Disjoint Sets
Part 1

Ira A. Fulton Schools of Engineering
Arizona State University

Data Structures for Disjoint Sets

A **disjoint set data structure** maintains a collection (set)

$$S = \{S_1, S_2, \ldots S_k\}$$

of disjoint dynamic (sub) sets.

Each element *x* of a set could be a **pointer** to an object, possibly with multiple fields.

Representative of a set: We choose one element of a set to identify the set, e.g., we use the root of a tree to identify a tree, or the head element of a linked list to access the linked list

Note that the representative is an element in the set.

Why Disjoint Sets?

- The universe is composed of many disjoint sets
- An element belongs to a unique set
- Given an element, we need to find the set it belongs to
- Given two elements, we need to decide whether they are in the same set
- Given two disjoint sets, we may need to replace them by their union

Operations on Disjoint Sets

Make-Set(x)

creates a new set whose only member is *x*. Obviously, x is the representative.

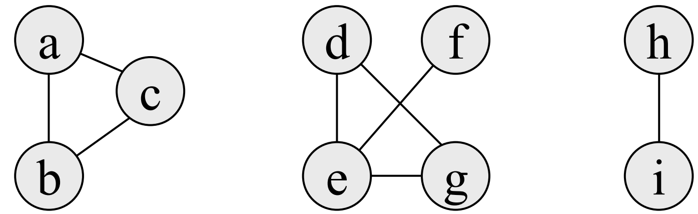
Union(x, y)

replaces the two sets S_x and S_y that contain elements *x* and *y* by their union. Sets are assumed to be disjoint (no element overlap). The representative of S_x or S_y becomes the representative of the united set.

Find-Set(x)

returns the pointer to the representative of the (unique) set containing *x*.

Application: Finding Connected Components



Connected-Components(G)

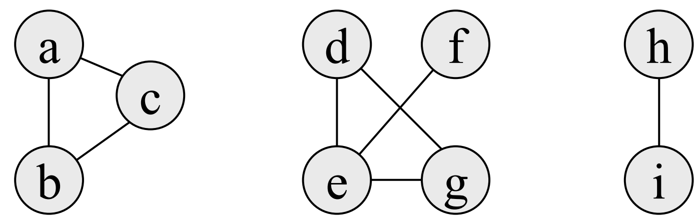
for each vertex v ∈ V(G) **do**
  Make-Set(v)
for each edge (u, v) ∈ E(G) **do**
  if Find-Set(u) ≠ Find-Set(v)
    then Union(u, v)

edge processed	collection of disjoint sets								
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}

Application: Finding Connected Components

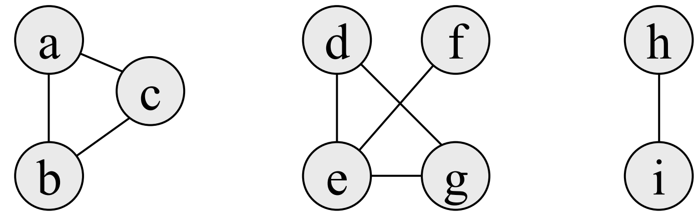
Connected-Components(G)

```
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)
```



edge processed	collection of disjoint sets								
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}
(a, b)	{a b}		{c}	{d}	{e}	{f}	{g}	{h}	{i}

Application: Finding Connected Components



Connected-Components(G)

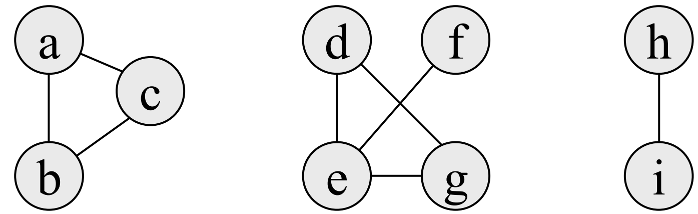
```
for each vertex v ∈ V(G) do
  Make-Set(v)
for each edge (u, v) ∈ E(G) do
  if Find-Set(u) ≠ Find-Set(v)
     then Union(u, v)
```

edge processed	collection of disjoint sets								
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}
(a, b)	{a b}		{c}	{d}	{e}	{f}	{g}	{h}	{i}
(a, c)	{a b c}			{d}	{e}	{f}	{g}	{h}	{i}

Application: Finding Connected Components

Connected-Components(G)

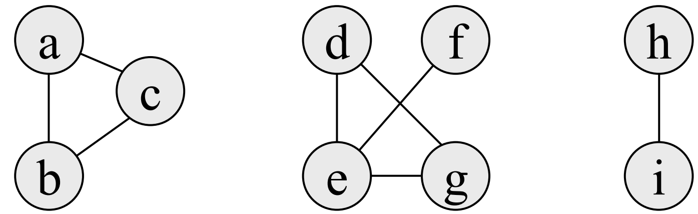
```
for each vertex v ∈ V(G) do
  Make-Set(v)
for each edge (u, v) ∈ E(G) do
  if Find-Set(u) ≠ Find-Set(v)
      then Union(u, v)
```



edge processed	collection of disjoint sets								
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}
(a, b)	{a b}		{c}	{d}	{e}	{f}	{g}	{h}	{i}
(a, c)	{a b c}			{d}	{e}	{f}	{g}	{h}	{i}
(b, c)	Find-Set(b) = Find-Set(c)								

Application: Finding Connected Components

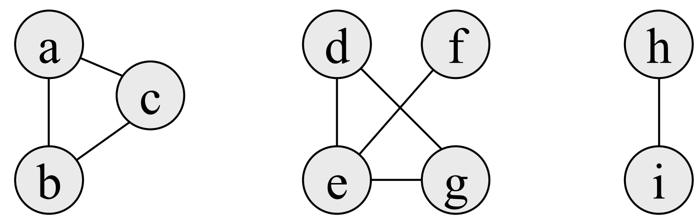
```
Connected-Components(G)
for each vertex v ∈ V(G) do
   Make-Set(v)
for each edge (u, v) ∈ E(G) do
   if Find-Set(u) ≠ Find-Set(v)
       then Union(u, v)
```



edge processed	collection of disjoint sets								
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}
(a, b)	{a b}		{c}	{d}	{e}	{f}	{g}	{h}	{i}
(a, c)	{a b c}			{d}	{e}	{f}	{g}	{h}	{i}
(b, c)	Find-Set(b) = Find-Set(c)								
(d, e)	{a b c}			{d e}		{f}	{g}	{h}	{i}

Application: Finding Connected Components

```
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)
```

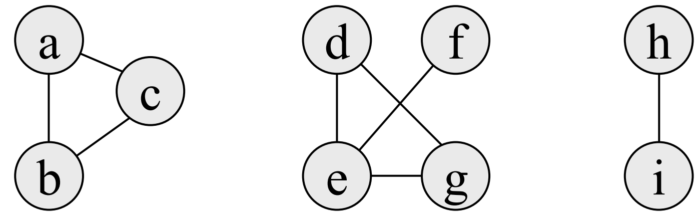


edge processed	collection of disjoint sets								
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}
(a, b)	{a b}		{c}	{d}	{e}	{f}	{g}	{h}	{i}
(a, c)	{a b c}			{d}	{e}	{f}	{g}	{h}	{i}
(b, c)	Find-Set(b) = Find-Set(c)								
(d, e)	{a b c}			{d e}		{f}	{g}	{h}	{i}
(d, g)	{a b c}			{d e g}		{f}		{h}	{i}

Application: Finding Connected Components

Connected-Components(G)

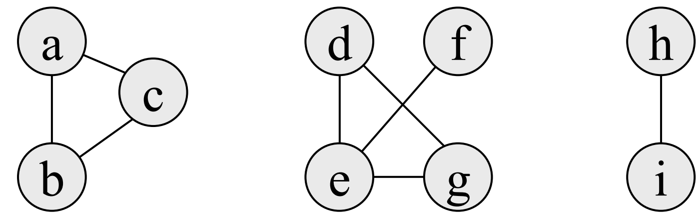
```
for each vertex v ∈ V(G) do
    Make-Set(v)
for each edge (u, v) ∈ E(G) do
    if Find-Set(u) ≠ Find-Set(v)
        then Union(u, v)
```



edge processed	collection of disjoint sets								
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}
(a, b)	{a b}		{c}	{d}	{e}	{f}	{g}	{h}	{i}
(a, c)	{a b c}			{d}	{e}	{f}	{g}	{h}	{i}
(b, c)	Find-Set(b) = Find-Set(c)								
(d, e)	{a b c}			{d e}		{f}	{g}	{h}	{i}
(d, g)	{a b c}			{d e g}		{f}		{h}	{i}
(e, f)	{a b c}			{d e g f}				{h}	{i}

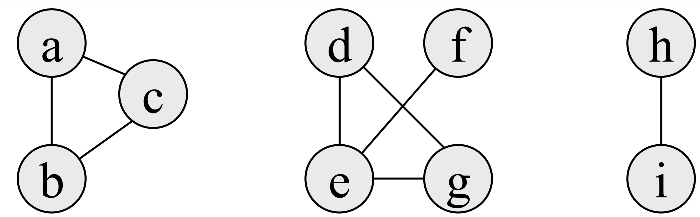
Application: Finding Connected Components

```
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)
```



edge processed	collection of disjoint sets								
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}
(a, b)	{a b}		{c}	{d}	{e}	{f}	{g}	{h}	{i}
(a, c)	{a b c}			{d}	{e}	{f}	{g}	{h}	{i}
(b, c)	Find-Set(b) = Find-Set(c)								
(d, e)	{a b c}			{d e}		{f}	{g}	{h}	{i}
(d, g)	{a b c}			{d e g}		{f}		{h}	{i}
(e, f)	{a b c}			{d e g f}				{h}	{i}
(e, g)	Find-Set(e) = Find-Set(g)								

Application: Finding 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) $\neq$ Find-Set(v)

    then Union(u, v)

edge processed	collection of disjoint sets							
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}
(a, b)	{a b}		{c}	{d}	{e}	{f}	{g}	{h}	{i}
(a, c)	{a b c}			{d}	{e}	{f}	{g}	{h}	{i}
(b, c)	Find-Set(b) = Find-Set(c)								
(d, e)	{a b c}			{d e}		{f}	{g}	{h}	{i}
(d, g)	{a b c}			{d e g}		{f}		{h}	{i}
(e, f)	{a b c}			{d e g f}				{h}	{i}
(e, g)	Find-Set(e) = Find-Set(g)								
(h, i)	{a b c}			{d e g f}				{h i}	

Summary

- Disjoint set is an important data structure
- It can be used to find connected components in an undirected graph
- It can be used to compute minimum spanning trees

Ira A. Fulton Schools of Engineering

Arizona State University

Disjoint Sets
Part 2

Ira A. Fulton Schools of Engineering
Arizona State University

Disjoint-set Forest

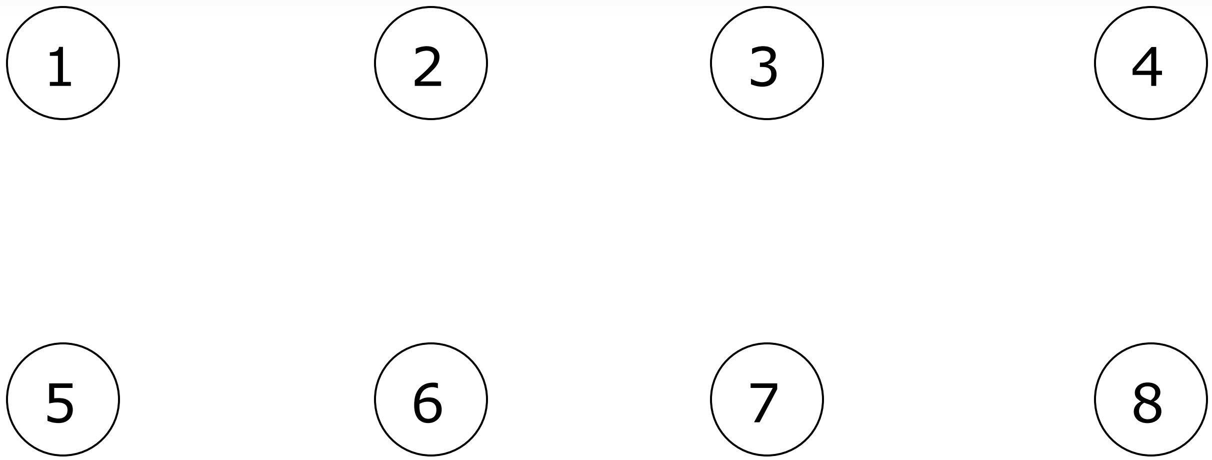
- We can use a forest to represent disjoint sets.
- Each set is represented by a tree.
- The root is the representative.
- How to perform Make-Set?
- How to perform Find(x)?
- How to perform Union(x, y)?
- What are the time complexities?
- How do we represent the forest?

Array Implementation of Disjoint Sets

- We use a linear array A with index from 1 to n to implement disjoint sets on the first n integers.
- **Make-Set(x)** is accomplished by
  - A[x] := 0; // x is the root, the rank of the tree at x is 0.
  - Refer to p. 530 of the textbook, where two fields are used

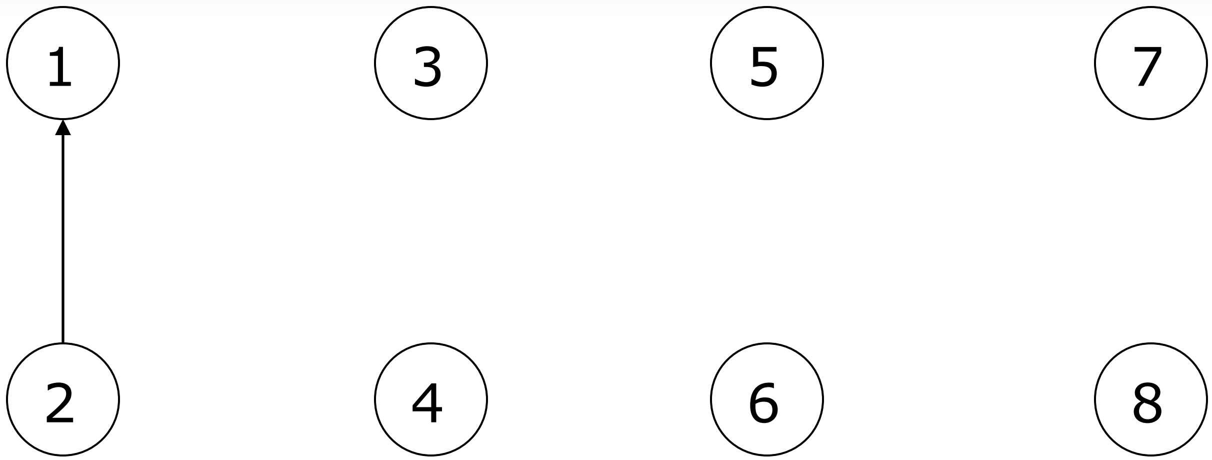
- A[x] > 0 indicates that x is NOT the root and that the parent of x is at location A[x].
- A[x] <= 0 indicates that x IS the root and that the rank of the tree rooted at x is –A[x].

