Chapter 4
Data Structures

Algorithm Theory
WS 2014/15

Fabian Kuhn

Fibonacci Heaps

Lacy-merge variant of binomial heaps:

- Do not merge trees as long as possible…

Structure:

A Fibonacci heap H consists of a collection of trees satisfying the min-heap property.

Variables:

- $H.min$: root of the tree containing the (a) minimum key
- $H.rootlist$: circular, doubly linked, unordered list containing the roots of all trees
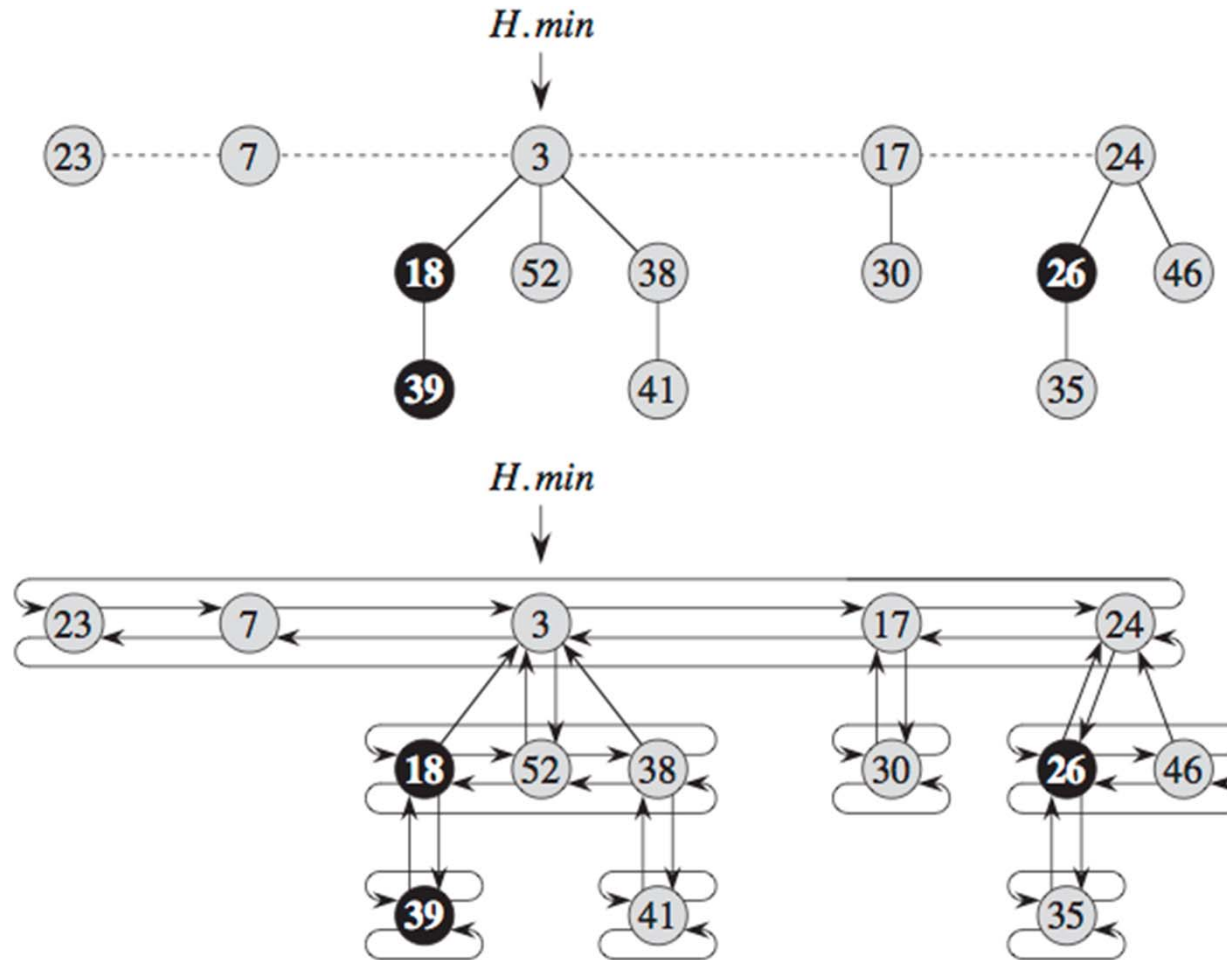- $H.size$: number of nodes currently in H

Example

Figure: Cormen et al., Introduction to Algorithms

Cost of Delete-Min & Decrease-Key

Delete-Min:

1. Delete min. root r and add $r.child$ to $H.rootlist$

   time: $O(1)$

2. Consolidate $H.rootlist$

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

- Step 2 can potentially be linear in n (size of H)

Decrease-Key (at node v):

1. If new key $<$ parent key, cut sub-tree of node v

   time: $O(1)$

2. Cascading cuts up the tree as long as nodes are marked

   time: $O(\text{number of consecutive marked nodes})$

- Step 2 can potentially be linear in n

Exercises: Both operations can take $\Theta(n)$ time in the worst case!

