Sets, Dictionaries & Hash Tables

CS16: Introduction to Data Structures & Algorithms

Spring 2019

Q: how would you build a (basic) search engine?

What's so Hard about Search Engines?

"The **Google** Search **index** contains hundreds of billions of webpages and is well over 100,000,000 gigabytes in **size**."

How Google Search Works | Crawling & Indexing
https://www.google.com › search › crawl...

Google

Brown University

All News Images Maps Videos More Settings Tools

About 3,730,000,000 results (1.05 seconds)

Brown University
https://www.brown.edu/
Brown University, founded in 1764, is a member of the Ivy League and recognized for the quality of its teaching, research, and unique curriculum. Providence ...

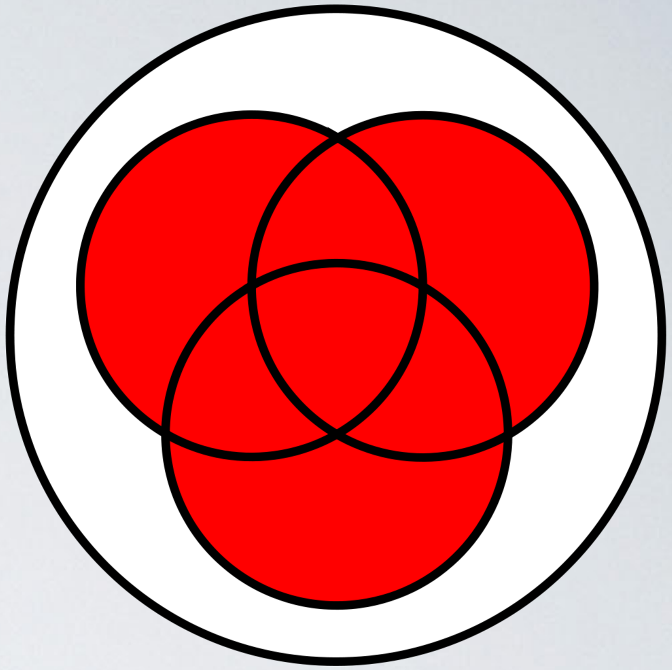
Search Through Each Page?

- Assume Google indexes `200,000,000,000` pages
- If we could scan `1` page in `1` microsecond
  - one search would take `55` hours
- How do we improve search time
  - when we have to look through billions of documents?

Outline

- Sets
- Dictionaries
- Hash Tables
- Ex: Search engine

Sets

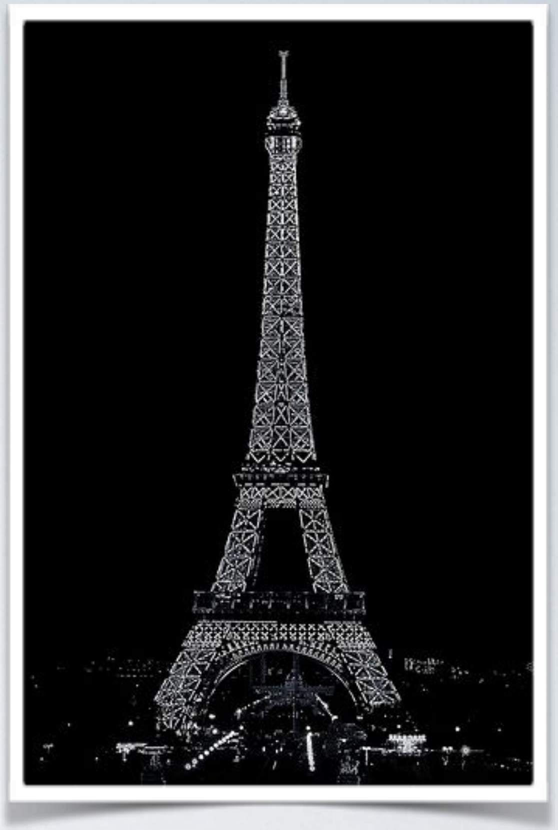


- Collection of elements that are
  - distinct
  - unordered (unlike lists or arrays)

Set ADT

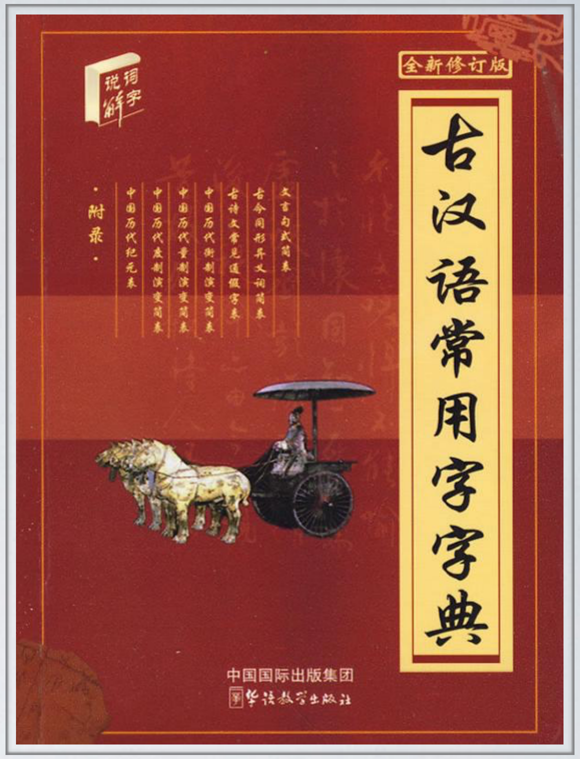
- **add**(object):
  - adds object to set if not there
- **remove**(object):
  - removes object from set if there
- boolean **contains**(object):
  - checks if object is in set
- *int* **size**( ):
  - returns number objects in set
- *boolean* **isEmpty**( ):
  - returns TRUE if set is empty; FALSE otherwise
- list **enumerate**( ):
  - returns list of objects in set (in arbitrary order)

