Chapter 4
Data Structures

Algorithm Theory
WS 2013/14

Fabian Kuhn

Fibonacci Heaps: Marks

Cycle of a node:

1. Node v is removed from root list and linked to a node

   $\boldsymbol{v.mark = \textbf{false}}$

2. Child node u *of* v is cut and added to root list

   $\boldsymbol{v.mark = \textbf{true}}$

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

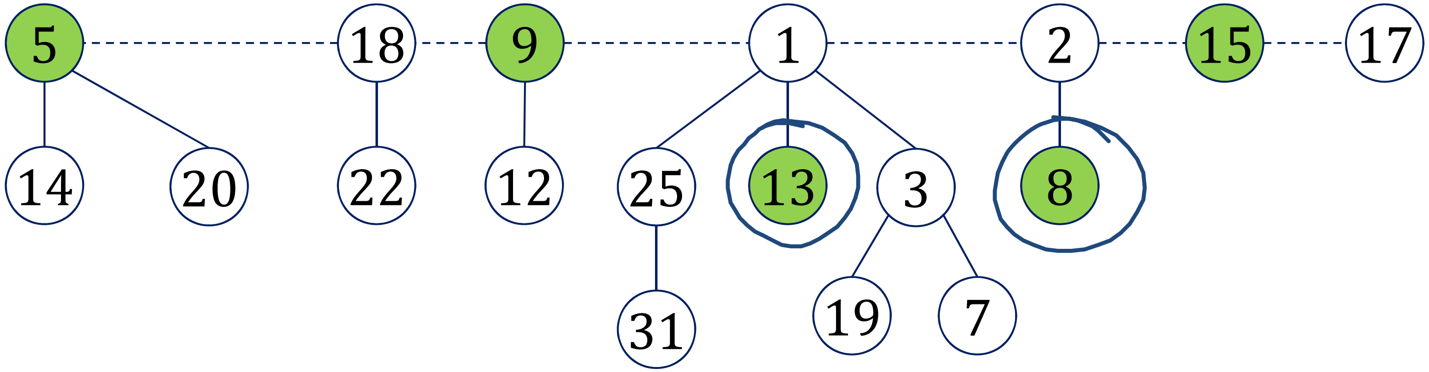
System state characterized by two parameters:

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

Potential function:

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

Example:



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

Actual Time of Operations

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

  actual time: $O(1)$

  - Normalize unit time such that

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

- Operation ***delete-min***:
  - Actual time: $O(\text{length of } H.rootlist + D(n))$
  - Normalize unit time such that

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

- Operation **descrease-key**:
  - Actual time: $O(\text{length of path to next unmarked ancestor})$
  - Normalize unit time such that

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

Amortized Time of Delete-Min

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

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

$a_i = t_i + \underbrace{\phi_i - \phi_{i-1}}$

Potential function $\Phi = R + 2M$:

- R: changes from $\underset{R_{i-1}}{H.rootlist}$ to at most $D(n)$
- 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 increase!

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

$$\Phi_i \leq \Phi_{i-1} + \underbrace{D(n) - |H.rootlist|}$$

Amortized Time: $\leq D(n) + |H.rootlist|$

$-$

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

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 $\boldsymbol{\Phi = R + \boxed{2}M}$**:** $t_i \leq k+1$

- 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:

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

Complexities Fibonacci Heap

- **Initialize-Heap**: $\boldsymbol{O(1)}$
- **Is-Empty**: $\boldsymbol{O(1)}$
- **Insert**: $\boldsymbol{O(1)}$
- **Get-Min**: $\boldsymbol{O(1)}$
- **Delete-Min**: $\underline{\boldsymbol{O(D(n))}}$
- **Decrease-Key**: $\underline{\boldsymbol{O(1)}}$
- **Merge** (heaps of size m and n, $m \leq n$): $\boldsymbol{O(1)}$

amortized

$O(n + n_d \cdot D(n))$

- **How large can $D(n)$ get?**

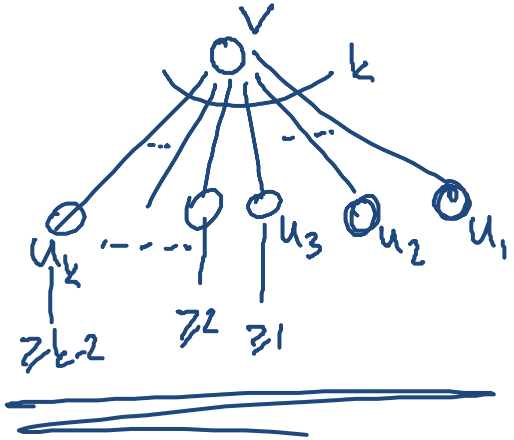
Rank of Children

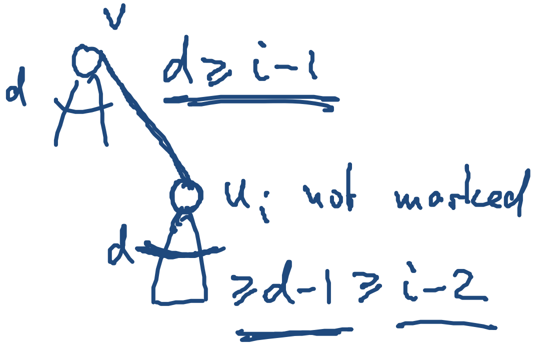
Lemma:

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

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

Proof:





Size of Trees

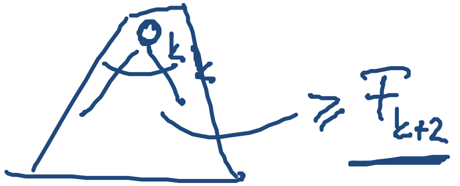
Fibonacci Numbers:

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

$$\overset{F_0}{0},\ \overset{F_1}{1},\ \overset{F_2}{1},\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ \ldots$$

Lemma:

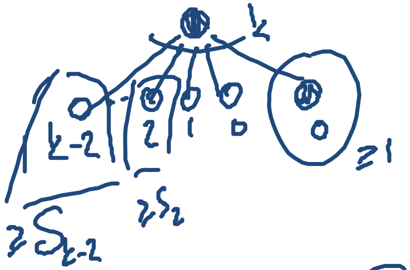
In a Fibonacci heap, the size of the sub-tree of a node v with rank k is at least $\boxed{F_{k+2}.}$



Proof:

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

$S_0 = 1,\ S_1 = 2$



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

Size of Trees

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

- Claim about Fibonacci numbers:

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

$F_0 = 0, \; F_1 = 1, \; F_k = F_{k-1} + F_{k-2}$

$k = 0 : \; F_2 = 1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1$ ✓

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

$$\underset{(I.H.)}{=} F_k + 1 + \sum_{i=0}^{k-1} F_i = \underline{\underline{1 + \sum_{i=0}^{k} F_i}}$$

Size of Trees

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

- Claim of lemma: $\boxed{\underline{S_k \ge F_{k+2}}}$

Ind. on k:

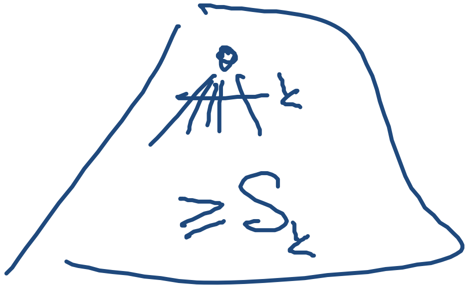
base: $S_0 \ge F_2 = 1$ ✓ $S_1 \ge F_3 = 2$ ✓

step: $\underline{S_k} \ge 2 + \sum_{i=0}^{k-2} S_i \overset{\text{I.H.}}{\ge} 2 + \sum_{i=0}^{k-2} F_{i+2}$

$= 2 + \sum_{j=2}^{k} F_j$

$\overset{0}{F_0} + \overset{1}{F_1} + F_2 + F_3 + \dots + F_k$

$= 1 + \sum_{j=0}^{k} F_j = \underline{F_{k+2}}$



Size of Trees

Lemma:

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

Theorem:

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

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

Proof:

- The Fibonacci numbers grow exponentially:

$$\longrightarrow \quad F_k = \frac{1}{\sqrt{5}} \cdot \left(\underbrace{\left(\frac{1+\sqrt{5}}{2}\right)}_{\varphi}^{k} - \underbrace{\left(\frac{1-\sqrt{5}}{2}\right)}_{\psi}^{k} \right)$$

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

Summary: Binomial and Fibonacci Heaps

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

* **amortized time**

Dijkstra: $O(m + n\log n)$

$O(m \log n)$

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:

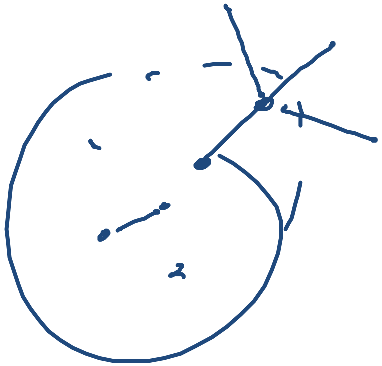
1. Start with an empty edge set
2. In each step:
   Add minimum weight edge e such that e does not close a cycle

Implementation of Prim Algorithm

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

s

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

using Fib. heaps:

$O(m + n \log n)$

#edges

#nodes

Implementation of Prim Algorithm

Implementation with Fibonacci heap:

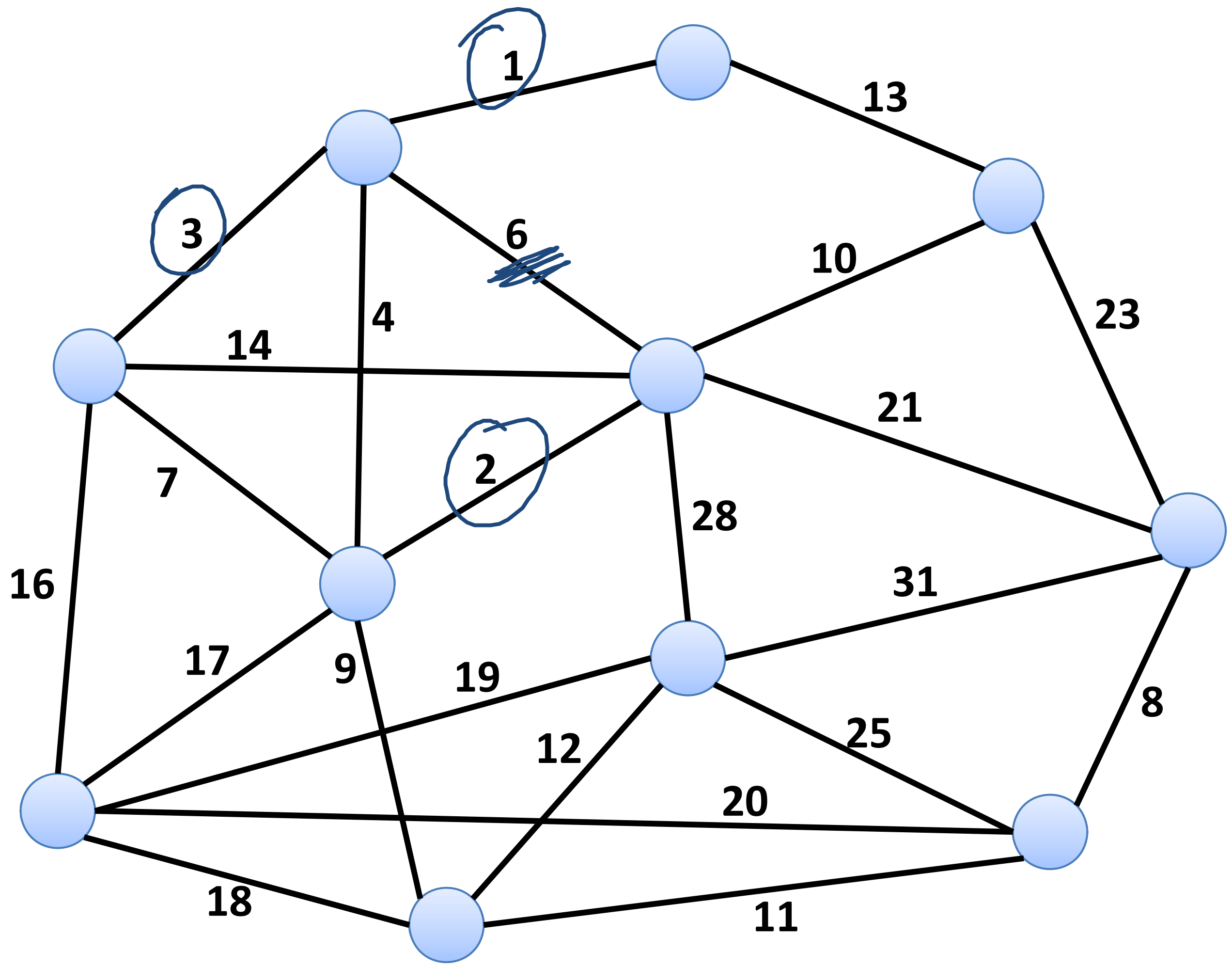
- Analysis identical to the analysis of Dijkstra's algorithm:

  $O(n)$ 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 an efficient way to check whether e closes a cycle

   add *e* to the current solution

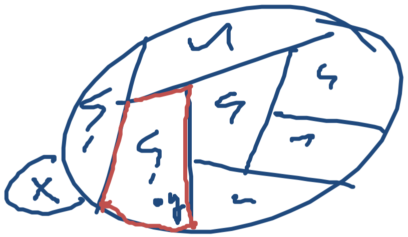
   update data struct.

