Data Structures
Disjoint Sets 2

CS 225
Brad Solomon

October 18, 2024

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Learning Objectives

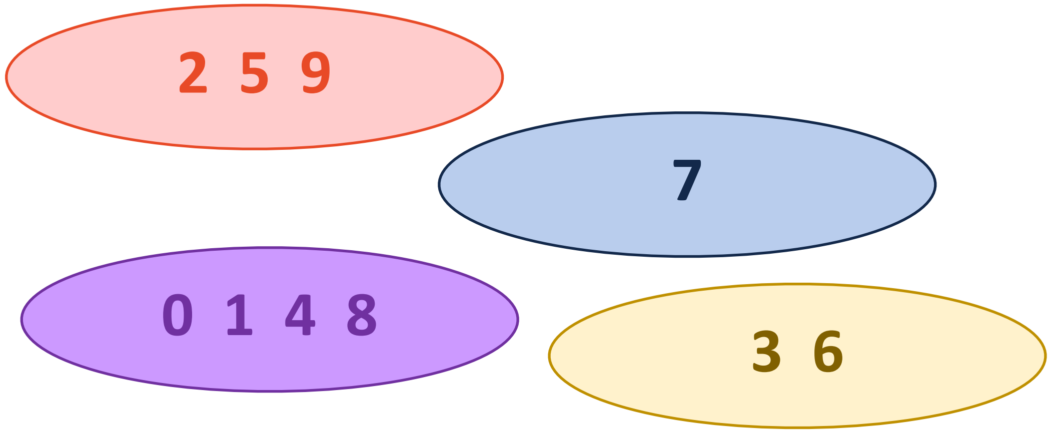
Continue to improve implementation of disjoint sets

Discuss how improvements affect efficiency

Disjoint Sets

ADT:

```
makeSet(vector<T> items)

Find(T key)

Union(T k1, T k2)
```



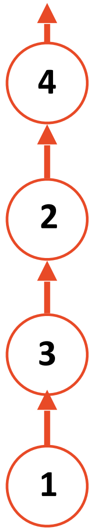
Key Ideas:

Every item exists in exactly one set

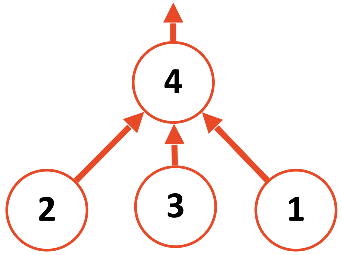
Every item in each set has same representation

Every set has a different representation

Disjoint Sets – Best and Worst UpTree



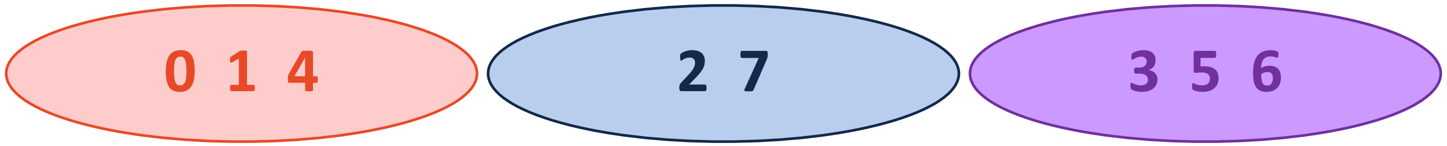
0	1	2	3	4
	3	4	2	-1



0	1	2	3	4
	4	4	4	-1

Disjoint Set Implementation

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



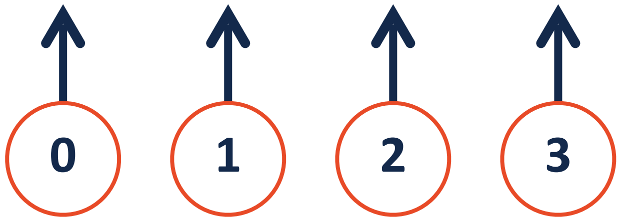
0	1	2	3	4	5	6	7
	0			0	3	3	2

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

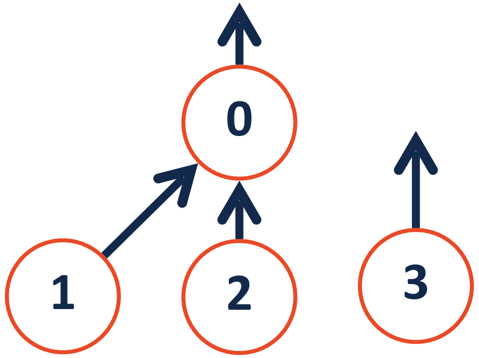
Union(k_1, k_2): Update ***smaller*** canonical item to point to larger

