Announcements

- HW2 out - due Feb 6
- PACE ICE down Jan 23-25

Recap from previous class

There is also a cache-friendly version (B-skip list)

Binary trees

Skip lists

B/B+-trees

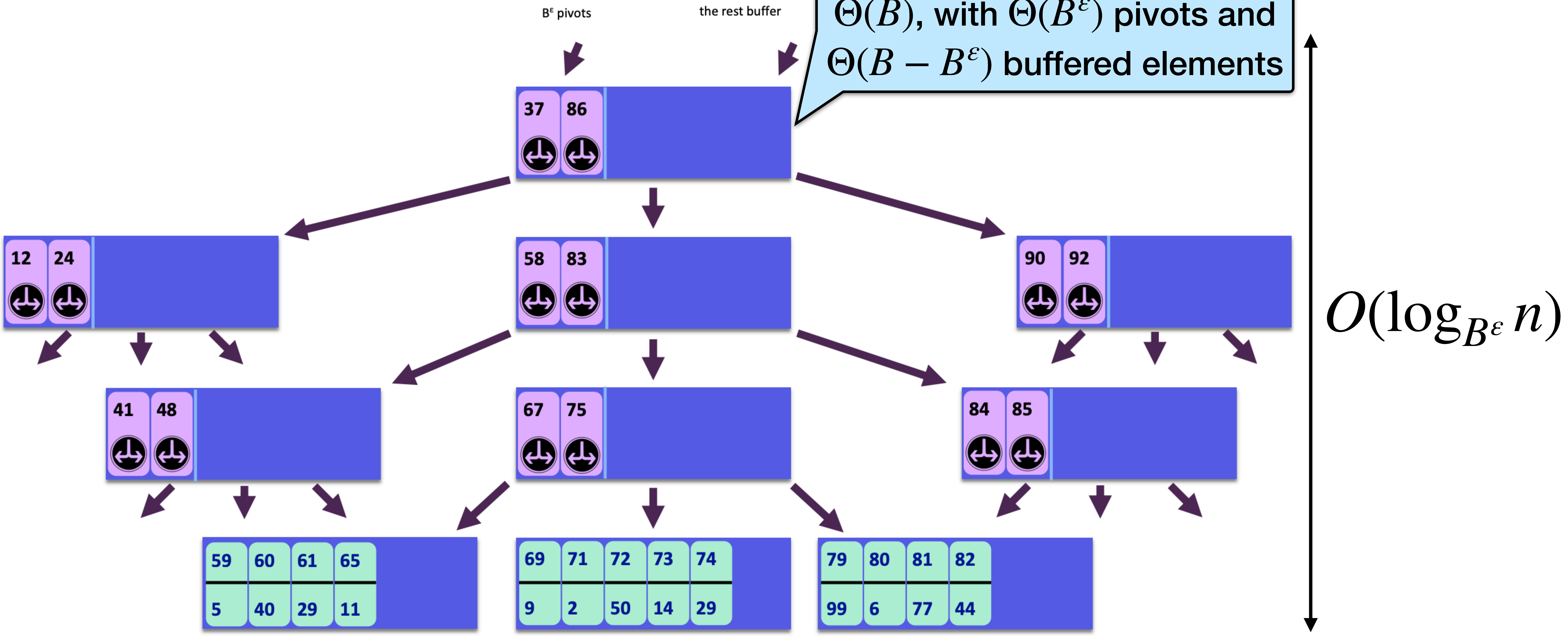
B^{ε} trees

Not I/O-optimal

I/O-optimal, but **requires knowledge of cache parameter B**

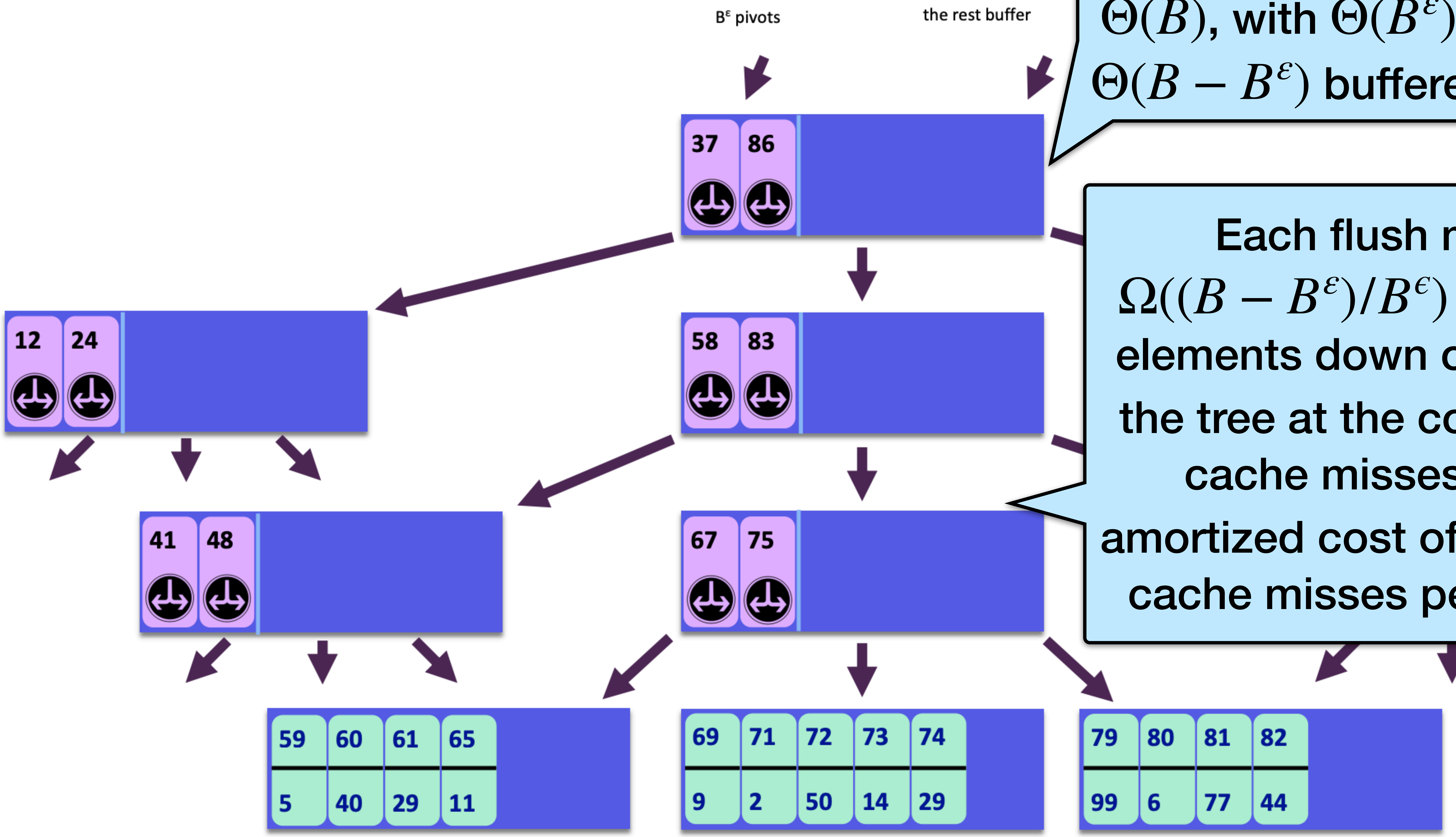
Proving B^{ε} tree bounds

Each internal node is size $\Theta(B)$, with $\Theta(B^{\varepsilon})$ pivots and $\Theta(B - B^{\varepsilon})$ buffered elements



Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

Proving B$^{\varepsilon}$ tree bounds

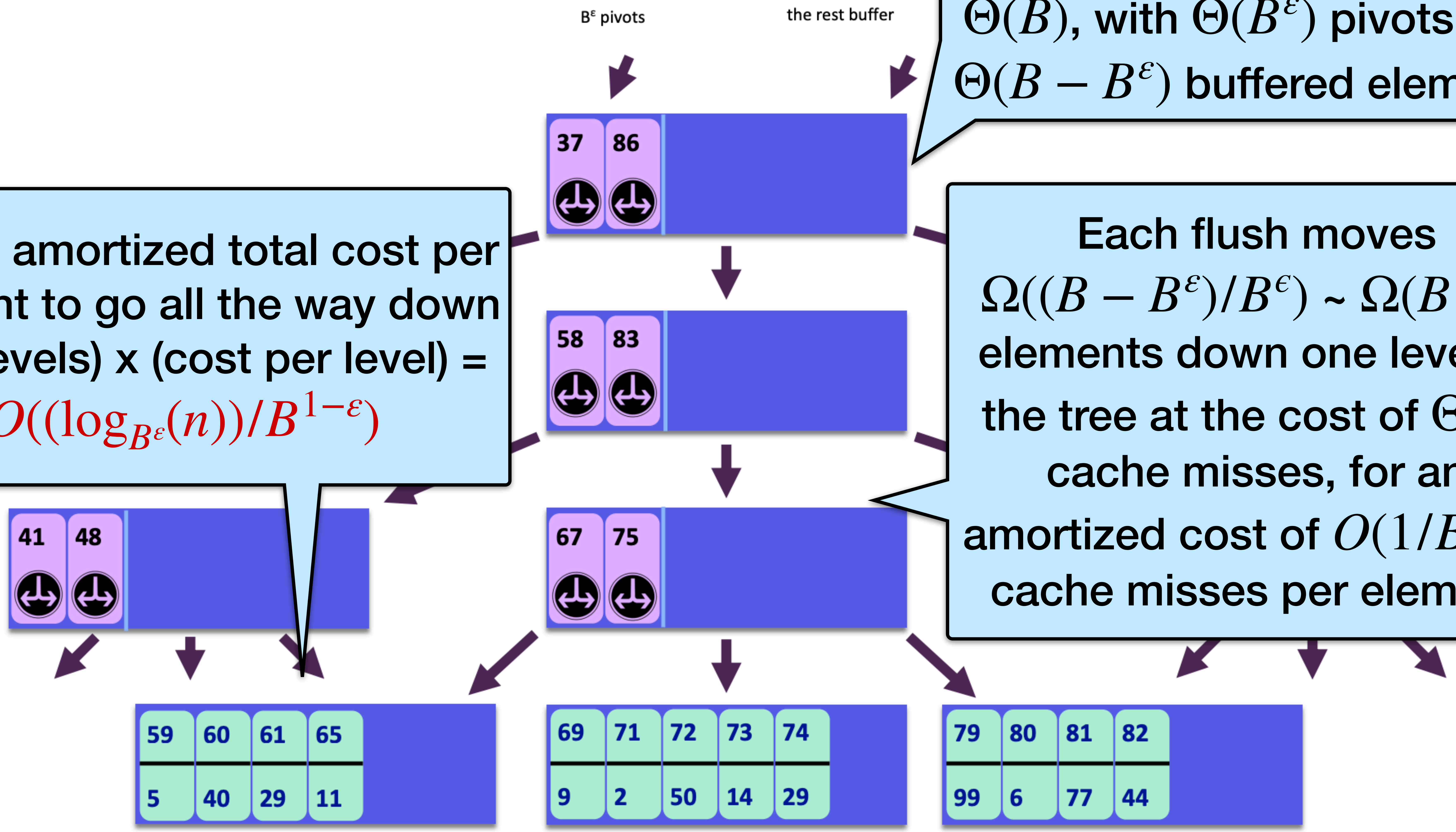


Each internal node is size $\Theta(B)$, with $\Theta(B^{\varepsilon})$ pivots and $\Theta(B - B^{\varepsilon})$ buffered elements

Each flush moves $\Omega((B - B^{\varepsilon})/B^{\epsilon}) \sim \Omega(B^{1-\varepsilon})$ elements down one level of the tree at the cost of $\Theta(1)$ cache misses, for an amortized cost of $O(1/B^{1-\varepsilon})$ cache misses per element

$O(\log_{B^{\varepsilon}} n)$

Proving B$^{\varepsilon}$ tree bounds



Each internal node is size $\Theta(B)$, with $\Theta(B^{\varepsilon})$ pivots and $\Theta(B - B^{\varepsilon})$ buffered elements

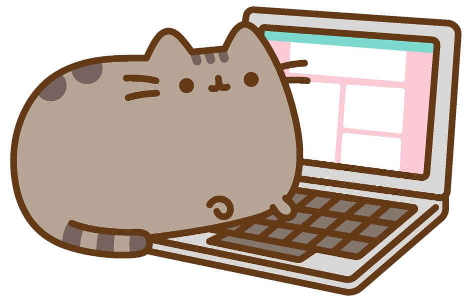
Each flush moves $\Omega((B - B^{\varepsilon})/B^{\epsilon})$ ~ $\Omega(B^{1-\varepsilon})$ elements down one level of the tree at the cost of $\Theta(1)$ cache misses, for an amortized cost of $O(1/B^{1-\varepsilon})$ cache misses per element

So the amortized total cost per element to go all the way down is (# levels) x (cost per level) = $O((\log_{B^{\varepsilon}}(n))/B^{1-\varepsilon})$

$O(\log_{B^{\varepsilon}} n)$

Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture10.pdf

CSE 6230:
HPC Tools and Applications

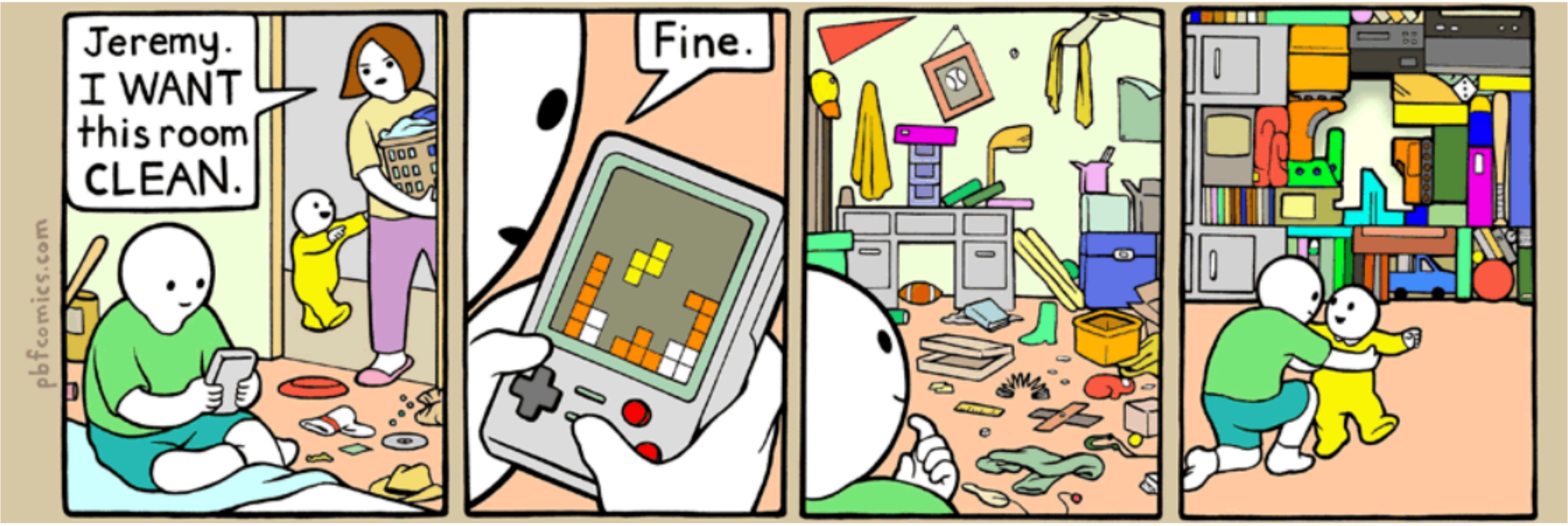

+


Lecture 5: I/O-efficient Data Structures (Part 2)

Helen Xu
hxu615@gatech.edu

Georgia Tech College of Computing
School of Computational Science and Engineering

Recall: Logging vs Indexing

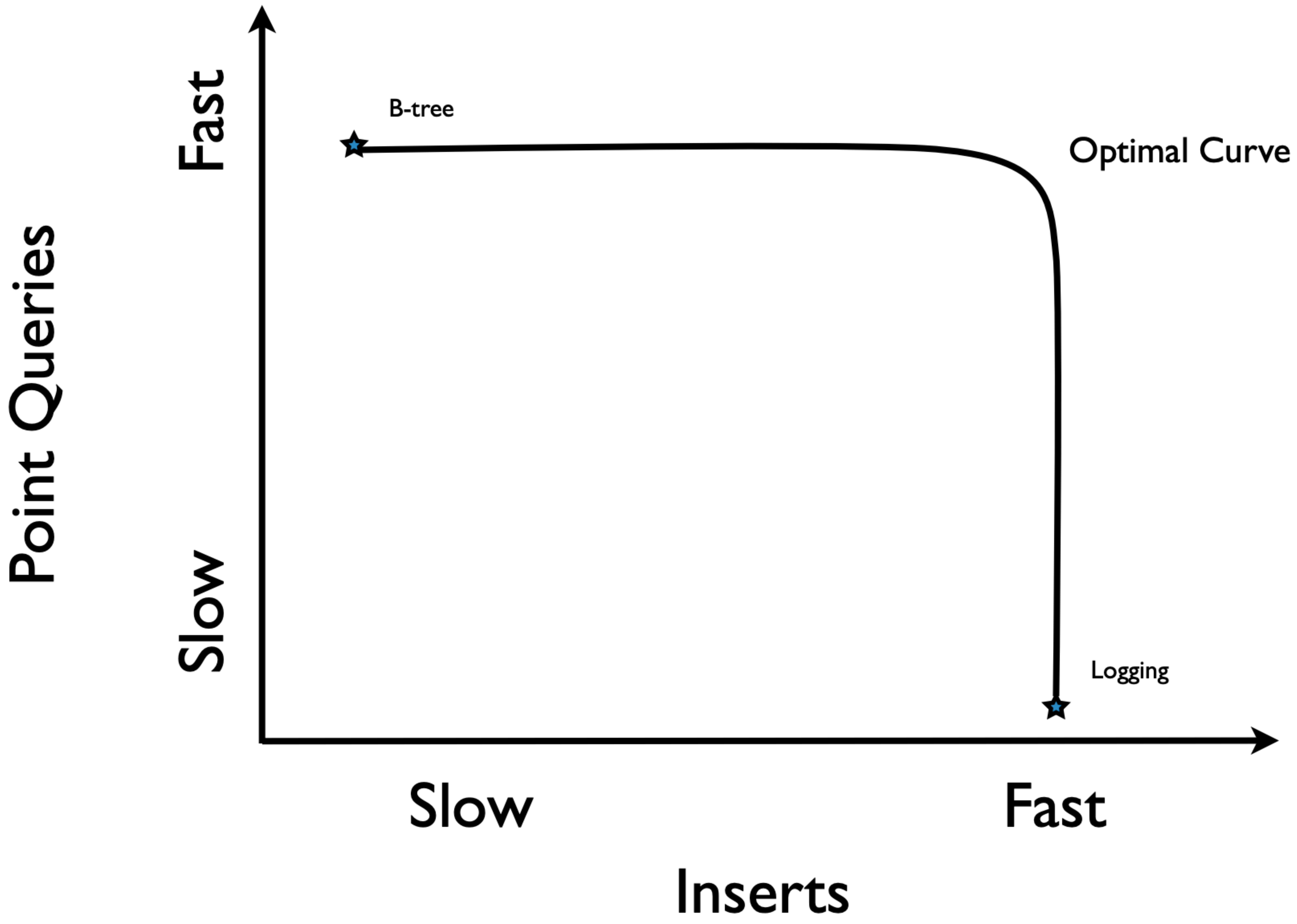


Logging
e.g., with an array

Indexing
e.g., B-trees

Slide from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture11.pdf

Optimal Search-Insert Tradeoff [Brodal, Fagerberg 03]



Slides from: https://www3.cs.stonybrook.edu/~bender/talks/2012-Bender-Dagstuhl-write-optimized-talk.pdf

Goal: Optimal Data Structures

	insert	point query
Optimal tradeoff (function of ε=0...1)	$O\left(\frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}}\right)$	$O\left(\log_{1+B^\varepsilon} N\right)$

B-trees are optimal for search but **not for update**.

Goal: Data structure with **inserts that beat B-tree inserts** without sacrificing on queries.

This is the promise of **write optimization**

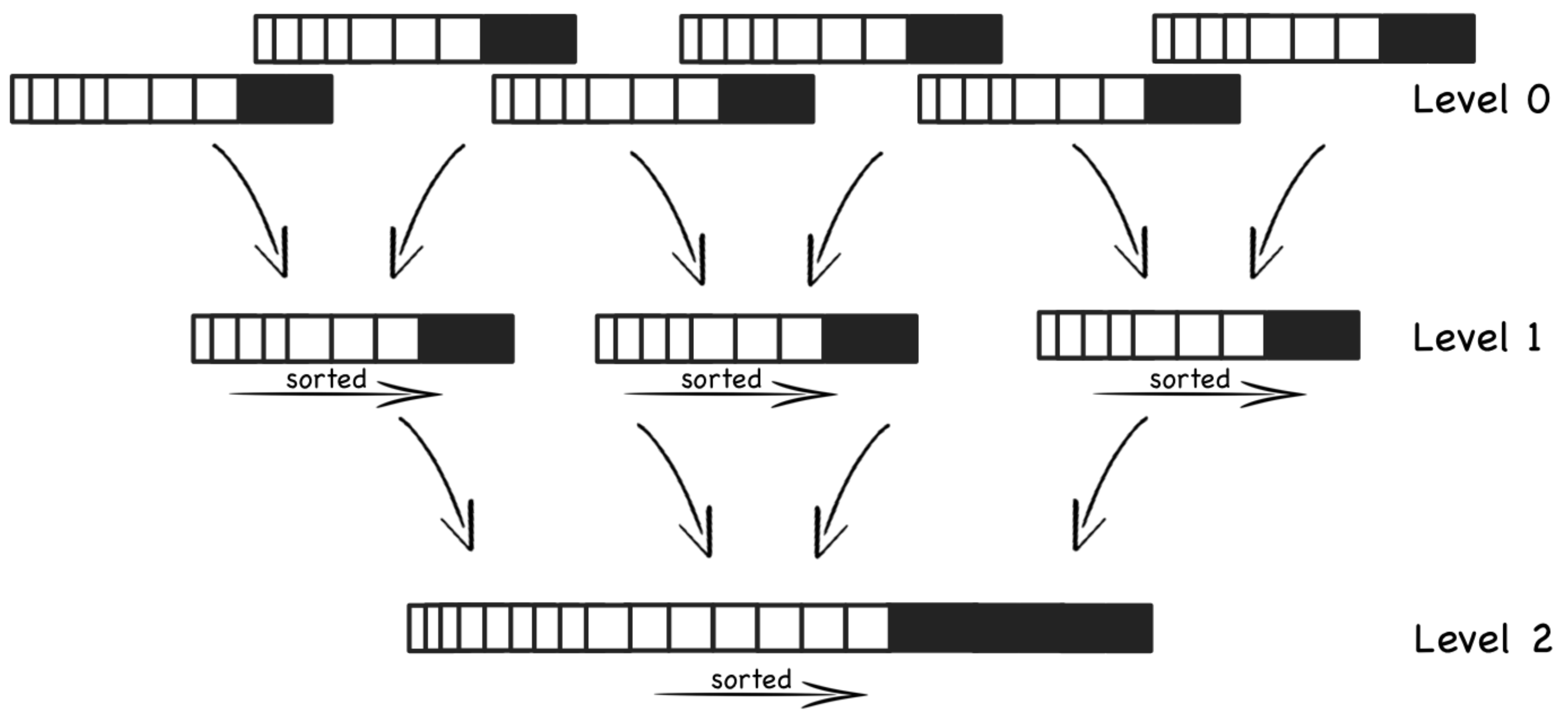
Slide from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture11.pdf

Log-Structured Merge (LSM) Trees

Not optimal straight out of the box, but we will show how to get them there.

Applications of LSM trees

- Proposed by O'Neil, Cheng, and Gawlick in 1996
- Uses **write-optimized techniques** to significantly speed up inserts.
- Have become popular over the past ~10-15 years or so
- Accumulo, Bigtable, bLSM, Cassandra, HBase, Hypertable, LevelDB are LSM trees (or based on LSM trees)



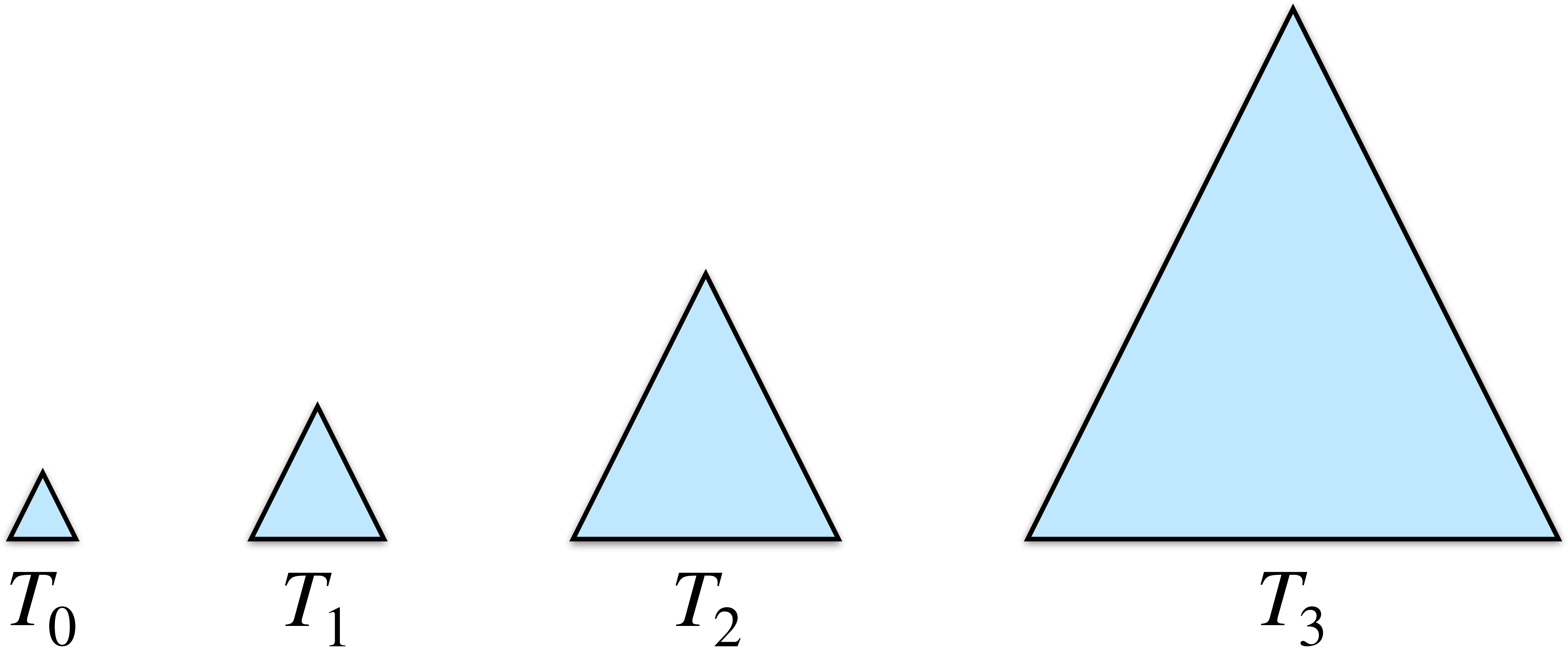
Compaction continues creating fewer, larger and larger files

Log-Structured Merge Trees

An LSM tree is a **cascade of B-trees**.

