Hashing and Heavy Hitters

"Average" analysis of algorithms

(without sacrificing worst-case robustness)

1. Amortized analysis ("average" in hindsight)

2. Randomized algorithms ("average" over random bits)

Both are very practical, different perspectives

Today:

② Randomized algorithms ("average" over random bits)

Probability	Algorithms
Hash functions,	Hash tables,
Markov's Inequality	Frequency Estimation, Heavy Hitters (in streams)

5.5. Dictionaries

Another useful data type built into Python is the *dictionary* (see Mapping Types — dict). Dictionaries are sometimes found in other languages as "associative memories" or "associative arrays". Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing `list(d)` on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use `sorted(d)` instead). To check whether a single key is in the dictionary, use the `in` keyword.

Here is a small example using a dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

compact1, compact2, compact3
java.util

Interface Map<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

All Known Subinterfaces:

`Bindings`, `ConcurrentMap<K,V>`, `ConcurrentNavigableMap<K,V>`, `LogicalMessageContext`, `MessageContext`, `NavigableMap<K,V>`, `SOAPMessageContext`, `SortedMap<K,V>`

All Known Implementing Classes:

`AbstractMap`, `Attributes`, `AuthProvider`, `ConcurrentHashMap`, `ConcurrentSkipListMap`, `EnumMap`, `HashMap`, `Hashtable`, `IdentityHashMap`, `LinkedHashMap`, `PrinterStateReasons`, `Properties`, `Provider`, `RenderingHints`, `SimpleBindings`, `TabularDataSupport`, `TreeMap`, `UIDefaults`, `WeakHashMap`

```
public interface Map<K,V>
```

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the `Dictionary` class, which was a totally abstract class rather than an interface.

The `Map` interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the `TreeMap` class, make specific guarantees as to their order; others, like the `HashMap` class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the `equals` and `hashCode` methods are no longer well defined on such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void (no arguments) constructor which creates an empty map, and a constructor with a single argument of type `Map`, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.

The "destructive" methods contained in this interface, that is, the methods that modify the map on which they operate, are specified to throw `UnsupportedOperationException` if this map does not support the operation. If this is the case, these methods may, but are not required to, throw an `UnsupportedOperationException` if the invocation would have no effect on the map. For example, invoking the `putAll(Map)` method on an unmodifiable map may, but is not required to, throw the exception if the map whose mappings are to be "superimposed" is empty.

Some map implementations have restrictions on the keys and values they may contain. For example, some implementations prohibit null keys and values, and some have restrictions on the types of their keys. Attempting to insert an ineligible key or value throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible key or value may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible key or value whose completion would not result in the insertion of an ineligible element into the map may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

Many methods in Collections Framework interfaces are defined in terms of the `equals` method. For example, the specification for the `containsKey(Object key)` method says: "returns `true` if and only if this map contains a mapping for a key `k` such that `(key==null ? k==null : key.equals(k))`." This specification should *not* be construed to imply that invoking `Map.containsKey` with a non-null argument `key` will cause `key.equals(k)` to be invoked for any key `k`. Implementations are free to implement optimizations whereby the `equals` invocation is avoided, for example, by first comparing the hash codes of the two keys. (The `Object.hashCode()` specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying `Object` methods wherever the implementor deems it appropriate.

Some map operations which perform recursive traversal of the map may fail with an exception for self-referential instances where the map directly or indirectly contains itself. This includes the `clone()`, `equals()`, `hashCode()` and `toString()` methods. Implementations may optionally handle the self-referential scenario, however most current implementations do not do so.

This interface is a member of the Java Collections Framework.

Since:

`1.2`

See Also:

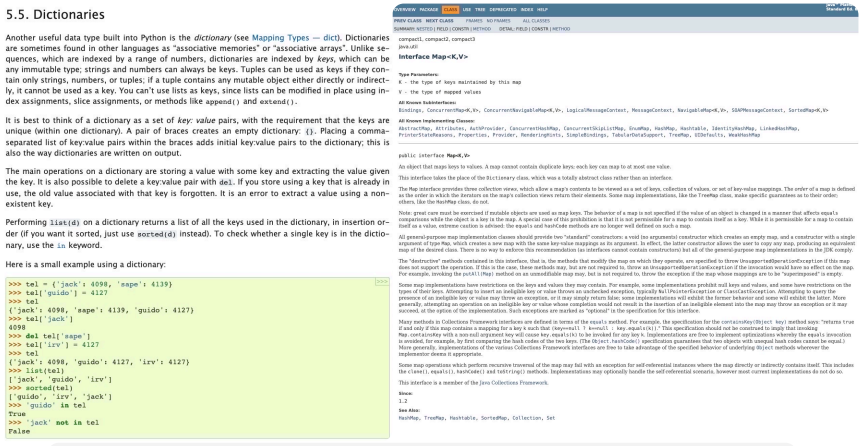
`HashMap`, `TreeMap`, `Hashtable`, `SortedMap`, `Collection`, `Set`

"Dictionaries" / "Maps"

maintain key-value pairs

$(k_1, v_1), (k_2, v_2), \ldots$

- insertion
- deletion
- lookup



5.5. Dictionaries

Another useful data type built into Python is the *dictionary* (see Mapping Types — dict). Dictionaries are sometimes found in other languages as "associative memories" or "associative arrays". Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing `list(d)` on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use `sorted(d)` instead). To check whether a single key is in the dictionary, use the `in` keyword.

Here is a small example using a dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util

Interface Map<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Known Subinterfaces:

Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext, NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>

All Known Implementing Classes:

AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

public interface **Map<K,V>**

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the `Dictionary` class, which was a totally abstract class rather than an interface.

The `Map` interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the `TreeMap` class, make specific guarantees as to their order; others, like the `HashMap` class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void (no arguments) constructor which creates an empty map, and a constructor with a single argument of type `Map`, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.

The "destructive" methods contained in this interface, that is, the methods that modify the map on which they operate, are specified to throw `UnsupportedOperationException` if this map does not support the operation. If this is the case, these methods may, but are not required to, throw an `UnsupportedOperationException` if the invocation would have no effect on the map. For example, invoking the `putAll(Map)` method on an unmodifiable map may, but is not required to, throw the exception if the map whose mappings are to be "superimposed" is empty.

Some map implementations have restrictions on the keys and values they may contain. For example, some implementations prohibit null keys and values, and some have restrictions on the types of their keys. Attempting to insert an ineligible key or value throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible key or value may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible key or value whose completion would not result in the insertion of an ineligible element into the map may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

Many methods in Collections Framework interfaces are defined in terms of the `equals` method. For example, the specification for the `containsKey(Object key)` method says: "returns `true` if and only if this map contains a mapping for a key `k` such that `(key==null ? k==null : key.equals(k))`." This specification should not be construed to imply that invoking `Map.containsKey` with a non-null argument key will cause `key.equals(k)` to be invoked for any key `k`. Implementations are free to implement optimizations whereby the `equals` invocation is avoided, for example, by first comparing the hash codes of the two keys. (The `Object.hashCode()` specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying `Object` methods wherever the implementor deems it appropriate.

Some map operations which perform recursive traversal of the map may fail with an exception for self-referential instances where the map directly or indirectly contains itself. This includes the `clone()`, `equals()`, `hashCode()` and `toString()` methods. Implementations may optionally handle the self-referential scenario, however most current implementations do not do so.

