



# **Chapter 5**

# **Data Structures**

**Algorithm Theory**  
**WS 2018/19**

**Fabian Kuhn**

# Examples

## Dictionary:

- Operations:  $\text{insert}(key, value)$ ,  $\text{delete}(key)$ ,  $\text{find}(key)$
- 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

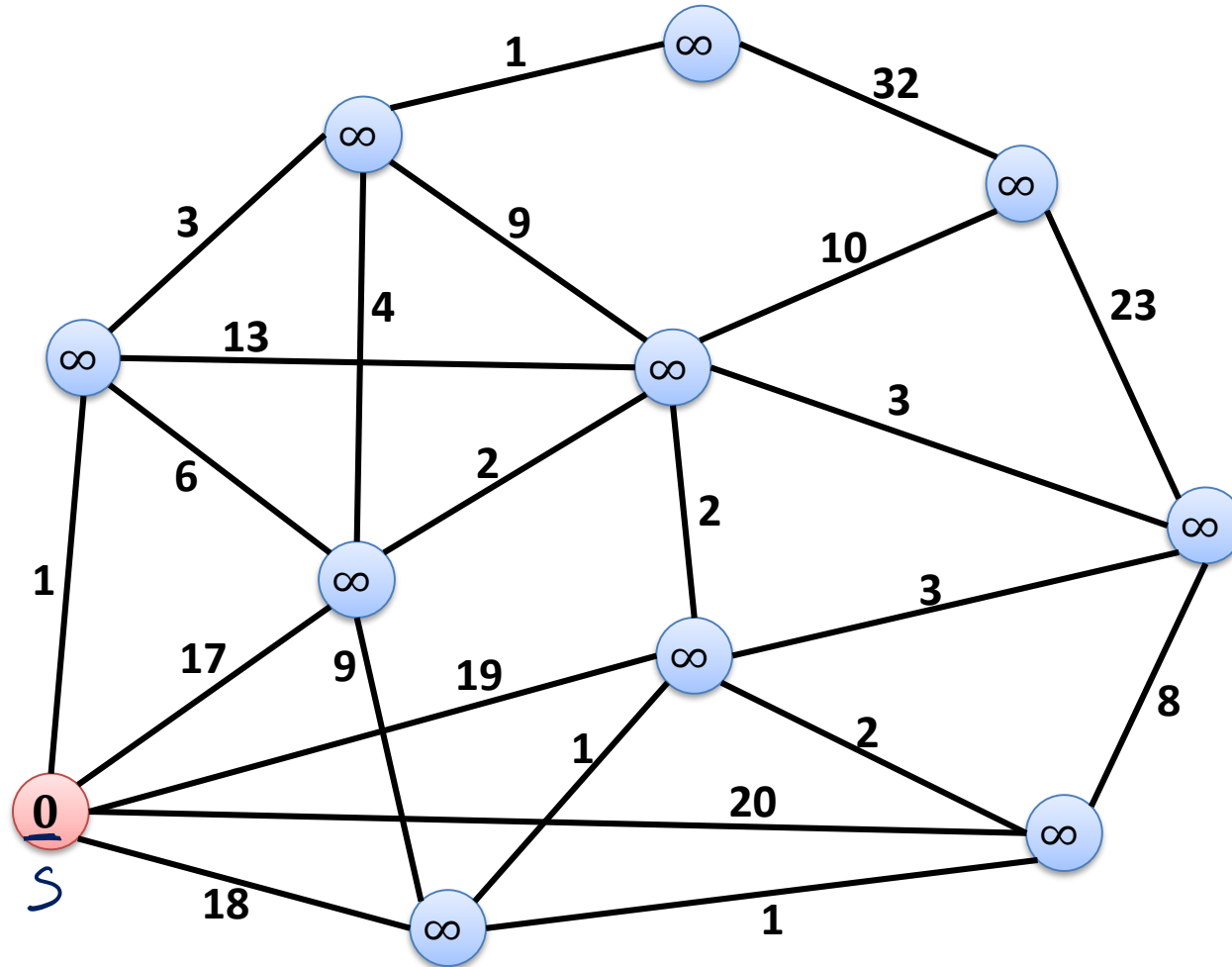
## Single-Source Shortest Path Problem:

- **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  $v \in V$

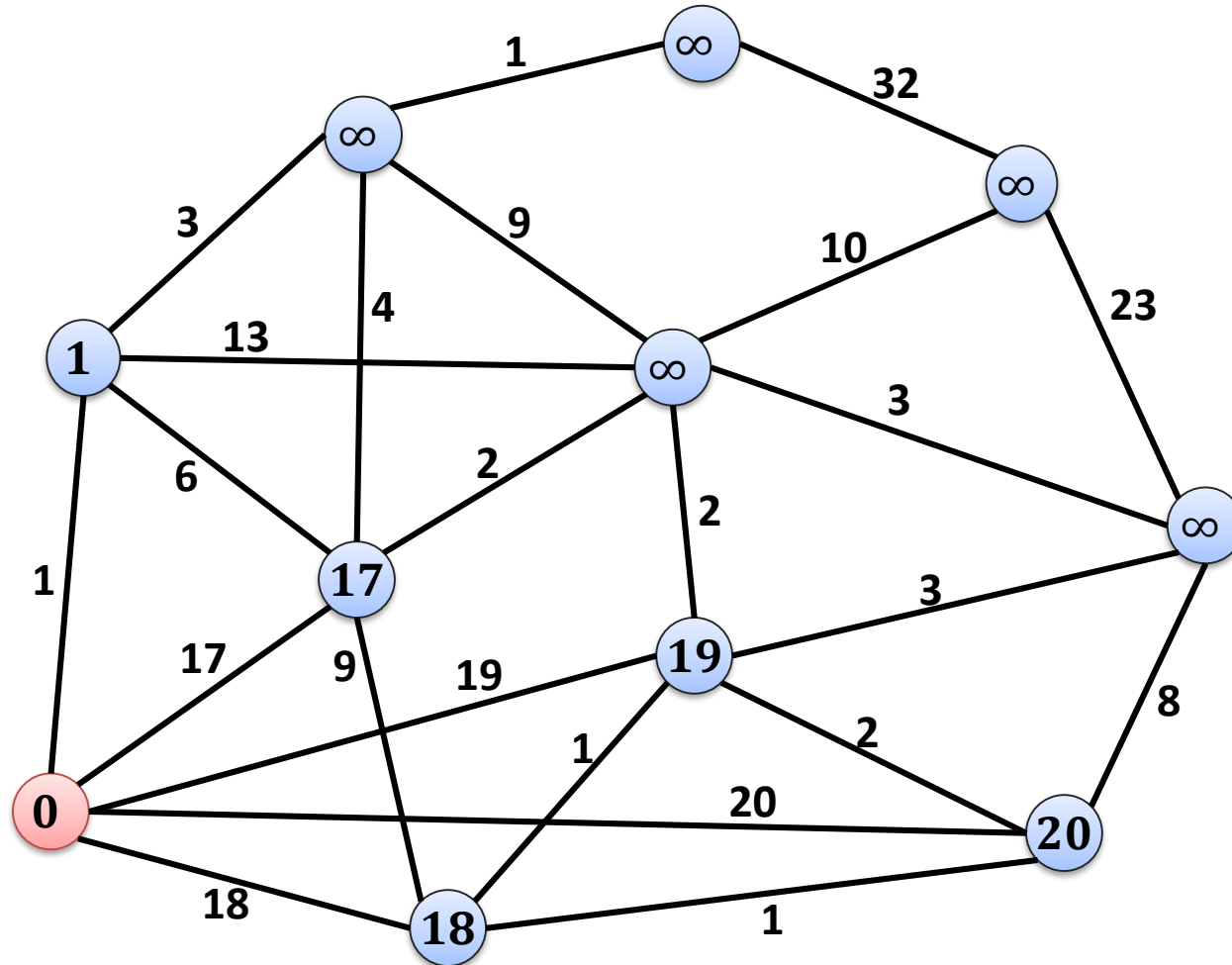
## Dijkstra's Algorithm:

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  $e = \{u, v\} \in E$ ,  $d(s, v) = \min\{\underline{d(s, v)}, \underline{d(s, u) + w(e)}\}$
5. mark node  $u$
6. Until all nodes are marked

