



# **Chapter 4**

# **Data Structures**

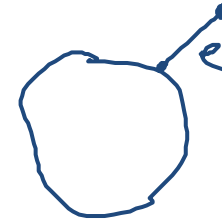
**Algorithm Theory**  
**WS 2014/15**

**Fabian Kuhn**

# Minimum Spanning Trees

**Given:** weighted graph

**Goal:** spanning tree with minimum total weight



## Prim 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

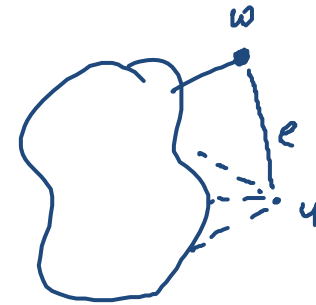
## Kruskal Algorithm:

1. Start with an empty edge set
2. In each step:  
Add minimum weight edge  $e$  such that  $e$  does not close a cycle

# Implementation of Prim 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  $s \geq v$  are unmarked  
*add all nodes to a priority queue*
3. Get unmarked node  $u$  which minimizes  $d(u)$ :



- get-min*

4. For all  $e = \{u, v\} \in E$ ,  $d(v)$  =  $\min\{\underline{d(v)}, \boxed{w(e)}\}$   
*if update needed: decrease-key*
5. mark node  $u$   
*delete-min*
6. Until all nodes are marked

# Implementation of Prim Algorithm

## Implementation with Fibonacci heap:

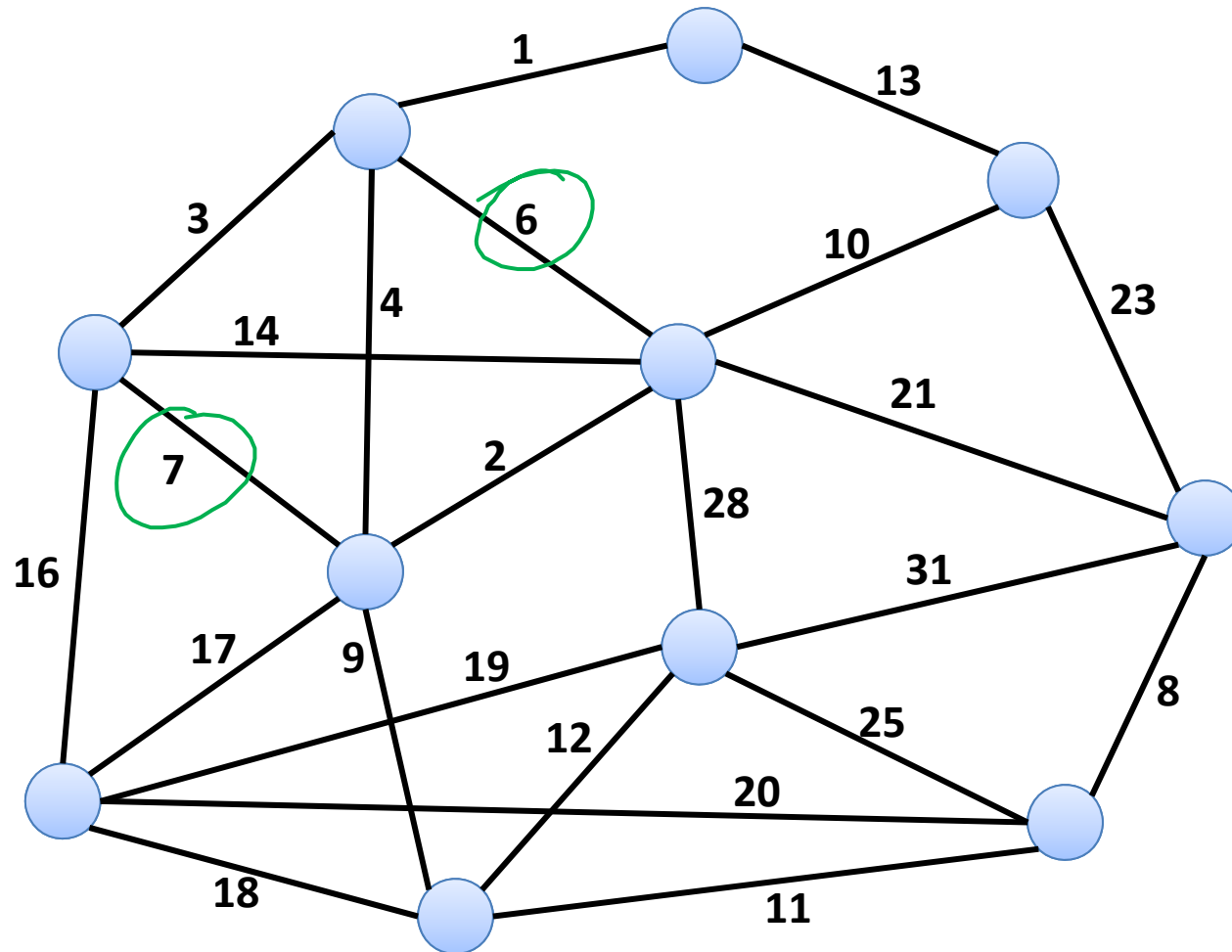
- Analysis identical to the analysis of Dijkstra's algorithm:

$O(n)$   <sup>$O(1)$</sup>  insert and  <sup>$O(\log n)$</sup>  delete-min operations  
 $O(m)$   <sup>$O(1)$</sup>  decrease-key operations

*amortized*

- Running time:  $O(m + n \log n)$

# Kruskal Algorithm



1. Start with an empty edge set
2. In each step: Add minimum weight edge  $e$  such that  $e$  does not close a cycle

# Implementation of Kruskal Algorithm

1. Go through edges in order of increasing weights

sort edges by weight  $O(m \log n)$  time

2. For each edge  $e$ :

**if  $e$  does not close a cycle then**

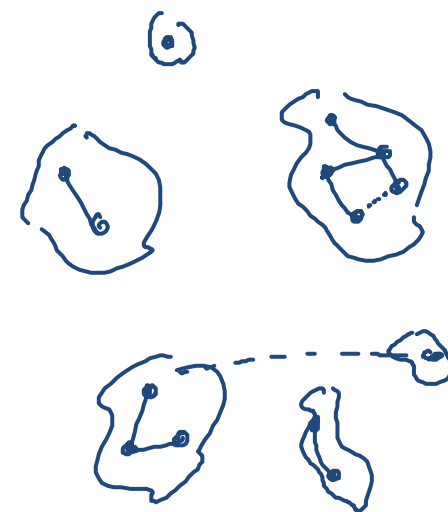
need to check whether  $e = \{u, v\}$  closes a cycle

need to check whether  $u$  and  $v$  are in the same component

**add  $e$  to the current solution**

add edge  $e = \{u, v\}$

need to merge components of  $u$  and  $v$

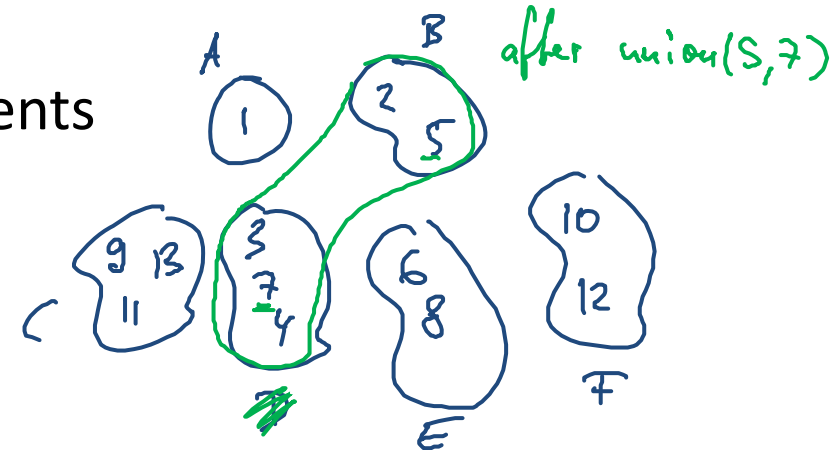


# Union-Find Data Structure

Also known as Disjoint-Set Data Structure...

