Algorithms
Chapter 21
DS for Disjoint Sets

Associate Professor: Ching-Chi Lin
林清池 副教授
chingchi.lin@gmail.com

Department of Computer Science and Engineering
National Taiwan Ocean University

Outline

- **Disjoint-set operations**
- Linked list representation of disjoint sets
- Disjoint-set forests
- Analysis of union by rank with path compression

Overview

- **Disjoint-set data structures**
  - Also known as "union find."
  - Maintain a collection $S = \{S_1, S_2, \ldots, S_k\}$ of **disjoint dynamic** sets.
  - Each set is identified by a **representative**, which is some member of the set.
  - Doesn't matter which member is the representative, as long as if we ask for the representative twice without modifying the set, we get the same answer both times.

Operations

- **Disjoint-set data structures** support the following three operations.
  - MAKE-SET(x): create a new set $S_i = \{x\}$, and add S_i to S.
  - UNION(x, y): unite the dynamic sets that contain x and y, say S_x and S_y, into a new set.
    - if $x \in S_x$, $y \in S_y$, then $S \leftarrow S - S_x - S_y \cup \{S_x \cup S_y\}$.
    - The representative of the resulting set is any member of $S_x \cup S_y$.
    - Since we require the sets in the collection to be disjoint, we "destroy" sets S_x and S_y.
  - FIND-SET(x): return the representative of the set containing x.

Analyzing the running times

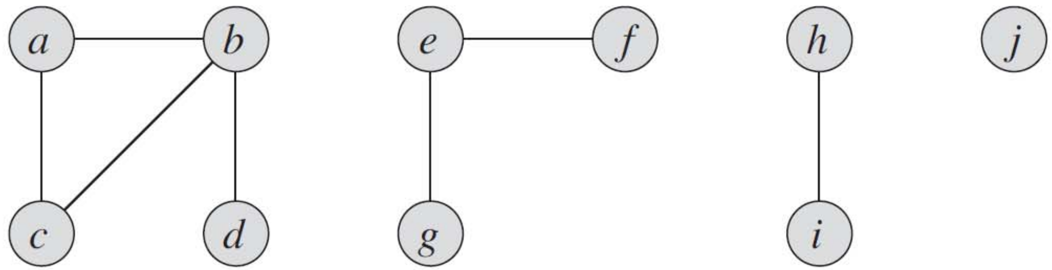
- Two parameters:
  - n = number of elements = number of MAKE-SET operations.
  - m = total number of MAKE-SET, UNION, and FIND-SET operations.
- Analysis:
  - $m \geq n$.
  - Have at most $n - 1$ UNION operations.
  - Assume that the first n operations are MAKE-SET.

An application

- **Determining the connected components**
  - For a graph $G = (V, E)$, vertices u, v are in same connected component if and only if there's a path between them.
  - Connected components partition vertices into equivalence classes.

```
CONNECTED-COMPONENTS(G)
    for each vertex v ∈ V[G]
        MAKE-SET(v)
    for each edge (u, v) ∈ E[G]
        if FIND-SET(u) ≠ FIND-SET(v)
            UNION(u, v)
```

```
SAME-COMPONENT(u, v)
    if FIND-SET(u) = FIND-SET(v)
        return TRUE
    else return FALSE
```



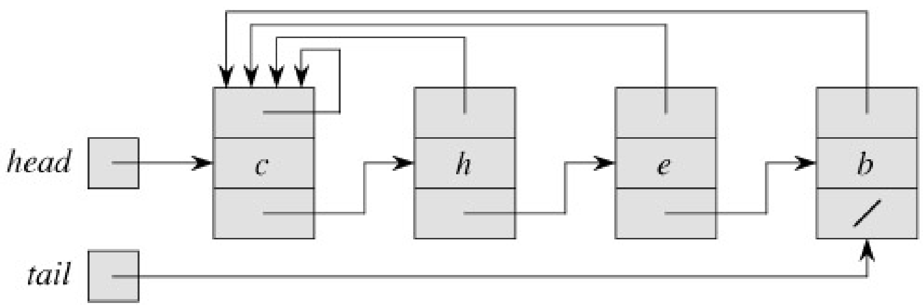
Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

Outline

- Disjoint-set operations
- **Linked list representation of disjoint sets**
- Disjoint-set forests
- Analysis of union by rank with path compression

Linked list representation

- The first object in each linked list serves as its set's representative.
- Each object in the linked list contains
  - a set member,
  - a pointer to the next set member, and
  - a pointer back to the representative.
- Each list maintains pointers **head**, to the representative, and **tail**, to the last object in the list.



