

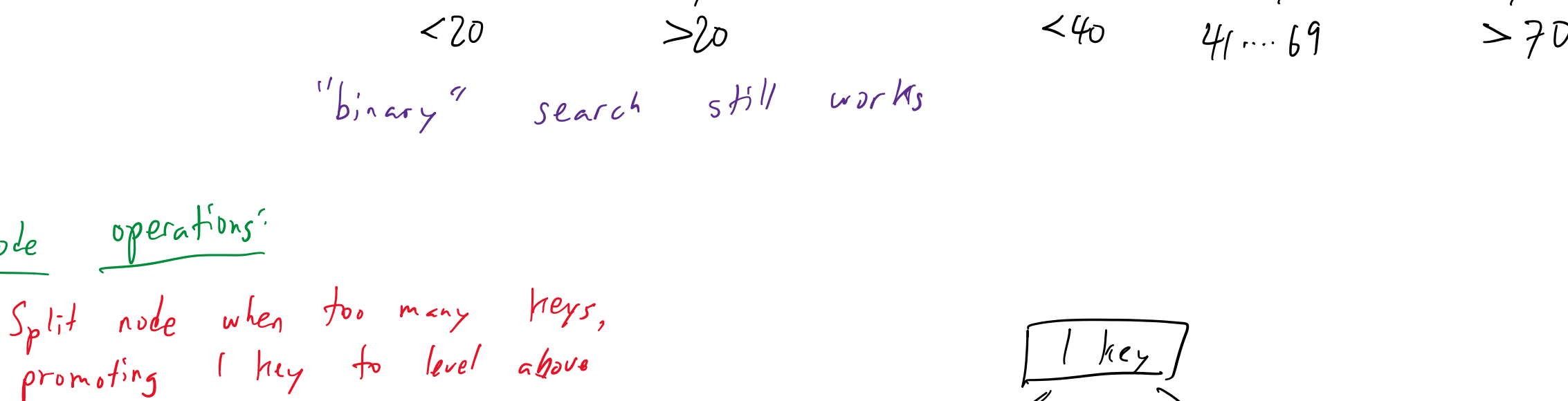
We just saw one way to "balance" out operations in the splay tree.

Another alternative is to force exact balance by varying node sizes.

a_b-trees

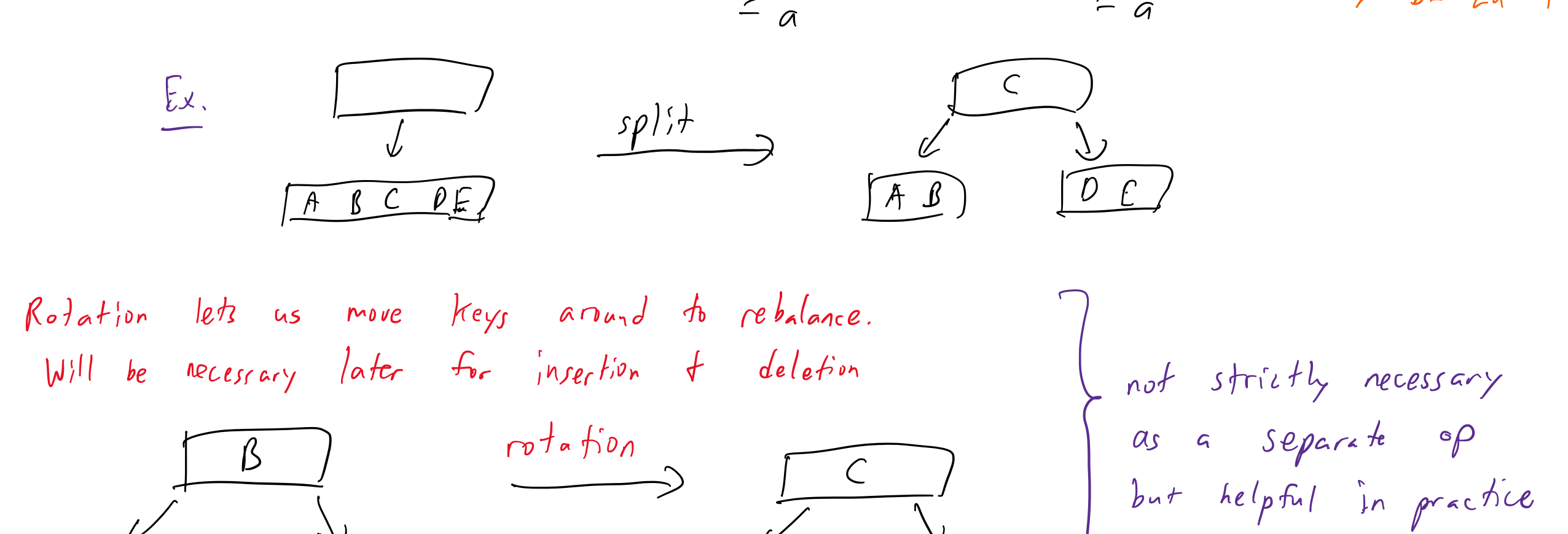
- All leaves at same level
- Internal nodes have x children, where $a \leq x \leq b$. Root has $b/4$ + b children.
 note $x-1$ keys in node (max $b-1$ keys)
- $a \geq 2$ (to branch at all)
- $b \geq 2a-1$ (enough children to make splitting work)