Cost of Delete-Min & Decrease-Key

- Cost of delete-min and decrease-key can be $\Theta(n)$...
  - Seems a large price to pay to get insert and merge in $O(1)$ time

- Maybe, the operations are efficient most of the time?
  - It seems to require a lot of operations to get a long rootlist and thus, an expensive consolidate operation
  - In each decrease-key operation, at most one node gets marked: We need a lot of decrease-key operations to get an expensive decrease-key operation

- Can we show that the average cost per operation is small?

- We can → requires **amortized analysis**

Amortization

- Consider sequence $o_1, o_2, \ldots, o_n$ of n operations
  (typically performed on some data structure D)

- t_i: execution time of operation o_i
- $T := t_1 + t_2 + \cdots + t_n$: total execution time

- The execution time of a single operation might

  vary within a large range (e.g., $t_i \in [1, O(i)]$)

- The worst case overall execution time might still be small

  $\rightarrow$ average execution time per operation might be small in
  the worst case, even if single operations can be expensive

Analysis of Algorithms

- Best case

- Worst case

- Average case

- Amortized worst case

  What it the average cost of an operation in a worst case sequence of operations?

Example: Binary Counter

Incrementing a binary counter: determine the bit flip cost:

Operation	Counter Value	Cost
	00000	
1	00001	1
2	00010	2
3	00011	1
4	00100	3
5	00101	1
6	00110	2
7	00111	1
8	01000	4
9	01001	1
10	01010	2
11	01011	1
12	01100	3
13	01101	1

Accounting Method

Observation:

- Each increment flips exactly one 0 into a 1

$$00100\textbf{0}1111 \implies 00100\textbf{1}0000$$

Idea:

- Have a bank account (with initial amount 0)
- Paying x to the bank account costs x
- Take "money" from account to pay for expensive operations

Applied to binary counter:

- Flip from 0 to 1: pay 1 to bank account (cost: 2)
- Flip from 1 to 0: take 1 from bank account (cost: 0)
- Amount on bank account = number of ones
  $\rightarrow$ We always have enough "money" to pay!

Accounting Method

Op.	Counter	Cost	To Bank	From Bank	Net Cost	Credit
	0 0 0 0 0					
1	0 0 0 0 1	1				
2	0 0 0 1 0	2				
3	0 0 0 1 1	1				
4	0 0 1 0 0	3				
5	0 0 1 0 1	1				
6	0 0 1 1 0	2				
7	0 0 1 1 1	1				
8	0 1 0 0 0	4				
9	0 1 0 0 1	1				
10	0 1 0 1 0	2				

Potential Function Method

- Most generic and elegant way to do amortized analysis!
  - But, also more abstract than the others...

- State of data structure / system: $S \in \mathcal{S}$ (state space)

  Potential function $\Phi : \mathcal{S} \to \mathbb{R}_{\geq 0}$

- **Operation i:**
  - t_i: actual cost of operation i
  - S_i: state after execution of operation i (S_0: initial state)
  - $\Phi_i := \Phi(S_i)$: potential after exec. of operation i
  - a_i: amortized cost of operation i:

  $$a_i := t_i + \Phi_i - \Phi_{i-1}$$

Potential Function Method

Operation i:

actual cost: t_i amortized cost: $a_i = t_i + \Phi_i - \Phi_{i-1}$

Overall cost:

$$T := \sum_{i=1}^{n} t_i = \left(\sum_{i}^{n} a_i \right) + \Phi_0 - \Phi_n$$

Binary Counter: Potential Method

- **Potential function:**

$$\Phi: \text{number of ones in current counter}$$

- Clearly, $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all $i \geq 0$

- Actual cost t_i:
  - 1 flip from 0 to 1
  - $t_i - 1$ flips from 1 to 0

- Potential difference: $\Phi_i - \Phi_{i-1} = 1 - (t_i - 1) = 2 - t_i$

- Amortized cost: $a_i = t_i + \Phi_i - \Phi_{i-1} = 2$

Back to Fibonacci Heaps

- Worst-case cost of a single delete-min or decrease-key operation is $\Omega(n)$

- Can we prove a small worst-case amortized cost for delete-min and decrease-key operations?

