CSE 373
Data Structures and Algorithms

Lecture 24: Disjoint Sets

Kruskal's Algorithm Implementation

Kruskals():

sort edges in increasing order of length (e_1, e_2, e_3, ..., e_m).

T := {}.

for i = 1 to m

if e_i *does not add a cycle:*

add e_i *to T.*

return T.

- How can we determine that adding e_i to *T* won't add a cycle?

Disjoint-set Data Structure

- Keeps track of a set of elements partitioned into a number of disjoint subsets
  - Two sets are *disjoint* if they have no elements in common
- Initially, each element e is a set in itself:
  - e.g., { $\{e_1\}$, $\{e_2\}$, $\{e_3\}$, $\{e_4\}$, $\{e_5\}$, $\{e_6\}$, $\{e_7\}$}

Operations: Union

- Union(*x*, *y*) – Combine or merge two sets *x* and *y* into a single set
  - Before:

    $\{\{e_3, e_5, e_7\}, \{e_4, e_2, e_8\}, \{e_9\}, \{e_1, e_6\}\}$

  - After Union(e_5, e_1):

    $\{\{e_3, e_5, e_7, e_1, e_6\}, \{e_4, e_2, e_8\}, \{e_9\}\}$

Operations: Find

- Determine which set a particular element is in
  - Useful for determining if two elements are in the same set

- Each set has a unique name
  - Name is arbitrary; what matters is that find(*a*) == find(*b*) is true only if *a* and *b* in the same set
  - One of the members of the set is the "representative" (i.e. name) of the set

  e.g., {{e_3, e_5, e_7, e_1, e_6} , {e_4, e_2, e_8}, {e_9}}

Operations: Find

- Find(x) – return the name of the set containing x.
  - $\{\{e_3, e_5, e_7, e_1, e_6\}, \{e_4, e_2, e_8\}, \{e_9\}\}$
  - Find(e_1) = e_5
  - Find(e_4) = e_8

Kruskal's Algorithm (Revisited)

Kruskals():

sort edges in increasing order of length $(e_1, e_2, e_3, ..., e_m)$.

initialize disjoint sets.

T := {}.

for i = 1 to m

let $e_i = (u, v)$.

if find(u) != find(v)

union(find(u), find(v)).

add e_i *to T.*

return T.

- What does the disjoint set initialize to?
- Assuming *n* nodes and *m* edges:
  - How many times do we do a union?
    n-1
  - How many times do we do a find?
    2 * m
  - What is the total running time?
    O(m log m + U * n + F * m)

Disjoint Sets with Linked Lists

- Approach 1: Create a linked list for each set.
  - Last/first element is representative
  - Cost of union? find?
    O(1) O(n)

- Approach 2: Create linked list for each set. Every element has a reference to its representative.
  - Last/first element is representative
  - Cost of union? find?
    O(n) O(1)

Disjoint Sets with Trees

- Observation: *trees* let us find many elements given one root (i.e. representative)

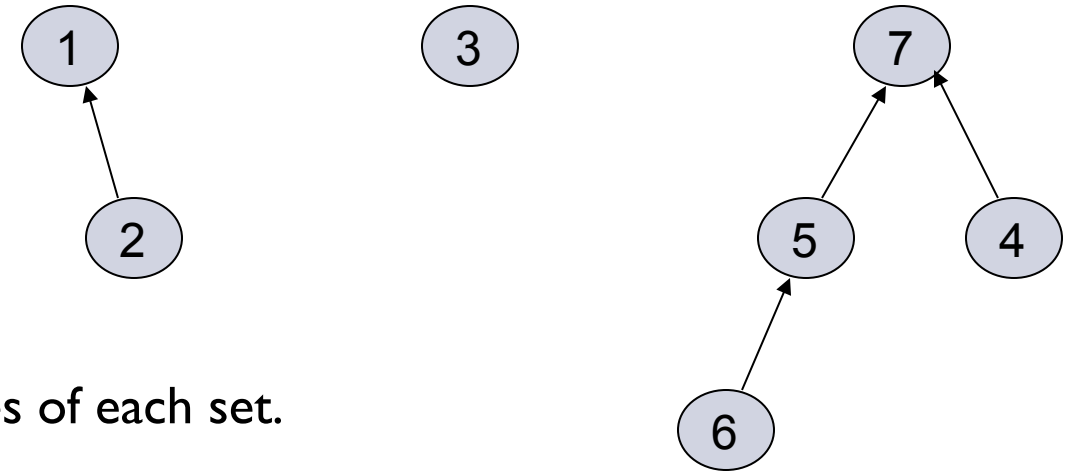
- Idea: If we *reverse* the pointers (make them point up from child to parent), we can find a single root from many elements.

