Lazy Data Structures

This week:

"Average" analysis of algorithms

(without 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, 101, ...

111111111111111111111111111111,

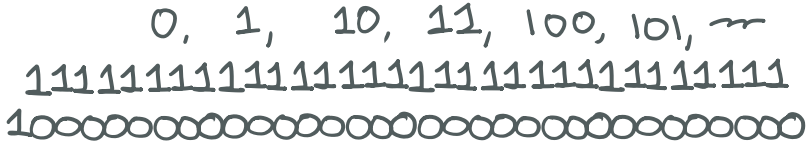
1000000000000000000000000000000

k bits ⇒ k flips (not constant time!)

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

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

0, 1, 10, 11, 100, 101, ⋯



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

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

each increment takes $O(1)$ <u>on average</u>

"$O(1)$ amortized time"

Amortized Analysis

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

"T amortized time"

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

Extends to multiple operations $OP_1, \sim, OP_k$

"OP_1 takes T_1 amortized time, OP_2 takes $\sim$, OP_k takes T_k amortized time"

Ways to do amortized analysis

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

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

"T amortized time"
$\Rightarrow$ n operations take $O(nT)$ time

Extends to multiple operations $OP_1, \dots, OP_k$
"OP_1 takes T_1 amortized time, OP_2 takes $\dots$, OP_k takes T_k amortized time"

Credit schemes

Idea:

Buy credits ahead of time to pay for work later

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

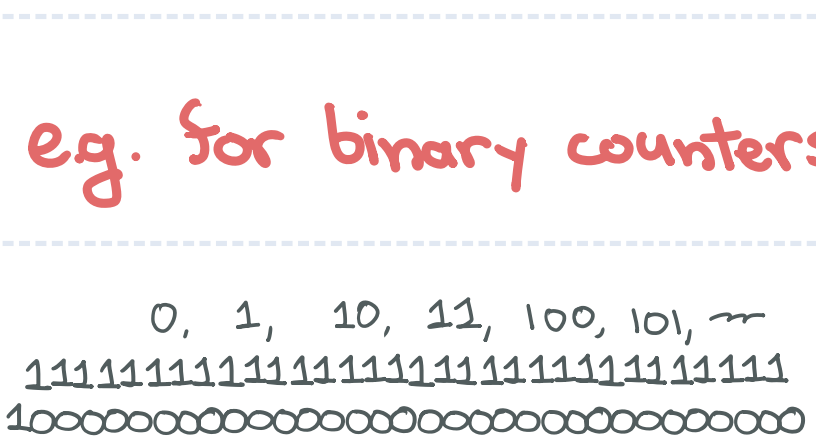
Credit schemes

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

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

eg. for binary counters, when incrementing:

0, 1, 10, 11, 100, 101, ⋯



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

put:
- 1 credit on first bit
- ½ credit on 2nd bit
- ¼ credit on 3rd bit
- ⋮
- + $\frac{1}{2^{i-1}}$ credits on ith bit

---

2 credits total

whenever 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

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

Main rule:

only spend credits that you already saved

e.g. for binary counters, when incrementing:

0, 1, 10, 11, 100, 101, ⋯

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

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

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

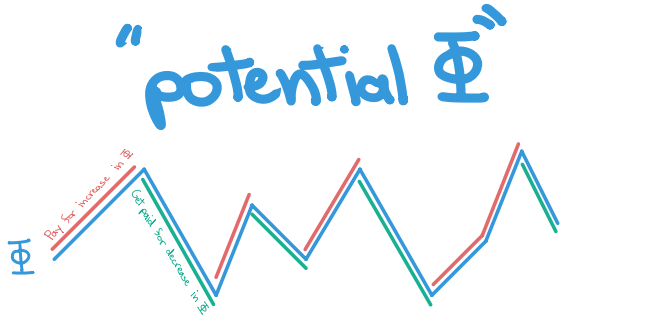
Potential method

Invent "potential Φ"

Amortized cost of operation will be

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

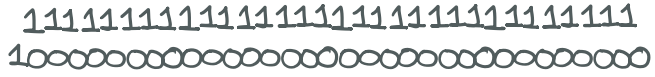
Potential method



Amortized cost of operation will be

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

0, 1, 10, 11, 100, 101, ⋯



e.g. binary counter

Φ = # 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	$\lfloor \frac{n}{4} \rfloor$
⋮		
• kth bit	every 2^{k-1} times	$\lfloor \frac{n}{2^{k-1}} \rfloor$

then for each increment,

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

(each bit reset to 0 paid by $\Delta\Phi$)

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

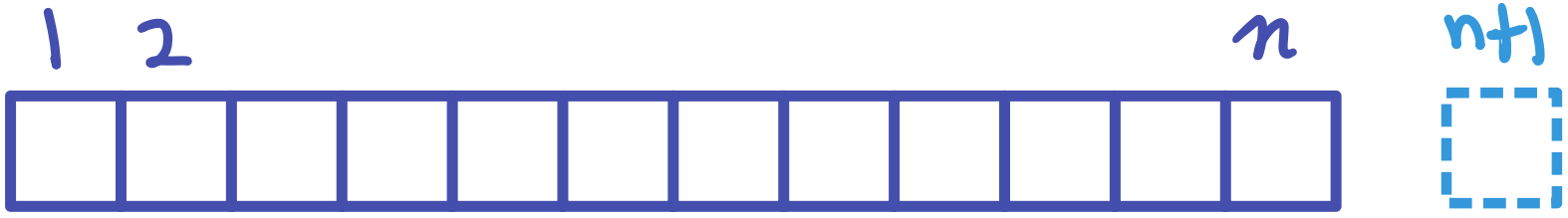
"T amortized time"
$\Rightarrow$ n operations take $O(nT)$ time

Multiple operations $OP_1, \sim, OP_k$
"OP_1 takes T_1 amortized time, OP_2 take $\sim$ OP_k takes T_k amortized time"

