Lazy Data Structures

"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, 110, ...

Running time of i++?

Incrementing Binary Counters

```
i++;          //increment i
```

In bits: 0, 1, 10, 11, 100, 101, ...

111111111111111111111111111111,

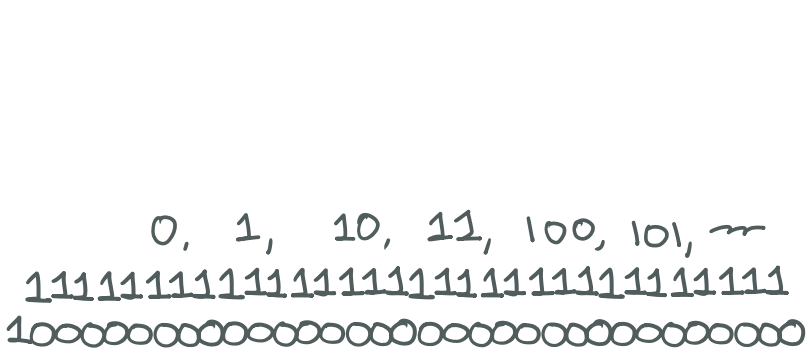
1000000000000000000000000000000

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

"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:

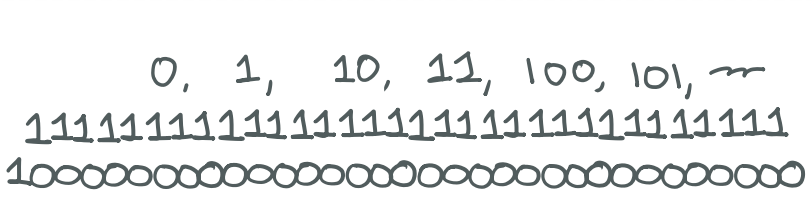
Buy credits ahead of time to pay for work later

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

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:



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

Ideas?

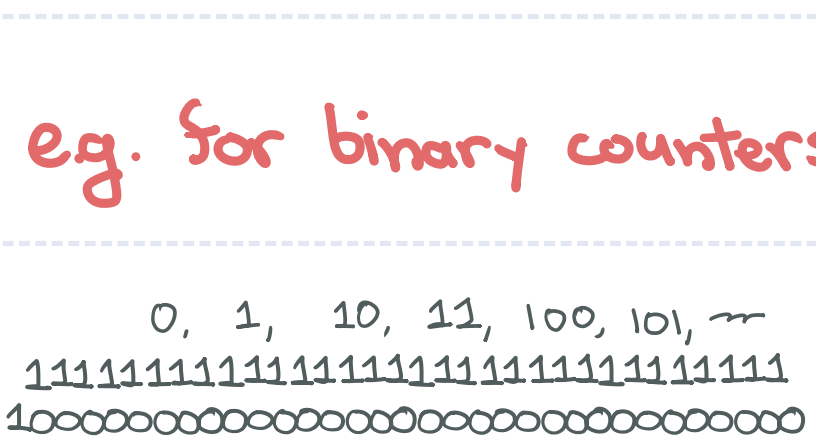
Credit schemes

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:

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

Potential method

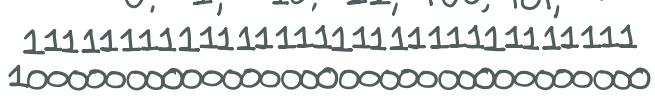
"potential Φ"

Amortized cost of operation:

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

e.g. binary counter

0, 1, 10, 11, 100, 101, ⋯



Φ = [ideas?]

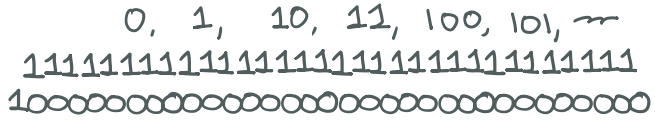
	how often	total
• 1st bit	every time	n
• 2nd bit	every 2 times	$\lfloor \frac{n}{2} \rfloor$
• 3rd bit	every 4 times	$\lfloor \frac{n}{4} \rfloor$
⋮		
• kth bit	every 2^{k-1} times	$\lfloor \frac{n}{2^{k-1}} \rfloor$

Potential method

Amortized cost of operation:

(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)

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

- credit schemes ("banker method")

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

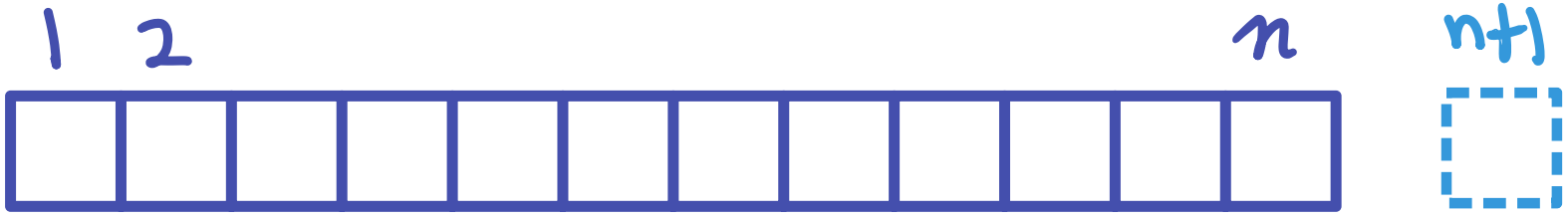
"T amortized time"
$\Rightarrow$ n operations take $O(nT)$ time

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"

- potential function method

Φ = # bits set to 1

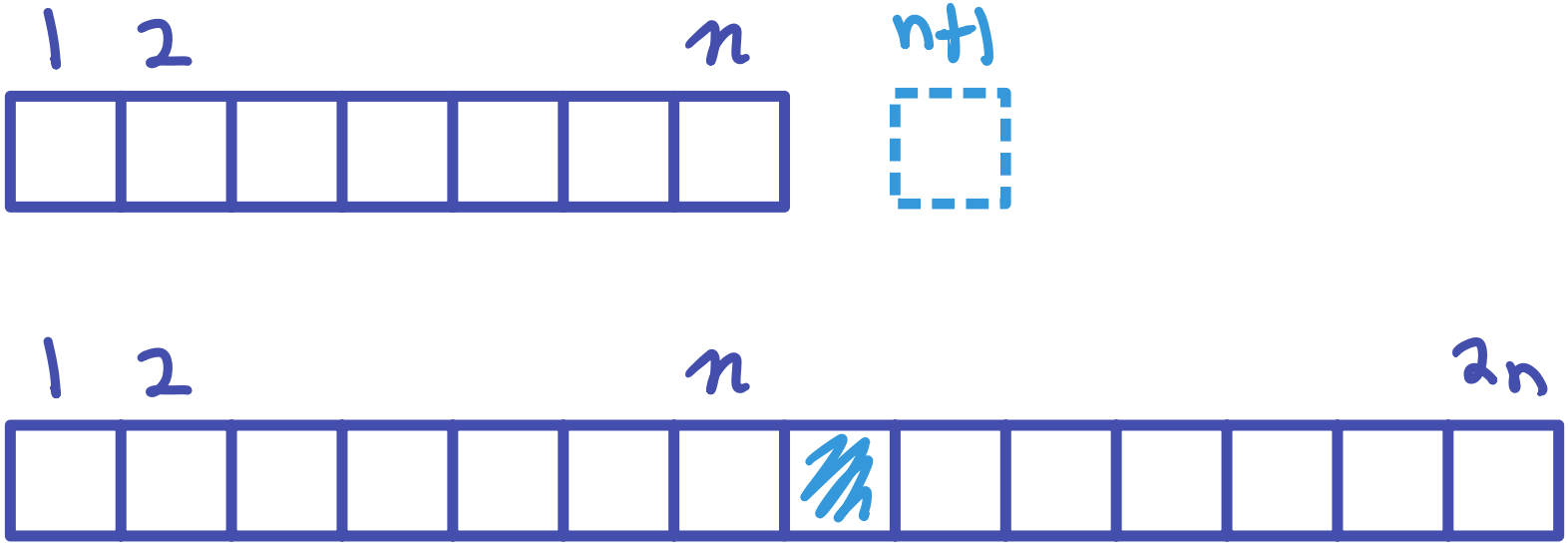
Array-backed lists



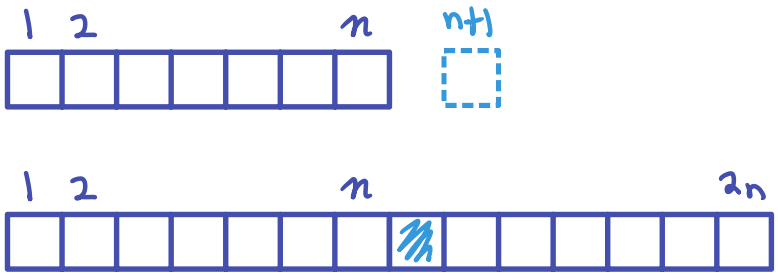
Good: O(1) access by index

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

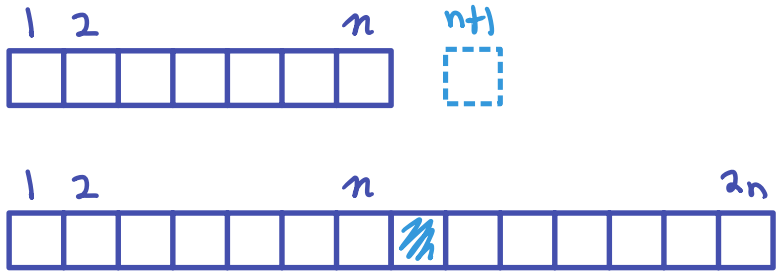
- lookup in O(1)
- append in O(1) amortized

Proof

pay 2 credits each append

Double: → after $\frac{n}{2}$ appends
copy n → n credits

Doubling Trick



Theorem w/ doubling trick, arraylist has

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

Proof

$\Phi = (\#full - \#empty) + 1$

append:

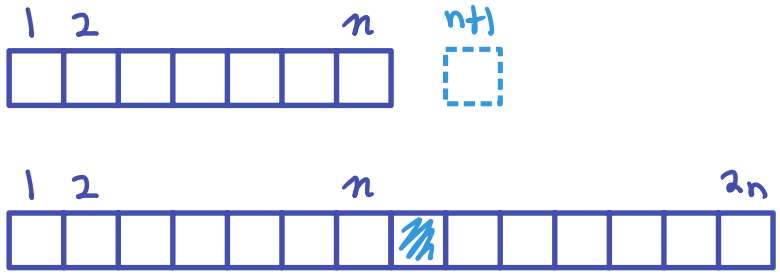
(normally): $\Phi \uparrow 2$

↳ $O(1)$ amortized

double: $\Phi \to \Phi - n$

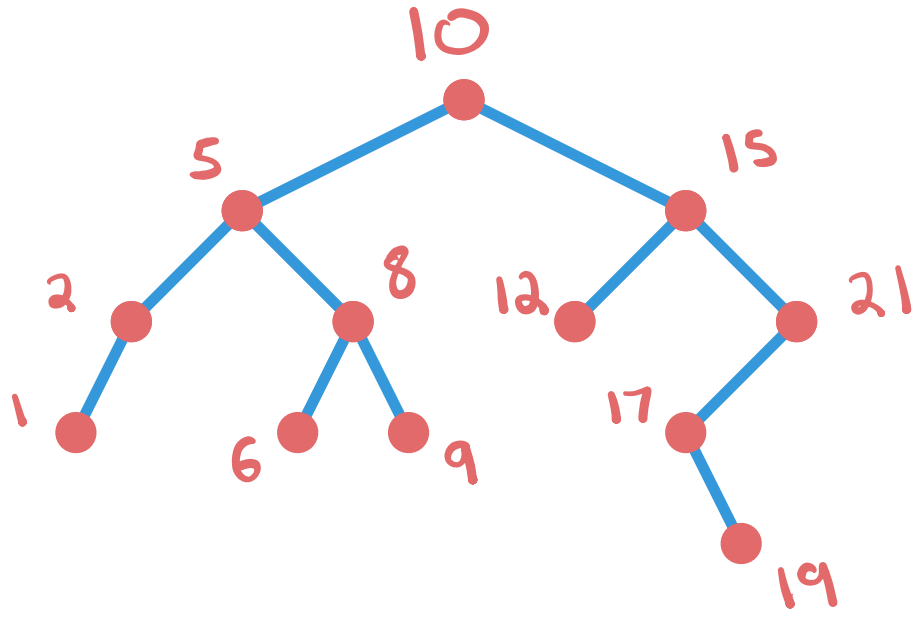
$O(n)$ work → $O(n)$ amortized

Doubling Trick



Search trees

Organizes keys
in a tree



Invariant:

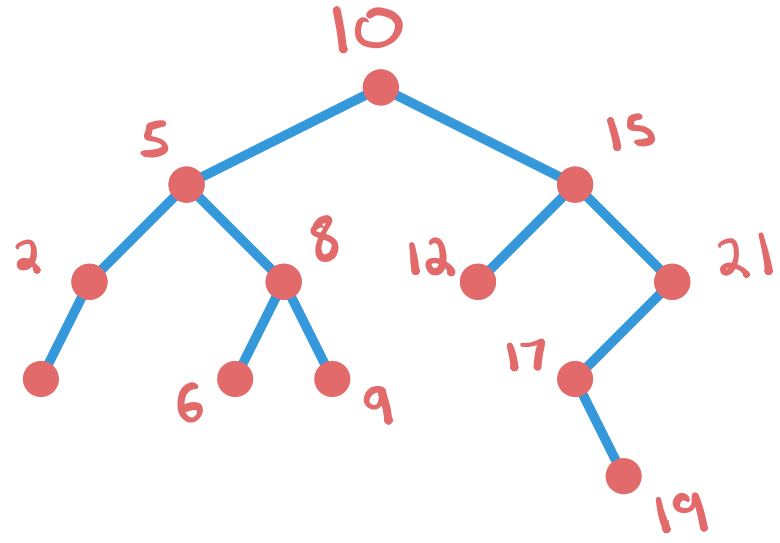
a node's # is

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

Invariant:

a node's # is

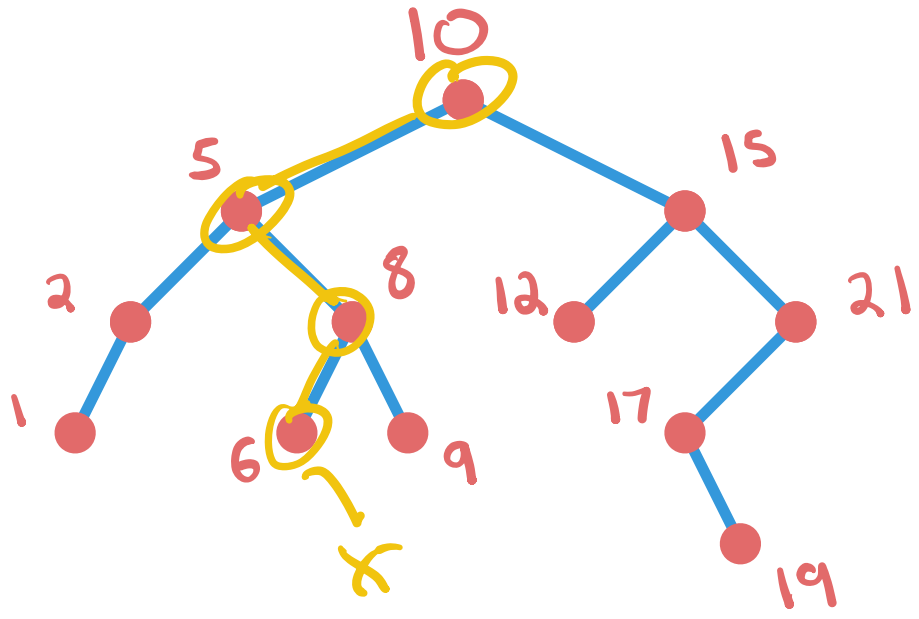
- ≥ any # in left subtree
- ≤ any # in right subtree



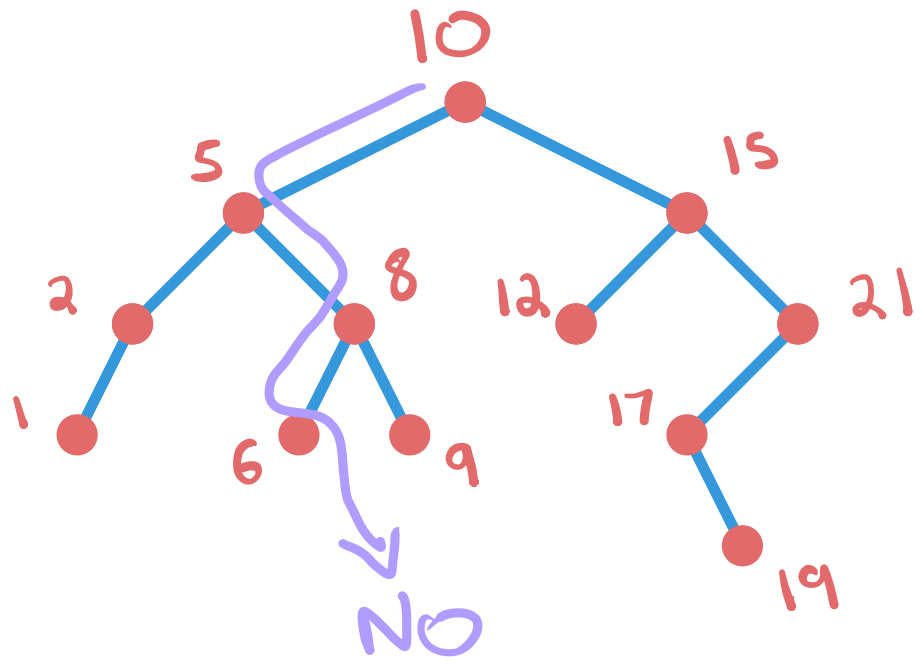
Leads to operations like:

- 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?

