DSC 40B

Lecture 14 : Hashing

Warmup

Exercise

- How fast can we query/insert with these data structures?

	Query	Insert
Unsorted linked list	$\Theta(?)$	$\Theta(\ )$
Unsorted array	$\Theta(\ )$	$\Theta(\ )$
Sorted array	$\Theta(\ )$	$\Theta(\ )$
Balanced BST	$\Theta(\ )$	$\Theta(\ )$
Unbalanced BST	$\Theta(\ )$	$\Theta(\ )$

Exercise

- How fast can we query/insert with these data structures?

	Query	Insert
Unsorted linked list	$\Theta(n)$	$\Theta(\textbf{?})$
Unsorted array	$\Theta(?)$	$\Theta(?)$
Sorted array	$\Theta(?)$	$\Theta(?)$
Balanced BST	$\Theta(?)$	$\Theta(?)$
Unbalanced BST	$\Theta(?)$	$\Theta(?)$

Exercise

- How fast can we query/insert with these data structures?

	Query	Insert
Unsorted linked list	Θ(n)	Θ(1) #end
Unsorted array	Θ(**?**)	Θ(?)
Sorted array	Θ(?)	Θ(?)
Balanced BST	Θ(?)	Θ(?)
Unbalanced BST	Θ(?)	Θ(?)

Exercise

- How fast can we query/insert with these data structures?

	Query	**Insert**
Unsorted linked list	Θ(n)	Θ(1) #end
Unsorted array	Θ(n)	Θ(**?**)
Sorted array	Θ(?)	Θ(?)
Balanced BST	Θ(?)	Θ(?)
Unbalanced BST	Θ(?)	Θ(?)

Exercise

- How fast can we query/insert with these data structures?

	Query	Insert
Unsorted linked list	Θ(n)	Θ(1) #end
Unsorted array	Θ(n)	Θ(n)
Sorted array	Θ(**?**)	Θ(?)
Balanced BST	Θ(?)	Θ(?)
Unbalanced BST	Θ(?)	Θ(?)

Exercise

- How fast can we query/insert with these data structures?

	Query	Insert
Unsorted linked list	Θ(n)	Θ(1) #end
Unsorted array	Θ(n)	Θ(n)
Sorted array	Θ(log n)	Θ(**?**)
Balanced BST	Θ(?)	Θ(?)
Unbalanced BST	Θ(?)	Θ(?)

Exercise

- How fast can we query/insert with these data structures?

	Query	Insert
Unsorted linked list	Θ(n)	Θ(1) #end
Unsorted array	Θ(n)	Θ(n)
Sorted array	Θ(log n)	Θ(n)
Balanced BST	Θ(**?**)	Θ(?)
Unbalanced BST	Θ(?)	Θ(?)

Exercise

- How fast can we query/insert with these data structures?

	Query	Insert
Unsorted linked list	$\Theta(n)$	$\Theta(1)$ #end
Unsorted array	$\Theta(n)$	$\Theta(n)$
Sorted array	$\Theta(\log n)$	$\Theta(n)$
Balanced BST	$\Theta(\log n)$	$\Theta($ **?** $)$
Unbalanced BST	$\Theta(?)$	$\Theta(?)$

Exercise

- How fast can we query/insert with these data structures?

	Query	Insert
Unsorted linked list	Θ(n)	Θ(1) #end
Unsorted array	Θ(n)	Θ(n)
Sorted array	Θ(log n)	Θ(n)
Balanced BST	Θ(log n)	Θ(log n)
Unbalanced BST	Θ(**?**)	Θ(?)

Exercise

- How fast can we query/insert with these data structures?

	Query	Insert
Unsorted linked list	Θ(n)	Θ(1) #end
Unsorted array	Θ(n)	Θ(n)
Sorted array	Θ(log n)	Θ(n)
Balanced BST	Θ(log n)	Θ(log n)
Unbalanced BST	Θ(n)	Θ(**?**)

Exercise

- How fast can we query/insert with these data structures?

	Query	Insert
Unsorted linked list	Θ(n)	Θ(1) #end
Unsorted array	Θ(n)	Θ(n)
Sorted array	Θ(log n)	Θ(n)
Balanced BST	Θ(log n)	Θ(log n)
Unbalanced BST	Θ(n)	Θ(n)

Direct Address Tables

Counting Frequencies

- How many times does each age appear?