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

Φ = # bits set to 1

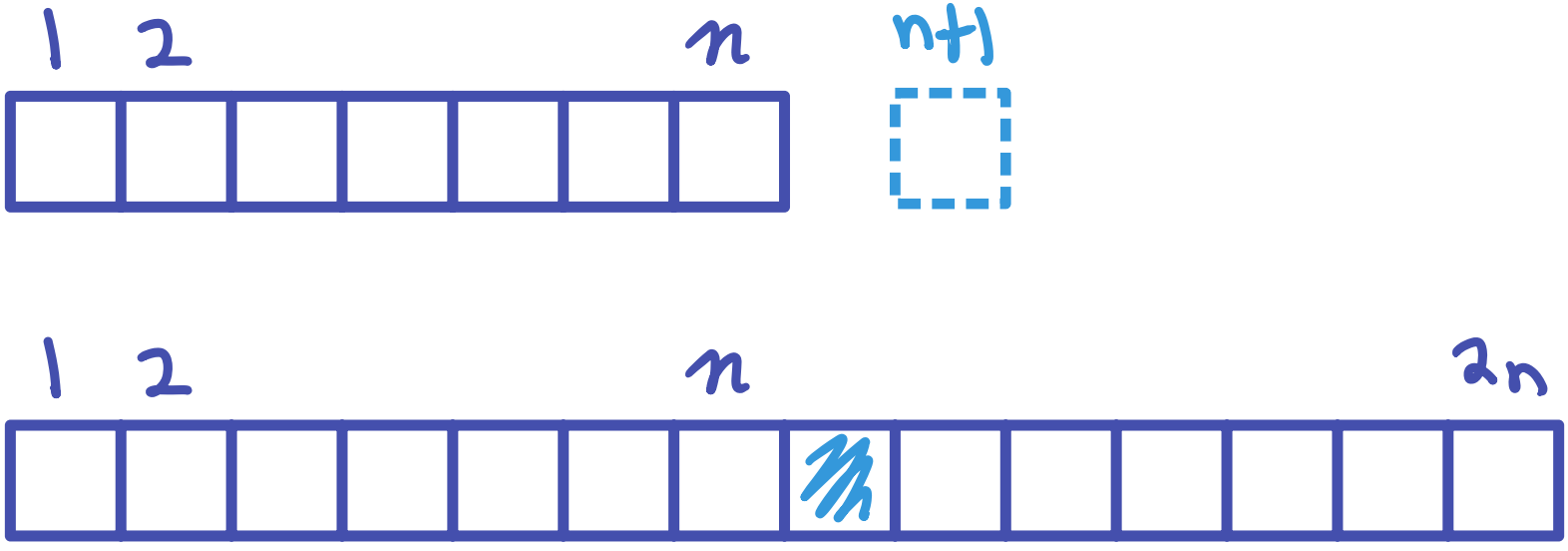
Array-backed lists



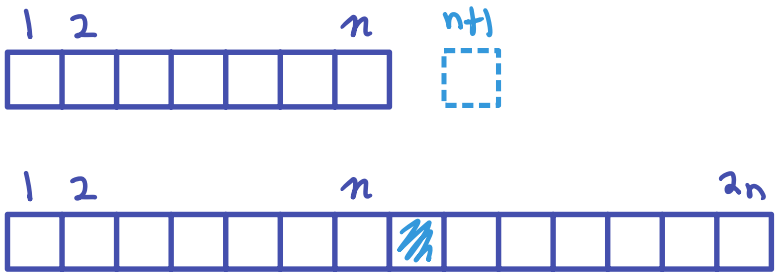
Good: O(1) access by index

Bad: O(n) append

Doubling Trick



Doubling Trick



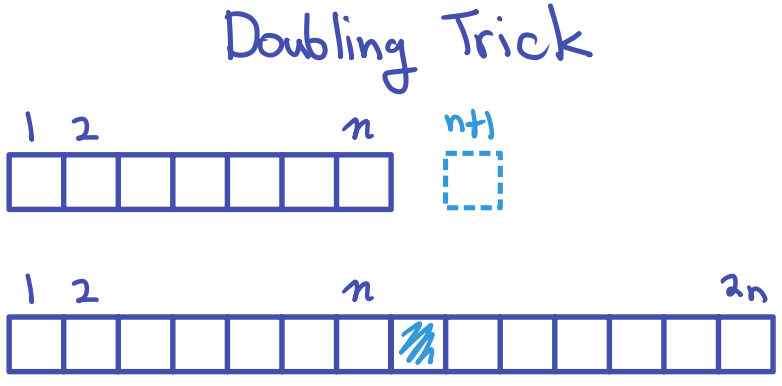
Theorem w/ doubling trick, arraylist has

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

Theorem w/ doubling trick, arraylist has

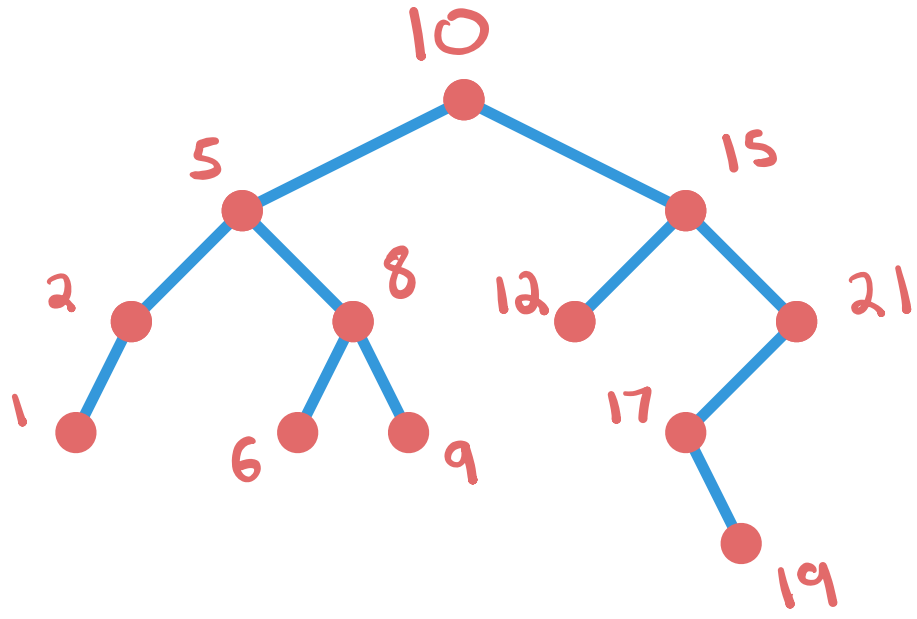
- lookup in O(1)
- append in O(1) amortized

Proof



Search trees

organizes keys in a tree



Invariant:

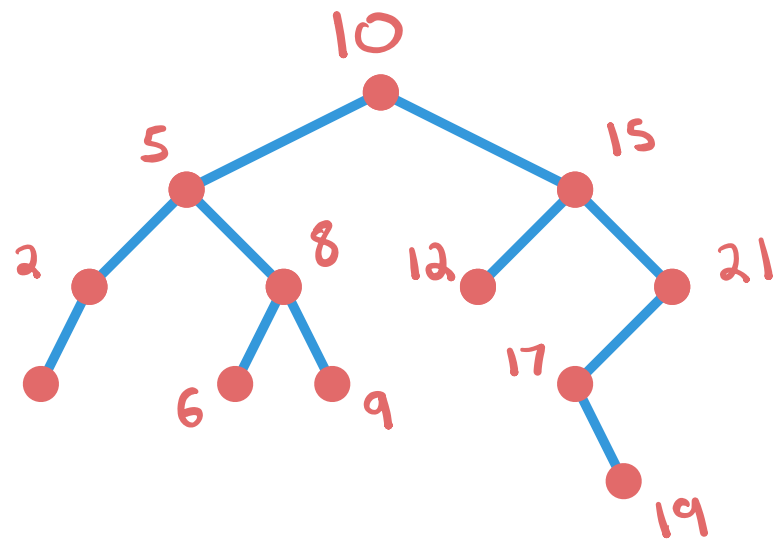
a node's # is

- ≥ any # in left subtree
- ≤ any # in right subtree

Invariant:

a node's # is

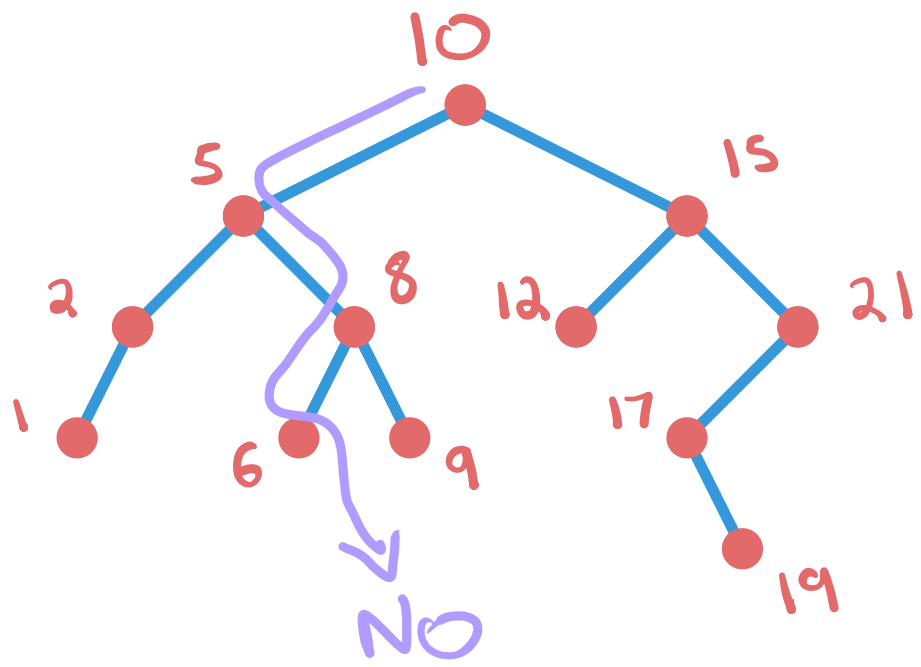
- ≥ any # in left subtree
- ≤ any # in right subtree