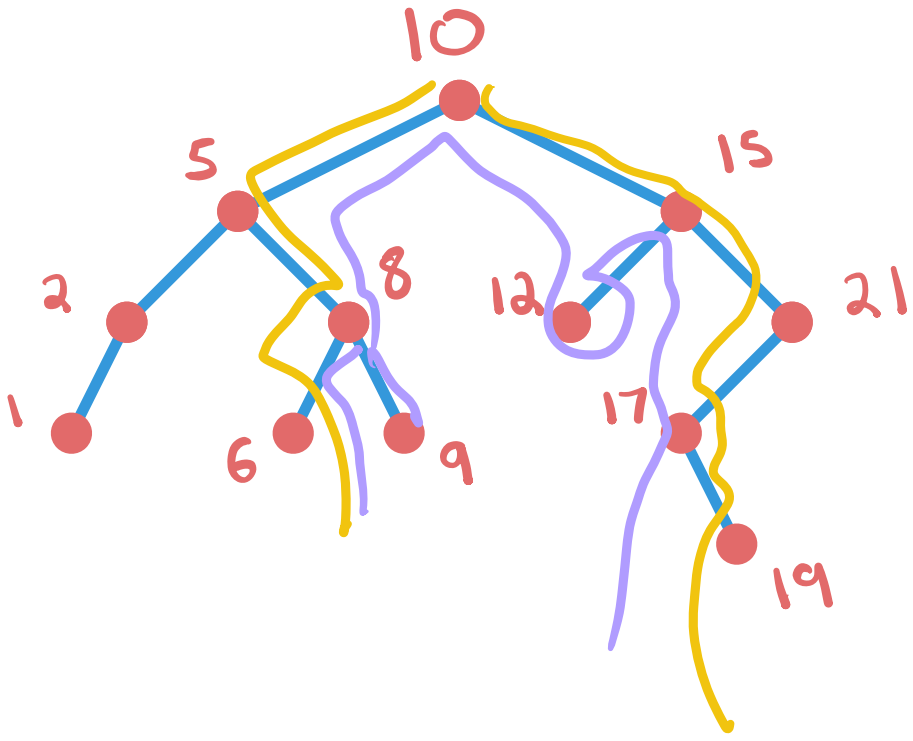
does the tree contain 7?



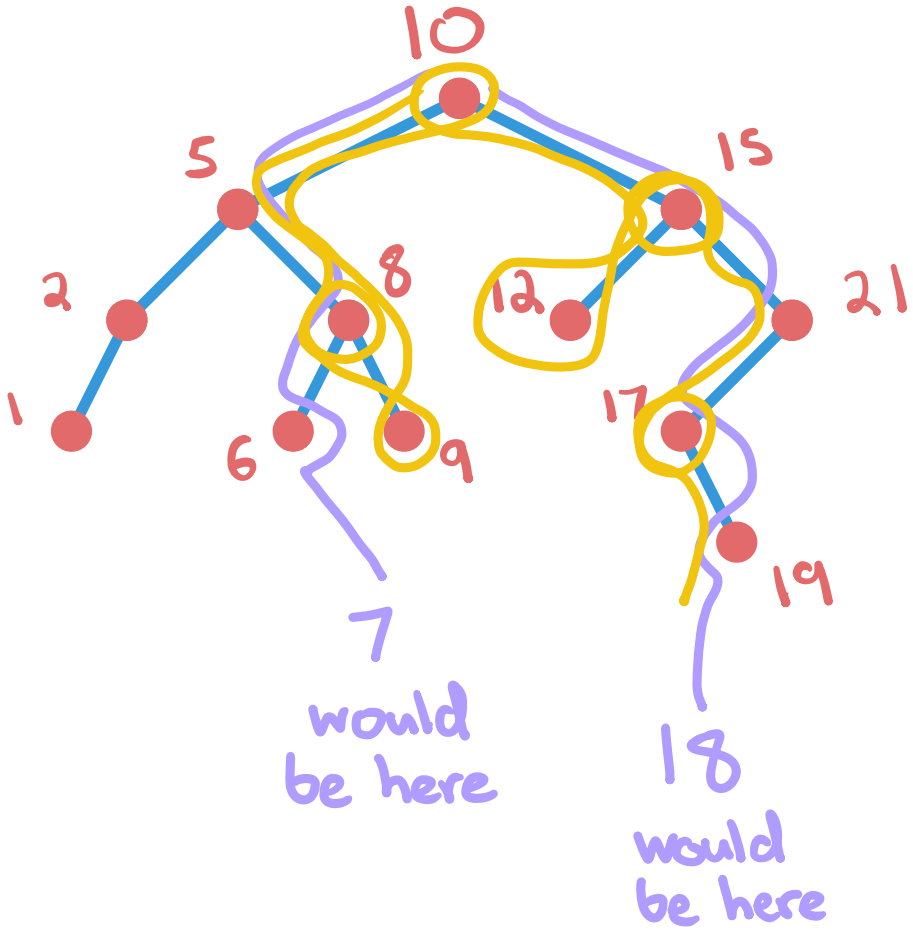
does the tree contain 7?



