

Lazy Data Structures

Roadmap for rest of semester

"Average" analysis of algorithms
↑
(w/out sacrificing worst-case robustness)

① Amortized analysis ("average" in hindsight)

② Randomized algorithms ("average" over random bits)

Both are very practical, different perspectives

Incrementing Binary Counters

`i++;`

`//increment i`

In bits:

0, 1, 10, 11, 100

From $i = 1$ to n , how many total slips?

*least significant

how often total

• 1st* bit

every time

n

• 2nd bit

every 2 times

$\frac{n}{2}$

• 3rd bit

every 4 times

$\frac{n}{4}$

⋮

x • kth bit

every 2^{k-1}

times $\frac{n}{2^{k-1}}$

$2n$

0, 1, 10, 11, 100, 101, ...
1111111111111111111111111111111111
1000000000000000000000000000000000

• 1st^{*} bit

how often total
every time n

• 2nd bit

every 2 times $\lfloor \frac{n}{2} \rfloor$

• 3rd bit

every 4 times $\lfloor \frac{n}{4} \rfloor$

⋮

• kth bit

every 2^{k-1} times $\lfloor \frac{n}{2^{k-1}} \rfloor$

n increments $\Rightarrow O(n)$ total time

each increment takes $O(1)$ on average

" $O(1)$ amortized time"

Amortized Analysis

	how often	total
• 1st [*] bit	every time	n
• 2nd bit	every 2 times	$\lfloor \frac{n}{2} \rfloor$
• 3rd bit	every 4 times	$\lfloor \frac{n}{4} \rfloor$
⋮		
• kth bit	every 2^{k-1} times	$\lfloor \frac{n}{2^{k-1}} \rfloor$

" T amortized time"

$\Rightarrow$ n operations take nT time

Multiple operations $OP_1, \dots, OP_k$

" OP_1 takes T_1 amortized time, OP_2 take $\dots$

OP_k takes T_k amortized time"

$OP_{i_1} + OP_{i_2} + OP_{i_3} + \dots \Rightarrow T_{i_1} + T_{i_2} + T_{i_3} + \dots$

Ways to do amortized analysis

- direct analysis (like above)
- credit schemes ("banker method")
- potential function method

	how often	total
• 1st [*] bit	every time	n
• 2nd bit	every 2 times	$\lfloor \frac{n}{2} \rfloor$
• 3rd bit	every 4 times	$\lfloor \frac{n}{4} \rfloor$
⋮		
• kth bit	every 2^{k-1} times	$\lfloor \frac{n}{2^{k-1}} \rfloor$

"T amortized time"

⇒ n operations take $O(nT)$ time

Multiple operations $OP_1, \dots, OP_k$

" OP_1 takes T_1 amortized time, OP_2 take $\dots$

OP_k takes T_k amortized time"

Credit schemes

Idea: buy credits ahead of time to pay for work later

$$\underline{(\text{running time})} + \alpha(1) \cdot \underline{(\Delta \text{ credit})} \leq \text{amortized time}$$

(net increase)

Credit schemes Idea: buy credits ahead of time to pay for work later

$$\text{(running time)} + \underbrace{\alpha(1) \cdot (\Delta \text{ credit})}_{\text{(net increase)}} \leq \text{amortized time}$$

e.g. for binary counter, when incrementing:

0, 1, 10, 11, 100, 101, ...
11111111111111111111111111111111
10000000000000000000000000000000

	how often	total
• 1 st bit	every time	n
• 2nd bit	every 2 times	$\lfloor \frac{n}{2} \rfloor$
• 3rd bit	every 4 times	$\lfloor \frac{n}{4} \rfloor$
⋮		
• kth bit	every 2^{k-1} times	$\lfloor \frac{n}{2^{k-1}} \rfloor$

put: 1 credit on first bit
 $\frac{1}{2}$ credit on 2nd bit
 $\frac{1}{4}$ credit on 3rd bit
 $\vdots$
 $+ \frac{1}{2^{i-1}}$ credits on i th bit

2 credits total

when we need to flip bit i , we have already saved 1 credit to pay for it

Credit schemes

Idea: buy credits ahead of time to pay for work later

$$(\text{running time}) + \underbrace{\alpha(i)}_{(\text{net increase})} \cdot (\Delta \text{ credit}) \leq \text{amortized time}$$

Main rule:

Only spend credits that you already saved

eg. for binary counter, when incrementing:

0, 1, 10, 11, 100, 101, ...
11111111111111111111111111111111
10000000000000000000000000000000

	how often	total
• 1st bit	every time	n
• 2nd bit	every 2 times	$\lfloor \frac{n}{2} \rfloor$
• 3rd bit	every 4 times	$\lfloor \frac{n}{4} \rfloor$
⋮		
• kth bit	every 2^{k-1} times	$\lfloor \frac{n}{2^{k-1}} \rfloor$

put: 1 credit on first bit
 $\frac{1}{2}$ credit on 2nd bit
 $\frac{1}{4}$ credit on 3rd bit
⋮
+ $\frac{1}{2^{i-1}}$ credits on ith bit

2 credits total

when we need to flip bit i , we have already saved 1 credit to pay for it

Potential method

Invent "potential Φ "



Amortized cost of operation will be

$$(\text{real time}) + \Delta\Phi$$

(change in potential)

Potential method

Amortized cost of operation will be

(real time) + $\Delta\Phi$
(change in potential)

e.g. binary counter

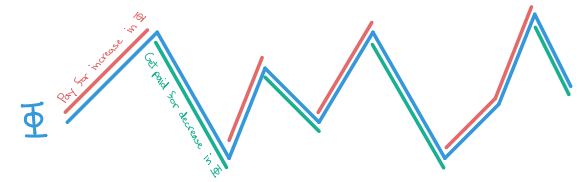
Φ = # bits set to 1

then for each inc.,

(real time) + $\Delta \bar{\Phi} \leq O(1)$

leach bit reset to 0 paid by $\Delta\Phi$)

"potential $\bar{\Phi}$ "

[illegible]

	how often	total
• 1st [*] bit	every time	n
• 2nd bit	every 2 times	$\lfloor \frac{n}{2} \rfloor$
• 3rd bit	every 4 times	$\lfloor \frac{n}{4} \rfloor$
⋮		
• kth bit	every 2^{k-1} times	$\lfloor \frac{n}{2^{k-1}} \rfloor$

Slip

Potential method

Amortized cost of operation will be

(real time) + $\Delta\Phi$
(change in potential)

e.g. binary counter

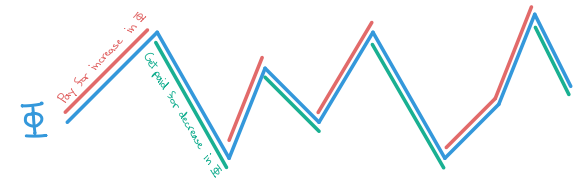
Φ = # bits set to 1

then for each inc.,

$$(\text{real time}) + \Delta \bar{\Phi} \leq O(1)$$

leach bit reset to 0 paid by $\Delta\Phi$)

"potential Φ "

[illegible]

	how often	total
• 1st [*] bit	every time	n
• 2nd bit	every 2 times	$\lfloor \frac{n}{2} \rfloor$
• 3rd bit	every 4 times	$\lfloor \frac{n}{4} \rfloor$
⋮		
• kth bit	every 2^{k-1} times	$\lfloor \frac{n}{2^{k-1}} \rfloor$

Potential method

Invent "potential Φ "



Pay for increase in Φ

Get paid for decrease in Φ

" Amortized cost of operation will be "

$$(\text{real time}) + \underbrace{\Delta\Phi}_{(\text{change in potential})}$$

why? suppose n operations

T_i = time of i th operation

Φ_i = Φ after i operations

$$\Delta\Phi_i = \Phi_i - \Phi_{i-1}$$

$$\sum_{i=1}^n (T_i + \Delta \Phi_i) = (\text{total time}) + \overset{\geq 0}{\Phi_n}$$

$$\geq (\text{total time})$$

$$(\text{total time}) \leq \sum_{i=1}^n (T_i + \Delta \Phi_i)$$

Ways to do amortized analysis

- direct analysis (like above)
- credit schemes ("banker method")

put: 1 credit on first bit
 $\frac{1}{2}$ credit on 2nd bit
 $\frac{1}{4}$ credit on 3rd bit
 $\vdots$
+ $\frac{1}{2^{i-1}}$ credits on i th bit

2 credits total

- potential function method

$\Phi = \# \text{ bits set to } 1$

	how often	total
• 1st [*] bit	every time	n
• 2nd bit	every 2 times	$\lfloor \frac{n}{2} \rfloor$
• 3rd bit	every 4 times	$\lfloor \frac{n}{4} \rfloor$
$\vdots$		
• k th bit	every 2^{k-1} times	$\lfloor \frac{n}{2^{k-1}} \rfloor$

"T amortized time"

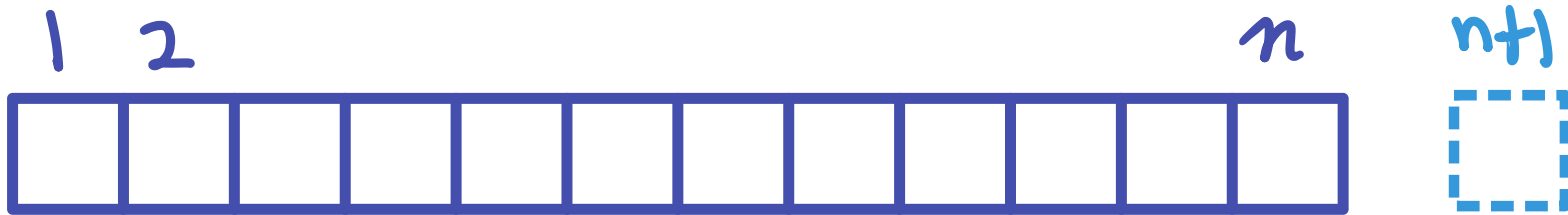
$\Rightarrow n$ operations take $O(nT)$ time

Multiple operations $OP_1, \dots, OP_k$

" OP_1 takes T_1 amortized time, OP_2 take $\dots$

OP_k takes T_k amortized time"

