Chapter 21

Data Structures for Disjoint Sets

The slides for this course are based on the course textbook: Cormen, Leiserson, Rivest, and Stein, Introduction to Algorithms, 3rd edition, The MIT Press, McGraw-Hill, 2010.

Many of the slides were provided by the publisher for use with the textbook. They are copyrighted, 2010.

These slides are for classroom use only, and may be used only by students in this specific course this semester. They are NOT a substitute for reading the textbook!

Chapter 21 Topics

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

Disjoint Sets

Problem: Maintain a collection of disjoint dynamic (changing over time) sets $\zeta = \{S_1, S_2, \ldots, S_n\}$

- each set is identified by a representative
- a representative is some member of the set
- It often does not matter which element is the representative
- if we ask for the representative twice without modifying the set, we should get the same answer
- Also known as “union find”

Disjoint Set Operations

MAKE-SET (x):

- Create new set whose only member is *x*
- the representative will also be *x*
- x cannot be a member of some other set.
- $S_i = \{x\}$, and $\zeta \leftarrow \zeta \cup S_i$

Disjoint Set Operations

UNION (x, y):

- Create a new set that is the union of the set containing *x* and the set containing *y*
- destroy sets *x* and *y* (maintains disjoint property)

$\zeta \leftarrow \zeta - S_x - S_y \cup \{S_x \cup S_y\}$

Disjoint Set Operations

- FIND-SET (x): Return a pointer to the representative of the (unique) set containing *x*.

Analysis

Analysis is in terms of two parameters:

n = number of elements, and also

n = number of MAKE-SET operations

m = total number of operations

- The constraint $m \geq n$ holds. Why? Because MAKE-SET counts toward the total number of operations

Analysis

- How many sets after n-1 Unions? Only 1. So maximum number of Union operations possible is n-1.
- Assume that the first n operations are MAKE-SET

Application: Dynamic 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(V, E)

for each vertex $v \in V$
    do MAKE-SET(v)
for each edge $(u, v) \in E$
    do if FIND-SET$(u) \neq$ FIND-SET(v)
        then UNION(u, v)

SAME-COMPONENT(u, v)

if FIND-SET$(u) =$ FIND-SET(v)
  then return TRUE
  else return FALSE

Note: If actually implementing connected components,

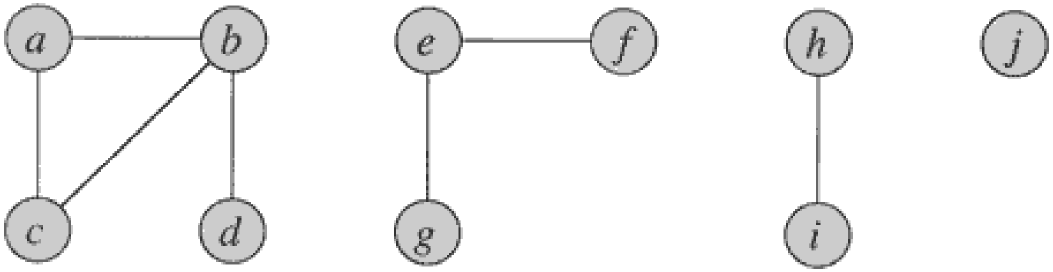
- each vertex needs a handle to its object in the disjoint-set data structure,
- each object in the disjoint-set data structure needs a handle to its vertex.

Connected-Components Algorithm

```
Connected-Components(G)
  for each vertex v ∈ V[G]do
     Make-Set(v)
  for each edge (u,v) ∈ E[G]do
     if Find-Set(u) ≠ Find-Set(v)
        then Union(u,v)
```

Same-Component Algorithm

```
Same-Component(u,v)
  if Find-Set(u) = Find-Set(v)
     then return TRUE
     else return FALSE
```



(a)

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)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