PID	Name	Age
A1843	Wan	24
A8293	Deveron	22
A9821	Vinod	41
A8172	Aleix	17
A2882	Kayden	4
A1829	Raghu	51
A9772	Cui	48
⋮	⋮	⋮

Exercise

What data structure would you use to store the age counts?

Direct Address Tables

- **Idea**: keep an **array** arr of length, say, 125.
- Initialize to zero.
-

0	0	0	0	…	0	0	0
0	1	2	3	…	5	6	124

Direct Address Tables

- **Idea**: keep an **array** `arr` of length, say, 125.
- Initialize to zero.

0	0	0	0	…	0	0	0
0	1	2	3	…	5	6	124

- If we see age x, increment arr[x] by one. Say, we see 2:
  - `arr[2] += arr[2] + 1`

0	0	1	0	…	0	0	0
0	1	2	3	…	5	6	124

Building the Table

```
# loading the table
table = np.zeros(125)

for age in ages:
   table[age] += 1
```

Building the Table

```
# loading the table
table = np.zeros(125)

for age in ages:
   table[age] += 1
```

Time complexity if there are n people?

A: Constant

B: log n

C: n

D: n^2

Query

```
# query: how many people are 55?
print(table[55])
```

- Time complexity if there are n people?

Time complexity if there are n people?

A: Constant

B: log n

C: n

D: n^2

Query

```
# query: how many people are 55?

print(table[55])
```

- Time complexity if there are n people?
  - $\Theta(1)$

Counting Names

PID	Name	Age
A1843	Wan	24
A8293	Deveron	22
A9821	Vinod	41
A8172	Aleix	17
A2882	Kayden	4
A1829	Raghu	51
A9772	Cui	48
⋮	⋮	⋮

Downsides

- DATs are **fast**.
- What are the downsides of DATs?
- Could we use a DAT to store:
  - zip codes?
  - phone numbers?
  - credit card numbers?
  - names?

Downsides

- Things being stored must be integers, or convertible to integers.
  - why? valid array indices
- Must come from a small range of possibilities
  - why? memory usage.
  - example: phone numbers

Running time for Direct Hashing or DAT

```
class StudentDataBase:
  allStudents = np.zeros(4294967268)
  def add (s):
      index = s.studentId
      allStudents[index] = s
  def get (s):
      index = s.studentId
      return allStudent[index]
  def remove(s):
      index = s.studentId
      allStudents[index] = None
```

Space complexity is horrible :(

- Allocating 4 billion entries for a UCSD student database is extremely wasteful.
  - The universe of keys is far larger than the number we expect to ever want to store.
- What if we decrease the size of the table from 4 billion entries to storing, say, just 100,000?
- 100,000 is still far higher than the actual number of UCSD students, so there should be plenty of space.

Hash Tables

- **Insight**: anything can be "converted" to an integer through **hashing**.
- But not *uniquely*!
- Hash tables have many of the same advantages as DATs, but work more generally.

Hashing

Hashing

- One of the most important ideas in CS.
- **Tons** of uses:
    - Verifying message integrity.
    - Fast queries on a large data set.
    - Identify if file has changed in version control.

Hash Function

- A **hash function** takes a (large) *object* and returns a (smaller) "fingerprint" of that object.
- Usually the fingerprint is a number, guaranteed to be in some range.

How?

- Looking at certain bits, combining them in ways that ***look*** random (but aren’t!)

Hash Function Properties

- Hashing same thing **twice** returns the **same** hash.
- Unlikely that different things have same fingerprint.
  - But not impossible!

Collisions

- Hash functions map objects to numbers in a defined range.
  - **Example**: given image, return number in [0, 1, 2, … , 1024]
- There will be two images with the same hash.
  - **Pigeonhole principle**: if there are n pigeons, $< n$ holes, there will a hole with ≥ 2 pigeons.
- **Collision**: two objects have the **same** hash

“Good” Hash Functions

- A **good hash** function tries to minimize collisions.

