Data Structures

Disjoint Sets 2

CS 225

Brad Solomon

March 25, 2026

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

Continue to improve implementation of disjoint sets

Discuss how improvements affect efficiency

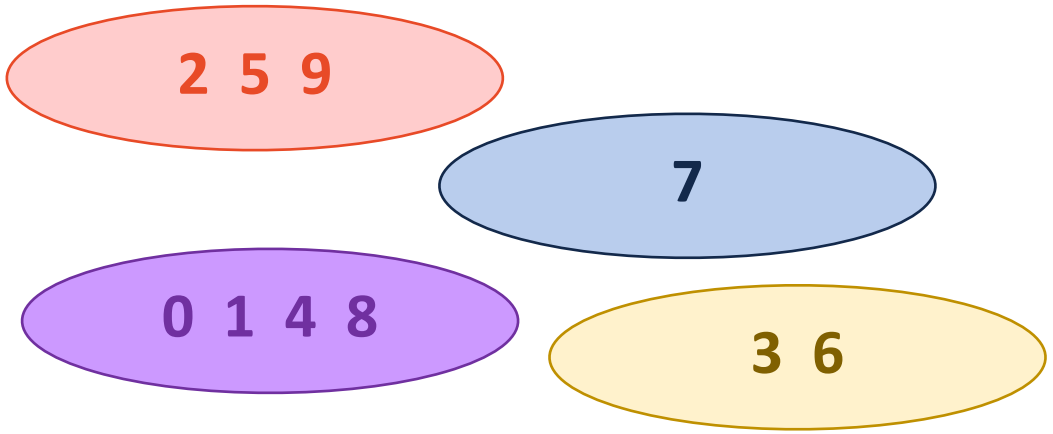
Disjoint Sets

ADT:

```
makeSet(vector<T> items)

Find(T key)

Union(T k1, T k2)
```



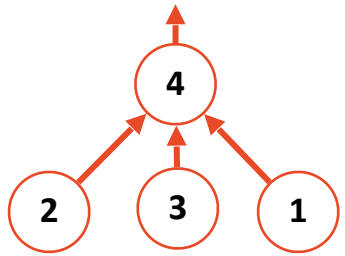
Key Ideas:

Every item exists in exactly one set

Every item in each set has same representation

Every set has a different representation

Disjoint Sets – Best and Worst UpTree

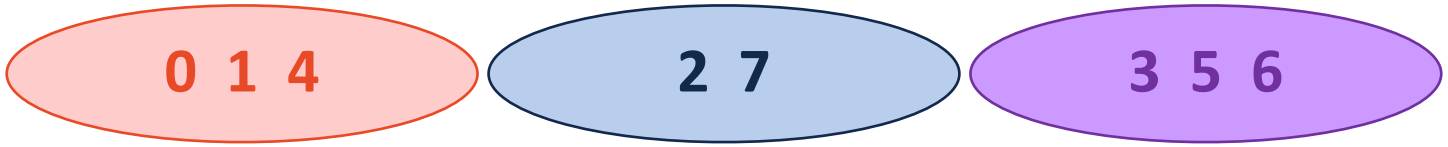


0	1	2	3	4
	3	4	2	-1

0	1	2	3	4
	4	4	4	-1

Disjoint Set Implementation

Store an UpTree as an array, canonical items store **height** / **size**



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

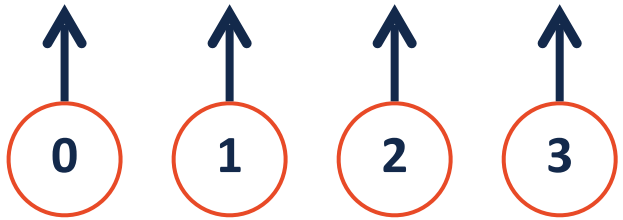
Find(k): Repeatedly look up values until **negative value**