Forest View: 8 trees



A:	0	0	0	0	0	0	0	0

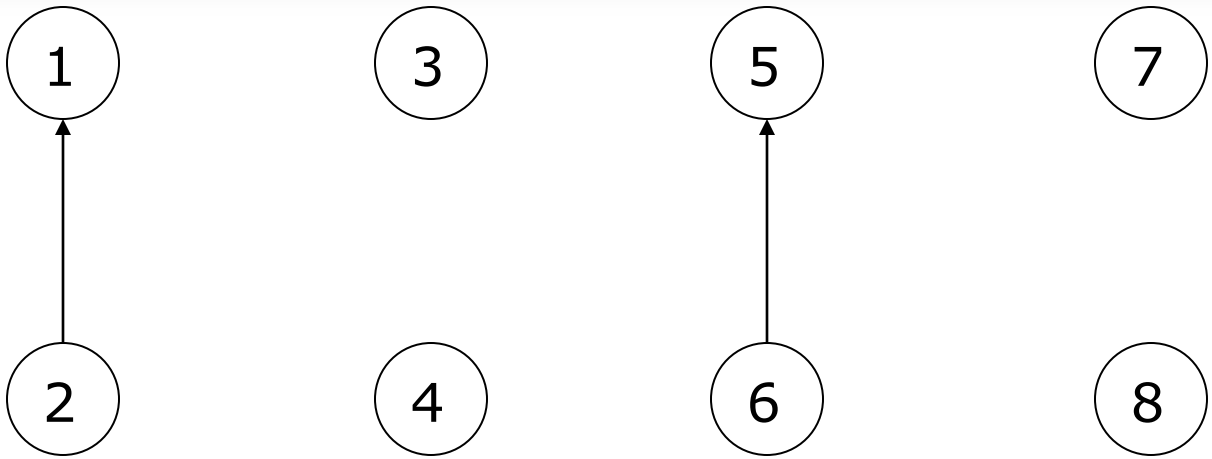
Forest View: 7 trees



A:

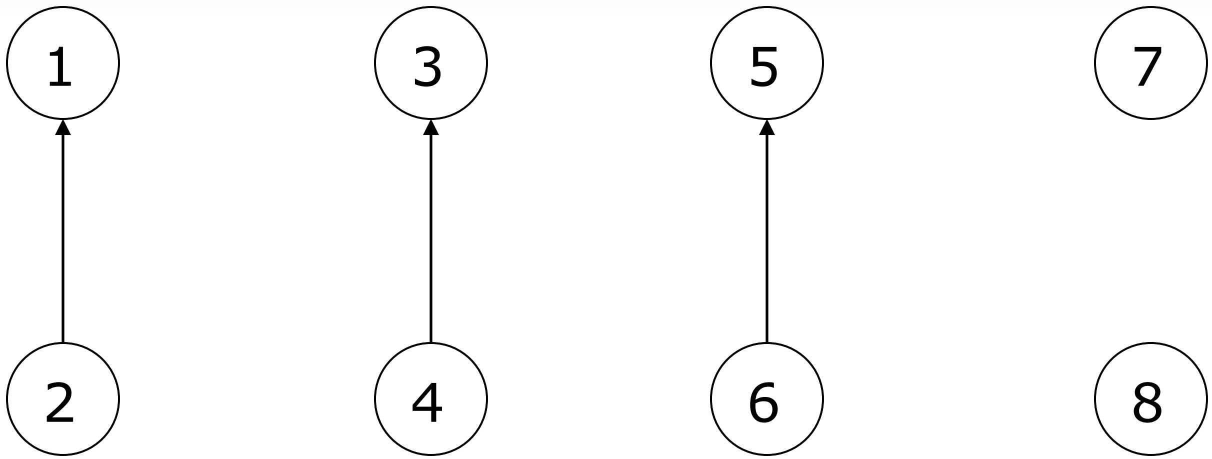
-1	1	0	0	0	0	0	0

Forest View: 6 trees



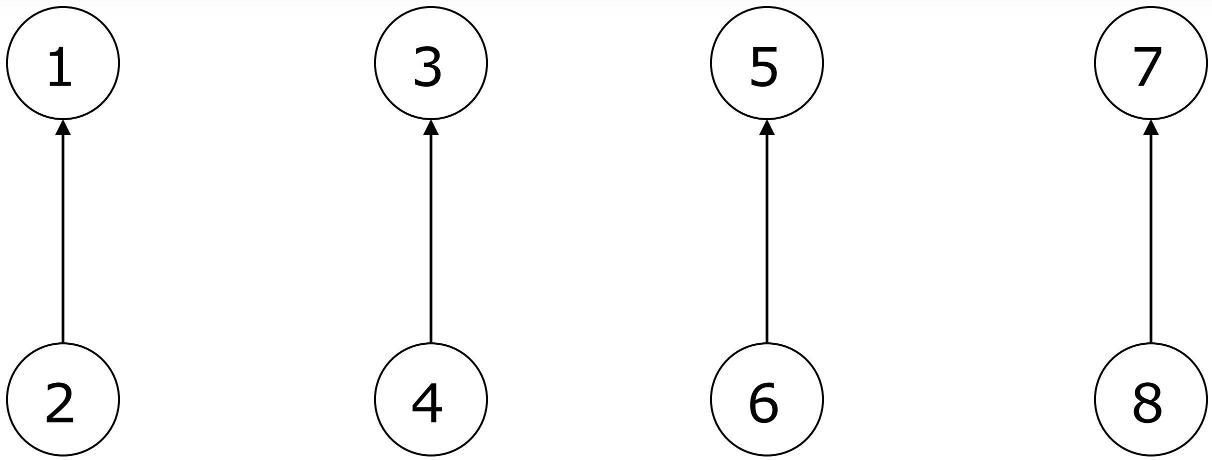
A:	-1	1	0	0	-1	5	0	0

Forest View: 5 trees



A:	-1	1	-1	3	-1	5	0	0

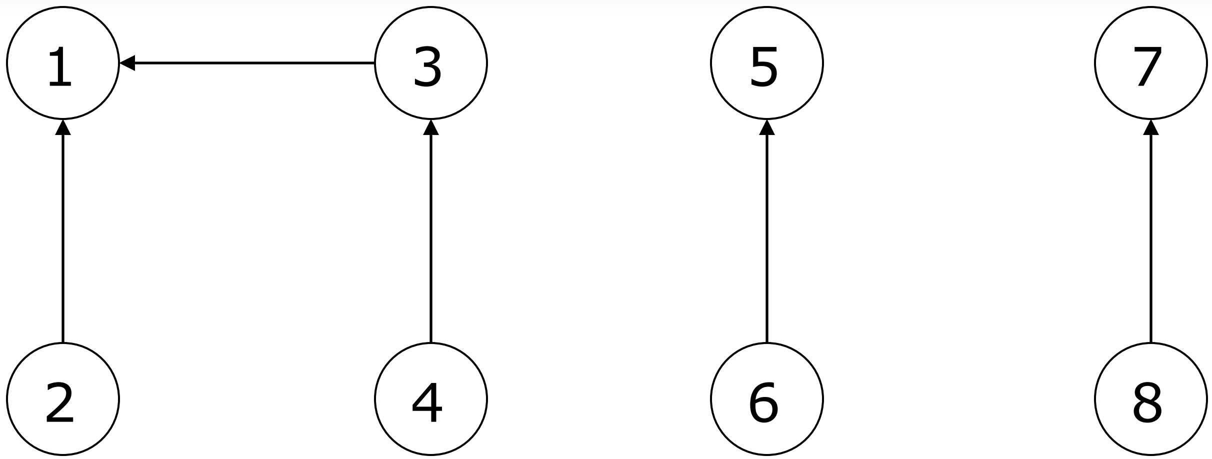
Forest View: 4 trees



A:

-1	1	-1	3	-1	5	-1	7

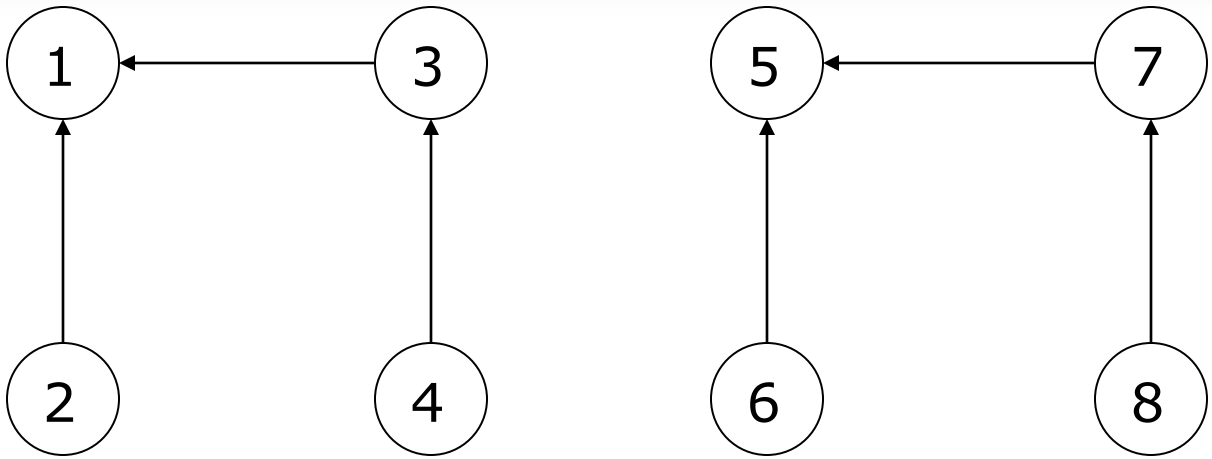
Forest View: 3 trees



A:

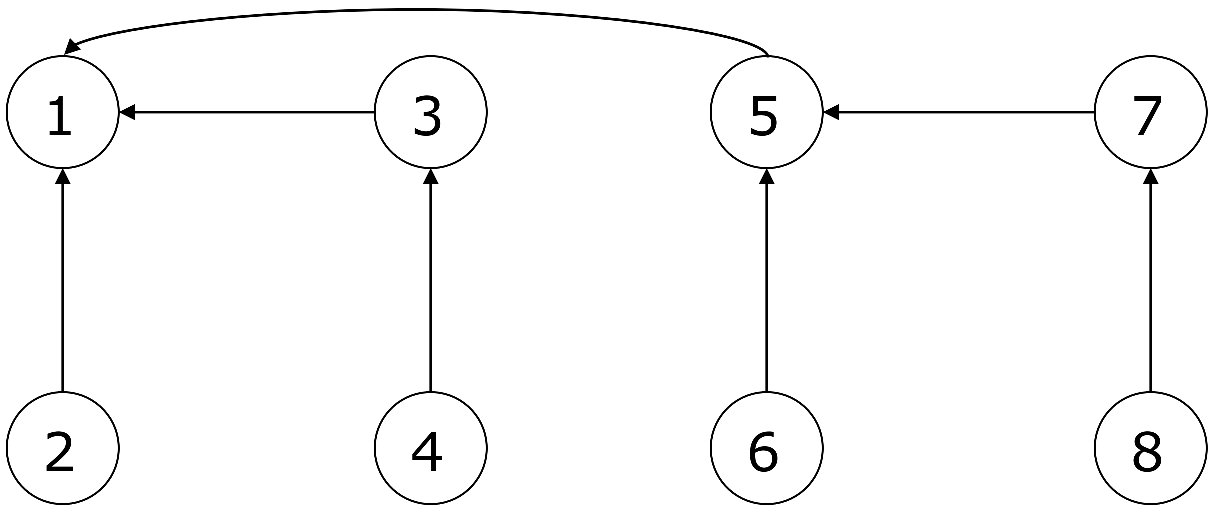
-2	1	1	3	-1	5	-1	7

Forest View: 2 trees



A:	-2	1	1	3	-2	5	5	7

Forest View: 1 tree



A:

-3	1	1	3	1	5	5	7

Summary

- We use a linear array A with index from 1 to n to implement disjoint sets on the first n integers.
- **Make-Set(x)** is accomplished by
  - **A[x] := 0;** // x is the root, the rank of the tree at x is 0.
- A[x] > 0 indicates that x is NOT the root and that the parent of x is at location A[x].
- A[x] <= 0 indicates that x IS the root and that the <u>rank</u> of the tree rooted at x is –A[x].

Ira A. Fulton Schools of Engineering

Arizona State University

Disjoint Sets
Part 3

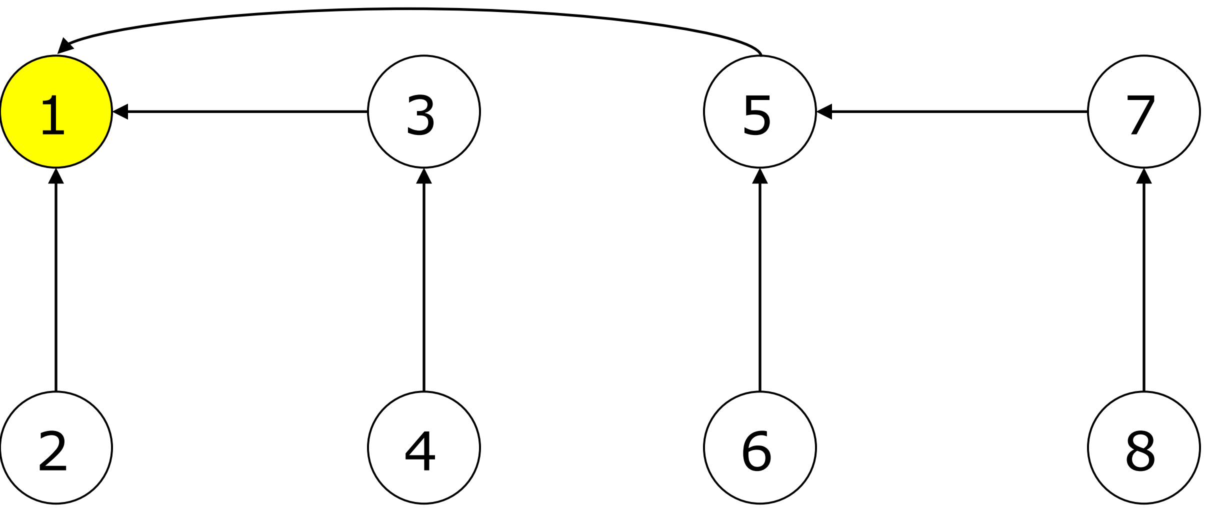
Ira A. Fulton Schools of Engineering
Arizona State University

Find-Set with path compression

- **Find-Set(x) is accomplished by**

```
Find-Set(x){
  if A[x]<=0 then{
   return x;
 }else{
   A[x]=Find-Set(A[x]);
   return A[x];
 }
}
```

Examples of Find-Set(1)

