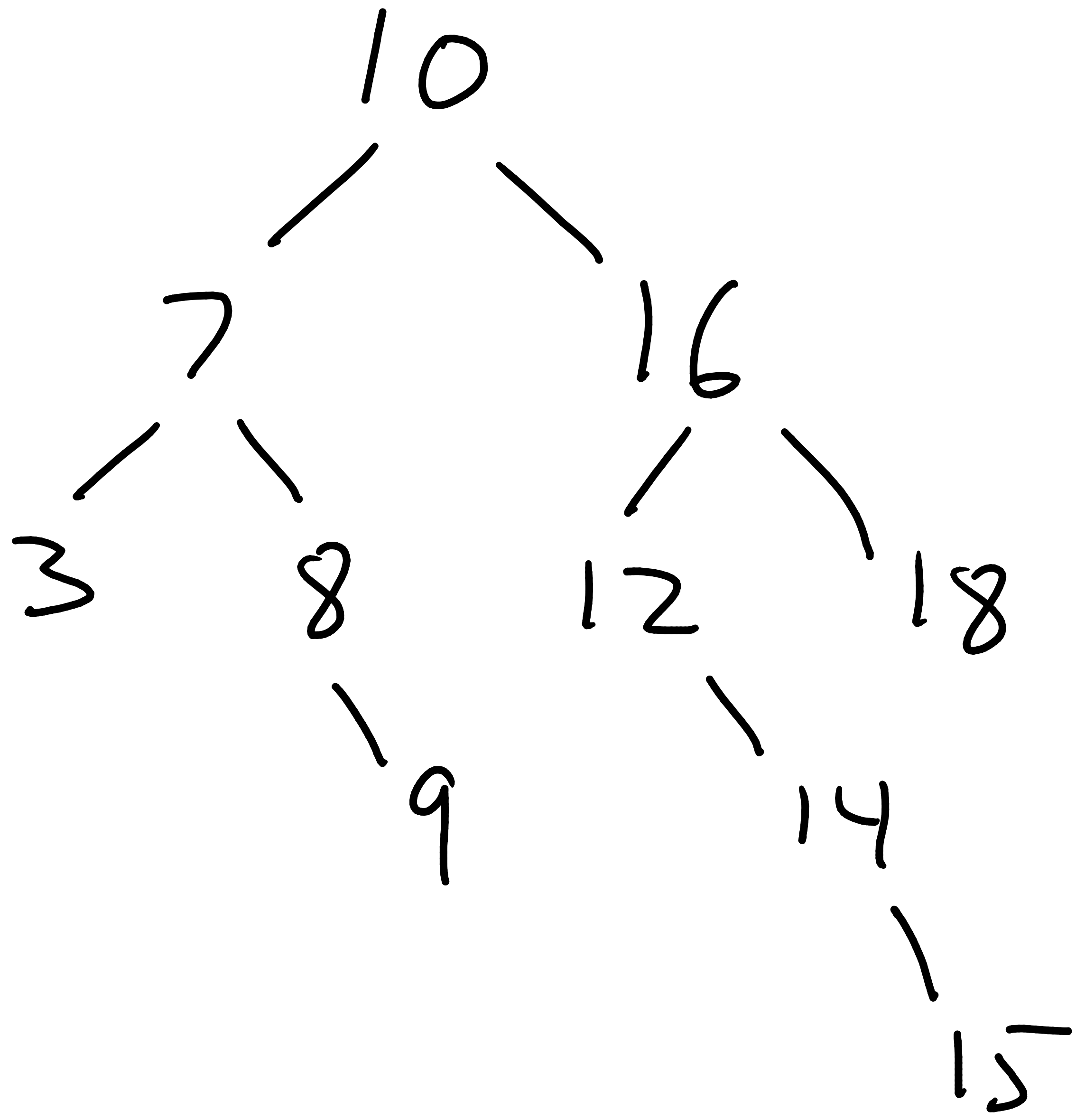
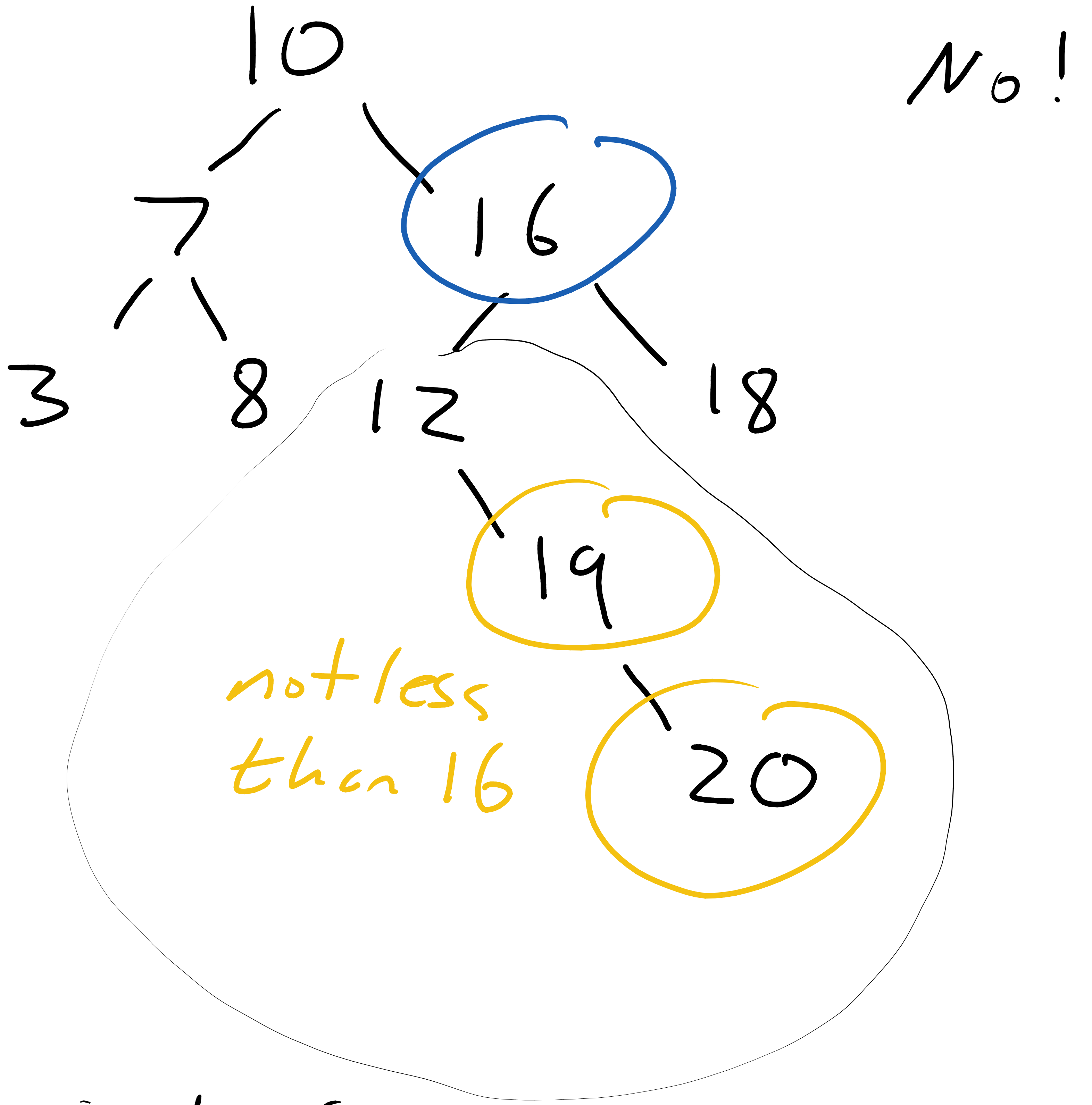
Binary search trees (BST)

