



# **Chapter 5**

# **Data Structures**

**Algorithm Theory**  
**WS 2019/20**

**Fabian Kuhn**

# Minimum Spanning Trees

- **Minimum spanning tree (MST)** problem
  - Classic graph-theoretic optimization problem
- **Given:** weighted graph
- **Goal:** spanning tree with min. total weight
- Several greedy algorithms work
- Kruskal's algorithm:
  - Start with empty edge set
  - As long as we do not have a spanning tree:  
**add minimum weight edge that doesn't close a cycle**

## 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
3. Get unmarked node  $u$  which minimizes  $d(u)$ :
4. For all  $e = \{u, v\} \in E$ ,  $d(v) = \min\{d(v), w(e)\}$
5. mark node  $u$
6. Until all nodes are marked

# Implementation of Prim Algorithm

## Implementation with Fibonacci heap:

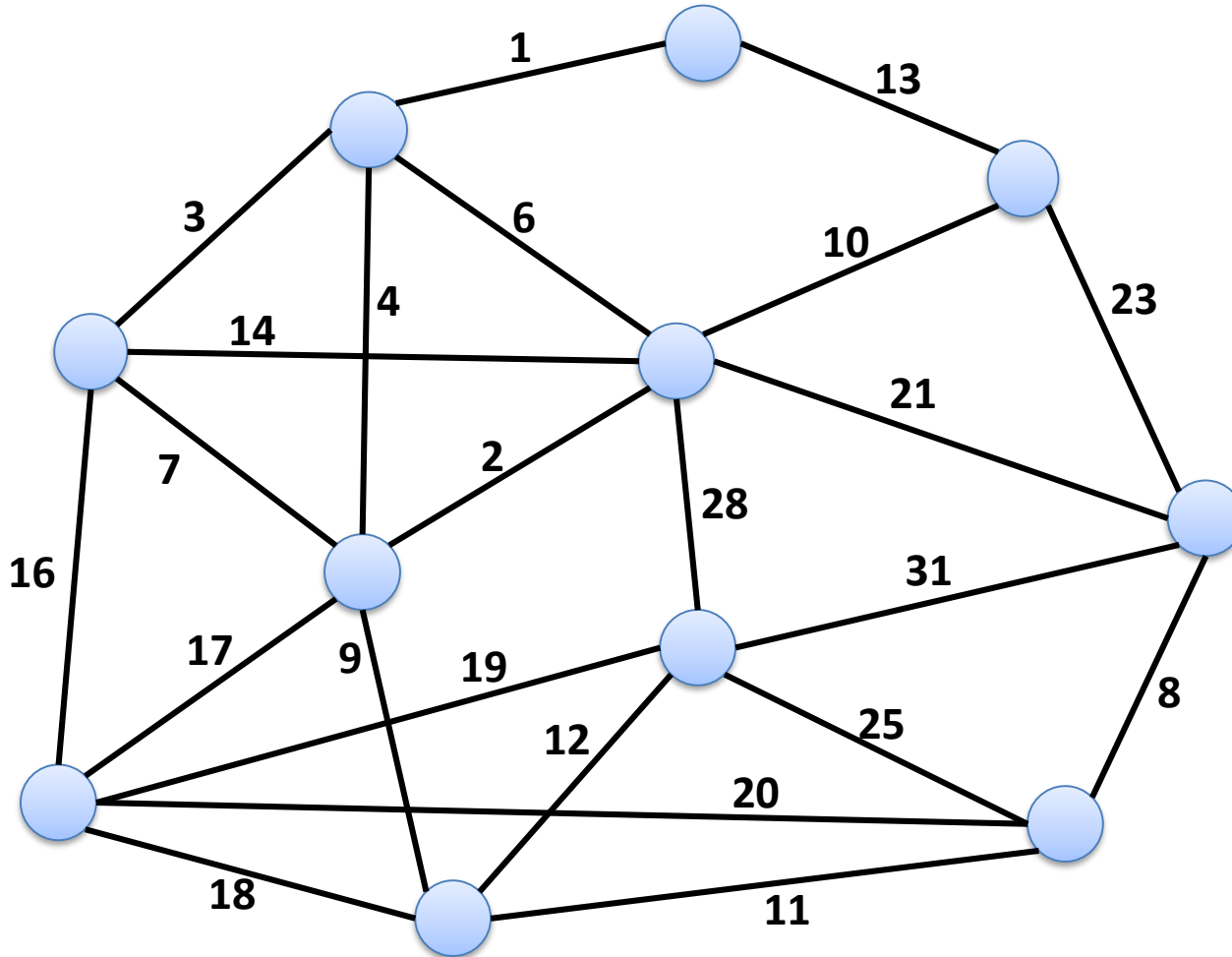
- Analysis identical to the analysis of Dijkstra's algorithm:

$O(n)$  insert and delete-min operations

$O(m)$  decrease-key operations

- 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

2. For each edge  $e$ :

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

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

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

# 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

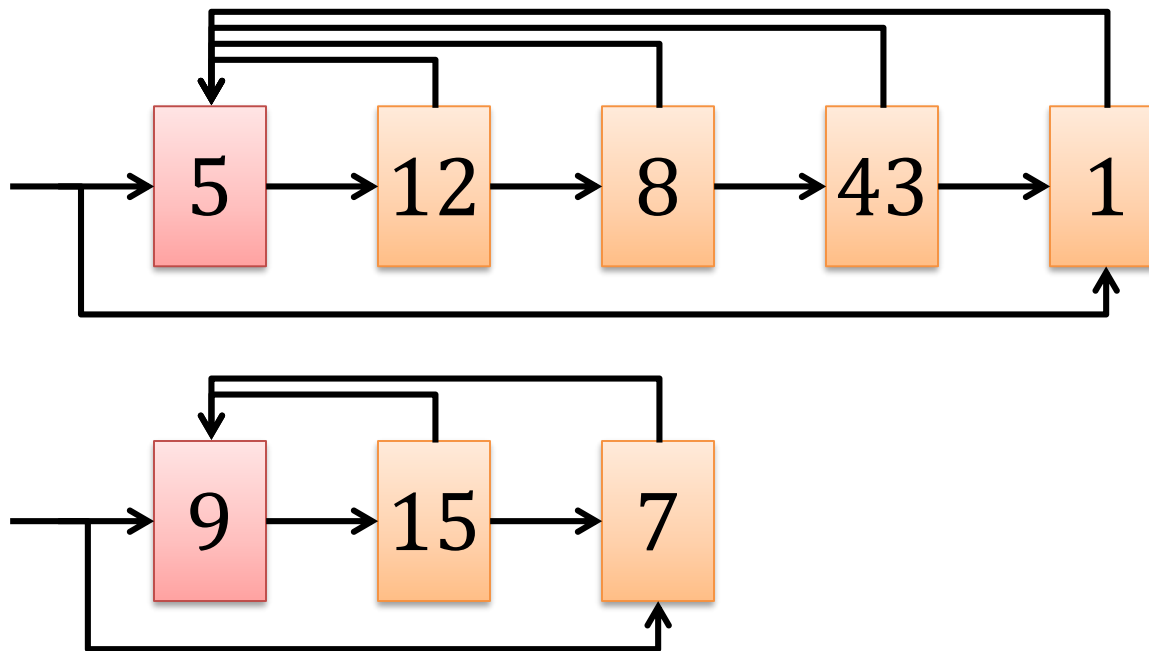
## Remark:

- 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



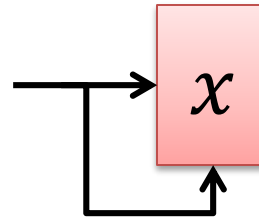
- sets:  $\{1,5,8,12,43\}$ ,  $\{7,9,15\}$ ; representatives: 5, 9

# Linked List Implementation

## **make\_set( $x$ ):**

- Create list with one element:

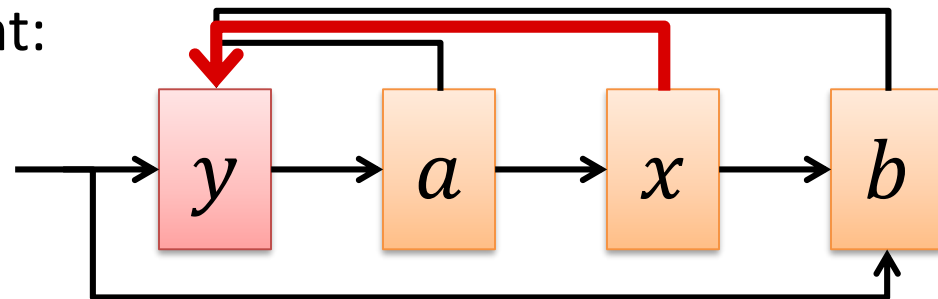
**time:  $O(1)$**



## **find( $x$ ):**

- Return first list element:

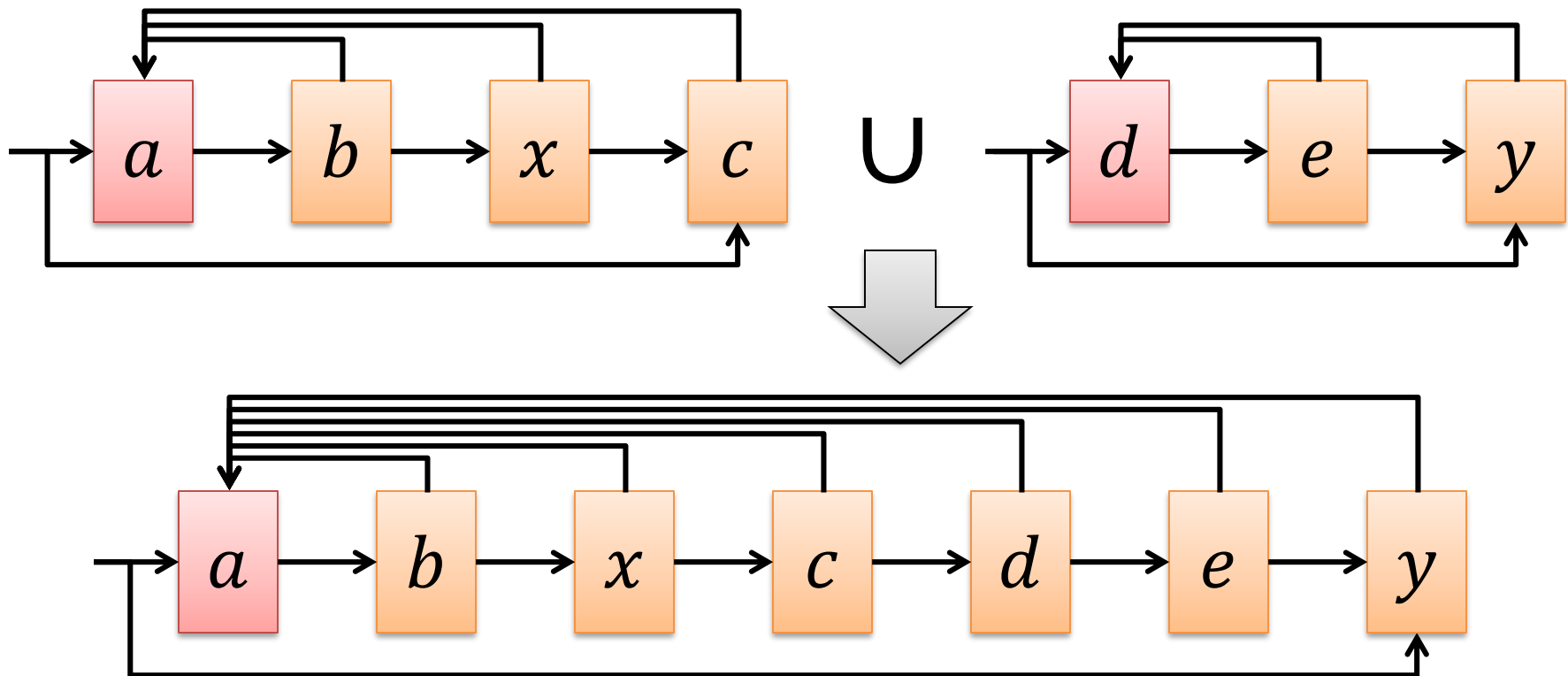
**time:  $O(1)$**



# 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  $\Theta(n^2)$ :

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



# 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  $S_y$ :  $O(\min\{|S_x|, |S_y|\})$

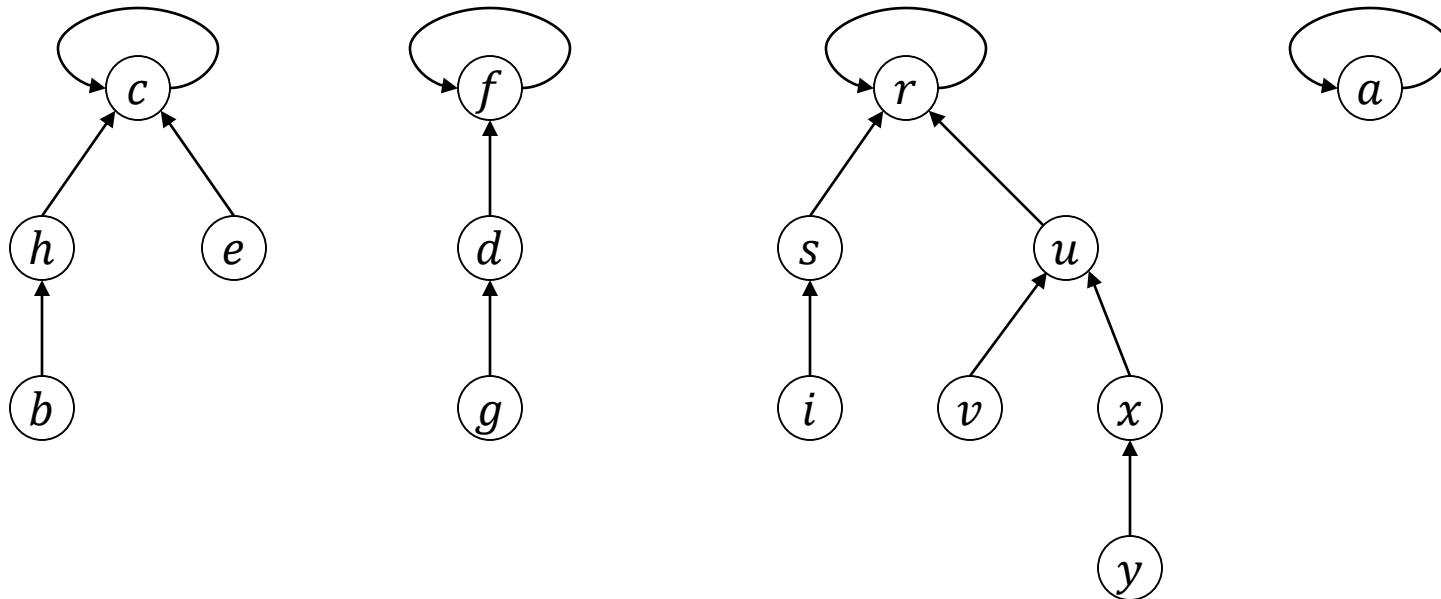
