Algorithms and Data Structures

Lecture 9

Graph Algorithms II: Minimum Spanning Trees

Fabian Kuhn
Algorithms and Complexity

Graphs

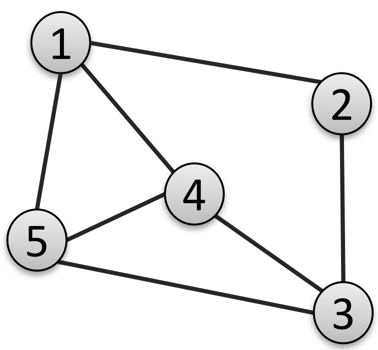
Node Set $\boldsymbol{V}$, typically $n \coloneqq |V|$ (alternatively, node = **v**ertex)

Edge Set $\boldsymbol{E}$, typically $m \coloneqq |E|$

- Undirected graph: $E \subseteq \big\{\{u, v\} : u, v \in V\big\}$

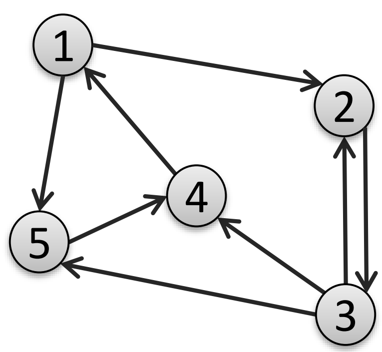
In this lecture: only undirected graphs

Examples:



$V = \{1, 2, 3, 4, 5\}$
$E = \big\{\{1,2\}, \{1,4\}, \{1,5\}, \{2,3\}, \{3,4\}, \{3,5\}, \{4,5\}\big\}$

$V = \{1, 2, 3, 4, 5\}$
$E = \{(1,2), (1,5), (2,3), (3,4), (3,4), (3,5), (4,1), (5,4)\ \}$

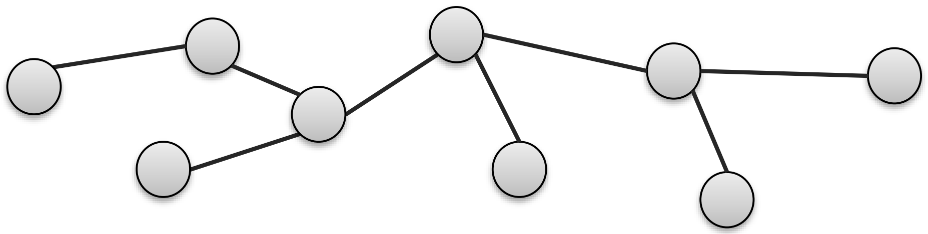


Trees

- Considered as undirected graphs (with n nodes)…

Tree:

- Connected undirected graph without cycles
  - A cycle-free not necessarily connected (undirected) graph is called a forest
  - Number of edges: $n-1$ (each edge reduces the no. of components by 1)



Equivalent Definitions:

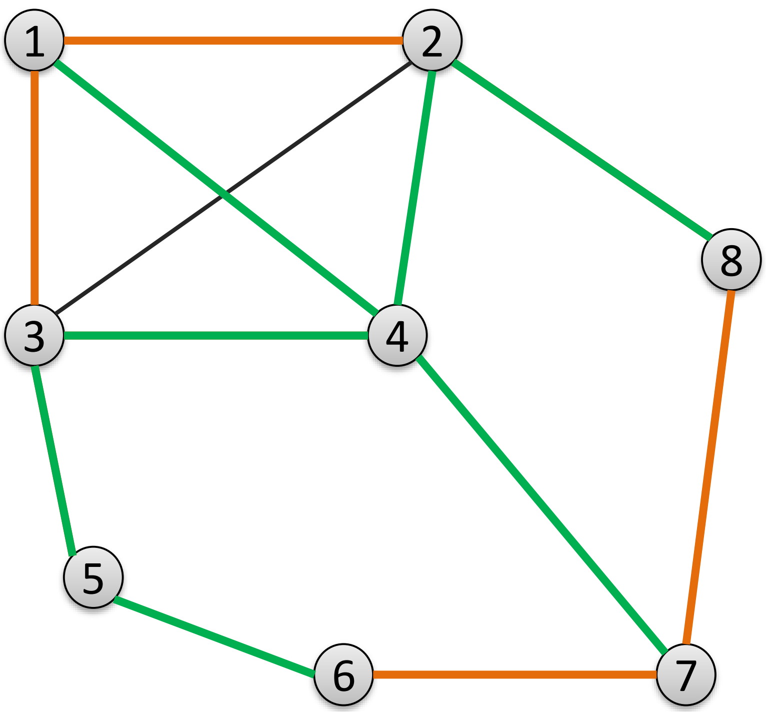
- minimal connected graph
- maximal cycle-free graph
- a unique path between every pair of nodes
- connected graph with $n-1$ edges

Spanning Tree

Given: Connected, undirected graph $G = (V, E)$

Spanning Tree $\boldsymbol{T = (V, E_T)}$: subgraph $(E_T \subseteq E)$

- T is a tree that contains all nodes of G
- Alternatively: T is a tree with $n - 1$ edges from E

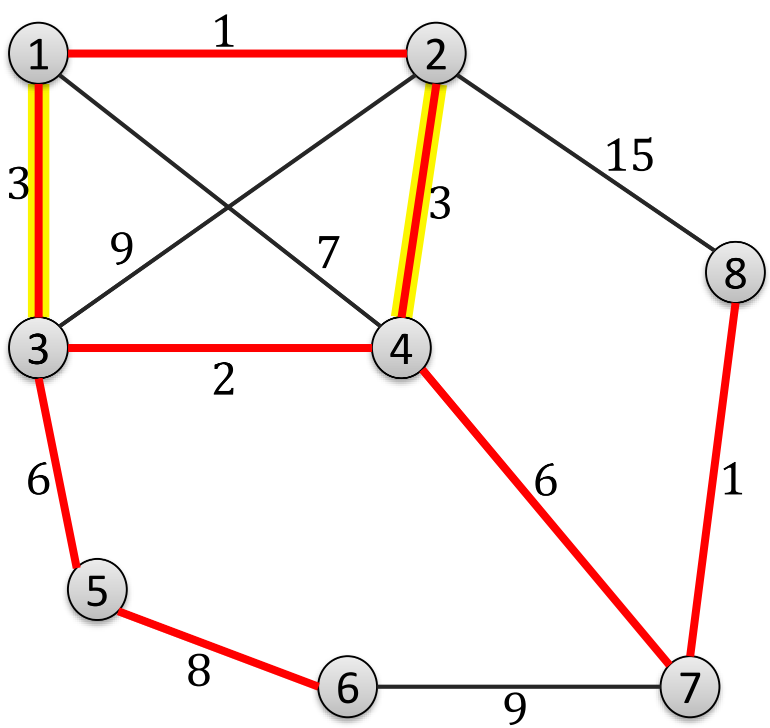


Minimum Spanning Tree (MST)

Given: Connected, undirected graph $G = (V, E, w)$
with edge weights $w : E \to \mathbb{R}$

Minimum Spanning Tree $T = (V, E_T)$:

- A spanning tree with smallest total weight



Minimum Spanning Trees

Goal: Given an undirected, connected graph G, find a spanning tree with minimum total weight.

