Path-Compression
during Find-Set, make each node on the find path point
directly to root

Using union by rank and path compression:
$O(m\alpha(m,n))$ running time for $m+n$ operations
$\alpha(m,n)$ is a function that is almost constant
(for any conceivable application, $\alpha(m,n)<5$)
The best implementation for disjoint sets: disjoint-set forest

Find-Set -- chase parent pointers until get to root (representative)
Union -- make root of one tree point to the root of the other tree

Augmentations that improve running time:
Union by rank -- just like weighted union
the root of the tree with fewer nodes points to the root of larger tree

Using Union by rank: $O(m + n \lg n)$ running time for $m+n$ operations
Theorem: with weighted-union, a sequence of \( s+n \) operations, 
n of which are Make-Set, takes \( O(s+n \log n) \) time

Proof: -- \( n \) Make-Set operations take \( O(n) \)
   -- all Find-Sets: at most \( O(s) \)
   -- all Unions: at most \( O(s) \) for making 1 set and \( \sum \) for changing pointers
   -- consider some element \( x \); each time \( x \)'s representative pointer is updated, \( x \) must have started in the smaller set:
     1st time \( x \)'s pointer was updated, resulting set had \( \geq 2 \) members
     2nd time \( \geq 4 \) members
     3rd time \( \geq 8 \) members
     ....
     \( i \)-th time \( \geq 2^i \) members
     \( (\log n) \)-th time \( \geq n \) members
   -- the largest any set can ever become is having \( n \) members,
     so \( x \)'s pointer can be updated at most \( \log n \) times
   -- there are \( n \) objects \( "x" \) -- total time for all Unions is \( O(n \log n) \)
   -- \( T(n) = O(n \log n) + O(m) + O(n) = O(m+n \log n) \)
augmentation: each element has a pointer back to representative

Make-Set takes $O(1)$ time
Union takes $O(\text{length of list})=O(n)$ time

Find-Set takes $O(1)$

Weighted-union: always append the smaller list to the larger for this, need another augmentation:
keep the size of the list with the representative
Connected-Components(G)
for each vertex v in V
  Make-Set(v)
for each edge (u,v) in E
  if \( \text{Find-Set}(u) \neq \text{Find-Set}(v) \) then Union(u,v)

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

Implementation of disjoint sets: each set as a linked list
first element in list is the representative

Make-Set takes O(1) time
Union takes O(length of list)=O(n) time
Find-Set takes O(length of list)=O(n) time
TOPIC 12: Disjoint-set Data Structure

Disjoint-set data structure

- maintain a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets
- each set is identified by a representative (ex: some member of set)

operations needed on this data structure:
- Make-Set(x) -- create a new set whose only member is x
- Union(x,y) -- unite the sets containing x and y, say $S_x$ and $S_y$, into a new set; $S_x$ and $S_y$ get destroyed
- Find-Set(x) -- find the set (representative) containing x

Application: finding connected components of a graph

$$G=(V,E) \quad V=\text{set of vertices} \quad E=\text{set of edges}$$