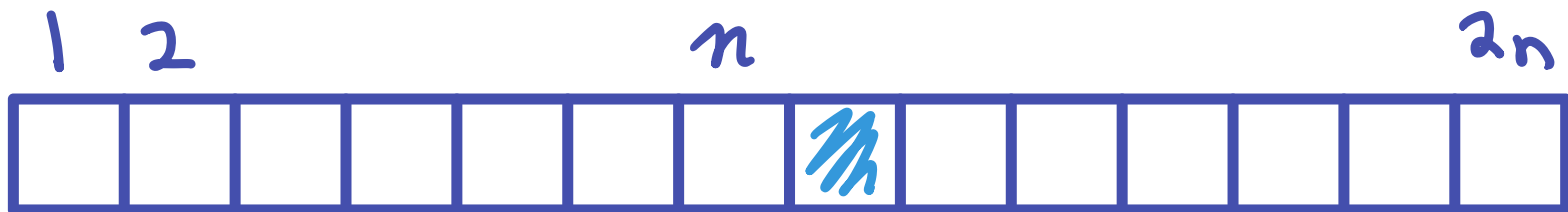
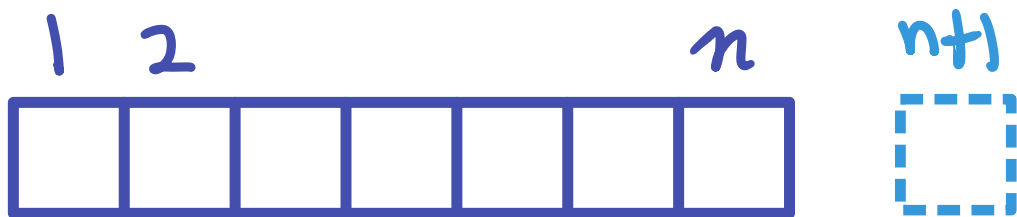
Array-backed lists



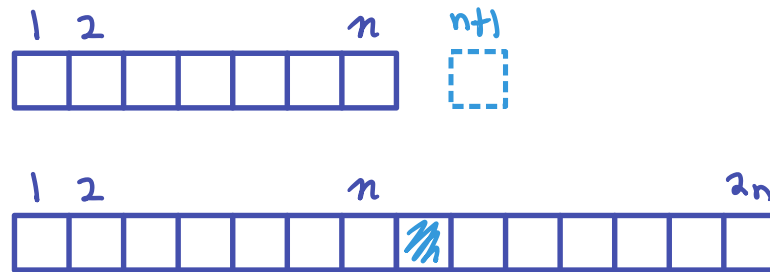
Good:

Bad:

Doubling Trick



Doubling Trick



Theorem

w/ doubling trick, arraylist has

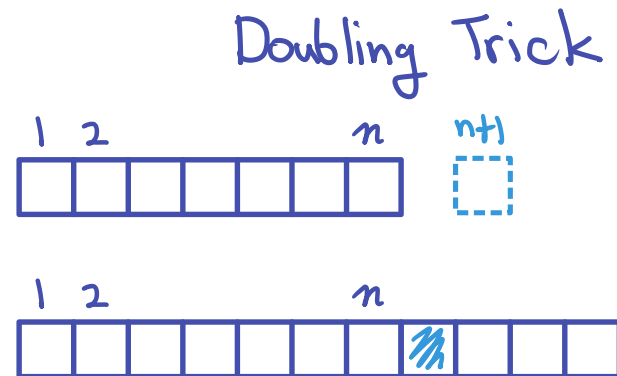
- lookup in $O(1)$
- append in $O(1)$ amortized

Theorem

w/ doubling trick, arraylist has

- lookup in $O(1)$
- append in $O(1)$ amortized

Proof



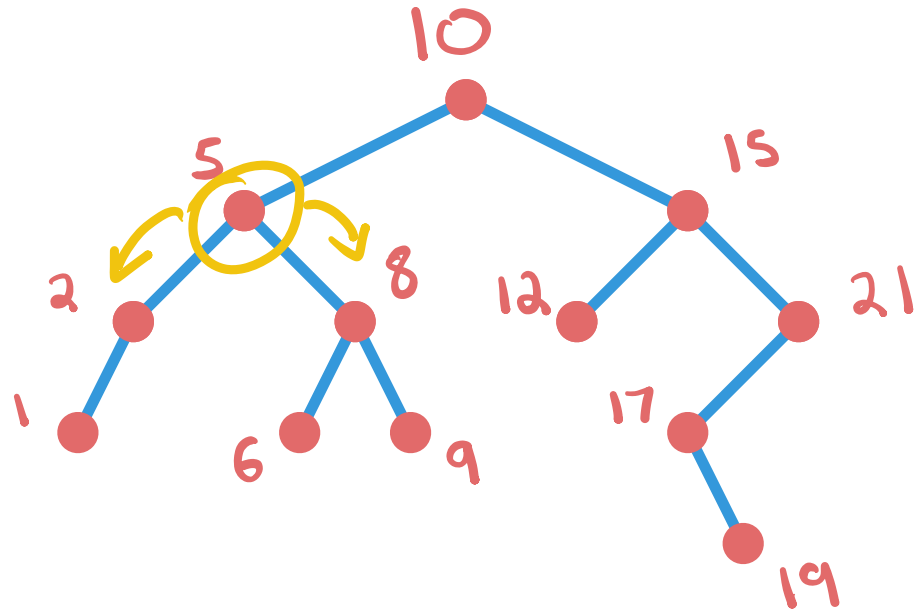
Search trees

organizes keys
in a tree

Invariant:

a node's # is

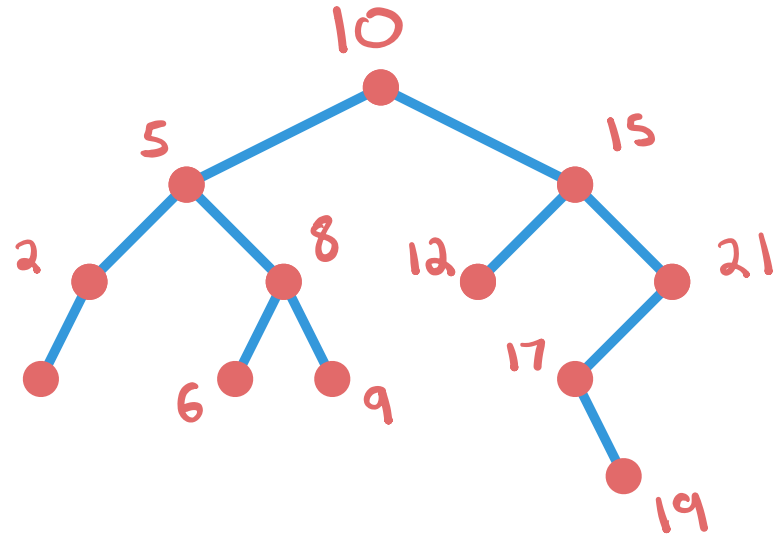
- $\geq$ any # in left subtree
- $\leq$ any # in right subtree



Invariant:

a node's # is

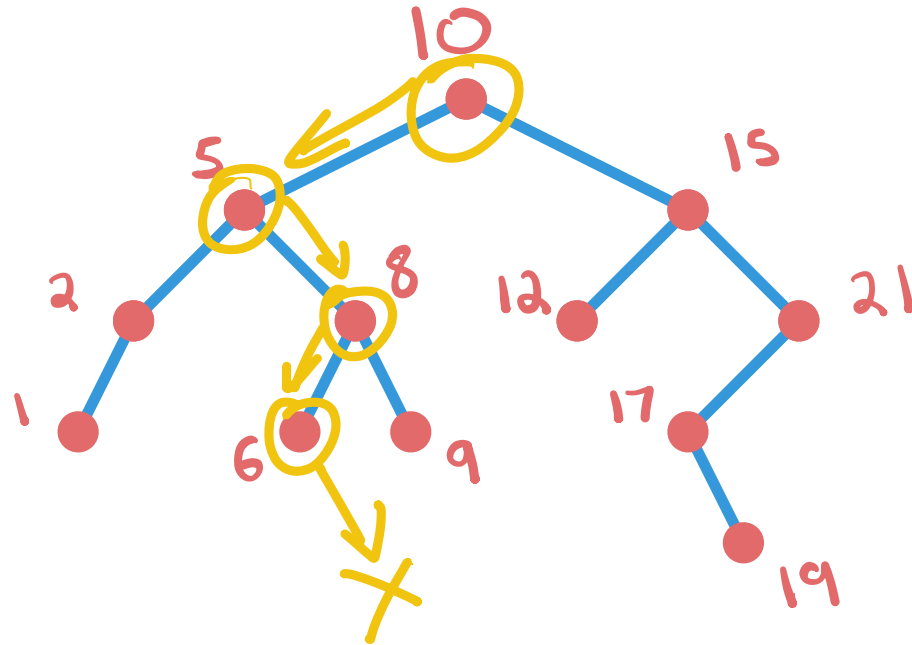
- $\geq$ any # in left subtree
- $\leq$ any # in right subtree



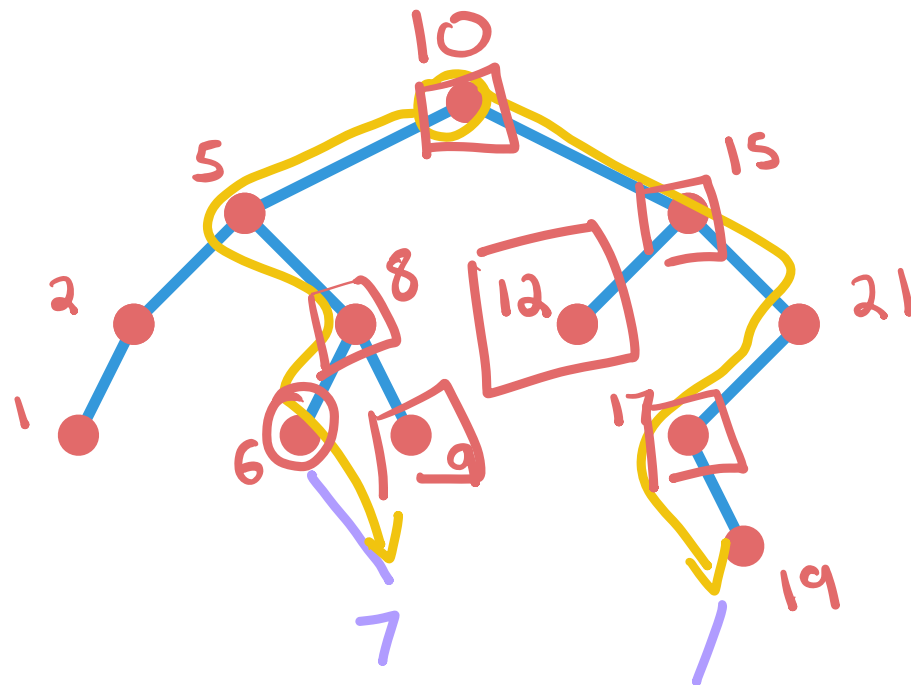
Leads to operations like:

- is a given key in the set?
- what is the first key $\geq x$
- list all keys between x and y
- how many keys are between x and y

does the tree contain 7?



list all keys between 7 and 18

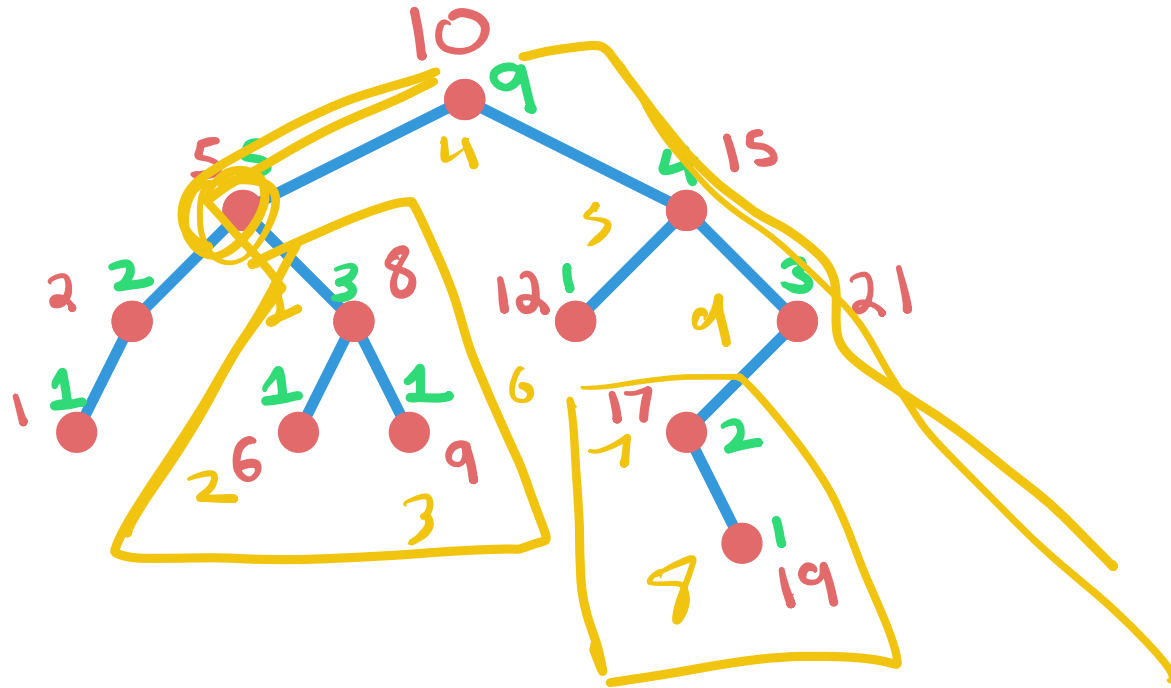


$O(\# \text{ keys in between})$

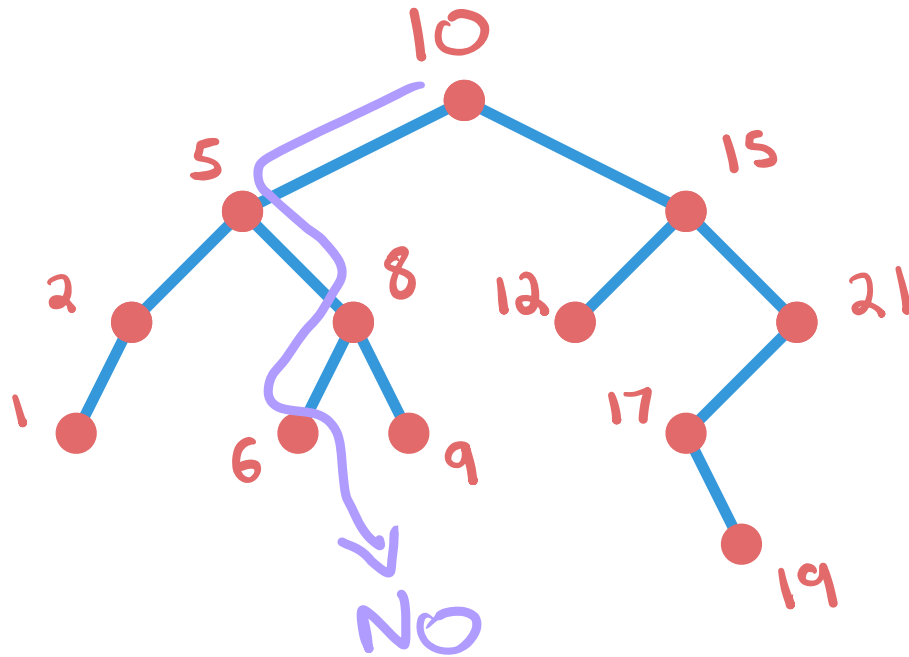
would be
here

18
would be
here

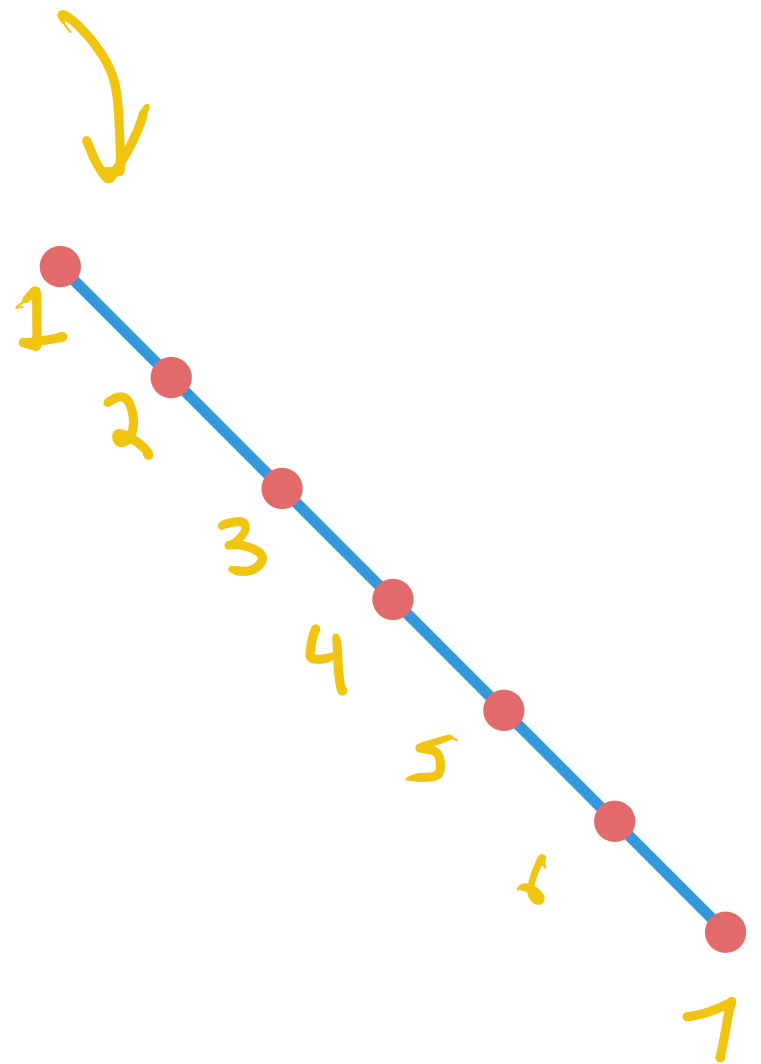
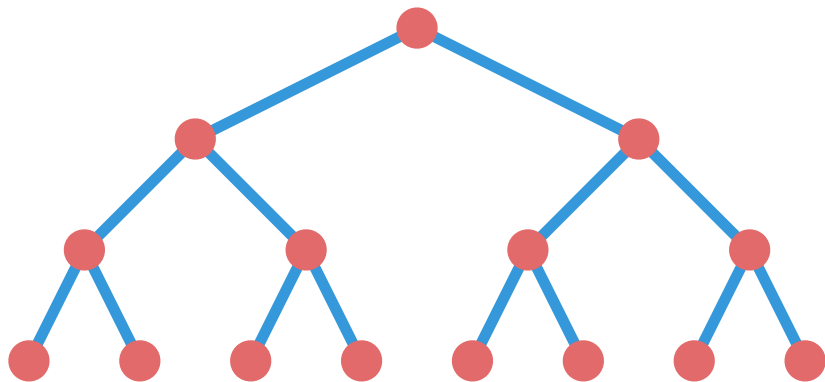
how many keys are between 5 and 25



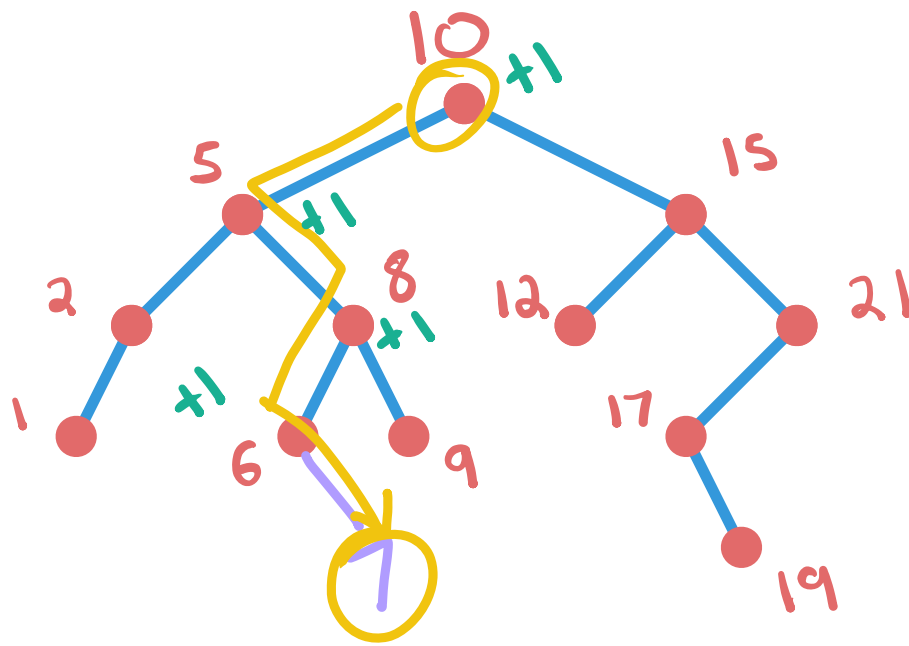
if we write down # nodes in each subtree, then faster