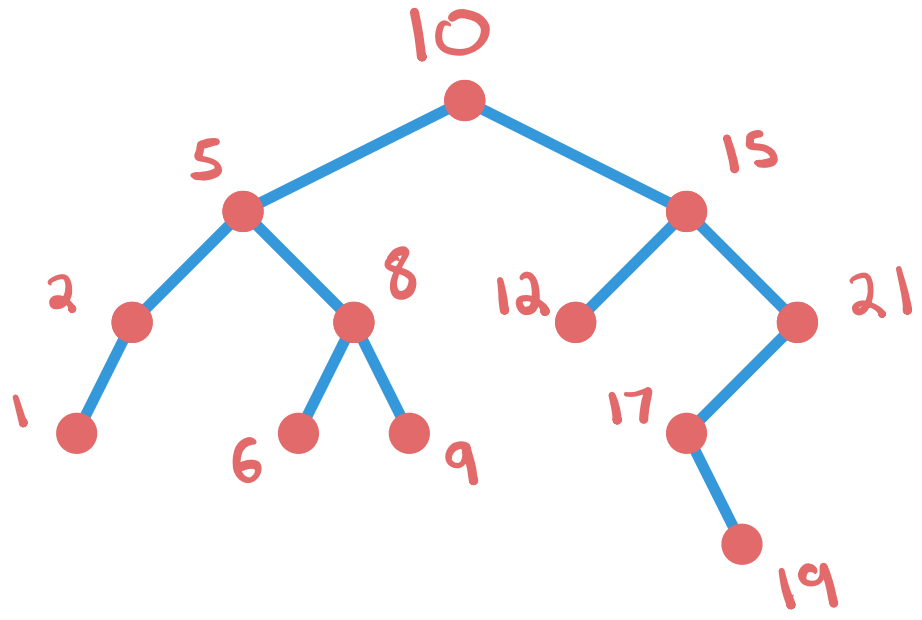
list all keys between 7 and 18



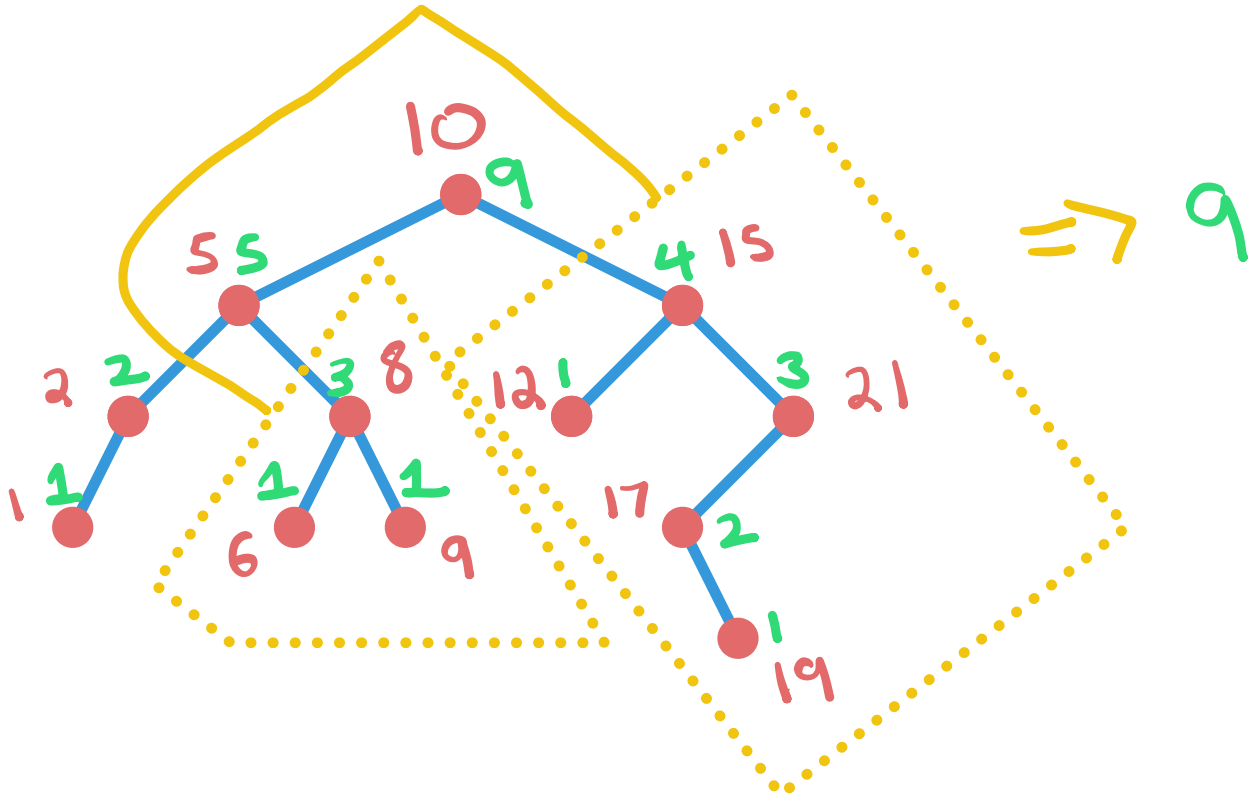
list all keys between 7 and 18



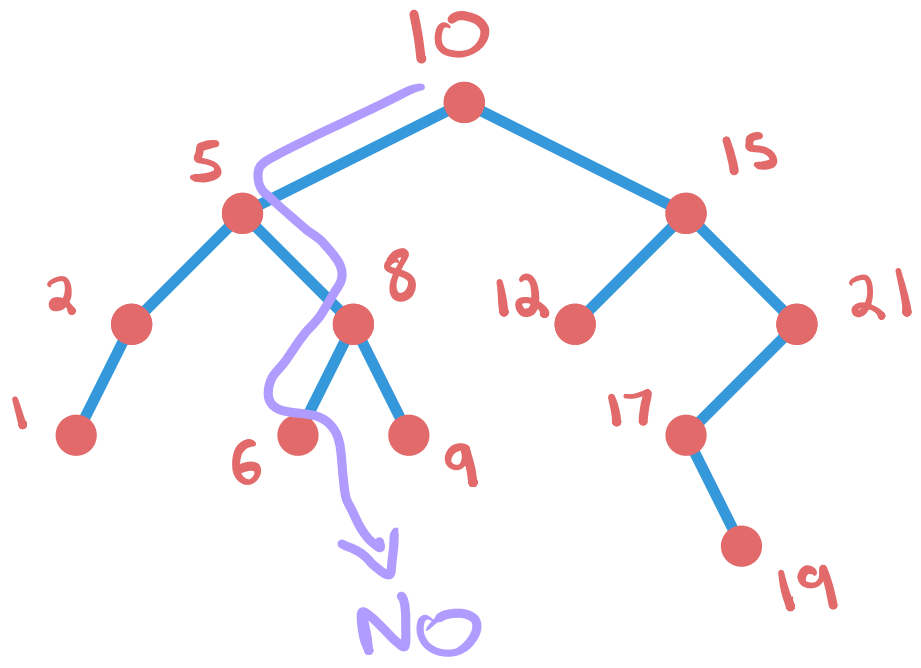
how many keys are between 5 and 25



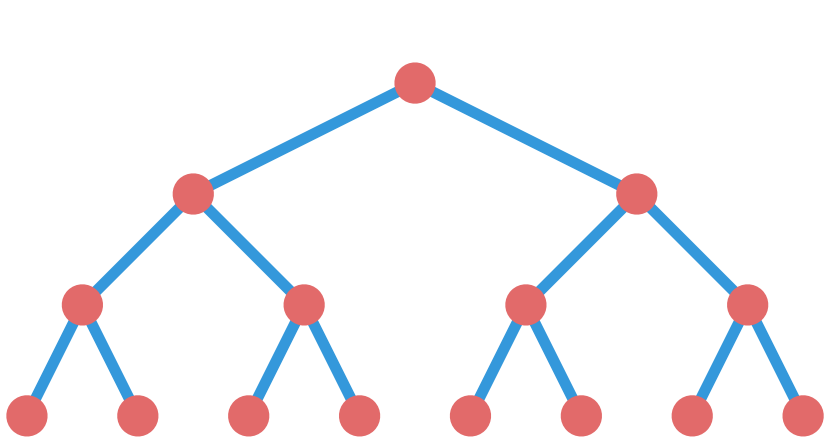
how many keys are between 5 and 25

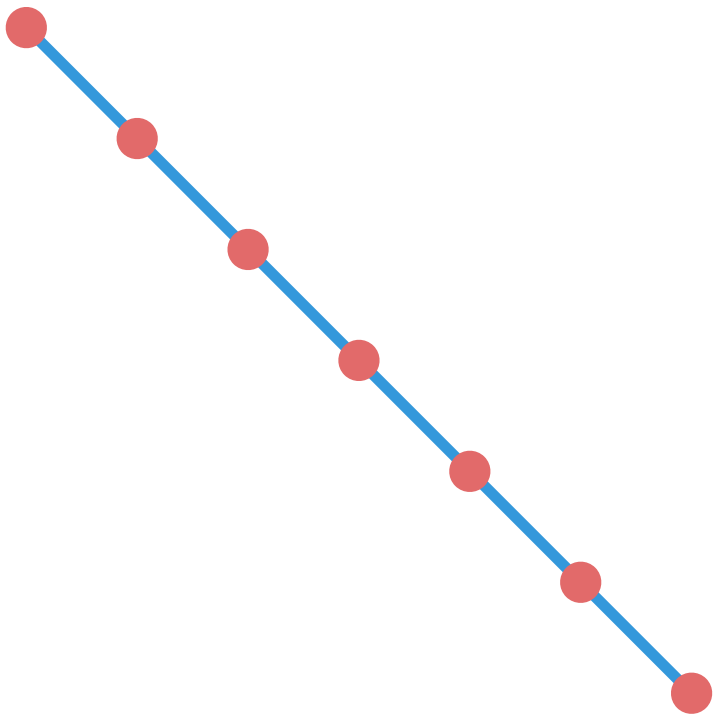


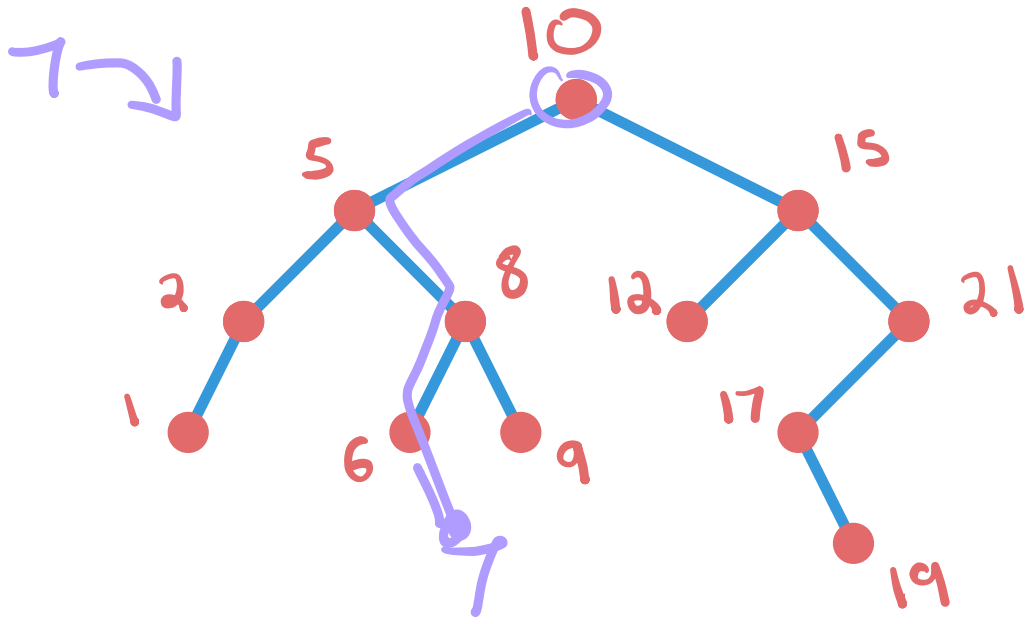
if we write down # nodes in each subtree, then faster



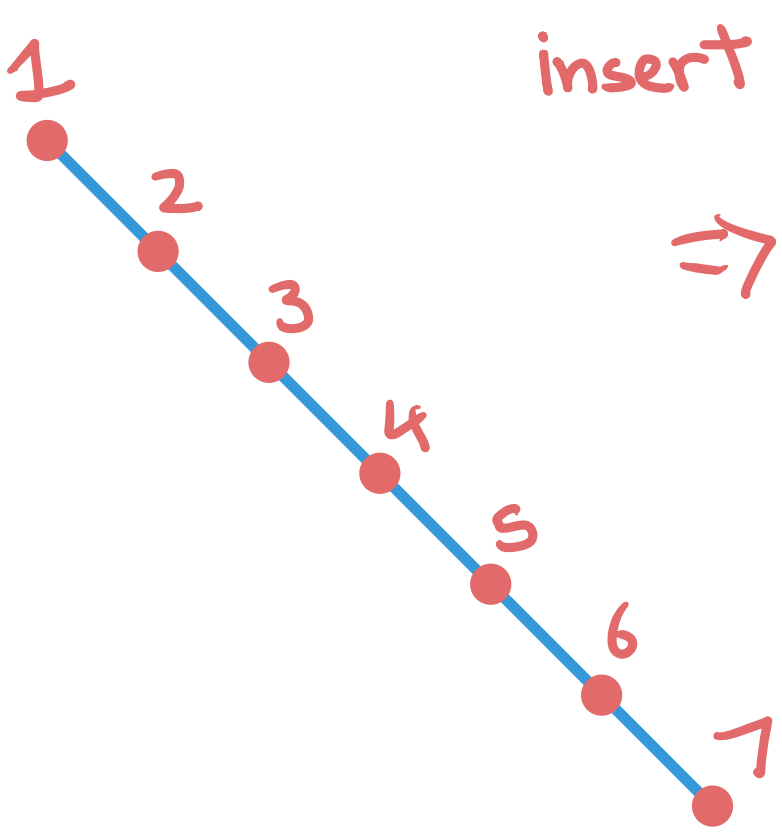
operations take time proportional to the height of a tree





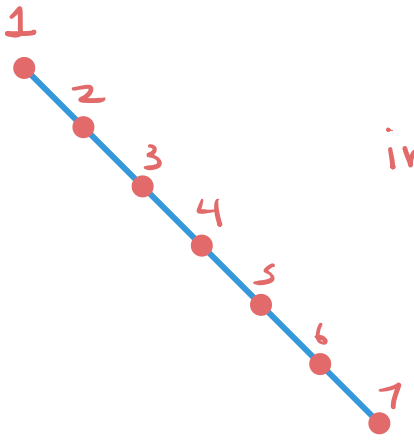


Most natural method for insertion is as leaves
(search for where the key would be, and put it there)



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

⇒ imbalanced



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

⇒ imbalanced

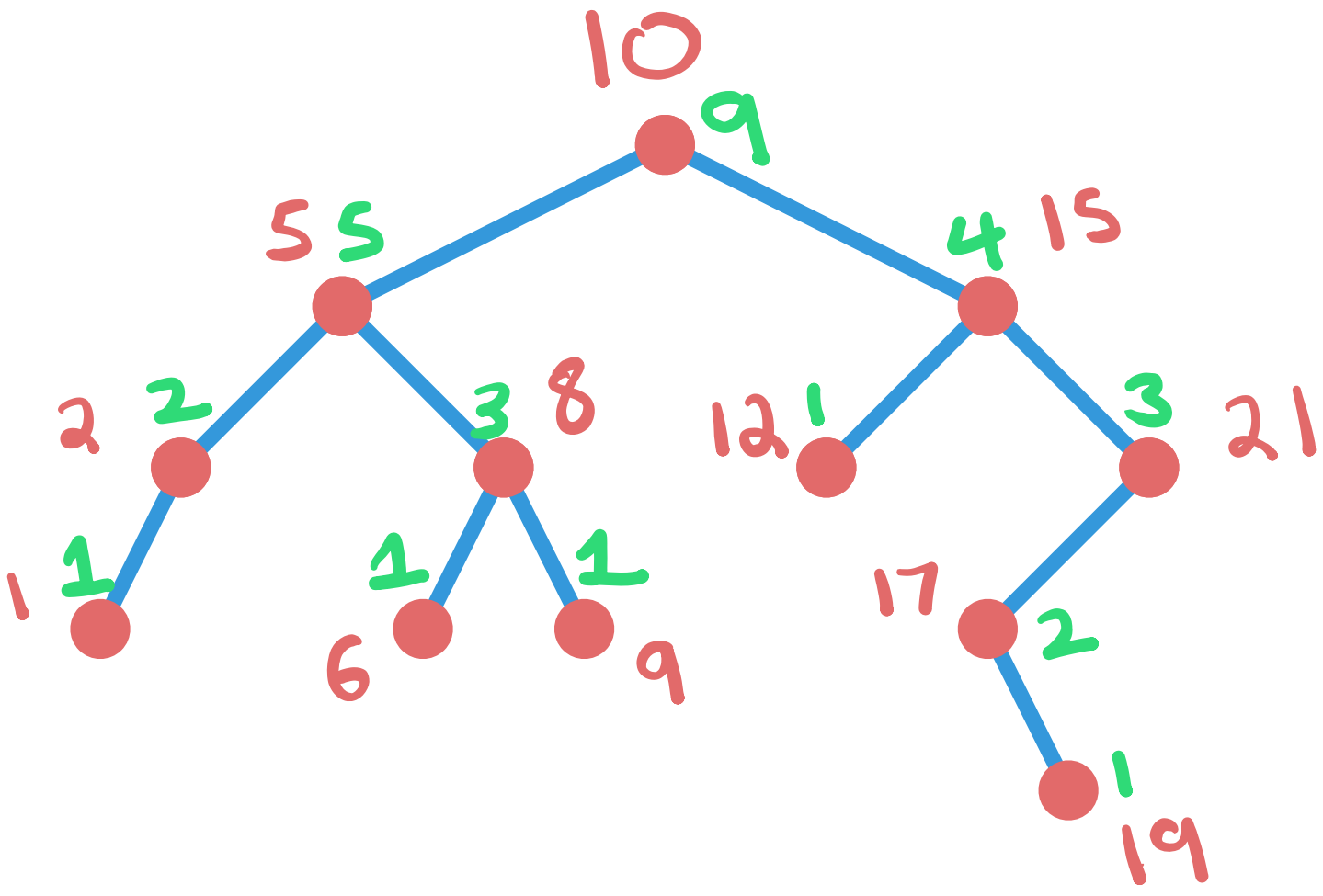
How to keep a tree balanced?

Red Black Tree 2-3 tree

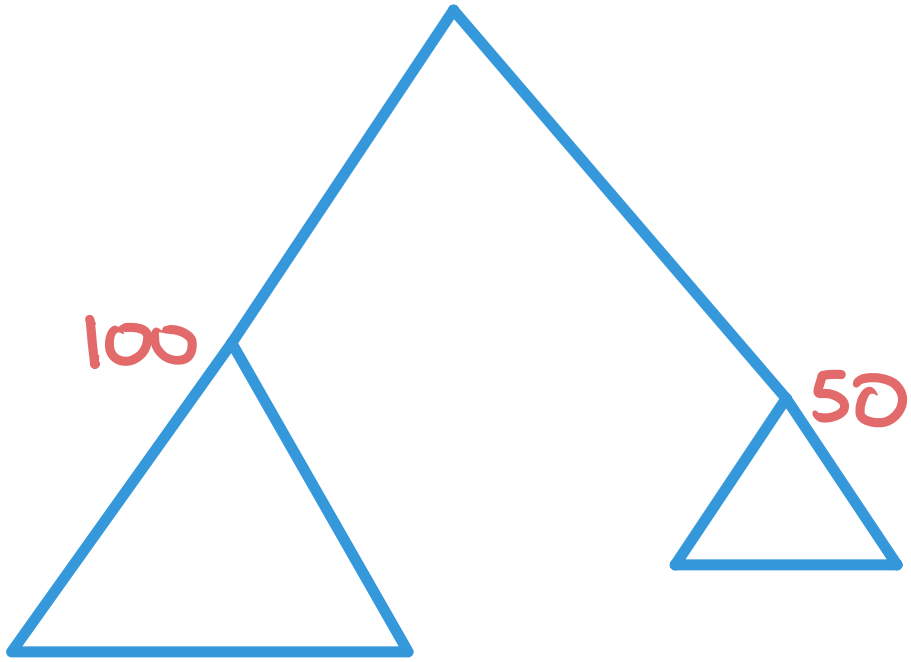
AVL trees

Today we will be very lazy

write down # nodes in each subtree

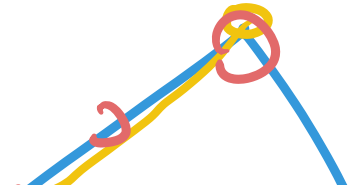


If we detect big imbalance:



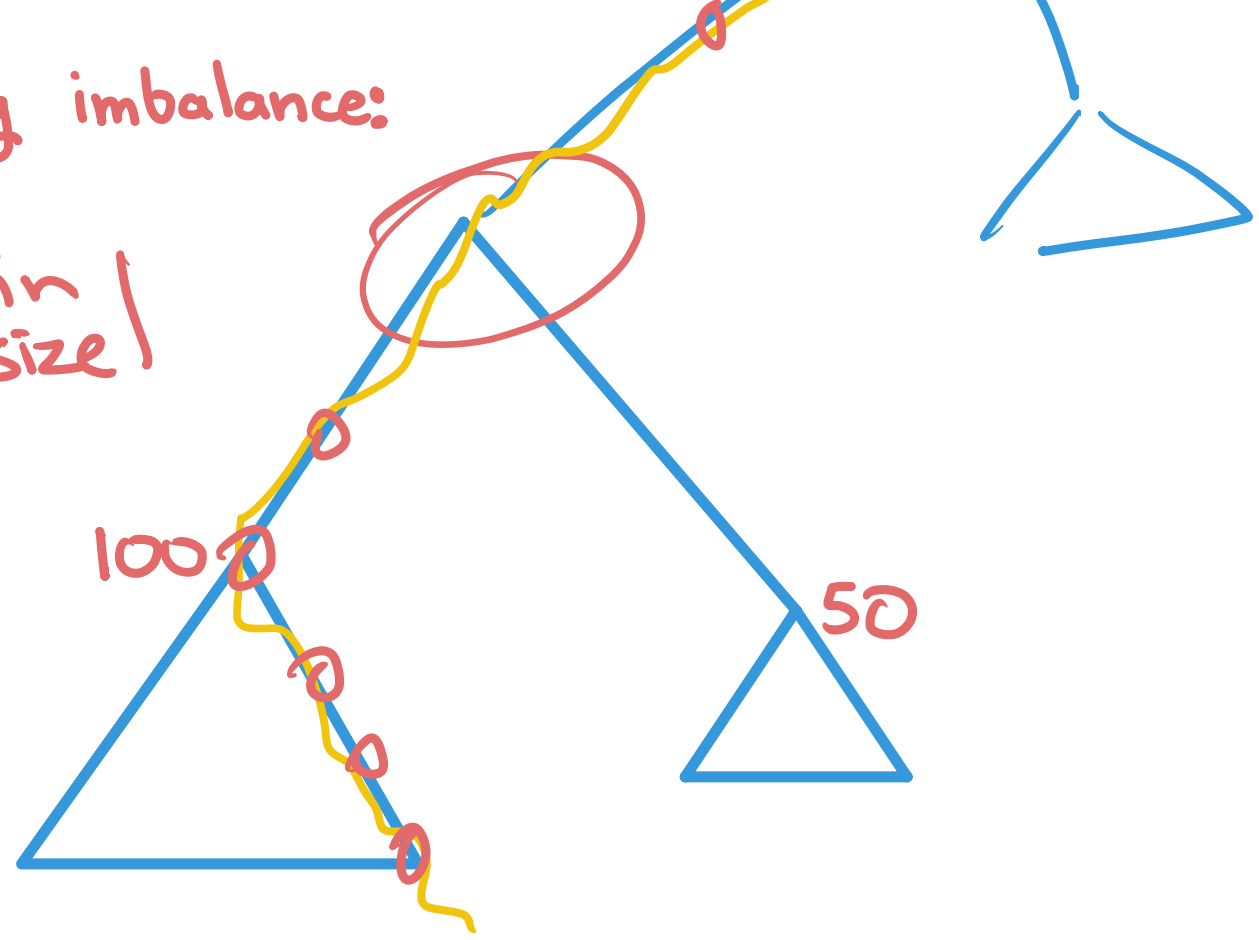
$$\left(\begin{array}{c}\#\text{ in one} \\ \text{subtree}\end{array}\right) > 2 \times \left(\begin{array}{c}\#\text{ in other} \\ \text{subtree}\end{array}\right)$$

(ideas?)



