Data structure for Disjoint sets

Fatemeh Panahi
Department of Computer Science
University of Toronto
CSC263-Fall 2017
Lecture 10

Today

- Disjoint set Operations
  - Make-Set
  - Find-Set
  - Union
- Implementation
  - Linked List
    - ✓ Union by weight
  - Disjoint Forest
    - ✓ Union by rank
    - ✓ Path compression
    - ✓ Union by rank and path compression

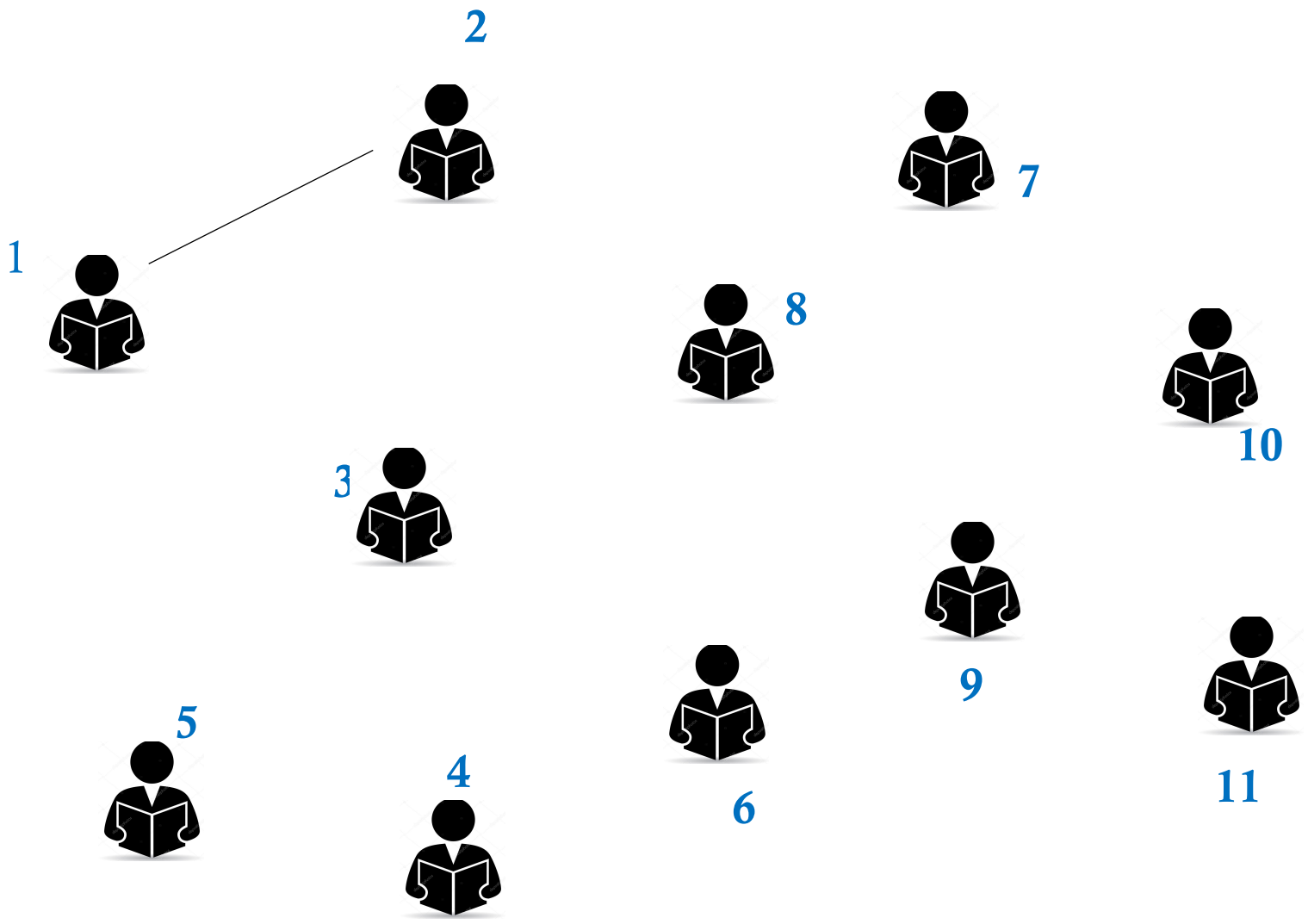
Announcements

- $A3$ solution are out, $A4$ handout will be released at the weekend
- Tutorial on Friday 2 problems on Disjoint sets.

Reading Assignments

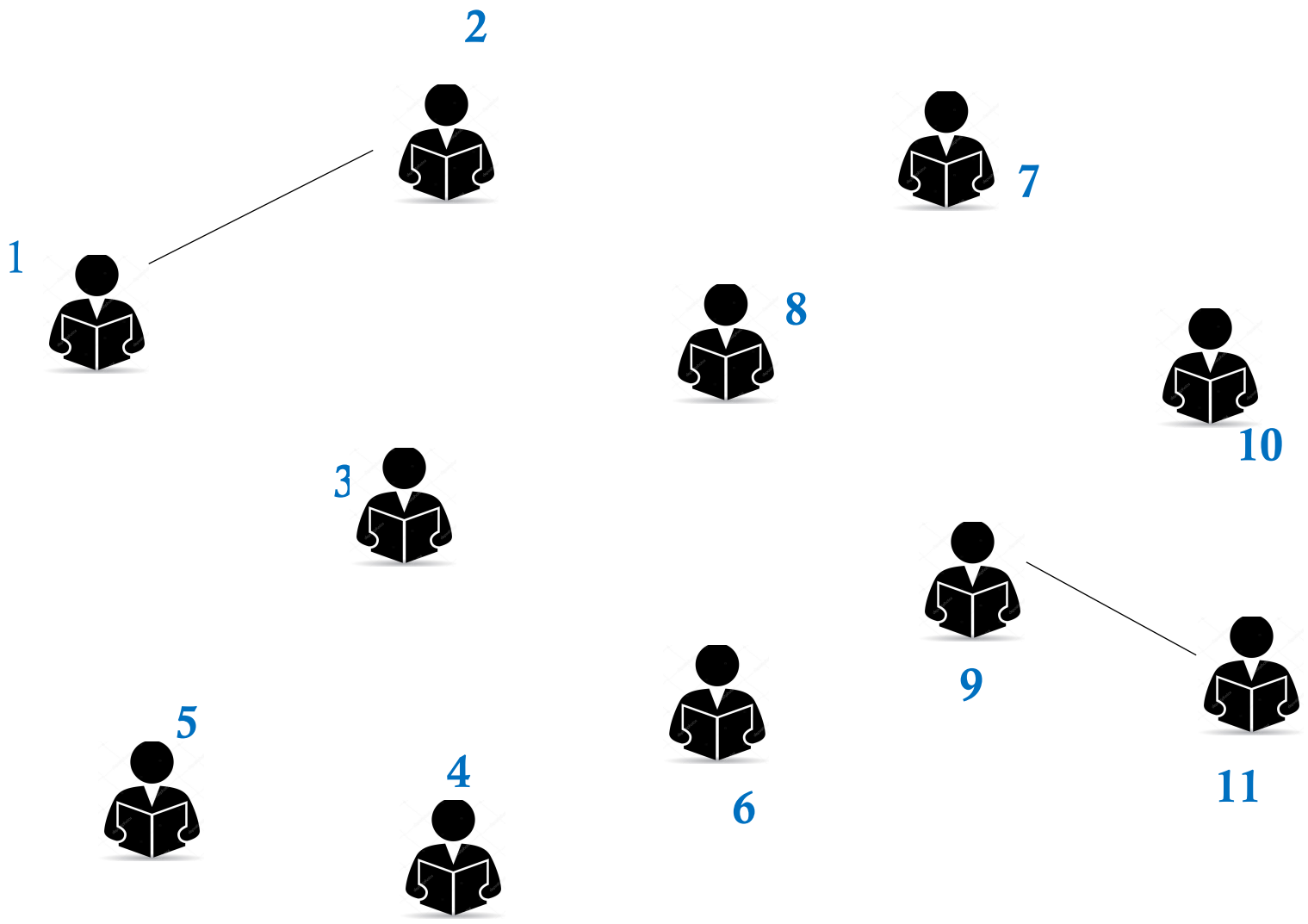
21.1, 21.2, 21.3

Real world example: Merging Communities



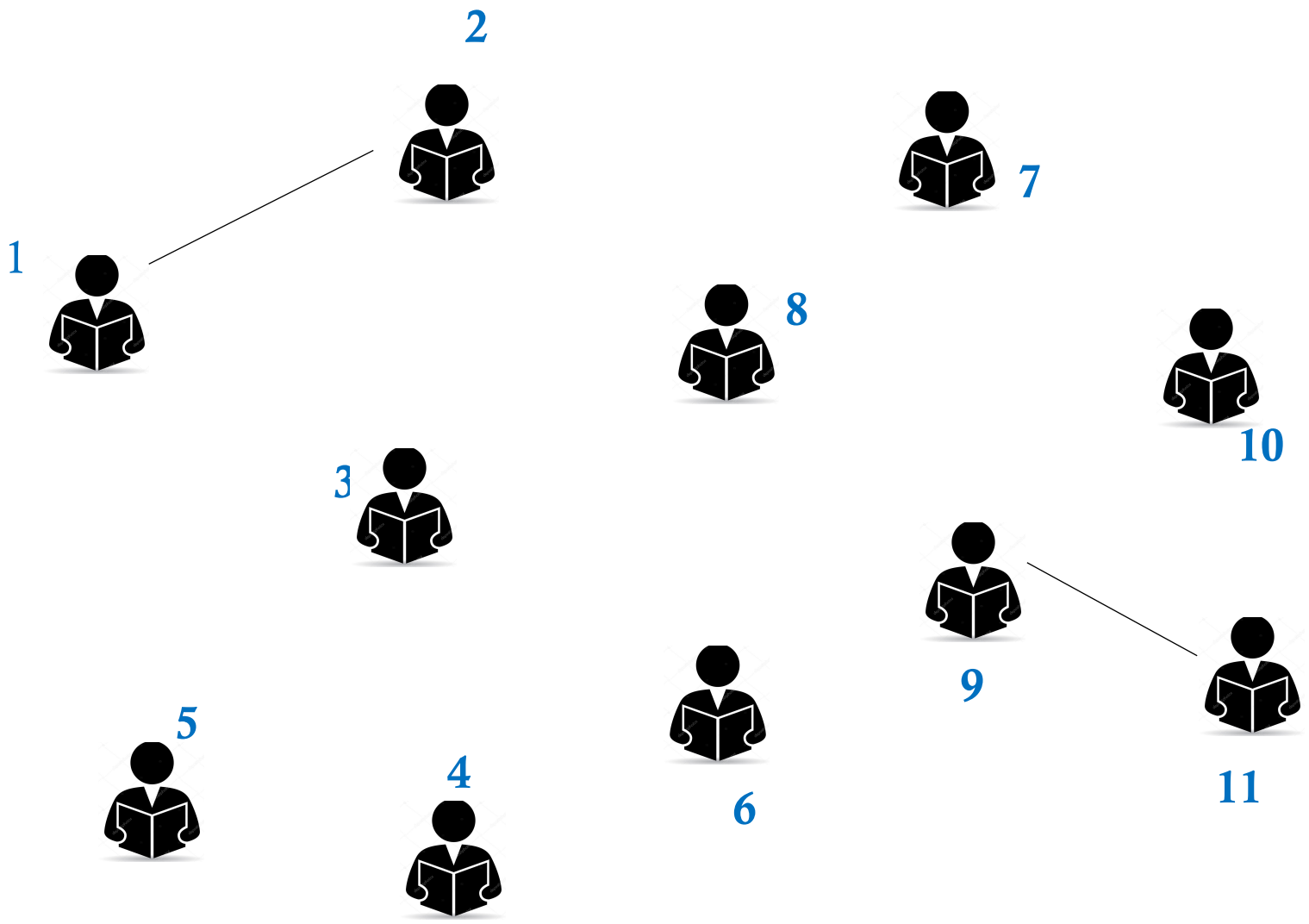
- {1}
- {2}
- {3}
- {4}
- {5}
- {6}
- {7}
- {8}
- {9}
- {10}
- {11}

