```
SELECT *
FROM movies
WHERE year > 2015
```

movies

movie_id	title	year
0126029	Shrek	2001
0181689	Minority Report	2002
0212720	A.I. Artificial Intelligence	2001
0983193	The Adventures of Tintin	2011
4975722	Moonlight	2016
5010201	Dunkirk	2017
5012394	Maigret Sets a Trap	2016

This sort of query can be answered efficiently with an index.

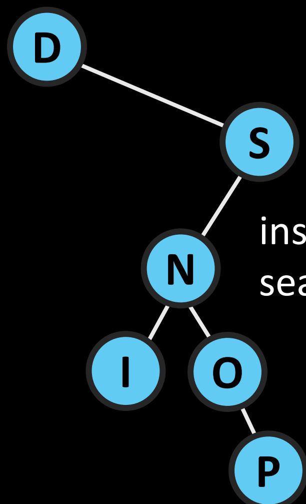
```
CREATE INDEX ind1 ON movies (year)
```

year	movie_id
2001	0126029
2001	0212720
2002	0181689
2011	0983193
2016	5012394
2016	4975722
2017	5010201

- An index contains a set of (key,value) pairs, ordered by key.
- It should support efficient search, as well as efficient insert / delete.

Crunch-time Charlie
(quick and dirty,
too harried to learn)

FREE-FORM
BINARY SEARCH TREE



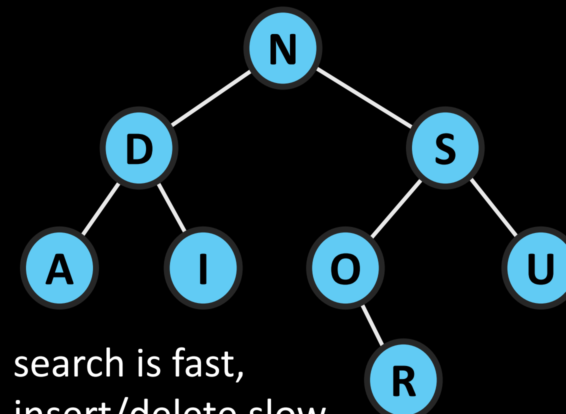
insert/delete fast,
search can be slow

Timely Terry
(no sweat,
plans ahead)

Q. Can we design a roughly-balanced search tree, but without being obsessive about it?

Fastidious Frances
(everything pristine
all of the time)

BALANCED
BINARY SEARCH TREE

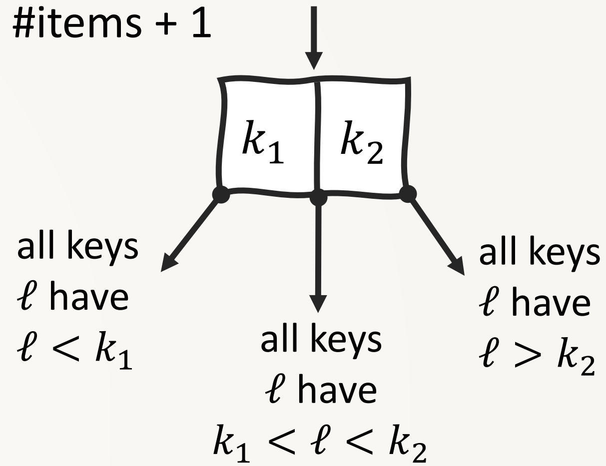


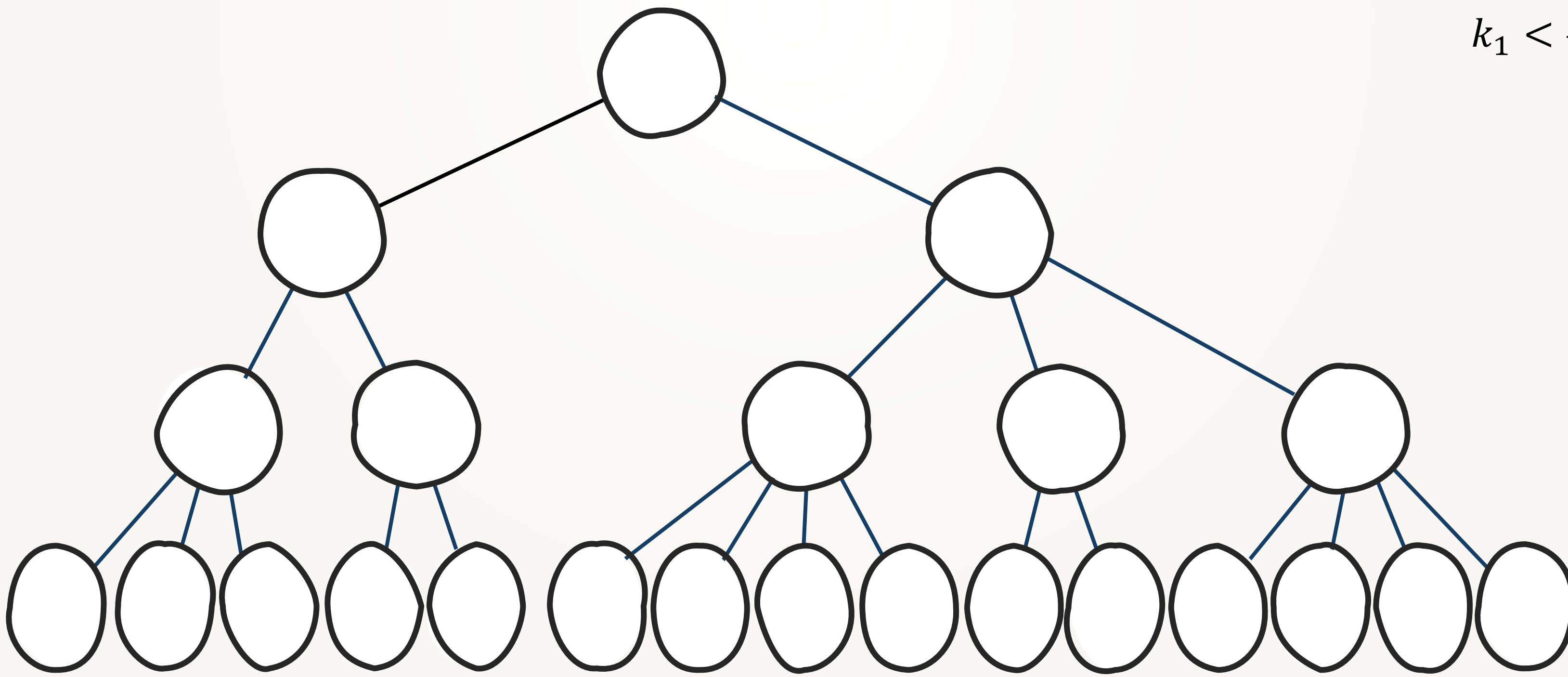
search is fast,
insert/delete slow

Genius idea: let's keep the depth perfectly balanced, but let the nodes have a variable number of children.

E.g. let's require that each non-leaf node have 2, 3, or 4 children.

