CACHE-OBLIVIOUS MAPS

Edward Kmett
McGraw Hill Financial

CACHE-OBLIVIOUS MAPS

Edward Kmett
McGraw Hill Financial

CACHE-OBLIVIOUS MAPS

- Indexing and Machine Models
- Cache-Oblivious Lookahead Arrays
- Amortization of Functional Data Structures
- Performance in Practice

MOTIVATION

- I need a Map that has support for very efficient range queries
- It also needs to support very efficient writes
- But I can let point query performance suffer
- It needs to support unboxed data
- ...and I don't want to give up all the conveniences of Haskell

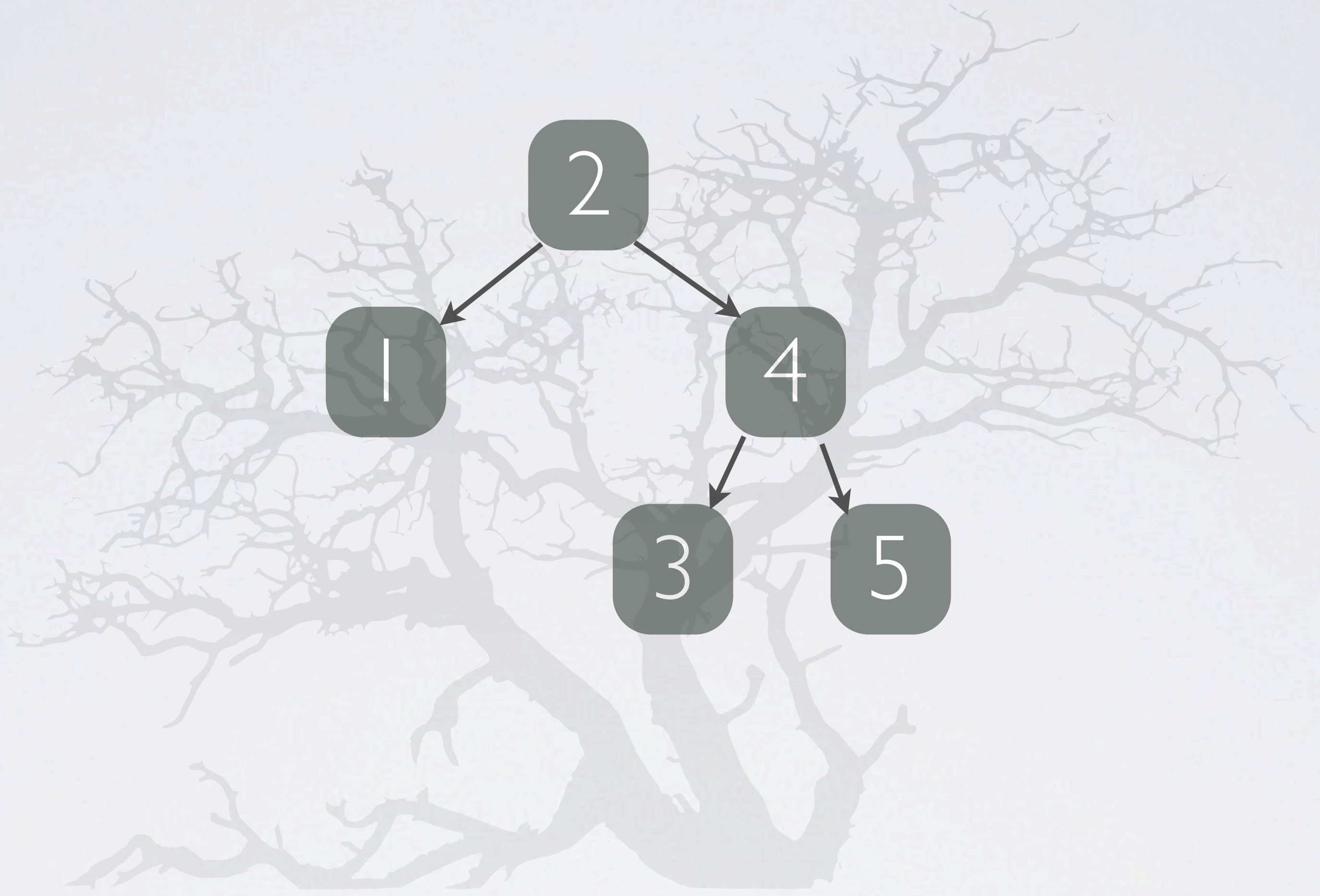
MACHINE MODELS

- Turing Machine
- Random-Access Machine / Word Model
- External-Memory Model
- Cache-Oblivious Model

RAM MODEL

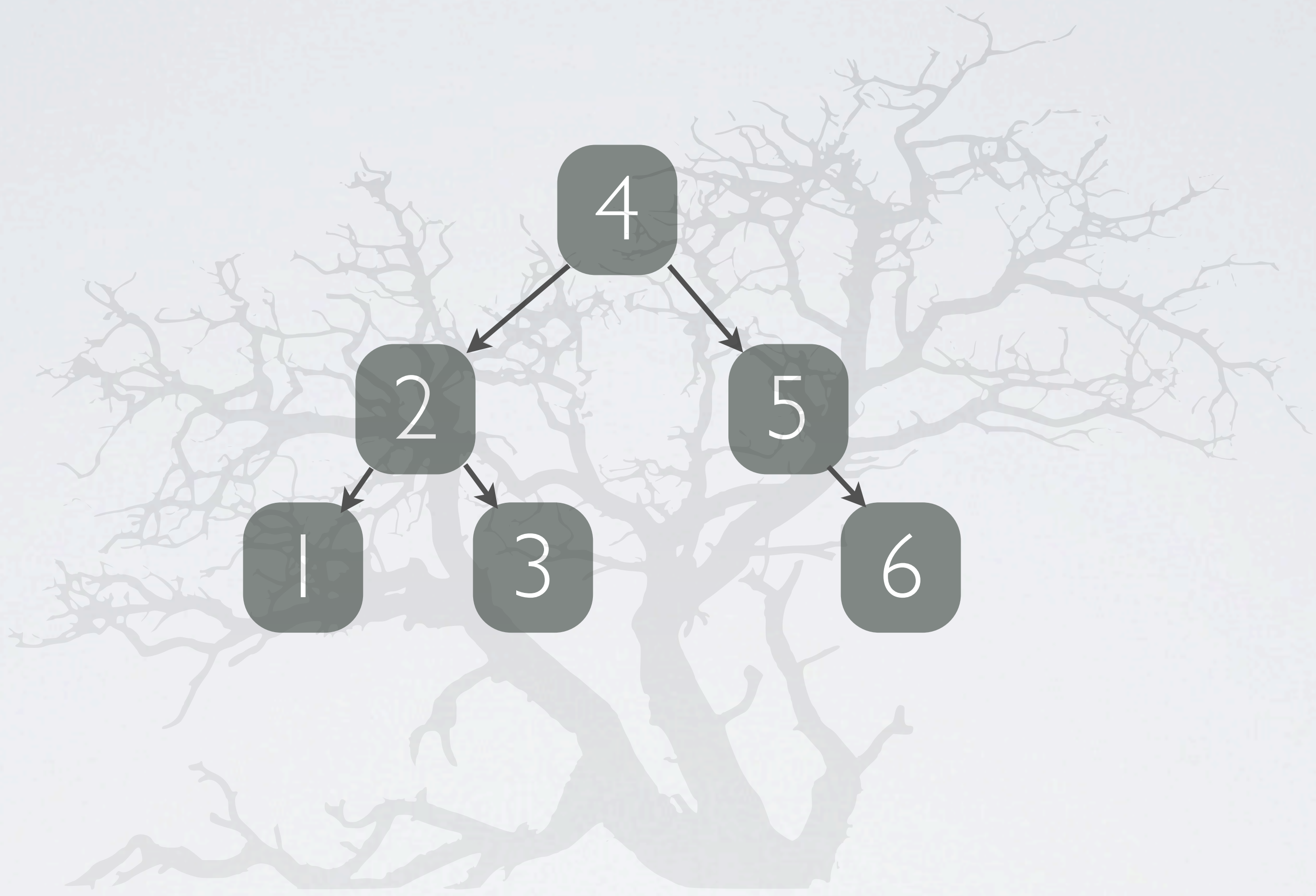
- Almost everything you do in Haskell assumes this model
- Good for ADTs, but not a realistic model of today's hardware

