(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

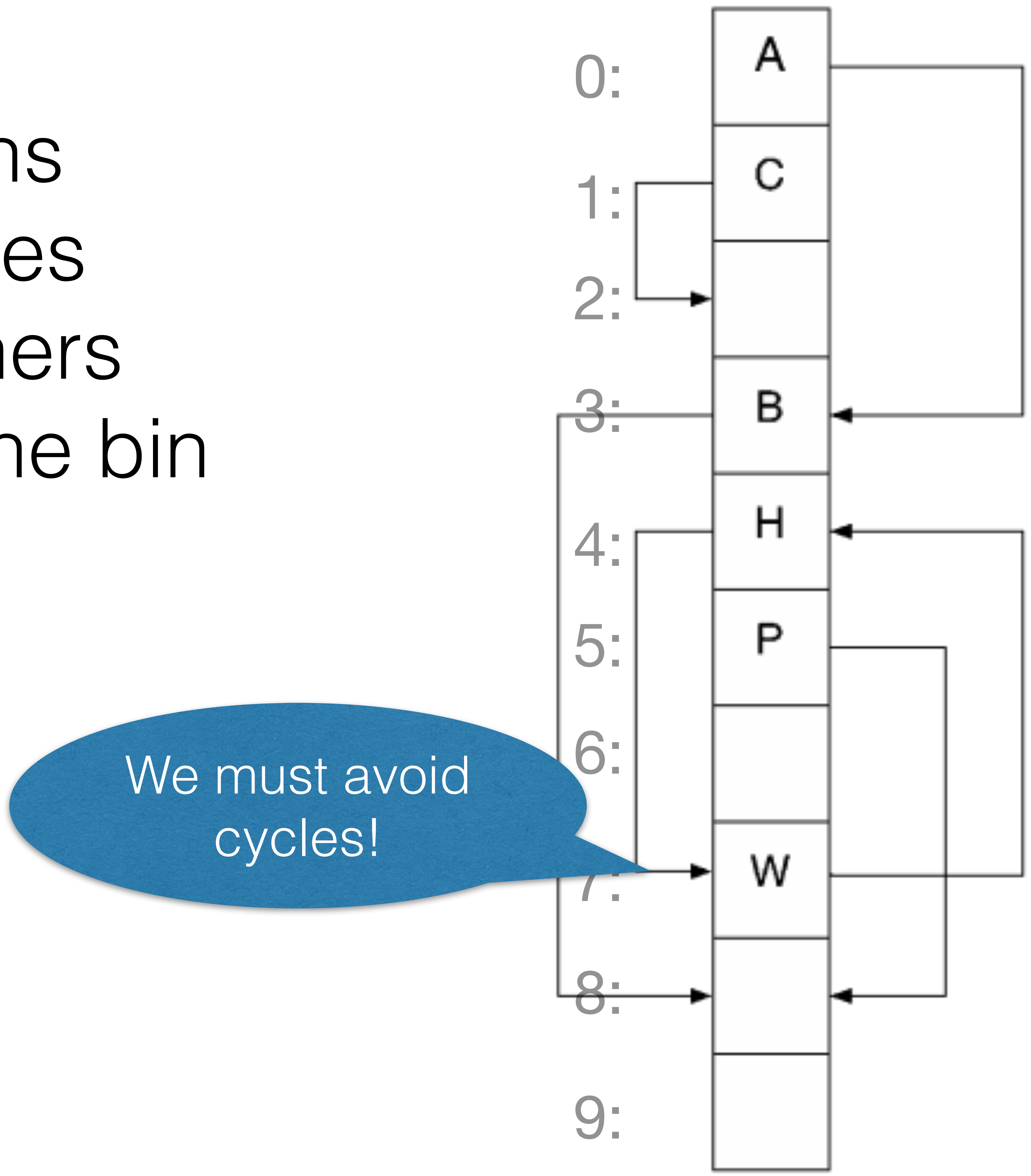
Refresher: Hashtable Basics

- 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:
- Unfortunately we usually have a universe of items U we may wish to store, where $|U|$ is <u>*much much*</u> bigger than m.
- We need strategies for resolving collisions
  - 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”

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
        - We may have a “chain” of evictions, but if chain is too long, we simply “rehash and rebuild”
    - At most 2 checks per lookup
- General technique is called power of two choices

(Extra: Problem)
Membership Queries

Intersection of Systems and Theory

We've spent this class thinking about performance in terms of Big-O

- Great for understanding scaling behavior of our algorithms/DSes
- Not so great for optimizing a given data structure

Problems with Big-O?

- Hides limitations of hardware/environment
  - Ignores importance of locality: both temporal & physical
- We often "count operations", treating different operations as if they were the same cost

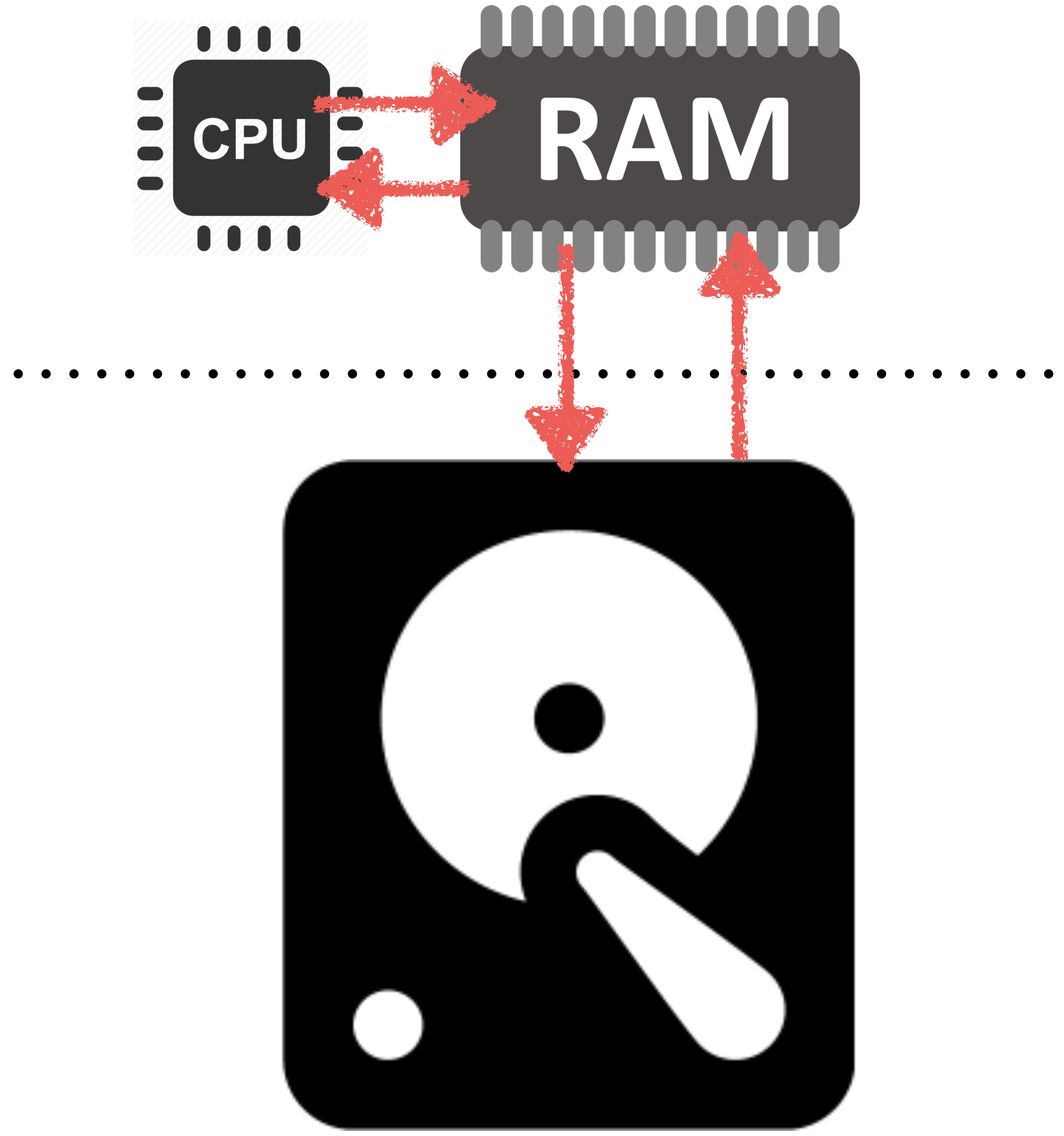
Exciting problems show up when we think about physical implications during our algorithmic design/analysis!