To fit the standard BST design, let's store 1, 2, or 3 items at each node, #children = #items + 1

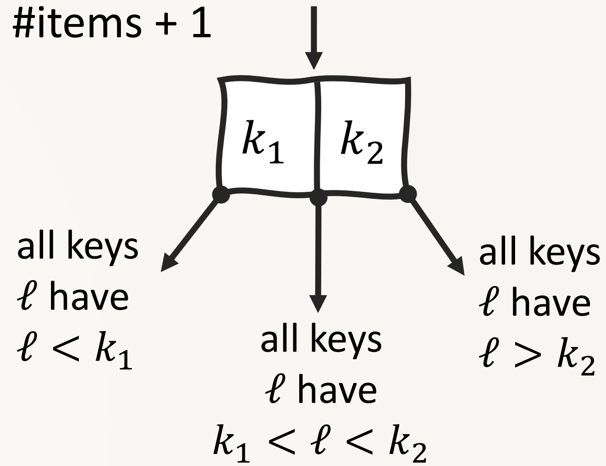




Genius idea: let's keep the depth perfectly balanced, but let the nodes have a variable number of children.

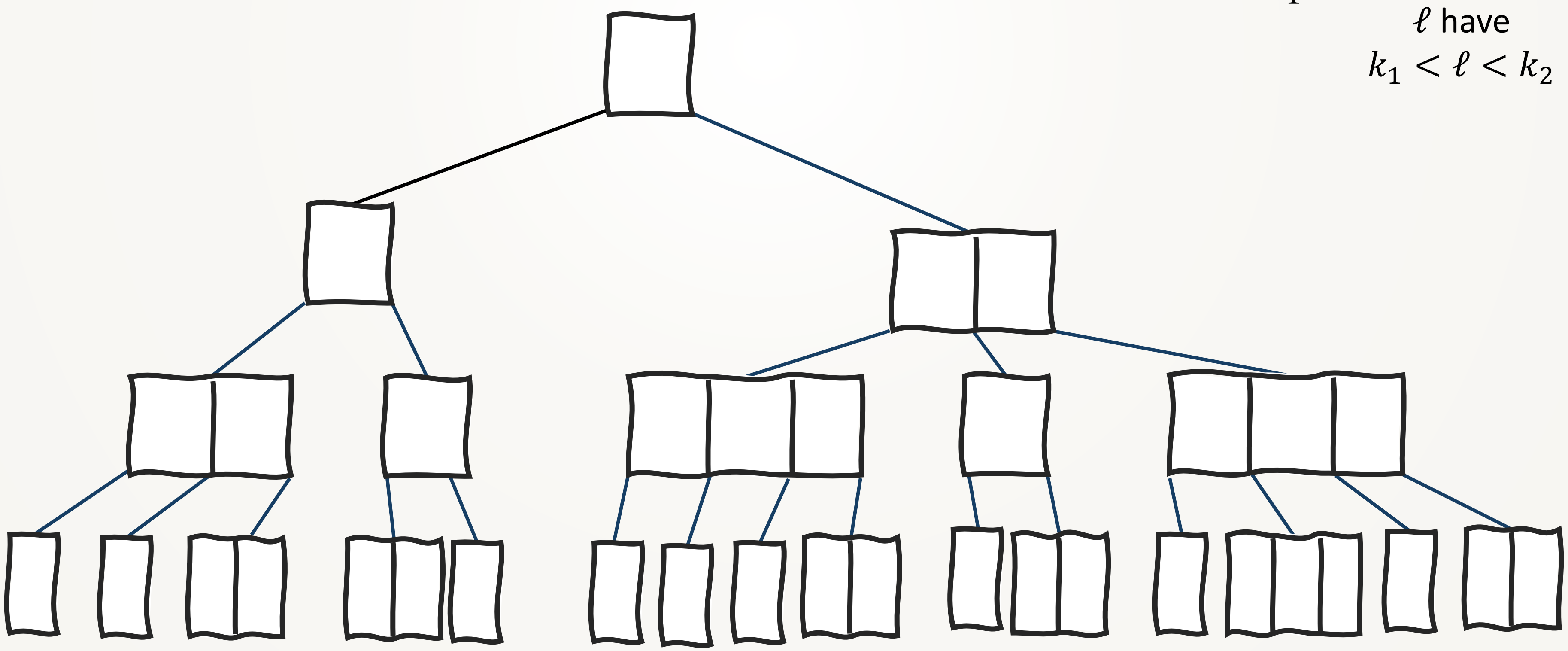
E.g. let's require that each non-leaf node have 2, 3, or 4 children.

To fit the standard BST design, let's store 1, 2, or 3 items at each node, #children = #items + 1

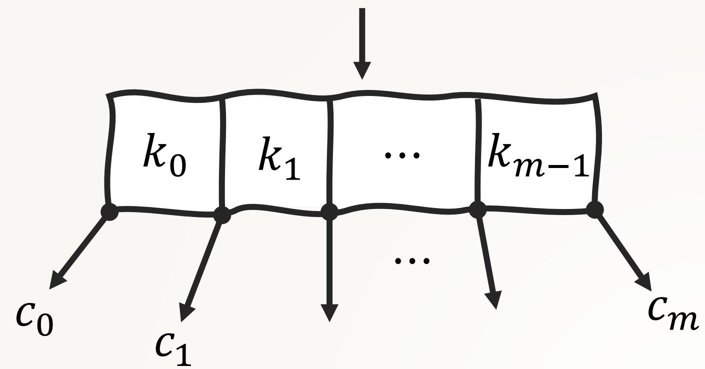


Q1. Is this balanced enough to give $O(\log n)$ search, where n is the number of (key,value) pairs?

Q2. Is this flexible enough that we can do insert/delete in $O(\log n)$, while maintaining the rough balance?



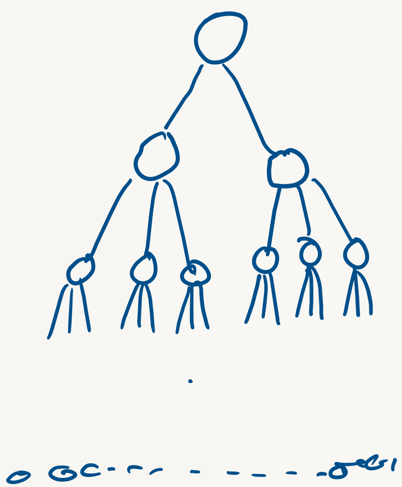
A B-tree is a perfectly height-balanced search tree, where each node has $\#\text{keys} \in \{k_{\min}, \dots, k_{\max}\}$ (apart perhaps from the root, which may have fewer) $(*)$ — so $\#\text{children} \in \{k_{\min}+1, \dots, k_{\max}+1\}$



For a node with m (key,value) pairs,

- There are $m+1$ child subtrees (unless it's a leaf)
- All keys ℓ in child c_i satisfy $k_{i-1} < \ell < k_i$
  (with appropriate adjustment at $i = 0$ and $i = m$)

What's the smallest possible # keys in a tree of height h?



root, depth 0 : allowed to have 1 key by $(*)$

depth 1 : 2 nodes, each with $k_{\min}$ keys

depth 2: $2 \times (k_{\min}+1)$ nodes, each with $k_{\min}$ keys

depth 3: $2 \times (k_{\min}+1)^2$ nodes, each with $k_{\min}$ keys

depth h : $2 \times (k_{\min}+1)^{h-1}$ nodes, each with $k_{\min}$ keys.

