Data Structures and Algorithms

- Set -

School of Electrical Engineering
Korea University

Equivalence Relations

- A relation **R** is defined on a set S if for every pair of elements $(\boldsymbol{a}, \boldsymbol{b})$ where $\boldsymbol{a}, \boldsymbol{b} \in$ S, <u>$\boldsymbol{a}$ **R** $\boldsymbol{b}$ is either true or false</u>.
- If $\boldsymbol{a}$ **R** $\boldsymbol{b}$ is true, then $\boldsymbol{a}$ is related to $\boldsymbol{b}$.
- An equivalence relation is a relation **R** that satisfies three properties:
  1. (Reflexive) $\boldsymbol{a}$ **R** $\boldsymbol{a}$, for all $\boldsymbol{a} \in S$.
  2. (Symmetric) $\boldsymbol{a}$ **R** $\boldsymbol{b}$ if and only if $\boldsymbol{b}$ **R** $\boldsymbol{a}$.
  3. (Transitive) $\boldsymbol{a}$ **R** $\boldsymbol{b}$ and $\boldsymbol{b}$ **R** $\boldsymbol{c}$ implies that $\boldsymbol{a}$ **R** $\boldsymbol{c}$

Equivalence Relations

- Example
  - ≤ relationship: reflexive, transitive, but not symmetric
  - Electrical connectivity: reflexive, transitive, symmetric
  - Membership relationship if two cities are in the same country

Example

- Given an equivalence relation $\sim$, it is easy to decide if $a \sim b$ when the relation is stored as a two-dimensional array of Booleans.
- What if the relation is implicit?

  → want to be able to infer this quickly

(Ex) Suppose an equivalence relation '$\sim$' over the set $\{a_1, a_2, a_3, a_4, a_5\}$ with the following relation instances: $a_1 \sim a_2$, $a_3 \sim a_4$, $a_5 \sim a_1$, $a_4 \sim a_2$

then $a_1 \sim a_4$?

Equivalence class

- The equivalence class of an element $a \in S$ is the subset of S that contains all the elements that are related to a
- The equivalence classes form a partition of S: every member of S appears in exactly one equivalence class
- $a \sim b$ can be checked by checking whether a and b are in the same equivalence class

Equivalence problem

- The input is initially a set of *N* sets, each with one element.
- Each set has a different element, so that $S_i \cap S_j = \varnothing$; Disjoint
- Two permissible operations:
  - Find returns the name of the set containing a given element (namely, equivalence class)
  - Union merges the two equivalence classes containing *a* and *b*

Equivalence Problem

- Do not perform any operations comparing the relative values of elements, but merely require knowledge of their location

  -> all the elements have been numbered sequentially from 1 to N
- The name of the set returned by *Find* is actually fairly arbitrary. What matters is that *Find* (*a*) = *Find* (*b*) if and only if *a* and *b* are in the same set

Equivalence Problem

- These operations are important in many graph theory problems
- Two strategies
  - The Find instruction can be executed in constant worst-case time
  - The Union instruction can be executed in constant worst-case time
  - Both cannot be done simultaneously in constant worst-case time

Data Structure

- It is not required that a Find operation return any specific name.
- Rather, Finds on two elements return the same answer if and only if they are in the same set
- One idea is to use tree since each element in the tree has the same root
- Represent a set by a tree, a set of sets by a forest.

Tree representation

- The name of a set is given by the node at the root.
- A parent pointer is used
- Since only the name of the parent is required, this tree is stored implicitly in an array.
- The tree is stored implicitly in an array
  - each entry $P[i]$ in the array represents the parent of element i
  - If i is a root, then $P[i] = 0$

Implicit Array Representation

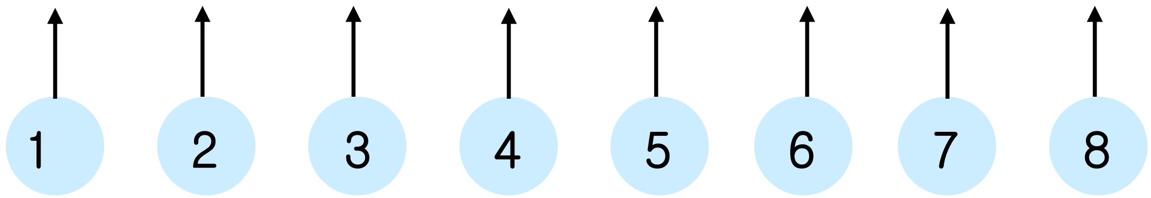


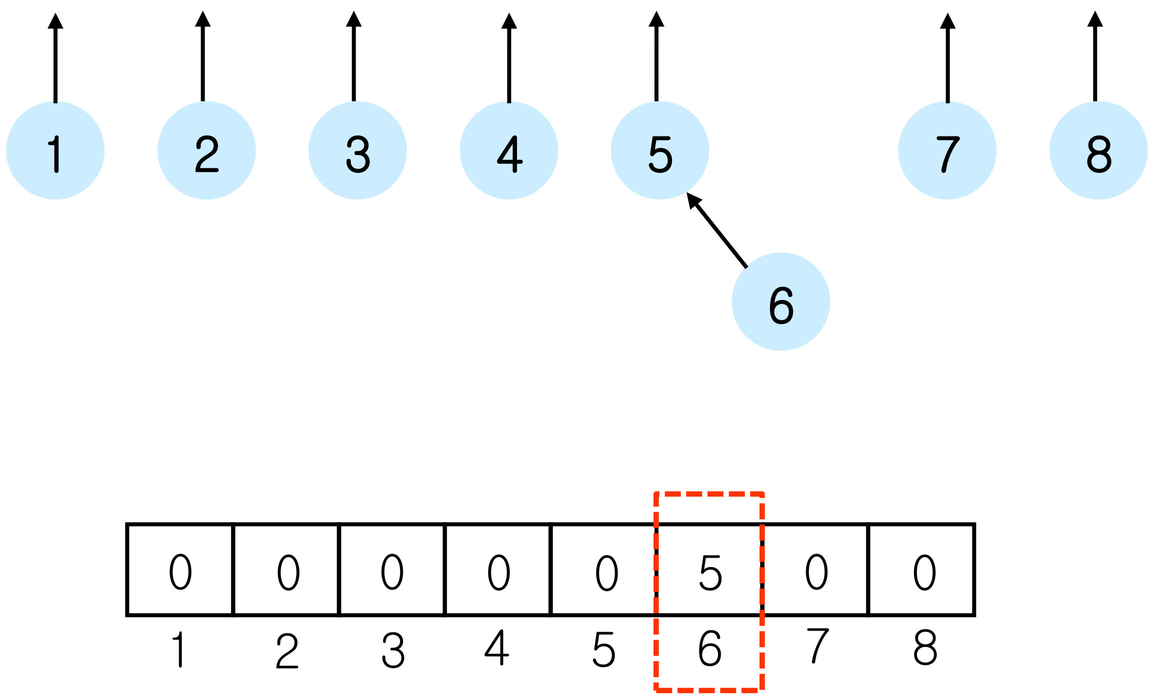
Fig. 8.1 Eight elements, initially in different sets

