Data Structures and Algorithms

Hashing

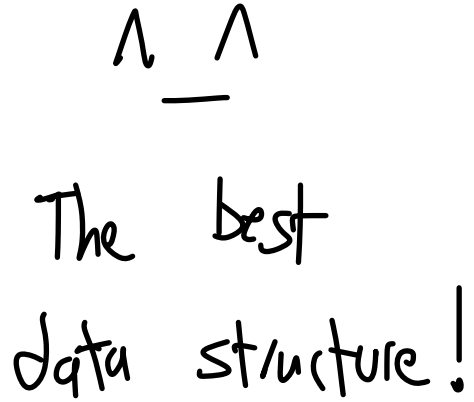
CS 225

Brad Solomon

April 17, 2026

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science



Randomization in Algorithms

weakest!

1. Assume input data is random to estimate average-case performance
   - ↳ BST
     - A random input dataset is probably O(log n) height bounded
     - data is random
2. Use randomness inside algorithm to estimate expected running time
   - ↳ Randomized Quicksort
     - ↳ selection of pivot is random
       - for any dataset
       - 100% accuracy
3. Use randomness inside algorithm to approximate solution in fixed time
   - ↳ Some loss of accuracy here
   - ↳ primality test
     - ↳ All prime #s = 1
     - ↳ Some not prime #s = 1
       - # of this is very small

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

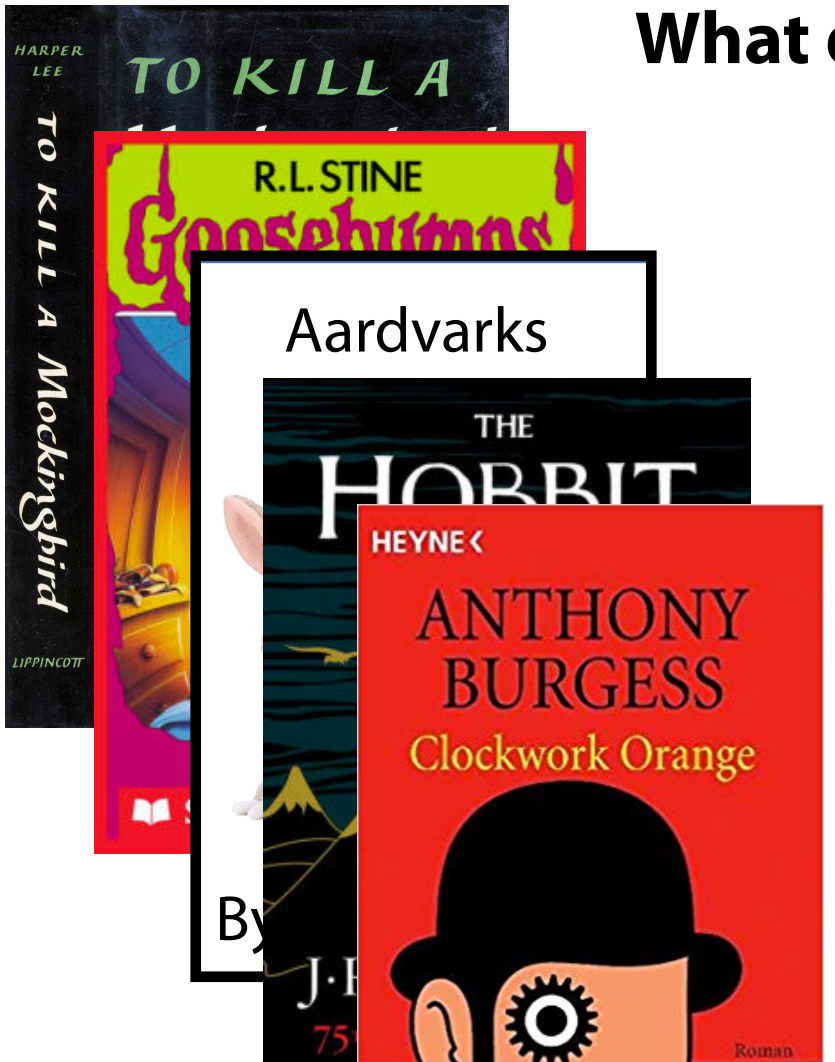
we want like hash tables yet

Data Structure Review

Key: Book title

Value: Book contents

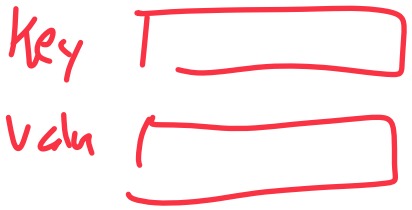
I have a collection of books and I want to store them in a dictionary!



What data structures can I use here?

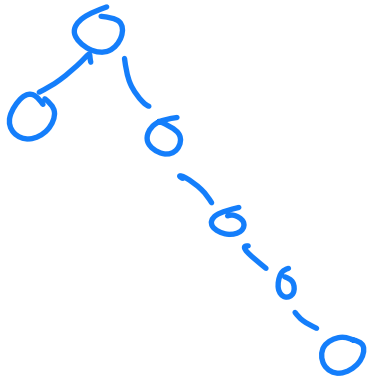
Parallel Arrays

- Insert O(1)*
- Find O(n)



BST — Each node K,V pair

- Insert O(1) ??
- Find O(n)

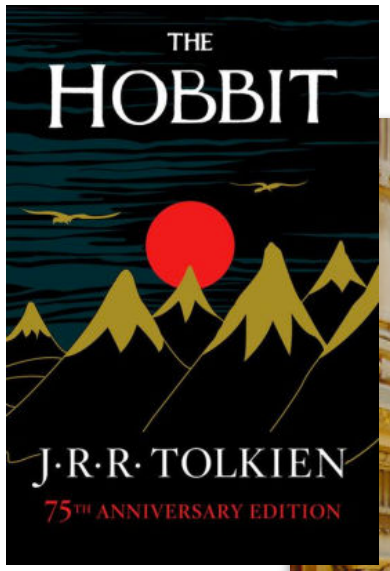


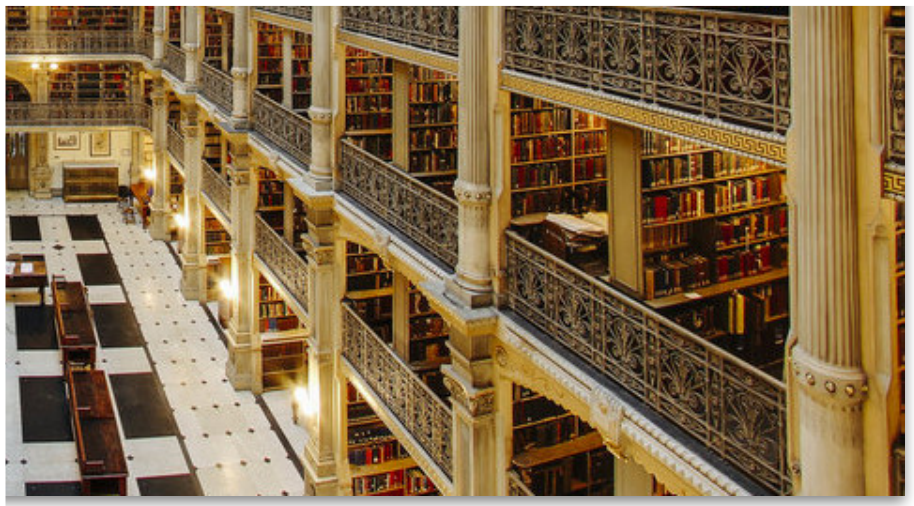
Balanced BST ← * Balanced log-log

- Insert
- Find

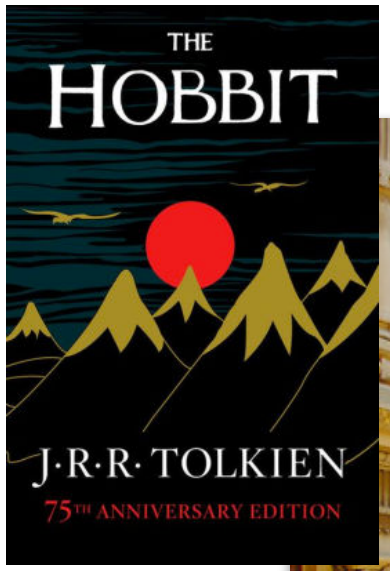
O(log n)
O(log n)

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





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

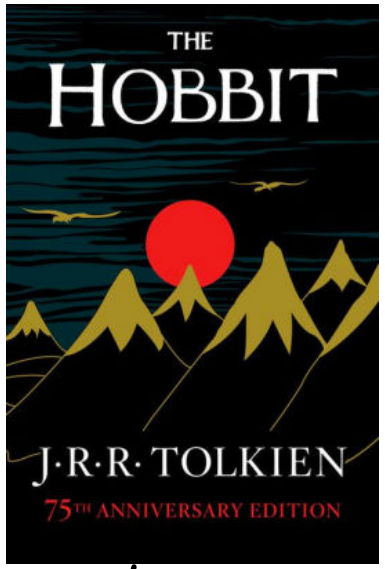


O(1)

Address

O(1)

A Hash Table based Dictionary



Key

O(1)

Black Box

A number Address

ISBN: 9780062265722

Call #: PR
6068.O93
H35 1937

ISBN: 9780062265722

Call #: PR
6068.O93
H35 1937

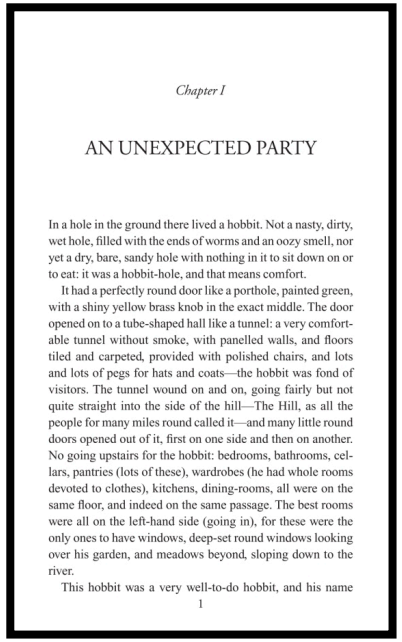
Address

O(1)

find value

O(1)

return value



Chapter I

AN UNEXPECTED PARTY

In a hole in the ground there lived a hobbit. Not a nasty, dirty, wet hole, filled with the ends of worms and an oozy smell, nor yet a dry, bare, sandy hole with nothing in it to sit down on or to eat: it was a hobbit-hole, and that means comfort.

