Lecture 13

Disjoint-Set Data Structure (contd.)

Source: Introduction to Algorithms, CLRS

Disjoint-Sets as Trees

Disjoint-Sets as Trees

Idea: We maintain the dynamic disjoint sets in the following way:

Disjoint-Sets as Trees

Idea: We maintain the dynamic disjoint sets in the following way:

- Keep sets as rooted trees.

Disjoint-Sets as Trees

Idea: We maintain the dynamic disjoint sets in the following way:

- Keep sets as rooted trees.
- Each node points to its parent.

Disjoint-Sets as Trees

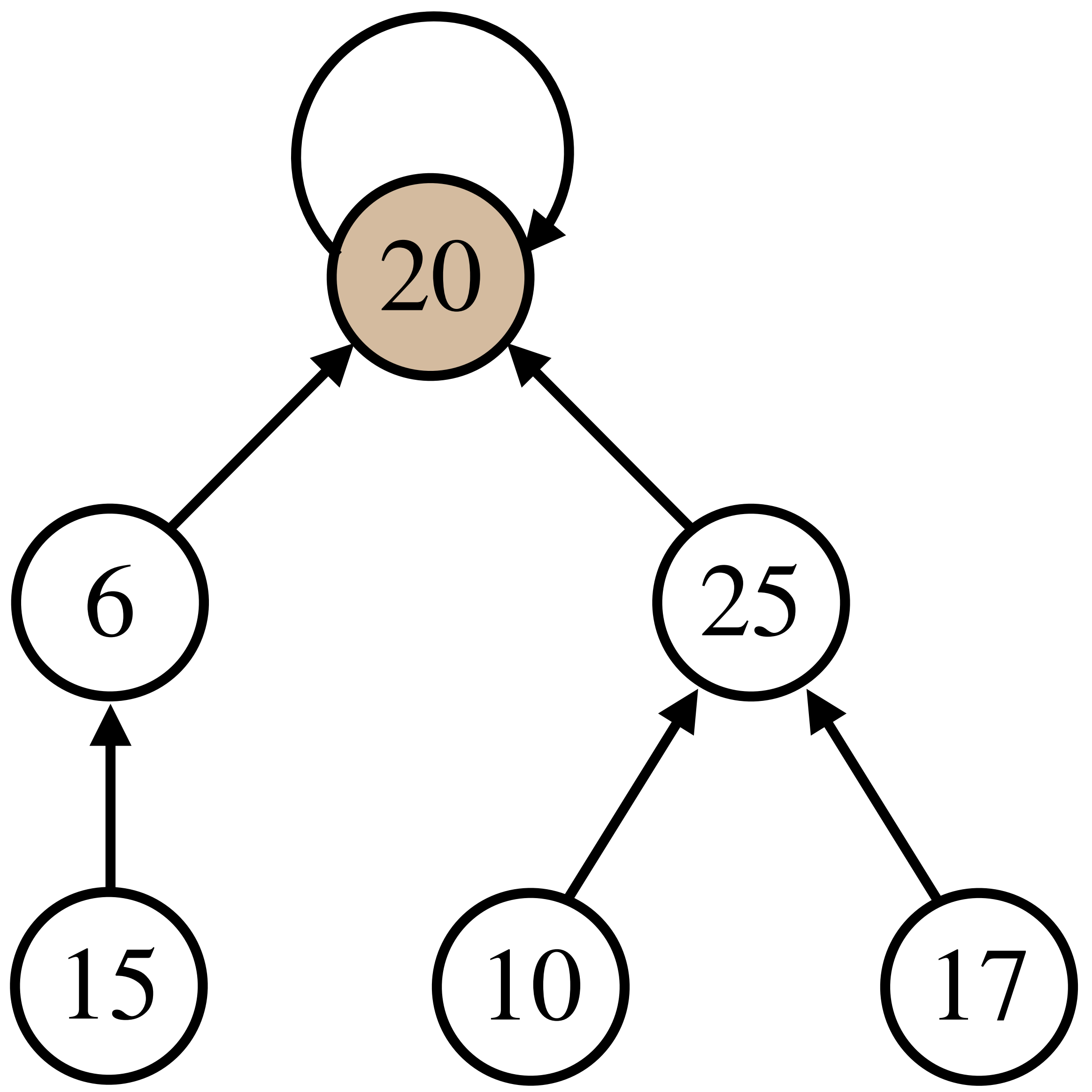
Idea: We maintain the dynamic disjoint sets in the following way:

- Keep sets as rooted trees.
- Each node points to its parent.
- Root is its own parent and the representative of the set.

Disjoint-Sets as Trees

Idea: We maintain the dynamic disjoint sets in the following way:

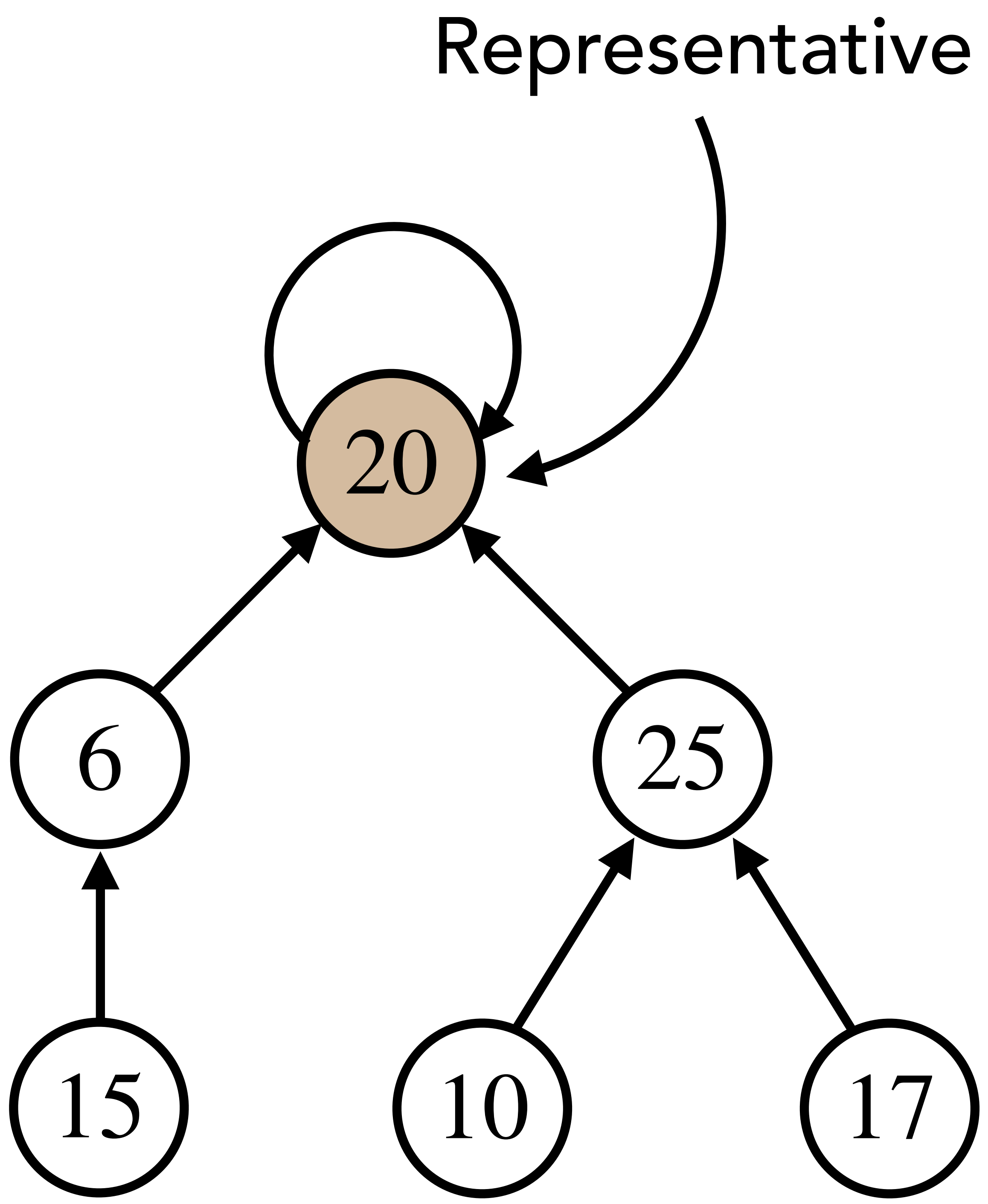
- Keep sets as rooted trees.
- Each node points to its parent.
- Root is its own parent and the representative of the set.



Disjoint-Sets as Trees

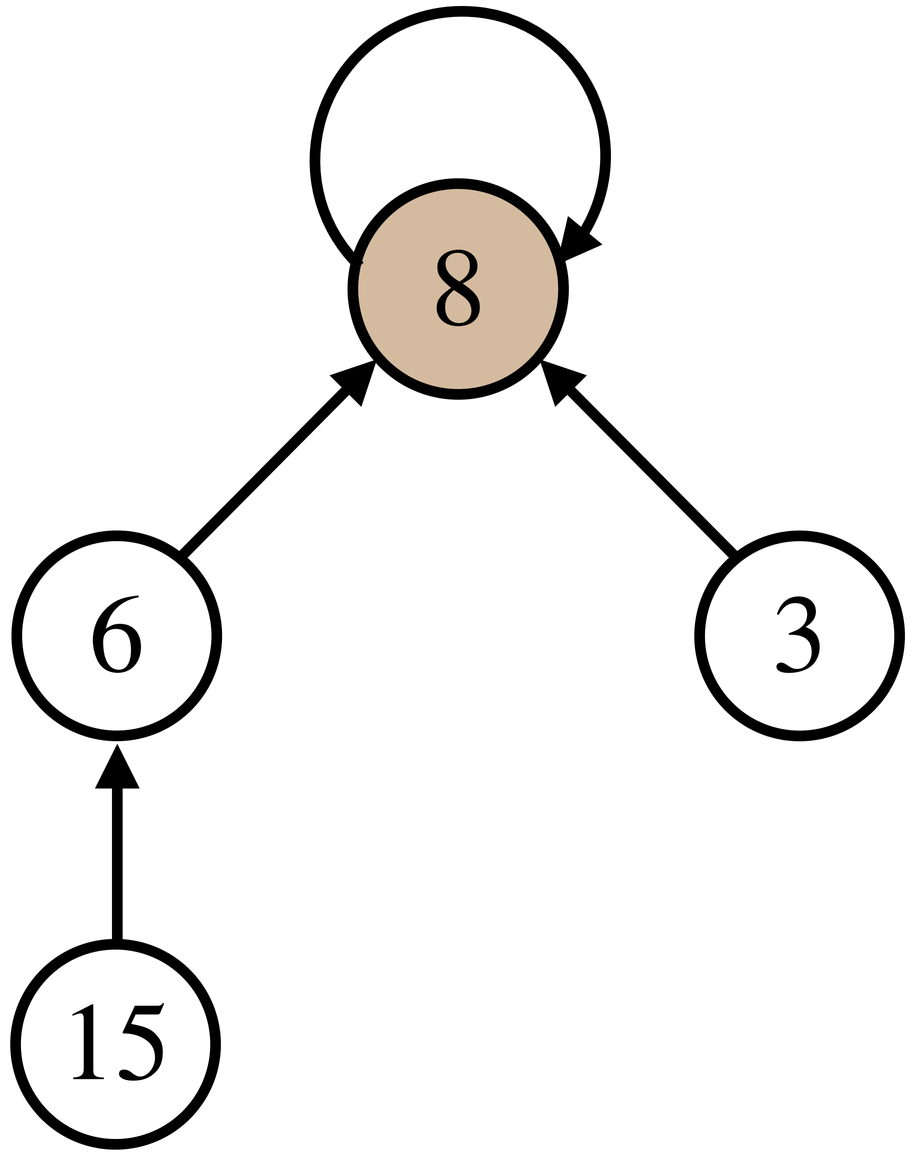
Idea: We maintain the dynamic disjoint sets in the following way:

- Keep sets as rooted trees.
- Each node points to its parent.
- Root is its own parent and the representative of the set.

