Adv. Data Structures

B-Trees

Recap

- HW1 due
- HW2 - out later this week, over (more binary trees

Motivation:

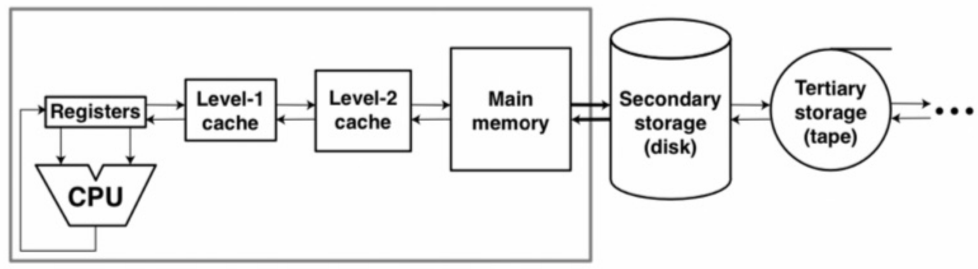
So far, we've mostly worried about running time.

With good reason! But not the only thing.

Random Access Machine (RAM) model of computation:

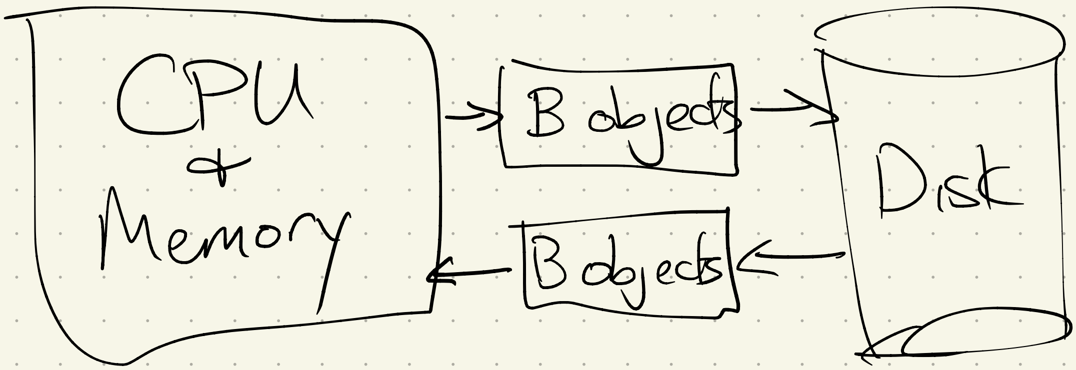
A traditional random access machine

Clearly, not realistic!



A more accurate (but still oversimplified) model of a modern computer system.

External-Memory Model of Computation: [Aggarwal-Vitter '88]



- Single processor + memory, holding M objects
- I/O operations that move B objects at a time

<u>Note</u>: $1 \le B \le M$

typical values:

4-byte objects

$B = 2^{11}$

$M = 2^{27}$

Then:

- Cost of an algorithm is measured in # I/Os. (Computation is free)
- Size of a data structure is # of ~~blocks~~ objects it uses, N

Memory holds:

$m = M/B$ blocks

DS needs:

$n = N/B$ blocks

We'll assume $\underline{N > M}$.

Why? If all fits in memory, then easy.

Before we get to complex data structures:

Searching:

Given a ~~list~~ an array of N objects, is X in the list?

Algorithms:

Linear search:

$$O\left(N/B\right) = O(n)$$

Can we binary search, if array is sorted?

Load middle block:

1 I/O

$\frac{N}{2} - B$

$O(\log n)$

Goal: get $O(\ldots n \ldots)$

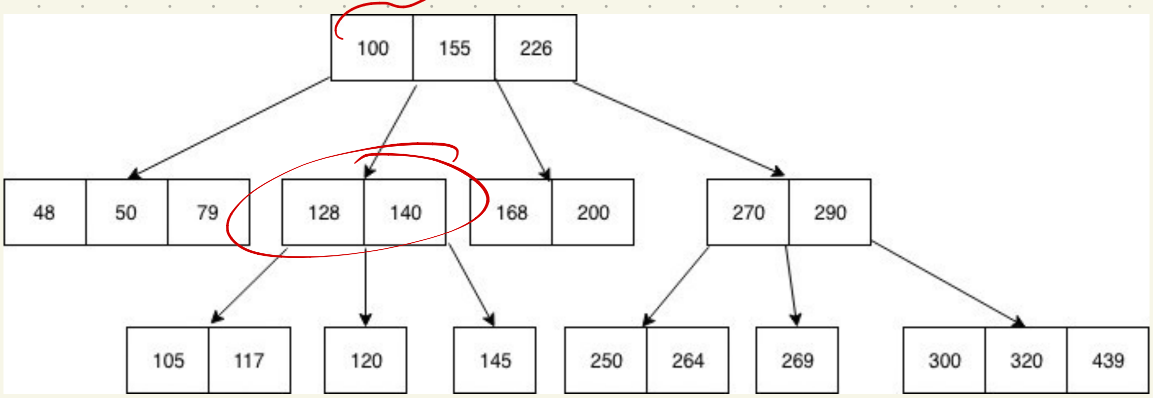
not $O(\ldots N \ldots)$

B-trees: [Bayer–McCreight '70]

Generalization of BSTs:

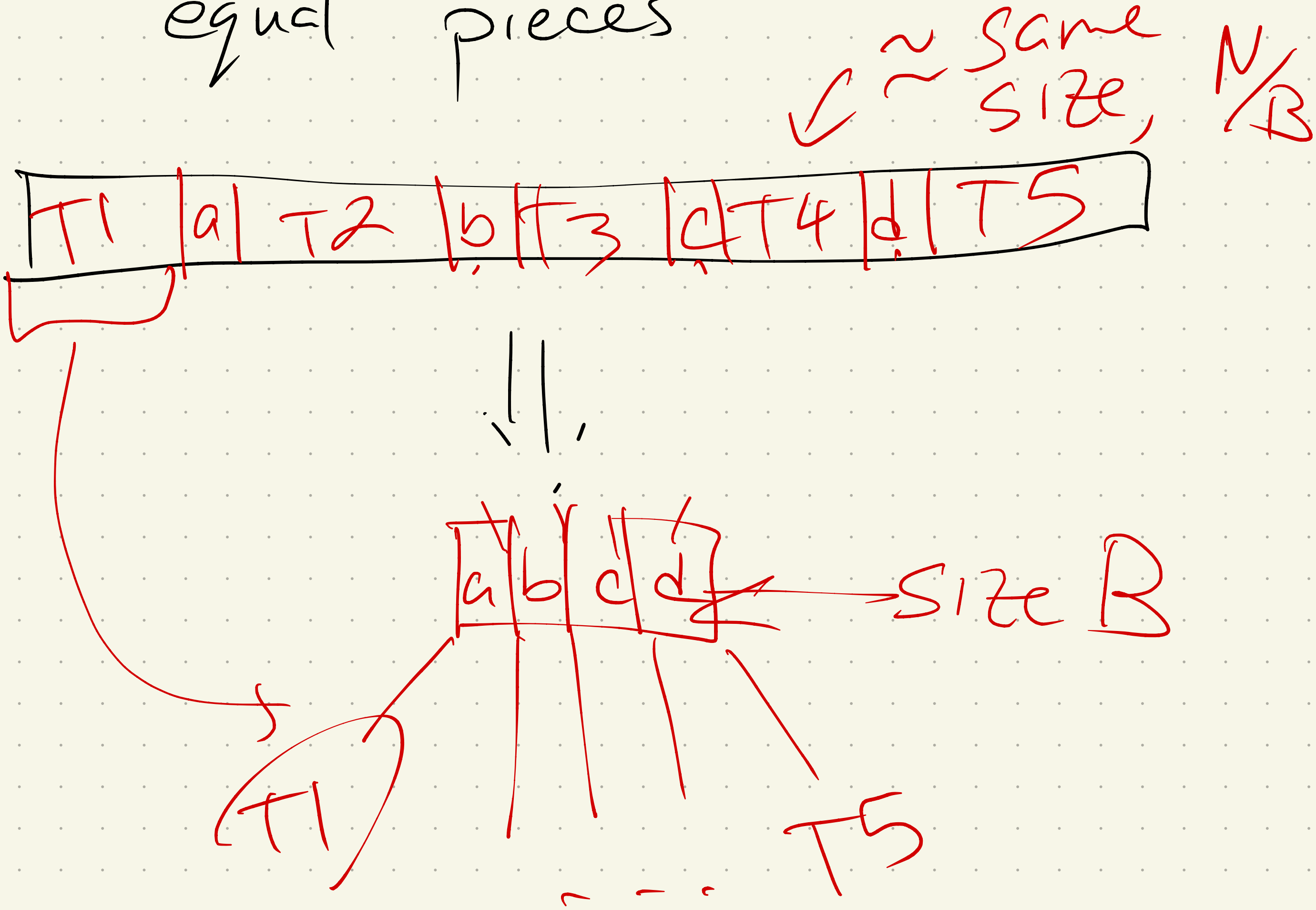
- Up to B values per node
  root / leaves can have $< B/2$
- Up to B+1 children per node:
  root ≥ 2, internal $\geq B/2$