Memory Hierarchy

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

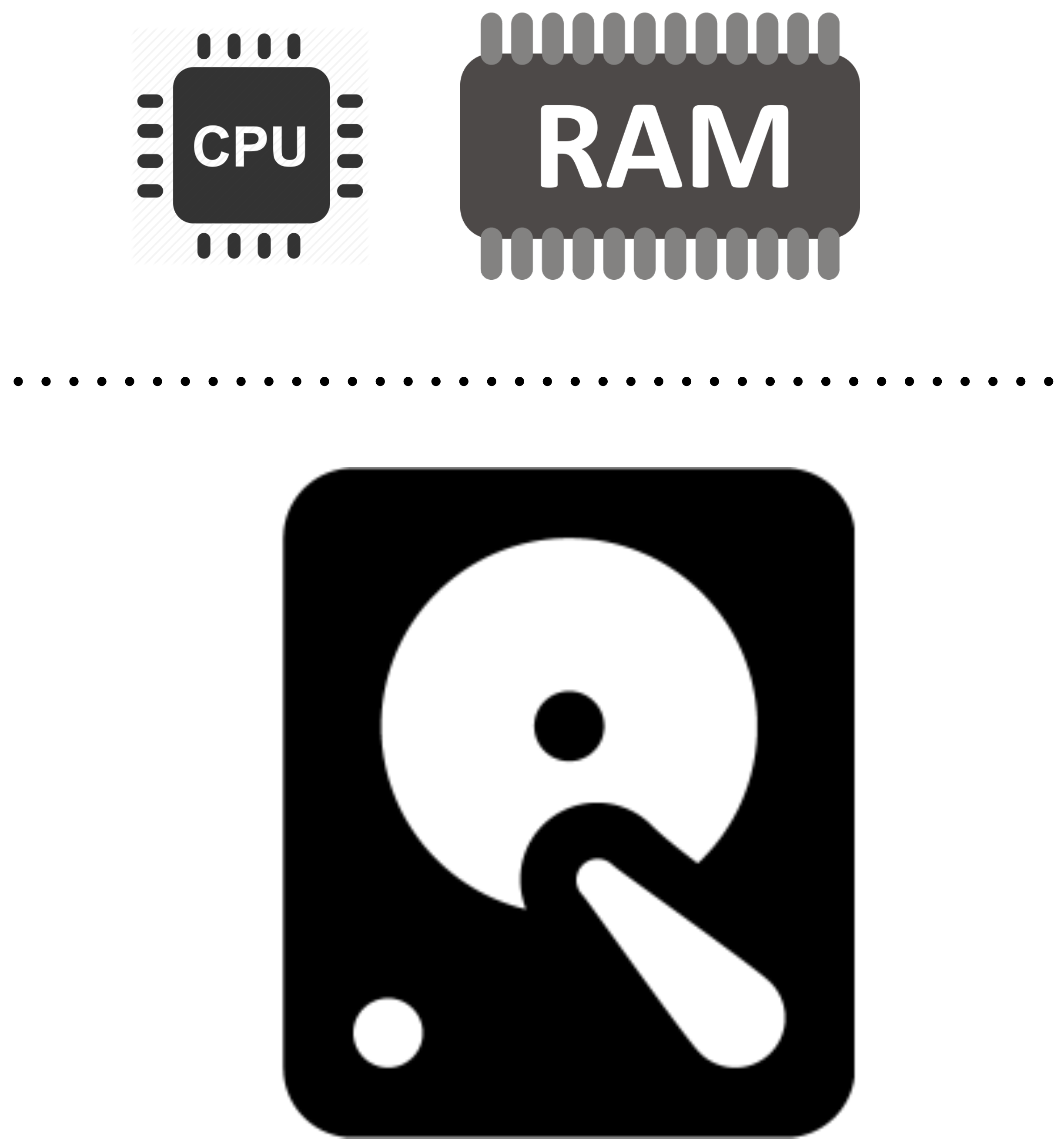
- **Solution:** Store a subset of our data in a cache

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



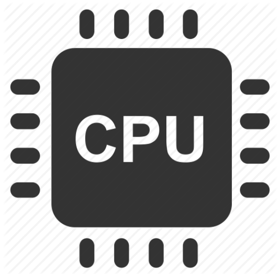
Memory Hierarchy

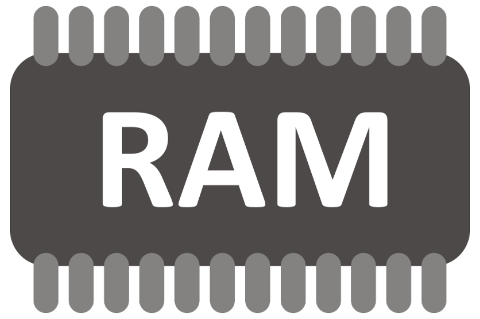
- **Problem 2:** Not all levels in our cache have the same cost



Memory Hierarchy

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

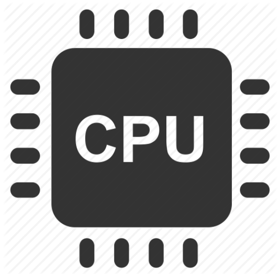


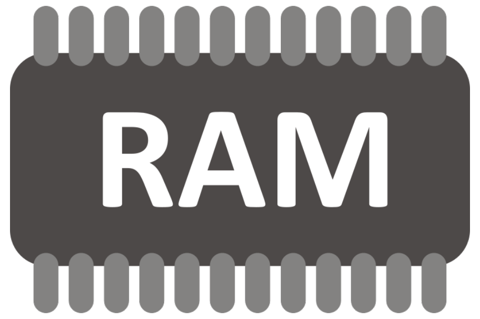


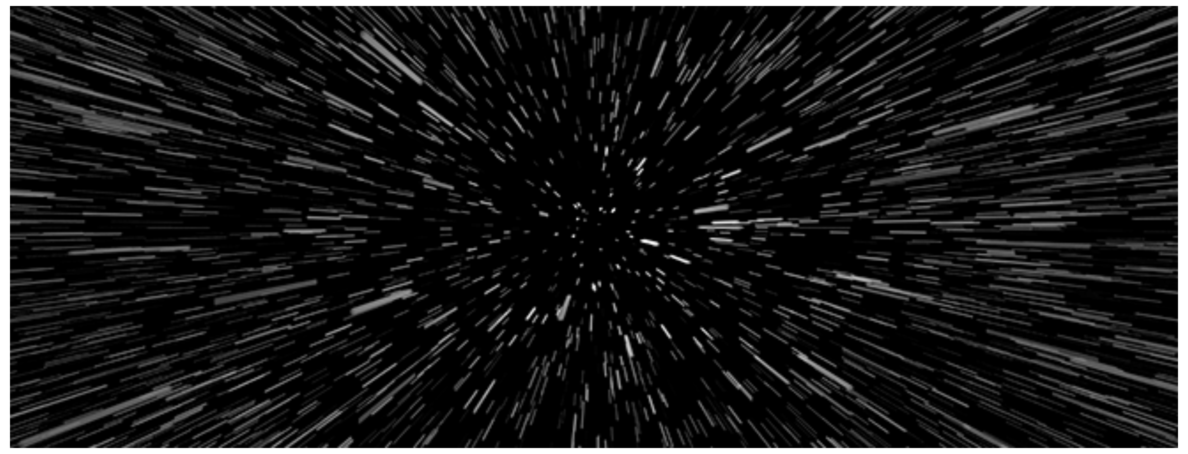
https://www.istockphoto.com/photo/pile-of-money-gm172637949-581154
http://www.freephotosbank.com/photographers/photos1/45/med_53ff4957d796d0ff0a7d3151ec4e4a20.jpg

Memory Hierarchy

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

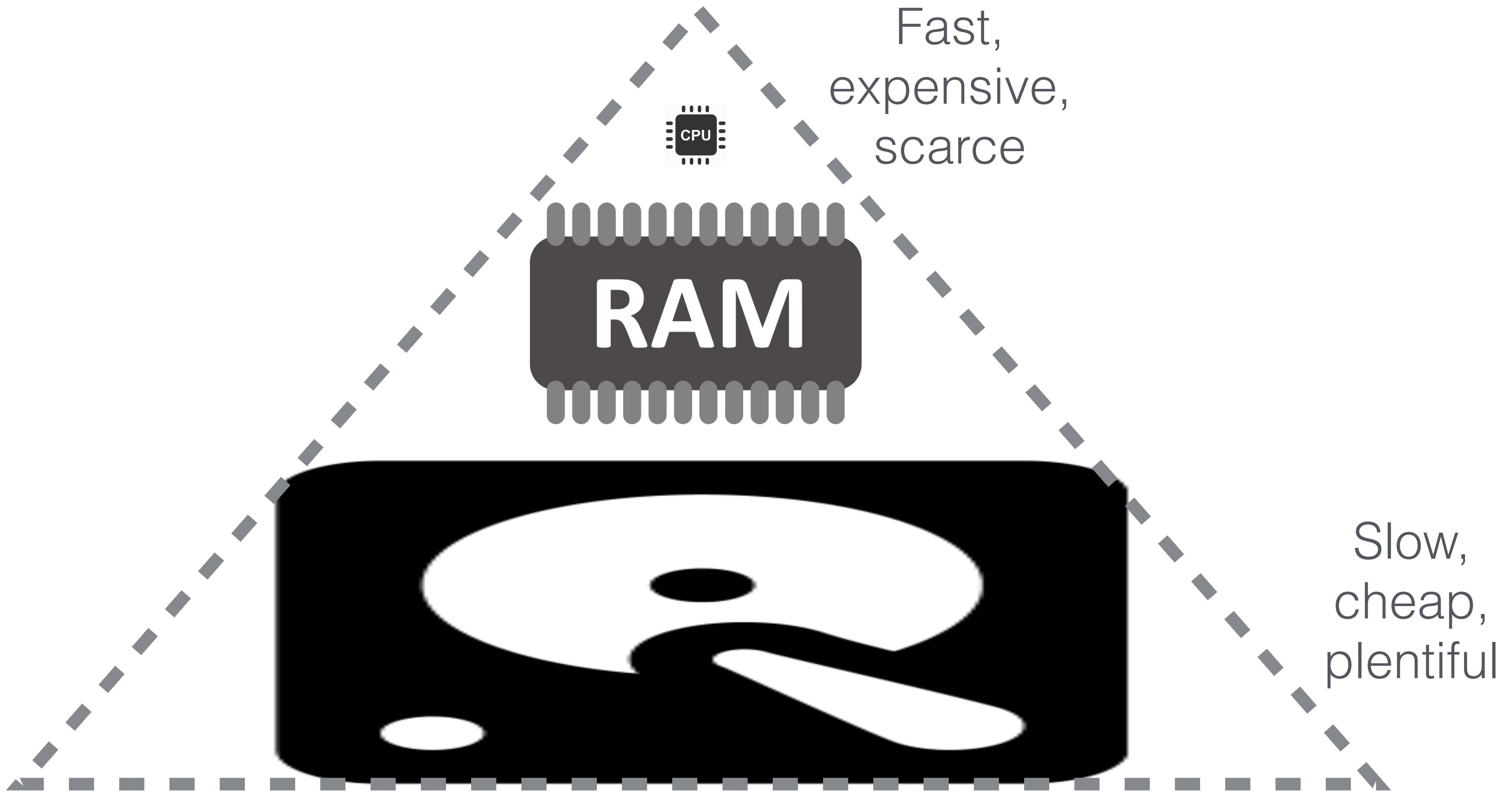







Memory Hierarchy

- 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



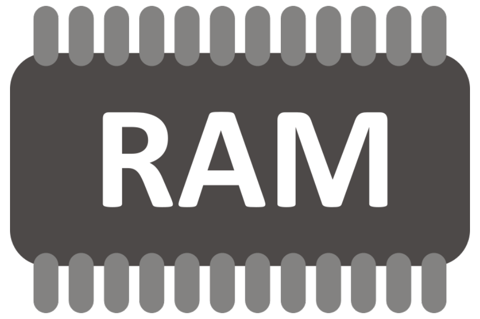
(Contrived) Scenario: Photo Storage

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

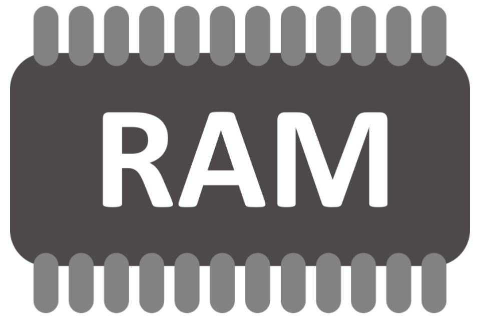


Big, slow

Memory Hierarchy

`get(cat)`

Small, fast



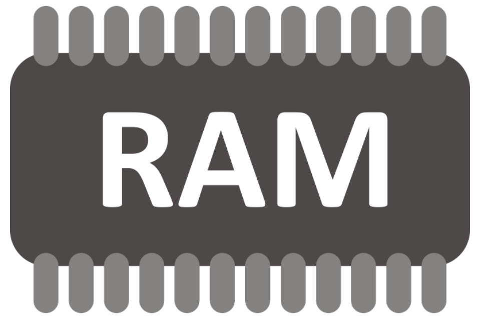
?

Big, slow

Memory Hierarchy

`get(cat)`

Small, fast

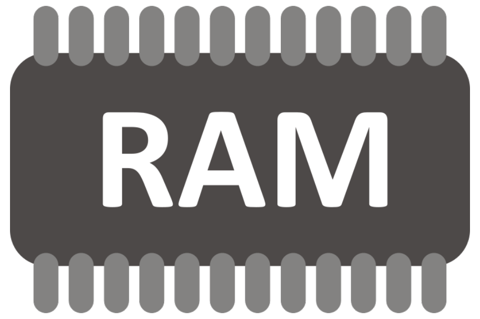


Big, slow

Memory Hierarchy

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

Small, fast



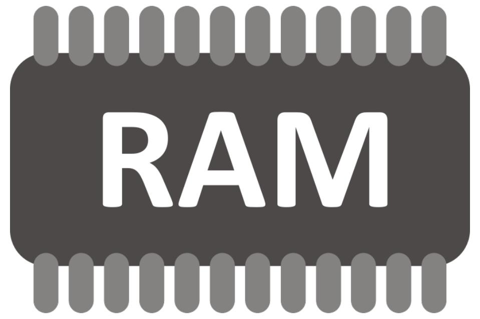
?

Big, slow

Memory Hierarchy

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

Small, fast

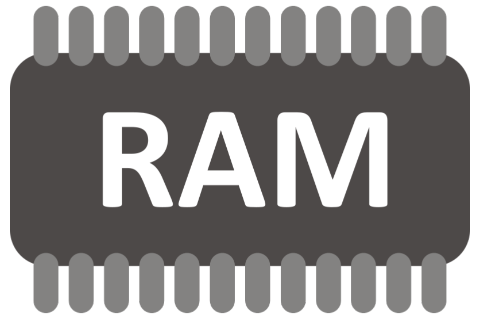


Big, slow

Memory Hierarchy

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

Small, fast



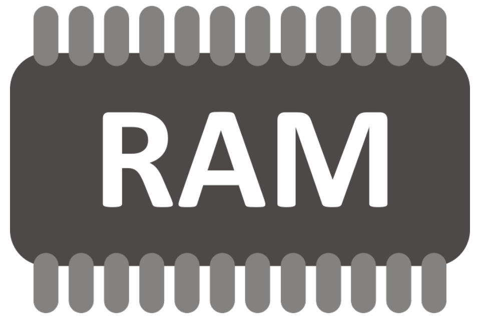
?

Big, slow

Memory Hierarchy

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

Small, fast

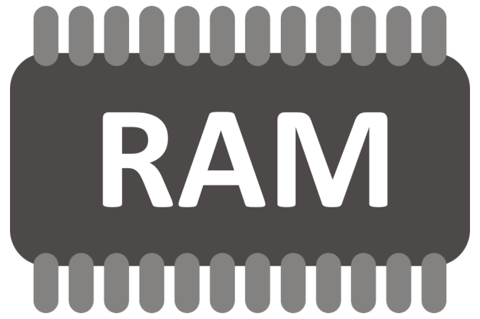


Big, slow

Memory Hierarchy

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

Small, fast



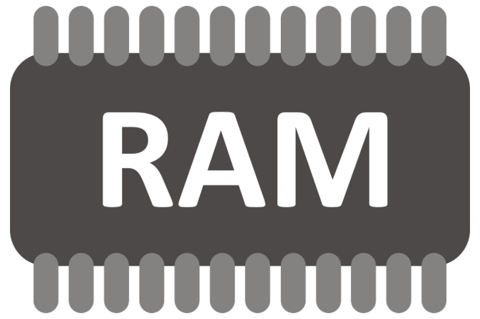
?

Big, slow

Memory Hierarchy

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

Small, fast

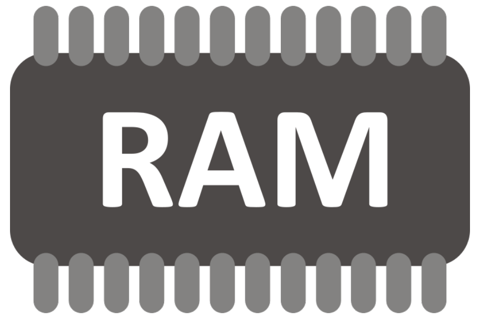


Big, slow

Memory Hierarchy

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

Small, fast



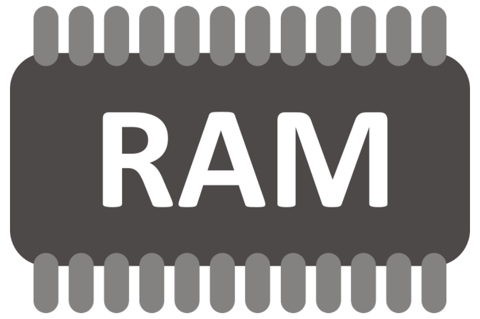
?