Set Data Structure



- How can we implement a Set?
- Expandable array
  - add (to end): `O(1)`
  - contains (scan): `O(n)`
  - remove (find & compress): `O(n)`
- Can we do better?

Dictionary



- Collection of key/value pairs
  - distinct keys
  - unordered
- Supports value lookup by key
- AKA a **map**
  - maps keys to values
- ex: name → address; word → definition

Dictionary ADT

- **add**(key, value):
  - adds key/value pair to dict.
- object **get**(key):
  - returns value mapped to key
- **remove**(key):
  - removes key/value pair
- *int* **size**( ):
  - returns number key/value pairs
- *boolean* **isEmpty**( ):
  - returns TRUE if dict. is empty; FALSE otherwise

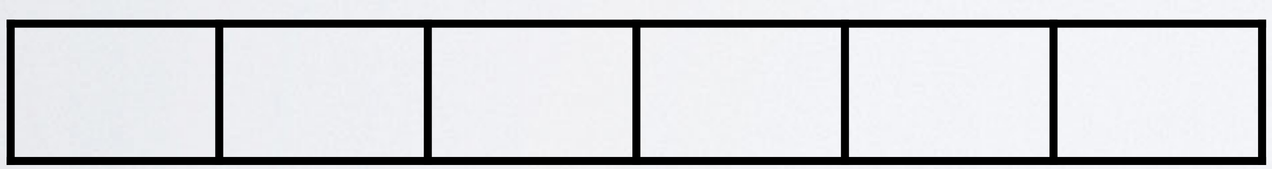
Q: how can we implement a dictionary?

Array-based Dictionary

- Use an expandable array `A`
- **add**(`k,v`):
  - store `(k,v)` at first empty cell of `A`
  - takes `O(1)`
- **get**(`k`):
  - scan `A` to find value with key `key=k`
  - takes `O(n)`
- **remove**(`k`):
  - scan `A` to find pair with `key=k` & remove
  - takes `O(n)`
- *Q: Can we do better?*

Yes! with a Hash Table

- What is a hash table?
  - a Dictionary data structure composed of
    - an array **`A`** and
    - a “hash” function **`h: X`**$\longrightarrow$**`Y`**



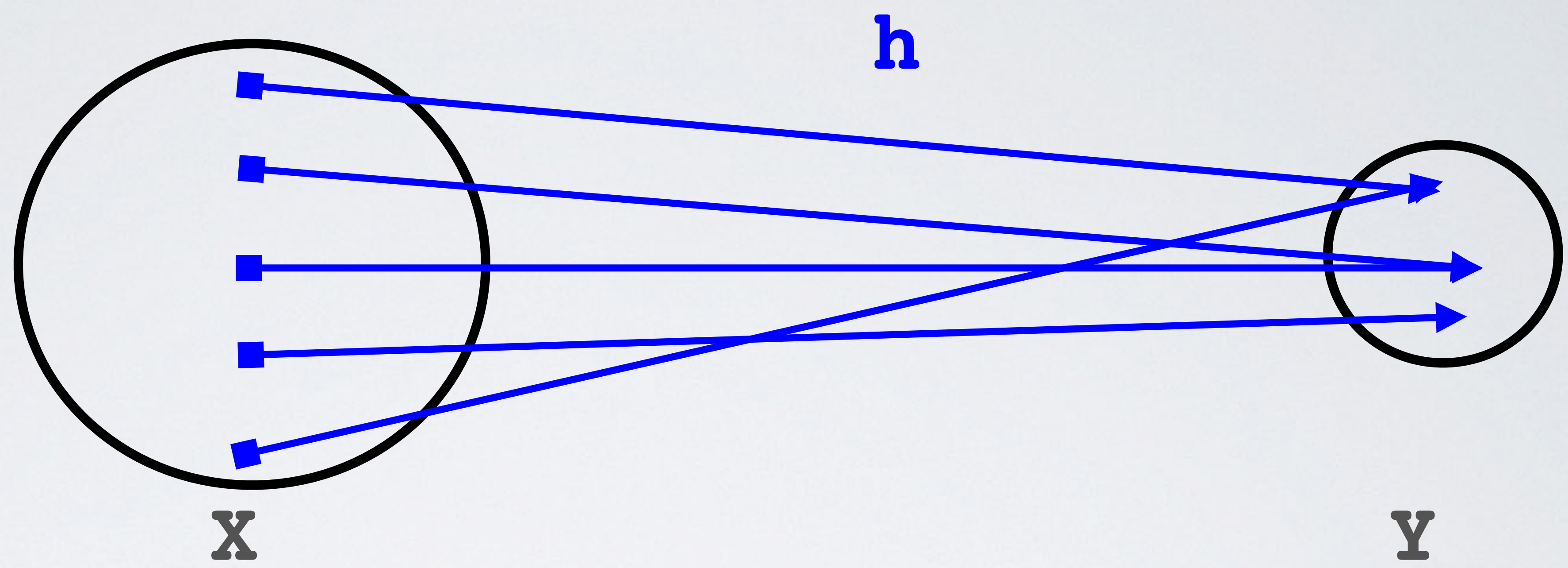
& **`h(x)`**

Yes! with a Hash Table

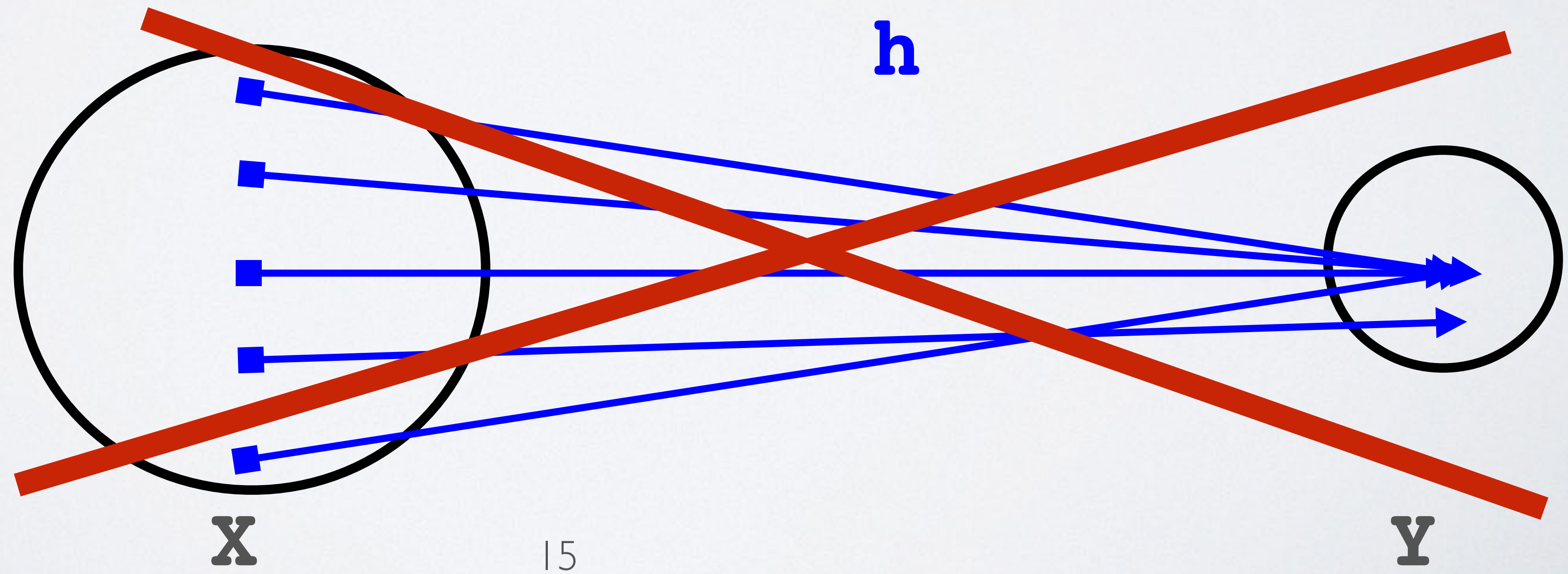
- What is a "hash" function?
  - a function **`h: X⟶Y`** that is "shrinking", i.e., that maps elements from an input space **`X`** to a **smaller** output space **`Y`**
  - such that the elements of **`X`** are "well-spread" over **`Y`**

Yes! with a Hash Table

- Shrinking



- Well-spread over **`Y`**

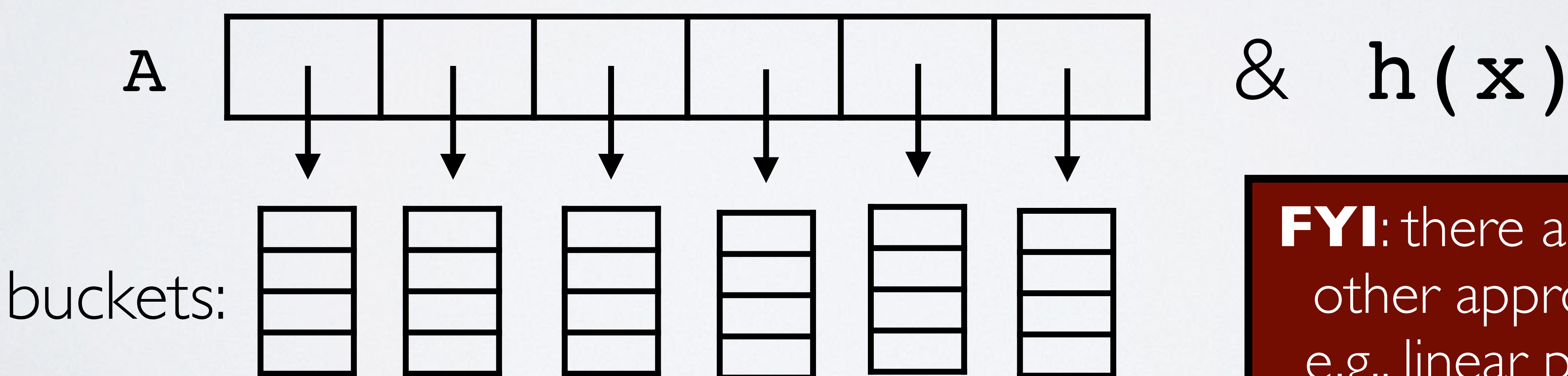


Building a Dictionary w/ a Hash Table

- Choose a hash function `h:X⟶Y` with
  - `X` = universe of keys and `Y` = indices of array
- **add**(`k,v`): set `A[h(k)]=v` — `O(1)`
- **get**(`k`): return `v=A[h(k)]` — `O(1)`
- **remove**(`k`): delete `A[h(k)]` — `O(1)`
- *Q: What's the problem with this?*
  - since $|Y|<|X|$ some keys in `X` will be hashed to same location!
  - so some values will be overwritten
  - this is called a **collision**

Overcoming Collisions

- Hash Table with Chaining
  - store *multiple* values at each array location
  - each array cell will store a "*bucket*" of pairs
    - can implement bucket as a list or expandable array or …



FYI: there are many other approaches e.g., linear probing, quadratic probing, cuckoo hashing,…

Hash Table

```
table: array
h: hash function
```

```
function add(k, v):
  index = h(k)
  table[index].append(k, v)
```

O(1) if hash is O(1)

```
function get(k):
  index = h(k)
  for (key, val) in table[index]:
    if key = k:
      return val
  error("key not found")
```

