Data Structures

Disjoint Sets 3

CS 225
Brad Solomon & G Carl Evans

October 20, 2023

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Learning Objectives

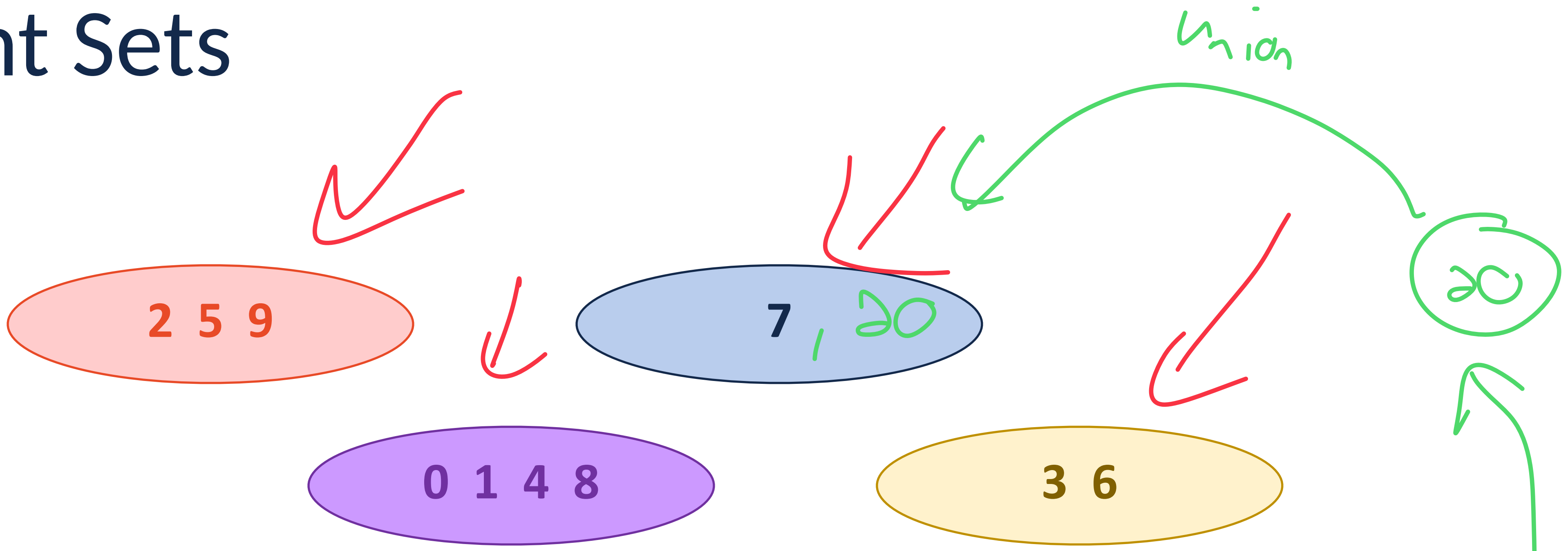
Discuss efficiency of disjoint sets

union by size
always efficient

Introduce path compression and rank

Prove efficiency of disjoint sets (again)

Disjoint Sets



Key Ideas:

- Each element exists in exactly one set.
- Every item in each set has the same representation
- Each set has a different representation

makeset(20)

Disjoint Sets Representation

We can represent a disjoint set as an array where the key is the index

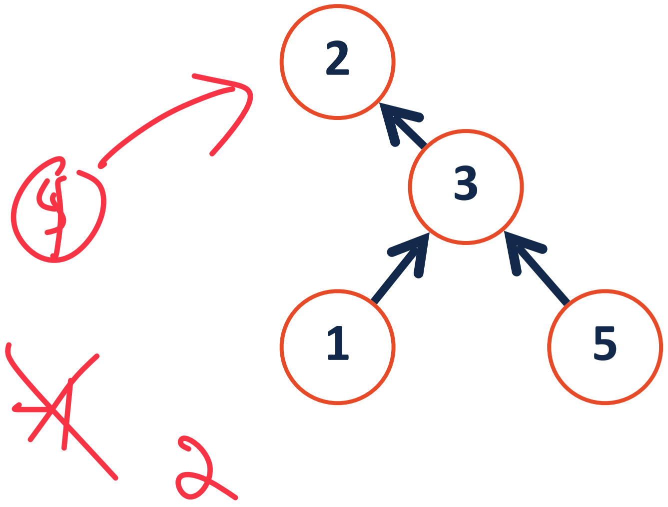
The values inside the array stores our sets as a pseudo-tree (UpTree)

-1

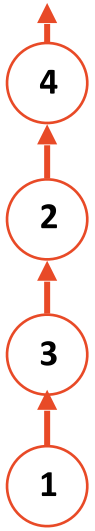
Negative values denote representative elements (the root)

All other set members store the index to a parent of the UpTree

Store height/size in the roots



Disjoint Sets – Best and Worst UpTree



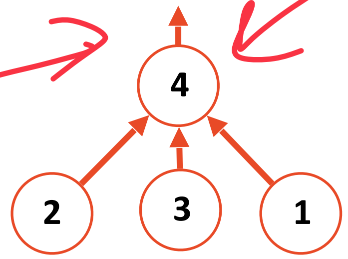
1

3

0	1	2	3	4
	3	4	2	-1

How do we design union?