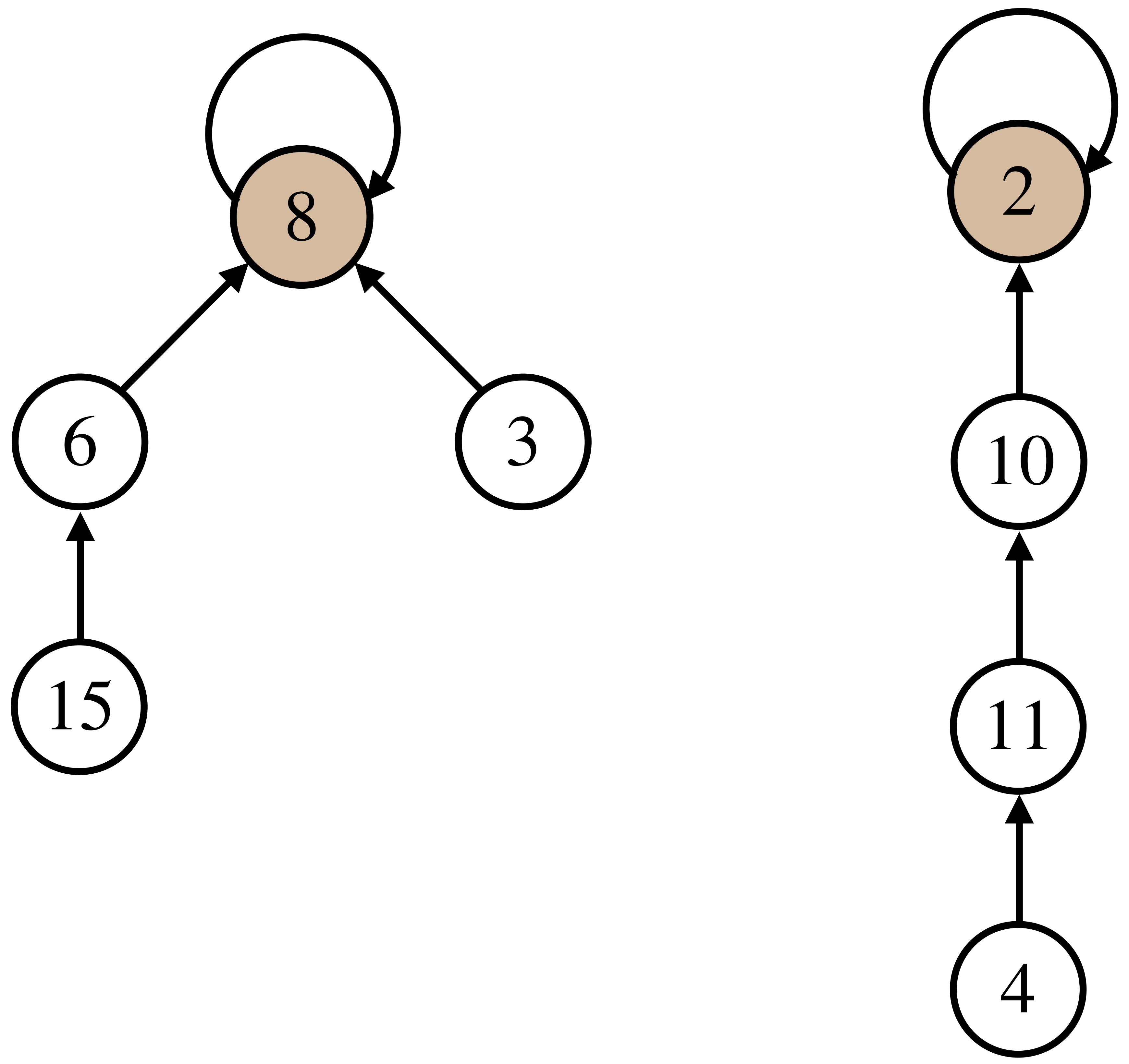


Union on Disjoint-Sets as Trees

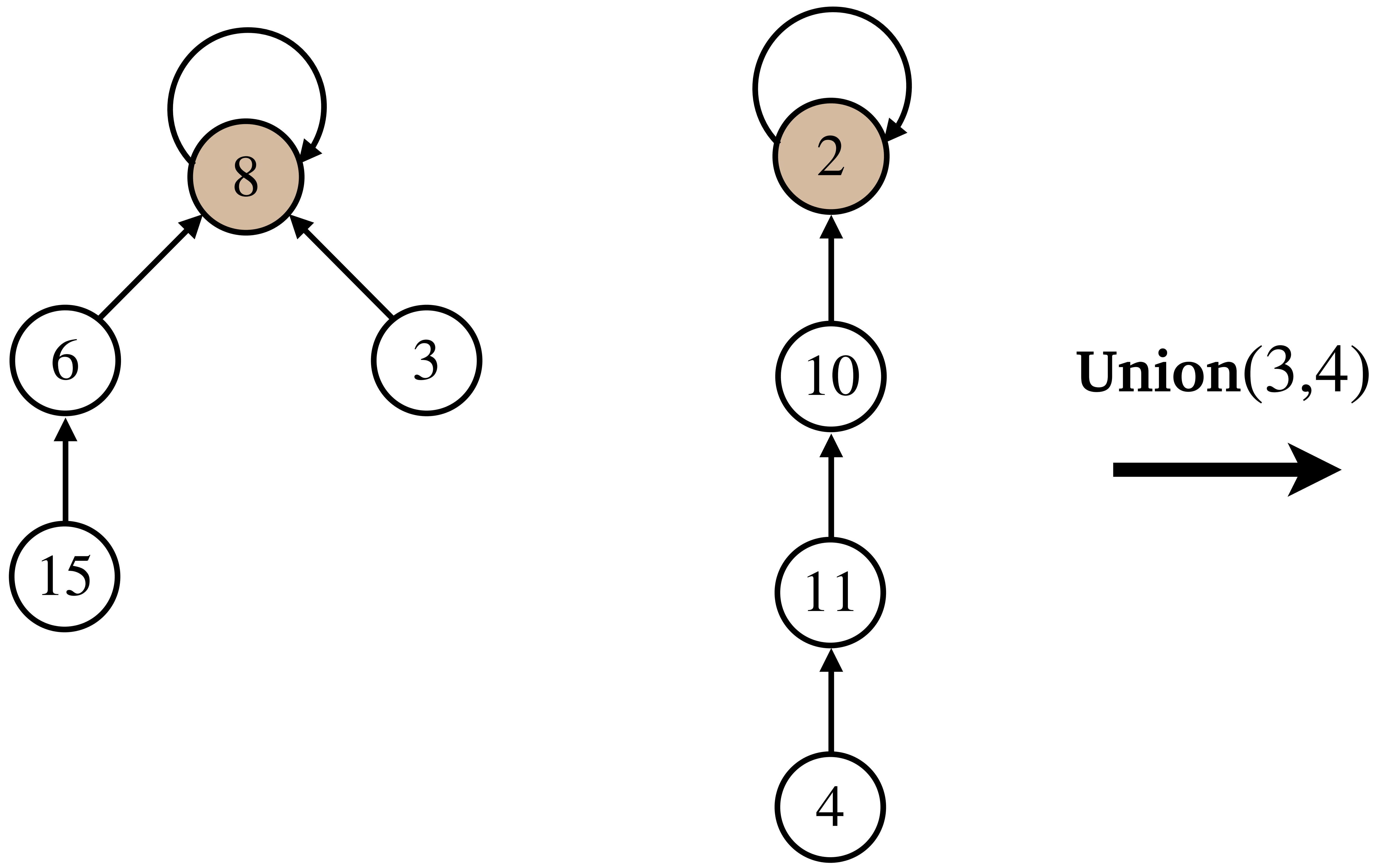
Union on Disjoint-Sets as Trees



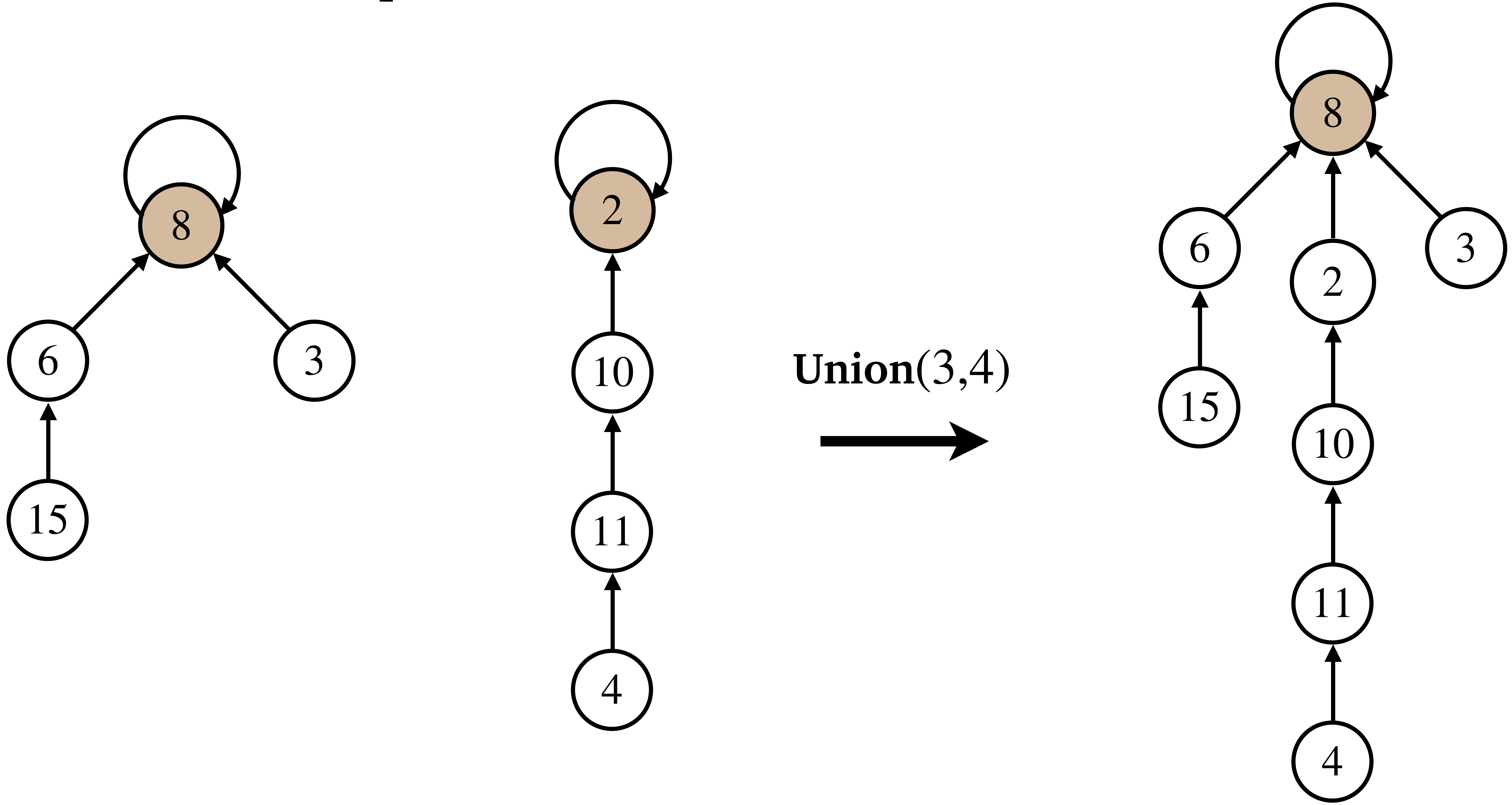
Union on Disjoint-Sets as Trees



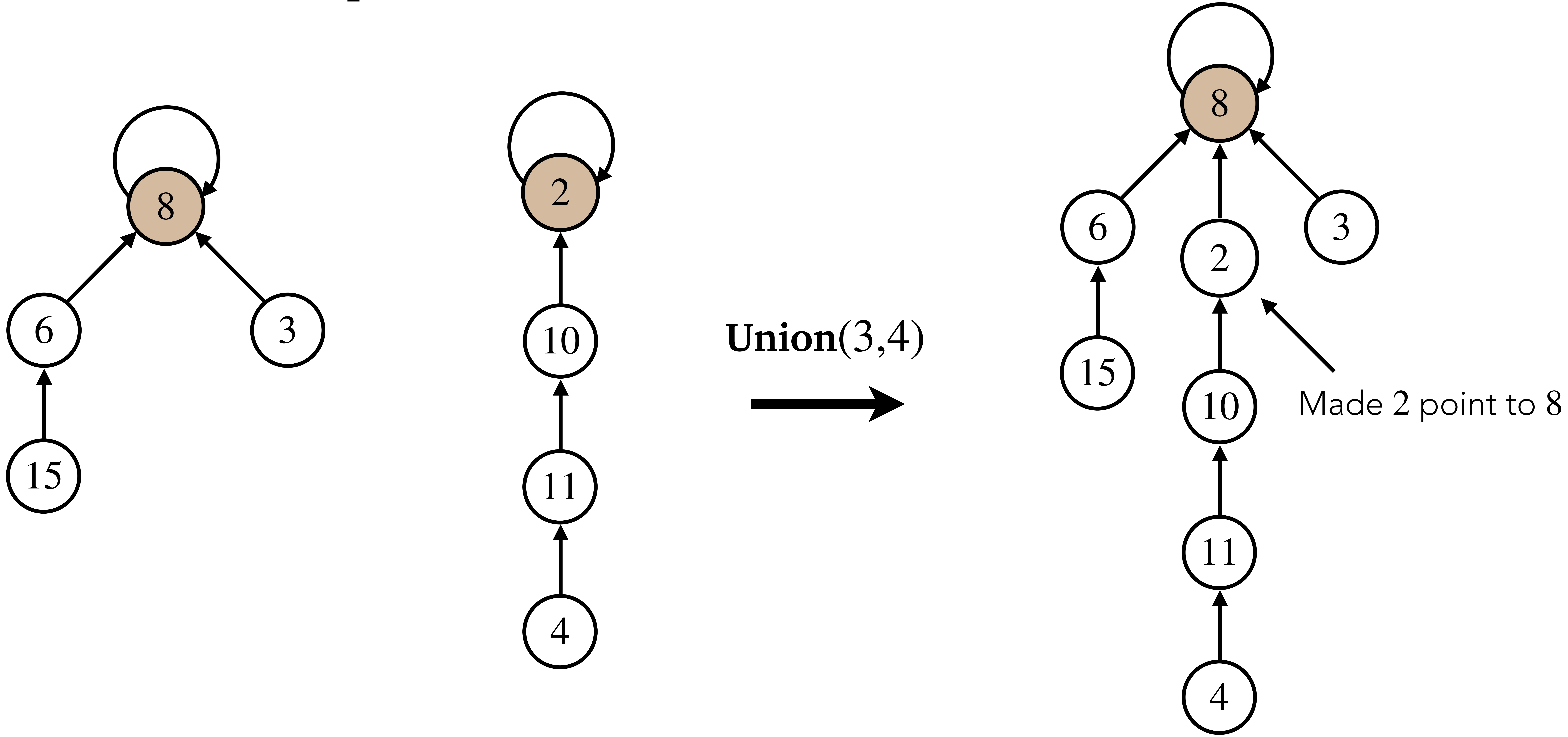
Union on Disjoint-Sets as Trees



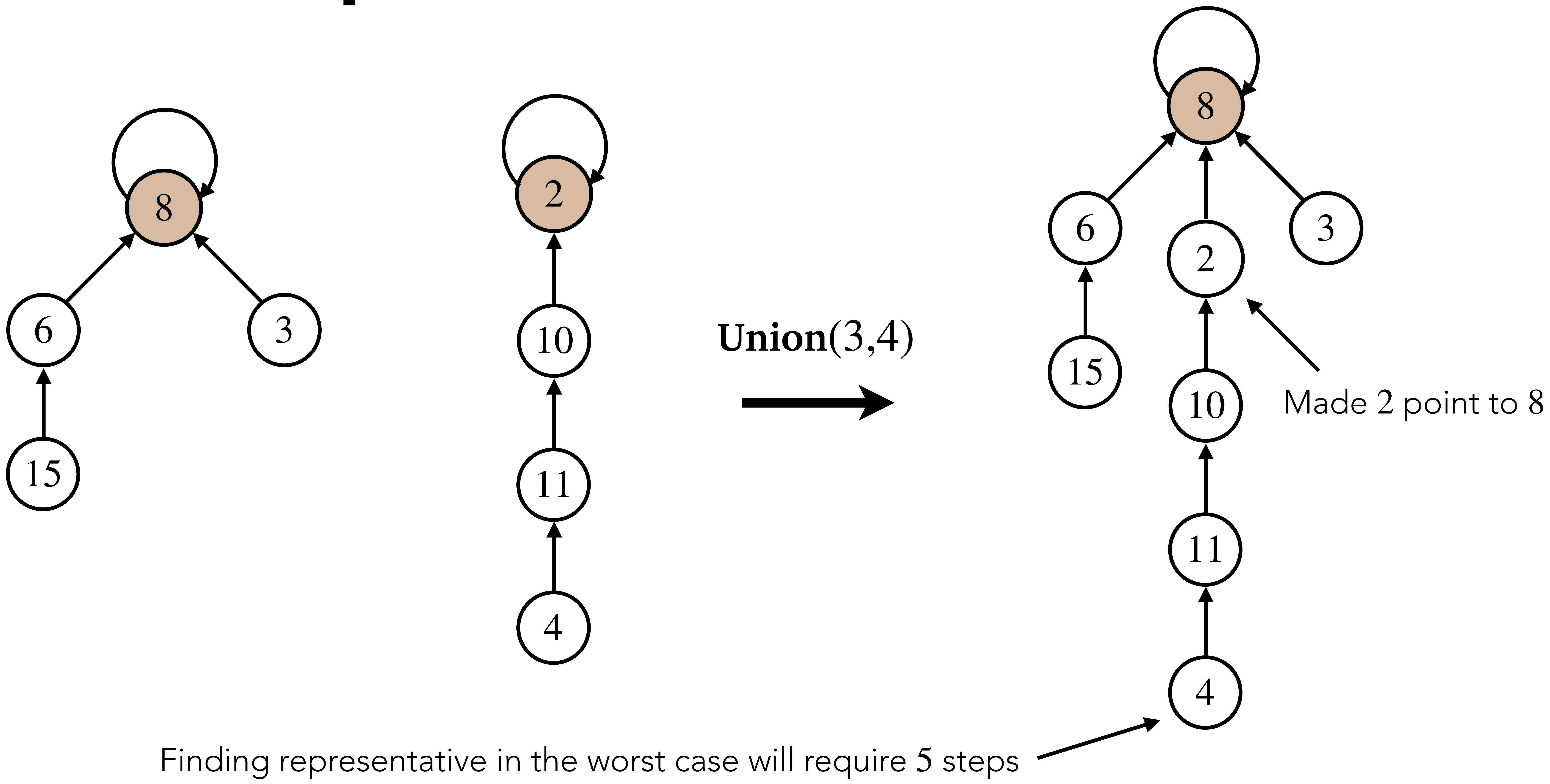
Union on Disjoint-Sets as Trees



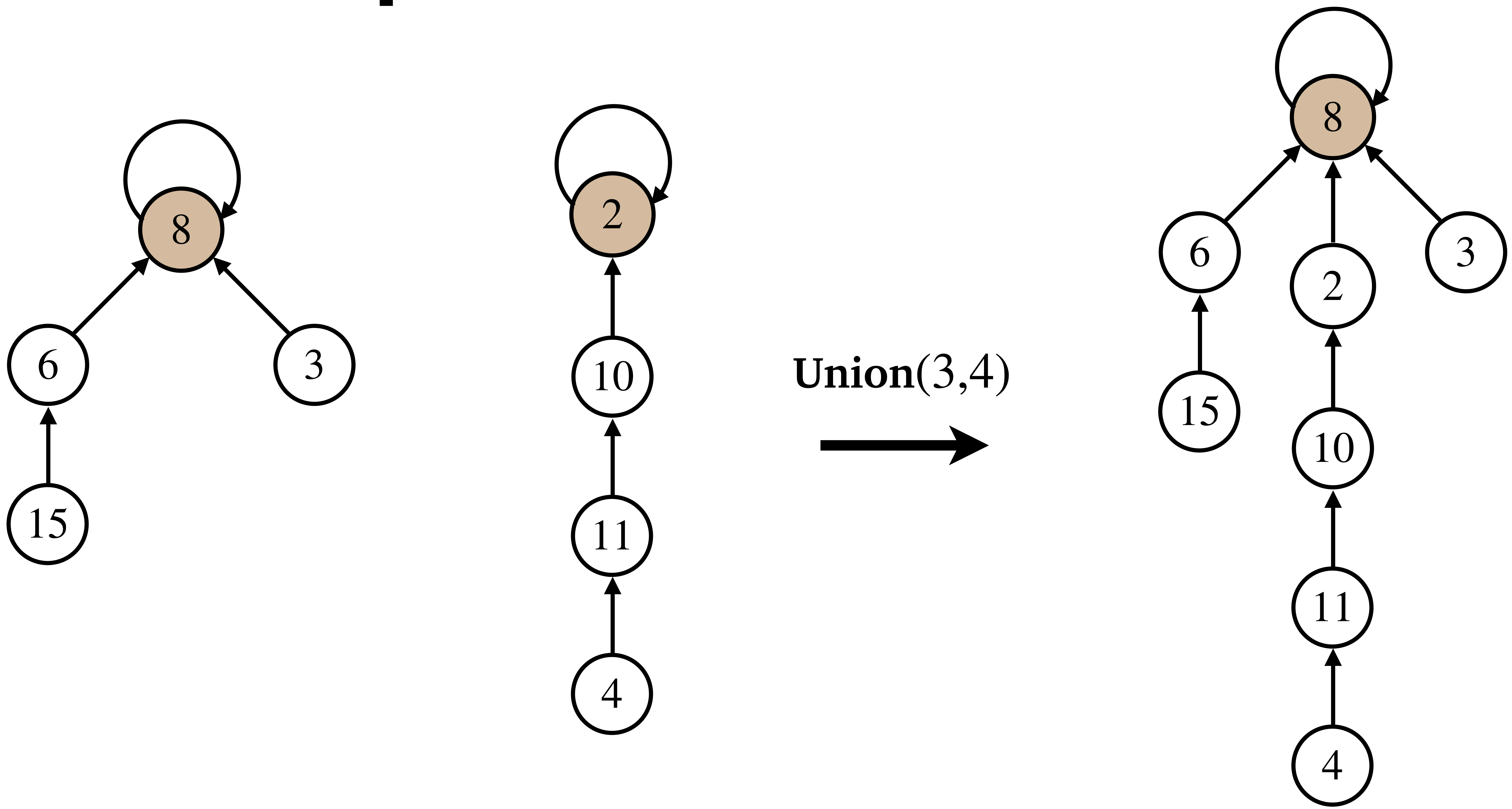
Union on Disjoint-Sets as Trees



Union on Disjoint-Sets as Trees

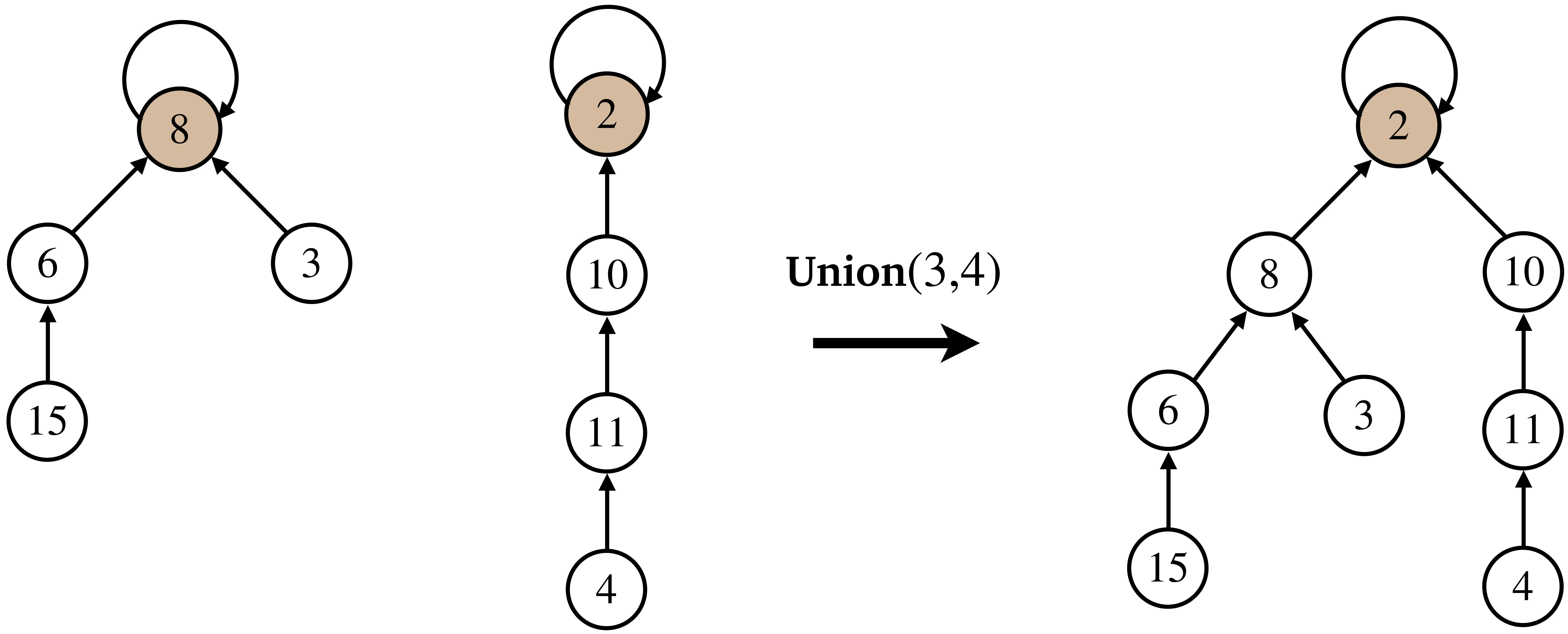


Union on Disjoint-Sets as Trees

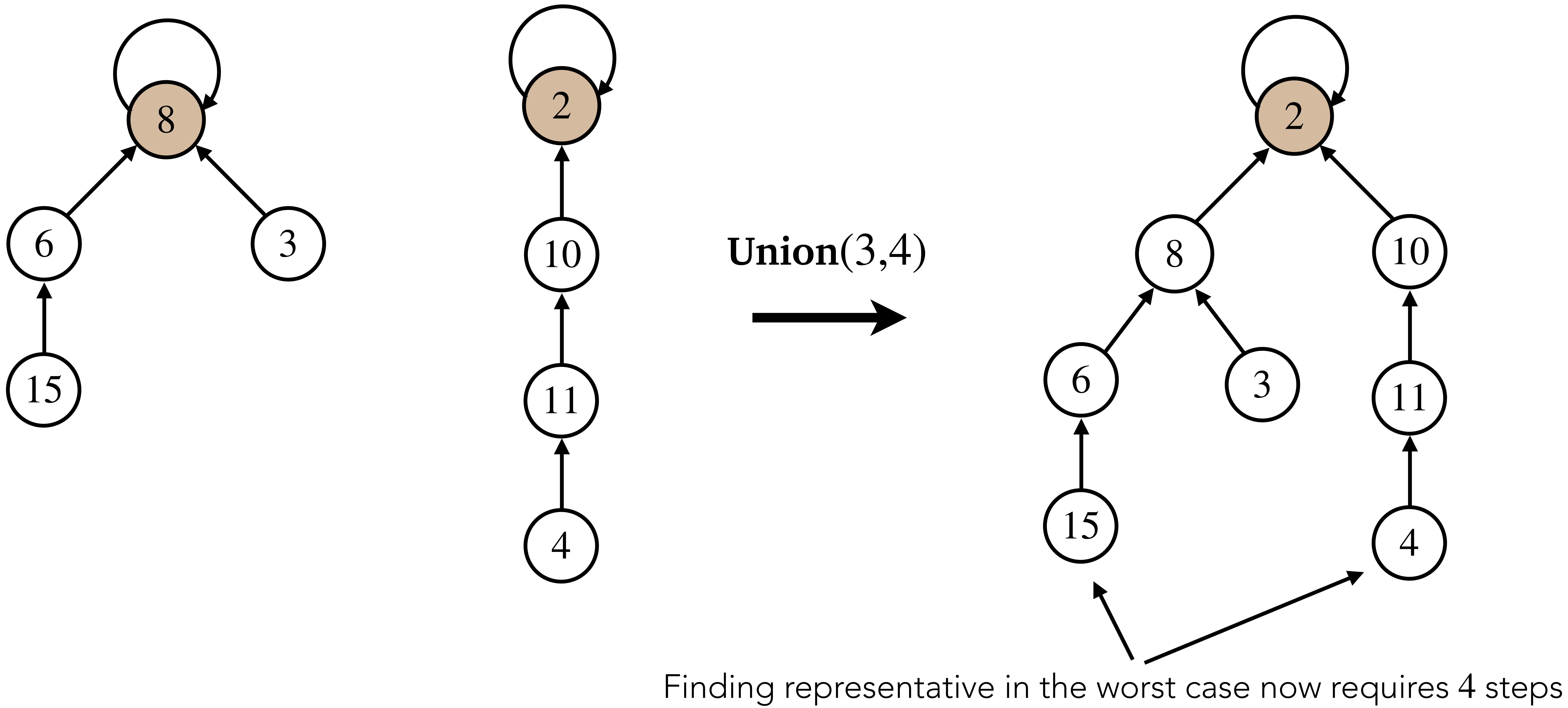


Shouldn't root with smaller height point to root with larger height?

Union on Disjoint-Sets as Trees



Union on Disjoint-Sets as Trees



Union on Disjoint-Sets as Trees using Rank

Union on Disjoint-Sets as Trees using Rank

Idea:

Union on Disjoint-Sets as Trees using Rank

Idea:

- For every node keep track of its rank which denotes its height in the tree.

Union on Disjoint-Sets as Trees using Rank

Idea:

- For every node keep track of its rank which denotes its height in the tree.
- During **Union:**

Union on Disjoint-Sets as Trees using Rank

Idea:

- For every node keep track of its rank which denotes its height in the tree.
