# CSE 373: Data Structures and Algorithms

## Lecture 23: 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.
```

- But 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 disjoint subsets
  - two sets are said to be 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
  - $\{\{e_3, e_5, e_{7,} e_1, e_6\}, \{e_4, e_2, e_8\}, \{e_9\}\}$

# Operations: Find

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

# Kruskal's Algorithm Implementation (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?
- How many times do we do a union?
- How many time do we do a find?
- What is the total running time if we have *n* nodes and *m* edges?

# Disjoint Sets with Linked Lists

- Approach 1: Create a linked list for each set.
  - last/first element is representative
  - cost of union?  find?

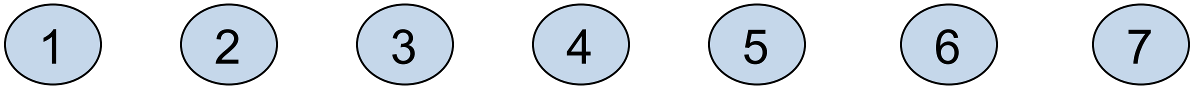
- Approach 2: Create linked list for each set. Every element has a reference to its representative.
  - last/first element is representative
  - cost of union?  find?

# Disjoint Sets with Trees

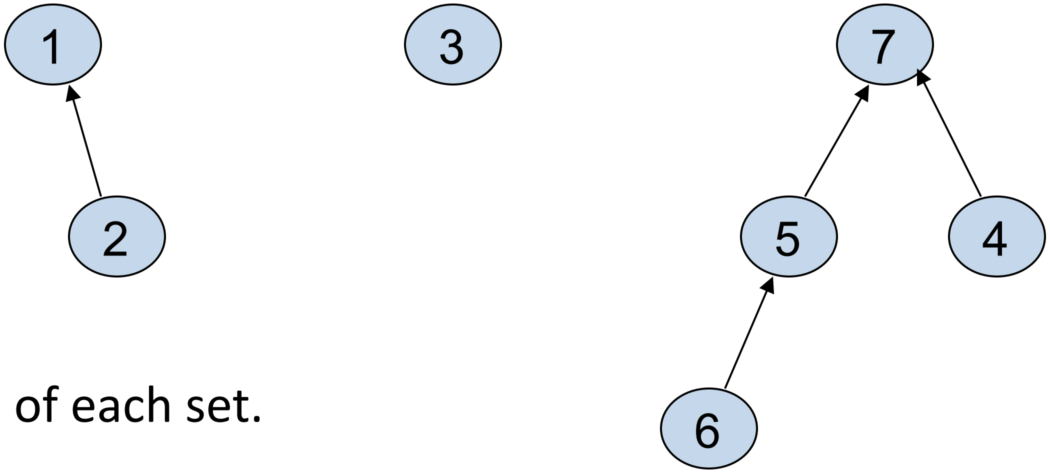
- Observation: *trees* let us find many elements given one root (i.e. representative)…
- Idea: if we *reverse* the pointers (make them point up from child to parent), we can find a single root from many elements…