at every node, all nodes to the left have smaller key values
right larger



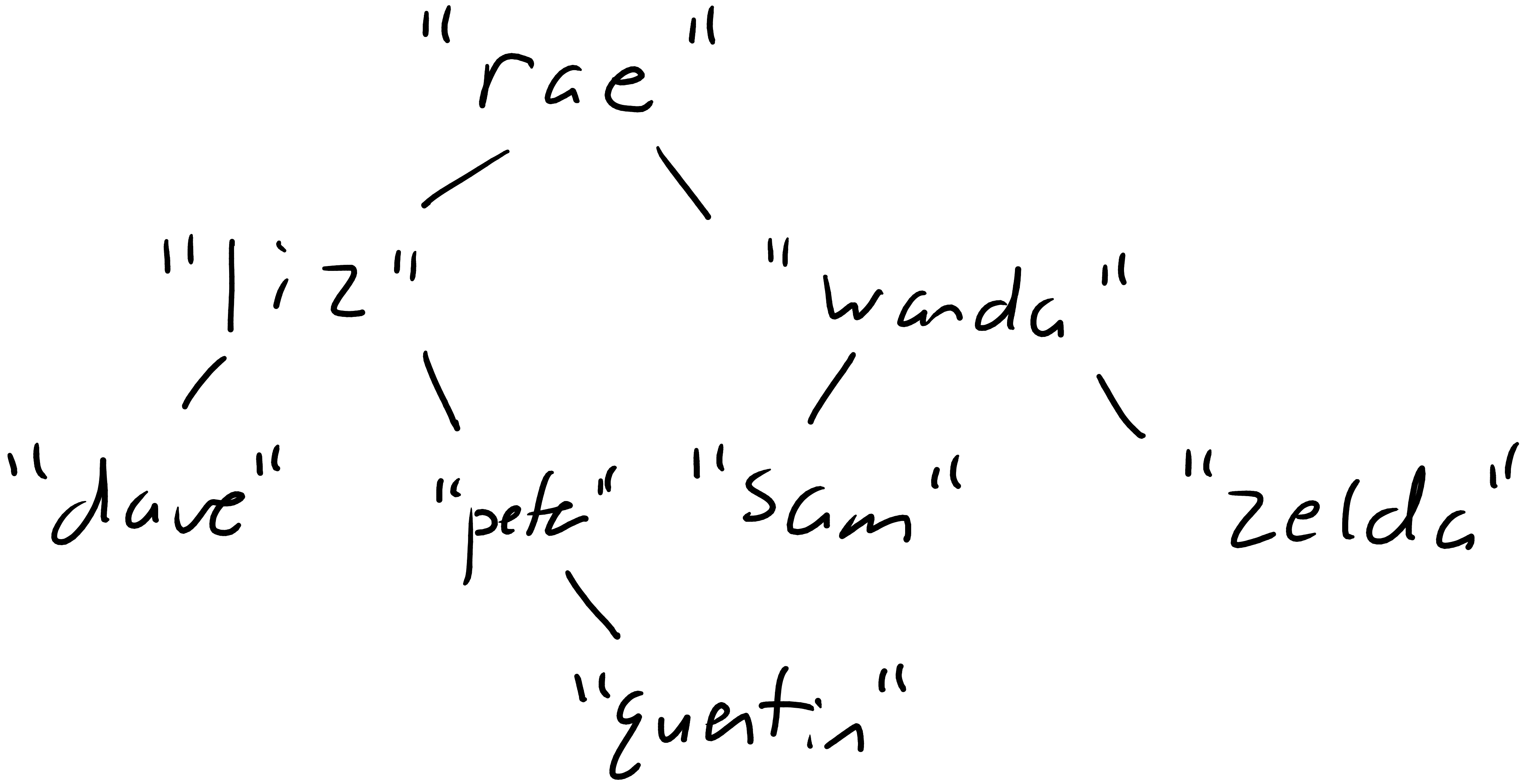
Is this a correct BST?