Big, slow

Memory Hierarchy

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

Small, fast

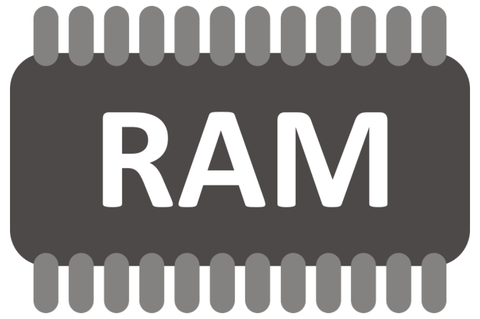


Big, slow

Memory Hierarchy

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

Small, fast



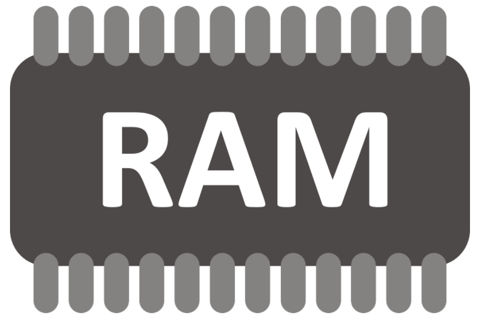
?

Big, slow

Memory Hierarchy

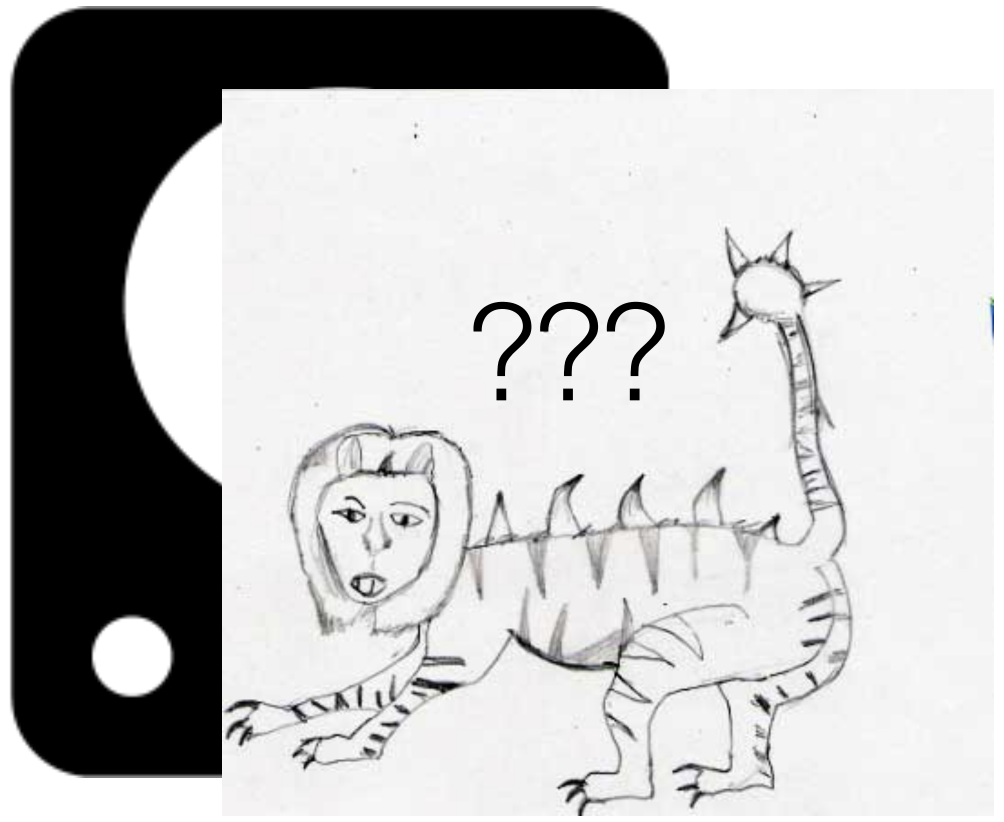
```
get(cat)
get(cow)
get(dog)
get(goat)
get(cat)
get(liger)
```

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
  - False negatives are correctness bugs; that’s **not** OK
- We will build a structure that does **approximate membership queries** and is more efficient than a set.

Bloom Filters

Goal: approximately represent a set of **n** elements using a bit array

- Returns either:
  - Definitely NOT in the set
  - Possibly in the set

Parameters: **m**, **k**

- **m**: Number of bits in the array
- **k**: Set of **k** hash functions { h_1, h_2, …, h_k }, each with range {**0**…**m–1**}

Bloom Filters

Insert(key):

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

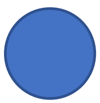
Query(key):

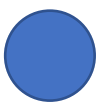
```
for hashFunction_i in hashFuncions_i…k:
   if (bitmap[hashFunction_i(key) % m] != 1):
      return “not in set”
return “maybe in set”
```