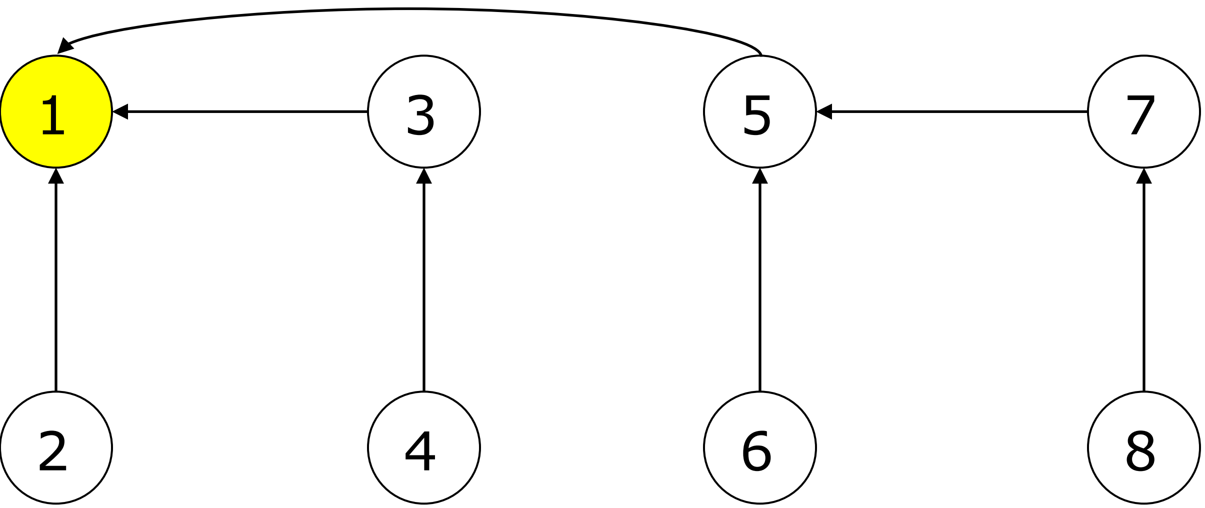



```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:	-3	1	1	3	1	5	5	7

- `Find-Set(1)`

Examples of Find-Set(1)

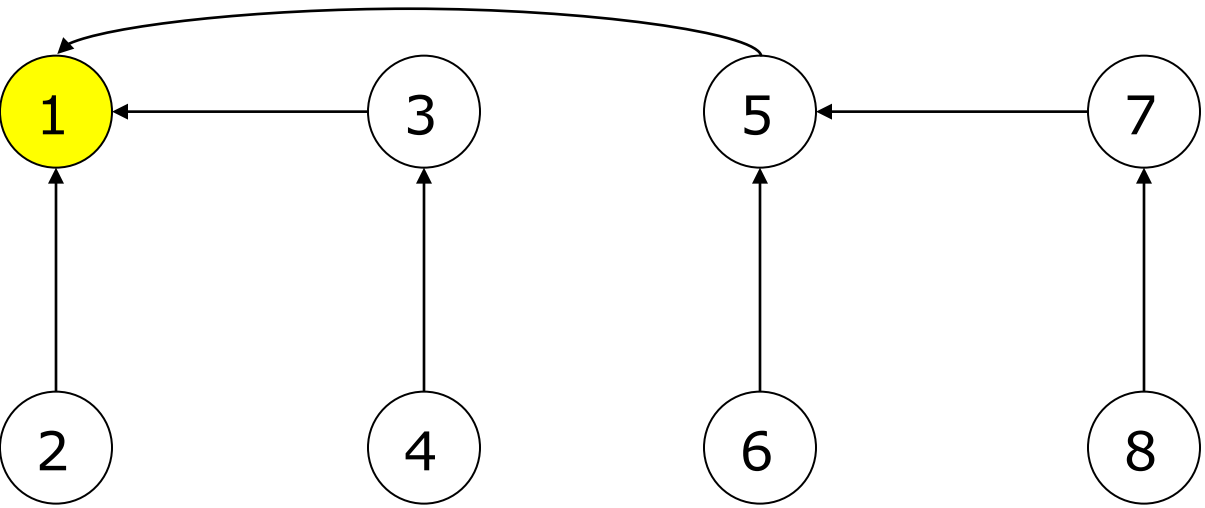



```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:	-3	1	1	3	1	5	5	7

- `Find-Set(1)`
- `A[1]=-3 <= 0`

Examples of Find-Set(1)

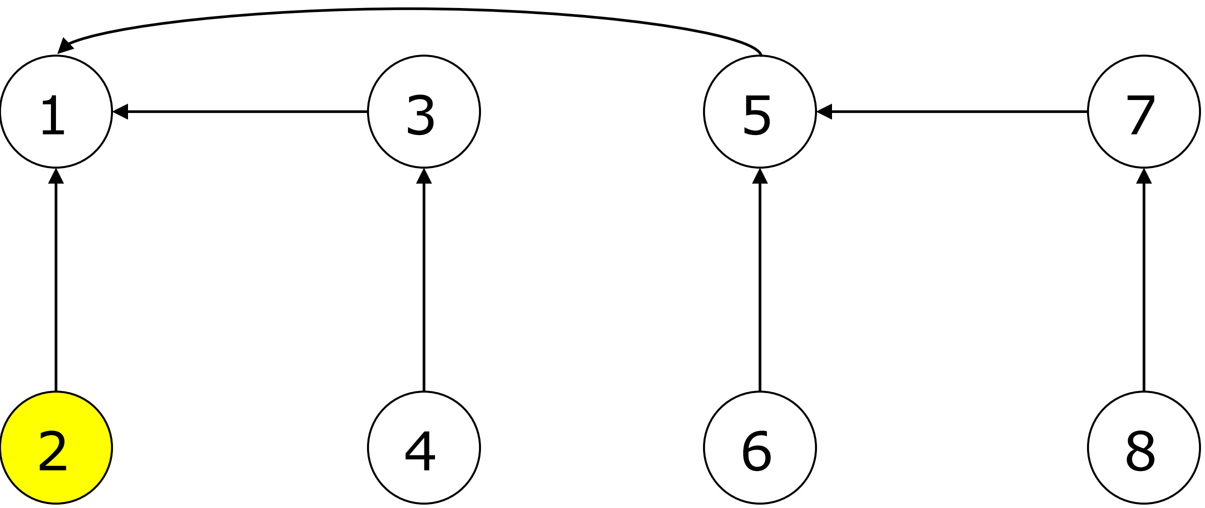



```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:								
	-3	1	1	3	1	5	5	7

- `Find-Set(1)`
- `A[1]=-3 <= 0`
- `return 1`

Examples of Find-Set(2)

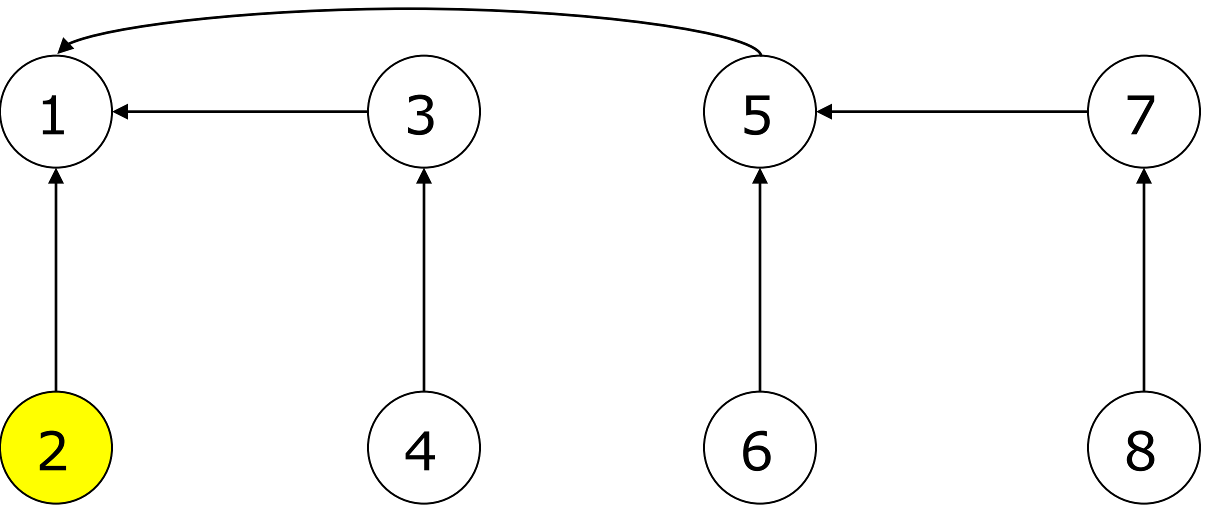



```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:								
	-3	1	1	3	1	5	5	7

- `Find-Set(2)`

Examples of Find-Set(2)



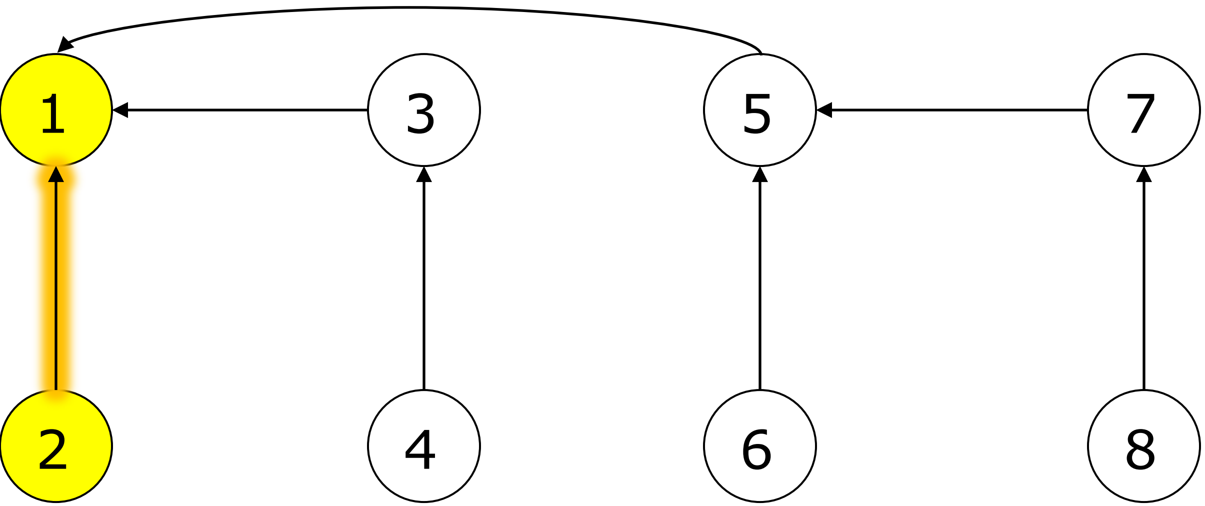

```
Find-Set(x){
   if A[x]<=0 then{
     return x
   }else{
     A[x]=Find-Set(A[x])
     return A[x]
   }
}
```

A:

-3	1	1	3	1	5	5	7

- `Find-Set(2)`
- `A[2]=1 > 0`

Examples of Find-Set(2)



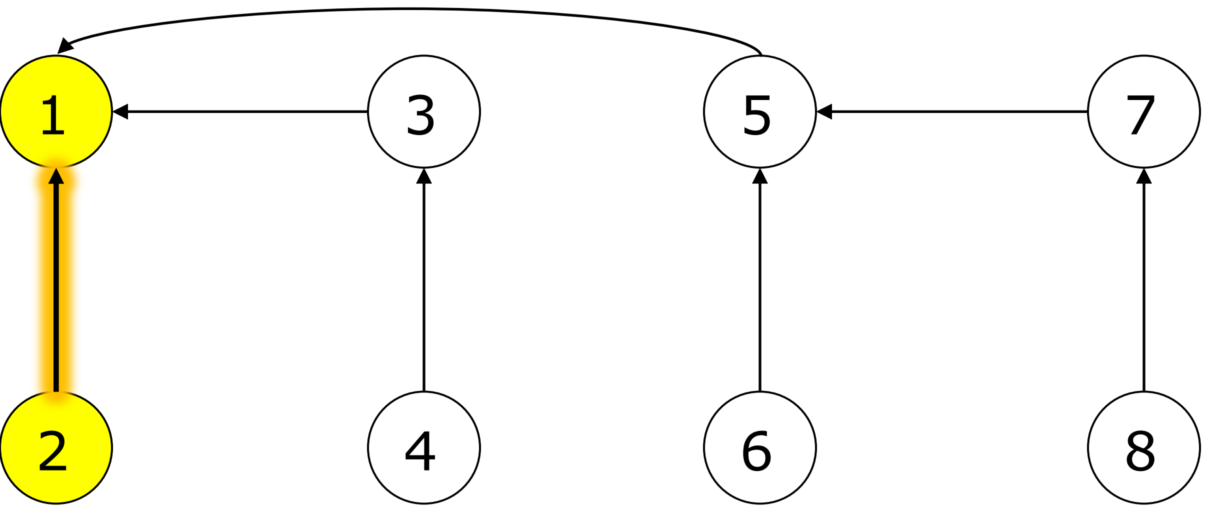

```
Find-Set(x){
   if A[x]<=0 then{
      return x
   }else{
      A[x]=Find-Set(A[x])
      return A[x]
   }
}
```

A:

-3	1	1	3	1	5	5	7

- `Find-Set(2)`
- `A[2]=1 > 0`
- `A[2]=Find-Set(1)`

Examples of Find-Set(2)



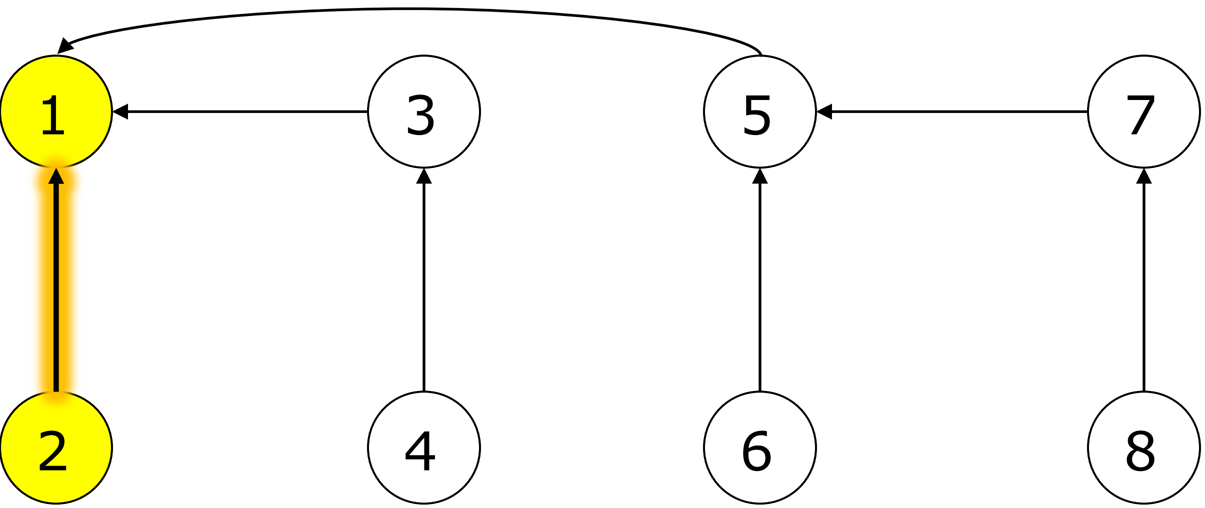

```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:

-3	**1**	1	3	1	5	5	7

- `Find-Set(2)`
- `A[2]=1 > 0`
- `A[2]=Find-Set(1)=1`

Examples of Find-Set(2)



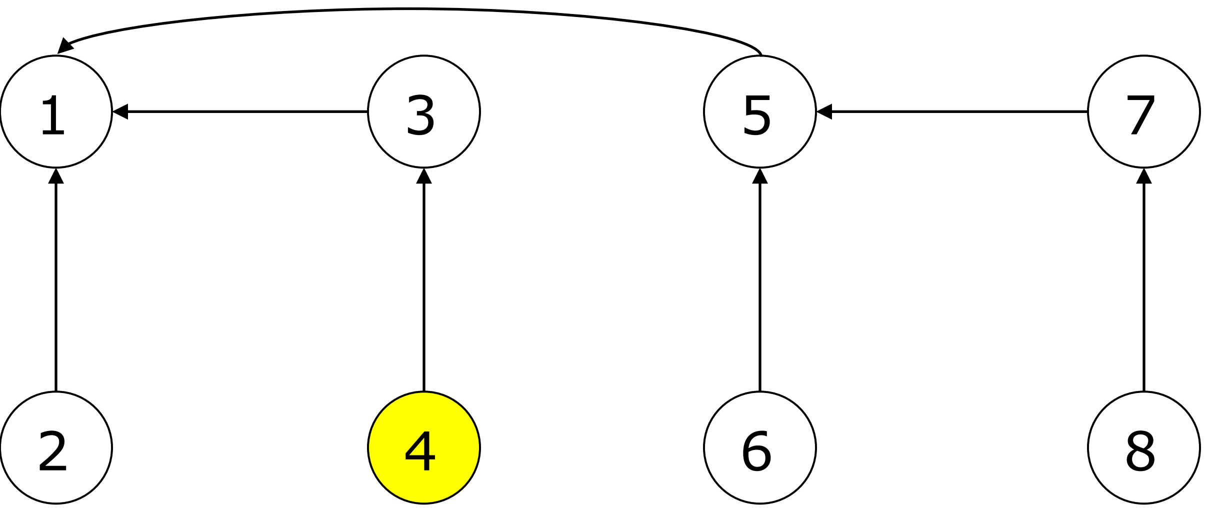

