Introduction to Algorithms

Meng-Tsung Tsai

11/19/2019

Data Structures for Disjoint Sets

What is a data structure for disjoint sets?

It is a data structure that maintains a collection of disjoint dynamic sets $C = \{S_1, S_2, \ldots, S_t\}$.

(1) disjoint: $S_i \cap S_j = \varnothing$ for every $i \neq j$
(2) dynamic: S_i changes over time for every i

We need the data structure to support the following operations:

(1) Make-Set(x)
(2) Union(x, y)
(3) Find-Set(x)

Make-Set(x) operation

Make-Set(x) operation creates a singleton set $\{x\}$.

Example.

Let $C = \{S_1 = \{a, b\}, S_2 = \{c, d, e\}\}$.

After Make-Set(f), we have

$C = \{S_1 = \{a, b\}, S_2 = \{c, d, e\}, S_3 = \{f\}\}$.

Union(x, y) operation

Union(x, y) unites the sets that contain x and y, say S_x and S_y, into a new set that is a union of these two sets.

Example.

Let $C = \{S_1 = \{a, b\}, S_2 = \{c, d, e\}, S_3 = \{f\}\}$.

After Union(b, d), we have

$C = \{S_1 = \{a, b, c, d, e\}, S_2 = \{f\}\}$.

Find-Set(x) operation

Find-Set(x) returns the representative element of the set that contains x.

Example.

Let $C = \{S_1 = \{a, b\}, S_2 = \{c, d, e\}, S_3 = \{f\}\}$. Our algorithm picks an arbitrary element as the representative for each set S_i.

Find-Set(a) returns b.
Find-Set(b) returns b.
Find-Set(e) returns d.
Find-Set(f) returns f.

Find-Set(x) operation

Find-Set(x) returns the representative element of the set that contains x.

Example.

Let $C = \{S_1 = \{a, b\}, S_2 = \{c, d, e\}, S_3 = \{f\}\}$. Our algorithm picks an arbitrary element as the representative for each set S_i.

Find-Set(a) returns b.
Find-Set(b) returns b.
Find-Set(e) returns d.
Find-Set(f) returns f.

If S_i doesn't change, the representative element of S_i should be fixed.

Applications of Disjoint-Set Data Structures

Computing the connected components

Let G = (V, E) be an undirected graph.

```
Connected-Component(G){
    foreach node x in V{
        Make-Set(x);
    }
    foreach edge (u, v) in E{
        if(Find-Set(u) ≠ Find-Set(v)){
            Union(u, v);
        }
    }
}
```

Computing the connected components

Example.

G = (V = {1, 2, 3, 4, 5, 6, 7}, E = {{1, 2}, {1, 3}, {2, 3}, {5, 6}}).

Initially, C = {{1}, {2}, {3}, {4}, {5}, {6}, {7}}.

If (Find-Set(1) $\neq$ Find-Set(2)) Union(1, 2);

After which, C = {{1, 2}, {3}, {4}, {5}, {6}, {7}}.

Computing the connected components

Example.

G = (V = {1, 2, 3, 4, 5, 6, 7}, E = {{1, 2}, {1, 3}, {2, 3}, {5, 6}}).

C = {{1, 2}, {3}, {4}, {5}, {6}, {7}}.

If (Find-Set(1) $\neq$ Find-Set(3)) Union(1, 3);

After which, C = {{1, 2, 3}, {4}, {5}, {6}, {7}}.

Computing the connected components

<u>Example</u>.

$G = (V = \{1, 2, 3, 4, 5, 6, 7\}, E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{5, 6\}\})$.

$C = \{\{1, 2, 3\}, \{4\}, \{5\}, \{6\}, \{7\}\}$.

If (Find-Set(2) = Find-Set(3)) do nothing;

After which, $C = \{\{1, 2, 3\}, \{4\}, \{5\}, \{6\}, \{7\}\}$.

Computing the connected components

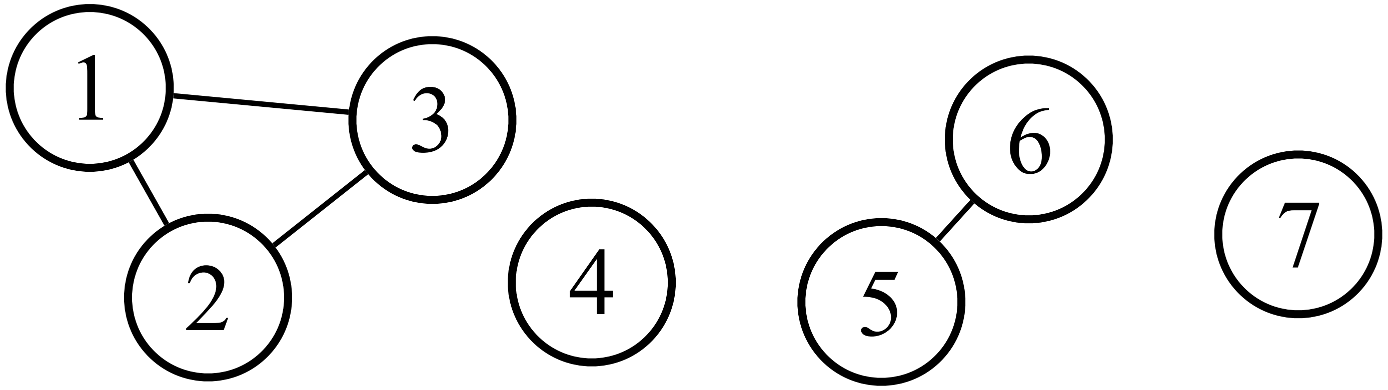
Example.

$G = (V = \{1, 2, 3, 4, 5, 6, 7\}, E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{5, 6\}\})$.

$C = \{\{1, 2, 3\}, \{4\}, \{5\}, \{6\}, \{7\}\}$.

If (Find-Set(5) $\neq$ Find-Set(6)) Union(5, 6);

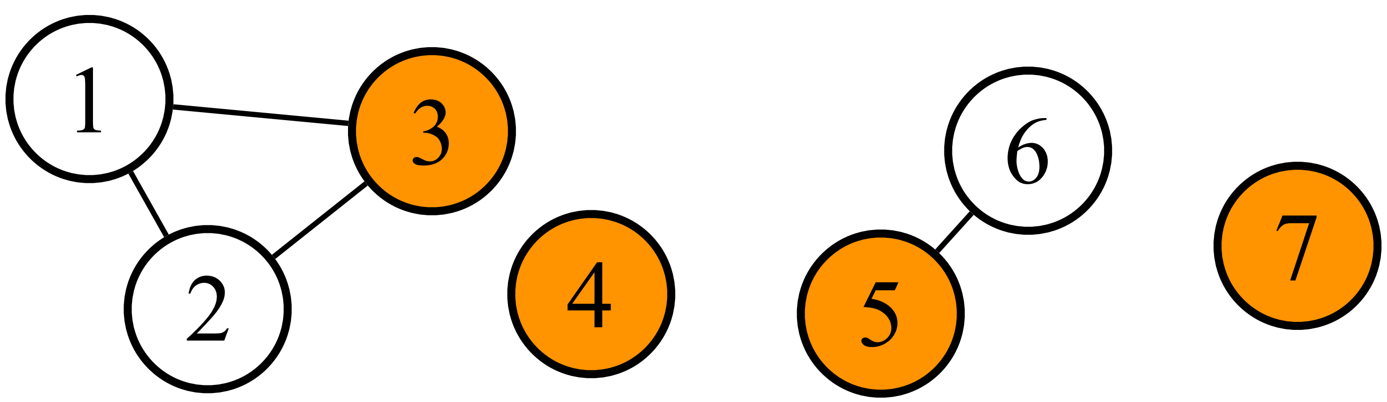
After which, $C = \{\{1, 2, 3\}, \{4\}, \{5, 6\}, \{7\}\}$.



