CSE 332: Data Structures & Parallelism
Lecture 26: Disjoint Sets

Arthur Liu

Spring 2022

Announcements

- Exercise 15 Extended to be Due Tuesday May 31st
- Last few assignments:
  - P3 due Tuesday
  - EX16 due next Friday
  - Final!!
- Memorial Day Monday, Scheduled OH are cancelled

Remember Kruskal's Algorithm for MST?

Sort all the edges (*read: min-heap*)

while (don't have MST yet) {

    get next smallest edge, e   *(edge from u, v)*

    check if group u == group v

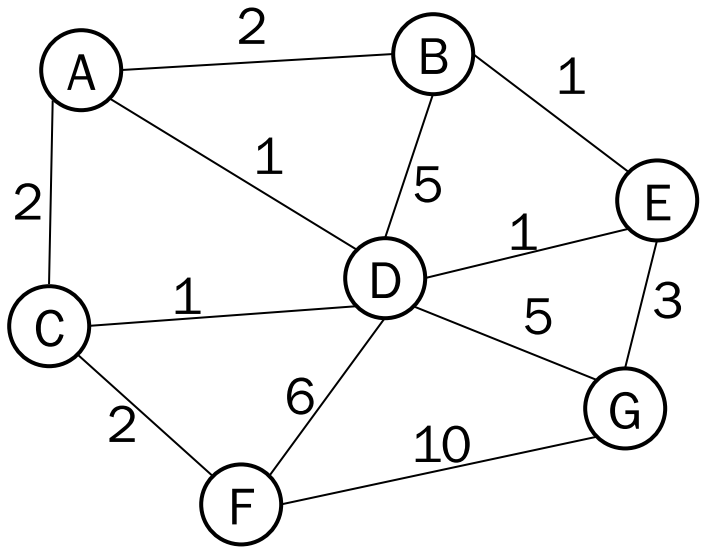
    if (same group) {

        add that edge to our MST

        union groups u, v

    }

}



Remember Kruskal's Algorithm for MST?

Sort all the edges (*read: min-heap*)

while (don't have MST yet) {

    get next smallest edge, e   *(edge from u, v)*

    check if group u == group v

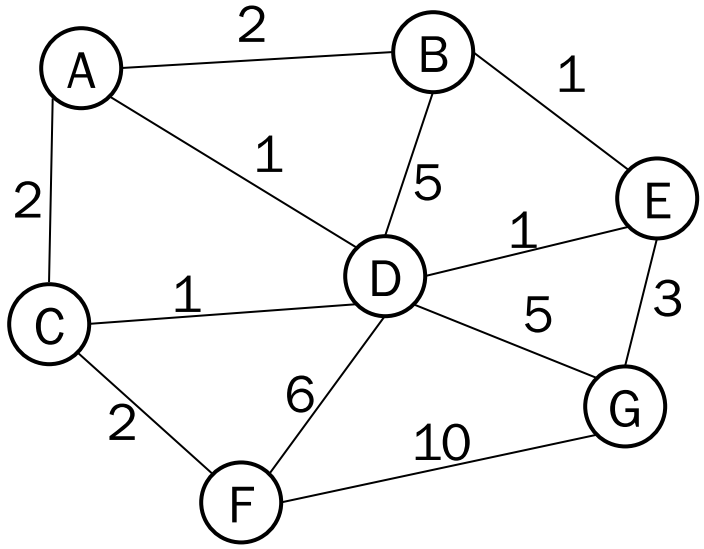
    if (same group) {

        add that edge to our MST

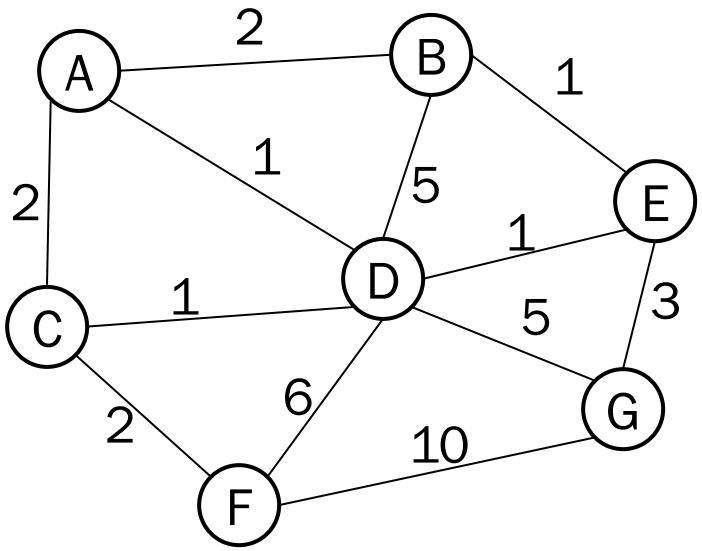
        union groups u, v

    }

}



Example: Find MST using Kruskal's



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

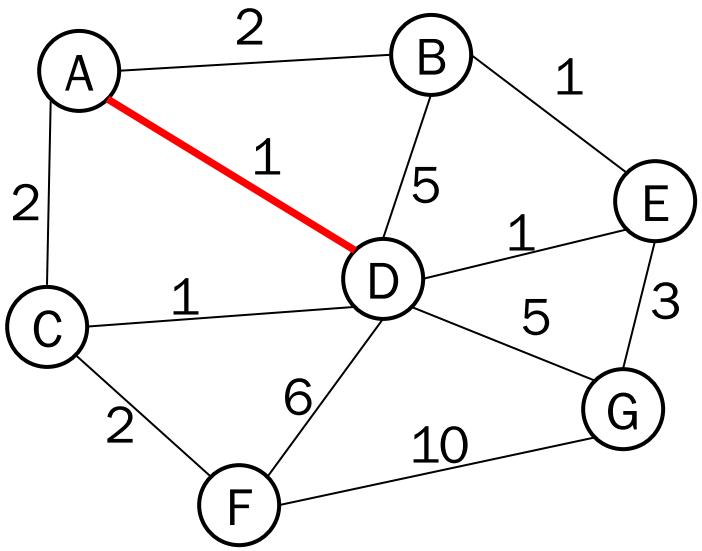
10: (F,G)

Output:

Visualization of State of Disjoint Set :

Group 1	Group 2	Group 3	Group 4	Group 5	Group 6	Group 7
A	B	C	D	E	F	G

Example: Find MST using Kruskal's



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

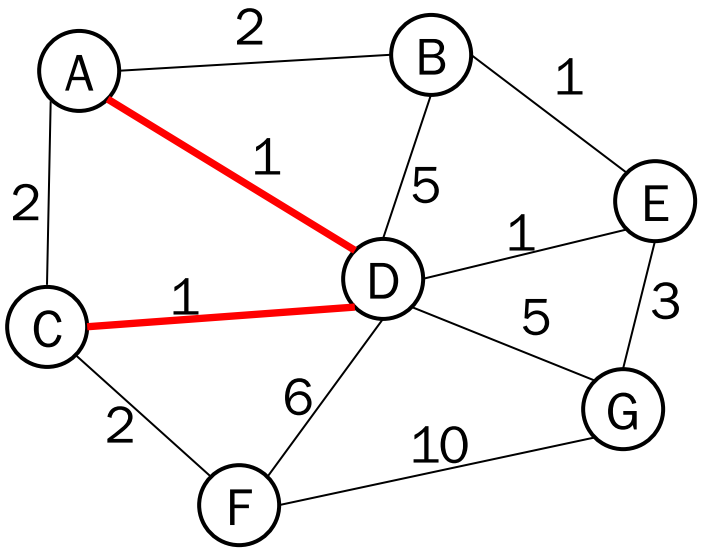
10: (F,G)

Output: (A,D)

Visualization of State of Disjoint Set :

Group 1	Group 2	Group 3	Group 5	Group 6	Group 7
A D	B	C	E	F	G

Example: Find MST using Kruskal's



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

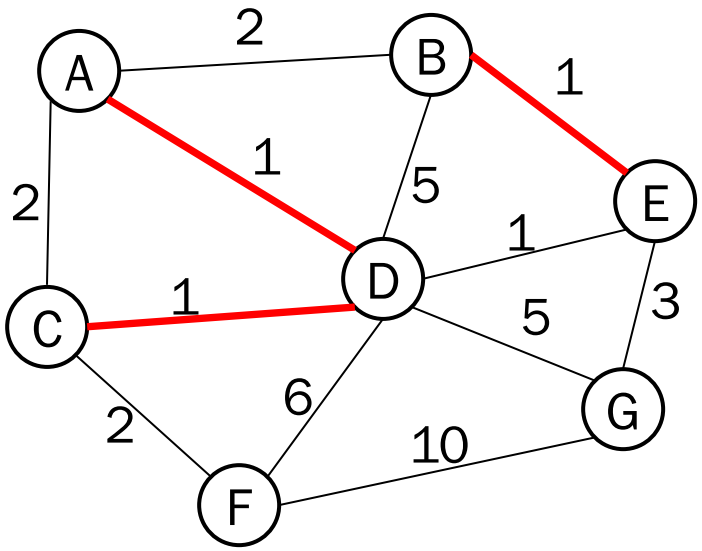
10: (F,G)

Output: (A,D), (C,D)

Visualization of State of Disjoint Set :

Group 1	Group 2	Group 5	Group 6	Group 7
A D C	B	E	F	G

Example: Find MST using Kruskal's



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

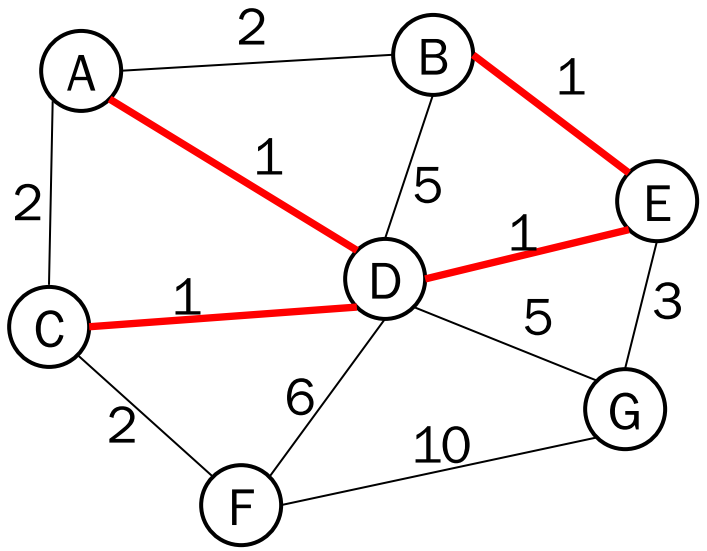
10: (F,G)

Output: (A,D), (C,D), (B,E)

Visualization of State of Disjoint Set :

