

We often don't want to find just a number, but an object associated with a number

Dictionary abstract data type:

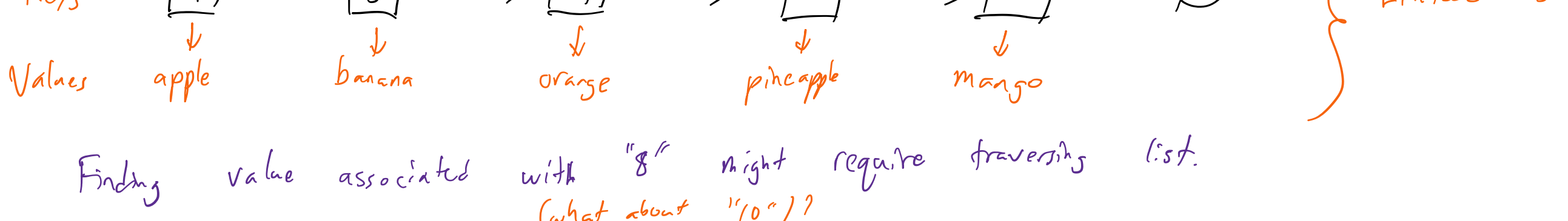
- insert (key, value)
- delete (key)
- value = find (key)

Ex. Linked list with attached values

Ex. Hash table " " "

Ex. Binary search tree

Many different ways to implement



Finding value associated with "8" might require traversing list.
(what about "10"?)

BST property:

Given a node (k,v), all nodes in left subtree have keys < k, in right subtree have keys > k.
(disallow duplicate keys for simplicity)

Binary search tree

every node has 2 children

attached items to "key"

rooted

Ex.

(sorted [1, 2, 3, 5, 7, 8, 9, 10])

Find (4)

L 4 < 5, L
L 4 > 2, R
L 4 > 3, R X
⇒ not present

Find (9)

L 9 > 5, right
L 9 > 8, right
L 9 < 10, left

Find:
if bigger, right
if smaller, left

Find Min()

L always walk left

Insert: Same as find, but add a child when you reach an appropriate empty spot. (otherwise, emit error on duplicate)

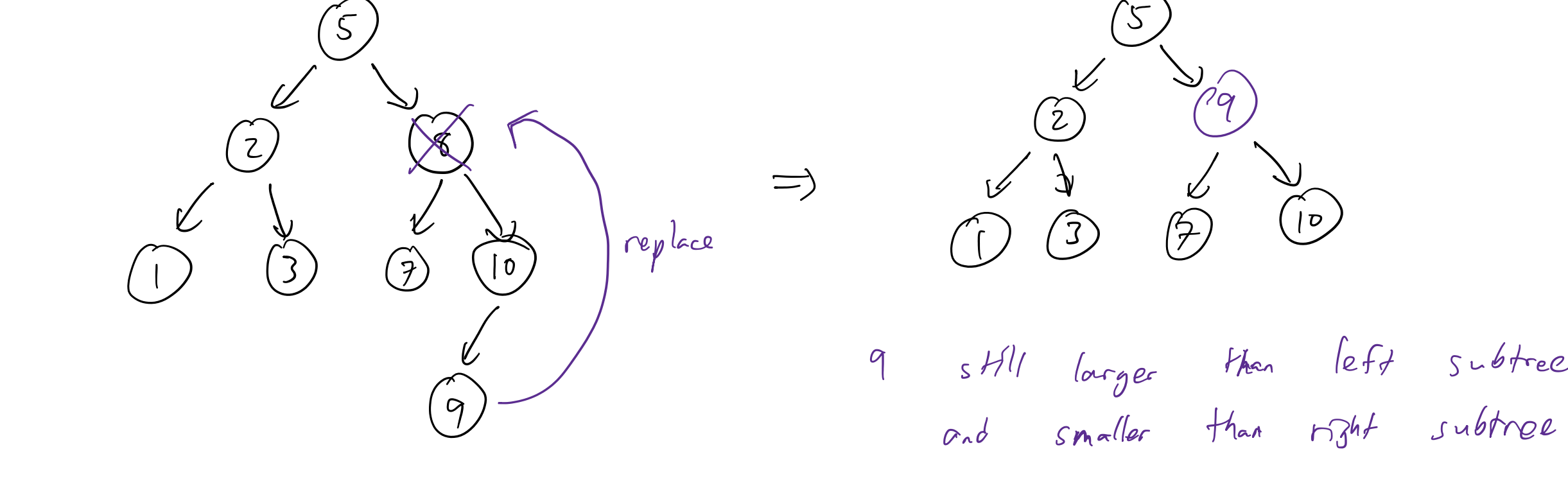
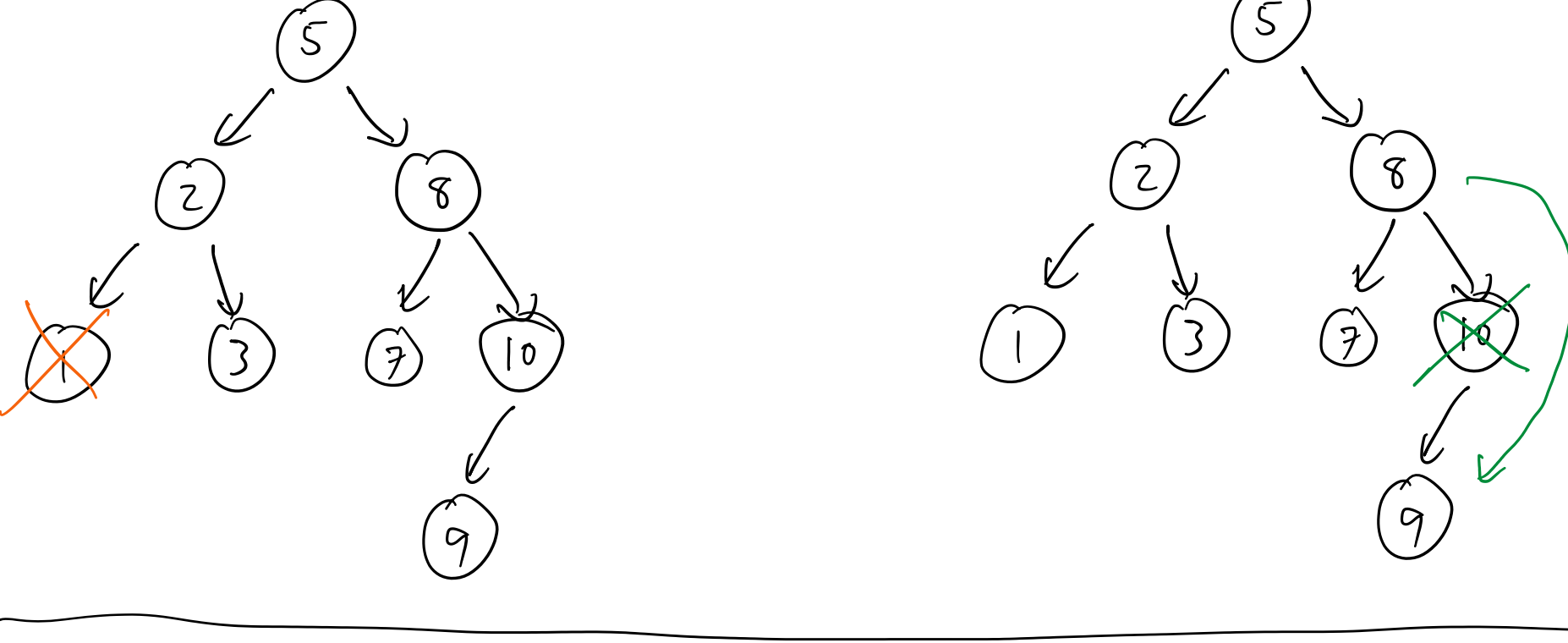
Delete:

Find node u with parent p

→ If u is a leaf, just delete

→ If u has 1 child c, delete u, and make c a child of p

→ If u has 2 children, find smallest node in right subtree of u, delete it, and replace u with it.



What can go wrong? Linearity

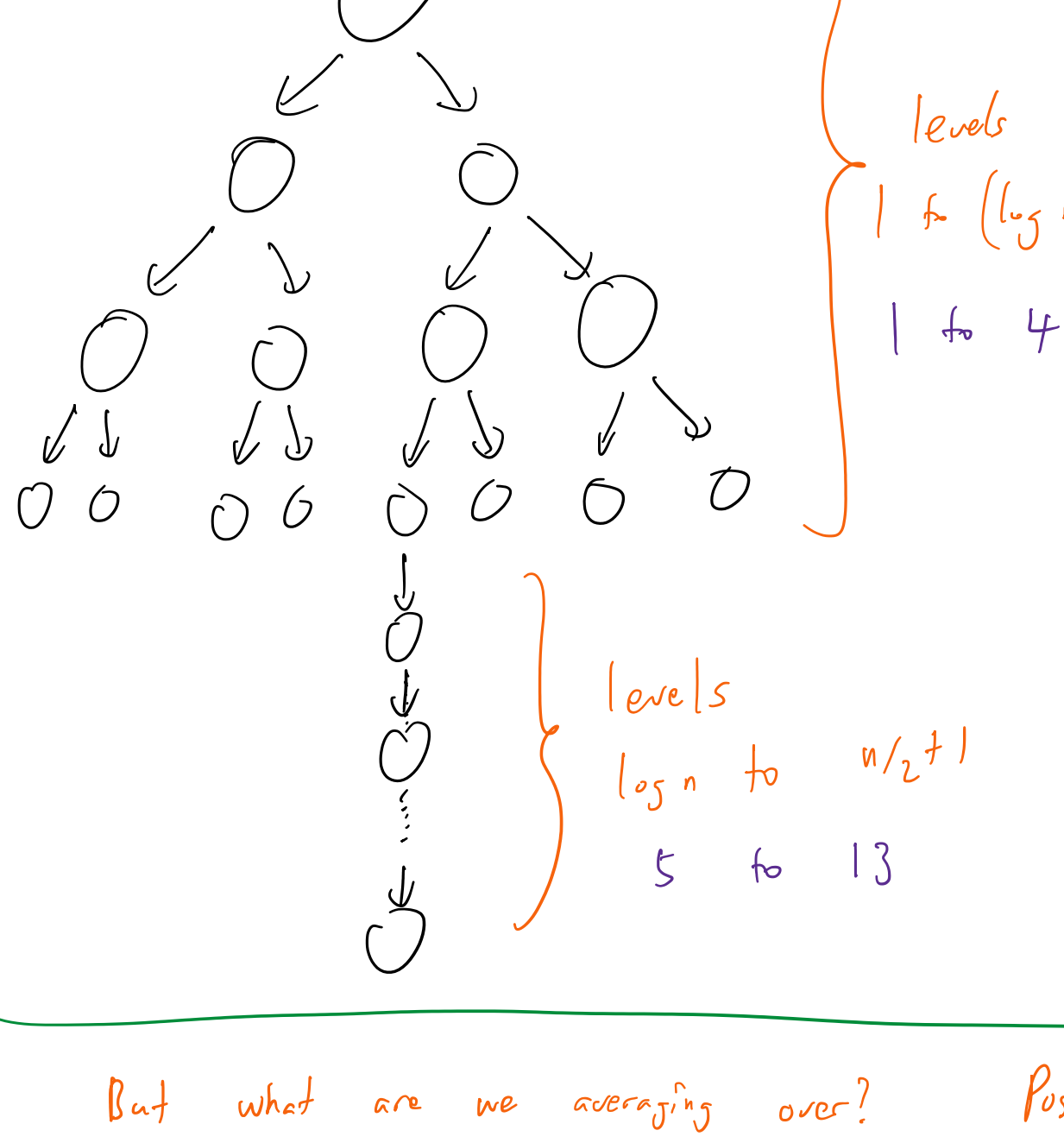
What would be optimal? Balance

How can we achieve balance?