Union(k_1, k_2): Update ***smaller*** canonical item to point to larger

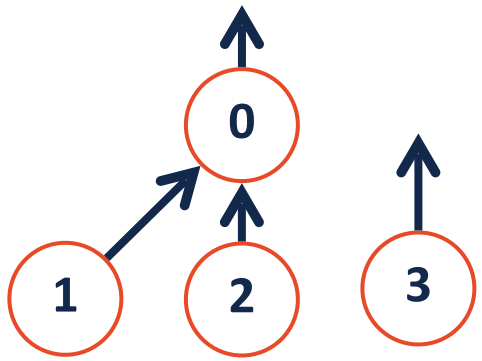
Update value of remaining canonical item

Disjoint Sets Union by Size

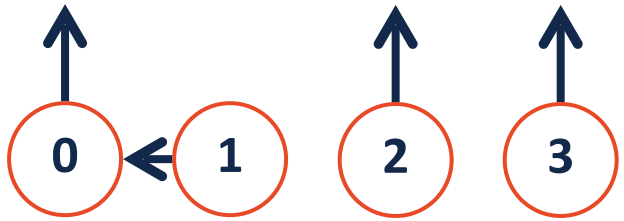
Join Code: 225



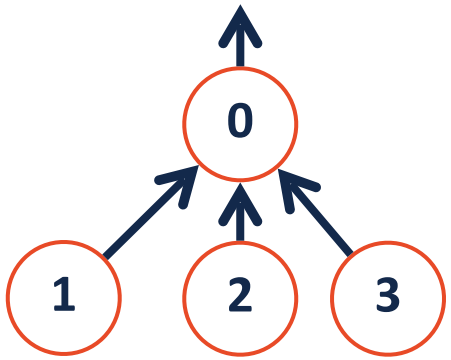
0	1	2	3
-1	-1	-1	-1



0	1	2	3



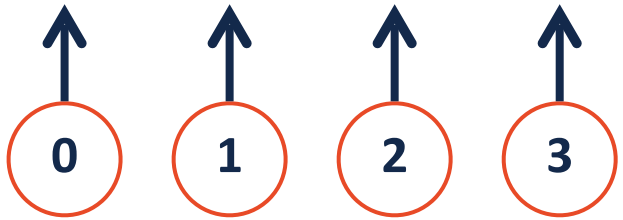
0	1	2	3



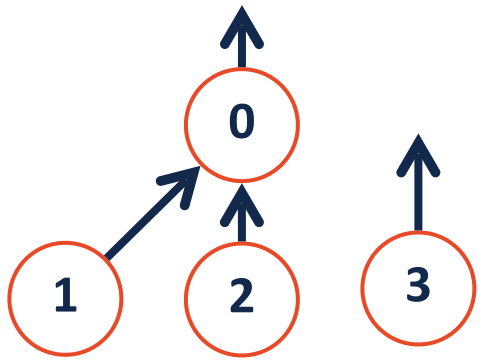
0	1	2	3

Disjoint Sets Union by Size

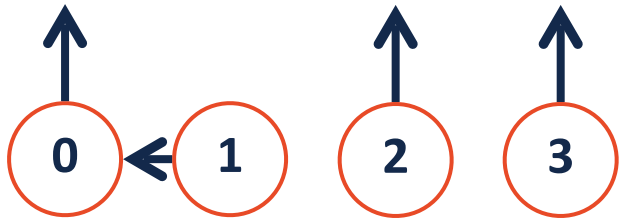
Join Code: 225



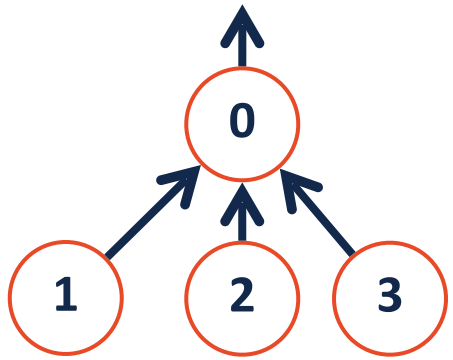
0	1	2	3
-1	-1	-1	-1



0	1	2	3
-3	0	0	-1



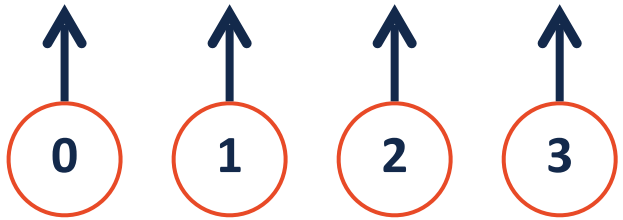
