B^{ε}-trees

CSCI 333

Williams College

Logistics

- Lab 2b
  - Office hours Tuesday night, 7-9pm
- Final Project Proposals
  - Due Friday — Come see me!

Last Class

- General principles of write optimization
- LSM-trees
  - Operations
  - Performance
- LevelDB - SSTables store key-value pairs at each level
- PebblesDB - Fragmented LSM
- WiscKey - Separates keys (LSM) from values (log)

This Class

- B^{ε}-trees
  - Operations
  - Performance
- Choosing Parameters
- Compare to B-trees and LSM-trees

But first… Tradeoffs

What are some of the tradeoffs we've discussed so far in topics we've covered?

Big Picture: Write-Optimized Dictionaries

- New class of data structures developed in the '90s
  - LSM Trees[O'Neil, Cheng Gawlick, & O'Neil '96]
  - B^{ε}-trees[Brodal & Fagerberg '03]
  - COLAs[Bender, Farach-Colton, Fineman, Fogel, Kuzmaul & Nelson '07]
  - xDicts[Brodal, Demaine, Fineman, Iacono, Langerman & Munro '10]

- WOD queries are asymptotically as fast as a B-tree (at least they *can be* in "good" WODs)

- WOD inserts/updates/deletes are **orders-of-magnitude** faster than a B-tree

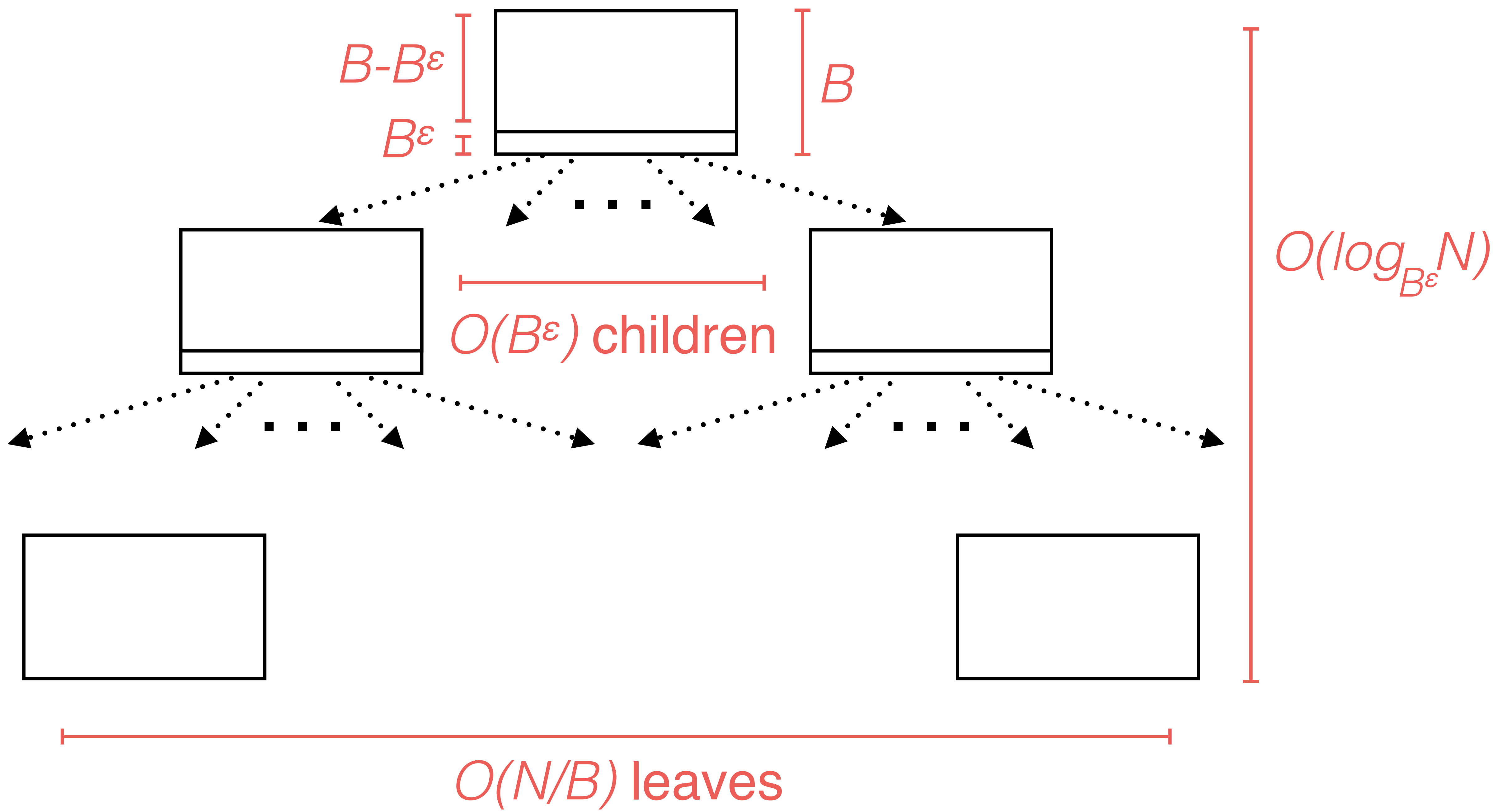
B^{ε}-trees [Brodal & Fagerberg '03]

- B^{ε}-trees: an asymptotically optimal key-value store
  - Fast in best cases, bounds on worst-cases
- B^{ε}-tree searches are just as fast as* B-trees
- B^{ε}-tree updates are **orders-of-magnitude** faster*

*asymptotically, in the DAM model

B and ε are parameters:

- **B ➡ how much "stuff" fits in one node**
- **ε ➡ fanout ➡ how tall the tree is**



B^{ε}-trees [Brodal & Fagerberg '03]

- B^{ε}-tree leaf nodes store key-value pairs
- Internal B^{ε}-tree node buffers store *messages*
  - Messages target a specific key
  - Messages encode a mutation
- Messages are *flushed* downwards, and eventually *applied* to key-value pairs in the leaves

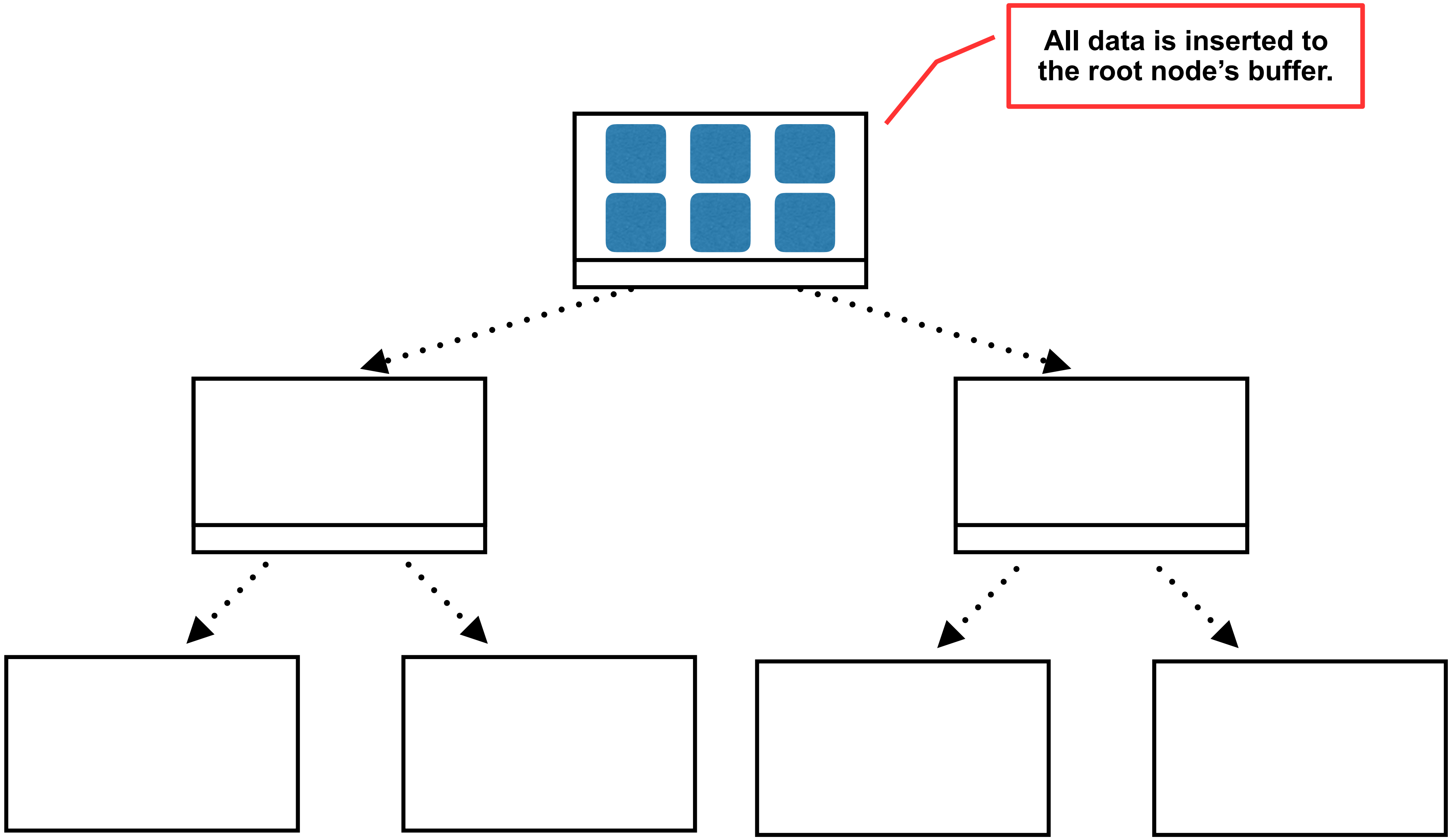
High-level: messages + LSM/B-tree hybrid