Real world example: Merging Communities



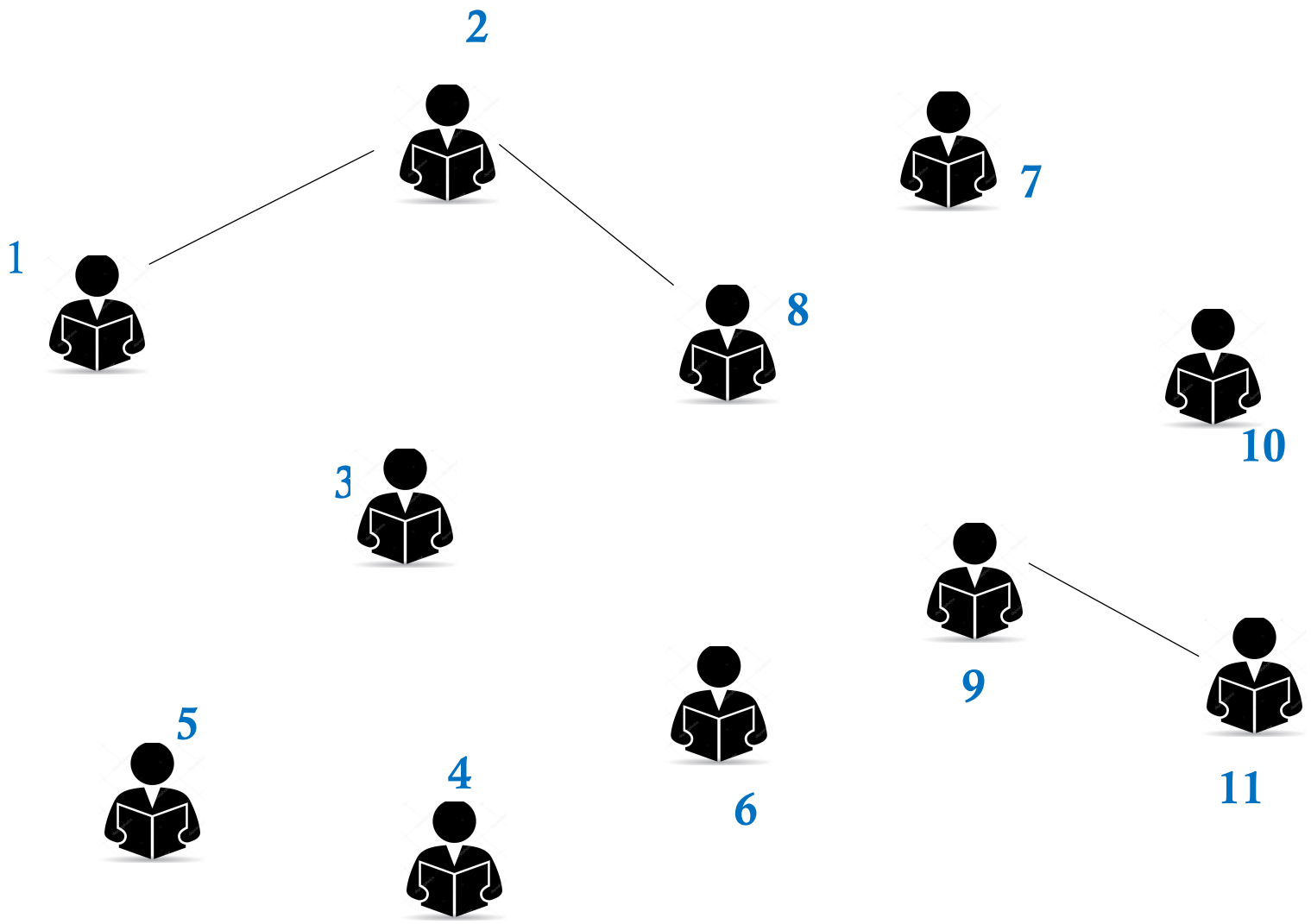
- {1, 2}
- {3}
- {4}
- {5}
- {6}
- {7}
- {8}
- {9}
- {10}
- {11}

Real world example: Merging Communities



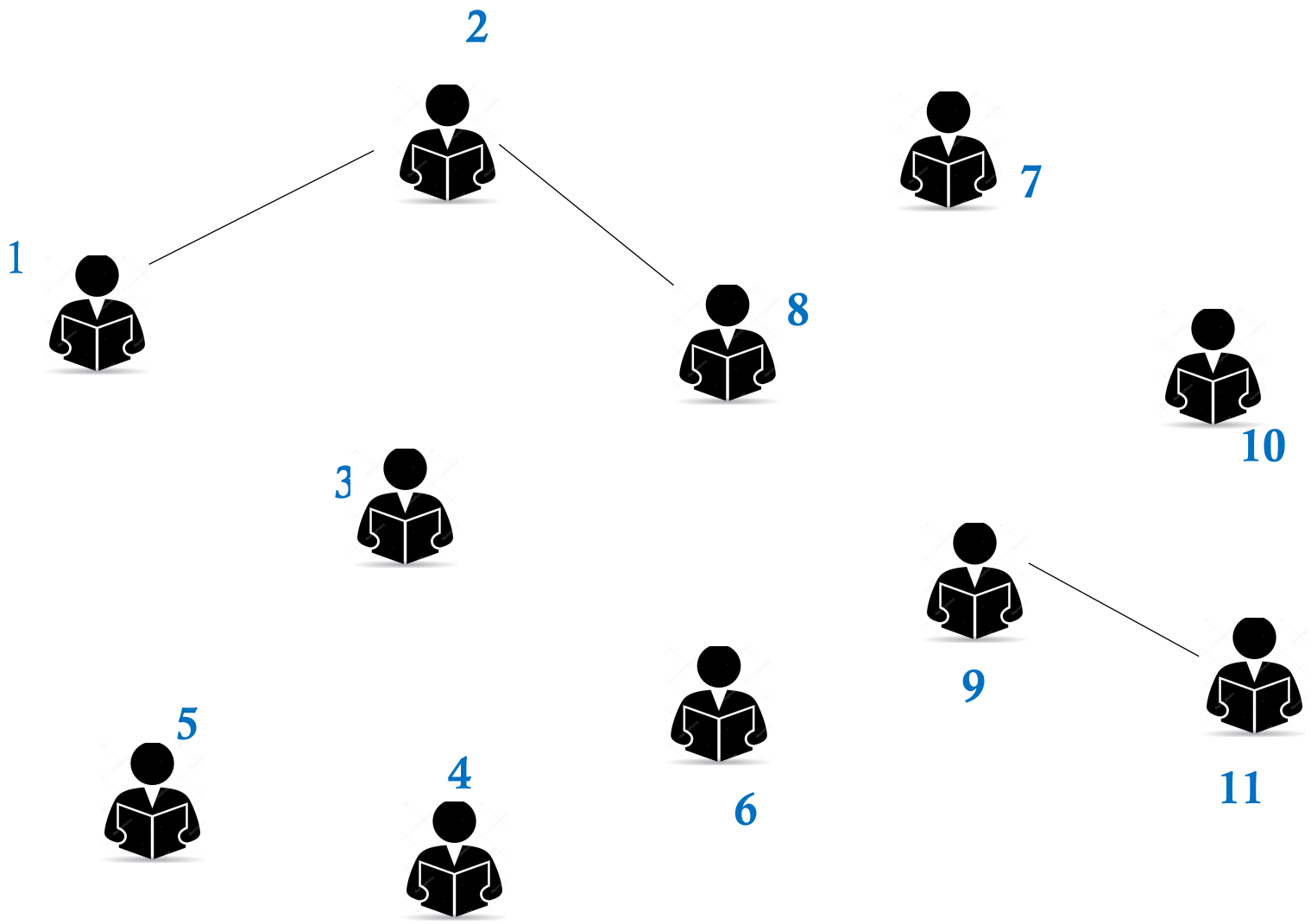
- {1, 2}
- {3}
- {4}
- {5}
- {6}
- {7}
- {8}
- {9, 11}
- {10}