Hashing in Python

The **hash** function computes a hash.

```
>>> hash("This is a test")
-6704585799574772O3
>>> hash("This is a test")
-6704585799574772O3
>>> hash("This is a test!")
1860306055874153109
```

Example

- MD5 is a *cryptographic* hash function.
  - Hard to “reverse engineer” input from hash.
- Returns a really large number in `hex`.

  a741d8524a853cf83ca21eabf8cea190

- Used to “fingerprint” whole files.

Example

```
> echo "My name is Marina" | md5
60b5504d3410d0f97796c95829adebfd
> echo "My name is Marina" | md5
60b5504d3410d0f97796c95829adebfd
> echo "My name is marina" | md5
4a3c494e411e94d2be7c32ac7461208f
```

Example

```
>>> md5 finalA.pdf
MD5 (finalA.pdf) = ff5bcc7dcc4815aa221774b1507c42c5
```

Why? Useful for security reasons

- I release a piece of software.
- I host it on Google Drive.
- Someone (Google, US Gov., etc.) decides to insert extra code into software to spy on users.
- You have no way of knowing.

Another Use: De-duplication

- Building a massive training set of images.
- Given a new image, is it already in my collection?
- Don't need to compare images pixel-by-pixel!
- Instead, compare **hashes**.

Hashing for Data Scientists

- Don’t need to know much about *how* the hash function works.
- But should know how they are used.

Hash Tables

Membership Queries

- **Given**: a collection of n numbers and a target t.
- **Find**: determine if t is in the collection.

Goal

- DATs are fast, but won't work for things that aren't numbers in a small range.
- **Idea**: hash objects to numbers in a small range, use a DAT.
- But must **deal with collisions**.

Hash Tables

- Pick a table size m.
  - Usually $m \approx$ number of things you'll be storing.
- Create hash function to turn input into a number in $\{0, 1, \dots, m-1\}$.
- Create DAT with m bins.

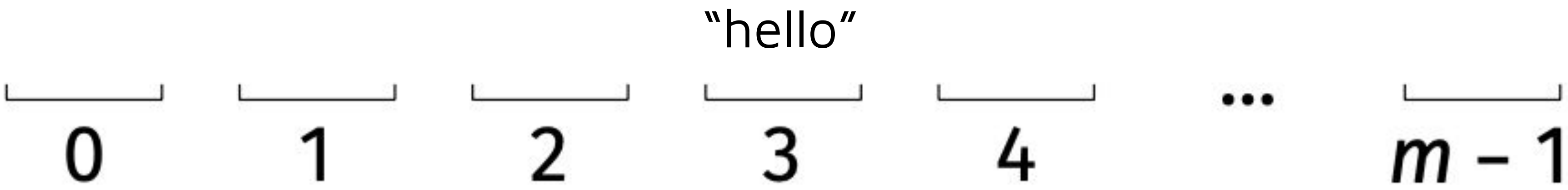
Example

```
hash('hello') == 3
hash('data') == 0
hash('science') == 4
```

0	1	2	3	4	...	$m - 1$

Example

```
hash('hello') == 3
hash('data') == 0
hash('science') == 4
```



Example

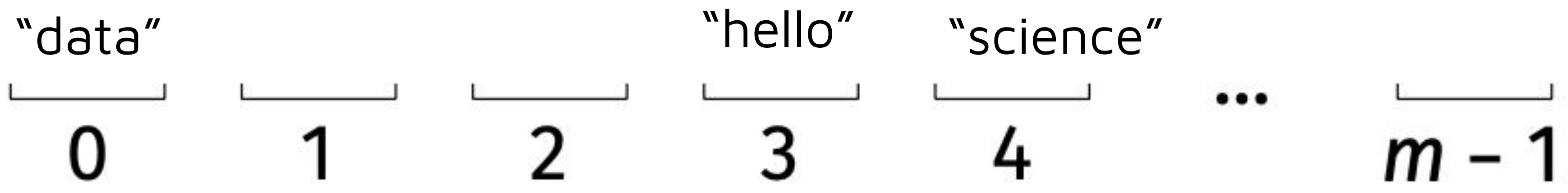
```
hash('hello') == 3
hash('data') == 0
hash('science') == 4
```

"data" "hello"