Group 1	Group 2	Group 6	Group 7
A D C	B E	F	G

Example: Find MST using Kruskal's



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

10: (F,G)

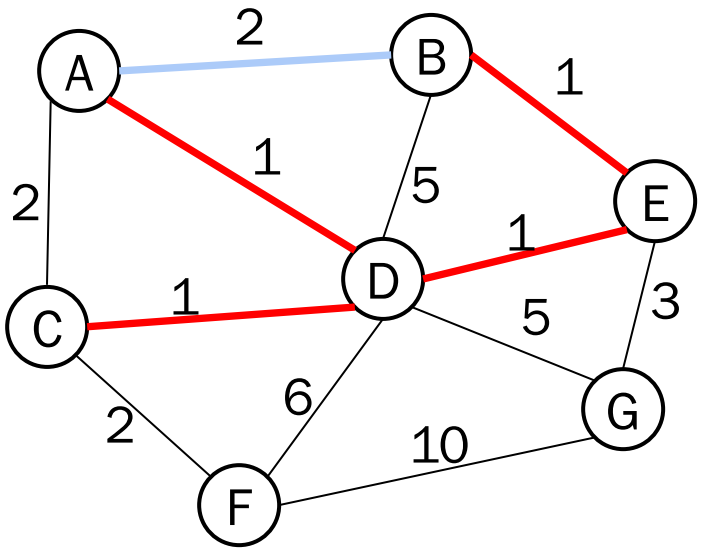
Output: (A,D), (C,D), (B,E), (D,E)

Visualization of State of Disjoint Set :

Group 1
A D C B E

Group 6	Group 7
F	G

Example: Find MST using Kruskal's



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

10: (F,G)

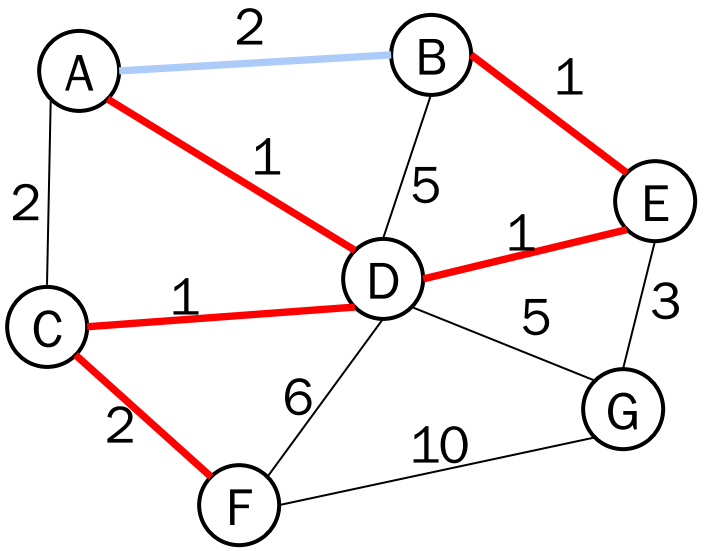
Output: (A,D), (C,D), (B,E), (D,E)

Visualization of State of Disjoint Set :

Group 1
A D C B E

Group 6	Group 7
F	G

Example: Find MST using Kruskal's



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

10: (F,G)

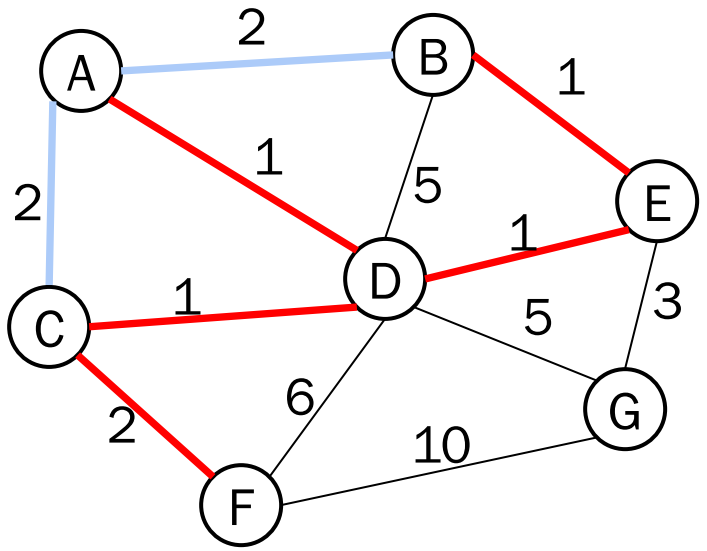
Output: (A,D), (C,D), (B,E), (D,E), (C,F)

Visualization of State of Disjoint Set :

Group 1
A D C B E F

Group 7
G

Example: Find MST using Kruskal's



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

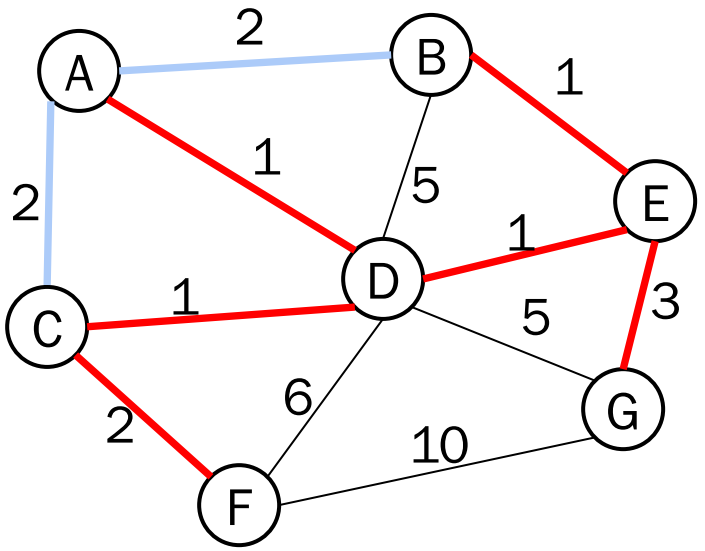
6: (D,F)

10: (F,G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F)

Visualization of State of Disjoint Set :

Example: Find MST using Kruskal's



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

10: (F,G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F), (E,G)

Visualization of State of Disjoint Set :

Group 1
A D C B E F G

Equivalence Relations

Recall 311: a relation, R, relates two elements in our set. (set can be a set of fruit {pears, peaches, apples…} or nodes in graph)

An equivalence relation satisfies three properties:

Let "=" be our example relation R, then…

Reflexive: x = x

Symmetric: if x = y then y = x

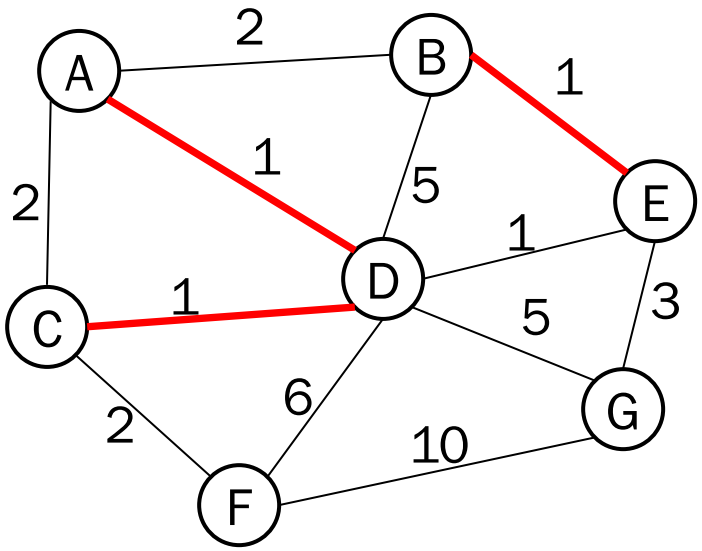
Transitive: if x = y, y = z, then x = z

Equivalence Class

Equivalence class: just the name we give to the subset of elements that are equivalent to each other

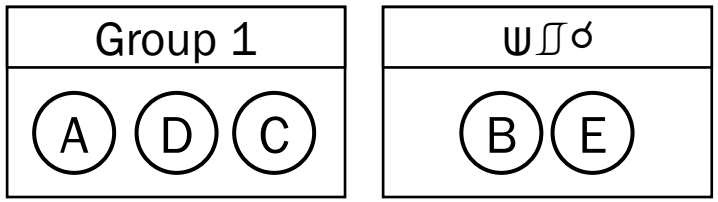
- Disjoint Set ADT helps us keeps track of equivalence classes
- Everyone belongs to a unique class
- Everyone in an equivalence class is related to each other

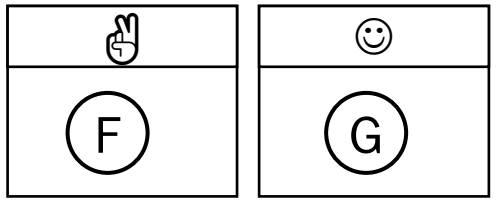
Example: Find MST using Kruskal's



Name of equivalence class/group is not significant, as long as unique to other group names

Visualization of State of Disjoint Set :





Formalization: Disjoint Set ADT

`Find(x)`:

- Returns the name of the set containing x.
  - Name can be anything as it long as it uniquely identifies the set and everyone agrees to that name
- Ex: Given sets
  - **Find(10)** returns 1
  - **Find(4)** returns 2

Group 1	Group 2	Group 3
3, 5, 10, 13	2, 4, 8	9

Find Runtime:
amortized O(1)
Worst case O(log n)

`Union(group1, group2)`:

- Takes the union of two sets named `group1` and `group2`
- Ex: Given the above sets
  - **Union(2, 3)** =>

Group 1	Group 2
3, 5, 10, 13	2, 4, 8, 9

Union Runtime:
Worst case O(1)

Practical Runtime Goals of DS ADT

Suppose we have n elements

Our consideration: an “overall runtime”, ie: what is the total cost of: m finds and $\leq n-1$ unions?

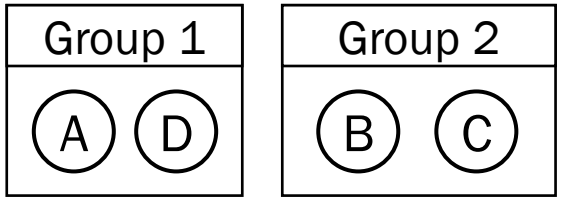
Goal : $O(m + n)$

Why we need a new datastructure?

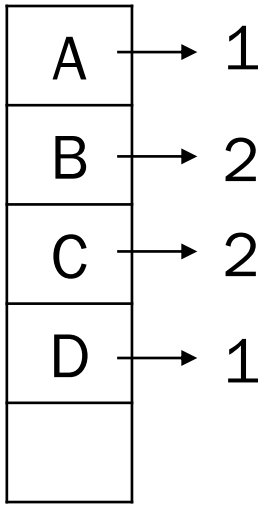
Let’s try to implement the Disjoint Set ADT using a familiar datastructure: HashMap!