Real world example: Merging Communities



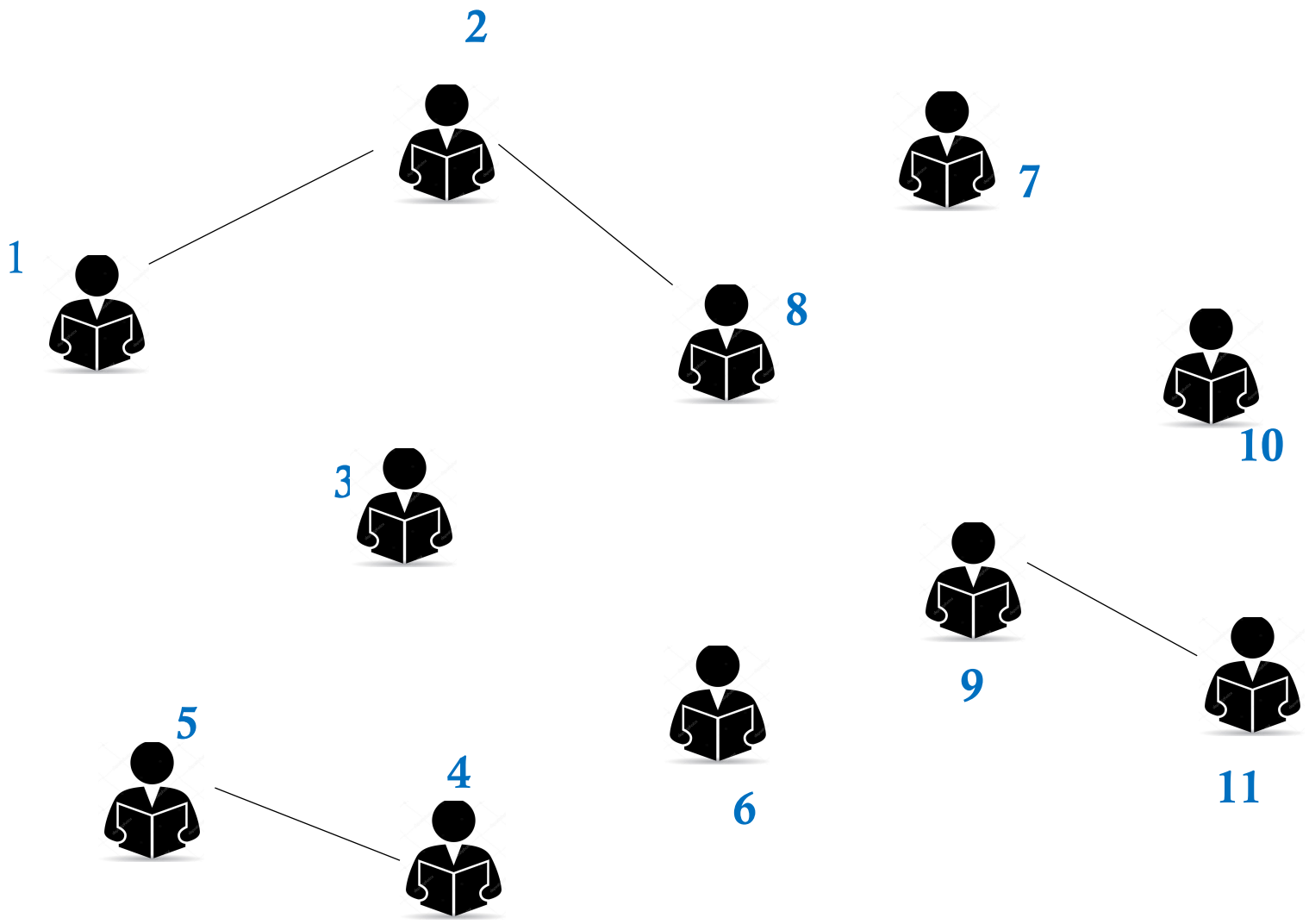
- {1, 2}
- {3}
- {4}
- {5}
- {6}
- {7}
- {8}
- {9, 11}
- {10}

Real world example: Merging Communities



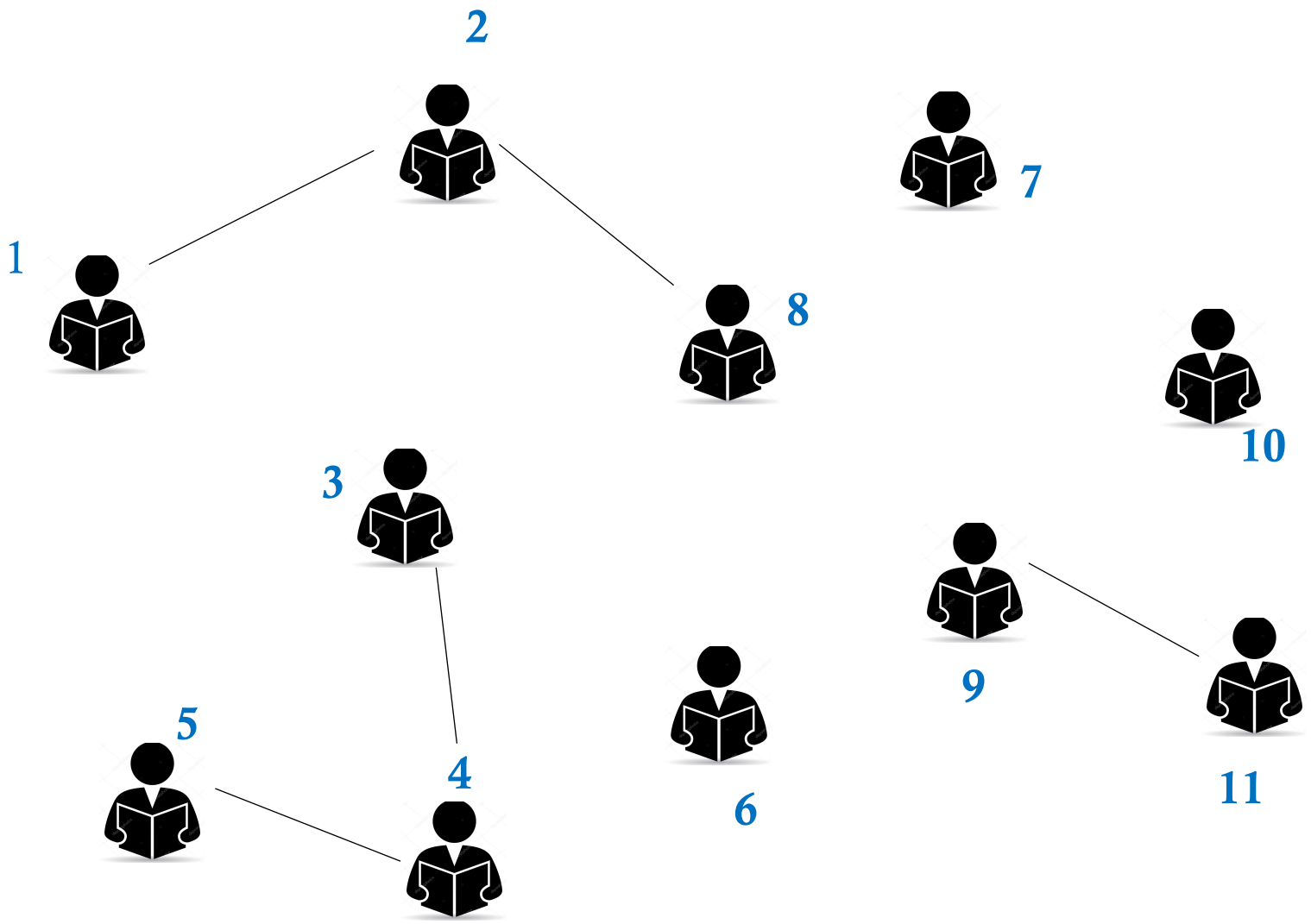
- {1, 2, 8}
- {3}
- {4}
- {5}
- {6}
- {7}
- {9, 11}
- {10}

Real world example: Merging Communities



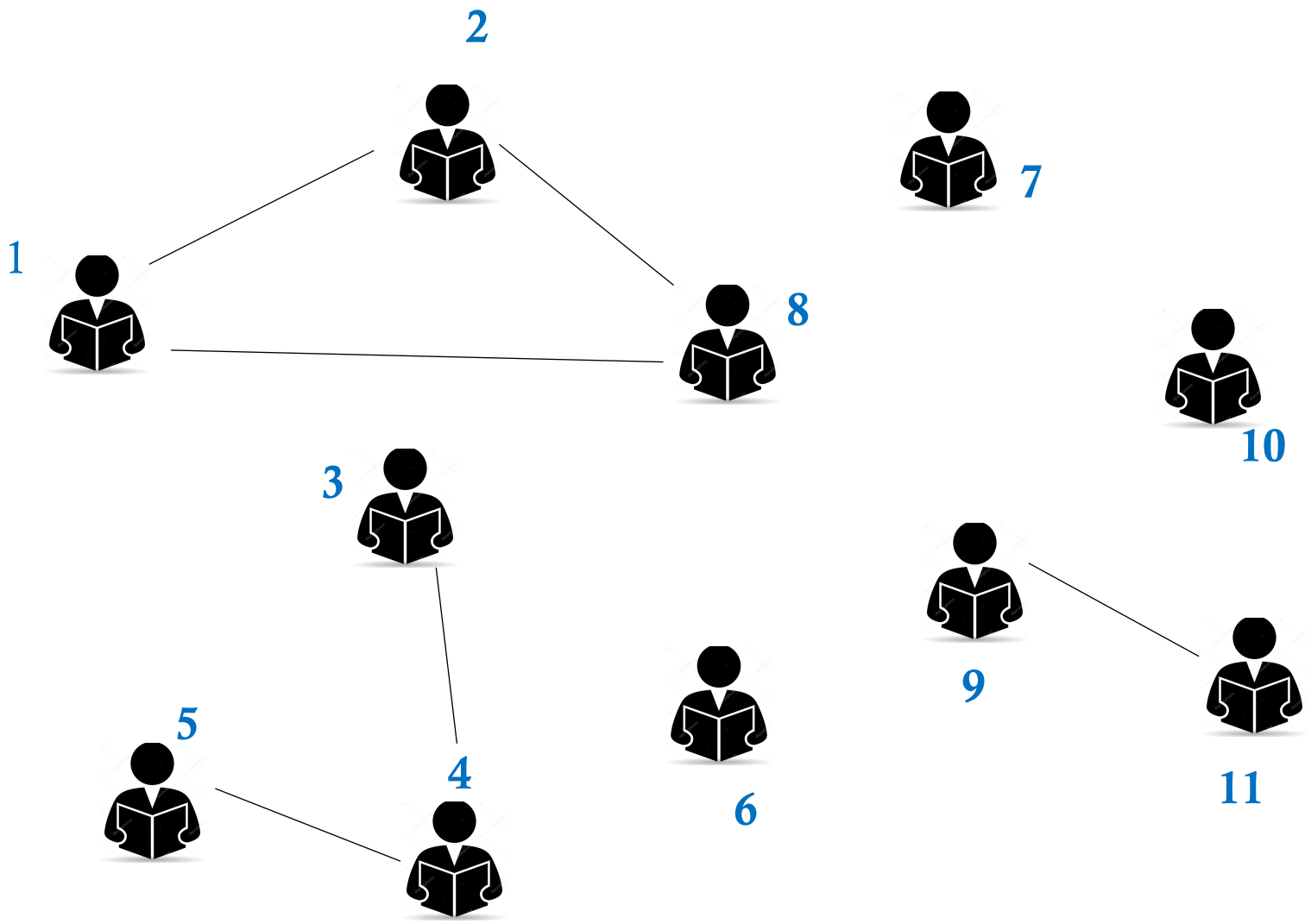
- {1, 2, 8}
- {3}
- {4}
- {5}
- {6}
- {7}
- {9, 11}
- {10}