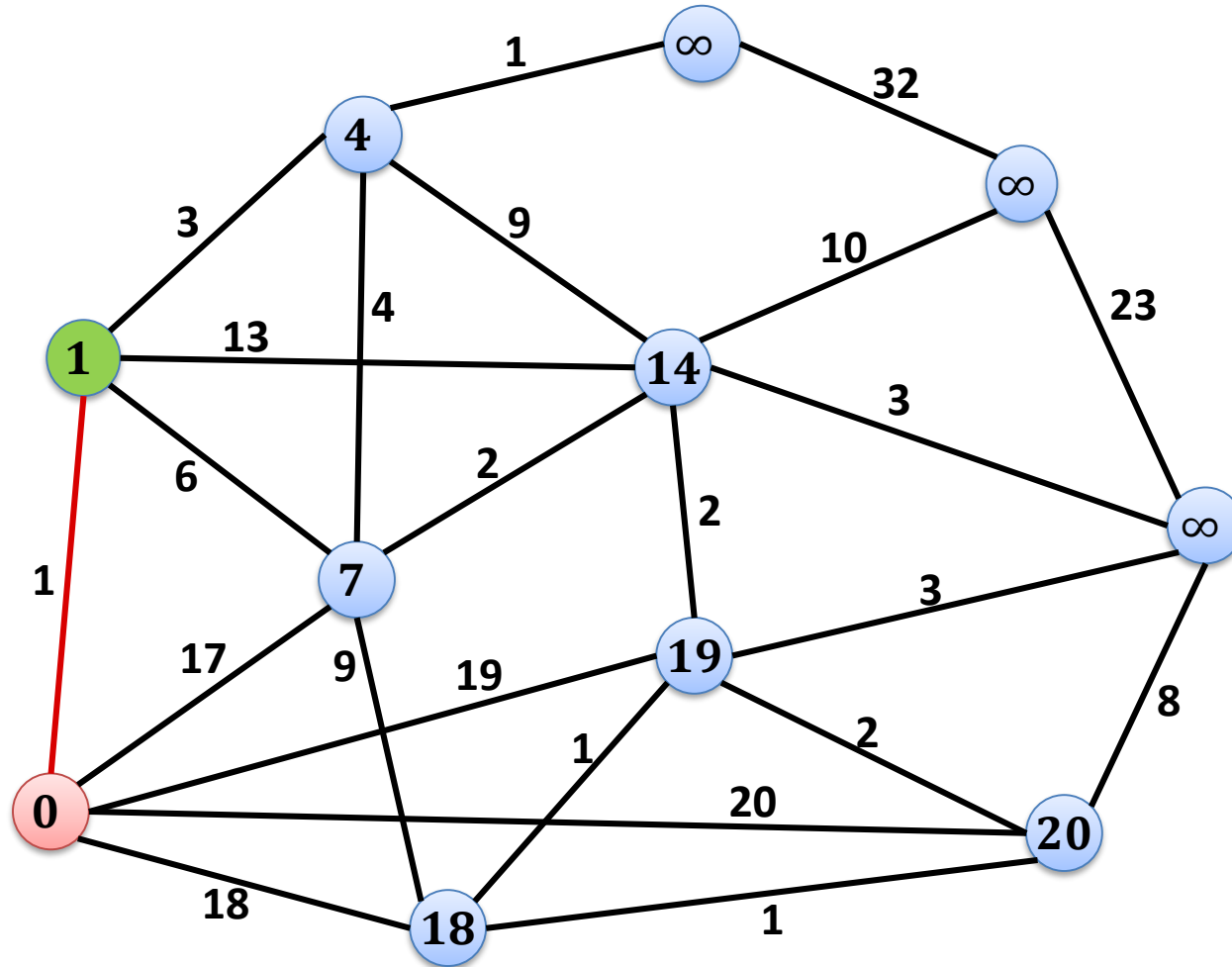
# Example



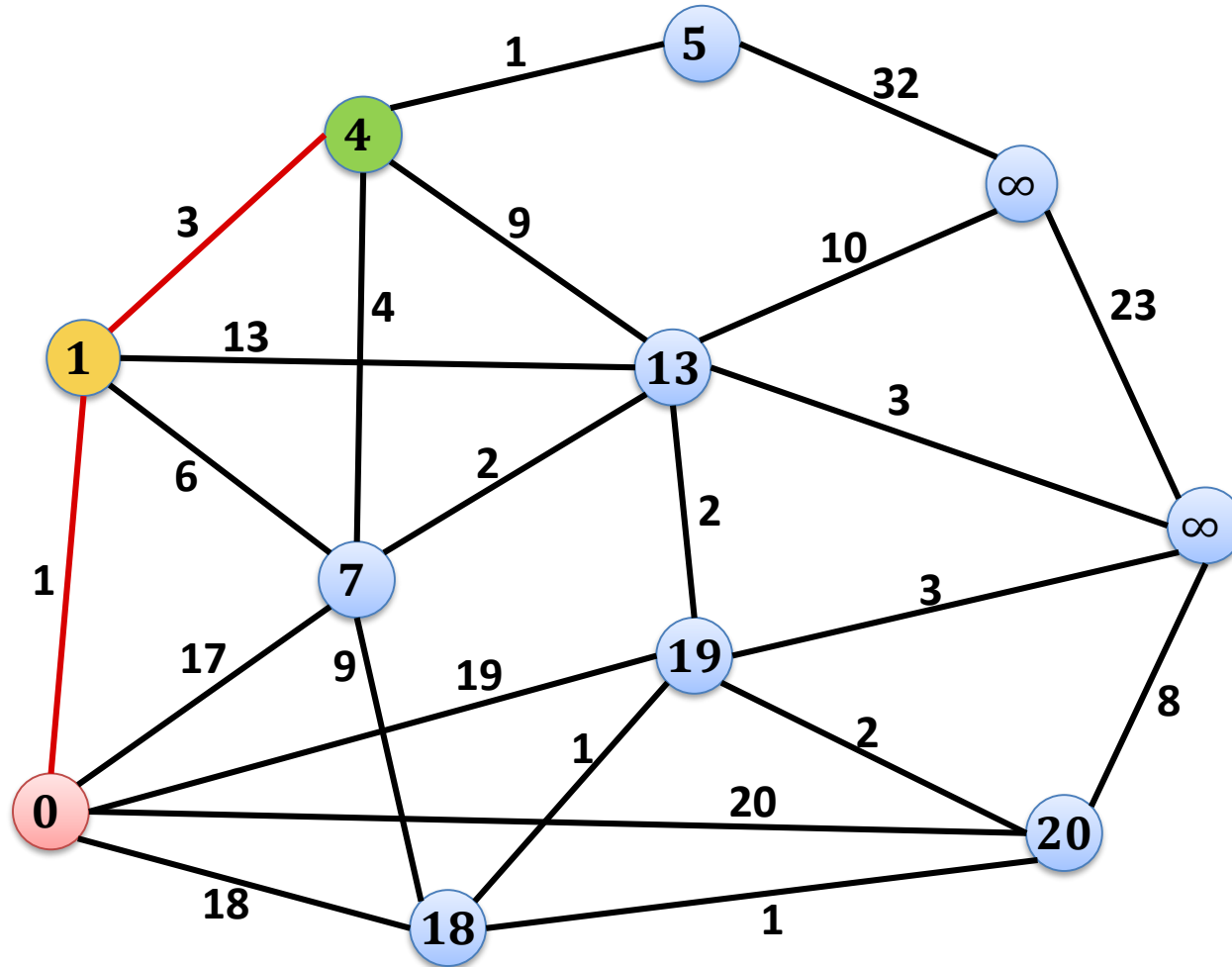
# Example



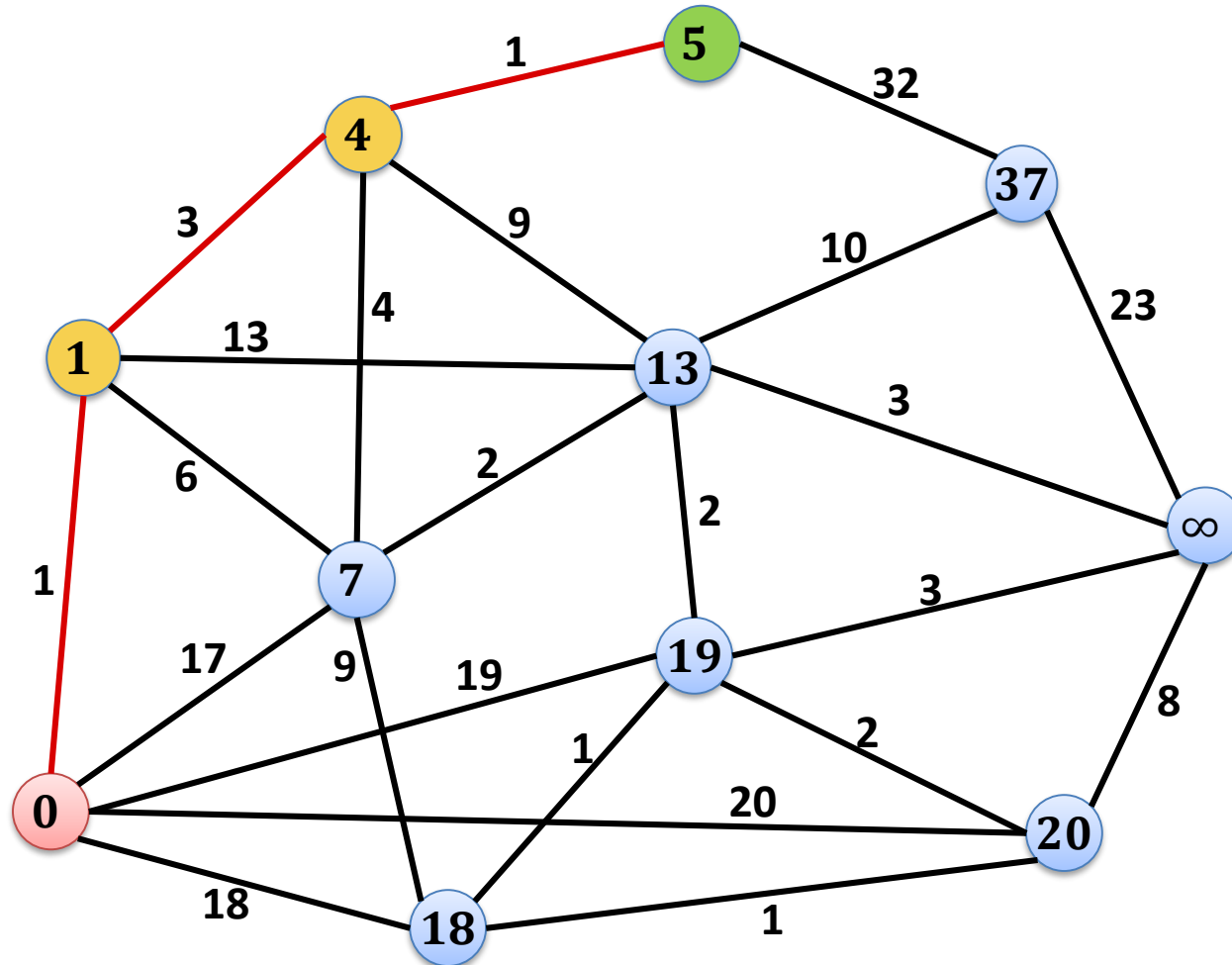
# Example



# Example

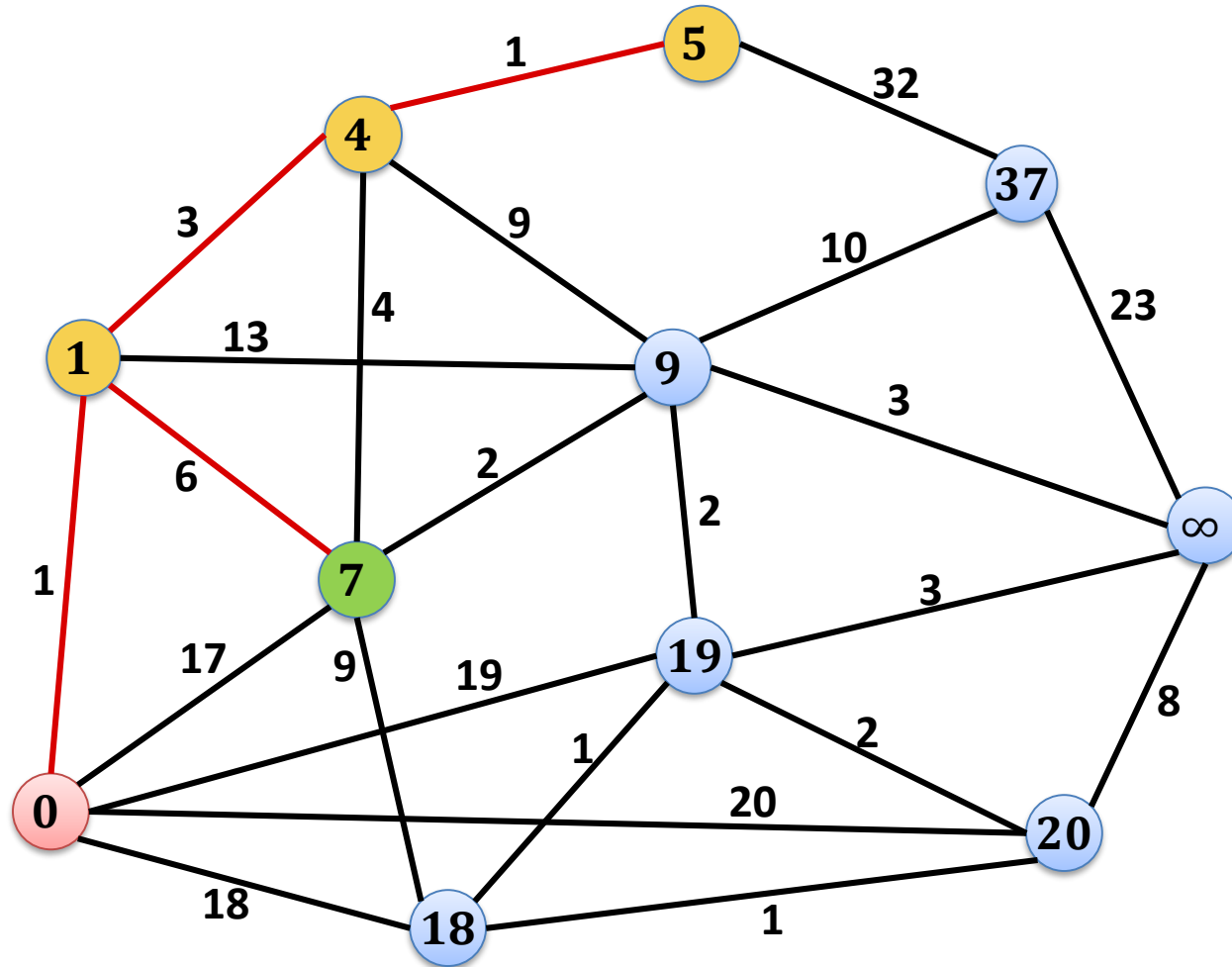


# Example

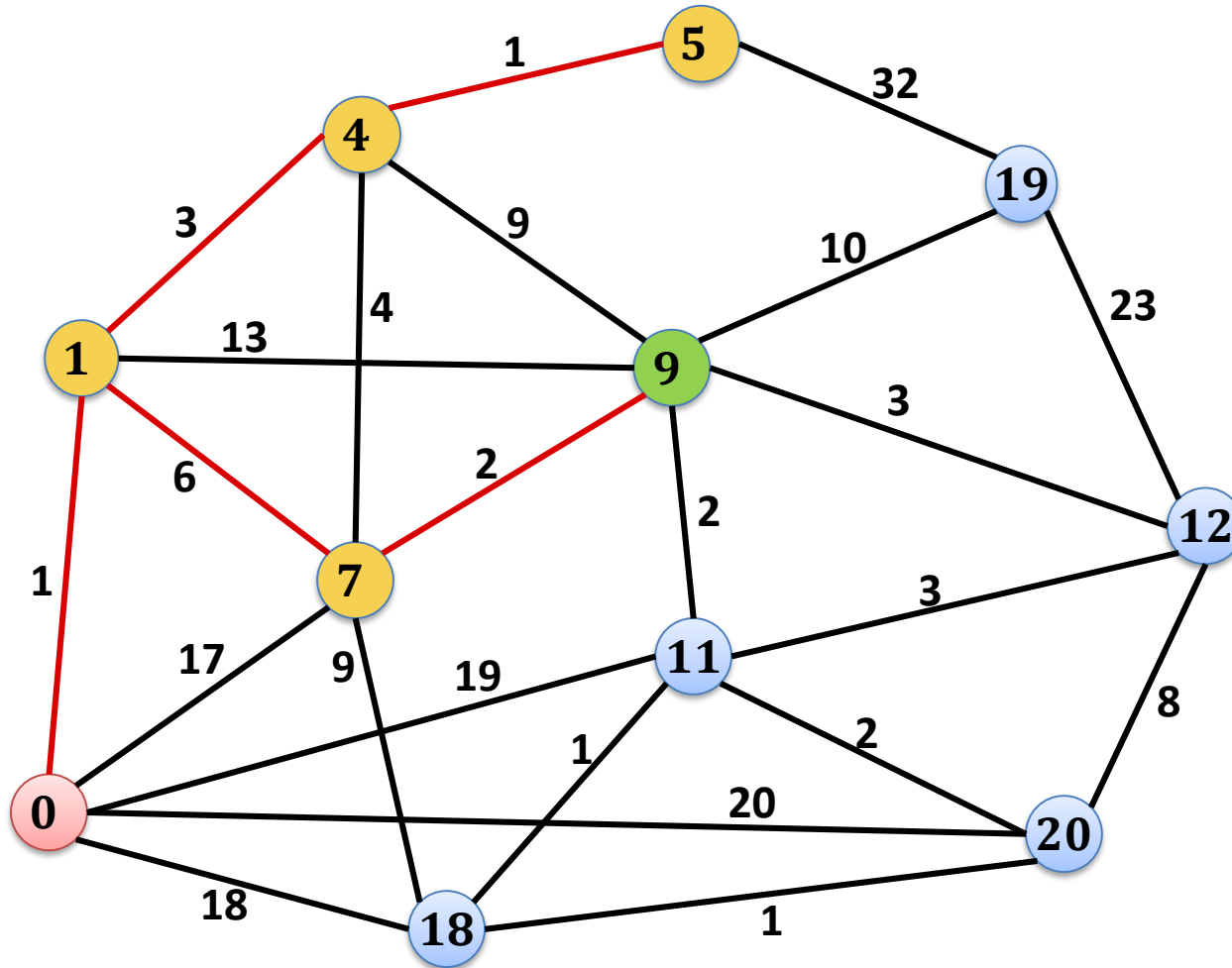




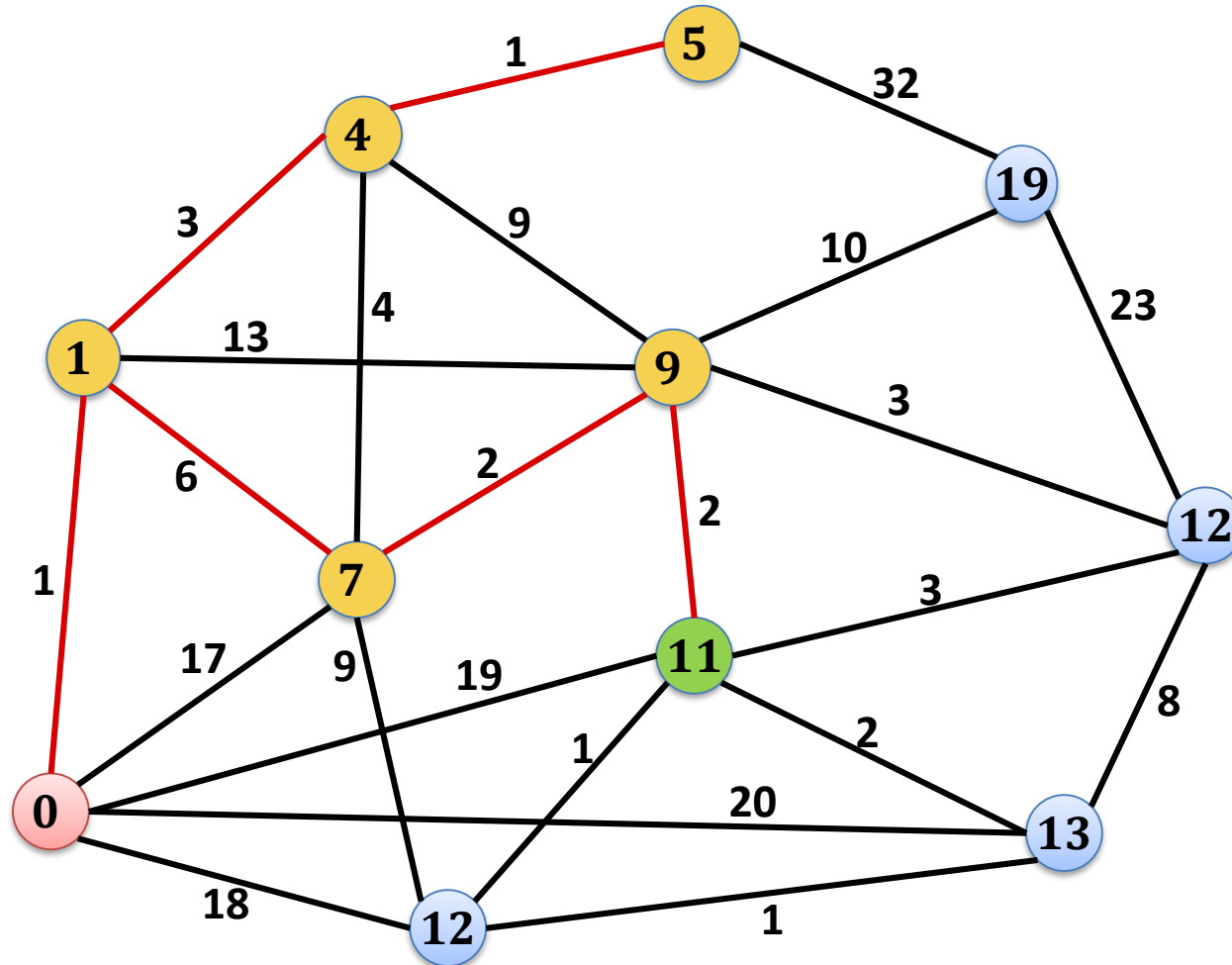