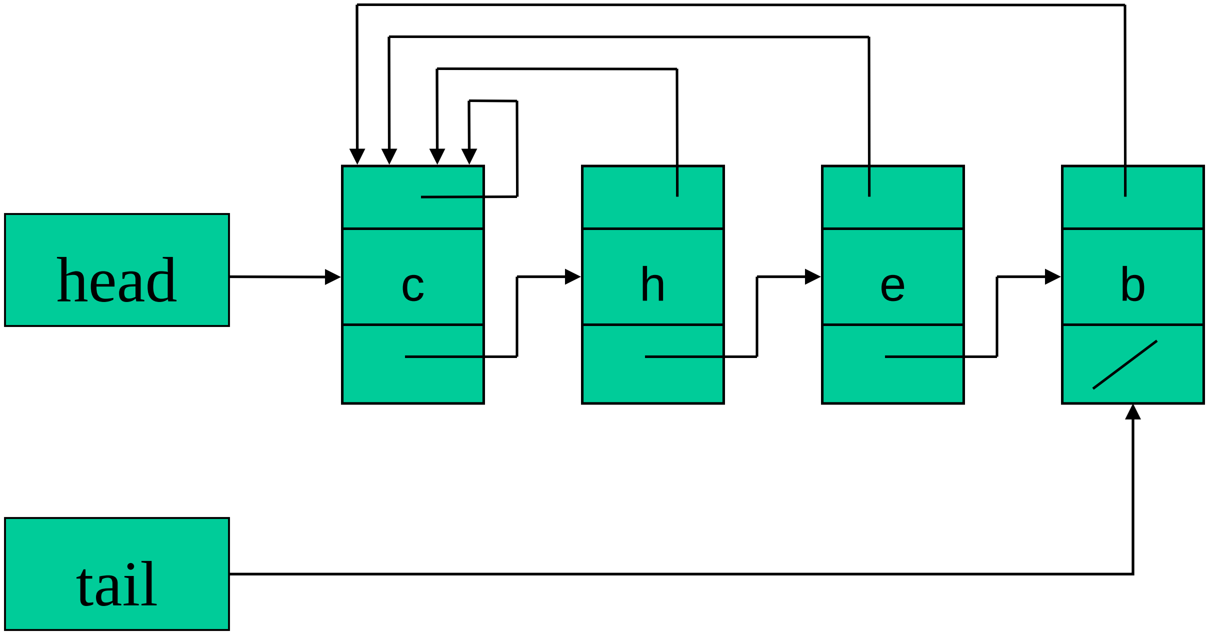
(b)

Figure 21.1 **(a)** A graph with four connected components: $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h, i\}$, and $\{j\}$. **(b)** The collection of disjoint sets after each edge is processed.

Linked List Implementation of Disjoint Set

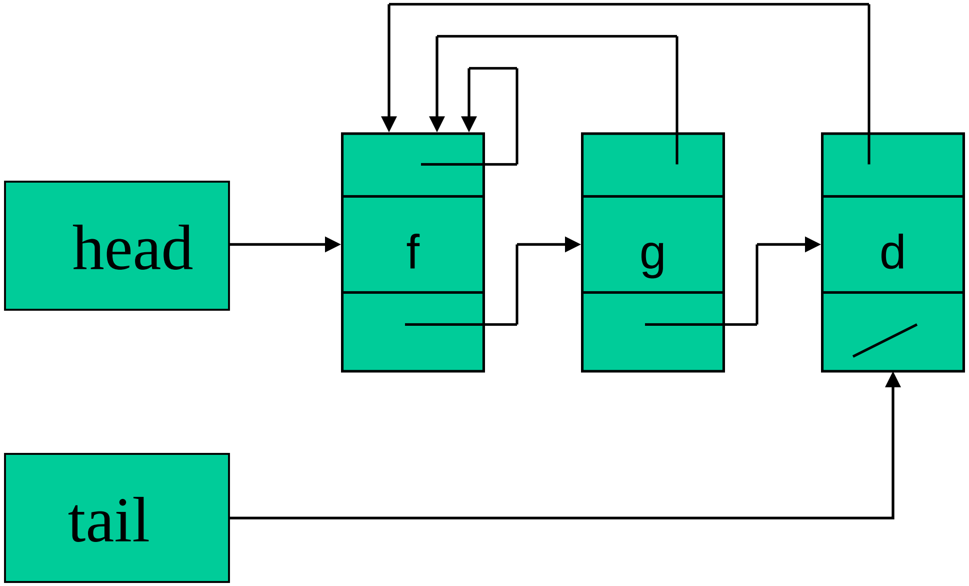
- Each set is represented as a linked list.
- Each node of a list contains:
  - the object
  - a pointer to the next item in the list
  - a pointer back to the representative for the set

