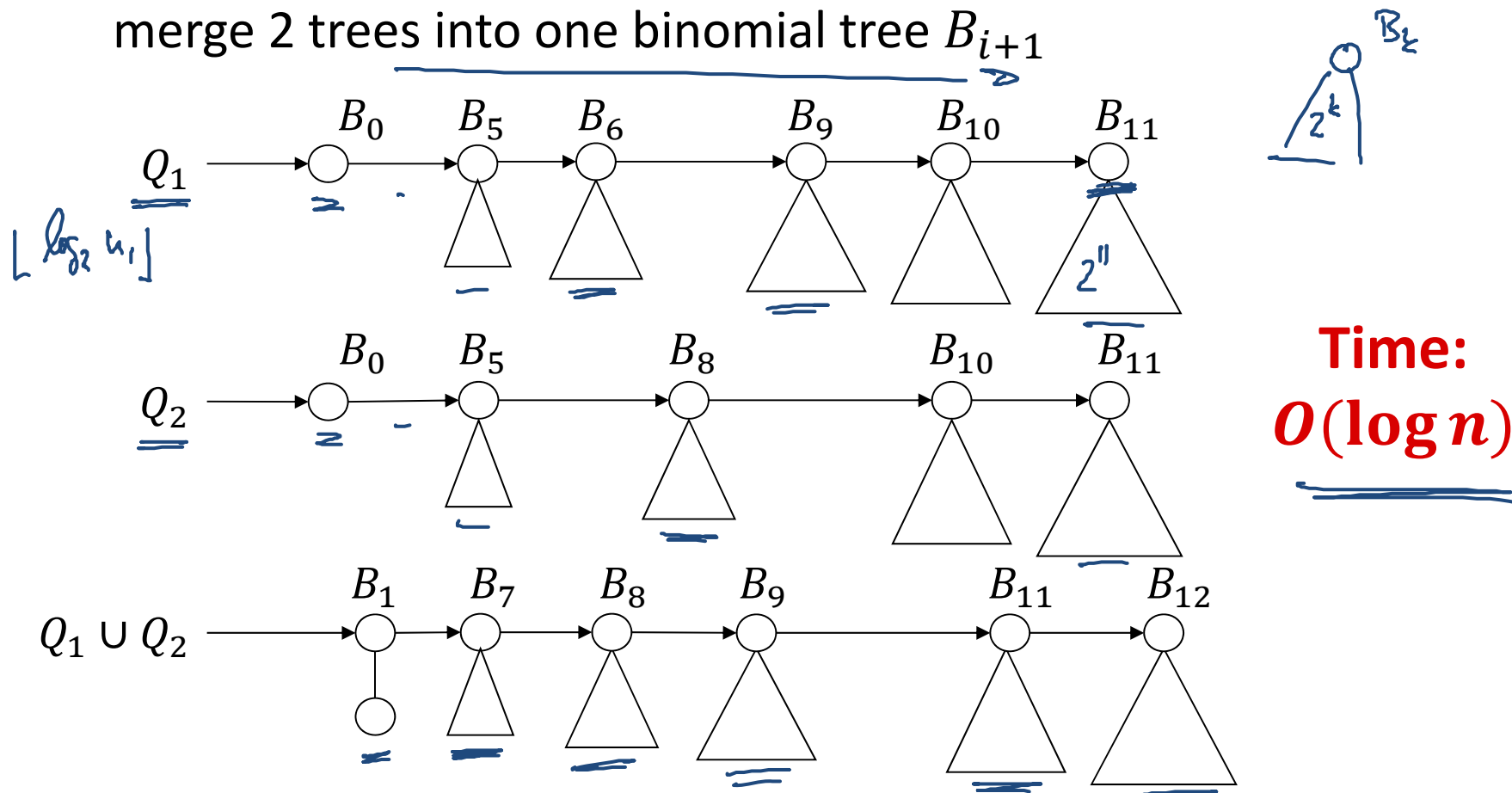


# Merge Operation

Merging two binomial heaps:

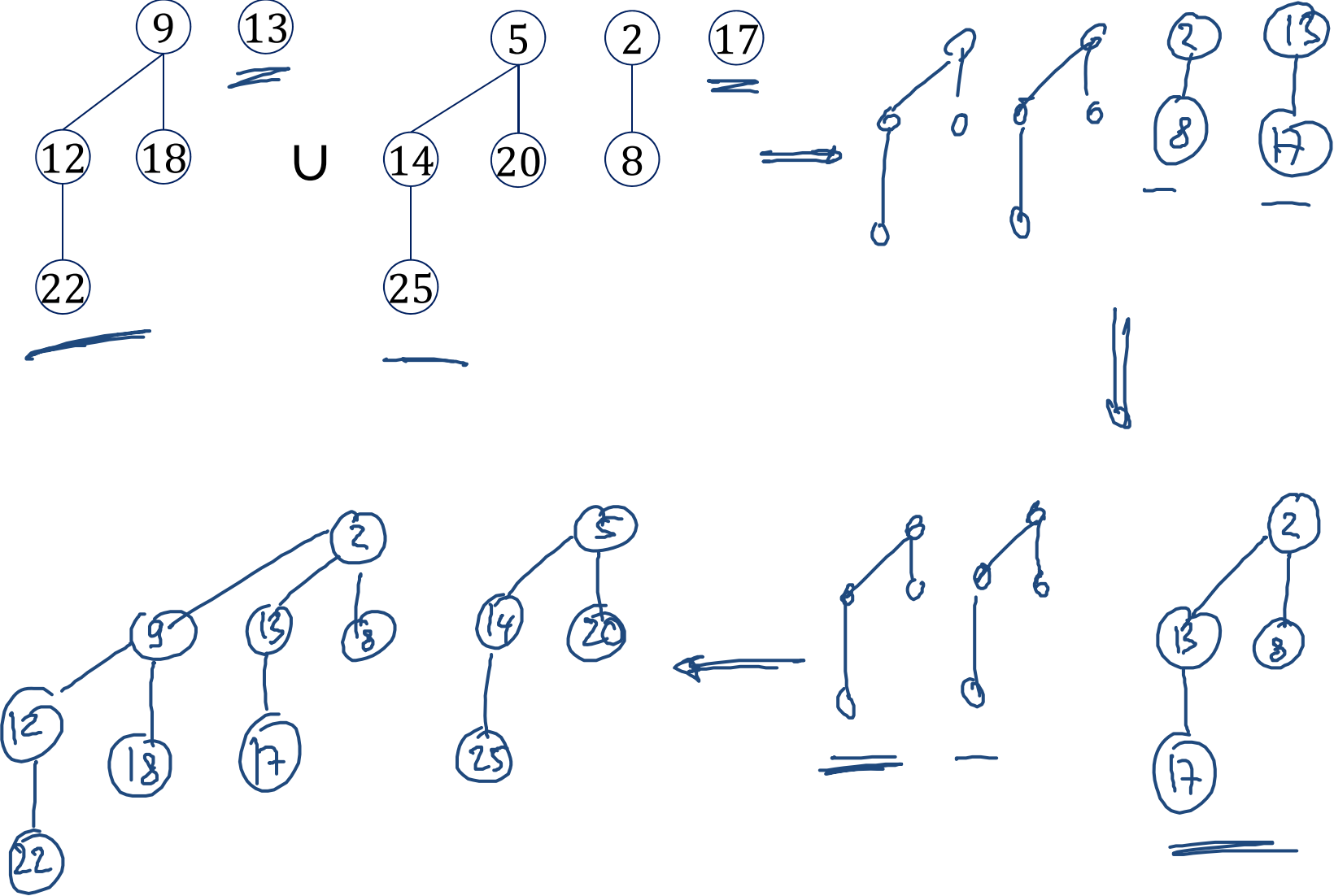
- **For  $i = 0, 1, \dots, \log n$ :**

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:**  
 $O(\log n)$

# 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:  $O(\log n)$**   $\leftarrow$  depth of  $B_k$  is  $k$

**Insert** key  $x$  into queue  $Q$ :

1. Create queue  $Q'$  of size 1 containing only  $x$   $O(1)$  time
2. Merge  $Q$  and  $Q'$   $O(\log n)$  time

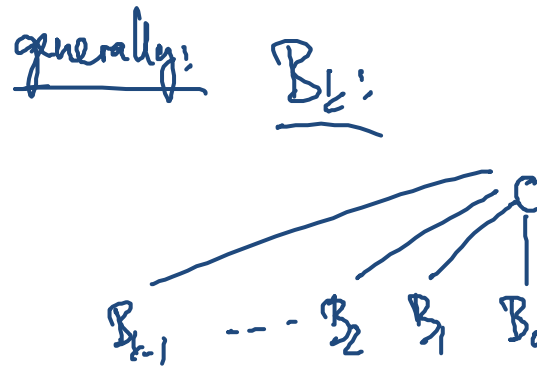
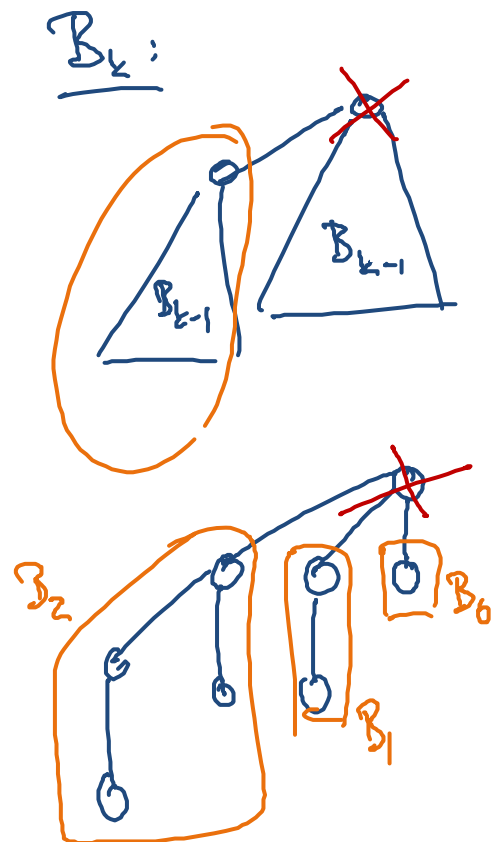
- **Time for insert:  $O(\log n)$**



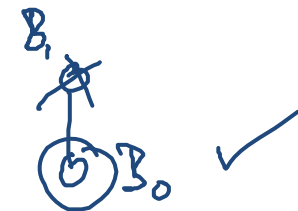
# Operations

## Delete-Min Operation:

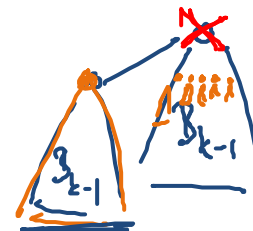
- Smallest key is at the root of some tree
- Removing the root of a binomial tree:



Proof!  
by induction on  $k$ :  $k=1$ :



$k-1 \Rightarrow k$ :

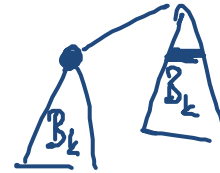


same subtrees as in  $B_{k-1}$   
and one additional  $B_{k-1}$   
off.: subtr. of root of  $B_{k-1}$   
are  $B_0, B_1, \dots, B_{k-2}$

# Operations

## Delete-Min Operation:

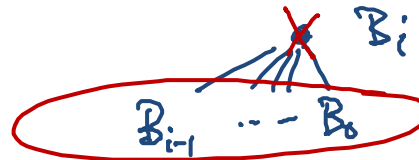
1. Find tree  $B_i$  with minimum root  $r$   
with min-pointers:  $O(1)$  time



2. Remove  $B_i$  from queue  $Q \rightarrow$  queue  $Q'$   
 $O(1)$  time



3. Children of  $r$  form new queue  $Q''$   
 $O(\log n)$  time



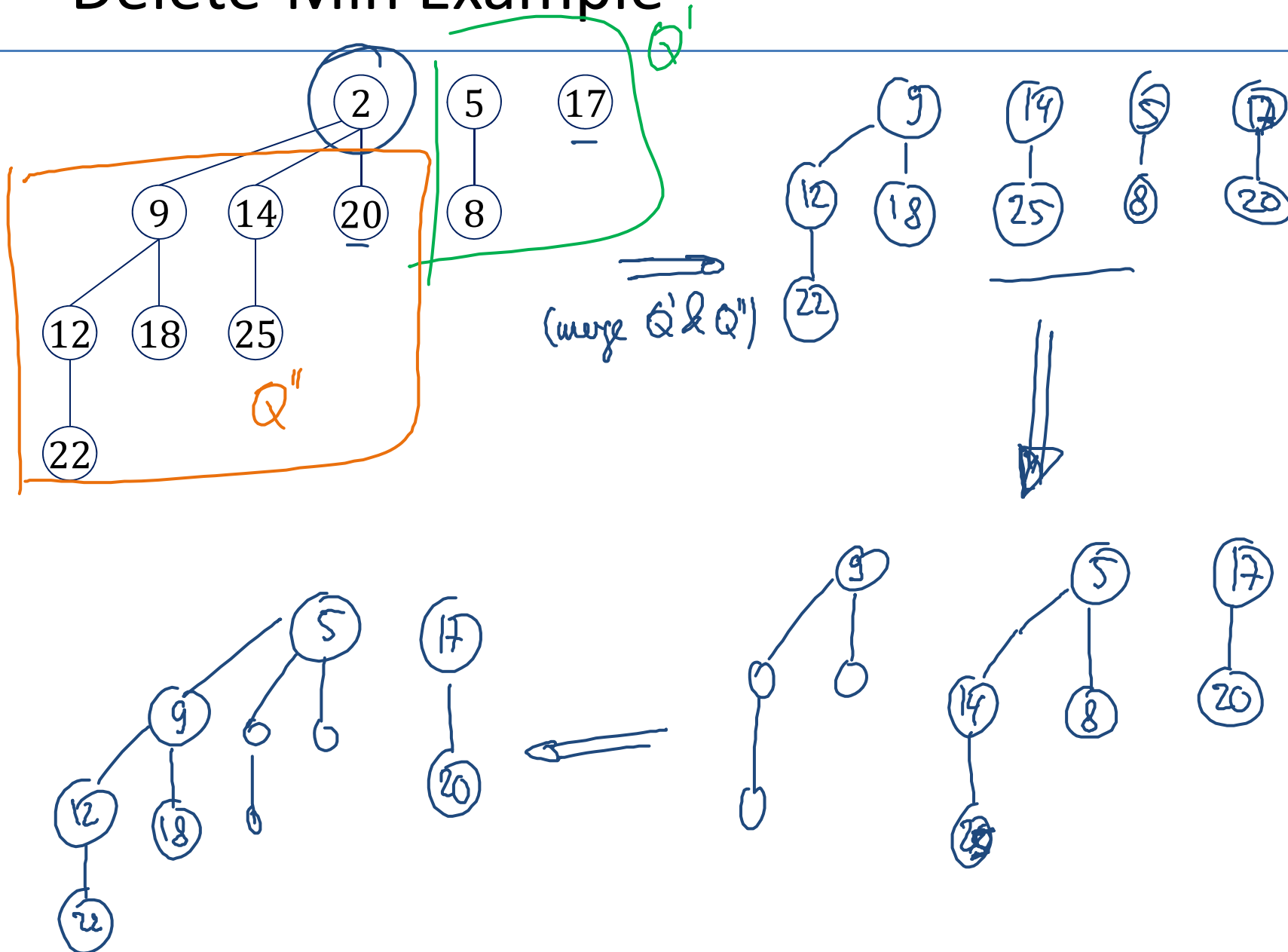
4. Merge queues  $Q'$  and  $Q''$   
 $O(\log n)$  time