B^ε-tree Operations

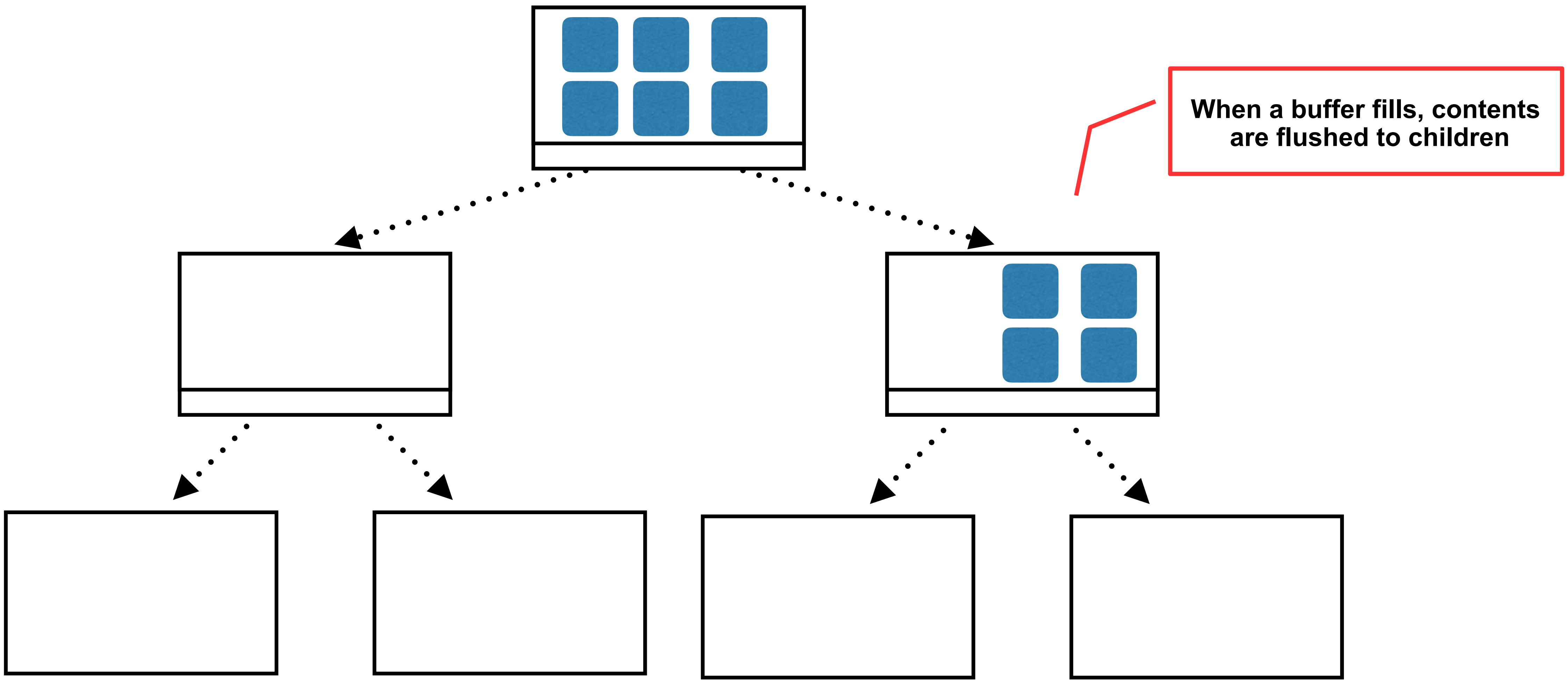
- Implement a dictionary on key-value pairs
  - `insert(`$\mathbf{k}$`,`$\mathbf{v}$`)`
  - $\mathbf{v}$ `= search(`$\mathbf{k}$`)`
  - $\{(\mathbf{k}_i,\mathbf{v}_i), \ldots (\mathbf{k}_j, \mathbf{v}_j)\}$ `= search(`$\mathbf{k}_1$`, `$\mathbf{k}_2$`)`
  - `delete(`$\mathbf{k}$`)`
- New operation:
  - `upsert(`$\mathbf{k}$`, `f`, `$\mathbf{\Delta}$`)`

Talk about soon!

