Data Structures and Algorithms

Hashing

CS 225
G Carl Evans

April 12, 2024

Department of Computer Science

Slides by Brad Solomon

Learning Objectives

Motivate and formally define a hash table

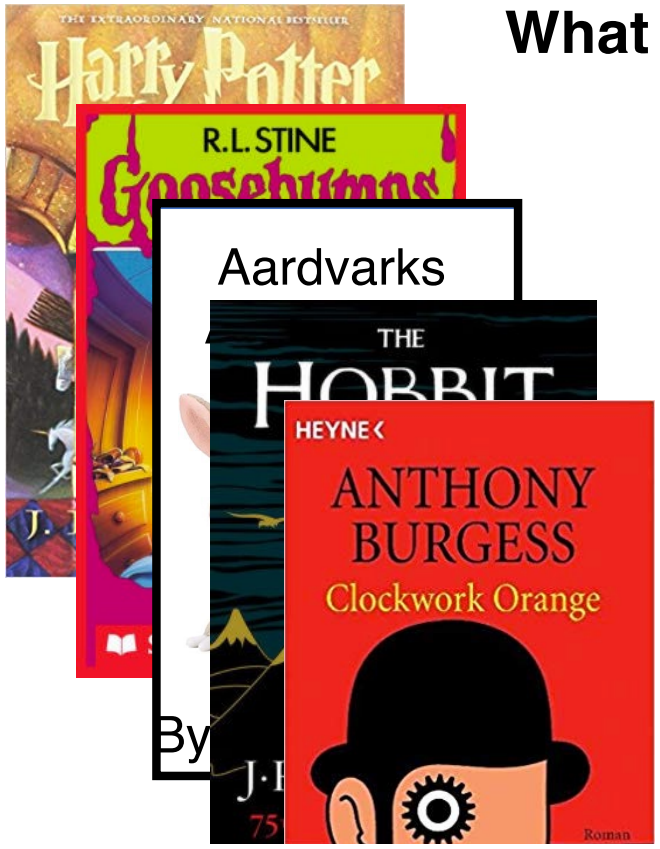
Discuss what a ‘good’ hash function looks like

Identify the key weakness of a hash table

Introduce strategies to “correct” this weakness

Data Structure Review

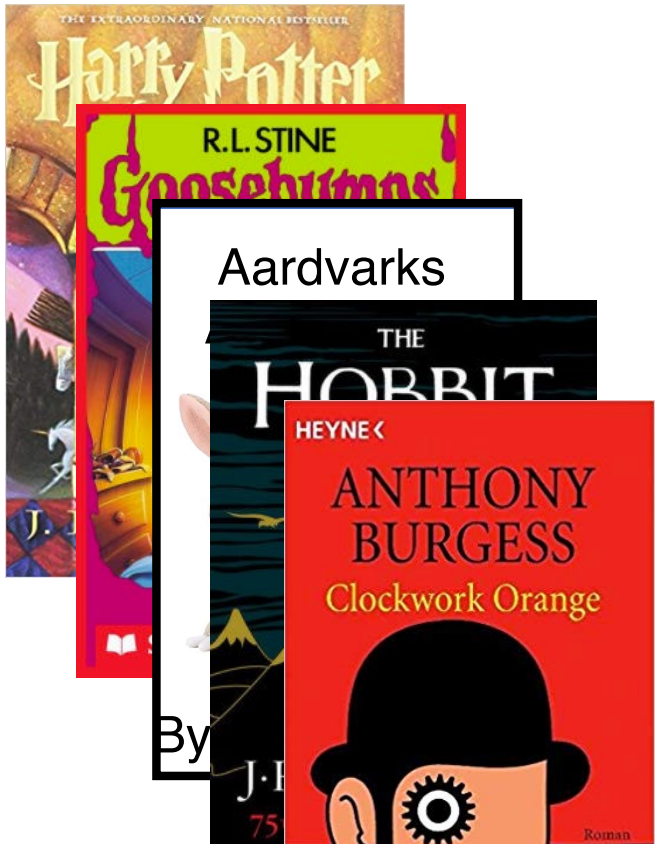
I have a collection of books and I want to store them in a dictionary!



What data structures can I use here?

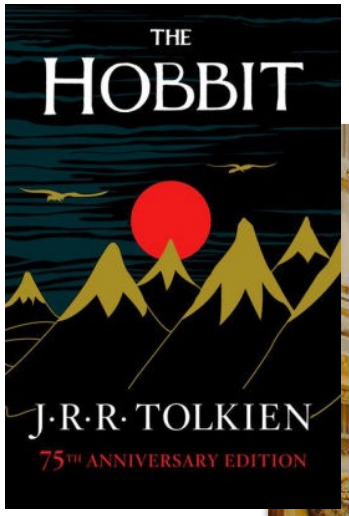
Data Structure Review

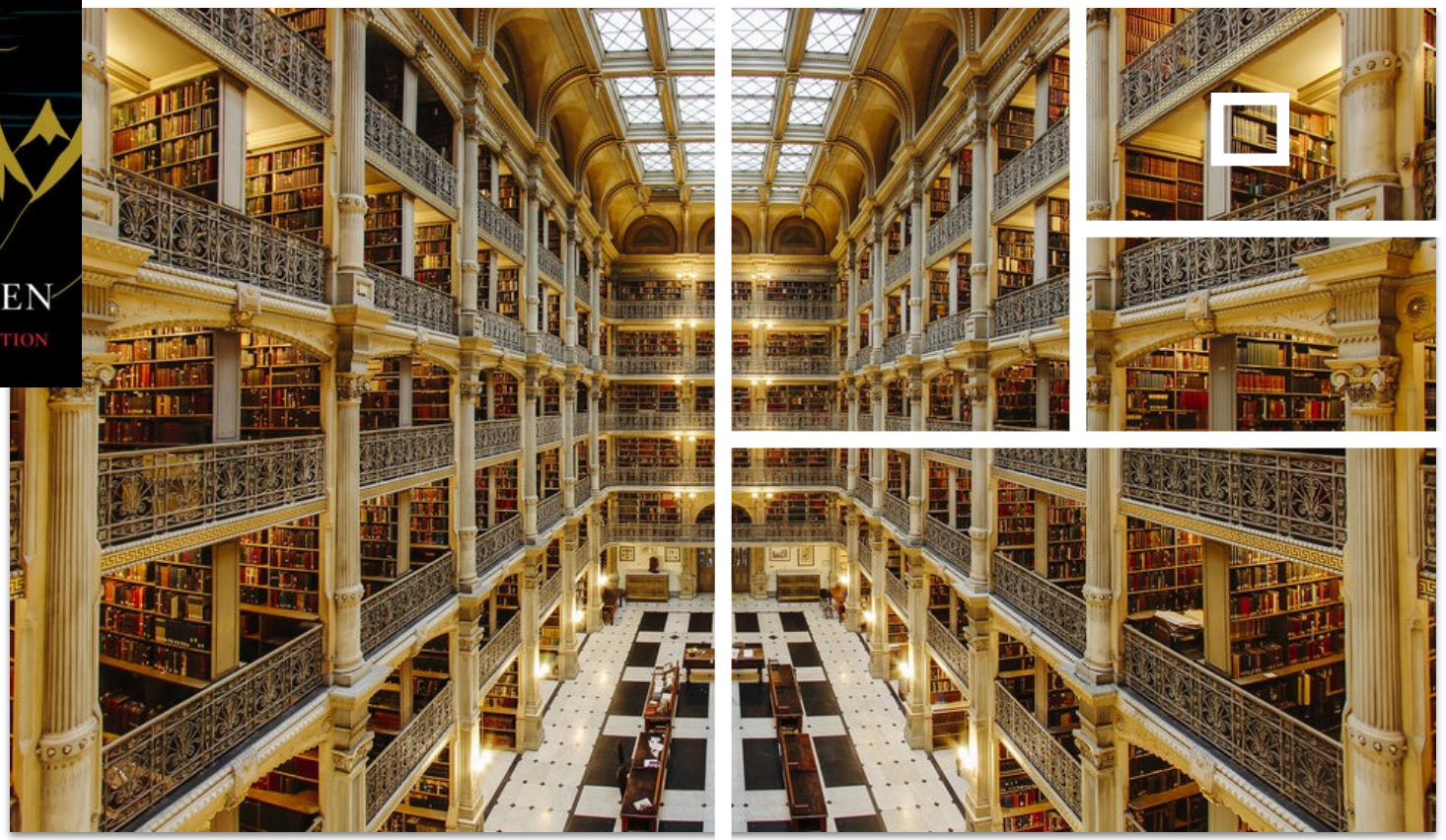
I have a collection of books and I want to store them in a dictionary!



