FUNCTIONALLY OBLIVIOUS

(AND SUCCINCT)

Edward Kmett
McGraw Hill Financial

BUILDING BETTER TOOLS

- Cache-Oblivious Algorithms
- Succinct Data Structures

DATA.MAP

- Production:
  - `empty :: Ord k => Map k a`
  - `insert :: Ord k => k -> a -> Map k a -> Map k a`
- Consumption:
  - `null :: Ord k => Map k a -> Bool`
  - `lookup :: Ord k => k -> Map k a -> Maybe a`

DATA.MAP

- Built by Daan Leijen.
- Maintained by Johan Tibell and Milan Straka.
- Battle Tested. Highly Optimized. In use since 1998.
- Built on Trees of Bounded Balance
- The defacto benchmark of performance.
- Designed for the Pointer/RAM Model

WHAT I WANT

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

THE DUMBEST THING THAT CAN WORK

- Take an array of **(key, value)** pairs sorted by key and arrange it contiguously in memory
- Binary search it.
- Eventually your search falls entirely within a cache line.

BINARY SEARCH

```haskell
-- | Binary search assuming 0 <= l <= h.
-- Returns h if the predicate is never True over [l..h)

search :: (Int -> Bool) -> Int -> Int -> Int
search p = go where
  go l h
    | l == h    = l
    | p m       = go l m
    | otherwise = go (m+1) h
    where m = l + unsafeShiftR (h - l) 1
{-# INLINE search #-}
```

OFFSET BINARY SEARCH

Pro Tip!

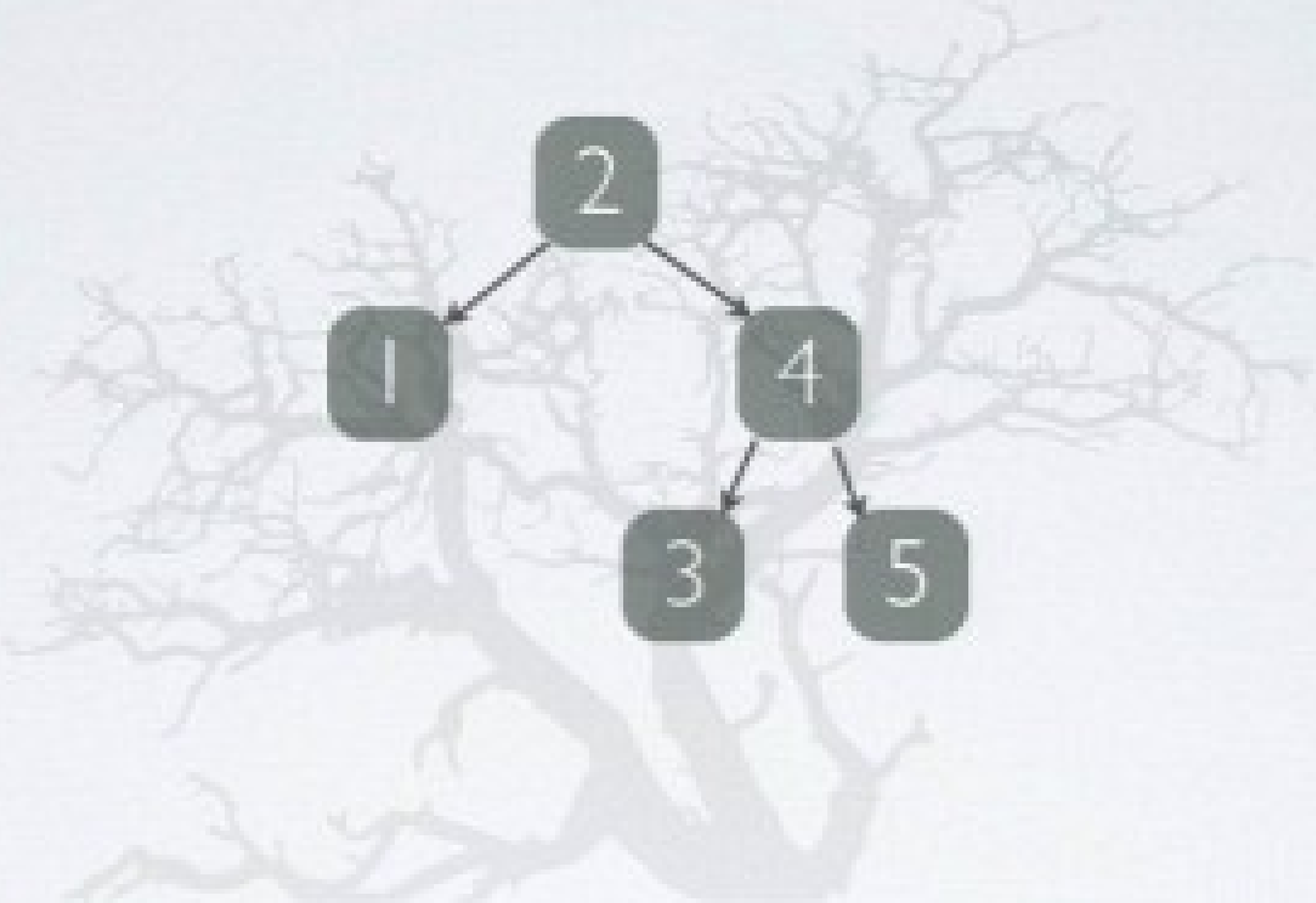
```haskell
-- | Offset binary search assuming 0 <= l <= h.
-- Returns h if the predicate is never True over [l..h)

search :: (Int -> Bool) -> Int -> Int -> Int
search p = go where
  go l h
    | l == h    = l
    | p m       = go l m
    | otherwise = go (m+1) h
    where hml = h - l
          m = l + unsafeShiftR hml 1 + unsafeShiftR hml 6
{-# INLINE search #-}
```

Avoids thrashing the same lines in k-way set associative caches near the root.

