Data Structures for Disjoint Sets

Application:

Connected Components

Minimum Spanning Tree

Dr. Md. Abul Kashem Mia, Professor, CSE Dept, BUET

Disjoint Sets

- Some applications require maintaining a collection of disjoint sets.

- A Disjoint Set ***S*** is a collection of sets $S_1, \ldots\ldots S_n$

  where $\forall_{i \neq j} S_i \cap S_j = \phi$

- Each set has a representative which is a member of the set (usually the minimum if the elements are comparable)

Disjoint Set Operations

- Make-Set(x) – Creates a new set S_x where x is it's only element (and therefore it is the representative of the set).
  $O(1)$ time.

- Union(x, y) – Replaces S_x, S_y by $S_x \cup S_y$.
  One of the elements of $S_x \cup S_y$ becomes the representative of the new set.
  $O(\log n)$ time.

- Find(x) – Returns the representative of the set containing x
  $O(\log n)$ time.

Analyzing Operations

- We usually analyze a sequence of *m* operations, of which *n* of them are Make_Set operations, and *m* is the total of Make_Set, Find, and Union operations.
- Each union operations decreases the number of sets in the data structure, so there can not be more than *n*-1 Union operations.

Applications

- Equivalence Relations (e.g Connected Components)
- Minimum Spanning Trees

Connected Components

- Given a graph G we first preprocess G to maintain a set of connected components

  CONNECTED_COMPONENTS(G)

- Later a series of queries can be executed to check if two vertexes are part of the same connected component

  SAME_COMPONENT(u, v)

Connected Components

```
CONNECTED_COMPONENTS(G)
    for each vertex v in V[G] do
        MAKE_SET (v)

    for each edge (u, v) in E[G] do
        if FIND_SET(u) != FIND_SET(v) then
            UNION(u, v)


SAME_COMPONENT(u, v)
    if FIND_SET(u) == FIND_SET(v) then
        return TRUE
    else return FALSE
```

Connected Components : Question

- During the execution of CONNECTED-COMPONENTS on a undirected graph $G = (V, E)$ with k connected components, how many time is FIND-SET called? How many times is UNION called? Express your answers in terms of $|V|$, $|E|$, and k.

Connected Components : Solution

- FIND-SET is called $2|E|$ times.
  - FIND-SET is called twice on Line 4, which is executed once for each edge in $E[G]$.

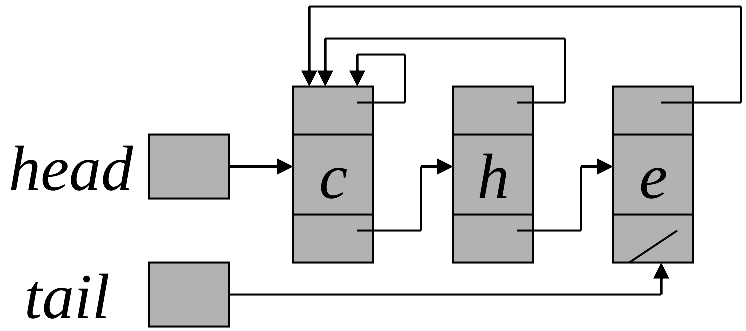
- UNION is called $|V| - k$ times.
  - Lines 1 and 2 create $|V|$ disjoint sets.
  - Each UNION operation decreases the number of disjoint sets by one. At the end there are k disjoint sets, so UNION is called $|V| - k$ times.

Disjoint-Set Implementation: Linked List

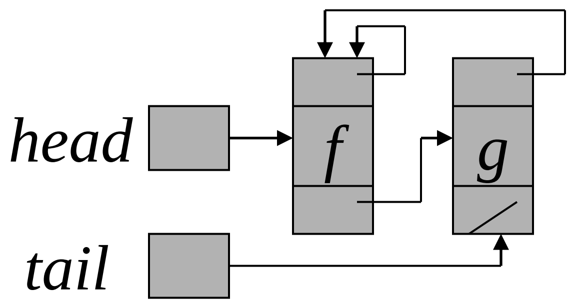
- We maintain a set of linked list, each list corresponds to a single set.
- All elements of the set point to the first element which is the representative
- A pointer to the tail is maintained so elements are inserted at the end of the list

Linked-lists for two sets

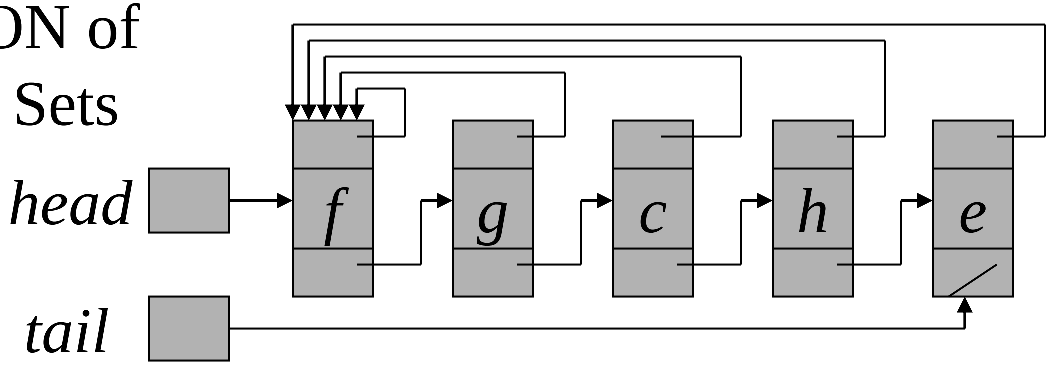
Set {*c*, *h*, *e*}



Set {*f*, *g*}



UNION of two Sets



UNION Implementation

- A simple implementation: UNION(x, y) just appends x's list to the end of y's list, updates all back-to-representative pointers in x's list to the head of y's list.
- Each UNION takes time linear in the x's length.
- Suppose n MAKE-SET(x_i) operations ($O(1)$ each) followed by n-1 UNION
  - UNION(x_1, x_2), $O(1)$,
  - UNION(x_2, x_3), $O(2)$,
  - …..
  - UNION(x_{n-1}, x_n), $O(n-1)$
- The UNIONs cost $1+2+\ldots+n-1=\Theta(n^2)$
- So $2n$-1 operations cost $\Theta(n^2)$, average $\Theta(n)$ each.
- Not good!! How to solve it ???