Linked List Implementation of Disjoint Set



$x = \{c, h, e, b\}$

Linked List Implementation of Disjoint Set



y = {f, g, d}

Implementation of Operations

MAKE-SET (x):

- Create new linked list whose only object is *x*.

FIND-SET (x):

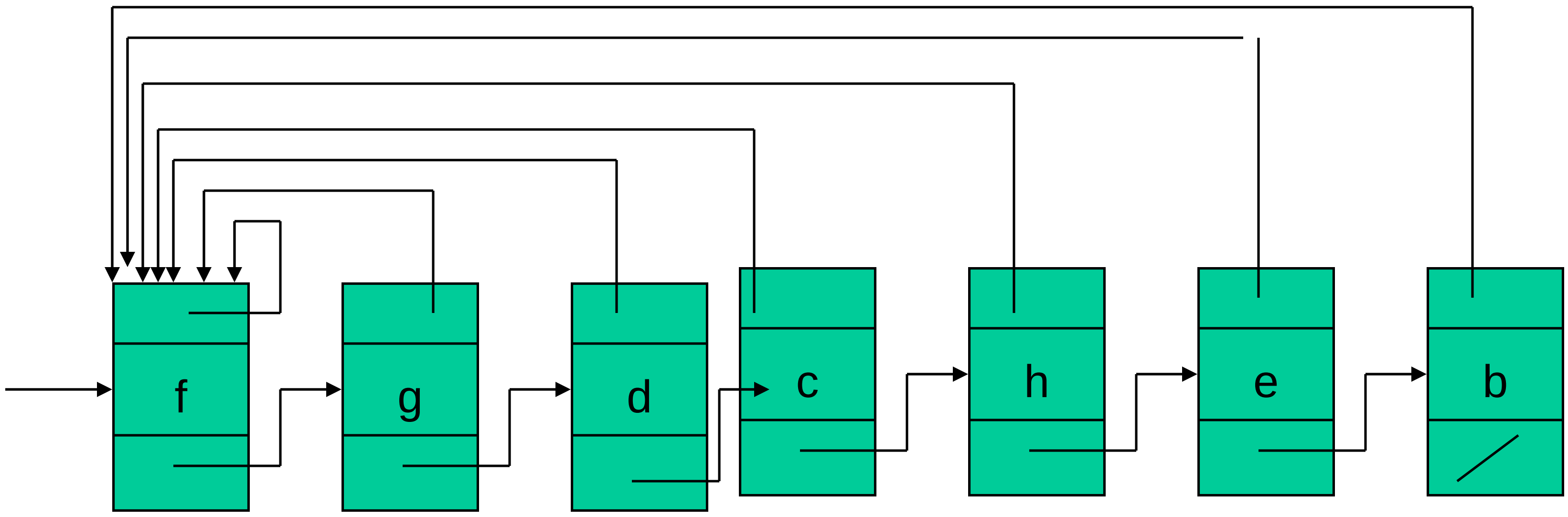
- Return the pointer from *x* back to the representative.

Implementation of Operations

UNION (x, y):

- *Append x's list to y's list using the tail pointer for y's* list. Update the representative pointer for each object that was in *x*'s list.
  - **Weighted-union heuristic:** Store length of list in each list so we can be sure to append the shorter list to the end of the longer list.