Why we need a new datastructure?

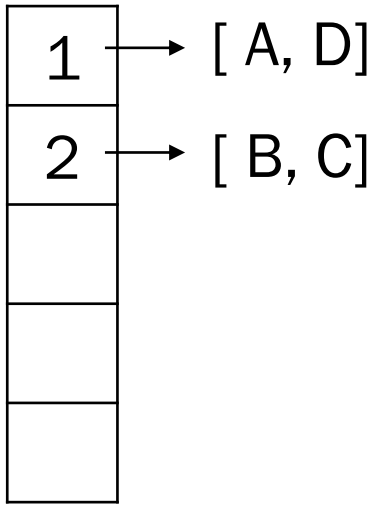
Let's try to implement the Disjoint Set ADT using a familiar datastructure: HashMap! *(assume no hash collisions)*



Approach 1: Value -> Set ID



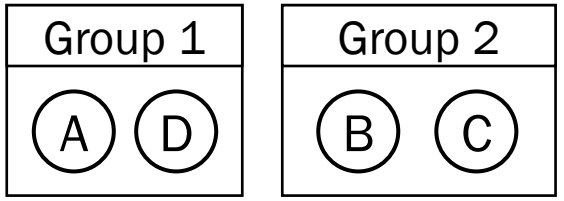
Approach 2: Set ID -> Values (LinkedList)



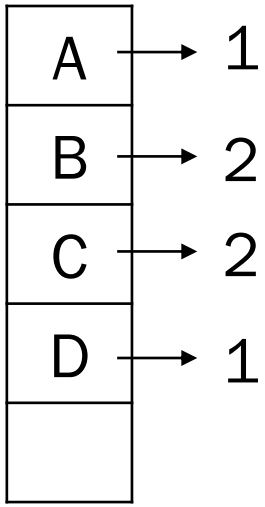
n elements	Find()	Union()
Approach 1		
Approach 2		

Why we need a new datastructure?

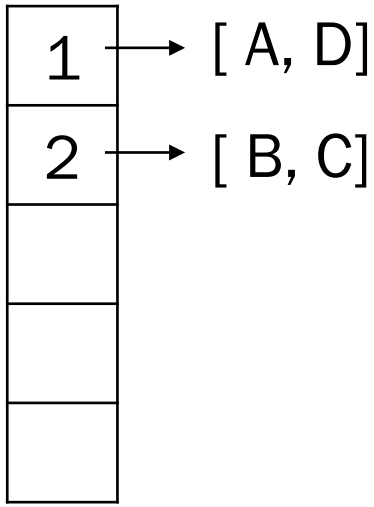
Let's try to implement the Disjoint Set ADT using a familiar datastructure: HashMap! *(assume no hash collisions)*



Approach 1: Value -> Set ID



Approach 2: Set ID -> Values (LinkedList)

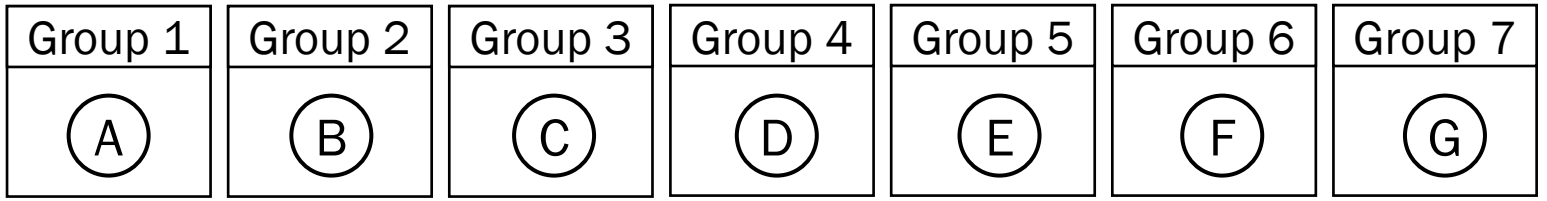


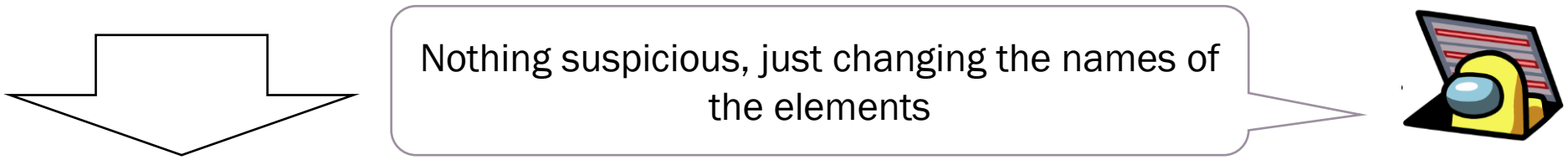
n elements	Find()	Union()
Approach 1	O(1)	O(n)
Approach 2	O(n)	O(1)
New ADT :))	O(log n)	O(1)

Data Structure for DS ADT

- **Observation**: trees let us find many elements given one root…
- **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 set. The name of the set is the tree root

Up-Tree Datastructure

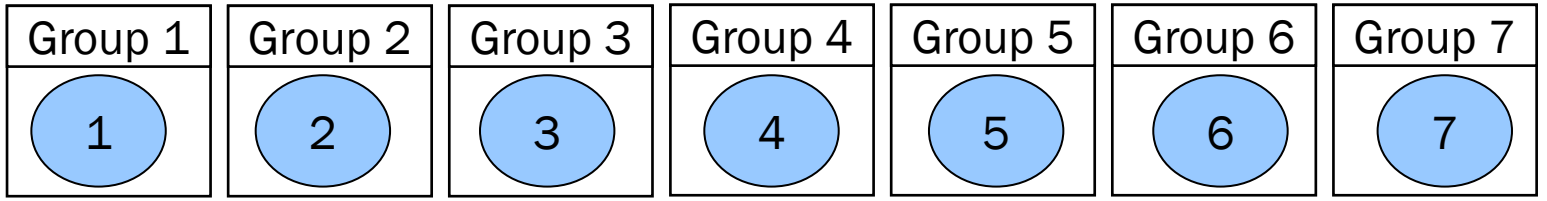




Changing names of elements to be integers for ease of use! (We'll see why in a second)

Can be done using a HashMap or just having an extra field on custom object to go back and forth!

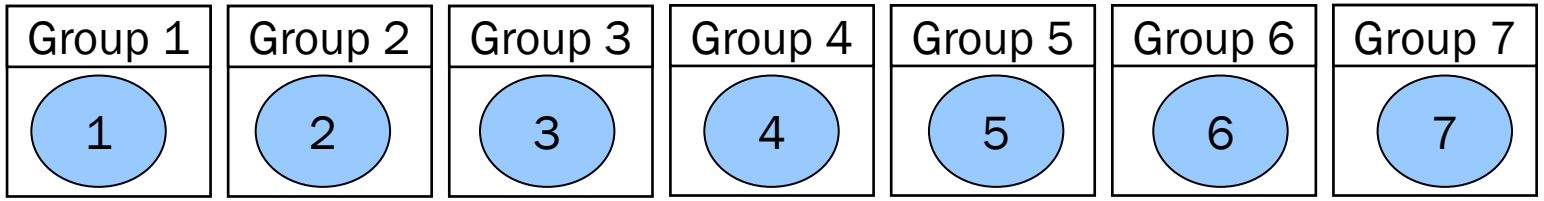
Initial State:



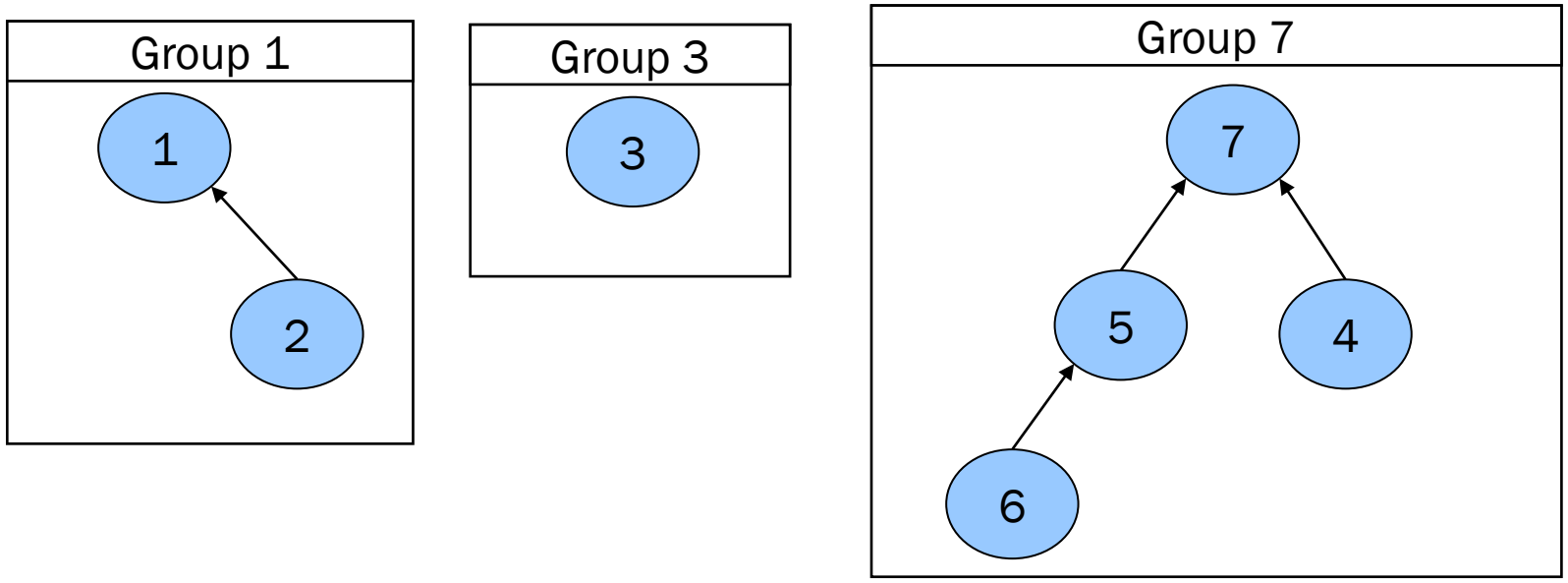
Up-Tree Datastructure

Roots are the names of each set

Initial State:

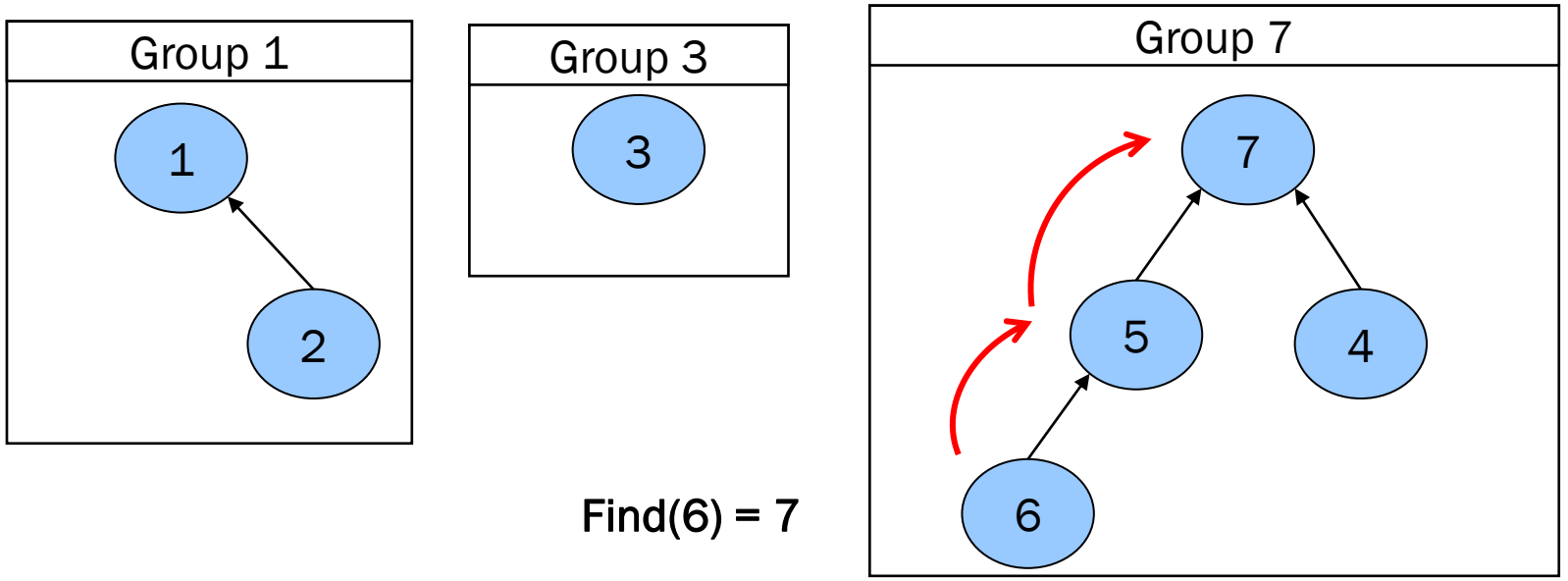


After several unions:



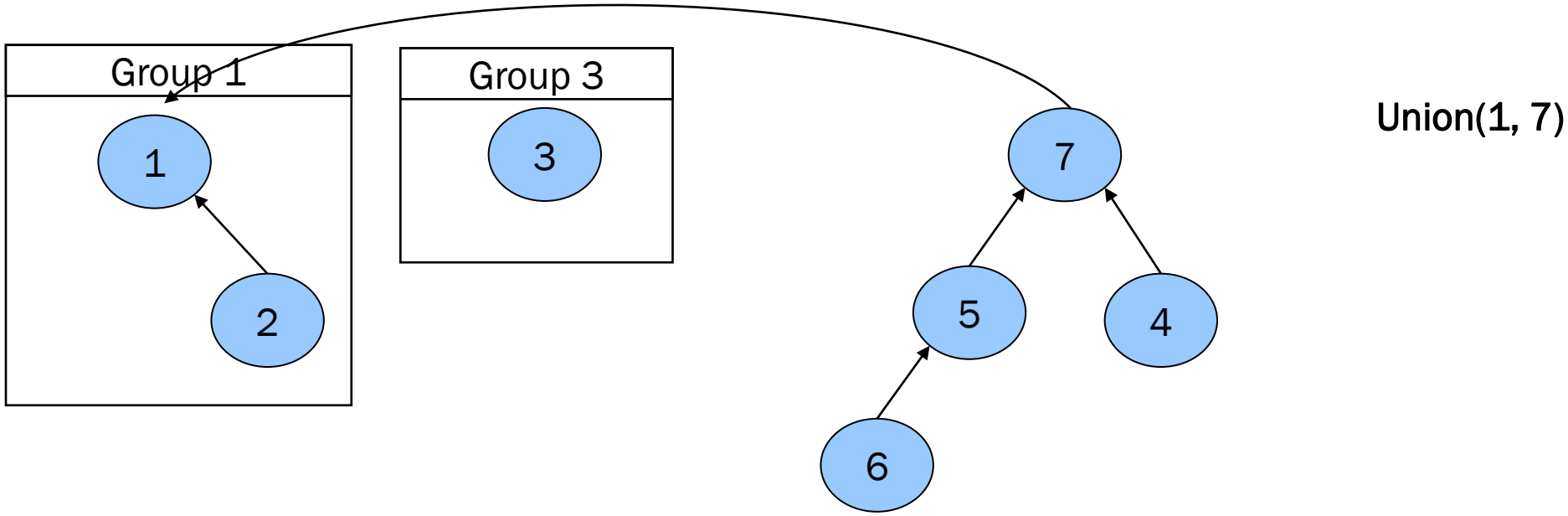
Find Operation

Find(x) – follow x to the root and return the root



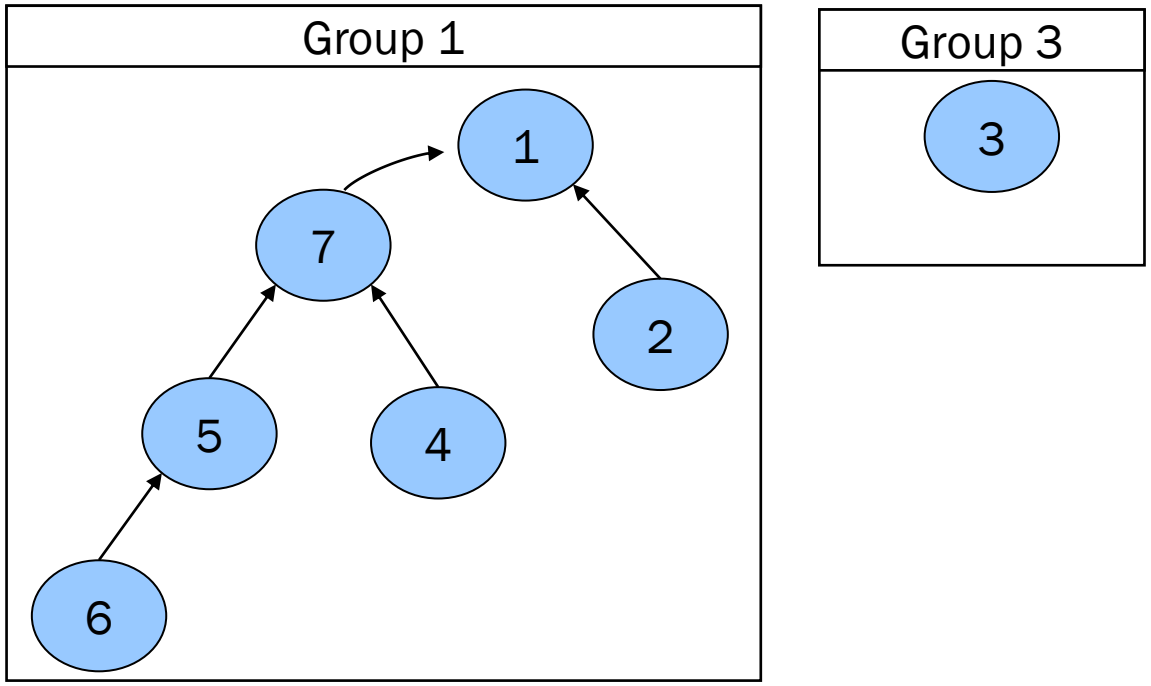
Union Operation

Union(x, y) – point y to x, (assume x, y are names of sets… aka roots)



Union Operation

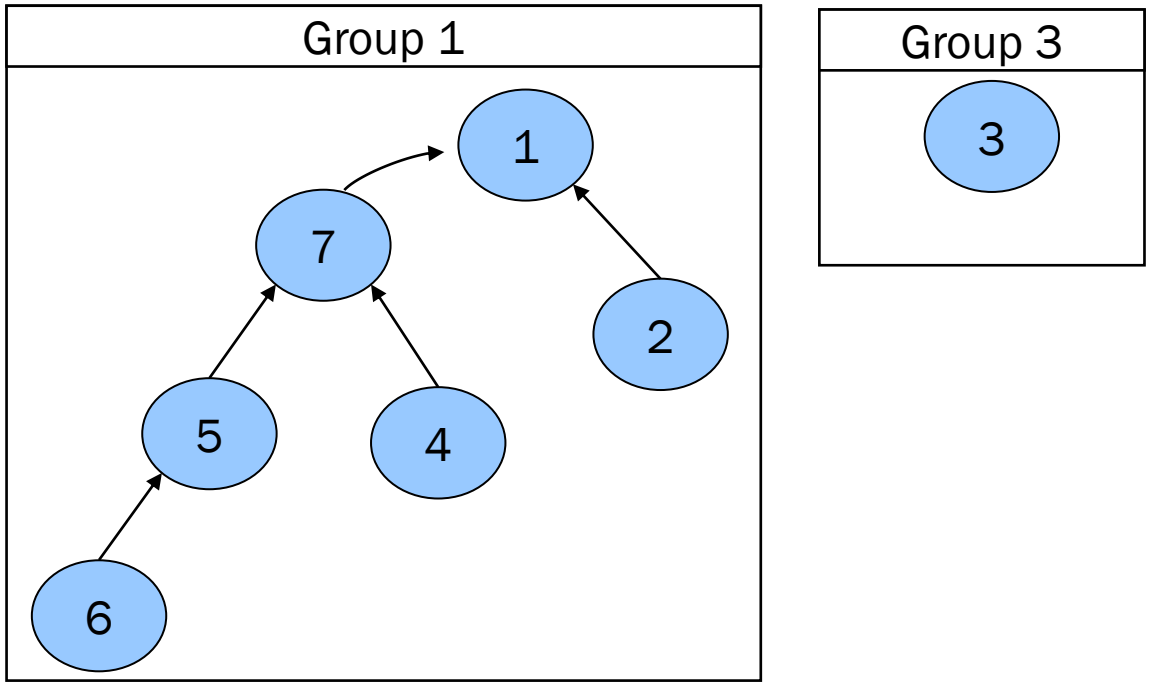
Union(x, y) –point y to x, (assume x, y are names of sets… aka roots)



Union(1, 7)