So, for an arbitrary tree, $\#\text{keys} \geq 1 + 2k_{\min}\left(1 + (k_{\min}+1) + \cdots + (k_{\min}+1)^{h-1}\right) = 2(k_{\min}+1)^h - 1$

QUESTION. For a tree with n keys in total, what's the largest possible height?

We've just argued that $n \geq 2(k_{min}+1)^h - 1$

Therefore $h \leq \log_{k_{min}+1}\left(\frac{n+1}{2}\right)$.

So h is $O(\log n)$ for <u>any</u> $k_{min} \geq 1$.

QUESTION. Why put an upper bound on #keys per node?

If we didn't bound # keys, we might have a node with $\Omega(n)$ keys. It'd take time $\Omega(n)$ to scan through them, slowing down search & insert/delete.

For <u>any</u> finite bound $k_{max} < \infty$, we ensure that the work per node is $O(1)$.

Putting these together, search is $O(\log n)$.

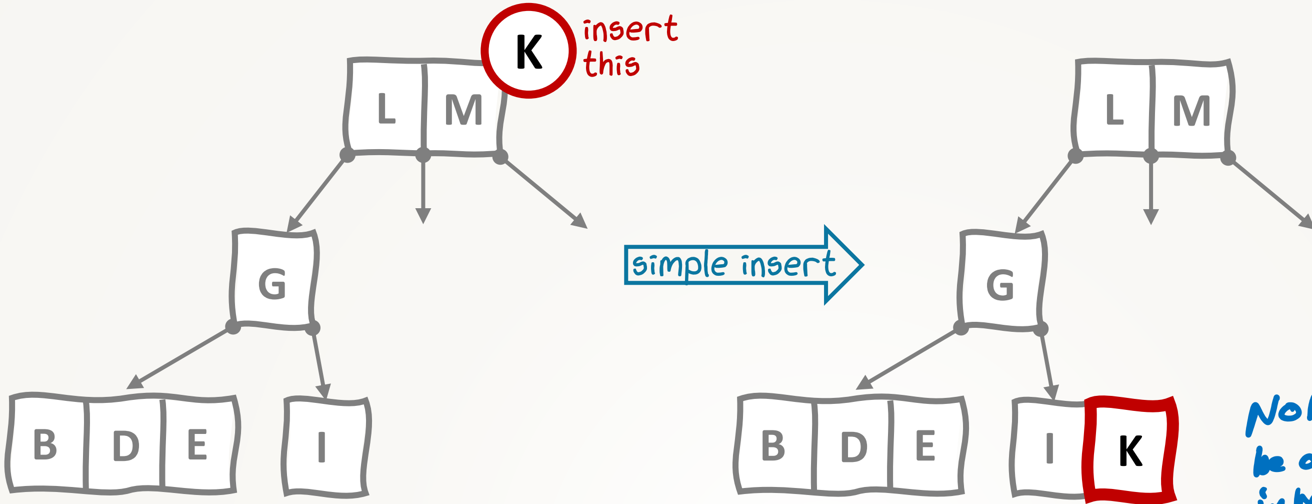
SECTIONS 4.4 & 4.6

2-3-4 trees and B-trees

insert(k, v) into a tree with $k_{\min} = 1$ and $k_{\max} = 3$

EASY CASE

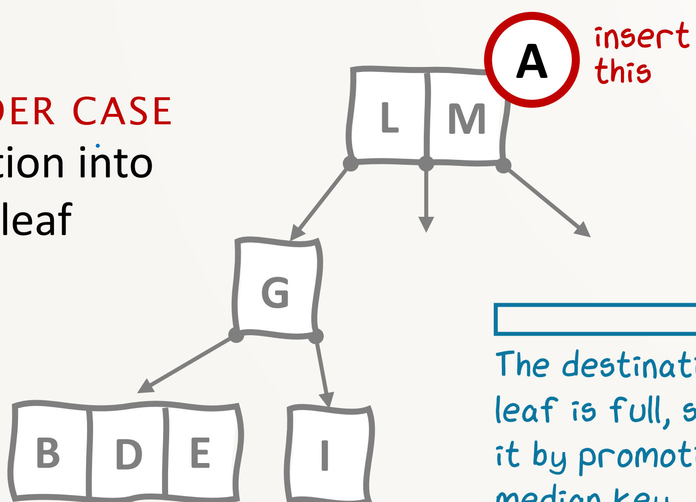
simple insertion into a leaf



Note: insertions have to be at the bottom, not in internal nodes.