It had a perfectly round door like a porthole, painted green, with a shiny yellow brass knob in the exact middle. The door opened on to a tube-shaped hall like a tunnel: a very comfortable tunnel without smoke, with panelled walls, and floors tiled and carpeted, provided with polished chairs, and lots and lots of pegs for hats and coats—the hobbit was fond of visitors. The tunnel wound on and on, going fairly but not quite straight into the side of the hill—The Hill, as all the people for many miles round called it—and many little round doors opened out of it, first on one side and then on another. No going upstairs for the hobbit: bedrooms, bathrooms, cellars, pantries (lots of these), wardrobes (he had whole rooms devoted to clothes), kitchens, dining-rooms, all were on the same floor, and indeed on the same passage. The best rooms were all on the left-hand side (going in), for these were the only ones to have windows, deep-set round windows looking over his garden, and meadows beyond, sloping down to the river.

This hobbit was a very well-to-do hobbit, and his name

1

Randomized Data Structures

Sometimes a data structure can be **too ordered / too structured**

BBST can find in $O(\log n)$

↳ As tradeoff it costs $O(\log n)$ to insert

Randomized data structures rely on **expected** performance

I can place in expectation $O(1)^*$

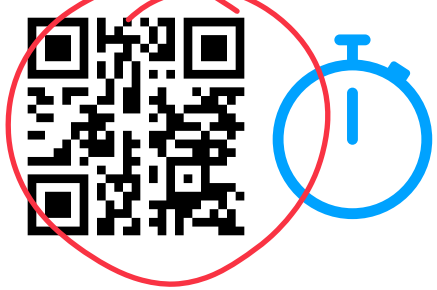
Str#1 of hash table!

Randomized data structures 'cheat' tradeoffs!

↳ Big O / worst case may be bad

but w/ randomness

A Hash Table based Dictionary



User Code (is a map):

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

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

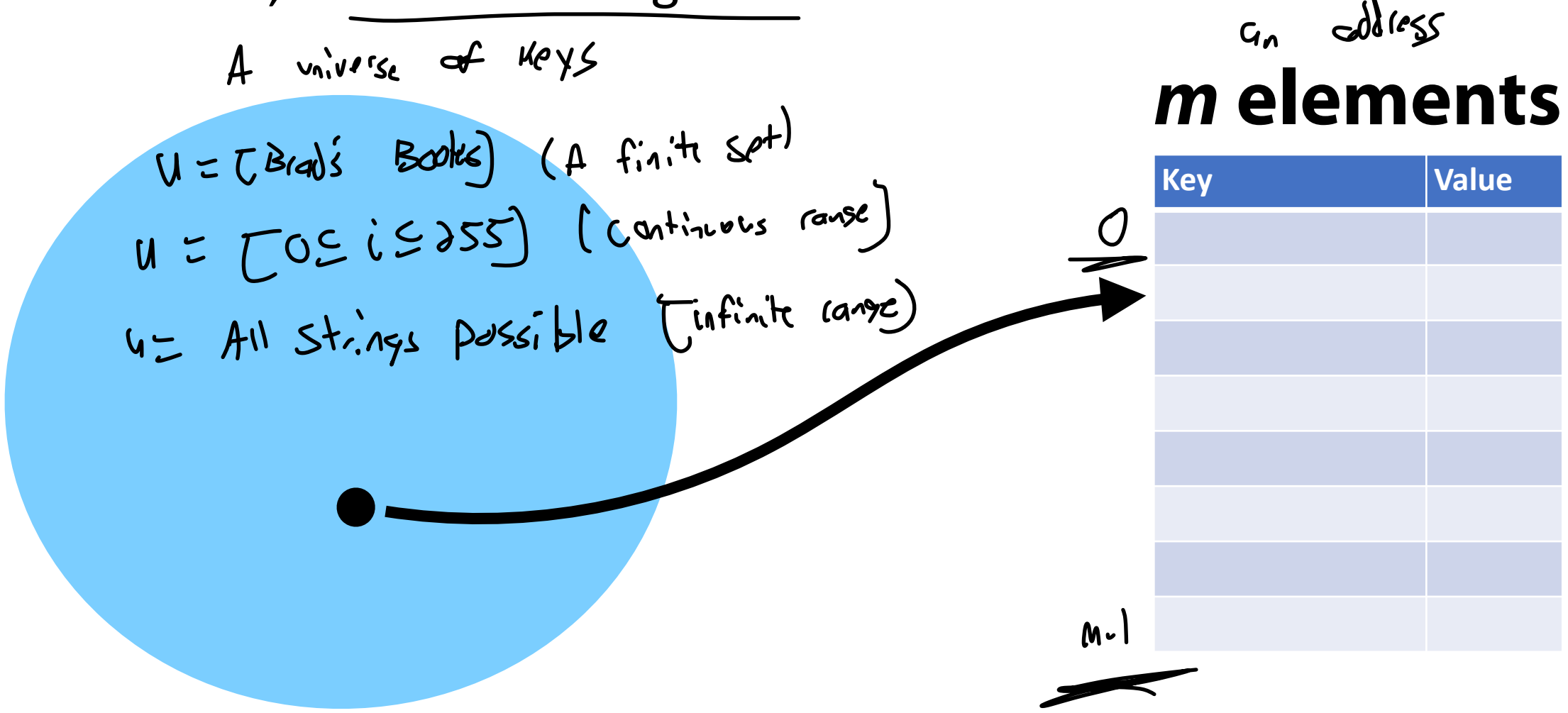
1. A hash function $h(k)$: $K \rightarrow int$ (address / hash value)

2. A data storage structure — An array (>0%)

3. ?? This is some way to handle "chaos"

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:

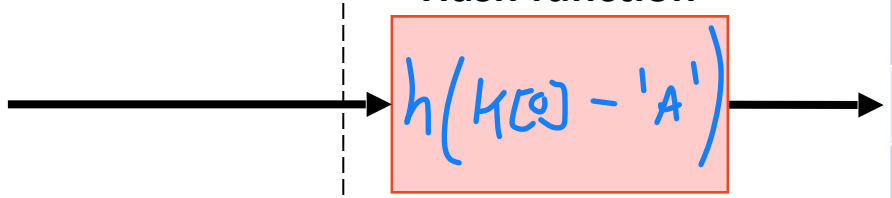
- **Deterministic**: $\forall k_1, k_2$ if $(k_1 == k_2)$ then $h(k_1) = h(k_2)$
  - ↳ IF you give the same key twice, get same address/ hash value out!
- **Efficient**: $O(1)$
- **Defined for a certain size table:** size m table → $[0, m-1]$ entries
  - ↑ I choose m

Hash Function

(key) (value)

Last Name, class taught

(Angrave, CS 241)
(Beckman, CS 421)
(Challon, CS 125)
(Davis, CS 101)
(Evans, CS 225)
(Fagen-Ulmschneider, CS 107)
(Gunter, CS 422)
(Herman, CS 233)



Key	Value

Hash Function

Problem #1: Collisions

1) What is a collision?

Both Angrave Alawini want to be @ 0!

(Angrave, CS 241) Alawini 411
(Beckman, CS 421)
(Challon, CS 125)
(Davis, CS 101)
(Evans, CS 225)
(Fagen-Ulmschneider, CS 107)
(Gunter, CS 422)
(Herman, CS 233)

Hash function

(key[0] - 'A')

	Key	Value
0	Angrave	241
1	Beckman	421
2	Challon	125
	Davis	101
	Evans	225
	Fagen-U	107
	Gunter	422
	Herman	233

Soloman 325

Doesn't fit in array!

General Hash Function

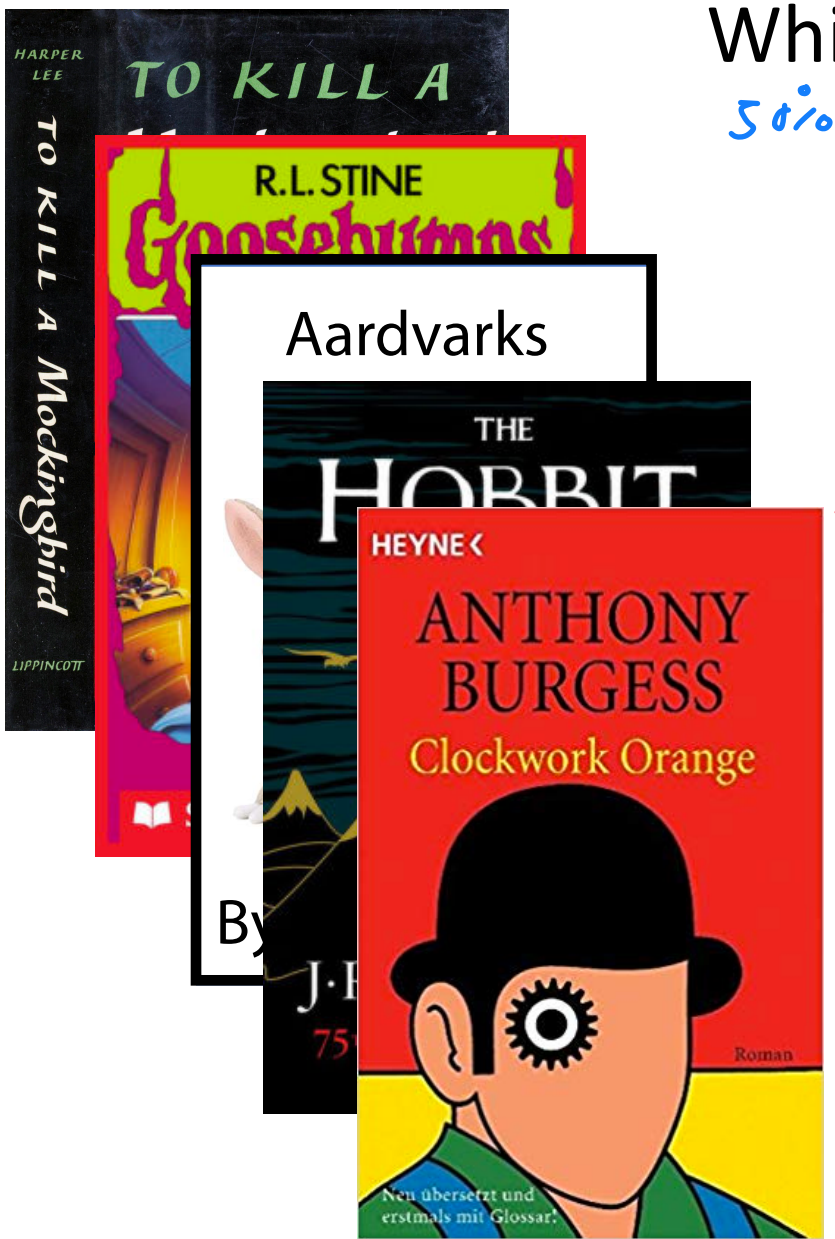
An $O(1)$ deterministic operation that maps all keys in a universe U to a defined range of integers $[0, \ldots, m-1]$

- A **hash**:
- A **compression**:

Choosing a good hash function is tricky…

- Don't create your own (yet*)

Hash Function



Which of the following are valid hashes?