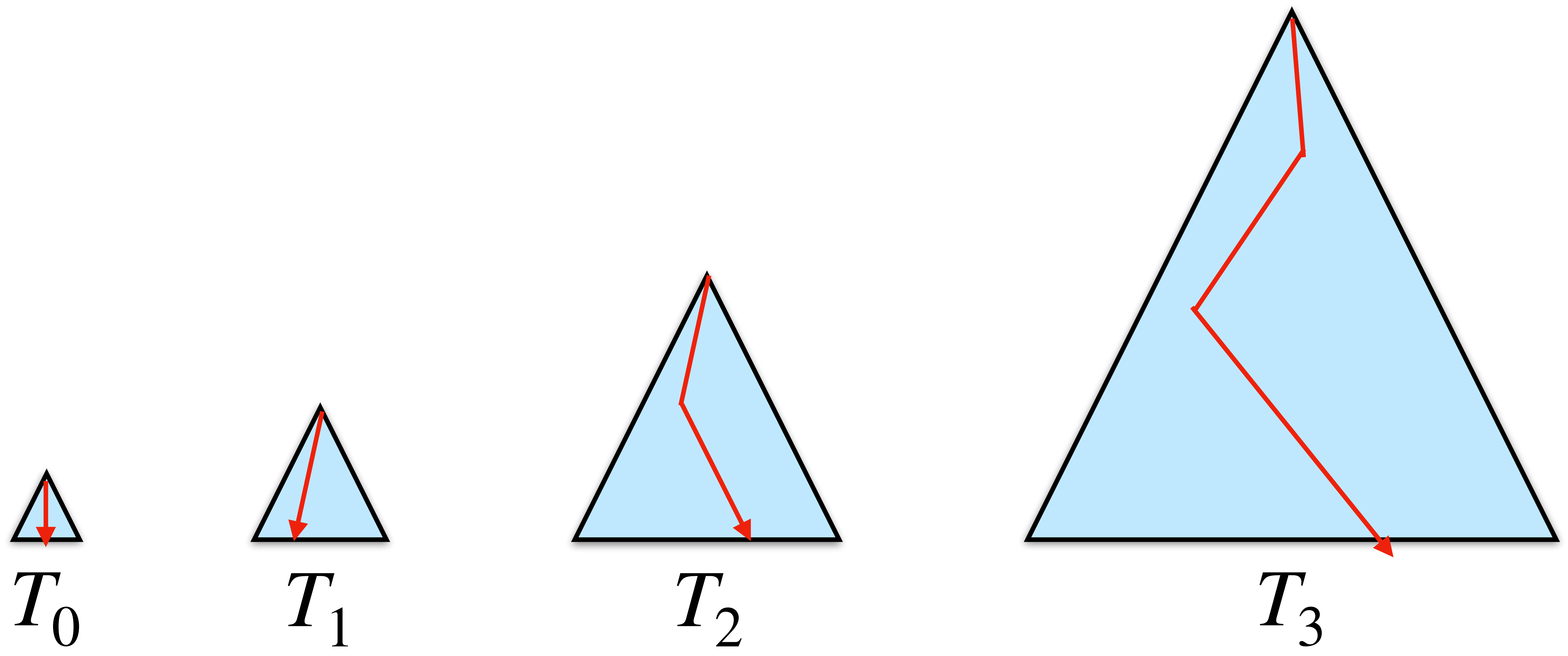
Each tree T_j has a **target size** $|T_j|$.

The target sizes are **exponentially increasing**: typically, $|T_{j+1}| = 10|T_j|$



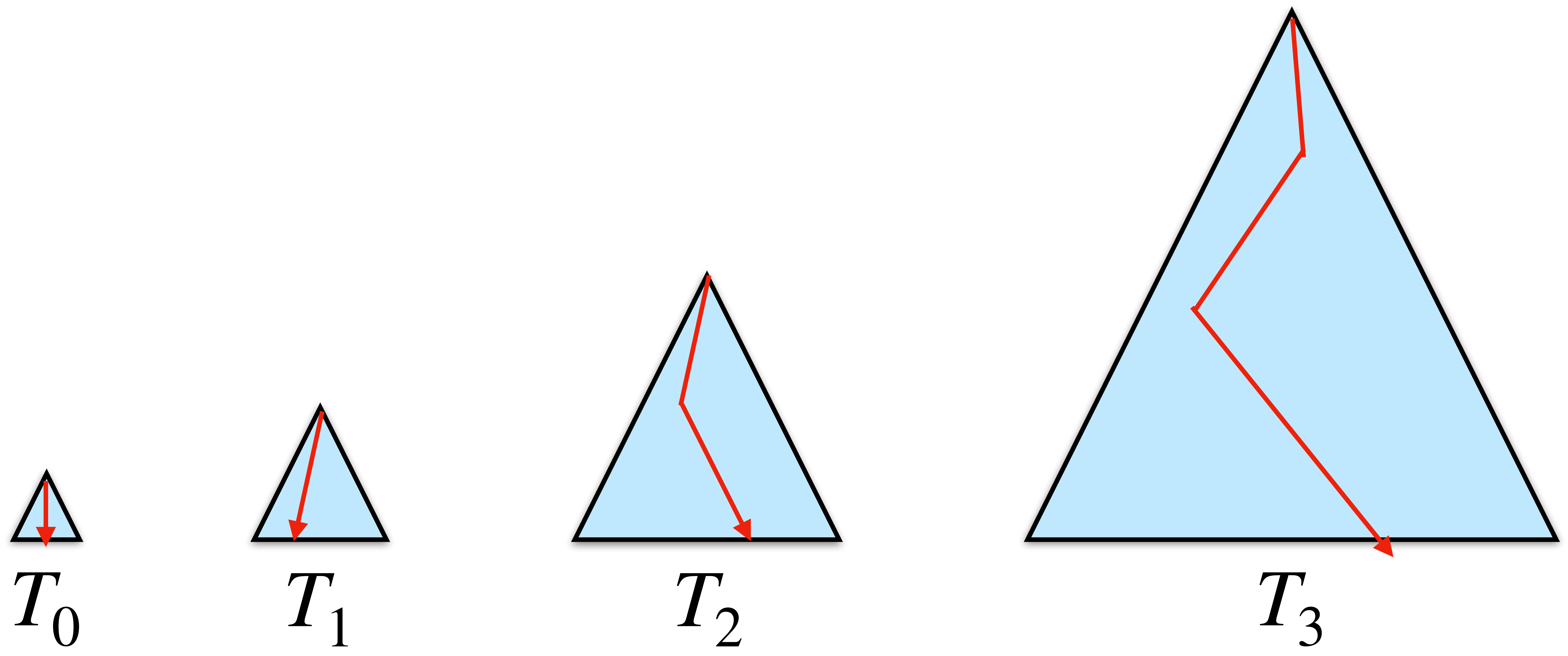
LSM Tree Operations

Point queries:



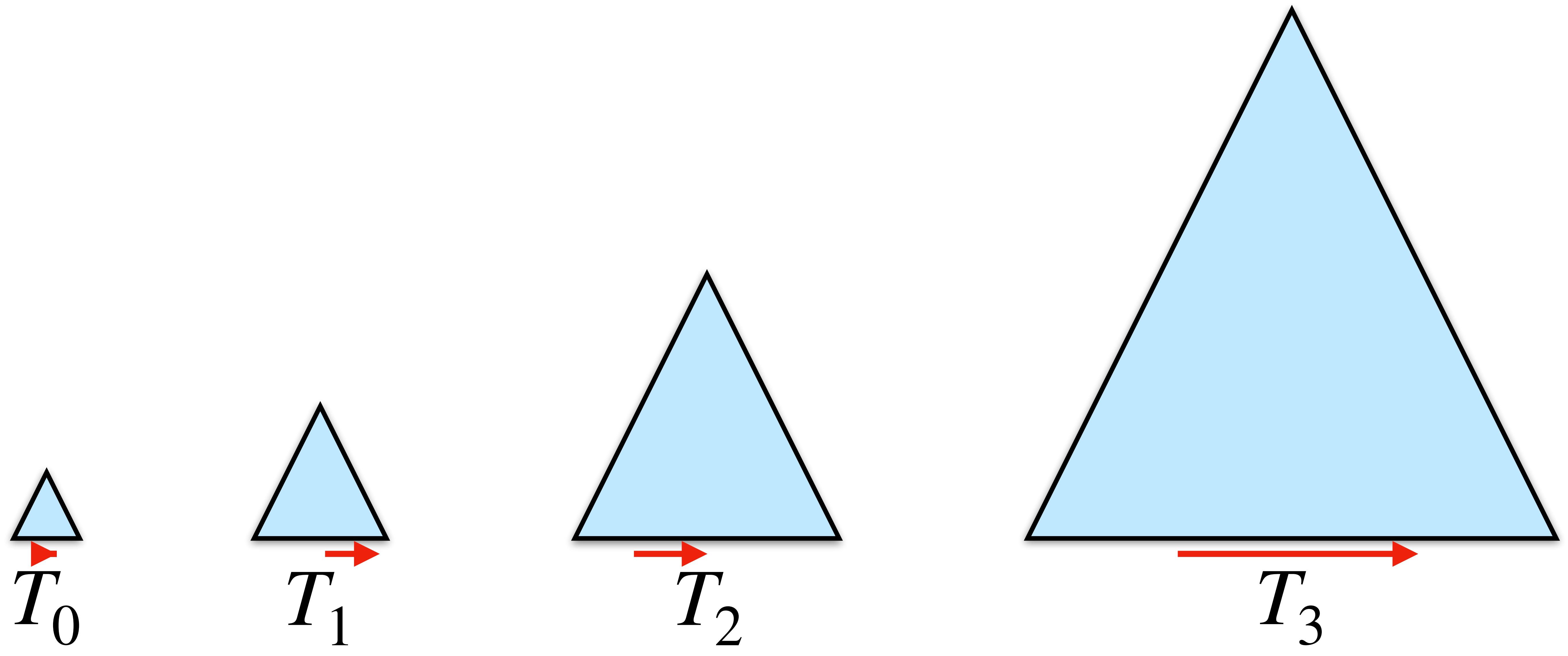
LSM Tree Operations

Point queries:



Range queries:

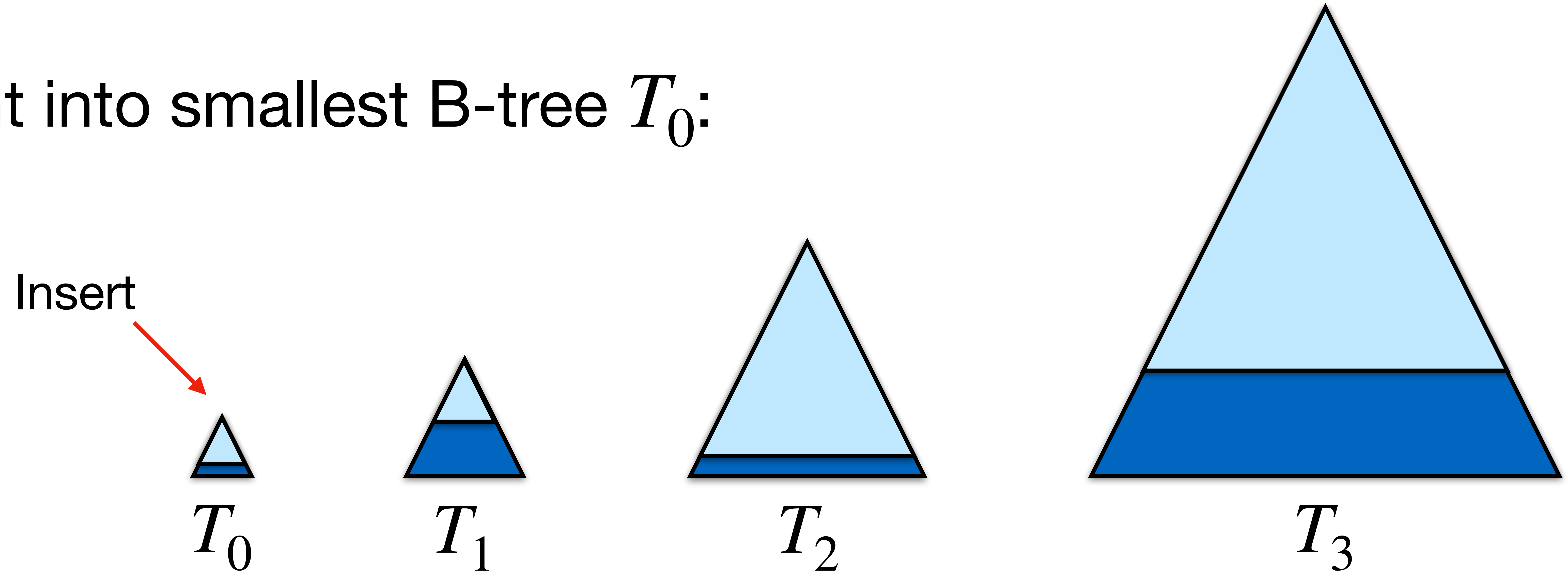
e.g., get all elements between x and y



LSM Tree Operations

Insertions

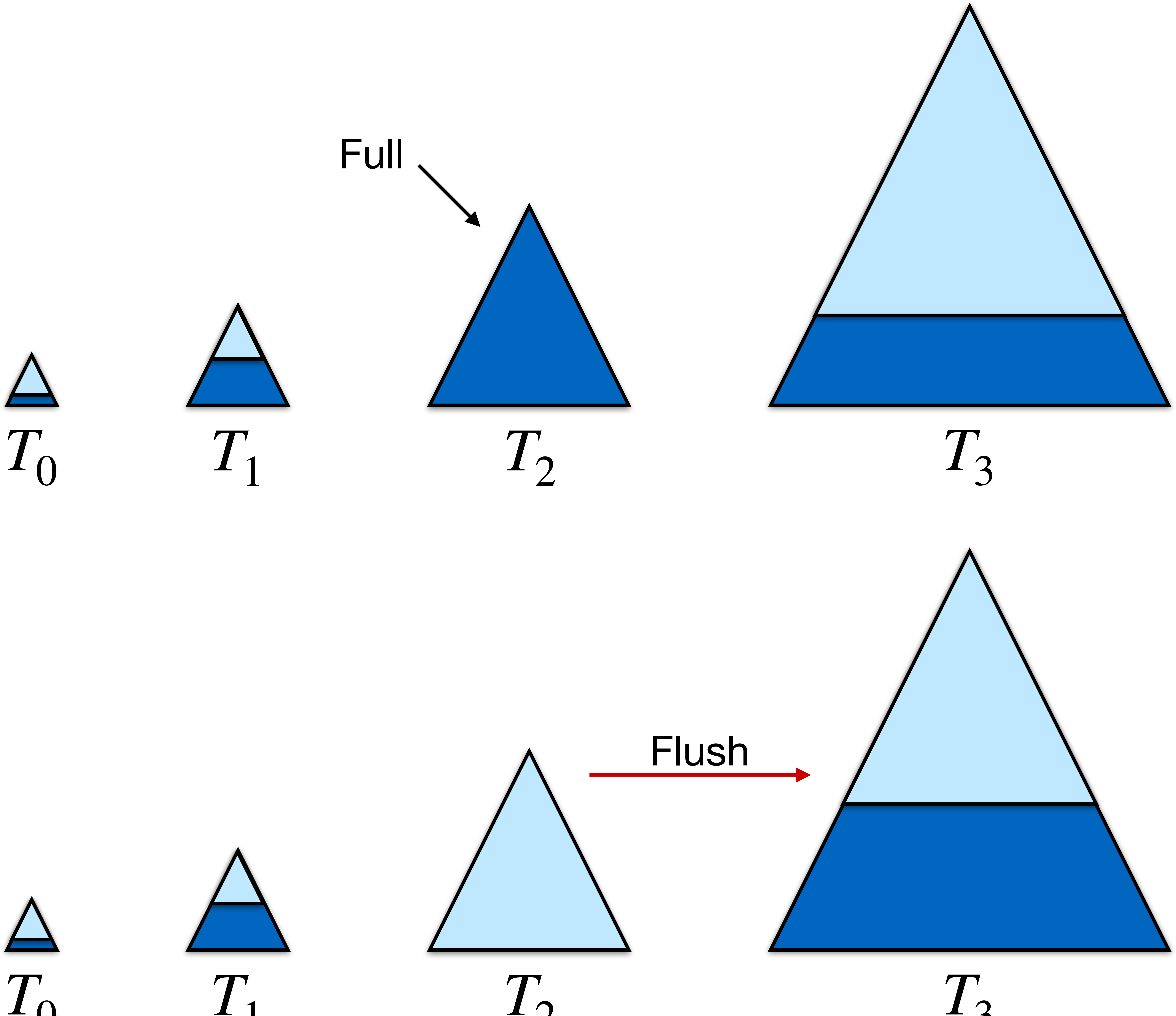
Always insert element into smallest B-tree T_0:



LSM Tree Operations

Insertions

When a B-tree T_j fills up, flush into T_{j+1}:

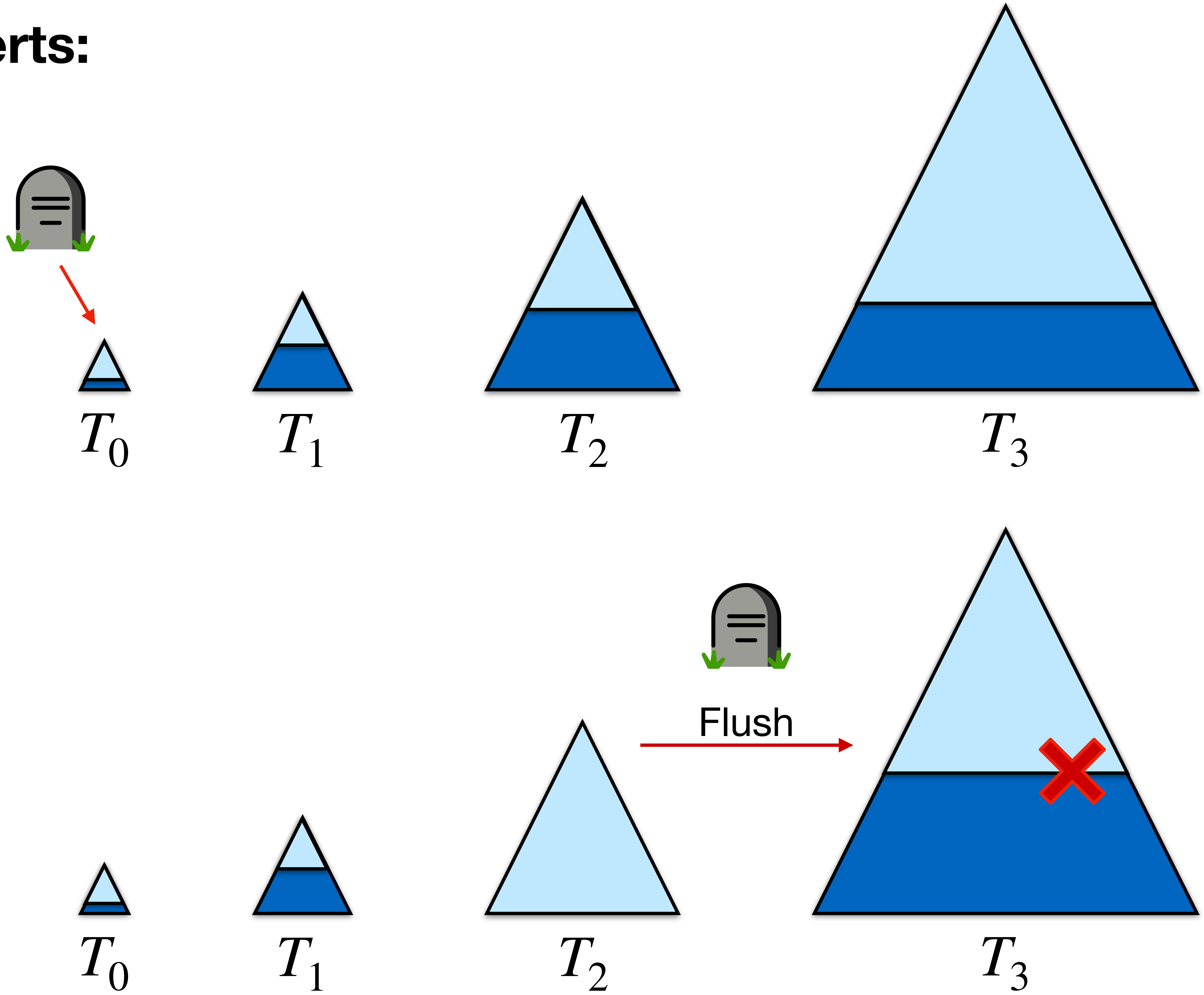


LSM Tree Operations

Deletes are like inserts:

Instead of deleting an element directly, insert **tombstones**.

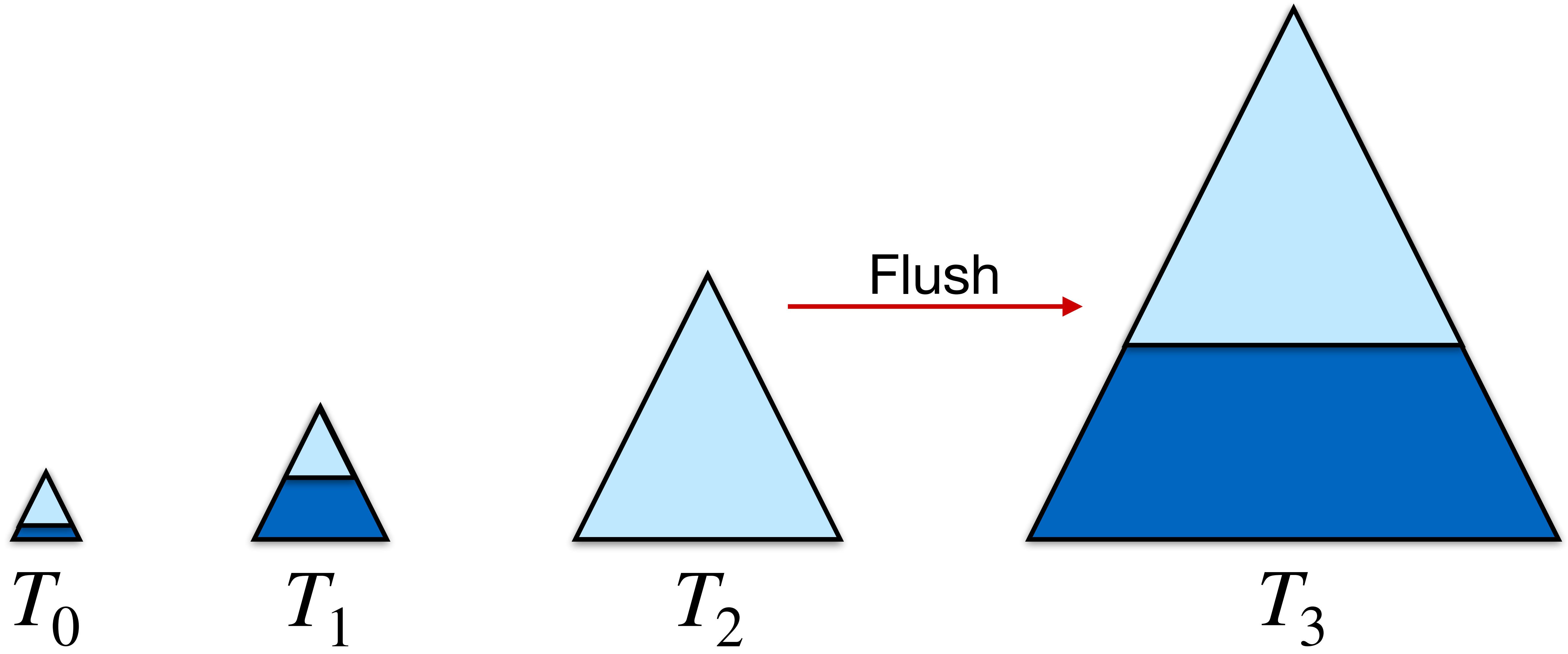
A tombstone knocks out a "real" element when it lands in the **same tree**.



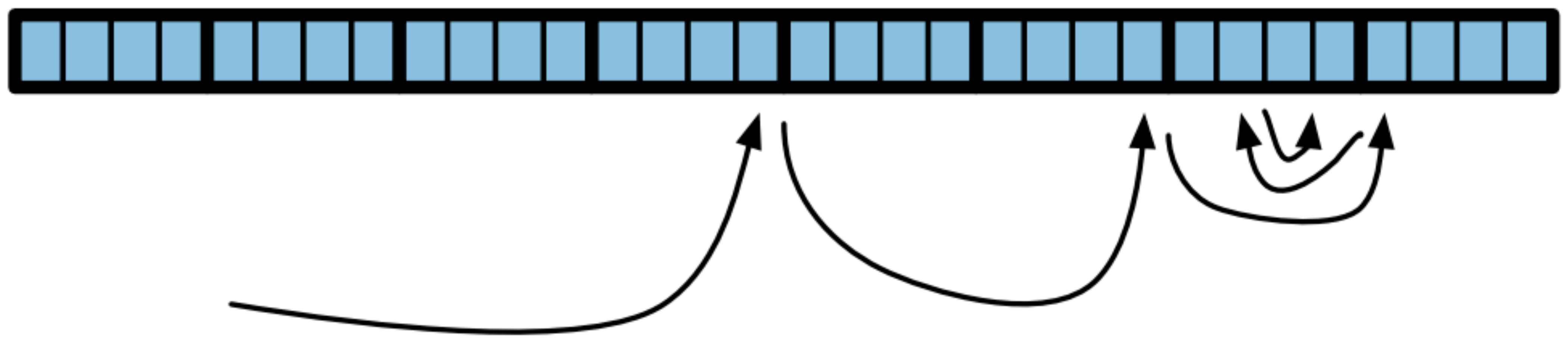
Static-to-Dynamic Transformation

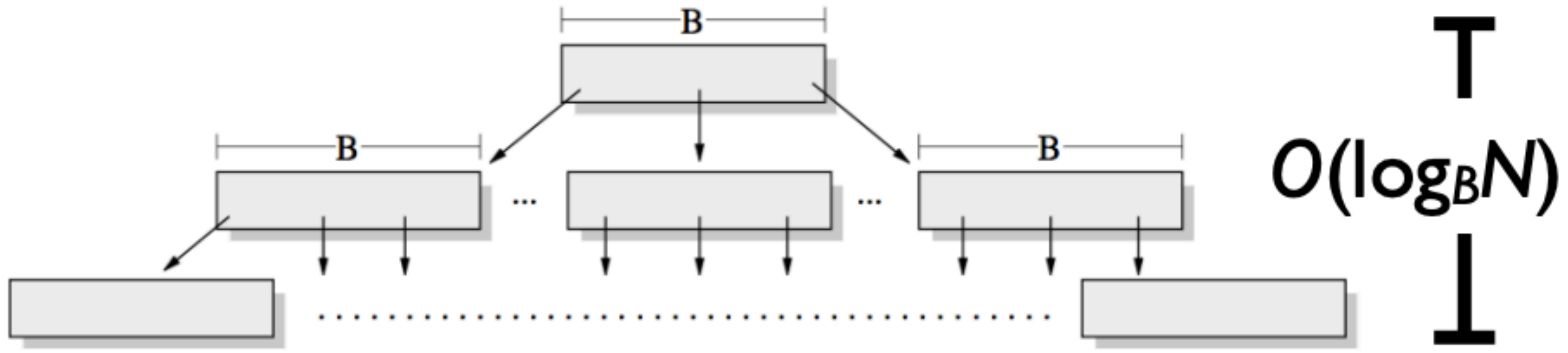
An LSM tree is an example of a **"static-to-dynamic" transformation** [Bentley and Saxe '80].

- An LSM tree can be built out of **static B-trees**.
- When T_j flushes into T_{j+1}, T_{j+1} is **rebuilt from scratch**.



Recall: Searching an Array vs. B-tree





$$O\left(\log_2 \frac{N}{B}\right) \approx O(\log_2 N)$$

$$O(\log_B N) = O\left(\frac{\log_2 N}{\log_2 B}\right)$$

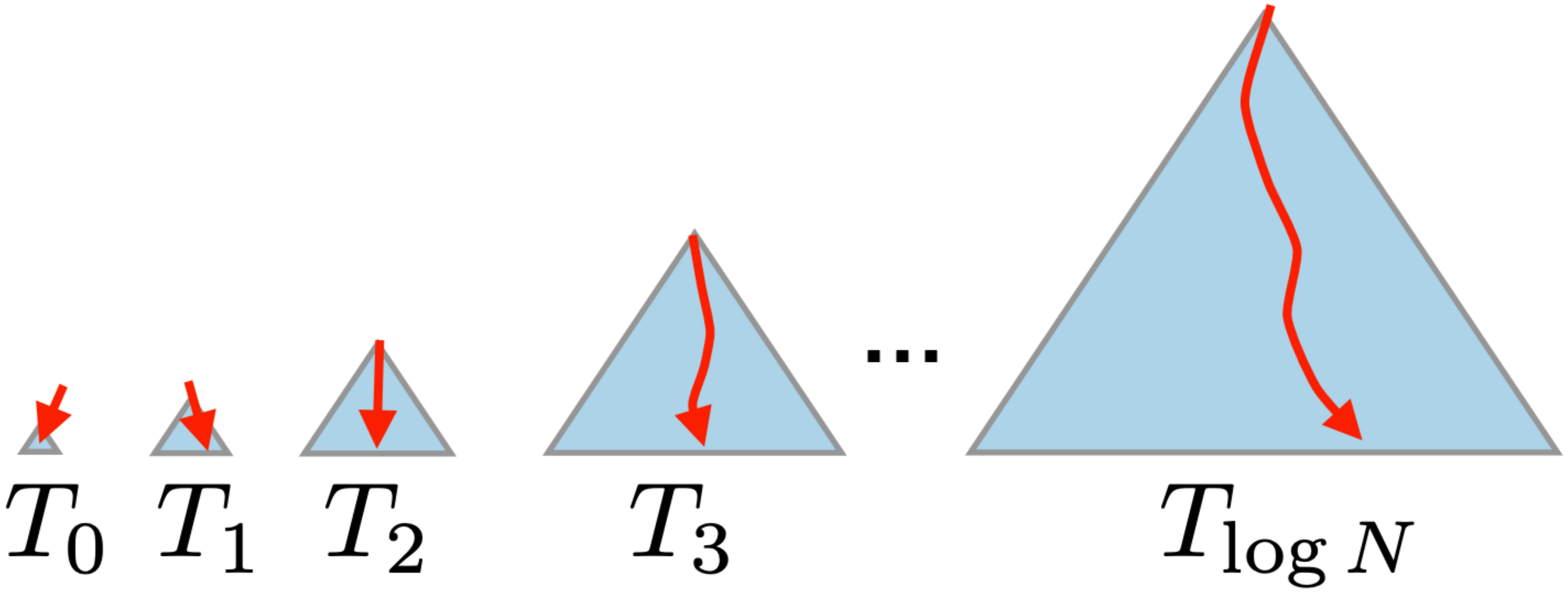
Analysis of Point Queries

Search cost:

$$\log_B N + \log_B N/2 + \log_B N/4 + \cdots + \log_B B$$

$$= \frac{1}{\log B}(\log N + \log N - 1 + \log N - 2 + \log N - 3 + \cdots + 1)$$

$$= O(\log N \log_B N)$$



Analysis of Inserts

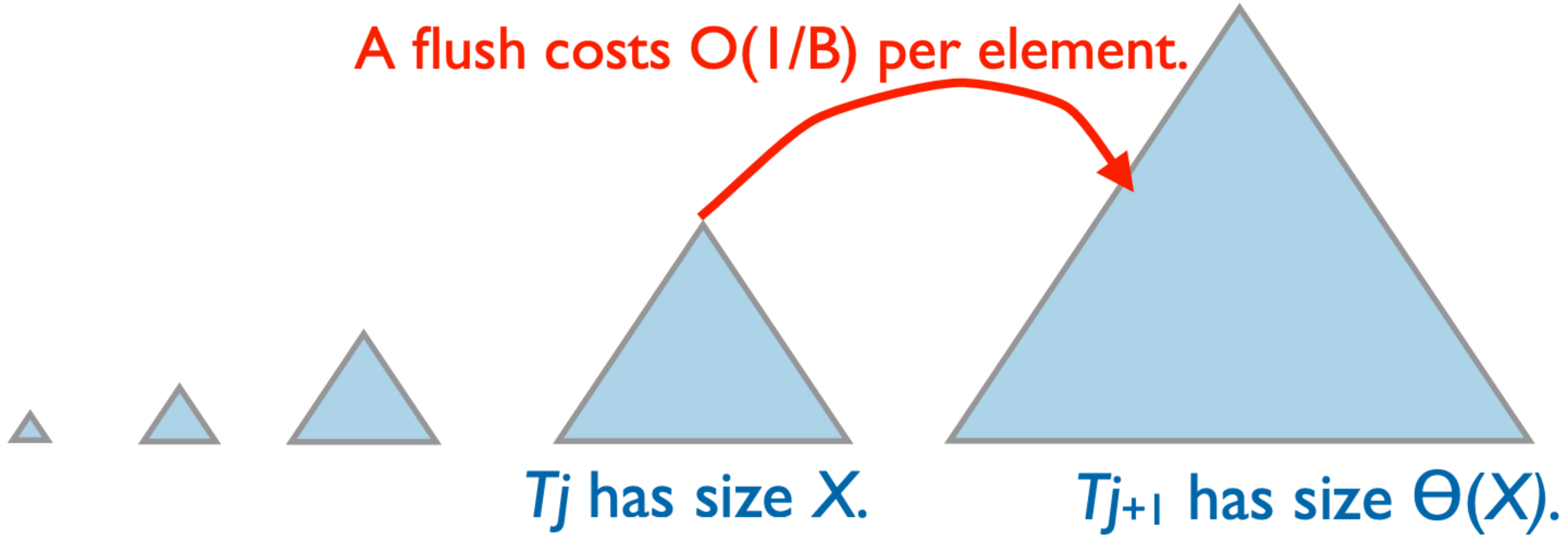
The cost to flush a tree *Tj* of size *X* is O(*X*/*B*).

- Flushing and rebuilding a tree is just a linear scan.

The cost per element to flush *Tj* is O(*1*/*B*).

The # times each element is moved is ≤ log *N*.

The insert cost is O((log *N*)/*B*) amortized memory transfers.



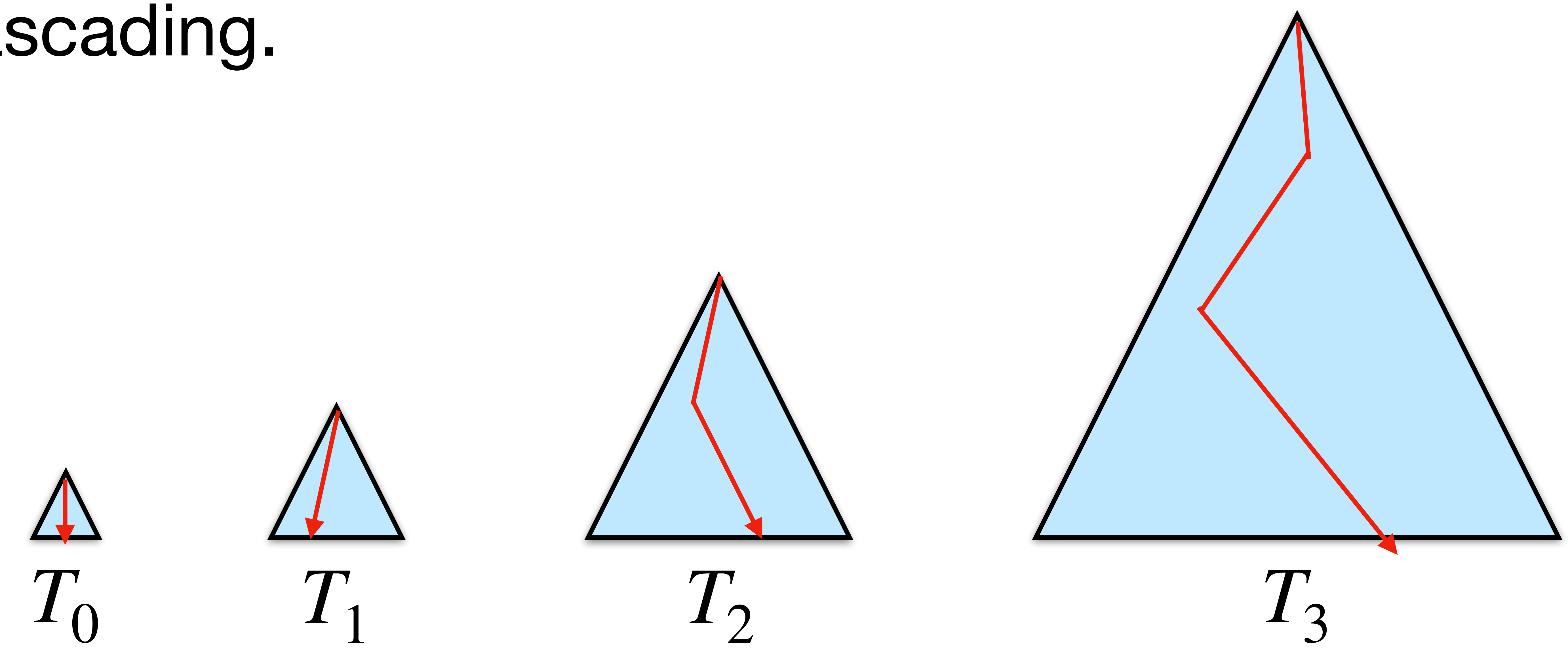
Samples from LSM Tradeoff Curve

	insert	point query
tradeoff (function of ε)	$O\left(\frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}}\right)$	$O\left((\log_B N)(\log_{1+B^\varepsilon} N)\right)$
sizes grow by *B* (ε=1)	$O\left(\log_B N\right)$	$O\left((\log_B N)(\log_B N)\right)$
sizes grow by *B*$^{1/2}$ (ε=1/2)	$O\left(\frac{\log_B N}{\sqrt{B}}\right)$	$O\left((\log_B N)(\log_B N)\right)$
sizes double (ε=0)	$O\left(\frac{\log N}{B}\right)$	$O\left((\log_B N)(\log N)\right)$

How to improve LSM-tree point queries?

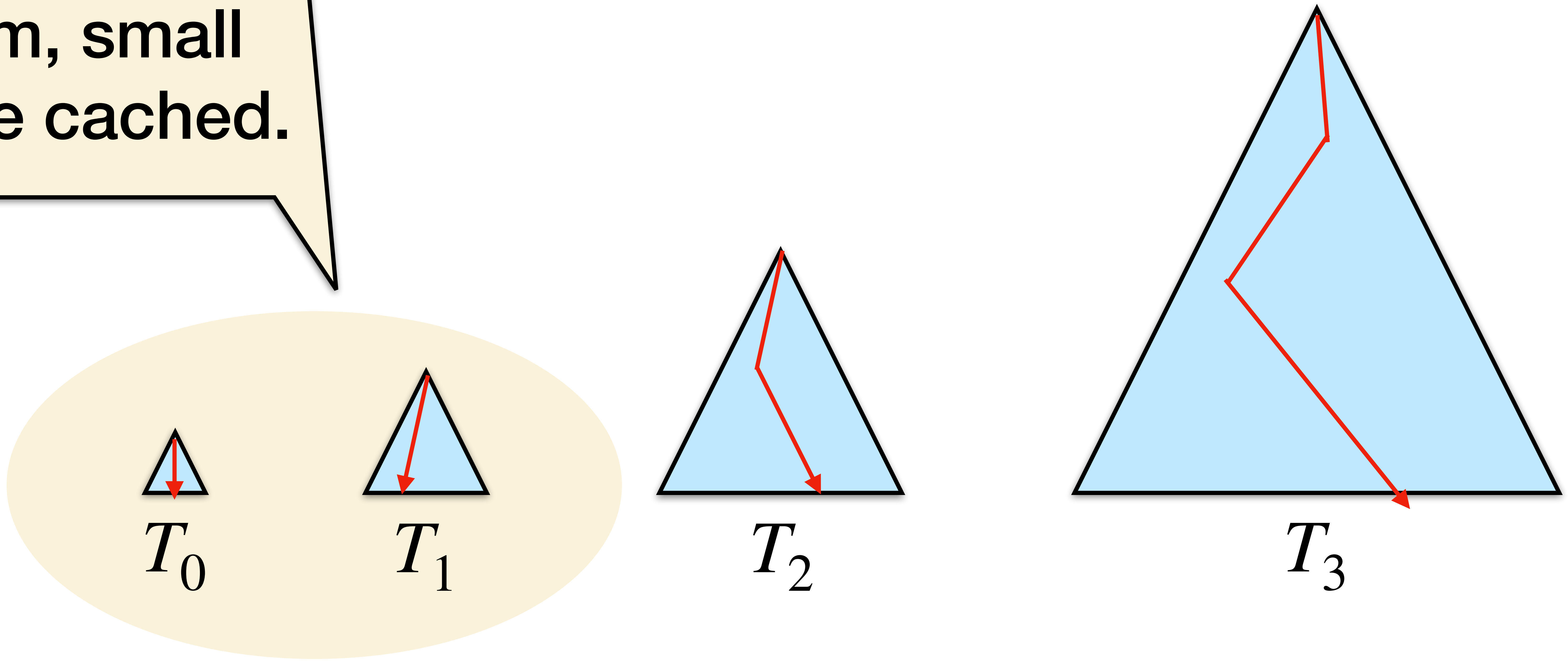
Looking in all those trees is expensive, but can be improved by:

- caching,
- filters (e.g., Bloom), and
- fractional cascading.



Caching in LSM trees

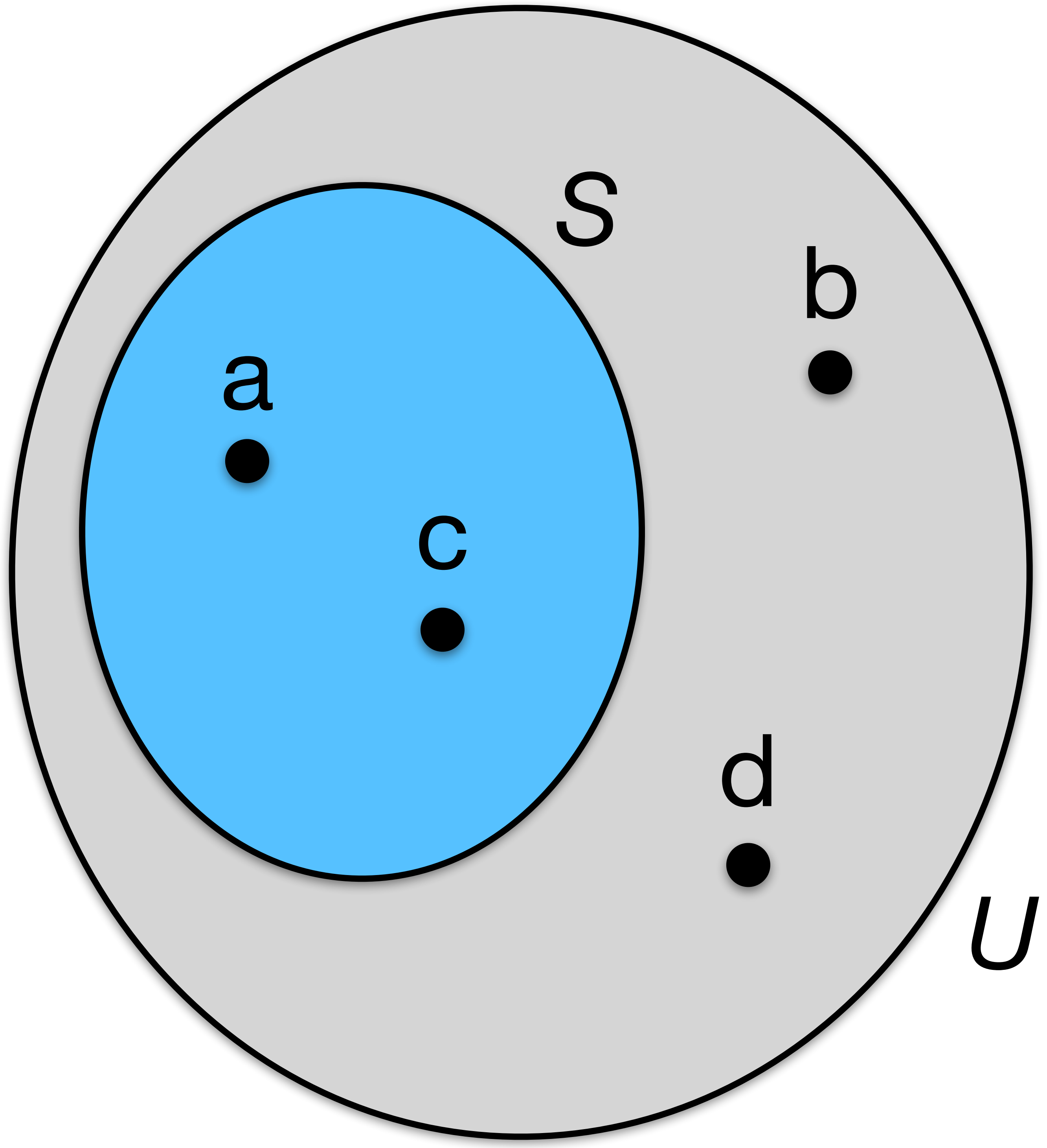
When the cache is warm, small trees are cached.



(Sidebar into filters)

Recap: Dictionary Data Structure

A dictionary maintains a set *S* from a universe *U*.



member(a): yes

member(b): no

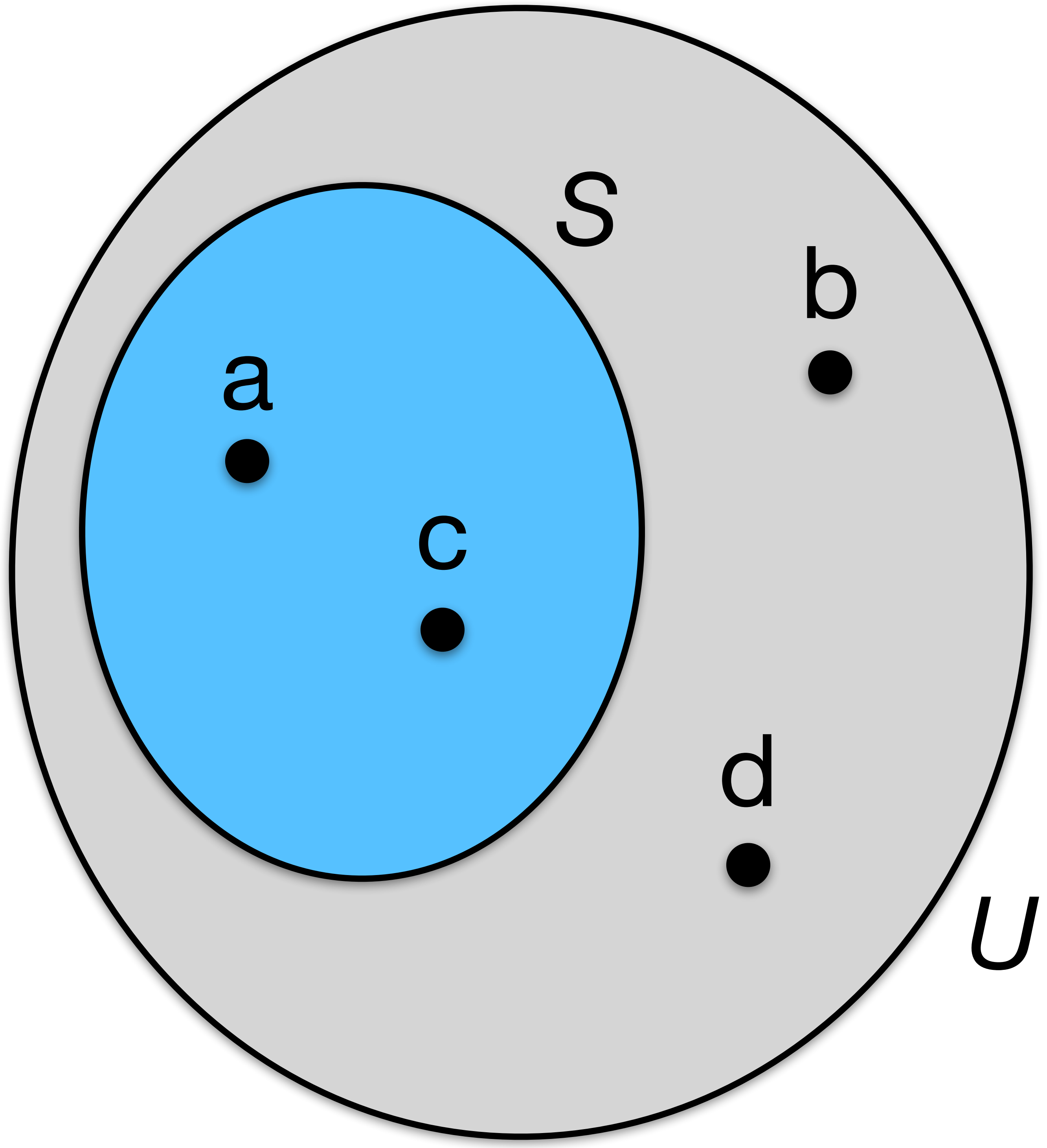
member(c): yes

member(d): no

A dictionary supports membership queries on *S*.

Filter Data Structure

A filter is an **approximate** dictionary.



member(a): yes 💚

member(b): no 💚

member(c): yes 💚

member(d): yes 😡

False positive

A filter supports **approximate membership queries** on *S*.