- **Minimum Spanning Tree = MST**
- A fundamental optimization problem on graphs
  - one of many optimization problems on graphs
- Often appears as a subproblem
- MSTs are however also interesting by themselves
  - For example in the context of networks
  - A minimum spanning tree is the cheapest way of connecting all the nodes of a network.

Generic MST Algorithm

Idea: Start with an empty edge set and add edges step-by-step until we have a spanning tree.

Invariant:
At all times, the algorithm has an edge set A, such that A is the subset of the edges of a minimum spanning tree.

- In the beginning, we have $A = \emptyset$
- Afterwards, we always add an edge that preserves the invariant.
- We call an edge for which we can be sure that we can add the edge to A (and preserve the invariant), a safe edge for A
- How one can find safe edges, we will see…

Generic MST Algorithm

Invariant:
At all times, the algorithm has an edge set A, such that A is the subset of the edges of a minimum spanning tree.

Generic MST Algorithm:

A = $\emptyset$
while A is not a spanning tree **do**
    Find a safe edge $\{u, v\}$ for A
    A = $A \cup \{\{u, v\}\}$
return A

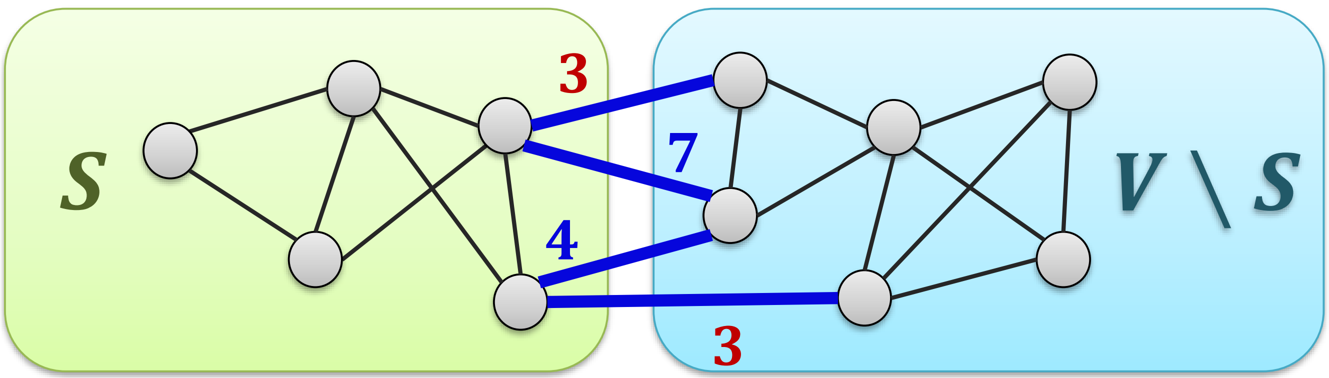
- Invariant is a valid loop invariant
- Invariant + condition for exiting the loop $\Longrightarrow A$ is an MST!

How can we find safe edges?

- Invariant → there is always a safe edge
  - A is the subset of an MST and can therefore be extended to an MST

- We first need some terminology …

Cut $(S, V \setminus S)$, $S \neq \emptyset$, $S \neq V$:



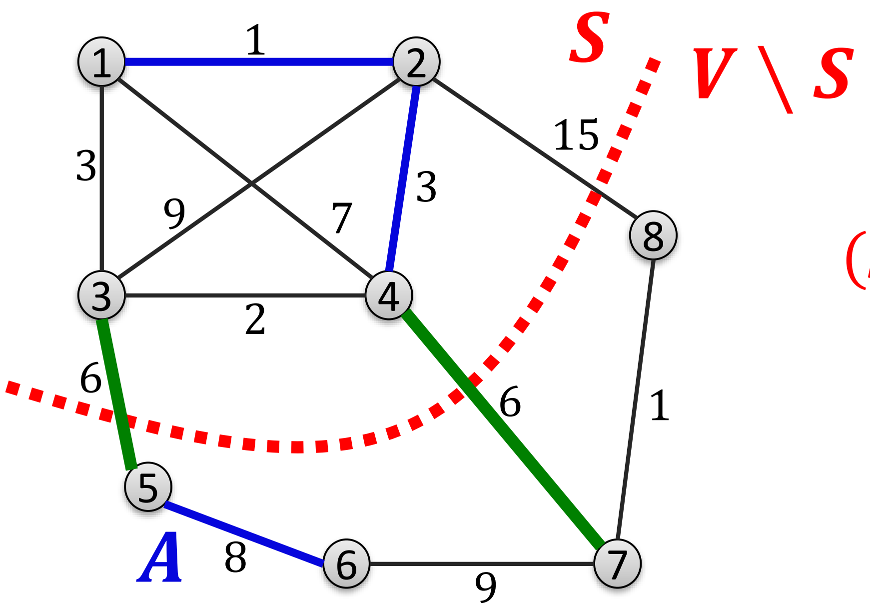
- An edge $\{u, v\} \in E$ is a cut edge w.r.t. $(S, V \setminus S)$ if one node of the edge is in S and one node of the edge is in $V \setminus S$.
- We call an edge $\{u, v\}$ a **light cut edge** w.r.t. $(S, V \setminus S)$ if the edge has the smallest weight among all cut edges.

Safe Edges

Assumption:

- $G = (V, E, w)$ is a connected, undir. graph with edge weights $w(e)$
- A is a subset of the edges of an MST

Theorem: Let $(S, V \setminus S)$ be a cut s.t. A does not contain any cut edges and let $\{u, v\}$, $u \in S$, $v \in V \setminus S$ be a light cut edge w.r.t. $(S, V \setminus S)$. Then, $\{u, v\}$ is a safe edge for A.



$\boldsymbol{A}$: edge set that is subset of the edges of an MST.

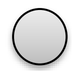
$(\boldsymbol{S}, \boldsymbol{V} \setminus \boldsymbol{S})$: cut for which no edge in A is a cut edge.

Light cut edges are safe edges for A and can thus be added to A.

Kurzer Exkurs zu Bäumen

Theorem: A connected (undirected) graph $G = (V, E)$ with n nodes and $n - 1$ edges is a tree.

