Chapter 5
Data Structures

Algorithm Theory
WS 2016/17

Fabian Kuhn

Examples

Dictionary:

- Operations: insert(*key,value*), delete(*key*), find(*key*)
- Implementations:
  - Linked list: all operations take $O(n)$ time (n: size of data structure)
  - Balanced binary tree: all operations take $O(\log n)$ time
  - Hash table: all operations take $O(1)$ times (with some assumptions)

Stack (LIFO Queue):

- Operations: push, pull
- Linked list: $O(1)$ for both operations

(FIFO) Queue:

- Operations: enqueue, dequeue
- Linked list: $O(1)$ time for both operations

Here: **Priority Queues (heaps)**, **Union-Find data structure**

Dijkstra's Algorithm

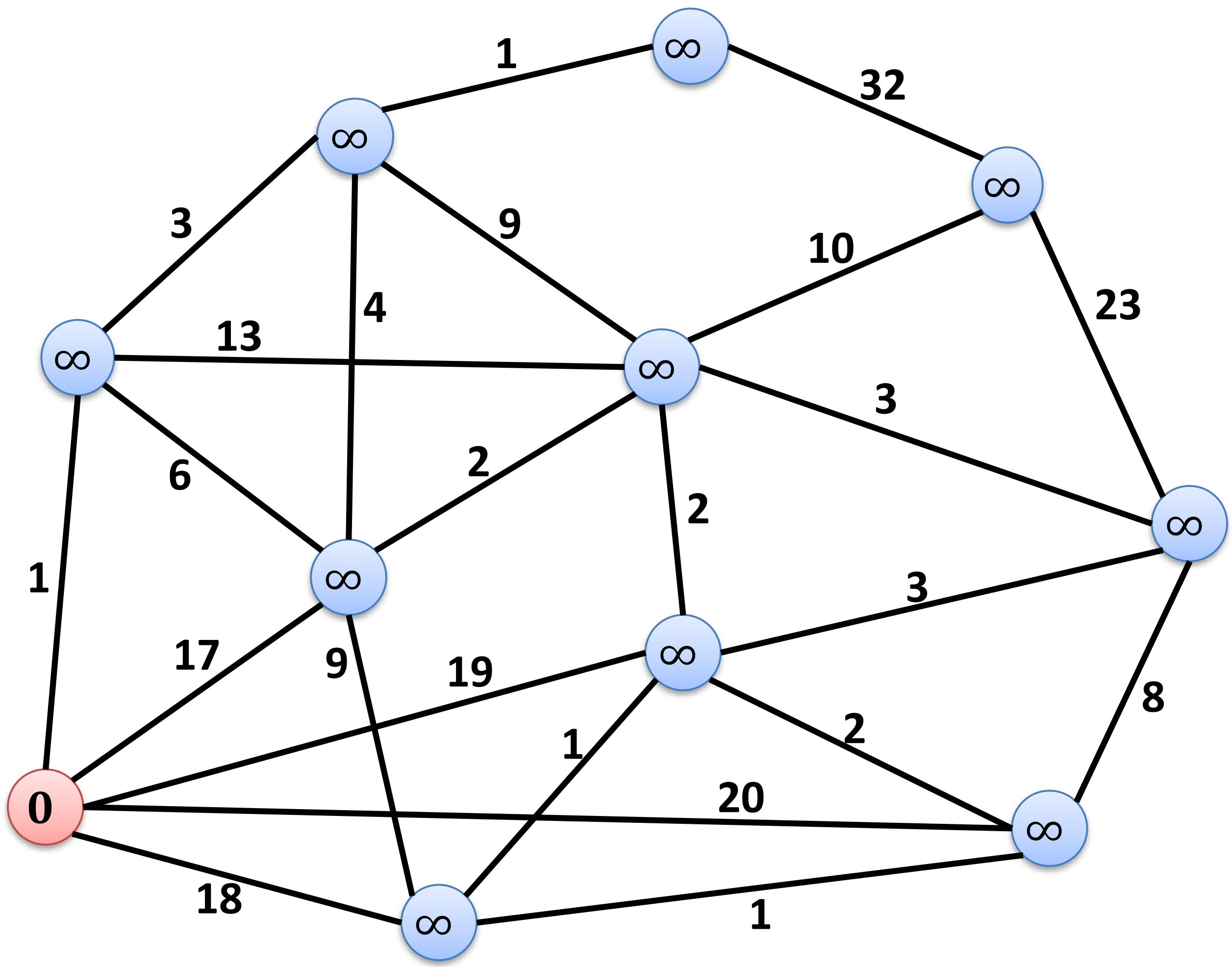
Single-Source Shortest Path Problem:

- **Given:** graph $G = (V, E)$ with edge weights $w(e) \geq 0$ for $e \in E$
  source node $s \in V$
- **Goal:** compute shortest paths from s to all $v \in V$

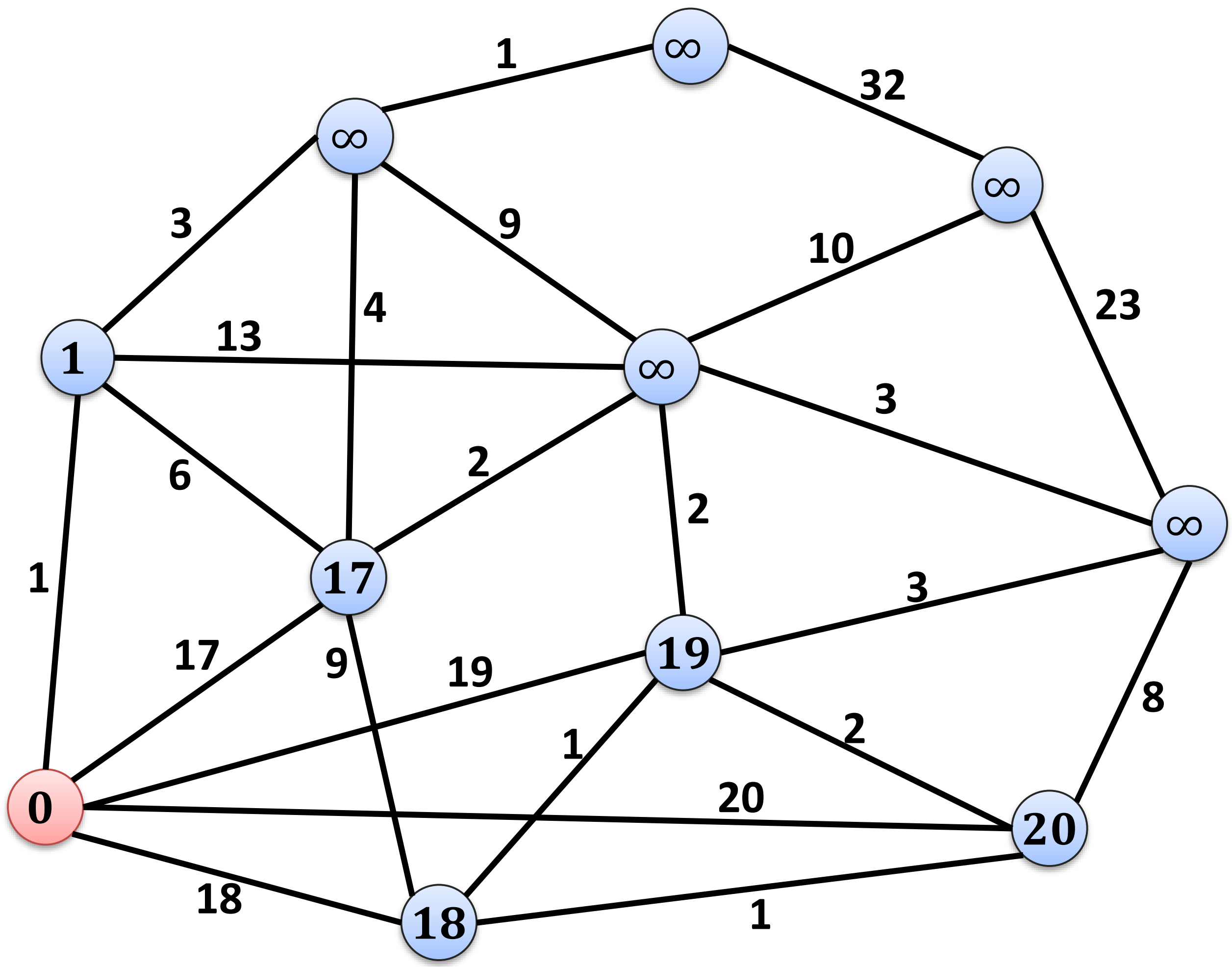
Dijkstra's Algorithm:

1. Initialize $d(s, s) = 0$ and $d(s, v) = \infty$ for all $v \neq s$
2. All nodes are unmarked
3. Get unmarked node u which minimizes $d(s, u)$:
4. For all $e = \{u, v\} \in E$, $d(s, v) = \min\{d(s, v), d(s, u) + w(e)\}$
5. mark node u
6. Until all nodes are marked