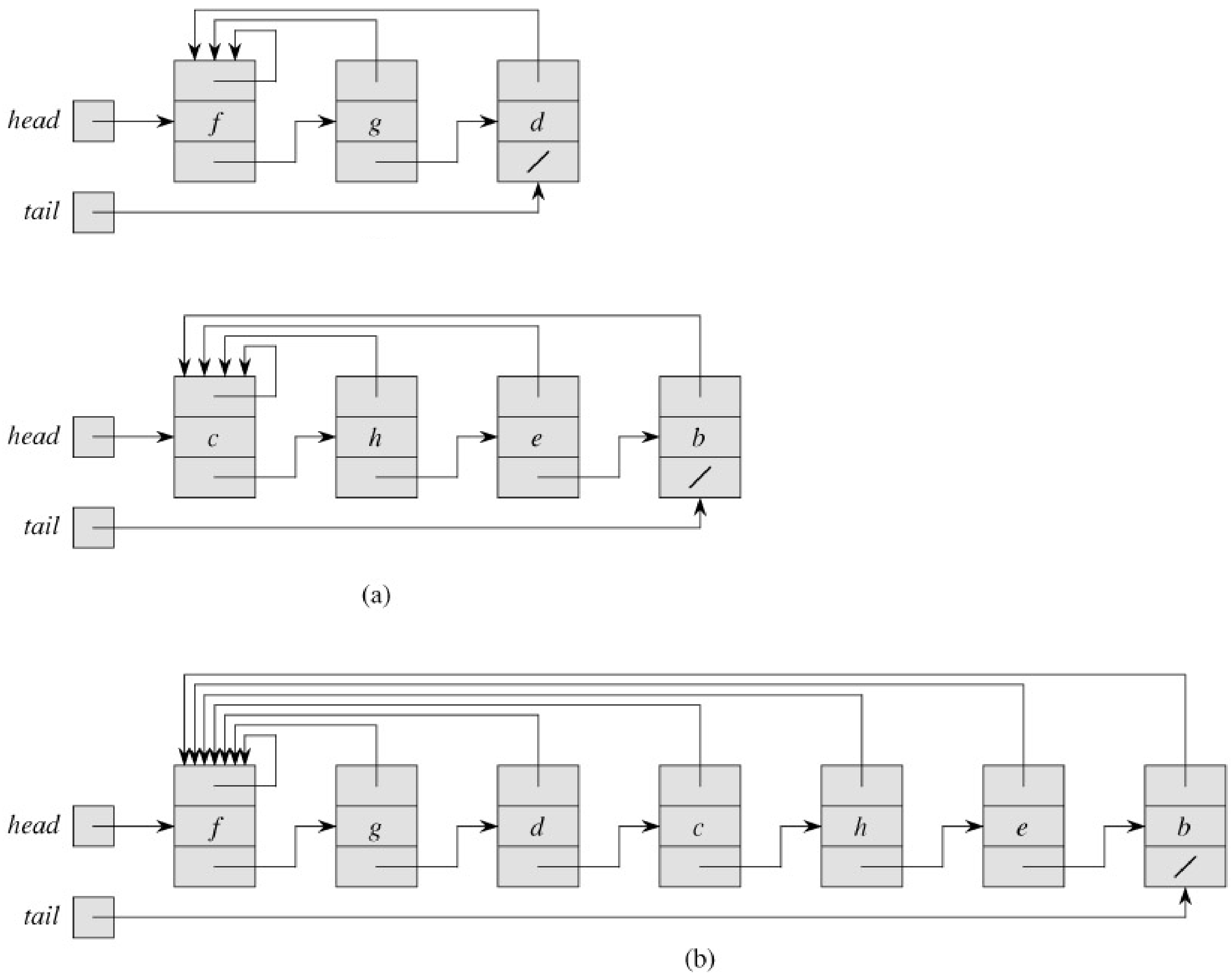


Figure 21.2 (a) Linked-list representations of two sets. (b) The result of UNION(g, e).

Operations

- MAKE-SET(*x*): create a new linked list whose only object is *x*.
  - *O*(1) time.
- FIND-SET(*x*):return the pointer from *x* back to the representative.
  - *O*(1) time.
- UNION(*x*, *y*): append *y*'s list onto the end of *x*'s list.
  - Use *x*’s tail pointer to find the end.
  - Need to update the representative for each object on *y*'s list.
  - Take time linear in the length of *y*'s list.

Worst case for implementation of the union

- Suppose that we have objects x_1, x_2,..., x_n and execute the following sequence of operations.