- During **Union:**
  - Root with smaller rank will point to root with larger rank.

Union on Disjoint-Sets as Trees using Rank

Idea:

- For every node keep track of its rank which denotes its height in the tree.
- During **Union:**
  - Root with smaller rank will point to root with larger rank.
  - If roots have the same rank then anyone can point to the other one and rank of the new

Union on Disjoint-Sets as Trees using Rank

Idea:

- For every node keep track of its rank which denotes its height in the tree.
- During **Union:**
  - Root with smaller rank will point to root with larger rank.
  - If roots have the same rank then anyone can point to the other one and rank of the new representative will increase by one.

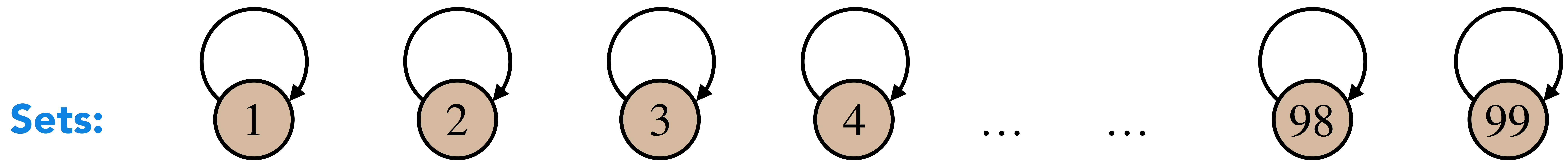
Union on Disjoint-Sets as Trees using Rank

Rank starts with 0

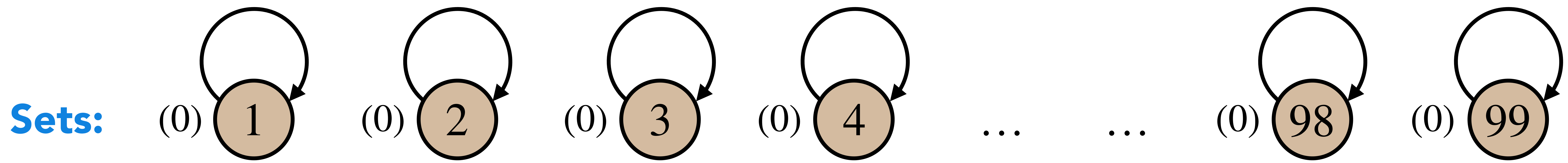
Idea:

- For every node keep track of its rank which denotes its height in the tree.
- During **Union:**
  - Root with smaller rank will point to root with larger rank.
  - If roots have the same rank then anyone can point to the other one and rank of the new representative will increase by one.