Ex: A 3-tree:



(Sometimes data is only in leaves, & rest are artificial "pivot" values)

Goal: balance, so B values split the array into roughly equal pieces



Search time: $n = N/B$ blocks in tree

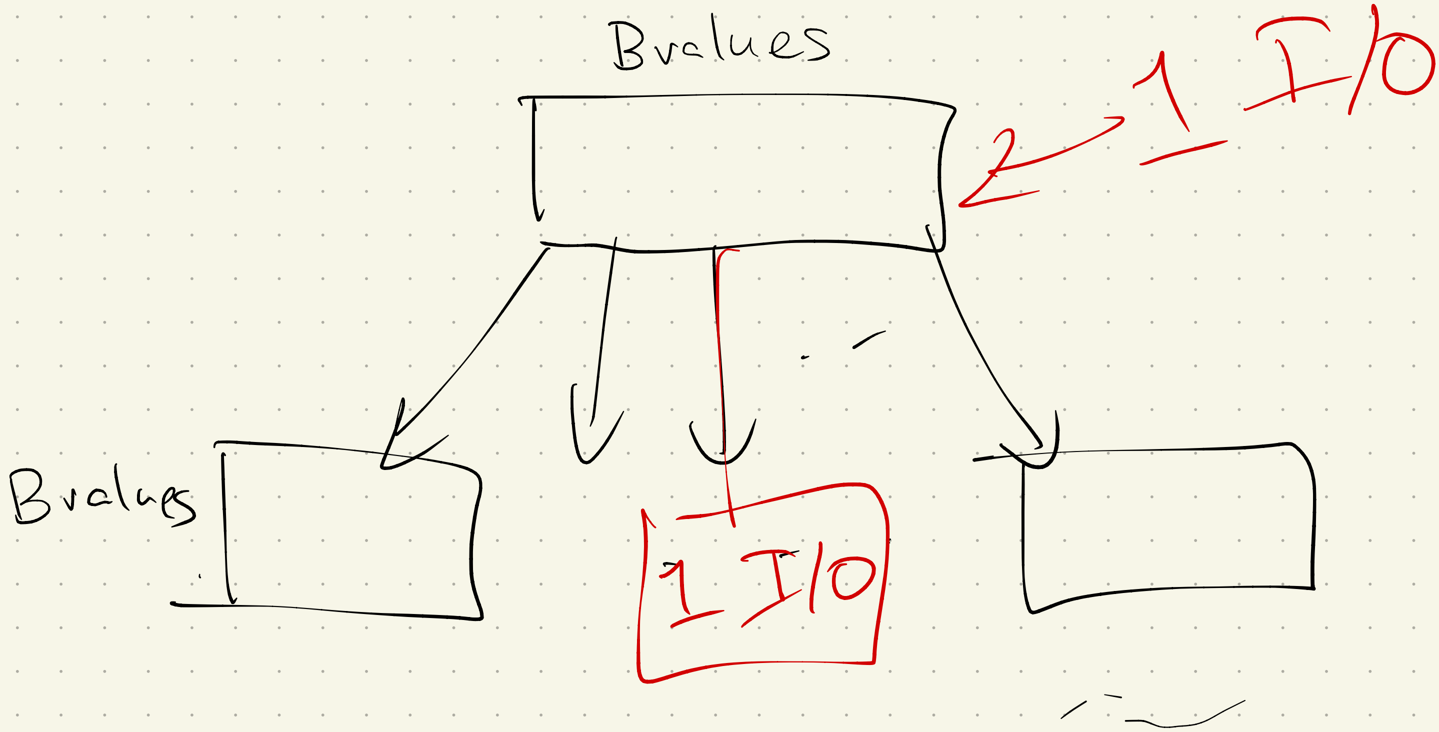
so each level has $\leq \frac{N}{(B/2)}$

$\Rightarrow \frac{N}{B^{depth}} = 1 \quad N = B^{depth}$

$\log_B N = depth$

$\Rightarrow O(\log_B n)$

Obvious advantage in external memory!