Update value of remaining canonical item

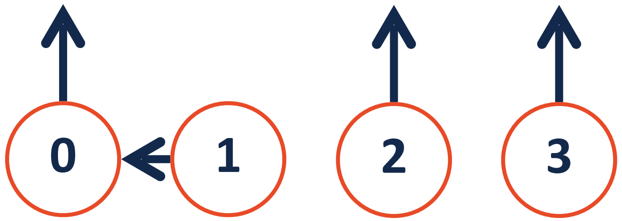
Disjoint Sets Union by Size



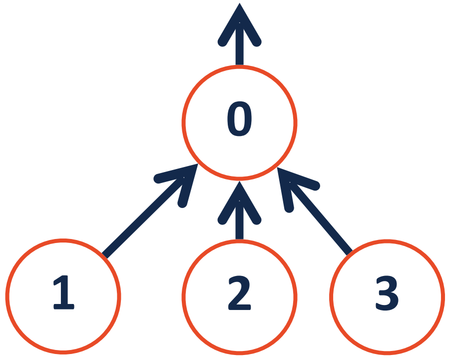
0	1	2	3
-1	-1	-1	-1



0	1	2	3

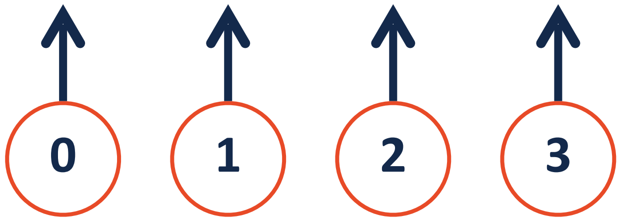


0	1	2	3

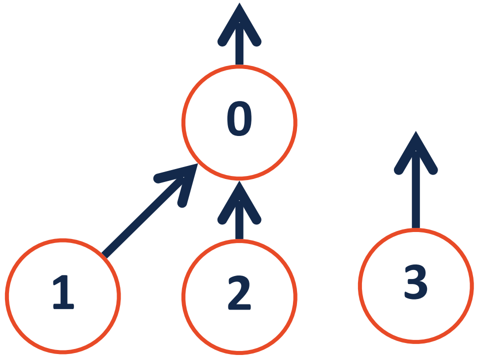


0	1	2	3

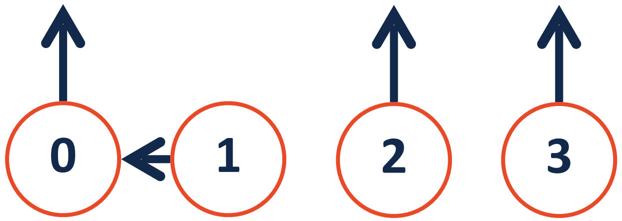
Disjoint Sets Union by Size



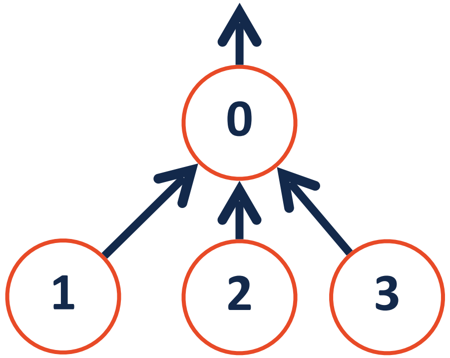
0	1	2	3
-1	-1	-1	-1



0	1	2	3
-3	0	0	-1

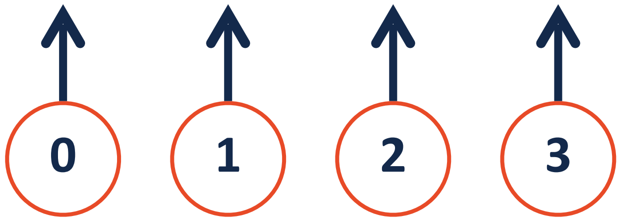


0	1	2	3
-2	0	-1	-1

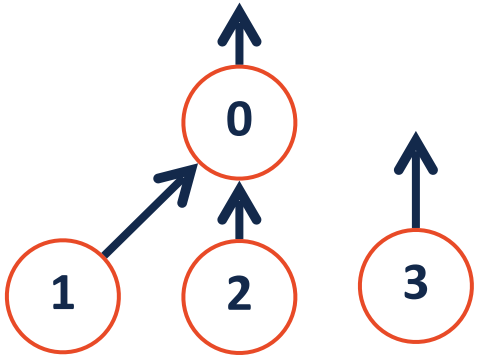


0	1	2	3
-4	0	0	0

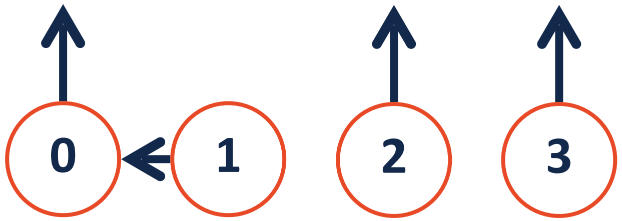
Disjoint Sets Union by Height



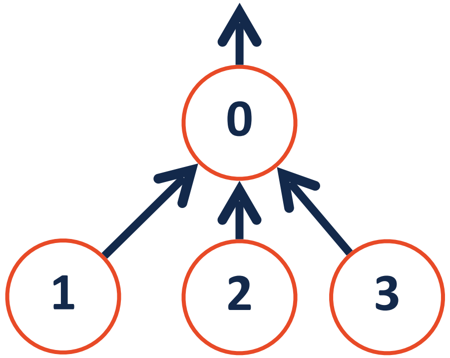
0	1	2	3
-1	-1	-1	-1



0	1	2	3