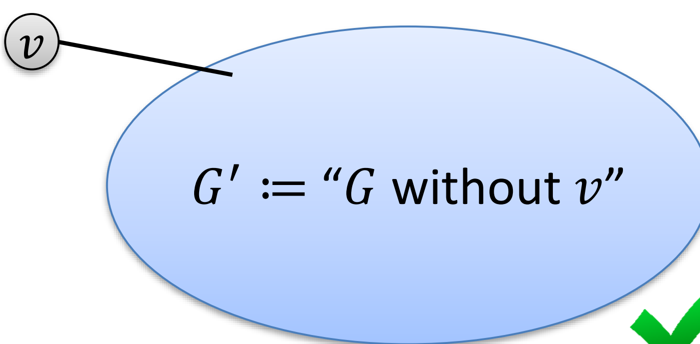
Proof: By induction on n

- **Induction Base** ($n = 1$): 

- **Induction Step** ($n - 1 \ \rightarrow \ n$):
  - A graph with n nodes and $n - 1$ edges has a node of degree ≤ 1

$$\text{avgdeg}(G) = \frac{1}{n} \cdot \sum_{v \in V} \deg(v) = \frac{2|E|}{n} = \frac{2n - 2}{n} < 2$$

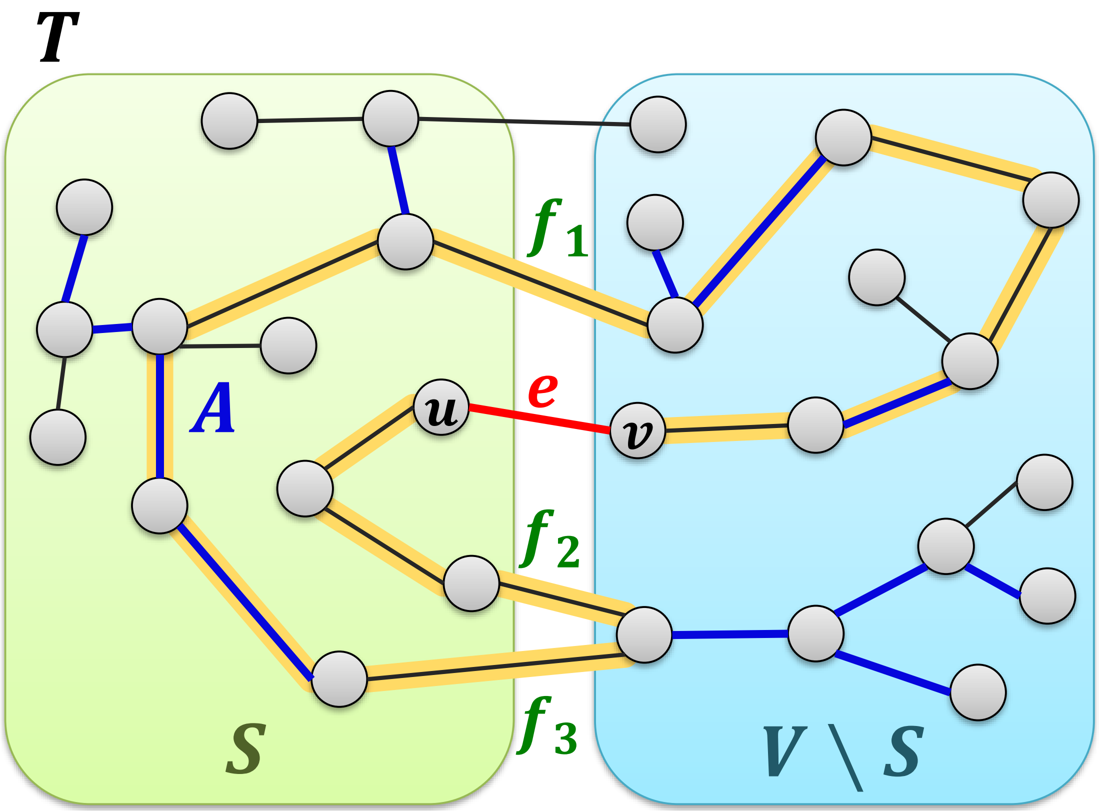
  - If G is connected : $\exists v \in V : \deg(v) = 1$



Sichere Kanten

Theorem: Let $(S, V \setminus S)$ be a cut s.t. A does not contain any cut edges and let $\{u, v\}$, $u \in S$, $v \in V \setminus S$ be a light cut edge w.r.t. $(S, V \setminus S)$. Then, $\{u, v\}$ is a safe edge for A.

Proof: Consider an MST T that contains the edges in A.



Cut edge f_i on u-v path

- $T' := (V, E \setminus \{f_i\} \cup \{e\})$
  $\Longrightarrow T'$ is connected.
- T' has $n-1$ edges
  $\Longrightarrow T'$ is a tree.
- e light edge
  $\Longrightarrow w(e) \leq w(f_i)$
- $w(T') \leq w(T)$
  $\Longrightarrow T'$ is an MST that contains A and e.

Prim's MST Algorithm

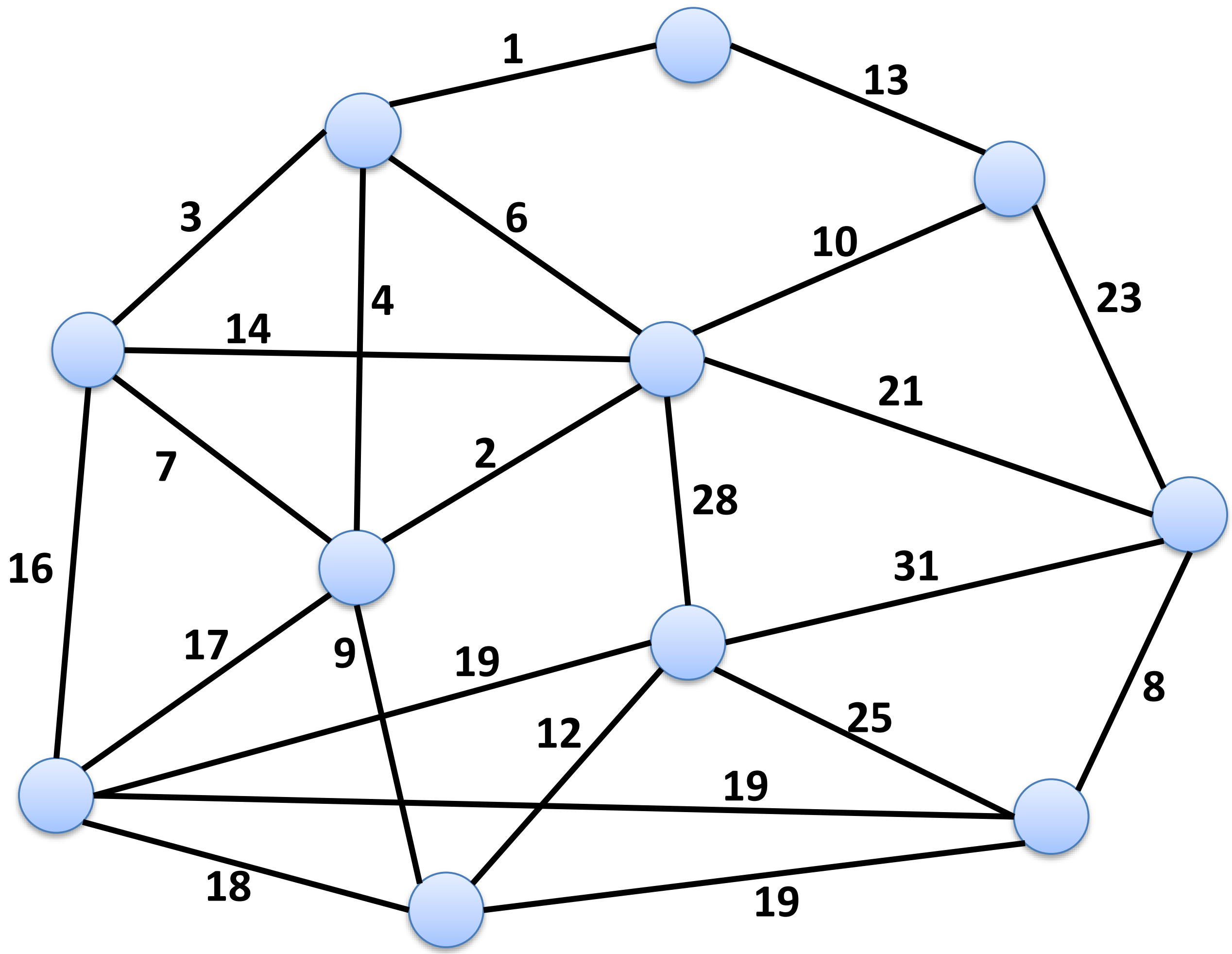
- Should be called Jarník's algorithm
  - was discovered by Prim in 1957 and published by Jarník already in 1930