```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:

-3	**1**	1	3	1	5	5	7

- `Find-Set(2)`
- `A[2]=1 > 0`
- `A[2]=Find-Set(1)=1`
- `return 1`

Examples of Find-Set(4)

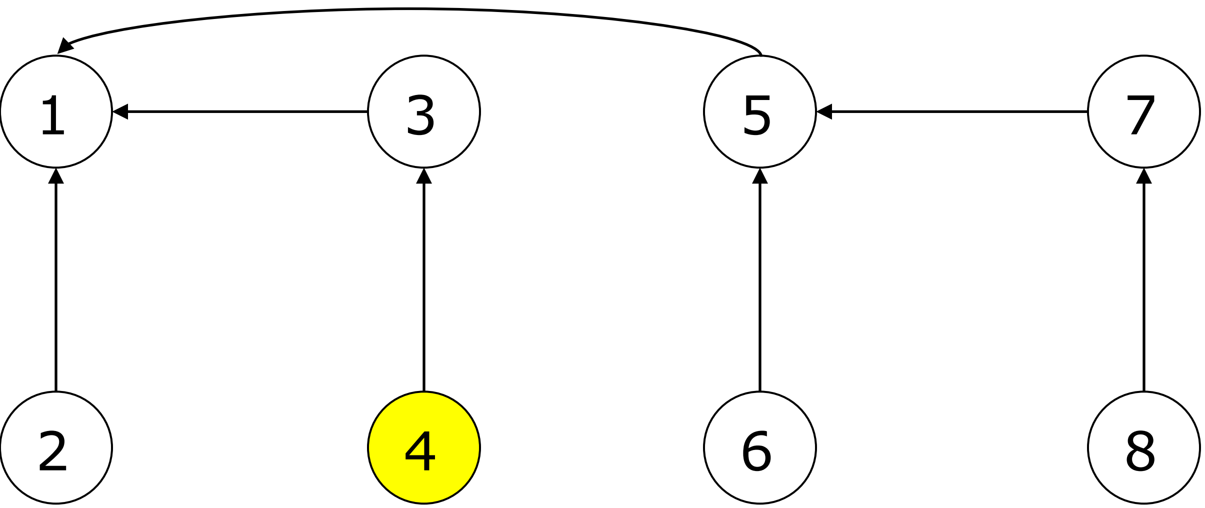



```
Find-Set(x){
   if A[x]<=0 then{
     return x
   }else{
     A[x]=Find-Set(A[x])
     return A[x]
   }
}
```

A:							
-3	1	1	3	1	5	5	7

- `Find-Set(4)`

Examples of Find-Set(4)

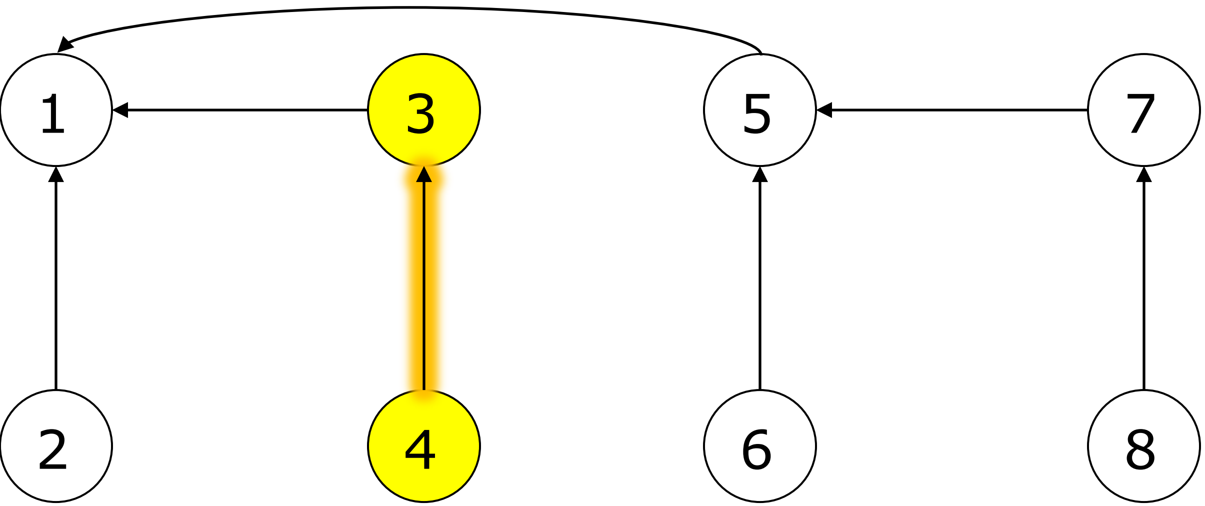



```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:								
	-3	1	1	3	1	5	5	7

- `Find-Set(4)`
- `A[4]=3 > 0`

Examples of Find-Set(4)



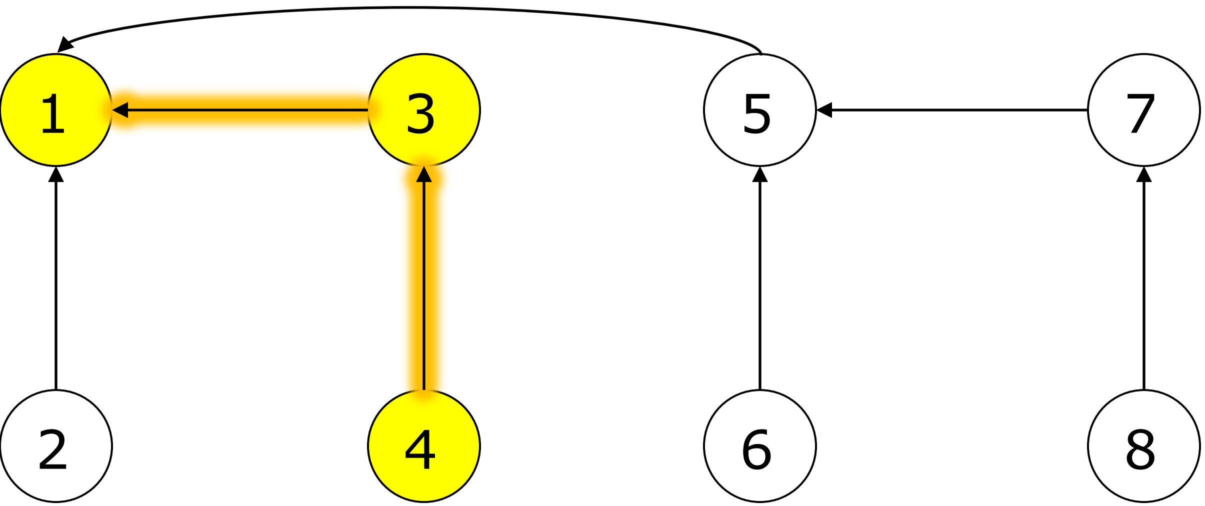

```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:

-3	1	1	3	1	5	5	7

- `Find-Set(4)`
- `A[4]=3 > 0`
- `A[4]=Find-Set(3)`

Examples of Find-Set(4)



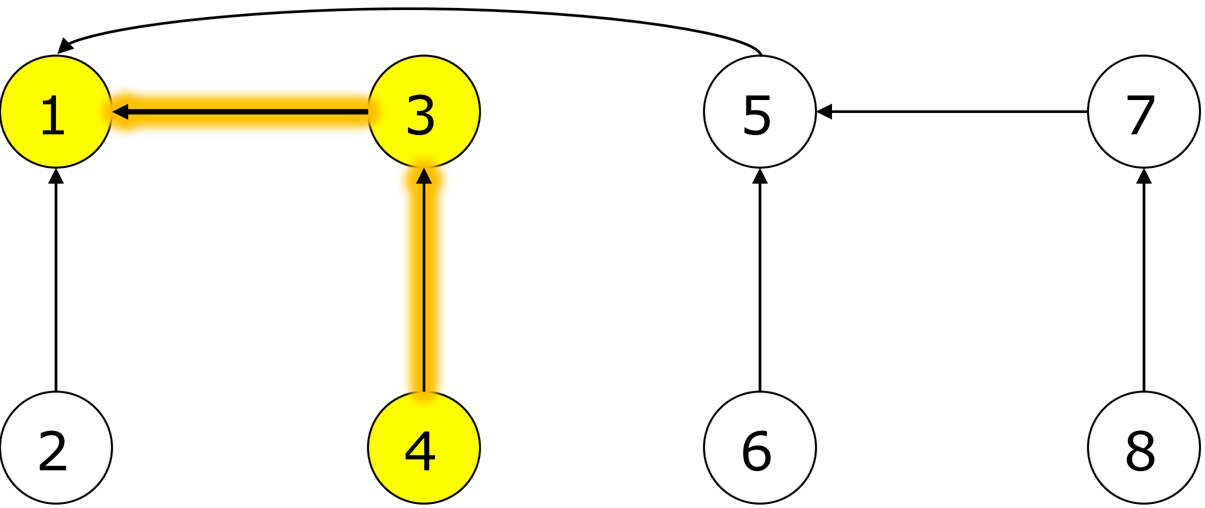

```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:

-3	1	1	3	1	5	5	7

- `Find-Set(4)`
- `A[4]=3 > 0`
- `A[4]=Find-Set(3)=Find-Set(Find-Set(1))`

Examples of Find-Set(4)

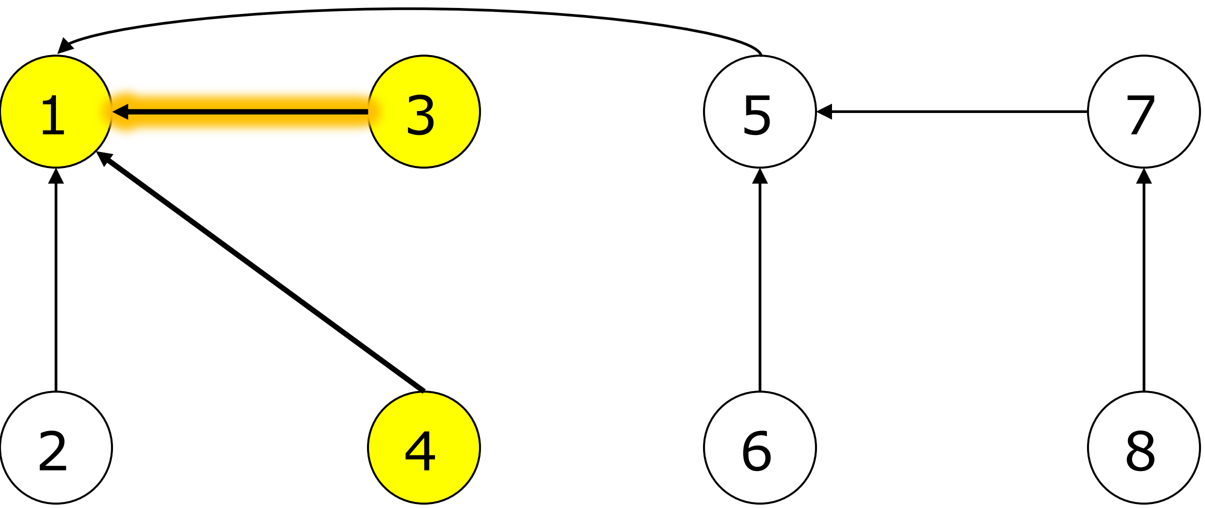



```
Find-Set(x){
   if A[x]<=0 then{
      return x
   }else{
      A[x]=Find-Set(A[x])
      return A[x]
   }
}
```

A:								
	-3	1	**1**	3	1	5	5	7

- `Find-Set(4)`
- `A[4]=3 > 0`
- `A[4]=Find-Set(3)=Find-Set(Find-Set(1))=1`

Examples of Find-Set(4)

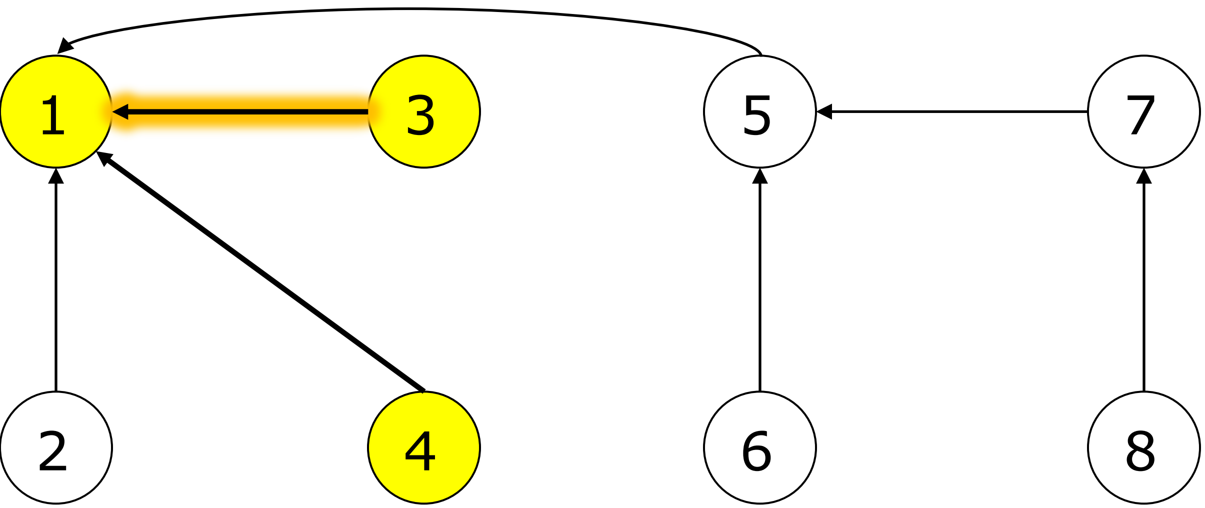