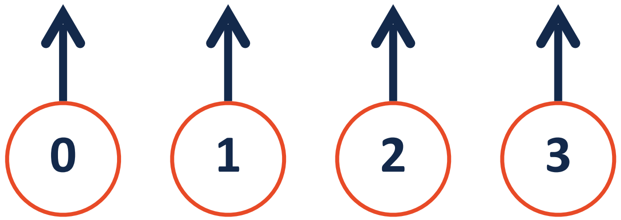


0	1	2	3

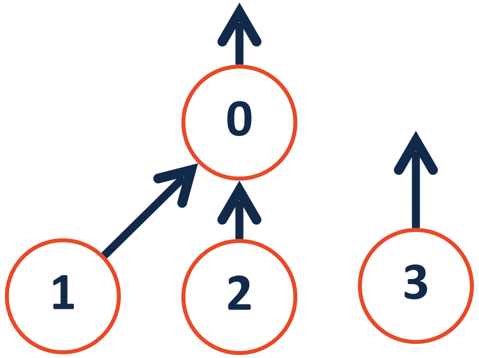


0	1	2	3

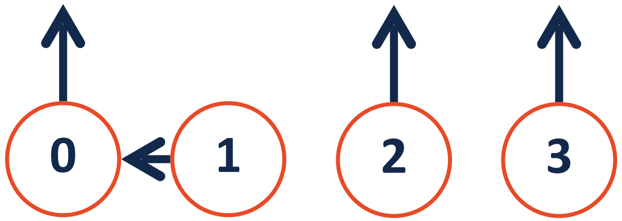
Disjoint Sets Union by Height



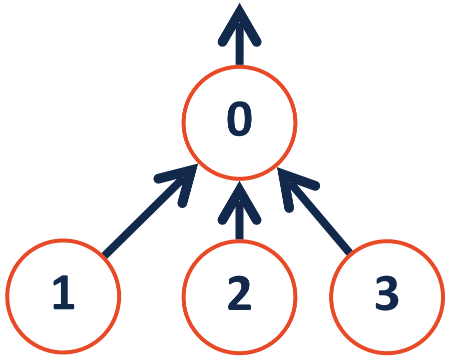
0	1	2	3
-1	-1	-1	-1



0	1	2	3
-2	0	0	-1



0	1	2	3
-2	0	-1	-1

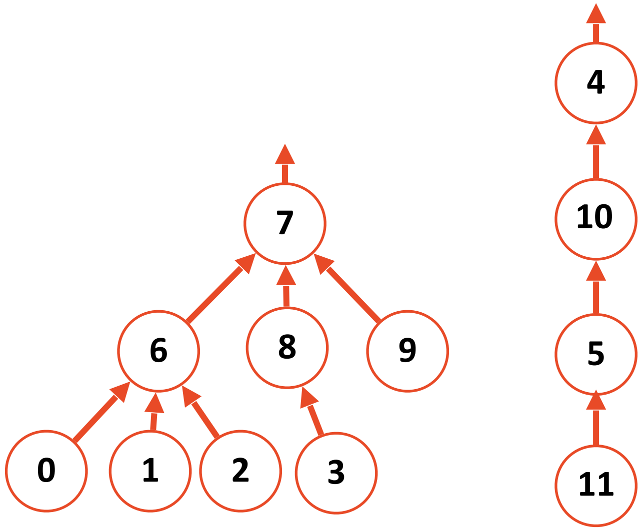


0	1	2	3
-2	0	0	0

Disjoint Sets – Smart Union

Two O(1) methods of combining two sets

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



Union by height

Before Union

4	…	7
-4		-3

After Union

4	…	7
-4		4

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

Union by size

Before Union

4	…	7
-8		-4

After Union

4	…	7
7		-12

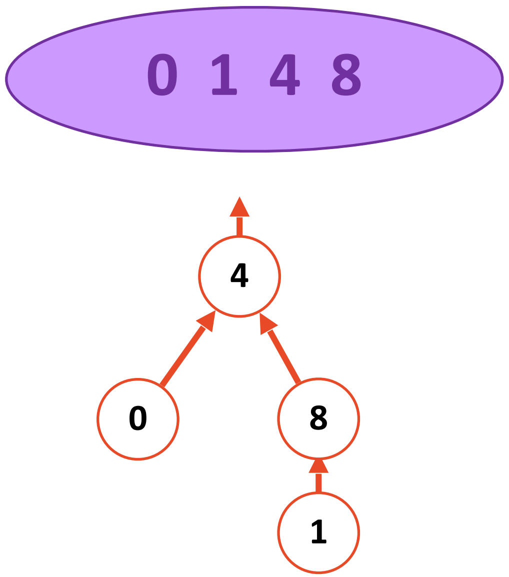
Idea: *Minimize the number of nodes that increase in height*