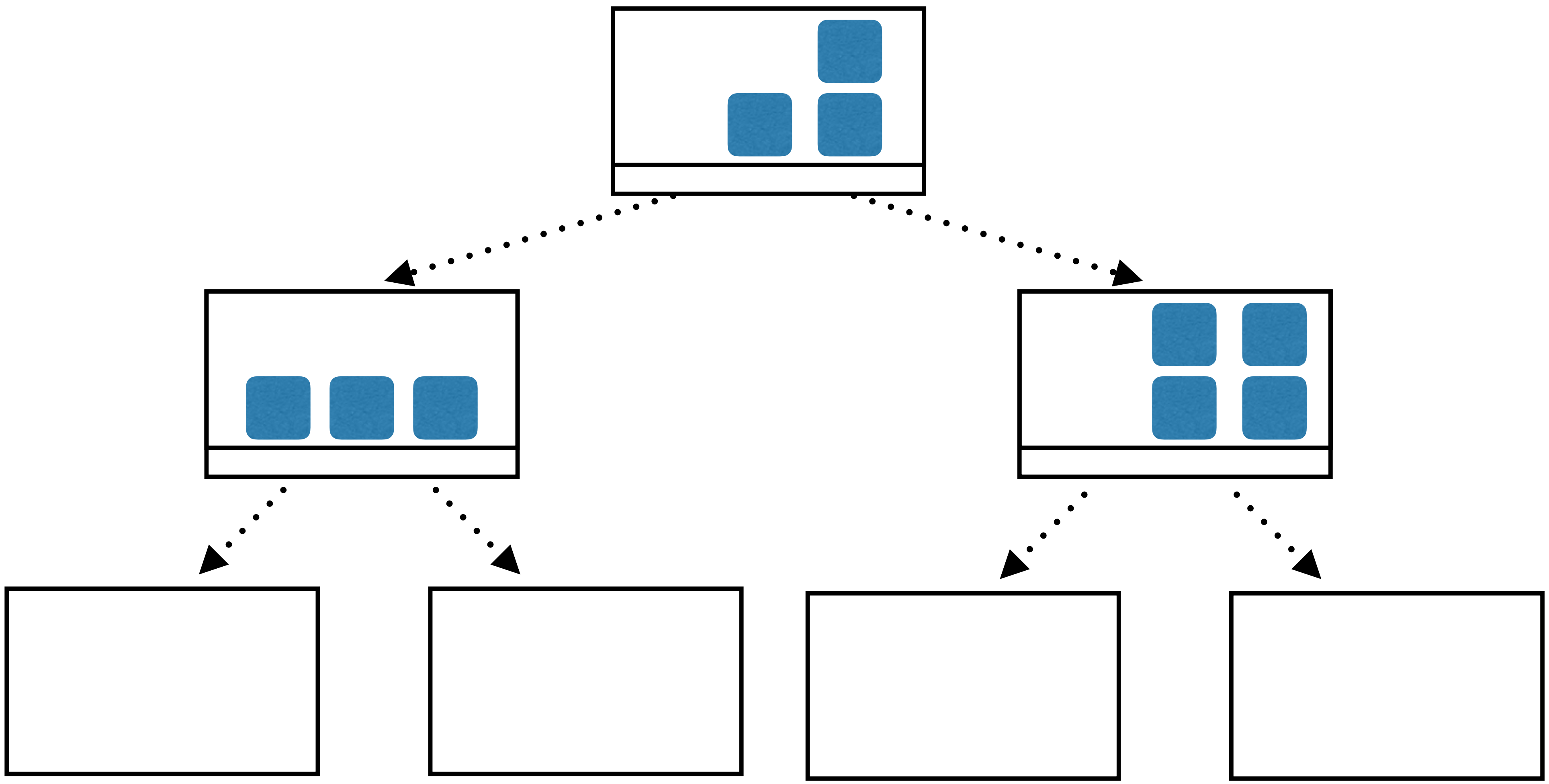
Bε-tree Inserts



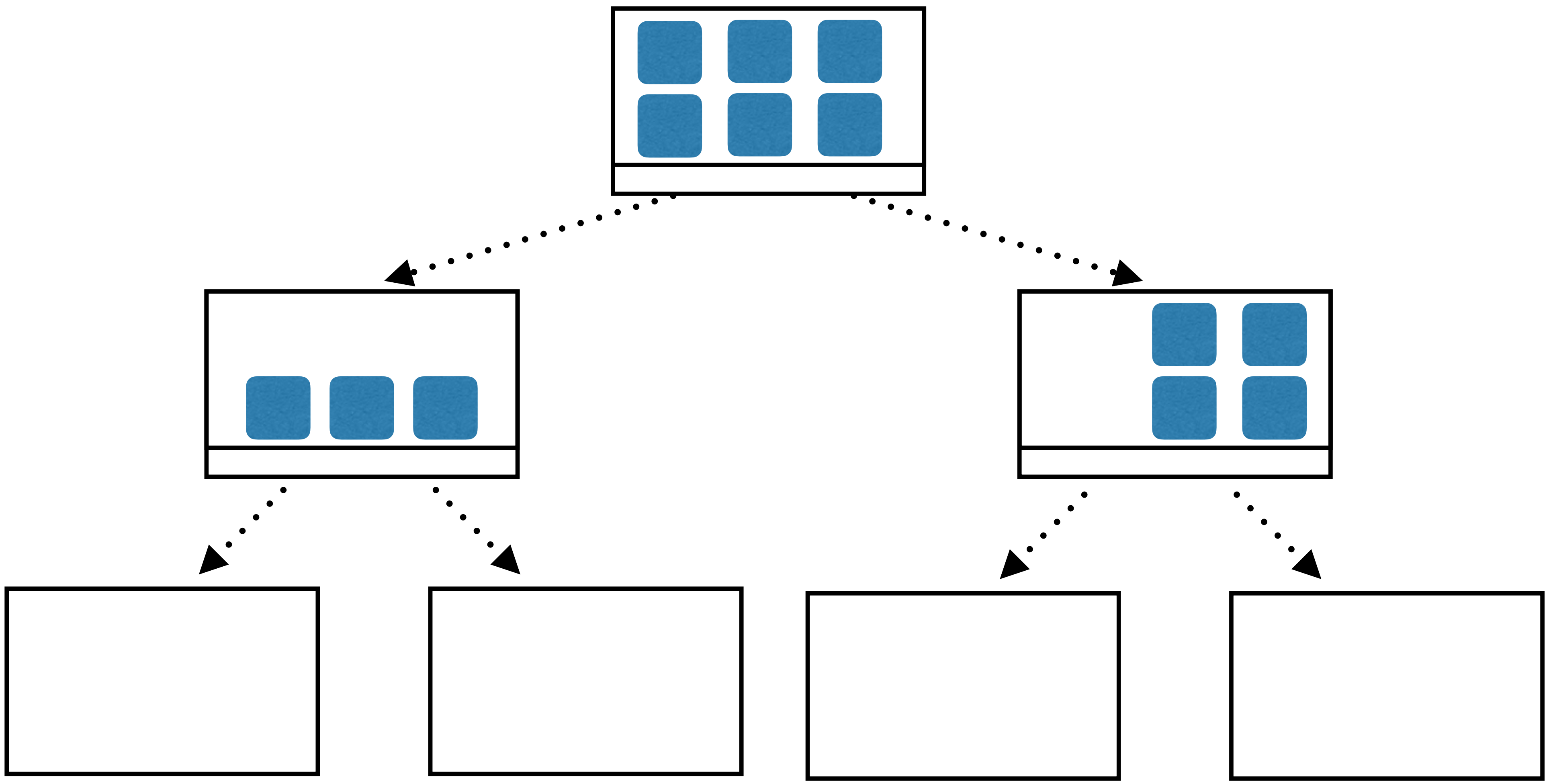
Bε-tree Inserts



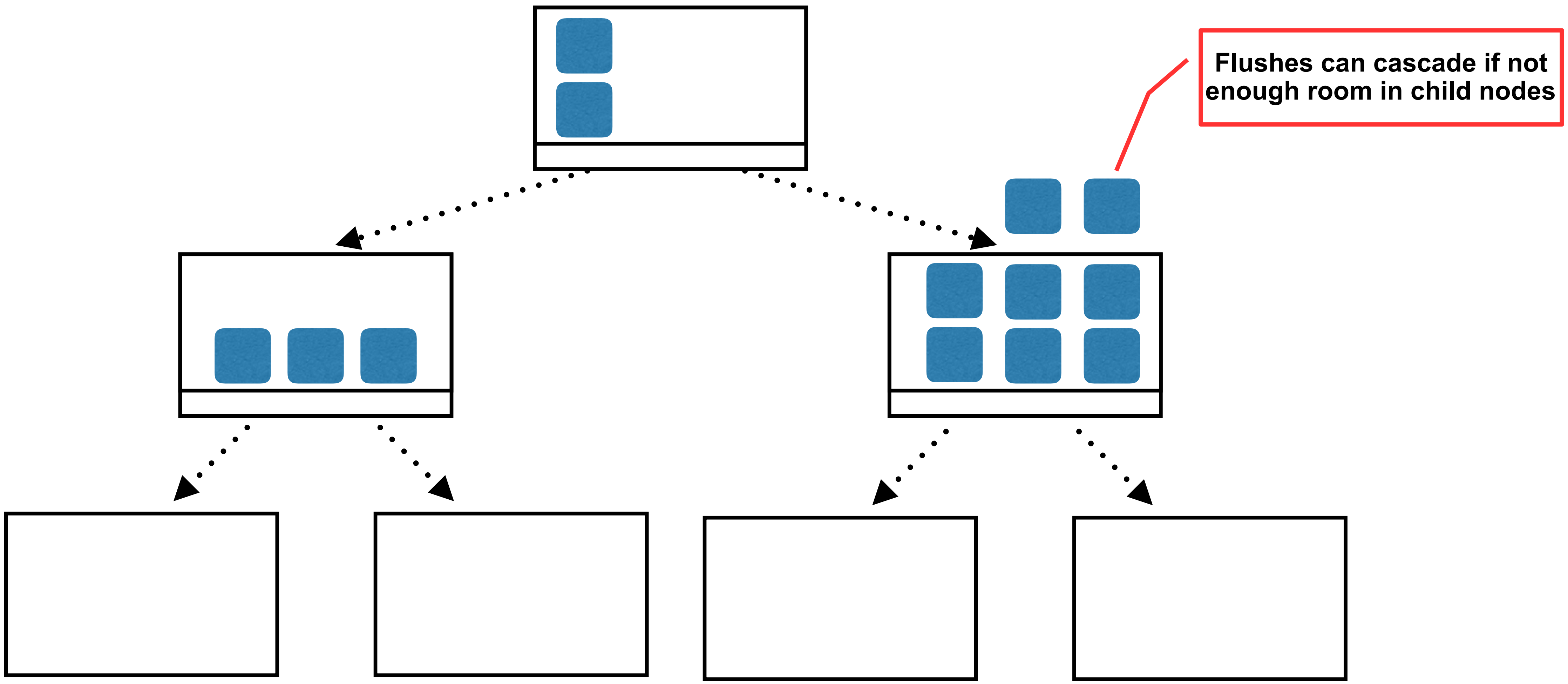
Bε-tree Inserts



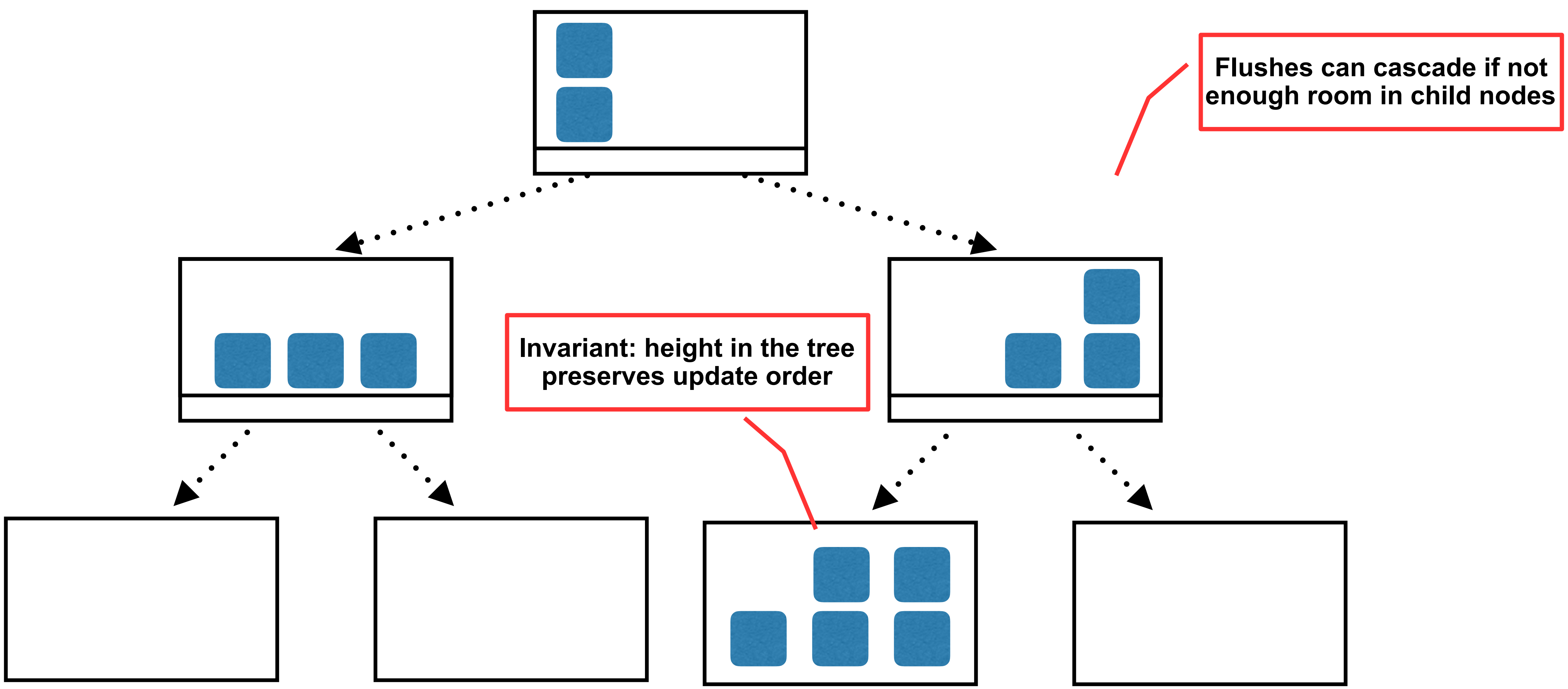
B$^\varepsilon$-tree Inserts



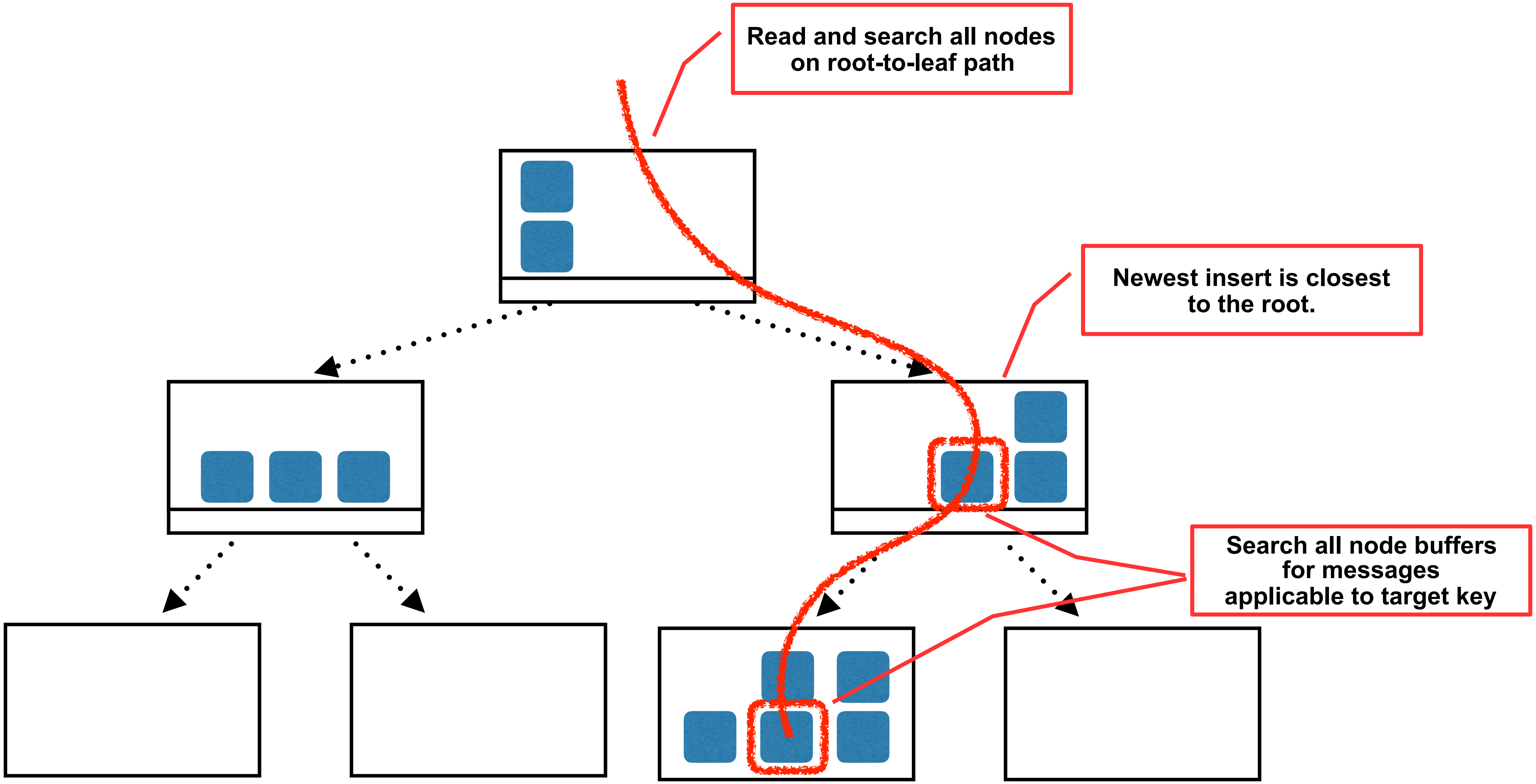
Bε-tree Inserts



Bε-tree Inserts



Bε-tree Searches



Updates

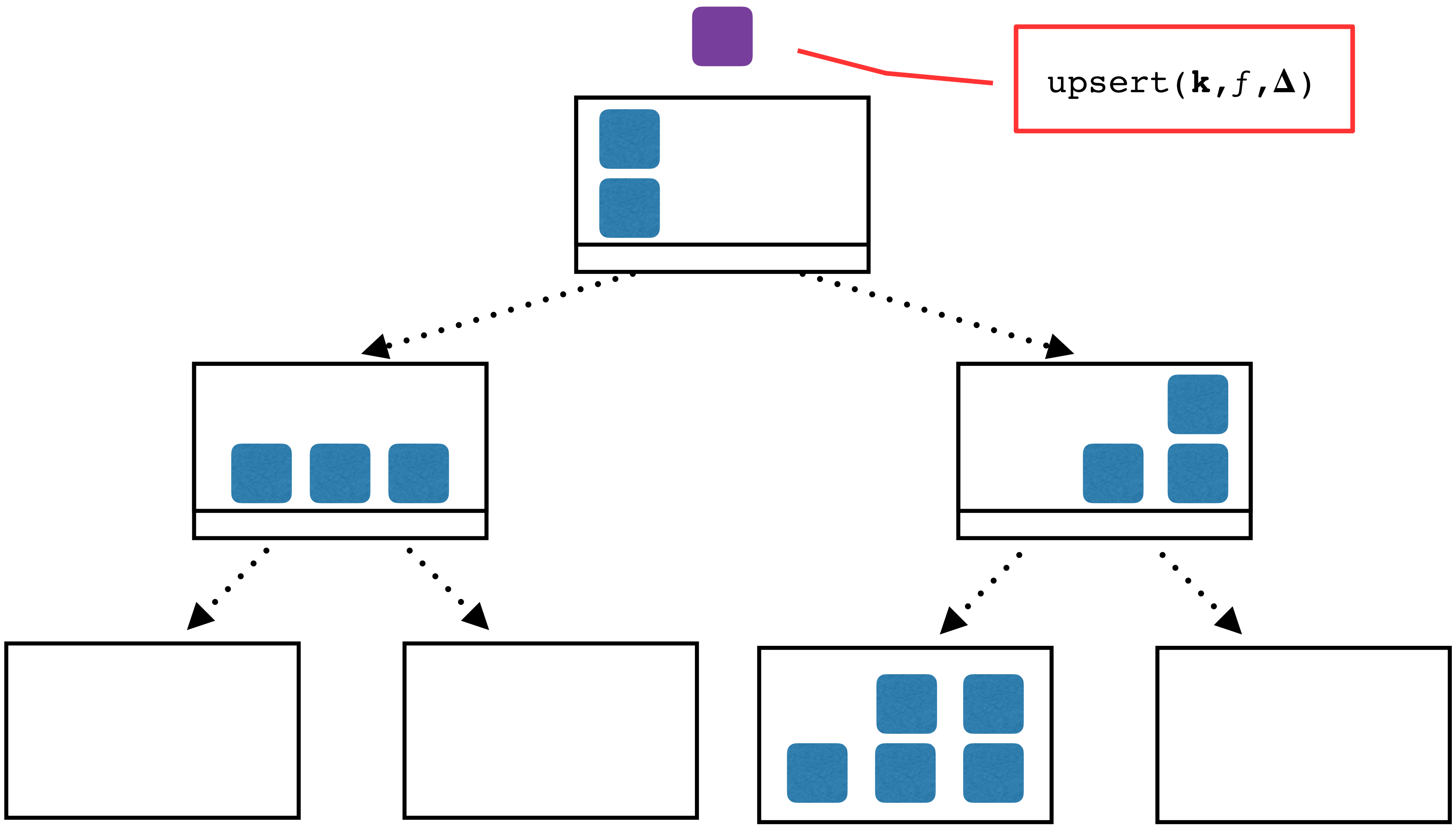
- In most systems, updating a value requires:
  read, modify, write FUSE FAT write?
- **Problem:** B^{ε}-tree inserts are faster than searches
  - fast updates are impossible if we must search first

```
upsert = update + insert
```

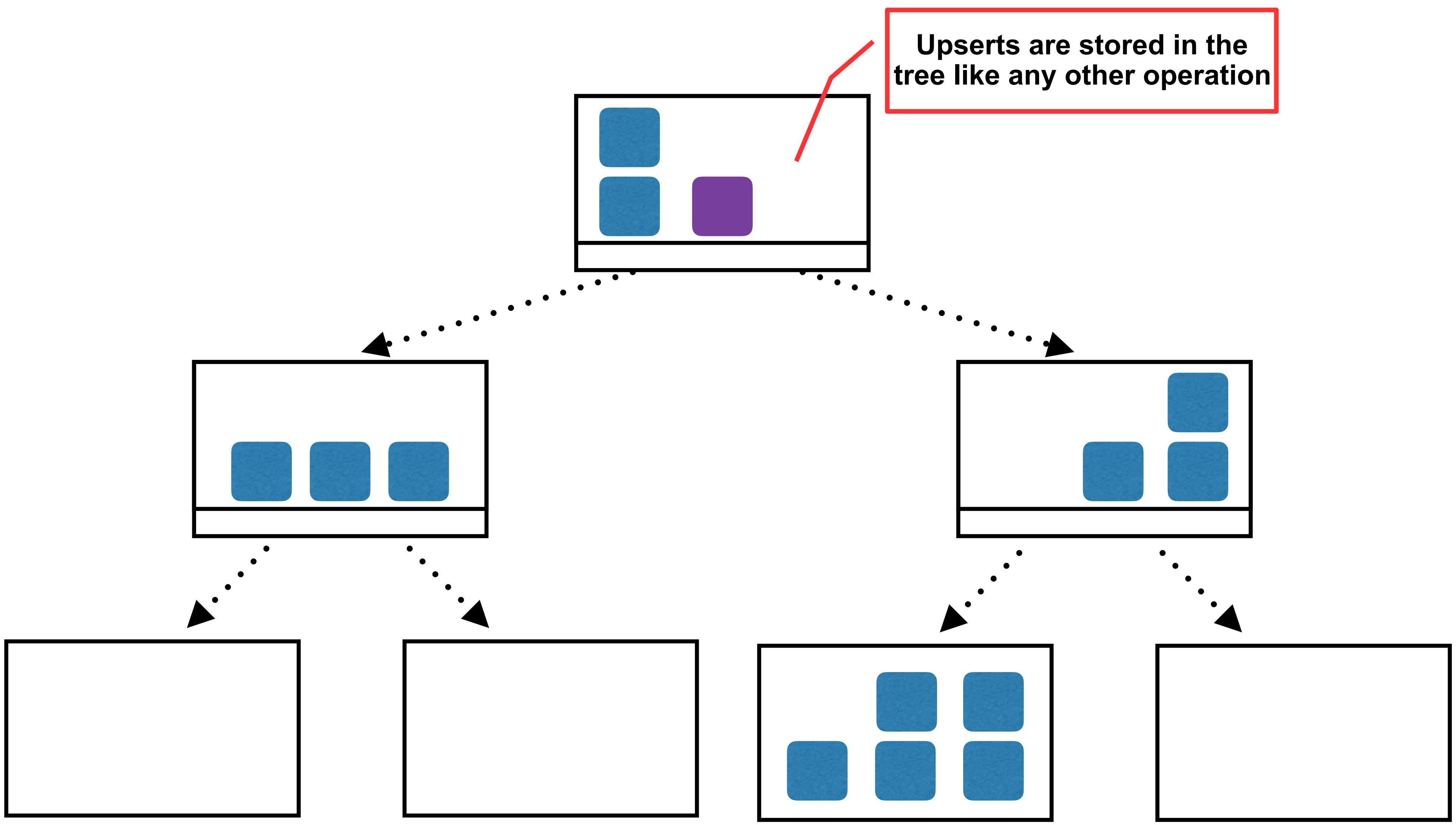
Upsert messages

- Each upsert message contains a:
  - Target key, $\mathbf{k}$
  - Callback function, $\boldsymbol{f}$
  - Set of function arguments, $\boldsymbol{\Delta}$
- Upserts are added into the B^{ε}-tree like any other message
- The callback is evaluated whenever the message is applied
  - Upserts can specify a modification and lazily do the work