0 1 2 3 4 ... $m - 1$

Example

```
hash('hello') == 3
hash('data') == 0
hash('science') == 4
```

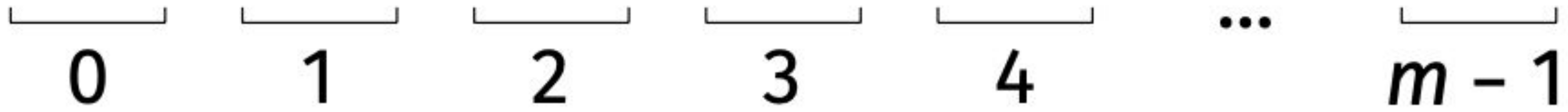


Collisions

- The **universe** is the set of all *possible* inputs.
- This is usually **much larger** than m (even infinite).
- **Not possible** to assign each input to a unique bin.
- If hash(a) == hash(b), there is a **collision**.

Example

```
hash('hello') == 3
hash('data') == 0
hash('science') == 3
```



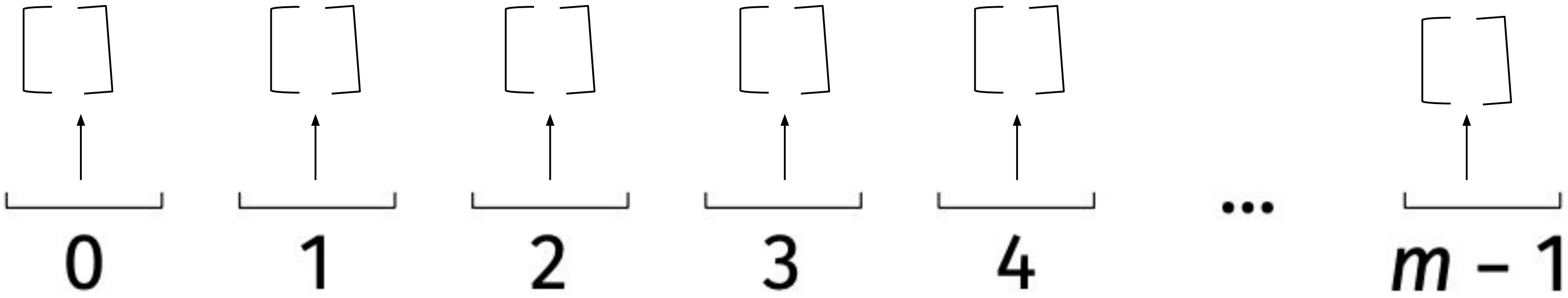
Example

```
hash('hello') == 3
hash('data') == 0
hash('science') == 3
```

"data"			"hello"			
0	1	2	3	4	...	$m - 1$

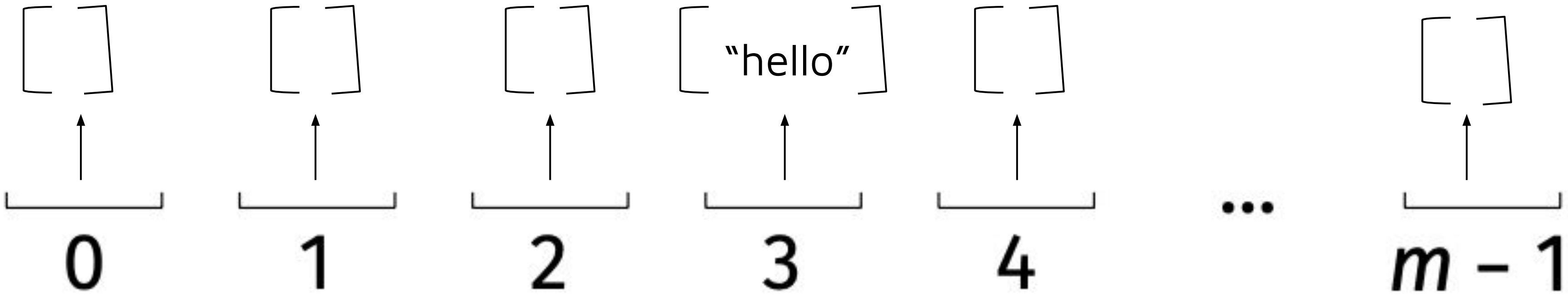
Example

```
hash('hello') == 3
hash('data') == 0
hash('science') == 3
```



