Chapter 5
Data Structures

Algorithm Theory
WS 2019/20

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

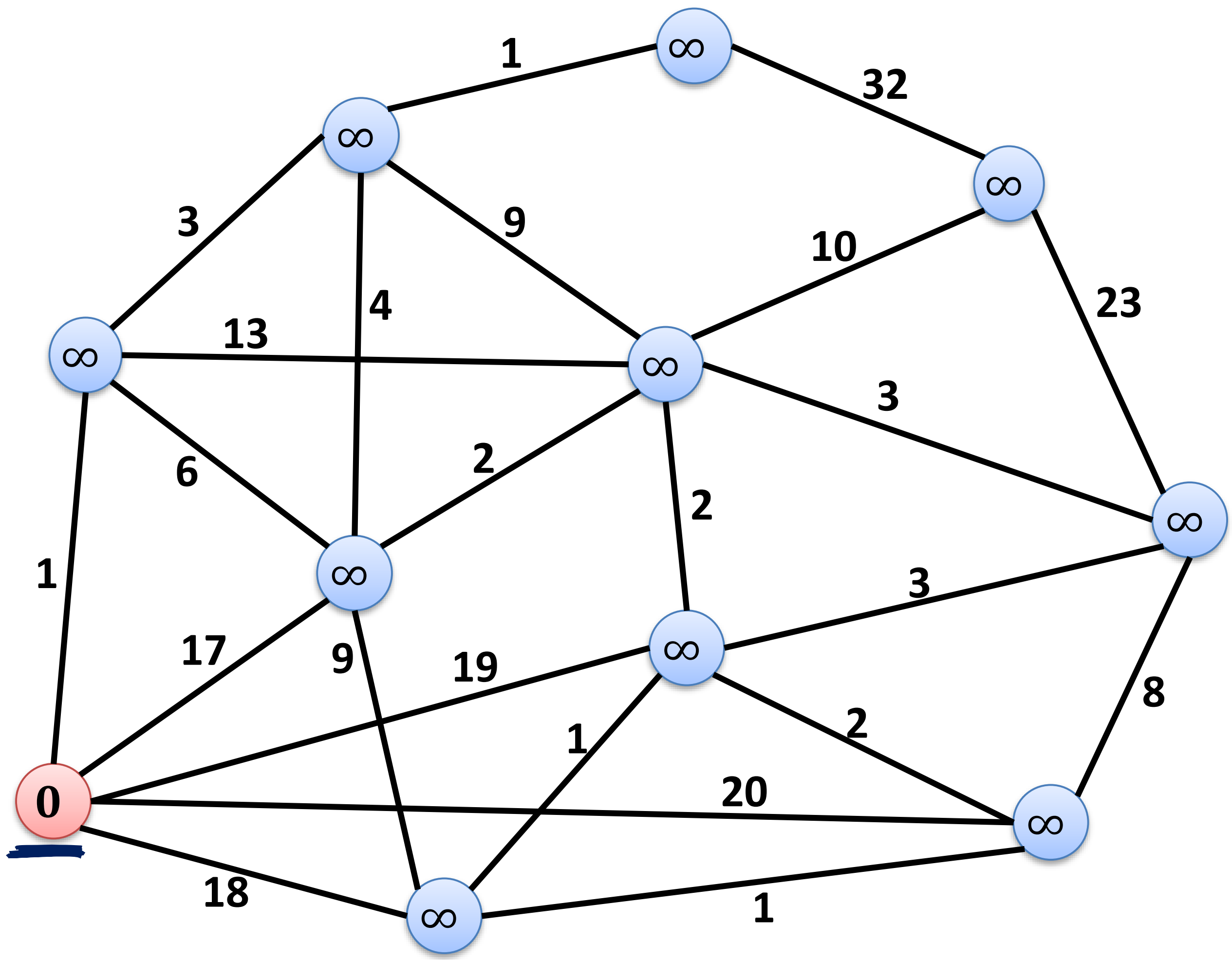
Single-Source Shortest Path Problem:

- **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 $v \in V$

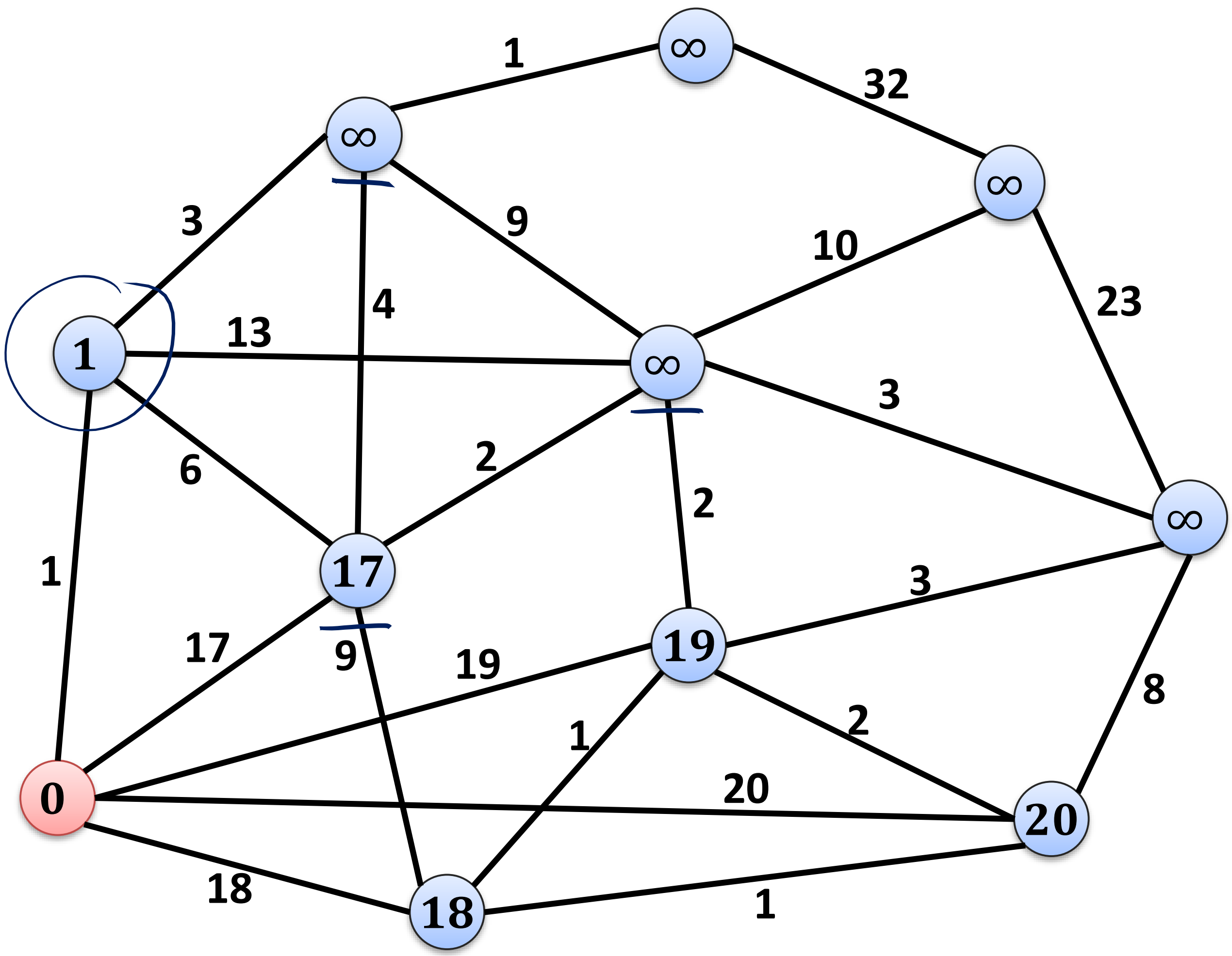
Dijkstra's Algorithm:

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

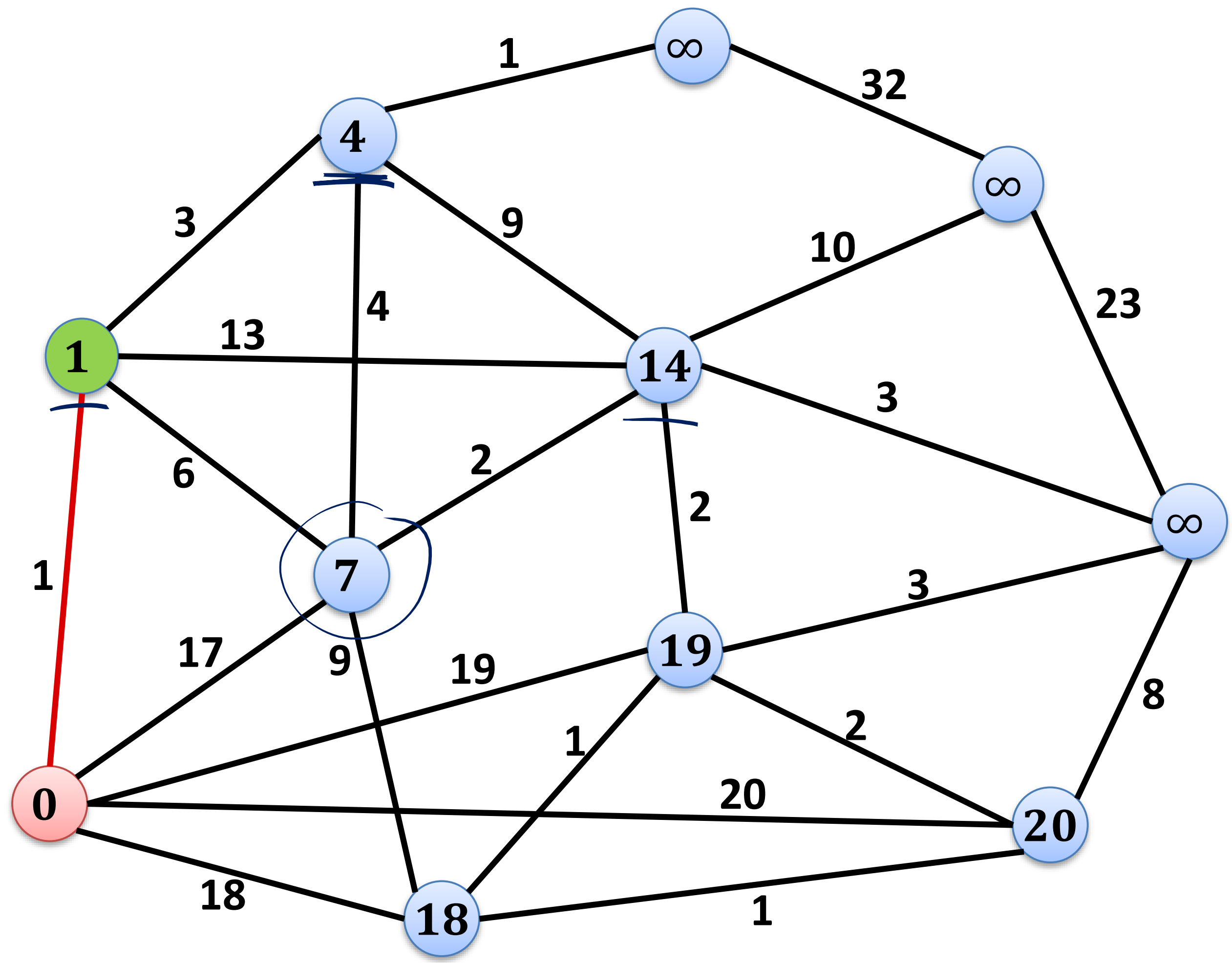
Example



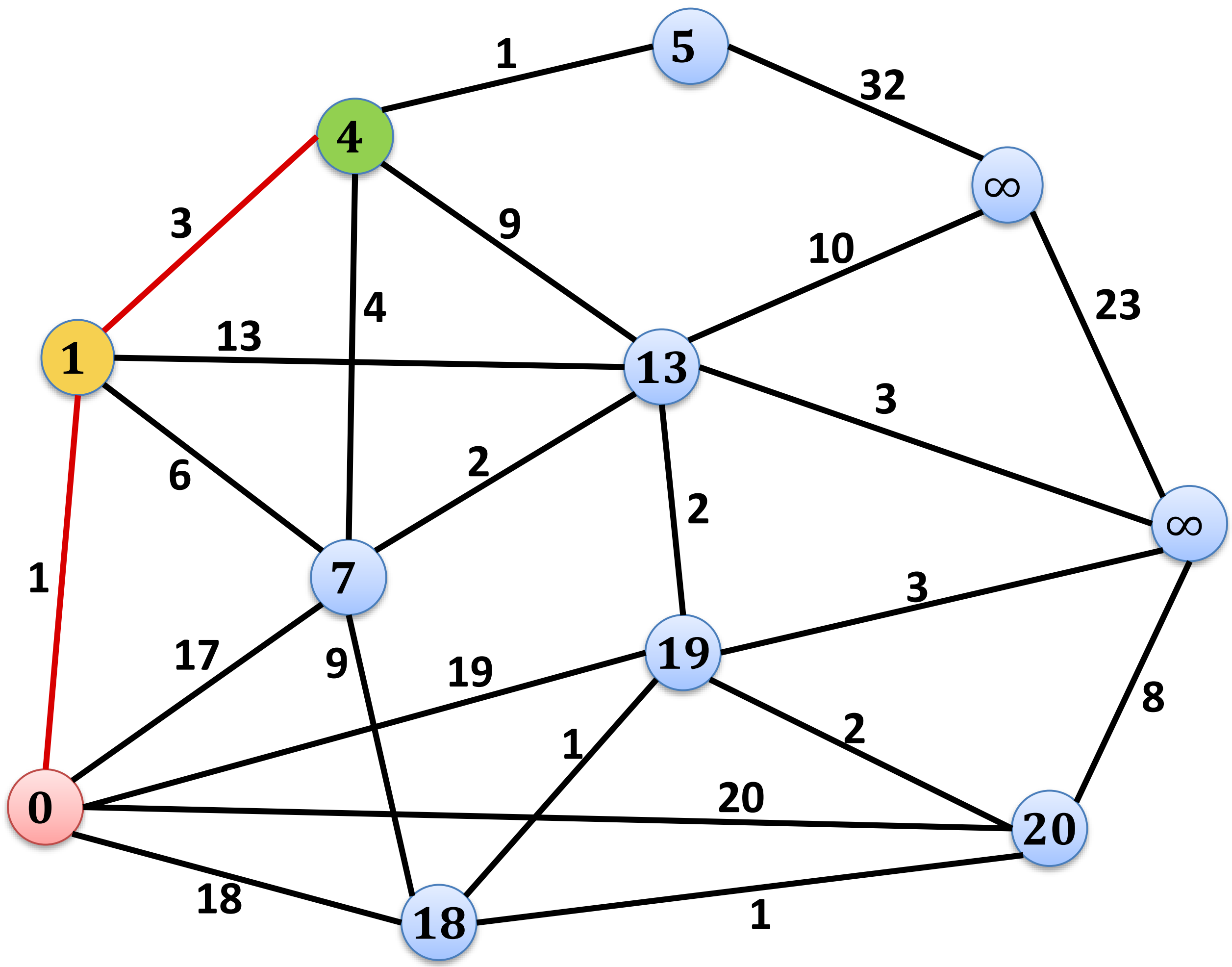
Example