Is it just for numbes?

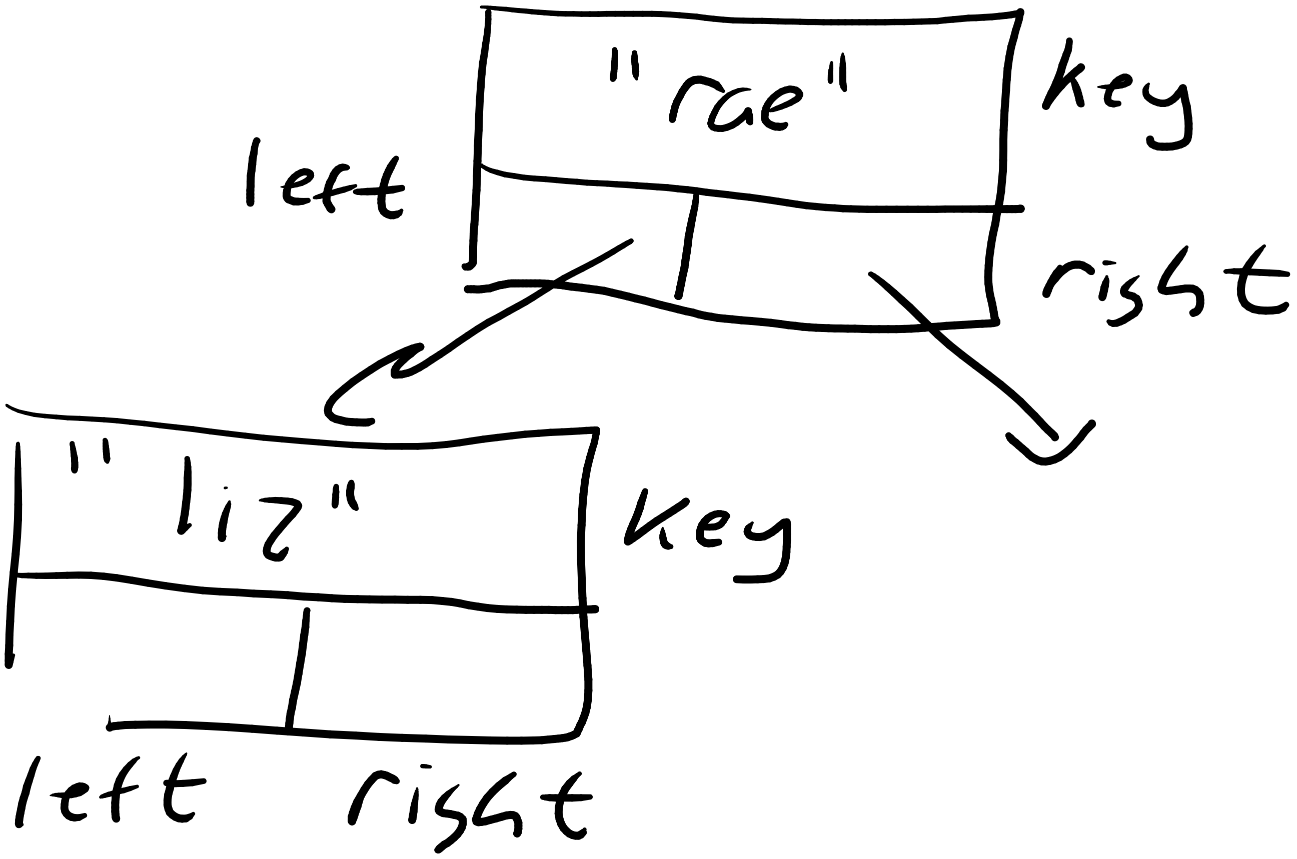
No, it's for anything you can order

Strings

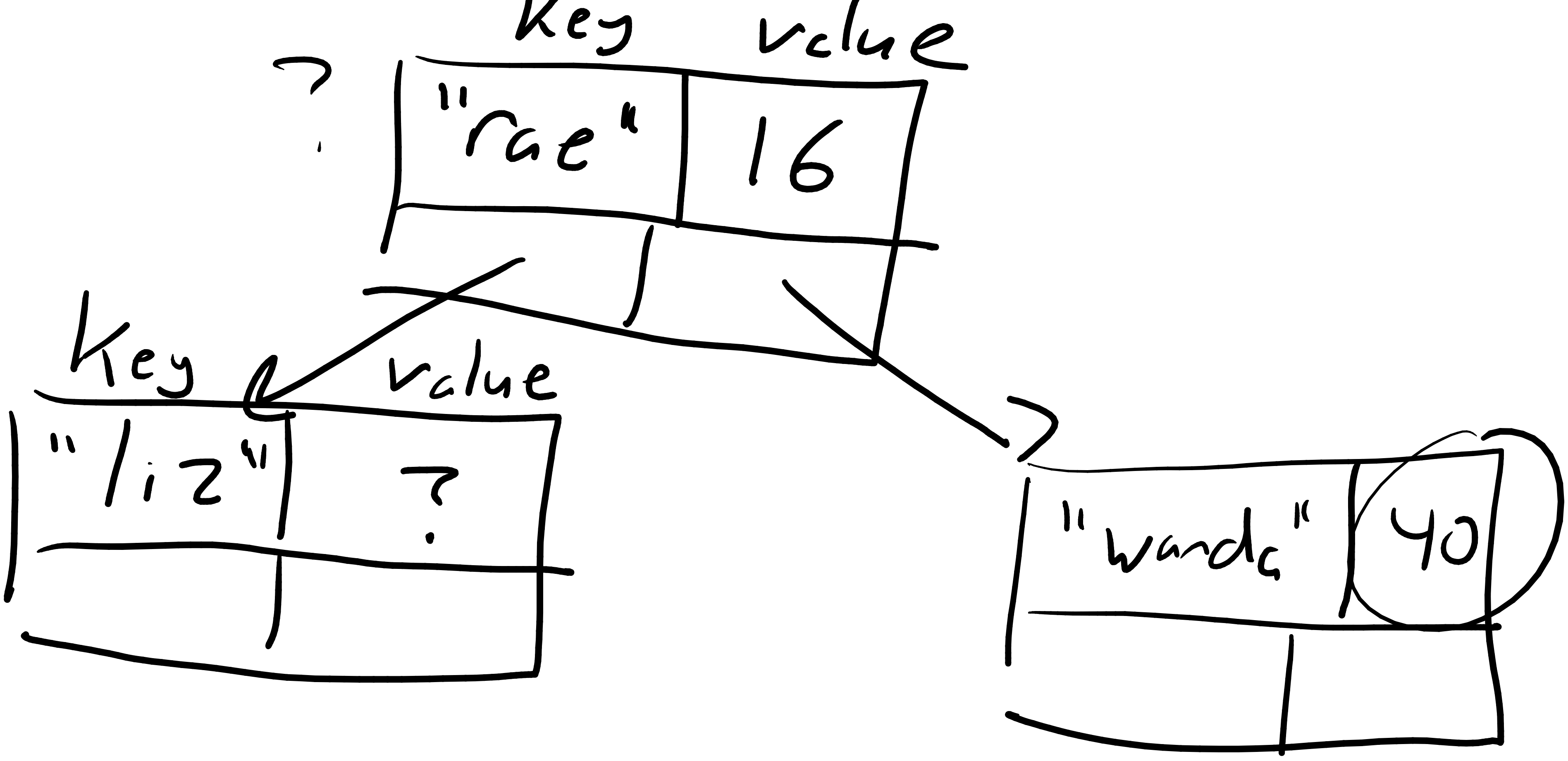