Union of the Two Sets x and y



$x \cup y = \{c, h, e, b, f, g, d\}$

Analysis

- Suppose we have *m* operations
- All *m* are UNION
- Max size of a set is *n*
- So complexity in worst case would be

  $O(m(n-1)) = O(m^2)$
- Can we do better with amortized analysis?

Amortized Analysis

- See chapter 17
- In amortized analysis, the time required to perform a sequence of data-structure operations is averaged over all the operations performed.
- Amortized analysis can be used to show that the average cost of an operation is small, even though a single operation within the sequence may be expensive.

Amortized Analysis

- Amortized analysis is not average-case analysis:
- Average-case analysis involves determining the *probability* of various cases occurring.
- Amortized analysis guarantees the *average* performance of each operation in the *worst* case.
- Example: n operations, one takes $O(n)$, but all others take $O(1)$. What's the run time?

Example

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

Analysis

We remember that $\sum_{i=1}^{n} i = \frac{n(n-1)}{2} = \frac{n^2 - n}{2}$

which is $O(n^2)$.

So we can see that, using the linked-list representation and the simple implementation of Union, a sequence of $2n$-1 operations on n objects takes $\Theta(n^2)$ time, or $\Theta(n)$ time per operation, on the average.

Two Unions

- The union of linked lists requires that we update the representative pointer for every node on "*x*'s" list
- If we are appending a large list onto a small list, this can take a while, giving θ(n) amortized cost per operation.
- Idea: append the smallest list to the largest!
- Weighted-union heuristic (choosing the smallest list to append to the largest) reduces this time

Weighted Union Heuristic

- This is an obvious thing to try – always append the shortest list to the end of the longest
- Question: Will this give us any asymptotic improvement in performance?
- In order to implement, just store the number of items in the set along with the representative.

Weighted-Union

- *Weighted-union heuristic*: always append the smaller list onto the larger.
- Does this always save time? No; consider a situation in which both sets have n/2 elements. The union still requires O(n) time.
- But suppose we have two randomly generated lists with a total of *n* elements, then the probability is small that the two lists will have the same or "almost" the same number of elements.

Theorem 21.1

Using the linked-list representation of disjoint sets and 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.

To prove this, we need to keep track of the *number of times* an element's *representative pointer* is *updated*.

Theorem 21.1 - Proof

Assuming that we are using the weighted-union heuristic, let's follow a single element, *x*:

The 1st time the set that *x* is in is unioned with another set, the set it ends up in must be at least size 2.

The 2nd time the set that *x* is in is unioned, there are two possibilities:

a. It is in the larger set. In that case, its representative pointer doesn't need to be updated, because it already points to the correct one.
b. It is in the smaller set. In that case, the set it ends up in must be of size ≥ 4. (Why? Because it currently is in a set of size ≥ 2, so the set it is unioned with must be at least that size.)

Theorem 21.1 - Proof

The 3rd time the set that *x* is in is unioned, either *x* doesn't need to have its representative pointer updated or the set it ends up in must be at least size 8.

The 4th time the set that *x* is in is unioned, either *x* doesn't need to have its representative pointer updated or the set it ends up in must be at least size 16.

. . .

Theorem 21.1

You can see where this is going; assuming that we are using the weighted-union heuristic, each time the set that *x* is in is unioned, if it had to have its representative pointer updated, it must have ended up in a set at least twice as big as its previous set.

How many total unions will we perform before *x* ends up in a set of size *n*?

That's right: a maximum of $\log_2 n$

Each of the *n* elements will be involved in $\lceil \lg n \rceil$ unions, so their represetative pointers will have to be updated $\leq$ lg n times each, for a total cost of O(n lg n).