↳ many options, such as AVL trees, red-black trees, splay trees, etc

Worst-case runtime

What's the worst that can happen?



24 nodes

15 in balanced complete tree

9 in long linear branch

$O(n)$

Worst-case: find goes to bottom (18)

Average-case: find is faster because don't have to go to bottom (but still $O(n)$ in this case)

But what are we averaging over?

Possible inputs? A probability distribution?

Ex. Looking at average value over

Rolling 10-sided die many times

randomized value

vs.

shuffling & dealing deck of 10 cards 1-10

amortized value

⇒ both have an expected average of 5.5.

Randomized: can get average of 10 if unlucky.

Amortized: can never get worst answer every time, though can occasionally.

Average is over "random" chance.

Average is over a constrained sequence of events

In-class demo: 10-sided die vs. deck of cards

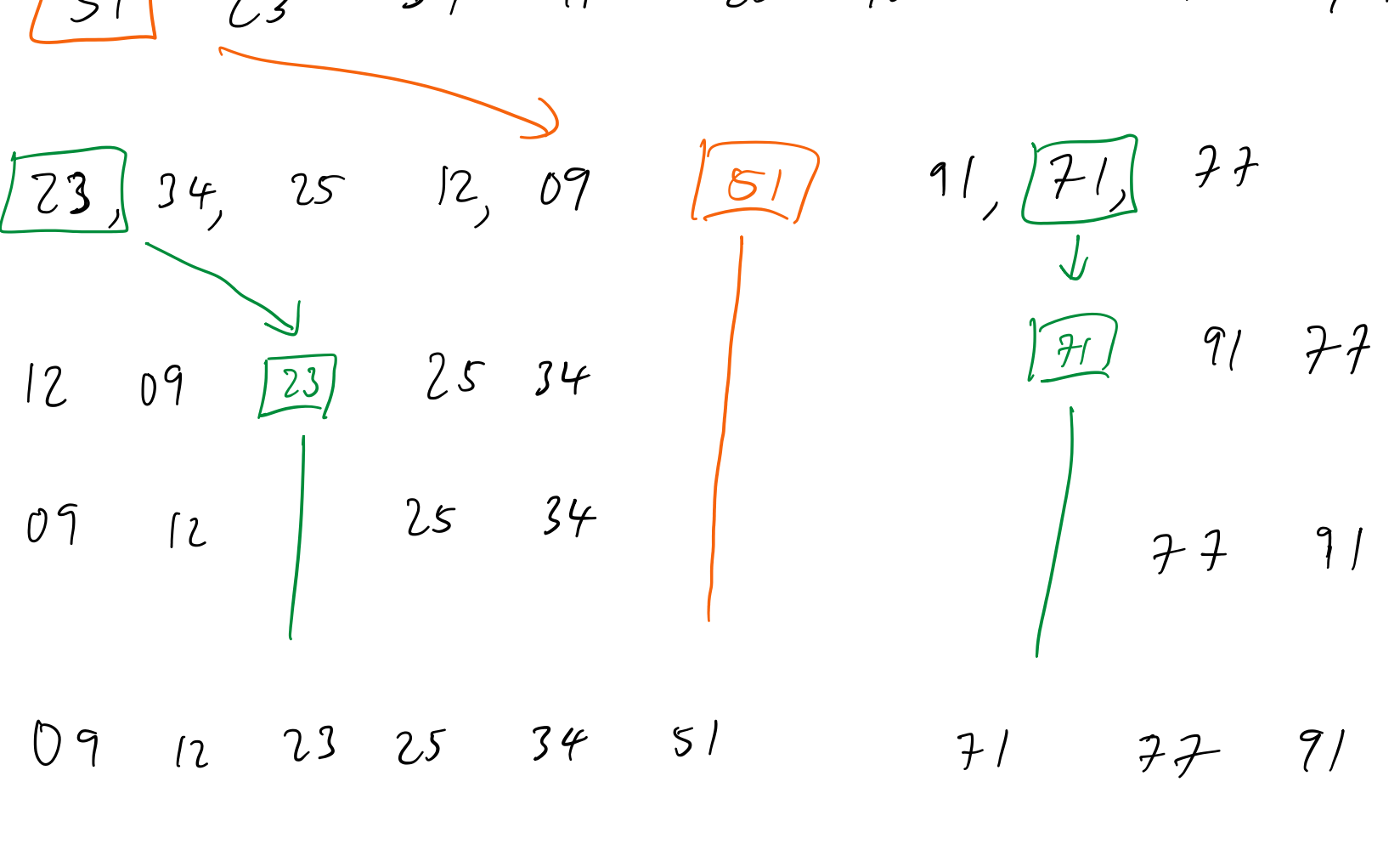
Expected-time complexity of randomized alg:

Given a known probability distribution we impose via "coin flips", what is the average amount of time an algorithm takes to run

Ex.

Quicksort:

Given an unsorted list, pick a random item as pivot, and divide into two smaller lists, one where everything is smaller, and one where everything is larger, & recurse.



Runtime: On average, each division creates two "half-size" lists.

Need $O(\log n)$ divisions.

Each iteration takes $O(n)$ comparisons to divide up set items

⇒ $O(n \log n)$ time (expected).

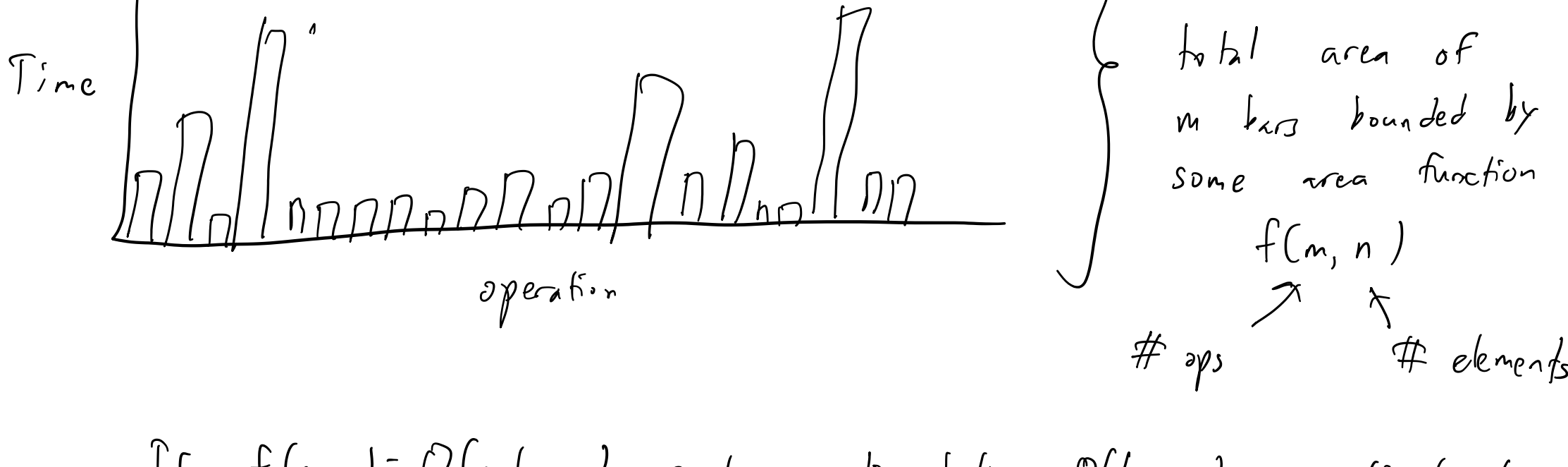
Worst-case: Always "accidentally" pick smallest item as pivot

Then need $O(n)$ divisions ⇒ $O(n^2)$ time.

Amortized time-complexity

• Average cost of operation over a series of operations

↳ some ops costly, some cheap



If $f(m, n) = O(m \log n)$, each operation takes $O(\log n)$ amortized time.

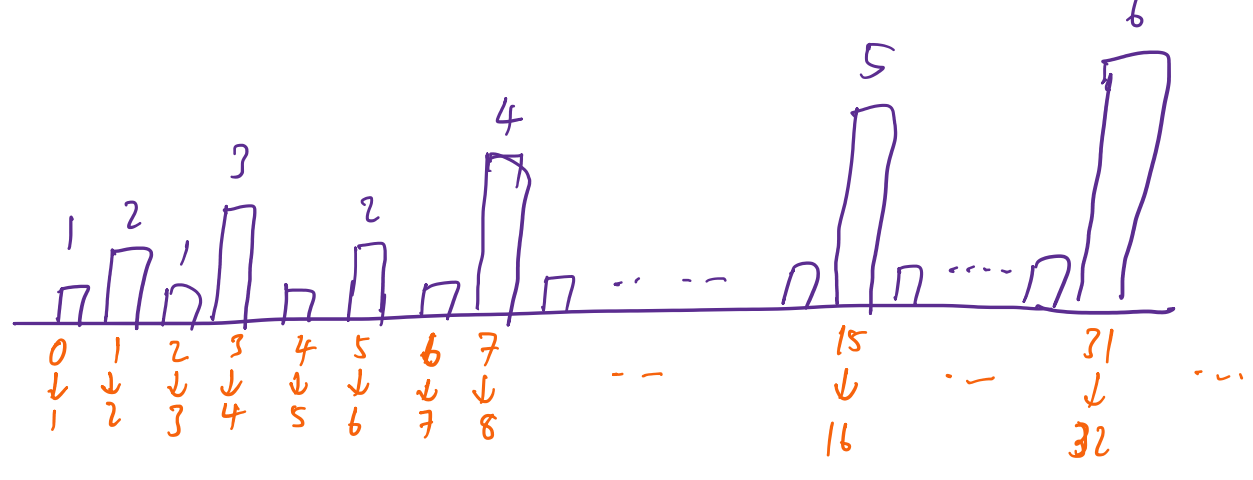
Ex. Binary counter: Cost to flip a bit is \$1

0: 00000000
1: 00000001
2: 00000010
3: 00000011
4: 00001000
...
33: 000100001

Worst-case cost to increment counter to k is

\$ $\log k$, since might need to flip all bits.

Amortized average case?



Claim: cost to increment counters from 1 to k is only \$2k.

Method: Prepay for 1→0 flips when we do a 0→1 flip.

i.e. when we flip a counter 0→1, pay \$1 to flip, and pay \$1 to store dollar on counter.

Then when we flip it back from 1→0, we have already paid for the flip back.

proof: On each increment, only one counter goes from 0→1.

Any other changing counters go from 1→0, so have already been paid for.

⇒ incrementing to k takes \$2k dollars.

□

⇒ amortized cost of bit flip is \$2, not \$ $\log k$.

Alt proof:

Counter 1 flips every increment, so k times

Counter 2 flips every 2nd inc, so $\frac{k}{2}$ times

" 3 " " 4th inc, so $\frac{k}{4}$ times

$\frac{k}{8}$ times

⋮

2k flips total.

□