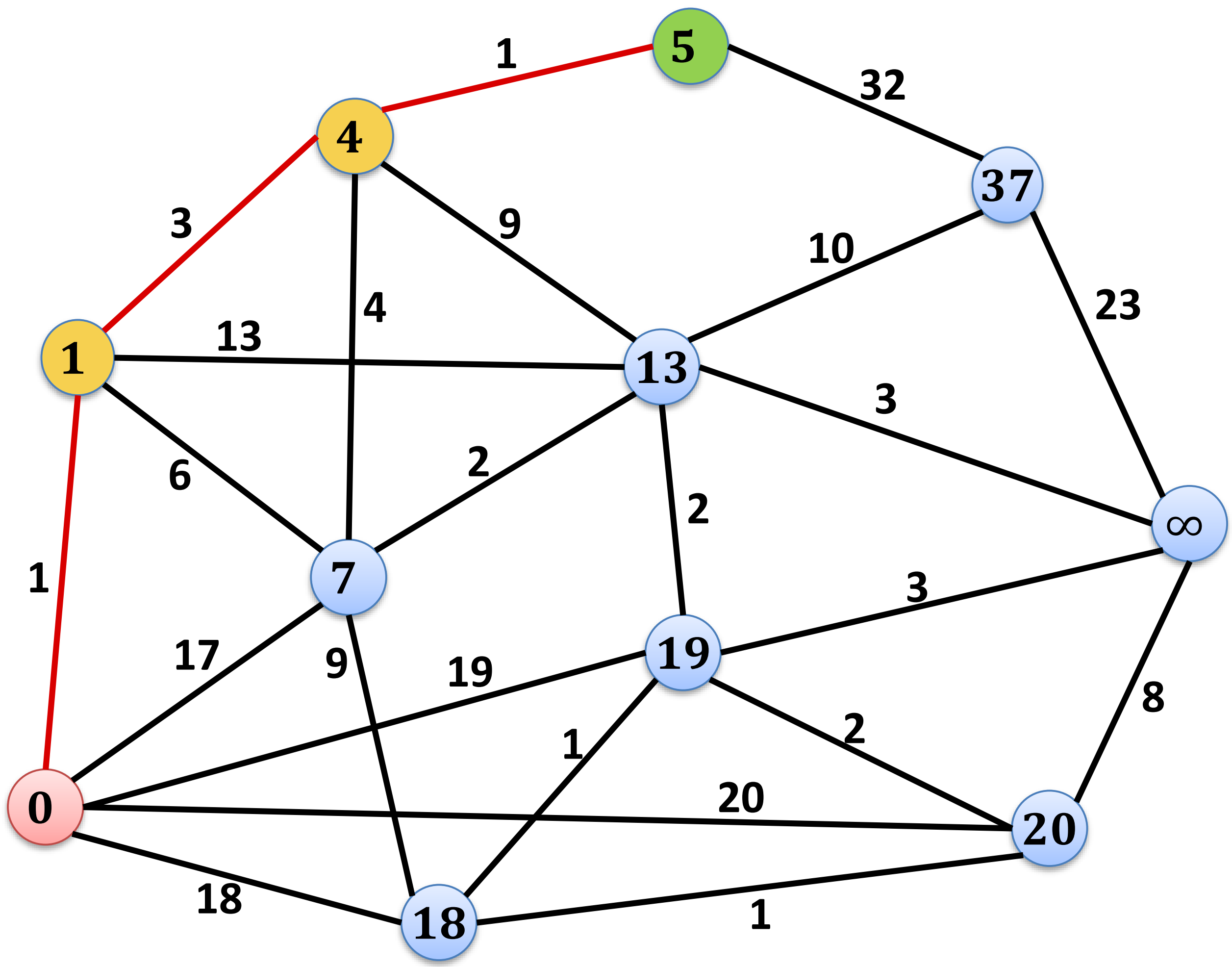
Example



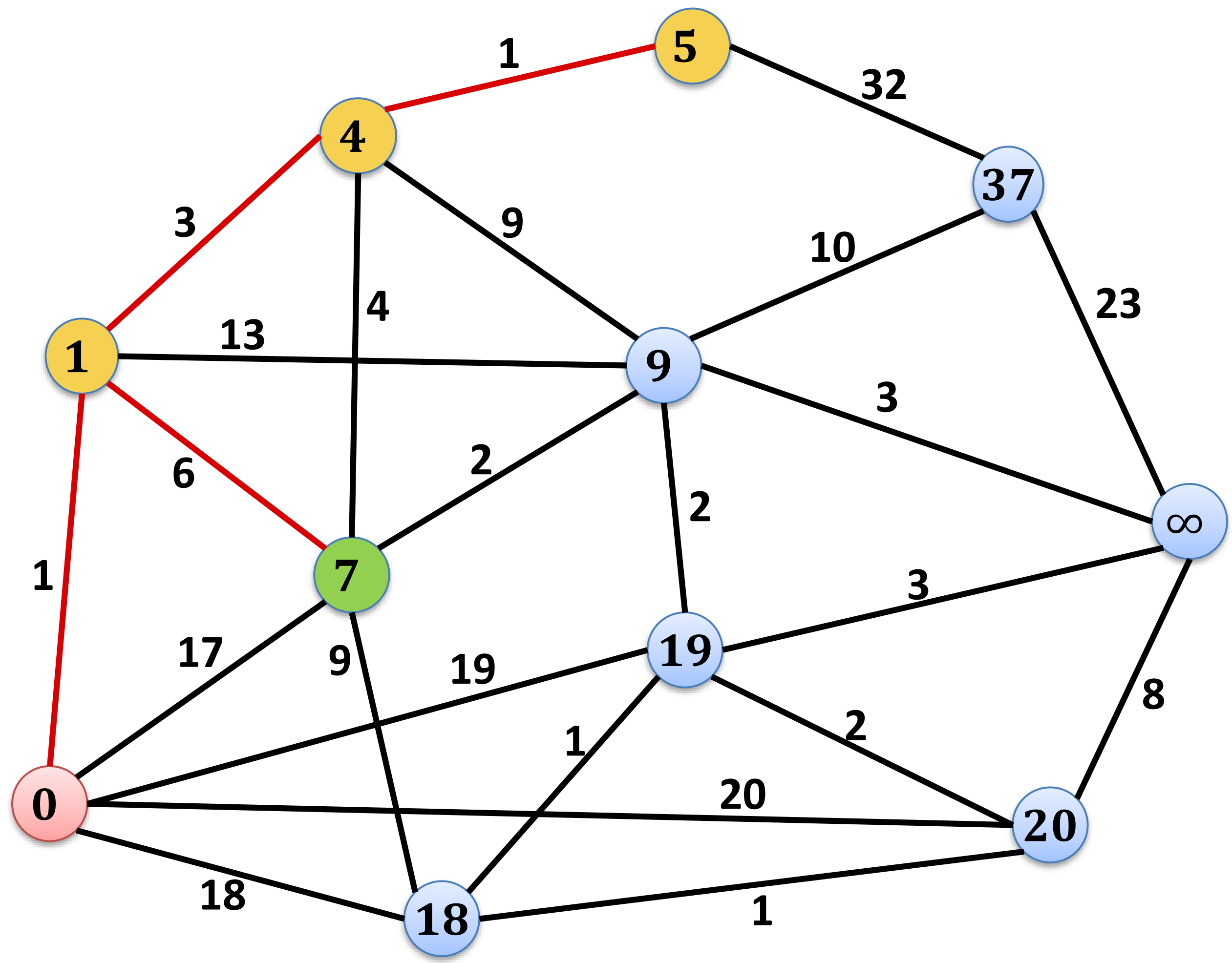
Example



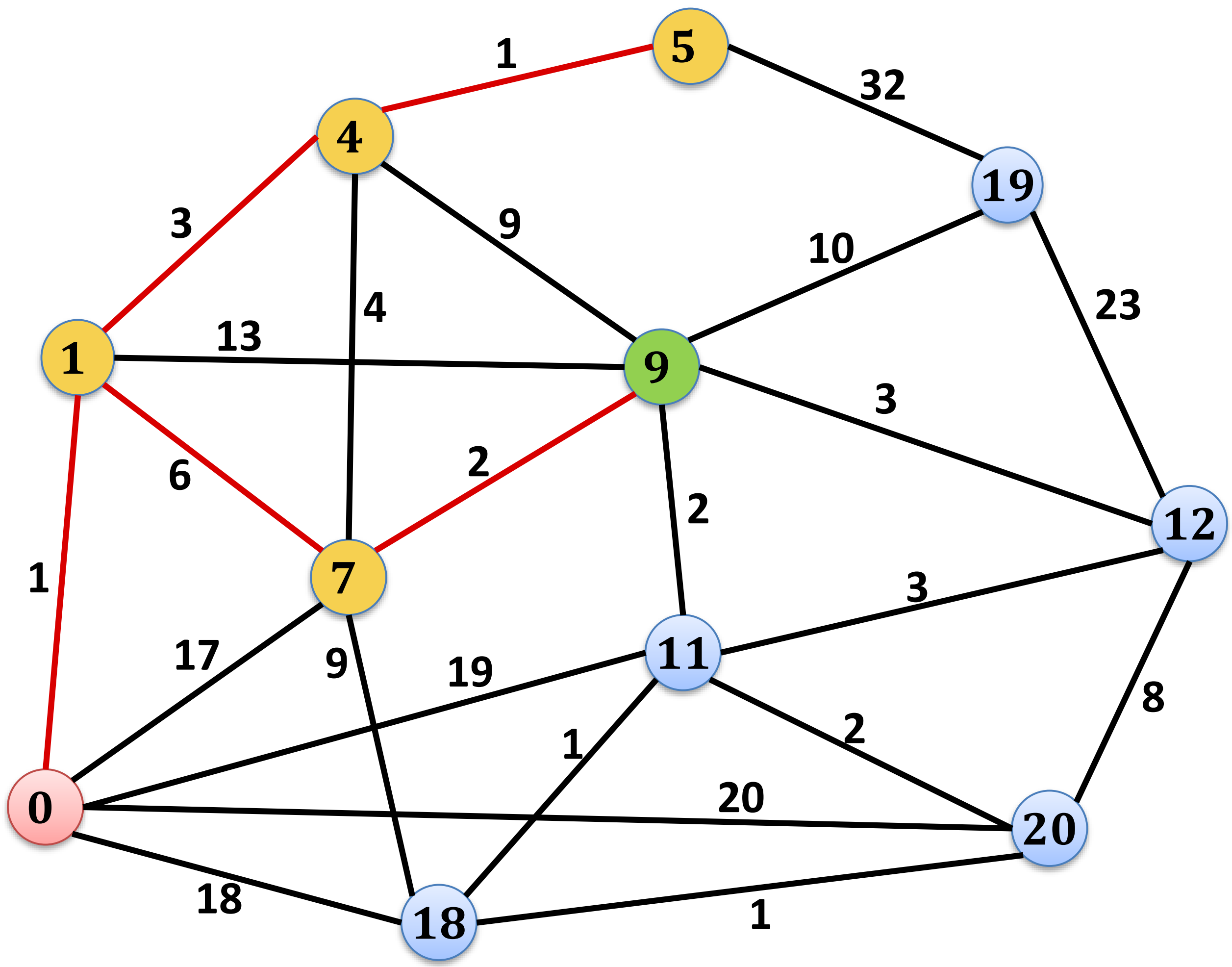
Example



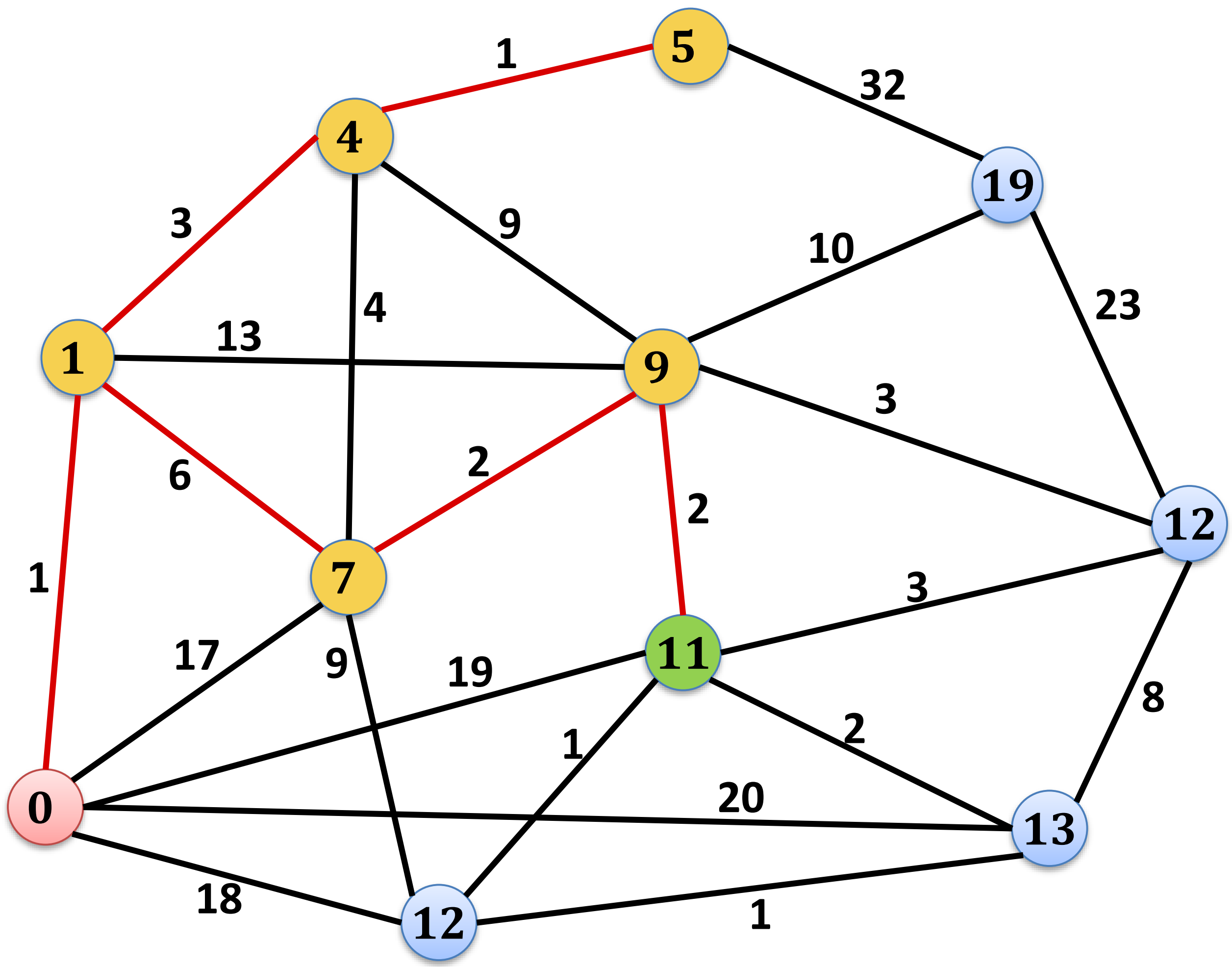
Example



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 to manage unmarked nodes
   add all nodes with their initial dist. estimate to DS

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

   "get node" from 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 ($\uparrow$ decrease) $d(s,v)$ for all unmarked neighbors v of u

