Algorithm Theory

Chapter 5
Data Structures

Part III:
Priority Queues, Warm-Up

Fabian Kuhn

Dijkstra's Algorithm

Single-Source Shortest Path Problem:

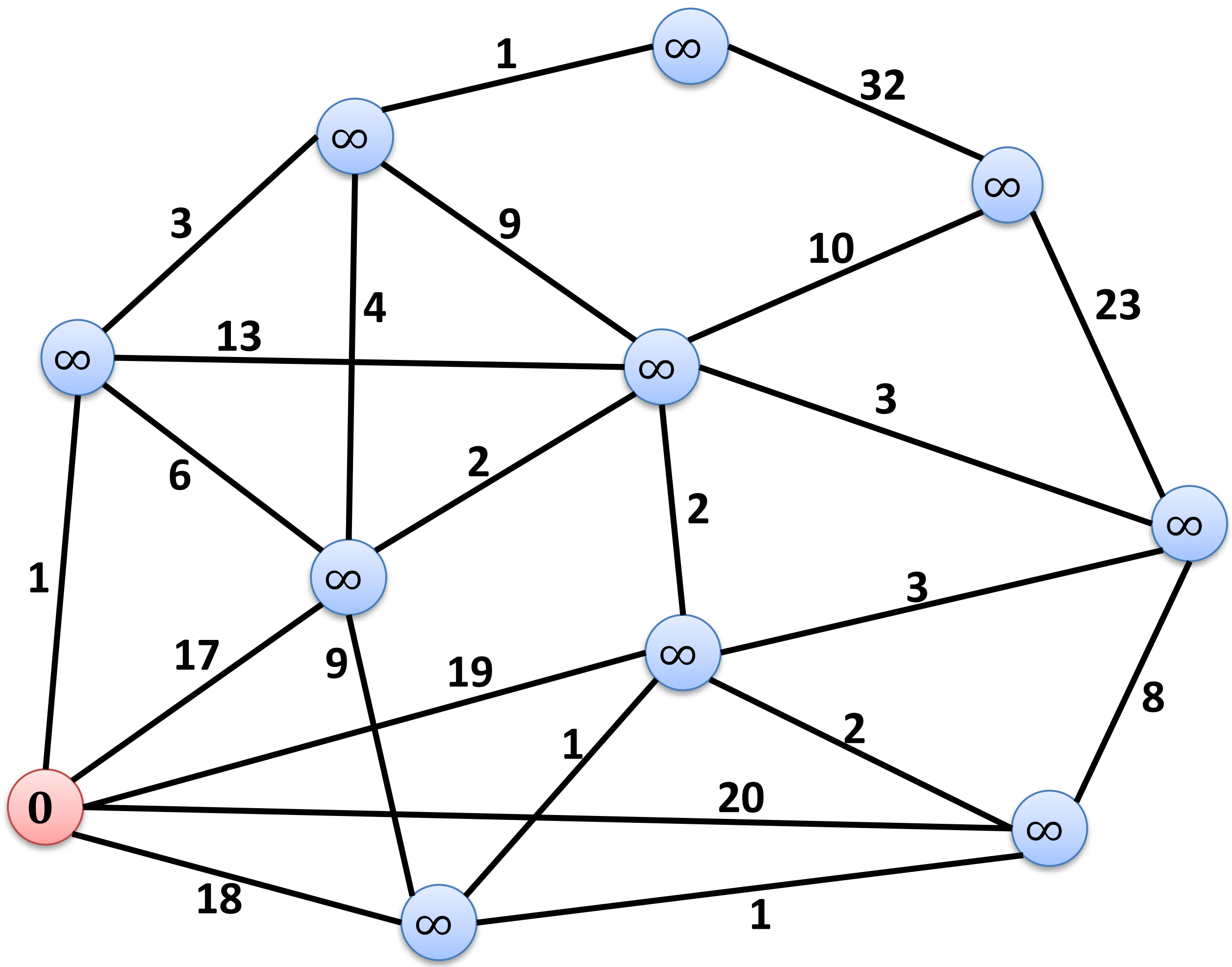
- **Given:** graph $G = (V, E)$ with edge weights $w(e) \geq 0$ for $e \in E$
  source node $s \in V$
- **Goal:** compute shortest paths from s to all $v \in V$

Dijkstra's Algorithm:

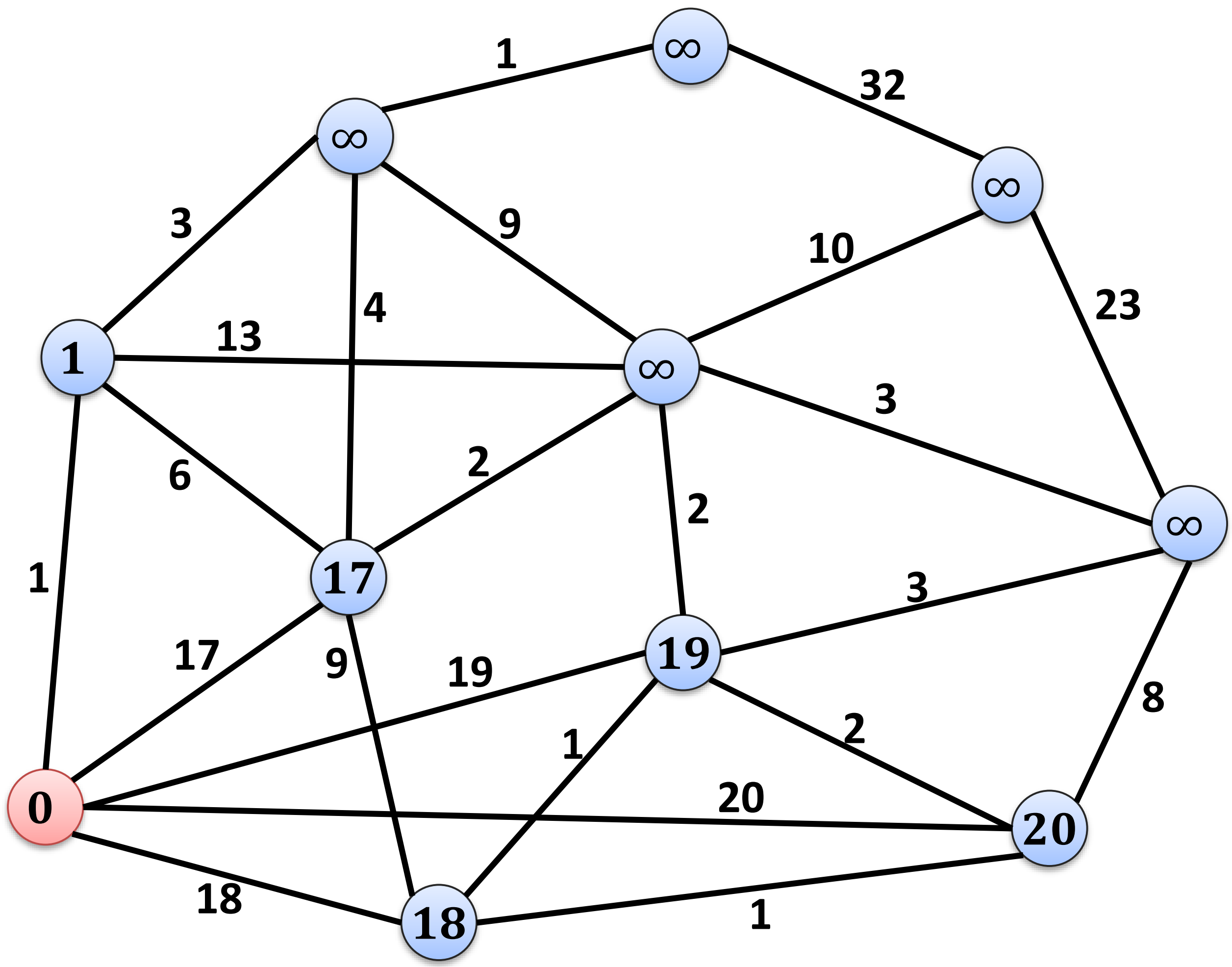
1. Initialize $d(s, s) = 0$ and $d(s, v) = \infty$ for all $v \neq s$
2. All nodes are unmarked
3. Get unmarked node u which minimizes $d(s, u)$:
4. mark node u
5. For all $e = \{u, v\} \in E$, $d(s, v) = \min\{d(s, v), d(s, u) + w(e)\}$
6. Until all nodes are marked