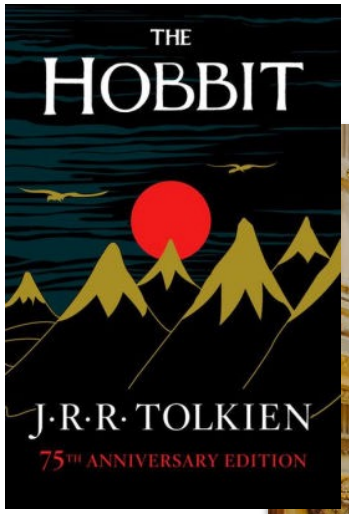
	Sorted Array	BST	AVL Tree
Find			
Insert			
Remove			

What if $O(log\ n)$ isn't good enough?



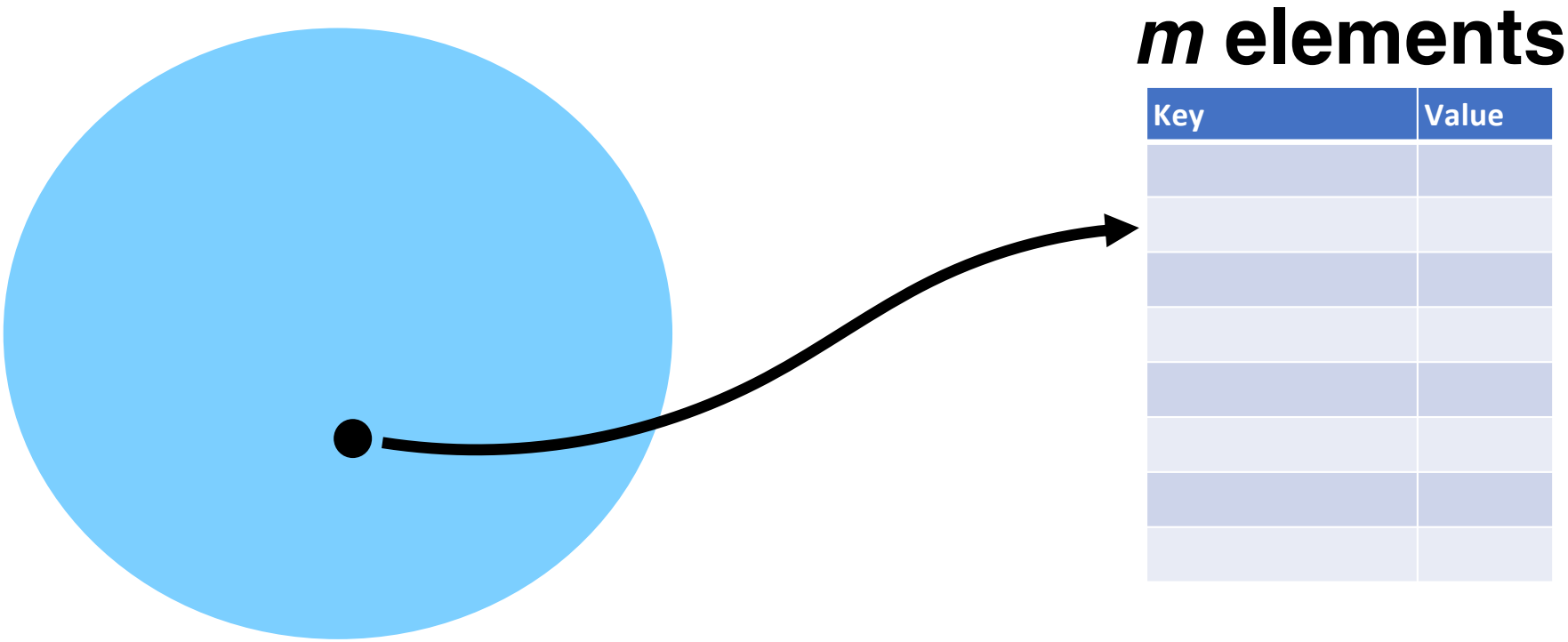


What if $O(log\ n)$ isn't good enough?



Hash Function

Maps a **keyspace**, a (mathematical) description of the keys for a set of data, to a set of integers.



Hash Function

A hash function ***must*** be:

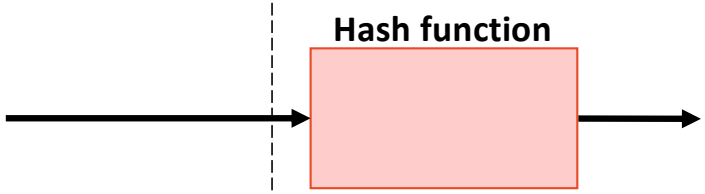
- **Deterministic**:

- **Efficient**:

- **Defined for a certain size table:**

Hash Function

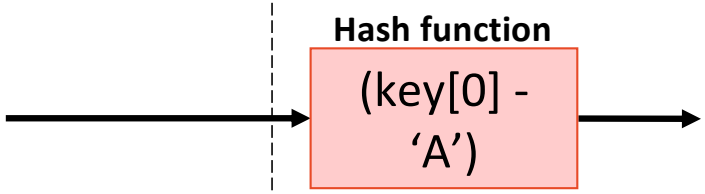
(Angrave, CS 341)
(Beckman, CS 101)
(Challon, CS 125)
(Davis, CS 105)
(Evans, CS 225)
(Fagen-Ulmschneider, CS 107)
(Gunter, CS 422)
(Herman, CS 233)



Key	Value

Hash Function

(Angrave, CS 341)
(Beckman, CS 101)
(Challon, CS 125)
(Davis, CS 105)
(Evans, CS 225)
(Fagen-Ulmschneider, CS 107)
(Gunter, CS 422)
(Herman, CS 233)



Key	Value
Angrave	341
Beckman	101
Challon	125
Davis	105
Evans	225
Fagen-U	107
Gunter	422
Herman	233

General Hash Function

An $O(1)$ deterministic operation that maps all keys in a universe U to a defined range of integers $[0, \ldots, m-1]$

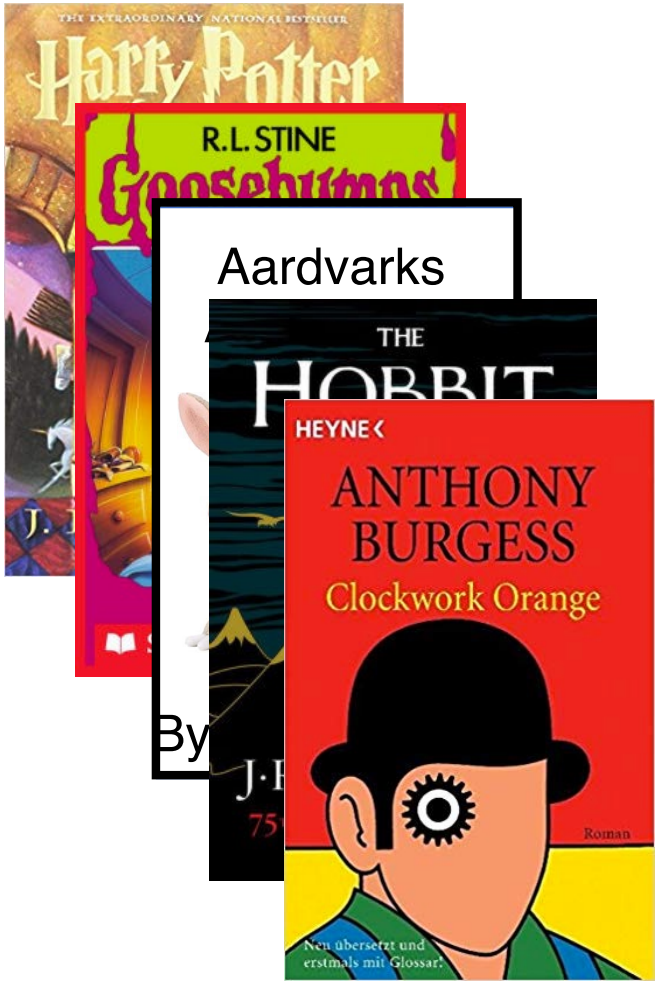
- A **hash**:

- A **compression**:

Choosing a good hash function is tricky…

- Don't create your own (yet*)

Hash Function

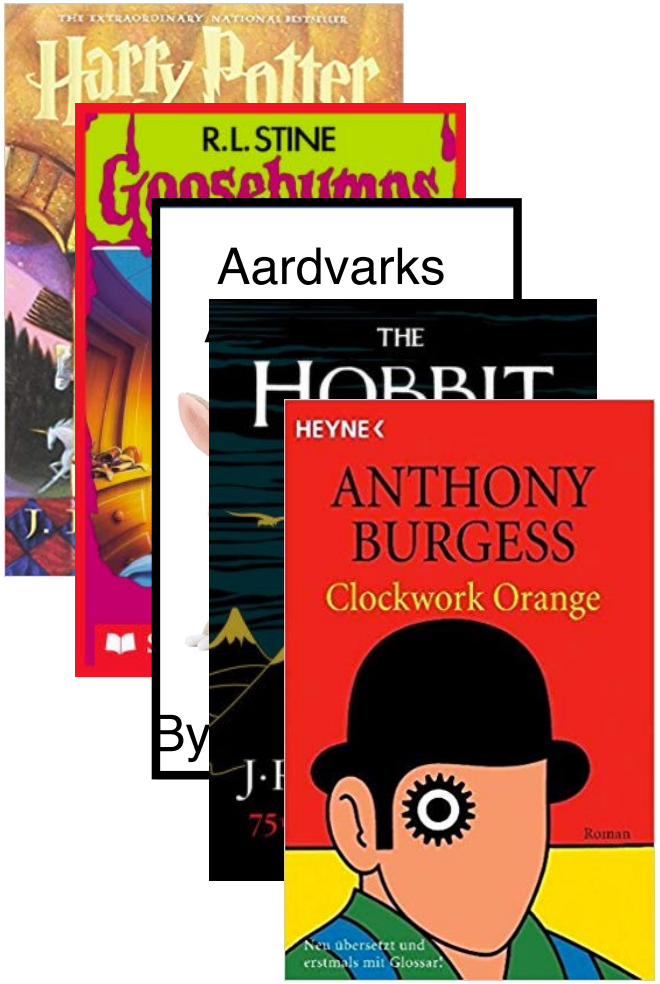


$$h(k) = (k.firstName[0] + k.lastName[0]) \% m$$

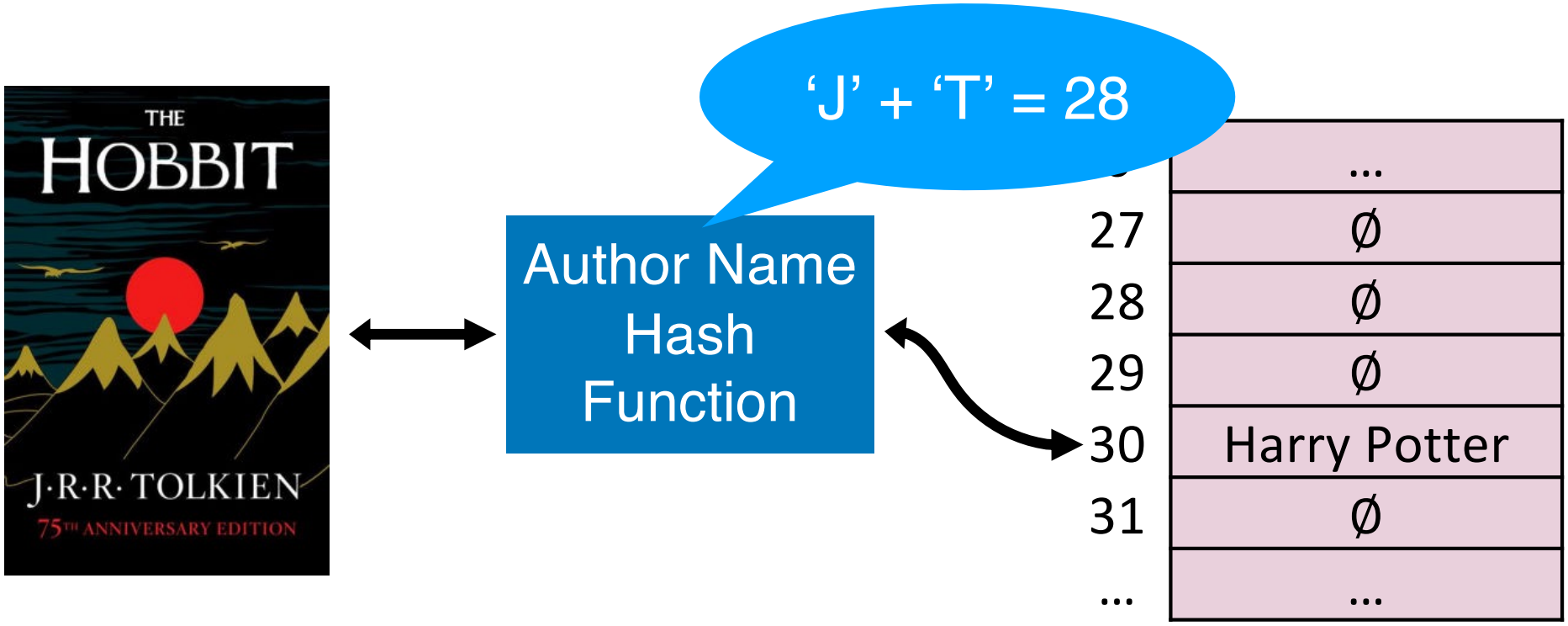
$$h(k) = (rand() * k.numPages) \% m$$

$$h(k) = (k.order\_1st\_read\_by\_me) \% m$$

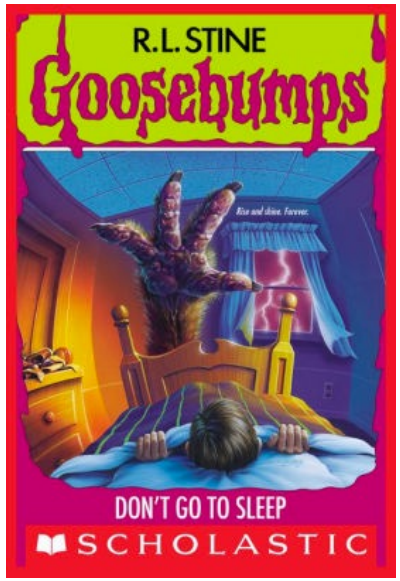