**Theorem:** The overall cost of  $m$  operations of which at most  $n$  are `make_set` operations is  **$O(m + n \log n)$** .

# 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:**

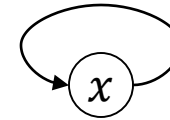
# Disjoint-Set Forests



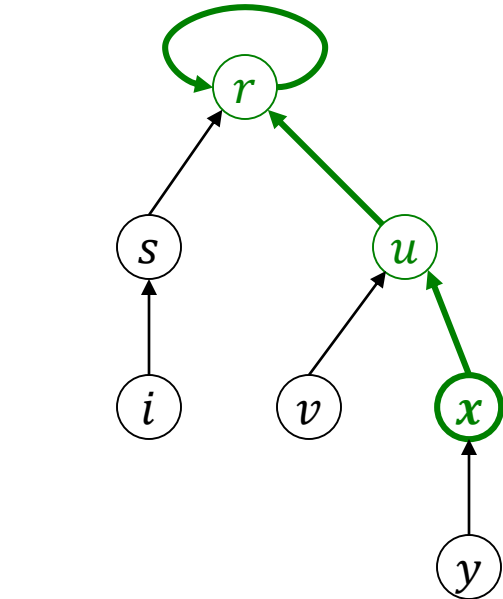
- Represent each set by a tree
- Representative of a set is the root of the tree