Real world example: Merging Communities



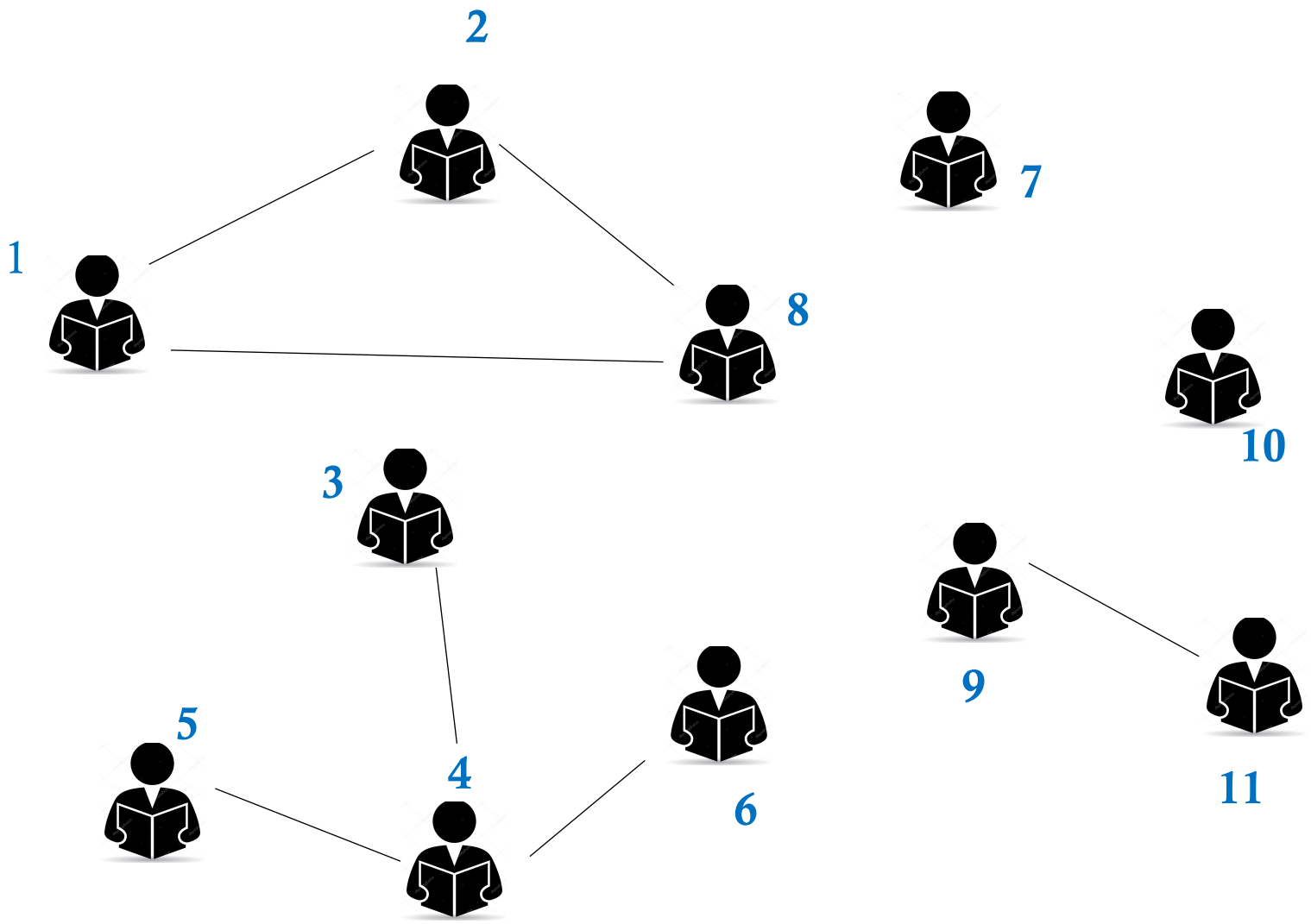
- {1, 2, 8}
- {3}
- {4, 5}
- {6}
- {7}
- {9, 11}
- {10}

Real world example: Merging Communities



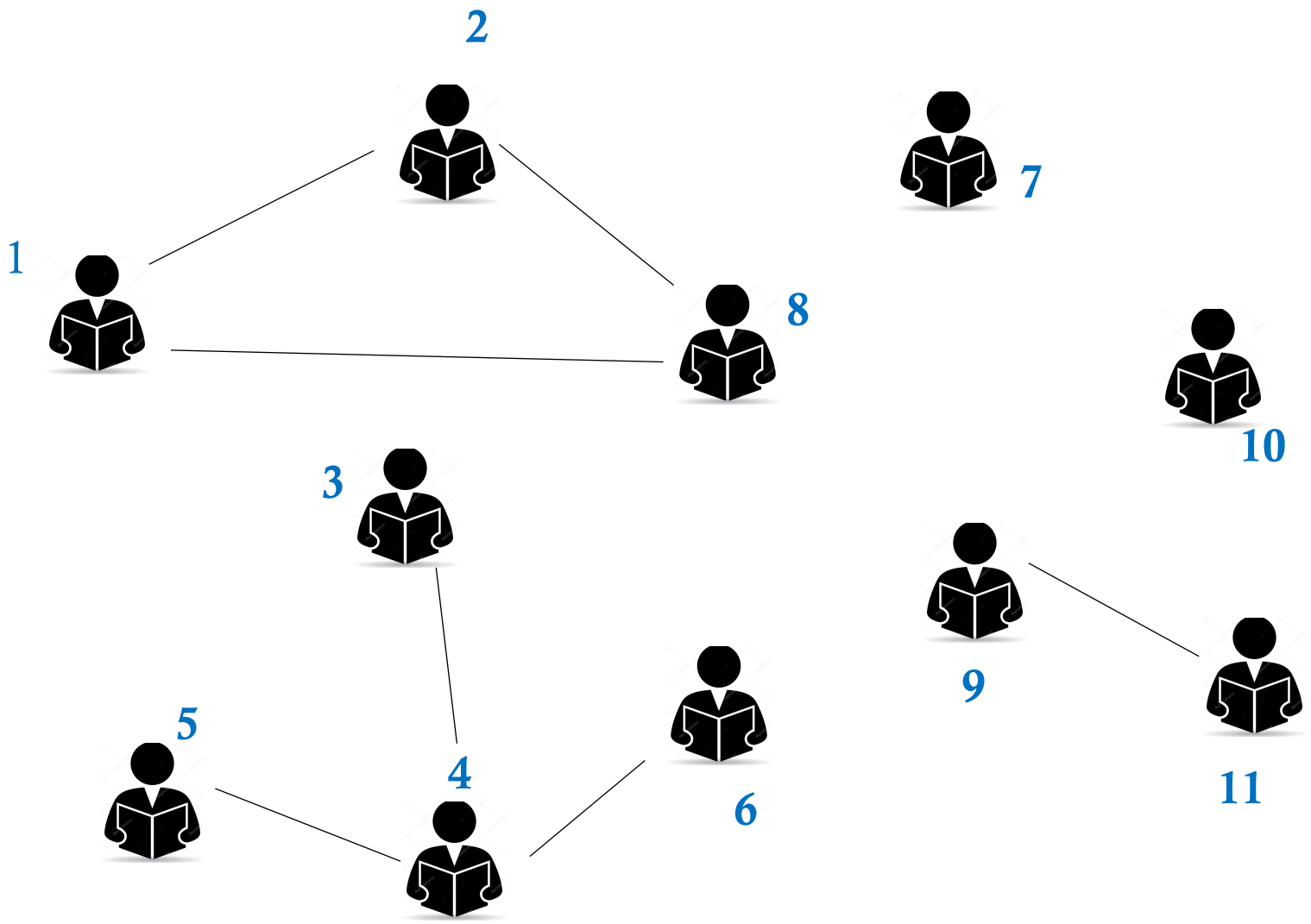
- {1, 2, 8}
- {4, 5, 3}
- {6}
- {7}
- {9, 11}
- {10}

Real world example: Merging Communities



- {1, 2, 8}
- {4, 5, 3}
- {6}
- {7}
- {9, 11}
- {10}

Real world example: Merging Communities



- {1, 2, 8}
- {4, 5, 3, 6}
- {7}
- {9, 11}
- {10}

Today

- Disjoint Set Operations
  - Make-Set
  - Union
  - Find-Set

- Data dtructures:
  - Linked list
    - Weighted Union
  - Disjoint set forest
    - Union by rank
    - Path Compression

Disjoint Set Operations:

We have **a collection of disjoint sets of elements**. Each set is identified by a representative element.

- Make-Set(x): Create new set $\{x\}$ with representative x.
- Union(x, y): x and y are elements of two sets. Remove these sets and add their union. Choose a representative for it.
- Find-Set(x): return the representative of the set containing x.

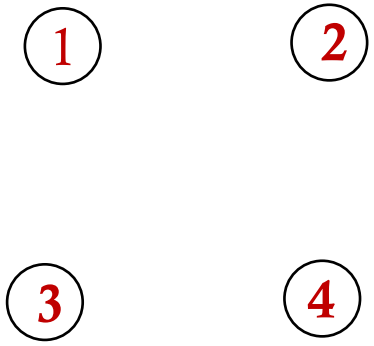
Disjoint Set Operations- Example:

Make-Set(1) {1}

Make-Set(2) {2}

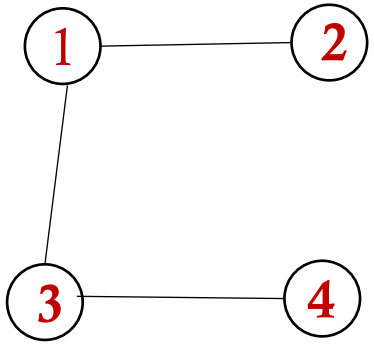
Make-Set(3) {3}

Make-Set(4) {4}



Disjoint Set Operations- Example:

Make-Set(1)	{1}	Find-Set(3)	(returns 3)	
Make-Set(2)	{2}	Find-Set(2)	(returns 2)	
Make-Set(3)	{3}	UNION(1,2)	(representative 1)	{1,2}
Make-Set(4)	{4}	Find-Set(2)	(returns 1)	
		Find-Set(1)	(returns 1)	
		UNION(3,4)	(representative 4)	{3,4}
		Find-Set(4)	(returns 4)	
		Find-Set(3)	(returns 4)	
		UNION(1,3)	(representative 4)	{1,2,3,4}



Finding the connected components:

```
CONNECTED-COMPONENTS(G)
    for each vertex v ∈ G.V
        MAKE-SET(v)
    for each edge (u, v) ∈ G.E
        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
```

From the textbook page 563

Time complexity:

We consider the amortized cost of the operations:

- m operations in total:

  n make-Set, $m - n$ Find-Set and Union

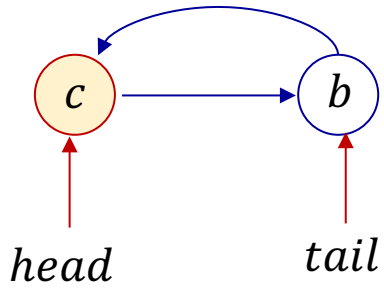
- What is the maximum number of Union operations?