```
Find-Set(x){
   if A[x]<=0 then{
     return x
   }else{
     A[x]=Find-Set(A[x])
     return A[x]
   }
}
```

A:								
	-3	1	**1**	**1**	1	5	5	7

- `Find-Set(4)`
- `A[4]=3 > 0`
- `A[4]=Find-Set(A[4])=Find-Set(Find-Set(1))=1`

Examples of Find-Set(4)



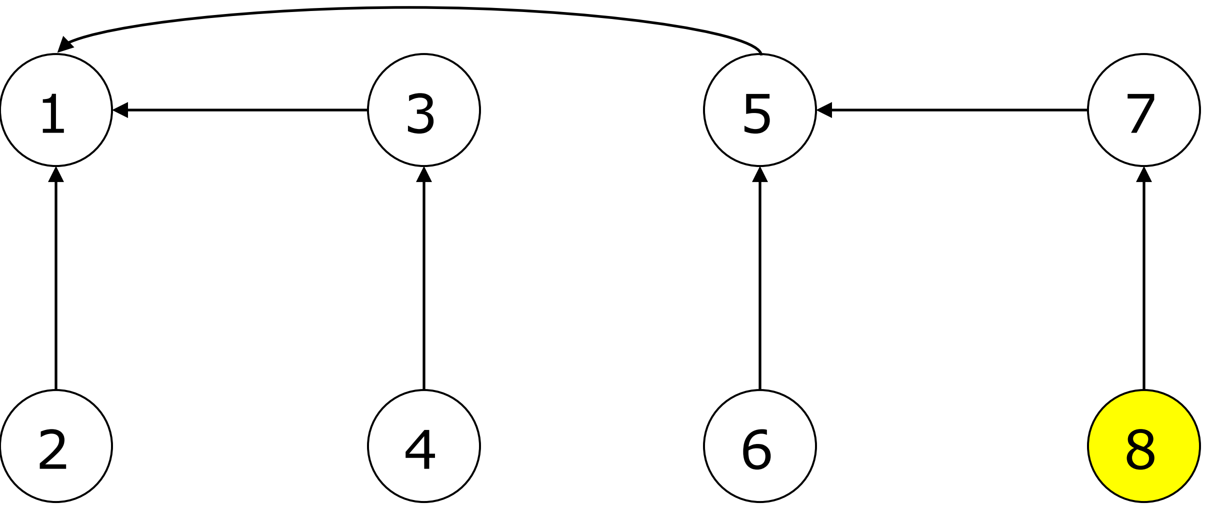

```
Find-Set(x){
   if A[x]<=0 then{
      return x
   }else{
      A[x]=Find-Set(A[x])
      return A[x]
   }
}
```

A:

-3	1	**1**	**1**	1	5	5	7

- `Find-Set(4)`
- `A[4]=3 > 0`
- `A[4]=Find-Set(A[4])=Find-Set(Find-Set(1))=1`
- `return 1`

Examples of Find-Set(8)

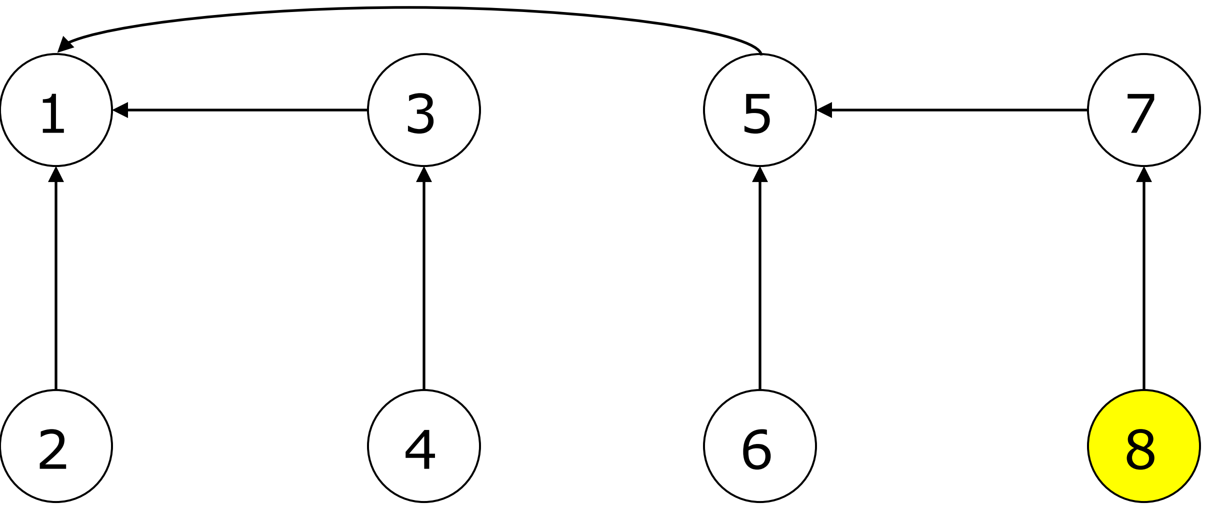



```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:								
	-3	1	1	3	1	5	5	7

- `Find-Set(8)`

Examples of Find-Set(8)

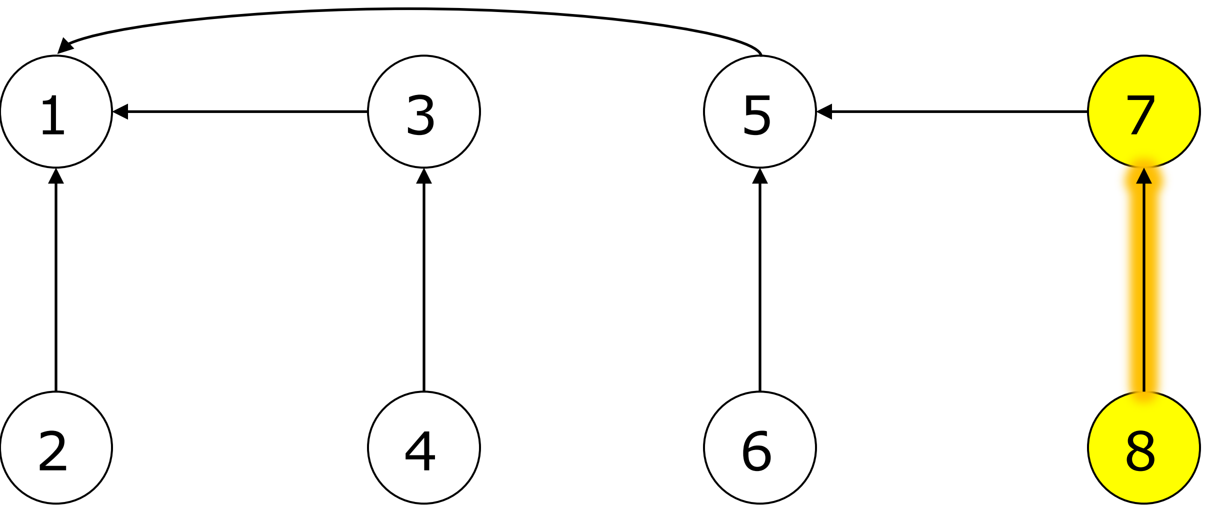



```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:								
	-3	1	1	3	1	5	5	7

- `Find-Set(8)`
- `A[8]=7 > 0`

Examples of Find-Set(8)



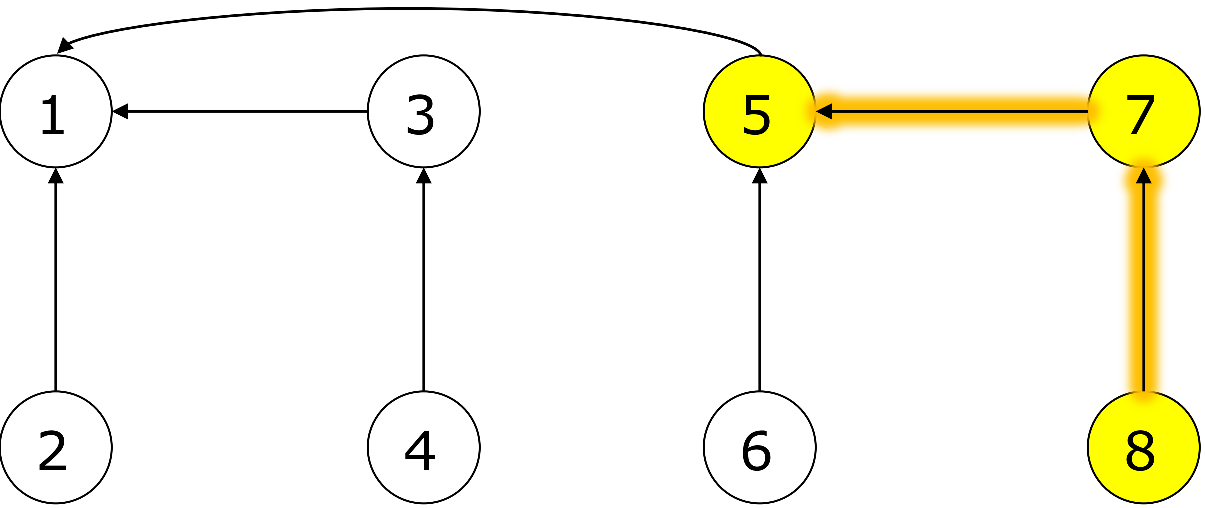

```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:

-3	1	1	3	1	5	5	7

- `Find-Set(8)`
- `A[8]=7 > 0`
- `A[8]=Find-Set(7)`

Examples of Find-Set(8)



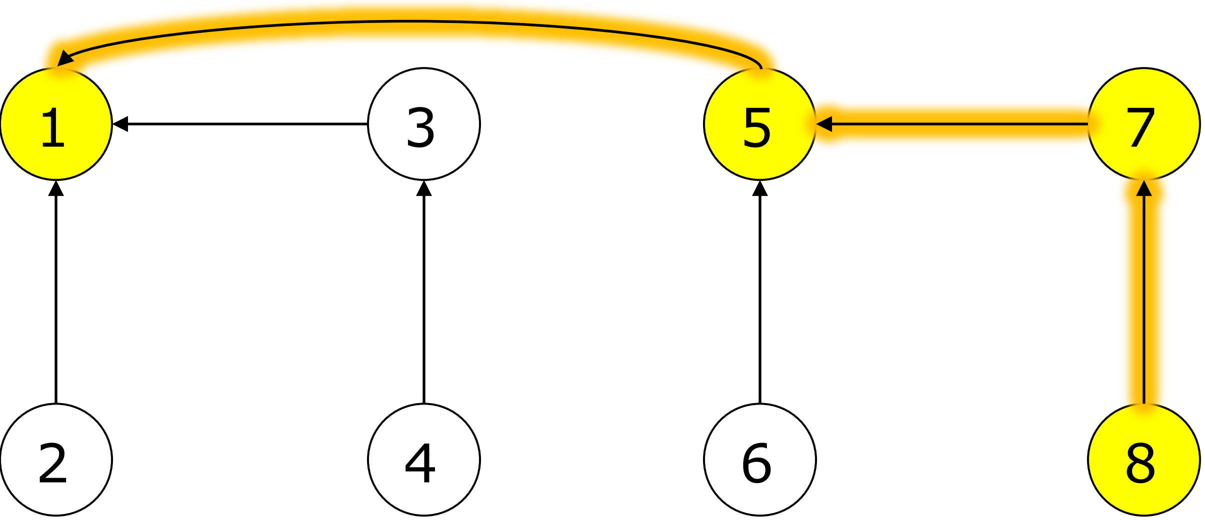

```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:

-3	1	1	3	1	5	5	7

- `Find-Set(8)`
- `A[8]=7 > 0`
- `A[8]=Find-Set(7)=Find-Set(5)`

Examples of Find-Set(8)



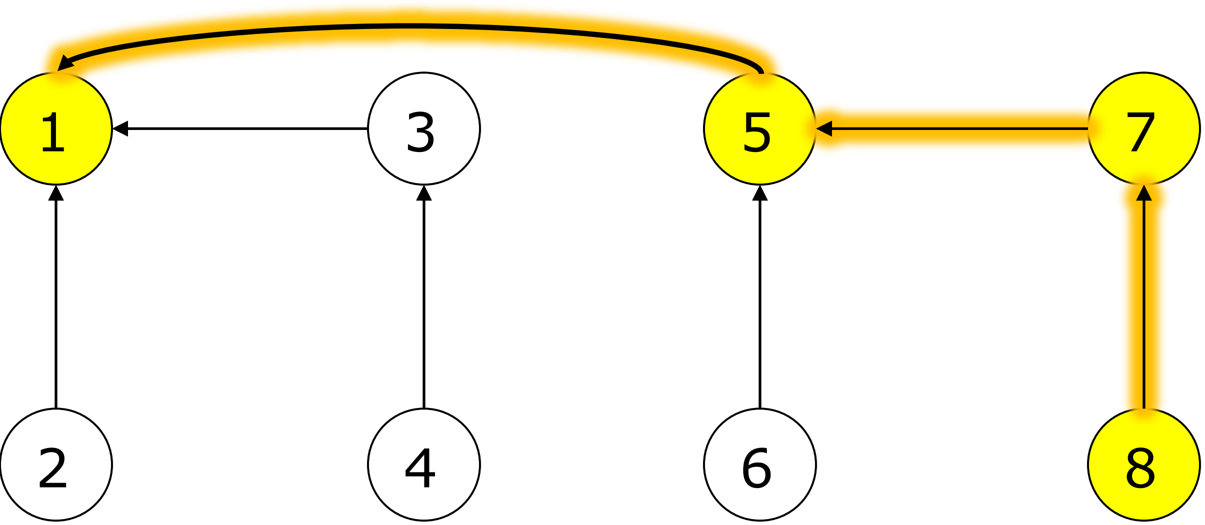

```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:

-3	1	1	3	1	5	5	7

- `Find-Set(8)`
- `A[8]=7 > 0`
- `A[8]=Find-Set(7)=Find-Set(5)=Find-Set(1)`

Examples of Find-Set(8)



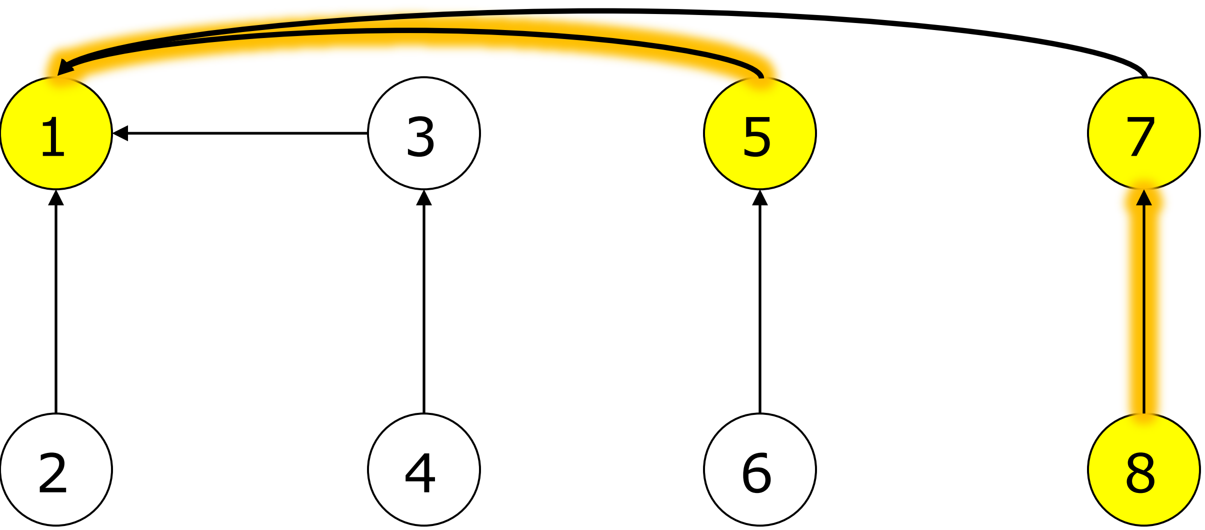

```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:

-3	1	1	3	**1**	5	5	7

- `Find-Set(8)`
- `A[8]=7 > 0`
- `A[8]=Find-Set(7)=Find-Set(5)=Find-Set(1)=1`

Examples of Find-Set(8)



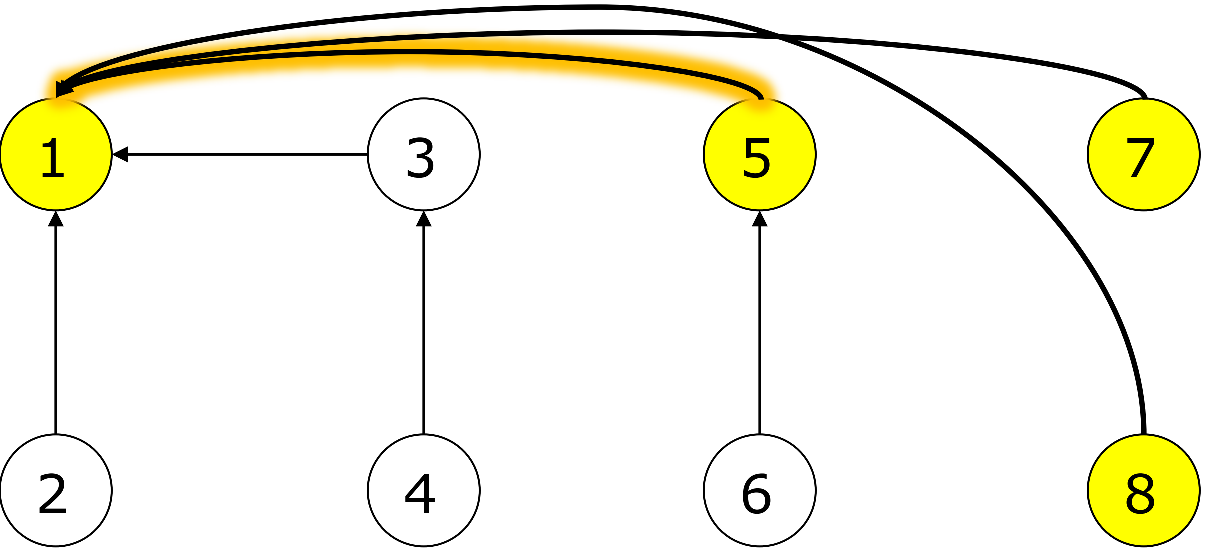