Best



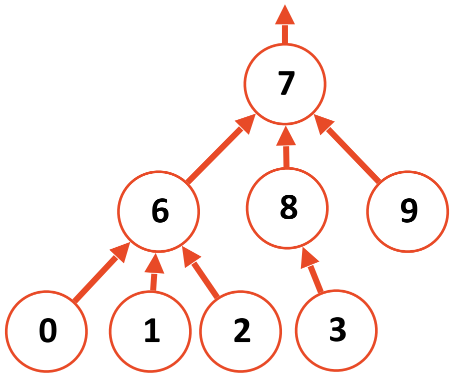
distance 1

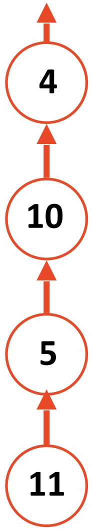
find() O(1)

0	1	2	3	4
	4	4	4	-1

Just b/c 0 is not in set

Disjoint Sets – Smart Union





- height - 1

Union by height

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-4	10	7	-3	7	7	4	5

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

Union by size

- size

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-4	10	7	-8	7	7	4	5

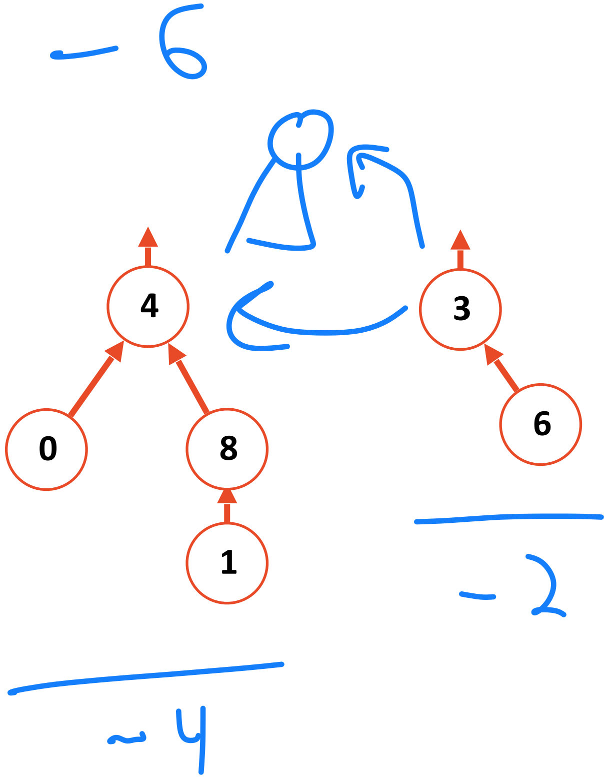
***Idea**: Minimize the number of nodes that increase in height*

Claim that both guarantee the height of the tree is: O(log n).

Disjoint Sets Union by Size

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;