- Idea: Use one tree for each subset. The name of the class is the tree root.

# Up-Tree for Disjoint Sets

Initial state



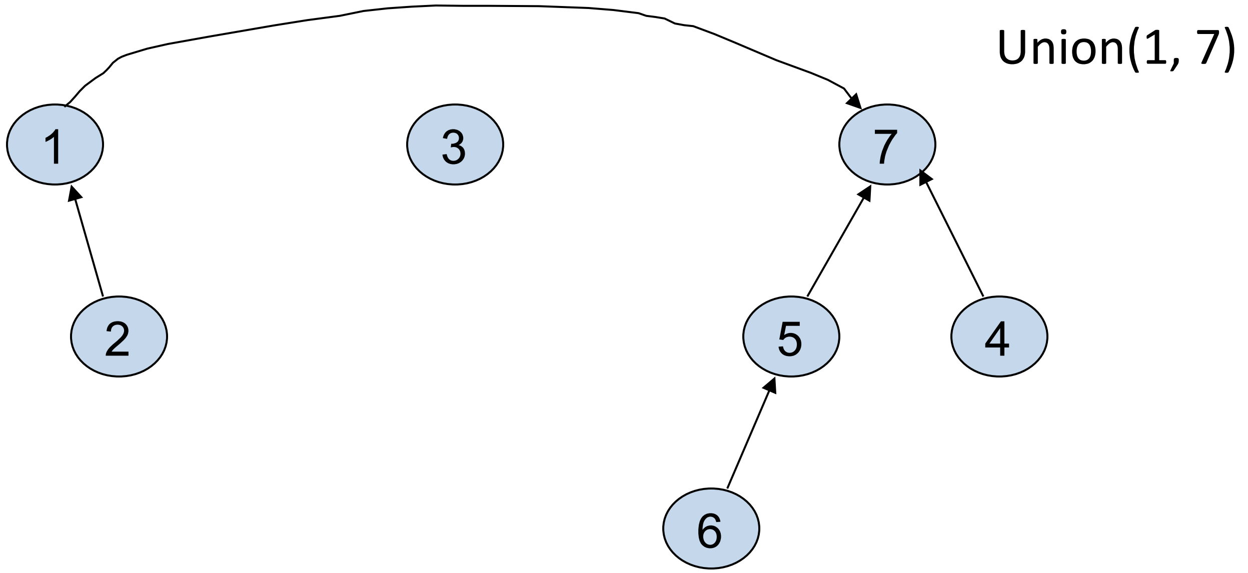
Intermediate state



Roots are the names of each set.

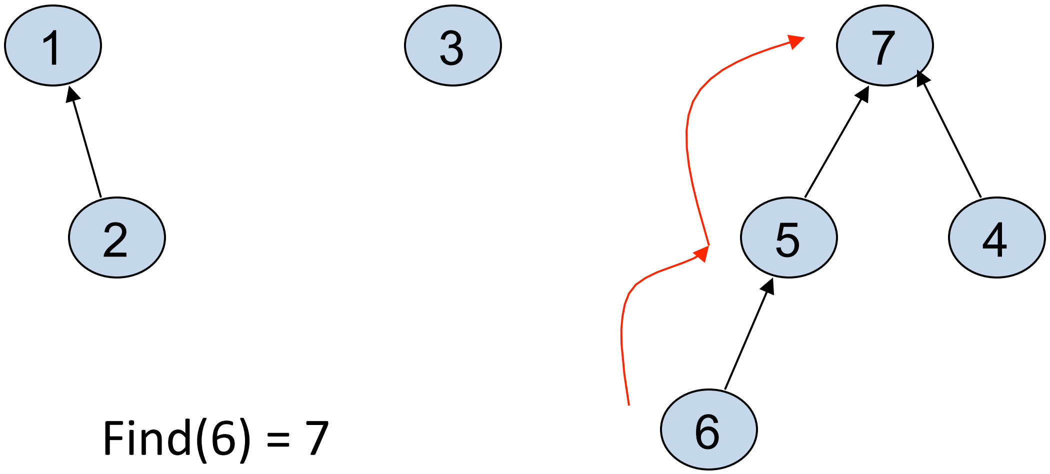
# Union Operation

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



# Find Operation

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



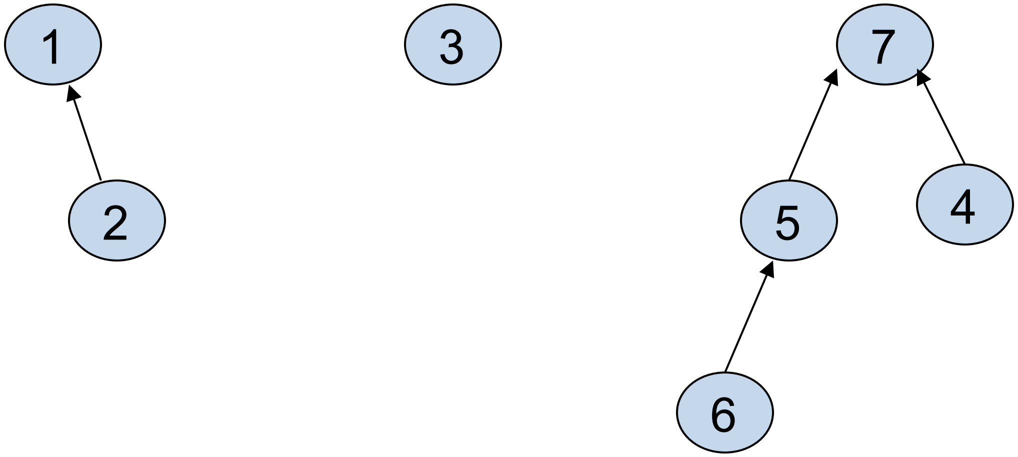
Find(6) = 7

# 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

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

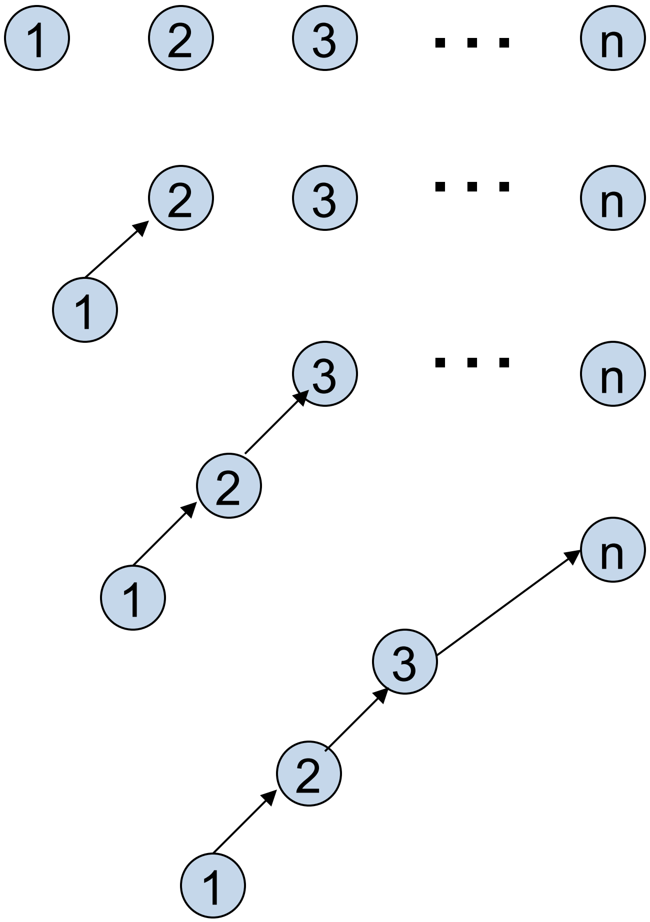
Constant Time!

# Find