Weighted-Union Heuristic

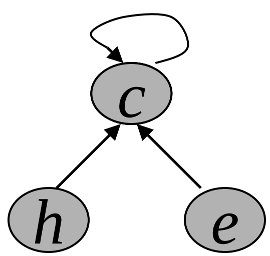
- Instead appending x to y, append the shorter list to the longer list.
- Associated a length with each list, which indicates how many elements are in the list.
- Result: a sequence of m MAKE-SET, UNION, FIND-SET operations, n of which are MAKE-SET operations.

  The running time is $O(m + n \log n)$. Why???

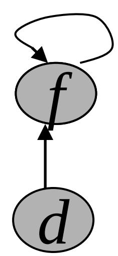
Hints: Count the number of updates to back-to-representative pointer for any x in a set of n elements. Consider that each time, the UNION will <u>at least double the length of united set</u>, it will take at most log n UNIONS to unite n elements. So each x's back-to-representative pointer can be updated at most log n times.

Disjoint-Set Implementation: Forests

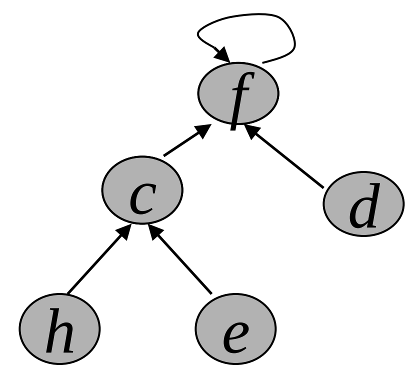
- Rooted trees, each tree is a set, root is the representative. Each node points to its parent. Root points to itself.



Set $\{c, h, e\}$



Set $\{f, d\}$



UNION

Straightforward Solution

- Three operations
  - MAKE-SET(x): create a tree containing x. $O(1)$
  - FIND-SET(x): follow the chain of parent pointers until to the root. $O(h)$, h is height of x's tree
  - UNION(x, y): let the root of one tree point to the root of the other. $O(1)$

- It is possible that n-1 UNIONs results in a tree of height n-1. (just a linear chain of n nodes).

- So n FIND-SET operations will cost $O(n^2)$.

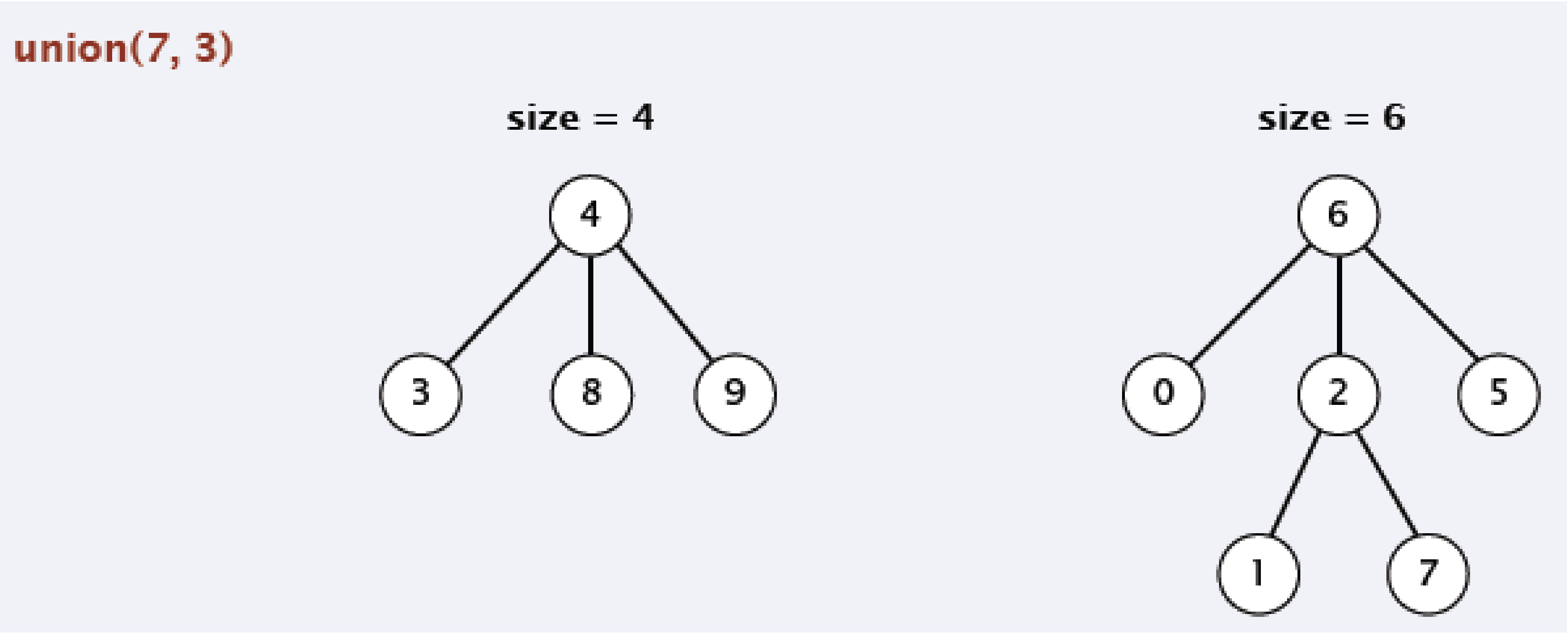
Union by Rank & Path Compression Heuristics

- Union by Rank: Each root is associated with a rank. Then when UNION, let the root with smaller rank point to the root with larger rank.
  - Link by Size, which is the number of nodes in the subtree rooted at the node
  - Link by Height, which is the height of the subtree rooted at the node

- Path Compression: used in FIND-SET(x) operation, make each node in the path from x to the root directly point to the root. Thus reduce the tree height.