4 connected components.

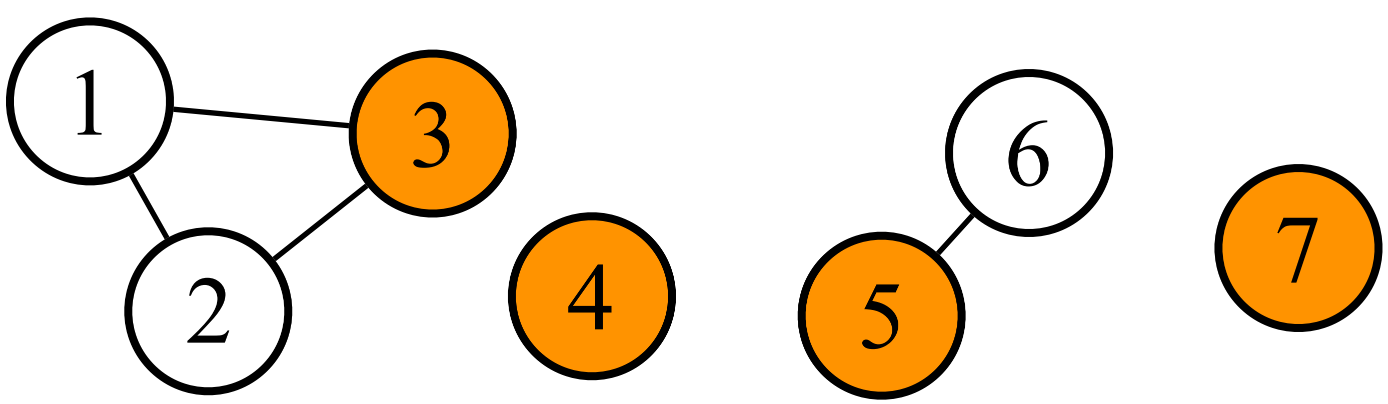
Deciding whether two nodes are in the same connected component

```
Same-Component(u, v){
   if(Find-Set(u) = Find-Set(v)){
      return True;
   }else{
      return False;
   }
}
```



Deciding whether two nodes are in the same connected component

```
Same-Component(u, v){
   if(Find-Set(u) = Find-Set(v)){
      return True;
   }else{
      return False;
   }
}
```

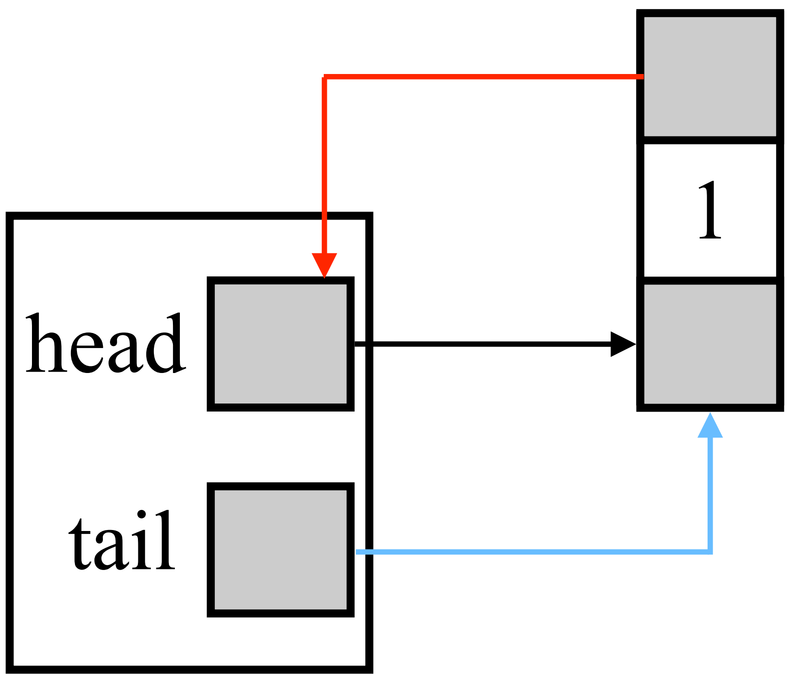


Same-Component(1, 2) returns True because Find-Set(1) = 3 and Find-Set(2) = 3.