root list:  
go from small to large trees  
(for merging)

child list:  
sorted from large to small

• **Overall time:  $O(\log n)$**

# Delete-Min Example



# Complexities Binomial Heap

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

except for merge op, not better than array implementation

# 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:  $2^k$  elements  $\rightarrow$  1 tree  
 $2^k - 1$  elements  $\rightarrow$   $k-1$  trees
  - 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:**

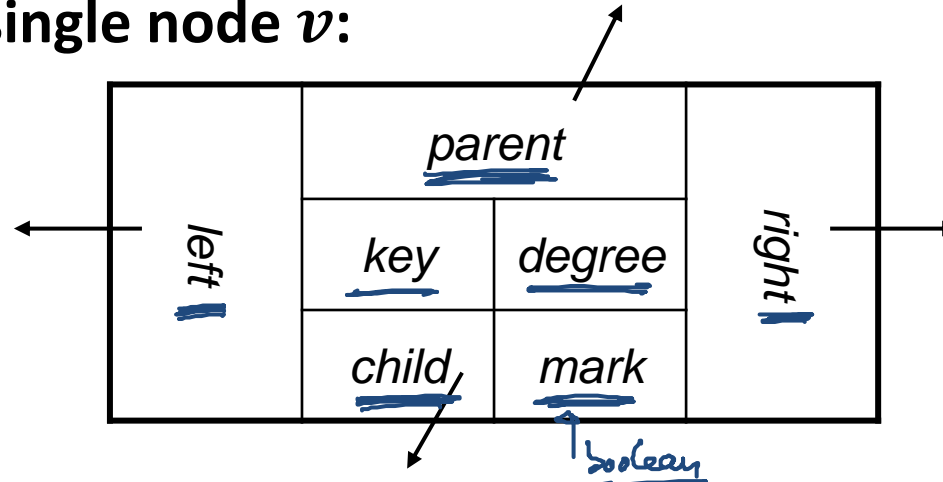
A Fibonacci heap  $H$  consists of a collection of trees satisfying the min-heap property.

## **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

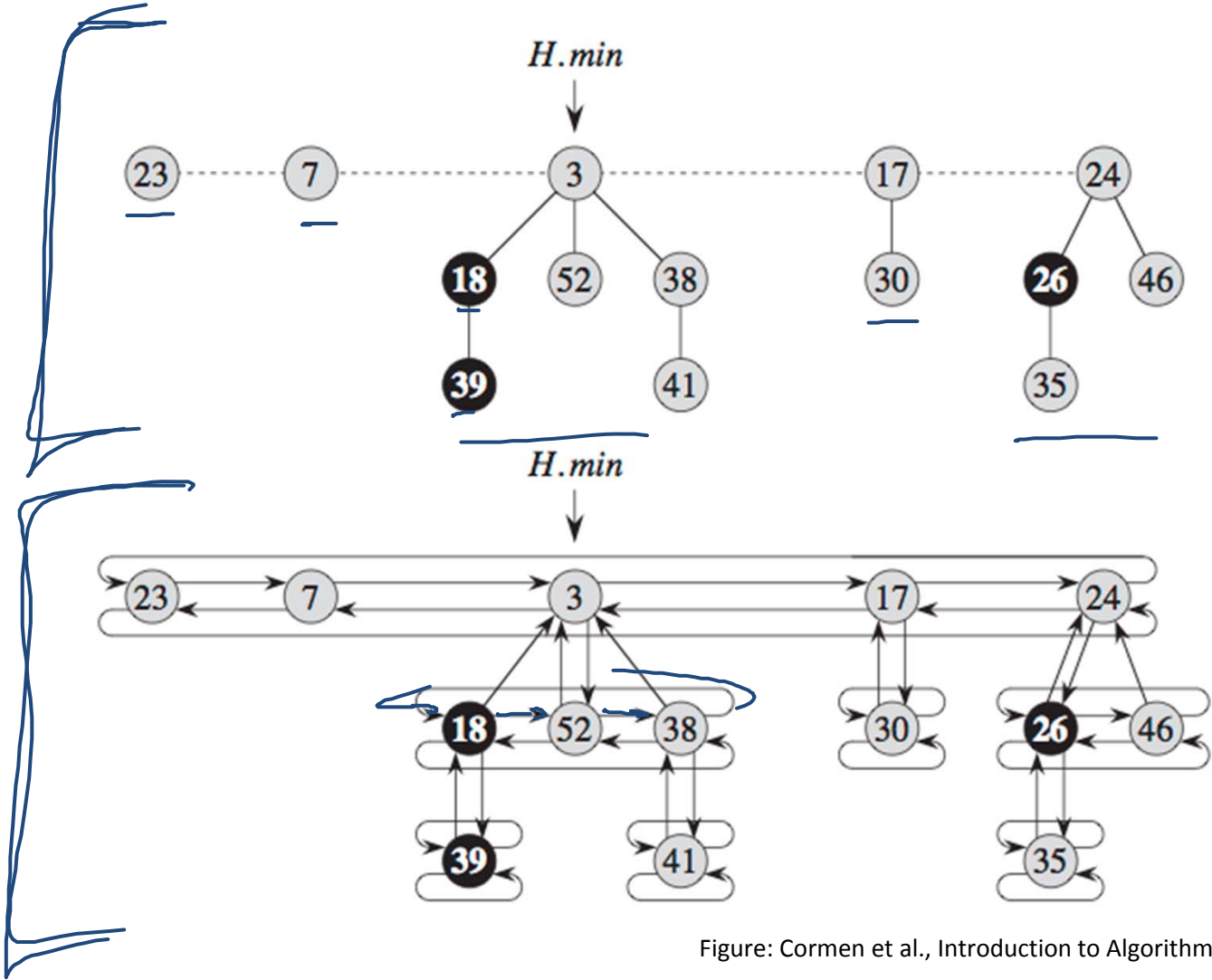


Figure: Cormen et al., Introduction to Algorithms

# Simple (Lazy) Operations

## Initialize-Heap $H$ :

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

## Merge heaps $H$ and $H'$ :

- concatenate root lists  $O(1)$  time
- update  $H.min$

## Insert element $e$ into $H$ :

- create new one-node tree containing  $e \rightarrow H'$
- merge heaps  $H$  and  $H'$