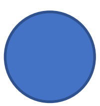
Concrete Example: `k=3`, `m=10`

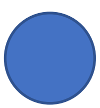
M =

0	0	0	0	0	0	0	0	0	0

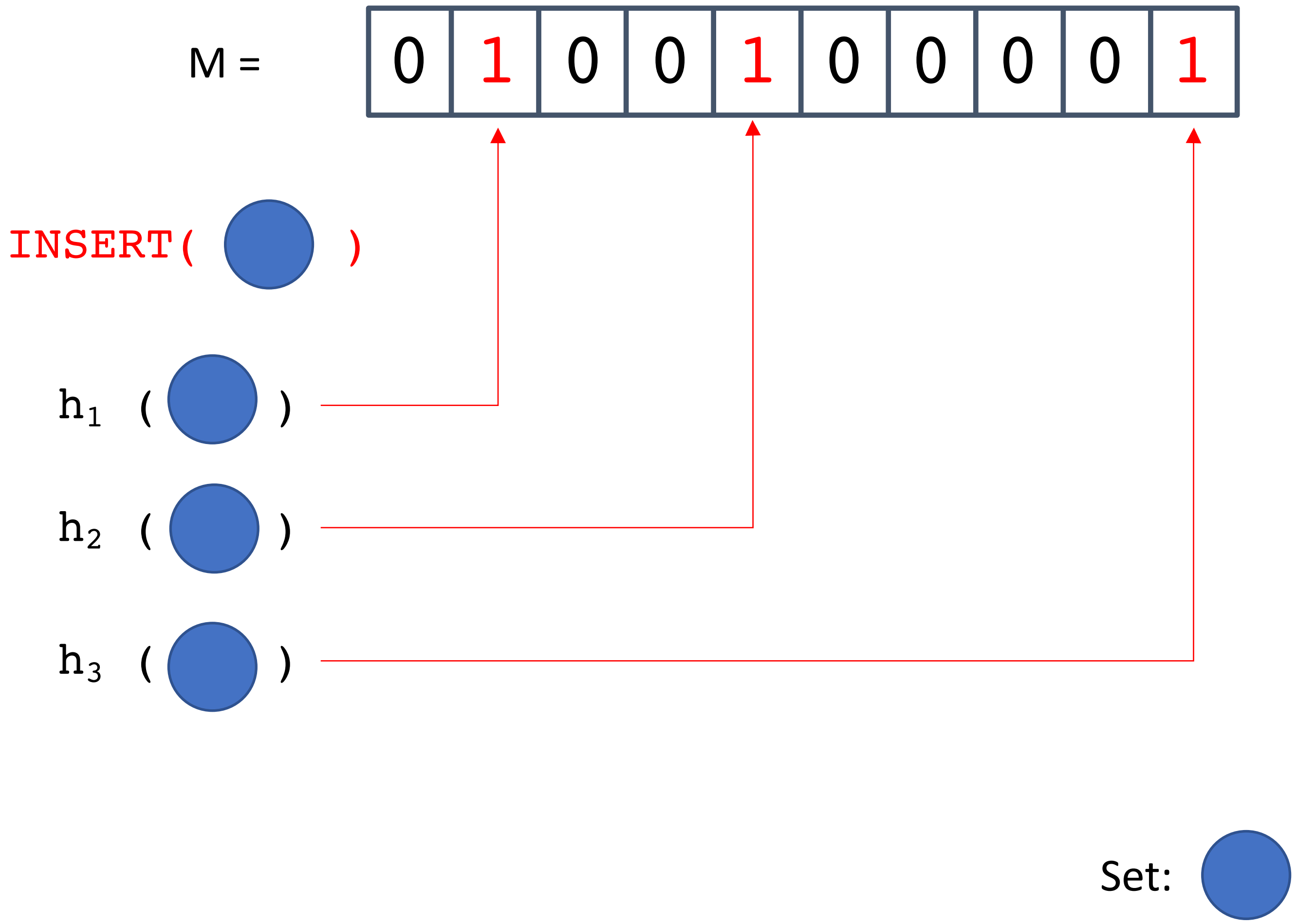
`INSERT(`  `)`

h_1 ()

h_2 ()

h_3 ()

Concrete Example: k=3, m=10



Concrete Example: `k=3`, `m=10`

M =

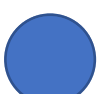
0	1	0	0	1	0	0	0	0	1

`INSERT(`  `)`

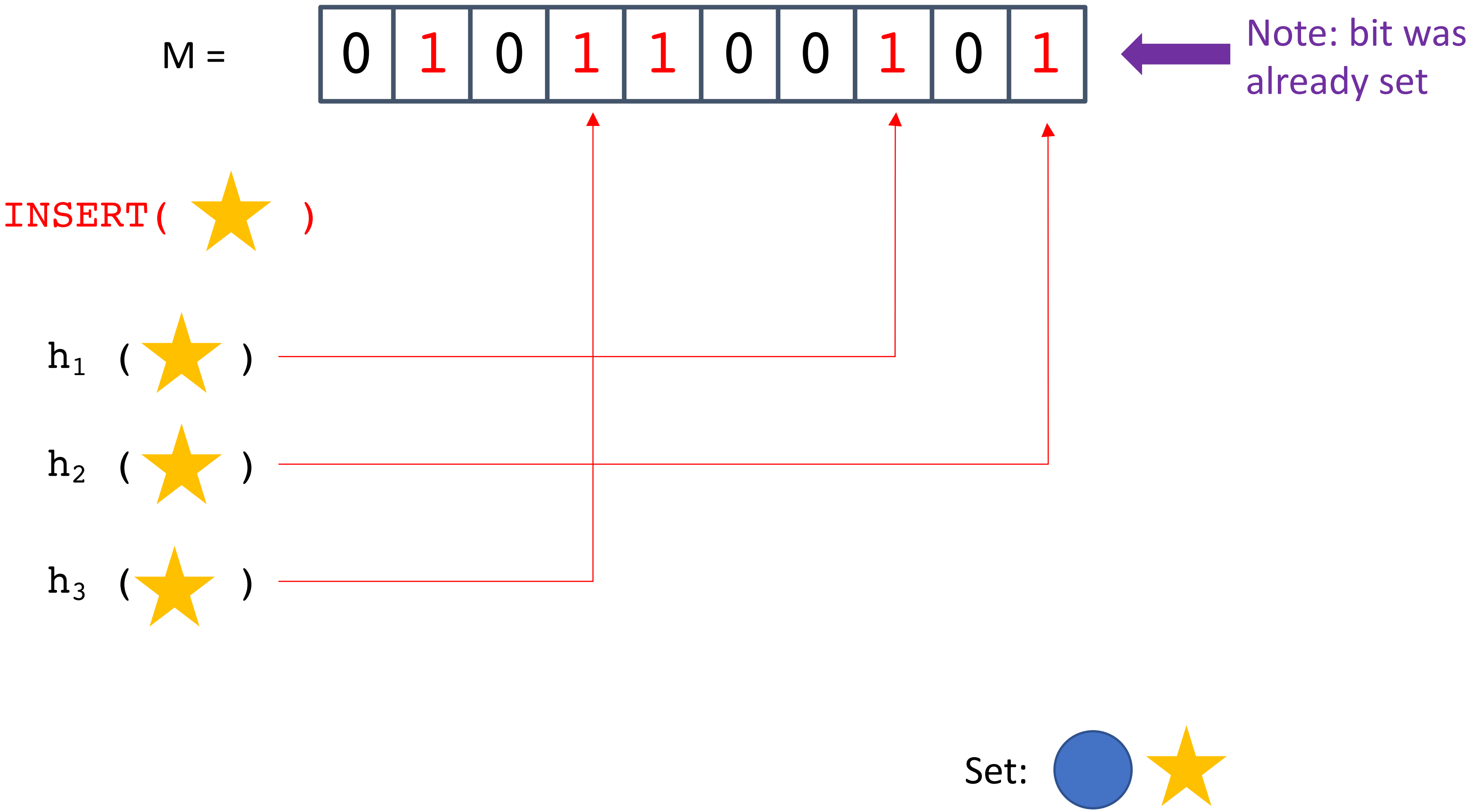
h_1 ()

h_2 ()