DATA.MAP



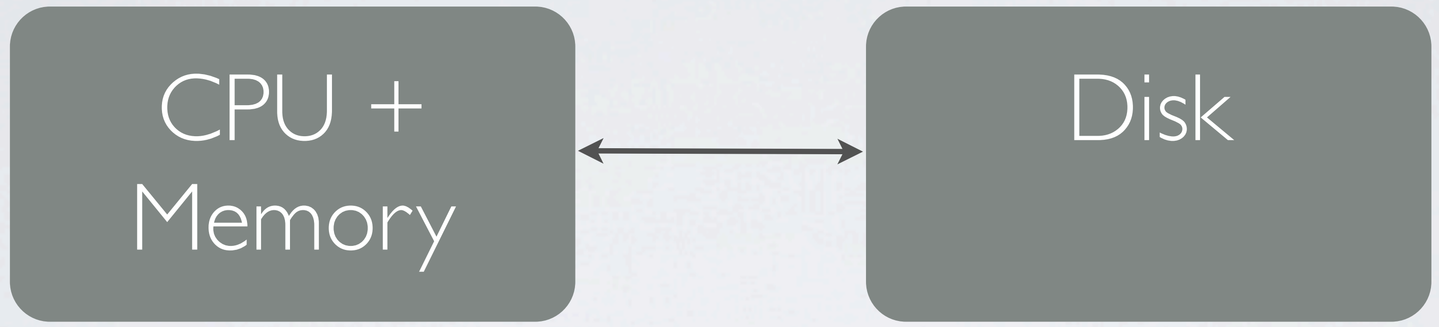
"Binary search trees of bounded balance"

DATA.MAP

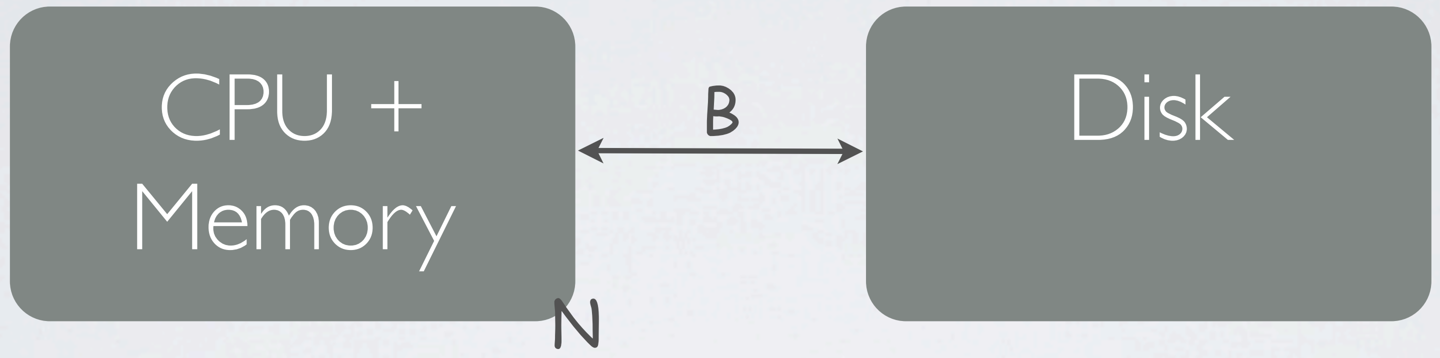

$$\begin{tikzcd}
 & & 4 \arrow[dl] \arrow[dr] & \\
 & 2 \arrow[dl] \arrow[dr] & & 5 \arrow[dr] \\
1 & & 3 & & 6
\end{tikzcd}$$

"Binary search trees of bounded balance"

EXTERNAL-MEMORY MODEL

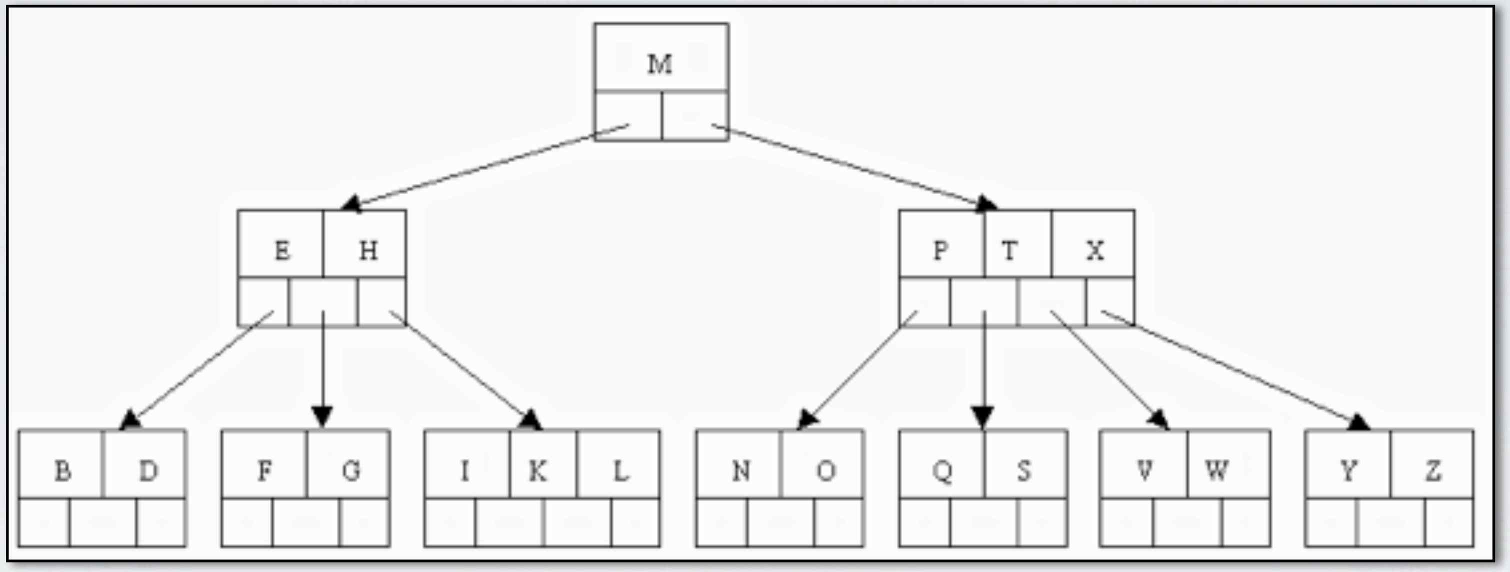


EXTERNAL-MEMORY MODEL



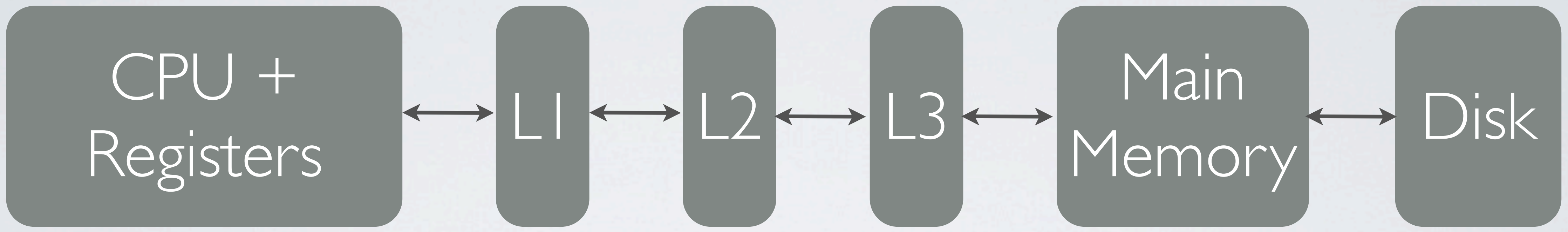
- Can Read/Write Contiguous Blocks of Size **B**
- Can Hold **M/B** blocks in working memory
- All other operations are "Free"

B-TREES

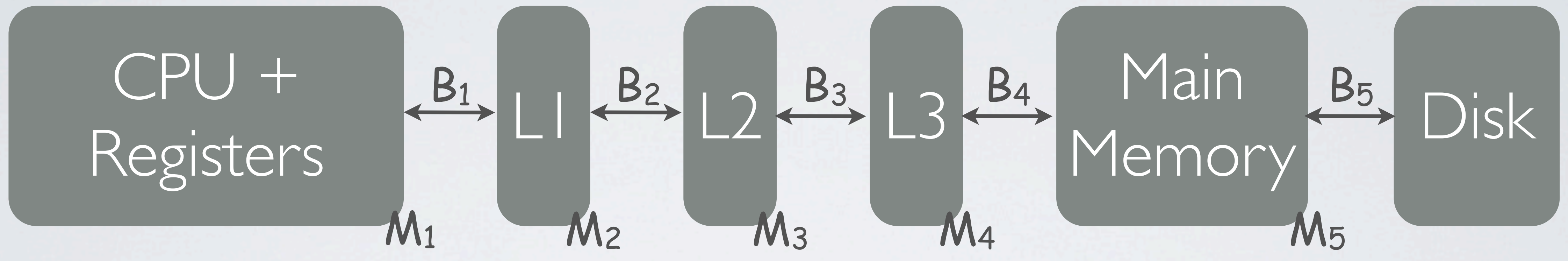


- Occupies **O(N/B)** blocks worth of space
- Update in time **O(log(N/B))**
- Search **O(log(N/B) + a/B)** where **a** is the result set size

