Advanced Data Structures

Union-Find Analysis

Today

- Questions on setup from last time?
- Schedule has been updated - Let me know if you see any issues!
- Today: U-F analysis

Formally: 3 operations:

makeSet(x): take an item & create a one element set for it

find(x): return "canonical" element of set containing x

union(x,y): Assuming that $x \neq y$, form a new set that is the union of the 2 sets holding x & y, destroying the 2 old sets. (Also selects & returns a canonical element for new set)

How to implement?

- certainly use existing DS.

Use a table:

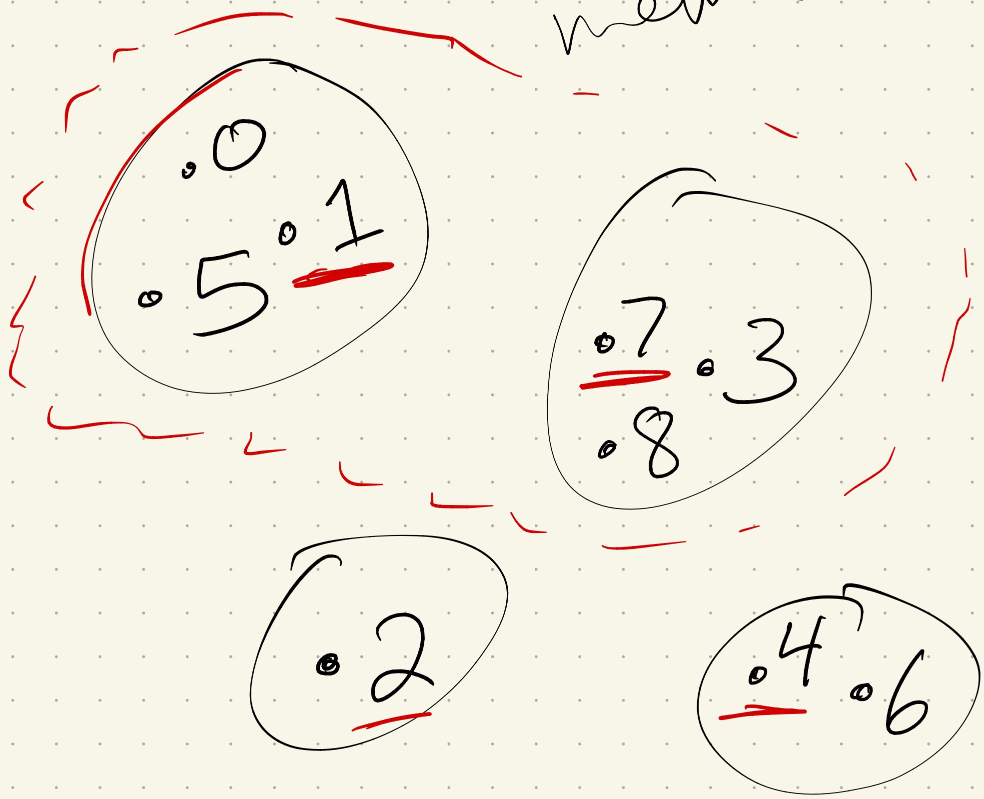
For each entry, record its set label!

Ex:

Table

	representative
0	1
1	1
2	2
3	~~7~~ 1
4	4
5	1
6	4
7	~~7~~ 1
8	~~7~~ 1

Sets & members



find → table lookup

Then: union(5, 8):

2 finds:

find(5)

find(8)

loop to reset all of one type

Runtime?	Why?
makeset : $O(1)$	create 1 new entry
find : $O(1)$	lookup one entry
union : $O(n)$	linear loop

So tradeoff w/ this approach:

Bad if many unions.

Better: Use trees!

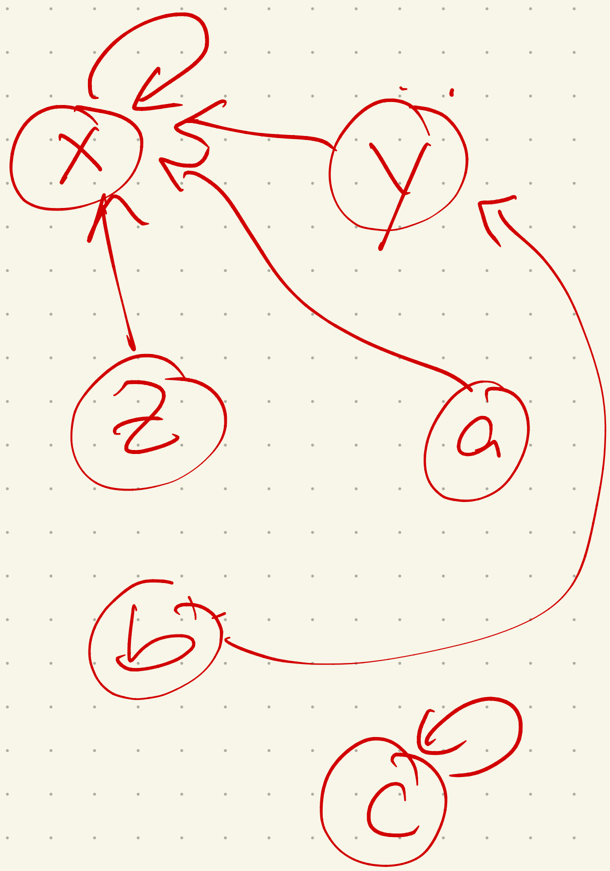
(Galler + Fisher, 1964)

Each set will be a rooted tree, where elements are in the tree & the root is the canonical element.

