Data Structures with "Randomness": Hashtables

Flashback to Data Structures…

Recall the `Dictionary` interface

- What are the `Dictionary` operations?
- What concrete `Dictionary` implementations did we study?

- What are the tradeoffs between binary search trees and hashtables?
- How often do we need to do successor/range operations?
    - Similarly: How much does locality matter?

Let's develop a data structure with excellent (expected) point lookup/update performance but no support for range operations.

Hashtable Basics

- We have an underlying array of size m
    - We say this array has m slots or buckets
- Suppose we want to store n items, where $n < m$. What is ideal situation?
    - If every element has a unique, designated location, get $O(1)$ operations:
        - Insert a new item → update slot
        - Look up an item → check slot
        - Delete an item → clear slot
- Unfortunately we usually have a universe of items U we may wish to store, where $|U|$ is *much much* bigger than m. Example universes?
    - Punchline: even with $n < m$, we can't guarantee those n items their own dedicated locations because we don't know which particular n items from our universe U that we will be storing…

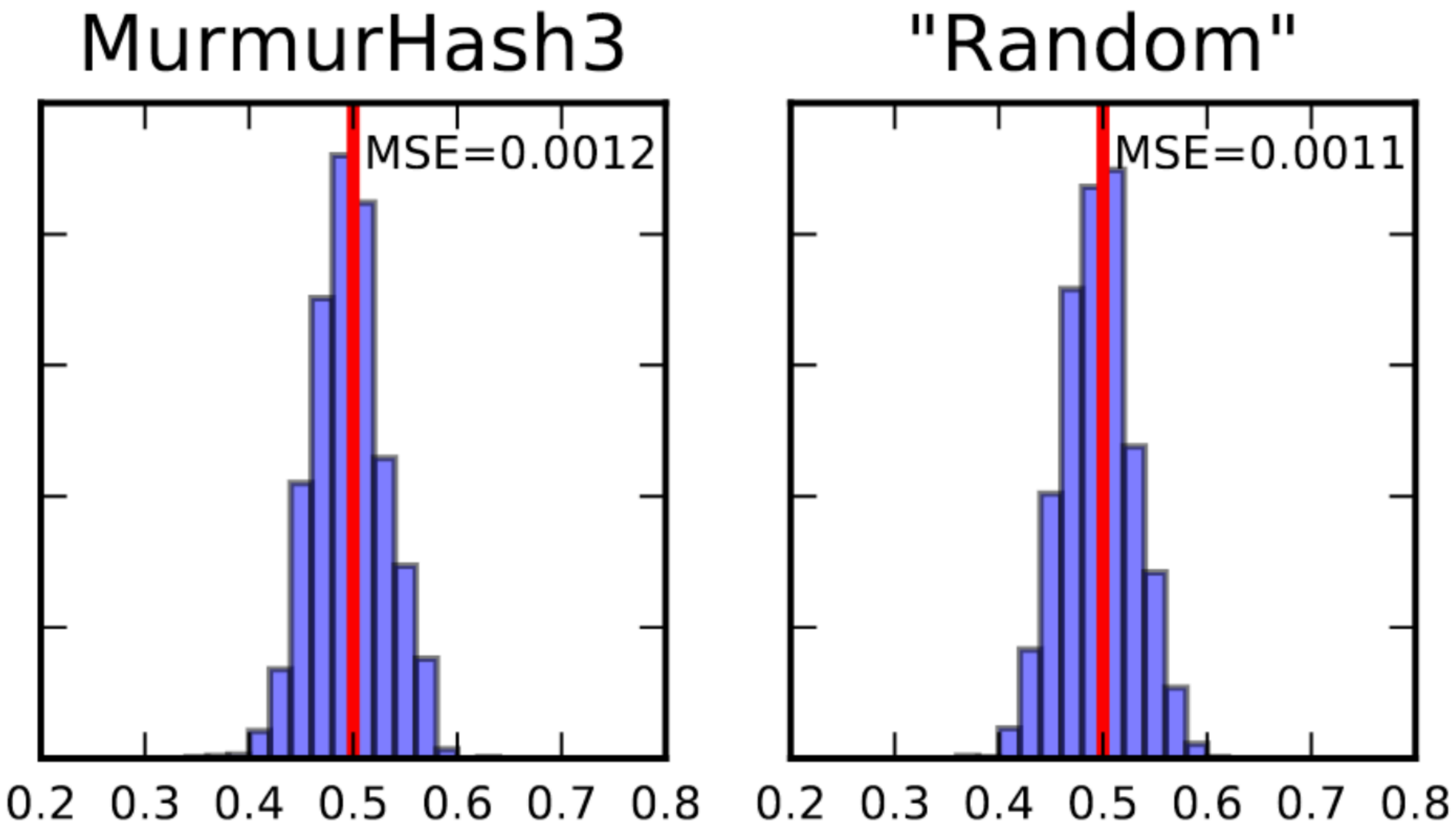
Hash table

- But we still want $O(1)$ operations! Plus, you've been told we achieve that!
  - In reality, we settle for *expected* $O(1)$ performance…
- Idea: use a **hash function** to map each item to a slot
  - h is a one-way function that maps the universe U of keys to slots in our array A:
    $$h : U \rightarrow \{0,1,\ldots,m-1\}$$
- So, we say an item with key k hashes to slot $h(k)$, and that $h(k)$ is the item's hash value
  - Textbook gives example hash functions (and why some are bad)
  - Textbook discusses universal hashing
  - Instead, we're going to focus on analyzing the data structure under the assumption that we have a uniform hash function

Hash function: theory versus practice

- We will *assume* hash function h is *ideal* :
  - For all $i \in U, k$, assume $\Pr(h(i) = k) = 1/m$
  - Assume the hashes of all items are independent:
    $\Pr(h(i) = k \,|\, h(i_2) = k_2, h(i_3) = k_3, \ldots) = 1/m$

Dahlgaard et al. 2017

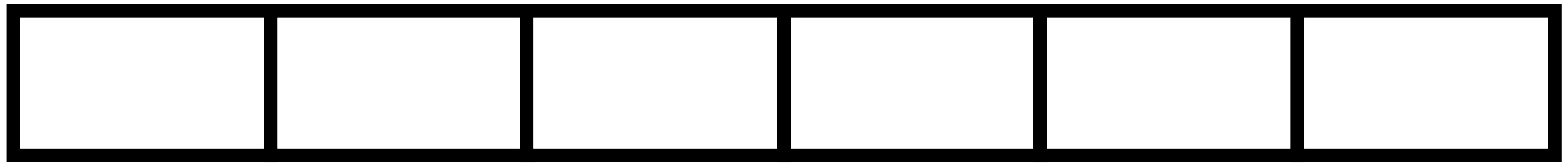


- Such hs called **uniform random hash functions**
- Good hash functions do behave this way in practice
- Lots of theoretical work about weaker assumptions on the hash functions

Hash table

- Hash function h, array A
- Item i is stored in $A[h(i)]$
- $m = 6$

Amir



Beth

Chris

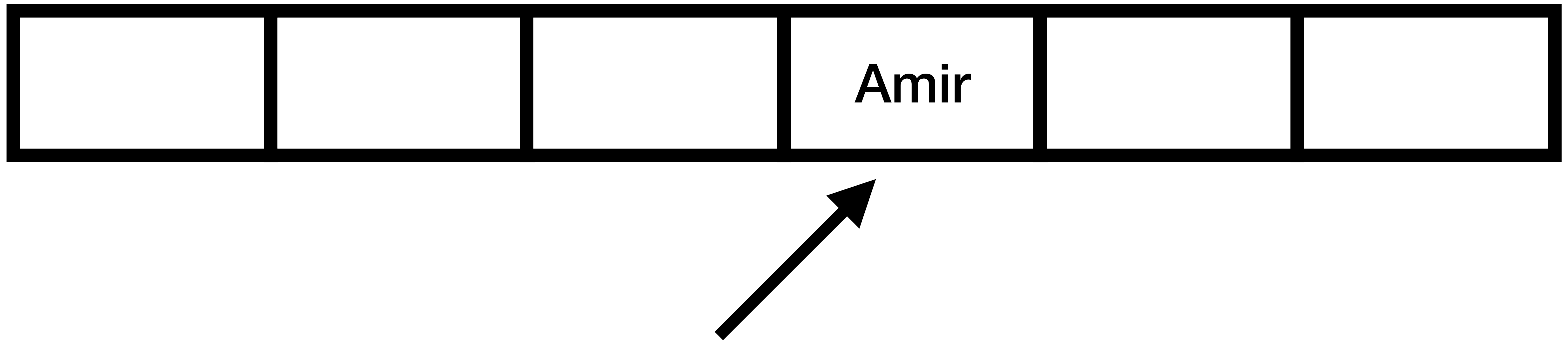
Hash table

- Hash function h, array A
- Item i is stored in $A[h(i)]$

Amir

Beth

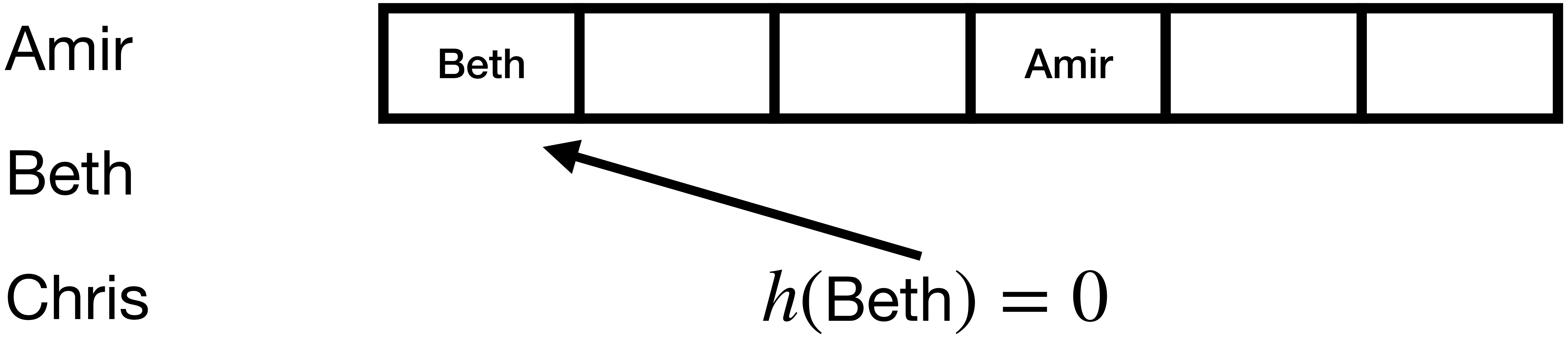
Chris



$h(\text{Amir}) = 3$

Hash table

- Hash function h, array A
- Item i is stored in $A[h(i)]$



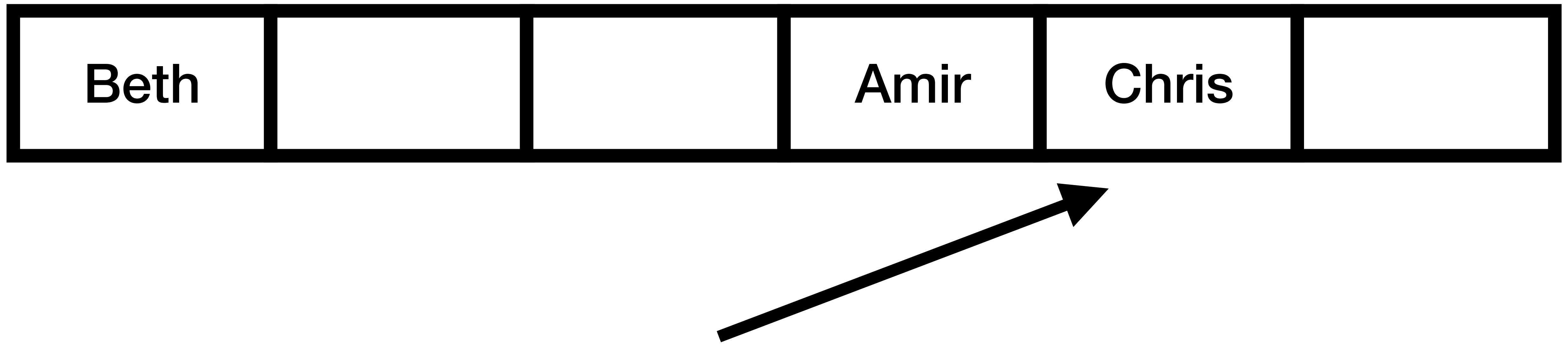
Hash table

- Hash function h, array A
- Item i is stored in $A[h(i)]$

Amir

Beth

Chris



$h(\text{Chris}) = 4$

Hashtable Basics

- We said that even with $n < m$, we can't guarantee those n items their own dedicated locations because we don't know which particular n items from our universe U that we will be storing…
    - So we say a collision occurs when two unique items hash to the same slot $(h(x_1) = h(x_2), x_1 \neq x_2)$
- Practically, we need a way to manage collisions
    - Recall any strategies from data structures?
- Theoretically, we need a way to analyze the impact of collisions on our data structure performance
    - Our collision strategy needs to maintain our expected $O(1)$ performance (luckily, several do!)

Managing Collisions via Chaining

- Idea: store a linked list at each array entry (what kind?)
- When an item hashes to a slot, store it in the (possibly empty) linked list

Amir

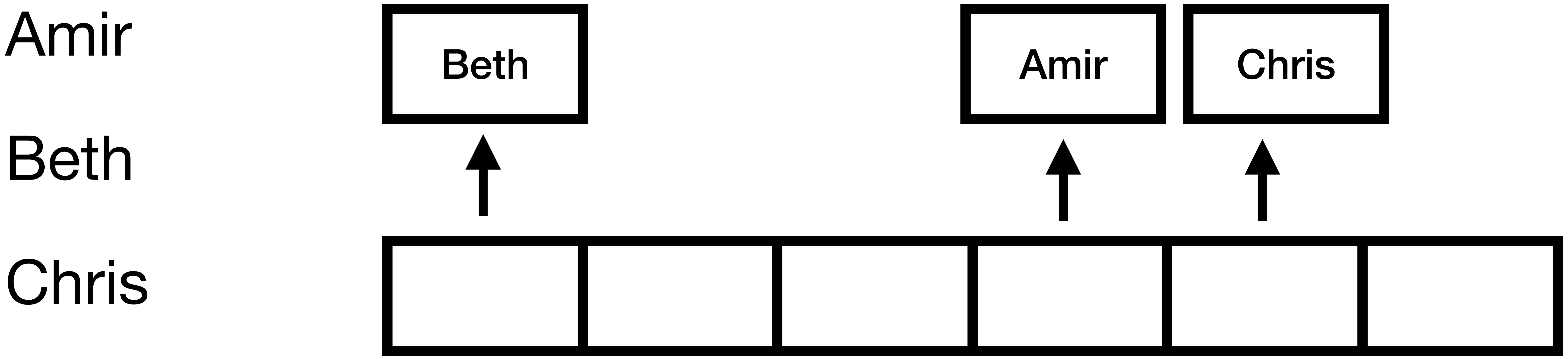
Beth

Chris

Beth			Amir	Chris	

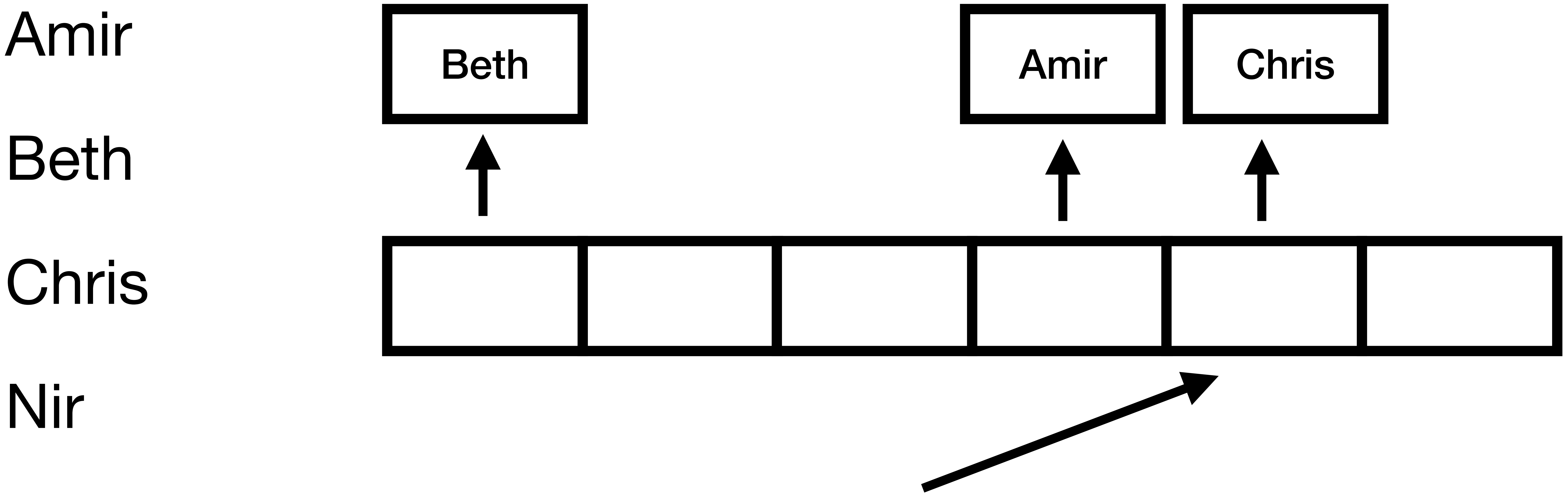
Managing Collisions via Chaining

- Idea: store a linked list at each array entry (what kind?)
- When an item hashes to a slot, store it in the (possibly empty) linked list



Managing Collisions via Chaining

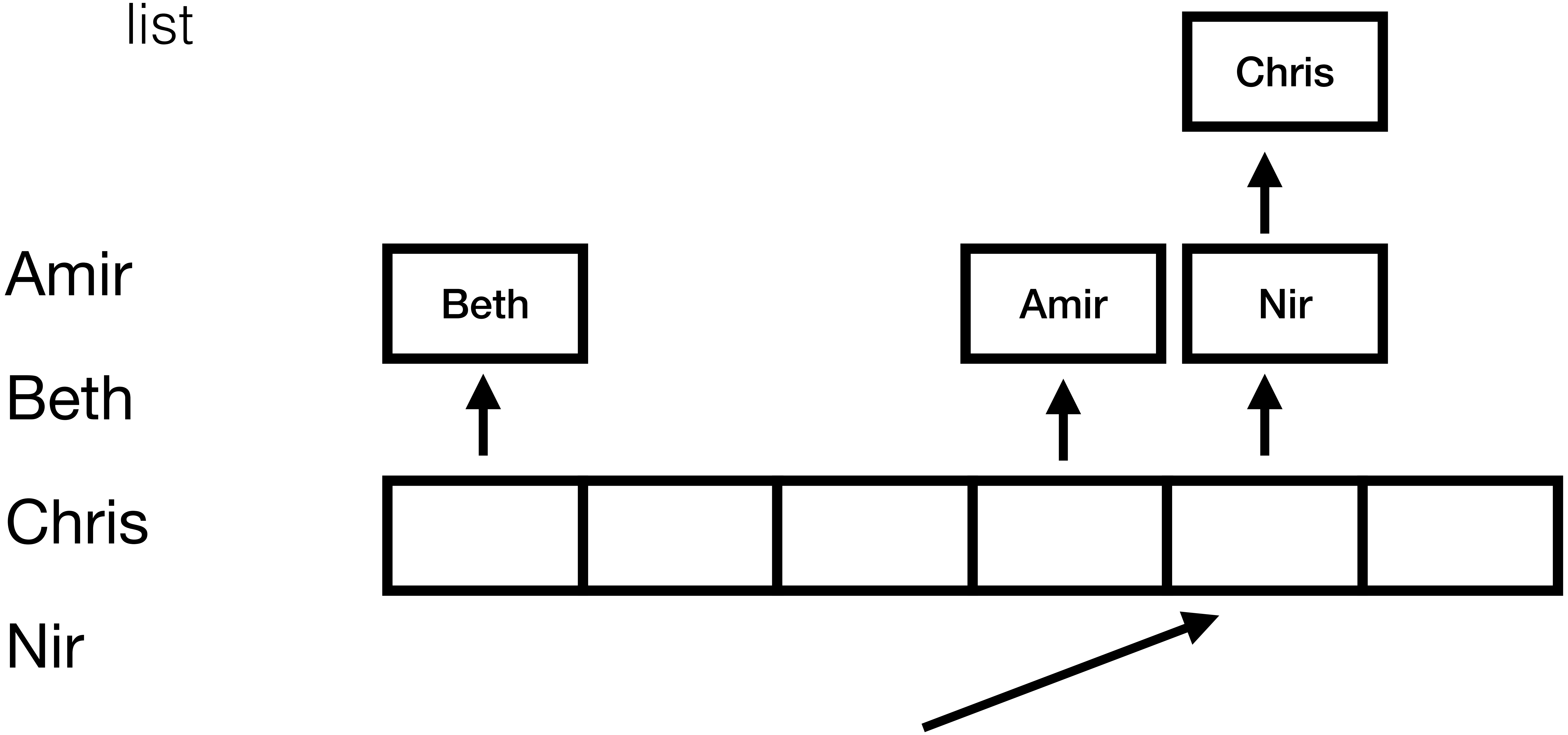
- Idea: store a linked list at each array entry (what kind?)
- When an item hashes to a slot, store it in the (possibly empty) linked list



$h(\text{Nir}) = 4$

Managing Collisions via Chaining

- Idea: store a linked list at each array entry (what kind?)
- When an item hashes to a slot, store it in the (possibly empty) linked list



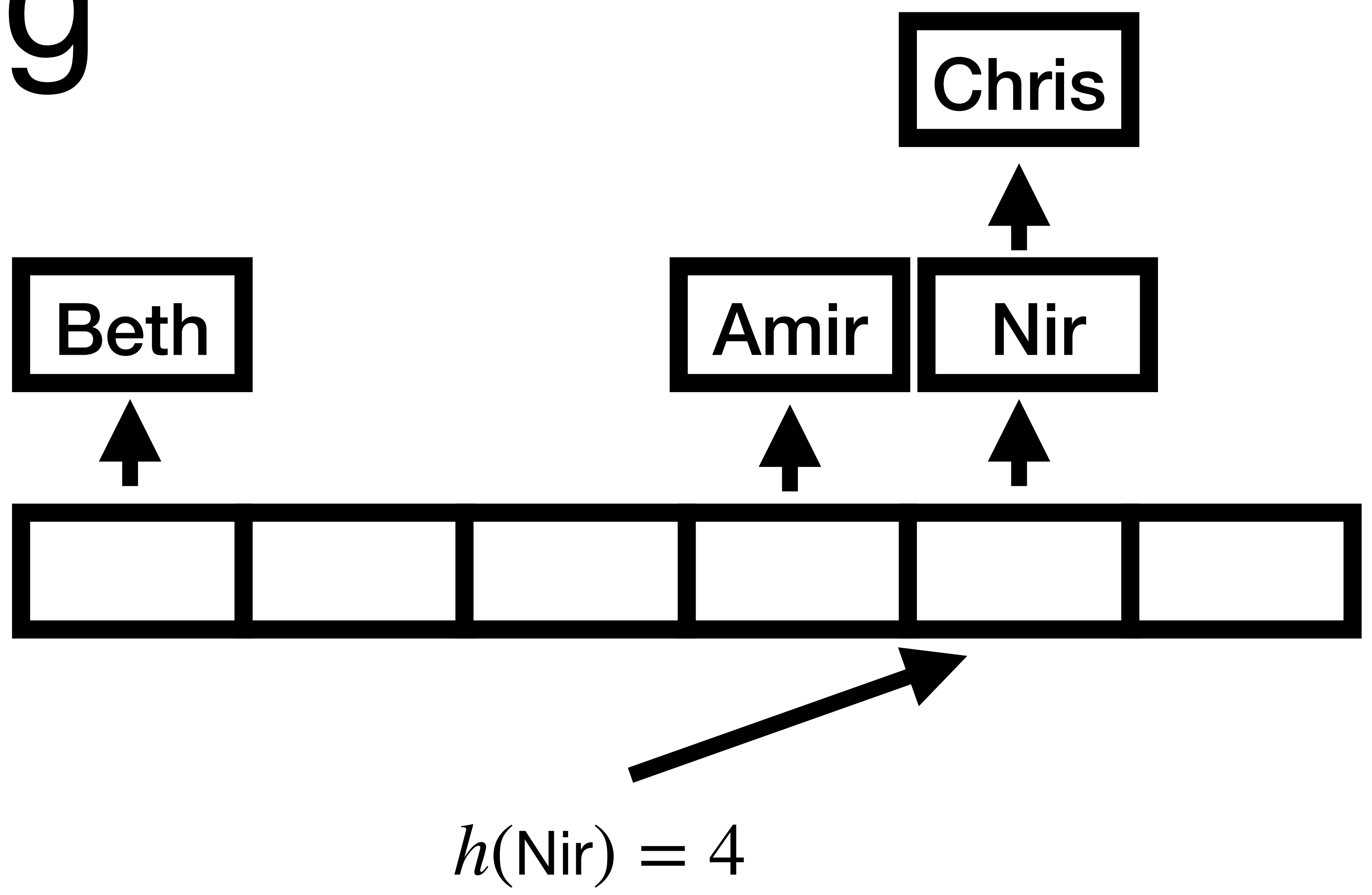
$$h(\text{Nir}) = 4$$

Managing Collisions via Chaining

- Store a doubly linked list at each array entry
- When an item hashes to a slot, **prepend** it to the linked list

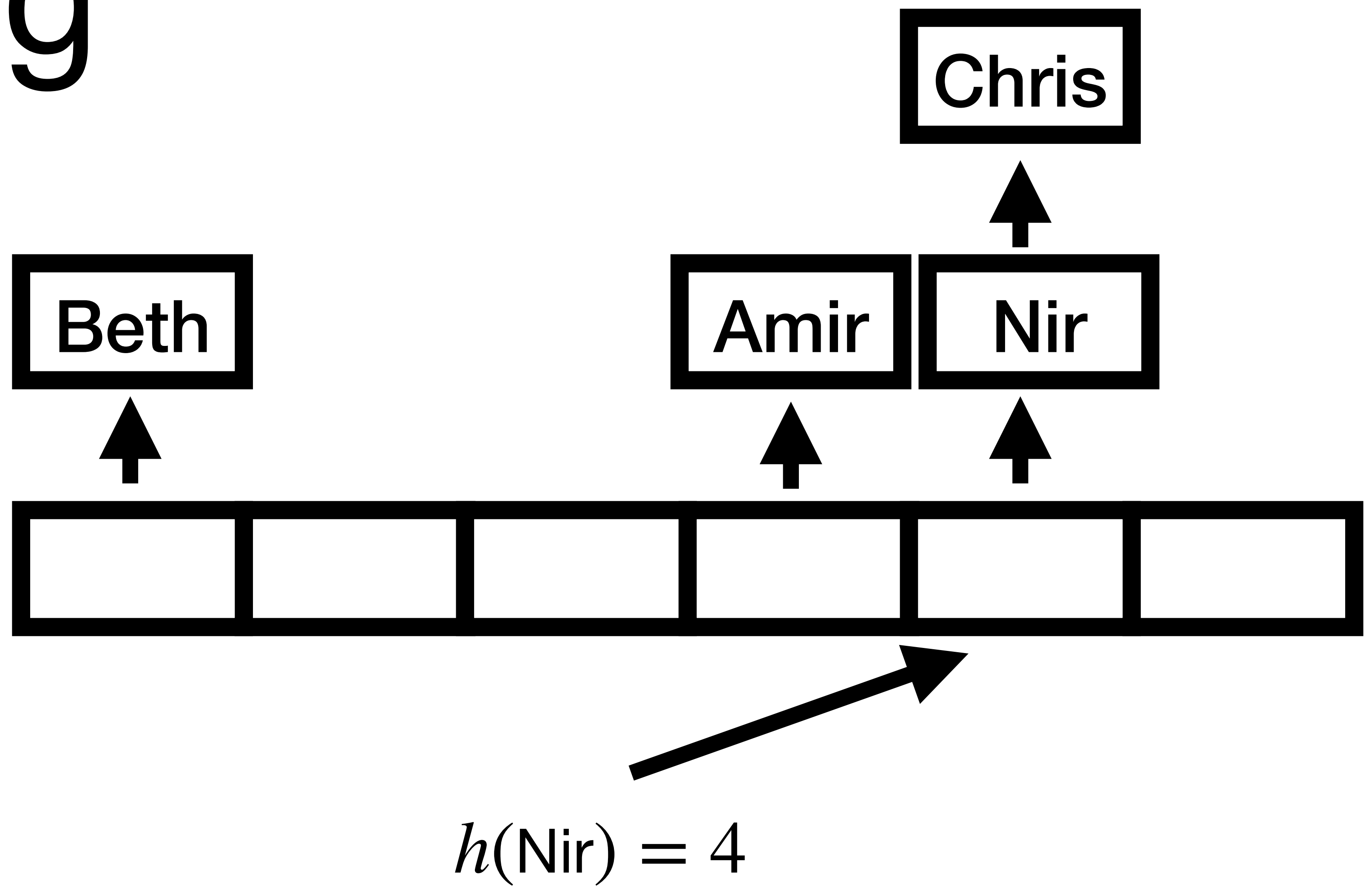
- How can we insert? (See above…)
- How can we lookup?
- How can we delete?

- (Harder) How much time do these operations take?



Managing Collisions via Chaining

- Store a doubly linked list at each array entry
- When an item hashes to a slot, **prepend** it to the linked list



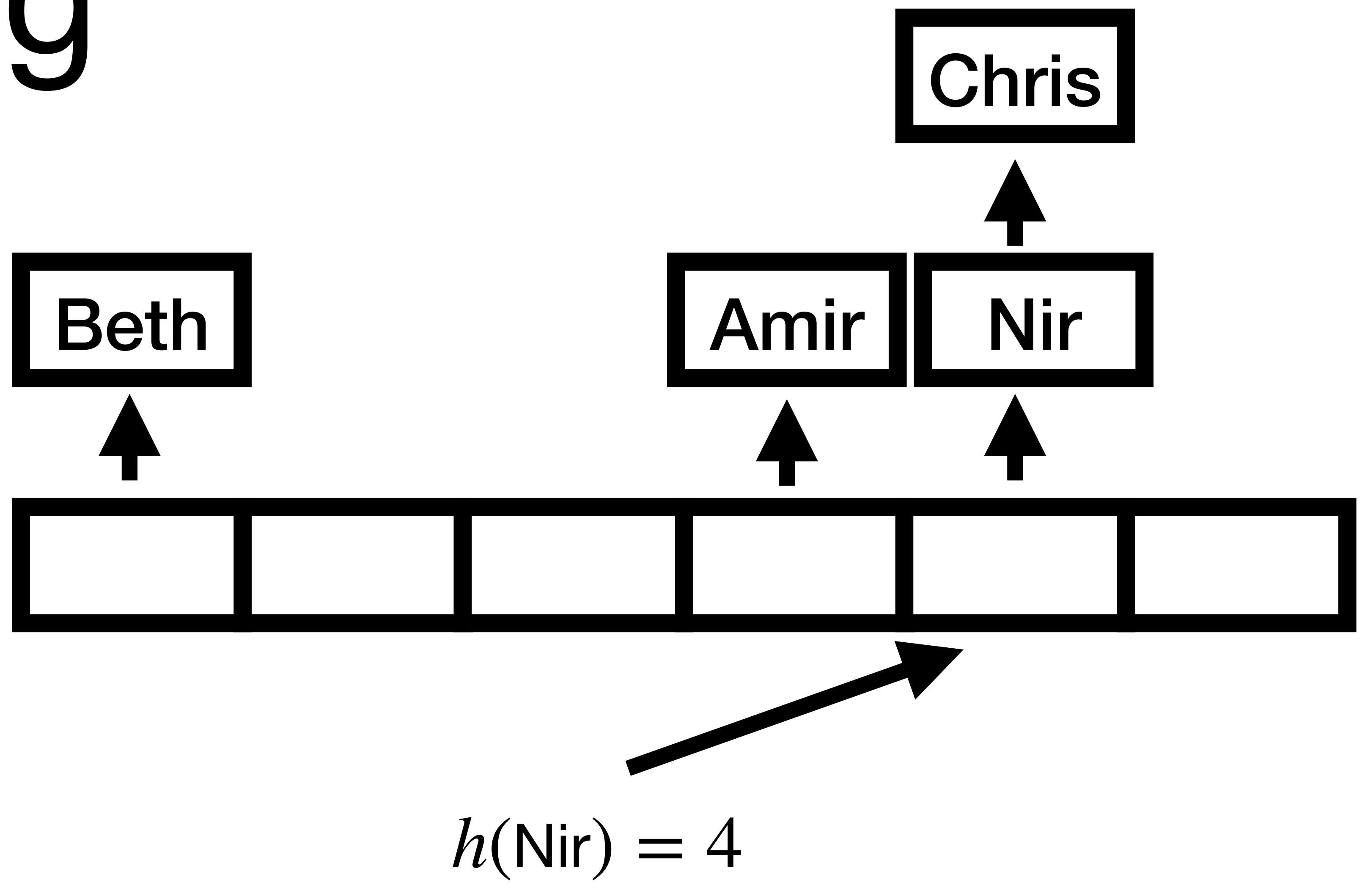
Insert(k):

Prepend k at the head of the list $A[h(k)]$

- Runtime?
  - $O(1)$ — exactly; not in expectation!
  - Note, we assume k is not in hashtable
    - If don't want that assumption, do a lookup first!

Managing Collisions via Chaining

- Store a doubly linked list at each array entry
- When an item hashes to a slot, **prepend** it to the linked list



Delete(k):

Scan the list $A[h(k)]$, and delete the entry with key k

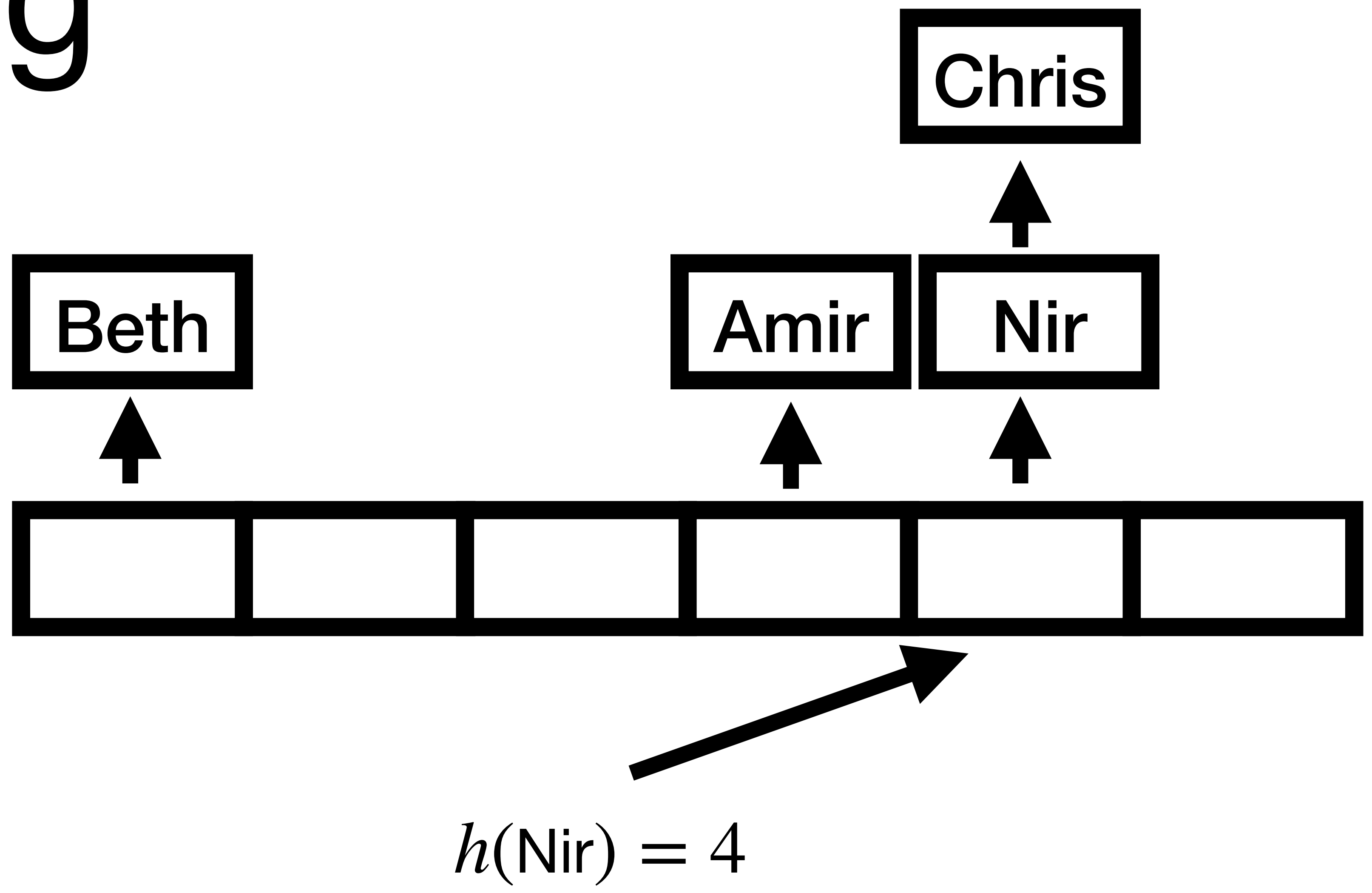
- Runtime?
  - $O(L)$, where L is the length of the chain in slot $h(k)$
  - What is L?

Hashing and Chain Length

Worst-case delete time in a hash table with chaining: number of balls in a particular bin. **Question:** Expected number of balls in a particular bin b?

- Let X_i denote indicator r.v. that item i hashes to bucket b
  - Assuming uniform hashing, $Pr(X_i = 1) = \frac{1}{m}$
- Let $X = \sum_{i=1}^{n} X_i$ denote the number of items that hash to bucket b
- By linearity of expectation, $E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m}$

Managing Collisions via Chaining



- Store a doubly linked list at each array entry
- When an item hashes to a slot, **prepend** it to the linked list

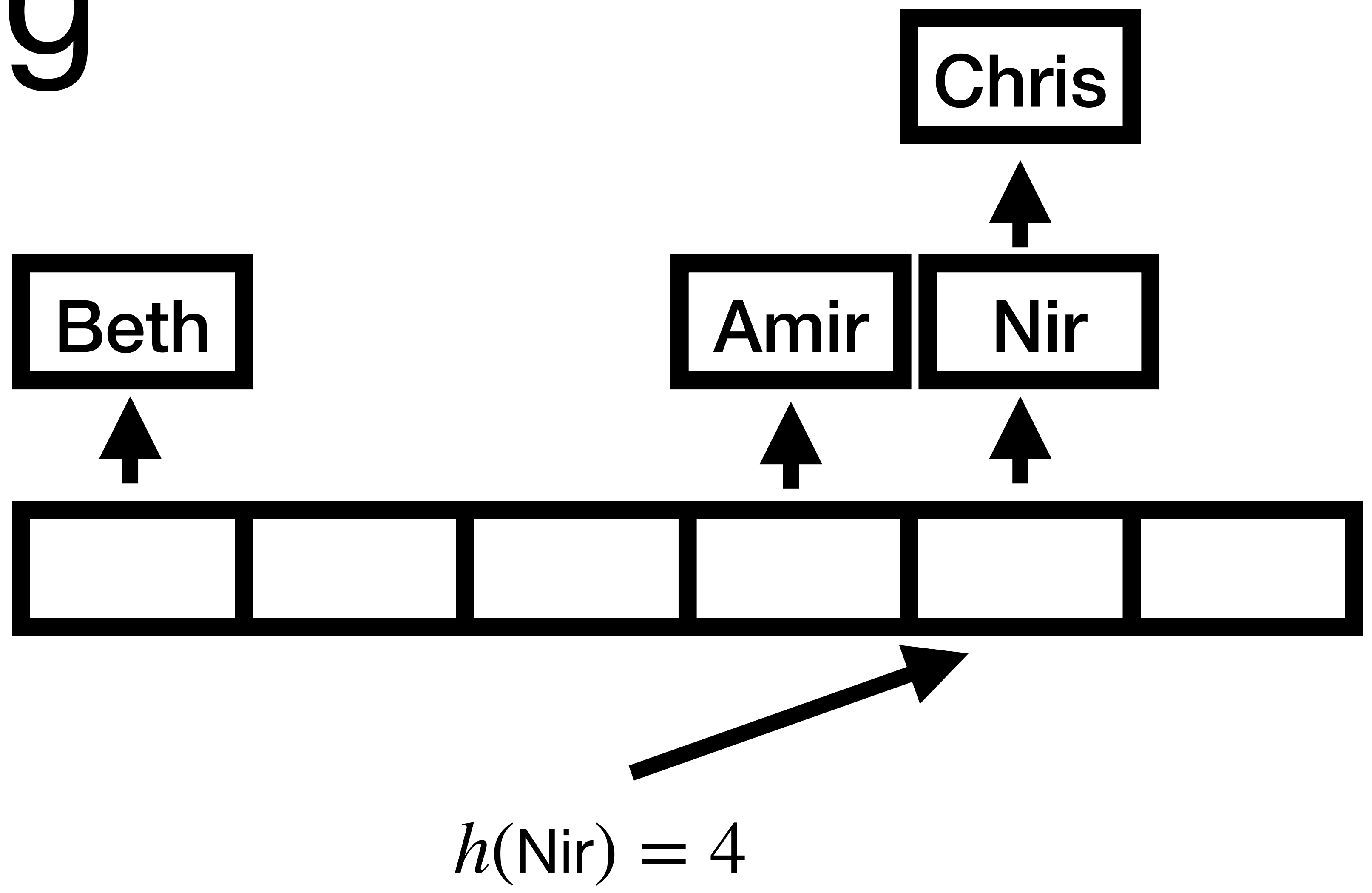
Delete(k):

Scan the list $A[h(k)]$, and delete the entry with key k

- Runtime?
  - $O(L)$, where L is the length of the chain in slot $h(k)$
  - What is L?
    - $E[L] = \dfrac{n}{m}$. We'll also call this the hashtable's **load factor**

Managing Collisions via Chaining

- Store a doubly linked list at each array entry
- When an item hashes to a slot, **prepend** it to the linked list



Lookup(k):

Scan the list $A[h(k)]$; return the entry with key k if an entry exists

- Runtime?
  - (Surprisingly?) Lookup behavior is different in two cases!
    - “Successful” lookup vs. “unsuccessful”
      - Why?

Hashing and Chain Length

Worst-case lookup time in a hash table with chaining: number of balls in a particular bin. **Question:** what's different about successful and unsuccessful cases?

- Unsuccessful lookup: must scan through entire chain
  - Cost is $O(L)$, and we showed that $E[L] = \dfrac{n}{m}$
- Successful lookup stops once we find the target element. Analysis is tricky because we always insert at the front of the list!
  - Expected cost to lookup item x when x is in the hashtable is the expected number of items that collided with x ***after x was inserted***

Cost of Successful Lookup

- Assume that element x is equally likely to be any of table's n elements
    - Number of elements checked is 1 plus number of elements that appear before x in list $A[h(x)]$
    - Observation: all elements are placed at the front of the list, so this is precisely the number of elements that collided with x and were inserted after x was
- Let x_i be the i^{th} element inserted into the list
- Let X_{ij} be the indicator r.v. that equals 1 when $h(x_i) = h(x_j)$
    - i.e., X_{ij} is 1 when there is a collision between x_i and x_j, 0 otherwise
- Under uniform hashing assumption, $E[X_{ij}] = 1/m$

Cost of Successful Lookup

Expected number of collisions with x that occur after x is inserted?

- Let x_i be the i^{th} element inserted into the list
    - In other words, we insert $x_1, x_2, \ldots, x_n$ into A
- Let X_{ij} be the indicator r.v. that equals 1 when $h(x_i) = h(x_j)$
    - Note: X_{ij} is 1 when there is a collision between x_i and x_j, 0 otherwise
- Under our uniform hashing assumption, $E[X_{ij}] = 1/m$
- With this, can we reason about the number of elements examined in a successful search?

Cost of Successful Lookup

The expected number of elements examined in a successful search is:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

Since x may be any of the n elements we insert, we average the contribution of each of the n items

# of comparisons to find x_i are 1 plus the expected number of collisions among all items inserted <u>after</u> x_i

Cost of Successful Lookup

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right)$$

by Linearity of Expectation

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) = \frac{1}{n}\sum_{i=1}^{n}1+\frac{1}{mn}\sum_{j=i+1}^{n}1$$

$$= 1+\frac{1}{mn}\sum_{i=1}^{n}(n-i) = 1+\frac{1}{mn}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)$$

$$= 1+\frac{1}{mn}\left(n^2-\frac{n(n+1)}{2}\right) = 1+\frac{1}{nm}\left(\frac{2n^2-n^2-n}{2}\right)$$

$$= 1+\frac{n-1}{2m} = 1+\frac{\frac{n}{m}}{2}-\frac{\frac{n}{m}}{2n} = O(1+\frac{n}{m})$$

Hashtable Summary

We can get close to $O(1)$ performance for insert, lookup, and delete operations ($O(1 + n/m)$ in expectation, where n/m can be controlled by resizing)

- There are other strategies for resolving collisions, but analyzing their performance is tricky
  - Linear probing: $h(k, i) = (h(k) + i) \mod m$
  - Quadratic probing: $h(k, i) = (h(k) + c_1 i + c_2 i^2) \mod m$
  - Double hashing: $h(k, i) = h(k \,||\, i)$
  - Power-of-two-choices: stored at $h_1(k)$ or $h_2(k)$, uses "cuckooing"

Hashtables are a great data structure for many applications

- As long as you don't need to iterate or sort!

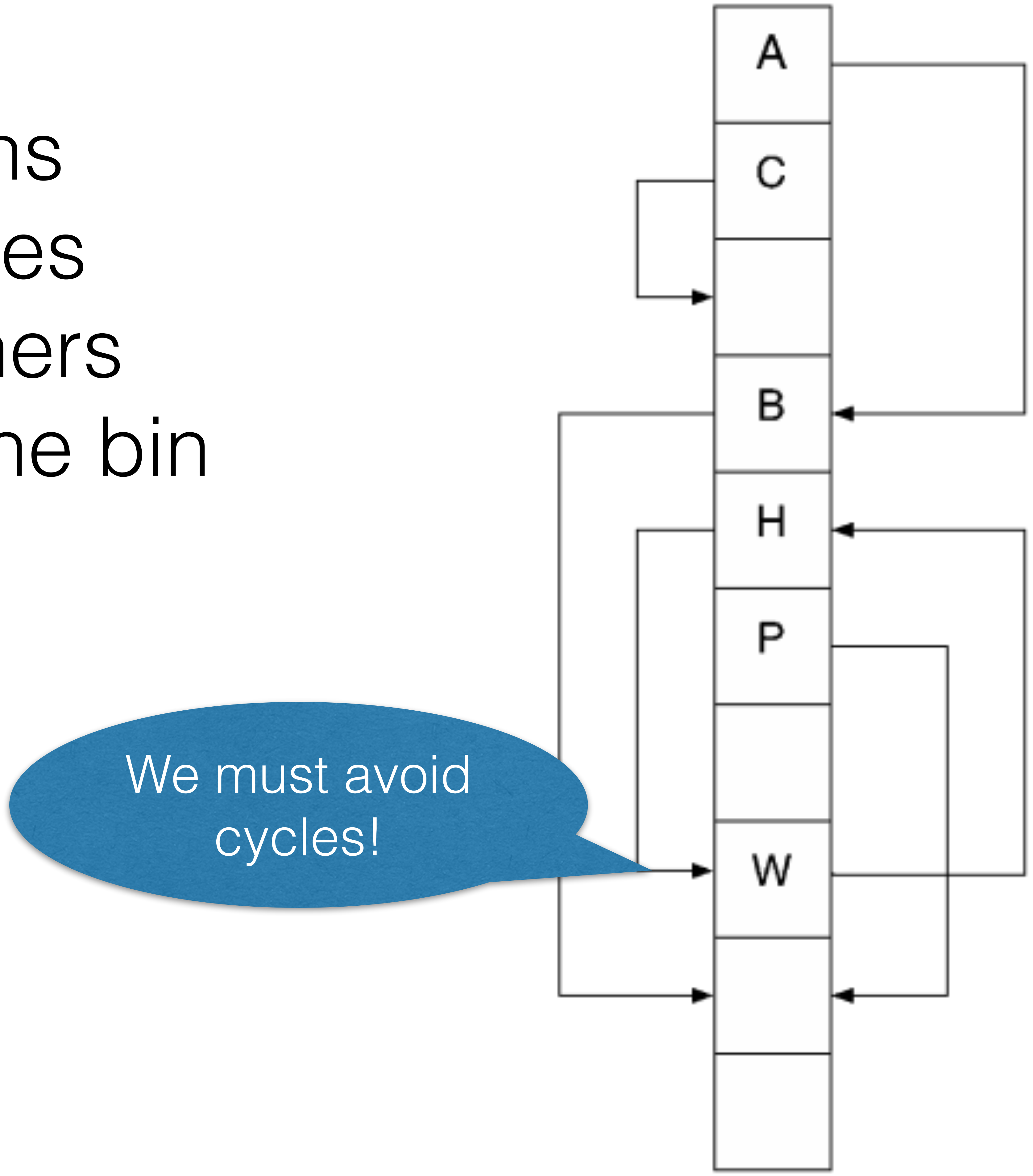
(Extra: Technique) Cuckoo Hashing

img: http://phenomena.nationalgeographic.com/files/2016/04/Cuculus_canorus_vogelartinfo.jpg
img: https://en.wikipedia.org/wiki/File:Eastern_Phoebe-nest-Brown-headed-Cowbird-egg.jpg

Techniques to Resolve Collisions

- **Cuckoo Hashing**
  - Select 2 independent hash functions
    - A key can now land in 1 of 2 places
  - Resolve collisions by “pushing” others out of our bin and placing them in the bin associated with their other hash
  - The process may need to repeat

  - What happens when we:
    - put(X) where $hash_1(X) = 0$?
    - put(Y) where $hash_1(Y) = 7$?



src: https://en.wikipedia.org/wiki/Cuckoo_hashing#/media/File:Cuckoo.svg

Cuckoo Hashing

- For independent hash functions and low load factor, expected O(1)
- No runs like we have with linear probing
    - No shifting “down the line” on inserts
    - At most 2 checks per lookup

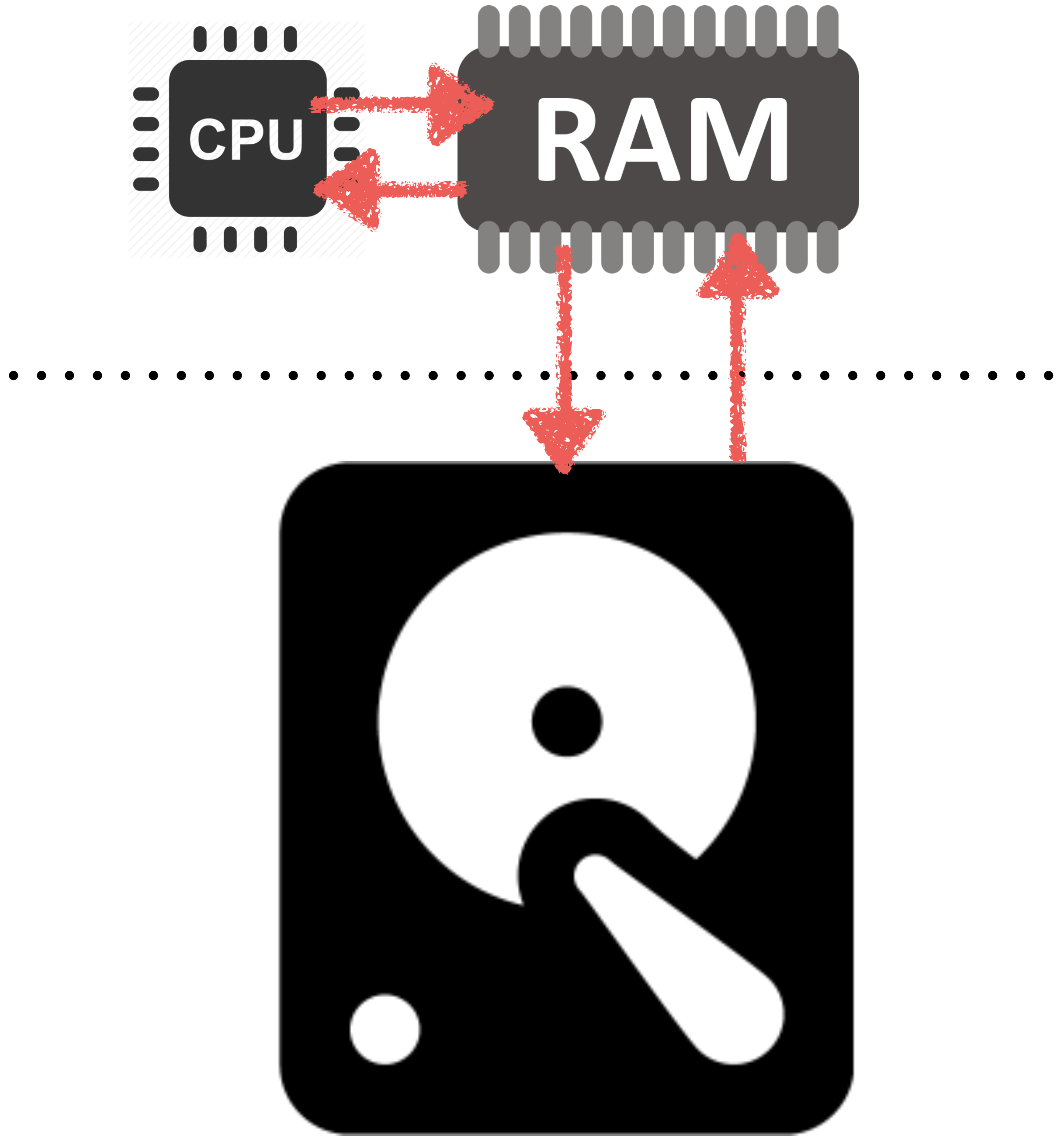
(Extra: Problem)
Membership Queries

Memory Hierarchy

- **Problem 1:** Sometimes (almost always?) we have more data than fits in memory

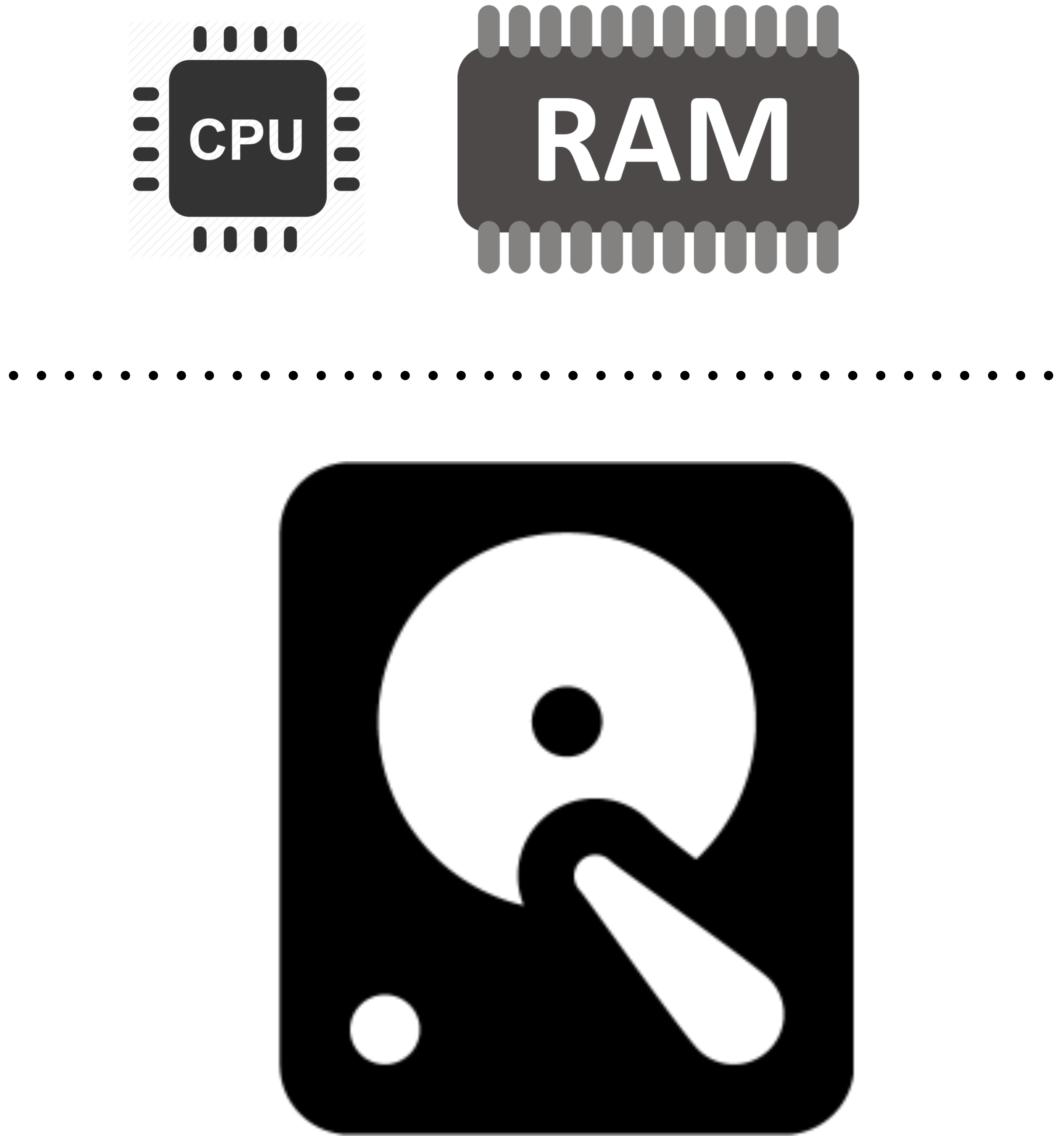
- **Solution:** Store a subset of our data in a cache

    - When we need something that isn't in cache, we kick out the least valuable things to make room for the thing we need



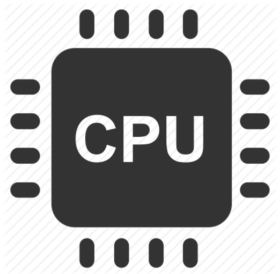
Memory Hierarchy

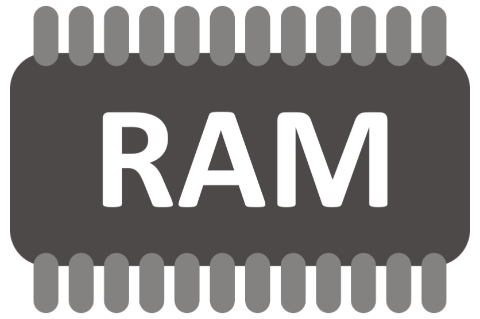
- **Problem 2:** Not all levels in our cache have the same cost



Memory Hierarchy

- **Problem 2:** Not all levels in our cache have the same cost

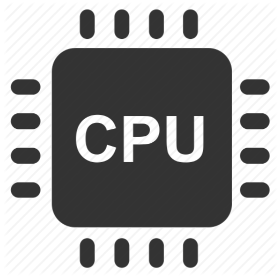


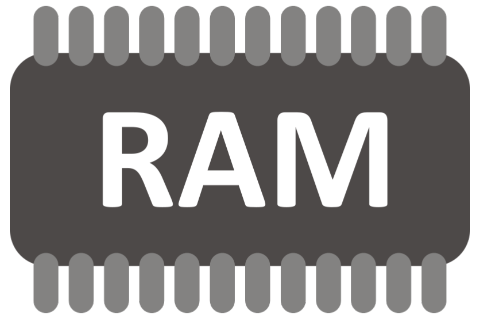


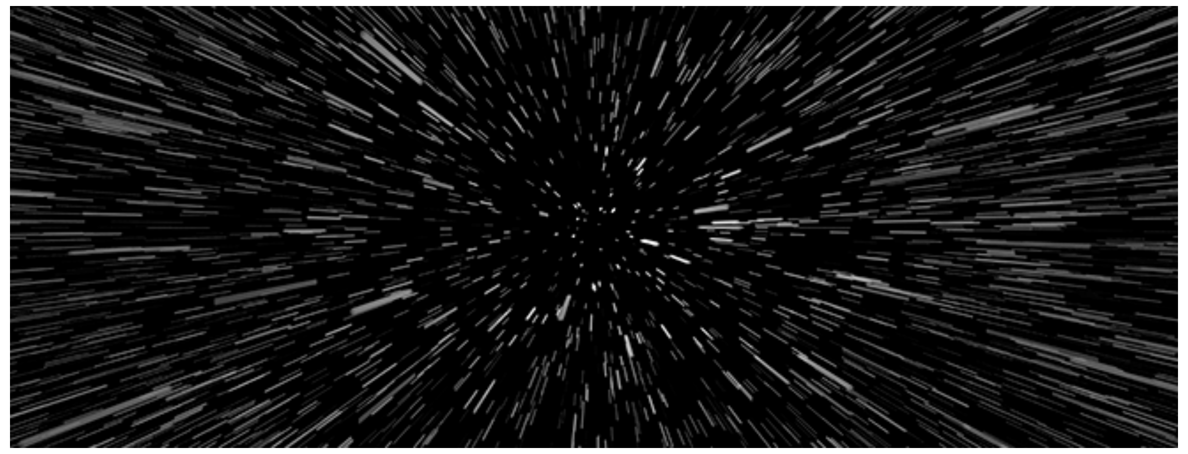
https://www.istockphoto.com/photo/pile-of-money-gm172637949-581154
http://www.freephotosbank.com/photographers/photos1/45/med_53ff4957d796d0ff0a7d3151ec4e4a20.jpg

Memory Hierarchy

- **Problem 3:** Not all levels in our cache have the same speed

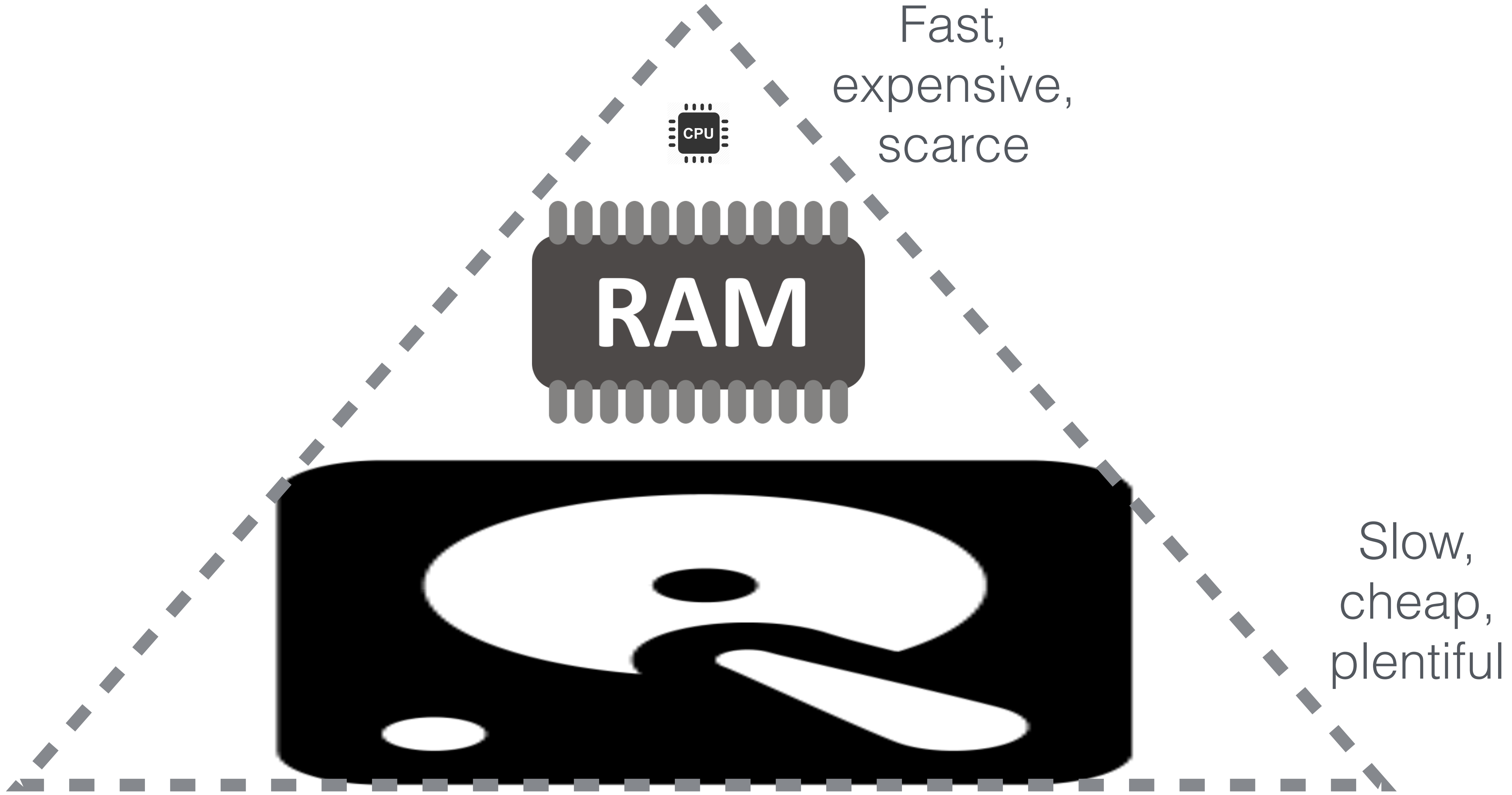






Memory Hierarchy

- Result: we have a lot of slow, cheap storage, less RAM, and very little CPU cache.
  - We will focus on the interaction between RAM and disk



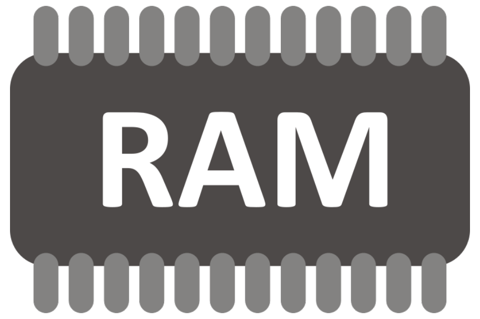
Scenario: Photo Storage

Suppose:

- We have a small RAM cache that holds 2 photos
- Our cache is initially empty
- We read from disk into cache, and evict the least recently used photo when we need space

Memory Hierarchy

Small, fast

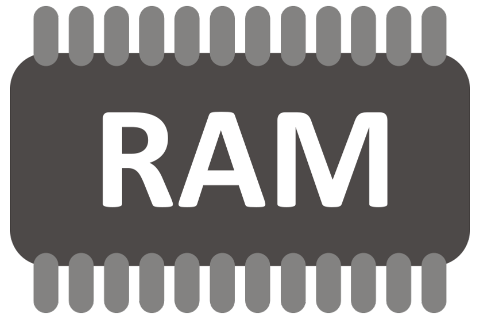


Big, slow

Memory Hierarchy

`get(cat)`

Small, fast



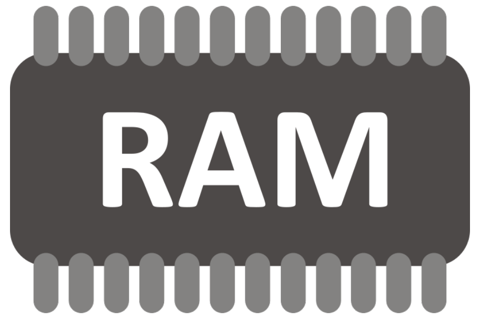
?

Big, slow

Memory Hierarchy

```
get(cat)
```

Small, fast



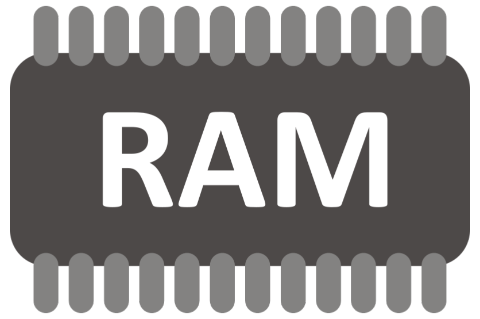
Big, slow




Memory Hierarchy

```
get(cat)
get(cow)
```

Small, fast



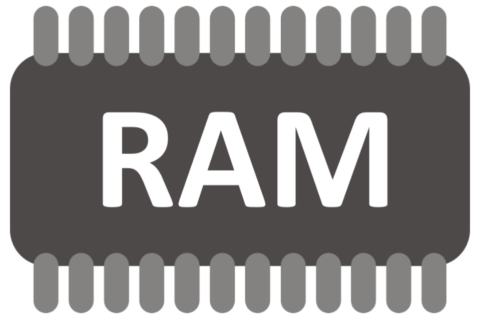
?

Big, slow

Memory Hierarchy

```
get(cat)
get(cow)
```

Small, fast

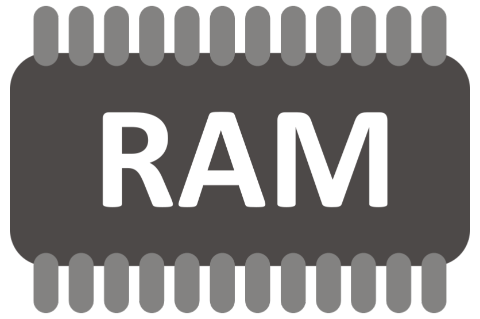


Big, slow

Memory Hierarchy

```
get(cat)
get(cow)
get(dog)
```

Small, fast



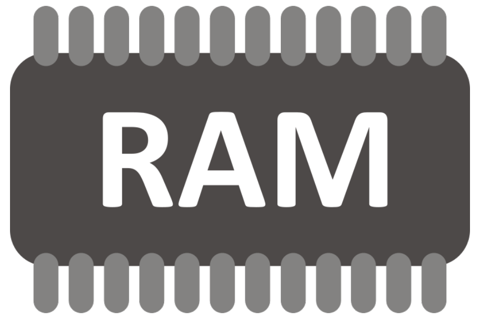
?

Big, slow

Memory Hierarchy

```
get(cat)
get(cow)
get(dog)
```

Small, fast

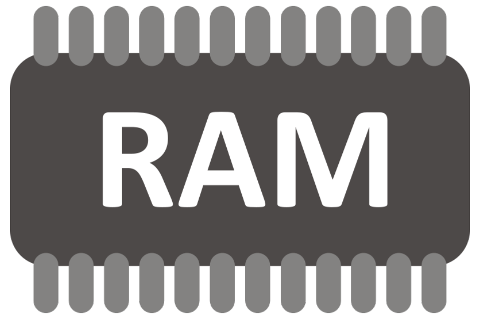


Big, slow

Memory Hierarchy

```
get(cat)
get(cow)
get(dog)
get(goat)
```

Small, fast



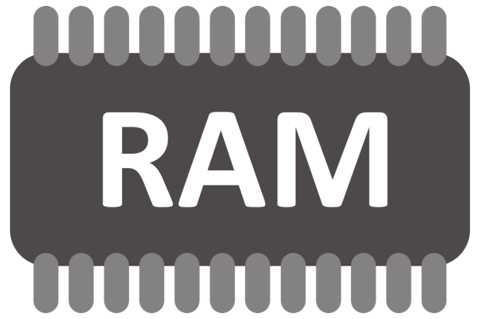
?

Big, slow

Memory Hierarchy

```
get(cat)
get(cow)
get(dog)
get(goat)
```

Small, fast

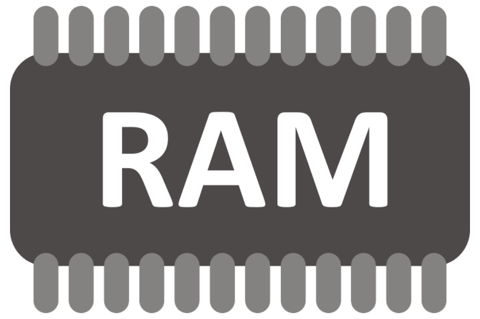


Big, slow

Memory Hierarchy

```
get(cat)
get(cow)
get(dog)
get(goat)
get(cat)
```

Small, fast



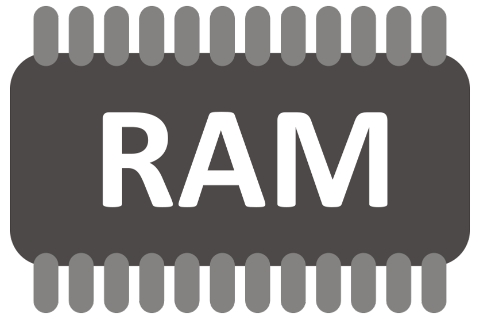
?

Big, slow

Memory Hierarchy

```
get(cat)
get(cow)
get(dog)
get(goat)
get(cat)
```

Small, fast

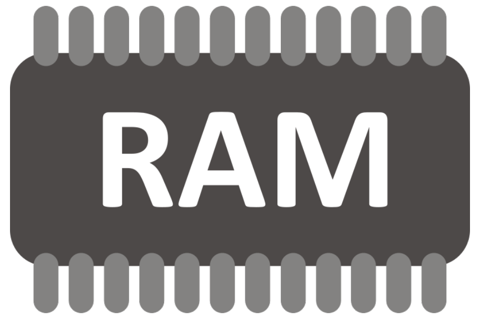


Big, slow

Memory Hierarchy

```
get(cat)
get(cow)
get(dog)
get(goat)
get(cat)
get(liger)
```

Small, fast



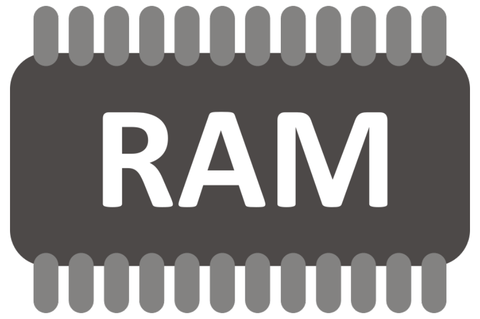
?

Big, slow

Memory Hierarchy

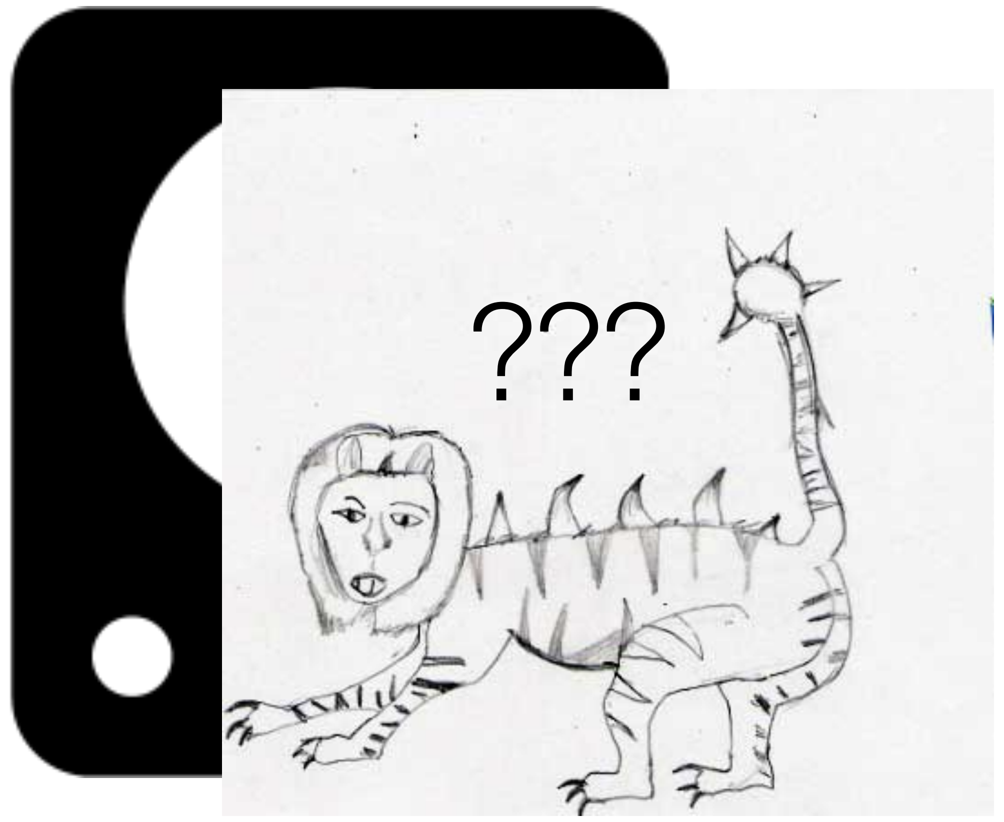
```
get(cat)
get(cow)
get(dog)
get(goat)
get(cat)
get(liger)
```

Small, fast



?

Big, slow



Memory Hierarchy

- **Problem:** We paid an expensive cost just to find out the thing we were looking for didn't exist!!

- **Idea**: Cache a set of all the keys (names of all photos on disk)

  1. Check the names set first *before* checking disk

  2. Don't go to disk if we know the thing isn't there

Membership Queries

- How to implement our name set?
  - If we want to look things up quickly, use a hash set

- If we want to avoid collisions:
  - Make it big
  - Use a large hash so to uniquely **fingerprint** each file (`P(collision) == small`)

- **New problem**: keys can be long, fingerprints are large. Now our set takes up a large portion of our cache

Membership Queries

- **Insight**: we don’t need to be perfect.
- If we go to disk an extra time, no worse off
  - False positives are not ideal, but they are OK
- If we don’t go to disk when something exists, BAD (or sick)
  - False negatives are correctness bugs, not OK
- We will build a structure that does **approximate membership queries** and is more efficient than a set.

Bloom Filter

- Answers with “possibly in set” or “definitely not in set”
- We save space by not explicitly storing hashes or keys

- How it works:
  - Create a bit array of *m* bits
  - Select *k* hash functions
  - Hash each element *k* times and set all *k* bits
  - An element is missing if **any** of its *k* bits is unset
  - An element may be present if **all** of its *k* bits are set

Bloom Filters

Insert(key):

```
for hashFunction_i in hashFuncions_i…k:
   bitmap[hashFunction_i(key) % m] = 1
```

Query(key):

```
for hashFunction_i in hashFuncions_i…k:
   if (bitmap[hashFunction_i(key) % m] != 1):
      return "not in set"
return "maybe in set"
```

Bloom Filters

- Deleting keys?
  - A key maps to *k* bits, and although setting any one of those *k* bits to zero would remove that key from the set, it will also remove **every key** that maps to one of those bits.
  - Deleting would introduce **false negatives**!

- Resizing Bitmap?
  - No way to grow array using just the bit values
  - Although keys are not stored, they are often available
  - When the false positive rate gets too high (overloaded, too many "deletes" still in bitmap), read keys from slower media and resize+rehash

Related DS: Quotient Filters

- A nifty idea with an even nifty-er paper name (Don't Thrash: How to Cache your Hash in Flash)
- Uses linear probing to support efficient deletes and merges
- "Write-optimized" data structure (my research area)
- Based on an end-of-chapter problem in an undergraduate data structures textbook
  - Takeaway: You can publish a paper with the skills you already have!

Acknowledgments

- Some of the material in these slides are taken from
  - Shikha Singh
  - CLRS