Alcan 5

<u>Summary</u>

Binary Search Trees

using:

$\lg n$ W, S { find, insert, delete

singleton
split
join

$W: O\left(m \cdot \lg\left(\frac{n+m}{m}\right)\right)$
$S: O\left(\lg^2(m+n)\right)$ { intersection, union, difference

optimal

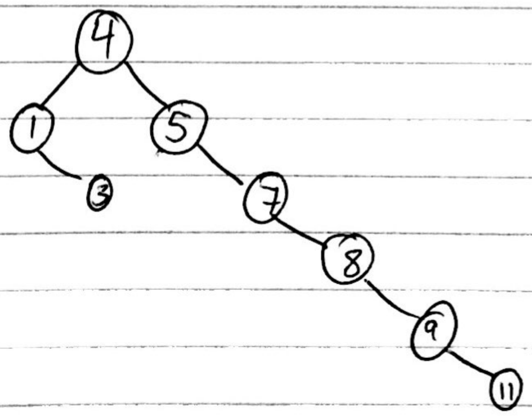
<u>Balancing</u> (many ways)

This lecture – <u>randomized</u> technique

given set of keys

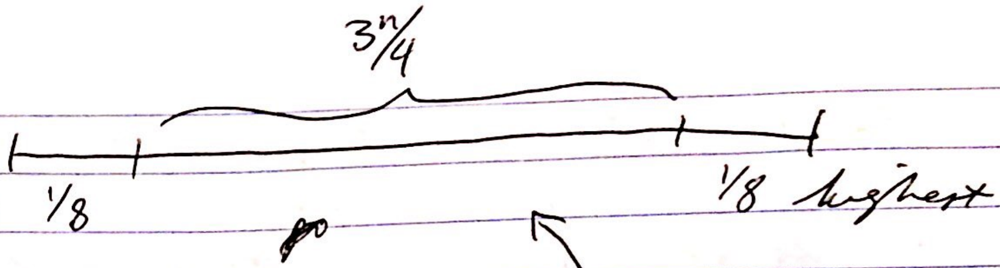
4, 1, 5, 7, 8, 9, 3, 11

w/ random permutation



could look terrible (in-order / rev. order)

probably average would look more balanced order keys



imagine throwing darts, here w/ 3/4 prob.
(which key goes first)

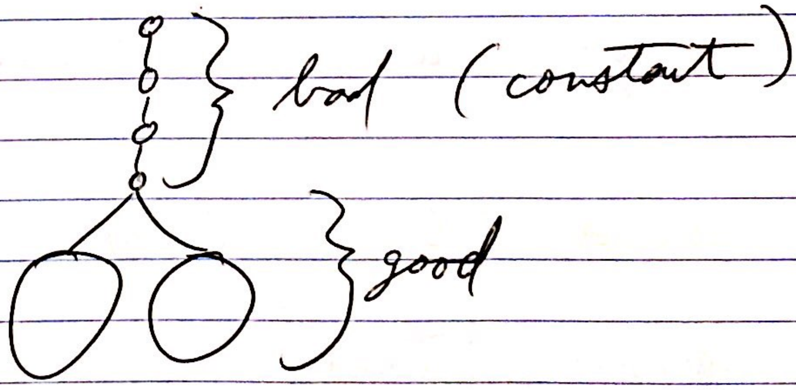
get a constant factor partition of trees

throw 4 times $\frac{1}{4}\cdot\frac{1}{4}\cdot\frac{1}{4}\cdot\frac{1}{4}$

very unlikely to hit bad part (extremes)

basic idea behind treaps

there might be a few bad keys
but (eventually) soon hit a good key



build data structure out of this intuition

— assign $\boxed{\text{priority}}$ to each key

intuition:
position of key in insertion order

~~pick~~ simulate random permutation

pretend priorities are permutation

observation: path root → leaf
is always sorted

node has smaller priority than children

(∴ heap ordered)

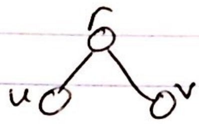
keys are tree-ordered

called tree-heap

hence, $\boxed{\text{treap}}$

Scanned with CamScanner

```
datatype α treap                    priority
  = Node of α treap × (α × int) × α treap
  | Leaf
```



priority
$p(r) \leq p(u)$
$p(r) < p(v)$

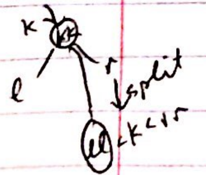
balanced $\left\{ \begin{array}{l} \\ \text{tree height} \end{array} \right.$ $O(\lg n)$ w/ high probability

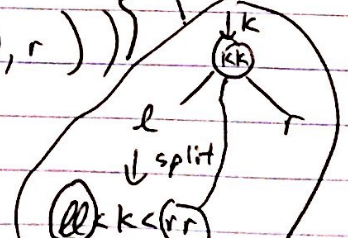
```
fn singleton k =
  let p = random()
  in Node(Leaf, (k,p), Leaf)
```

want each key to have unique priority.
- so should be at least couple 'n' times
- collisions dont break but bounds arent as tight

```
fn split t k =
  case t of
    Leaf => (false, Leaf, Leaf)
    Node(l, (kk,p), r) =>
      if k=kk then (true, l, r)
      elseif k < kk then
        let (found, ll, rr) = split l k
        in (found, ll, join(rr, (kk,p), r))
      else (k > kk) then
        let (found, ll, rr) = split r k
        in (found, join(l,(kk,p),ll), rr)
```

(know ll < k < RR from recursive split)





join

use priorities
(not used in split)

```
fun join t u =
  case (t, u) of
    (Leaf, u) => u
    (t, Leaf) => t
    (Node(lt, (kt, pt), rt),
     Node(lu, (ku, pu), ru))
    => if pt < pu
       then Node(lt, (kt, pt), join(rt, u))
       else Node(join(t, lu), (ku, pu), ru)
```

ok because of the assumption that the keys in left arg to join are less than keys in right arg.

(tweak split to use joinM instead of Node)

all known balancing operations happen through these ~~op~~ operation (joins)

Augmentation

want to keep info about subtrees

ex/ might want to ask for the n'th element.

want to index into structure

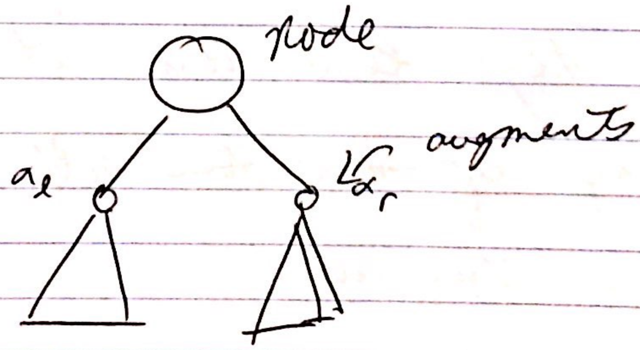
not hard to augment BST

w/ such operations

ex/ want max salary of elements

keep info at root of subtree

(summarizes info)



can just run fn ~~of~~ on α_l, α_r

to set result for node.

max salary - just use $\max(\alpha_l, \alpha_r)$

ex/ for indexed lookup

keep size of subtree

can skip ahead based on index & size of subtree

select ~~rank~~ query

ith element in the sorted order of key in lg n work

rank query

given a key,

find # of keys less than it

(sum up left ~~most~~ ~~trees~~ subtree size ~~rank~~)

lg n time (work)