5. mark node u

   delete u from DS

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
  - has to be consistent with get-min operation
- **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 priority 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 $e = \{u,v\} \in E$, $d(s,v) = \min\{d(s,v), d(s,u) + w(e)\}$

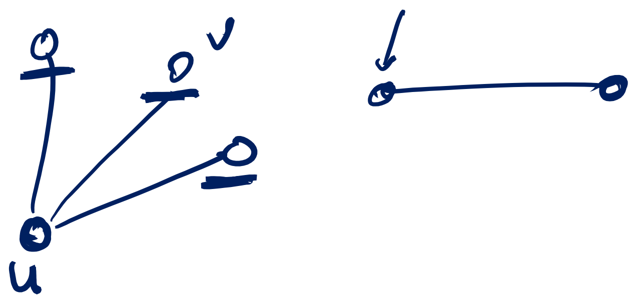
   for all neighbors of u: potentiall call decrease-key

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

Analysis

Number of priority queue operations for Dijkstra:

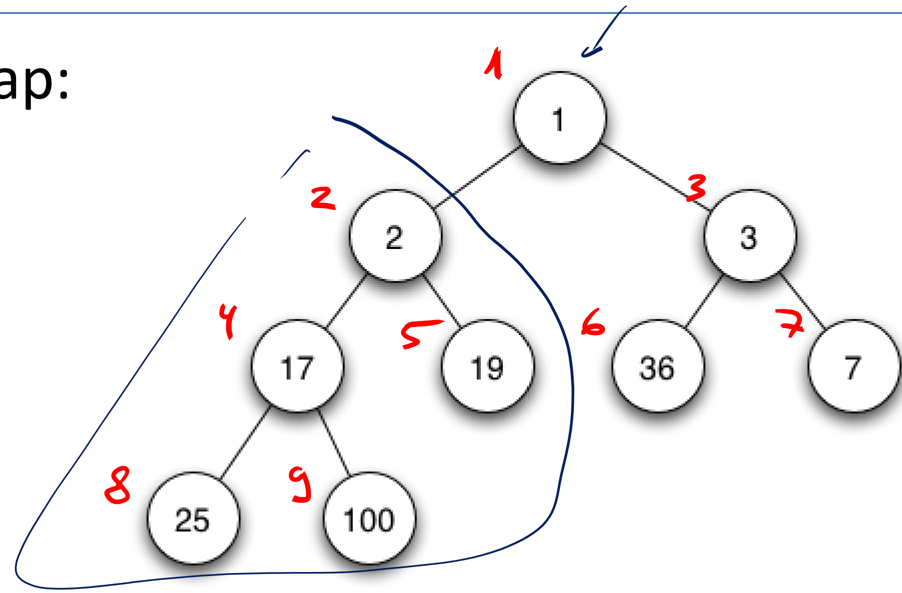
- **Initialize-Heap**: $\mathbf{1}$
- **Is-Empty**: $|V|$
- **Insert**: $|V|$
- **Get-Min**: $|V|$
- **Delete-Min**: $|V|$
- **Decrease-Key**: $|E|$
- **Merge**: $\mathbf{0}$

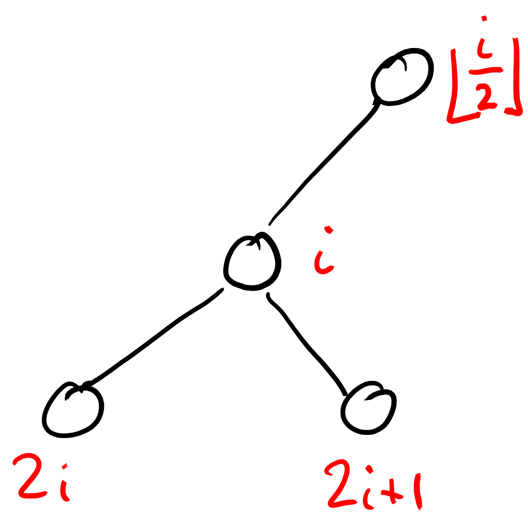


Priority Queue Implementation

Implementation as min-heap:

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

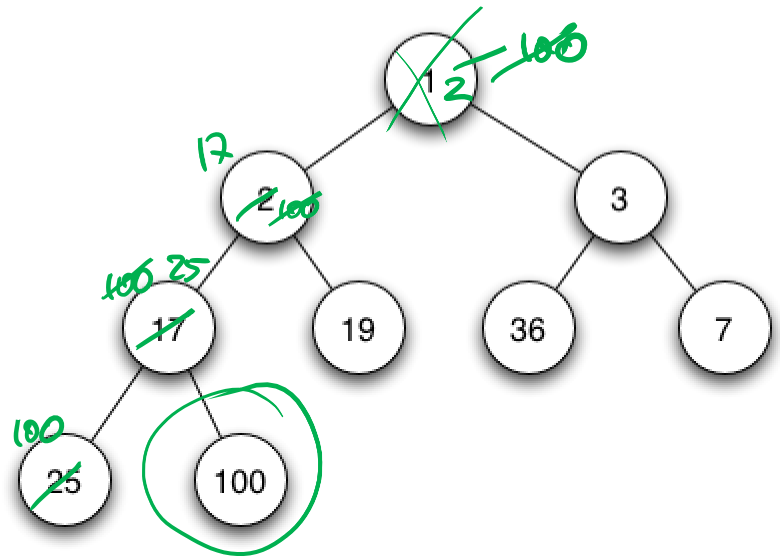




Priority Queue Implementation

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

insert(6)

delete-min

Dijkstra:

$|E| = m$

$|V| = n$

$O(m \cdot \log n)$

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?