So each element has a pointer to its parent (& root points to itself)

Ex:

makeset(x)
makeset(y)
makeset(z)
union(x,z)
makeset(a)
makeset(b)
union(a, x)
union(b, y)
makeset(c)
union(z, b)



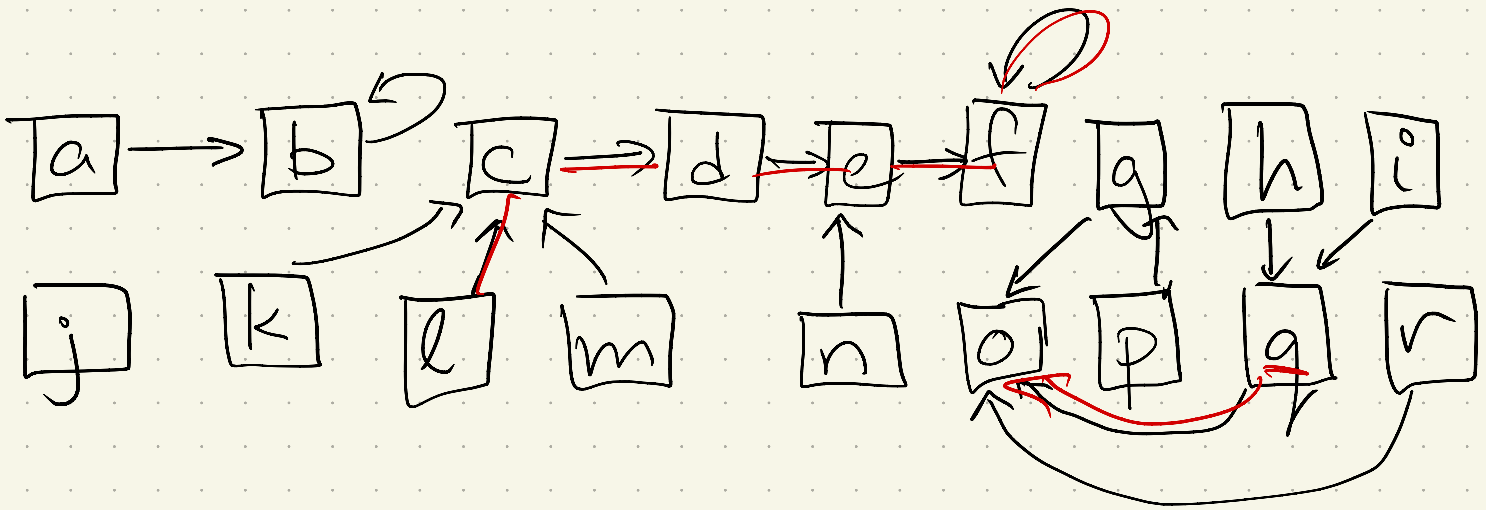
Then: <u>makeset(x)</u>:

create a node w/ value x,
+ points its pointer to
itself

<u>find(x)</u>:
travel up the the parent
pointer of x, until it
points to itself

<u>union(x, y)</u>:
combine 2 trees into
a single tree by
making one of the roots
a child of the other
root