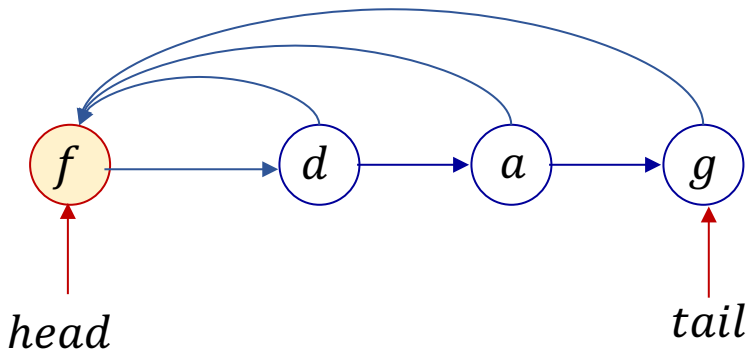
n-1

Implementation:

Linked List

Linked List Implementation:





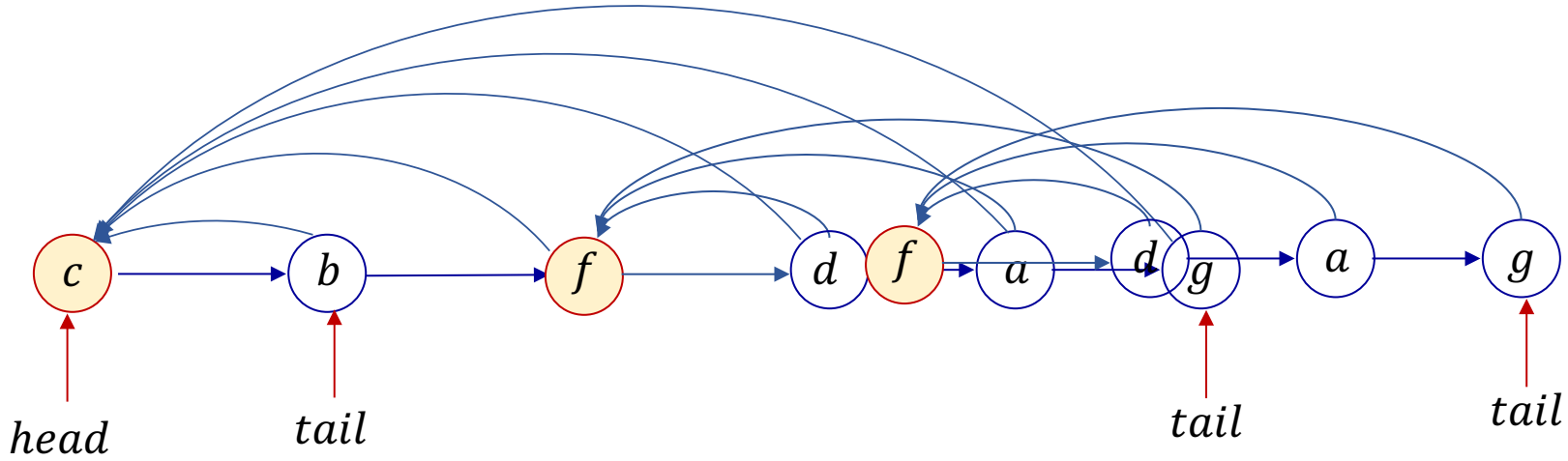
Make-Set

Find-Set: Return the representative

Union: Append one list to the other, then update the pointers to head

Union(a, b): Copy elements of A into B, $O(A.length)$ where $a \in A$, b $\in B$

Linked List Implementation:



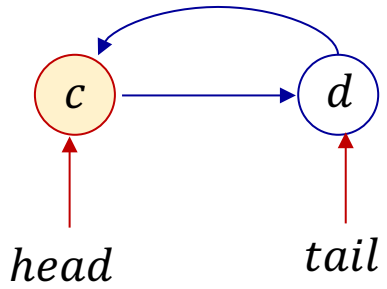
Make-Set

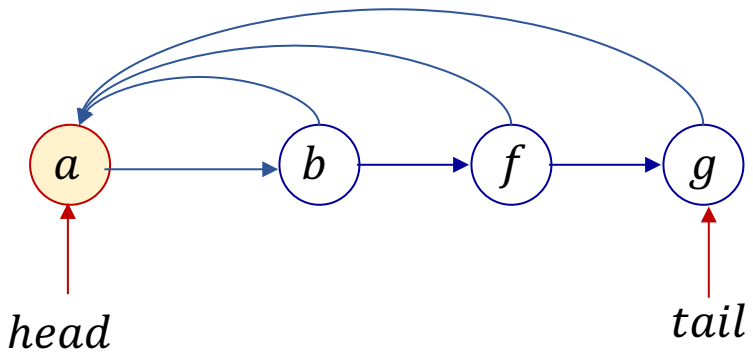
Find-Set: Return the representative

Union: Append one list to the other, then update the pointers to head

Union(a, b): Copy elements of A into B, $O(A.length)$ where $a \in A$, b $\in B$

Linked List Implementation:



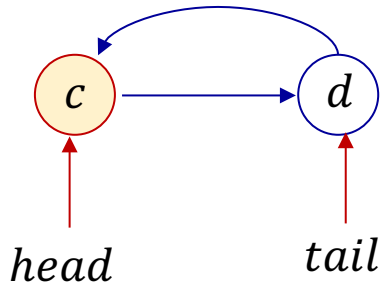


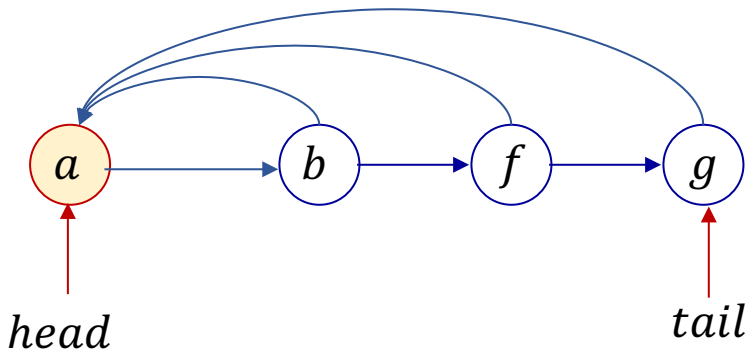
Make-Set: $\boldsymbol{O(1)}$

Find-Set: ?

Union$(\boldsymbol{a}, \boldsymbol{b})$**:** ?

Linked List Implementation:





Make-Set: $\boldsymbol{O(1)}$

Find-Set: $\boldsymbol{O(1)}$

Union$(\boldsymbol{a}, \boldsymbol{b})$**:** In the worst case $n - 1$ union operation:

$$1 + 2 + 3 + 4 + \ \dots \ + \ n - 1 \ = \ O(n^2)$$

Weighted-union heuristic

Suppose that each list stores the length of the list.
Always append the **shorter list to larger list.**

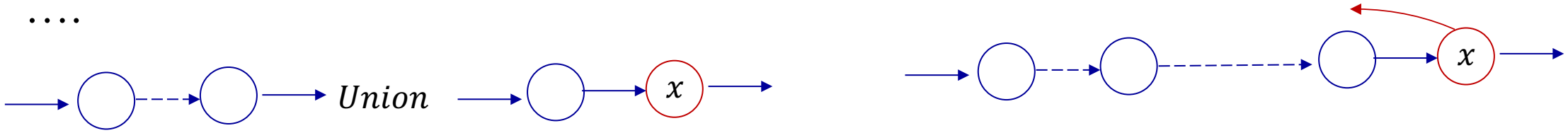
Running time: m operations in total

$n - 1$ Union operations

The **first** time an object's pointer is updated the list has at least 2 objects.

The **second** time the object's pointer is updated the list has at least 4 objects.

....



Question: How many times an object's pointer is updated after m operations?

Weighted-union heuristic

Running time: m operations in total

$n - 1$ Union operations

The **first** time an object's pointer is updated the list has at least 2 objects.

The **second** time the object's pointer is updated the list has at least 4 objects.

….

The ith time the object's pointer is updated the list has at least 2^i objects.

if the length of the list after a Union operation is k, each object's pointer has been updated at most $\lceil \log k \rceil$ times.

Total time for $n - 1$ union operations: $O(n \log n)$

Total cost of m operation: $O(m + n \log n)$

Implementation:

Disjoint Forest

Disjoint Forest:

Each set is an "inverted" tree.

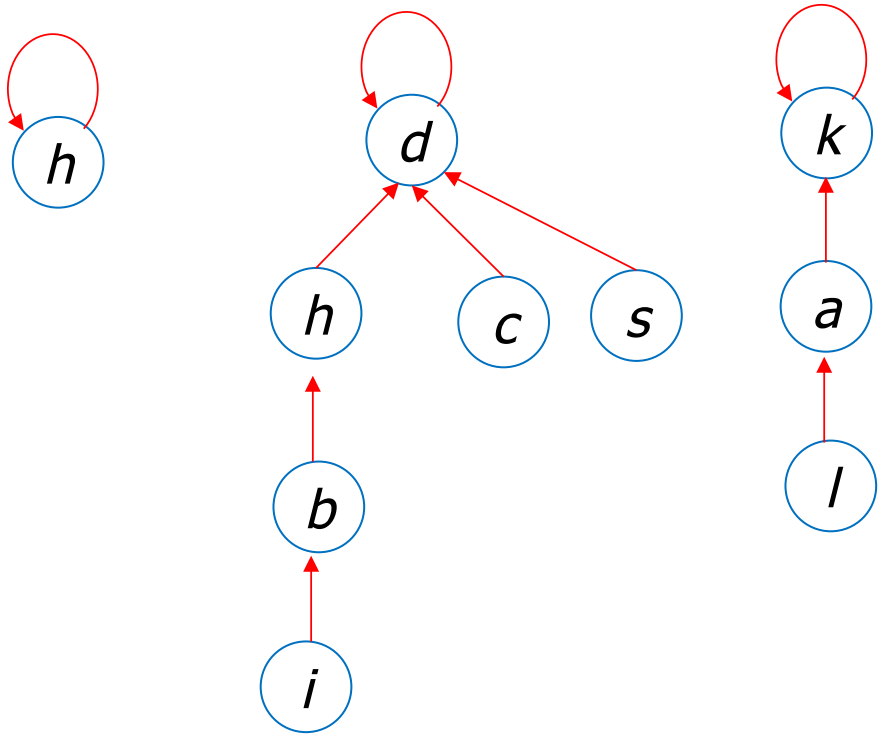
- Each element keeps a pointer to its parent in the tree
- The root points to itself (test root by x.p = x)
- The representative is the root
- NOT necessarily a binary or balanced tree

Running time:

Make-Set ?

Find-Set ?