0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8

– Implicit array representation –

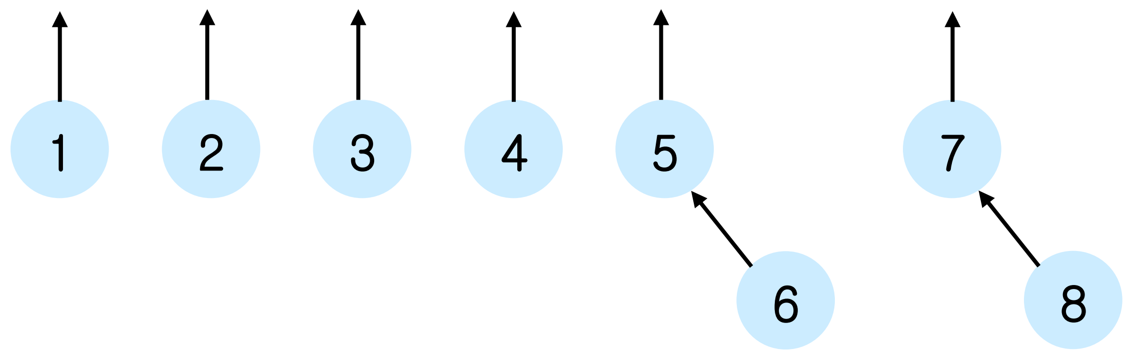
Implicit Array Representation

Union(5, 6)



Implicit Array Representation

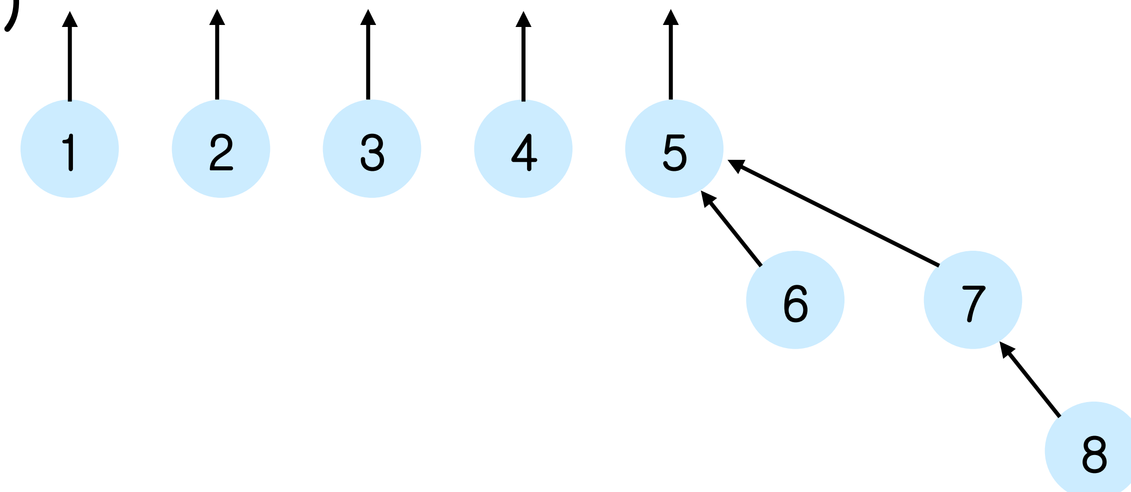
Union(7, 8)



0	0	0	0	0	5	0	7
1	2	3	4	5	6	7	8

Implicit Array Representation

Union(5, 7)



0	0	0	0	0	5	5	7
1	2	3	4	5	6	7	8

Disjoint set type declaration

```
typedef int DisjSet[ NumSets + 1]
typedef int SetType;
typedef int ElementType;

void Initialize( DisjSet S );
void SetUnion( DisjSet S, SetType R1, SetType R2);
SetType Find( ElementType X, DisjSet S );
```

Initialization routine

```
void
Initialize ( DisjSet  S )
{
    int  i ;

   for ( i = NumSets ;  i > 0 ;  i-- )
        S[ i ] = 0;
}
```

Union routine (not the best way)

```
/* Assumes R1 and R2 are roots */
/* union is a C keyword, so this routine is */
/* named SetUnion */

void
SetUnion (DisjSet S, SetType R1, SetType R2)
{
   S[R2] = R1;
}
```

Find routine

```
SetType
Find (ElementType X, DisjSet S)
{
    if ( S[X] <= 0 )
        return X;
    else
        return Find( S[X], S)
}
```