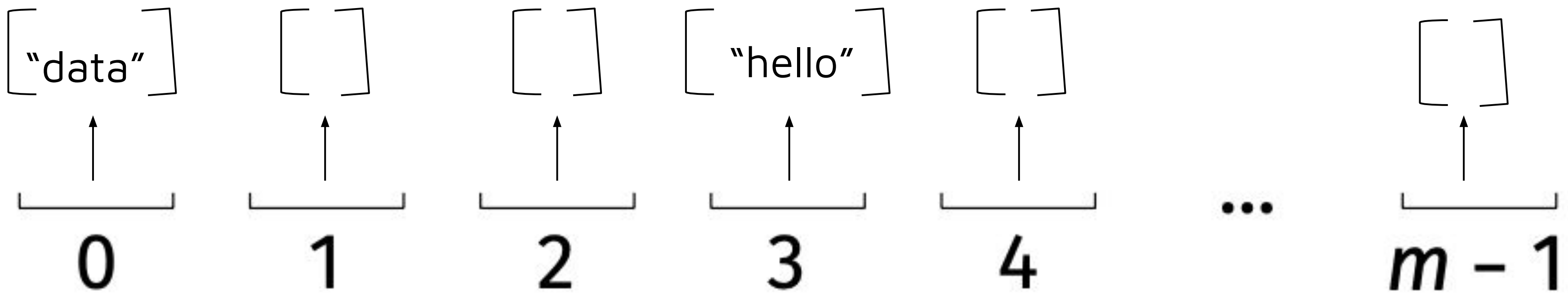
Example

```
hash('hello') == 3
hash('data') == 0
hash('science') == 3
```



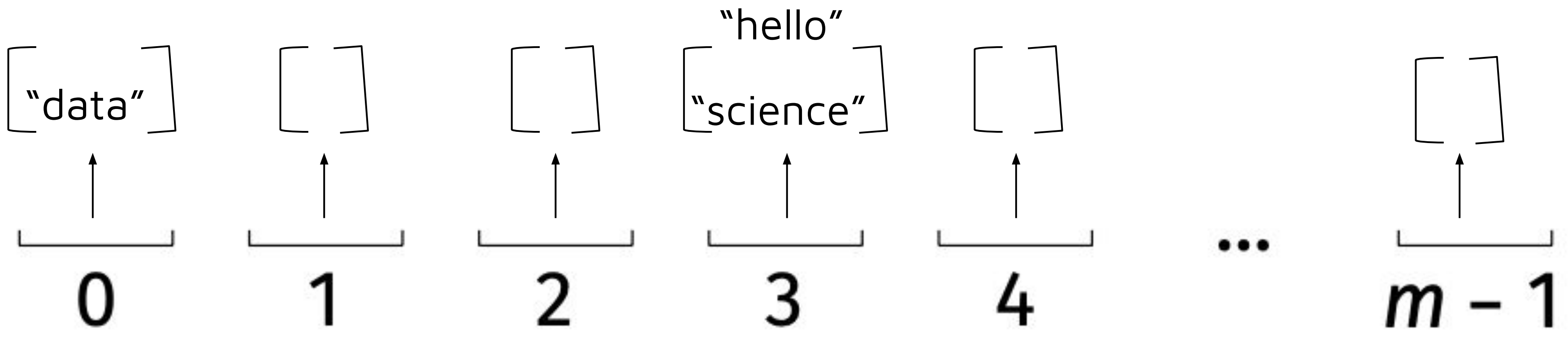
Example

```
hash('hello') == 3
hash('data') == 0
hash('science') == 3
```