Remark:

- Data structure that allows operations $O_1, \dots, O_k$

- We say that operation O_p has amortized cost a_p if for every execution the total time is

$$T \leq \sum_{p=1}^{k} n_p \cdot a_p \, ,$$

where n_p is the number of operations of type O_p

Amortized Cost of Fibonacci Heaps

- **Initialize-heap**, **is-empty**, **get-min**, **insert**, and **merge** have **worst-case cost $O(1)$**

- **Delete-min** has **amortized cost $O(\log n)$**
- **Decrease-key** has **amortized cost $O(1)$**

- Starting with an empty heap, any sequence of n operations with at most n_d delete-min operations has total cost (time)

$$T = O(n + n_d \log n).$$

- We will now need the marks…

- Cost for Dijkstra: $O(|E| + |V| \log |V|)$

Simple (Lazy) Operations

Initialize-Heap H:

- $H.rootlist \coloneqq H.min \coloneqq null$

Merge heaps H and H':

- concatenate root lists
- update $H.min$

Insert element e into H:

- create new one-node tree containing e → H'
- merge heaps H and H'

Get minimum element of H:

- return $H.min$

Operation Delete-Min

Delete the node with minimum key from H and return its element:

1. $m := H.min$;

2. **if** $H.size > 0$ **then**

3.   remove $H.min$ from $H.rootlist$;

4.   add $H.min.child$ (list) to $H.rootlist$

5. $\boldsymbol{H.Consolidate()};$

   // Repeatedly merge nodes with equal degree in the root list
   // until degrees of nodes in the root list are distinct.
   // Determine the element with minimum key

6. **return** m

Rank and Maximum Degree

Ranks of nodes, trees, heap:

Node v:

- $rank(v)$: degree of v

Tree T:

- $rank(T)$: rank (degree) of root node of T

Heap H:

- $rank(H)$: maximum degree of any node in H

Assumption (n: number of nodes in H):

$$rank(H) \leq D(n)$$

  – for a known function $D(n)$

Operation Decrease-Key

Decrease-Key(v, x): (decrease key of node v to new value x)

1. **if** $x \geq v.key$ **then return**;

2. $v.key := x$; update $H.min$;

3. **if** $v \in H.rootlist \;\vee\; x \geq v.parent.key$ **then return**

4. **repeat**

5. $parent := v.parent$;

6. $\boldsymbol{H.cut(v)}$;

7. $v := parent$;

8. **until** $\neg(\boldsymbol{v.mark}) \;\vee\; v \in H.rootlist$;

9. **if** $v \notin H.rootlist$ **then** $\boldsymbol{v.mark} := \textbf{true}$;

Fibonacci Heaps: Marks

Cycle of a node:

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

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

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

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

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:

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

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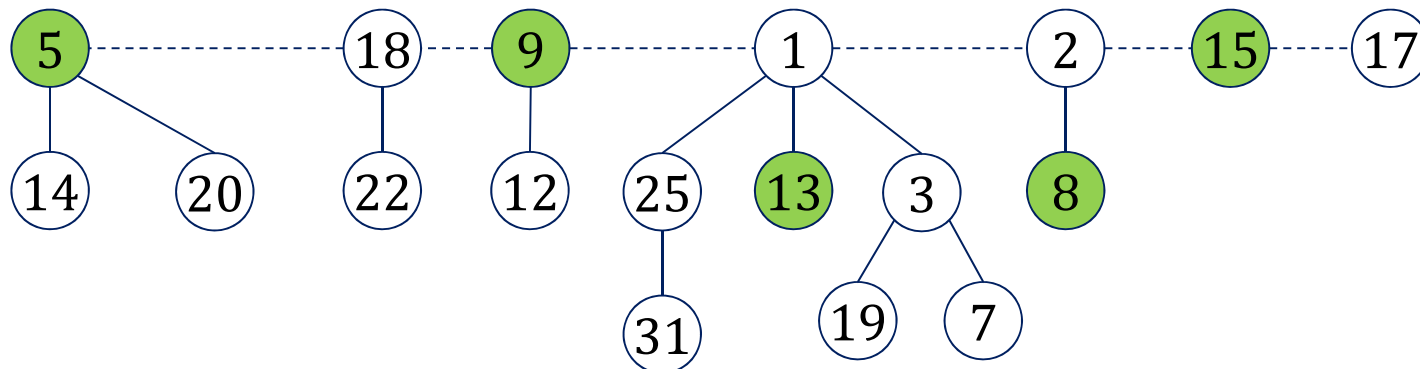
- $R = 7, M = 2 \rightarrow \Phi = 11$

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: $O\big(\text{length of } H.rootlist + D(n)\big)$
  - 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)

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

- **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:

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

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

- R: changes from $H.rootlist$ 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
    $\rightarrow$ value of M does not increase!