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
⋮	⋮
UNION(x_n, x_{n-1})	$n - 1$

- The running time for the $2n - 1$ operations is $\Theta(n^2)$.
- The amortized time of an operation is $\Theta(n)$.

A weighted-union heuristic$_{1/2}$

- Append the smaller list onto the longer.
- With this simple **weighted-union heuristic**, a single union can still take $\Omega(n)$ time, e.g., if both sets have $n/2$ members.
- **Theorem 21.1** Using the weighted-union heuristic, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, takes $O(m+n\lg n)$ time.

Proof:

- Each MAKE-SET and FIND-SET still takes $O(1)$, and there are $O(m)$ of them.
- How many times can each object's representative pointer be updated?
  - It must be in the smaller set each time.

A weighted-union heuristic$_{2/2}$

times updated	size of resulting set
1	≥ 2
2	≥ 4
3	≥ 8
$\vdots$	$\vdots$
k	$\geq 2^k$
$\vdots$	$\vdots$
$\lg n$	$\geq n$

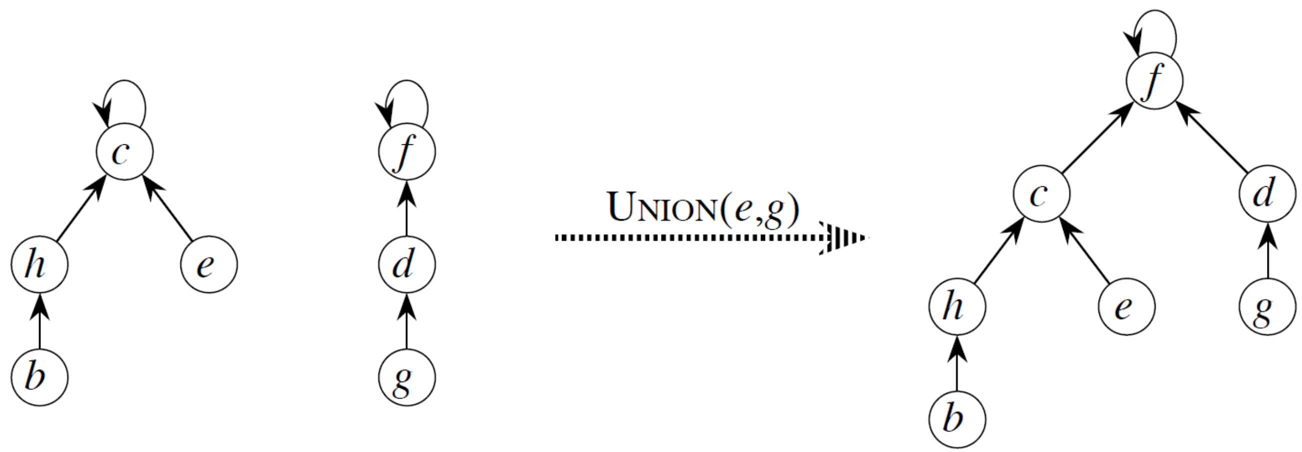
- The first time x's representative pointer was updated, the resulting set must have had at least 2 members.
- Therefore, each representative is updated $\leq \lg n$ times.
- The total time used in updating pointers over all UNION operations is thus $O(n \lg n)$.
- The total time for the entire sequence is thus $O(m + n \lg n)$.

Outline

- Disjoint-set operations
- Linked list representation of disjoint sets
- **Disjoint-set forests**
- Analysis of union by rank with path compression