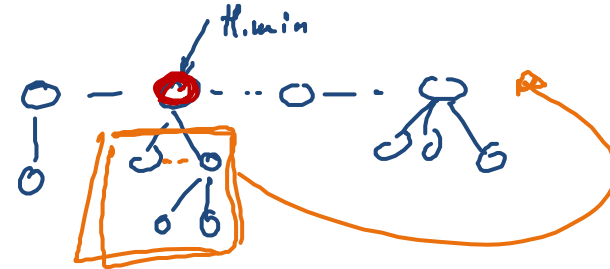
## Get minimum element of $H$ :

- 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.  **$H.Consolidate();$**



// 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

degree of root of tree

6. **return  $m$**

# Rank and Maximum Degree

## Ranks of nodes, trees, heap:

### Node $v$ :

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

### 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 $H$ ):

$$\underline{rank(H) \leq D(n)}$$

– for a known function  $D(n)$

*max degree of any node*  
*#children*

# Merging Two Trees

*min-heap prop.*

**Given:** Heap-ordered trees  $T, T'$  with  $\underline{rank(T) = rank(T')}$

- Assume: min-key of  $T$   $\leq$  min-key of  $T'$

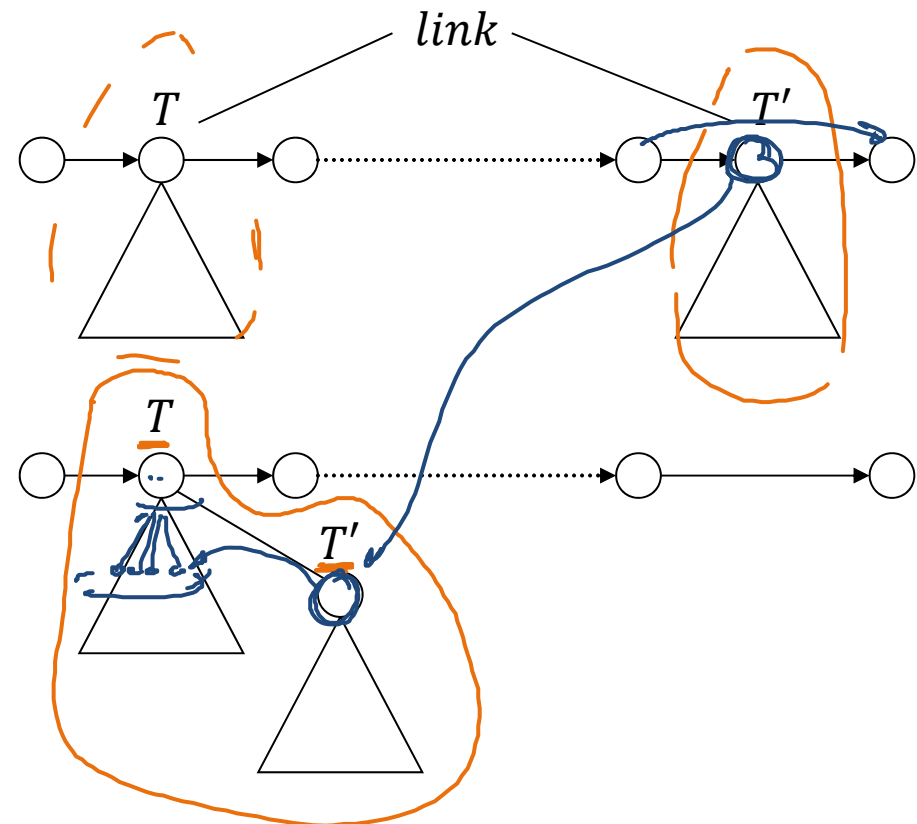
**Operation  $link(T, T')$ :**

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

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

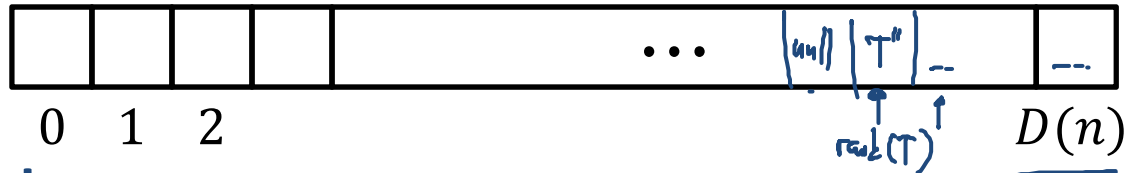
- **$T'.mark := false$**

*mark of root of  $T'$*



# 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  $H.min$

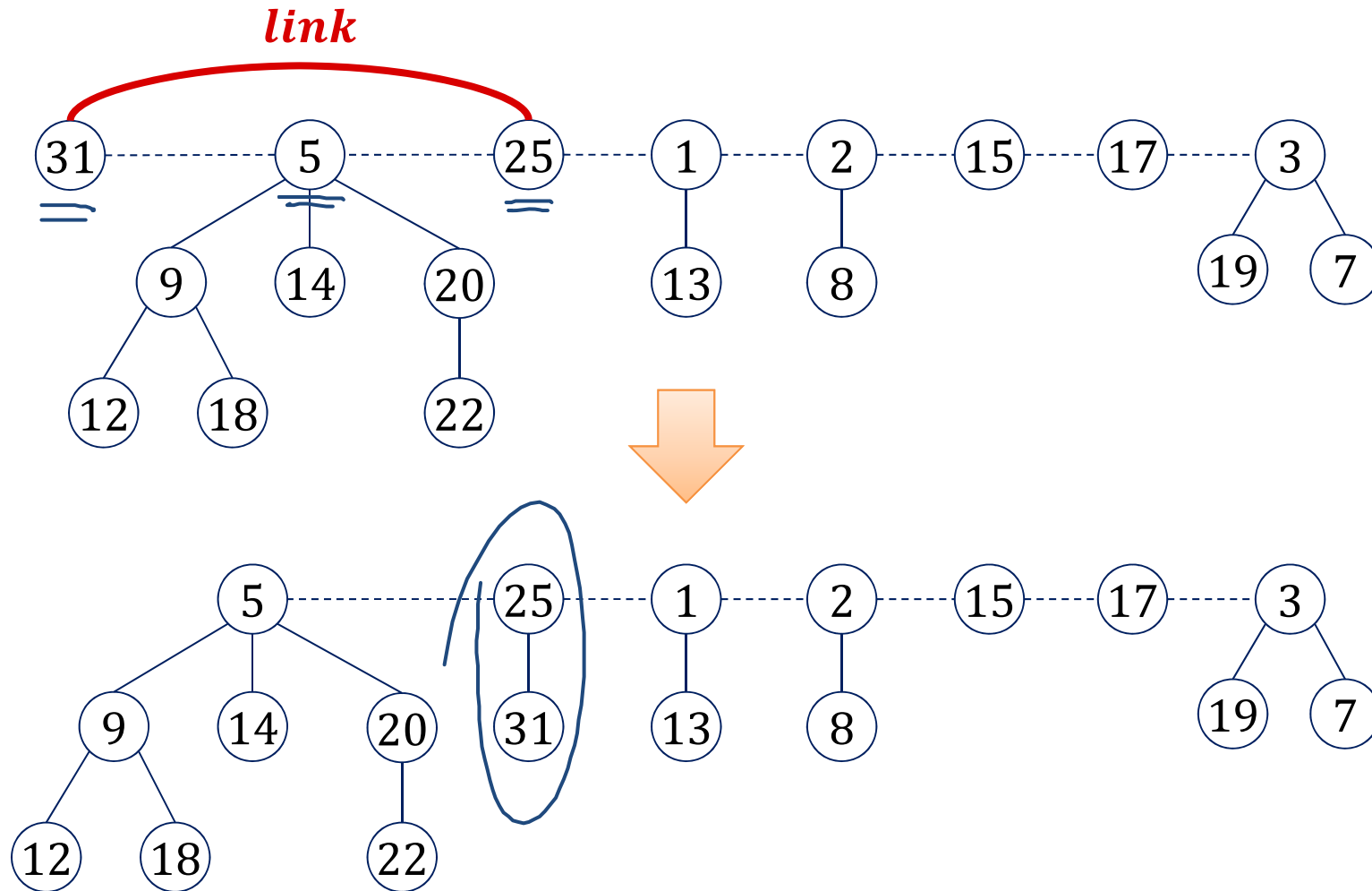
**Time:**  
 $O(|H.rootlist| + D(n))$

↑  
before consolidate

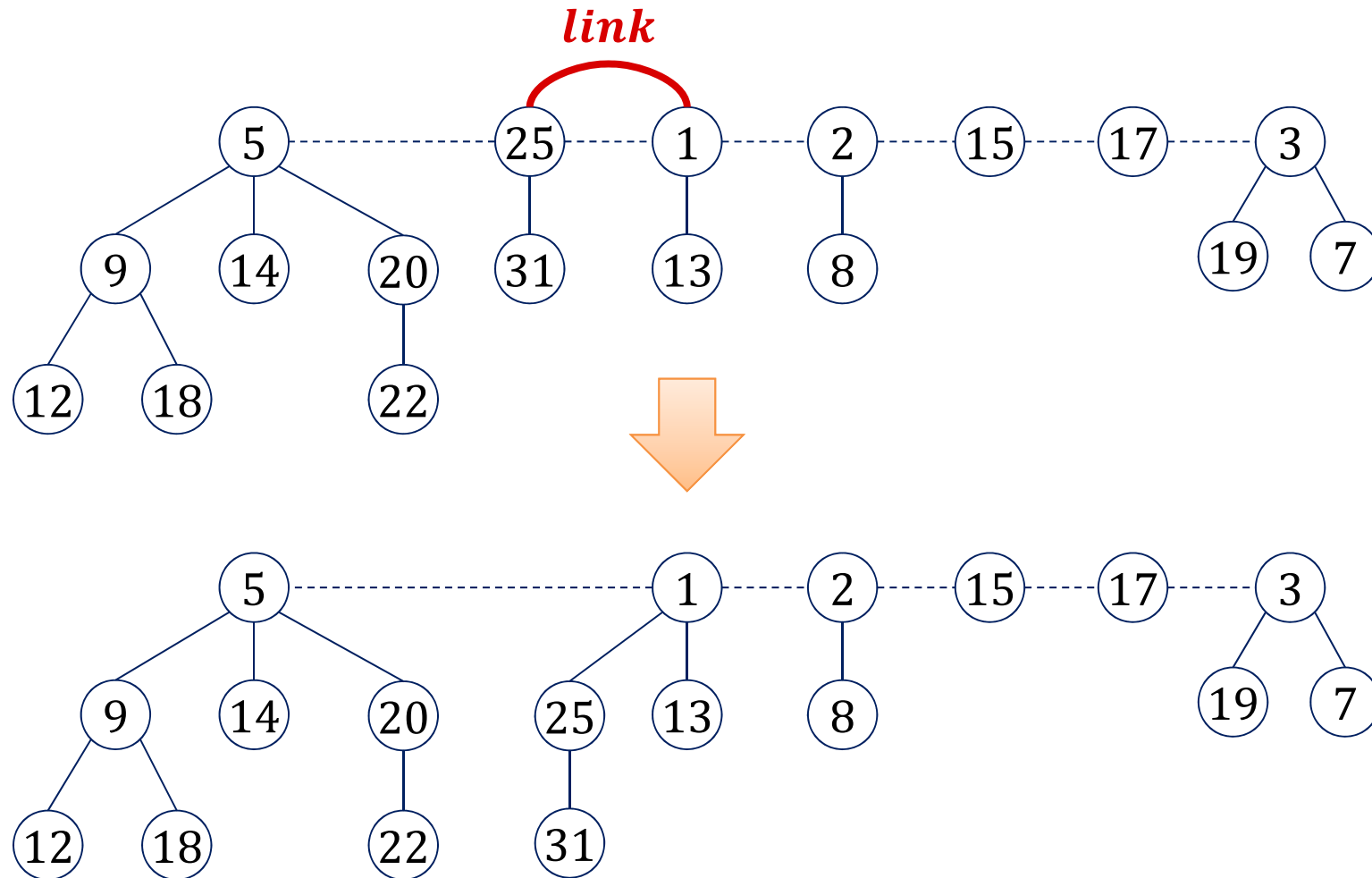
Time:  
 length of rootlist  
 + #link op.  
 +  $O(D(n))$