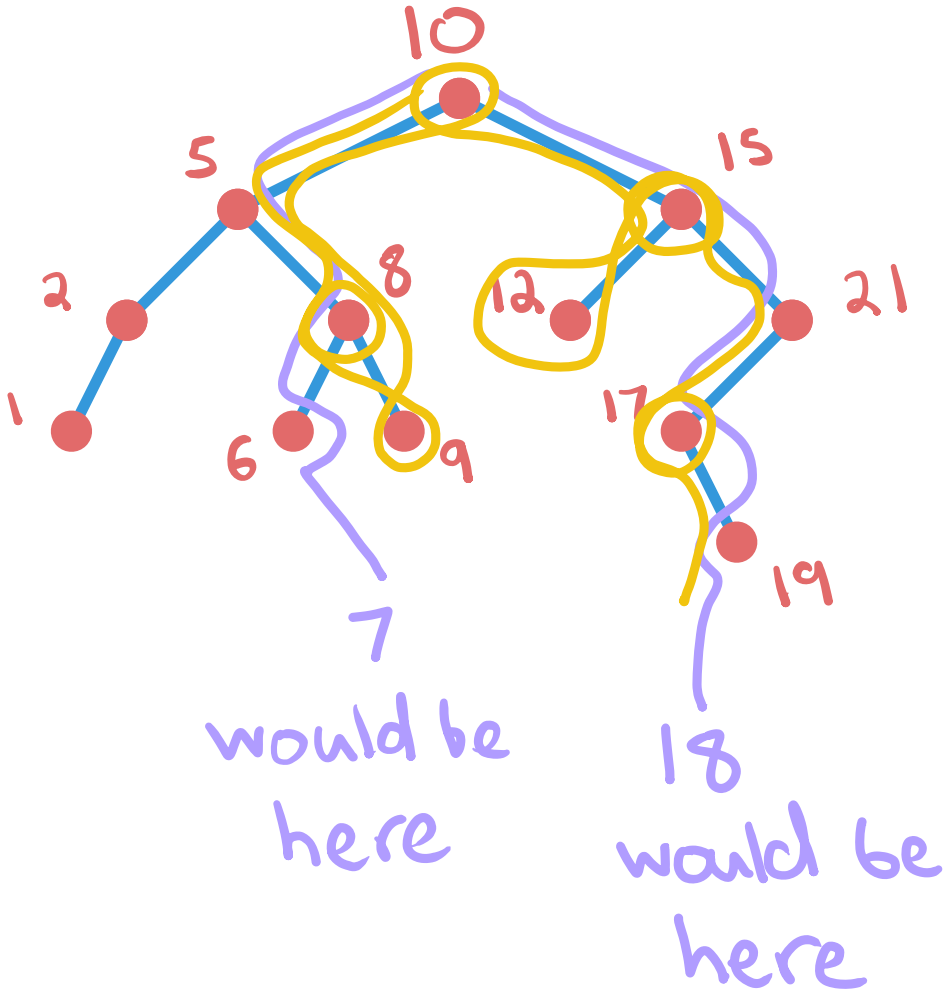
Leads to operations like:

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

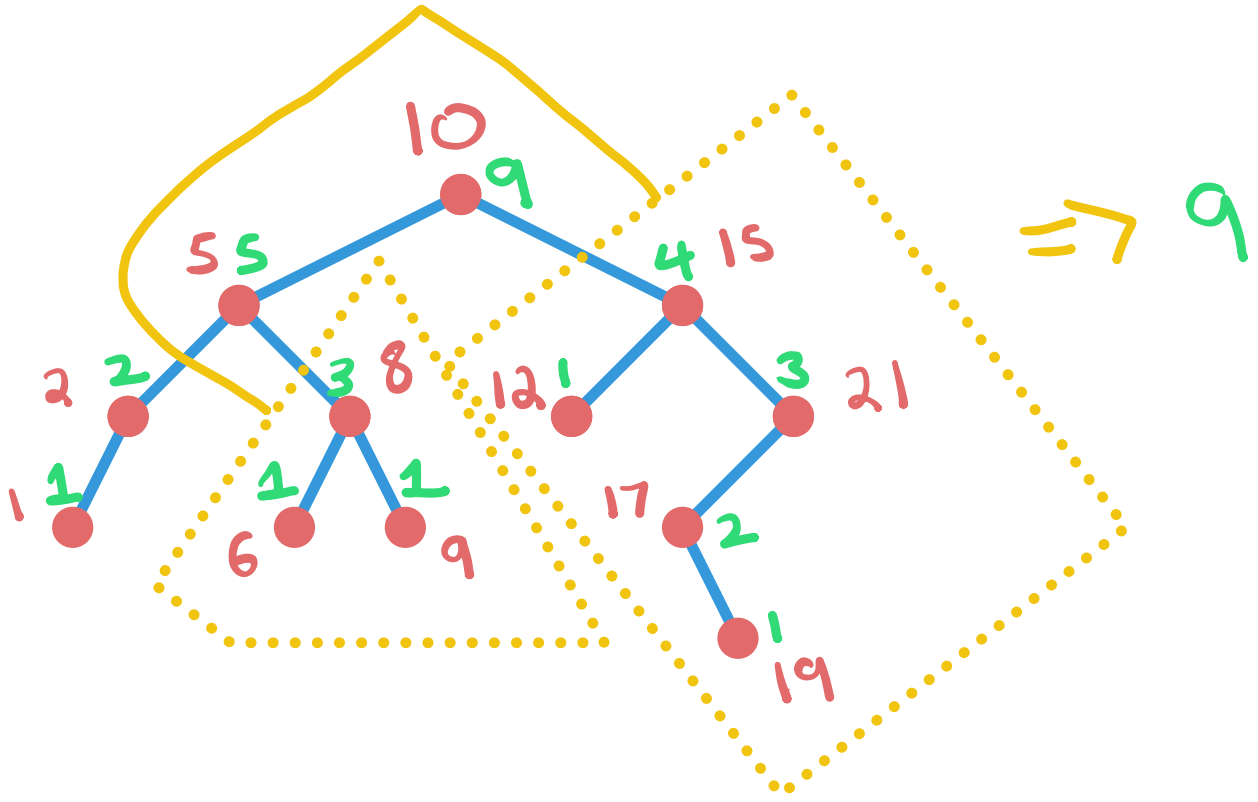
does the tree contain 7?