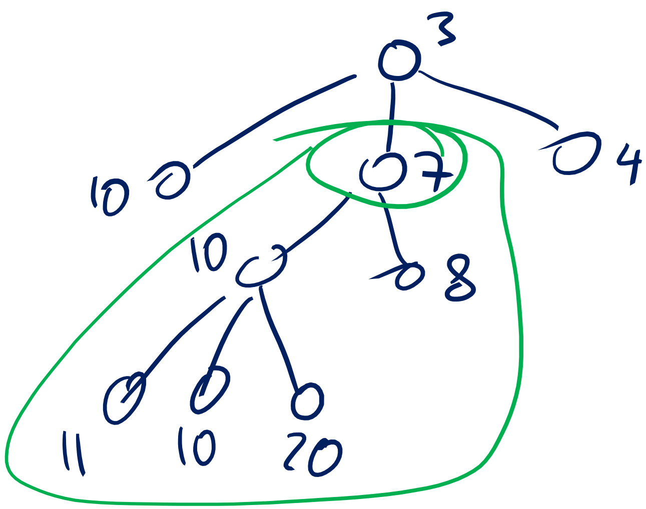
Fibonacci Heaps

Structure:

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

Min-Heap Property:

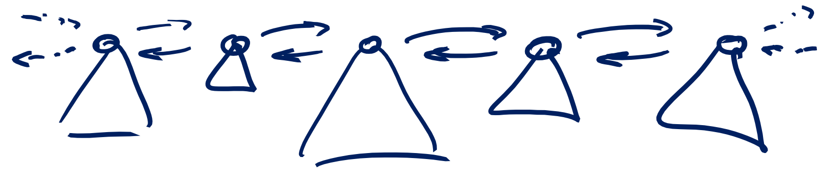
Key of a node $v \leq$ keys of all nodes in any sub-tree of v



Fibonacci Heaps

Structure:

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



Variables:

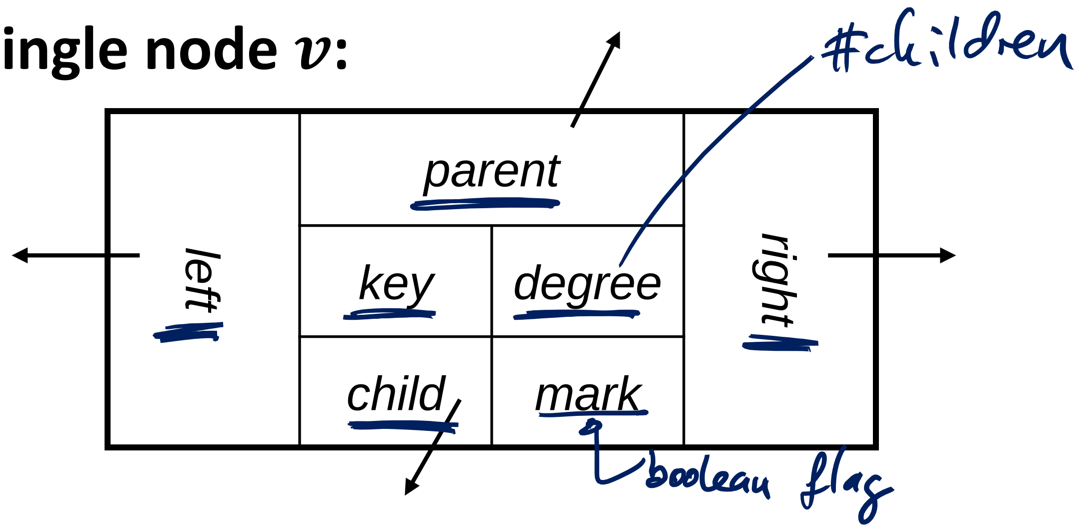
- $H.min$: root of the tree containing the (a) minimum key
- $H.rootlist$: circular, doubly linked, unordered list containing the roots of all trees
- $H.size$: number of nodes currently in H