- Idea: Use one tree for each subset. The name of the class is the tree root.

Up-Tree for Disjoint Sets

Initial state

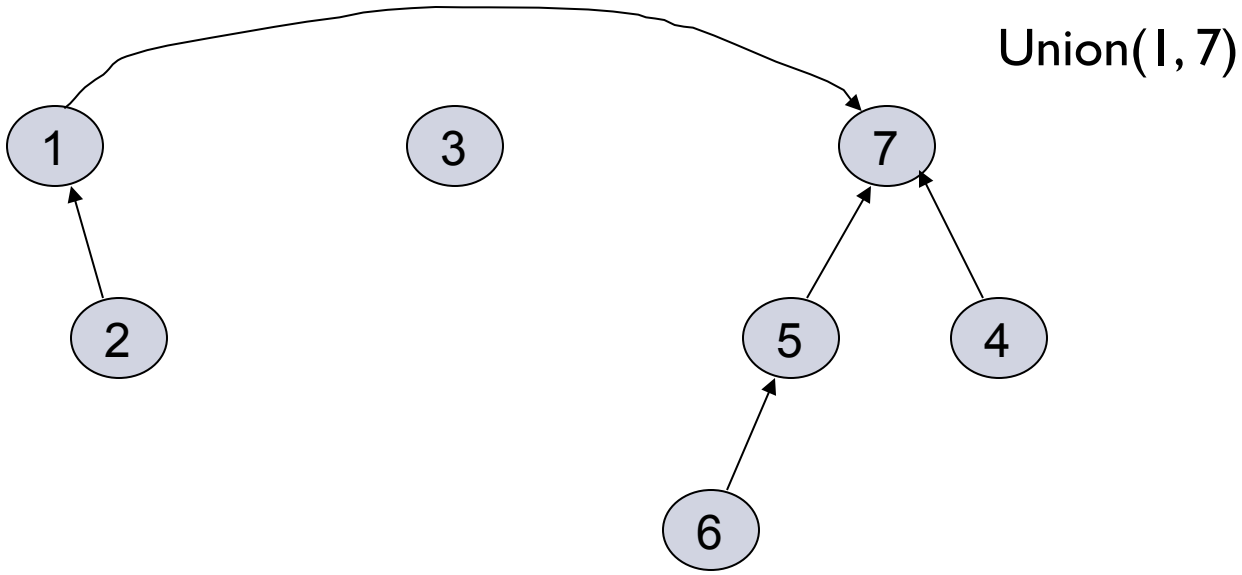
Intermediate state



Roots are the names of each set.