Larger example



18 elements, 4 sets

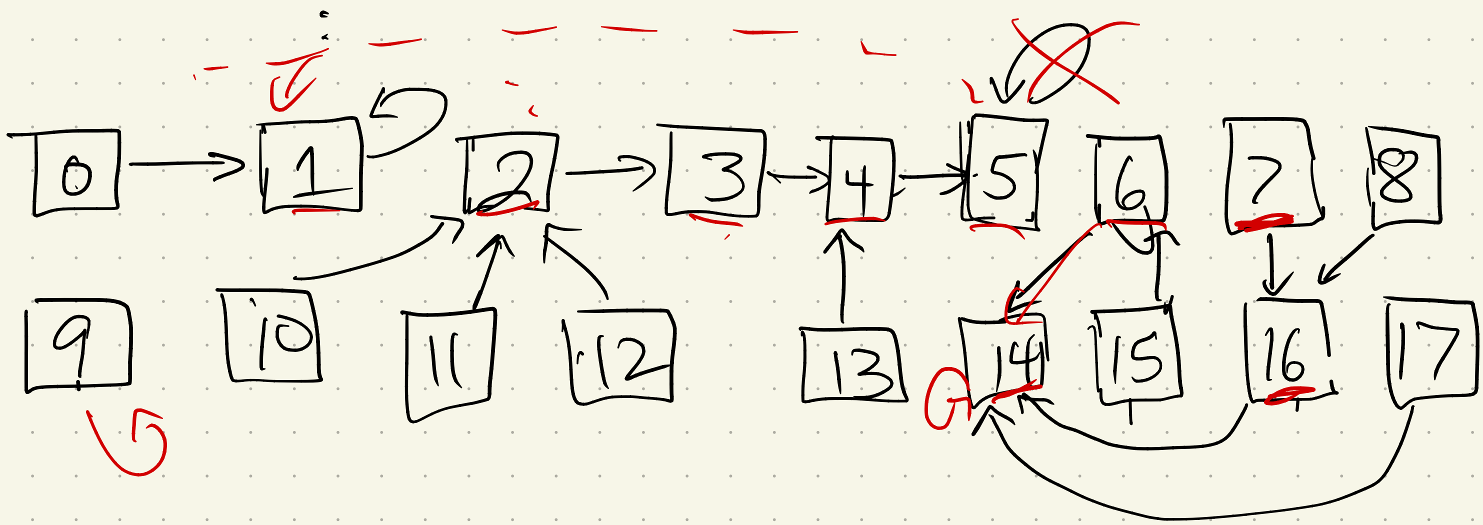
find(q) = o

find(l) = f

Implementation:

Don't actually need ptrs! Keep a "parent" array:

index	p[]	v[] value
0	1	a
1	1	b
2	3	c
3	4	d
4	5	e
5	~~5~~ ← new root	f
6	14	g
7	16	h
8	16	i
9	9	j
10	2	k
11	2	l
⋮		



Implementation:

$O(n)$

Find(x):

(while not root)

```
while (p[x] != x)
    x = p[x]
return x
```

Union(x, y):

$$\bar{x} = \text{find}(x)$$

$$\bar{y} = \text{find}(y)$$

$$\text{if}(\bar{x} \;!= \bar{y})$$

$$p[\bar{x}] = y$$

$O(1)$

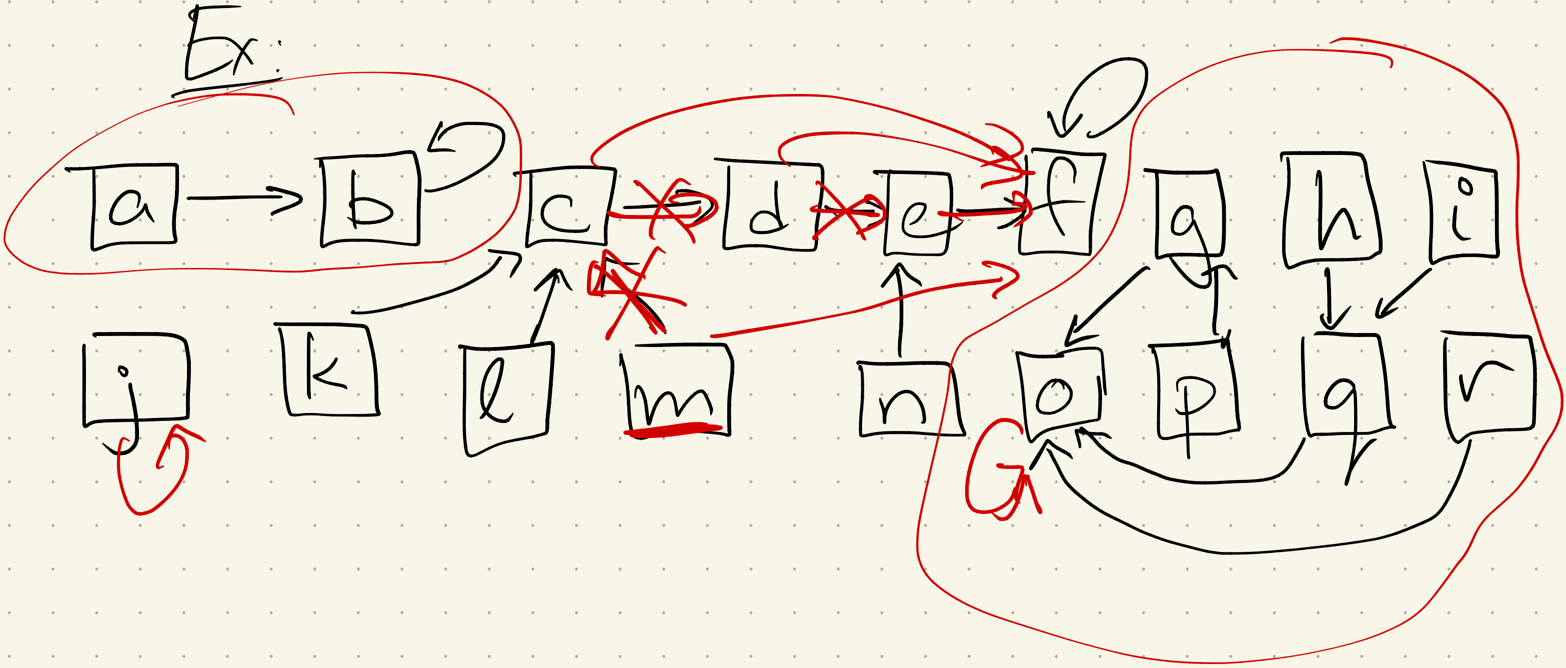
Still some flexibility:

1. Need to decide which root becomes the root of new set: ie: union(a,h)

union by rank

2. Can also use "path compression": try to point as many things to the root as possible, so later queries get faster. ie: find(m)

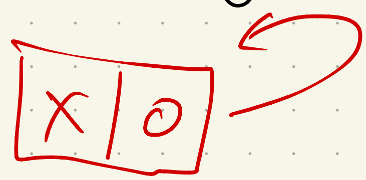
Ex:



1 Union by rank:

(introduced several places)

Each time you union, make smaller tree tree's root the child of larger one.



How?

- Each node will have a "rank" field, initialized to 0.
- In a union:

  If one's rank is smaller:

  smaller "points" to larger

  If both are equal:

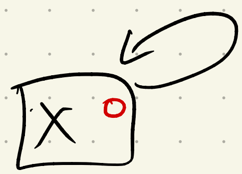
  point one to other, + increment new root's rank

Rank only changes for a root.

(Once a node is not a root, it can never become one again)

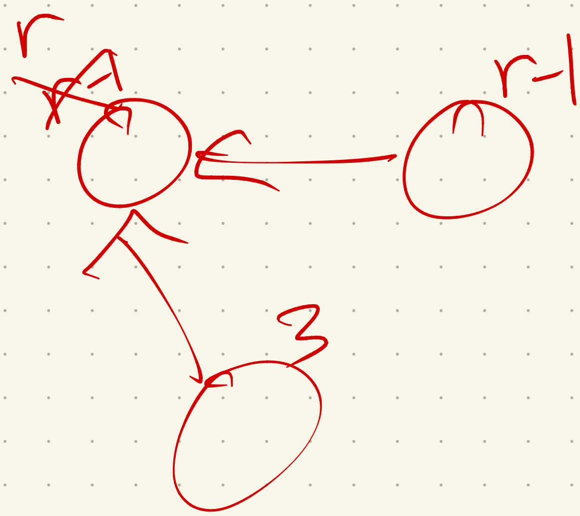
Lemma: $\mathrm{rank}(x) \le \mathrm{rank}(\mathrm{parent}(x))$
(with equality only if $\mathrm{parent}(x) = x$)

pf: induction!
True for a tree of rank 1:



if rank $r > 1$:

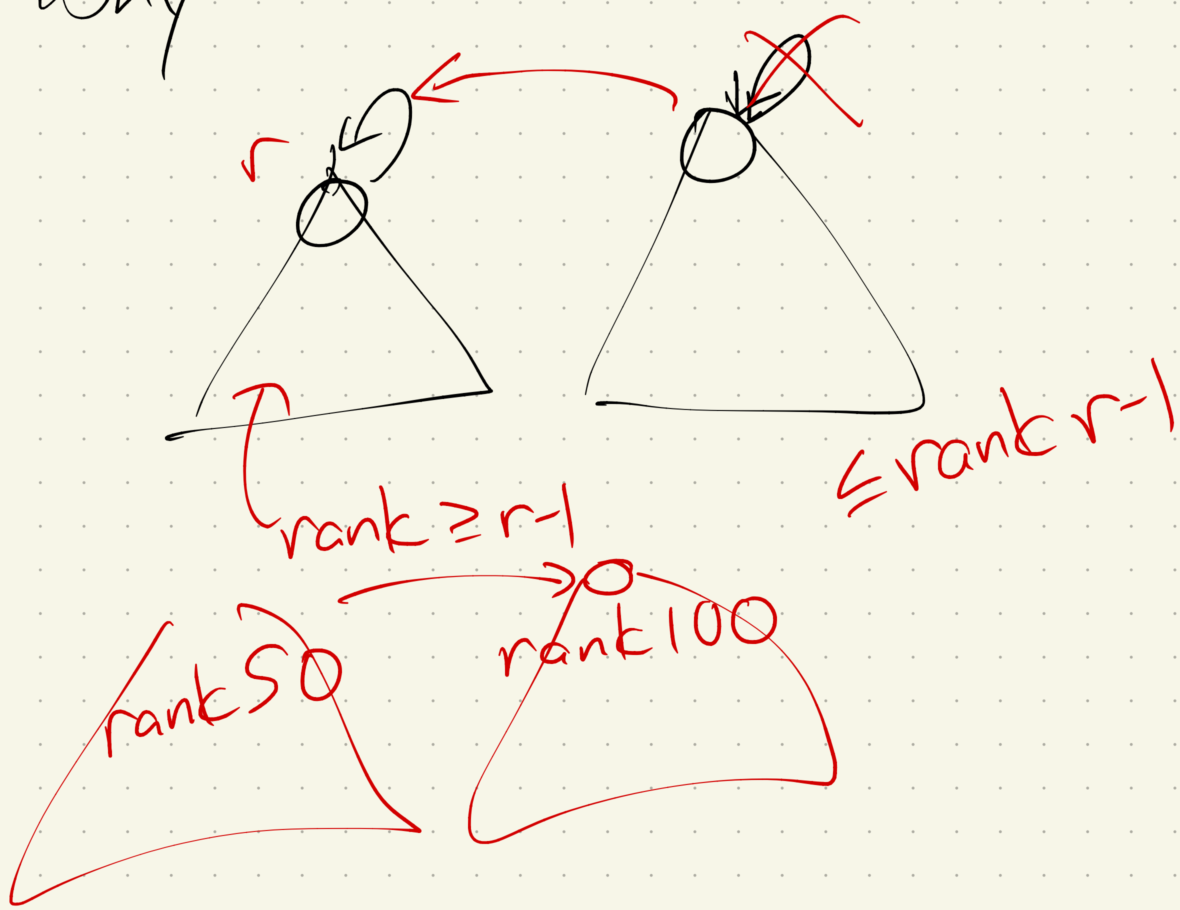
Consider when incremented from $r-1$ to r