50%

$$h_1(k) = \left(k.firstName[0] + k.lastName[0]\right) \% m$$

First letter first & last name

↳ Deterministic? Yes! ↳ O(1)? Yes! ↳ Any name? Yes!

A bad function → many possible collisions

35%

$$h_2(k) = \left(rand() * k.numPages\right) \% m$$

Generate random # * num pages

↳ Deterministic? No

If we generate only one # then it is fine!

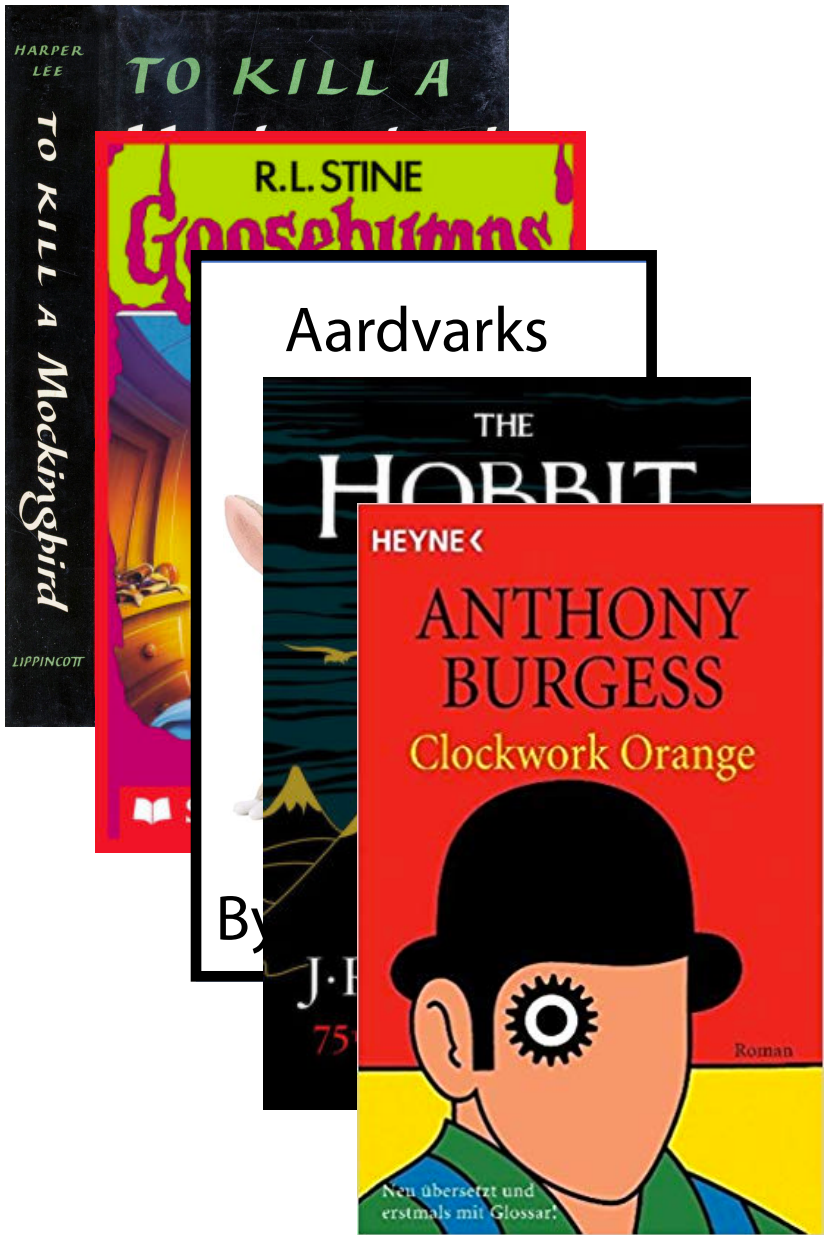
↳ (=2

$$h_3(k) = (\text{return Count++;}) \% m$$

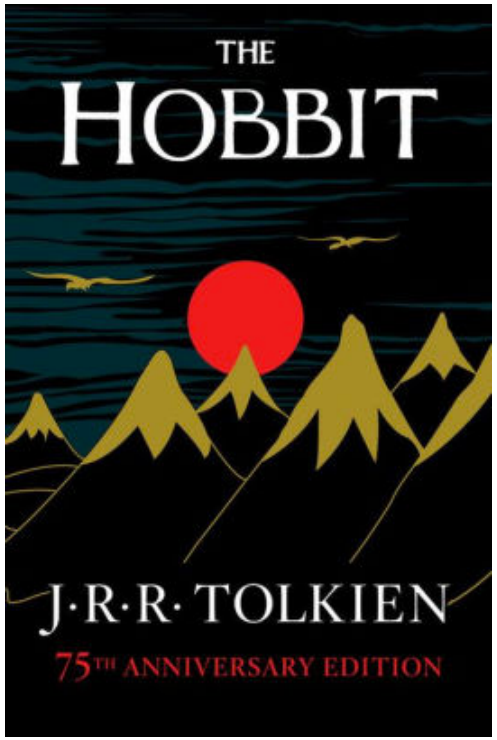
Index position in order seen items [Seeded random hash]

↳ Same book diff count value each time

Hash Function



A Hash Table based Dictionary

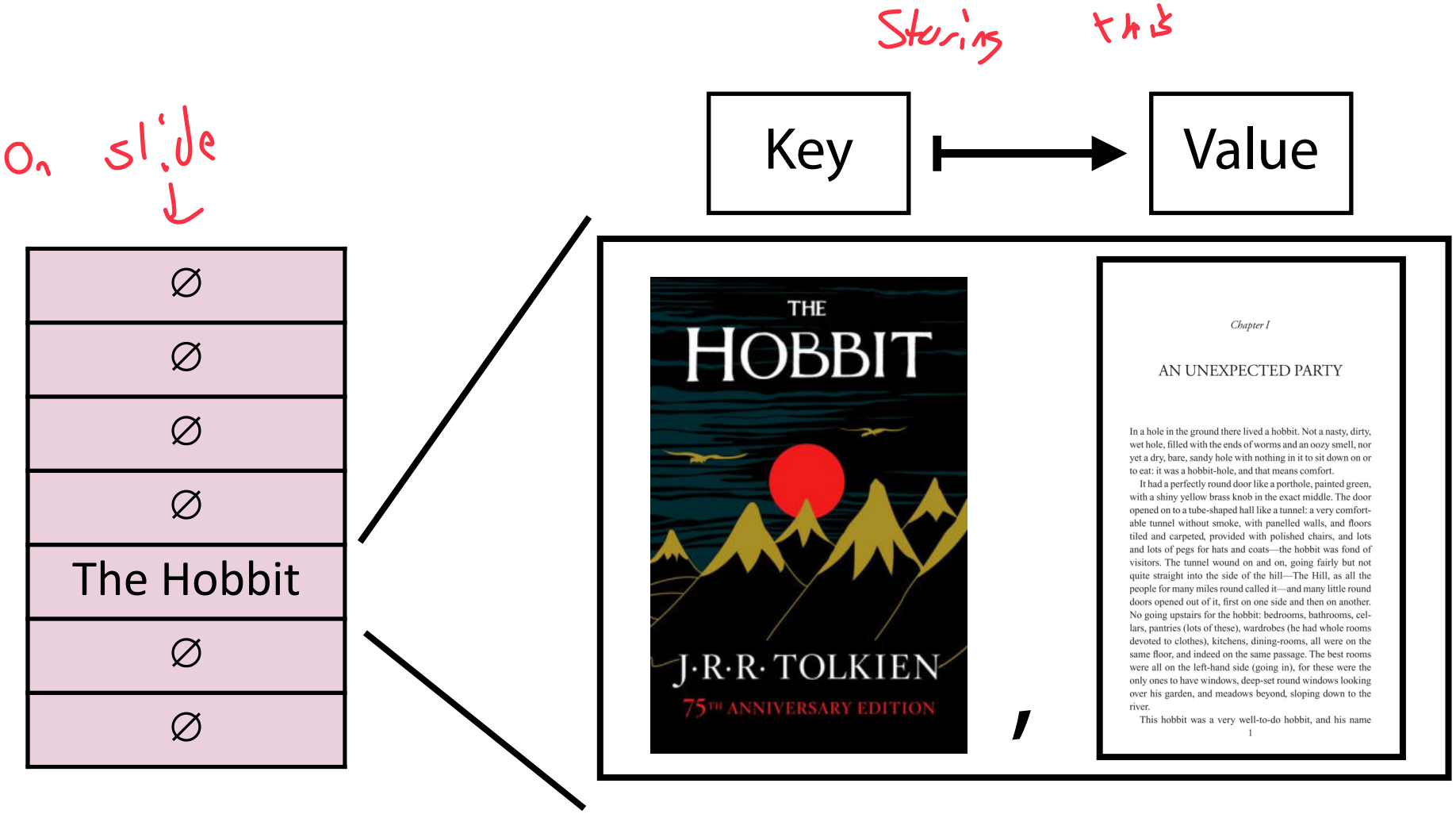


'J' + 'T' = 30

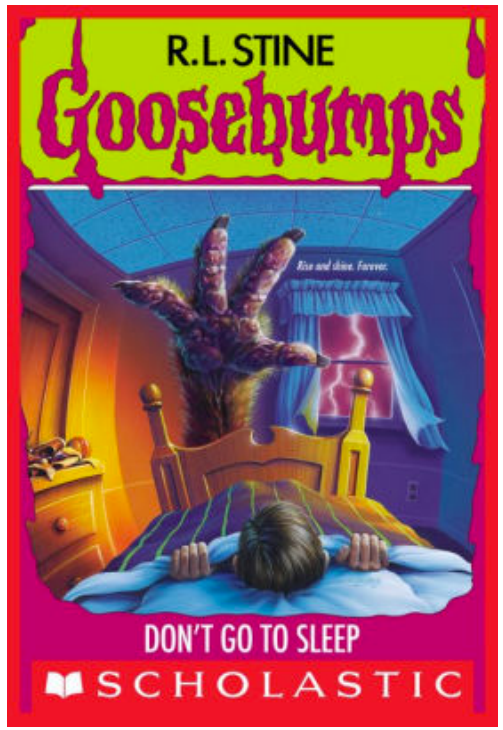
Author Name Hash Function

...	...
27	∅
28	∅
29	∅
30	The Hobbit
31	∅
...	...