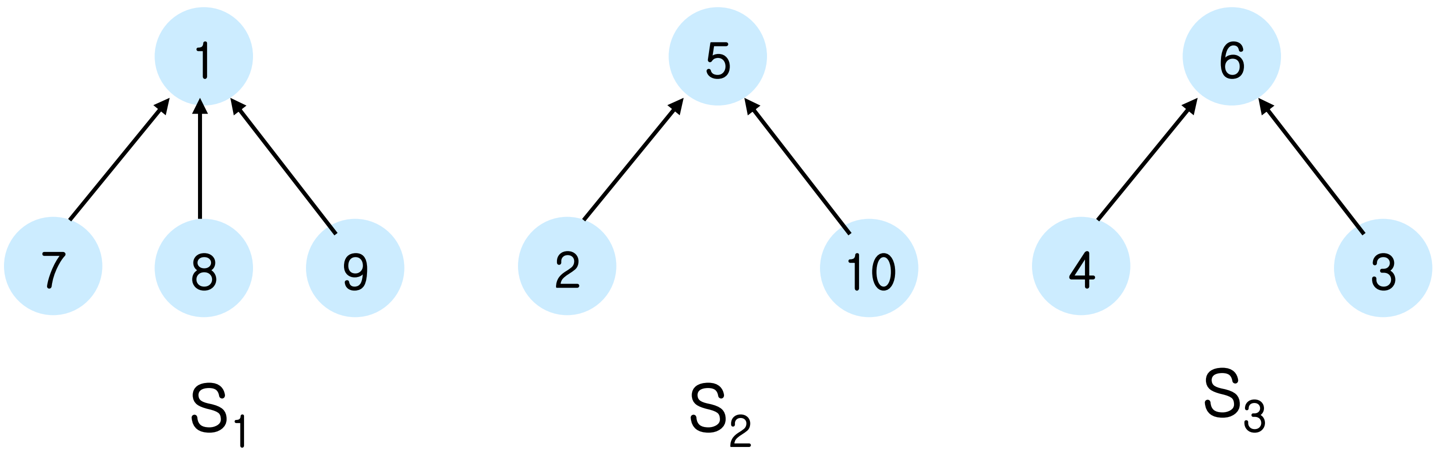
Tree representation: Example

- For 10 elements numbered 1 through 10,

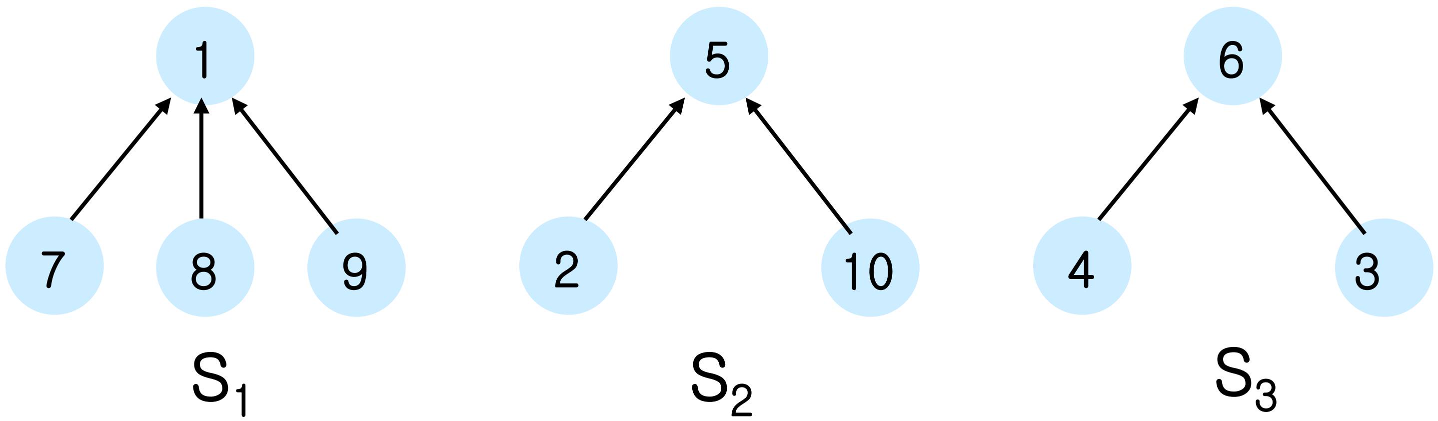
  $S_1 = \{1, 7, 8, 9\}$

  $S_2 = \{2, 5, 10\}$

  $S_3 = \{3, 4, 6\}$

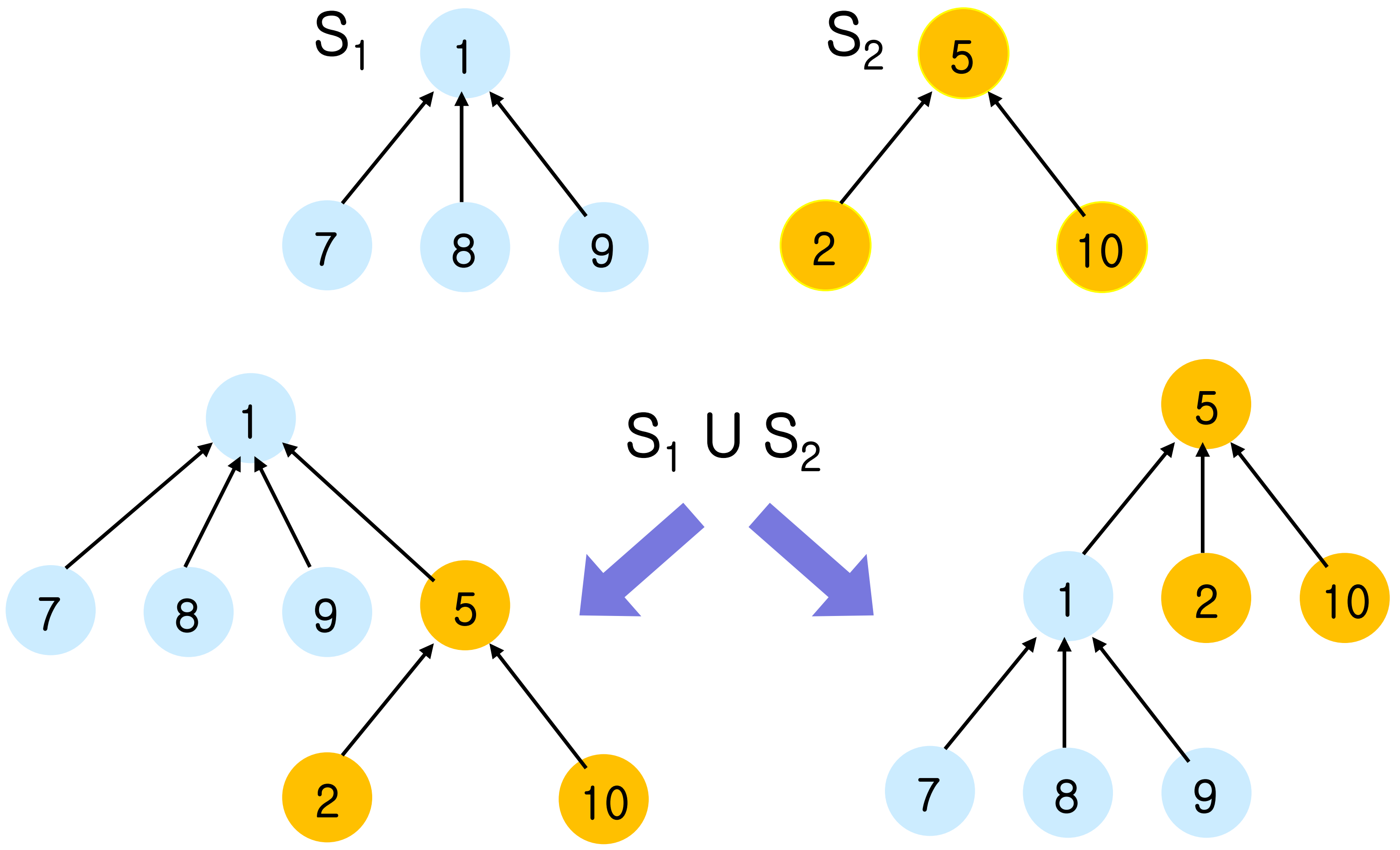