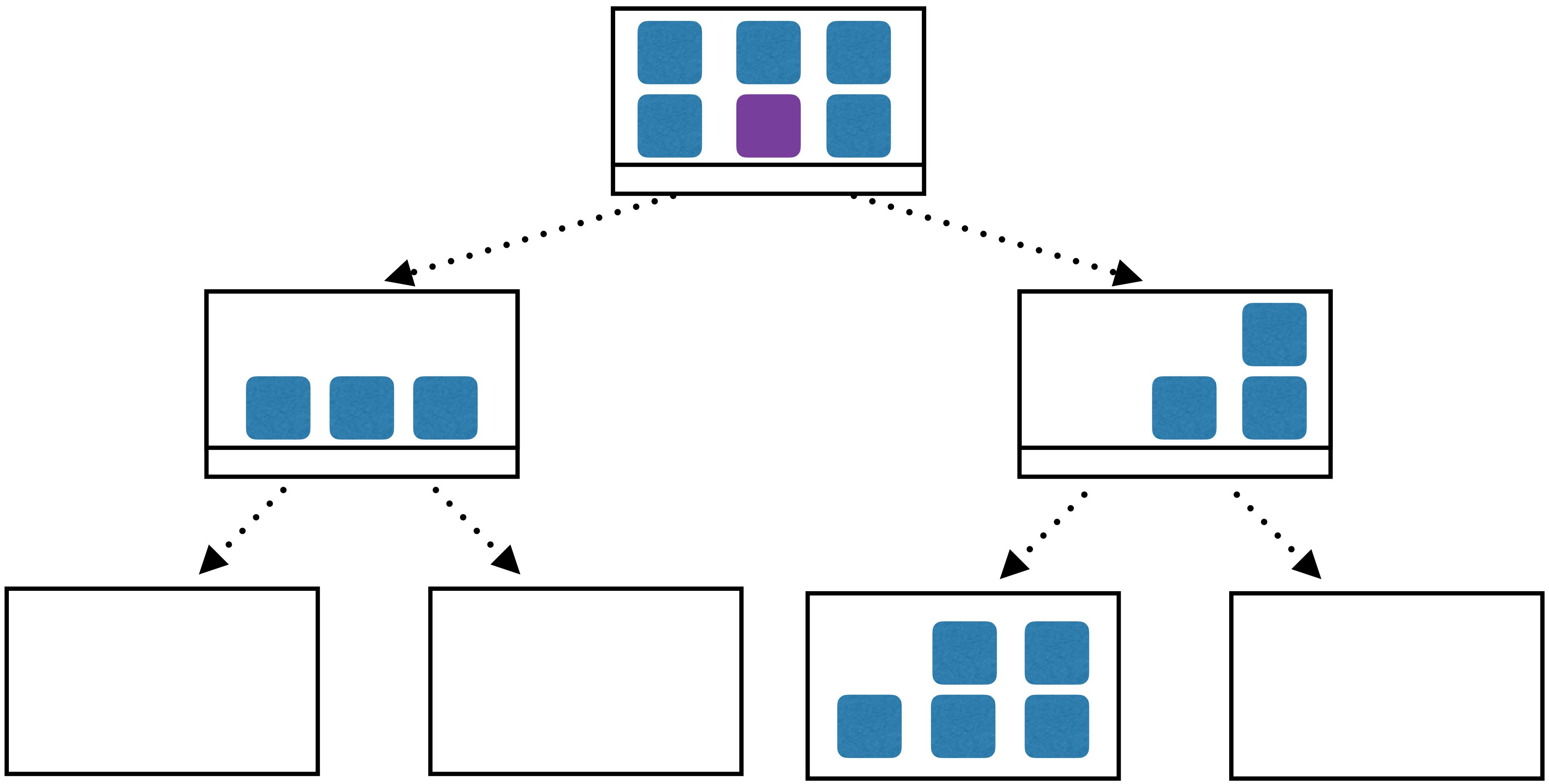
Bε-tree Upserts



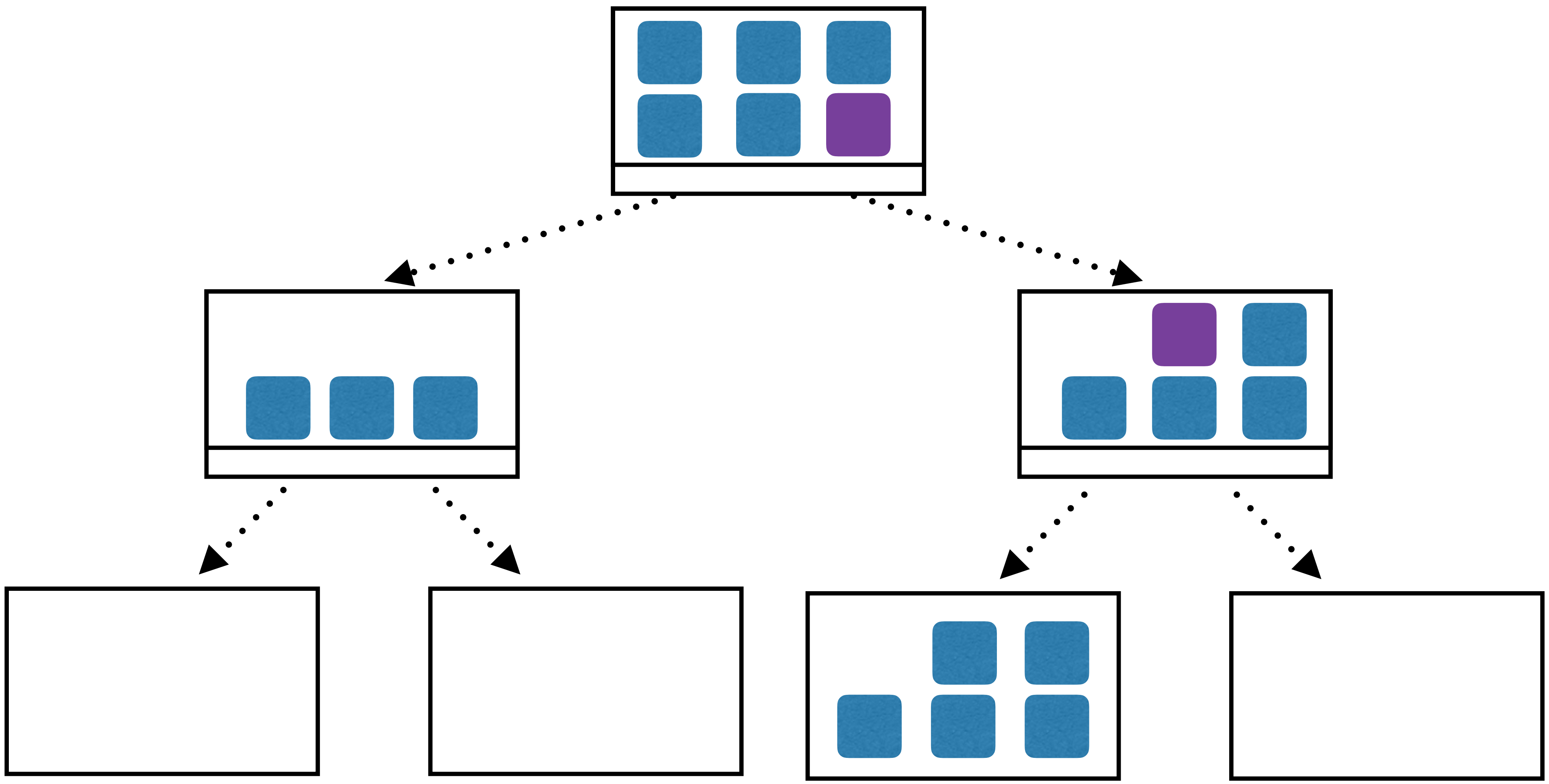
Bε-tree Upserts



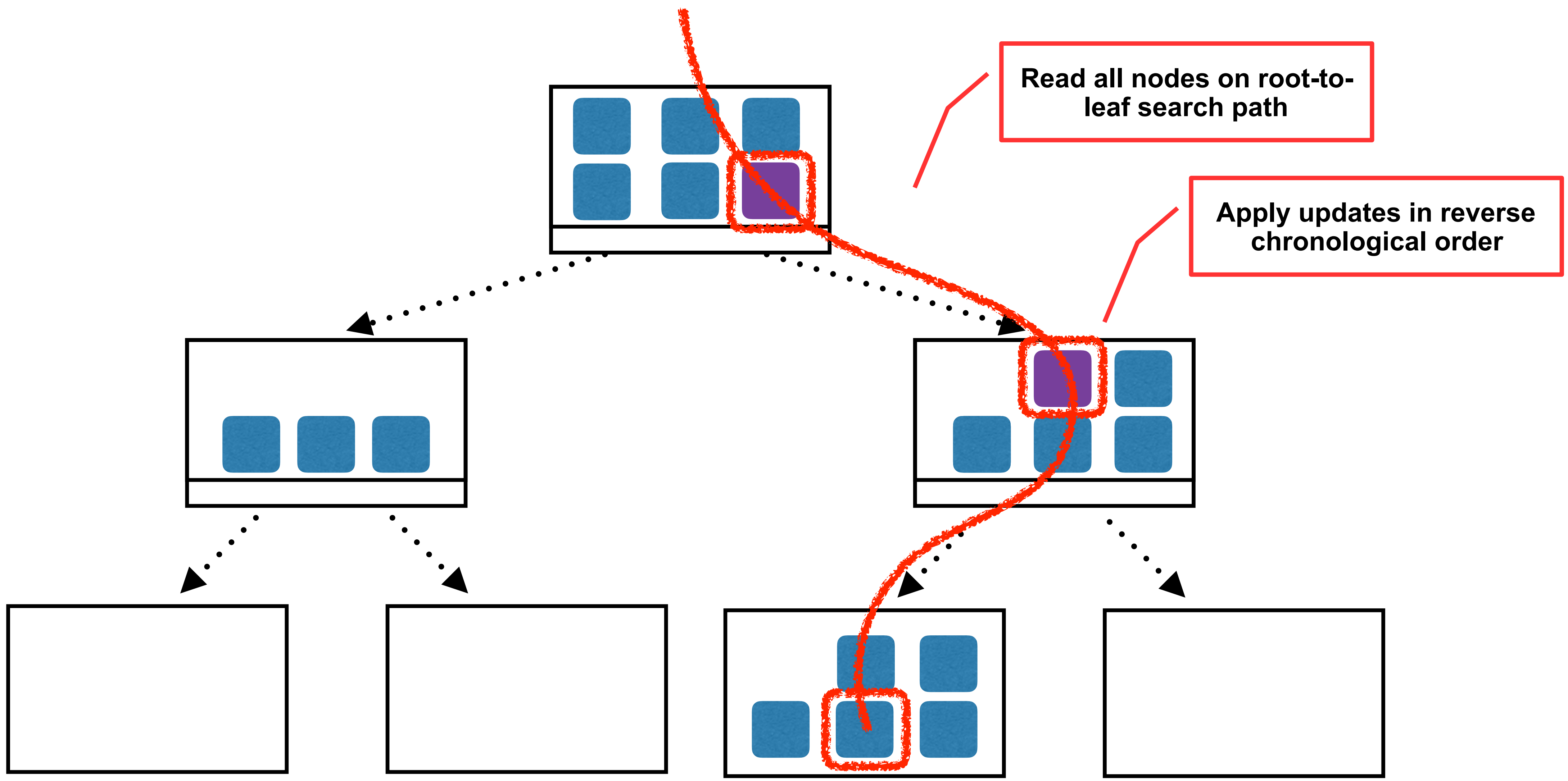
Bε-tree Upserts



B$^{\varepsilon}$-tree Upserts



Searching with Upserts



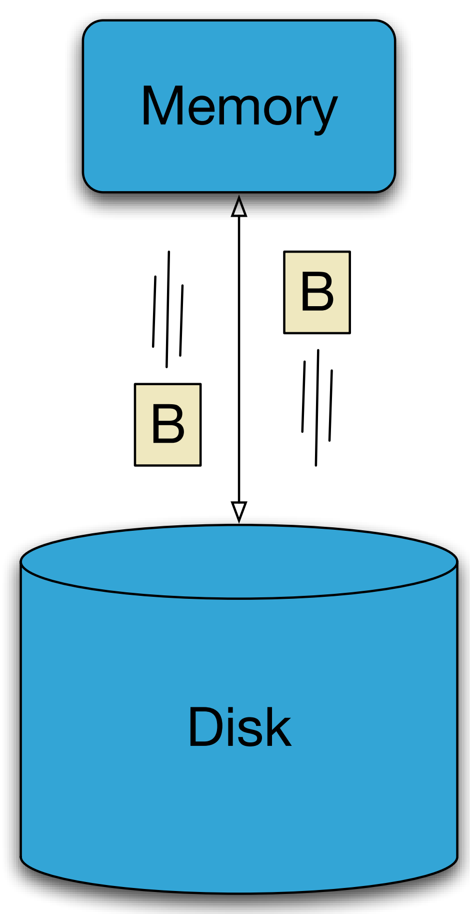
Upserts don’t harm searches, but they let us perform **blind updates**.

Thought Question

- What types of operations might naturally be encoded as upserts?

Performance Model

- Disk Access Machine (DAM) Model [Aggarwal & Vitter '88]
- Idea: expensive part of an algorithm's execution is transferring data to/from memory
- Parameters:
  - **B**: block size
  - **M**: memory size
  - **N**: data size