A Hash Table based Dictionary



A Hash Table based Dictionary

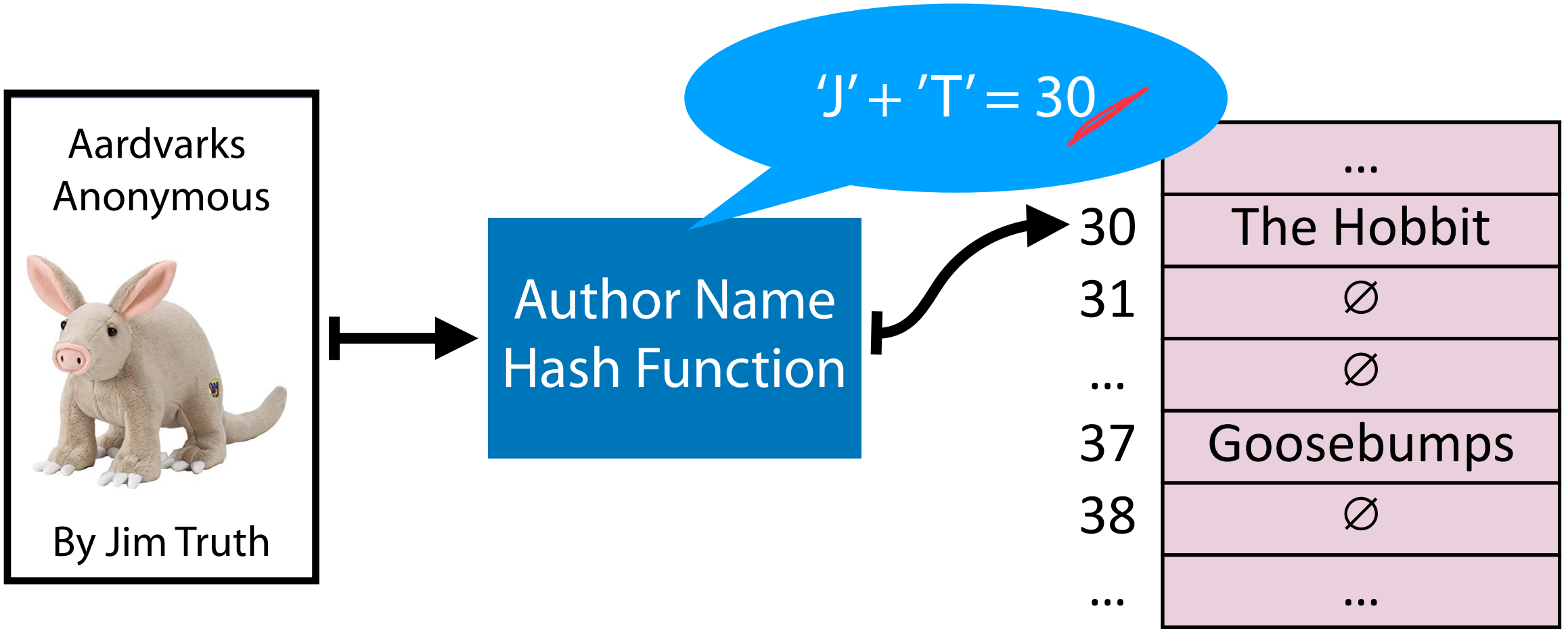


'R' + 'S' = 37

Author Name Hash Function

…	
30	The Hobbit
31	∅
…	∅
37	Goosebumps
38	∅
…	…

A Hash Table based Dictionary

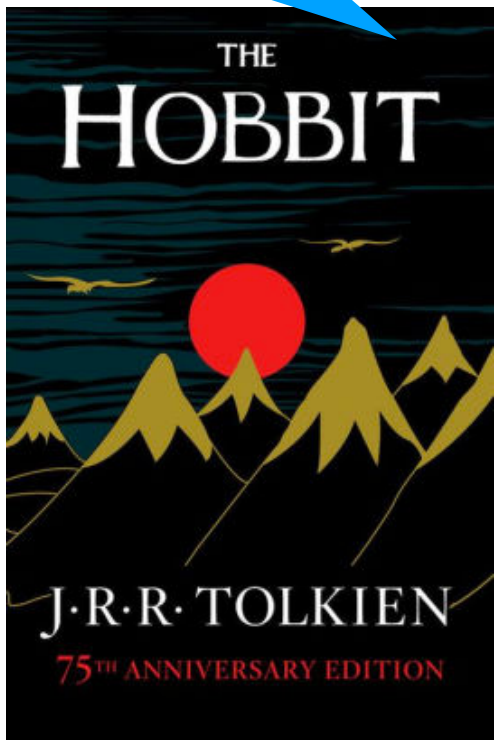


Hash Collision

How to handle this?

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

J.R.R Tolkien = 30!



Jim Truth = 30!