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

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

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 $\Phi = R + 2M$:

- Assume, node u and nodes $u_1, \ldots, u_k$ are moved to root list
  - $u_1, \ldots, u_k$ are marked and moved to root list, $v.$ 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:

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

Complexities Fibonacci Heap

- **Initialize-Heap**: $O(1)$

- **Is-Empty**: $O(1)$

- **Insert**: $O(1)$

- **Get-Min**: $O(1)$

- **Delete-Min**: $O(D(n))$ ← amortized

- **Decrease-Key**: $O(1)$ ←

- **Merge** (heaps of size m and n, $m \leq n$): $O(1)$

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

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,

$$rank(u_i) \geq i - 2.$$

Proof:

Size of Trees

Fibonacci Numbers:

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

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

Size of Trees

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

- Claim about Fibonacci numbers:

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

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: $S_k \geq F_{k+2}$

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

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

Proof:

- The Fibonacci numbers grow exponentially:

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

- 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	$O(1)$	$O(1)$

* **amortized time**

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:

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
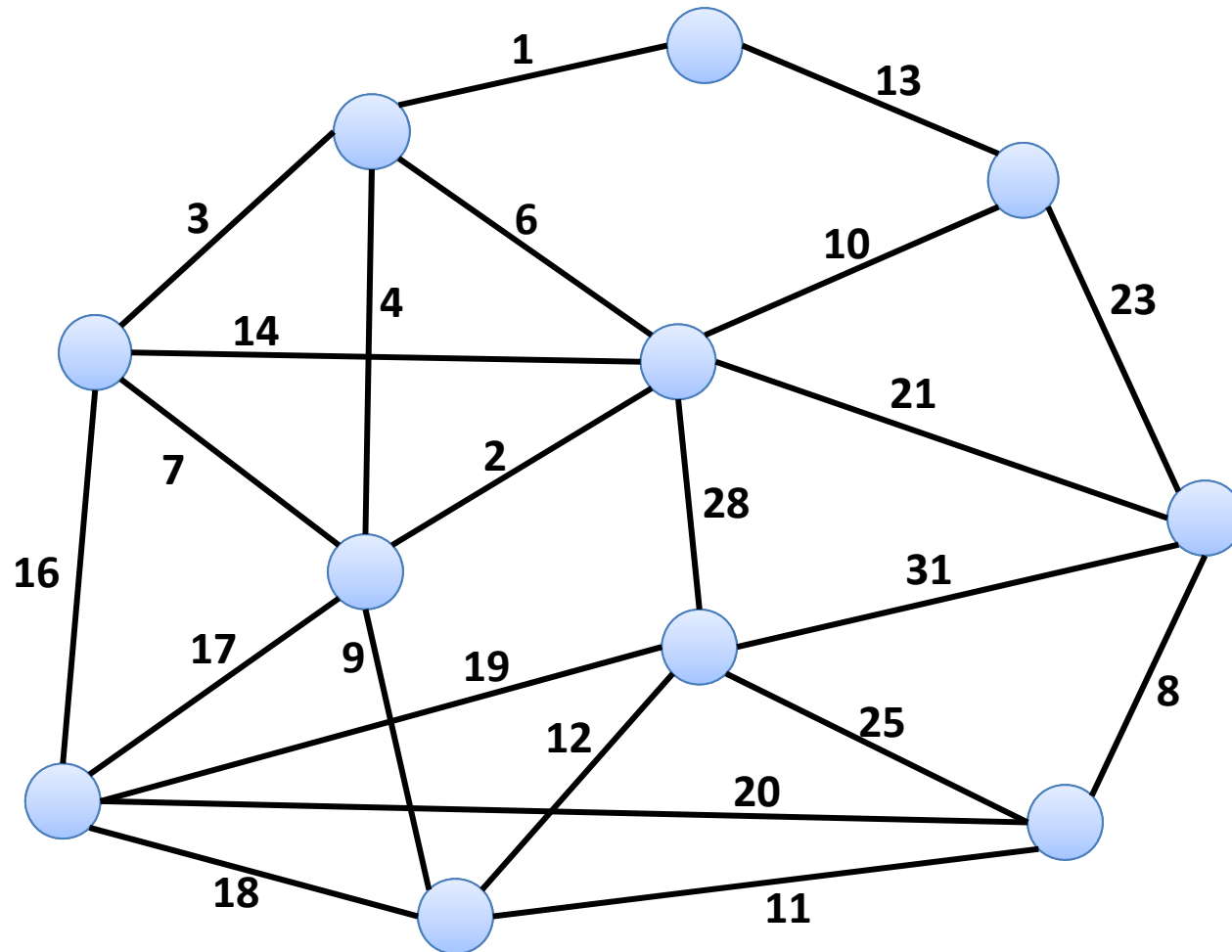
1. Start with an empty edge set