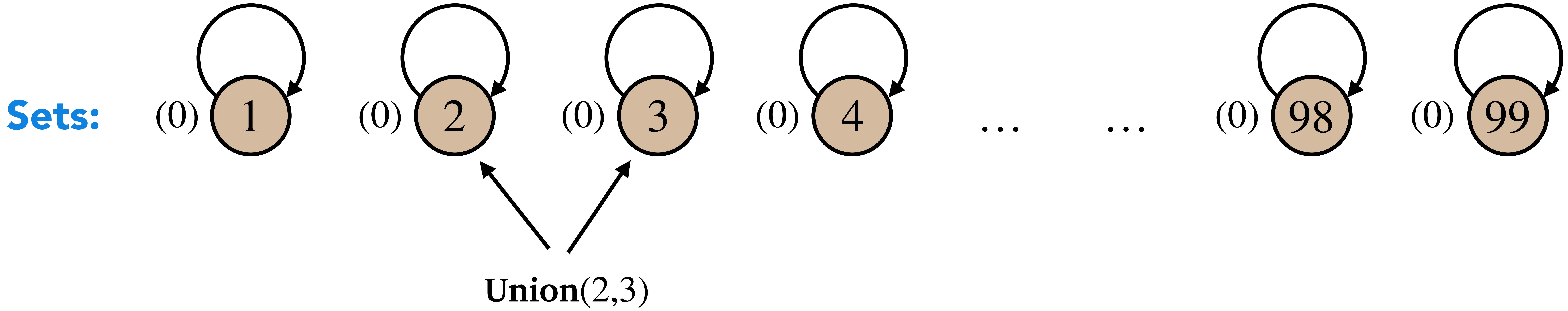
Union on Disjoint-Sets as Trees using Rank



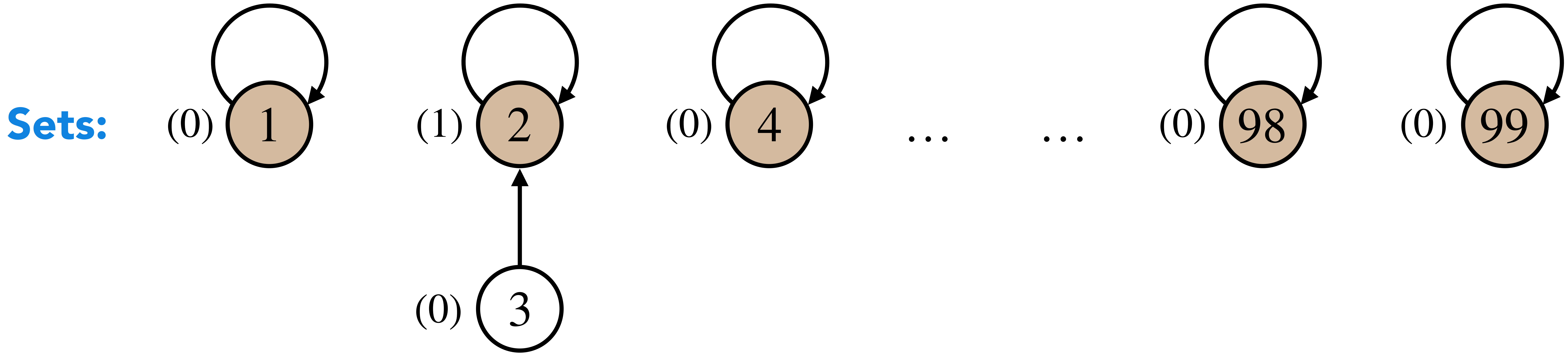
Union on Disjoint-Sets as Trees using Rank



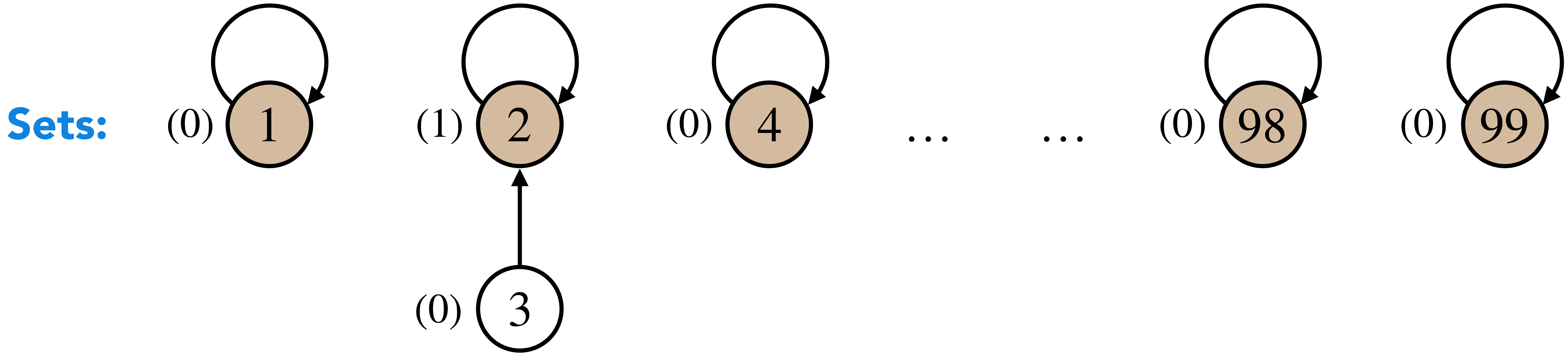
Union on Disjoint-Sets as Trees using Rank



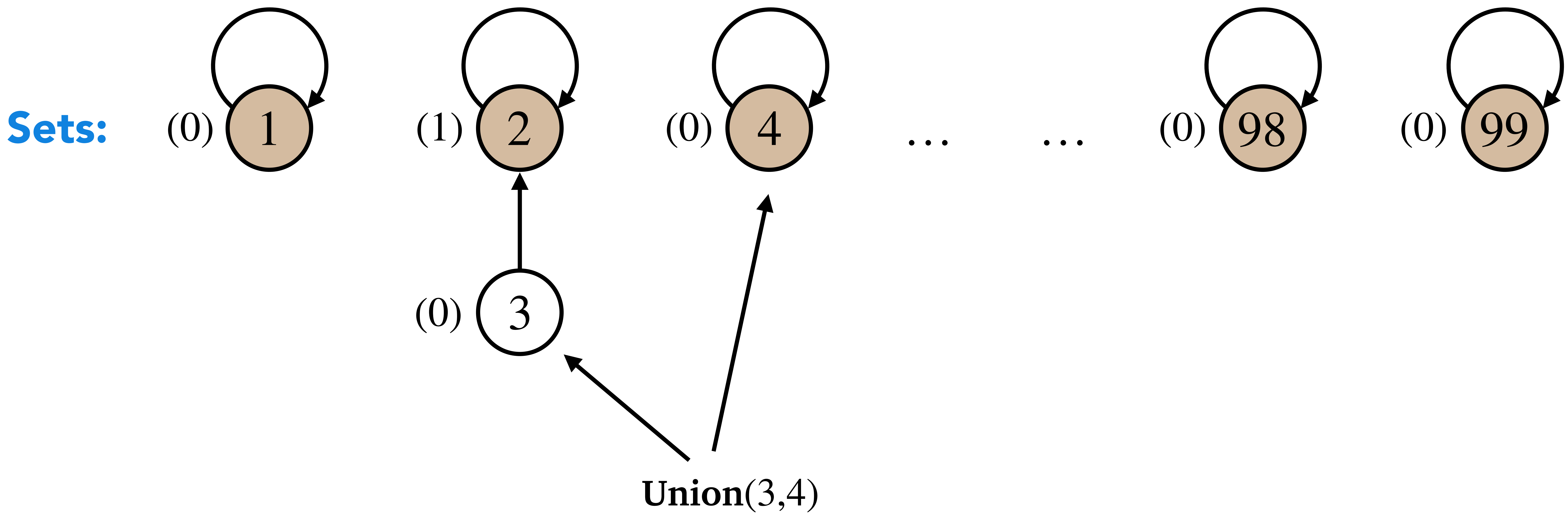
Union on Disjoint-Sets as Trees using Rank



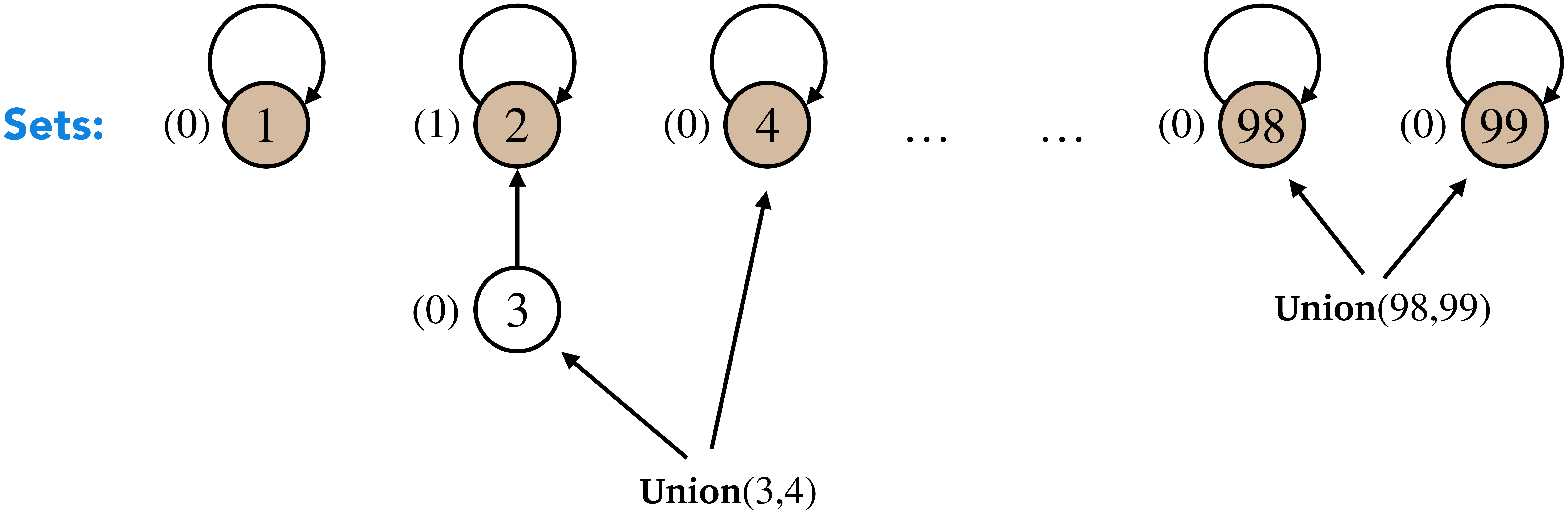
Union on Disjoint-Sets as Trees using Rank