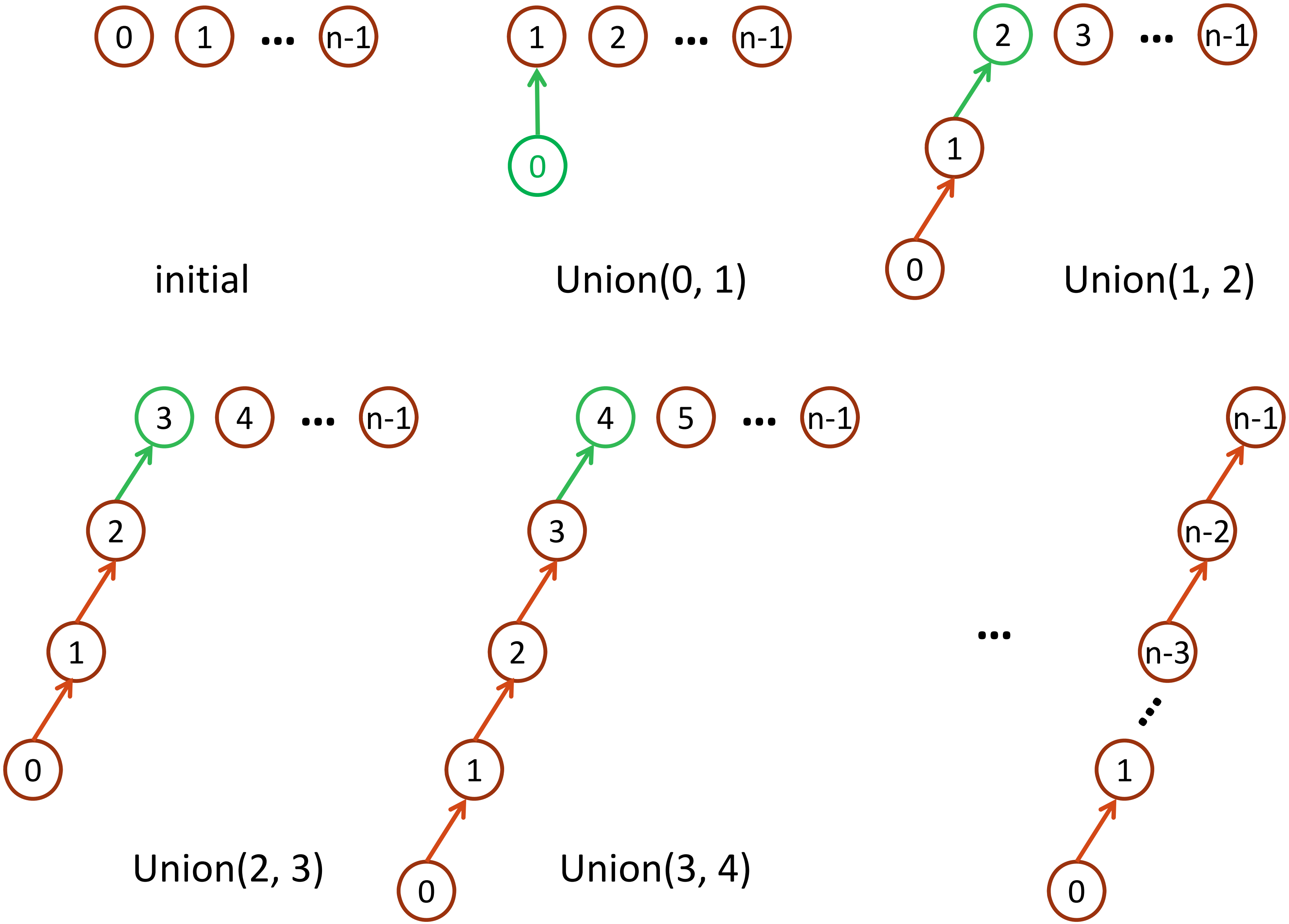
Disjoint-set forest

- In a **disjoint-set forest**:
  - Each tree represents one set;
  - Each member points only to its parent;
  - The root contains the representative; and
  - The root is its own parent.



Operations

- MAKE-SET(x): creates a tree with just one node.
  - $O(1)$ time.
- FIND-SET(x): follow parent pointers until we find the root.
  - $O(h)$ time.
  - The nodes visited on this path toward the root constitute the **find path**.
- UNION(x, y): causes the root of one tree to point to the root of the other.
  - $O(h)$ time.
- **Problem**: A sequence of $n - 1$ UNION operations may create a tree that is just a linear chain of n nodes.


initial
Union(0, 1)
Union(1, 2)
Union(2, 3)
Union(3, 4)
0
1
2
3
4
5
n-3
n-2
n-1
...