Union Operation

Union(x, y) –point y to x, (assume x, y are names of sets… aka roots)



What does this union look like?
Union(1, 3)

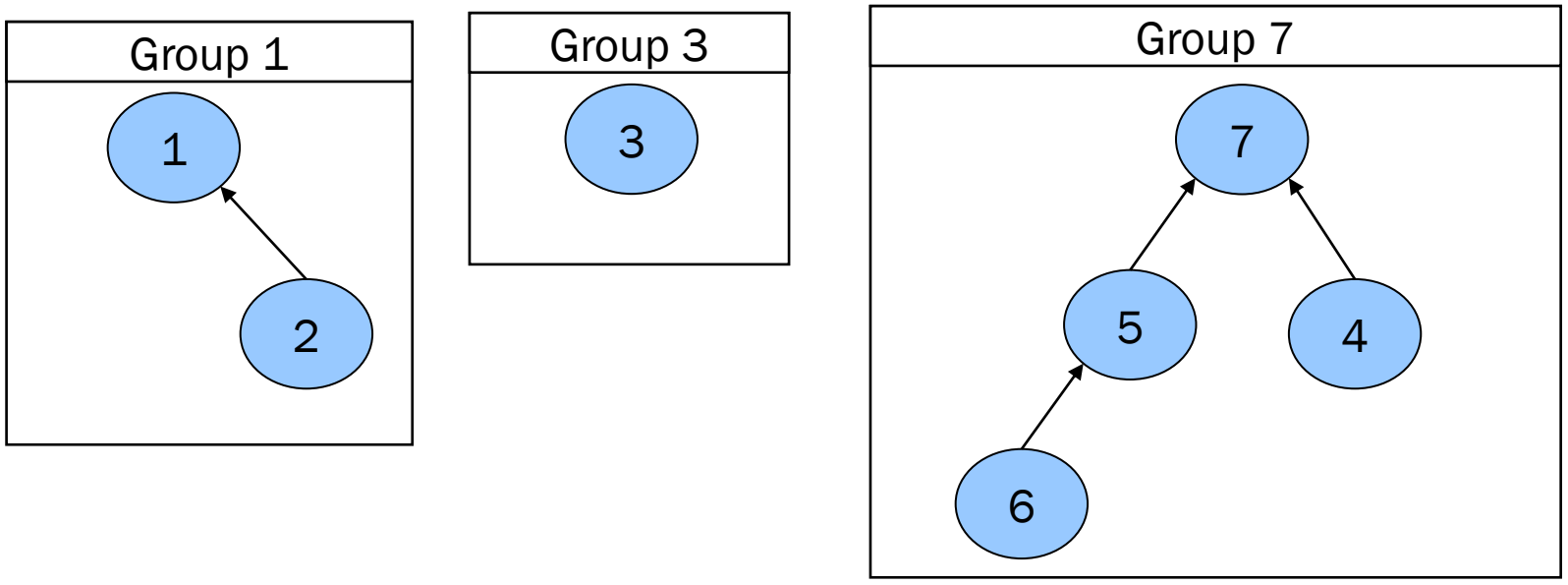
Does NOT have to be a binary tree!

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



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

Draw up-tree from this array:

Implementation

```
int Find(int x) {

  while (up[x] != 0) {
    x = up[x];
  }

  return x;
}
```

```
void Union(int x, int y) {
  up[y] = x
}
```

Worst runtime for Union():

Worst runtime for Find():

Overall Runtime:
(*m* finds and ≤ *n-1* unions)

Goal: *O(m + n)*

Implementation

```
int Find(int x) {

  while (up[x] != 0) {
    x = up[x];
  }

  return x;
}
```

```
void Union(int x, int y) {
  up[y] = x
}
```

Worst runtime for Union():

O(1)

Worst runtime for Find():

O(n)

Overall Runtime: O(mn + n)

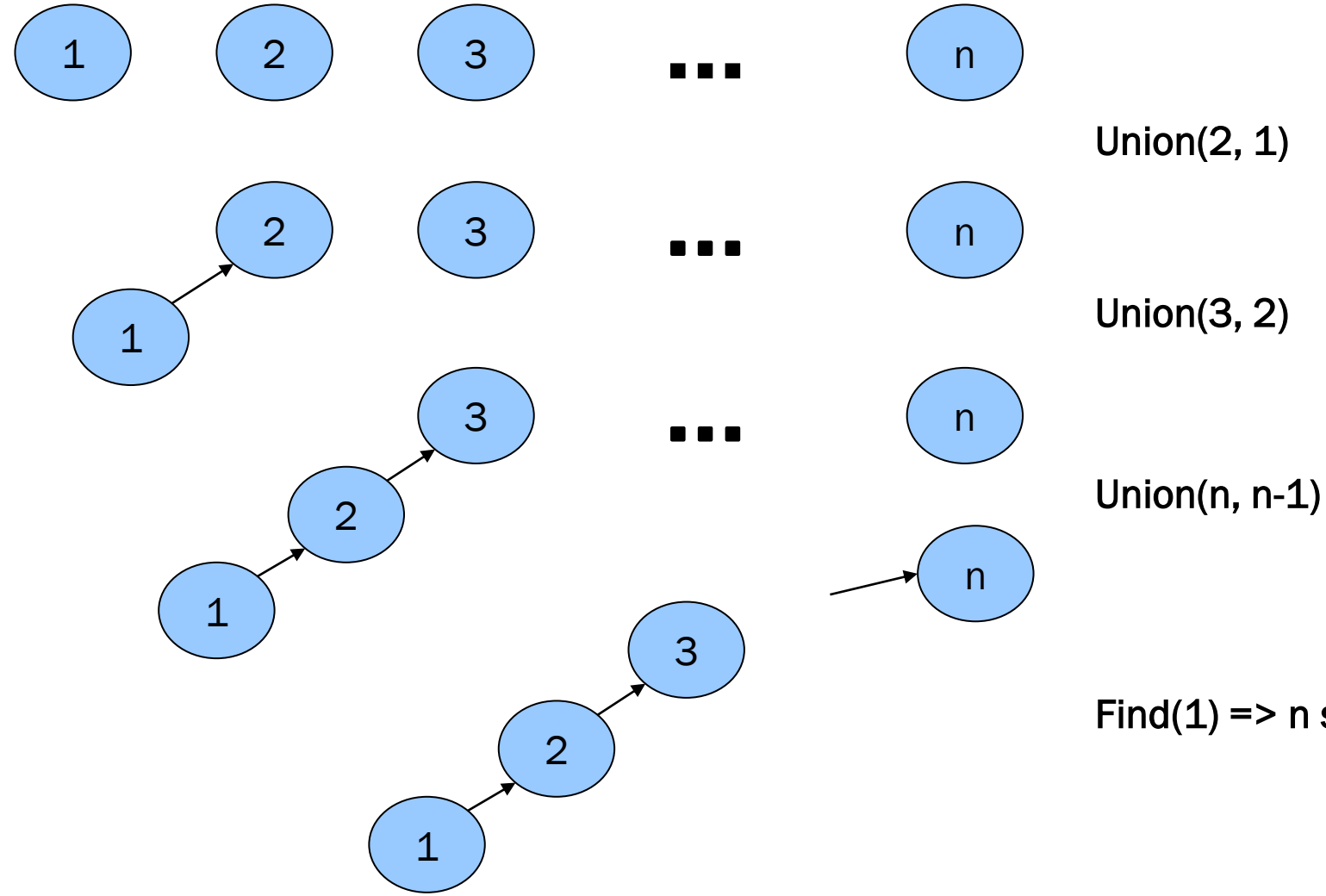
(*m* finds and ≤ *n-1* unions)

Goal: *O(m + n)*

A bad case :(

Union(x, y) - “point y to x”

```
void Union(int x, int y) {
  up[y] = x
}
```



Making Improvements

```
int Find(int x) {

  while (up[x] != 0) {
    x = up[x];
  }

  return x;
}
```

```
void Union(int x, int y) {
  up[y] = x
}
```

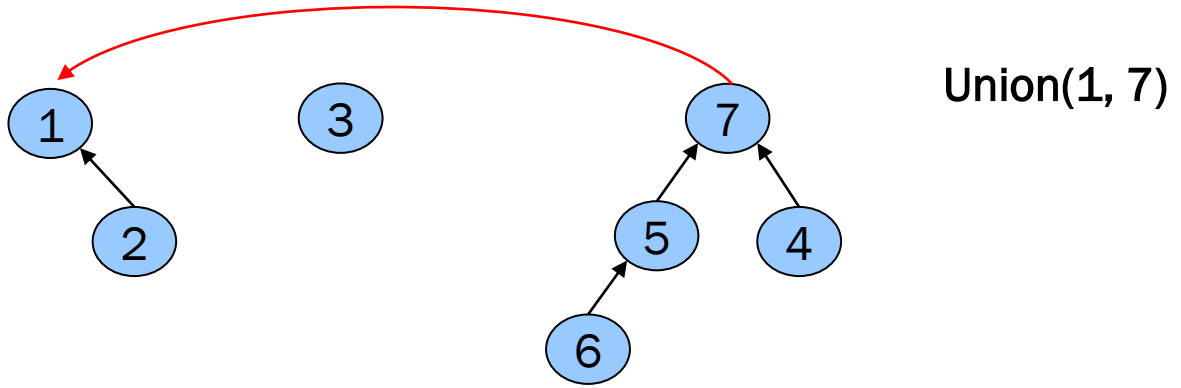
1. Upgrade **Union()** so that **Find()** becomes O(log n)
- Strategy: Union by size
- "Overall" Runtime becomes O(m logn + n)

2. Upgrade **Find()** so it is more optimized
- Strategy: Path compression
- "Overall" becomes almost O(m + n)

Union-by-size / Weighted Union

PREVIOUSLY

Union(x, y) –point y to x, (assume x, y are names of sets… aka roots)