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

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

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

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

         add e to the current solution

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

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

- $\text{make\_set}(v)$ 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

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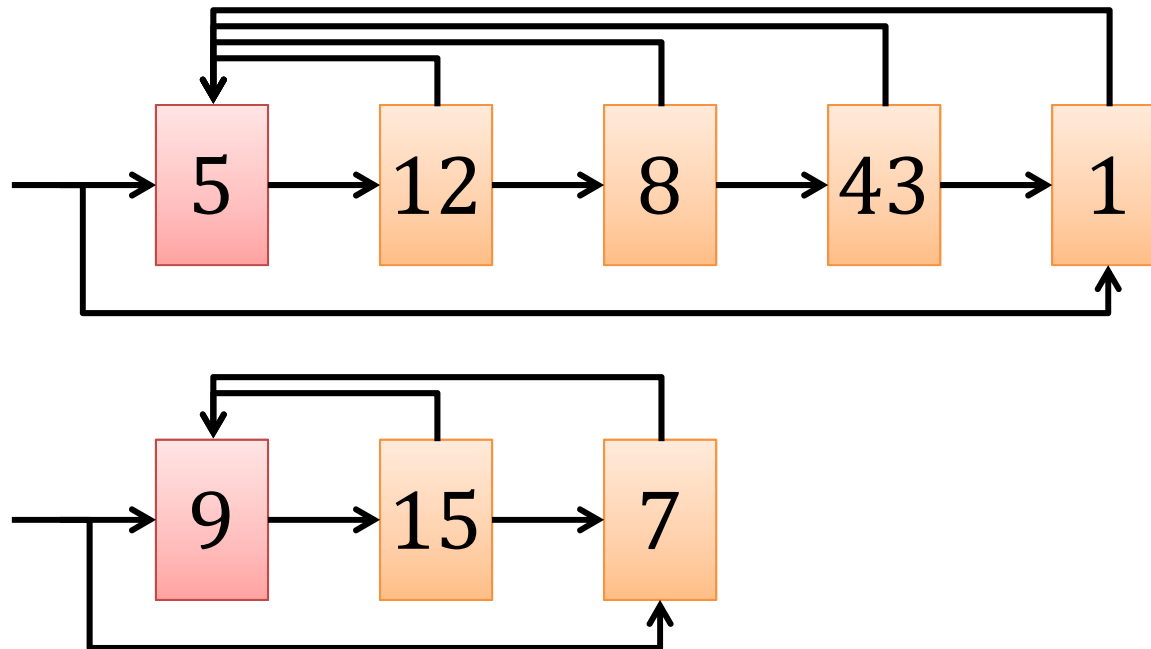
- sets: {1,5,8,12,43}, {7,9,15}; representatives: 5, 9

Linked List Implementation

make_set(x):

- Create list with one element:

  time: $O(1)$
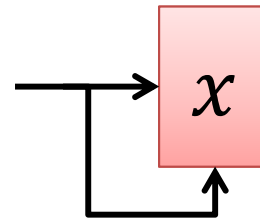
find(x):

- Return first list element:

  time: $O(1)$
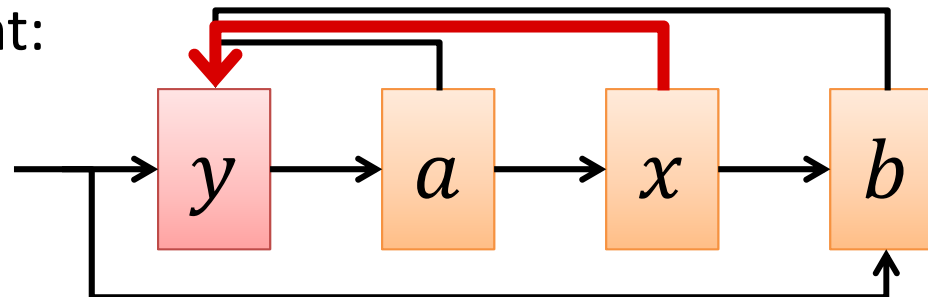
Linked List Implementation

union(x, y):