  }
}
```



Only ~~be~~ called on two roots

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

Disjoint Sets Union by Size

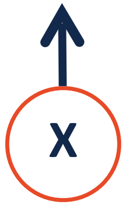
(uptree)

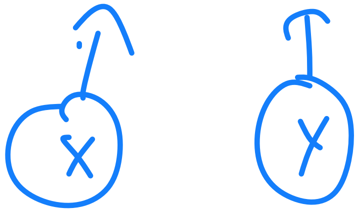
Claim: Sets unioned by size have a height of at most $O(\log_2 n)$

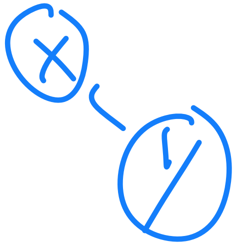
Claim: An UpTree of height **h** has nodes $\geq 2^h$

Base Case:

Base case height is 0, has one node.







$2 \geq 2^1$
nodes

vs.

$2^0 = 1$

Base case holds!

Disjoint Sets Union by Size

Let A, B be two sets being unioned

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

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

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

A B

A == B

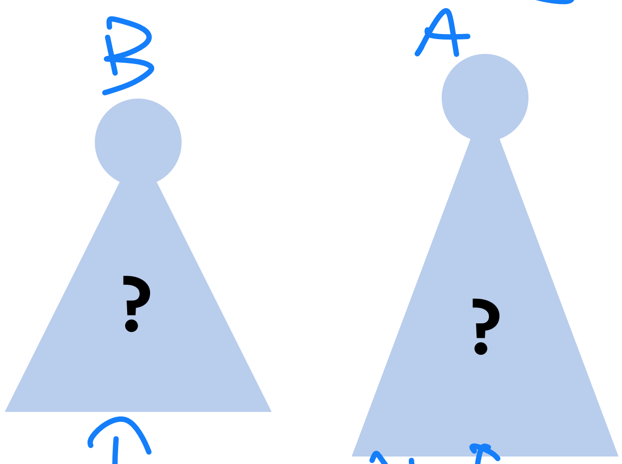
Bigger than A

n(B) $\geq$ n(A)

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

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

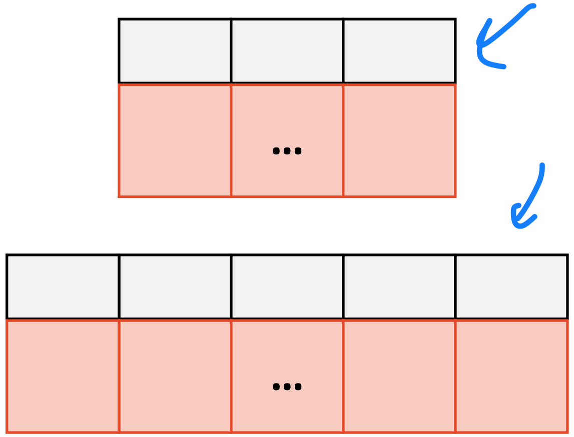
Case 3: h(A) > h(B)



T_A or B, T

A

B



Disjoint Sets Union by Size

$2^{h(B)}$

$n(B) \geq n(A)$

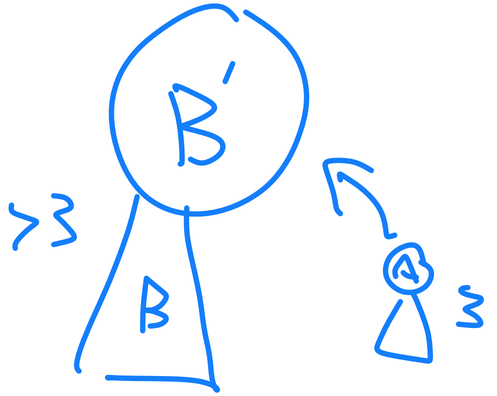
Claim: An UpTree of height **h** has nodes $\geq 2^{h}$

$h(B)$

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

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

↳ Ideal case! B/c B is our root



1) My height doesn't change!

