Expandable Stack

```
Stack( ):
  data = array of size 20
  count = 0
  capacity = 20
```

```
function push(object):
  data[count] = object
  count++
  if count == capacity
    new_capacity = capacity + e /* incremental */
                 = capacity * 2 /* doubling */
    new_data = array of size new_capacity
    for i = 0 to capacity - 1
      new_data[i] = data[i]
    capacity = new_capacity
    data = new_data
```

Run time depends on count which depends on # of *previous* pushes

Amortized Analysis of Incremental

- Summary
  - Total cost of **n** pushes: $S(n) = O(n^2)$
  - Amortized cost of **n** pushes: $S(n)/n = O(n)$

Amortized Analysis of Double

Amortized Analysis of Doubling

- Doubling stack with initial capacity **c=5**?

cost of pushes

$$\frac{S(n)}{n} = \frac{S(5)}{5} = \frac{5+5}{5} = 2$$ **cost of exp**

$$\frac{S(n)}{n} = \frac{S(10)}{10} = \frac{10+5+10}{10} = 2.5$$ **cost of exp #2**

$$\frac{S(n)}{n} = \frac{S(20)}{20} = \frac{20+5+10+20}{20} = 2.75$$ **cost of exp #3**

Amortized Analysis of Doubling

cost of n pushes **cost of last exp** **cost of second to last exp**

$$S(n) = n + n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^{k-1}}$$ **cost of exp #1**

$$= n + n \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{k-1}}\right)$$

$$< n + n \cdot 2$$ **using:** $$\boxed{\lim_{k\to\infty} \sum_{i=0}^{k} \frac{1}{2^i} = 2}$$

$$= 3n$$

Assume:
c=2
n=2^k

$$\boxed{\frac{S(n)}{n} = O(1)}$$

Amortized Analysis

- Summary for Incremental
  - Total cost of $\mathtt{n}$ pushes: $\mathtt{S(n) = O(n^2)}$
  - Amortized cost of $\mathtt{n}$ pushes: $\mathtt{S(n)/n = O(n)}$
- Summary for Doubling
  - Total cost of $\mathtt{n}$ pushes: $\mathtt{S(n) = O(n)}$
  - Amortized cost of n pushes: $\mathtt{S(n)/n = O(1)}$
- In practice: always use doubling

How do we feel about amortized analysis?

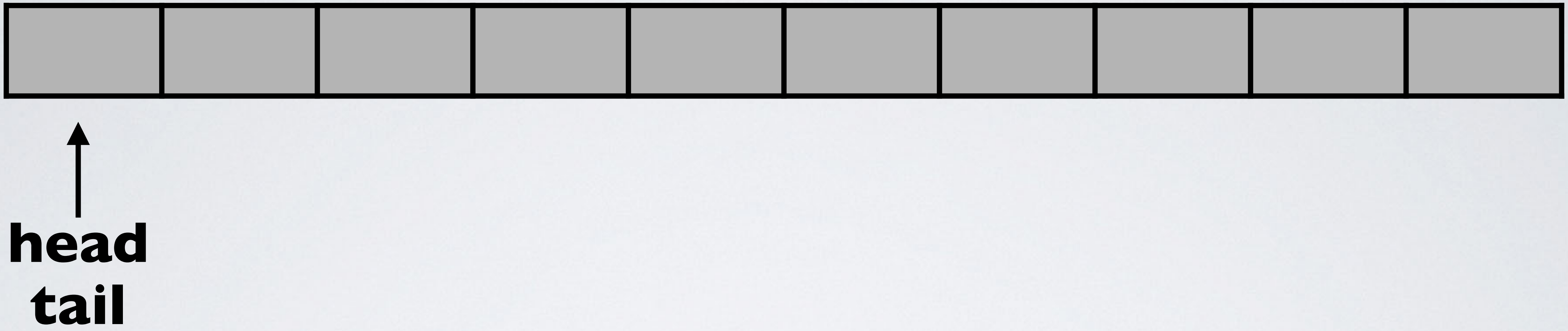
- Situations where worst case is most important?

Expandable Queue

Queue ADT

- **`enqueue`**`(object)`:
  - inserts object
- `object` **`dequeue( )`**
  - returns and removes first inserted object
- `int` **`size( )`**
  - returns number objects in queue
- `boolean` **`isEmpty( )`**
  - returns **`TRUE`** if empty; **`FALSE`** otherwise

Expandable Queue



- Can be implemented with expandable array
  - need to keep track of head and tail

Expandable Queue

- Can be implemented with expandable array
    - need to keep track of head and tail

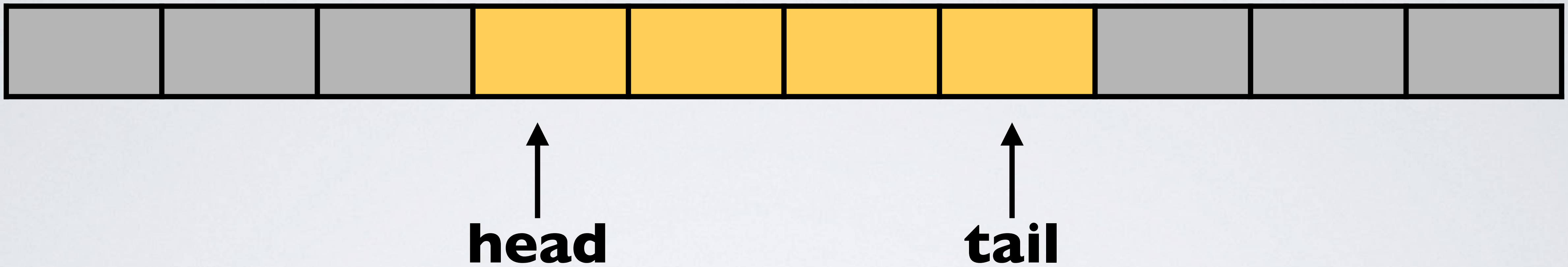
Expandable Queue

- Can be implemented with expandable array
  - need to keep track of head and tail