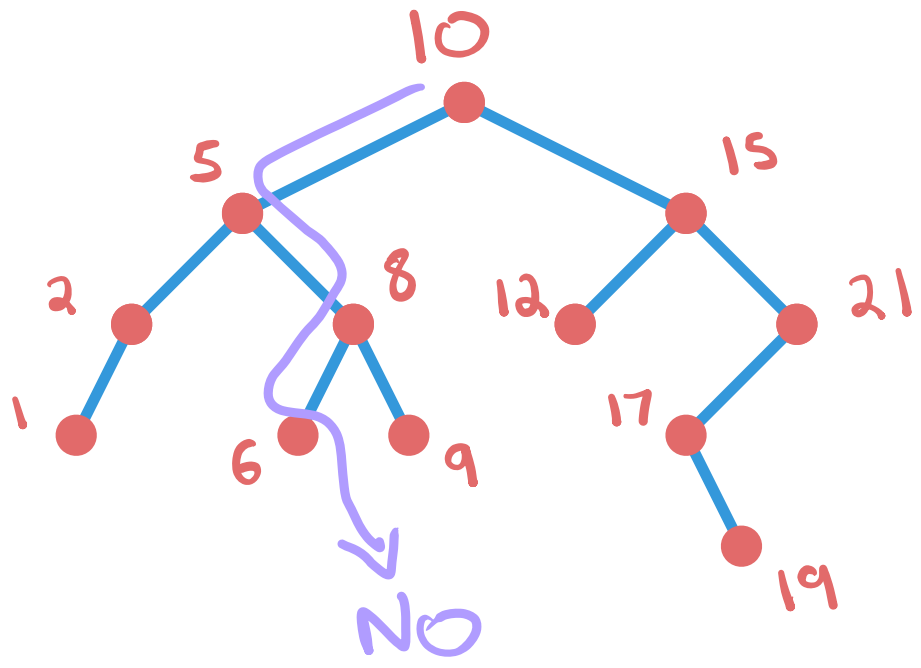
list all keys between 7 and 18



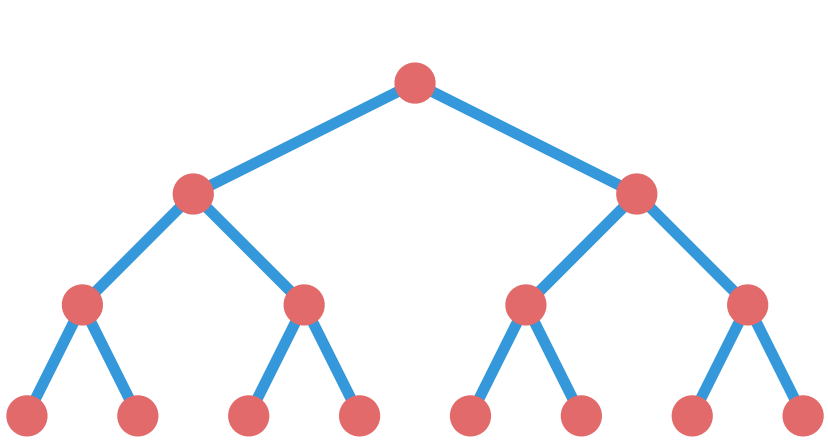
how many keys are between 5 and 25

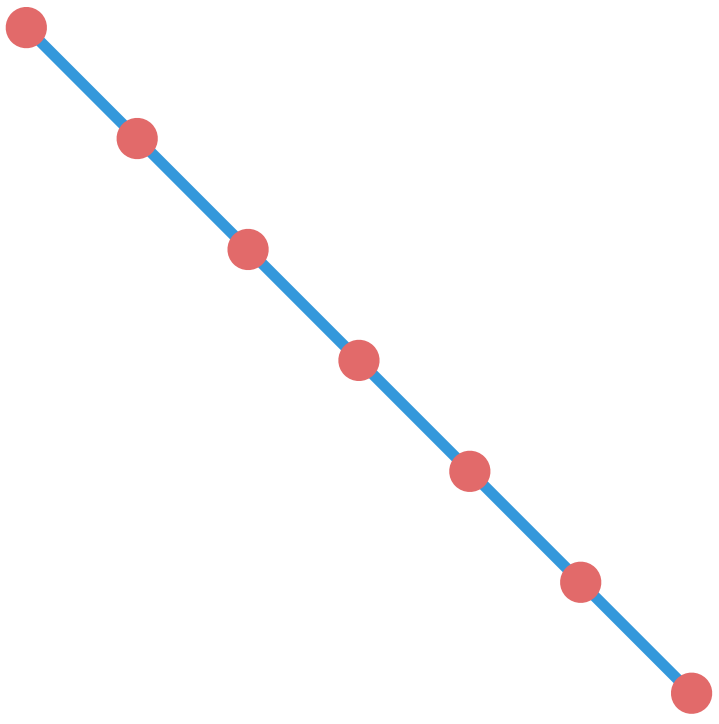


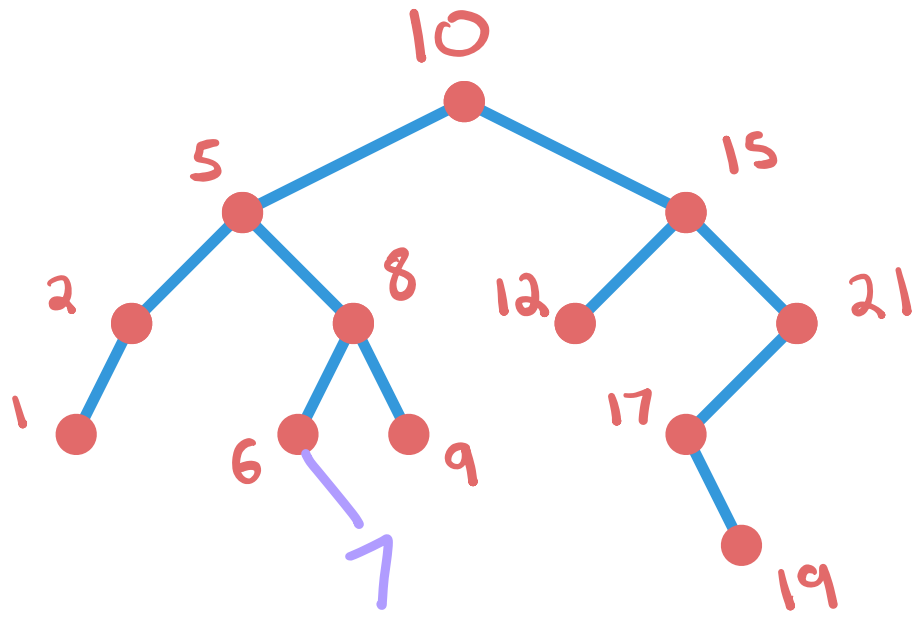
if we write down # nodes in each subtree, then faster



operations take time proportional to the height of a tree

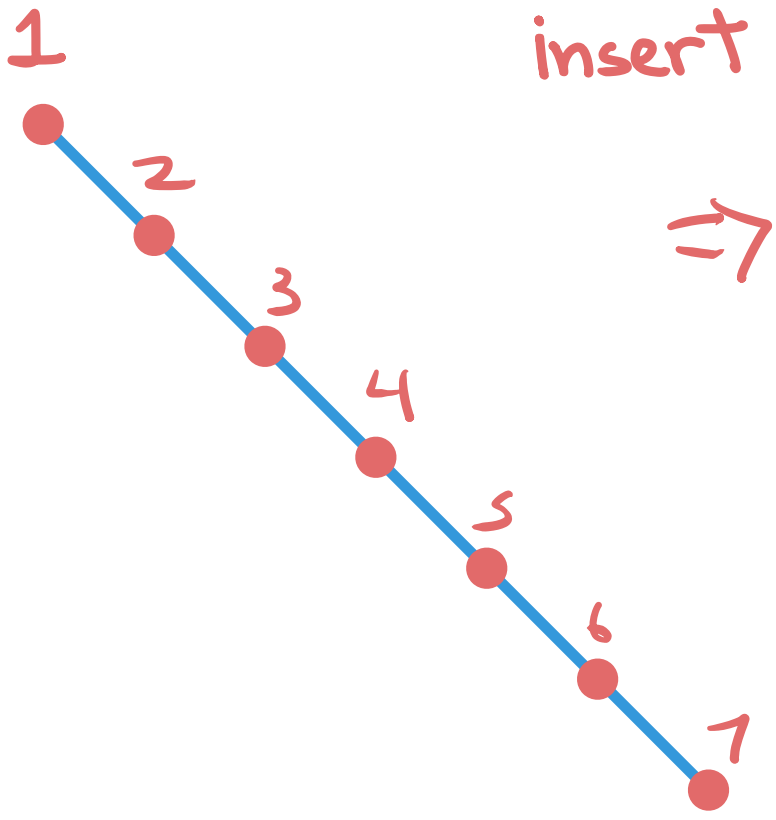






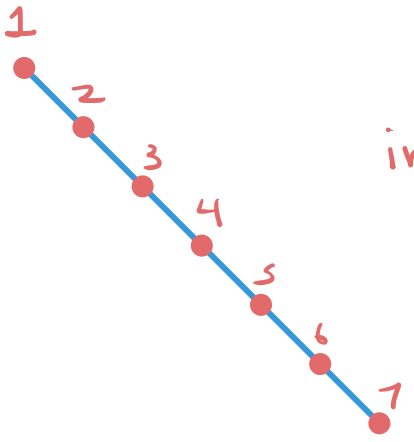
Most natural method for insertion is as leaves

(search for where the key would be, and put it there)



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

⇒ imbalanced



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

⇒ imbalanced

How to keep a tree balanced?