Linked-list Representation of Disjoint Sets

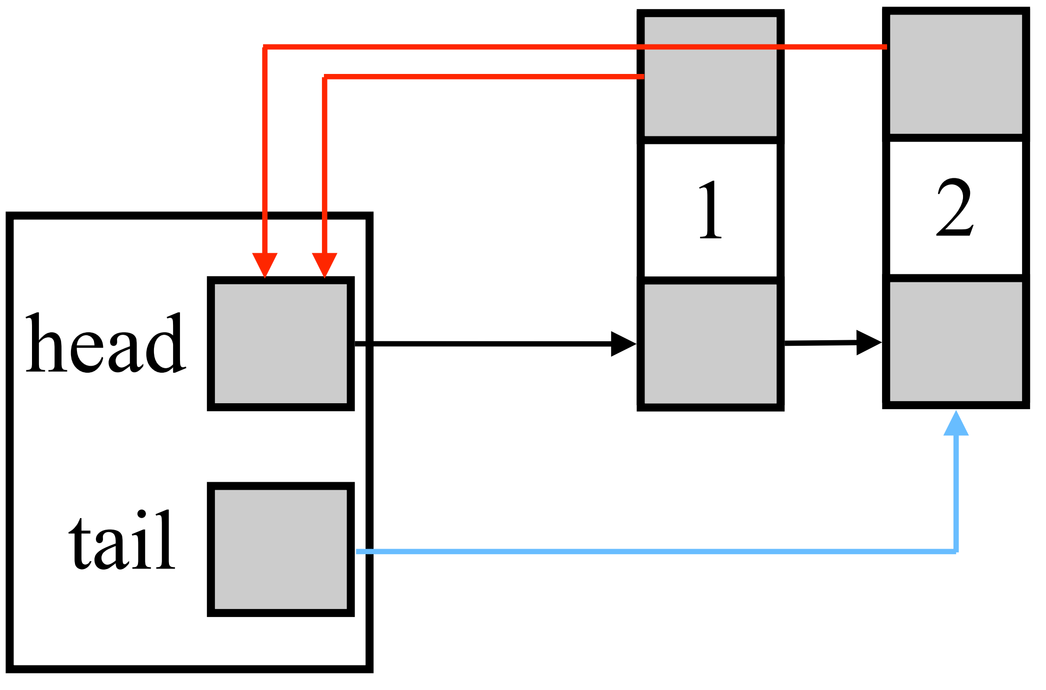
Implementation by linked-lists

Make-Set(1): // O(1) time



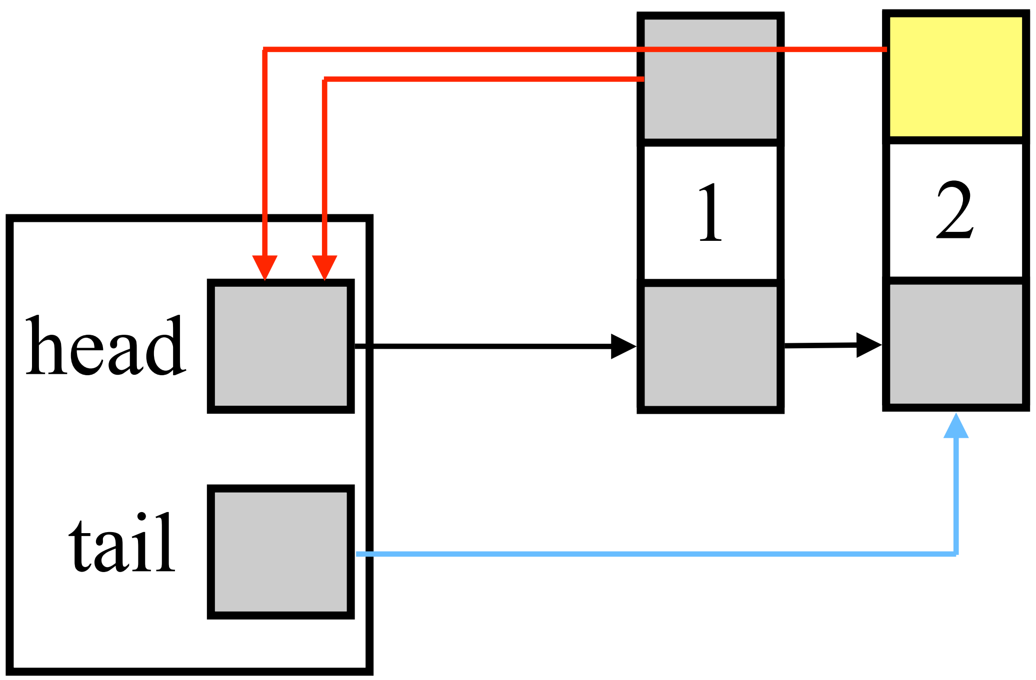
Implementation by linked-lists

Find-Set(2): // O(1) time. Let the first element on the linked list be the representative.



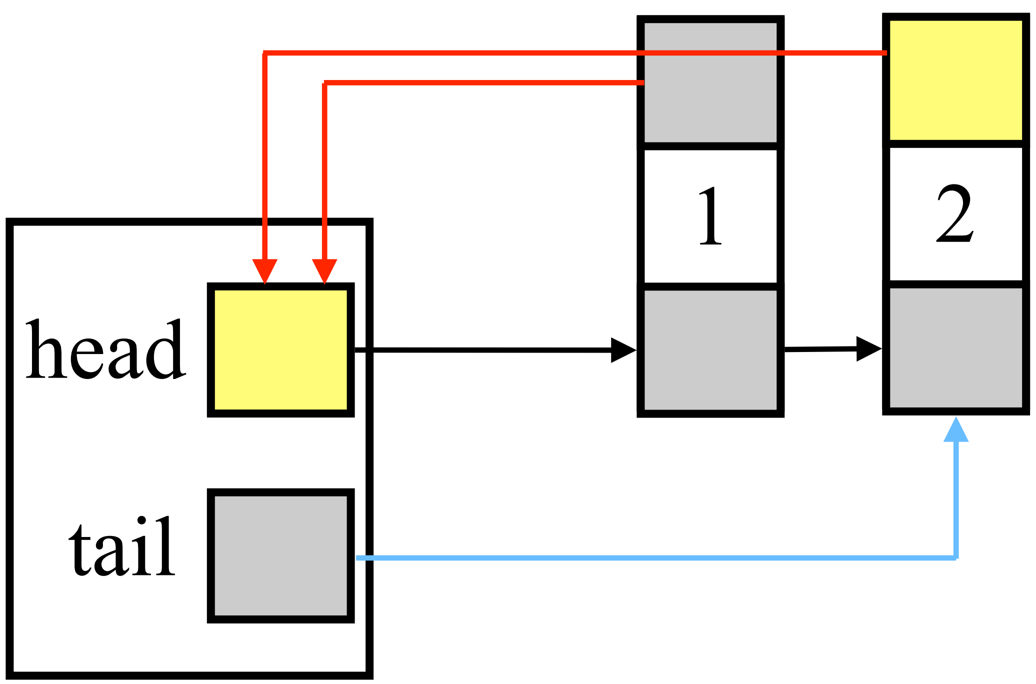
Implementation by linked-lists

Find-Set(2): // O(1) time. Let the first element on the linked list be the representative.



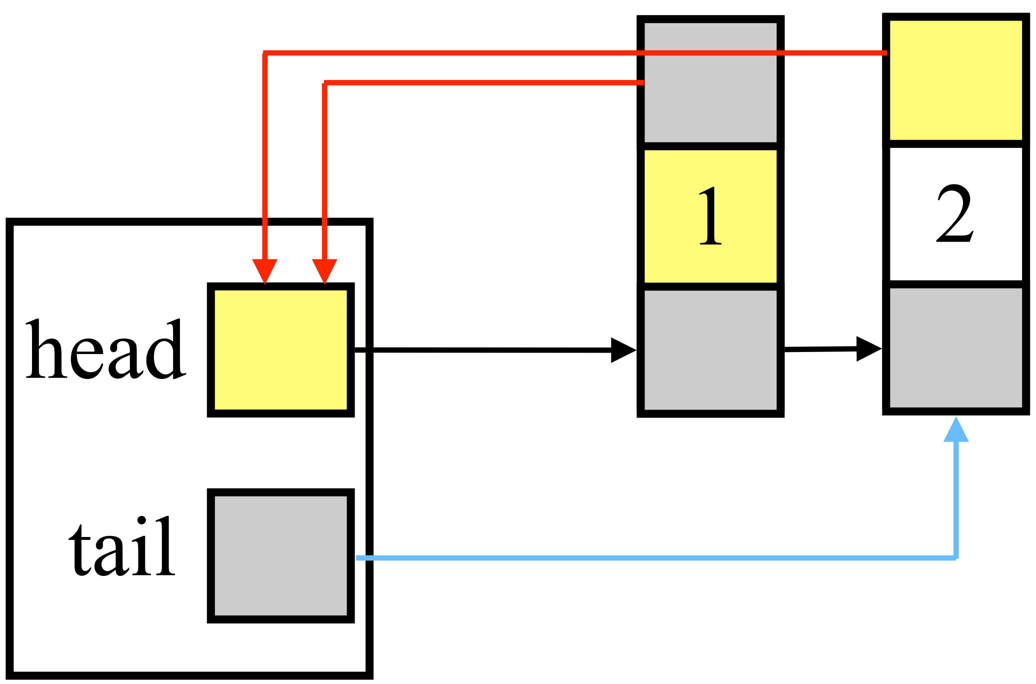
Implementation by linked-lists

Find-Set(2): // O(1) time. Let the first element on the linked list be the representative.



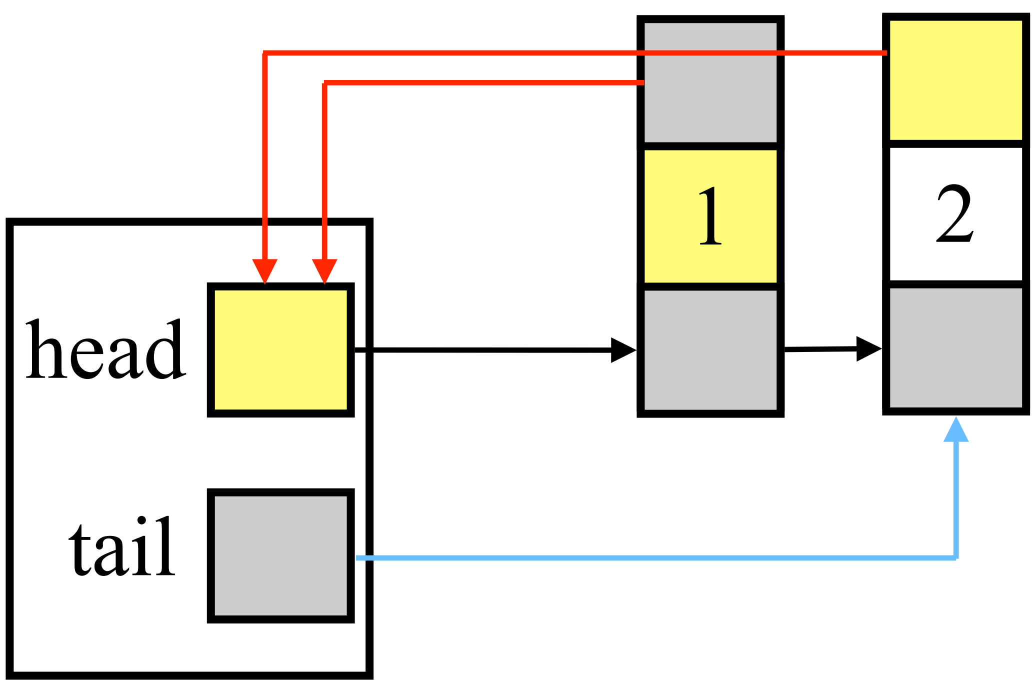
Implementation by linked-lists

Find-Set(2): // O(1) time. Let the first element on the linked list be the representative.