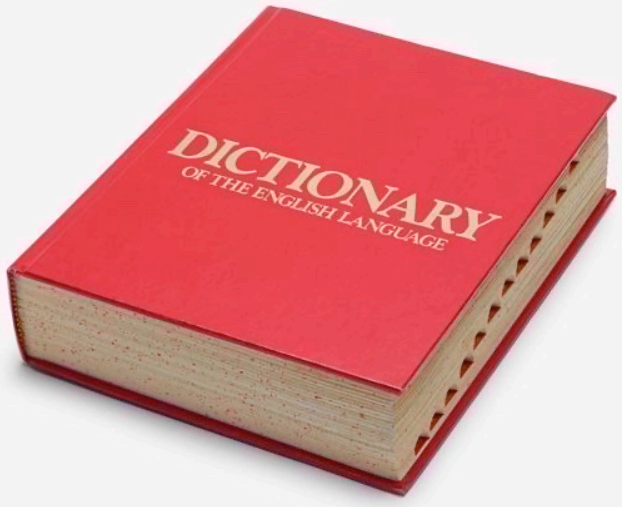
This interface is a member of the Java Collections Framework.

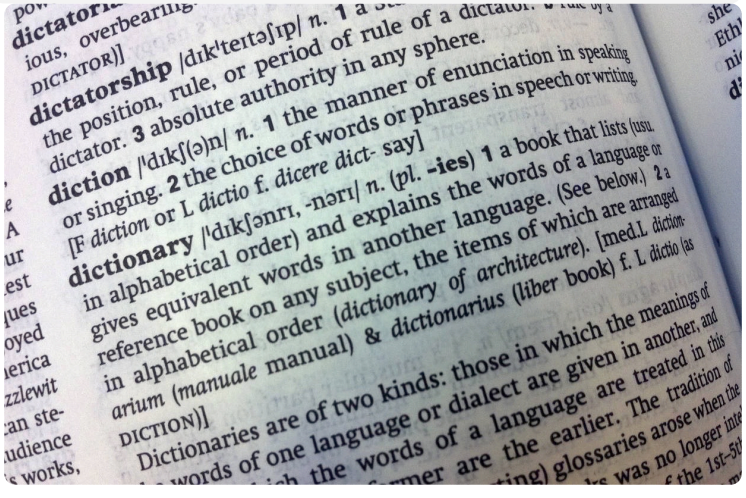
Since:

1.2

See Also:

HashMap, TreeMap, Hashtable, SortedMap, Collection, Set



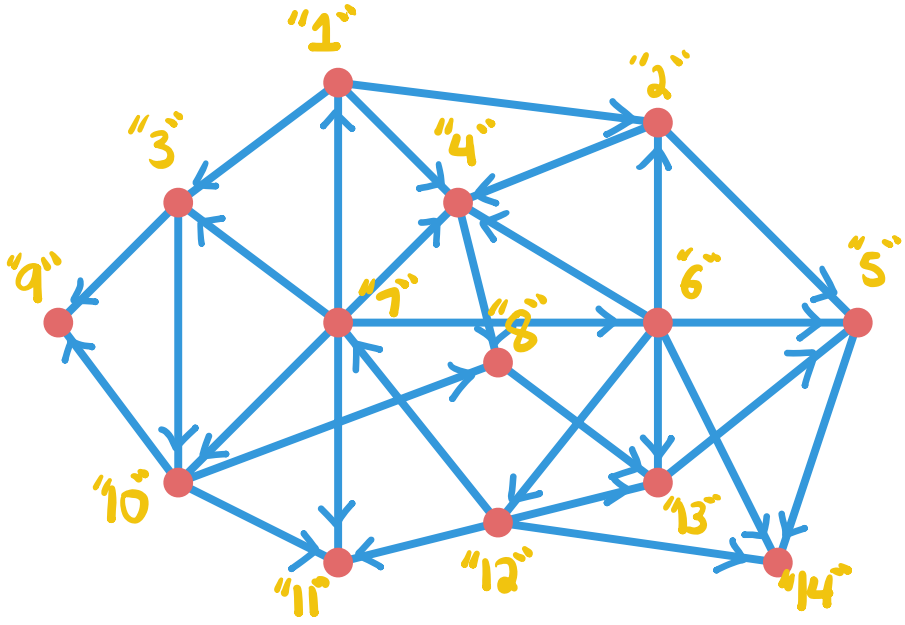


Very special case

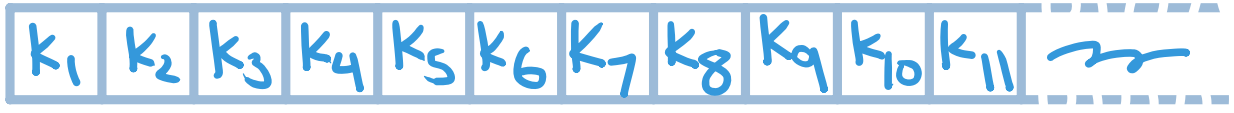
keys = contiguous integers $\{1, \ldots, n\}$

e.g. index vertices of a graph $v_1, \ldots, v_n$

(or after reindexing anything, really)



Solution: Array $A[1 \ldots n]$

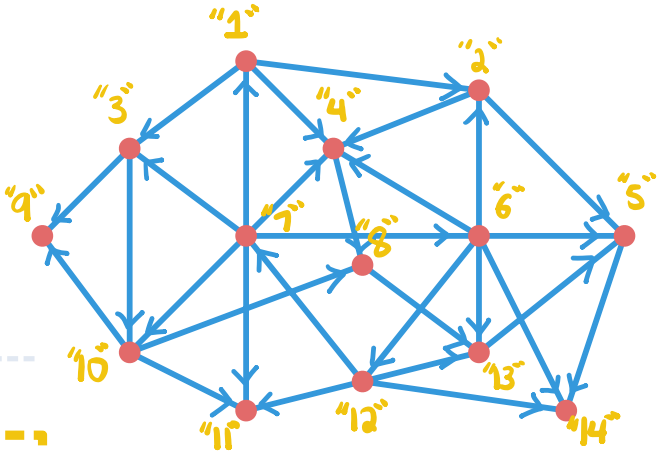


$O(1)$ access, $O(n)$ space

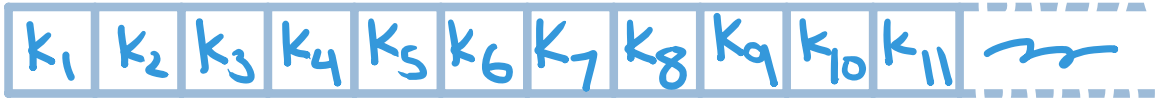
keys = contiguous integers $\{1, \ldots, n\}$

e.g. index vertices of a graph $v_1, \ldots, v_n$

(or after reindexing anything, really)



Solution: Array $A[1 \ldots n]$



$O(1)$ access, $O(n)$ space

Perfect for static settings where we can reindex the input once

Bad for dynamic settings where keys change

More general:

(U for "universe", U large)

keys = n integers out of $\{1, \ldots, U\}$

e.g. strings = #'s via bit encodings

H E L L O

ASCII 72 69 76 76 79

$\Rightarrow 72 + 69 \times 128 + 76 \times 128^2 + 76 \times 128^3 + 79 \times 128^4$

Idea 1: Array $A[1, \ldots, U]$

$O(1)$ lookup / insert / delete

$O(U)$ space ($\ggg O(n)$)

keys = n integers out of $\{1, \ldots, U\}$

(U for "universe", U large)

e.g. strings = #'s via bit encodings

H E L L O

ASCII 72 69 76 76 79

$\Rightarrow 72 + 69 \times 128 + 76 \times 128^2 + 76 \times 128^3 + 79 \times 128^4$

Idea 1: Array $A[1, \ldots, U]$

$O(1)$ lookup / insert / delete

$O(U)$ space ($\ggg O(n)$)