RAM MODEL

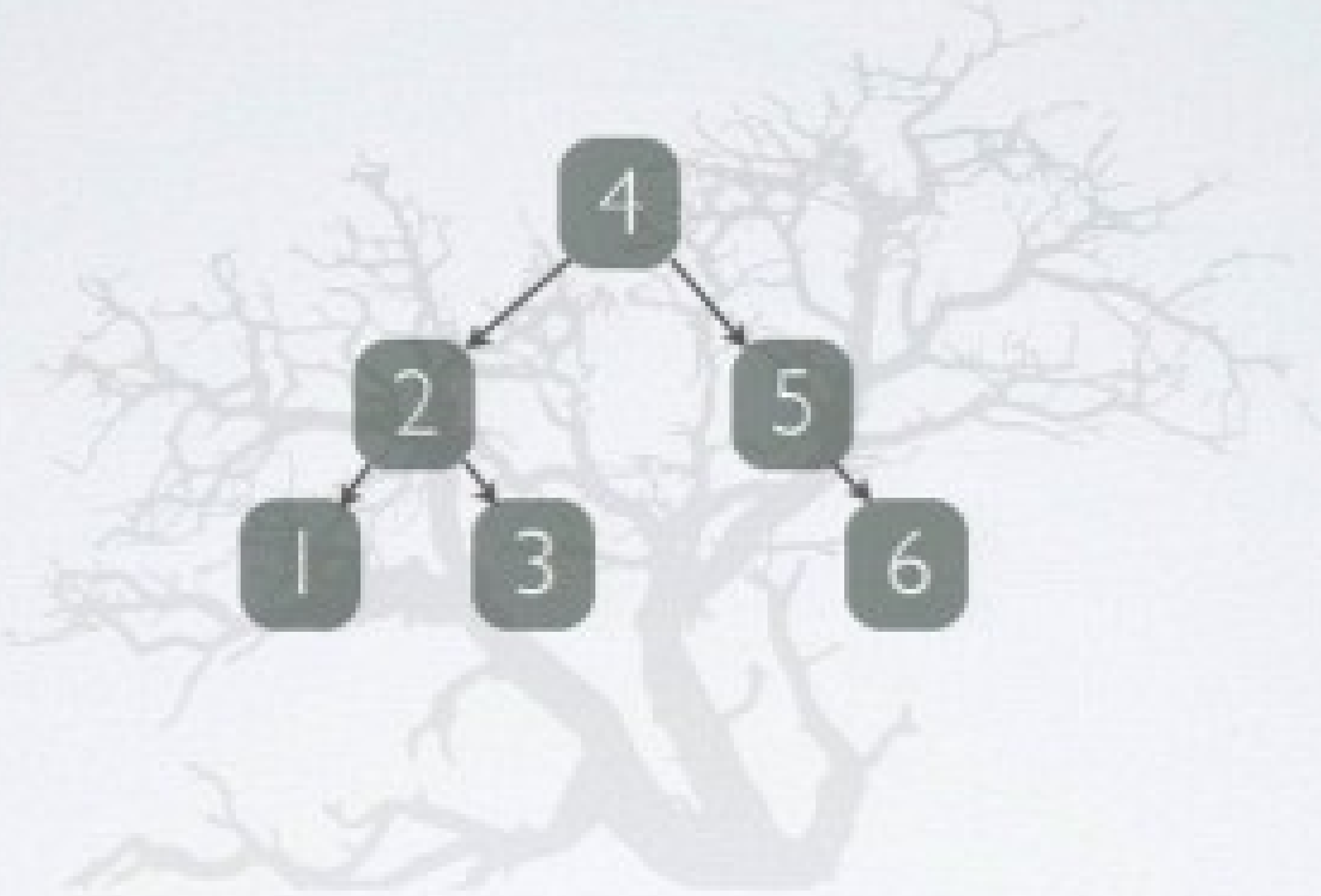
- Almost everything you do in Haskell assumes this model
- Good for ADTs, but not a realistic model of today's hardware

DATA.MAP



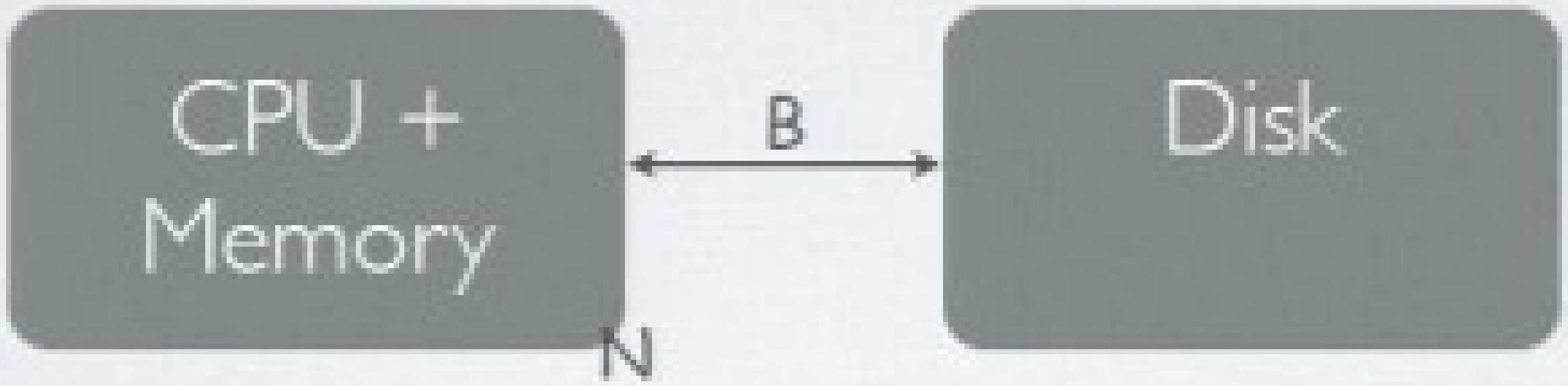
"Binary search trees of bounded balance"

DATA.MAP



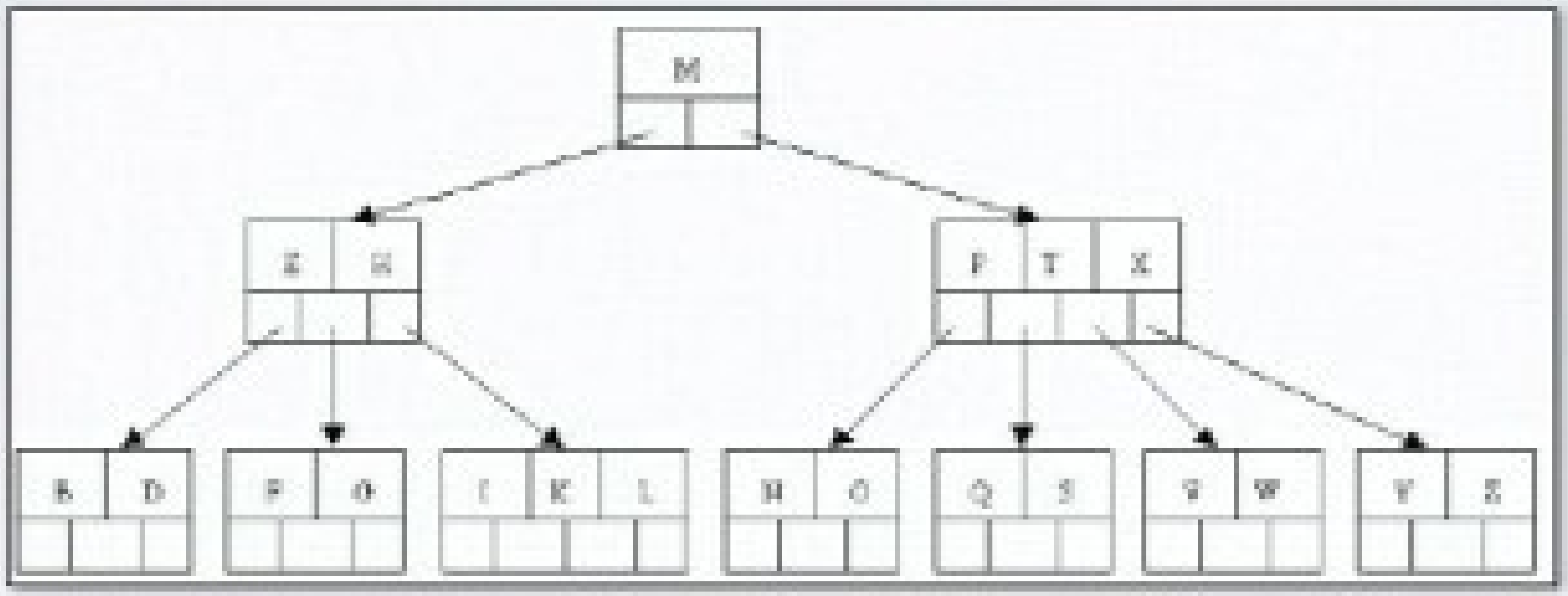
"Binary search trees of bounded balance"