Search tree prop: keys within node split range of child keys

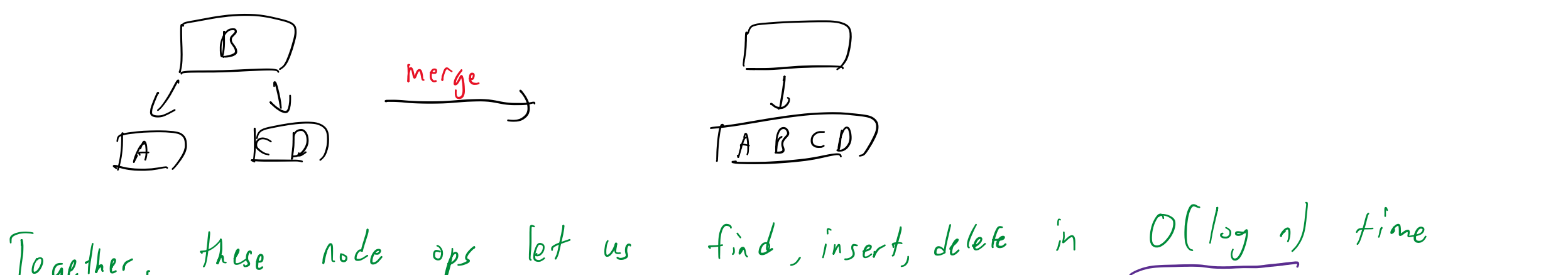


node operations:

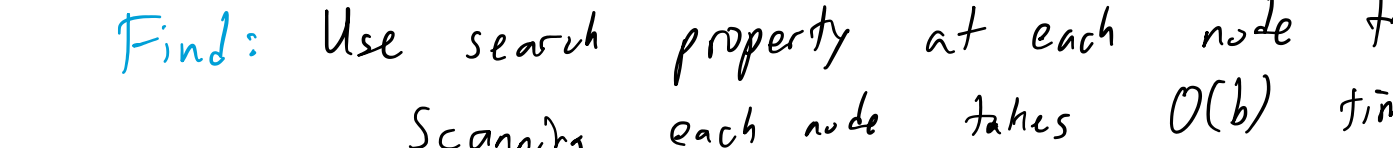
Split node when too many keys, promoting 1 key to level above



Rotation lets us move keys around to rebalance. Will be necessary later for insertion & deletion