Union on Disjoint-Sets as Trees using Rank

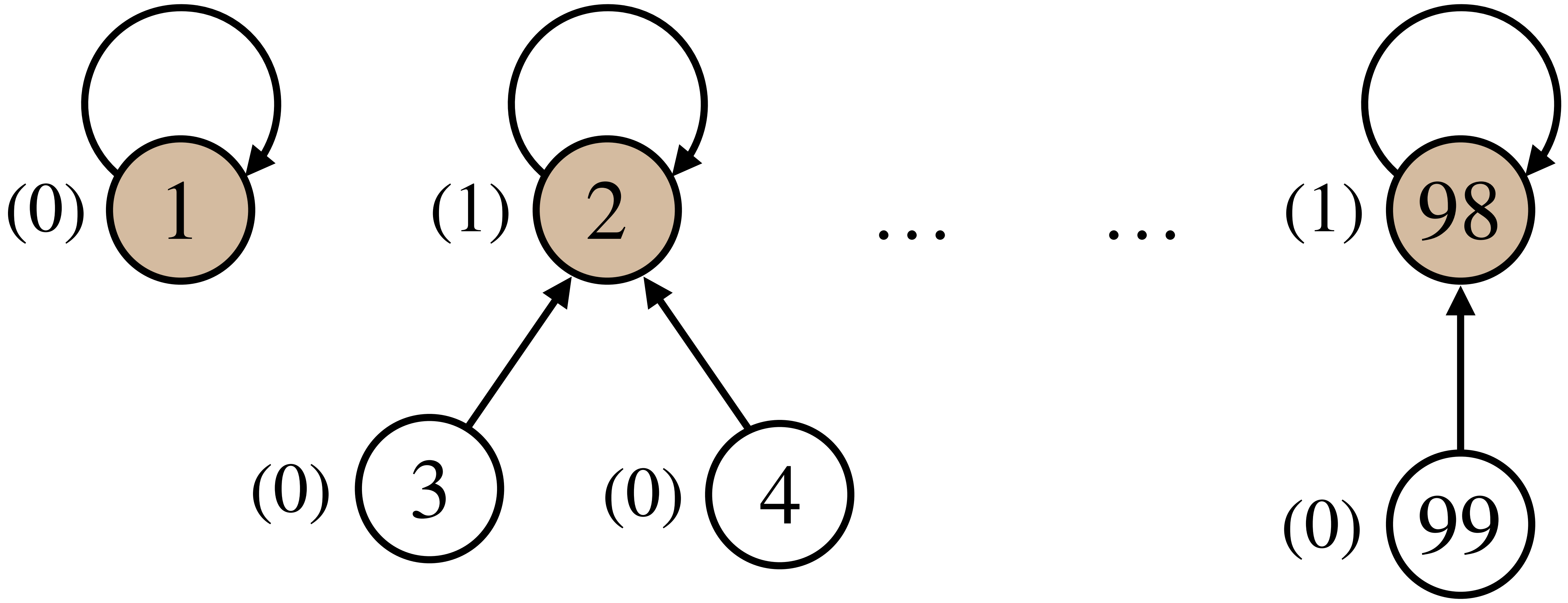


Union on Disjoint-Sets as Trees using Rank

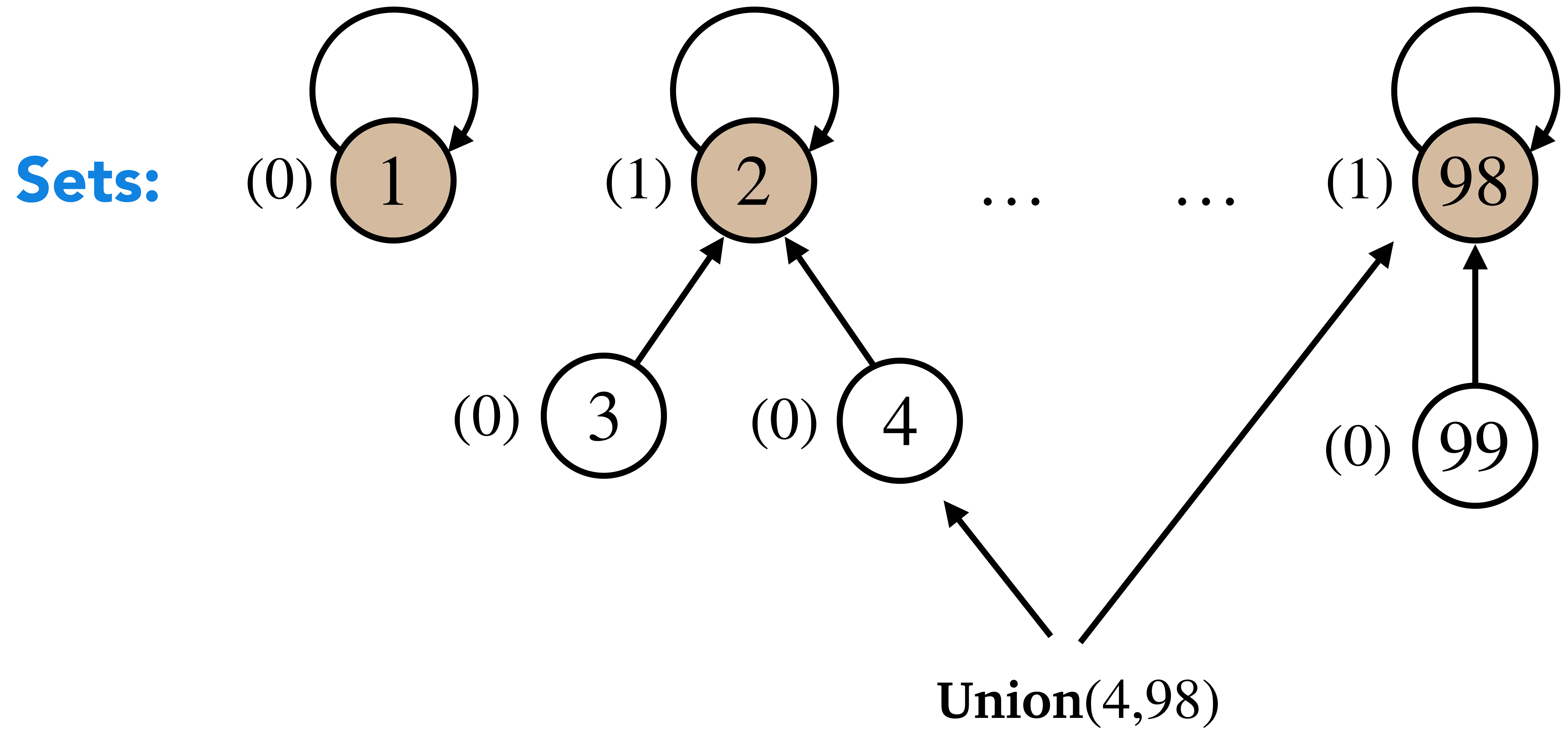


Union on Disjoint-Sets as Trees using Rank

Sets:

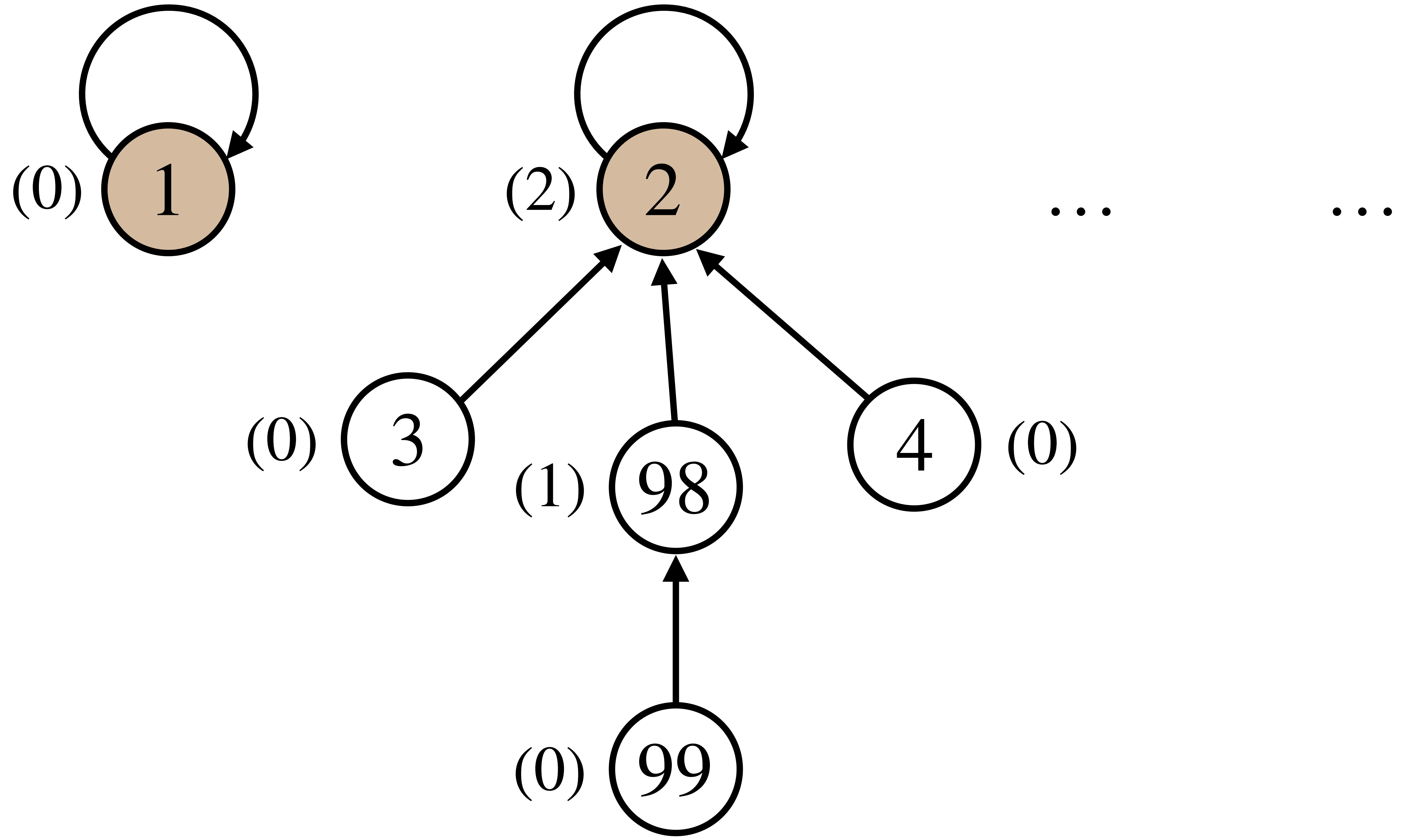


Union on Disjoint-Sets as Trees using Rank



Union on Disjoint-Sets as Trees using Rank

Sets:



Disjoint-Sets as Trees: Operations

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x)**:**

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x)**:**

1. $x.p = x$

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x)**:**

1. $x\,.\,p = x$
2. $x\,.\,rank = 0$

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x)**:**

1. $x\,.\,p = x$
2. $x\,.\,rank = 0$

Find-Set(x)**:**

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x)**:**

1. $x\,.\,p = x$
2. $x\,.\,rank = 0$

Find-Set(x)**:**

1. **if** $x \neq x\,.\,p$

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x)**:**

1. $x\,.\,p = x$
2. $x\,.\,rank = 0$

Find-Set(x)**:**

1. **if** $x \neq x\,.\,p$
2.   **return Find-Set**$(x\,.\,p)$

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x)**:**

1. $x.p = x$
2. $x.rank = 0$

Find-Set(x)**:**

1. **if** $x \neq x.p$
2. **return Find-Set**$(x.p)$
3. **else**

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x)**:**

1. $x\,.\,p = x$
2. $x\,.\,rank = 0$

Find-Set(x)**:**

1. **if** $x \neq x\,.\,p$
2.   **return Find-Set**$(x\,.\,p)$
3. **else**
4.   **return** x

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x):

1. $x\,.\,p = x$
2. $x\,.\,rank = 0$

Find-Set(x):

1. **if** $x \neq x\,.\,p$
2. **return Find-Set**($x\,.\,p$)
3. **else**
4. **return** x

Union(x, y):

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x):

1. $x.p = x$
2. $x.rank = 0$

Find-Set(x):

1. **if** $x \neq x.p$
2.   **return Find-Set**($x.p$)
3. **else**
4.   **return** x

Union(x, y):

1. $x =$ **Find-Set**(x), $y =$ **Find-Set**(y)

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x):

1. $x.p = x$
2. $x.rank = 0$

Find-Set(x):

1. **if** $x \neq x.p$
2.   **return Find-Set**$(x.p)$
3. **else**
4.   **return** x

Union(x, y):

1. $x =$ **Find-Set**(x), $y =$ **Find-Set**(y)
2. **if** $x.rank > y.rank$

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x):

1. $x.p = x$
2. $x.rank = 0$

Find-Set(x):

1. **if** $x \neq x.p$
2.   **return Find-Set**$(x.p)$
3. **else**
4.   **return** x

Union(x, y):

1. $x =$ **Find-Set**(x), $y =$ **Find-Set**(y)
2. **if** $x.rank > y.rank$
3. $y.p = x$

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x)**:**

1. $x.p = x$
2. $x.rank = 0$

Find-Set(x)**:**

1. **if** $x \neq x.p$
2.   **return Find-Set**$(x.p)$
3. **else**
4.   **return** x

Union(x, y)**:**

1. $x =$ **Find-Set**(x), $y =$ **Find-Set**(y)
2. **if** $x.rank > y.rank$
3. $y.p = x$
4. **else if** $x.rank < y.rank$

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x):

1. $x\,.\,p = x$
2. $x\,.\,rank = 0$

Find-Set(x):

1. **if** $x \neq x\,.\,p$
2.   **return Find-Set**$(x\,.\,p)$
3. **else**
4.   **return** x

Union(x, y):

1. $x =$ **Find-Set**(x), $y =$ **Find-Set**(y)
2. **if** $x\,.\,rank > y\,.\,rank$
3. $y\,.\,p = x$
4. **else if** $x\,.\,rank < y\,.\,rank$
5. $x\,.\,p = y$

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x)**:**

1. $x.p = x$
2. $x.rank = 0$

Find-Set(x)**:**

1. **if** $x \neq x.p$
2.   **return Find-Set**$(x.p)$
3. **else**
4.   **return** x

Union(x, y)**:**

1. $x =$ **Find-Set**(x), $y =$ **Find-Set**(y)
2. **if** $x.rank > y.rank$
3. $y.p = x$
4. **else if** $x.rank < y.rank$
5. $x.p = y$
6. **else**

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x):

$x.p = x$
$x.rank = 0$

Find-Set(x):

if $x \neq x.p$
    return Find-Set$(x.p)$
else
    return x

Union(x, y):

$x =$ **Find-Set**(x), $y =$ **Find-Set**(y)
if $x.rank > y.rank$
    $y.p = x$
else if $x.rank < y.rank$
    $x.p = y$
else
    $x.p = y$

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x):

1. $x\,.\,p = x$
2. $x\,.\,rank = 0$

Find-Set(x):

1. **if** $x \neq x\,.\,p$
2.   **return Find-Set**$(x\,.\,p)$
3. **else**
4.   **return** x

Union(x, y):

1. $x =$ **Find-Set**(x), $y =$ **Find-Set**(y)
2. **if** $x\,.\,rank > y\,.\,rank$
3. $y\,.\,p = x$
4. **else if** $x\,.\,rank < y\,.\,rank$
5. $x\,.\,p = y$
6. **else**
7. $x\,.\,p = y$
8. $y\,.\,rank = y\,.\,rank + 1$

Disjoint-Sets as Trees: Operations

Recall that we need to perform three operations: **Make-Set**(x), **Union**(x, y), and **Find-Set**(x).

Make-Set(x):

$x.p = x$
$x.rank = 0$

Find-Set(x):

if $x \neq x.p$
    return Find-Set($x.p$)
else
    return x

assume x and y are in different sets

Union(x, y):

$x =$ **Find-Set**(x), $y =$ **Find-Set**(y)
if $x.rank > y.rank$
    $y.p = x$
else if $x.rank < y.rank$
    $x.p = y$
else
    $x.p = y$
    $y.rank = y.rank + 1$

Disjoint-Sets as Trees: Analysis

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Proof:

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Proof: We will prove it using induction on h.

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Proof: We will prove it using induction on h.

Basis Step:

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Proof: We will prove it using induction on h.

Basis Step: Nodes in subtree of a node with height 0 contains 1 node.

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Proof: We will prove it using induction on h.

Basis Step: Nodes in subtree of a node with height 0 contains 1 node. Trivially true.

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Proof: We will prove it using induction on h.

Basis Step: Nodes in subtree of a node with height 0 contains 1 node. Trivially true.

Inductive Step:

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Proof: We will prove it using induction on h.

Basis Step: Nodes in subtree of a node with height 0 contains 1 node. Trivially true.

Inductive Step: Assuming the claim is true for nodes with rank $\leq i$, we will prove it for

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Proof: We will prove it using induction on h.

Basis Step: Nodes in subtree of a node with height 0 contains 1 node. Trivially true.

Inductive Step: Assuming the claim is true for nodes with rank $\leq i$, we will prove it for nodes with rank $i + 1$.

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Proof: We will prove it using induction on h.

Basis Step: Nodes in subtree of a node with height 0 contains 1 node. Trivially true.

Inductive Step: Assuming the claim is true for nodes with rank $\leq i$, we will prove it for nodes with rank $i + 1$.

Let x be a node with rank $i + 1$.

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Proof: We will prove it using induction on h.

Basis Step: Nodes in subtree of a node with height 0 contains 1 node. Trivially true.

Inductive Step: Assuming the claim is true for nodes with rank $\leq i$, we will prove it for nodes with rank $i + 1$.

Let x be a node with rank $i + 1$.

Case 1**:** The first time when x's rank changed from i to $i + 1$ and it became root of tree, say T,

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Proof: We will prove it using induction on h.

Basis Step: Nodes in subtree of a node with height 0 contains 1 node. Trivially true.

Inductive Step: Assuming the claim is true for nodes with rank $\leq i$, we will prove it for nodes with rank $i + 1$.

Let x be a node with rank $i + 1$.

Case 1**:** The first time when x's rank changed from i to $i + 1$ and it became root of tree, say T, it must have been a union of two trees say T_1 and T_2 with their roots' height i.

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Proof: We will prove it using induction on h.

Basis Step: Nodes in subtree of a node with height 0 contains 1 node. Trivially true.

Inductive Step: Assuming the claim is true for nodes with rank $\leq i$, we will prove it for nodes with rank $i + 1$.