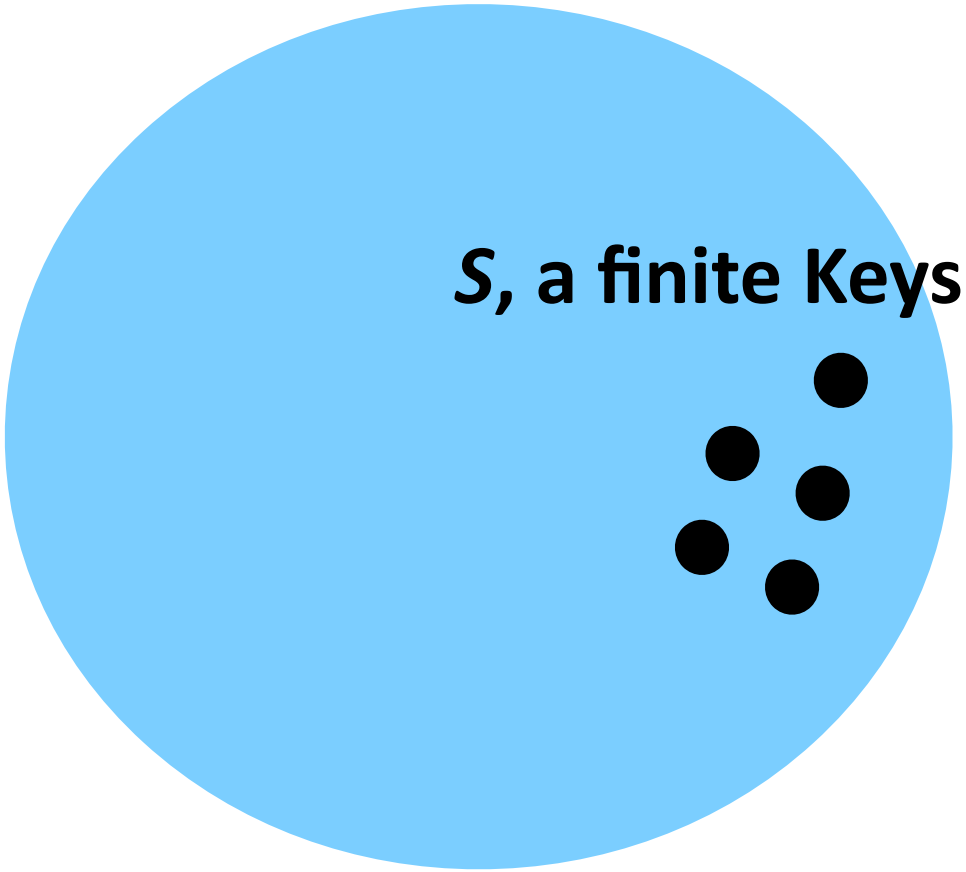
Aardvarks Anonymous

By Jim Truth

…	…
30	???
31	∅
…	∅
37	Goosebumps
38	∅
…	…

Perfect Hashing

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



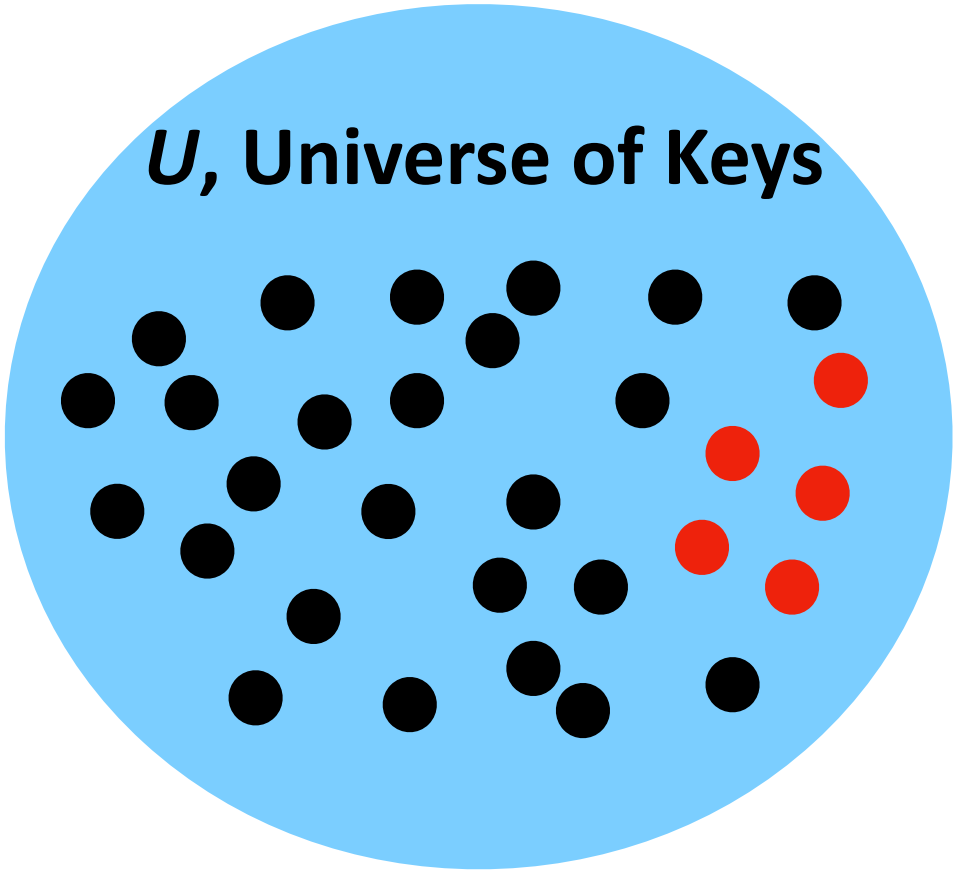
Size of array

m elements

Key	Value

General Purpose Hashing

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

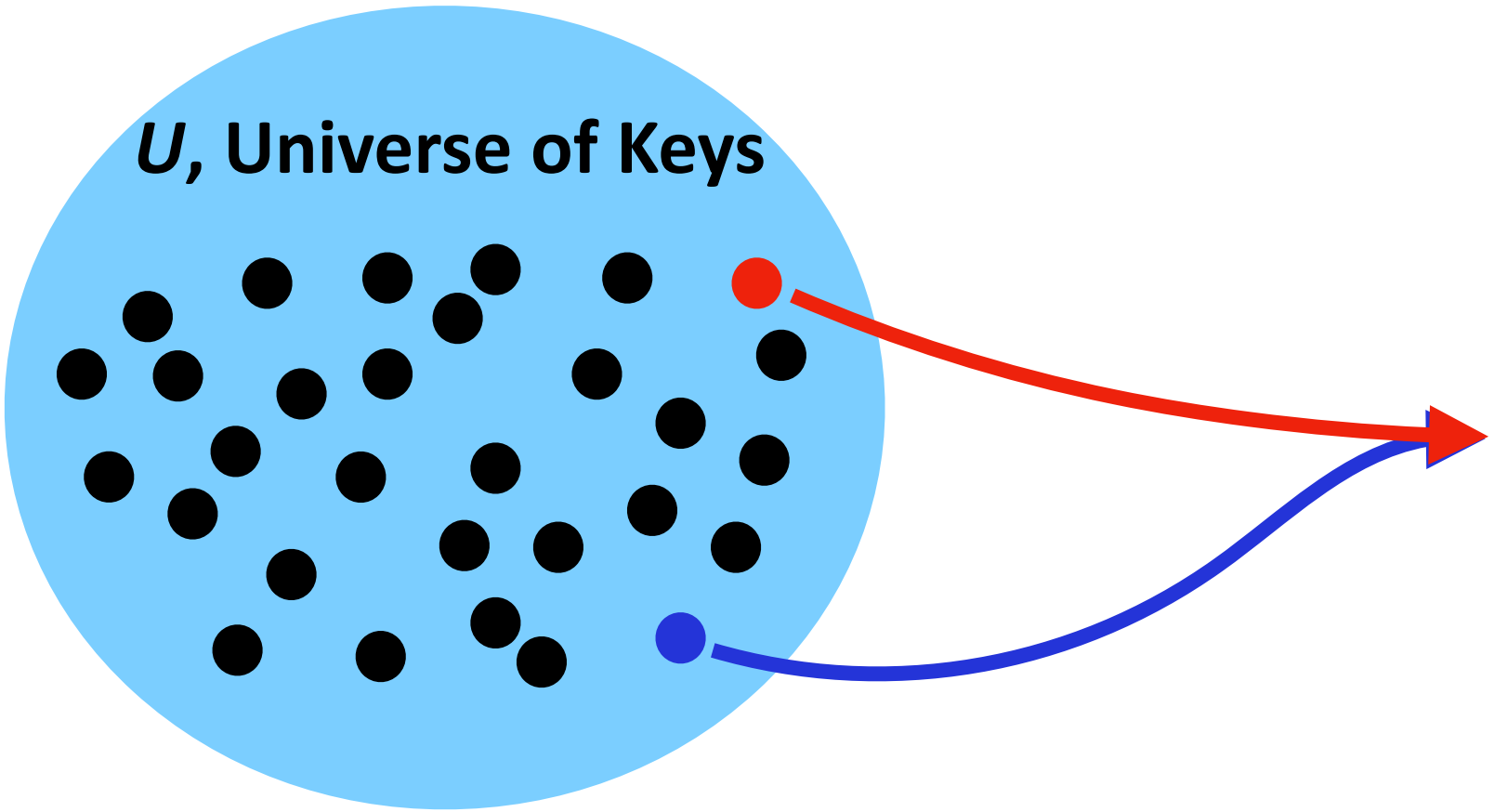


***m* elements**

Key	Value

General Purpose Hashing

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

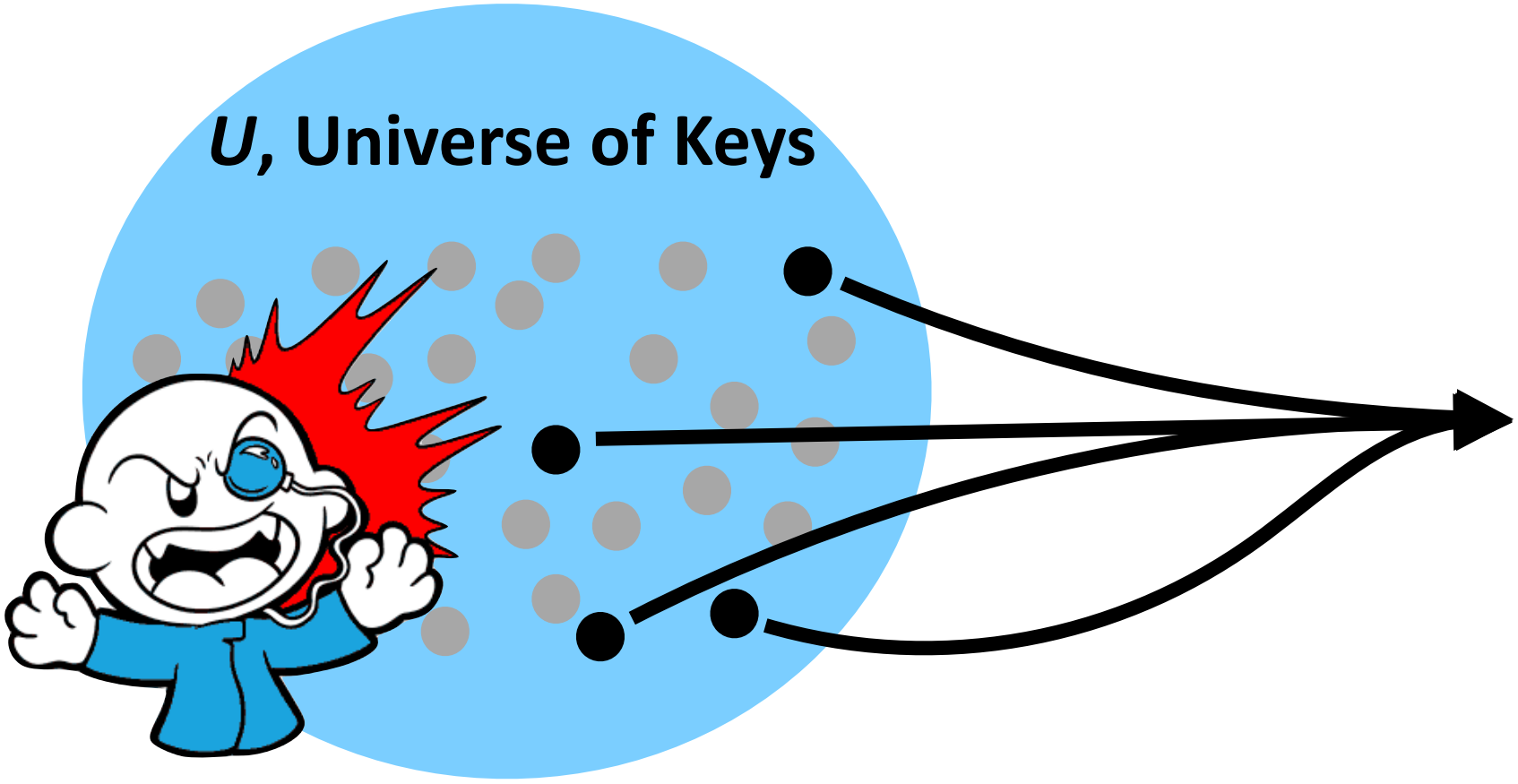


m elements

Key	Value

General Purpose Hashing

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



***m* elements**

Key	Value

Image by Matthew Loffhagen

A Hash Table based Dictionary

User Code (is a map):

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

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

1. A hash function

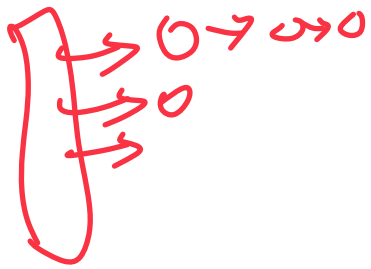
2. A data storage structure

3. A method of addressing *hash collisions*

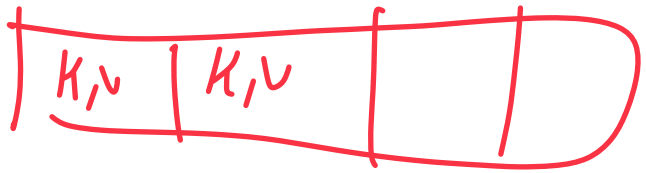
Open vs Closed Hashing

Addressing hash collisions depends on your storage structure.