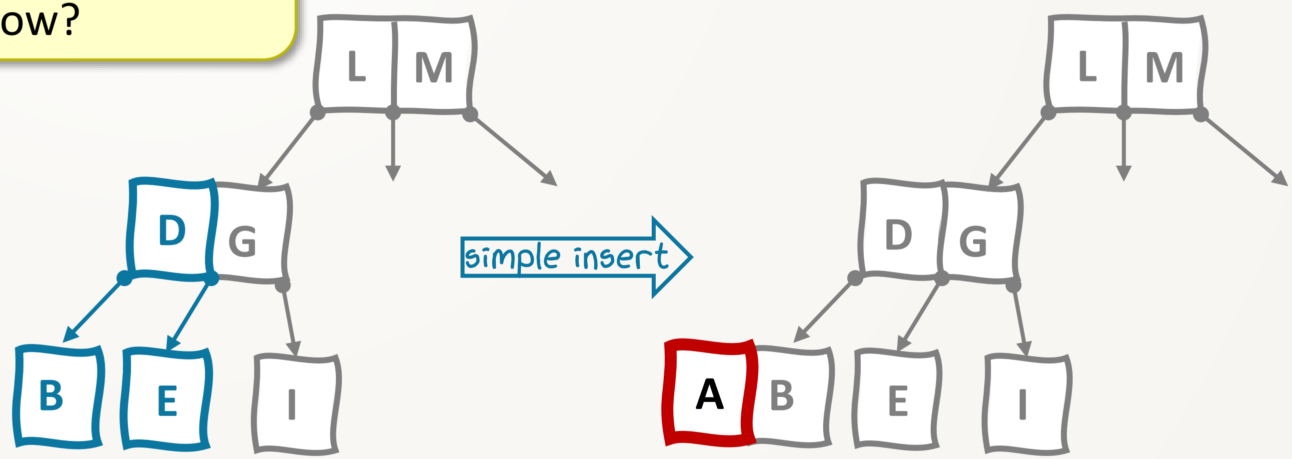
HARDER CASE

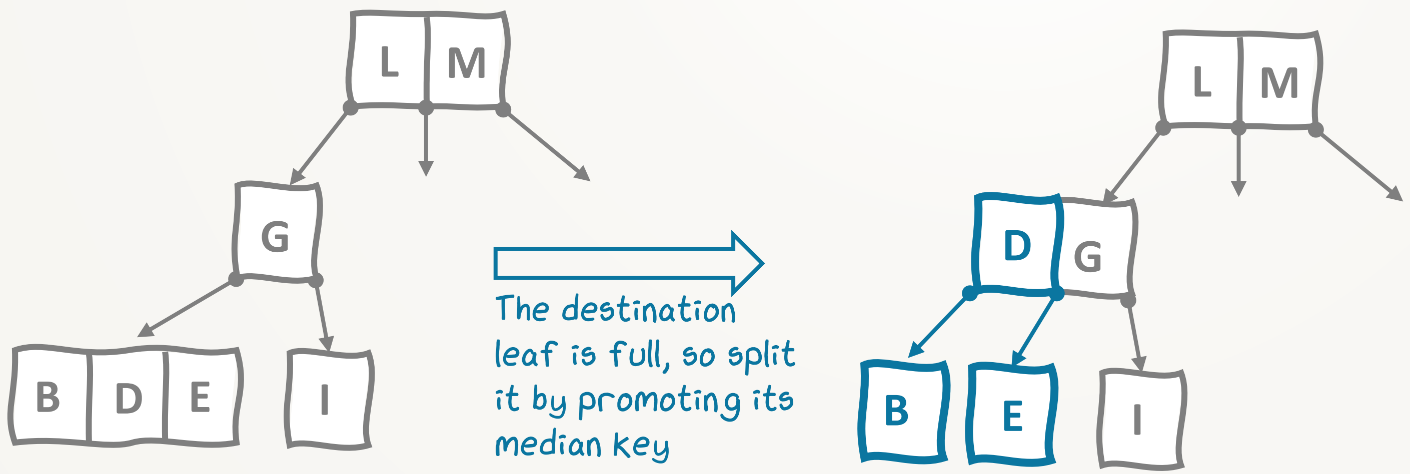
insertion into a full leaf

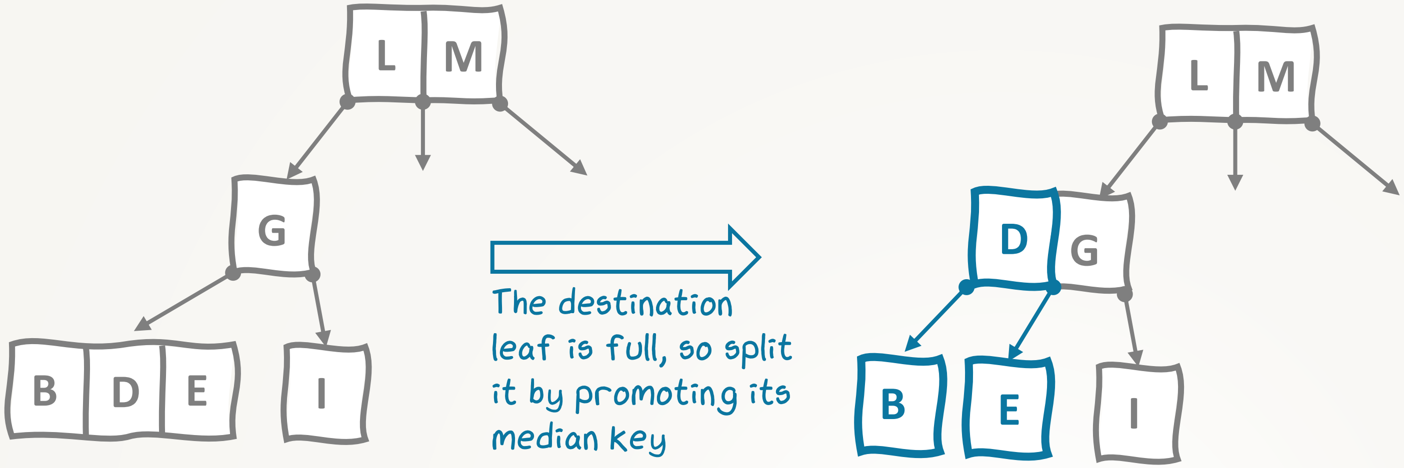


QUESTION. Where do we insert key **A**? And how?

The destination leaf is full, so split it by promoting its median key



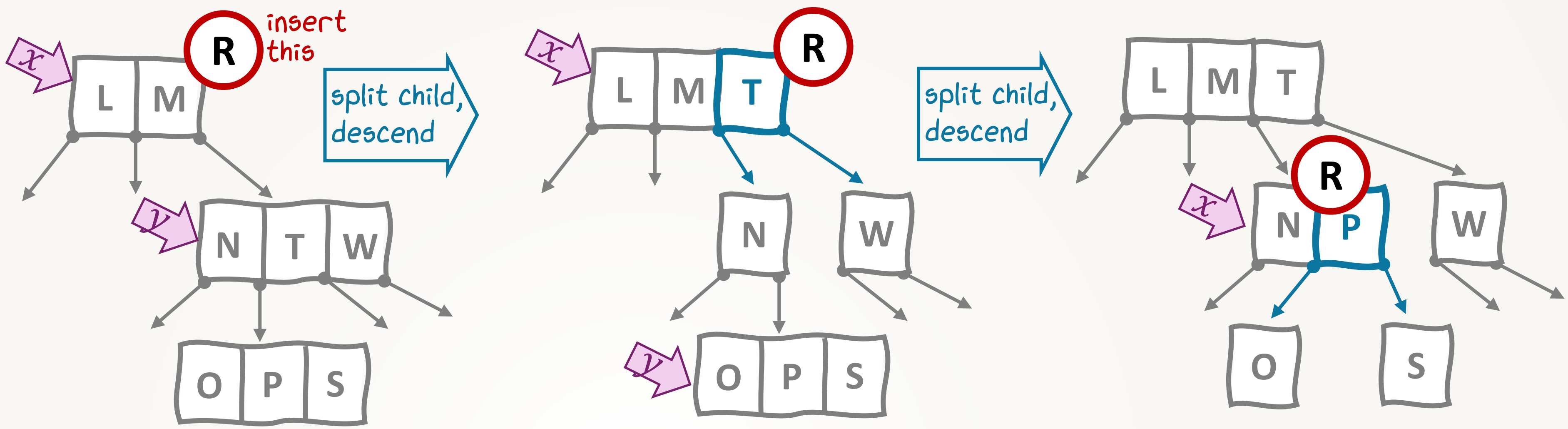

L
M
G
B
D
E
I
The destination leaf is full, so split it by promoting its median key
L
M
D
G
B
E
I

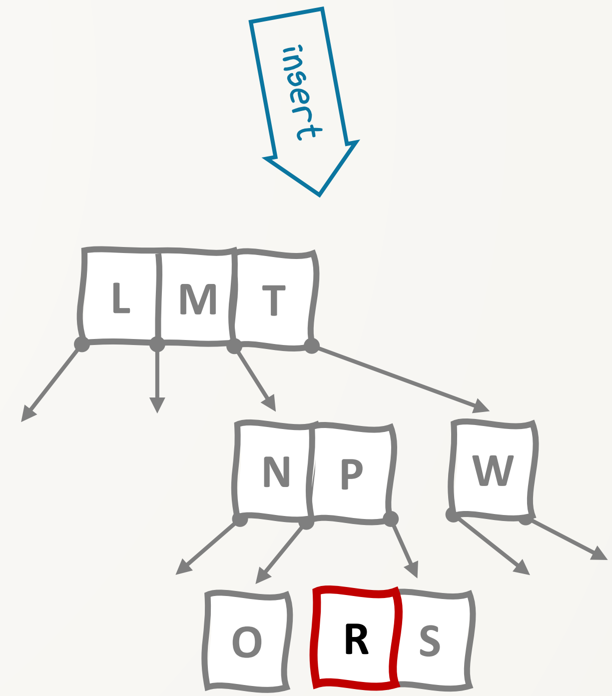


To keep our tree balanced, excess keys need to be pushed *up*.