```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:

-3	1	1	3	**1**	5	**1**	7

- `Find-Set(8)`
- `A[8]=7 > 0`
- `A[8]=Find-Set(7)=Find-Set(5)=Find-Set(1)=1`

Examples of Find-Set(8)



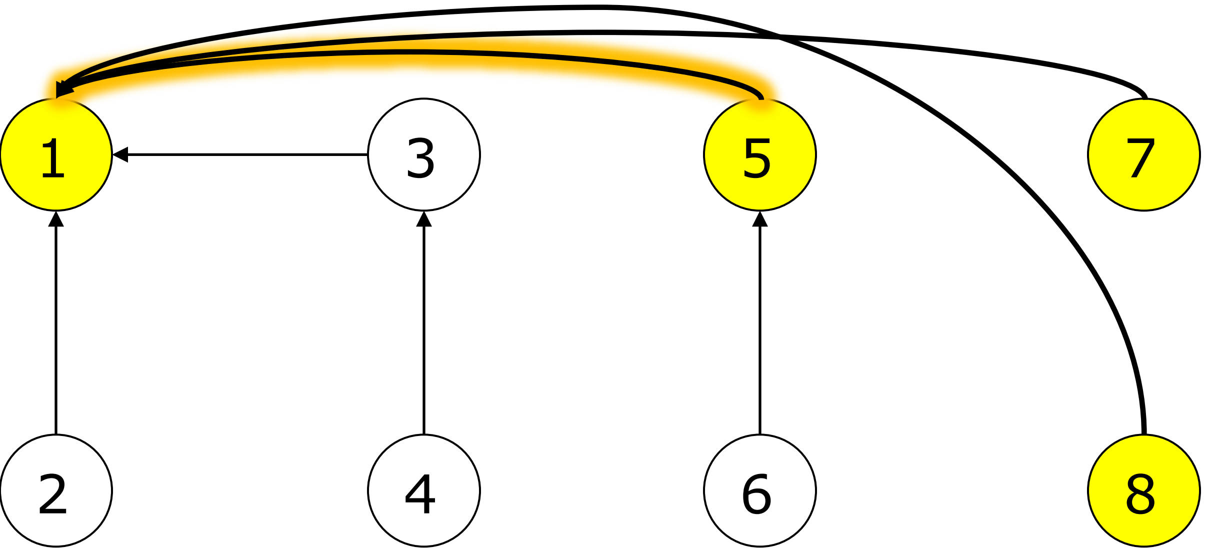

```
Find-Set(x){
  if A[x]<=0 then{
    return x
  }else{
    A[x]=Find-Set(A[x])
    return A[x]
  }
}
```

A:

-3	1	1	3	**1**	5	**1**	**1**

- `Find-Set(8)`
- `A[8]=7 > 0`
- `A[8]=Find-Set(7)=Find-Set(5)=Find-Set(1)=1`

Examples of Find-Set(8)




```
Find-Set(x){
   if A[x]<=0 then{
      return x
   }else{
      A[x]=Find-Set(A[x])
      return A[x]
   }
}
```

A:								
	-3	1	1	3	**1**	5	**1**	**1**

- `Find-Set(8)`
- `A[8]=7 > 0`
- `A[8]=Find-Set(7)=Find-Set(5)=Find-Set(1)=1`
- `return 1`

Summary

- Find-set() may rewrite array A
- Find-set() may reduce the height of the tree, but does not change the rank of any node
- The time complexity of Find-set() is bounded by the height of the tree

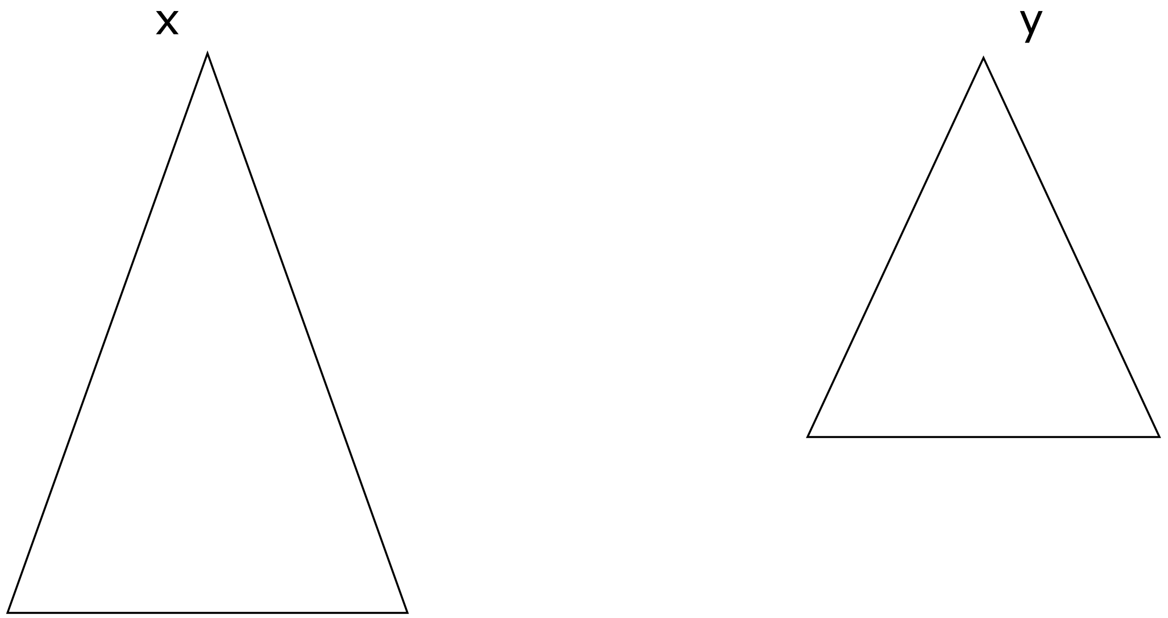
Ira A. Fulton Schools of
Engineering
Arizona State University