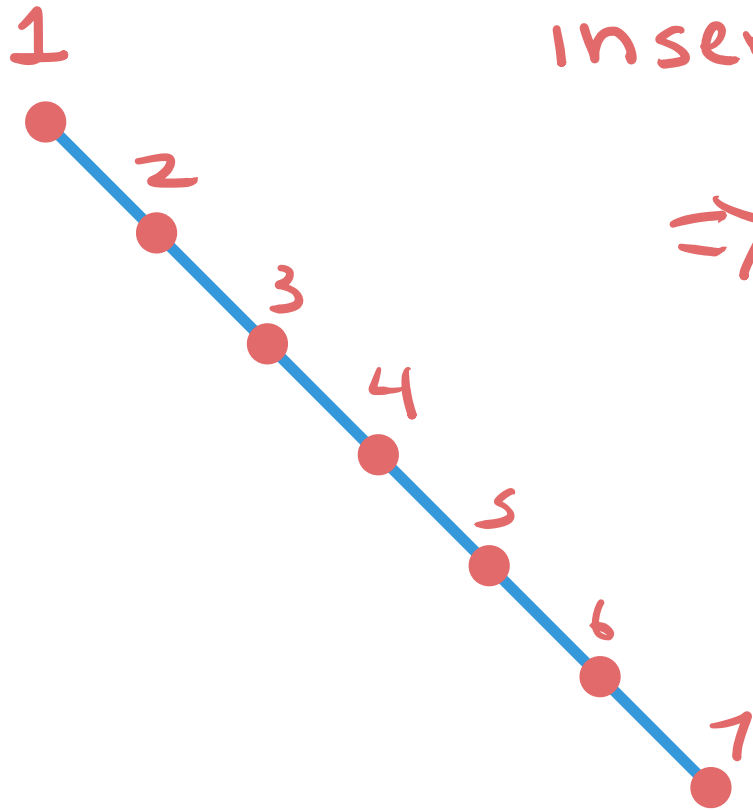
operations take time proportional
to the height of a tree



Insert 7

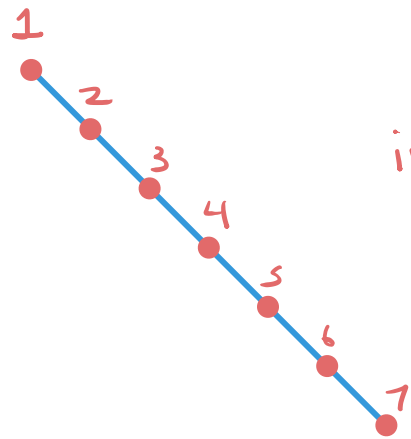


Most natural method for insertion
is as leaves (search for where
the key would be, and put it there)



insert 1, 2, 3, 4, 5, 6, 7

$\Rightarrow$ imbalanced



insert 1, 2, 3, 4, 5, 6, 7

$\Rightarrow$ imbalanced

How to keep a tree balanced?

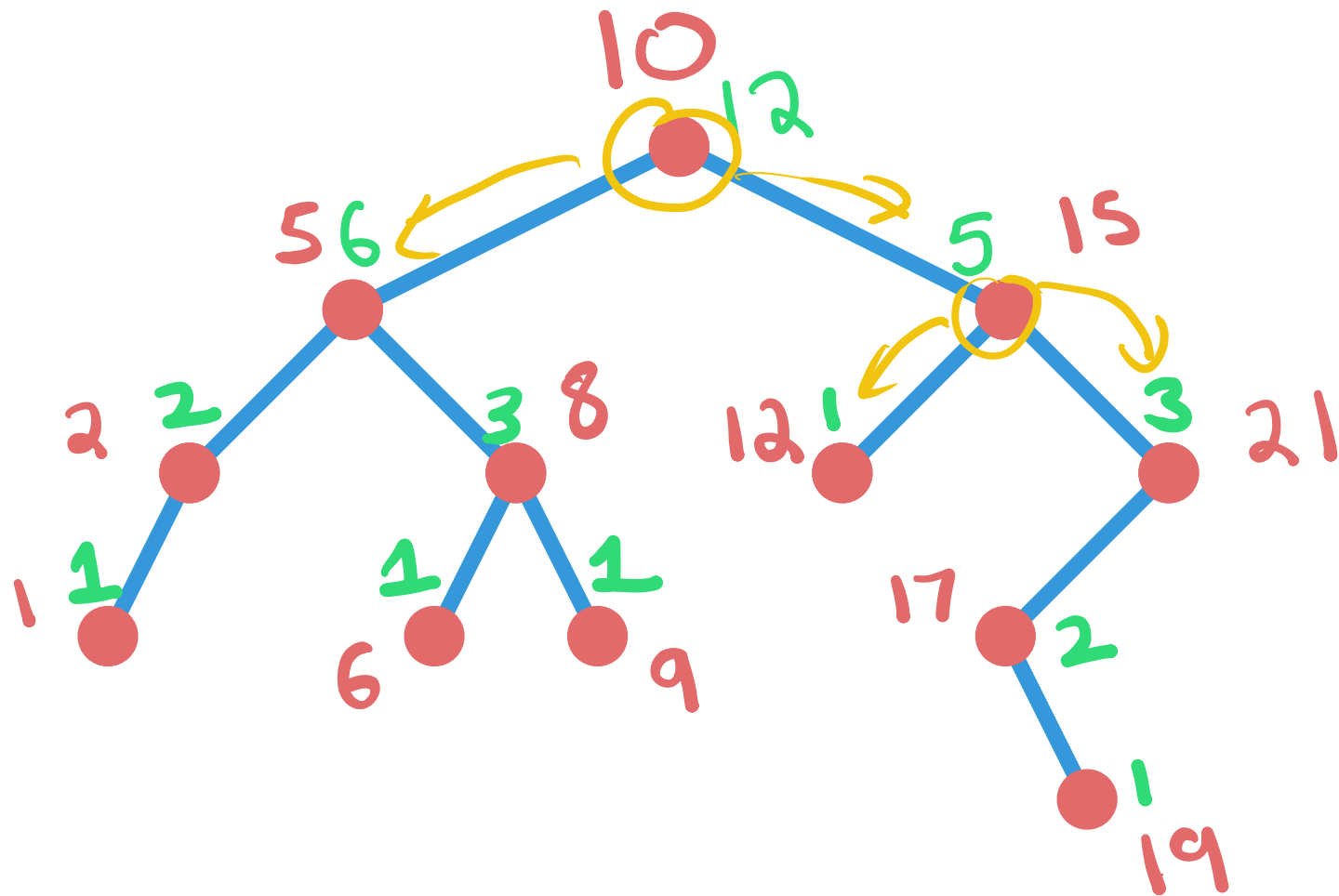
AVL trees

Red-black trees

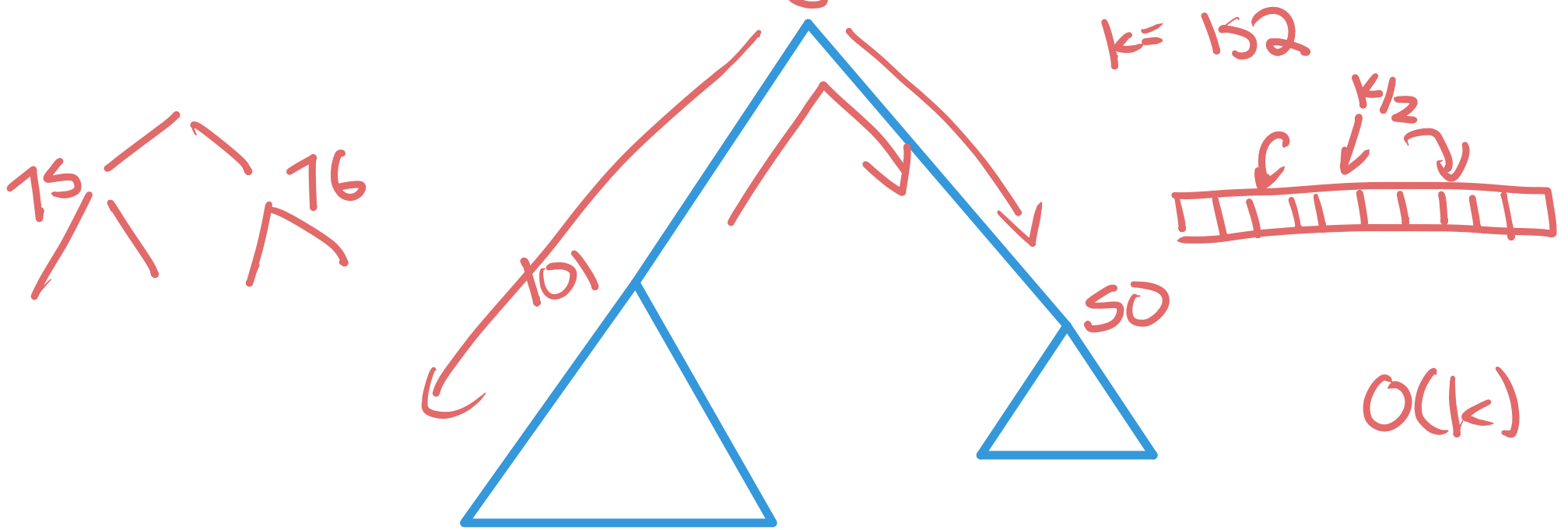
typically covered in
UG data structures

Today we will be very lazy

write down # nodes in each subtree



If we detect big imbalance:

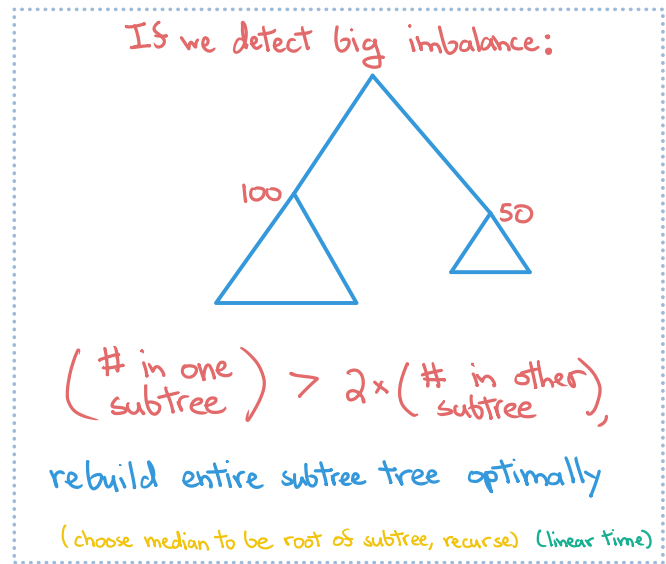
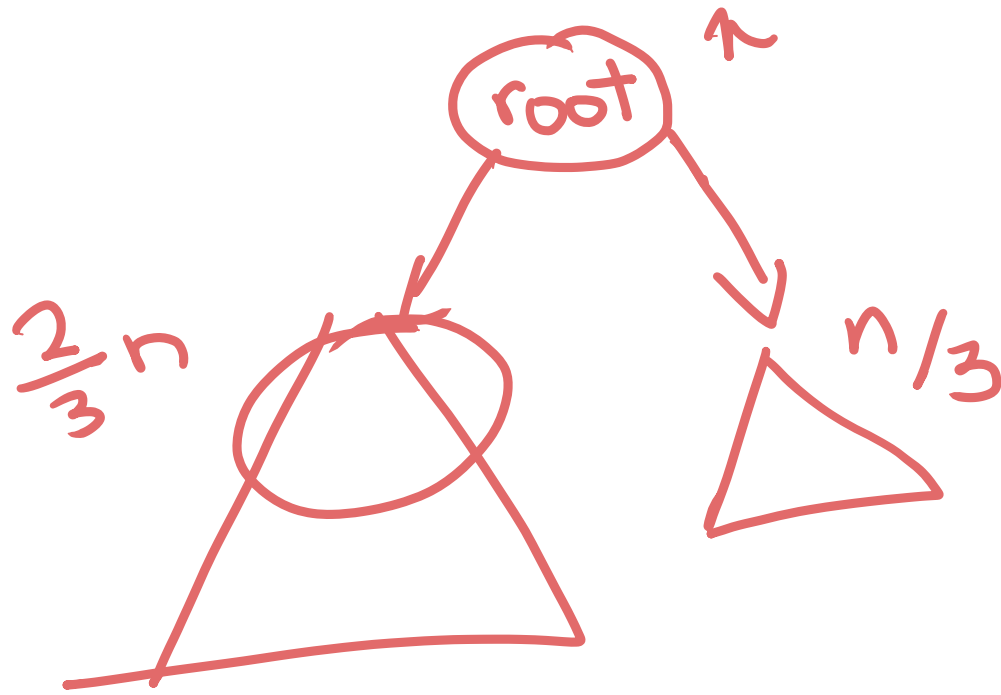


$$(\text{\# in one subtree}) > 2 \times (\text{\# in other subtree}),$$

rebuild entire subtree tree optimally

(choose median to be root of subtree, recurse) (linear time)

Lemma: height is $O(\log n)$
(always)



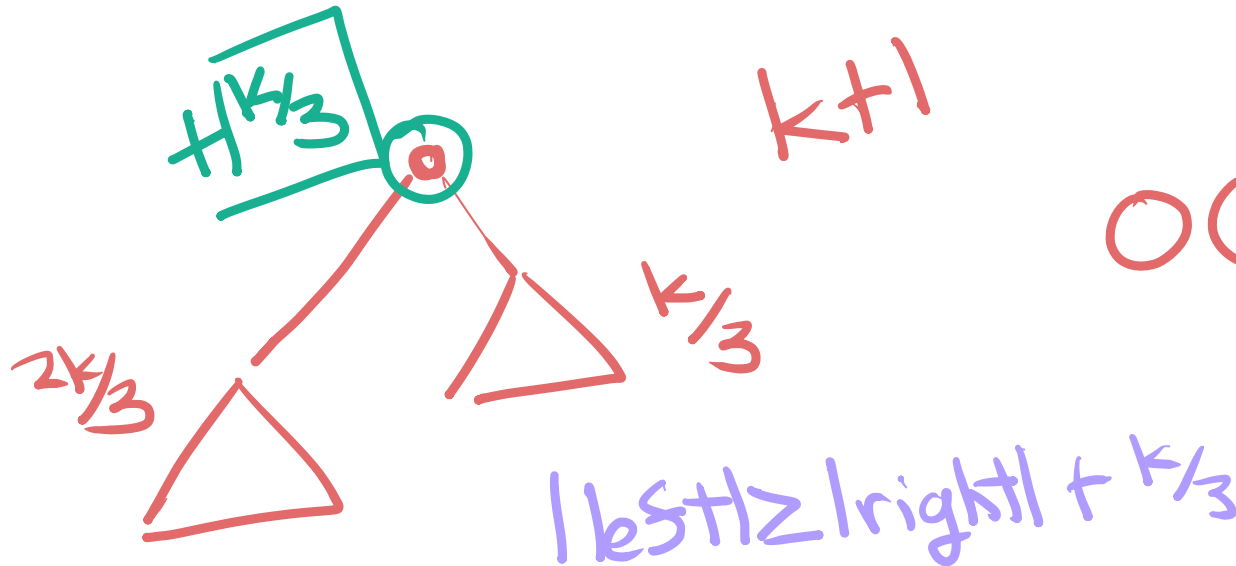
$$\log_{3/2} n \\ = O(\log n)$$

Theorem insertion takes

$O(\log n)$ amortized time

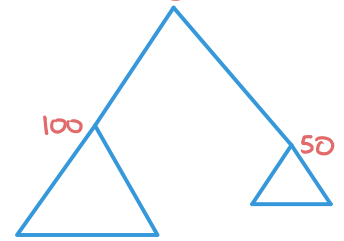
Lemma: height is $O(\log n)$
(always)

Theorem insertion takes
 $O(\log n)$ amortized time



$O(k)$

If we detect big imbalance:



(# in one subtree) $>$ $2 \times$ (# in other subtree),

rebuild entire subtree tree optimally

(choose median to be root of subtree, recurse) (linear time)

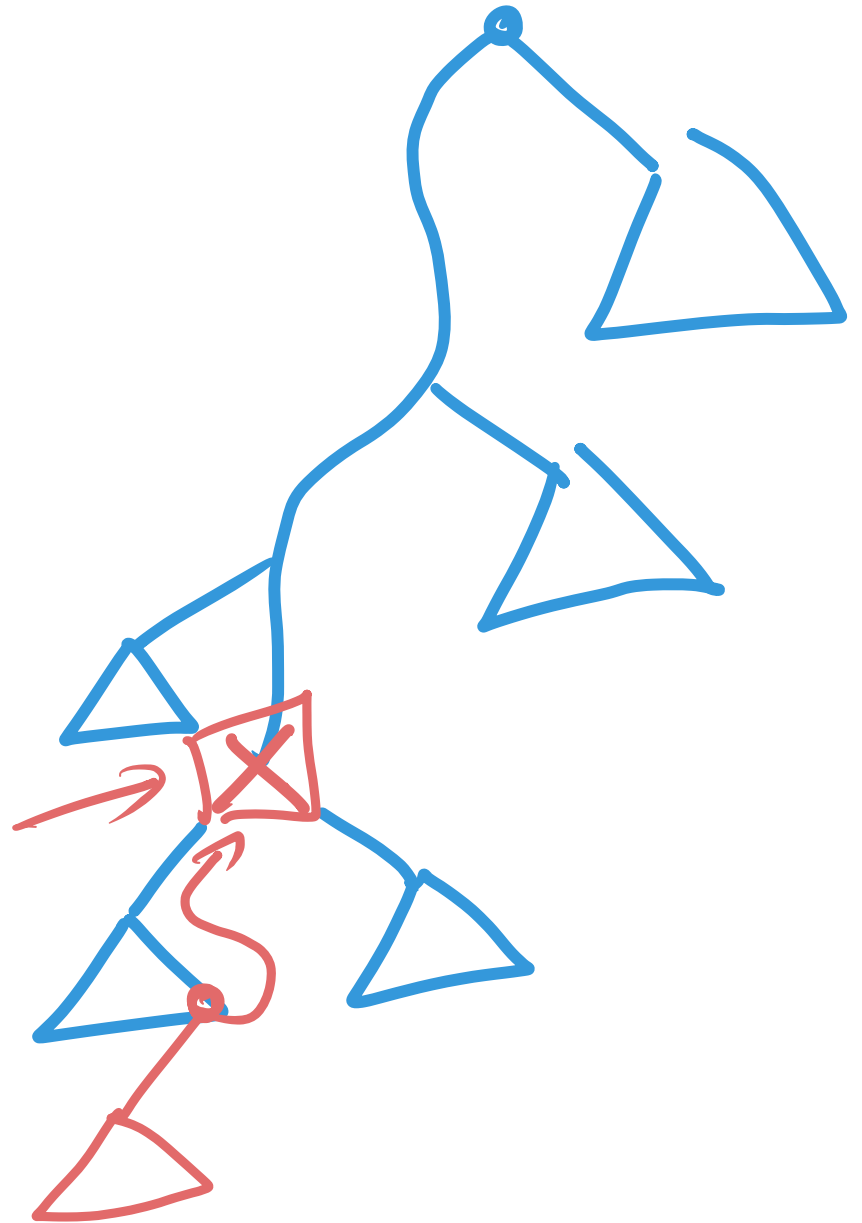
Deletions (remove key from BST)