A Filter Guarantees a False-Positive Rate ε

If $q \in S$, return **yes** with probability 1 — **true positive**

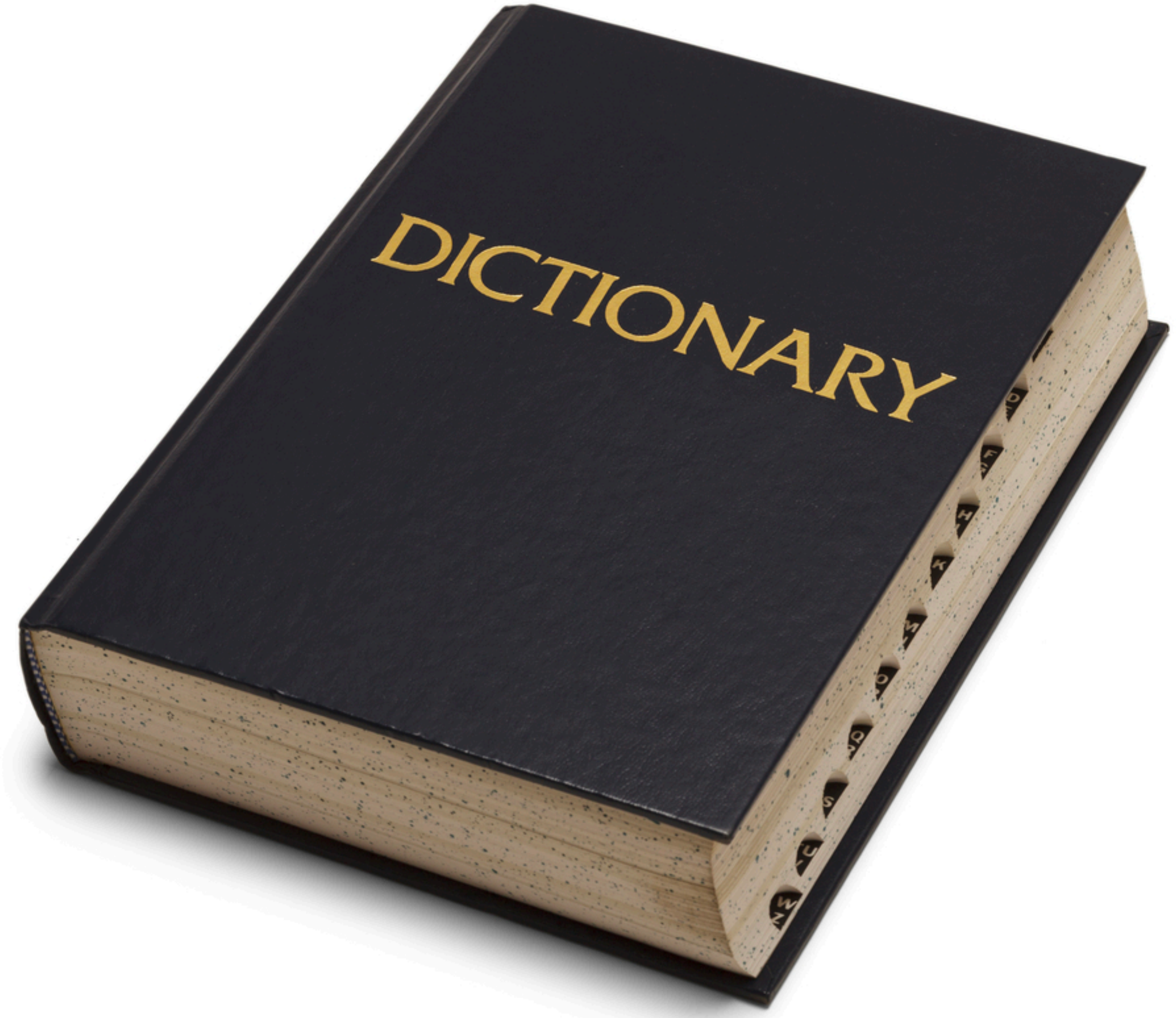
If $q \notin S$, return
- **no** with probability $> 1 - \varepsilon$ — **true negative**
- **yes** with probability $\leq \varepsilon$ — **false positive**

One-sided error (no false negatives)

False-positive rate enables filters to be compact

space of filter $\geq n \log(1/\varepsilon)$

space of dictionary $= \Omega(n \log |U|)$



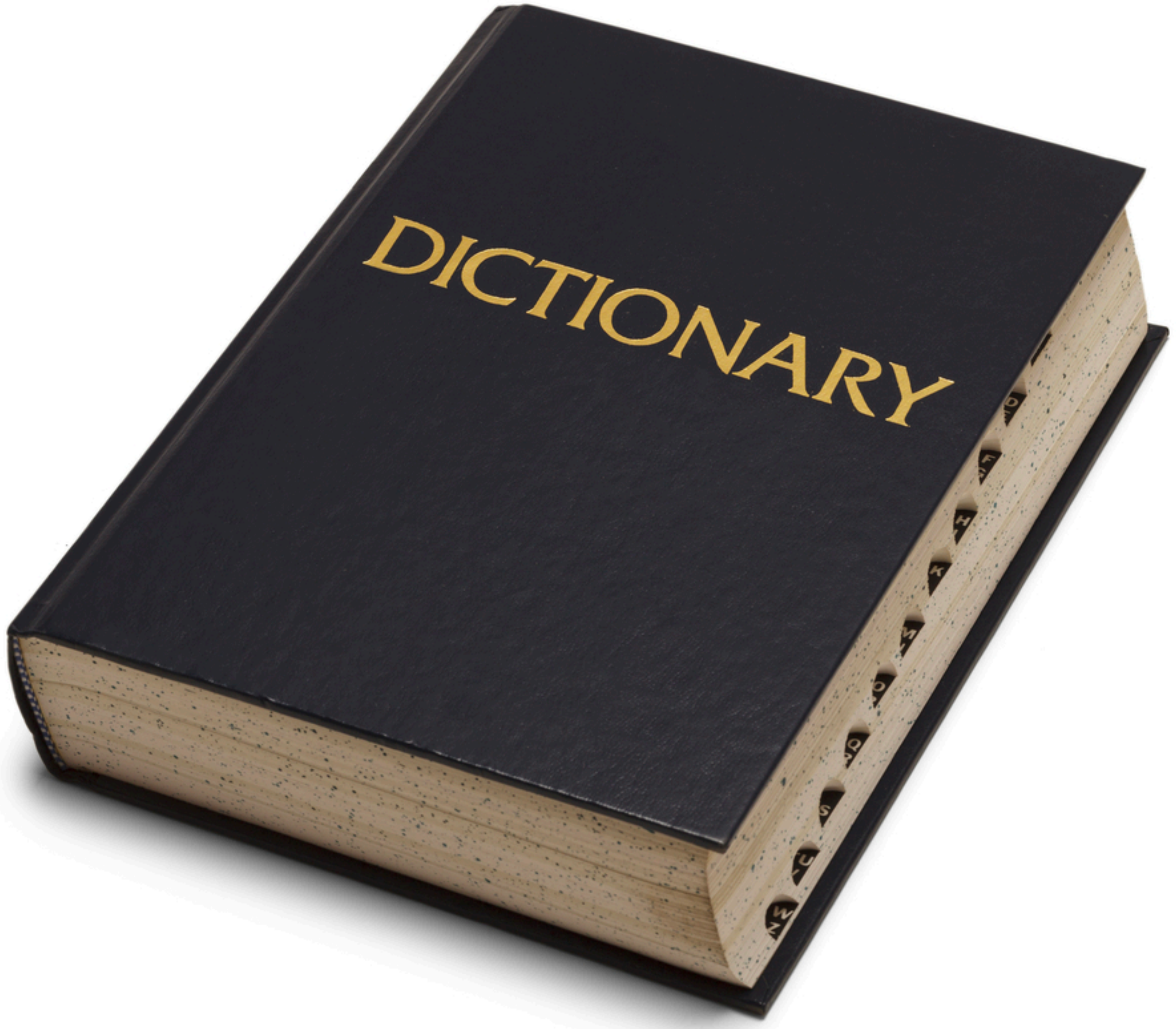
False-positive rate enables filters to be compact

small

space of filter $\geq n \log(1/\varepsilon)$

large

space of dictionary $= \Omega(n \log |U|)$

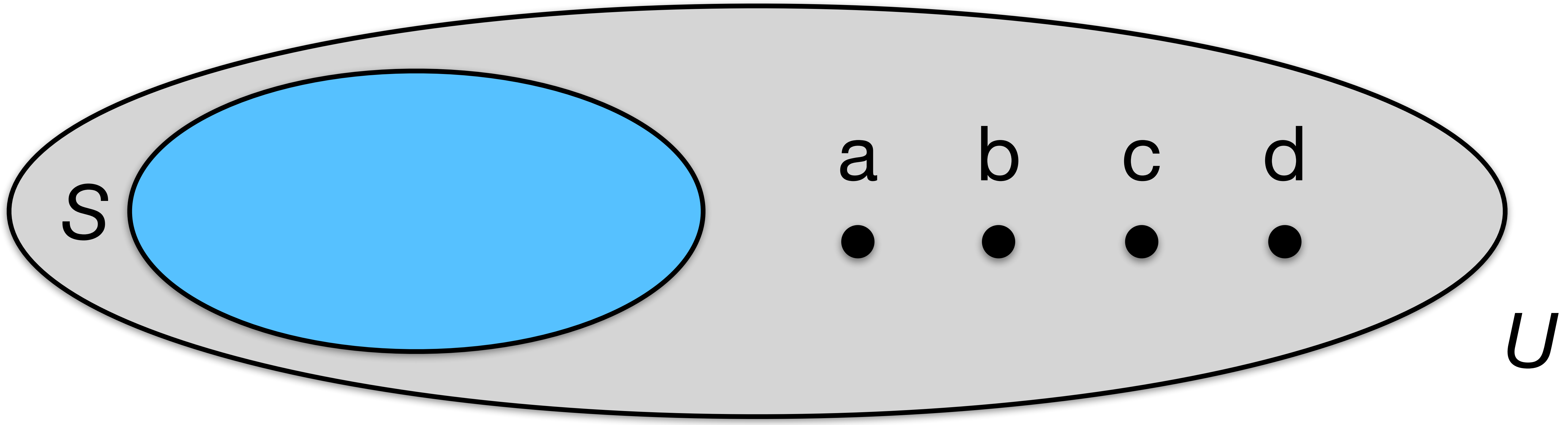


Classic Filter: The Bloom Filter [Bloom '70]

the most well-known one

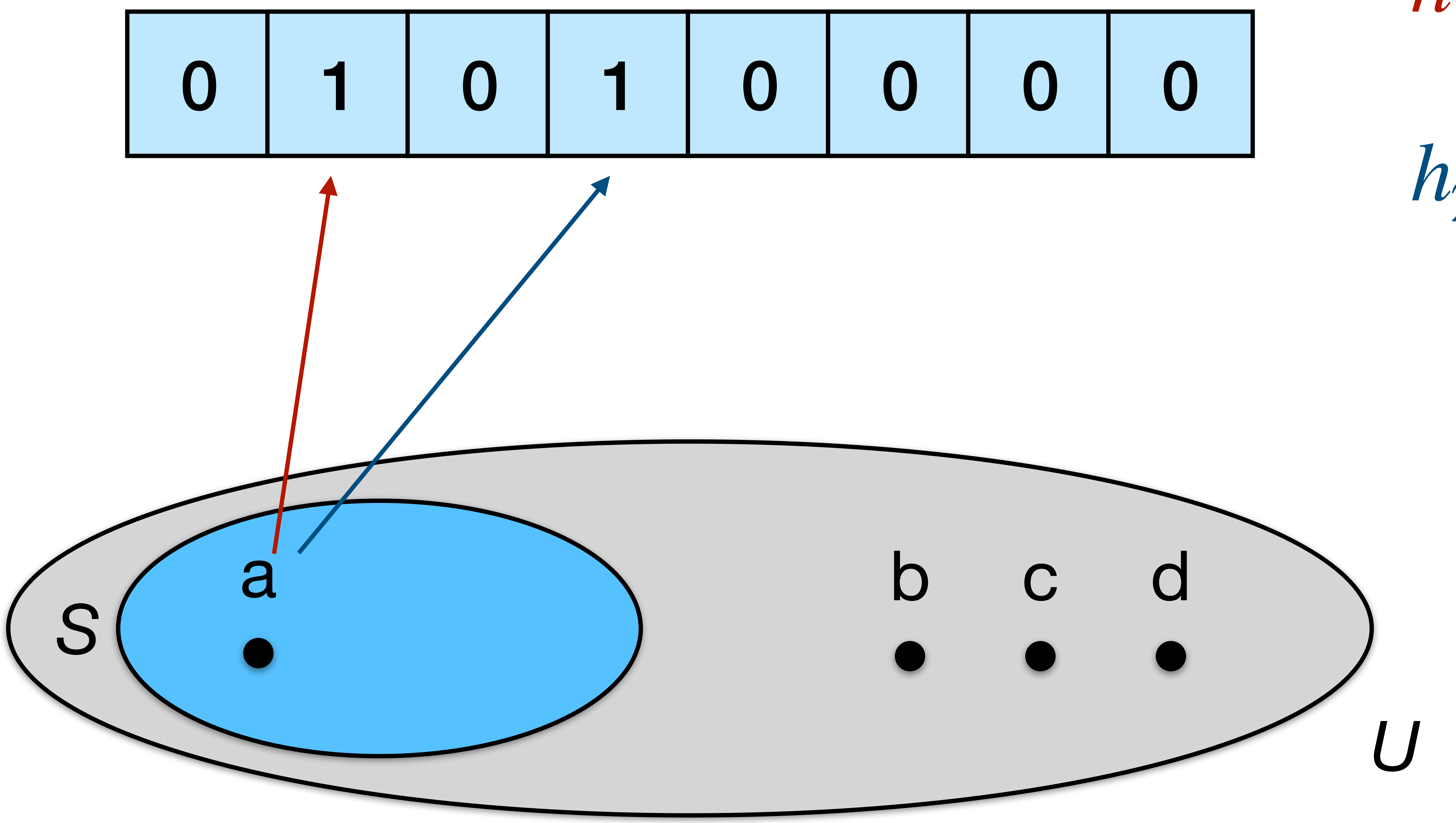
Bloom filter: a bit array + k hash functions (k=2 in this example)

0	0	0	0	0	0	0	0



Classic Filter: The Bloom Filter [Bloom ’70]

Bloom filter: a bit array + k hash functions (k=2 in this example)

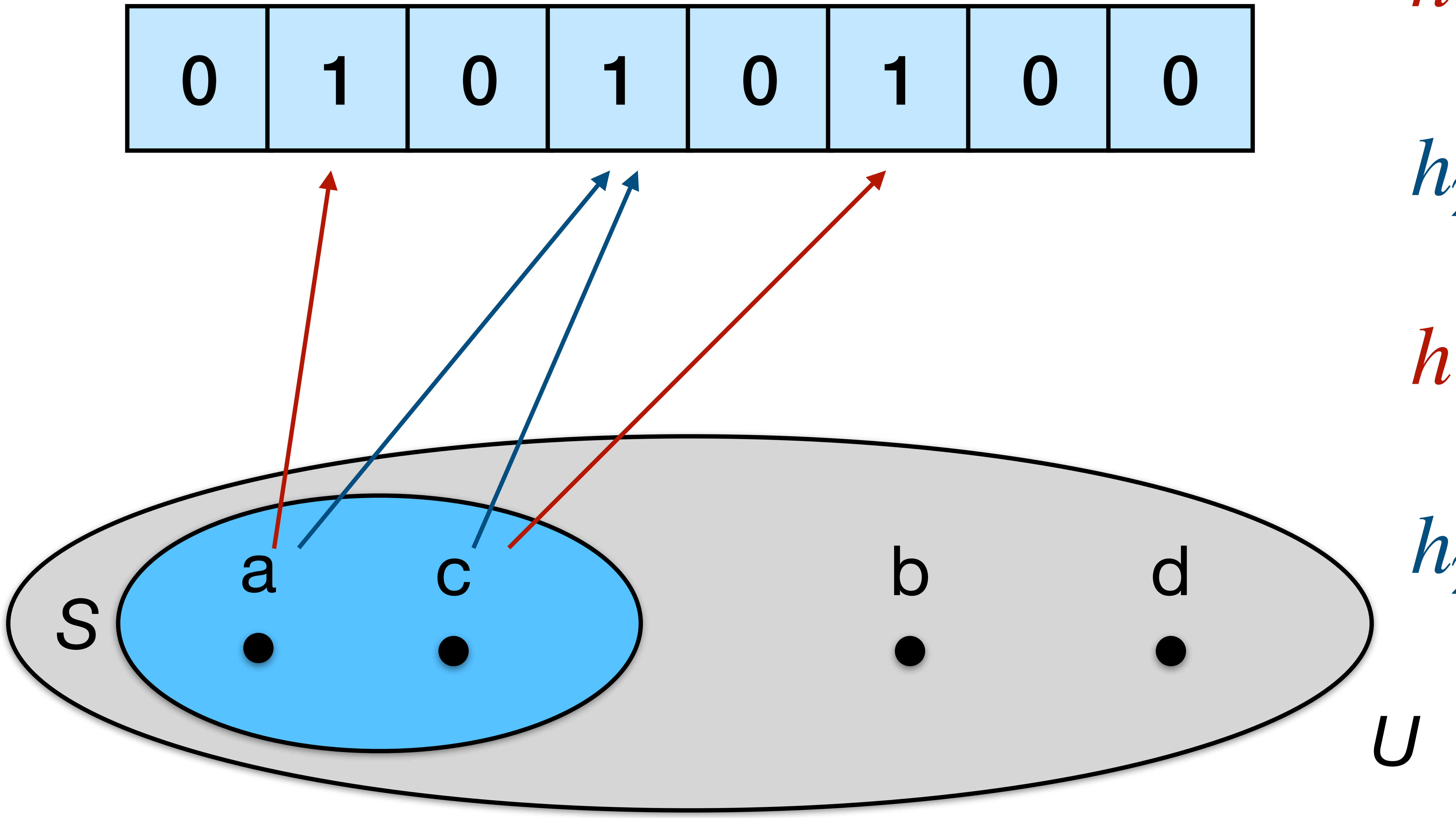


$h_1(a) = 1$

$h_2(a) = 3$

Classic Filter: The Bloom Filter [Bloom '70]

Bloom filter: a bit array + k hash functions (k=2 in this example)



$h_1(a) = 1$

$h_2(a) = 3$

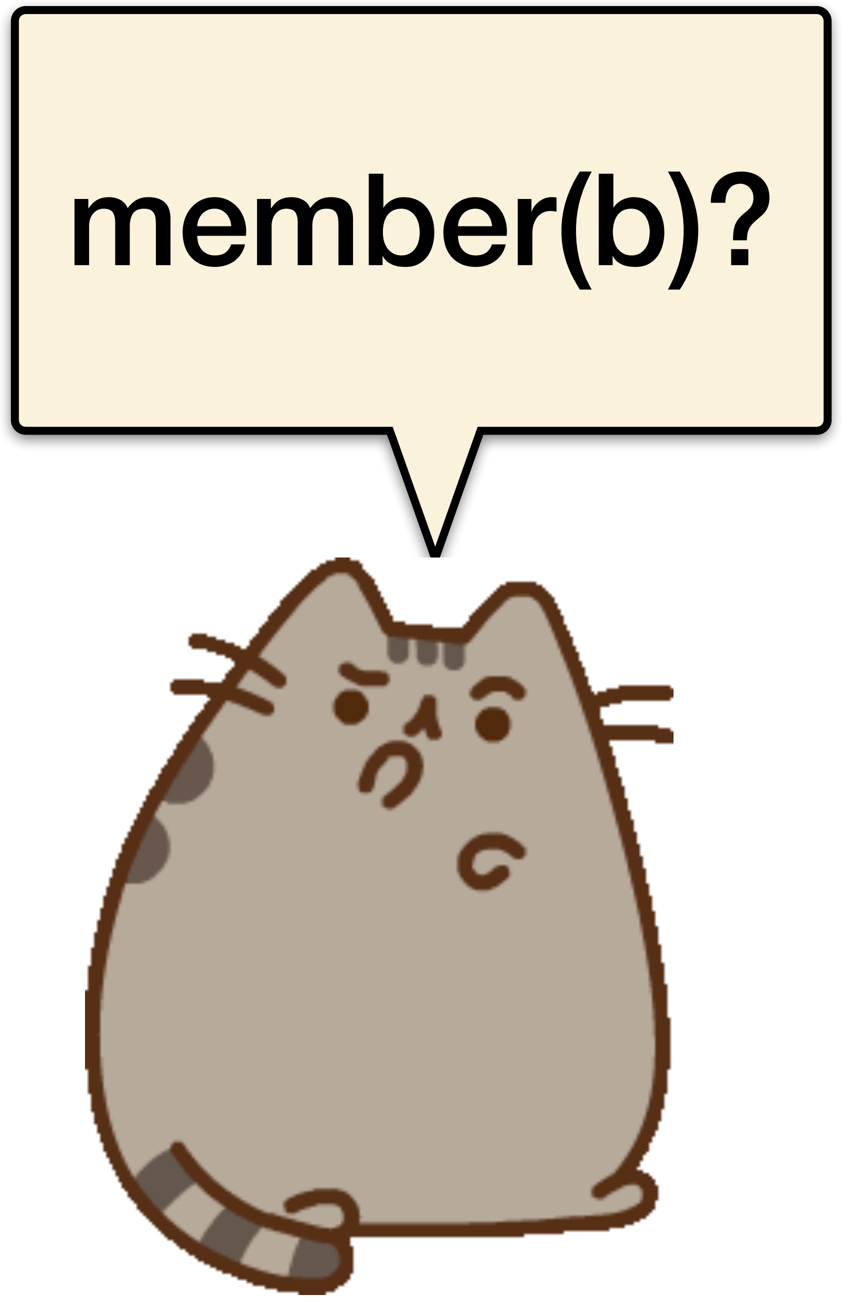
$h_1(c) = 5$

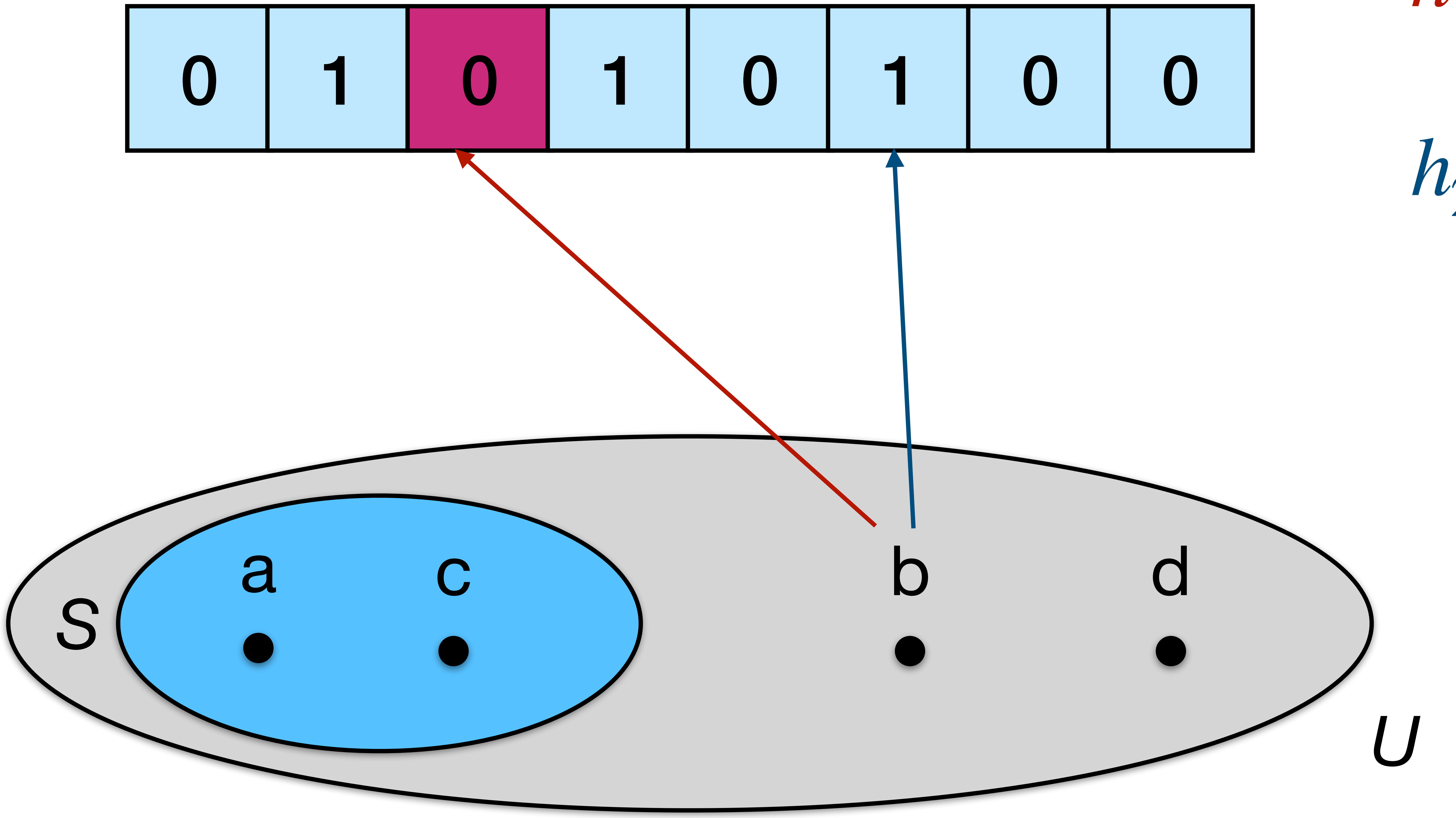
$h_2(c) = 3$

Slide from Michael Bender

Classic Filter: The Bloom Filter [Bloom ’70]

Bloom filter: a bit array + k hash functions (k=2 in this example)





