Chapter 4
Data Structures
Fibonacci Heaps, Union Find

Algorithm Theory
WS 2012/13

Fabian Kuhn

Fibonacci Heaps: Marks

Cycle of a node:

1. Node v is removed from root list and linked to a node
   $v.mark = \text{false}$

2. Child node u *of* v is cut and added to root list
   $v.mark = \text{true}$

3. Second child of v is cut
   node v is cut as well and moved to root list

The boolean value $v.mark$ indicates whether node v has lost a child since the last time v was made the child of another node.

Potential Function

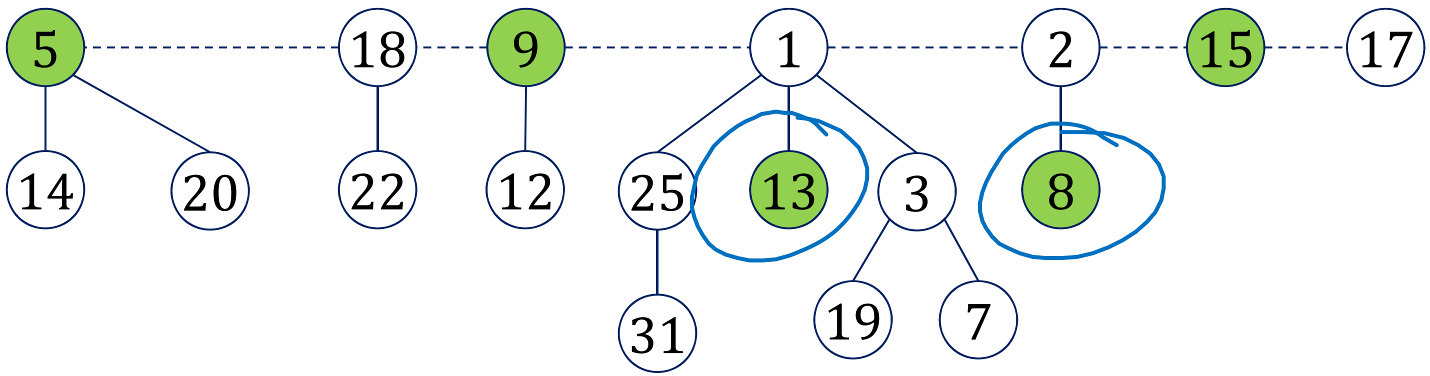
System state characterized by two parameters:

- $\boldsymbol{R}$: number of trees (length of $H.rootlist$)
- $\boldsymbol{M}$: number of marked nodes that are not in the root list

Potential function:

$$\boldsymbol{\Phi := R + 2M}$$

Example:



- $R = 7, M = 2 \rightarrow \Phi = 11$

Actual Time of Operations

- Operations: ***initialize-heap***, ***is-empty***, ***insert***, ***get-min***, ***merge***

  actual time: $O(1)$

  – Normalize unit time such that

  $$t_{init}, t_{is-empty}, t_{insert}, t_{get-min}, t_{merge} \leq 1$$

- Operation ***delete-min***:

  – Actual time: $O(\text{length of } H.rootlist + D(n))$

  – Normalize unit time such that

  $$t_{del-min} \leq D(n) + \text{ length of } H.rootlist$$

- Operation **descrease-key**:

  – Actual time: $O(\text{length of path to next unmarked ancestor})$

  – Normalize unit time such that

  $$t_{decr-key} \leq \text{length of path to next unmarked ancestor}$$

Amortized Times

Assume operation i is of type:

- **initialize-heap:**
  - actual time: $t_i \leq 1$, potential: $\Phi_{i-1} = \Phi_i = 0$
  - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$

- **is-empty, get-min:**
  - actual time: $t_i \leq 1$, potential: $\Phi_i = \Phi_{i-1}$ (heap doesn't change)
  - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$

- **merge:**
  - Actual time: $t_i \leq 1$
  - combined potential of both heaps: $\Phi_i = \Phi_{i-1}$
  - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$

Amortized Time of Insert

Assume that operation i is an *insert* operation:

- **Actual time:** $t_i \leq 1$
- **Potential function:**
  - M remains unchanged (no nodes are marked or unmarked, no marked nodes are moved to the root list)
  - R grows by 1 (one element is added to the root list)

$$M_i = M_{i-1}, \qquad \underline{R_i = R_{i-1} + 1}$$
$$\Phi_i = \Phi_{i-1} + 1$$

- **Amortized time:**

$$\boldsymbol{a_i = t_i + \Phi_i - \Phi_{i-1} \leq 2}$$

Amortized Time of Delete-Min

Assume that operation i is a *delete-min* operation:

R_{i-1}

Actual time: $t_i \leq D(n) + |H.rootlist|$

Potential function $\boldsymbol{\Phi = R + 2M}$**:** $a_i = t_i + \Phi_i - \Phi_{i-1}$

- R: changes from $H.rootlist$ to at most $D(n)$ $R_i \leq D(n), R_{i-1}$
- M: (# of marked nodes that are not in the root list)
  - no new marks
  - if node v is moved away from root list, $v.mark$ is set to false
    → value of M does not change!

$$M_i \leq M_{i-1}, \qquad R_i \leq R_{i-1} + D(n) - |H.rootlist|$$
$$\Phi_i \leq \Phi_{i-1} + D(n) - |H.rootlist|$$

Amortized Time:

$$\boldsymbol{a_i = t_i + \Phi_i - \Phi_{i-1} \leq 2D(n)}$$

Amortized Time of Decrease-Key

Assume that operation i is a *decrease-key* operation at node u:

Actual time: $t_i \leq$ length of path to next unmarked ancestor v

Potential function $\boldsymbol{\Phi = R + 2M}$**:**

- Assume, node u and nodes $u_1, \dots, u_k$ are moved to root list
  - $u_1, \dots, u_k$ are marked and moved to root list, $v.\mathrm{mark}$ is set to true
- $\geq k$ marked nodes go to root list, ≤ 1 node gets newly marked
- R grows by $\leq k+1$, M grows by 1 and is decreased by $\geq k$

$$R_i \leq R_{i-1} + k + 1, \qquad M_i \leq M_{i-1} + 1 - k$$
$$\Phi_i \leq \Phi_{i-1} + (k+1) - 2(k-1) = \Phi_{i-1} + 3 - k$$

Amortized time:

$$\boldsymbol{a_i = t_i + \Phi_i - \Phi_{i-1} \leq \cancel{k} + 1 + 3 - \cancel{k} = 4}$$

Complexities Fibonacci Heap

- **Initialize-Heap**: $\boldsymbol{O(1)}$
- **Is-Empty**: $\boldsymbol{O(1)}$
- **Insert**: $\boldsymbol{O(1)}$
- **Get-Min**: $\boldsymbol{O(1)}$
- **Delete-Min**: $\boxed{\boldsymbol{O(D(n))}}$
- **Decrease-Key**: $\boldsymbol{O(1)}$

amortized

- **Merge** (heaps of size m and n, $m \leq n$): $\boldsymbol{O(1)}$
- **How large can $\boldsymbol{D(n)}$ get?**

$\underline{D(n) = O(\log n)}$

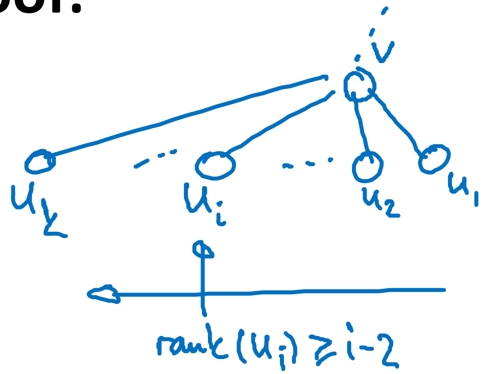
Rank of Children

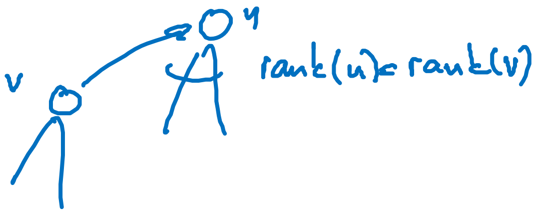
Lemma:

Consider a node v of rank k and let $u_1, \dots, u_k$ be the children of v in the order in which they were linked to v. Then,

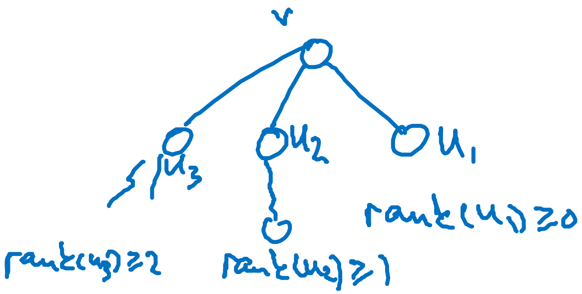
$$\underline{\boldsymbol{rank(u_i) \geq i - 2}}.$$

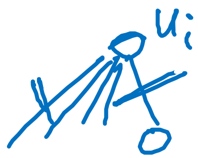
Proof:





when u_i is linked to v, $rank(u_i) \geq i-1$





if a child is cut $\rightarrow$ u_i.mark = true

u_i has lost ≤ 1 child

$\square$

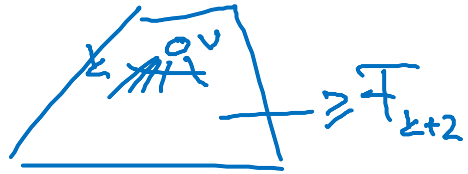
Size of Trees

Fibonacci Numbers:

$$F_0 = 0, \qquad F_1 = 1, \qquad \forall k \geq 2: F_k = F_{k-1} + F_{k-2}$$

$F_0 = 0, F_1 = 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots \to$ grow exponentially

Lemma:
In a Fibonacci heap, the size of the sub-tree of a node v with rank k is at least F_{k+2}.

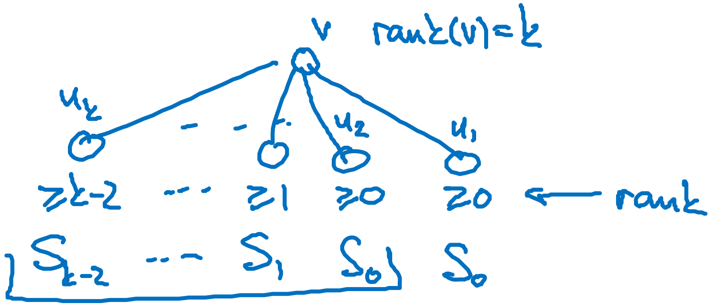


Proof:

- S_k: minimum size of the sub-tree of a node of rank k

$S_0 = 1$, $S_1 = 2$

$\text{rank}(u_i) \geq i-2$



$$S_k \geq 2 + \sum_{i=0}^{k-2} S_i$$

Size of Trees

0, 1, 1, 2, 3, ...