- Append list of y to list of x:
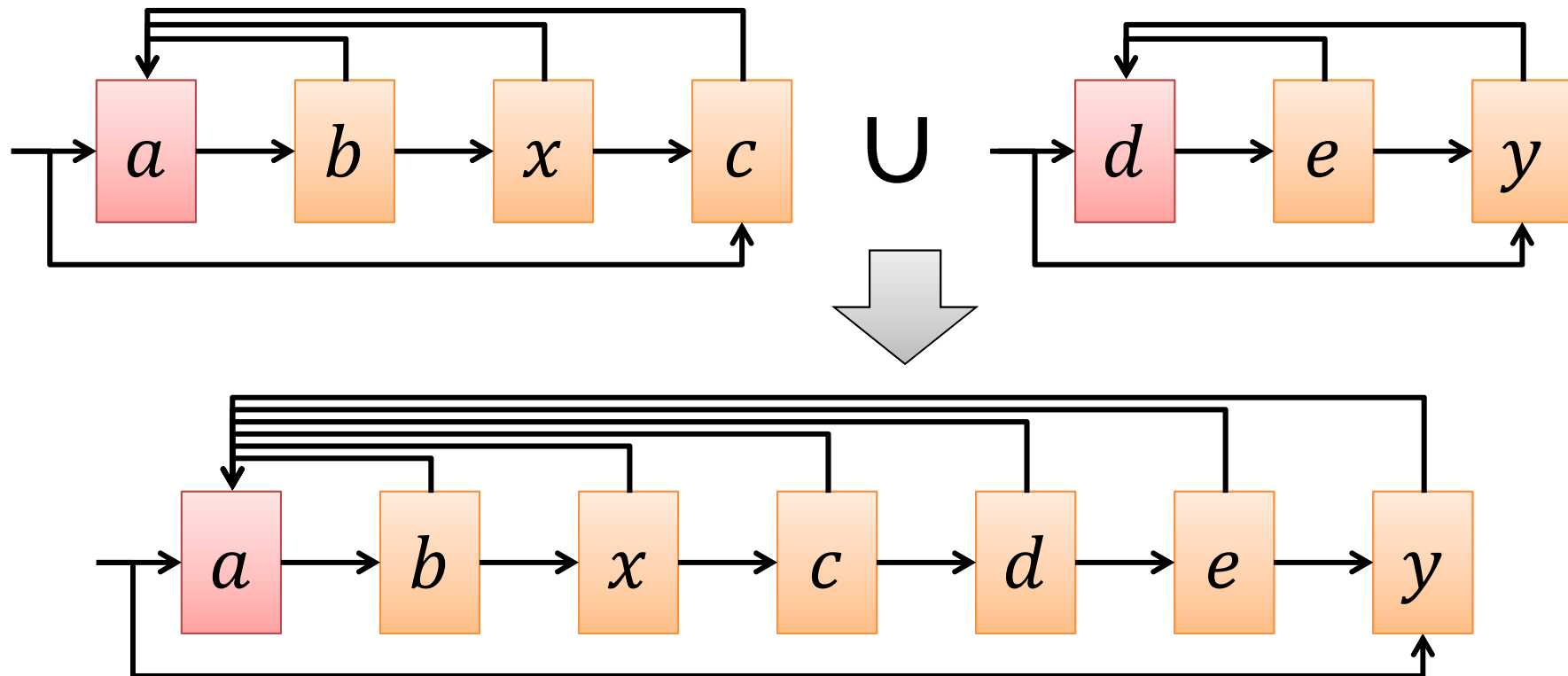
Time: $O($length of list of $y)$

Cost of Union (Linked List Implementation)

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

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

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

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

Weighted-Union Heuristic

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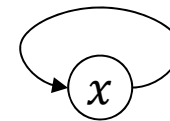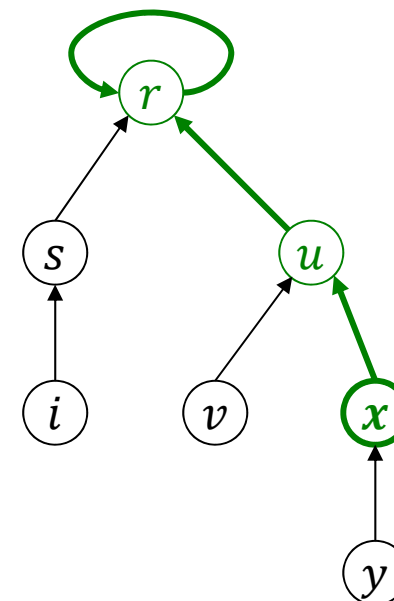
Proof:

Disjoint-Set Forests
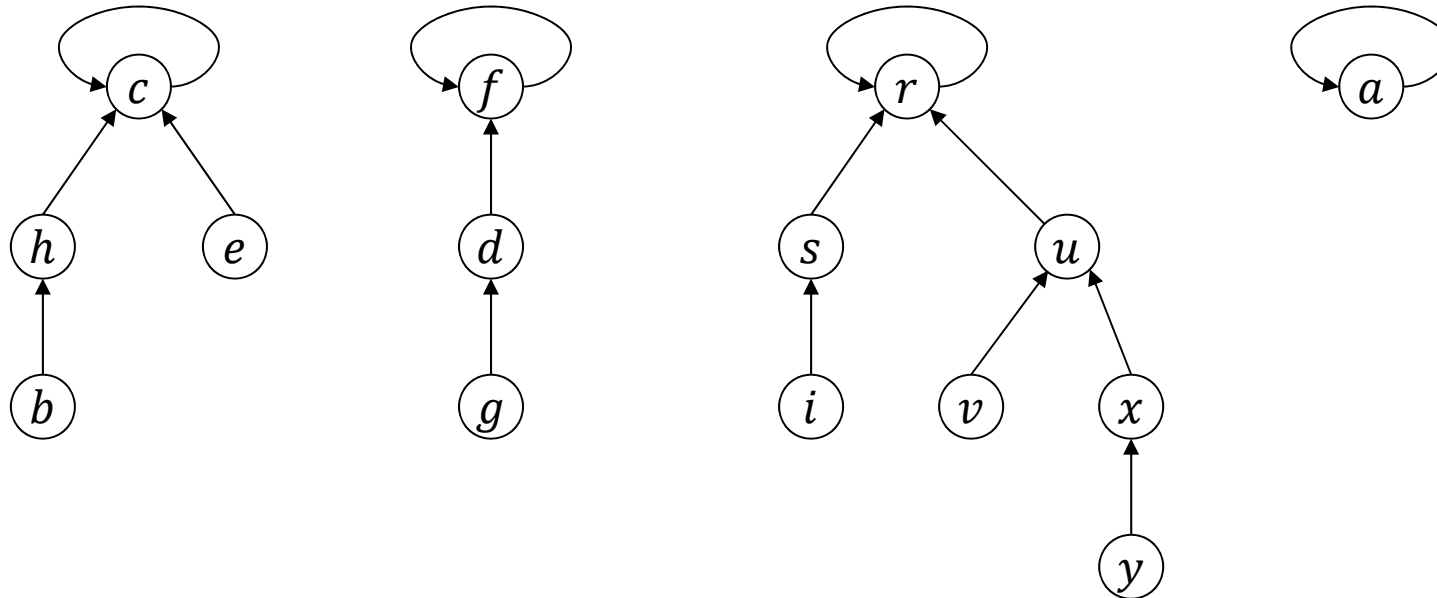
- Represent each set by a tree

- Representative of a set is the root of the tree
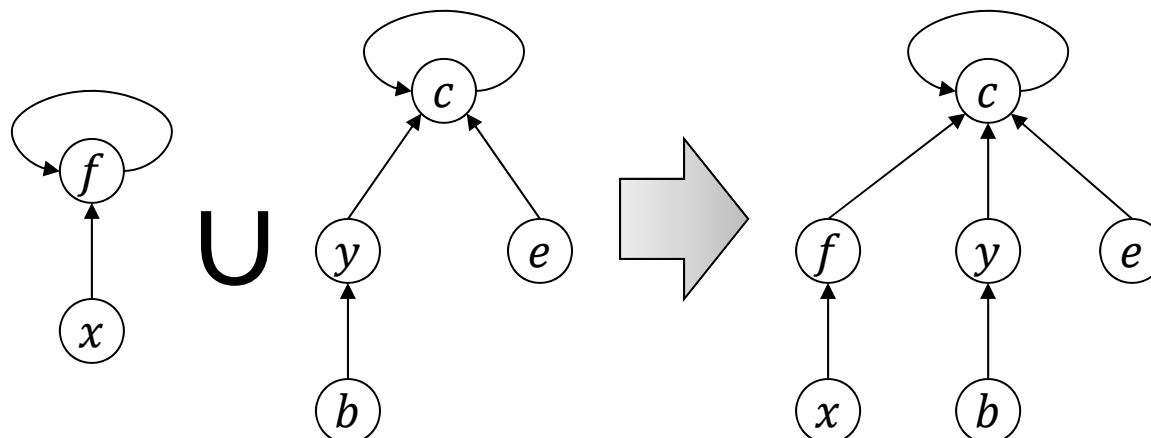
Disjoint-Set Forests