0	1	2	3
-2	0	-1	-1



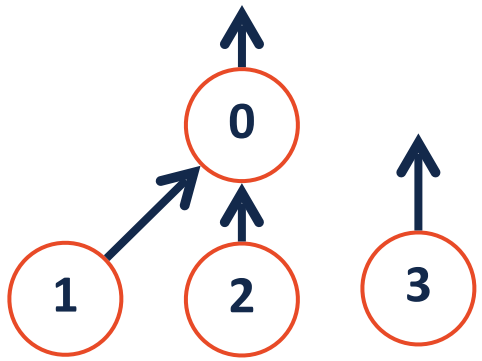
0	1	2	3
-4	0	0	0

Disjoint Sets Union by Height

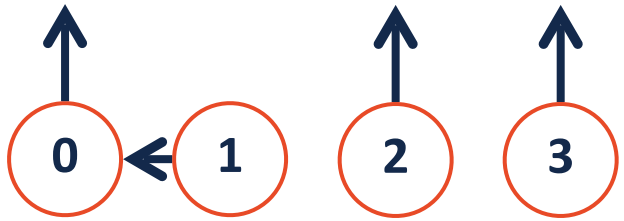
Join Code: 225



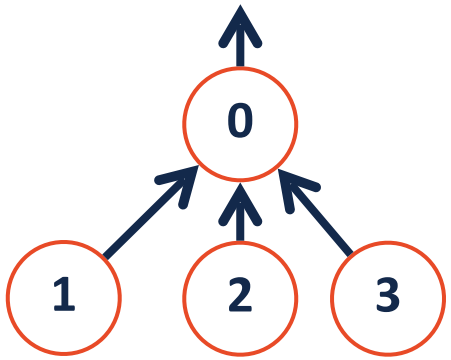
0	1	2	3
-1	-1	-1	-1



0	1	2	3



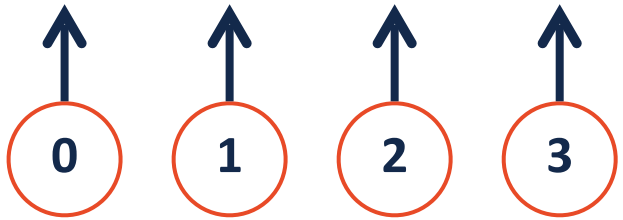
0	1	2	3



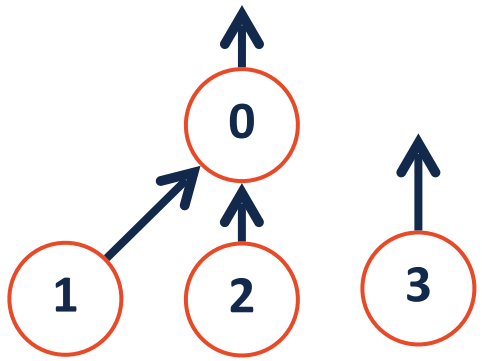
0	1	2	3

Disjoint Sets Union by Height

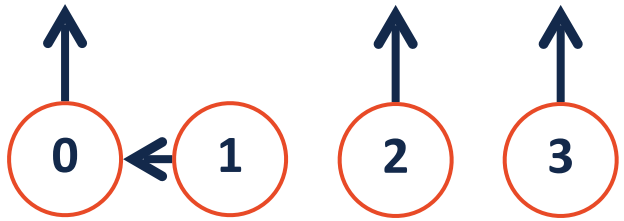
Join Code: 225



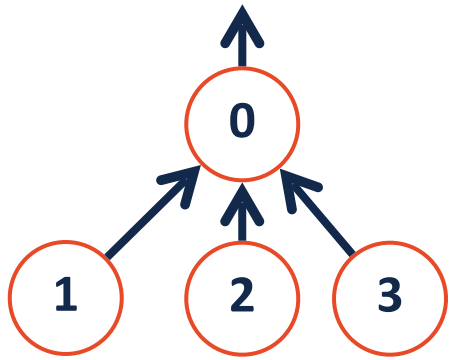
0	1	2	3
-1	-1	-1	-1



0	1	2	3
-2	0	0	-1



0	1	2	3
-2	0	-1	-1

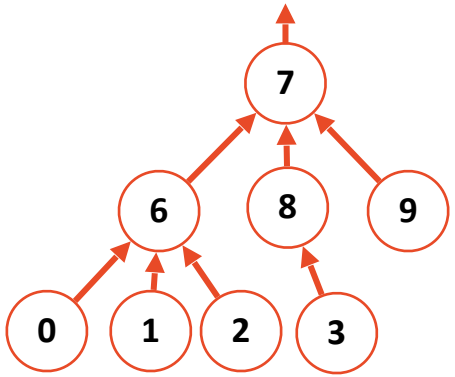


0	1	2	3
-2	0	0	0