Thm: height of one of these trees is $O(\log n)$

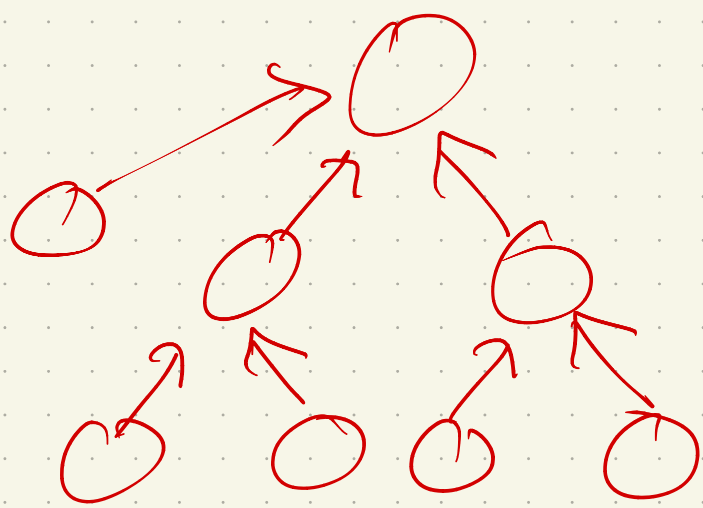
pf: Every time a node's leader has changed, the set is at least twice as big.

Why?



So: if n items in set, how many times could it have doubled?

$\log_2 n$



(Note: there are examples which are $\Omega(\log n)$ in height.)

Result: Runtimes are:

- makeset:

$O(1)$

- find: travels a path to root

$O(\log_2 n)$

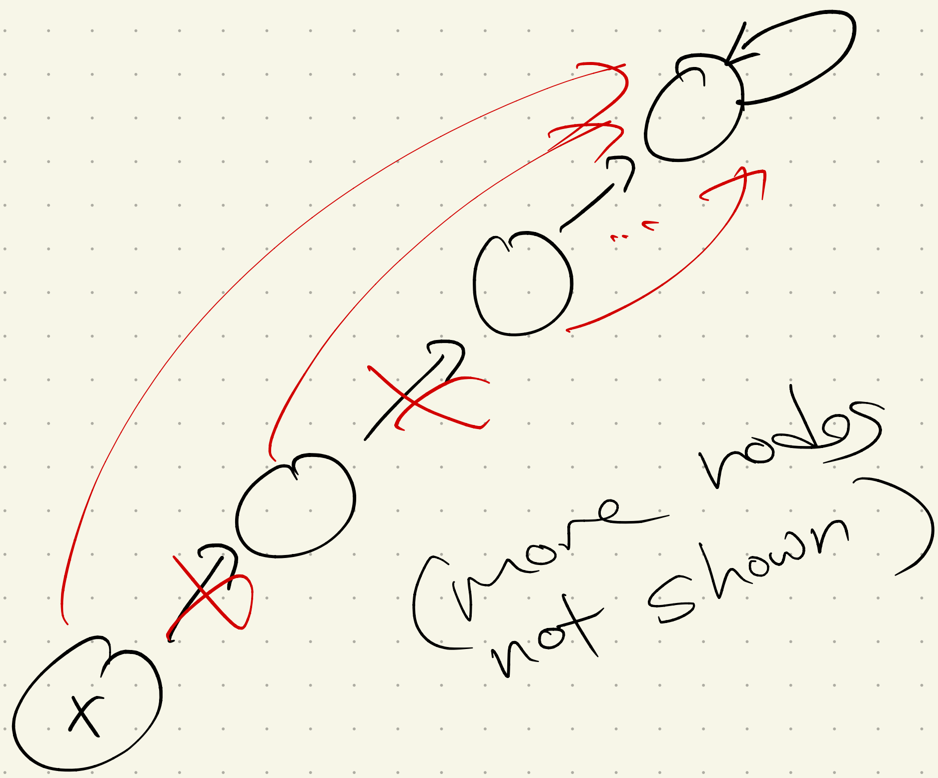
- union: 2 finds $+ O(1)$ updates

$\Rightarrow O(\log_2 n)$

$O(\lg n)$

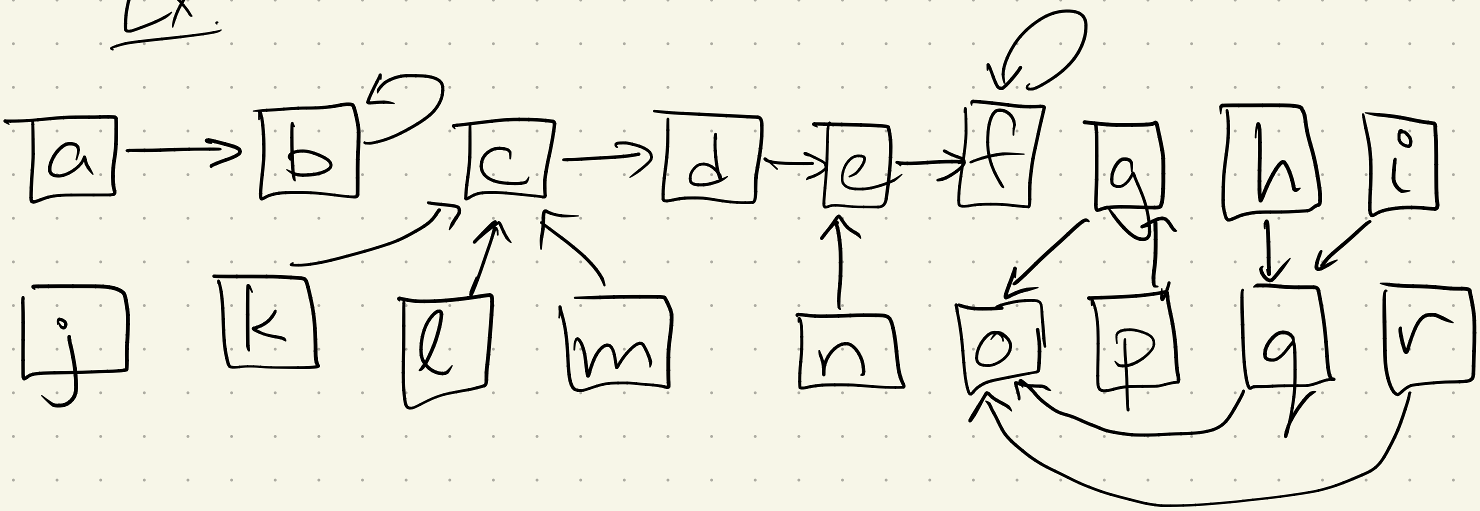
② Path compression:

During each find(x) make every node on the path from x to the root point to the root:



So: find(c):

Ex:



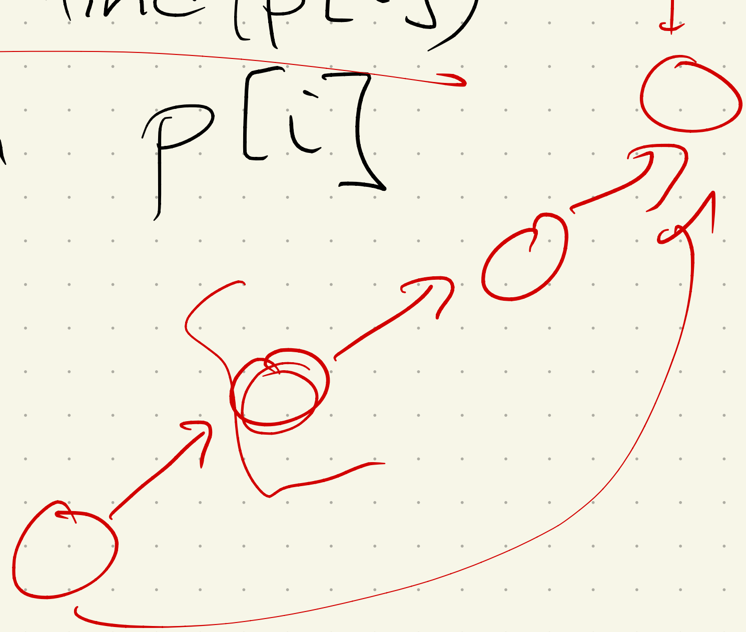
find(r):

Result: If find takes a long time, then later queries get faster!