Merging is the opposite of split



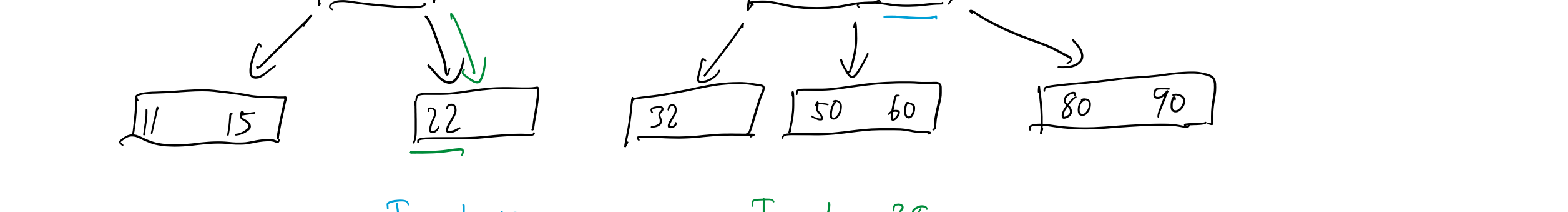
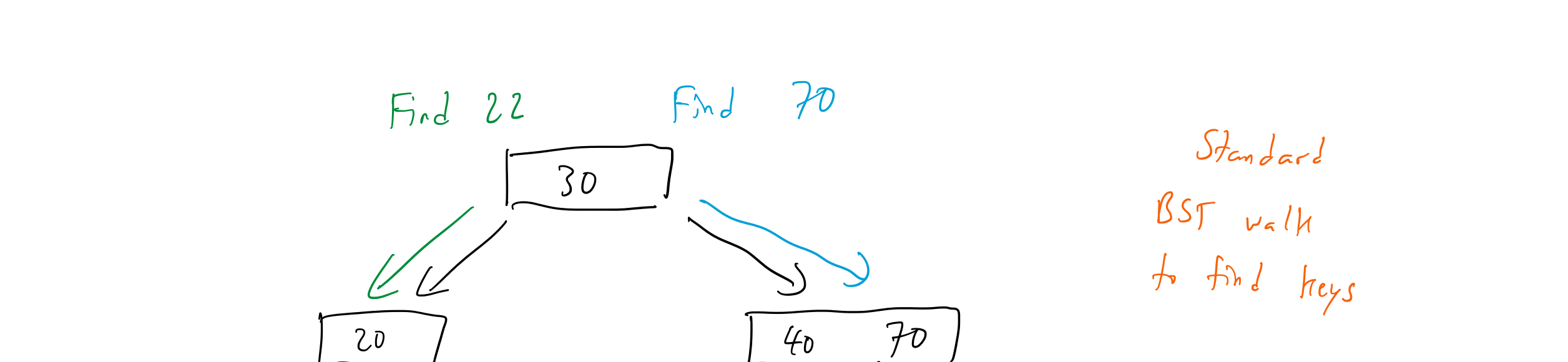
Together, these node ops let us find, insert, delete in $O(\log n)$ time

Find: Use search property at each node to find child node. Scanning each node takes $O(b)$ time, and # levels is $\log_a n$.

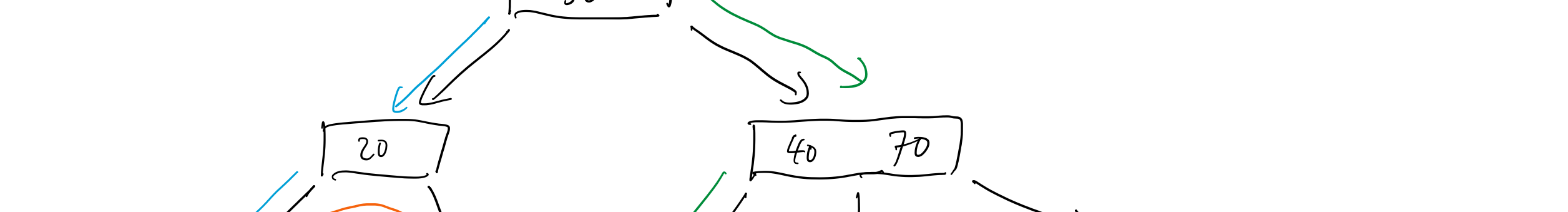
Insert: Insert as leaf. If node is full, either rotate to borrow from neighbors or split node and promote key up to parent, which might need to be split recursively. new level always gets created at root, so leaves remain constant height

Delete: Promote child key up, either in-order predecessor or successor. Rotate or merge if child node too small, possibly recursively. height of tree only decreases when root is empty & merging its children.