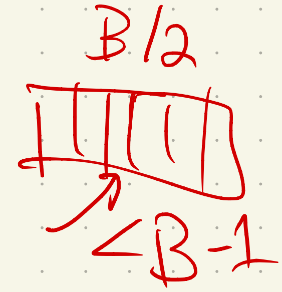
$O(\log_B n)$ levels

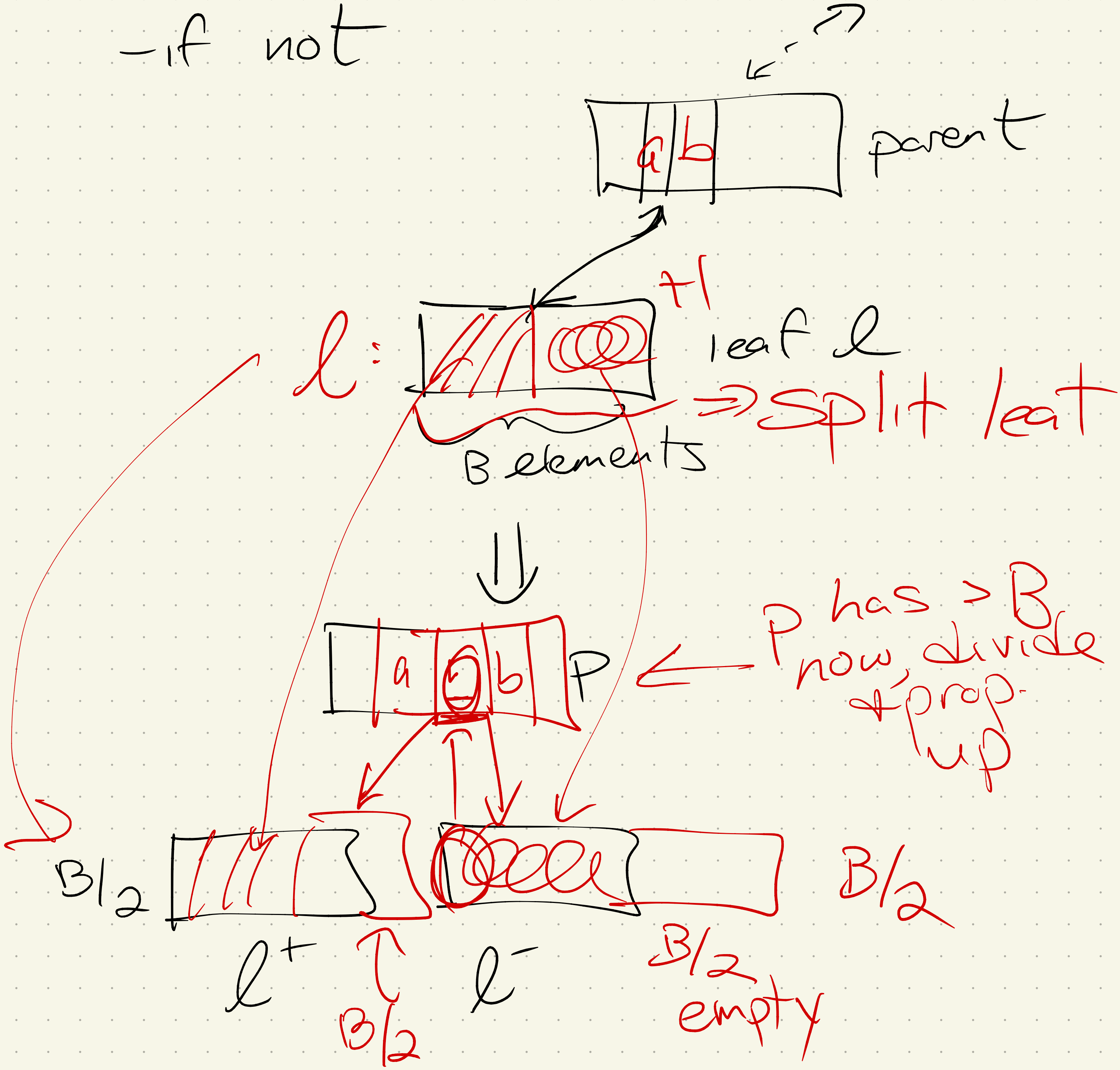
$\Rightarrow$ # I/Os

So Find!