Implementation:

find(i) {
    if p[i] == -1
        return i
    else
        ~~return find(p[i])~~
}

p[i] = find(p[i])
return p[i]



Formalizing the improvement

Amortized Analysis:

Worst case here:

Still $O(\log n)$!

Why?

Might get tree of height $\log n$

Amortized Analysis:

However, if we do <u>many</u> find (or unions), things get faster.

<u>Ex:</u> find(x) ← $O(\log n)$
find(x) ← $O(1)$

So: looking for average runtime of one operation, if doing many of them.

UF: size n

m finds

(Assume $m >> n$)

Thm: Any m find or union operations run in time $O((n+m)\log^* n)$.

??

Amortized cost of each:

$O(\log^* n)$

Actually $O(n\,\alpha^{-1}(m))$

not here!

tiny

(Next time)

$\log_2^* n$:= the number of times you apply the $\log_2$ until the result is ≤ 1.

$$= \begin{cases} 1 & \text{if } n \leq 2 \\ 1 + \log_2^*(\log_2 n) & \text{otherwise} \end{cases}$$

n	$\log^* n$
$(0, 1]$	0
$(1, 2]$	1
$(2, 2^2]$	2
$(4, 16] = (2^2, 2^{2^2}]$	3
$(16, 2^{16}] = (2^{2^2}, 2^{2^{2^2}}]$	4
$(2^{16}, 2^{(2^{16})}] = (2^{2^{2^2}}, 2^{2^{2^{2^2}}}]$	5
$\vdots$	

$2^{16} \approx 65{,}000$

Facts we need:

- If x is not a root, $\text{rank}(x) < \text{rank}(p[x])$
- When $p[x]$ changes, new leader's rank gets bigger
- Size of a set rooted at x is $\geq 2^{\text{rank}(x)}$

proof:

induction!

BC: rank = 0:

IS: rank $r > 0$:
At creation time, had two of equal rank

<u>Also</u>: For any r, there are at most $n/2^r$ objects with rank r.

<u>proof</u>: Fix r.

Note, only group leaders can change rank (going up by one)

So: when set leader changes from $r-1$ to r, mark entire set.

How many?

Leaders only increase, so each object is marked only once

$\Longrightarrow$

Back to the $\log_2^* n$ stuff:

<u>Define</u> $\text{Tower}(i) = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$ $\Big\}$ height i

so $\log_2^*(\text{Tower}(i)) = i$

<u>Define</u>: $\text{Block}(i) = [\text{Tower}(i-1)+1, \text{Tower}(i)]$

$\text{Block}(0) = [0,1]$ (just b/c)

$\text{Block}(1) = [2,2]$

$\text{Block}(2) = [3,4]$

$\text{Block}(3) = [5,16]$

$\text{Block}(4) = [17, 65536]$

$\text{Block}(5) = [65{,}536, 2^{65536}]$

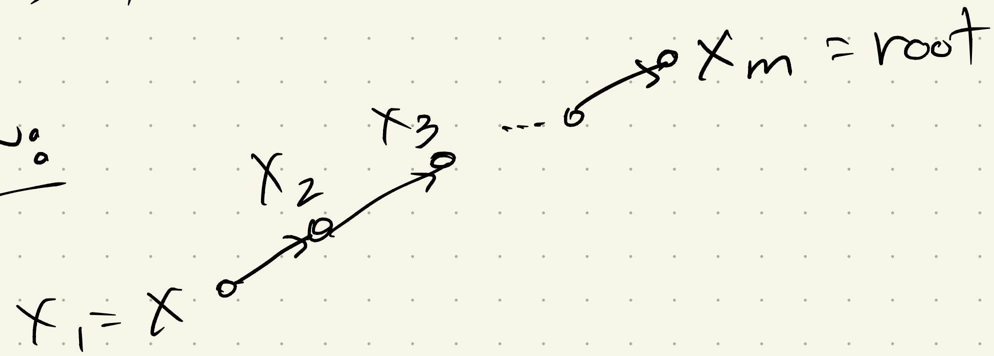
$\vdots$

<u>Now</u>: We know runtime of find(x) = length of x to root path:

Let our path π =

$x = x_1,\ p(x) = x_2,\ p(x_2) = x_3, \ldots, x_m = \text{root}$

<u>π:</u>

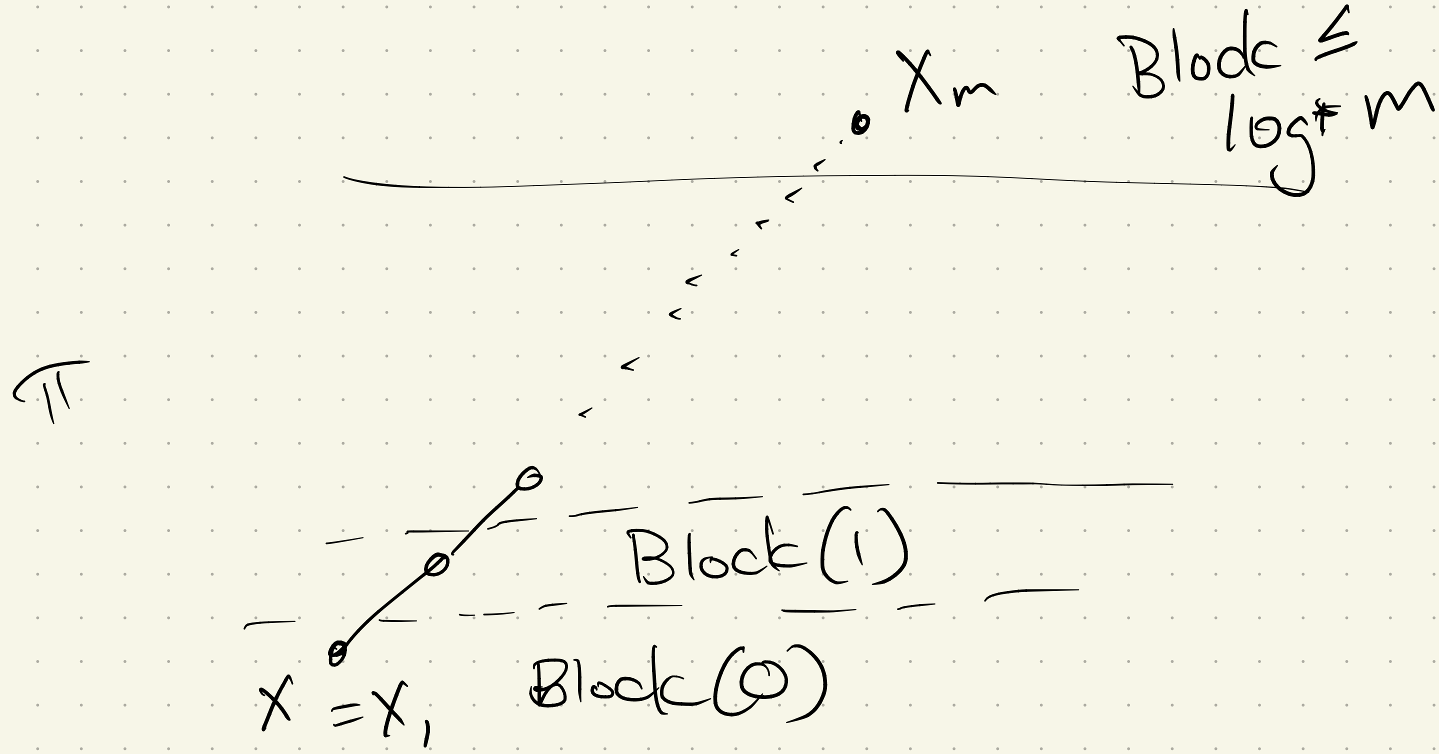


Say a node y is in $i^{\underline{th}}$ block if $\text{rank}(y) \in \text{Block}(i)$

In UF, max rank of <u>any</u> node is $\log n$.

(So only have $\log^* n$ blocks total.)

In these Blocks:



When we move $x_k \to p(x_k)$,
could stay in a block
(an internal jump)

or move to higher block
(a jump between
blocks)

Lemma: If x is an element in Block(i), at most |Block(i)| finds can pass through it until it moves to Block($i+1$).

pf:

Lemma: At most $\frac{n}{\text{Tower}(i)}$ nodes have rank in Block(i) over entire algorithm.

Pf: For rank r, know
$\leq \frac{n}{2^r}$ elements
at that rank.

$$\text{Block}(i) = [\text{Tower}(i-1)+1, \text{Tower}(i)]$$

so:

$$\sum_{k \in \text{Block}(i)} \frac{n}{2^k}$$

$$= \sum_{k=} \frac{n}{2^k}$$

$$=$$

Finally:

The number of internal jumps in i^{th} block is $O(n)$ (over entire set of m finds).

pf: • x in Block(i) can have $|\text{Block}(i)|$ internal jumps

• $|\text{Block}(i)| \leq \frac{n}{\text{Tower}(i)}$

So # internal jumps $\leq$

Thm: m operations on n elements in U-F take $O((m+n)\log_2^* n)$ total time.

pf: Either an operation is $O(1)$, or its runtime is $\approx$ (# internal jumps) + (# jumps b/t blocks)

internal jumps:

jumps b/t blocks: