B$^{\varepsilon}$-trees

CSCI 333
Williams College
Bill Jannen

Last Class

- General principles of write optimization
  - Buffer updates and apply changes in large batches
- LSM-trees
  - Operations (Dictionary API, i.e., key-value store interface)
  - Performance
- LevelDB - SSTables store key-value pairs at each level
- Compaction strategies
  - **Size-tiered** - compact K SSTables together when there is enough data to merge into the next "size tier"
  - **Level-tiered** - compact one SSTable into all SSTables in the next that have overlapping key ranges

This Class

- B^{ε}-trees
    - Operations
    - Performance
- Choosing parameters to tune performance
- Compare against B-trees and LSM-trees

Big Picture: Write-Optimized K-V Stores

- New class of data structures first 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]

- Queries are asymptotically as fast as a B-tree (at least they *can be* in “good” data structures)

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

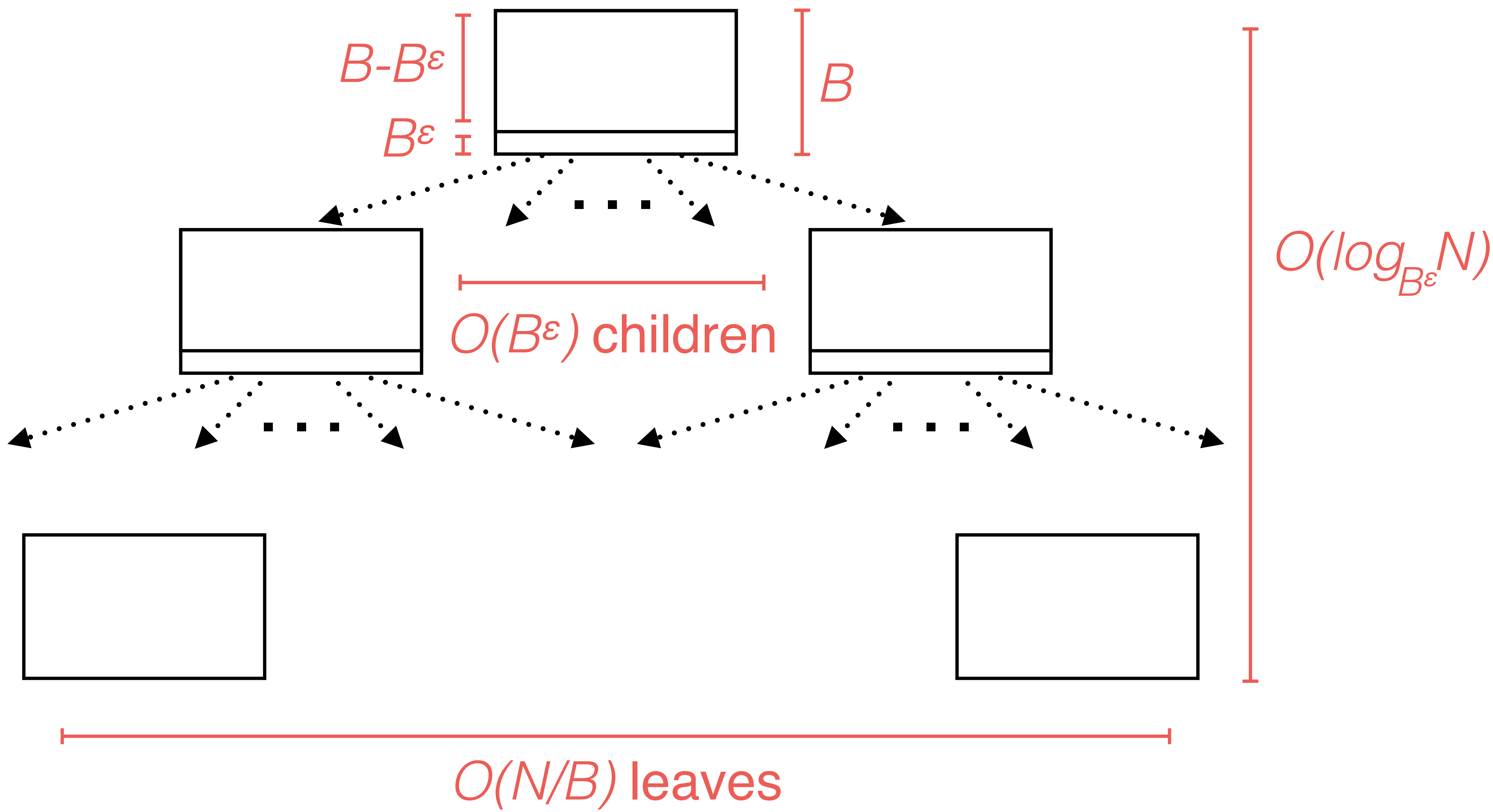
B^{ε}-trees [Brodal & Fagerberg ’03]

- B^{ε}-trees: an asymptotically optimal key-value store
  - Fast in the best cases, good bounds on the 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$^{\varepsilon}$-trees [Brodal & Fagerberg '03]

- B$^{\varepsilon}$-tree leaf nodes store key-value pairs
- Internal B$^{\varepsilon}$-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

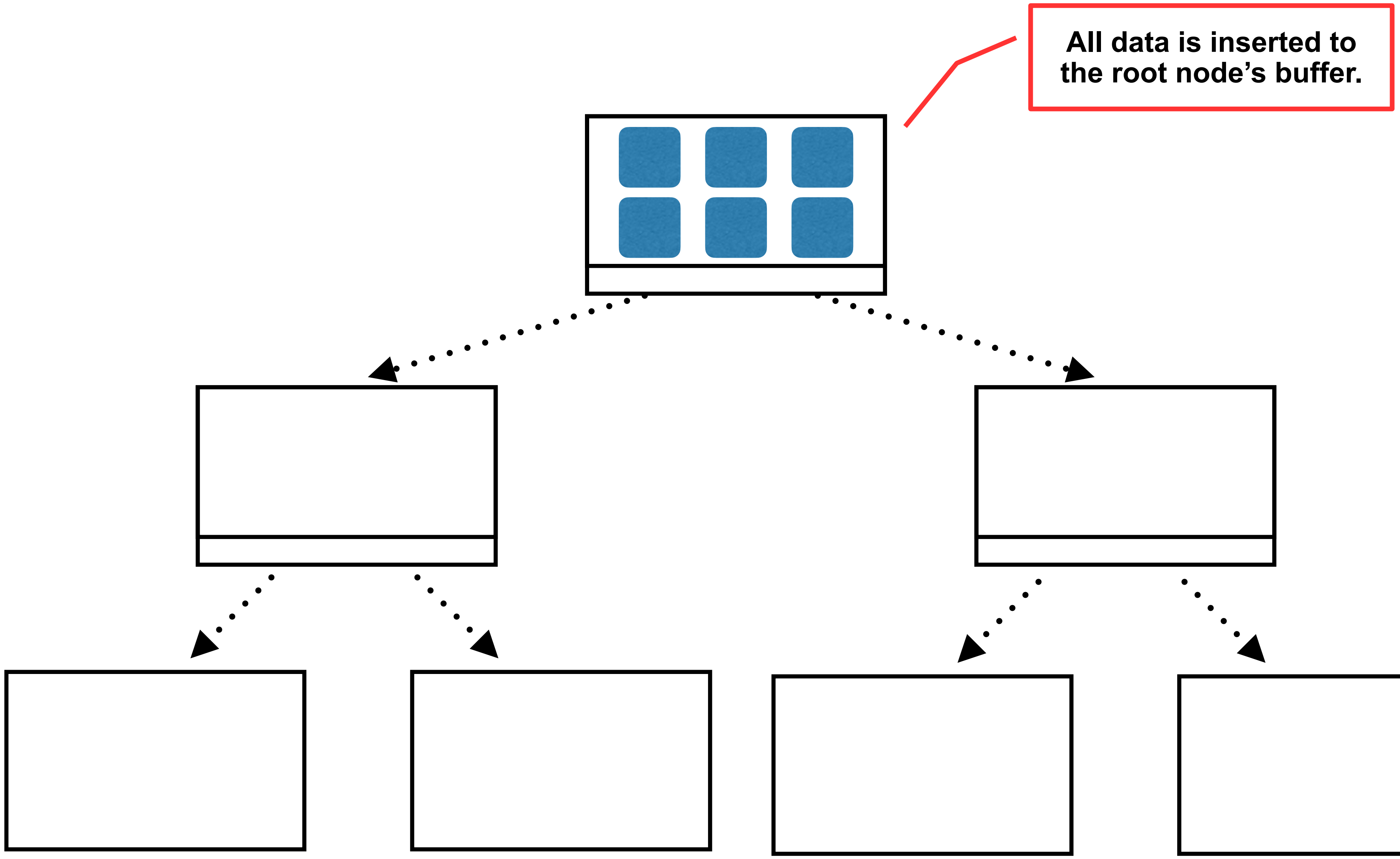
B^ε-tree Operations

- Implement a dictionary on key-value pairs
  - `insert(`**k**`,`**v**`)`
  - **v** `= search(`**k**`)`
  - `{(`$\mathbf{k_i}$`,`$\mathbf{v_i}$`), … (`$\mathbf{k_j}$`, `$\mathbf{v_j}$`)} = search(`$\mathbf{k_1}$`, `$\mathbf{k_2}$`)`
  - `delete(`**k**`)`

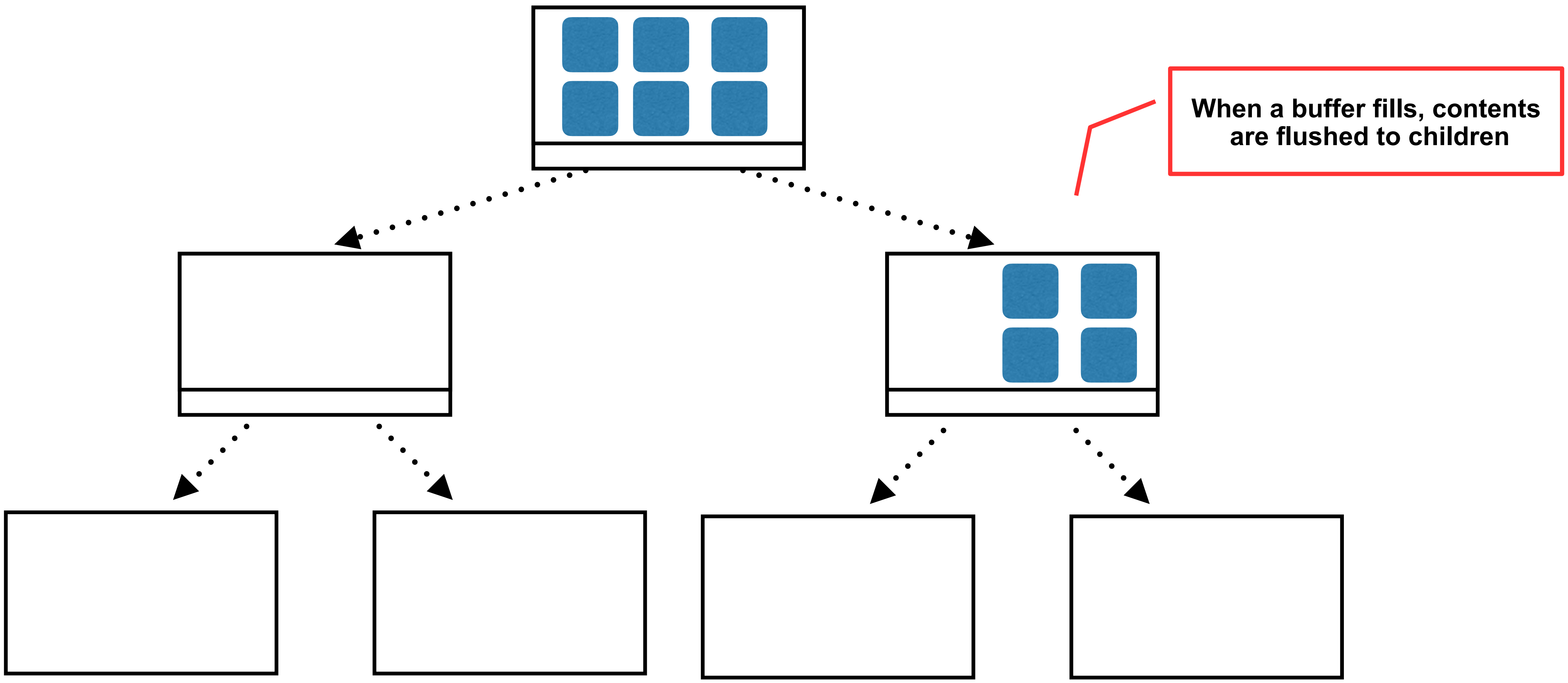
- New operation:
  - `upsert(`**k**`, `f`, `$\mathbf{\Delta}$`)`

Talk about soon!

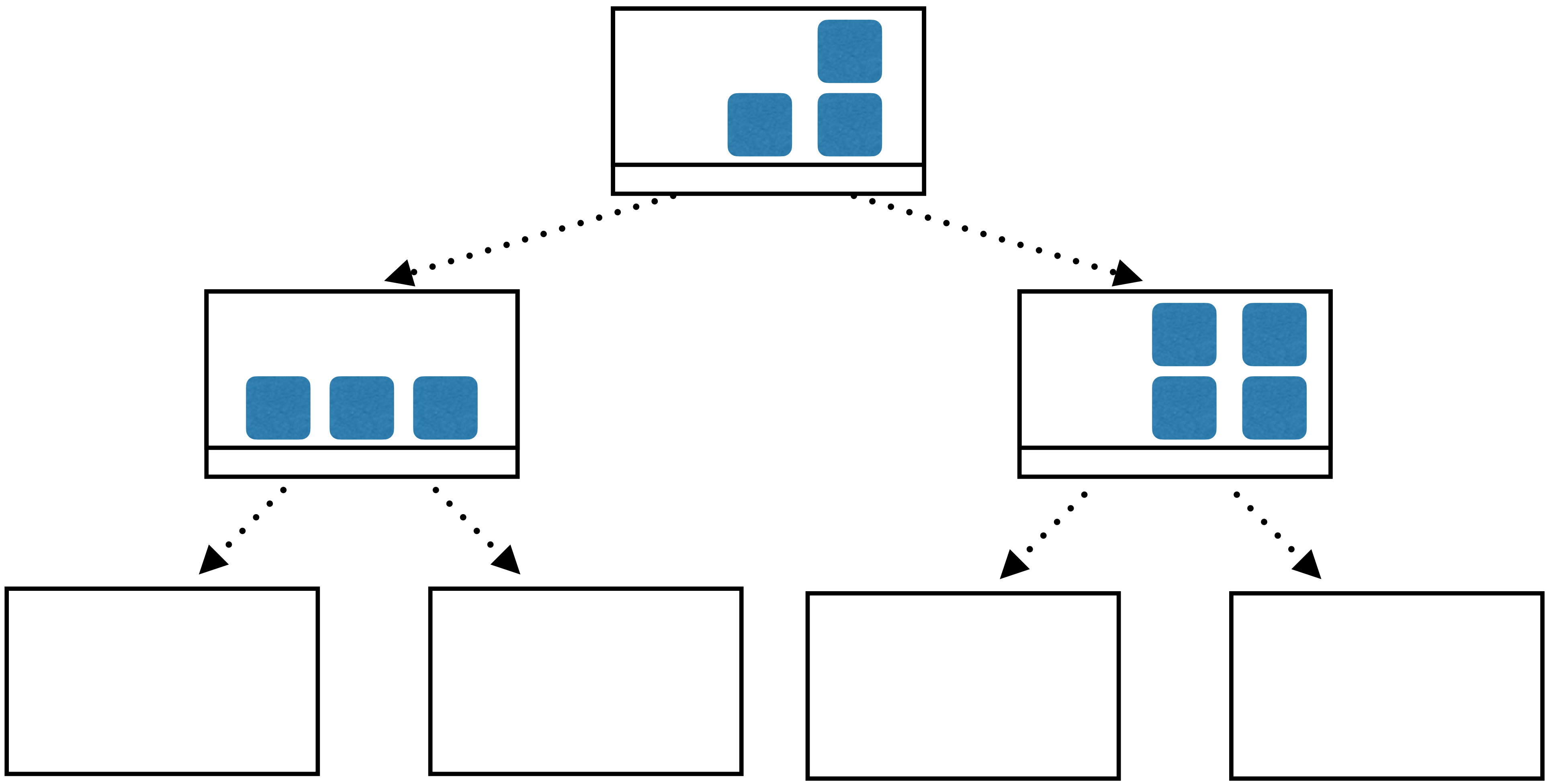
Bε-tree Inserts



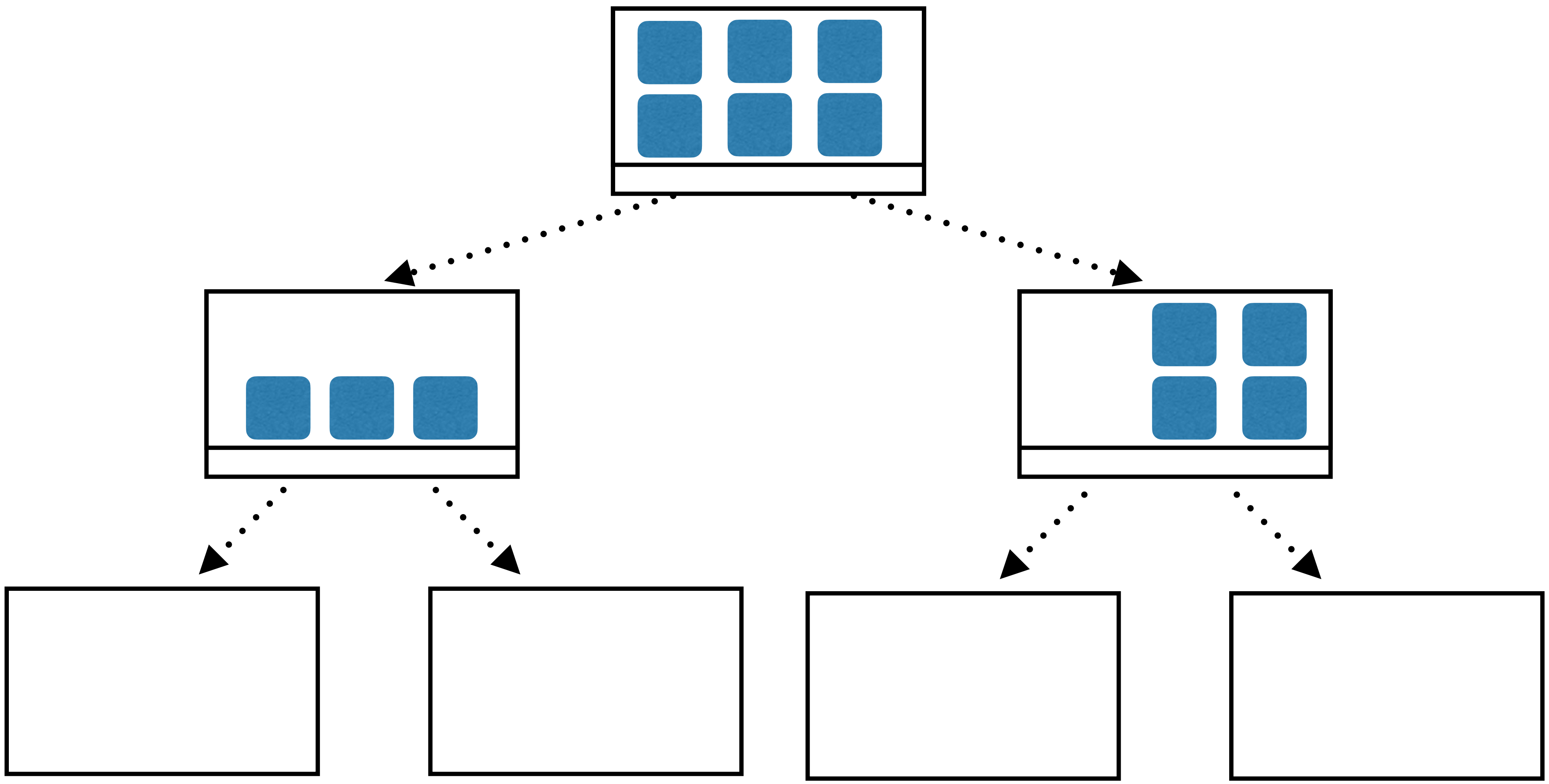
Bε-tree Inserts



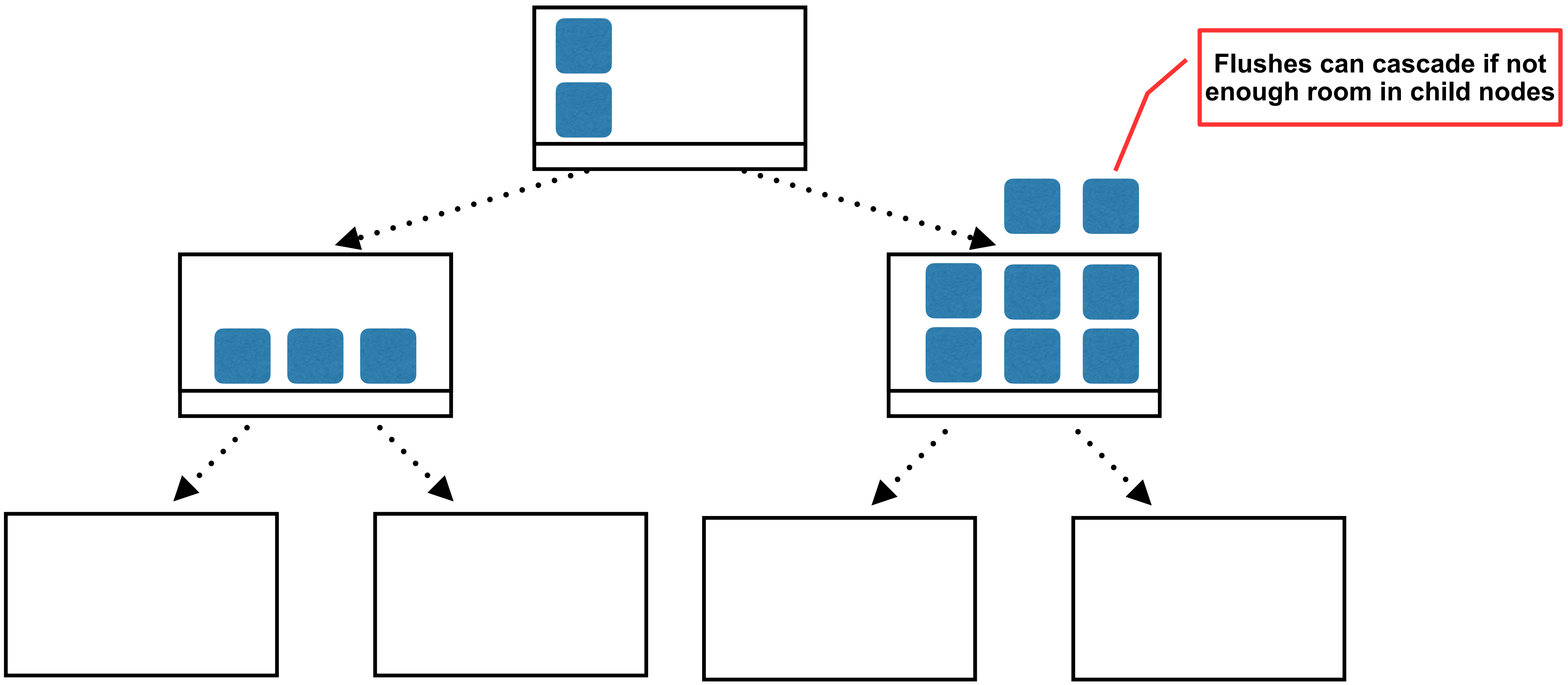
B$^\varepsilon$-tree Inserts



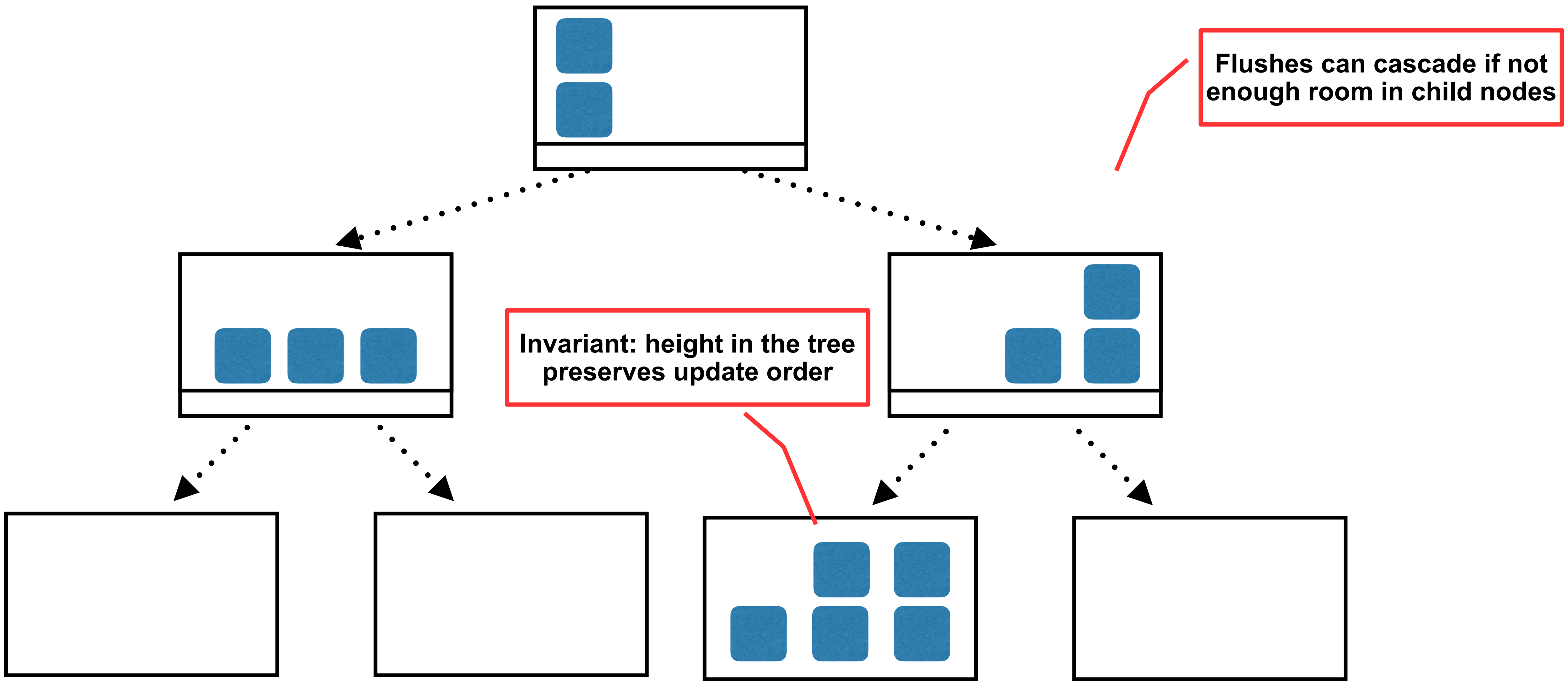
Bε-tree Inserts



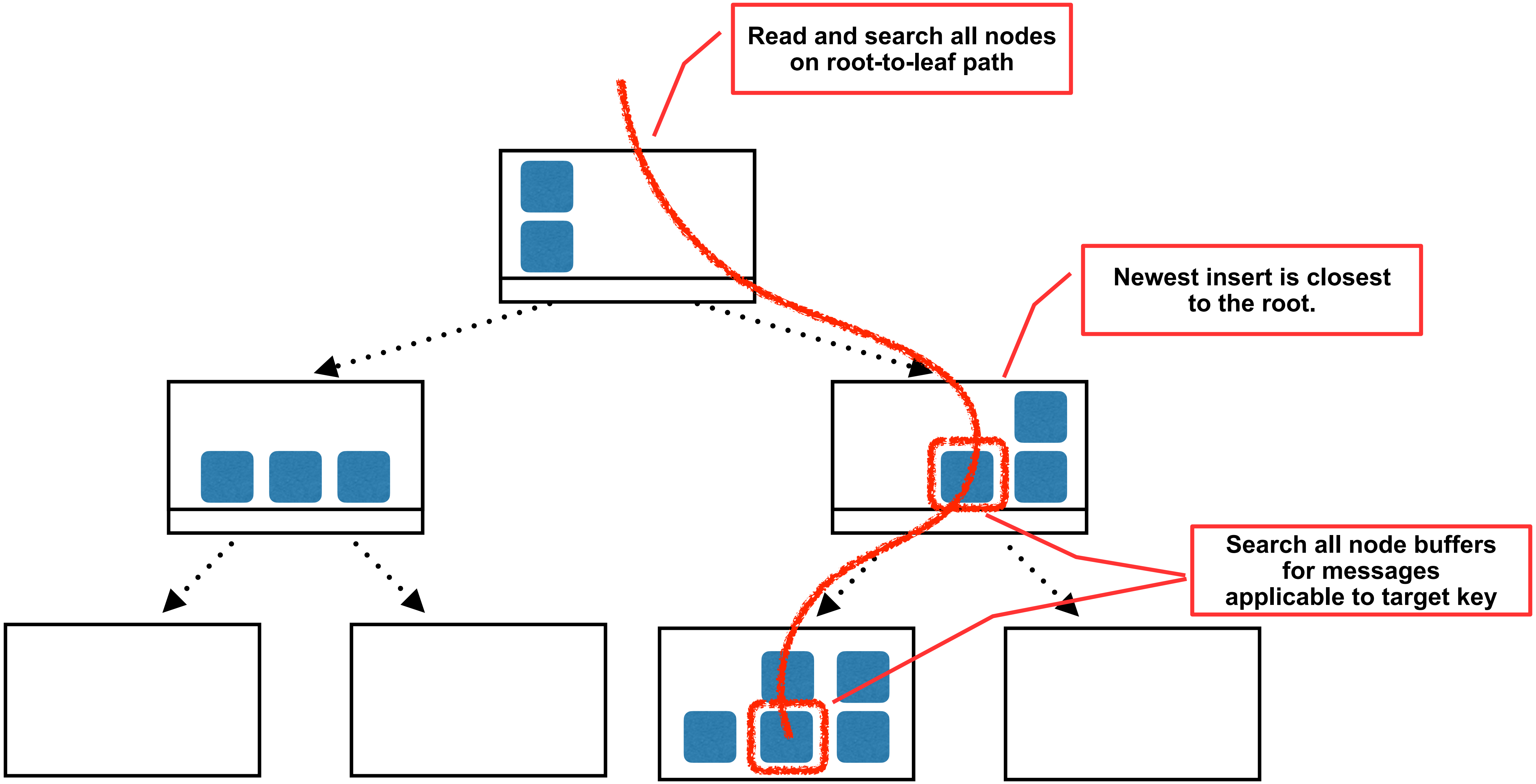
Bε-tree Inserts



Bε-tree Inserts



Bε-tree Searches



Updates

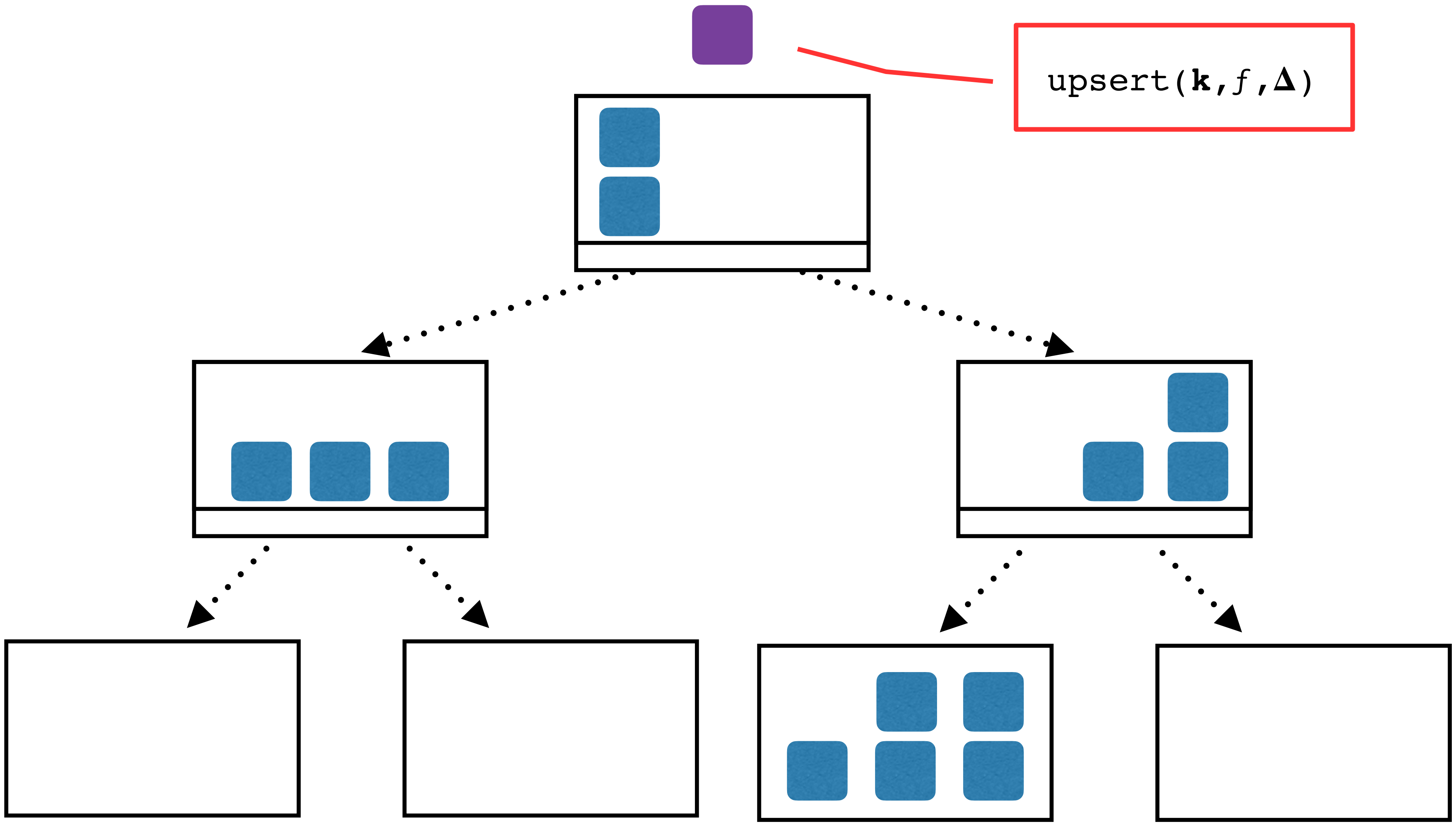
- In many systems, updating a value requires:
  read, modify, write — e.g., FFS writes, SSD blocks
- **Problem:** B^{ε}-tree inserts are faster than searches
  - fast updates are impossible if we must search first

```
upsert = update + insert
```

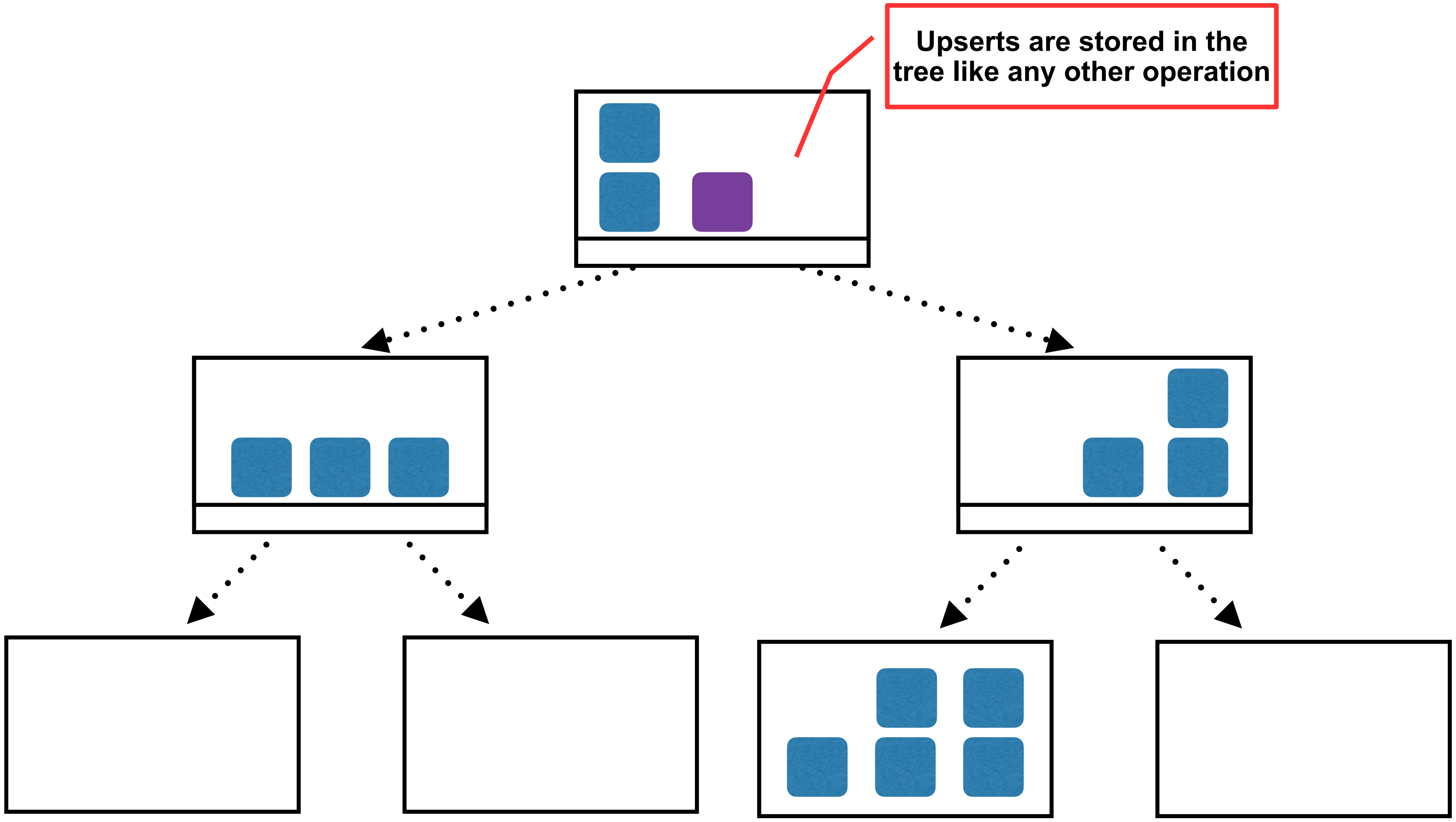
Upsert messages

- Each upsert message contains a:
  - Target key, **k**
  - Callback function, ***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

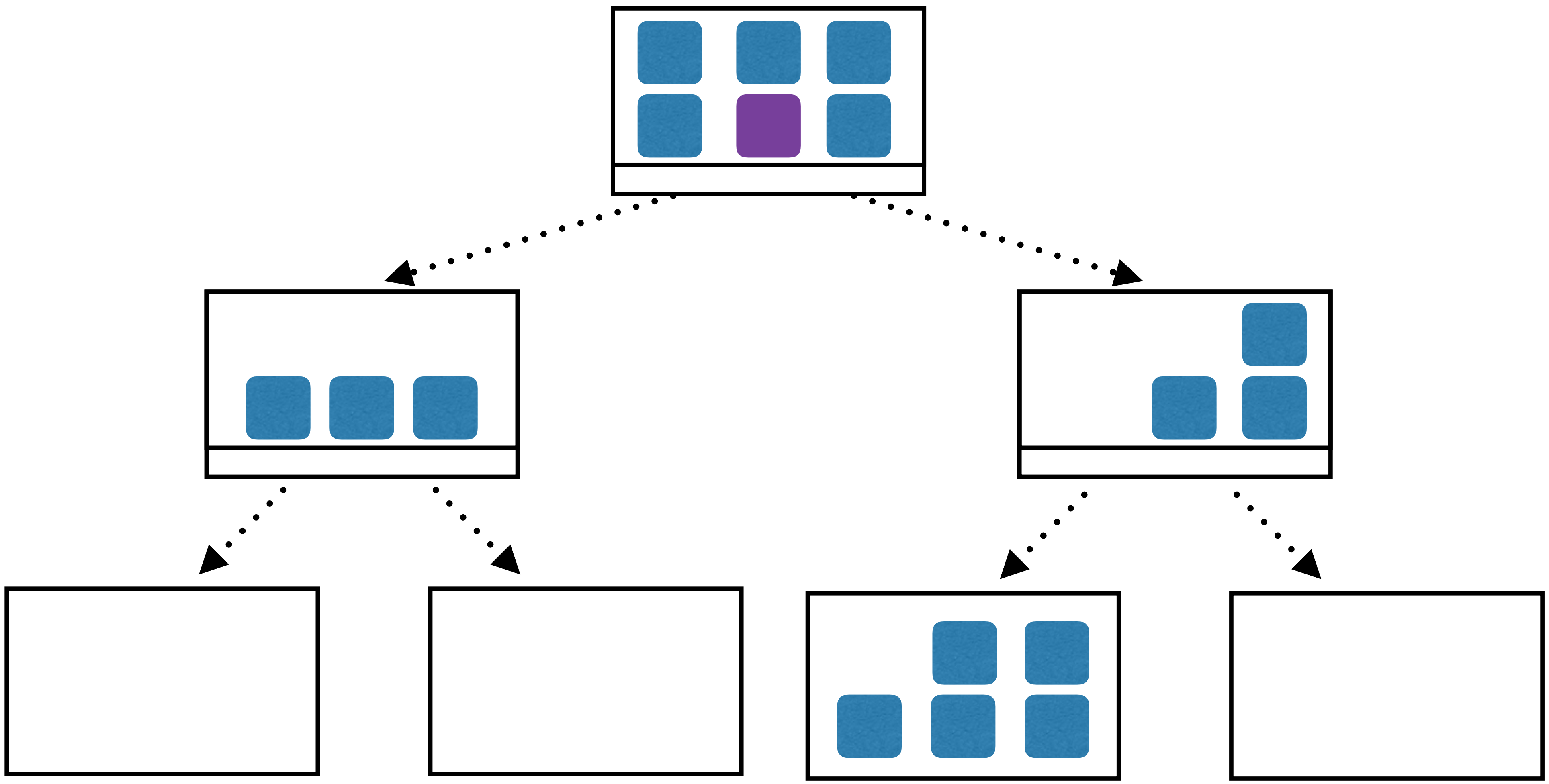
Bε-tree Upserts



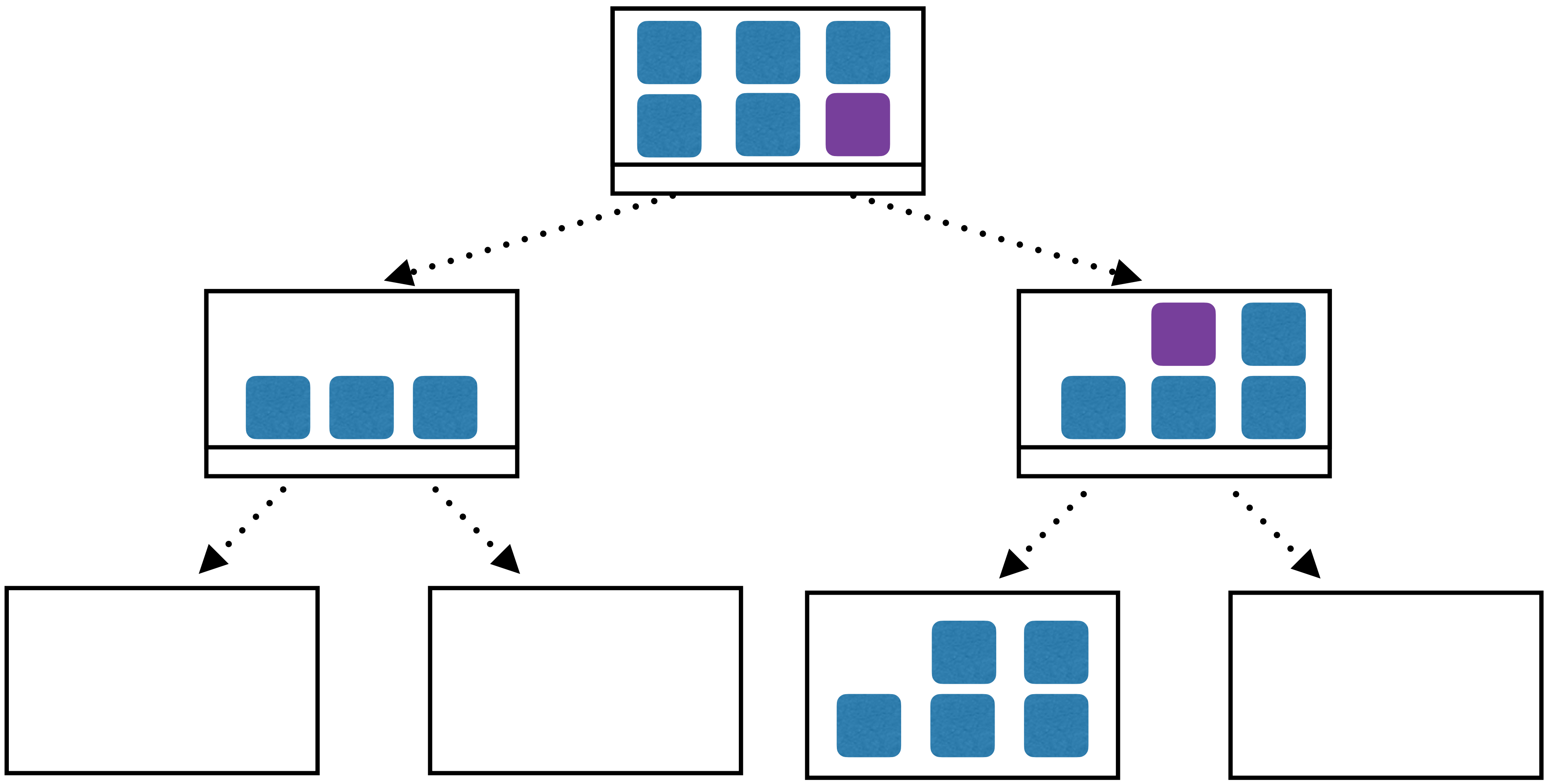
Bε-tree Upserts



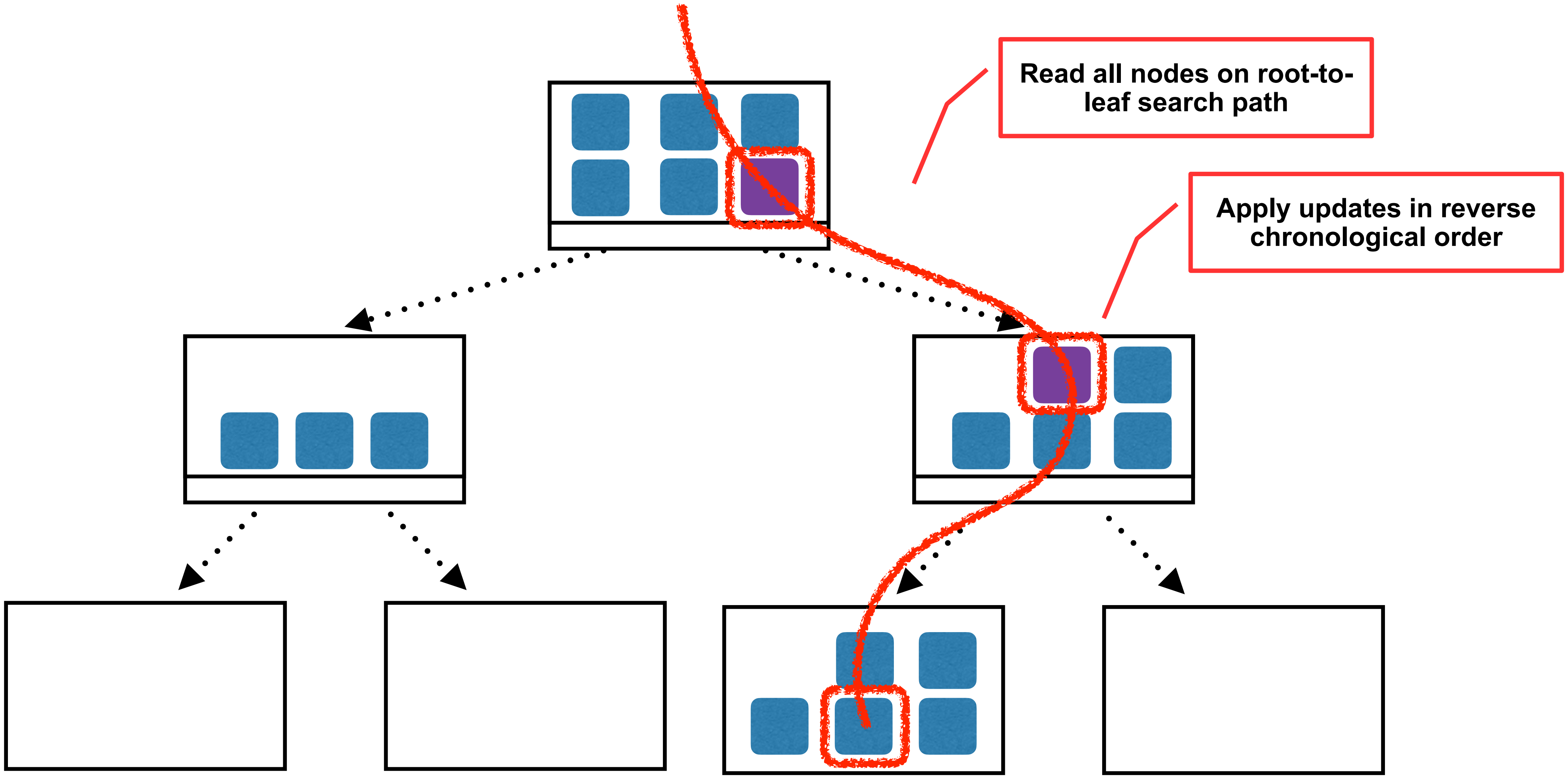
Bε-tree Upserts



Bε-tree Upserts



Searching with Upserts



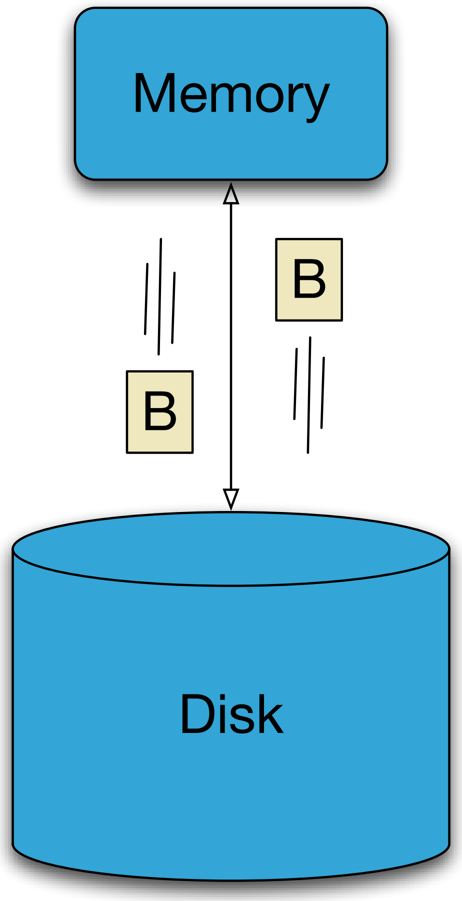
Upserts don't harm searches, but they let us perform **blind updates**.

Thought Question

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

Performance Model (Refesher)

- 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

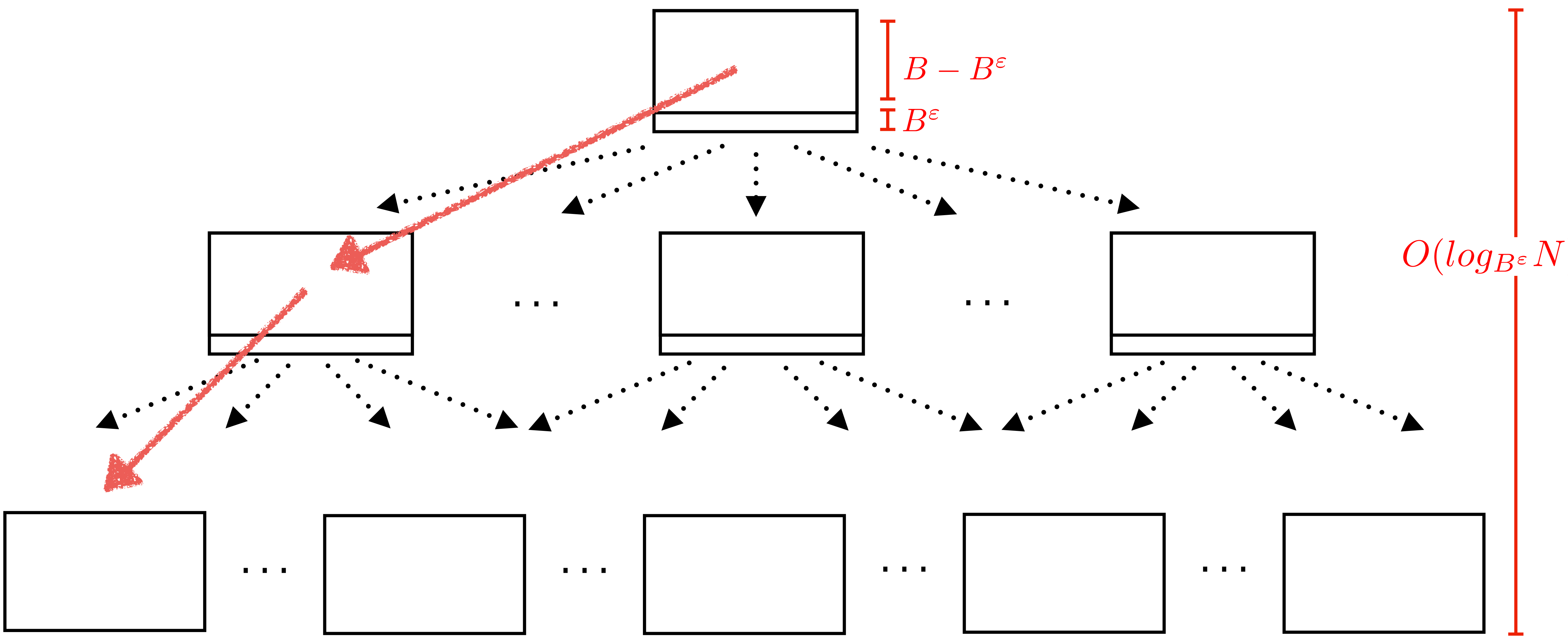


Performance = (# of I/Os)

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: $\log_b(\mathrm{M}\cdot\mathrm{N}) = \log_b \mathrm{M} + \log_b \mathrm{N}$

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

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

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/]

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

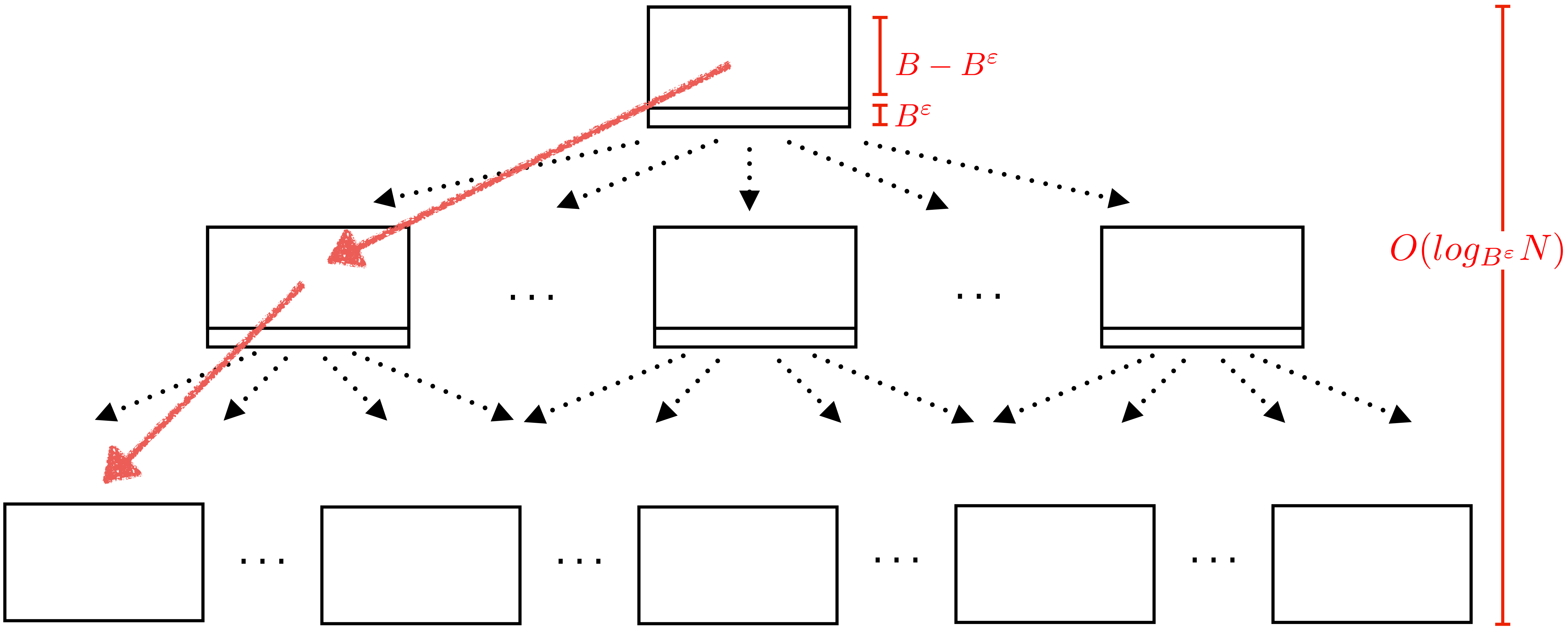
[https://www.khanacademy.org]

$$\log_{B^e} N \overset{\text{Change of base}}{=} \frac{\log_B N}{\log_B B^e} \overset{\text{Rule 6}}{=} \frac{\log_B N}{e}$$

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

Range Query: **?**

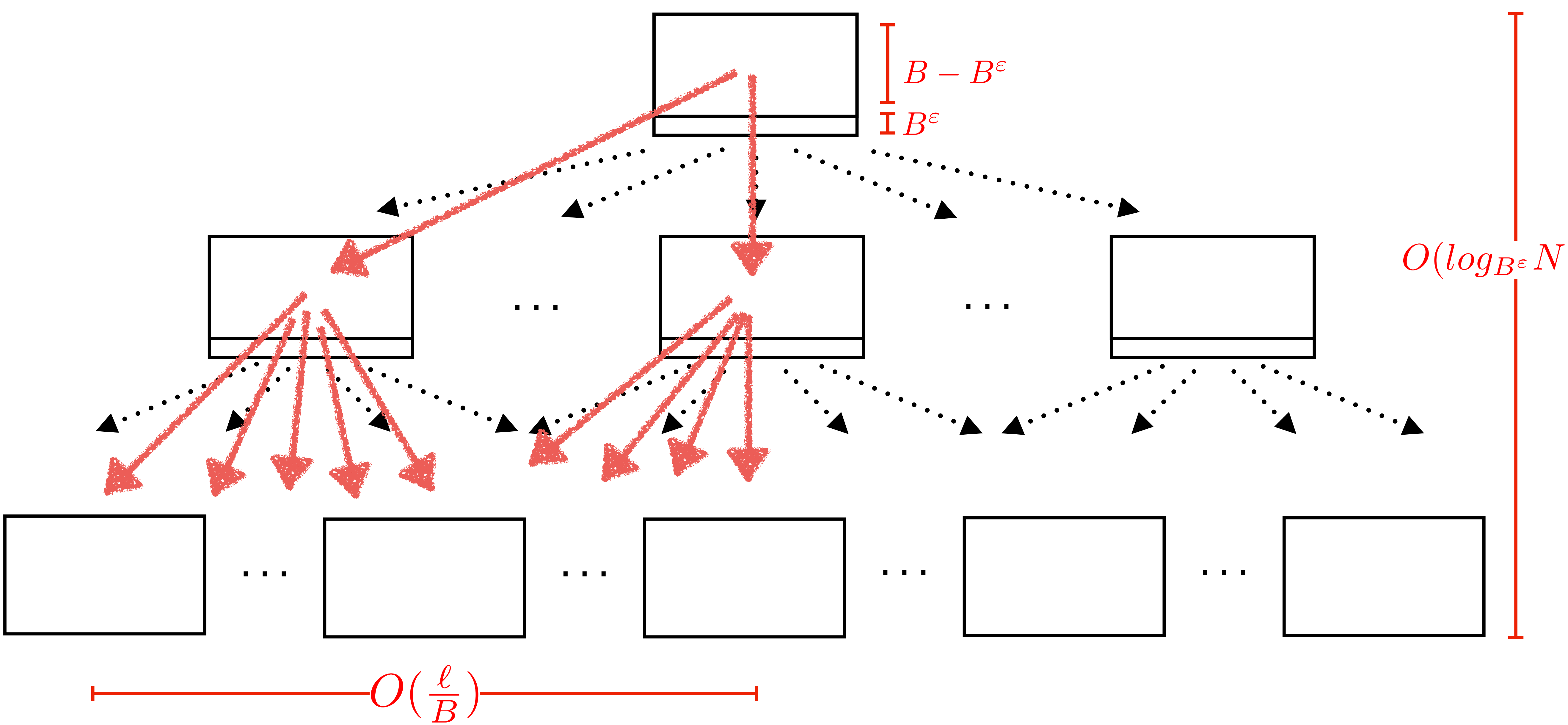
Insert/upsert:



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

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

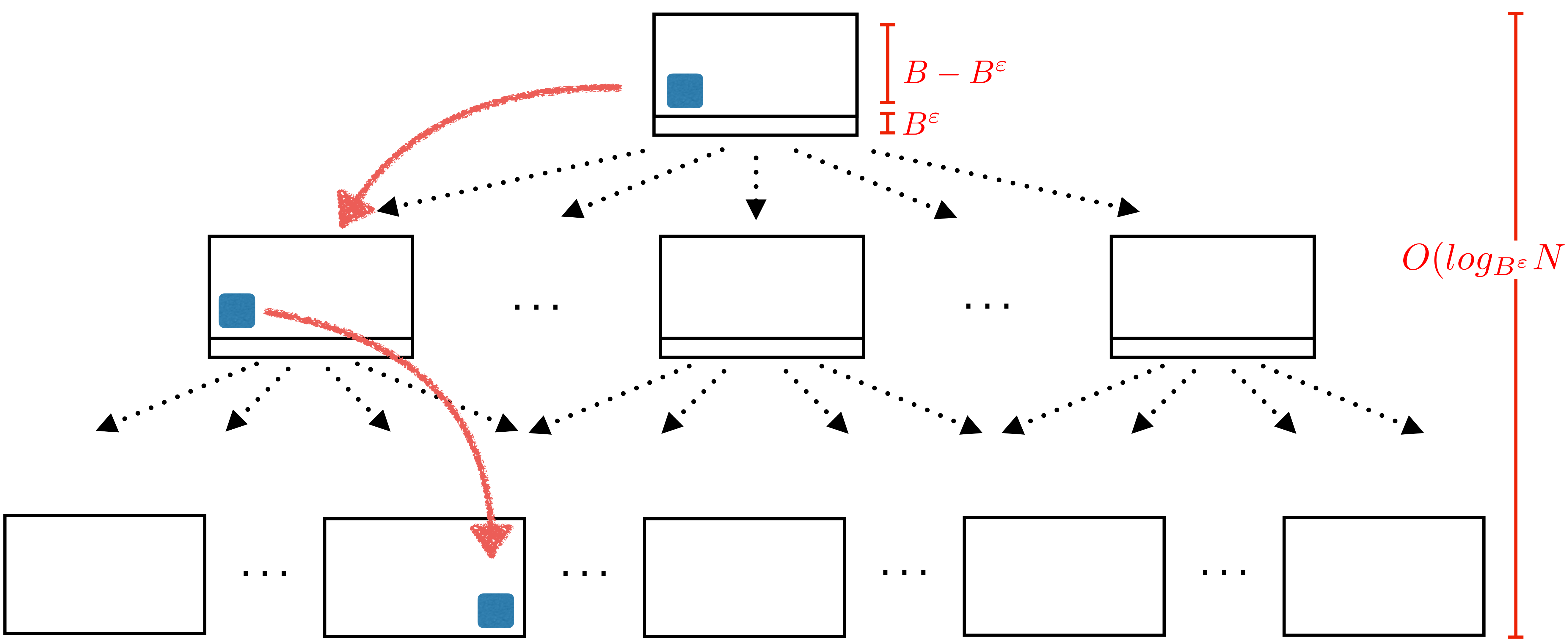
Insert/upsert: ?



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

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

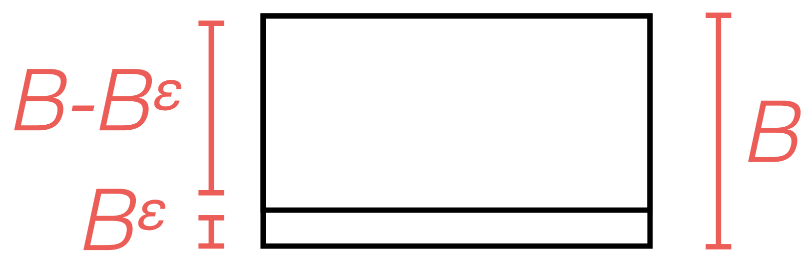
Insert/upsert: **?**



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?

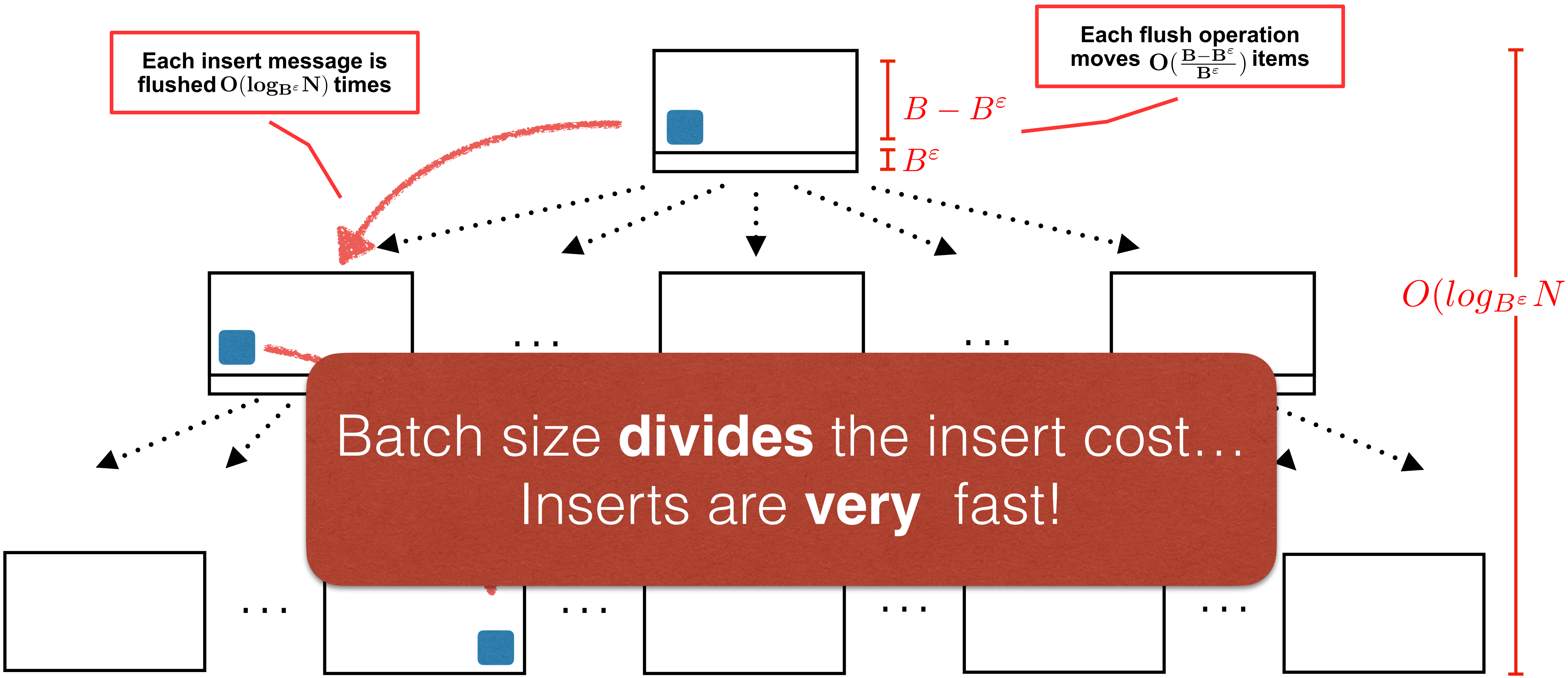
- Divide by the total cost by the number of messages

$$\frac{B-B^e}{B^e} = \frac{B^1}{B^e} - \frac{B^e}{B^e} = B^{1-e}-1$$

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

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

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



Recap/Big Picture

- Setup costs 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

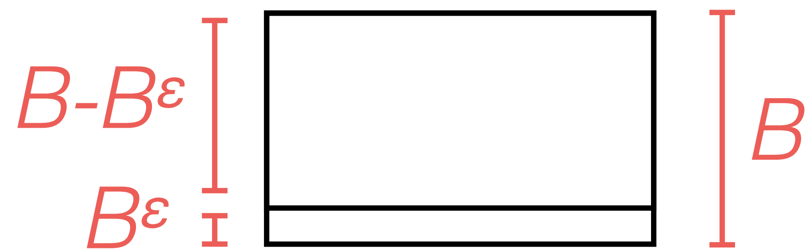
- How do we choose **ε**?

- 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

ε = 1 corresponds to a B-tree
ε = 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!