insert(k, v) into a tree with $k_{\min} = 1$ and $k_{\max} = 3$

HARDEST CASE
insertion into a full leaf on a full path



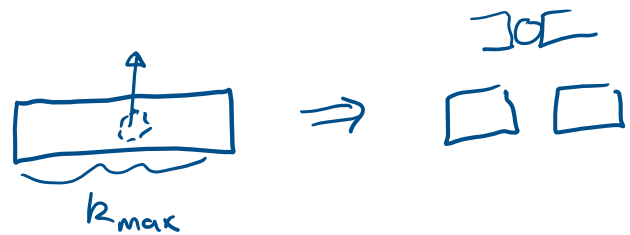



```
def insert(k,v):
  if root node is full:
    split it, and create new root
  x ← root node
  while x is not a leaf:
    # assert x is not full
    scan x to find which child y we want k in
    if y is not full:
      x ← y
    else:
      split y into y1 and y2 and promote a key
      x ← y1 or y2 as appropriate
  insert (k,v) into x
```

QUESTION. How do we insert key **R**?

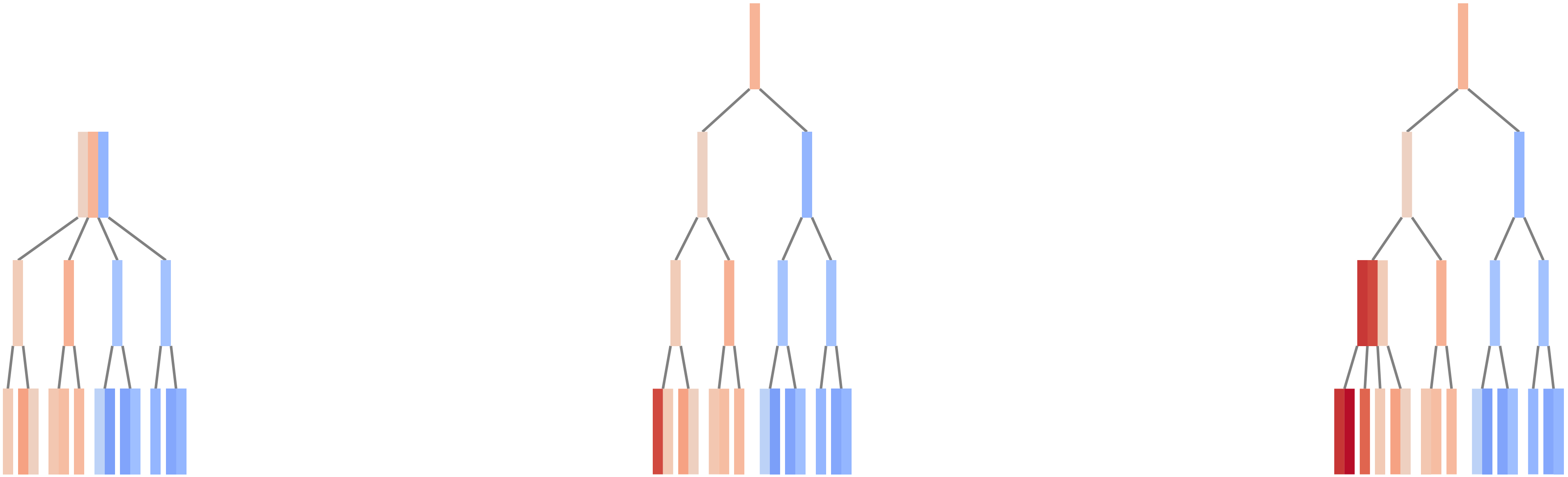
NOTE. This code is suitable for the general case. It may unnecessarily split some nodes at the top, but who cares?

QUESTION. Does this splitting operation constrain $k_{\min}$ and $k_{\max}$?



For this split to result in legal nodes,

we need $\quad k_{max} - 1 \geq 2\, k_{min}$.

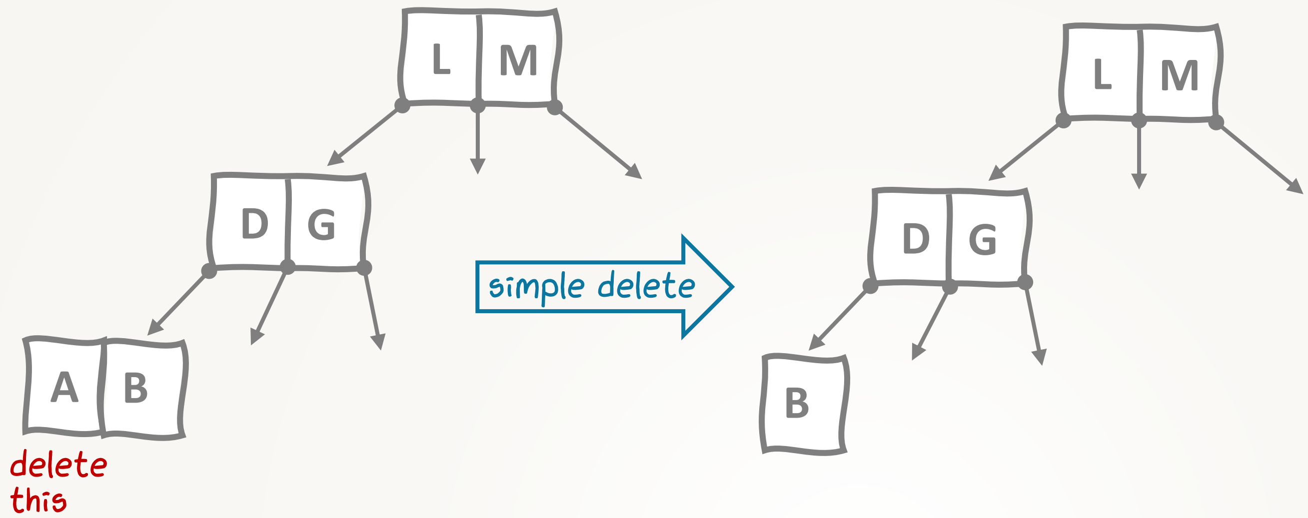


To keep our tree balanced, excess keys need to be pushed *up*.