If we detect big imbalance:

Φ = |difference in subtree size|



e.g. 50

$$\begin{pmatrix}\#\text{ in one}\\ \text{subtree}\end{pmatrix} > 2 \times \begin{pmatrix}\#\text{ in other}\\ \text{subtree}\end{pmatrix}$$

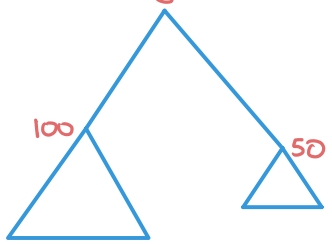
rebuild entire subtree optimally

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

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

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

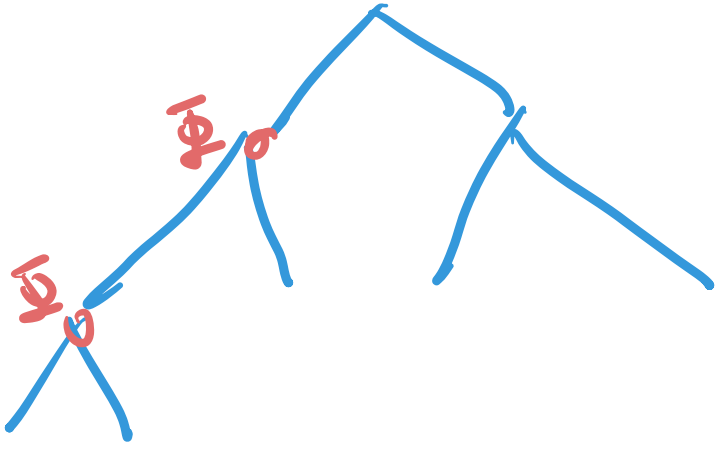
If we detect big imbalance:



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

rebuild entire subtree tree optimally

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



Φ = height tree

= "deepest" - "shallowest"

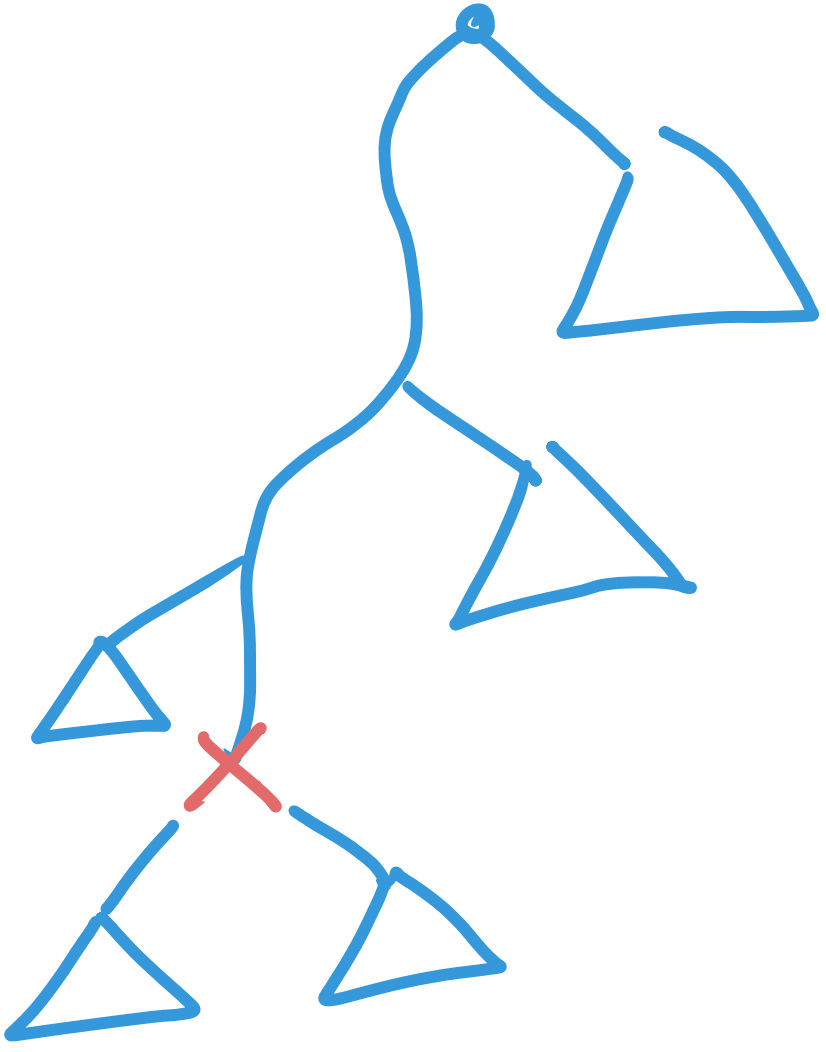
$O(\log n)$

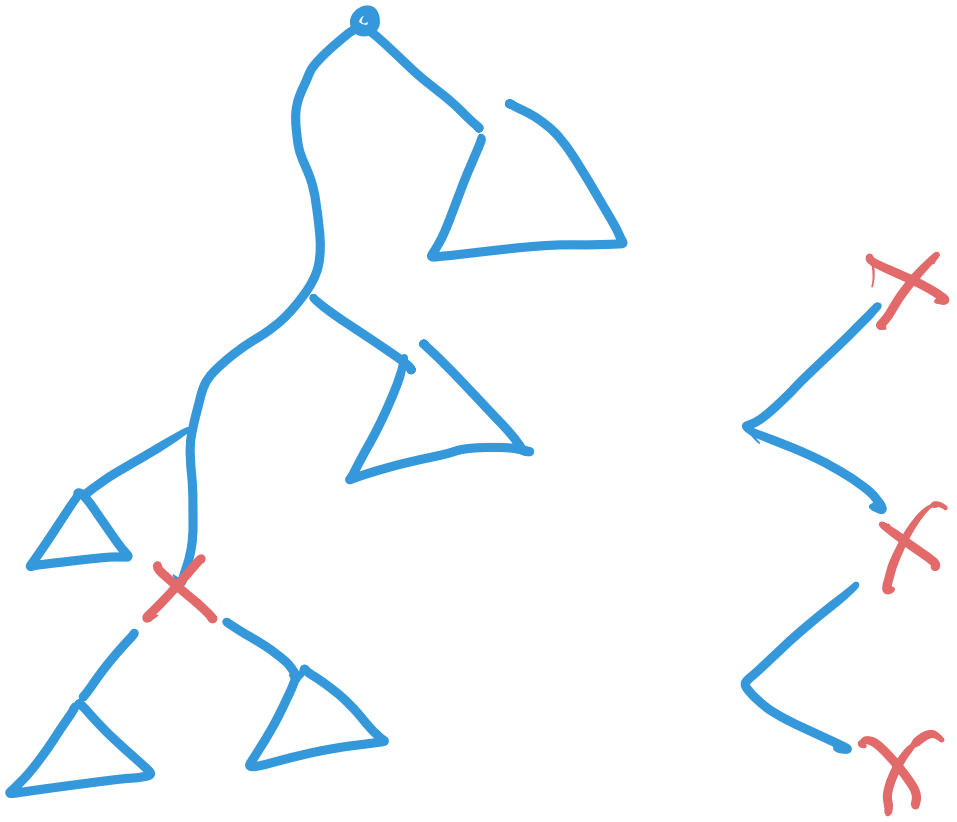
Deletions (remove key from BST)

Old-fashioned way:

1. search for key
2. If key is not a leaf:

chaos



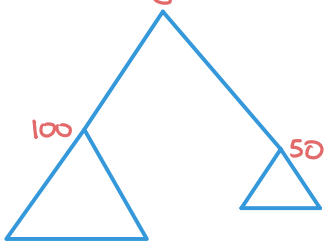


Lazy idea: mark node as deleted

if ≥50% marked for deletion,

rebuid optimally

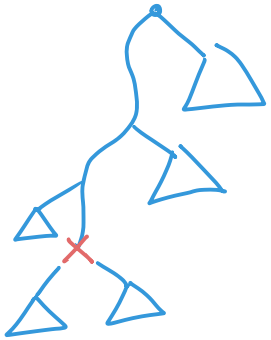
If we detect big imbalance:



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

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

$\Psi = \#\text{unmarked} - \#\text{marked}$

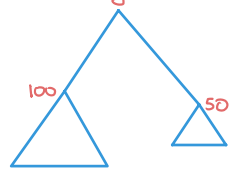
$= \#\text{marked}$

$\Phi_v = |\text{abs diff size of v's subtrees (includes marked)}|$

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

Proof

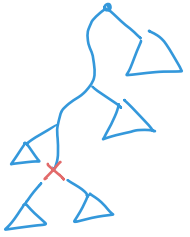
If we detect big imbalance:



(# 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

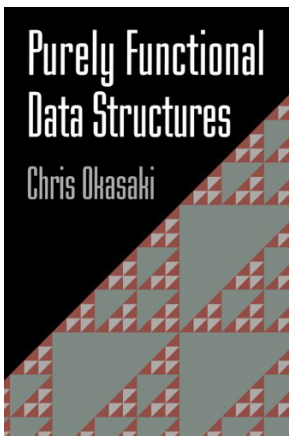
if ≥ 50% marked for deletion, rebuild optimally

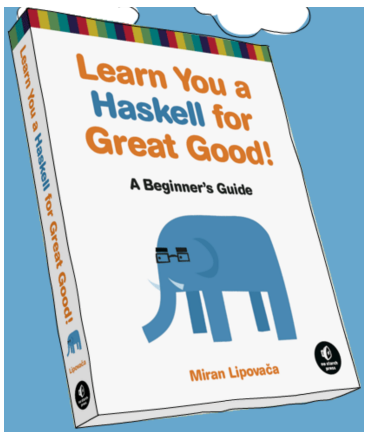
Immutable data structures

Once something is written, never changes

Restrictions bring benefits:

- no side effects
- simplifies concurrency
- helps compiler



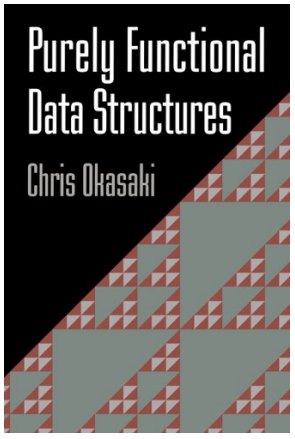


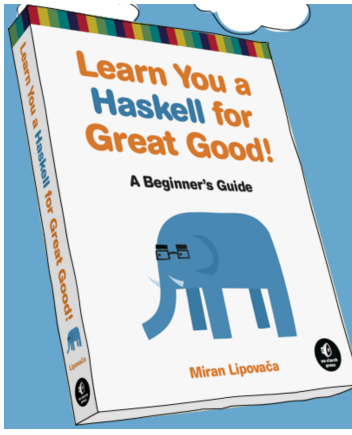
Immutable data structures

Once something is written, never changes

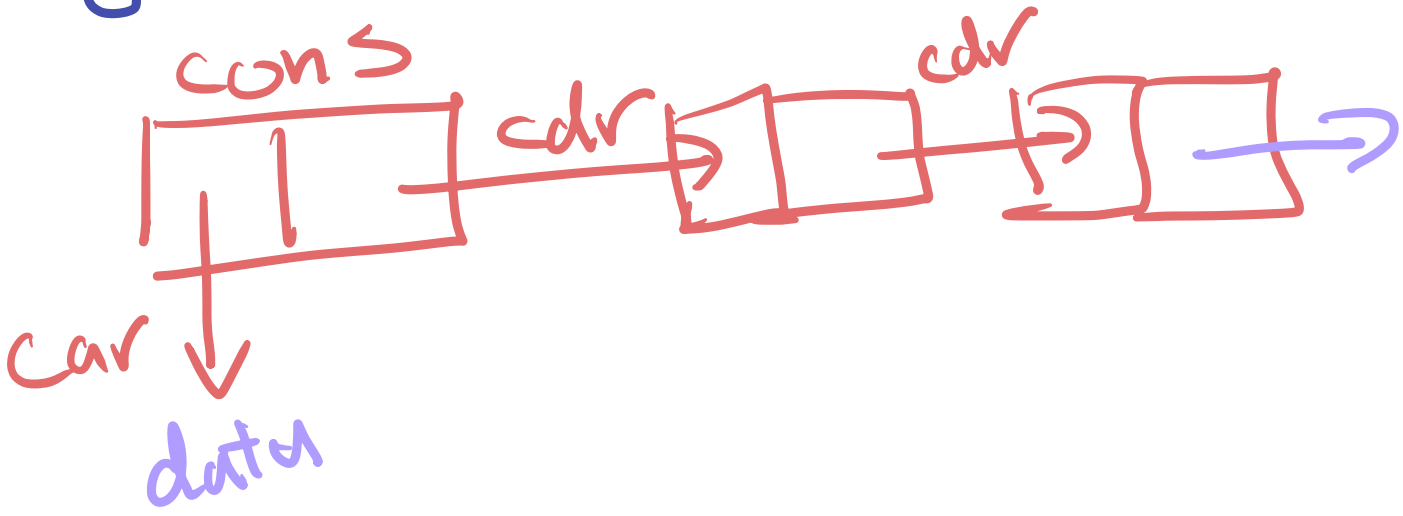
Restrictions bring benefits:

- no side effects
- simplifies concurrency
- helps compiler



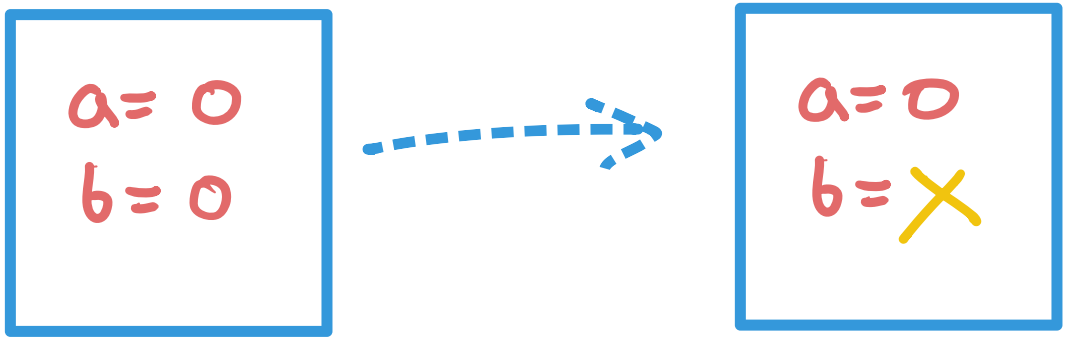


e.g. linked list

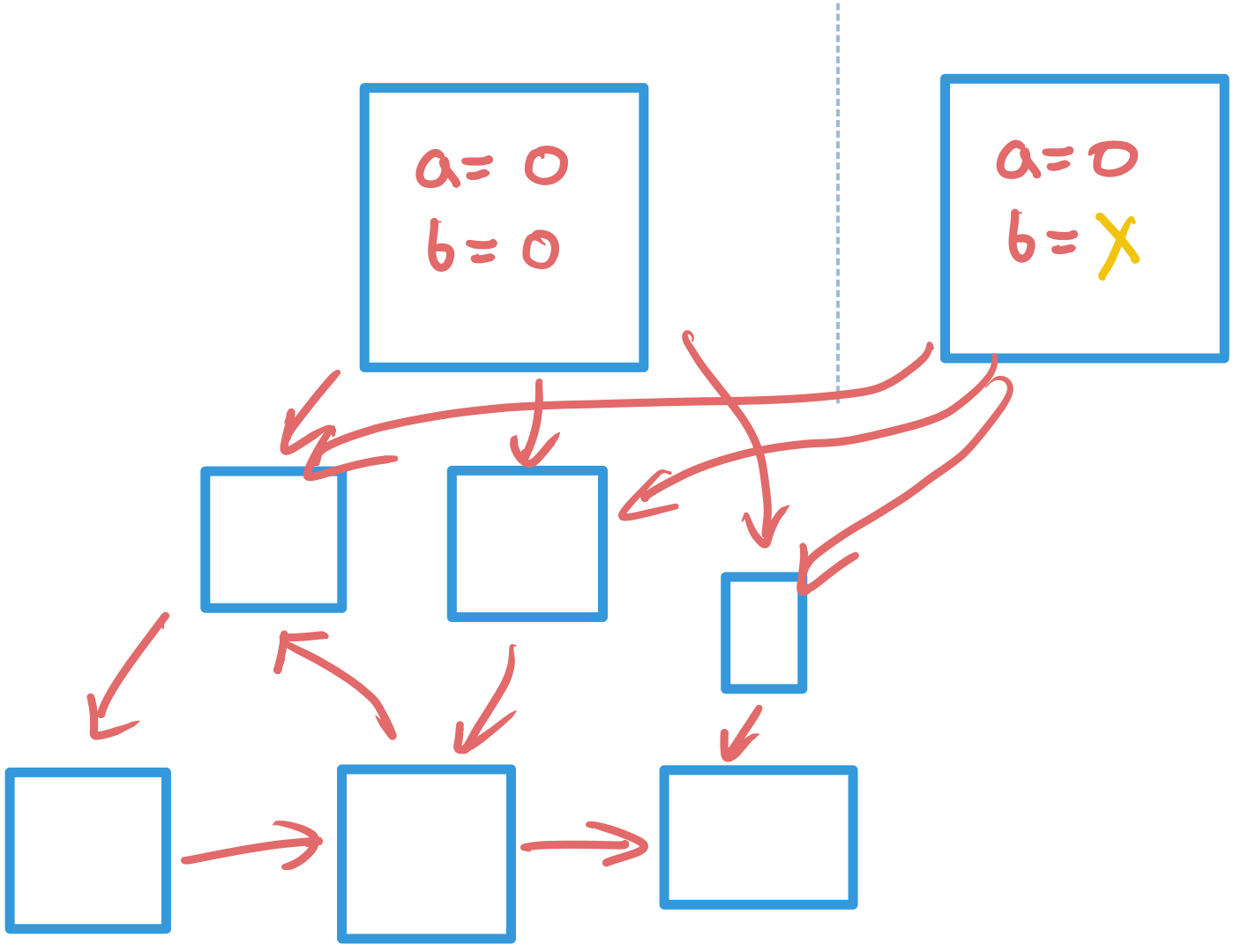


Editing objects by copying

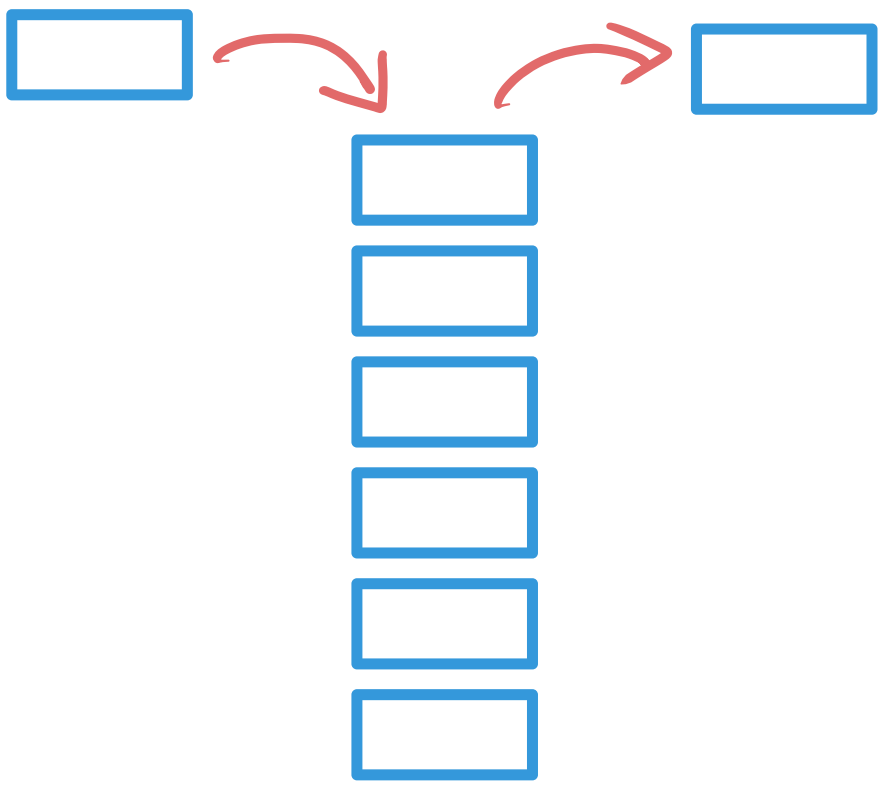
small change => copy whole object



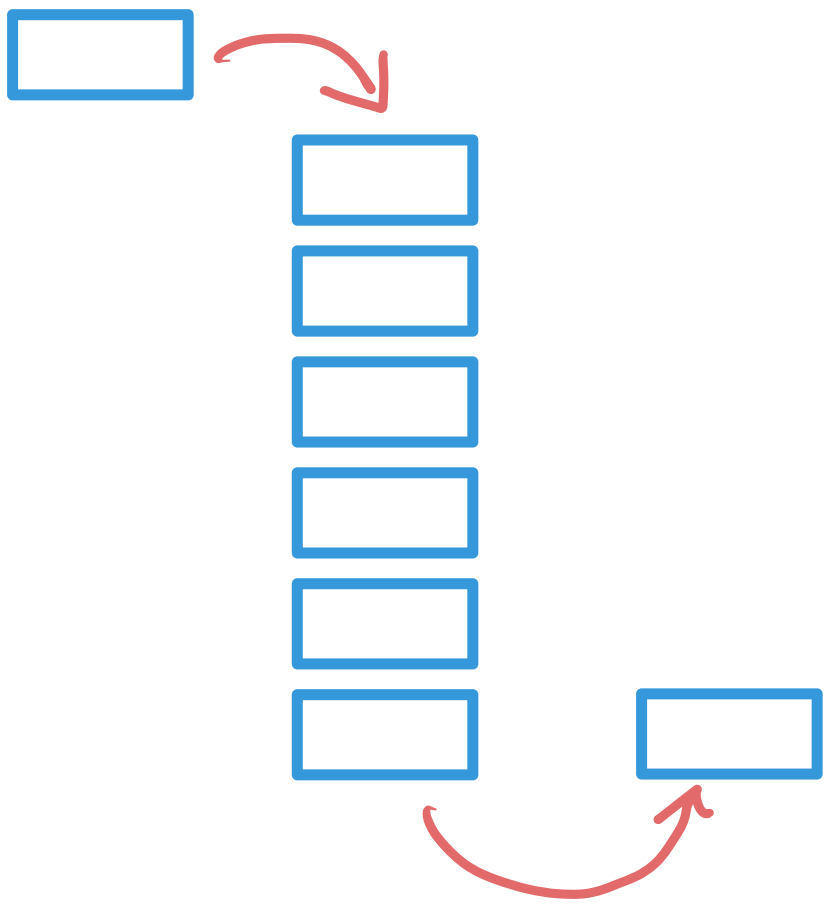
what about "deep" objects?