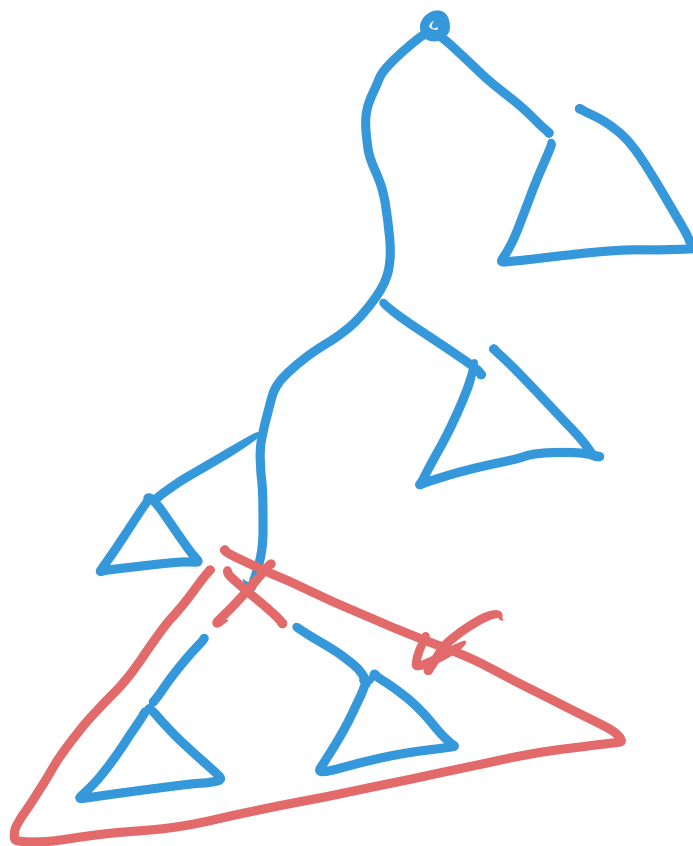
Old-fashioned way:

① search for key

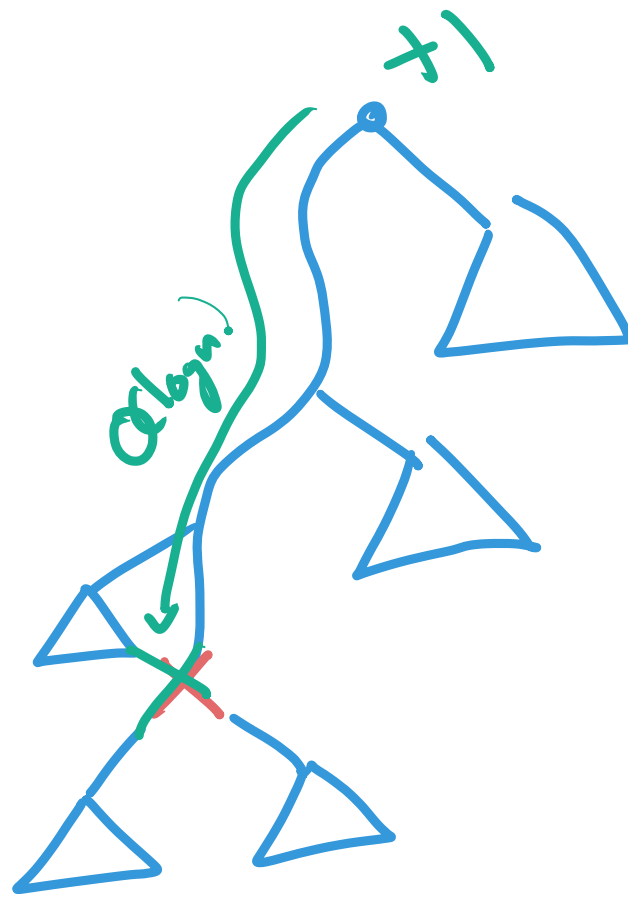
② If key is not a leaf:

chaos



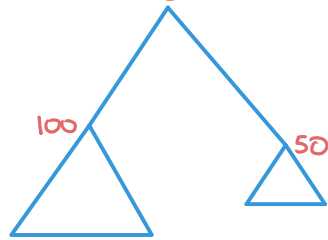


Lazy idea:



Lazy idea: mark node as deleted
if $\geq 50\%$ marked for deletion,
rebuild optimally

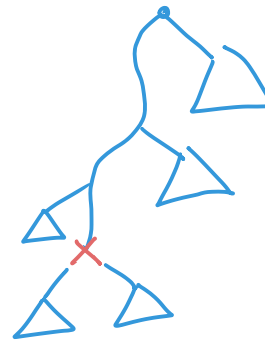
Is we detect big imbalance:



$(\# \text{ in one subtree}) > 2 \times (\# \text{ in other subtree})$,

rebuild entire subtree tree optimally

(choose median to be root of subtree, recurse) (linear time)



Lazy idea: mark node as deleted

if $\geq 50\%$ marked for deletion,

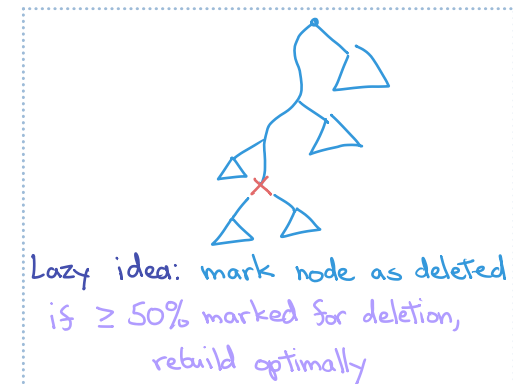
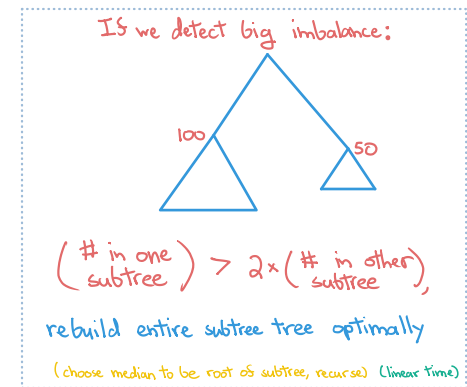
rebuild optimally

Theorem: lazy insertion and lazy deletion take

$O(\log n)$ amortized time

Theorem: lazy insertion and lazy deletion take
 $O(\log n)$ amortized time

Proof



Immutable data structures

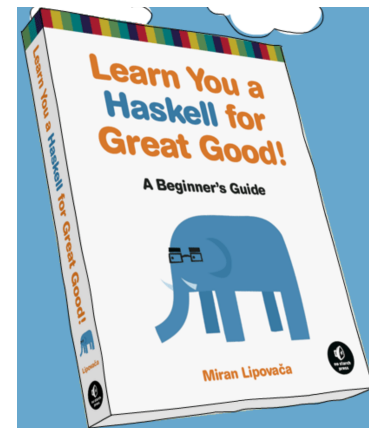
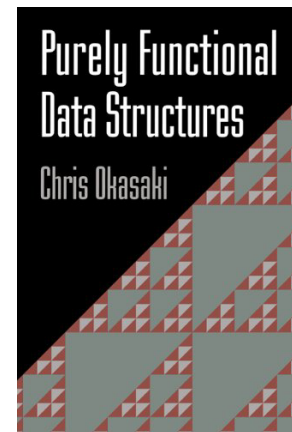
Once something is written, never changes

Restrictions bring benefits:

no side effects

simplifies concurrency

helps compiler



Immutable data structures

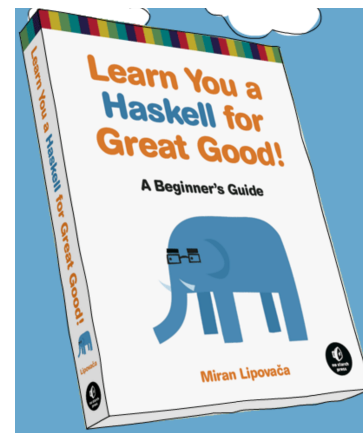
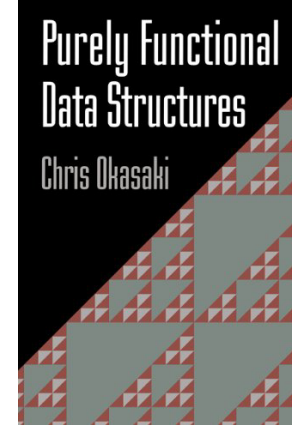
Once something is written, never changes

Restrictions bring benefits:

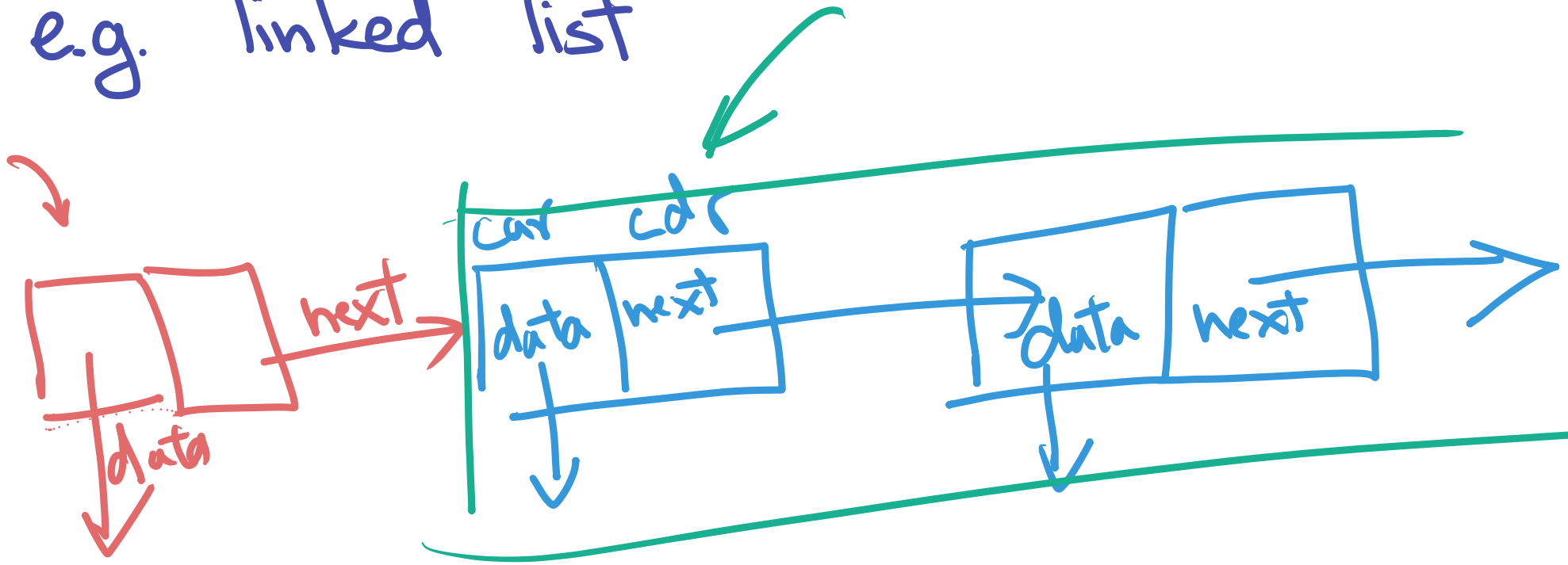
no side effects

simplifies concurrency

helps compiler

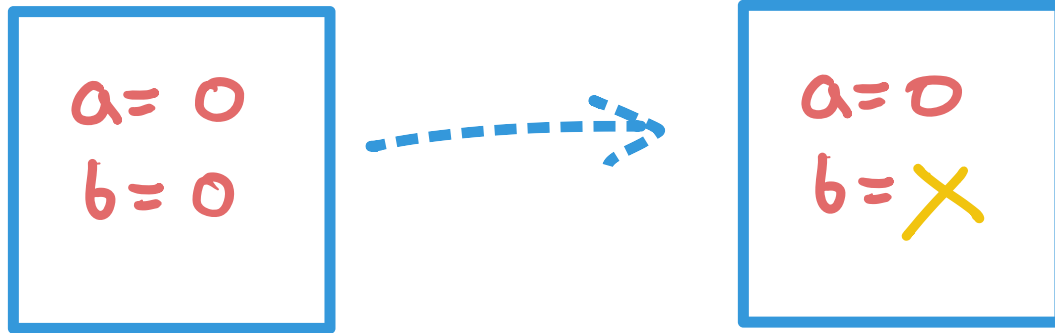


e.g. linked list

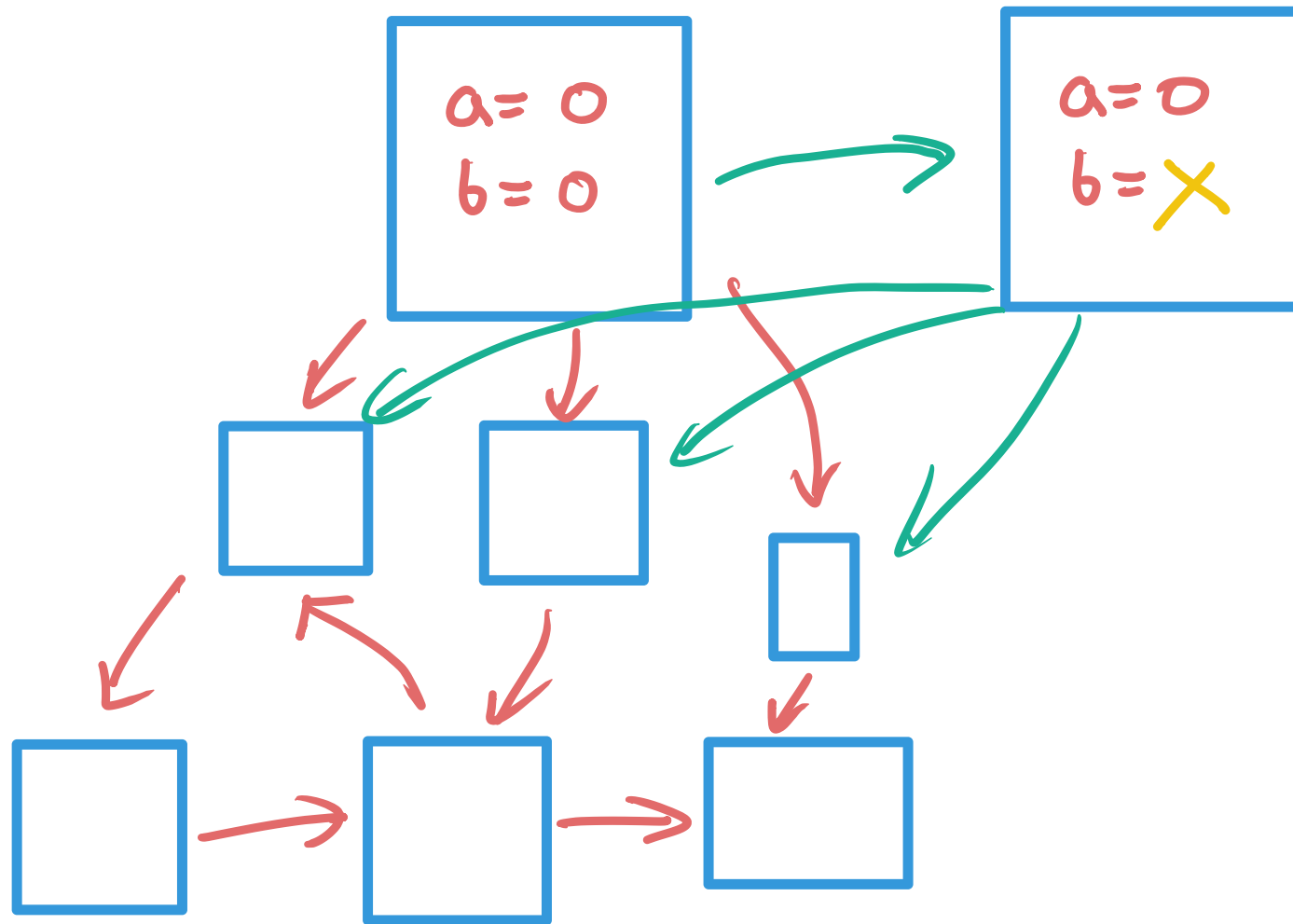


Editing objects by copying

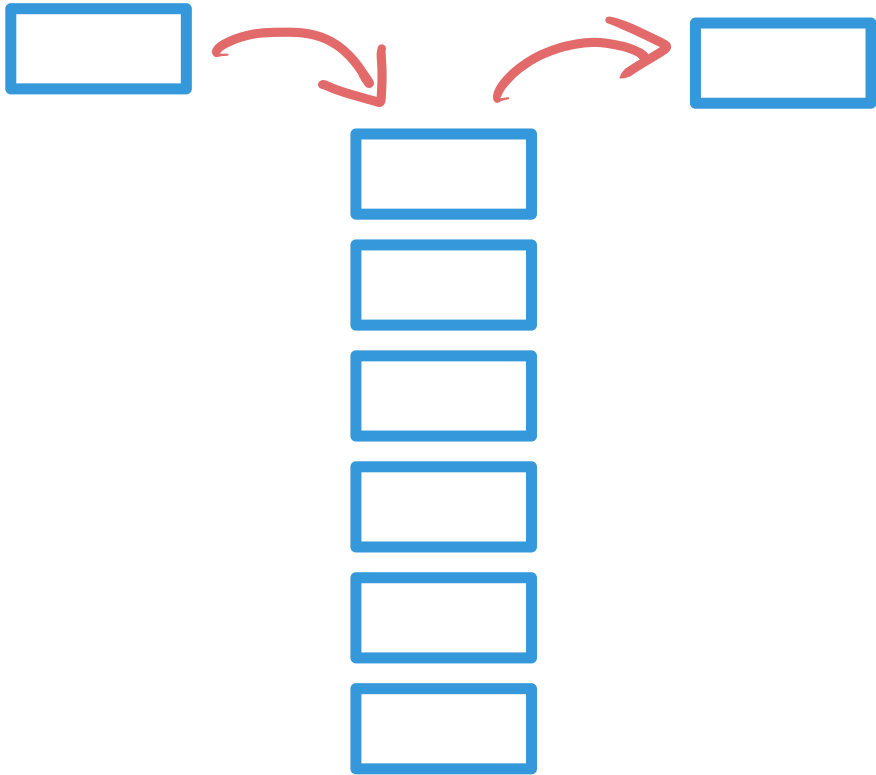
small change $\Rightarrow$ copy whole object



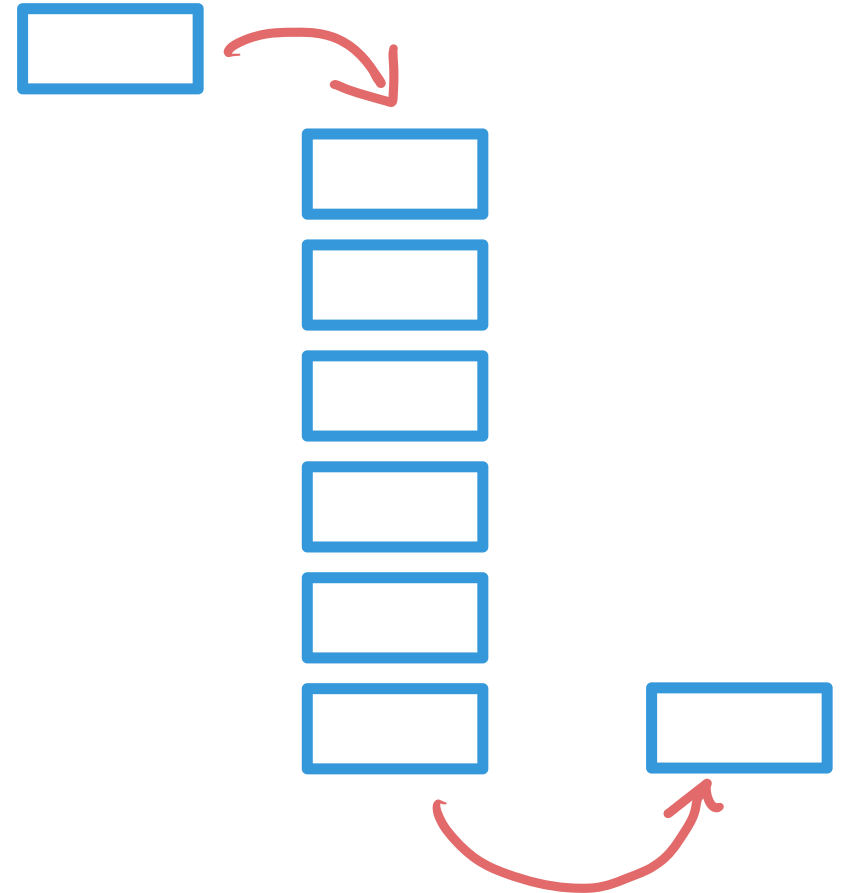
What about "deep" objects?