2) By IH, both A & B are good disjoint sets

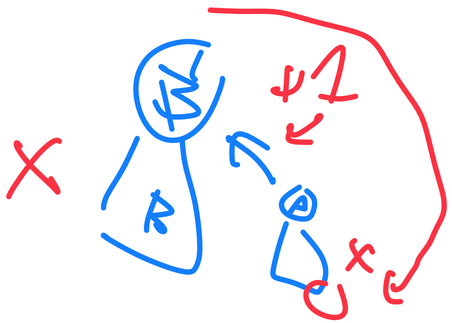
$n(B) \geq 2^{h(B)}$ and $n(A) \geq 2^{h(A)}$

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

$n(B') = n(B) + n(A)$

Disjoint Sets Union by Size

$n(B) \geq n(A)$



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

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

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

1) My height $[h(B') = 1 + \text{height}(B)]$

By IH: $n(B') = n(B) + n(A)$

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

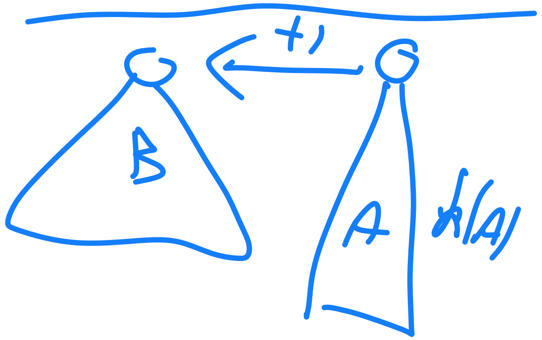
$2^{h(B)} + 2^{h(B)}$

$2 \cdot 2^{h(B)} =$

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

Disjoint Sets Union by Size

$n(B) \geq n(A)$



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

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

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

know $n(A) \geq 2^{h(A)}$

$n(B) \geq$

$n(B) = n(A) + n(B) \geq 2n(A)$

$2 \cdot 2^{h(A)}$

$2^{h(A)+1}$

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

Disjoint Sets Union by Size

$n(B) \geq n(A)$

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 $n \geq 2^h$.

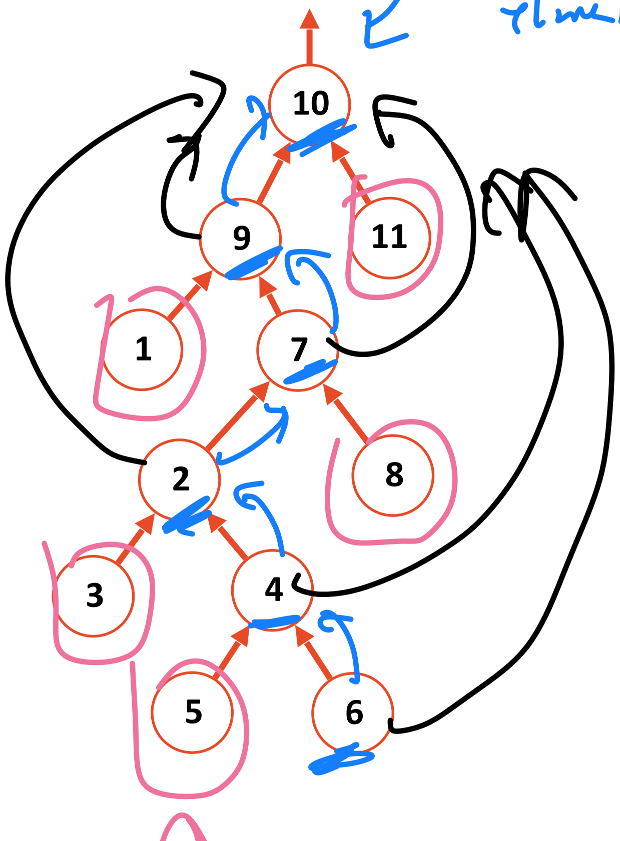
$h \approx O(\log n)$

An uptree unioned by size still has good height

Path Compression

Find(6)
↳ 10 = 10

canonical element



Not touched!