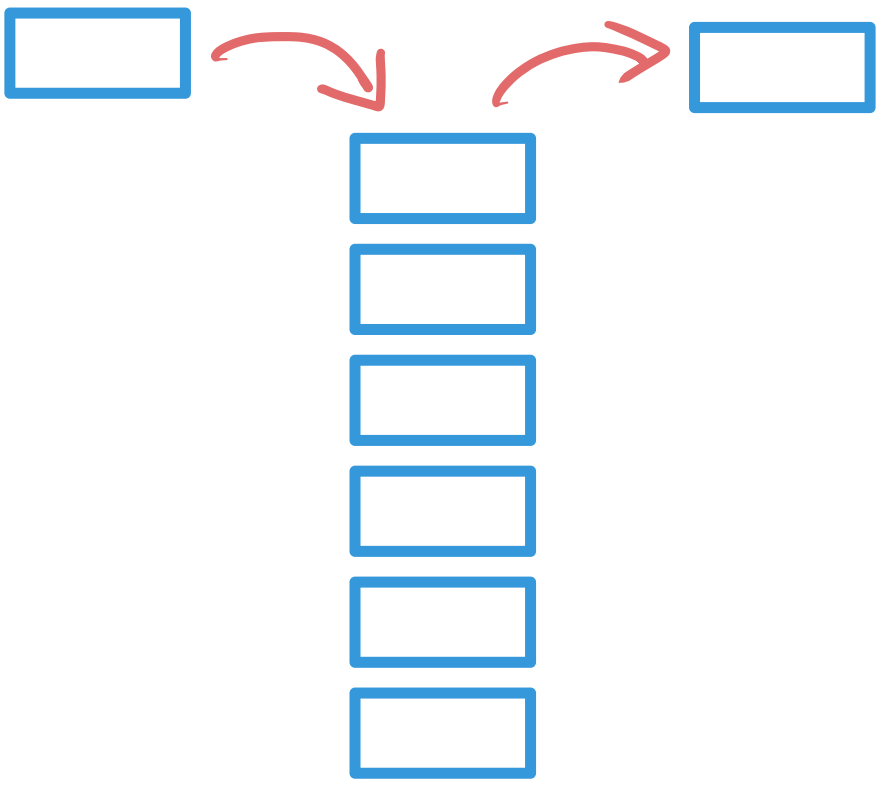
Stacks



Queues

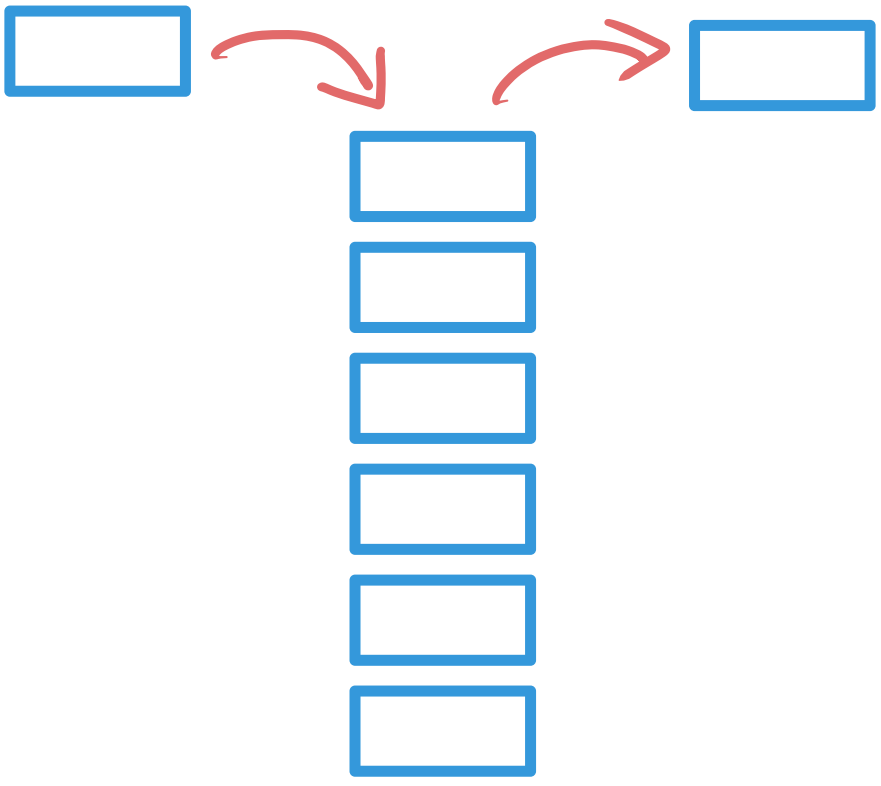


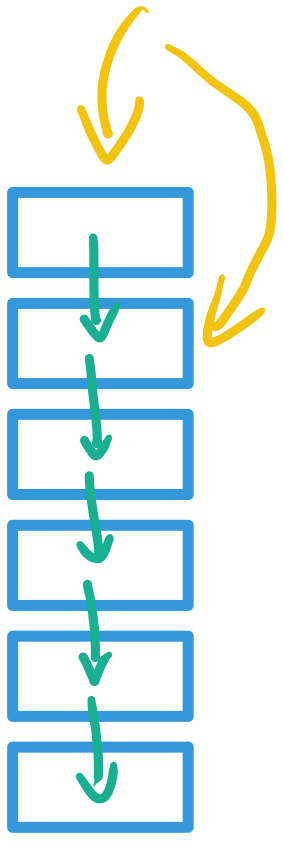
Stacks



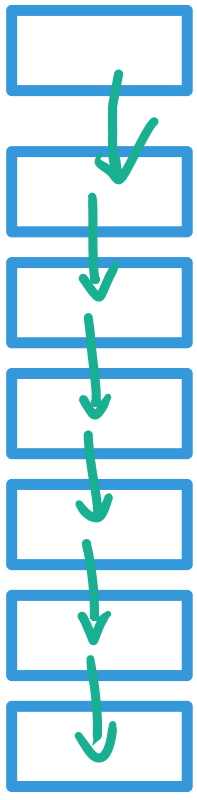
Immutable stacks

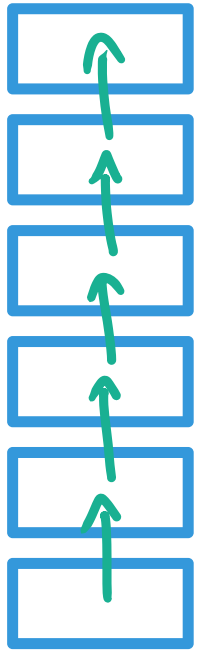
Immutable stacks

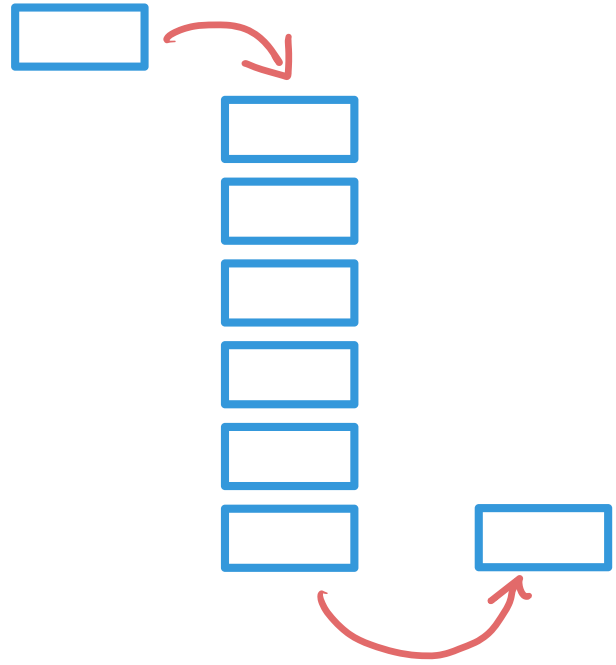




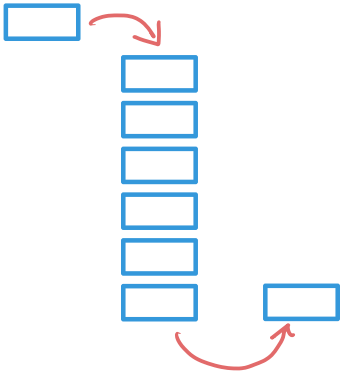
Immutable queues

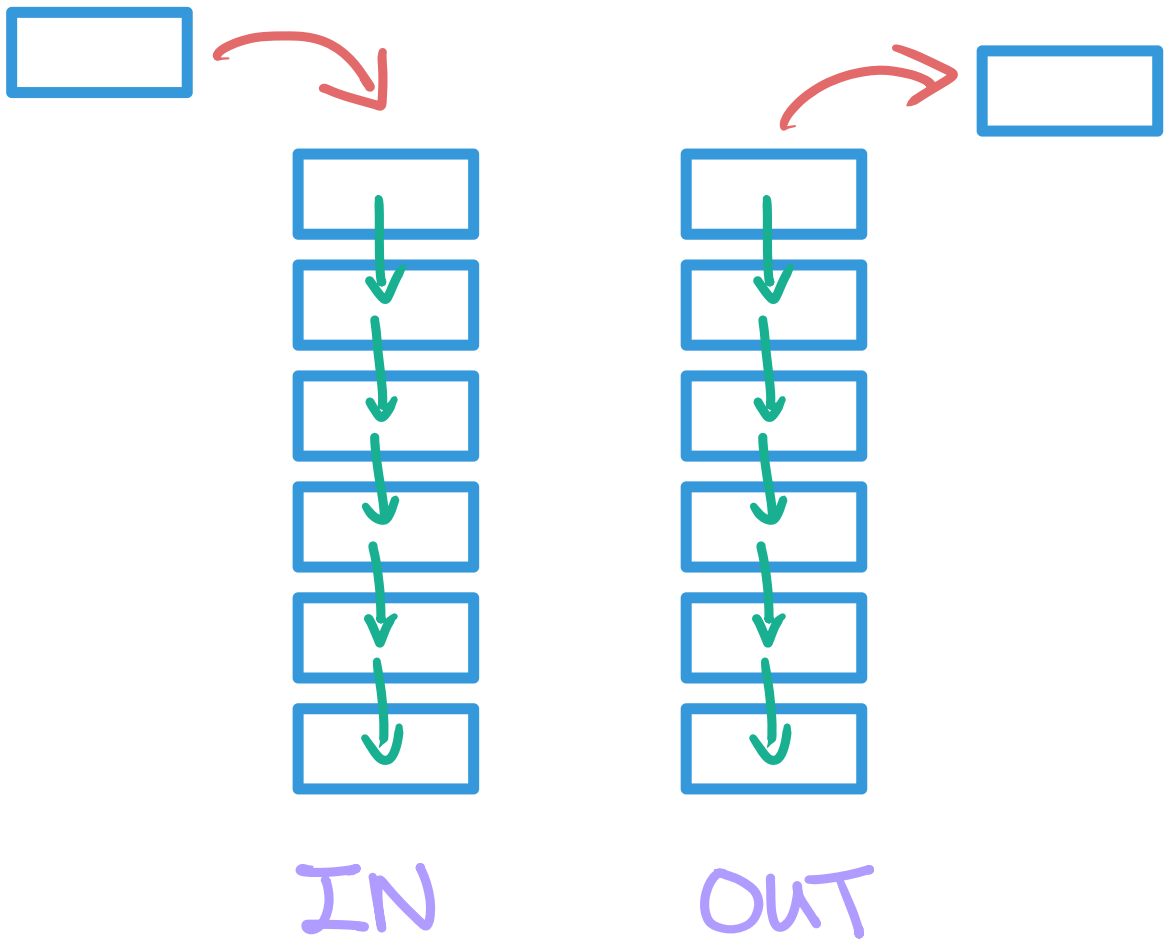


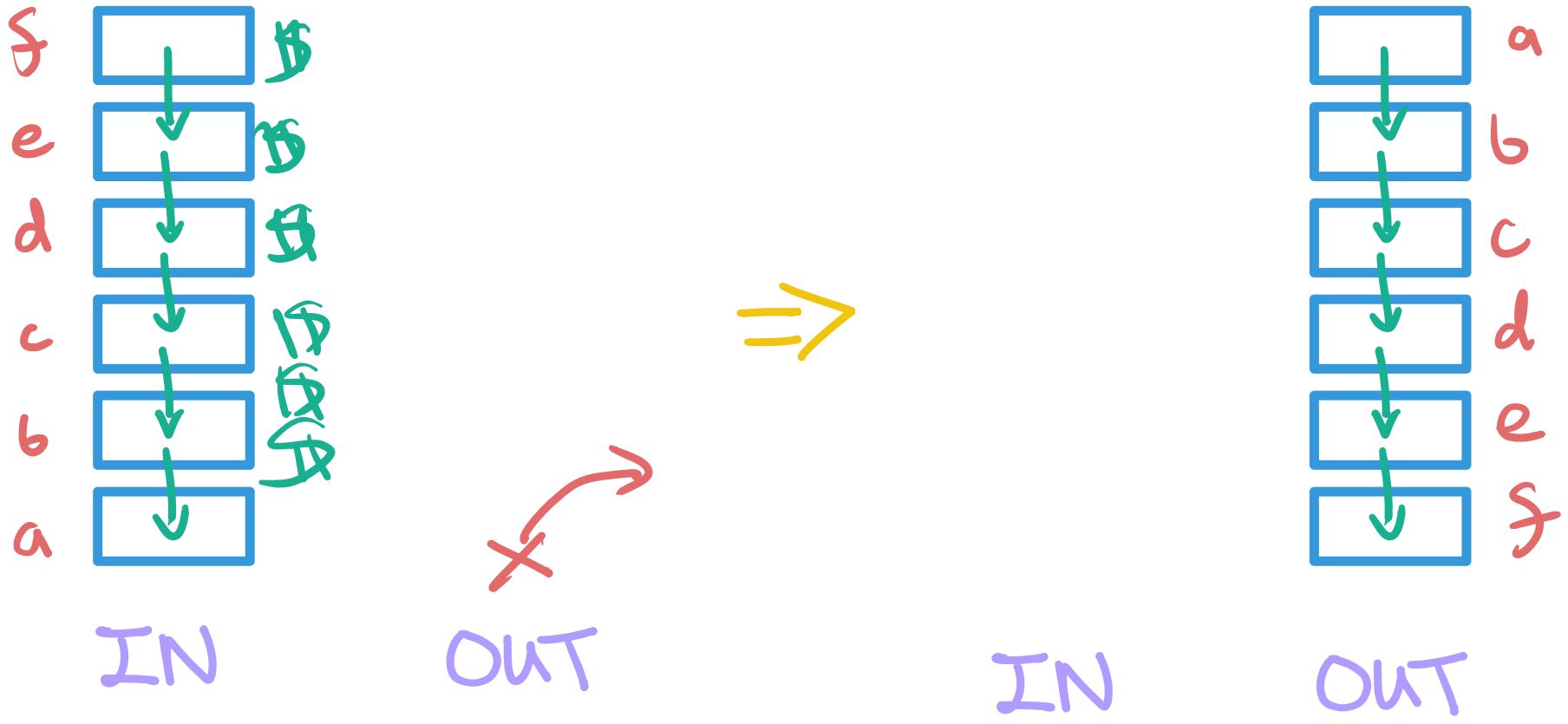


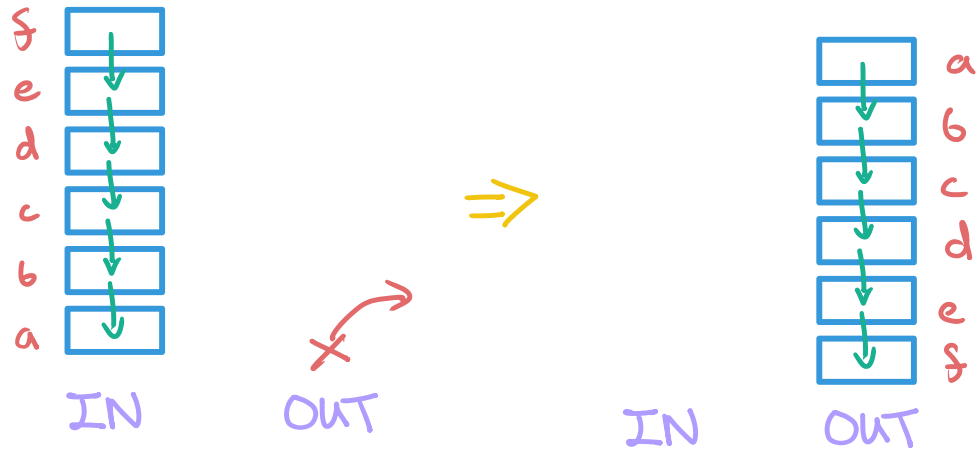


Immutable queues






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



Theorem: insertion/deletion take O(1) amortized

Exercise 22.1. Recall the array-backed lists from section 22.3, where we showed how to support `append` in $O(1)$ amortized time via the "doubling trick". As discussed at the end of section 22.3, one may also want to include a `remove` operation that removes the last item in the list. To implement this, one could just unallocate the corresponding spot in memory, and decrease the counter tracking the size of our list. However, after many such removals, we may end up with an array that is mostly empty and much larger then the number of items in the list. It would be desirable to maintain the array to be within a constant factor. Here we will develop an analog of the doubling trick to facilitate deletions.

1. A first approach to addressing deletions might be as follows.

   > *After removing the element from the array and decreasing the size counter, if the size of the list is now less than half of the capacity of the array, allocate a new array of half the capacity and copy the elements of the list over.*

   Unfortunately, this approach will not run in $O(1)$ amortized time. To prove this, devise a sequence of n `append` / `remove` operations (for n arbitrarily larger) on which the data structure would take $\Omega(n^2)$ time.

2. While the approach above didn't quite work, a slight modification of it will. Implement `remove` so that `append` and `remove` both take $O(1)$ amortized time. (Analyze these operations as well.) Your implementation should still follow the general format of allocating a new array and copying all the elements at certain points in time.

Chapter 22

Lazy data structures

22.1 As easy as 1, 2, 3

We have all coded the line

```
i++;
```

meaning, take an integer i, and increase it by 1. It's the simplest operation in the world, and understood to be very fast. Now consider the underlying bits as we repeatedly increment i, starting from 1.

`1`, `10`, `11`, `100`, `101`...

No big deal. But subtly, in the short list above, one of the increments takes a little bit longer than the others. To go from `11` (3) to `100` (4), we actually flipped *two bits*, rather than one. Of course two bits is no big deal. If we take the number $2^{32} - 1$, and add one:

`11111111111111111111111111111111`, `10000000000000000000000000000000`, ...

we flip 32 bits![1] In general, a k bit number may flip k bits to increase by 1. How odd: the worst case performance of an increment is not uniform, and can be larger for large numbers. Yet we all have coded `i++` many times before and never really worried about this. Should we be worried?

When we count from 1 to n, how many total bits are flipped? Clearly the first (least significant) bit flips every time. The second bit flips every other time. The third bit flips every four times, and the fourth bit flips every 8 times. In general, the kth bit flips every 2^{k-1} times. So the number of bits flips is

$$\sum_{k=1}^{\infty} \left\lfloor \frac{2n}{2^k} \right\rfloor \leq n \sum_{k=1}^{\lceil \log n \rceil} \frac{1}{2^{k-1}} < 2n.$$

[1] Of course, for modern computers, 32 bits is no big deal either. But that's besides the point.

It is *as if* we flipped at most two bits with every increment. *In the aggregate*, an increment acts like a constant time operation. *On average*, each increment takes $O(1)$ time. Realistically, unless we are timing each increment individually (and usually we aren't), we don't notice the difference from a uniformly (worst-case) $O(1)$ running time and an on-average $O(1)$ running time. The technical term to express this $O(1)$ "average time" is to say that incrementing takes $O(1)$ *amortized time*.

22.2 Amortized analysis

In general, an operation takes "T amortized time" if any sequence of n invocations to the operation takes nT total time. To generalize this to multiple operations, suppose we have some data structure that has k different operations $\mathsf{op}_1, \ldots, \mathsf{op}_k$. Let $T_1, \ldots, T_k$ be k different values. We say that each operation op_i takes T_i amortized time if the total running time of any sequence of operations $\mathsf{op}_{i_1}, \mathsf{op}_{i_2}, \ldots, \mathsf{op}_{i_n}$ is

$$O(T_{i_1} + T_{i_2} + \cdots + T_{i_n}).$$

The example of a binary counter above has a single operation, increment, which was shown to take $O(1)$ amortized time.

The binary counter is a good example of the practical tradeoffs of amortized analysis. The algorithm itself is simple and perhaps the most natural one. The analysis was more subtle, and we had to relax our insistence on worst-case bounds. Crucially, amortized analysis does not lose the rigor and robustness of worst-case analysis: we still require the algorithm to attain certain running times on average *in the worst case over all inputs*.

Amortized analysis can be tricky, since once has to "zoom out" to see why a sequence of operations is always good in the aggregate. At the end of the day, we appeal to the formal definition above to establish amortized running times. Fortunately there are a couple of well-worn techniques that sometimes offer an easier way to obtain amortized amounts. Two approaches here we discuss here are:

1. Credit schemes (a.k.a. the "banker's method").
2. Potential functions.

We explain each below and use binary counters as an illustrative example.

Credit schemes. The idea behind credit schemes is to have the cheap operations pay additional "credit" that can be exchanged for work later. Each unit of credit is worth $O(1)$ work. The rule is that we can only spend credit that was saved up in previous iterations.

Let us illustrate with the binary counter. Whenever we call `i++`, imagine placing one unit of credit on the first bit, half a unit of credit on the second bit, one fourth a unit of credit on the third bit, and so forth. This spreads 2 units of credit total.

In this credit scheme, the ith bit gets $1/2^{i-1}$ units of credit in each increment. Recall also that we only flip the ith bit every 2^{i-1} increments. Consequently every time we have to flip the ith bit, it has already accumulated one unit of credit to "pay" for the flip. For each increment, then, we have the following:

$$(\text{time for increment}) + (\text{net increase in credit}) \leq O(1).$$

Over n increments, we have

$$\sum_{k=1}^{n} \left(\begin{pmatrix} \text{time for } k\text{th} \\ \text{increment} \end{pmatrix} + \begin{pmatrix} \text{net increase in credit} \\ \text{for } k\text{th increment} \end{pmatrix} \right) \leq O(n).$$

We also have

$$\sum_{k=1}^{n} \left(\begin{pmatrix} \text{time for } k\text{th} \\ \text{increment} \end{pmatrix} + \begin{pmatrix} \text{net increase in credit} \\ \text{for } k\text{th increment} \end{pmatrix} \right) = (\text{total time}) + \begin{pmatrix} \text{total credit} \\ \text{at the end} \end{pmatrix}.$$

Combining these two equations, rearranging, and observe that the amount of credit is always nonnegative, we have

$$(\text{total time}) \leq O(n) - \begin{pmatrix} \text{total credit} \\ \text{at the end} \end{pmatrix} \leq O(n).$$

So again we conclude that increment takes $O(1)$ amortized time.

The analysis above leveraged the following facts that are are necessary for any legal "credit system".

1. Initially, there was 0 total credits.
2. The total amount of credits is always nonnegative.

Lemma 22.1. *Let operation $\mathsf{op}_1, \ldots, \mathsf{op}_k$ be k operations. Suppose we have a credit scheme (obeying the requirements above) such that for each operation op_i, we have that*

$$(\textit{time for } \mathsf{op}_i) + (\textit{change in credit}) \leq T_i$$

for values $T_1, \ldots, T_k \in \mathbb{N}$. Then each op_i takes T_i amortized time.

Proof. The general analysis here is very similar to the one specific to binary counters above. Consider a sequence of m operations $\mathsf{op}_{i_1}, \ldots, \mathsf{op}_{i_m}$. On one hand, we have

$$\sum_{j=1}^{m} \Big(\text{time for } \mathsf{op}_{i_j} \Big) + \begin{pmatrix} \text{change in credit} \\ \text{for } j\text{th operation} \end{pmatrix} \leq \sum_{j=1}^{m} T_{i_j}$$

by assumption. On the other hand we have

$$\sum_{j=1}^{m} \Big(\text{time for } \mathsf{op}_{i_j} \Big) + \begin{pmatrix} \text{change in credit} \\ \text{for } j\text{th operation} \end{pmatrix} = (\text{total time}) + \begin{pmatrix} \text{total credit} \\ \text{at the end} \end{pmatrix}.$$

Combining these two equations, rearranging, and observe that the amount of credit is always nonnegative, we have

$$\begin{aligned} (\text{total time}) &= \left(\sum_{j=1}^{m} \Big(\text{time for } \mathsf{op}_{i_j} \Big) + \begin{pmatrix} \text{change in credit} \\ \text{for } j\text{th operation} \end{pmatrix} \right) - \begin{pmatrix} \text{total credit} \\ \text{at the end} \end{pmatrix} \\ &\leq O\left(\sum_{j=1}^{m} T_{i_j} \right) - \begin{pmatrix} \text{total credit} \\ \text{at the end} \end{pmatrix} \leq O\left(\sum_{j=1}^{m} T_{i_j} \right), \end{aligned}$$

as desired. □

22.2.1 Potential function arguments.

The second method we present for amortized analysis is called the *potential function method*, and is inspired by the use of potential functions in physics. The idea is to define Φ as a nonnegative function of the state of our data structure. It is important that Φ is initially 0, and always nonnegative. We then try to obtain an upper bound on

$$(\text{running time}) + (\text{change in } \Phi)$$

for each operation, which leads to an amortized running time. Henceforth, we write $\Delta\Phi$ as a shorthand for the change in potential Φ. We note that there are similarities between the notions of "potential" Φ here and the "total credit" above; this connection is discussed further at the end of the section. The graph below charts a potential function Φ going up and down over time; the red lines highlight segments where Φ is increasing, and which increases the amortized cost; the green lines highlight segments where Φ is decreasing, which decreases the amortized cost.

We first demonstrate the use of this method on binary counters. We define a potential function Φ by

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

Note that Φ is 0 initially, and always nonnegative.

Consider Φ when we call `i++`. Recall that the net effect of `i++` is to flip h bits from 1 to 0 (for some $h \in \mathbb{Z}_{\geq 0}$), and at most one bit from 0 to 1. Thus the operation takes $O(h)$ time, and the potential Φ decreases by $h - 1$. That is, $\Delta\Phi \leq 1 - h$. The decrease in $\Delta\Phi$ by h offsets the time spent for flipping the h bits. Together we have that for any increment,

$$(\text{running time}) + \Delta\Phi = O(1) \tag{22.1}$$

Now consider a sequence of n increments. We have

$$\sum_{k=1}^{n} \left(\begin{array}{c} \text{time for } k\text{th} \\ \text{increment} \end{array} \right) \overset{\text{(a)}}{=} \sum_{k=1}^{n} \left(\left(\begin{array}{c} \text{time for } k\text{th} \\ \text{increment} \end{array} \right) + \left(\begin{array}{c} \Delta\Phi \text{ for } k\text{th} \\ \text{increment} \end{array} \right) \right) - (\Phi \text{ at the end})$$
$$\overset{\text{(b)}}{=} O(n) - (\Phi \text{ at the end}) \overset{\text{(c)}}{\leq} O(n).$$

Here, in (a), we observe that the sum of changes in Φ from beginning to end is simply the value of Φ at the end. (Note that $\Phi = 0$ initially). In (b), we apply (22.1) to each increment. (c) observes that Φ is always nonnegative.

The potential-based argument above is generalizes beyond binary counters. The only point specific to binary counters was (22.1). The following lemma gives sufficient conditions so that for other applications, obtaining the analog of (22.1) implies the corresponding amortized bounds.

Lemma 22.2. *Let operation $\mathsf{op}_1, \ldots, \mathsf{op}_k$ be k operations, Φ a nonnegative potential function initially set to 0, and $T_1, \ldots, T_k$ k values, such that each time we call op_i, we have*

$$(\textit{time for } \mathsf{op}_i) + \Delta\Phi \leq T_i. \tag{22.2}$$

Then each op_i takes T_i amortized time.

Proof. The proof follows essentially the same format as the analysis above for binary counter; the only difference is that this proof is generic. Consider any sequence of operations $\mathsf{op}_{i_1}, \ldots, \mathsf{op}_{i_m}$. Let $\Phi_0 = 0$ denote the initial potential, and for each $k \in \mathbb{N}$, let Φ_k denote the potential after the kth operation op_{i_k}. Let $\Delta_k \Phi = \Phi_k - \Phi_{k-1}$ denote the change in potential from the kth operation. We have

$$\begin{aligned}
&(\text{time for } \mathsf{op}_{i_1}) + \cdots + (\text{time for } \mathsf{op}_{i_m}) \\
&\overset{\text{(a)}}{=} ((\text{time for } \mathsf{op}_{i_1}) + \Delta_1 \Phi) + \cdots + ((\text{time for } \mathsf{op}_{i_m}) + \Delta_m \Phi) - \Phi_m \\
&\overset{\text{(b)}}{=} T_{i_1} + \cdots + T_{i_m} - \Phi_m \overset{\text{(c)}}{\leq} T_{i_1} + \cdots + T_{i_m},
\end{aligned}$$

as desired. Here (a) observes that $\sum_{i=1}^m \Delta_i \Phi = \sum_{i=1}^m \Phi_i - \Phi_{i-1} = \Phi_m - \Phi_0 = \Phi_m$. (b) applies (22.2) to each operation. (c) is because the potential is always nonnegative. □

Remark 22.3. The charging and potential schemes are implicitly similar to each other: the potential acts as credit that pays for heavy operations later on; conversely, the total credit defines a potential function. Perhaps this is not surprising since they are both implicitly achieving the same goal of proving amortized running times. Let us not take the question of their formal distinction too seriously: ultimately these are metaphors meant only to aid our thinking.

22.3 Array Lists

We all know that an array is a sequence of entries $A[1..n]$ where each $A[i]$ can be accessed in constant time. The entries $A[1..n]$ are literally arranged in consecutive slots in memory. To retrieve $A[i]$, we really only need to know the location of $A[1]$, and then add $i - 1$ memory units[2] to compute the location of $A[i]$. We can then jump straight to the location of $A[i]$ in memory.

The downside to this convenience is that, after allocating $A[1..n]$, it is hard to adjust n. If we decide later to append an $(n+1)$th element, there is no clear place to put it. One can use pointers and start making a more involved data structure, but then we are giving up the direct memory access that makes arrays so appealing. Alternatively, we could make a new array $B[1..n+1]$ of size $n+1$, copy over the n elements from A, and store the $(n+1)$th item in the $(n+1)$th slot. But of course this takes $O(n)$ time to do the copying. And after that, what happens on the $(n+2)$th item? Do we copy everything over again?

[2]For lack of a better word.

compact1, compact2, compact3
java.util

Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

All Implemented Interfaces:
`Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `List<E>`, `RandomAccess`

Direct Known Subclasses:
`AttributeList`, `RoleList`, `RoleUnresolvedList`

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

(https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)

Figure 22.1: The beginning of the documentation for the `ArrayList` class in Java.

A slight modification of this last strategy, sometimes called the "doubling trick", is assumed by the ubiquitous `ArrayList` data structure in `Java` (see section 22.3) and by the built-in lists in `Python`. When we want to append elements beyond the capacity of $A[1..n]$, we allocate an array $B[1..2n]$ of *twice* the capacity, and copy the n elements of A over to the first n slots of B. See fig. 22.2 for pseudocode.

Now, it seems like common sense to allocate generously so that we don't have to copy as often. But in terms of analyzing the performance, we clearly don't have a $O(1)$ worst-case update time. To justify this approach, we turn to amortized analysis.

Theorem 22.4. *The array list supports constant time access (by index) to any element in the list, and can append an element to the list in $O(1)$ amortized time.*

We present three proofs illustrating three different methods of amortized analysis.

Direct proof. Over k append operations, the total time spent by the data structure is equal to $O(k)$ plus the total time doubling and rebuilding the array. We rebuild every

append(x)

/* *We assume the ArrayList data structure is defined by a* ***capacity*** m*, a* ***size*** $n \leq m$*, and an array* $A[1..m]$ *of which the first* n *entries are occupied by the* n *elements of the list, in order.* */

1. If $m = n$:
   /* *Double the capacity.* */
   A. Allocate a new array A' of size $2m$.
   B. Cope the n elements of A into the first n entries of A'.
   C. Replace A with A' and n with $2n$.
2. Set $n = n + 1$ and $A[n] = 1$.

Figure 22.2: Appending an element to an array-backed list, doubling the capacity when needed.

time the size of the list doubles, and rebuilding takes time proportional to the size of the list. This gives us a total running time of

$$O(k) + \sum_{i=1}^{\lceil \log k \rceil} \Big(\text{time to double in size from } 2^i \text{ to } 2^{i+1}\Big)$$
$$= O(k) + \sum_{i=1}^{\lceil \log k \rceil} O\Big(2^i\Big) = O(k).$$

So each insertion takes $O(1)$ amortized time. □

Proof by charging. Whenever we append an element, we first pay an additional 1 unit of credit worth $O(1)$ additional work later.

Now, each append takes $O(1)$ time plus $O(n)$ time whenever we rebuild for a list of size n. Whenever we need to rebuild a list of size n, the list has doubled in size since its last rebuild. Consequently we have at least $n/2$ credits to pay for the $O(n)$ time required to rebuild the array. □

Proof by potential method. We define a potential

$$\Phi = \max\{n - \lfloor m/2 \rfloor, 0\},$$

where m is the capacity of the array, and n is the number of elements in the list. Note that Φ is always nonnegative, and equal to 0 when the array is empty.

Recall that the algorithm doubles the array if $m = n$, to ensure there is space, and then adds the $(n+1)$th element. Consider a rebuild. Since $m = n$, we have $\Phi \geq n/2$. After doubling the array and setting $m = 2n$, we have $\Phi = 0$. So $\Delta\Phi = -n/2$, and this decrease in potential offsets the $O(n)$ time needed to build the new array.

After that, adding the $(n+1)$th element to the array takes constant time and increases Φ by 1. All put together, whether or not we rebuild, we have

$$(\text{running time}) + O(1)\Delta\Phi = O(1).$$

We now invoke lemma 22.2 and conclude that the array list supports $O(1)$ time per insertion. □

It may be desirable to extend our array-backed list with a `remove` operation, that removes the last element from the list. (Together, `append` and `remove` would define an array-based stack.) Removing the element is fairly easy to do; we clear out the corresponding entry in the area, and decrease our counter tracking the size of the list. Unfortunately, after removing many elements, we may end up with an array of far greater capacity than needed. We would prefer to keep the capacity of the array within a constant factor of the size of the list. This question is explored further in exercise 22.1.

22.4 Lazy search trees

The reader has likely encountered binary search trees (BST's) before and we give a brief overview. The idea is that we want to maintain a collection of ordered keys under insertions and deletions. Other auxiliary operations of interest include (a) retrieving the first key that comes before or after a query, (b) aggregate (e.g., sum) the values associated with all keys k in the range $a < k < b$, for given points a and b, and (c) list all the keys k in the range $a \leq k \leq b$, in order. We will assume the keys are distinct for simplicity though it is easy to adjust the ideas to accommodate duplicate keys.

Binary search trees arrange the data in a rooted tree. Each node contains one key and each key belongs to one node. For each subtree we maintain the following simple rule. Let k be the key at the root of the tree. Any key in the subtree that is $< k$ is placed in the left subtree. Any key in the subtree that is $> k$ is placed in the right subtree. This rule makes it easy to navigate the tree. Suppose we are looking for a particular key k. If the root contains k, then we are done. Otherwise we compare k to the key at the root and recurse on the left or right subtree appropriately. See fig. 22.3 for a diagram sketch, and fig. 22.4 for pseudocode.

It is easy to insert keys while maintaining the invariant. We search for where the key would be, if it were in the tree as a leaf. We place the key there as a leaf.

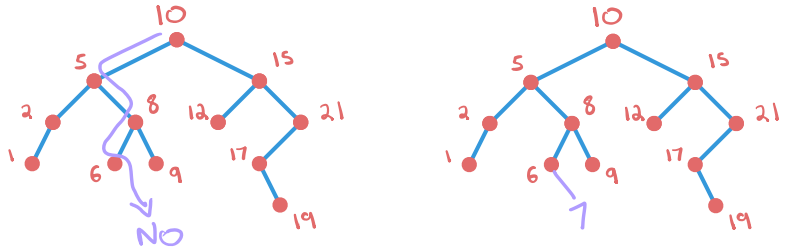
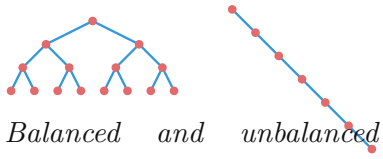


Figure 22.3: Searching for and inserting the key $k = 7$ in a binary search tree.



Balanced and unbalanced trees.

For both search and insertion, the worst case time to traverse a tree is entirely proportional to the height of the tree. Ideally, a tree would be *balanced*, where for each node, each subtree contains at most half of all the descendants. This would give a height of $O(\log n)$ for n keys, which is the minimum possible height. Given the list of keys already in sorted order, it is easy to build a perfectly balanced binary search tree with height $O(\log n)$. But when the set of keys are changing, it is easy to insert keys in an order that leads to a completely unbalanced tree. In the worst case the tree can have height $\Omega(n)$ and this ruins the running time of all our operations.

Thus considerable attention is paid to maintaining a balanced binary tree as keys are inserted and deleted. Many ingenious schemes have been invented to achieve this. Red-black trees are one such data structure sometimes included in introductory data structure classes. Splay trees are another ingenious approach analyzed later in **??**. Here we will describe and analyze a very simple and lazy approach that heavily leverages amortized analysis.

```
/* We assume each node x has pointers x.left and x.right to the roots of their left and right
   subtrees, respectively. Each node x represents a distinct key k stored at x.key, associated with
   a value stored at x.value. (The implementation here is immutable, making fresh copies of
   each node that is modified, and never writing over existing data.) */
```

`search(node` x`, key` k`)`

```
/* Return the value associated with key k in the binary search tree rooted by x. Return null if
   key k is not in the search tree. */
```

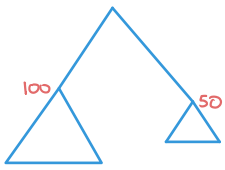
1. If x is `null` then return `null`.
2. If $k = x$`.key` then return x`.value`.
3. If $k < x$`.key` then return `search(`x`.left, `k`)`.
4. If $k > x$`.key` then return `search(`x`.right, `k`)`.

`insert(node` x`, key` k`, value` v`)`

```
/* Insert value v at key k in the search tree rooted at x, return the root of the updated search
   tree. If key k is not present in the tree, then z has the same key as another node y in the
   search tree, then we replace y with x. */
```

1. If x is `null` then return a new node y with y`.left` $=$ `null`, y`.right` $=$ `null`, y`.key` $= k$, and y`.value` $= v$.
2. If $k < x$`.key` then return a new node y copying x except with y`.left` $=$ `insert(`x`.left, `k`, `v`)`.
3. If $k > x$`.key` then return a new node y copying x except with y`.right` $=$ `insert(`x`.right, `k`, `v`)`.
4. If $k = x$`.key` then return a new node copying x except with x`.value` $= v$.

Figure 22.4: Implementation of a binary search tree mapping keys to values.



The high-level idea is very simple. We start from the standard (unbalanced) binary search tree, with the following adjustment. As we insert keys, whenever we notice that a subtree rooted at a node x is very unbalanced – meaning, one subtree of x has at least twice as many nodes as the other – then we rebuild the subtree as a balanced tree. We call this strategy *lazy rebuilding*.