Disjoint Sets – Smart Union

Two O(1) methods of combining two sets

Claim: Both limit height to: O(log n).



	Before Union			After Union		
Union by height	4	…	7	4	…	7
	-4		-3	-4		4
Union by size	4	…	7	4	…	7
	-8		-4	7		-12

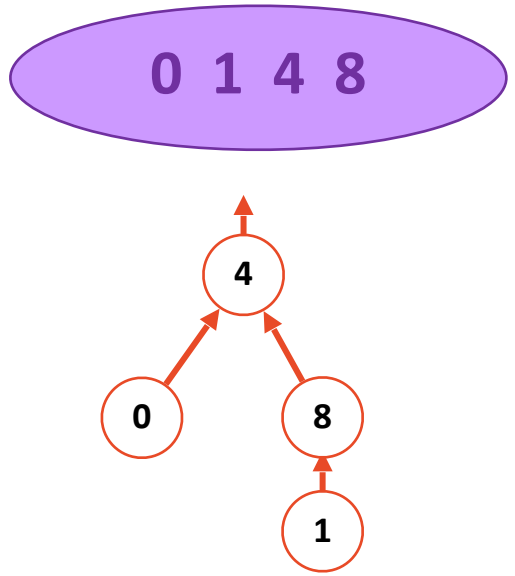
***Idea**: Keep the height of the tree as small as possible.*

***Idea**: Minimize the number of nodes that increase in height*

Disjoint Sets Find

Find(1)

```
1  int DisjointSets::find(int i) {
2    if ( s[i] < 0 ) { return i; }
3    else { return find( s[i] ); }
4  }
```



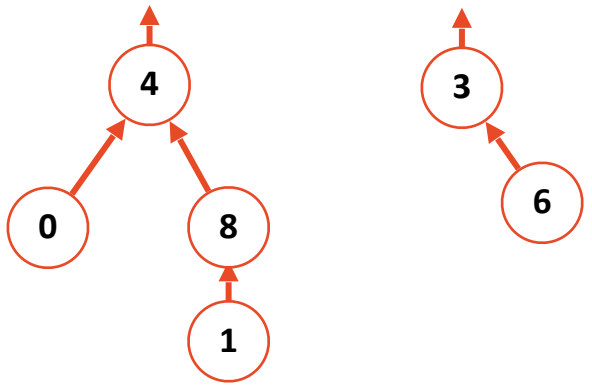
Does implementation work on **height / size**?

