Advanced Data Structures

Splay trees

Recap

Middle of U-F

- Union by rank
- Path compression

Facts we need:

- Once a node stops being a root, it will never be a root again.

  Why? consider unions + finds

  find: only changes parents, stops at root

  union: one root becomes a child - can be path compressed, but not a root

- Once not a root, a node's rank never changes.

  Why? Well, when does rank get changed?

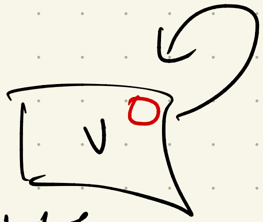
  not in find

  in union, only changes the end root

- Ranks are increasing in any leaf-to-root path.

Proof: induction on time (i.e # of ops)

base case Singleton

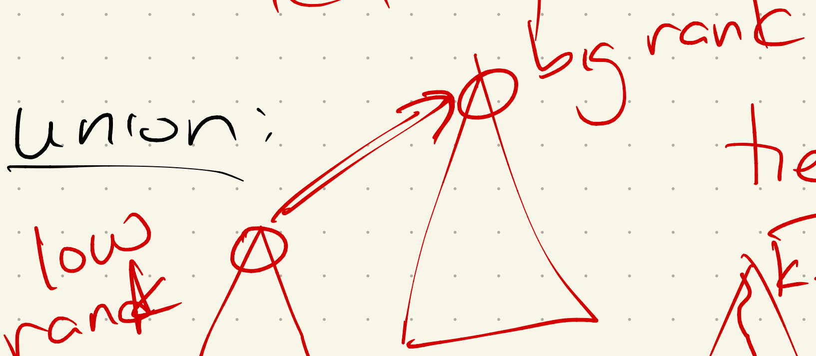


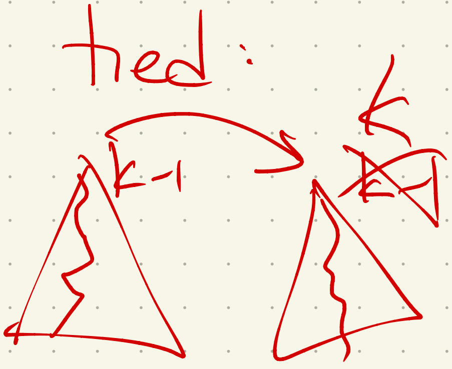
Ind step: Consider t^th operation = either:

makeset:

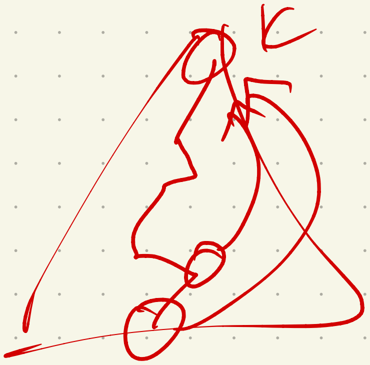
no other root to leaf paths change

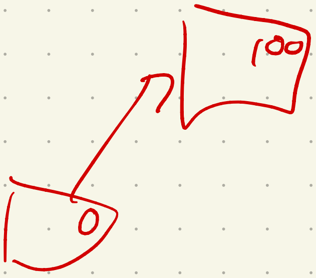
union:





find:

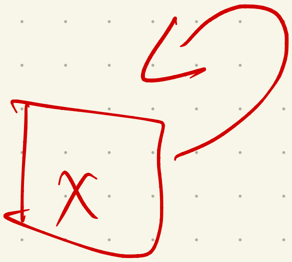




Lemma: When a node gets rank k, there are $\geq 2^k$ items in its tree.

Proof: induction on rank:

$r=0$: $2^0 = 1$



Now assume true for anything $< r$, & consider the first time rank $= r$:

↳ must be a union, with two roots that have rank $r-1$

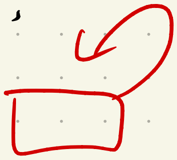
By IH, those each have $\geq 2^{r-1}$

there are 2 of them.

total # $\geq 2 \cdot 2^{r-1} = 2^r$ ∎

Lemma: For any r, there are at most $(\leq)$ $n/2^r$ objects with rank r through *entire* execution.

Proof: More induction!

r=0: rank 0: 

n elements: $\frac{n}{2^0} = n$

r>0: If a node v has rank r:

we will "charge" it to the two nodes u & v of rank r-1 at time of union.

After union, neither can *ever* make another rank r node.

So: if $\frac{n}{2^{r-1}}$ at rank r-1,

then it takes 2 of rank r-1 to make one of rank r: $\frac{n}{2^{r-1}} \cdot \frac{1}{2} = \frac{n}{2^r}$

Side note:

Worst case $\log n$:

$\frac{n}{2^r}$ at rank r.