Operations: Find

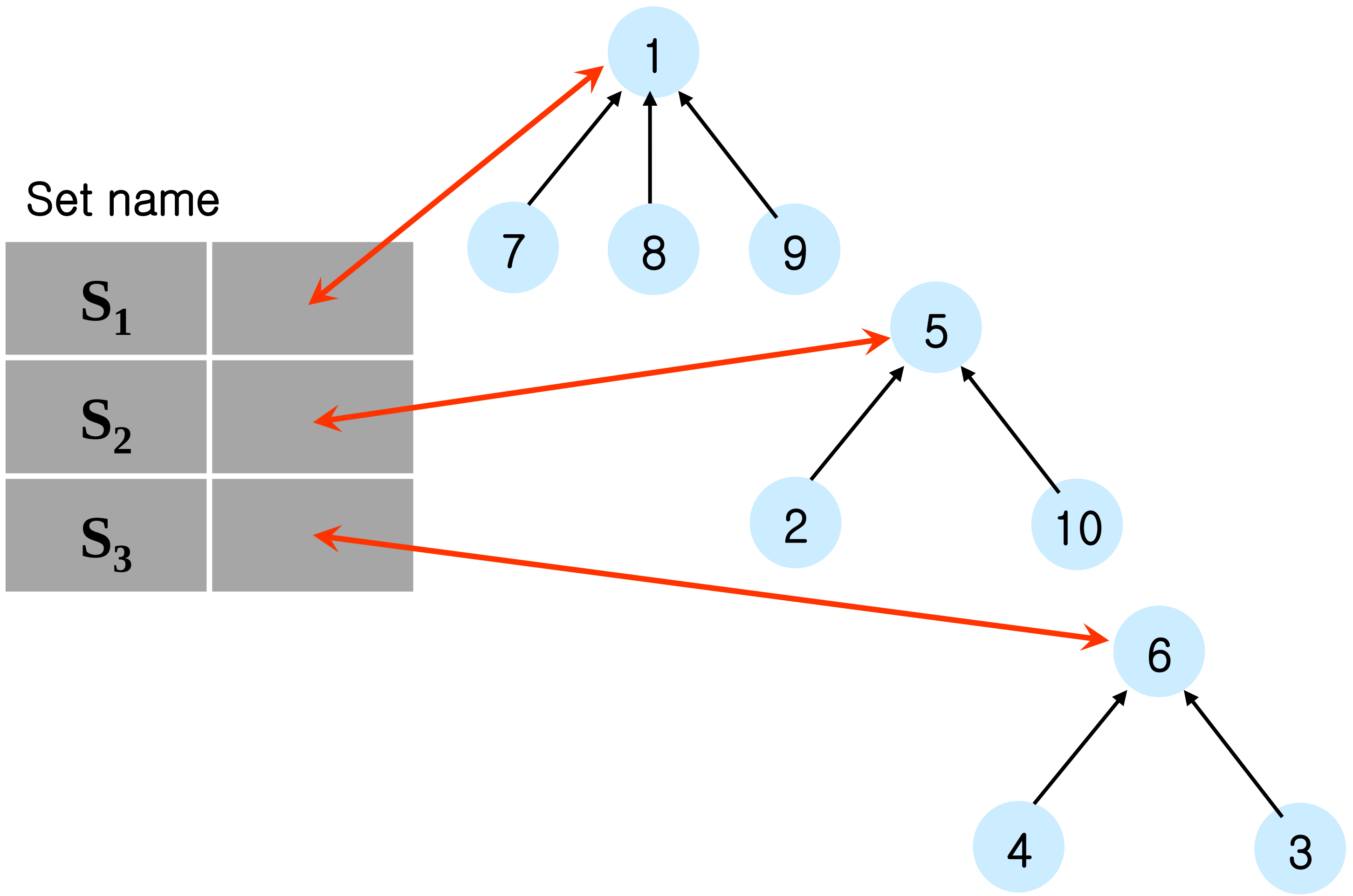


- Find(4) → S_3
- Find(9) → S_1

Operations: Union

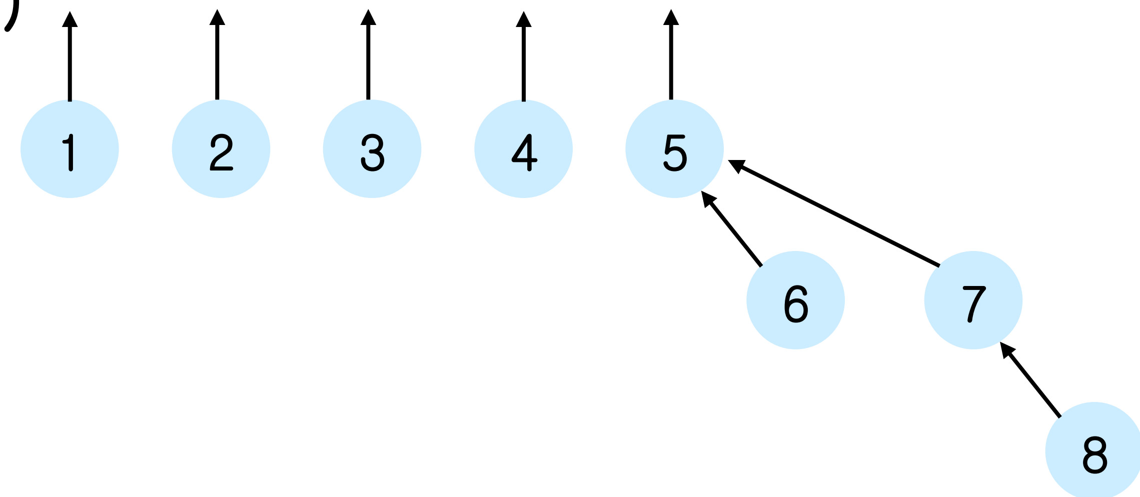


Set name



Union Strategy

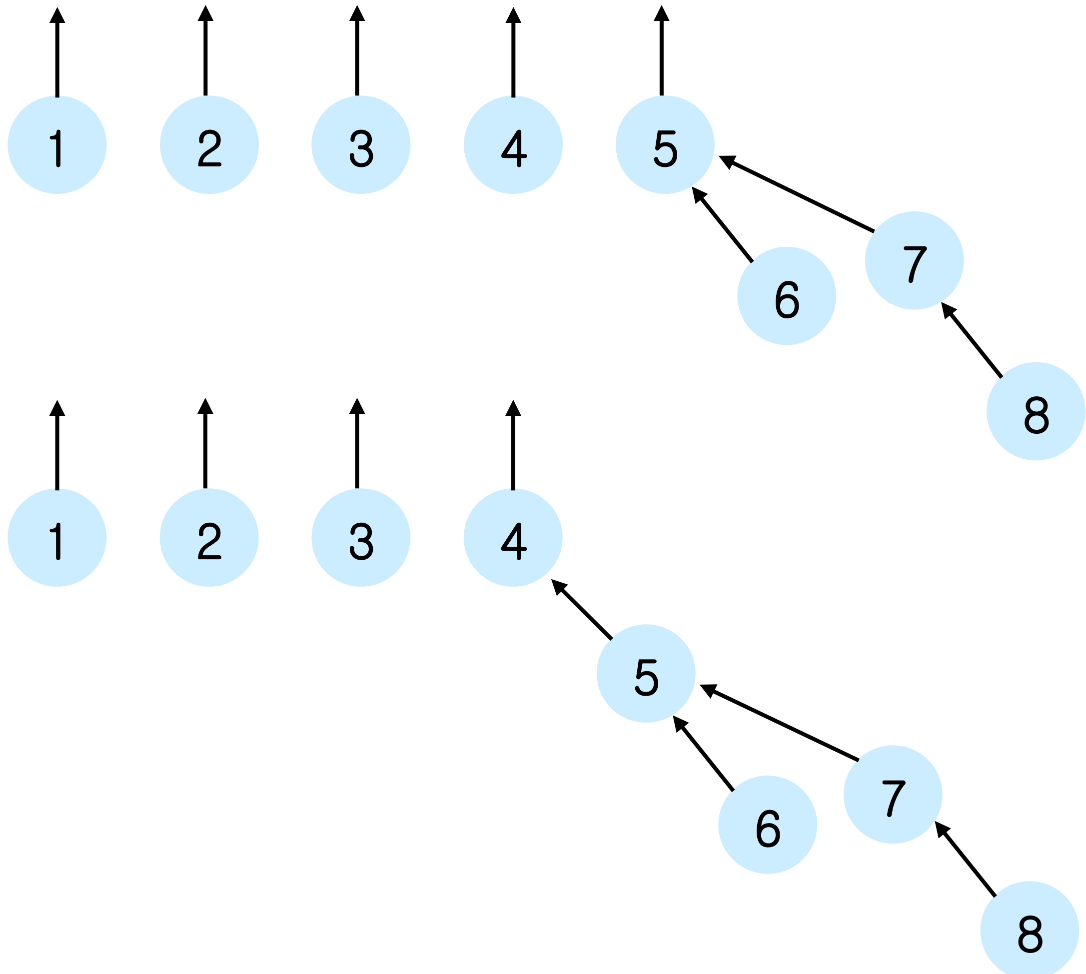
Union(5, 7)