Expandable Queue

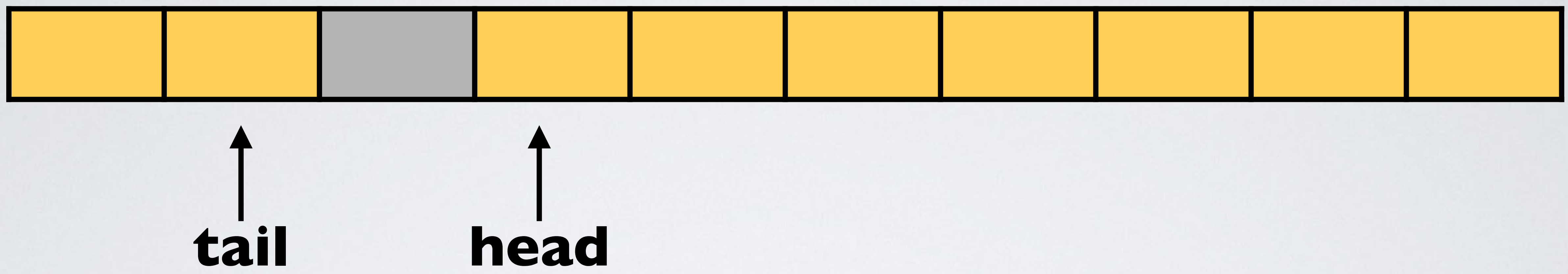
- Can be implemented with expandable array
  - need to keep track of head and tail

Expandable Queue



- Can be implemented with expandable array
  - need to keep track of head and tail
- What happens when tail reaches end?
  - Is the queue full?
- So when should we expand array?

Expandable Queue



- Wrap around until array is completely full
- When expanding re-order objects properly

Expandable Queue

```
function enqueue(object):
  if size == capacity
    double array and copy contents
    reset head and tail pointers
  data[tail] = object
  tail = (tail + 1) % capacity
  size++
```

$$\boxed{\frac{S(n)}{n} = O(1)}$$

```
function dequeue( ):
  if size == 0
    error("queue empty")
  element = data[head]
  head = (head + 1) % capacity
  size--
  return element
```

Sets, Dictionaries & Hash Tables

CS16: Introduction to Data Structures & Algorithms

Summer 2021

Arrays (Non-expandable)

“WY”	**“VT”**	**“AK”**	**“ND”**	**“SD”**	**“DE”**	**“MT”**	**“RI”**
0	1	2	3	4	5	6	7

Arrays (Non-expandable)

“WY”	**“VT”**	**“AK”**	**“ND”**	**“SD”**	**“DE”**	**“MT”**	**“RI”**
0	1	2	3	4	5	6	7
`1000`	`1001`	`1002`	`1003`	`1004`	`1005`	`1006`	`1007`

Arrays (Non-expandable)

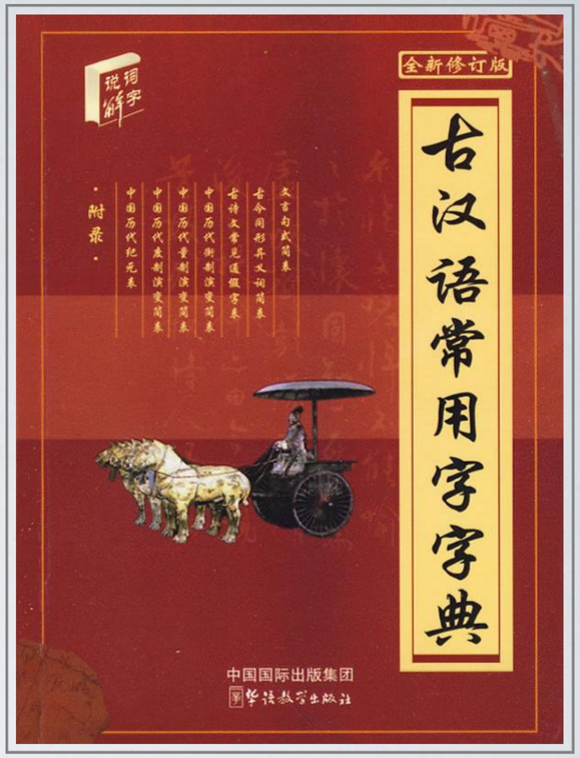
"WY"	**"VT"**	**"AK"**	**"ND"**	**"SD"**	**"DE"**	**"MT"**	**"RI"**
0	1	2	3	4	5	6	7
`1000`	`1001`	`1002`	`1003`	`1004`	`1005`	`1006`	`1007`

- Runtime to get the 5th element?
- Runtime to get the index of "RI"?

Arrays (expandable)

- Implemented like expandable stacks/queues
- Resize when full
- Accesses still O(1) on average

Dictionary



- Collection of key/value pairs
  - distinct and unordered keys
- Supports value lookup by key
- Also known as a *map*
  - “maps” keys to values
- examples
  - name → address
  - word → definition
  - postal abbreviation → state name

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

Array-based Dictionary

- Can we use an expandable array `A`?
- **`add`**(`k,v`):
  - store `(k,v)` in first empty cell of `A`
- **`get`**(`k`):
  - scan `A` to find value with key `key=k`
- **`remove`**(`k`):
  - scan `A` to find pair with `key=k` & remove
- Runtimes?

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...

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 ...

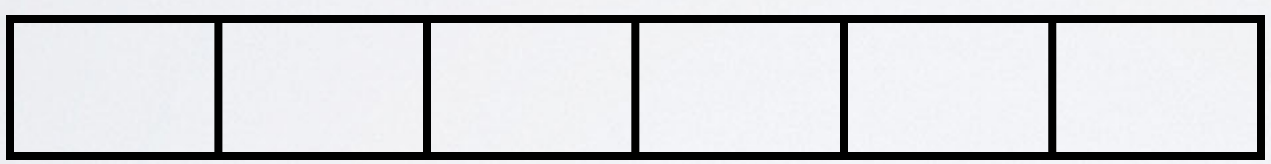
Search Through Each Page?

- Assume Google indexes **200** billion pages
- If we scan **1** page in **1** microsecond
  - each search would take **55** hours
- How can we improve search time?

Q: can we do better?

Yes! with a Hash Table

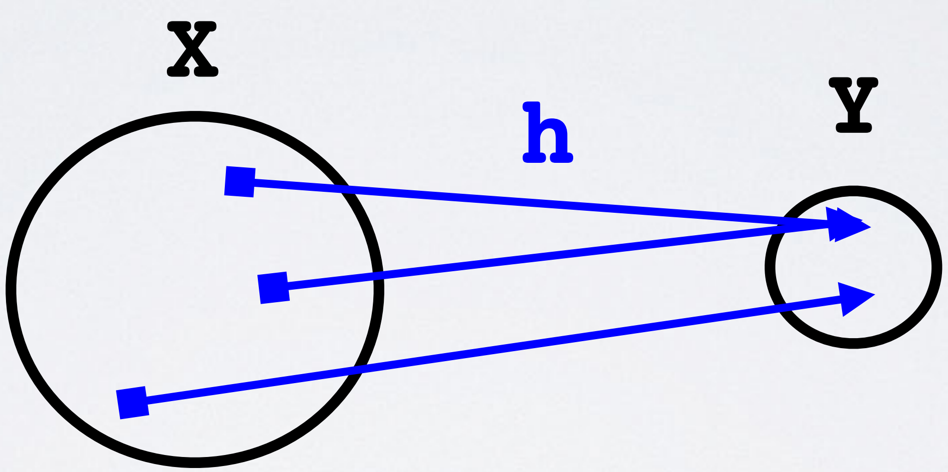
- Hash tables are composed of
  - an (expandable) array **A**
  - and a "hash" function $\mathbf{h\colon X \longrightarrow Y}$