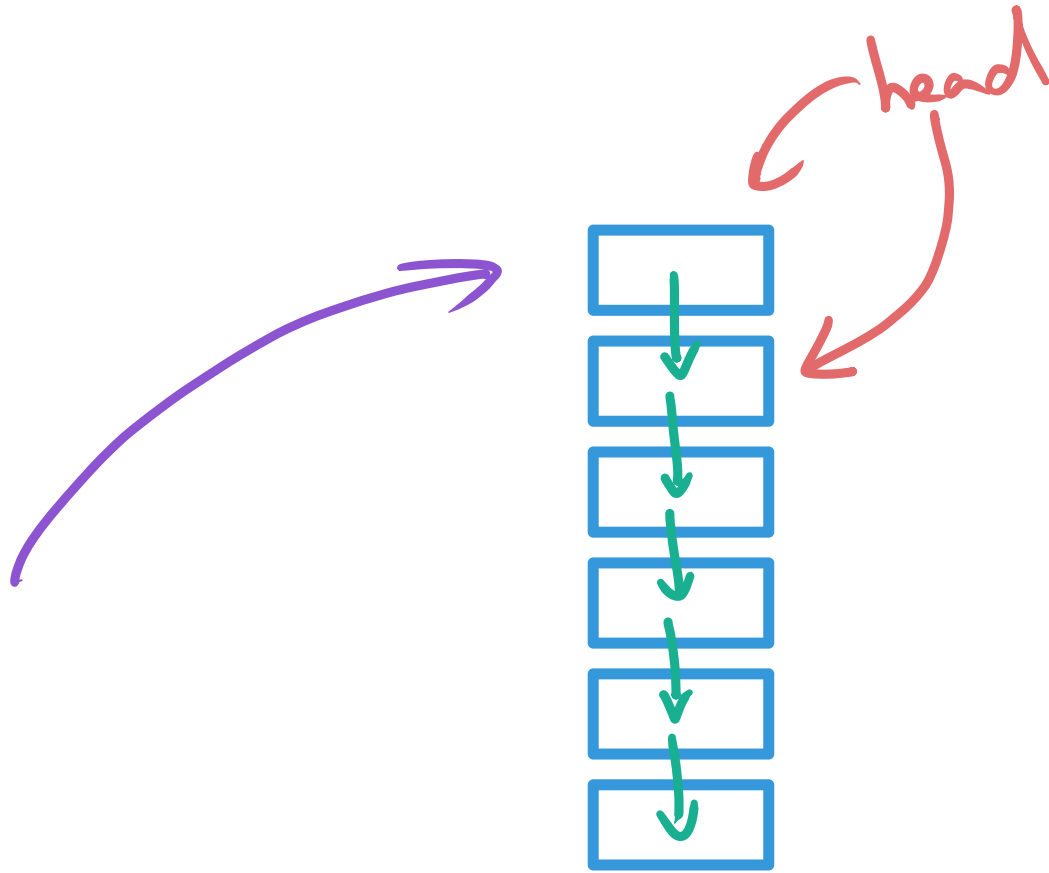
Stacks



Queues

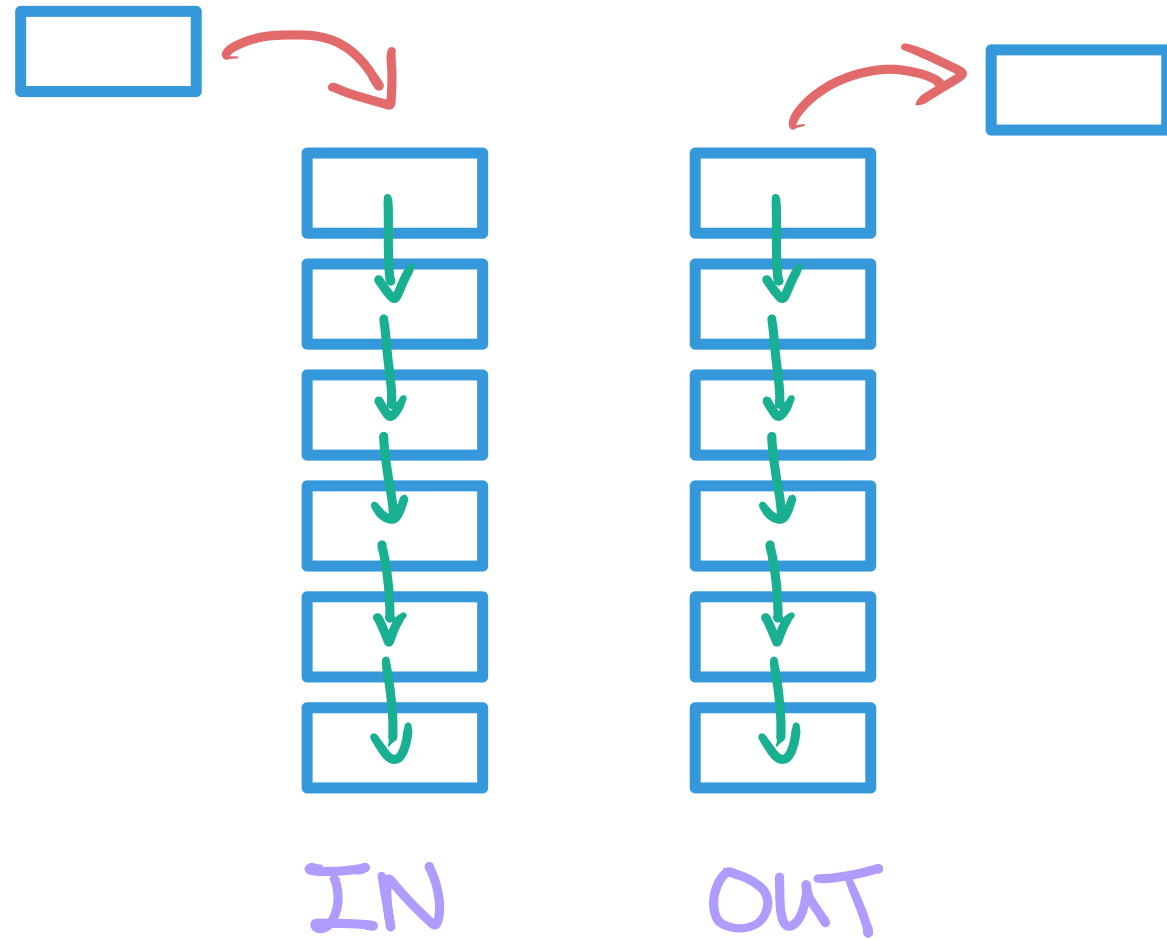


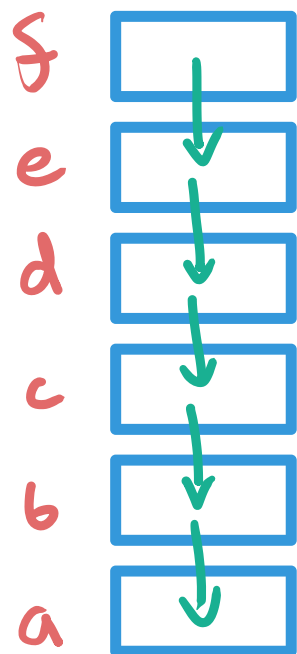
Immutable stacks



(just a linked list)

Immutable Queues

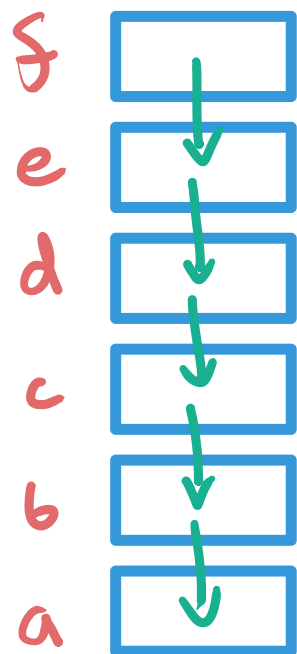




IN



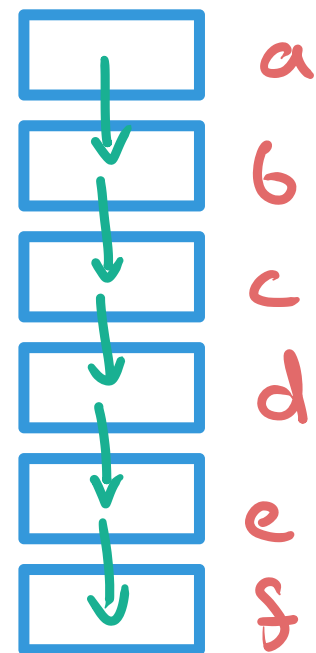
OUT



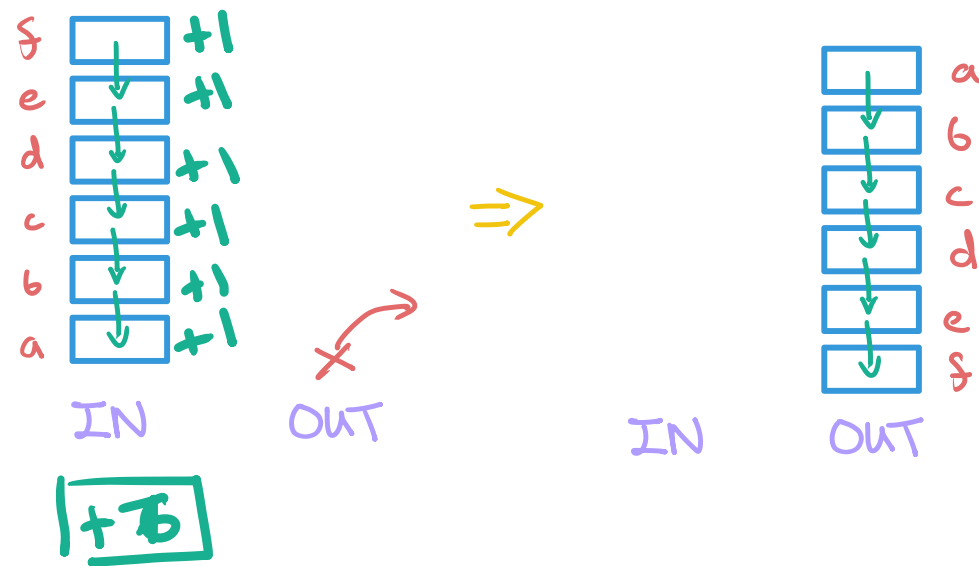
IN



IN



OUT



Theorem: insertion/deletion take $O(1)$ amortized