# Disjoint-Set Forests

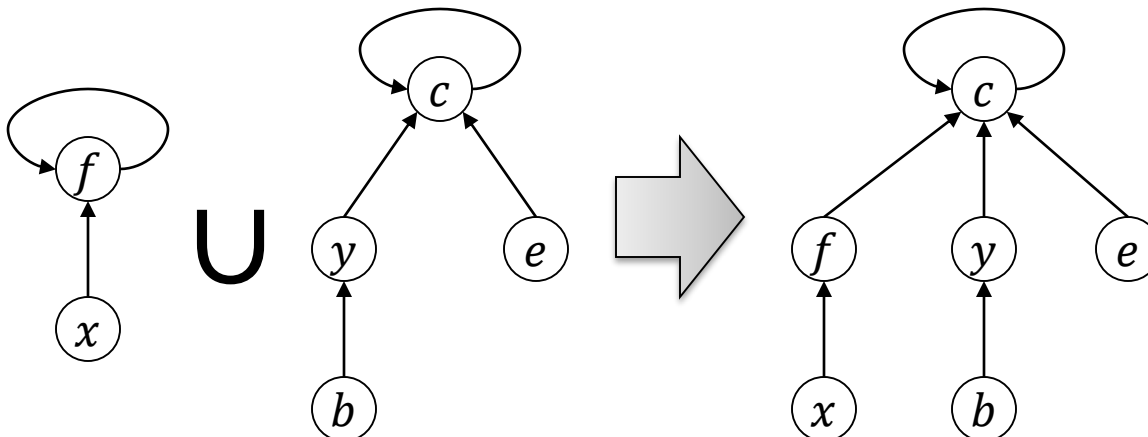
**make\_set(x)**: create new one-node tree



**find(x)**: follow parent pointer to root  
(parent pointer to itself)



**union(x, y)**: attach tree of x to tree of y



# Bad Sequence

Bad sequence leads to tree(s) of depth  $\Theta(n)$

- $\text{make\_set}(x_1), \text{make\_set}(x_2), \dots, \text{make\_set}(x_n),$   
 $\text{union}(x_1, x_2), \text{union}(x_1, x_3), \dots, \text{union}(x_1, x_n)$

# Union-By-Size Heuristic

## Union of sets $S_1$ and $S_2$ :

- Root of trees representing  $S_1$  and  $S_2$ :  $r_1$  and  $r_2$
- W.l.o.g., assume that  $|S_1| \geq |S_2|$
- **Root of  $S_1 \cup S_2$ :  $r_1$**  ( $r_2$  is attached to  $r_1$  as a new child)

**Theorem:** If the union-by-size heuristic is used, the **worst-case cost of a find-operation is  $O(\log n)$**

**Proof:**

**Similar Strategy:** **union-by-rank**

- rank: essentially the depth of a tree

# Union-Find Algorithms

Recall:  $m$  operations,  $n$  of the operations are `make_set`-operations

## Linked List with Weighted Union Heuristic:

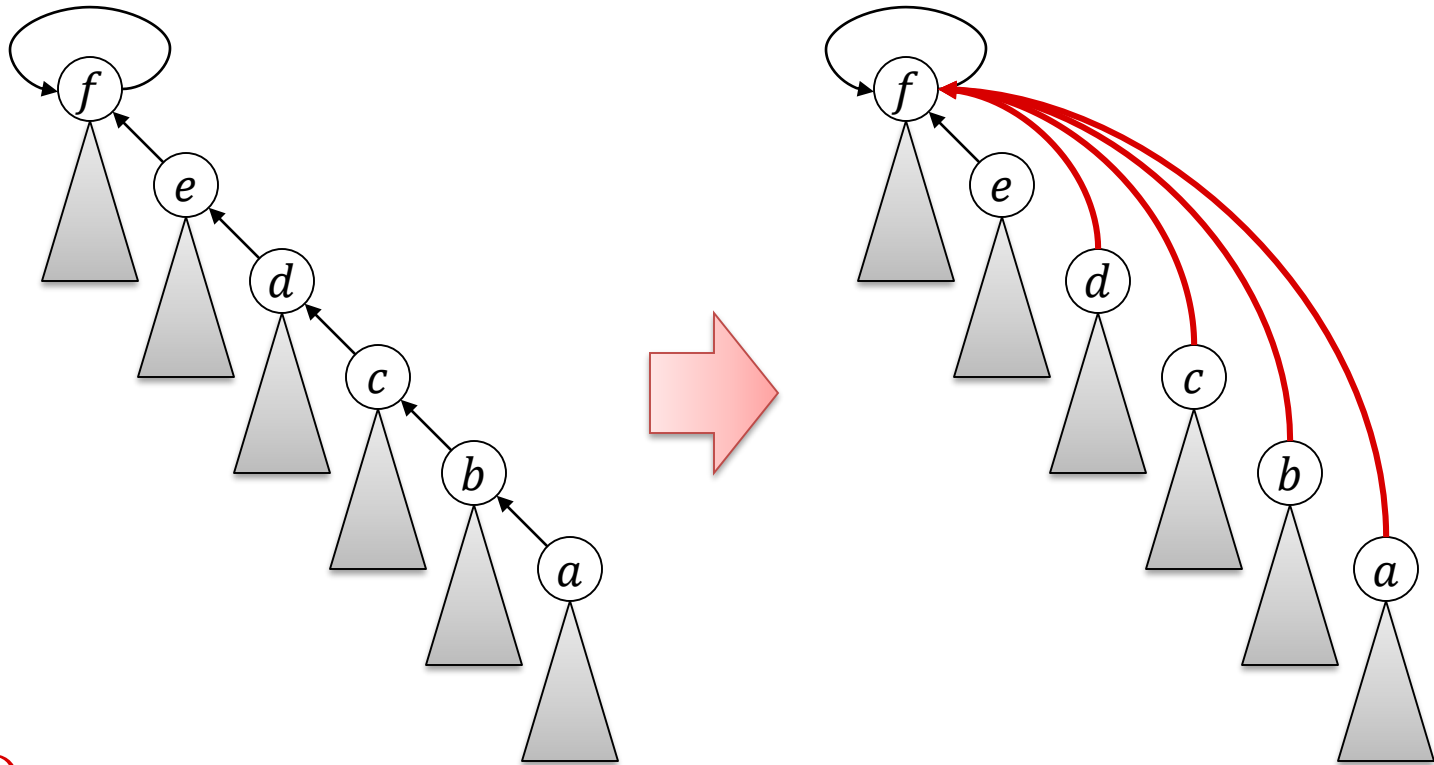
- `make_set`: **worst-case** cost  $O(1)$
- `find` : **worst-case** cost  $O(1)$
- `union` : **amortized** worst-case cost  $O(\log n)$