Implementation by linked-lists

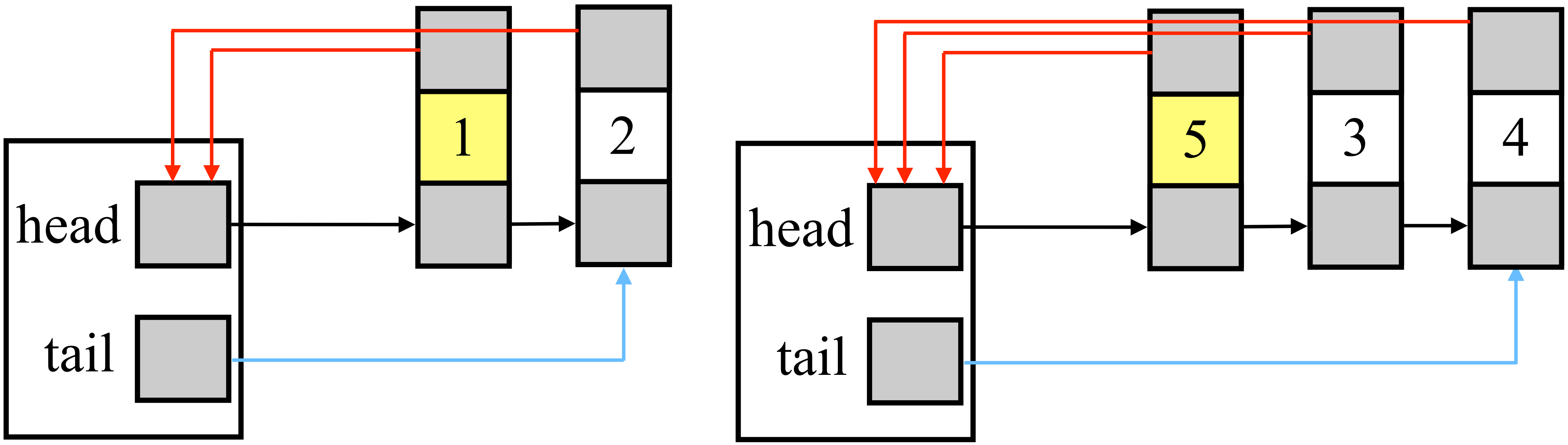
Find-Set(2): // O(1) time. Let the first element on the linked list be the representative.



Find-Set(2) returns 1. It takes O(1) steps.

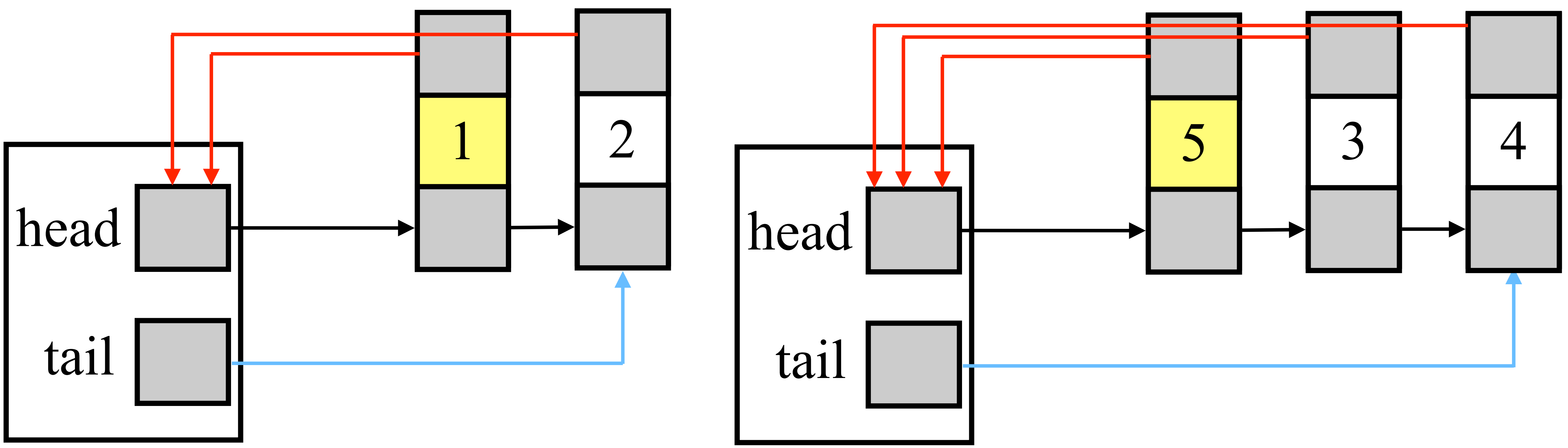
Implementation by linked-lists

Union(2, 4): // O(length of one list)



Implementation by linked-lists

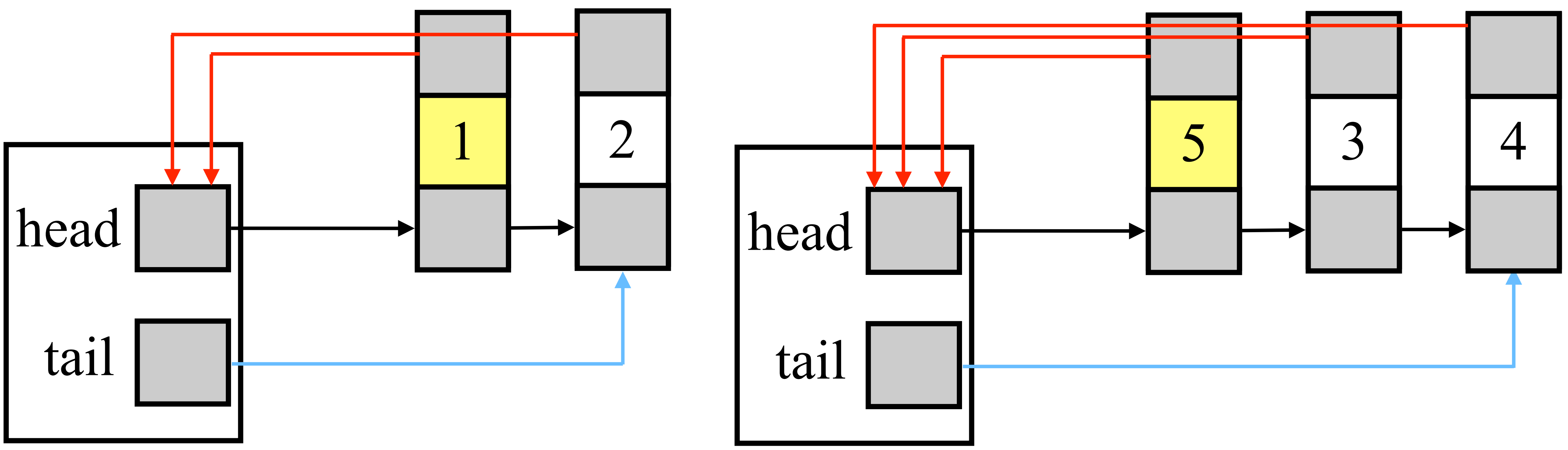
Union(2, 4): // O(length of one list)