```
Find(up[] : integer array, x : integer) : integer {
    //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!!

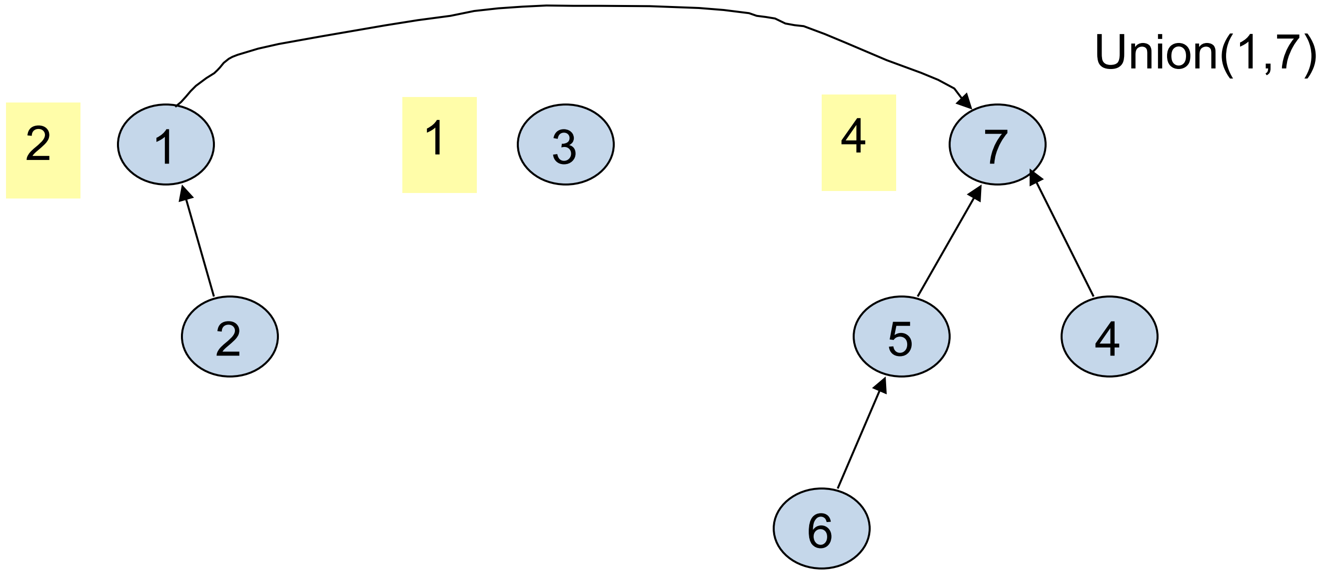
# Improving Find

Can we do better? *Yes!*

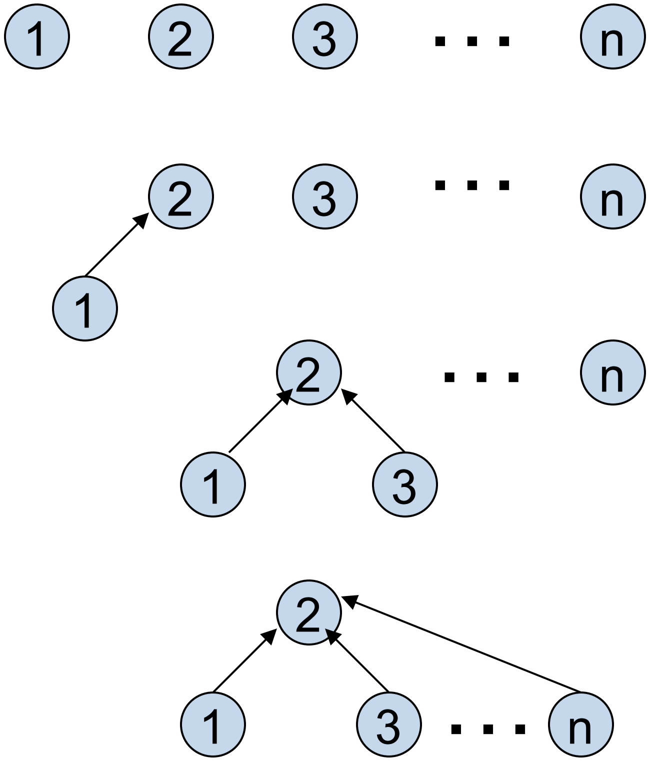
1. Improve union so that *find* only takes Θ(log $n$)
    - Union-by-size
    - Reduces complexity to Θ($m$ log $n$ + $n$)
2. Improve find so that it becomes even better!
    - Path compression
    - Reduces complexity to <u>almost</u> Θ($m$ + $n$)