find (6) // 10
↳ find(4) // 10
↳ .... find(10)

find(2) // 10

return 10
return 10 . . .
returns 10

when we recurse up, on the way back set all nodes to point to root directly

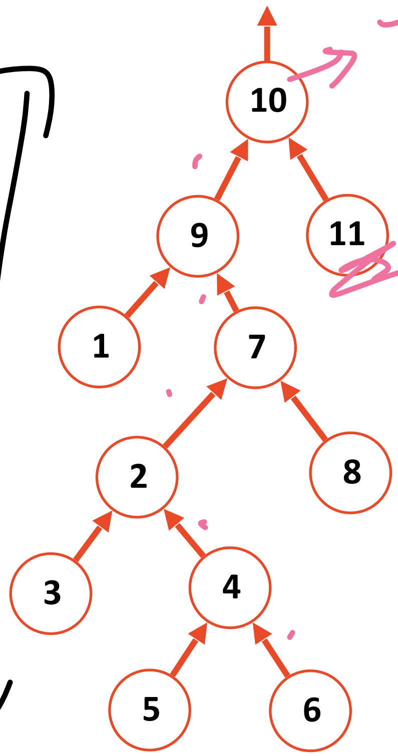
Path Compression

Find(6)

Assume log(n)

The more find() the closer we get

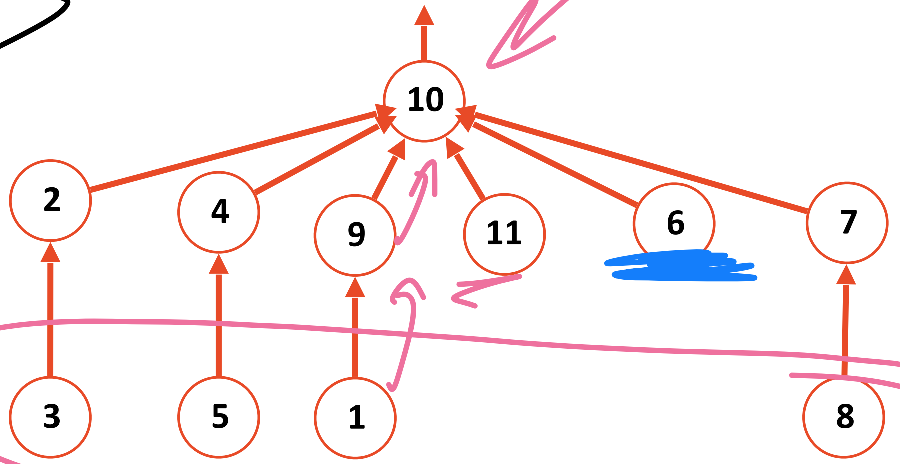
~O(1)



O(h)

-6

O(1) · h times



-6

No more work than find() was done!

find(1)

Intuitively: I only do work once

Nothing else changes

O(h)

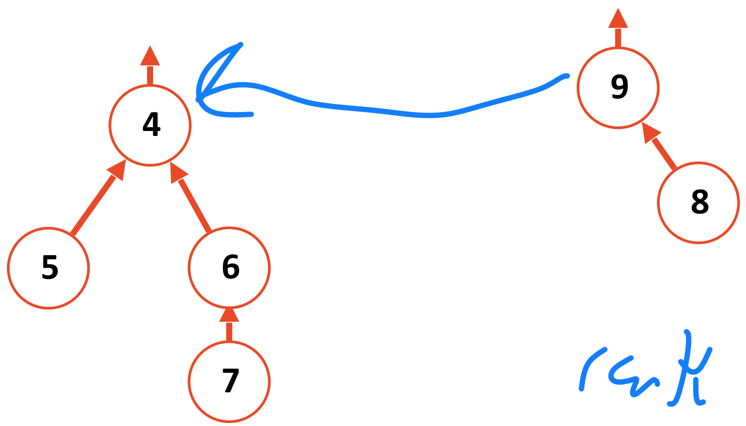
Disjoint Sets – Union by Rank (not height!)

union is max(A, B)+1

rank = 1

rank : 0 0 0 0

[zero]



rank 2

rank 3

Height + Path compression == lies!

-3

4

X

We will also store as - rank

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

Union by Rank (Not Height)

motivated by Path compression

The change: New UpTrees have rank = 0

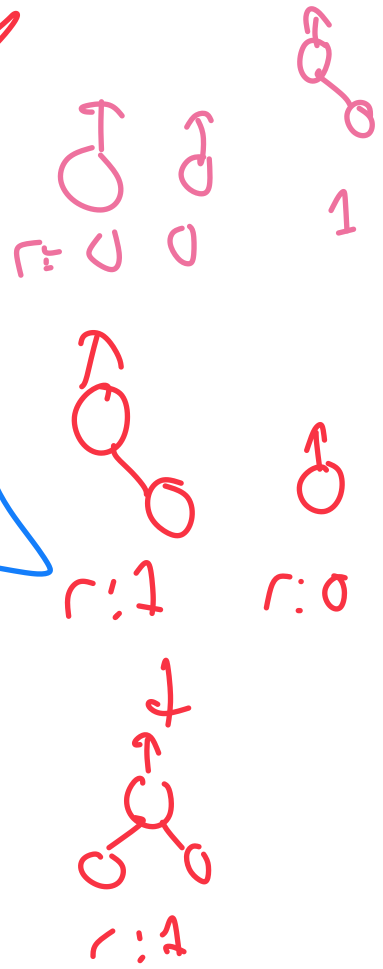
Let A, B be two sets being unioned. If:

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)

This is identical to height (with a different starting base)!



Union by Rank

Claim: An UpTree of rank **r** has nodes $\geq 2^r$.

Base Case:

Inductive Step: IH holds for all UpTrees up to $k < r$

Exercise for viewers

Try solving yourself before seeing answer (next slide)!

Union by Rank - Proof

Much like before we will show that in a tree with a root of rank r there are $nodes(r) \geq\ 2^r$