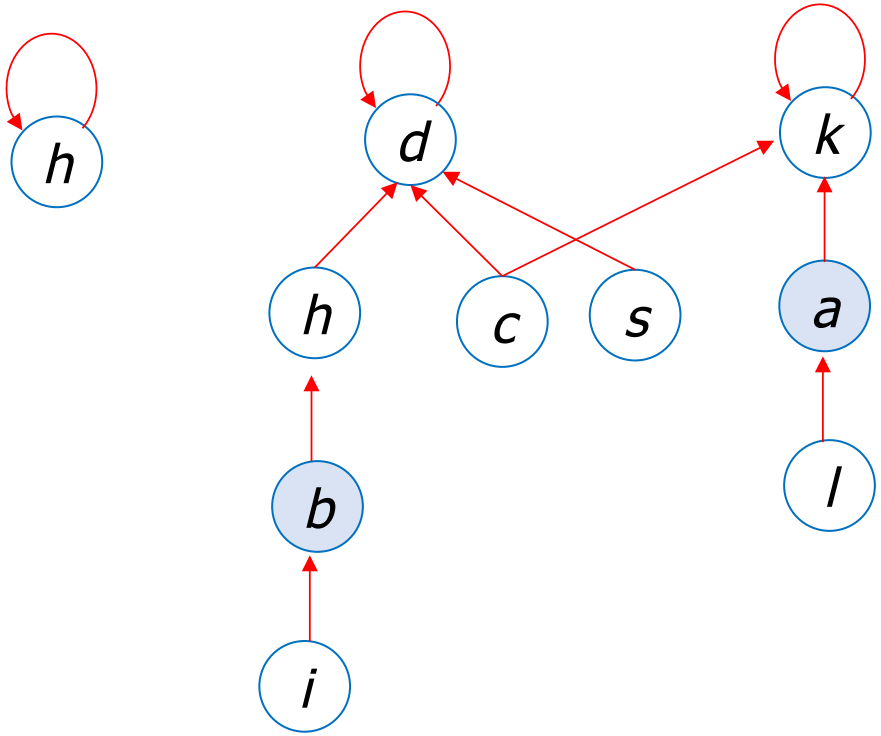
Union(a, b)?



Disjoint Forest:

Union(a, b):

1. Call Find-Set(x) and Find-Set(y) to locate the representatives, $O(h)$
2. Let one tree's root point to the other tree's root, $O(1)$

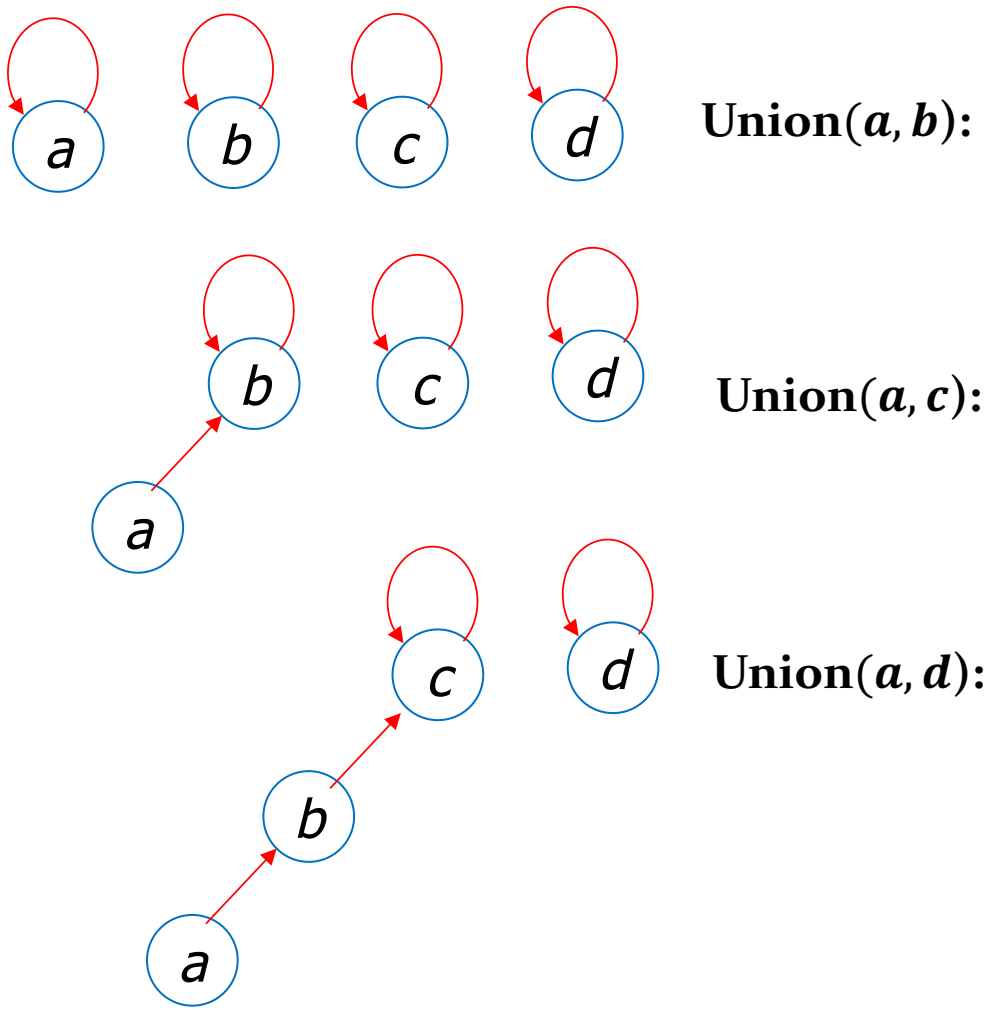


Disjoint Forest:

runtime

The worst-case sequence of m operations.
$m/2$ MakeSets,
$m/2 - 1$ Union,

Total cost in worst-case sequence: $\Theta(m^2)$



Could we have been smarter about this?

Disjoint Forest:

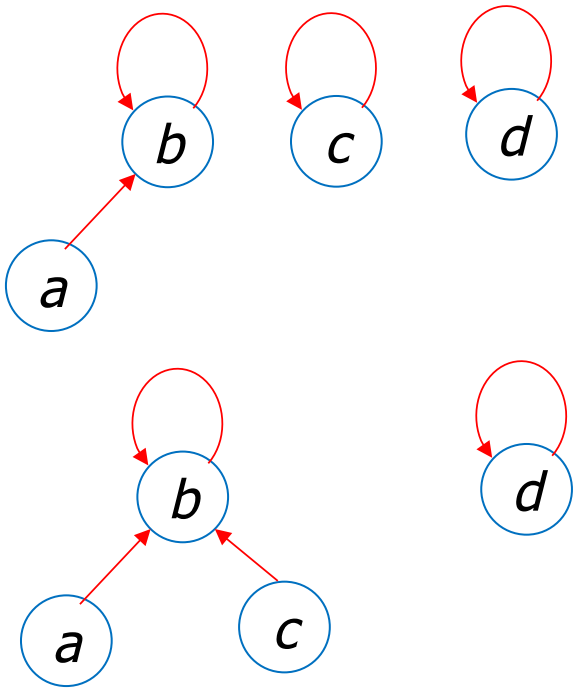
1- Union by rank

First heuristics: Union by Rank

FindSet takes O(h), so the height of tree matters

To keep the unioned tree's height small, we should let the taller tree's root be the root of the unioned tree.

So, we need a way to keep track of the height of the tree.



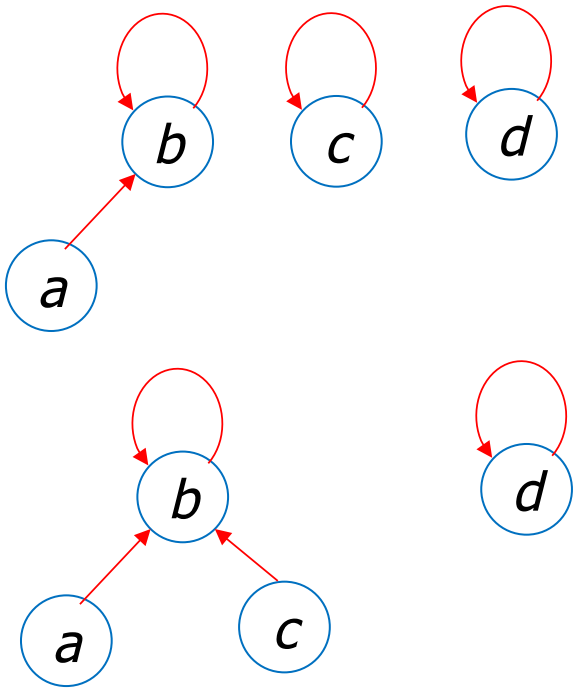
Union(a, c):

First heuristics: Union by Rank

FindSet takes O(h), so the height of tree matters

During Union, we make the node with smaller rank point to the node with larger rank.

So, we need a way to keep track of the height of the tree.



Union(a, c):

First heuristics: Union by Rank

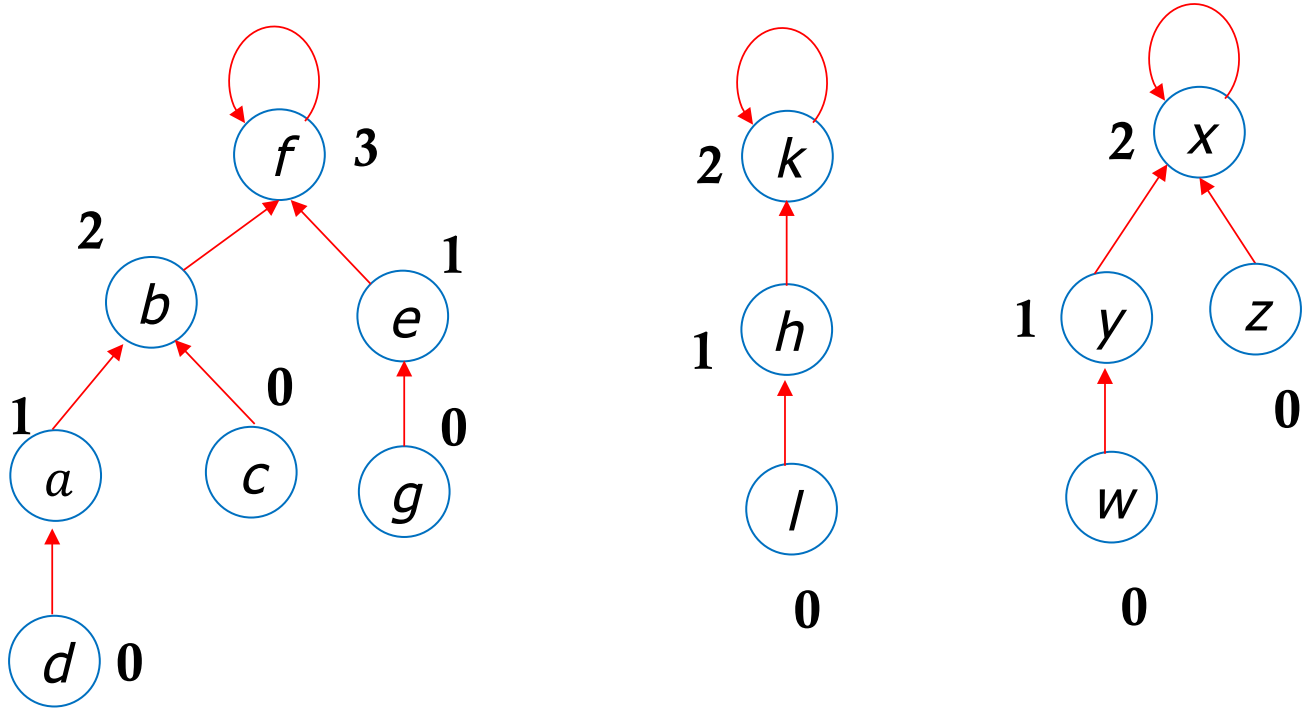
Each node keeps a values rank.