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

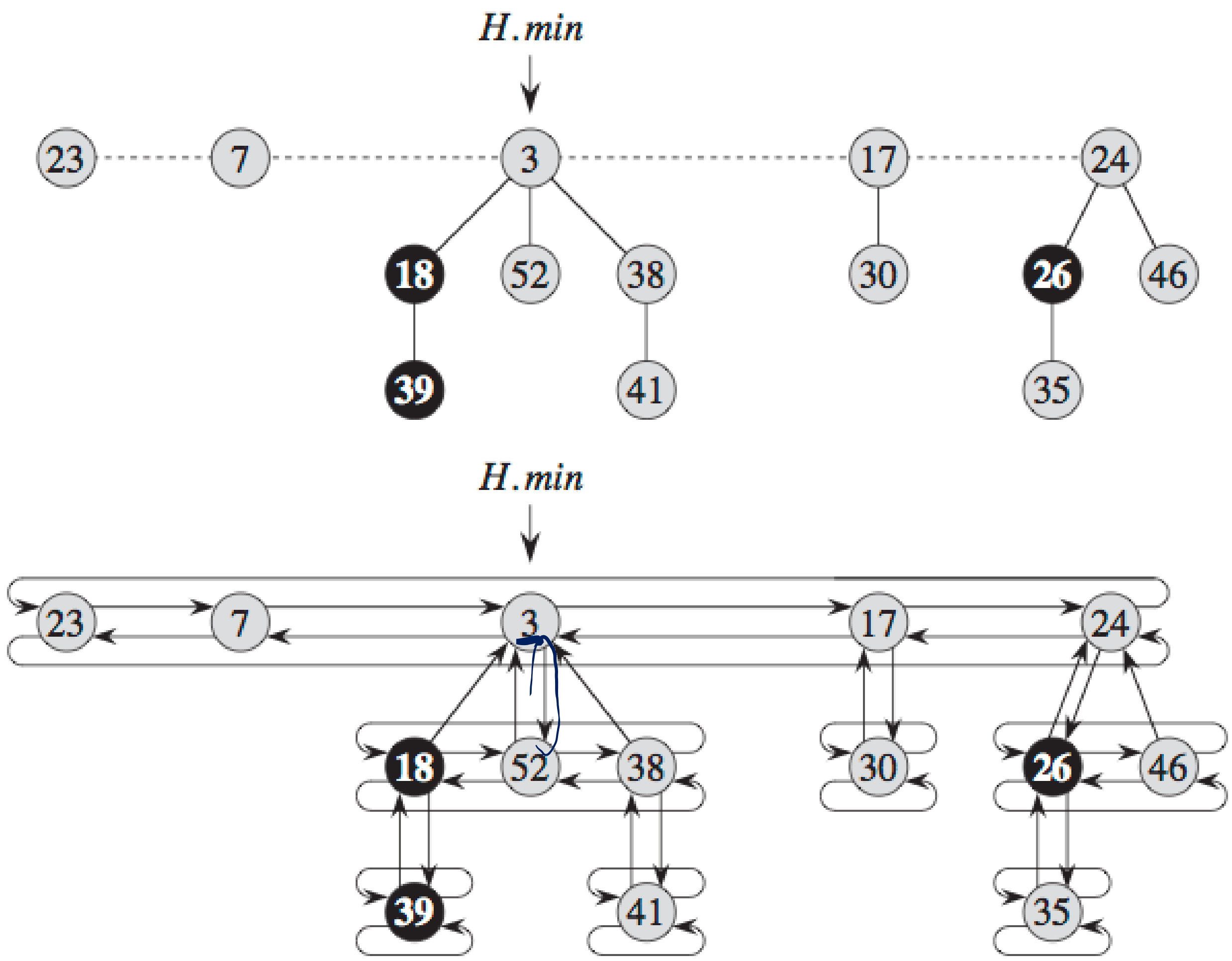


Figure: Cormen et al., Introduction to Algorithms

Simple (Lazy) Operations

Initialize-Heap H:

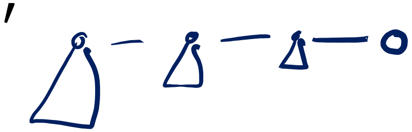
- $H.rootlist := H.min := null$

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 **false**



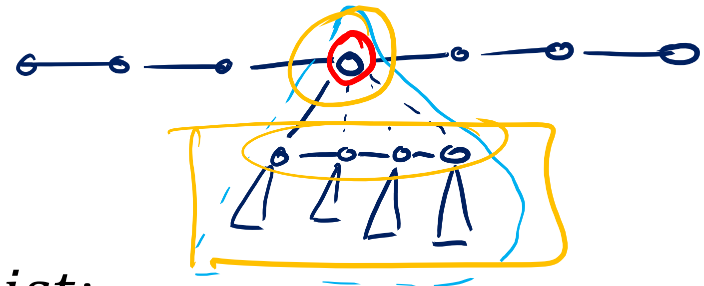
- merge heaps H and H'

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 $D(n)$

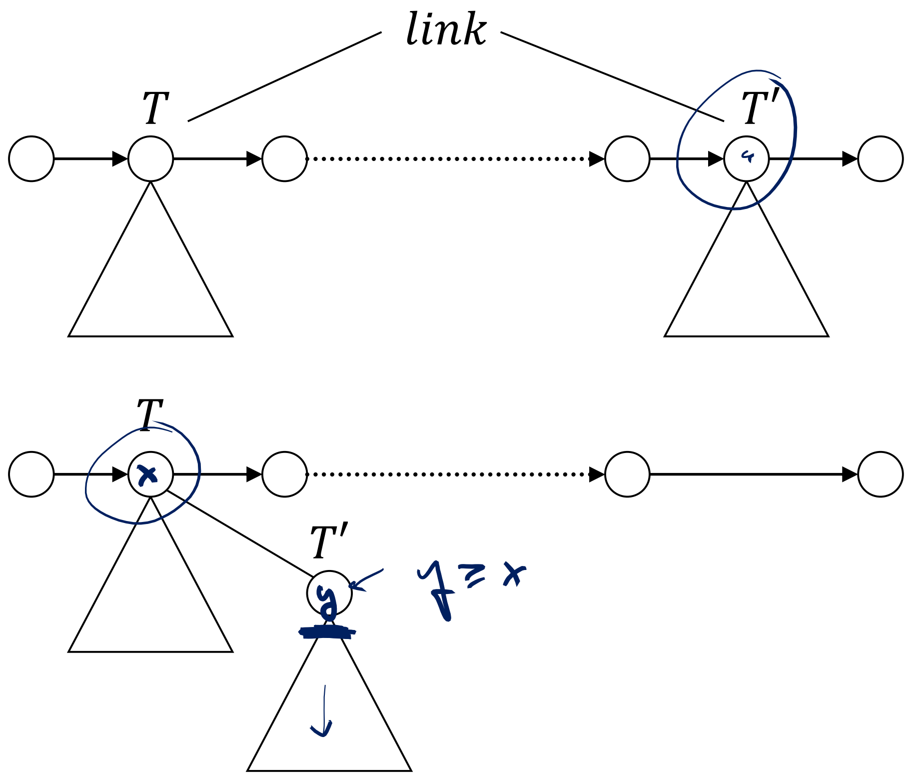
Merging Two Trees

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

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

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

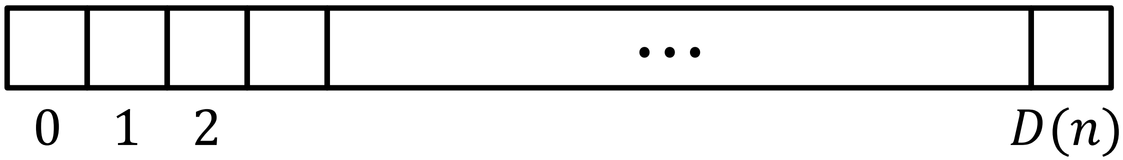
- Removes tree T' from root list and adds T' to child list of T



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

Consolidation of Root List

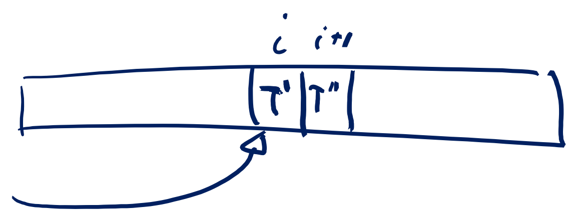
Array A pointing to find roots with the same rank:



Consolidate:

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

1. **for** $i := 0$ **to** $D(n)$ **do** $A[i] :=$ null;
2. **while** $H.rootlist \neq$ null **do**
3. $\quad T :=$ "delete and return first element of $H.rootlist$"
4. $\quad$ **while** $A[rank(T)] \neq$ null **do**
5. $\quad\quad T' := A[rank(T)]$;
6. $\quad\quad A[rank(T)] := null$;
7. $\quad\quad T := link(T, T')$
8. $\quad A[rank(T)] := T$
9. Create new $H.rootlist$ and $H.min$



time:

$O(|H.rootlist| + D(n) + \#\text{link ops.})$

rootlist size before consolidate