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: $\mathrm{make\_set}(v)$

n singleton sets

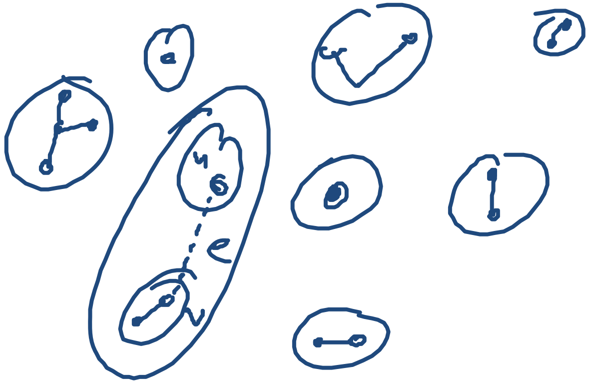
2. Go through edges in order of increasing weights:
   Sort edges by edge weight

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

   if $\mathbf{find}(u) \neq \mathbf{find}(v)$ then

   add e to the current solution

   $\mathbf{union}(u, v)$



n nodes
m edges

make-set : n
find: 2m
union: n−1

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

- $\mathrm{make\_set}(v)$ for every node v

- $\mathrm{find}(v)$ returns component containing v

- $\mathrm{union}(u, v)$ 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:

- $\mathrm{make\_set}(x)$: 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:

- $\underline{n}$: total number of $\mathrm{make\_set}$ operations
- $\underline{m}$: total number of operations ($\underline{\mathrm{make\_set}, \mathrm{find}, \text{and } \mathrm{union}}$)

Clearly:

- $\underline{m \geq n}$
- There are at most $\underline{n - 1\ \mathrm{union}}$ operations

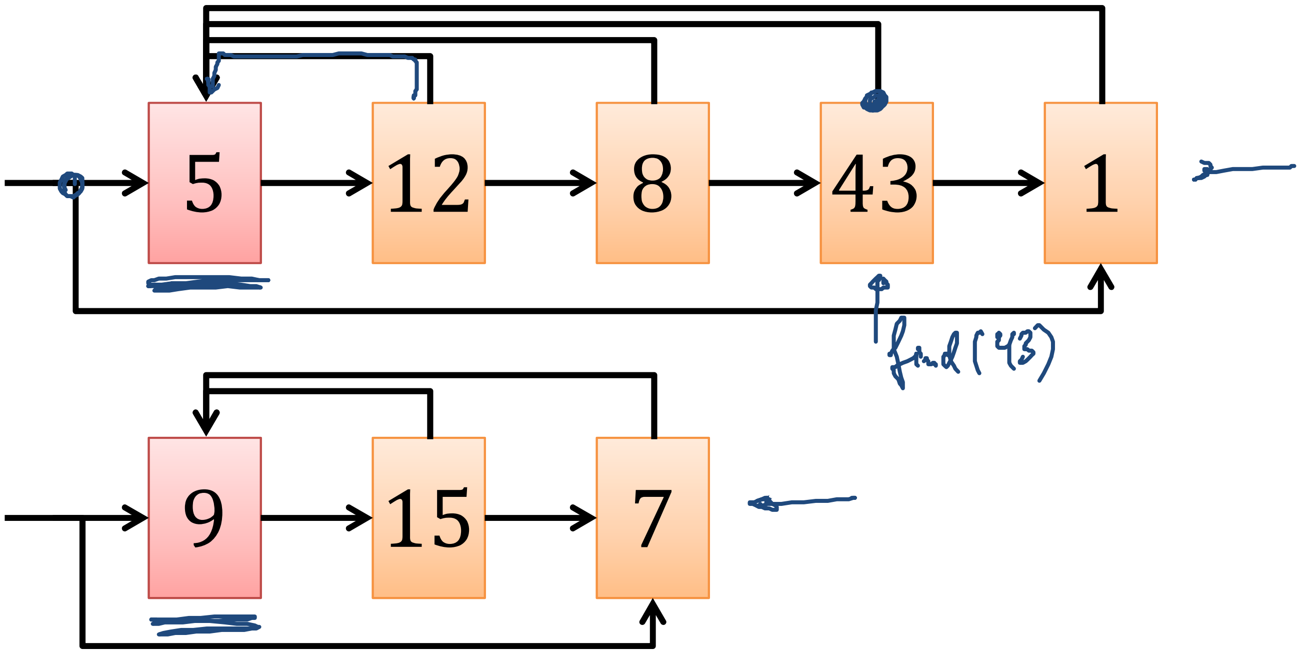
Remark:

- We assume that the $\underline{n\ \mathrm{make\_set}}$ 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



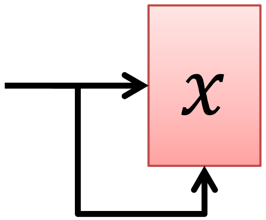
- sets: $\{1,5,8,12,43\}$, $\{7,9,15\}$; representatives: 5, 9

Linked List Implementation

$\mathbf{make\_set}(x)$:

- Create list with one element:

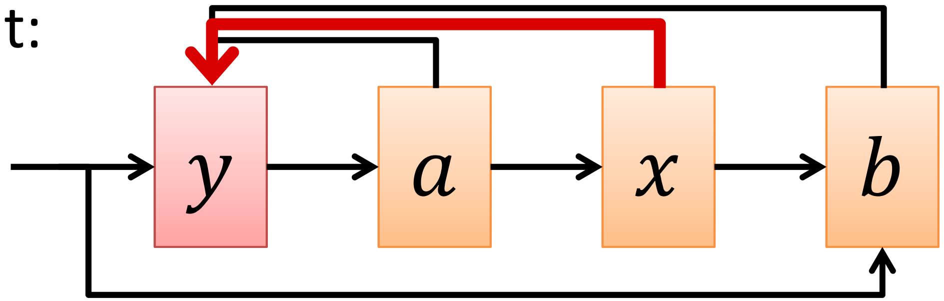
  time: $O(1)$



$\mathbf{find}(x)$:

- Return first list element:

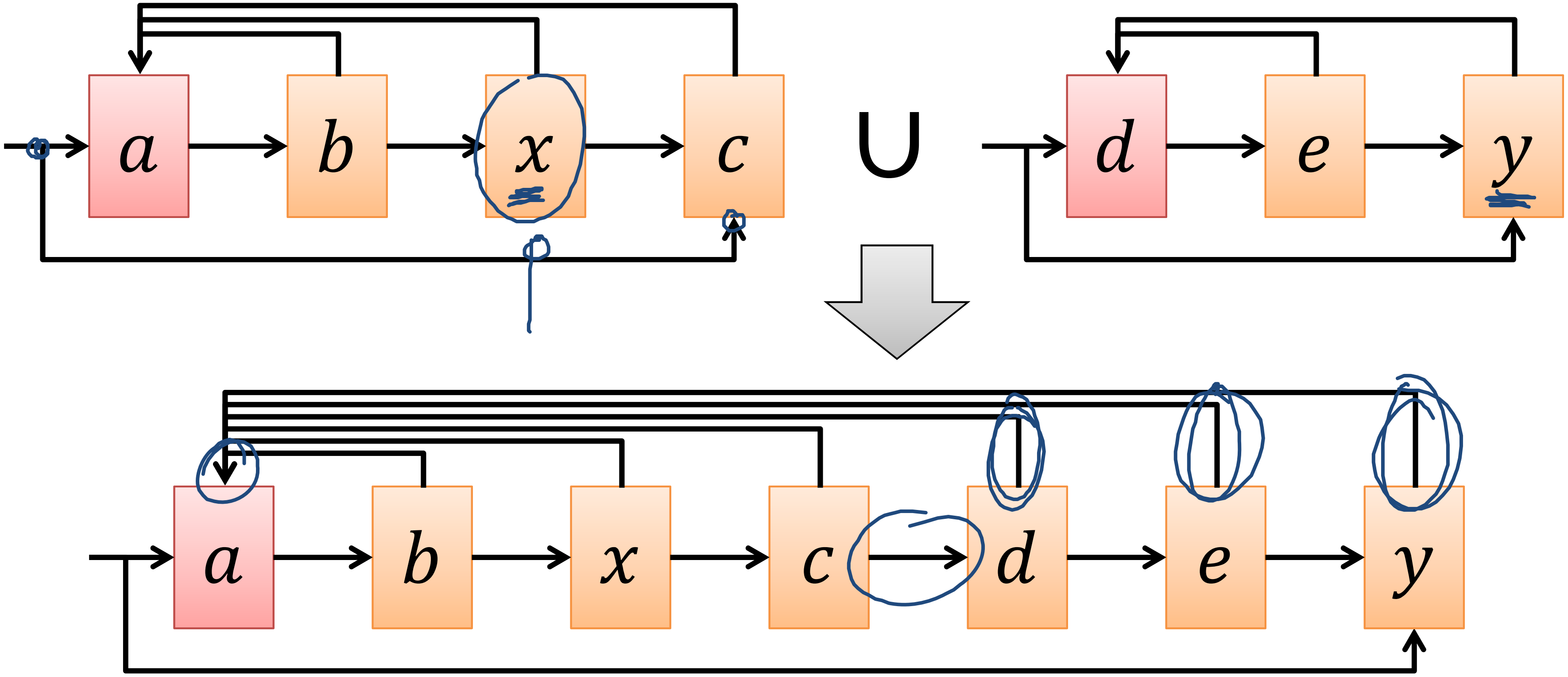
  time: $O(1)$



Linked List Implementation

union$(\boldsymbol{x}, \boldsymbol{y})$:

- Append list of y to list of x:

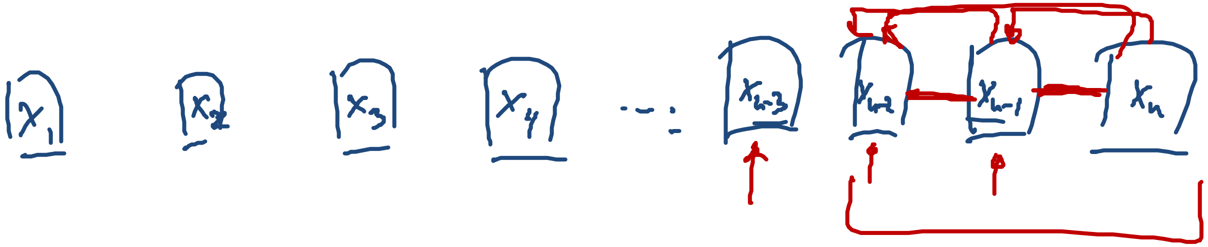


Time: $\boldsymbol{O}$(length of list of $\boldsymbol{y}$)

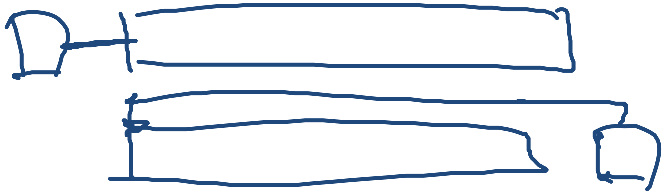
Cost of Union (Linked List Implementation)

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

- make_set(x_1), make_set(x_2), … , make_set(x_n), union(x_{n-1}, x_n), union(x_{n-2}, x_{n-1}), … , union(x_1, x_2)



redirect $1+2+\ldots+n-1 = \Theta(n^2)$



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

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

n_u

$$O(m + n_u \log n)$$

Weighted-Union Heuristic

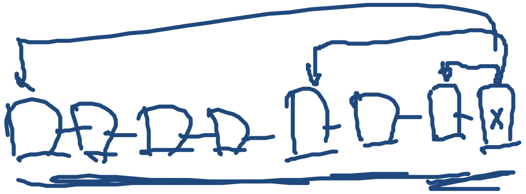
Theorem: The overall cost of m operations of which at most n are make_set operations is $\underline{\boldsymbol{O}(\boldsymbol{m} + \underset{u_n}{\boldsymbol{n}} \log \boldsymbol{n})}$.

Proof:

make-set, find: $O(1)$

total union cost $= O(\text{total \# of pointer redirections})$

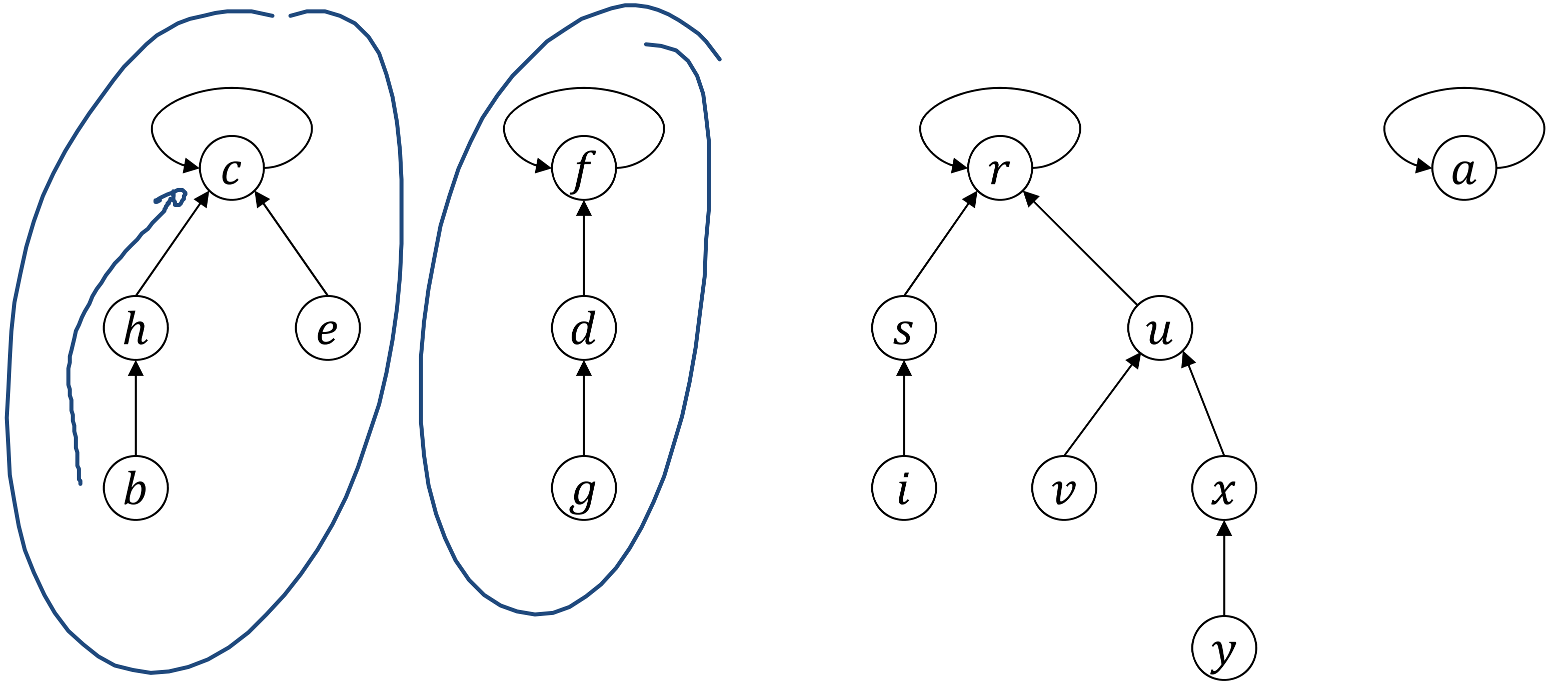
$$O(n \cdot \underbrace{\#\text{ redir. per element}}_{\leq \log n})$$