# 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)

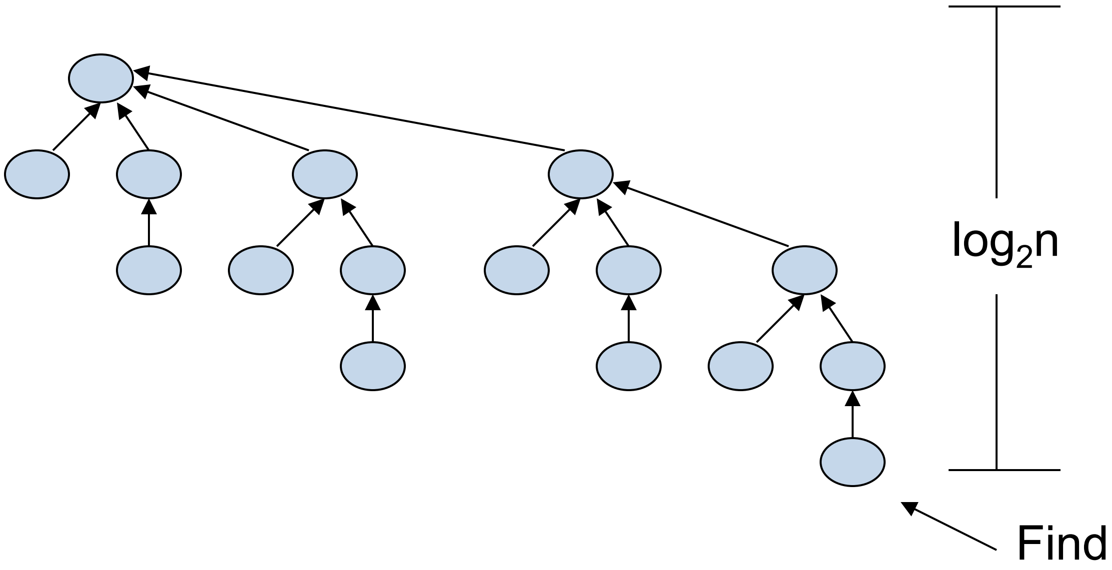
⋮

Union(n-1,n)

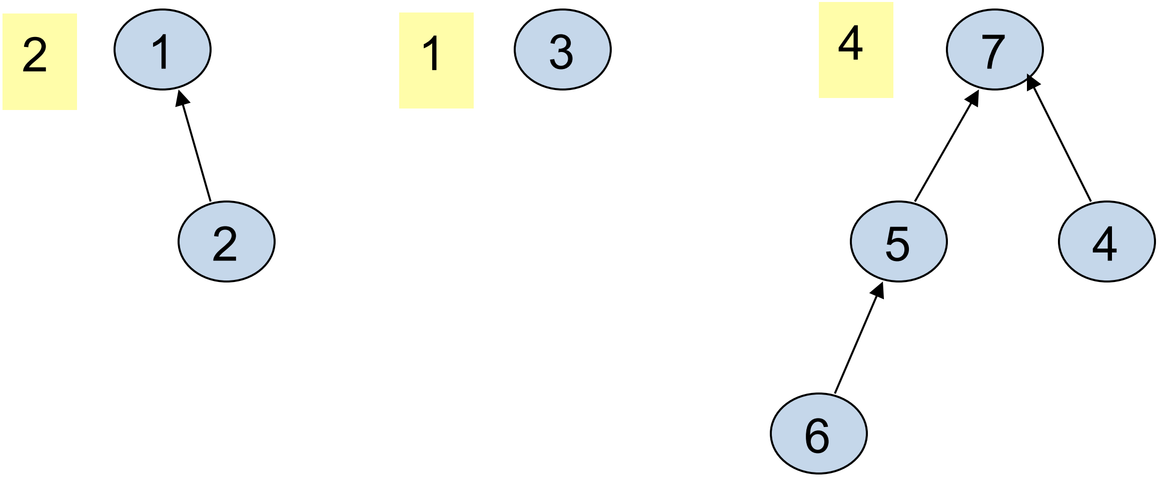
Find(1) constant time

# Improved 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

```
Union(i,j : index){
  //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 Implementation (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.
```