From time to time, we may have to add a new node *at the top.* The tree becomes higher, but it remains perfectly height-balanced, delete(k) from a tree with $k_{\min} = 1$ and $k_{\max} = 3$

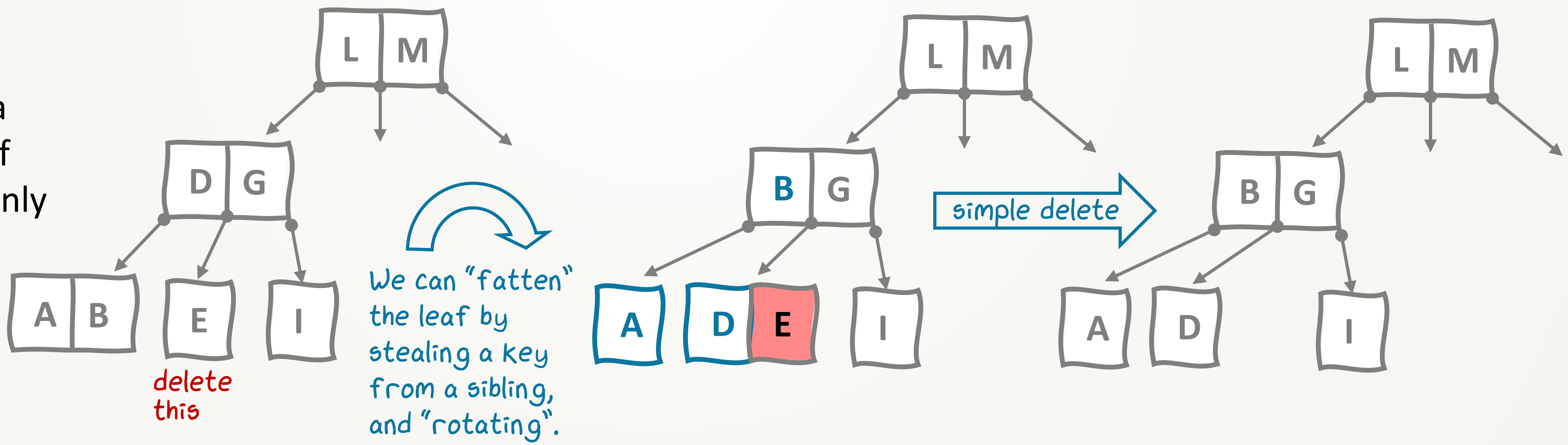
EASY CASE

simple deletion from a leaf



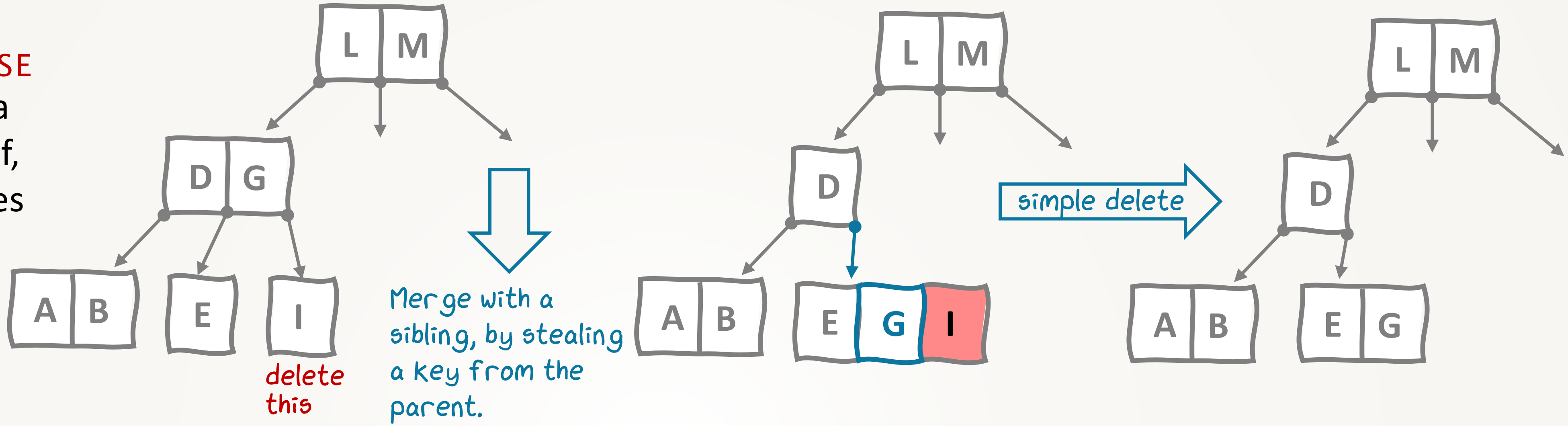
HARDER CASE

deletion from a bare-bones leaf (i.e. one with only $k_{\min}$ keys)

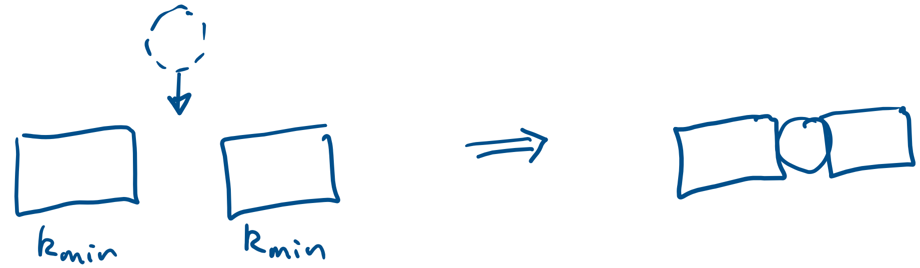


$\texttt{delete}(k)$ from a tree with $k_{\min} = 1$ and $k_{\max} = 3$

HARDERER CASE
deletion from a bare-bones leaf, with bare-bones siblings



QUESTION. Does this merging operation constrain $k_{\min}$ and $k_{\max}$?