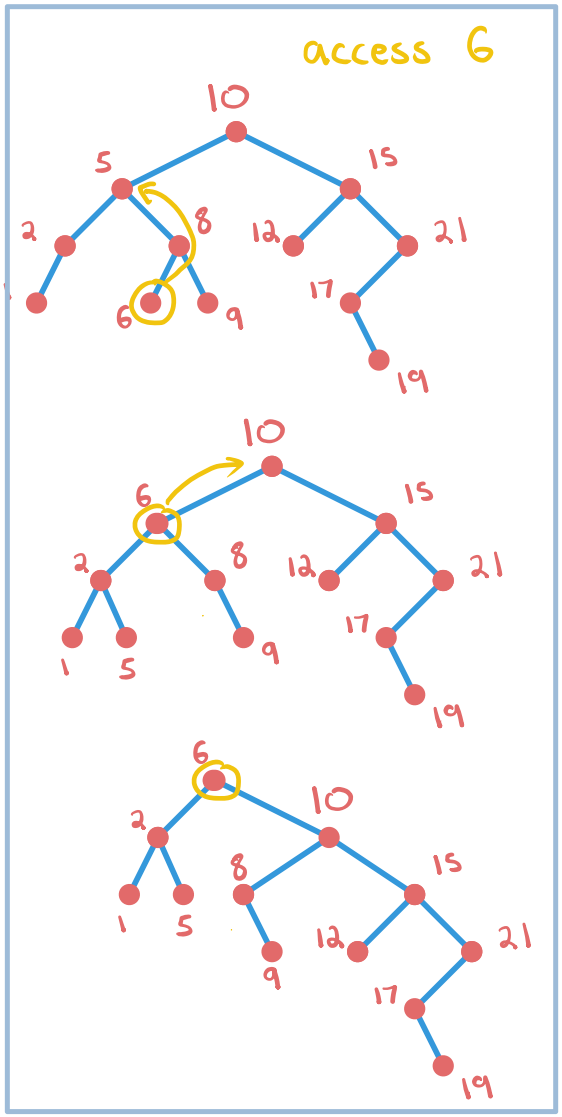
Idea 2: self-balanced trees

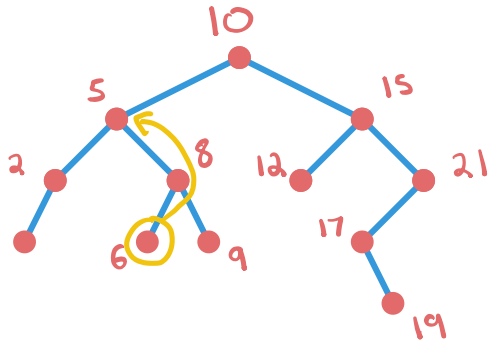
$O(n)$ space

$O(\log n)$ time per lookup, insertion, deletion

(maybe amortized)

e.g. splay trees



	Arrays	Search Trees
		
Time	$O(1)$	$O(\log n)$
Space	$O(U)$	$O(n)$

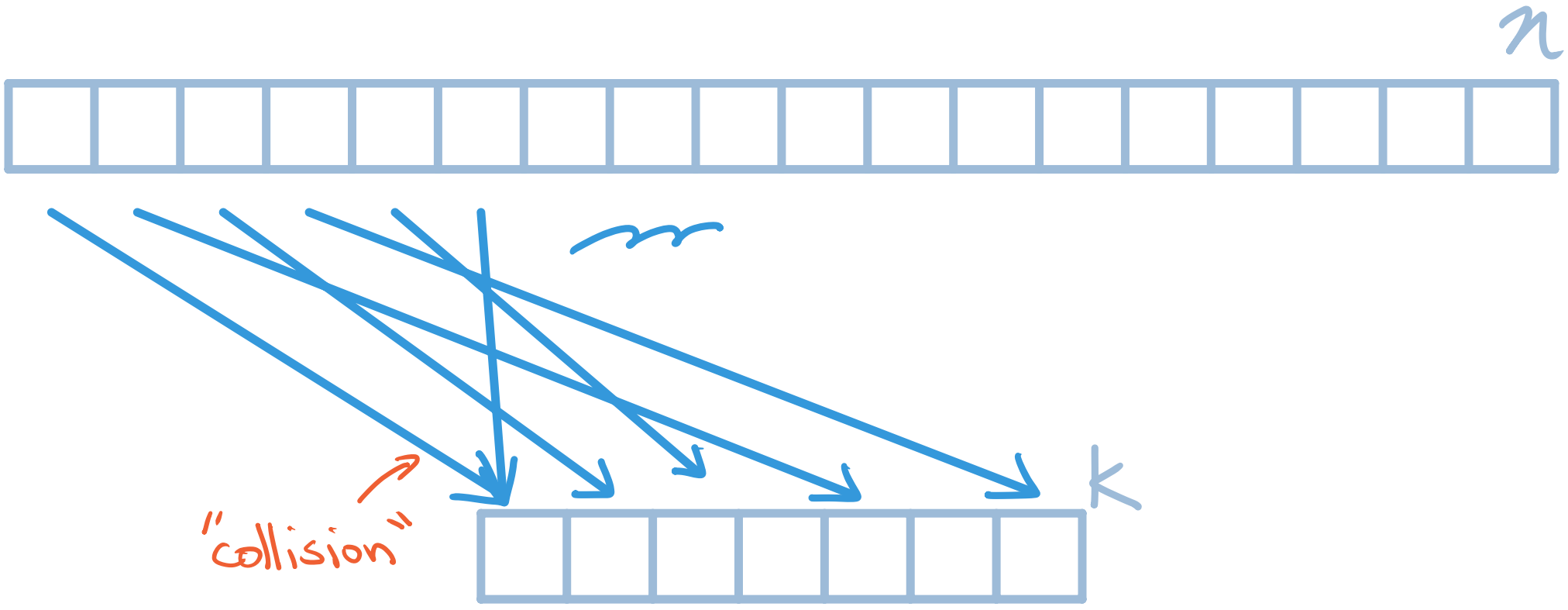
Question: can we get the best of both worlds?

Yes, with randomization

② Hash Functions

Randomly (initialized) constructed function

big $h: [n] \to [k]$ rel. small

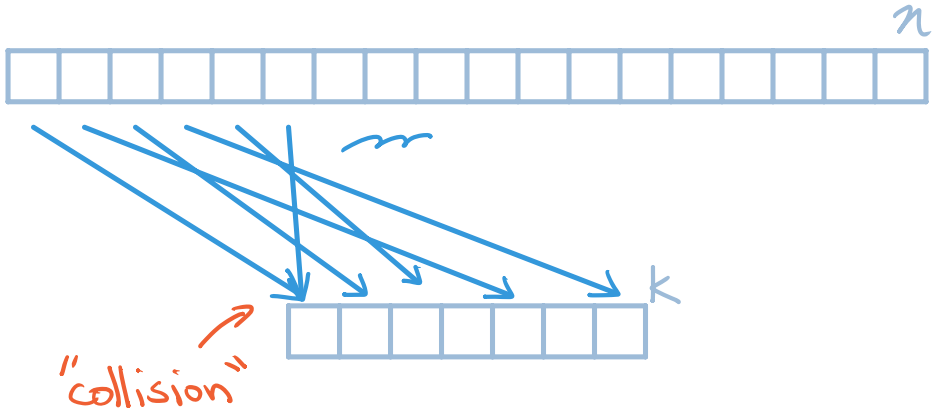


② Hash Functions

Randomly constructed function

$h:[n] \to [k]$

big n → $[n]$; $[k]$ ← rel. small



"Ideal Hash Function"