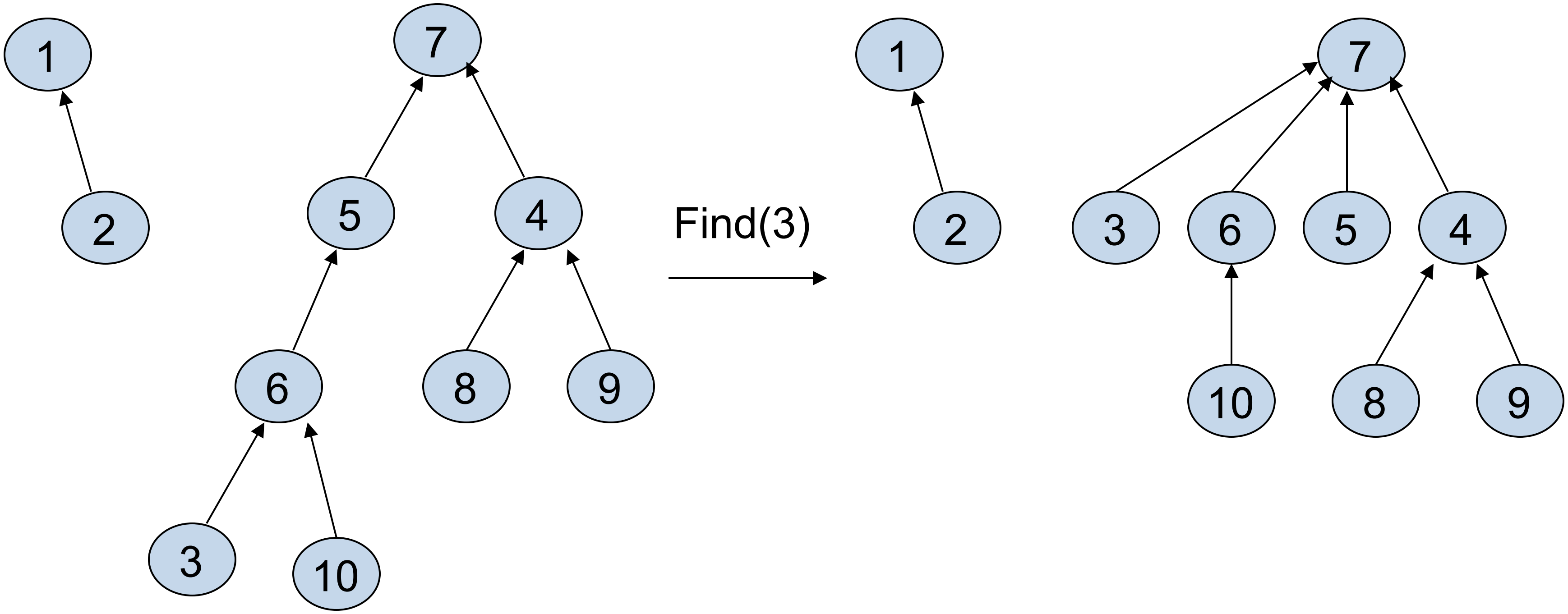
# Kruskal's Algorithm Running Time (Revisited)

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

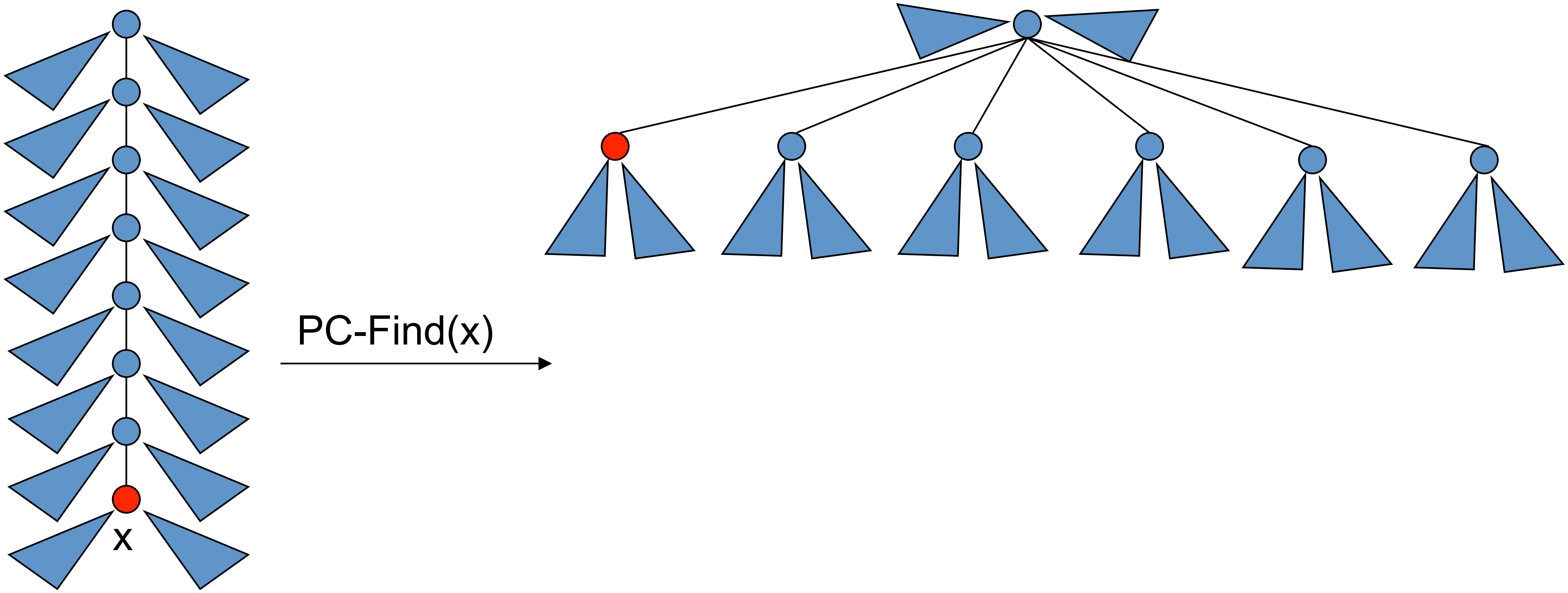
- Total running time: O (*m* log *n* + *n* + *m* log *n* + *m*) = O (*m* log *n*)
  - note: log n and log m are within a constant factor of one another