after k redir. of x's pointer

set of x has $\geq 2^k$ elements

Disjoint-Set Forests

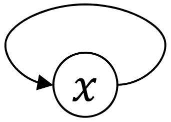


- Represent each set by a tree

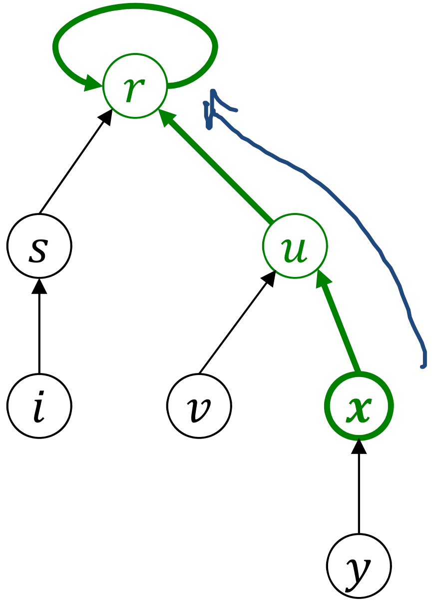
- Representative of a set is the root of the tree

Disjoint-Set Forests

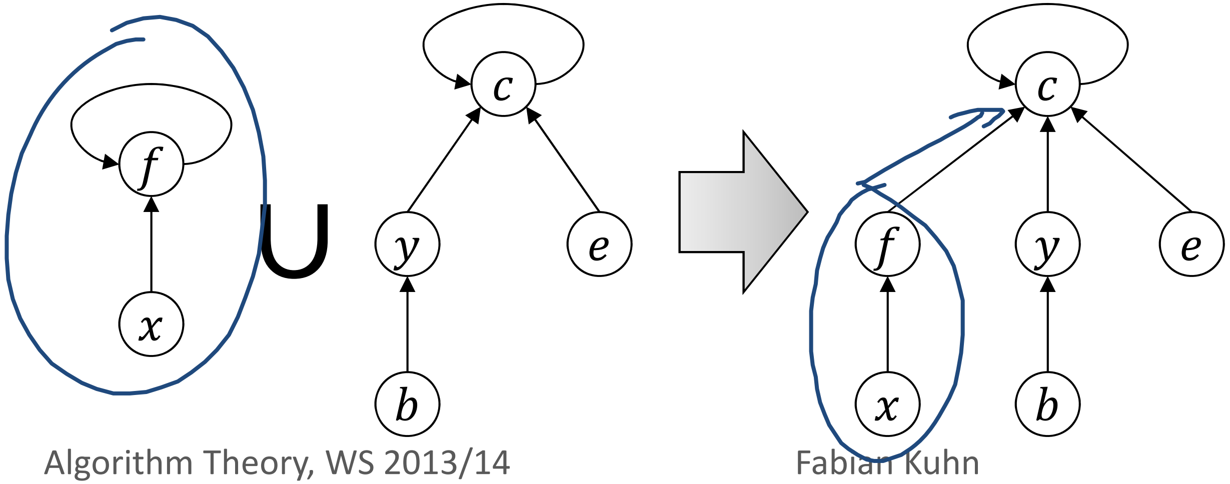
make_set(x): create new one-node tree



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



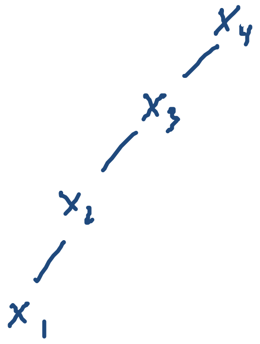
union(x, y): attach tree of x to tree of y