0	1	2	3	4	5	6	7	8	9
4	8			-3/-4				4	

Disjoint Sets Union

unionBySize(4, 3)

```
 1  void DisjointSets::unionBySize(int root1, int root2) {
 2    int newSize = arr_[root1] + arr_[root2];
 3
 4    if ( arr_[root1] < arr_[root2] ) {
 5
 6      arr_[root2] = root1;
 7
 8      arr_[root1] = newSize;
 9
10    } else {
11
12      arr_[root1] = root2;
13
14      arr_[root2] = newSize;
15
16    }
  }
```



0	1	2	3	4	5	6	7	8	9
4	8		-2	-4		3		4	

Disjoint Sets Union by Size

Claim: Sets unioned by size have a height of at most $O(\log_2 n)$

Claim: An UpTree of height **h** has nodes $\geq$ _____

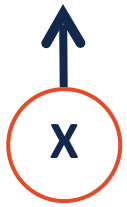
Base Case:

Disjoint Sets Union by Size

Claim: Sets unioned by size have a height of at most $O(\log_2 n)$

Claim: An UpTree of height **h** has nodes $\geq 2^h$

Base Case: $h = 0$



Base case height is 0, has one node.

vs.

$$2^0 = 1$$

Base case holds!

Disjoint Sets Union by Size

Claim: An UpTree of height **h** has nodes $\geq 2^h$

IH:

Disjoint Sets Union by Size

Claim: An UpTree of height **h** has nodes $\geq 2^h$

IH: Claim is true for $< i$ unions, prove for ith union (sets A and B).

(We have done $i - 1$ total unions and plan to do ***one*** more)

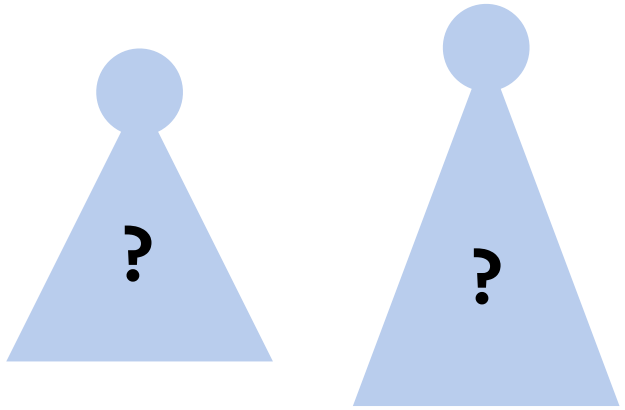
Without loss of generality, let B be the larger set **BY SIZE**

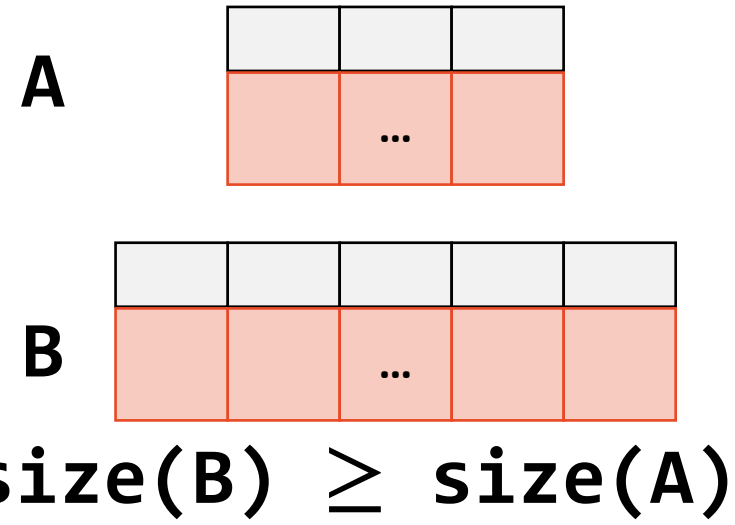
We must explore how height changes for each case:

Case 1: h(A) < h(B)

Case 2: h(A) == h(B)

Case 3: h(A) > h(B)





Disjoint Sets Union by Size

size(B) $\geq$ size(A)

Claim: An UpTree of height **h** has nodes $\geq 2^h$

IH: Claim is true for $< i$ unions, prove for ith union (sets A and B).