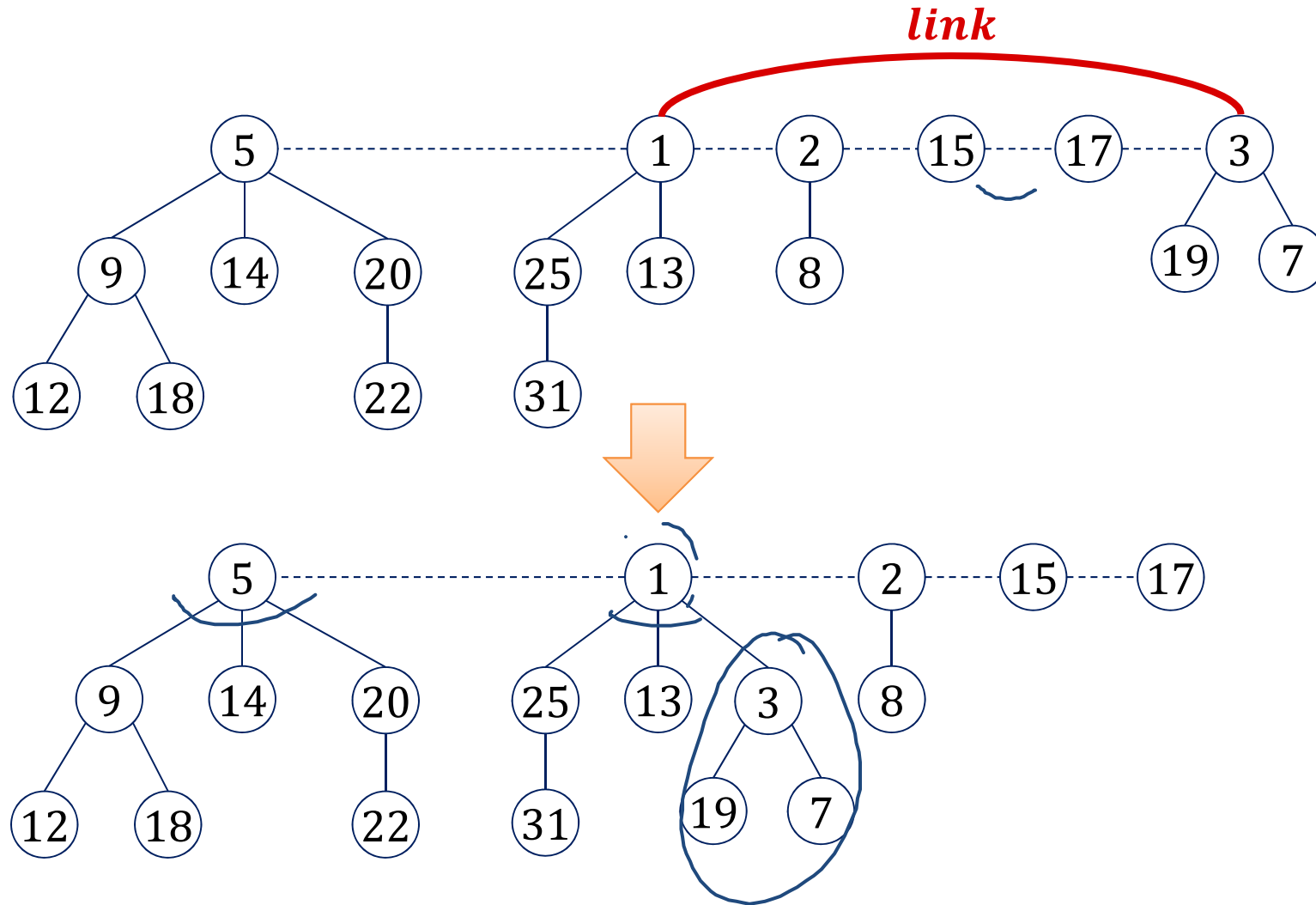
# Consolidate Example



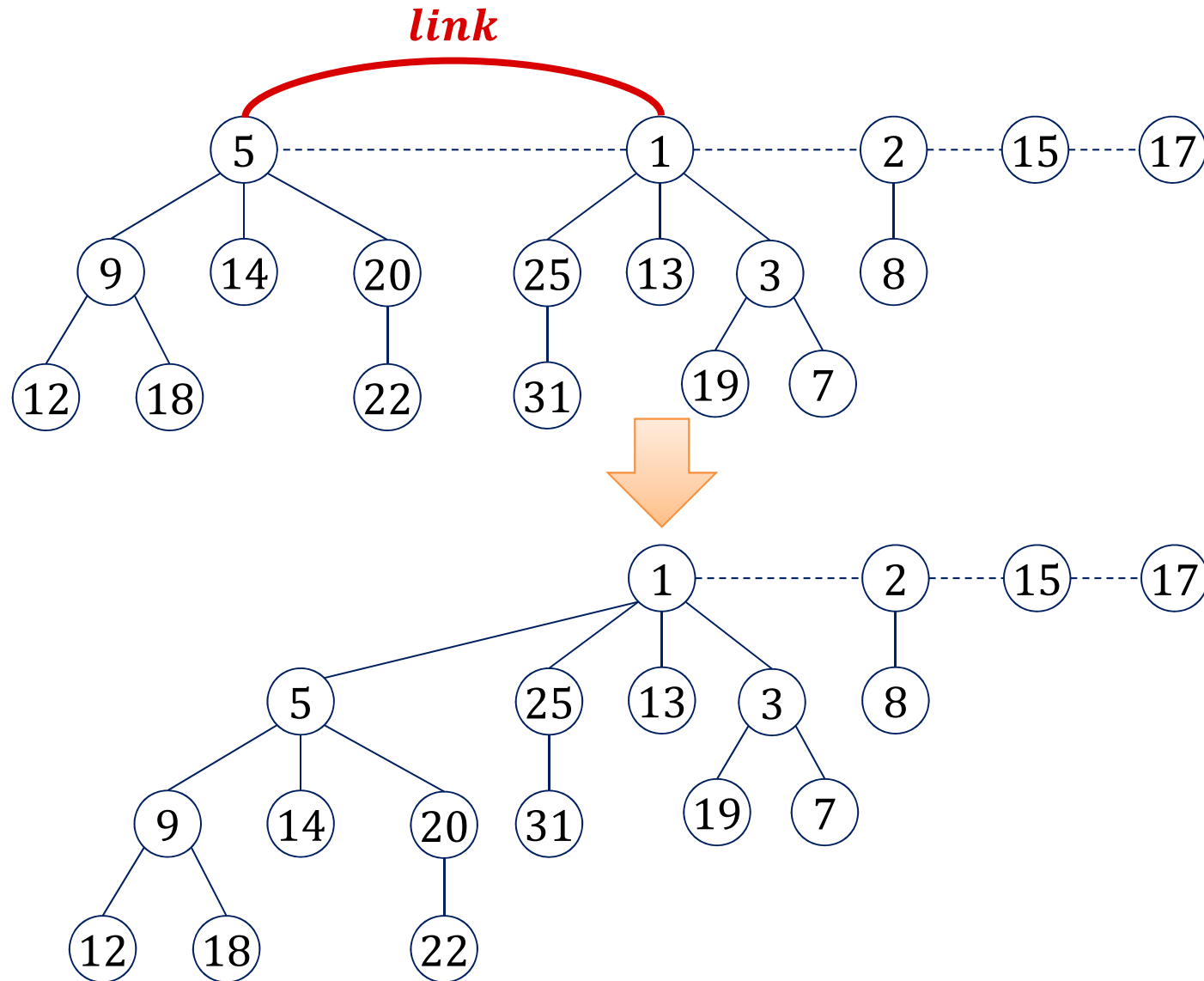
# Consolidate Example



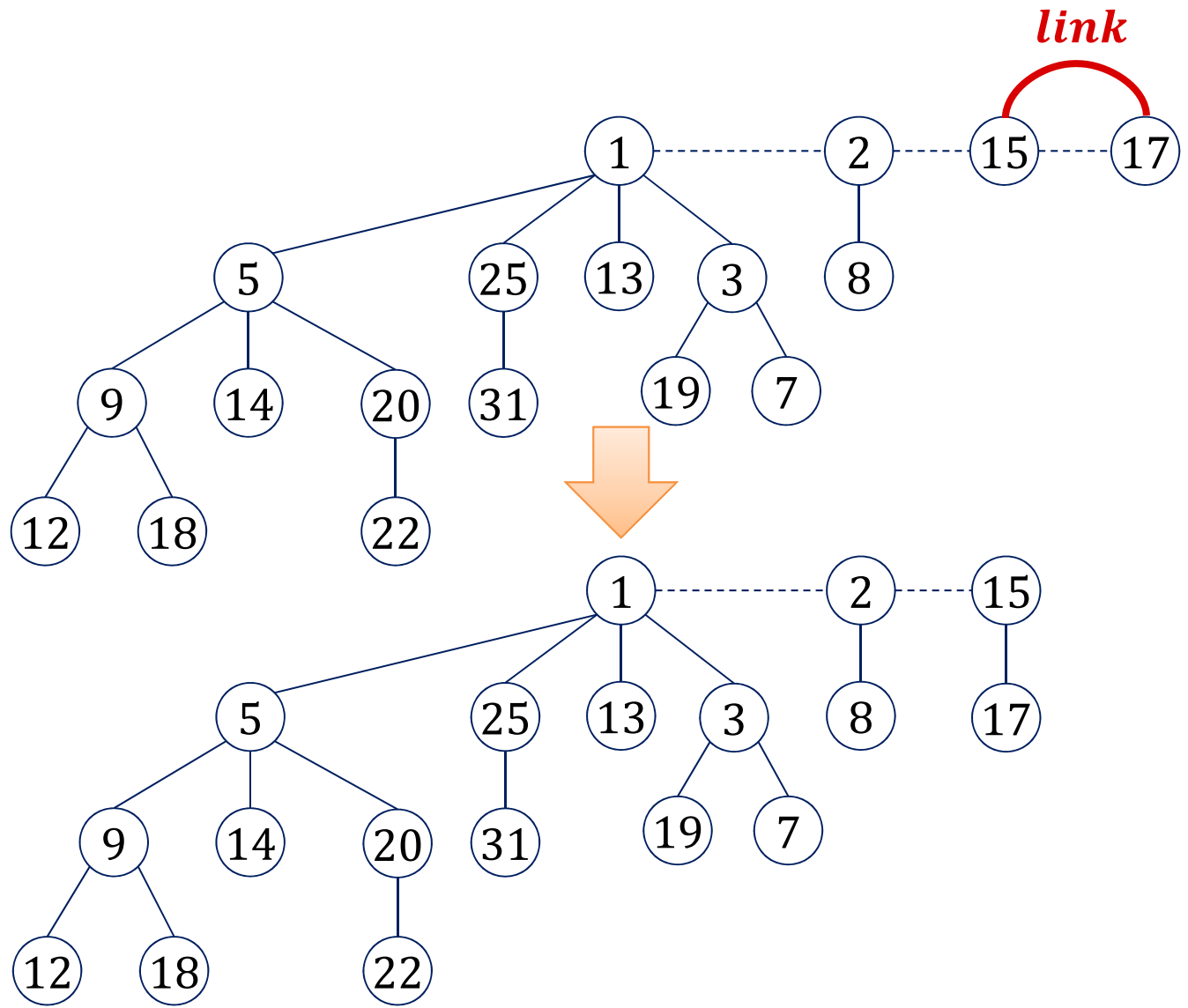
# Consolidate Example



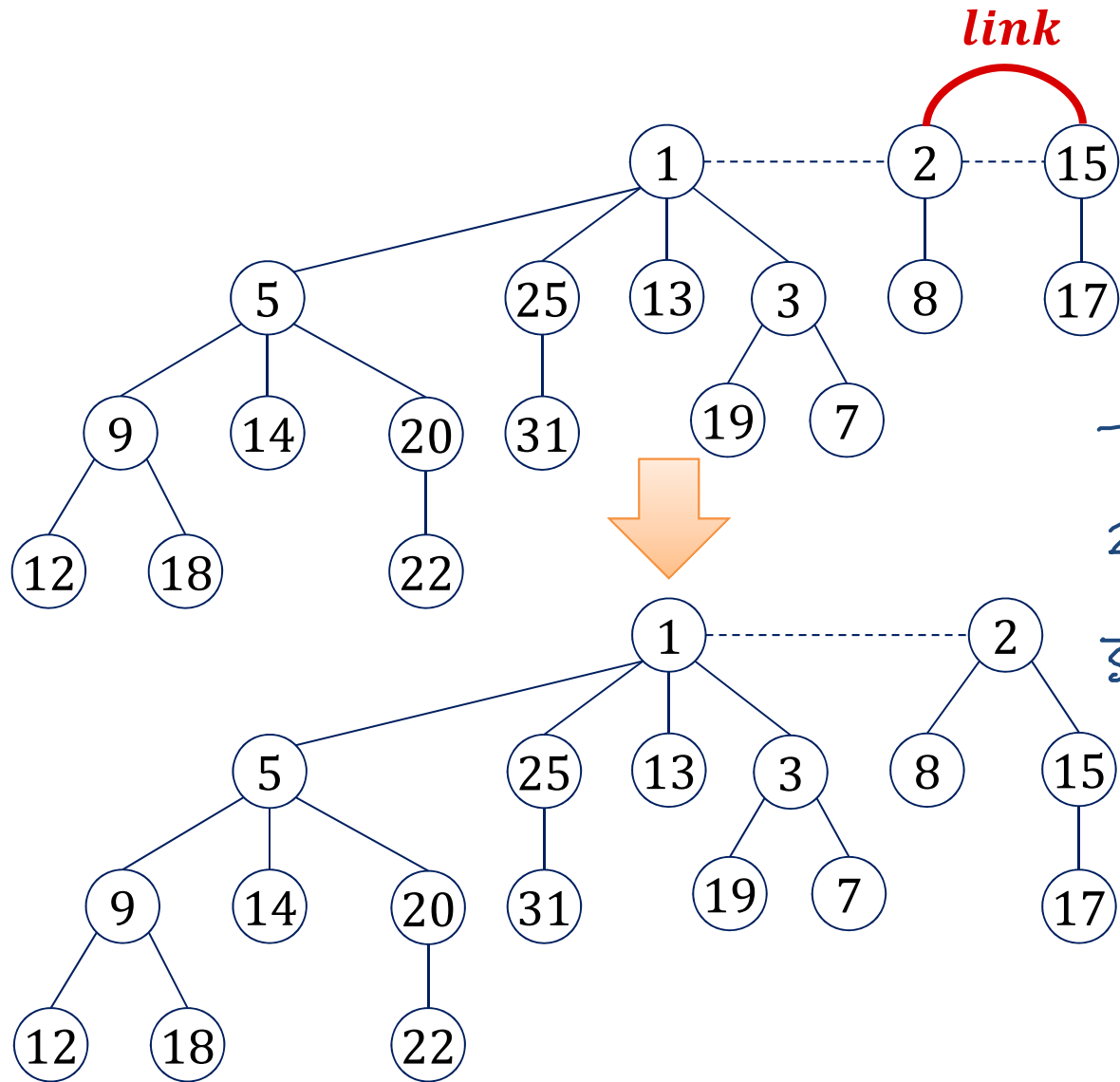
# Consolidate Example



# Consolidate Example



# Consolidate Example



delete-min

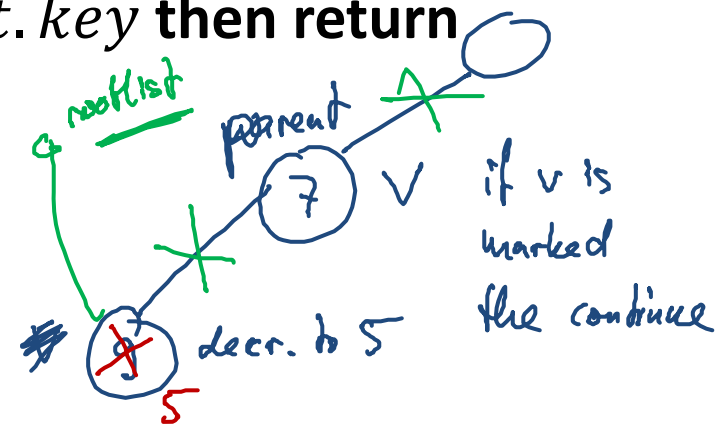
1. delete smallest root
2. add child list to root list } O(1)  
done
3. consolidate