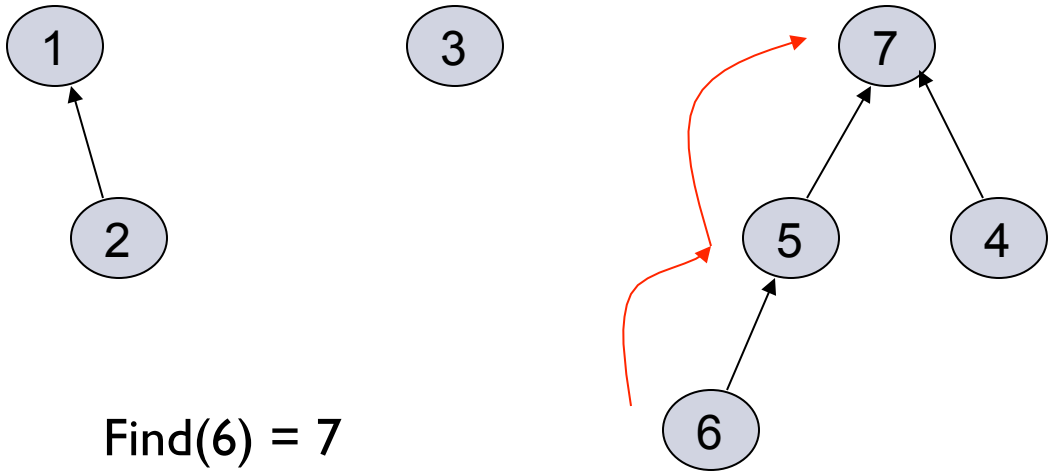
Union Operation

- Union(*x*, *y*) – assuming *x* and *y* roots, point *x* to *y*.



Find Operation

- Find(x): follow x to root and return root

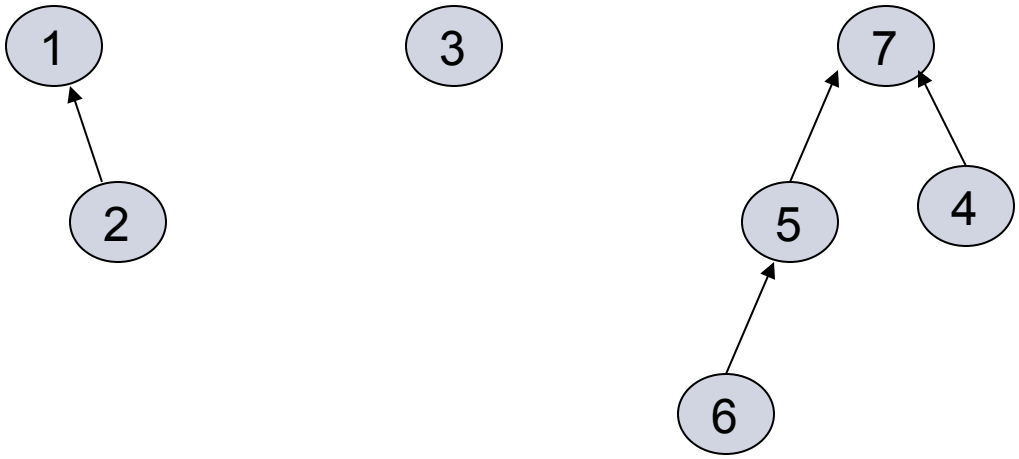


Simple Implementation

- Array of indices

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

Up[x] = 0 means
x is a root.



Union

```
void Union(int[] up, int x, int y) {
    // precondition: x and y are roots
    up[x] = y
}
```

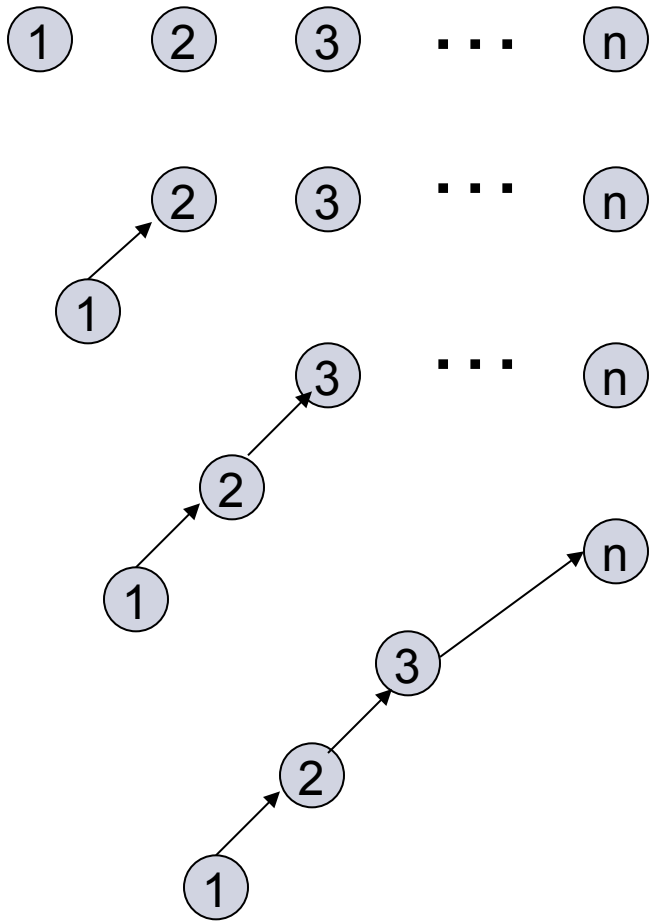
Constant Time!

Find

```
int Find(int[] up, int x) {
    // precondition: x is in the range 1 to size
    if up[x] == 0
        return x
    else
        return Find(up, up[x])
}
```

- Exercise: Write an iterative version of Find.

A Bad Case



Union(1,2)

Union(2,3)

:

Union(n-1, n)

Find(1) n steps!!

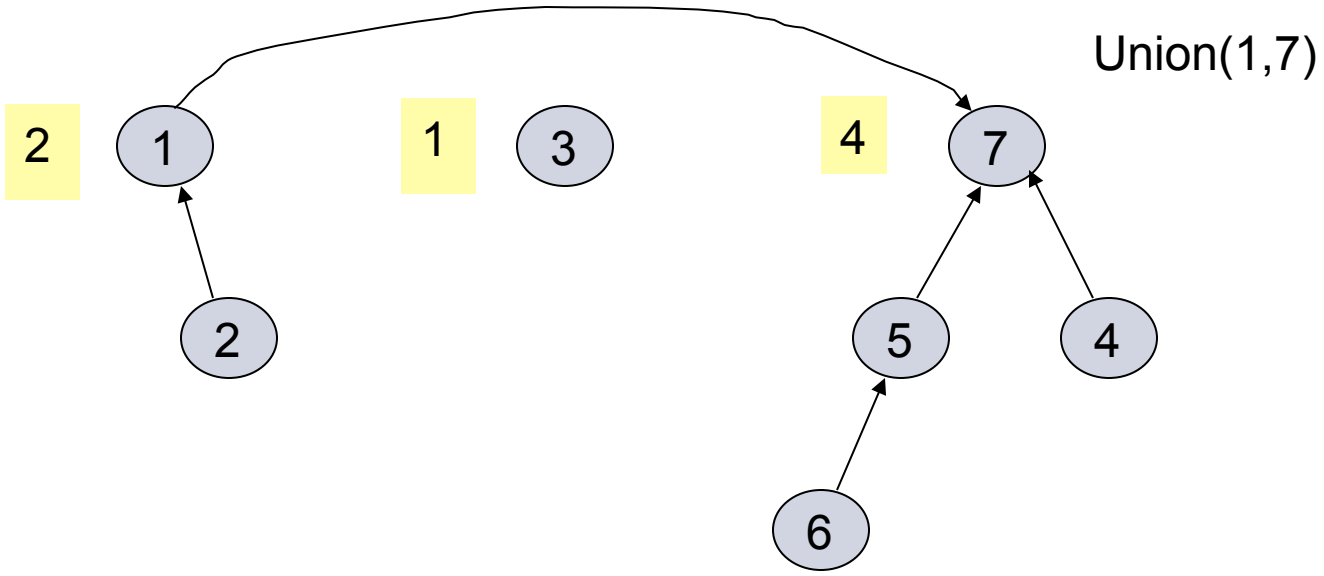
Improving Find

- Improve union so that find only takes $\Theta(\log n)$