Disjoint Sets Find

Find(1)

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

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

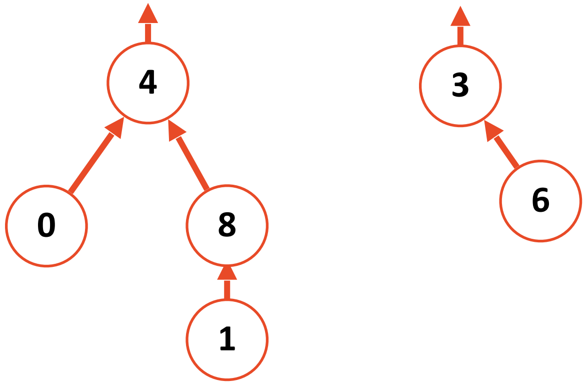


0	1	2	3	4	5	6	7	8	9
4	8			-3/-4				4	

Disjoint Sets Union

unionBySize(4, 3)

```
void DisjointSets::unionBySize(int root1, int root2) {
  int newSize = arr_[root1] + arr_[root2];

  if ( arr_[root1] < arr_[root2] ) {

    arr_[root2] = root1;

    arr_[root1] = newSize;

  } else {

    arr_[root1] = root2;

    arr_[root2] = newSize;

  }
}
```



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 $\geq$ ______________

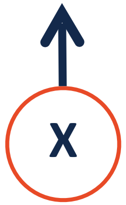
Base Case:

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)