$O(|H_{rootList}| + D(n))$

# 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**  $v.mark := true$ ;

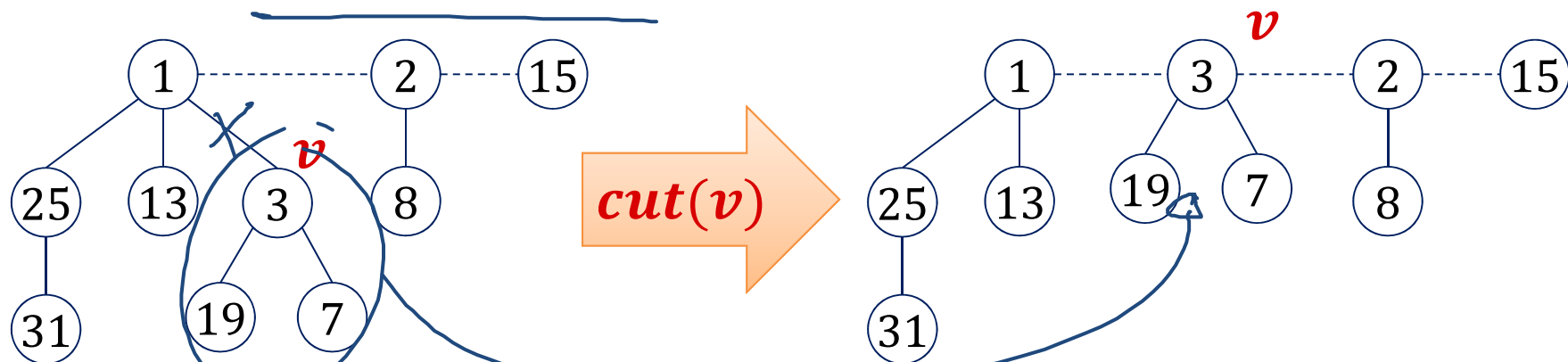


# Operation $\text{Cut}(v)$

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

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

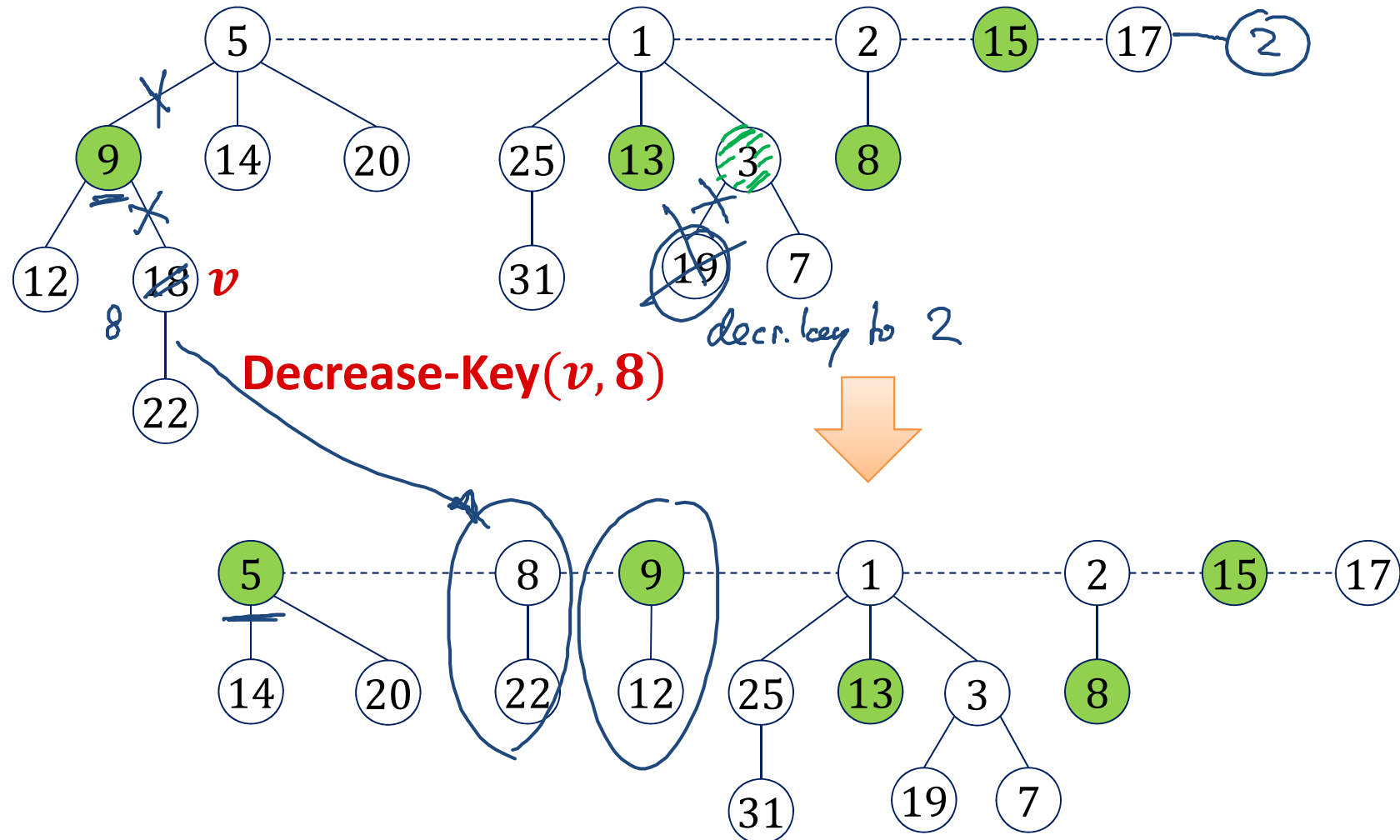
1. **if  $v \notin H.\text{rootlist}$  then**
2.     // cut the link between  $v$  and its parent
3.      $\text{rank}(v.\text{parent}) := \text{rank}(v.\text{parent}) - 1$ ;
4.     remove  $v$  from  $v.\text{parent}.\text{child}$  (list)
5.      $v.\text{parent} := \text{null}$ ;
6.     add  $v$  to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



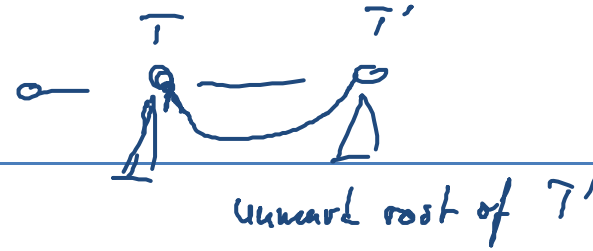


# Decrease-Key Example

- Green nodes are marked



# 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



**$H.cut(v)$**

- 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  $\rightarrow$  requires **amortized analysis**