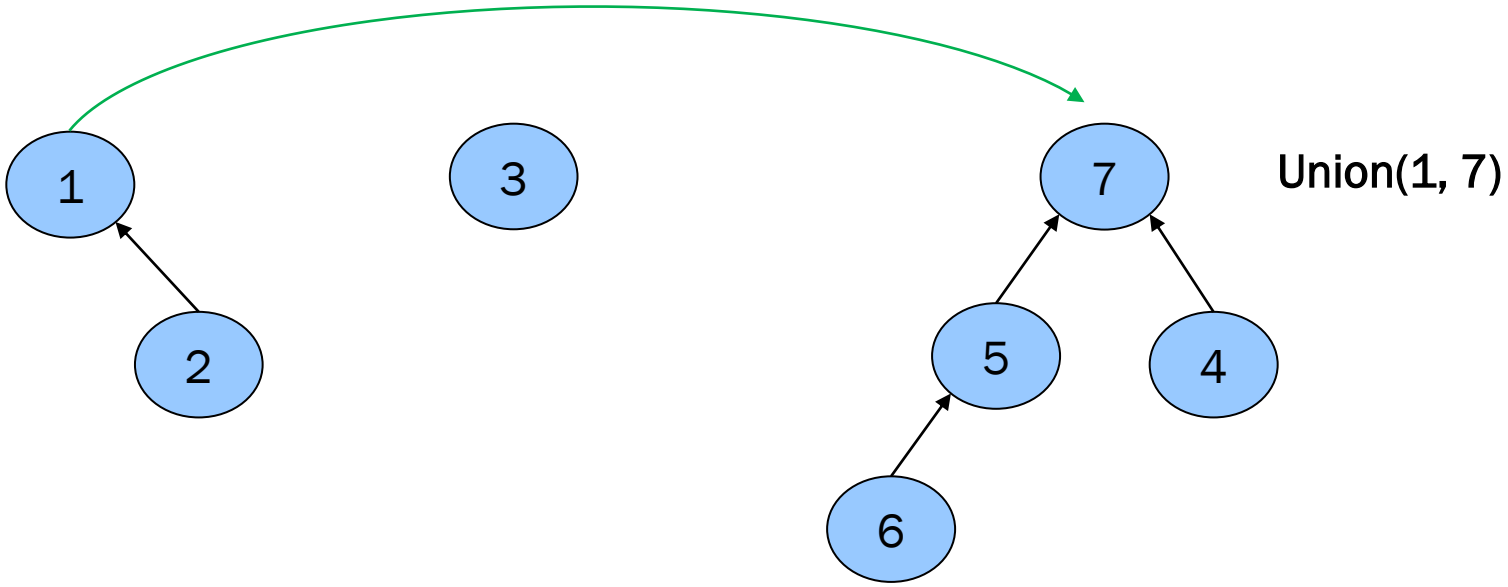
Union(1, 7)

NEW AND IMPROVED!

W-Union(x, y) –always point the smaller (total # of nodes) tree to the root of the larger tree

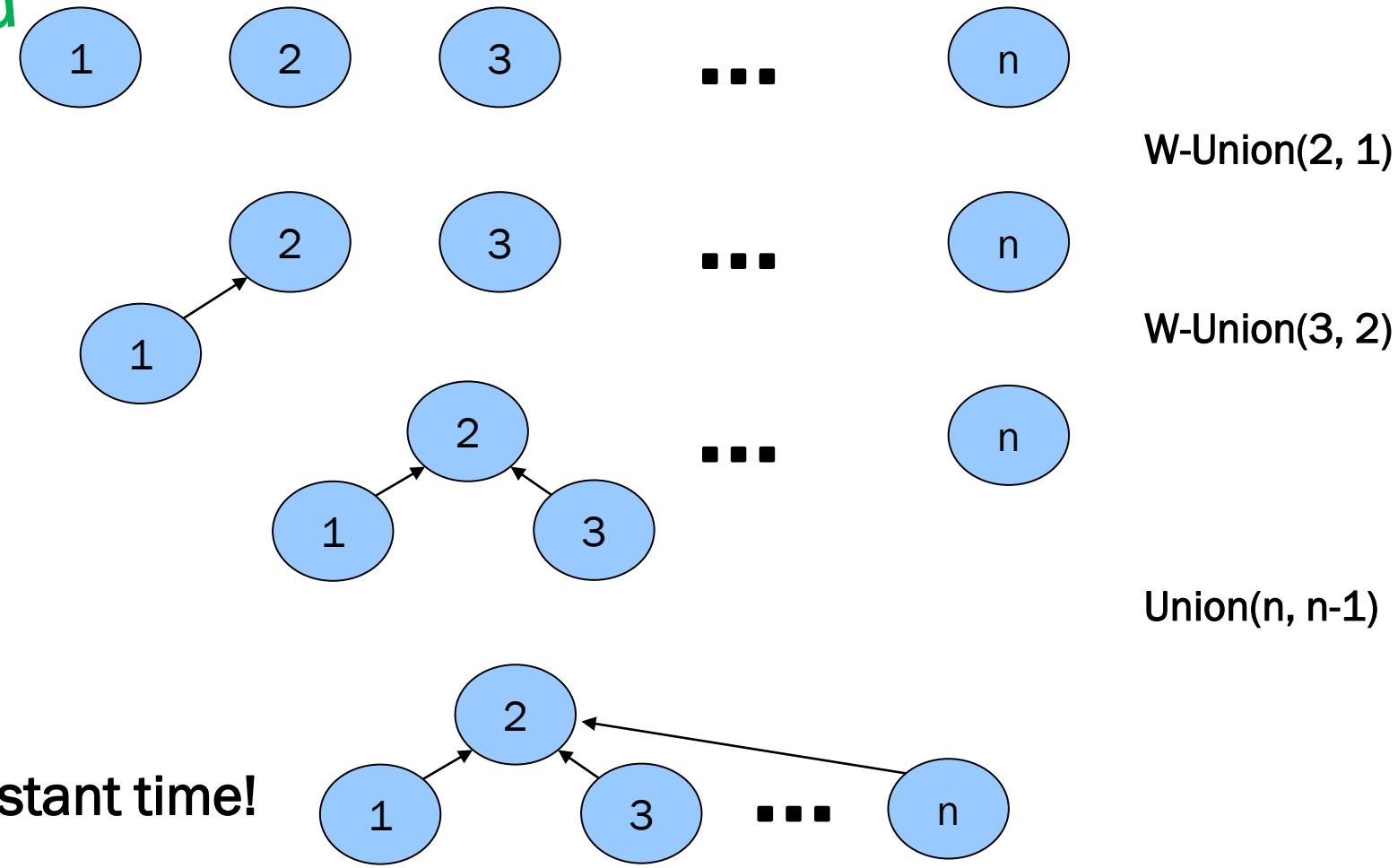
Union-by-size / Weighted Union

W-Union(x, y) –always point the smaller (total # of nodes) tree to the root of the larger tree



A ~~bad~~ case :(

pretty good



Find(1) constant time!

Analysis of Weighted Union Guarantees

- **Observation**: runtime of Find() is bounded by height of up-tree

- **Idea**: find relationship between height and *n* so that we can determine runtime bound for Find()

Analysis of Weighted Union Guarantees

- **Claim:** Weighted-Union up-tree of height *h* has weight (ie: number of nodes) *at least* 2^h
- **Proof by induction:**
  - **Basis:** _____
  - **Inductive Hypothesis:** _____
  - **Inductive Step:**

Read the book ☺

Informal Analysis of Weighted Union Guarantees

If we want to increase the height of a weighted union uptree,

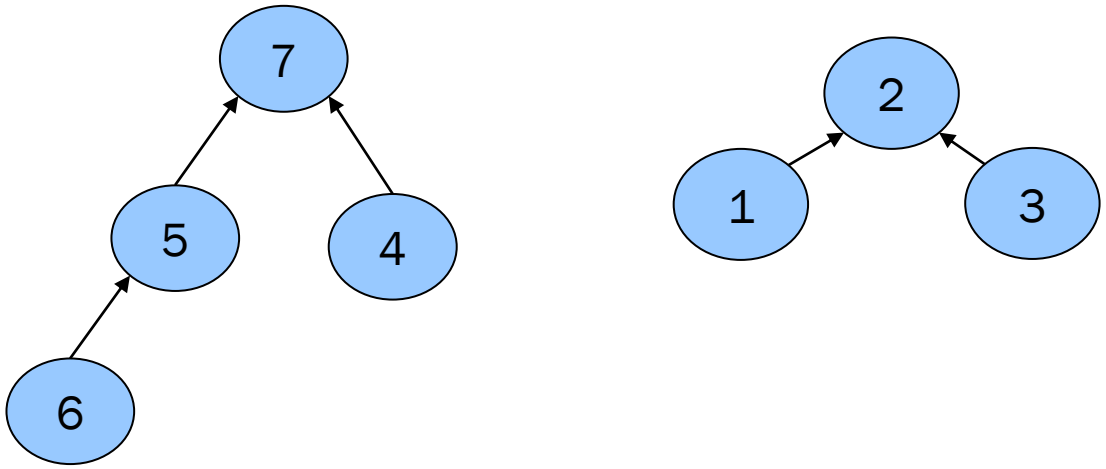
we need to union another tree with the same height (or larger)

Informal Analysis of Weighted Union Guarantees

If we want to increase the height of a weighted union uptree,

we need to union another tree with the same height (or larger)

Why? Because what if the other tree is shorter?



Nice analysis, so what?

If number of nodes doubles as we increase height

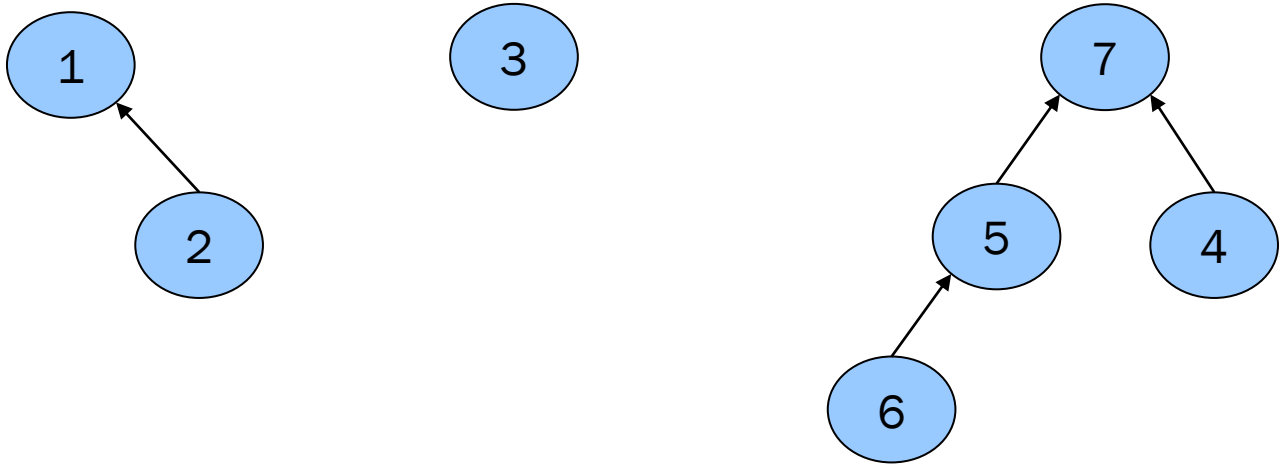
$$n \approx 2^h$$

Then height grows in log(n)

$$\log(n) \approx h$$

➔ Find() is O(log(n))