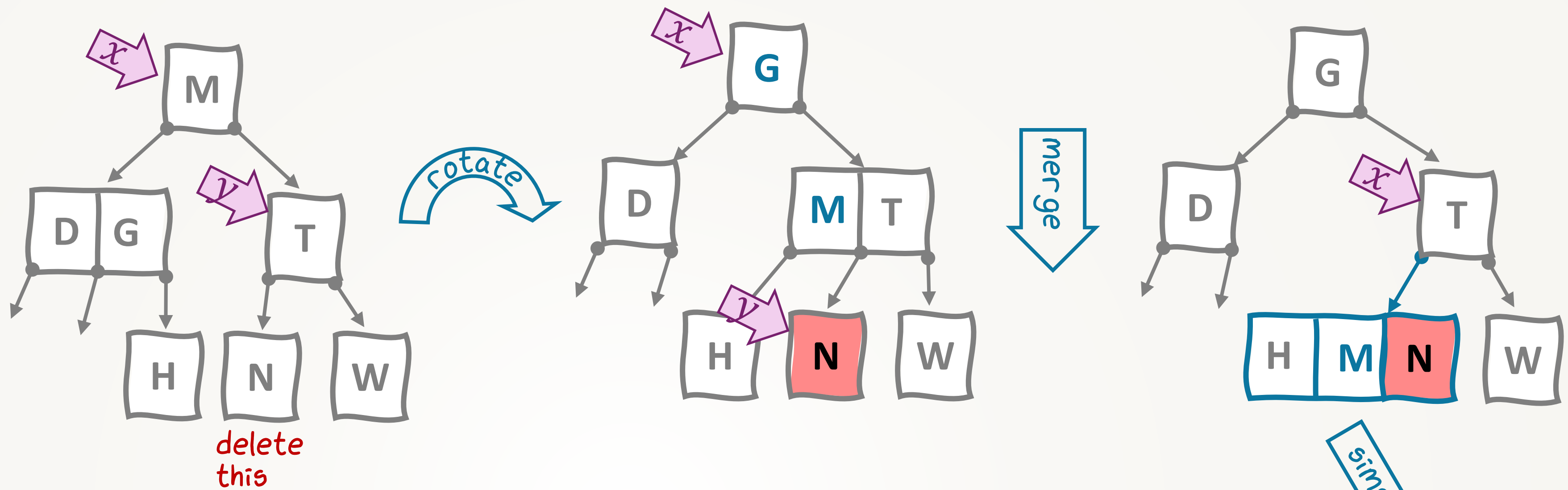


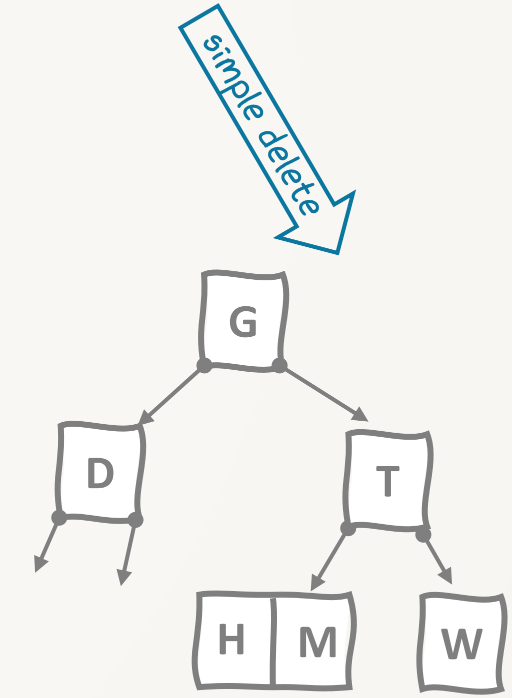
For this merging to result in a legal node, we need $2k_{\min}+1 \le k_{\max}$.

(This is exactly the same inequality we saw when we checked splitting.)

delete(k) from a tree with $k_{\min} = 1$ and $k_{\max} = 3$

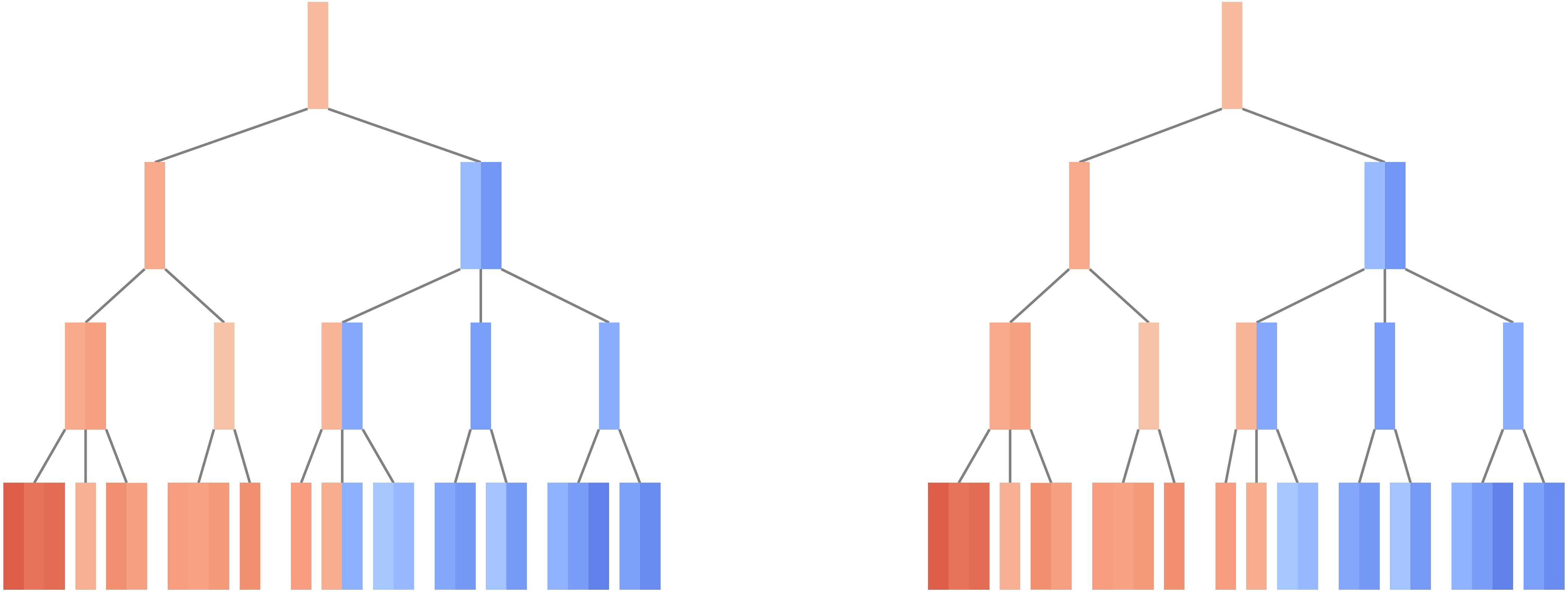
HARDEST CASE
deletion from a bare-bones leaf on a bare-bones path

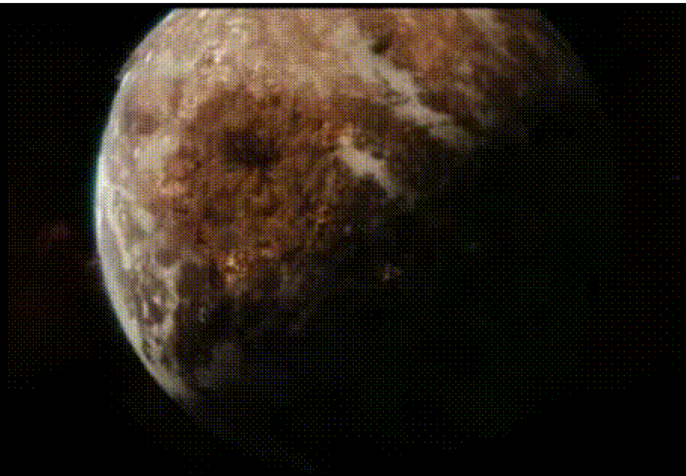





```
def delete(k):
  x ← root node
  while x is not a leaf:
    # assert x has > k_min keys (or is root)
    if x has key k:
      …
    else:
      scan x to find the child y that has k
      if y has k_min keys:
        either rotate or merge to make y fatter
      x ← y
  delete k from x
```

See trick from last lecture on how to delete keys from internal nodes.





To keep our tree balanced, deletions *suck in* keys from beside or above.

From time to time, we may suck down the root when merging its children. The tree becomes shorter, and it remains perfectly height-balanced,

How should we choose $k_{\min}$ and $k_{\max}$?

- height $= O(\log n)$ as long as $k_{\min} \geq 1$
- The work at each node is $O(1)$ as long as $k_{\max} < \infty$
- We need $k_{\max} \geq 2k_{\min} + 1$ for merging/splitting to work

Are there any other considerations that can guide us to specific choices?