Base Case: UpTree of rank = 0 has 1 node $2^0 = 1$

Inductive Hypothesis: for all trees of ranks $k,\ k < r,\ \ nodes(k) \geq 2^k$

A root of rank r is created by merging two trees of rank $r - 1$

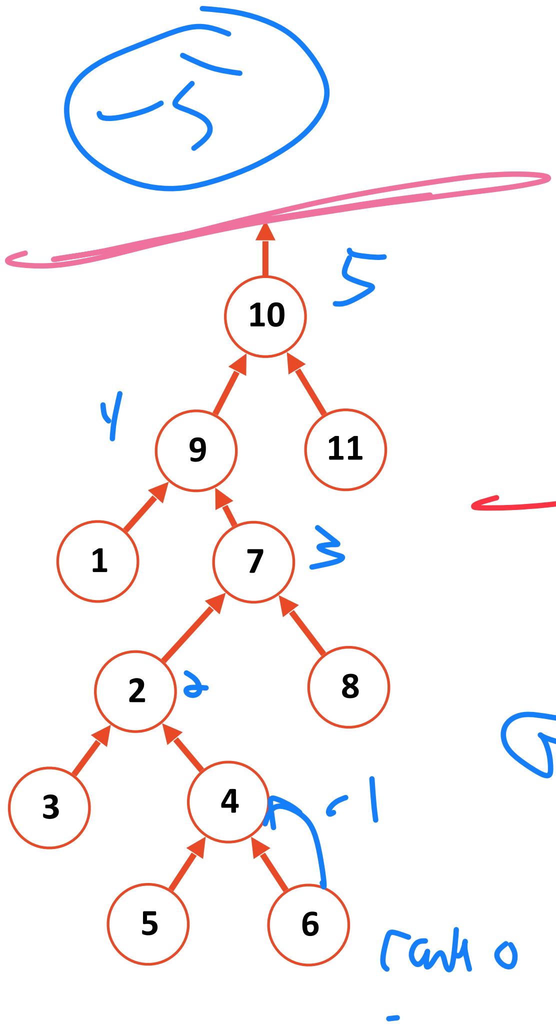
by IH each of those trees have $nodes(r-1) \geq 2^{r-1}$

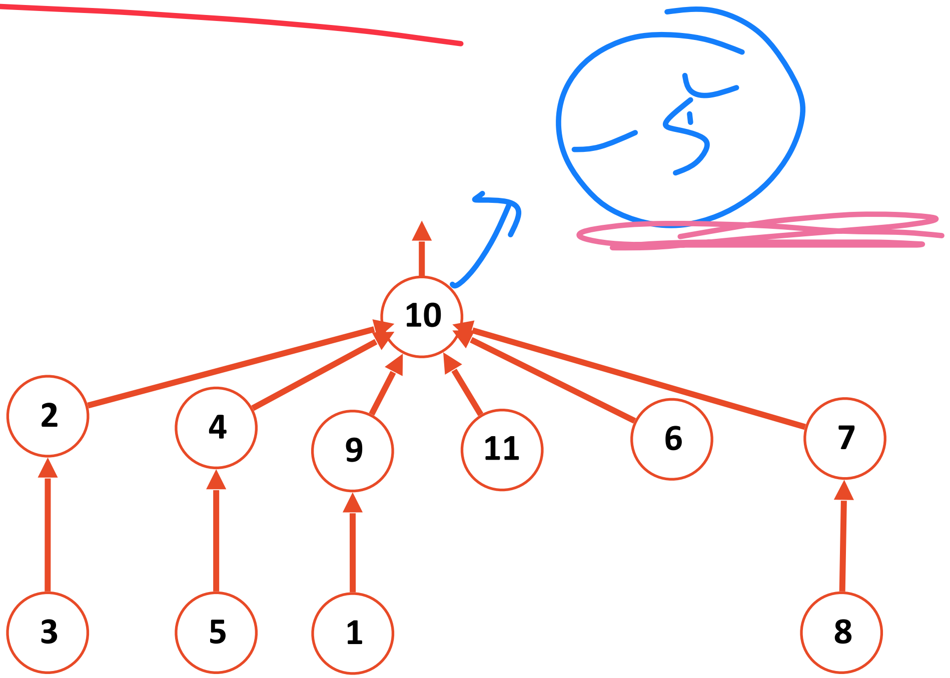
so, tree a of rank r has $nodes(r) \geq 2 \times 2^{r-1} \geq 2^r$

Taking the inverse, we get a height of $O(\log(n))$

Union by Rank w/ Path Compression

How does rank w/ path compression affect our runtime?





Find(i) → O(log n) worst

Union by Rank w/ Path Compression

1. Rank only changes for roots and can only increase (unlike height!)

↳ canonical elements

$r == r \Rightarrow r+1$

This is why union is O(1)

2. For all non-root nodes x, **rank(x) < rank(parent(x))**

3. If parent(x) changes, then our new parent has larger rank.

$r-1, r-1 \Rightarrow r$ $\quad r_1 < r \Rightarrow r$ $\quad r, r \Rightarrow r+1$

Union by Rank w/ Path Compression

4. min(nodes) in a set with a root of rank r has $\geq 2^r$ nodes.

↳ proof I skipped

$n \geq 2^r$

restating 4

5. Since there are only n nodes the highest possible rank is $\lfloor log\ n \rfloor$.

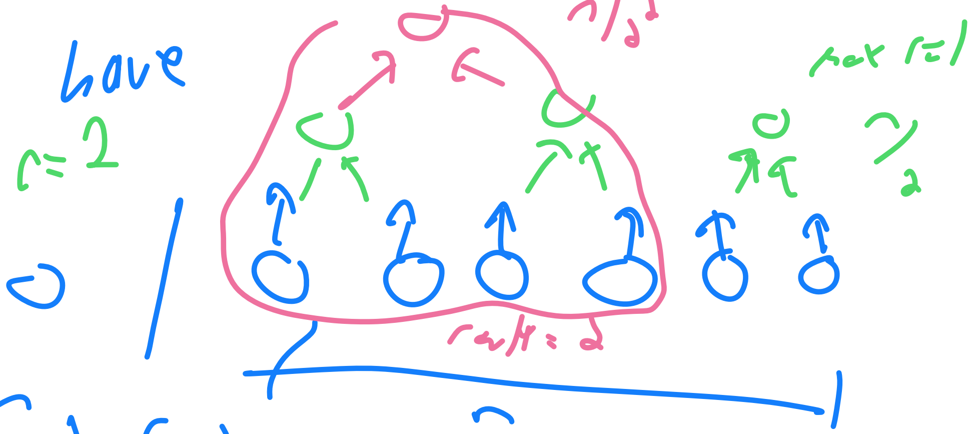
$h \sim O(\log n)$

Union by Rank w/ Path Compression