unmarked v

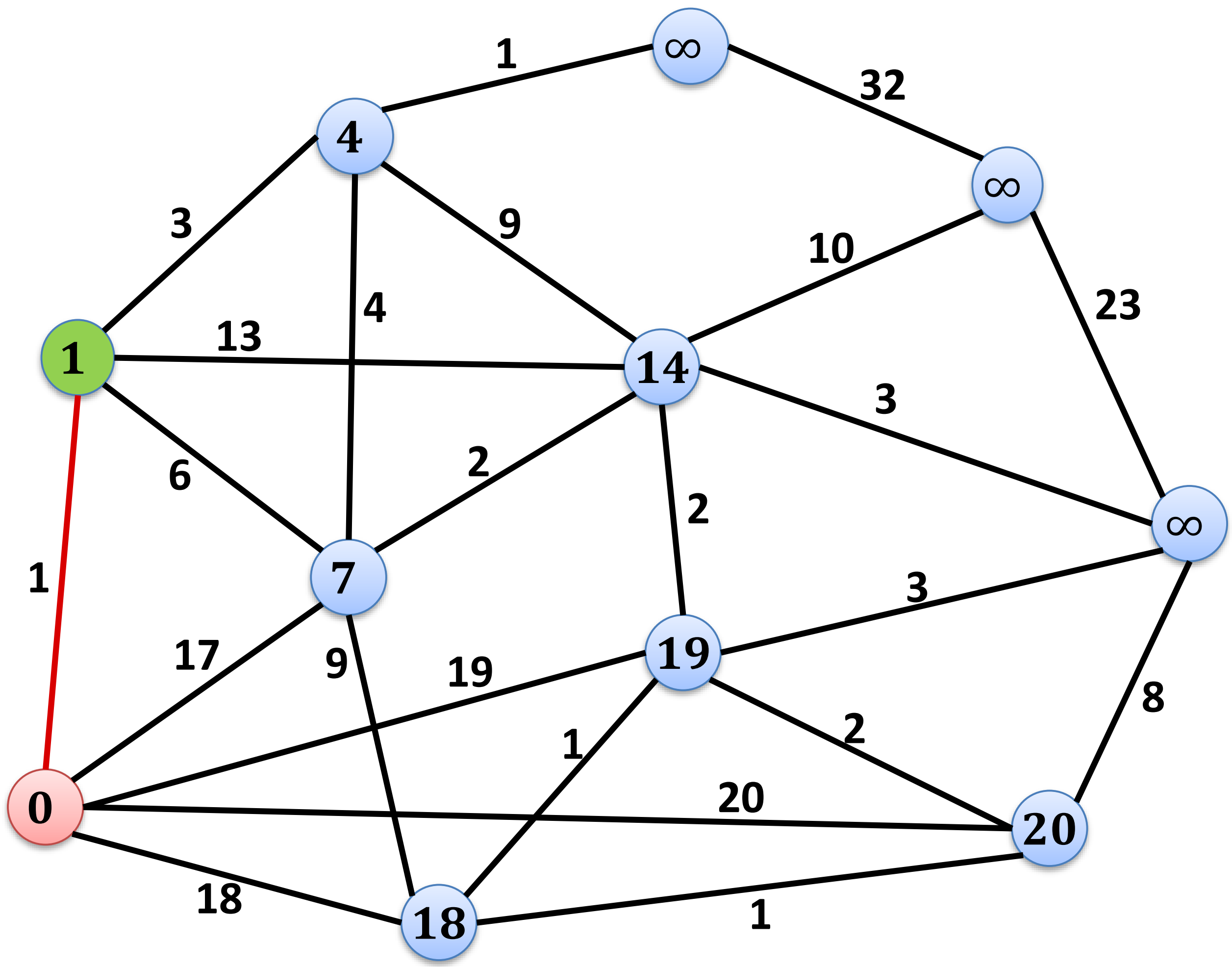
Example



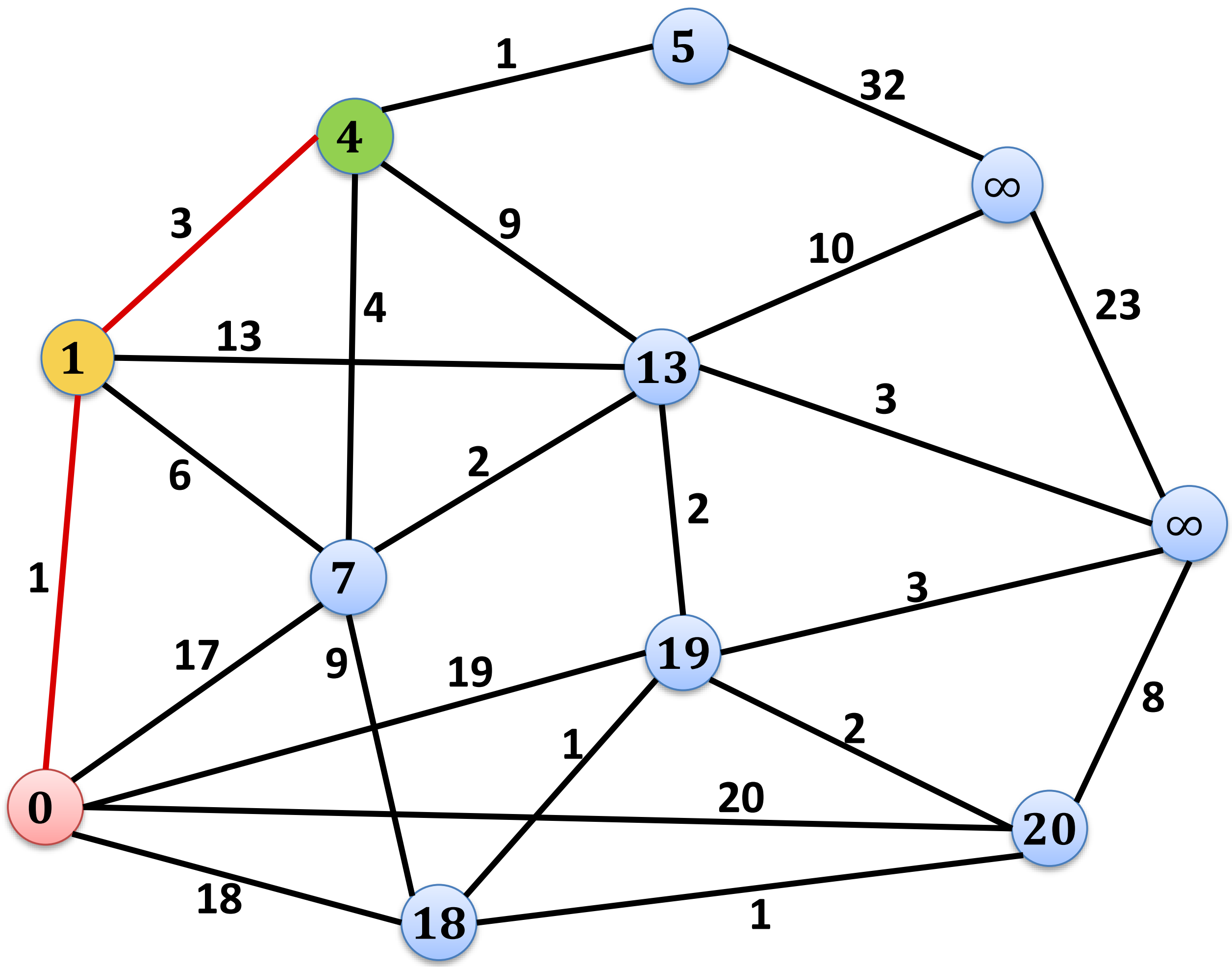
Example



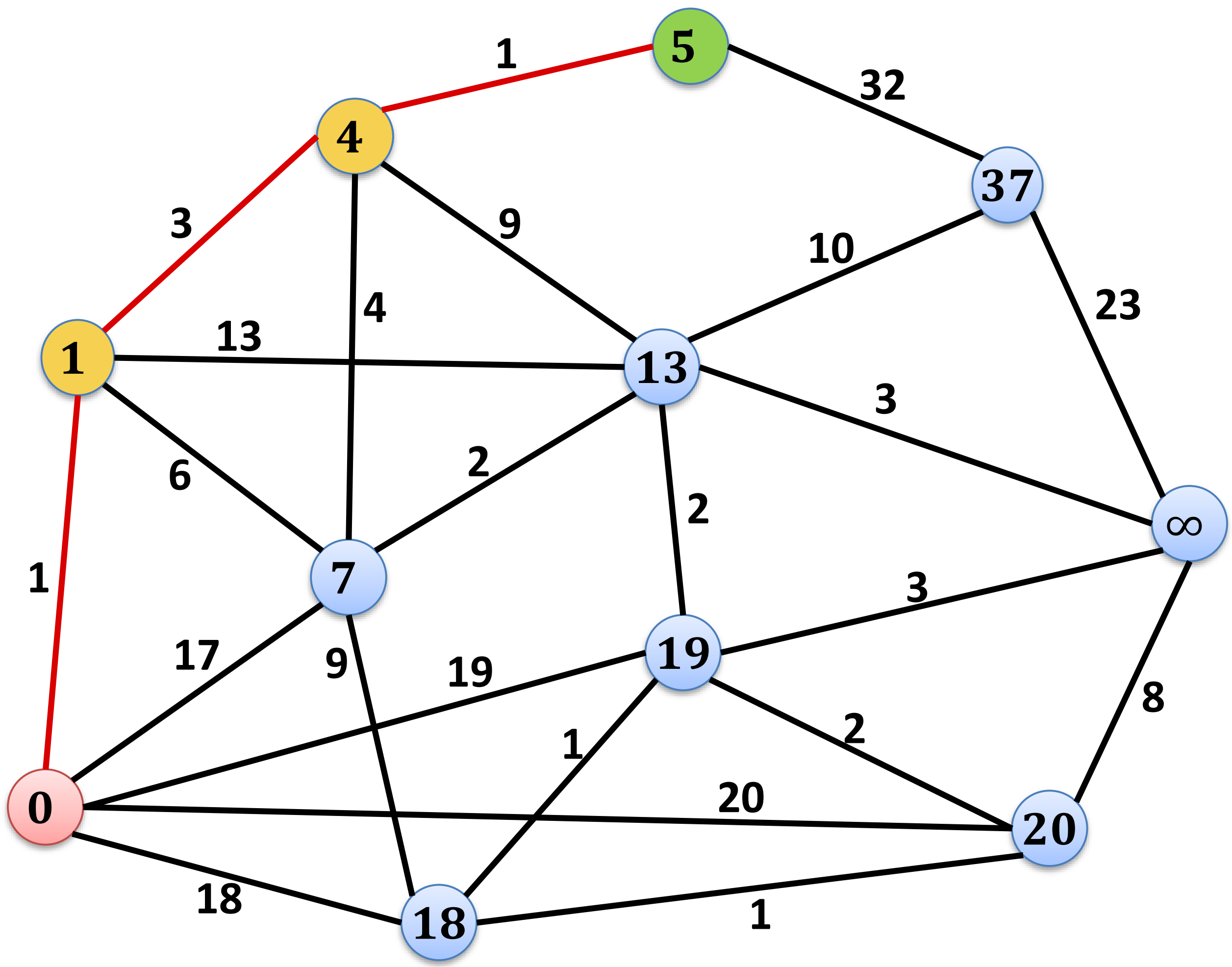
Example



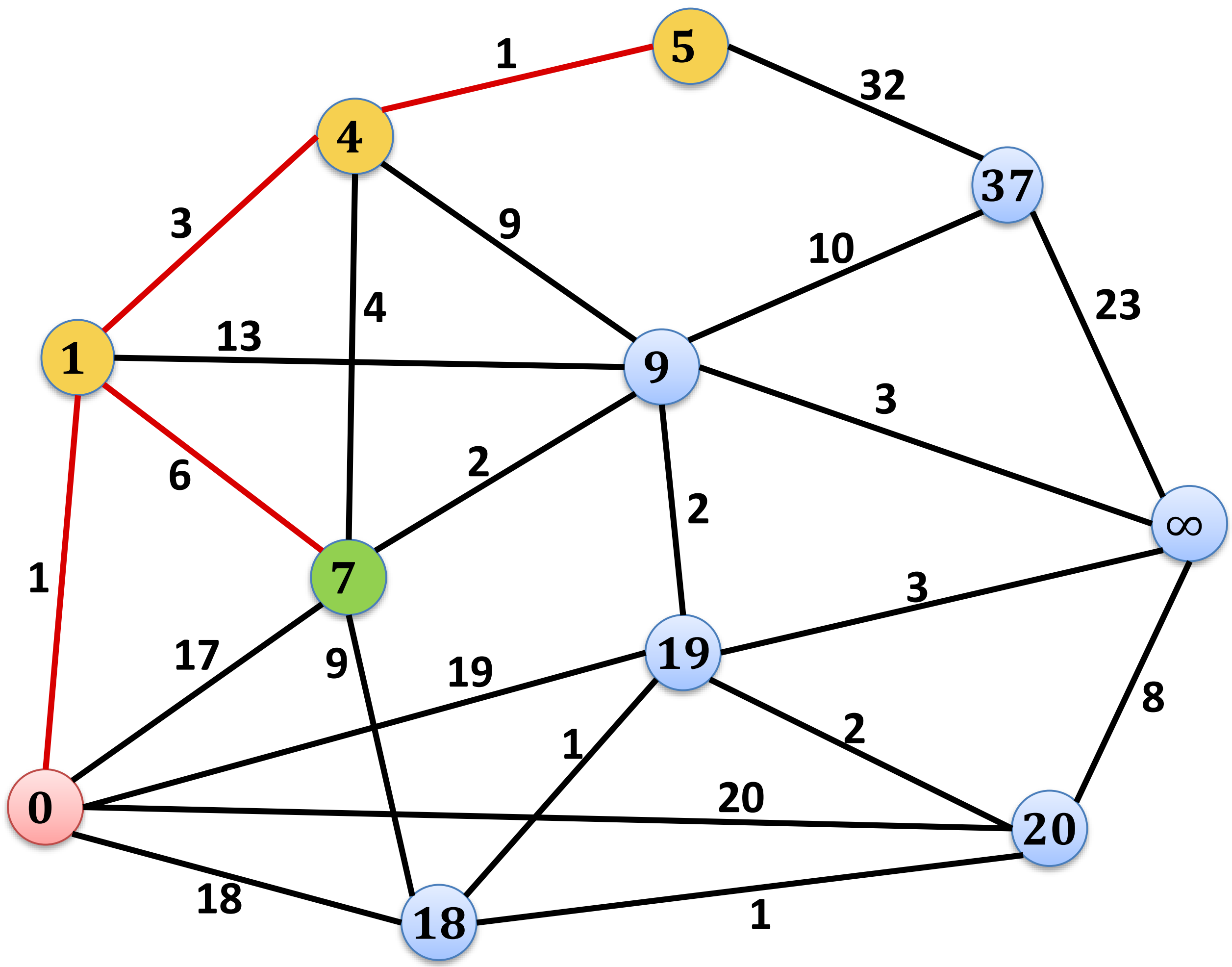
Example



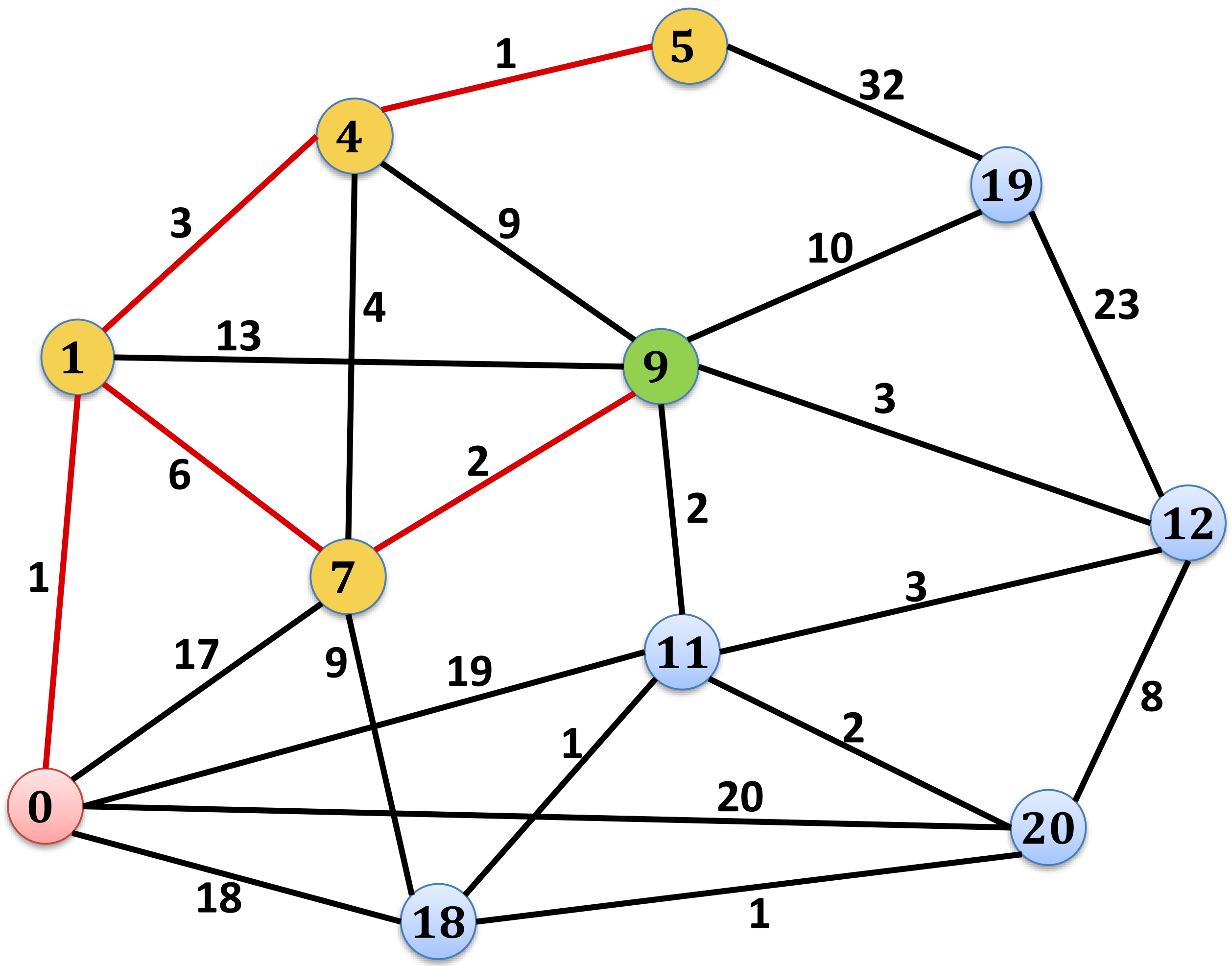
Example



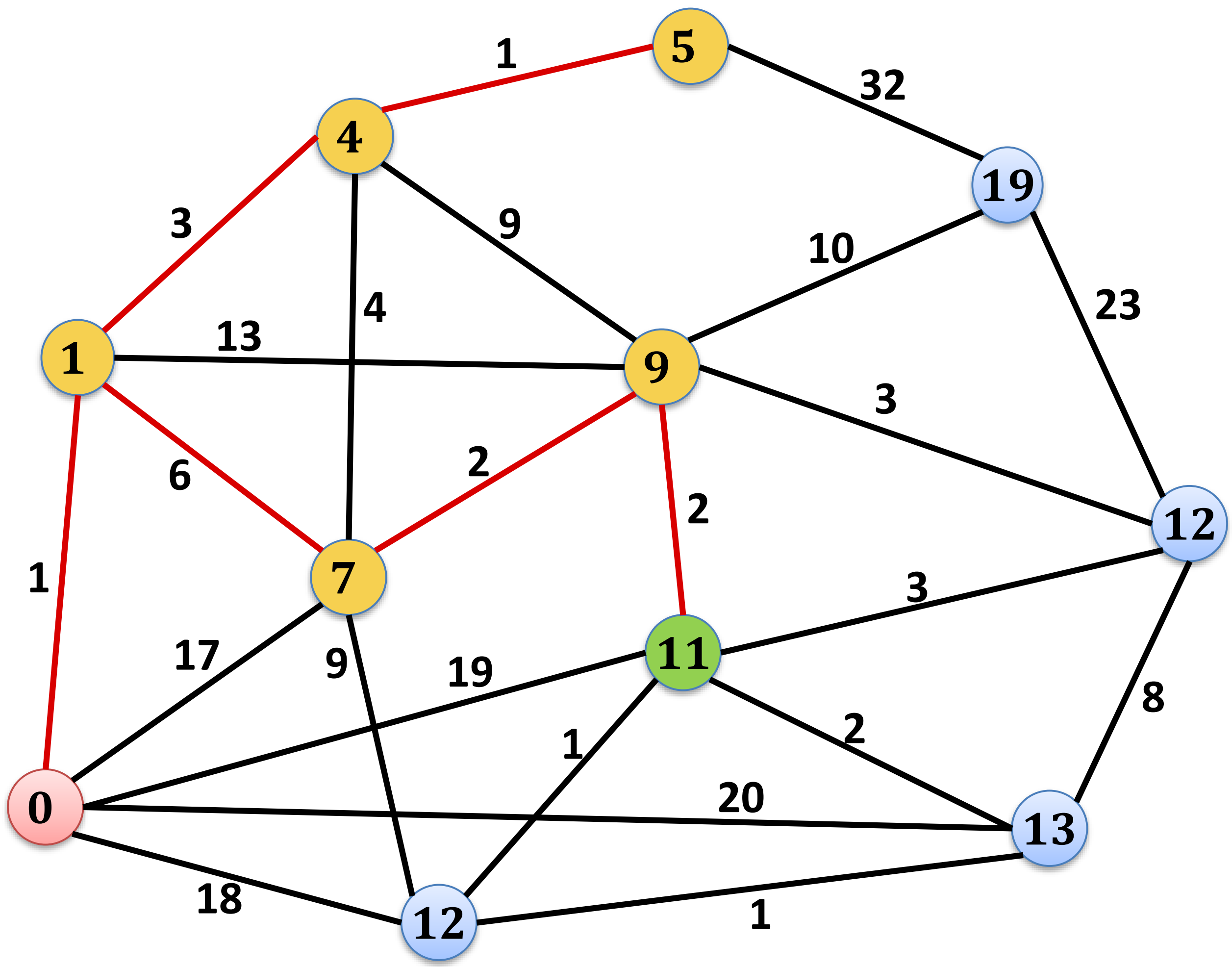
Example



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 (DS) to manage all unmarked nodes,
add all nodes to DS with initial distance estimates $d(s,v)$

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

Get node u from DS with minimum $d(s,u)$,
delete u from DS

unmarked v

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