Inserting:

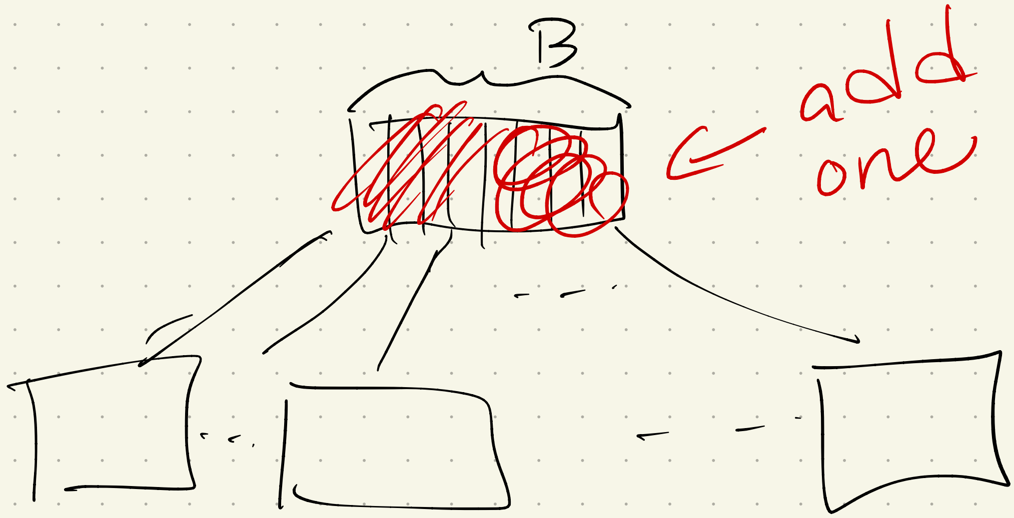
- locate leaf it should go in (using search)
- if leaf has space, done
- if not



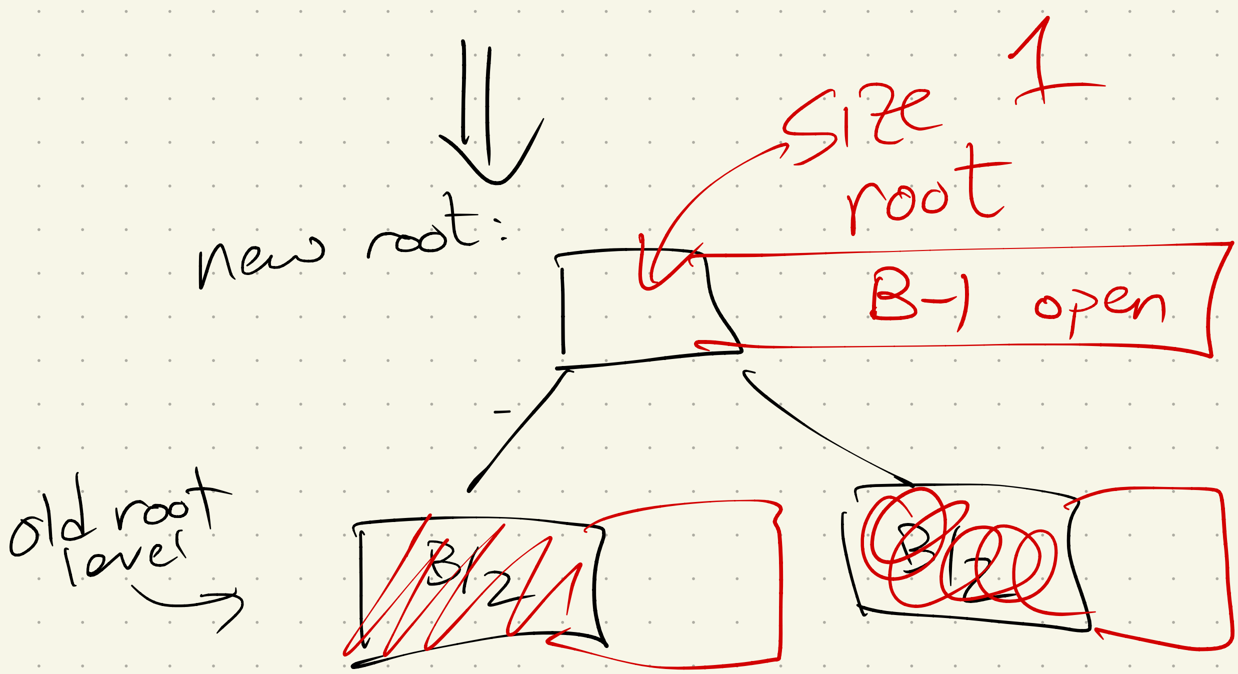


These splits can propogate up to root

$\Rightarrow$ We create a new root, of size



$\Downarrow$



Insert runtime:

- $O(\log_B n)$ to find
- Then split $O(\log_B n)$ blocks

"Time" to split:

block accesses

I/Os

$\leq 4 \log_B n$

identity

$\boxed{\log_c d = \frac{\log d}{\log c}}$

$= O(\log_B n)$

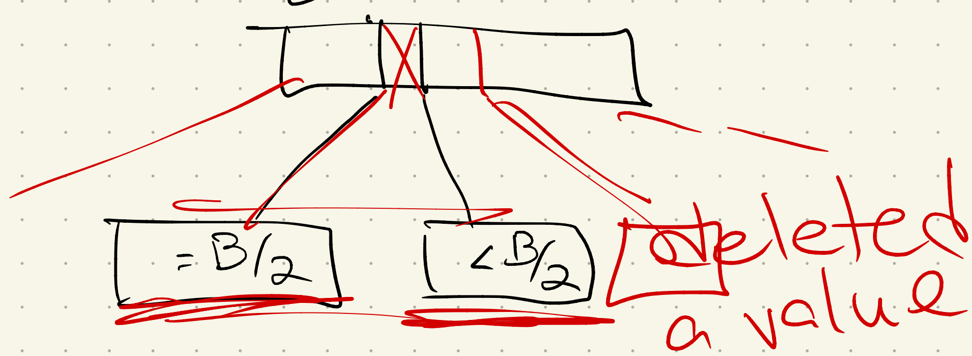
$= \frac{\log_2 n}{\log_2 B}$

Delete: Opposite of insert:

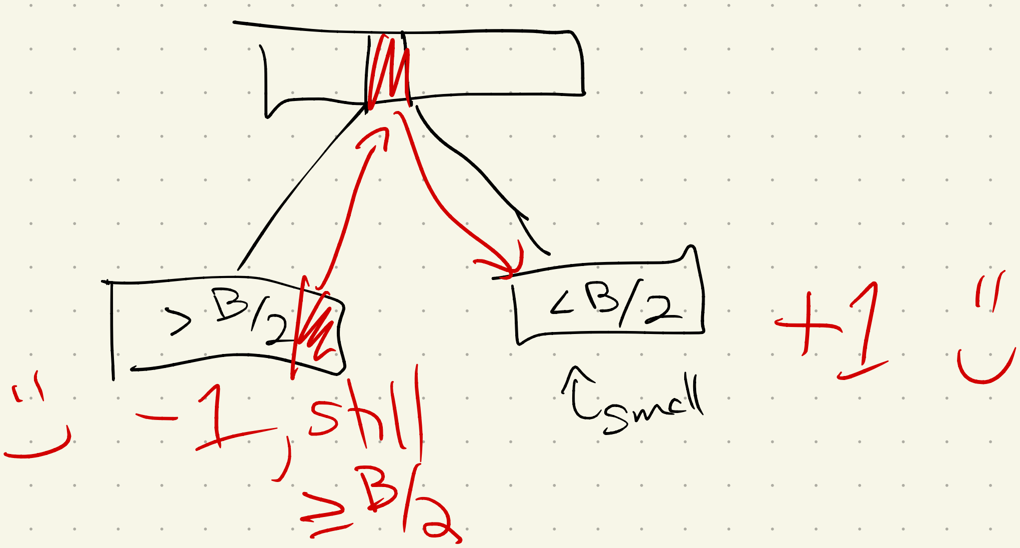
Find x & delete it.

If size is $< B/2$:

- there is either an immediate sibling of size $= B/2$



- or an immediate sibling of size $> B/2$



Agan, delete can propogate up, since we may need to remove a key from the internal node (if 2 merged)

Path to root has size:

$$\Rightarrow O(\log_B n)$$

One more note:

Suppose we're back in RAM-model, & have to pay for searches inside a block.