depends on bucket size

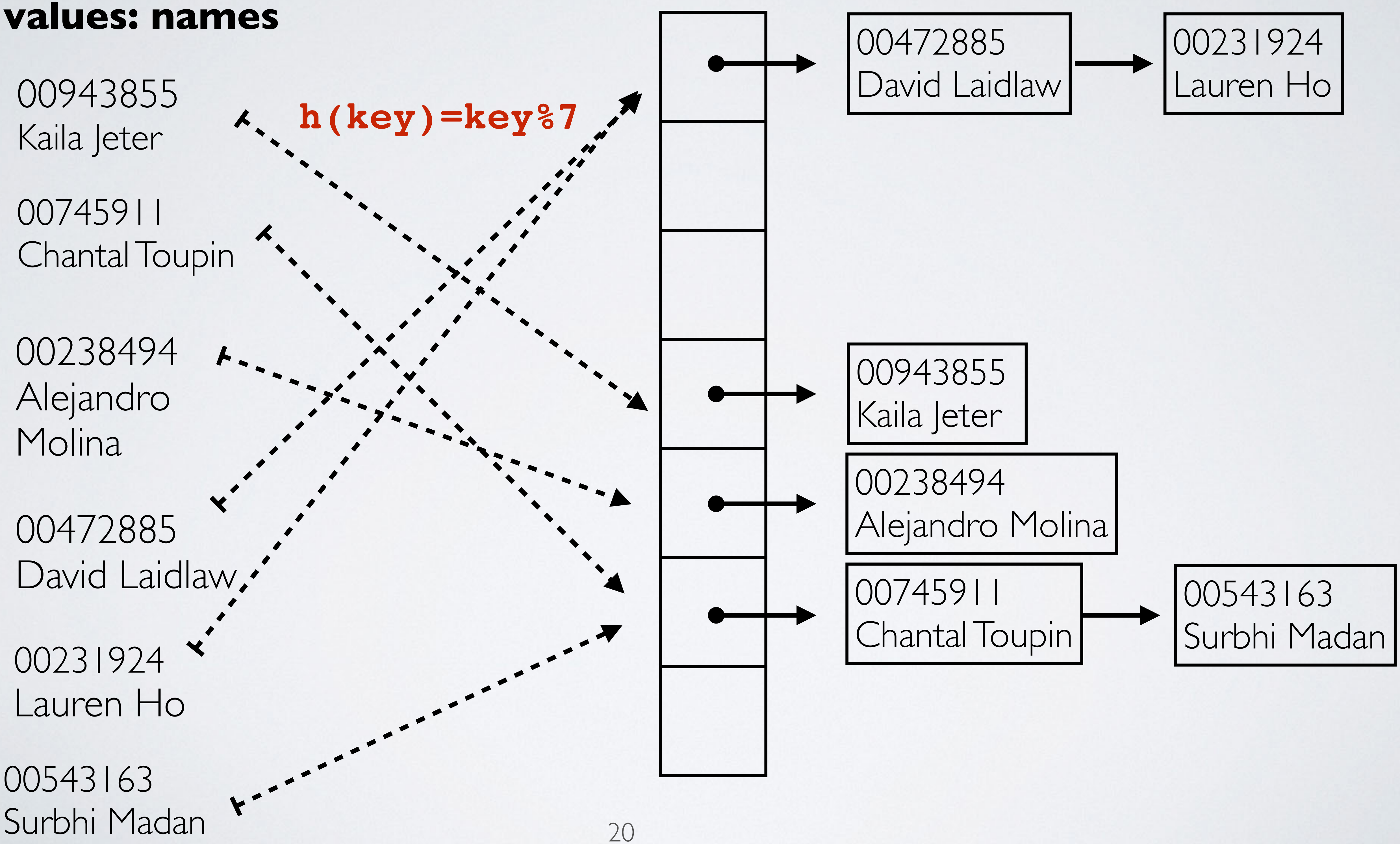
Hash Table

- Let's do an example!
  - build a dictionary that maps Banner IDs to Names
  - Let's use a Hash Table with Chaining!
- We'll use the following hash function
  - `h(banner_id)=banner_id % 7`
  -

Hash Table — Add

keys: banner IDs
values: names

Array of buckets w/ key/value pairs



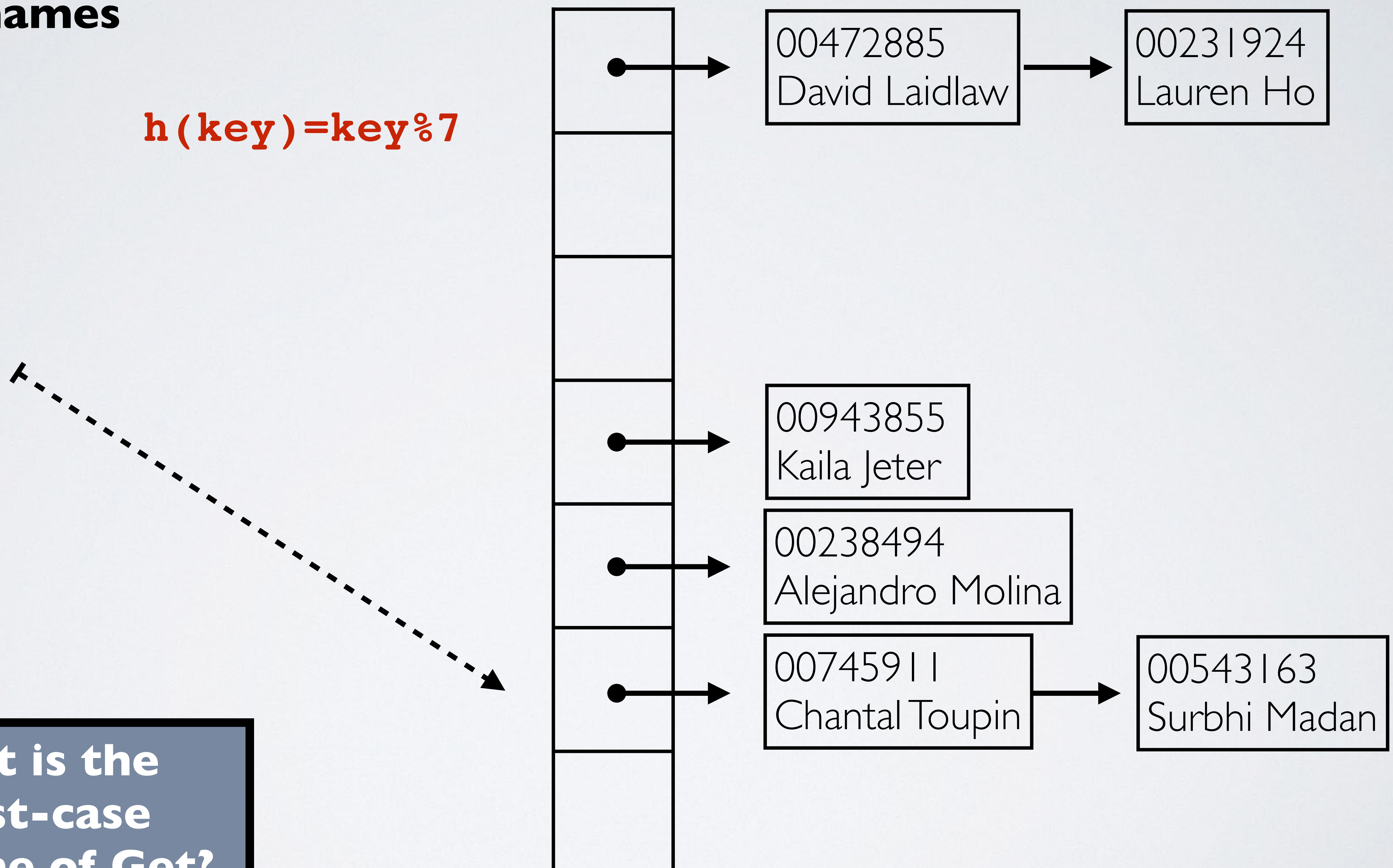
Hash Table — Get

keys: banner IDs
values: names

Array of buckets w/ key/value pairs

`h(key)=key%7`

00543163



What is the worst-case run time of Get?