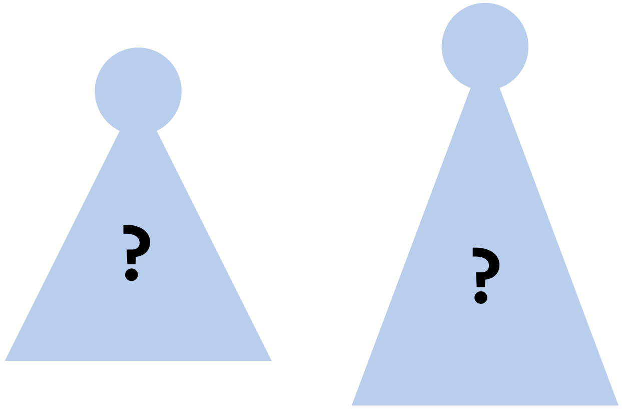
Without loss of generality, let B be the larger set **BY SIZE**

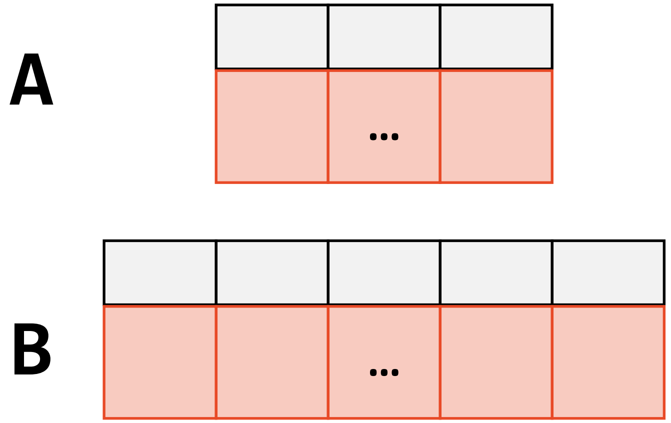
We must explore how height changes for each case:

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

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

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





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

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) $\geq$ 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)

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.

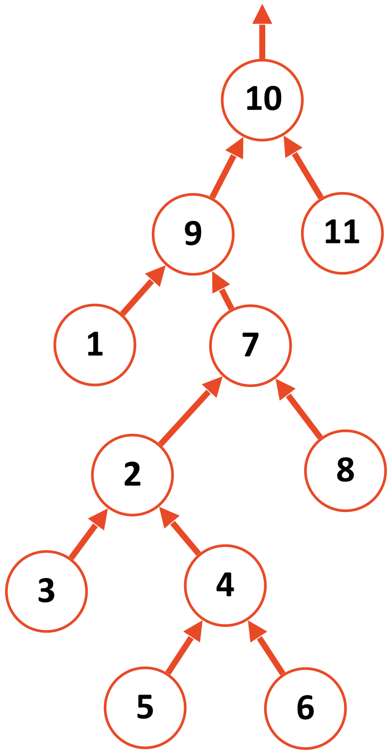
Each case we saw we have $n \geq 2^h$.

Disjoint Sets Find

Find(6)