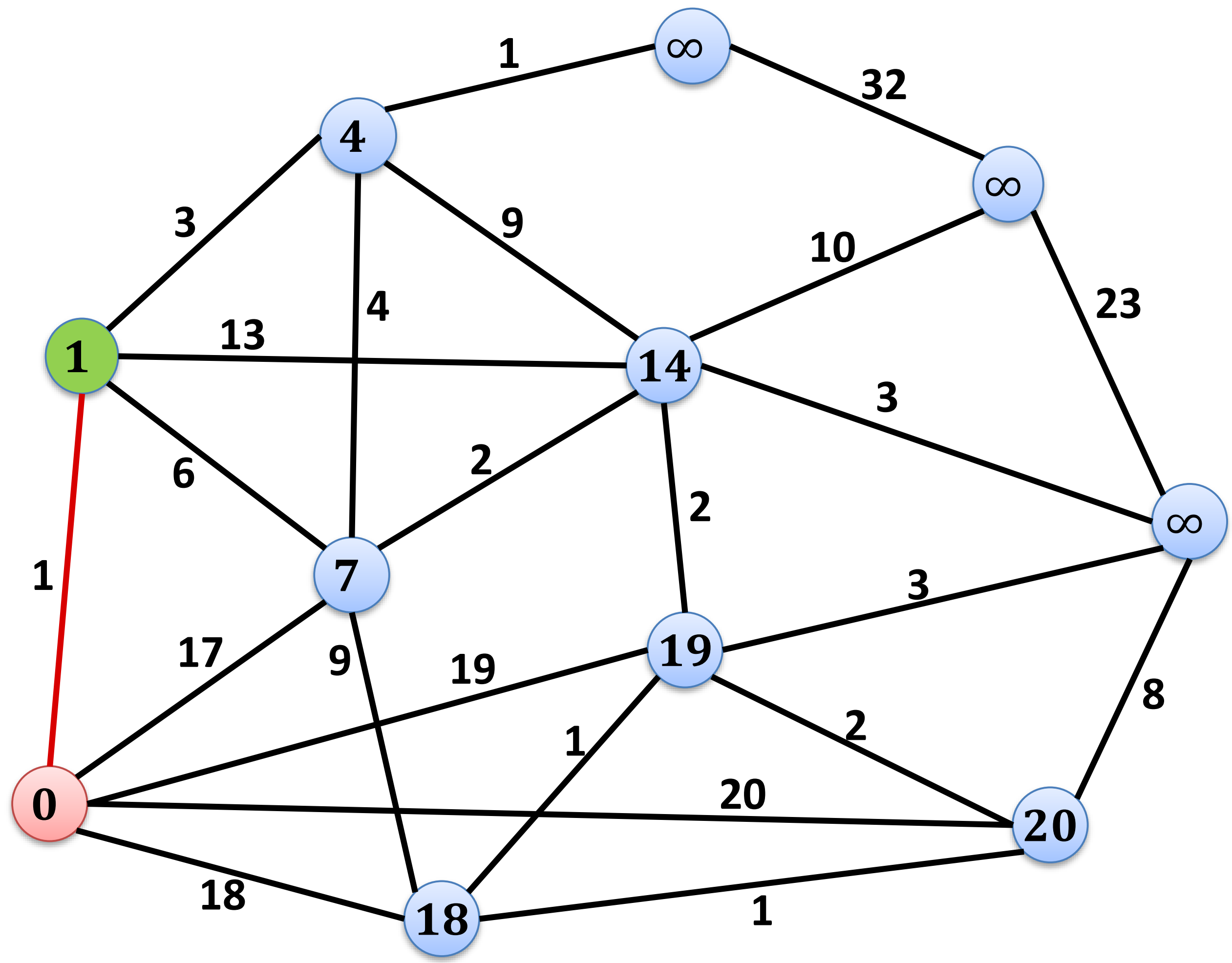
Example



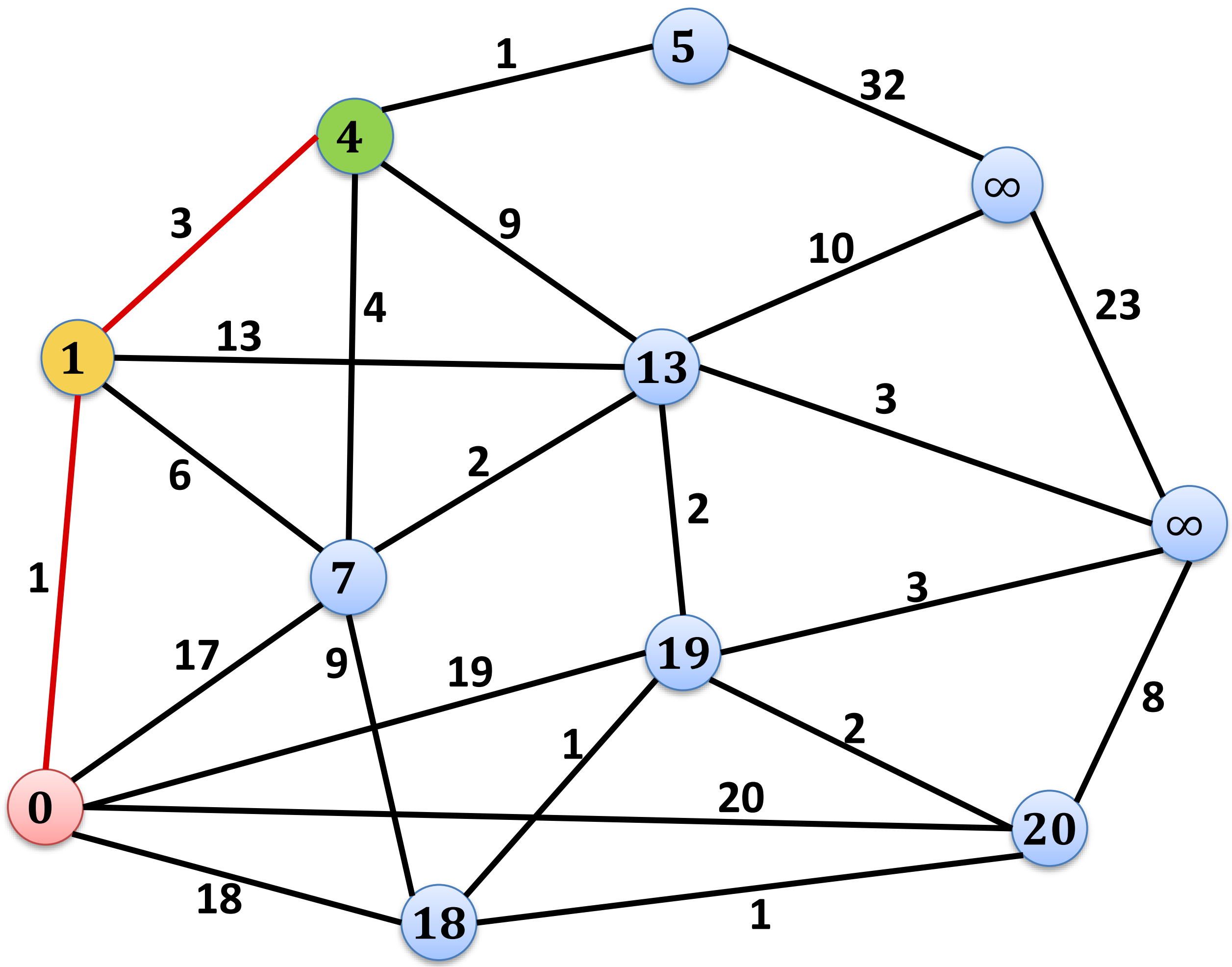
Example



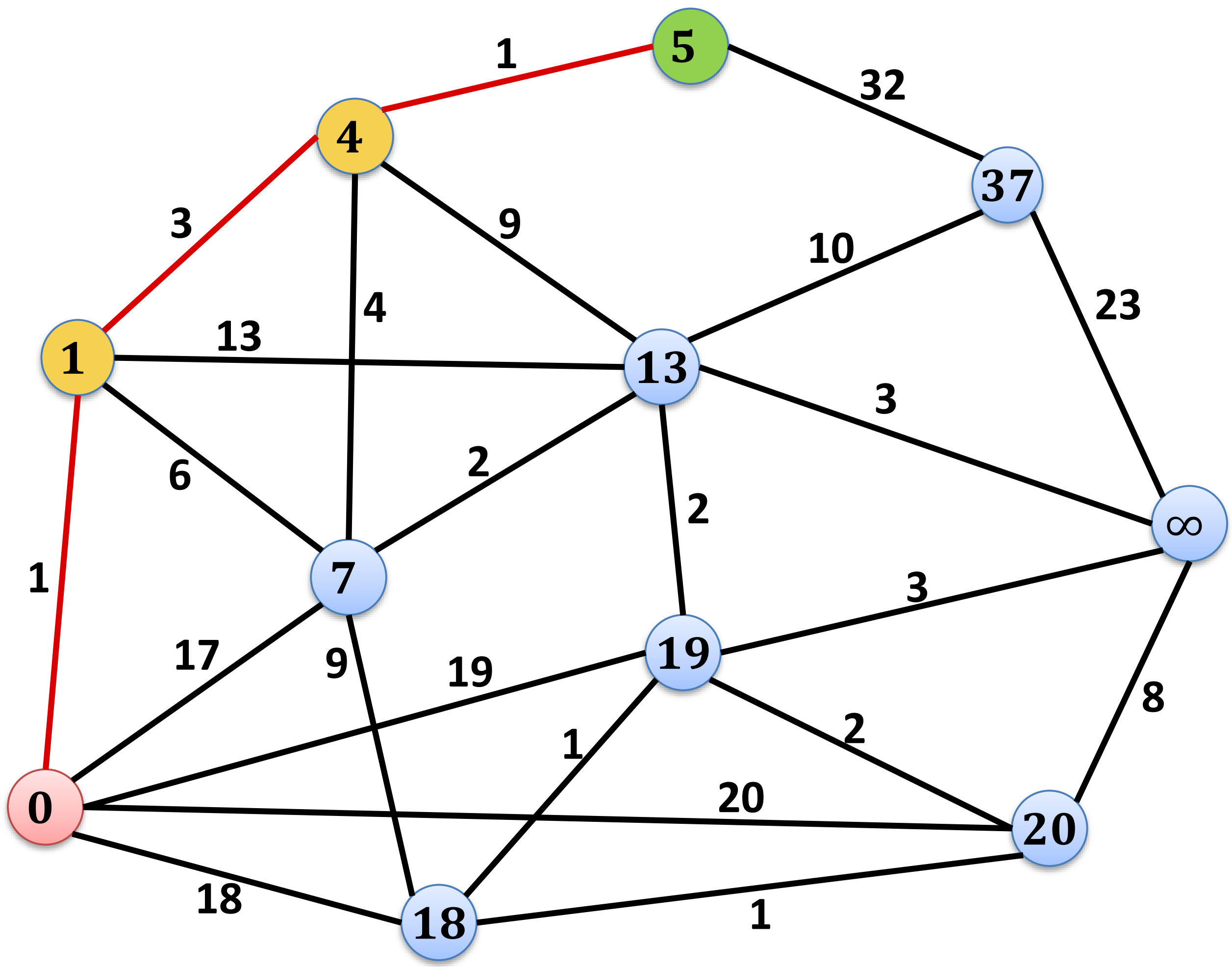
Example



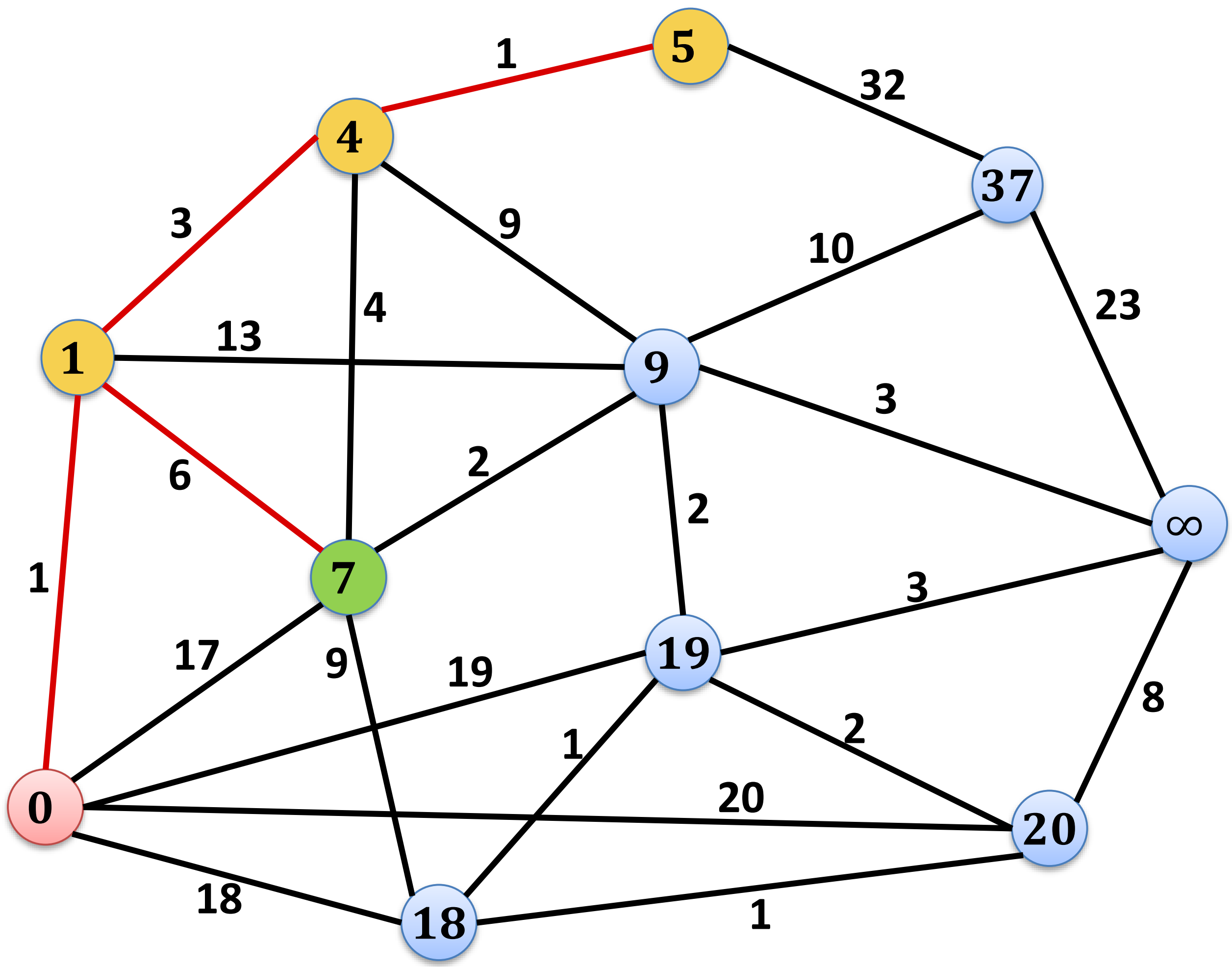
Example



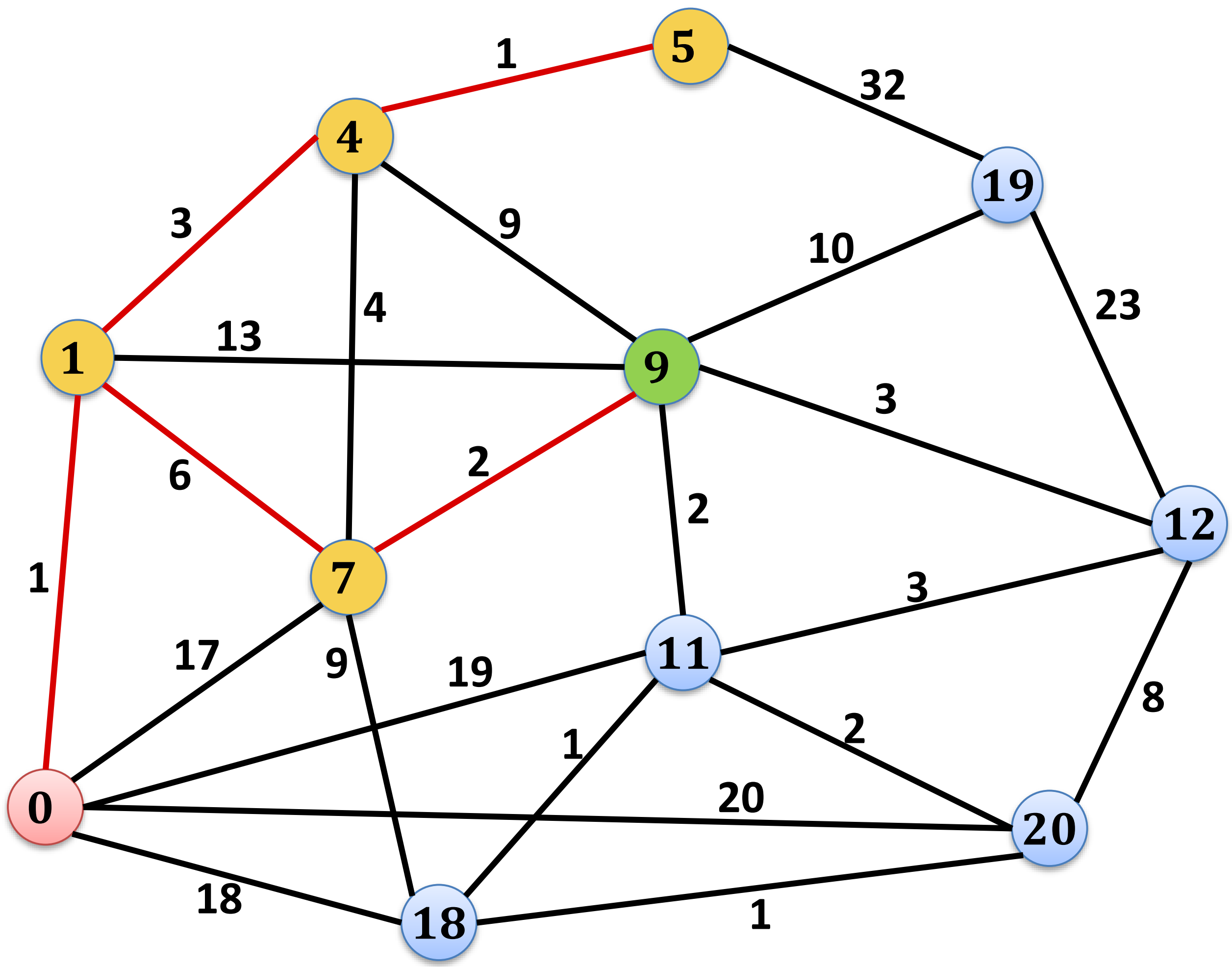
Example



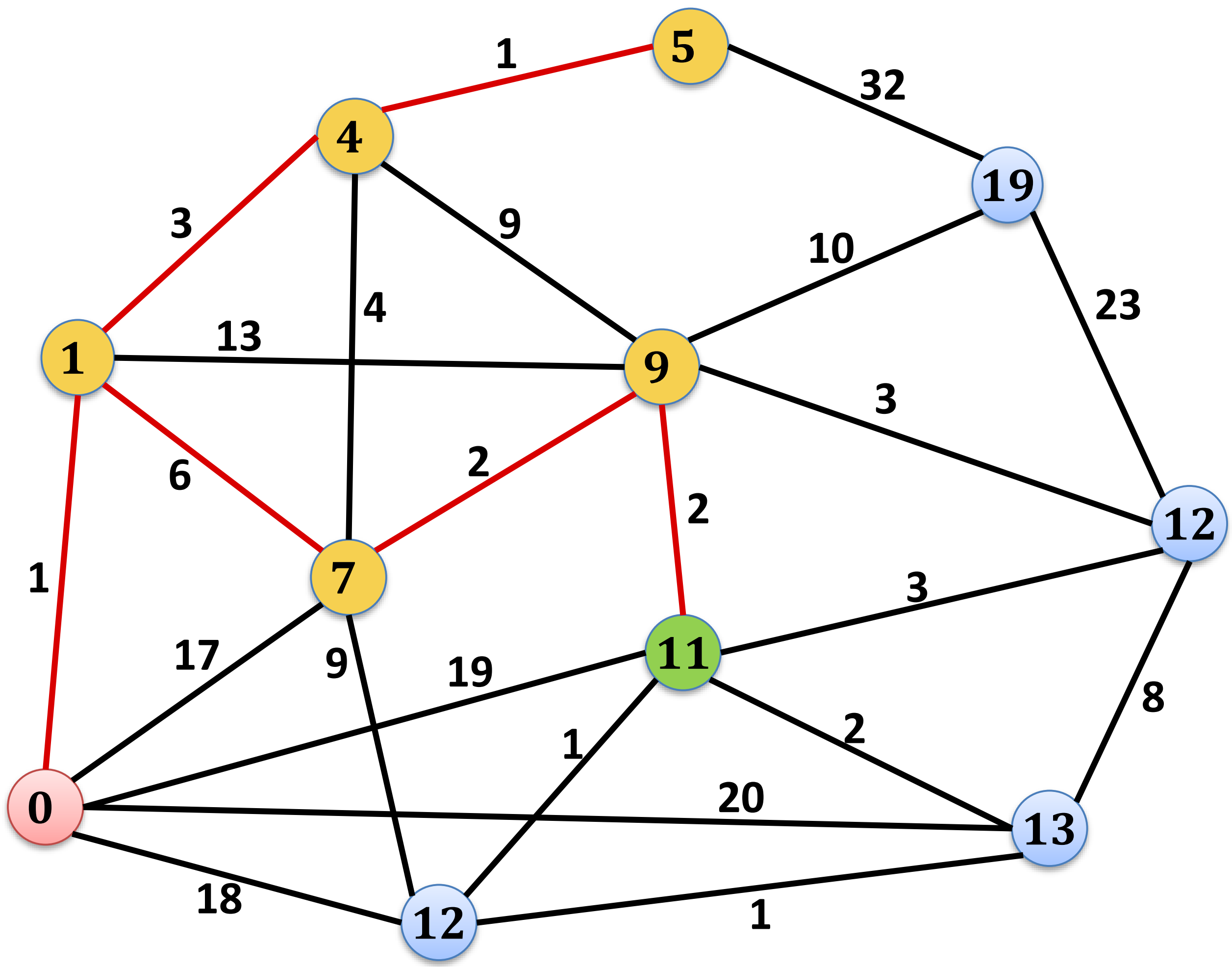
Example



Example



Example



Implementation of Dijkstra's Algorithm

Dijkstra's Algorithm:

1. Initialize $d(s,s) = 0$ and $d(s,v) = \infty$ for all $v \neq s$
2. All nodes $v \neq s$ are unmarked

3. Get unmarked node u which minimizes $d(s,u)$:

4. For all $e = \{u,v\} \in E$, $d(s,v) = \min\{d(s,v), d(s,u) + w(e)\}$

5. mark node u

6. Until all nodes are marked