Hash Table

- What is the worst-case runtime of Get?
  - ≈ size of largest bucket
- What is the size of largest bucket?
  - assume we have n students and a table of size m
  - **if** h “spreads” keys roughly evenly then
    - each bucket has size ≈ n/m
    - ex: if $n=150$ and $m=7$ each buckets has size ≈ $150/7 = 21$
- What is the size of largest bucket *asymptotically*?
  - assume m is a constant (i.e., it does not grow as a function of n)
    - each bucket has size ≈ $n/m = n/c = O(n)$ ☹

Q: Can we do better than `O(n)`?

Beating `O(n)` — Idea #1

- **Idea:** use larger table
- Banner IDs have `8` digits so max ID is `99,999,999`
- Use table of size `m=100,000,000`
  - w/ hash function `h(key)=key`
- Are there any collisions in this case?
  - no collisions because every pair gets its own cell
  - What is run time of Get?
    - `O(1)` since we don't need to scan buckets
- What is the problem with this approach?
  - what if we only store `150` students? we're *wasting* `99,999,850` cells

Beating `O(n)` — Idea #2

- **Idea**: use table of size `m=n`
- If we know we will only store `n=150` students
  - use table of size `m=150`
    - w/ hash function `h(key) = key % 150`
    - no waste of space!
  - **if** `h` "spreads" keys roughly evenly then each bucket has size
    - `≈ n/m = 150/150 = 1 = O(1)`

Banner ID Hashing

Form groups of `10`

Activity #1

5 min

Banner ID Hashing

Activity #1

5 min

Banner ID Hashing

Activity #1

4 min

Banner ID Hashing

Activity #1

3 min

Banner ID Hashing

Activity #1

2 *min*

Banner ID Hashing

Activity #1

1 min

Banner ID Hashing

Activity #1

0 min

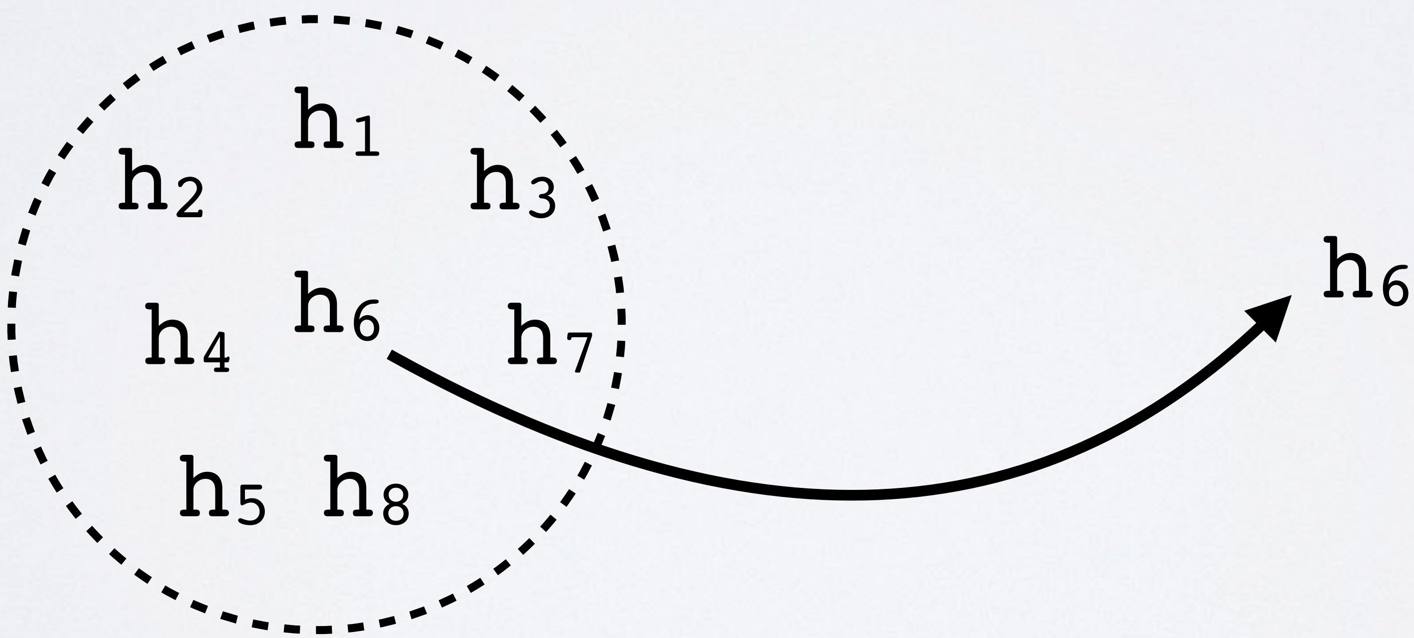
Beating `O(n)` — Idea #2

- Idea #2 relied on an assumption:
  - ***if*** *`h` "spreads" keys roughly evenly* then each bucket has size
    - `≈ n/m = 150/150 = 1 = O(1)`
- *Will* **`h`** spread banner IDs evenly?
  - it depends on the banner IDs…
  - if banner IDs are chosen randomly then Yes
  - But what if next year all banner IDs are multiples of `150`?
  - Then *all* banner IDs will map to `0`!
  - So there will be a bucket with size `150` (all others will have size 0)
  - so *worst-case* runtime of Get will be `O(n)`

Since keys are not necessarily random, we make the hash function random

Universal Hash Functions

- Special "families" of hash functions
  - `UHF = {h₁,h₂,…,h_q}`
  - designed so that if we pick a function from the family at random and use it on a set of keys, then the function will "spread" the keys roughly evenly (with high probability)



Example of Universal Hash Functions

- Setup to store $\mathtt{n}$ key/value pairs
  - choose *prime* $\mathtt{p}$ larger than $\mathtt{n}$
  - choose 4 numbers $\mathtt{a}_1$, $\mathtt{a}_2$, $\mathtt{a}_3$, $\mathtt{a}_4$ *at random* between 0 and $\mathtt{p-1}$
- Hashing a key $\mathtt{k}$
  - break $\mathtt{k}$ into 4 parts
    - $\mathtt{k}_1$, $\mathtt{k}_2$, $\mathtt{k}_3$, $\mathtt{k}_4$
  - output

$$h(k) = \sum_{i=1}^{4} a_i \cdot k_i \mod p$$

- Setup to store 150 students
  - choose $\mathtt{p=151}$
  - choose $\mathtt{a}_1=12$, $\mathtt{a}_2=43$, $\mathtt{a}_3=105$, $\mathtt{a}_4=83$
- Hashing a key $\mathtt{k=00238918}$
  - break $\mathtt{k}$ into $\mathtt{k}_1=00$, $\mathtt{k}_2=23$, $\mathtt{k}_3=89$, $\mathtt{k}_4=18$
  - output

$$h(00238918) = 50$$

Hash Table with UHFs

- Hash table + universal hash functions
  - *Worst-case* runtime of Get is `O(n)`
  - But UHFs guarantee that worst-case happens *very rarely*
  - We should expect to see a Get runtime that is `O(1)`
- What do we mean by expect?
  - remember that with UHFs we picked one function from family at random
    - in example we picked the values (a_1,a_2,a_3,a_4) at random
  - for some functions in family, keys will be well-spread & for others keys may be clustered
  - but if we were to compute the runtime of Hash Table with `h` a million times, where each time we sample a hash function at random from the family…
  - …then the average of those runtimes would be `O(1)`
  - This is called "expected running time"

Why does Universal Hashing Work?

- Why does it result in expected `O(1)` Gets?
  - see Chapter 1.5.2 in Dasgupta et al.

Proof of Universal Hashing

Inverses

- What is the inverse of a fraction `x/y`?
  - `y/x` because `(x/y)(y/x)=1`
  - inverse is whatever we need to multiply it by to get `1`
- What is the inverse of an int x (not `1`)?
  - `1/x` because `(x)(1/x)=1`
- What is the "integer" inverse of an int `x` (not `1`)
  - there is none…
  - you can't multiply an int w/ another int to get `1` (unless `1`)

Modular Arithmetic

- If working modulo some number
  - Integers can have integer inverses!
- ex: let's work `mod 7`
  - inverse of `2 mod 7` is `4` because `2x4 mod 7 = 1`
  - inverse of `5 mod 7` is `3` because `5x3 mod 7 = 1`
- Is this always true?
  - ex: does `2` have an inverse `mod 4`?
  - `2x0 mod 4 = 0`; `2x1 mod 4 = 2`
    `2x2 mod 4 = 0`; `2x3 mod 4 = 2`
  - No!
- But it is true when we work modulo a prime number
  - mod a prime, every number except `0` has a unique inverse

Analysis

- Prime p is the size of array
- x_1, x_2, x_3, x_4 are a banner ID in chunks
- y_1, y_2, y_3, y_4 are another banner ID in chunks
- If IDs are different, at least 1 of the chunks are diff
- Let's assume (wlog) it is the last one so
  - $x_4 \; != \; y_4$
- What is the probability that
  - $h(x_1,x_2,x_3,x_4) = h(y_1,y_2,y_3,y_4)$

Analysis

- What is the probability that
  - $\mathtt{h(x_1,x_2,x_3,x_4) = h(y_1,y_2,y_3,y_4)}$
- Step #1:
  - find equivalent formulation of event
  - that makes the randomness explicit
  - what is the randomness here?
- Step #2:
  - what is probability of equivalent formulation?