F_2

$$S_0 = 1, \qquad S_1 = 2, \qquad \forall k \geq 2: S_k \geq 2 + \sum_{i=0}^{k-2} S_i$$

- Claim about Fibonacci numbers:

$$\forall k \geq 0: F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

induction

$k=0$ $\quad F_2 = 1 + F_0 = 1$ ✓

step: $F_{k+2} = F_{k+1} + F_k$

$= F_k + 1 + \sum_{i=0}^{k-1} F_i = 1 + \sum_{i=0}^{k} F_i$ ✓

Size of Trees

$$S_0 = 1, S_1 = 2, \forall k \geq 2: S_k \geq 2 + \sum_{i=0}^{k-2} S_i, \qquad F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

- Claim of lemma: $S_k \geq F_{k+2}$

induction on k:

base: $S_0 \geq F_2 = 1$, $S_1 \geq F_3 = 2$ ✓

step: $k \geq 2$:

ind. hyp.

$$S_k \geq 2 + \sum_{i=0}^{k-2} S_i \geq 2 + \sum_{i=0}^{k-2} F_{i+2}$$

$$= 2 + \sum_{i=2}^{k} F_i$$

$$= 1 + \sum_{i=0}^{k} F_i.$$

$\square$

Size of Trees

Lemma:
In a Fibonacci heap, the size of the sub-tree of a node v with rank k is at least F_{k+2}.

Theorem:
The maximum rank of a node in a Fibonacci heap of size n is at most

$$\boxed{\boldsymbol{D(n) = O(\log n)}}\,.$$

Proof:

- The Fibonacci numbers grow exponentially:

$$F_k = \frac{1}{\sqrt{5}} \cdot \left(\boxed{\left(\frac{1+\sqrt{5}}{2}\right)^k} - \left(\frac{1-\sqrt{5}}{2}\right)^k \right)$$

- For $D(n) \geq k$, we need $n \geq F_{k+2}$ nodes.

Summary: Binomial and Fibonacci Heaps

	Binomial Heap	Fibonacci Heap
initialize	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(1)$
get-min	$O(1)$	$O(1)$
delete-min	$O(\log n)$	$O(\log n)$ *
decrease-key	$O(\log n)$	$O(1)$ *
merge	$O(\log n)$	$O(1)$
is-empty	$O(1)$	$O(1)$

$\boxed{\text{* amortized time}}$

$O(m + n \log n)$

Minimum Spanning Trees

Prim Algorithm:

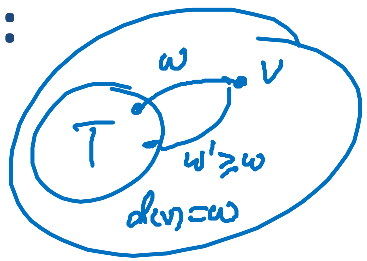
1. Start with any node v (v is the initial component)
2. In each step:
   Grow the current component by adding the minimum weight edge e connecting the current component with any other node

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:



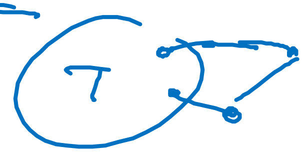
$d(s,s)$

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

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

$d(u) + w(e)$

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:

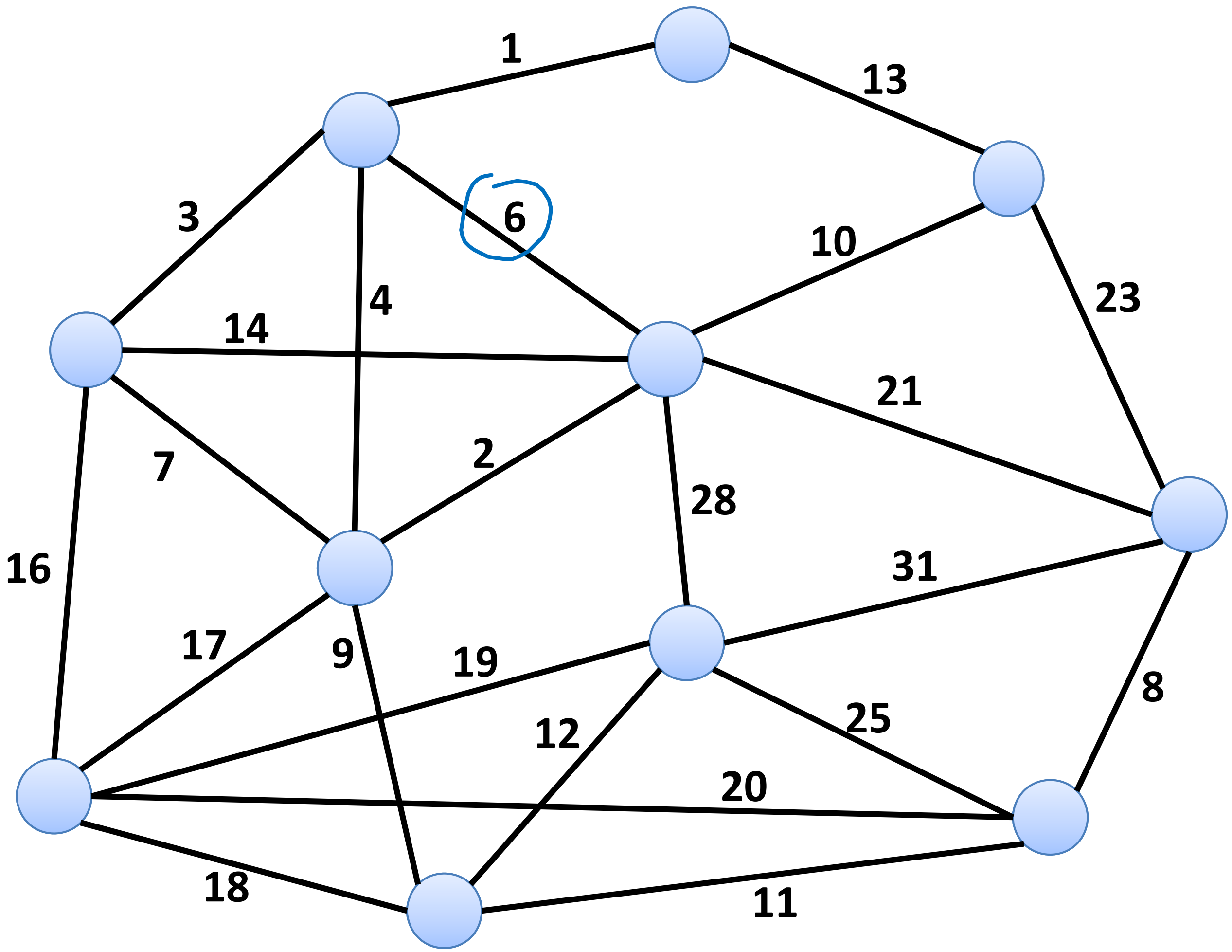
- Analysis identical to the analysis of Dijkstra's algorithm:

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

  $O(m)$ decrease-key operations

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

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 $O(m \log n)$

2. For each edge e:

   if *e* does not close a cycle then

   need efficient way to check if e closes a cycle

   add *e* to the current solution

can be done in time $O(m\,\alpha(m,n))$

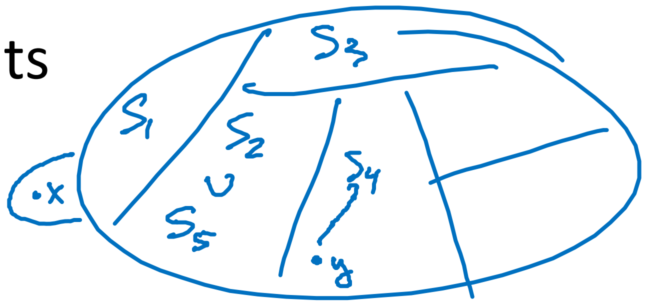
grows extremely slowly

Union-Find Data Structure

Also known as **Disjoint-Set Data Structure**…

Manages partition of a set of elements

- set of disjoint sets



Operations:

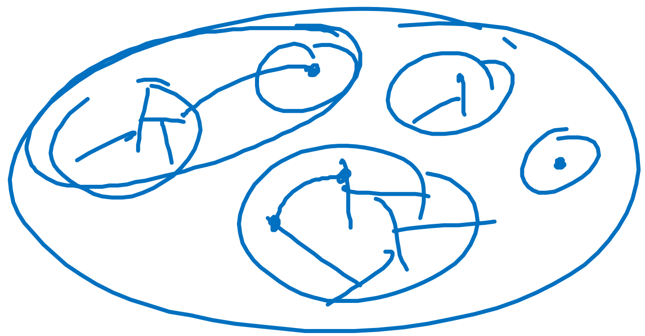
- $\mathbf{make\_set}(\boldsymbol{x})$**:** create a new set that only contains element x
- $\mathbf{find}(\boldsymbol{x})$**:** return the set containing x
- $\mathbf{union}(\boldsymbol{x}, \boldsymbol{y})$**:** merge the two sets containing x and y

Implementation of Kruskal Algorithm

1. Initialization:
   For each node v: $\text{make\_set}(v)$

   n sets

2. Go through edges in order of increasing weights:
   Sort edges by edge weight



3. For each edge $e = \{u, v\}$:

   if $\textbf{find}(\boldsymbol{u}) \neq \textbf{find}(\boldsymbol{v})$ **then**

   add e to the current solution

   $\textbf{union}(\boldsymbol{u}, \boldsymbol{v})$

n make_set op.
m find op.
$n-1$ union op.

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

- $\text{make\_set}(v)$ for every node v
- $\text{find}(v)$ returns component containing v
- $\text{union}(u, v)$ 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

find(x)

Representation of sets:

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

Operations:

- $\mathrm{make\_set}(x)$: x is the representative of the new set $\{x\}$
- $\mathrm{find}(x)$: return representative of set S_x containing x
- $\mathrm{union}(x, y)$: 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

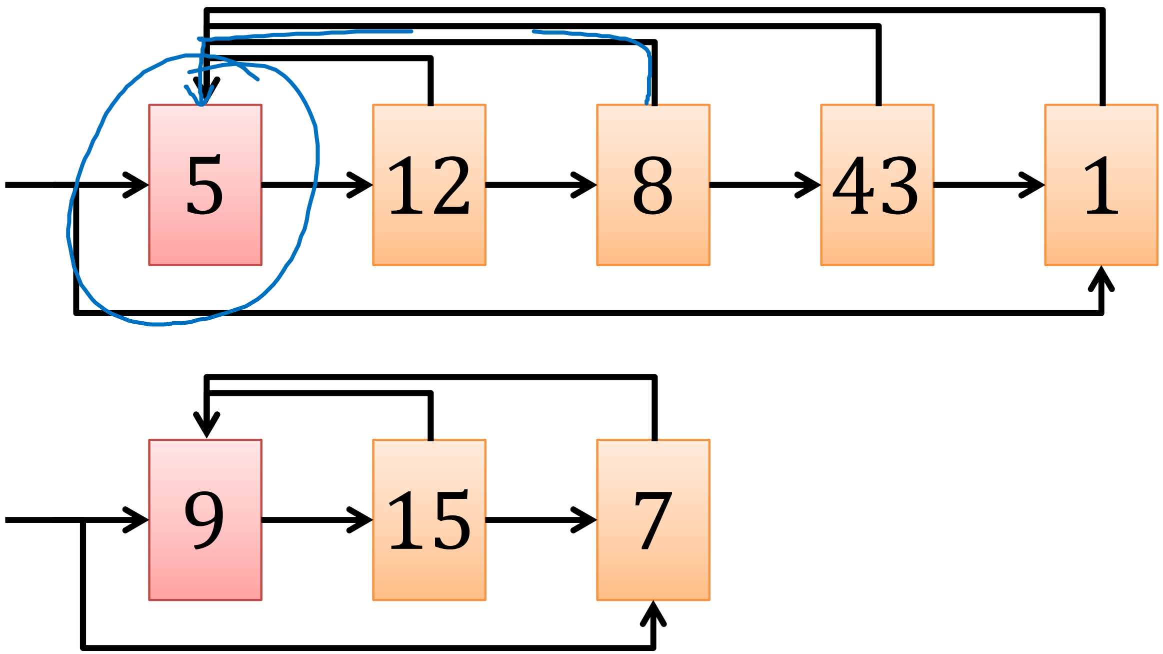
Remark:

- 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



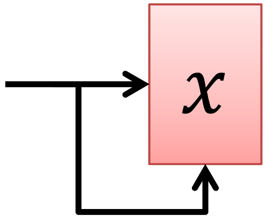
- sets: $\{1,5,8,12,43\}, \{7,9,15\}$; representatives: $5, 9$

Linked List Implementation

make_set(x):

- Create list with one element:

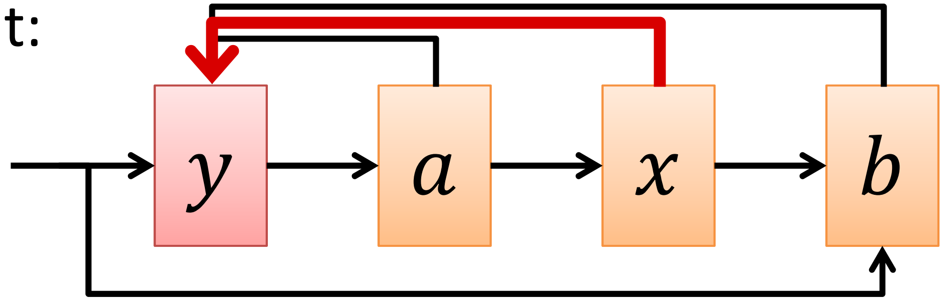
  time: $O(1)$



find(x):

- Return first list element:

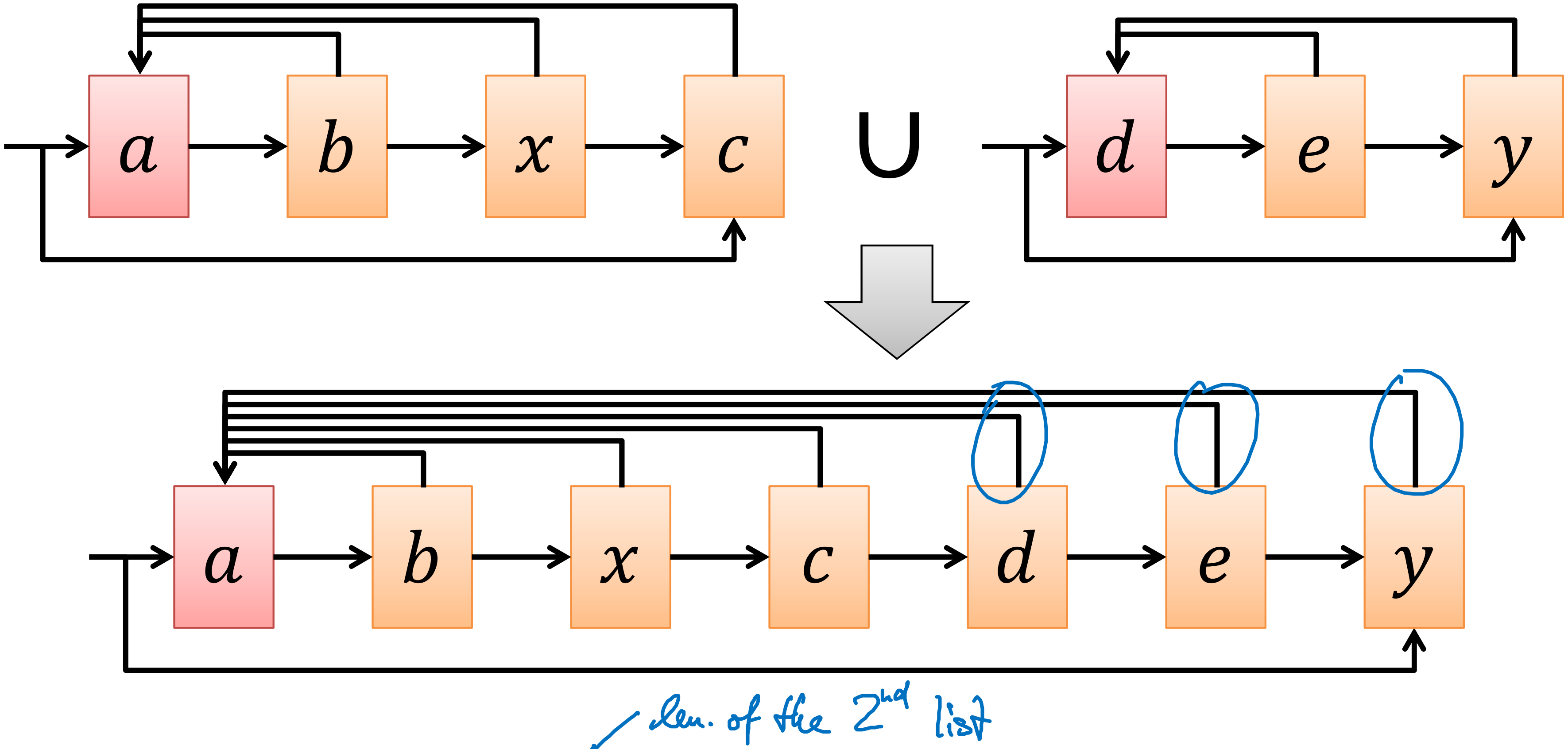
  time: $O(1)$



Linked List Implementation

union(x, y):