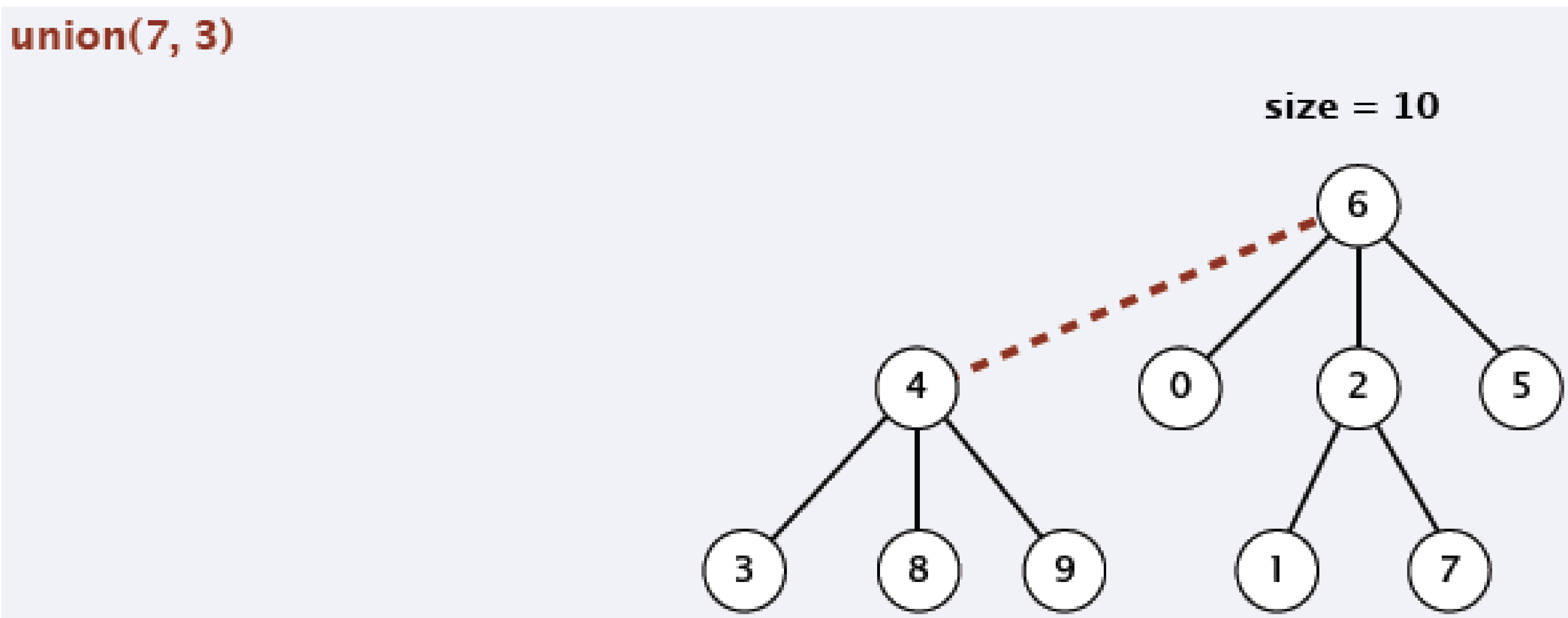
Link by Size

- Maintain a subtree count for each node, initially 1.
- Link root of smaller tree to root of larger tree (breaking ties arbitrarily).



Link by Size

- Maintain a subtree count for each node, initially 1.
- Link root of smaller tree to root of larger tree (breaking ties arbitrarily).



Link by Size

- Maintain a subtree count for each node, initially 1.
- Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

MAKE-SET (x)

$parent(x) \leftarrow x.$

$size(x) \leftarrow 1.$

FIND (x)

WHILE $(x \neq parent(x))$

  $x \leftarrow parent(x).$

RETURN $x.$

UNION-BY-SIZE (x, y)

$r \leftarrow$ FIND $(x).$

$s \leftarrow$ FIND $(y).$

IF $(r = s)$ RETURN.