Why? Useful for sets and maps, just like hash tables (dicts)

For a map, just store a value next to the key



For a map, just also store a value

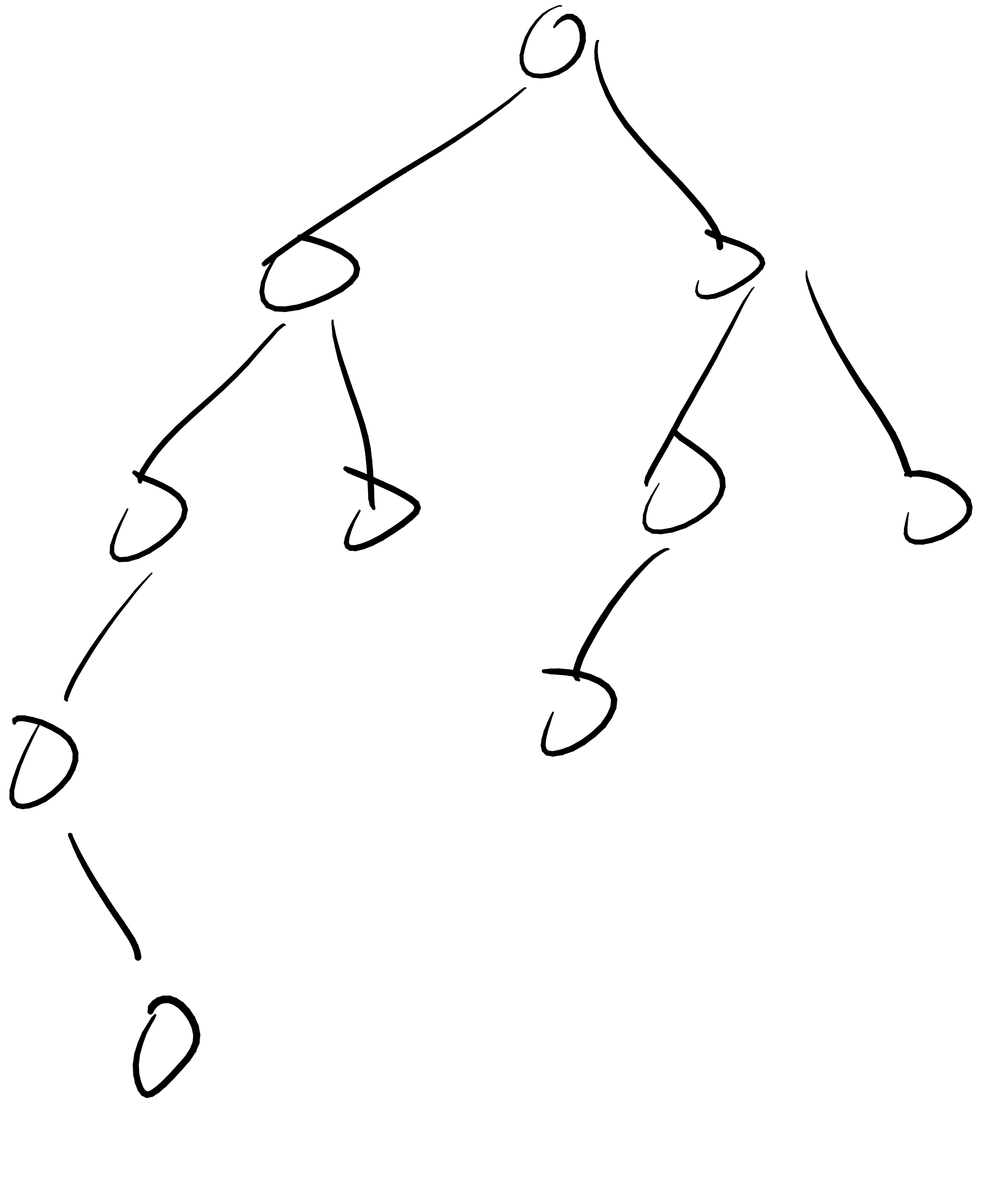


Lookup "wanda"