Hash Function

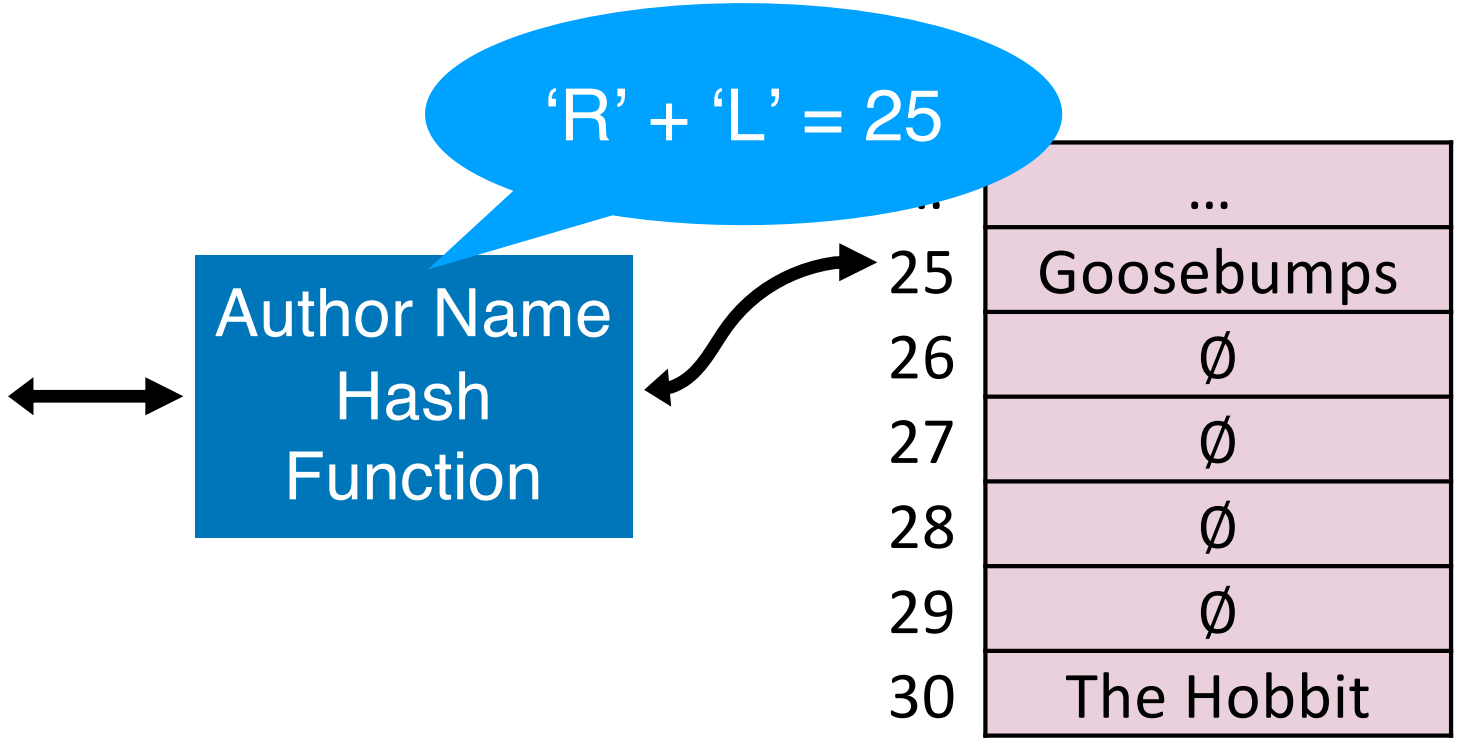


Hash Function

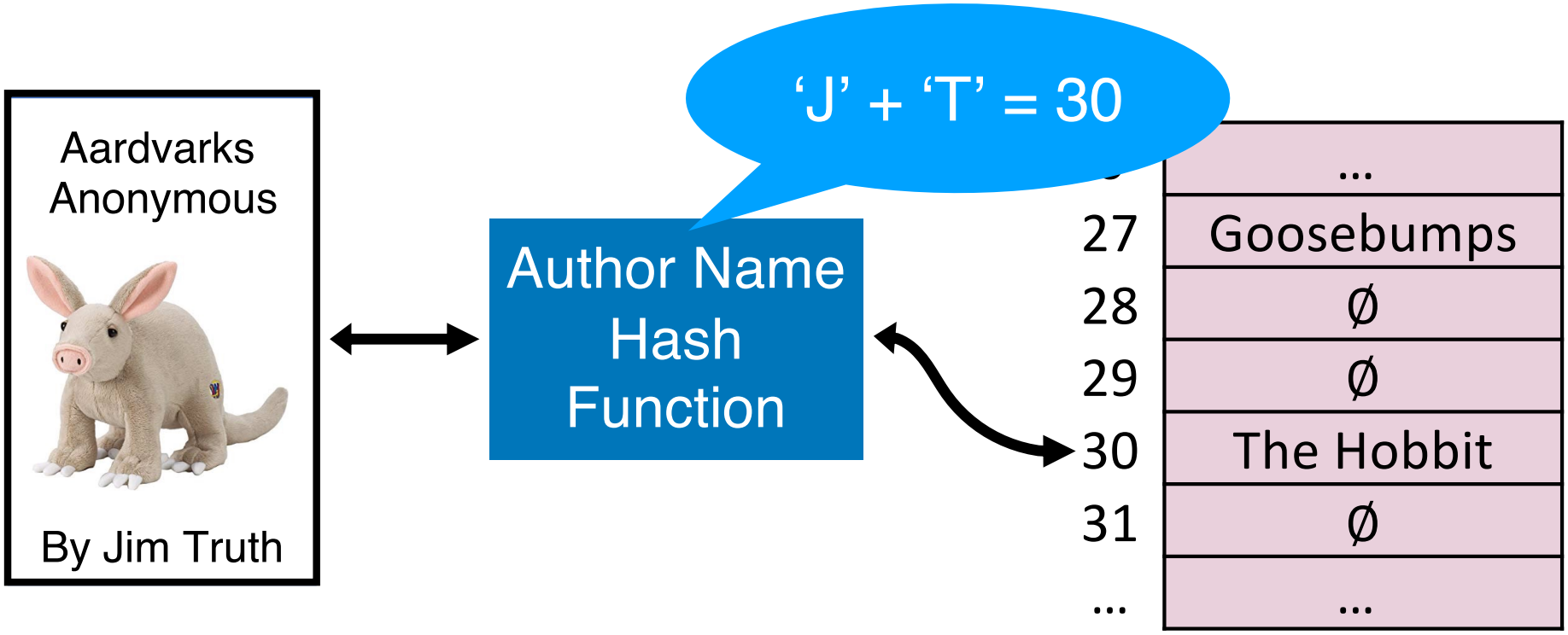


Hash Function



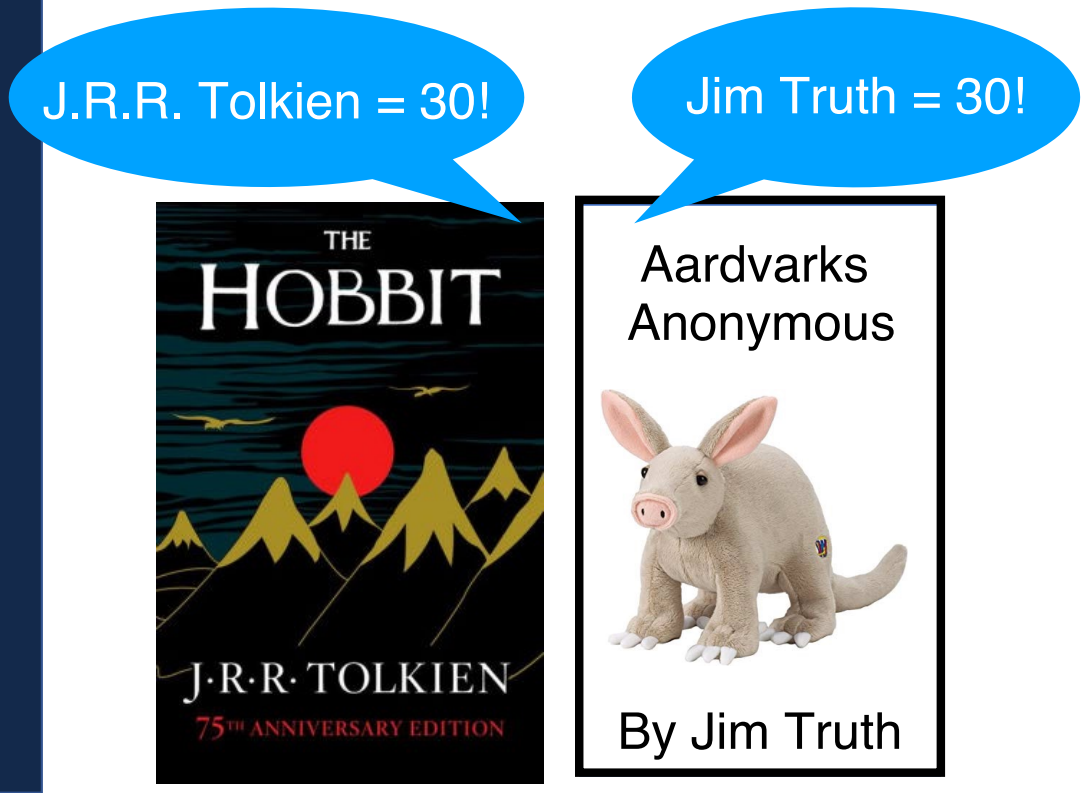


Hash Function



Hash Collision

A ***hash collision*** occurs when multiple unique keys hash to the same value



...	...
27	Ø
28	Ø
29	Ø
30	???
31	Ø
...	...