Manages partition of a set of elements

- set of disjoint sets



**Operations:**

- **make\_set( $x$ ):** create a new set that only contains element  $x$
- **find( $x$ ):** return the set containing  $x$
- **union( $x, y$ ):** merge the two sets containing  $x$  and  $y$

# Implementation of Kruskal Algorithm

1. Initialization:  
For each node  $v$ : make\_set( $v$ )
2. Go through edges in order of increasing weights:  
Sort edges by edge weight
3. For each edge  $e = \{u, v\}$ :  
**if find( $u$ )  $\neq$  find( $v$ ) then**  
    add  $e$  to the current solution ←  
    **union( $u, v$ )**

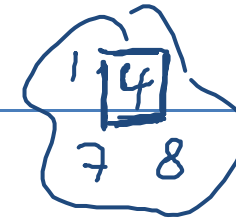


# Managing Connected Components

---

- Union-find data structure can be used more generally to manage the connected components of a graph
  - ... if edges are added incrementally
- **make\_set( $v$ )** for every node  $v$
- **find( $v$ )** returns component containing  $v$
- **union( $u, v$ )** merges the components of  $u$  and  $v$   
(when an edge is added between the components)
- Can also be used to manage biconnected components

# Basic Implementation Properties



## Representation of sets:

- Every set  $S$  of the partition is identified with a representative, by one of its members  $x \in S$

## Operations:

- **make\_set( $x$ )**:  $x$  is the representative of the new set  $\{x\}$
- **find( $x$ )**: return representative of set  $S_x$  containing  $x$
- **union( $x, y$ )**: unites the sets  $S_x$  and  $S_y$  containing  $x$  and  $y$  and returns the new representative of  $S_x \cup S_y$

# Observations

---

## Throughout the discussion of union-find:

- $n$ : total number of make\_set operations
- $m$ : total number of operations (make\_set, find, and union)

## Clearly:

- $m \geq n$
- There are at most  $n - 1$  union operations

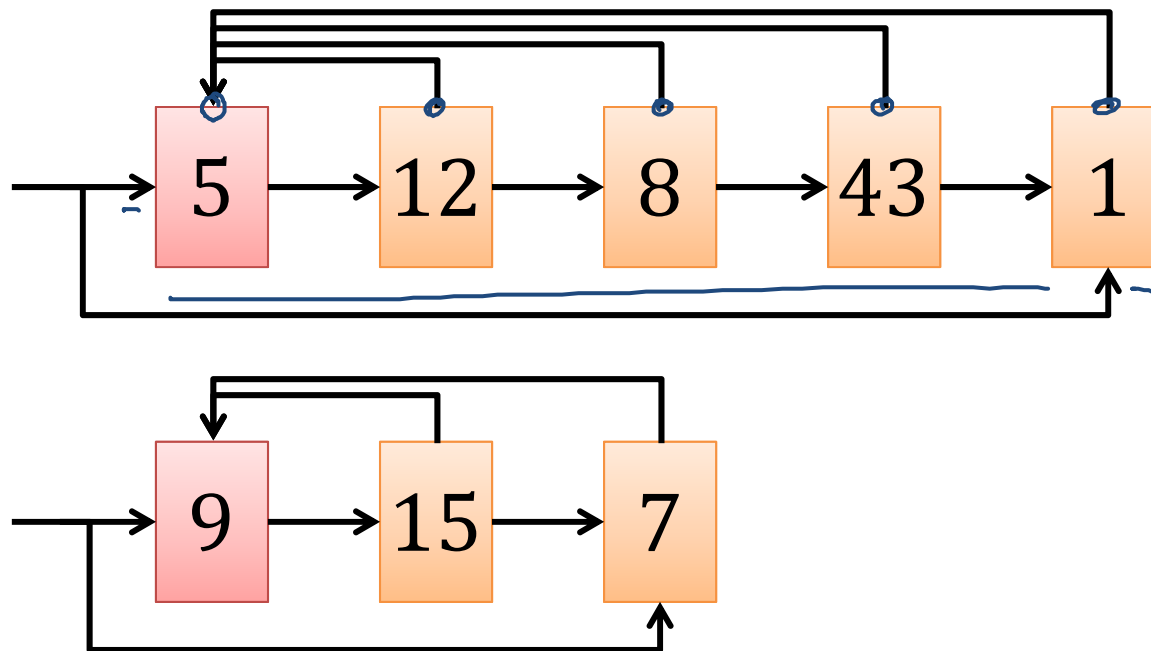
## Remark:

- We assume that the  $n$  make\_set operations are the first  $n$  operations
  - Does not really matter...