ELSE IF $(size(r) > size(s))$

  $parent(s) \leftarrow r.$

  $size(r) \leftarrow size(r) + size(s).$

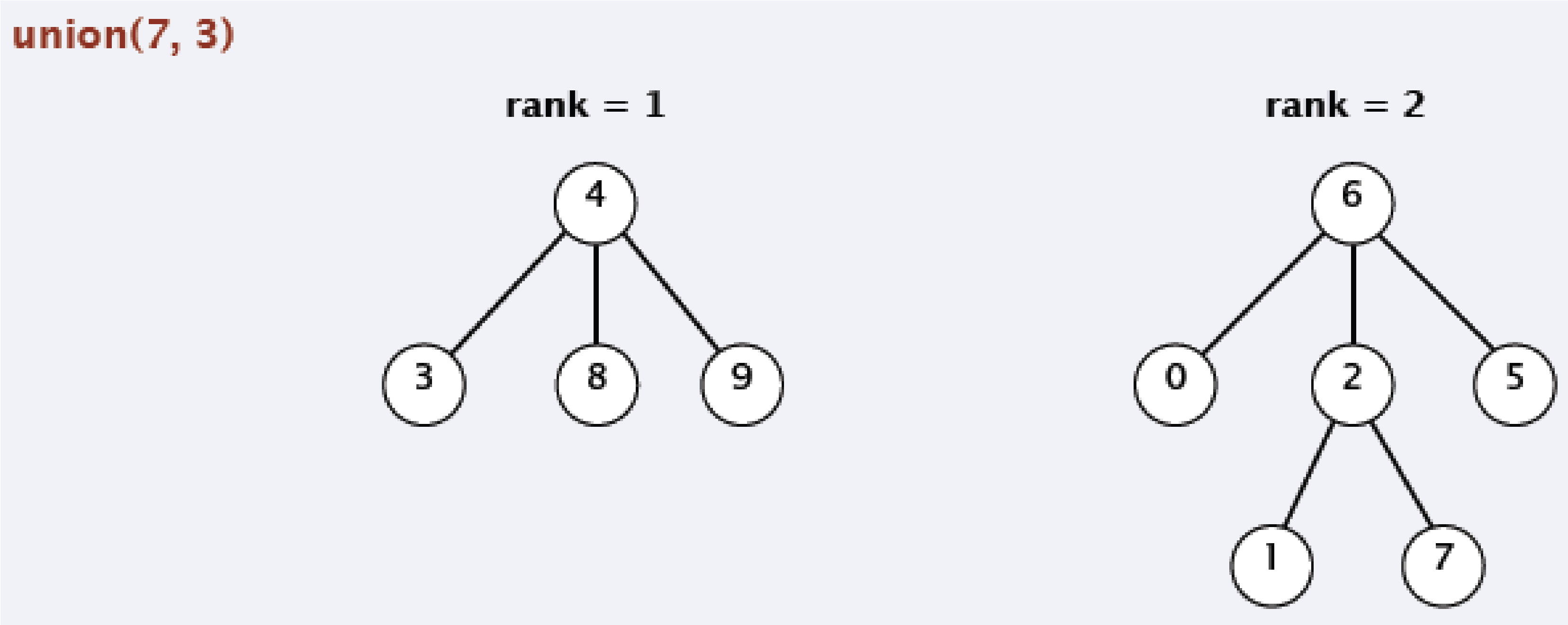
ELSE

  $parent(r) \leftarrow s.$

  $size(s) \leftarrow size(r) + size(s).$

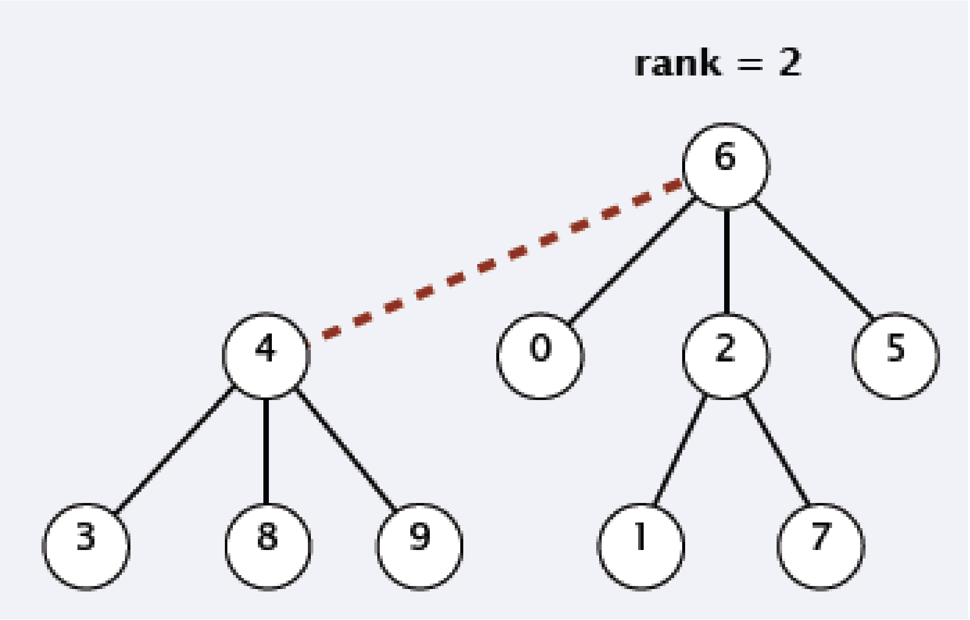
Link by Height

- Maintain an integer rank (height) for each node, initially 0.
- Link root of smaller rank (height) to root of larger rank (height); if tie, increase rank (height) of new root by 1.



Link by Height

- Maintain an integer rank (height) for each node, initially 0.
- Link root of smaller rank (height) to root of larger rank (height); if tie, increase rank (height) of new root by 1.



Link by Height

- Maintain an integer rank (height) for each node, initially 0.
- Link root of smaller rank (height) to root of larger rank (height); if tie, increase rank (height) of new root by 1.

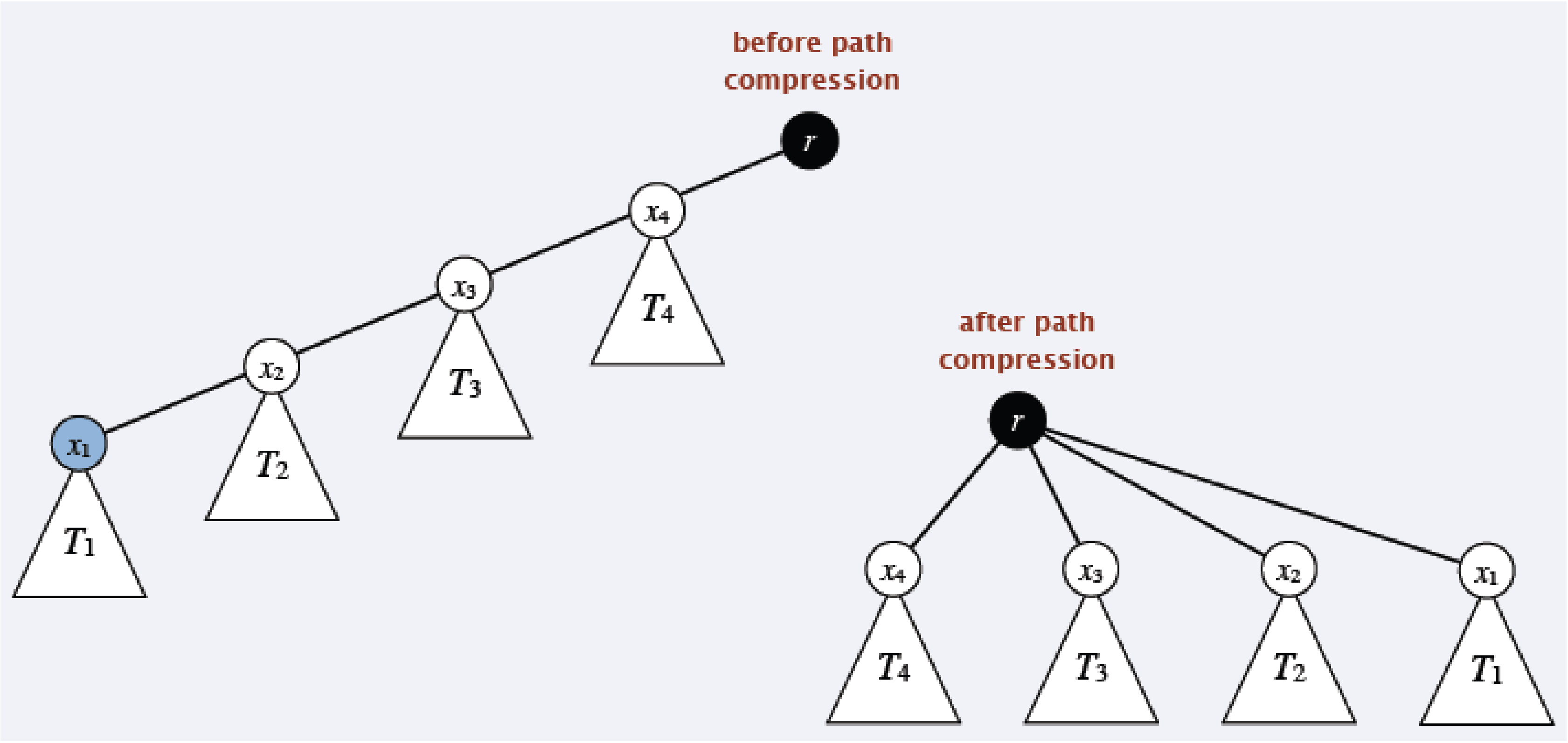
```
MAKE-SET (x)
-----------------
parent(x) ← x.
rank(x) ← 0.
```

```
FIND (x)
-----------------
WHILE x ≠ parent(x)
    x ← parent(x).
RETURN x.
```

```
UNION-BY-RANK (x, y)
-----------------
r ← FIND (x).
s ← FIND (y).
IF (r = s) RETURN.
ELSE IF rank(r) > rank(s)
    parent(s) ← r.
ELSE IF rank(r) < rank(s)
    parent(r) ← s.
ELSE
    parent(r) ← s.
    rank(s) ← rank(s) + 1.
```