Potentially update $d(s,v)$ for all unmarked neighbors of u

update = decrease

6. Until all nodes are marked

Minimum Spanning Trees

- We saw Kruskal’s algorithm for computing an MST
- An alternative algorithm to compute an MST is Prim’s algorithm
  - The algorithm is commonly known as Prim’s algorithm because it was published by Robert Prim in 1957.
  - The algorithm should better be called **Jarník's** algorithm because the Czech mathematician Vojtěch Jarník found it already 1930.

Prim/Jarník Algorithm:

1. Start with any node v (v is the initial component)
2. In each step:
   Grow the current component by adding the minimum weight edge e connecting the current component with any other node

Implementation of Prim/Jarník Algorithm

Start at node s, very similar to Dijkstra's algorithm :

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

data structure (DS) to manage all unmarked nodes,
add all nodes to DS with initial distance estimates $d(v)$

3. Get unmarked node u which minimizes $d(u)$:
4. mark node u

Get node u from DS with minimum $d(u)$,
delete u from DS

unmarked v

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

Potentially update $d(v)$ for all unmarked neighbors of u

update = decrease

6. Until all nodes are marked

Priority Queue / Heap

- Stores (*key,data*) pairs
  - like a dictionary, but with a 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 (and returns) 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

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

create empty priority queue Q,
add all nodes to Q with initial key $d(s,v)$

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

$u := Q$.delete_min()

unmarked v

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

For all unmarked neighbors v of u: potentially call Q.decrease_key

6. Until all nodes are marked

until Q is empty

Implementation of Prim/Jarník Algorithm

Start at node s, very similar to Dijkstra's algorithm :

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

   create empty priority queue Q,
   add all nodes to Q with initial key $d(v)$

3. Get unmarked node u which minimizes $d(u)$:
4. mark node u

   u := Q.delete_min()

   unmarked v

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

   For all unmarked neighbors v of u: potentially call Q.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: $\boldsymbol{n}$
- **Insert:** $\boldsymbol{n}$
- Get-Min: $\mathbf{0}$
- **Delete-Min:** $\boldsymbol{n}$
- **Decrease-Key:** $\leq \boldsymbol{m}$
- Merge: $\mathbf{0}$

Assumption:

$n = |V|$ (number of nodes)
$m = |E|$ (number of edges)

- $m \geq n - 1$

#Decrease-Key:

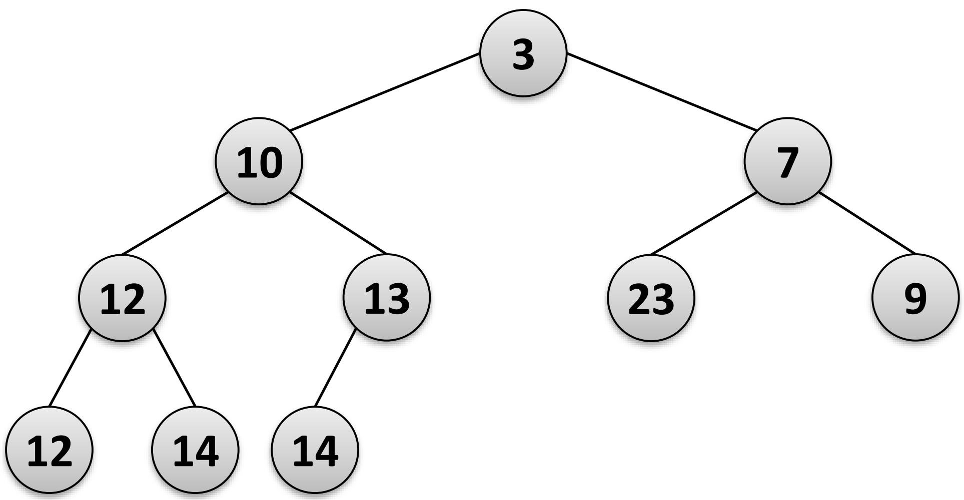
Always for an unmarked neighbor v of a newly marked node u

$\Longrightarrow \leq 1$ decrease-key per edge

Basic Priority Queue Implementation

Binary Heap:

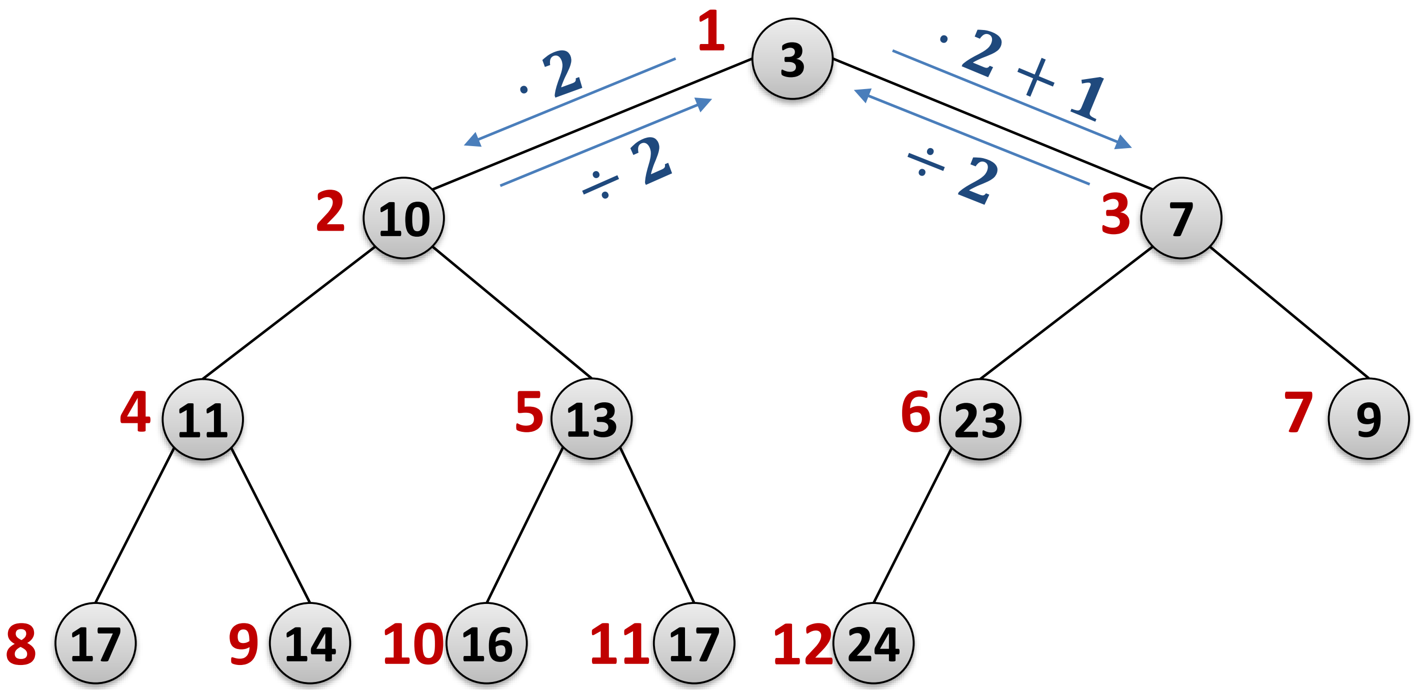
- Implementation as a **binary tree** with the **min-heap property**
- A tree has the min-heap property if in every subtree, the root has the smallest key.
- Tree is always as balanced as possible
  - All levels except for bottom level are full, bottom-most level is filled from left to right.
- **insert**, **delete-min**, **decrease-key** all have **worst-case time $O(\log n)$**.



Array Implementation of Binary Heaps

Store everything in an array at positions 1 to n

- This is possible because the binary tree is perfectly balanced



- For a node at position i
  – Left child is at position $j = 2 \cdot i$, right child is at position $j = 2 \cdot i + 1$
  – Parent is a position $j = i/2$ (integer division, i.e., $j = \lfloor i/2 \rfloor$)

Can We Do Better?

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

$$O(m \cdot \log n)$$

- **Binary 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?