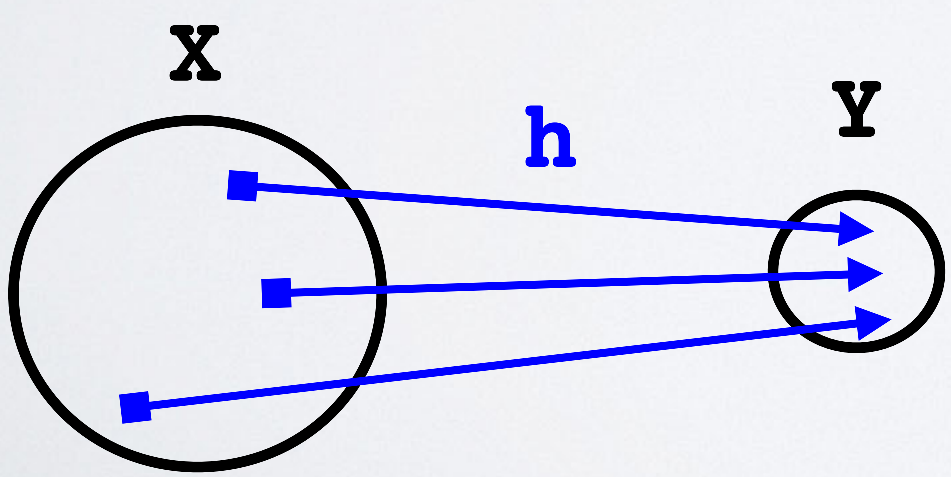
& **h(x)**

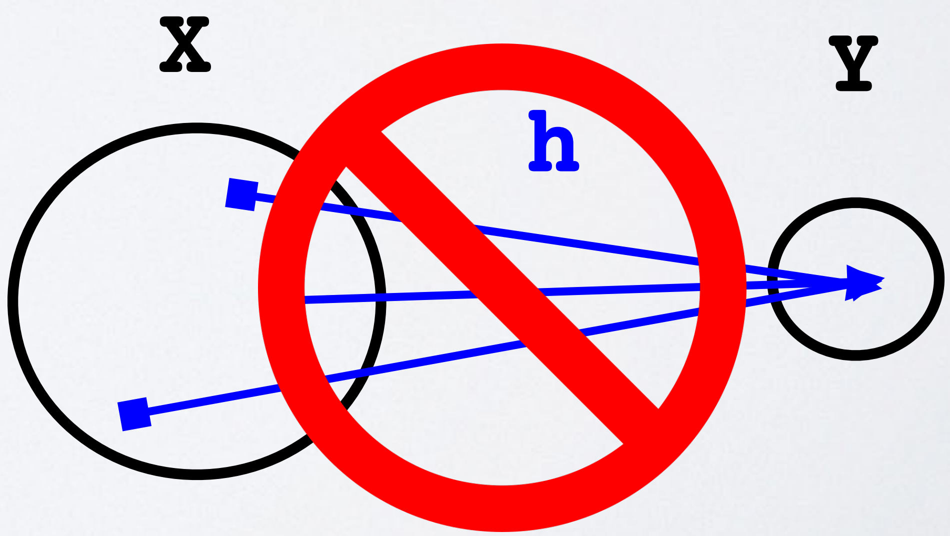
Yes! with a Hash Table

- A hash function is function **`h: X⟶Y`** that
  - *shrinks*: maps elements from a large input space to a *smaller* output space



  - *well spread*: **`h`** spreads elements of **`X`** over **`Y`**





Dictionary vs. Hash Table

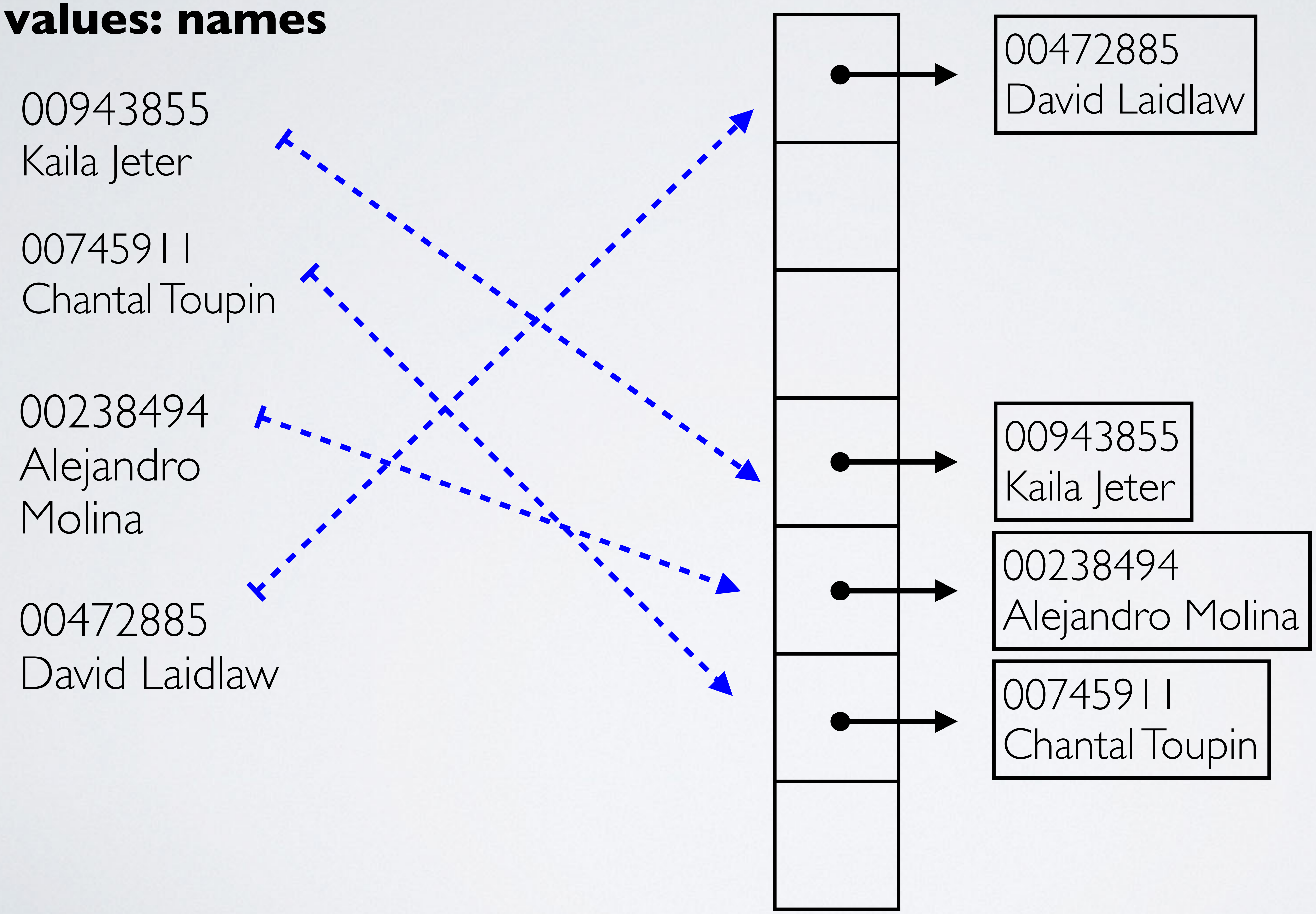
- A dictionary (or map) is an *abstract data type*
  - can be implemented using many different *data structures*
- A hash table is a dictionary *data structure*
  - one specific way to implement a dictionary

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`**
  - **`get`**(`k`)
    - return **`v=A[h(k)]`**
  - **`remove`**(`k`)
    - delete **`A[h(k)]`**
  - Runtimes?