Array Implementation



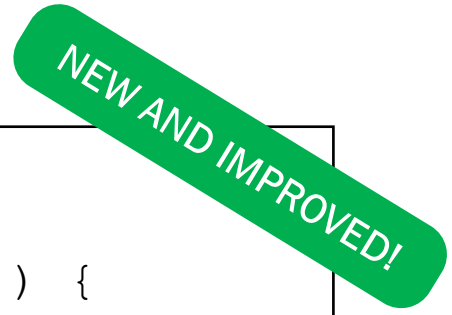
	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0
weight	2		1				4

Weighted Union Upgrade!

```
int Find(int x) {

  while (up[x] != 0) {
    x = up[x];
  }

  return x;
}
```



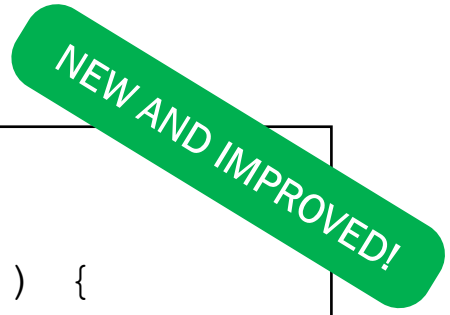

```
void Union(int x, int y) {
  // Is x smaller?
  if (weight[x] < weight[y]) {
    up[x] = y;
    weight[y] += weight[x]
  } else {
    up[y] = x;
    weight[x] += weight[y]
  }
}
```

Weighted Union Upgrade!

```
int Find(int x) {

  while (up[x] != 0) {
    x = up[x];
  }

  return x;
}
```




```
void Union(int x, int y) {
  // Is x smaller?
  if (weight[x] < weight[y]) {
    up[x] = y;
    weight[y] += weight[x]
  } else {
    up[y] = x;
    weight[x] += weight[y]
  }
}
```

New worst-case runtime for Union():

O(1)

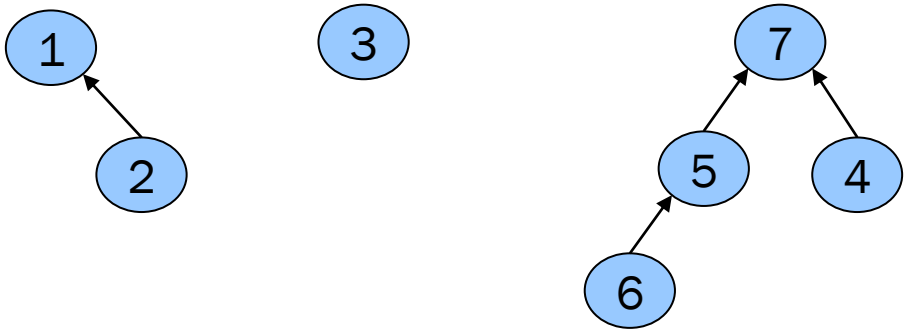
New worst-case runtime for Find():

O(logn)

New "overall" runtime for *m* finds and ≤ *n-1* unions: O(mlogn + n)

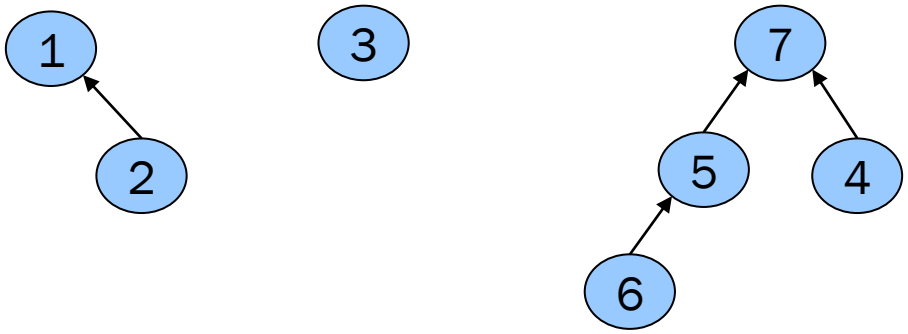
Nifty Storage Trick

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



Nifty Storage Trick

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



Store –size for the roots!

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

Weighted Union Upgrade!

```
void Union(int x, int y) {
  // Is x smaller?
  if (weight[x] < weight[y]) {
    up[x] = y;
    weight[y] += weight[x]
  } else {
    up[y] = x;
    weight[x] += weight[y]
  }
}
```

1. Upgrade **Union()** so that **Find()** becomes O(log n)

- Strategy: Union by size
- "Overall" Runtime becomes O(m logn + n)

(m finds, n-1 unions)

Next Upgrade! PC

```
int Find(int x) {

  while (up[x] != 0) {
    x = up[x];
  }

  return x;
}
```

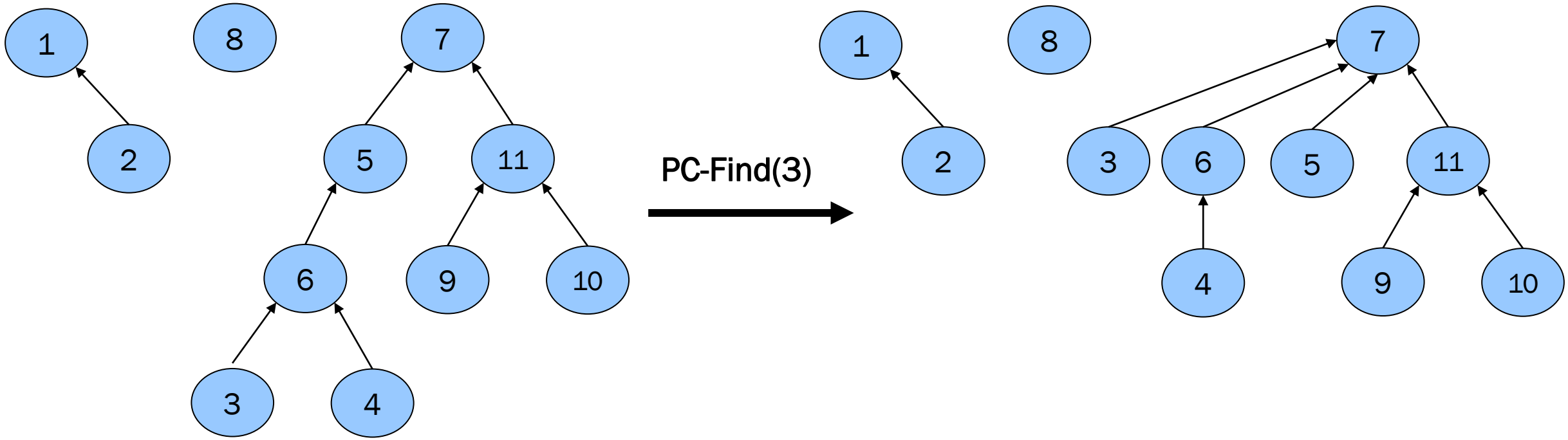
Next up!

2. Upgrade **Find()** so it is more optimized

- Strategy: Path compression
- "Overall" becomes almost O(m + n)

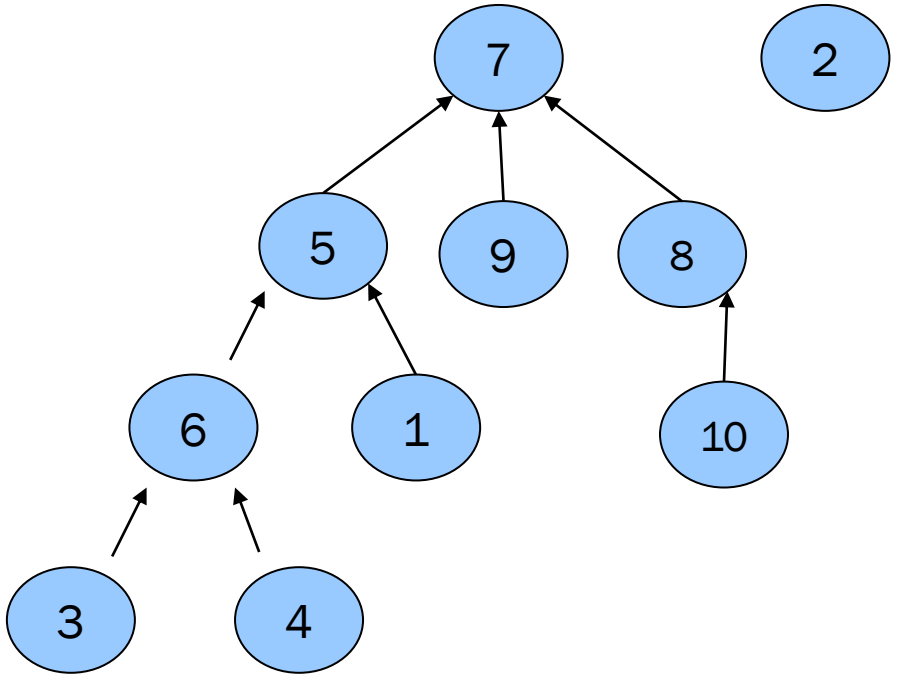
Path Compression

On a Find() operation, point all the nodes on the search path directly to the root



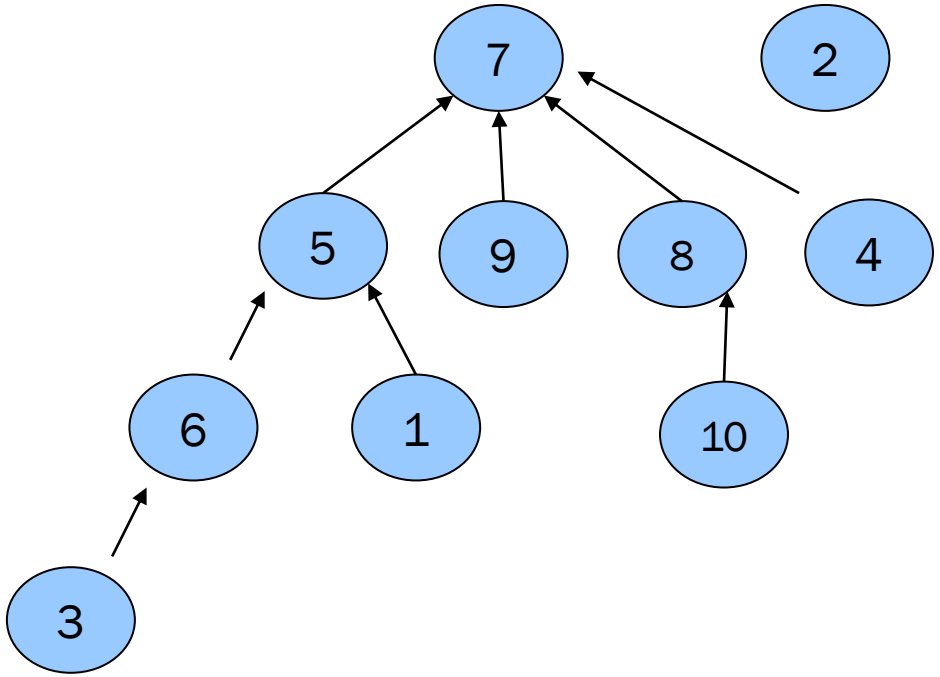
Try it yourself

Draw result of PC-Find(4)