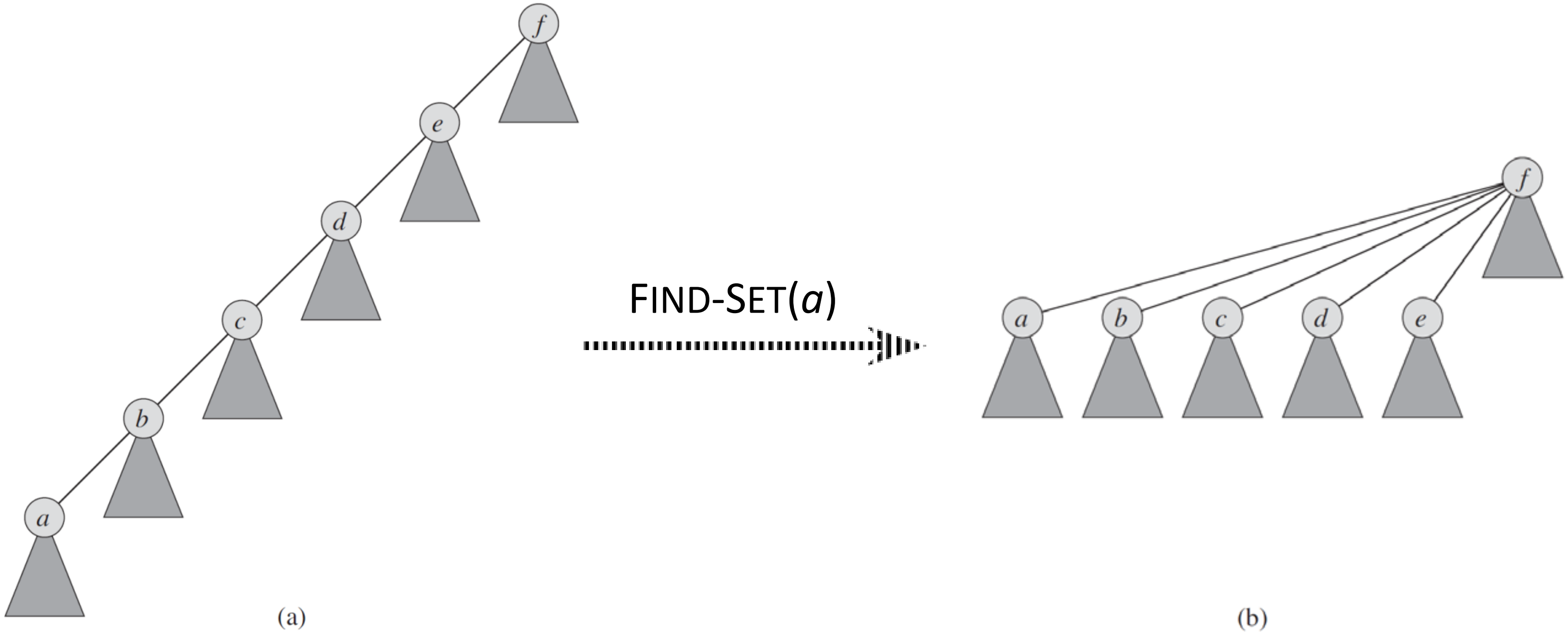
Heuristics to improve the running time

- By using two heuristics, however, we can achieve a running time that is almost linear in the total number of operations *m*.

- **Union by rank**: make the root of the tree with fewer nodes a child of the root of tree with more nodes.
  - Don't actually use **size**.
  - Use **rank**, which is an upper bound on height of node.
  - Make the root with the smaller rank into a child of the root with the larger rank.

- **Path compression**: make all nodes on the find path direct children of root.

Path compression



- Triangles represent subtrees whose roots are the nodes shown.
- In Figure b, each node on the find path now points directly to the root after executing FIND-SET(*a*).

Pseudocode for disjoint-set forest

```
MAKE-SET(x)
    p[x] ← x
    rank[x] ← 0

UNION(x, y)
    LINK(FIND-SET(x), FIND-SET(y))

FIND-SET(x)
    if x ≠ p[x]
        p[x] ← FIND-SET(p[x])
    return p[x]
```

```
LINK(x, y)
    if rank[x] > rank[y]
        p[y] ← x
    else p[x] ← y
        if rank[x] = rank[y]
            rank[y] ← rank[y] + 1
```

- The FIND-SET procedure is a **two-pass method**:
  - it makes one pass up the find path to find the root; and
  - a second pass back down the find path to update each node to point directly to root.

Effect of the heuristics on the running time$_{1/2}$

- **Union by rank** yields a running time of $O(m \lg n)$.

- **Path-compression** gives a worst-case running time

$$\Theta(n + f \cdot (1 + \log_{2+f/n} n)).$$

  - n = number of MAKE-SET operations.
  - f = number of FIND-SET operations.

Effect of the heuristics on the running time$_{2/2}$

- When we use both union by rank and path compression, the worst-case running time is $O(m\alpha(n))$, where $\alpha(n)$ is a very slowly growing function.

- In any conceivable application, $\alpha(n) \le 4$.

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \le n \le 2, \\ 1 & \text{for } n = 3, \\ 2 & \text{for } 4 \le n \le 7, \\ 3 & \text{for } 8 \le n \le 2047, \\ 4 & \text{for } 2048 \le n \le A_{4}(1). \end{cases}$$

$A_4(1) >> 10^{80}$

“>>” = “much-greater-than”