Case 1: height(A) < height(B)

Disjoint Sets Union by Size

size(B) $\geq$ size(A)

Claim: An UpTree of height **h** has nodes $\geq 2^h$

IH: Claim is true for $< i$ unions, prove for ith union (sets A and B).

Case 1: height(A) < height(B)

Ideal case where size and height in agreement!

Height doesn't change $\left(h(B') = h(B)\right)$.

By IH: $size(A) \geq 2^{h(A)}$ $size(B) \geq 2^{h(B)}$

$$size(B') = size(A) + size(B) = 2^{h(A)} + 2^{h(B)} \geq 2^{h(B)} = 2^{h(B')}$$

Disjoint Sets Union by Size

size(B) $\geq$ size(A)

Claim: An UpTree of height **h** has nodes $\geq 2^h$

IH: Claim is true for $< i$ unions, prove for ith union (sets A and B).

Case 2: height(A) == height(B)

Disjoint Sets Union by Size

size(B) ≥ size(A)

Claim: An UpTree of height **h** has nodes $\geq 2^h$

IH: Claim is true for $< i$ unions, prove for ith union (sets A and B).

Case 2: height(A) == height(B)

If we merge two equal height trees, height always increase by 1

By IH: $size(A) \geq 2^{h(A)}$ $size(B) \geq 2^{h(B)}$

$$size(B') = size(A) + size(B) = 2^{h(A)} + 2^{h(B)}$$

$$= 2^{h(B)} + 2^{h(B)}$$

$$= 2 * 2^{h(B)} = 2^{h(B)+1} \geq 2^{h(B')}$$

Disjoint Sets Union by Size

size(B) $\geq$ size(A)

Claim: An UpTree of height **h** has nodes $\geq 2^h$

IH: Claim is true for $< i$ unions, prove for ith union (sets A and B).

Case 3: height(A) > height(B)

Disjoint Sets Union by Size

size(B) $\geq$ size(A)

Claim: An UpTree of height **h** has nodes $\geq 2^h$

IH: Claim is true for $< i$ unions, prove for ith union (sets A and B).

Case 3: height(A) > height(B)

Merging taller tree into smaller — height increase to height(A)+1!

By IH: $size(A) \geq 2^{h(A)}$ $size(B) \geq 2^{h(B)}$

$$size(B') = size(A) + size(B) \geq 2\ size(A)$$

$$= 2 * 2^{h(A)} = 2^{h(A)+1} \geq 2^{h(B')}$$

Disjoint Sets Union by Size

size(B) $\geq$ size(A)

Proven: An UpTree of height **h** has nodes $\geq 2^h$

IH: Claim is true for $< i$ unions, prove for ith union.

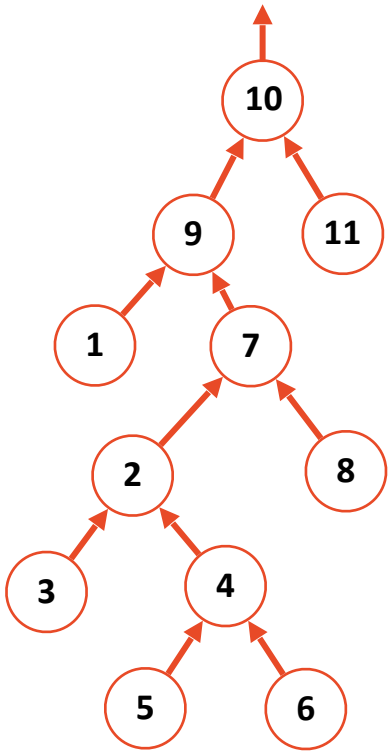
Each case we saw we have $n \geq 2^h$.

Disjoint Sets Find

Find(6)

```
1  int DisjointSets::find(int i) {
2    if ( s[i] < 0 ) { return i; }
3    else { return find( s[i] ); }
4  }
```

As we walk up a tree, why cant we fix it?