EXTERNAL-MEMORY MODEL

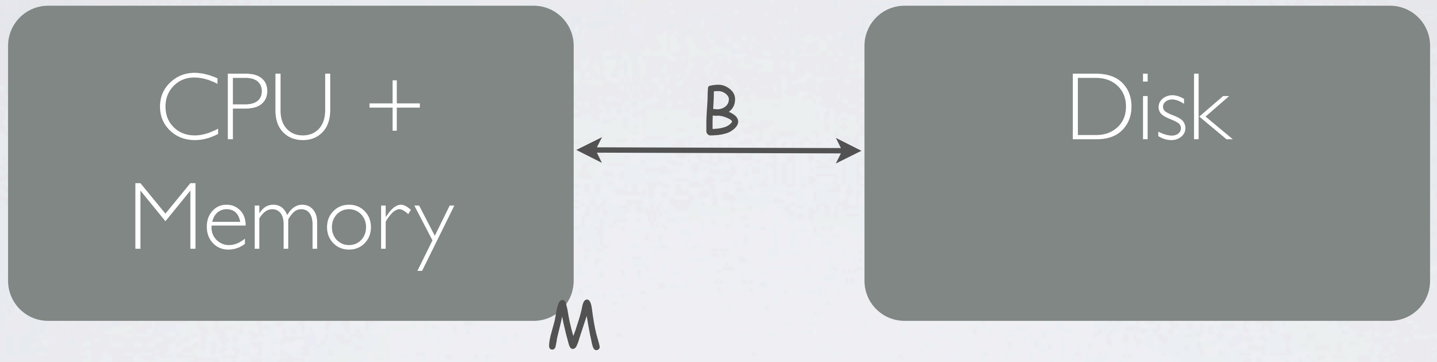


EXTERNAL-MEMORY MODEL



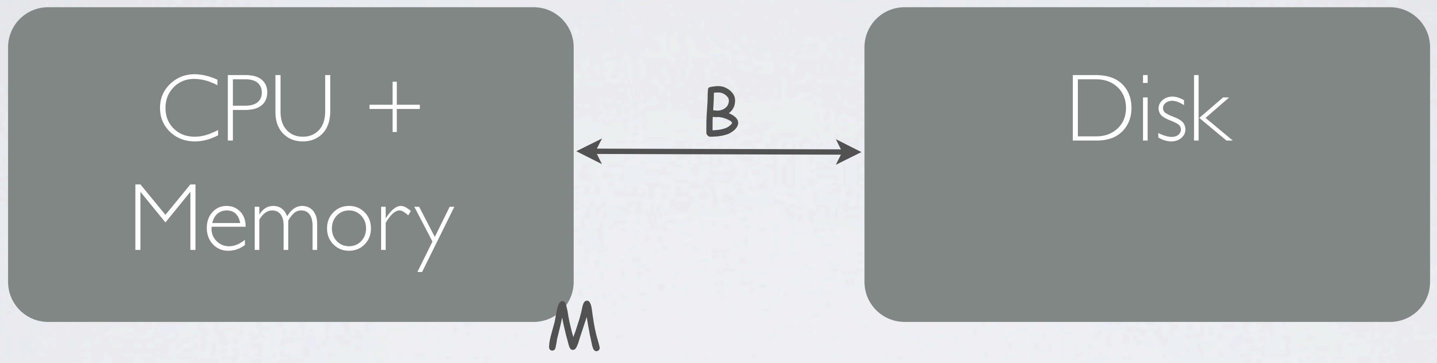
- Huge numbers of constants to tune
- Optimizing for one necessarily sub-optimizes others
- Caches grows exponentially in size and slowness

CACHE-OBLIVIOUS MODEL



- Can Read/Write Contiguous Blocks of Size **B**
- Can Hold **M/B** Blocks in working memory
- All other operations are "Free"
- But now you don't get to know **M** or **B**!
- Various refinements exist *e.g.* the tall cache assumption

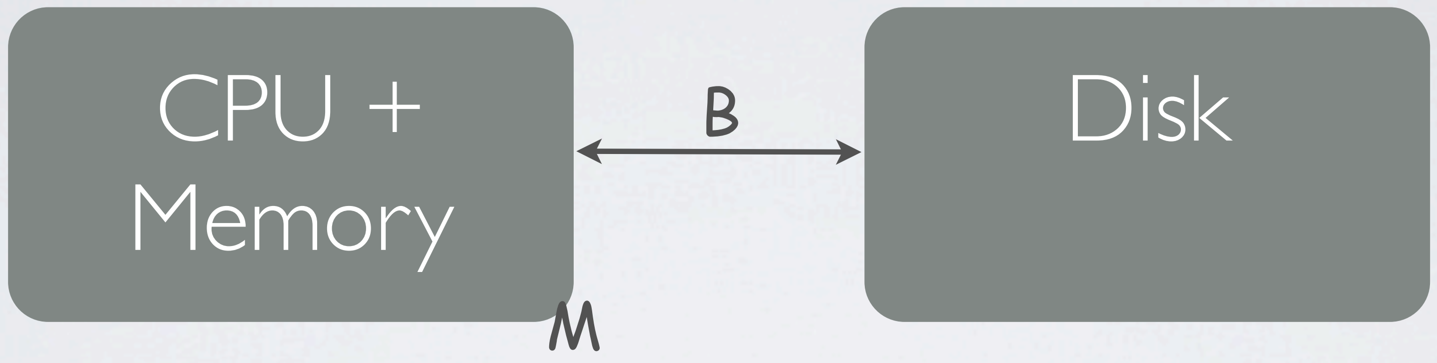
CACHE-OBLIVIOUS MODEL



- If your algorithm is asymptotically optimal for an unknown cache with an optimal replacement policy it is *asymptotically* optimal for *all* caches at the same time.

- You can relax the assumption of optimal replacement and model LRU, **k**-way set associative caches, and the like via caches by modest reductions in **M**.

CACHE-OBLIVIOUS MODEL



- As caches grow taller and more complex it becomes harder to tune for them at the same time. Tuning for one provably renders you suboptimal for others.
- The overhead of this model is largely compensated for by ease of portability and vastly reduced tuning.
- This model is becoming more and more true over time!

THE GOAL

- Provide an API that a Haskell programmer would want to use!
- Beat **Data.Map, Data.IntMap,** and **Data.HashMap** *for range queries and inserts*
- Get **B**-Tree performance without tuning parameters

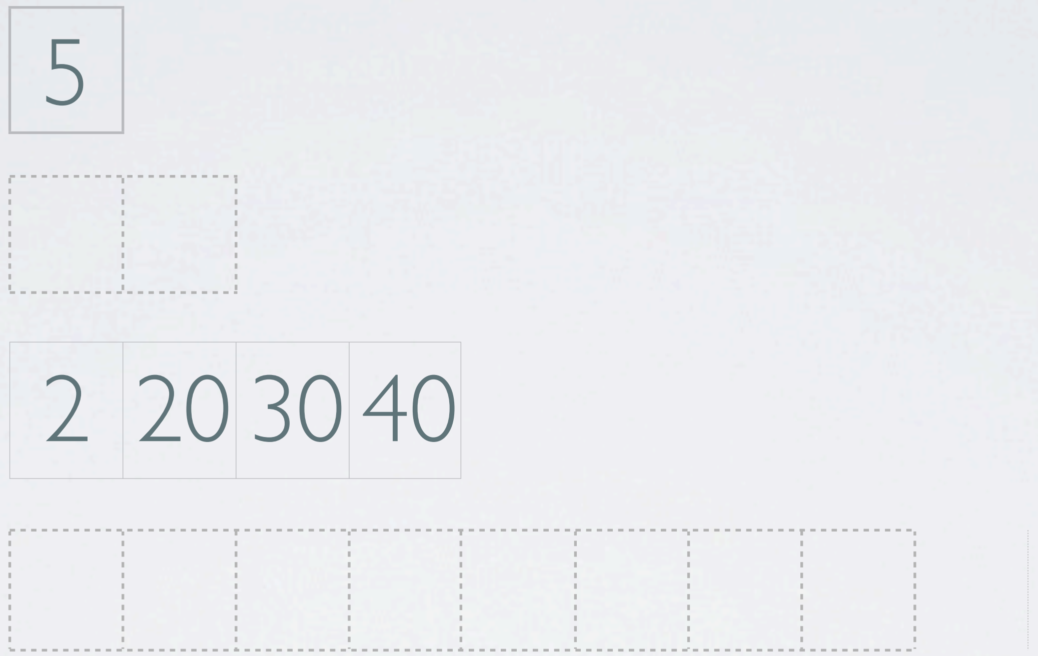
CACHE-OBLIVIOUS LOOKAHEAD ARRAYS (COLA)