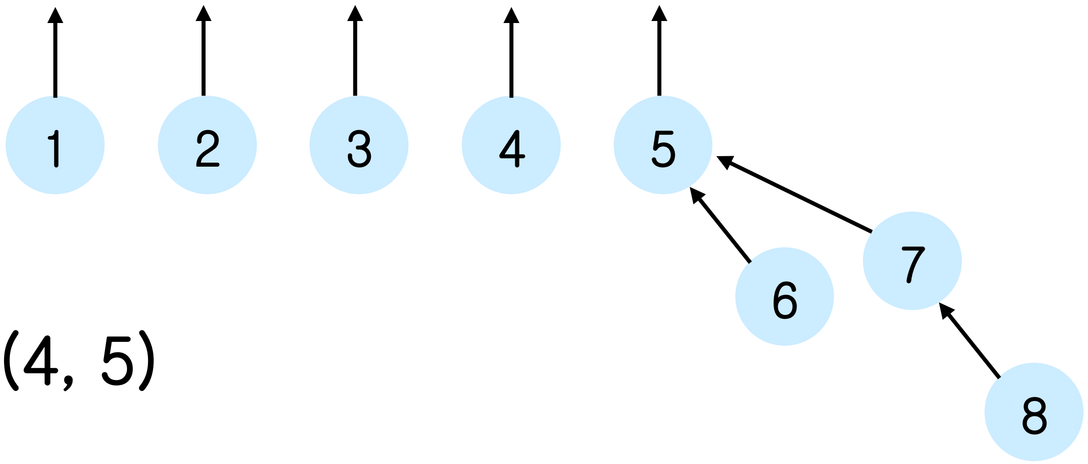
0	0	0	0	0	5	5	7
1	2	3	4	5	6	7	8

Union by previous rule

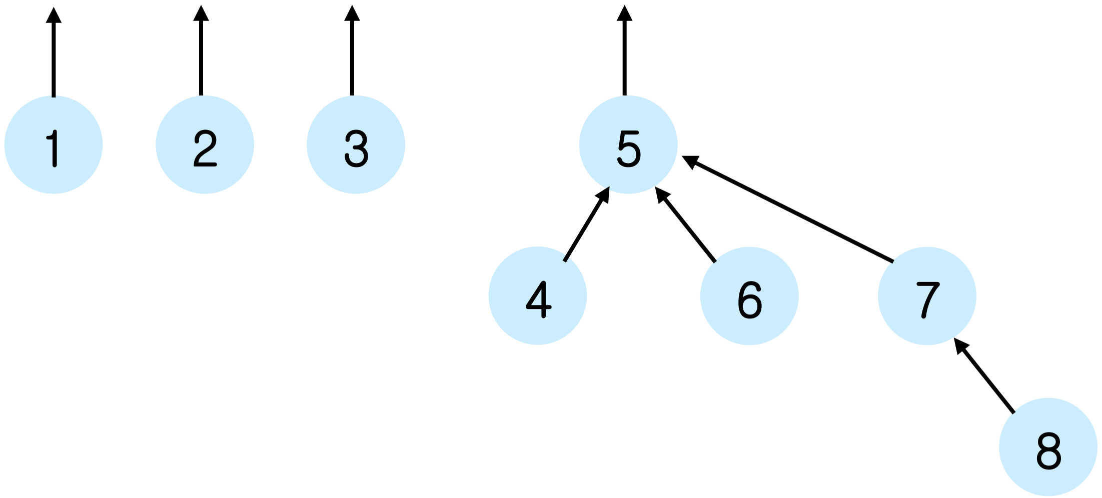
Union(4, 5)



Union by Size

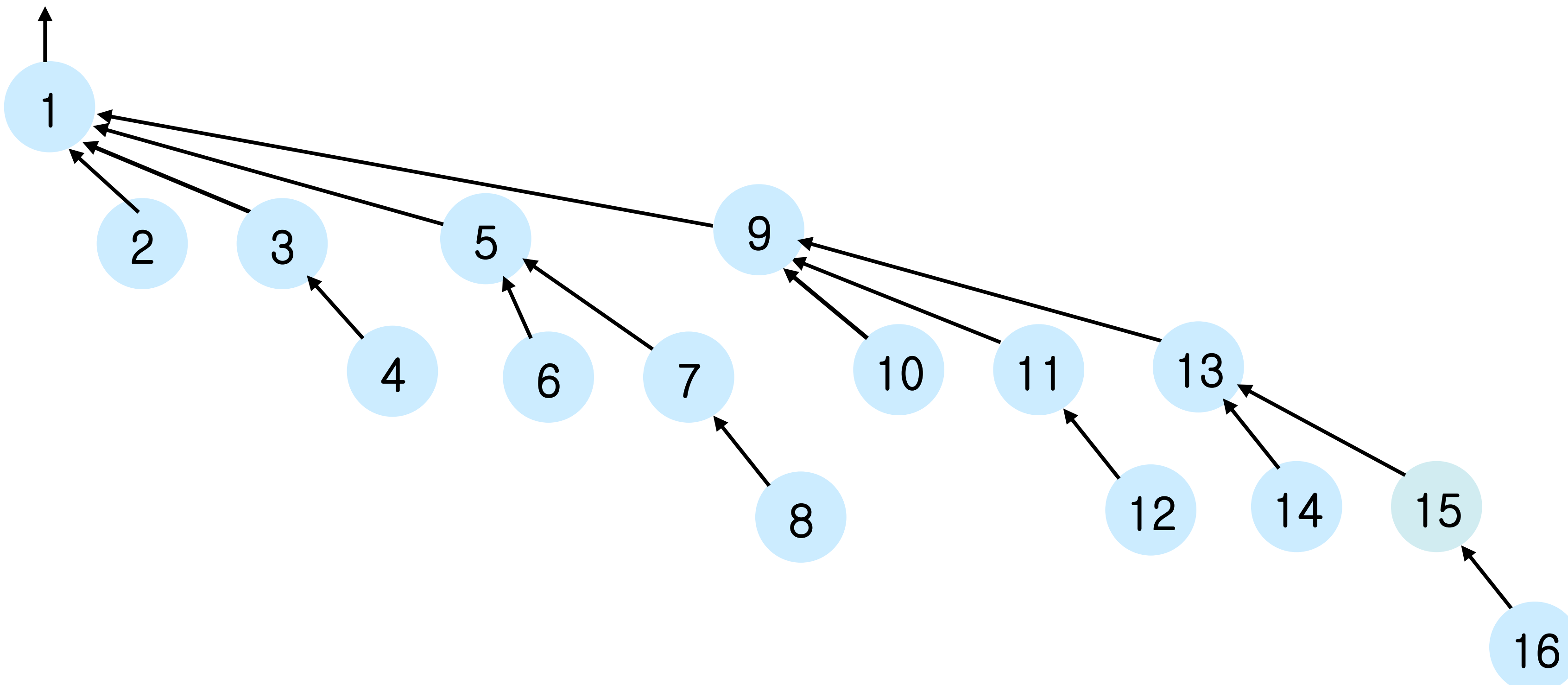


UnionBySize(4, 5)