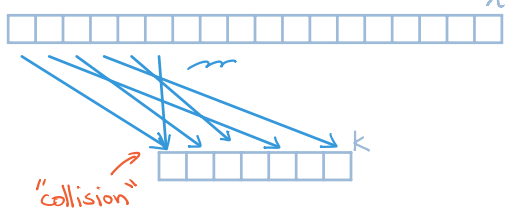
$h:[n] \to [k]$

w/ each $h(i)$ indep. unif. random in $[k]$

↑

requires $\Omega(\log(k^n)) = \Omega(n \log k)$ space ∴

Universal Hash Functions



$h: [n] \to [k]$ s.t. for all distinct $i, j \in [n]$

$$Pr[h(i) = h(j)] \leq \frac{1}{k}$$

(much weaker than ideal hash)

e.g. $h(x) = ((ax + b) \bmod p) \bmod k$

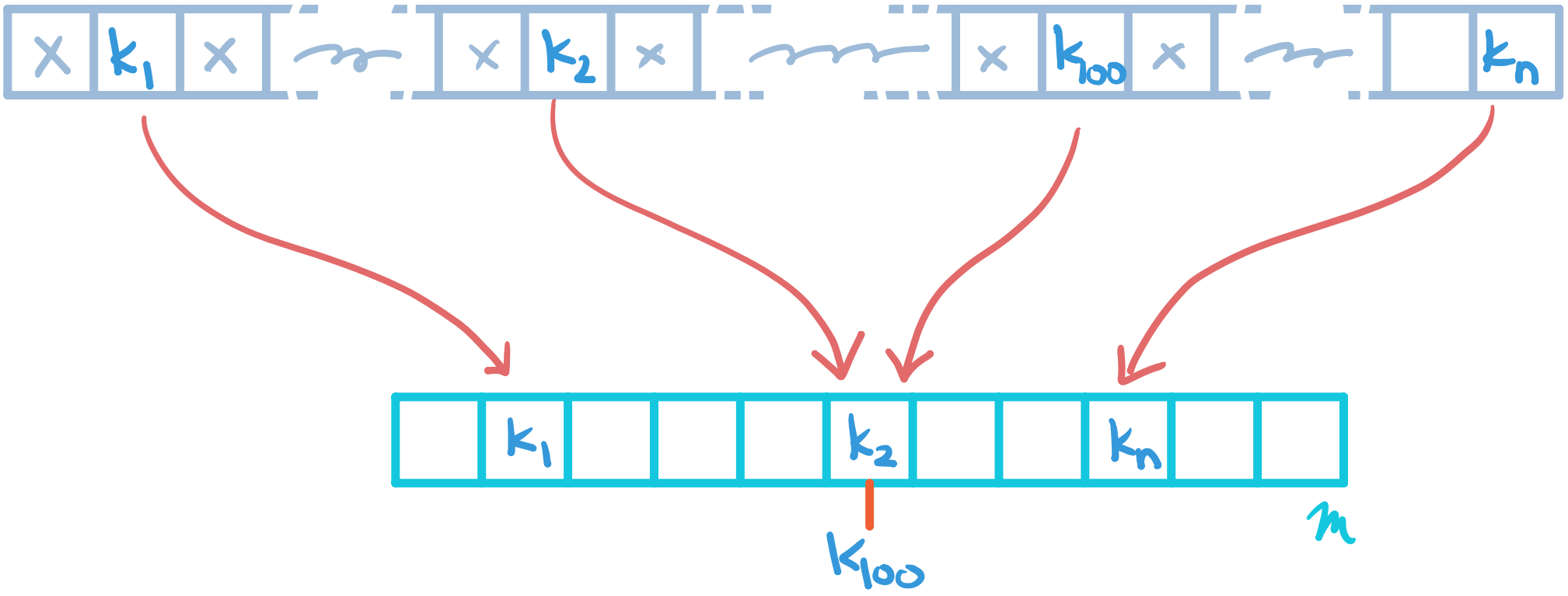
where

- p prime, $> k$
- a in $\{1, \ldots, p\}$ unif. random
- b in $\{0, \ldots, p-1\}$ unif. random

(proof left as exercise)

Hashing

k_i (is stored in) → $h(k_i) \bmod m$



Goal: hash into a smaller array and achieve array-like performance

One problem: collisions

$h(k_1) = h(k_2)$ (same slot)

Some ways to handle collisions

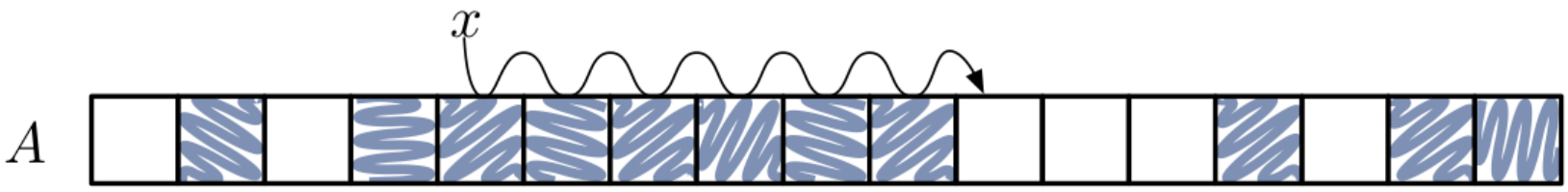
① make array big enough that there are (probably) no collisions

② "chaining": linked list at each cell

③ "cuckoo hashing" use two hash functions h_1 and h_2.

if $h_1(k)$ and $h_2(k)$ already occupied, try to reassign one of the conflicts to their other hash, etc.

④ if A[h(k)] filled, try A[h(k)+1], A[h(k)+2], ...



"linear probing"

(and more)

Chaining + Universal Hash Function

$$h(x) = (ax+b \bmod p) \bmod m$$

Universal Hash Functions

$h: [n] \to k$ s.t. for all pairs $i, j \in [n]$,

$P_r[h(i) = h(j)] \leq \frac{1}{k}$ (much weaker than ideal hash)

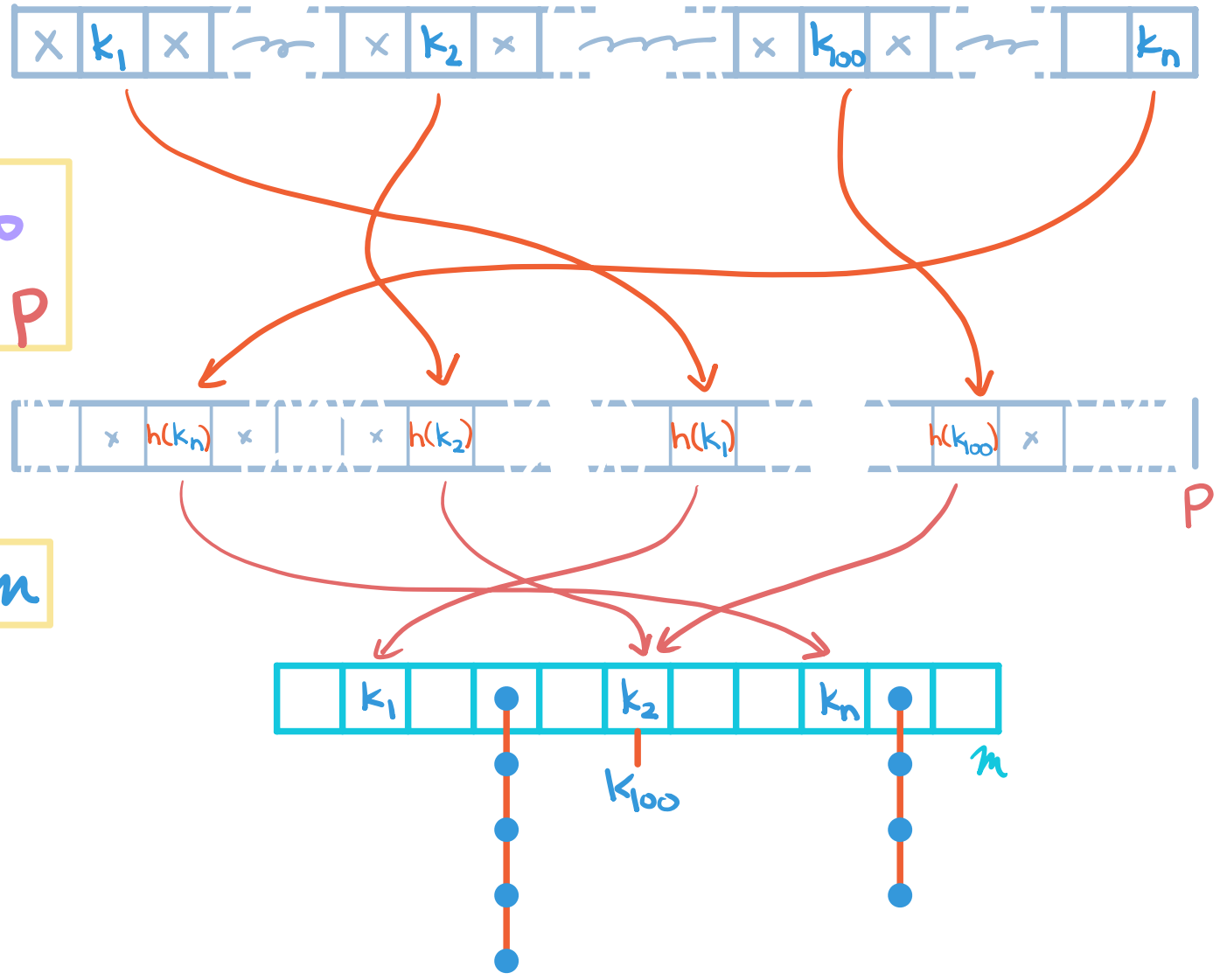
e.g. $h(x) = ((ax+b) \bmod p) \bmod k$

where
- p prime, $> k$
- a in $\{1, \ldots, p\}$ unif. random
- b in $\{0, \ldots, p-1\}$ unif. random

(proofs left as exercise)

$ax+b \bmod p$

$\bmod m$



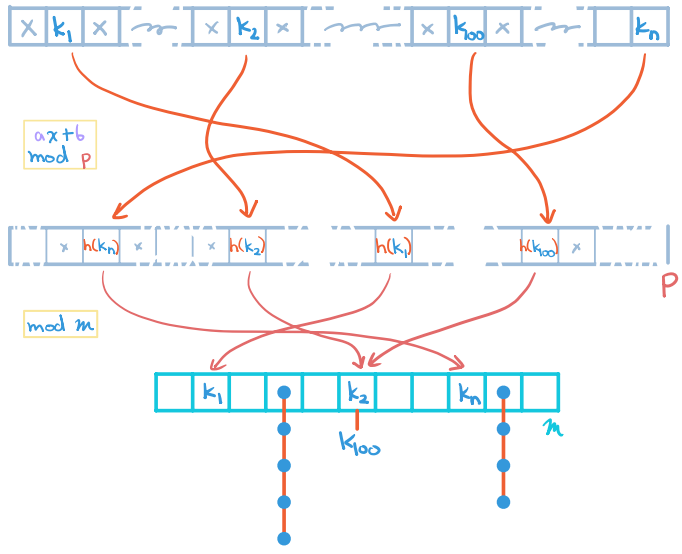
Theorem insert/delete/lookup take

$O(1+n/m)$ time in expectation

Proof

Chaining + universal hash

$h(x) = (ax+b \bmod p) \bmod m$



n = #keys m = size of array

Google Trends

United States

k

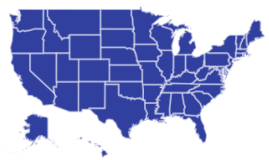
Explore what the world is searching

Enter a search term or a topic

Or start with an example

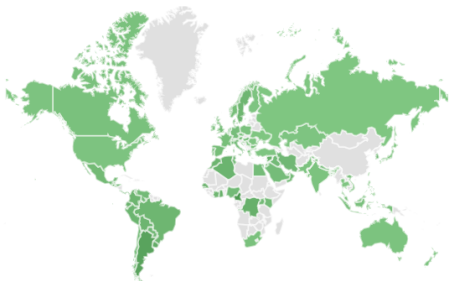
HIDE

Taylor Swift Kim Kardashian



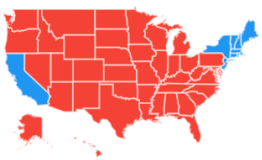
Interest by subregion, Past 7 days, United States

World Cup



Interest by region, Past 7 days, Worldwide

Football American football



Interest by subregion, 2004 - present, United States

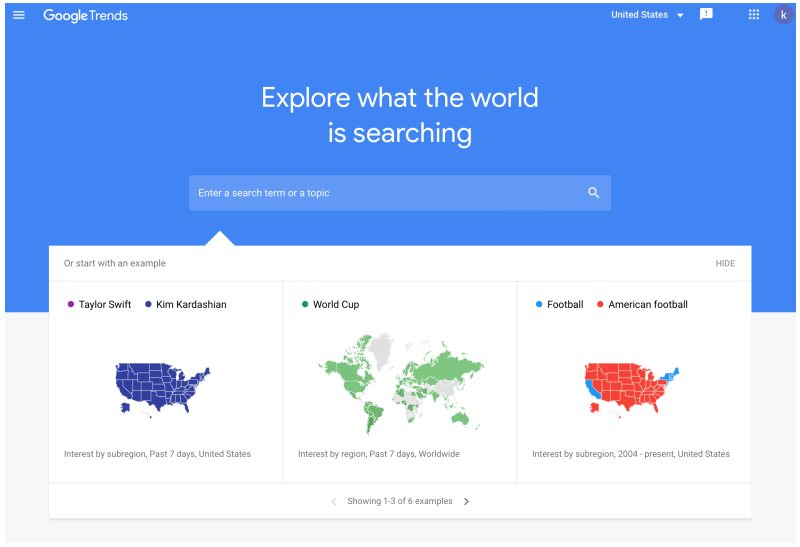
Showing 1-3 of 6 examples

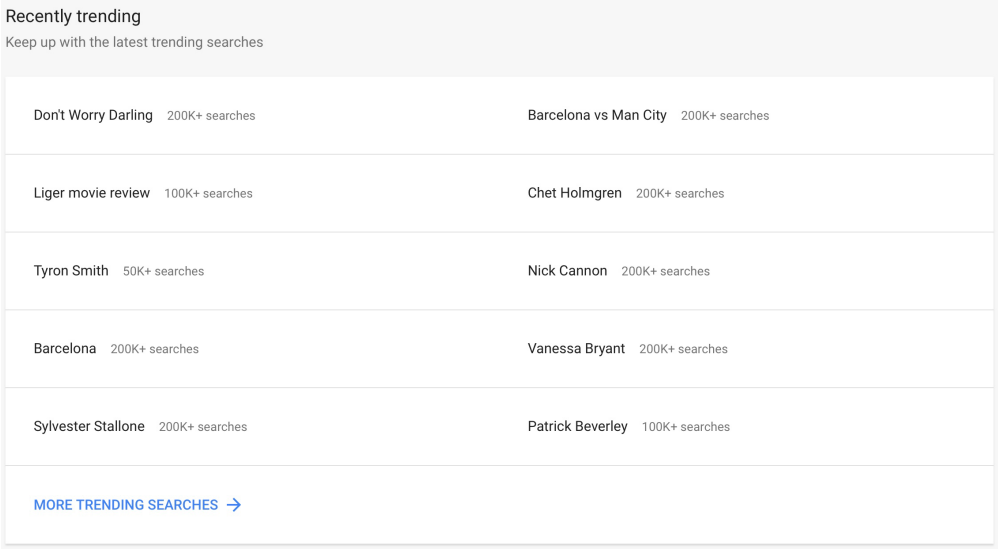
Recently trending

Keep up with the latest trending searches

Don't Worry Darling 200K+ searches	Barcelona vs Man City 200K+ searches
Liger movie review 100K+ searches	Chet Holmgren 200K+ searches
Tyron Smith 50K+ searches	Nick Cannon 200K+ searches
Barcelona 200K+ searches	Vanessa Bryant 200K+ searches
Sylvester Stallone 200K+ searches	Patrick Beverley 100K+ searches

MORE TRENDING SEARCHES →





tracking most popular queries is important for caching

But how?

Way too many different queries to maintain counters for each

Heavy Hitters In Streams

- Elements one by one online
  (can repeat)

a, b, c, b, a, d, a, e, a, b, a, ...

- 5% Heavy Hitters: elements that appear ≥ 5% of the time

  (more generally 5% → parameter $\varepsilon > 0$)

Heavy Hitters In Streams

- Elements one by one online
  (can repeat)

  a, b, c, b, a, d, a, e, a, b, a, $

- 5% Heavy Hitters: elements that appear $\geq 5\%$ of the time

  (more generally 5% → parameter $\varepsilon > 0$)

Goal: identify all heavy hitters w/ sublinear space

Saving all counts in a dictionary takes too much space

e.g. $O(\sqrt{n})$, $O(\log n)$ where all elements are IDs between 1 and n

- 5% Heavy Hitters: elements that appear $\geq 5\%$ of the time

  (more generally 5% → parameter $\varepsilon > 0$)

Goal: identify <u>all</u> heavy hitters w/ sublinear space

How is this even possible?

(a) Only 20 (5%)-heavy hitters

(b) Randomization: allow small prob. of failure

(c) Approximation: allow some margin of error

rigorous, worst case bounds

Today:

② Randomized algorithms ("average" over random bits)

Probability

Hash functions,

Markov's Inequality

Algorithms

Hash tables,

Frequency Estimation,

Heavy Hitters (in streams)

Markov's Inequality (special case)

Let $X \geq 0$ be nonnegative RV

$$\Pr[X \geq 2E[X]] \leq \frac{1}{2}$$

e.g.

$\leq 50\%$ of US popul. earns 2× average

$\leq 50\%$ of NBA players score 2× league average points

$\leq 50\%$ students get 2× average midterm score

Markov's Inequality

Let $X \geq 0$ be nonnegative RV

$$\Pr[X \geq 2E[X]] \leq \frac{1}{2}$$

more generally, for $\alpha \geq 1$,

$$\Pr[X \geq \alpha E[X]] \leq \frac{1}{\alpha}$$

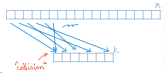
Today:

② Randomized algorithms ("average" over random bits)

Probability

Hash functions,

Universal Hash Functions



$h: [n] \to k$ s.t. for all pairs $i, j \in [n]$,

$$\Pr[h(i) = h(j)] \leq \frac{1}{k}$$

(much weaker than ideal hash)

e.g. $h(x) = ((ax+b) \bmod p) \bmod k$

where
- p prime, $> k$
- a in $\{1, \sim, p\}$ unif. random
- b in $\{0, \sim, p-1\}$ unif. random

Markov's Inequality

Markov's Inequality (special case)

Let $X \geq 0$ be nonnegative RV

$$\Pr[X \geq 2E[X]] \leq \frac{1}{2}$$

Algorithms

Frequency Estimation, Heavy Hitters (in streams)

Heavy Hitters In Streams

- Elements one by one online
  (can repeat)

a, b, c, b, a, d, a, e, a, b, a, f

- 5% Heavy Hitters: elements that appear ≥ 5% of the time

  (more generally 5% → parameter $\varepsilon > 0$)

Heavy Hitters In Streams

- Elements one by one online
  (can repeat)

  a, b, c, b, a, d, a, e, a, b, a, ...

- 5% Heavy Hitters: elements that appear ≥ 5% of the time
  (more generally 5% → parameter $\varepsilon > 0$)

Goal: identify all heavy hitters w/ sublinear space

Saving all counts in a dictionary takes too much space

e.g. $O(\sqrt{n})$, $O(\log n)$ where all elements are IDs between 1 and n

From now on:

ε-heavy hitters for fixed $\varepsilon > 0$

Related Problem

Heavy Hitters In Streams

- Elements one by one online (can repeat)

  a, b, c, b, a, d, a, e, a, b, a, $

- 5% Heavy Hitters: elements that appear $\geq 5\%$ of the time

  (more generally 5% → parameter $\varepsilon > 0$)

estimate relative frequency of each element

(up to ε additive error)

(absolute freq x) = # times x appears in stream

$$\begin{pmatrix}\text{relative}\\ \text{freq. of } x\end{pmatrix} = \frac{(\text{abs. freq. } x)}{\text{total length of stream}}$$

ε-heavy hitter $\Leftrightarrow$ relative freq $\geq \varepsilon$

we will focus on estimating up to ε-error

So to recap:

want to estimate all relative frequencies
up to ε-additive error

not enough space for all counters

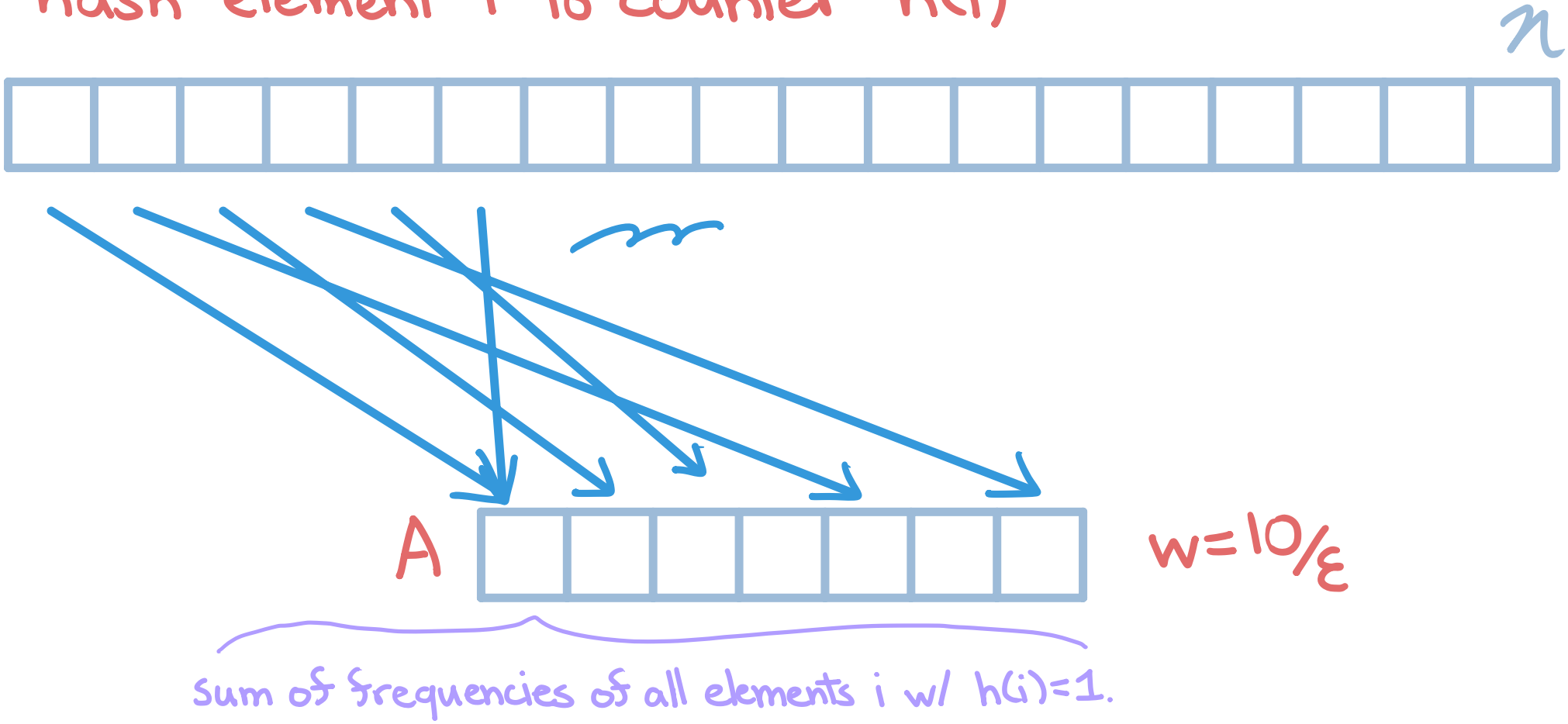
lucky for us:
0 is a good estimate for all but $\frac{1}{\varepsilon}$ elements