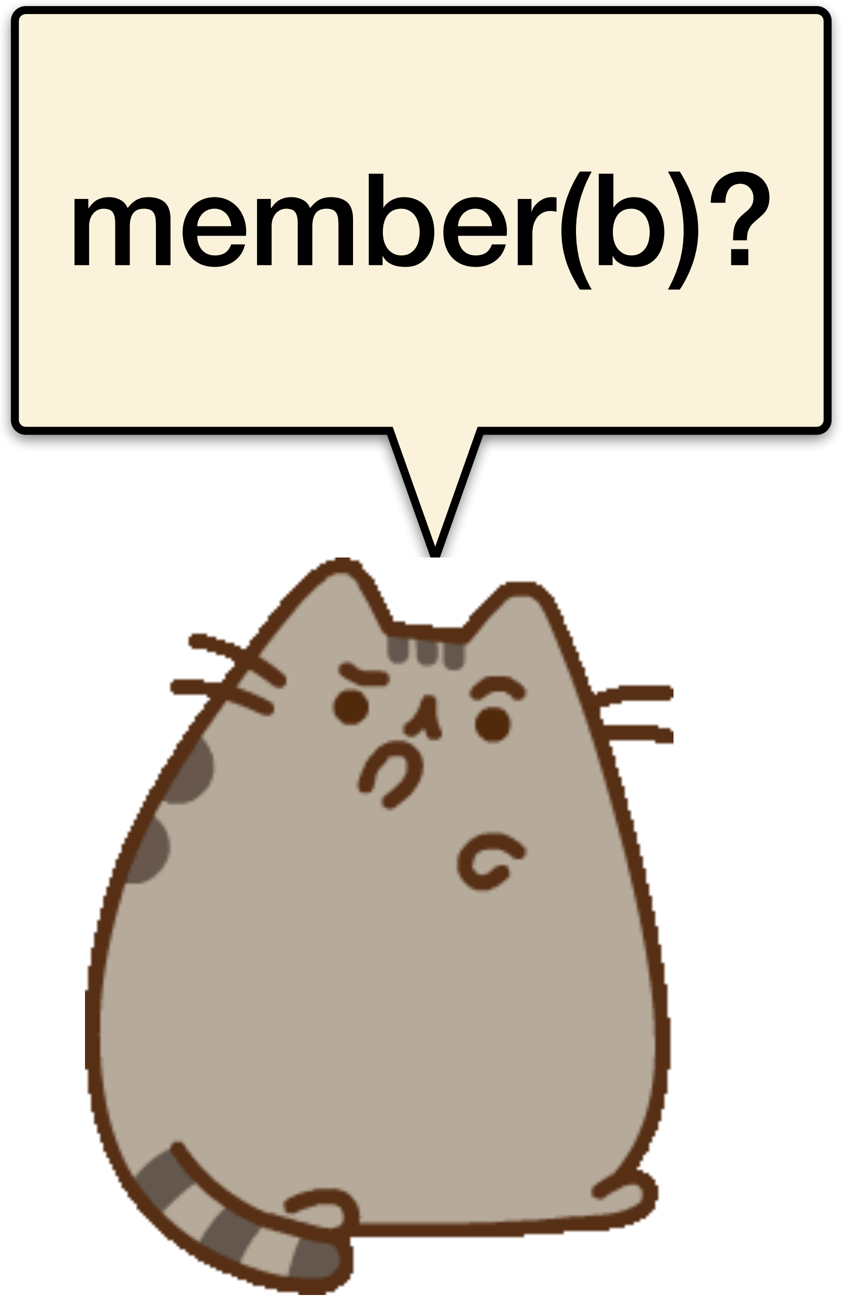
$h_1(b) = 2$

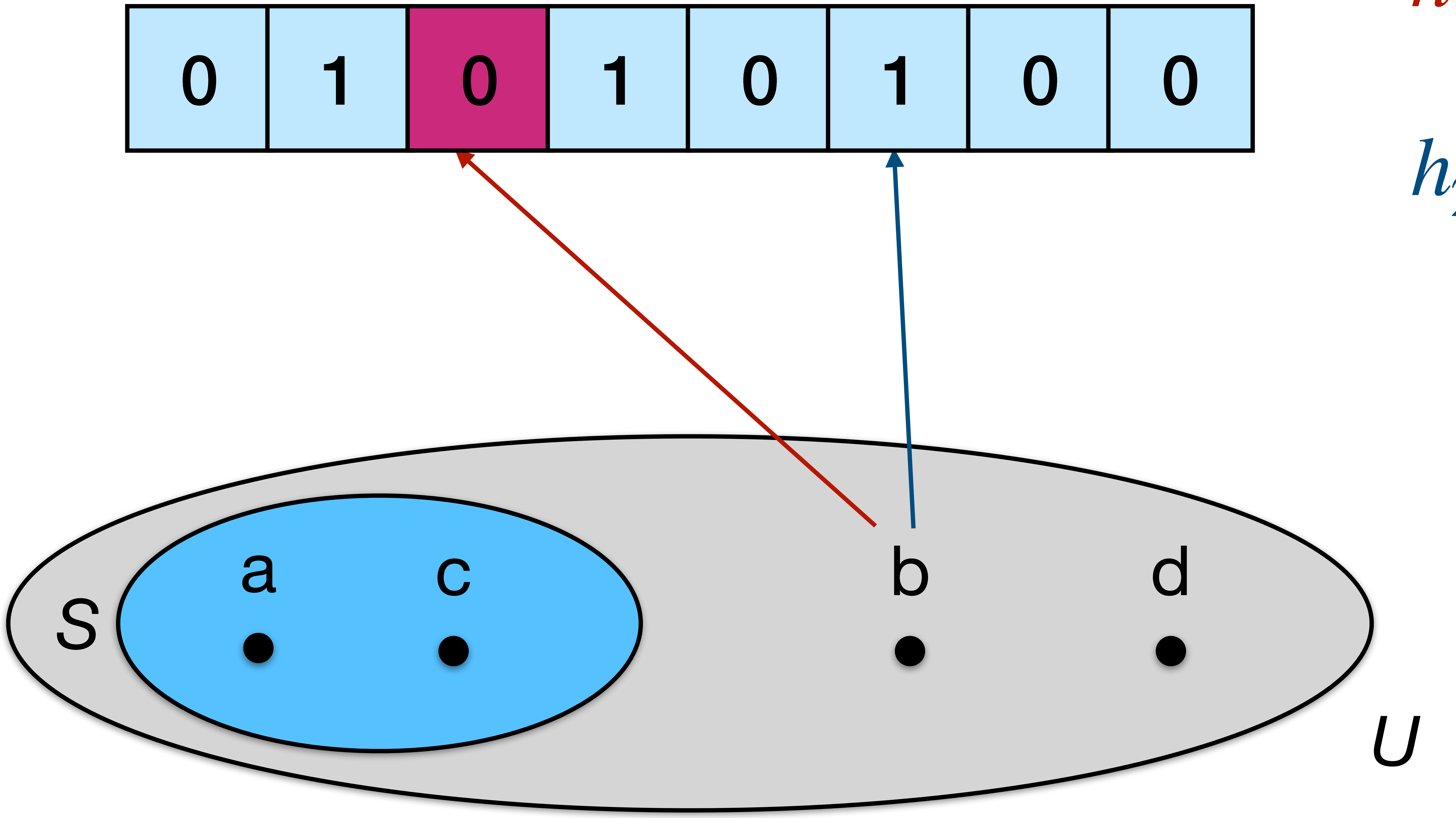
$h_2(b) = 5$

Slide from Michael Bender

Classic Filter: The Bloom Filter [Bloom ’70]

Bloom filter: a bit array + k hash functions (k=2 in this example)





$h_1(b) = 2$

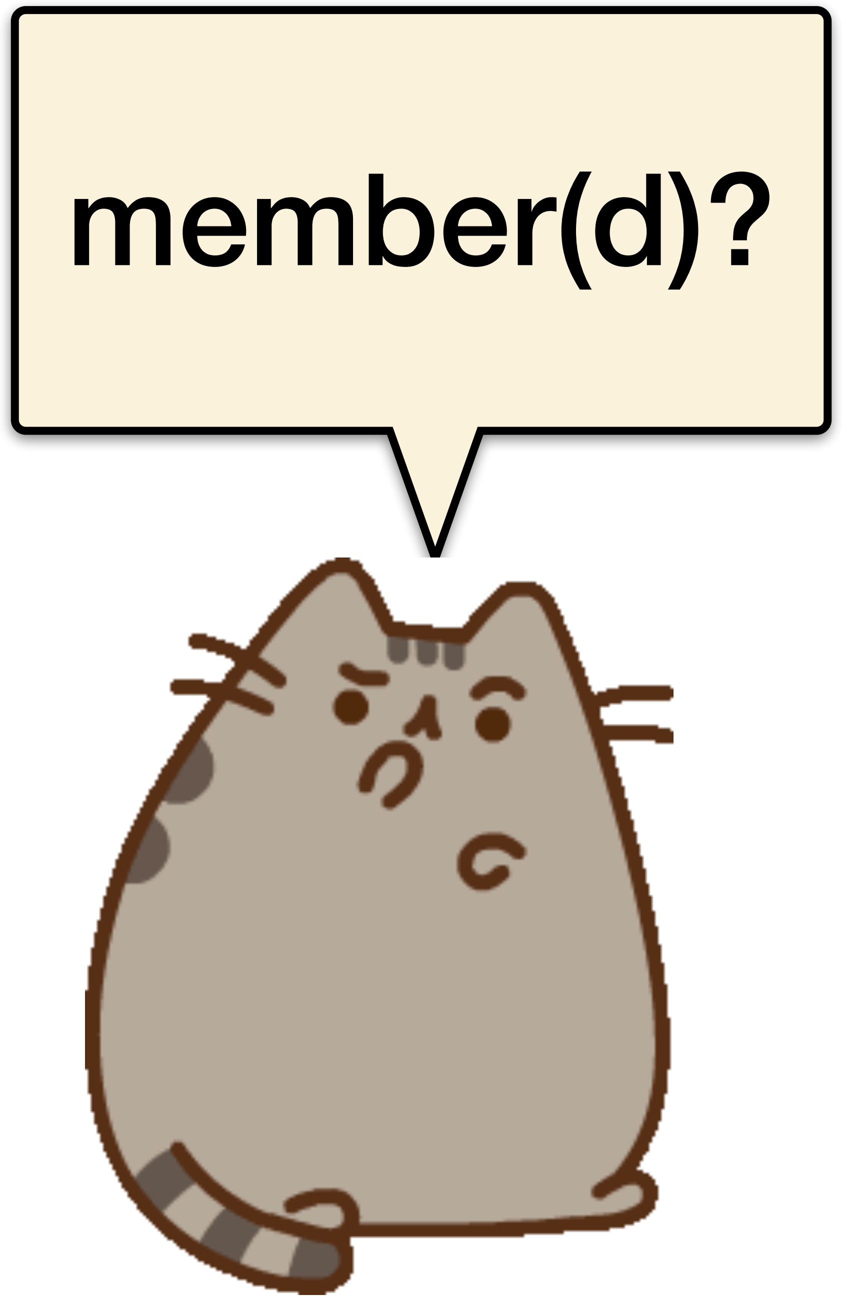
$h_2(b) = 5$

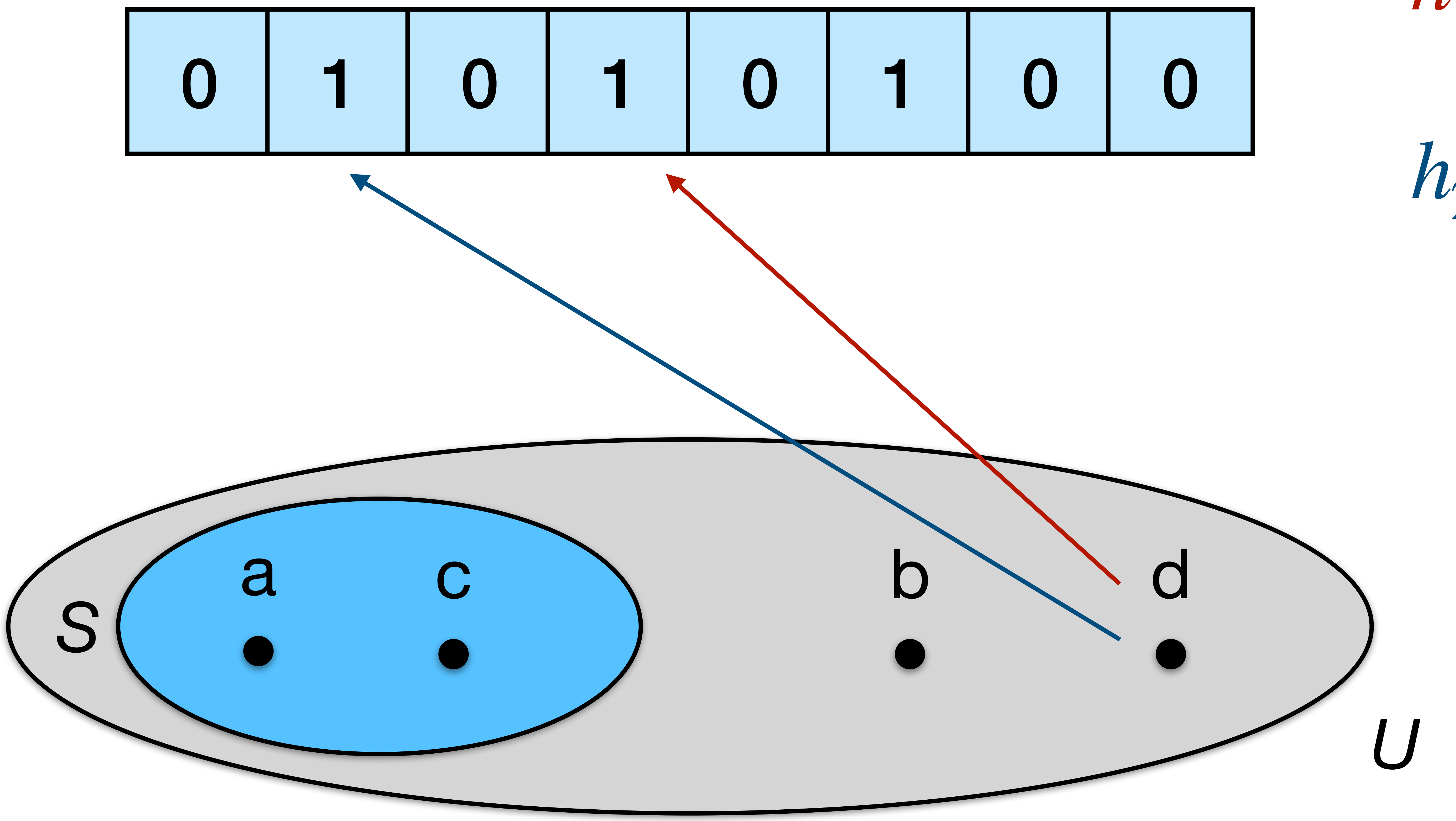
Classic Filter: The Bloom Filter [Bloom ’70]

Bloom filter: a bit array + k hash functions (k=2 in this example)

$h_1(d) = 3$

$h_2(d) = 1$

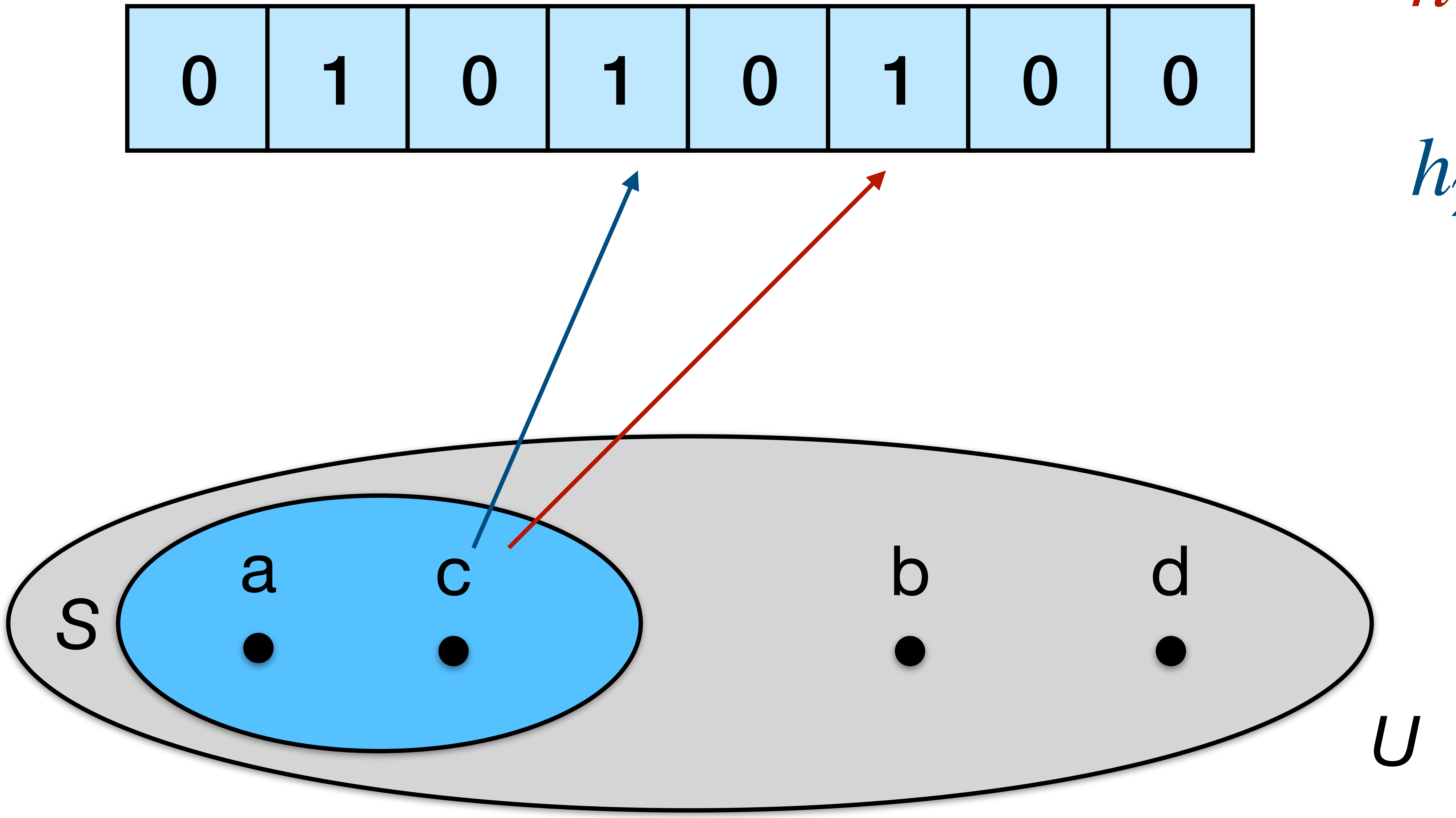




False positive

Bloom filters don't support deletes

Issue: on a delete, which 1s get decremented?



$h_1(c) = 5$

$h_2(c) = 3$

Bloom Filter Space Usage

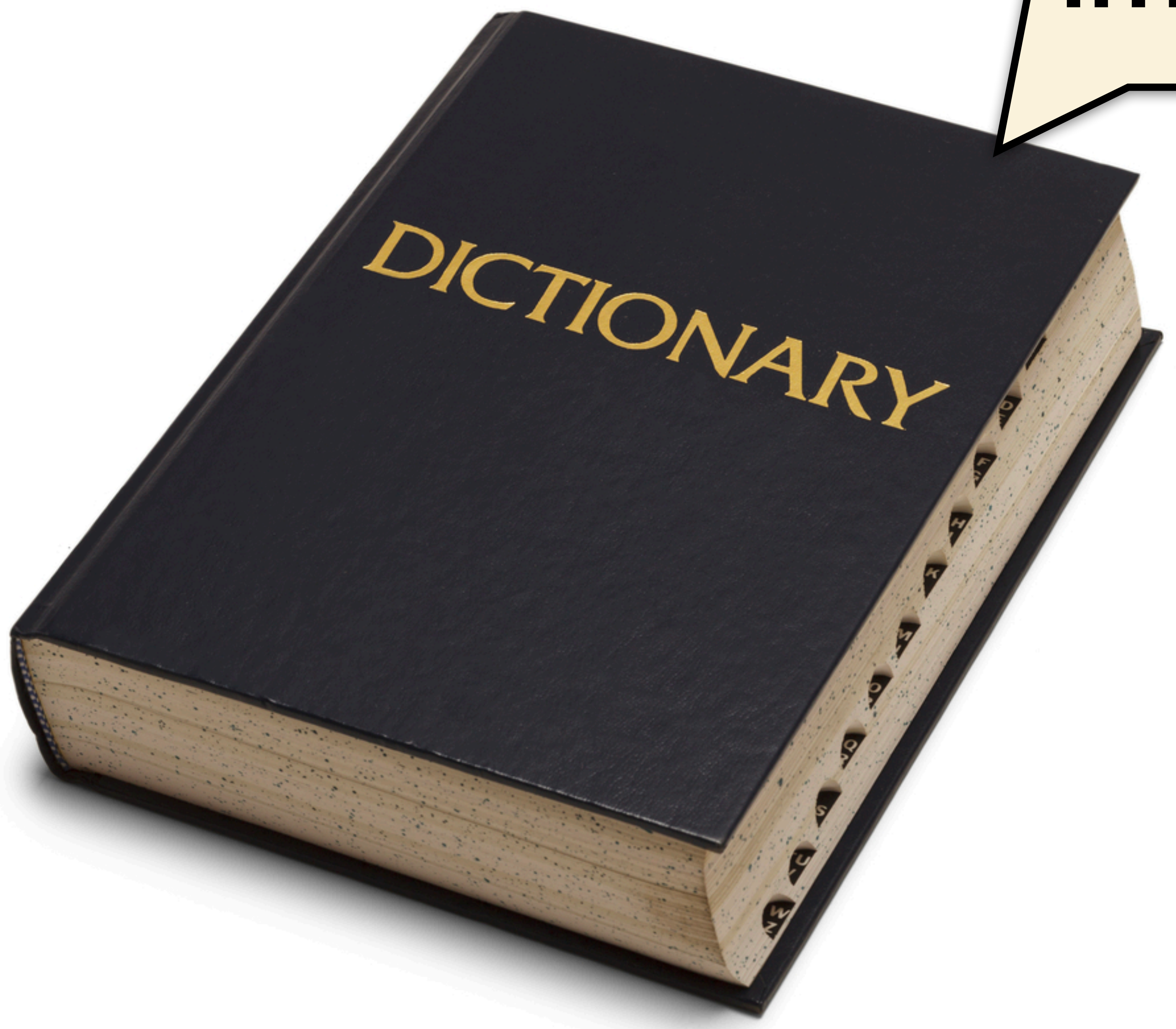
Bloom filter space usage with false-positive rate ε:

~ $1.44 \lg(1/\varepsilon)$ bits per element.

Example: for ε = 2%, bits / element ~ 8.

Usually about 1 byte per element

Usually at least 4 or 8 bytes per element (in implementation)



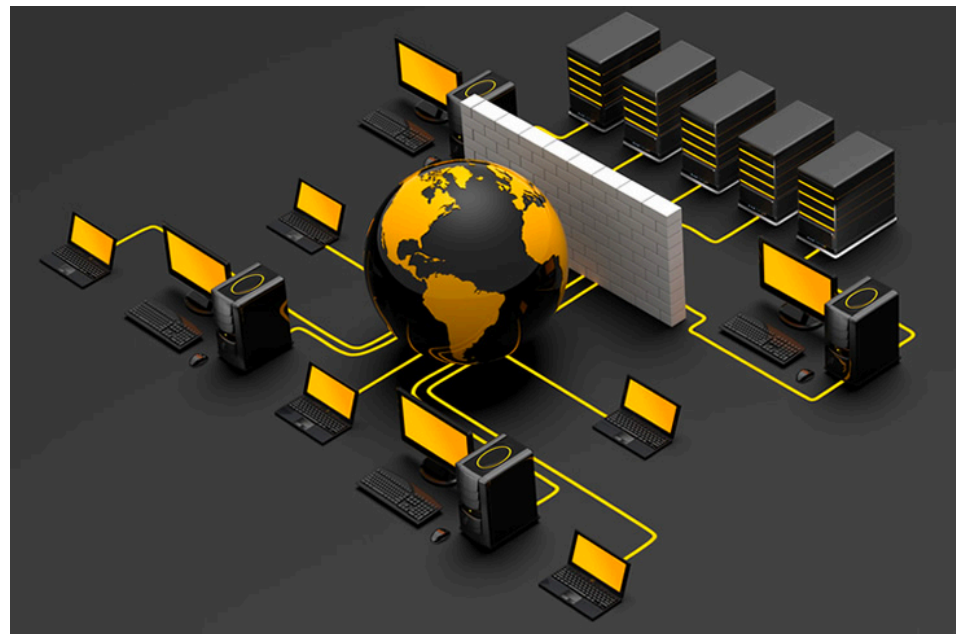
Slide from Michael Bender

Bloom filters are ubiquitous

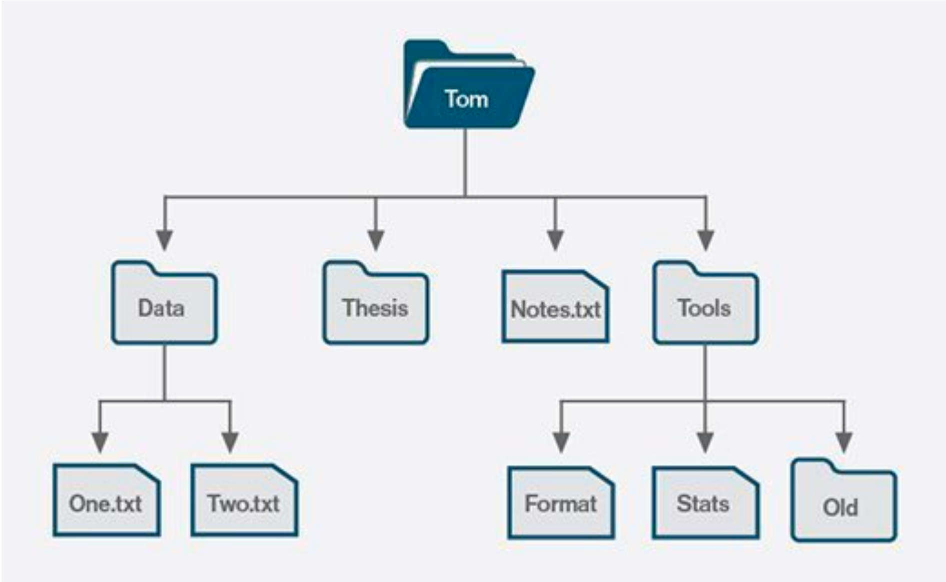
Over 10k citations

Computational biology

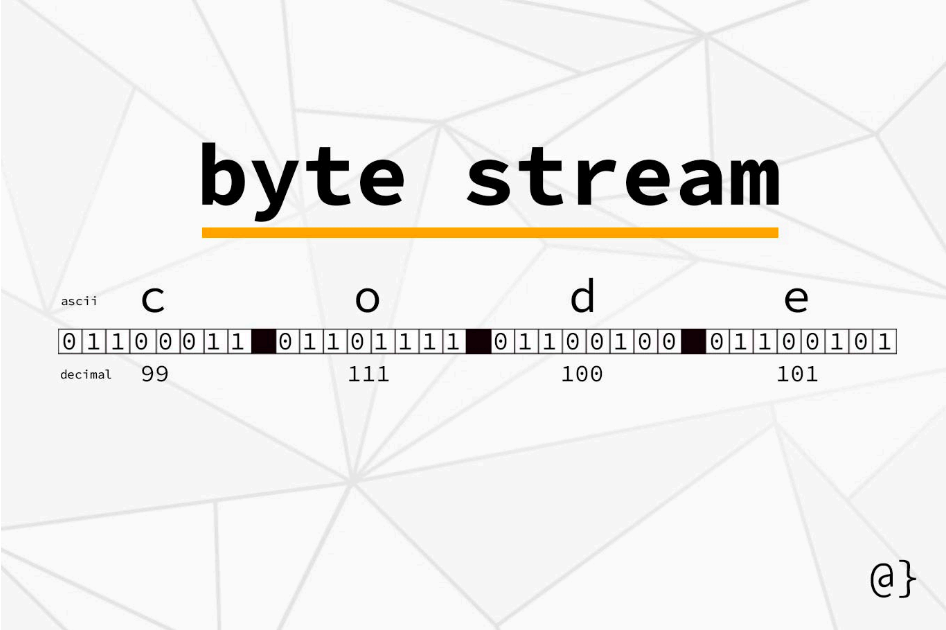
Databases



Networking



Storage systems



Streaming applications

Types of Filters

Bloom filters are the most common filter, but there are many more types:

- Bloom filters
- Quotient filter
- Cuckoo filter
- XOR filter
- Ribbon filter

Dynamic: set of items not known in advance

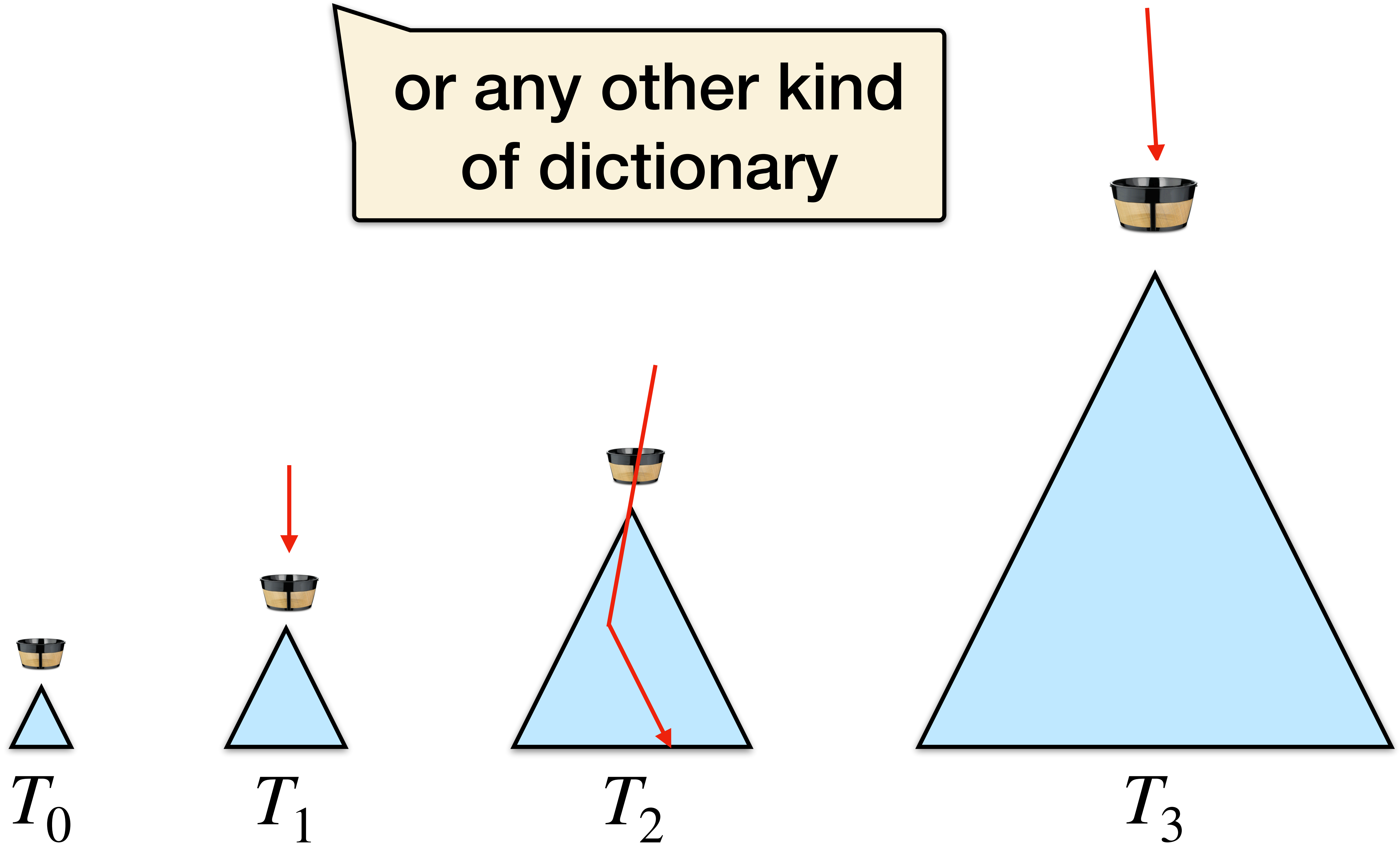
Can support **deletes**

Static: set of items is known in advance

Bloom Filters in LSM Trees

Bloom filters can **avoid point queries** for elements that are not in a particular B-tree in a LSM tree.

or any other kind of dictionary



Speedup from Filter Usae

Suppose we have a workload with *A* positive and *B* negative queries.

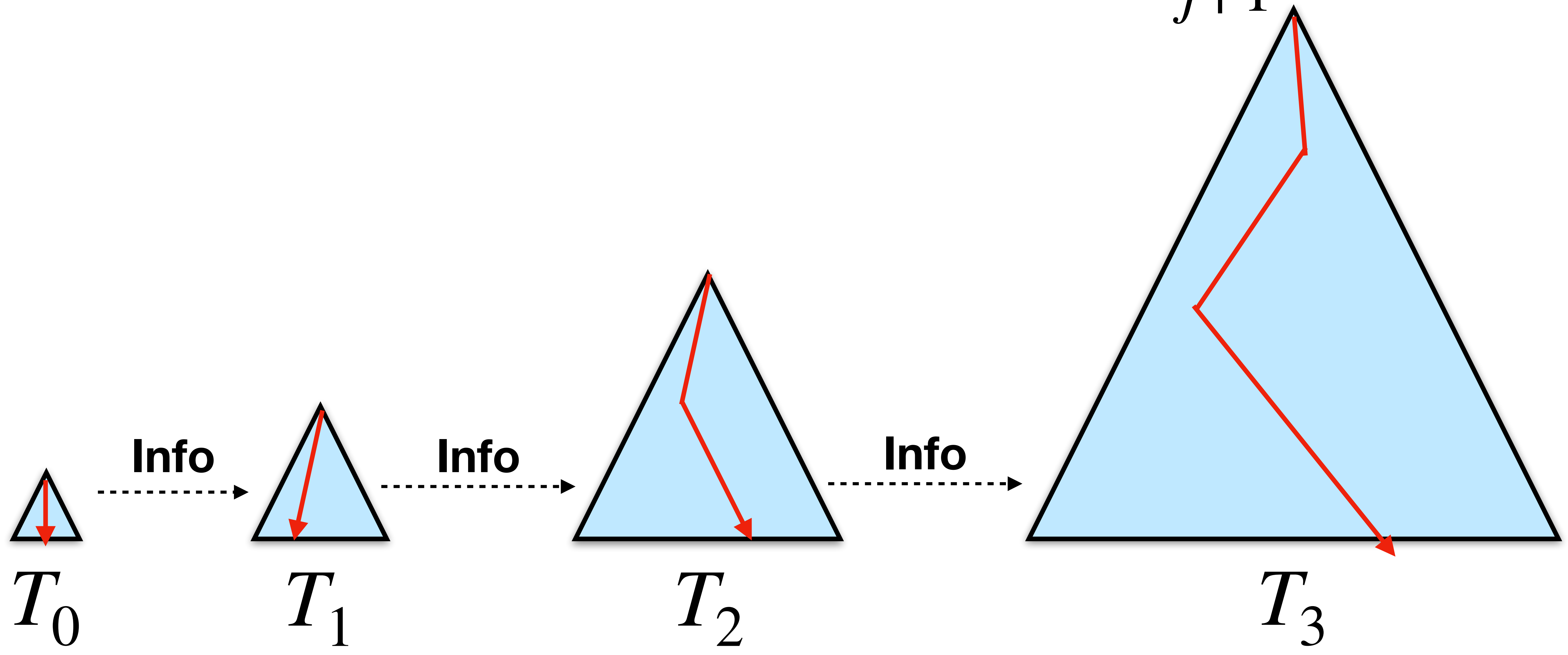
Dictionary without filter	**Dictionary with filter**
$A + B$	$A + \varepsilon B$
😡	💚

Remote access to dictionary

Fractional cascading in LSM trees

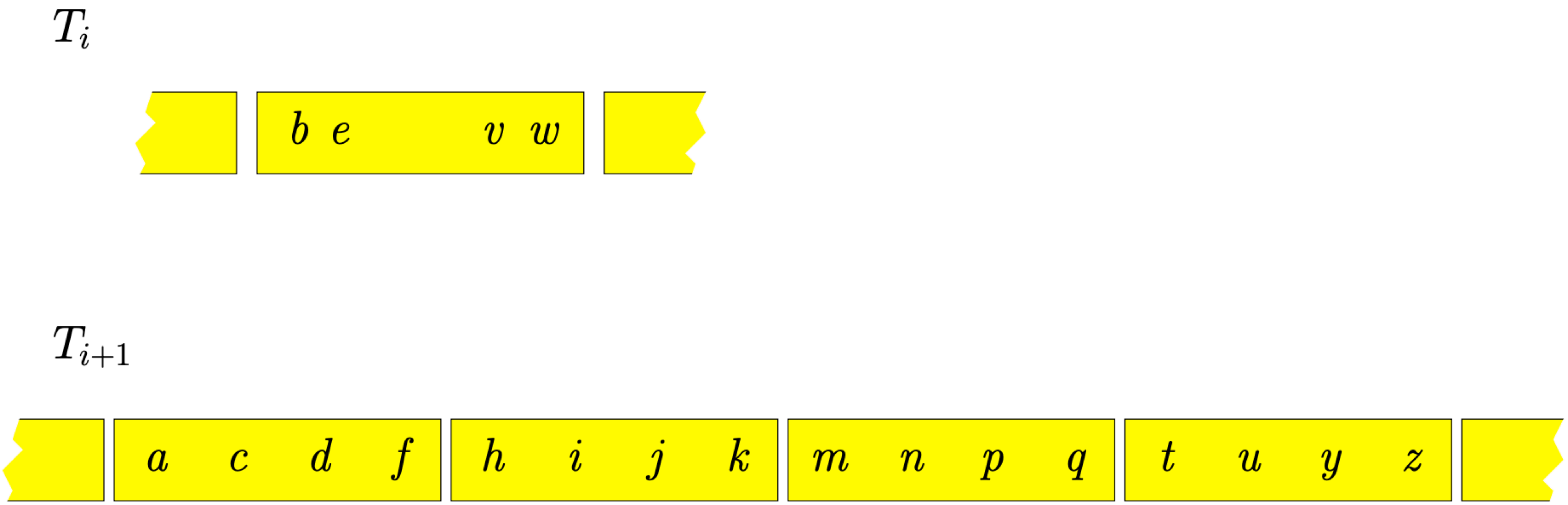
Instead of avoiding searches in trees (e.g., with filters), we can use a technique called **fractional cascading** to reduce the cost of searching each B-tree to $O(1)$.

Idea: We're looking for a key, and we already know where it should have been in T_j, try to **use that information** to search T_{j+1}.



Searching in one tree helps the next

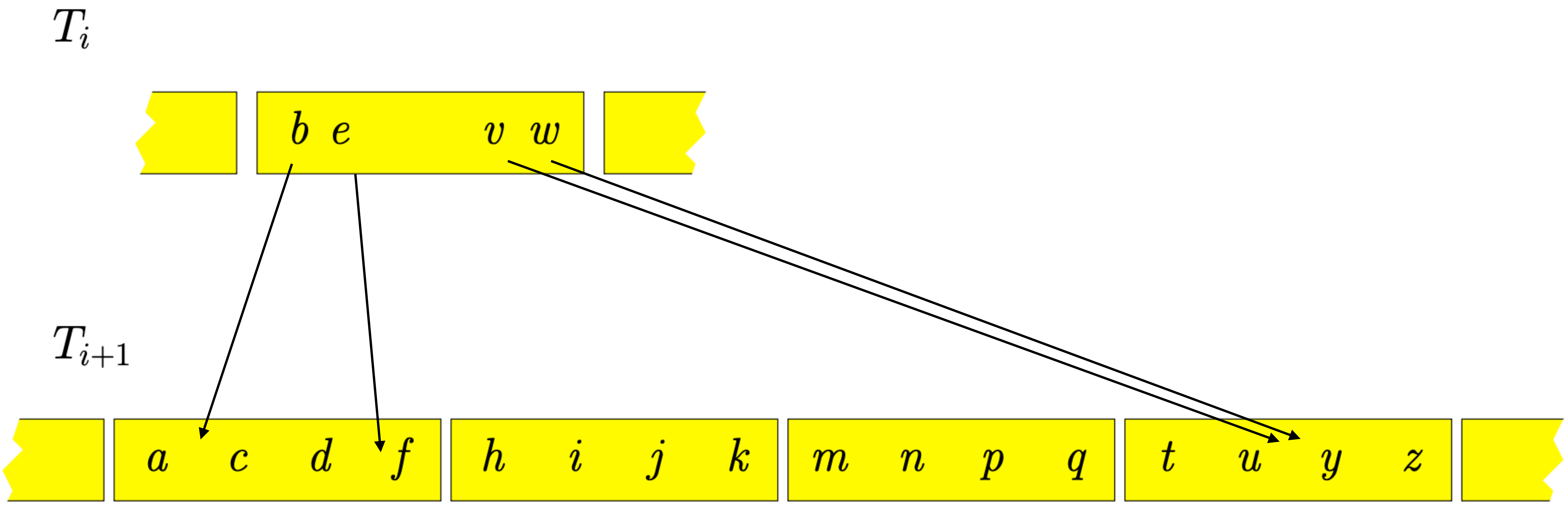
Looking up c, in T_i we know its between b and e.



(illustrating only the leaf level of each tree)

Forwarding pointers

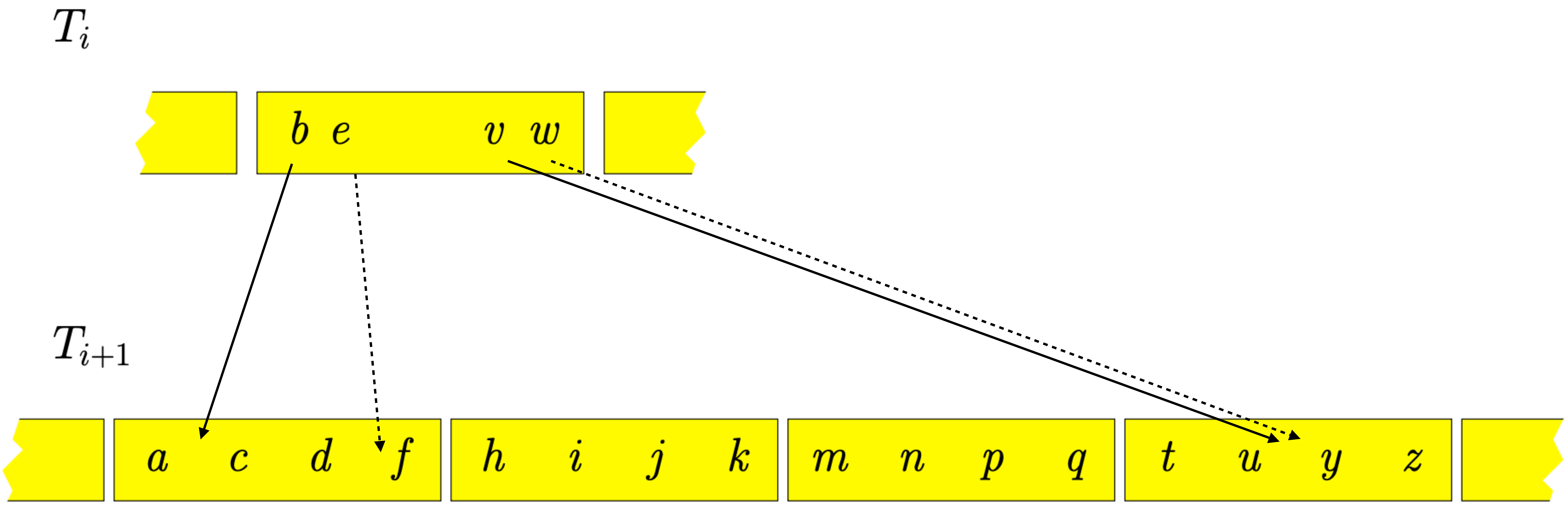
If we add **forwarding pointers** to the first tree, we can jump straight to the node in the second tree, to find c.



(illustrating only the leaf level of each tree)

Removing redundant forward pointers

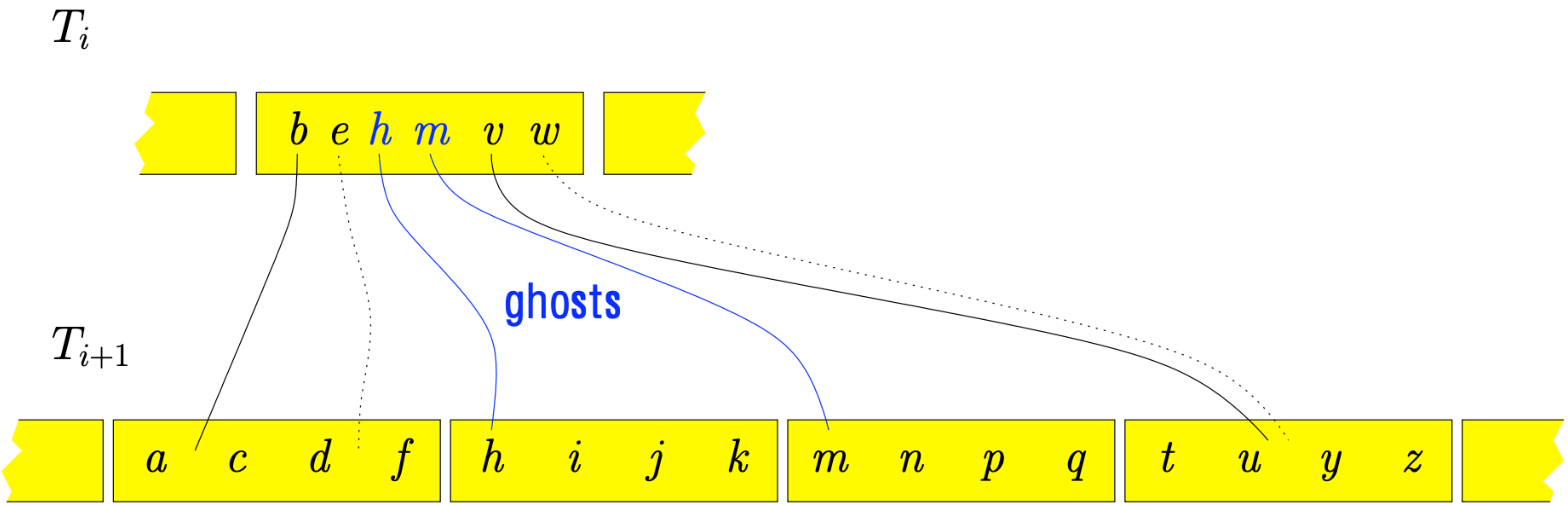
We need only **one forwarding pointer for each block** in the next tree. Remove the redundant ones.



(illustrating only the leaf level of each tree)

Ghost pointers

We need a forwarding pointer for every block in the next tree, even if there are no corresponding pointers in this tree. Add **ghosts**.

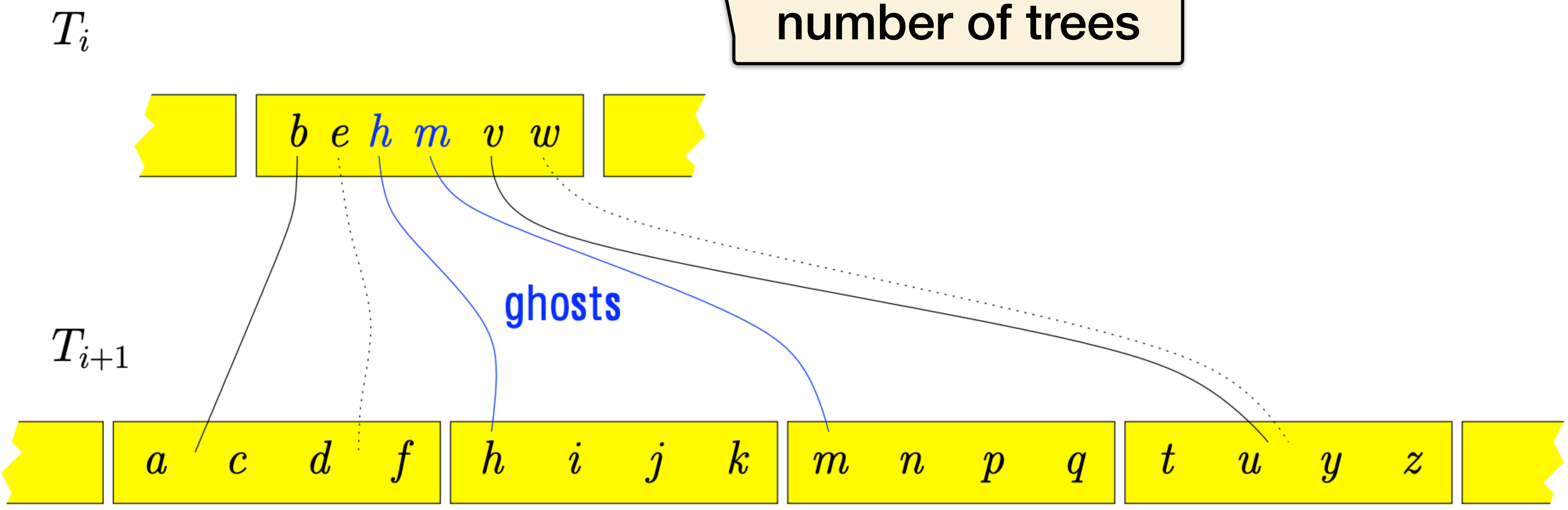


(illustrating only the leaf level of each tree)

LSM tree + forward + ghost = fast queries

With forward pointers and ghosts, LSM trees require only one I/O per tree, and point queries cost only $O(\log_R N)$

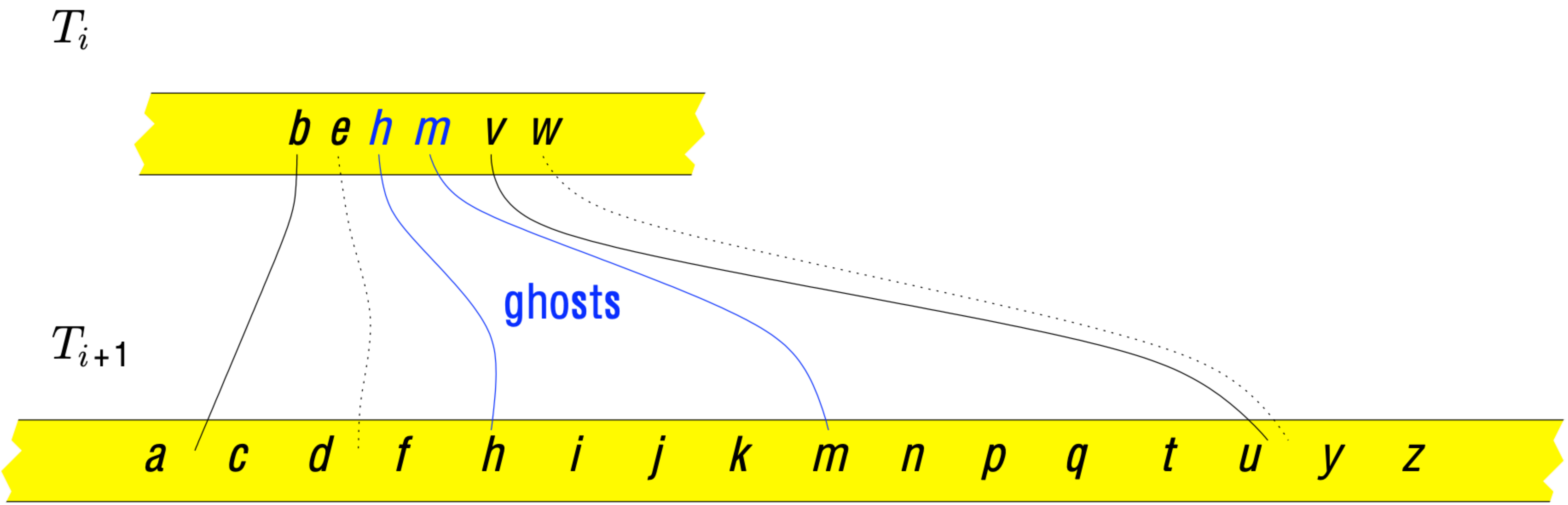
number of trees



From "Cache-oblivious streaming B-trees." by Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson ’07.

LSM tree + forward + ghost = Cache-oblivious lookahead array (COLA)

This data structure no longer uses the internal nodes of the B-trees, and each of the trees can be **implemented by an array**.



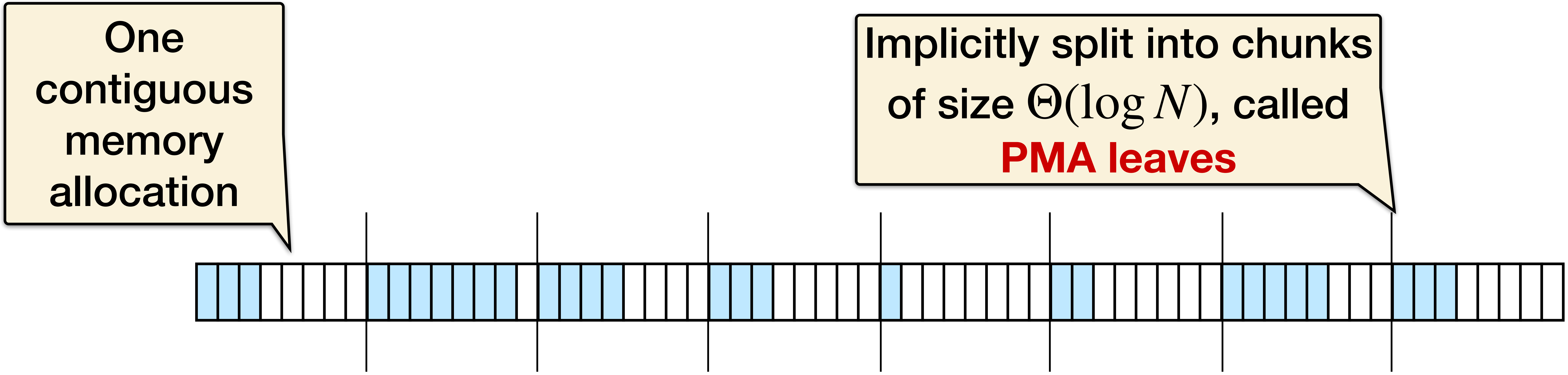
From "Cache-oblivious streaming B-trees." by Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson '07.

Packed Memory Array

Packed Memory Array

The Packed Memory Array (PMA) [ItaiKaRo81, BenderDeFa00] is a **cache-oblivious** ordered dictionary data structure that stores elements in a **contiguous array** with (a constant factor of) spaces for updatability.

That is, the PMA stores N elements in m = $\Theta(N)$ cells.

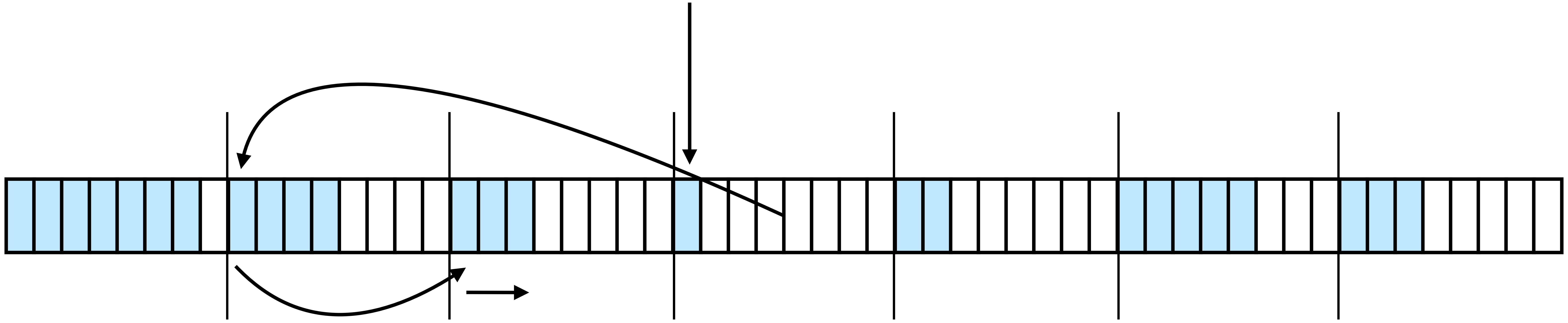


Searching in a PMA

Searching a PMA involves a **binary search** on the first element of each PMA leaf.

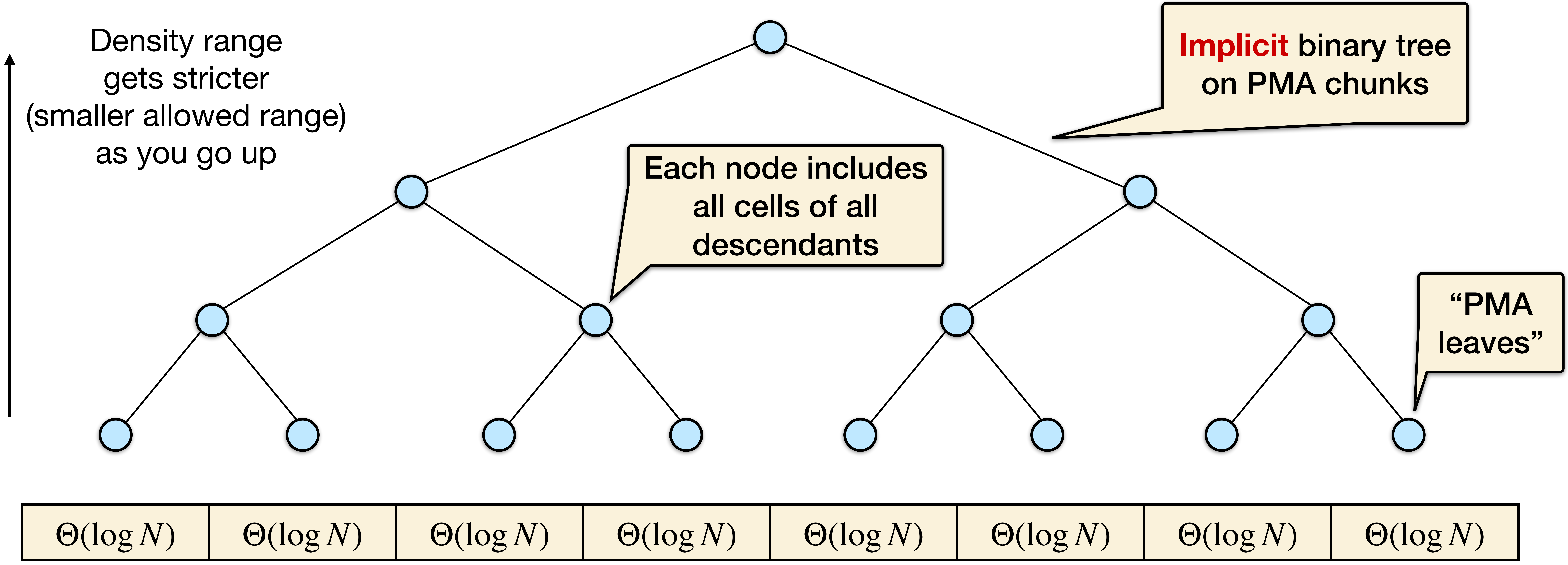
Once you reach the correct leaf, perform a **linear pass** through the chunk to look for the element.

The search costs $O(\log(N/\log(N)) + \log(N)/B) = O(\log(N)$ cache misses.



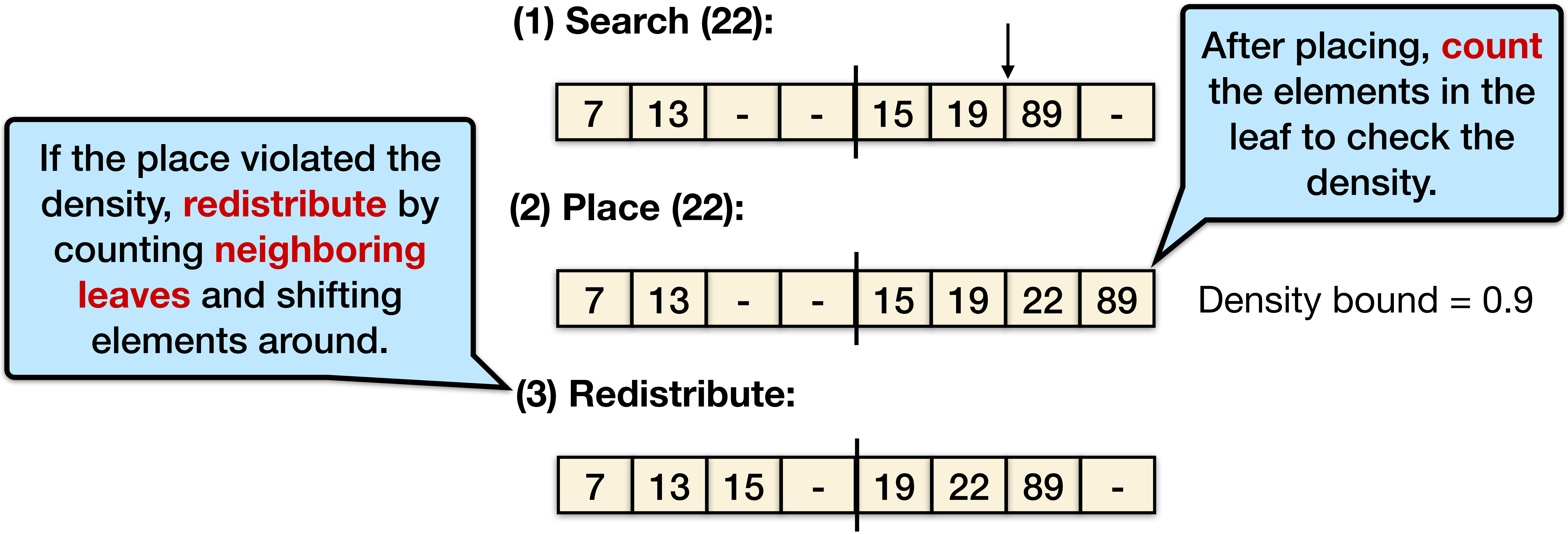
PMA Structure

The PMA maintains empty spaces according to density bounds, where the density is the **ratio of filled cells to total cells** per contiguous region.



Insert in a PMA

The PMA maintains density bounds during updates by **redistributing** elements after each update.



The insert cost of a particular element in a PMA **depends on the input distribution** and the state of the PMA.

PMA Asymptotic Guarantees

Worst-case
amortized

	Insert	Search	Scan
PMA	$O((\log N^2)/B + \log N)$ 😡	$O(\log N)$ 😡	$O(N/B)$ 💚
B-tree	$O(\log_B N)$ 💚	$O(\log_B N)$ 💚	$O(N/B)$ 💚

PMA Asymptotic Guarantees

Worst-case amortized - depends on input distribution

	Insert	Search	Scan
PMA	$O((\log^2 N)/B + \log N)$	$O(\log N)$	$O(N/B)$
B-tree	$O(\log_B N)$ 💚	$O(\log_B N)$ 💚	$O(N/B)$ 💚

Why are we studying PMAs if B-trees are always at least as good (in a big-O sense)?

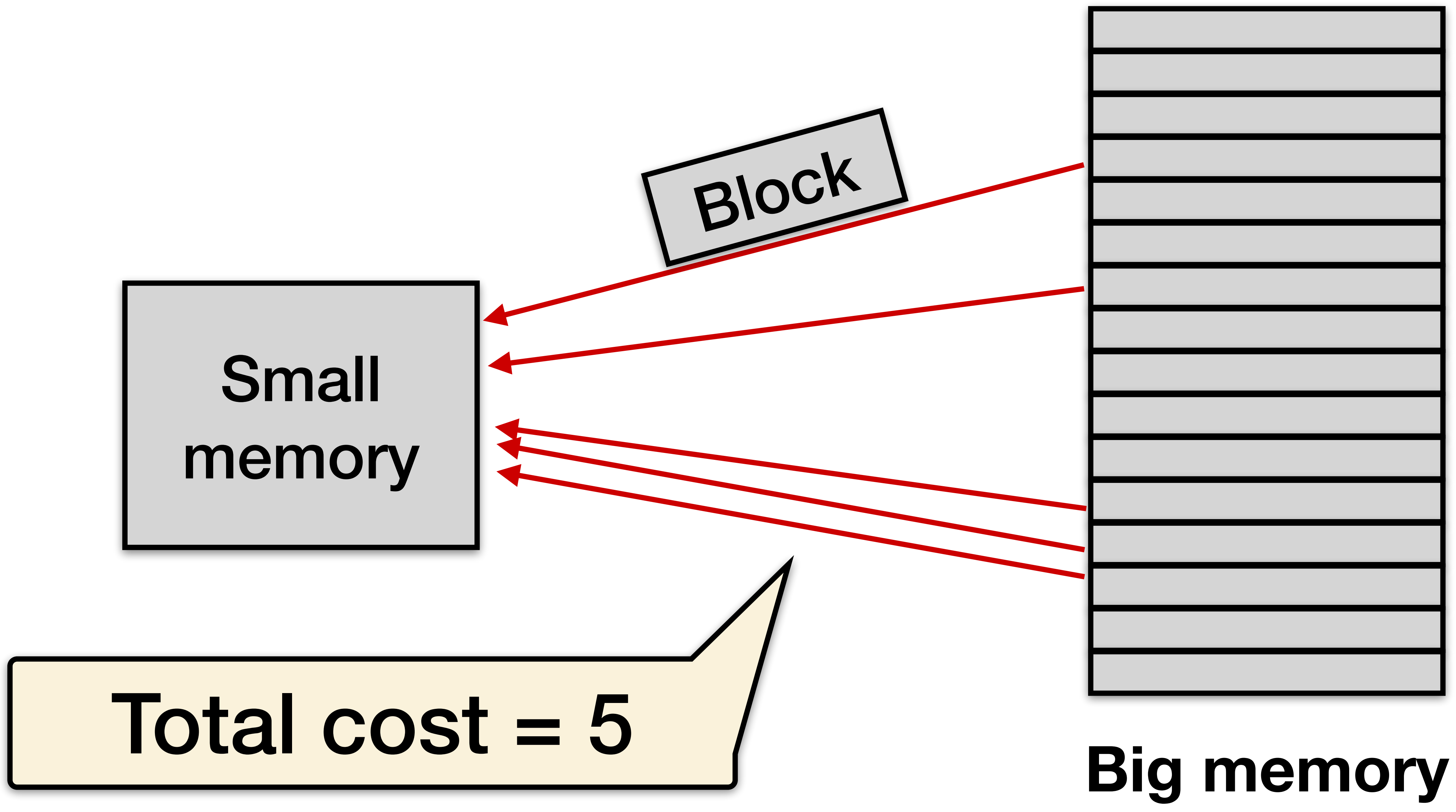
Affine Model

Cost of access in Disk-Access Model (DAM)

Similar to Ideal-Cache model, without tall-cache assumption

The DAM [Aggarwal and Vitter, '88] is a classical model that measures disk page access (or cache-line accesses, in RAM).

Each memory block fetch has **unit cost**.

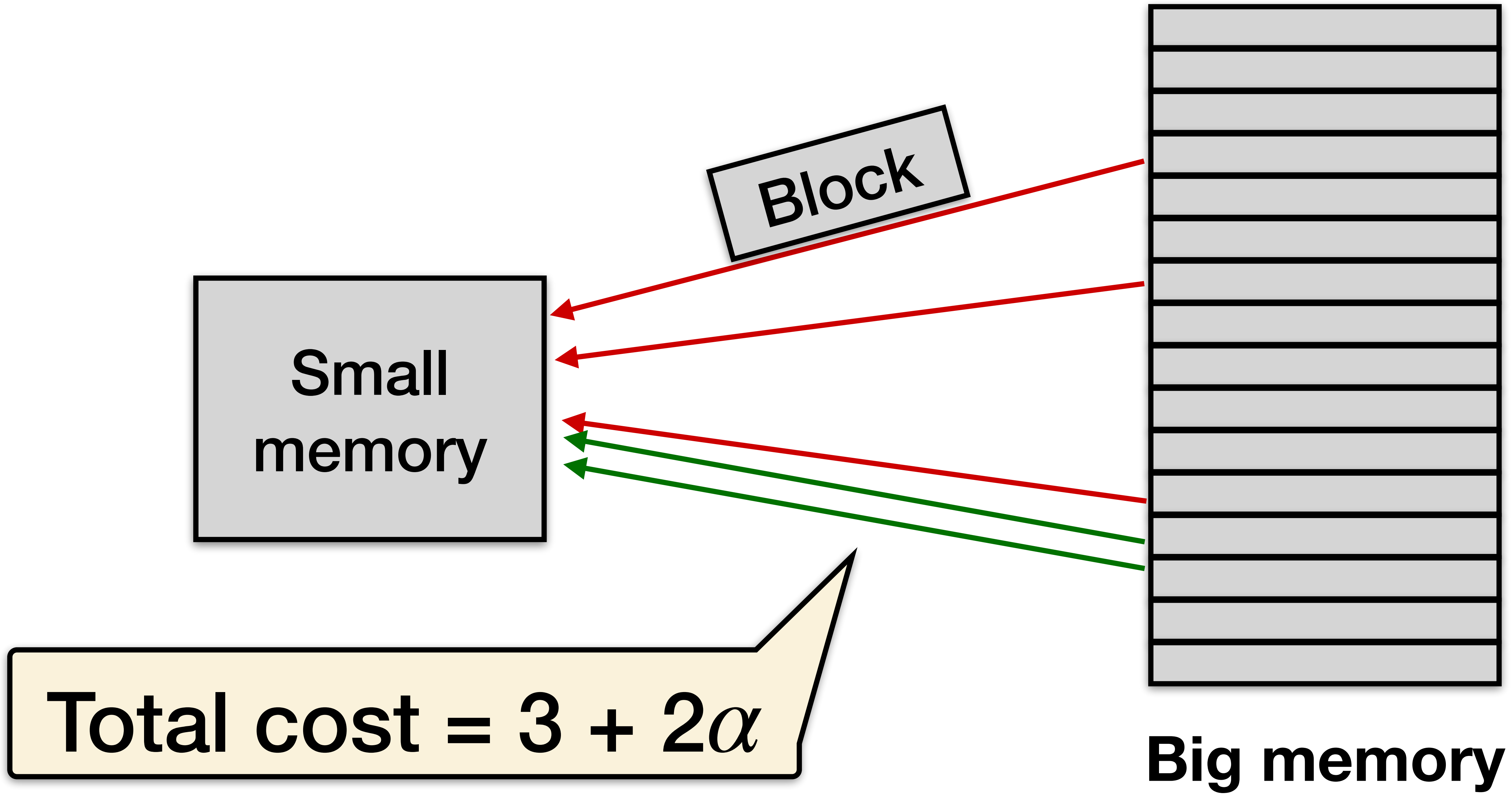


Random vs Sequential Access Cost in the Affine Model

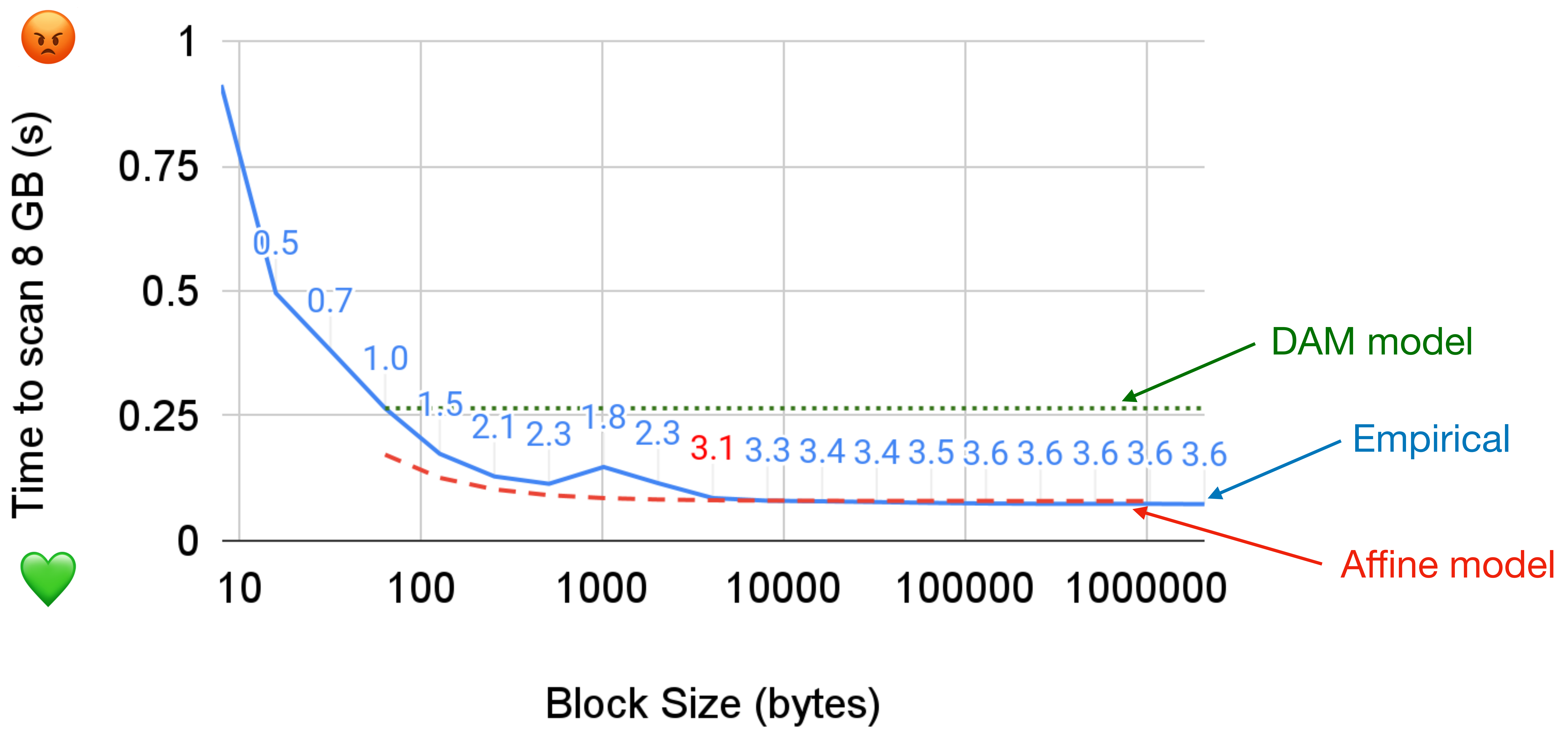
The affine model [ABZ96, BCF+19] accounts for sequential block accesses being faster than random (due to prefetching, etc.).

Random access has unit cost, and **sequential access has cost $\alpha < 1$**.

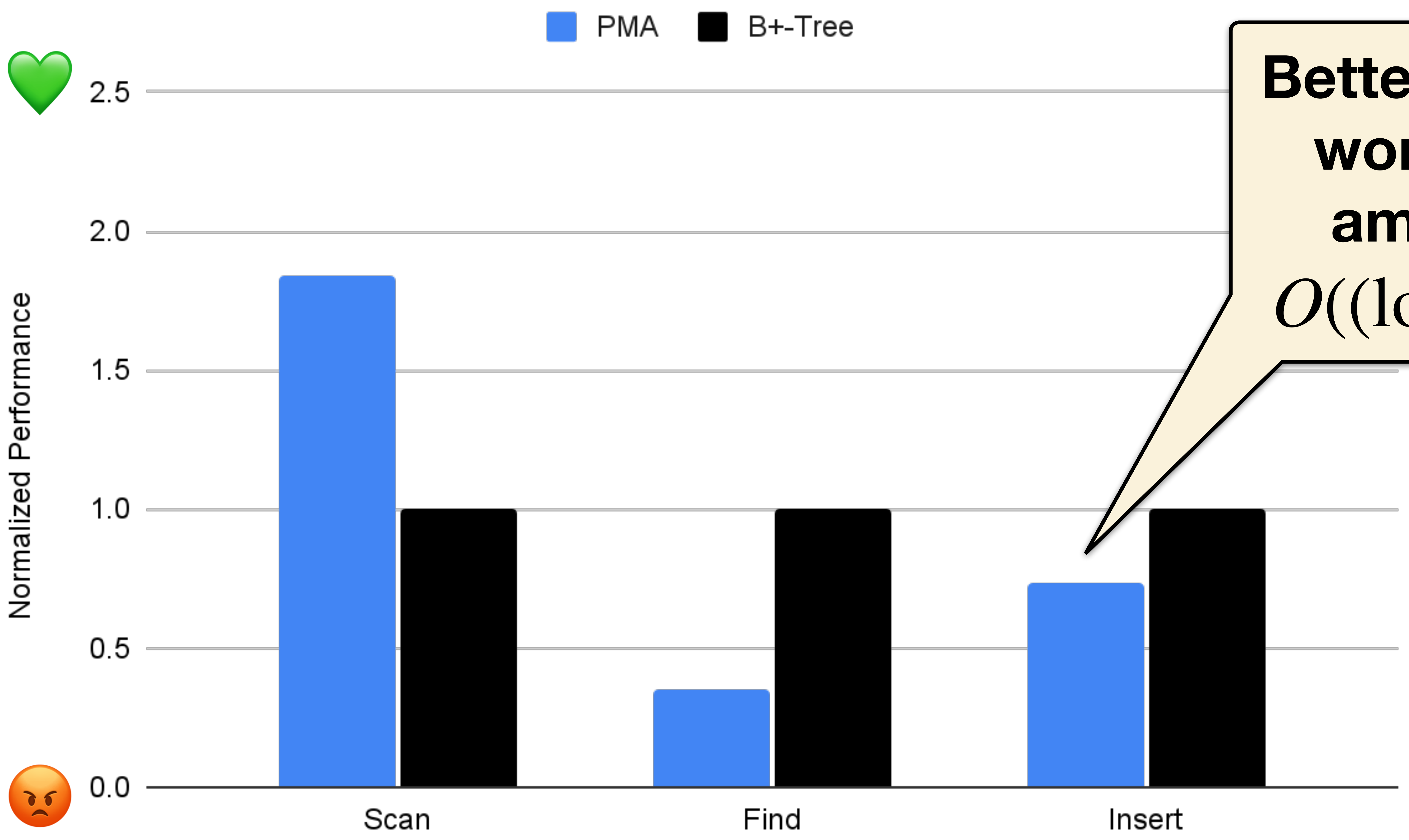
Originally designed for disks and accounted for disk seek vs read.



Empirically validating the affine model in memory



Point-Range Tradeoff



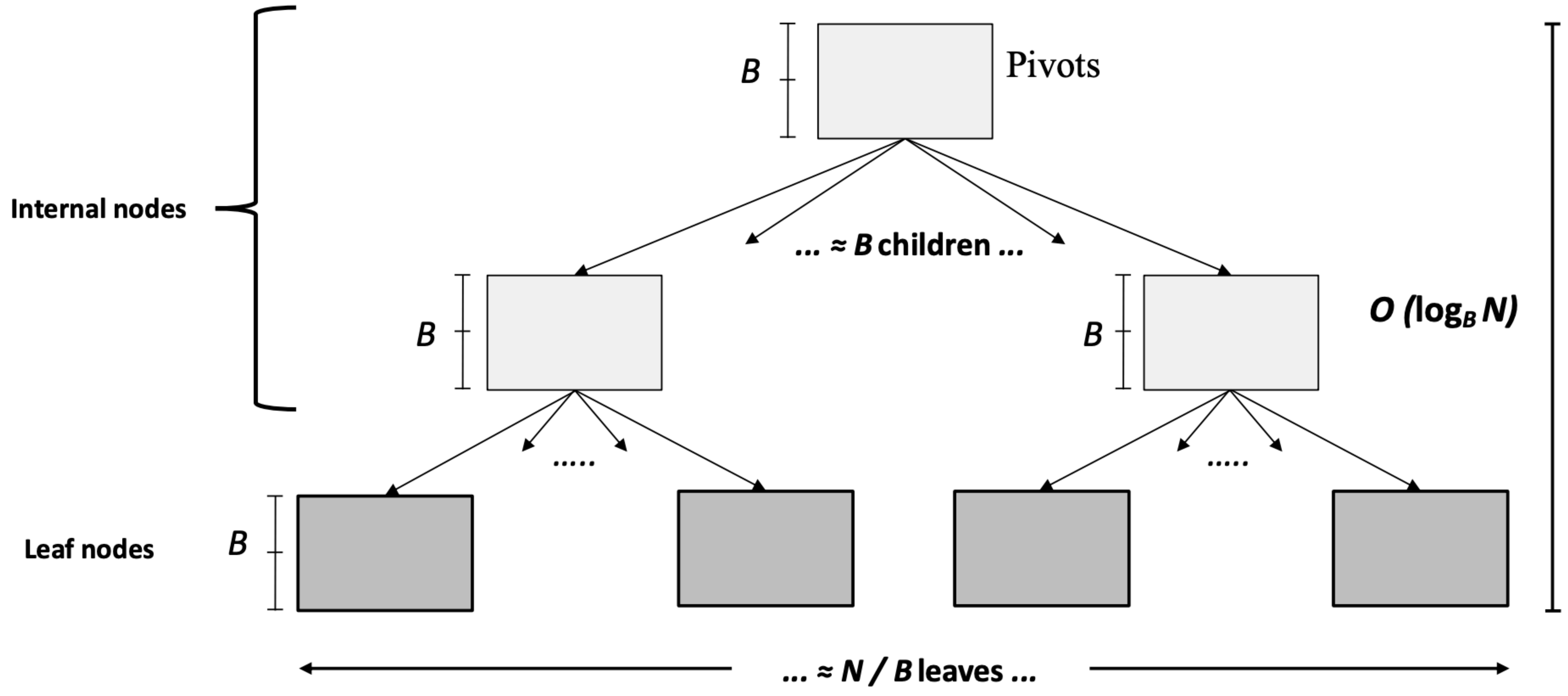
Better than the worst-case amortized $O((\log^2 N)/B)$

Tries

General drawbacks of tree data structures

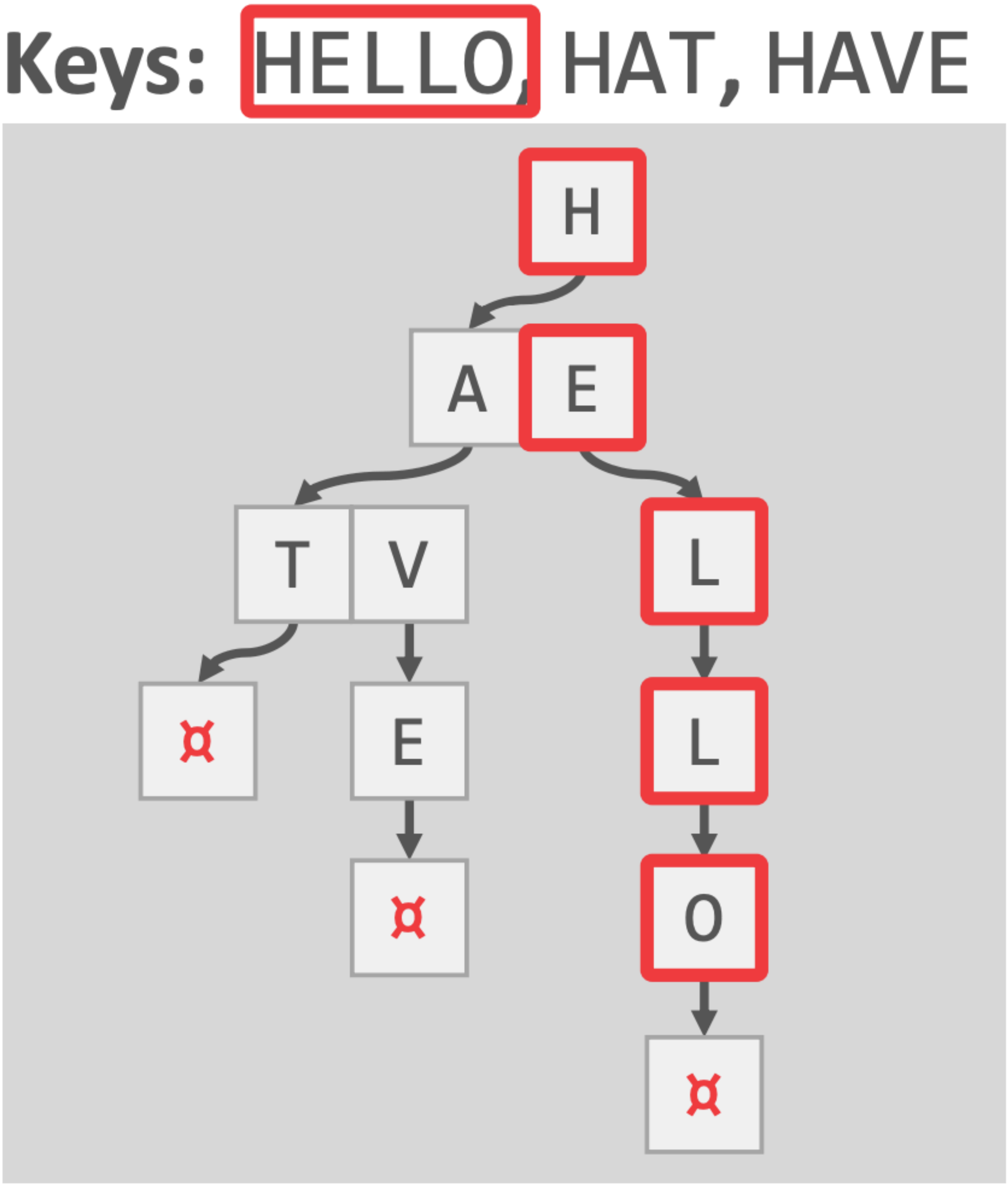
- The inner nodes alone in a B+-tree or buffered B-tree (or LSM, PMA, etc.) cannot tell you whether a key exists in the index. You must always **traverse to the leaf** node (unless you find it earlier).

- This means you could have (at least) **one cache miss** per level in the tree.



Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture03.pdf

Trie index



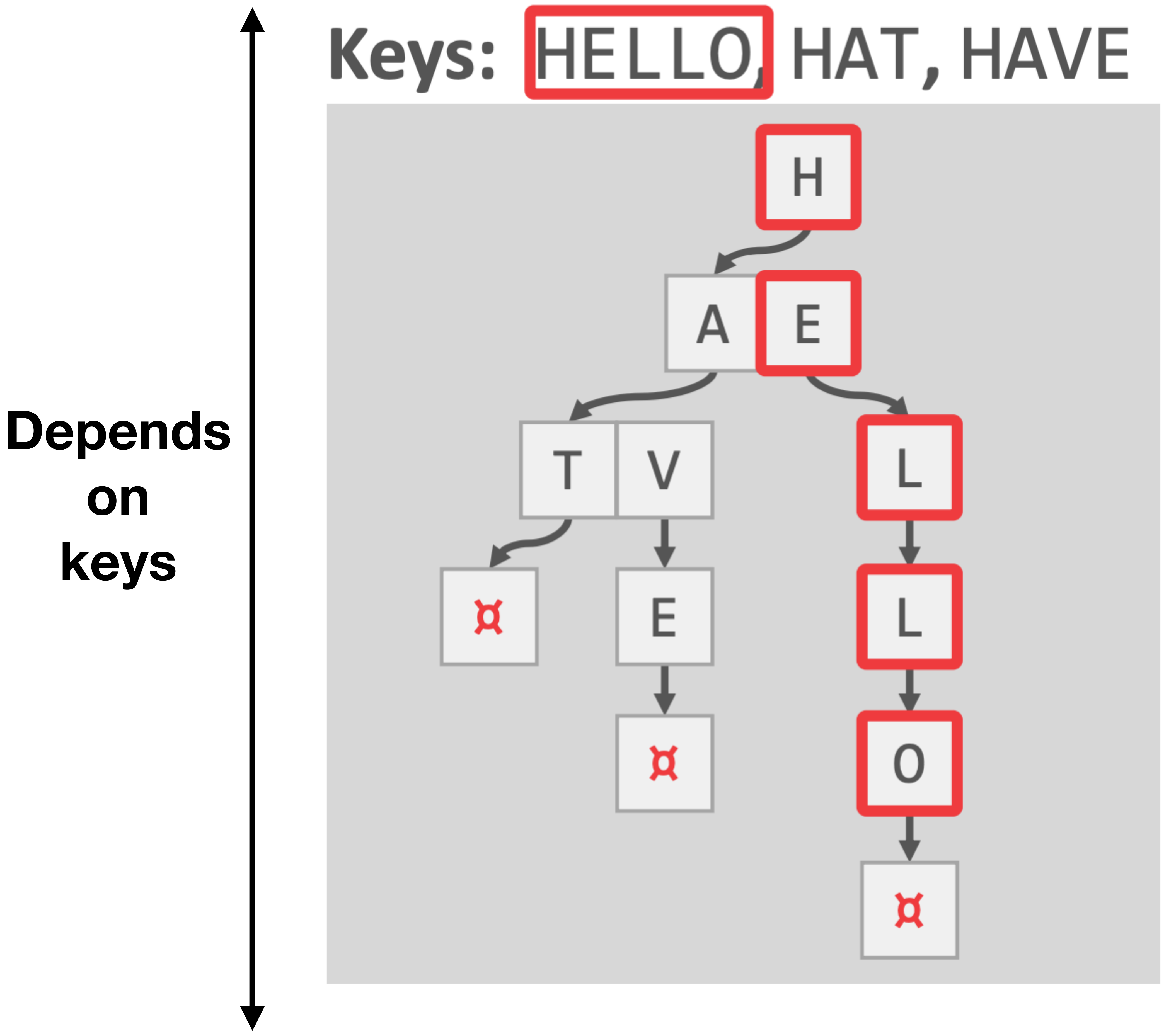
Use a digital representation of keys to **examine prefixes one-by-one** instead of comparing entire key.

Also known as **Digital Search Tree**, **Prefix Tree**.

Used in predictive text / approximate matching algorithms.

Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture03.pdf

Trie index properties



Shape only depends on **key space and lengths**.

- Does not depend on existing keys or insertion order.
- **Does not require rebalancing** operations.

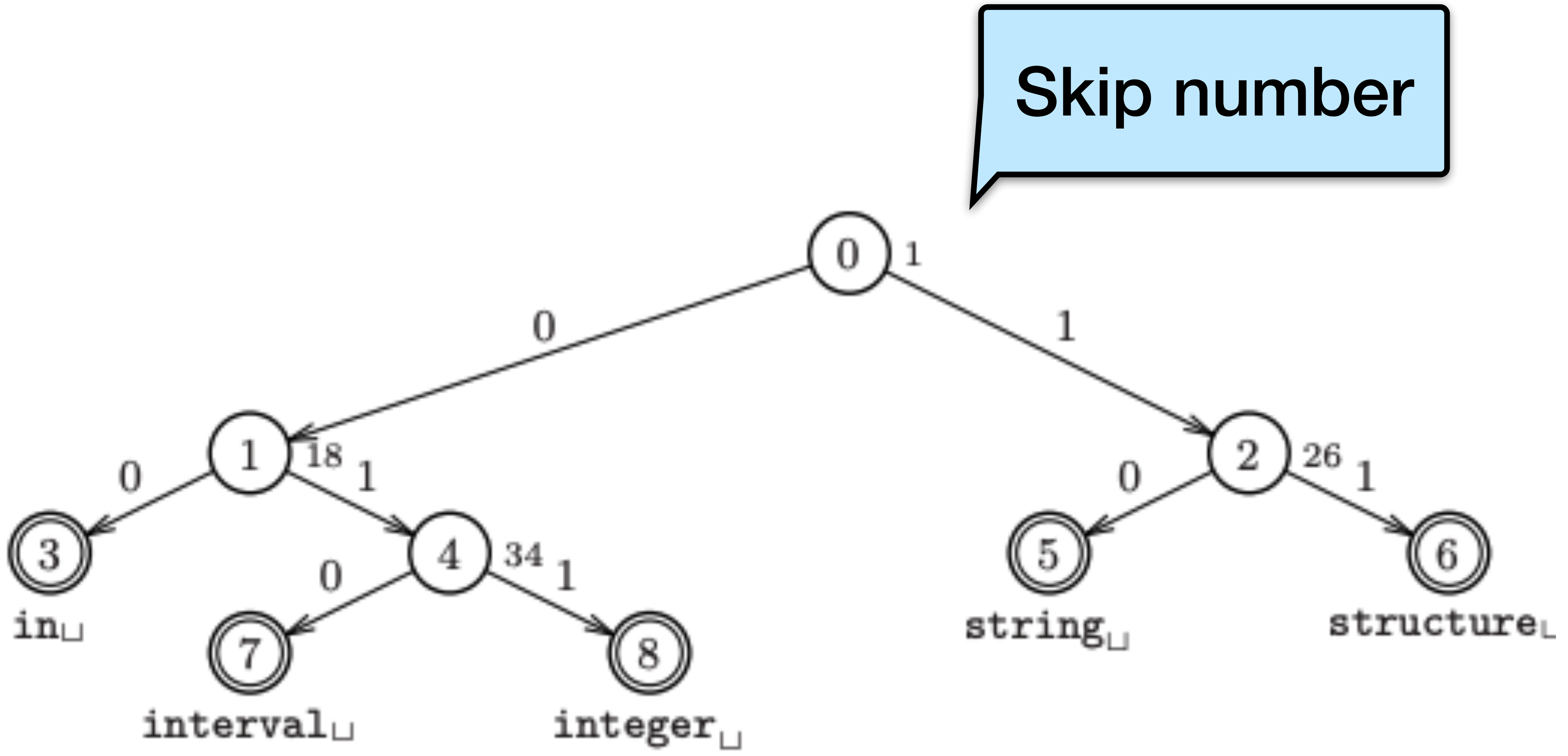
- All operations have **O(k)** complexity where **k** is the length of the key.
  - The path to a leaf node represents the key of the leaf
  - Keys are stored implicitly and can be reconstructed from paths.

Radix tree

More **space efficient** than a naive trie implementation

Uses the **binary encoding of string keys** in its representation.

Every node in a radix tree (or Patricia tree) contains an index, known as a **“skip number”** that stores the node’s branching index to avoid empty subtree traversals.



Decimal and binary ASCII codes of the symbols

		7	6	5	4	3	2	1	0
␣	32	0	0	1	0	0	0	0	0
a	97	0	1	1	0	0	0	0	1
c	99	0	1	1	0	0	0	1	1
e	101	0	1	1	0	0	1	0	1
g	103	0	1	1	0	0	1	1	1
i	105	0	1	1	0	1	0	0	1
l	108	0	1	1	0	1	1	0	0
n	110	0	1	1	0	1	1	1	0
r	114	0	1	1	1	0	0	1	0
s	115	0	1	1	1	0	0	1	1
t	116	0	1	1	1	0	1	0	0
u	117	0	1	1	1	0	1	0	1
v	118	0	1	1	1	0	1	1	0

Strings of X as sequences of bits

	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31	32 33 34 35
in␣	1 **0** 0 1 0 1 1 0	0 1 1 1 0 1 1 0	0 0 **0** 0 0 1 0 0		
integer␣	1 **0** 0 1 0 1 1 0	0 1 1 1 0 1 1 0	0 0 **1** 0 1 1 1 0	1 0 1 0 0 1 1 0	0 1 **1** 0 ···
interval␣	1 **0** 0 1 0 1 1 0	0 1 1 1 0 1 1 0	0 0 **1** 0 1 1 1 0	1 0 1 0 0 1 1 0	0 1 **0** 0 ···
string␣	1 **1** 0 0 1 1 1 0	0 0 1 0 1 1 1 0	0 1 0 0 1 1 1 0	1 0 **0** 1 0 1 1 0	0 1 1 1 ···
structure␣	1 **1** 0 0 1 1 1 0	0 0 1 0 1 1 1 0	0 1 0 0 1 1 1 0	1 0 **1** 0 1 1 1 0	1 1 0 0 ···

https://en.wikipedia.org/wiki/Trie

Trie variants

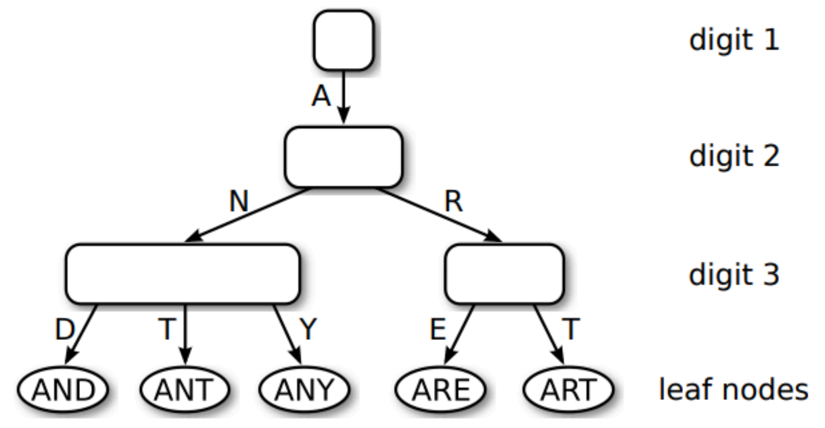
Judy arrays (HP)

- Variant of a 256-way radix tree. First known radix tree that supports adaptive node representation.

https://judy.sourceforge.net/

ART Index (HyPER)

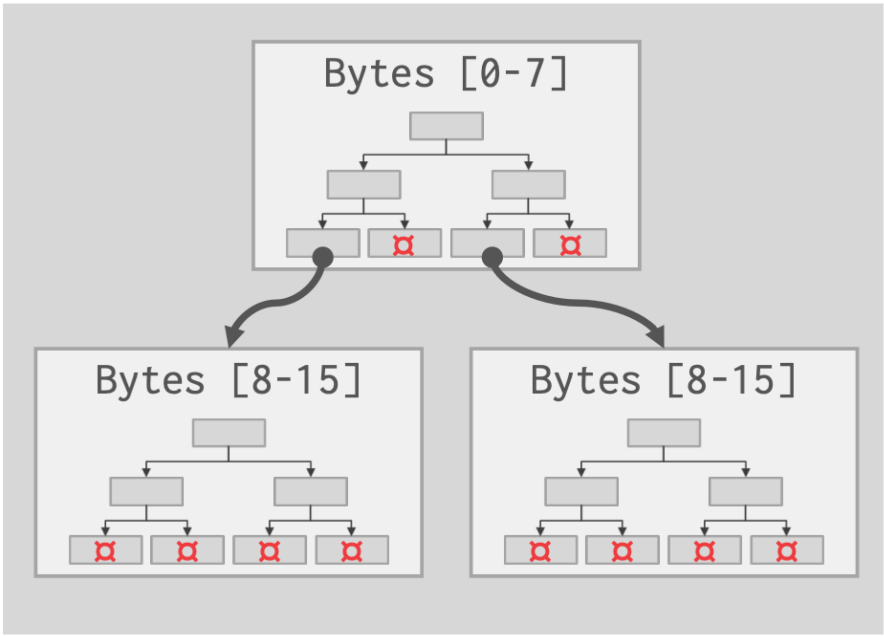
- 256-way radix tree that supports different node types based on its population.



https://db.in.tum.de/~leis/papers/ART.pdf

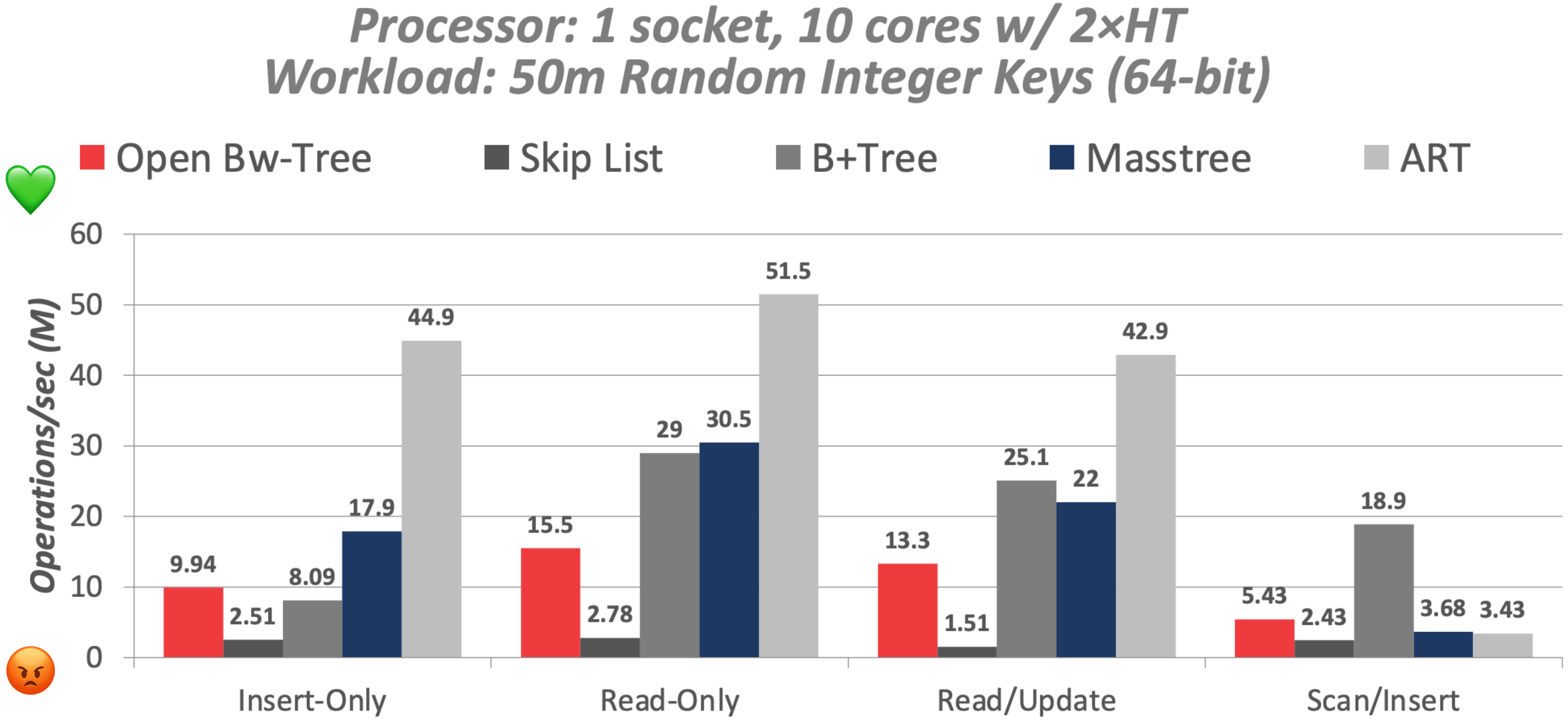
Masstree (Silo)

- Instead of using different layouts for each trie node based on size, use an entire B+-tree.



https://pdos.csail.mit.edu/papers/masstree:eurosys12.pdf

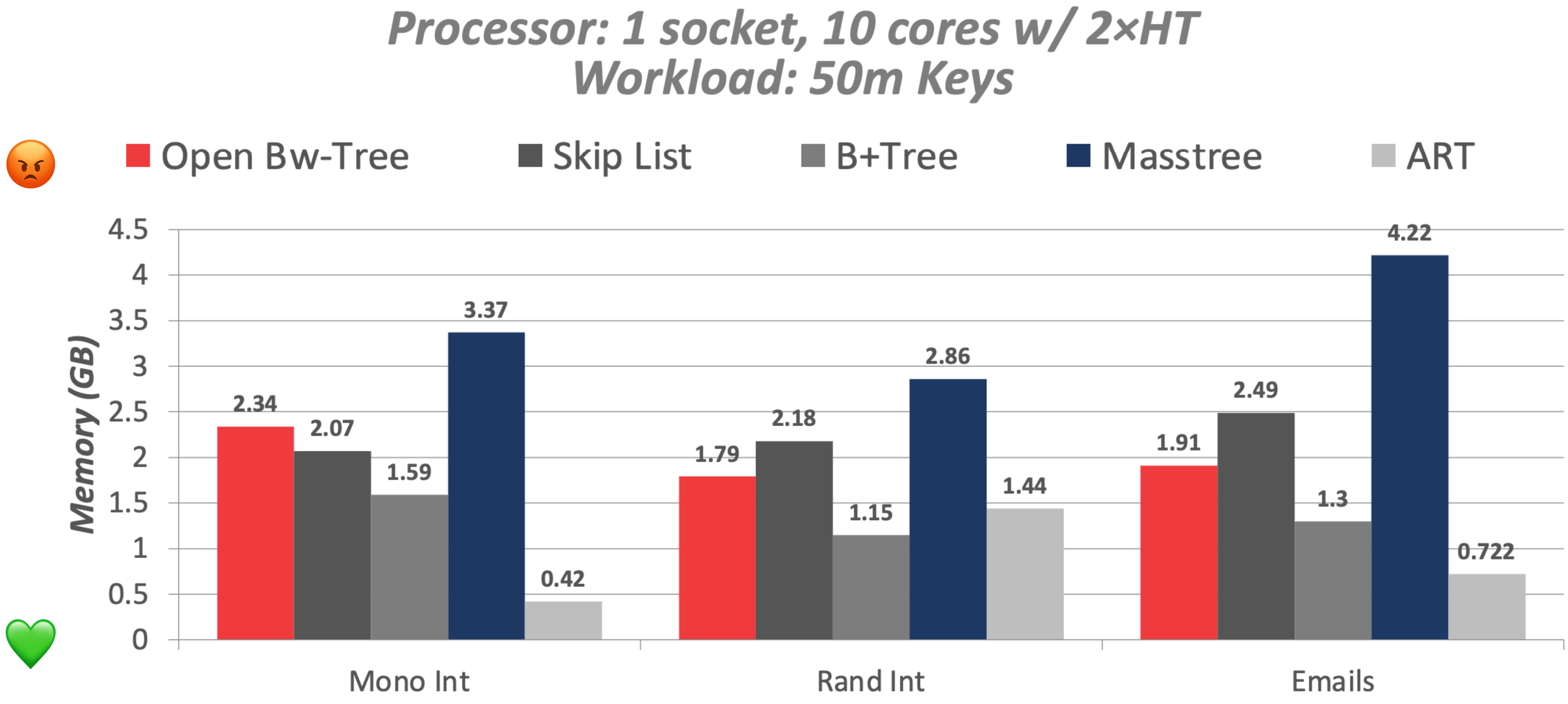
Performance comparison



Source: https://github.com/wangziqi2016/index-microbench

Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture03.pdf

Size comparison



Source: https://github.com/wangziqi2016/index-microbench
Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture03.pdf

Summary

- **B+-trees are the go-to** in-memory indexing data structure.

- B^{ε} trees achieve **asymptotically faster insert** without theoretical loss in search performance, but are **harder to implement** and have worse write amplification.

- Skip lists are great if you don't want to implement self-balancing algorithms.

Slides from: https://users.cs.utah.edu/~pandey/courses/cs6530/fall23/slides/Lecture03.pdf