Path Compression

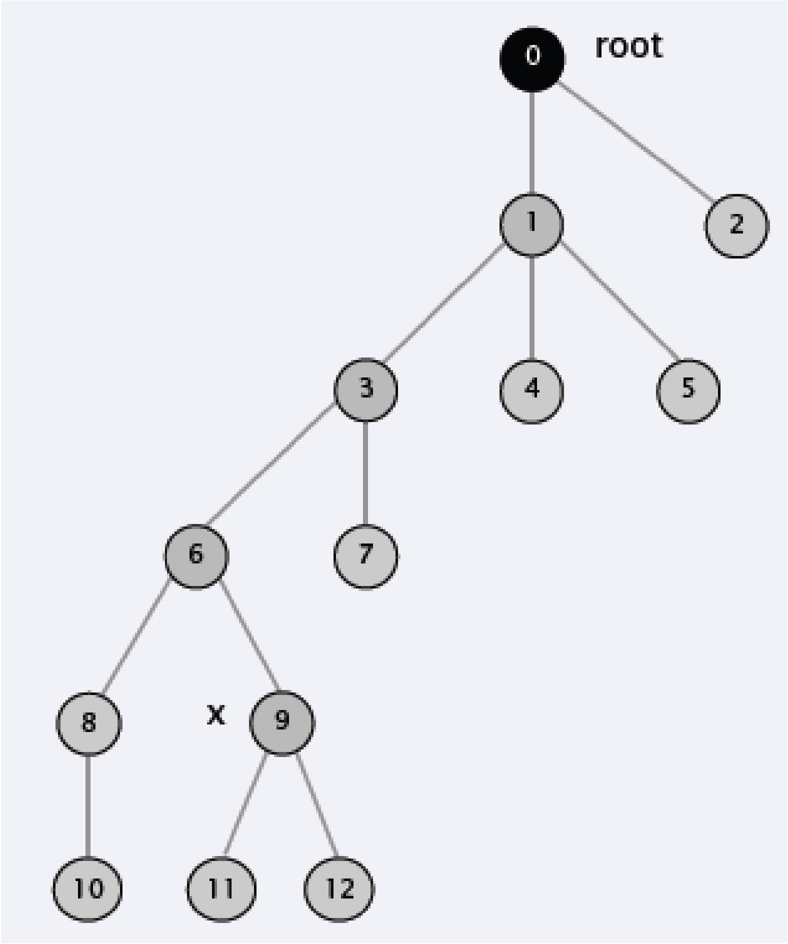
- After finding the root *r* of the tree containing *x*, change the parent pointer of all nodes along the path to point directly to *r*.