IO MODEL



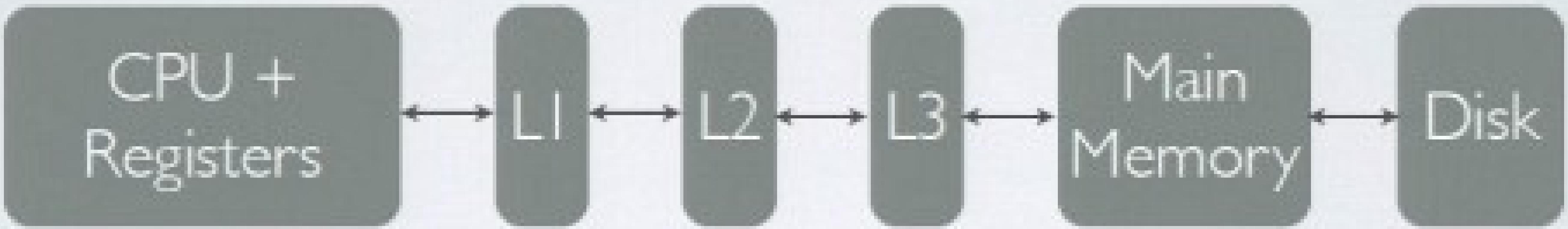
- Can Read/Write Contiguous Blocks of Size **B**
- Can Hold **M/B** blocks in working memory
- All other operations are "Free"

B-TREES

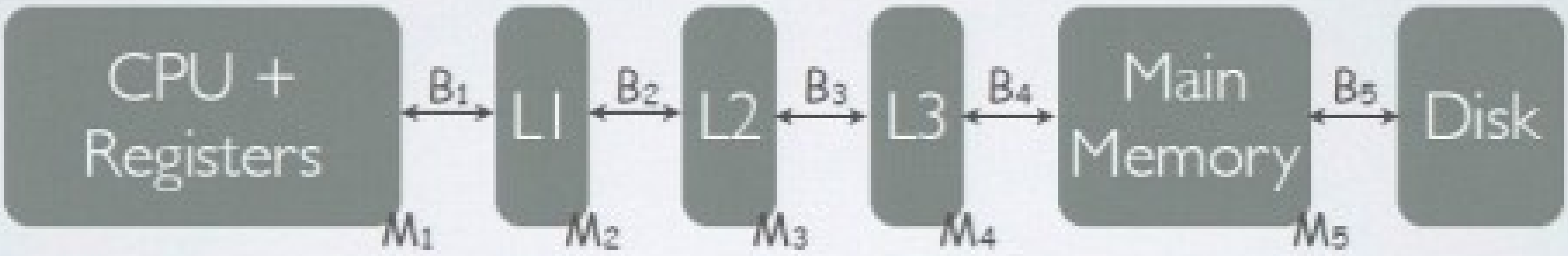


- Occupies **O(N/B)** blocks worth of space
- Update in time **O(log(N/B))**
- Search **O(log(N/B) + α/B)** where **α** is the result set size