# Path Compression

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

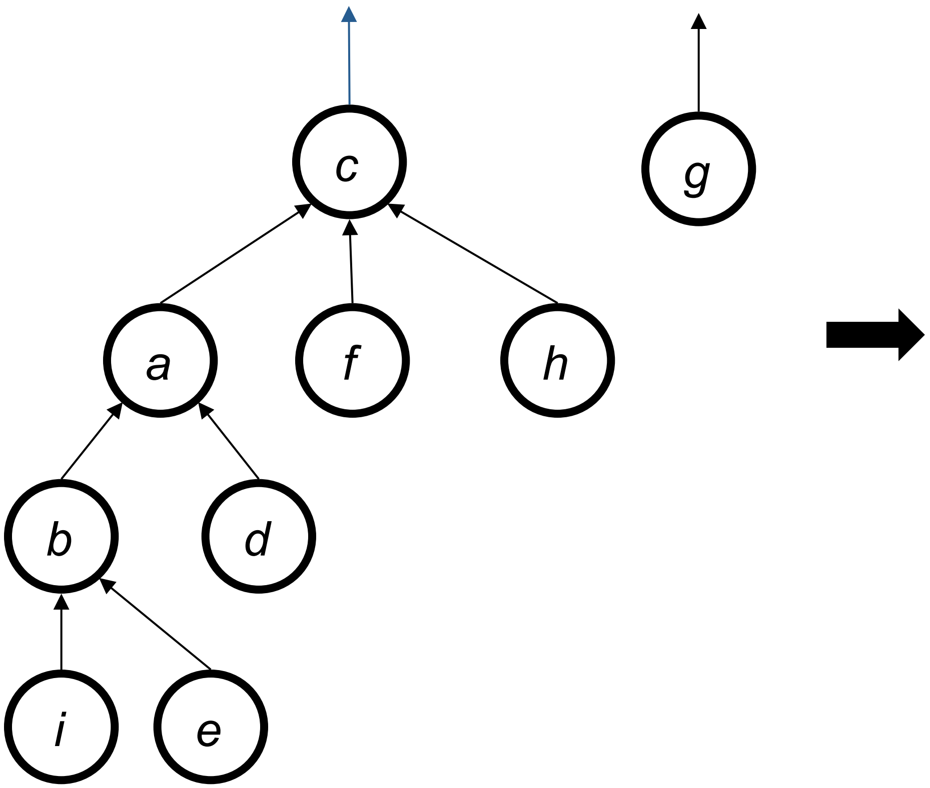


# Self-Adjustment Works



# Path Compression Exercise:

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



# Path Compression Find

```
PC-Find(i : index) {
  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)
}
```

# Disjoint Union / Find with Union By Rank and Path Comp.

- Worst case time complexity for a Union using Union by Rank is $\Theta(1)$ and for Find using Path Compression is $\Theta(\log n)$.

- Time complexity for $m \geq n$ operations on n elements is $\Theta(m \log^* n)$
  - log * is the number of times you need to apply the log function before you get to a number <= 1
  - log * n < 5 for all reasonable n. Essentially constant time per operation!

# Amortized Complexity

- For disjoint union / find with union by rank and path compression
  - average time per operation is essentially a constant
  - worst case time for a Find is $\Theta(\log n)$

- An individual operation can be costly, but over time the average cost per operation is not

- This means the bottleneck of Kruskal's actually becomes the sorting of the edges

# 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)