Hash Table — Add

keys: banner IDs
values: names

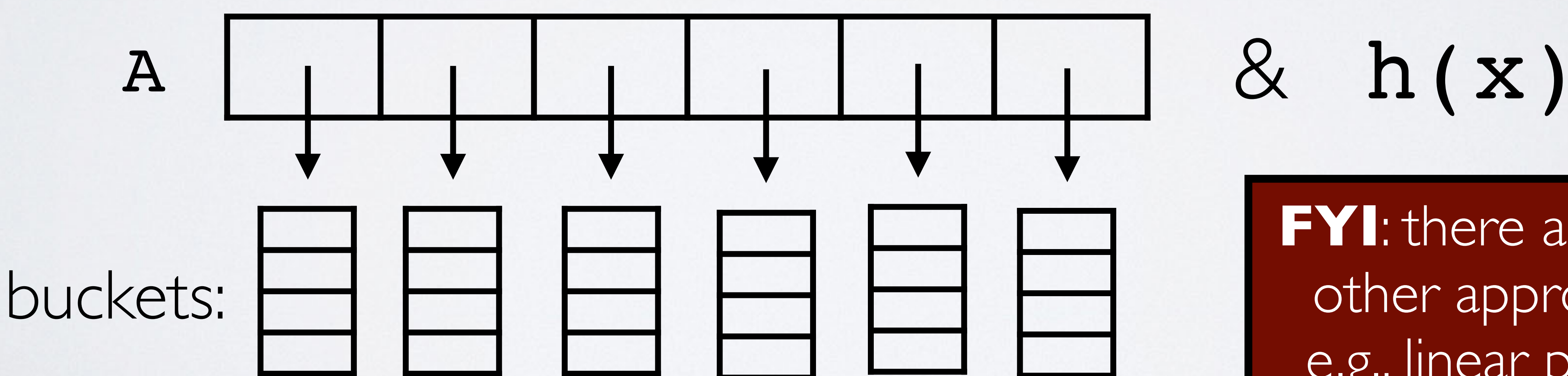


Building a Dictionary w/ a Hash Table

- ***Q:*** What is the problem with this?
  - Remember that $|\mathtt{Y}| < |\mathtt{X}|$
    - (here $|\mathtt{X}|$ denotes size of $\mathtt{X}$)
  - …so some keys in $\mathtt{X}$ will be hashed to the same location!
    - this is called the *pigeonhole principle*
    - there just isn't enough room in $\mathtt{Y}$ to fit all of $\mathtt{X}$
  - …therefore some values in array will be overwritten
    - this is called a *collision*

Overcoming Collisions

- Hash Table with *Chaining*
  - store *multiple* values at each array location
  - each array cell stores 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 computing hash function is O(1)

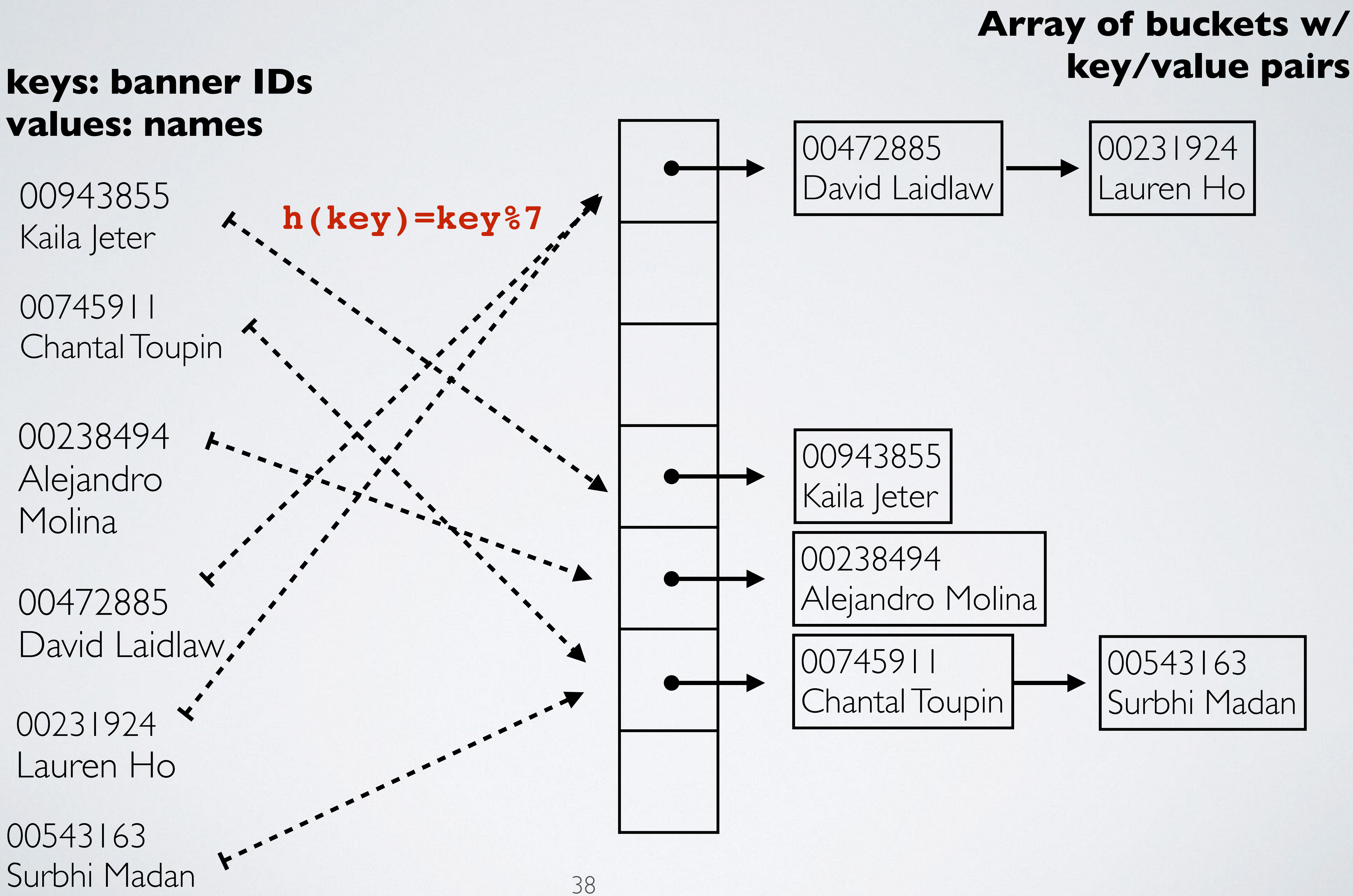
```
function get(k):
  index = h(k)
  for (key, val) in table[index]:
    if key = k:
      return val
  error("key not found")
```

runtime depends on bucket size

Hash Table

- Let's do another example but with Chaining!
- We'll use the following hash function
  - `h(banner_id)=banner_id % 7`

Hash Table — Add



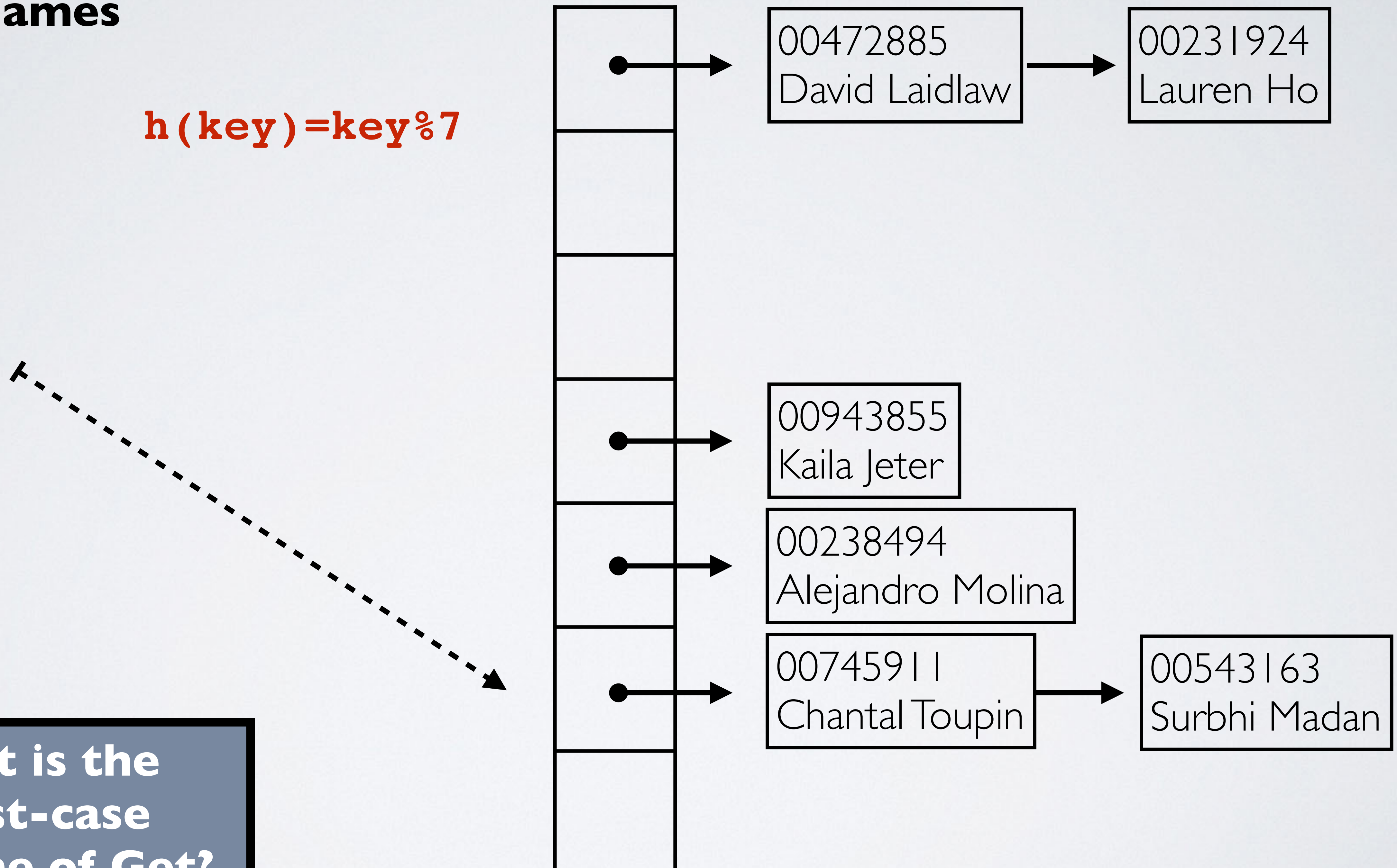
Hash Table — Get

Array of buckets w/ key/value pairs

keys: banner IDs
values: names

h(key)=key%7

00543163



What is the worst-case run time of Get?