Perfect Hashing

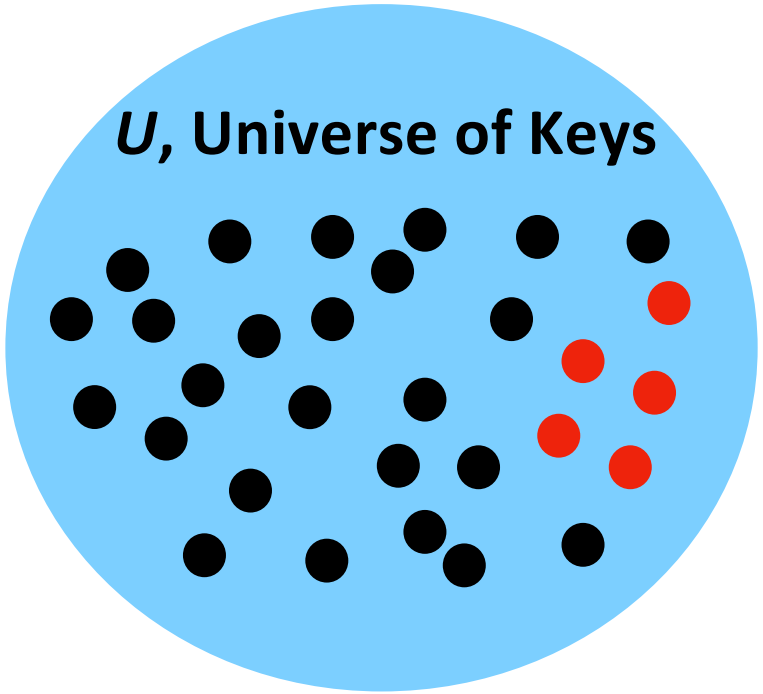
If $m \geq S$, we can write a *perfect* hash with no collisions

m elements

Key	Value

General Purpose Hashing

In CS 225, we want our hash functions to work *in general*.

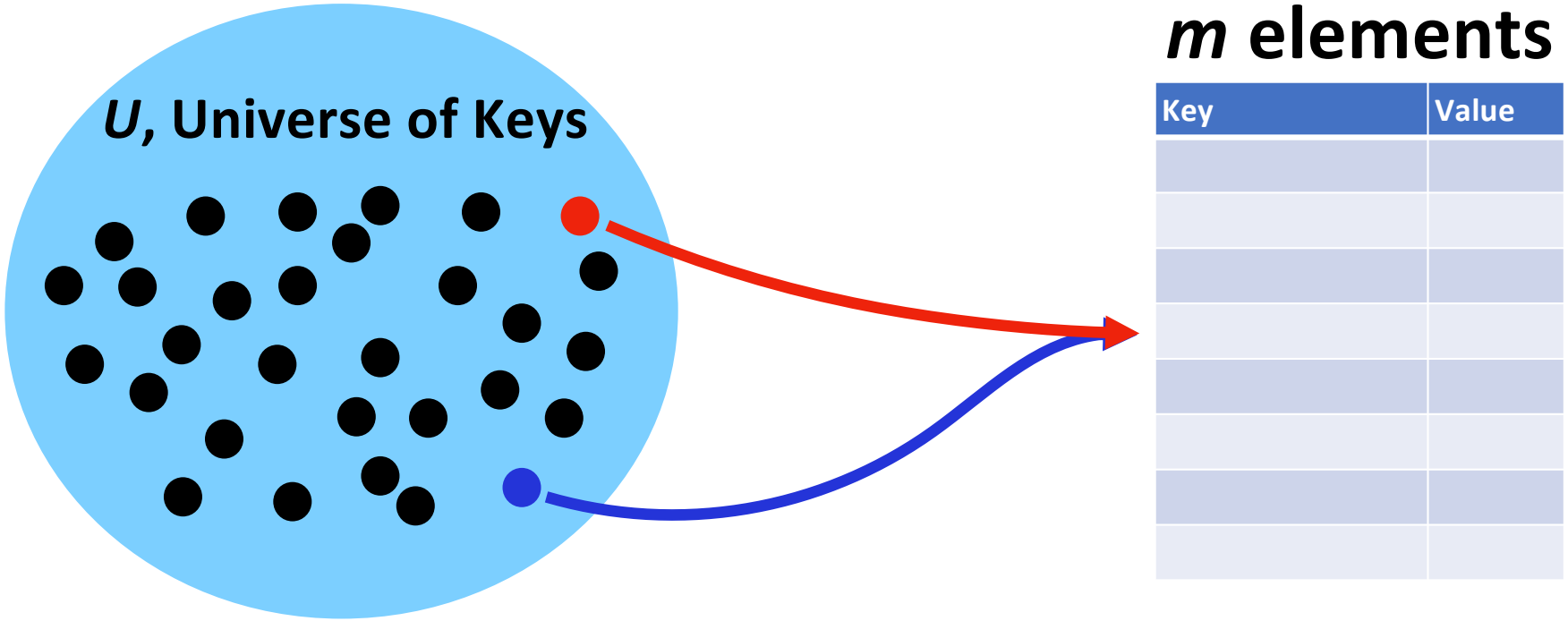


***m* elements**

Key	Value

General Purpose Hashing

If $m < U$, there must be at least one hash collision.



General Purpose Hashing

By fixing h, we open ourselves up to adversarial attacks.