⇒ highest rank? $\log n$

$\frac{n}{2}/2/2/2$

(And so tree height can't be larger)

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

Define $\text{Tower}(i) = \left. 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \right\} \text{height } i$

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

Define: $\text{Block}(i) =$

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

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

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

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

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

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

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

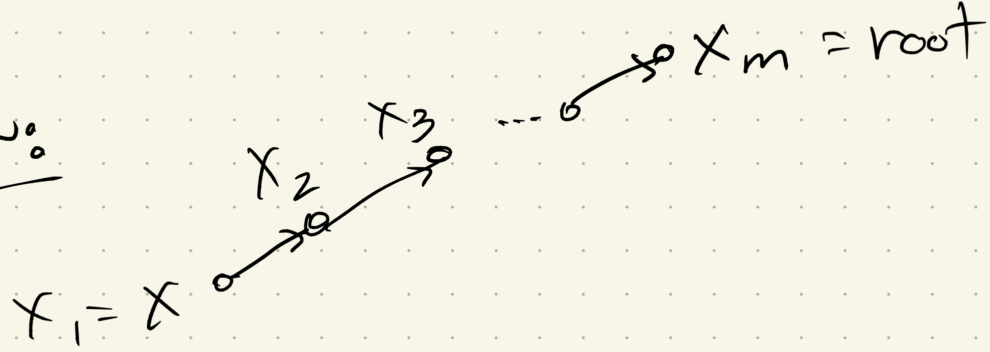
$\vdots$

<u>Now</u>: We know runtime of $\text{find}(x)$ = length of x to root path:

Let our path Π =

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

Π:

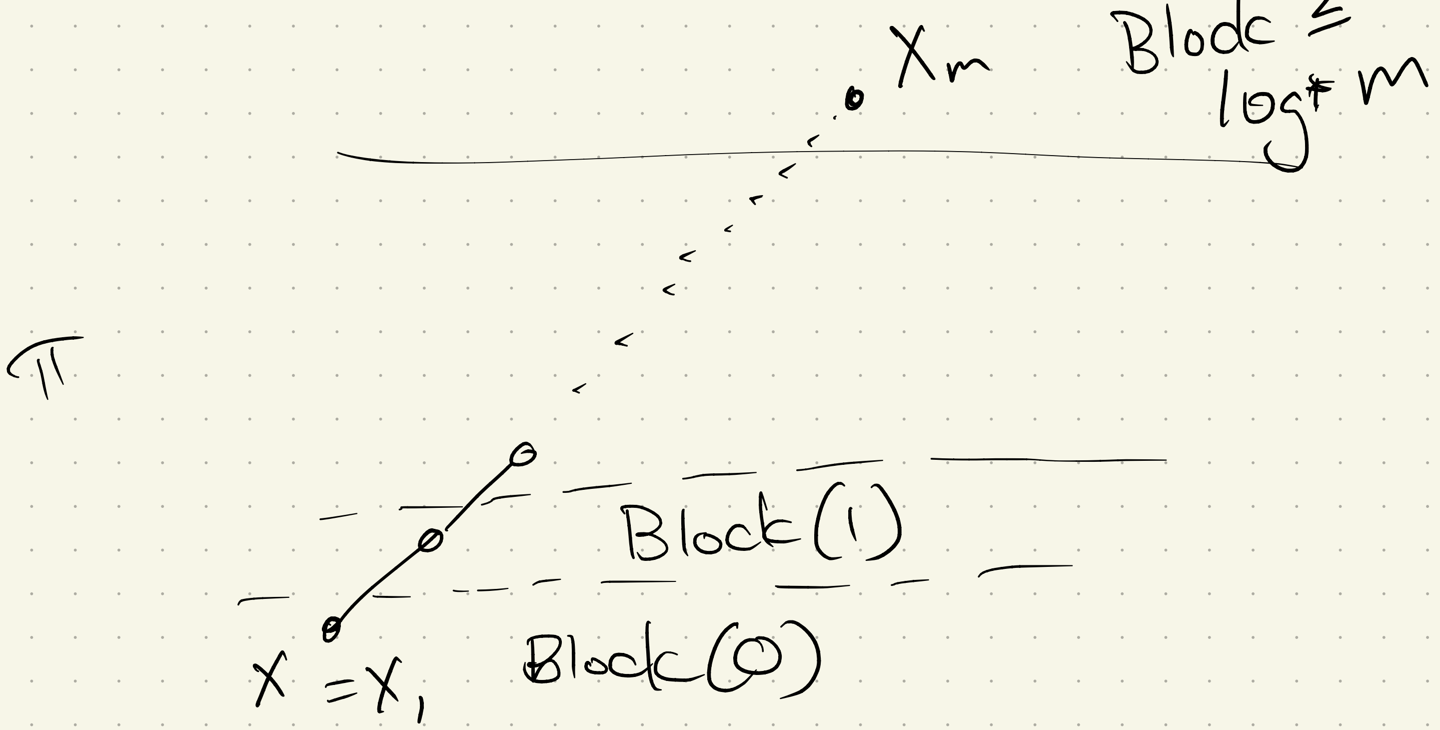


Say a node y is in i^{th} block if $\text{rank}(y) \in \text{Block}(i)$

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)