Hash Table with Chaining

- 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`**
- But what is the size of the 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 large 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 a table of size equal to the number of students + "good" hash function
  - set the table size to `m=n`
  - use a hash function `h` that spreads keys well
- No wasted space since `n = m`
  - in other words, "table size" = "number of students"
- If `h` spreads keys roughly evenly then each bucket has size
  - `≈ n/m = n/n = 1 = O(1)`
- What hash function should we use?
  - Suppose `n = 150` (i.e., we want to insert `150` students)
  - should we use the hash function `h(key) = key % 150` ?

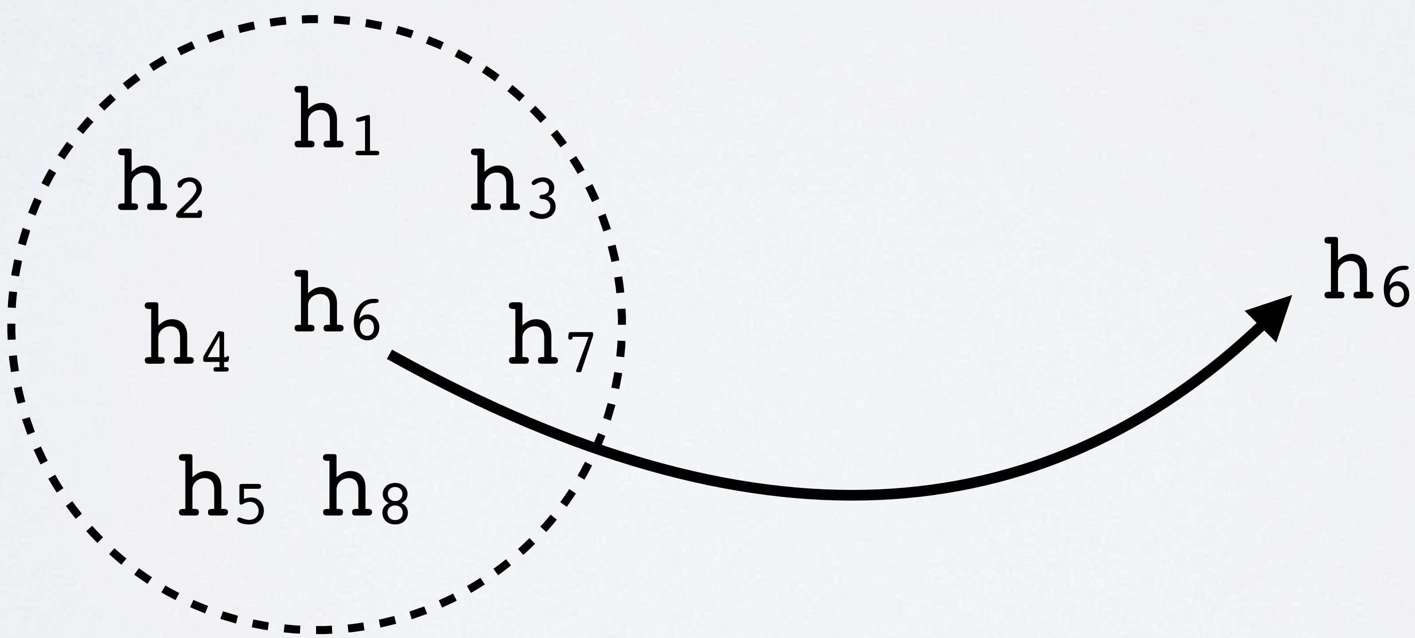
Beating `O(n)` — Idea #2

- Idea #2 relied on an assumption:
  - ***if*** *h* ***spreads*** *keys roughly evenly* then each bucket has size
    - `≈ n/m = n/n = 1 = O(1)`
- *Will* `h(ID)=ID%11` 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 `11`?
  - Then *all* banner IDs will map to `0`!
  - So there will be one bucket with all IDs
  - 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
  - $\mathtt{UHF} = \{h_1, h_2, \ldots, h_q\}$
  - designed so that if we pick a function from the family at random and use it on a set of keys, then it is *very likely* that the function will "spread" the keys (roughly) evenly



JOURNAL OF COMPUTER AND SYSTEM SCIENCES 18, 143–154 (1979)

Universal Classes of Hash Functions

J. LAWRENCE CARTER AND MARK N. WEGMAN

IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598

Received August 8, 1977; revised August 10, 1978

This paper gives an *input independent* average linear time algorithm for storage and retrieval on keys. The algorithm makes a random choice of hash function from a suitable class of hash functions. Given any sequence of inputs the expected time (averaging over all functions in the class) to store and retrieve elements is linear in the length of the sequence. The number of references to the data base required by the algorithm for any input is extremely close to the theoretical minimum for any possible hash function with randomly distributed inputs. We present three suitable classes of hash functions which also can be evaluated rapidly. The ability to analyze the cost of storage and retrieval without worrying about the distribution of the input allows as corollaries improvements on the bounds of several algorithms.