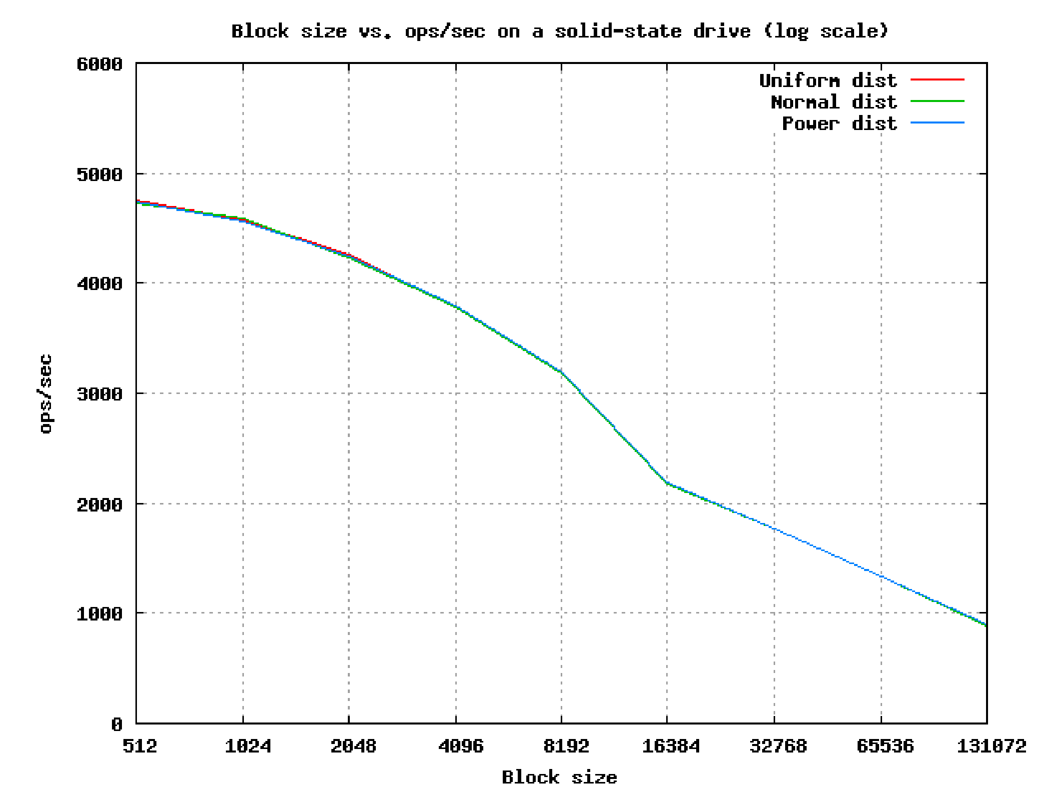
Rethinking B-tree block sizes on SSDs

RethinkDB Team OCTOBER 05, 2009 benchmarks

One of the first questions to answer when running databases on SSDs is what B-tree block size to use. There are a number of factors that affect this decision:

- The type of workload
- I/O time to read and write the block size
- The size of the cache

That's a lot of variables to consider. For this blog post we assume a fairly common OLTP scenario - a database that's dominated by random point queries. We will also sidestep some of the more subtle caching effects by treating the caching algorithm as perfectly optimal, and assuming the cost of lookup in RAM is insignificant.

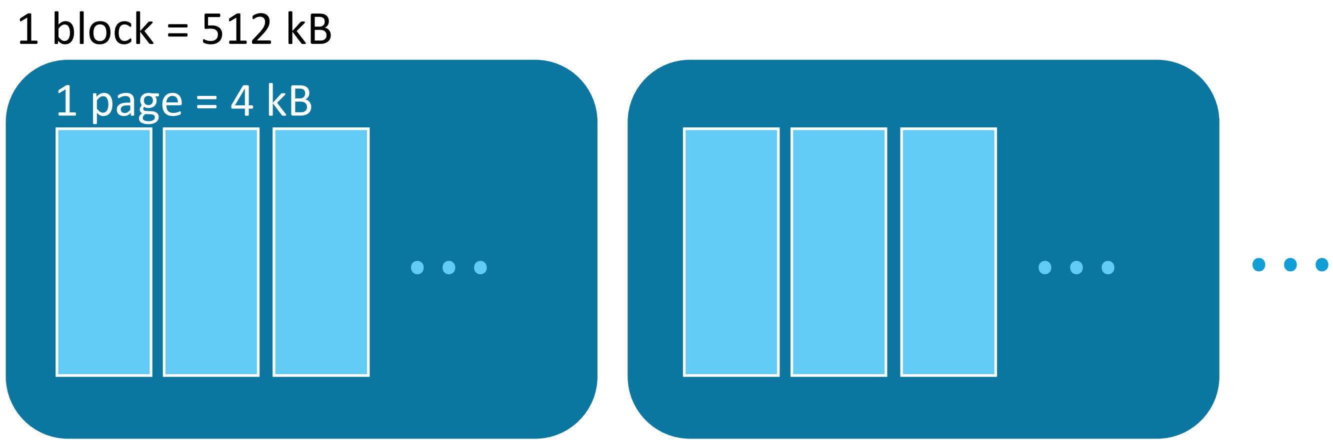


1kb (32 keys) 4579 IOPS	2kb (64 keys) 4254 IOPS	4kb (128 keys) 3780 IOPS	8kb (256 keys) 3197 IOPS	16kb (512 keys) 2186 IOPS	32kb (1024 keys) 1769 IOPS	64kb (2048 keys) 1334 IOPS
5.98 hops	4.98 hops	4.27 hops	3.74 hops	3.32 hops	2.98 hops	2.72 hops
765 q./sec	854 q./sec	**885 q./sec**	854 q./sec	658 q./sec	593 q./sec	490 q./sec

So, if we have no cache the optimal block size is 4KB.

Experimenting with different node sizes, they found that max-size = 4kB gave best performance (for a database stored on SSD).

If we're storing our index on an SSD:

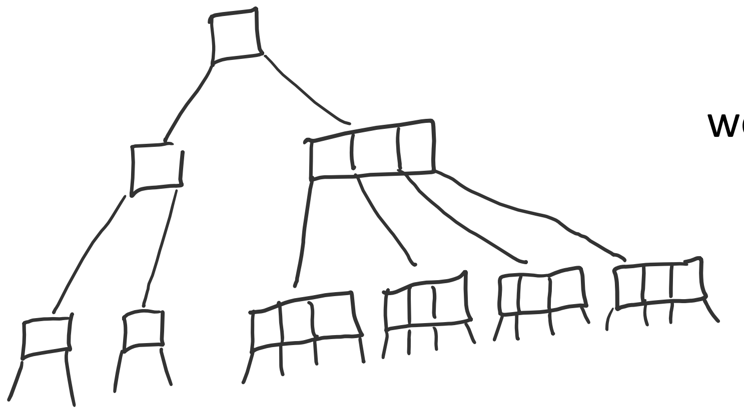


- An SSD consists of many *blocks*, each made of many *pages*
- We read and write an entire page at a time
- Reading and writing to an SSD is very slow, compared to main memory access

⇒ Choose $k_{\max}$ so that a node takes up an entire page,
and choose $k_{\min}$ as large as possible, i.e. $k_{\min} = (k_{\max} - 1)/2$, to keep pages full
(This explains the experimental finding from RethinkDB, that 4kB nodes are best.)

This is called a B-tree.

If we're storing our index in main memory ...



we'll make a different choice of $k_{\min}$ and $k_{\max}$.