Proof

- We have established an upper bound of lg n on the number of times an element’s representative pointer was updated.
- Total time for updating the representative pointers is O(n lg n)
- Updating the head and tail pointers and the lists lengths costs O(1) per Union operation.
- Each Make-Set and Find-Set costs O(1), and there are *m* of them, for a total cost of O(m).
- Total time for *m* operations is O(m + n lg n)

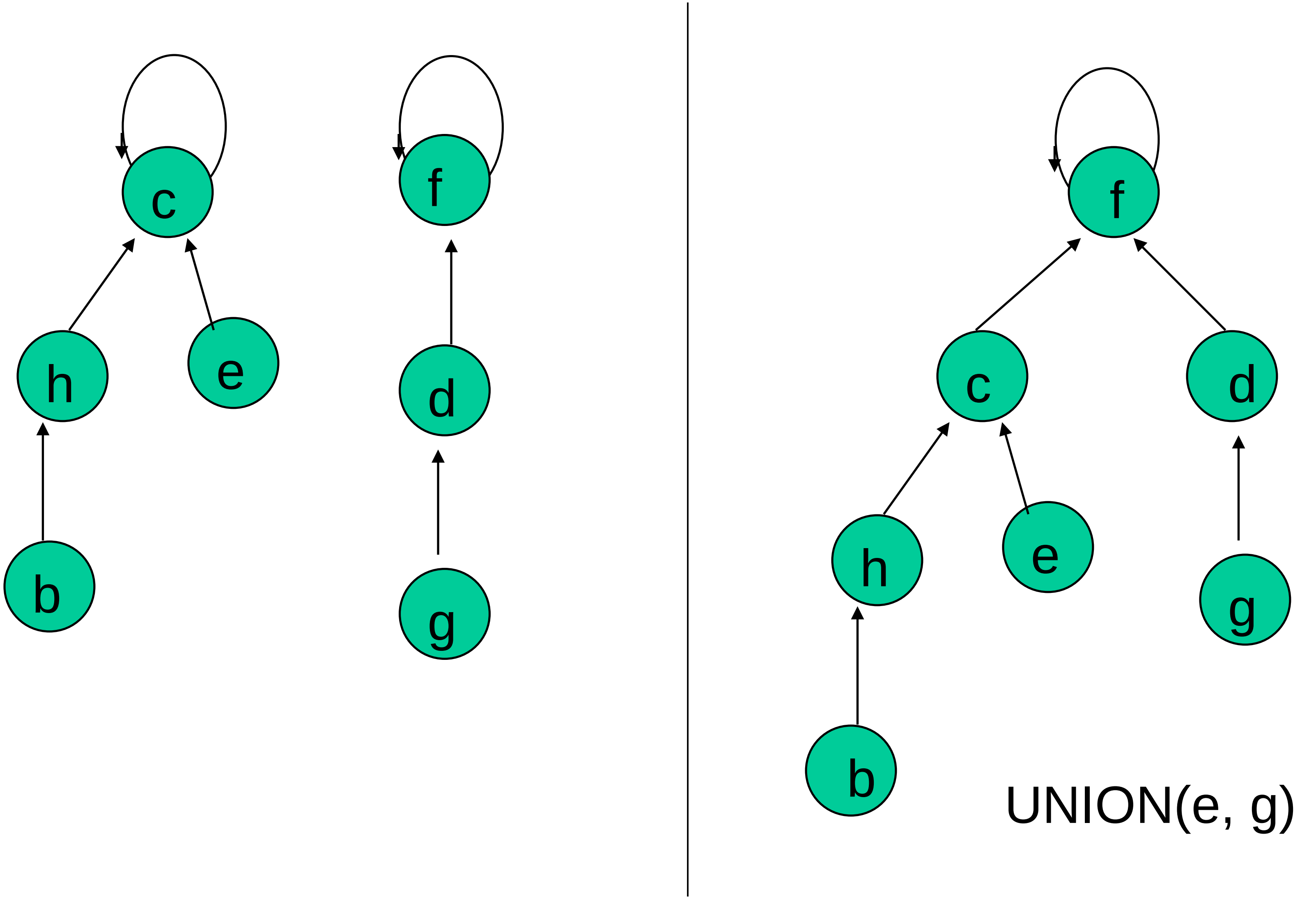
Another Improvement

- Use trees to represent each set instead of linked lists
- Each member points to its parent only.
- Root points to itself
- Straightforward implementation is no faster than linked list
- Two heuristics can make it very fast

Tree Implementation of Disjoint Set

- **Disjoint-set forests:**
  - Each tree represents one set.
  - Each node contains one member.
  - Each member points only to its parent.
  - Each root contains the representative and is its own parent.