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

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



Disjoint Sets Find

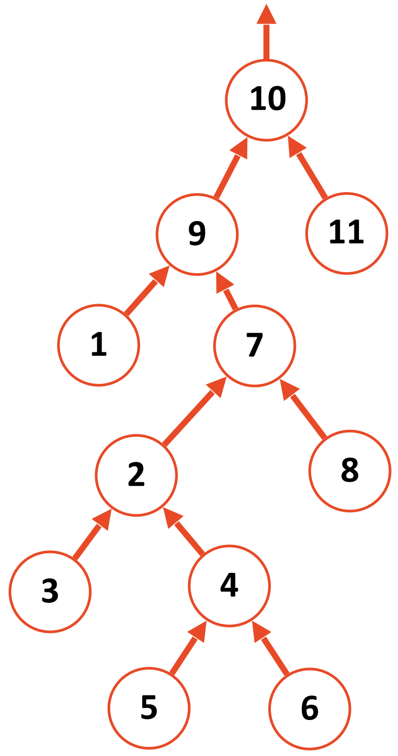
`Find(6)`

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

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

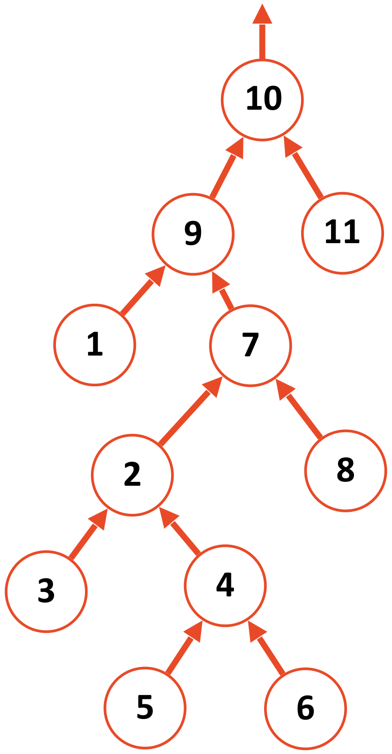
This is **path compression:**

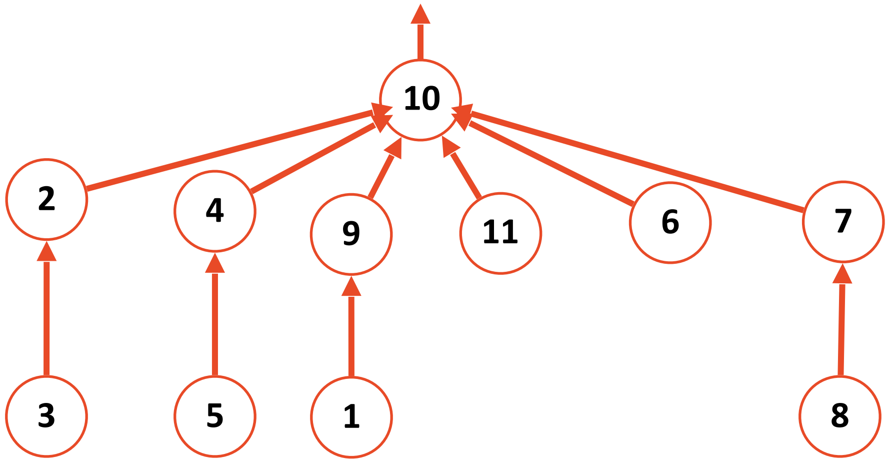
```
int DisjointSets::find(int i) {
  if ( s[i] < 0 ) { return i; }
  else {
    int root = find( s[i] );
    s[i] = root;
    return root;
  }
}
```



Path Compression

Find(6)





This seems good — but how good in theory?

Path Compression Analysis

Two major problems here:

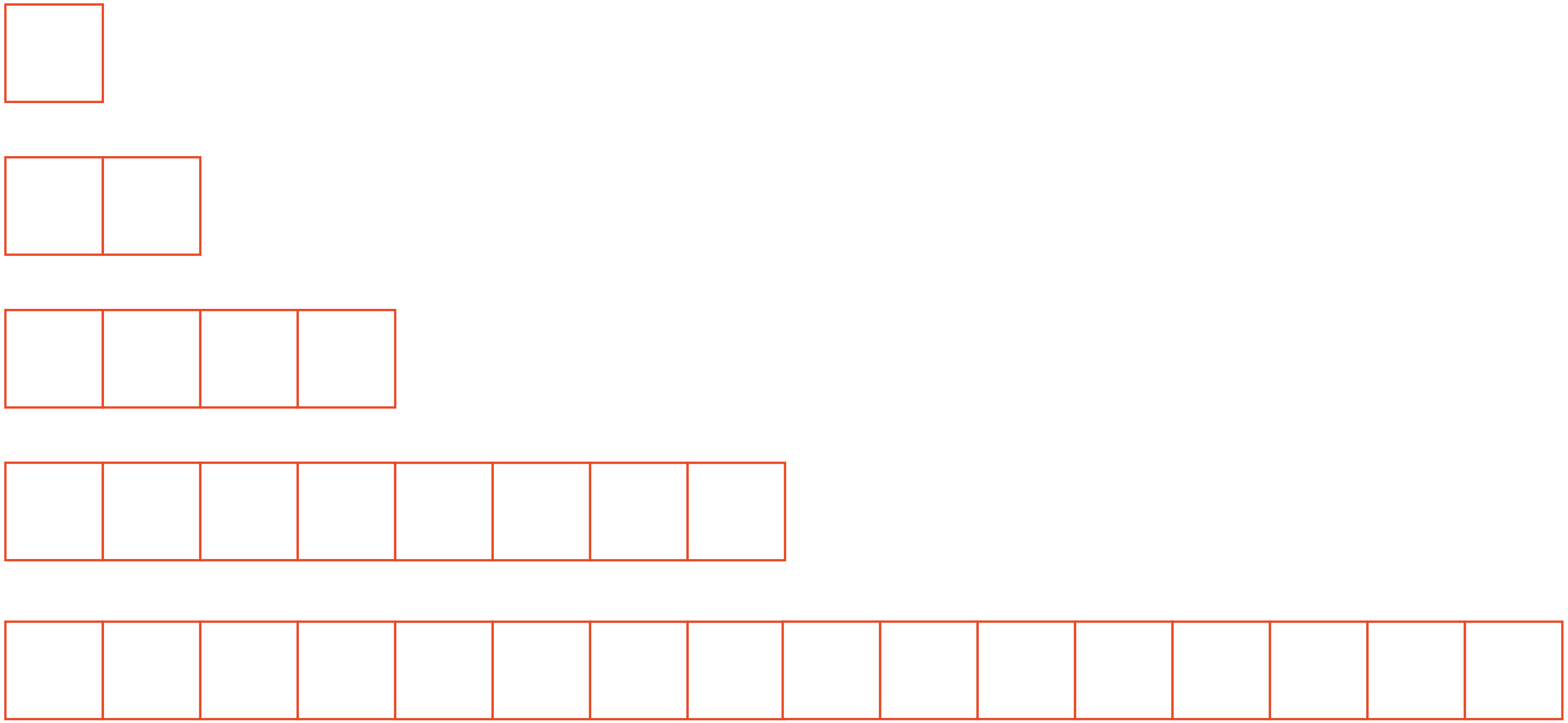
1) Our efficiency changes ***over repeated calls to find()***