Bad Sequence

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

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



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:

tree with k elements has depth $\leq O(\log_2 k)$

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:

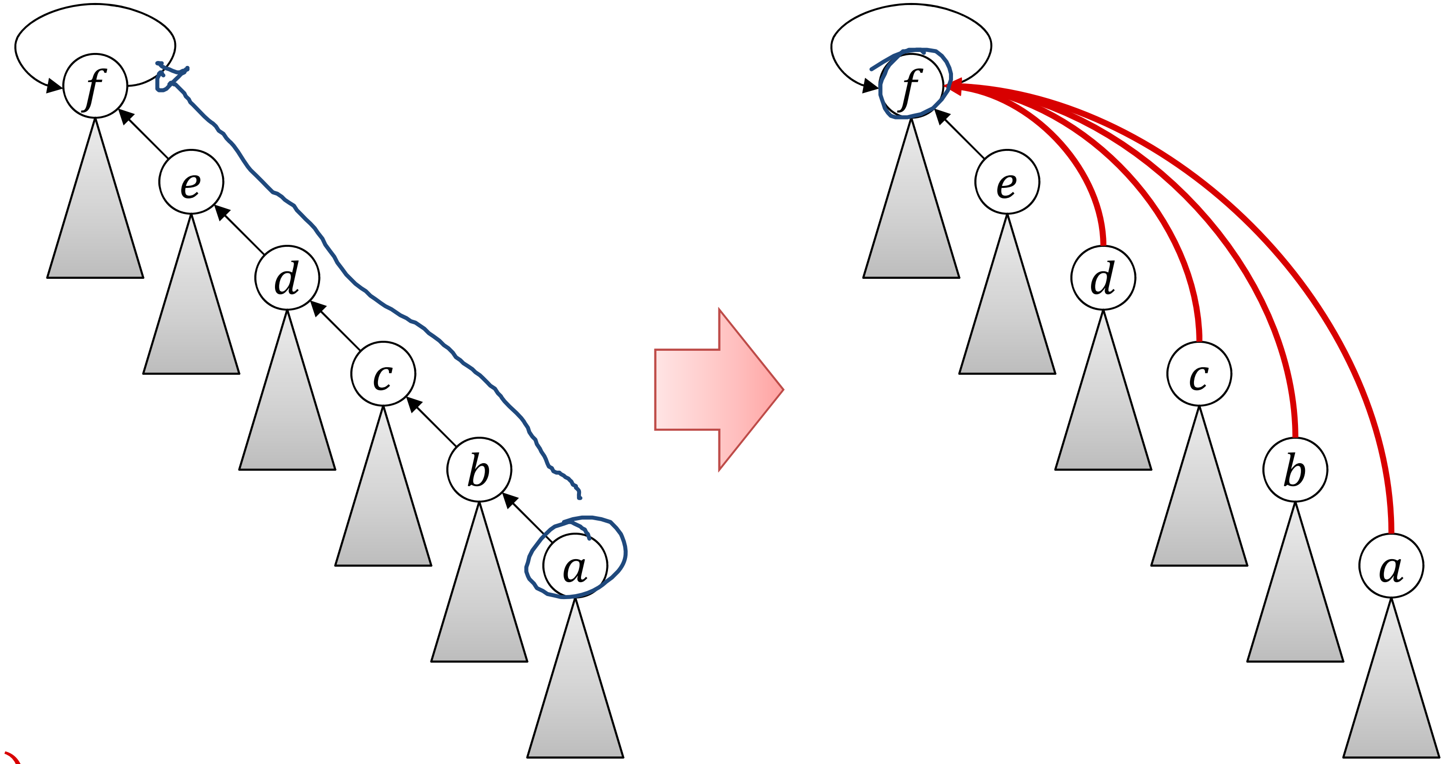
- make_set: worst-case cost $O(1)$
- find : worst-case cost $O(1)$
- union : amortized worst-case cost $O(\log n)$

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):

```
if $a \neq a.parent$ then
    $a.parent := \text{find}(a.parent)$
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: $\mathbf{O}\left(\cancel{n}m + f \cdot \underbrace{\left\lceil \log_{2+f/n} n \right\rceil}\right) = O\left(m + f \cdot \log_{2+m/n} n\right)$

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

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

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

grows extremely slowly

in practice ≤ 4

Ackermann Function and its Inverse

Ackermann Function:

For $k, \ell \geq 1$,

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

Inverse of Ackermann Function:

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

Inverse of Ackermann Function

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

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

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