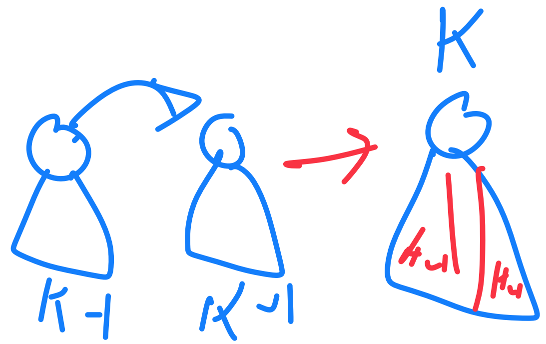
6. For any integer r, there are at most $\dfrac{n}{2^r}$ nodes of rank r.

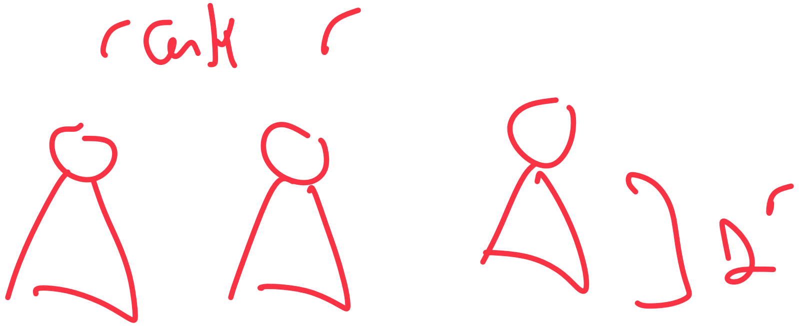
Intuitively: The larger my rank, the more nodes stored below me, the fewer total nodes I can have

Base case: $r = 0 \rightarrow n$ nodes of rank 0



For any k, $r = k$ only when we union $(k-1) \times (k-1)$





$n/2^r$ is the max # I can have of rank r

Amortized Time (Rank w/ Path Compression)

will be different

For **n** calls to makeSets() [**n items**] and **m** find() calls the max work is…

?? amount of union

Built best uptree

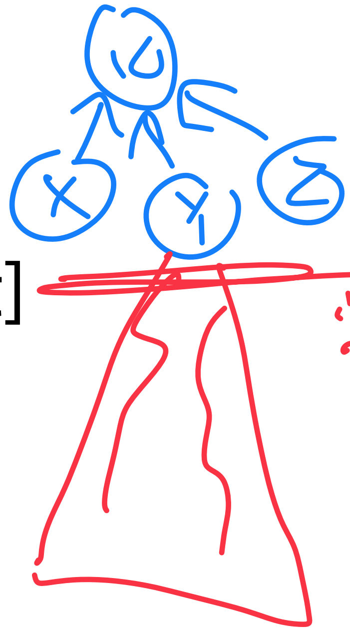
This gives us a more accurate picture since each find can make our search a faster!

Two cases of find():

canonical element

1. We search for root [or a node whose parent is root]

$O(1)$



Path compression

2. We search for a node where neither above apply.

Proof comes in

Do this Monday

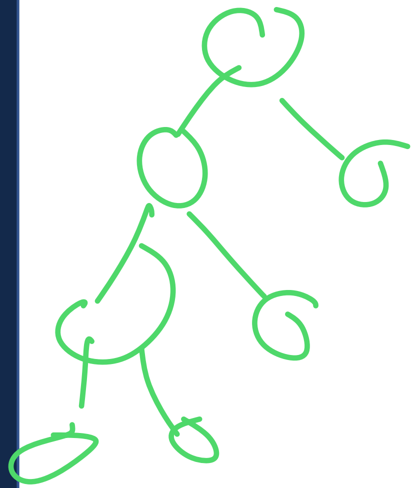
Amortized Time (Rank w/ Path Compression)

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

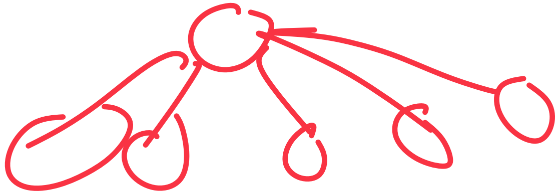
Ranks	Bucket
0	0
1	1
2 - 3	2
4 - 15	3
16 – 65535	4
65536 – 2^{65536}-1	5

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

Max work:

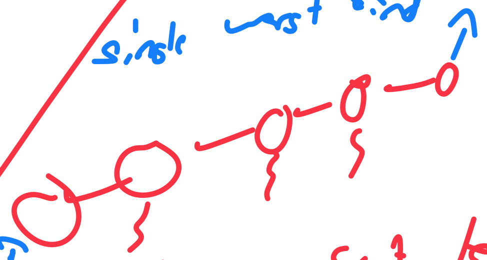


After n steps have explored every path



I'm missing, I have ~~bad~~

single worst find