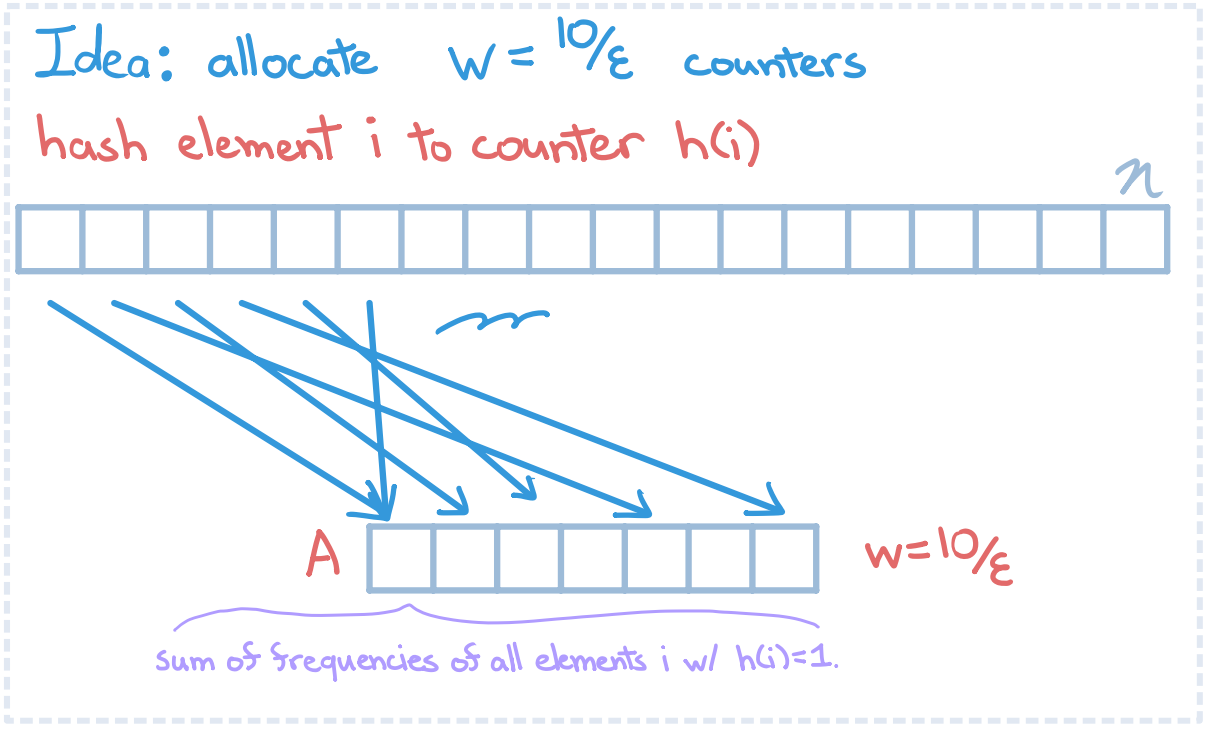
still: don't know which $\frac{1}{\varepsilon}$ elements to count

Hashing has entered the chat

Idea: allocate $w = 10/\varepsilon$ counters

hash element i to counter $h(i)$

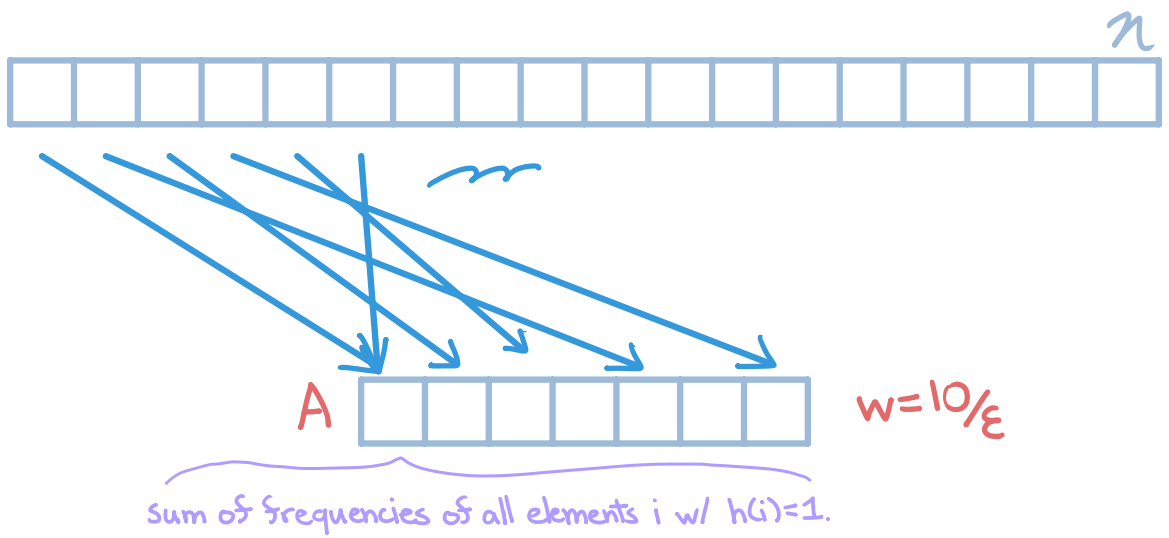




To estimate frequency f_i of i:

return h(i)

(overcounts f_i)



Notation

$A[1, \sim, w]$ = array of counters

f_i = (true) frequency of i

$h: [n] \to [w]$: universal hash function

m = length of stream

$A[j]$

e.g. $A[j] = \sum_{i: h(i)=j} f_i$

Lemma For all i,

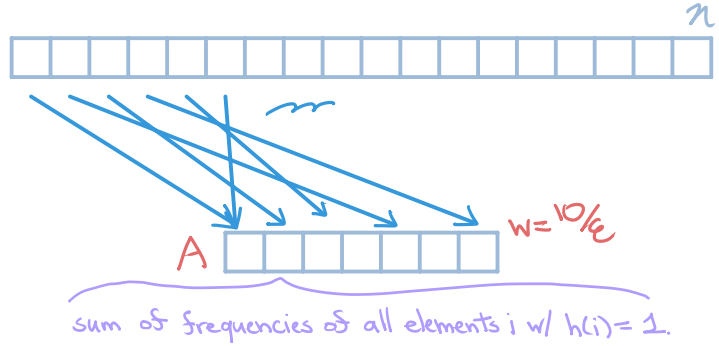
w/ probability $\geq \frac{1}{2}$,

$$f_i \leq A[h(i)] \leq f_i + \frac{\varepsilon}{5} m$$

Universal Hash Functions

$h: [n] \to [k]$ s.t. for distinct $i, j \in [n]$

$$Pr[h(i) = h(j)] \leq \frac{1}{k}$$



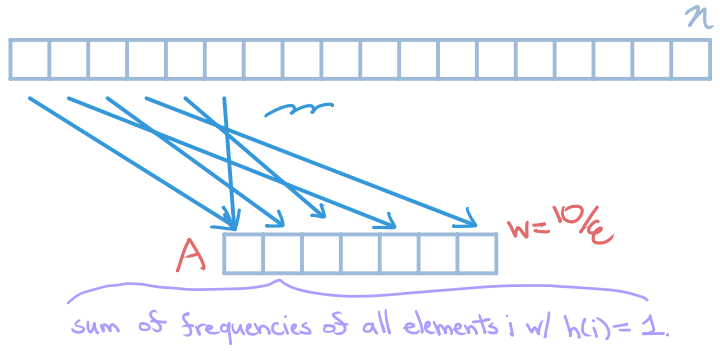
$A[1 \ldots m, w]$ = array of counters

f_i = (true) frequency of i

$h: [n] \to [w]$: universal hash fn.

m = length of stream

Recap:



$A[1\ldots m, w]$ = array of counters

f_i = (true) frequency of i

$h: [n] \to [w]$: universal hash fn.