Step 1: Equivalent Formulation

$$h(x_1, x_2, x_3, x_4) = h(y_1, y_2, y_3, y_4)$$

by definition

$$a_1 x_1 + \cdots + a_4 \boxed{x_4} \equiv a_1 y_1 + \cdots + a_4 \boxed{y_4} \pmod{p}$$

different

move things

$$a_4 x_4 - a_4 y_4 \equiv \boxed{(a_1 y_1 + a_2 y_2 + a_3 y_3) - (a_1 x_1 + a_2 x_2 + a_3 x_3)} \pmod{p}$$

just some number; let's call it c

$$a_4 \cdot (x_4 - y_4) \equiv c \pmod{p}$$

$$a_4 \equiv c \cdot (x_4 - y_4)^{-1} \pmod{p}$$

Step 2: Probability of Equiv. Formulation

- So hashes are equal when

$$a_4 \equiv c \cdot (x_4 - y_4)^{-1} \pmod{p}$$

- But
  - $\mathbf{x_4}$ and $\mathbf{y_4}$ are different so $x_4 - y_4 \neq 0$
  - and $\mathbf{p}$ is prime
  - so $\mathbf{(x_4-y_4)}$ has unique inverse mod $\mathbf{p}$
- So $\mathbf{c(x_4-y_4)^{-1}}$ can only take on one value
  - therefore $\mathbf{a_4}$ can only take on one value
- What is the probability $\mathbf{a_4}$ takes on that value?
  - $\mathbf{a_4}$ is randomly chosen from $\mathbf{p}$ possible values so probability is $\mathbf{1/p}$

Putting it all Together

- Prob. that some ID will collide w/ another ID
  - `1/p = 1/151`
- For some ID,
  - expected # of collisions w/ all other IDs is
  - `149/151 = 0.986…`
- Expected size of an ID's bucket is
  - `1+0.986… = 1.986… = O(1)`

End of Universal Hashing Proof

Summary

- Array-based Dictionaries
  - Add is *worst-case* `O(n)`
  - Get is *worst-case* `O(n)`
- Hash Table-based Dictionaries (with UHFs)
  - Add is
    - *worst-case* `O(n)` but *expected* `O(1)`
  - Get is
    - *worst-case* `O(n)` but *expected* `O(1)` time

Q: what can we build from dictionaries?

Sets from Hash Tables

- We can implement sets with a hash table
- Sometimes called a Hash Set

```
function add(object):
  index = h(object)
  table[index].append(object)
```

```
function contains(object):
  index = h(object)
  for elt in table[index]:
    if elt == object:
      return true
  return false
```

A (Basic) Search Engine

- Build a dictionary that maps keywords to URLs
  - takes `O(n)` time
- Query dictionary on keyword to retrieve URLs
  - takes expected `O(1)`
- In context of search engines
  - the dictionary is often called an Index

A (Basic) Search Engine

- For a each keyword **`word`** w/ a list of relevant URLs $\texttt{url}_1,\ldots,\texttt{url}_m$
  - store the pairs $(\texttt{word}|1,\texttt{url}_1),\ldots,(\texttt{word}|\texttt{m},\texttt{url}_m)$ in a dict **`Index`**
  - where "|" is string concatenation
  - Store the pair **`(word,m)`** in an auxiliary dictionary **`Counts`**
- To search for a keyword **`Brown`**
  - retrieve the count for **`Brown`** by querying **`Count.get(Brown)`**
  - to recover URLs, query **`Index`** on keys **`Brown|1,…,Brown|m`**
    - **`Index.get(word|1),…,Index.get(word|m)`**

Build Index

```
function build_index(page_list):
    index = dict()
    counts = dict()
    for page in page_list:
       for word in page:
           try:
               count = counts.get(word)
           except KeyError:
               counts.put(word,0)
               count = counts.get(word)
           counts.put(word, counts[word] + 1)
           key = word + str(counts.get(word))
           index.put(key, page.url)
    return index
```

- build_index is `O(nm)` time
  - where **n** is number of pages and **m** is maximum number of words per page

Search Index

```
def search_index(index, word):
    output_list = list()
    count = 1
    while True:
        try:
            url = index.get(word + str(count))
            count = count + 1
        except KeyError:
            break
        output_list.append(url)
    return output_list
```

- If dictionary is implemented with hash table
  - search_index is expected `O(1)` time
  - fast no matter how many pages and words

A (Basic) Search Engine

- What's missing from our "search engine"?
  - No ranking
  - But we'll learn about that later in the course

Dictionary vs. Hash Table

- A dictionary (or map) is an abstract data type
  - can be implemented using many ≠ data structures
- A hash table is a dictionary data structure
  - one particular way to implement a dictionary

HashMap vs. HashSet

- Java HashMaps and HashSets
- HashMap
  - Hash table implementation of a dictionary
- HashSet
  - Hash table implementation of a set