  - Union-by-size

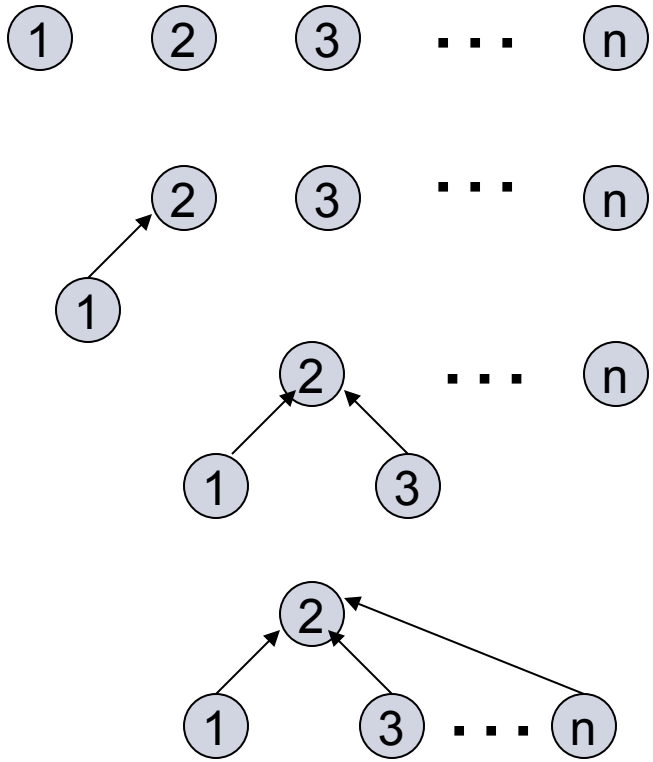
- Improve find so that it becomes even better!
  - Path compression

Union by Rank

- Union by Rank (also called Union by Size)
  - Always point the smaller tree to the root of the larger tree



Example Again



Union(1,2)

Union(2,3)

:

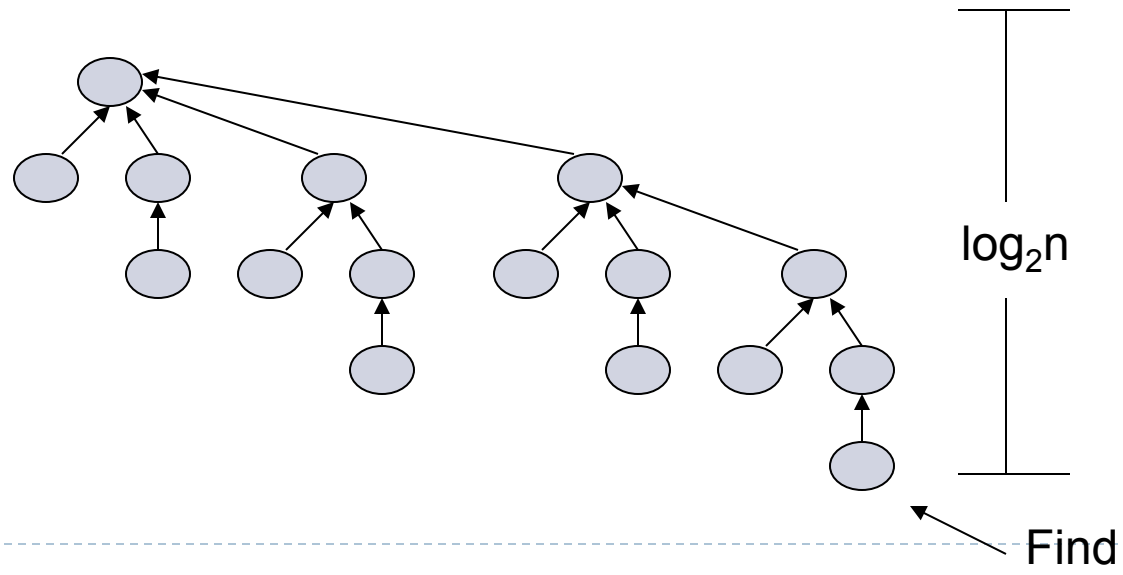
:

Union(n-1,n)