- A possible implementation of the generic algorithm

  A = $\emptyset$
  while A is not a spanning tree **do**
      Find a safe edge $\{u, v\}$ for A
      A = $A \cup \big\{\{u, v\}\big\}$
  return A

- **Idea:** A is always a connected subtree
  - Start with an arbitrary node $s \in V$
  - Tree grows from s by always adding a light edge of the cut that is induced by the set of nodes that are already connected by the edges in A.

Prim's MST Algorithm: Example



Prim's MST Algorithm

```
S := {s};  A := ∅
while (S, A) is not a spanning tree do
    e = {u, v} is an edge with minimum weight,
    such that u ∈ S and v ∉ S
    S := S ∪ {v};  A := A ∪ {{u, v}}
```

We need to show that e is a safe edge for A.

- Follows directely because
  - S always exactly contains the nodes that are contained in some edge of A.
  - There therefore cannot be a cut edge of $(S, V \setminus S)$ in A.
  - $e = \{u, v\}$ is such an edge with smallest weight
  - The theorem from before therefore implies that e is a safe edge for A.

Implementation of Prim's Algorithm

- **Nodes in S are called marked**
  - These are exactly the nodes that are in the subtree defined by A.
- **A step of the algorithm:**
  - One looks for an edge with smallest weight to connect a marked node (a node in S) with an unmarked node.
  - This edge can in principle connect any unmarked node $u \in V \setminus S$ with any marked node in S.
- **Nodes $u \in V \setminus S$:**
  - $\alpha(u)$ is the closest neighbor of u in the subtree defined by the edges in A.
  - $d(u) = \text{dist}\big(u, \alpha(u)\big)$
    - $d(u) = \infty$ if u has no neighbor in $V \setminus S$
  - We thus always look for a node $u \in V \setminus S$ with smallest $d(u)$ and add the edge $\{u, \alpha(u)\}$ to A.
  - For this, the values $d(u)$ have to be updated after every step.

Implementation of Prim's Algorithm

- Nodes in S are marked
- Node u in $V \setminus S$:
  - $\alpha(u)$ is the closest neighbor of u in S (if defined)
  - $d(u) = \mathrm{dist}\big(u, \alpha(u)\big)$ (or $d(u) = \infty$ if $u \notin S$ or $\alpha(u) = \mathrm{NULL}$)

```
for all u ∈ V \ {s} do
    u.marked = false; d(u) = ∞; α(u) = NULL
d(s) = 0; A = ∅                         // We start at node s
while there are unmarked nodes do
    u = unmarked node with minimal d(u)
    for all unmarked neighbors v of u do
        if w({u,v}) < d(v) then
            α(v) = u; d(v) = w({u,v})
    u.marked = true
    if u ≠ s then A = A ∪ {{u, α(u)}}
```

Abstract Data Types : Priority Queue

Heap / Priority Queue:

- Manages a set of *(key,value) pairs*

Operations:

- *create()* : creates an empty heap
- *H.insert(x, key)* : inserts element *x* with key *key*
- *H.getMin()* : returns element with smallest key
- *H.deleteMin()* : deletes element with smallest key
  - *H.getMin()* and *H.deleteMin()* need to be consistent
- *H.decreaseKey(x, newkey)* : If *newkey* is smaller than the key of *x*, the key is changed from *x* to *newkey*

Implementation of Prim's Algorithm

```
H = new priority queue; A = ∅
for all u ∈ V \ {s} do
    H.insert(u, ∞); α(u) = NULL
H.insert(s, 0)

while H is not empty do
    u = H.deleteMin()
    for all unmarked neighbors v of u do
        if w({u,v}) < d(v) then
            H.decreaseKey(v, w({u,v}))
            α(v) = u
    u.marked = true
    if u ≠ s then A = A ∪ {{u, α(u)}}
```

Analysis of Prim's MST Algorithm

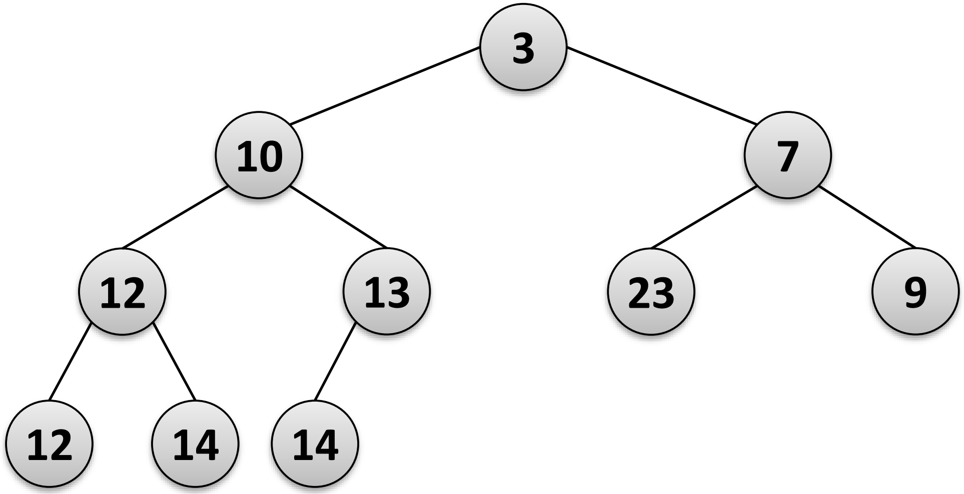
Number of priority queue operations

- **create** $\mathbf{1}$
- **insert** $\boldsymbol{O(n)}$ (every node exactly once)
- **getMin / deleteMin** $\boldsymbol{O(n)}$ (every node exactly once)
- **decreaseKey** $\boldsymbol{O(m)}$ (for every edge at most once, when the first node of the edge is added to S)