IO MODEL

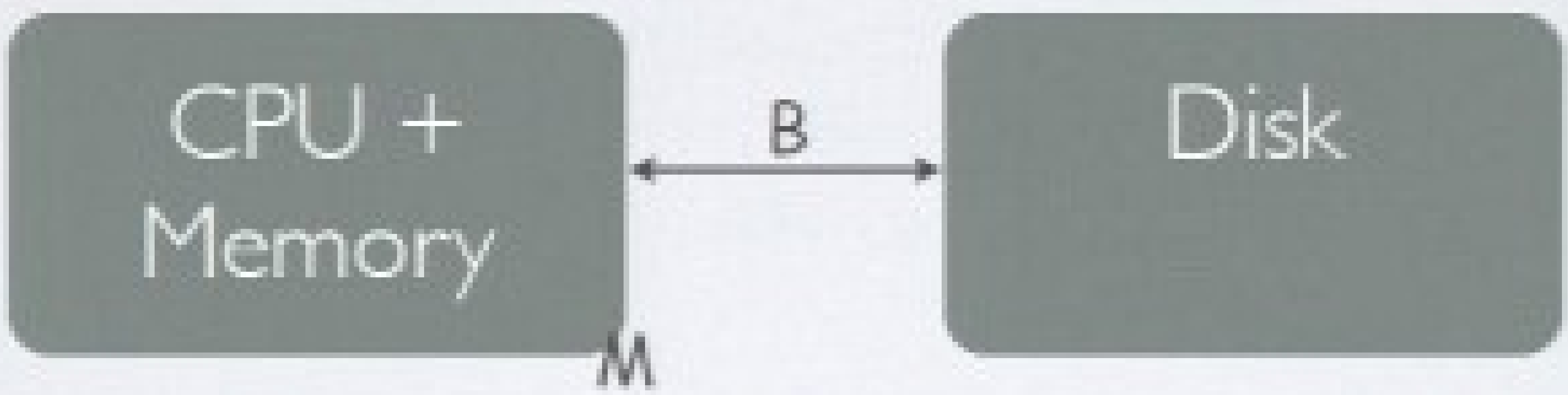


IO MODEL



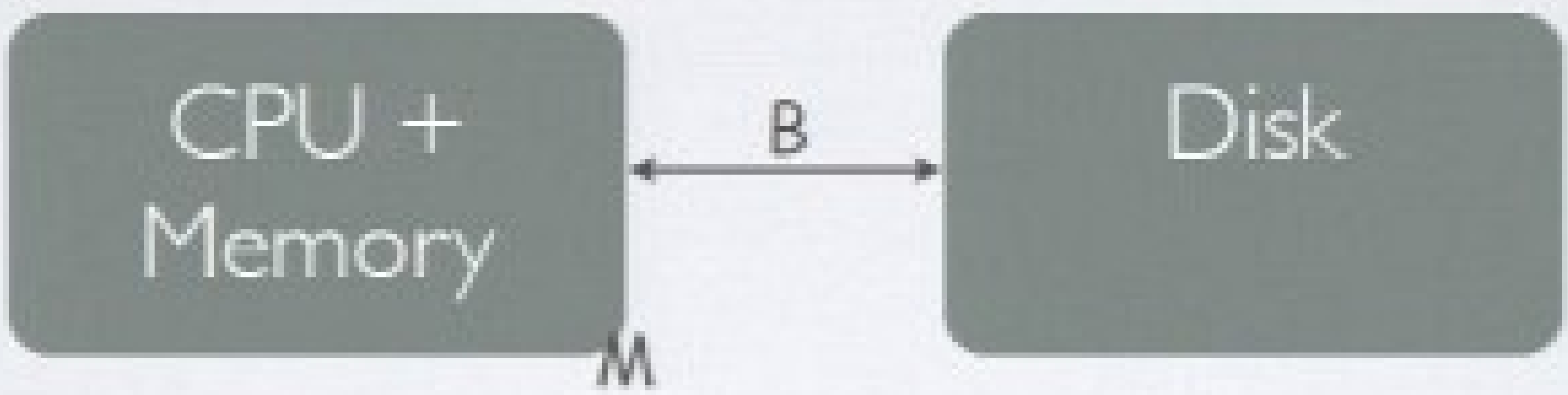
- Huge numbers of constants to tune
- Optimizing for one necessarily sub-optimizes others
- Caches grows exponentially in size and slowness

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

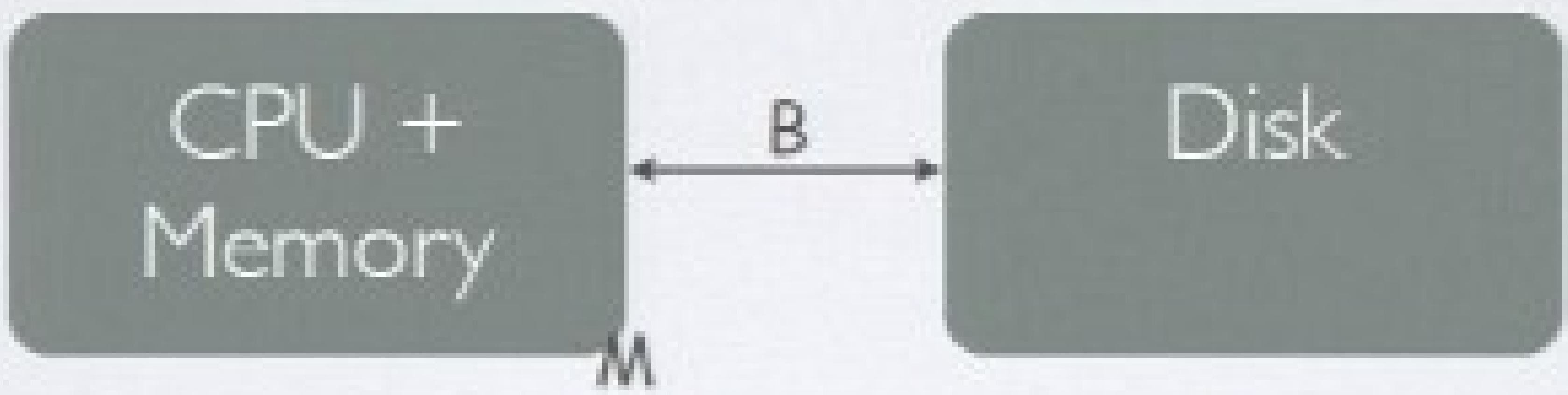
CACHE-OBLIVIOUS MODEL



- 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.

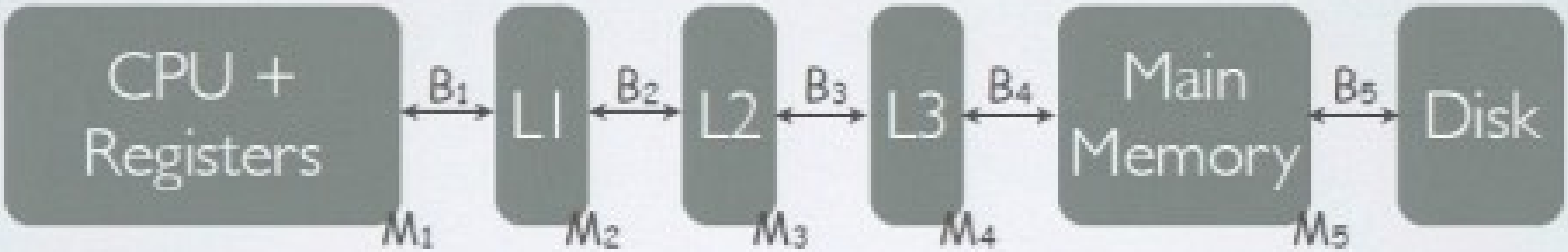
- You can relax the assumption of optimal replacement and model LRU, **k**-way set associative caches, and the like via caches by modest reductions in **M**.

CACHE-OBLIVIOUS MODEL



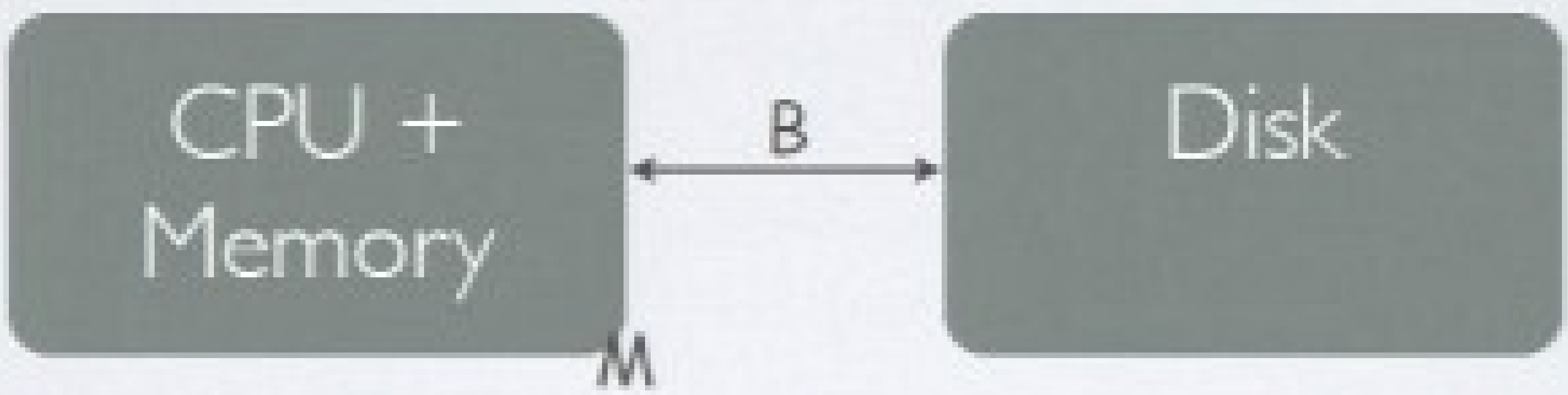
- As caches grow taller and more complex it becomes harder to tune for them at the same time. Tuning for one provably renders you suboptimal for others.
- The overhead of this model is largely compensated for by ease of portability and vastly reduced tuning.
- This model is becoming more and more true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

IO MODEL



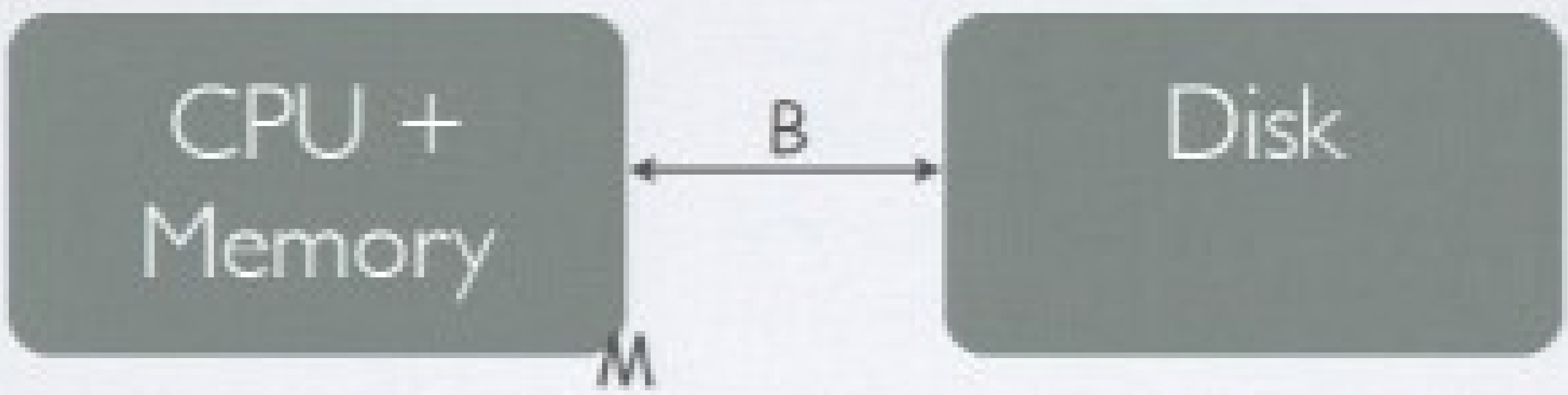
- Huge numbers of constants to tune
- Optimizing for one necessarily sub-optimizes others
- Caches grows exponentially in size and slowness

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



- 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!

DYNAMIZATION

- We have a static structure that does what we want
- How can we make it updatable?
- Bentley and Saxe gave us one way in 1980.

BENTLEY-SAXE

- Linked list of our static structure.
- Each a power of 2 in size.
- The list is sorted strictly monotonically by size.
- Bigger / older structures are later in the list.
- We need a way to merge query results.
- Here we just take the first.

BENTLEY-SAXE

Now let's insert 7

BENTLEY-SAXE

BENTLEY-SAXE

Now let's insert 8

BENTLEY-SAXE

Next insert causes a cascade of carries!

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

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

We computed that oblivous to **B**

SLOPPY AND DYSFUNCTIONAL

- Chris Okasaki would not approve!
- Our analysis used assumed linear/ephemeral access.
- A sufficiently long carry might rebuild the whole thing, but if you went back to the old version and did it again, it'd have to do it all over.
- You can't earn credits and spend them twice!

BENTLEY-SAXE

Next insert causes a cascade of carries!

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

Amortized insert time is $O((\log N)/B)$

We computed that oblivous to B

SLOPPY AND DYSFUNCTIONAL

- Chris Okasaki would not approve!
- Our analysis used assumed linear/ephemeral access.
- A sufficiently long carry might rebuild the whole thing, but if you went back to the old version and did it again, it'd have to do it all over.
- You can't earn credits and spend them twice!

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.

$$amortized_i = actual_i + \Phi_i - \Phi_{i-1}$$

$$actual_i = amortized_i + \Phi_{i-1} - \Phi_i$$

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

NUMBER SYSTEMS

- Unary - Linked List
- Binary - Bentley-Saxe
- Skew-Binary - Okasaki's Random Access Lists
- Zeroless Binary - ?

0				0
1				1
2			1	0
3			1	1
4		1	0	0
5		1	0	1
6		1	1	0
7		1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0

ZEROLESS BINARY

- Digits are all 1, 2.
- Unique representation

0			0
1			1
2			2
3			1
4		1	2
5		1	1
6		2	2
7		2	1
8		3	2
9		3	1
10	1	2	2

MODIFIED ZEROLESS BINARY

- Digits are all 1, 2 or 3.
- Only the leading digit can be 1
- Unique representation
- Just the right amount of lag

0			0
1			1
2			2
3			3
4		1	2
5		1	3
6		2	2
7		2	3
8		3	2
9		3	3
10	1	2	2

Binary

0				0
1				1
2			1	0
3			1	1
4		1	0	0
5		1	0	1
6		1	1	0
7		1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0

Zeroless Binary

0			
1			1
2			2
3			1
4		1	2
5		1	1
6		2	2
7		2	1
8	1	1	2
9	1	1	1
10	1	2	2

Modified Zeroless Binary

0			
1			1
2			2
3			3
4		1	2
5		1	3
6		2	2
7		2	3
8		3	2
9		3	3
10	1	2	2

PERSISTENTLY AMORTIZED

```haskell
data Map k a
  = M0
  | M1 !(Chunk k a)
  | M2 !(Chunk k a) !(Chunk k a) (Chunk k a) !(Map k a)
  | M3 !(Chunk k a) !(Chunk k a) !(Chunk k a) (Chunk k a) !(Map k a)

data Chunk k a = Chunk !(Array k) !(Array a)

-- | O(log(N)/B) persistently amortized. Insert an element.
insert :: (Ord k, Arrayed k, Arrayed v) => k -> v -> Map k v -> Map k v
insert k0 v0 = go $ Chunk (singleton k0) (singleton v0) where
  go as M0                   = M1 as
  go as (M1 bs)              = M2 as bs (merge as bs) M0
  go as (M2 bs cs bcs xs)    = M3 as bs cs bcs xs
  go as (M3 bs _ _ cds xs) = cds `seq` M2 as bs (merge as bs) (go cds xs)
{-# INLINE insert #-}
```

Binary

0				0
1				1
2			1	0
3			1	1
4		1	0	0
5		1	0	1
6		1	1	0
7		1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0

Zeroless Binary

0			
1			1
2			2
3			1
4		1	2
5		1	1
6		2	2
7		2	1
8	1	1	2
9	1	1	1
10	1	2	2

Modified Zeroless Binary

0			
1			1
2			2
3			3
4		1	2
5		1	3
6		2	2
7		2	3
8		3	2
9		3	3
10	1	2	2

PERSISTENTLY AMORTIZED

```haskell
data Map k a
  = M0
  | M1 !(Chunk k a)
  | M2 !(Chunk k a) !(Chunk k a) (Chunk k a) !(Map k a)
  | M3 !(Chunk k a) !(Chunk k a) !(Chunk k a) (Chunk k a) !(Map k a)

data Chunk k a = Chunk !(Array k) !(Array a)

-- | O(log(N)/B) persistently amortized. Insert an element.
insert :: (Ord k, Arrayed k, Arrayed v) => k -> v -> Map k v -> Map k v
insert k0 v0 = go $ Chunk (singleton k0) (singleton v0) where
  go as M0                   = M1 as
  go as (M1 bs)              = M2 as bs (merge as bs) M0
  go as (M2 bs cs bcs xs)    = M3 as bs cs bcs xs
  go as (M3 bs _ _ cds xs) = cds `seq` M2 as bs (merge as bs) (go cds xs)
{-# INLINE insert #-}
```

WHY DO WE CARE?

- Inserts are ~7-10x faster than Data.Map and get faster with scale!
- The structure is easily mmap'd in from disk for offline storage
- This lets us build an "unboxed Map" from unboxed vectors.
- Matches insert performance of a B-Tree without knowing B.
- Nothing to tune.

PROBLEMS

- Searching the structure we've defined so far takes

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

- We only matched insert performance, but not query performance.

- We have to query **O(log n)** structures to answer queries.

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)**

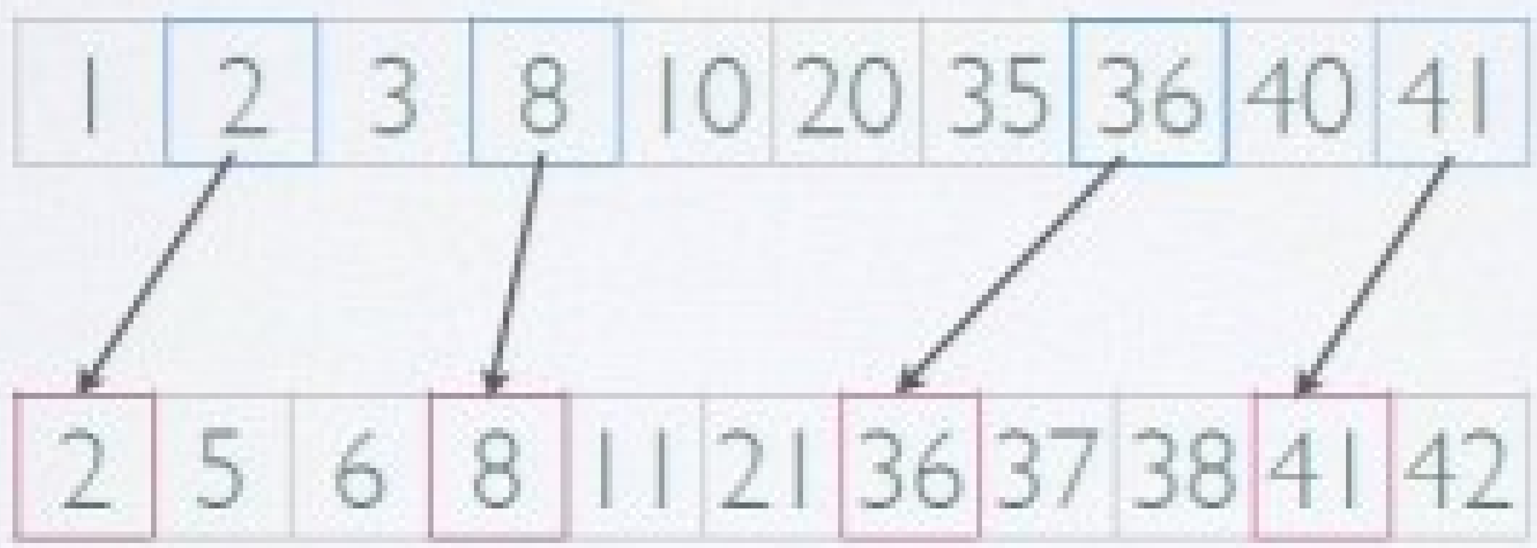
FRACTIONAL CASCADING

- Consider 2 sorted lists *e.g.*

1	3	10	20	35	40					

2	5	6	8	11	21	36	37	38	41	42

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



- After a failed search in the first, you now 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

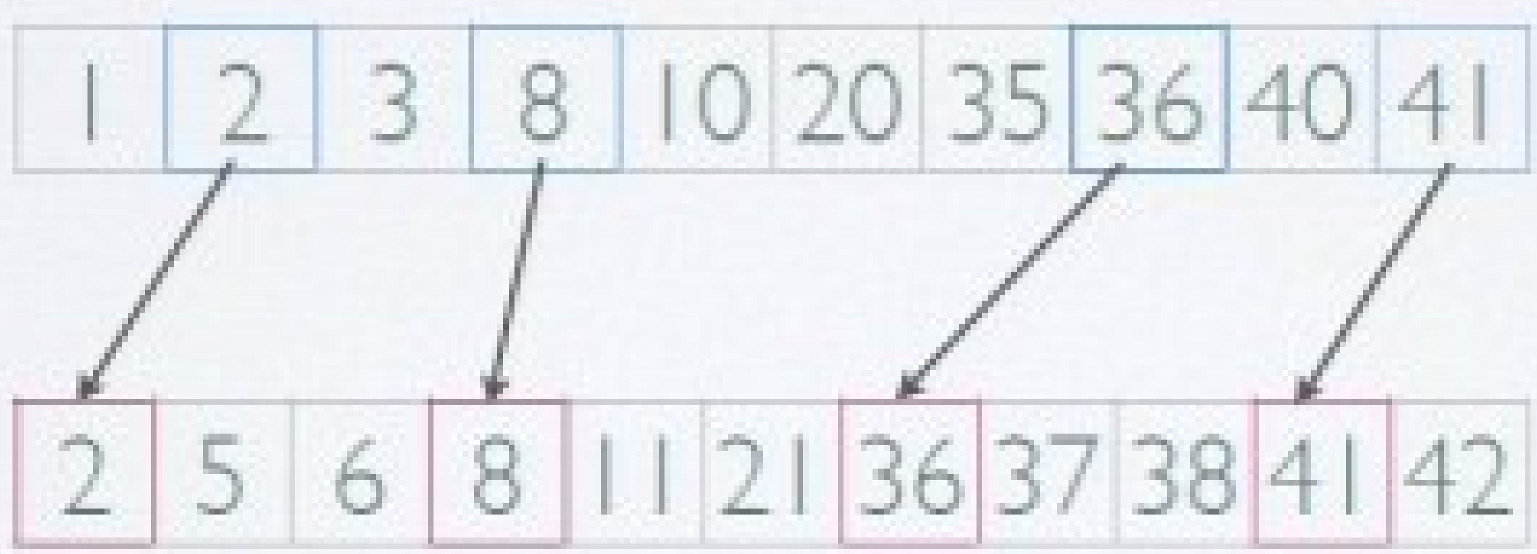
FRACTIONAL CASCADING

- Consider 2 sorted lists *e.g.*

1	3	10	20	35	40

2	5	6	8	11	21	36	37	38	41	42

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



- After a failed search in the first, you now 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

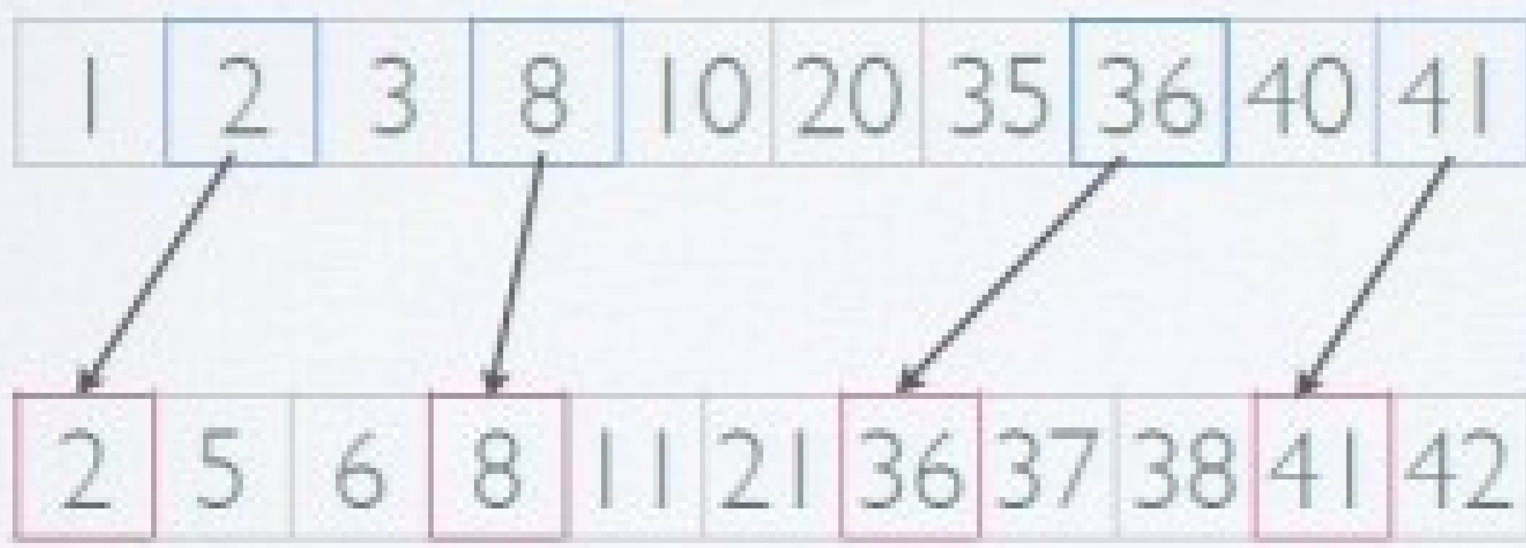
FRACTIONAL CASCADING

- Consider 2 sorted lists *e.g.*

1	3	10	20	35	40

2	5	6	8	11	21	36	37	38	41	42

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



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

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.

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

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

$rank_a(i)$ = # of occurrences of a in $S[0..i)$

$select_a(i)$ = position of the ith a in S

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.

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

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

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

$\text{select}_a(i)$ = position of the ith a in S

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 can use non-succinct algorithms. (rank9, poppy)

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.

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

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

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

$\text{select}_a(i)$ = position of the ith a in S

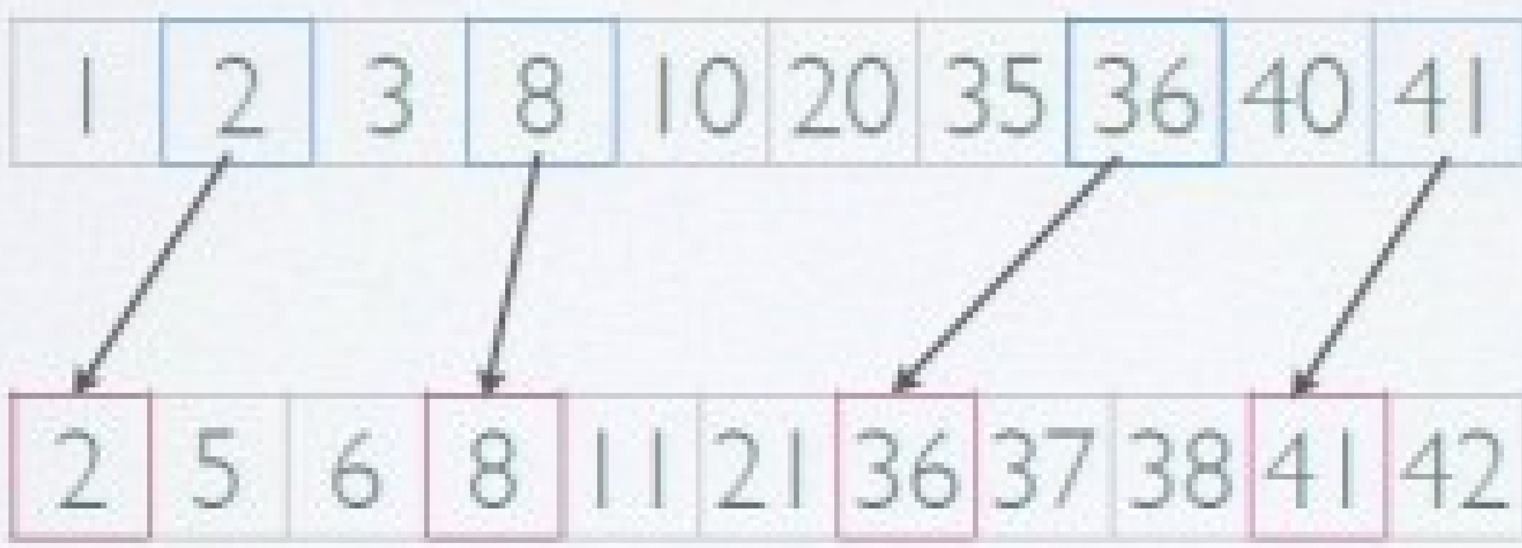
FRACTIONAL CASCADING

- Consider 2 sorted lists *e.g.*

1	3	10	20	35	40

2	5	6	8	11	21	36	37	38	41	42

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



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

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.

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

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

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

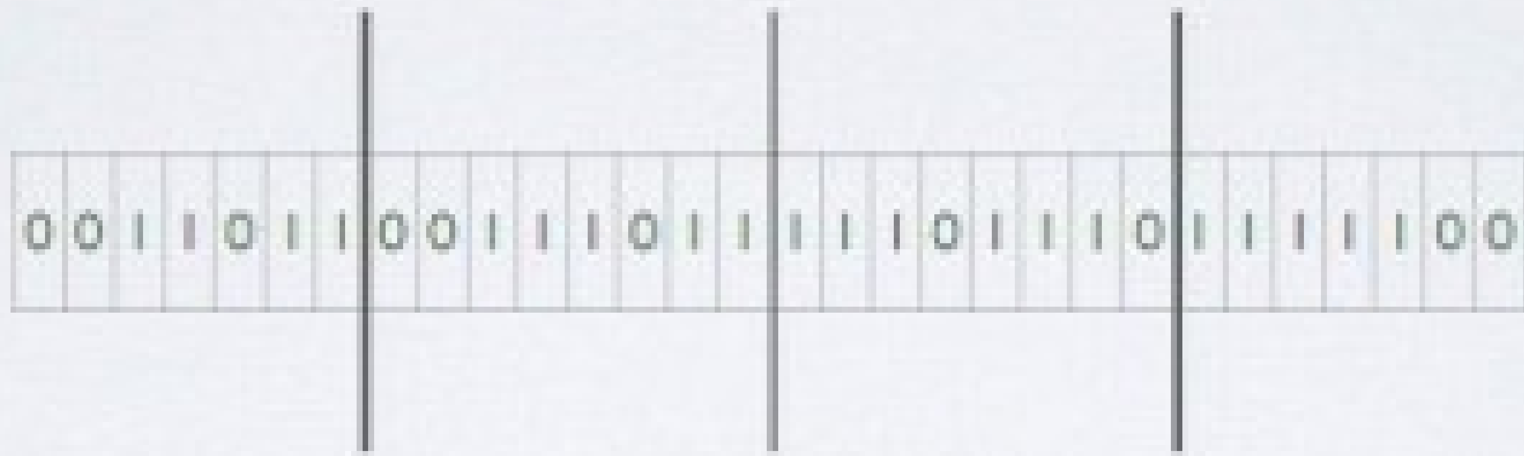
$\mathrm{select}_a(i)$ = position of the ith a in S

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 can use non-succinct algorithms. (rank9, poppy)

NON-SUCCINCT DICTIONARIES

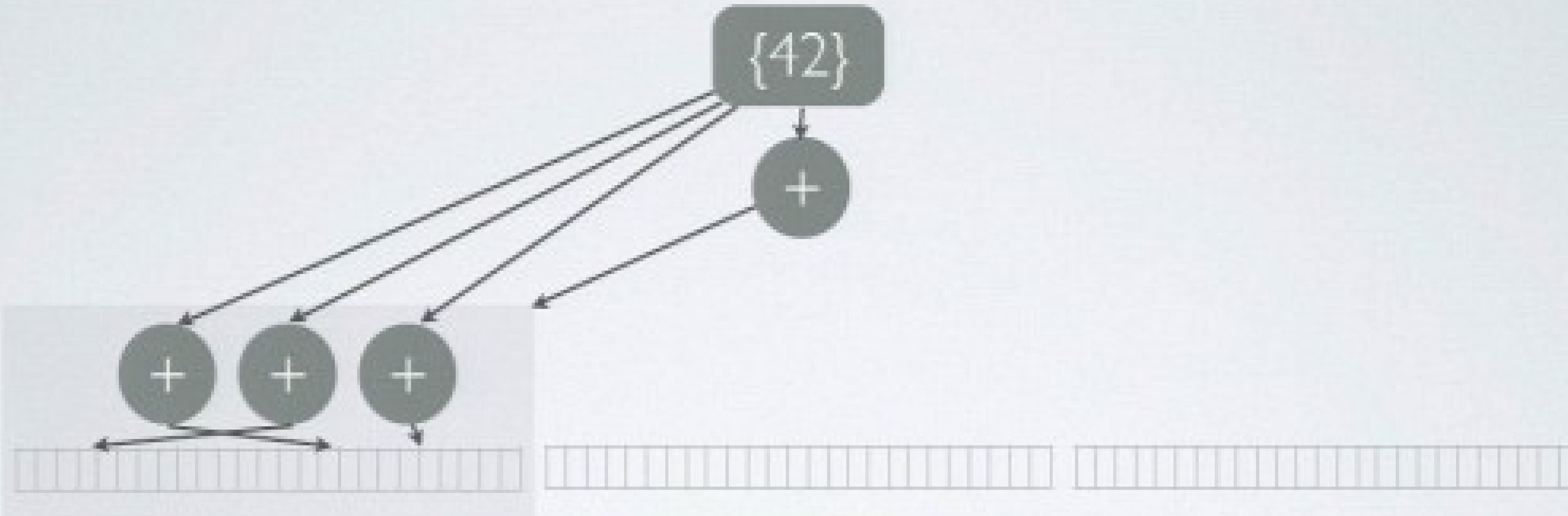
- Given a bit vector of length **n** containing **k** ones *e.g.*



- Break it into chunks of size **log(n)** (or 64)
- Store a prefix sum up to each chunk
- With just **2n** total space we get an **O(1)** version of:

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

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! 

BENEFITS

- Match the asymptotic B-Tree performance without knowing B
- Fully persistent, can edit previous versions.
- Always uses sequential writes on disk
- We get ~10x faster inserts than Data.Map
- We can reuse these techniques for other problem domains

QUESTIONS?

- The code is on github:

  http://github.com/ekmett/structures

  http://github.com/ekmett/succinct

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

Binary

0				0
1				1
2			1	0
3			1	1
4		1	0	0
5		1	0	1
6		1	1	0
7		1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0

Zeroless Binary

0			
1			1
2			2
3			1
4		1	2
5		1	1
6		2	2
7		2	1
8	1	1	2
9	1	1	1
10	1	2	2

Modified Zeroless Binary

0			
1			1
2			2
3			3
4		1	2
5		1	3
6		2	2
7		2	3
8		3	2
9		3	3
10	1	2	2

PERSISTENTLY AMORTIZED

```haskell
data Map k a
  = M0
  | M1 !(Chunk k a)
  | M2 !(Chunk k a) !(Chunk k a) (Chunk k a) !(Map k a)
  | M3 !(Chunk k a) !(Chunk k a) !(Chunk k a) (Chunk k a) !(Map k a)

data Chunk k a = Chunk !(Array k) !(Array a)

-- | O(log(N)/B) persistently amortized. Insert an element.
insert :: (Ord k, Arrayed k, Arrayed v) => k -> v -> Map k v -> Map k v
insert k0 v0 = go $ Chunk (singleton k0) (singleton v0) where
  go as M0                   = M1 as
  go as (M1 bs)              = M2 as bs (merge as bs) M0
  go as (M2 bs cs bcs xs)    = M3 as bs cs bcs xs
  go as (M3 bs _ _ cds xs) = cds `seq` M2 as bs (merge as bs) (go cds xs)
{-# INLINE insert #-}
```