m = length of stream

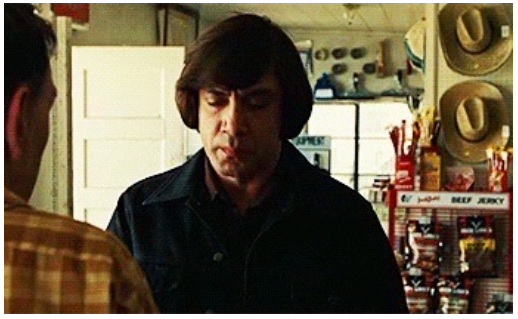
Lemma For all i, w/ probability $\geq \frac{1}{2}$,

$$f_i \leq A[h(i)] \leq f_i + \frac{\varepsilon}{5} m$$

we have: for each element, good estimate of frequency w/ constant probability

we want: estimates of all frequencies w/ high probability

we want to "amplify" our bounds



Repeatedly flip coins:

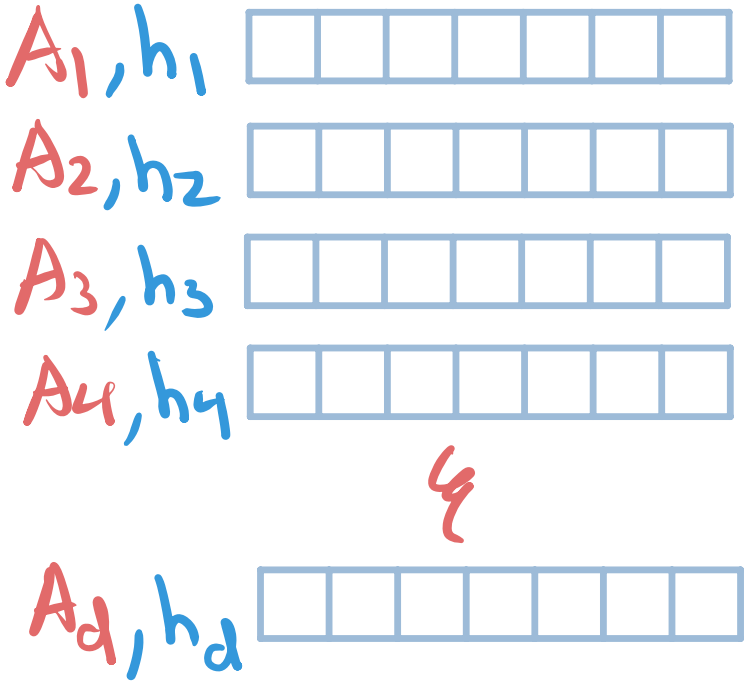
$P[\text{first toss is tails}] = 1/2$

$P[\text{first 2 tosses are tails}] = 1/4$

$\vdots$

$P[\text{first 100 tosses are tails}] = 1/2^{100}$

make d copies of hashed counters



total space
$O(wd)$

estimate f_i w/ $\min_{j=1,\dots,d} A_j[h_j(i)]$

bad estimate iff ALL hashed counters are bad

prob $1/2^d$

set $d = 10\log(n)$

each element is "bad" w/ prob.

$$\leq \frac{1}{2^d} = \frac{1}{n^{10}}$$

goal:

ensure <u>all</u> elements are correct

union bound:

$$P[\text{some } e \text{ bad}] \leq \sum_e Pr[e \text{ bad}] \leq n \cdot \frac{1}{n^{10}} = \frac{1}{n^9}$$

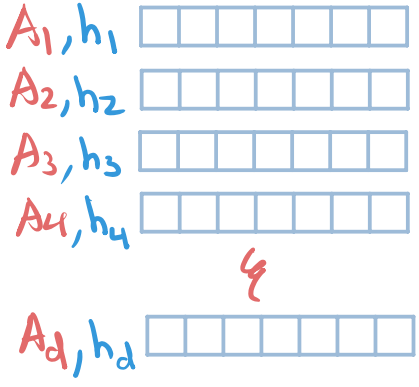
Heavy Hitters In Streams

- Elements one by one online (can repeat)

  a, b, c, b, a, d, a, e, a, b, a, $\S$

- 5% Heavy Hitters: elements that appear $\geq 5\%$ of the time

  (more generally 5% → parameter $\varepsilon > 0$)



estimate $\S_i$

w/ $\min_{j=1,\ldots,d} A_j[h_j(i)]$

bad estimate iff ALL hashed counters are bad

prob $1/2^d$

$\Rightarrow$ all elements estimated accurately w/ prob. $\geq 1 - \frac{1}{n^9}$

Space:

$$O(wd) = O(\log(n)/\varepsilon)$$

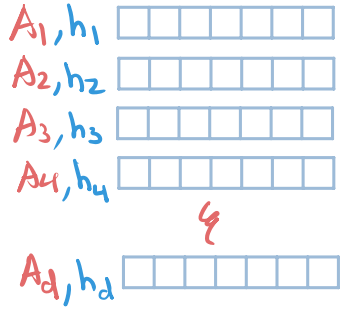
Heavy Hitters In Streams

- Elements one by one online (can repeat)

a, b, c, b, a, d, a, e, a, b, a, $\S$

- 5% Heavy Hitters: elements that appear $\geq$ 5% of the time

(more generally 5% $\rightarrow$ parameter $\varepsilon > 0$)



estimate f_i

w/ $\min_{j=1,\ldots,d} A_j[h_j(i)]$

bad estimate iff ALL hashed counters are bad (prob $1/2^d$)

set $d = 10 \log(n)$

each element is "bad" w/ prob. $\leq 2^d = \frac{1}{n^{10}}$

$$\text{Prob[any element bad]} \leq \sum_{i=1}^{n} \Pr[\text{element } i \text{ bad}]$$

$$\leq n \cdot \frac{1}{n^{10}} = \frac{1}{n^9}$$

$\Rightarrow$ all elements estimated accurately w/ prob. $\geq 1 - \frac{1}{n^9}$

$\Rightarrow$ all elements estimated accurately w/ prob. $\geq 1 - \frac{1}{n^9}$

Space: $O(wd) = O(\log(n)/\varepsilon)$

All put together

Estimate <u>all</u> rel. frequencies up to $\pm\varepsilon$ error w/

- $O(\log n/\varepsilon)$ space
- $\geq 1 - \frac{1}{n^{O(1)}}$ prob. of success

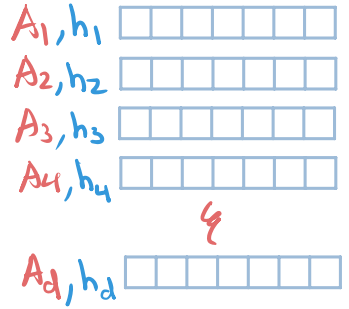
Heavy Hitters In Streams

- Elements one by one online (can repeat)

a, b, c, b, a, d, a, e, a, b, a, $

- 5% Heavy Hitters: elements that appear $\geq$ 5% of the time

(more generally 5% $\rightarrow$ parameter $\varepsilon > 0$)



estimate f_i

w/ $\min_{j=1,\ldots,d} A_j[h_j(i)]$

bad estimate iff ALL hashed counters are bad (prob $1/2^d$)

set $d = 10\log(n)$

each element is "bad" w/ prob. $\leq 2^d = \frac{1}{n^{10}}$

$\text{Prob}[\text{any element bad}] \leq \sum_{i=1}^{n} \Pr[\text{element } i \text{ bad}]$

$\leq n \cdot \frac{1}{n^{10}} = \frac{1}{n^9}$

$\Rightarrow$ all elements estimated accurately w/ prob. $\geq 1 - \frac{1}{n^9}$

Today:

② Randomized algorithms ("average" over random bits)

Probability	Algorithms
Hash functions,	Hash tables,
Markov's Inequality	Frequency Estimation, Heavy Hitters (in streams)