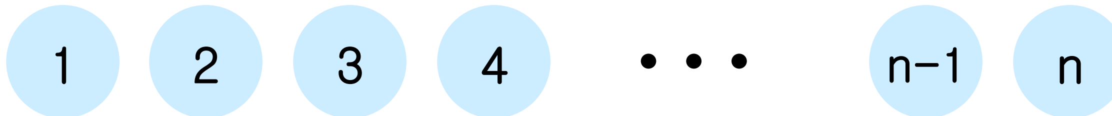


Worst case tree for N = 16

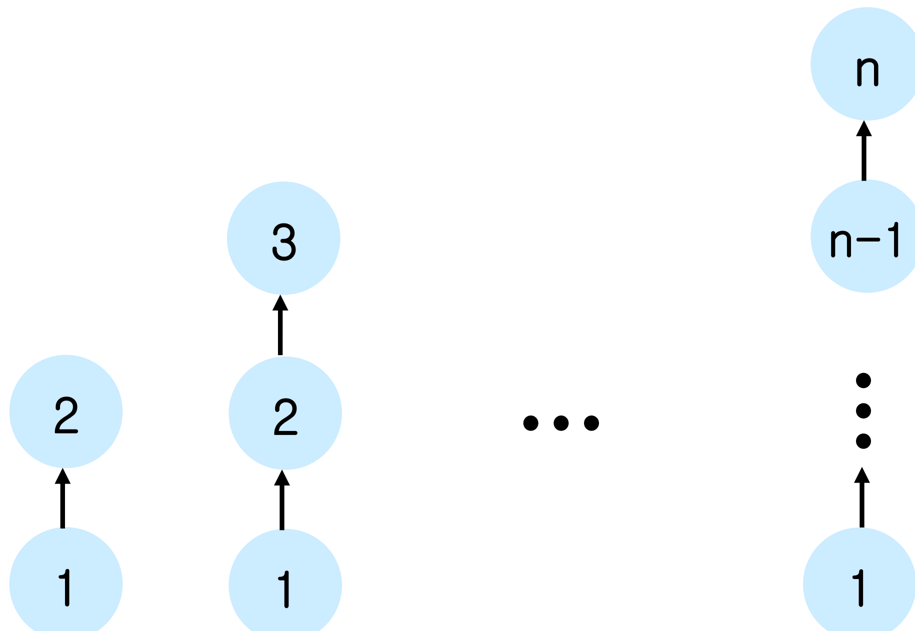
- If Unions are done by size, the depth of any node is never more than log *N*.



Performance



- U(2,1),F(1),U(3,2),F(1),U(4,3),F(1), …, F(1),U(n,n−1)



Performance

- All the n−1 unions take O(n): each one takes a constant time.

- The total time needed to process n−2 finds is

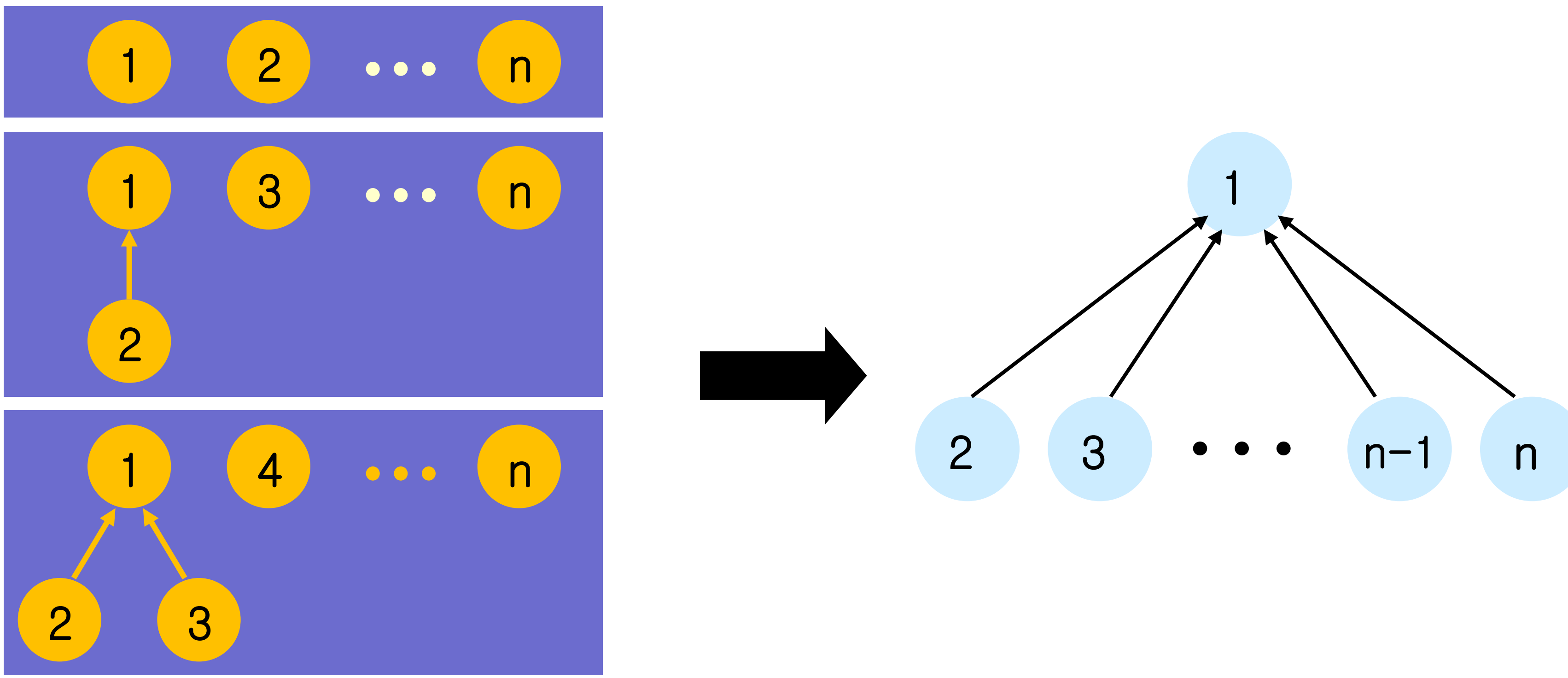
  $O(\sum_{i}^{n-2} i) = O(n^2)$

- How to avoid the worst case behavior

  → Use weighting rule

Weighting Rule for UNION (*i*, *j*):

If the number of nodes in (the height of) tree ***i*** is less than the number in (the height of) tree ***j***, then make ***j*** the parent of ***i***, otherwise make ***i*** the parent of ***j***.

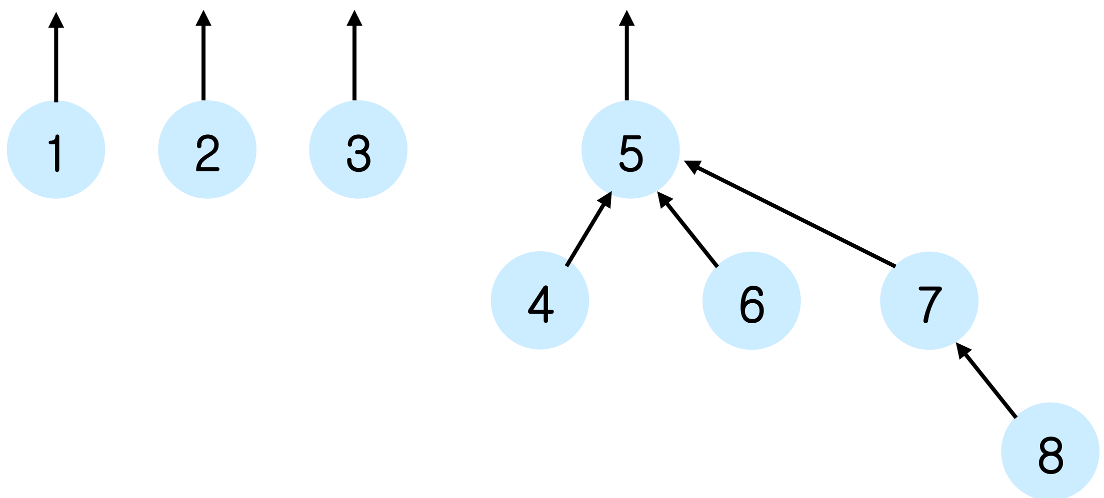


Weighting rules

- Using tree height
  - Make the shallow tree a subtree of the deeper tree
- Using tree size
  - Depending on the number of nodes in the tree
- How to store the number of nodes or height in a tree?