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

Amortized Time Review

We have **n items**. We make **n insert()** calls.

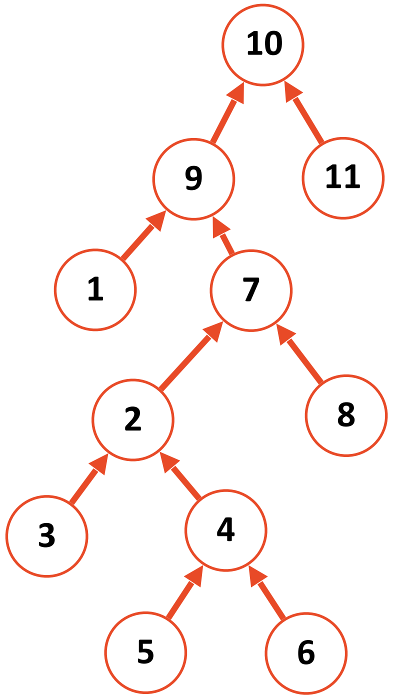
We are interested in the **worst case work** possible **over n calls**.



Amortized Time (Path Compression)

We have **n items** in an Uptree. We make **m find()** calls.

We are interested in the **worst case work** possible **over m calls**.



Union by Rank (Not Height)

Once I do path compression, I change the height of tree!

So we need a new way of approximating height.

Rank is a way of remembering what our height was before P.C.

Union by Rank (Not Height)

New UpTrees have rank = 0

Let A, B be two sets being unioned. If:

rank(A) == rank(B): The merged UpTree has rank + 1

rank(A) > rank(B): The merged UpTree has rank(A)

rank(B) > rank(A): The merged UpTree has rank(B)

Key Properties of UpTree by rank w/ PC

The parent of a node is always higher rank than the node.

There are at least $\geq 2^r$ nodes in a root of rank r.

For any integer r, there are at most $\frac{n}{2^r}$ nodes of rank r.

Key Properties of UpTree by rank w/ PC

The parent of a node is always higher rank than the node.

This comes from how we set up rank union

(Take larger of two rank or add one if tied)

There are at least $\geq 2^r$ nodes in a root of rank r.

Proof by Induction: To create rank r set, we merge two $r-1$ sets

By IH (not shown), those sets have $2^{r-1} + 2^{r-1} = 2^r$ nodes

For any integer r, there are at most $\frac{n}{2^r}$ nodes of rank r.

A rewrite of the above logic given n nodes

Amortized Time (Rank w/ Path Compression)

Put every non-root node in a bucket by rank!

Structure buckets to store ranks $[r,\ 2^r - 1]$

Where did number range come from?