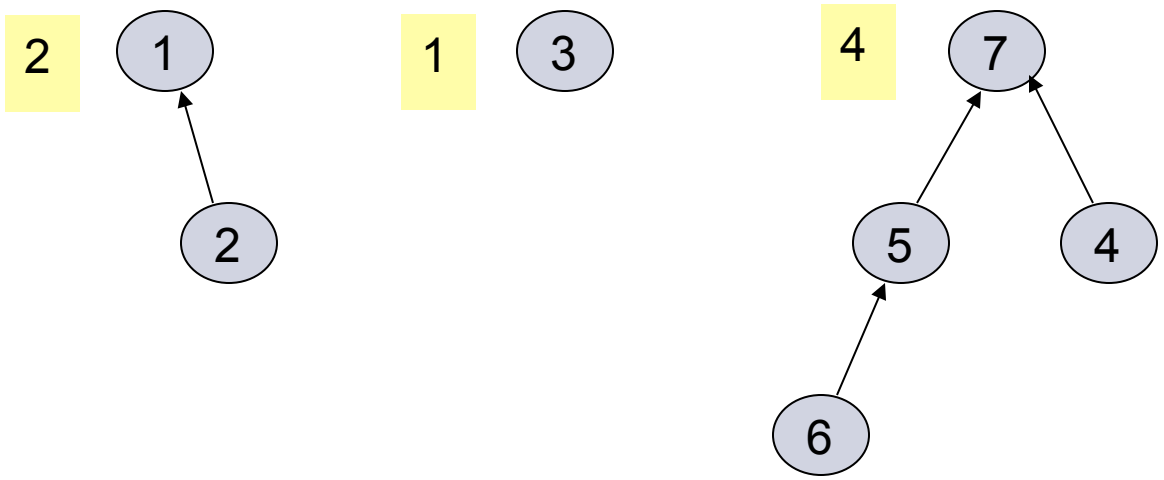
Find(1) constant time

Runtime for Find via Union by Rank

- Depth of tree affects running time of Find
- Union by rank only increases tree depth if depth were equal
- Results in O(log *n*) for Find



Elegant Array Implementation



	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0
weight	2		1				4

Union by Rank

```
void Union(int i, int j){
  // i and j are roots
  wi = weight[i];
  wj = weight[j];
  if wi < wj then
    up[i] = j;
    weight[j] = wi + wj;
  else
    up[j] = i;
    weight[i] = wi + wj;
}
```

Kruskal's Algorithm (Revisited)

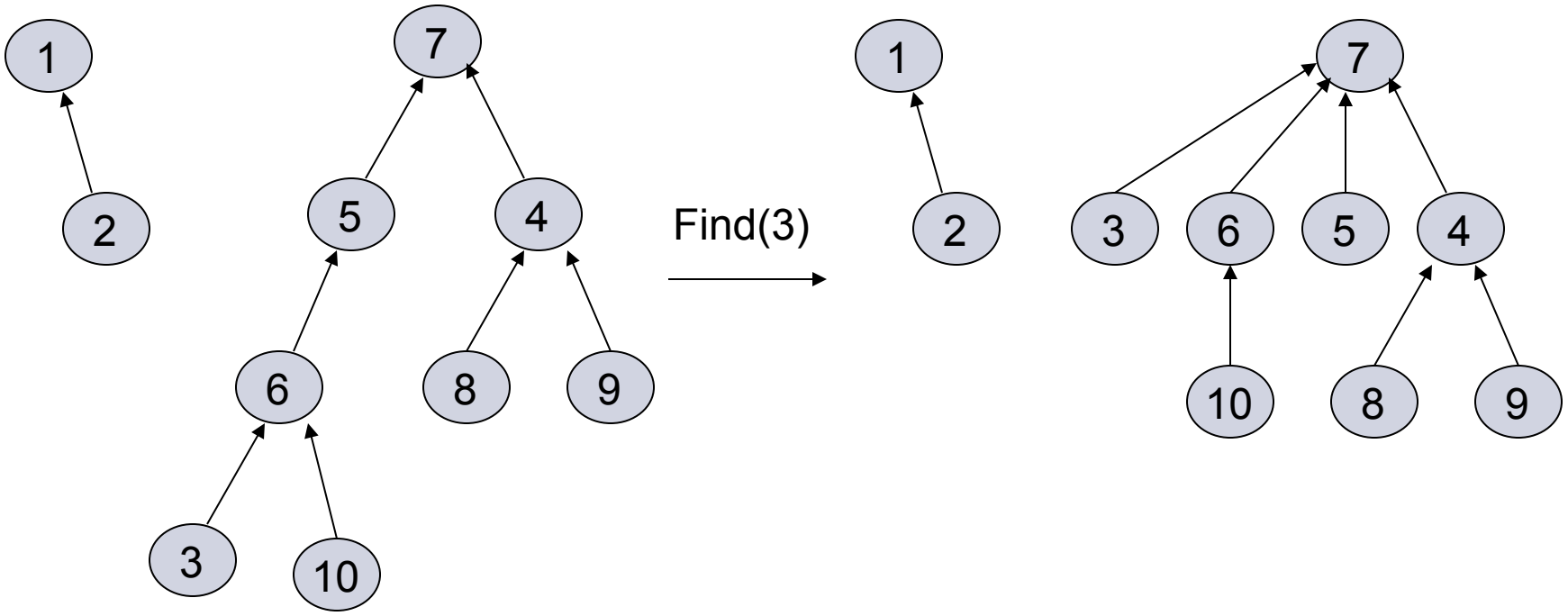
```
Kruskals():
    sort edges in increasing order of length (e1, e2, e3, ..., em).

    initialize disjoint sets.

    T := {}.

    for i = 1 to m
        let ei = (u, v).
        if find(u) != find(v)
            union(find(u), find(v)).
            add ei to T.

    return T.
```