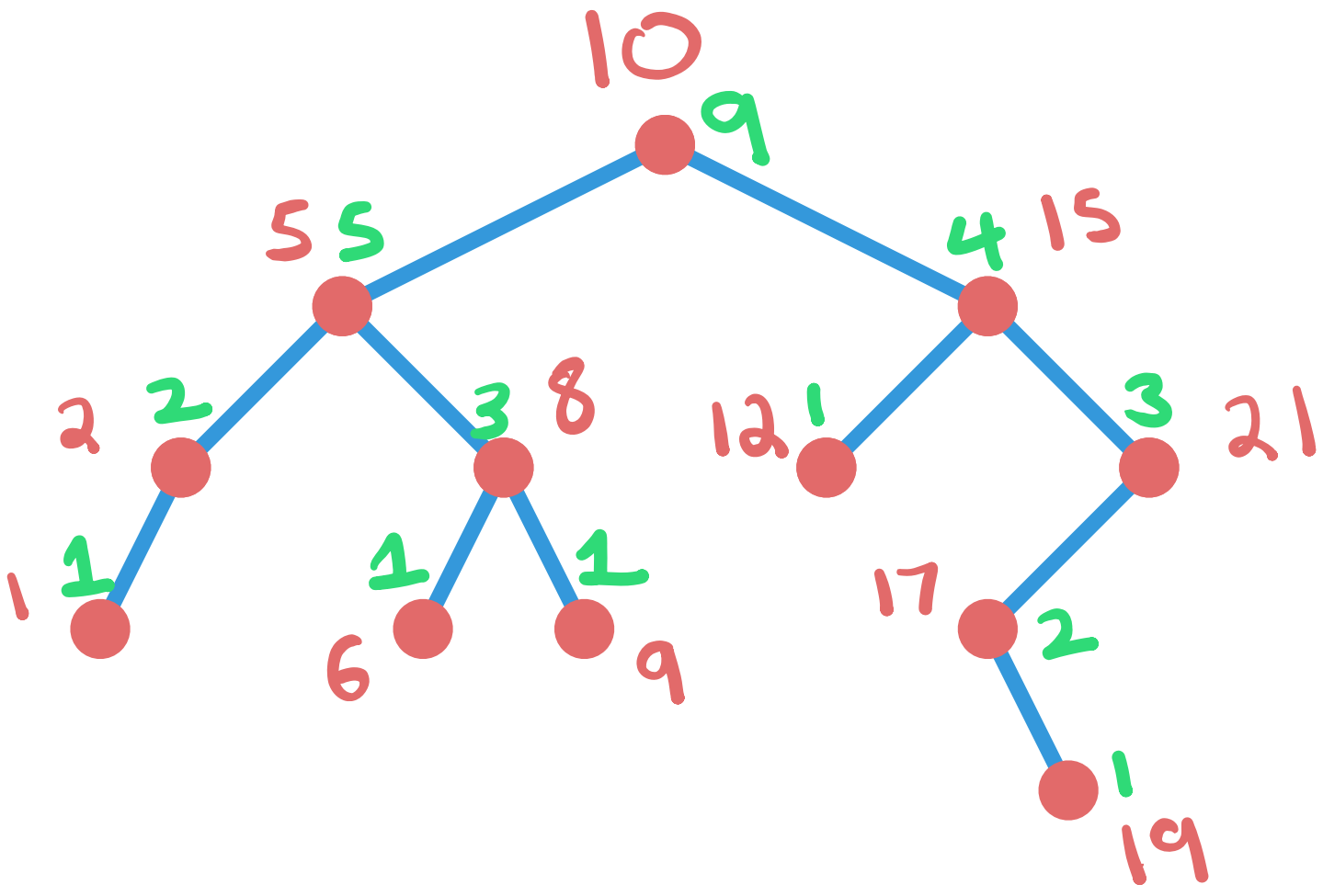
AVL trees

Red-black trees

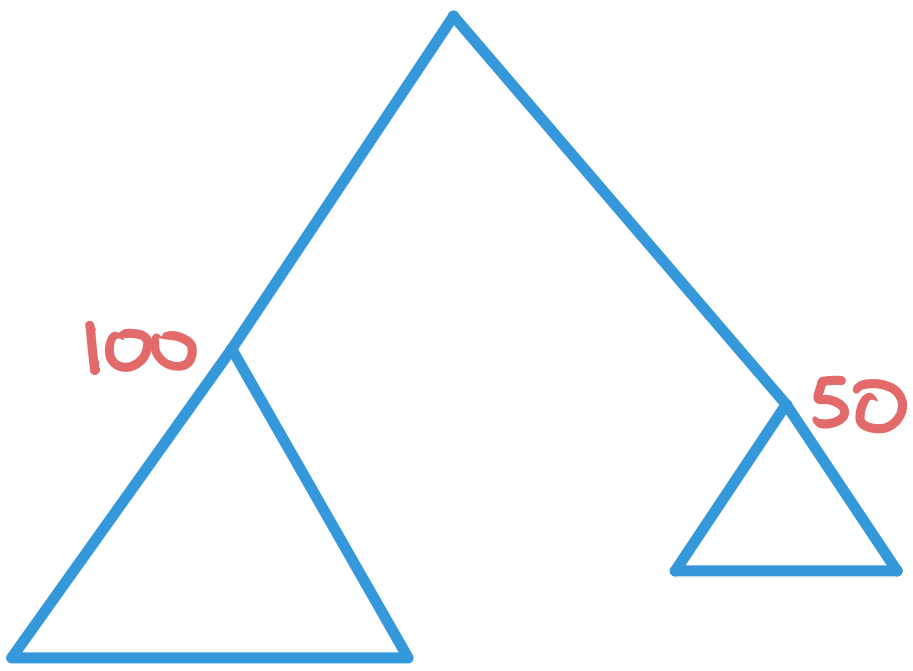
typically covered in UG data structures

Today we will be very lazy

write down # nodes in each subtree



If we detect big imbalance:



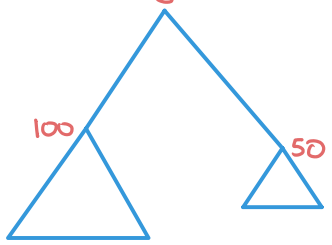
$$\left(\begin{array}{c}\#\text{ in one} \\ \text{subtree}\end{array}\right) > 2 \times \left(\begin{array}{c}\#\text{ in other} \\ \text{subtree}\end{array}\right),$$

rebuild entire subtree tree optimally

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

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

If we detect big imbalance:



$$\left(\begin{array}{c}\#\text{ in one}\\ \text{subtree}\end{array}\right) > 2\times\left(\begin{array}{c}\#\text{ in other}\\ \text{subtree}\end{array}\right),$$

rebuild entire subtree tree optimally

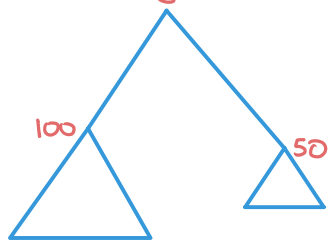
(choose median to be root of subtree, recurse) (linear time)

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

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

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

If we detect big imbalance:



(# in one subtree) > 2 × (# in other subtree),

rebuild entire subtree tree optimally

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

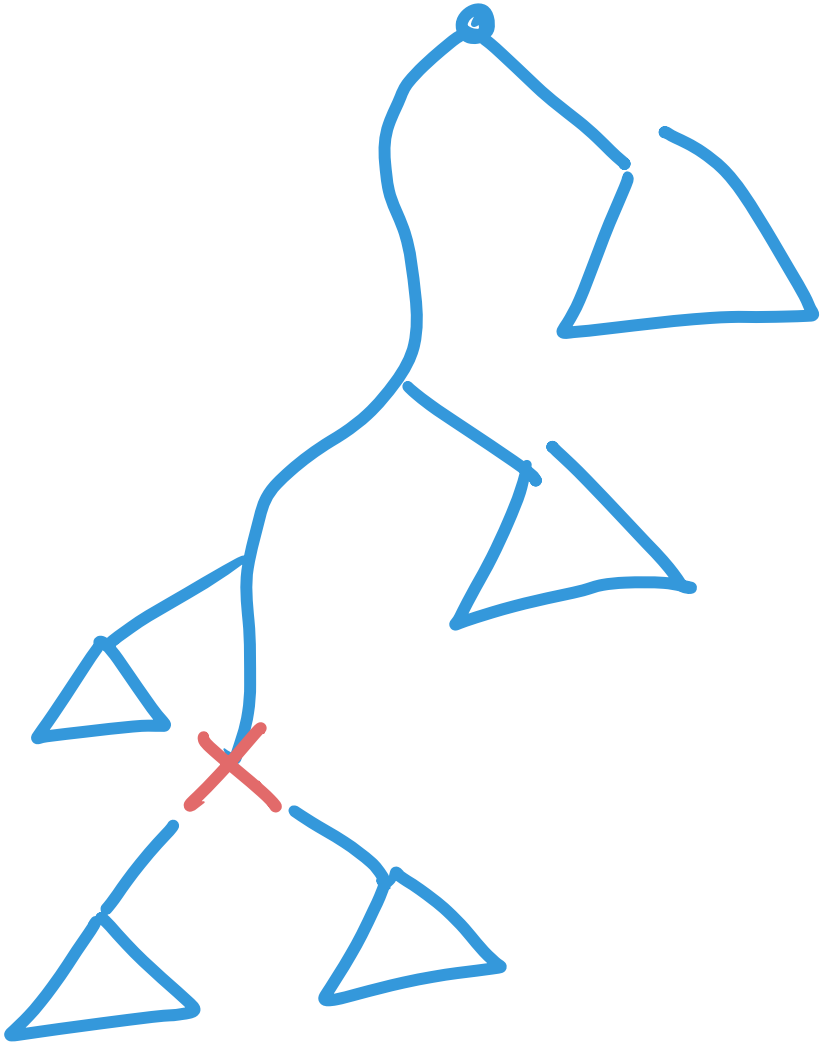
Deletions (remove key from BST)