Performance

For n items, what is worst-case number of ops for insert/delete//lookup

- all 3 of these potaticlly involve going out to a leaf



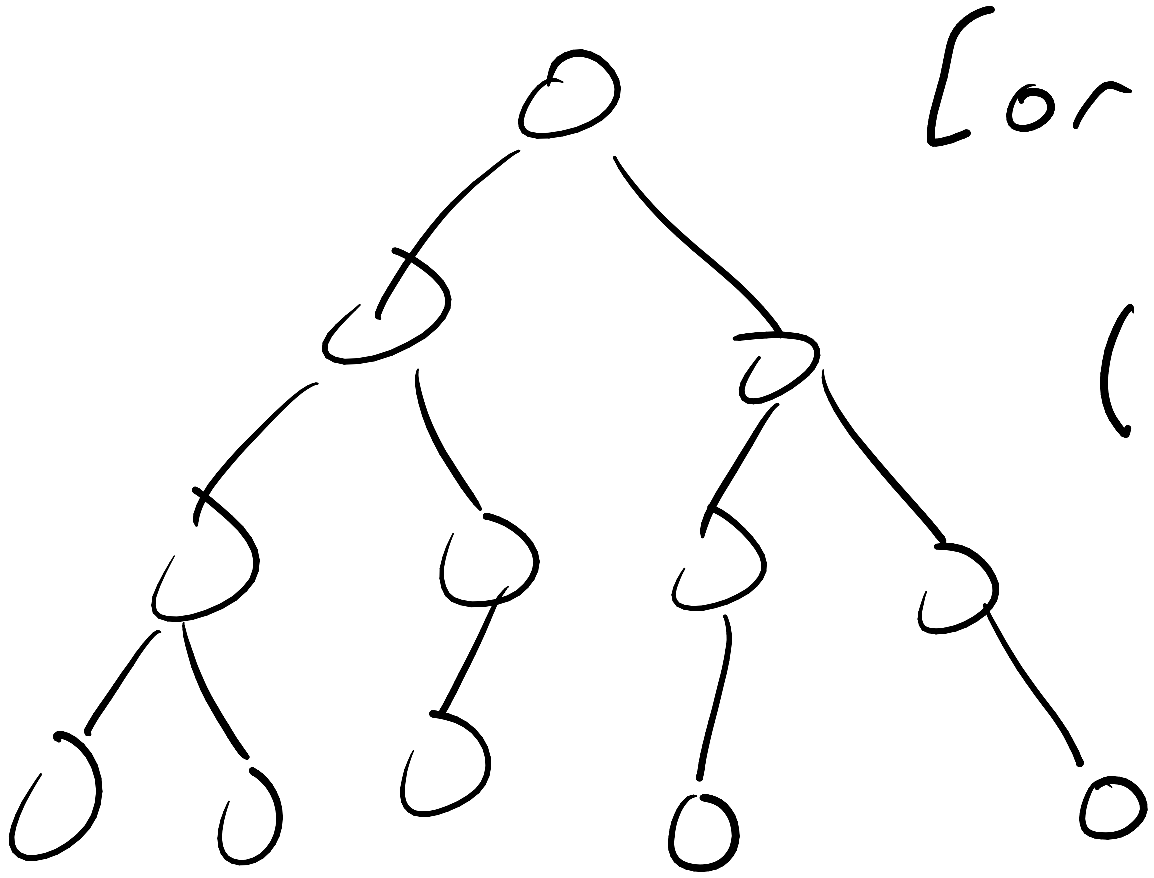
Worst case, how deep is tree?

Two cases:

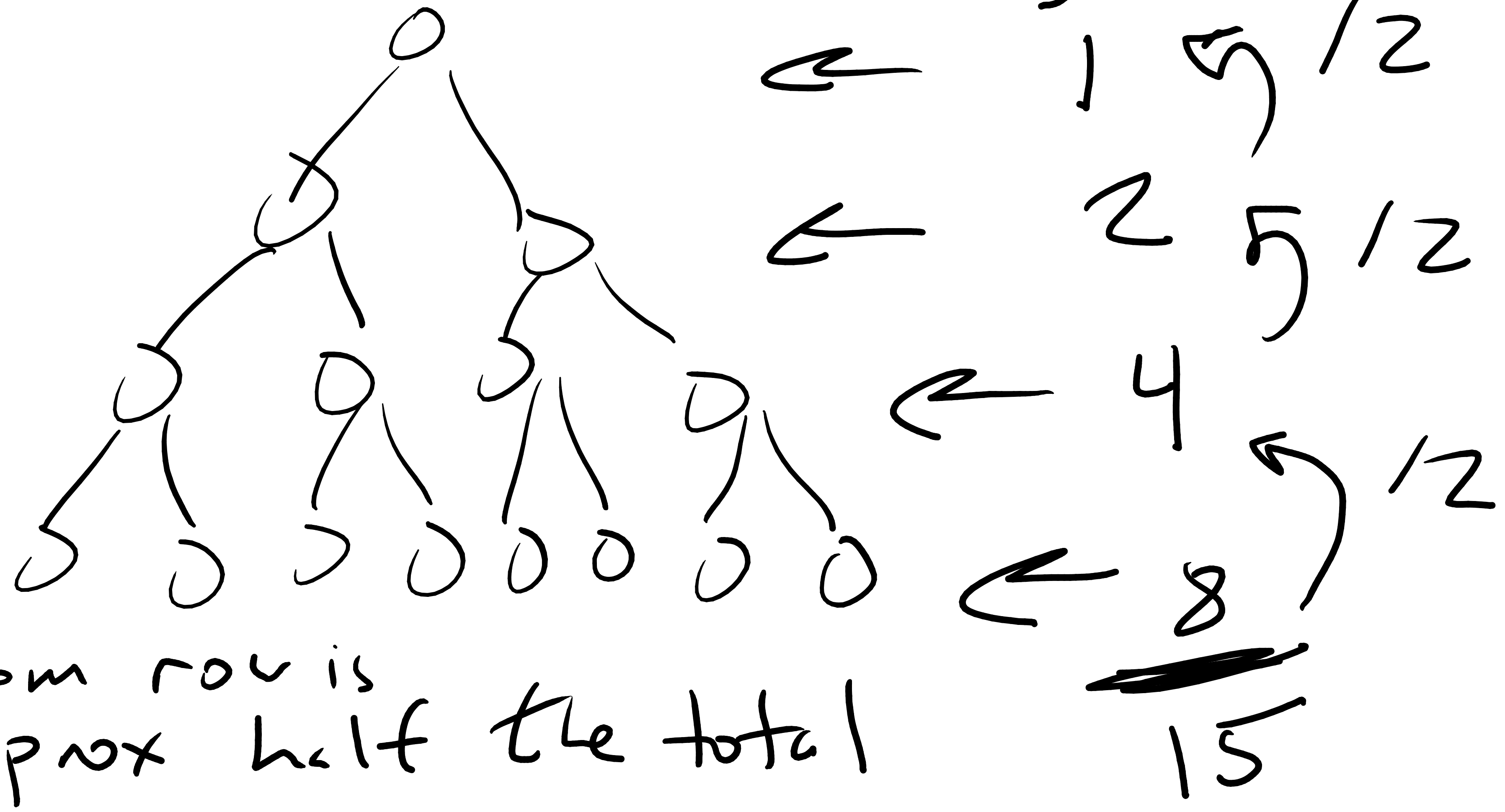
- if tree is balanced
- if tree is not balanced