We extend the code in fig. 22.4 to implement lazy rebuilding as follows. We augment each node with a counter that tracks the number of nodes in the subtree rooted at that node. We update the `insert` subroutine to update these counters as nodes are inserted. With these counters, each node can easily compare the difference in size of its two subtrees. Whenever we notice that, for some node x, one subtree of

x has at least half as many nodes as the other subtree, we rebuild the entire subtree rooted at x as a balanced search tree. Here we note that one can easily balance a tree with n elements in $O(n)$ time.[3]

In the following, we let n denote the number of elements in the binary search tree.

Theorem 22.5. *With lazy building, one can maintain a binary search tree of height $O(\log n)$ in $O(\log n)$ amortized time per insertion.*

Proof. Lazy rebuilding ensures that at any point in time, the number of nodes in one subtree is at most $2/3$ the number of nodes in the parent subtree. Therefore the tree always has height at most $\log_{3/2}(n) = O(\log n)$.

For each node v in the tree, let T_v denote the subtree rooted at v, and let $|T_v|$ denote the number of nodes in the tree.

We will define a potential Φ_v for each node v, and take the total potential as the sum of all these potentials. Let v be a node with children x and y. We define Φ_v as roughly absolute differences in size between the two subtrees T_x and T_y:

$$\Phi_v = \max\{0, |T_x| - |T_y| - 1, |T_y| - |T_x| - 1\}.$$

Above, the -1 term is for the case when $|T_x| + |T_y|$ is odd. If $||T_x| - |T_y|| = 1$, then this is as balanced as possible, and the potential is 0. Put another way, the -1 term ensures that $\Phi_v = 0$ iff v is a median over T_v.

We now define a global potential Φ as the sum of all the individual potentials,

$$\Phi = \sum_v \Phi_v.$$

Initially, when the tree is empty, $\Phi = 0$. More generally, we have $\Phi = 0$ iff the BST is perfectly balanced in the sense that each node represents the median of its subtree.

Claim 1. The part of insertion that comes before rebuilding $O(\log n)$ amortized time.

Because the tree has height $O(\log n)$, it takes $O(\log n)$ time to find the right leaf to place the key. The insertion increases the potential Φ_v by at most 1 for each of the $O(\log n)$ nodes v on the root-to-leaf path.

Claim 2. Rebuilding at an imbalanced node v takes $O(1)$ amortized time.

[3]For example, one can first traverse the subtree in order linear time, writing down the data in order in an array. With this array, one can split the array into subarrays by the median, and recursively continue to split the subarrays by their medians. Each median represents the root of a subtree; each subarray represents a subtree. Choosing the median ensures it is balanced.

Suppose v has k nodes in its subtree T_v. Then v's potential, Φ_v, must have been at least $k/3 - 1$. Optimally rebalancing the subtree T_v takes $O(k)$ time and decreases the potential Φ_x to 0 for every node in T_v, included v. The decrease in Φ_v alone pays for the time spent rebuilding.

Combining the two claims above gives $O(\log n)$ amortized time overall. □

22.4.1 Lazy deletions

Suppose now we wanted to support deletions as well. Here it is not entirely obvious what to do even in an old-fashioned BST. If we simply remove a node from the tree, unless it is a leaf, it creates a hole, and we have to scramble to fill it with some other node. One way to do this is to conduct another search to find the successor or predecessor node in the tree and somehow reattach subtrees appropriately. This approach (and others) can be quite messy and is also slow in the worst case.

Here we will analyze a much simpler and lazier alternative. We simply mark the node as deleted, and adjust the search routine to bypass keys that are marked for deletion. The remaining issue is that after many deletions, the vast majority of elements may be marked for deletion, which is inefficient. Let m denote the total number of nodes, marked for deletion or not, in the tree, and let n denote the number of unmarked (undeleted) keys in the tree. Our tree will have height $O(\log m)$, rather than $O(\log n)$, and our space usage will be $O(m)$, rather than $O(n)$.

We want to keep the tree size m within a constant factor of the "true size" n in a lazy fashion. We maintain a counter for the number of nodes marked for deletion; whenever it exceeds $m/2$, we clean up everything by deleting all the marked nodes and rebuilding the entire tree in $O(m) = O(n)$ time. These operations maintain the following invariant.

Lemma 22.6. *At any point in time, less than half the nodes in the tree are marked for deletion.*

Consequently we maintain optimal height:

Lemma 22.7. *At any point in time, the tree has height $O(\log n)$, where n refers to the number of keys that have not been deleted.*

Proof. By the same argument as above, the tree has height logarithmic in the total number of nodes (marked or unmarked). But the total number of nodes is within a constant factor of the total number of unmarked vertices. □

Theorem 22.8. *Lazy rebalancing and lazy deletions maintains a binary search tree with height $O(\log n)$ in $O(\log n)$ amortized time per insertion or deletion, where n is the number of keys in the tree.*

Proof. Let m denote to the total number of nodes in the tree, marked or unmarked. Let n denote the number of unmarked nodes; i.e., the number of keys represented by the search tree.

The lazy deletion keeps $n \geq m/2$ at all times; consequently $O(\log n) = O(\log m)$ and we do not distinguish the two quantities below.

Compared to the insertion-only analysis above, we continue to have the same per-vertex potential functions Φ_v, but here we should clarify that $|T_v|$ denotes the number of nodes (marked or unmarked) in the subtree rooted at v. We again let $\Phi = \sum_v \Phi_v$.

We also introduce a new potential function Ψ that counts the total number of vertices marked for deletion. The total potential is now $\Phi + \Psi$.

Each insertion takes $O(\log n)$ amortized time by the same argument as before; to recap, the part where we add a leaf to the BST increases Φ by $O(\log n)$, while the time spent rebuilding a subtree is paid for by the accompanying decrease in Φ.

Each deletion first takes $O(\log n)$ time to find the key to mark for deletion (because the height is $O(\log n)$), and then increases Ψ by 1. When we rebuild on account of deletions, Ψ is at least $m/2$. The rebuild takes $O(m)$ time and decreases Ψ to 0. So the decrease in potential pays for the rebuild. We conclude that deletion takes $O(\log n)$ time as well. □

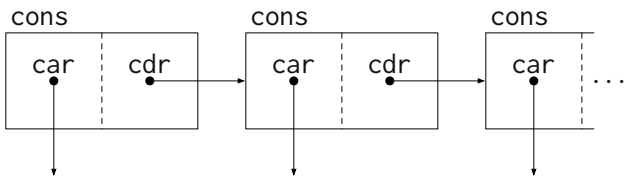
Remark 22.9. To support other miscellaneous tree operations, it is sometimes preferable to ensure that in every subtree, at most half the nodes are marked for deletion. To this end, we can keep a count of the number of nodes marked for deletion for every subtree, and rebuild the entire subtree whenever this count reaches half the total number of nodes in the subtree. This will still have $O(\log n)$ amortized time by a similar analysis as above.

22.5 Immutable data structures

In purely functional programming languages such as Haskell[4], all data is required to be *immutable*. Immutability is a fancy name for one simple rule: once something is written down, it can never be changed. This might seem restrictive, but assuming immutability can greatly simplify the reasoning in your programs. If you hand off the data to some function, you are promised that the function won't mess with your data. This is especially helpful in concurrent programming, when it is otherwise hard to keep track of all the changes made by other threads. Immutability also allows the compiler to be more aggressive when producing optimized lower level code.

[4] http://learnyouahaskell.com/chapters - have fun!

A (singly) linked list is inherently immutable. A linked list consists of `nodes`, each of which contains two pointers. One pointer is to the next node in the list, or `null` if it is the end of a list. The other pointer is to some piece of data that we would want to associate with the node. In `lisp`, a node is called a `cons` cell, the pointer to the data is called the `car`, and the pointer to the next `cons` is called the `cdr`.



To insert (`push` / `cons`) an element x to the beginning of a linked list, we create a new node that points to x and to the first node in the list. Henceforth the new node is treated at the beginning of the list. Observe that no changes were made in the existing list. In particular, any other pointers to the top of the list will not notice any changes.

Likewise we can immutably remove (`pop`) elements from the beginning of a linked list. Given a pointer to the beginning of a list, we just follow the pointer to the second element in the list, and use that as the top of the pointer to the list. Note that we didn't make any change to the data structure from an external point of view. We simply updated our own reference to the data structure, and the data structure is changed only from our own perspective.

Immutability means that when we want to change a small part of an object, we instead have to make an entire copy of that object that incorporates the small small change. This might seem wasteful. But often it is not so burdensome. Many object-oriented systems are composed of individually small objects with pointers to many others. Suppose we want to update an object that has some links to other objects. When we make a new version of that copy, we don't have to follow the links and recursively copy those objects as well. We simply copy over the pointers to the objects in the new version.

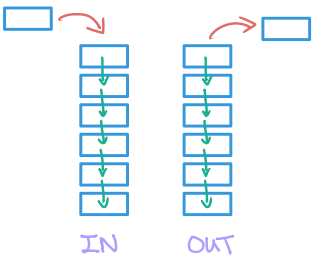
As a concrete example, consider the lazy search trees in section 22.4. They can easily be made immutable. The primary difference is that, when we insert a key at the bottom, we have to rebuild the nodes along the path from the root to the leaf. Note that we don't have to rebuild the subtrees hanging off the paths along the way. (Unless the algorithm calls for us to rebuild the entire tree, which we account for separately.) Fortunately for us, the lazy search trees ensure that the height of the tree is at most $O(\log n)$. So we only create $O(\log n)$ nodes. Each node is constant size, so this all takes $O(\log n)$ time.

A free benefit of immutable data structures is that you automatically produce snapshots of all previous states of the data structure.

22.5.1 Stacks and Queues

Recall that a *stack* is a data structure that allows us to insert and delete items strictly in *first-in-last-out* order. Namely, there are two operations, `push` and `pop`. `push` inserts an object. `pop` removes the most recently `push`'d object that had not yet been popped. Imagine a stack of plates. The linked list discussed above gives an immutable implementation of a stack.

What about a *queue*? A *queue* inserts and removes objects in *first-in-first-out* order. A singly linked-list will not suffice because we are inserting and deleting from opposite ends of a list. In fact it seems difficult to achieve constant (worst-case) update time. But we will relax our requirements and allow for constant *amortized* update time. We challenge the reader to try to figure how to implement an immutable queue with $O(1)$ update time before proceeding below the fold.



It turns out that there is a clever way to achieve this, using only two stacks (which are already immutable). Here are the ingredients.

1. Maintain two (immutable) stacks, called the input and output stacks.
2. Insert into the input stack.
3. Remove from the output stack.
4. If the output stack is empty, then replace the output stack with the input stack reversed, and set the input stack to be empty.

Theorem 22.10. *The two-stack queue implements insertion and deletion in $O(1)$ amortized time.*

Proof. We maintain a potential function Φ equal to the number of elements in the input stack. Each insertion takes $O(1)$ (real) time and then increases the potential by 1, so it takes $O(1)$ amortized time. Each deletion, excluding the part where we rebuild the output stack, takes $O(1)$ real time. Rebuilding the list takes $O(\Phi)$ time, where Φ is the number of elements in the input stack. Meanwhile the potential Φ decreases to 0. □

If we allow for amortized running times, then many data structures can be made immutable while retaining the same asymptotic performance. We refer the reader to [Oka99] for more examples.

22.6 Additional notes and references

Amortized analysis has long been used implicitly to analyze data structures and algorithms, and the general concept was more formally introduced by Tarjan [Tar85]. See [Eri13a], [DDL15, Lecture 5][5], and [Blu11a, Chapter 7] for additional notes on amortized analysis. Slightly different lazy rebuilding strategies for BSTs are described in [Eri13b].

Spring 2023 lecture materials. Click on the links below for the following files:

- Handwritten .pdf prepared before the lecture (and .note file).
- Handwritten .pdf annotated during the presentation (and .note file).
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

22.7 Exercises

Additional exercises can be found in [Eri13a].

Exercise 22.1. Recall the array-backed lists from section 22.3, where we showed how to support `append` in $O(1)$ amortized time via the "doubling trick". As discussed at the end of section 22.3, one may also want to include a `remove` operation that removes the last item in the list. To implement this, one could just unallocate the corresponding spot in memory, and decrease the counter tracking the size of our list. However, after many such removals, we may end up with an array that is mostly empty and much larger then the number of items in the list. It would be desirable to maintain the array to be within a constant factor. Here we will develop an analog of the doubling trick to facilitate deletions.

1. A first approach to addressing deletions might be as follows.

[5] https://www.youtube.com/watch?v=3MpzavN3Mco

> *After removing the element from the array and decreasing the size counter, if the size of the list is now less than half of the capacity of the array, allocate a new array of half the capacity and copy the elements of the list over.*

Unfortunately, this approach will not run in $O(1)$ amortized time. To prove this, devise a sequence of n `append` / `remove` operations (for n arbitrarily larger) on which the data structure would take $\Omega(n^2)$ time.

2. While the approach above didn't quite work, a slight modification of it will. Implement `remove` so that `append` and `remove` both take $O(1)$ amortized time. (Analyze these operations as well.) Your implementation should still follow the general format of allocating a new array and copying all the elements at certain points in time.

Exercise 22.2. Recall that a linked-list based stack supports the `push(`x`)` and `pop()` operations in $O(1)$ worst-case time. Consider the following operation, called `multipop`, that takes as input $k \in \mathbb{N}$, and removes and returns the top k items on the stack.

`multipop(stack` S`,` k`)`

1. $L \leftarrow$ empty linked list
2. For $i = 1, \ldots, k$ as long as S is not empty:
   A. Let $x = S$`.pop()`, and attach x to the top of the list.
3. Reverse L and return it.

Prove that after incorporating `multipop`, all three operations `push`, `pop`, and `multipop` each take $O(1)$ amortized time.

Exercise 22.3. Section 22.5.1 showed how to use two (immutable) stacks to implement a queue with $O(1)$ amortized performance. Here we will develop a more sophisticated, *double-ended queue* using only two immutable stacks.

A double ended queue is like a queue except we can insert and delete from both ends of the queue. Here we designate one end as the "front" and the other end as the "back", and define the following four operations.

1. `push-front(`x`)`: inserts element x at the front of the queue.
2. `push-back(`x`)`: inserts element x at the back of the queue.

3. `pop-front()`: removes the first element of the queue.

4. `pop-back()`: removes the last element of the queue.

(The data structure throws an error if the user calls either `pop` operation and there are no elements remaining.)

We analyze a data structure that maintains two stacks that we call the "front stack" and the "back stack". At a high-level, `push-front` and `pop-front` try to push-onto and pop-from the front stack, and `push-back` and `pop-back` try to push onto and pop from the back stack.

If we call `pop-front` and the front stack is empty, then we split the back stack in half (with the bottom half bigger, if the size is odd). The bottom half of the back stack is reversed and moved to the front stack. The top half of the back stack remains the back stack. The front stack is now nonempty, and we pop the top element.

Similarly if we call `pop-back` and the back stack is empty, then we move the bottom half (rounded up) from the front stack to the back stack.

Analyze the double-ended queue data structure by proving that each of the four double-ended queue operations take $O(1)$ amortized time.

Exercise 22.4. A *stacqueué* is a French list-like data structure supporting both the standard operations of a stack and a queue:

1. `dequeue`: removes an item from the beginning of the list.

2. `push` (a.k.a. `enqueue`): inserts an item at the end of the list.

3. `pop`: removes an item from the end of the list

Recall that a stack implements the `push` and `pop` operations in $O(1)$ worst case time. Design and analyze a stacqueué using only stacks that supports each operation in $O(1)$ amortized time.

Exercise 22.5. Implement `merge-sort` in $O(n \log n)$ time using only (immutable) stacks. Here the input consists of a stack of elements. The output is a stack of elements in sorted order (increasing from the top of the stack to the bottom.)