- |E| = m edges, |V| = n nodes
- Sort edges: O(m log m)
- Initialization: O(n)
- Finds: O(2 * m * log n)
  = O(m log n)
- Unions: O(n)

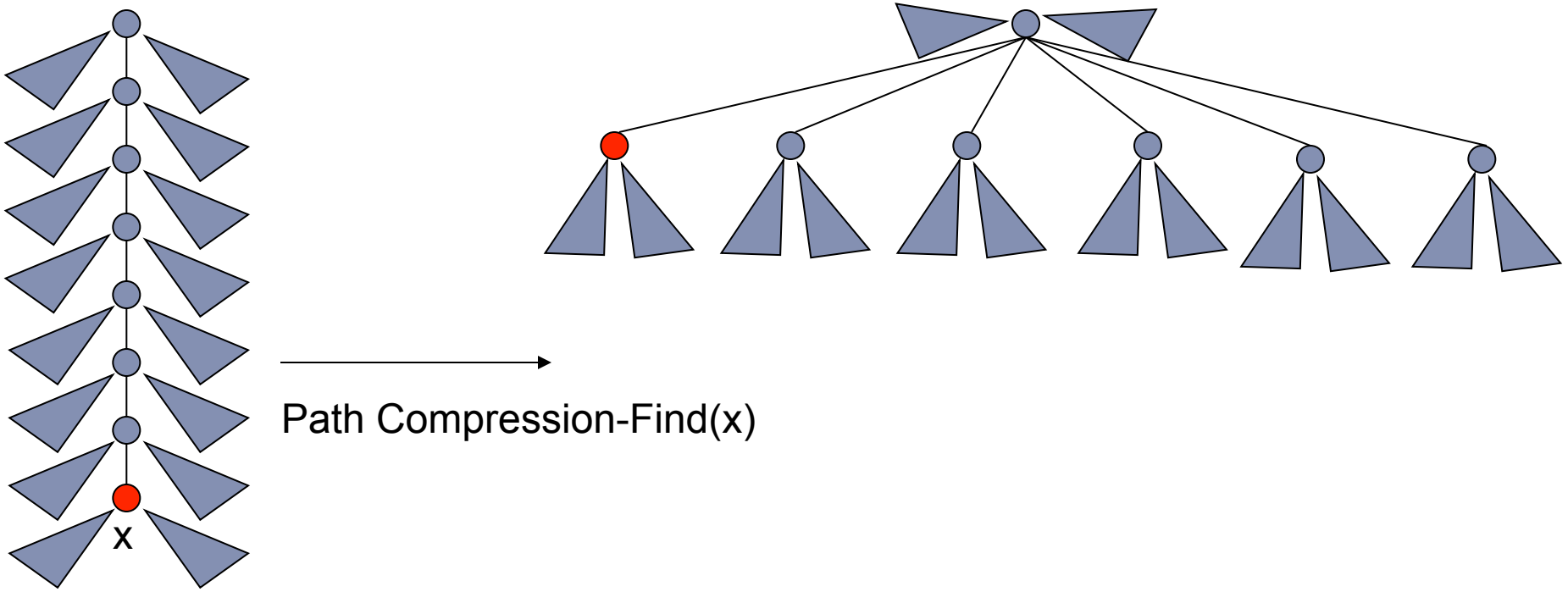
- Total running time:
  O (m log m + n + m log n + n)
  = O(m log n)
  - Note: log n and log m are within a constant factor of one another (Why?)