- **Open Hashing:** store K,V pairs externally to hash table



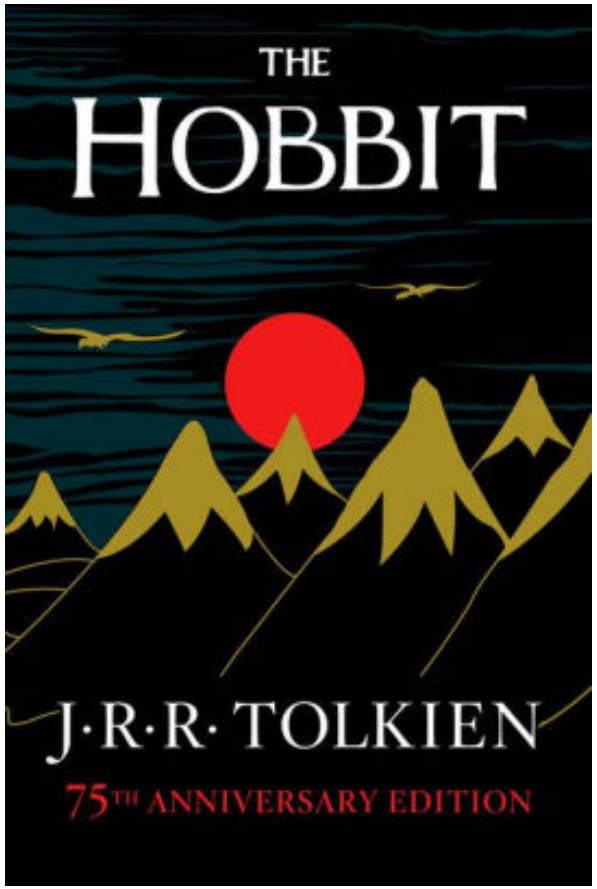
- **Closed Hashing:** internally

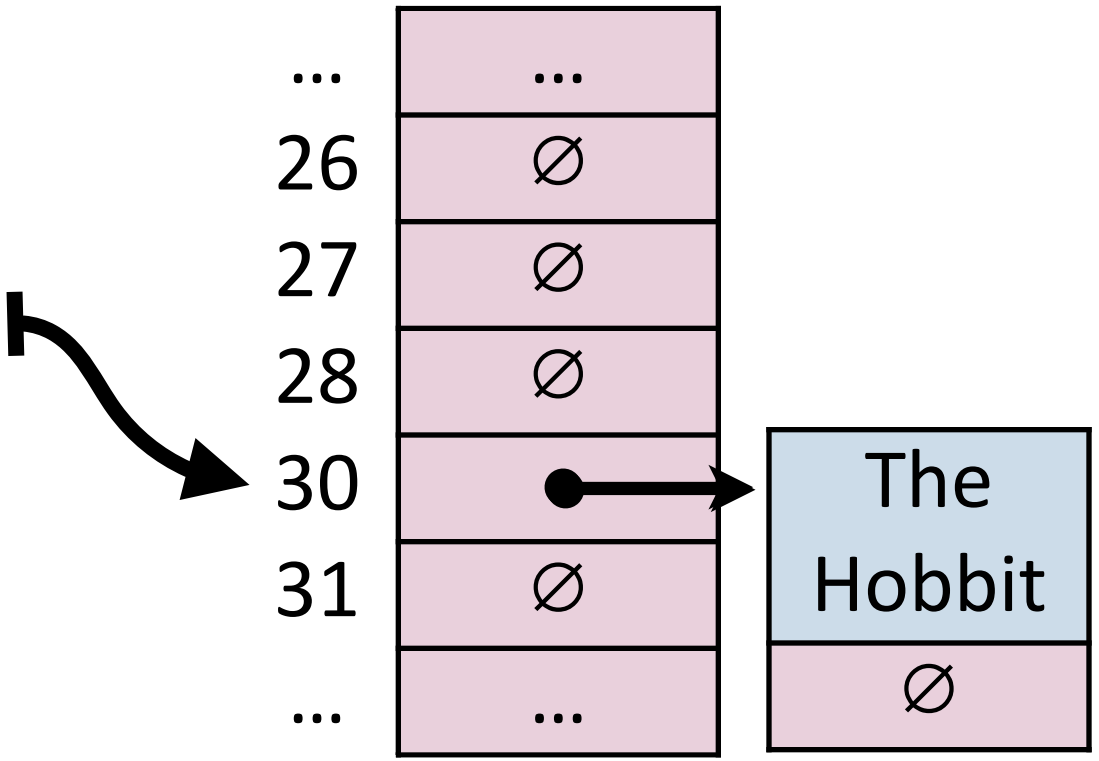


Open Hashing

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

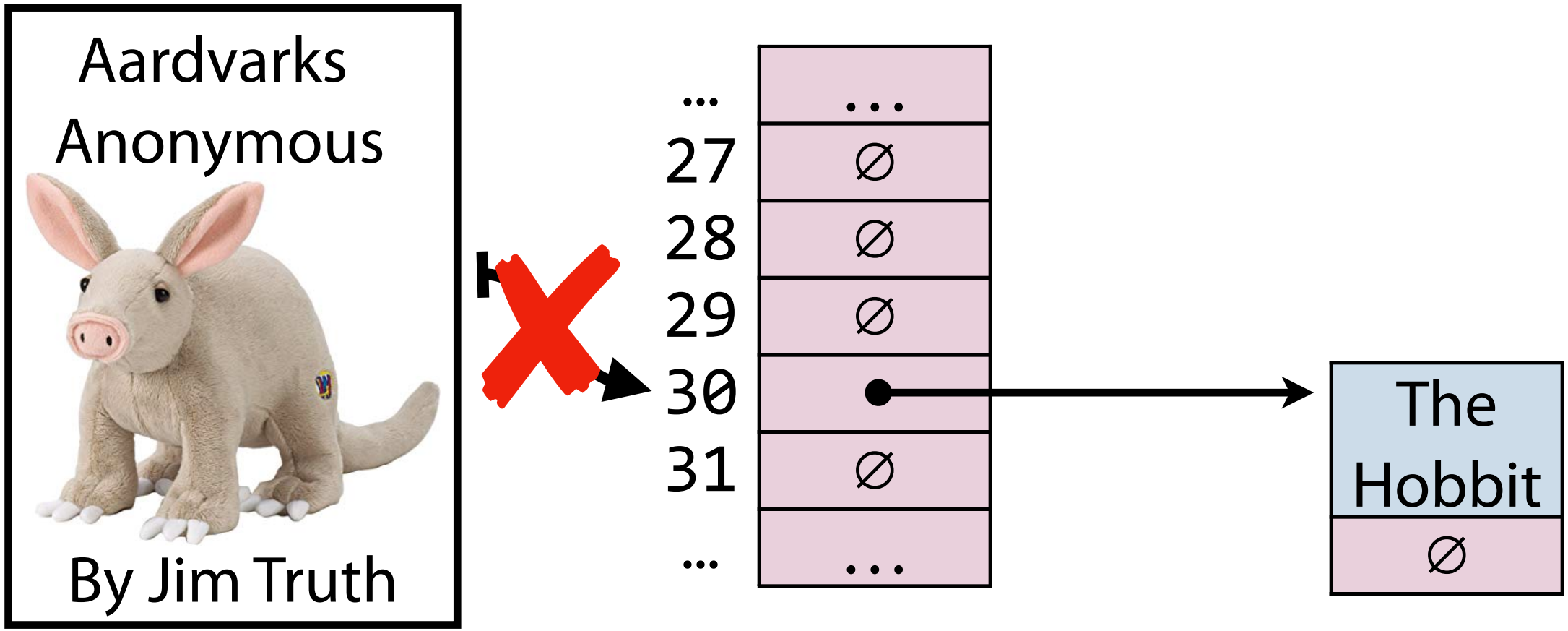
↳ a list of k,v pairs!





Hash Collisions (Open Hashing)

A ***hash collision*** in an open hashing scheme can be resolved by adding to linked list. This is called ***separate chaining***.



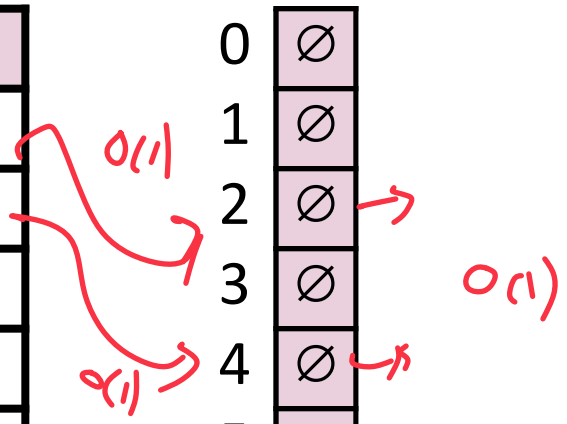
Insertion (Separate Chaining)

```
_insert("Bob")

_insert("Anna")
```

O(1)

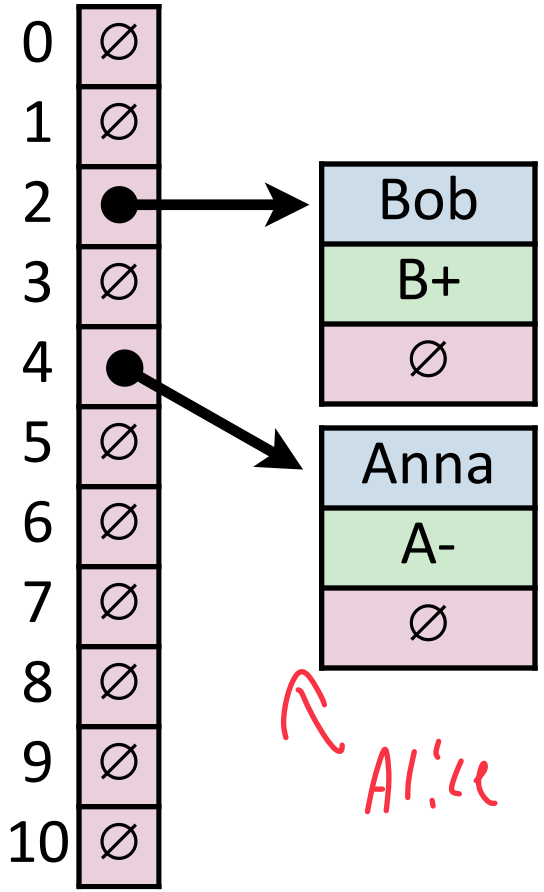
Key	Value	Hash
Bob	**B+**	**2**
Anna	**A-**	**4**
Alice	A+	4
Betty	B	2
Brett	A-	2
Greg	A	0
Sue	B	7
Ali	B+	4
Laura	A	7
Lily	B+	7