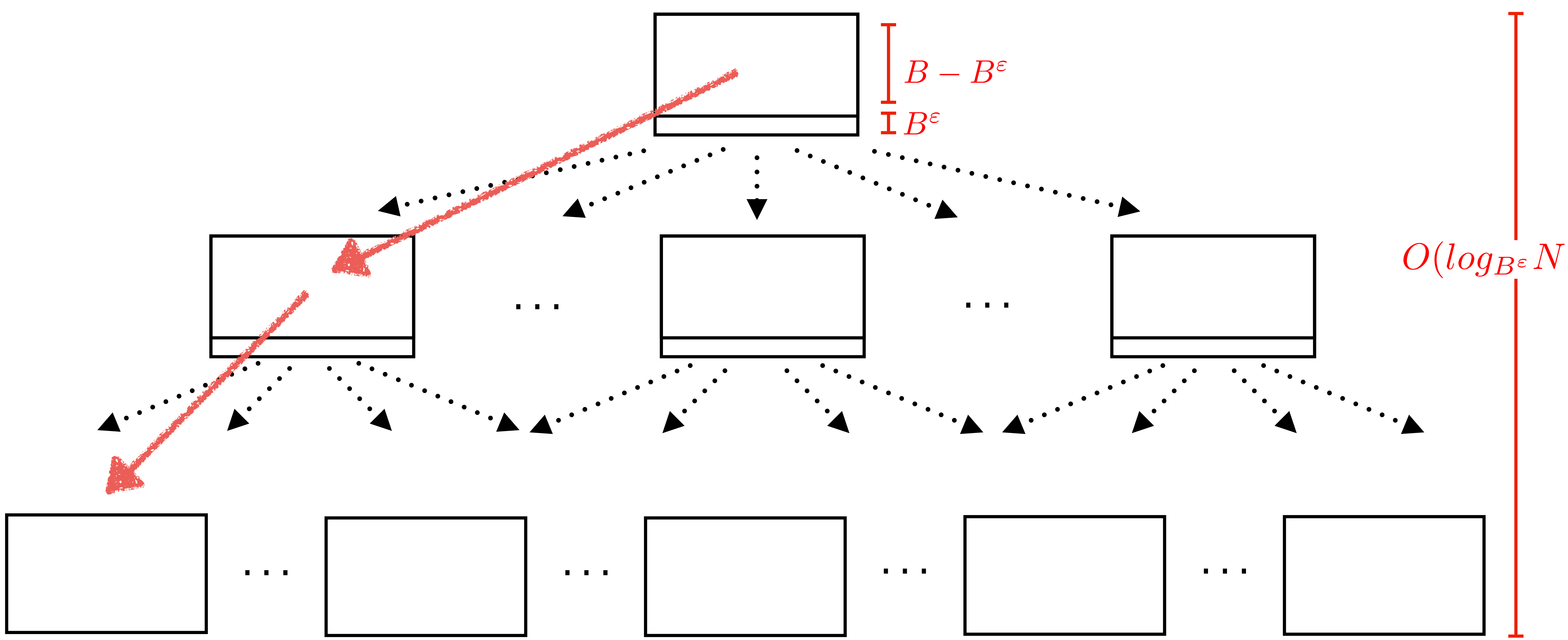


Performance = (# of I/Os)

Point Query: ?

Range Query:

Insert/upsert:



Goal: Compare query performance to a B-tree $O(\log_B N)$

➡ B^ε-tree fanout: B^ε

➡ B^ε-tree height: $O(log_{B^\varepsilon} N)$

Different bases…

Rule 1: $\log_b(M \cdot N) = \log_b M + \log_b N$

Rule 2: $\log_b\left(\frac{M}{N}\right) = \log_b M - \log_b N$

Rule 3: $\log_b\left(M^k\right) = k \cdot \log_b M$

Rule 4: $\log_b(1) = 0$

Rule 5: $\log_b(b) = 1$

Rule 6: $\log_b\left(b^k\right) = k$

Rule 7: $b^{\log_b(k)} = k$

Where: $b > 1$, and M, N and k can be any real numbers

but M and N must be positive!

[https://www.chilimath.com/lessons/advanced-algebra/logarithm-rules/]

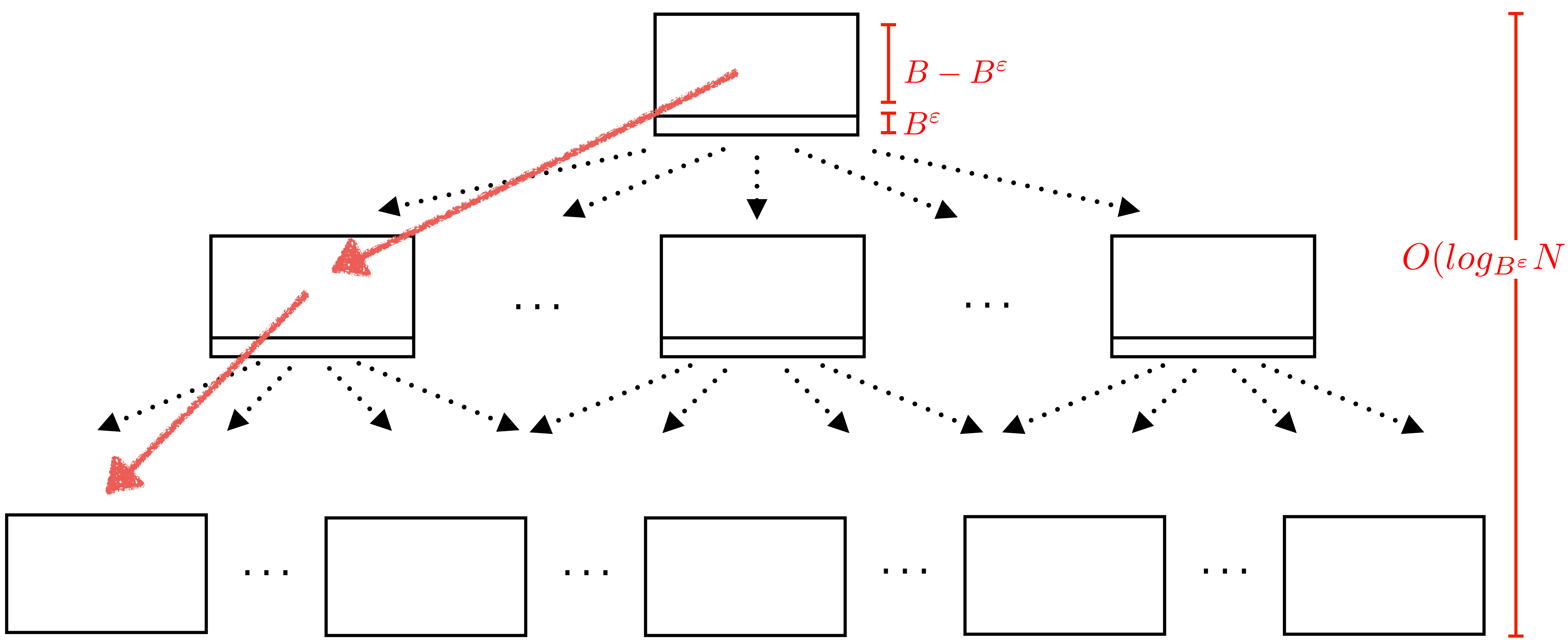
$$\log_b(a) = \frac{\log_x(a)}{\log_x(b)}$$

[https://www.khanacademy.org]

Point Query: $O(\frac{\log_B N}{\varepsilon})$

Range Query: ?

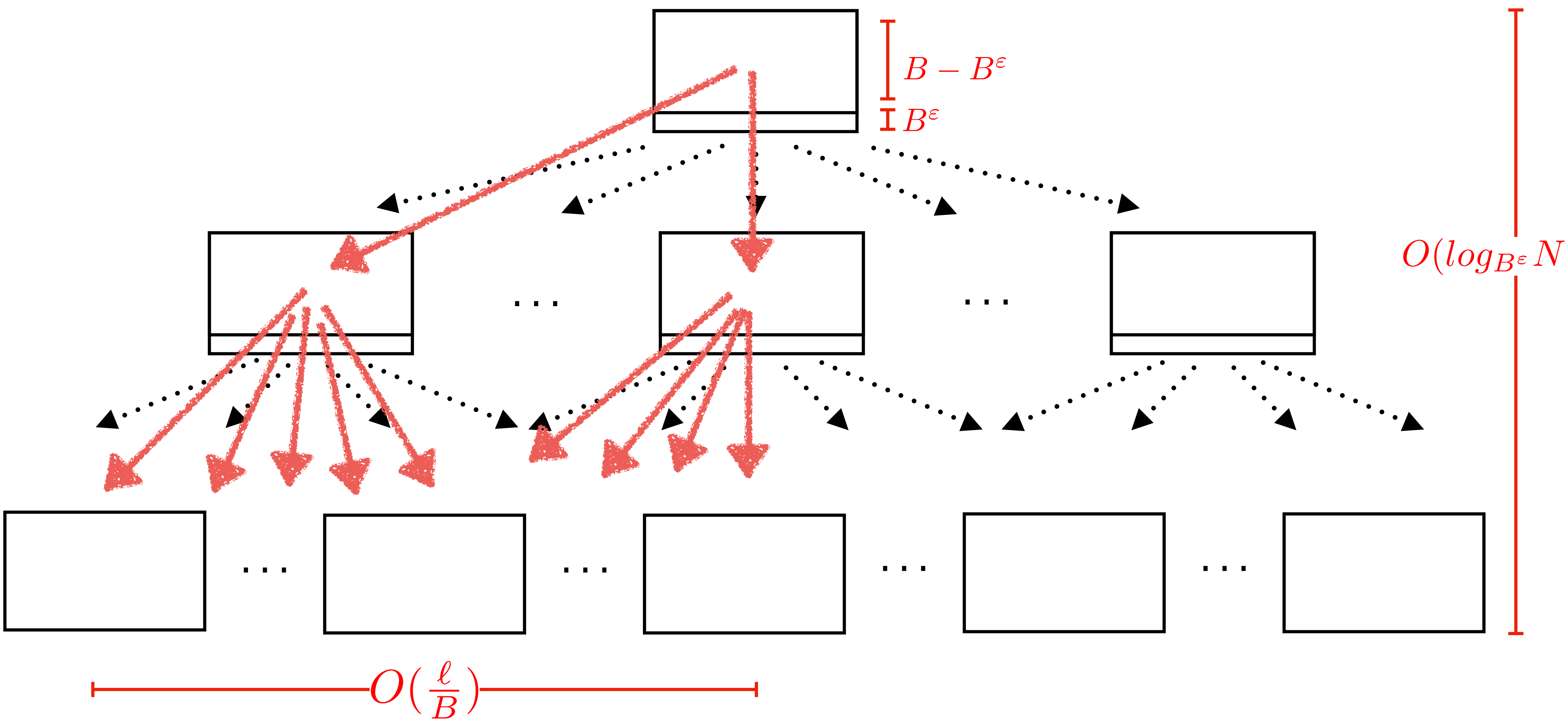
Insert/upsert:



Point Query: $O(\frac{\log_B N}{\varepsilon})$

Range Query: $O(\frac{\log_B N}{\varepsilon} + \frac{\ell}{B})$

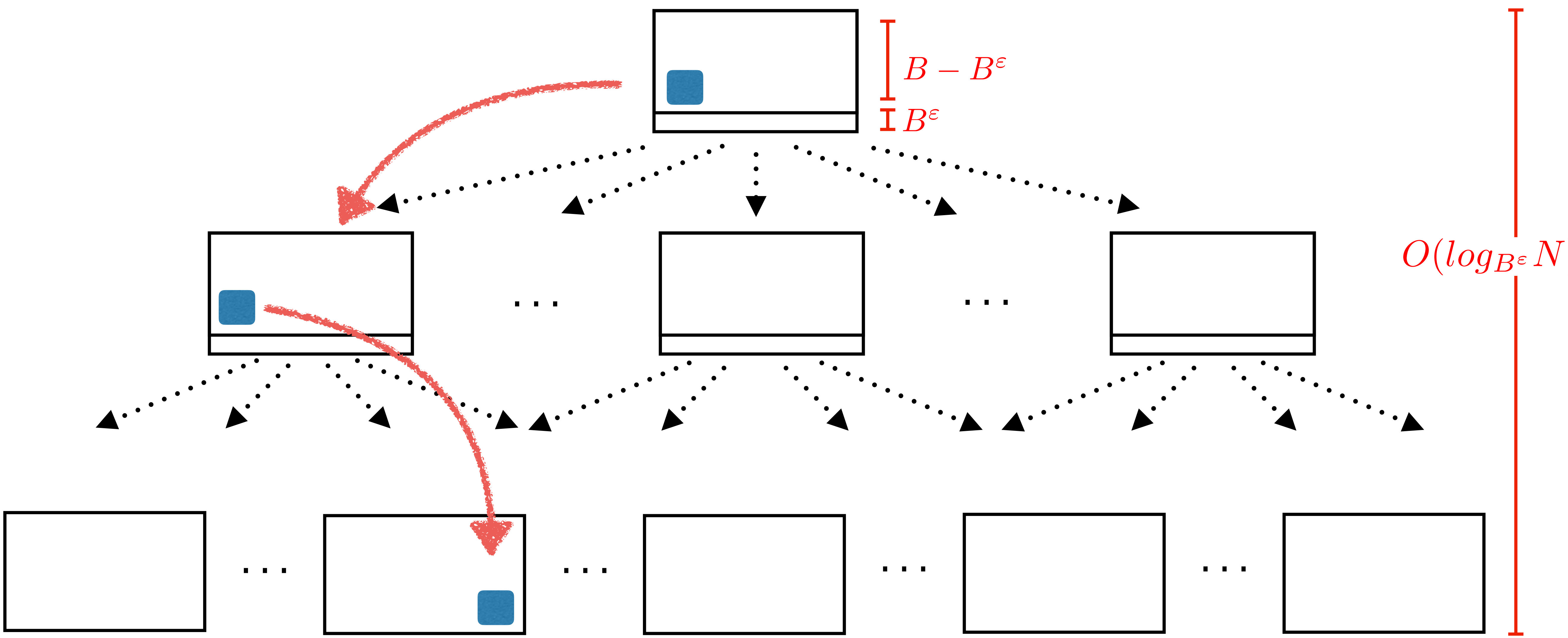
Insert/upsert: **?**



Point Query: $O(\frac{\log_B N}{\varepsilon})$

Range Query: $O(\frac{\log_B N}{\varepsilon} + \frac{\ell}{B})$

Insert/upsert: **?**



Goal: Attribute the cost of flushing across all messages that benefit from the work.

➡ How many times is an insert flushed? $O(log_{B^\varepsilon} N)$

➡ How many messages are moved per flush? $\mathbf{O}(\frac{\mathbf{B}-\mathbf{B}^\varepsilon}{\mathbf{B}^\varepsilon})$

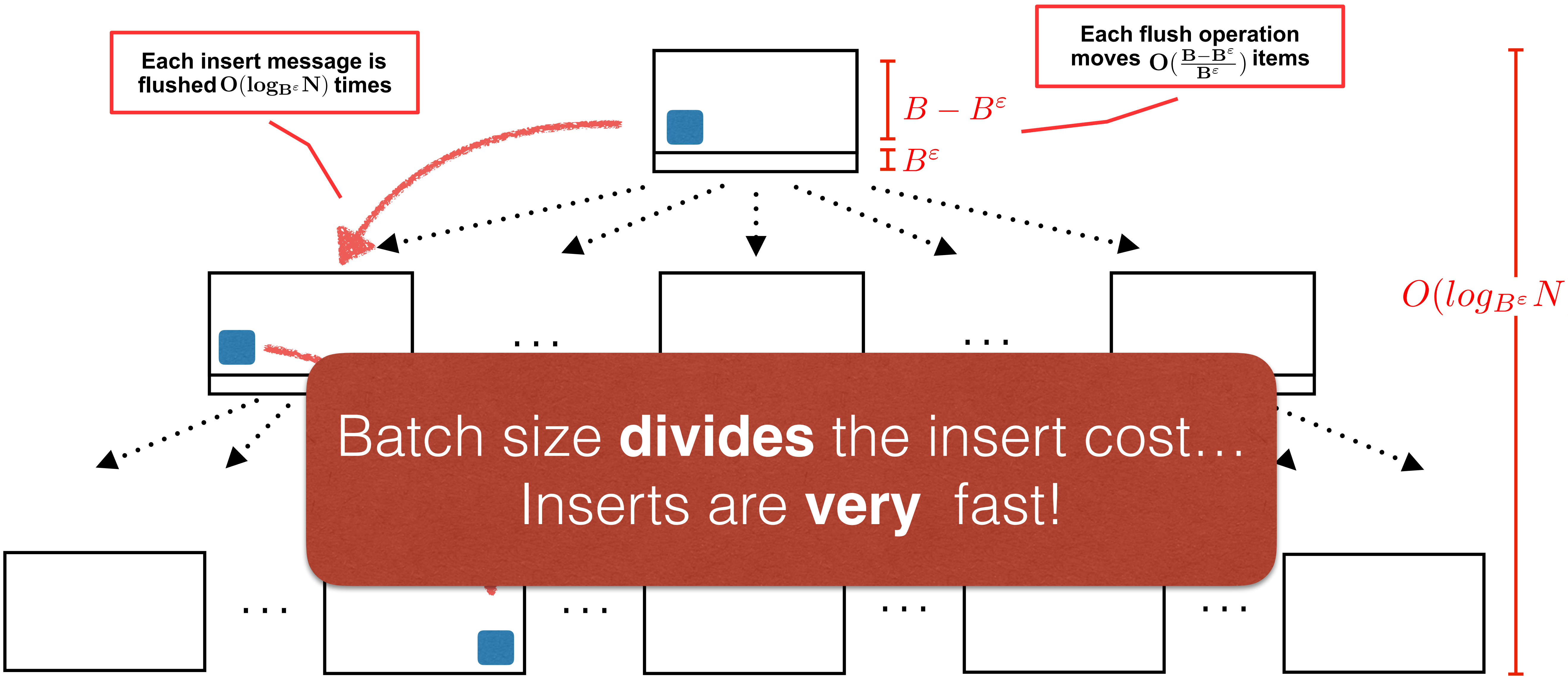
➡ How do we “share the work” among the messages?

- Divide by the total cost by the number of messages

Point Query: $O(\frac{\log_B N}{\varepsilon})$

Range Query: $O(\frac{\log_B N}{\varepsilon} + \frac{\ell}{B})$

Insert/upsert: $O(\frac{\log_B N}{\varepsilon B^{1-\varepsilon}})$



Batch size **divides** the insert cost…
Inserts are **very** fast!

Recap/Big Picture

- Disk seeks are slow ➡ big I/Os improve performance
- B^{ε}-trees convert small updates to large I/Os
  - Inserts: orders-of-magnitude faster
  - Upserts: let us update data without reading
  - Point queries: as fast as standard tree indexes
  - Range queries: near-disk bandwidth (w/ large B)

Question: How do we choose **B** and **ε**?

Thought Questions

- How do we choose **ε?**

- Original paper didn't actually use the term B^{ε}-tree (or spend very long on the idea). Showed there are various points on the trade-off curve between B-trees and Buffered Repository trees

$\varepsilon = 1$ corresponds to a B-tree
$\varepsilon = 0$ corresponds to a Buffered Repository tree

Thought Questions

- How do we choose **B**?

- Let's first think about B-trees
  - What changes when B is large?
  - What changes when B is small?

- B^{ε}-trees buffer data; batch size *divides* the insert cost
  - What changes when B is large?
  - What changes when B is small?

In practice choose **B** and "fanout".
B ≈ 2-8MiB, fanout ≈16

Thought Questions

- How does a B^{ε}-tree compare to an LSM-tree?
    - Compaction vs. flushing
    - Queries (range and point)
    - Upserts

Thought Questions

- How would you implement
  - `copy(old, new)`
  - `delete(“large”)` :: kv-pair that occupies a whole leaf?
  - `delete(“a*|b*|c*”)` :: a contiguous range of kv-pairs?

Next Class

- From Be-tree to file system!