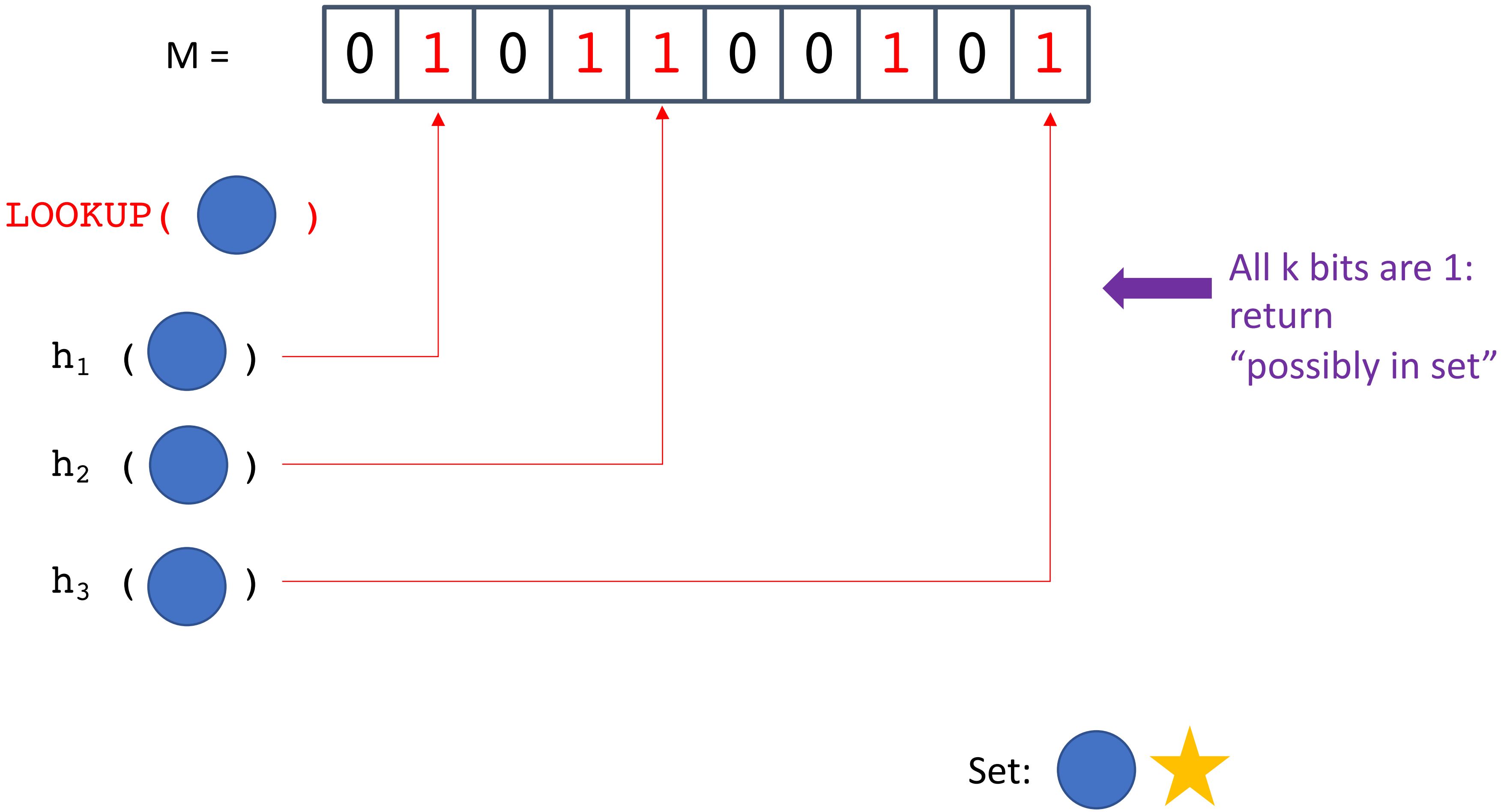
h_3 ()

Set: 