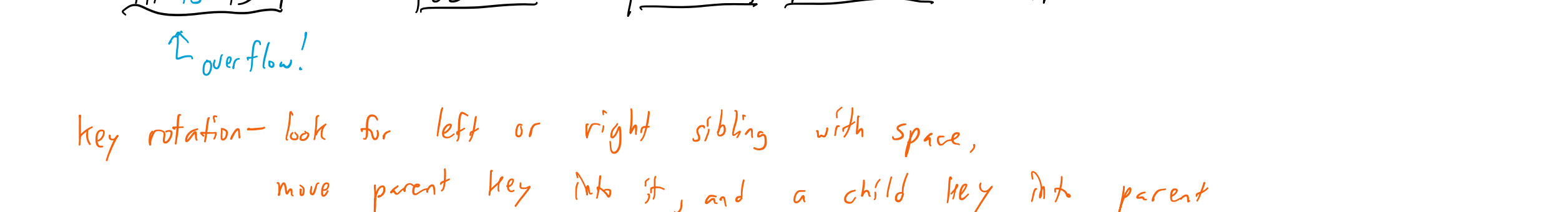
Example of (2,3)-tree



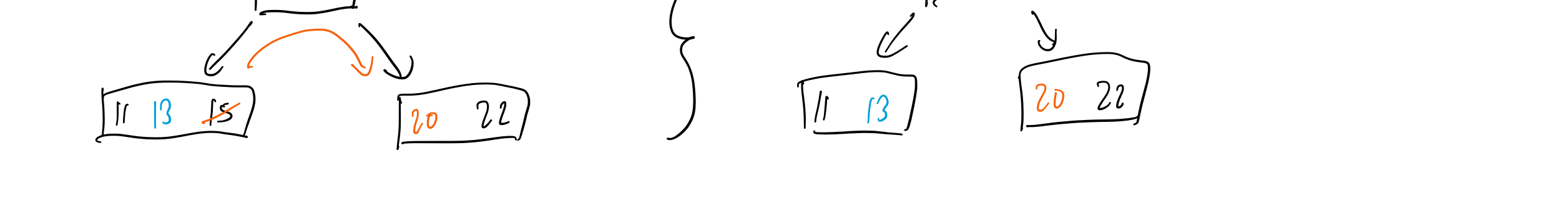
key rotation - look for left or right sibling with space, move parent key into it, and a child key into parent



Key rotation failure => split node when node has 3 values & can't rotate. Push middle value up to parent



