How big is a pointer?

And what's the point of asking?

Martín Farach-Colton
New York University, USA

How big is a pointer?

How big is a pointer?

To address a space of size n

How big is a pointer?

To address a space of size n

$\geq \log n$ bits

How big is a pointer?

To address a space of size n

$\geq \log n$ bits

The end?

Can we compress pointers?

Tiny Pointers: $o(\log n)$-bit pointers

Impossible in general

This talk is about how to make this impossibility a reality for interesting cases

Can we compress pointers?

Tiny Pointers: $o(\log n)$-bit pointers

Questions:

- How?
- Why? When is the size of pointers a bottleneck?

In this talk:

- Theory of tiny pointers [Bender, Conway, FC, Kuszmaul, Tagliavini SODA '23]
- Uses of tiny pointers

A running example of *how* and *why* for Tiny Pointers: Succinct Search Trees

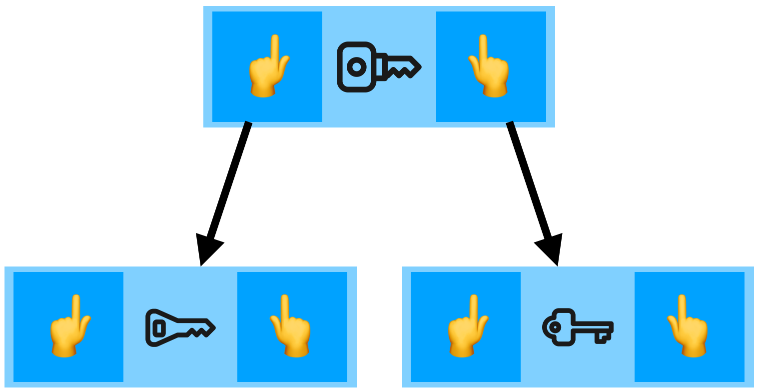
When is a search trees succinct?

Consider your favorite binary search tree: red-black, splay, …

How much space does it take?

- Total pivot key space $nw \geq n \log n$ bits (no matter how we build the tree)
- Total pointer space $\Theta(n \log n)$ bits
- Total $nw + O(n \log n)$

Succinctness = $nw + o(n \log n)$ bits total



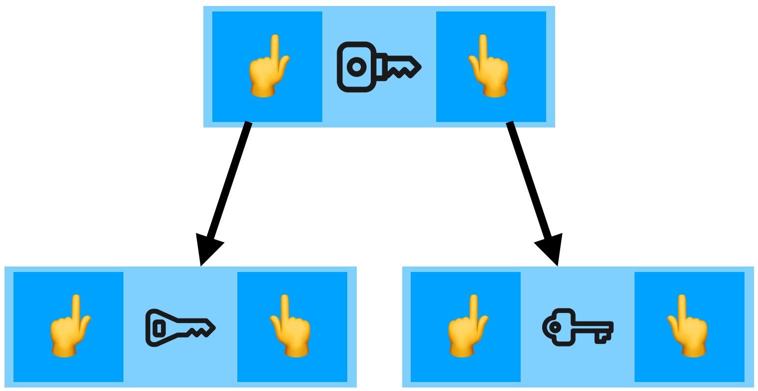
When is a search trees succinct?

Consider your favorite binary search tree: red-black, splay, …

How much space does it take?

- Total pivot key space $nw \geq n \log n$ bits (no matter how we build the tree)
- Total pointer space $\Theta(n \log n)$ bits
- Total $nw + O(n \log n)$

Succinctness = $nw + o(n \log n)$ bits total



Succinct trees: what's known?

Previous literature replaces pointers with other structure with $2n + o(n)$ bits

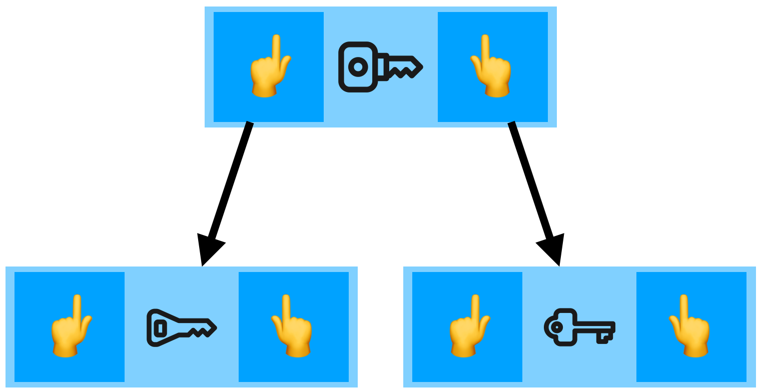
- [Cordova, Navarro. TCS '16][Davoodi et al. MCS '217][Farzan, Munro. ICALP '11], ...
- These structures are small but slow

	Previous	New
Space	Very very very small $nw + 2n + o(n)$ bits	Very very small $nw + o(n \log n)$ bits
Time	Polylogarithmic (or more) overhead for many operations	O(1) time overhead for all operations

Ok, but how do we reduce the size of pointers?

Let's look at one node:

- The only information we have is the key



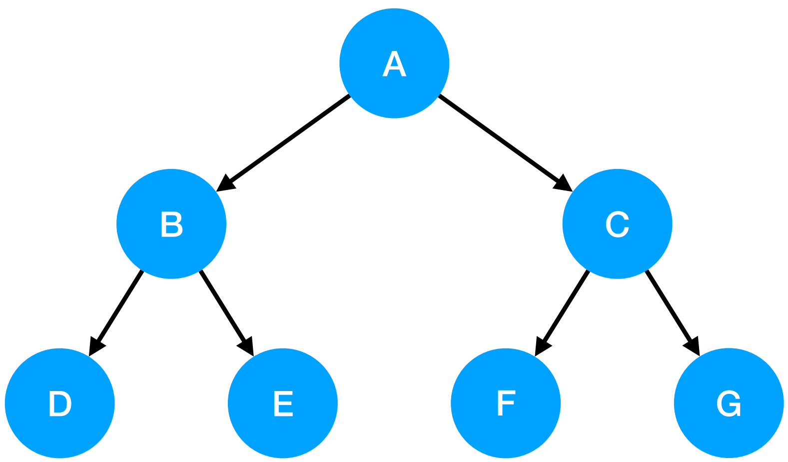
We can reduce the bits in 👆 if it's a function of 🔑

- How???

Tiny pointers through hash tables?

Store needed information in a hash table

- The information we need is key of left child or key of right child



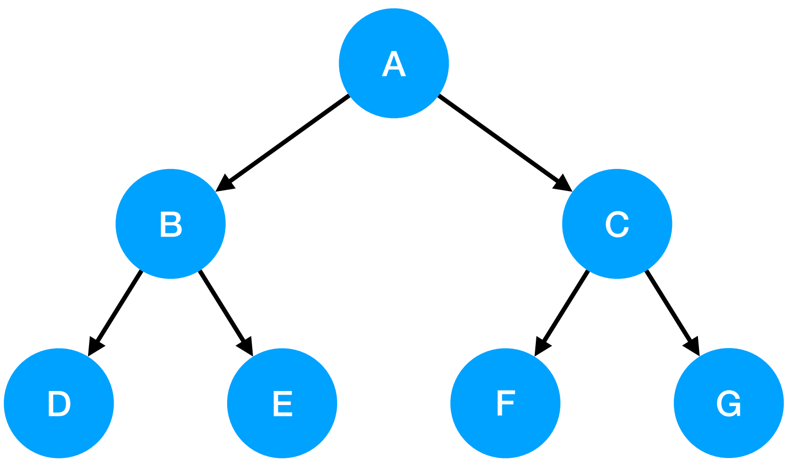
Query("left child of B")→D

Good news: No pointers at all!
Bad news: Space overhead from hash table (e.g. each key is stored twice in table)

Tiny pointers through dereference tables

Our idea: Replace hash table with a ***dereference table***

- What's a dereference table?



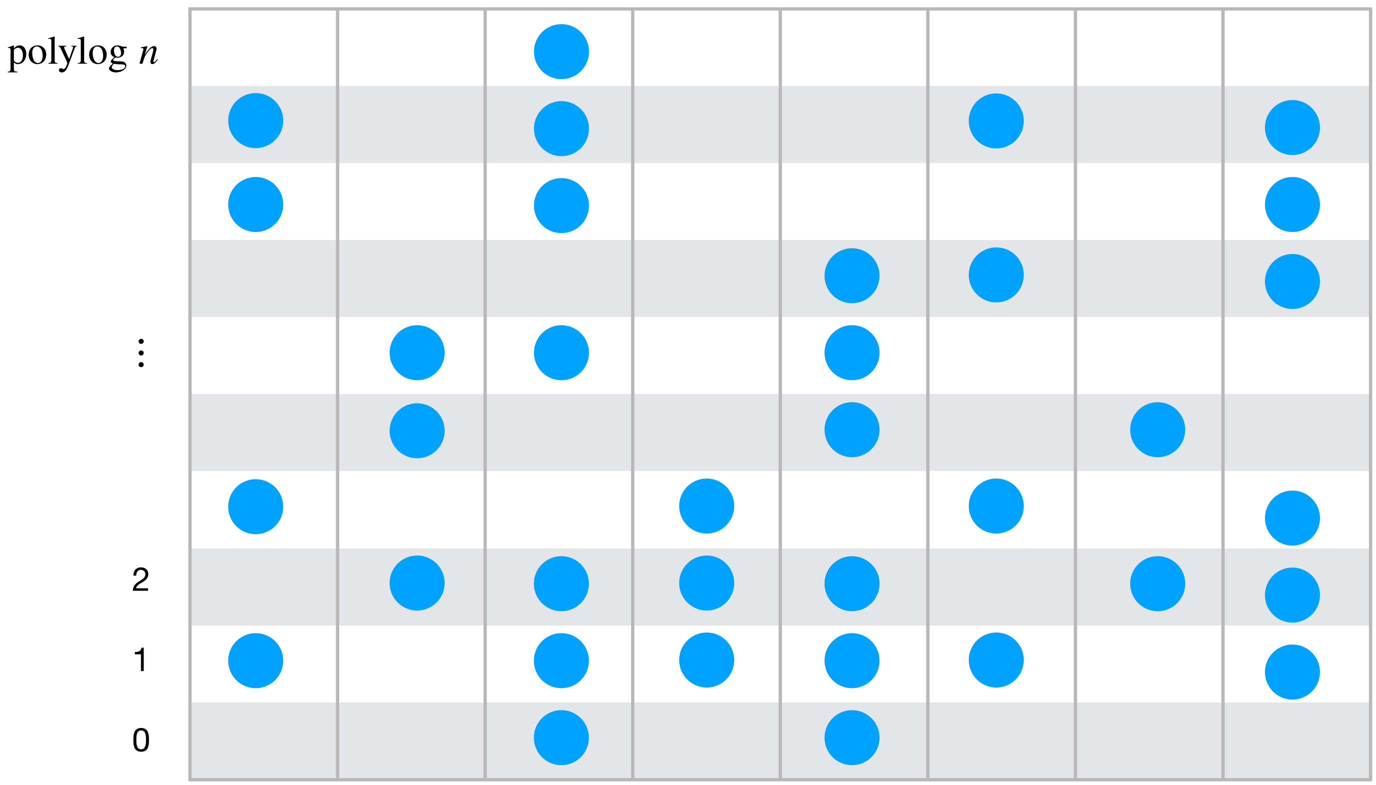
Query(B, left-tiny-pointer)→D, left-tiny-pointer, right-tiny-pointer