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

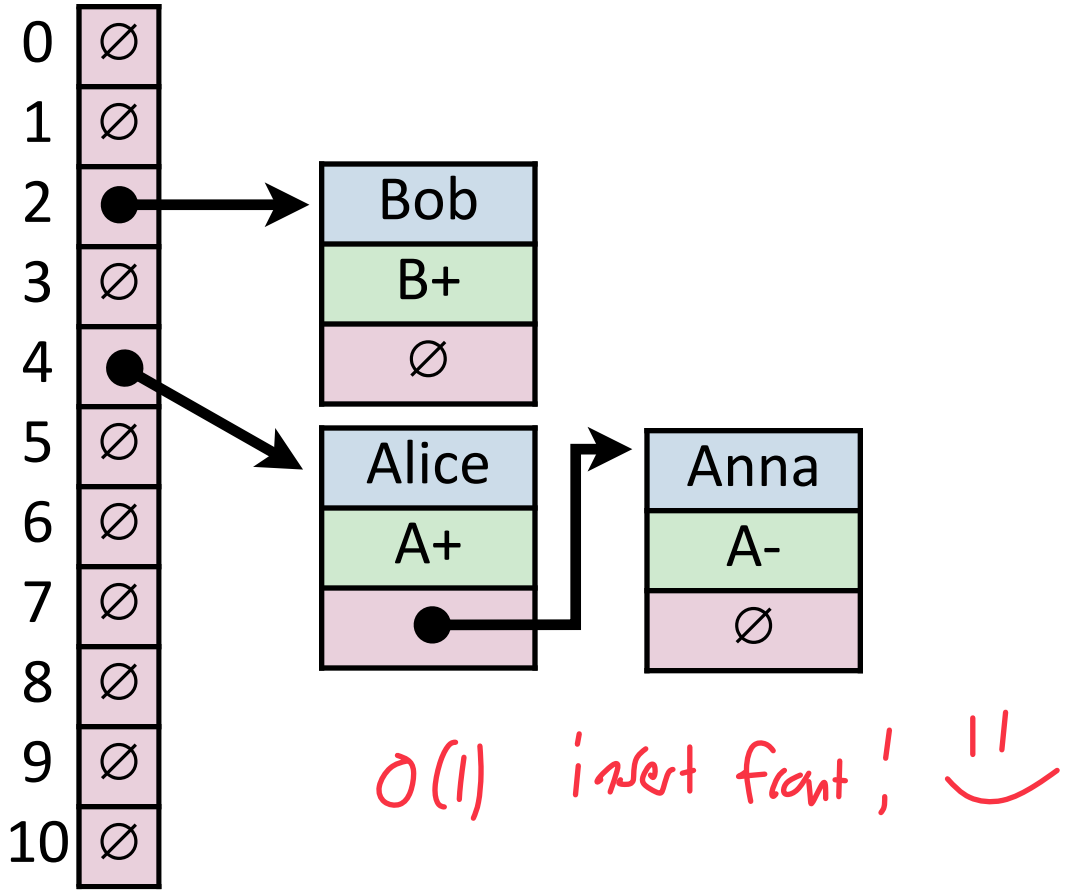
Where does Alice end up in the hash table?

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)

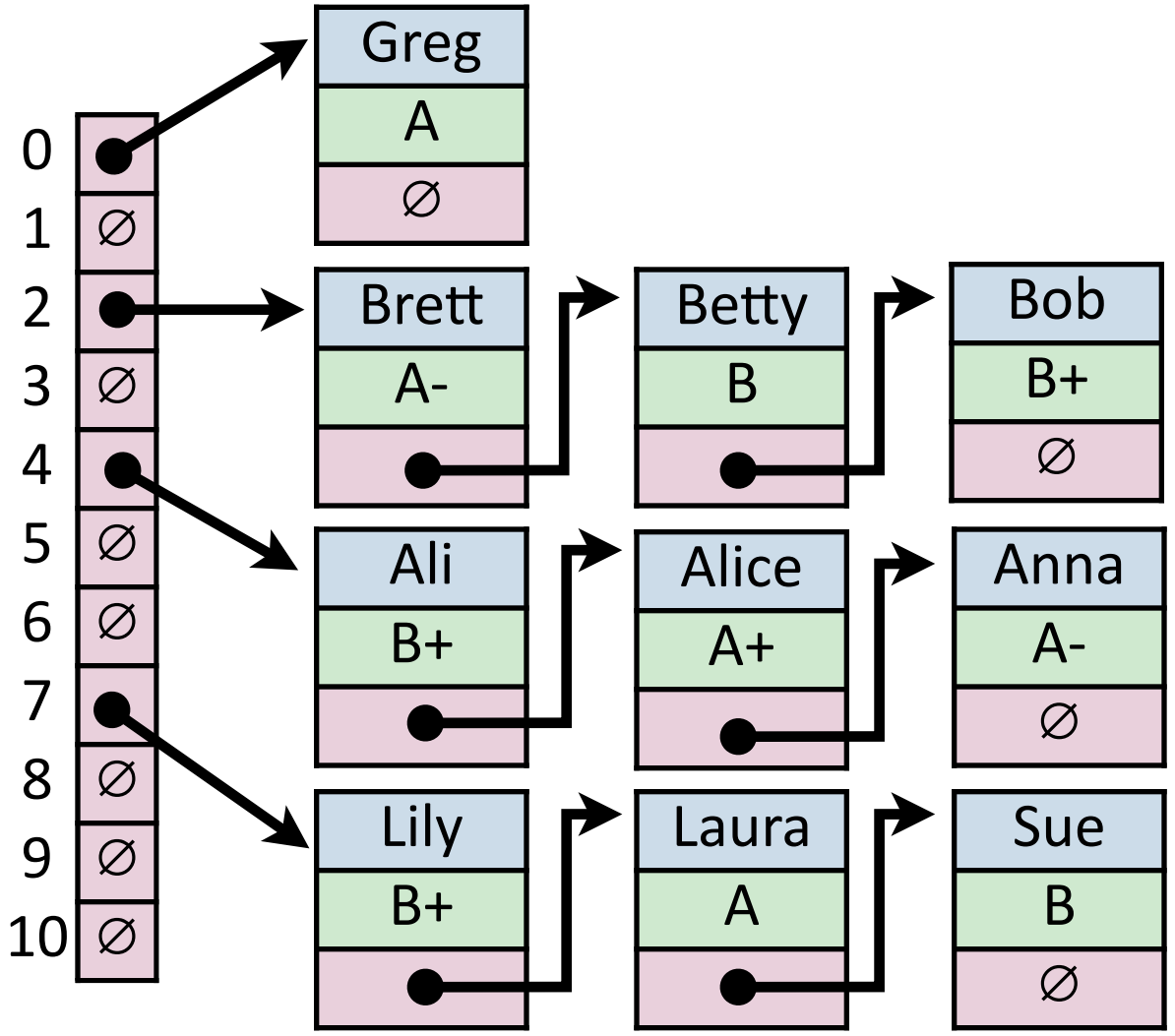
Key	Value	Hash
Bob	B+	2
Anna	A-	4
Alice	A+	4
Betty	**B**	**2**
Brett	A-	2
Greg	A	0
Sue	B	7
Ali	B+	4
Laura	A	7
Lily	B+	7



O(1) insert front! !!

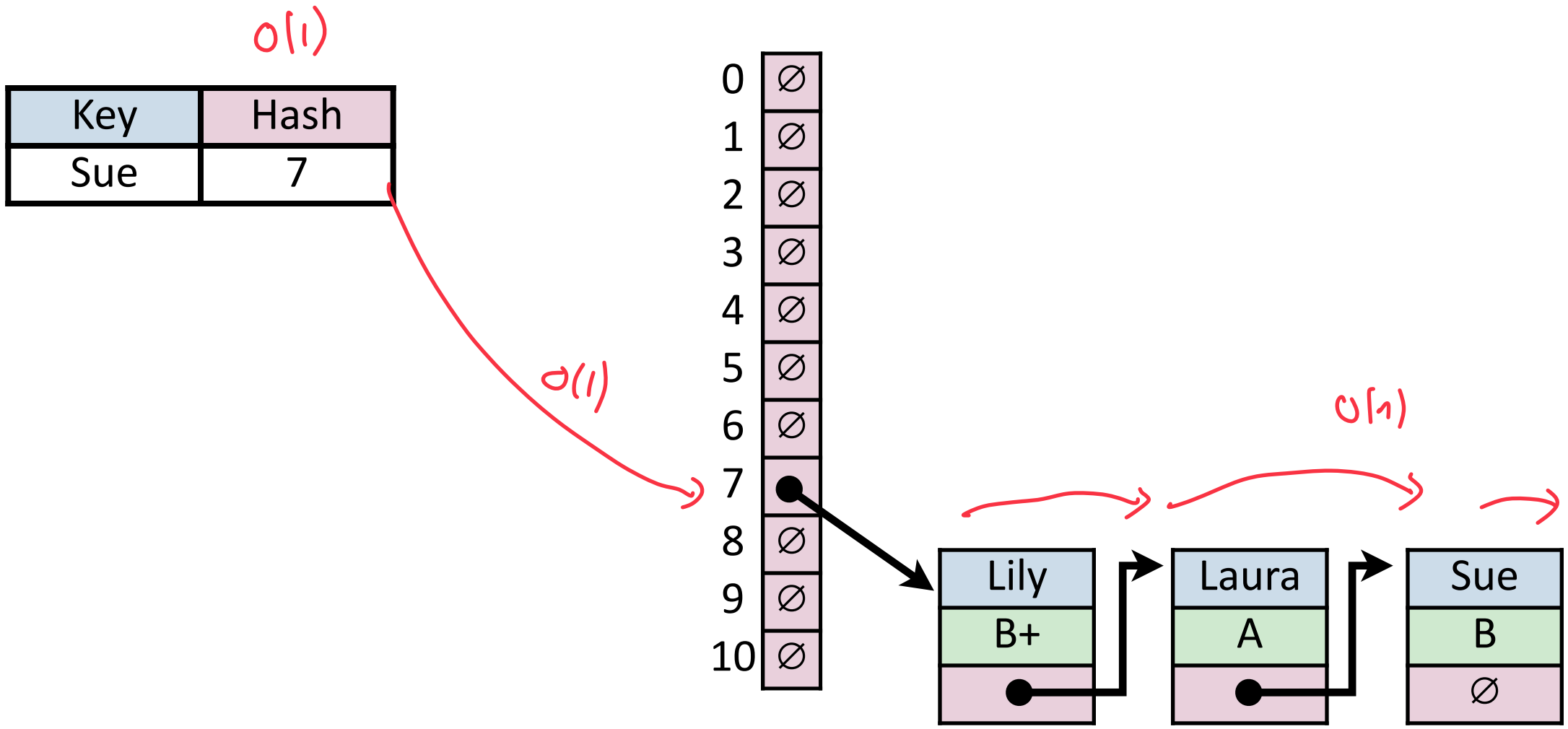
Insertion (Separate Chaining)