Priority Queues

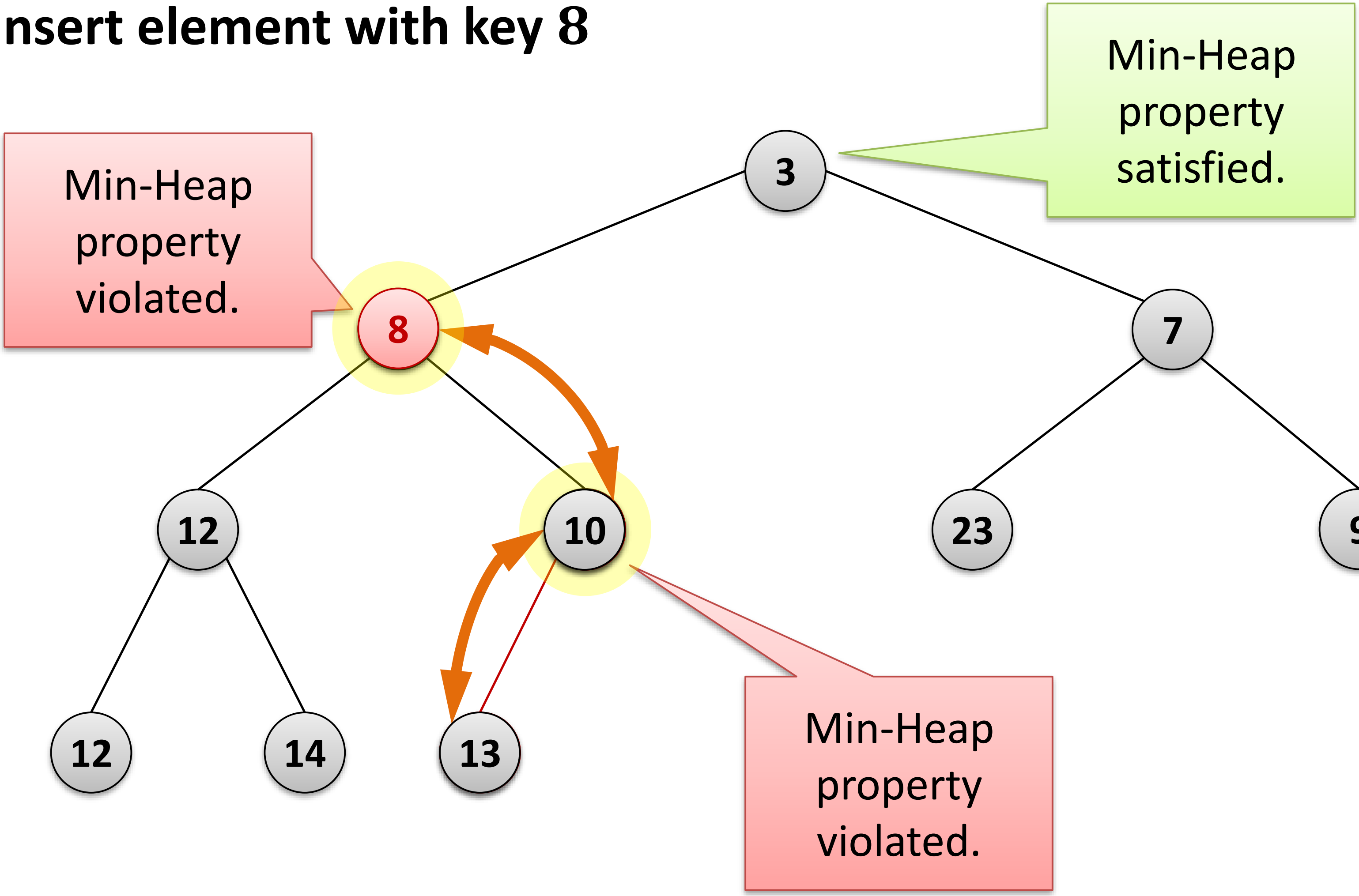
Implementation as a binary tree with the min-heap property

- This data structure is often also called a heap
- A tree has the min-heap property if in every subtree, the root has the smallest key.
- getMin operation: trivial!
- Tree is always as balanced as possible
  - All except for the bottom level is full.
  - The bottom-most level are filled from left to right.