If(Find-Set(2) $\neq$ Find-Set(4))
    Link(Find-Set(2), Find-Set(4));

Implementation by linked-lists

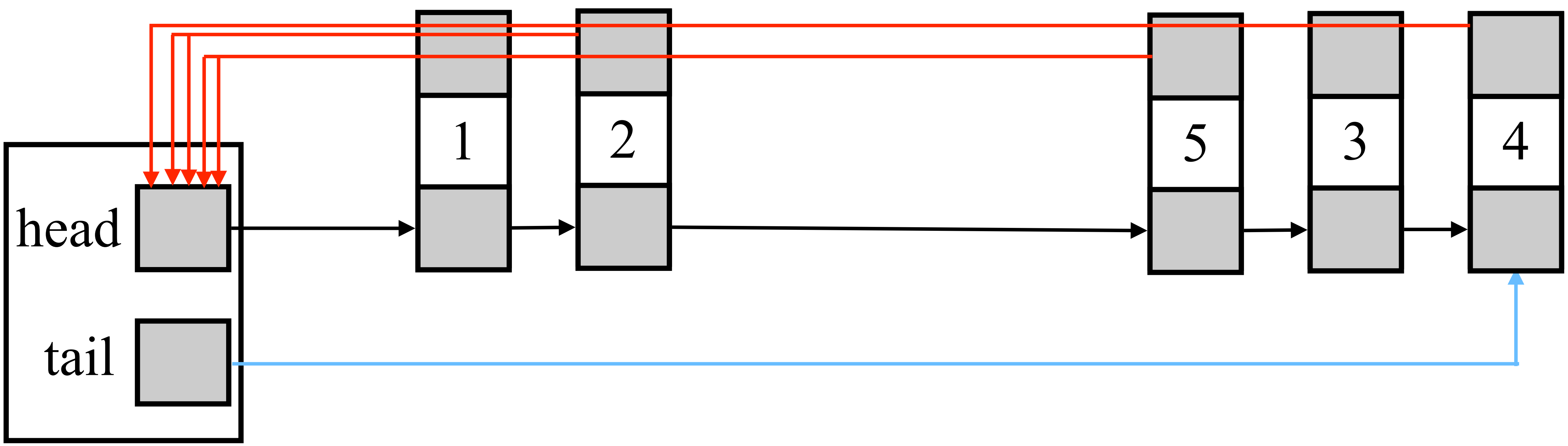
Union(2, 4): // O(length of one list)



It needs to modify all the head-pointers of one list to the new head.

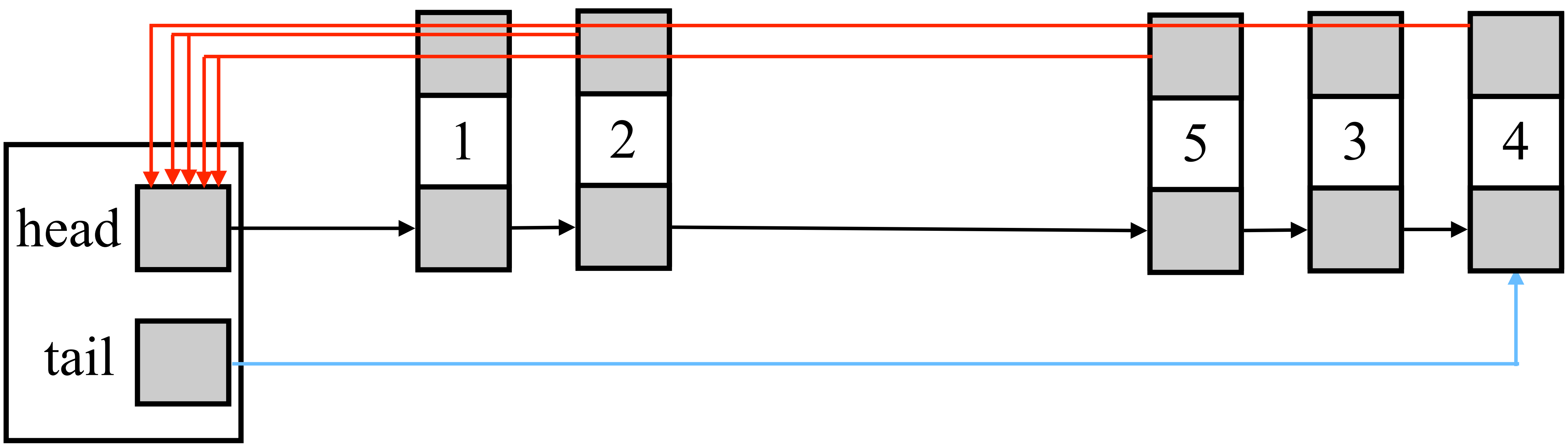
Implementation by linked-lists

Union(2, 4): // O(length of one list)



Implementation by linked-lists

Union(2, 4): // O(length of one list)



Alternatively, we could append the shorter list to the longer one, which we call weighted-union heuristic.

Running time of using linked-list representation and weighted-union heuristic

Claim. It takes $O(n \log n + m)$ running time to perform any sequence of m Make-Set(x), Find-Set(x), Union(x, y) operations, in which there are n Make-Set(x) operations.

Proof. Every operation runs in $O(1)$ time, except that a Union(x, y) operation may update $\omega(1)$ head-pointers.

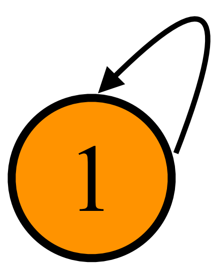
For each element x, every time we link its head-pointer to the new head, the length of x's list is doubled (because x is on the shorter list). Hence, in all $(n-1)$ Union operations, x's head-pointer is updated at most $O(\log n)$ times.

In total, the head-pointers of the n elements are updated at most $O(n \log n)$ times. We thus get the $O(m+n \log n)$ bound.

