Today:

- a bit more on BSTs
- (balanced) 2-3 trees (not in book)

Def: a tree rooted at x is a BST if for all nodes y in x's left subtree, $y.key \leq x.key$ & for all y in x's right subtree, $y.key \geq x.key$.

Building a BST:

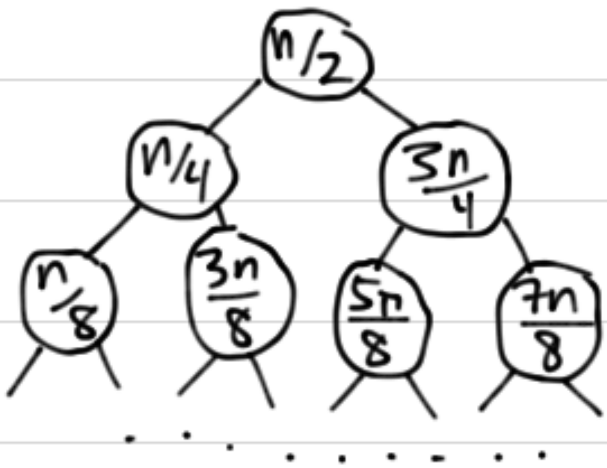
```
TREE-INSERT(x, z)   // assumes z.key set, z.left = z.right = NIL
    if z.key < x.key
        if x.left ≠ NIL : TREE-INSERT(x.left, z)
        else x.left = z , z.parent = x
    else <symmetric code>
```

Depending on order of insertions, tree can have many forms:

UNBALANCED (bad)

inserted 1,2,3,...

BALANCED (good)



Something to think about: how does the balance of the tree for a certain insertion order relate to the runtime of QuickSort for a corresponding sequence of pivots?

Keeping a tree balanced: we don't have to stick with the tree we get from a particular insertion order! Since most ops are $O(h)$ time, we want a balanced tree.

Balancing operation: "rotation".

Verify that this preserves BST property (and can be done in $O(1)$ time.)

If α is deeper than β and γ, then right rotation improves overall depth. If γ is deeper, then left rotation improves depth.

By carefully incorporating rotations into inserts and deletes, it's possible to guarantee $O(\log n)$ height and $O(\log n)$ operations (INS, DEL, SEARCH, MIN, MAX).

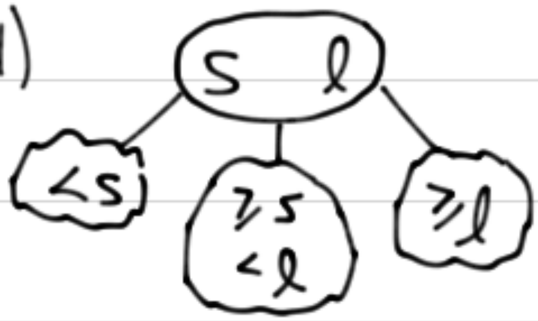
"Red-black trees" – in the book.

Another kind of balanced tree that's conceptually easier (but a bit harder to implement in practice).

"2-3 trees": no longer a strictly <u>binary</u> tree!

- "2 node" (internal – nonleaf)

- "3 node" (internal)

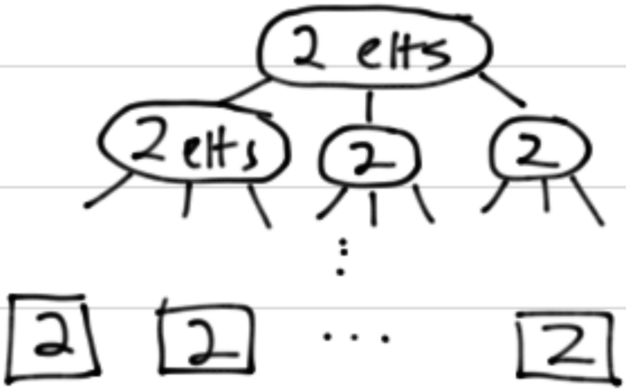


- leaf can have 1 or 2 elements
- A 2-3 tree must remain <u>perfectly balanced</u>: every root → leaf path must have exact <u>same length</u>.

TREE-WALK and -SEARCH are essentially the same as with BSTs; just have to deal w/ 3-nodes and 2-leaves.

How does <u>height</u> relate to <u># elements in tree</u>?

Max # nodes for height h:



$$2$$
$$2\cdot 3$$
$$2\cdot 3^2$$
$$\vdots$$
$$2\cdot 3^{h-1}$$

$$2\cdot\left(\frac{3^h-1}{3-1}\right) = 3^h - 1.$$

Min # of nodes is $2^h - 1$, just as in BST. So: $2^h - 1 \le n \le 3^h - 1$

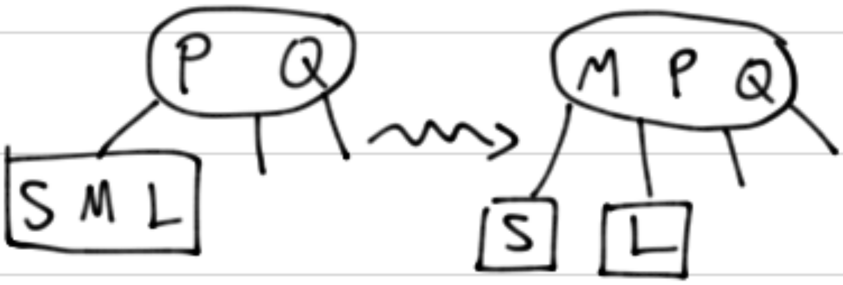
$$\iff \log_3(n+1) \le h \le \log_2(n+1)$$

Good: $h = \Theta(\log n)$. Now we just need to <u>maintain</u> the 2-3, fixed height property upon insertions and deletions.

INSERTION: follow root ⇝ leaf path and put element in a leaf. If leaf now has $\underline{3}$ elements ....

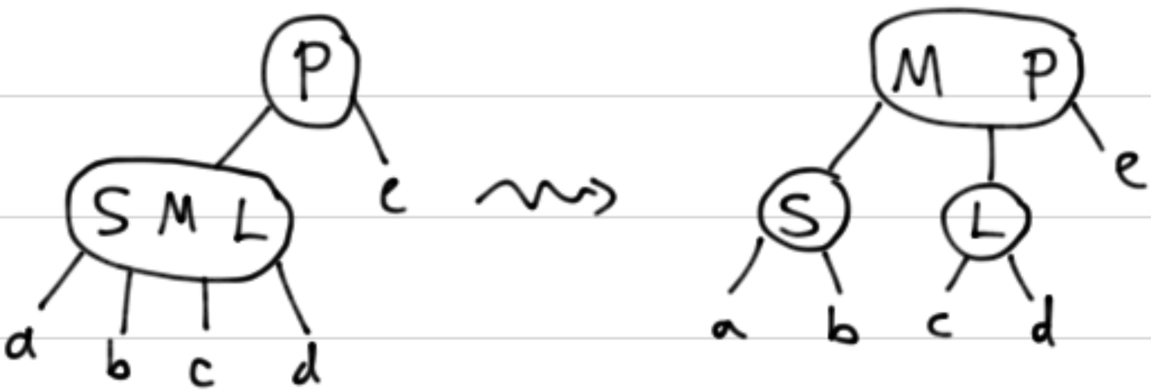
SPLIT leaf:

(similarly for right child leaf)



Uh oh! Node has 3 elts and 4 children, so ....

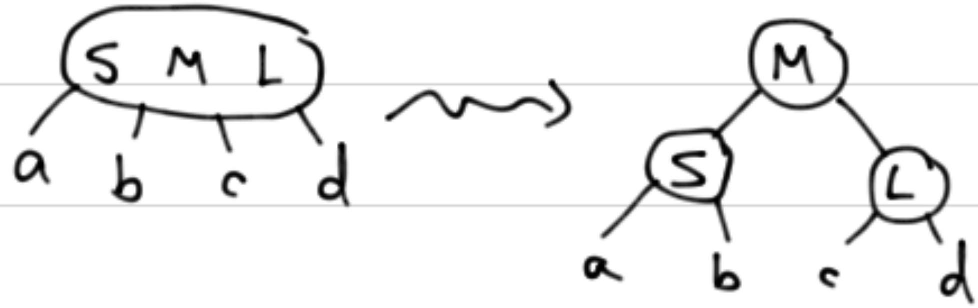
SPLIT INTERNAL NODE:



(same height)

If parent is P Q, proceed similarly but then <u>that</u> node will have 3 elts, so split it ....

SPLIT root:



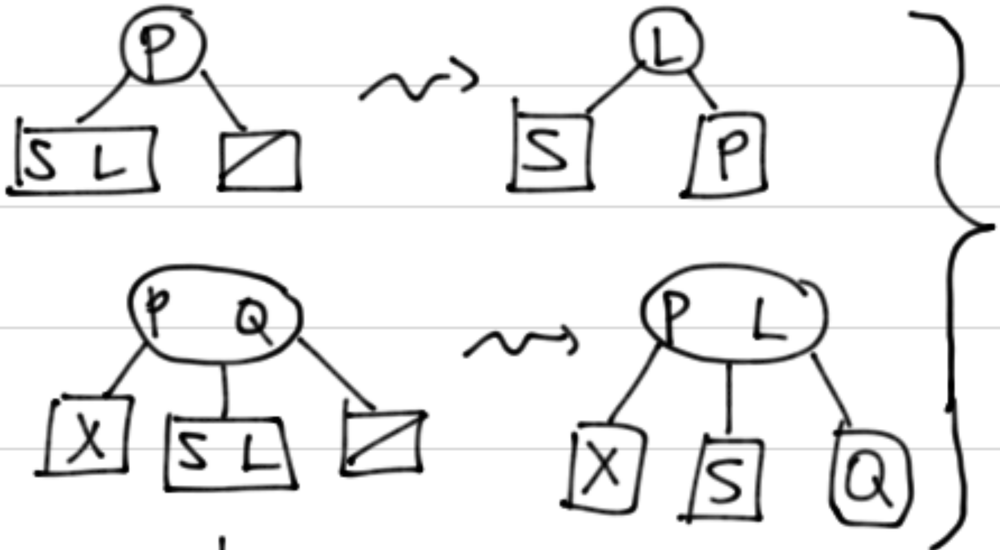
(height increases, but still perfectly balanced)

- Each split takes $O(1)$ time, and we do at most $h = O(\log n)$ per INSERT.

DELETION mainly reverses the process by MERGING.
One twist: we may be deleting an elt in an <u>internal</u> node.

Step ①: find elt. If in an internal node, swap w/ SUCCESSOR, which must exist, and be in a leaf because it's the MIN of some subtree.

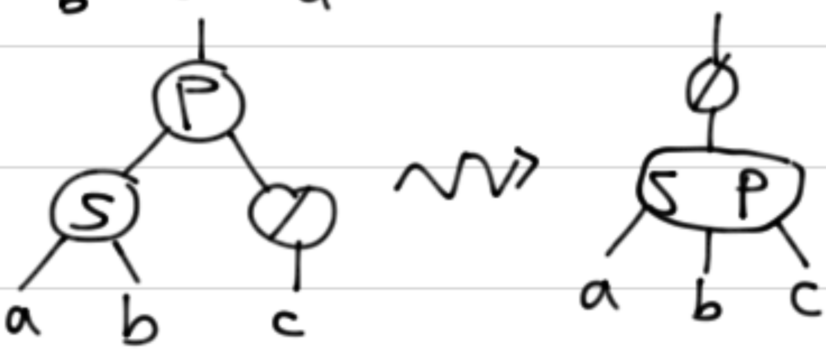
So now we're always deleting from a leaf.
If we leave a leaf empty, several cases...



etc-
redistribute
rules

percolate up
to handle empty
internal node...

handling empty
internal node



percolate up again.

ETC. ETC. 12 rules
in all ...

Bottom line: each merge/redistribute takes $O(1)$ time, and we do at most $h = O(\log n)$ per delete.

Therefore, 2-3 trees have $O(\log n)$ time operations where n is current # elts in tree.