- Designed by *Bender et al.*
- Obliviously matches worst-case performance for a **B**-Tree
- Very fast inserts
- Fast range queries
- **Not** a purely functional data structure

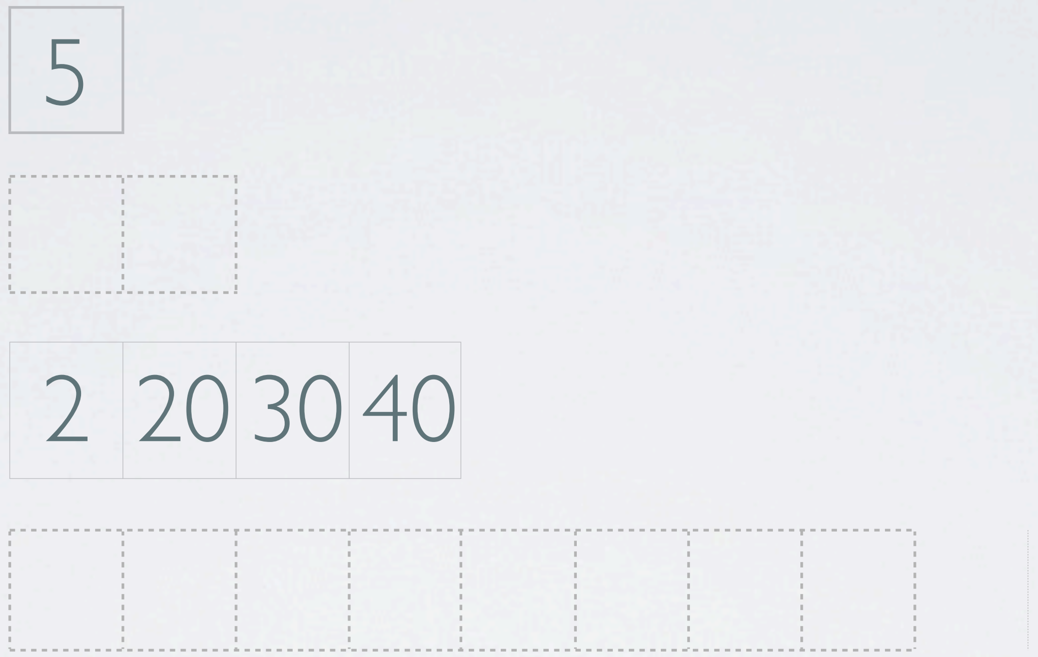
A SIMPLIFIED COLA

- Linked list of sorted arrays.
- Each a power of 2 in size.
- The list is sorted strictly monotonically by size.
- Bigger vectors are later in the list.

A SIMPLIFIED COLA

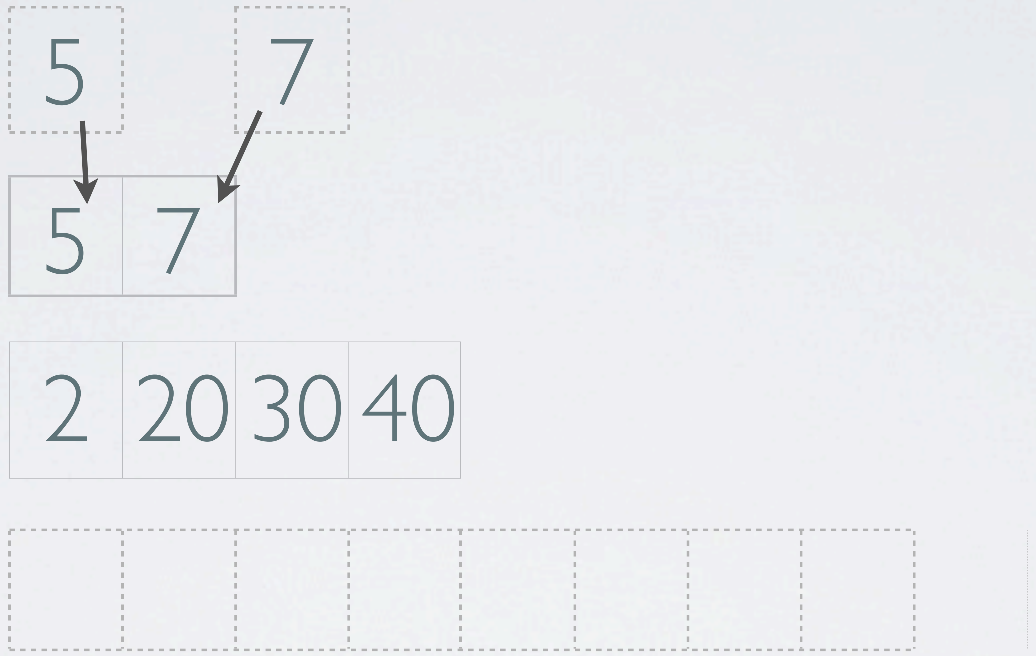


A SIMPLIFIED COLA

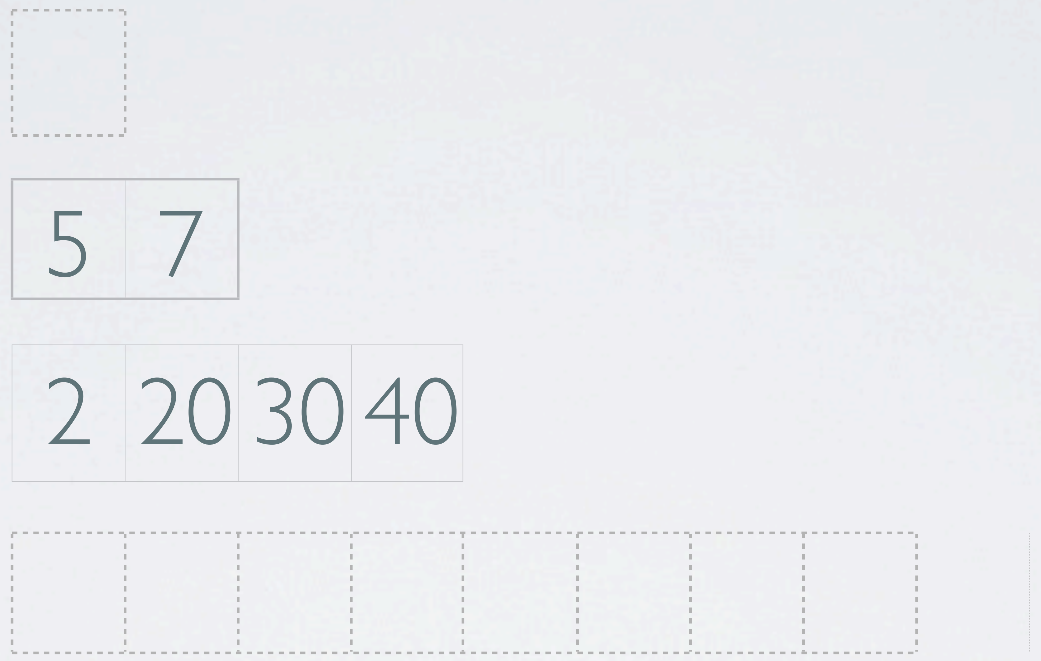


Now let's insert 7

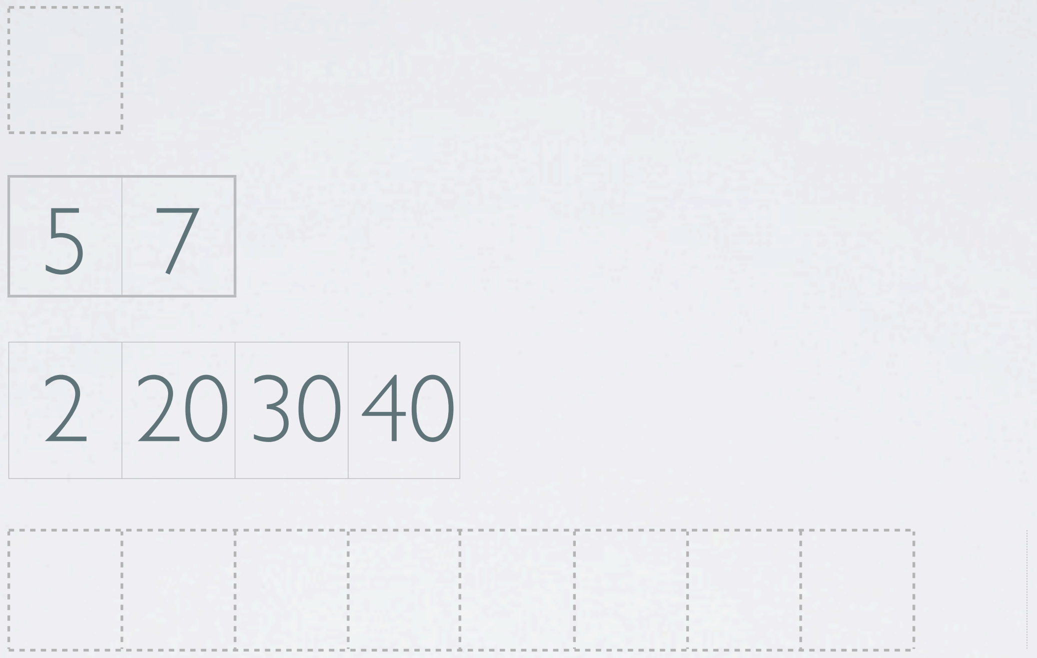
A SIMPLIFIED COLA



A SIMPLIFIED COLA

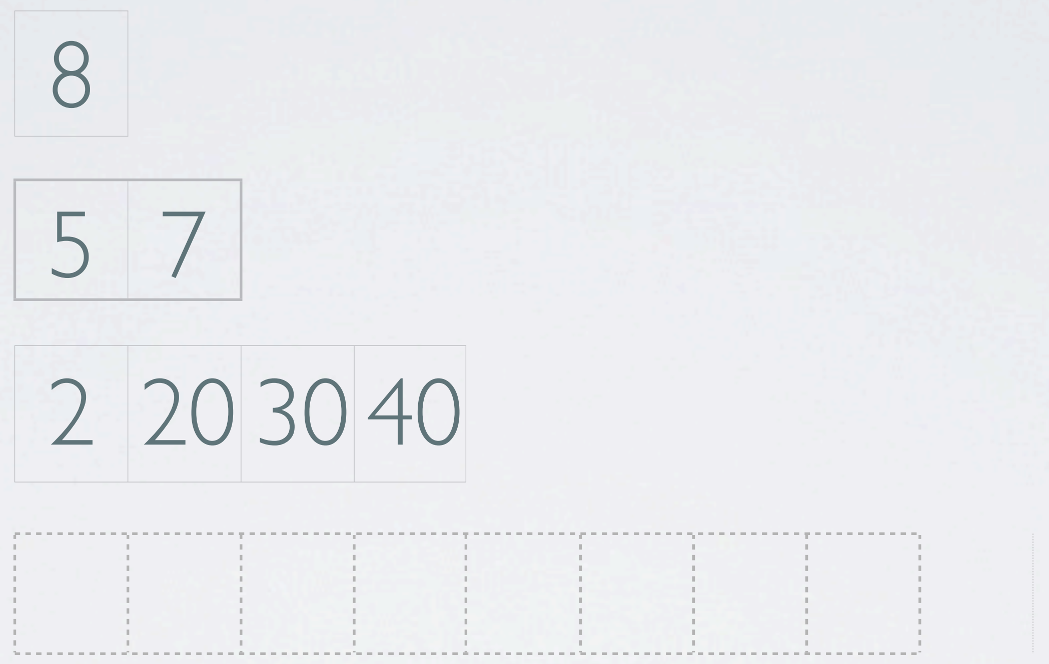


A SIMPLIFIED COLA

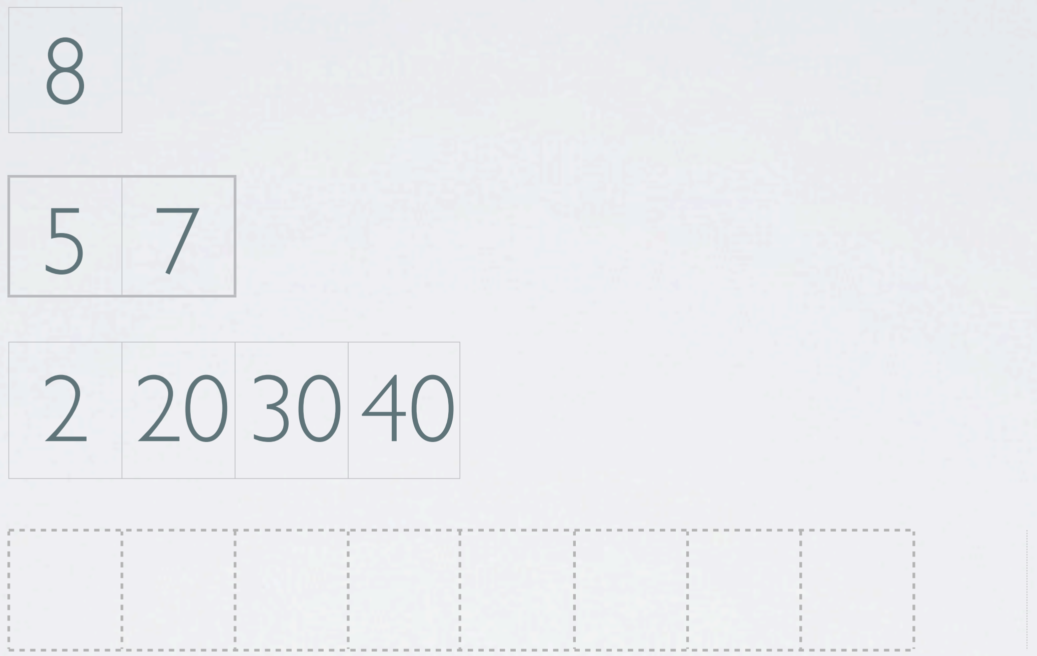


Now let's insert 8

A SIMPLIFIED COLA



A SIMPLIFIED COLA



Next insert causes a cascade of carries!

Worst-case insert time is **O(N/B)**

Amortized insert time is **O((log N)/B)**

AMORTIZATION

Given a sequence of **n** operations:

$$a_1, a_2, a_3 \,..\, a_n$$

What is the running time of the whole sequence?

$$\forall k \leq n.\ \sum_{i=1}^{k} \text{actual}_i \leq \sum_{i=1}^{k} \text{amortized}_i$$

There are algorithms for which the amortized bound is provably better than the achievable worst-case bound *e.g.* Union-Find

BANKER'S METHOD

- Assign a price to each operation.
- Store savings/borrowings in state around the data structure
- If no account has any debt, then

$$\forall k \leq n.\ \sum_{i=1}^{k} \text{actual}_i \leq \sum_{i=1}^{k} \text{amortized}_i$$

PHYSICIST'S METHOD

- Start from savings and derive costs per operation
- Assign a "potential" Φ to each state in the data structure
- The amortized cost is actual cost plus the change in potential.

$$\text{amortized}_i = \text{actual}_i + \Phi_i - \Phi_{i-1}$$

$$\text{actual}_i = \text{amortized}_i + \Phi_{i-1} - \Phi_i$$

- Amortization holds if $\Phi_0 = 0$ and $\Phi_n \geq 0$

DEAMORTIZED COLA

- *Bender et al.* deamortize *ephemeral* inserts w/ redundant binary
- Race a thread merging arrays w/ one producing new ones.
- Each entry moves through **log n** arrays, so do **O(log n)** merges before each insert.
- Each array is immutable, but de-amortizing using this scheme isn't purely functional.

REDUNDANT BINARY

- Digits are 0, 1, and 2, but still worth 1,2,4,8,16..
- Between any two 2s a 0 must exist.
- Addition can be implemented by touching at most 3 digits.

DOUBLE DIPPING

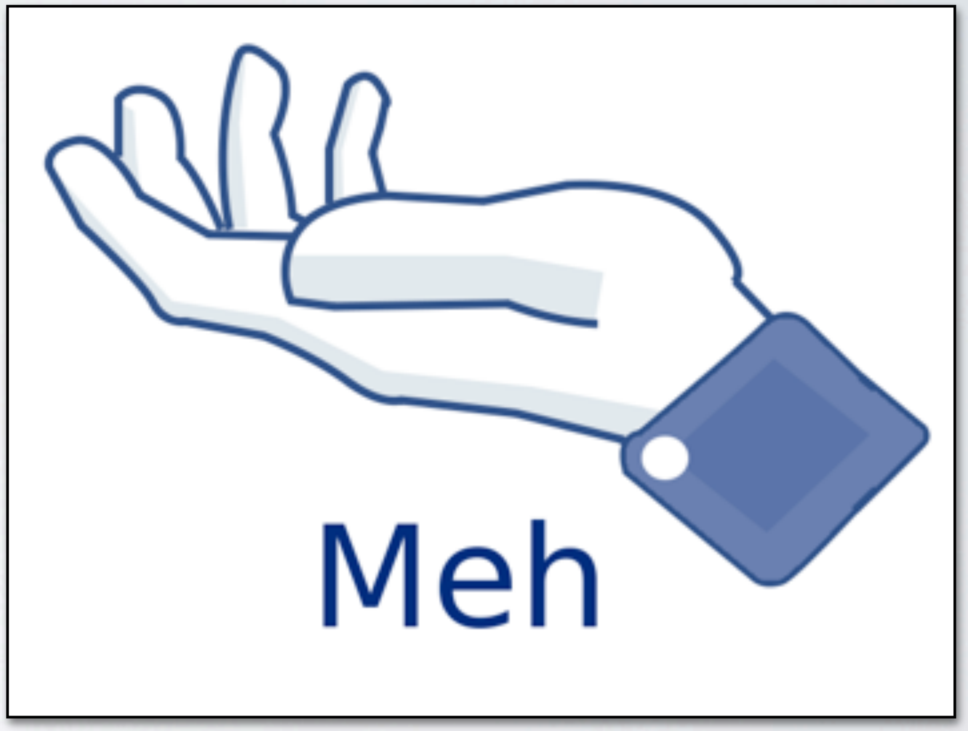
- These analyses so far are flawed in a persistent setting!
- After I insert an element into a Map I might go back and insert another into the old Map.
- I can't just execute operations cheaply and use the credit towards big operations later!

LAZY PERSISTENT AMORTIZATION

- You can often use laziness to resolve the the issue with multiple futures by setting up the expensive operation as a thunk that will eventually be forced.
- In that way multiple versions of the structure can share the thunk and it will be forced by the first one that needs it, but the others who go to force the same thing will share the result.
- This hasn’t worked for me thus far for the COLA.

IF YOU DON'T LIKE THE PROBLEM CHANGE THE PROBLEM

- Give up on worst-case asymptotics
- Give up on *persistent* amortization
- ???



SKEW BINARY

- In the code that follows, instead of Redundant Binary I use Skew Binary.
- Digits are now worth $2^{k+1}-1$ *e.g.* 1, 3, 7, 15, 31, ...
- Digits are 0, 1, or 2, but use of 2 is restricted
- Can only use the 2 as the least significant non-zero digit
- Each number can be represented precisely one way
- Incrementing by 1 never carries more than one place
- Not enough to deamortize, but it ensures I only do one merge at a time.

15	7	3	1	#
			0	0
			1	1
			2	2
		1	0	3
		1	1	4
		1	2	5
		2	0	6
	1	0	0	7
	1	0	1	8
	1	0	2	9
	1	1	0	10
	1	1	1	11
	1	1	2	12
	1	2	0	13
	2	0	0	14
1	0	0	0	15

FRACTIONAL CASCADING

- Searching the structure we've defined so far takes

$$O(\log^2(N/B) + a/B)$$

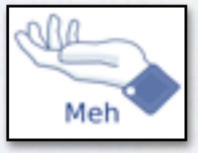
- I resolved this using fractional cascading as in the original paper

FRACTIONAL CASCADING

- Searching the structure we've defined so far takes

$$O(\log^{2}(N/B) + a/B)$$

- I resolved this using fractional cascading as in the original paper

- ... and obtained a structure that inserted ~10x slower and searched ~3x slower than **Data.Map.** 

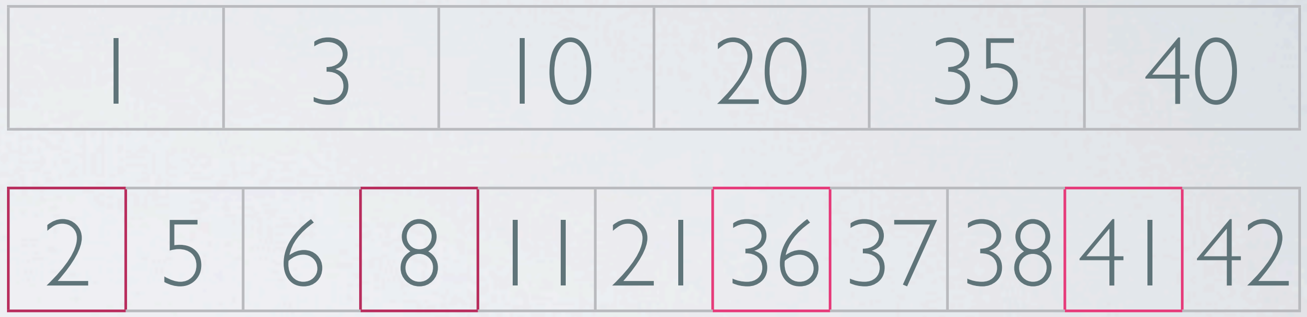
- **Data.Map** has excellent constant factors!

FRACTIONAL CASCADING

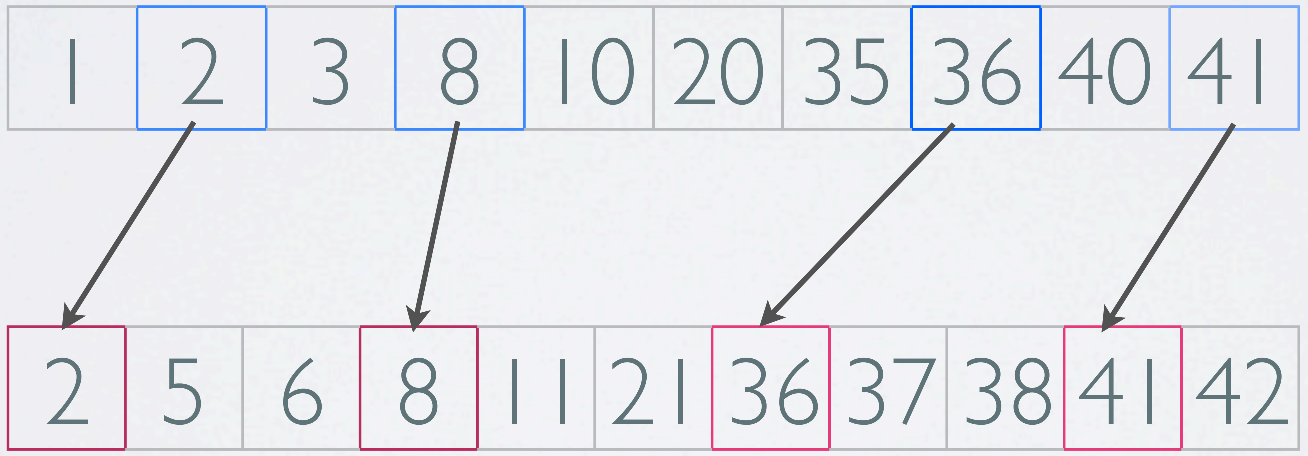
- Search **m** sorted arrays each of sizes up to **n** at the same time.
- Precalculations are allowed, but not a huge explosion in space
- Very useful for many computational geometry problems.
- Naïve Solution: Binary search each separately in $O(m \log n)$
- With Fractional Cascading: $O(\log mn) = O(\log m + \log n)$

FRACTIONAL CASCADING

- Consider 2 sorted lists *e.g.*



- Copy every **k**th entry from the second into the first



- After a failed search in the first, you know have to search a *constant* **k**-sized fragment of the second.

IMPLICIT FRACTIONAL CASCADING

- New trick:
- We copy every **k**th entry up from the next largest array.
- If we had a way to count the number of forwarding pointers up to a given position we could just multiply that # by **k** and not have to store the pointers themselves

SUCCINCT DICTIONARIES

- Given a bit vector of length **n** containing **k** ones *e.g.*

0	0	1	1	0	1	1	0	0	1	1	1	0	1	1	1	1	1	0	1	1	1	0	1	1	1	1	1	0	0

- There exist $\binom{n}{k}$ such vectors.

- Knowing nothing else we could store that choice in H_0 bits

$$H_0 = \log \binom{n}{k} + 1$$

- With just $H_0 + o(H_0)$ total space we get an $O(1)$ version of:

$$\text{rank}_a(S,i) = \text{\# of occurrences of } a \text{ in } S[0..i)$$

IMPLICIT FORWARDING

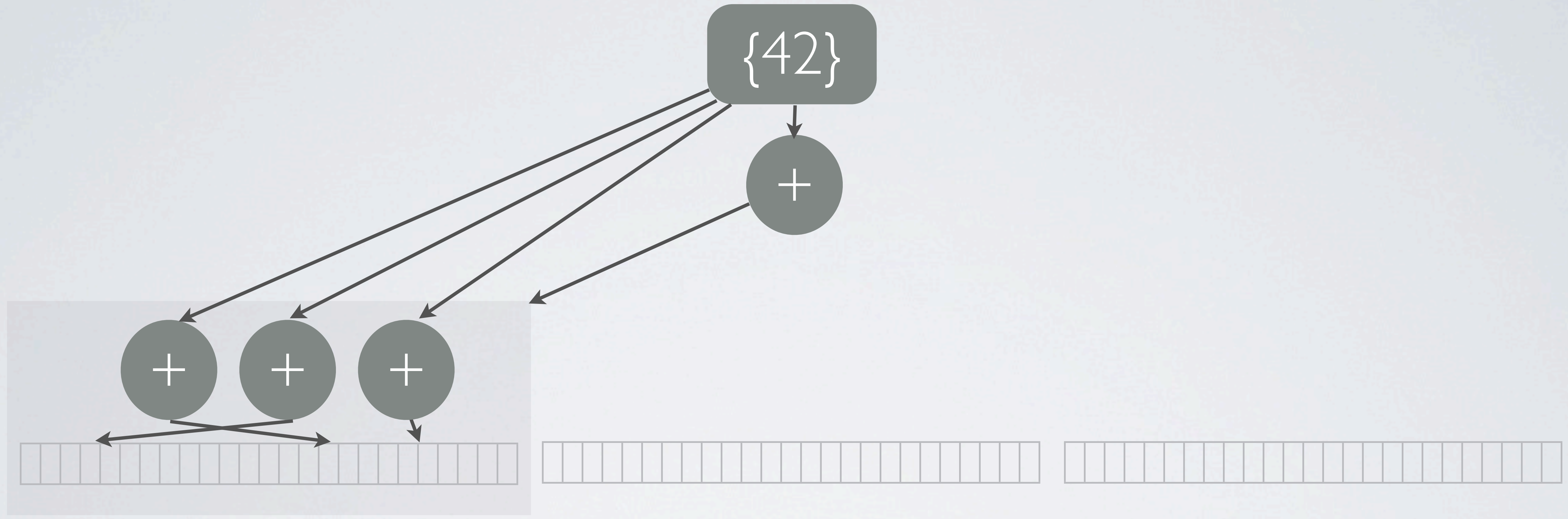
- Store a bitvector for each key in the vector that indicates if the key is a forwarding pointer, or has a value associated.
- To index into the values use rank up to a given position instead.
- This can also be used to represent deletion flags succinctly.
- In practice we use non-succinct algorithms. (rank9, poppy)

IMPLICIT FRACTIONAL CASCADING

IMPLICIT FRACTIONAL CASCADING

- ... still has ~8x slower inserts and searches ~3x slower than `Data.Map`.

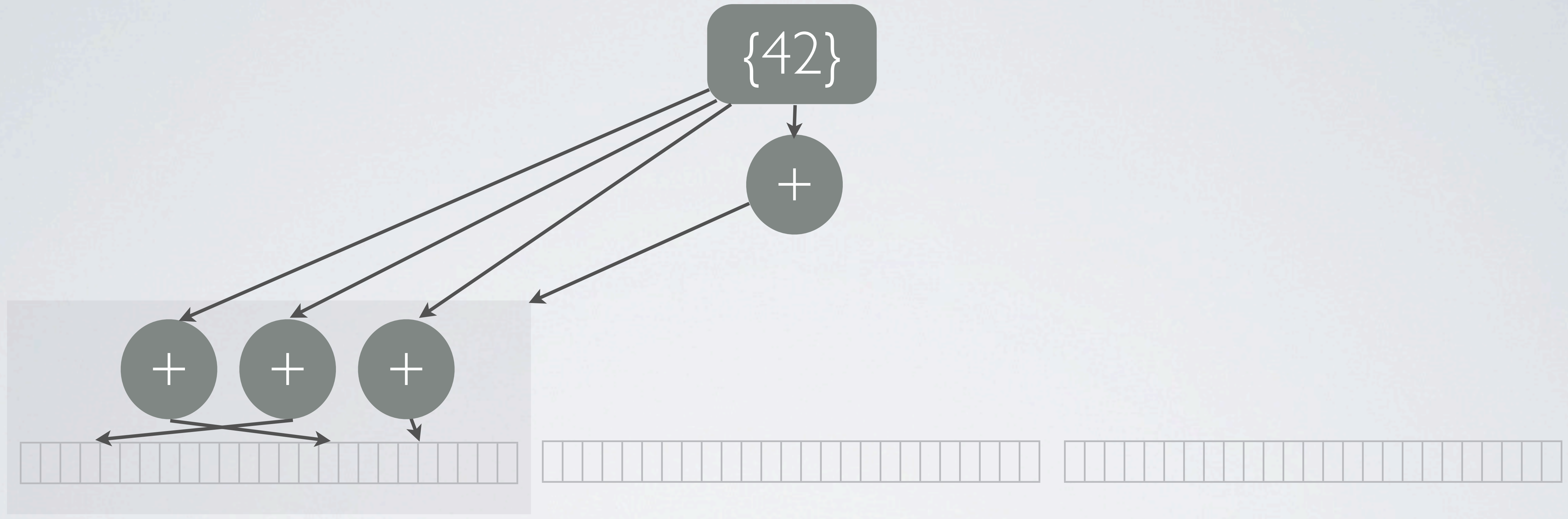
BLOOM-FILTERS



- Associate a *hierarchical* Bloom filter with each array tuned to a false positive rate that balances the cost of the cache misses for the binary search against the cost of hashing into the filter.

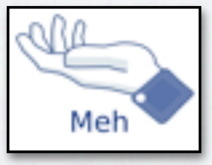
- Improves upon a version of the "Stratified Doubling Array"

BLOOM-FILTERS

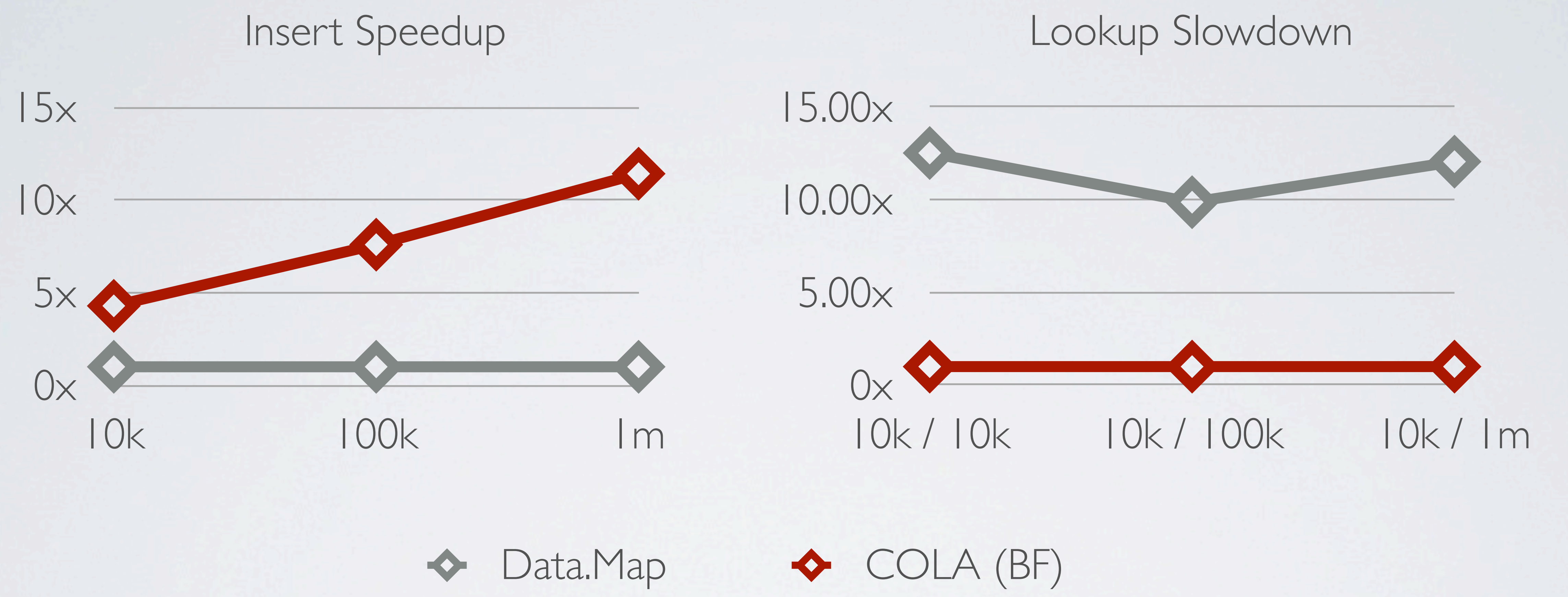


- Associate a *hierarchical* Bloom filter with each array tuned to a false positive rate that balances the cost of the cache misses for the binary search against the cost of hashing into the filter.

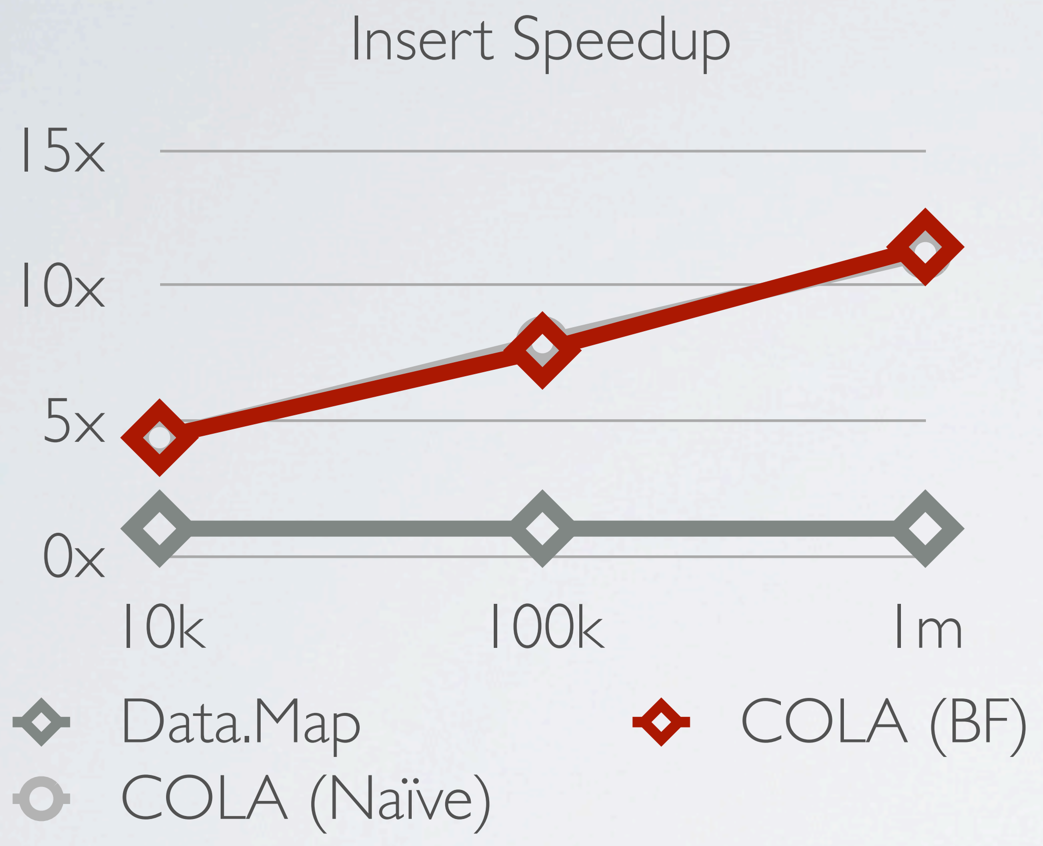
- Improves upon a version of the "Stratified Doubling Array"

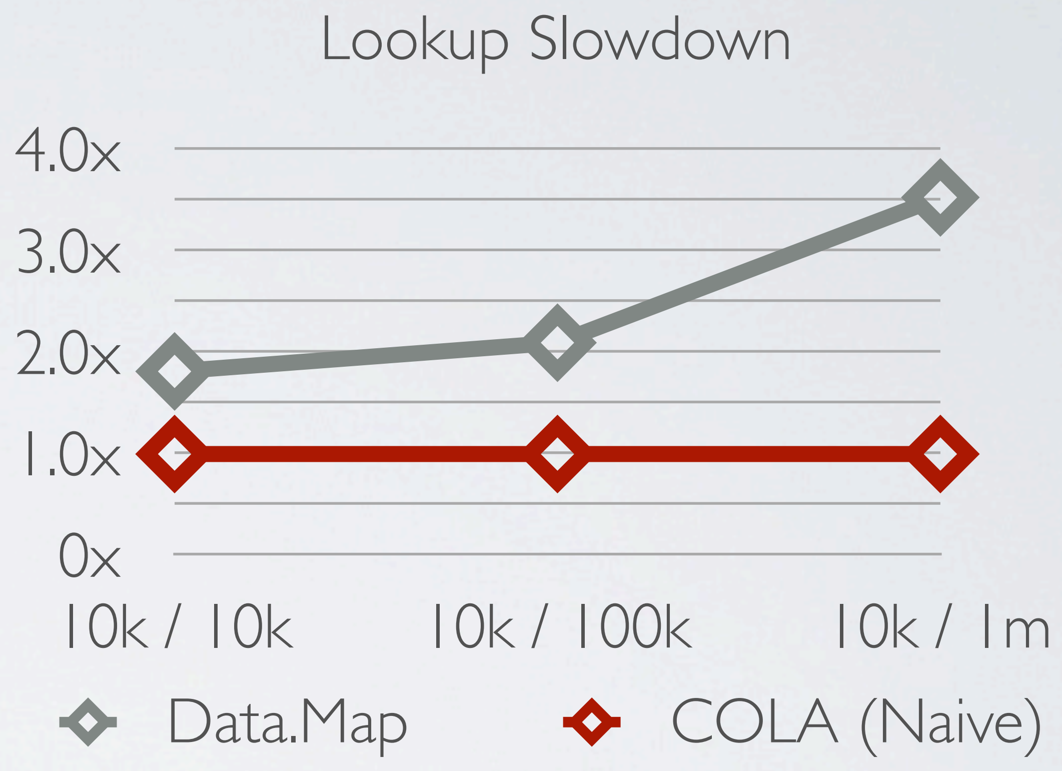
- Not Cache-Oblivious! 

BLOOM-FILTERS



NAÏVE SKEW COLA





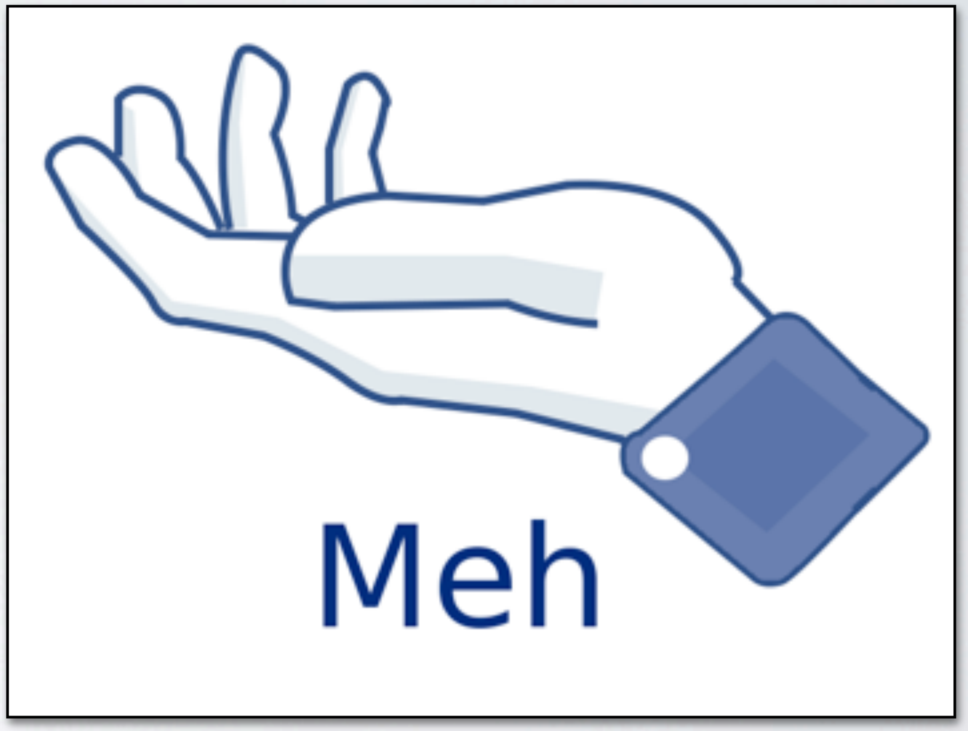
- Just doing skew binary merges
- Purely functional, but inserts only amortize ephemerally.
- Easily portable

WALKTHROUGH

- Development Haddocks: http://ekmett.github.io/structures/

IF YOU DON'T LIKE THE PROBLEM CHANGE THE PROBLEM

- Gave up on worst-case asymptotics!
- Gave up on *persistent* amortization!
- Gave up the asymptotic bounds!
- But it is still fast.



OUT OF TIME

- Integrate deletes and range queries.
- Using this as a backend for my sparse matrix multiplier yields a "planless" table join algorithm when combined with the compressed Morton-order keys from **analytics**.
- We could borrow the MVCC approach from the Stratified B-Tree. This gives us a principled version of **split**, and an efficient **datalog**, but a weird API
- Wavelet trees are the most interesting data structure I've worked with in my lifetime.
- Stream fusion is both a blessing and a curse
- Unboxed vectors of Sums

QUESTIONS?

- The code is on github: http://github.com/ekmett/structures
- New stickers!