Path compression can cause a very deep tree to become very shallow

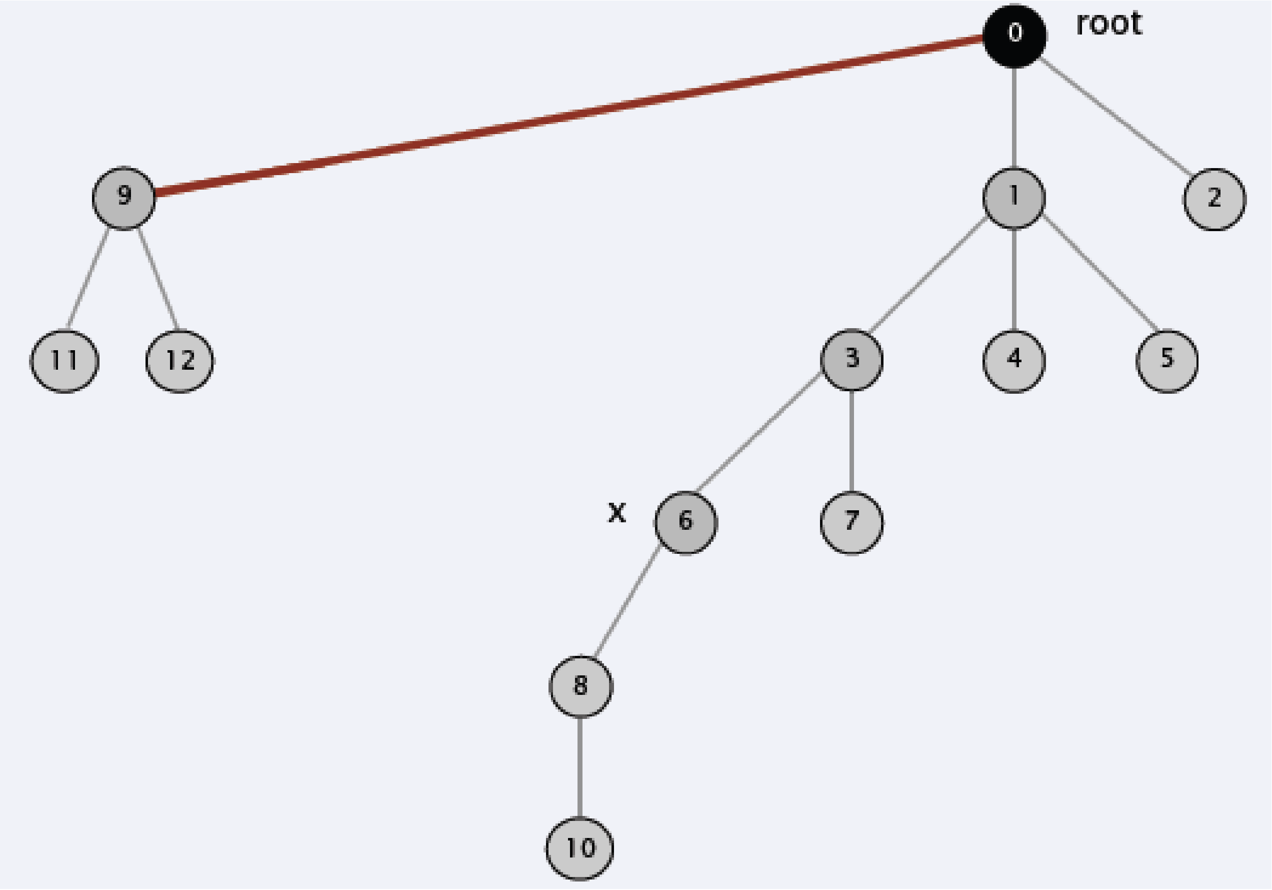
Path Compression

- After finding the root *r* of the tree containing *x*, change the parent pointer of all nodes along the path to point directly to *r*.



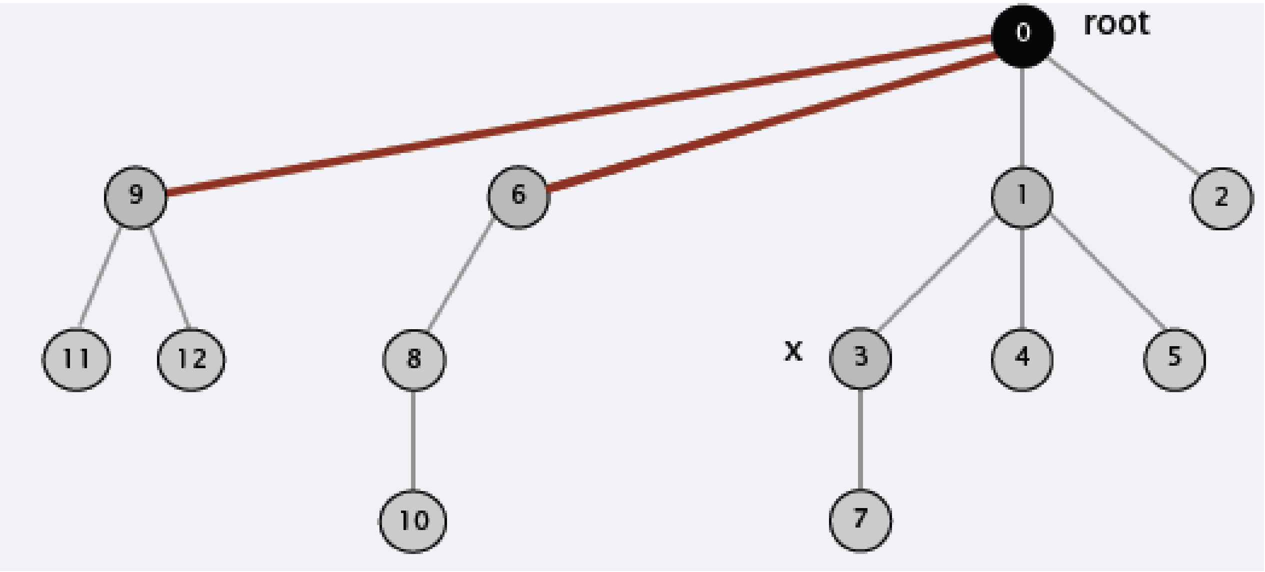
Path Compression

- After finding the root *r* of the tree containing *x*, change the parent pointer of all nodes along the path to point directly to *r*.



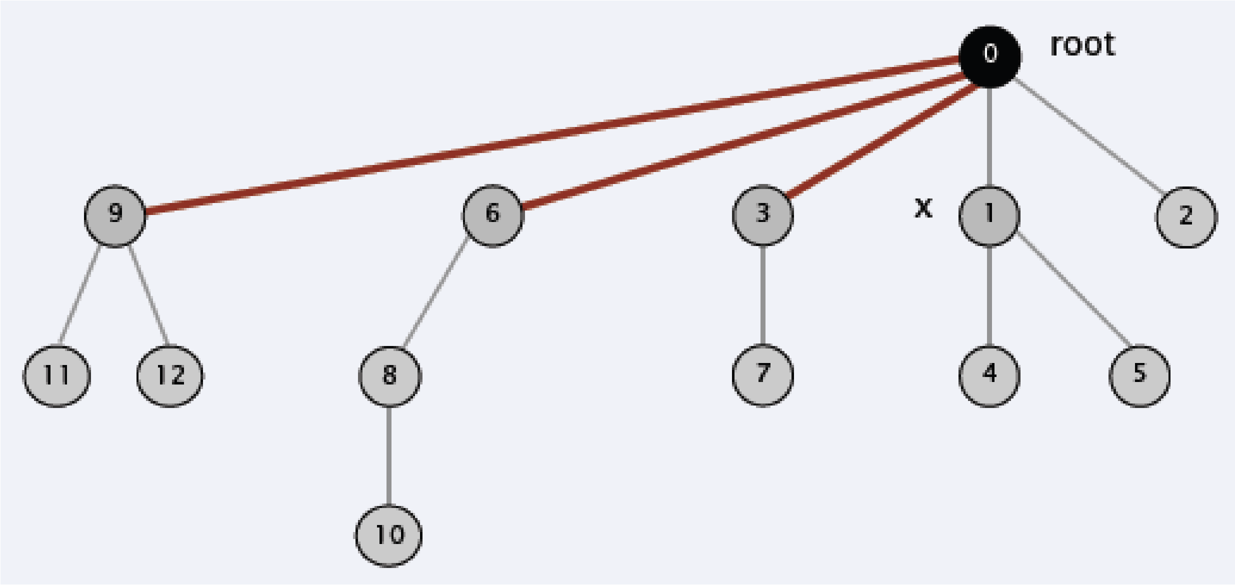
Path Compression

- After finding the root *r* of the tree containing *x*, change the parent pointer of all nodes along the path to point directly to *r*.



