Chapter 5
Data Structures

Algorithm Theory
WS 2015/16

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

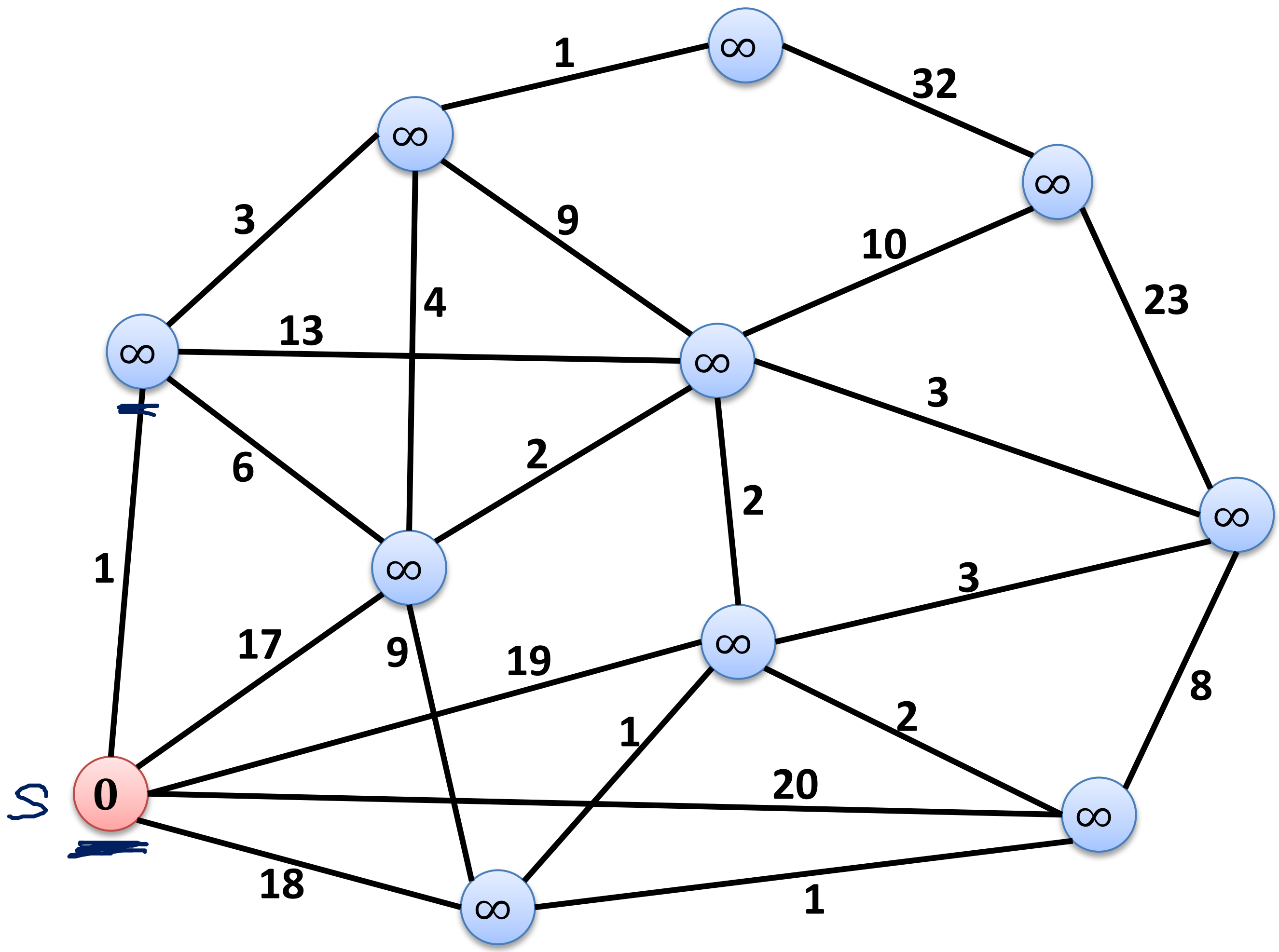
Single-Source Shortest Path Problem:

- **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 $v \in V$

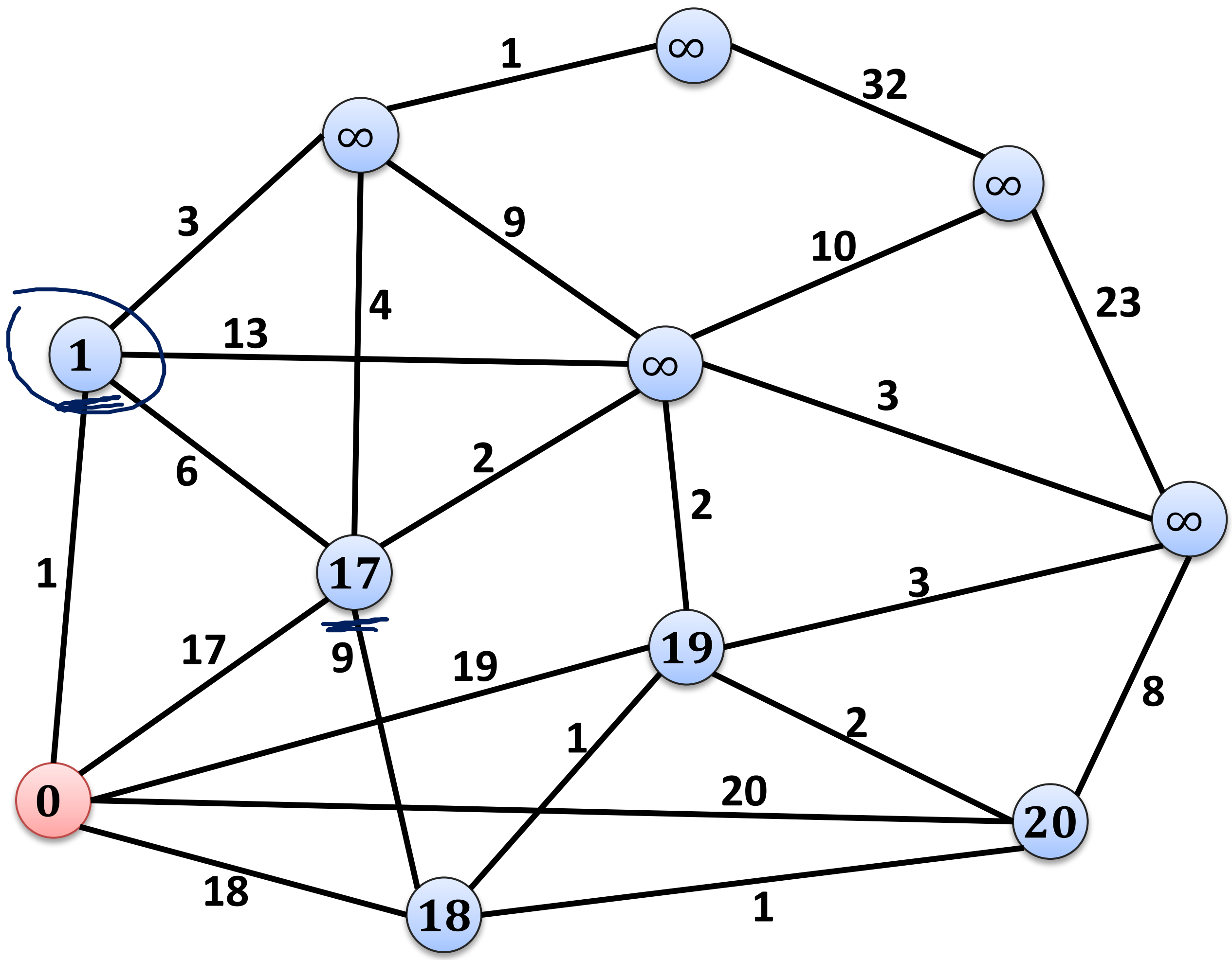
Dijkstra's Algorithm:

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

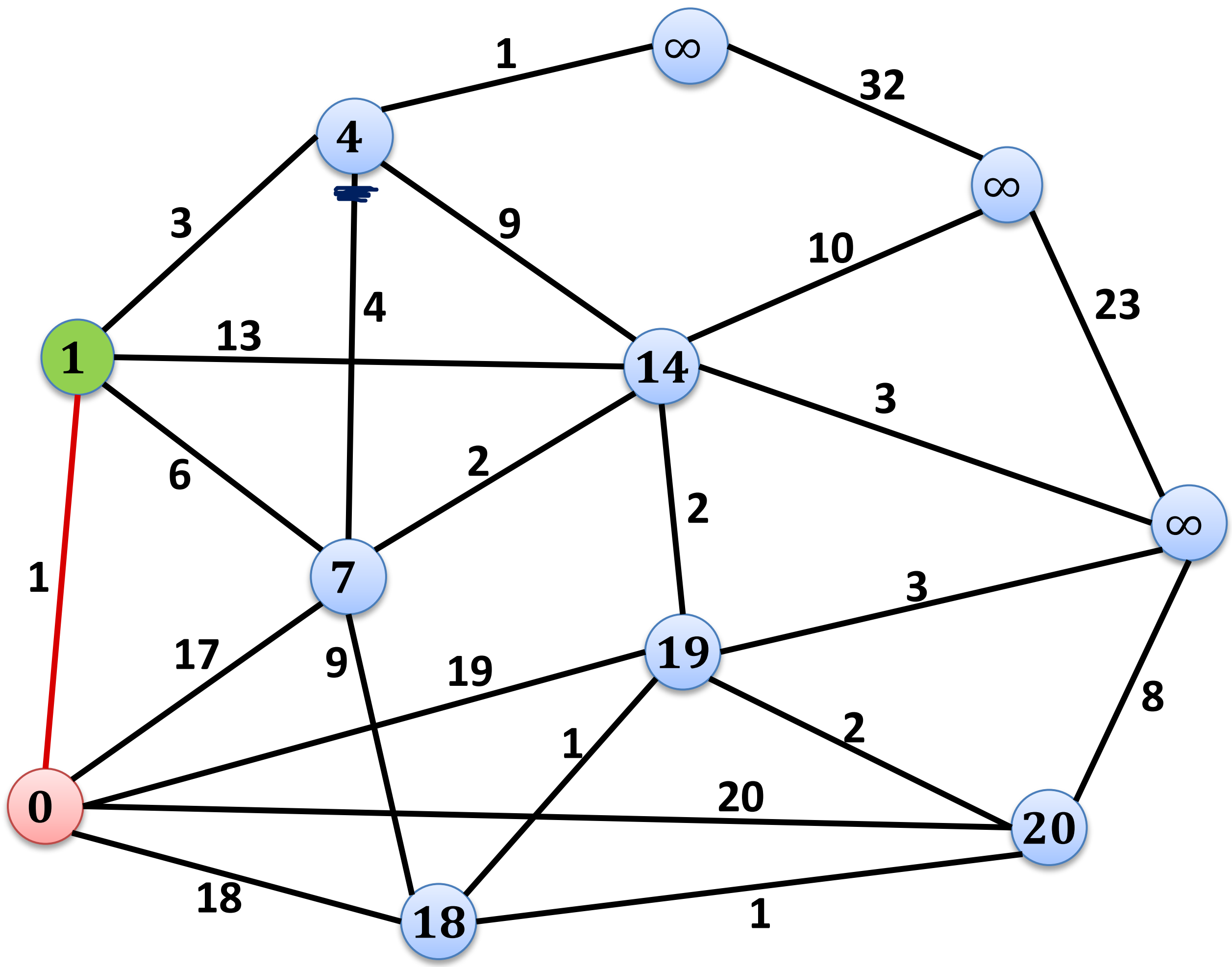
Example



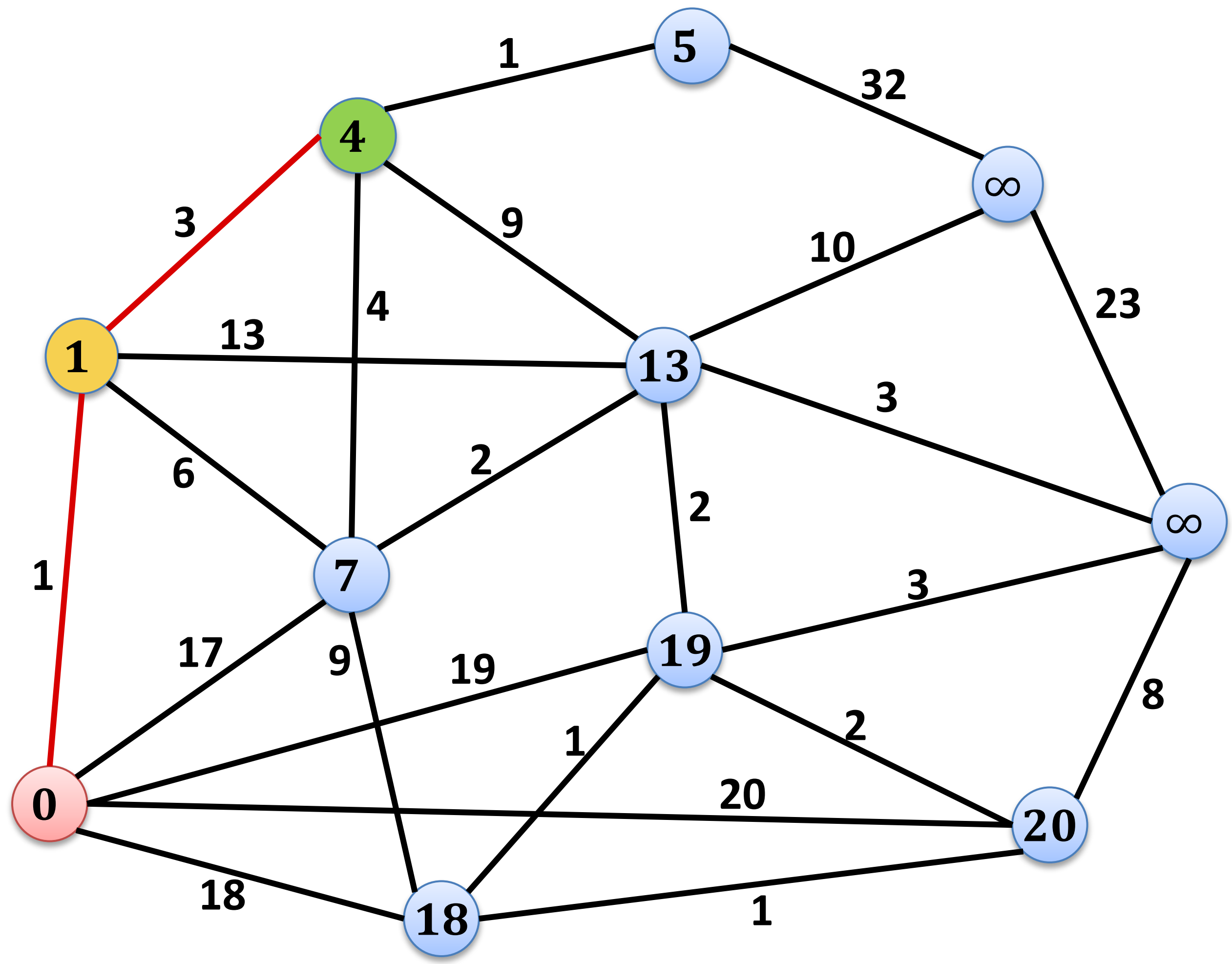
Example



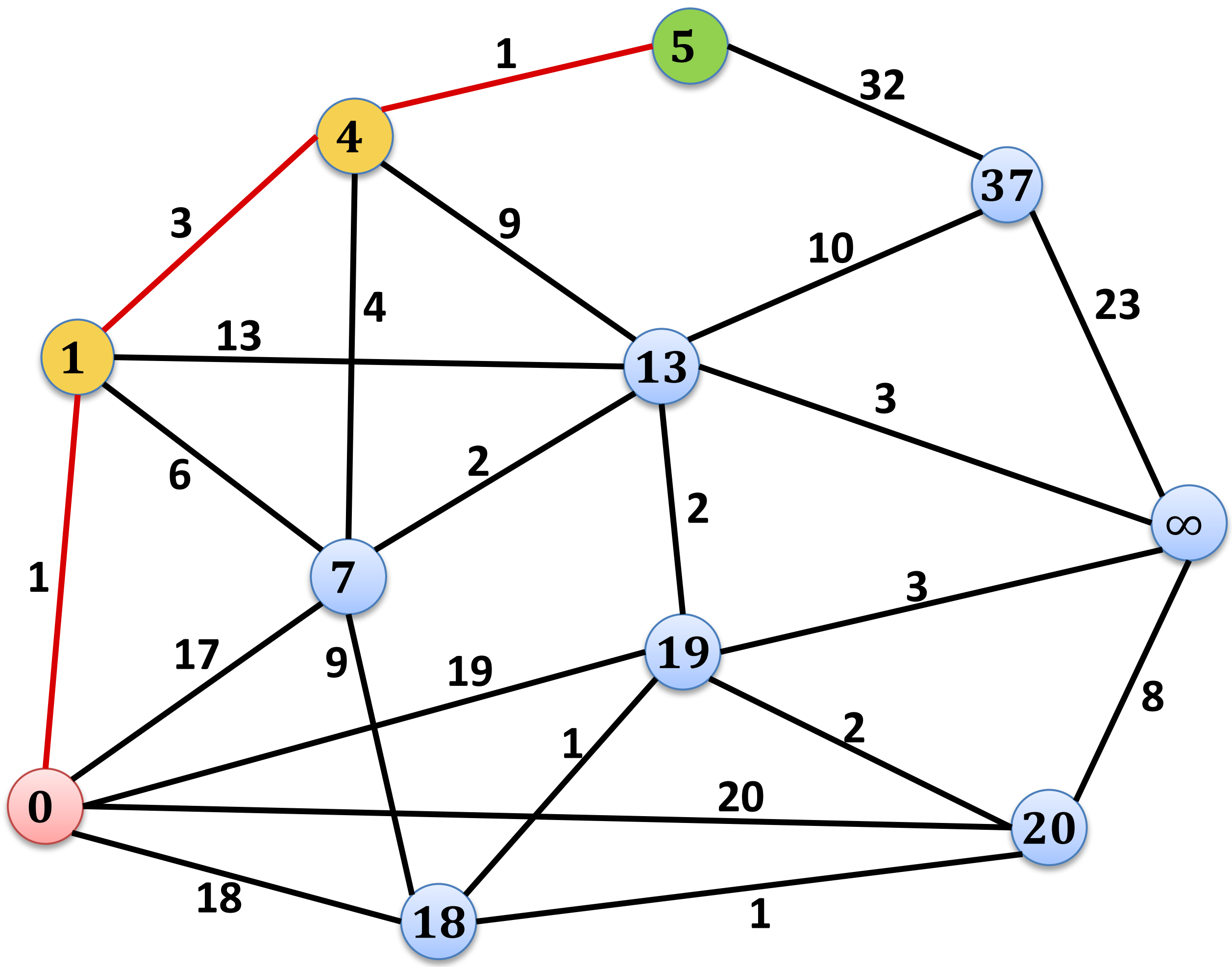
Example



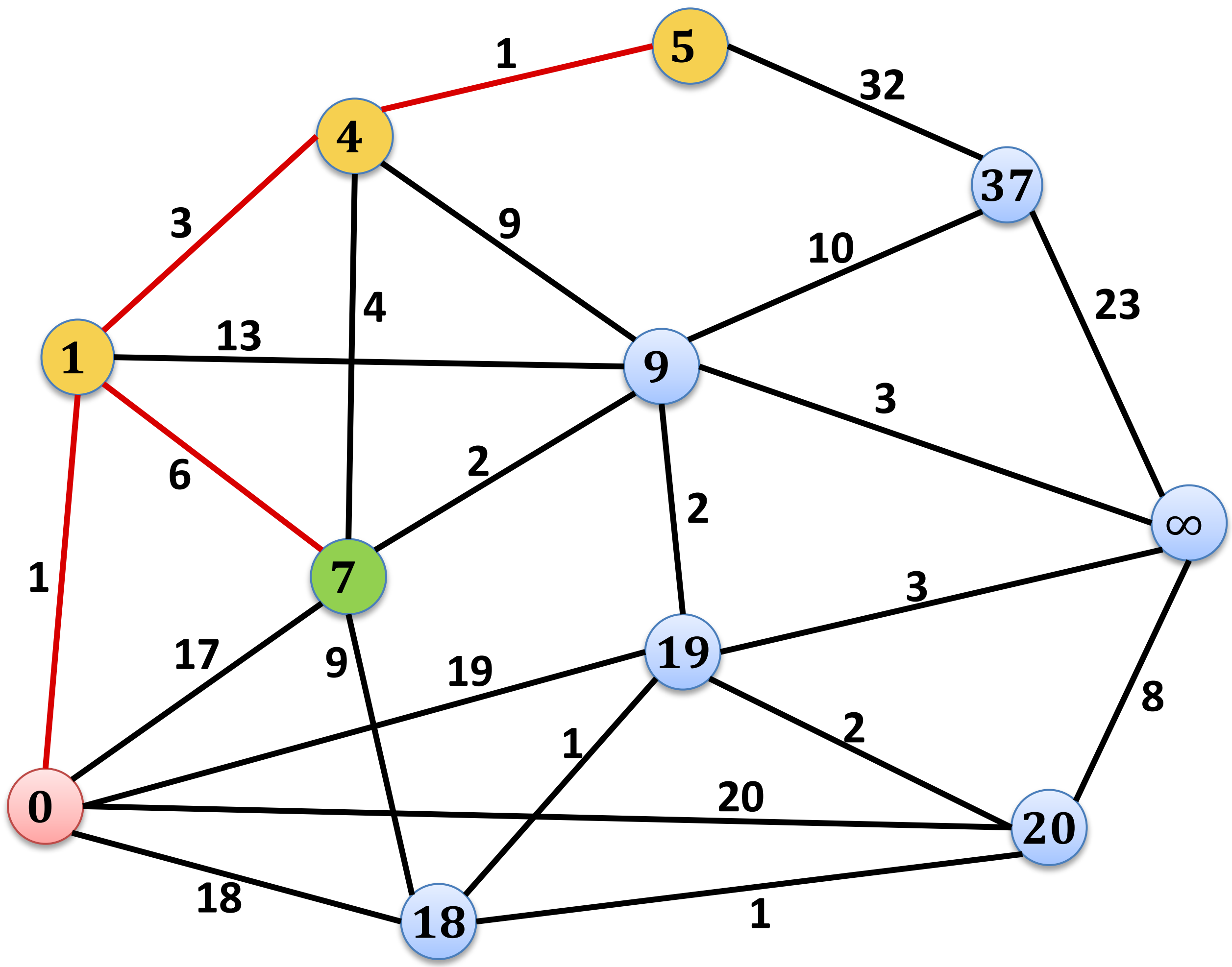
Example



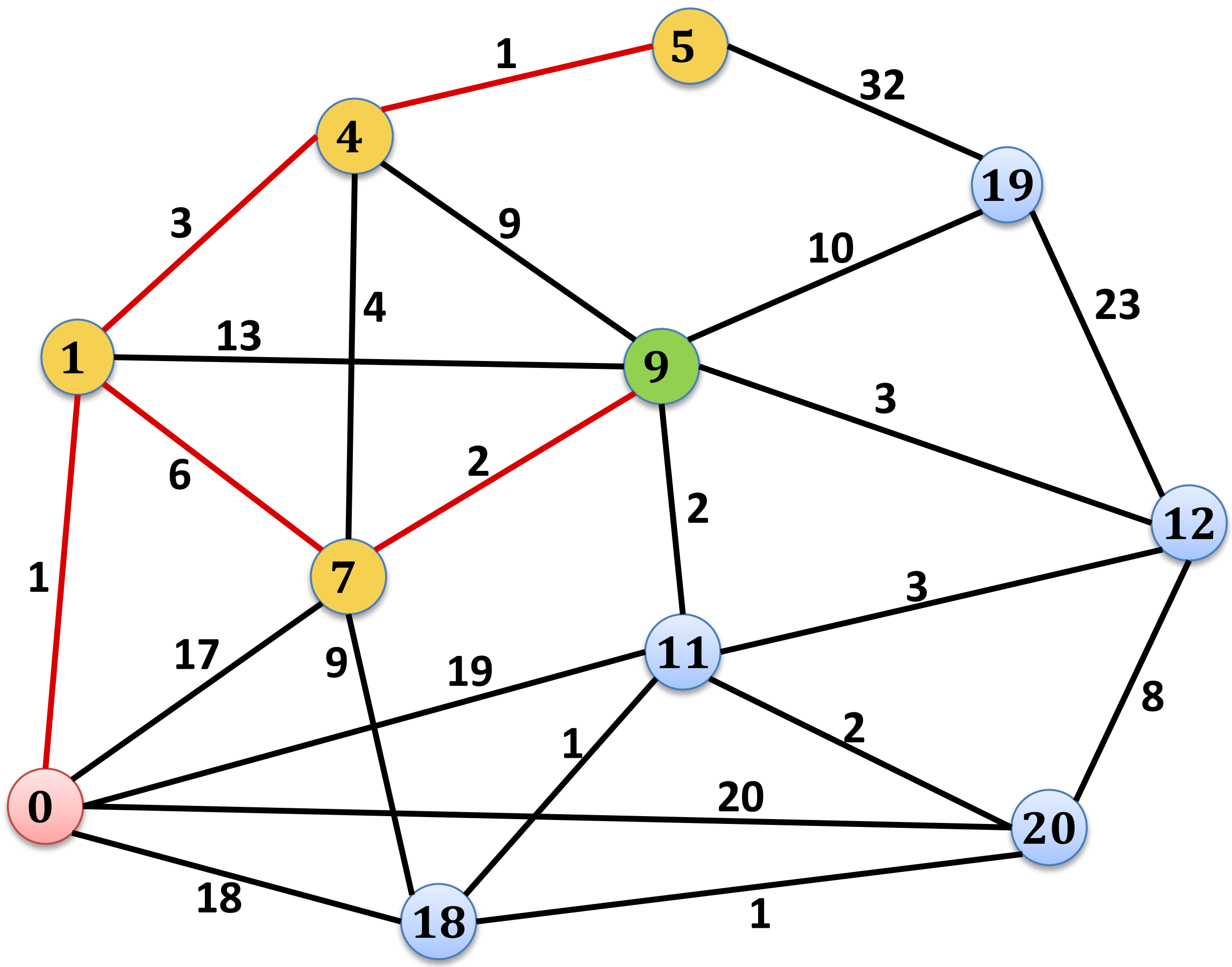
Example