Forest Representation of Disjoint Sets

Implementation by forests

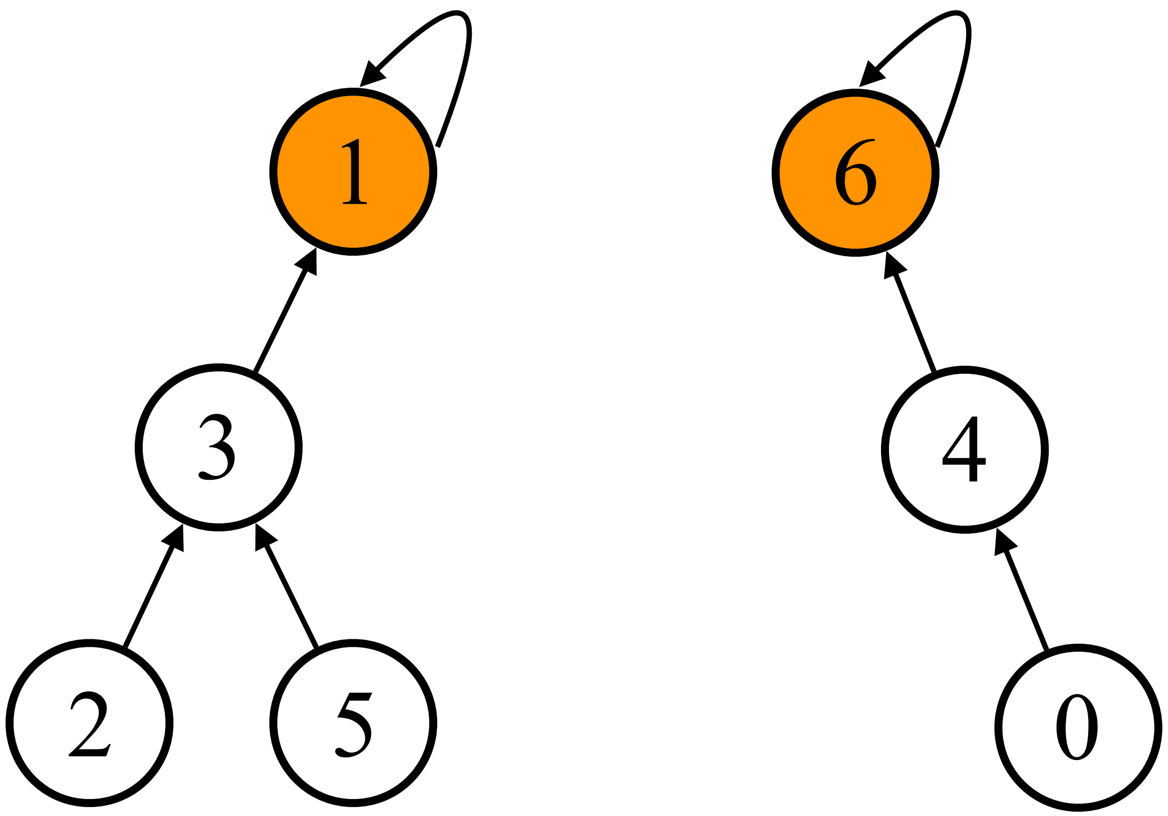
Make-Set(1): // O(1) time




```
Make-Set(x){
    x.parent = x;
    x.rank = 0; // x.rank is an
    inaccurate upper bound of
    the height of x (# edges in
    the longest simple path
    between x and any
    descendant leaf)
}
```

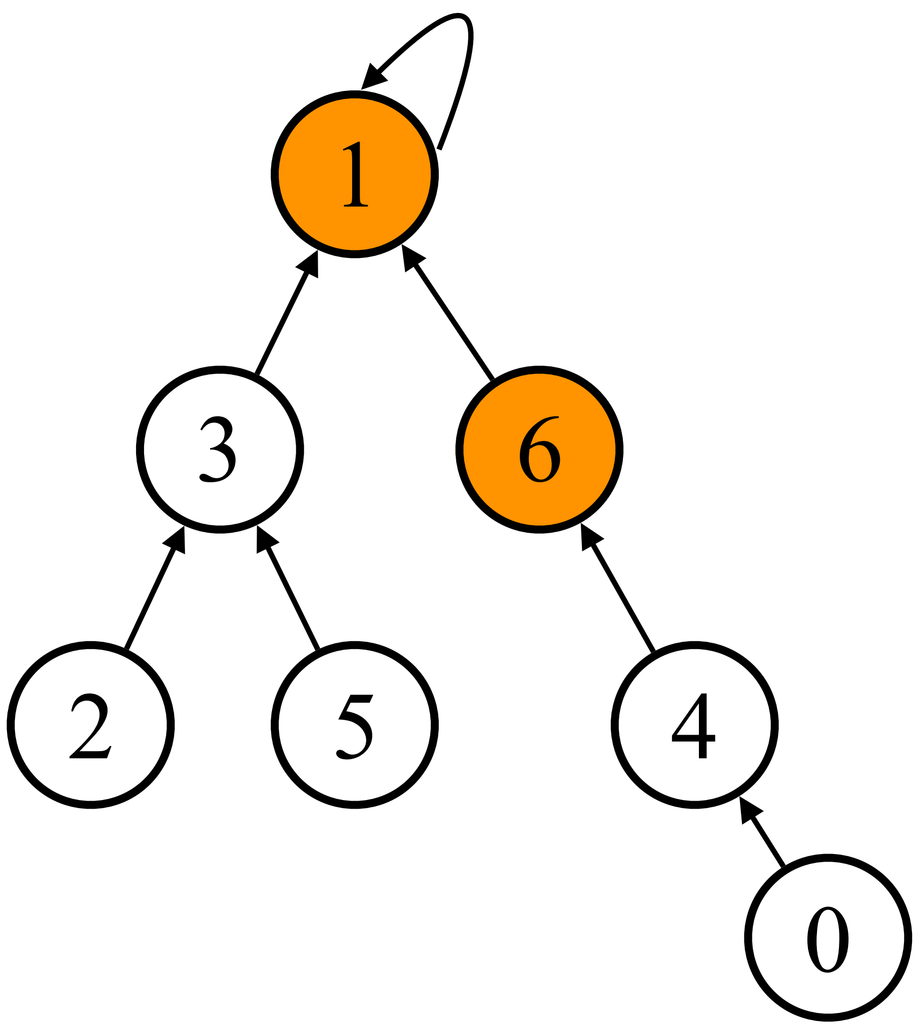
Implementation by forests

Union(2, 4): // O(1)



Implementation by forests

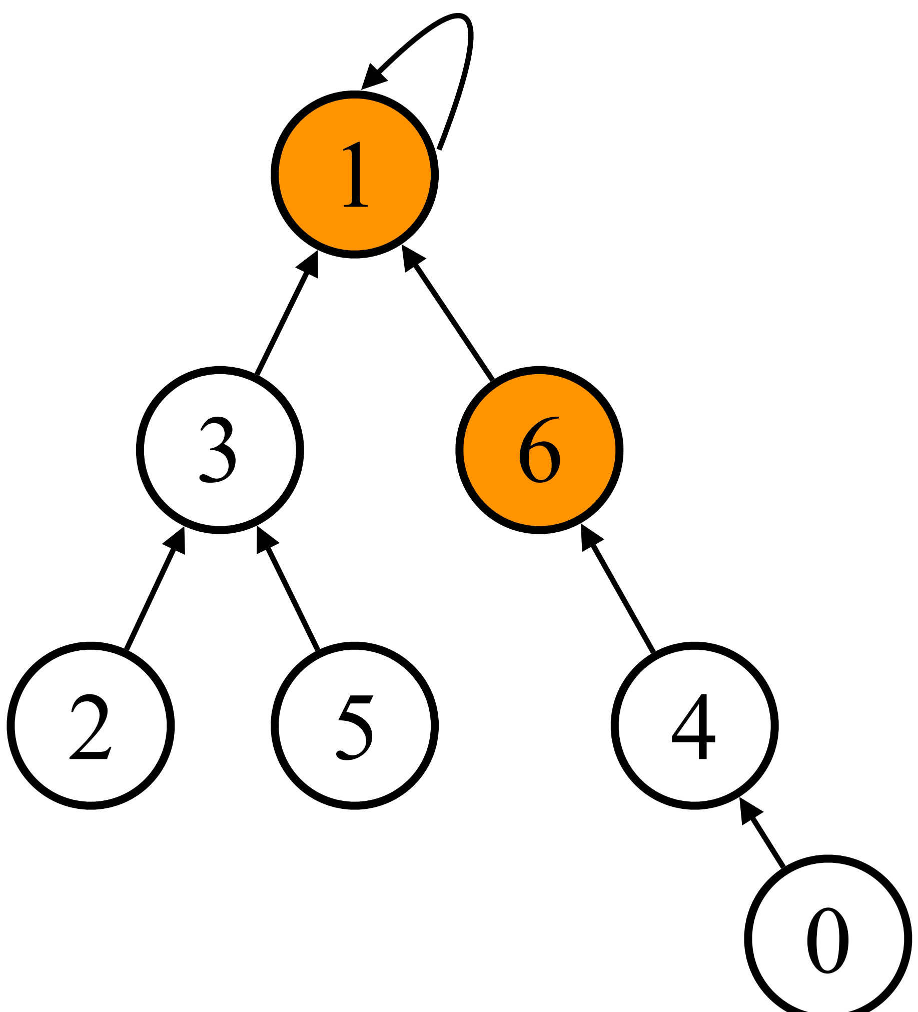
Union(2, 4): // O(1)




```
Union(x, y){
    if(Find-Set(x) ≠ Find-Set(y)){
        Link(Find-Set(x), Find-Set(y));
    }
}
Link(x, y){
    if(x.rank > y.rank){
        y.parent = x;
    }else{
        x.parent = y;
        if(x.rank equals y.rank)
            y.rank ← y.rank + 1;
    }
}
```

Implementation by forests

Union(2, 4): // O(1)

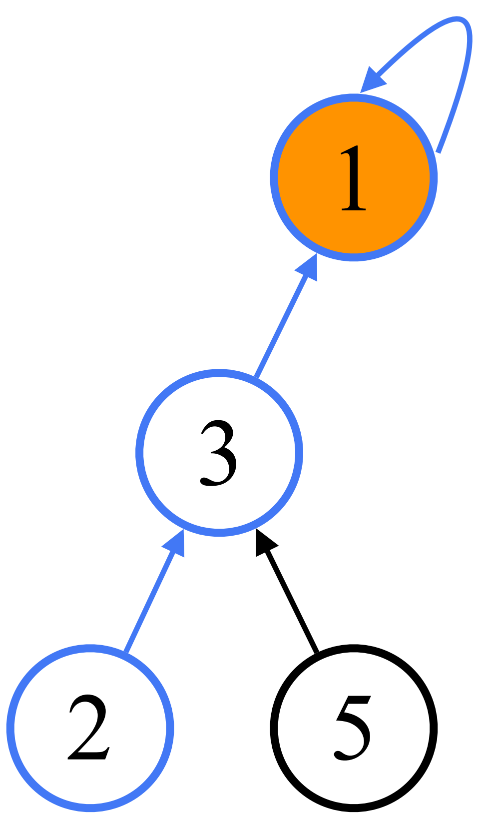



```
Union(x, y){
    if(Find-Set(x) ≠ Find-Set(y)){
        Link(Find-Set(x), Find-Set(y));
    }
}
Link(x, y){
    if(x.rank > y.rank){
        y.parent = x;
    }else{
        x.parent = y;
        if(x.rank equals y.rank)
            y.rank ← y.rank + 1;
    }
}
```

Similarly, we could append the shorter tree to the taller one, which we call union by rank.

Implementation by forests

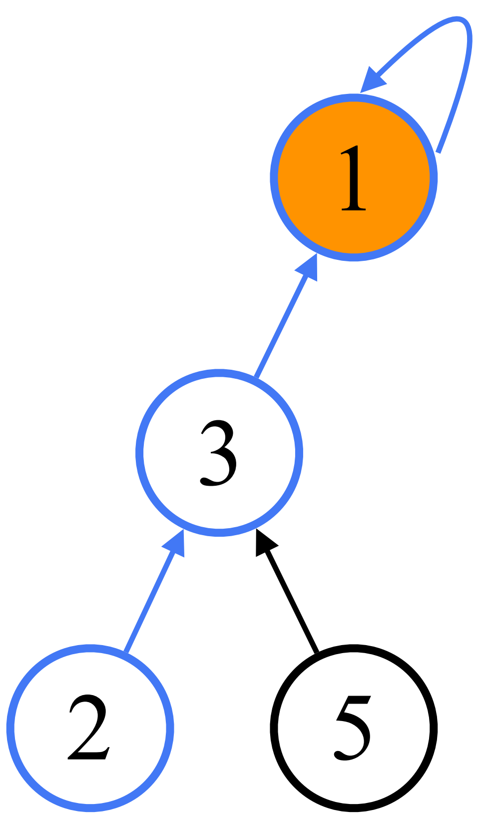
Find-Set(2): // O(height of the tree) time




```
Find-Set(x){
    while(x.parent ≠ x){
        x = x.parent;
    }
    return x; // the representative
}
```

Implementation by forests

Find-Set(2): // O(height of the tree) time




```
Find-Set(x){
    while(x.parent ≠ x){
        x = x.parent;
    }
    return x; // the representative
}
```

Using forest representation and union by rank, the height of tree is at most O(log n).

Exercise

Prove that the height of trees is O(log n) in the forest representation that uses union by rank to merge two trees.

Running time of using forest representation and union by rank heuristic

<u>Claim</u>. It takes O(m log n) running time to perform any sequence of m Make-Set(x), Find-Set(x), Union(x, y) operations, in which there are n Make-Set(x) operations.

Proof. Make-Set(x) and Union(x, y) operations need O(1) time each. Find-Set(x) can be done in O(log n) time. We thus get the O(m log n) bound.

Running time of using forest representation and union by rank heuristic

<u>Claim</u>. It takes O(m log n) running time to perform any sequence of m Make-Set(x), Find-Set(x), Union(x, y) operations, in which there are n Make-Set(x) operations.

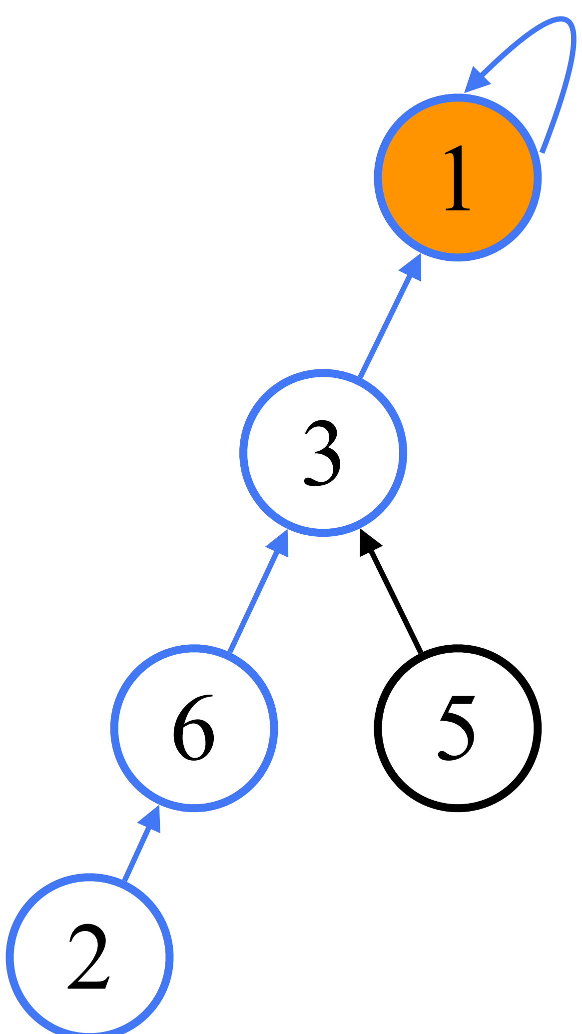
Proof. Make-Set(x) and Union(x, y) operations need O(1) time each. Find-Set(x) can be done in O(log n) time. We thus get the O(m log n) bound.

Note that n must be $\leq$ m. Hence, O(m log n) is slower than O(n log n + m).

Implementation by forests (faster)

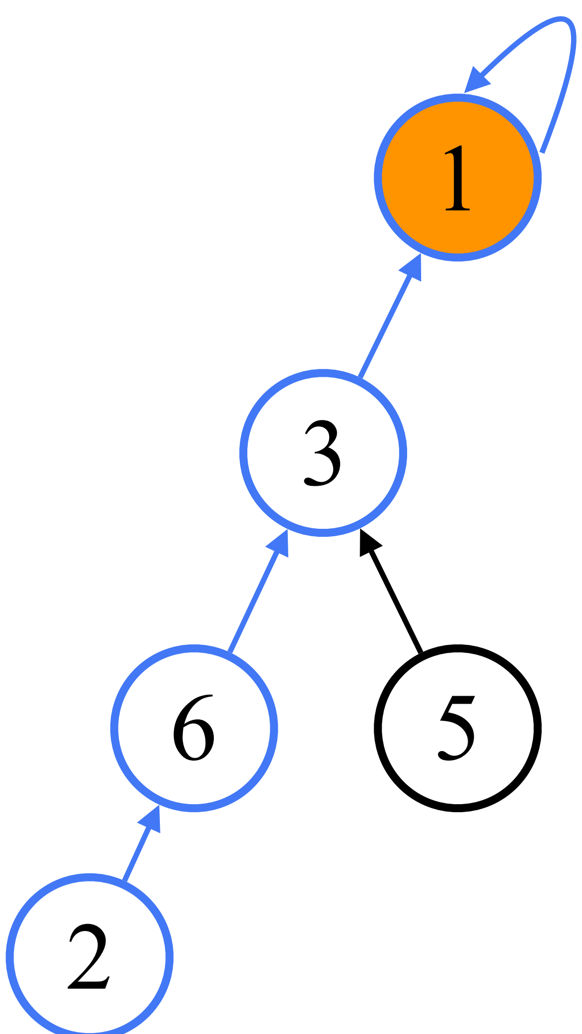
Find-Set(2): // O(height of the tree) time. In the meanwhile, we flatten the tree so that Find-Set(x) runs faster next time.

```
Find-Set(x){
    if(x.parent ≠ x){
        x.parent = Find-Set(x.parent);
    }
    return x.parent; // the representative
}
```



Implementation by forests (faster)

Find-Set(2): // O(height of the tree) time. In the meanwhile, we flatten the tree so that Find-Set(x) runs faster next time.

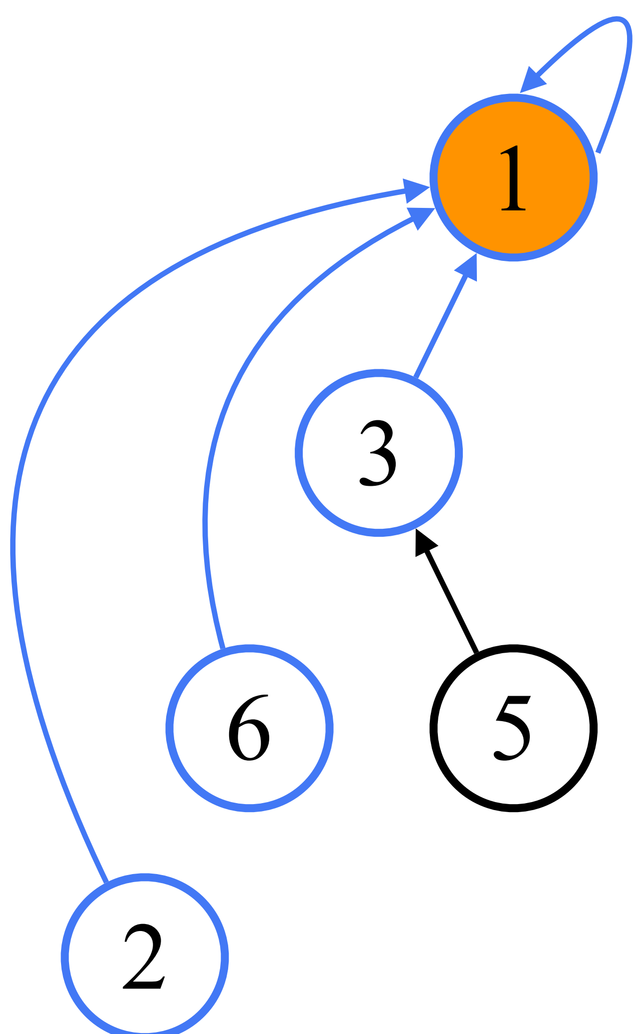



```
Find-Set(x){
    if(x.parent ≠ x){
        x.parent = Find-Set(x.parent);
    }
    return x.parent; // the representative
}
```

This technique is called path compression.

Implementation by forests

Find-Set(2): // O(height of the tree) time. In the meanwhile, we flatten the tree so that Find-Set(x) runs faster next time.




```
Find-Set(x){
    if(x.parent ≠ x){
        x.parent = Find-Set(x.parent);
    }
    return x.parent; // the representative
}
```

This technique is called path compression.

Running time of using forest representation and union by rank and path compression

Claim. It takes $O(m\ \alpha(n))$ running time to perform any sequence of m Make-Set(x), Find-Set(x), Union(x, y) operations, in which there are n Make-Set(x) operations.

Proof. The proof can be found in Chap 21.4.

Running time of using forest representation and union by rank and path compression

Claim. It takes $O(m\ \alpha(n))$ running time to perform any sequence of m Make-Set(x), Find-Set(x), Union(x, y) operations, in which there are n Make-Set(x) operations.

Proof. The proof can be found in Chap 21.4.

Note that $\alpha(n)$ is a very slowly growing function that $\alpha(n) \leq 4$ for any $n \leq 10^{80}$.
Hence, $O(m\ \alpha(n))$ is virtually linear in practice.

A very quickly growing function $A_k(j)$

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j+1) & \text{if } k > 1 \end{cases}$$

where $A_{k-1}^{(0)}(j) = j$ and $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$

A very quickly growing function $A_k(j)$

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j+1) & \text{if } k > 1 \end{cases}$$

where $A_{k-1}^{(0)}(j) = j$ and $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$

By simple calculations, we have:

(1) $A_1(j) = 2j+1$ and $A_1(1) = 3$
(2) $A_2(j) = 2^{j+1}(j+1) - 1$ and $A_2(1) = 7$
(3) $A_3(1) = 2047$
(4) $A_4(1) \gg 10^{80}$
(5) $A_1(1) < A_2(1) < A_3(1) < \dots$

A very slowly growing function $\alpha(n)$

Let $\alpha(n) = \min\{k : A_k(1) \geq n\}$, the inverse of A_k.

A very slowly growing function $\alpha(n)$

Let $\alpha(n) = \min\{k : A_k(1) \geq n\}$, the inverse of A_k.

For every $n \leq 10^{80}$, $\alpha(n) \leq 4$.

In practice

Running time of using forest representation and ~~union by rank and~~ path compression

<u>Claim</u>. It takes $O(n + f\,(1+\log_{2+f/n} n))$ running time to perform any sequence of n Make-Set(x), f Find-Set(x), and <n Union(x, y) operations.

Running time of using forest representation and ~~union by rank and~~ path compression

Claim. It takes $O(n + f\,(1+\log_{2+f/n} n))$ running time to perform any sequence of n Make-Set(x), f Find-Set(x), and <n Union(x, y) operations.

Path compression is already efficient enough for the practical use. You may ignore union by rank.

Pseudocode of using forest representation and ~~union by rank and~~ path compression

```
int rep[1..n];

for(int i=1; i ≤ n; ++i){
  rep[i] = i; // set i to be the representive of the i-th tree
}
Find-Set(x, rep){
   if(rep[x] equals x) return x;
   return rep[x] = Find-Set(rep[x], rep); // path compression
}
Union(x, y, rep){
   if(Find-Set(x, rep) ≠ Find-Set(y, rep))
       rep[rep[x]] = rep[y]; // unite arbitrarily
}
```