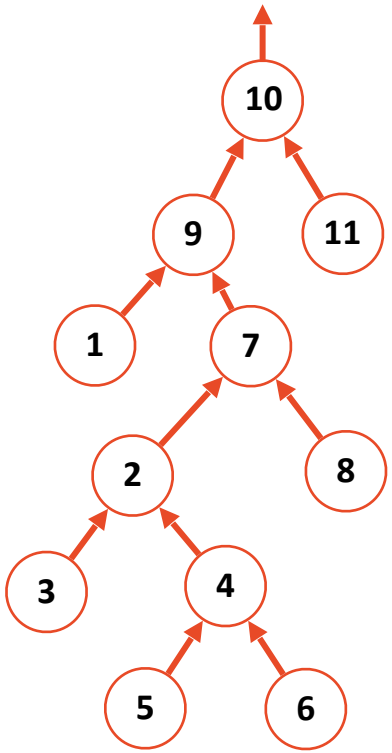


Disjoint Sets Find

Find(6)

```
1  int DisjointSets::find(int i) {
2    if ( s[i] < 0 ) { return i; }
3    else { return find( s[i] ); }
4  }
```

As we walk up a tree, why cant we fix it?

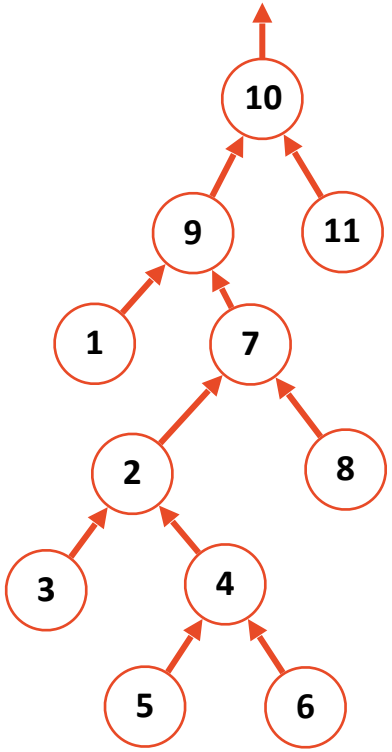


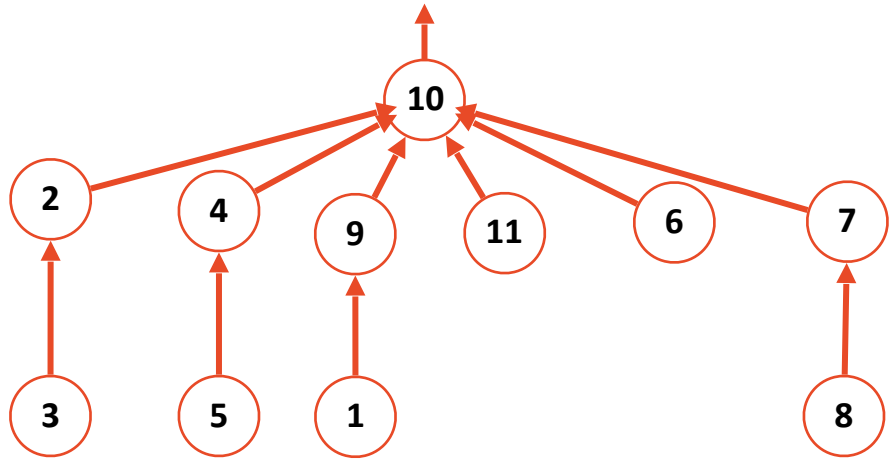
This is **path compression:**

```
1  int DisjointSets::find(int i) {
2    if ( s[i] < 0 ) { return i; }
3    else {
4      int root = find( s[i] );
5      s[i] = root;
6      return root;
7    }
8  }
```

Path Compression

`Find(6)`





This seems good — but how good in theory?

Path Compression Analysis

Two major problems here:

1) Our efficiency changes ***over repeated calls to find()***

2) Our height changes so we cant use union by height