The tiny pointers are (small) hints that let us save space

Pointers vs Hash Tables vs Dereference Tables

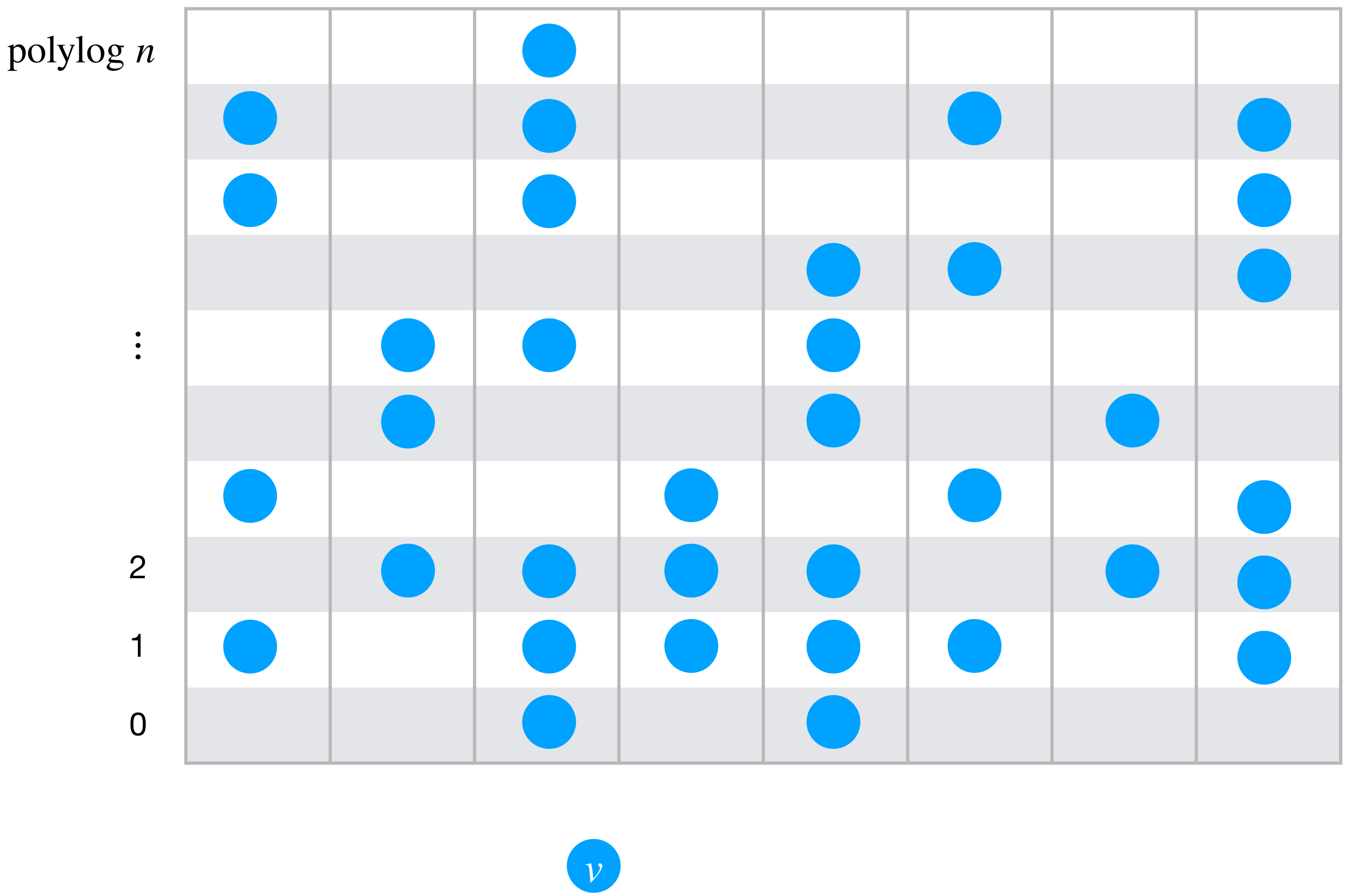
Pointers	Hash Tables	Dereference Tables
Malloc→ptr; *ptr = value	Insert(key, value)	Insert(key, value)→tiny-ptr
*ptr = value	Update(key, value)	Update(key, value, tiny-ptr)
*ptr	Query(key)	Query(key, tiny-ptr)
Free(ptr)	Delete(key)	Delete(key, tiny-ptr)

A simple Dereference Table



Array of size: $(1 + 1/\log n)n$
Bins of size: polylog n

A simple Dereference Table

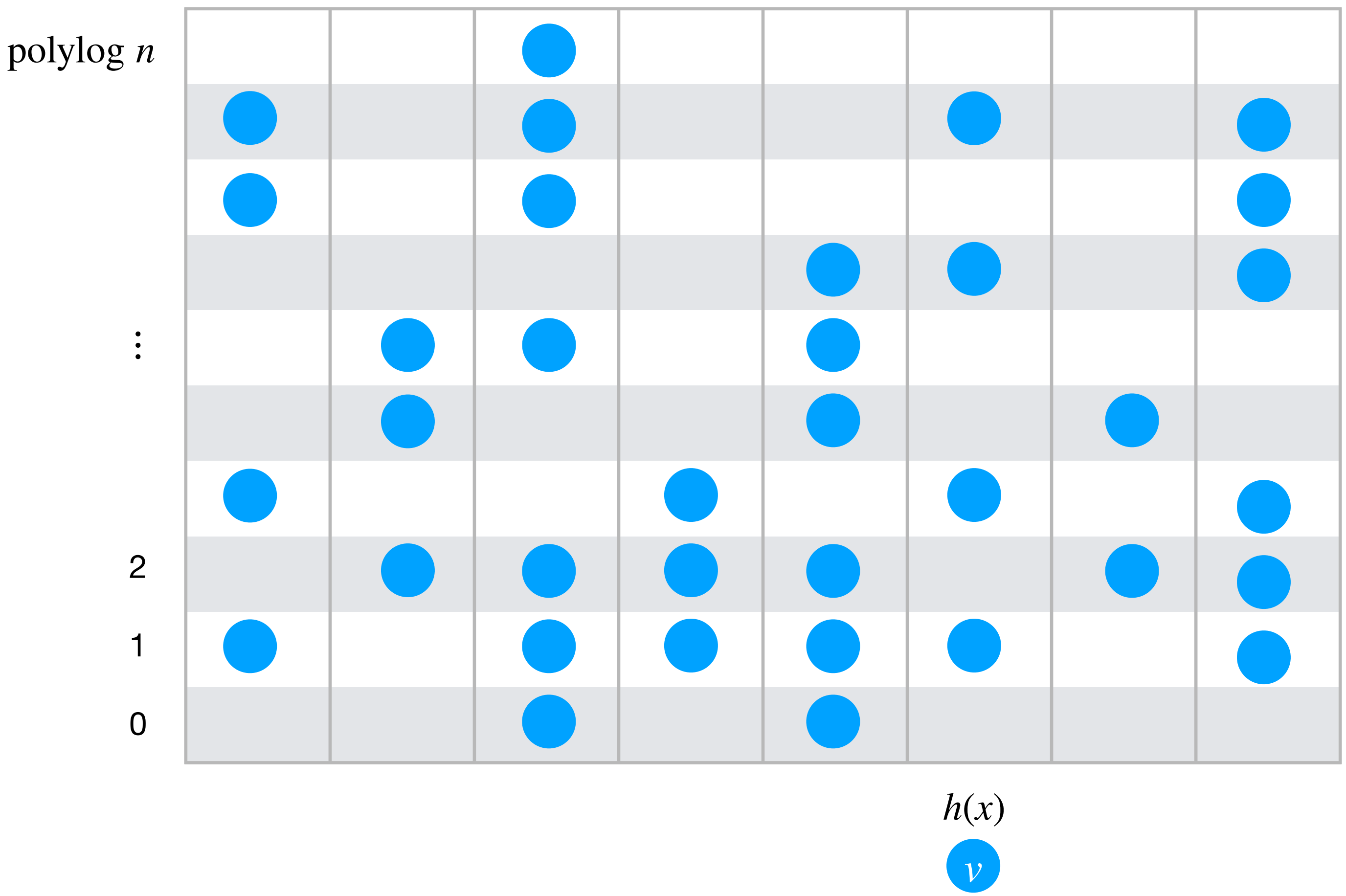


Array of size: $(1 + 1/\log n)n$
Bins of size: polylog n

Inserting an item x, v:

hash into some bin $h(x)$
find an empty slot and store v
return the offset of the slot

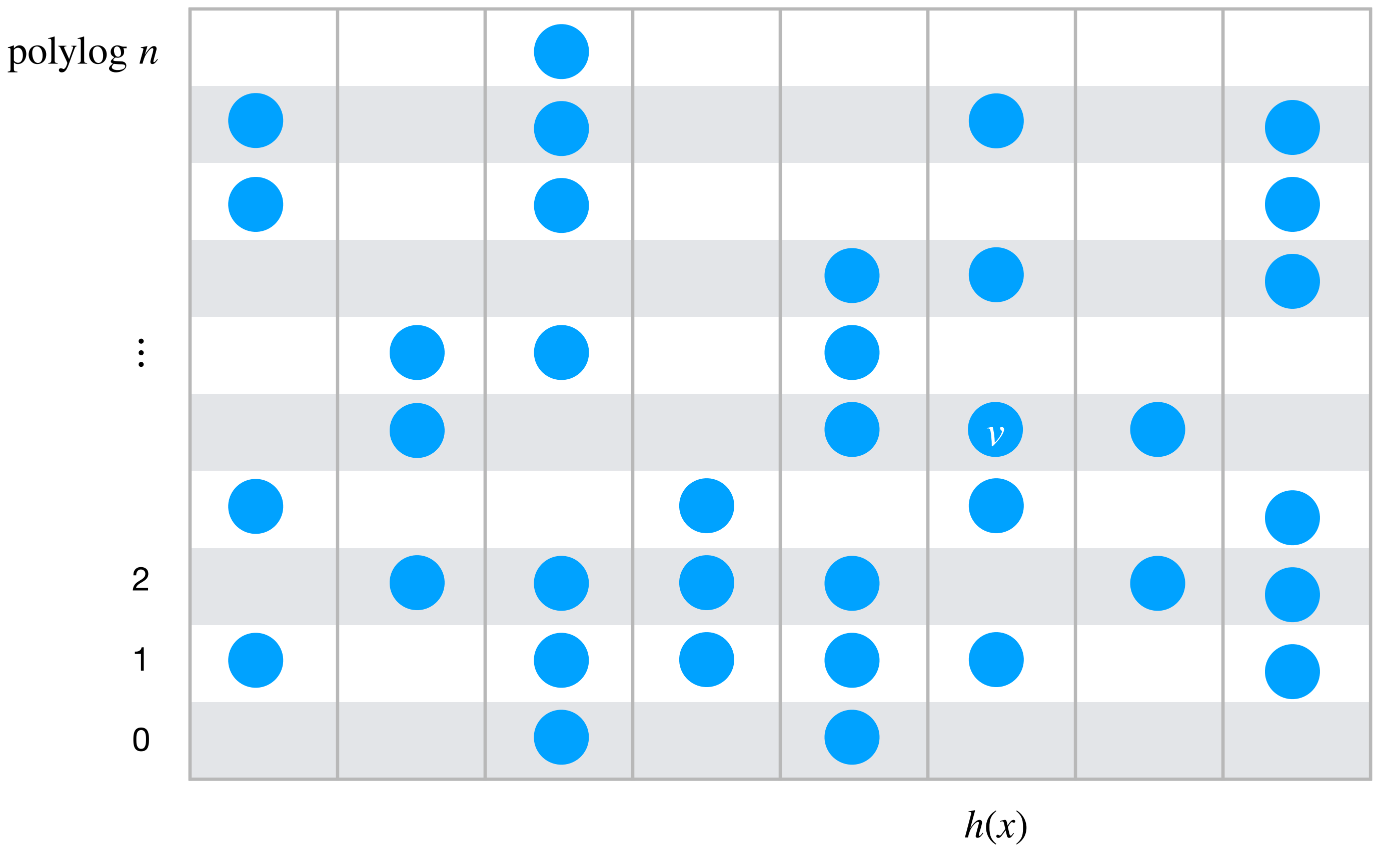
A simple Dereference Table



Array of size: $(1 + 1/\log n)n$
Bins of size: polylog n

Inserting an item x, v:
- hash into some bin $h(x)$
- find an empty slot and store v
- return the offset of the slot

A simple Dereference Table

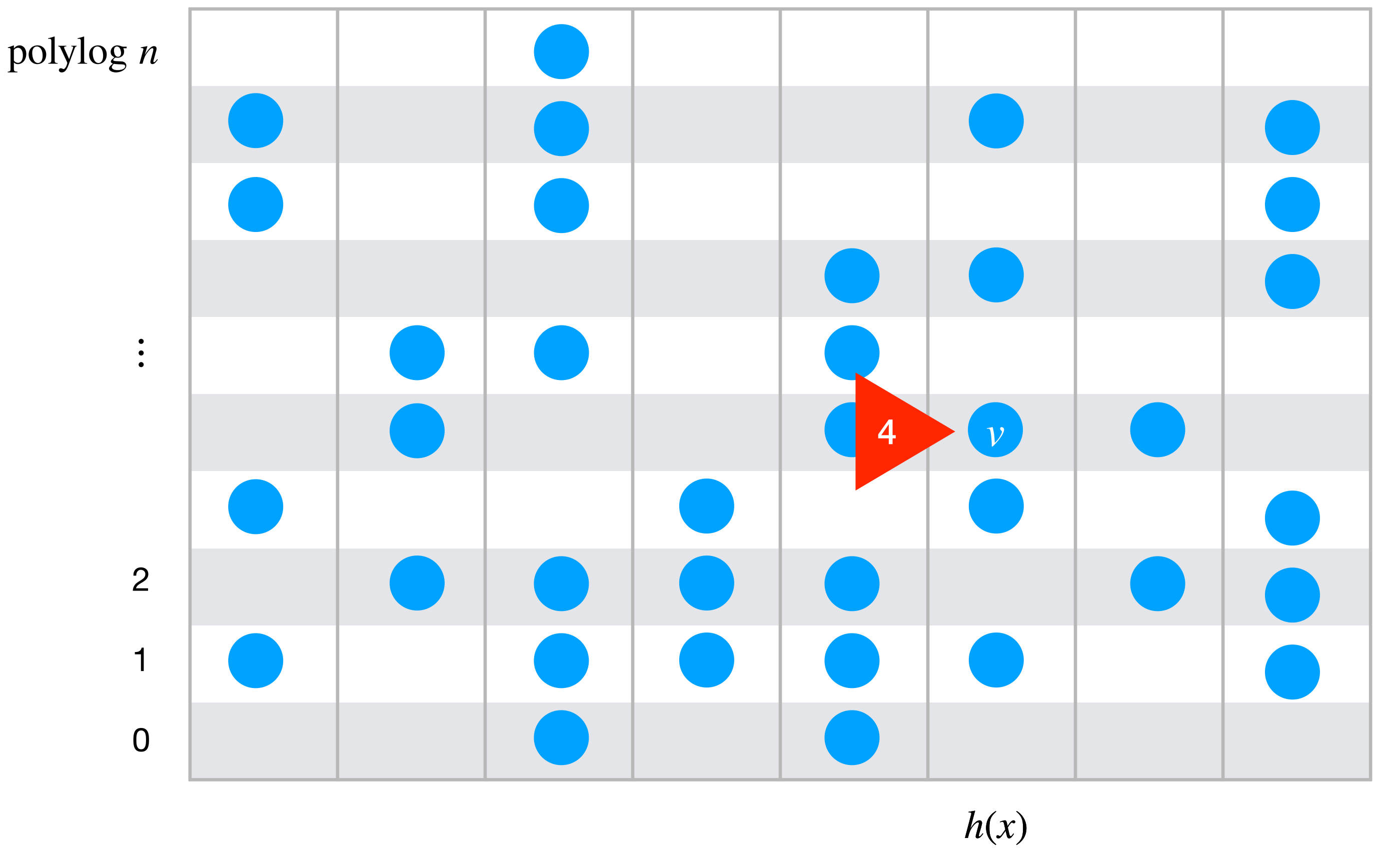


Array of size: $(1 + 1/\log n)n$
Bins of size: polylog n

Inserting an item x, v**:**

hash into some bin $h(x)$
find an empty slot and store v
return the offset of the slot