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



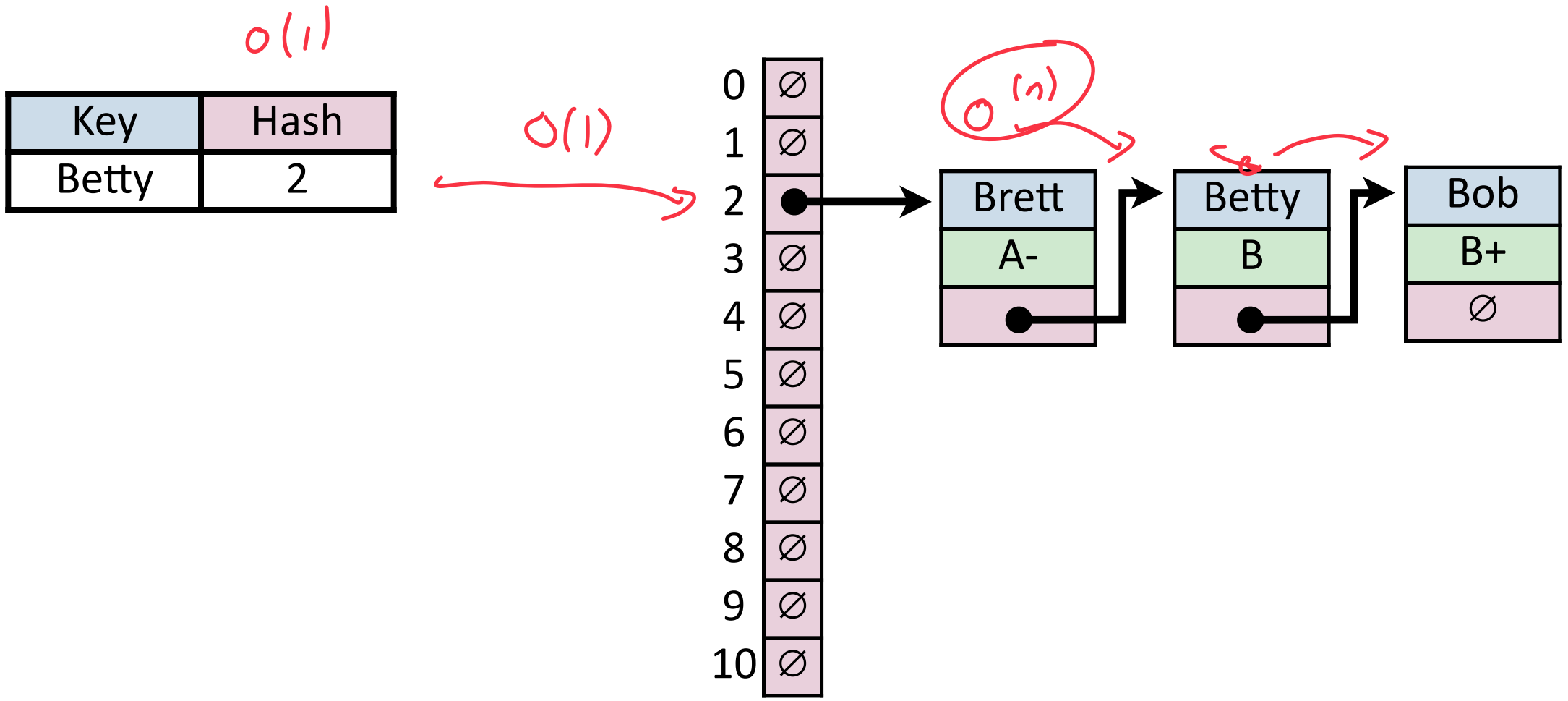
Find (Separate Chaining)

```
_find("Sue")
```



Remove (Separate Chaining)

`_remove("Betty")`



Hash Table (Separate Chaining)

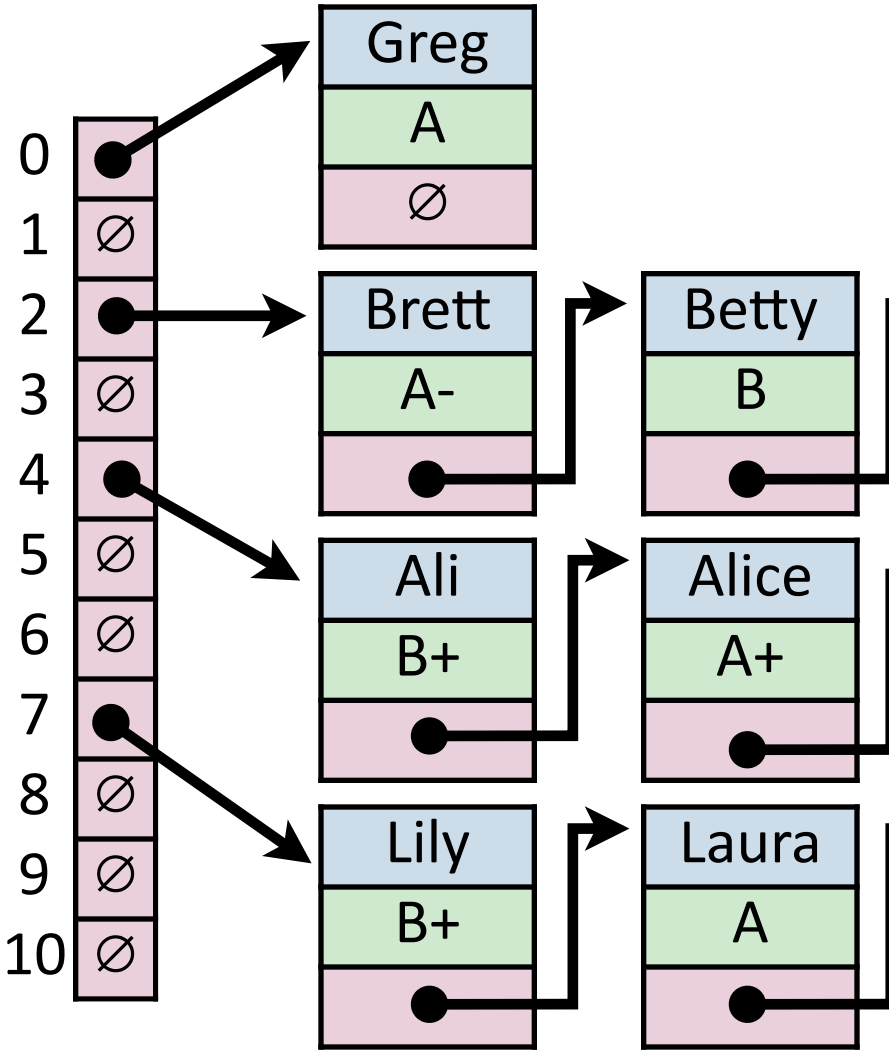
For hash table of size *m* and *n* elements:

Find runs in: O(n) (by key)

Insert runs in: O(1)

Remove runs in: O(n) (by key)

How can randomness save us?



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

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

Uniform:

Independent:

Separate Chaining Under SUHA

Table Size: m

Num objects: n

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

$\alpha_j =$ expected # of items hashing to position j

$$\alpha_j = \sum_i H_{i,j}$$

$$H_{i,j} = \begin{cases} 1 \text{ if item i hashes to j} \\ 0 \text{ otherwise} \end{cases}$$

Separate Chaining Under SUHA

Table Size: m

Num objects: n

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

$\alpha_j =$ expected # of items hashing to position j

$$\alpha_j = \sum_i H_{i,j}$$

$$H_{i,j} = \begin{cases} 1 \text{ if item i hashes to j} \\ 0 \text{ otherwise} \end{cases}$$

$$E[\alpha_j] = E\left[\sum_i H_{i,j}\right]$$

Separate Chaining Under SUHA

Table Size: m

Num objects: n

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

$\alpha_j =$ expected # of items hashing to position j

$$\alpha_j = \sum_i H_{i,j}$$

$$H_{i,j} = \begin{cases} 1 \text{ if item i hashes to j} \\ 0 \text{ otherwise} \end{cases}$$

$$E[\alpha_j] = E[\sum_i H_{i,j}]$$

$$E[\alpha_j] = \sum_i Pr(H_{i,j} = 1) * 1 + Pr(H_{i,j} = 0) * 0$$

Separate Chaining Under SUHA

Table Size: m

Num objects: n

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

$\alpha_j =$ expected # of items hashing to position j

$$\alpha_j = \sum_i H_{i,j}$$

$$H_{i,j} = \begin{cases} 1 \text{ if item i hashes to j} \\ 0 \text{ otherwise} \end{cases}$$

$$E[\alpha_j] = E[\sum_i H_{i,j}]$$

$$E[\alpha_j] = \sum_i Pr(H_{i,j} = 1) * 1 + Pr(H_{i,j} = 0) * 0$$

$$E[\alpha_j] = n * Pr(H_{i,j} = 1)$$

Separate Chaining Under SUHA

Table Size: m

Num objects: n

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

$\alpha_j =$ expected # of items hashing to position j

$$\alpha_j = \sum_i H_{i,j}$$

$$H_{i,j} = \begin{cases} 1 \text{ if item i hashes to j} \\ 0 \text{ otherwise} \end{cases}$$

$$E[\alpha_j] = E[\sum_i H_{i,j}]$$

$$Pr[H_{i,j} = 1] = \frac{1}{m}$$

$$E[\alpha_j] = n * Pr(H_{i,j} = 1)$$

Separate Chaining Under SUHA

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

Table Size: m

Num objects: n

α_j = expected # of items hashing to position j

$$\alpha_j = \sum_i H_{i,j}$$

$$H_{i,j} = \begin{cases} 1 \text{ if item i hashes to j} \\ 0 \text{ otherwise} \end{cases}$$

$$E[\alpha_j] = E[\sum_i H_{i,j}]$$

$$Pr[H_{i,j} = 1] = \frac{1}{m}$$

$$E[\alpha_j] = n * Pr(H_{i,j} = 1)$$

$$\mathbf{E[\alpha_j] = \frac{n}{m}}$$

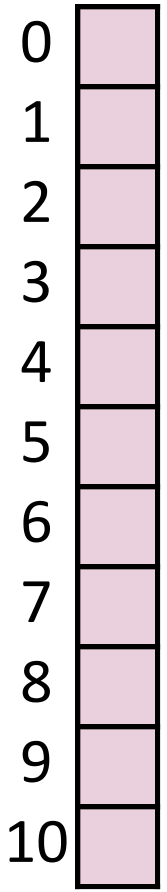
Separate Chaining Under SUHA

Under SUHA, a 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