Example of Disjoint-Set Forest



Implementation of Operations

MAKE-SET (x):

- Create new tree whose only object is x.

FIND-SET (x):

- Follow parent pointers from x to the root; return pointer to the root.

UNION (x, y):

- Make root of one tree point to root of the other.

Analysis

- A series of *n* UNION operations could create a tree that is a linear chain of *n* nodes.
- A FIND-SET operation could then require O(n)
- Sequence of *m* operations is still $O(m^2)$

Disjoint Forests Heuristics

- As it stands, our technique is not so good because we could still get a linear chain of nodes
- But there are some great heuristics yet to be used.
- Heuristics for improving performance
  - Union by rank
  - Path compression
- Running time using both heuristics
  - $O(m \bullet \alpha(m,n))$
    - $\alpha(m,n)$ is inverse of Ackermann's function
    - $\alpha(m,n)$ is essentially constant for almost all conceivable applications of a disjoint-set data structure

Union by Rank

Union by rank: make the root of the smaller tree (fewer nodes) the child of the root of the large tree

- Don't actually use *size*.
- Use *rank*: the rank of a node is the height of the subtree rooted at that node
- Make the root with the smaller rank into a child of the root with the larger rank

Union by Rank

When we use union by rank to merge two trees, we always choose the shorter tree to merge with the taller one.

This results in a combined tree that is no higher than the taller of the two trees.

The only time merging two trees produces a taller combined tree is when both original trees were the same size.

Union by Rank

What are the implications of union by rank?

- A root node of rank k is created by merging two trees of rank $k - 1$.
- So a root node of rank k has at least 2^k nodes in its tree.
- If we started off with n elements, then there are at most $n/2^k$ nodes of rank k.
- Consequently, the maximum rank is $\log_2 n$.
- So all trees have a height $\leq \log_2 n$.
- Therefore, FIND-SET and UNION have a running time of $O(\log_2 n)$.

Path Compression

- Use during FIND-SET operations to make each node on the find path point directly to the root.
- See next slide