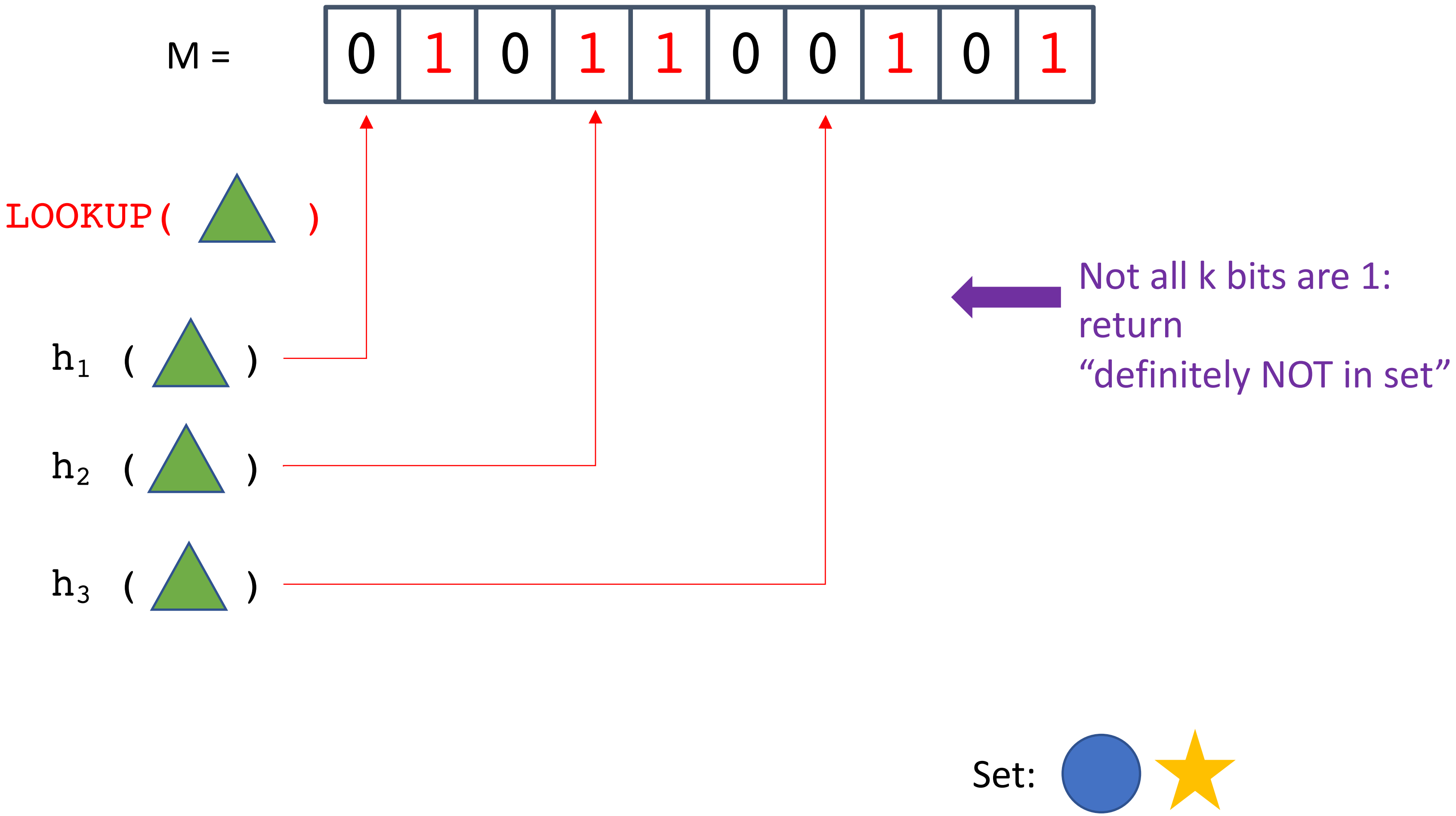
Concrete Example: $k=3$, $m=10$



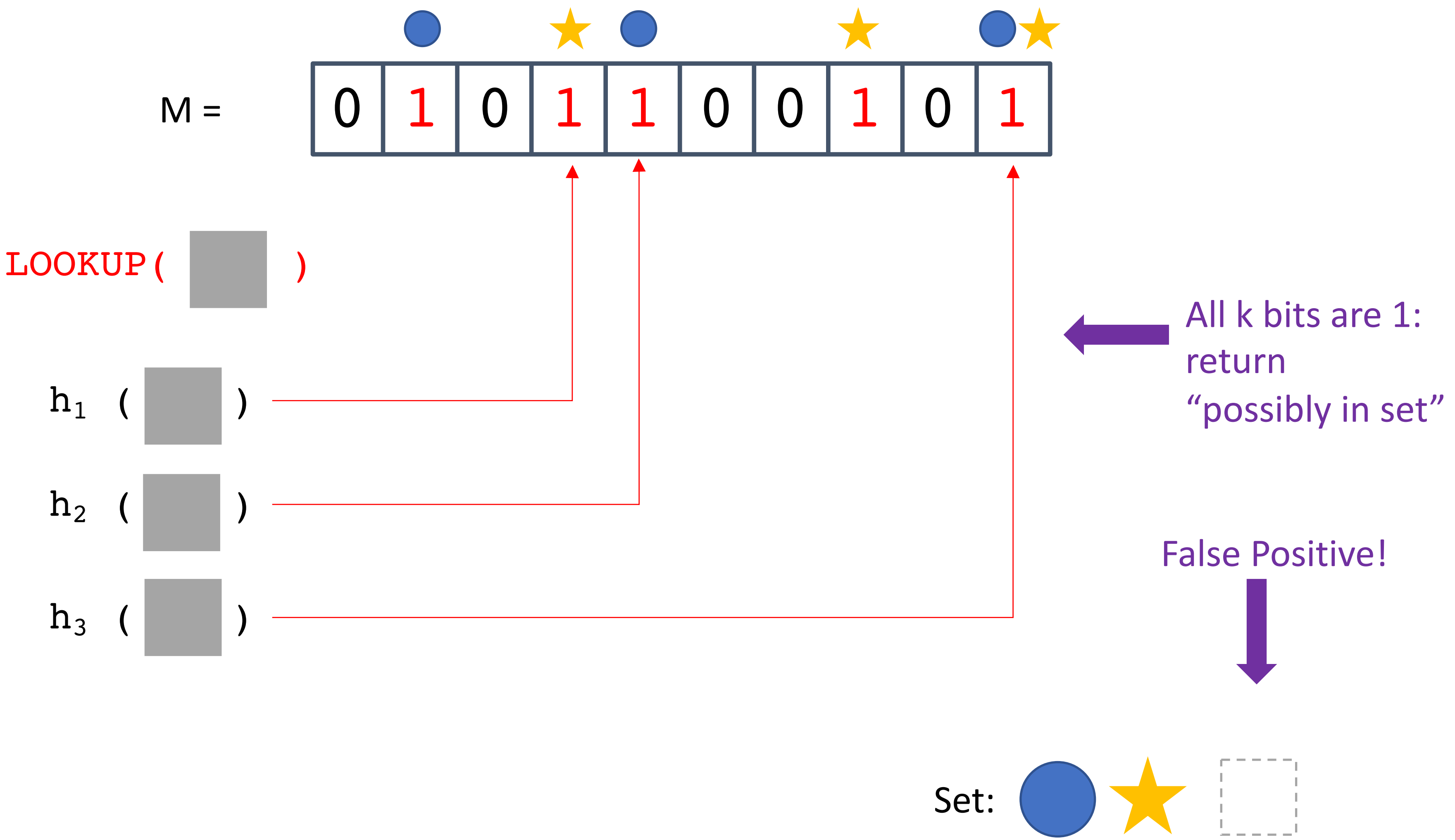
Concrete Example: `k=3`, `m=10`



Concrete Example: $k=3$, $m=10$



Concrete Example: `k=3`, `m=10`



Tuning False Positives

- What happens if we increase `m`?
- What happens if we increase `k`?

- False positive rate f is:

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k$$

P(a given bit is still 0 after n insertions with k independent hash functions)

Bloom Filters

- Are there any problems with Bloom filters?
  - What operations do they support/not support?
  - How do you grow a Bloom filter?
  - What if your filter itself exceeds RAM (how bad is locality)?
    - What does the cache behavior look like?

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

Bloom Filters: Challenges

- What if your filter itself exceeds RAM?
  - What does the cache behavior look like?
    - Good hash functions intentionally create a uniform distribution to avoid “clumping”
    - So even if the filter fits in RAM, the cache locality is poor due to k random accesses

  - If the data set is truly large, there are a few options:
    - Use fewer bits per item (sacrifice precision)
    - Tolerate higher false positive rates
    - Use caching techniques, adding potential for expensive misses

Bloom Filters: Challenges

- What operations do they support/not support?
  - insert? Yes!
  - query? Yes!
  - delete? No! (Multiple items may have “set” any given bit)
  - rename? No! (rename = delete + insert)
  - “count”? No! (maybe/no answers only)

Bloom filter extensions that add support for additional operations do exist, but these operations are not supported by the standard data structure.

Filters: the BIG idea

- Filters are not exact. By embracing approximation, filters can be *memory efficient* data structures
  - Some false positives are allowed
    - Claim something is in the set when it is actually not present
  - But false negatives are never tolerated
    - Claim that something is absent when it is actually present

- Many applications are OK with this behavior
  - Typically filters are used in applications where a wrong answer just wastes work, but does not harm correctness
    - Recall the photo example from before:
      - If we confirm the photo doesn't exist, we don't search (correct)
      - If we mistakenly say the photo exists, all we do is waste the time that we would have needed in the absence of the filter (correct, but slow)