Let x be a node with rank $i + 1$.

Case 1**:** The first time when x's rank changed from i to $i + 1$ and it became root of tree, say T, it must have been a union of two trees say T_1 and T_2 with their roots' height i.

From inductive hypothesis each of T_1 and T_2 contain at least 2^i nodes.

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Proof: We will prove it using induction on h.

Basis Step: Nodes in subtree of a node with height 0 contains 1 node. Trivially true.

Inductive Step: Assuming the claim is true for nodes with rank $\leq i$, we will prove it for nodes with rank $i+1$.

Let x be a node with rank $i+1$.

Case 1**:** The first time when x's rank changed from i to $i+1$ and it became root of tree, say T, it must have been a union of two trees say T_1 and T_2 with their roots' height i.

From inductive hypothesis each of T_1 and T_2 contain at least 2^i nodes.

Hence, $T = T_1 \cup T_2$ will contain at least $2^i + 2^i = 2^{i+1}$ nodes.

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Proof: We will prove it using induction on h.

Basis Step: Nodes in subtree of a node with height 0 contains 1 node. Trivially true.

Inductive Step: Assuming the claim is true for nodes with rank $\leq i$, we will prove it for nodes with rank $i + 1$.

Let x be a node with rank $i + 1$.

Case 1**:** The first time when x's rank changed from i to $i + 1$ and it became root of tree, say T, it must have been a union of two trees say T_1 and T_2 with their roots' height i.

From inductive hypothesis each of T_1 and T_2 contain at least 2^i nodes.

Hence, $T = T_1 \cup T_2$ will contain at least $2^i + 2^i = 2^{i+1}$ nodes.

Case 2**:** At rank $i + 1$ as root, the **union** operations only increase the nodes in x's subtree.

Disjoint-Sets as Trees: Analysis

Claim: A node with rank (or height) h has at least 2^h nodes in its subtree.

Proof: We will prove it using induction on h.

Basis Step: Nodes in subtree of a node with height 0 contains 1 node. Trivially true.

Inductive Step: Assuming the claim is true for nodes with rank $\leq i$, we will prove it for nodes with rank $i + 1$.

Let x be a node with rank $i + 1$.

Case 1**:** The first time when x's rank changed from i to $i + 1$ and it became root of tree, say T, it must have been a union of two trees say T_1 and T_2 with their roots' height i.

From inductive hypothesis each of T_1 and T_2 contain at least 2^i nodes.

Hence, $T = T_1 \cup T_2$ will contain at least $2^i + 2^i = 2^{i+1}$ nodes.

Case 2**:** At rank $i + 1$ as root, the **union** operations only increase the nodes in x's subtree. $\blacksquare$

Disjoint-Sets as Trees: Analysis

Disjoint-Sets as Trees: Analysis

Claim: Every node has rank at most $\lfloor \lg n \rfloor$ in the disjoint-set via trees using rank heuristic.

Disjoint-Sets as Trees: Analysis

Claim: Every node has rank at most $\lfloor \lg n \rfloor$ in the disjoint-set via trees using rank heuristic.

Proof:

Disjoint-Sets as Trees: Analysis

Claim: Every node has rank at most $\lfloor \lg n \rfloor$ in the disjoint-set via trees using rank heuristic.

Proof: Suppose a node has rank $\lfloor \lg n \rfloor + k$, where $k > 0$.

Disjoint-Sets as Trees: Analysis

Claim: Every node has rank at most $\lfloor \lg n \rfloor$ in the disjoint-set via trees using rank heuristic.

Proof: Suppose a node has rank $\lfloor \lg n \rfloor + k$, where $k > 0$.

Then, from previous claim its subtree should contain at least $2^{\lfloor \lg n \rfloor + k}$ nodes.

Disjoint-Sets as Trees: Analysis

Claim: Every node has rank at most $\lfloor \lg n \rfloor$ in the disjoint-set via trees using rank heuristic.

Proof: Suppose a node has rank $\lfloor \lg n \rfloor + k$, where $k > 0$.

Then, from previous claim its subtree should contain at least $2^{\lfloor \lg n \rfloor + k}$ nodes.

But, $2^{\lfloor \lg n \rfloor + k} > n$, which is not possible.

Disjoint-Sets as Trees: Analysis

Claim: Every node has rank at most $\lfloor \lg n \rfloor$ in the disjoint-set via trees using rank heuristic.

Proof: Suppose a node has rank $\lfloor \lg n \rfloor + k$, where $k > 0$.

Then, from previous claim its subtree should contain at least $2^{\lfloor \lg n \rfloor + k}$ nodes.

But, $2^{\lfloor \lg n \rfloor + k} > n$, which is not possible. ∎

Disjoint-Sets as Trees: Analysis

Claim: Every node has rank at most $\lfloor \lg n \rfloor$ in the disjoint-set via trees using rank heuristic.

Proof: Suppose a node has rank $\lfloor \lg n \rfloor + k$, where $k > 0$.

Then, from previous claim its subtree should contain at least $2^{\lfloor \lg n \rfloor + k}$ nodes.

But, $2^{\lfloor \lg n \rfloor + k} > n$, which is not possible. ∎

Claim: A sequence of m **Make-Set**, **Union**, & **Find-Set** operations,

Disjoint-Sets as Trees: Analysis

Claim: Every node has rank at most $\lfloor \lg n \rfloor$ in the disjoint-set via trees using rank heuristic.

Proof: Suppose a node has rank $\lfloor \lg n \rfloor + k$, where $k > 0$.

Then, from previous claim its subtree should contain at least $2^{\lfloor \lg n \rfloor + k}$ nodes.

But, $2^{\lfloor \lg n \rfloor + k} > n$, which is not possible. ■

Claim: A sequence of m **Make-Set**, **Union**, & **Find-Set** operations, first n of which are **Make-Set**

Disjoint-Sets as Trees: Analysis

Claim: Every node has rank at most $\lfloor \lg n \rfloor$ in the disjoint-set via trees using rank heuristic.

Proof: Suppose a node has rank $\lfloor \lg n \rfloor + k$, where $k > 0$.

Then, from previous claim its subtree should contain at least $2^{\lfloor \lg n \rfloor + k}$ nodes.

But, $2^{\lfloor \lg n \rfloor + k} > n$, which is not possible. ■

Claim: A sequence of m **Make-Set**, **Union**, & **Find-Set** operations, first n of which are **Make-Set** operations, takes $O(m \lg n)$ time in the tree using rank implementation.

Disjoint-Sets as Trees: Analysis

Claim: Every node has rank at most $\lfloor \lg n \rfloor$ in the disjoint-set via trees using rank heuristic.

Proof: Suppose a node has rank $\lfloor \lg n \rfloor + k$, where $k > 0$.

Then, from previous claim its subtree should contain at least $2^{\lfloor \lg n \rfloor + k}$ nodes.

But, $2^{\lfloor \lg n \rfloor + k} > n$, which is not possible. ■

Claim: A sequence of m **Make-Set**, **Union**, & **Find-Set** operations, first n of which are **Make-Set** operations, takes $O(m \lg n)$ time in the tree using rank implementation.

Proof:

Disjoint-Sets as Trees: Analysis

Claim: Every node has rank at most $\lfloor \lg n \rfloor$ in the disjoint-set via trees using rank heuristic.

Proof: Suppose a node has rank $\lfloor \lg n \rfloor + k$, where $k > 0$.

Then, from previous claim its subtree should contain at least $2^{\lfloor \lg n \rfloor + k}$ nodes.

But, $2^{\lfloor \lg n \rfloor + k} > n$, which is not possible. ■

Claim: A sequence of m **Make-Set**, **Union**, & **Find-Set** operations, first n of which are **Make-Set** operations, takes $O(m \lg n)$ time in the tree using rank implementation.

Proof: **Make-Set** operations take constant time.

Disjoint-Sets as Trees: Analysis

Claim: Every node has rank at most $\lfloor \lg n \rfloor$ in the disjoint-set via trees using rank heuristic.

Proof: Suppose a node has rank $\lfloor \lg n \rfloor + k$, where $k > 0$.

Then, from previous claim its subtree should contain at least $2^{\lfloor \lg n \rfloor + k}$ nodes.

But, $2^{\lfloor \lg n \rfloor + k} > n$, which is not possible. ■

Claim: A sequence of m **Make-Set**, **Union**, & **Find-Set** operations, first n of which are **Make-Set** operations, takes $O(m \lg n)$ time in the tree using rank implementation.

Proof: **Make-Set** operations take constant time.

Union operations take the same time as **Find-Set**.

Disjoint-Sets as Trees: Analysis

Claim: Every node has rank at most $\lfloor \lg n \rfloor$ in the disjoint-set via trees using rank heuristic.

Proof: Suppose a node has rank $\lfloor \lg n \rfloor + k$, where $k > 0$.

Then, from previous claim its subtree should contain at least $2^{\lfloor \lg n \rfloor + k}$ nodes.

But, $2^{\lfloor \lg n \rfloor + k} > n$, which is not possible. ∎

Claim: A sequence of m **Make-Set**, **Union**, & **Find-Set** operations, first n of which are **Make-Set** operations, takes $O(m \lg n)$ time in the tree using rank implementation.

Proof: **Make-Set** operations take constant time.

Union operations take the same time as **Find-Set**.

Find-Set operations take $O(h)$ time, where $h = \lg n$ is the rank of the root of the tree.

Disjoint-Sets as Trees: Analysis

Claim: Every node has rank at most $\lfloor \lg n \rfloor$ in the disjoint-set via trees using rank heuristic.

Proof: Suppose a node has rank $\lfloor \lg n \rfloor + k$, where $k > 0$.

Then, from previous claim its subtree should contain at least $2^{\lfloor \lg n \rfloor + k}$ nodes.

But, $2^{\lfloor \lg n \rfloor + k} > n$, which is not possible. ■

Claim: A sequence of m **Make-Set**, **Union**, & **Find-Set** operations, first n of which are **Make-Set** operations, takes $O(m \lg n)$ time in the tree using rank implementation.

Proof: **Make-Set** operations take constant time.

Union operations take the same time as **Find-Set**.

Find-Set operations take $O(h)$ time, where $h = \lg n$ is the rank of the root of the tree.

Hence, m operations take $O(m \lg n)$ time.

Disjoint-Sets as Trees: Analysis

Claim: Every node has rank at most $\lfloor \lg n \rfloor$ in the disjoint-set via trees using rank heuristic.

Proof: Suppose a node has rank $\lfloor \lg n \rfloor + k$, where $k > 0$.

Then, from previous claim its subtree should contain at least $2^{\lfloor \lg n \rfloor + k}$ nodes.

But, $2^{\lfloor \lg n \rfloor + k} > n$, which is not possible. ■

Claim: A sequence of m **Make-Set**, **Union**, & **Find-Set** operations, first n of which are **Make-Set** operations, takes $O(m \lg n)$ time in the tree using rank implementation.

Proof: **Make-Set** operations take constant time.

Union operations take the same time as **Find-Set**.

Find-Set operations take $O(h)$ time, where $h = \lg n$ is the rank of the root of the tree.

Hence, m operations take $O(m \lg n)$ time. ■

Path-Compression Heuristic

Path-Compression Heuristic

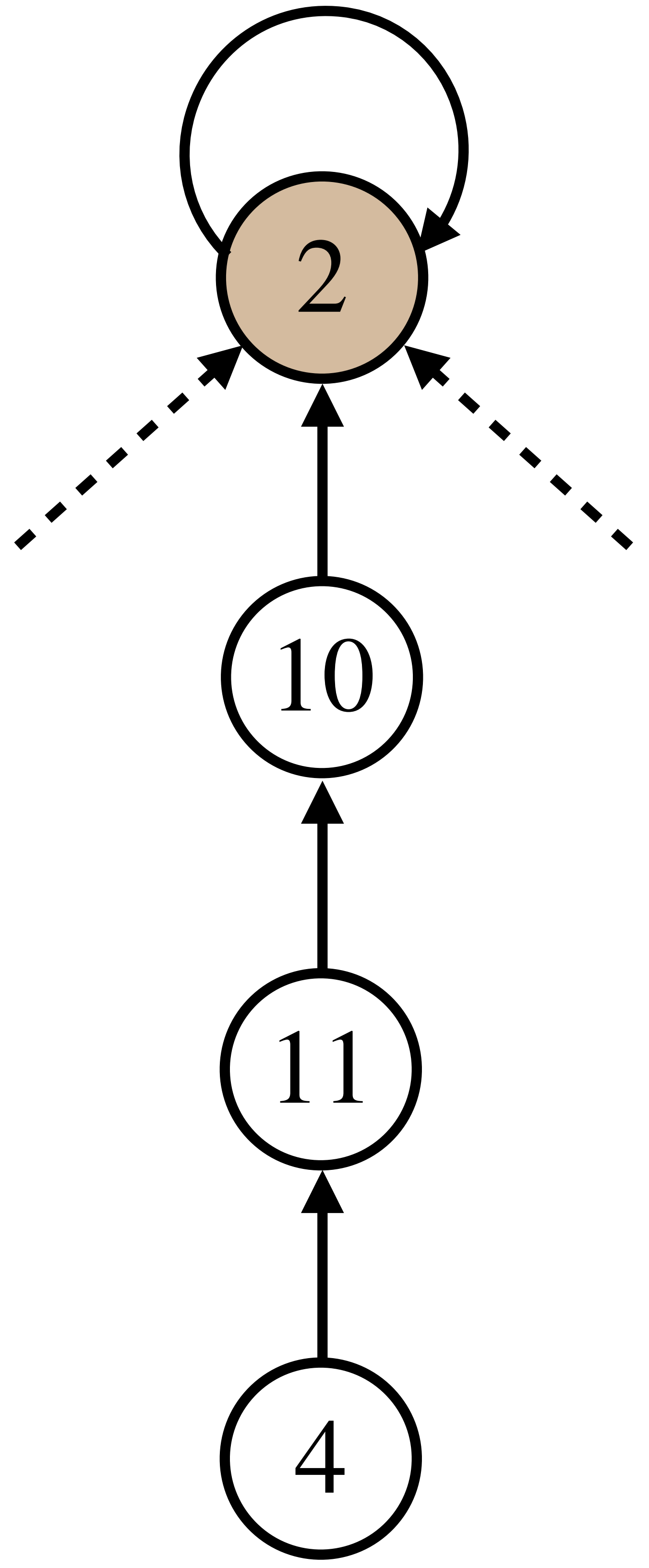
In Path-Compression, while performing **Find-Set**(x)

Path-Compression Heuristic

In Path-Compression, while performing **Find-Set**(x) we make root the parent of every node