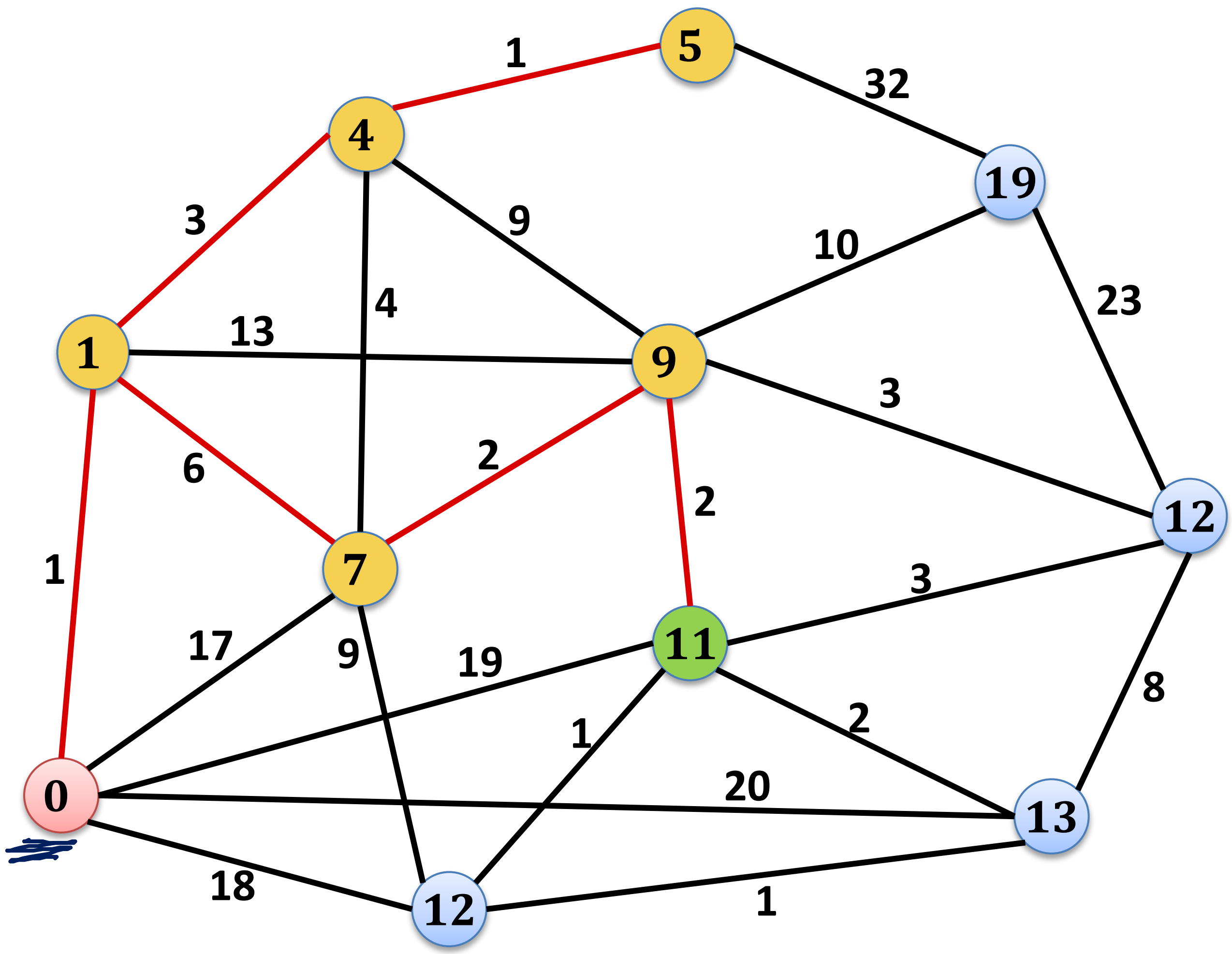
Example



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

   data structure with unmarked nodes
   add all nodes with dist. est.

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

   get node in DS with min $d(s,u)$

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

   potentially update dist. est. of neighbors
   decrease

5. mark node u

   delete u from DS

6. Until all nodes are marked

Priority Queue / Heap

priorities

- 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 $d(s,v)$ 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

   create empty PQ, insert all nodes

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

   get-min

4. mark node u

   delete-min

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

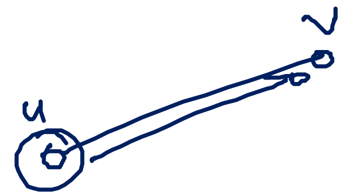
   decrease-key

6. Until all nodes are marked

Analysis

$|V| = n \quad |E| = m$

Number of priority queue operations for Dijkstra:

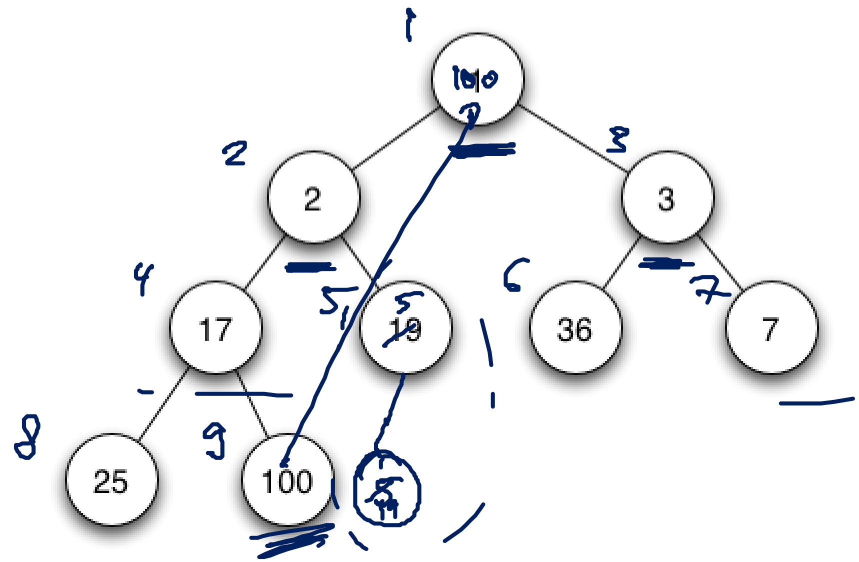


- **Initialize-Heap**: $\mathbf{1}$
- **Is-Empty**: $|\boldsymbol{V}|$
- **Insert**: $|\boldsymbol{V}|$
- **Get-Min**: $|\boldsymbol{V}|$
- **Delete-Min**: $|\boldsymbol{V}|$
- **Decrease-Key**: $\underline{|\boldsymbol{E}|}$
- **Merge**: $\mathbf{0}$

Priority Queue Implementation

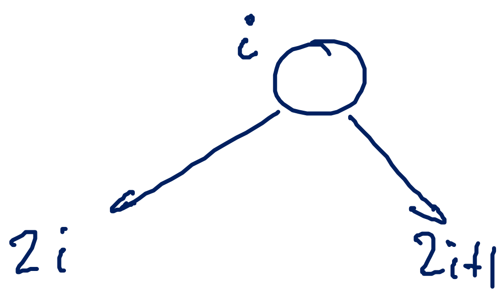
Implementation as min-heap:

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



insert(5)

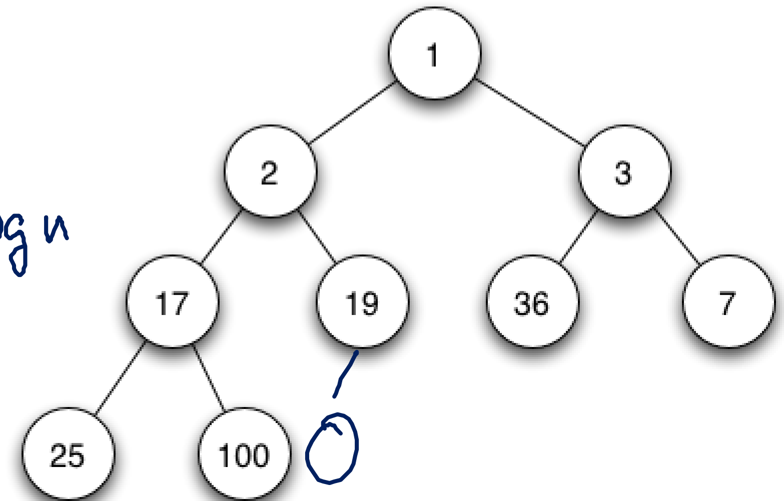
min-heap property



Priority Queue Implementation

Implementation as min-heap:

log n



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

- 1 • **Initialize-Heap**: $\boldsymbol{O(1)}$
- n • **Is-Empty**: $\boldsymbol{O(1)}$
- n • **Insert**: $\boldsymbol{O(\log n)}$
- n • **Get-Min**: $\boldsymbol{O(1)}$
- n • **Delete-Min**: $\boldsymbol{O(\log n)}$
- m • **Decrease-Key**: $\boldsymbol{O(\log n)}$ $O(m \cdot \log n)$
- **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

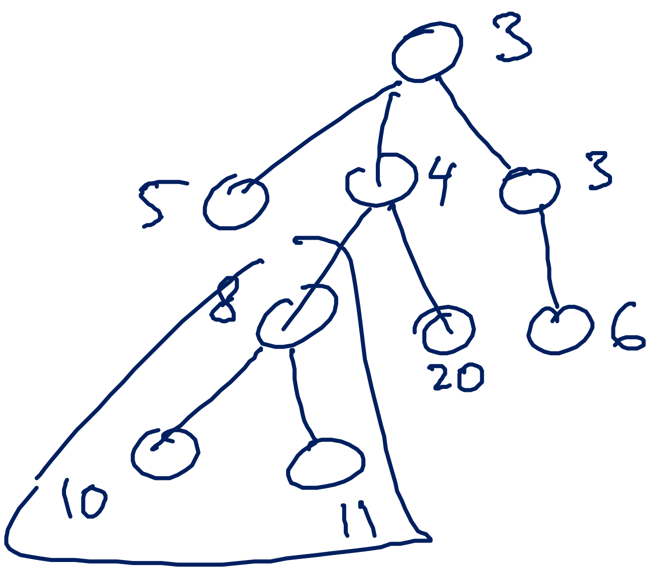
Binomial Heaps

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:

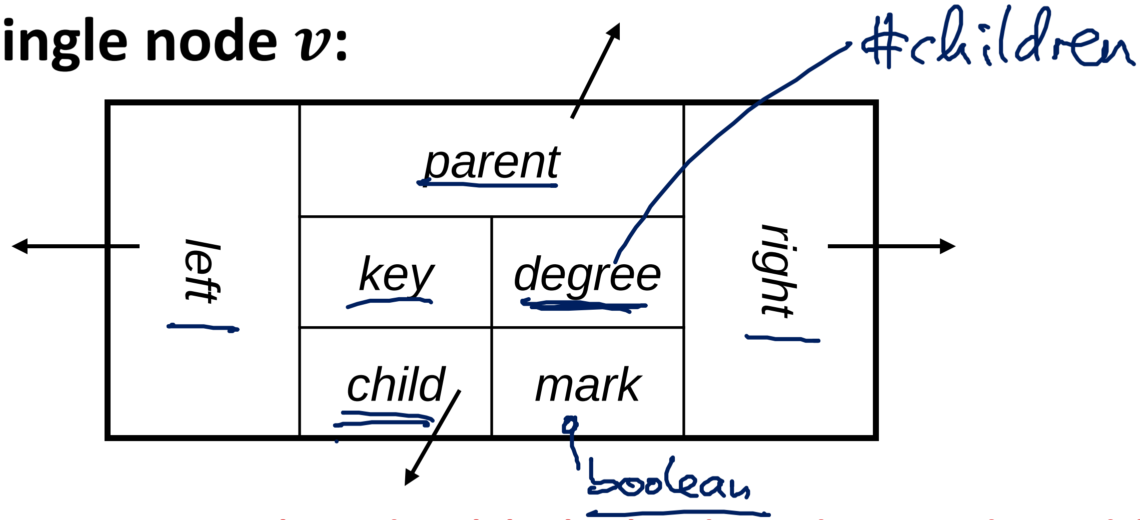
- $H.min$: root of the tree containing the (a) minimum key
- $H.rootlist$: circular, doubly linked, unordered list containing the roots of all trees
- $H.size$: number of nodes currently in H

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

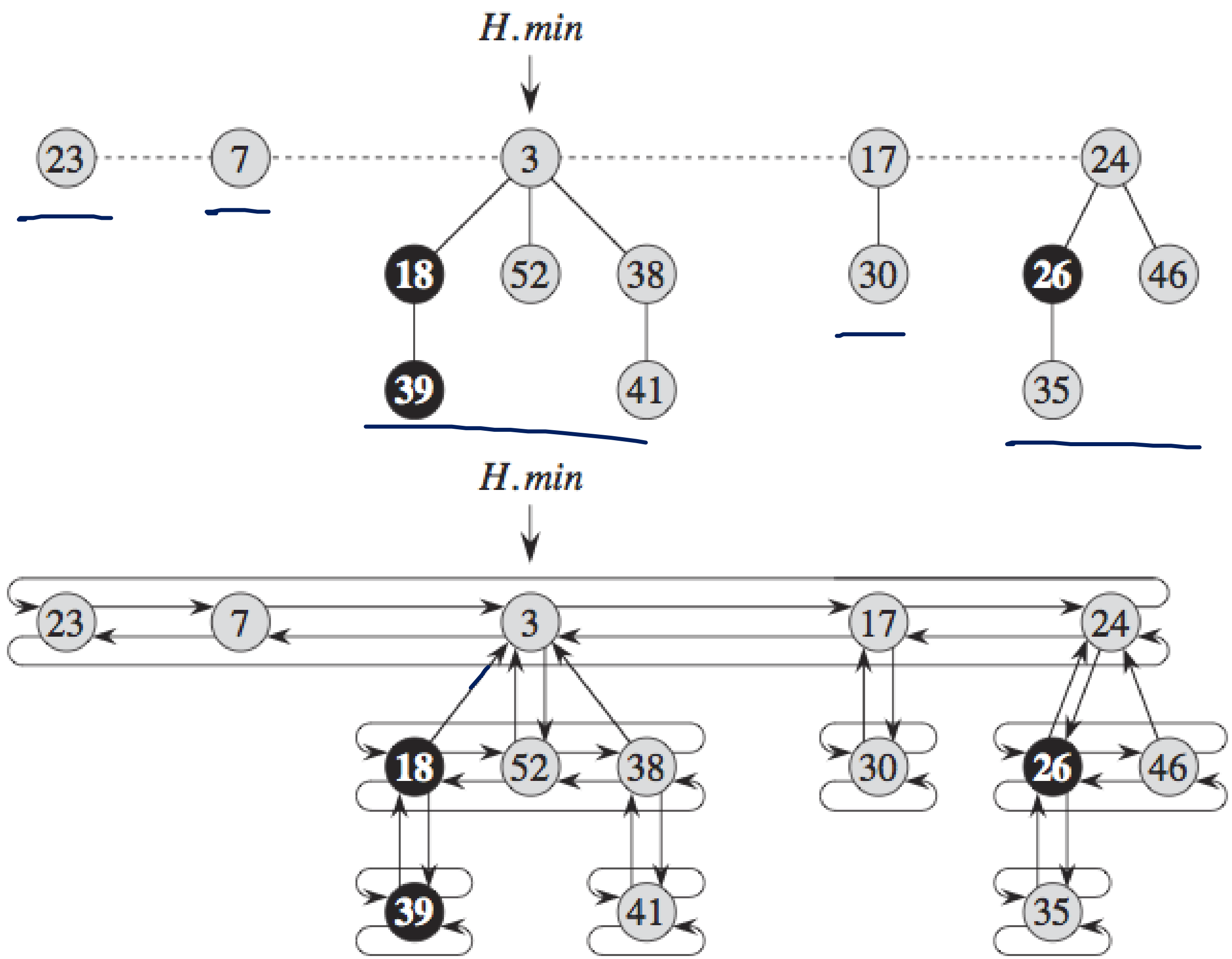


Figure: Cormen et al., Introduction to Algorithms

Simple (Lazy) Operations

Initialize-Heap H:

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

delete-min

decrease-key

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 **false**
- merge heaps H and H'

Get minimum element of H:

- return $H.min$