Find:

Know: $O(\log_B n)$ blocks to load

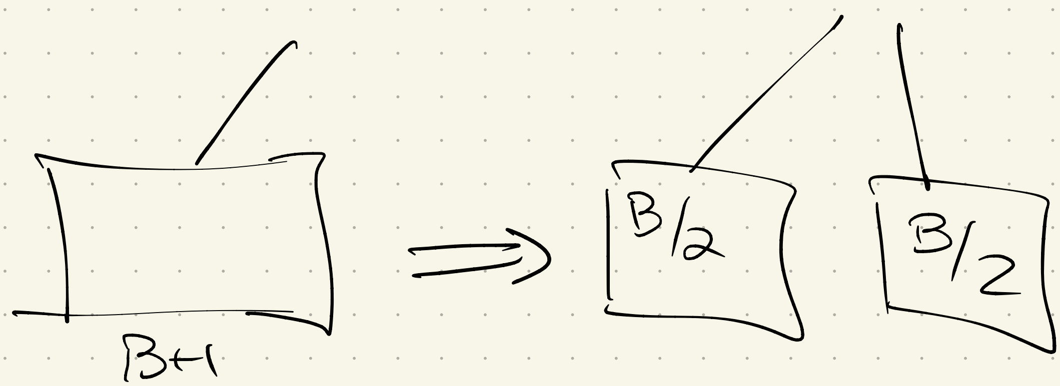
Inside each block: size B array. We need to find here!

Insert: A bit more complex:

$O(\log_B n)$ loads

Then traveling back up:

if leaf is full:



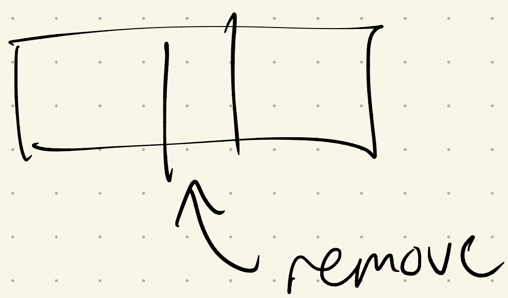
How long?

Runtime:

Delete:

$O(\log_B n)$ loads

Inside each:



So Bad news: (in RAM-model)

Find : $O(\log n)$

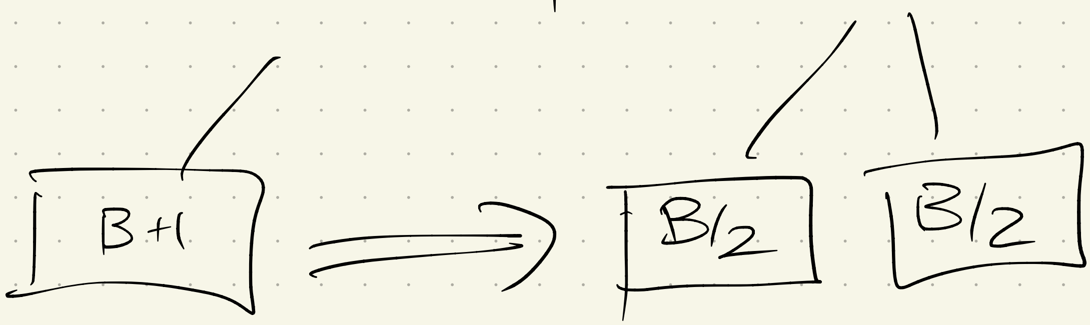
Insert: $O(B \log n)$

Delete : $O(B \log n)$

Well, really?

Think of insert:

after we split



things are empty!

(Remember that push-back in a vector is worst case $O(n)$, but amortized $O(1)$ time?)

Thm: Any sequence of m Insert/Remove operations results in $O(m)$ splits, merges, or borrows.

Result: $O(\log n)$ amortized time per operation

Proof: Accounting version again.

Each insert "pays" \$3 (instead on \$1)

By the time a node buffer is full, has built up $\$(3-1)\times(B/2) = \$B$ to pay for its split/merge.

Practical notes

These are (arguably) the most used BST's!

- File systems: Apple's HFS+, MS's NTFS, + Linux Ext4
- Every major database system
- Cloud computing

See linked reference (in "Open DS") for code: Java, Python, or C++

One reason: these work better than expected.

- B is usually big: 100's or 1000's, at least
- So 99% of data is in the leaves

Result:

- Load entire tree in RAM/local memory
- Then a single leaf access to get data