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

Implementation of Dijkstra's Algorithm

Store nodes in a priority queue, use $\boldsymbol{d(s,v)}$ as keys:

1. Initialize $d(s,s) = 0$ and $d(s,v) = \infty$ for all $v \neq s$
2. All nodes $v \neq s$ are unmarked

3. Get unmarked node u which minimizes $d(s,u)$:

4. mark node u

5. For all $e = \{u,v\} \in E$, $d(s,v) = \min\{d(s,v), d(s,u) + w(e)\}$

6. Until all nodes are marked

Analysis

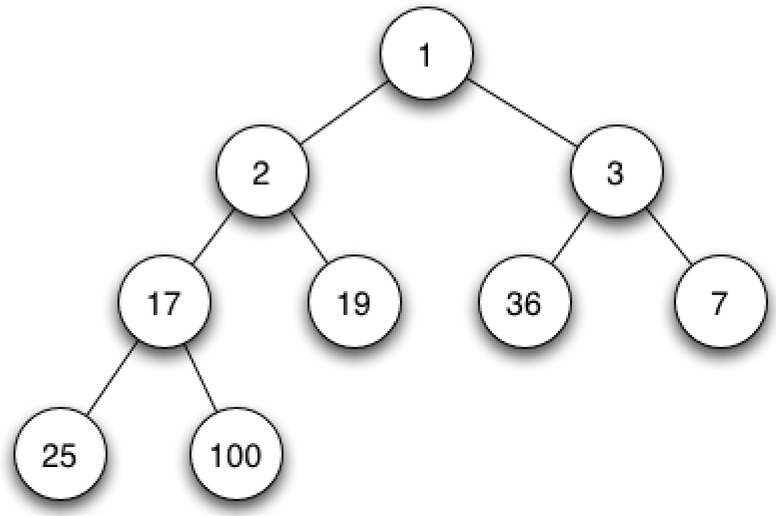
Number of priority queue operations for Dijkstra:

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

Priority Queue Implementation

Implementation as min-heap:

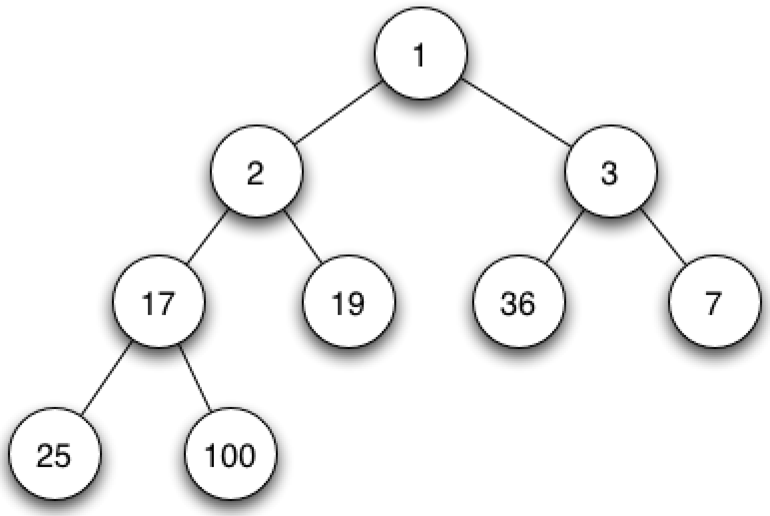
→ complete binary tree,
e.g., stored in an array



Priority Queue Implementation

Implementation as min-heap:

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



- **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, $m \leq n$): $\boldsymbol{O(m \log n)}$

Can We Do Better?

- Cost of **Dijkstra** with **complete binary min-heap** implementation:

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

- **Binary heap:**
  insert, delete-min, and decrease-key cost $O(\log n)$
  merging two heaps is expensive

- 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 merge, decrease-key, and one of the other two operations?

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:

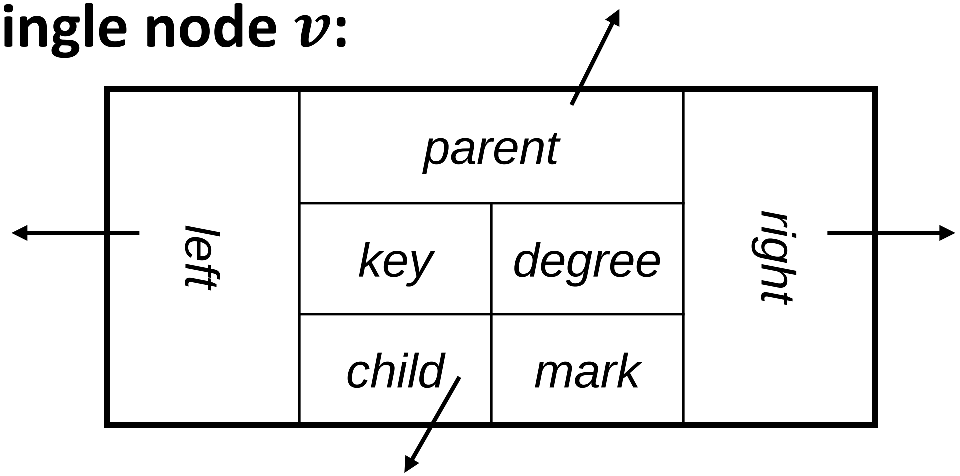
- $H.min$: root of the tree containing the (a) minimum key
- $H.rootlist$: circular, doubly linked, unordered list containing the roots of all trees
- $H.size$: number of nodes currently in H

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

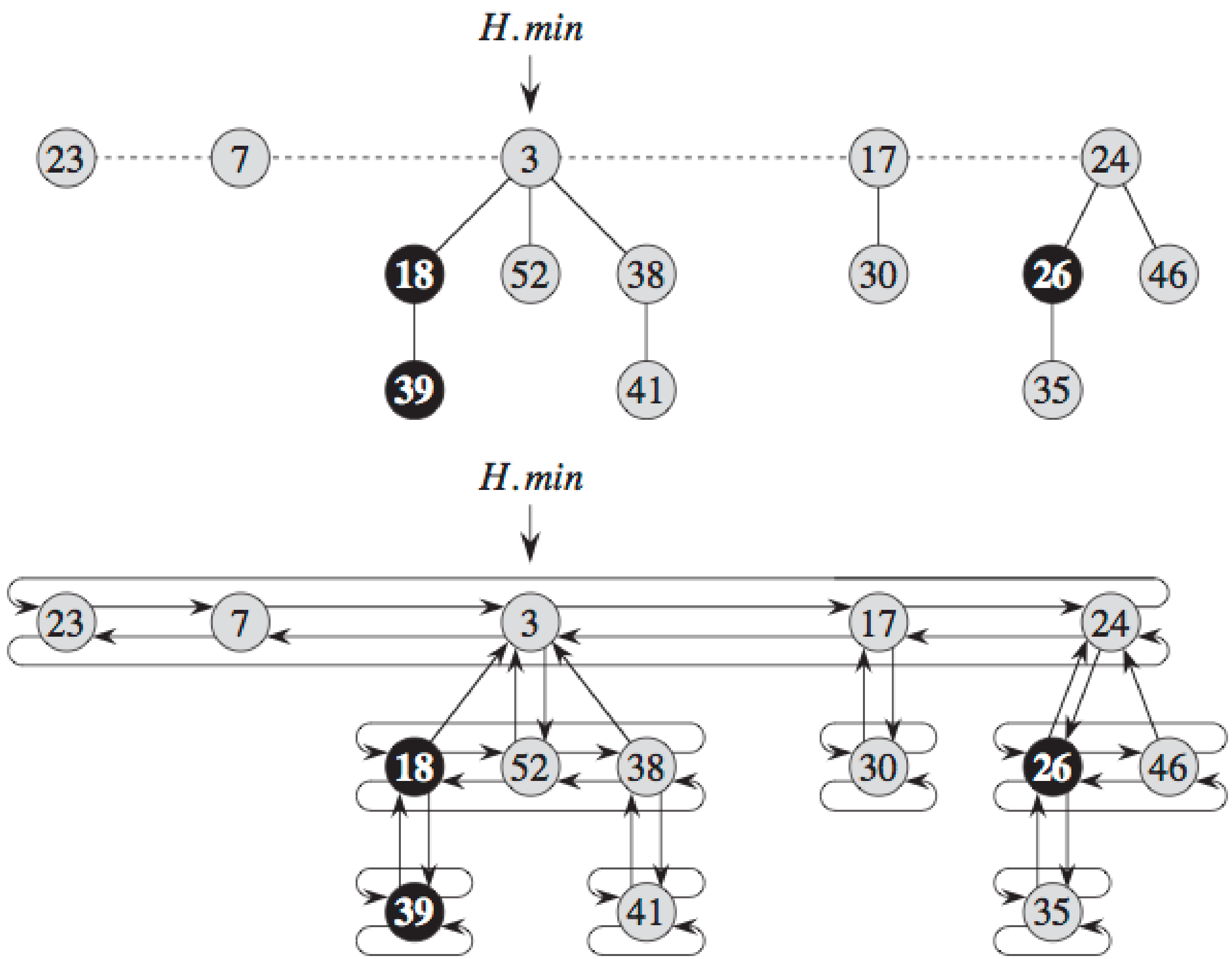


Figure: Cormen et al., Introduction to Algorithms

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:

- $rank(v)$: 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 $D(n)$

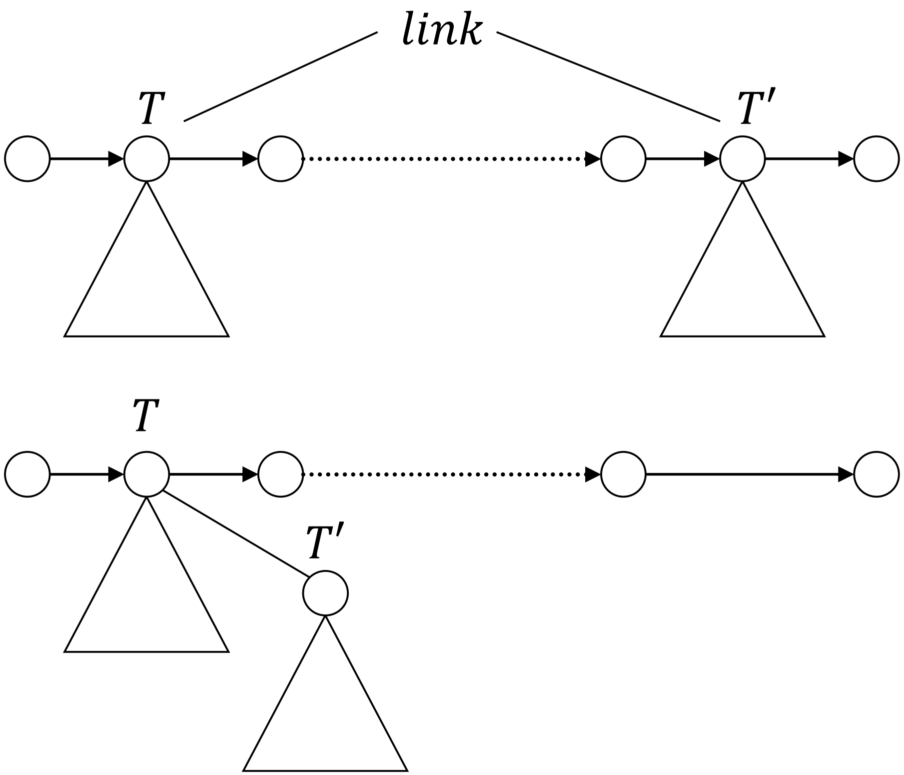
Merging Two Trees

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

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

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

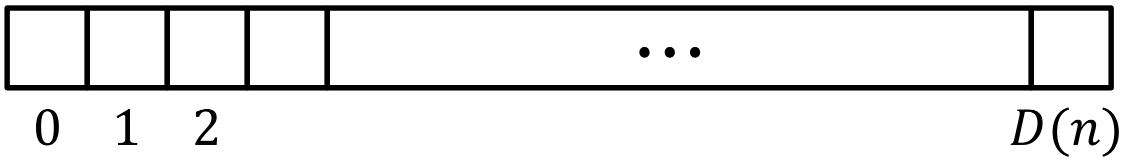
- Removes tree T' from root list and adds T' to child list of T



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

Consolidation of Root List

Array A pointing to find roots with the same rank:

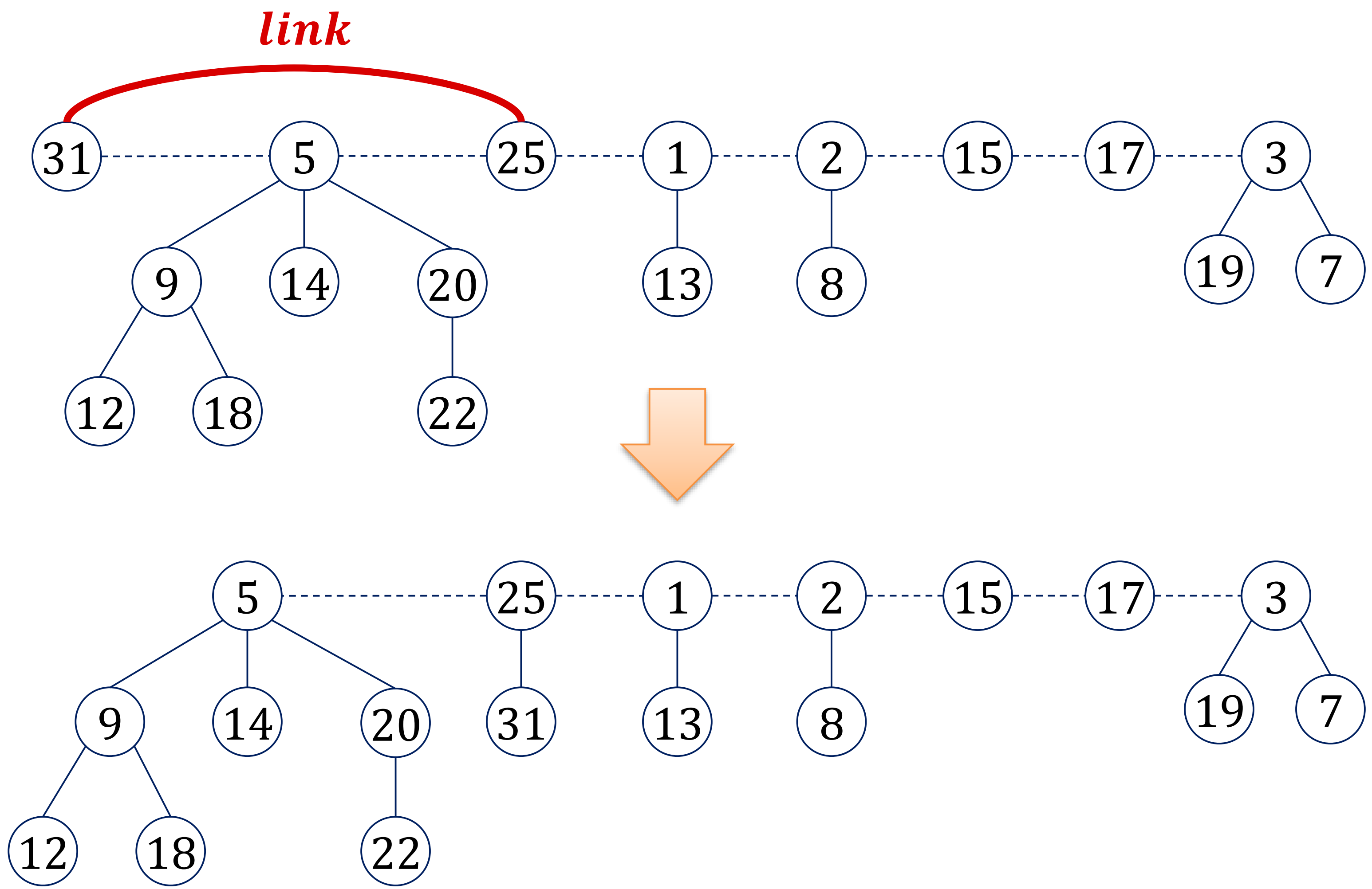


Consolidate:

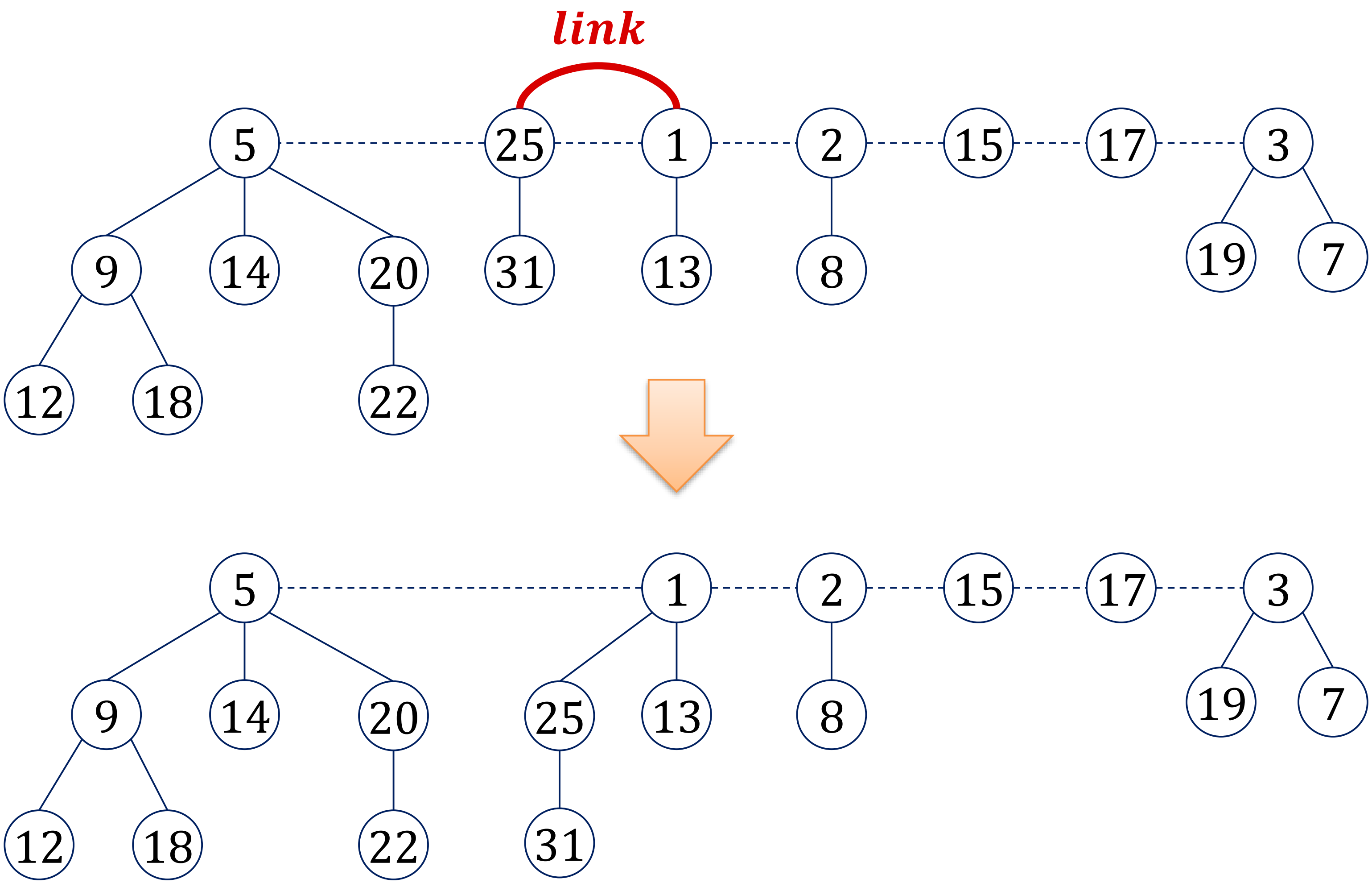
for $i := 0$ **to** $D(n)$ **do** $A[i] := \text{null}$;
while $H.rootlist \neq \text{null}$ **do**
    $T :=$ "delete and return first element of $H.rootlist$"
    while $A[rank(T)] \neq \text{null}$ **do**
        $T' := A[rank(T)]$;
        $A[rank(T)] := null$;
        $T := link(T, T')$
    $A[rank(T)]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

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

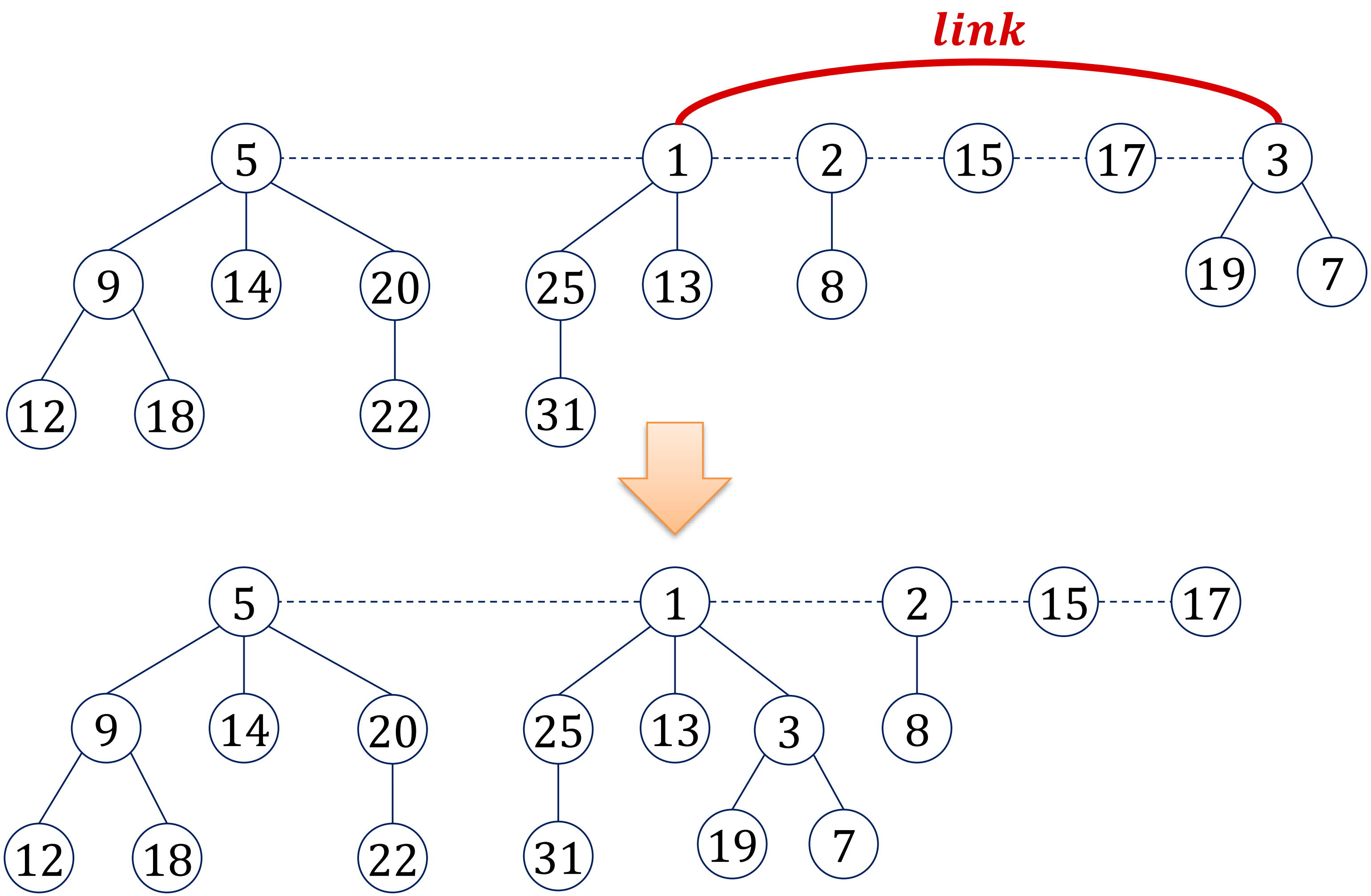
Consolidate Example



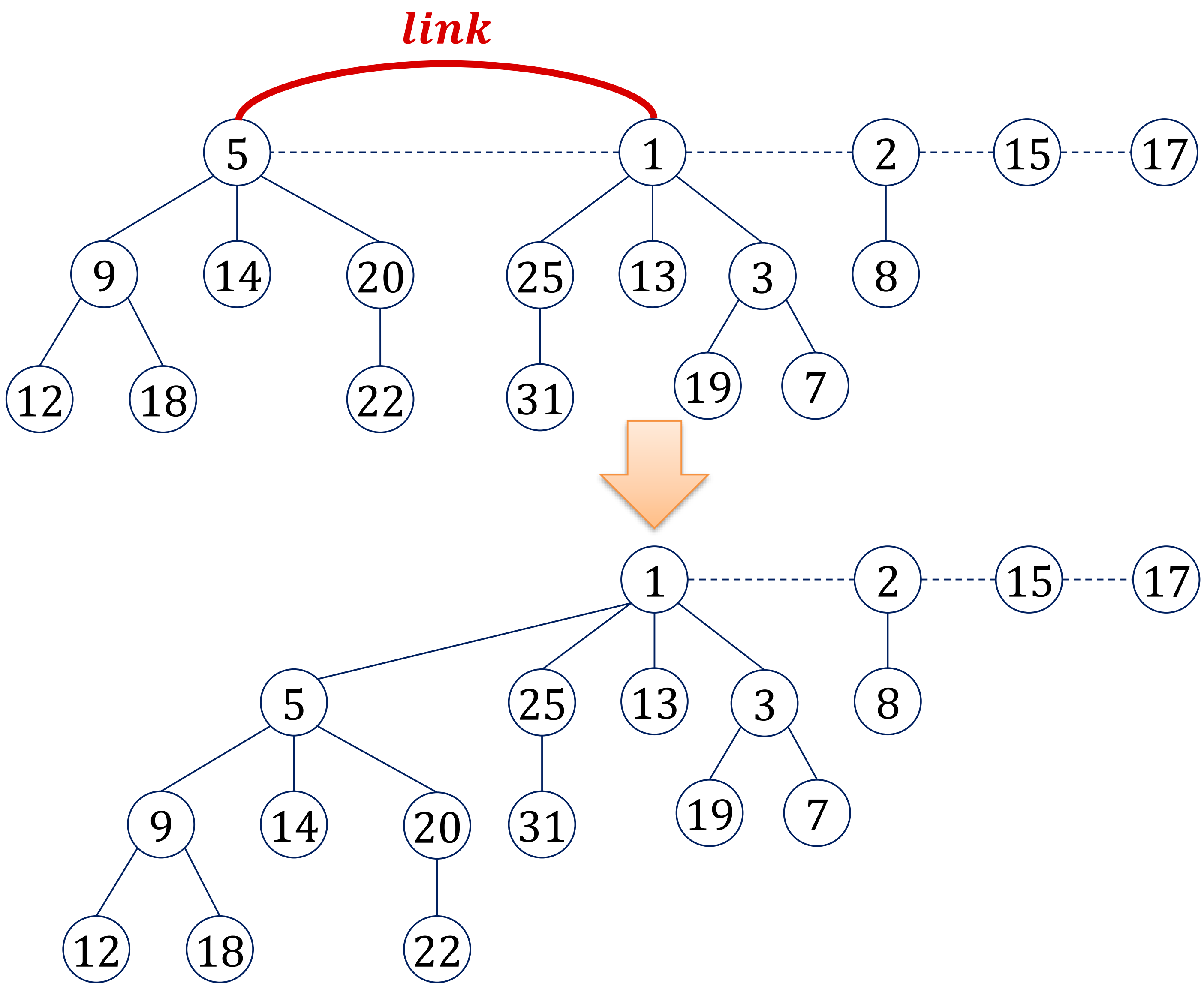
Consolidate Example



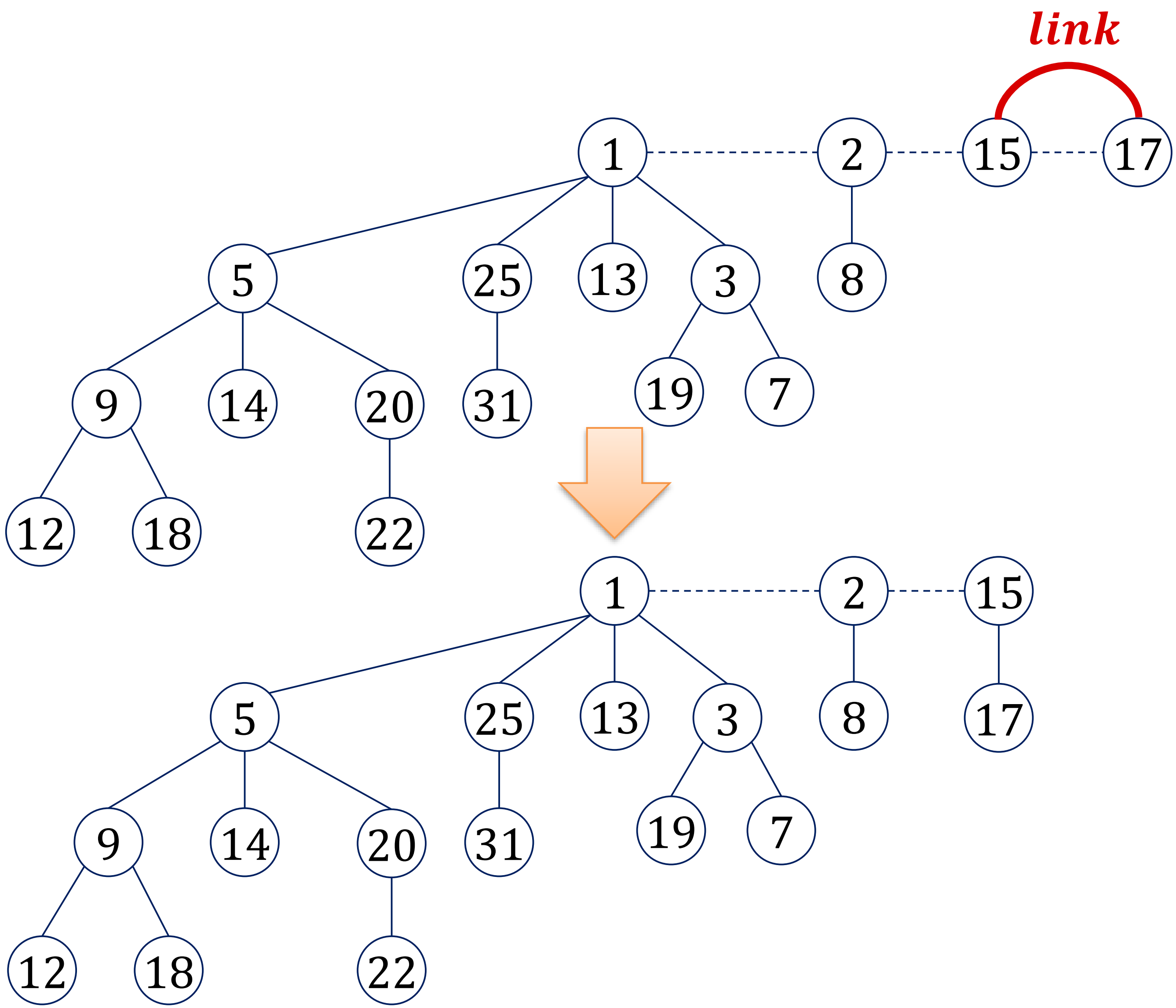
Consolidate Example



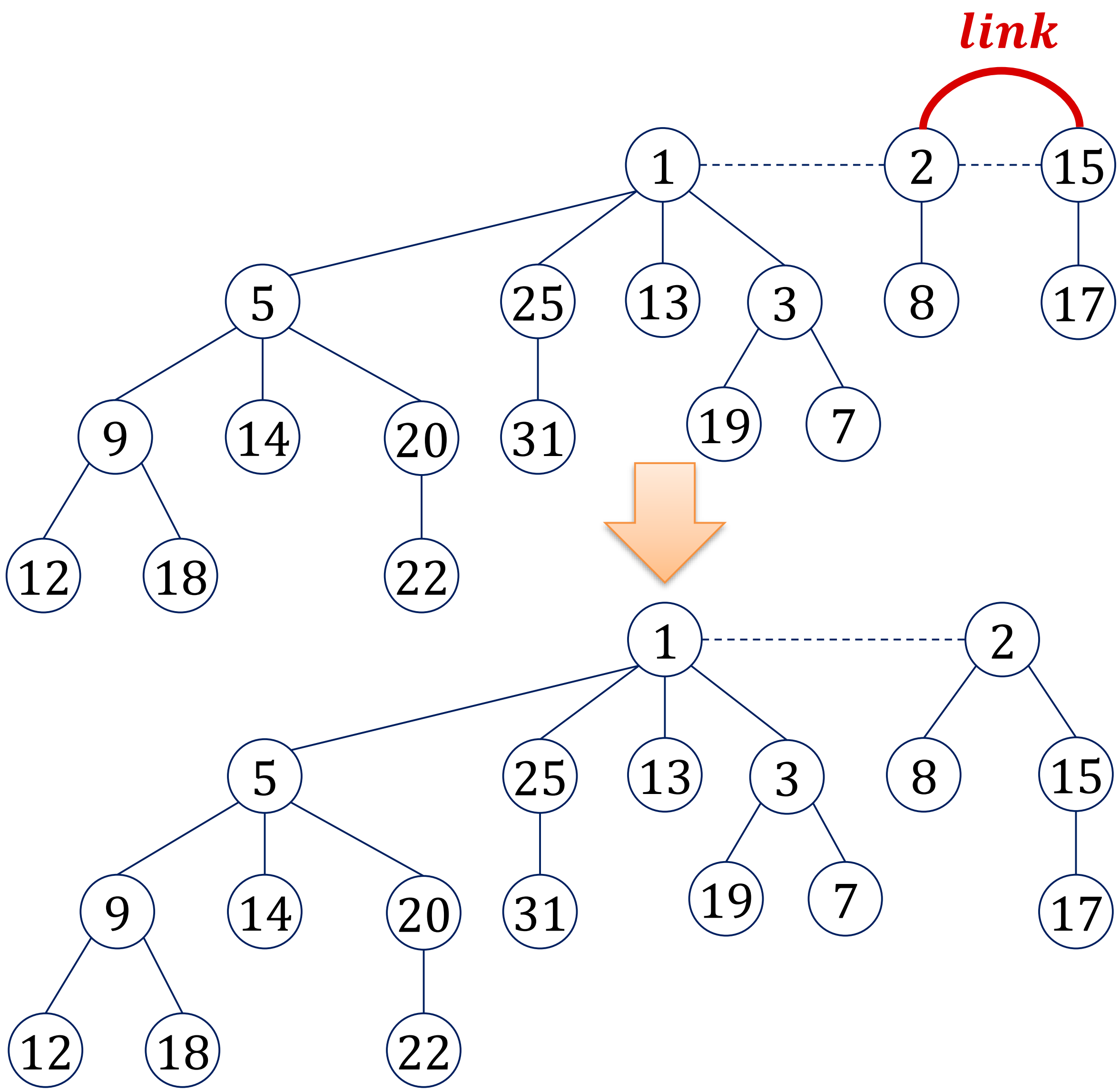
Consolidate Example



Consolidate Example



Consolidate Example



Operation Decrease-Key

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

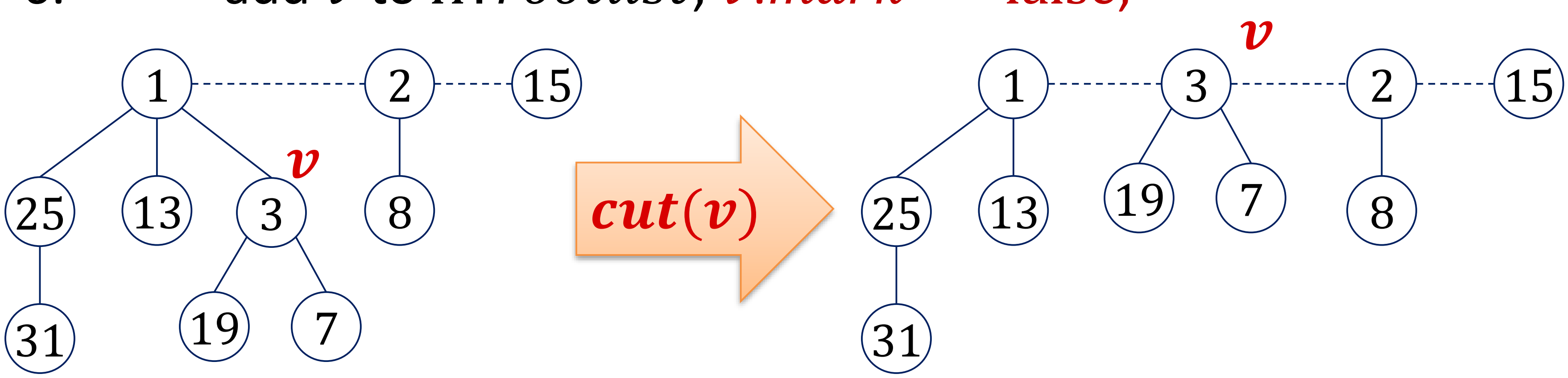
if $x \geq v.key$ **then return**;
$v.key := x$; update $H.min$;
if $v \in H.rootlist \ \lor\ x \geq v.parent.key$ **then return**
repeat
    $parent := v.parent$;
    $\boldsymbol{H.cut(v)}$;
    $v := parent$;
until $\neg(\boldsymbol{v.mark}) \ \lor\ v \in H.rootlist$;
if $v \notin H.rootlist$ **then** $\boldsymbol{v.mark} := \mathbf{true}$;

Operation Cut(v)

Operation $H.cut(v)$:

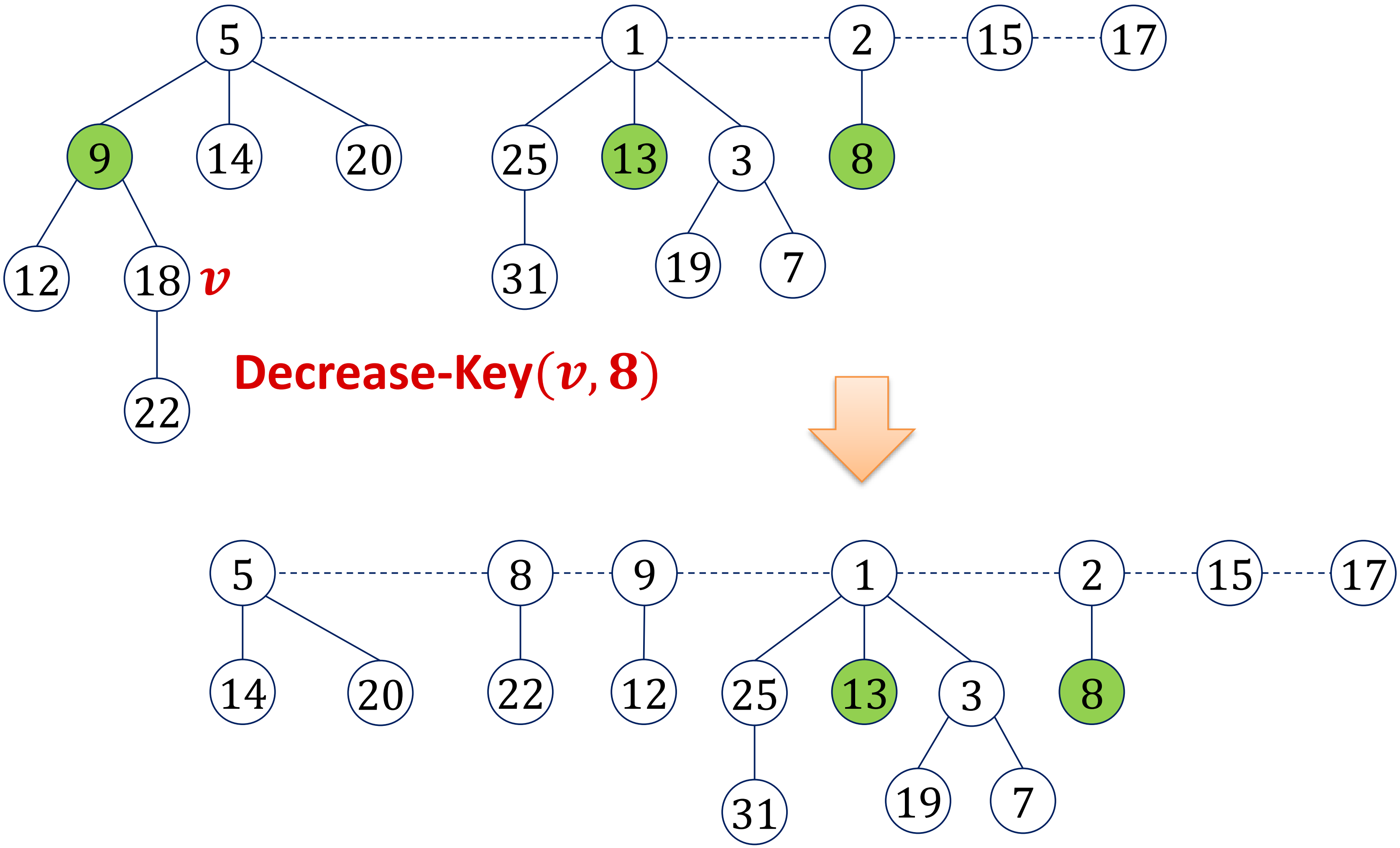
- Cuts v's sub-tree from its parent and adds v to rootlist

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 ⇨ $v.mark = \textbf{false}$

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

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

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

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**