$\log^* n$ blocks, & never move down

$\Rightarrow (\log^* n)\, n$

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: What happens with each find? path compression!

Must get higher ranked parent each time we path compress. are only |Block(i)| options in Block(i)

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

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

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

so:

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

$$= \sum_{k=\text{Tower}(i-1)+1}^{\text{Tower}(i)} \frac{n}{2^k}$$

$$= n \left[\sum_{k=\text{Tower}(i-1)+1}^{\text{Tower}(i)} 2^{-k} \right] \le n \left(\frac{1}{2^{\text{Tower}(i-1)}} \right) = \frac{n}{\text{Tower}(i)}$$

Finally:

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

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

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

So # internal jumps $\leq$

$$|\text{Block}(i)| \cdot \#\text{in block}(i)$$

$$\leq \text{Tower}(i) \cdot \frac{n}{\text{Tower}(i)}$$

$$\leq n$$

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

(This is upper bnd)

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

\# internal jumps:

$O(n)$ per ~~level~~ block

\# jumps b/t blocks:

$\log^* n$

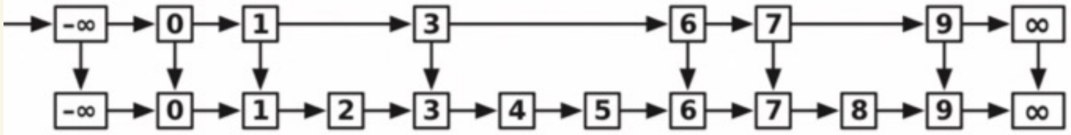
Next: Skip Lists

(Bill Pugh, 1990)

An alternative to balanced binary search trees.

Essentially, just a sorted list where we add shortcuts - but to speed up, we'll duplicate some elements.

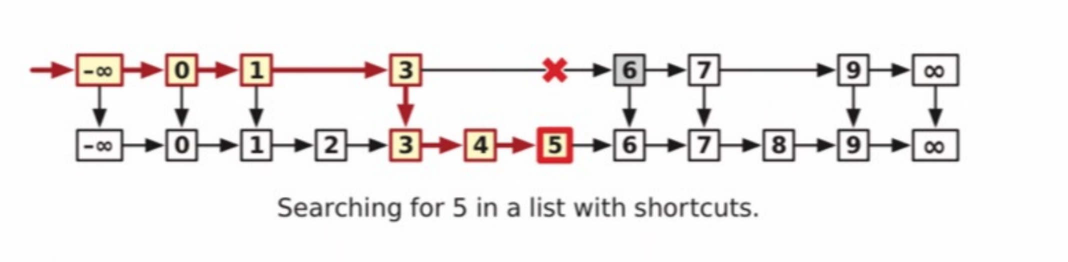
For each item, duplicate with probability $\frac{1}{2}$:



A linked list with some randomly-chosen shortcuts.

plus some sentinal nodes

Searching:



Searching for 5 in a list with shortcuts.

Scan in top list.

If found, great!

Otherwise:

Some probability!

Expectation: $\sum_{\text{values possible}} (\text{value})(\text{prob of value})$

<u>Ex</u>: 6 sided dice

$E[\text{value}) =$

Each node is copied with prob $= \frac{1}{2}$,

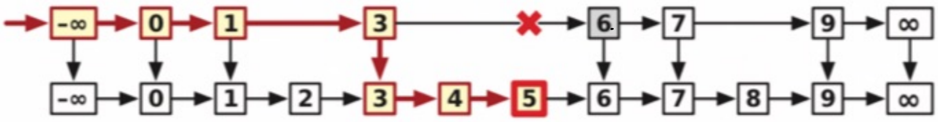
$E[\#\text{ nodes in top}] =$

Prob[a node is follow by k without duplicates]