- Append list of y to list of x:

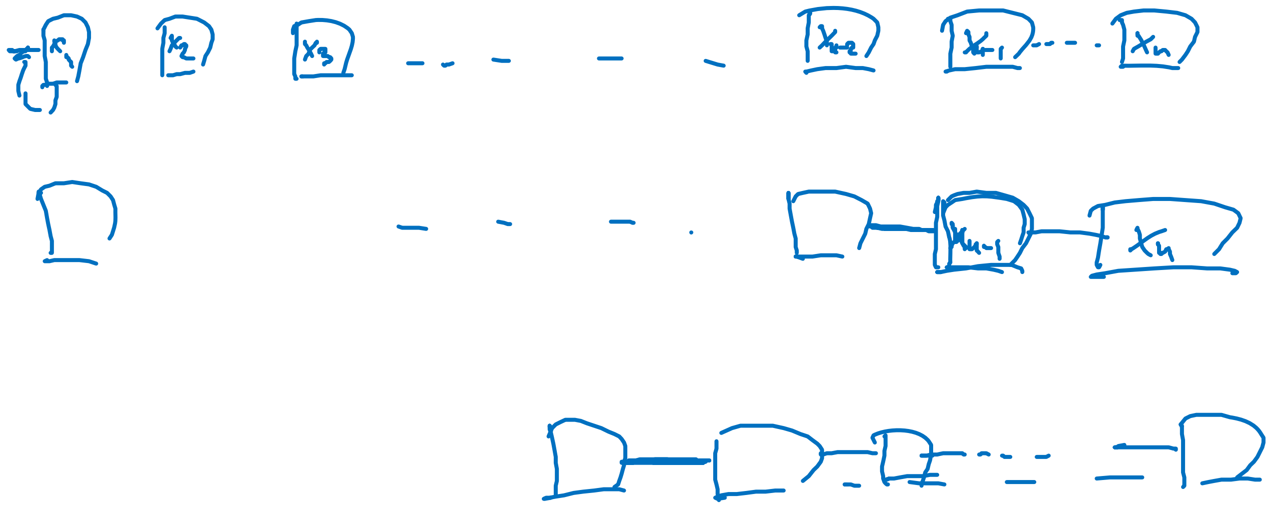


Time: O(length of list of y)

Cost of Union (Linked List Implementation)

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

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



$$1 + 2 + 3 + \dots + n-1 = \Theta(n^2)$$

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!

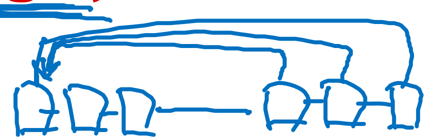
Cost for union of sets S_x and S_y: $O\left(\min\{|S_x|, |S_y|\}\right)$

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

Weighted-Union Heuristic

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

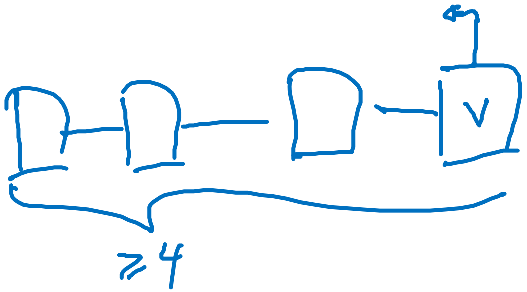
Proof:



make_set, find ops. cost $O(1)$

total union cost $= O(\text{total \# of redirected pointers})$

$= O(n \cdot \underbrace{\text{\# pointer redir. per node}}_{\leq \log n})$ element

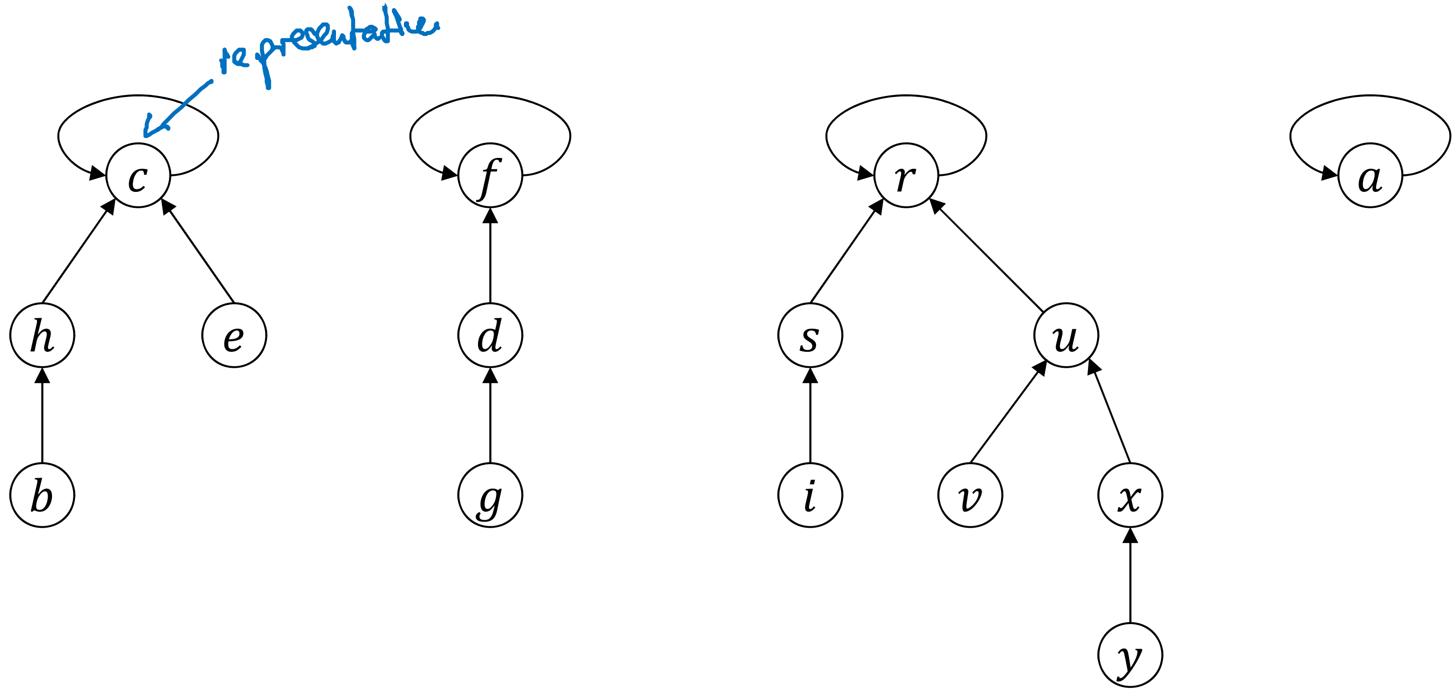


after k redirections of v's pointer

length of v's list: 2^k

$\Longrightarrow k \leq \log n$

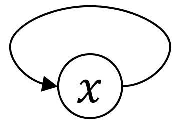
Disjoint-Set Forests



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

Disjoint-Set Forests

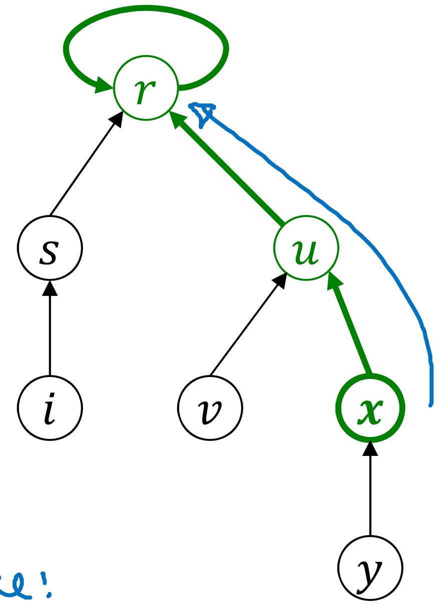
make_set(x): create new one-node tree



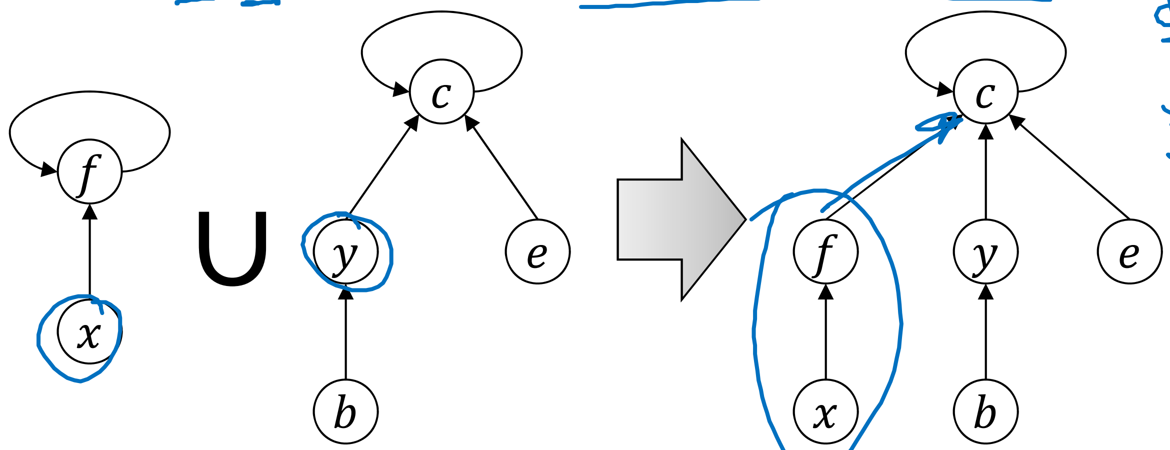
time: $O(1)$

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

time: $O(\text{depth of } x \text{ in its tree})$



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

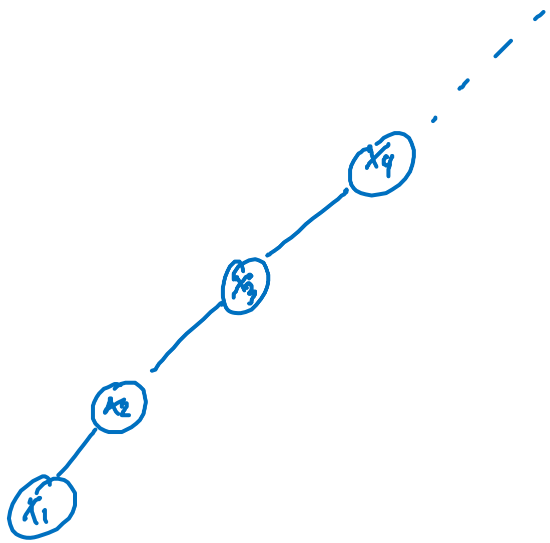


time:
$\underbrace{\text{find}(x)}_{\text{time of}} + \text{find}(y) + O(1)$

Bad Sequence

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

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

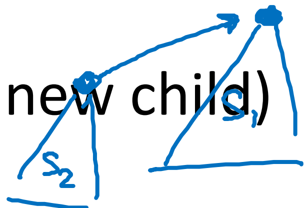


union and find can cost $\Theta(n)$

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-rank heuristic is used, the **worst-case cost of a find-operation is $O(\log n)$**

Proof: show that depth of each tree $= O(\log n)$

Union-Find Algorithms

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

Linked List with Weighted Union Heuristic:

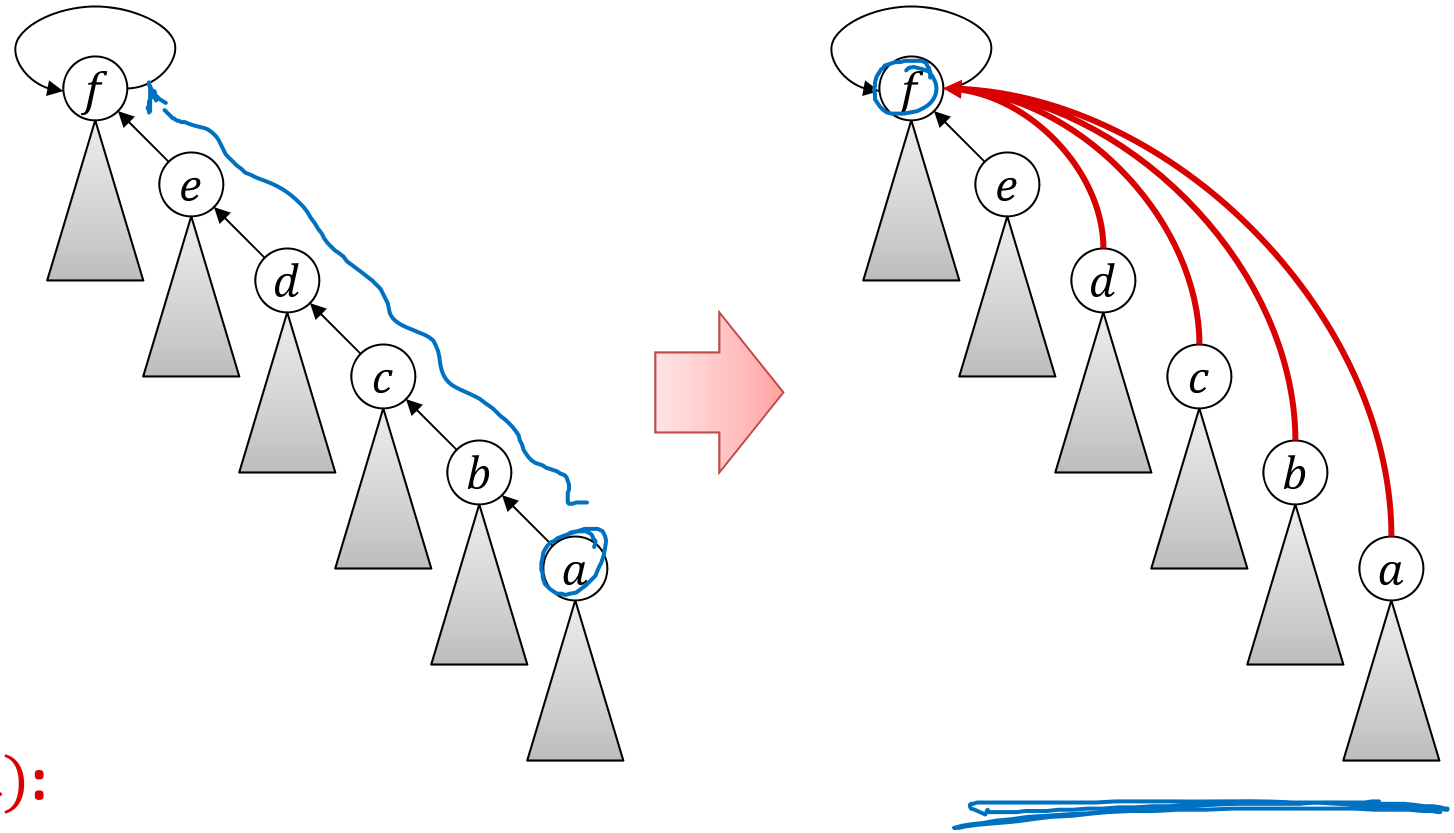
- make_set: worst-case cost $O(1)$ // amortized $O(\log n)$
- find : worst-case cost $O(1)$
- union : amortized worst-case cost $O(\log n)$

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

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: $\mathbf{O}\left(\boldsymbol{n} + \boldsymbol{f} \cdot \left\lceil \mathbf{log}_{2+f/n} \boldsymbol{n} \right\rceil\right) = \boldsymbol{O}\left(\boldsymbol{m} + \boldsymbol{f} \cdot \mathbf{log}_{2+m/n} \boldsymbol{n}\right)$

Union-By-Size and Path Compression

Theorem:

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

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

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

Ackermann Function and its Inverse

Ackermann Function:

For $k, \ell \geq 1$,

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

Inverse of Ackermann Function:

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

Inverse of Ackermann Function

- $\alpha(m,n) \coloneqq \min\{k \geq 1 \mid A(k, \lfloor {}^m/_n \rfloor) > \log_2 n\}$

  $m \geq n \Longrightarrow A(k, \lfloor {}^m/_n \rfloor) \geq A(k,1) \Longrightarrow \alpha(m,n) \leq \min\{k \geq 1 | A(k,1) > \log n\}$

- $A(1,\ell) = 2^\ell, \quad A(k,1) = A(k-1,2),$
  $A(k,\ell) = A\big(k-1, A(k,\ell-1)\big)$