Balanced: all leaves on same [or off-by-one] level

(oversimplified defn, not quite right)



If I have n items, approx how deep is the tree if balanced? (how many levels?)

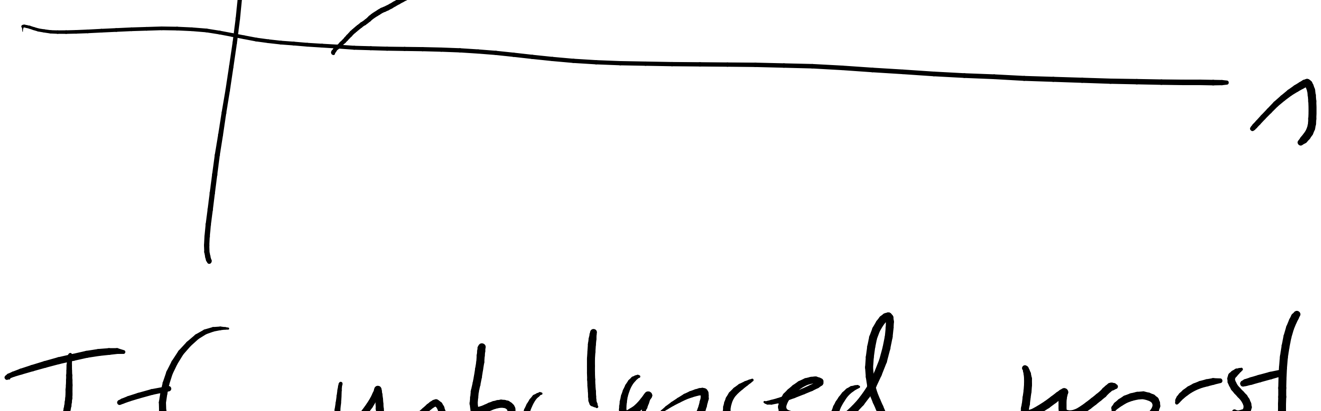


bottom row is approx half the total

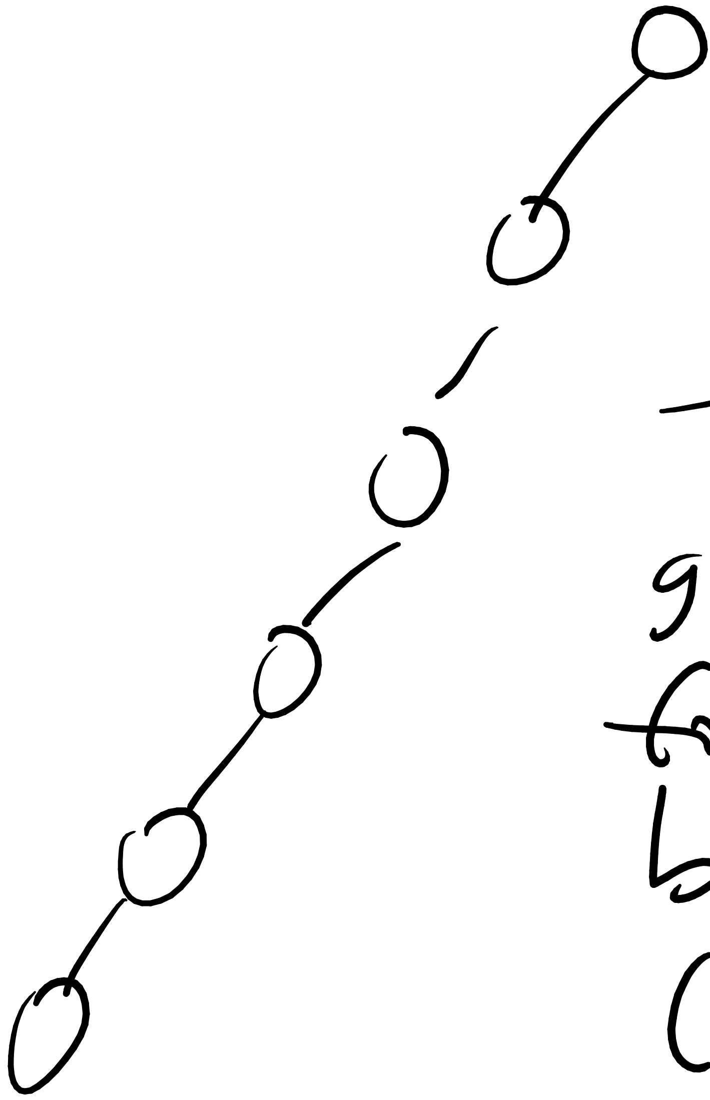
Numbe of levels is the numbe of times I can keep cutting n in half (approx) until I get 1