Example of Universal Hash Functions

- Setup to store `n` key/value pairs
  - choose *prime* `p` larger than `n`
  - choose `4` numbers a_1, a_2, a_3, a_4 *at random* between `0` and `p-1`
- Hashing a key `k`
  - break `k` into `4` parts
    - k_1, k_2, k_3, k_4
  - output

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

- Setup to store `150` students
  - choose `p=151`
  - choose $a_1=12$, $a_2=43$, $a_3=105$, $a_4=83$
- Hashing a key `k=00238918`
  - break `k` into $k_1=00$, $k_2=23$, $k_3=89$, $k_4=18$
  - output

$$h(00238918) = 50$$

Hash Table with UHFs

- Hash table w/ chaining using a universal hash function family
  - *Worst-case* runtime of Get is `O(n)` ☹
  - But UHFs guarantee that worst-case happens *very rarely*
  - We can "expect" that Get will have runtime `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
  - but for some functions in the 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"

Hash Table with UHFs

- Hash table w/ chaining using a universal hash function family
  - We can "expect" that Get will have runtime `O(1)`
- What do we mean by expect?
  - remember that with UHFs we picked *one* function from family at random
    - in the example we picked the values (a_1, a_2, a_3, a_4) at random
  - for some functions in the family, keys will be well-spread…
  - …while for others keys will be poorly spread, e.g., all mapped to same value
  - but if we were to compute the runtime of Hash Table with 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?

- See Chapter `1.5.2` in Dasgupta et al.
  - and/or read the proof in lecture slides
  - You do not need to know the 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)`

Q: what can we build from dictionaries?

A (Basic) Search Engine

- Build a dictionary that maps keywords to URLs
  - query dictionary on keyword to retrieve URLs
- 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 $\mathtt{url}_1,\ldots,\mathtt{url}_m$
  - store the pairs $(\mathtt{word}|1,\ \mathtt{url}_1),\ldots,(\mathtt{word}|m,\ \mathtt{url}_m)$ 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)`**

Idea from Cash et al., NDSS '14

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

Idea from Cash et al., NDSS '14

Search Index

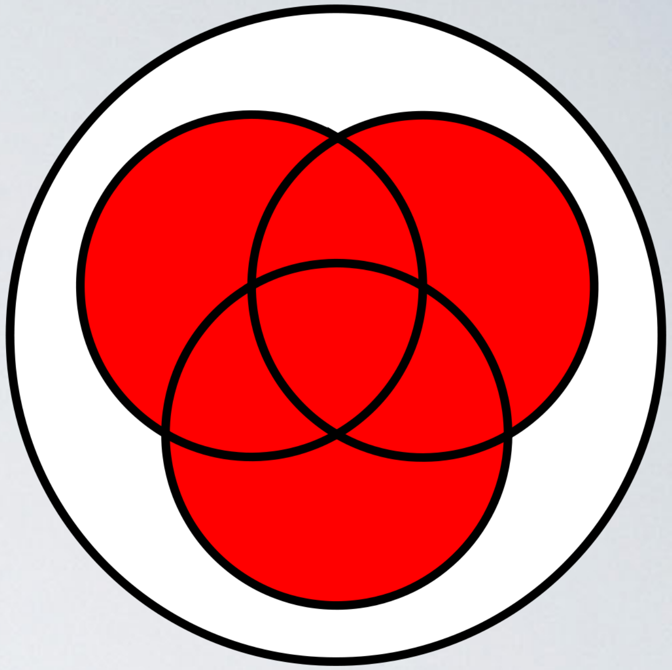
```
def search_index(index, word):
    output_list = list()
    count = 1
    while True:
        try:
            url = index.get(word + str(count))
            count = count + 1
        except KeyError:
            break
        output_list.append(url)
    return output_list
```

- 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 how to rank later in the course
    - …after we learn about graphs

Sets

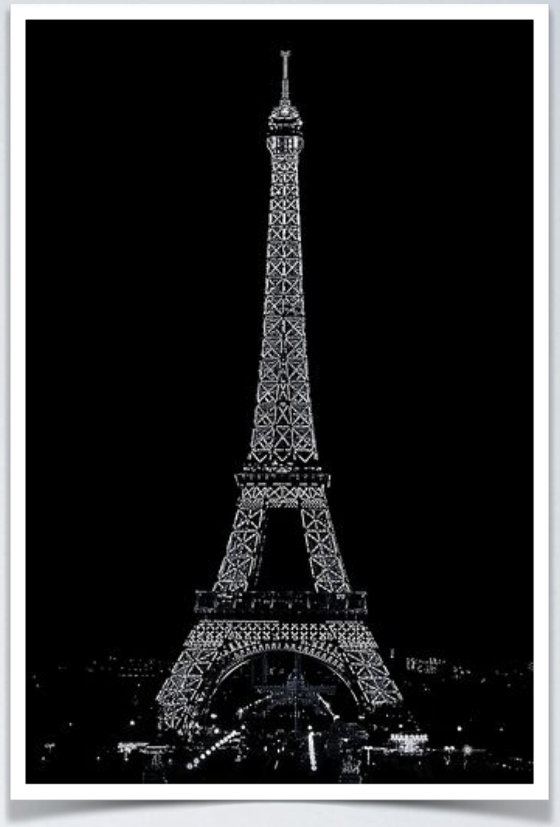


- Collection of elements that are
  - distinct and unordered
  - …unlike lists and arrays

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?
- Using an *expandable array*
  - add: `O(1)`
  - contains: `O(n)` (scan array)
  - remove: `O(n)` (find & compress)
- Can we do better?

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)
```

Expected `O(1)`

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

Expected `O(1)`

HashMap vs. HashSet

- HashMap, Python dictionaries
  - Hash table implementation of a dictionary
- HashSet, Python sets
  - Hash table implementation of a set