Path Compression

- On a Find operation point all the nodes on the search path directly to the root.

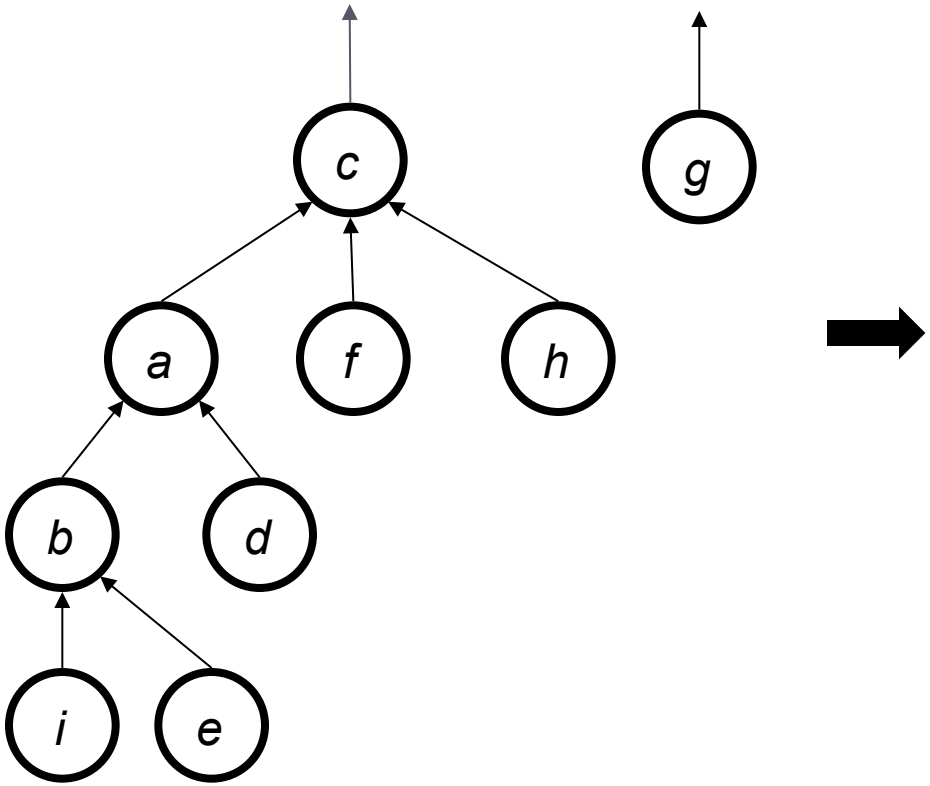


Self-Adjustment Works



Path Compression Exercise:

- Draw the resulting up tree after Find(e) with path compression.



Path Compression Find

```
void PC-Find(int i) {
  r = i;
  while up[r] ≠ 0 do // find root
    r = up[r];
  if i ≠ r then  // compress path
    k = up[i];
    while k ≠ r do
      up[i] = r;
      i = k;
      k = up[k]
  return r;
}
```

Other Applications of Disjoint Sets

- Good for applications in need of clustering
  - Cities connected by roads
  - Cities belonging to the same country
  - Connected components of a graph
- Forming equivalence classes (see textbook)
- Maze creation (see textbook)