So in general, the height of tree is approx $\log_2 n$

Time to do insert/delete/lookup is $O(\log n)$



If unbalanced, worst case is a disaster.

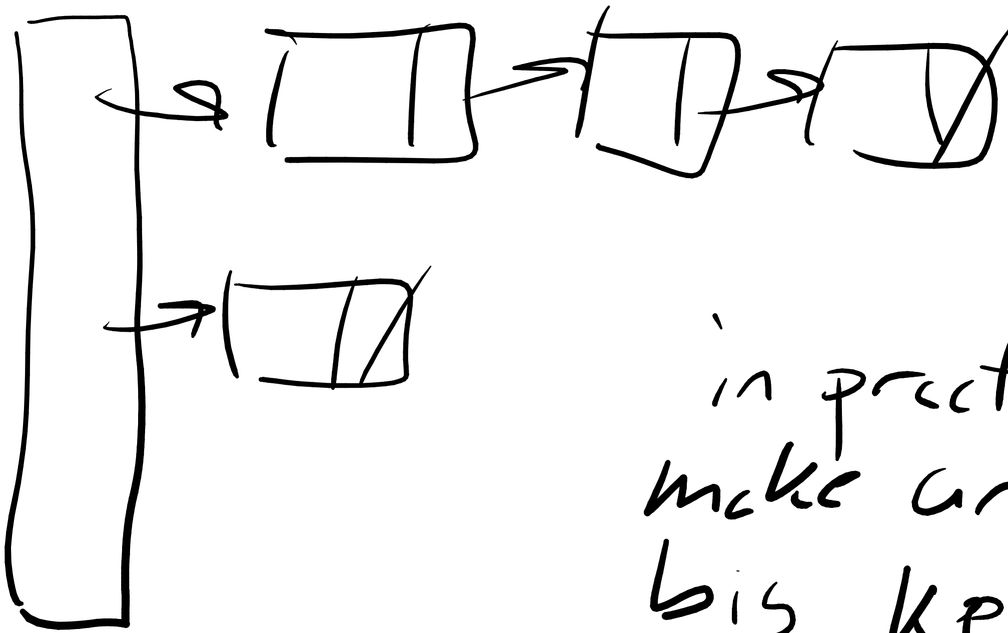


Worst case time is $O(n)$

There are lots of great approaches for forcing a tree to be balanced. [AVL trees] (not today)

In practice, you can get $O(\log n)$ performance.

How does this compare w/ hashing?



In practice

$$O(\lambda) \leftarrow \frac{\#\text{ of items}}{\text{size of array}}$$

in practice, make array big, keep λ small, $O(1)$

Hashing: $O(\lambda)$ ⇒ in practice, 2, or 3, always (just make array big enough)

BSTs, in practice $O(\log n)$

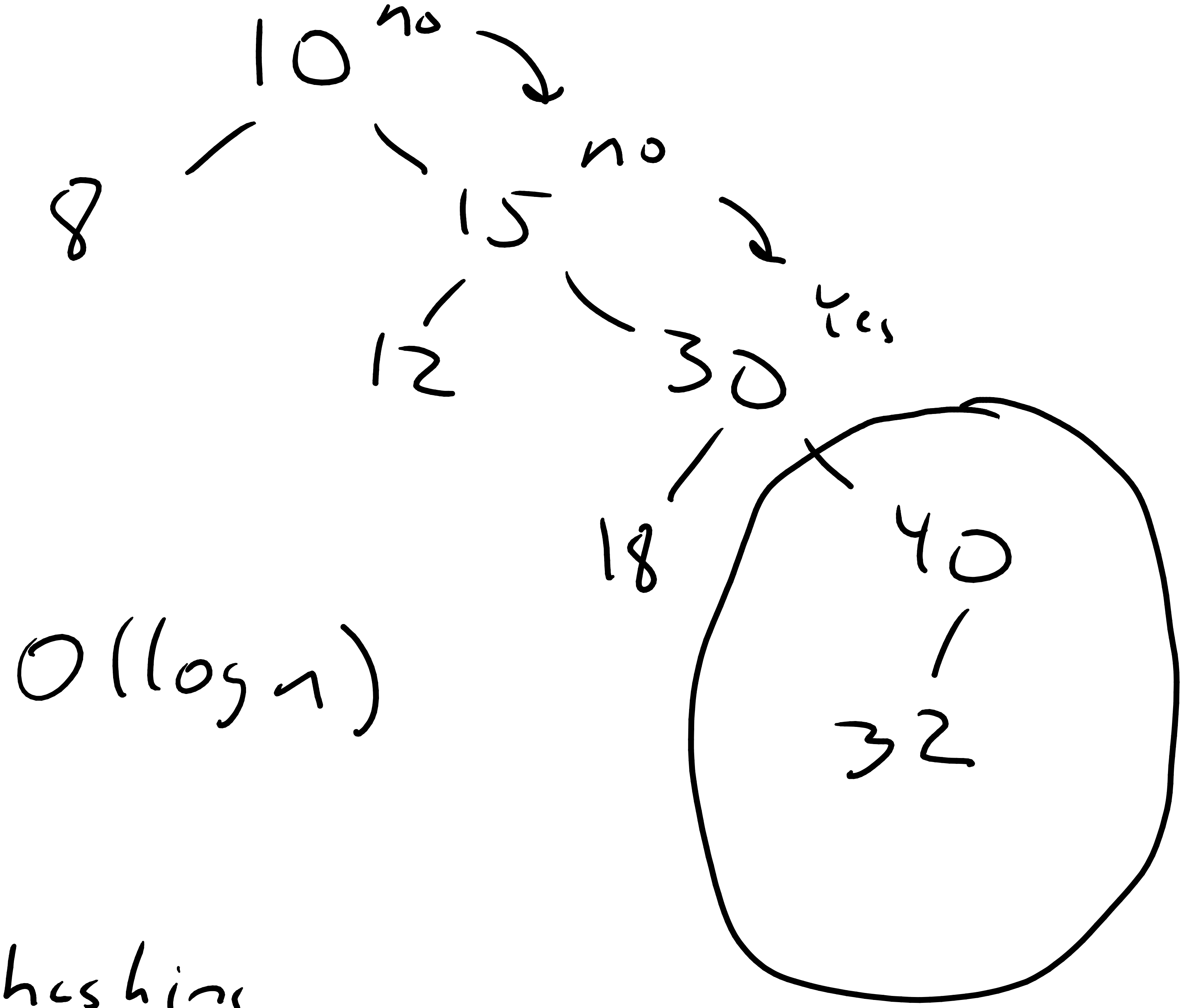
Hashing is definitely faster (when you do it right)
- you need to set the size of the array right

Real main advantage of BST:

"Get me all values for keys > 30"

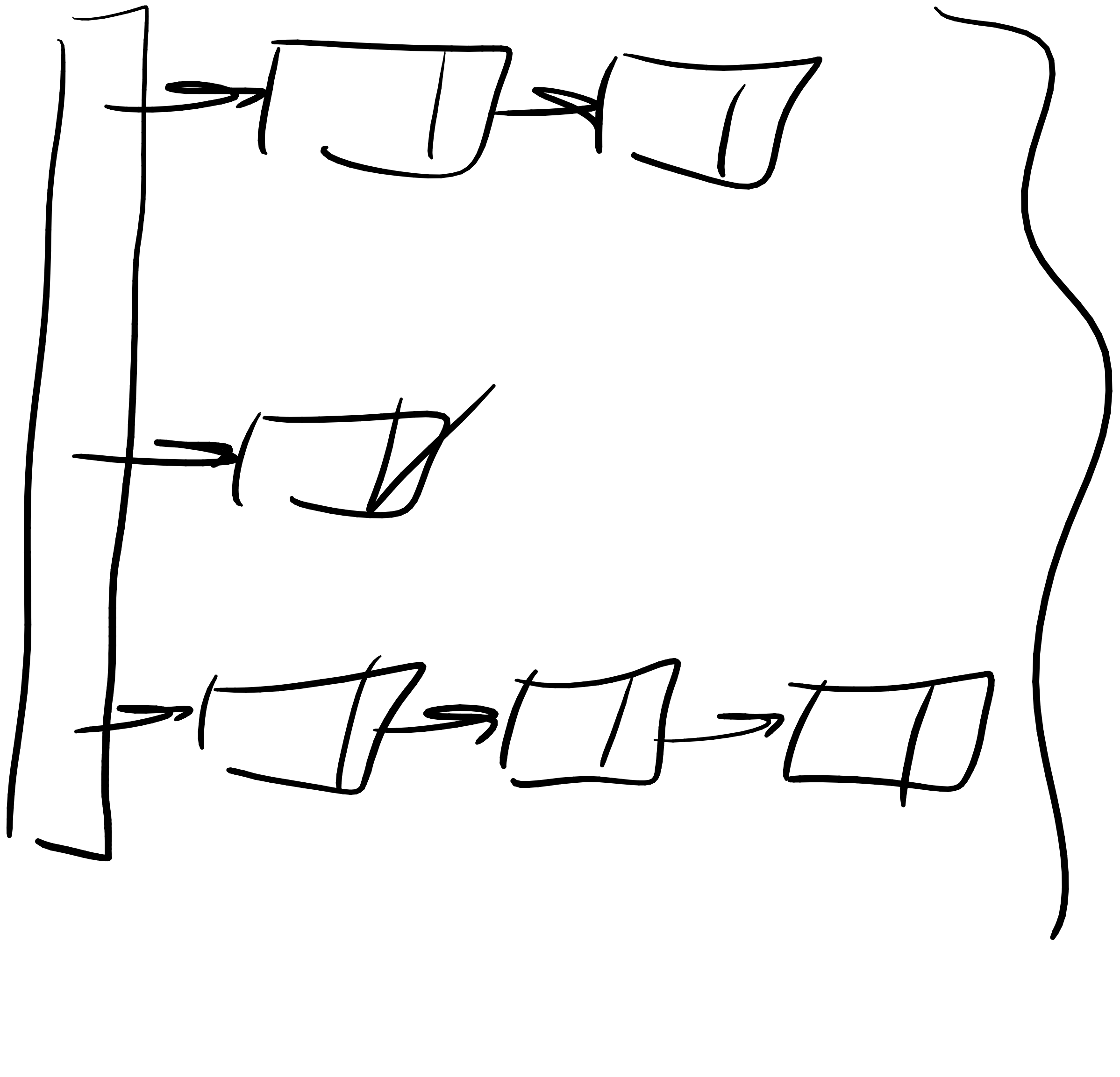
In a BST

Look for 30:



O(log n)

With hashing



have to look at everything

O(n)