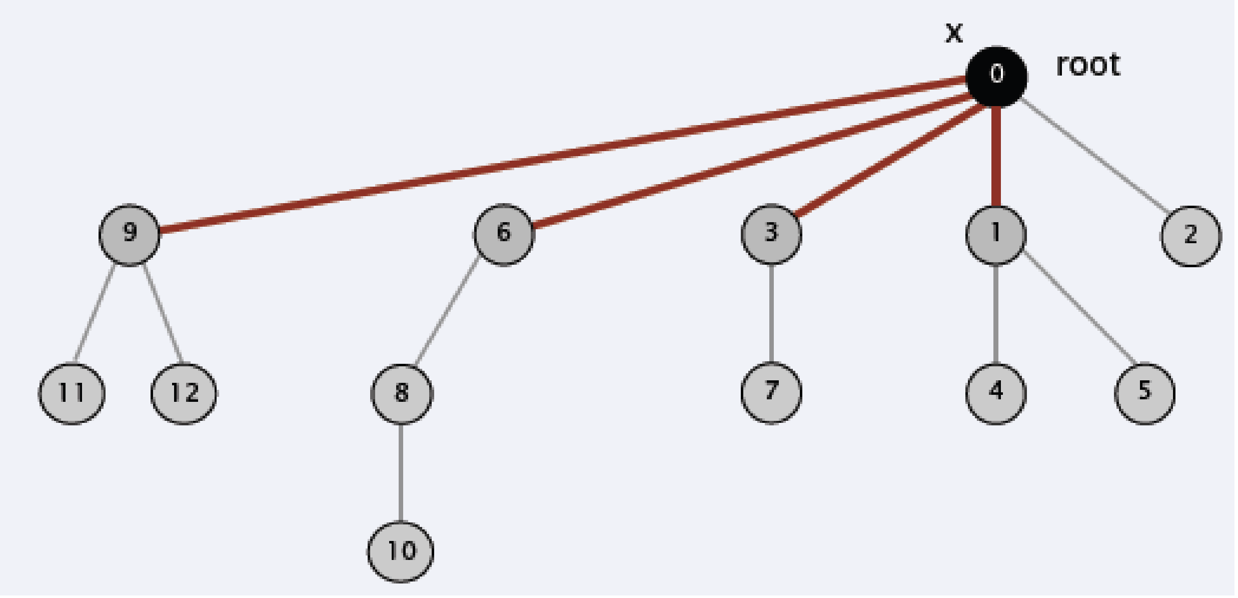
Path Compression

- After finding the root *r* of the tree containing *x*, change the parent pointer of all nodes along the path to point directly to *r*.



Path Compression

- After finding the root *r* of the tree containing *x*, change the parent pointer of all nodes along the path to point directly to *r*.



Path compression can cause a very deep tree to become very shallow

Path Compression

- After finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.

FIND (x)

IF $x \neq parent(x)$

  $parent(x) \leftarrow$ FIND $(parent(x))$.

RETURN $parent(x)$.

Note: Path compression does not change the rank of a node;
So $height(x) \leq rank(x)$ but they are not necessarily equal.

Algorithm for Disjoint-Set Forest

MAKE-SET(x)

1. $p[x] \leftarrow x$

2. $rank[x] \leftarrow 0$

UNION(x, y)

1. LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)

1. **if** $rank[x] > rank[y]$

2. **then** $p[y] \leftarrow x$

3. **else** $p[x] \leftarrow y$

4. **if** $rank[x] = rank[y]$

5. **then** $rank[y]$++

FIND-SET(x)

1. **if** $x \neq p[x]$

2. **then** $p[x] \leftarrow$ FIND-SET($p[x]$)

3. **return** $p[x]$

- Worst case running time for m MAKE-SET, UNION, FIND-SET operations is: $O(m \cdot \alpha(n))$, where $\alpha(n) \leq 4$. So nearly linear in m.
- The find operation does not change: $O(\log n)$

Exercise

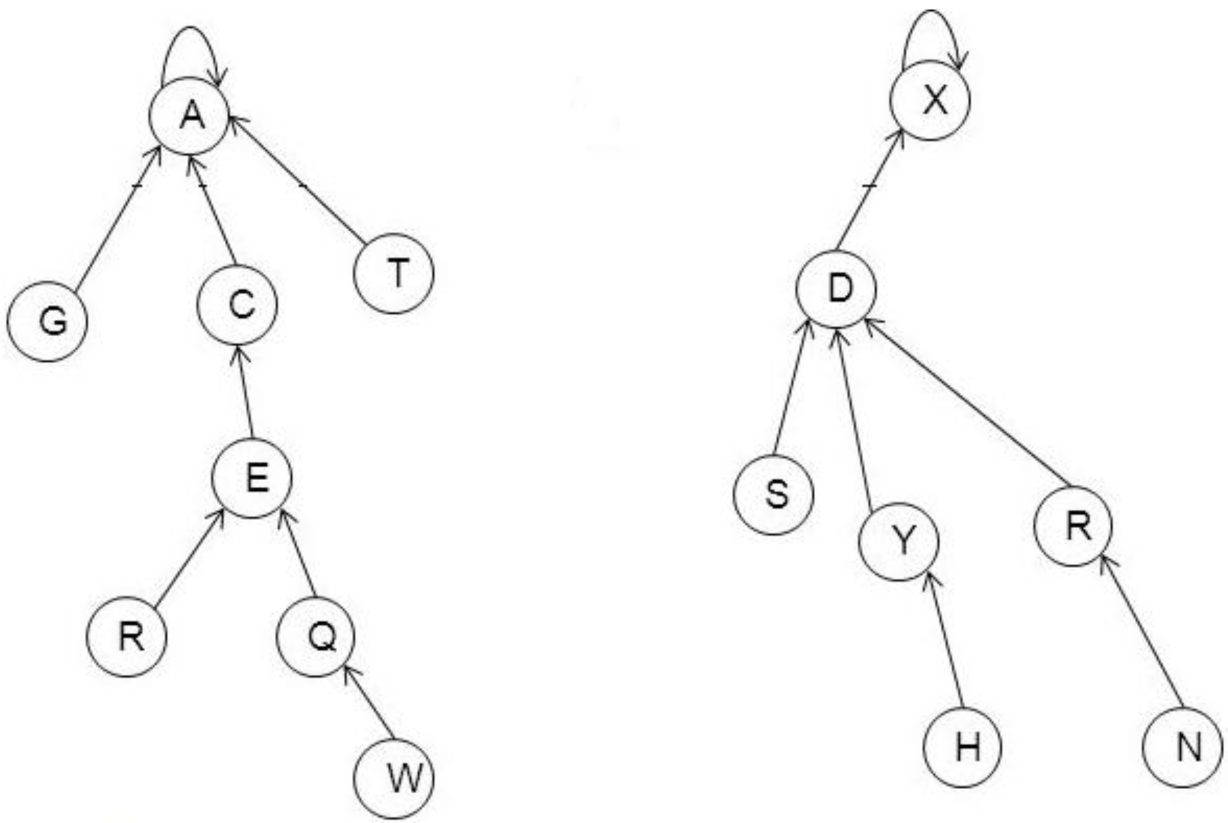
- Use the Disjoint-Sets Forest data structure with union-by-rank and path-compression to identify the connected components in the graph $G = (V, E)$, where
  - $V = \{v_1, v_2, \ldots, v_9, v_{10}\}$ and
  - $E = \{(v_1, v_2), (v_3, v_4), (v_2, v_4), (v_1, v_4), (v_3, v_2), (v_5, v_6), (v_7, v_8), (v_5, v_8), (v_4, v_7), (v_9, v_{10})\}$.