# Linked List Implementation

Each set is implemented as a linked list:

- representative: first list element (all nodes point to first elem.)  
in addition: pointer to first and last element



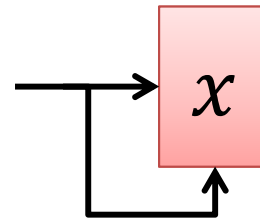
- sets:  $\{1,5,8,12,43\}$ ,  $\{7,9,15\}$ ; representatives: 5, 9

# Linked List Implementation

## make\_set( $x$ ):

- Create list with one element:

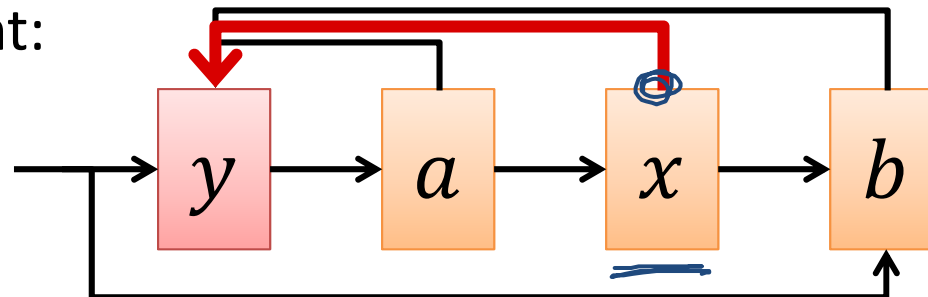
**time:  $O(1)$**



## find( $x$ ):

- Return first list element:

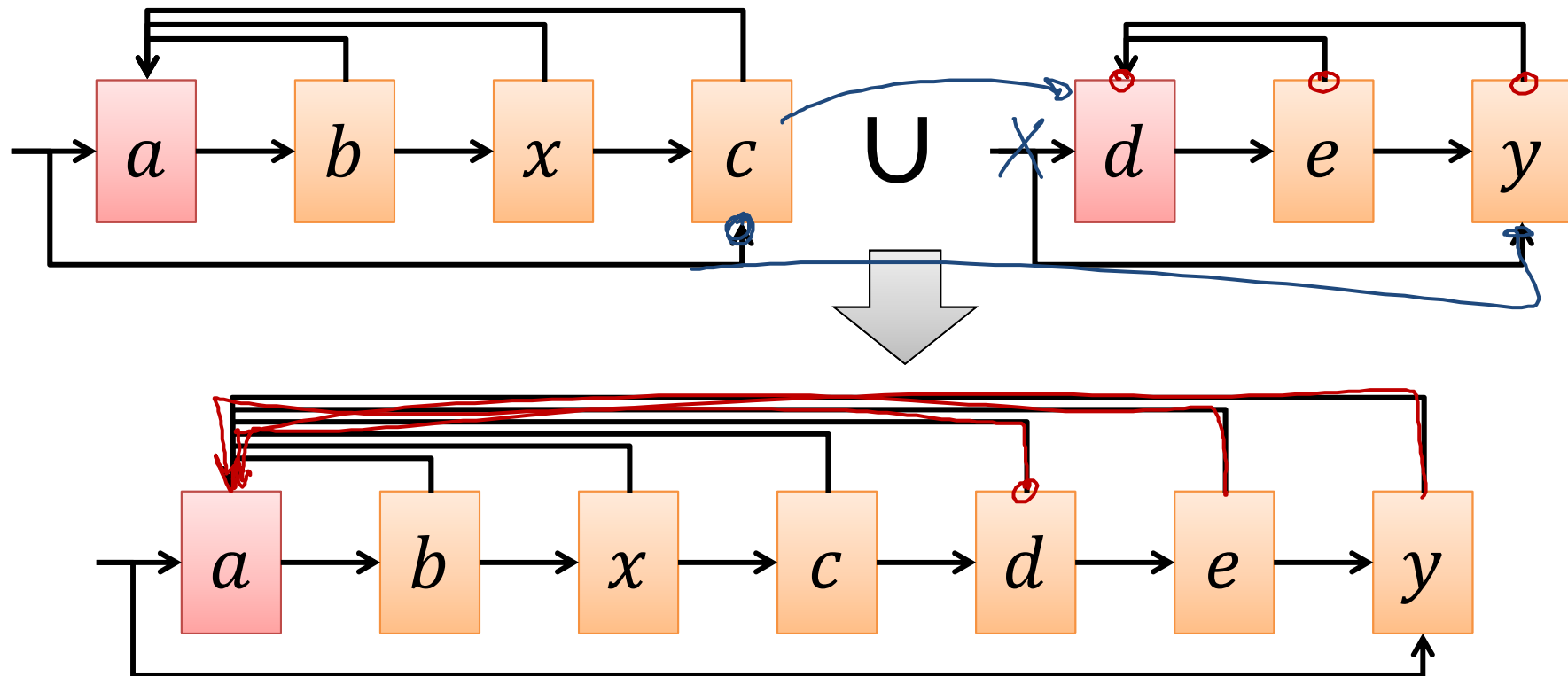
**time:  $O(1)$**



# Linked List Implementation

**union(x, y):**

- Append list of  $y$  to list of  $x$ :



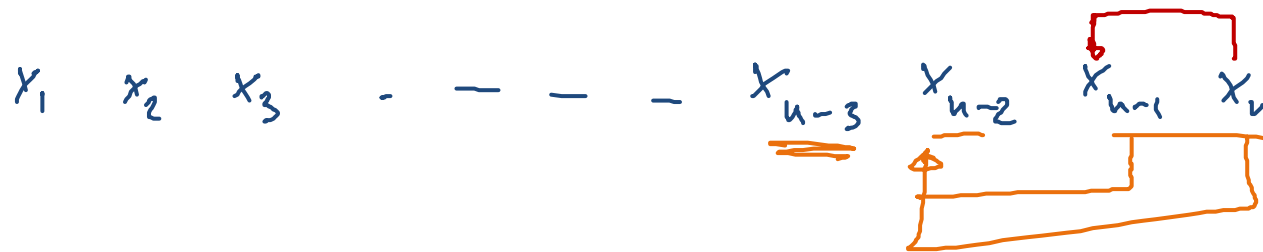
**Time:  $O(\text{length of list of } y)$**

# Cost of Union (Linked List Implementation)



Total cost for  $n - 1$  union operations can be  $\Theta(n^2)$ :

- $\text{make\_set}(x_1), \text{make\_set}(x_2), \dots, \text{make\_set}(x_n),$   
 $\text{union}(x_{n-1}, x_n), \text{union}(x_{n-2}, x_{n-1}), \dots, \text{union}(x_1, x_2)$



$$\# \text{ pointer redirections: } 1 + 2 + 3 + \dots + n - 1 = \Theta(n^2)$$

# Weighted-Union Heuristic

- In a bad execution, **average cost per union** can be  $\Theta(n)$
- Problem: The longer list is always appended to the shorter one

## Idea:

- In each union operation, append shorter list to longer one!

Cost for union of sets  $S_x$  and  $S_y$ :  $O(\min\{|S_x|, |S_y|\})$

**Theorem:** The overall cost of  $m$  operations of which at most  $n$  are make\_set operations is  $O(m + n \log n)$ .



# Weighted-Union Heuristic

**Theorem:** The overall cost of  $m$  operations of which at most  $n$  are make\_set operations is  $O(m + n \log n)$ .

**Proof:**

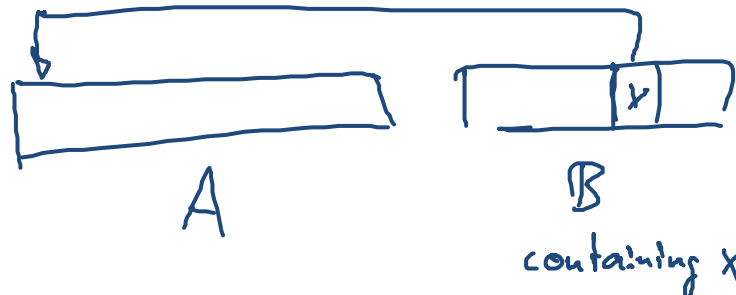
total cost of make-set & find op:  $O(m)$

need to bound cost of union operations:

= # of pointer redirections

considers a fixed element  $x$

How often do we need to redirect the pointer of  $x$ ?



Size of set containing  $x$  at least doubles

$\Rightarrow$  at  $\log_2 n$  redirections of pointer of  $x$

for MST algorithm  
 Sorting:  $O(m \log n)$   
union-find:  
 $O(m + n \log n)$