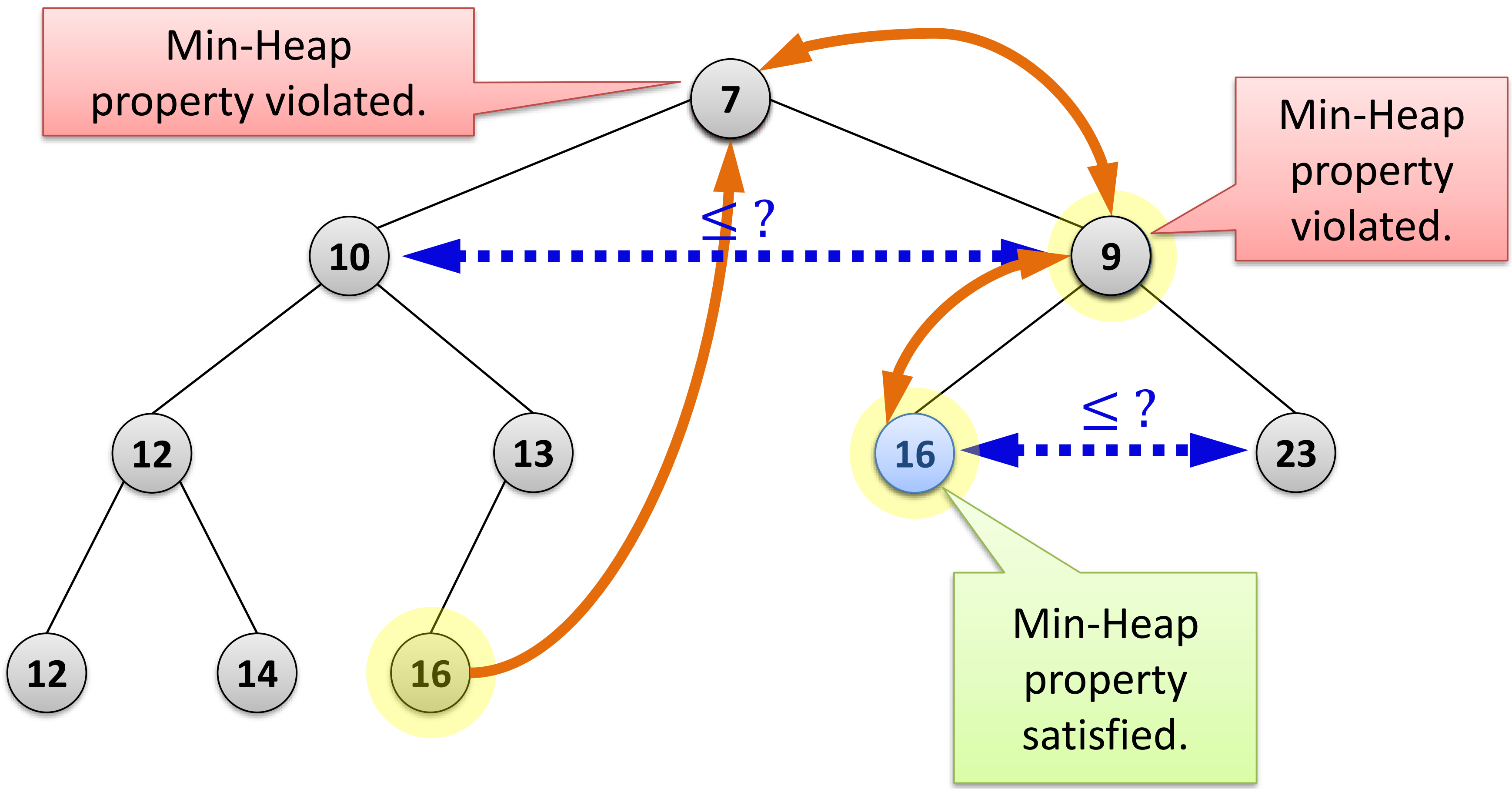
Priority Queues : Insert

Insert element with key 8



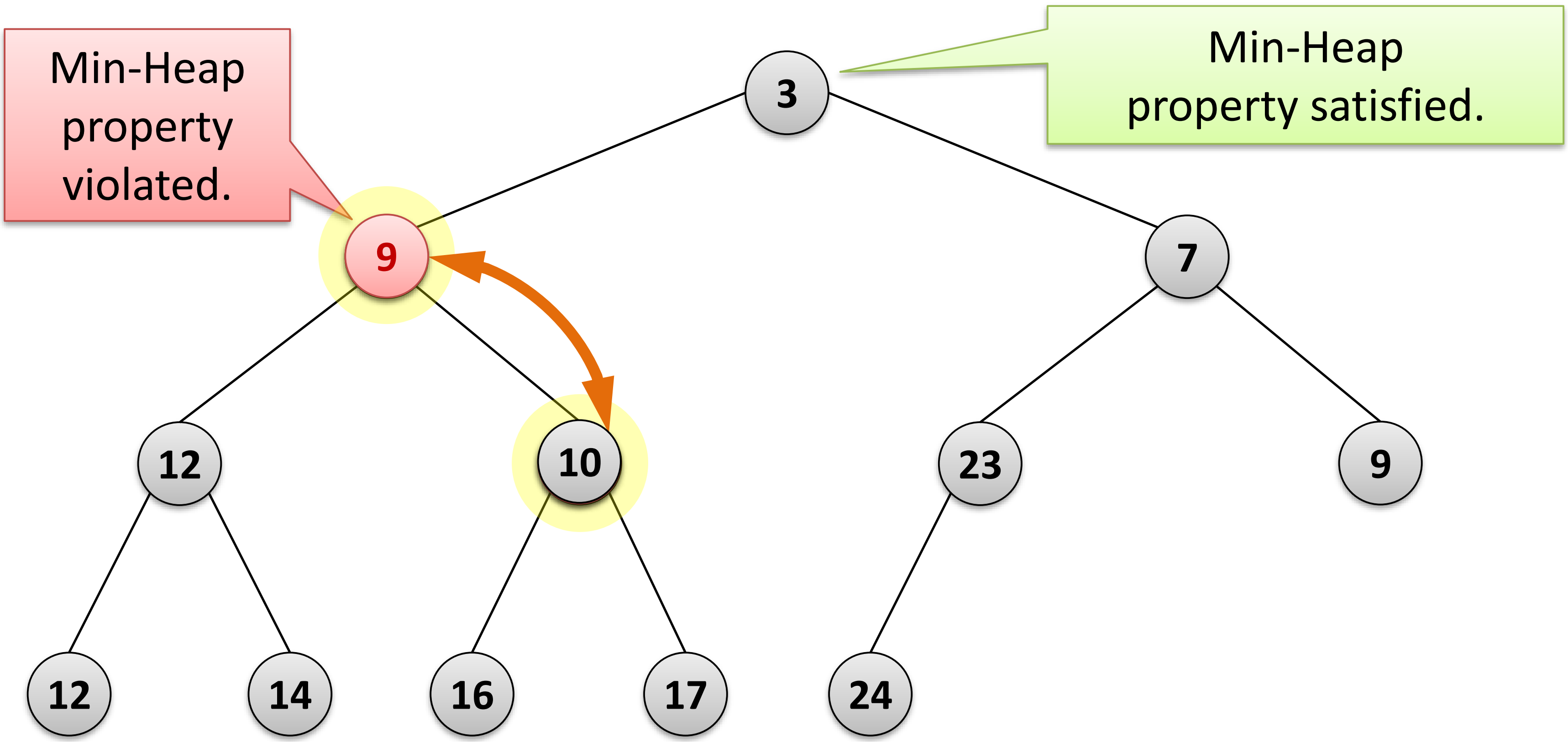
Priority Queues : Delete-Min

Delete element at the root (with minimum key)



Priority Queues : Decrease-Key

Decrease Key: Node with key 13 $\Longrightarrow$ new key 9



For the decrease-key operation, one needs to have a reference to the node of which the key has to be decreased.

Priority Queues : Analysis

- The discussed variant is also called a **binary heap**
  - durch einen Binärbaum mit Min-Heap-Eigenschaft implementiert
- **Height (or depth) of the tree** is always exactly $\lfloor \log_2 n \rfloor$
  - Number of nodes in a full binary tree of height is $2^{i+1} - 1$

    Number nodes at distance j from the root is 2^j:

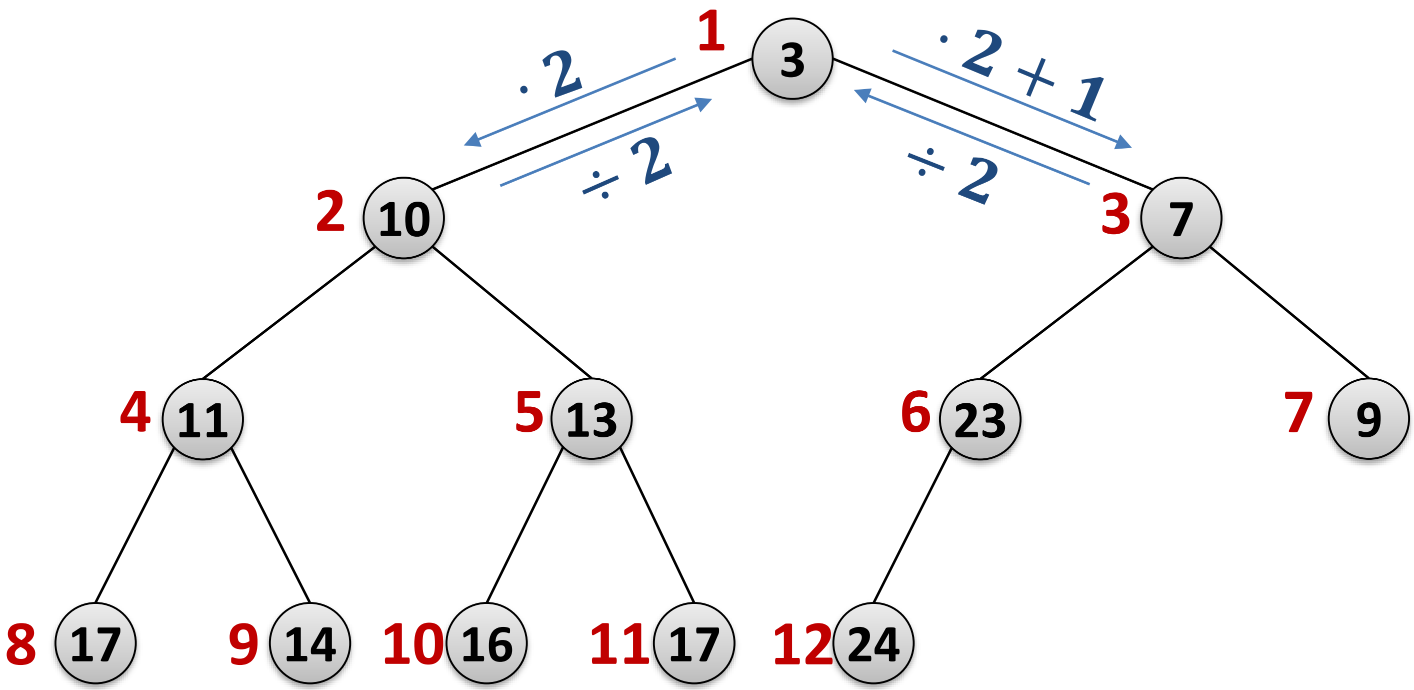
$$\#\text{nodes} = \sum_{j=0}^{i} 2^j = 2^{i+1} - 1.$$