A simple Dereference Table



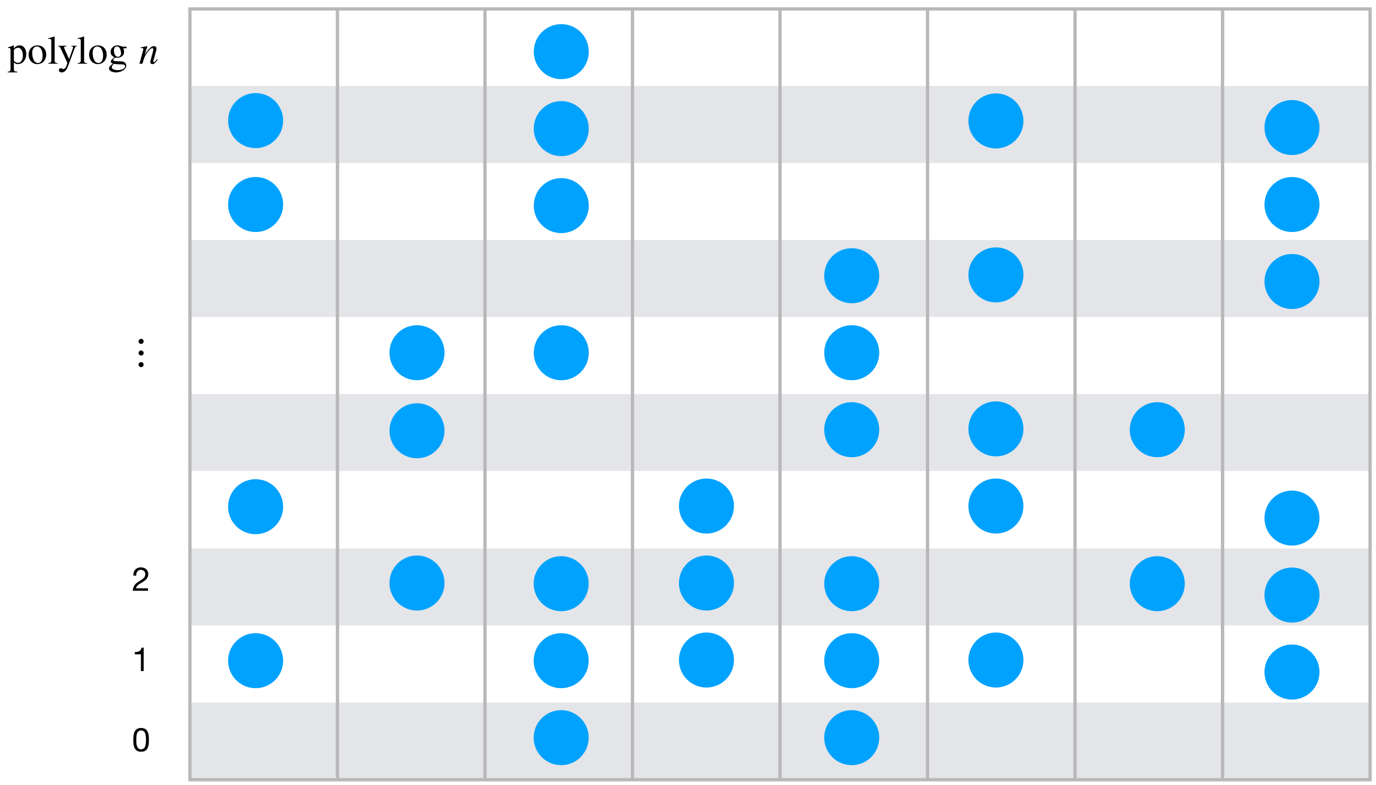
return: 4

Array of size: $(1 + 1/\log n)n$
Bins of size: polylog n

Inserting an item x, v**:**

hash into some bin $h(x)$
find an empty slot and store v
return the offset of the slot

A simple Dereference Table

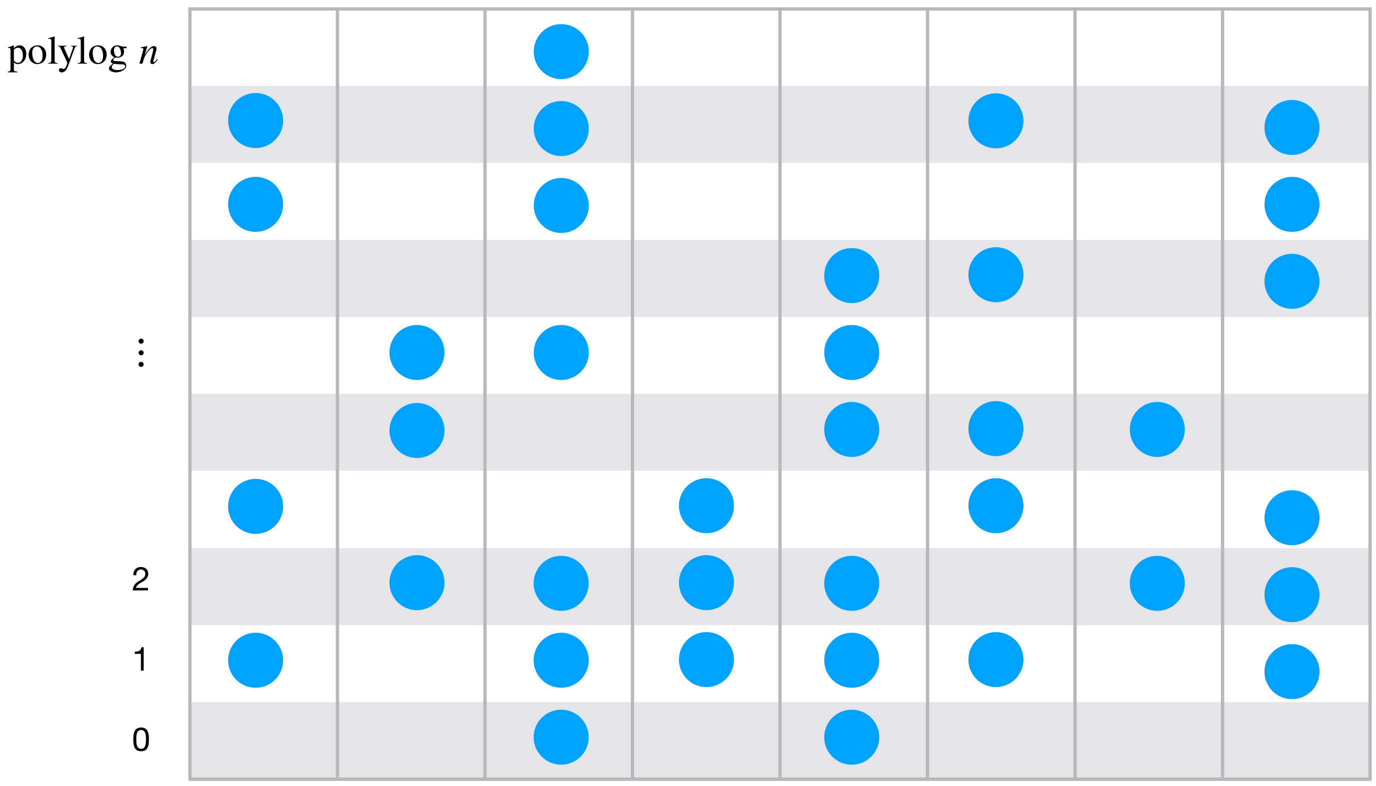


Array of size: $(1 + 1/\log n)n$
Bins of size: polylog n

Inserting an item x, v**:**
hash into some bin $h(x)$
find an empty slot and store v
return the offset of the slot

Key observation: n balls can be thrown into this array without a bin overflowing w.h.p.

A simple Dereference Table



Query(x, ptr)

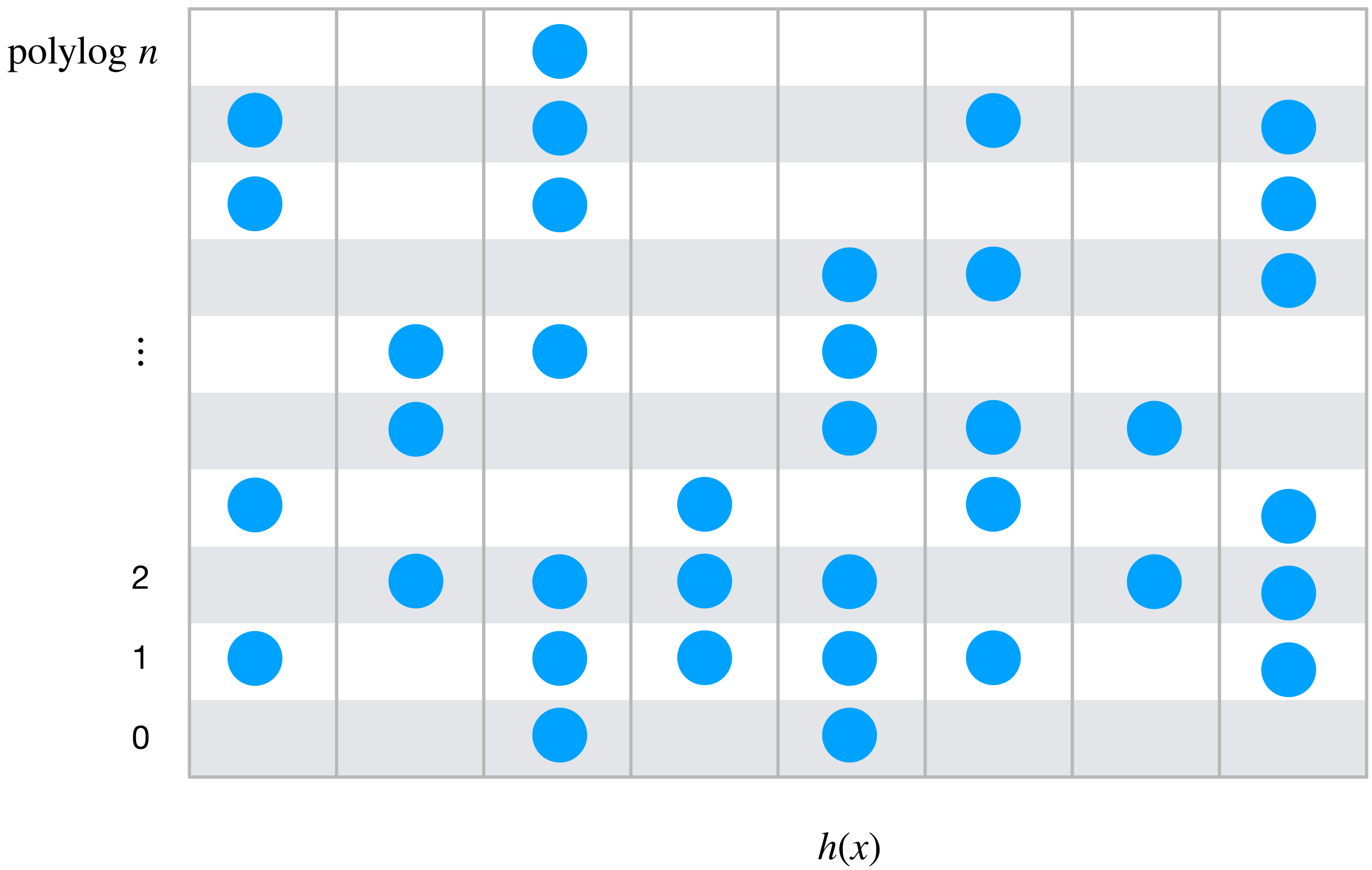
Array of size: $(1 + 1/\log n)n$
Bins of size: polylog n

Inserting an item x, v**:**
hash into some bin $h(x)$
find an empty slot and store v
return the offset of the slot

Key observation: n balls can be thrown into this array without a bin overflowing w.h.p.