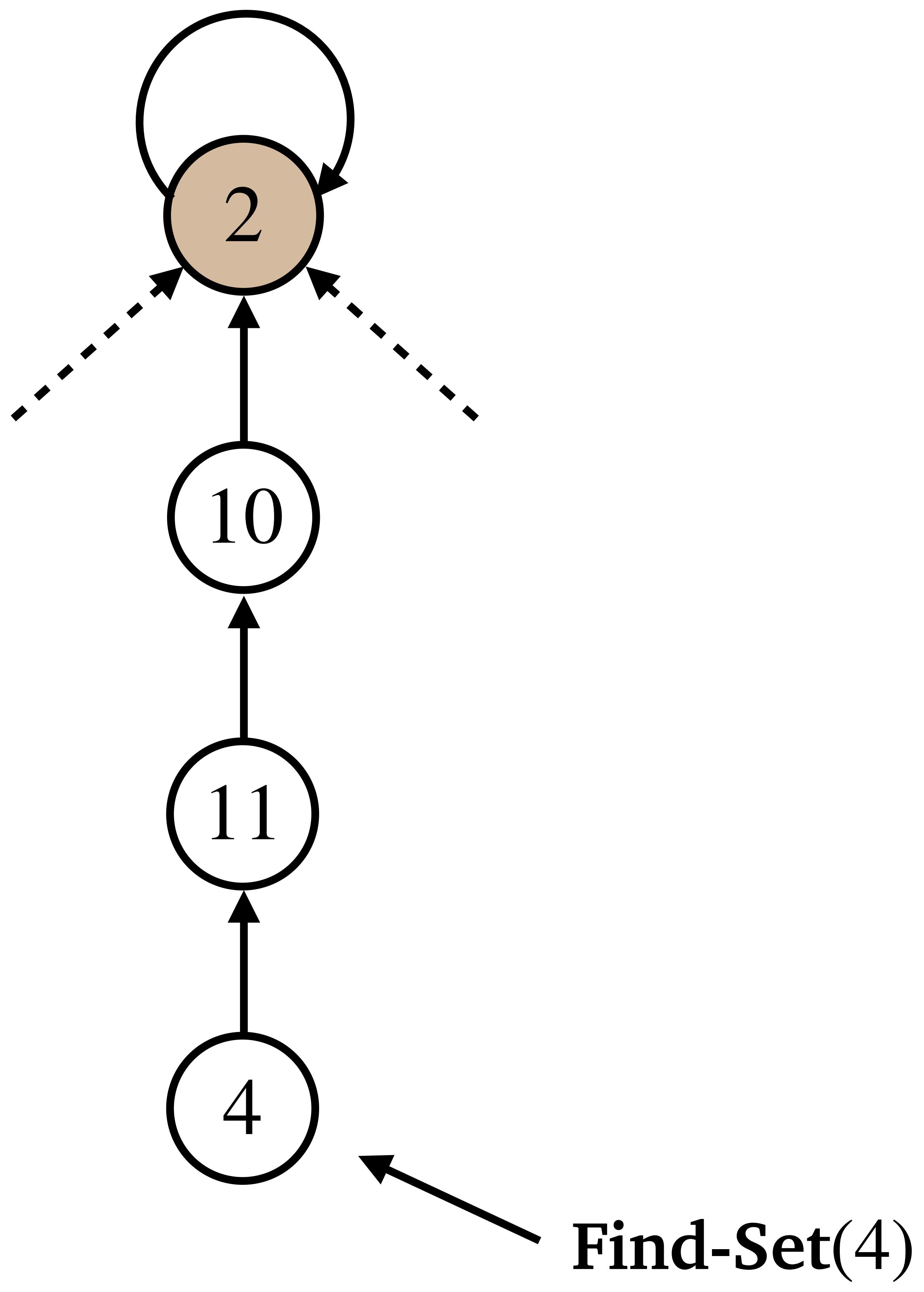
Path-Compression Heuristic

In Path-Compression, while performing **Find-Set**(x) we make root the parent of every node on path from x to root.



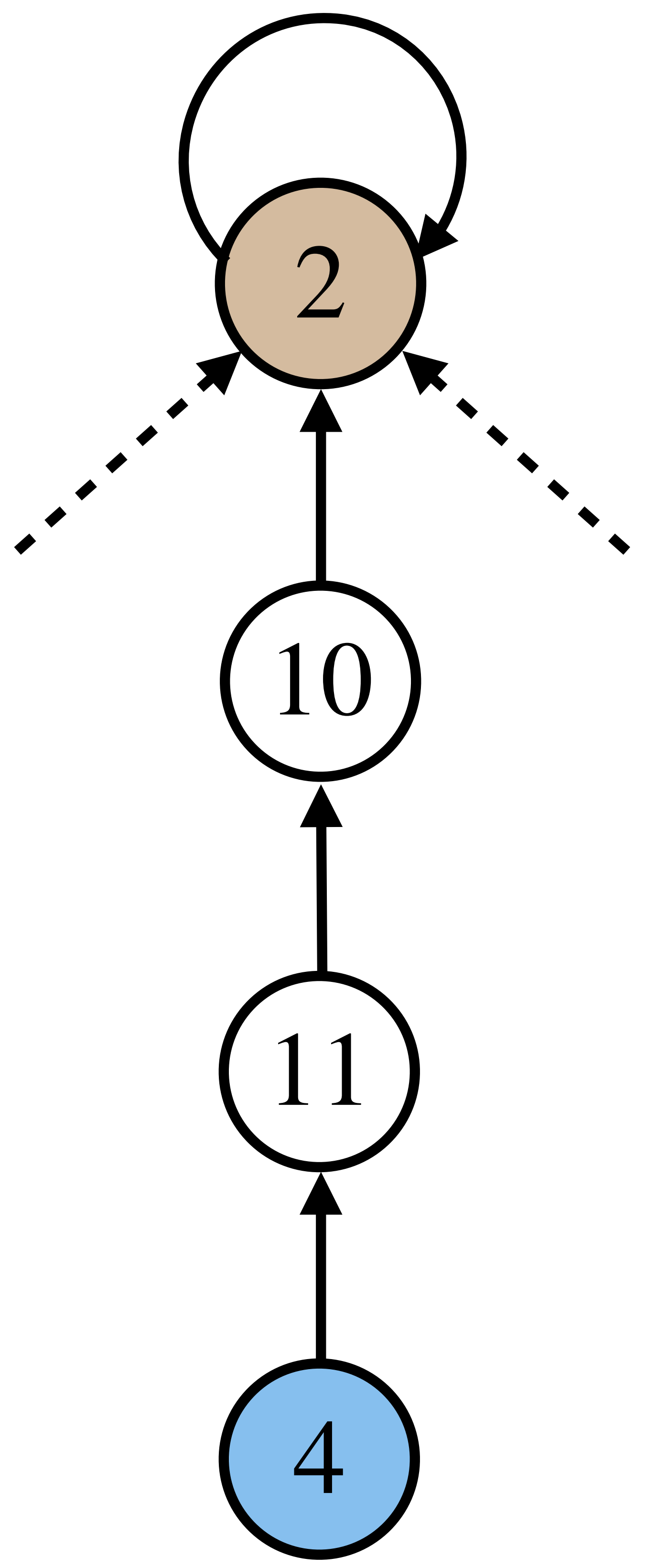
Path-Compression Heuristic

In Path-Compression, while performing **Find-Set**(x) we make root the parent of every node on path from x to root.



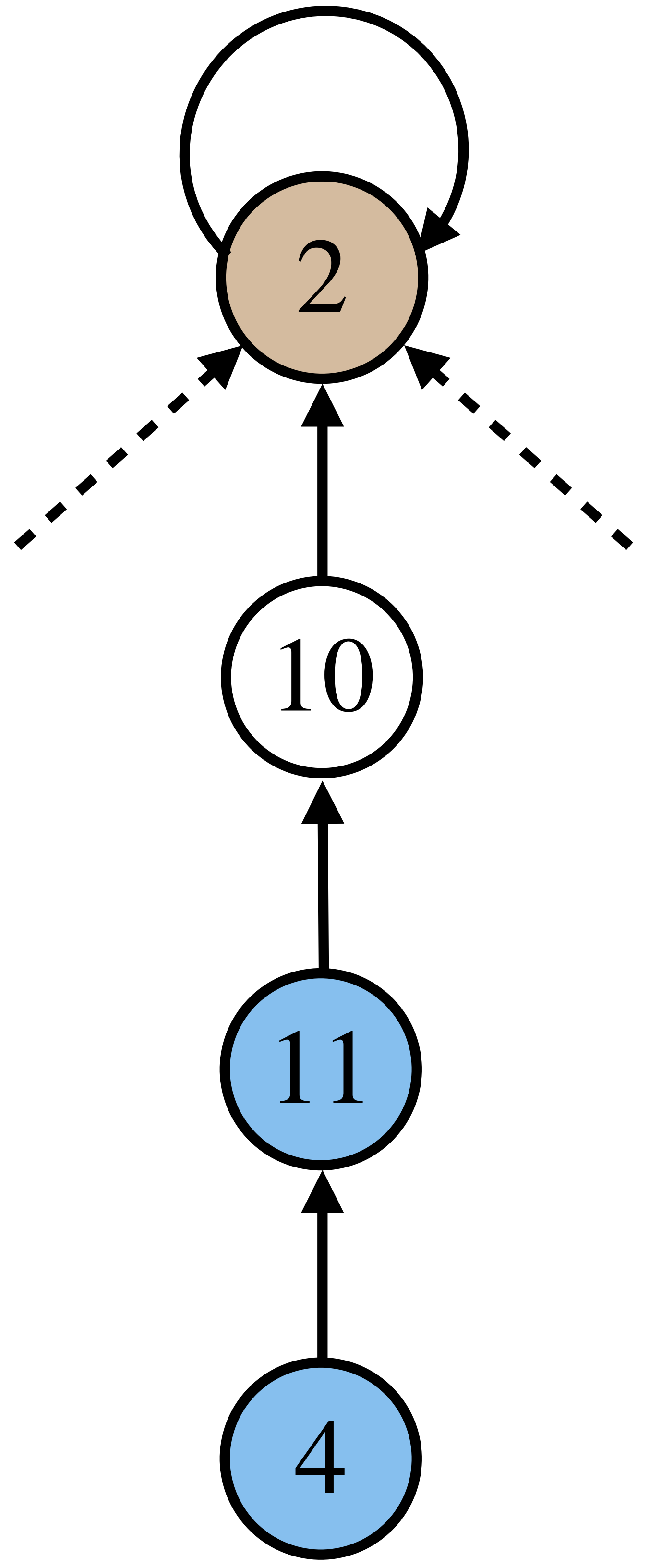
Path-Compression Heuristic

In Path-Compression, while performing **Find-Set**(x) we make root the parent of every node on path from x to root.



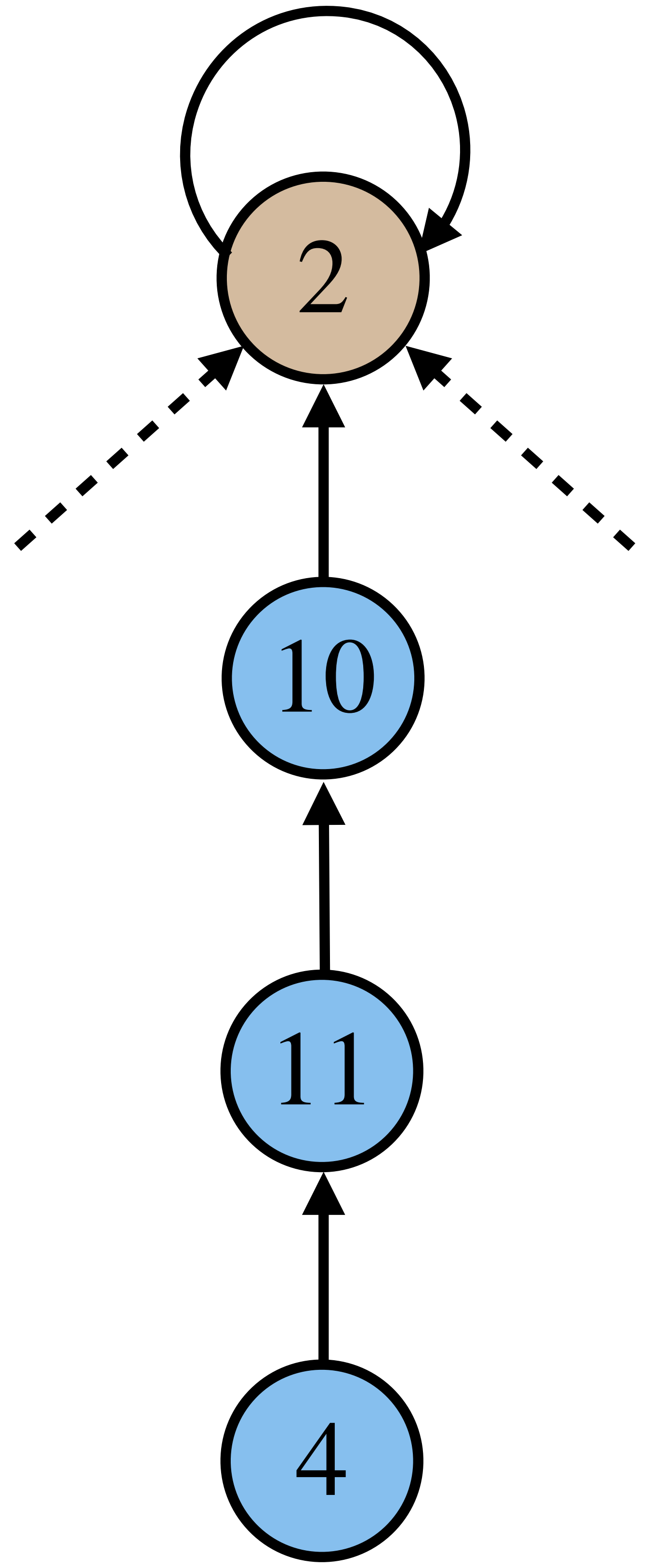
Path-Compression Heuristic

In Path-Compression, while performing **Find-Set**(x) we make root the parent of every node on path from x to root.