  Inspect edges in the order they appear in E in your simulation and show the state of the forest after each edge inspection.

Exercise

- What would the resultant forest be after calling UNION(*W*, *Y*) on the disjoint-sets forest of the following figure?

  You must use the *union-by-rank* and the *path-compression* heuristics.



Exercise

- Describe a data structure that supports the following operations:

  - find(x) – returns the representative of x
  - union(x, y) – unifies the groups of x and y
  - min(x) – returns the minimal element in the group of x

Solution

- We modify the disjoint set data structure so that we keep a reference to the minimal element in the group representative.

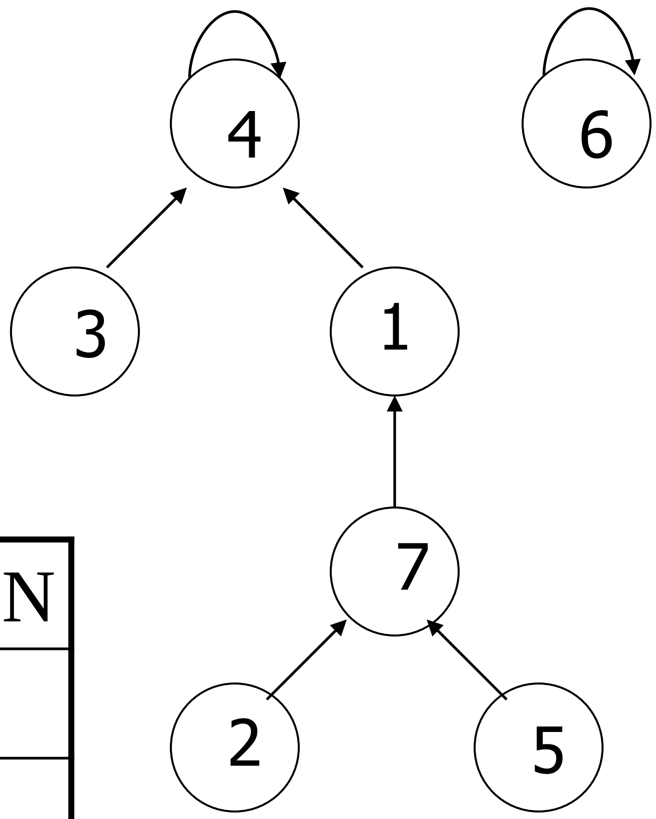
- The find operation does not change (log(n))

- The union operation is similar to the original union operation, and the minimal element is the smallest between the minimal of the two groups

Example

- Executing find(5)

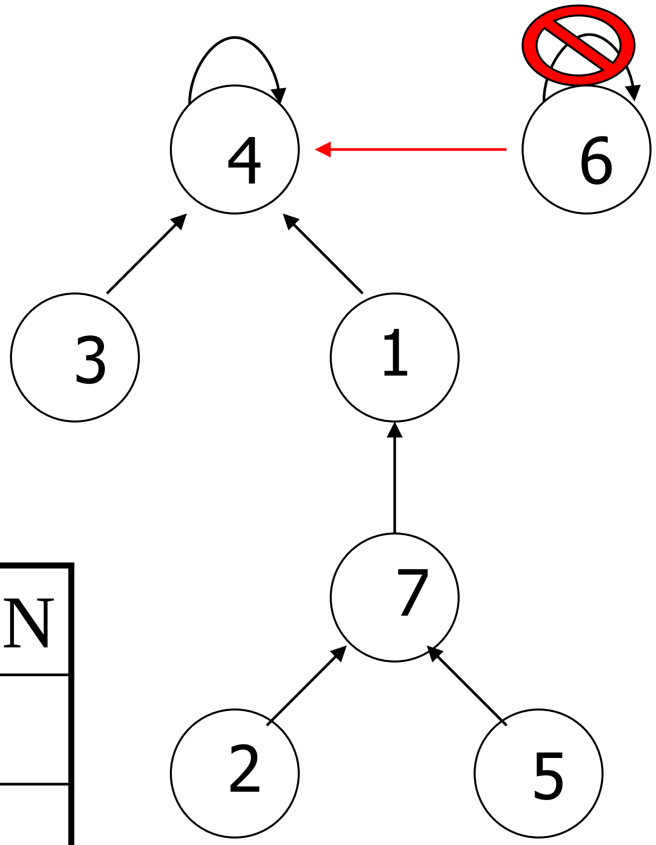
  7 → 1 → 4 → 4



	1	2	3	4	5	6	..	N
Parent	4	7	4	4	7	6		
min				1		6		

Example

- Executing union(4,6)



	1	2	3	4	5	6	..	N
Parent	4	7	4	4	7	4		
min				1		1		