Rank: an upper bound for the height of the node.
For now, a node's rank is the same as its height, but it will be different later.

When Union, let the root with lower rank point to the root with higher rank.

If the two roots have the same rank, choose either root as the new root and increment its rank.

Otherwise, the ranks do not change



Upper bound on rank

Rank of a tree with n nodes is at most $\log n$, i.e., $r(n) \leq \log n$

Proof:

Equivalently, prove $n(r) \geq 2^r$.

Use **induction** on r

Base step: if $r = 0$ (single node), $n(0) = 1$, TRUE

Inductive step: assume $n(r) \geq 2^r$ a tree with root rank $r + 1$ is a result of unioning two trees with root rank r, so

$$n(r+1) = n(r) + n(r) \geq 2 \times 2^r = 2^{r+1}$$

Make-Set pseudocode

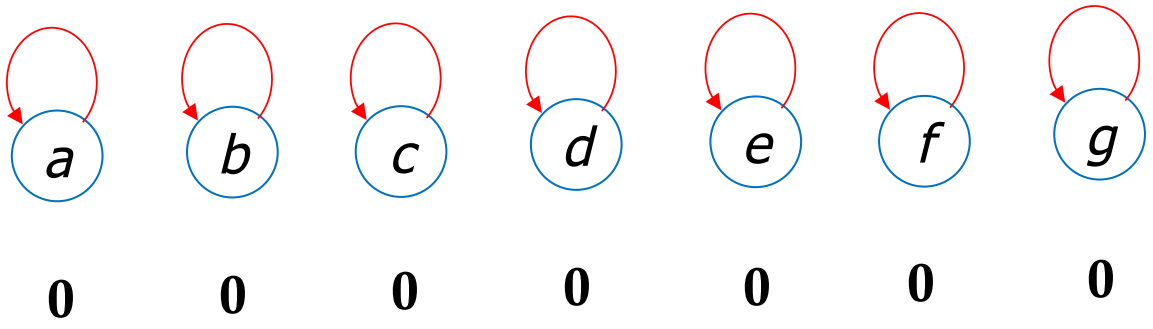
Make-Set $(\boldsymbol{x})$

if x is not already present:

add x to the disjoint-set tree

$x.parent \longleftarrow x$

$x.rank \longleftarrow 0$



Find-Set pseudocode

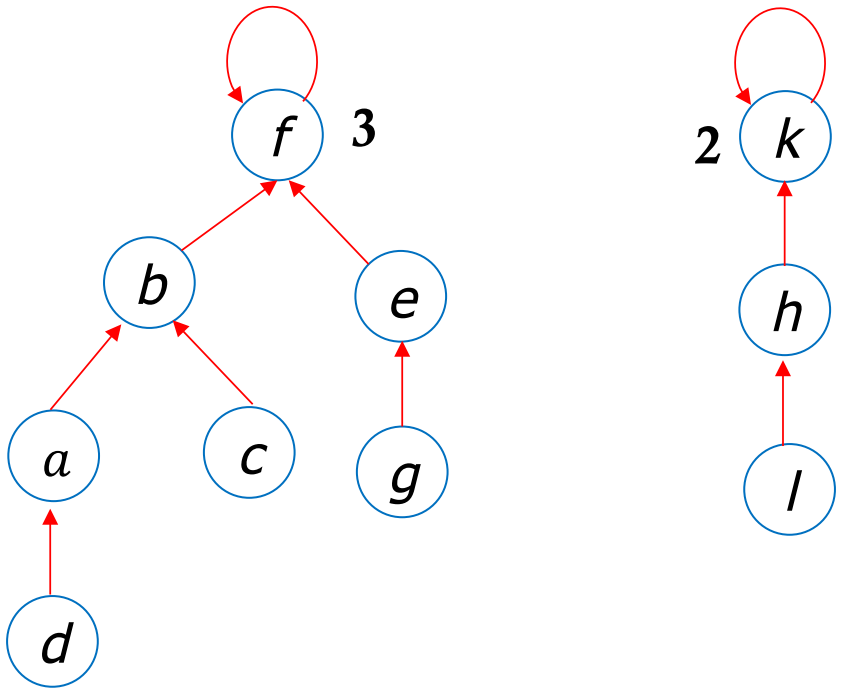
$Find\text{-}Set\ (x)$

If $x.parent = x$

return x

else

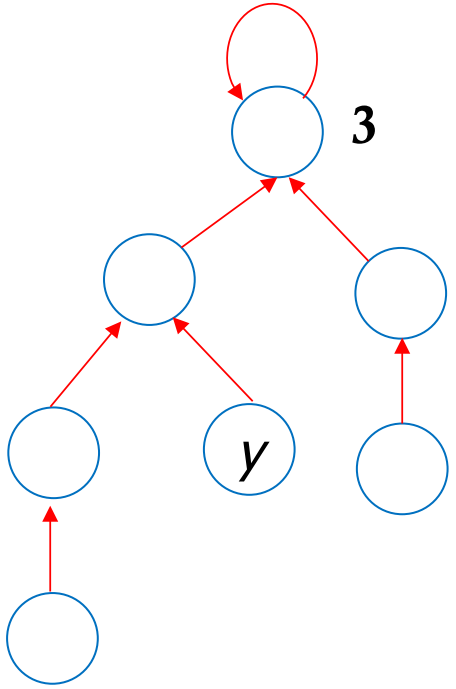
return $Find\text{-}Set(x.parent)$

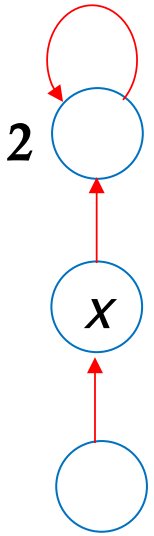


What is the running time?

Union pseudocode

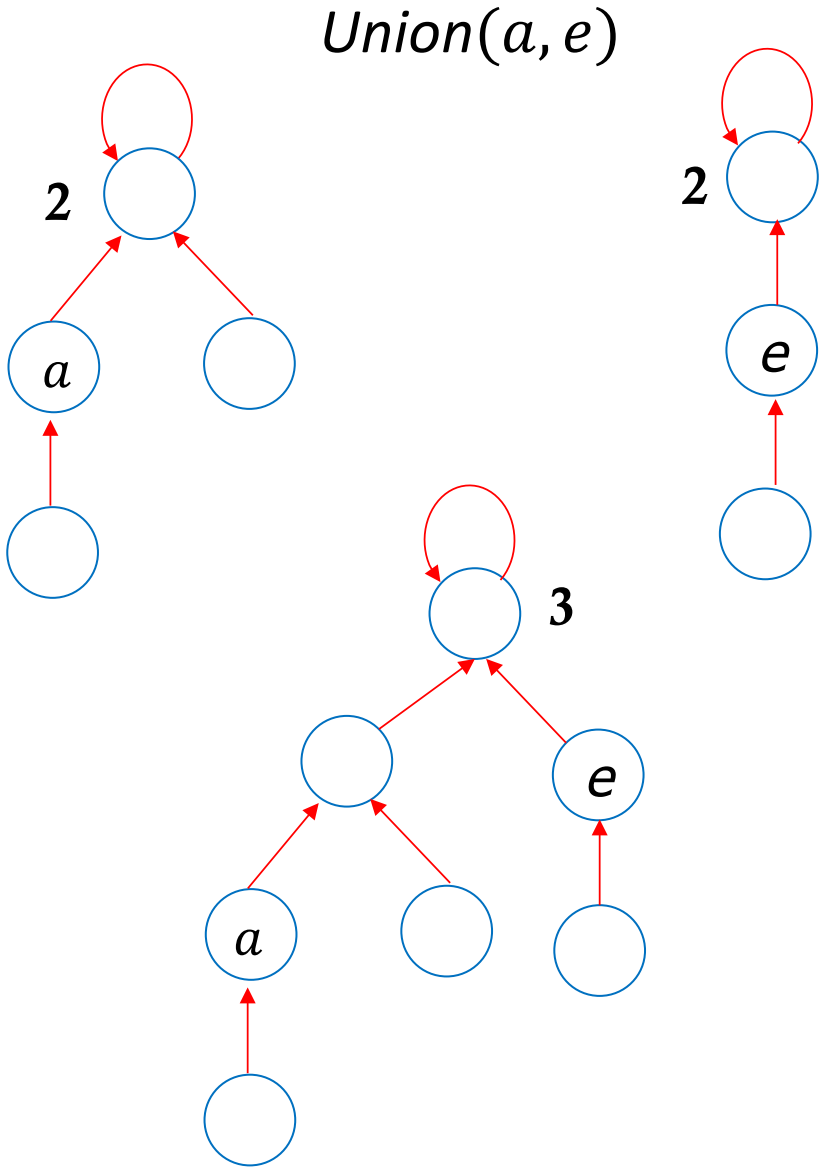
```
Union(x, y)
    xRoot ⟵ Find-Set(x)
    yRoot ⟵ Find-Set(y)
    if xRoot = yRoot
        return
    if xRoot.rank < yRoot.rank
        xRoot.parent ⟵ yRoot
    else
        yRoot.parent ⟵ xRoot
        if xRoot.rank = yRoot.rank
            xRoot.rank ⟵ xRoot.rank + 1
```





Union pseudocode

```
Union(x, y)
    xRoot ⟵ Find-Set(x)
    yRoot ⟵ Find-Set(y)
    if xRoot = yRoot
        return
    if xRoot.rank < yRoot.rank
        xRoot.parent ⟵ yRoot
    else
        yRoot.parent ⟵ xRoot
        if xRoot.rank = yRoot.rank
            xRoot.rank ⟵ xRoot.rank + 1
```