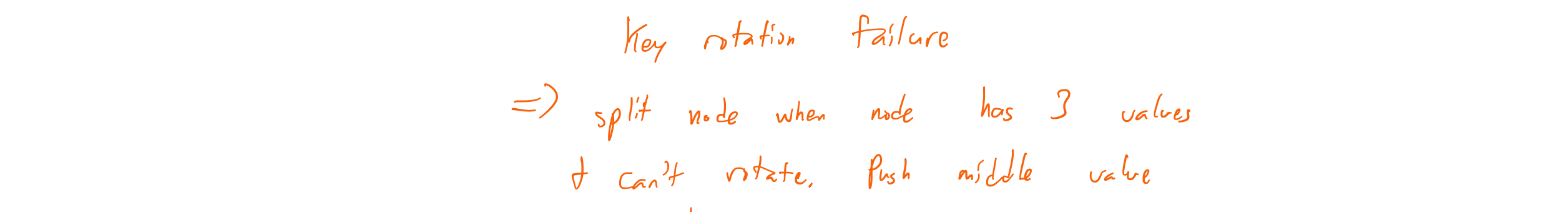
may need to recursively rotate or split



recursively split



New root node one level up

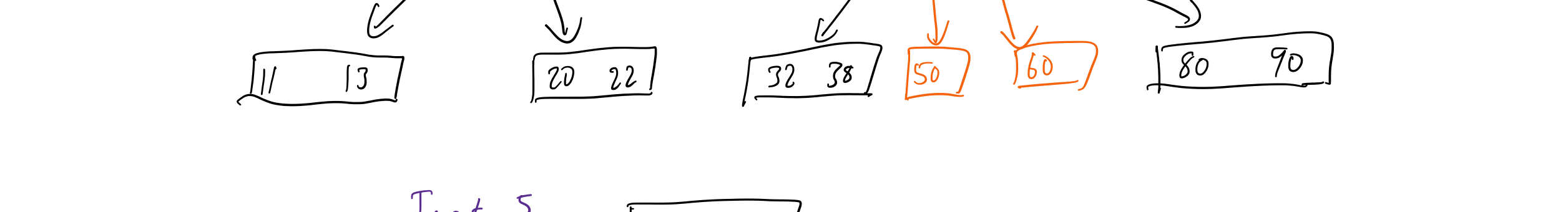


Splitting process always keeps same tree height for levels because we add on top

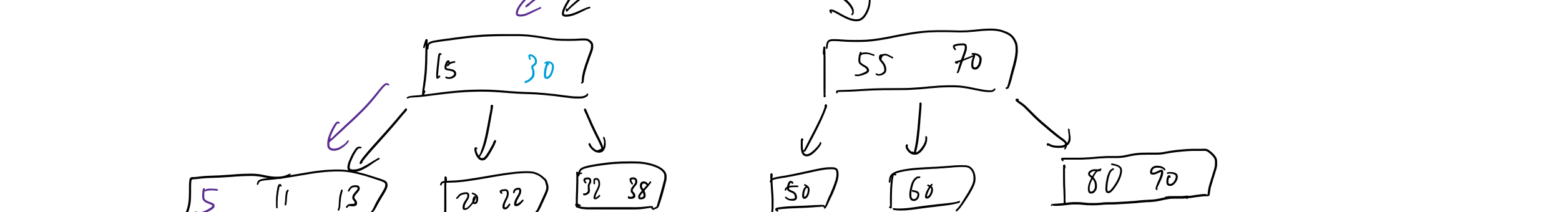
tree height between $\log_2 n$ and $\log_3 n$.

Delete 22, 32

Delete 20. Need to rotate or merge



Delete 1, then 2:



effectively a rotation

File system model

Page - contiguous block of data (e.g. 4096-byte chunk)

Probe - first access to a page (e.g. from disk to mem)

Time to probe is much larger than accessing data within a page

Cost model - minimize expensive probes

Goal

B⁺-trees

usually chosen to match device characteristics / pages

A B-tree of order b is an a_b -tree with $b = 2a - 1$

choose largest allowed a .

Want large b if bringing nodes into memory is slow, but scanning in memory is fast.

Ex. B-tree of order 1023 has $a = 512$

$n = 10,000,000 \Rightarrow$ height $O(\log_a n) \approx 2.58 \Rightarrow$ read 3 blocks from disk

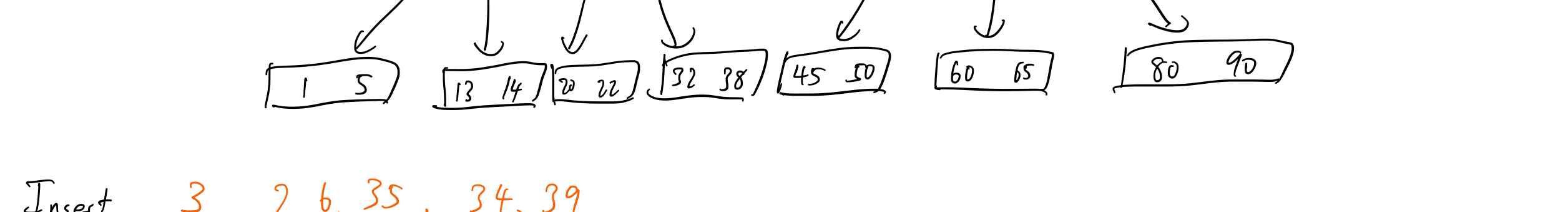
Properties

(a, b) -tree. Let $M = 2a + 1$. All nodes have between a & $M-1$ keys, except root, which has at least 2.

All client keys are in leafs / external nodes

Internal nodes contain copies of keys to guide search

Ex. $M = 6$ $a = 3$



Searching as usual in (a, b) -tree

Follow max link that is $\leq$ key

Insertion splits nodes of size M recursively upward

Never need to rotate keys, though you can

Deletion may require rotating keys & merging

Balance: # levels = $O(\log n)$

$\log_{M-1} n \leq \# \text{ levels} \leq \log_{\frac{M}{2}} n$ because every internal non-root node has $b/4$ $M-1$ + $\frac{M}{2}$ links

$\Rightarrow$ insert/search need $b/4$ $\log_{M-1} n$ + $\log_{\frac{M}{2}} n$ probes

In practice # probes is very small.

e.g. $M = 1024$, $n = 62$ billion $\Rightarrow \log_2 n \leq 4$

Aside: linearly searching through b keys is $O(1)$ if b constant, but might be slow.

However, can use other balanced tree (e.g. splay tree) at each node for more efficient searching