=

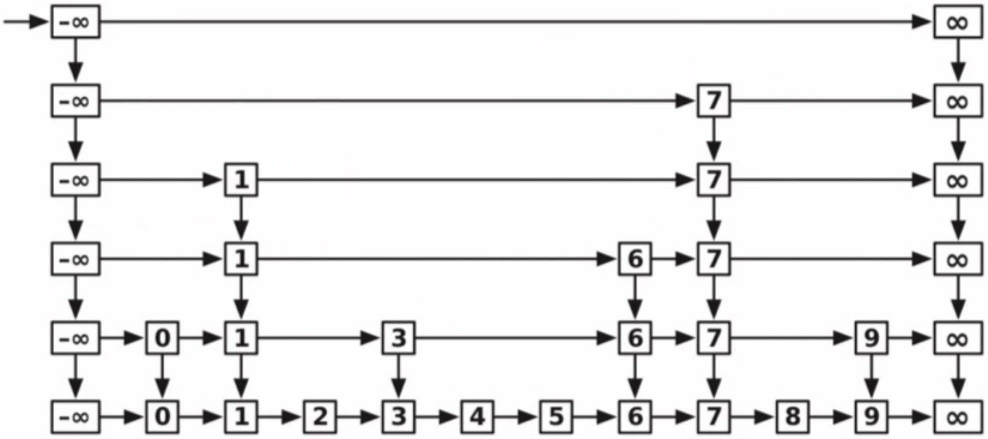
So: Expected [# comparisons in lower list]

= 1 +



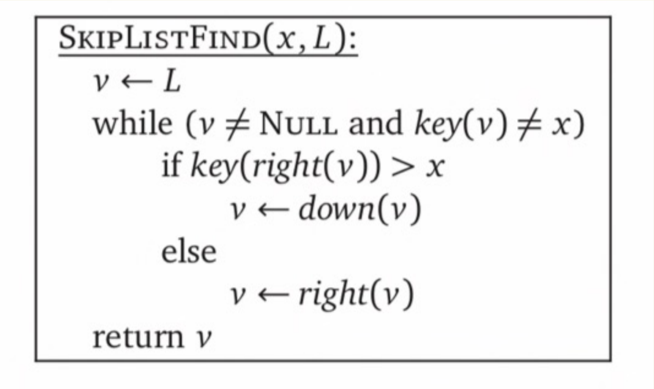
Searching for 5 in a list with shortcuts.

What next? Recurse!

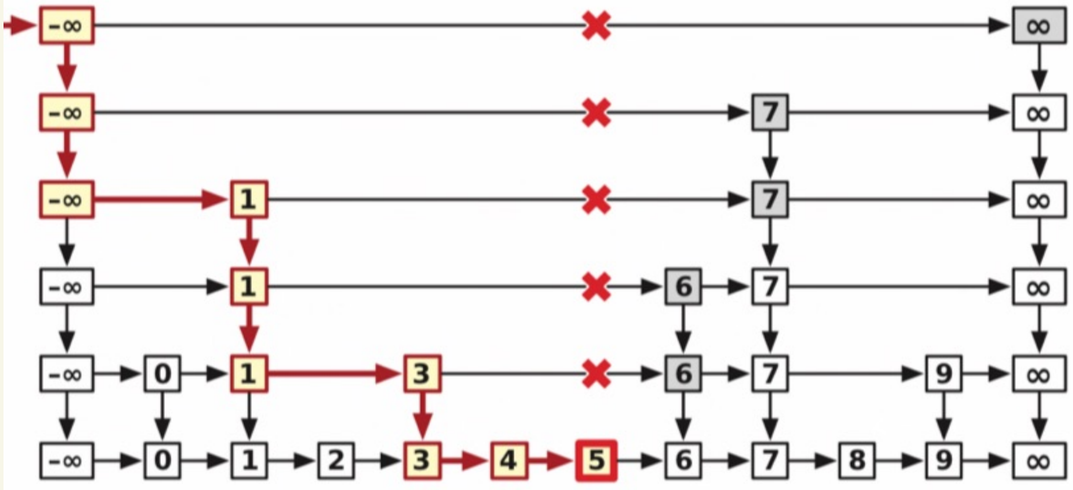


A skip list is a linked list with recursive random shortcuts.

To search:



```
SKIPLISTFIND($x, L$):
    $v \leftarrow L$
    while ($v \neq$ NULL and $key(v) \neq x$)
        if $key(right(v)) > x$
            $v \leftarrow down(v)$
        else
            $v \leftarrow right(v)$
    return $v$
```



Searching for 5 in a skip list.

How many levels?

Well, $E[\text{size at level } i]$

$= \frac{1}{2} E[\text{size at level } i-1]$

So (intuitively):

$O(\log n)$ runtime

Each time we add a level,

$E[\# \text{ searches}]$

goes down by $\frac{1}{2}$.

More formally?

See posted notes!

(Assumes some probability...)