Finding item x**, ptr:**
go to ptr'th item in $h(x)$
return value stored there

A simple Dereference Table



Query(x, ptr)

Array of size: $(1 + 1/\log n)n$
Bins of size: polylog n

Inserting an item x, v:

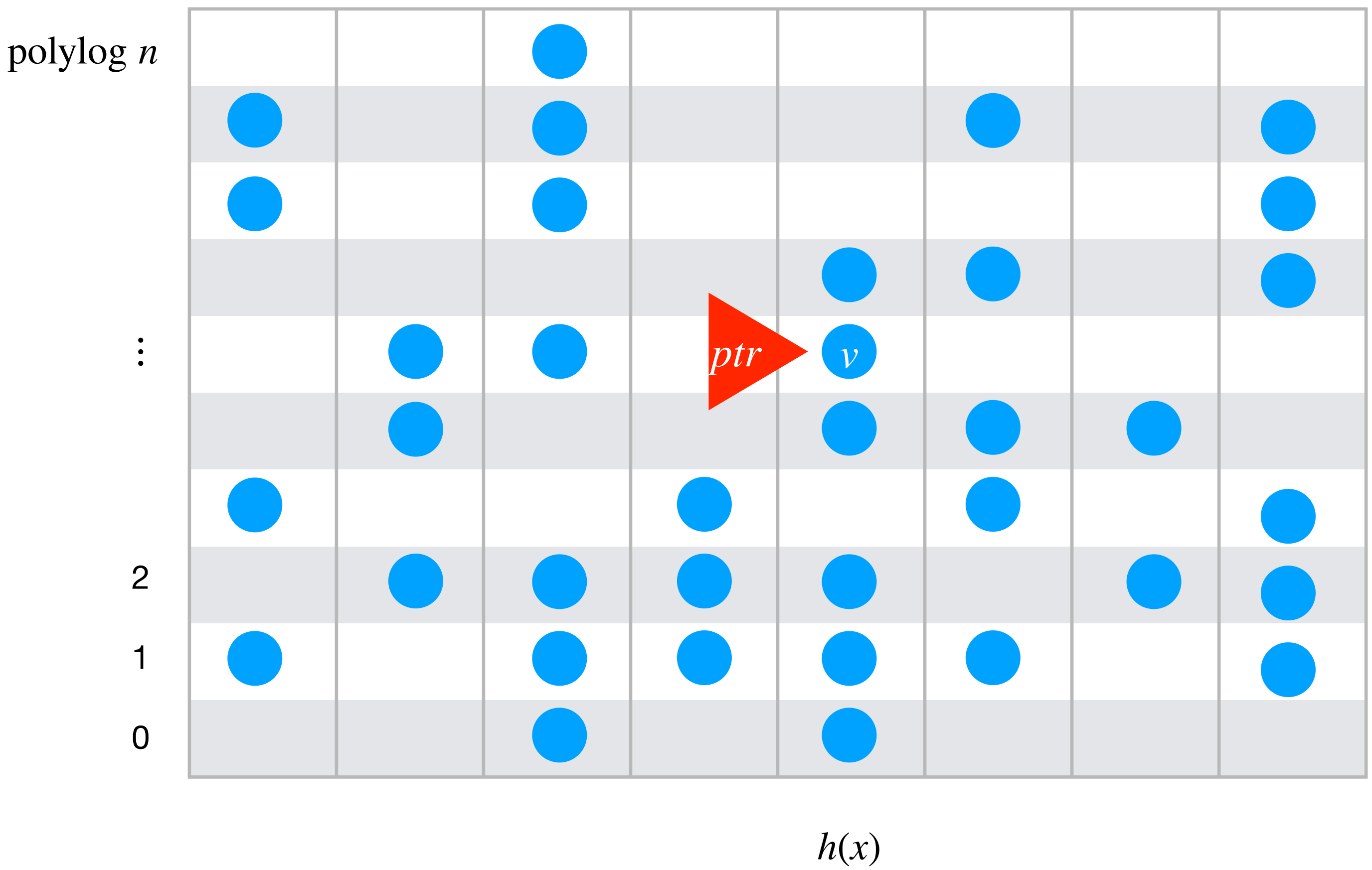
hash into some bin $h(x)$
find an empty slot and store v
return the offset of the slot

Key observation: n balls can be thrown into this array without a bin overflowing w.h.p.

Finding item x, ptr:

go to ptr'th item in $h(x)$
return value stored there

A simple Dereference Table



Query(x, ptr)

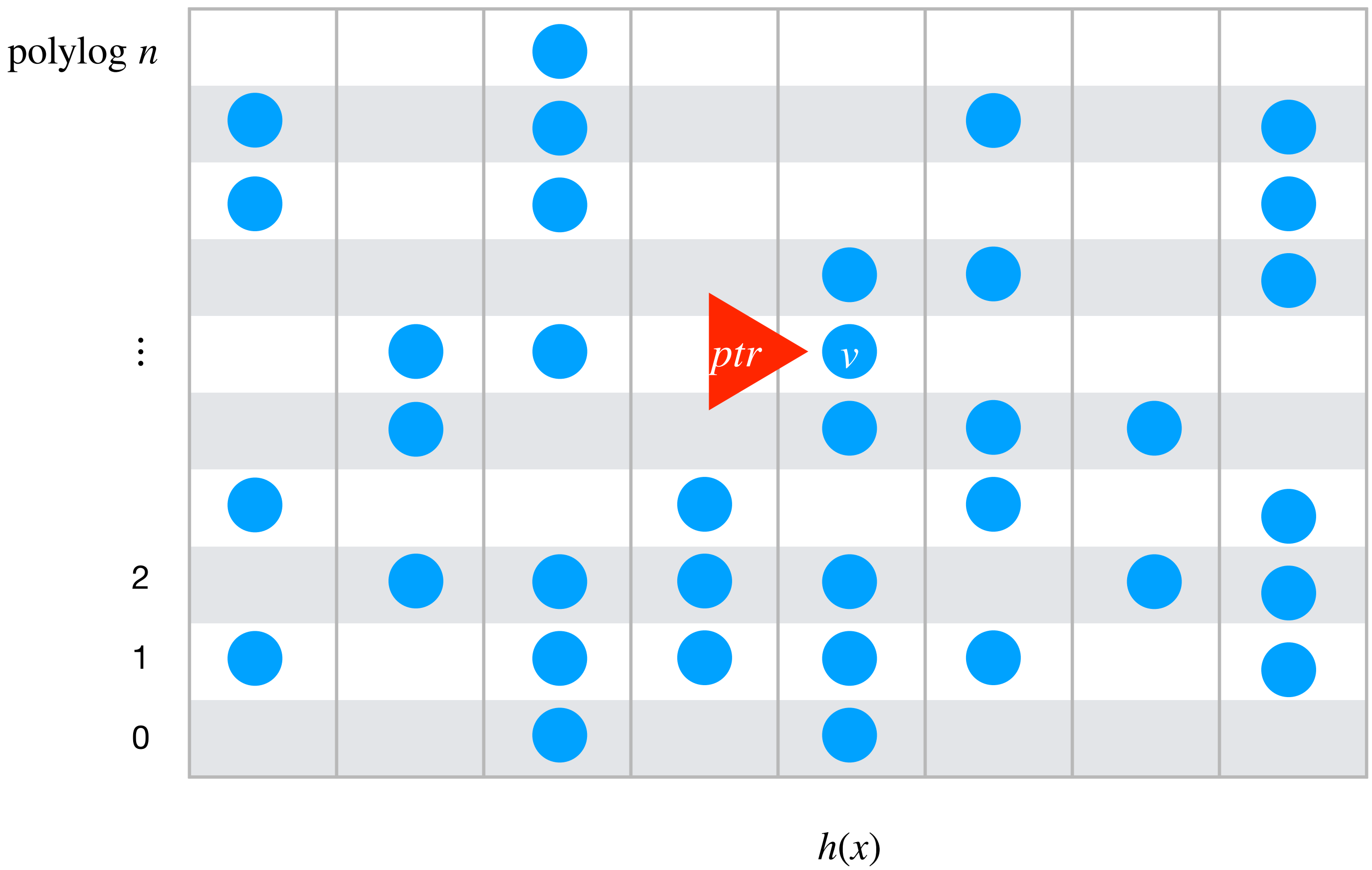
Array of size: $(1 + 1/\log n)n$
Bins of size: polylog n

Inserting an item x, v:
hash into some bin $h(x)$
find an empty slot and store v
return the offset of the slot

Key observation: n balls can be thrown into this array without a bin overflowing w.h.p.

Finding item x, ptr:
go to ptr'th item in $h(x)$
return value stored there

A simple Dereference Table



Query(x, ptr) returns v

Array of size: $(1 + 1/\log n)n$
Bins of size: polylog n

Inserting an item x, v**:**

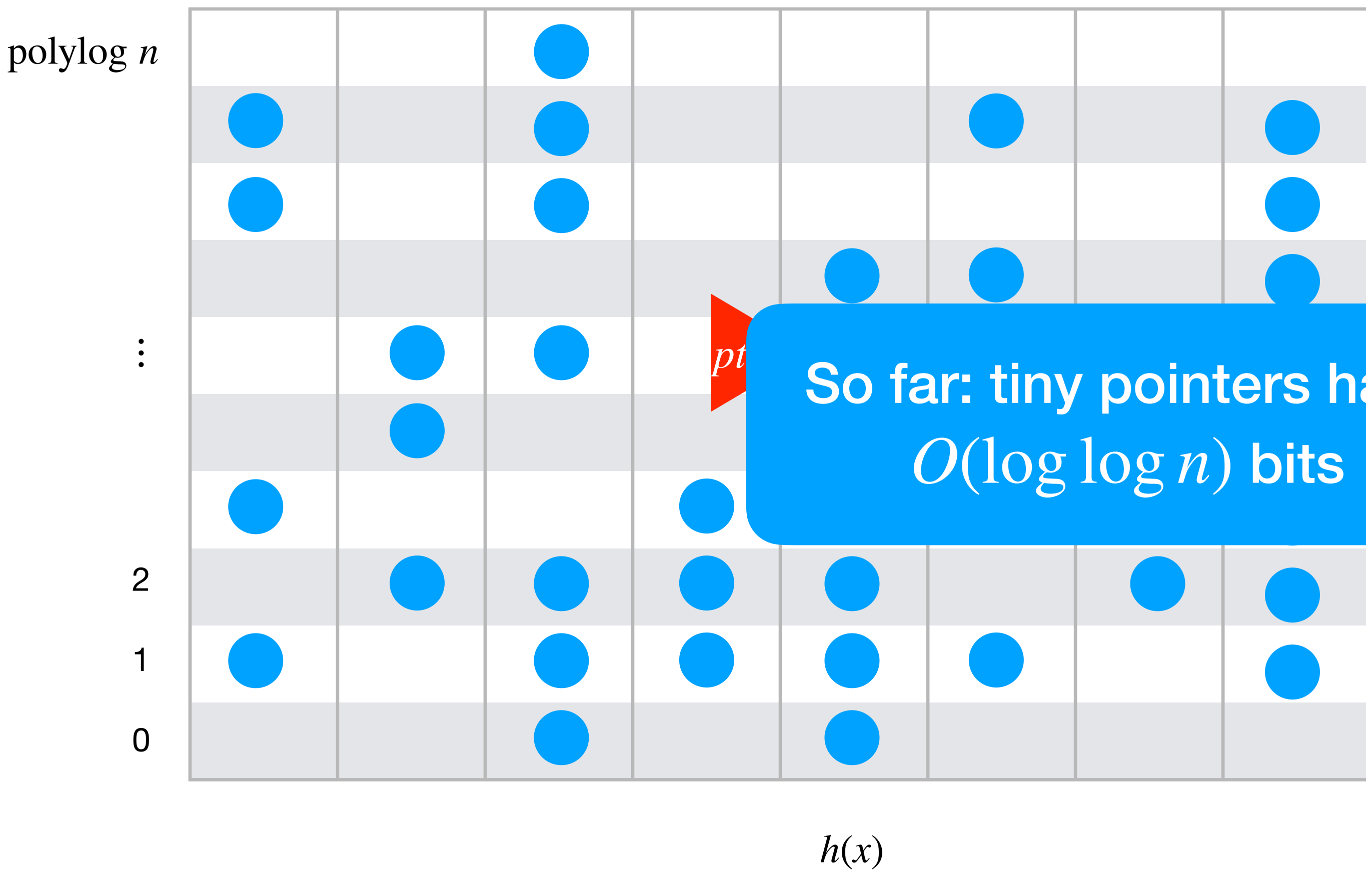
hash into some bin $h(x)$
find an empty slot and store v
return the offset of the slot

Key observation: n balls can be thrown into this array without a bin overflowing w.h.p.

Finding item x**, ptr:**

go to ptr'th item in $h(x)$
return value stored there

A simple Dereference Table



Query(x, ptr) returns v

So far: tiny pointers have $O(\log \log n)$ bits

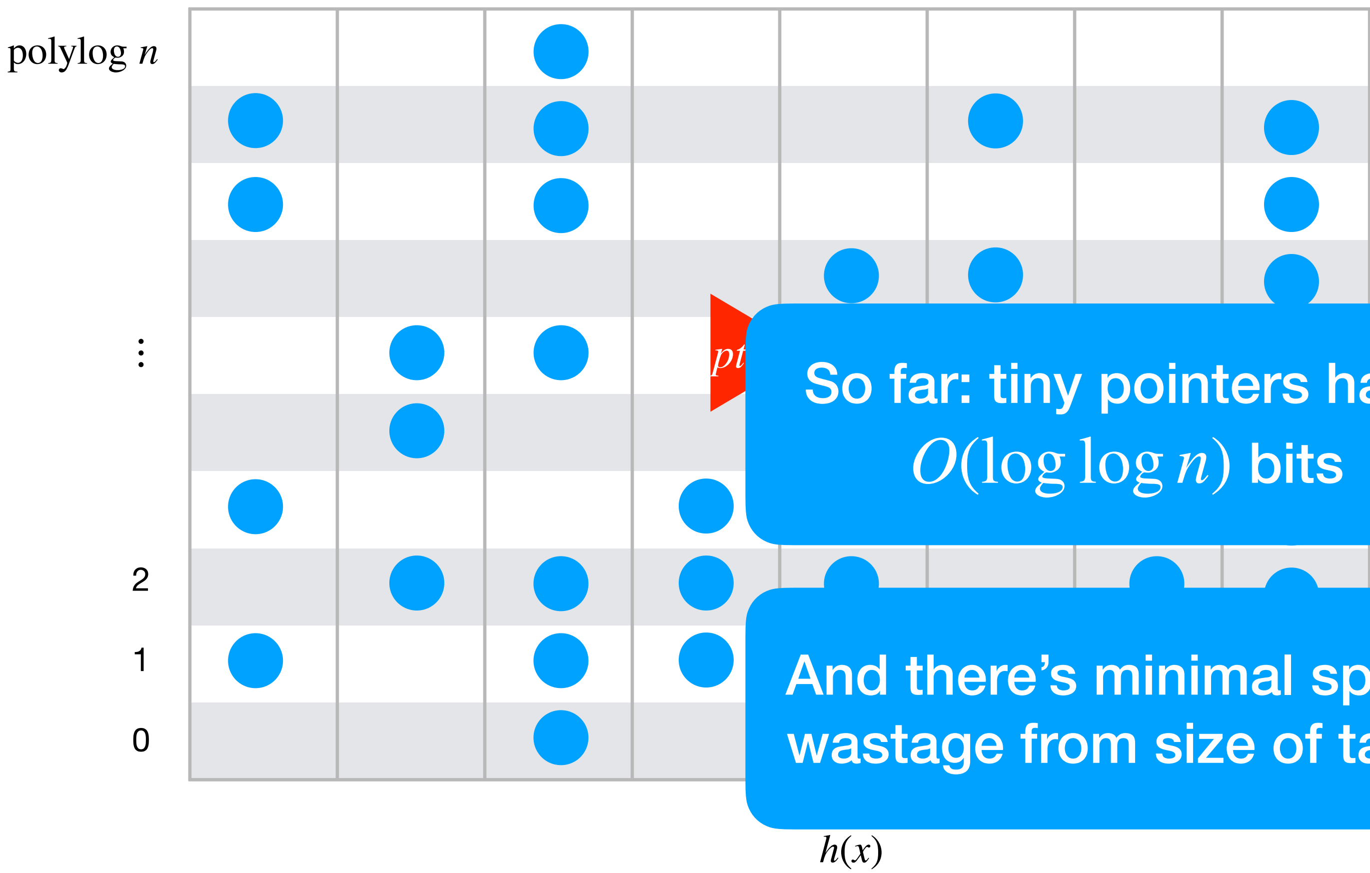
Array of size: $(1 + 1/\log n)n$
Bins of size: polylog n

Inserting an item x, v:
ash into some bin $h(x)$
nd an empty slot and store v
turn the offset of the slot

y observation: n balls can be thrown into this array without a bin overflowing w.h.p.

Finding item x, ptr:
go to ptr'th item in $h(x)$
return value stored there

A simple Dereference Table



Query(x, ptr) returns v

So far: tiny pointers have $O(\log \log n)$ bits

And there's minimal space wastage from size of table

Array of size: $(1 + 1/\log n)n$
Bins of size: polylog n

Inserting an item x, v:
ash into some bin $h(x)$
d an empty slot and store v
turn the offset of the slot

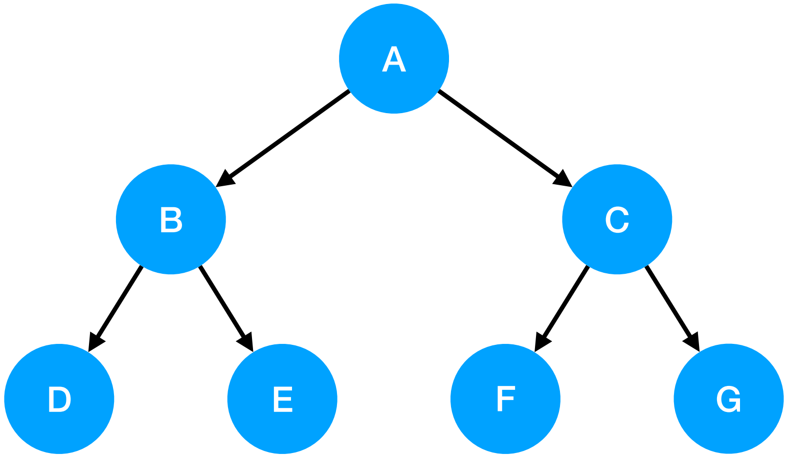
y observation: n balls can be thrown into this array without a bin flowing w.h.p.

ing item x, ptr:
go to ptr'th item in $h(x)$
return value stored there

BSTs through tiny pointers (take 1)

Theorem: Any rotation-based BST can be implemented with

- $O(1)$-time overhead per operation
- Space $nw + O(n \log \log n)$



Revisiting Dereference Tables

Pointers	Hash Table	Dereference Table
Malloc→ptr; *ptr = value	Insert(key, value)	Insert(key, value)→tiny-ptr
*ptr = value	Update(key, value)	Update(key, value, tiny-ptr)
*ptr	Query(key)	Query(key, tiny-ptr)
Free(ptr)	Delete(key)	Delete(key, tiny-ptr)
Only positive queries	Positive and negative queries	Only positive queries

Revisiting Dereference Tables

Pointers	Hash Table	Dereference Table
Malloc→ptr; *ptr = value	Insert(key, value)	Insert(key, value)→tiny-ptr
*ptr = value	Update(key, value)	Update(key, value, tiny-ptr)
*ptr	Query(key)	Query(key, tiny-ptr)
Free(ptr)	Delete(key)	Delete(key, tiny-ptr)
Only positive queries	Positive and negative queries	Only positive queries
Stable	Stable or not	Stable

Stable = Slots (or contents) can't move once slots have allocated

Revisiting Dereference Tables

Pointers	Hash Table	Dereference Table
Malloc→ptr; *ptr = value	Insert(key, value)	Insert(key, value)→tiny-ptr
*ptr = value	Update(key, value)	Update(key, value, tiny-ptr)
*ptr	Query(key)	Query(key, tiny-ptr)
Free(ptr)	Delete(key)	Delete(key, tiny-ptr)
Only positive queries	Positive and negative queries	Only positive queries
Stable	Stable or not	Stable
Arbitrary Associativity	Arbitrary Associativity	Low associativity

Associativity: # of locations and item can go

Size of hash and dereference tables:

Hash tables:

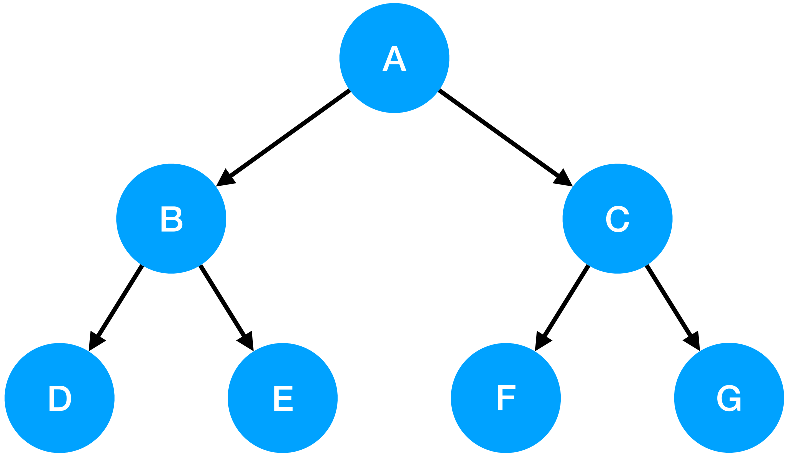
- [Bender, FC, Kuszmaul, Kuszmaul, Liu STOC 22]: There is a hash table with $O(1)$-time lookups such that
  - For any integer $d \geq 1$, insertions/deletions take time d w.h.p.
  - Space is $\log \binom{u}{n} + O(n \log^{(d+1)} n) = n \log u/n + +O(n \log^{(d+1)} n)$ bits
- [Li, Liang, Yu, Zhou FOCS 23]: This is optimal for $O(1)$-time lookup hash tables

Dereference tables:

- [Bender, Conway, FC, Kuszmaul, Tagliavini SODA 23]: The optimal dereference tables with $O(1)$-time looks with $(1+\epsilon)n$ cells has associativity $O(\varepsilon^{O(1)} \log \log n)$
  - So tiny pointers have $O(\log \log \log n + \log \varepsilon^{-1})$ bits; and this is tight

BST through Tiny Pointers (take 2)

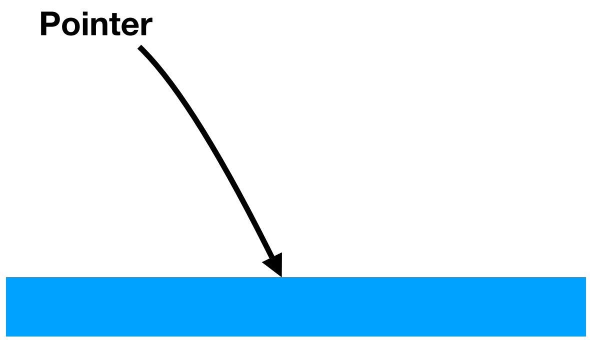
Better theorem: Any rotation-based BST can be implemented with $O(1)$-time overhead and space $nw + O(n \log \log \log n)$



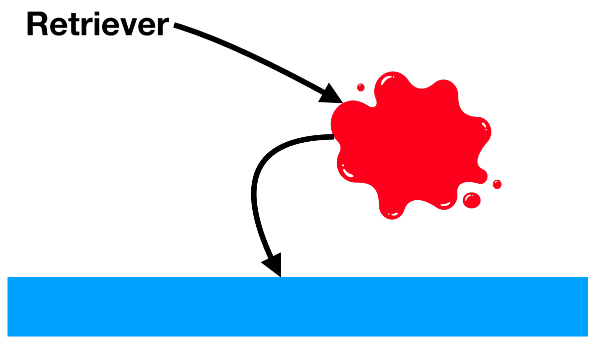
From Pointers to Retrievers

From Pointers to Retrievers

Pointers vs Retrievers



queries are computed from
key, ptr, random bits



queries gets to use an
auxiliary data structure
in addition to key, ptr, random bits

We’ll see why we care about the difference

Tiny Pointers Bounds

Theorem: Can we build a dereference table on w-bit items, where:

- Each operation takes $O(1)$ time
- Table size is $(1 + \varepsilon)nw$ bits
- Pointer size is $O(\log \varepsilon^{-1} + \log \log \log n)$ bits

Theorem: Matching lower bound. Tradeoff curve is tight!

Better Tiny Pointers Bounds

Theorem: Can we build a dereference table on w-bit items, where:

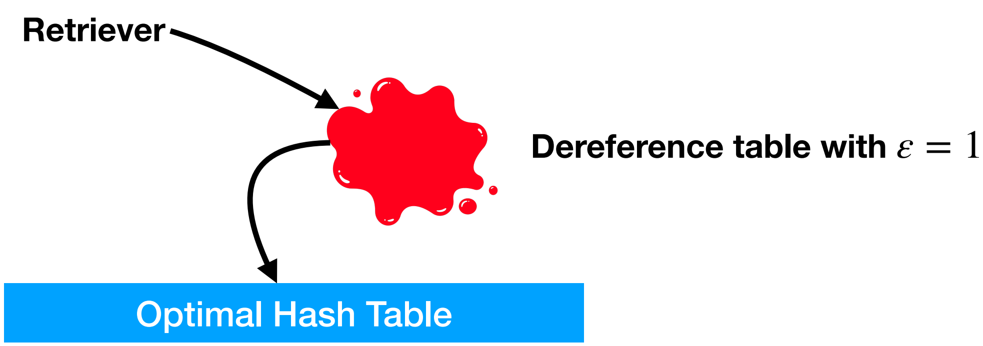
- Each operation takes $O(1)$ time
- Table size is $(1 + \varepsilon)nw$ bits
- Expected pointer size is $O(\log \varepsilon^{-1} + \log \log \log n)$ bits

Theorem: Matching lower bound. Tradeoff curve is tight!

Tiny Retriever Bounds

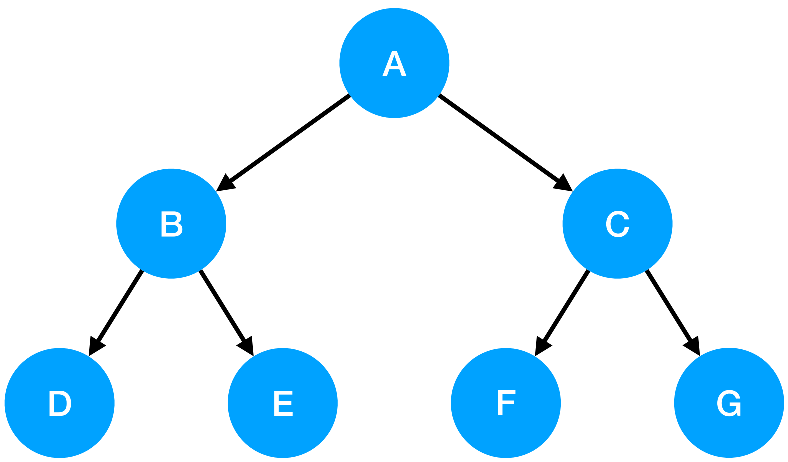
Theorem: Can we build a dereference table on w-bit items, where:

- Each operation takes $O(1)$ time
- Table size is $nw + O(n \log \log \cdots \log n)$ bits $= nw + O(n \log^{(k)} n)$ bits, for any constant k
- Expected pointer size is $O(1)$ bits



BSTs through tiny retrievers

Theorem: Any rotation-based BST can be implemented with $O(1)$-time overhead and space $nw + O(n \log\log \cdots \log\log n)$



Succinct trees: what’s known?

Previous literature replaces pointers with other structure with $2n + o(n)$ bits

- [Cordova, Navarro. TCS ’16][Davoodi et al. MCS ‘217][Farzan, Munro. ICALP ’11], ...
- These structures are small but slow

	Previous	New
Space	Very very very small $nw + 2n + o(n)$ bits	Very very small $nw + O(n \log\log \ldots \log n)$ bits
Time	Polylogarithmic (or more) overhead for many operations	O(1) time overhead for all operations

Tiny Pointers/Retriever solve many open problems

Space efficient stable dictionaries

- [Demaine, Meyer auf der Heide, Pagh, Pătraşcu ’06], [Sanders ’18], [Bender et al. ’21]

Space efficient dictionaries with variable-size value

- [Arbitman, Naor, Segev ’10], [Bercea, Even ’14], [Bercea, Even ’19]

The internal-memory stash problem

- [Larson, Kaijla ’84], [Gonnet, Larson ’88], [Larson ’88]

Some use pointers, some retrievers

Tiny Pointer Open Problem

How do you use them in graphs?

- This is hard because multiple “owners of pointers” need to be able to point to the same location
- Maybe this is as hard as the general pointer problem?

And now for something completely different

Low Associativity Paging

Paging Problem

Classic online problem

- Cache of size m
- Sequence $p_1, p_2, \ldots$ of page requests
- Cost model for an algorithm to service page p_i is:

$$cost(p_i) = \begin{cases} 0 & \textbf{if } p_i \textbf{ is in the cache} \\ 1 & \textbf{choose a page to evict and store } p_i \end{cases}$$

Famously [Seator, Tarjan 85], various page eviction algorithms are 2-competitive with resource augmentation of 2.

- But this result only applies to ***fully associative*** caches

What about paging on limited associativity caches?

Paging on low-associativity caches

Almost all caches in the world have low associativity

- But almost all theoretical results only apply to fully associative caches

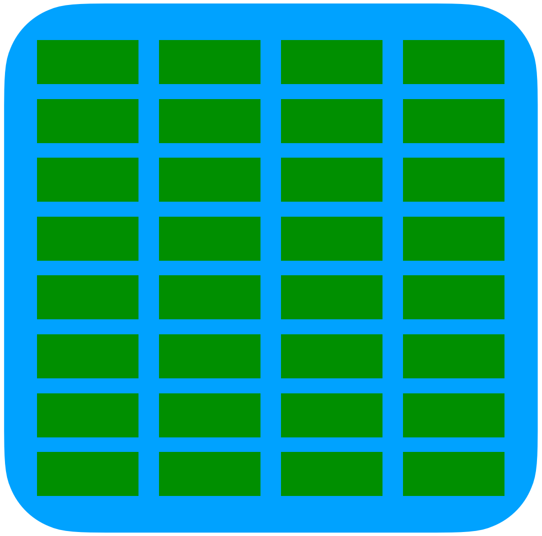
Questions:

- How does paging work on low-associativity caches?
- Can we design a very low associativity cache where we can page very well?

Associativity of RAM or of Page Evictions?

Normally we think of the associativity of a cache as a hardware constraint

But it can also be a feature of the page eviction algorithm

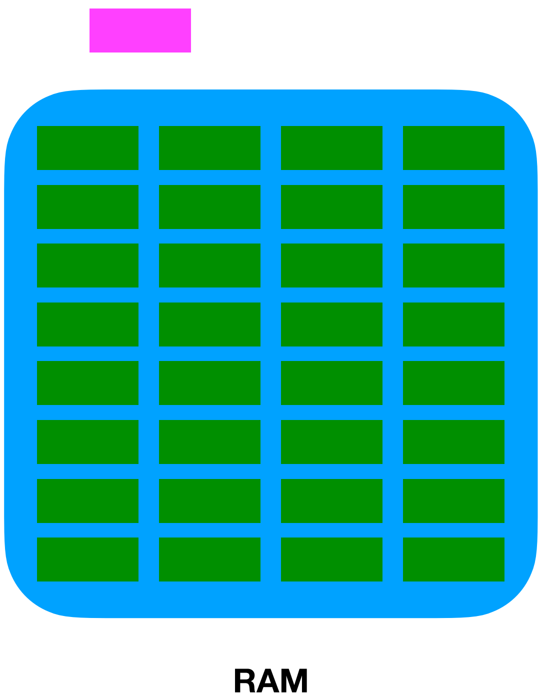


RAM

Associativity of RAM or of Page Evictions?

Normally we think of the associativity of a cache as a hardware constraint

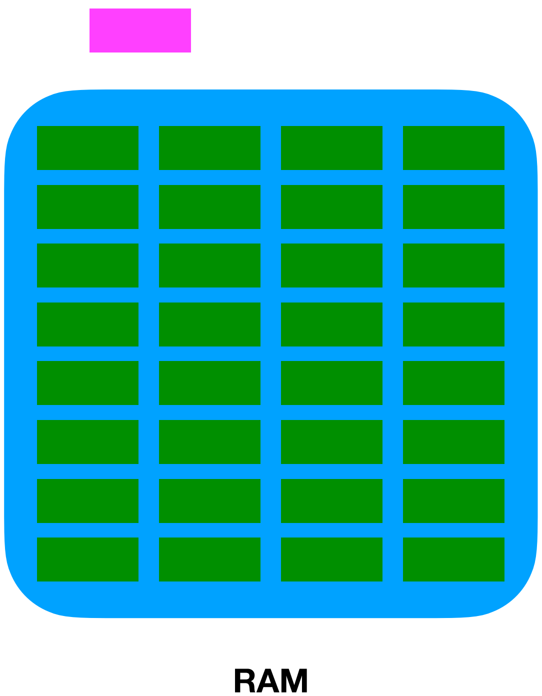
But it can also be a feature of the page eviction algorithm



Associativity of RAM or of Page Evictions?

Normally we think of the associativity of a cache as a hardware constraint

But it can also be a feature of the page eviction algorithm



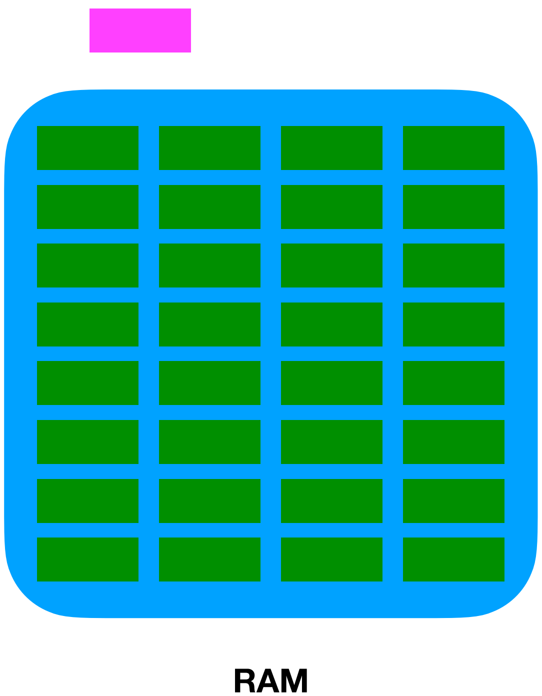
We'll see examples where the each matters

… oh, and we'll see what this has to do with tiny pointers

Associativity of Page Evictions

Normally we think of the associativity of a cache as a hardware constraint

But it can also be a feature of the page eviction algorithm



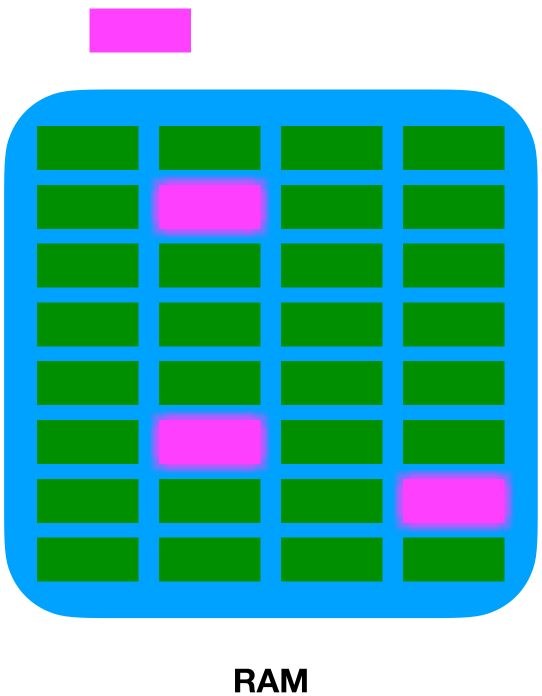
In either case, the ***associativity*** of a page eviction algo is

- max # of location a page can map to

Associativity of Page Evictions

Normally we think of the associativity of a cache as a hardware constraint

But it can also be a feature of the page eviction algorithm



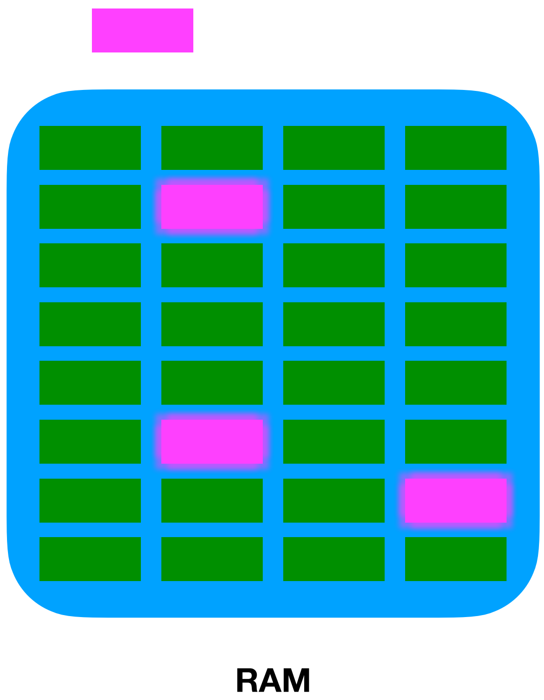
In either case, the ***associativity*** of a page eviction algo is

- max # of location a page can map to
- Here pink maps to three locations
- And if we ever see pink again, it maps to the ***same three locations***

Associativity of Page Evictions

Normally we think of the associativity of a cache as a hardware constraint

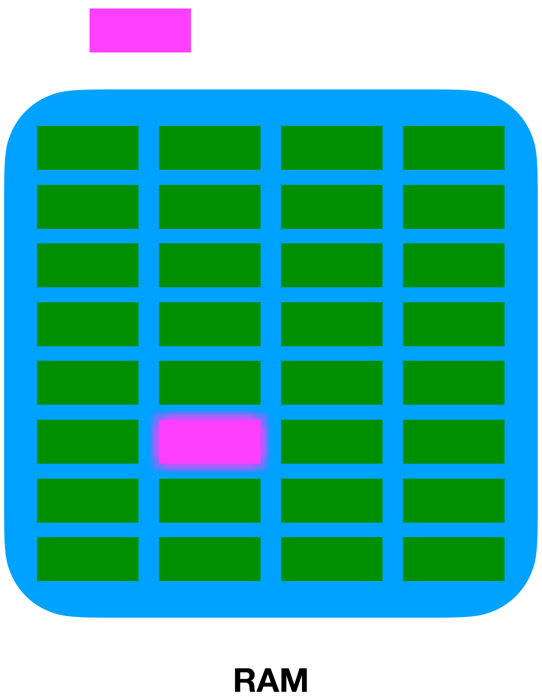
But it can also be a feature of the page eviction algorithm



The page eviction algorithm may only evict one of the three pink pages

Associativity of 1

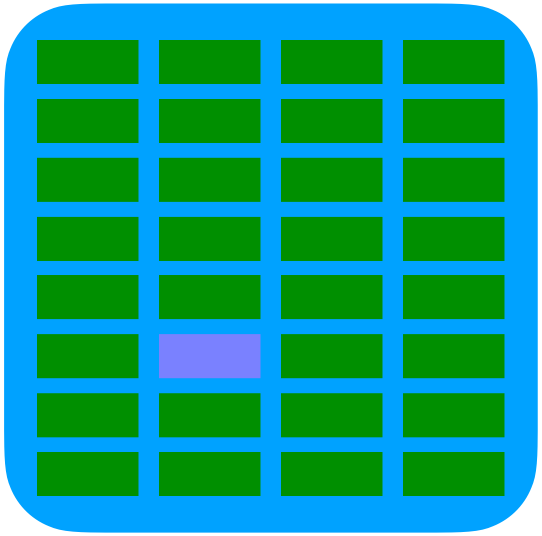
Taking page eviction associativity to the extreme



There will be lots of contention for the same slots

Associativity of 1

Taking page eviction associativity to the extreme



There will be lots of contention for the same slots

- So there will be lots and lots of paging

How much memory would we need to match the paging cost of a fully associative?

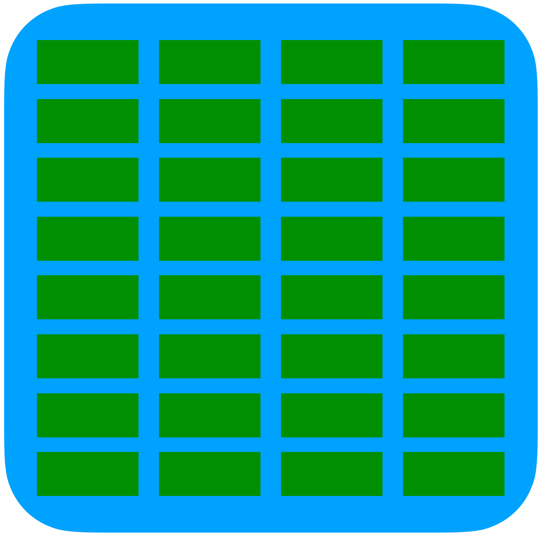
- $\Omega(m^2)$ by birthday paradox

How low can you go?

How low can associativity be without blowing up the paging cost?
And without blowing up the size of the cache?

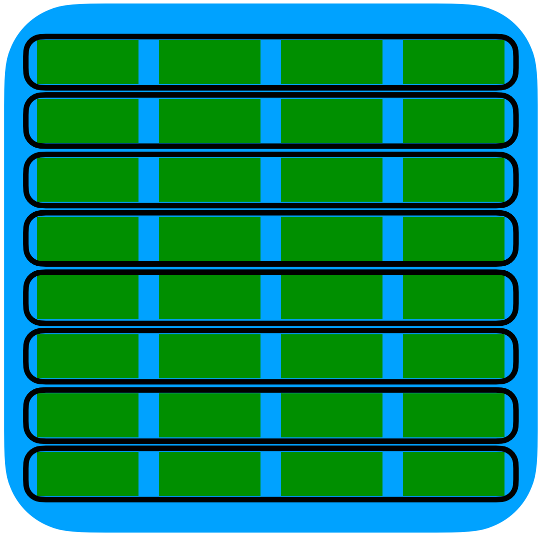
Key concept: set vs general associative caches

A ***set associative cache*** partitions the cache



Key concept: set vs general associative caches

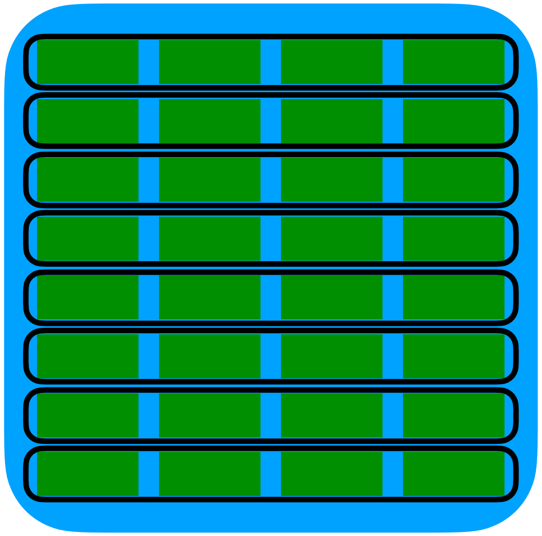
A ***set associative cache*** partitions the cache



Key concept: set vs general associative caches

A ***set associative cache*** partitions the cache

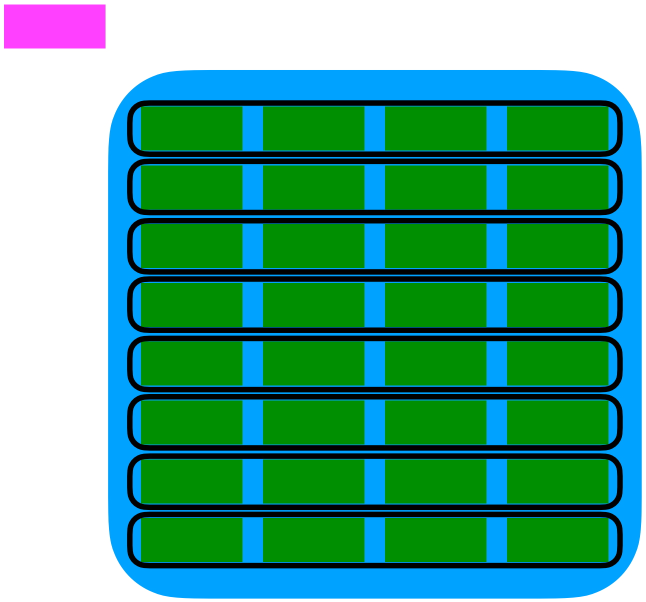
- Any page maps to a single partition
- It can go to any position in that partition



Key concept: set vs general associative caches

A ***set associative cache*** partitions the cache

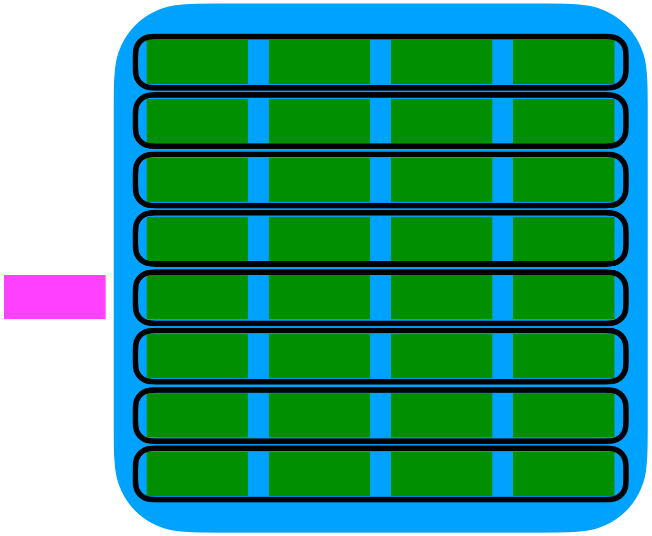
- Any page maps to a single partition
- It can go to any position in that partition



Key concept: set vs general associative caches

A ***set associative cache*** partitions the cache

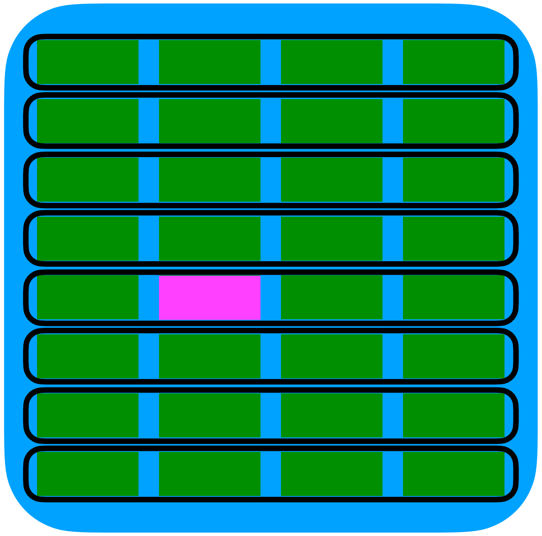
- Any page maps to a single partition
- It can go to any position in that partition



Key concept: set vs general associative caches

A ***set associative cache*** partitions the cache

- Any page maps to a single partition
- It can go to any position in that partition

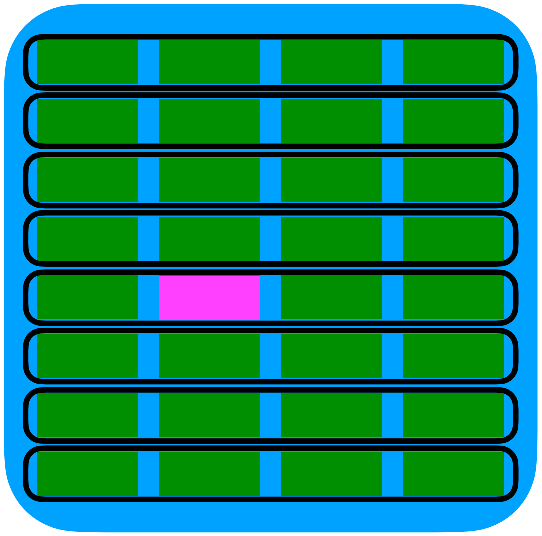


The associativity of such a cache is the size of the largest partition — usually all partitions have the same size

Key concept: set vs general associative caches

A ***set associative cache*** partitions the cache

- Any page maps to a single partition
- It can go to any position in that partition



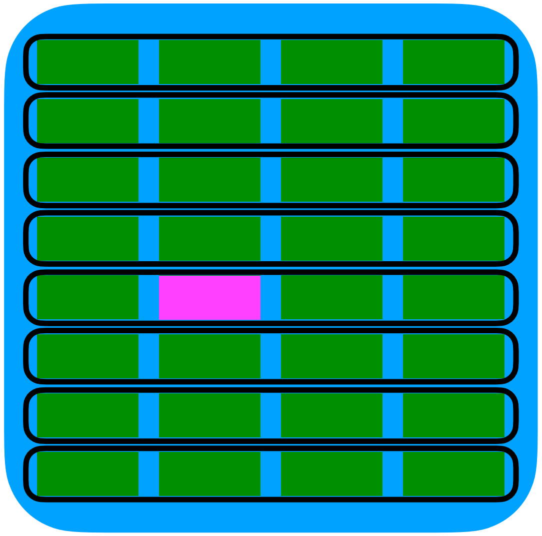
The associativity of such a cache is the size of the largest partition — usually all partitions have the same size

In a general d-associative cache, each page can map to up to d pages (not necessarily a partition)

Key concept: set vs general associative caches

A ***set associative cache*** partitions the cache

- Any page maps to a single partition
- It can go to any position in that partition



Almost all hardware is either fully associative or set associative

So how does the associativity of a set-associative cache affect the amount of paging?

How should we measure the paging?

Competitive analysis

Definition: $\mathscr{A}$ is c-competitive with $\mathscr{B}$ with high probability, using r-resource augmentation, on request sequences of length ℓ, if

w.h.p., for every request sequence σ with $|\sigma| \leq \ell$, where $k = r \cdot k'$.

Competitive analysis

Definition: $\mathscr{A}$ is c-competitive with $\mathscr{B}$ with high probability, using r-resource augmentation, on request sequences of length ℓ, if

$$C(\mathscr{A}_k, \sigma) \leq c \cdot C(\mathscr{B}_{k'}, \sigma) + O(1)$$

Paging cost of $\mathscr{A}$ with cache size k

Paging cost of $\mathscr{B}$ with cache size k'

w.h.p., for every request sequence σ with $|\sigma| \leq \ell$, where $k = r \cdot k'$.

Related work

So how does the associativity of a set-associative cache affect the amount of paging?

Competitive analysis

Not set associative

- Fully associative algorithms vs. fully associative OPT
  - [Sleator & Tarjan ’85, Fiat et al. ’91, Young ’94, Dorrigiv et al. ’09, …]

Related work

So how does the associativity of a set-associative cache affect the amount of paging?

Competitive analysis

- Fully associative algorithms vs. fully associative OPT
  - [Sleator & Tarjan ’85, Fiat et al. ’91, Young ’94, Dorrigiv et al. ’09, …]
- Limited associative algorithms vs. limited associative OPT
  - [Brehob et al. ’01, Peserico ’03, Mendel & Seiden ’04, Buchbinder et al. ’14]

This baseline doesn’t help to understand the effect of reducing associativity

Related work

So how does the associativity of a set-associative cache affect the amount of paging?

Competitive analysis

- Fully associative algorithms vs. fully associative OPT
  - [Sleator & Tarjan ’85, Fiat et al. ’91, Young ’94, Dorrigiv et al. ’09, …]
- Limited associative algorithms vs. limited associative OPT
  - [Brehob et al. ’01, Peserico ’03, Mendel & Seiden ’04, Buchbinder et al. ’14]

Competitive analysis assuming σ is distributional

Real workloads can be adversarial

- Set-associative algorithms vs. fully associative algorithms
  - [Smith ’54, Smith ’55, Rao ’78]

Related work

So how does the associativity of a set-associative cache affect the amount of paging?

Related work

So how does the associativity of a set-associative cache affect the amount of paging?

Theorems in prior works are not tight enough

Related work

So how does the associativity of a set-associative cache affect the amount of paging?

Theorems in prior works are not tight enough

Traditional competitive analyses:

- Vs. OPT, $O(1)$-competitive, $O(1)$-resource augmentation

Related work

So how does the associativity of a set-associative cache affect the amount of paging?

Theorems in prior works are not tight enough

Traditional competitive analyses:

- Vs. OPT, $O(1)$-competitive, $O(1)$-resource augmentation

We need:

- Vs. LRU, $(1 + o(1))$-competitive, $(1 + o(1))$-resource augmentation

Related work

So how does the associativity of a set-associative cache affect the amount of paging?

Theorems in prior works are not tight enough

Traditional competitive analyses:

- Vs. OPT, $O(1)$-competitive, $O(1)$-resource augmentation

We need:

Use practical algorithm as baseline

- Vs. LRU, $(1 + o(1))$-competitive, $(1 + o(1))$-resource augmentation

Related work

So how does the associativity of a set-associative cache affect the amount of paging?