## Disjoint-Set Forest with Union-By-Size Heuristic:

- `make_set`: **worst-case** cost  $O(1)$
- `find` : **worst-case** cost  $O(\log n)$
- `union` : **worst-case** cost  $O(\log n)$

Can we make this faster?

# Path Compression During Find Operation



**find( $a$ ):**

1. **if**  $a \neq a.\text{parent}$  **then**
2.      $a.\text{parent} := \text{find}(a.\text{parent})$
3. **return**  $a.\text{parent}$



# Complexity With Path Compression

When using only path compression (without union-by-rank):

$m$ : total number of operations

- $f$  of which are find-operations
- $n$  of which are make\_set-operations  
→ at most  $n - 1$  are union-operations

**Total cost:**  $O\left(m + f \cdot \left\lceil \log_{2+f/n} n \right\rceil\right) = O\left(m + f \cdot \log_{2+m/n} n\right)$

# Union-By-Size and Path Compression

## Theorem:

Using the combined union-by-rank and path compression heuristic, the running time of  $m$  disjoint-set (union-find) operations on  $n$  elements (at most  $n$  make\_set-operations) is

$$\Theta(m \cdot \alpha(m, n)),$$

Where  $\alpha(m, n)$  is the inverse of the Ackermann function.

# Ackermann Function and its Inverse

## Ackermann Function:

For  $k, \ell \geq 1$ ,

$$A(k, \ell) := \begin{cases} 2^\ell, & \text{if } k = 1, \ell \geq 1 \\ A(k - 1, 2), & \text{if } k > 1, \ell = 1 \\ A(k - 1, A(k, \ell - 1)), & \text{if } k > 1, \ell > 1 \end{cases}$$

## Inverse of Ackermann Function:

$$\alpha(m, n) := \min\{k \geq 1 \mid A(k, \lfloor m/n \rfloor) > \log_2 n\}$$

# Inverse of Ackermann Function

- $\alpha(m, n) := \min\{k \geq 1 \mid A(k, \lfloor m/n \rfloor) > \log_2 n\}$   
 $m \geq n \Rightarrow A(k, \lfloor m/n \rfloor) \geq A(k, 1) \Rightarrow \alpha(m, n) \leq \min\{k \geq 1 \mid A(k, 1) > \log n\}$
- $A(1, \ell) = 2^\ell, \quad A(k, 1) = A(k - 1, 2),$   
 $A(k, \ell) = A(k - 1, A(k, \ell - 1))$