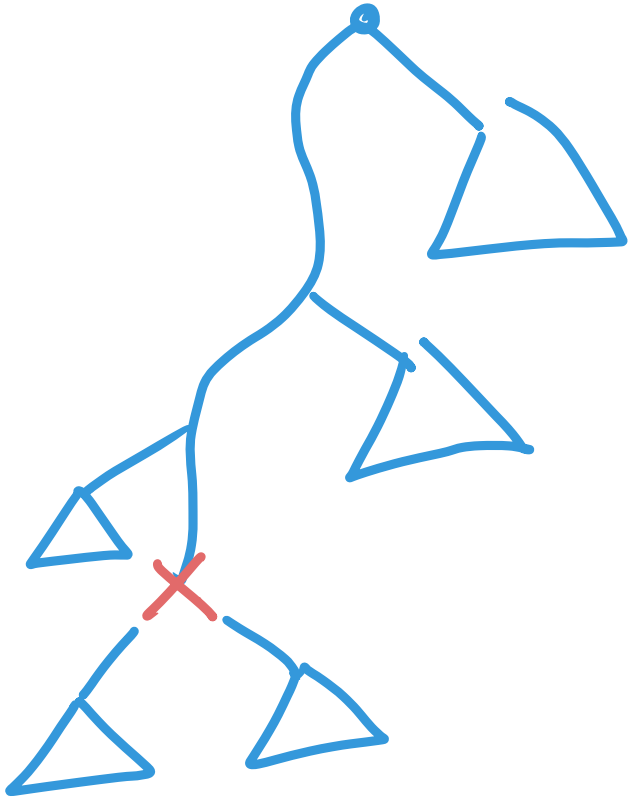
Old-fashioned way:

① search for key

② If key is not a leaf:

chaos

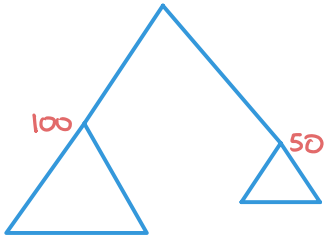




Lazy idea: mark node as deleted

if ≥ 50% marked for deletion,
rebuild optimally

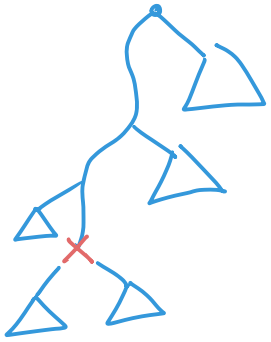
If we detect big imbalance:



(# in one subtree) > 2 × (# 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 ≥ 50% marked for deletion,

rebuild optimally

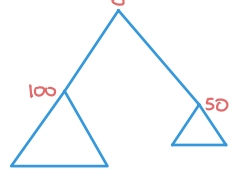
Theorem: lazy insertion and lazy deletion take

$O(\log n)$ amortized time

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

Proof

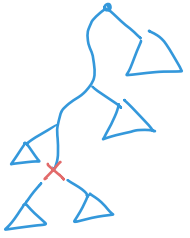
If we detect big imbalance:



(# in one subtree) > 2 × (# in other subtree),

rebuild entire subtree tree optimally

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



Lazy idea: mark node as deleted

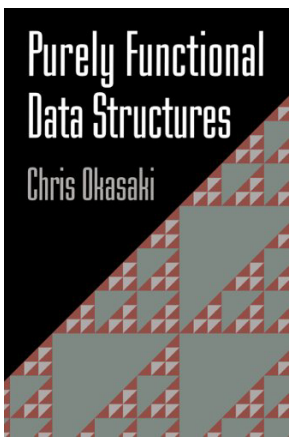
if ≥ 50% marked for deletion, rebuild optimally

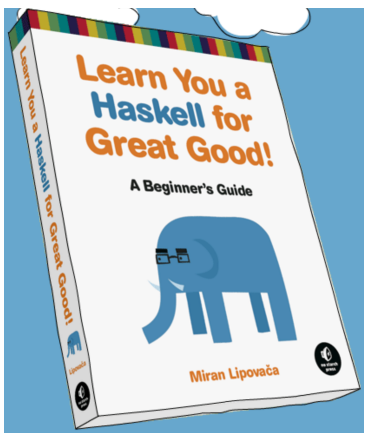
Immutable data structures

Once something is written, never changes

Restrictions bring benefits:

- no side effects
- simplifies concurrency
- helps compiler





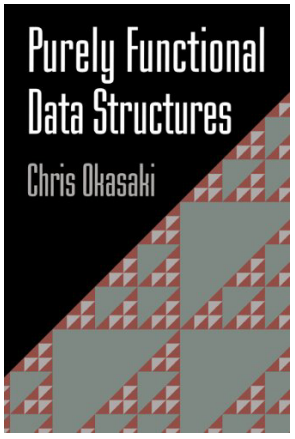
Immutable data structures

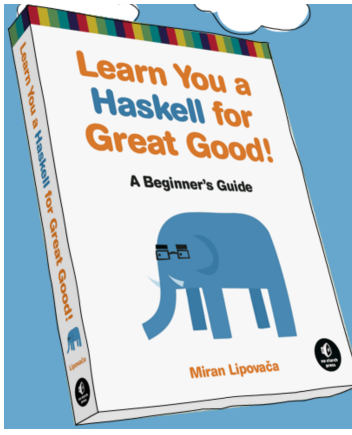
Once something is written, never changes

Restrictions bring benefits:

- no side effects
- simplifies concurrency
- helps compiler

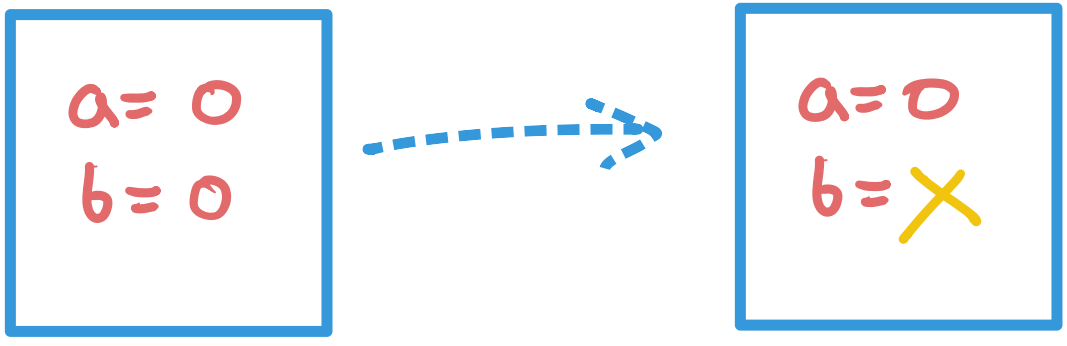
e.g. linked list



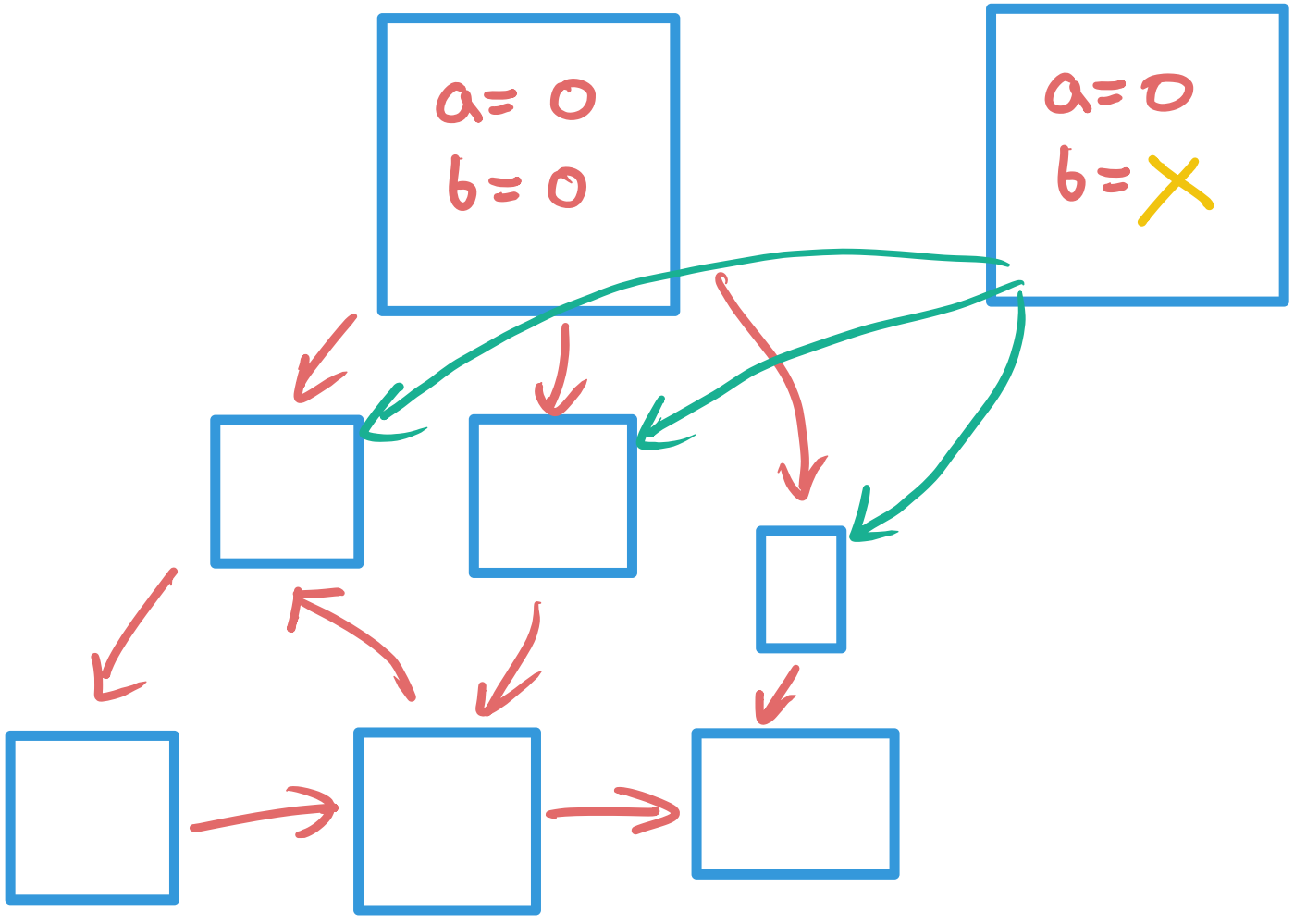


Editing objects by copying

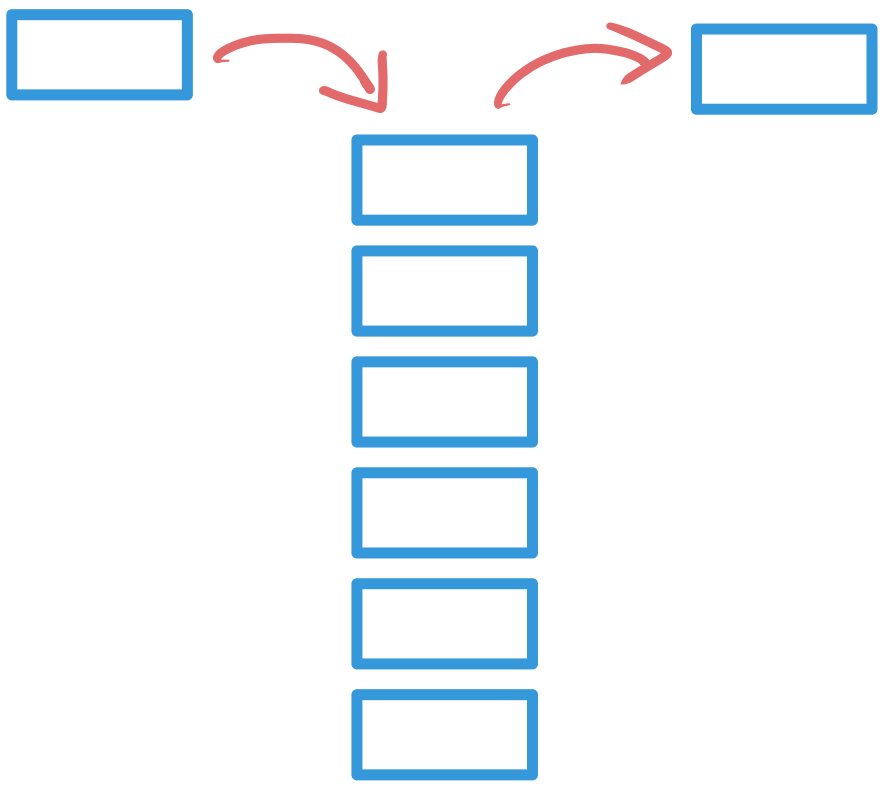
small change ⇒ copy whole object



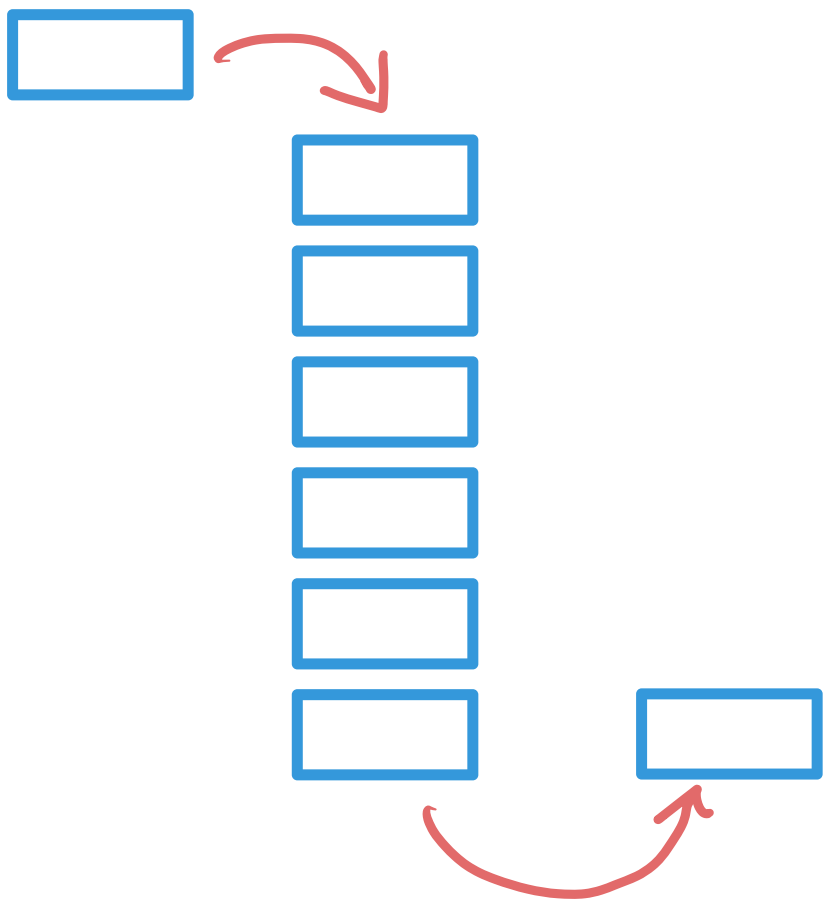
What about "deep" objects?