# Example



# 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 struct. with unmarked nodes*  
*add all nodes with their initial dist. estimate*
3. Get unmarked node  $u$  which minimizes  $d(s, u)$ :  
*get node from D.S. 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. estimates of all neighbors*  
*decrease*
5. mark node  $u$   
*delete  $u$  from D.S.*
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 } *consistent*
- **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 pr. queue  $Q$ , insert all nodes*

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

*get-min*

4. mark node  $u$

*delete-min*

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

*for all neighbors of  $u$ : potentially call decrease-key*

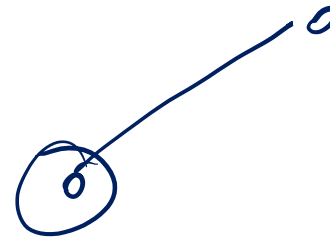
6. Until all nodes are marked (until  $Q$  is empty)

# Analysis

graph  $G = (V, E)$   
 $|V| = n$   $|E| = m$

Number of priority queue operations for Dijkstra:

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



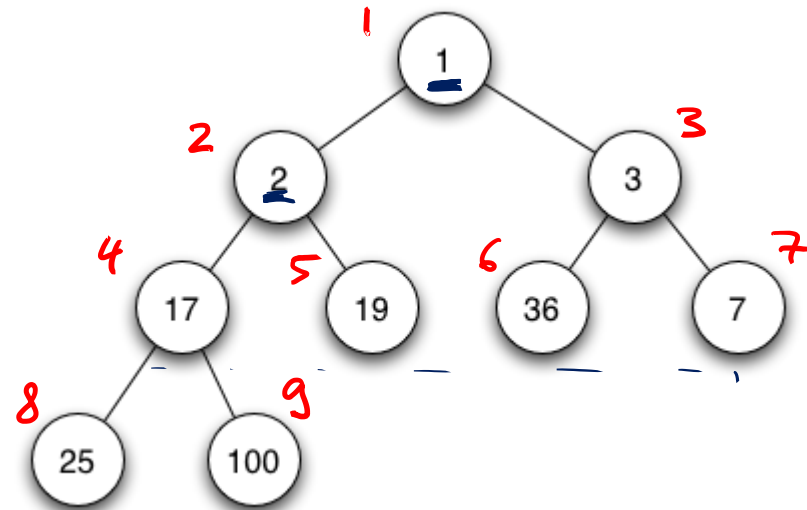
# Priority Queue Implementation



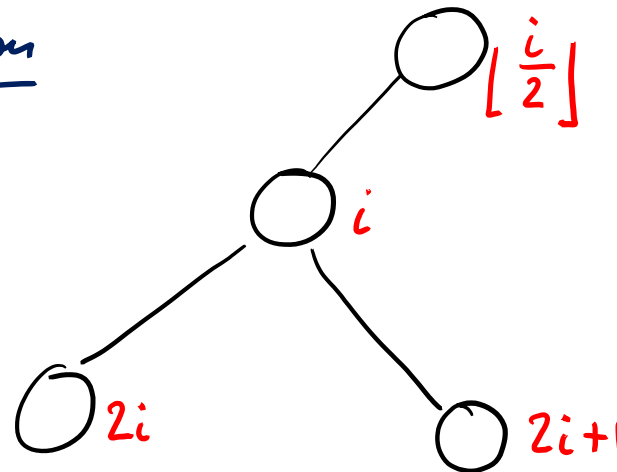
Implementation as min-heap:

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

*min-heap property*



array implementation



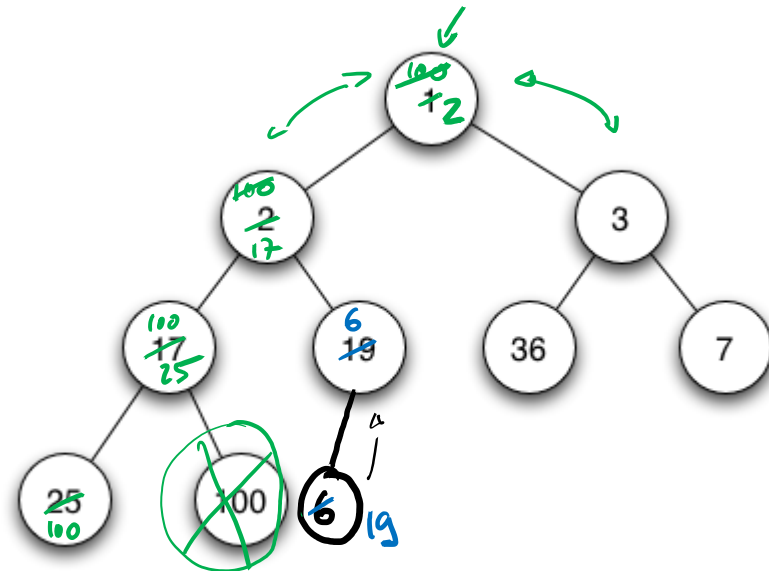


# Priority Queue Implementation

Implementation as min-heap:

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

- Initialize-Heap:  $O(1)$
- Is-Empty:  $O(1)$
- VI • Insert:  $O(\log n)$
- VI • Get-Min:  $O(1)$
- VI • Delete-Min:  $O(\log n)$
- VI • Decrease-Key:  $O(\log n)$



insert(6)

Dijkstra:

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

- Merge (heaps of size  $m$  and  $n$ ,  $m \leq n$ ):  $O(m \log n)$

# 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?

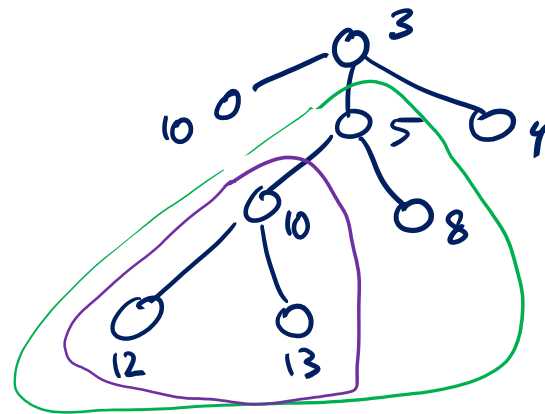
# Fibonacci Heaps

## Structure:

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

## Min-Heap Property:

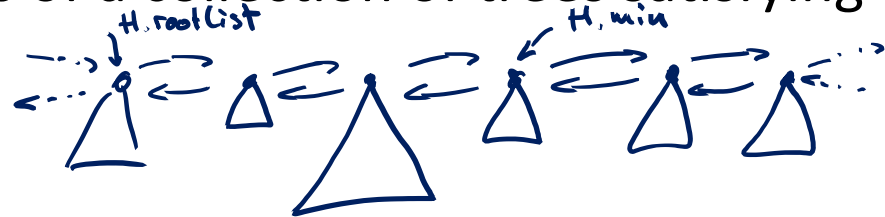
Key of a node  $v \leq$  keys of all nodes in any sub-tree of  $v$



# Fibonacci Heaps

## Structure:

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



## Variables:

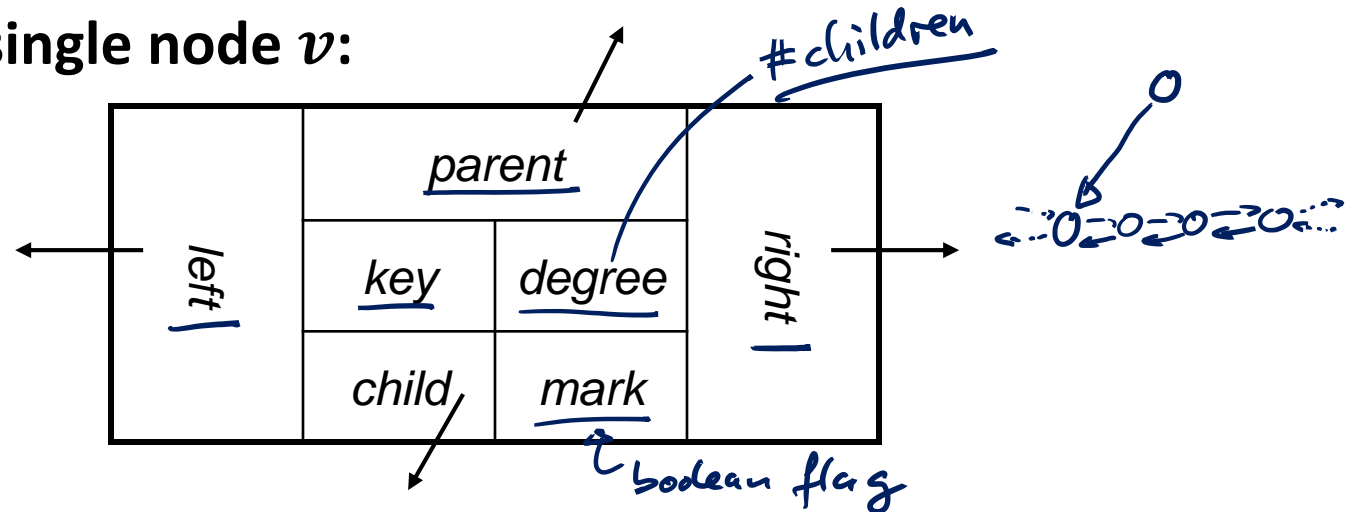
- $H.min$ : root of the tree containing the (a) minimum key
- $H.rootlist$ : circular, doubly linked, unordered list containing the roots of all trees
- $H.size$ : number of nodes currently in  $H$