Example

```
hash('hello') == 3
hash('data') == 0
hash('science') == 3
```

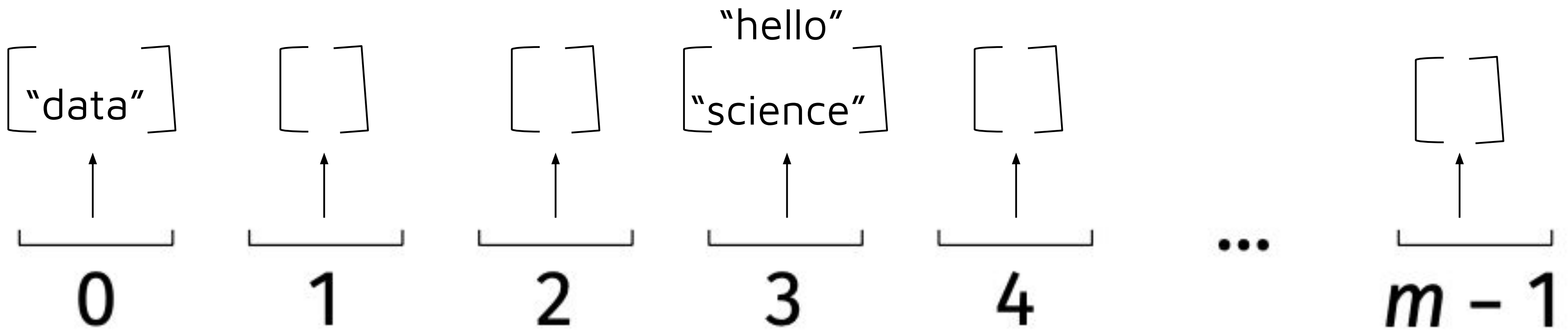


Chaining

- Collisions stored in same bin, in linked list.
- **Query**: Hash to find bin, then linear search.

Chaining

- Collisions stored in same bin, in linked list.
- **Query**: Hash to find bin, then linear search.



The Idea

- A **good** hash function will utilize all bins evenly.
  - Looks like ***uniform*** random distribution.
- If $m \approx n$, then **only a few** elements in each bin.
- As we add more elements, we need to add bins.

Average Case

- n elements.
- m bins.
- Assume elements placed randomly (but *deterministically*) in bins.
- **Expected** size of linked list inside the bin: $\boldsymbol{n/m}$.

Analysis

- **Query**:
  - **$\Theta(1)$** to find bin (hashing step)
  - **$\Theta(n/m)$** for a linear search in a list.
  - **Total**: $\Theta(1 + n/m)$. $(m \approx n)$
  - We *usually* guarantee $m = O(n) \Longrightarrow \Theta(1)$.

Worst Case mic

- Everything **hashed to same bin**.
    - Really unlikely!
    - **Adversarial attack** (next slide)
- **Query**:
    - $\Theta(1)$ to find bin
    - $\Theta(n)$ for linear search
    - Total: $\Theta(n)$

Adversarial attack: HashDoS attack (back)

- A `HashDoS` attack targets systems that use hash tables by flooding them with **keys that all hash to the same value**.
- How?
  - **Weak** hash functions (like early implementations of hash() in some languages).
  - **Unprotected** hash maps in web servers, APIs, or input-parsing code.
  - By sending thousands of colliding inputs, they slow down or crash the system.

Exercise

What is the worst case time complexity of inserting an element into a hash table that uses chaining with linked lists?

Growing the Hash Table

- Insertions take Θ(1) **unless** the hash table needs to *grow*.
- We need to ensure that $m \le c \cdot n$.
  - Otherwise, **too many** collisions.
- If we add a bunch of elements, we'll need to **increase** m.
- Increasing m mean allocating a new array, Θ(m) = Θ(n) time.

Main Idea

Hash tables support **constant** (*expected*) time insertion and membership queries.

Dictionaries

- Hash tables can also be used to store (key, value) pairs.
- Often called `dictionaries` or `associative arrays`.

Hashing in Python

- `dict` and `set` implement hash tables.
- Querying is done using **`in`**:

```
# make a set
>>> L = {3, 6, -2, 1, 7, 12}
>>> 1 in L          # Theta(1)
True

>>> 8 in L          # Theta(1)
False
```

Hashing in Python

- Querying is done using **in**:

```
# make a list
>>> L = [3, 6, -2, 1, 7, 12]
>>> 1 in L      # Theta(?)
True

>>> 8 in L      # Theta(?)
False
```

Hashing in Python

- Querying is done using **in**:

```
# make a list
>>> L = [3, 6, -2, 1, 7, 12]
>>> 1 in L      # Theta(n)
True

>>> 8 in L      # Theta(n)
False
```

Thank you!

Do you have any questions?

CampusWire!