- **Running time of all operations: $O(\log n)$**
  - If the binary tree if somehow implemented in a reasonable way.
  - One only needs to go up the tree once (for insert, decreaseKey) or down (for deleteMin)
  - We will next see an elegant way of implementing binary heaps

Binary Heaps : Array Implementation

Idea: 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., $j = \lfloor i/2 \rfloor$)

Heapsort

- The array implementation of heaps (priority queues) provides another very efficient sorting algorithm.

Heapsort (H is a binary heap, sort array A)

```
H = new BinaryHeap()
for i = 0 to n - 1 do
    H.insert(A[i])
for i = 0 to n - 1 do
    A[i] = H.deleteMin()
```

- **Running time:** $O(n \log n)$

Prim's Algorithm with Binary Heaps

```
H = new BinaryHeap(); A = ∅
for all u ∈ V \ {s} do
    H.insert(u, ∞); α(u) = NULL
H.insert(s, 0)
while H is not empty do
    u = H.deleteMin()
    for all unmarked neighbors v of u do
        if w({u,v}) < d(v) then
            H.decreaseKey(v, w({u,v})); α(v) = u
    u.marked = true
    if u ≠ s then A = A ∪ {{u, α(u)}}
```

Running time: $O(m \cdot \log n)$

- n insert operations and deleteMin operations
- $\leq m$ decreaseKey operations

Prim's Algorithm without Decrease-Key

```
H = new BinaryHeap(); A = ∅
for all u ∈ V \ {s} do
    H.insert(u, ∞); α(u) = NULL
H.insert(s, 0)
while H is not empty do
    u = H.deleteMin()
    if not u.marked then
        for all unmarked neighbors v of u do
            if w({u,v}) < d(v) then
                H.insert(v, w({u,v})); α(v) = u
        u.marked = true
        if u ≠ s then A = A ∪ {{u,α(u)}}
```

Running time: $O(m \cdot \log n)$

- $O(m)$ insert operations and deleteMin operations

Prim's Algorithmus: Better Running Time

Running time with binary heaps: $\boldsymbol{O(m \cdot \log n)}$

- $n \leq m + 1$ insert operations and deleteMin operations
- $\leq m$ decreaseKey operations

Best implementation of priority queues:

- Fibonacci Heaps (see algorithm theory lecture)
- Running time of operations (deleteMin, decreaseKey amortized)

  insert: $\boldsymbol{O(1)}$, deleteMin: $\boldsymbol{O(\log n)}$, decreaseKey: $\boldsymbol{O(1)}$

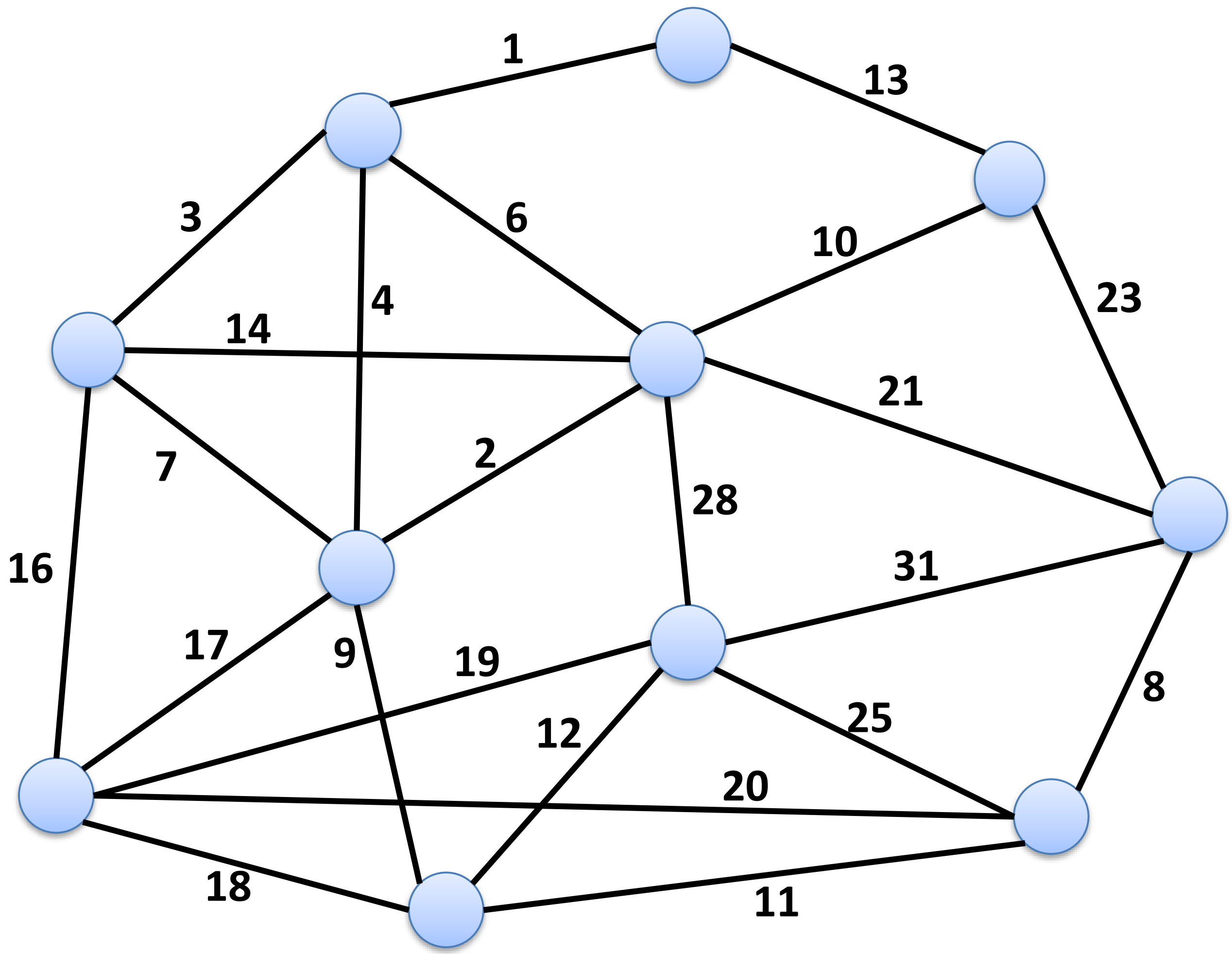
Running time with Fibonacci heaps: $\boldsymbol{O(m + n \cdot \log n)}$