Disjoint Sets
Part 4

Ira A. Fulton Schools of Engineering
Arizona State University

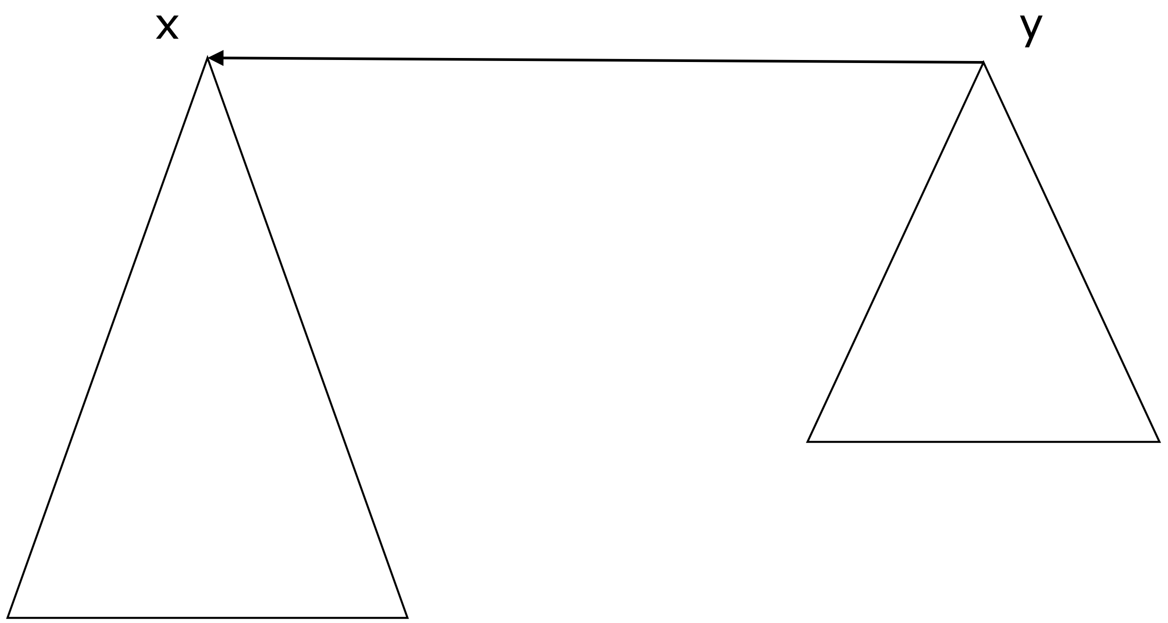
link(x, y), rank of x > rank of y

- if $-A[x] > -A[y]$



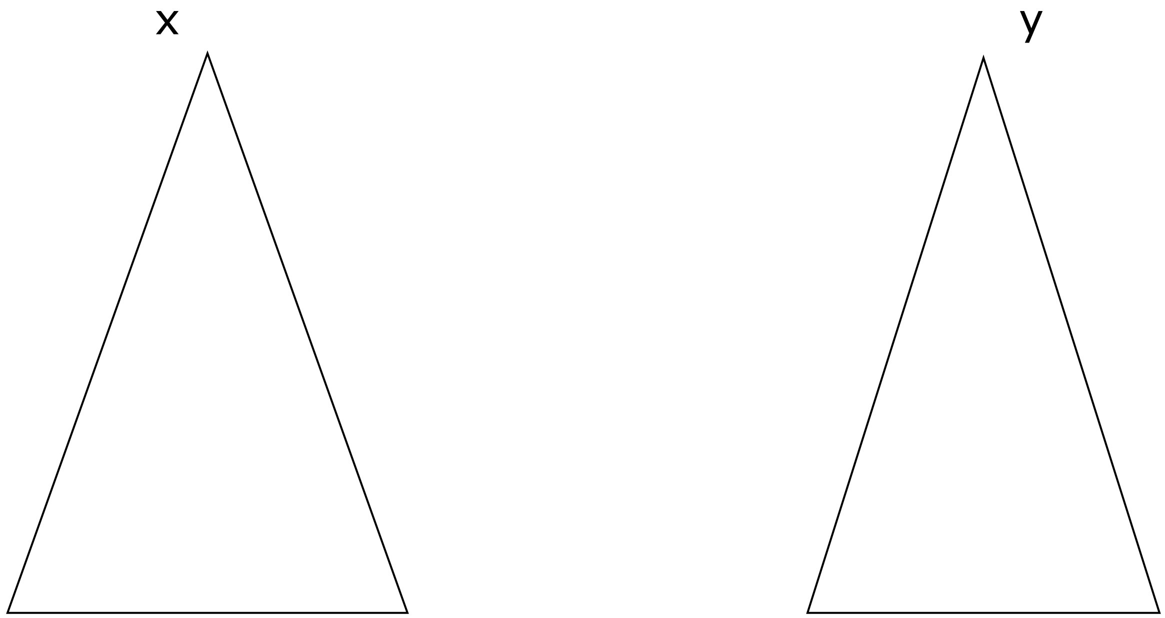
link(x, y), rank of x > rank of y

- if $-A[x] > -A[y]$
- make x the root, rank does not change



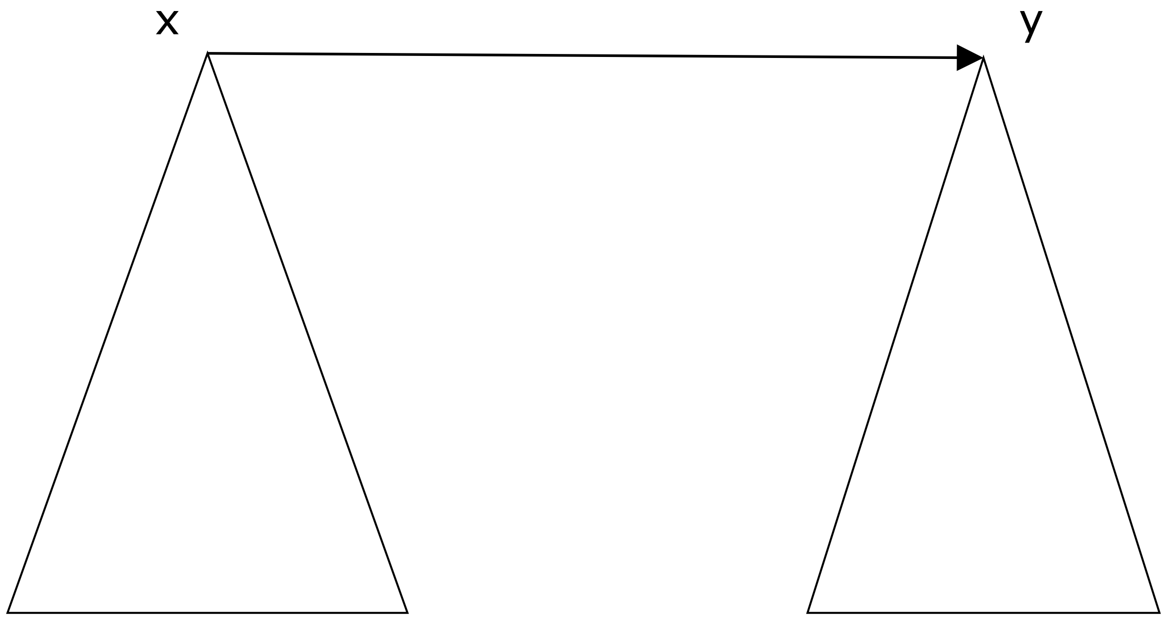
link(x, y), rank of x ≤ rank of y

- if $-A[x] = -A[y]$



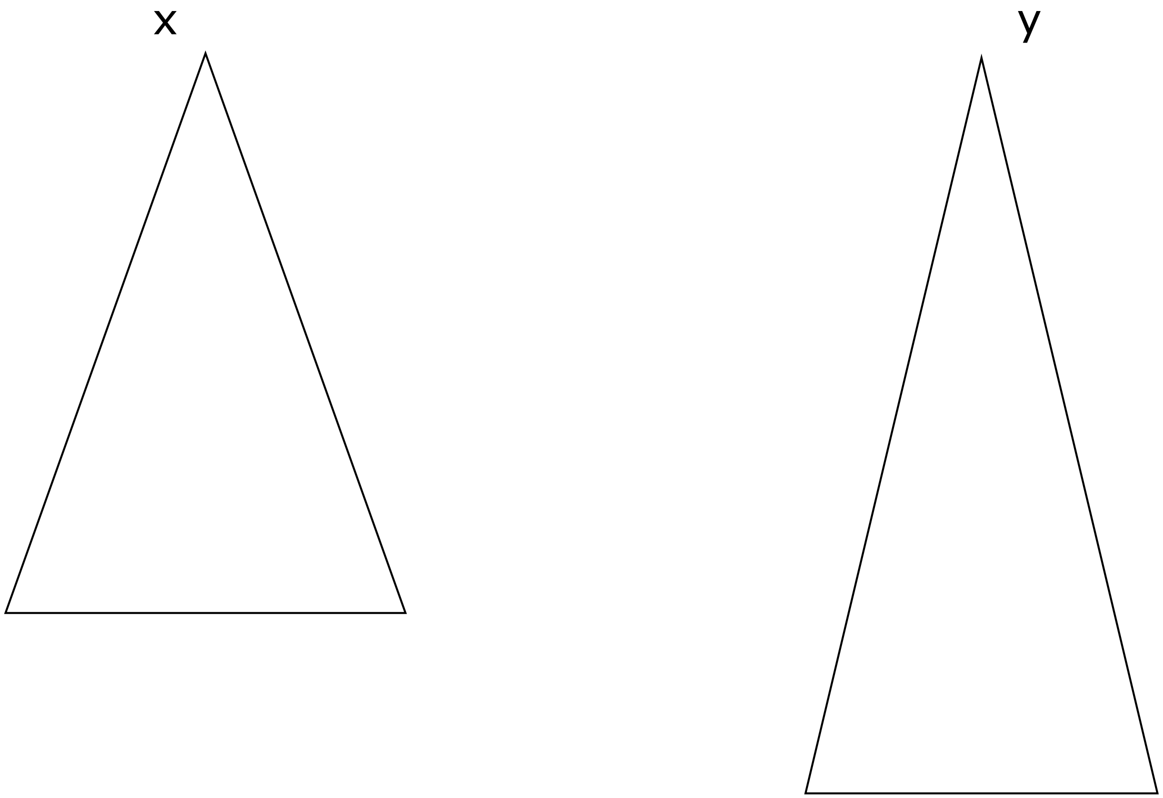
link(x, y), rank of x ≤ rank of y

- if $-A[x] = -A[y]$
- make y the root, rank increased by 1



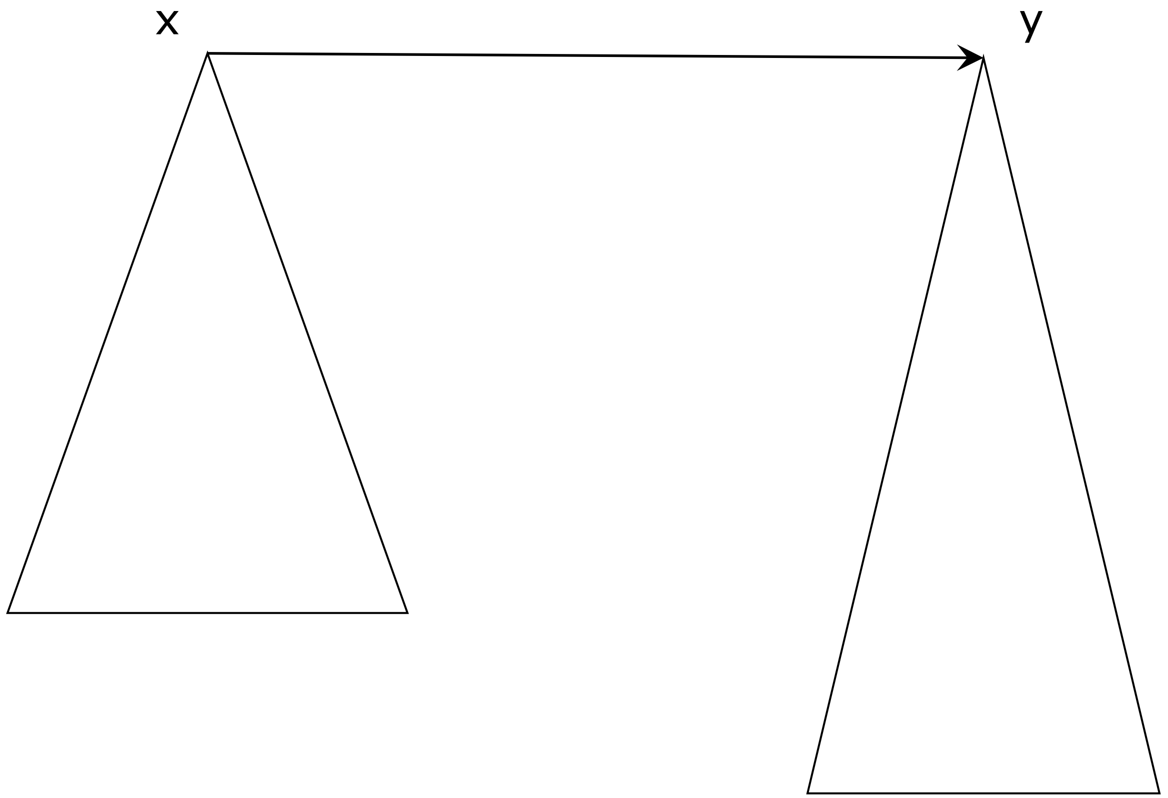
link(x, y), rank of x ≤ rank of y

- if $-A[x] < -A[y]$



link(x, y), rank of x ≤ rank of y

- if $-A[x] < -A[y]$
- make y the root, rank does not change



Summary

- Link (x, y) may change the rank of node y
- Link (x, y) takes O(1) time

Union by rank

- Union(x, y) is accomplished by
  - Link(Find-Set (x), Find-Set (y));

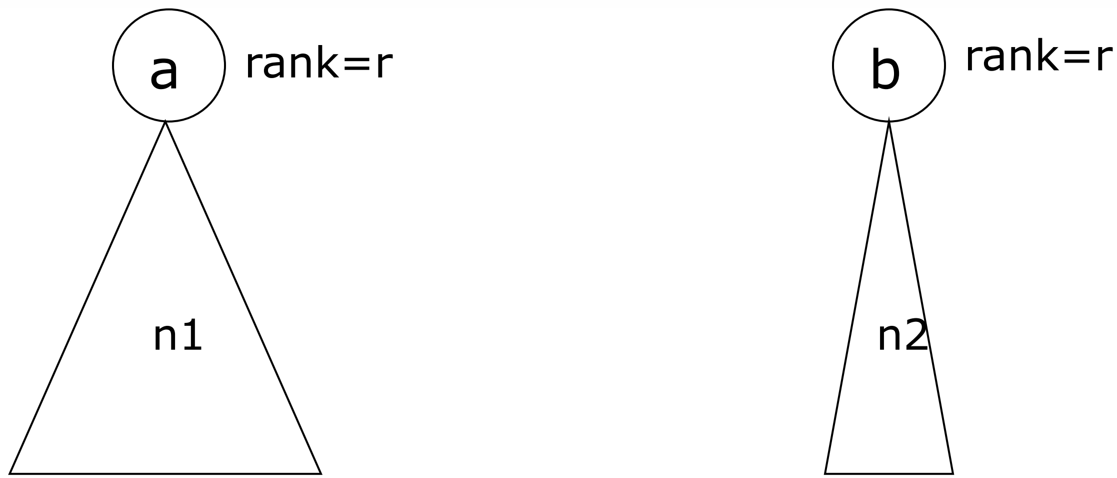
Union by rank

- THEOREM: Let n be the number of nodes in a tree and let r be the rank of the tree. Assuming we are using union by rank, then $n \geq 2^r$.
- Proof by Induction. This is true when r=0 (n=1).
- Assume that we are taking the union of two trees with the same rank r. By the assumption, the number of nodes in either tree is at least 2^r. The rank of the new tree is r+1, and the number of nodes in the new tree is at least $2^r+2^r = 2^{r+1}$. Hence the inequality remains true.

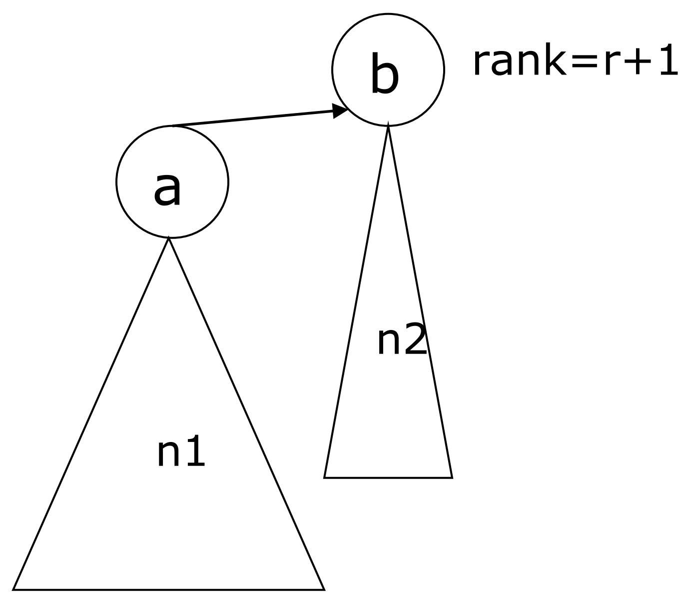
Illustration of the proof

$n1 \geq 2^r$

$n2 \geq 2^r$



$n1+n2 \geq 2^r + 2^r = 2^{r+1}$



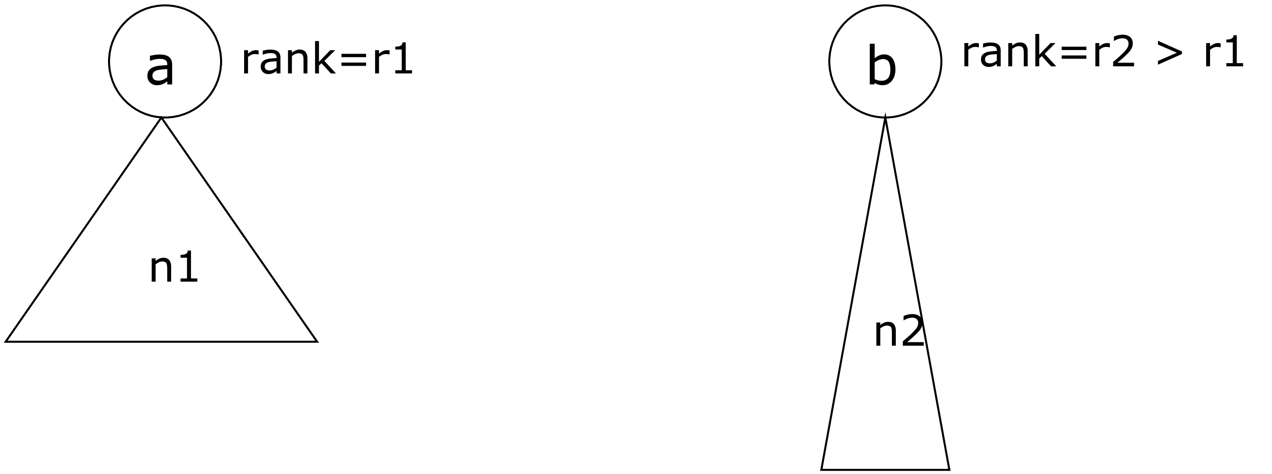
Union by rank

- Proof. Continued…
- Assume that we are taking the union of two trees with different ranks. One tree has n1 nodes and rank r1, the other has n2 nodes and rank r2. Without loss of generality, assume that r1 < r2. The new tree has rank r=r2. The number of nodes in the new tree is $n1+n2 \geq n2 \geq 2^{r2} = 2^{r}$. Hence the inequality remains true.