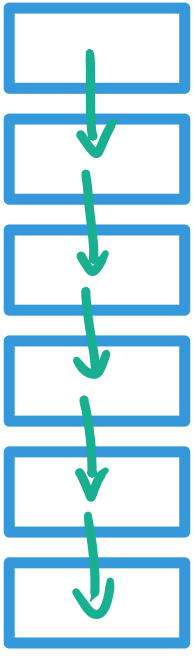
Stacks



Queues

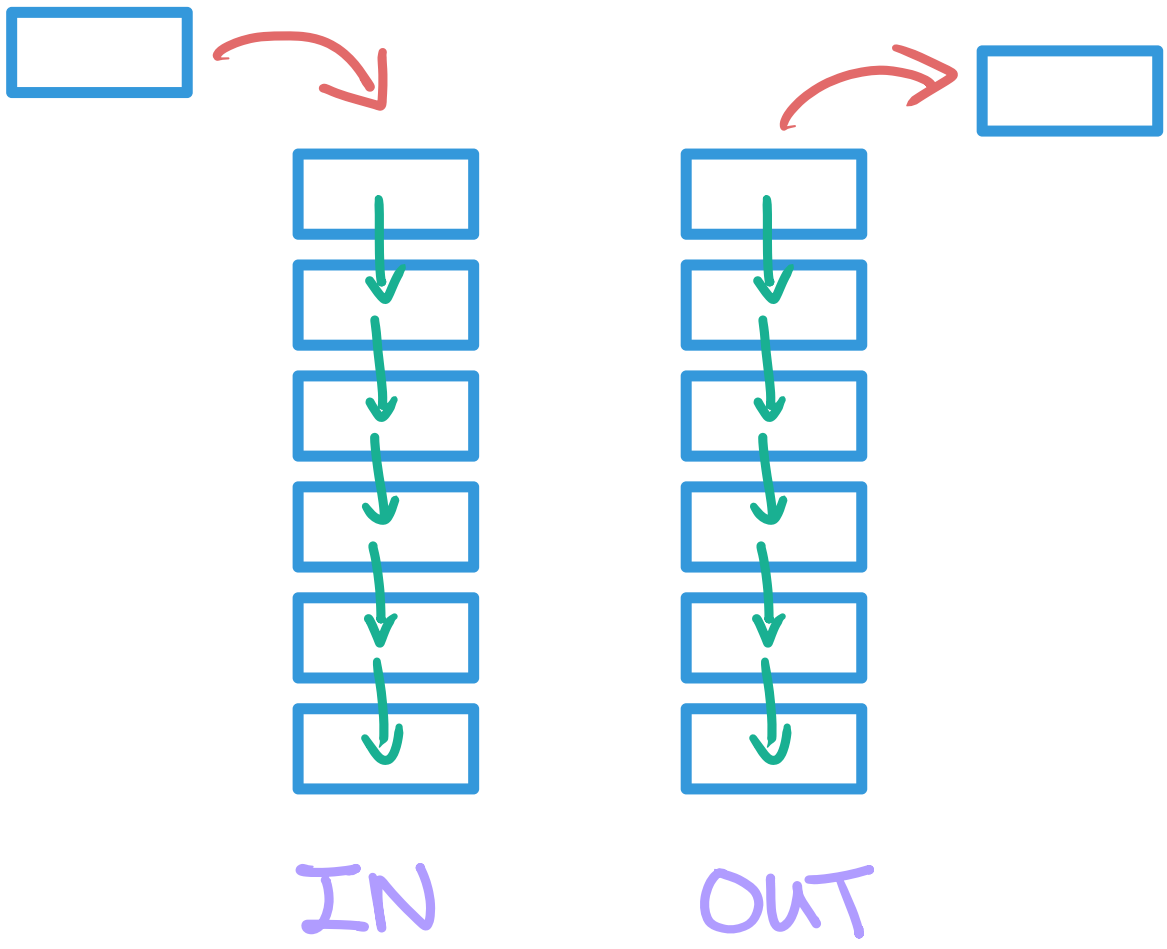


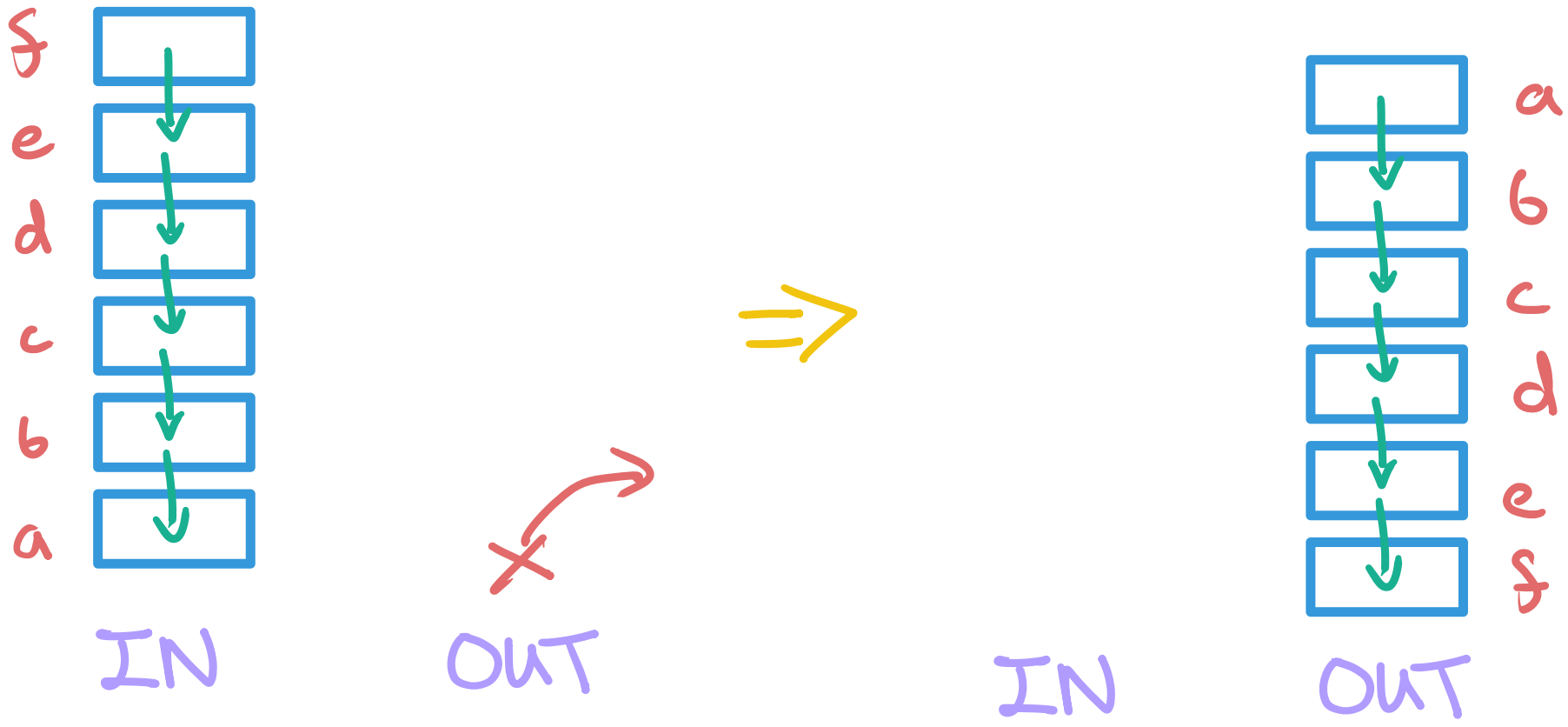
Immutable stacks

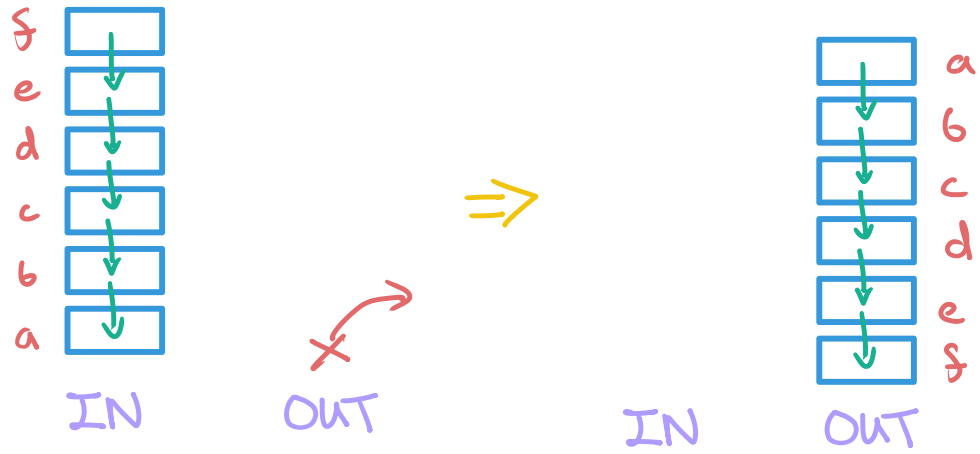


(just a linked list)

Immutable Queues




f
e
d
c
b
a
IN
OUT
IN
OUT
a
b
c
d
e
f



Theorem: insertion/deletion take O(1) amortized