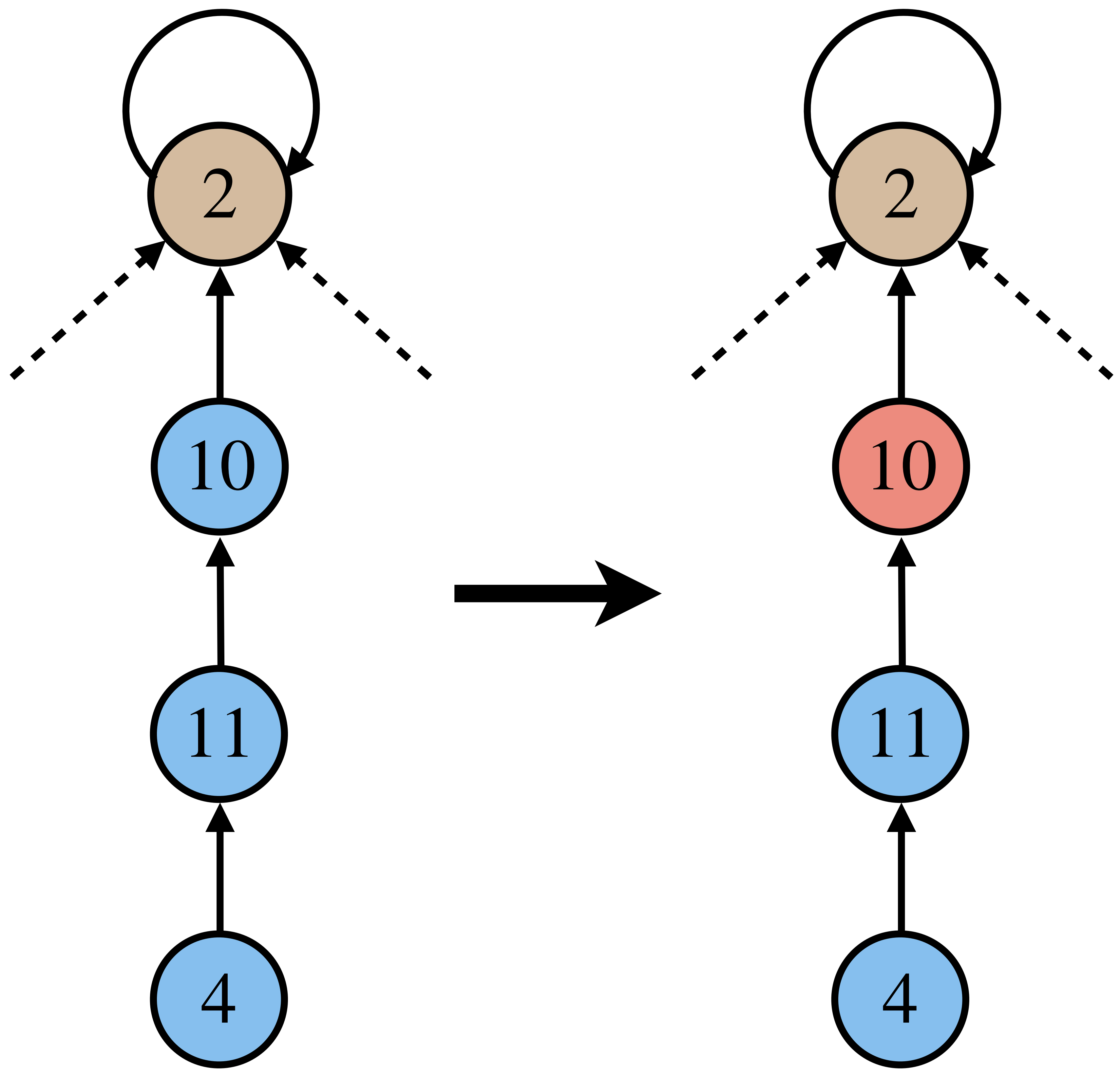
Path-Compression Heuristic

In Path-Compression, while performing $\textbf{Find-Set}(x)$ we make root the parent of every node on path from x to root.



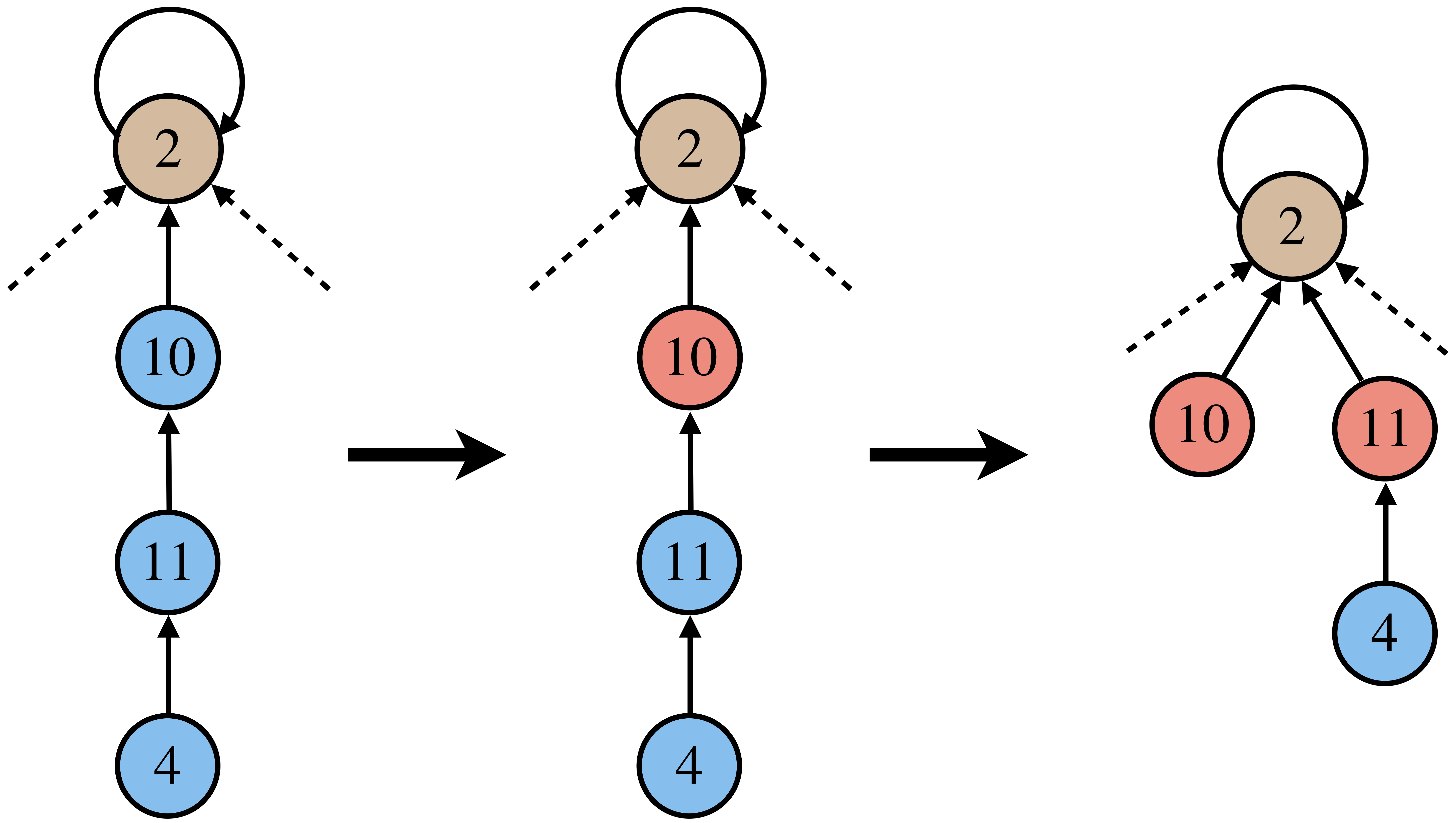
Path-Compression Heuristic

In Path-Compression, while performing **Find-Set**(x) we make root the parent of every node on path from x to root.



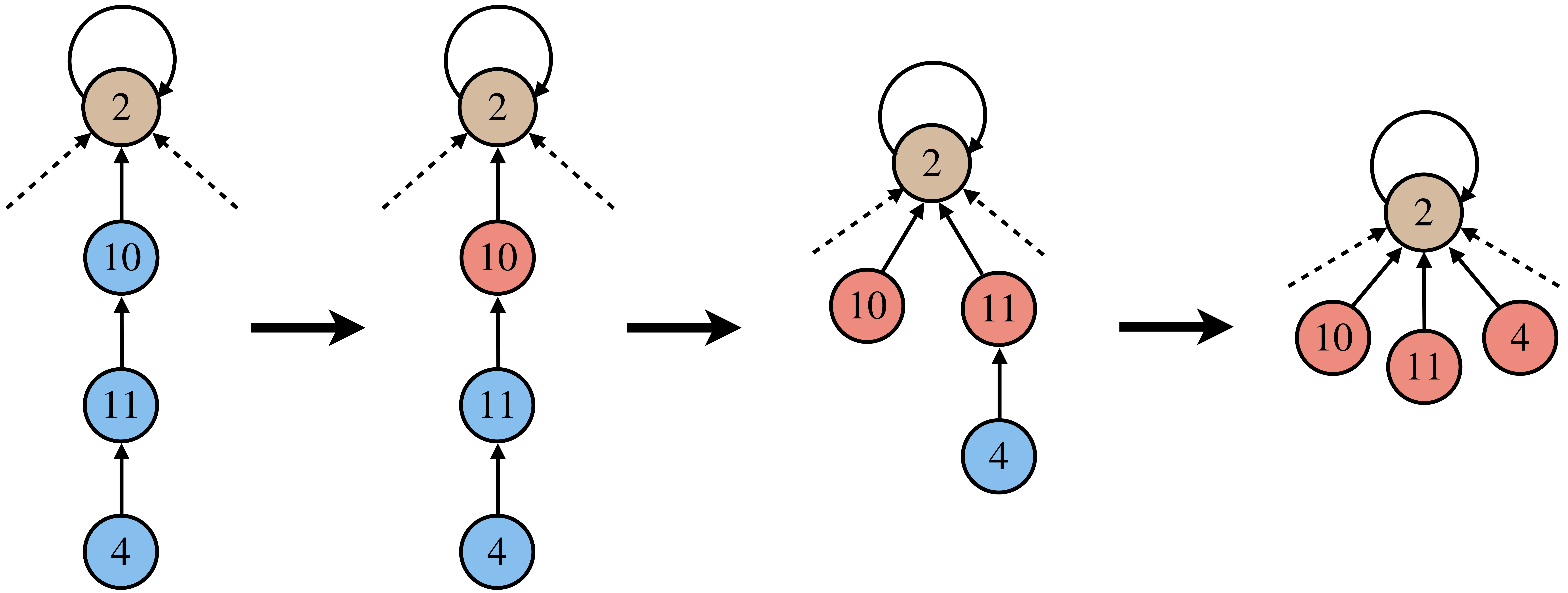
Path-Compression Heuristic

In Path-Compression, while performing **Find-Set**(x) we make root the parent of every node on path from x to root.



Path-Compression Heuristic

In Path-Compression, while performing **Find-Set**(x) we make root the parent of every node on path from x to root.



Disjoint-Sets as Trees: Operations

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x)**:**

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x)**:**

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

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x):

1. **if** $x \neq x\,.\,p$
2. **return Find-Set**($x\,.\,p$)

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x):

1. **if** $x \neq x\,.\,p$
2.   **return Find-Set**($x\,.\,p$)
3. **else**

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x)**:**

1. **if** $x \neq x\,.\,p$
2. **return Find-Set**$(x\,.\,p)$
3. **else**
4. **return** x

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x):
1. **if** $x \neq x\,.\,p$
2.   **return Find-Set**($x\,.\,p$)
3. **else**
4.   **return** x

Old **Find-Set**(x)

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x):

1. **if** $x \neq x\,.\,p$
2. **return Find-Set**$(x\,.\,p)$
3. **else**
4. **return** x

Old **Find-Set**(x)

Find-Set(x):

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x):

1. **if** $x \neq x\,.\,p$
2. **return Find-Set**$(x\,.\,p)$
3. **else**
4. **return** x

Old **Find-Set**(x)

$\longrightarrow$

Find-Set(x):

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x):

1. **if** $x \neq x\,.\,p$
2. **return Find-Set**$(x\,.\,p)$
3. **else**
4. **return** x

Old **Find-Set**(x)

→

Find-Set(x):

1. **if** $x \neq x\,.\,p$

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x):

1. **if** $x \neq x\,.\,p$
2. **return Find-Set**$(x\,.\,p)$
3. **else**
4. **return** x

Old **Find-Set**(x)

$\longrightarrow$

Find-Set(x):

1. **if** $x \neq x\,.\,p$
2. $x\,.\,p =$ **Find-Set**$(x\,.\,p)$

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x):
1. **if** $x \neq x\,.\,p$
2.   **return Find-Set**$(x\,.\,p)$
3. **else**
4.   **return** x

Old **Find-Set**(x)

$\longrightarrow$

Find-Set(x):
1. **if** $x \neq x\,.\,p$
2. $x\,.\,p =$ **Find-Set**$(x\,.\,p)$
3. **return** $x\,.\,p$

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x):

1. **if** $x \neq x\,.\,p$
2. **return Find-Set**$(x\,.\,p)$
3. **else**
4. **return** x

Old **Find-Set**(x)

→

Find-Set(x):

1. **if** $x \neq x\,.\,p$
2. $x\,.\,p =$ **Find-Set**$(x\,.\,p)$
3. **return** $x\,.\,p$

Find-Set(x) with path-compression

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x):
1. **if** $x \neq x\,.\,p$
2. **return Find-Set**$(x\,.\,p)$
3. **else**
4. **return** x

Old **Find-Set**(x)

$\longrightarrow$

Find-Set(x):
1. **if** $x \neq x\,.\,p$
2. $x\,.\,p =$ **Find-Set**$(x\,.\,p)$
3. **return** $x\,.\,p$

Find-Set(x) with path-compression

Claim: A sequence of m **Make-Set**, **Union**, & **Find-Set** operations,

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x):

if $x \neq x \,.\, p$
    return Find-Set$(x \,.\, p)$
else
    return x

Old **Find-Set**(x)

Find-Set(x):

if $x \neq x \,.\, p$
    $x \,.\, p =$ **Find-Set**$(x \,.\, p)$
return $x \,.\, p$

Find-Set(x) with path-compression

Claim: A sequence of m **Make-Set**, **Union**, & **Find-Set** operations, first n of which are **Make-Set**

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x):
1. **if** $x \neq x\,.\,p$
2. **return Find-Set**($x\,.\,p$)
3. **else**
4. **return** x

Old **Find-Set**(x)

$\longrightarrow$

Find-Set(x):
1. **if** $x \neq x\,.\,p$
2. $x\,.\,p =$ **Find-Set**($x\,.\,p$)
3. **return** $x\,.\,p$

Find-Set(x) with path-compression

Claim: A sequence of m **Make-Set**, **Union**, & **Find-Set** operations, first n of which are **Make-Set** operations, takes $O(m\alpha(n))$ time using **rank** and **path-compression** heuristic.

Disjoint-Sets as Trees: Operations

Change **Find-Set**(x) to implement path-compression.

Find-Set(x):
1. **if** $x \neq x\,.\,p$
2. **return Find-Set**$(x\,.\,p)$
3. **else**
4. **return** x

Old **Find-Set**(x)

$\longrightarrow$

Find-Set(x):
1. **if** $x \neq x\,.\,p$
2. $x\,.\,p =$ **Find-Set**$(x\,.\,p)$
3. **return** $x\,.\,p$

Find-Set(x) with path-compression

Claim: A sequence of m **Make-Set**, **Union**, & **Find-Set** operations, first n of which are **Make-Set** operations, takes $O(m\alpha(n))$ time using **rank** and **path-compression** heuristic.

$\alpha(n) \leq 4$ for all practical purposes