Ranks	Bucket
0	0
1	1
2 - 3	2
4 - 15	3
16 – 65535	4
65536 – 2^{65536}-1	5

Iterated Logarithm Function ($log^* n$)

The number of times you can take a log of a number

$$log^*(n) = \begin{cases} 0 & , n \le 1 \\ 1 + log^*(log(n)) & , n > 1 \end{cases}$$

$$log^*(2^{65536}) = 5$$

$$2^{65536}$$
$$2^{16} = 65536$$
$$2^4 = 16$$
$$2^2 = 4$$
$$2^1 = 2$$
$$2^0 = 1$$

Amortized Time (Rank w/ Path Compression)

The work of **find(x)** are the steps taken on the path from a node x to the root (or immediate child of the root) of the UpTree containing x

We can split this into two cases:

Case 1: We take a step from one bucket to another bucket.

Case 2: We take a step from one item to another inside the same bucket.

Amortized Time (Rank w/ Path Compression)

The work of **find(x)** are the steps taken on the path from a node x to the root (or immediate child of the root) of the UpTree containing x

We can split this into two cases:

Case 1: We take a step from one bucket to another bucket.

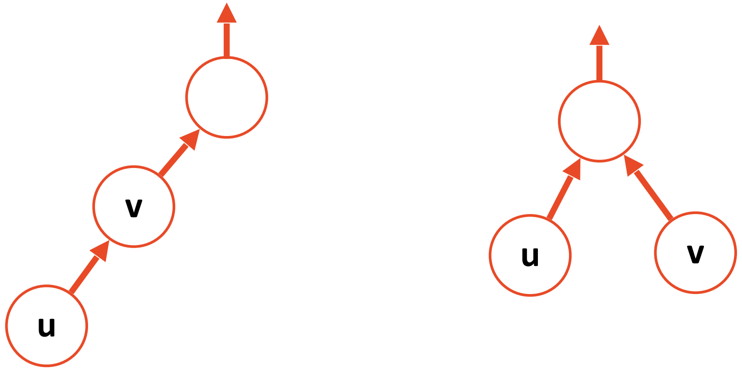
We have at most $log*(n)$ buckets so for **m** finds, this is $O(m\ log*n)$

Case 2: We take a step from one item to another inside the same bucket.

Let's call this the step from **u** to **v**.

Every time we do this, we do path compression:

We set parent(u) a little closer to root



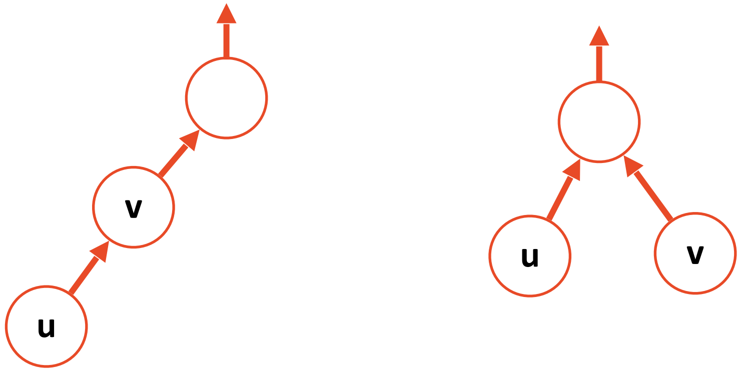
Amortized Time (Rank w/ Path Compression)

Case 2: We take a step from one item to another *inside* the same bucket.

Let's call this the step from **u** to **v**.

Every time we do this, we do path compression:

We set parent(u) a little closer to root



How many total times can I do this for each **u** in a bucket?

By definition of our bucket ranges $\sim 2^r$

How many nodes are in bucket **r**?

By definition of how we set up rank: $\frac{n}{2^r}$

Given we have **log*(n)** buckets:

Case 2 work is $n\ log*(n)$

Final Result

We have **n items** in an Uptree. We make **m find()** calls. Total work is:

Amortized $(n + m)\ log * (n)$

In terms of real world data, this is practically a constant.

Alternative Not-Actually-A-Proof

Unproven Claim: A disjoint set implemented with smart union and path compression with **m** find calls and **n** items has a worst case running time of **inverse Ackerman.** $\left[O(m\ \alpha(n))\right]$

This grows *very* slowly to the point of being treated a constant in CS.