Time complexity of m operations: O(m log m)

Disjoint Forest:

2-Path compression

Second heuristics: Path Compression

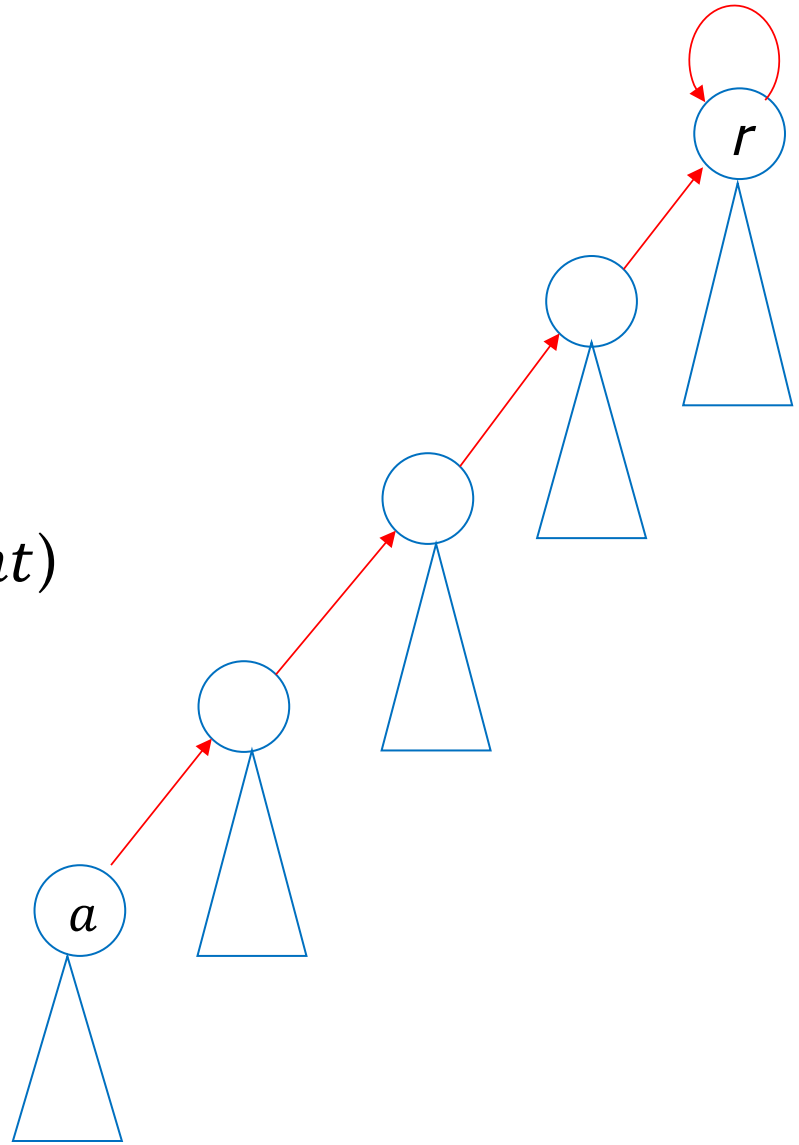
Find-Set **(x)**

If $x.parent = x$

return x

else

return $Find\text{-}Set(x.parent)$



Second heuristics: Path Compression

***Find-Set* (x)**

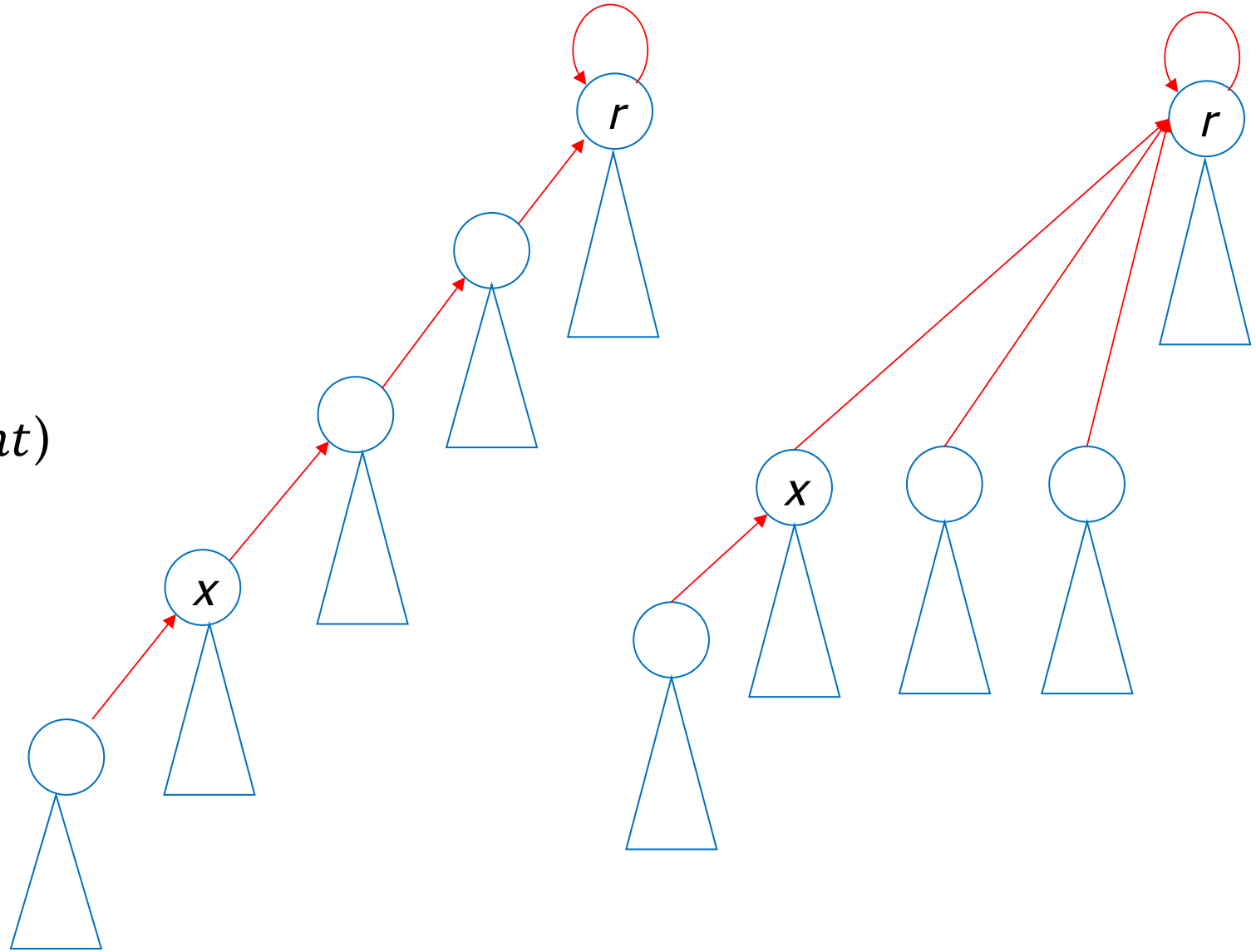
If $x.parent = x$

return x

else

x.parent = Find-Set(x.parent)

return $Find\text{-}Set(x.parent)$



Time complexity

m operations:

O(m log m) in the worst case. The proof is not part of this course.

Disjoint Forest:

3-Union by rank with path compression

Union by rank with path compression

Path compression happens in the Find-Set operation

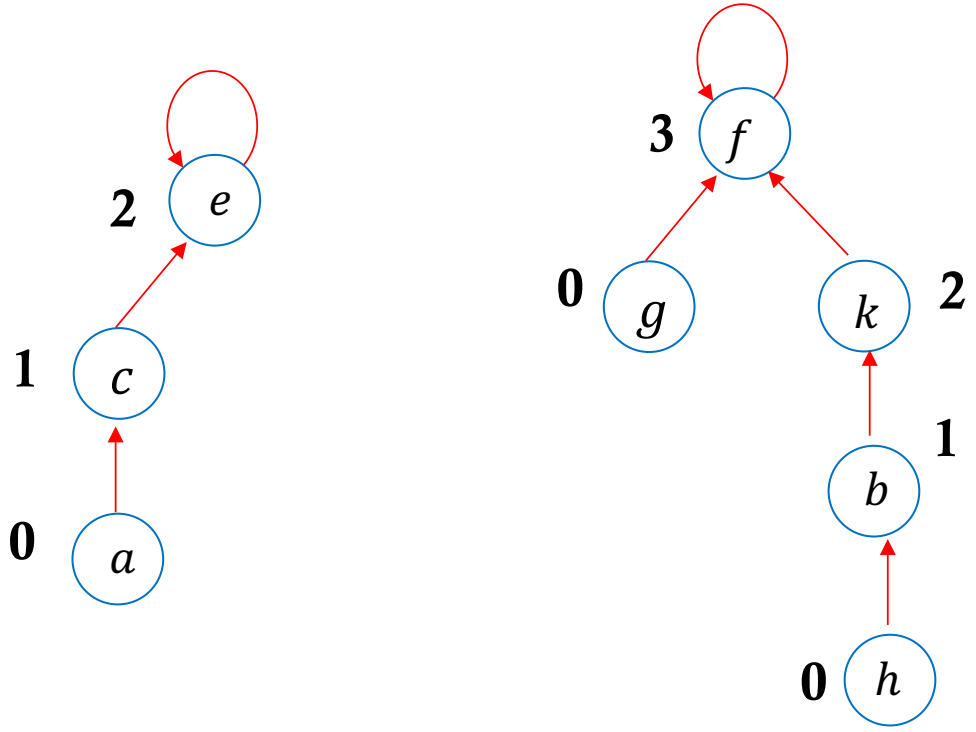
- Union-by-rank happens in the Union operation (outside Find-Set)
- So they don't really interfere with each other, simply use them both!

Note: rank ≠ height

- because path compression does NOT maintain height info
- a node's rank is an **upper-bound** on its height

Exercise

Draw the result after Union(a,b).
using both union-by-rank and path compression.



Union by rank with path compression

Path compression happens in the FindSet operation

- Union-by-rank happens in the Union operation (outside FindSet)
- So they don't really interfere with each other, simply use them both!

Note: rank ≠ height

- because path compression does NOT maintain height info
- a node's rank is an **upper-bound** on its height

Summary

- Linked list with extra pointers $\Theta(m^2)$
  - Linked list with union-by-weight $\Theta(\cancel{m \log m})$ m + nlogn
- Disjoint forest $\Theta(m^2)$
  - Disjoint forest with union by rank $\Theta(m \log \overset{n}{\cancel{m}})$
  - Disjoint forest with path compression $\Theta(m \log \overset{n}{\cancel{m}})$
  - Disjoint forest with union by rank and path compression $\Theta(m \log^* \overset{n}{\cancel{m}})$

n #of element in the forest = V

mlog*n

Where m is the number of operations

Questions?