Theorems in prior works are not tight enough

Traditional competitive analyses:

- Vs. OPT, $O(1)$-competitive, $O(1)$-resource augmentation

We need:

Use practical algorithm as baseline

- Vs. LRU, $(1 + o(1))$-competitive, $(1 + o(1))$-resource augmentation

Tighter results

A threshold phenomenon for set-associative caches

A threshold phenomenon for set-associative caches

Theorem: If $d = \omega(\log m)$, then d-way set-associative LRU on a cache of size m is 1-competitive with fully associative LRU w.h.p., using $(1 + o(1))$ -resource augmentation. [Bender, Das, FC, Tagliavini SPAA '23]

A threshold phenomenon for set-associative caches

Theorem: If $d = \omega(\log m)$, then d-way set-associative LRU on a cache of size m is 1-competitive with fully associative LRU w.h.p., using $(1 + o(1))$ -resource augmentation. [Bender, Das, FC, Tagliavini SPAA '23]

Theorem: If $d = o(\log m)$, then d-way set-associative LRU on a cache of size m is $\omega(1)$-competitive with fully associative LRU, using ***any polynomial*** resource augmentation. [Bender, Das, FC, Tagliavini SPAA '23]

A threshold phenomenon for set-associative caches

<u>Theorem:</u> If $d = \omega(\log m)$, then d-way set-associative LRU on a cache of size m is 1-competitive with fully associative LRU w.h.p., using $(1 + o(1))$-resource augmentation. [Bender, Das, FC, Tagliavini SPAA ’23]

<u>Theorem:</u> If $d = o(\log m)$, then d-way set-associative LRU on a cache of size m is $\omega(1)$-competitive with fully associative LRU, using ***<u>any polynomial</u>*** resource augmentation. [Bender, Das, FC, Tagliavini SPAA ’23]

Takehome message: Current set-associative cache (IRL) are a little too small

Enough about existing caches

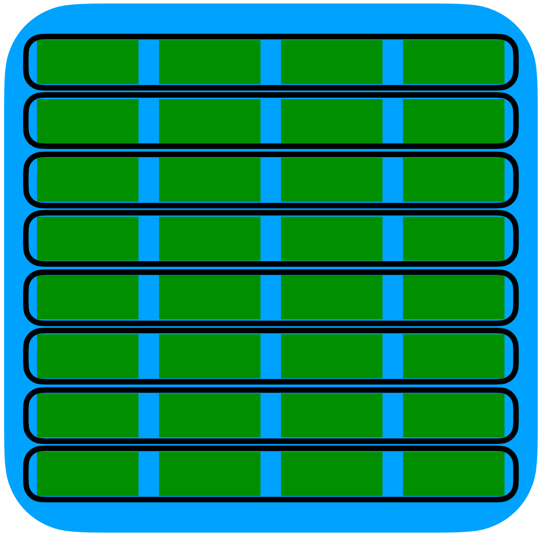
What if we can design a limited associativity eviction policy?

Spoiler: this will get us back to tiny pointers

General associativity

Having general associativity changes things

- Consider, again, a partitioned cache
- Each page maps to two sets and evicts the LRU page in the union of the two sets

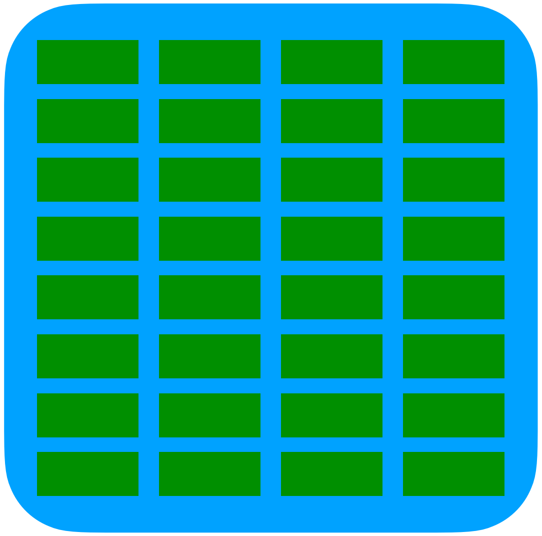


Theorem: The LRU-of-two-sets algorithm with associativity $O(\log \log m)$ on a cache of size $m + (m/\log m)$ is $(1 + o(1))$-competitive vs full associative LRU on a cache of size m. [Bender, Bhattacharjee, Conway, FC, Johnson, Kannan, Kuszmaul, Mukherjee, Porter, Tagliavini, Vorobyeva, West SPAA '21]

Is this the lowest associativity you can get?

General associativity

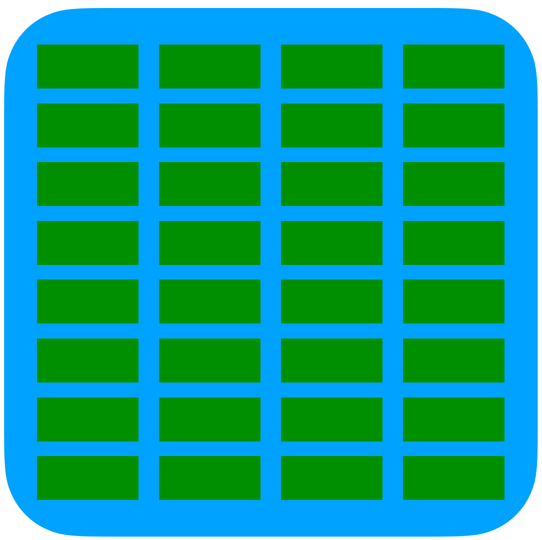
Theorem: For any $d = \omega(1)$, there is a d-associative paging algorithm that is $(1 + o(1))$-competitive vs fully associative LRU with $(1 + o(1))$ resource augmentation. [In preparation.]



Consequence: you can specify the location of any page in the cache using any $\omega(1)$ number of bits without blowing up the paging costs.

Relationship with Dereference Tables

Theorem: You can specify the location of any page in the cache using any $\omega(1)$ number of bits without blowing up the paging costs.



Dereference tables: low associativity vs probability of failing

Low associativity paging: low associativity vs probability of paging

Both: tiny pointers

Why do we want tiny pointers into a cache?

How does paging actually work?

Programs refer to pages by a ***virtual addresses***

Computers refer to pages by ***physical addresses***

- Physical address ≈ location in the cache

Every memory reference requires an ***address translation***

This is slow so there is a very small hardware cache called a TLB to make this fast

Paging Problem

Which leads us to a more complete cost model for paging:

$$cost(p_i) = \begin{cases} 0 & \textbf{if } p_i \textbf{ is in the RAM and translation is in TLB} \\ \epsilon & \textbf{if } p_i \textbf{ is in the RAM but translation is not in TLB} \\ 1 & \textbf{if } p_i \textbf{ is not in RAM} \end{cases}$$

Sociological note:

- Theoreticians want to minimize the number of time we hit the third line (cost of a RAM miss)
- Systems people want to minimize the number of times we hit the second line

Using tiny pointers in TLBs

Without getting into details

TLB is a piece of hardware

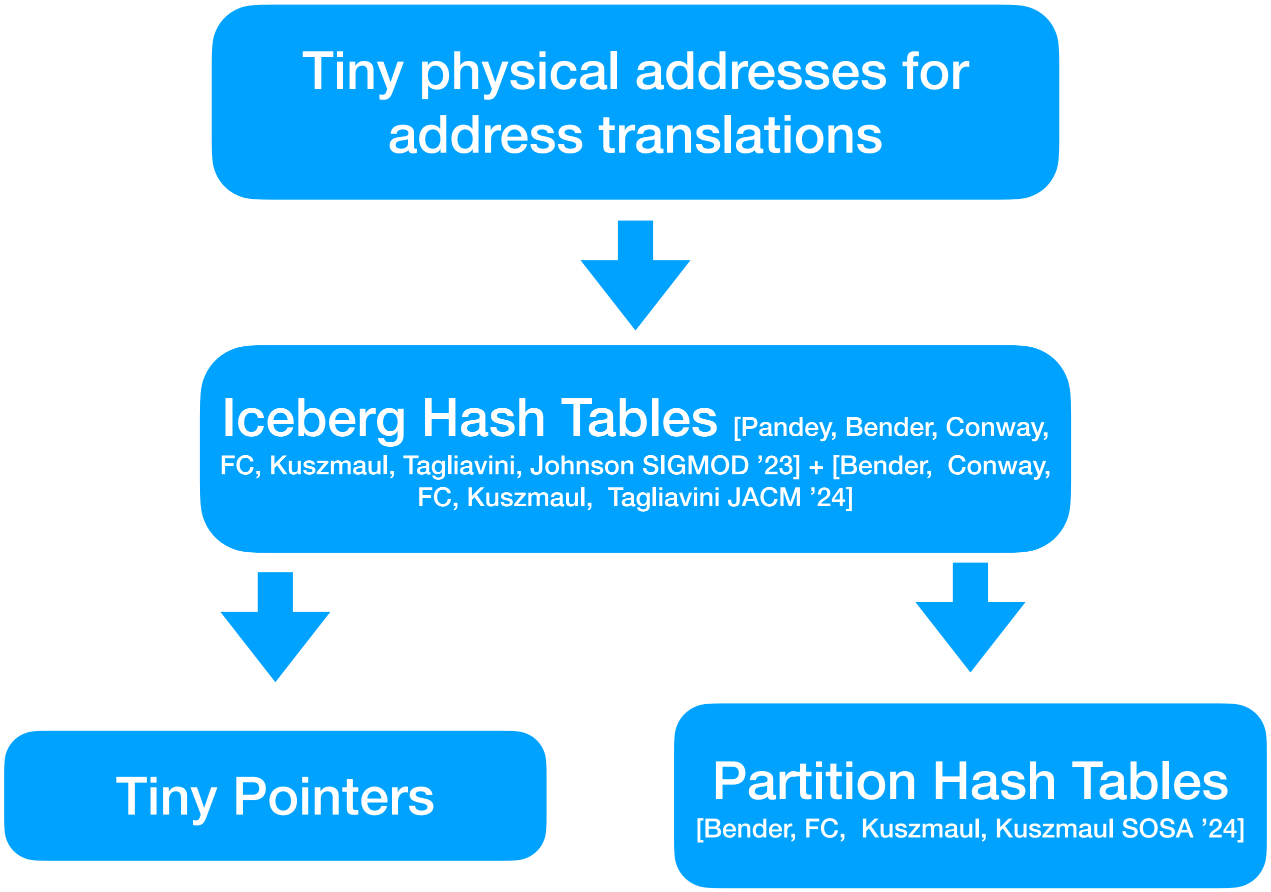
- And we can't make it much bigger
- It consists of a bunch of (virtual, physical) address pairs

We changed the paging algorithm to make tiny physical pointers

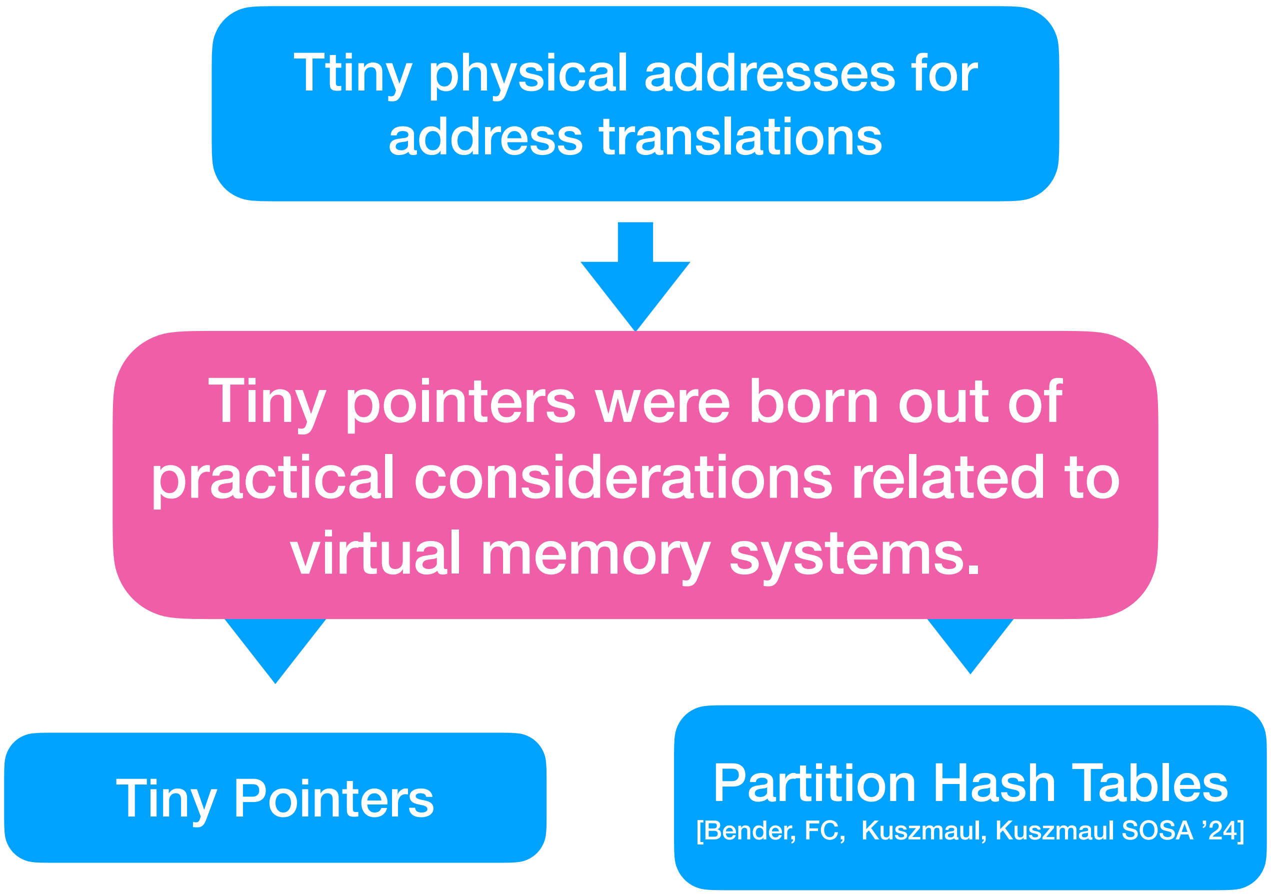
We implemented a TLB that consists of (range of virtual addresses, sequence of tiny physical addresses)

This works in practice: ASPLOS best paper + VMware is building a chip using this idea

The story of tiny pointers



The story of tiny pointers



What’s next, after tiny pointers and retrievers?

What’s next, after tiny pointers and retrievers?

Tiny hounds!

Tiny hounds!