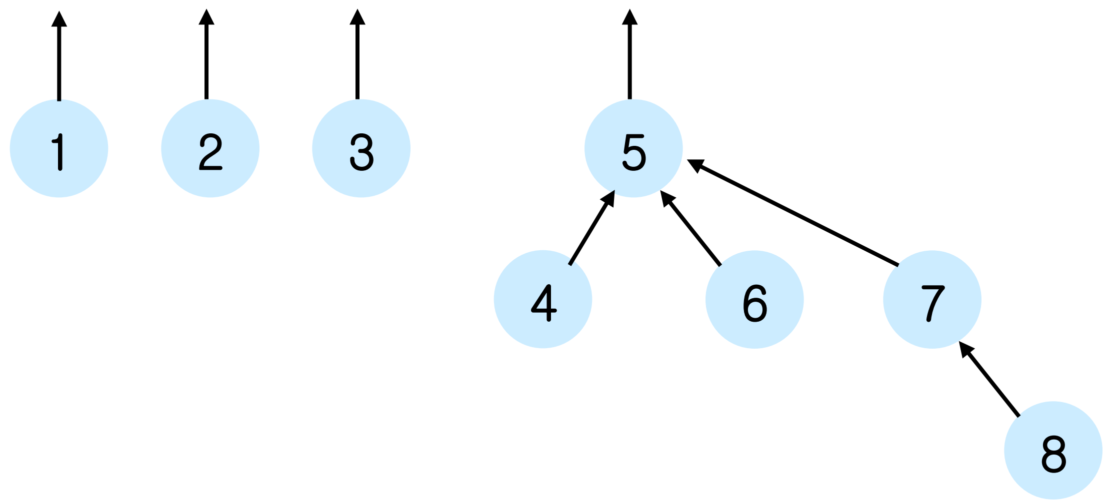
  → Use count field in the root of every tree

Ordinary Array Representation



0	0	0	5	0	5	5	7
1	2	3	4	5	6	7	8

Using Weight Rules



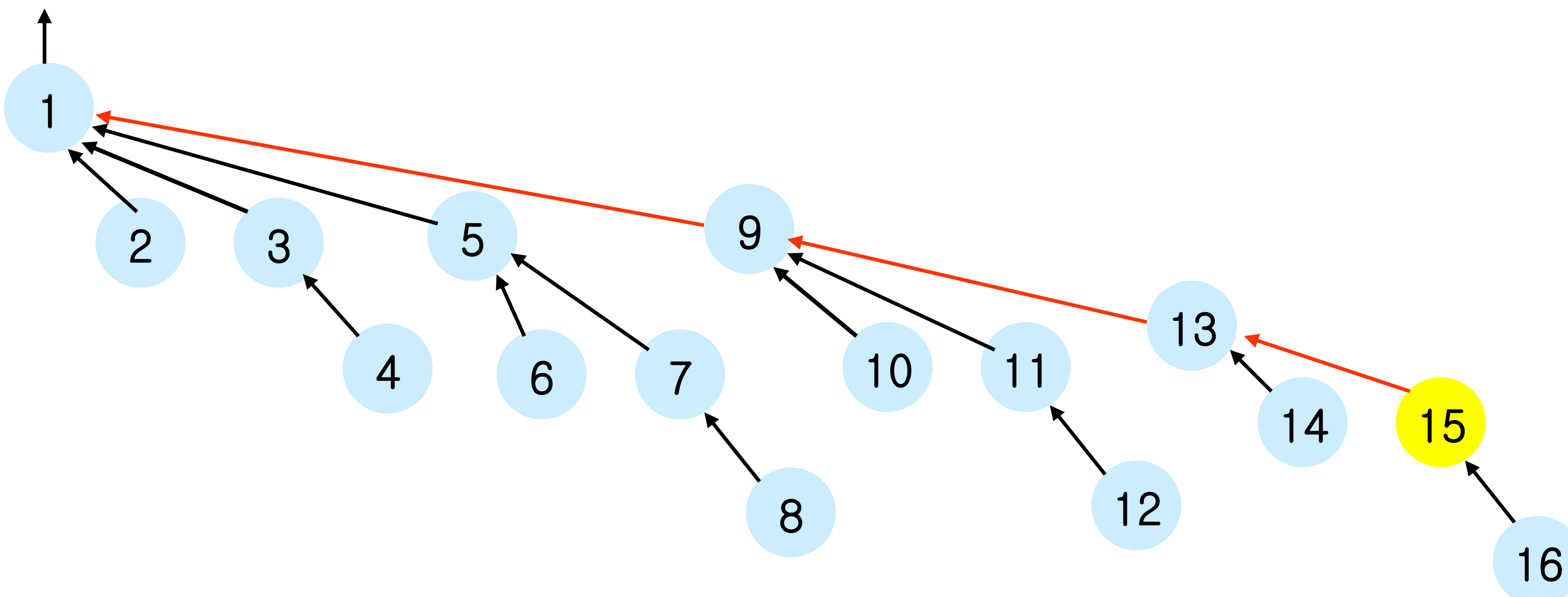
Union-by-size	−1	−1	−1	5	−5	5	5	7
	1	2	3	4	5	6	7	8
Union-by-height	0	0	0	5	−2	5	5	7
	1	2	3	4	5	6	7	8

Union by Height Algorithm

```
void SetUnion (DisjSet S, SetType R1, SetType R2)
{
   if (S[R2] < S[R1])              /* R2 is deeper set  */
        S[R1] = R2;                /* Make R2 new root */
   else
   {
        if (S[R1] == S[R2])   /* Same height,  */
             S[R1]--;         /* so update      */
        S[R2] = R1;
   }
}
```