find()

b/c we can't visit have direct LL

b/c $n \geq 2^h$

Iterated Logarithm Function (log^*n)

log^*n is piecewise defined as

0 if $n \leq 1$

otherwise

$1 + log^*(\log n)$

Amortized Time (Rank w/ Path Compression)

Let $|B_r|$ be the size of the bucket with min rank r.

What is $max(|B_r|)$?

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

Amortized Time (Rank w/ Path Compression)

The work of **find(x)** is 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.

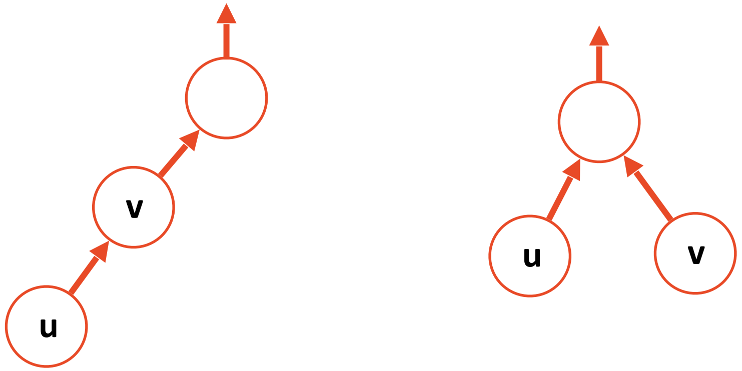
Amortized Time (Rank w/ Path Compression)

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 $|B_r|$?

How many nodes are in $|B_r|$?

Final Result

For **n** calls to makeSets() [**n items**] and **m** find() calls the max work is:

Even Better

In case that still seems too slow tightest bound is actually

$$\Theta(m\ \alpha(m, n))$$

Where $\alpha(m, n)$ is the inverse Ackermann function which grows much slower than log*n.

Proof well outside this class.

Randomized Algorithms

A **randomized algorithm** is one which uses a source of randomness somewhere in its implementation.

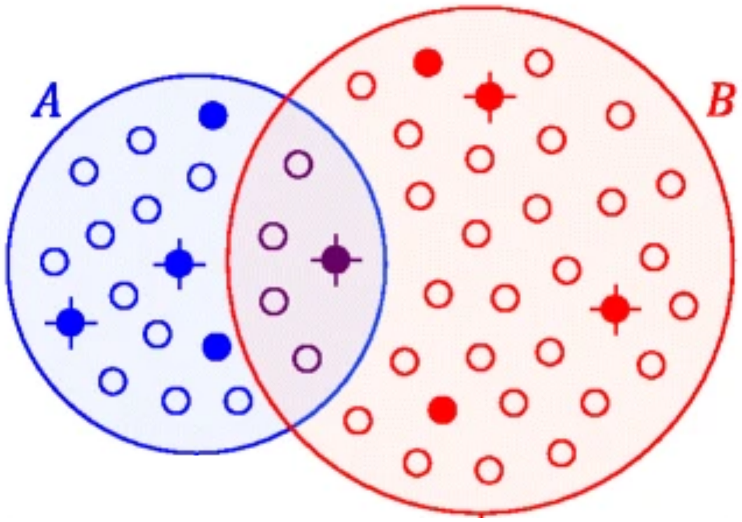
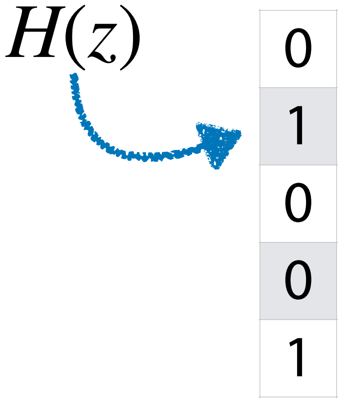
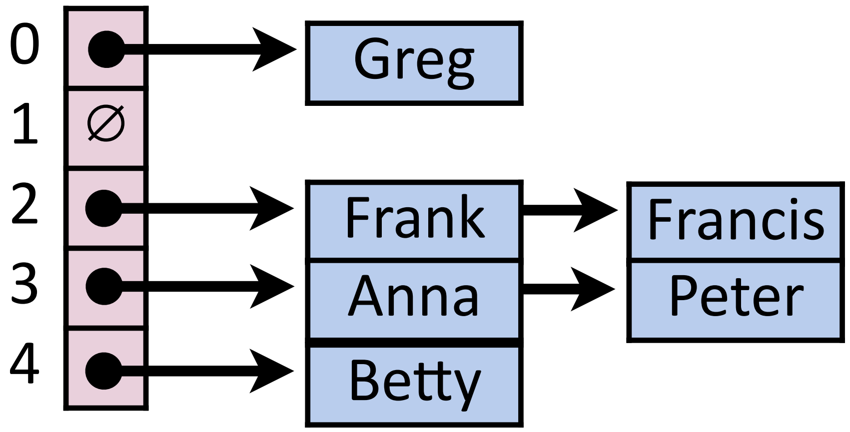


Figure from Ondov et al 2016





$H(x)$	0	2	1	0	0	4	0	2	0	6
$H(y)$	1	0	2	3	1	0	3	4	0	1
$H(z)$	2	1	0	2	0	1	0	0	7	2