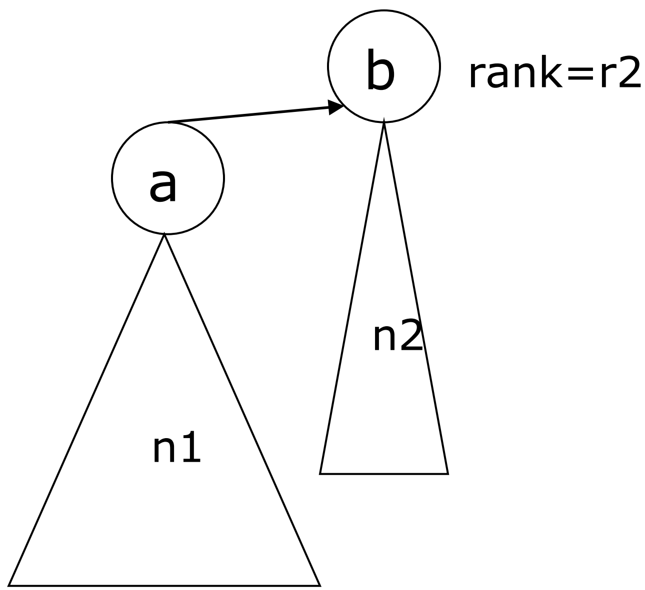
Union by rank

$n1 \geq 2^{r1}$

$n2 \geq 2^{r2}$



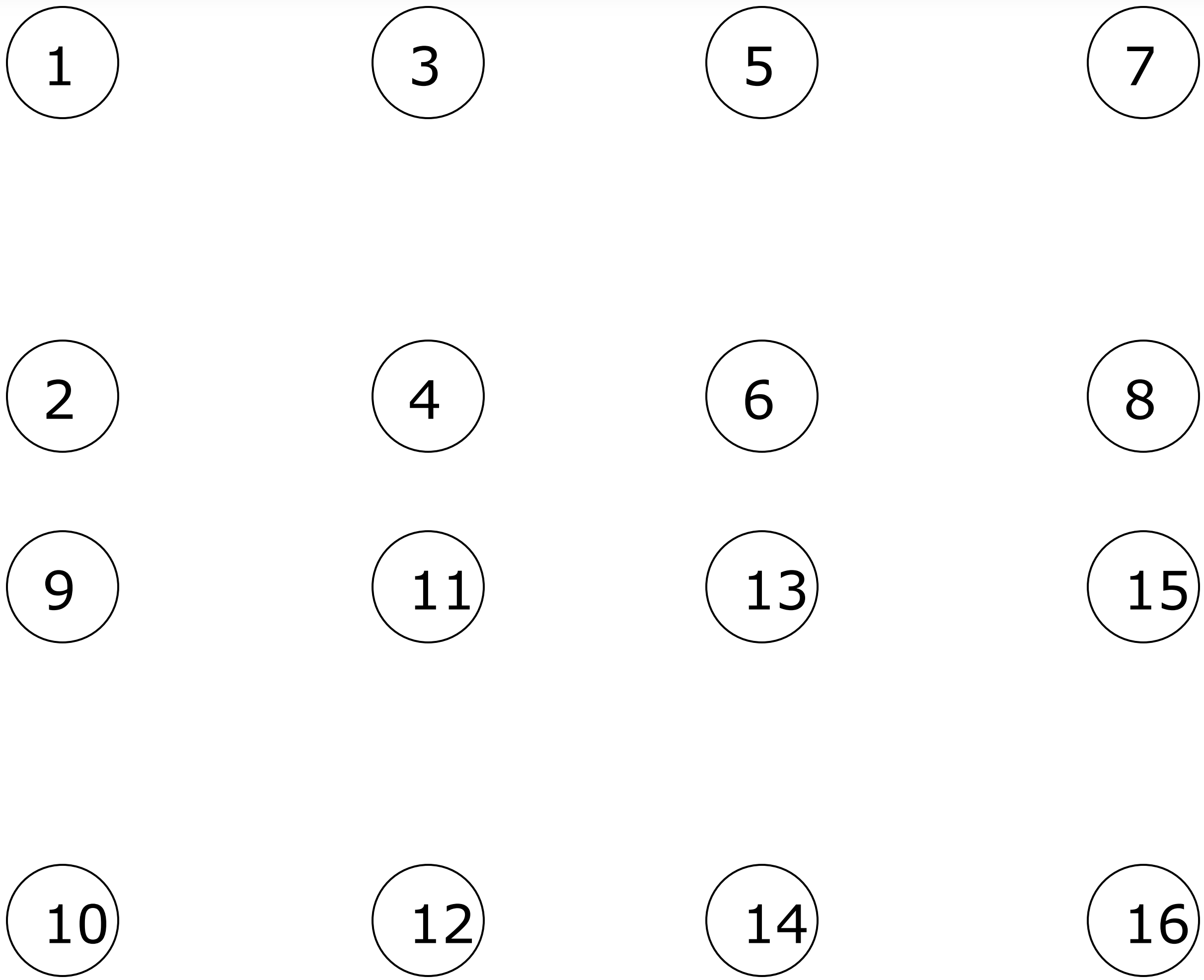
$n1+n2 \geq n2 \geq 2^{r2}$



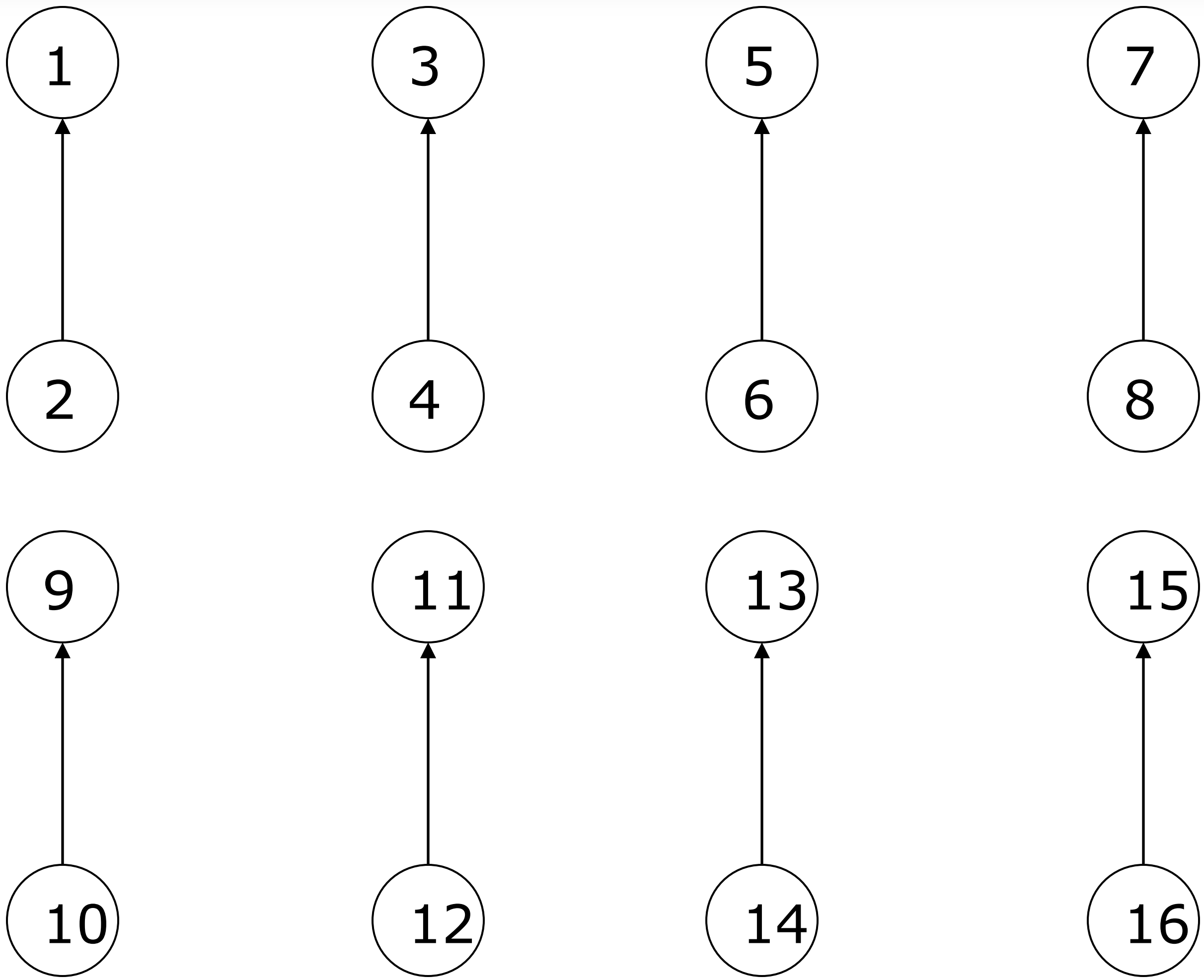
Union by rank

- **THEOREM: Let n= 2^r be the number of elements. There exists a sequence of n Make-Set operations followed by n-1 union operations such that the resulting tree has height equal to r.**
- **Proof. Perform n/2 union operations to have n/2 trees each with height=1. Then perform n/4 unions to have n/4 trees each with height=2. Continue this process this process, we will have a single tree with height=r.**

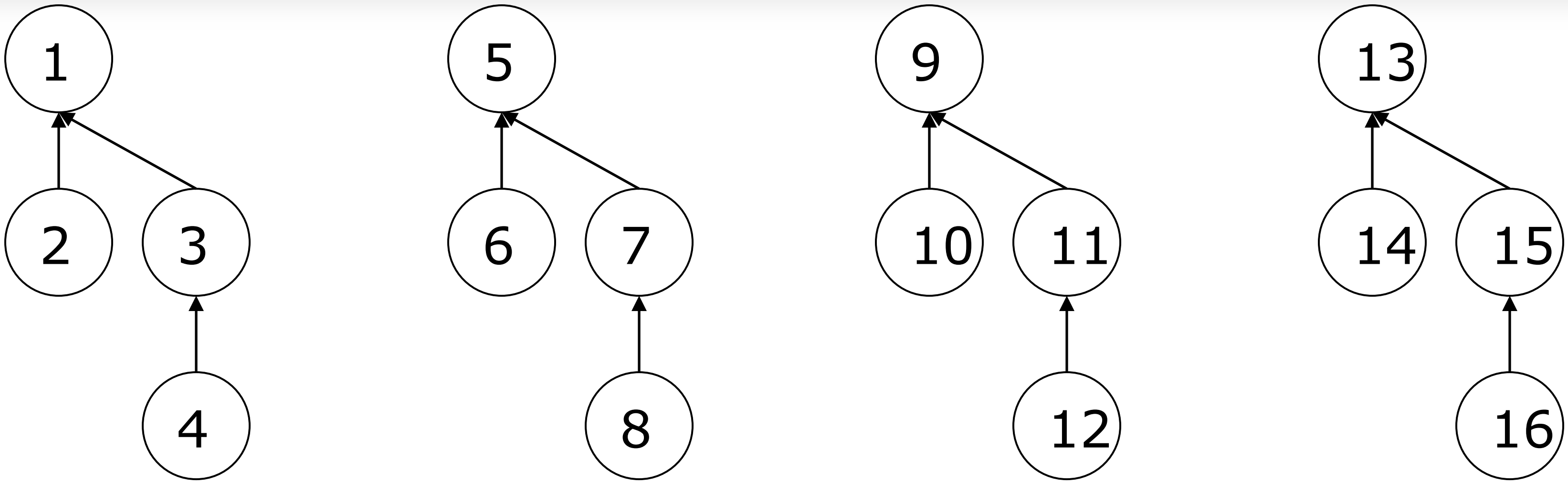
Example sequences of unions



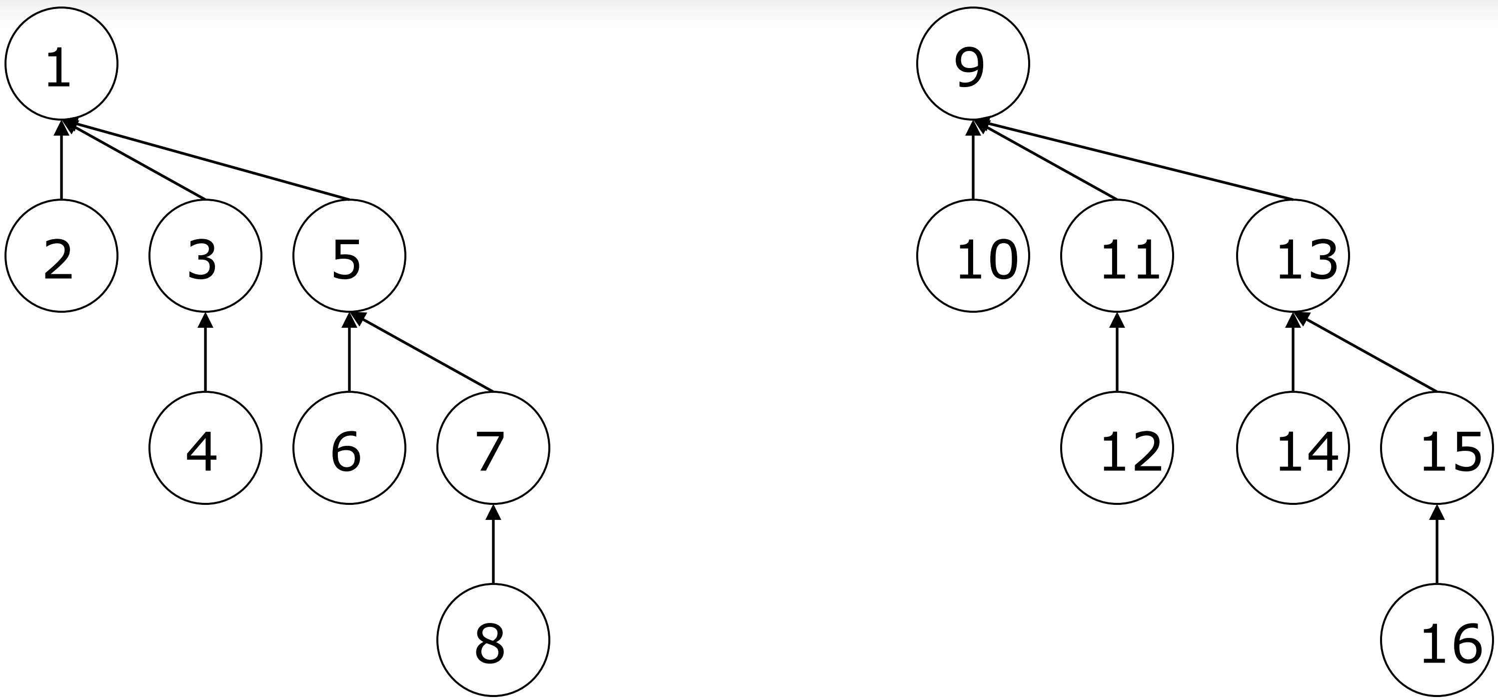
Example sequences of unions



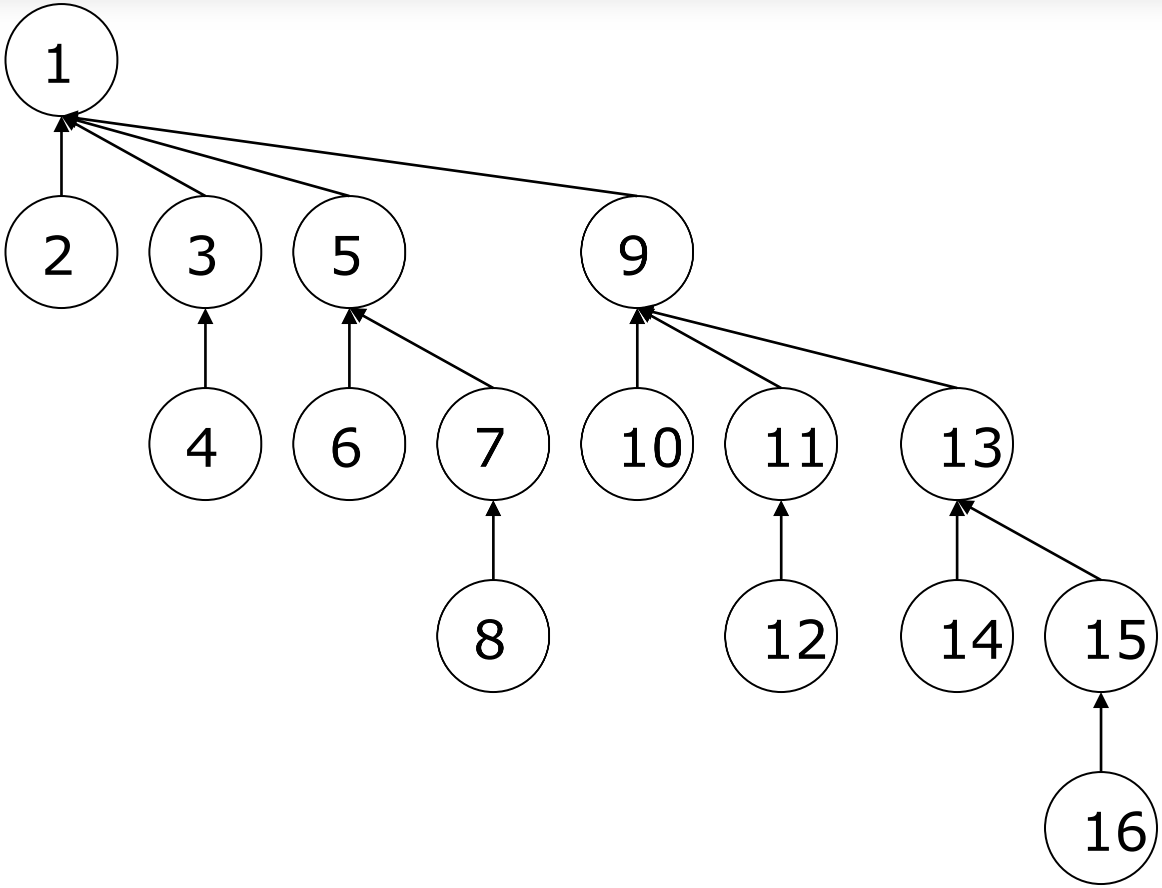
Example sequences of unions



Example sequences of unions



Example sequences of unions



Union by rank

- Running time of Find-set() is $O(\log n)$
- Running time of Link() is $O(1)$
- Running time of Union() is $O(\log n)$
- Let m be the total number of operations, including n Make-Set() operations. The running time of these m operations is bounded by $O(m \log n)$.
- We can do MUCH better analysis.

Union by rank

- Let m be the total number of operations, including n Make-Set() operations. The running time of these m operations is bounded by $O\big(m\,\alpha(m,n)\big)$, where $\alpha(m,n)$ is the inverse Ackermann function.
- For fixed m, $\alpha(m,n)$ increases with n, and goes to ∞ when n goes to ∞.
- For fixed n, $\alpha(m,n)$ increases with m, and goes to ∞ when m goes to ∞.
- For practical values of m and n, $\alpha(m,n) \le 4$.

Ira A. Fulton Schools of Engineering

Arizona State University