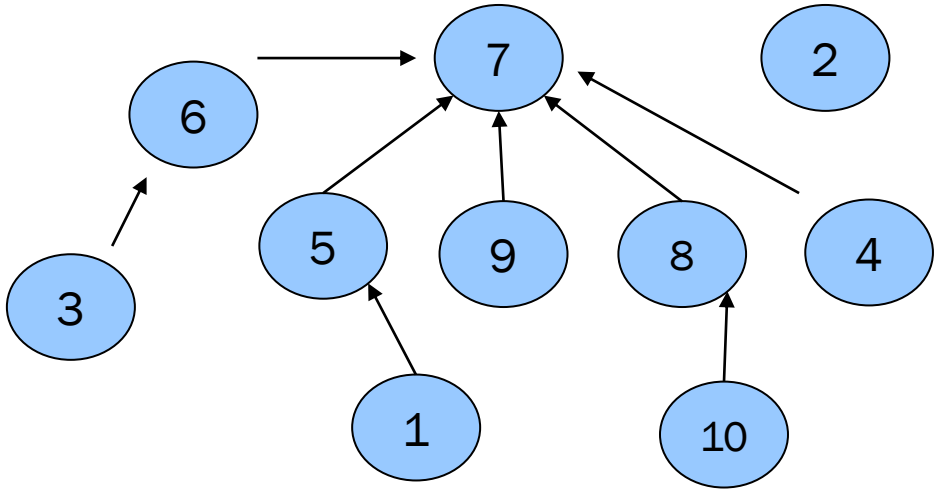
Try it yourself

Draw result of PC-Find(4)



Try it yourself

Draw result of PC-Find(4)



Path Compression Find

NEW AND IMPROVED!

```
int PC-Find(int x) {
  root = x
  while (up[root] != 0) {
    root = up[root]
  }

  update = x
  while (up[update] != 0) {
    temp = up[update]
    up[update] = root
    update = temp
  }

  return root;
}
```

New worst-case runtime for Find():

O(logn)

Aw, still the same?

Interlude: A Really Slow Function

iterated logarithm (log star):

log* x = number of times you compute log to bring value down to at most 1

log* 2 = 1

log* 4 = log* 2^2 = 2

log* 16 = log* 2^{2^2} = 3

log* 65536 = log* $2^{2^{2^2}}$ = 4

log* 2^{65536} = ... = 5

(log log log 16 = 1)

(log log log log 65536 = 1)

(log log log log log 2^{65536} = 1)

m union and find operations on a set of *n* elements has worst case complexity of O(m log*n)

Interlude: An even slower Function

Ackermann's function: is a REALLY REALLY big function A(x, y) with inverse α(x, y) which is REALLY REALLY small

But get this: α(x, y) grows even slower than log* n !!

human brain contains about
170.68 billion cells = $10^{11.23}$ cells

Herculano-Houzel, S. 2016. *The Human Advantage: A New Understanding of How Our Brain Became Remarkable*. Cambridge, MA: The MIT Press.

Number of atoms in the observable universe ≈ **10^{82} atoms**

https://www.livescience.com/how-many-atoms-in-universe.html

$$A(4, 2) = 2^{65536} \approx 10^{19739}$$

$$\alpha(10^{90}, 10^{11}) = 4$$

(look up Ackermann's function on Wikipedia if interested)

Okay back to Path Compression

Tarjan proved that, with these optimizations, m union and find operations on a set of n elements have worst case complexity of $O(m \cdot \alpha(m, n))$

Which for *all practical purposes* is amortized constant time:

$O(m \cdot 4)$ for m operations

Looking back :,)

PREVIOUSLY

```
int Find(int x) {

  while (up[x] != 0) {
    x = up[x];
  }

  return x;
}
```

PREVIOUSLY

```
void Union(int x, int y) {
  up[y] = x
}
```

1. Upgrade **Union()** so that **Find()** becomes O(log n)
- Strategy: Union by size
- "Overall" Runtime becomes O(m logn + n)

2. Upgrade **Find()** so it is more optimized
- Strategy: Path compression
- "Overall" becomes <u>almost</u> O(m + n)

Another Use-case Maze Creation:

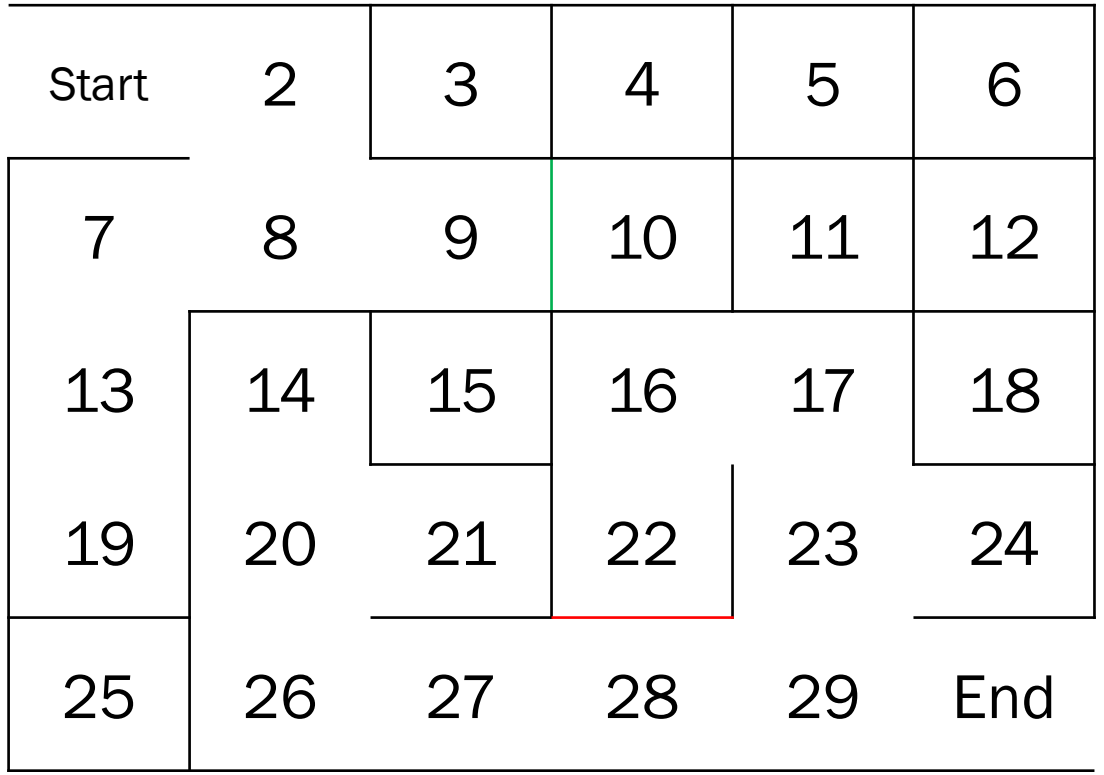
How can you use disjoint sets to create mazes? (Same question: what edges should be included?)

Start	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	End

Another Use-case Maze Creation:

How can you use disjoint sets to create mazes? (Same question: what edges should be included / what properties should maze have?)

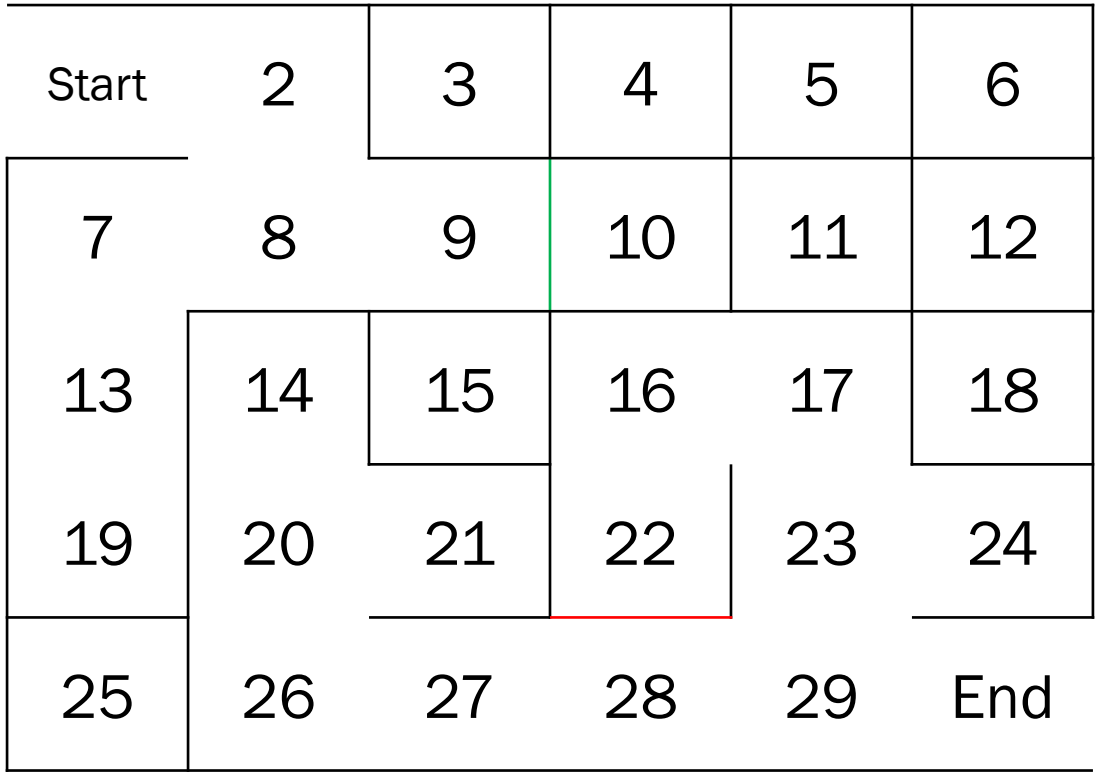
- Start and end must be connected
- Outside edges included
- All squares can be reached from every other square
- Cycles are bad



Another Use-case Maze Creation:

- Start and end must be connected
- Outside edges included
- All squares can be reached from every other square
- Cycles are bad

```
E = all non-outside edges
DS = ({start}, {2}, {3}, … {end})
while size(DS) > 1:
    pick random unused edge from e
    if (find(u) != find(v)):
            union(u, v)
            remove e from E
    else:
            mark e as used
Return E
```



Conclusion

We have a pretty clever and cool datastructure for the DS ADT :D

- Understand the implementation
- Understand the improvements and runtime analysis

Have a great weekend!