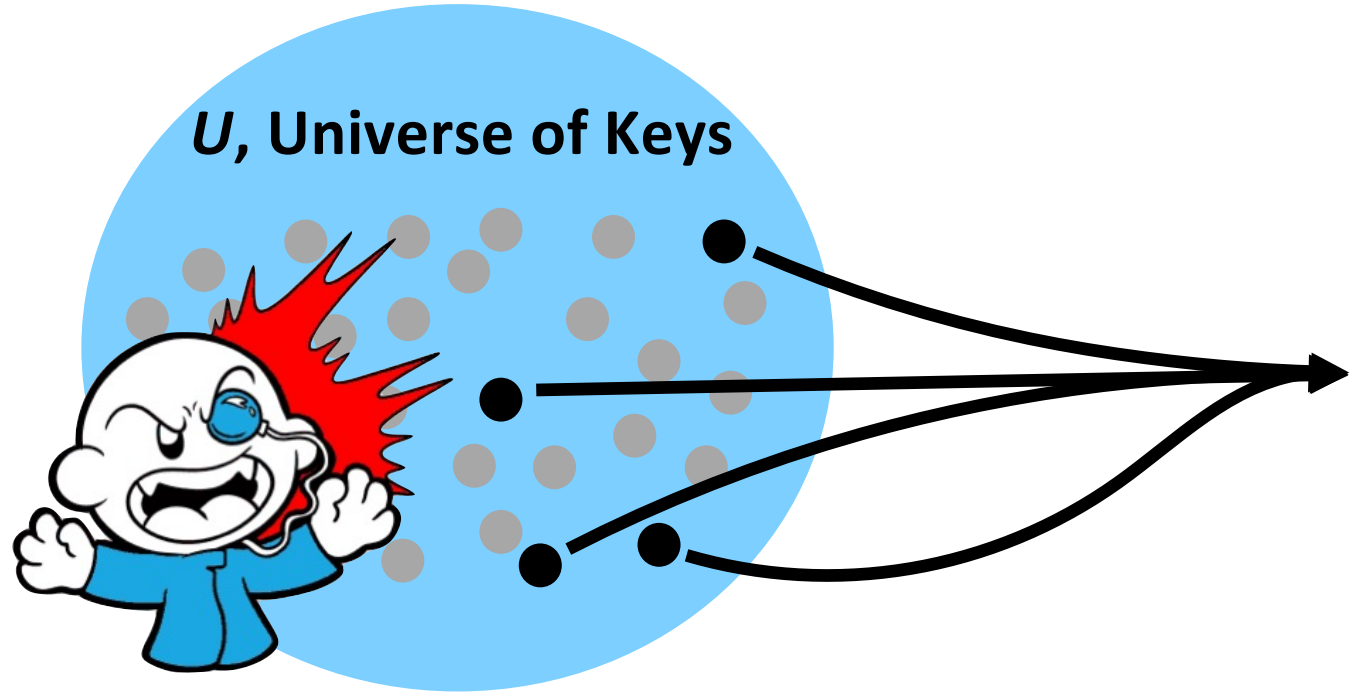


Image by Matthew Loffhagen

A Hash Table based Dictionary

Client Code:

```
1  Dictionary<KeyType, ValueType> d;
2  d[k] = v;
```

A **Hash Table** consists of three things:

1. A hash function

2. A data storage structure

3. **A method of addressing *hash collisions***

Open vs Closed Hashing

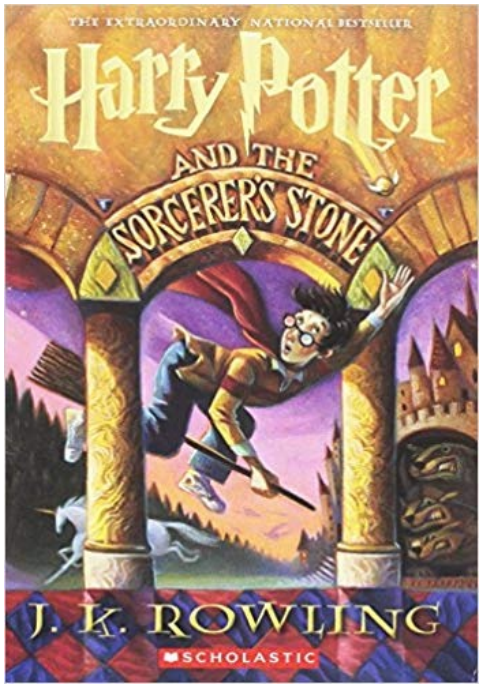
Addressing hash collisions depends on your storage structure.

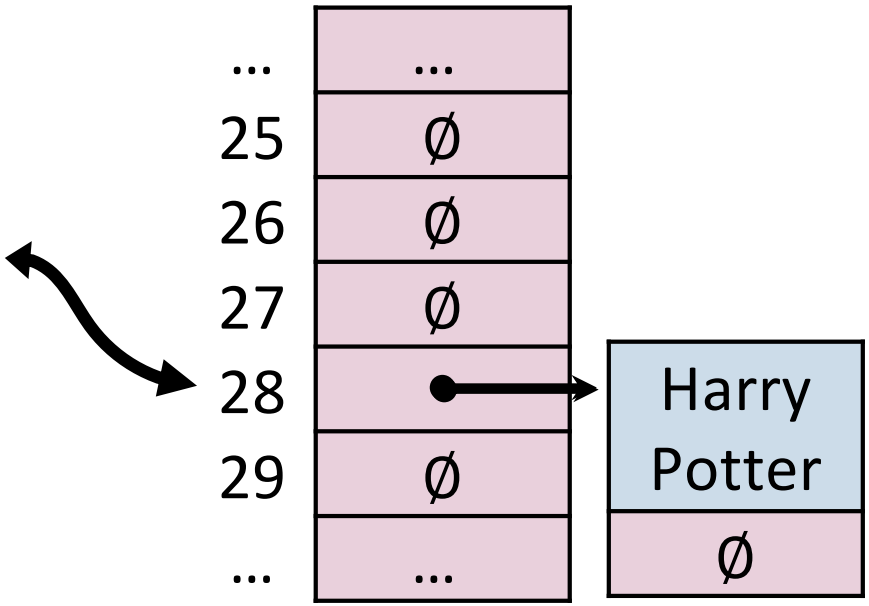
- **Open Hashing:**

- **Closed Hashing:**

Open Hashing

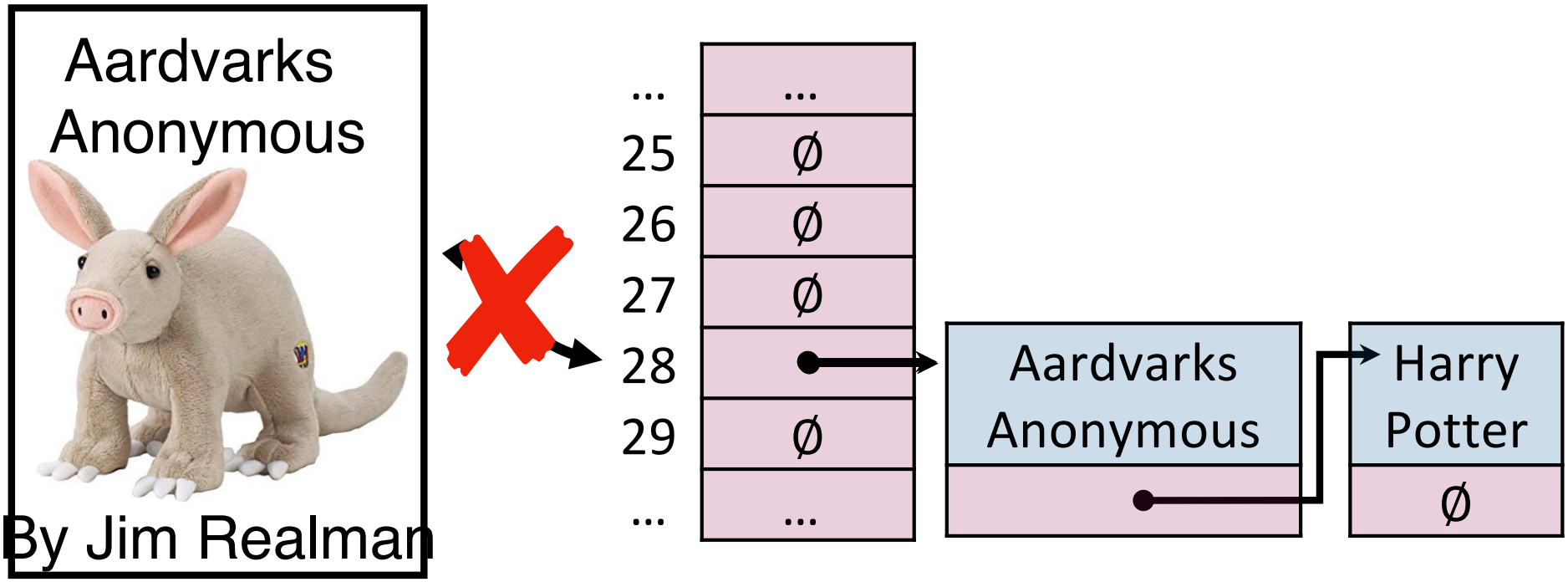
In an ***open hashing*** scheme, key-value pairs are stored externally (for example as a linked list).





Hash Collisions (Open Hashing)

A ***hash collision*** in an open hashing scheme can be resolved by _____. This is called ***separate chaining*.**



Insertion (Separate Chaining)

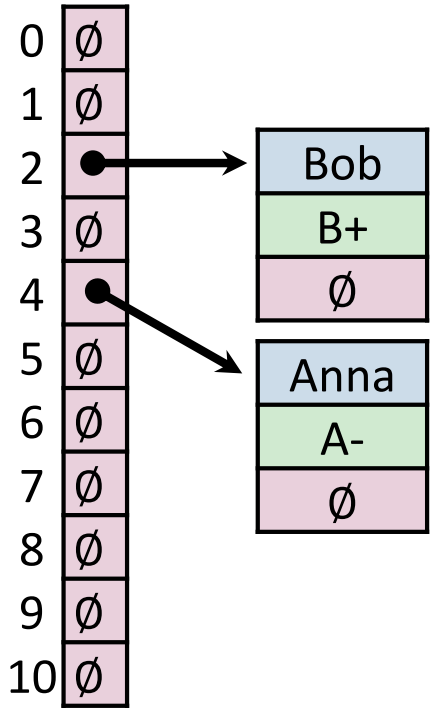
```
_insert("Bob")

_insert("Anna")
```

Key	Value	Hash
Bob	**B+**	**2**
Anna	**A-**	**4**
Alice	A+	4
Betty	B	2
Brett	A-	2
Greg	A	0
Sue	B	7
Ali	B+	4
Laura	A	7
Lily	B+	7

0	Ø
1	Ø
2	Ø
3	Ø
4	Ø
5	Ø
6	Ø
7	Ø
8	Ø
9	Ø
10	Ø

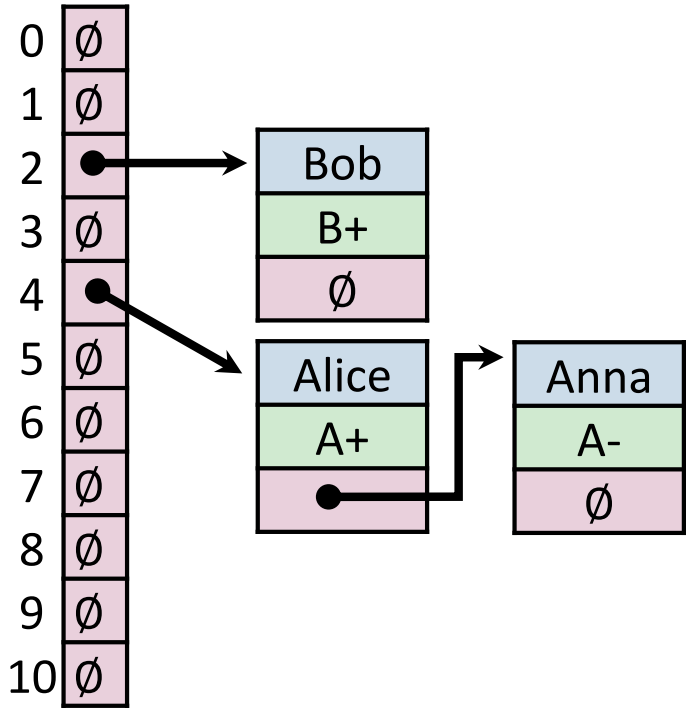
Insertion (Separate Chaining) `_insert("Alice")`

Key	Value	Hash
Bob	B+	2
Anna	A-	4
Alice	**A+**	**4**
Betty	B	2
Brett	A-	2
Greg	A	0
Sue	B	7
Ali	B+	4
Laura	A	7
Lily	B+	7



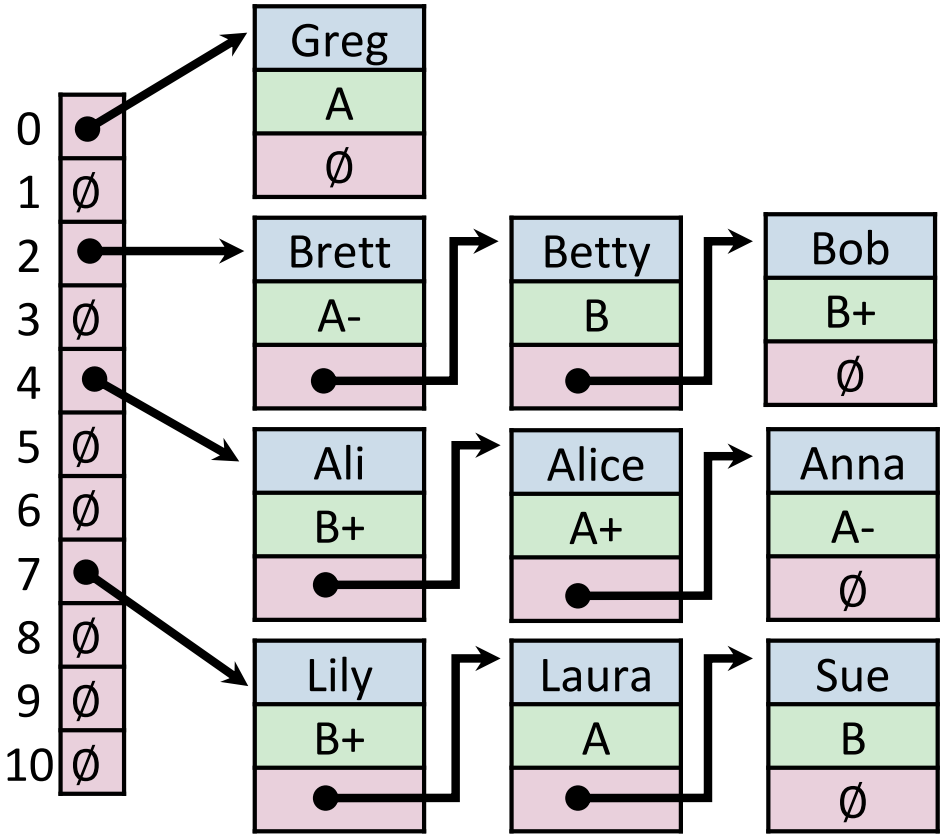
Insertion (Separate Chaining)

Key	Value	Hash
Bob	B+	2
Anna	A-	4
Alice	A+	4
Betty	**B**	**2**
Brett	A-	2
Greg	A	0
Sue	B	7
Ali	B+	4
Laura	A	7
Lily	B+	7



Insertion (Separate Chaining)

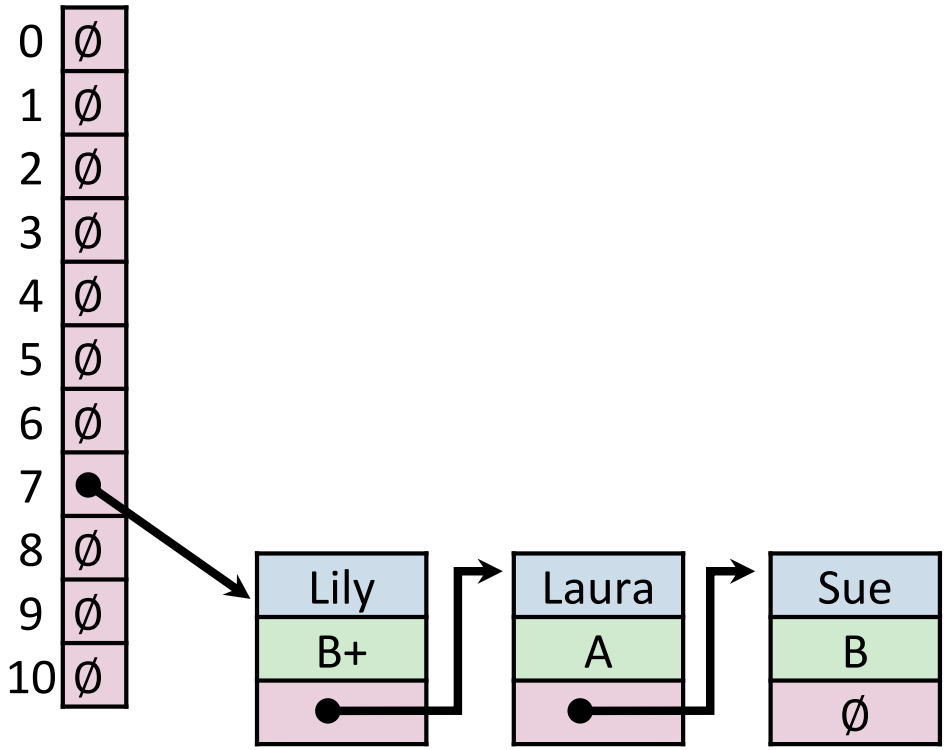
Key	Value	Hash
Bob	B+	2
Anna	A-	4
Alice	A+	4
Betty	B	2
Brett	A-	2
Greg	A	0
Sue	B	7
Ali	B+	4
Laura	A	7
Lily	B+	7



Find (Separate Chaining)

`_find("Sue")`

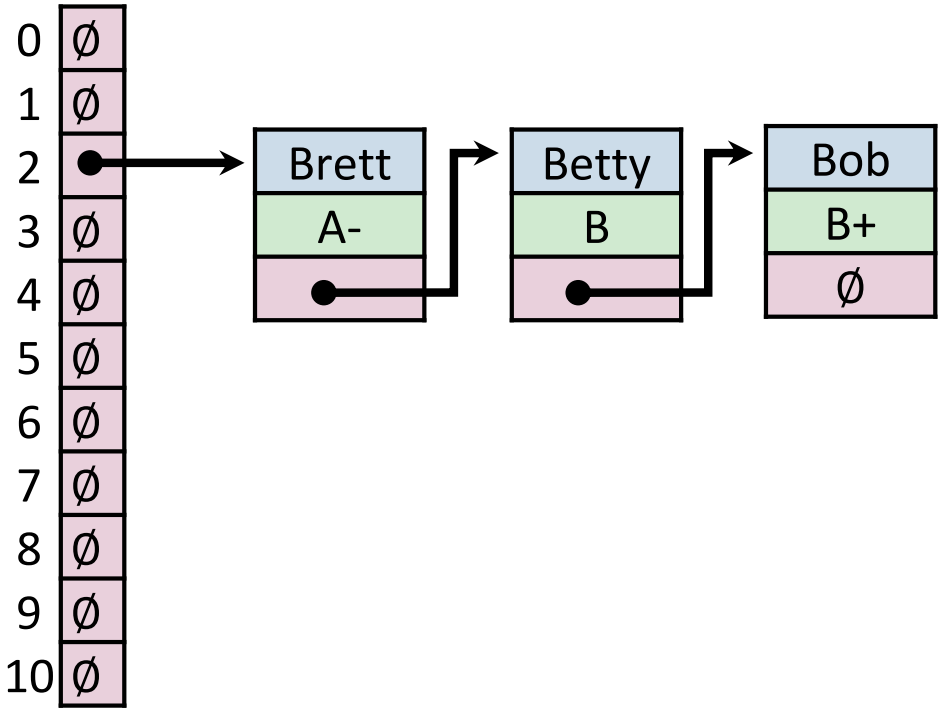
Key	Hash
Sue	7



Remove (Separate Chaining)

`_remove("Betty")`

Key	Hash
Betty	2



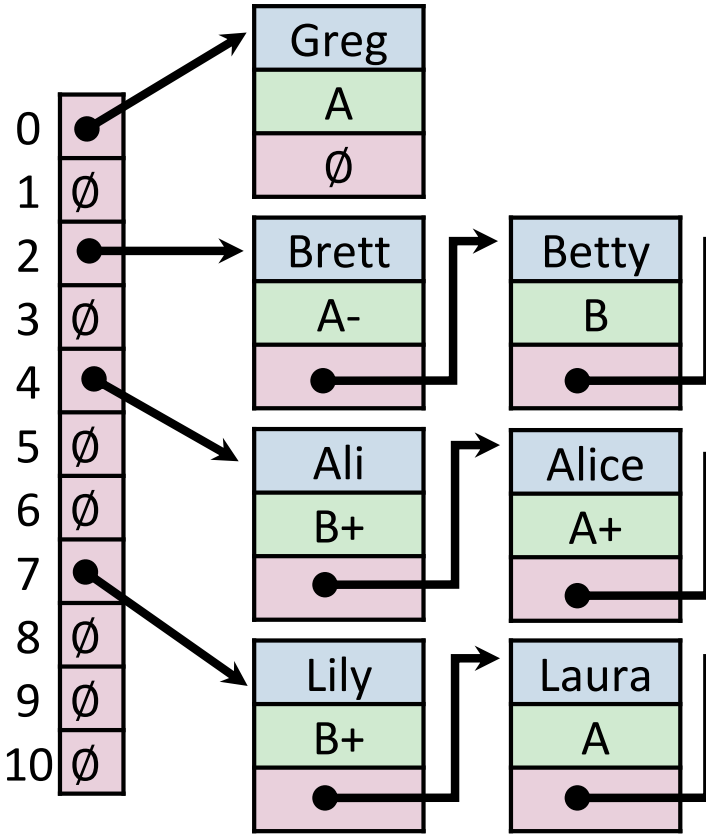
Hash Table (Separate Chaining)

For hash table of size *m* and *n* elements:

Find runs in: _____

Insert runs in: _____

Remove runs in: _____



Hash Table

Worst-Case behavior is bad — but what about randomness?

1) **Fix *h***, our hash, and assume it is good for ***all keys***:

2) Create a ***universal hash function family:***

Simple Uniform Hashing Assumption

Given table of size m, a simple uniform hash, h, implies

$\forall k_1, k_2 \in U$ where $k_1 \neq k_2$, $Pr(h[k_1] = h[k_2]) = \frac{1}{m}$

Uniform:

Independent:

Separate Chaining Under SUHA

Given table of size m and n inserted objects

Claim: Under SUHA, expected length of chain is $\frac{n}{m}$

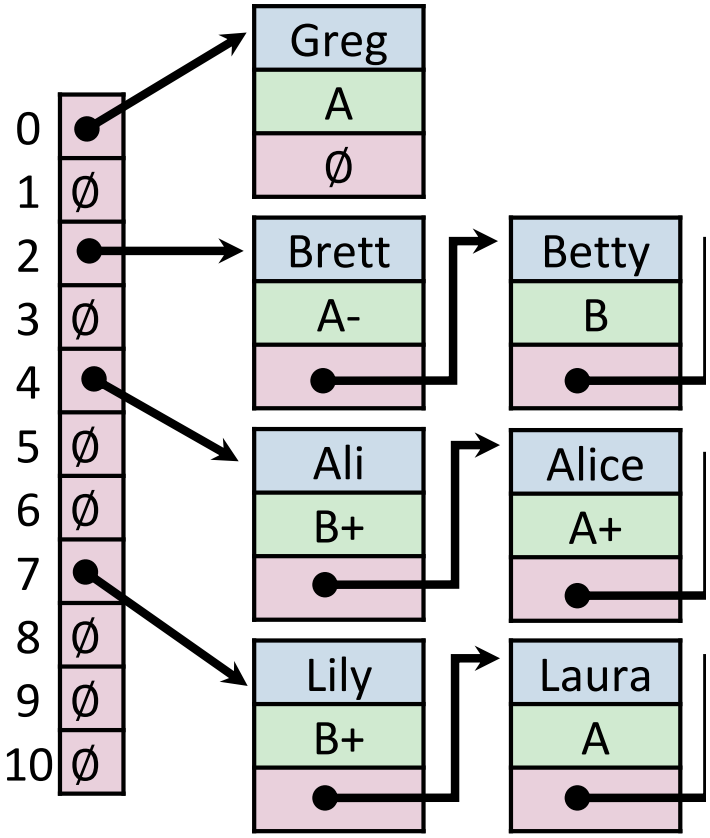
Hash Table (Separate Chaining w/ SUHA)

For hash table of size *m* and *n* elements:

Find runs in: _____

Insert runs in: _____

Remove runs in: _____



Separate Chaining Under SUHA

Pros:

Cons:

Next time: Closed Hashing

Closed Hashing: store *k,v* pairs in the hash table

S = { 1, 8 , 15}

h(k) = k % 7