- $n \leq m + 1$ insert operations and deleteMin operations
- $\leq m$ decreaseKey operations
  (in this case, Prim needs to be implemented with decrease-key)

Kruskal's MST Algorithm

```
A = ∅
while A is not a spanning tree do
    e = {u, v} is an edge with smallest weight
    s.t. A ∪ {{u, v}} does not contain a cycle
    A = A ∪ {{u, v}}
```

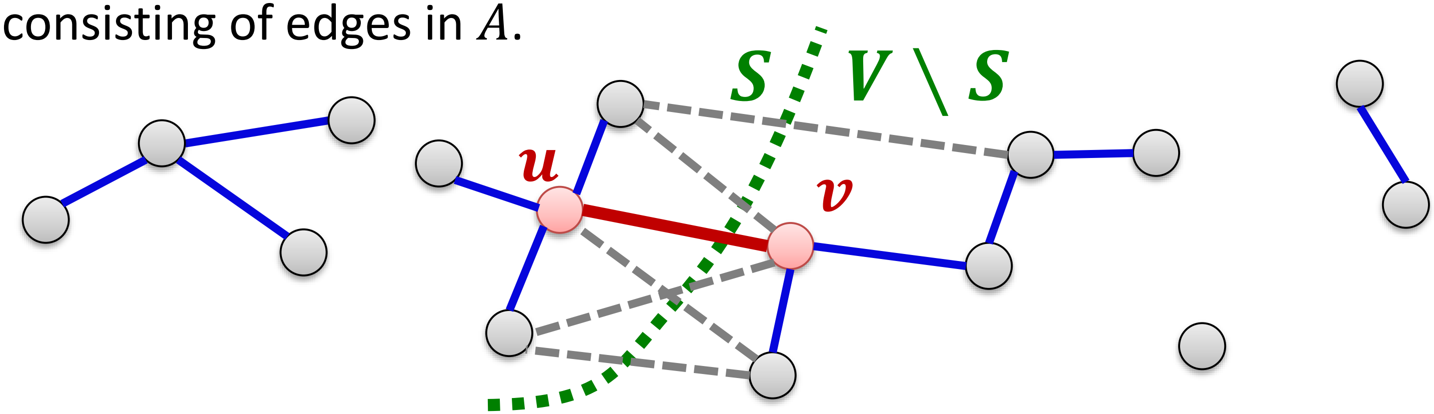
Kruskal's MST Algorithm: Example



Kruskal's MST Algorithm

```
A = ∅
while A is not a spanning tree do
    e = {u,v} is an edge with smallest weight
    s.t. A ∪ {{u,v}} does not contain a cycle
    A = A ∪ {{u,v}}
```

- We have to show that e is a safe edge for A
  - As $A \cup \{\{u,v\}\}$ is cycle-free, u and v are not connected through a path consisting of edges in A.



  - There is a cut $(S, V \setminus S)$ s. t. A does not contain any cut edges, s. t. $u \in S$ and $v \in V \setminus S$, and s. t. $\{u,v\}$ is a light cut edge.

Implementation of Kruskal's Algorithm

Kruskal's Algorithm (Pseudocode)

1. A = $\emptyset$
2. Sort edges by edge weight
3. **for** e = $\{u, v\} \in E$ (in sorted order) **do**
4.   **if** u and v are in different components **then**
5. A = $A \cup \{e\}$

- We must manage the connected componenten of the graph defined by the edges in A efficiently
- **Running time:** $O(m \log n)$ for sorting and the overall running time for managing the components…

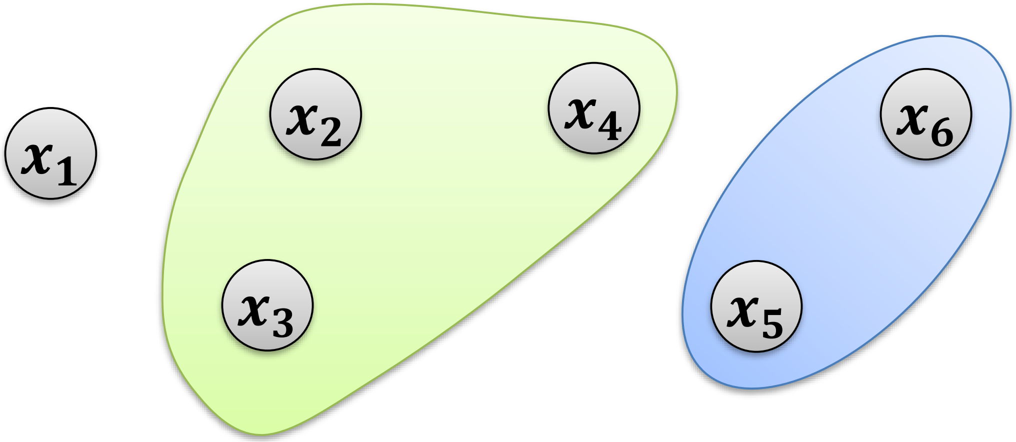
Abstract Data Types : Union-Find

Union-Find / Disjoint Sets:

- Manages a partition of elements

Operationen:

- *create()* : creates an empty union-find data structure
- *U.makeSet(x)* : adds set $\{x\}$ to the partition
- *U.find(x)* : returns the set of element x
- *U.union(S1, S2)* : merges sets $S1$ and $S2$ to set $S1 \cup S2$



Kruskal with Union-Find

Kruskal's Algorithm

$A = \emptyset$
U = create new
for all $u \in V$ **do**
    U.makeSet(u)
Sort edges by edge weight
for all $e = \{u, v\} \in E$ (in sorted order) **do**
    S_u = U.find(u); S_v = U.find(v)
    if $S_u \neq S_v$ **then**
        $A = A \cup \{e\}$
        U.union(S_u, S_v)

Kruskal: Running Time Analysis

Best Union-Find Data Structure

- Running time for m union-find operations on n elements (n makeSet operations):

$$\boldsymbol{O\big(m \cdot \alpha(n)\big)}$$

- $\alpha(n)$ is the inverse of the Ackermann function and grows extremaly slowly (for all practically relevant n, $\alpha(n) \leq 5$)

Running Time Kruskal

- Sort edges: $O(m \cdot \log n)$
  - If the weights are integers from $0, \dots, n^{O(1)}$, the edges can be sorted with radix sort in linear time.
- Union-Find operations: $O\big(m \cdot \alpha(n)\big)$
- Overall: $O(m \cdot \log n)$
  - Better if the edges can be sorted faster

Prim and Kruskal: Remarks

Both algorithms are typical examples for so-called

greedy algorithms

- In greedy algorithms, a solution is built in a step-by-step manner.
- In each step, the current best "element" is added to the solution.
- Already changes parts of the solution are not altered any more.

Prim and Kruskal algorithms to compute an MST

- We start with an empty edge set.
- In each step, the currently best edge is added
  - For Prim: best edge that keeps the already added part connected
  - For Kruskal: best edge s.t. the set can still be extended to a spanning tree.
- A chosen edge is never discarded later.