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 $\Theta(n)$

- $\mathrm{make\_set}(x_1), \mathrm{make\_set}(x_2), \ldots, \mathrm{make\_set}(x_n),$
  $\mathrm{union}(x_1, x_2), \mathrm{union}(x_1, x_3), \ldots, \mathrm{union}(x_1, x_n)$

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
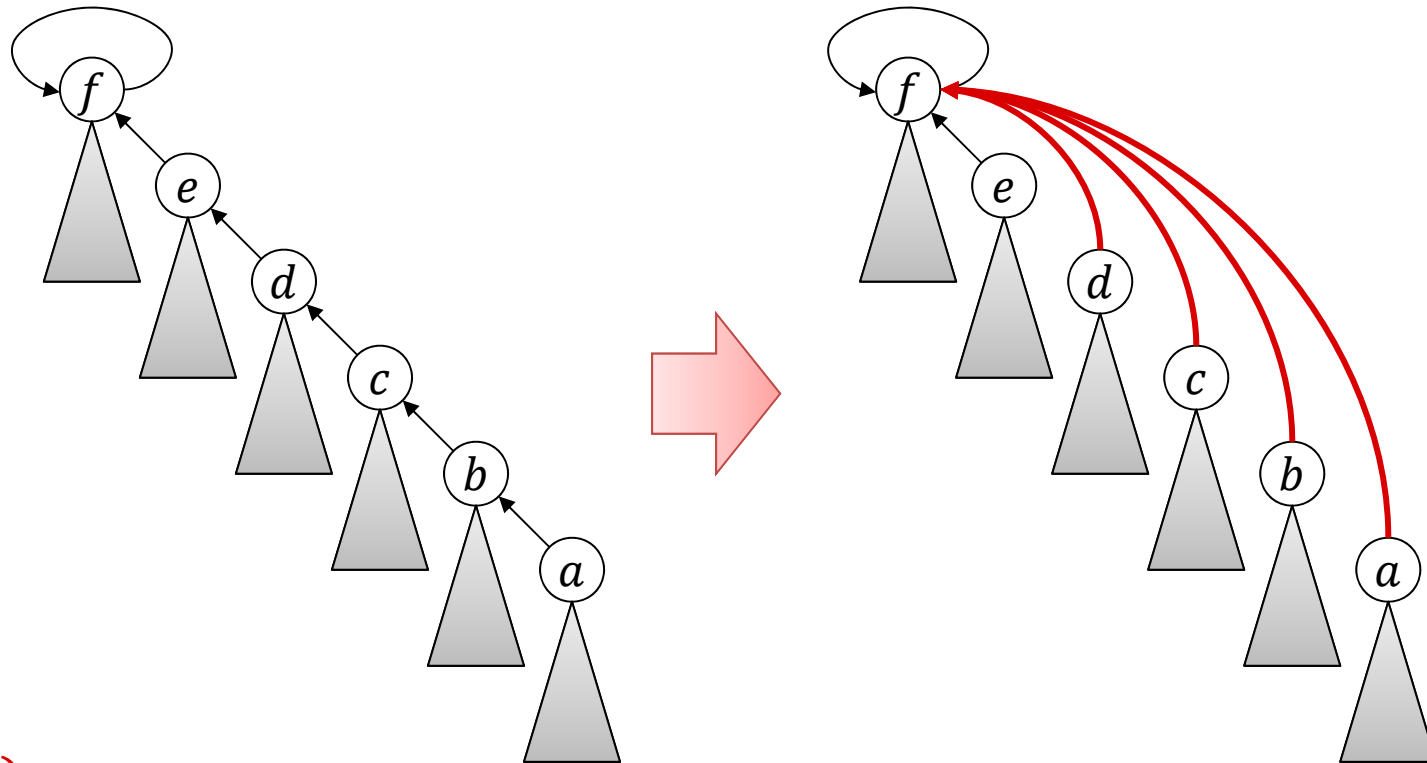
Linked List with Weighted Union Heuristic:

- 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** $a \neq a.parent$ **then**

2. $a.parent := \mathrm{find}(a.parent)$

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
  $\rightarrow$ at most $n - 1$ are union-operations

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

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

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

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

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