## 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

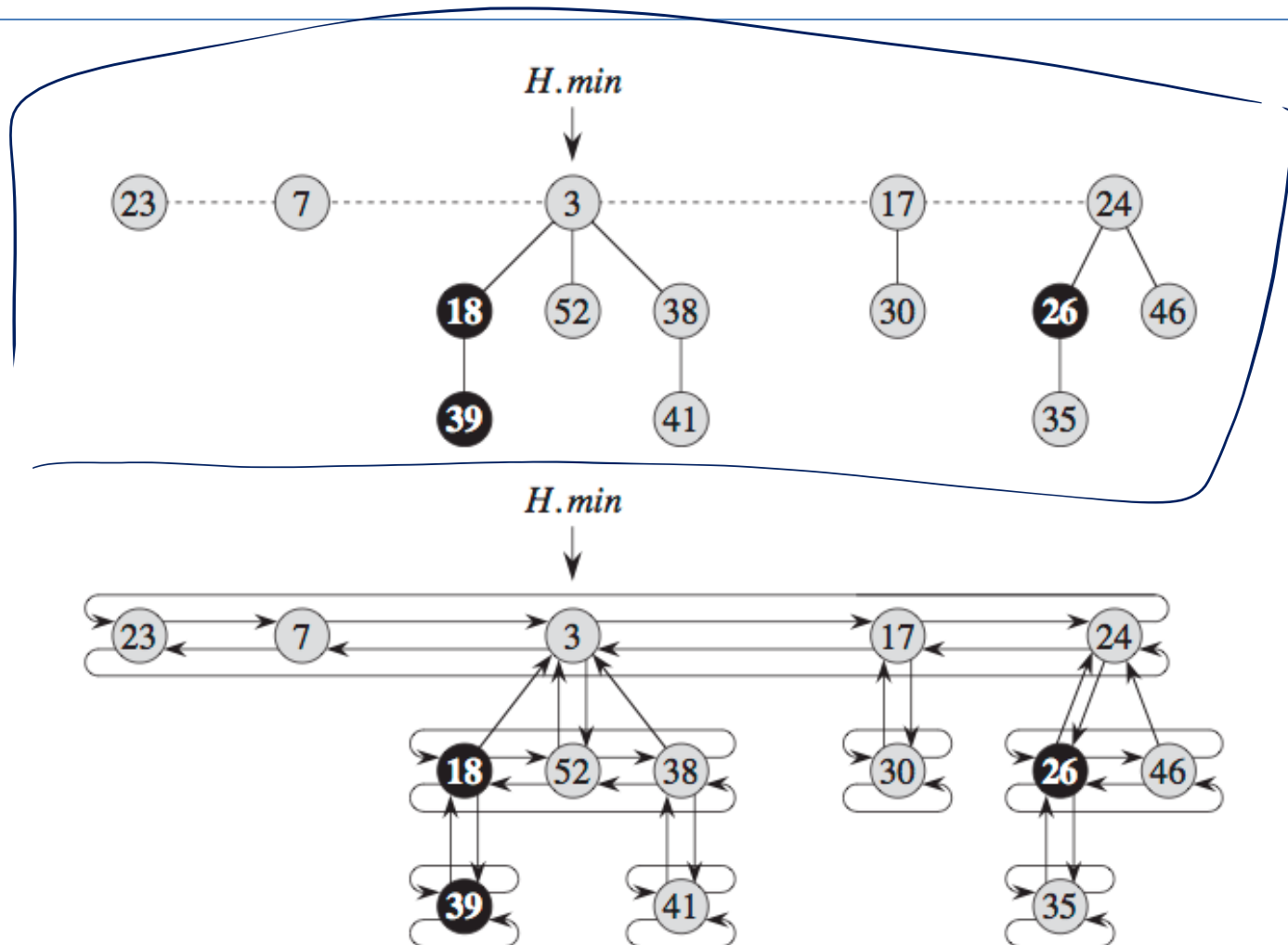


Figure: Cormen et al., Introduction to Algorithms

# 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  $e \rightarrow H'$ 
  - mark of root node is set to **false**
- merge heaps  $H$  and  $H'$

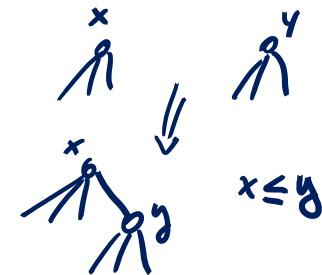
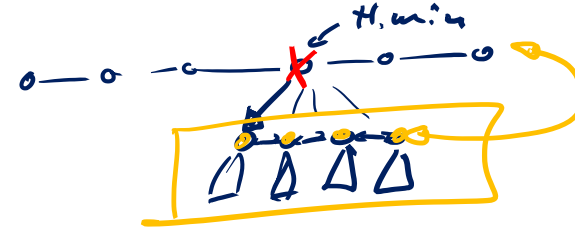
## Get minimum element of $H$ :

- 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.  $H.Consolidate();$



// 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  $D(n)$

$$D(n) = O(\log n)$$

# Merging Two Trees

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

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

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

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

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

- $(T'.mark = \mathbf{false})$