Path Expression

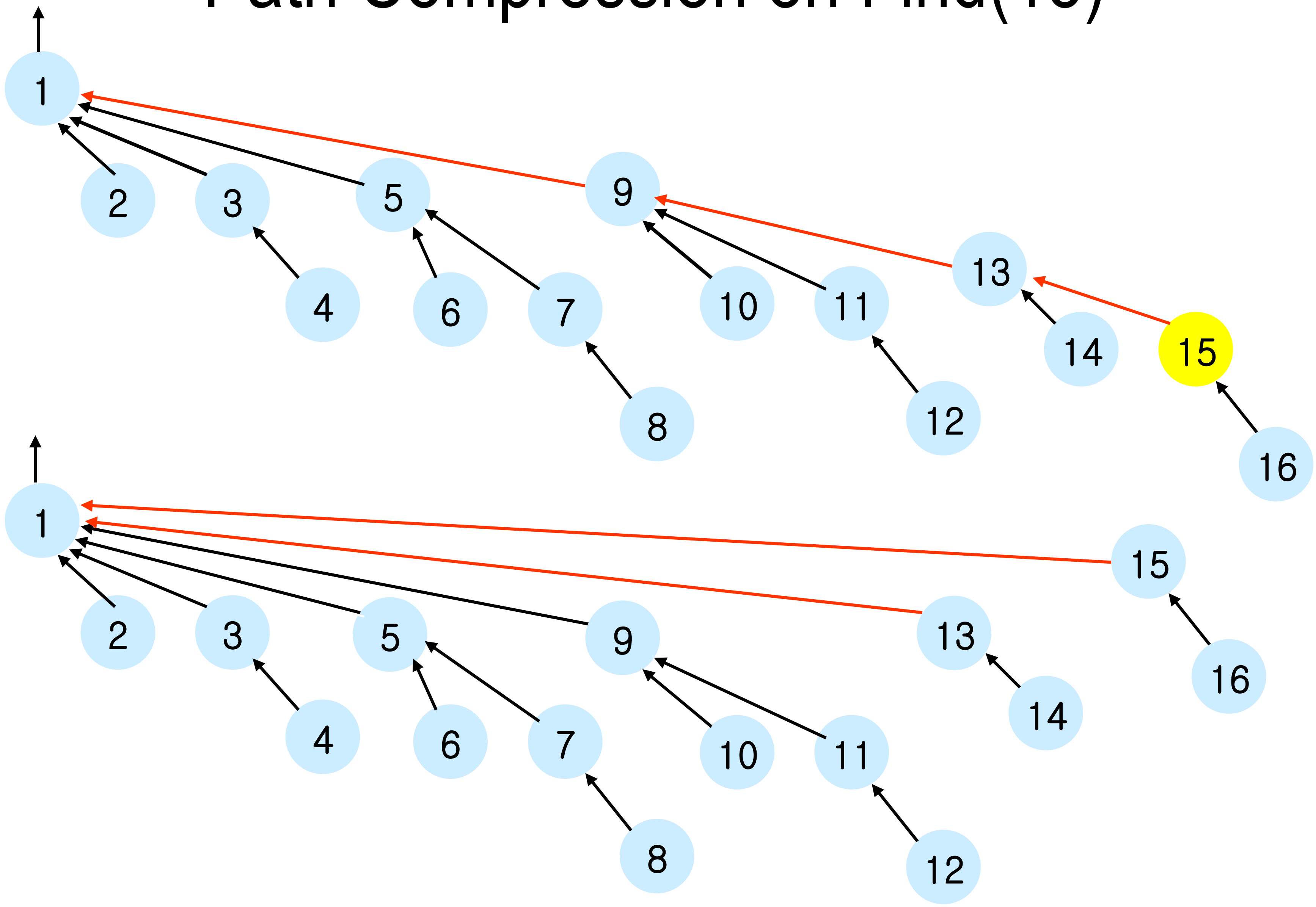
- When we put all set sets on a queue and repeatedly dequeue the first two sets and enqueue their union



Path Compression

- If there are many more ***Finds*** than ***Unions***, the running time is worse than that of the quick-find algorithm.
- The only way to speed up the algorithm without reworking the data structure entirely is to do something clever on the ***Find*** operation
- Useful when more ***Find***s are required
- Performed during the ***Find*** operation
- Every node on the path from ***X*** to the root has its parent changed to the root for ***Find***(***X***)

Path Compression on Find(15)



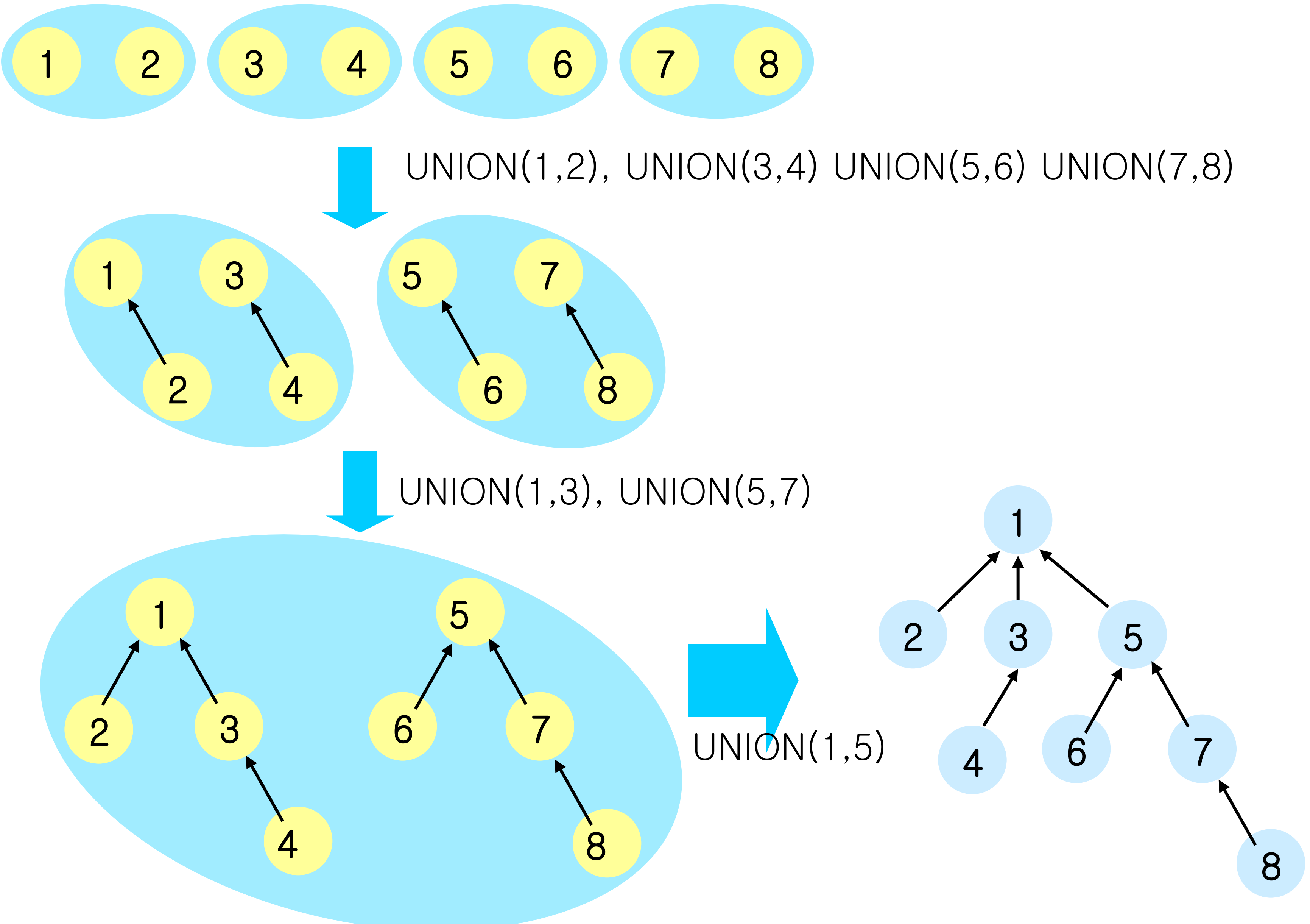
Revised FIND algorithm

```
SetType Find (ElementType X, DisjSet S)
{
   if ( S[X] <= 0 )
        return X ;
   else
        return S[X] = Find ( S[X], S ) ;
        return Find( S[X], S) // Original version
}
```

Lemma

(Lemma 1) Let T be a tree with n nodes created as a result of algorithm UNION. No node in T has level greater $\lfloor \log_2 n \rfloor + 1$

- As a result of lemma 1, the maximum time to process a find is at most O(log *n*) if there are *n* elements in a tree.
- Further improvement is possible using the *Collapsing Rule*.
- Collapsing Rule: If *j* is a node on the path from *i* to its root and PARENT(*j*) $\neq$ root(*i*), then set PARENT(*j*) $\leftarrow$ root(*i*)


1 2 3 4 5 6 7 8
UNION(1,2), UNION(3,4) UNION(5,6) UNION(7,8)
UNION(1,3), UNION(5,7)
UNION(1,5)