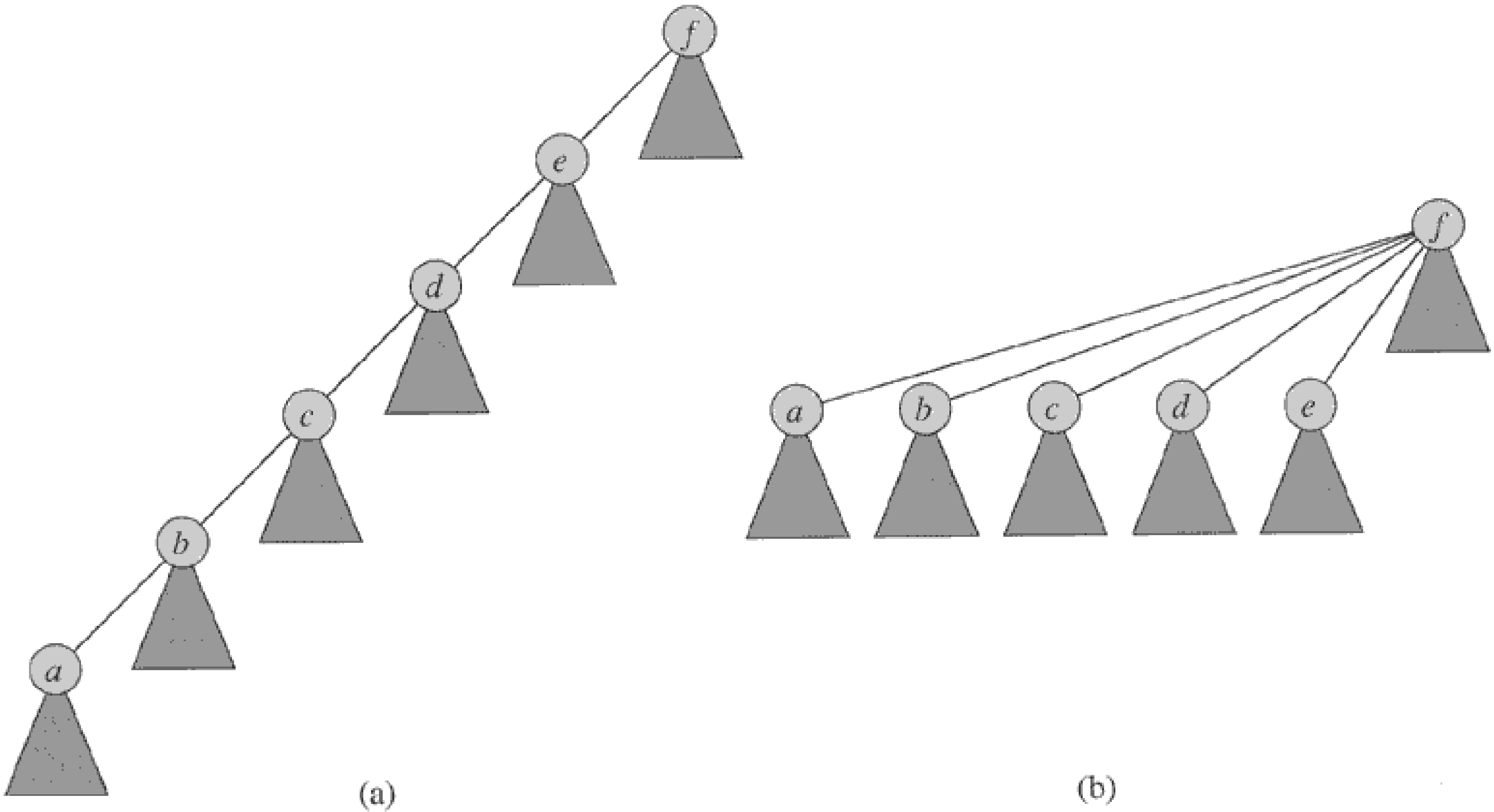


Figure 21.5 Path compression during the operation FIND-SET. Arrows and self-loops at roots are omitted. **(a)** A tree representing a set prior to executing FIND-SET(a). Triangles represent subtrees whose roots are the nodes shown. Each node has a pointer to its parent. **(b)** The same set after executing FIND-SET(a). Each node on the find path now points directly to the root.

Implementation

- With each node *x*, maintain an integer rank[x] (upper bound of height)
- rank[x] is 0 for a new subtree made with MAKE-SET
- FIND-SET leaves rank unchanged
- UNION makes the root of higher rank the root of the new tree
- p[x] is the parent of *x*

Algorithm for Disjoint-Set Forests

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

FIND-SET (x)
1  if x ≠ p[x]
2     then p[x] ← FIND-SET (p[x])
3  return p[x]
```

Alg. for Disjoint-Set Forests (cont)

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

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

Analysis for Disjoint-Set Forests

- If we use both union by rank and path compression, the running time is O(m*α*n)
- Here are some values of α*n:*

n	***αn***
0 - 2	0
3	1
4 - 7	2
8 - 2047	3
2048 – A(4, 1)	4

Analysis for Disjoint-Set Forests

What is $\alpha_4(1)$?

A is the Akermann function:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Akermann's function grows extremely rapidly – faster than factorial, exponential, etc.

α is 1/*A*, or the inverse of Akermann's function; it grows extremely slowly.

Conclusion

Disjoint sets may be represented by several different data structures: lookup table, linked-list, trees.

We need to perform certain operations on these disjoint sets.

The choice of data structure dramatically affects the running time of the operations.

So, choose an appropriate data structure!